Broadview
www.broadview.com.cn

安全技术大系

看雪软件安全
http://www.pediy.com

GOOD
畅销书升级版

加密与解密

第三版

段钢 编著

揭示软件加密与解密最核心
看雪安全技术团队全力支持

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

★专家讲坛 全面探讨

软件安全问题与解决之道

★技术剖析 深入浅出

分析加密与解密技术核心

★共同进步 循序渐进

迅速提升读者的专业水平

本书技术支持：

看雪软件安全网站提供本书的全面技术支持服务，阅读此书过程中，如有什么问题或学习心得，欢迎光临论坛与这些传说中的好手共同交流。

网上订购：www.dearbook.com.cn
第二书店 · 第一服务

策划编辑：郭 立
责任编辑：葛 娜
责任美编：谢丹丹

上架建议：安全技术>软件安全

ISBN 978-7-121-06644-3

定价：59.00元

安全技术大系

看雪软件安全
http://www.pediy.com

加密与解密

第三版

段钢 编著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书以加密与解密为切入点，讲述了软件安全领域许多基础知识和技能，如调试技能、逆向分析、加密保护、外壳开发、虚拟机设计等。读者在掌握本书的内容时，很容易在漏洞分析、安全编程、病毒分析、软件保护等领域扩展，这些知识点都是相互的，彼此联系。国内高校对软件安全这块领域教育重视程度还不够，许多方面还是空白，而近年来社会和企业对软件安全技术人才需求逐年上升。从就业角度来说，掌握这方面技术，可以提高自身的职场竞争能力；从个人成长角度来说，研究软件安全技术有助于掌握许多系统底层知识，是提升职业技能的重要途径。作为一名合格的程序员，除了掌握需求分析、设计模式等外，如能掌握一些系统底层知识，熟悉整个系统的底层结构，对自己的工作必将获益良多。

本书可以作为大中专学校或培训机构的软件安全辅助教材，是安全技术爱好者、调试人员、程序开发人员不可多得的一本好书。

图书在版编目（CIP）数据

加密与解密 / 段钢编著. —3 版. —北京：电子工业出版社，2008.7
（安全技术大系）
ISBN 978-7-121-06644-3

Ⅰ. 加… Ⅱ. 段… Ⅲ. 电子计算机－密码术 Ⅳ. TP309.7

中国版本图书馆 CIP 数据核字（2008）第 064470 号

策划编辑：郭 立
责任编辑：葛 娜
印 刷：北京智力达印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：880×1230 1/16 印张：35.5 字数：1018 千字
印 次：2008 年 7 月第 1 次印刷
印 数：6000 册 定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

作 者 简 介

本书由看雪软件安全网站（看雪学院）站长段钢主持编著。在本书的编写过程中，参与创作的每位作者倾力将各自擅长的专业技术毫无保留地奉献给广大读者，使得本书展现出了极具价值的丰富内容。如果读者在阅读本书后，能够感受到管窥技术奥秘带来的内心的喜悦，并愿意与大家分享这份感受，这是作者最大的愿望。

主 编：段 钢

编 委：（按章节顺序排列）

Blowfish，沈晓斌，丁益青，单海波，王勇，赵勇，唐植明，softworm，afanty，李江涛，林子深，印豪，冯典，罗翼，林小华，郭春杨

编委档案

Blowfish

看雪首席版主。经验丰富的大龄程序员。1992 年上大学始接触电脑，1997 年读研期间接触网络并自学加密与解密技术，一发不可收拾，其时常在教育网 BBS 灌水。喜多方涉猎，亦能抓住一点深入钻研，对逆向分析技术尤为痴迷。多年来常在看雪论坛灌水，见证了论坛的风风雨雨，也结识了一些不错的朋友。

参与章节：第 5 章　5.1 序列号保护方式

第 14 章　14.5 软件保护的若干忠告

沈晓斌

看雪核心专家团队成员。看雪论坛 ID 为 cnbragon，现攻读密码学专业硕士学位。最初的爱好是网络安全，进而研究软件的逆向工程，对密码学的兴趣由此而发。对密码学的各个方面都有所涉猎，尤其擅长密码学在软件保护中的应用研究。独立完成了一个加密算法库 CryptoFBC。译作有《程序员密码学》。

个人主页：www.cnbragon.cn

参与章节：第 6 章　加密算法

丁益青

看雪技术专家。看雪论坛 ID 为 cyclotron，复旦大学在读硕士研究生，复旦大学日月光华 BBS 黑客与系统安全版版主，致力于 Windows 环境下可执行文件的加密解密与逆向工程研究。主要作品有 EmbedPE、IDT Protector、PEunLOCK 等。

个人主页：cyclotron.yculblog.com

参与章节：第 8 章　8.3 伪编译

单海波

看雪核心专家团队成员。看雪论坛 ID 为 tankaiha，生于六朝古都南京，硕士研究生毕业，现任某研究

所工程师，工作之余好与计算机为伴。2002 年接触汇编并热衷于病毒技术学习，后偶遇看雪学院，遂终日游戏于程序加密与解密，不可自拔。2006 年与 kanxue 及坛中数位好友成立.net 安全小组 DST（Dotnet Reverse Team），共同探讨.net 平台下的软件安全技术。

个人主页：http://vxer.cn/blog

参与章节：第 9 章　.Net 平台加解密

王　勇

看雪技术专家。毕业于石油大学（华东）计算机科学与技术专业。擅长 C/C++、ASM 和驱动程序开发。对面向对象程序设计和 Windows 系统底层的研究有丰富的经验。很高兴这次能与各位高手一起合作，也希望能与编程爱好者及加密解密爱好者更多的交流。

主页：http://www.w-yong.com

参与章节：第 10 章　10.15 编写 PE 分析工具

赵　勇

看雪技术专家。来自江苏江阴，计算机业余爱好者，兴趣爱好广泛。

参与章节：第 13 章　13.6 附加数据

唐植明

看雪技术核心权威。看雪论坛 ID 为 DiKeN，2002 年毕业于兰州大学，计算机科学与技术专业。爱好逆向工程，iPB（inside Pandora's Box）组织创始人（在这儿更是要感谢组织的兄弟姐妹们，大家团结友好，互相学习，为 iPB 的成功做出了巨大努力），曾在 2002 年编写过《加密与解密实战攻略》算法部分。

参与章节：第 13 章　13.10 静态脱壳

softworm

看雪技术天才。70 后一代，非计算机专业的业余爱好者。1998 年开始接触逆向与破解，迄今已近 10 年，终于达到了“知道自己不知道”的境界。感兴趣的方向包括壳、虚拟机保护、病毒引擎、Rootkit。后两项还处于只知道名字的水平，愿与有共同爱好的朋友们一起学习。

E-mail：softworm2003@hotmail.com

参与章节：第 13 章　13.9.2 Thmedia 的 SDK 分析

afanty

看雪技术专家。多年专业研究软件加/解密技术。

参与章节：第 14 章　14.1 防范算法求逆

李江涛

看雪技术核心权威。看雪论坛 ID 为 ljtt，喜欢学习编程技术，常用编程语言为 VC/MASM。对 PB、VFP 的反编译有深入的研究，写过 DePB、FoxSpy 等程序。平时大多数时间都在电脑上耕作，最大的希望是能够领悟到编程的精髓，写一个自己比较满意的作品。

E-mail：shellfan@163.com

参与章节：第 14 章　14.2.2 SMC 技术实现

林子深

看雪技术导师。看雪论坛 ID 为 forgot，1989 年生，看雪论坛外壳开发小组组长。熟悉 Win32 平台和 80x86 汇编，擅长代码的逆向，对壳的研究比较多。

E-mail：forgot@live.com

参与章节：第 12 章　12.4.1 虚拟机介绍

第 14 章　14.2.4 简单的多态变形技术

第 15 章　反跟踪技术

印　豪

看雪资深技术权威。看雪论坛 ID 为 Hying，擅长加壳技术，拥有独立创作的加密利器。

E-mail：newhying001@163.com

参与章节：第 16 章　外壳编写基础

冯　典

看雪技术天才。看雪论坛 ID 为 bughoho，1990 年生，来自四川，看雪论坛虚拟机开发小组组长，目前工作主要是从事逆向研究。

个人自述：记得 14 岁时家里买了台电脑，使我对编程有了极大的兴趣。16 岁上高一时已对读书彻底不感兴趣，于是退学（现在的我才发现，我并不是对读书感兴趣，而是对教育制度的反感）。后来听了家人的意见，转读四川新华电脑学校，感受颇多，一月之后便退学，至于为什么我就不说了。17 岁时，一个偶然的机会，使我对逆向有了浓厚的兴趣，并接触到看雪论坛，也认识到了 kanxue。承蒙 kanxue 抬举，让我执笔虚拟机这一章，由于我并不是一个才高八斗的人，所以写得也没有那么的妙笔生花、鬼斧神工了。

参与章节：第 17 章　虚拟机的设计

罗　翼

看雪技术专家。资深程序员，由加/解密知识起接触编程，对 Windows 底层机制有多年的研究经验。后由于工作需要，接触 C++/ATL/COM 等技术。现致力于研究各种 Moder C++的元素的应用范围及其对降低程序复杂度所起的作用，热切关注 ISO C++以及分布式计算相关内容的进展。

参与章节：第 18 章　18.2.1 跨进程内存存取机制

18.2.2 Debug API 机制

18.2.3 利用调试寄存器机制

林小华

看雪资深版主。看雪论坛 ID 为 linhansh，武汉大学电力系统及其自动化专业，『工具分区』区版主，对论坛的工具版块发展做出了重大贡献。

个人主页：http://blog.csdn.net/linhanshi

参与章节：第 18 章　18.4 补丁工具

郭春杨

看雪技术专家。看雪论坛 ID 为 Yonsm，软件工程师，从事视频编解码和多媒体软件设计工作。对 Windows 和 Windows Mobile 系统有比较深入的了解。

主页：WWW.Yonsm.NET

E-mail：Yonsm@163.com

参与章节：附录 B　在 Visual C++中使用内联汇编

前　言

软件安全是信息安全领域的重要内容，涉及到软件相关的加密、解密、逆向分析、漏洞分析、安全编程以及病毒分析等。目前，国内高校对软件安全教育重视程度不够，许多方面还是空白。随着互联网应用的普及和企业信息化程度的不断提升，社会和企业对软件安全技术人才需求逐年上升，在计算机病毒查杀、网游安全、网络安全、个人信息安全等方面人才缺口很大，相关职位待遇较高。从就业角度来看，掌握软件安全相关知识和技能，不但可以提高自身的职场竞争能力，而且有机会发挥更大的个人潜力，获得满意的薪酬；从个人成长方面来说，研究软件安全技术有助于掌握许多系统底层知识，是提升职业技能的重要途径。作为一名合格的程序员，除了掌握需求分析、设计模式等外，如能掌握一些系统底层知识，熟悉整个系统的底层结构，对自己的工作必将获益良多。

本书以软件加密与解密为切入点，讲述了软件安全领域相关基础知识和技能。读者在阅读了本书的内容后，很容易在漏洞分析、安全编程、病毒分析等领域得到扩展。这些知识点的相互关联性，将促使读者开阔思路，使所学融会贯通，领悟更多的学习方法，提升自身学习能力。

本书是《加密与解密》的第三版，此书今天能够与读者见面，完全是广大读者的热情和鼓舞带来的成果，作者深表谢意。

关于看雪学院

本书作者是软件安全主题网站——“看雪学院”的站长。看雪软件安全网站（www.pediy.com）由 kanxue（作者网名）创建于 2000 年。网站历经 8 年多的发展，脱颖而出，凭借自身实力，已经成为中国软件安全领域公认的最权威的技术站点，影响深远。

2000 年初，笔者想找一些研究软件加解密的朋友交流一下，但十分令人遗憾的是，那时国内这方面的技术资料很缺乏，不成系统，大家的交流也十分有限。因此，笔者自己建立了一个主页“看雪学院”，期望与兴趣相投的朋友共同探讨加密与解密的知识。当初这个简单的网站，就是今天看雪软件安全网站的雏形，并且是当时国内唯一从技术角度研究软件加/解密的站点。很短的时间，这个站点就获得了大家的认同，并在广大网友的支持下，健康地成长起来。随着我们的努力，网站推出的软件调试论坛逐渐成为国内知名度最高的软件安全论坛，吸引了众多高手。

本着知识共享，一切免费的建站宗旨，看雪软件安全网站汇聚了大量高水平的技术文章，至今为止原创了数千余篇精华文章，极大地推动了国内软件安全技术的发展。2007 年论坛改名为看雪软件安全论坛，论坛在保持已有的软件加密与解密研究方面外，在漏洞分析、系统底层、病毒分析、Rootkit 等技术领域进行全面扩展，逐步发展为信息安全的综合服务网站。

多年来，看雪软件安全网站一直遵循纯技术的发展策略，不但在行业中树立了令人尊敬的专业形象，更使一大批专业人士和专家聚集在这里，形成了一个技术交流的网上家园，带动了大批对软件安全感兴趣的网友加入进来，构建起了一个围绕软件安全主题的活跃的大社区，历久弥新。正是看到这种技术气氛，不少知名的公司都很关注论坛技术人才，如微软公司信息安全部门、珠海金山毒霸公司、深圳腾讯公司、

360 安全中心、启明星辰以及部分网游公司等。

为了推进软件安全技术为社会和企业服务的理念，我们正在努力提升看雪网站的社会作用和价值，从而为关注信息安全的大众，提供更好的服务和技术产品。

看雪软件安全网站，汇聚了许多志趣相投的朋友，经历了风风雨雨的 8 年，一直走到今天实属不易。作为网站站长和此书的作者，本人在此由衷地感谢所有关心和支持我们的共同事业，参与共同发展的朋友们！每到网站最困难的时候，是你们伸出无私的援助之手，才让网站渡过了一个个难关，能有今天的大好局面！在此特别鸣谢以下朋友和机构的大力支持：

海城金航网络科技有限公司阿男为网站提供网站空间

雅联网络服务有限责任公司李智勇为论坛提供独立服务器

南京慧速科技发展有限公司刘小荣为论坛提供独立服务器

感谢陈超达为服务器安全维护所做的大量工作

本书的缘起

当今的信息社会里，安全技术越来越重要了，如何普及软件安全知识是作者始终关注的一个大问题。正是为了更好地将软件安全知识普及到社会各个领域的愿望，促成了本书的问世。

依托看雪学院的技术背景，由作者主编和主导的看雪软件安全系列书籍，目前已出版发行了《加密与解密——软件保护技术及完全解决方案》（简体版，繁体版）、《加密与解密（第二版）》（简体版，繁体版）、《软件加密技术内幕》等书籍；基于电子资料的形式，历年发行的《看雪论坛精华》被众多网站转载，保守计算，其下载量已经超过数百万份，极大地推动了国内软件安全技术的发展。

这是一本很难写的书，因为 2000 年时，软件安全是一个全新的领域。从 Windows 95 面世以来的 6 年内，市面上没有一本这方面的书，网上也缺乏相关资料。为了填补国内 Windows 平台上加密与解密书籍的空白，作者与看雪论坛的一流好手努力合作，克服种种困难，于 2001 年 9 月推出了国内第一本全面介绍 Windows 平台下软件加密与解密技术的书籍，这就是本书的第一版《加密与解密——软件保护技术及完全解决方案》。

在第一版中，我们试图从软件加密和解密这两个方面对当今流行的软件保护技术进行分析。希望读者看过此书之后，能够对各种流行的软件保护与破解技术有所了解。

第一版一面世就得到了广大读者的喜爱和认可，获得了 2002 年全国优秀畅销书奖（科技类）！在全国很多计算机专业书店获得了名列前茅的销售业绩，而且一年来在著名的华储网销售排行中都被排在前几名内。次年，本书在台湾发行了繁体版，得到了台湾读者的热烈欢迎。

2003 年 6 月以本书第一版为基础，完成了本书的第二版《加密与解密》。

笔者从 2004 年开始第三版的更新准备工作，这个版本编写时间比较长，前后用了四年多的时间才得已完稿。这是所有参与者共同的努力，是他们把自己才华中最精彩部分展现给大家了。

现在读者看到的这本 500 多页的图书，几乎包含了当今 Windows 32 位环境下软件保护技术的绝大部分内容，从基本的跟踪调试到深层的拆解脱壳；从浅显的分析注册到中高级软件保护与分析，其跨度之广、内容之深，国内至今尚无同类出版物能与之比肩。

第三版的变化

第三版是在《加密与解密》第二版与《软件加密技术内幕》两本书的基础上完成的，删除了第二版中的过时内容，将《软件加密技术内幕》一些知识点补充融合进来，结构更加合理。

1．讲解通俗，突出基础

本书加强了基础部分的篇幅，系统讲解软件逆向的整个基本流程，包括动态分析、静态分析，以及逆向分析的基础知识。比如重点讲解了逆向必备工具 OllyDbg 和 IDA 的用法，并详细讲述逆向分析的基础知识，初学者通过相关几章的学习，可以轻松入门。

2．案例丰富，覆盖面广

书中提供了大量的案例分析，方便读者理论与实践相结合。通过实际操作，提高读者的调试分析能力。

3．加强了密码学算法

密码学算法越来越多地应用在软件保护领域，调试软件必须对比较知名的密码学算法有一定的了解。“加密算法”这一章，讲解了常见密码学算法的应用。

4．新增.Net 技术

随着微软.Net 平台的推广，越来越多的开发者开始关注.Net 程序的安全。.Net 这章向读者普及了.Net 安全的基本知识。

5．加强脱壳基础知识的篇幅

脱壳一章的结构和内容规划，参考了大量的建议，组织更加合理，完全为脱壳新手量身定做。

6．软件保护技术实施

相关章节详细研究了大量极具商业价值的保护技术，包括反跟踪技术、外壳编写基础、虚拟机的设计等，读者完全可以将这些技术应用到自己软件保护之中去。

7．二次开发与补丁技术

“代码的二次开发”一章中讲解如何在没有源码的情况下，扩充程序功能，打造开发接口；“补丁技术”一章讲解如何自己编程实现内存补丁或内存注册机。

本书预备知识

在阅读本书前，读者应该对汇编语言有大致的了解。汇编语言是大学计算机的必修课，这方面的书籍品种很多，如《IBM PC 汇编语言程序设计》，虽然大多数书以 DOS 汇编为讲解平台，但对理解汇编指令功能依然有益。

读者如果熟悉和了解 C 语言，对阅读本书是很有帮助的。

建议掌握一些 Win32 编程，不论研究加密与解密，还是编程，都应该了解 Win32 编程。Win32 编程是 API 方式的 Windows 程序设计，学习 Windows API 能使读者更深入地了解 Windows 工作方式。此类书籍推荐您阅读 Charles Petzold 所著的《Windows 程序设计》，该书堪称经典之作，它以 C 语言为讲解平台。

到此为止，作者将不再假设你已经具有任何加/解密的经验了。

适合的读者

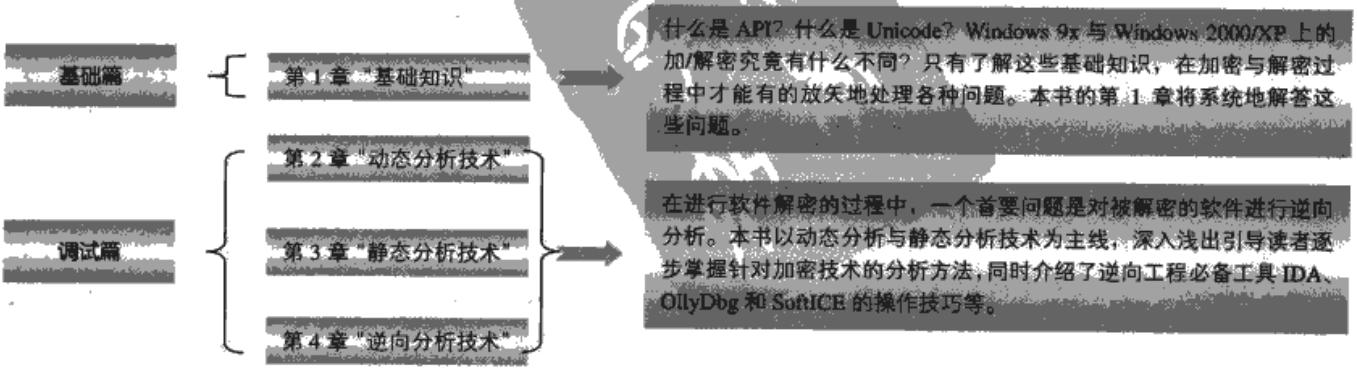
本书适合以下读者：

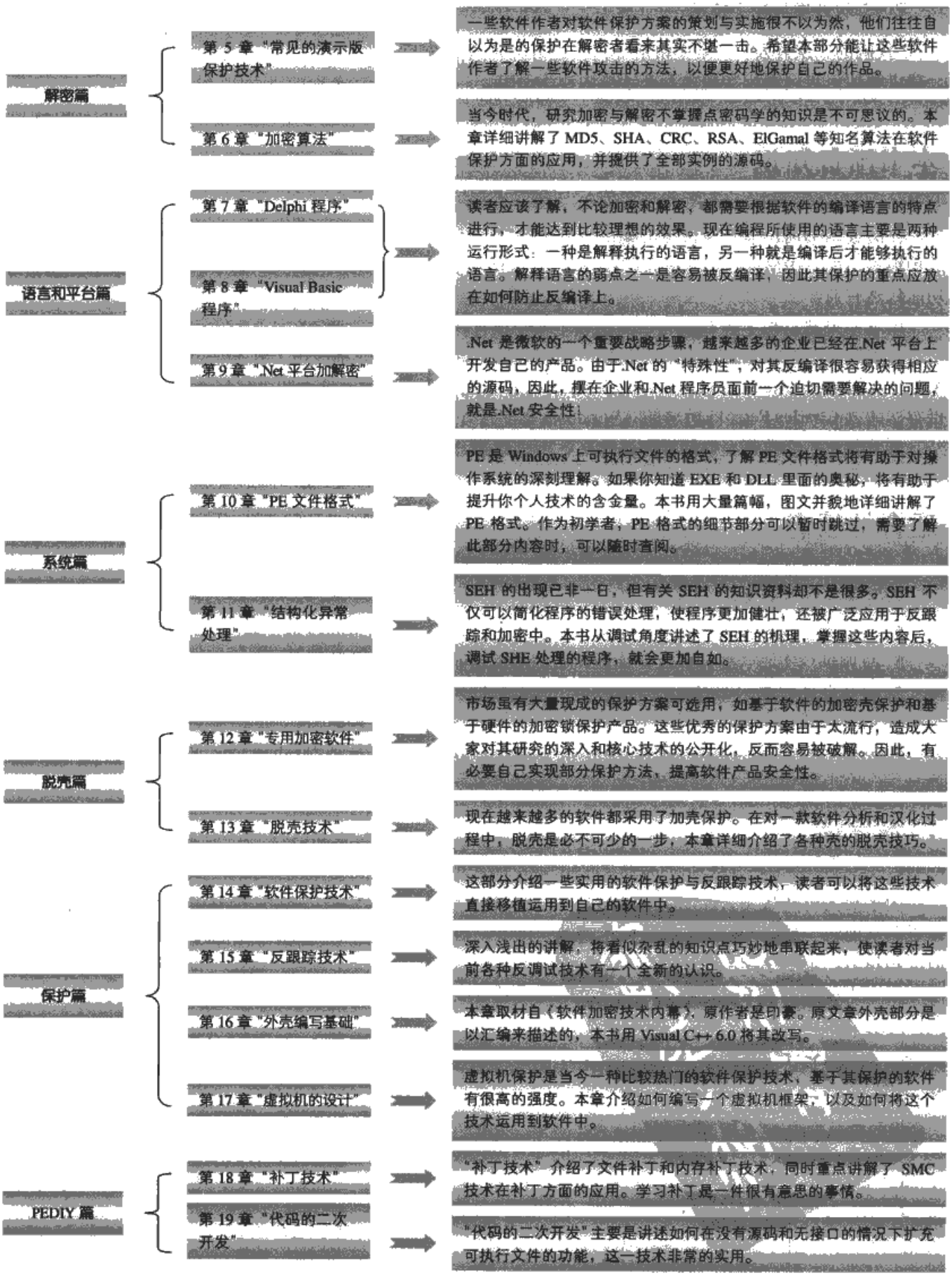
- 软件安全技术相关工作者：本书是软件安全研究的一本不错的技术字典；
- 对调试技术感兴趣的读者：提高读者的调试技能，增强软件的质量；
- 对软件保护感兴趣的软件开发人员：更好地保护你的作品；
- 大中专在校学生：通过本书掌握的相关知识和技能，将使你获得职场竞争的秘密武器；
- 其他：关注个人信息安全、计算机安全技术，并且想了解技术内幕的朋友，可以从中获得答案。

内容导读

大多数人可能认为软件加密与解密是一门高深的学问。造成这种认识的原因是以前这方面的技术资料缺乏，从而将“加密与解密”这一技术“神”化了。初学者一般不知从何下手，由于没方向，花费了大量时间和精力，走了不少弯路。本书给对这方面感兴趣的读者指明一个方向，提供一个捷径。

本书大部分章节既关联又彼此独立，因此读者可以根据自己情况，选择合适自己的内容阅读。




解密篇
第 5 章 "常见的演示版保护技术"
一些软件作者对软件保护方案的策划与实施很不以为然，他们往往自以为是的保护在解密者看来其实不堪一击。希望本部分能让这些软件作者了解一些软件攻击的方法，以便更好地保护自己的作品。
第 6 章 "加密算法"
当今时代，研究加密与解密不掌握点密码学的知识是不可思议的。本章详细讲解了 MD5、SHA、CRC、RSA、ElGamal 等知名算法在软件保护方面的应用，并提供了全部实例的源码。
语言和平台篇
第 7 章 "Delphi 程序"
第 8 章 "Visual Basic 程序"
读者应该了解，不论加密和解密，都需要根据软件的编译语言的特点进行，才能达到比较理想的效果。现在编程所使用的语言主要是两种运行形式：一种是解释执行的语言，另一种就是编译后才能够执行的语言。解释语言的弱点之一是容易被反编译，因此其保护的重点应放在如何防止反编译上。
第 9 章 ".Net 平台加解密"
.Net 是微软的一个重要战略步骤，越来越多的企业已经在.Net 平台上开发自己的产品。由于.Net 的"特殊性"，对其反编译很容易获得相应的源码，因此，摆在企业和.Net 程序员面前一个迫切需要解决的问题，就是.Net 安全性！
系统篇
第 10 章 "PE 文件格式"
PE 是 Windows 上可执行文件的格式，了解 PE 文件格式将有助于对操作系统的深刻理解。如果你知道 EXE 和 DLL 里面的奥秘，将有助于提升你个人技术的含金量。本书用大量篇幅，图文并貌地详细讲解了 PE 格式。作为初学者，PE 格式的细节部分可以暂时跳过，需要了解此部分内容时，可以随时查阅。
第 11 章 "结构化异常处理"
SEH 的出现已非一日，但有关 SEH 的知识资料却不是很多。SEH 不仅可以简化程序的错误处理，使程序更加健壮，还被广泛应用于反跟踪和加密中。本书从调试角度讲述了 SEH 的机理，掌握这些内容后，调试 SHE 处理的程序，就会更加自如。
脱壳篇
第 12 章 "专用加密软件"
市场虽有大量现成的保护方案可选用，如基于软件的加密壳保护和基于硬件的加密锁保护产品。这些优秀的保护方案由于太流行，造成大家对其研究的深入和核心技术的公开化，反而容易被破解。因此，有必要自己实现部分保护方法，提高软件产品安全性。
第 13 章 "脱壳技术"
现在越来越多的软件都采用了加壳保护。在对一款软件分析和汉化过程中，脱壳是必不可少的一步，本章详细介绍了各种壳的脱壳技巧。
保护篇
第 14 章 "软件保护技术"
这部分介绍一些实用的软件保护与反跟踪技术，读者可以将这些技术直接移植运用到自己的软件中。
第 15 章 "反跟踪技术"
深入浅出的讲解，将看似杂乱的知识点巧妙地串联起来，使读者对当前各种反调试技术有一个全新的认识。
第 16 章 "外壳编写基础"
本章取材自《软件加密技术内幕》，原作者是印霁。原文章外壳部分是以汇编来描述的，本书用 Visual C++ 6.0 将其改写。
第 17 章 "虚拟机的设计"
虚拟机保护是当今一种比较热门的软件保护技术，基于其保护的软件有很高的强度。本章介绍如何编写一个虚拟机框架，以及如何将这个技术运用到软件中。
PEDIY 篇
第 18 章 "补丁技术"
"补丁技术"介绍了文件补丁和内存补丁技术，同时重点讲解了 SMC 技术在补丁方面的应用。学习补丁是一件很有意思的事情。
第 19 章 "代码的二次开发"
"代码的二次开发"主要是讲述如何在没有源码和无接口的情况下扩充可执行文件的功能，这一技术非常的实用。

特别致谢

首先真诚感谢我的父母、妻子、女儿对我的大力支持，使得我顺利完成此书的编写！我所有的荣耀都属于你们。

谨此对电子工业出版社博文视点公司所有相关人员致以真诚的谢意！

特别感谢电子工业出版社博文视点公司总经理郭立所做的大量工作！

特别感谢上海盛大网络发展有限公司徐海侠、王峰、刘庆民、蒋渭华、李明、张子雁、史昕峰、彭伟、张静盛等对本书的大力支持！

特别感谢微软公司大中华区首席安全官江明灶和微软的战略安全架构专家裔云天对本书的支持！

特别感谢珠海金山毒霸事业部陈勇、赵闽的技术支持！

特别感谢看雪软件安全论坛核心管理团队 CCDebuger、Ivanov、riijj、michael 的支持！

特别感谢看雪软件安全论坛各版主及各技术小组成员，对本书的大力支持！他们是：

（1）北极星 2003、笨雄、rackabcer、nbragon、inhanshi、LOVE、monkeycz、逍遥风、小虾、zmworm

（2）软件调试小组：aker、hawking、elance、theOcrat

（3）虚拟机技术小组：bughoho、linxer、wangdell、Isaiah

（4）外壳开发小组：forgot、dummy、bithaha

（5）工具开发小组：doskey、netsowell、freecat、wak、menting

（6）编程技术小组：北极星 2003、没有风、CCDeath、Combojiang、Sislcb

（7）PTG 翻译小组：arhat、thinkSJ、kkbing、aalloverred、月中人、alpsdew、jdxyw、Jhlqb、mjahuolong

（8）.Net 小组：tankaiha、backer、dreaman、inraining、kkbing、lccracker、oep1、rick、slan、tracky、菩提!、MegaX

感谢 CCDebuger 对“第 2 章　动态分析技术”和“第 13 章　脱壳技术”校对！

感谢 gzgzlxg 对“第 3 章　静态分析技术”提出的修正和补充意见！

感谢 zmworm 对工具 IDA 使用的补充建议！

感谢 Intel 公司中国企业应用技术支持部的段夕华对“第 4 章 逆向分析技术”提出的宝贵修正意见！

感谢 WiNrOOt 翻译的 www.datarescue.com 提供的 IDA 简易教程，IDA 部分参考了一下！

感谢 riijj 为“5.6 网络验证”一节提供的实例！

感谢 cnbragon 参与的“第 6 章　加密算法”！

感谢 cyclotron 参与的“8.3 伪编译”！

感谢 tankaiha 参与的“第 9 章　.Net 平台加解密”！

感谢 Hume 对“第 11 章　结构化异常处理”提供的技术支持！

感谢 DiKeN 参与的“13.10 静态脱壳”！

感谢 softworm 参与的“13.9.2 Themidia 的 SDK 分析”！

感谢 forgot 参与的“14.2.4 简单的多态变形技术”、“第 15 章　反跟踪技术”！

感谢 Hying 参与的“第 16 章　外壳编写基础”！

感谢 bughoho 参与的“第 17 章　虚拟机的设计”！

感谢 afanty 参与的“14.1 防范算法求逆”！

感谢并参考老罗（www.luocong.com）“矛与盾的较量——CRC 实践篇”！

感谢 Lenus 在内存 Dump 和内存断点方面给予的技术支持！

感谢 TiANWEi 翻译的 SoftICE 手册！

感谢 wynney 签名制作的帮助！

感谢 skylly 为脱壳一章提供的脚本制作的技术支持！

感谢 hnhuqiong 提供的 ODbgScript 脚本教学！

感谢 linhansh 在工具方面提供的帮助！

感谢 VolX 为本书配套光盘映像文件提供的 Aspr2.XX_unpacker.osc 脚本！

感谢 CoDe_Inject 对“18.2.4 DLL 劫持技术”一节提供的帮助！

感谢武汉科锐软件培训中心（www.51asm.com）Backer 为“18.2.4 DLL 劫持技术”提供 lpk.cpp！

感谢 frozenrain、jero、mocha、NWMonster、petnt、sudami、tankaiha、wynney、XPoy、王清、小虾等朋友为术语表所做的工作！

感谢 Sun Bird、JoJo、kvllz 等人对本书的大力支持！

感谢 fonge 等诸多看雪论坛会员持续一年多来的发帖签名支持新书！

同时，也要感谢那些共同参与《加密与解密》（第一、二版）、《软件加密技术内幕》组稿的看雪软件安全论坛的众多一流好手，是他们的参与和奉献才让此书得以顺利完成。

这次的第三版改动较大，参考引用了如下朋友在《加密与解密》（第一、二版）中的文章：

（1）Blowfish 在第一版参与的“第 5 章　软件保护技术”；

（2）Fisheep 参与的“浮点指令小结”和“信息隐藏技术”；

（3）吴朝相（http://www.souxin.com）参与的“认识壳”；

（4）mr.wei 参与的“DeDe 用法”；

（5）感谢 pll621 在扩展 PE 功能开拓性的研究；

（6）娃娃（王凌迪）提供的“MD5 算法”资料。

参考并引用了如下朋友在《软件加密技术内幕》中的文章：

（1）Hying 的 Anti_Dump；

（2）Hume 的“第 4 章　Windows 下的异常处理”；

（3）王勇的“编写 PE 分析工具”；

（4）罗翼的“3.3 利用调试 API 制作内存补丁”；

（5）郭春杨的“在 Visual C++中使用内联汇编”；

（6）Ljtt 参与的“花指令”、“SMC 技术实现”、“壳的加载过程”；

（7）dREAMtHEATER 翻译的 Matt Pietrek　An In-Depth Look into the Win32 Portable Executable File Format。

在此，还要感谢看雪软件安全论坛其他朋友的支持和帮助！是你们提供的帮助，才使得我能够完成此书。如果以上未提及对您的谢意，在此，我表示由衷的感谢！

关于本书配套光盘映像文件

本书不提供配套光盘，光盘映像文件可以到本站主页下载。

由于版权问题，光盘映像文件仅提供书中提到的免费软件或共享软件。如果从学习角度需要使用那些有版权的软件，建议读者通过搜索引擎查找。

光盘映像文件提供的软件经过多方面检查测试，绝无病毒。但一些加/解密工具采用了某些病毒技术，因此部分代码与某些病毒的特征码类似，会造成查毒软件的误报。请自行决定使用。

建议将光盘映像中的文件拷贝到硬盘，并去除只读属性再调试，以免出现一些无法解释的错误。

光盘映像文件下载：http://book.pediy.com

反馈信息

我们非常希望能够了解读者对本书的看法。如果您有什么问题或自己的学习心得，欢迎发到看雪软件安全网站——看雪学院。

技术支持：http://www.pediy.com

邮件地址：kanxue@pediy.com

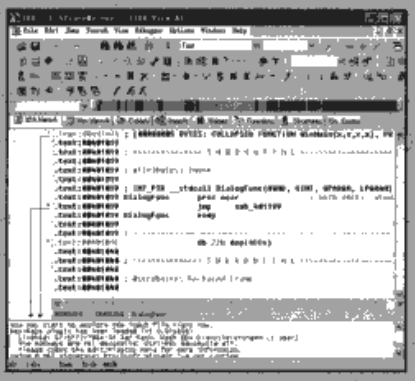
段　钢

2008.5.1 于上海

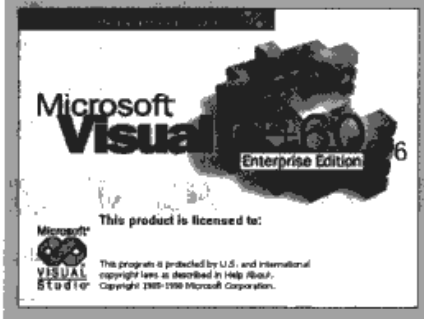
目　录

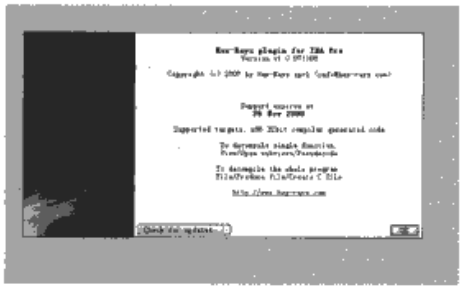
第一篇　基础篇

第二篇　调试篇

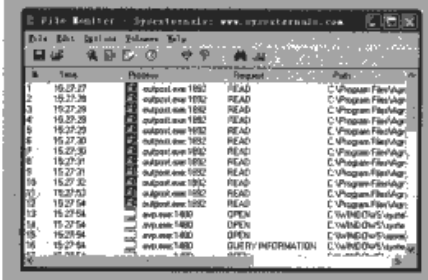


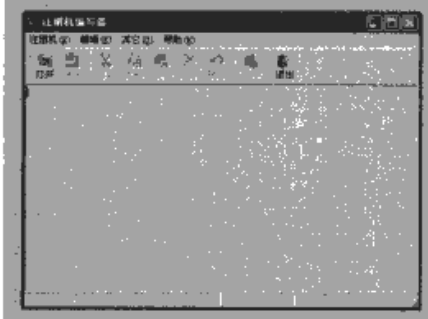
第 4 章 逆向分析技术 71



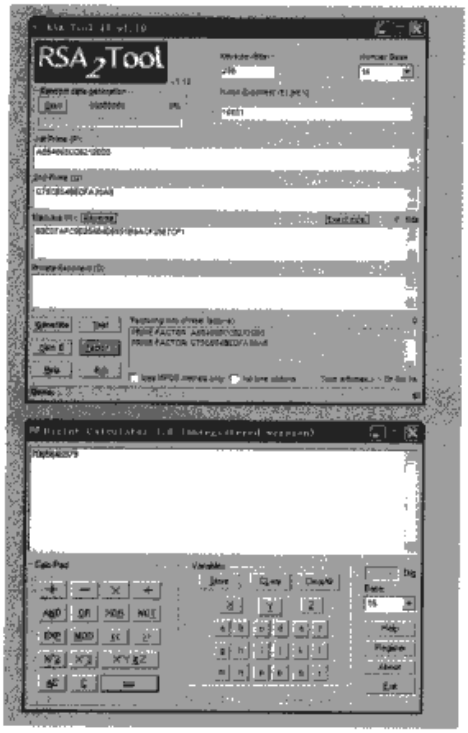


第三篇 解密篇

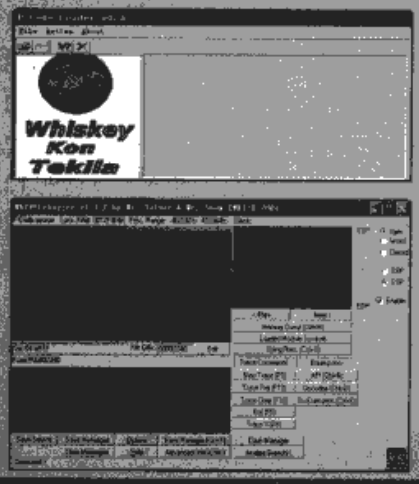




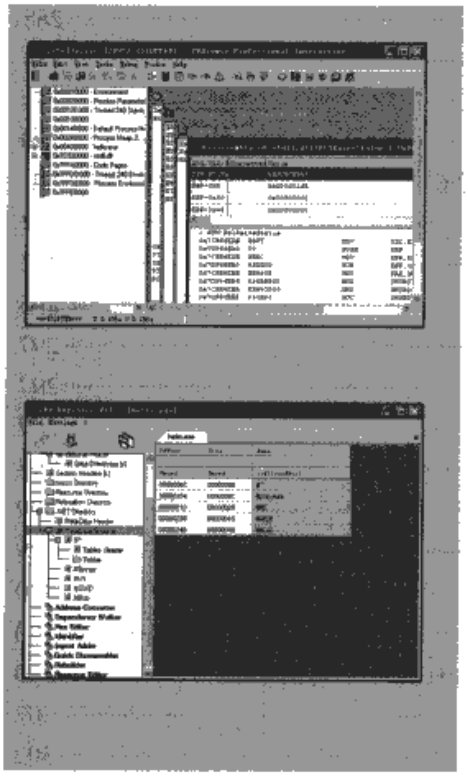
即使这些算法的强度很高，但是使用方法也要得当，否则效果就和普通的四则运算效果没有什么两样



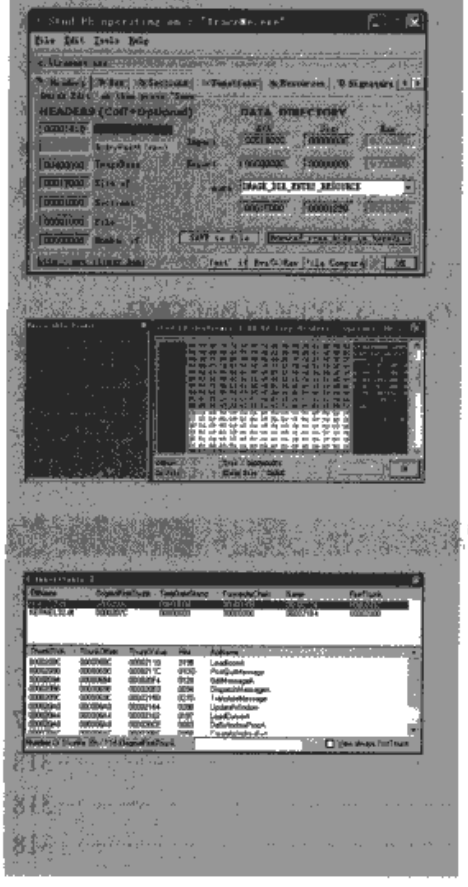
第四篇　语言和平台篇

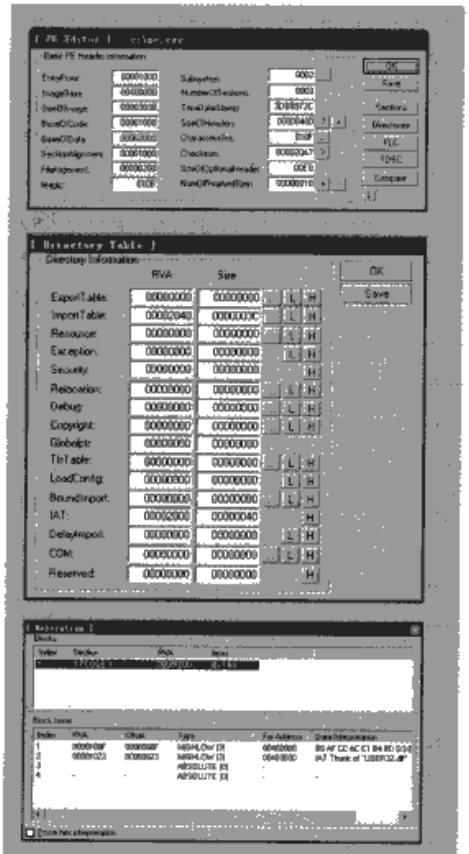


由于对.Net 的反编译很容易获得其源码，摆在企业和.Net 程序员面前一个迫切需要解决的问题，就是.Net 安全性！

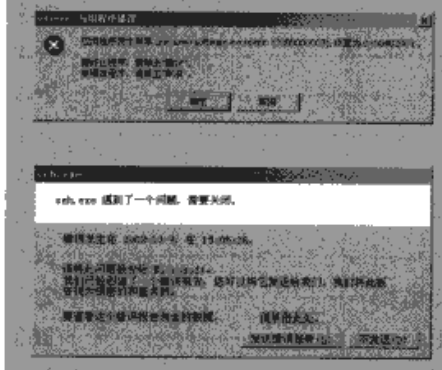


第五篇 系统篇



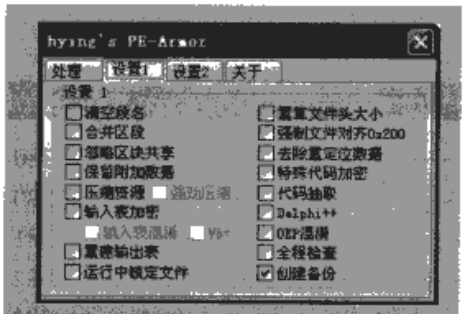


第 11 章 结构化异常处理 306

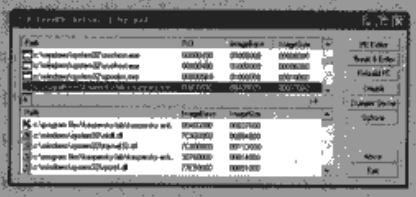


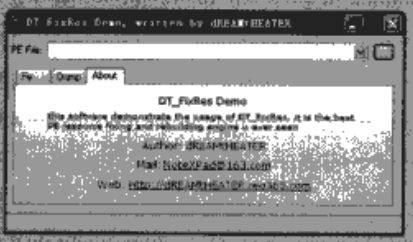
第六篇 脱壳篇

第 12 章 专用加密软件 316



第 13 章 脱壳技术 324

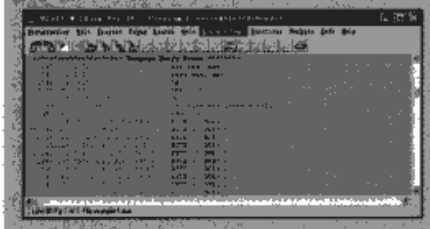




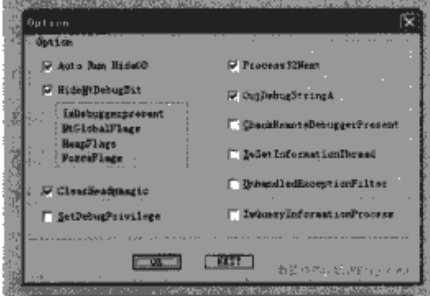
第七篇　保护篇

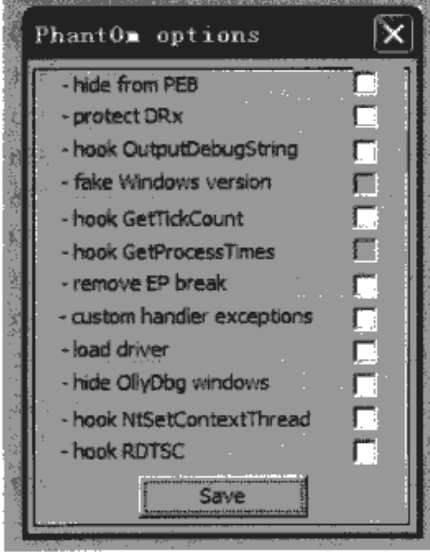
第 14 章　软件保护技术　380

市场上虽有大量现成的保护方案可选用，如基于软件的加密壳保护和基于硬件的加密锁保护。这些优秀的保护方案由于太流行，造成大家对其研究的透彻，反而容易被破解。所以，有必要自己实现相关的保护方法。

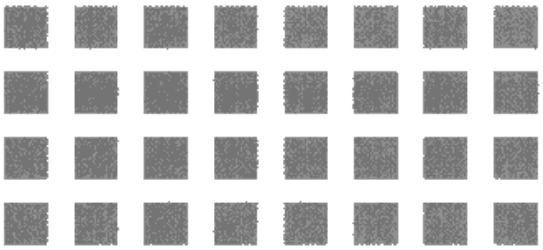


第 15 章　反跟踪技术　401





第八篇　PEDIY 篇



第 1 篇　基础篇

■ 第 1 章　基础知识

什么是 API？什么是 Unicode？Windows 9x 与 Windows 2000/XP 上的加密/解密究竟有什么不同？只有了解这些基础知识，在加密与解密过程中才能有的放矢地处理各种问题。本书的第 1 章将系统地解答这些问题。

第1章 基础知识

研究加密与解密，必须要了解一些 Windows 系统的基础知识，这样在分析的过程中才能有的放矢地处理各种问题。

1.1 文本字符

在学习过程中会与各类字符打交道，它们在 Windows 里扮演着重要角色。

1.1.1 字节存储顺序

多字节数据是按怎样的顺序存放的呢？实际情况和 CPU 有关，微处理机中的存放顺序有正序（Big-Endian）和逆序（Little-Endian）之分。常见的 Intel 体系芯片使用的编码方式属于 Little-Endian 类；某些 RISC 架构的 CPU，如 IBM 的 Power-PC 等属于 Big-Endian 类。

两种编码区别：

- Big-Endian　高位字节存入低地址，低位字节存入高地址，依次排列；
- Little-Endian　低位字节存入低地址，高位字节存入高地址，反序排列。

例如，将 12345678h 写入到以 1000h 开始的内存中，则结果如图 1.1 所示。本书以运行在 Intel x86 CPU 上的 Windows 为讲解平台，因此涉及的编码皆为 Little-Endian 类。

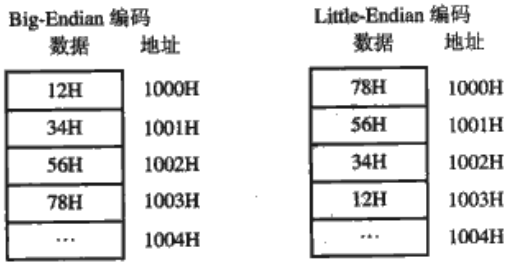


图 1.1　Big-Endian 与 Little-Endian 内存存储方式

1.1.2 ASCII 与 Unicode 字符集

美国信息交换标准码（ASCII）是一个 7 位的编码标准，包括 26 个小写字母、26 个大写字母、10 个数字、32 个符号、33 个控制代码和一个空格，总共 128 个代码。由于计算机通常用“字节”（byte）这个 8 位的存储单位来进行信息交换，因此不同的计算机厂家对 ASCII 进行了扩充，增加了 128 个附加的字

符来补充 ASCII，它们的值在 127 以上的部分是不统一的。例如 ANSI,Symbol,OEM 等字符集，其中 ANSI 是系统预设的标准文字存储格式。表 1-1 列出了用十六进制数（Hex）与十进制数（Dec）表示的部分常用字符的 ASCII 值。

表 1-1 常用字符的 ASCII 值

Hex	Dec	字 符	Hex	Dec	字 符	Hex	Dec	字 符	Hex	Dec	字 符
00H	00D	NUL	34H	52D	4	4DH	77D	M	66H	102D	f
07H	07D	BEL	35H	53D	5	4EH	78D	N	67H	103D	g
0AH	10D	LF	36H	54D	6	4FH	79D	O	68H	104D	h
0CH	12D	FF	37H	55D	7	50H	80D	P	69H	105D	i
0DH	13D	CR	38H	56D	8	51H	81D	Q	6AH	106D	j
20H	32D	SP	39H	57D	9	52H	82D	R	6BH	107D	k
21H	33D	!	3AH	58D	:	53H	83D	S	6CH	108D	l
22H	34D	"	3BH	59D	;	54H	84D	T	6DH	109D	m
23H	35D	#	3CH	60D	<	55H	85D	U	6EH	110D	n
24H	36D	$	3DH	61D	=	56H	86D	V	6FH	111D	o
25H	37D	%	3EH	62D	>	57H	87D	W	70H	112D	p
26H	38D	&	3FH	63D	?	58H	88D	X	71H	113D	q
27H	39D	'	40H	64D	@	59H	89D	Y	72H	114D	r
28H	40D	(	41H	65D	A	5AH	90D	Z	73H	115D	s
29H	41D	)	42H	66D	B	5BH	91D	[	74H	116D	t
2AH	42D	*	43H	67D	C	5CH	92D	\	75H	117D	u
2BH	43D	+	44H	68D	D	5DH	93D	]	76H	118D	v
2CH	44D	,	45H	69D	E	5EH	94D	↑	77H	119D	w
2DH	45D	-	46H	70D	F	5FH	95D	←	78H	120D	x
2EH	46D	.	47H	71D	G	60H	96D	`	79H	121D	y
2FH	47D	/	48H	72D	H	61H	97D	a	7AH	122D	z
30H	48D	0	49H	73D	I	62H	98D	b	7BH	123D	{
31H	49D	1	4AH	74D	J	63H	99D	c	7CH	124D	|
32H	50D	2	4BH	75D	K	64H	100D	d	7DH	125D	}
33H	51D	3	4CH	76D	L	65H	101D	e	7EH	126D	~

Unicode 是 ASCII 字符编码的一个扩展。只不过在 Windows 中，用两个字节对其进行编码，也称为宽字符集（Widechars）。Unicode 是一种双字节编码机制的字符集，使用 0～65535 之间的双字节无符号整数对每个字符进行编码。在 Unicode 中，所有的字符都是 16 位，包括所有的 7 位 ASCII 码都被扩充为 16 位（注意，高位扩充的是零）。如字符串“pediy”，它的 ASCII 码是：

70h 65h 64h 69h 79h

其 Unicode 码的十六进制是：

0070h 0065h 0064h 0069h 0079h

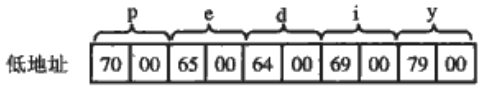


图 1.2 内存中的 Unicode 码

Intel 处理器在内存中，一个字存入存储器要占有相继的两个字节，这个字存放时就按 Little-Endian 方式存入，即低位字节存入低地址，高位字节存入高地址，如图 1.2 所示

1.2 Windows 操作系统

本书是研究Windows平台上的加解密，因此要求读者必须对操作系统要有所了解。如有可能，看看Windows操作系统原理方面的书籍，这对以后的一些深入理解是有帮助的。

1.2.1 Win API 简介

现在很多讲程序设计的书都是基于 MFC 库和 OWL 库的 Windows 设计，对 Windows 实现的细节鲜有讨论，而调试程序是和系统底层打交道的，所以很有必要掌握一些 API 函数的知识。

对于一个初学者来说，API 函数也许是一个时常耳闻却感觉有些神秘的东西。API 的英文全称为 Application Programming Interface（应用程序编程接口）。对这个定义的理解，需要追溯到操作系统的发展历史。当 Windows 操作系统开始占据主导地位的时候，开发 Windows 平台下的应用程序成为人们的需要。而在 Windows 程序设计领域处于发展的初期，Windows 程序员所能使用的编程工具唯有 API 函数。这些函数提供应用程序运行所需要的窗口管理、图形设备接口、内存管理等各项服务功能。这些功能以函数库的形式组织在一起，形成了 Windows 应用程序编程接口（API），简称 Win API。Win API 子系统负责将 API 调用转换成 Windows 操作系统的系统服务调用，所以，可以认为 API 函数是构筑整个 Windows 框架的基石，在它的下面是 Windows 的操作系统核心，而它的上面则是 Windows 应用程序，如图 1.3 所示。对于应用程序开发人员而言，所看到的 Windows 操作系统实际上就是 Win API，操作系统的其他部分对开发人员来说是完全透明的。

用于16位版本Windows的API(Windows 1.0到Windows 3.1)现在称做Win16。用于32位版本Windows的API（Windows 9x/NT/2000/XP/2003）现在称做Win32。API函数调用在从Win16到Win32的转变中保持兼容，并在数量和功能上不断增强，从Windows 1.0支持不到450个函数调用，到现在已有几千个函数。

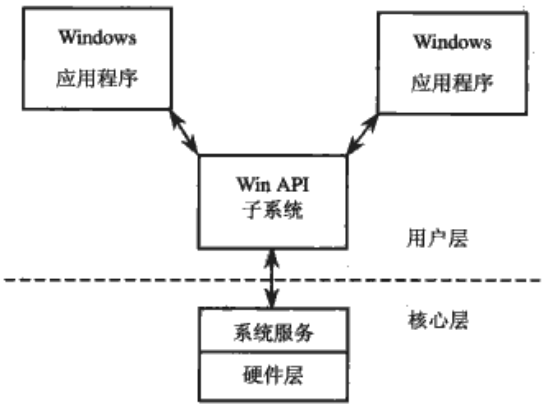


图 1.3　Windows 应用程序与操作系统的关系

所有 32 位版本的 Windows 都支持 Win16 API（以确保和旧应用程序兼容）和 Win32 API（以运行新应用程序）。非常有趣的是，Windows NT/2000/XP 与 Windows 9x 的工作方式不同。在 Windows NT/2000/XP 中，Win16 函数调用通过一个转换层被转化为 Win32 函数调用，然后被操作系统处理。在 Windows 9x 中，该操作正好相反：Win32 函数调用通过转换层转换为 Win16 位函数调用，再由操作系统处理。

Windows 运转的核心是一个称做“动态链接”的概念。Windows 提供了应用程序可利用的丰富的函数调用，这些函数采用动态链接库即 DLL 实现。在 Windows 9x 中通常位于\WINDOWS\SYSTEM 子目录中，在 Windows NT/2000/XP 中通常位于系统安装目录里的\SYSTEM 和\SYSTEM32 子目录中。

在早期，Windows 的主要部分只需要在三个动态链接库中实现。这代表了 Windows 的三个主要子系统，它们分别叫做 Kernel、User 和 GDI。

- Kernel（由 16 位的 KRNL386.EXE 和 32 位的 KERNEL32.DLL 实现）：操作系统核心功能服务，包括进程与线程控制、内存管理、文件访问等；
- User（由 16 位的 USER.EXE 和 32 位的 USER32.DLL 实现）：负责处理用户接口，包括键盘和鼠标输入、窗口和菜单管理等；
- GDI（由 16 位的 GDI.EXE 和 32 位的 GDI32.DLL 实现）：图形设备接口，允许程序在屏幕和打印机上显示文本和图形。

除了上述模块以外，Windows 还提供了其他一些 DLL 以支持另外一些功能，包括对象安全性、注册表操作（ADVAPI32.DLL）、通用控件（COMCTL32.DLL）、公共对话框（COMDLG32.DLL）、用户界面外壳（SHELL32.DLL）、图形引擎（DIBENG.DLL），以及网络（NETAPI32.DLL）。

注意：Windows 9x 是一个 16 位与 32 位的混合体，因此系统将 Kernel、User 和 GDI 等链接库的 16 位与 32 位版本一起加载到系统内存里。Windows NT/2000/XP 是一个纯 32 位操作系统，并没有加载 KRNL386.EXE 等 16 位链接库到内存中(也许为了兼容特殊的 16 位程序，才保留在 SYSTEM32 目录中)。

Win 32 API 是一个基于 C 语言的接口，但是 Win32 API 中的函数可以由用不同语言编写的程序调用，只要在调用时遵循调用的规范即可。

1.2.2 常用 Win32 API 函数

由于 Win32 程序大量调用系统提供的 API 函数，而 Win32 平台上的调试器，如 OllyDbg 等，恰好有针对 API 函数设置断点的强大功能，因而掌握常用的 API 函数具体用法会给跟踪调试程序带来极大的方便。本节将把常用的 Win32 API 函数介绍一下，详细的 Win32 API 参考文档可以从 MSDN 中获得。建议读者掌握一定的 Win32 编程知识，如阅读《Windows 程序设计》一书，这对合理选择 API 函数有很大的帮助。

API 函数是区分字符集的：A 表示 ANSI；W 表示 Widechars，即 Unicode。前者就是通常使用的单字节方式，后者是宽字节方式，以方便处理双字节字符。用字符串作参数的每个 Win32 函数在操作系统中都有两种方式的版本。例如，编程时使用 MessageBox 函数，而在 USER32.DLL 中，没有 32 位 MessageBox 函数的入口点。实际上，有两个入口点，一个名为 MessageBoxA（ANSI 版），另一个名为 MessageBoxW（宽字符版）。幸运的是，程序员通常不必关心这个问题，代码中只需要使用 MessageBox，开发工具中的编译模块就会根据设置决定采用 MessageBoxA 还是 MessageBoxW。

常用 API 函数详解：

（1）hmemcpy 函数

```
void hmemcpy(                          // 目的数据地址
  void _huge *hpvDest,                 // 源数据地址
  const void _huge *hpvSource,         // 数据大小（字节）
  long cbCopyn
);
```

这是一个 Win16 API 函数，位于 16 位的 KRNL386.EXE 链接库里。但一般的编程书籍上很少提到，原因是它是系统底层的东西，没有特殊需要，一般不直接调用。它执行的操作很简单，只是将内存中的一块数据拷贝到另一个地方。

上文已说过，在 Windows 9x 中，Win32 函数调用通过转换层转换为 Win16 函数调用，所以 Windows 9x 底层频繁地调用 hmemcpy 这个 16 位的函数来拷贝数据。由于这个特性，它常被解密者作为断点拦截数据，从而有个别称“万能断点”。在 Windows NT/2000 系统上，相关的函数是 Memcpy，但在 Windows NT/2000 上不同于在 Windows 9x 上，应用程序很少再调用 Memcpy 来处理数据，用此，函数设置断点基本上什么也拦不住。

（2）GetWindowText 函数

此函数在 USER32.DLL 用户模块中，它的作用是取得一个窗体的标题文字，或者一个文本控件的内容。函数原型：

```
int GetWindowText(
  HWND hWnd,                    // 窗口或文本控件句柄
  LPTSTR lpString,              // 缓冲区地址
  int nMaxCount                 // 复制的最大字符数
);
```

返回值：如果成功就返回文本长度；失败则返回零值。

ANSI 版是 GetWindowTextA，Unicode 版是 GetWindowTextW。

（3）GetDlgItem 函数

此函数在 USER32.DLL 用户模块中，它的作用是获取指定对话框的句柄。函数原型：

```
HWND GetDlgItem(
    HWND hDlg,                  // 对话框句柄
    int nIDDlgItem              // 控件标识
);
```

返回值：如果成功就返回对话框的句柄；失败则返回零。

（4）GetDlgItemText 函数

此函数在 USER32.DLL 用户模块中，它的作用是获取对话框文本。函数原型：

```
UINT GetDlgItemText(
    HWND hDlg,                  // 对话框句柄
    int nIDDlgItem,             // 控件标识（ID 号）
   LPTSTR lpString,             // 文本缓冲区指针
    int nMaxCount               // 字符缓冲区的长度
);
```

返回值：如果成功就返回文本长度；失败则返回零。

ANSI 版是 GetDlgItemTextA，Unicode 版是 GetDlgItemTextW。

（5）GetDlgItemInt 函数

此函数在 USER32.DLL 用户模块中，它的作用是获取对话框整数值。函数原型：

```
UINT GetDlgItemInt(
    HWND hDlg,                  // 对话框句柄
    int nIDDlgItem,             // 控件标识
    BOOL *lpTranslated,         // 接收成功/失败指示的指针
    BOOL bSigned                // 指定是有符号数还是无符号数
);
```

返回值：如果成功，lpTranslated 被设置为 TRUE，返回文本对应的整数值；如果失败，lpTranslated 被设置为 FALSE，返回值为零。

（6）MessageBox 函数

此函数是在 USER 32.DLL 用户模块中，创建和显示信息框。函数原型：

```
int MessageBox(
  HWND hWnd,                    // 父窗口句柄
  LPCTSTR lpText,               // 消息框文本地址
  LPCTSTR lpCaption,            // 消息框标题地址
  UINT uType                    // 消息框样式
);
```

ANSI 版是 MessageBoxA，Unicode 版是 MessageBoxW。

1.2.3 什么是句柄

句柄（Handle）在 Windows 中使用非常频繁，它是 Windows 标识，由应用程序建立或使用的对象所使用的一个唯一的整数值（通常为 32 位）。Windows 要使用各种各样的句柄来标识诸如应用程序实例、窗口、图标、菜单、输出设备、文件等对象。程序通过调用 Windows 函数获取句柄，然后在其他 Windows 函数中使用这个句柄，以引用它代表的对象。句柄的实际值对程序来说无关紧要，这个值是被 Windows 模块内部用来引用相应对象的。

当一个进程被初始化时，系统要为它分配一个句柄表，句柄值是放入进程的句柄表中的索引。当调试一个应用程序并且观察内核对象句柄的实际值时，会看到一些较小的值，如 1,2 等。请记住，句柄的含义并没有记入文档资料，并且可能随时变更。实际上，在 Windows 2000 中，返回的值用于标识放入进程的句柄表的该对象的字节数，而不是索引号本身。因此，在 Windows 的不同版本下调试程序时，就不要为句柄值的表达形式不同而疑惑了。

1.2.4 Windows 9x 与 Unicode

Windows 9x 操作系统不是一种全新的操作系统。为了向下兼容，它继承了 16 位 Windows 3.x 操作系统的特性。因此，微软并没有把所有的 16 位函数全部改写成 32 位的，而是仅仅通过将其包装进 32 位代码重用了现有的 16 位代码。这样，这些 32 位代码会转过来调用 16 位。在这种 Win32 API 实现下，KERNEL32 的大多数函数都转到了 KRNL386，USER32 转到了 USER.EXE，GDI32 转到了 GDI.EXE。

如果要增加 Windows 9x 对 Unicode 的支持，其工作量相当大。Windows 9x 几乎都是使用 ANSI 字符串来进行所有的内部操作的。但 Windows 9x 里还是有少量函数具有支持 Unicode 的能力，如下面的 Win9x_Unicode.exe 程序：

```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
{
    MessageBox(NULL, TEXT ("Windows 9x 支持 Unicode? "), TEXT ("Hello"), 0) ;
    return 0 ;
}
```

在 VC 编译器中，设置好 Unicode 标识符，按 Unicode 方式编译，生成 Unicode 程序。可以用反汇编工具查看其汇编代码：

```
00401000  push    0                               ; /Style = MB_OK|MB_APPLMODAL
00401002  push    40505C                          ; |Title = "Hello"
00401007  push    405030                          ; |Text = "Windows 9x 支持 Unicode? "
0040100C  push    0                               ; |hOwner = NULL
0040100E  call    dword ptr [404094]              ; \MessageBoxW
00401014  xor     eax, eax
00401016  retn    10
```

很明显，程序中存在一个 Unicode 版本的 MessageBox 函数，这时在 Windows 9x 上执行这个程序，结果程序正常运行。

这个实例演示了部分 Unicode 函数能在 Windows 9x 上运行得很好。现在来看一看 Windows 98 里 MessageBoxW 函数的内部结构：

```
int MessageBoxW(
  MessageBoxExW{                                // 调用 MessageBoxExW() 函数
    WideCharToMultiByte( );                     // 取得 Unicode 字符串“Windows 9x 支持 Unicode”的长度
    GlobalAlloc( );                             // 按字符长度分配内存
    WideCharToMultiByte( );                     // 将字符串“Windows 9x 支持 Unicode”转换成 ANSI 字符串
```

```
    WideCharToMultiByte( );                    // 取得 Unicode 字符串“Hello”长度
    GlobalAlloc( );                            // 分配内存
    WideCharToMultiByte( );                    // 将 Unicode 字符串“Hello”转换成 ANSI 字符串
    MessageBoxExA( );                          // 最终还是调用 ANSI 版的 MessageBoxExA 函数显示窗口
    GlobalFree( );                             // 释放内存
    GlobalFree( );                             // 释放内存
  }
);
```

原来，MessageBoxW 函数将 Unicode 字符串转换成 ANSI 字符串，最终调用 ANSI 版的 MessageBoxExA 函数来显示窗口。

这个实例涉及 Unicode 与 ANSI 之间转换字符串的操作，程序调用了 WideCharToMultiByte 函数进行转换。

如要将 ANSI 字符串转换成 Unicode 字符串，可调用 MultiByteToWideChar 函数来实现。

Windows 9x 系统上还有其他几个比较常用的 Unicode 函数，如 lstrlenW, FindResourceW, GetCommandLine, ExtTextOut, TextOutW, MultiByteToWideChar 等。

Windows 9x 上其他成百上千的函数只提供了接受 Unicode 参数的进入点，但是这些函数并不将 Unicode 字符串转换成 ANSI 字符串，只返回运行失败的消息。这些函数只有 ANSI 版本才能正确运行。所以，如要为 Windows 9x 系统开发软件，只能用 ANSI 版函数开发应用程序，Unicode 版本的程序在 Windows 9x 下无法正常运行。

1.2.5 Windows NT/2000/XP 与 Unicode

在 NT 架构上，Win32 API 也是以 KERNEL32、USER32 和 GDI32 动态链接库提供的。然而，这种实现完全从底层做起没有使用任何的 16 位代码，因此是 Win32 API 的纯 32 位实现。甚至 16 位代码都最终要调用这些 32 位 API。

Unicode 影响到计算机工业的每个部分，对操作系统和编程语言的影响最大。NT 系统是使用 Unicode 标准字符集从头进行开发的，其系统核心完全是用 Unicode 函数工作的。调用任何一个 Windows 函数并给它传递一个 ANSI 字符串，那么系统首先要将字符串转换成 Unicode，然后再将 Unicode 传递给操作系统。相反，如果希望函数返回 ANSI 字符，系统就会首先将 Unicode 字符串转换成 ANSI 字符串，然后将结果返回给应用程序。也就是说，在 NT 架构下，Win32 API 能接受 Unicode 和 ASCII 两种字符集，而其内核则只能使用 Unicode。所有这些操作对用户来说都是透明的，但进行这些字符串的转换需要占用系统资源。

现在来看一看 Windows 2000 里 MessageBoxA 函数的内部结构：

```
int MessageBoxA(
  MessageBoxExA(                               // 调用 MessageBoxExA 函数
    MBToWCSEx( )                               // 将 MessageBoxA 消息框主体文字转换成 Unicode 字符串
    MBToWCSEx( )                               // 将 MessageBoxA 消息框标题栏上的文字转换成 Unicode 字符串
    MessageBoxExW( )                           // 调用 MessageBoxExW 函数
    HeapFree( )                                // 释放内存
  )
);
```

这个试验结果表明 MessageBoxExA 函数其实是一个替换翻译层，用于分配内存，并将 ANSI 字符串转换成 Unicode 字符串，系统最终是调用 Unicode 版的 MessageBoxExW 函数执行。当 MessageBoxExW 返回时，它便释放内存缓存。在这个过程中，系统必须执行这些额外的转换操作，因此 ANSI 版的应用程序需要更多的内存和占用更多的 CPU 资源。而 Unicode 版的程序在 Windows 2000/XP 下执行效率就高多了。

Windows 2000/XP 既支持 Unicode，也支持 ANSI，所有新增和未过时的函数在 Windows 2000/XP 中都同时拥有 ANSI 和 Unicode 两个版本。

1.2.6 Windows 消息机制

Windows 是一个消息（Message）驱动式系统，Windows 消息提供应用程序与应用程序之间、应用程序与 Windows 系统之间进行通信的手段。应用程序想要实现的功能由消息来触发，并且靠对消息的响应和处理来完成。

Windows 系统中有两种消息队列：一种是系统消息队列，另一种是应用程序消息队列。计算机的所有输入设备由 Windows 监控。当一个事件发生时，Windows 先将输入的消息放入系统消息队列中，再将输入的消息拷贝到相应的应用程序队列中，应用程序中的消息循环从它的消息队列中检索每个消息并且发送给相应的窗口函数中。一个事件的发生，到达处理它的窗口函数必须经历上述过程。值得注意的是消息的非抢先性，即不论事件的急与缓，总是按到达的先后排队（一些系统消息除外），这就使得一些外部实时事件可能得不到及时的处理。

由于 Windows 本身是由消息驱动的，所以调试程序时跟踪一个消息会得到相当底层的答案。下面将常用的 Windows 消息函数列出，以方便需要时参考。

（1）SendMessage 函数

调用一个窗口的窗口函数，将一条消息发给那个窗口。除非消息处理完毕，否则该函数不会返回。

```
LRESULT SendMessage(
  HWND hWnd,                    // 目的窗口的句柄
  UINT Msg,                     // 消息标识符
  WPARAM wParam,                // 消息的 WPARAM 域
  LPARAM lParam                 // 消息的 LPARAM 域
);
```

返回值：由具体的消息决定。如消息投递成功，则返回 TRUE（非零）。

（2）WM_COMMAND 消息

当用户从菜单或按钮中选择一条命令或者一个控件时发送给它的父窗口，或者当一个快捷键被释放时发送。Visual C++的 WINUSER.H 文件里定义了 WM_COMMAND 消息对应的十六进制是 0111h。

```
WM_COMMAND
  wNotifyCode = HIWORD(wParam);     // 通告代码
  wID = LOWORD(wParam);             // 菜单条目、控件或快捷键的标识符
  hwndCtl = (HWND) lParam;          // 控件句柄
```

返回值：如果应用程序处理这条消息，则返回值为零。

（3）WM_DESTROY 消息

当一个窗口被破坏时发送。WM_DESTROY 消息的十六进制是 02h。

这条消息无参数。

返回值：如果应用程序处理这条消息，则返回值为零。

（4）WM_GETTEXT 消息

应用程序发送一条 WM_GETTEXT 消息，将一个对应窗口的文本拷贝到一个呼叫程序提供的缓冲区中。WM_GETTEXT 消息的十六进制是 0Dh。

```
WM_GETTEXT
  wParam = (WPARAM) cchTextMax;     // 需要拷贝的字符数
  lParam = (LPARAM) lpszText;       // 接收文本的缓冲区地址
```

返回值：被拷贝的字符数。

（5）WM_QUIT 消息

当应用程序调用 PostQuitMessage 函数时，生成消息 WM_QUIT。WM_QUIT 消息的十六进制数是 012h。

```
WM_QUIT
 nExitCode = (int) wParam;                    // 退出代码
```

返回值：这条消息没有返回值。

（6）WM_LBUTTONDOWN 消息

当光标在一个窗口的客户区并且用户按下鼠标左键时，WM_LBUTTONDOWN 消息被发送。如果鼠标动作未被捕获，这条消息被发送给光标下的窗口；否则，被发送给已经捕获鼠标动作的窗口。WM_LBUTTONDOWN 消息的十六进制是 0201h。

```
WM_LBUTTONDOWN
fwKeys = wParam;                              // key 旗标
xPos = LOWORD(lParam);                        // 光标的水平位置
yPos = HIWORD(lParam);                        // 光标的垂直位置
```

返回值：如果应用程序处理这条消息，则返回值为零。

1.3 保护模式简介

学过 8088/8086 汇编语言的读者，一定熟悉 AX, BX, CX, DX, SI, DI, SP, BP 和 IP 这些 16 位的寄存器；在 80386 中，这些寄存器被扩展到了 32 位，即 EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EIP，段寄存器增加了两个：FS 和 GS。

在 64 位 CPU 中，这些寄存器被扩展到了 64 位，即 RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, RIP，新增 8 个通用寄存器（R8～R15）。所有的这些寄存器能够具有字节、字、双字和四字 4 个级别。

一般来说，80x86（80386 及其以后的各代 CPU）可在实模式、保护模式和虚拟 86 模式 3 种模式下运转。实模式就是古老的 MS-DOS 的运行环境，只能利用这些 32 位寄存器的前 16 位，而后面的 16 位就浪费了。当前流行的 Windows 操作系统运行在保护模式下，在保护模式下程序可以利用更多的内存，可以实现多任务系统。

1.3.1 虚拟内存

在保护模式下，CPU 的寻址方式与实模式不同。实模式下的寻址方式是“段基址＋段偏移”，段的默认大小为 64KB，所有段都是可读/写的，唯有代码段是可执行的，段的特权级为 0。而在保护模式下内存是“线性”的，因为这时段寄存器的意义不同，它里面存放的不再是段基地址，而是存放着段选择子，这个值是不直接参与寻址的，只是全局描述符表（Global Descriptor Table，GDT）或本地描述符表（Local Descriptor Table，LDT）的一个指针，不同段寄存器有不同的属性（读、写、执行、特权级等），如图 1.4 所示。尽管如此，在继续看保护模式内存结构时，仍请记住段/偏移量的概念，不妨把段寄存器看做是对保护模式中选择子的一个模拟。关于段描述符的定义，读者可参考其他保护模式的书籍资料。

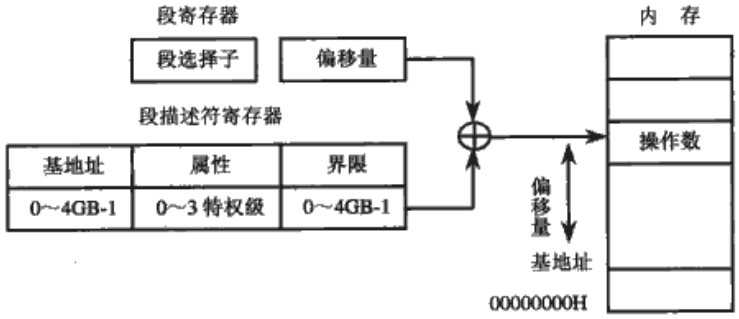


图 1.4 保护模式的寻址操作

Win32 的平坦内存模式使每个进程拥有赋予它自己的虚拟空间，对于 32 位进程来说，这个地址空间是 4GB，因为 32 位指针可以拥有从 00000000h～FFFFFFFFh 之间的任何一个值。此时，程序的代码和数据都放在同一地址空间中，即不必区分代码段和数据段。程序员也不必了解段寄存器 CS, DS, ES 等的具体内容。

虚拟内存（Virtual Memory）不是真正的内存，它通过映射（Map）的方法，使可用的虚拟地址（Virtual Address）达到 4GB，每个应用程序可以被分配 2GB 的虚拟地址，剩下的 2GB 留给操作系统自己用。在 Windows NT 中，应用程序甚至可有 3GB 的虚拟地址。Windows 是一个分时的多任务操作系统，CPU 时间被分成一个个的时间片后分配给不同程序，在一个时间片里，和这个程序执行不无关的东西并不映射到线性地址中。因此每个程序都有自己的 4GB 寻址空间，互不干扰。在物理内存中，操作系统和系统 DLL 代码需要供每个应用程序调用，所以在所有的时间必须映射；用户 EXE 程序只在自己所属的时间片内被映射，而用户 DLL 则有选择地被映射。

简单地说，虚拟内存的实现方法和过程如下：

① 当一个应用程序被启动时，操作系统就创建一个新进程，并给每个进程分配 2GB 的虚拟地址（不是内存，只是地址）；

② 虚拟内存管理器将应用程序的代码映射到那个应用程序的虚拟地址中的某个位置，并把当前所需要的代码读取到物理地址中（注意：虚拟地址和应用程序代码在物理内存中的位置是没有关系的）；

③ 如果使用动态链接库 DLL，DLL 也被映射到进程的虚拟地址空间，在需要的时候才被读入物理内存；

④ 其他项目（例如数据、堆栈等）的空间是从物理内存中分配的，并被映射到虚拟地址空间中；

⑤ 应用程序通过使用它的虚拟地址空间中的地址开始执行，然后虚拟内存管理器把每次的内存访问映射到物理位置。

如果看不明白上面的步骤也不要紧，但要明白以下几点：

- 应用程序是不会直接访问物理地址的；
- 虚拟内存管理器通过虚拟地址的访问请求，控制所有的物理地址访问；
- 每个应用程序都有相互独立的 4GB 寻址空间，不同应用程序的地址空间是隔离的；
- DLL 程序没有自己的“私有”空间，它们总是被映射到其他应用程序的地址空间中，作为其他应用程序的一部分运行。因为如果它不和其他程序同属一个地址空间，应用程序就无法调用它。

使用虚拟内存的好处是：简化了内存的管理，并可弥补物理内存的不足；可以防止多任务环境下各个应用程序之间的冲突。

1.3.2 保护模式的权限级别

在保护模式下，所有的应用程序都有权限级别（Privilege Level，PL），这个权限级别按优先次序分为 4 等：0, 1, 2 和 3，其中 3 特权级别最低，0 特权级别最高。特权级环如图 1.5 所示。

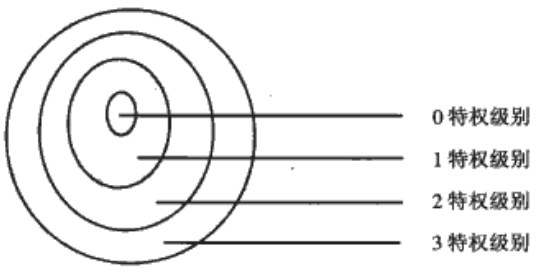


图 1.5 特权级环

如果应用程序拥有第 0 级的权限（也就是说，其 PL=0，或者它是运行在 Ring 0 的应用程序），它就可以执行所有的指令并访问所有数据；如果应用程序拥有的权限级别是第 3 级，它能执行的指令是有限的，能访问的数据也是有限的。操作系统核心层是运行在 Ring 0 级的，而 Win32 子系统（如动态链接库 KERNEL32.DLL、USER32.DLL 和 GDI32.DLL）是运行在 Ring 3 级的，以提供与应用程序的接口。

图 1.6 所示是 Windows 2000/XP 体系结构简图，核心态工作在 Ring 0 级，用户态工作在 Ring 3 级。HAL 是一个可加载的核心模块 HAL.DLL，它为运行在 Windows 2000/XP 上的硬件平台提供低级接口。Windows 2000/XP 的执行体是 NTOSKRNL.EXE 的上层（内核是其下层）。用户层导出并且可以调用的函数接口在 NTDLL.DLL 中，通过 Win32 API 或一些其他的环境子系统对它们进行访问。

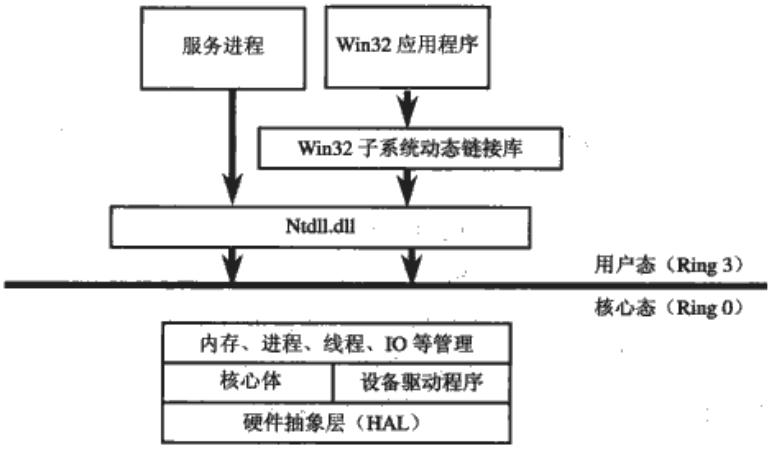


图 1.6　Windows 2000/XP 体系结构简图

另外，用户的应用程序也是运行在 Ring 3 级的（就是用 Visual C++, Borland C++, Visual Basic, Delphi, Borland C++ Builder 等 SDK 工具开发的应用程序），也就是说，享有的权限是最低的——换言之，受到保护模式的“保护”。该类应用程序没有权限去破坏操作系统，只能规矩地使用 Win32 API 接口函数与系统打交道。如想控制系统，就必须取得 0 特权级，比如调试工具 SoftICE 就是工作在 0 特权级上的。

1.4 认识 PE 格式

Windows 的可执行文件（EXE, DLL）是 PE（Portable Executable）格式。本节先简单介绍一下 PE 文件，详细的 PE 格式请参考第 10 章“PE 文件格式”。

PE 文件使用的是一个平面地址空间，所有代码和数据都被合并在一起，组成一个很大的结构。文件的内容被分割为不同的区块（Section，又称区段、节等），块中包含代码或数据。每个块都有它自己在内存中的一套属性，比如：这个块是否包含代码、是否只读或可读/写等。

每一个区块都有不同的名字，这个名字用来表示区块的功能。例如，一个叫.rdata 的区块表明它是一个只读区块。常见的块有.text, .rdata, .data, .idata, .rsrc 等。各种块的含义如下。

- .text：是在编译或汇编结束时产生的一种块，它的内容全是指令代码；
- .rdata：是运行期只读数据；
- .data：是初始化的数据块；
- .idata：包含其他外来 DLL 的函数及数据信息，即输入表；

- .rsrc：包含模块的全部资源，如图标、菜单、位图等。

PE 文件非常好的一个地方就是在磁盘上的数据结构与在内存中的结构是一致的，如图 1.7 所示。装载一个可执行文件到内存中，主要就是将一个 PE 文件的某一部分映射到地址空间中。这样，PE 文件的数据结构在磁盘和内存中是一样的。

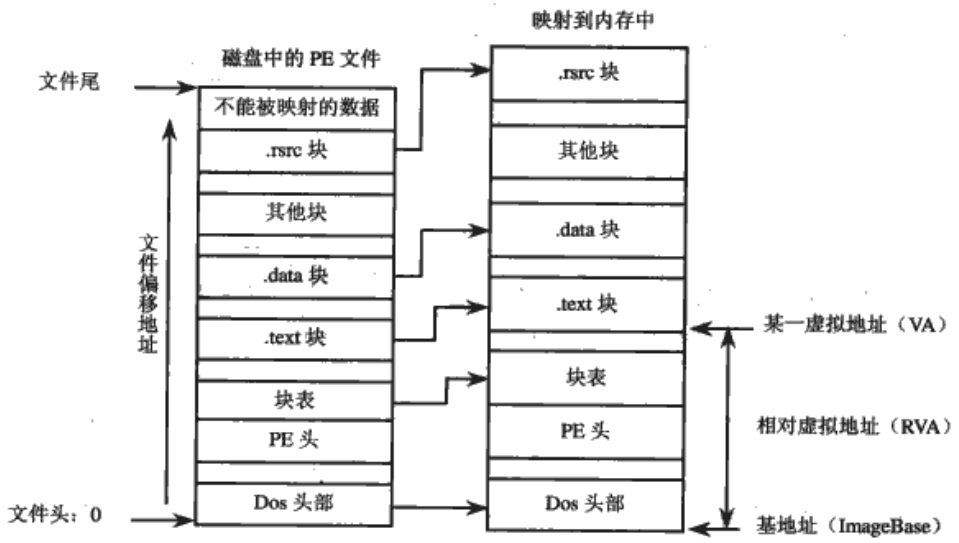


图 1.7 PE 文件结构

PE 相关名词解释如下。

（1）入口点（Entry Point）

PE 文件执行时的入口点（Entry Point）。也就是说，程序在执行时的第一行代码的地址应该就是这个值。

（2）文件偏移地址（File Offset）

当 PE 文件储存在磁盘上时，各数据的地址称做文件偏移地址（File Offset）。文件偏移地址从 PE 文件的第一个字节开始计数，起始值为 0。

（3）虚拟地址（Virtual Address，VA）

由于 Windows 程序运行在 386 保护模式下，所以程序访问存储器所使用的逻辑地址称为虚拟地址（Virtual Address，VA），又称为内存偏移地址（Memory Offset）。与实地址模式下的分段地址类似，虚拟地址也可写成“段:偏移量”的形式，这里的段是指段选择子。例如，“0167: 00401000”就是这种表示方法，其中：

- 0167：这是段选择子，其数据保存在 CS 段选择器里。同一程序在不同系统环境下，此值可能不同，一般也不需要关心此值。
- 00401000：此处表示内存的虚拟地址（Virtual Address），一般来说，同一程序的同一条指令在不同系统环境下，此值相同。

（4）基地址（ImageBase）

文件执行时将被映射到指定内存地址中，这个初始内存地址称为基地址（ImageBase）。这个值是由 PE 文件本身设定的。按照默认设置，用 Visual C++建立的 EXE 文件基地址是 00400000h，DLL 文件基地址是 10000000h。但是，可以在创建应用程序的 EXE 文件时改变这个地址，方法是在链接应用时使用链接程序的/BASE 选项。

用 PE 编辑工具（如 LordPE）可以查看可执行的 PE 字段。单击 LordPE 的“PE Editor”打开任意一个可执行文件，如图 1.8 所示。

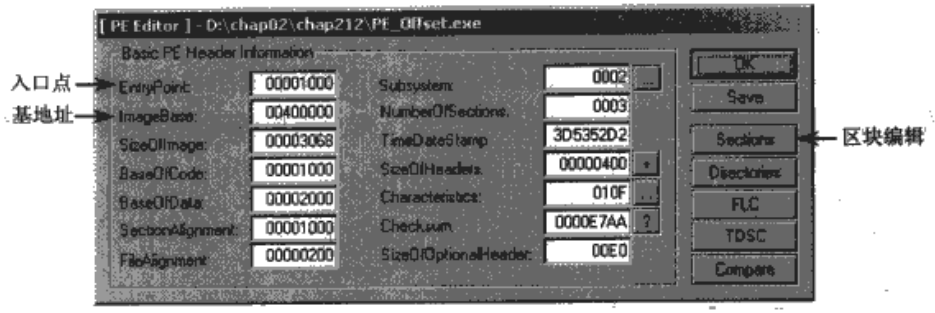


图 1.8　LordPE 中的 PE 编辑器

LordPE 功能非常强大，其显示的各项内容，学习第 10 章后就能理解。若要查看区块的信息，单击“Sections”按钮来打开区块编辑器（见图 1.9）。

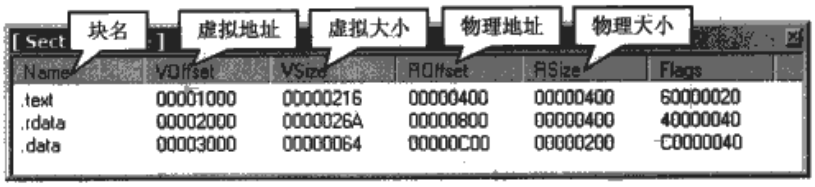
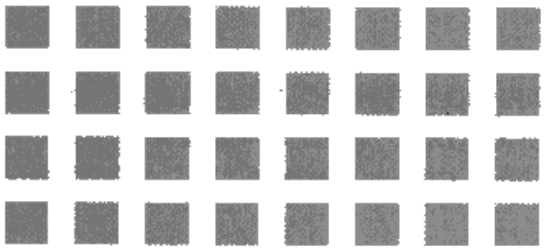


图 1.9　区块（Section）数据

在图 1.9 中显示出可执行文件在磁盘与内存中各区块的地址、大小等信息。虚拟地址和虚拟大小是指该区块在内存中的地址和大小。物理地址和物理大小是指该区块在磁盘文件中的地址和大小。



第2篇 调试篇

调试逆向是软件安全技术的基础。本篇以极大的篇幅，以动态分析与静态分析技术为主线，讲解了代码的逆向分析技巧，同时介绍了逆向工程必备工具OllyDbg、SoftICE、IDA的操作技巧等。

第2章 动态分析技术

动态分析技术中最重要的工具是调试器，分为用户模式和内核模式两种类型。用户模式调试器是指用来调试用户模式应用程序的调试器，它们工作在 Ring 3 级，如 OllyDbg、Visual C++等编译器自带的调试器。内核模式调试器是指能调试操作系统内核的调试器，它们处于 CPU 和操作系统之间，工作在 Ring 0 级，如 SoftICE 等。

2.1 OllyDbg 调试器

OllyDbg（简称 OD）是由 Oleh Yuschuk（www.ollydbg.de）编写的一款具有可视化界面的用户模式调试器，可以在当前各种 Windows 版本上运行，但 NT 的系统架构更能发挥 OllyDbg 强大功能。OllyDbg 结合了动态调试和静态分析，具有 GUI 界面，非常容易上手，并且对异常的跟踪处理相当灵活，这些特性使得 OllyDbg 成为调试 Ring 3 级程序的首选工具。它的反汇编引擎很强大，可识别数千个被 C 和 Windows 频繁使用的函数，并能将其参数注释出。它会自动分析函数过程、循环语句、代码中的字符串等。此外，开放式的设计给了这个软件很强的生命力，爱好者不断地修改、扩充 OllyDbg，脚本执行能力和开放插件接口使得其变得越来越强大。

2.1.1 OllyDbg 界面

本节以 OllyDbg 1.10 讲述其用法，支持 32 位程序。OllyDbg 发行版本是一个 ZIP 压缩包，只要将其解压缩到一个目录下，然后运行 ollydbg.exe 即可。打开目标程序后，OllyDbg 会打开多个子窗口，单击菜单 View 或单击工具栏标签 L、E、M 等按钮可在各子窗口间切换，如图 2.1 所示。这些按钮依次对应 Log 窗口、Executable modules 窗口、Memory 窗口、Threads 窗口、Windows 窗口、Handles 窗口、CPU 窗口、Patches 窗口、Call stack 窗口、Breakpoints 窗口、References 窗口、Run trace 窗口、Source 窗口等。更多窗口请查看 View 菜单，各窗口功能请参考 OllyDbg 帮助文档的描述。

图 2.1 窗口切换面板

默认的当前窗口是 CPU 窗口，它在 OllyDbg 中是最重要的窗口。调试程序的绝大部分操作都要在这个窗口中进行。它包括以下 5 个面板窗口：反汇编面板、寄存器面板、信息面板、数据面板、堆栈面板，如图 2.2 所示。

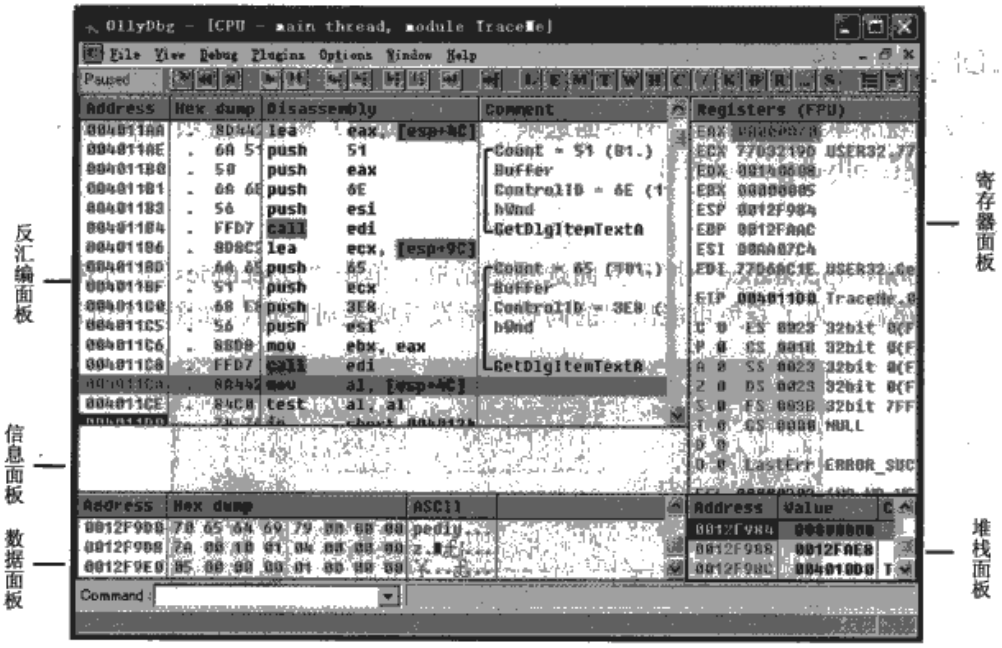


图 2.2 OllyDbg 主界面

各窗口的外观属性，如标题栏（bar）、字体（font）等在右键菜单“Appearance”（界面选项）里控制。

1. 反汇编面板窗口（Disassembler window）

反汇编面板窗口显示被调试程序的代码。它有 4 个列：地址（Address）、机器码（Hex dump）、反汇编代码（Disassembly）和注释（Comment）。最后一列注释栏显示相关 API 参数或运行简表，非常有用。

在反汇编面板窗口的列中（注：不是列标题），默认时，双击完成以下动作。

- Address 列：显示相对被单击地址的地址，再次双击返回到标准地址模式；
- Hex dump 列：设置或取消无条件断点，对应的快捷键是 F2 键；
- Disassembly 列：调用汇编器，可直接修改汇编代码；
- Comment 列：允许增加或编辑注释，对应的快捷键是“;”键。

要从键盘上选择多行，按下 Shift 键和上下光标箭头或者 PgUp / PgDn 键，也可利用右键菜单命令。按 Ctrl 键并按上下光标箭，一行一行地滚动汇编窗口（当数据与代码混合时，此功能非常有用）。

2. 信息面板窗口（Information window）

动态跟踪时，显示与指令相关的各寄存器值、API 函数调用提示和跳转提示等信息。

3. 数据面板窗口（Dump window）

以十六进制和字符方式显示文件在内存中的数据。要显示数据，可单击鼠标右键“Go to expression”命令或按“Ctrl+G”键打开地址窗口，输入地址。

4. 寄存器面板窗口（Registers window）

显示 CPU 各寄存器的值，支持浮点、MMX 和 3DNow!寄存器，可以单击鼠标右键切换或单击窗口标题切换显示寄存器的方式。

5. 堆栈面板窗口（Stack window）

显示了堆栈的内容，即 ESP 指向地址的内容。堆栈窗口非常重要，各 API 函数和子程序等都利用它传递参数和变量等。

2.1.2 OllyDbg 的配置

OllyDbg 的设置在菜单 Options 里，有界面选项（Appearance）和调试选项（Debugging options）等。这些选项配置都保存在 ollydbg.ini 文件里。

1. 界面设置

单击菜单“Options/Appearance”打开界面选项对话框，单击“Directories”（目录）标签，这里设置 UDD 文件和插件的路径，为了避免可能出现的问题，请设置成绝对路径，如图 2.3 所示。

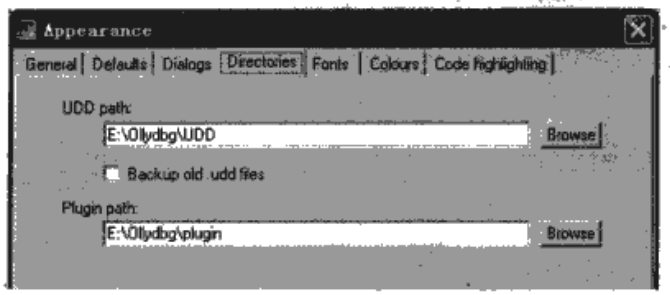


图 2.3 UDD 文件及插件路径设置

UDD 文件是 OllyDbg 的工程文件，用以保存当前调试的一些状态，如断点、注释等，以便下次调试时继续使用。

插件用以扩充功能，路径设置正确后，将插件复制到这个目录，在 OllyDbg 的主菜单里就会出现“Plugin”（插件）这个菜单项。

OllyDbg 界面外观完全可以定制，这些外观由 Appearance 选项里的 Fonts, Colours, Code highlighting 控制。

颜色（Colours）：这里是 OllyDbg 颜色的主题，可以根据自己的喜欢设置。设置时，先选择合适的颜色主题（Colour scheme），例如本例是“Black on white”，如图 2.4 所示，然后进行配色。主题配置好后，就可在其他窗口应用这个主题，如在 CPU 窗口单击右键，选择菜单“Appearance/Colours/ Black on white”，这样新的配置才会生效。

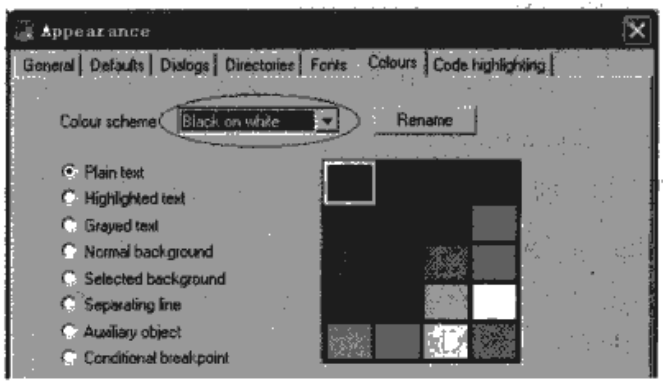


图 2.4 OllyDbg 界面颜色设置

代码高亮显示（Code highlighting）：此选项主要作用于 CPU 窗口，可设置相关代码的颜色，提高代码可读性。操作与颜色设置一样。

2. 调试设置

单击菜单“Options/Debugging options”打开调试设置选项对话框，一般按默认设置即可。其中异常（Exceptions），可以设置让 OllyDbg 忽略或不忽略那些异常，现在建议全部选上，如图 2.5 所示。有关异常的知识后面章节会讲解。

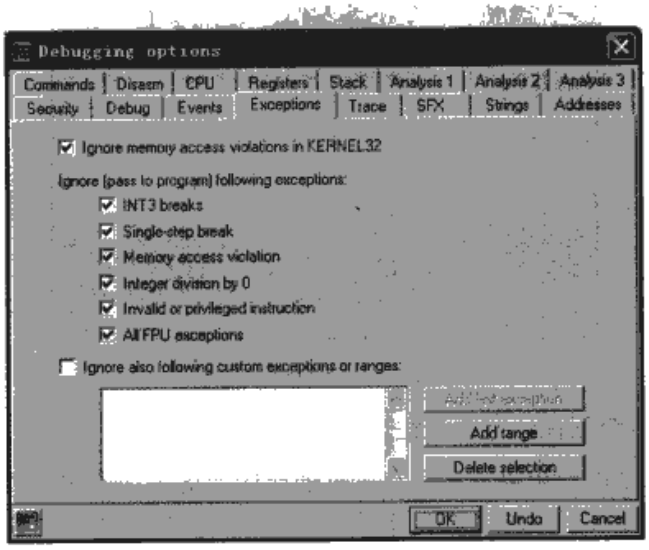


图 2.5 调试选项中的异常设置

3. 加载符号文件

这个功能类似 IDA 的 FLIRT，使用符号库（Lib），可以让 OllyDbg 以函数名显示 DLL 中的函数。例如 MFC42.DLL 是以序号输出函数的，这时在 OllyDbg 显示的是序号，如果让其加载 MFC42.DLL 调试符号，则以函数名显示相关输出函数。加载方法是单击菜单“Debug/Select import libraries”来打开导入库窗口，如图 2.6 所示。

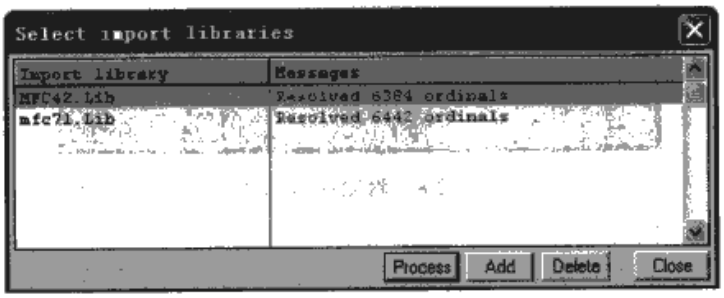


图 2.6 加载调试符号库

4. 关联到右键菜单

可以将 OllyDbg 关联到资源管理器右键菜单里，调试程序时，只需要在 EXE 或 DLL 文件上单击右键，就会出现“Open with Ollydbg”菜单。要实现关联只需要单击菜单“Options/Add to Explorer”，再单击“Add OllyDbg to menu in Windows Explorer”按钮即可关联。

2.1.3 加载程序

OllyDbg 可以用两种方式加载目标程序调试，一种是通过 CreateProcess 创建进程；另一种是利用 DebugActiveProcess 函数将调试器捆绑到一个正在运行的进程上。

1. 利用 CreateProcess 创建进程

单击菜单“File/Open”或按快捷键 F3 打开目标文件，这样会调用 CreateProcess 创建一个用以调试的新进程。OllyDbg 将接收到目标进程发生的调试事件，而对其子进程的调试事件将不予理睬。

OllyDbg 除了直接加载目标程序外，也支持带参数的程序，方法是：在打开对话框中的“Arguments”栏中输入参数行，如图 2.7 所示。

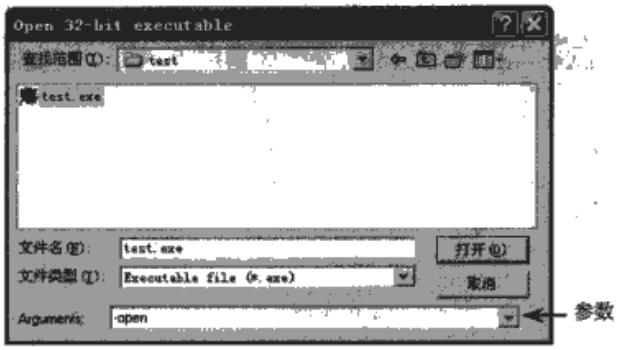


图 2.7　带参数调试程序

2. 将 OllyDbg 附加到一个正在运行的进程上

OllyDbg 的一个实用的功能是可以调试正在运行的程序，这个功能称为“附加（Attach）”。其原理是利用 DebugActiveProcess 函数可以将调试器捆绑到一个正在运行的进程上，如果执行成功，则效果类似于利用 CreateProcess 创建的新进程。

单击菜单“File/Attach”打开附加对话框，如图 2.8 所示。选中正在运行的目标进程，单击 Attach 按钮即可附加目标进程。附加后，目标程序会暂停在 Ntdll.dll 的 DbgBreakPoint 处，在 OllyDbg 里按一下 F9 键或 Shift+F9 键让程序继续运行。接着就可对目标程序进行调试分析了。

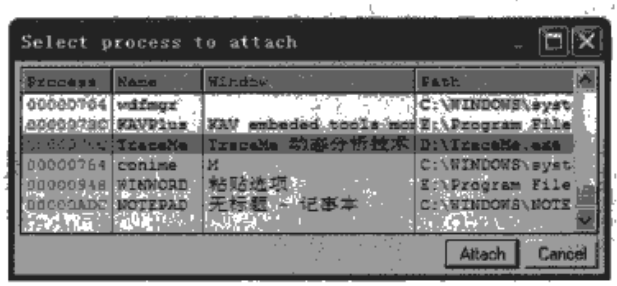


图 2.8　附加目标进程

注意：附加一个程序时，尽量用新打开的 OllyDbg，这样附加的成功率高些。

如果是隐藏进程，就不能用上述方法附加了。OllyDbg 有一个-p 启动参数，只要得到进程的 pid 就可以附加了。用 IceSword 等工具获得隐藏进程的 pid，然后在控制台窗口用-p 参数附加即可。注意，pid 的值是十进制。

```
C:\OllyDbg.exe -p pid值
```

如果附加不成功，可以巧妙利用 OllyDbg 的即时调试器功能来调试。先看一个例子，运行 A.exe，其会调用 B.exe，此时用 OllyDbg 附加 B.exe，OllyDbg 会无响应。解决办法：在“Options/Just-in-time debugging”中设置 OllyDbg 为即时调试器，将 B.exe 的入口改成 CC，即 INT 3 指令，同时记下原指令。运行 A.exe，其调用 B.exe，运行到 INT 3 指令会导致异常，OllyDbg 会作为即时调试器启动并加载 B.exe，此时再将 INT 3 指令恢复原指令，继续调试。

2.1.4　基本操作

对于习惯 Borland 开发环境的朋友来说，用 OllyDbg 比较容易上手，例如单步功能是用 F7 键和 F8 键等，这与 Borland 的产品习惯完全一样。

在这里，以一个用 Visual C++ 6.0 编译的程序 TraceMe 来讲解 OllyDbg 的操作，编译时优化选项按默

认设置为“Maximize speed”。读者也可以按“Minimize Size”优化选项编译一下，并与本文比较。因为优化选项不同，生成的汇编代码也会有所不同。

1. 准备工作

拆解一个 Windows 程序要比拆解一个 DOS 程序容易得多，因为在 Windows 中，只要 API 函数被使用，想对寻找蛛丝马迹的人隐藏一些东西是比较困难的。因此分析一个程序，用什么 API 函数作为切入点就显得比较关键了，如果有些编程经验，这方面就更得心应手了。

为了便于理解，先简单地看一下 TraceMe 的序列号验证流程，如图 2.9 所示。将姓名与序列号输入到文字框中，程序调用 GetDlgItemTextA 函数把字符读出来，然后进行计算，最后用函数 lstrcmp 进行比较。因此，这些调用的函数就是解密跟踪的目标，用这些函数作为断点，跟踪程序的序列号验证过程就能找出正确的序列号。

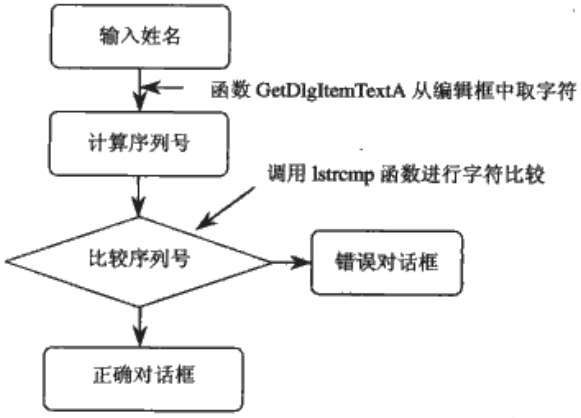


图 2.9 TraceMe 程序序列号验证过程

2. 加载目标文件调试

为了能让 OllyDbg 中断在程序的入口点，加载程序前必须要设置一下。运行 OllyDbg 后，单击菜单“Options/Debugging options”，打开调试选项配置对话框，再单击“Event”标签，如图 2.10 所示。这里设置 OllyDbg 对中断入口点、模块加载/卸载、线程创建结束等事件的处理，一般调试只需要将暂停点设置在“Entry point of main module”或“WinMain”即可。

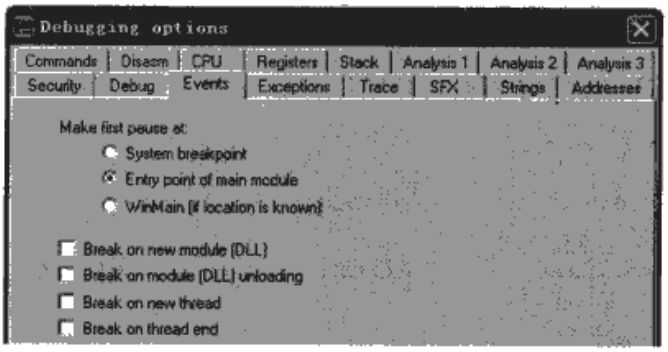


图 2.10 设置 OllyDbg 第一次暂停

- System breakpoint：系统断点，OllyDbg 用 CreateProcessA 加载 DEBUG_ONLY_THIS_PROCESS 参数执行，程序运行之后会触发一个 INT 3，在系统空间里。
- Entry point of main module：主模块的入口点，即文件的入口点。
- WinMain：程序的 WinMain()函数入口点，即使设置这个选项，OllyDbg 一般也只会中断在文件入口点处。

设置好后，单击菜单“File/Open”打开 TraceMe.exe，此时 OllyDbg 会中断在 TraceMe 的入口点，如图 2.11 所示。光条停在 4013A0 这行，这个 4013A0 就是程序开始执行的入口地址（EntryPoint）。

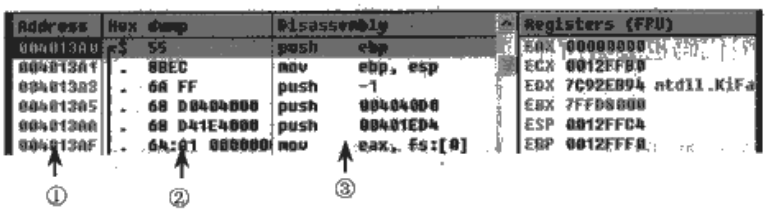


图 2.11　OllyDbg 加载目标程序停在入口点

在图 2.11 中，代码中各部分含义如下：

- ① 虚拟地址：一般情况下，同一程序的同一条指令在不同系统环境下此值相同；
- ② 机器码：这就是 CPU 执行的机器代码；
- ③ 汇编指令：和机器码对应的程序代码。

3. 单步跟踪

调试器的一个最基本功能就是动态跟踪，OllyDbg 在菜单“Debug”里控制运行的命令，各个菜单项都有相应的快捷键。OllyDbg 的单步跟踪功能键如表 2-1 所示。

表 2-1　OllyDbg 的单步跟踪功能键

OllyDbg 功能键	功　能
F7	单步步进，遇到 CALL 跟进
F8	单步步过，遇到 CALL 跳过，不跟进
Ctrl+F9	直到出现 RET 指定时中断
Alt+F9	若进入系统领空，此命令可瞬间回到应用程序领空
F9	运行程序

F8 键在调试中用得很频繁，可以一句句地单步执行汇编指令，遇到 CALL 指令不会跟进，而路过。例如：

```
004013F7  xor    esi, esi
004013F9  push   esi
004013FA  call   00401DA0              ;按 F8 键不会进去，而直接路过这个 CALL
004013FF  pop    ecx
00401400  test   eax, eax
```

F7 和 F8 功能键的主要差别就在于若遇到 CALL、LOOP 等指令，F8 键是路过，而 F7 键是跟进去。

```
004013F7  xor    esi, esi
004013F9  push   esi
004013FA  call   00401DA0              ;按 F7 键会进入这个 CALL
{
   00401DA0  xor    eax, eax           ;上面那句 4013FA，按 F7 键就会来到这里
   00401DA2  push   0
   00401DA4  cmp    [esp+8], eax
   00401DA8  push   1000
   00401DAD  sete   al
   ......
}
```

当要重复按多次 F7 键或 F8 键时，OllyDbg 提供了“Ctrl+F7”和“Ctrl+F8”快捷键，直到用户按 Esc 键、F12 键或遇到其他断点时停止。

当位于某个 CALL 中，这时想返回到调用这个 CALL 的地方时，可以按“Ctrl+F9”快捷键执行“执行到返回（Execute till return）”功能。OllyDbg 就会停在遇到的第一个返回命令（RET、RETF 或者 IRET），

这样可以很方便地略过一些没用的代码。例如上面的代码，在 401DA0 这行，如果按“Ctrl+F9”快捷键就会回到 4013FA 这句。遇到 RET 指令是暂停还是步过可以在选项里设置，方法是：打开调试设置选项对话框，在“Trace”页面，设置“After Execting till RET,step over RET（执行到 RET 后，单步步过 RET）”。

如果跟进系统 DLL 提供的 API 函数中，此时想返回到应用程序领空里，可以按快捷键“Alt+F9”执行“Execute till user code（执行到用户代码）”命令。例如：

```
004013C0  push    ebx
004013C1  push    esi
004013C2  push    edi
004013C3  mov     [ebp-18], esp
004013C6  call    [<&KERNEL32.GetVersion>]     ;按 F7 键跟进 KERNEL32.dll 里
004013CC  xor     edx, edx
```

在上面的 4013C6 一行，按 F7 键就可跟进系统 KERNEL32.DLL 里的领空：

```
7C8114AB kernel32.GetVersion mov     eax, fs:[18]
7C8114B1                     mov     ecx, [eax+30]     ;假设当前光标在这行
7C8114B4                     mov     eax, [ecx+B0]
7C8114BA                     movzx   edx, word ptr [ecx+AC]
7C8114C1                     xor     eax, FFFFFFFE
```

像地址 7C8114AB 等都是系统 DLL 所在的地址空间，这时只要按一下快捷键“Alt+F9”就可回到应用程序领空里。代码如下：

```
004013C0   push    ebx
004013C1   push    esi
004013C2   push    edi
004013C3   mov     [ebp-18], esp
004013C6   call    [<&KERNEL32.GetVersion>]
004013CC   xor     edx, edx                    ;会返回到此行
```

注意：所谓领空，实际上是指在某一时刻，CPU 的 CS:EIP 所指向的某段代码的所有者。

如果不想单步跟踪，让程序直接运行起来，可以按 F9 键或单击工具栏中的 ▶ 按钮。如果想重新调试目标程序，可以按“Ctrl+F2”快捷键或单击工具栏中的 ⏮ 按钮，Ollydbg 结束被调试进程并重新加载它。有时程序进入死循环，可以按 F12 键暂停程序。

4. 设置断点

断点是调试器的一个重要功能，可以让程序中断在需要的地方，从而方便对其分析。最常用的断点是 INT 3 断点，其原理是 Ollydbg 将断点地址处的代码修改为 INT 3 指令。在图 2.12 中，将光标移动到 4013A5 一行，按 F2 键即可设置一个断点，再按一次 F2 键取消断点。也可以用鼠标双击“Hex dump”列中相应的行设置断点，再次双击取消断点。当关闭程序时，OllyDbg 会自动将当前应用程序的断点位置保存在其安装目录*.udd 文件中，以便下次运行时，这些断点继续有效。如果将断点设置到当前应用程序代码外，OllyDbg 将会警告。可以在菜单“Options/Debugging options/Security”选项里将“Warn when breakpoint is outside the code section”取消选中，以关闭这个警告。

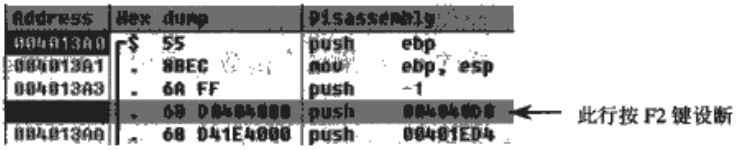


图 2.12 设置断点

现在开始一个完整的调试分析过程，取消开始设置的所有断点，在 OllyDbg 里按 F9 键将实例 TraceMe.exe 运行起来，如图 2.13 所示。

图 2.13　实例运行起来的界面

字符通常利用 Windows 文本框输入。为了检查输入的字符，程序常采用下面这些函数把文本框中的内容读出来，如表 2-2 所示。

表 2-2　程序采用的将文本框中内容读出来的函数

16 位	32 位（ANSI 版）	32 位（Unicode 版）
GetDlgItemText	GetDlgItemTextA	GetDlgItemTextW
GetWindowText	GetWindowTextA	GetWindowTextW

一般事先不会知道程序具体是调用了什么函数来处理字符的，只好多试几遍，找出相关的函数。

首先，需要在 OllyDbg 中设定一个“陷阱”（或称断点）。因为这个 TraceMe 是 32 位 ANSI 版的程序，所以在 GetDlgItemTextA 处设一个断点。按“Ctrl+G”键打开跟随表达式的窗口，输入 GetDlgItemTextA 字符，如图 2.14 所示。

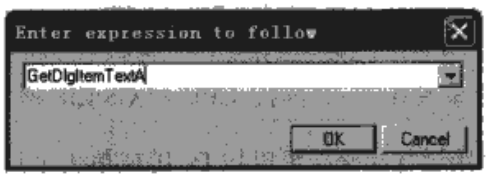


图 2.14　打开跟随表达式窗口

注意：OllyDbg 里对 API 的大小写敏感，输入的函数名大小写必须正确。

单击 OK 按钮后，会来到系统 USER32.DLL 中的 GetDlgItemTextA 函数入口处，如图 2.15 所示。

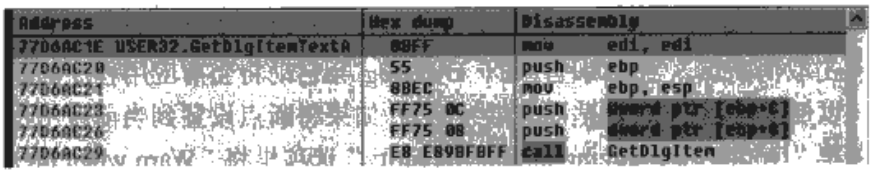


图 2.15　跳到函数入口处

在 77D6AC1E 这一行，按 F2 键设个断点，即在 GetDlgItemTextA 函数入口处设了断点（操作系统版本不同，这个函数入口地址是不一样的），如果这个函数被调用，OllyDbg 就会中断。

注意：在 Windows 9x 系统中，OllyDbg 是无法对 API 函数入口点下断的，因此不能用此方法设断。

下一步可以列出所有断点来检查一下，按“Alt+B”快捷键或单击按钮打开断点窗口，如图 2.16 所示。

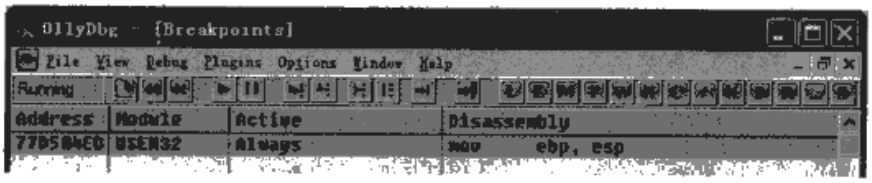


图 2.16 断点窗口

这里可以显示除硬件断点外的其他断点，其中“Always”表示断点处于激活状态，“Disable”表示断点停用，按空格键可切换其状态，也可以用鼠标右键菜单管理这些断点。

现已经设定了断点，可以捕捉任何对 GetDlgItemTextA 函数的调用。然后输入姓名和序列号，如姓名为“pediy”，序列号为“1212”。单击“Check”按钮，程序中断在 OllyDbg 中，就在函数 GetDlgItemTextA 开始的地方。

也可通过输入表设置断点。在 OllyDbg 里，按“Ctrl+N”键打开应用程序的输入表，会发现 USER32.GetDlgItemTextA 函数，在这个函数上按 Enter 键或右键菜单执行“Find references to import”命令打开调用此函数的参考代码窗口，找到相应的代码，按 Enter 键即可切换到相应的代码，接下来按 F2 键设置断点。

5. 调试分析

按“Alt+F9”键，回到调用函数的地方，当然也可以按 F8 键单步走出 GetDlgItemTextA 这个函数。OllyDbg 非常强大，已将各函数的调用参数及当前值都注释出来了。相关代码如下：

```
004011AE  push    51              ; /Count = 51 (81.)
004011B0  push    eax             ; |Buffer
004011B1  push    6E              ; |ControlID = 6E (110.)
004011B3  push    esi             ; |hWnd
004011B4  call    edi             ; \GetDlgItemTextA
004011B6  lea     ecx, [esp+9C]   ; 从GetDlgItemTextA函数里出来会回到这行
004011BD  push    65              ; /Count = 65 (101.)
004011BF  push    ecx             ; |Buffer
004011C0  push    3E8             ; |ControlID = 3E8 (1000.)
004011C5  push    esi             ; |hWnd
004011C6  mov     ebx, eax        ; |
004011C8  call    edi             ; \GetDlgItemTextA
```

来到 TraceMe 领空后，可以按“Alt+B”键打开断点窗口，将 GetDlgItemTextA 处的断点禁止。

很多时候必须重复跟踪同一段代码，因此可以先设置一个断点。将光标移到 4011AE 一行，按 F2 键设置新的断点，以方便反复跟踪调试。

中断后的代码如下（可结合源码阅读）：

```
;len=GetDlgItemText(hDlg,IDC_TXT0,cName,sizeof(cName)/sizeof(TCHAR)+1)
004011AA    lea eax, [esp+4C]
004011AE    push 00000051              ; 参数：最大字符数
004011B0    push eax                   ; 参数：文本缓冲区指针
004011B1    push 0000006E              ; 参数：控件标识（ID号），见resource.h
004011B3    push esi                   ; 参数：对话框句柄
004011B4    call edi                   ; 调用GetDlgItemTextA函数取姓名
004011B6    lea ecx, [esp+9C]          ; 上句执行后，姓名长度返回到eax中
;-------------------------------------------
;GetDlgItemText(hDlg,IDC_TXT1,cCode,sizeof(cCode)/sizeof(TCHAR)+1)
```

```
004011BD    push 00000065              ; 最大字符数
004011BF    push ecx                   ; 文本缓冲区指针
004011C0    push 000003E8              ; 控件标识（ID 号）
004011C5    push esi                   ; 对话框句柄
004011C6    mov ebx, eax               ; 将用户名的长度转到 ebx 中
004011C8    call edi                   ; 调用 GetDlgItemTextA 函数取序列号
;--------------------------------------
; if (cName[0] == 0 || len<5)
004011CA    mov al, byte ptr [esp+4C]; 将用户名第一个字节给 al
004011CE    test al, al                ; 检查有没有输入用户名
004011D0    je 00401248                ; 如果没有输入用户名跳走，告知输入字符太少
004011D2    cmp ebx, 00000005          ; 用户名长度是<5
004011D5    jl 00401248
;--------------------------------------
;GenRegCode(cCode, cName ,len)  (GenRegCode 子程序采用 C 调用约定)
004011D7    lea edx, [esp+4C]          ; 用户名地址放到 edx 中
004011DB    push ebx                   ; 用户名长度入栈（len 参数）
004011DC    lea eax, [esp+000000A0]  ; 序列号地址放到 eax 中
004011E3    push edx                   ; 用户名入栈（cName 参数）
004011E4    push eax                   ; 序列号入栈（cCode 参数）
004011E5    call 00401340              ; 这个 CALL 就是 GenRegCode 函数
004011EA    mov edi, [004040BC]
004011F0    add esp, 0000000C          ; 平衡堆栈（C 调用约定）
004011F3    test eax, eax              ; eax=0 注册失败，eax=1 注册成功
004011F5    je 0040122E
```

在阅读这些代码时：

- 要搞清各 API 函数的定义（查看相关 API 手册）。
- API 函数基本采用的是__stdcall 调用约定，即函数入口参数按从右到左的顺序入栈，并由被调用者清理栈中参数，返回值放在 eax 寄存器中。因此，对相关的 API 函数要分析其前的 push 指令，这些指令将参数放进堆栈以传送给 API 调用。整个跟踪过程中要关注堆栈数据变化。
- C 代码中的子程序采用的是 C 调用约定，函数入口参数按从右到左的顺序入栈，由调用者清理栈中的参数。

有关调用约定、参数传递等知识，可以从本书第 4 章获得。阅读上面代码时，需理解的 GetDlgItemTextA 函数原型如下：

```
UINT GetDlgItemText(
    HWND hDlg,                          // 对话框句柄
    int nIDDlgItem,                     // 控件标识（ID 号）
    LPTSTR lpString,                  // 文本缓冲区指针
    int nMaxCount                       // 最大字符数
);
```

GetDlgItemText 采用标准调用约定，参数按从右到左的顺序入栈。例如本例中的汇编代码：

```
004011AE  push    51               ; int nMaxCount
004011B0  push    eax              ; LPTSTR lpString,
004011B1  push    6E               ; int nIDDlgItem
004011B3  push    esi              ; HWND hDlg
004011B4  call    GetDlgItemTextA
```

当 GetWindowText 函数执行后，将把取出的文本放到由 lpString（LPTSTR 是一个长的指针，指向由空字符终止的字符串）指定的位置。如想看到输入的字符串，跟踪的时候，在 4011B0 一行停住，在 eax 寄存器单击右键，执行菜单“Follow in Dump”命令查看数据窗口中的内容，当然此时数据窗口中没什么有价值的东西。继续按 F8 键单步执行完下面一句：

```
004011B4   call edi              ; GetDlgItemTextA 函数取姓名
```

此时 GetDlgItemTextA 函数已将字符串取出，放到 eax 所指的地址里。数据窗口右边字符段显示出刚输入的字符“pediy”，如图 2.17 所示。

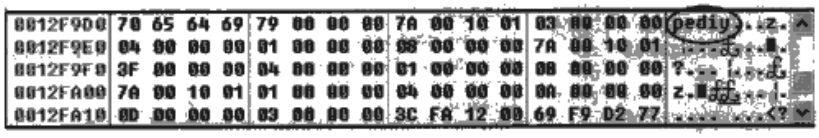


图 2.17 数据窗口查看字符

6. 保存修改后的文件

在上一节中，已找到序列号的判断核心，这里的一段代码是关键：

```
004011E5   call  00401340              ; 序列号计算的 CALL
004011EA   mov   edi, [<&USER32.GetDlgItem>]
004011F0   add   esp, 0C
004011F3   test  eax, eax              ; eax=0 注册失败，eax=1 注册成功
004011F5   je    short 0040122E        ; 不跳转则成功
```

只要 4011F5 一句不跳转即可注册成功。在调试过程中，当执行到 4011F5 一句时，有几种方法可以验证判断。

- 在 OllyDbg 寄存器面板，用鼠标单击标志寄存器 ZF（即“Z”），单击一次 ZF 的值取反，如果原来是 1，执行后为 0，如图 2.18 所示。

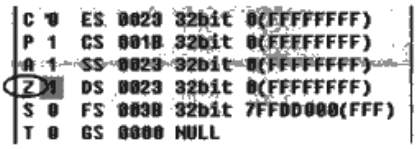


图 2.18 改变标志寄存器的值

- 在 4011F5 一行，双击鼠标或单击空格键，输入指令：“NOP”，这个指令机器码是 90，如图 2.19 所示，此处用“9090”取代“7437”。

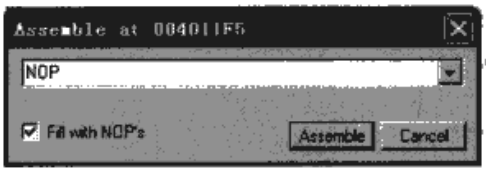


图 2.19 键入汇编代码

修改好后的代码如下：

```
004011E5   E8 56010000        call   00401340
004011EA   8B3D BC404000      mov    edi, [<&USER32.GetDlgItem>]
004011F0   83C4 0C            add    esp, 0C
004011F3   85C0               test   eax, eax
004011F5   90                 nop                    ; 这里
004011F6   90                 nop                    ; 这里
```

现在随意输入姓名与序列号，这个 CrackMe 都会提示注册成功。目前修改的是内存中数据，为了使修改一直有效，就必须将这个变化写进磁盘文件中去。OllyDbg 也提供了这个功能，方法是用鼠标选中修改过的代码，单击鼠标右键，执行“Copy to executable/Selection”命令，如图 2.20 所示。

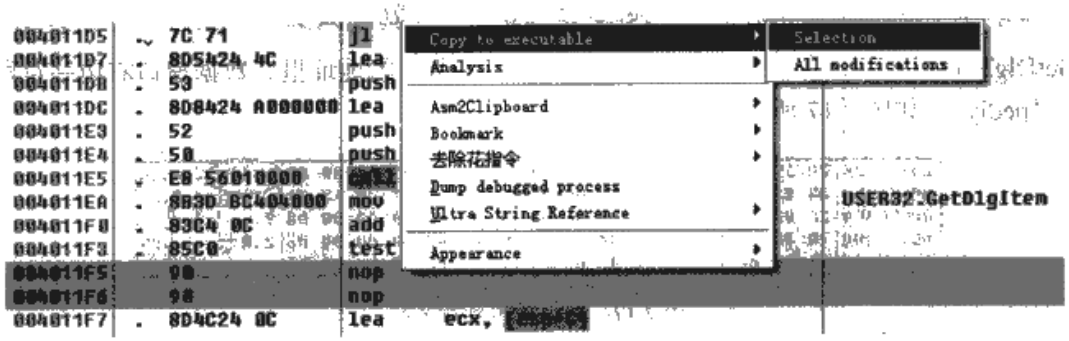


图 2.20　保存修改的文件

执行复制到可执行文件的命令后，将打开文件编辑窗口，如图 2.21 所示。单击鼠标右键，执行命令“Save File”即可将修改保存到文件。像这种屏蔽程序的某些功能或改变程序流程，使程序的保护方式失效的方法称为 patch（补丁）或“爆破”。

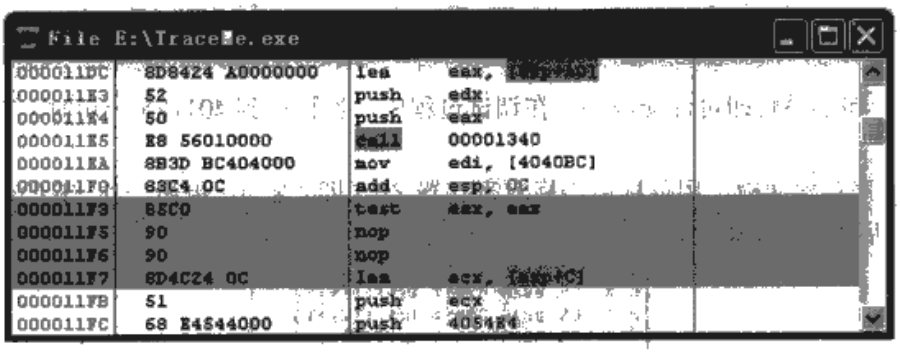


图 2.21　文件编辑器

7．算法分析

接下来，分析序列号算法的原理，通常这个过程比较复杂。序列号核心计算部分的高级语言（C 语言）形式如下：

```
unsigned char Table[8] = {0xC,0xA,0x13,0x9,0xC,0xB,0xA,0x8}; // 数据表，全局变量
BOOL GenRegCode( TCHAR  *rCode,TCHAR  *name,int len)
{
    int i,j;
    unsigned long code=0;
    for(i=3,j=0;i<len;i++,j++) {
            if(j>7) j=0;
            code+=((BYTE)name[i])*Table[j];
        }
        wsprintf(name,TEXT("%ld"),code);
        if(lstrcmp(rCode, name)==0)
            return true;
        else
            return false;
}
```

下面代码是调用 GenRegCode()函数，在 4011E5 一行，按 F7 键进入这个函数，同时，注意堆栈窗口的数据变化。

```
004011DB  push    ebx                 ; int len
004011DC  lea     eax, [esp+A0]
```

```
0040i1E3  push    edx                ; TCHAR  *name
004011E4  push    eax                ; TCHAR  *rCode
004011E5  call    00401340           ; GenRegCode ()
```

进入子程序 call 00401340 后，子程序初始化堆栈，此时堆栈情况如图 2.22 所示。

图 2.22 堆栈面板窗口

程序在此利用 esp 来访问各参数，ebp 用来处理字符串 name[i]。详细过程如下：

```
00401340     push ebp                    ; ebp 入栈，保护现场
00401341     mov ebp, [esp+0C]           ; 将参数从栈中传给 ebp（用户名 cName 指针）
00401345     push esi                    ; esi 入栈，保护现场
00401346     push edi                    ; edi 入栈，保护现场
;-----------------------------------------
;for(i=3,j=0;i<len;i++,j++)
00401347     mov edi, [esp+18]           ; 将参数从堆栈中传给 edi（len 参数的值）
0040134B     mov ecx, 00000003           ; i=3，ecx 作为变量 i 使用
00401350     xor esi, esi                ; code=0
00401352     xor eax, eax                ; j=0，eax 作为变量 j 使用
00401354     cmp edi, ecx                ; i<len 吗
00401356     jle 00401379
00401358     push ebx                    ; 注意这句与 401378 行呼应
;if(j>7) j=0
00401359     / cmp eax, 00000007         ; j>7 吗
0040135C     | jle 00401360
0040135E     | xor eax, eax              ; j=0
;code+=((BYTE)name[i])*Table[j]          ; (BYTE) 是防止处理中文时符号扩展
00401360     | xor edx, edx              ; 清 0
00401362     | xor ebx, ebx
00401364     | mov dl, [ecx+ebp]         ; name[i]
00401367     | mov bl, [eax+405030]      ; Table[j]，00405030 地址处放的是数据表
0040136D     | imul edx, ebx             ; edx= name[i]*Table[j]
00401370     | add esi, edx              ; code+= edx
00401372     | inc ecx                   ; i++
00401373     | inc eax                   ; j++
00401374     | cmp ecx, edi              ; i<len 吗
00401376     \ jl 00401359               ; 如小于，则循环，计算下一位
00401378     pop ebx                     ; ebx 出栈
;-----------------------------------------
; wsprintf(name,TEXT("%ld"),code)
00401379     push esi                    ; code
0040137A     push 00405078               ; "%ld"
0040137F     push ebp                    ; name
00401380     Call [0040409C]             ; wsprintfA 函数将数字转换成字符
00401386     mov eax, [esp+1C]           ; 将参数从堆中传给 eax（序列号 cCode 指针）
0040138A     add esp, 0000000C           ; wsprintf 是唯一一个需手动平衡堆栈 API 函数
0040138D     push ebp                    ; ebp 指向的是计算出的真正序列号
0040138E     push eax                    ; eax 指向的是输入的序列号
0040138F     Call [00404004]             ; lstrcmp 函数比较字符
00401395     neg eax                     ; 相等则 eax=0
00401397     sbb eax, eax
```

```
00401399    pop edi                          ; edi 出栈，恢复现场
0040139A    pop esi
0040139B    inc eax                          ; eax+1，即序列号相等 eax=1，否则 eax=0
0040139C    pop ebp
0040139D    ret                              ; 子程序的返回值通过 eax 寄存器返回
```

计算序列号用的数据表可从这句指令中查到：

```
:00401367   mov bl, byte ptr [eax+00405030]
```

停在这一句，来到数据窗口，按“Ctrl+G”键输入地址：405030，查看数据窗口，如图 2.23 所示。

图 2.23　查看数据窗口

数据窗口显示的就是 Table 表，其值为：0C 0A 13 09 0C 0B 0A 08。

TraceMe 最后调用了函数 lstrcmp 来比较字符，它的原型是：

```
int lstrcmp(
    LPCTSTR  lpString1      // 第一个字符串地址
    LPCTSTR  lpString2      // 第二个字符串地址
   );
返回值：如相等返回零
```

调用代码如下：

```
0040138D      push ebp                        ;计算出的真正序列号
0040138E      push eax                        ;输入的序列号
0040138F      Call dword ptr [00404004]       ;lstrcmp 函数比较字符
```

因此执行到 40138F 一句时，堆栈窗口中就会显示出正确的序列号 2470。如图 2.24 所示，左边是数据窗口显示的数据，右边是堆栈窗口，直接将指向的字符串显示出来了。

图 2.24　数据窗口查看序列号字符

这样一个程序就分析完了，读者感兴趣，可以写出这段代码逆算法，写出注册机。

2.1.5　断点

常用的断点有 INT 3 断点、硬件断点、内存断点等。调试时，合理使用断点，能大大提高效率。

1. INT 3 断点

当执行一个 INT 3 断点时，该地址处的内容被调试器用 INT 3 指令替换了，此时 OllyDbg 将 INT 3 隐藏了，显示出来仍是下断前的指令，如图 2.12 按 F2 键下的断点。实际上，4013A5 处的指令 68 已被替换成 CC 了：

```
004013A5      CC D0404000
```

这个 INT 3 指令，其机器码是 CCh，也常称为 CC 指令。当被调试进程执行 INT 3 指令导致一个异常时，调试器就会捕捉这个异常从而停在断点处，然后将断点处的指令恢复成原来指令。当然，如果自己写调试器，也可用其他一些指令代替 INT 3 来触发异常。

用 INT 3 断点的好处是可以设置无数个断点，缺点是改变了原程序指令，容易被软件检测到。例如为了防范 API 被下断，一些软件会检测 API 的首地址是否为 CCh，以此来判断是否被下了断点。在这用 C

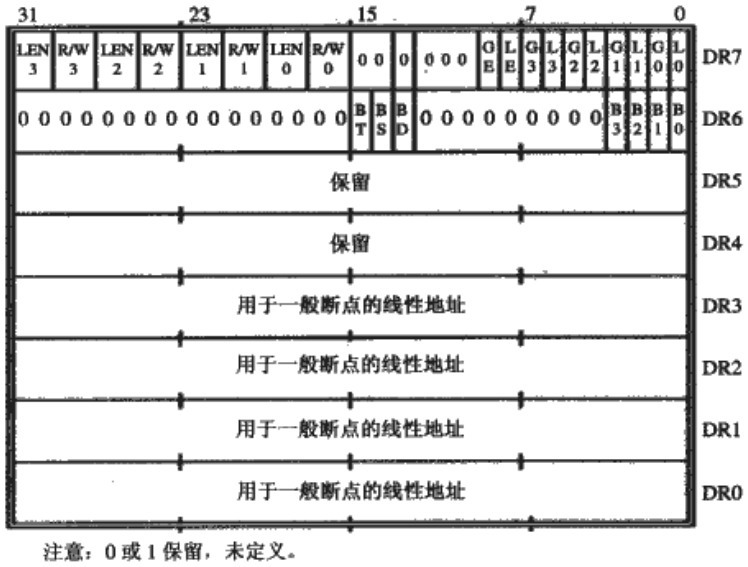
语言来实现这个检测，方法是取得检测函数的地址，然后读取它的第一个字节，判断它是否等于“CCh”。下面这段代码就是对 MessageBoxA 函数进行的断点检测：

```
FARPROC Uaddr ;
BYTE Mark = 0;
(FARPROC&) Uaddr =GetProcAddress ( LoadLibrary("user32.dll"), "MessageBoxA");
Mark = *((BYTE*)Uaddr);  // 取 MessageBoxA 函数第一字节
if(Mark ==0xCC)          // 如该字节为 CC，则认为 MessageBoxA 函数被下断
 return TRUE             // 发现断点
```

程序编译后，对 MessageBoxA 设断，程序将会发现自己被设断跟踪。当然躲过检测的方法是将断点下在函数内部或末尾，例如可以将断点下在函数入口的下一行，就可躲过检测了。

2. 硬件断点

硬件断点和 DRx 调试寄存器有关。从 Intel CPU 体系架构手册中，可以找到 DRx 调试寄存器的介绍，如图 2.25 所示。



注意：0 或 1 保留，未定义。

图 2.25 Intel 调试寄存器示意图

DRx 调试寄存器总共有 8 个，从 DR0 到 DR7。每个寄存器的特性如下：

- DR0～DR3：调试地址寄存器，保存需要监视的地址，如设置硬件断点；
- DR4～DR5：保留，未公开具体作用；
- DR6：调试寄存器组状态寄存器；
- DR7：调试寄存器组控制寄存器。

硬件断点原理是使用 4 个调试寄存器（DR0,DR1,DR2,DR3）来设定地址，以及 DR7 设定状态，因此最多只能设置 4 个断点。

OllyDbg 支持调试寄存器，其称为硬件断点。设断方法是在指定的代码行单击鼠标右键，执行“Breakpoint/Hardware,on execution（断点/硬件执行）”命令。

为了便于理解，这里演示一下。加载实例 TraceMe.exe，右键单击寄存器面板窗口，执行“View debug registers（查看调试寄存器）”，接着在 4013AA 这行设置硬件断点。按 F9 键执行程序，程序就会中断在 4013AA 这一行，查看调试寄存器，会发现 DR0 的值为 4013AA，如图 2.26 所示。

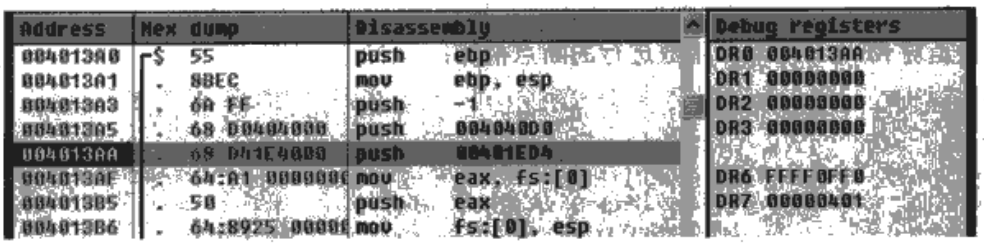


图 2.26　演示硬件断点

设置断点后，OllyDbg 实际上就是将 DR0～DR3 其中的一个设置为 44013AA，然后在 DR7 中设定相应的控制位。这样当被调试进程运行到 4013AA 时，CPU 就会给 OllyDbg 发送异常信息，OllyDbg 将该信息做初步处理后，中断下来，让用户继续进行操作。

硬件断点删除稍有些麻烦，单击菜单“Debug/Hardware breakpoints（调试/硬件断点）”，打开硬件断点面板，如图 2.27 所示，然后单击“Delete”按钮删除相应的硬件断点。

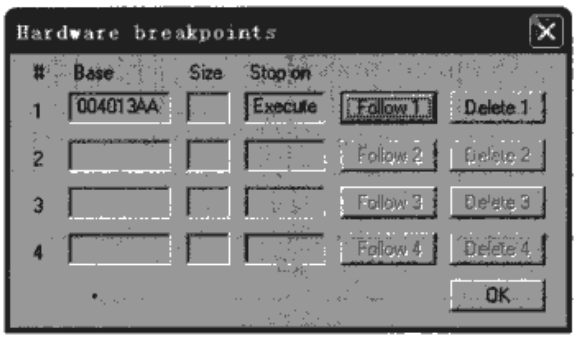


图 2.27　删除硬件断点

OllyDbg 提供了一个快捷键 F4，可以执行到光标所在的行，也是利用调试寄存器原理，中断后自动删除，相当于一次性硬件断点。

硬件断点优点是速度快，在 INT 3 断点容易被发现的地方，使用硬件断点来代替会有很好的效果；缺点就是最多能使用 4 个断点。

3．内存断点

OllyDbg 可以设置内存访问断点或内存写入断点，原理是对所设的地址设为不可访问/不可写属性，这样当访问/写入的时候就会产生异常，OllyDbg 截获异常后比较异常地址是不是断点地址，如果是就中断，让用户继续进行操作。

内存断点会降低 OllyDbg 速度，因为每次异常时都要通过比较来确定是否应该停下，也许 OllyDbg 可能在速度上做了考虑而只实现一个内存断点。

程序运行时会有 3 种状态：读取、写入、执行。

```
004013D0    mov   dword ptr  [405528], edx    ;对[405528]地址处的内存写入
004013D0    mov   dword ptr  edx,[405528]     ;对[405528]地址处的内存读取
```

用 OllyDbg 加载实例 TraceMe.exe，看到 4013D0 一行有一个写内存的指令：

```
004013D0  8915 28554000  mov   dword ptr [405528], edx
```

就用这个地址来演示一下如何下内存断点。在数据窗口对 405528 下内存写断点，方法是将光标移到 405528 地址处，选中需要下断点的地址区域，单击鼠标右键，执行“Breakpoint/Memory，on write（断点/内存写入）”，如图 2.28 所示。

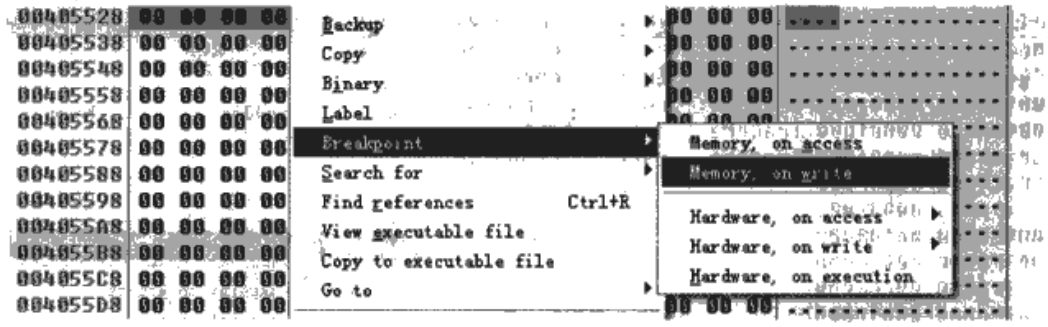


图 2.28 设置内存写入断点

下了内存写断点后，按 F9 键让程序跑起来，会马上中断在“4013D0 mov [405528], edx”这行。如果要清除内存断点，单击鼠标右键，执行“Breakpoint/Remove memory breakpoint（断点/删除内存断点）”。同样，内存访问断点操作类似。

对代码也可下内存访问断点。在 OllyDbg 里重新加载实例，随意定位一行代码，如 4013D6，单击鼠标右键，执行“Breakpoint/Memory，on access（断点/内存访问）”，如图 2.29 所示。

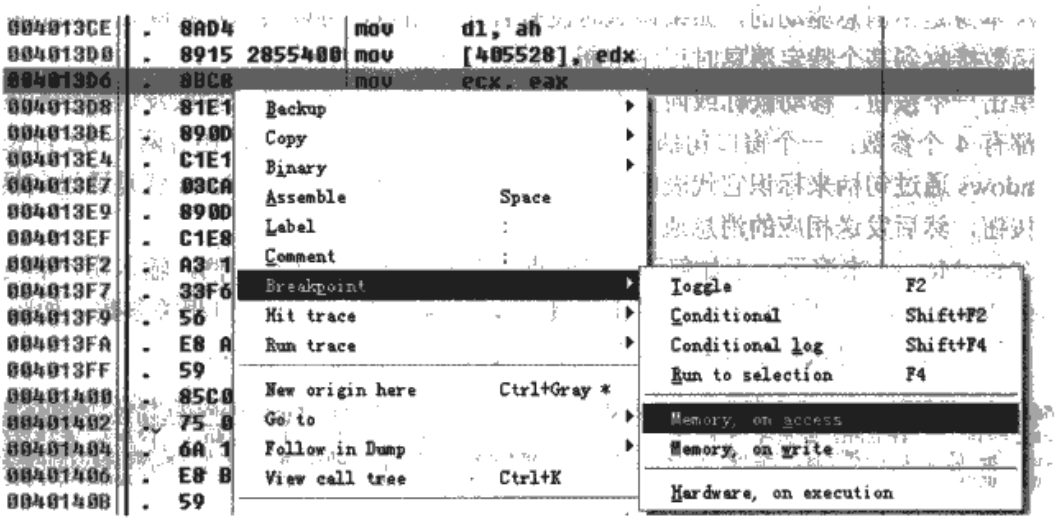


图 2.29 设置内存访问断点

当然要执行内存地址 4013D6 的代码时需要“访问”它，因此按 F9 键让实例在 OllyDbg 里跑起来，就会中断在 4013D6 这行所下的内存访问断点上。这个实验表明内存执行的地方，也可以用内存访问中断。

内存断点不修改原代码。它不会像 INT 3 断点那样，因为修改代码被程序校验而导致中断失败，因此在遇到代码校验，并且硬件断点失灵情况下，可以用内存断点来代替。

4. 内存访问一次性断点

Windows 对内存使用段页式的管理，在 OllyDbg 里按“Alt+M”键显示内存，可以看到许多段，每个段都有不可访问、读、写、执行属性。在相应的段上单击右键，如图 2.30 所示，会发现一个命令“Set break-on-access（在访问上设置断点）”，其快捷键是 F2 键，对整个内存块设置该类断点。这个断点是一次性断点，当所在段被读取或执行时就中断，中断发生以后，断点将被删除。想捕捉调用或返回到某个模块时，如后面章节中的脱壳时，该类断点就显得特别有用。右键中的“Set memory breakpoint on access（设置内存访问断点）”和“Set break-on-access”功能一样，所不同的是它不是一次性断点。这类断点仅在 NT 架构下可用。

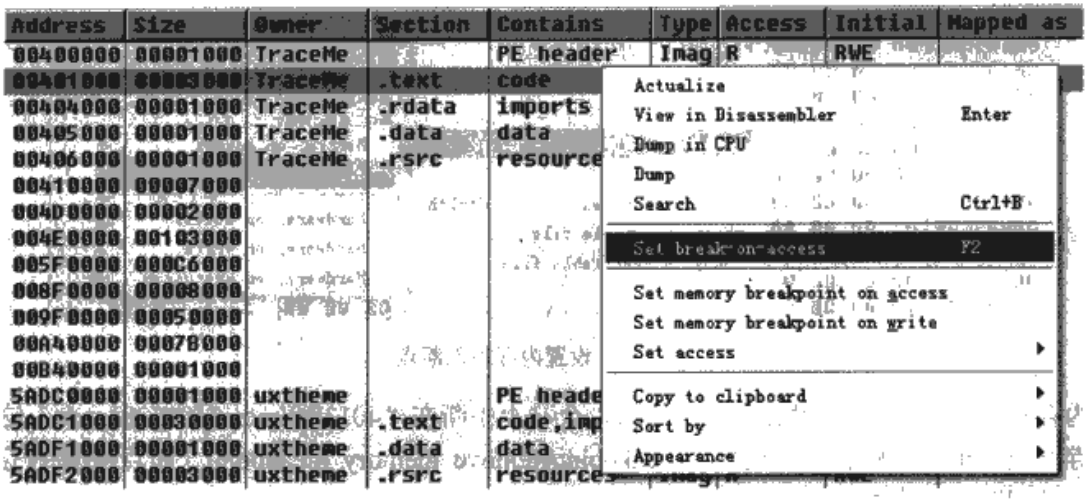


图 2.30　对区块设置内存断点

5. 消息断点

Windows 本身是由消息驱动的，如果调试时没有合适的断点，可以尝试消息断点。消息断点使得当某个特定窗口函数接收到某个特定消息时程序中断。

当用户单击一个按钮、移动鼠标或向文本框中键入文字时，一条消息就会被发送给当前的窗体。所有发送的消息都有 4 个参数：一个窗口句柄（hwnd），一个消息编号（msg），还有两个 32 位长度（Long）的参数。Windows 通过句柄来标识它代表的对象，比如单击某个按钮，Windows 就是通过句柄来判断是点击了哪一个按钮，然后发送相应的消息通知程序。

用实例 TraceMe.exe 来演示一下如何下消息断点。在 OllyDbg 里运行实例，输入用户名与序列号，单击菜单“View/Windows（查看/窗口）”或单击工具栏中的 按钮，列出窗口相关参数，如图 2.31 所示。如果界面无内容显示，此时执行鼠标右键菜单中的“Actualize（刷新）”命令。

图 2.31　列出窗口相关参数

这里用于列出所有属于被调试程序窗口及其窗口相关的重要参数，比如按钮、对应的 ID 以及句柄（Handle）等。现在要对 Check 按钮下断点，当单击按钮时中断。在 Check 条目上单击鼠标右键，如图 2.32 所示。

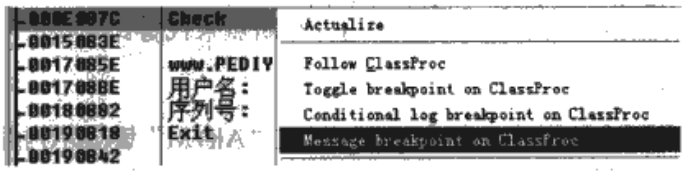


图 2.32　设置消息断点

在弹出的右键菜单中，执行“Message breakpoint on ClassProc（在 ClassProc 上设置消息断点）”，会弹出如图 2.33 所示的设置窗口。

当用鼠标左键单击按钮并松开时，会发送 WM_LBUTTONUP 这个消息，单击图 2.33 中的下拉菜单选择“202 WM_LBUTTONUP”，再单击“OK”按钮，至此消息断点设置好了。

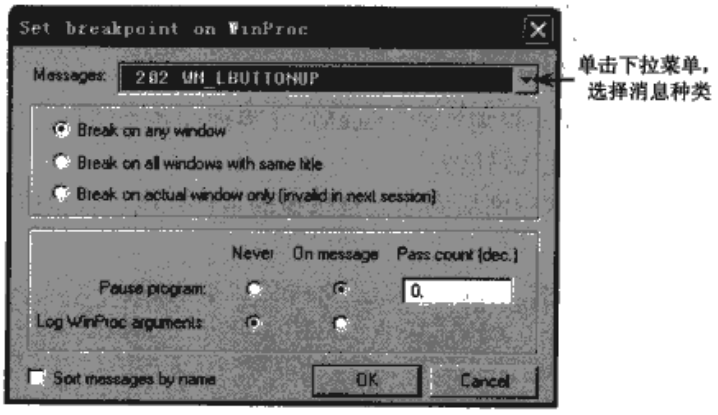


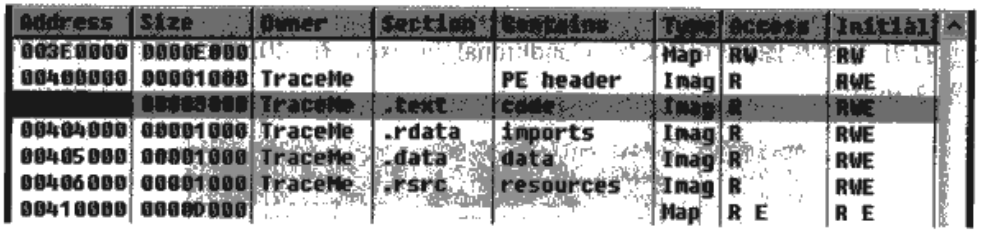
图 2.33 在 WinProc 上设消息断点

回到 TraceMe 界面，单击“Check”按钮，鼠标松开时，将会中断在 Windows 系统代码里。代码如下（不同版本的系统，代码会不同的）：

```
77D3B00E [ESP+8]=WM_LBUTTONUP   mov     edi, edi
77D3B010                        push    ebp
77D3B011                        mov     ebp, esp
77D3B013                        mov     ecx, dword ptr [ebp+8]
77D3B016                        push    esi
77D3B017                        call    77D184D0
```

现在消息捕捉到了，但处于系统底层代码里，这时企图使用“Alt+F9”键或“Ctrl+F9”键返回到 TraceMe 程序的领空代码里是徒劳的。

VC 可执行文件的执行代码是存放在代码段里的，本例就是.text 区块里。当从系统代码回到应用程序代码段的时候，正是代码段的执行，因此对代码段下内存断点就能返回应用程序的代码领空。按“Alt+M”键打开内存窗口，对.text 区块下内存访问断点，执行右键菜单命令“Set break-on-access（在访问上设置断点）”或按快捷键 F2，如图 2.34 所示。



Address	Size	Owner	Section	Contains	Type	Access	Initial
003E0000	0000E000				Map	RW	RW
00400000	00001000	TraceMe		PE header	Imag	R	RWE
00401000	00003000	TraceMe	.text	code	Imag	R	RWE
00404000	00001000	TraceMe	.rdata	imports	Imag	R	RWE
00405000	00001000	TraceMe	.data	data	Imag	R	RWE
00406000	00001000	TraceMe	.rsrc	resources	Imag	R	RWE
00410000	0000D000				Map	R E	R E

图 2.34 对代码段下内存访问断点

现在按 F9 键运行程序，立即中断在程序的空间 004010D0 处，这里正是程序的消息循环处。

```
004010D0   sub     esp, 0F4
004010D6   push    esi
004010D7   push    edi
004010D8   mov     ecx, 5
004010DD   mov     esi, 00405060
......
00401132   sub     eax, 10          ; Switch (cases 10..111)
00401135   mov     dword ptr [esp+10], edx
00401139   movs    byte ptr es:[edi], byte ptr [esi]
0040113A   je      00401314
00401140   sub     eax, 100
```

```
00401145   je      004012CD
0040114B   dec     eax
0040114C   jnz     004012C0
```

这段代码是一个消息循环，不停地处理 TraceMe 主界面的各类消息，此时可能不是直接处理按钮事件，如果单步跟踪，可能会跟进系统代码里去。

可以重复这个过程，在几次中断后到达处理按钮的事件代码。很快就能发现“Check”按钮事件的代码了：

```
0040119C   mov     esi, dword ptr [esp+100]    ; Case 3F5 of switch 0040115E
004011A3   mov     edi, dword ptr [<&USER32. GetDlgItemTextA >
004011A9   push    ebx
004011AA   lea     eax, dword ptr [esp+4C]
004011AE   push    51                          ; /Count = 51 (81.)
004011B0   push    eax                         ; |Buffer
004011B1   push    6E                          ; |ControlID = 6E (110.)
004011B3   push    esi                         ; |hWnd
004011B4   call    edi                         ; \GetDlgItemTextA
```

最后，可以将消息断点删除，方法是按“Alt+B”键切换到断点窗口，选中消息断点，直接删除，如图 2.35 所示。

图 2.35　删除消息断点

6. 条件断点

在调试过程中，经常希望断点满足一定条件时才中断，这类断点称为条件断点。OllyDbg 的条件断点可以按寄存器、存储器、消息等设断。条件断点是一个带有条件表达式的普通 INT 3 断点。当调试器遇到这类断点时，它将计算表达式的值，如果结果非零或者表达式无效，则断点生效（即暂停被调试程序）。有关条件表达式的规则描述请参考 OllyDbg 帮助文档。

（1）按寄存器条件中断

用 OllyDbg 打开光盘映像文件中实例 Conditional_bp.exe，在 40147C 这行，按条件断点的快捷键“Shift+F2”，如图 2.36 所示。

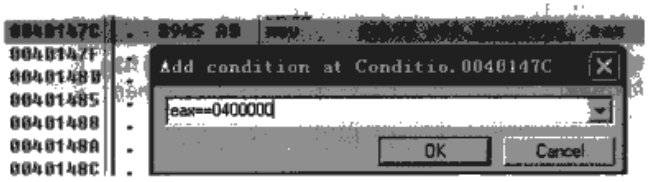


图 2.36　下条件断点

在条件框内，输入条件表达式“eax==0400000”。这样，程序执行到 40147C 这行时，如果 eax 值为 400000h，OllyDbg 将中断。如果安装了命令行插件，也可在命令行里直接输入：

```
bp 401476 eax==0400000
```

（2）按存储器条件中断

在这以 CreateFileA 函数演示一下。在实际情况中，程序可能成百上千次地调用 CreateFileA 函数，让 OllyDbg 在 CreateFileA 打开所需要文件时中断，显得十分有必要。先来看一下 CreateFile 函数的定义：

```
HANDLE CreateFile(
    LPCTSTR lpFileName,                          // 指向文件名的指针
```

```
    DWORD dwDesiredAccess,                  // 访问模式
    DWORD dwShareMode,                      // 共享模式
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // 指向安全属性的指针
    DWORD dwCreationDistribution,           // 如何创建文件
    DWORD dwFlagsAndAttributes,             // 文件属性
    HANDLE hTemplateFile                    // 用于复制文件句柄
);
```

运行实例 Conditional_bp，对 CreateFileA 设断，单击“OpenTest”按钮，如图 2.37 所示是当 OllyDbg 弹出时从堆栈中看到的。图中最左侧标识了各参数相对于当前 ESP 的地址，打开这个功能的方法是在堆栈窗口单击右键，执行“Address/Relative to ESP（地址/相对于 ESP）”菜单。

```
ESP   >  0012F940  00401279  CALL to CreateFileA from Conditio.00401274
ESP+4    0012F944  00406184  FileName = "\\.\NTICE"
ESP+8    0012F948  C0000000  Access = GENERIC_READ|GENERIC_WRITE
ESP+C    0012F94C  00000003  ShareMode = FILE_SHARE_READ|FILE_SHARE_WR
ESP+10   0012F950  00000000  pSecurity = NULL
ESP+14   0012F954  00000003  Mode = OPEN_EXISTING
ESP+18   0012F958  00000080  Attributes = NORMAL
ESP+1C   0012F95C  00000000  hTemplateFile = NULL
```

图 2.37 CreateFileA 参数刚入堆栈的情形

CreateFile 采用标准调用约定，参数按从右到左的顺序入栈。因为在函数刚执行时，EBP 堆栈结构还未建立，只得用 ESP 访问这些参数。CreateFile 函数第一个参数 FileName 是文件名指针，在 OllyDbg 里如要得到第一个参数的值，可以用[ESP+4]来获得；如果还要得到此处地址所指的字符串，必须用[[ESP+4]]来表示。实例 Conditional_bp 调用了 4 次 CreateFileA 函数，假设当 CreateFile 打开“c:\\1212.txt”时需要 OllyDbg 中断，条件断点可以这样设置，将光标移到 CreateFileA 函数第一行，按快捷键“Shift+F2”，键入字符“[STRING [esp+4]]=="c:\\1212.txt"”，STRING 前缀在 OllyDbg 中的解释是以零作为结尾的 ASCII 字符串，如图 2.38 所示。

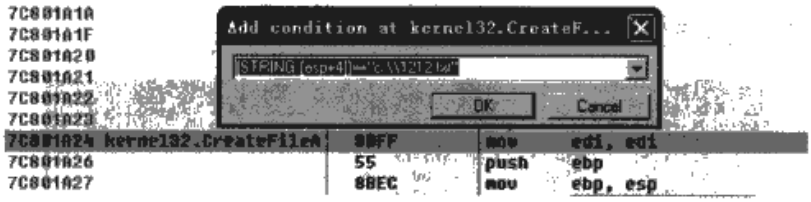


图 2.38 条件断点

如果安装了命令行插件，也可以直接输入：

```
bp CreateFileA,[STRING [esp+4]]=="c:\\1212.txt"
```

7. 条件记录断点

条件记录断点除了具有条件断点作用，还能记录断点处函数表达式或参数的值。也可以设置通过断点的次数，每次符合暂停条件时，计数器减一。

例如要记录 Conditional_bp 实例调用 CreateFileA 函数的情况，在 CreateFileA 函数第一行，按“Shift+F4”键，出现条件记录窗口，如图 2.39 所示。

在 Condition（条件）域中输入要设置的条件表达式。Explanation（说明）域中由用户自己设置一个名称。Expression（表达式）域中是要记录的内容的条件，只能设置一个表达式，例如要记录 EAX 的值，可以输入 EAX。还有一个 Decode value of expression as（解码表达式的值）下拉选择框，这是对记录的数据进行分析，例如在图 2.39 中，如果 Expression 域中填的是[esp+4]，则得在下拉选择框中选择“Pointer to ASCII String（指向 ASCII 字符串的指针）”，才能得到正确的结果，功能相当于 STRING 前缀。

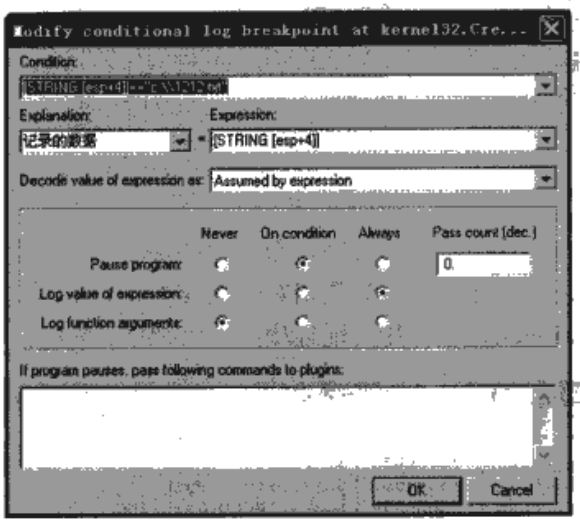


图 2.39　设置条件记录断点

Pause program（暂停程序）是指 OllyDbg 遇到断点时是否中断；Log value of expression（记录表达式值）是指遇到断点时是否记录表达式的值；Log function arguments（记录函数参数）是指遇到断点时是否记录函数参数。可以根据需要设置 Never（从不）、On condition（按条件）、Always（永远）等条件。

条件记录断点允许传递一个或多个命令给插件。当应用程序因条件断点暂停，并且断点包含有传递给插件的命令时，都会调用回调函数 ODBG_Plugincmd(int reason,t_reg *registers,char *cmd)。例如，当程序暂停时，传送一个命令“d esp”给 CmdBar 插件，只要在图 2.39 所示的框中输入字符“.d esp”，注意命令前有一个点字符“.”，那么当条件断点断下时，就会执行“d esp”这个命令，这时我们就可以在数据窗口中看到 ESP 地址处的数据了。

设置好条件记录断点后，单击实例 Conditional_bp 的“OpenTest”按钮，运行后，OllyDbg 会在 Log data 窗口（快捷键“Alt+L”）记录下数据，如图 2.40 所示。

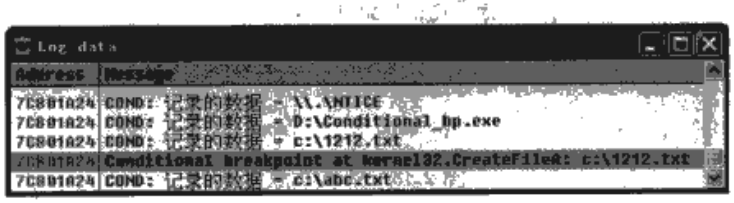


图 2.40　查看 Log data 窗口

2.1.6　插件

OllyDbg 支持插件，这意味它的功能扩展性很好，可以按自己需要扩展相关的功能，提高调试的灵活性。首先按图 2.3 设置插件所在的目录，将相应的插件复制到这个目录下，重新运行 OllyDbg 就可加载插件。OllyDbg 默认只能加载 32 个插件，另外，各插件之间有可能会冲突，因此建议插件目录仅放常用的插件，以减少各类问题出现。

在此介绍一下自带的命令行插件。

1．常用插件介绍

光盘映像文件中的 OllyDbg 已集成了常用的插件，这些插件使用都比较简单，本节不过多介绍，读者可参考相关的资料。

（1）命令行插件 CmdBar

单击菜单“Plugins/command line/command line”打开命令行插件，如图 2.41 所示。

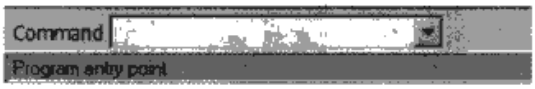


图 2.41 加载命令行插件的效果

命令参数较多，详细信息请参考其自带的帮助文件，在此列出几个常用命令，如表 2-3 所示。

表 2-3 CmdBar 插件常用命令

命 令	含 义
? 表达式	计算表达式的值，如：? 34*45-4
D(DB,DW,DD) 表达式	查看内存数据，如：D 401000，D esp+c
BP 表达式 [,条件式]	设置断点，如：bp GetDlgItemTextA
Hw 表达式	设置硬件写断点

（2）OllyScript 插件

OllyDbg 的脚本插件，可以用 OllyScript 脚本来完成一些复杂的操作或重复的操作，具体使用可参考其 ReadMe。

2. 插件的开发

开发插件，到 OllyDbg 官方站点下载文档 OllyDbg Plugin API v1.10 及 SDK，具体方法可以参考光盘映像文件中提供的样例。

2.1.7 Run trace

Run trace（Run 跟踪）可以把被调试程序执行过的指令保存下来，了解以前发生的事件。它能将地址、寄存器的内容、消息等记录到 Run trace 缓冲区中。在运行 Run trace 前，要将缓冲区设置大些，否则执行的指令太多造成缓冲区溢出，这时 OllyDbg 会自动丢弃老的记录。可以在“Debugging options/Trace（调试选项/跟踪）”面板中设置，如图 2.42 所示。

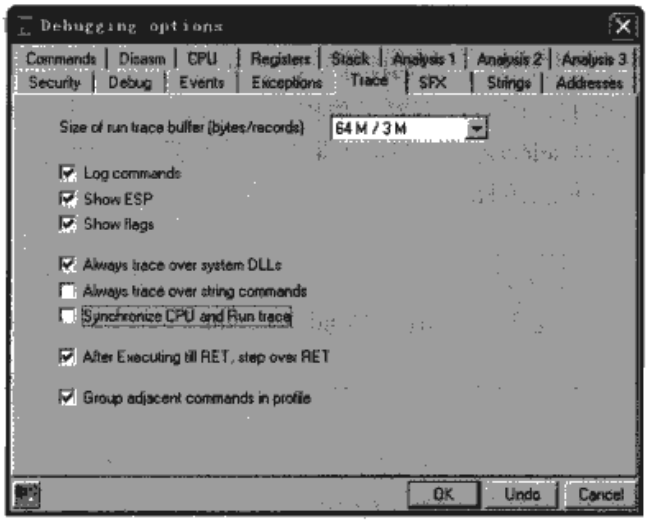


图 2.42 设置 Run trace 缓冲区

如果要将 Run trace 的数据保存到文件，在跟踪之前，单击菜单“View/Run trace”或 按钮，打开 Run trace 窗口，单击右键执行“Log to file（记录到文件）”，如图 2.43 所示。

需要运行 Run trace 时，单击菜单“Debug/Open or clear run trace（打开或清除 Run 跟踪）”。在打开 Run trace 缓冲区后，OllyDbg 会记录执行过程中的所有暂停，然后通过使用“+”键和“-”键（“+”键必须是数字键盘上的）来反方向浏览程序的执行，此时 OllyDbg 会使用实际的内存状态来解释寄存器、堆栈的变化。

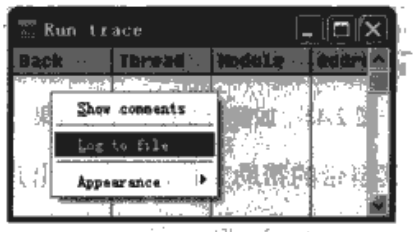


图 2.43　设置 Run trace 缓冲区

在反汇编窗口显示的是被调试程序领空时，在反汇编窗口的快捷菜单中选择“Run trace/Add entries of all procedures（Run trace/添加所有函数过程的入口）”，这样能够检查每个可识别的函数被调用的次数，如图 2.44 所示。运行后，可以在 Run trace 窗口的快捷菜单中执行“Profile module（统计模块）”查看统计次数。

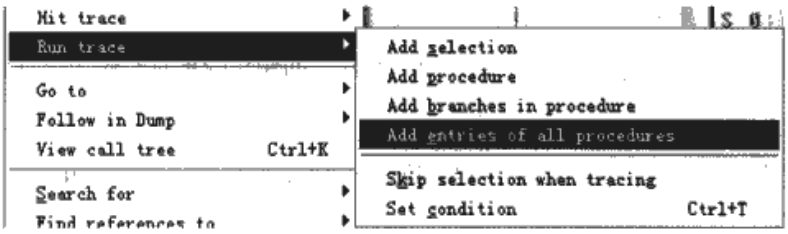


图 2.44　添加 Run trace 选项

按 F9 键或者按“Ctrl+F12”组合键（跟踪步过）让程序运行，查看 Run trace 结果只需单击菜单“View/Run trace”或 按钮，打开 Run trace 窗口。

2.1.8　Hit trace

Hit trace 能够让调试者辨别哪一部分代码执行了，哪一部分没有。OllyDbg 的实现方法相当简单，它将选中区域的每一条命令处均设置一个 INT 3 断点，当中断发生的时候，OllyDbg 便把它去除掉。在使用 Hit trace 的时候，不能在数据中设置断点，否则程序可能会崩溃。

当碰到一段跳转分支比较多的代码时，需要了解程序执行线路，可以用 Hit trace。方法是选中这段代码，单击右键执行“Hit trace/Add selection”，将需要监视的代码选中，然后按 F9 键让程序运行，OllyDbg 会在已被执行过的指令前用另一种颜色标记出来。

注意：如果右键菜单中没有 Hit trace，则必须打开相关的菜单选项，进行代码分析。比如可以按“Ctrl+A”键或执行快捷键中“Analysis/Analyse code（分析/分析代码）”命令重新分析一下代码。

2.1.9　符号调试技术

高级语言编译器都附带源代码级的调试器，如 Visual C++、Delphi 等。OllyDbg 等调试器除了进行汇编级调试外，也可在源代码状态下调试应用程序，当然要实行源码调试需要符号信息。

1. 符号格式

符号表（又称调试符）的作用是将十六进制数转换为源文件代码行、函数名以及变量名称。符号表还包含程序使用的类型信息，调试器使用类型信息可以获取原始数据，并将这些原始数据显示为在程序中所定义的结构或变量。

（1）SYM 格式

SYM 格式早期用于 MS-DOS 和 16 位 Windows 系统，现在只用做 Windows 9x 的调试符（因为 Windows 9x 多数内核仍然是 16 位代码）。

（2）COFF 格式

COFF 格式（Common Object File Format）是 UNIX 供应商所遵循的规范的一部分。Windows NT 2.1 首次引进使用，现在微软逐渐抛弃 COFF 格式，而使用更流行的符号表达式。

（3）C7 格式（CodeView 格式）

最早是在 MS-DOS 作为 Microsoft C/C++ 7 的一部分而问世的，现在已经支持 Win32 系统。CodeView 是早期 Microsoft 调试器的名称，其支持的调试符号为 C7 格式。C7 格式在执行模块中是自我包含的，符号信息与二进制代码混合，进而意味着调试文件会非常大。

（4）PDB 格式

PDB（Program Database）格式是现今最常用的一种符号格式。Visual C++和 Visual Basic 都支持 PDB 格式。PDB 与 C7 不同，PDB 符号根据应用程序不同的链接方式，保存在单独的一个文件或多个文件中。

（5）DBG 格式

DBG 是系统调试符号，有了系统调试符号，调试器才可以显示系统函数名。DBG 文件与其他符号格式不同，因为链接器并不创建 DBG 文件，DBG 文件基本是一个包含其他调试符号的文件，如包含 COFF 或 C7 等类型调试符号。微软将操作系统调试符都分配在 DBG 文件中，当然这些文件只包括公用信息和全局信息，如 ntdll.dbg、kernel32.dbg 等。Windows 9x 没有系统调试符，Windows NT/2000/XP 提供了系统调试符。

（6）MAP 文件

MAP 文件是程序的全局符号、源文件和代码行号信息的唯一文本表示方法。MAP 文件在任何地方、任何时候都可以使用，不要求支持程序，通用性极好。

2. 创建调试文件

进行源码级调试的首要条件是生成的文件包含调试信息。调试信息包含程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号。调试器利用这些信息使源代码和机器码相关联。各语言编译器都能产生相关调试信息，具体见表 2-4。

表 2-4 编译器设置调试信息

编译语言	产生调试信息
Borland C++ 4.5 和 5.0	产生 Borland 标准调试信息： • 编译器参数：/v • 连接器选项：/v
MASM	产生 CodeView 格式的调试信息： • 编译器参数：ML/Zi /COFF • 连接器选项：/DEBUG/DEBUGTYPE:CV [/PDB:NONE]
Visual C++ 2.x,4.0,4.1, 4.2, 5.0 和 6.0	产生程序数据库（Program Database，PDB）调试信息： • 编译器用“Program Database”选项；命令行参数：/Zi • 连接器选项：/DEBUG/DEBUGTYPE:CV 注意：VxD 文件需要 PDB 调试信息 产生 CodeView 格式的调试信息： • 编译器用“C7-compatible”选项；命令行参数：/Z7 • 连接器选项：/DEBUG/DEBUGTYPE:CV /PDB:NONE 注意：如果是用标准的 Windows NT DDK 开发，环境变量如下设置： NTDEBUG=ntsd an NTDE-BUGTYPE=windbg

在这以 Visual C 6.0 为例，建立带有 PDB 调试信息的调试版本：

① 单击“Build”菜单，单击“Set Active Configuration”配置，从对话框中选择“Win32 Debug”配置。

② 在“Project”菜单中单击“Settings”打开设置对话框，单击“C/C++”标签，从“Category”下拉框中选择“General”，在调试信息（Debug Info）部分，选择“Program Database”选项。该选项产生一个存储程序信息的数据文件（.PDB），它包含了类型信息和符号化的调试信息。

③ 单击“Link”标签，从“Category”下拉框中选择“Debug”，在调试信息（Debug Info）部分，选择“Debug info”、“Microsoft format”、“Separate types”，在此也可选择“Generate mapfile”来生成 MAP 文件。

3．用符号文件调试

用 Visual C 6.0 编译光盘映像文件中提供的 TraceMe 源码，运行 OllyDbg 打开 Debug 目录下带有调试信息的 TraceMe，这次显示的汇编代码带有调试符，具有更好的可读性。

```
00401047   mov    dword ptr [hInst], eax
```

在 OllyDbg 主菜单中单击“View/Source”打开源码窗口，可看到源码。OllyDbg 不能直接在源码窗口单步调试，此时必须做些调整，用 CPU 窗口来配合调试。同时打开 CPU 窗口和 Source 窗口，并适当地调整它们的位置和大小。然后在“Debugging options”里设置“CPU”标签，勾选“Synchronize source with CPU”，这样跟踪时汇编与源码就能同步了。也可以在 CPU 窗口中，单击第 4 栏的标题，直接在 CPU 窗口里同步显示汇编和源码。

可单击菜单“View/Source files”打开源码文件路径窗口查看，一些路径不正确的源文件，OllyDbg 默认将不显示。需要显示，可以在“Debugging options”里设置，切换到“Debug”标签，将“Hide non-existing source files”选中去除，即可显示不存在的源文件路径。

2.1.10 OllyDbg 常见问题

OllyDbg 这款调试器非常强大，初学者刚接触可能会遇到许多问题，在这将一些常见的问题列出。

1．乱码的问题

当用 OllyDbg 跟踪程序时，可能会出现如下情况：

```
004010CC    55          db    55
004010CD    8B          db    8B
004010CE    EC          db    EC
004010CF    83          db    83
004010D0    56          db    56
```

这是因为 OllyDbg 将这段代码当成数据了，没有进行反汇编识别。在右键快捷菜单中，执行“Analysis/Analyse code（分析/分析代码）”或按“Ctrl+A”键强迫 OllyDbg 重新分析一下代码即可。如果还不行，尝试在快捷菜单中执行“Analysis/Remove analysis from module（分析/从模块中删除分析）”或在 UDD 目录中删除相应的.udd 文件。

调整后就以代码显示了：

```
004010CC    55          push   ebp
004010CD    8BEC        mov    ebp, esp
004010CF    83EC 44     sub    esp, 44
004010D2    56          push   esi
```

2．如何快速回到当前领空

在 OllyDbg 中有时查看代码可能翻页到其他地方，如想快速回到当前 CPU 所在的指令上，可以双击寄存器面板中的 EIP 或单击 按钮。

3. OllyDbg 如何修改 EIP

首先将光标移到所需要的地址上，然后执行右键菜单“New origin here（此处新建 EIP）”或使用快捷键“Ctrl+*”。

4. 什么是 UDD

OllyDbg 把所有程序或模块相关的信息保存至单独的文件中，并在模块重新加载时继续使用。这些信息包括了标签、注释、断点、监视、分析数据、条件等。

5. 为什么删除了断点，OllyDbg 重新加载时，这些断点都会重新出现

设置 ollydbg.ini，在配置文件里改成：Backup UDD files=1

6.OllyDbg 反汇编窗口键入汇编代码时，输入“push E000”会提示未知标识符（见图 2.45）

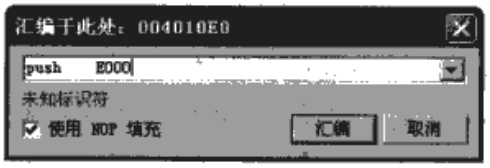


图 2.45　添加 Run trace 选项

这是因为 OllyDbg 的反汇编引擎不能正确识别字符“E000”中的 E 是字母还是数字。解决的办法是，在字母前加个 0，表示这是数字，即“push 0E000”。

7. OllyDbg 出会现假死现像

用 OllyDbg 调试一些加壳程序，程序跑到断点（包括硬件断点）时，OllyDbg 会出现假死现象。解决方法是打开配置文件 ollydbg.ini，如果“Restore windows”是一个很大的值，现在只需要设“Restore windows =0”即可。

8. 如何微调窗口显示

可以通过“Ctrl + ↑”键或“Ctrl+ ↓”键对反汇编窗口或数据窗口翻动一个字节。

9. 执行复制到可执行文件时，出错“Unable to locate data in executable file”

这个要修改的地方不在 RawSize 范围内，修改 PE 文件，使“RawSize = VirtualSize”。

10. 能否把 CALL 调用改成函数名形式

比如“call 401496”，假设 401496 处是 amsg_exit 函数，将光标停在 401496，单击“Shift+;”键，出来一个标签框，输入字符“amsg_exit”，这样，所有调用 401496 的 CALL 都变成“call <amsg_exit>”形式。

2.2 SoftICE 调试器

SoftICE 是一款优秀的系统级调试工具。Compuware 公司已在 2006 年 4 月宣布 DriverStudio 停止开发，这意味着 SoftICE 将在新系统中消失。本节只简单介绍一下 SoftICE 基本用法，详细的请参考其自带的文档 Using SoftICE.pdf 和 CommRef.chm。

本书实例用 OllyDbg 都可调试，当遇到 Ring 0 级程序时，才需要 SoftICE、WinDBG，因此读者可以跳过此节的学习。

由于篇幅限制，本节内容以电子文档形式放在光盘映像文件中提供。

第3章 静态分析技术

用高级语言编写的程序有两种形式，一种被编译成机器语言在CPU上执行，如Visual C++、Pascal等。由于机器语言与汇编语言几乎是对应的，因此可将机器语言转化成汇编语言，这个过程称为反汇编（Disassembler）。例如，在x86系统中，机器码“EBh”对应的汇编语句是“jmp short xx”。另一种高级语言是一边解释一边执行的，称为解释性语言，如Visual Basic 3.0/4.0、Visual FoxPro等，这类语言的编译后程序可以被还原成高级语言的原始结构，这个过程称为反编译（Decompiler）。

所谓静态分析，即从反汇编、反编译手段获得程序汇编代码或源代码，然后从程序清单上分析程序流程，了解模块完成的功能。

3.1 文件类型分析

静态分析的第一步就是分析程序的类型，了解程序是用什么语言编写的或用什么编译器编译的？程序是否被某种加密程序处理过？然后才能有的放矢地进行下一步工作。

常见的文件分析工具有PEiD、FileInfo等，本节简单地讲一讲它们的用法。

3.1.1 PEiD工具

PEiD是一款常用的文件检测分析工具，具有GUI界面。它能检测大多数编译语言、病毒和加密的壳。图3.1显示被分析的文件是用Microsoft Visual C++ 6.0编译的，分析不出类型的文件可能报告是“PE Win GUI”，Win GUI就是Windows图形用户界面程序的统称。使用时，建议设置一下Options，勾上“Register Shell Extensions”，即可在鼠标右键添上快捷菜单。

PEiD这类文件分析工具是利用查特征串搜索来完成识别工作的。各种开发语言都有固定的启动代码部分，利用这点就可识别出是何种语言编译的。被加壳程序处理过的程序，在壳里会留下相关加壳软件的信息，利用这点就可识别是被何种壳所加密的。

PEiD提供了一个扩展接口文件userdb.txt，用户可以自定义一些特征码，这样就可识别出新的文件类型。签名的制作可以用插件Add Signature来完成，必要时，还必须用OllyDbg等调试器来配合修正。有些外壳程序为了欺骗PEiD等文件识别软件，会将一些加壳信息去除，并伪造启动代码部分。例如，将入口代码改成与Visual C++ 6.0所编程程序入口处类似代码，即可达到欺骗目的。所以，文件识别工具所给出的结果只是个参考，文件是否被加壳处理过，还得跟踪分析程序代码才可得知。

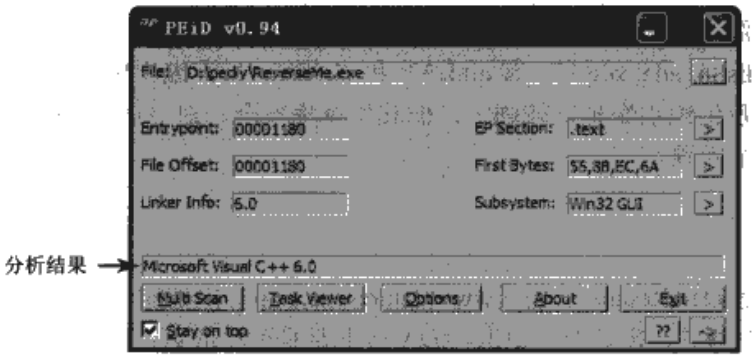


图 3.1 PEiD 主界面

3.1.2 FileInfo 工具

FileInfo（简称 Fi）是另一款不错的文件检测工具。Fi 运行时是 DOS 界面，在 DOS 窗口中运行程序相当不便，建议采用下面的技巧：

- 第一种方法是用鼠标将文件拖到 Fi.exe 主文件上。
- 第二种方法是先建 Fi 的一个快捷方式，然后把这个快捷方式放进 Windows 的 SendTo 文件夹里（Winodws XP 用户需将“隐藏文件和文件夹”功能关闭，才能看到 SendTo 文件夹）。以后要分析某文件，只需选中文件，然后右键单击，选择“发送到”功能就可打开 Fi（见图 3.2）。建议读者把一些常用工具放进 SendTo 文件夹，以方便操作。

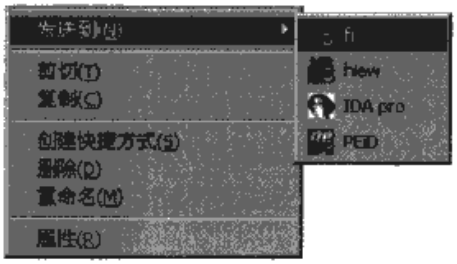


图 3.2 SendTo 菜单

FileInfo 与 PEiD 相比有更强的文件识别能力，也就是说，更难欺骗些。但 FileInfo 的识别库不能自定义，并且升级也缓慢；而 PEiD 用户可自定义特征码，因此识别的壳的类型更多。

3.2 静态反汇编

本节主要介绍常见的反汇编工具用法。在进行反汇编前，建议用 FileInfo、PEiD 等检测工具分析一下文件是否加壳。如果加壳，就需利用后面章节介绍的脱壳技术脱壳，然后再反汇编。常用的反汇编工具有 W32Dasm、C32asm、IDA Pro 等。W32Dasm、C32asm 这两款工具已不升级，不支持 64 位程序。W32Dasm 使用比较简单，其操作方法以电子文档形式放到光盘映像文件中提供。C32asm 这款工具十分优秀，其字符串提取功能等十分强大，并且将十六进制工具等功能结合了起来。IDA Pro 是一款商业软件，属于专家级产品，功能强大但操作也复杂，逆向工程必备工具。一些简单的程序，可以用 W32Dasm、C32asm 等来分析，比较方便。

3.2.1 反汇编引擎

反汇编引擎的作用是把机器码解析成可以汇编指令，开发反汇编引擎需要对 intel 公司的 i386 的机器指令编码有深入的了解。不过，一般不需要自己开发，网上有开源或收费的反汇编引擎，利用这些反汇编

引擎，自己也可开发一款反汇编分析工具。

常见的反汇编引擎有 udis86, ade, xde 等，像 mlde32 和 virxasm 反汇编引擎也比较有特色，大小仅 400 字节左右。OllyDdg 自带的反汇编引擎也比较强大，但其指令集不全，对 MMX, SSE 支持的不好，不过 FullDisasm 插件可以解决这个问题。

3.2.2 IDA Pro 简介

IDA Pro（简称 IDA）是 DataRescue 公司（www.datarescue.com）出品的一款交互式反汇编工具，它功能强大、操作复杂，要完全掌握它，需要很多知识。IDA 最主要的特性是交互和多处理器。操作者可以通过对 IDA 的交互来指导 IDA 更好地反汇编，IDA 并不自动解决程序中的问题，但它会按用户的指令找到可疑之处，用户的工作是通知 IDA 怎样去做。比如人工指定编译器类型，对变量名、结构定义、数组等定义等。这样的交互能力在反汇编大型软件时显得尤为重要。多处理器特点是指 IDA 支持常见处理器平台上的软件产品。

IDA 支持的文件类型非常丰富，除了常见的 PE 格式，还支持 Windows,DOS,UNIX,Mac,Java,.NET 等平台的文件格式。单击菜单“File/Open”打开目标软件 ReverseMe.exe，IDA 一般能自动识别出格式，如图 3.3 所示。也可单击菜单“File/New”以向导模式打开文件。

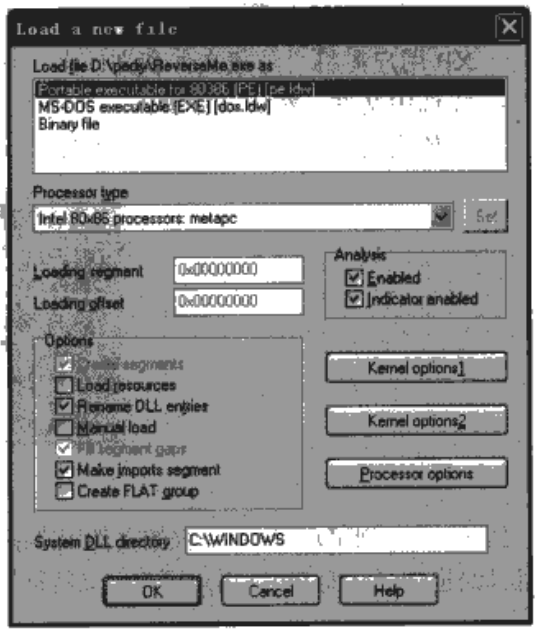


图 3.3 IDA 打开文件

IDA 打开文件分自动和手工两种模式，默认是自动模式，手工模式（Manual load）可以自由选择需要加载的区块。IDA 装载 PE 文件时，是按区块来装载的，比如.text（代码块）、.data（数据块）、.rsrc（资源块）、.idata（输入表）、.edata（输出表）等。IDA 反汇编的时间与程序大小及复杂程度有关，需要等一段时间才能完成。此过程分两个阶段，第一阶段将程序的代码和数据分开，然后为各函数作标记并分析其参数调用，分析跳转、调用等指令关系并给标签赋值等。在第二阶段，如果 IDA 识别出文件的编译类型，就装载对应的编译器特征文件，然后给各函数赋名。

Kernel option1、Kernel option2、Processor option 这三个选项可以控制反汇编引擎的工作状态，一般按默认即可。IDA 会自动识别程序类别与处理器类型，在大多数情况下，分析选项的默认值在准确性与方便性之间提供一个折中参数，如果 IDA 分析出有问题的代码时，将核心选项 1 中的“Make final analysis pass”选项关闭是一个很好的方法。在有些情况下，一些代码因不在预计的位置而不被确认，尝试将核心选项 2 中的“Coagulate Data Segments in the final pass”选上是有帮助的。

IDA 打开文件后，默认是处于图形化模式下，如果要切换到代码模式，执行右键菜单“Text view”。

3.2.3 IDA 的配置

合理配置 IDA 文件，可以大大提高工作效率。Windows 图形界面的主程序是 idag.exe，可通过菜单“Options（选项）”来配置 IDA，但是这种配置仅对当前的项目有效，新建一个项目时，会恢复成默认配置，改变默认配置必须编辑 ida.cfg 文件。

在 IDA 的 cfg 目录下查找 ida.cfg 或 idagui.cfg 文件，这是一个文本文件。不能用 Windows 记事本打开 ida.cfg 编辑，因为记事本对一些特殊字符识别不好，继续编辑和保存文件，文件将会被破坏。此时配置文件会导致 IDA 运行失败，如图 3.4 所示。建议用户用 EditPlus、UltraEdit 等工具修改 ida.cfg 等配置文件。

图 3.4 ida.cfg 文件损坏警告窗口

ida.cfg 文件由两部分组成。第一部分定义文件的扩展名、内存、屏幕等设置；第二部分配置普通参数，如代码显示格式、ASCII 字符串显示格式、脚本定义和处理器选项等。另外，一些问题也与 ida.cfg 有关，如 MAX_ITEM_LINES，默认为 5000 行，对于许多大文件经常会不够使用，因而发生错误。

1．反汇编选项（Disassembly）

这个选项直接控制反汇编窗口的代码显示格式。单击菜单“Options（选项）”中的“General（常规）”，将出现如图 3.5 所示的对话框。

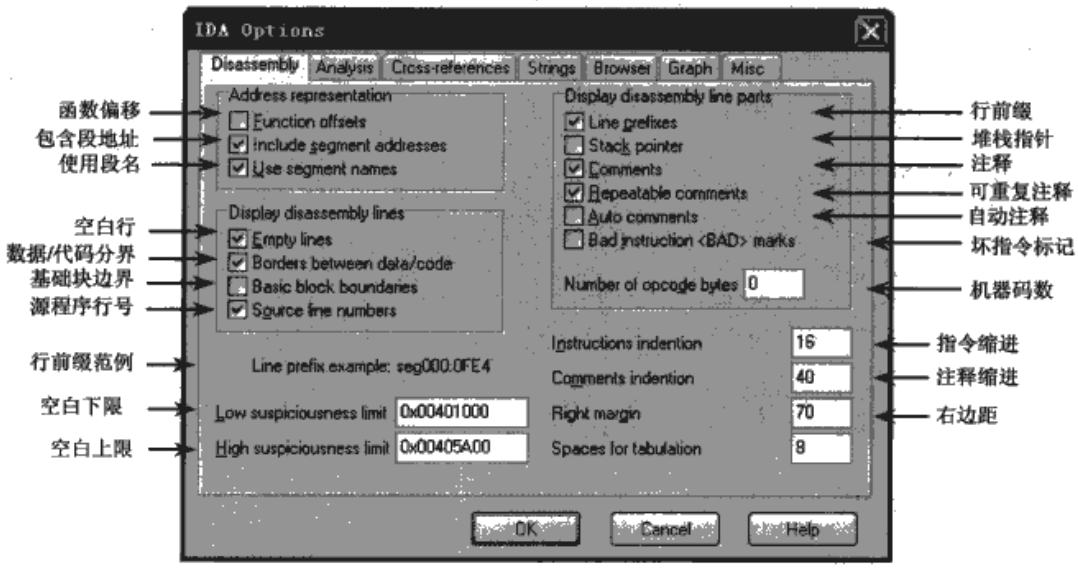


图 3.5 反汇编选项配置

ida.cfg 与这部分配置对应的是文本格式（Text representation）。由于其项目较多，下面只列出重点部分。配置如下：

```
OPCODE_BYTES            = 6             ;机器码字节数，默认是 0
INDENTION               = 0             ;指令缩进，默认是 16，设为 0 代码整洁一些
COMMENTS_INDENTION      = 30            ;注释缩进，默认是 40
MAX_TAIL                = 16            ;交叉参考的深度，默认是 16
MAX_XREF_LENGTH         = 80            ;交叉参考显示的右边距
MAX_DATALINE_LENGTH     = 100           ;主窗口代码右边距，默认是 70
SHOW_AUTOCOMMENTS       = NO            ;自动注释
SHOW_BAD_INSTRUCTIONS   = NO            ;坏指令标记
SHOW_BORDERS            = YES           ;数据与代码之间分界
SHOW_EMPTYLINES         = YES           ;显示空白，使汇编代码易读
SHOW_LINEPREFIXES       = YES           ;显示行前缀（如 1000:0000）
SHOW_XREFS              = 15            ;显示大量的交叉参考，默认值是 2
SHOW_ORIGINS            = NO            ;产生“org”标记，默认是 YES
```

2. ASCII 字符串与符号（ASCII strings & names）

ASCII 字符串风格可在菜单“Options/ASCII String styles”中打开字符串设置窗口，对应的 ida.cfg 部分配置如下：

```
ASCII_GENNAMES          = YES           ;生成符号名
ASCII_TYPE_AUTO         = YES           ;自动产生符号名
ASCII_PREFIX            = "a"           ;符号名前缀

#define ASCII_STYLE_C         0x00000000   ;ASCII 字符串
#define ASCII_STYLE_UNICODE   0x00000003   ;Unicode 字符串
```

3. 显示中文字符

IDA 默认是不显示中文字符串的，只需要在 ida.cfg 中搜索 AsciiStringChars，将 cp866 version 这段注释掉，恢复 full version 这段即可显示中文。

```
// (cp866 version)
//AsciiStringChars =
//       "\r\n\a\v\b\t\x1B"
//       " !\"#$%&'()*+,-./0123456789:;<=>?"
//       "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
         ......
//       "噌杆溴骁栝觇祉铒";
//(full version)
AsciiStringChars =
     "\r\n\a\v\b\t\x1B"
     " !\"#$%&'()*+,-./0123456789:;<=>?"
     "@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_"
     ......
     "懒旅呐魄壬仕掏蜗醒矣哉肿到谜茬捶"
     "噌杆溴骁栝觇祉铒寐蝗趱鳏      ?";
```

3.2.4 IDA 主窗口界面

IDA 分析完目标程序后进入主窗口，界面显得专业和复杂。IDA 相当智能，尽量分析程序各模块的功能，并给出相应提示，例如为 API 函数的参数自动加上注释，相当直观易懂。对于那些 IDA 不能正常分析的代码，则需要手工来辅助分析。

1. 翻页

当执行跳转功能后，需要返回时，只要在工具栏中单击 按钮或按 Esc 键，列表便会往后跳一页；若要往前一页，单击 按钮或按“Ctrl+Enter”键，有点像浏览器。

2. 子窗口

在工具栏上单击 按钮或从菜单“View/Open subviews/Disassembly”中打开反汇编子窗口，这样可以用多个子窗口来分析同一段程序，不必来回翻页查看代码了。

3. 导航器

单击菜单“View/Toolbars/Navigation”打开导航器，如图 3.6 所示。各部分含义：Library function 为库函数，Regular function 为规则函数，Instruction 为指令，Data 为数据，Unexplored 为未查过的，External symbol 为外部符号，这样根据需要可快速跳到相关代码部分。

在导航器中执行右键菜单“Zoom in”、“Zoom out”可以调整导航条的显示比例。对于手工分析，导航器的作用非常大，选择适当的倍率可以起到意想不到的效果。

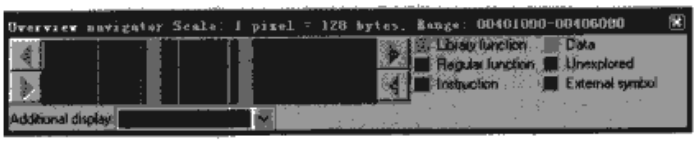


图 3.6 导航器

4. 注释

IDA 可以方便地在代码后面输入注释。在窗口右边空白处单击鼠标右键，显示输入注释的菜单，一种是 Enter comment（快捷键是冒号），另一种是 Enter repeatable comment（快捷键是分号）。按分号键输入的注释，所有交叉参考处都会出现，按冒号键输入的注释只在该处出现。如果一个地址有两种注释，则只显示非重复注释。

5. 提示窗口

IDA 最下面的提示窗口主要反馈各种信息。若同时安装了 IDA 的高低不同版本，有可能导致该窗口消失。解决办法是打开注册表，查找“Datarescue”并删除该主键即可。

3.2.5 交叉参考

通过交叉参考（XREF）可以知道指令代码相互调用的关系。如图 3.7 所示，这句“CODE XREF: sub_401120+B ↑j”，表示该调用地址是 401120，“j”表示跳转（jump）。其他一些符号：“o”表示偏移值（offset），“p”表示子程序（procedure）。双击这里或按回车键可以跳到调用该处的地方。

```
.text:00401165                  loc_401165:                             ; CODE XREF: sub_401120+Bfj
.text:00401165 6A 00                push 0                              ; nExitCode
.text:00401167 FF 15 B0 40 40 00    call ds:PostQuitMessage
```

图 3.7 交叉参考

在 loc_401165 字符上按 X 键，将打开交叉参考窗口，如图 3.8 所示。

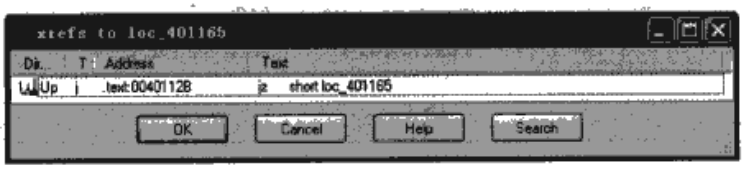


图 3.8 交叉参考窗口

3.2.6 参考重命名

“参考重命名（Renaming of reference）”是 IDA 的一个极好功能，增加了代码的可读性。如图 3.9 所示的这段代码是窗口函数 WndClass 的开始处，IDA 默认用“loc_401120”命名，但“loc_401120”这个字符没有多大意义。

图 3.9 重命名参考点

若加注解，只有这一行才有意义。但用参考重命名功能便可把所有参考点一次改动。在“loc_401120”上单击鼠标右键，弹出右键菜单，在菜单上选择重命名“Rename”，或按 N 键，打开“Rename Address”对话框，如图 3.10 所示。

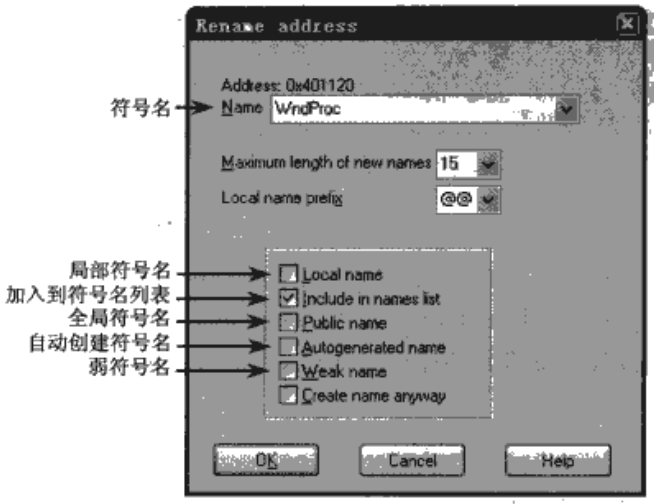


图 3.10　重命名对话框

在此处给它赋予“WndProc”这个有意义的名字。单击“OK”按钮后马上可以看到所有“loc_401120”标签都改变为新名称，如图 3.11 所示。

```
.text:00401120                  WndClass:
.text:00401120 56               push    esi
.text:00401121 57               push    edi
.text:00401122 8B 7C 24 10      mov     edi, [esp+10h]
.text:00401126 8B C7            mov     eax, edi
```

图 3.11　改名后的参考点

3.2.7　标签的用法

在菜单“Jump/Mark position”中打开“标记当前位置”功能，出现如图 3.12 所示的对话框。

图 3.12　标记当前位置

为这个标记（当前光标位置）加上标签，“WndProc”标签便是需要返回的位置；当离开这个标记而返回时，在菜单“Jump/Jump to marked position”中或按“Ctrl+M”键执行“跳到标记位置”功能（见图 3.13）。选择返回的标签，双击就转到指定代码处。

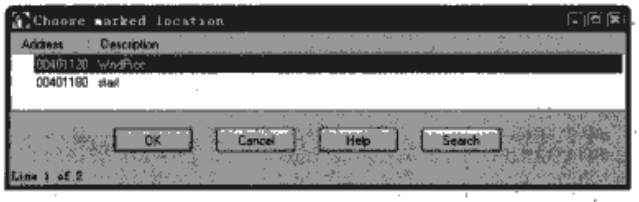


图 3.13　选择标签对话框

3.2.8　进制的转换

IDA 可以提供多种进制显示。先将光标移到需要转换的数据上，单击工具栏上的 按钮（见图 3.14），即可转换为所需要的进制。“Toggle leading zeroes”功能是用 0 填补数据前的空位。

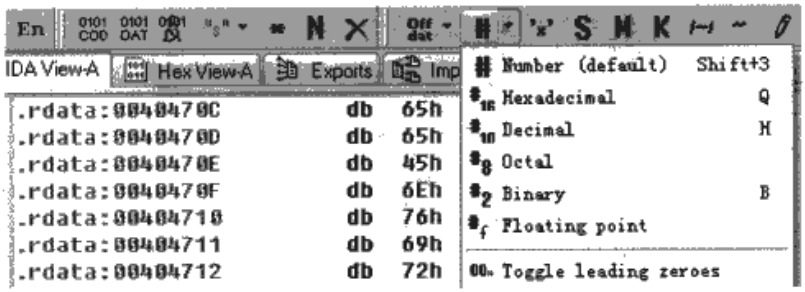


图 3.14 进制转换

3.2.9 代码和数据转换

很多工具在反汇编的时候可能无法正确区分数据和代码，IDA 也不例外。有些程序就是利用这点来对抗静态反汇编的。IDA 的交互性使得用户可以将某段十六进制数指定为代码或数据，即利用人脑来区分代码和数据。

如果确信某段十六进制数据是一段指令，只要将光标移到其第一个字节的偏移位置，执行菜单命令“Edit/Code”或按 C 键。按 P 键可以将某段代码定义为子程序，参数调用会列出。若要取消定义，执行菜单命令“Edit/Undefine”或按 U 键，数据重新以十六进制数据显示。这种交互式分析功能的介入，令 IDA 获得非交互式软件所不能达到的效果。

在代码行按 D 键，数据类型会在 db、dw 与 dd 间转换。当然可以设置更多的数据类型，执行菜单“Options/Setup data types”命令就可以设置，如图 3.15 所示。

如果一个字节已经被转换过，再次转换，IDA 将会提示让你确认，如图 3.16 所示。如果感觉这种提示比较麻烦，可以在“Options/Misc”菜单“Convert already defined bytes”命令里关闭。

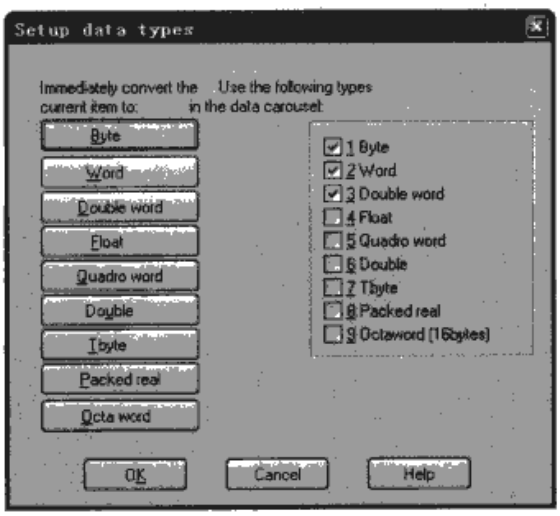


图 3.15 设置数据类型选项

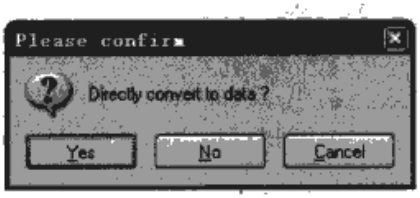


图 3.16 数据转换提示确认框

3.2.10 字符串

编程语言的不同造成字符串也有不同的格式，如以“0”结尾的 C 字符串，以“$”结尾的 DOS 字符串等，IDA 支持所有的格式。如果确信某段十六进制数据是一个字符串，只要将光标移到其第一个字符的偏移位置，执行菜单命令“Edit/Strings/ASCII”或在工具栏上单击 按钮或按 A 键。单击 按钮右边的小三角，会弹出 IDA 所支持的字符串格式，如图 3.17 所示。

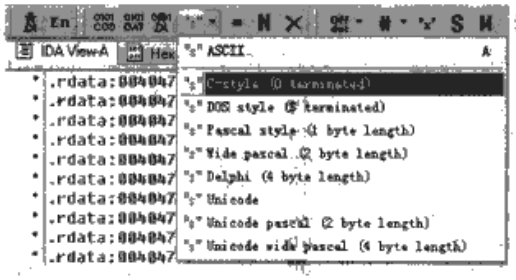


图 3.17　选择字符串类型

按 A 键默认是 C 字符串，也可以在菜单“Options/ASCII string style”中设置其他字符串格式为默认值，如图 3.18 所示。

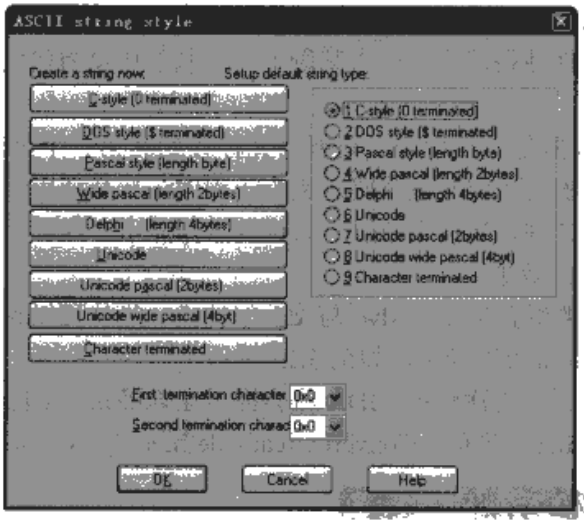


图 3.18　设置默认字符串格式

IDA 有时无法确定 ASCII 字符串，这种错误的发生是由于这个字符串在程序中没有直接被调用到。本例中，按字母 G，输入地址 40478E，会来到如下一段代码处：

```
.rdata:0040478E 47          db  47h ; G
.rdata:0040478F 65          db  65h ; e
.rdata:00404790 74          db  74h ; t
.rdata:00404791 46          db  46h ; F
.rdata:00404792 69          db  69h ; i
.rdata:00404793 6C          db  6Ch ; l
.rdata:00404794 65          db  65h ; e
.rdata:00404795 54          db  54h ; T
.rdata:00404796 79          db  79h ; y
.rdata:00404797 70          db  70h ; p
.rdata:00404798 65          db  65h ; e
.rdata:00404799 00          db    0
```

将光标移到 40478E 行并按 A 键，这句字符串就被定义并生成一个变量名，如要恢复，则按 U 键。IDA 会在生成的字符变量前面加个前缀“a”，如“aGetfiletype db 'GetFileType',0”。

可以在 Names 窗口看到这些字符串变量，只要单击 N 按钮或从菜单“View/Open subviews/Names”中打开这个窗口。

3.2.11 数组

在处理数据时，可以按数组形式显示。先将光标移到需要处理的数据处，选择菜单“Edit/Array”或单击 按钮或按*号键，打开数组排列调整窗口（见图 3.19），此处数据按 4×12 格式排列，见图 3.20。其中，若在“Items on a line”（每行项数）中填 0，则每行项数根据页面自动调整；若想每行显示更多数据，可以在反汇编选项中调整右边距（Right margin）。

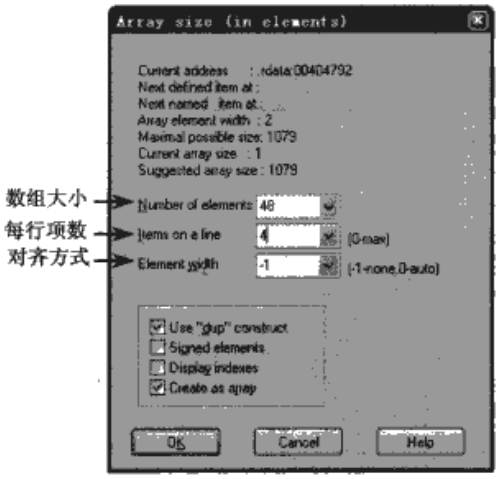


图 3.19 数组排列调整

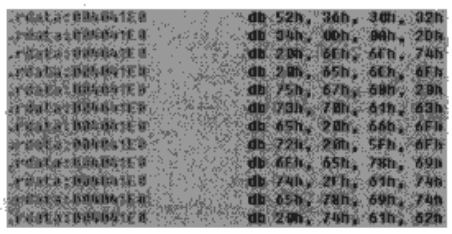


图 3.20 数据按 4×12 排列

3.2.12 结构体

IDA 会根据文件的类型自动加载相应的类型库，如 vc6win(Visual C++ 6.0)，用户做底层分析时，可以增加 mssdk(windows.h)、ntddk(ntddk.h)等。这些类型库中会有相应的结构体，用户分析代码时，可以直接引用。在工具栏上单击 按钮（或按快捷键“Shift+F11”），打开加载类型库窗口（Loaded Type Libraries），如图 3.21 所示。用鼠标右键选择“Load Type Library”（或按快捷键 Insert），在弹出的窗口中选择类型库，如图 3.22 所示。

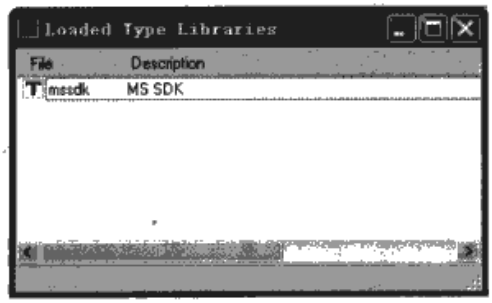


图 3.21 加载类型库窗口

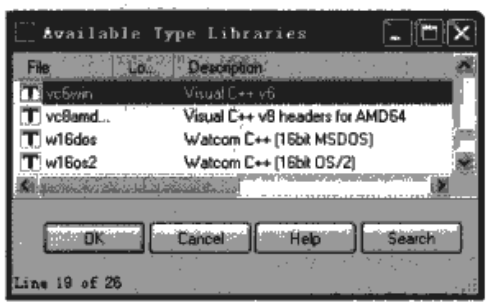


图 3.22 选择类型库

选择好类型库，就可查看内置的结构体数据结构了。选择菜单“View/Open subviews/Structures”或单击工具栏上的 按钮，打开结构体管理窗口。按 Insert 键，在弹出的窗口中单击“Add Standard Structure”，打开添加标准结构库窗口，查找需要的结构名，然后就可以正常使用这些库了。

在默认情况下，IDA 会加载常用的结构。对于 IDA 5.x 版本，在结构体管理窗口里按 Insert 键，再单击 Cancel 按钮，ReverseMe 程序内常用的结构体数据结构会显示出来。在 WNDCLASSA 结构一行按“+”键展开结构，程序代码的相应代码处直接以结构体形式表示，如图 3.23 所示。

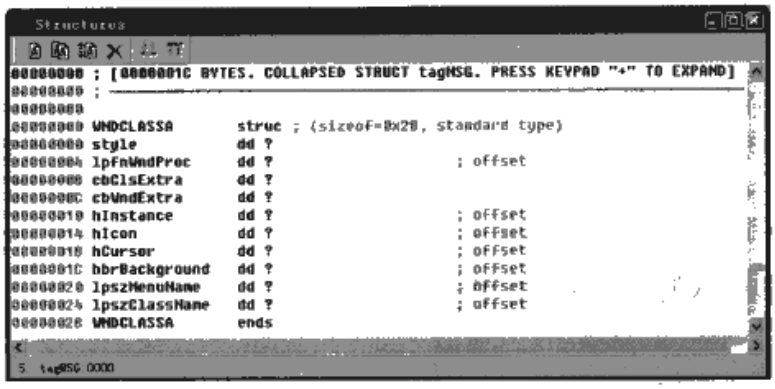


图 3.23　查看结构体

用户也可自定义结构体，例如下面这段 C 程序：

```
//Structures.cpp
struct student
    { int id;
      char name[20];
      int age;
    };
struct student stu[2]={{01,"Mary",14},{02,"Angela",15}};
int main(void)
{
    struct student *p;
    for(p=stu;p<stu+2;p++)
        printf("%5d  %-20s%4d\n",p->id,p->name,p->age);
    return 0;
}
```

如图 3.24 所示的代码是 IDA 在反汇编时由于没有定义结构体而自动生成的。

```
.text:00401000                push    esi
.text:00401001                mov     esi, offset dword_407030
.text:00401006
.text:00401006 loc_401006:
.text:00401006                mov     ecx, [esi+18h]
.text:00401009                mov     edx, [esi]
.text:0040100B                lea     eax, [esi+4]
.text:0040100E                push    ecx
.text:0040100F                push    eax
.text:00401010                push    edx
```

图 3.24　定义结构体前的代码

[esi+18h]等是调用结构体中的数据，在这可以用有含义的名字代替无意义的数字。双击“dword_407030”来到结构体数据处，在这利用 D 键、A 键或 [*] 按钮（数组的项数设置为 20），将数据重新定义一下。结果如下：

```
.data:00407030 dword_407030  dd 1
.data:00407034 aMary         db 'Mary',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
.data:00407048               dd 0Eh
.data:0040704C               dd 2
.data:00407050 aAngela       db 'Angela',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
.data:00407064               dd 0Fh
```

打开结构体窗口，按 Insert 键增加一个新的结构体类型，命名为 student，如图 3.25 所示。

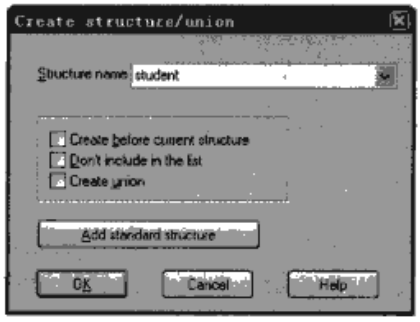


图 3.25 创建新的结构体

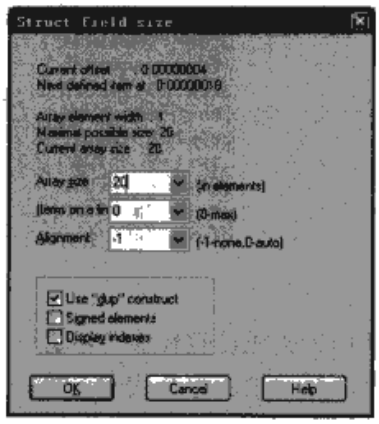


图 3.26 定义结构体中的数组

按 D 键加入数据（如 id、age），重复按 D 键可在 db、dw、dd 间切换，直到变成 dd，表示是 dword 类型。而按 A 键加入的 ASCII 字符（如 name）为结构的成员，此处数组大小填 20，如图 3.26 所示。如果要创建一个大小可变的结构体时，可以将此处自定义的数组元素大小设为 0。当增加新结构成员时，IDA 会自动加以命名，如 field_0 等；可按 N 键修改新结构成员的名字。新定义的结构体如图 3.27 所示。

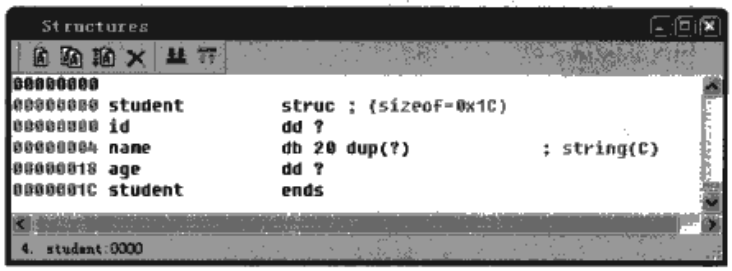


图 3.27 新定义的结构体

现在将鼠标指针放在 407030 这一行上，然后执行菜单“Edit/Structs/Struct var”命令，出现如图 3.28 所示的窗口，以供选择结构体类型。

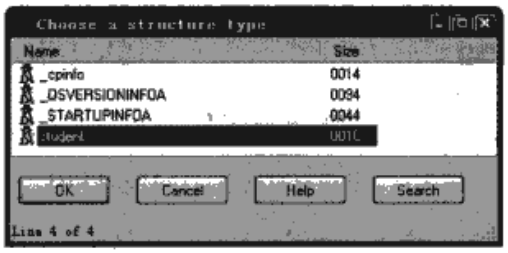


图 3.28 选择结构体类型

选择 student 结构体，单击“OK”按钮即可将数据纠正过来。同样方法，重复执行“Struct var”命令，将 40704C 这行数据转换成 student 类型。转换后的数据如下：

```
.data:00407030 stru_407030      student <1, 'Mary', 0Eh>
.data:0040704C                  student <2, 'Angela', 0Fh>
```

最后，可以在操作数类型中重新定义现有的数据。选中需要重新定义的数据，如[esi+18h]，单击菜单“Edit/Operand types/Offset/Offset(Struct)”或按 T 键执行结构偏移功能，再选择 student 结构体。依次将[esi]、[esi+4]重新定义，最终的效果见图 3.29。

```
.text:00401000                  push    esi
.text:00401001                  mov     esi, offset stru_407030
.text:00401006
.text:00401006 loc_401006:
.text:00401006                  mov     ecx, [esi+student.age]
.text:00401009                  mov     edx, [esi+student.id]
.text:0040100B                  lea     eax, [esi+student.name]
.text:0040100E                  push    ecx
.text:0040100F                  push    eax
.text:00401010                  push    edx
```

图 3.29　用结构体修饰过的代码

如果结构体的成员数较多，不必一个个替换，IDA 提供了批处理操作，可以一次操作就能替换全部。方法是选择需要替换的整个代码，执行“Offset(Struct)”菜单命令或按 T 键，打开结构偏移设置窗口，见图 3.30。右边窗口显示与 ESI 有关系的所有操作。结构各成员前面不同的符号表示经过计算后的状态，“钩”号就表示完全匹配。

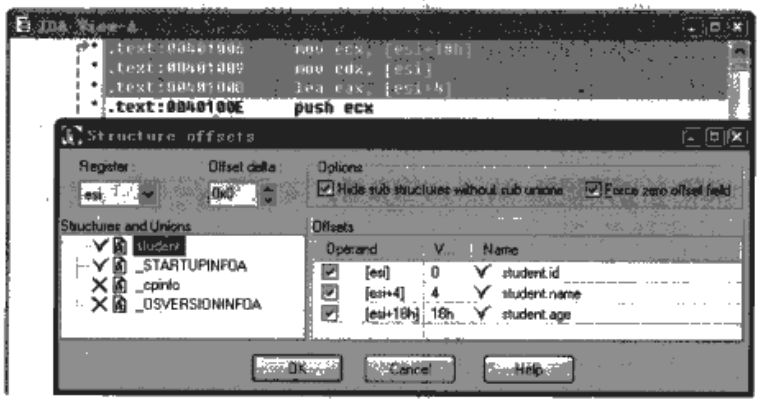


图 3.30　设定批量结构偏移

IDA 还可以从已经分析好的数据中来建立结构体。在地址 407030 处，选择一块已重新组织过的数据，用菜单“Edit/Structs/Create struct from data”命令来创建结构体，如图 3.31 所示。

图 3.31　从现有数据中自动创建结构体

结构体在结构体里又怎样？是的，这是可能的。首先定义结构体，然后在高于这个结构体的另一个级别上，按“Alt + Q”键嵌入另一个实例而成为该结构体的成员。结果如下：

```
;---------------------------------------------------------------
ASampleStructure  struc
Aword                           dw ?
AnArray                         dw 32 dup(?)
AByte                           db ?
field_50                        AnotherOne ?
AsampleStructure                ends
;---------------------------------------------------------------
AnotherOne                      struc
field_0                         db ?
AnotherOne                      ends
```

IDA 中可以像定义标准结构体那样来定义共用体（Union），IDA 认为共用体是一种特殊的结构体，因此其操作方法与结构体差不多。在结构体（Structures）窗口中，按 Insert 键打开创建结构体窗口，见图 3.25，将选项“Create union”选中，即可创建共用体。共用体的偏移量用菜单“Edit/Structs/Select union member”来操作。

IDA 虽然可以使用手工方式建立各类结构，但操作并不方便，从 C 文件头中导入结构是最好的选择。自己建立的头文件通过积累，将来在遇到类似需要的时候，可以快速地导入到 IDA 中。方法是：从菜单“Load file/Parse C header file”加载自定义的头文件，然后就可以在结构体窗口使用导入的结构名了。

3.2.13 枚举类型

可以在反汇编时用 IDA 去动态地定义与操作枚举类型（Enumerated Types）。看看下面这段简单的 C 程序。

```
//Enumerated.cpp
int main(void)
{
  enum weekday {MONDAY, TUESDAY, WEDNESDAY, THUSDAY, FRIDAY, SATURDAY, SUNDAY};
  printf("%d,%d,%d,%d,%d,%d,%d",MONDAY,TUESDAY,WEDNESDAY,THUSDAY,FRIDAY,SATURDAY,SUNDAY);
  return 0;
}
```

用 IDA 反汇编后得到的是一些没有意义的数字，如图 3.32 所示。

```
.text:00401000          push    6
.text:00401002          push    5
.text:00401004          push    4
.text:00401006          push    3
.text:00401008          push    2
.text:0040100A          push    1
.text:0040100C          push    0
.text:0040100E          push    offset aDDDDDDD ; "%d,%d,%d,%d,%d,%d,%d"
.text:00401013          call    sub_401020
.text:00401018          add     esp, 20h
.text:0040101B          xor     eax, eax
.text:0040101D          retn
```

图 3.32 定义枚举前的代码

可以用枚举类型来表示这些数字，执行菜单“View/Open subviews/Enumerations”或单击工具栏上的 En 按钮打开枚举窗口。按 Insert 键插入一个新的枚举类型，取名“weekday”。在新建的 weekday 枚举类型中按 N 键添加枚举成员（见图 3.33），0 对应 MONDAY，1 对应 TUESDAY，依此类推。

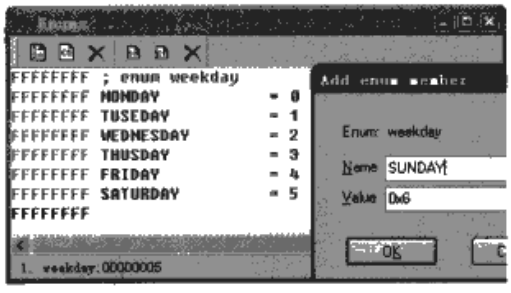


图 3.33 添加枚举成员

最后，可以在操作数类型中重新定义现有的数据。将光标移到需要重新定义的数据处，执行菜单“Edit/Operand types/Enum member”或单击 M 按钮或按 M 键转换成指定的枚举成员，也可以选中数字后，执行右键菜单“Symbolic constant”命令。处理后的代码如图 3.34 所示，IDA 用 MONDAY,TUESDAY 等代替了无意义的数字 0, 1 等，使代码非常易读。

```
.text:00401000          push    SUNDAY
.text:00401002          push    SATURDAY
.text:00401004          push    FRIDAY
.text:00401006          push    THUSDAY
.text:00401008          push    WEDNESDAY
.text:0040100A          push    TUSEDAY
.text:0040100C          push    MONDAY
.text:0040100E          push    offset aDDDDDDD ; "%d,%d,%d,%d,%d,%d,%d"
.text:00401013          call    sub_401020
.text:00401018          add     esp, 20h
.text:0040101B          xor     eax, eax
.text:0040101D          retn
```

图 3.34　直接显示枚举成员

IDA 也支持 Bit-fields，IDA 认为 Bit-fields 是一种特殊的枚举类型。在图 3.35 中创建枚举类型时，将选项“Bitfield”勾上，即可创建 Bit-fields 类型。

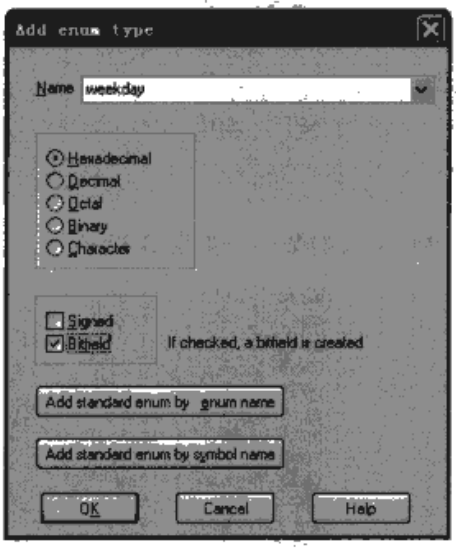


图 3.35　创建一个枚举类型

3.2.14　堆栈变量

先看看 W32Dasm 反汇编 ReverseMe 的一段代码。很明显，下面的这段代码可以改善，参数的传递并不明显，唯一知道的只是一些数据递交给这函数。

```
:004010EF 8D44240C          lea eax, dword ptr [esp+0C]
:004010F3 50                push eax
:004010F4 FFD7              call edi
```

IDA 会自动认出哪些参数放到堆栈中，如下所示：

```
.text:00401000 Msg          = tagMSG ptr -44h
.text:004010EF              lea eax, [esp+50h+Msg]
.text:004010F3              push eax                ; lpMsg
.text:004010F4              call edi                ; TranslateMessage
```

与前面一样，在 IDA 里，堆栈变量也是可以给出有意义的名称的。在任何函数堆栈上（如 Msg 上）双击鼠标左键或按“Ctrl+K”键打开堆栈窗口，将鼠标移到 tagMSG 上，将显示其各结构成员。

3.2.15 IDC 脚本

IDA 能够称之为专业的一个重要因素是 IDC，它是一种嵌入式语言，IDC 的存在极大地提高了 IDA 的扩展性，使得 IDA 中许多重复的任务可以交由 IDC 来完成，在令其自动化的同时，又可对一些特殊情况进行控制。

IDC 本身是一种类 C 的语言的脚本控制器，语法基本与 C 类似，简单易学。所有的 IDC 脚本都有一条包含 idc.idc 文件的语句，其为 IDA 的标准库函数，各函数的含义参考 IDA 帮助文件。变量定义形式为：auto var。其他一些逻辑、循环等语句与 C 类似。

实例 1 查看输入函数

现在编写一个查看输入表的 IDC 脚本程序。通常，某些编译器给输入表所在的区段生成的默认名字是.idata。实际上，输入表可以放在任意一个区段中，.idata 不是输入表的标志，定位输入表的正确方法是根据 PE 文件的结构特征。此例目的是演示脚本编写，就不考虑特殊情况了，假设输入表在.idata 块中。

下面这段 IDC 程序查看 PE 文件的输入函数：

```
//Imports.idc 列出当前程序的输入函数
#include <idc.idc>                                  // 所有的 IDC 脚本都有这一条
static GetImportSeg()
{
    auto ea, next, name;                            // 定义变量
    ea = FirstSeg();                                // 得到第一个段的起始地址
    next = ea;
    while ( (next = NextSeg(next)) != -1) {         // 判断该段是不是 idata 段
        name = SegName(next);
        if ( substr( name, 0, 6 ) == ".idata" ) break;
    }
    return next;
}
static main()
{
    auto BytePtr, EndImports;
    BytePtr = SegStart( GetImportSeg() );           // 确定 idata 段的起始地址
    EndImports = SegEnd( BytePtr );                 // 确定 idata 段的结束地址
    Message(" \n" + "Parsing Import Table...\n");
    while ( BytePtr < EndImports ) {
        if (LineA(BytePtr, 1) != "")                // 判断前一行是否为字符串
            Message("\n" + "____" + LineA(BytePtr,1) + "____" + "\n");
        Message(Name(BytePtr) + "\n");              // 将当前地址的函数名显示出来
        BytePtr = NextAddr(BytePtr);a
    }
    Message("\n" + "Import Table Parsing Complete\n");
}
```

单击菜单“File/IDC file”，打开脚本文件选择窗口，选中“Imports.idc”文件，出现如图 3.36 所示的窗口。单击左边的“Imports”按钮查看 Imports.idc 文件，单击右边的“Imports”按钮执行 IDC 脚本文件。如果打开了多个脚本，将依次排列，在相应按钮上单击右键可删除相应脚本。脚本输出结果在 IDA 提示窗口中。

图 3.36 脚本窗口

实例2 用IDC分析加密代码

一些特殊反汇编任务需要IDC的协助，如对代码段进行加密的程序，可以用IDC先写一段解密的代码，在解密后再反汇编就可以得到正确的反汇编结果。用IDA反汇编光盘映像文件中的encrypted.exe程序。先分析程序入口点处代码：

```
.text:00401020              push    ebp
.text:00401021              mov     ebp, esp
.text:00401023              sub     esp, 8
.text:00401026              call    401080
.text:0040102B              call    401060
.text:00401030              xor     eax, eax
.text:00401032              mov     esp, ebp
.text:00401034              pop     ebp
.text:00401035              retn    10h
```

分析一下“call 401060”呼叫的子程序，对有疑问的地方IDA以红颜色显示。很明显，下面这段代码毫无意义：

```
.text:00401060 6B 31 69          imul    esi, [ecx], 69h
.text:00401063 1D 31 41 01 69    sbb     eax, 69014131h
.text:00401068 01 31             add     [ecx], esi
.text:0040106A 41                inc     ecx
.text:0040106B 01 6B 01          add     [ebx+1], ebp
.text:0040106E FE 14 0D 21       dd 210D14FEh
.text:00401072 41 01             db 41h, 1
.text:00401074 C2 90 90          retn    9090h
```

稍微分析一下，就会发现原来程序是调用“call 401080”子程序对上述代码解密的，解密后才执行。解密代码如下：

```
.text:00401080 B8 60 10 40 00    mov     eax, offset loc_401060
.text:00401085 8A 18             mov     bl, [eax]
.text:00401087 80 F3 01          xor     bl, 1
.text:0040108A 88 18             mov     [eax], bl
.text:0040108C 40                inc     eax
.text:0040108D 3D 74 10 40 00    cmp     eax, 00401074
.text:00401092 7F 02             jg      short locret_401096
.text:00401094 EB EF             jmp     short loc_401085
.text:00401096 C3                retn
```

这段代码利用了SMC技术（Self Modifying Code，自己修改自己的代码），就是在可执行文件中保存着加密的数据。只有程序在运行时，才会由程序在某处由一段还原代码来解密这段加密数据。然后，程序才执行这段还原后的代码。具体过程如下：

```
......
Call ModifyTheProc    // 此子程序的作用就是修改（或称解密）“TheProc”子程序的指令代码
Call TheProc          // 进入，执行已修改的指令代码
......
```

具体来看看这段代码执行。首先指令“mov eax,401060”将待解密数据的首地址放进eax寄存器，接着“mov bl, [eax]”将待解密的数据取一位放进bl，然后对该数据做异或操作（xor bl, 01），指令“inc eax”指向待解密程序的下一个字节，“cmp eax, 00401074”检查待解密的数据是否结束。归纳一下，这段程序的功能就是将40011AB到4011C3之间的数据与1做异或运算，从而还原数据。在此利用一段IDC子程序去

模拟这段解密过程，还原后的数据就是在 IDA 中可以“看”到的真实代码。

```
#include <idc.idc>
//from:解密代码起始地址; size:代码长度; key:此例值为 1
static decrypt(from, size, key ) {
    auto i, x;
    for ( i=0; i < size; i=i+1 ) {
            x = Byte(from);                    // 取得解密数据
            x = (x^key);                       // 异或操作（解密）
            PatchByte(from,x);                 // 将解密数据放回原处
            from = from + 1;                   // 下一个数据
    }
}
```

单击菜单“File/IDC file”将此段程序载入 IDA 中，再单击菜单“File/IDC command”或按“Shift+F2”键打开 IDC 命令执行窗口（见图 3.37），以十六进制形式输入命令“decrypt(0x00401060,0x15,0x1);”。

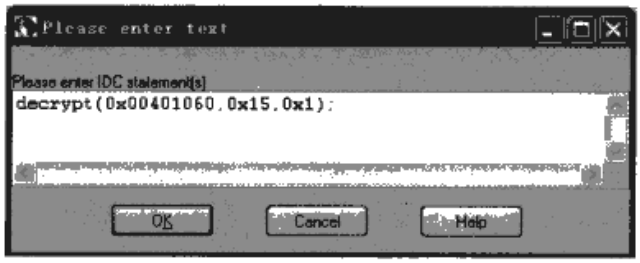


图 3.37 IDC 命令执行窗口

执行完语句后，这组字节就被解码了：

```
.text:00401060 6A 30 68          push    30h
.text:00401063 1C 30 40 00 68    sbb     al, 30h
.text:00401068 00 30             add     [eax], dh
.text:0040106A 40                inc     eax
.text:0040106B 00 6A 00          add     [edx+0], ch
.text:0040106E FF 15 0C 20       dd 200C15FFh
.text:00401072 40 00             db 40h, 0
.text:00401074 C3 90 90          retn
```

最后就是手动通知 IDA 重新分析这段代码。先用 U 命令将所有的代码以数据形式显示出来，然后在 401060 处用 C 命令指示 IDA 将此段代码重新分析。结果如下：

```
.text:00401060 6A 30             push    30h
.text:00401062 68 1C 30 40 00    push    40301Ch
.text:00401067 68 00 30 40 00    push    403000h
.text:0040106C 6A 00             push    0
.text:0040106E FF 15 0C 20 40 00 call    ds:MessageBoxA
.text:00401074 C3                retn
```

遗憾的是，大多数加密程序比一个简单的 xor 更复杂。解决方法是一样的，只是操作更复杂些。

在 IDA 中对于使用 SMC 或其他技术加密代码，也可以使用其他方法将数据解码，如 OllyDbg 动态调试，然后通过 IDA 的“Additional binary file”功能将解码文件重新加载，这样的做法比 IDC 脚本来得更有效。具体使用请参考 13.10“静态脱壳”一节。

IDA 自带的 IDC 脚本语言比较简陋，写起脚本来非常不方便。于是一些爱好者写了专门的脚本开发工具来改善，如 IDApython、IDAperl 等，这些工具功能比较强大，并且有些还支持调试的功能。

3.2.16 FLIRT

IDA 的另一项卓越的能力是库文件快速识别与鉴定技术（Fast Library Identification and Recognition Technology，FLIRT）。这项技术使 IDA 能在一系列编译器的标准库文件里自动找出调用的函数，使反汇编清单清晰明了。

通常的反汇编软件对于各种开发库显得无能为力，只能给出其反汇编结果，而不能给出库函数的名称。例如标准的 C 函数 strlen()，在反汇编中它可能显示为“call 406E40”，这样的反汇编结果虽然是正确的，但却是无意义的。IDA 的 FLIRT 技术可以使反汇编结果中正确标记出所调用的库函数名称，例如以“call strlen”形式显示，这样就极大地提高了反汇编结果的可读性。许多反汇编器都有类似的函数注解功能，但通常限于所调用 DLL 的输出函数。而 IDA 试图扩展到包含尽可能多的函数，如流行的开发库 MFC, OWL, BCL 等。FLAIR 的思想是为每个可标识的库函数创建一个“签名”，以使 IDA 在分析汇编代码时能认知和标记它。

IDA 通常可识别一些编译器，但不一定都会成功，如反汇编一些特定版本编译器产生的程序，像微软的记事本程序。另一种情况导致识别失败的原因是程序里编译器的资料被删除了，如用高级语言编写的病毒程序。最后一种情况是编译器的不支持而失败。

如果 IDA 对程序的编译器支持，但 FLIRT 没有自动识别出，此时可强制使用其编译器特征文件。在这以一个 Delphi 5 写的程序演示一下，Delphi 5 的签名文件 d5vcl.sig 在 DA\SIG 目录中。运行 IDA 打开光盘映像文件中提供的 Delphi5.exe 程序，反汇编后某段代码如下，注意斜体字体代码处。

```
CODE:004416A2                mov     eax, [esi+2D8h]
CODE:004416A8                call    sub_4222DC
CODE:004416AD                cmp     [ebp+var_C], 0
CODE:004416B1                jnz     short loc_4416D1
CODE:004416B3                push    30h
CODE:004416B5                push    offset dword_4417F4
CODE:004416BA                push    offset dword_4417FC
CODE:004416BF                mov     eax, esi
CODE:004416C1                call    sub_4283AC
```

从上面的代码可见，FLIRT 没有起自动识别作用，必须强制使用 Delphi 5 的签名文件 d5vcl。单击菜单“View/Open subviews/Signatures”或按“Shift+F5”键或单击 按钮打开签名窗口，在该子窗口中单击鼠标右键选择“Apply new signature”或按 Insert 键打开库文件列表窗口（见图 3.38），选中“Delphi 5 Visual Component Library”即可激活 Delphi 5 的签名文件。

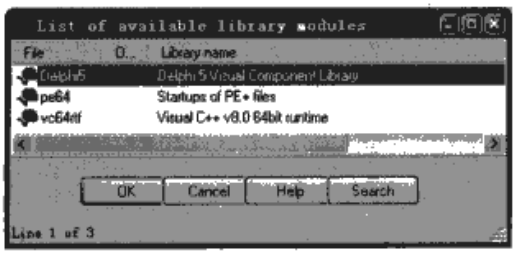


图 3.38 签名列表窗口

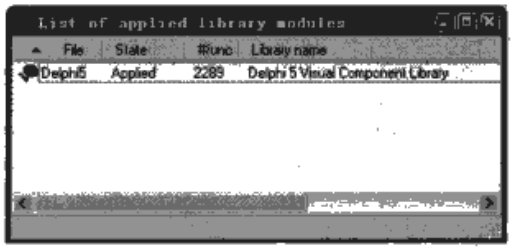


图 3.39 应用了 Delphi 5 签名

应用新的签名文件后，IDA 会自动重新分析全部代码，图 3.39 中的#func 栏的数字会跳动，表示已分析了多少个 Delphi 函数。如果由于某些原因，IDA 没自动分析，也可单击菜单“Options/Analysis”打开分析配置选项，单击“重新分析程序（Reanalyse program）”按钮。所有的函数被确认后，反汇编出来的结果就比较有意义了。重新分析后的代码如下：

```
CODE:004416A2                mov     eax, [esi+2D8h]
```

```
CODE:004416A8          call    @TControl@GetText
CODE:004416AD          cmp     [ebp+var_C], 0
CODE:004416B1          jnz     short loc_4416D1
CODE:004416B3          push    30h
CODE:004416B5          push    offset dword_4417F4
CODE:004416BA          push    offset dword_4417FC
CODE:004416BF          mov     eax, esi
CODE:004416C1          call    @TWinControl@GetHandle
```

IDA 为了方便用户自己制作识别库文件，提供了 FLIRT 数据库生成工具 FLAIR，这将给反汇编工作带来更多的便利。该工具是命令行工具，先用 pcf.exe 将*.lib 生成*.pat 文件。命令如下：

```
pcf *.lib  *.pat
```

再用 sigmake.exe 文件转换成签名文件。命令如下：

```
sigmake *.pat *.sig
```

如果只有 dll 文件，则反汇编该 dll，用 IDB2PAT 插件生成.pat 文件，再用 sigmake 生成.sig 文件。FLAIR 更详细用法，请阅读其自带的帮助文档。

3.2.17 插件

IDA 支持插件模块，从而可以扩展其功能并自动完成某些任务。一些功能 IDC 与插件都能完成，但插件提供了更多的函数，还可使用 IDA SDK 以外的函数，因此可以完成一些复杂的任务。开发插件，需要 IDA 软件开发集成工具包（SDK），可以与 IDA 产品一起获得。

例如 Hex-Rays Decompiler 插件，它是一款将汇编直接反编译成高级语言的插件，功能十分强大，虽然反编译效果还待改进，但已大大提高了代码的可读性。使用很简单，反汇编好目标程序后，按 F5 键即可得到源代码。

3.2.18 其他功能

1. 图形化模式

从 5.0 版本开始，IDA 又新支持了一种全新的分析模式——图形化模式。这种模式比过去的文本模式有更好的可视性，更容易看清函数的代码流程。

在图形化模式下，当前的函数是由节点和流程连线组成的。用户可以通过空格键切换文本模式和图形化模式。由于在图形化模式下，函数的所有代码可能在屏幕中显示不全，IDA 中专门提供了总览窗口（Graph overview），通过单击窗口中感兴趣的位置，可以快速定位该代码片断。用户也可以通过总览窗口了解目前正在分析的局部代码在函数中所处的位置。

2. 加载符号文件

IDA 5.0 以上支持在菜单“Load file/PDB file”里加载 DBG 和 PDB 文件的功能。此项功能对于系统软件而言，非常强大，机器只要联网，IDA 会自动到微软的网站去寻找最适合当前文件版本的 PDB 或 DBG 文件。

3. API 帮助关联

将 idagui.cfg 中的“HELPFILE”指向本地 Windows API 帮助文件即可实现 API 帮助关联。这样，用户在分析文档时，看到不熟悉的 API 时，选中该 API，并按“Ctrl+F1”键，即可获得该 API 的帮助信息。

4．文件的输出

反汇编代码输出功能在菜单“File/Produce file”处。可以按 MAP, ASM, LST, EXE, DIF 等格式输出。

（1）MAP 文件

输出 MAP 文件，MAP 是一个文本文件，记录了各函数等符号信息。

（2）ASM 文件

仅输出汇编代码部分，每行代码前无地址。若仅想输出一段代码，选择要保存的代码，然后执行“File/Create ASM file”即可。

（3）Dump database to IDC file

这个命令将当前 IDA 数据变化记录到 IDC 文件中，以供恢复当前数据使用。每个 IDA 新版本都有它自己的数据格式，互不兼容。利用该命令可将低版本中的工作记录转换到高版本中。首先在低版本中利用此命令输出一个 IDC 脚本文件，然后在 IDA 新版本中重新装载分析原文件，完成后按 F2 键打开该 IDC 脚本执行，结束后，所有的注释、重命名等记录都导入到新版本的 IDA 中。

（4）Dump typeinfo to IDC file

该命令主要将一些用户自定义数据类型保存到 IDC 文件中，如关于结构体、枚举等的用户自定义类型。可以利用此命令将一个程序的数据类型导入到另一个程序中。

3.2.19 小结

IDA 在提供专业的反汇编能力的同时，也提供了很多相当优秀的辅助功能，如制作流程图和动态调试等。动态调试与静态反汇编本身的紧密联系使得 IDA 分析程序更得心应手了。

IDA Pro 是目前最好的反编译器，改变了人们反汇编的方法，它更像一个智能的反汇编工具。IDA 从对程序代码进行反汇编开始，然后分析程序流程、变量和函数调用等。IDA 很难使用，并需要有关程序行为的高级知识，但它的技术层次反映了逆向工程真正的本质特征。IDA 提供了操纵程序特征的完整的 API 调用，因此用户能定性分析。对于 Windows SDK 开发的程序，用 IDA 配合一般都能逆向出源代码，特别是 Hex-Rays Decompiler 插件的出现，使得源代码获得更轻而易举。

IDA 也不是十全十美的，其最大的缺陷是其反汇编的速度，由于各种高级功能的引进，IDA 在反汇编速度一项上，落后其他大多数反汇编工具，其较慢的反汇编速度成为 IDA 被批评的最主要因素。另一个缺陷来自于用户界面，由于 IDA 的强大交互性，导致 IDA 的界面过于专业，大量的窗口、反汇编术语可能导致初级用户无所适从；但反过来也可以说，这样的用户界面又再次加强了 IDA 在专业领域的地位。

3.3 可执行文件的修改

IDA 适合分析文件，若要对文件进行编辑修改，则需要专门的十六进制工具了。常用的有 Hiew, HexWorkshop, WinHex 等。各工具都有其特色，HexWorkshop 提供了文件比较功能；WinHex 可以查看内存映像文件；Hiew 可以在汇编状态下修改代码。上一章的 OllyDbg 也可修改保存代码，并且汇编语法更加强大，因此读者可以跳过本节。

本节主要讲解如何利用 Hiew 修改 PE 文件中的指令代码，其他工具读者可自行摸索操作。

1．安装

将 Hiew 压缩包解压到指定目录中就可完成其安装。为了使用方便，建议配置一下 hiew.ini 文件：

```
StartMode          = Code          ; Text   | Hex      | Code
```

StartMode 项默认是 Text。建议将其设置为 Code，这样一进入 Hiew 就自动切入代码界面。Hiew 是控

制台用户界面，不够友善。运行方式有：

- 将要修改的文件拖到 hiew.exe 上；
- 将 hiew.exe 快捷方式放进 SendTo 目录（推荐）。

由于屏幕属性等原因，Hiew 在某些系统上运行不了。此时打开命令提示符窗口，在“属性/布局/屏幕缓冲区大小”中将高度改成 25。或新建一个 pif 文件，把代码页改成英文，把屏幕高度改成 25。

2. 修改指令的操作

用 Hiew 打开实例文件 ReverseMe.exe。按回车键，将在 3 种模式间循环：文本（Text）、十六进制（Hex）和汇编代码（Decode）。按 F1 键将列出各模式下的帮助清单。

切换到汇编代码模式下，文件以反汇编代码形式显示，用户可以在汇编状态下修改和分析程序，这是 Hiew 的最强大之处。

Hiew 的汇编模式语法如下：

- “byte/word/dword/pword/qword/tbyte”可简写成“b/w/d/p/q/t”；
- 所有的数字都是十六进制，所以“h”操作符是可选的；
- 以 A, B, C, E 或 F 开头的十六进制数据，必须加个前缀 0，如 0A1256 等；
- 无条件跳转指令“jmp xxxx”将转换成“0E9 xx xx…”形式，因此近转移 jmp（0EB）需要按如下形式输入：jmp short xxxxx（或 jmps xxxxx）。远转移的形式是：jmp xxxxx，其中 xxxxx 是文件偏移地址。在这一点上要格外小心，若将近转移写成长指令形式，或将偏移地址写成虚拟地址，当时在 Hiew 可能看不出来，但一执行就会出错。（从这点看出 OllyDbg 的反汇编引擎更强大，自动处理远转移和近转移。）

在此以修改 ReverseMe 为例，讲解一下 Hiew 的文件修改功能。

（1）为 ReverseMe 实例增加水平和垂直滚动条

滚动条显示是由 CreateWindowEx 函数的样式参数（dwStyle）控制的，具体参数可以查 API 手册。用 IDA 反汇编 ReverseMe 后，在输入表窗口查找 CreateWindowExA，双击来到如图 3.40 所示的代码处。

图 3.40 IDC 命令执行窗口

双击 WinMain(x,x,x,x)+AF ↑ r 这里，会来到调用 CreateWindowExA 代码处：

```
.text:00401083 6A 00              push    0               ; lpParam
.text:00401085 56                 push    esi             ; hInstance
.text:00401086 6A 00              push    0               ; hMenu
.text:00401088 6A 00              push    0               ; hWndParent
.text:0040108A 68 00 00 00 80     push    80000000h       ; nHeight
.text:0040108F 68 00 00 00 80     push    80000000h       ; nWidth
.text:00401094 68 00 00 00 80     push    80000000h       ; Y
.text:00401099 68 00 00 00 80     push    80000000h       ; X
.text:0040109E 68 00 00 CF 00     push    0CF0000h        ; dwStyle
.text:004010A3 68 38 50 40 00     push    offset WindowName ; 静态分析
.text:004010A8 68 30 50 40 00     push    offset ClassName ; "chap231"
.text:004010AD 6A 00              push    0               ; dwExStyle
.text:004010AF FF 15 C0 40 40 00  call    ds:CreateWindowExA
```

要显示滚动条，只需给 dwStyle 参数再加上两个值：WS_HSCROLL 和 WS_VSCROLL 即可。查 VC 头文件 WINUSER.H 得知：WS_HSCROLL 定义为 00100000h，WS_VSCROLL 定义为 00200000h。这些参数以逻辑或（OR）做运算（可用 Windows 自带的计算器）：

```
0CF0000h OR 100000h OR 200000h = 00FF0000h
```

经过上面计算得知，将 0040109E 一行指令改为“push 00FF0000”，即可加上滚动条。用 Hiew 打开 ReverseMe，切换到汇编代码模式（见图 3.41），此时主窗口显示的地址是虚拟地址。按 F5 键可跳转到指定的地址，此时输入文件的偏移地址：109E，再按回车键就可跳到 40109E 处。

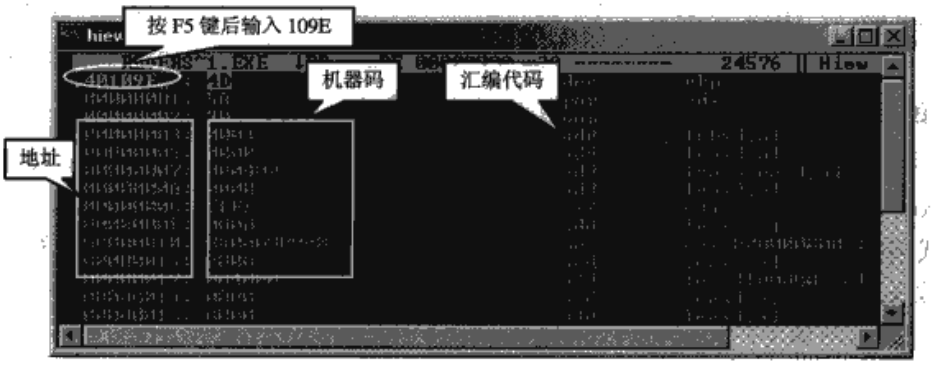


图 3.41　Hiew 运行界面

技巧：按 F5 键后，也可输入虚拟地址，格式是在地址前加个点号“.”，如本例输入“.40109E”。

跳到 40109E 处，按 F3 键进入编辑状态，此时主窗口显示的地址是文件偏移地址。此时可直接修改机器码的数据，也可在指定行上按回车键或按 F2 键修改汇编代码（见图 3.42）。

图 3.42　进入汇编代码编辑状态

输入 push 0FF0000 后（记得 F 前加个 0），按回车键跳到下一行，按 Esc 键返回。确认无误后，按 F9 键存盘。再运行 ReverseMe，出现了水平和垂直滚动条。

（2）数据块加密

此项功能可以对数据或代码进行一些简单的加解密运算。用 Hiew 打开 IDA 一节的 encrypted 程序，在十六进制模式或代码模式下，按 F3 键进入编辑状态，然后按 F7 键进入加密模式界面，再按回车键输入指令（见图 3.43）。数据的运算可以按 byte/word/dword 格式进行，按 F2 键循环切换。

图 3.43　数据加解密操作

指令代码不支持跳转指令，用 loop 指令代替，其含义是“jmp/stop”。rol/ror 指令要求两个操作符大小需相等。32 位寄存器不支持“div”、“mul”指令。AL/AX/EAX 寄存器中存放的是待运算的数据。下面是几个示例。

- 与 01h 字节进行异或运算：

```
1.  XOR  al,01h
2.  LOOP 1
```

- 除以 2：

```
1.  MOV  cl,2
2.  MOV  ah,0
3.  DIV  cl
```

- 计算 ax=(ax×3)/2：

```
1.  MOV  bx,3
2.  MOV  cx,2
3.  MUL  bx              ; 结果保存在 (DX:AX)中
3.  DIV  cx              ; 用 (DX:AX) 除以 CX
```

输入指令后，按 F9 键可将当前的算式保存到文件中，下次需要时按 F10 键调出。然后按 F7 键（Exit）或按 Esc 键回到编辑界面。将光标移到需要修改的数据处，按 F7 键（Crypt），对当前光标所在的数据进行计算。当然，若是简单的 XOR 运算，可直接用 F8 键的 XOR 运算功能。

3 静态分析技术应用实例

学到这里，读者所掌握的理论知识已经可以进行一些简单的应用了，如逆向工程、病毒分析等。

3.4.1 解密初步

现在的软件一般采取人机对话方式，因此从提示信息入手很快就能找到要害。在这里，利用本书光盘映像文件中的一个小程序来讲解。为了叙述方便，将为了练习解密而特别编写的这个程序称为 CrackMe。运行 CrackMe，随意输入几个字符，出现如图 3.44 所示的提示信息。

图 3.44 出错提示信息

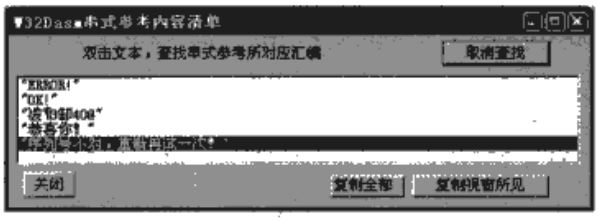


图 3.45 串式参考窗口

用 W32Dasm 或 IDA 对 CrackMe.exe 进行反汇编，在串式数据参考中找提示信息，找到“序列号不对，重新再试一次!”一句，如图 3.45 所示。双击它来到相关代码处：

```
* Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004010C9(C)
|
* Possible StringData Ref from Data Obj ->"ERROR!"
:004010E8 681C304000          push 0040301C
* Possible StringData Ref from Data Obj ->"序列号不对，重新再试一次!"
:004010ED 6800304000          push 00403000
```

```
:004010F2 6A00                  push 00000000
* Reference To: USER32.MessageBoxA, Ord:01BEh
:004010F4 FF1524204000          Call dword ptr [00402024]
```

现在必须从这行起向上找，直到找到 cmp,jne,je,test 等比较、跳转指令。注意这一行代码：

```
 * Referenced by a (U)nconditional or (C)onditional Jump at Address:
|:004010C9(C)
```

004010C9(C) 表示指令由 4010C9 转到此，来到这行代码处：

```
:004010BD 52                    push edx
:004010BE 50                    push eax
* Reference To: KERNEL32.lstrcmpA, Ord:02FCh
:004010BF FF1504204000          Call dword ptr [00402004]
:004010C5 85C0                  test eax, eax
:004010C7 6A00                  push 00000000
:004010C9 751D                  jne 004010E8
* Possible StringData Ref from Data Obj ->"OK!"
:004010CB 6830304000            push 00403030
* Possible StringData Ref from Data Obj ->"恭喜你！"
:004010D0 6824304000            push 00403024
:004010D5 6A00                  push 00000000
* Reference To: USER32.MessageBoxA, Ord:01BEh
:004010D7 FF1524204000          Call dword ptr [00402024]
:004010DD B801000000            mov eax, 00000001
:004010E2 83C414                add esp, 00000014
:004010E5 C21000                ret 0010
```

注意如下几句，比较关键：

```
:004010BF   Call KERNEL32.lstrcmpA          ;调用函数 lstrcmpA 比较真假序列号
:004010C5   test eax, eax                   ;用 eax 当旗帜，如相等，则 eax=0
.....
:004010C9   jne 004010E8                    ;如不跳转则注册成功
```

看明白了吗？要让程序接受任何注册码就只要把 jne（不相等就跳）改成 je（相等就跳），或改成空指令 nop 即可。

启动 Hiew，打开 CrackMe.exe，进入代码模式，按 F5 键输入虚拟地址“.4010C9”（或输入偏移地址 10C9），将指令 jne 004010E8 改成 nop 指令，由于 jne 指令是 2 个字节，因此用 2 个 nop 指令替代。具体如下：

```
.004010C9: 90                   nop
.004010CA: 90                   nop
.004010CB: 6830304000           push   000403030
```

此时输入任何序列号，CrackMe.exe 均提示注册成功。这种跳过算法分析，而直接修改关键跳转指令使程序注册成功的方法，常被解密者称为爆破法。

此例的算法只是将用户输入的序列号与参照值比较，以判断真假。其真正的核心就是一句比较指令：

```
if (lstrcmp(输入的密码,参照值)==0)
        // 密码正确
else
        // 密码错误
```

这种直接比较的程序参照值一般会存在程序中，大多数情况下，编译器会将初始变量放在数据区块（.data 区块）中，用十六进制工具打开文件，跳到.data 块处，会发现一个数字“9981”，这个就是正确的密码，如图 3.46 所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00003000	D0	F2	C1	D0	BA	C5	B2	BB	B6	D4	A3	AC	D6	D8	D0	C2	序列号不对，重新
00003010	D4	D9	CA	D4	D2	BB	B4	CE	A3	A1	00	00	45	52	52	4F	再试一次！..ERRO
00003020	52	21	00	00	B9	A7	CF	B2	C4	E3	A3	A1	00	00	00	00	R!..恭喜你！....
00003030	4F	4B	21	00	39	39	38	31	00	00	00	00	00	00	00	00	OK!.9981........

图 3.46 用十六进制工具查看.data 区块

所以，建议软件开发者在注册码验证过程中，不要让正确的注册码直接出现在程序中。另外，也不要采用明显的提示信息，以便被解密者利用，快速找到判断核心。

3.4.2 逆向工程初步

一般将为了练习逆向工程而特别编写的程序称为 ReverseMe，本例 ReverseMe01 有如下要求：

- 移去“Okay,for now,mission failed”对话框；
- 显示一个 MessageBox 对话框，上面有读者输入的字符；
- 再次显示一个对话框，以告知输入序列号正确还是错误：“Good/Bad serial”；
- 将按钮标题由“Not Reversed”改为“- Reversed -”；
- 使序列号为“pediy”。

1. 移去“mission failed”对话框

用 IDA 打开 ReverseMe01 进行反汇编，查看 Strings 窗口，双击字符“Okay, for now, mission failed !”转到定义此字符串的代码处。再双击后面交叉参考转到调用此字符串的代码处：

```
.text:0040123B  push  0                                ;uType，消息框样式
.text:0040123D        push  offset aReverseme1         ;lpCaption，消息框标题地址
.text:00401242        push  offset aOkayForNowMiss     ;lpText，消息框文本地址
.text:00401247        push  0                          ;hWnd，父窗口句柄
.text:00401249        call  MessageBoxA                ;显示提示窗口
.text:0040124E  push  200h
```

按要求，可以用 nop 指令代替此处的代码，也可用一句跳转指令跳过此提示窗口部分。用 Hiew 打开 ReverseMe01，来到 0040123B 处，将此处改为“jmp 0040124E”（注意是近转移，输入形式是：jmps 64E）。

2. 将输入字符显示到对话框

取得编辑框字符的函数有 GetWindowText、GetDlgItemText 等，可在程序的输入表中查看。在 IDA 中，输入、输出等函数显示在 Name 窗口中，在 Name 窗口中双击 GetWindowTextA 来到调用处，会发现有两处调用了此函数，其中第一处比较可疑：

```
.text:0040120C  mov     eax, 40123B              ;将地址放进 eax
.text:00401211      jmp     eax                  ;跳到 40123B 执行，将其 NOP 掉
.text:00401213      push    200h                 ;复制的最大字符数
.text:00401218  +-- push    4030CC               ;lpString，缓冲区地址
.text:0040121D  |   push    hWnd                 ;文本控件句柄
.text:00401223  |   call    GetWindowTextA       ;得到文本控件内容
.text:00401228  |   push    0                    ;uType
.text:0040122A  |   push    aReverseme1          ;lpCaption，消息框标题地址
.text:0040122F  +-- push    4030CC               ;lpText，消息框文本地址
.text:00401234      push    0                    ;父窗口句柄
.text:00401236      call    MessageBoxA          ;显示文本控件中的内容
```

从上面分析可知，只要将 00401211 指令 NOP 掉（即将机器码改成 9090），程序就可执行这段程序。GetWindowTextA 函数将文本控件内容取出放进缓冲区 4030CC 中，MessageBoxA 函数从该缓冲区中读取文本并显示到消息框中。

3. 修改字符

用 HexWorkshop 或 Hiew 查找字符“Not Reversed”，将其改成“- Reversed -”（不要忘记字符串是以 00h 结尾的）。再将“Good Number”改成“Good Serial”，“Bad Number”改成“Bad Serial”。最后一步，找个空间存放序列号字符：“pediy”，省事的办法是将窗口类名（ClassName）“Supremedickhead”改为“pediy”，其虚拟地址为 403000h。

4. 完成序列号验证

现在开始写检测序列号的代码。一个较好的地方是在 401270h 处，该处是原来的算法代码空间。

```
.text:0040125E      call    GetWindowTextA          ;取得文本控件内容
.text:00401263      mov     ecx, eax                ;文本长度返回到 eax
.text:00401265      xor     edx, edx                ;edx 清零
.text:00401267      or      ebx, 0FFFFFFFFh         ;ebx=-1
.text:0040126A      inc     ebx                     ;ebx=0（即 ebx 清零）
.text:0040126B      mov     eax, 4030CC             ;控件框中的文本地址传给 eax
-------------------------------- 以下用 Hiew 工具输入
.text:00401270 83F905     cmp     ecx, 5            ;检测文本长度是否为 5
.text:00401273 7558       jnz     4012CD            ;如果不正确，跳到出错对话框
.text:00401275 BA0030400  mov     edx, 403000       ;将正确序列号 pediy 地址传给 edx
.text:0040127A 8A18       mov     bl, [eax]         ;将控件框中的文本字符传给 bl
.text:0040127C 8A3A       mov     bh, [edx]         ;将“pediy”字符串传一位字符给 bh
.text:0040127E 3ADF       cmp     bl, bh            ;比较两者是否相等
.text:00401280 754B       jnz     4012CD            ;如果不正确，跳到出错对话框
.text:00401282 40         inc     eax               ;控件框文本的下一个字符
.text:00401283 42         inc     edx               ;“pediy”的下一个字符
.text:00401284 49         dec     ecx               ;计数器减一
.text:00401285 7431       jz      4012B8            ;计数器为 0，序列号正确
.text:00401287 EBF1       jmp     40127A            ;循环，下一个字节检测
```

到现在，大家基本掌握了静态分析相关工具的使用了，但只是第一步，需要掌握一定的逆向分析技能，才能更好地调试分析程序，剖析软件的最深处。

第4章 逆向分析技术

将可执行程序反汇编，通过分析反汇编代码来理解其代码功能，如各接口的数据结构等，然后用高级语言重新描述这段代码，逆向分析原软件的思路。这个过程被称做“逆向工程（Reverse Engineering）”，或者有时只是简单地称作“逆向（Reversing）”。这是一个很重要的技能，需要扎实的编程功底和汇编知识。逆向分析的首选工具就是 IDA，其中它的一款插件 Hex-Rays Decompiler 能完成许多代码反编译的工作，逆向时可以作为一款辅助工具参考。

逆向工程可以让人们了解程序的结构以及程序的逻辑，因此利用逆向工程可以深入洞察程序的运行过程。一般所谓的“软件破解”只是逆向工程中非常初级的一部分。本节探讨的代码分析技术是基于 IA-32 处理器体系结构的。

4.1 启动函数

在编写 Win32 应用程序时，都必须在源码里实现一个 WinMain 函数。但 Windows 程序执行并不是从 WinMain 函数开始的，首先被执行的是启动函数相关代码，这段代码是编译器生成的。启动代码完成初始化进程，再调用 WinMain 函数。

对于 Visual C++程序来说，它调用的是 C/C++运行时启动函数，该函数负责对 C/C++运行库进行初始化。Visual C++配有 C 运行库的源代码，可以在 crt\src\crt0.c 文件中找到启动函数的源代码（安装时 Visual C++必须选取安装源代码选项）；而用于控制台程序的启动代码存放在 crt\src\wincmdln.c 文件中。

所有的 C/C++运行时启动函数的作用基本都是相同的：检索指向新进程的命令行指针，检索指向新进程的环境变量指针，全局变量初始化，内存堆栈初始化等。当所有的初始化操作完毕后，启动函数就调用应用程序的进入点函数。调用 WinMain 函数如下所示：

```
GetStartupInfo (&StartupInfo);
Int nMainRetVal = WinMain(GetModuleHandle(NULL),NULL,pszCommandLineAnsi,(StartupInfo.\
                  dwFlags&STARTF_USESHOWWINDOW)?StartupInfo.wShowWindow:SW__SHOWDEFAULT);
```

当进入点返回时，启动函数便调用 C 运行库的 exit 函数，将返回值（nMainRetVal）传递给它，进行一些必要处理，最后调用系统函数 ExitProcess 退出。

下面是一个 Visual C++编译的程序，程序启动代码的汇编代码如下：

```
00401180    push ebp
00401181    mov ebp, esp
00401183    push FFFFFFFF
00401185    push 004040D0
0040118A    push 00401CB4
```

```
0040118F    mov  eax, dword ptr fs:[00000000]
00401195    push eax
00401196    mov dword ptr fs:[00000000], esp
0040119D    sub esp, 00000058
004011A0    push ebx
004011A1    push esi
004011A2    push edi
004011A3    mov dword ptr [ebp-18], esp
004011A6    Call KERNEL32.GetVersion              ; 确定 Windows 系统版本
......
004011F4    Call KERNEL32.GetCommandLineA         ; 指向进程的完整命令行的指针
......
0040121E    push eax
0040121F    Call KERNEL32.GetStartupInfoA         ; 获取一个进程的启动信息
......
00401241    push esi
00401242    Call KERNEL32.GetModuleHandleA        ; 返回进程地址空间执行文件基地址
00401248    push eax
00401249    call 00401000                         ; 调用用户编写的进入点函数 WinMain
                                                  ; 分析程序时，直接跳到 401000 即可
0040124E    mov dword ptr [ebp-60], eax
00401251    push eax
00401252    call 004012EC                         ; 退出程序
......
0040126A    ret
```

开发人员也可以修改启动源代码，这样会造成即使是同一编译器，而生成的启动代码也会不同。其他一些编译器，如 Delphi、BorLand C++开发包中都有相应的启动代码，感兴趣的读者可以自己研究一下。

注意：在分析程序过程中，启动代码可以略过，直接将重点放到 WinMain 函数体内。

4.2 函 数

程序都是由不同功能的函数（又称子程序、过程或类似概念的东西）组成的，因此逆向分析中将重点放在函数的识别以及参数的传递上是明智的，这样可以将注意力集中在某一段代码上。函数是一个程序模块，用来实现一个特定的功能。一个函数有如下几部分：函数名、入口参数、返回值、函数功能。

4.2.1 函数的识别

程序中通过调用程序调用函数，而在函数执行完后又返回调用程序继续执行。函数是如何知道要返回的地址呢？实际上，调用函数的代码保存了一个返回地址，并连同参数一起传递给被调用的函数。有多种方法实现这个功能，在绝大多数情况下，编译器都使用 CALL 与 RET 指令来调用函数与返回调用位置。

CALL 指令与跳转指令功能类似，所不同的是，CALL 保存返回信息，即将其之后的指令地址压入堆栈的顶部，当遇到 RET 时返回到这个地址处。也就是说，CALL 指令给出的地址就是被调用函数的起始地址，RET 指令结束函数的执行（当然不是所有的 RET 指令都标识函数的结束）。这一机制使得很容易地把函数调用和其他跳转指令区别开来。

于是可以通过定位 CALL 机器指令或利用 RET 指令结束的标志来识别函数。CALL 指令的操作数就是所调用函数的首地址。先看一个例子：

```
int Add(int x,int y);
main( )
{
```

```
    int a=5,b=6;
    Add(a,b);
    return 0;
}

Add(int x,int y)
{
    return(x+y);
}
```

编译结果的大致情况如下：

```
00401000  push    6
00401002  push    5
00401004  call    00401010              ;此处调用 Add()函数，401010 为函数首地址
00401009  add     esp, 8
0040100C  xor     eax, eax
0040100E  retn
0040100F  nop

; 下面是 Add()函数的程序代码
00401010  mov     eax, dword ptr [esp+8]
00401014  mov     ecx, dword ptr [esp+4]
00401018  add     eax, ecx
0040101A  retn                          ;这是 Add 函数的结尾，以 ret 指令结束
```

这种函数直接调用方式，使得程序很简单，所幸大部分情况都是这样的。也有一些情况，程序调用函数是间接调用的，即通过寄存器传递函数地址或动态计算函数地址。例如：

```
CALL [4*eax+10h]
```

4.2.2 函数的参数

函数传递参数有三种方式：堆栈方式、寄存器方式以及通过全局变量进行隐含参数的传递。如果参数是通过堆栈传递的，就需要定义参数在堆栈中的顺序，并约定函数被调用后，由谁来平衡堆栈。如果参数是通过寄存器传递的，就要确定参数存放在哪个寄存器中。每种机制都有其优缺点，并且与使用的编译语言有关。

1. 利用堆栈传递参数

堆栈是一种“后进先出”的存储区，栈顶指针 ESP 指向堆栈中第一个可用的数据项。调用函数时，调用者依次把参数压栈，然后调用函数，函数被调用以后，在堆栈中取得数据，并进行计算。函数计算结束以后，或者调用者，或者函数本身修改堆栈，使堆栈恢复原样（即平衡堆栈）。

在参数传递中，有两个很重要的问题必须得到明确说明：当参数个数多于一个时，按照什么顺序把参数压入堆栈？函数结束后，由谁来平衡堆栈？这些都必须有个约定，这种在程序设计语言中为了实现函数调用而建立的协议称为调用约定（Calling Convention）。这种协议规定了函数中的参数传送方式、参数是否可变和由谁来处理堆栈等问题。不同的语言定义了不同的调用约定，常用的调用约定见表 4-1。

表 4-1 调用约定

约定类型	__cdecl（C 规范）	PASCAL	stdcall	Fastcall
参数传递顺序	从右到左	从左到右	从右到左	使用寄存器和堆栈
平衡堆栈者	调用者	子程序	子程序	子程序
允许使用 VARARG	是	否	是	

注：VARARG 表示参数的个数可以是不确定的；stdcall 如果使用 VARARG 参数类型，则是调用程序平衡堆栈，否则是被调用程序平衡堆栈。

C 规范（即__cdecl）函数参数按照从右到左的顺序入栈，由调用者负责清除堆栈。__cdecl 是 C 和 C++ 程序的默认调用约定。C/C++和 MFC 程序默认使用的调用约定是__cdecl，也可以在函数声明时加上__cdecl 关键字来手工指定。

PASCAL 规范按从左到右的顺序压参数入栈，要求被调用函数负责清除堆栈。

stdcall 调用约定是 Win32 API 函数采用的约定方式，它是“标准调用（Standard CALL）”之意，它采用 C 调用约定的入栈顺序和 PASCAL 调用约定的调整栈指针方式，即函数入口参数按从右到左的顺序入栈，并由被调用的函数在返回前清理传送参数的内存栈，函数参数个数固定。由于函数体本身知道传进来的参数个数，因此被调用的函数可以在返回前用一条“ret n”指令直接清理传递参数的堆栈。在 Win32 API 中，也有一些函数是__cdecl 调用的，如 wsprintf。

为了了解不同类型约定的处理方式，来看这个例子。假设调用函数 test1(Par1, Par2, Par3)，按__cdecl、PASCAL 和__stdcall 的调用约定，其汇编代码如表 4-2 所示。

表 4-2　汇编代码

__cdecl 调用约定	PASCAL 调用约定	__stdcall 调用约定
push par3　;参数按右到左传递	push par1　;参数按左到右传递	push par3　;参数按右到左传递
push par2	push par2	push par2
push par1	push par3	push par1
call test1	call test1　;函数内平衡堆栈	call test1　;函数内平衡堆栈
add esp,0C　;平衡堆栈		

可以清楚地看到，__cdecl 类型和__stdcall 类型是先把右边参数压入堆栈，而 PASCAL 则相反。在堆栈平衡上，__cdecl 类型是调用者用“add esp,0c”指令把 12 个字节参数空间清除，而 PASCAL 和__stdcall 类型则是子程序负责清除。

函数对参数的存取和局部变量都是通过堆栈来定义的，非优化编译器用一个专门的寄存器（通常是 ebp）对参数进行寻址。C、C++、Pascal 等高级语言的函数（子程序）执行过程基本都是一致的。情况如下：

（1）调用者将函数（子程序）执行完毕时应返回的地址、参数压入堆栈；

（2）子程序使用“ebp 指针+偏移量”对堆栈中的参数寻址，并取出，完成操作；

（3）子程序使用 ret 或 retf 指令返回。此时，CPU 将 eip 置为堆栈中保存的地址，并继续予以执行。

堆栈在整个过程中发挥着非常重要的作用。堆栈是一个“先进后出”的区域，堆栈只有一个出口，即当前栈顶，堆栈操作的对象只能是字操作数（占 4 个字节）。例如，按__stdcall 约定调用函数 test2(Par1, Par2)（有 2 个参数），其汇编代码大致情况如下：

```
push par2          ;参数 2
push par1          ;参数 1
call test2         ;调用子程序 test2()
{
        push ebp                          ;保护现场原先的 EBP 指针
        mov  ebp, esp                     ;设置新的 EBP 指针，指向栈顶
        mov  eax, dword ptr [ebp+0C]      ;调用参数 2
        mov  ebx, dword ptr [ebp+08]      ;调用参数 1
        sub  esp, 8                       ;若函数要用局部变量，则要在堆栈中留出点空间
        ...
        add  esp, 8                       ;释放局部变量占用的堆栈
        pop  ebp                          ;恢复现场的 ebp 指针
        ret  8                            ;返回（相当于 ret; add esp,8）
                                          ;ret 后的值是参数个数×4h
}
```

因为 esp 是堆栈指针，所以一般使用 ebp 来存取堆栈。其堆栈建立情况如下：

（1）此例函数中有两个参数，假设执行函数前堆栈指针的 esp 为 K；

（2）根据__stdcall 调用约定，先将参数 Par2 压进栈，此时 esp 为 K-04h；

（3）再将参数 Par1 压进栈，此时 esp 为 K-08h；

（4）参数进栈结束后，程序开始执行 call 指令，call 指令把返回地址压入堆栈，这时候 esp 为 K-0Ch；

（5）这时已经在子程序中了，可以开始使用 EBP 来存取参数了，但为了在返回时恢复 ebp 的值，用“push ebp”保存 ebp 的值，这时 esp 为 K-10h；

（6）再执行一句“mov ebp,esp”，ebp 被用来在堆栈中寻找调用者压入的参数，这时候[ebp + 8]就是参数 1，[ebp + c]就是参数 2；

（7）“sub esp, 8”，在堆栈中定义局部变量，局部变量 1 和 2 对应的地址分别是[ebp-4]和[ebp-8]。函数结束时，调用“add esp,8”释放局部变量占用的堆栈。局部变量的范围从它的定义到它定义所在的代码块的结束为止，也就是说，当函数调用结束后局部变量便消失。

（8）最后调用“ret 8”指令来平衡堆栈，ret 指令后面加一个操作数表示在 ret 后把堆栈指针 esp 加上操作数，完成同样的功能。

处理完毕后，就可以开始用 ebp 存取参数和局部变量了，图 4.1 说明了这个过程。

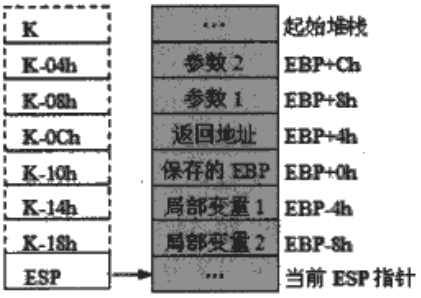


图 4.1　函数堆栈创建图

此外，还有一组指令，即 enter 和 leave，它们可以帮助进行堆栈的维护。enter 语句的作用就是“push ebp/mov ebp,esp/sub esp,xxx”，而 leave 则完成“add esp,xxx/pop ebp”的功能。所以，上面的程序可以改成：

```
enter xxxx,0        ;0 表示创建 xxxx 空间放局部变量
...
leave               ;恢复现场
ret 8               ;返回
```

在许多情况下，编译器会按优化方式编译程序，堆栈寻址稍有不同。这时编译器为了把 ebp 寄存器省下来或尽可能减少代码以提高速度，会直接通过 esp 对参数进行寻址。esp 的值在函数执行期间要发生变化，该变化出现在每次有数据进出堆栈时。要确定是对哪个变量进行寻址，就需要知道程序当前位置的 esp 值是多少，为此必须从函数的开始部分跟踪。

同样，上例中的 test2(Par1, Par2)函数，在 VC 6.0 里，优化选项设置为“Maximize Speed”，重新编译，其汇编代码可能如下：

```
push par2           ;参数 2
push par1           ;参数 1
call test2          ;调用子程序 test2
{
        mov eax, dword ptr [esp+04]        ;调用参数 1
        mov ecx, dword ptr [esp+08]        ;调用参数 2
        ...
        ret 8                              ;返回
}
```

这时程序就用 esp 来传递参数了，其堆栈建立情况如图 4.2 所示。

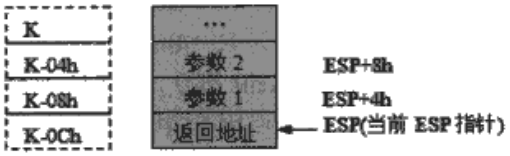


图 4.2　函数堆栈创建图

（1）假设执行函数前堆栈指针 esp 的值为 K；
（2）根据 stdcall 调用约定，先将参数 Par2 压进栈，此时 esp 为 K-04h；
（3）再将 Par1 压进栈，此时 esp 为 K-8h；
（4）参数进栈结束后，程序开始执行 call 指令，call 指令把返回地址压入堆栈，这时候 esp 为 K-0Ch；
（5）这时已经在子程序中了，可以开始使用 esp 来存取参数了。

2. 利用寄存器传递参数

寄存器传递参数的方式并没有一个标准，所有与平台相关的方法都是由编译器开发人员制定的。尽管没有统一的标准，但绝大多数编译器提供商都在不对兼容性声明的情况下，遵循相应的规范，即__fastcall 规范。__fastcall，顾名思义，特点就是快，因为它是靠寄存器来传递参数的。

不同编译器实现的__fastcall 稍有不同，如 Microsoft Visual C++编译器采用__fastcall 规范传递参数时，最左边的两个不大于 4 个字节（DWORD）的参数分别放在 ecx 和 edx 寄存器。当寄存器用完后，就要使用堆栈，其余参数仍然按从右到左的顺序压入堆栈，被调用的函数在返回前清理传送参数的堆栈。浮点值、远指针和__int64 类型总是通过堆栈来传递的。而 Borland Delphi/C++编译器采用__fastcall 规范传递参数时，最左边的三个不大于 4 个字节（DWORD）的参数分别放在 eax、edx 和 ecx 寄存器，寄存器用完后，多余参数按照从左至右的 PASCAL 方式来压栈。另外一款编译器 Watcom C 总是通过寄存器来传递参数的，其严格为每一个参数分配一个寄存器，默认时第一个参数用 eax，第二个参数用 edx，第三个参数用 ebx，第四个参数用 ecx。如寄存器用完，就会用堆栈来传递参数。Watcom C 可以由程序员指定任意一个寄存器传递参数，因此，其参数实际上可能通过任何寄存器进行传递。

来看一个 Microsoft Visual C++ 6.0 编译的__fastcall 调用实例：

```
int __fastcall Add(char,long,int,int);
main(void)
 {
       Add(1,2,3,4);
       return 0;
 }

int __fastcall Add(char a, long b, int c, int d)
{
      return (a + b + c + d);
}
```

用 Visual C++编译，Optimizations 选项为 Default，编译后查看其反汇编代码：

```
00401000  push ebp
00401001  mov  ebp, esp
00401003  push 00000004       ; 后两个参数从右到左入栈，先压入 4
00401005  push 00000003       ; 再将第三个参数数值 3 入栈
00401007  mov  edx, 00000002 ; 将第二个参数数值 2 放入 edx
0040100C  mov  cl, 01         ; 传递第一个参数（字符类型的变量是 8 位大小）
0040100E  call 00401017       ; Add()函数
```

```
00401013  xor     eax, eax
00401015  pop     ebp
00401016  ret

; 下面是Add()函数的程序代码
00401017  push    ebp
00401018  mov     ebp, esp
0040101A  sub     esp, 00000008    ; 为局部变量分配8个字节
0040101D  mov     [ebp-08], edx    ; 第二个参数放到局部变量[ebp-08]中
00401020  mov     [ebp-04], cl     ; 第一个参数放到局部变量[ebp-04]中
00401023  movsx    eax, [ebp-04]   ; 将字符型整数符号扩展为一个双字
00401027  add     eax, [ebp-08]    ; 将左边2个参数相加
0040102A  add     eax, [ebp+08]    ; 再将eax中的结果加上第三个参数
0040102D  add     eax, [ebp+0C]    ; 再将eax中的结果加上第四个参数
00401030  mov     esp, ebp
00401032  pop     ebp
00401033  ret     0008
```

另一个调用规范 thiscall 也用到了寄存器传递参数。thiscall 是 C++中的非静态类成员函数的默认调用约定，对象的每个函数隐含接收 this 参数。采用 thiscall 约定时，函数参数按照从右到左的顺序入栈，被调用的函数在返回前清理传送参数的栈，只是另外通过 ecx 寄存器传送一个额外的参数：this 指针。

定义一个类，并在类中定义一个成员函数：

```
class CSum
{
    public:
    int Add(int a, int b) //实际Add原型具有如下形式:Add(this,int a,int b)
    {
        return (a + b);
    }
};

void main()
{
   CSum sum;
   sum.Add(1, 2);
}
```

用 Visual C++编译，Optimizations 选项为 Default，编译后查看其反汇编代码：

```
:00401000   push    ebp
:00401001   mov     ebp, esp
:00401003   push    ecx
:00401004   push    00000002          ; 第三个参数
:00401006   push    00000001          ; 第二个参数
:00401008   lea ecx, [ebp-04]         ; this指针通过ecx寄存器传递
:0040100B   call    00401020          ; sum.Add(1, 2)
:00401010   mov     esp, ebp
:00401012   pop     ebp
:00401013   ret

; sum.Add()函数实现部分汇编代码
:00401020   push    ebp
:00401021   mov     ebp, esp
:00401023   push    ecx
:00401024   mov     [ebp-04], ecx
:00401027   mov     eax, [ebp+08]
```

```
:0040102A   add     eax, [ebp+0C]
:0040102D   mov     esp, ebp
:0040102F   pop     ebp
:00401030   ret     0008
```

3. 名称修饰约定

在C++中，为了允许操作符重载和函数重载，C++编译器往往按照某种规则改写每一个入口点的符号名，以便允许同一个名字（具有不同的参数类型或者是不同的作用域）有多个用法，而不会打破现有的基于C的链接器。这项技术通常被称为名称改编（Name Mangling）或者名称修饰（Name Decoration）。许多C++编译器厂商选择了自己的名称修饰方案。

在VC++中，函数修饰名由编译类型（C或C++）、函数名、类名、调用约定、返回类型、参数等多种因素共同决定。关于名称修饰东西很多，下面仅简单谈一下常见的C编译、C++编译函数名修饰。

C编译时函数名修饰约定规则：

- __stdcall调用约定在输出函数名前加上一个下画线前缀，后面加上一个“@”符号和其参数的字节数，格式为：_functionname@number。
- __cdecl调用约定仅在输出函数名前加上一个下画线前缀，格式为：_functionname。
- __fastcall调用约定在输出函数名前加上一个“@”符号，后面也是一个“@”符号和其参数的字节数，格式为：@functionname@number。

它们均不改变输出函数名中的字符大小写，这和PASCAL调用约定不同，PASCAL约定输出的函数名无任何修饰且全部大写。

C++编译时函数名修饰约定规则：

- __stdcall调用约定以“?”标识函数名的开始，后跟函数名；函数名后面以“@@YG”标识参数表的开始，后跟参数表；参数表的第一项为该函数的返回值类型，其后依次为参数的数据类型，指针标识在其所指数据类型前；参数表后以“@Z”标识整个名字的结束，如果该函数无参数，则以“Z”标识结束。其格式为：“?functionname@@YG*****@Z”或“?functionname@@YG*XZ”。
- __cdecl调用约定规则同上面的_stdcall调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YA”。
- __fastcall调用约定规则同上面的_stdcall调用约定，只是参数表的开始标识由上面的“@@YG”变为“@@YI”。

4.2.3 函数的返回值

函数被调用执行完后将向调用者返回一个或多个执行结果，称为函数返回值。返回值最常见的方式是用return操作符，其他的还有通过参数按传引用方式返回值，通过全局变量返回值等。

1. 用return操作符返回值

一般情况下，函数的返回值放在eax寄存器中返回，如果处理结果超过了eax寄存器容量，其高32位就会放到edx寄存器中。例如下面这段Pascal程序：

```
function MyAdd1(x, y :integer) : integer;      // 函数声明
var erg: integer;                              // 局部变量
begin
    erg := x + y;                              // 计算
    myadd1 := erg;                             // 返回值
end;
```

这是一个普通的函数，它将两个整数相加。这个函数有两个参数，并用了一个局部变量临时保存结果。其汇编实现代码如下：

Pascal 主程序		MyAdd1 函数	
push x	; 参数 1	push ebp	; 保存 ebp
push y	; 参数 2	mov ebp, esp	; 设置新的 ebp 指针
call MyAdd	; 调用函数	sub esp, 4	; 为局部变量分配空间
	; 堆栈在 MyAdd 函数里平衡	mov ebx, [ebp+0C]	; 取得第一个参数
mov ..., eax	; 返回值在 eax 中	mov ecx, [ebp+8]	; 取得第二个参数
		add ebx, ecx	; 相加
		mov [ebp-4], ebx	; 将结果放到局部变量中
		mov eax, [ebp-4]	; 将局部变量指针返回到 eax
		leave	; 恢复现场
		ret 08	; 平衡堆栈，同时返回

2. 通过参数按传引用方式返回值

给函数传递参数的方式有两种，即传值和传引用。传值调用时是建立参数的一份拷贝并把它传给调用函数，在调用函数中修改参数值的拷贝不影响原始的变量值。传引用调用允许调用函数修改原始变量的值。在调用某个函数时，当把变量的地址传递给函数时，可以在函数中用间接引用运算符修改调用函数中内存单元中的该变量的值。例如下面这个例子，在调用函数 max 时，需要用两个地址（或两个指向整数的指针）作为参数，函数会将结果较大的数放到参数 a 所在的内存单元地址中返回。

```
#include <stdio.h>
void max(int *a, int *b);

main( )
{
    int a=5,b=6;
    max(&a, &b);
    printf("a、b中较大的数是%d",a); //将最大的数显示出来
}

void max( int *a, int *b)
{
    if(*a < *b)
        *a=*b;        //经比较后，将较大的数放到 a 变量之中
}
```

其可能的汇编代码如下：

```
                                        ; 设此时的 esp=k
00401000    sub     esp, 00000008       ; 为局部变量分配内存
00401003    lea     eax, dword ptr[esp+04]  ; eax 指向变量，值为 k-4
00401007    lea     ecx, dword ptr[esp]     ; ecx 指向变量，值为 k-8
0040100B    push    eax                 ; 指向参数 b 的字符指针入栈
0040100C    push    ecx                 ; 指向参数 a 的字符指针入栈
0040100D    mov     [esp+08], 00000005  ; esp+08 的值为 k-8，参数 a 的值放入
00401015    mov     [esp+0C], 00000006  ; esp+0C 的值为 k-4，参数 b 的值放入
0040101D    call    00401040            ; max(&a, &b)
00401022    mov     edx, [esp+08]       ; 利用变量[esp+08]返回函数值
00401026    push    edx
00401027    push    00407030
0040102C    call    00401060            ; printf 函数
00401031    xor     eax, eax
00401033    add     esp, 18
00401036    retn

; max(&a, &b)函数的汇编代码
```

```
00401040    mov     eax, dword ptr [esp+08]  ; 执行后，eax 就是指向参数 b 的指针
00401044    mov     ecx, dword ptr [esp+04]  ; 执行后，ecx 就是指向参数 a 的指针
00401048    mov     eax, dword ptr [eax]     ; 将参数 b 的值加载到寄存器 eax 中
0040104A    mov     edx, dword ptr [ecx]     ; 将参数 b 的值加载到寄存器 edx 中
0040104C    cmp     edx, eax                 ; 比较参数 a、b 的大小
0040104E    jge     00401052                 ; 若 a < b 则不跳转
00401050    mov     dword ptr [ecx], eax     ; 将较大的数放到参数 a 所指的数据区
00401052    ret
```

4.3 数据结构

数据结构是计算机存储、组织数据的方式。逆向分析时，确定了数据结构后，算法就容易得到了。有些时候事情也会反过来，根据特定算法来判断数据结构。本节讨论常见的数据结构以及它们在汇编语言中的实现方式。

4.3.1 局部变量

局部变量是一个函数内部定义的变量，只有在函数内才能使用，如计数器，临时变量等。使用局部变量带来的好处，使程序模块化封装变得可能。从汇编角度来看，局部变量就是在堆栈中进行分配，函数执行完毕后释放这些堆栈，或者直接把局部变量放在寄存器中。

1. 利用堆栈存放局部变量

局部变量在堆栈中具体形成过程请参看图 4.1，程序用“sub esp, 8”语句为局部变量分配空间，用[ebp-xxxx]寻址调用这些变量，而参数调用相对于 ebp 偏移量是正的，即[ebp+xxxx]，因此在逆向时比较容易区分开。编译器在优化模式时，则通过 esp 寄存器直接对局部变量与参数寻址了。当函数退出时，用“add esp, 8”指令平衡堆栈，以释放局部变量所占据的内存。有些编译器，如 Delphi 通过给 esp 加一个负值来进行内存分配。另外，编译器可能会用“push reg”指令来取代“sub esp,4”指令，以节省几个字节。局部变量分配堆栈几种形式见表 4-3。

表 4-3 局部变量分配与清除堆栈几种形式

形式一	形式二	形式三
sub esp, n	add esp,-n	push reg
…	…	…
add esp,n	sub esp,-n	pop reg

来看下面这个实例是如何用“push reg”指令来取代“sub esp,4”指令的。

```
int add(int x,int y);

int main(void)
{
    int a=5,b=6; // 声明局部变量 a、b，同时对变量初始化
    add(a,b);
    return 0;
}

int add(int x,int y)
{
    int z;  // 声明局部变量 z
    z=x+y;
```

```
    return(z);
}
```

用 Microsoft Visual C++ 6.0 编译，不优化，其汇编代码如下：

```
; 这段是main()主函数
00401000    push ebp
00401001    mov ebp, esp
00401003    sub esp, 00000008              ; 为局部变量分配内存
00401006    mov [ebp-04], 00000005         ; 参数1放进局部变量[ebp-04]中
0040100D    mov [ebp-08], 00000006         ; 参数2放进局部变量[ebp-08]中
00401014    mov eax, dword ptr [ebp-08]
00401017    push eax
00401018    mov ecx, dword ptr [ebp-04]
0040101B    puash ecx
0040101C    call 0040102A
00401021    add esp, 00000008
00401024    xor eax, eax
00401026    mov esp, ebp
00401028    pop ebp
00401029    ret

; 函数add(int x,int y)
0040102A    push ebp
0040102B    mov ebp, esp
0040102D    push ecx                       ; 为局部变量分配内存（相当于sub esp,4）
0040102E    mov eax, dword ptr [ebp+08]    ; 取参数1
00401031    add eax, dword ptr [ebp+0C]
00401034    mov dword ptr [ebp-04], eax    ; 将a+b的值放到局部变量[ebp-04]
00401037    mov eax, dword ptr [ebp-04]
0040103A    mov esp, ebp
0040103C    pop ebp
0040103D    ret
```

在函数 add()里不存在“sub esp,n”之类指令，程序是通过一句“push ecx”指令来开辟一块堆栈的，然后用[ebp-04]来访问这个局部变量。

局部变量的起始值是随机的，是其他函数执行后留在堆栈中的垃圾数据，因此需要对其初始化。初始化局部变量有两种方法：一种是通过 mov 指令为变量赋值，如“mov [ebp-04], 5”；另一种是使用 push 指令直接将值压入堆栈，如“push 5”。

2. 利用寄存器存放局部变量

除了堆栈占用了 2 个寄存器外，编译器会利用剩下的 6 个通用寄存器尽可能有效地存放局部变量，这样可以产生最小的代码，提高程序效率。如果寄存器不够用，编译就会将变量放到堆栈中。逆向分析时，要注意局部变量的生存周期比较短，必须及时确定当前寄存器的变量是哪个变量。

4.3.2 全局变量

全局变量作用于整个程序，一直存在，它放在全局变量的内存区；而局部变量则是存在于函数的堆栈区，当函数调用结束后便消失。在大多数程序中，常数一般放在全局变量中，如一些注册版标记、测试版标记等。

在大多数情况下，在汇编代码中识别全局变量比其他结构要容易得多。全局变量通常位于数据区块（.data）的一个固定地址上，当程序需要访问全局变量时，一般会用一个固定的硬编码的地址直接对内存寻址。一个样例如下：

```
mov eax, dword ptr [4084C0h]      ;直接调用全局变量，其中4084C0h是全局变量的地址
```

全局变量可以被同一文件中所有的函数修改，某个函数改变了全局变量的值，就能影响到其他函数，相当于各个函数间的传递通道。因此就可以利用全局变量传递参数、传递函数返回值等。全局变量在程序的全部执行过程中都占用内存单元，而不像局部变量需要时开辟单元。

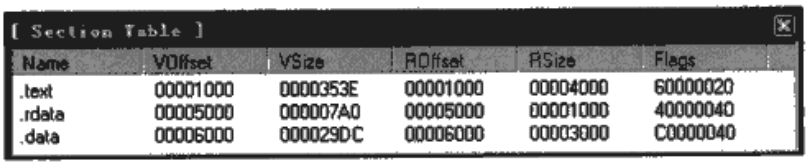
来看一个利用全局变量传递参数的实例，具体代码如下：

```
int z;                    // 全局变量z
int add(int x,int y);

int main(void)
 {
    int a=5,b=6;
    z=7;
    add(a,b);
    return 0;
 }

 int add(int x,int y)
 {
    return(x+y+z);
 }
```

用 Microsoft Visual C++ 6.0 编译，不优化，其汇编代码如下：

```
00401000    push    ebp
00401001    mov     ebp, esp
00401003    sub     esp, 00000008
00401006    mov     [ebp-04], 00000005          ; [ebp-04]是局部变量，放参数1
0040100D    mov     [ebp-08], 00000006          ; [ebp-08]是局部变量，放参数2
00401014    mov     dword ptr [004084C0], 07    ; 对全局变量[004084C0]初始化
0040101E    mov     eax, dword ptr [ebp-08]
00401021    push    eax
00401022    mov     ecx, dword ptr [ebp-04]
00401025    push    ecx
00401026    call    00401034
0040102B    add     esp, 00000008
0040102E    xor     eax, eax
00401030    mov     esp, ebp
00401032    pop     ebp
00401033    ret

; add(x,y)函数代码:
00401034    push    ebp
00401035    mov     ebp, esp
00401037    mov     eax, dword ptr [ebp+08]     ; [ebp+08]为参数1
0040103A    add     eax, dword ptr [ebp+0C]     ; [ebp+0C]为参数2
0040103D    add     eax, dword ptr [004084C0]   ; 调用了全局变量[004084C0]
00401043    pop     ebp
00401044    ret
```

用 LordPE 打开编译后的程序，查看区块，区块信息如图 4.3 所示。会发现全局变量 4084C0h 在.data 区块，该区块的属性可读写。



[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	0000353E	00001000	00004000	60000020
.rdata	00005000	000007A0	00005000	00001000	40000040
.data	00006000	000029DC	00006000	00003000	C0000040

图 4.3　区块信息

这种对内存直接寻址的硬编码方式，比较容易识别出这是一个全局变量。一般编译器会将全局变量放到可读写的区块里，如果放到只读区块里，那么这是一个常量。另外，与全局变量类似的是静态变量，都可以按直接方式寻址等。所不同的是，静态变量作用范围是有限的，仅在定义这些变量的函数内有效。

4.3.3　数组

数组是相同数据类型的元素的集合，它们在内存中按照顺序连续存放在一起。汇编状态下访问数组一般是基址加上某变量来实现的。

请看下面这个数组访问的实例：

```
int main(void)
{
 static int a[3]={0x11,0x22,0x33};
 int i,s=0,b[3];

for(i=0;i<3;i++)
{
    s=s+a[i];
    b[i]=s;
}

for(i=0;i<3;i++)
{
    printf("%d\n",b[i]);
}

    return 0;
}
```

用 Microsoft Visual C++ 6.0 编译，优化选项设置为 Maximize Speed，其汇编代码如下：

```
00401000  sub    esp, 0C                          ;为局部变量分配内存，用来存放 b[i]
00401003  xor    ecx, ecx                         ;s=0
00401005  xor    eax, eax                         ;i=0
00401007  push   esi
00401008  push   edi
00401009  /mov   edi, dword ptr [eax+407030]      ;407030h 指向数组 a[]，即数组的基址
0040100F  |add   eax, 4                           ;访问数组的索引
00401012  |add   ecx, edi                         ;s=s+a[i]
00401014  |cmp   eax, 0C
00401017  |mov   dword ptr [esp+eax+4], ecx       ;b[i]=s;
0040101B  \jl    short 00401009
0040101D  lea    esi, dword ptr [esp+8]
00401021  mov    edi, 3                           ;计数器
00401026  /mov   eax, dword ptr [esi]             ;ESI 指向 b[]数组
00401028  |push  eax
00401029  |push  40703C
```

```
0040102E  |call  00401050                    ;printf("%d\n",b[i]);
00401033  |add   esp, 8
00401036  |add   esi, 4                      ;指向数组下一元素
00401039  |dec   edi
0040103A  \jnz   short 00401026
```

数组在内存中可以存在于堆栈、数据段以及动态内存中。本例中的 a[]数组就保存在数据段.data 中，其寻址用“基址+偏移量”来实现。

```
mov eax, [407030h + eax]
          |       |
        基址     偏移量
```

这种间接寻址一般出现在给一些数组或结构赋值情况下，其寻址形式一般是[基址+n]，其中基址可以是常量，也可以是寄存器，为定值。随着 n 值的不同，就可对结构中相应单元赋值了。

b[]数组放在堆栈中，这些堆栈是编译时刻进行分配的。数组在声明时可以直接计算偏移地址，针对数组成员寻址是采用实际的偏移量完成的。

4.4 虚函数

C++是一门支持 OO 的语言，对面向对象的软件开发提供了丰富的语言支持。但要高效、正确地使用 C++中的继承、多态等语言特性，必须对这些特性的底层实现有一定的了解。

其实就核心概念而言，C++的对象模型的核心概念并不多，最核心的是虚函数。虚函数是在程序运行时刻定义的函数，虚函数的地址是不能在编译时刻确定的，它只能在调用即将进行之前加以确定。对所有虚函数引用通常都放在一个专用数组——虚函数表（Virtual Table，VTBL）中，每个至少使用一个虚函数的对象里面都具有的虚函数表指针（Virtual Table Pointer，VPTR)。虚函数通常通过指向虚函数表的指针间接地加以调用。

将实例 thiscall.exe 的普通成员函数改变为虚函数调用，来看看 VC 在虚函数上面是如何处理的。

```
class CSum
{
  public:
    virtual int Add(int a, int b)
    {
        return (a + b);
    }
    virtual int Sub(int a, int b )
    {
        return (a - b);
    }
};

void main()
{
    CSum*   pCSum = new CSum ;
    pCSum->Add(1,2);
    pCSum->Sub(1,2);
}
```

用 Microsoft Visual C++ 6.0 编译，编译时设置优化选项为 Maximize Speed。其汇编代码如下：

```
00401000  push   esi
00401001  push   4
```

```
00401003  call   00401060                ;new(),为新建对象实例分配 4 字节内存
00401008  add    esp, 4
0040100B  test   eax, eax
0040100D  je     short 00401019
0040100F  mov    dword ptr [eax], 4050A0  ;将 4050A0h 写入创建的对象实例之中
                                          ;4050A0h 是 CSum 类虚函数表的指针
                                          ;(VTBL)，表中的元素是 CSum 类的虚函
                                          ;数，它们指向 CSum 函数的成员
00401015  mov    esi, eax                 ;ESI=VTBL
00401017  jmp    short 0040101B
00401019  xor    esi, esi                 ;用 NULL 指向对象实例指针，该分支在内
                                          ;存分配失败才会来到，空指针将激活 SEH
0040101B  mov    eax, dword ptr [esi]     ;EAX=*VTBL=**Add()
0040101D  push   2
0040101F  push   1
00401021  mov    ecx, esi                 ;ECX=this
00401023  call   dword ptr [eax]          ;对虚函数的调用，此时
                                          ;EAX=*VTBL=**Add()
                                          ;即 CALL 401040
00401025  mov    edx, dword ptr [esi]
00401027  push   2
00401029  push   1
0040102B  mov    ecx, esi                 ;ECX=this
0040102D  call   dword ptr [edx+4]        ;CALL[VTBL+4]
00401030  pop    esi
00401031  retn
```

这段代码首先调用 new 函数分配 class 所需的内存(new 函数的确定是用 IDA 来识别的)，调用成功后，eax 保存了分配的内存的指针，然后将对象实例指向 CSum 类虚函数表（VTBL）4050A0h。查看 4050A0h 数据，如图 4.4 所示。

图 4.4　查看 VTBL

VTBL 里有两组数据：

```
[VTBL]=401040h
[VTBL+4]=401050h
```

进一步看看这两个指针是什么内容，401040h 内容如下：

```
; Add()函数
00401040   mov    eax, dword ptr [esp+8]
00401044   mov    ecx, dword ptr [esp+4]
00401048   add    eax, ecx
0040104A   retn   8
```

401050h 内容如下：

```
; Sub()函数
00401050   mov    eax, dword ptr [esp+4]
00401054   mov    ecx, dword ptr [esp+8]
00401058   sub    eax, ecx
0040105A   retn   8
```

原来虚函数是通过指向虚函数表的指针间接地加以调用的，程序仍然用 ecx 作为 this 指针的载体传递

给虚成员函数，并且利用两次间接寻址，得到虚函数的正确地址，从而执行。

```
0040101B  mov    eax, dword ptr [esi]     ;EAX=*VTBL=**Add()
0040101D  push   2
0040101F  push   1
00401021  mov    ecx, esi                 ;ECX=this
00401023  call   dword ptr [eax]          ;pCSum->Add(1,2)
```

整个过程如图 4.5 所示。VPTR 是一个虚函数表指针，所有的虚函数的入口点被排成一个表，就是虚函数表（VTBL）。

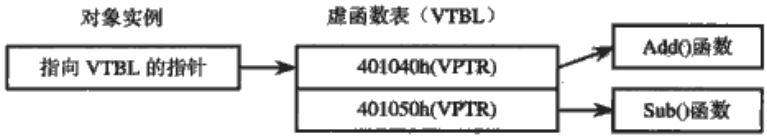


图 4.5　虚函数的调用实现

4.5 控制语句

在高级语言中用 IF-THEN-ELSE、SWITCH-CASE 等语句来构成程序的判断流程，不仅条理清楚，并且维护性好。而其汇编代码则复杂得多，会看到 cmp 等指令的后面跟着各类跳转指令 jz, jnz 等。识别关键跳转是软件解密的一个重要技能，许多软件用一个或多个跳转来实现注册或非注册功能。

4.5.1 IF-THEN-ELSE 语句

将语句 IF-THEN-ELSE 编译成汇编代码后，整数用 cmp 指令比较，而浮点值用 fcom、fcomp 等比较。

语句 IF-THEN-ELSE 编译后，其汇编代码形式一般如下：

```
cmp a,b
jz(jnz) xxxx
```

cmp 指令不修改操作数，根据两个操作数的相减结果，影响处理的几个标志，如零标志、进位标志、符号标志和溢出标志。jz 等指令就是条件跳转指令，根据 a、b 的值大小决定跳转方向，更多条件指令见表 4-4。

实际上，许多情况下编译器都用 test 或 or 之类的较短的逻辑指令来替换 cmp 指令。一般形式为“test eax,eax”，如 eax 为 0，则其逻辑与运算结果为零，就设置 ZF 为 1；否则设为 0。

来看一个实例，源码如下：

```
#include <stdio.h>
int main(void)
{
    int a,b=5;
    scanf("%d",&a);

     if(a==0)
         a=8;
     return a+b;
}
```

用 Microsoft Visual C++ 6.0 编译，优化选项设置为 Maximize Speed，其汇编代码如下：

```
:00401000   push    ecx                    ; 为局部变量分配内存，相当于 sub esp,4
```

```
:00401001   lea     eax, dword ptr [esp] ; eax指向局部变量空间
:00401005   push    eax
:00401006   push    00407030             ; 指向字符串"%d"
:0040100B   call    00401030             ; C语言的scanf函数
:00401010   mov     eax,dword ptr[esp+8] ; 将输入的字符传出
:00401014   add     esp,00000008         ; 由于是__cdecl调用，函数外平衡堆栈
:00401017   test    eax,eax              ; 若eax为0，则ZF置1，否则ZF置0
:00401019   jne     00401020             ; 若ZF=1不跳转，否则跳转
:0040101B   mov     eax,00000008
:00401020   add     eax,00000005
:00401023   pop     ecx                  ; 释放局部变量用到的内存，相当于add esp,4
:00401024   ret
```

4.5.2 SWITCH-CASE 语句

SWITCH 语句是多分支选择语句。SWITCH 语句编译后，实质就是多个 IF-THEN 语句嵌套组合。编译器会将 SWITCH 编译成一组不同关系运算组成的语句。具体来看一例子：

```
#include <stdio.h>
int main(void)
 {
    int a;
    scanf("%d",&a);
    switch(a)
     {
        case 1  :printf("a=1");
                 break;
        case 2  :printf("a=2");
                 break;
        case 10 :printf("a=10");
                 break;
        default :printf("a=default");
                 break;
        }
     return 0;
 }
```

用 Microsoft Visual C++ 6.0 编译，没有优化，其反汇编代码如下：

```
:00401000   push    ebp
:00401001   mov     ebp, esp
:00401003   sub     esp, 00000008    ; 为局部变量分配内存
:00401006   lea     eax, [ebp-04]
:00401009   push    eax
:0040100A   push    00408030             ; 指向字符"%d"
:0040100F   call    004010A2             ; scanf("%d",&a)
:00401014   add     esp, 00000008
:00401017   mov     ecx, [ebp-04]        ; 输入的结果传给ecx
:0040101A   mov     [ebp-08], ecx
:0040101D   cmp     [ebp-08], 01         ; case 1
:00401021   je      00401031
:00401023   cmp     [ebp-08], 02         ; case 2
:00401027   je      00401040
:00401029   cmp     [ebp-08], 0A         ; case 10
:0040102D   je      0040104F
:0040102F   jmp     0040105E
```

```
:00401031   push    00408034
:00401036   call    00401071            ; printf("a=1")
:0040103B   add     esp, 00000004
:0040103E   jmp     0040106B
:00401040   push    00408038
:00401045   call    00401071            ;printf("a=2")
:0040104A   add     esp, 00000004
:0040104D   jmp     0040106B
:0040104F   push    0040803C
:00401054   call    00401071            ;printf("a=10")
:00401059   add     esp, 00000004
:0040105C   jmp     0040106B
:0040105E   push    00408044
:00401063   call    00401071            ;printf("a=default")
:00401068   add     esp, 00000004
:0040106B   xor     eax, eax
:0040106D   mov     esp, ebp
:0040106F   pop     ebp
:00401070   ret
```

如果编译时设置优化选项为 Maximize Speed，其汇编代码如下：

```
:00401000   push    ecx                 ; 为局部变量分配内存，相当于 sub esp,4
:00401001   lea     eax,[esp]
:00401005   push    eax
:00401006   push    0040804C
:0040100B   call    004010A1            ; scanf("%d",&a)
:00401010   mov     eax, [esp+08]       ; scanf 输入的结果传给 eax
:00401014   add     esp, 00000008
:00401017   dec     eax                 ; 检查 EAX 是否为 1，如是下句就跳转
:00401018   je      00401055            ; 相当于 case 1
:0040101A   dec     eax                 ; EAX 再减 1，即 EAX 原来的值是 2
:0040101B   je      00401044            ; 相当于 case 2
:0040101D   sub     eax, 00000008       ; EAX 两次减 1 后的值为 8，所以原值为 10
:00401020   je      00401033            ; 相当于 case 10
```

编译器优化时用“dec eax”指令代替 cmp 指令，这样指令更短，并且执行速度更快。并且优化后，编译会合理排列 switch 后各 case 节点，以最优化方式找到所需要的节点。

如果各 case 取值表示一个算术级数，那么编译器会利用一个跳转表（Jump Table）来实现。例如：

```
switch(a)
{
   case 1   :printf("a=1");break;
   case 2   :printf("a=2");break;
   case 3   :printf("a=3");break;
   case 4   :printf("a=4");break;
   case 5   :printf("a=5");break;
   case 6   :printf("a=6");break;
   case 7   :printf("a=7");break;
   default  :printf("a=default");break;
}
```

编译器编译后，“jmp dword ptr [4*eax+004010B0]”指令就相当于 switch(a)，其根据 eax 的值进行索引，计算出指向相应 case 处理代码的指针。汇编代码如下：

```
:00401017   lea eax, dword ptr [ecx-01]
:0040101A   cmp eax, 00000006               ; 判断是否为 default 节点
:0040101D   ja  0040109D
:0040101F   jmp dword ptr [4*eax+004010B0]  ; 跳转表
:00401026   push 00408054                   ; case 1 ;printf("a=1");
:0040102B   call 004010D0
```

```
:00401030   add esp, 00000004
:00401033   xor eax, eax
:00401035   pop ecx
:00401036   ret
更多case代码略……
```

在实际程序中，case路径中还可能包含其他跳转分支语句、循环语句等，这样会使问题复杂化。

4.5.3 转移指令机器码的计算

在软件分析过程中，经常需要计算转移指令机器码或修改指定的代码。虽然有许多工具可以做这些事，但掌握其原理和技巧是很有必要的。

根据转移的距离，转移指令有以下类型。

- 短转移（Short Jump）：无条件转移和条件转移的机器码都是两个字节。转移范围是-128～+127字节。
- 长转移（Long Jump）：无条件转移的机器码是5个字节，条件转移的机器码是6个字节。这是因为条件转移要用2个字节表示其转移类型（如je、jg和jns），其他4个字节表示转移偏移量。无条件转移仅用一个字节就可表示其转移类型（jmp），其他4个字节表示转移偏移量。
- 子程序调用指令（call）：call指令调用有两类。一类是平常经常接触到的，类似于长转移（Long Jump）；另一类其调用的参数涉及到寄存器、堆栈等值，比较复杂，如“call dword ptr [eax + 2]”。

条件转移指令的转移范围是16位模式遗留下的，当时为了使代码紧凑些，CPU开发人员只给目的地址分配了一个字节，这样限制了跳转的长度只能在255个字节的范围里。

表4-4列出了常用的转移指令机器码，通过该表就可根据转移偏移量计算出转移指令的机器码。

表4-4 转移指令的条件与机器码

转移类别	标 志 位	含 义	短转移机器码	长转移机器码
CALL	—	call调用指令	E8xxxxxxxx	E8xxxxxxxx
JMP	—	无条件转移	EBxx	E9xxxxxxxx
JO	OF=1	溢出	70xx	0F80xxxxxxxx
JNO	OF=0	无溢出	71xx	0F81xxxxxxxx
JB/JC/JNAE	CF=1	低于/进位/不高于等于	72xx	0F82xxxxxxxx
JAE/JNB/JNC	CF=0	高于等于/不低于/无进位	73xx	0F83xxxxxxxx
JE/JZ	ZF=1	相等/等于零	74xx	0F84xxxxxxxx
JNE/JNZ	ZF=0	不相等/不等于零	75xx	0F85xxxxxxxx
JBE/JNA	CF=1或ZF=1	低于等于/不高于	76xx	0F86xxxxxxxx
JA/JNBE	CF=0且ZF=0	高于/不低于等于	77xx	0F87xxxxxxxx
JS	SF=1	符号为负	78xx	0F88xxxxxxxx
JNS	SF=0	符号为正	79xx	0F89xxxxxxxx
JP/JPE	PF=1	“1”的个数为偶	7Axx	0F8Axxxxxxxx
JNP/JPO	PF=0	“1”的个数为奇	7Bxx	0F8Bxxxxxxxx
JL/JNGE	SF≠OF	小于/不大于等于	7Cxx	0F8Cxxxxxxxx
JGE/JNL	SF=OF	大于等于/不小于	7Dxx	0F8Dxxxxxxxx
JLE/JNG	SF≠OF或ZF=1	小于等于/不大于	7Exx	0F8Exxxxxxxx
JG/JNLE	SF=OF且ZF=0	大于/不小于等于	7Fxx	0F8Fxxxxxxxx

有两个因素制约转移指令机器码：一个是表4-4中列出的转移类型；另一个是转移的位移量。

（1）短转移指令机器码计算实例

例如，代码段中有一条如下所示的无条件转移指令：

```
...
:401000 jmp 401005
...
:401005 xor eax,eax
...
```

无条件短转移的机器码形式为 EBxx，其中 EB00～EB7F 是向后转移，EB80～EBFF 是向前转移。图 4.6 表示了该转移指令的机器语言，以及用位移量来表示转向地址的方法。由图 4.6 可见，位移量为 3h，CPU 执行完“jmp 401005”指令后的 EIP 值为 401002h，然后执行：“(EIP)←(EIP)+位移量”，执行后就跳转到 401005 地址处。即“jmp 401005”指令机器码形式是“EB 03”（见图 4.7）。

也就是说，转移指令的机器码形式是：

位移量 = 目的地址 - 起始地址 - 跳转指令本身的长度

转移指令机器码 =“转移类别机器码”+“位移量”

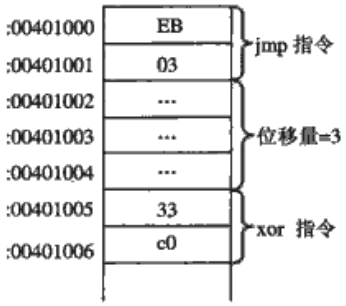


图 4.6　短转移举例

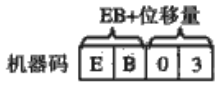


图 4.7　机器码组合形式

（2）长转移指令机器码计算实例

例如，代码段中有一条如下所示的无条件转移指令：

```
...
:401000 jmp 402398
...
:402398 xor eax,eax
...
```

无条件长转移指令的长度是 5 个字节，机器码是 E9。根据上面公式，此例转移的位移量为：

00402398h-00401000h-5h=00001393h

由图 4.8 可见，00001393h 在内存中以双字存储（32 位）。存放时，低位字节存入低地址，高位字节存入高地址，也就是说，00 00 13 93 以相反的次序存入，形成了 93 13 00 00 存储形式。

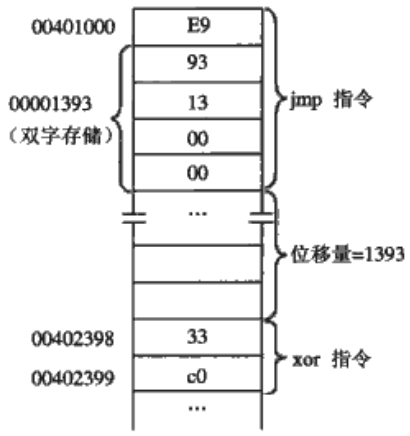


图 4.8　长转移举例

转移指令机器码=“转移类别机器码”+“位移量”

=“E9”+“93 13 00 00”

= E9 93 13 00 00

上面两个实例演示转移指令向后转移（由低地址到高地址）。如果是向前转移（由高地址到低地址），计算方法一样。

例如，代码段中有一条如下所示的无条件转移指令向前转移：

```
:401000 xor eax,eax
…
:402398 jmp 401000
…
```

位移量 = 401000h-402398h-5h =FFFFEC63h（取后 32 位）

转移机器码=“E9”+“63 EC FF FF”= E9 63 EC FF FF

4.5.4 条件设置指令（SETcc）

条件设置指令的形式是：SETcc r/m8，其中 r/m8 表示 8 位寄存器或单字节内存单元。

条件设置指令根据处理器定义的 16 种条件 cc，测试一些标志位，然后把结果记录到目标操作数中。当条件满足时，目标操作数会被置 1，否则置 0。这 16 种条件与条件转移指令 Jcc 中的条件是一样的，如表 4-5 所示。

表 4-5 条件设置指令

机 器 码	伪码指令	目标置 1 时的意义	标 志 位
0F 90	SETO r/m8	溢出	OF=1
0F 91	SETNO r/m8	未溢出	OF=0
0F 92	SETC/SETB/SETNAE r/m8	进位/低于/不高于等于	CF=1
0F 93	SETNC/SETAE/SETNB r/m8	无进位/高于等于/不低于	CF=0
0F 94	SETE/SETZ r/m8	相等/等于零	ZF=1
0F 95	SETNE/SETNZ r/m8	不相等/不等于零	ZF=0
0F 96	SETBE/SETNA r/m8	低于等于/不高于	CF=1 或 ZF=1
0F 97	SETNBE/SETA r/m8	不低于等于/高于	CF=0 且 ZF=0
0F 98	SETS r/m8	符号为负	SF=1
0F 99	SETNS r/m8	符号为正	SF=0
0F 9A	SETP/SETPE r/m8	“1”的个数为偶	PF=1
0F 9B	SETNP/SETPO r/m8	“1”的个数为奇	PF=0
0F 9C	SETL/SETNGE r/m8	小于/不大于等于	SF≠OF
0F 9D	SETGE/SETNL r/m8	大于等于/不小于	SF=OF
0F 9E	SETLE/SETNG r/m8	小于等于/不大于	ZF=1 或 ZF≠OF
0F 9F	SETG/SETNLE r/m8	大于/不小于等于	SF=OF 且 ZF=0

条件设置指令可以用来消除程序中的转移指令。在 C 语言里，常会见到执行以下功能的语句：

```
c = (a < b)? c1:c2;
```

如果允许出现条件分支，编译器会产生如下的代码或者是非常类似的代码：

```
    cmp     a, b
    mov     eax,c1
    jl      L1
    mov     eax,c2
L1:
```

如果使用条件设置指令，编译器将会产生不包含条件分支的逻辑判断代码：

```
xor     eax,eax
cmp     a, b
setge   al          ; a≥b，则al置1；否则置0
dec     eax
and     eax,(c1-c2)
add     eax,c2
```

也可用条件传输指令 cmov 或 fcmov 去除程序中的转移指令，但是它们仅被 Pentium Pro 以后的处理器支持。实现同样功能的代码如下：

```
mov     eax, c2
cmp     a, b
cmovl   eax, c1
```

4.5.5 纯算法实现逻辑判断

一些编译器优化时，在不改变原逻辑的情况下，使用数学技巧把源代码中一些逻辑分支语句转换成算术操作，消除或减少程序中出现的条件转移指令，提高 CPU 的流水线的性能。

来看这段 C 程序：

```
int main(void)
{
if(FindWindow(NULL,"计算器"))
    return 1;
else
    return 5;
}
```

用 Microsoft Visual C++ 6.0 编译，设置优化选项为 Maximize Speed。其反汇编代码如下：

```
00401000  push   406030                    ;/Title = "计算器"
00401005  push   0                         ;|Class = 0
00401007  call   dword ptr [40509C]        ;\FindWindowA
0040100D  neg    eax
0040100F  sbb    eax, eax
00401011  and    al, 0FC
00401013  add    eax, 5
00401016  retn
```

编译生成的代码没有一句条件转移指令，却实现原程序的逻辑。代码首先用 neg 指令检验 eax 是否为 0，结果存放在 CF 标志位中。sbb 指令将目的操作数减去源操作数，再减去借位 CF（进位），结果送到目的操作数。“sbb eax,eax”这句的结果由 CF 决定，当 CF 为 1 时，eax 为-1，否则为 0。用伪码来表示：

```
if (eax)
  CF = 1;
Else
  CF = 0;
eax = -CF;
```

接下来两句指令根据 eax 的值：FFFFFFFFh 和 0 来决定最终结果。当 eax 是 FFFFFFFFh 时，计算结果是 1；当 eax 是 0 时，计算结果为 5。

```
00401011  and    al, 0FC
00401013  add    eax, 5
```

这类代码比较常见，当知道是条件转移指令优化生成的，还原就比较容易了。

4.6 循环语句

循环是高级语言中可以进行反向引用的一种语言形式，其他类型的分支语句（如 IF-THEN-ELSE 等）都是由低向高端地址区域走的。因此，通过这点可以较方便地将循环语句识别出来。

如果确定某段代码是循环，就可分析其计数器，一般是用 ecx 寄存器做计数器，也有用其他方法来控制循环的，如“test eax, eax”等。下面是一段最简单的循环代码：

```
        xor ecx, ecx    ; ecx 清零
:00440000
        inc ecx         ; 计数
...
        cmp ecx, 05     ; 循环 4 次
        jbe 00440000    ; 重复
```

上面的汇编代码用高级语言 C 来描述有以下 2 种形式：

方案一	方案二

while (i < 5)
{
...
}
	for (i=0; i<5; i++)
	{
	...
	}

再来看一段较复杂的循环，比如这段 C 程序：

```
int main(void)
{
 int i,sum=0;
 for(i=0;i<=100;i++)
    sum =sum +i;
 return 0;
}
```

用 Microsoft Visual C++ 6.0 编译，没有优化，其反汇编代码如下：

```
:00401000   push    ebp
:00401001   mov     ebp, esp                    ; 建立堆栈页面
:00401003   sub     esp, 00000008               ; 为局部变量分配内存
:00401006   mov     [ebp-04], 00000000          ; 初始化局部变量[ebp-04],即 sum=0
:0040100D   mov     [ebp-08], 00000000          ; 初始化局部变量[ebp-08],即 i=0
:00401014   jmp     0040101F                    ; 循环从地址 40101F 处开始
:00401016   mov     eax, dword ptr [ebp-08]     ; 变量[ebp-08]值传给 eax
:00401019   add     eax, 00000001               ; eax 的值加 1
:0040101C   mov     dword ptr [ebp-08], eax     ; 更新[ebp-08]变量的值，即 i++
:0040101F   cmp     dword ptr [ebp-08], 64      ; 将变量 ebp-08 与 64h 比较
                                                ; 64h 十进制是 100，即 i<=100
:00401023   jg      00401030                    ; 如果 i>100 退出循环
:00401025   mov     ecx, dword ptr [ebp-04]     ; 将变量[ebp-04]放到 ecx 中
:00401028   add     ecx, dword ptr [ebp-08]     ; 相当于 sum +i
:0040102B   mov     dword ptr [ebp-04], ecx     ; 即 sum =sum +
:0040102E   jmp     00401016                    ; 这个跳转由低地址向高地址区域走的
                                                ; 识别循环的标志
:00401030   xor     eax, eax
:00401032   mov     esp, ebp
```

```
:00401034   pop     ebp                          ; 关闭堆栈页面
:00401035   ret
```

如果编译时设置优化选项为 Maximize Speed，然后看看汇编代码是如何变化的。其汇编代码如下：

```
:00401000   xor     ecx, ecx                     ; 变量初始化，即 sum=0
:00401002   xor     eax, eax                     ; 变量初始化，即 i=0
:00401004   add     ecx, eax                     ; 相当于 sum = sum + i
:00401006   inc     eax                          ; eax 加 1，即 i++
:00401007   cmp     eax, 00000064                ; 将变量 i 与 64h 比较
:0040100A   jle     00401004                     ; 如果 eax >100 退出循环
:0040100C   xor     eax, eax
:0040100E   ret
```

4.7 数学运算符

高级语言中的运算符范围很广，这里只介绍整数的加、减、乘、除运算。编译器如果没优化，这些运算符很好理解，可以参考相关的汇编书籍。本节主要认识一下经编译器优化过的运算符。

4.7.1 整数的加法和减法

一般情况下，整数的加法和减法编译成 add 和 sub 指令。编译优化时，比较喜欢用 lea 指令来代替 add 和 sub 指令。lea 指令允许用户在一个时钟内完成 c=a+b+78h 计算，其中 a、b 与 c 都是在寄存器的情况下才有效，会编译成“lea c,[a+b+78]”指令。

加法实例：

```
int main(void)
{
    int a,b;
    printf("%d",a+b+0x78);
    return 0;
}
```

用 Microsoft Visual C++ 6.0 编译，设置优化选项为 Maximize Speed。其反汇编代码如下：

```
:00401000   push ecx                             ;为局部变量分配内存
:00401001   mov  eax, dword ptr[esp]
:00401005   mov  ecx, dword ptr[esp]
:00401009   lea  edx, dword ptr[ecx+eax+78]      ;快速计算 ecx+eax+78 之和
:0040100D   push edx
:0040100E   push 00407030
:00401013   call 00401020                        ;printf 函数
:00401018   xor  eax, eax
:0040101A   add  esp, 0000000C
:0040101D   ret
```

在这句代码中，lea 指令只是一条纯算术指令，它的实际意义等价于“edx= ecx+eax+78”。

4.7.2 整数的乘法

乘法运算符一般编译成 mul、imul 指令。这些指令运行的速度比较慢，编译器会尽可能地提高代码的效率，从而倾向于使用其他指令来完成同样的计算。如果一个数是 2 的幂，那么会用左移指令 shl 来实现乘法。另外，加法对于提高 3,5,6,7,9 等数的乘法运算很有用，如：eax*5 可以写成“lea eax, [eax+4*eax]”。（lea 指令可以实现寄存器乘以 2、4 或 8 的运算。）

```
int main(void)
{
    int a;
    printf("%d %d %d", a*11+4,a*9,a*2);
    return 0;
}
```

用 Microsoft Visual C++ 6.0 编译，设置优化选项为 Maximize Speed。其反汇编代码如下：

```
:00401000   push ecx                                ;为局部变量 a 分配内存
:00401001   mov  eax, dword ptr [esp]
:00401005   lea  ecx, dword ptr [eax+eax]           ;即 a*2
:00401008   lea  edx, dword ptr [eax+8*eax]         ;edx=a+8*a=9*a
:0040100B   push ecx
:0040100C   lea  ecx, dword ptr [eax+4*eax]         ;ecx=a+4*a=5*a
:0040100F   push edx
:00401010   lea  edx, dword ptr [eax+2*ecx+4]       ;edx=a+2*ecx+4=11*a+4
:00401014   push edx
:00401015   push 00407030
:0040101A   call 00401030                           ;printf 函数
:0040101F   xor  eax, eax
:00401021   add  esp, 00000014
:00401024   ret
```

4.7.3 整数的除法

除法运算符一般编译成 div、idiv 指令。除法运算的代价是相当高的，大概比乘法多消耗 10 倍的 CPU 时钟。

如果被除数是一个未知数，编译器会使用 div 指令，程序执行效率会有所下降。

对于除数/被除数有一个是常量的情况，就会复杂很多。编译器将会使用一些技巧更有效地实现除法。如果除数是 2 的幂，那么可以用较快的移位指令“shr a,n”来替换，移位指令只需花费一个时钟，其中 a 是被除数，n 是基数 2 的指数。shr 适合无符号数计算，若是符号数则用 sar 指令。也会根据一定算法用乘法运算来取代除法运算。

具体来看除法的实例：

```
int main(void)
{
    int a;
    scanf("%d",&a);
    printf("%d ", a/11);
    return 0;
}
```

用 Microsoft Visual C++ 6.0 编译，不优化。其反汇编代码如下：

```
00401000  push   ebp                                ;建立堆栈页面
00401001  mov    ebp, esp
00401003  push   ecx                                ;为局部变量分配空间
00401004  lea    eax, dword ptr [ebp-4]
00401007  push   eax
00401008  push   408030                             ;ASCII "%d"
0040100D  call   00401065                           ;scanf("%d",&a)
00401012  add    esp, 8
00401015  mov    eax, dword ptr [ebp-4]             ;eax 中为输入的 a 值
00401018  cdq                                       ;将 eax 的值扩展为 4 字类型的值
```

```
00401019  mov    ecx, 0B                    ;除数 11（十六进制是 0Bh）放进 ecx 中
0040101E  idiv   ecx                        ;除法运算，商放到 eax 中，余数放 edx 中
00401020  push   eax
00401021  push   408034
00401026  call   00401034                   ;printf 函数
0040102B  add    esp, 8
0040102E  xor    eax, eax
00401030  mov    esp, ebp
00401032  pop    ebp
00401033  retn
```

除法指令需要用到符号扩展指令 cdq，其作用是把 eax 寄存器中的数视为有符号的数，将其符号位（即 eax 的最高位）扩展到 edx 寄存器，即若 eax 的最高位是 1，则执行后 edx 的每个位都是 1，结果 edx = FFFFFFFF；若 eax 的最高位是 0，则执行后 edx 的每个位都是 0，结果 edx = 00000000。这样就把 eax 中的 32 位带符号的数变成了 edx:eax 中的 64 位带符号的数，以满足 64 位运算指令的需要，但转换后的值没变。

编译器优化时，会采用乘法运算代替除法运算，这样能提高数倍的效率。不过，对于逆向分析来说，这样的代码比较难理解。

用于优化的公式比较多，最常用的就是倒数相乘。相关公式如下：

$$\frac{a}{b}=a\times\frac{1}{b}$$

用 Microsoft Visual C++ 6.0 编译，设置优化选项为 Maximize Speed。其反汇编代码如下：

```
00401000  push   ecx                        ;为局部变量 a 分配内存
00401001  lea    eax, dword ptr [esp]       ;变量 a 的值赋给 ecx
00401005  push   eax
00401006  push   408034
0040100B  call   00401071                   ;scanf("%d",&a)
00401010  mov    ecx, dword ptr [esp+8]
00401014  mov    eax, 2E8BA2E9              ;编译器生成的数，用于将除法转换乘法
00401019  imul   ecx                        ;进行乘法 a×2E8BA2E9h
0040101B  sar    edx, 1                     ;edx 中放的是乘法运算的高位双字节，即相当
                                            ;于 a×2E8BA2E9h 右移了 32 位，sar edx,1
                                            ;相当于右移了(32+1)，这三句指令功能是
                                            ;edx=(a×2E8BA2E9h)>>(32+1)
                                            ;=(a×2E8BA2E9h)/2^(32+1)
                                            ;=a×0.09090909094084054231643676 7
                                            ;≈a/11
0040101D  mov    ecx, edx                   ;将商复制到 ecx 中
0040101F  shr    ecx, 1F                    ;将 ecx 的值右移 31 位，即 ecx=ecx>>31
00401022  add    edx, ecx                   ;上句右移 31 位后只剩符号位了，如果是负数
                                            ;将对结果加 1
00401024  push   edx
00401025  push   408030
0040102A  call   00401040                   ;printf 函数
0040102F  xor    eax, eax
00401031  add    esp, 14
00401034  retn
```

这段代码就是一个简单的除法运算，编译器优化后的代码比一个 idiv 指令长，但其运行速度提高了 3 倍。还有更多的除法优化算法，不同编译器实行的方法也有所不同。

4.8 文本字符串

字符的识别和分析是软件逆向的一个重要步骤，特别是在一些序列号分析过程中，经常遇到各类字符操作。

4.8.1 字符串存储格式

程序中，一般将字符串作为字符数组来处理，但不同的编程语言，其字符存储格式是不同的。常见的字符串类型有 C 字符串、Pascal 字符串等。

1. C 字符串

C 字符串也称为 ASCIIZ 字符串，广泛应用于 Windows 与 UNIX 操作系统中，Z 表示其以“\0”为结束标志。“\0”代表 ASCII 码为 0 的字符，如图 4.9 所示。ASCII 码为 0 的字符不是一个可以显示的字符，而是一个“空操作符”。

P	E	D	I	Y	\0

图 4.9　C 字符串

2. DOS 字符串

在 DOS 下，输出行的函数以“$”字符作为终止字符，如图 4.10 所示。由于 DOS 的淘汰，目前已很少见这类字符格式了。

P	E	D	I	Y	$

图 4.10　DOS 字符串

3. Pascal 字符串

Pascal 字符串没有终止符，但在字符串的头部定义了一个字节，指示当前字符串的长度。由于只用一个字节来表示字符串的长度，所以字符串不能超过 255 个字符，如图 4.11 所示。字符串中的每个字符都属于 ANSIChar 类型（标准字符类型）。这种类型字符只存在于 Borland 公司的 Turbo Pascal 和 16 位 Delphi 中。

5	P	E	D	I	Y

图 4.11　Pascal 字符串

4. Delphi 字符串

为克服传统 Pascal 字符串的局限性，32 位 Delphi 增加了对长字符串的支持。

- 双字节 Delphi 字符串：表示长度的字段扩展为 2 个字节，可使字符串的最大长度达到 65535，如图 4.12 所示。

5	0	P	E	D	I	Y

图 4.12　Delphi 字符串

- 四字节 Delphi 字符串：表示长度的字段扩展为 4 个字节，使得字符串长度可达 4GB。目前这种字符类型很少使用。

4.8.2 字符寻址指令

80x86 支持寄存器直接寻址与寄存器间接寻址等模式。与字符指针处理相关的有 mov、lea 等指令。mov 指令将当前指令所在内存复制一份到目的寄存器中，其操作数可以是常量，也可以是指针。例如：

```
mov eax, [401000h]          ; 直接寻址，即把地址为 401000 的双字数据放入 eax 中
mov eax, [ecx]              ; 寄存器间接寻址，即把 ecx 中的地址所指的内容放入 eax 中
```

lea 意思是“装入有效地址（Load Effective Address）”，它的操作数就是地址，所以“lea eax,[addr]”就是将表达式 addr 的值放入 eax 寄存器中。

```
lea eax, [401000h]          ; 将值 401000h 写入 eax 寄存器中
```

lea 指令右边的操作数表示一个近指针，指令“lea eax, [401000h]”与“mov eax, 401000h”是等价的。

在计算索引与常量的和时，编译器一般将指针放在第一个位置，而不管它们在程序中的顺序。例如这段初始化代码：

```
mov dword ptr [eax+8],67452301
mov dword ptr [eax+c],EFCDAB89
```

编译器不仅广泛地用 lea 指令来传递指针，而且常用来计算常量的和，其等价于 add 指令。也就是说，“lea eax,[eax+8]”等价于“add eax,8”，不过 lea 指令的效率将远远高过 add，这种技巧可以使多个变量的求和在一个指令周期内完成，同时可以通过任何寄存器返回结果。

4.8.3 字母大小写转换

大写字母的 ASCII 码范围是 41h～5Ah，小写字母的 ASCII 码范围是 61h～7Ah，它们之间的转换就是 ASCII 码的值加减 20h。

下面这段汇编代码的功能是将小写字母转换成大写字母：

```
Label01:mov al, byte ptr [edx]   ; edx 是指向字符的指针，取出一个字符放到 al 中
        cmp al, 61               ; 如 al<61，则不是小写字母，可能是大写字母或数字
        jb  Label02              ; 跳到 Label02 不处理
        cmp al, 7A               ; al>z ?
        ja  Label02              ; 如大于 z，则不处理
        sub al, 20               ; 如是小写字母，则减 20h，转换成大写字母
Label02:mov byte ptr [esi], al   ; 将转换的大写字母放到 esi 指向的内存中
        inc edx                  ; edx 在这为计数器，加 1
        inc esi                  ; 同时 esi 也加 1
        dec ebx                  ; ebx 开始存放的字符串长度，这里长度减 1
        test ebx, ebx            ; 如 ebx=0，则认为字符串处理完毕
        jnz Label01              ; 如没处理完，循环继续处理
```

这段代码先用“a”来比较，如小于“a”，可能是大写字母或其他字符；然后再与“z”比较，如果大于“z”，则不是小写字母，并且不处理。如果确定是小写字母，将该字符的 ASCII 码减 20h，即可转换成大写字母。

另外，还有一种大、小写字母转换的方法。图 4.13 显示的是大写字母“A”与小写字母“a”的二进制形式，如果第 5 位是 0，则是大写字母；如果是 1，则是小写字母。

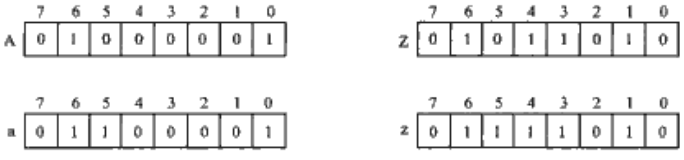


图 4.13 字母的二进制形式

因此，下面代码也能实现大、小写字母的转换：

```
MAIN    proc near
        lea bx, title+1
        mov cx,31
B20:
        mov ah,[bx]
        cmp ah,61h
        jb B30
        cmp ah,7Ah
        ja B30
        and ah,1101 1111b      ;and指令将 ah 第 5 位指令置 0 (11011111b=DFh)
        mov [bx],ah
B30:
        inc cx
        loop B20
        ret
MAIN    endp
```

4.8.4　计算字符串的长度

高级语言里会有特定函数来计算字符串的长度，如C语言中常用strlen()函数计算字符串的长度。strlen()在优化编译模式下的汇编代码如下：

```
    mov ecx, FFFFFFFF    ; 如果看到这句，程序很可能是要获得字符串的长度了
    sub eax, eax         ; eax 清零
    repnz                ; 重复串操作直到 ecx = 0 为止
    scasb                ; 把 AL 内容与 EDI 指定的在附加段中的数据逐个比较
    not ecx              ; ecx = 字符长度 + 1
    dec ecx              ; ecx 是真实的长度
    je xxxxxx            ; 如果 ecx=0，意味字符串的长度为 0
```

这段代码使用串扫描指令 scasb 操作，把 AL 的内容与 EDI 指向的附加段中的字节逐个比较，最后把 EDI 指向的字符串长度保存在 ecx 中。

4.9　指令修改技巧

在软件分析过程中，为了优化原程序或在一定空间里增添代码，需要一定的指令修改技巧。表 4-6 列出了常用指令修改技巧，在以后的实践操作中经常用到它们。

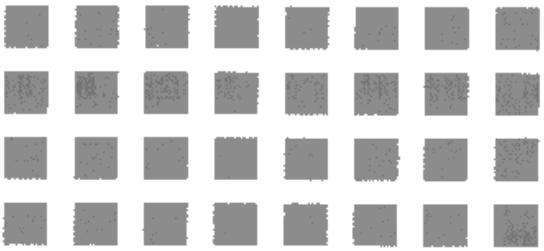
表 4-6　常用指令修改技巧

功　能	指　令	机 器 码	指令长度（bytes）
替换 1 个字节	nop	90	1
替换 2 个字节	nop	90	1
	nop	90	1
	mov edi, edi	8B FF	2
	push eax	50	1
	pop eax	58	1
	inc eax	40	1
	dec eax	48	1
	jmp xx	eb00	2

续表

功　能	指　令	机 器 码	指令长度（bytes）
寄存器清零	mov eax, 00000000h	B8 00 00 00 00	5
	push 0	6A 00	2
	pop eax	58	1
	sub eax, eax/ xor eax, eax	2B C0 /33 C0	2
测试寄存器是否为零	cmp eax, 00000000h	83 F8 00	3
	je _label_	74xx/0F84xxxxxxxx	2/6
	or eax, eax/ test eax, eax	0B C0 /85 C0	2
	je _label_	74xx/0F84xxxxxxxx	2/6
置寄存器为 0FFFFFFFFh	mov eax, 0ffffffffh	B8 FF FF FF FF	5
	xor eax, eax/ sub eax, eax	33 C0 /2B C0	2
	dec eax	48	1
	Stc	F9	1
	sbb eax, eax	2B C0	2
转移指令	jmp _label_	EBxx/E9xxxxxxxx	2/5
	push _label_	68 xx xx xx xx	5
	ret	C3	1

很多指令针对 eax 被做了优化，要尽可能多地使用 eax。例如，“xchg eax, ecx”只需要 1 个字节，而用其他寄存器则需要 2 个字节。



第 3 篇　解密篇

一些软件开发人员对软件保护方案的策划与实施很不以为然，他们往往自以为是的保护在解密者眼中不堪一击。希望本部分能让这些开发人员了解一些软件攻击的方法，以便更好地保护自己的作品。

第5章 常见的演示版保护技术

本章讲述一些常用的软件保护技术，对其优缺点进行分析，并给出软件保护的一般性建议。软件开发者将会从中获得一些有益的启发，以便更好地保护自己的智力成果。

5.1 序列号保护方式

先来看一看在网络上大行其道的序列号（又称为注册码，本书不再对此进行区分）保护的工作原理。当用户从网络上下载某个共享软件（Shareware）后，一般都有使用时间或功能上的限制。当过了共享软件的试用期后，必须到这个软件的公司去注册后方能继续使用。注册过程一般是用户把自己的私人信息（如用户名、电子邮件地址、机器特征码等）连同信用卡号码告诉软件公司，软件公司根据用户的信息利用预先写好的一个计算注册码程序（称为注册机，KeyGen）算出一个序列号，以电子邮件或传真等形式发给用户。用户在得到这个序列号后，按照注册需要的步骤在软件中输入注册信息和注册码，其注册信息的合法性由软件验证通过后，软件就会取消各种限制，如时间限制、功能限制等，而成为完全正式版本。软件在每次启动的时候，从磁盘文件或系统注册表中读出注册信息并对其进行检查。如果注册信息正确，则以完全正式版的模式运行，否则作为有功能限制或时间限制的版本来运行。注册的用户可以根据自己拥有的注册信息得到售后服务。当软件推出新版本后，注册的用户还可以向软件作者提供自己的注册信息来得到版本升级服务。这种保护实现起来比较简单，不需要额外的成本；用户购买也非常方便，因特网上80%的软件都是以这种方式保护的。

5.1.1 序列号保护机制

软件验证序列号的过程，其实就是验证用户名和序列号之间的数学映射关系。这个映射关系是由软件的设计者制定的，所以各个软件生成序列号的算法是不同的。显然，这个映射关系越复杂，注册码就越不容易被破解。根据映射关系的不同，程序检查注册码通常有如下4种基本的方法。

（1）以用户名等信息作为自变量，通过函数 F 变换之后得到注册码

将这个注册码和用户输入的注册码进行字符串比较或者数值比较，以确定用户是否为合法用户。其公式表示如下：

$$\text{序列号} = F(\text{用户名}) \tag{5-1}$$

由于负责验证注册码合法性的代码是在用户的机器上运行的，因此用户可以利用调试器等工具来分析程序验证注册码的过程。上述方法中计算出来的序列号是以明文形式在内存中出现的，很容易在内存中找到它，从而获得注册码。这种方法在检查注册码合法性的同时，也在用户机器上再现生成注册码的过程（即

在用户机器上执行了 F 函数)。实际上，这是非常不安全的。不论所采用的函数 F 有多么复杂，解密者只需把 F 函数的实现代码从软件中提取出来，就可编制一个通用的计算注册码程序。由此可见，这种检查注册码的方法是极其脆弱的。解密者也可通过修改比较指令的办法，通过注册码检查。

(2) 通过注册码验证用户名的正确性

软件作者在给注册用户生成注册码的时候，使用的仍然是下面的这种变换：

$$\text{序列号} = F(\text{用户名})$$

注意，这里要求 F 是个可逆变换。软件在检查注册码的时候则是利用 F 的逆变换 F^{-1} 对用户输入的注册码进行变换的。如果变换的结果和用户名相同，则说明是正确的注册码。即

$$\text{用户名} = F^{-1}(\text{序列号}) \tag{5-2}$$

可以看到，用来生成注册码的 F 函数未直接出现在软件代码中，而且正确注册码的明文也未出现在内存中，所以这种检查注册码的方法比第 1 种要安全一些。

破解这种注册码检查方法除了可以采用修改比较指令的办法之外，还有如下几种考虑：

- 由于 F^{-1} 的实现代码是包含在软件中的，所以可以通过 F^{-1} 来找出其逆变换即 F 函数，从而可以得到正确的注册码或者写出注册机；
- 给定一个用户名，利用穷举法找到一个满足式 (5-2) 的序列号，这只适用于穷举难度不大的函数；
- 给定一个序列号，利用式 (5-2) 变换得出一个用户名 (当然这个用户名中一般包含不可显示字符)，从而得到一个正确的用户名/序列号对。

(3) 通过对等函数检查注册码

如果输入的用户名和序列号满足式 (5-3)，则认为是正确的注册码。采用这种方法，同样可以做到在内存中不出现正确注册码的明文。如果 F_2 是一个可逆函数，则本方法实际上是第 2 种方法的一个推广，解密方法也类似。

$$F_1(\text{用户名}) = F_2(\text{序列号}) \tag{5-3}$$

上面所说的 3 种检查序列号的方法中所采用的自变量都只有一个，自变量是用户名或序列号。

(4) 同时采用用户名和序列号作为自变量，即采用二元函数

这种检查注册码的方法将采用如下的判断规则：当对用户名和序列号进行变换时，如果得出的结果和某个特定的值相等，则认为是合法的用户名/序列号对。

$$\text{特定值} = F_3(\text{用户名，序列号}) \tag{5-4}$$

这个算法看上去相当不错，用户名与序列号之间的关系不再那么清晰了，但同时可能也失去了用户名与序列号的一一对应关系，软件开发者自己都很有可能无法写出注册机，必须维护用户名称与序列号之间的唯一性，但这似乎不难办到，建个数据库就可以了。当然，也可根据这一思路把用户名称和序列号分为几个部分来构造多元的算法。

$$\text{特定值} = F_n(\text{用户名 1，用户名 2……序列号 1，序列号 2……})$$

以上所说的都是序列号与用户名相关的情况，实际上序列号也可以与用户名根本不存在任何关系，这完全取决于软件作者的考虑。

由上可见，注册码的复杂性问题归根到底是一个数学问题。要想设计难以求逆的算法，要求软件作者有一定的数学基础。当然，即使检查注册码的算法再复杂，如果可执行程序可以被任意修改，解密者还是可以通过修改比较跳转指令来使程序成为注册版。所以，光有好的算法是不够的，还得结合软件完整性检查等其他方法。

5.1.2 如何攻击序列号保护

若要找到序列号，或者修改判断序列号之后的跳转指令，最重要的是要利用各种工具来定位判断序列号的代码段。

一种办法是通过跟踪输入注册码之后的判断，从而找到注册码。一般都是在一个编辑框中输入注册码，软件需要调用一些标准的 API 将编辑框中输入的注册码字符串拷贝到自己的缓冲区中。利用调试器提供的针对 API 设断点的功能，就有可能找到判断注册码的地方。这些常用的 API 包括 GetWindowTextA(W)、GetDlgItemTextA(W)、GetDlgItemInt、Hmemcpy（仅 Windows 9x/Me）等。程序判断完注册码之后，一般显示一个对话框，告诉用户注册码是否正确，这也是一个切入点。显示对话框的常用 API 函数包括 MessageBoxA(W)、MessageBoxExA(W)、DialogBoxParamA(W)、CreateDialogIndirectParamA(W)、DialogBoxIndirectParamA(W)、CreateDialogParamA(W)、MessageBoxIndirectA(W)、ShowWindow 等。

另外一种办法就是跟踪程序启动时对注册码的判断，因为程序每次启动时都需要将注册码读出来加以判断，从而决定是否以注册版的模式工作。根据序列号的存放位置的不同，可以使用不同的 API 断点。如果序列号存放在注册表中，可以用 RegQueryValueExA(W)；如果序列号存放在 INI 文件中，可以用 GetPrivateProfileStringA(W)、GetProfileStringA(W)、GetPrivateProfileIntA(W)、GetProfileIntA(W)等函数；如果序列号存放在一般的文件中，可以用 CreateFileA(W)、_lopen()等函数。

1. 数据约束性的秘诀

这个概念是+ORC 提出的，只限于用明文比较注册码的保护方式。在大多数序列号保护的程序中，那个真正的、正确的注册码会于某个时刻出现在内存中，当然它出现的位置是不定的，但多数情况下它会在一个范围之内，即存放用户输入序列号的内存地址±90h 字节的地方。

数据约束性（Data constraint）或者“密码相邻性（Password proximity）”的依据就是加密者在编程的时候需要留意保护功能是否“工作”，必须“看到”用户输入的数字、用户输入转换结果和真正密码之间的关系，这种联系必须经常地检查以调用这些代码。通常，它们会共同位于一个小的堆栈区域（注意：参数或局部变量通常都是存储在堆栈中的，而软件作者一般都使用局部变量存放临时计算出来的注册码），使得它们可以在同一个监视（Watch）窗口中看到，所以在大多数情况下，真正的密码会在离保存用户输入密码不远的地方露出马脚来。

例：运行第 2 章的 TraceMe 程序，输入用户名：pediy；序列号：12121212。单击“Check”按钮，TraceMe 提示序列号错误，不要关闭此提示窗口。运行 WinHex，单击菜单“Tools/RAM Editor”或按“Alt+F9”键打开内存编辑工具，单击“TraceMe”选项打开“Primary Memory”内存查看。按“Ctrl+F”键打开查找对话框，输入假的序列号“12121212”，在这附近会发现另一个字符串“2470”，这个就是真序列号，结果如图 5.1 所示。

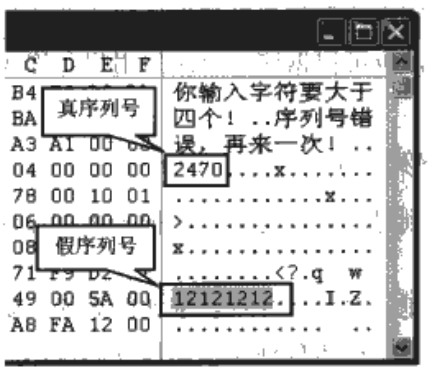


图 5.1 数据的约束性图

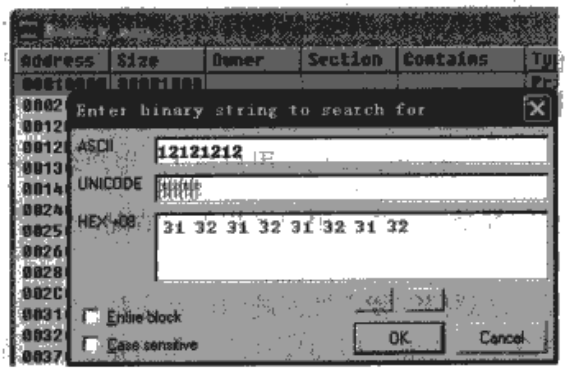


图 5.2 在 OllyDbg 内存窗口中查找字符串

用 OllyDbg 也可实现这种查找功能。用 OllyDbg 加载 TraceMe 输入假序列号，单击“Check”按钮直

到出现错误提示框。按“Alt+M”键打开内存窗口，在最上一行，按“Ctrl+B”键打开搜索框，搜索刚输入的序列号“12121212”，如图 5.2 所示。OllyDbg 这个数据查找功能非常有用，可以在当前进程的整个内存映像里查找数据。

由于不少软件作者不了解这些基本的解密知识，算法上虽然下了很大工夫，但最后还是采用明码比较，这样导致一个会使用 WinHex 的普通用户都能找到其序列号。

2．Hmemcpy 函数（俗称万能断点）

函数 Hmemcpy 是 Windows 9x 系统的内部函数，它的作用是将内存中的一块数据拷贝到另一个地方。由于 Windows 9x 系统频繁使用该函数处理各种字符串，因此用它作为断点很实用，它是 Windows 9x/Me 平台最常用的断点。在 Windows NT/2000 中没有这个断点，因为其内核和 Windows 9x 完全不同。

3．利用消息断点

许多序列号保护的软件都有一个按钮，当鼠标左键按下和释放时，将发送消息 WM_LBUTTONDOWN（0201h）和 WM_LBUTTONUP（0202h），因此用这个消息下断很容易找到按钮的事件代码。

4．利用提示信息

大多数软件在设计时，都采用了人机对话方式。所谓人机对话，即软件在执行完某一段程序之后，便显示一串提示信息，以反映该段程序运行后的状态。例如，TraceMe 实例输入假的序列号，如“序列号错误，再来一次!”。可以用 OllyDbg、IDA、W32Dasm 等反汇编工具查找相应字符串，定位到相关代码处。

用 OllyDbg 打开实例 TraceMe，单击鼠标右键，执行“Search for/All referenced text strings（查找/所有参考文本字串）”命令，OllyDbg 将列出程序中出现的字符串。但 OllyDbg 自带的这个功能对中文支持得不好，因此建议使用 Ultra String Reference 插件。安装好插件后，右键菜单中执行“Ultra String Reference/Find ASCII”，即列出中文字符串，双击相关字符串即可定位到所需代码上。

5.1.3　字符串比较形式

在序列号分析过程中，字符串处理是一个重点，必须掌握一定的分析技能。加密者为了有效防止解密者修改跳转指令，往往采取一些技巧，迂回比较字符串。

（1）寄存器直接比较

```
mov  eax [  ]     ;eax 或 ebx 放的是直接比较的两个数，一般是十六进制形式
mov  ebx [  ]     ;同上
cmp  eax,ebx      ;直接比较两个寄存器
jz(jnz)  xxxx
```

（2）函数比较 1

```
mov  eax [  ]     ;比较数字直接放在 eax 中，一般为十六进制形式，也可能是地址
mov  ebx [  ]     ;同上
call xxxxxxxx     ;比较功能的函数，可以是 API 函数，也可以是作者自己的比较函数
test eax eax
jz(jnz)
```

在这种情况下，call 一般是一个 BOOL 函数，其结果通过 eax 返回。分析时，关注该 call 里面返回时对 eax 处理的代码。例如 call 里面的代码：

```
cmp xxx,xxx
jz Lable
xor eax, eax      ;对 eax 清 0
Lable: pop edi
pop esi
```

```
pop ebp
ret              ;函数返回
```

（3）函数比较 2

```
push xxxx        ;参数 1，可以是地址、寄存器
push xxxx        ;参数 2
call xxxxxxxx    ;比较功能的函数，可以是 API 函数，也可以是作者自己的比较函数
test eax eax
jz(jnz)
```

（4）串比较

```
lea edi [ ]      ;edi 指向字符串 a
lea esi [ ]      ;esi 指向字符串 b
repz cmpsd       ;比较字符串 a、b
jz(jnz)
```

5.1.4 注册机制作

软件开发结束后，作者本人很有必要先做攻击测试，找出弱点，避免犯一些低级错误。一般注册算法其实是一些极为简单的算法，基本都是明码的，或者明码相近的，如查表、异或、换位、移位、累加和等，算法实现都比较容易。

1. 明码比较软件的攻击

只要正确的序列号在内存中曾以明码形式出现（不管比较时用不用明码）都属于这一类。有些软件采取了一机一号的保护方式，即软件根据用户硬件等产生一个唯一的机器号，注册码与机器号对应有效地防止了序列号散发。如果是明码比较，攻击还是很容易的。能轻易实现这一目的，就是利用 keymake 软件，它拦截程序指令并将出现的明码以某种方式直接显示出来。

实例 TraceMe 的序列号是明码比较的，相关代码如下：

```
004011E3   52           push   edx
004011E4   50           push   eax
004011E5   E8 56010000  call   00401340    ;进入子程序(keymakep 设置第一次中断地址)
{
    ...
    0040138D   55            push   ebp   ;第二次中断地址（D ebp 可查看到真序列号）
    0040138E   50            push   eax
    0040138F   FF15 04404000 call   [<&KERNEL32.lstrcmpA>]
}
```

运行 keymake 后，单击菜单“其他/内存注册机”，打开如图 5.3 所示的界面。

具体操作步骤如下：

① 单击“浏览”按钮，打开目标程序 TraceMe.exe。

②“内存方式”选择寄存器，本例是 EBP，即序列号保存在 EBP 所指向的内存地址中。

③ 中断地址列表：

- 中断地址，目标程序明码比较指令地址；
- 次数，在此地址中断的次数；
- 指令，此处指令指机器码第一个字节；

本例，keymake 需要拦截 TraceMe 两次，第一次是 4011E5h 的 call；进去后，再次拦截 40138Dh 地址，以便查看 EBP 指向的字符串。设置信息见图 5.3。

按上述设置好后，单击“生成”按钮就可生成一个注册机，使用时该注册机和目标程序放到同一目录

里。运行时，注册机装载目标程序，在指定地址处插入一个 INT 3 指令，目标程序会在此中断，然后注册机将内存或寄存器值读出，再恢复原程序指令。TraceMe 被装载后，输入用户名，单击“Check”按钮，注册机将跳出一个窗口告知正确的序列号（见图 5.4）。

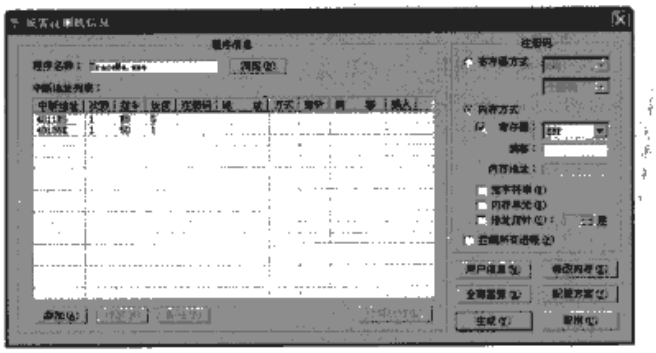


图 5.3　keymake 的内存注册机

图 5.4　生成的序列号

注意：杀毒软件会将 keymake 生成的文件报告为木马或病毒，读者可以参考“补丁技术”一章，自己编程实现内存注册机。

2. 非明码比较

实例 Serial.exe 是通过对等函数检查序列号。如果输入的用户名和序列号满足计算式：F_1（用户名）= F_2（序列号），则认为是正确的序列号。采用这种方法，可以做到在内存中不出现明码。

单击实例 Serial.exe 菜单“Help/Register”打开注册窗口，这个窗口是用 DialogBoxParamA 函数建立，EndDialog 函数关闭的。可以用函数 GetDlgItemTextA、EndDialog 等设断拦截。由于程序是先关闭对话框才开始比较序列号，因此要在系统里走一段才能回到 Serial.exe 程序领空。也可以直接从提示信息切入找到关键点。用 OllyDbg 装载 Serial.exe 后，输入姓名 pediy，序列号 1212，单击“OK”按钮后，将跳出“Incorrect!,Try Again”提示窗口，记下这串字符。用鼠标右键打开“Search for/All referenced text strings”交叉参考字符串窗口，找到“Incorrect!,Try Again”，双击就可来到关键代码处。很明显 401228h 这段代码处理输入的字符串“pediy”，在此按 F2 键设断，重新来一次，运行程序单步分析如下。

```
00401228   push   0040218E                ;字符串“pediy”入栈，设其为 Name
0040122D   call   0040137E                ;计算 k1，k1=F1(用户名)
00401232   push   eax                     ;将 k1 入栈（通过堆栈传递变量）
00401233   push   0040217E                ;字串“1234”入栈，设其为 Code
00401238   call   004013D8                ;计算 k2，k2=F2(序列号)
0040123D   add    esp, 4                 ;平衡堆栈，上面 CALL 的 k2 是通过 EBX 传出的
00401240   pop    eax                    ;将 k1 出栈
00401241   cmp    eax, ebx               ;比较 k1 和 k2，即:F1(用户名)= F2(序列号)
00401243   je     short 0040124C         ;如相等，注册成功
00401245   call   00401362               ;不相等，注册失败
```

call 0040137E 函数内部代码：

```
0040137E   mov    esi, dword ptr [esp+4] ;将 Name 地址放入 ESI 中
00401382   push   esi                    ;ESI 寄存器的值入栈
00401383   /mov   al, byte ptr [esi]     ;取出 Name 中的一个字节
00401385   |test  al, al                 ;检查 Name 中是否还有字符
00401387   |je    short 0040139C
00401389   |cmp   al, 41                 ;字符大于 A? 判断是否是字母(A~Z, a~z)
0040138B   |jb    short 004013AC         ;如不是字母，出错
0040138D   |cmp   al, 5A                 ;字符大于 Z 吗
```

```
0040138F   |jnb  short 00401394          ;如是小写字母，跳到CALL 004013D2
00401391   |inc  esi                     ;将指针移向下一个字符
00401392   |jmp  short 00401383
00401394   |call 004013D2                ;将小写字母转成大写字母
00401399   |inc  esi                     ;将指针移向下一个字符
0040139A   \jmp  short 00401383
0040139C   pop  esi                     ;ESI出栈，即ESI重新指向Name首位
0040139D   call 004013C2                ;对Name变形处理
004013A2   xor  edi, 5678               ;再将刚得到的结果与5678h异或运算
004013A8   mov  eax, edi                ;将结果放入EAX中准备结束此函数
004013AA   jmp  short 004013C1
004013AC   pop  esi                     ;如输入的不是字母，出错
004013AD   push 30                      ;/Style
004013AF   push 00402160                ;|Title = "Error!  "
004013B4   push 00402169                ;|Text = "Incorrect!,Try Again"
004013B9   push dword ptr [ebp+8]       ;|hOwner
004013BC   call <USER32.MessageBoxA>    ;\MessageBoxA
004013C1   retn

004013C2   xor  edi, edi                ;EDI清零
004013C4   xor  ebx, ebx                ;EBX清零
004013C6   /mov bl, byte ptr [esi]      ;将ESI所指的Name取出一位到bl
004013C8   |test bl, bl                 ;Name中还有字符吗
004013CA   |je   short 004013D1         ;没有计算结束
004013CC   |add  edi, ebx               ;EDI=EDI+EBX(依次将Name字符相加)
004013CE   |inc  esi                    ;将指针移向下一个字符
004013CF   \jmp  short 004013C6
004013D1   retn

004013D2   sub  al, 20                  ;ASCII值减去20h，将小写字母转成大写
004013D4   mov  byte ptr [esi], al      ;将转换后的字符放回[ESI]
004013D6   retn
```

上面的代码是计算 k1=F_1（用户名），用 C 语言来描述，代码如下：

```
int F1 (char *name)
{
    int i,k1=0;
    char ch;
    for(i=0;name[i]!=0;i++)
    {
      ch=name[i];
      if(ch<'A') break;
        k1+=(ch>'Z')?(ch-32):ch;
    }
  k1=k1^0x5678;
  return k1;
}
```

call 004013D8 函数内部代码：

```
004013D8   xor  eax, eax                ;清零
004013DA   xor  edi, edi                ;清零
004013DC   xor  ebx, ebx                ;清零
004013DE   mov  esi, dword ptr [esp+4]  ;ESI指向输入的序列号code[i]
004013E2   /mov al, 0A                  ;将10放到AL寄存器
004013E4   |mov bl, byte ptr [esi]      ;取出code[i]中的数字放到BL寄存器
```

```
004013E6   |test bl, bl                    ;检查code[i]中是否还有数字
004013E8   |je   short 004013F5            ;计算结束跳到4013F5
004013EA   |sub  bl, 30                    ;EBX=BL-30h=code[i]-30h
004013ED   |imul edi, eax                  ;EDI=EDI×10，实际上是左移一位
004013F0   |add  edi, ebx                  ;EDI= EDI+EBX=EDI+(code[i]-30h)
004013F2   |inc  esi                       ;指向code下一个字符，ESI相当于i
004013F3   \jmp  short 004013E2            ;循环继续计算
004013F5   xor  edi, 1234                  ;将EDI与1234h异或运算
004013FB   mov  ebx, edi                   ;将结果放到EBX中准备退出此子程序
004013FD   retn
```

上面的代码是计算 k2=F_2（序列号），用 C 语言来描述，代码如下：

```
int F2 (char *code)
{
    int i,k2=0;
    for(i=0;Code[i]!=0;i++)
    {
      k2= k2*10+Code[i]-48;
    }
    k2=k2^0x1234;
    return k2
}
```

只要满足关系式：k1=k2，注册就成功。写注册机就要对 F_1 或 F_2 函数逆变换，若 F_1 和 F_2 都不可逆，只能用穷举法。如要从用户名算出正确的序列号，只要写出 F_2 逆函数即可：k1= F_2^{-1}（序列号）。但求逆 F_2 函数有多个解，比较复杂，幸运的是，k1 的结果是个十六进制数，这样问题就简单了，F_2 函数功能可看作是将输入的十进制数转换成十六进制数。

注册机如下：

```
//注意：name[i]的位数是10位，原因见其调用的GetDlgItemTextA函数
int keygen(char *name)
{
    int i,k1=0,k2=0;
    char ch;
    for(i=0; name[i]!=0&&i<=9;i++)
    {
      ch=name[i];
      if(ch<'A')    break;
      k1+=(ch>'Z')?(ch-32):ch;
    }                          // 异或运算是可逆的
    k2=k1^0x5678^0x1234;       // 这里k2是以十进制表示的，直接输出就是序列号
    return k2;
}
```

算法求逆是有难度的，这需要一定的编程基本功。常见的加密配对指令有 xor/xor、add/sub、inc/dec、rol/ror 等，这些指令就是一条可以加密，另一条指令可以解密。

还有一种写注册机方法是不分析其运算过程，用 OllyDbg 的 Asm2Clipboard 插件、IDA 等工具可直接将序列号算法的汇编代码提取出来，再嵌入到高级语言中。这个方法的优点是不用理解算法实现的细节，只需要将汇编代码嵌入到注册机里即可。比如，F_1 函数就是这种情况，将 40137Eh 到 4013D6h 之间的汇编代码转换成 asm 文件格式，然后嵌入到高级语言中调用。代码转换中要注意堆栈平衡、数据进制、汇编语法格式、字符串引用等。下面就是直接提取汇编代码内嵌到 VC 里的代码：

```
int keygen(char *name)
{
```

```
    int k1=0,k2=0;
    BOOL bIsnum=FALSE;
    __asm
    {
        mov    esi,OFFSET  cName
        push   esi
    L002:
        mov    al, byte ptr [esi]
        test   al, al
        je L014
        cmp    al, 0x41
        jb L019
        cmp    al, 0x5A
        jnb L011
        inc    esi
        jmp L002
    L011:
        call L035
        inc    esi
        jmp L002
    L014:
        pop    esi
        call L026
        xor    edi, 0x5678
        mov    eax, edi
        jmp L025
    L019:
        pop    esi
        mov  bIsnum,1
    L025:
        jmp LEND
    L026:
        xor    edi, edi
        xor    ebx, ebx
    L028:
        mov    bl, byte ptr [esi]
        test   bl, bl
        je L034
        add    edi, ebx
        inc    esi
        jmp L028
    L034:
        retn
    L035:
        sub    al, 0x20
        mov    byte ptr [esi], al
        retn
    LEND:
        mov k1,eax
    }
    if(bIsnum)
        return 0;
    k2=k1^0x1234;
    return k2;
}
```

5.2 警告（Nag）窗口

Nag 的本义是烦人的意思。Nag 窗口是软件设计者用来不时提醒用户购买正式版本的窗口。软件设计者可能认为，当用户忍受不了试用版中的这些烦人的窗口时，就会考虑购买正式版本。它可能会在程序启动或退出时弹出来，或者在软件运行的某个时刻随机或定时地弹出来，确实比较烦人。

去除警告窗口常用的 3 种方法是：修改程序的资源、静态分析及动态分析。

使用资源修改工具去除警告窗口是个不错的方法，可以将可执行文件中的警告窗口的属性改成透明、不可见，这样就变相地去除了警告窗口。

若要完全去除警告窗口，只需找到创建此窗口的代码，跳过即可。显示窗口的常用函数有 MessageBoxA(W)、MessageBoxExA(W)、DialogBoxParamA(W)、ShowWindow、CreateWindowExA(W)等。然而，某些警告窗口用这些断点不管用，可试一试利用消息设置断点，一般都能拦截下来。

实例 Nag.exe 是一个显示警告窗口的程序，调用 DialogBoxParamA 函数来显示资源中的对话框。由于 Nag.exe 是调用资源来显示对话框的，因此用 eXeScope 或 Resource Hacker 打开它，显示警告窗口的资源如图 5.5 所示。

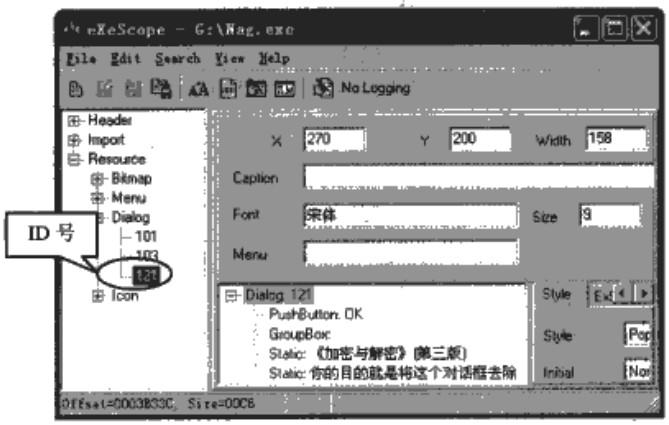


图 5.5　查看对话框资源

启动画面窗口的 ID 号是 121，换成十六进制就是 79h。用 W32Dasm 打开 Nag.exe，打开对话框参考，里面的“Dialog: DialogID_0079”项就是 Nag.exe 刚运行时跳出的对话框，双击此项可以来到相关代码处。W32Dasm 里的具体代码如下：

```
:0040104D   mov eax, dword ptr [esp+04]
:00401051   push 00000000                    ;初始化值
:00401053   push 004010C4                    ;对话框处理函数指针，指向一段子程序
:00401058   push 00000000                    ;父窗口句柄
:0040105A   push 00000079                    ;对话框 ID 号：DialogID_0079
:0040105C   push eax                         ;应用程序实例句柄，即 Nag.exe 的基地址
:0040105D   mov dword ptr [0040119C], eax
* Reference To: USER32.DialogBoxParamA, Ord:0093h
:00401062   Call dword ptr [00401010]        ;显示 Nag 对话框
:00401068   xor eax, eax
:0040106A   ret 0010
```

DialogBoxParam 一般和 EndDialog 配对使用，前者是打开对话框，后者是关闭对话框。因此，不能简单地将 DialogBoxParam 屏蔽掉。DialogBoxParam 原型如下：

```
int DialogBoxParam(
    HINSTANCE hInstance,                    // 应用程序实例句柄
    LPCTSTR lpTemplateName,                 // 对话框 ID 号
    HWND hWndParent,                        // 父窗口句柄
    DLGPROC lpDialogFunc,                   // 对话框处理函数指针
    LPARAM dwInitParam                      // 初始化值
);
```

从上面函数看出，lpDialogFunc 参数很重要，DialogBoxParam 函数将跳到其指向的地址执行，对 lpDialogFunc 参数（此处为 4010C4h）设断。中断后代码如下：

```
004010C4  mov    eax, dword ptr [esp+8]
004010C8  sub    eax, 110                ;Switch (cases 110..111)
004010CD  je     short 00401103
004010CF  dec    eax
004010D0  jnz    short 004010FF
004010D2  mov    eax, dword ptr [esp+C]  ;Case 111 of switch 004010C8
004010D6  dec    eax
004010D7  jnz    short 004010FF
004010D9  push   0
004010DB  push   dword ptr [esp+8]
004010DF  call   [<&USER32.EndDialog>]   ;关闭对话框
004010E5  push   0                       ;初始化值
004010E7  push   00401109                ;主对话框处理函数指针
004010EC  push   0                       ;父窗口句柄
004010EE  push   65                      ;主对话框 ID 号：DialogID_0065
004010F0  push   0
004010F2  call   [<&KERNEL32.GetModuleHandleA>]
004010F8  push   eax
004010F9  call   [<&USER32.DialogBoxParamA>]
```

原来主程序也是用 DialogBoxParam 函数显示的，因此有两种改法：

（1）跳过警告窗口那段代码

将“00401051 push 00000000”一句改成：“00401051 jmp 4010E5”。

修改时在 OllyDbg 里键入正确的代码，选择修改后的代码，执行右键菜单中的“复制到可执行文件”功能，即可将修改保存到磁盘文件中。

（2）将两个 DialogBoxParam 函数参数对换

DialogBoxParam 函数有两个参数很重要，一个是主对话框处理函数指针，另一个是对话框的 ID 号。这种方法的思路是将主窗口的这两个参数放到警告窗口的 DialogBoxParam 函数上。修改如下：

```
:00401051    push 00000000
:00401053    push 00401109              ;将此处指向主窗口的子处理程序
:00401058    push 00000000
:0040105A    push 00000065              ;指向主对话框的 ID 号：DialogID_0065
:0040105C    push eax
:0040105D    mov dword ptr [0040119C], eax
* Reference To: USER32.DialogBoxParamA, Ord:0093h
:00401062    Call dword ptr [00401010]  ;该函数就会调用主对话框窗口
:00401068    xor eax, eax
:0040106A    ret 0010                   ;主对话框关闭后将从这退出
```

在另外一些情况下，对话框不是以资源形式存在的，而常用断点又拦不下来，这时可试试消息断点，如 WM_DESTROY 等。

5.3 时间限制

时间限制程序有两类：一类是每次运行多少时间；另一类是每次运行时间不限，但是有个时间段限制，如用 30 天等。

5.3.1 计时器

这类程序每次运行时都有时间限制，例如运行 10 分钟或 20 分钟就停止，必须重新运行该程序才能正常工作。这些程序里面自然有个计时器统计程序运行的时间。那么如何实现计时器呢？在 DOS 系统下，应用程序可以通过接管系统的计时器中断（一般为 int 8h 或 int 1Ch）维护一个计时器，它能每 55 毫秒发生一次（18.2 次每秒）。在 Windows 下，使用计时器有如下几种不同的选择。

1. 使用 SetTime()函数

应用程序可在初始化时调用这个 API 函数来向系统申请一个计时器，并且指定计时器的时间间隔；还可提供一个处理计时器超时的回调函数。当计时器超时时，系统将会向申请该计时器的窗口过程发送消息 WM_TIMER，或者调用程序所提供的那个回调函数。

该函数的原型如下：

```
UINT SetTimer (HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc );
```

各参数的含义如下。

- hWnd：窗口句柄，当计时器时间到时时，系统将向这个窗口发送 WM_TIMER 消息。
- nIDEvent：计时器标识。
- uElapse：指定计时器时间间隔，以毫秒为单位。
- TIMERPROC：回调函数。当计时器超时时，系统将调用这个函数。如果本参数为 NULL，当计时器超时时将向相应的窗口发送 WM_TIMER 消息。这个回调函数的原型为：

```
void CALLBACK TimerProc(HWND hwnd, UINT uMsg, UINT idEvent, DWORD dwTime );
```

由于 SetTimer()是以 Windows 消息的方式工作的，所以其精度有一定的限制，但对于做软件保护来说已经够用。当程序不再需要计时器时，可以调用 KillTimer()来销毁计时器。

2. 使用高精度的多媒体计时器

多媒体计时器的精度最高可以达到 1 毫秒。应用程序可以通过调用 timeSetEvent()启动一个多媒体计时器。该函数的原型如下：

```
MMRESULT timeSetEvent ( UINT uDelay, UINT uResolution, LPTIMECALLBACK lpTimeProc,
                        DWORD_PTR dwUser, UINT fuEvent);
```

3. GetTickCount()

Windows 提供了 API 函数 GetTickCount()。该函数返回的是系统自成功启动以来所经过的毫秒数。将该函数的两次返回值相减，就可知道程序已经运行了多长时间。这个函数的精度取决于系统的设置。实际上，也可以在高级语言里面利用各自开发库中所提供的函数来实现计时，比如在 C 语言中就可使用 time()函数获得系统时间。

4. timeGetTime()

多媒体计时器函数 timeGetTime()也可以返回 Windows 自启动后所经过的时间，以毫秒为单位。一般情况下，不需要使用高精度的多媒体计时器，精度太高对系统性能也会有影响。

5.3.2 时间限制

这类保护的软件一般都有时间段的限制，例如试用 30 天等。当过了共享软件的试用期后，就不予运行。只有向软件作者付费注册之后才能得到一个无时间限制的注册版本。这种保护的实现方式大致如下。

首先在安装软件的时候由安装程序取得当前系统日期，或者主程序在第一次运行的时候获得系统日期，并且将其记录在系统中的某个地方：可能记录在注册表的某个不显眼的位置，也可能记录在某个文件或扇区中。这个时间统称为软件的安装日期。

程序在每次运行的时候都要取得当前系统日期，且将其与记录下来的那个安装日期进行比较，当其差值超出允许的天数（比如 30 天）时就停止运行。

可见，这种日期限制的机理很简单。但是在实现的时候，如果对各种情况处理得不够周全，就很容易被绕过，比如在过期之后简单地把机器时间调回去，软件又可以正常使用了。

如果考虑得比较周全，软件最少要保存两个时间值，一个就是上面所说的安装时间，这个时间可由安装程序在安装软件的时候记录，也可以在软件第一次运行的时候记录（即软件发现该值不存在时就将当前日期作为其值记录下来）。为了增加解密难度，最好把这个时间在不同的地方多存放几份，否则解密者可以通过 RegMon、FileMon 等监视工具轻易地找到存放该值的地方，然后删除该键值，这样又可以正常使用软件了。

另外一个时间值就是软件最近一次运行的日期，这是防止用户将机器日期改回去而设的。软件每次退出的时候都要将该日期取出来与当前日期相比较，如果当前日期大于该日期，则用当前日期替换掉该值，否则保持该值不变。同时，软件每次启动的时候要把该值读出来与当前日期进行比较，如果该值大于当前系统日期，则说明用户把机器时间改回去了，可以拒绝运行。

取得时间的 API 函数一般有 GetSystemTime、GetLocalTime 和 GetFileTime。软件作者可能不直接使用上面的函数来获得系统时间，比如采用高级语言中封装好的类来操作系统时间等。这些封装好的类实际上也是调用上面的函数。解密者在采用动态跟踪方法破解这种日期限制时，最常用的断点也是这几个。

还有一种比较方便地获得当前系统日期的办法，就是读取需要频繁修改的系统文件（比如 Windows 注册表文件 user.dat、system.dat 等）的最后修改日期，利用 FileTimeToSystemTime()将其转换为系统日期格式，从而得到当前系统日期。

需要指出的是，采用日期限制的软件必须能防 RegMon、FileMon 之类的监视软件，否则很容易被找到日期的存放位置。

5.3.3 拆解时间限制保护

实例 Timer.exe 程序用了 SetTimer 函数计时，每次运行 20 秒。其运行原理是，先用 SetTimer (hwnd,1,1000,NULL)设置一个计时器，时间间隔是 1000 毫秒，这个函数每秒发一次 WM_TIMER 消息。当应用程序收到消息时，将执行下面的语句：

```
case WM_TIMER :
    if(i<=19)
        i++;                                    //i 的初值是 0
    else
        SendMessage(hDlg, WM_CLOSE, 0, 0);    //关闭程序
return 0 ;
```

因此，可以用 SetTimer 函数设断拦截。

```
004010C2    mov     esi, dword ptr [esp+8]
004010C6    push    0                          ;/Timerproc = NULL
004010C8    push    3E8                        ;|Timeout = 1000. ms
```

```
004010CD   push   1                          ;|TimerID = 1
004010CF   push   esi                        ;|hWnd
004010D0   call    [<&USER32.SetTimer>]      ;\SetTimer
004010D6   mov    eax, dword ptr [403004]
```

去除时间有如下两种方法。

方法一：直接跳过 SetTimer 函数，不产生 WM_TIMER 消息。

来到地址 4010C6，输入修改指令“jmp 4010C6”。

方法二：利用 WM_TIMER 消息。

查 VC 的头文件 WINUSER.H，得知：#define WM_TIMER 0x0113。

在 W32Dasm 里查找 113 字串（当然，实际中有可能用其他形式检查是否为 113），如下所示：

```
00401175   cmp    eax, 113                  ;Case 113 (WM_TIMER)
0040117A   jnz    short 00401148
0040117C   mov    eax, dword ptr [403008]   ;[403008]存放的 i（定义了全局变量）
00401181   cmp    eax, 13                   ;超过 20 秒（13 是十六进制）
00401184   jg     short 00401137            ;超时就跳走退出，直接 NOP 掉
00401186   inc    eax                       ;i++
00401187   lea    ecx, dword ptr [esp+C]
0040118B   push   eax
0040118C   push   00403000
00401191   push   ecx
00401192   mov    dword ptr [403008], eax   ;将 i 放进[403008]
```

因此，只要修改 00401184 一行就能取消时间限制，用两个字节替换掉，如 9090 或 eb00。

另外，辅助工具变速齿轮可加快和缩短应用程序的时间，一般用来配合动态分析。例如，某软件运行一小时后才退出，此时可以用变速齿轮加速时间，几分钟后，软件就认为到了一小时而退出，从而更方便调试程序。

5.4 菜单功能限制

这类程序一般是 Demo 版，其菜单或窗口中的部分选项是灰色，无法使用。这种功能受限制的程序一般分成两种：第一种是试用版和正式版的软件完全分开的两个版本，被禁止的功能在试用版的程序中根本没有相应的程序代码，这些代码只有在正式版中才有，而正式版是无法免费下载的，只有向作者购买。对于这种程序，解密者要想在试用版中使用和正式版一样的功能几乎是不可能的，除非自己向可执行程序中添加相应的代码。第二种是试用版和注册版为同一个文件，没有注册的时候按照试用版运行，禁止用户使用某些功能。一旦用户注册之后就以正式版模式运行，用户可以使用全部功能。可见，被禁止的那些功能的程序代码其实是存在于程序之中的，解密者只要通过一定的方法恢复被限制的功能，就能使该 Demo 软件与正式版一样。对比一下就知道，前一种显然更好，因为它使得破解难度大大增加。如果采用功能限制的保护方式，强烈建议使用前一种方式。

5.4.1 相关函数

软件将菜单或窗口变灰或变为不可用，一般采用下面的几个函数。

1. EnableMenuItem

允许或禁止指定的菜单条目。原型如下：

```
BOOL EnableMenuItem(HMENU hMenu, UINT uIDEnableItem, UINT uEnable);
```

各参数的含义如下。

- hMenu：菜单句柄。
- uIDEnableItem：欲允许或禁止的一个菜单条目的标识符。
- uEnable：控制标志，有 MF_ENABLED（允许，0h）、MF_GRAYED （灰化，1h）、MF_DISABLED（禁止，2h）、MF_BYCOMMAND 和 MF_BYPOSITION。

返回值：返回菜单项以前的状态，如果菜单项不存在，就返回 FFFFFFFFh。

2. EnableWindow

允许或禁止指定窗口。原型如下：

```
BOOL EnableWindow(HWND hWnd, BOOL bEnable)
```

各参数的含义如下。

- hWnd：窗口句柄。
- bEnable：TRUE 允许，FALSE 禁止。

返回值：非 0 表示成功，0 表示失败。

5.4.2 拆解菜单限制保护

名称：EnableMenu，文件：光盘\ chap05\

这个程序是用 EnableMenuItem 函数禁止菜单，其关键代码如下：

```
:004011E3  6A01              push 00000001     ; 控制标志
:004011E5 68459C0000         push 00009C45     ; 标识符（Menu的ID=40005）
:004011EA 50                 push eax          ; 菜单句柄
:004011EB FF1524204000       Call USER32.EnableMenuItem
```

uEnable 控制标志为 0 时，恢复菜单的功能，具体操作如下：

将“004011E3 push 00000001”改成“push 0”。

5.5 KeyFile 保护

KeyFile 是一种利用文件来注册软件的保护方式。KeyFile 一般是一个小文件，可以是纯文本文件，也可以是包含不可显示字符的二进制文件。其内容是一些加密过或未加密的数据，其中可能有用户名、注册码等信息，文件格式则由软件作者自己定义。试用版软件没有注册文件。当用户向作者付费注册之后，会收到作者寄来的注册文件，其中可能包含用户的个人信息。用户只要将该文件放入指定的目录，就可以让软件成为正式版。该文件一般放在软件的安装目录中或系统目录下。软件每次启动时，从该文件中读取数据，然后利用某种算法进行处理，根据处理的结果判断是否为正确的注册文件。如果正确，则以注册版模式运行。

在实现这种保护的时候，建议软件作者采用稍大一些的文件作为 KeyFile，一般在几 KB 左右。其中可以加入一些垃圾信息以干扰解密者的企图；对于注册文件的合法性检查也可以分成几部分，分散在软件的不同模块中进行判断；对注册文件内的数据处理也尽可能地采用复杂的运算，而不要使用简单的异或运算。这些措施都可增大解密难度。和注册码一样，也可以让注册文件中的部分数据和软件中的关键代码或数据发生关系，使得软件无法被暴力破解。

5.5.1 相关 API 函数

由于 Key File 是一个文件，因此所有有关 Windows 文件操作的 API 函数都可作为动态跟踪破解的断点。这类常用的文件函数如表 5-1 所示。

表 5-1　与 Key File 相关的函数

API 函数	用于注册文件时的主要作用
FindFirstFileA	确定注册文件是否存在
CreateFileA、_lopen	确定文件是否存在；打开文件以获得其句柄
GetFileSize、GetFileSizeEx	获得注册文件的大小
GetFileAttributesA、GetFileAttributesExA	获得注册文件的属性
SetFilePointer、SetFilePointerEx	移动文件指针
ReadFile	读取文件内容

各 API 函数的具体含义参考 MSDN 或相关 API 文档。

5.5.2　拆解 KeyFile 保护

名称：PacMe，文件：光盘\chap05\

1．拆解 KeyFile 的一般思路

① 先用 FileMon 等工具监视软件对文件的操作，以找到 KeyFile 的文件名。

② 伪造一个 KeyFile 文件。用十六进制工具编辑和修改 KeyFile，普通的文本编辑工具不太适合。

③ 在调试器里用 CreateFileA 函数设断查看其打开文件名指针，并记下返回的句柄。

④ 用 ReadFile 设断，分析传递给 ReadFile 的文件句柄和缓冲区地址。文件句柄一般和第③步的相同（若不同，则说明不是读该 KeyFile，此外也可用条件断点）。缓冲区地址则是非常重要的，因为读进来的重要数据放在这里。对缓冲区中存放的字节设内存断点，监视读进来的 KeyFile 的内容。

⑤ 当然上述步骤只是大概轮廓，有的程序判断 KeyFile 时还会先判断文件大小和属性、移动文件指针等。

总之，分析 KeyFile 取决于对 Win32 File I/O API 的熟悉程度，也就是 API 编程的水平。

2．监视文件的操作

PacMe 的注册信息放在某一文件中，可以用文件监视工具得到答案。FileMon 是一个不错的选择，使用时，建议设置一下过滤器。所谓过滤器，其实就是一组条件，这组条件用来限制 FileMon 什么该显示，什么不该显示，如图 5.6 所示。

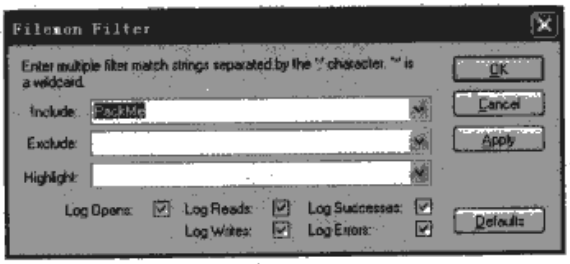


图 5.6　过滤器配置对话框

设置好后，按“Ctrl+E”键捕捉事件，按“Ctrl+X”键清除所有的记录。FileMon 按时间的顺序记录系统中发生的各种文件访问事件。

3．分析过程

除了用 FileMon 监视文件获得 KeyFile 文件名，也可以直接对文件相关函数设断获得 KeyFile 相关信息。用 OllyDbg 装载 PacMe 后，按 F9 键运行 PacMe。用 CreateFileA 设断，单击 PacMe 的“Check”按钮中断如下：

```
004016D8   push   edx                          ;|FileName => "KwazyWeb.bit"
004016D9   call   <jmp.&KERNEL32.CreateFileA>;\CreateFileA
```

```
004016DE    cmp     eax, -1
004016E1    je      short 00401747
```

OllyDbg 直接把 CreateFileA 读取的文件名显示出来。很明显，KeyFile 名为 KwazyWeb.bit。用十六进制工具伪造一个 KeyFile，建议内容为一些有规律的数字，如 12345…等，以便跟踪时有利于分析。重新运行程序，PacMe 将打开 KwazyWeb.bit 文件，读取数据进行计算比较。代码如下：

```
004016E8    push    0                       ;/pOverlapped = NULL
004016EA    push    00403448                ;|pBytesRead = PacMe.00403448
004016EF    push    1                       ;|BytesToRead = 1
004016F1    push    004034FA                ;|Buffer = PacMe.004034FA
004016F6    push    dword ptr [403444]      ;|hFile = NULL
004016FC    call    <&KERNEL32.ReadFile>    ;\ReadFile
00401701    movzx   eax, byte ptr [4034FA] ;读 KeyFile 第 1 个字节，保存在 4034FA
00401708    test    eax, eax                ;如是 0 则关闭 KeyFile
0040170A    je      short 00401747
0040170C    push    0                       ;/pOverlapped = NULL
0040170E    push    00403448                ;|pBytesRead = PacMe.00403448
00401713    push    eax                     ;|BytesToRead(字节大小)
00401714    push    00403288                ;|Buffer =00403288(缓存地址)
00401719    push    dword ptr [403444]      ;|hFile = NULL
0040171F    call    <&KERNEL32.ReadFile>    ;从 KeyFile 第二个字节开始读取数据
00401724    call    00401000                ;将刚读取的字节数求和（用户名求和）
{
    00401000    xor   eax, eax              ;清零
    00401002    xor   edx, edx              ;清零
    00401004    xor   ecx, ecx              ;清零
    00401006    mov   cl,byte ptr [4034FA] ;第一个字节，计数器（姓名的长度）
    0040100C    mov   esi, 00403288         ;ESI 指向第二个字节后的数据
    00401011    lods  byte ptr [esi]        ;将[ESI]指向的 1 字节数据放到 EAX 中
    00401012    add   edx, eax              ;将相加结果放到 EDX 中
    00401014    loopd short 00401011        ;循环
    00401016    mov   byte ptr [4034FB],dl ;将计算结果放到 4034FB 处
    0040101C    retn
}
00401729    push    0                       ;/pOverlapped = NULL
0040172B    push    00403448                ;|pBytesRead = PacMe.00403448
00401730    push    12                      ;|BytesToRead = 12 (18.)
00401732    push    004034E8                ;|Buffer = PacMe.004034E8
00401737    push    dword ptr [403444]      ;|hFile = NULL
0040173D    call    <&KERNEL32.ReadFile>    ;第三次读取数据，长度为 12h（18）
00401742    call    004010C9                ;验证的子程序，计算核心
00401747    push    dword ptr [403444]      ;/hObject = NULL
0040174D    call    <&KERNEL32.CloseHandle>;\CloseHandle，关闭文件
```

再来分析一下验证的核心代码：

```
004010C9    push    ebp
004010CA    mov     ebp, esp
004010CC    add     esp, -4
004010CF    push    00403365                ;/String2 ="****************C
==========把这个奇怪的字符串 String2 放大==============================
****************
C*......*...****
.*.****...*....*
.*..**********.*
```

```
..*....*...*...*
*.****.*.*...***
*.*....*.*******
..*.***..*.....*
.*..***.**.***.*
...****....*X..*
****************
这个就是经典的“吃豆子”游戏，“C”就是吃家，“*”是墙壁，“.”是通路，“X”是终点
=================================================================
004010D4    push    004031BC
004010D9    call    <&KERNEL32.lstrcpyA>        ;复制字串
004010DE    mov     [403184],004031CC           ;指向上面的图案中 C，设为 Current
004010E8    call    0040101D                    ;处理第三次读取的 12h(18)个数据
{
        0040101D    mov  dl, byte ptr [4034FB];第二次读取数据的和放到 DL 中
        00401023    mov  ecx, 12                ;ECX 是个计数器为 12H(18)
        00401028    mov  eax, 004034E8          ;第三次读取的数据指针放入到 EAX 中
        0040102D    xor  byte ptr [eax], dl     ;与 DL 异或
        0040102F    inc   eax                   ;指向下一个数据
        00401030    loopd short 0040102D        ;循环
        00401032    retn                        ;设 Strxor 指向异或后的 004034E8
}
004010ED    mov     byte ptr [ebp-2], 0         ;设[EBP-2] = x; x= 0
004010F1    xor     eax, eax                    ;清零
004010F3    xor     ecx, ecx                    ;清零
004010F5    /mov     byte ptr [ebp-1], 8        ;设[EBP-1] = len = 8
004010F9    |/sub    byte ptr [ebp-1], 2  ;     len =len-2=6
004010FD    ||movzx  ecx, byte ptr [ebp-2]      ;ECX = x
00401101    ||add    ecx, 004034E8              ;ECX+Strxor，加上异或后字节的地址
00401107    ||mov    al, byte ptr [ecx]         ;AL = StrToInt(string[ecx])
00401109    ;||mov   cl, byte ptr [ebp-1]       ;CL = len
0040110C    ||shr    al, cl                     ;右移 6 位
0040110E    ||and    al, 3                      ;使用低字节，代表 4 个方向
                                                ;将 Strxor 指向的十六进制转成四进制
00401110    ||call   00401033                   ;按照 AL 制定的方向移动“C”
                                                ;如果到达了 X 处，注册成功
{
        00401033    push   ebp
        00401034    mov    ebp, esp
        00401036    add    esp, -8
        00401039    mov    edx, dword ptr [403184]
        0040103F    mov    dword ptr [ebp-4], edx   ;将 Current 值保存在 [ebp-04]中
        00401042    or     al, al                   ;AL=0
        00401044    jnz    short 0040104F           ;AL=0，Current 减 10h（向上移）
        00401046    sub    dword ptr [403184], 10
        0040104D    jmp    short 0040106E
        0040104F    cmp    al, 1                    ;AL=1
        00401051    jnz    short 0040105B
        00401053    inc    dword ptr [403184]       ;Current 加 01h（向右移）
        00401059    jmp    short 0040106E
        0040105B    cmp    al, 2                    ;AL=2
        0040105D    jnz    short 00401068
        0040105F    add    dword ptr [403184], 10 ;Current 加 10h（向下移）
        00401066    jmp    short 0040106E
        00401068    dec    dword ptr [403184]       ;AL=3，Current 减 01h（向左移）
        0040106E    mov    edx, dword ptr [403184]
```

```
          00401074   mov    al, byte ptr [edx]      ;查看 Current 处的值
          00401076   cmp    al, 2A                  ;是字符 2Ah('*')吗
          00401078   jnz    short 00401080
          0040107A   xor    eax, eax                ;是'*', 则 EAX 返回 0
          0040107C   leave
          0040107D   retn
          0040107E   jmp    short 004010B3
          00401080   cmp    al, 58                  ;检查是否为字符'X'
          00401082   jnz    short 004010B3          ;如果不等于就跳，相等就注册成功
          00401084   push   0                       ;/Style
          00401086   lea    edx, dword ptr [403359];|
          0040108C   push   edx                     ;Title => "Success.."
          0040108D   lea    edx, dword ptr [4032EC];|
          00401093   push   edx                     ;|Text => "Congratulations!
          00401094   push   0                       ;|hOwner = NULL
          00401096   lea    edx, dword ptr [4017AC];|
          0040109C   call   edx                     ;\MessageBoxA
          0040109E   lea    edx, dword ptr [40327B]
          004010A4   push   edx                     ;/Text => "Cracked by :
          004010A5   push   dword ptr [403420]      ;|hWnd
          004010AB   lea    edx, dword ptr [4017DC];|
          004010B1   call   edx                     ;\SetWindowTextA
          004010B3   mov    edx, dword ptr [403184]
          004010B9   mov    byte ptr [edx], 43      ;将 Current 处的值改为 43h，即 C
          004010BC   mov    edx, dword ptr [ebp-4] ;将以前的 Current 值调出
          004010BF   mov    byte ptr [edx], 20      ;将其设为空格，表示已走过的路
          004010C2   mov    eax, 1                  ;EAX=1 后返回，继续下一步
          004010C7   leave
          004010C8   retn
}
00401115    ||test   eax, eax                      ;如果 EAX=0
00401117    ||je     short 0040112A                ;此处相当于高级语言的 Break
00401119    ||movzx  edx, byte ptr [ebp-1]         ;EDX = len
0040111D    ||test   edx, edx
0040111F    |\jnz    short 004010F9                ;没结束，继续小循环（共 4 次）
00401121    |inc    byte ptr [ebp-2]
00401124    |cmp    byte ptr [ebp-2], 12          ;把 12h 个字节遍历完
00401128    \jnz    short 004010F5                ;没结束，继续大循环
0040112A    leave
0040112B    retn
```

这是一个标准的迷宫，从“C”开始，一共走 18 次，每次可以走 4 步（18 次大循环和 4 次小循环）。碰到“*”就中断，直到遇见“X”时就注册成功。而且路线非常清楚，顺着“.”走。按照上面的程序分析，“0”代表↑，“1”代表→，“2”代表↓，“3”代表←，看着图一步步向前进，就可以得到一系列数据（见图 5.7）。

```
↓↓↓→  ↓↓↓←  ↓↓→→  ↑→↑↑  →→→↑  ↑←←←  ↑←↑↑  →→→→  →↓→→
2221  2223  2211  0100  1110  0333  0300  1111  1211
.
↑→→↓  →→→↓  ↓←←↓  ←←↑←  ←↓↓↓  ←↓↓→  →→↑↑  →→→→  ↓↓←←
0112  1112  2332  3303  3222  3221  1100  1111  2233
```

图 5.7　PacMe 的迷宫路线

上面是四进制数，转换成十六进制数为：

A9 AB A5 10 54 3F 30 55 65 16 56 BE F3 EA E9 50 55 AF

然后，程序通过用户名计算一数据，再与上面的十六进制数异或。

在此以用户名 pediy 推出 KeyFile，pediy 的十六进制代码是：7065646979。KeyFile 由三部分组成，如图 5.8 所示。

① 先计算 pediy 字符的和：70+65+64+69+79=21B，取低 8 位为：1B；

② 然后用 1B 依次与“A9 AB A5 10 54 3F 30 55 65 16 56 BE F3 EA E9 50 55 AF”异或，结果是“B2 B0 BE 0B 4F 24 2B 4E 7E 0D 4D A5 E8 F1 F2 4B 4E B4”。

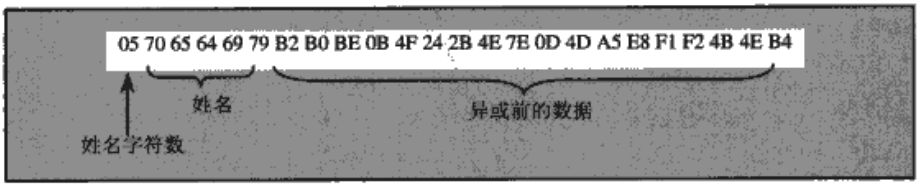


图 5.8　PacMe 的 KeyFile 内容

5.6 网络验证

网络验证是目前流行的一种保护技术，其优点是可以将一些关键数据放到服务器上，软件运行时，必须从服务器取得这些数据才能正确运行。拆解的思路是拦截服务器返回的数据包，分析程序是如何处理数据包的。

5.6.1 相关函数

当一个连接建立以后，就可以传输数据了，常用的传送数据的函数有 send 和 recv 两个 Socket 函数，另外还有微软的扩展函数 WSASend 和 WSARecv。

1. send 函数

客户程序一般用 send 函数向服务器发送请求，而服务器则通常用 send 函数来向客户程序发送应答。

```
int send(
  SOCKET s,                  //套接字描述符
  const char FAR *buf,       //缓冲区
  int len,                   //实际要发送的数据的字节数
  int flags                  //附加标志，一般为 0
);
```

2. recv 函数

不论是客户还是服务器应用程序都用 recv 函数从 TCP 连接的另一端接收数据。

```
int recv(
  SOCKET s,                  //套接字描述符
  char FAR *buf,             //缓冲区
  int len,                   //缓冲区 buf 的长度
  int flags                  //附加标志，一般为 0
);
```

5.6.2 网络验证破解一般思路

如果网络验证的数据包内容固定，可以将数据包抓取，写个本地服务端模拟服务器；如果验证的数据包内容不固定，则必须分析出其结构，找出相应的算法。

实例 CrackMeNet.exe 是一款网络验证实例，CrackMeNetS.exe 是服务端，并提供了一组正确的登录账

号。实际过程中，服务端是接触不到的，读者必须从客户端入手，并利用一组正确的账号，击破这个网络验证保护。

1. 分析发送的数据包

建议用 IDA 与 OllyDbg 一起来分析，IDA 能正确识别出 C 函数，方便分析。OllyDbg 加载客户端后，用 send 函数设断，输入正确的账号与口令，单击“Register”按钮，中断并回到当前领空。代码如下：

```
00401625   push  0                                ;/Flags = 0
00401627   mov   eax, dword ptr [ebp-23C]        ;|
0040162D   push  eax                             ;|DataSize
0040162E   lea   ecx, dword ptr [ebp-354]        ;|
00401634   push  ecx                             ;|Data
00401635   mov   edx, dword ptr [ebp-200]        ;|
0040163B   push  edx                             ;|Socket
0040163C   call  <jmp.&WS2_32.send>              ;\send
```

send 函数将把 Data 缓冲区中的数据发送到服务端，这里查看 Data 数据，发现是加密的。在 IDA 中向前查看代码，再结合 OllyDbg 分析，这段代码功能如下：

```
0040150E   push  ecx
0040150F   call  00401F40                  ;strlen(),取输入 Name 的长度
00401514   add   esp, 4
00401517   mov   dword ptr [ebp-240],eax ;nameLength = strlen( bufname)
0040151D   lea   edx, dword ptr [ebp-234]
00401523   push  edx
00401524   call  00401F40                  ;strlen(),取输入 Key 的长度
00401529   add   esp, 4
0040152C   mov   dword ptr [ebp-1F8],eax ;keyLength = strlen( bufkey);
00401532   push  0
00401534   call  00401DB0                  ;time(0)
00401539   add   esp, 4
0040153C   push  eax
0040153D   call  00401D70                  ;srand(time(0))
00401542   add   esp, 4
00401545   call  00401D80                  ;rand()
0040154A   and   eax, 800000FF             ;ran_K = rand() % 256
0040154F   jns   short 00401558
00401551   dec   eax
00401552   or    eax, FFFFFF00
00401557   inc   eax
00401558   mov   byte ptr [ebp-254], al  ;bufEncrypt[2] = ran_K;
0040155E   mov   al, byte ptr [ebp-240]
00401564   mov   byte ptr [ebp-354], al  ;bufEncrypt[0] = (BYTE)nameLength
0040156A   mov   cl, byte ptr [ebp-1F8]
00401570   mov   byte ptr [ebp-353], cl  ;bufEncrypt[1] = (BYTE)keyLength
00401576   mov   dl, byte ptr [ebp-254]
0040157C   mov   byte ptr [ebp-352], dl
00401582   mov   eax, dword ptr [ebp-240]
00401588   push  eax
00401589   lea   ecx, dword ptr [ebp-288]
0040158F   push  ecx
00401590   lea   edx, dword ptr [ebp-351]
00401596   push  edx
00401597   call  00401A30                  ;memcpy(bufEncrypt+3,bufname, Length)
0040159C   add   esp, 0C
```

```
0040159F   mov   eax, dword ptr [ebp-1F8]
004015A5   push  eax
004015A6   lea   ecx, dword ptr [ebp-234]
004015AC   push  ecx
004015AD   mov   edx, dword ptr [ebp-240]
004015B3   lea   eax, dword ptr [ebp+edx-351]
004015BA   push  eax
004015BB   call  00401A30    ;memcpy(bufEncrypt+3+nameLength,bufkey,Length)
004015C0   add   esp, 0C
004015C3   mov   ecx, dword ptr [ebp-1F8]
004015C9   mov   edx, dword ptr [ebp-240]
004015CF   lea   eax, dword ptr [edx+ecx+3]
004015D3   mov   dword ptr [ebp-23C], eax
004015D9   mov   dword ptr [ebp-238], 0
004015E3   jmp   short 004015F4
004015E5   /mov  ecx, dword ptr [ebp-238] ;下面这段循环，对 bufEncrypt[]异或加密
004015EB   |add  ecx, 1
004015EE   |mov  dword ptr [ebp-238], ecx
004015F4    mov  edx, dword ptr [ebp-238]
004015FA   |cmp  edx, dword ptr [ebp-23C]
00401600   |jge  short 00401625
00401602   |mov  eax, dword ptr [ebp-238]
00401608   |movsx    ecx, byte ptr [ebp+eax-354]
00401610   |xor  ecx, 0A6
00401616   |mov  edx, dword ptr [ebp-238]
0040161C   |mov  byte ptr [ebp+edx-354], cl
00401623   \jmp  short 004015E5
```

原来客户端将输入的 Name 及 Key 按图 5.9 所示的格式处理，并异或加密发送给服务端。

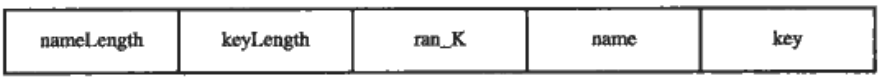


图 5.9　发送的数据包

2. 分析接收的数据包

服务端接收到数据后，经过计算，将包括正确的数据包返回给客户端。客户端程序用 recv 接收数据，相关代码如下：

```
00401655   push  0                          ;/Flags = 0
00401657   push  1F4                        ;|BufSize = 1F4 (500.)
0040165C   lea   eax, dword ptr [ebp-1F4];|
00401662   push  eax                        ;|Buffer
00401663   mov   ecx, dword ptr [ebp-200];|
00401669   push  ecx                        ;|Socket
0040166A   call  <jmp.&WS2_32.recv>         ;\recv
0040166F   mov   dword ptr [ebp-28C], eax
00401675   mov   dword ptr [ebp-238], 0
0040167F   jmp   short 00401690
00401681   /mov  edx, dword ptr [ebp-238]     ;下面这段循环，对接收到的数据异或解密
00401687   |add  edx, 1
0040168A   |mov  dword ptr [ebp-238], edx
00401690    mov  eax, dword ptr [ebp-238]
00401696   |cmp  eax, dword ptr [ebp-28C]
0040169C   |jge  short 004016BE
0040169E   |mov  ecx, dword ptr [ebp-238]
```

```
004016A4    |xor  edx, edx
004016A6    |mov  dl, byte ptr [ebp+ecx-1F4]
004016AD    |xor  edx, 6E                    ;异或解密
004016B0    |mov  eax, dword ptr [ebp-238]
004016B6    |mov  byte ptr [eax+41AE68], dl
004016BC    \jmp  short 00401681
004016BE    xor   ecx, ecx
004016C0    mov   cl, byte ptr [41AE6C]
```

上面这段代码接收到数据，并进行解密，解密后的数据存放在 41AE68h～41AEC1h 这段空间，如图 5.10 所示。

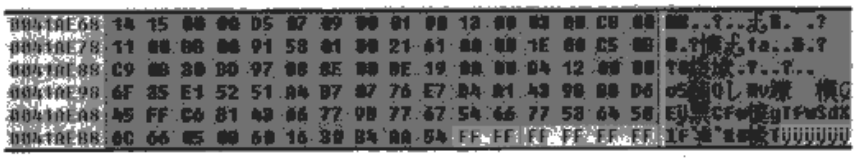


图 5.10　解密后的数据包

接下来程序会从 41AE68h～41AEC1h 读取所需要的字节，因此只要对这段数据下内存读断点，很容易定位到相关代码处。但在实际应用中，程序读取这部分数据可能比较隐蔽，比如运行一段时间再比较，或用到某功能后再比较等。因此有可能遗漏相关的读取代码。

本实例是用全局变量构建缓冲区的，由于是以 Debug 编译的程序，程序里会直接用如下指令读取缓冲区数据。

```
004016C0   8A0D 6CAE4100    mov    cl, byte ptr [41AE6C]
```

一个简单并且有效的办法，是在整个代码里搜索访问 41AE68h～41AEC1h 这段缓存区的 mov 指令。这时，IDA 强大就体现出来了，用 IDA 打开如下脚本，就可将读取指定内存的代码列出了。

```
//code by DarkNessOut
static Getasm(from ,to,range1,range2)
{
    auto ea,cmd,fp,deta,opcode;
    fp=fopen("c:\\code.txt","w");
    for (ea=from;ea<to;)
    {
        cmd=GetMnem(ea);
        if (strstr(cmd,"mov")==0 || strstr(cmd,"lea")==0 )
        {
            opcode=Dword(NextNotTail(ea)-4);

            if(opcode<0){//opcode<0 处理 mov  edx, [ebp-350]指令
                opcode=~opcode;
                opcode++;
            }
            Message("-> %08X %08X\n",ea,opcode);
            if (opcode>=range1 && opcode<=range2)
            {
                deta=opcode-range1;
                fprintf(fp,"%08X %s      //+0x%04X\n",ea,GetDisasm(ea),deta);
                MakeComm (ea,form("// +0x%04X",deta));  //加注释到 IDA 中
            }
        }
        ea=NextNotTail(ea);
    }
```

```
    fclose(fp);
    Message("OK!");
}
```

先用“File/IDC File”打开 getasm.idc 脚本，按“Shift+F2”键打开 IDC 命令行，键入“Getasm(0x401000,0x40F951,0x41AE68,0x0041AEC1);”，如图 5.11 所示。

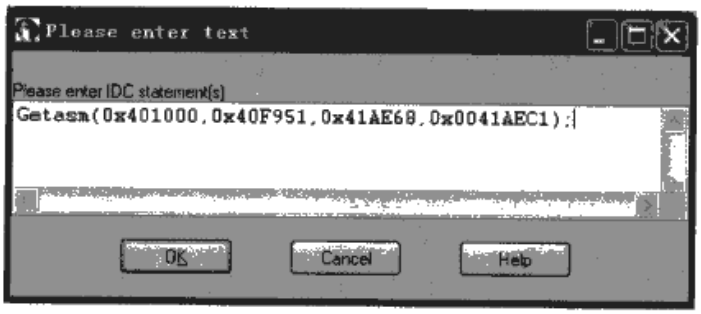


图 5.11　IDC 命令行输入命令

这个脚本就将整个程序访问缓冲区所有指令列出了。打开 c:\code.txt 文件，内容如下：

```
0040113D mov    cl, byte_41AEB0          //+0x0048
004016B6 mov    bufResultEncrypt[eax], dl//+0x0000
004016C0 mov    cl, byte_41AE6C          //+0x0004
004016D0 mov    dl, byte_41AE76          //+0x000E  (随机数 ran_K)
004016E7 mov    cl, byte_41AE72          //+0x000A
004016F7 mov    dl, byte_41AE8B          //+0x0023
00401707 mov    al, byte_41AEA4          //+0x003C
00401713 mov    cl, byte_41AEAA          //+0x0042
```

本例将缓冲区放全局变量，一般情况下，缓冲区一般是放局部变量的，其访问缓冲区数据的指令如下，其中 r/m32 表示 32 位寄存器。

```
mov    r/m16, [r/m32+n]
lea    r/m32, [r/m32+n]
mov    r/m32, [ebp-n]
```

本节提供的 getasm.idc 脚本也支持上述指令，只需要确实指令的范围（这里是 n 的范围），即可得到正确的访问缓冲区的指令。在实际情况中，访问同一数据块可能用不同指针，也可能将数据块复制到其他地方再读取。

3．解除网络验证

发送与接收的封包都已分析出，比较省事的解决方法是写个服务端，模拟服务器接收和发送数据。如果软件是用域名登录服务器的，可以修改 hosts 文件，使得域名指向本地 127.0.0.1。如果直接用 IP 连接服务器，可以用 inet_addr 或 connect 等设断，修改 IP 地址为本地。或使用代理软件将 IP 用代理指向本地。

除了写服务端外，也可直接修改客户端程序，将封包中的数据整合进去，下面主要讲述一下这种改法。

将 CrackMeNet.exe 复制一份，用 OllyDbg 打开，然后将开始截取的正确数据粘贴到 41AE68h～41AEC1h 这段地址处。

第一步，将发包功能（send）去除，再将随机数读取到 41AE76h 处。

```
0040163C    call    0040FAA8 //此处原是 send，现在跳到 0040FAA8 这个空白地址处
{
 0040FAA8    push    eax                          ;保存 eax
 0040FAA9    mov     al,byte ptr [ebp-254]        ;将随机数 ran_K 读到 al 中
 0040FAAF    mov     byte ptr [41AE76], al        ;将 ran_K 写到 41AE76
```

```
0040FAB4    pop     eax                             ;恢复 eax
0040FAB5    retn    10                              ;原 send 函数有 4 个参数入栈，现恢复
}
```

第二步，将 recv 函数去除，并跳过数 据解密代码，修改代码：

```
00401655    jmp     short 004016BE                  ;跳过 recv 函数及解密代码
00401657    push    1F4                             ;|BufSize = 1F4 (500.)
0040165C    lea     eax, dword ptr [ebp-1F4]        ;|
00401662    push    eax                             ;|Buffer
00401663    mov     ecx, dword ptr [ebp-200]        ;|
00401669    push    ecx                             ;|Socket
0040166A    call    <jmp.&WS2_32.#16>               ;\recv
0040166F    mov     dword ptr [ebp-28C], eax
00401675    mov     dword ptr [ebp-238], 0
0040167F    jmp     short 00401690
00401681    /mov    edx, dword ptr [ebp-238         ;以下代码用来解密
00401687    |add    edx, 1
0040168A    |mov    dword ptr [ebp-238], edx
00401690    |mov    eax, dword ptr [ebp-238]
00401696    |cmp    eax, dword ptr [ebp-28C]
0040169C    |jge    short 004016BE
0040169E    |mov    ecx, dword ptr [ebp-238]
004016A4    |xor    edx, edx
004016A6    |mov    dl, byte ptr [ebp+ecx-1F4]
004016AD    |xor    edx, 6E
004016B0    |mov    eax, dword ptr [ebp-238]
004016B6    |mov    byte ptr [eax+41AE68], dl
004016BC    \jmp    short 00401681
004016BE    xor     ecx, ecx
```

经过这样处理，再运行实例，单击“Register”按钮，跳出一对话框提示“Error: Connection failed.”，直接强行跳过即可。修改代码如下：

```
00401496        jnz     short 004014C1
```

从以上分析可以看出，网络验证关键就是数据包分析。数据包可以用一些工具辅助，如 WPE、iris 等。如果数据包加密或要彻底分析数据包处理过程，必须用发送/接收函数设断，跟踪程序对数据包的处理。

5.7 CD-Check

一些采用光盘形式发行的应用软件和游戏在使用时需要检查光盘是否插在光驱中，如果没有，则拒绝运行。这是为了防止用户将软件或游戏的一份正版拷贝安装在多台机器上并同时使用。这和 DOS 时代的钥匙盘保护是类似的，虽然能在一定程度上防止非法拷贝，但也给正版用户带来了一些麻烦：一旦光盘被划伤，用户就无法使用软件了。这里将介绍常见的光盘检测的实现方式，以及如何去除光盘检测的基本知识。其他一些光盘专业保护软件，如 SafeDisc 等很复杂，本节不作讲述。

最简单也最常见的光盘检测就是程序在启动时，判断光驱中的光盘上是否存在特定的文件。如果不存在，则认为用户没有使用正版光盘，拒绝运行。在程序运行的过程中，一般不再检查光盘的存在与否。Windows 下的具体实现一般是这样的：先用 GetLogicalDriveStrings()或 GetLogicalDrives()得到系统中安装的所有驱动器的列表，然后再用 GetDriveType()检查每个驱动器；如果是光驱，则用 CreateFileA()或 FindFirstFileA()函数检查特定的文件存在与否，并且可能进一步地检查文件的属性、大小、内容等。

这种光盘检查是比较容易被破解的，解密者只要利用上述函数设置断点，找到程序启动时检查光驱的

地方，然后修改判断指令就可以跳过光盘检查。

上述保护的一种增强类型，就是把程序运行时所需要的关键数据放在光盘中。这样，即使解密者能够强行跳过程序启动时的检查，由于没有使用正版光盘，也就没有程序运行时所需要的关键数据，程序自然崩溃，这样可以在一定程度上起到防破解的作用。

对付上述这种增强型光盘保护还是有办法的，比如简单地利用刻录、拷贝工具将光盘复制多份。也可采用虚拟光驱程序来模拟正版光盘。常用的虚拟光驱程序有 Virtual CD、Virtual Drive、Daemon Tools 等，尤其值得一提的是 Daemon Tools，它不仅是免费的，而且能够模拟一些加密光盘。这些光盘加密工具一般都在光轨上做文章，比如做暗记等。有的加密光盘可用工作在原始模式（Raw mode）的光盘拷贝程序来原样拷贝，比如用 Padus 公司的 DiscJuggler 和 Elaborate Bytes 公司的 CloneCD 等拷贝工具。对光盘加密感兴趣的读者可以查阅 ISO 9660 标准协议。

5.7.1 相关函数

1. GetDrivetypeA(W)

获取磁盘驱动器类型。

```
UINT GetDriveType(
  LPCTSTR lpRootPathName      // 根路径地址
);
返回值
    0 驱动器不能识别
    1 根目录不存在
    2 移动存储器
    3 固定驱动器（硬盘）
    4 远程驱动器（网络）
    5 CD-ROM 驱动器
    6 RAM disk
```

2. GetLogicalDrives

获取逻辑驱动器符号。

```
该函数没有参数
 返回值：如果失败，就返回零值，否则返回由位掩码表示的当前可用驱动器
bit 0        drive A
bit 1        drive B
bit 2        drive C
……
```

3. GetLogicalDriveStrings

获取当前所有逻辑驱动器的根驱动器路径。

```
DWORD GetLogicalDriveStrings(
    DWORD nBufferLength,    // 缓冲区大小
    LPTSTR lpBuffer         // 缓冲区地址，如成功，则返回结果为如下形式：c:\d:\
);
返回值：如果成功，就返回实际的字符数；否则返回零。
```

4. GetFileAttributes A(W)

判断指定文件的属性。

```
DWORD GetFileAttributes(
    LPCTSTR lpFileName      // 指定欲获取属性的一个文件的名字
);
```

5.7.2 拆解光盘保护

名称：CD_Check，文件：光盘\chap05\

这个程序先用 GetDriveTypeA 检测文件是否在光驱里，再用 CreateFileA 尝试打开光盘文件。如果存在，则成功。

```
00401346   push   dword ptr [ebp-18]
00401349   call   [<&KERNEL32.GetDriveTy>
0040134F   cmp    eax, 3                    ;判断各驱动器是否为硬盘
00401352   je     short 00401392            ;相等，则继续判断下一个驱动器
00401354   lea    eax, dword ptr [ebp-18]
00401357   push   00403058                  ;ASCII "CD_CHECK.DAT"
0040135C   push   eax
0040135D   lea    eax, dword ptr [ebp-20]
00401360   push   eax
00401361   call   <jmp.&MFC42.#924_operator+>
00401366   mov    eax, dword ptr [eax]
00401368   push   ebx                       ;/hTemplateFile
00401369   push   ebx                       ;|Attributes
0040136A   push   ebx                       ;|Mode
0040136B   push   ebx                       ;|pSecurity
0040136C   push   1                         ;|ShareMode = FILE_SHARE_READ
0040136E   push   80000000                  ;|Access = GENERIC_READ
00401373   push   eax                       ;|FileName
00401374   call   [<&KERNEL32.CreateFile>   ;\CreateFileA
0040137A   cmp    eax, -1                   ;尝试打开 CD_CHECK.DAT 文件
0040137D   lea    ecx, dword ptr [ebp-20]
00401380   sete   byte ptr [ebp-D]
00401384   call   <jmp.&MFC42.#800_CString::~CStri>
00401389   cmp    byte ptr [ebp-D], bl
0040138C   je     00401485                  ;如成功则跳转，改成：jmp 00401485
```

5.8 只运行一个实例

Windows 是一个多任务的操作系统，应用程序可以多次运行以形成多个运行实例。但有时出于某种考虑（比如安全性），要求程序只能运行一个实例。

5.8.1 实现方法

只运行一个实例的实现方法有多种，在此只列出几种常见的方法。

1. 查找窗口法

这是最为简单的一种方法。在程序运行前，用 FindWindow、GetWindowText 函数查找具有相同窗口类名和标题的窗口。如果找到了，就说明已经存在一个实例。

```
HWND FindWindowA(W)(
    LPCTSTR lpClassName,  // 指向窗口类名
```

```
    LPCTSTR lpWindowName // 指向窗口文本
  );
返回值：如未找到相符窗口，则返回零。
```

程序中代码如下：

```
TCHAR AppName[ ] = TEXT ("只运行一个实例") ;
hWnd=FindWindow(NULL, AppName);
if (hWnd ==0) 初始化程序
else 退出程序
```

2. 使用互斥对象

尽管互斥对象通常用于同步连接，但用在这个地方也是非常方便的。一般是用 CreateMutex 函数实现，它的作用是创建有名或者无名的互斥对象。

```
HANDLE CreateMutexA(W) (
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // 安全属性
    BOOL bInitialOwner,                      // 指定互斥对象的初始所属身份
    LPCTSTR lpName                           // 指向互斥对象名
    );
返回值：如果函数调用成功，返回值是互斥对象句柄。
```

程序中代码一般如下：

```
TCHAR AppName[] = TEXT ("只运行一个实例") ;
Mutex =CreateMutex(NULL,FALSE,AppName)
if GetLastError<>ERROR_ALREADY_EXISTS
   初始化                                    //如果不存在另一实例
else
   ReleaseMutex(Mutex);
```

3. 使用共享区块

创建一个共享区块（Section），该节拥有读取、写入和共享保护属性，让多个实例共享同一内存。将一个变量放到该区块中作为计数器，该应用程序的所有实例均可以共享该变量，以便通过该变量得知有没有实例在运行。

5.8.2 实例

“序列号保护方式”一节中的 Serial.exe 只能同时运行一个实例，该程序利用了 FindWindow 函数查找指定字串来确定程序是否运行了。对付这类保护最有效的办法是修改应用程序的窗口标题，修改 FindWindow 返回值也能取消其限制。

```
0040100C    push    0
0040100E    push    004020F4
00401013    call    <&USER32.FindWindowA>
00401018    or      eax, eax
0040101A    je      short 0040101D              ;判断点
```

5.9 常用断点设置技巧

设置正确的断点，对调试软件是非常重要的。如果掌握一些 Win32 编程，对设置合适的断点是非常有帮助的。下面分门别类整理出常用的断点集合，方便查阅。

字符串	hmemcpy(仅 Windows 9x) GetDlgItemTextA(W) GetDlgItemInt GetWindowTextA(W) GetWindowWord	注册表	RegCreateKeyA(W) RegDeleteKeyA(W) RegQueryValueA(W) RegCloseKey RegOpenKeyA(W)
文件访问	ReadFile WriteFile CreateFileA(W) SetFilePointer GetSystemDirectory	光驱相关	GetFileAttributes A(W) GetFileSize GetDriveType ReadFile CreateFileA(W)
INI 文件	GetPrivateProfileString GetPrivateProfileInt WritePrivateProfileString WritePrivateProfileInt	对话框	MessageBeep MessageBoxA(W) MessageBoxExA(W) DialogBoxParam A(W) CreateWindowEx A(W) ShowWindow UpdateWindow
时间相关	GetLocalTime GetFileTime GetSystemtime		

第6章 加密算法[①]

现有的序列号加密算法大多是软件开发者自行设计的，大部分相当简单，而且有些算法作者虽然下了很大的工夫，却往往达不到所希望的效果。其实，有很多成熟的算法可用，特别是密码学中的一些强度比较高的算法，比如 RSA、BlowFish、MD5 等。这些算法在因特网上有大量的源码或编译好的库（当然这些库中可能会有些漏洞），可以直接加以利用，所要做的只是利用搜索引擎找到它们并将其嵌入自己的程序中。应当指出，即使这些算法的强度很高，但是使用方法也要得当，否则效果就和普通的四则运算效果没有什么两样了，很容易被解密者算出注册码或者写出注册机来。

6.1 单向散列算法

单向散列函数算法也称 Hash（哈希）算法，是一种将任意长度的消息压缩到某一固定长度（消息摘要）的函数（该过程不可逆）。Hash 函数可用于数字签名、消息的完整性检测、消息起源的认证检测等。常见的散列算法有 MD5、SHA、RIPE-MD、HAVAL、N-Hash 等。

在软件的加密保护中，Hash 函数是经常用到的加密算法。但是，由于 Hash 函数为不可逆算法，所以软件只能使用 Hash 函数作为一个加密的中间步骤。例如，对用户名做一个 Hash 变换，将这个结果再进行一个可逆的加密变换（如对称密码），变换结果为注册码。从解密角度来说，一般不必了解 Hash 函数的具体内容（变种算法除外），只要能识别出是何种 Hash 函数就可以了，然后直接套用相关算法源码实现。

6.1.1 MD5 算法

MD5 消息摘要算法（Message Digest Algorithm）是由 R.Rivest 所设计的。它对输入的任意长度的消息进行运算，产生一个 128 位的消息摘要。近几年来，随着穷举攻击和密码分析的发展，应用最为广泛的 MD5 算法已经不再那么流行了。

1．算法原理

（1）数据填充

填充消息使其长度与 448 模 512 同余（即长度=448 mod 512）。也就是说，填充后的消息长度比 512 的倍数仅小 64 位的数。即使消息长度本身已经满足上述长度要求，仍然需要填充。

填充方法是附一个 1 在消息后面，然后用 0 来进行填充，直到消息的长度与 448 模 512 同余。至少填充 1 位，至多填充 512 位。

---

① 本章由沈晓斌编写。

（2）添加长度

在第一步的结果之后附上64位的消息长度。如果填充前消息的长度大于2^{64}，则只使用其低64位。这时，在添加完填充位和消息长度之后，最终消息的长度正好就是512的整数倍了。

令M[0...N-1]表示最终的消息，其中N是16的倍数。

（3）初始化变量

用到4个变量（A, B, C, D）来计算消息摘要。这里A, B, C, D分别都是一个32位的寄存器。这些寄存器以下面的十六进制数值来初始化：A=01234567h，B=89abcdefh，C=fedcba98h，D=76543210h。

并且在内存中是以低字节在前的形式来存储的，即如下格式：

```
01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

（4）数据处理

以512位分组为单位处理消息，首先定义4个辅助函数，每个都是以3个32位双字作为输入，输出一个32位双字。

F(X,Y,Z) = (X&Y)|((～X)&Z)

G(X,Y,Z) = (X&Z)|(Y&(～Z))

H(X,Y,Z) = X^Y^Z

I (X,Y,Z) = Y^(X|(～Z))

其中，&是与操作，| 是或操作，～是非操作，^ 是异或操作。

这4轮变换是对进入主循环的512位消息分组的16个32位字分别进行如下操作：将A, B, C, D的副本a, b, c, d中的3个经F, G, H, I运算后的结果与第4个相加，再加上32位字和一个32位字的加法常数，并将所得之值循环左移若干位，最后将所得结果加上a, b, c, d之一，并回送至A, B, C, D，由此完成一次循环。

所用的加法常数由这样一张表T[i]来定义，其中i为1至64之中的值，T[i]等于4294967296乘以abs(sin(i))所得结果的整数部分，其中i用弧度来表示。这样做是为了通过正弦函数和幂函数来进一步消除变换中的线性。

接着进行如下的操作（伪码表示）：

```
For i = 0 to N/16-1 do   /* 处理每一块分组 */
 For j = 0 to 15 do      /* 把分组i复制到块X */
  Set X[j] to M[i*16+j]
 End
 /* 把A,B,C,D分别保存为AA,BB,CC,DD */
 AA=A
 BB=B
 CC=C
 DD=D
 /* 第一轮，令[ABCD K S I]表示如下操作
   A=B+((A+F(B,C,D)+X[K]+T[I])<<<S)
   做如下的16次操作
 */
 [ABCD 0 7 1] [DABC 1 12  2] [CDAB  2 17  3] [BCDA  3 22  4]
 [ABCD 4 7 5] [DABC 5 12  6] [CDAB  6 17  7] [BCDA  7 22  8]
 [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
 [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
 /* 第二轮，令[ABCD K S I]表示如下操作
   A=B+((A+G(B,C,D)+X[K]+T[I])<<<S)
   做如下的16次操作
 */
```

```
  [ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
  [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
  [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
  [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
  /* 第三轮，令[ABCD K S I]表示如下操作
     A=B+((A+H(B,C,D)+X[K]+T[I])<<<S)
     做如下的16次操作
  */
  [ABCD 5 4 33] [DABC 8 11 14] [CDAB 11 16 35] [BCDA 14 23 36]
  [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
  [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
  [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
  /* 第四轮，令[ABCD K S I]表示如下操作
     A=B+((A+I(B,C,D)+X[K]+T[I])<<<S)
     做如下的16次操作
  */
  [ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
  [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
  [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
  [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
  /* 然后做如下的加法运算 */
  A=A+AA
  B=B+BB
  C=C+CC
  D=D+CC
End
```

（5）输出

当所有的512位分组都运算完毕后，ABCD的级联将被输出为MD5散列的结果。以上是MD5算法的简单描述，更为详细的实现程序，请参考光盘中的源代码。

2. MD5在加解密中的应用

MD5将任意长度的字符串变换成一个128位的大整数，并且是不可逆的。换句话说，即便MD5算法的源代码满天飞，使得任何人都可以了解MD5的详尽算法描述，但是绝对没有任何人可以将一个经由MD5算法加密过的字符串还原回原始的字符串。

在实际过程中，若单向散列算法运用不当，是没有什么保护效果的。看一看下面使用MD5判断注册码的伪码：

```
if (MD5(用户名) == 序列号)
    正确的注册码;
else
    错误的注册码;
```

由于正确的序列号以明文形式出现在内存中，因此解密者很容易找到正确的注册码，写注册机只要识别出程序采用的是MD5算法就够了。遗憾的是，不少软件作者都偏爱这种判断注册码的方法。

MD5代码的特点非常明显，跟踪时很容易发现。如果软件采用MD5算法，在数据初始化的时候必然会用到上面提到的4个常数（A,B,C,D）。实际上，像KANAL这样的算法分析工具不是通过上面的这4个常数来鉴别MD5，而是通过识别具有64个常量元素的表T来确定是不是MD5算法的。对于变形的MD5算法，常见的有三种情况：一是改变初始化时所用到的4个常数；二是改变填充的方法；三是改变Hash变换的处理过程。在解密时，只要跟踪到以上提到的这些点，然后相应地对MD5的源代码进行修改，就可以实现相应的注册机制了。

3. 实例分析

以光盘中的 MD5KeyGenMe 为例，讲述 MD5 算法在软件保护中的应用。先用 PEiD 插件 Krypto ANALyzer 分析光盘中的 MD5KeyGenMe.exe，得知该 KeyGenMe 含有 MD5 的迭代（压缩）常数，猜测可能使用了 MD5 算法，如图 6.1 所示。

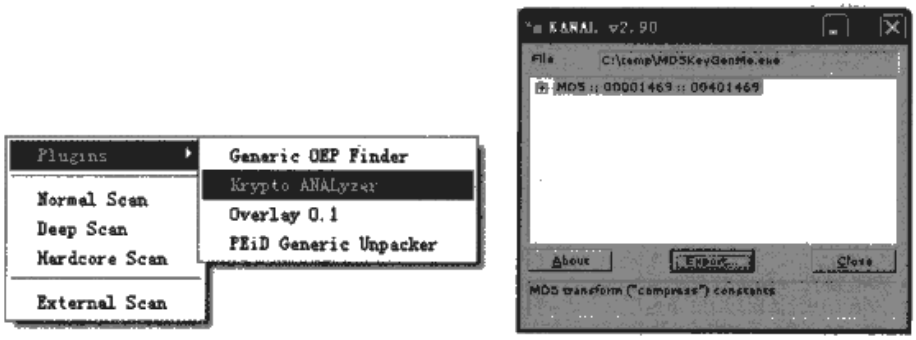


图 6.1 用 PEiD 的插件扫描目标程序的加密算法

用 OllyDbg 打开实例 MD5KeyGenMe.exe，按 F9 键运行 KeyGenMe。在命令栏中输入 bp GetDlgItemTextA，分别输入 Name：pediy 和 Serial Number：0123456789ABCDEF，单击“Check”按钮，程序将中断在 GetDlgItemTextA 函数开始处，按 F8 键来到实例代码中。

```
0040116F  call    ebp                             ;GetDlgItemTextA
00401171  mov     edi, eax                        ;eax 中的值是 Name 的长度
00401173  cmp     edi, ebx
00401175  je      00401289
0040117B  lea     edx, dword ptr [esp+60]
0040117F  push    0C9                             ;/Count = C9 (201.)
00401184  push    edx                             ;|Buffer
00401185  push    3E9                             ;|ControlID = 3E9 (1001.)
0040118A  push    esi                             ;|hWnd
0040118B  call    ebp                             ;\GetDlgItemTextA
0040118D  cmp     eax, 13                         ;比较输入的注册码长度是否为 0x13
00401190  jnz     00401289
00401196  mov     cl, byte ptr [esp+64]           ;注册码第 5 个字符是"-"吗
0040119A  mov     al, 2D
0040119C  cmp     cl, al
0040119E  jnz     00401289
004011A4  cmp     byte ptr [esp+69], al           ;注册码第 10 个字符是"-"吗
004011A8  jnz     00401289
004011AE  cmp     byte ptr [esp+6E], al           ;注册码第 15 个字符是"-"吗
004011B2  jnz     00401289
004011B8  mov     ecx, dword ptr [esp+65]
004011BC  mov     eax, dword ptr [esp+60]
004011C0  mov     edx, dword ptr [esp+6A]
004011C4  mov     dword ptr [esp+14], ecx
004011C8  mov     dword ptr [esp+10], eax
004011CC  mov     eax, dword ptr [esp+6F]
004011D0  lea     ecx, dword ptr [esp+128]
004011D7  mov     dword ptr [esp+18], edx
004011DB  push    ecx                             ;MD5 Context
004011DC  mov     dword ptr [esp+20], eax
004011E0  call    004012B0
```

在上面的这段代码中，004011DB 处的“push ecx, ecx”中即为 MD5 Context 地址，将用来存储 MD5 的结构。跟进 004012B0 可以看到如下代码：

```
004012B0  mov     eax, dword ptr [esp+4]
004012B4  xor     ecx, ecx
004012B6  mov     dword ptr [eax+14], ecx
004012B9  mov     dword ptr [eax+10], ecx
004012BC  mov     dword ptr [eax]   , 67452301
004012C2  mov     dword ptr [eax+4], EFCDAB89
004012C9  mov     dword ptr [eax+8], 98BADCFE
004012D0  mov     dword ptr [eax+C], 10325476
004012D7  retn
```

根据代码中的 4 个常数，很明显这是在进行 MD5 初始化。但是，仅仅根据这一点还不能完全认定这就是 MD5，需要进一步的判定。接下来进行信息的填充，首先把 Name 填充到 Context 中，接着填充一固定的字符串“www.pediy.com”，作为附加消息。在填充的过程中，若信息的长度满足一定的条件，将执行 MD5 Transform 操作，即对消息进行处理。代码如下：

```
00401431  not     eax
00401433  mov     ecx, ebx
00401435  and     eax, ebp
00401437  and     ecx, edi
00401439  or      eax, ecx
0040143B  mov     ecx, dword ptr [esp+1C]
0040143F  add     eax, ecx
00401441  lea     ecx, dword ptr [edx+eax+D76AA478]  ;正常数表 T 中的元素
00401448  mov     edx, edi
0040144A  mov     eax, ecx
0040144C  shr     eax, 19
0040144F  shl     ecx, 7
00401452  or      eax, ecx
```

对于在地址 00401441 处所出现的 D76AA478h，是前面所提到的 MD5 中的正弦函数表中的元素之一，在后续代码中出现了表中剩下的其他元素，并且根据这些代码所特有的操作特征（MD5 的 F,G,H,I 函数），就可以断定是 MD5 无疑了。

相关代码如下：

```
0040122A  push    ecx
0040122B  push    edx                     ;保存散列值的缓冲区地址
0040122C  call    00401390
```

经过上面的这个 call 之后，最终的散列值将保存在 edx（0012F824）中，如下：

```
0012F824  51 6A D5 A3 97 48 08 EE B5 C6 F0 06 AD FF E2 1F
```

上面这段代码计算出来的 MD5 值，相当于将输入的 Name 字符与“www.pediy.com”连接，再计算其 MD5 值。

```
00401231  add     esp, 14
00401234  xor     eax, eax
00401236  /mov    cl, [esp+eax+180]       ;取 Hash 值
0040123D  |and    ecx, 1F                 ;与 0x1F 相与，也就是取 ecx%32 的值
00401240  |inc    eax
00401241  |cmp    eax, 10
00401244  |mov    dl, [esp+ecx+3C]        ;查 base32 表
00401248  |mov    [esp+eax+30F], dl       ;将结果存入一缓冲区
```

```
0040124F  \jl     short 00401236
00401251  lea     eax, [esp+310]
00401258  lea     ecx, [esp+10]
0040125C  push    eax                           ;/String2  正确的注册码（不包括“-”）
0040125D  push    ecx                           ;|String1  输入的注册码（不包括“-”）
0040125E  call    [<&KERNEL32.lstrcmpA>]        ;\lstrcmpA
00401264  test    eax, eax
00401266  jnz     short 00401289
```

至此，该 KeyGenMe 的注册验证流程大致分析完了，可以根据此流程做出注册机。注册机源代码见本书光盘。

6.1.2 SHA 算法

安全散列算法（Secure Hash Algorithm）简称 SHA，有 SHA-1、SHA-256、SHA-384 和 SHA-512 几种，分别产生 160 位、256 位、384 位和 512 位的散列值。

下面以 SHA-1 为例，对 SHA 系列散列函数作简要介绍。

1．算法描述

SHA-1 是一种主流的散列加密算法，其设计时基于和 MD4 相同的原理，并且模仿了该算法。消息分组和填充方式与 MD5 相同。

SHA-1 使用了 $f_0, f_1, \cdots, f_{79}$ 这样一个逻辑函数序列。每一个 $f_t(0 \leqslant t \leqslant 79)$ 对 3 个 32 位双字 B,C,D 进行操作，产生一个 32 位双字的输出。f_t(B,C,D)定义如下：

$$f_t(B,C,D)=\begin{cases}(B\&C)\mid((\sim B)\&D) & 0\leqslant t\leqslant 19\\ B\hat{}C\hat{}D & 0\leqslant t\leqslant 39\\ (B\&C)\mid(B\&D)\mid(C\&D) & 40\leqslant t\leqslant 59\\ B\hat{}C\hat{}D & 60\leqslant t\leqslant 79\end{cases}$$

需要注意的是，在常见的 SHA-1 实现的程序中，这 4 个函数经常被定义成如下的形式（C 语言）：

```
#define F0(x,y,z)  (z^(x&(y^z)))
#define F1(x,y,z)  (x^y^z)
#define F2(x,y,z)  ((x&y) | (z&(x|y)))
#define F3(x,y,z)  (x^y^z)
```

这与上面的 f_t(B,C,D)的定义是等价的，将产生相同的散列值。

类似于 MD5，SHA-1 也使用了一系列的常数 $K(0), K(1), \cdots, K(79)$。以十六进制形式表示如下：

$$K_t=\begin{cases}5A827999 & 0\leqslant t\leqslant 19\\ 6ED9EBA1 & 20\leqslant t\leqslant 39\\ 8F1BBCDC & 40\leqslant t\leqslant 59\\ CA62C1D6 & 60\leqslant t\leqslant 79\end{cases}$$

SHA-1 产生 160 位的消息摘要，在对消息进行处理之前，初始散列值 H 先用 5 个 32 位双字初始化，这 5 个双字以十六进制形式表示如下：

$H_0 = 67452301h$

$H_1 = EFCDAB89h$

$H_2 = 98BADCFEh$

$H_3 = 10325476h$

$H_4 = C3D2E1F0h$

在解密者尝试分析算法时，可以通过上面的常数 K_t 及 H_t 来识别其为 SHA-1。

关于 SHA-1、SHA-256、SHA-384 和 SHA-512 更加详细的计算过程，请参考 FIPS 180-1 及 FIPS 180-2。最后附上 SHA-256、SHA-384 和 SHA-512 初始化数据（十六进制形式）。

SHA-256 初始化数据是：

```
6A09E667 BB67AE85 3C6EF372 A54FF53A
510E527F 9B05688C 1F83D9AB 5BE0CD19
```

SHA-384 初始化数据是：

```
CBBB9D5DC1059ED8 629A292A367CD507
9159015A3070DD17 152FECD8F70E5939
67332667FFC00B31 8EB44A8768581511
DB0C2E0D64F98FA7 47B5481DBEFA4FA4
```

SHA-512 初始化数据是：

```
6A09E66713BCC908 BB67AE8584CAA73B
3C6EF372FE94F82B A54FF53A5F1D36F1
510E527FADE682D1 9B05688C2B3E6C1F
1F83D9ABFB41BD6B 5BE0CD19137E2179
```

2. 实例分析

用 PEiD 的 Krypto ANALyzer 插件查看光盘中的 SHA1KeyGenMe.exe，得知该 KeyGenMe 中含有 SHA-1 的压缩常数，猜测可能使用了 SHA-1 算法（见图 6.2）。

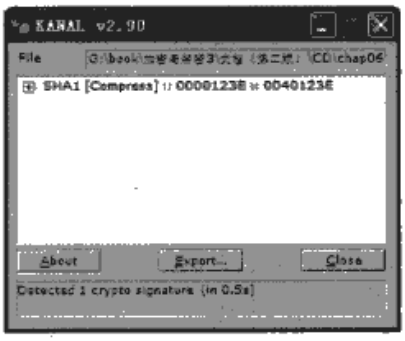


图 6.2 用 PEiD 的插件扫描目标程序的加密算法

用 OllyDbg 打开光盘中的 SHA1KeyGenMe.exe，在命令栏中输入“bpx GetDlgItemTextA”（bpx 命令可以将断点下在所有调用 GetDlgItemTextA 的代码上），按 F9 键运行 KeyGenMe，分别输入 Name：pediy 和 Serial Number：0123456789ABCDEFGHIJ，单击“Check”按钮，程序将中断在第一个 GetDlgItemTextA 处。

```
004014CD  call   edi                     ;GetDlgItemTextA，取输入的Name
004014CF  mov    esi, eax
004014D1  cmp    esi, ebx
004014D3  je     004015EA
004014D9  lea    edx, [esp+298]
004014E0  push   0C9                     ;/Count = C9 (201.)
```

```
004014E5  push   edx                          ;|Buffer
004014E6  push   3E9                          ;|ControlID = 3E9 (1001.)
004014EB  push   ebp                          ;|hWnd
004014EC  call   edi                          ;\GetDlgItemTextA
004014EE  cmp    eax, 14                      ;序列号长度为 0x14 位（十进制是 20 位）
004014F1  jnz    004015EA
004014F7  lea    eax, [esp+360]               ;SHA-1 结构
004014FE  push   eax
004014FF  call   00401000                     ;SHA-1 初始化函数
{
        00401000  /mov    edx, [esp+4]
        00401004  |push   edi
        00401005  |mov    ecx, 50
        0040100A  |xor    eax, eax
        0040100C  |lea    edi, [edx+28]
        0040100F  |rep    stos dword ptr es:[edi]
        00401011  |mov    [edx+4], eax
        00401014  |mov    [edx], eax

--------------------------------------------
下面这段代码正是标准 SHA-1 算法的散列值初始化的标志
--------------------------------------------
        00401016  |mov    dword ptr [edx+08],67452301
        0040101D  |mov    dword ptr [edx+0C],EFCDAB89
        00401024  |mov    dword ptr [edx+10],98BADCFE
        0040102B  |mov    dword ptr [edx+14],10325476
        00401032  |mov    dword ptr [edx+18],C3D2E1F0
        00401039  |pop    edi
        0040103A  \retn
}
00401504  add        esp, 4
00401507  xor        edi, edi
00401509  cmp        esi, ebx
0040150B  jle        short 0040152B
0040150D  /movsx ecx, byte [esp+edi+1D0]  ;取出 Name 的字符进行消息填充
00401515  |lea   edx, [esp+360]
0040151C  |push  ecx
0040151D  |push  edx
0040151E  |call  00401040                  ;对消息进行填充及 Hash 处理的函数
00401523  |add   esp, 8
00401526  |inc   edi
00401527  |cmp   edi, esi
00401529  \jl    short 0040150D
0040152B  lea    eax, [esp+108]
00401532  lea    ecx, [esp+360]
00401539  push   eax                          ;保存散列值的缓冲区地址
0040153A  push   ecx                          ;包含待散列消息的 SHA-1 结构
0040153B  call   004012A0                     ;SHA-1 最终散列函数
00401540  add    esp, 8
00401543  xor    eax, eax
00401545  /mov   dl, [esp+eax+34]             ;字符串“PEDIY Forum”
00401549  |mov   cl, [esp+eax+108]           ;20 个字节的散列值
00401550  |xor   dl, cl                      ;散列值与字符串进行异或
00401552  |mov   [esp+eax+40], dl            ;结果保存在缓冲区 szBuffer 中
00401556  |inc   eax
00401557  |cmp   eax, 11                     ;散列值的前 17 个字节
0040155A  \jl    short 00401545
0040155C  cmp    eax, 14
```

```
0040155F  jge     short 0040157C
00401561  lea     ecx, [esp+28]
00401565  sub     ecx, 11
00401568  /mov    dl, [ecx+eax]              ;字符串“pediy.com”
0040156B  |xor    dl, [esp+eax+108]          ;与散列值的最后3个字节分别进行异或
00401572  |inc    eax
00401573  |cmp    eax, 14
00401576  |mov    [esp+eax+3F], dl           ;结果保存在缓冲区szBuffer中
0040157A  \jl     short 00401568
0040157C  mov     ebx, [<&USER32.wsprintfA>]
00401582  xor     esi, esi
00401584  lea     edi, [esp+10]
00401588  /mov    al, [esp+esi+4A]           ;szBuffer后10个字节
0040158C  |mov    cl, [esp+esi+40]
00401590  |xor    cl, al                     ;与前10个字节进行异或操作
00401592  |mov    al, cl
00401594  |mov    [esp+esi+40], cl
00401598  |and    eax, 0FF
0040159D  |push   eax
0040159E  |push   0040604C                   ;ASCII "%02X"
004015A3  |push   edi
004015A4  |call   ebx                        ;以十六进制格式输出（wsprintfA）
004015A6  |add    esp, 0C
004015A9  |inc    esi
004015AA  |add    edi, 2
004015AD  |cmp    esi, 0A
004015B0  \jl     short 00401588
004015B2  lea     ecx, [esp+298]
004015B9  lea     edx, [esp+10]
004015BD  push    ecx                        ;/String2
004015BE  push    edx                        ;|String1
004015BF  call    [<&KERNEL32.lstrcmpA>]     ;\lstrcmpA  比较结果
```

通过上面的代码，可以得知注册码，由此可以做出注册机了。注册机源代码见本书光盘。

6.1.3 小结

以上简要介绍了两种常见的单向散列函数加密算法 MD5 和 SHA-1。除此之外，密码学中的 Hash 算法还有很多种，如 RIPEMD、HAVAL、Tiger 等，感兴趣的读者可以参考相关的资料。

需要引起重视的是，随着密码分析技术的发展，现有的散列算法都是不安全的。如 SHA-160、MD5、RIPEMD、HAVAL、Tiger 在某些条件下能够构造出碰撞。软件保护人员在使用散列算法进行保护时，建议选择 SHA-256/384/512，或者使用 Whirlpool。

如果在解密时碰到 Hash 算法，一般只要根据每种 Hash 算法的特征搞清楚是哪一种 Hash 算法以及该算法是否变形，继而通过该 Hash 的源代码即可做出注册机。

6.2 对称加密算法

对称加密算法的加密密钥和解密密钥是完全相同的。其安全性依赖于以下两个因素：第一，加密算法必须是足够强的，仅仅基于密文本身去解密信息在实践中是不可能的，可以抵抗现有的各种密码分析方法的攻击；第二，加密安全性依赖于密钥的秘密性，而不是算法的保密性。

若要采用对称算法检验注册码，正确的使用方法是把用户输入的注册码（或者注册码的一部分，注册码的散列值）作为加密算法或者解密算法的密钥。这样，解密者要想找到一个正确的注册码，只能采用穷

举法。为了增大穷举的难度，自然要求注册码有一定的位数。如果在检查注册码时，把用户输入的注册码作为算法的输入或者输出，则无论使用加密算法还是解密算法检查注册码，解密者都可利用调试器在内存中找到所用的密钥，从而可以将算法求逆，写出注册机来。

常见的对称分组加密算法有 DES（Data Encryption Standard）、IDEA（International Data Encryption Algorithm）、AES（Advanced Encryption Standard）、BlowFish、Twofish 等。本节将以常见的对称加密算法 TEA、IDEA、BlowFish、AES 及流密码 RC4 为例，介绍对称算法在软件保护中的应用。感兴趣的读者可以进一步阅读密码学相关书籍，如《对称密码学》（机械工业出版社）来了解更多的关于对称密码的知识。

6.2.1 RC4 流密码

RC4 于 1987 年由 Ron Rivest 设计，当时作为商业秘密并未公开。1994 年，其算法描述被匿名发表在 Cypherpunks 邮件列表中，不久传到 sci.crypt 新闻组进而在互联网上流传开来。感兴趣的读者可以在这里阅读当时发到 sci.crypt 的原始帖子：http://groups.google.com/group/sci.crypt/msg/10a300c9d21afca0。时至今日，RC4 已经成为最为流行的流密码，如应用于 SSL（Secure Sockes Layer）、WEP。随着众多的对其密码分析成果，密码学家认为 RC4 的安全性不是很强，但在实际应用中还是可以保证一定安全性的。

1. 算法原理

RC4 生成一种称为密钥流的伪随机流，它同明文通过异或操作相混合以达到加密的目的，解密时，同密文进行异或操作。其密钥流的生成由两部分组成：KSA 和 PRGA。

（1）KSA (the Key-Scheduling Algorithm)

RC4 首先使用密钥调度算法（KSA）来完成对大小为 256 的字节数组 S 的初始化及替换。在替换时使用密钥。其密钥的长度一般取 5～16 个字节，即 40～128 位，也可以更长，通常不超过 256 位。首先用 0～255 初始化数组 S，然后使用密钥进行替换。伪代码如下：

```
for i=0 to 255 do
    S[i]:=i;
j:=0;
for i= 0 to 255 do
    j:=(j+S[i]+key[i mod keylength]) mod 256; // 重复使用密钥
    Swap(S[i],S[j]);                          // 交换 S[i]，S[j]
```

（2）PRGA (the Pseudo-Random Generation Algorithm)

数组 S 在完成初始化之后，输入密钥便不再被使用。密钥流的生成是从 S[0]到 S[255]，对每个 S[i]，根据当前 S 的值，将 S[i]与 S 中的另一字节置换。当 S[255]完成转换后，操作继续重复执行。伪代码如下：

```
i,j=0;
while(明文未结束)
   i=(i+1) mod 256;
   j=(j+S[i]) mod 256;
   Swap(S[i],S[j]);
   t=(S[i]+S[j]) mod 256;
   k=S[t];
```

得到的子密码 k 用以和明文进行 XOR 运算，得到密文，解密过程也完全相同。由于 RC4 算法加密采用的是 XOR，所以，一旦子密钥序列出现了重复，密文就有可能被破解。推荐在使用 RC4 算法时，必须对加密密钥进行测试，判断其是否为弱密钥。

2. 实例分析

RC4 算法简单易懂，本书光盘中附有一个使用 RC4 算法对消息进行加密的例子 RC4 Sample，请读者参考详细源代码。下面是相关的汇编代码：

```
00401319  push    8
0040131B  rep     stos dword ptr es:[edi]
0040131D  lea     ecx, dword ptr [esp+10]
00401321  lea     edx, dword ptr [esp+218]
00401328  push    ecx
00401329  push    edx
0040132A  call    00401000                    ;初始化数组 S
0040132F  lea     eax, dword ptr [esp+20]
00401333  push    ebp                         ;待加密数据长度
00401334  lea     ecx, dword ptr [esp+224]
0040133B  push    eax                         ;待加密数据
0040133C  push    ecx                         ;数组 S
0040133D  call    00401070                    ;加密函数
00401342  add     esp, 18
```

RC4 加密与解密都是调用 XOR 指令，加密与解密都是调用同一个函数（本例是 call 00401070），其密钥也相同。

6.2.2 TEA 算法

TEA 全称为 Tiny Encryption Algorithm，于 1994 年由英国剑桥大学的 David J. Wheeler 发明。

1. 算法原理

TEA 的分组长度为 64 位，密钥长度为 128 位。采用 Feistel 网络。其作者推荐使用 32 次循环加密，即 64 轮。其加密过程如下所示：其中 K[0]～K[3]为密钥，v[0]～v[1]为待加密的消息：

```
void Encrypt(long* v, long* k)
{
   unsigned long y=v[0], z=v[1], sum=0, /* 初始化 */
   delta=0x9e3779b9 , /* 密钥调度常数 */
   n=32;
   while(n-->0)   /* 基本循环开始  */
   {
      sum+=delta;
      y+=((z<<4)+k[0])^(z+sum)^((z>>5)+k[1]);
      z+=((y<<4)+k[2])^(y+sum)^((y>>5)+k[3]);
   } /* 循环结束 */
   v[0]=y;
   v[1]=z;
}
```

其中，delta 是由黄金分割点得来的，$\text{delta} = (\sqrt{5}-1)\times 2^{31}$。解密算法是加密的逆过程，代码如下所示。

```
void Decrypt(long* v, long* k)
{
   unsigned long n=32, sum, y=v[0], z=v[1] /* 初始化 */
   delta=0x9e3779b9; /* 密钥调度常数 */
   sum=delta<<5;      /* 即 0xC6EF3720 */
   while(n-->0)       /* 基本循环开始  */
   {
      z-=((y<<4)+k[2])^(y+sum)^((y>>5)+k[3]);
      y-=((z<<4)+k[0])^(z+sum)^((z>>5)+k[1]);
      sum-=delta;
   } /* 循环结束 */
   v[0]=y;
   v[1]=z;
}
```

由以上 TEA 的加密与解密源码可以看出其算法简单易懂，容易实现。但是 TEA 存在相当大的缺陷，如相关密钥攻击。考虑到 TEA 的缺陷，密码学家也相继提出了一些改进算法，比如 XTEA。

2. 实例分析

本书光盘中的 TEAKeyGenMe.exe 是一个以 TEA 及 MD5 算法保护的 KeyGenMe，以此为例介绍 TEA 在软件保护中的应用。

用 OllyDbg 加载 TEAKeyGenMes.exe，按 F9 键运行输入用户名和假码，用“bpx GetDlgItemTextA”下断可以来到如下代码处：

```
004011D8  call    esi                     ;\GetDlgItemTextA
004011DA  cmp     eax, ebx                ;判断是否输入了用户名
004011DC  mov     [esp+28], eax
004011E0  je      00401361
004011E6  lea     edx, [esp+214]
004011ED  push    0C9                     ;Count = C9 (201.)
004011F2  push    edx                     ;|Buffer
004011F3  push    3E9                     ;|ControlID = 3E9 (1001.)
004011F8  push    ebp                     ;|hWnd
004011F9  call    esi                     ;\GetDlgItemTextA
004011FB  cmp     eax, 10
004011FE  jnz     00401361
......
;上面的一段代码是判断注册码是否为十六进制字符
00401251  xor     esi, esi
00401253  lea     edi, [esp+214]
0040125A  /lea     eax, [esp+esi+2C]
0040125E  |push    eax
0040125F  |push    0040808C               ;ASCII "%02X"
00401264  |push    edi
00401265  |call    00401F10               ;sscanf 函数
0040126A  |add     esp, 0C
0040126D  |inc     esi
0040126E  |add     edi, 2
00401271  |cmp     esi, 8
00401274  \jl      short 0040125A
00401276  mov     ecx, [esp+2C]           ;szBuffer 的前 4 个字节
0040127A  mov     edx, [esp+30]           ;szBuffer 的后 4 个字节
0040127E  lea     eax, [esp+1BC]
00401285  mov     [esp+10], ecx           ;转存到另一缓冲区，记为 dwMessage
00401289  push    eax
0040128A  mov     [esp+18], edx
0040128E  call    00401380
00401293  mov     ecx, [esp+2C]
00401297  lea     edx, [esp+2E0]
0040129E  push    ecx
0040129F  lea     eax, [esp+1C4]
004012A6  push    edx
004012A7  push    eax
004012A8  call    004013B0
004012AD  lea     ecx, [esp+1CC]
004012B4  lea     edx, [esp+104]
004012BB  push    ecx
004012BC  push    edx
004012BD  call    00401460
```

```
————————————————————————————————————————————————————————
根据前面章节对 MD5 的介绍，可以判断出 00401380，004013B0 及 00401460 这三处函数分别为
MD5Init，MD5Update，MD5Final 函数。即将用户名进行散列，结果保存在 szHash 缓冲区
————————————————————————————————————————————————————————
004012C2  mov     ecx, [esp+110]          ;下面将 szHash 转存到另一缓冲区，作为
004012C9  mov     edx, [esp+114]          ;TEA 的密钥，记为 TEA_Key
004012D0  mov     eax, [esp+10C]
004012D7  mov     [esp+34], ecx
004012DB  mov     [esp+38], edx
004012DF  lea     ecx, [esp+30]
004012E3  mov     [esp+30], eax
004012E7  mov     eax, [esp+118]
004012EE  lea     edx, [esp+28]
004012F2  push    ecx                     ;TEA_Key
004012F3  push    edx                     ;dwMessage
004012F4  mov     [esp+44], eax
004012F8  call    00401000
{
    ......
    00401006  mov     esi, [esp+20]       ;TEA_Key
    0040100A  mov     ecx, [esp+1C]       ;dwMessage
    0040100E  push    edi
    0040100F  mov     edi, [esi+4]        ;TEA_Key[1]
    00401012  xor     edx, edx
    00401014  mov     eax, [ecx]          ;dwMessage[0]->y
    00401016  mov     ecx, [ecx+4]        ;dwMessage[1]->z
    00401019  mov     [esp+10], edi
    0040101D  mov     edi, [esi]          ;TEA_Key[0]
    0040101F  mov     [esp+24], edi
    00401023  mov     edi, [esi+C]        ;TEA_Key[3]
    00401026  mov     esi, [esi+8]        ;TEA_Key[2]
    00401029  mov     [esp+18], edi
    0040102D  mov     [esp+14], esi
    00401031  mov     edi, 20             ;循环变量 n=32
    00401036  /mov    ebx, [esp+24]
    0040103A  |mov    ebp, [esp+10]
    0040103E  |mov    esi, ecx            ;z，即 dwMessage[1]
    00401040  |sub    edx, 61C88647       ;edx-0x61C88647=0x9e3779b7,
                                          ;即前面提到过的密钥调度常量
    00401046  |shl    esi, 4              ;z 左移 4 位
    00401049  |add    esi, ebx            ;z 左移后的结果再加上 TEA_Key[0]
    0040104B  |mov    ebx, ecx
    0040104D  |shr    ebx, 5              ;dwMessage[1]右移 5 位
    00401050  |add    ebx, ebp            ;右移 5 位后的结果再加上 TEA_Key[1]
    00401052  |mov    ebp, [esp+18]
    00401056  |xor    esi, ebx            ;将左移 4 位并加上 TEA_Key[0]的结果同右
    00401058  |lea    ebx, [edx+ecx]      ;移 5 位并加上 TEA_Key[1]的结果进行异或
                                          ;紧接着计算 z+sum 的值
    0040105B  |xor    esi, ebx            ;将上面的两个结果再次进行异或送入 esi
    0040105D  |mov    ebx, [esp+14]
    00401061  |add    eax, esi            ;y+=esi，这是 TEA 的第一轮
    00401063  |mov    esi, eax            ;y->esi
    00401065  |shl    esi, 4              ;esi 左移 4 位
    00401068  |add    esi, ebx            ;esi+=TEA_Key[2]
    0040106A  |mov    ebx, eax            ;y->ebx
    0040106C  |shr    ebx, 5              ;ebx 右移 5 位
```

```
0040106F  |add    ebx, ebp              ;ebx+=TEA_Key[3]
00401071  |xor    esi, ebx              ;esi^=ebx
00401073  |lea    ebx, [edx+eax]        ;ebx=y+sum
00401076  |xor    esi, ebx              ;esi^=ebx
00401078  |add    ecx, esi              ;将 esi 与 z 相加，完成 TEA 的第二轮
0040107A  |dec    edi                   ;进行 32 次循环，即完成 64 轮运算
0040107B  \jnz    short 00401036
0040107D  mov     edx, [esp+20]
00401081  pop     edi
00401082  pop     esi
00401083  pop     ebp
00401084  mov     [edx], eax            ;将最终的加密结果送回 dwMessage
00401086  mov     [edx+4], ecx
}
```

从上面的汇编代码分析可以判断出，这是 TEA 加密函数，使用用户名的 128 位散列，对用户输入的注册码的十六进制，共 64 位，进行加密。

```
004012FD  mov     eax, [esp+30]
00401301  mov     ecx, [esp+34]
00401305  add     esp, 20
00401308  mov     [esp+2C], eax         ;将 TEA 加密后的结果送回缓冲区 szBuffer
0040130C  mov     [esp+30], ecx
00401310  xor     eax, eax
00401312  /mov    dl, [esp+eax+F4]      ;用户名的 MD5 散列值
00401319  |mov    bl, [esp+eax+2C]      ;TEA 的加密结果
0040131D  |xor    bl, dl
0040131F  |mov    [esp+eax+2C], bl      ;将 TEA 的加密结果同散列值前 64 位异或
00401323  |inc    eax
00401324  |cmp    eax, 8
00401327  \jl     short 00401312
00401329  lea     eax, [esp+2C]
0040132D  lea     ecx, [esp+FC]         ;散列值的低 64 位
00401334  push    eax                   ;/String2
00401335  push    ecx                   ;|String1
00401336  call    [<&KERNEL32.lstrcmpA>] ;\lstrcmpA
0040133C  test    eax, eax
```

上面的结果是将 TEA 的加密结果，共 64 位，同散列值的高 64 位进行异或，异或的结果同散列值的低 64 位进行比较，若二者相同，则注册码正确，否则注册码不正确。

整个注册码验证的逆过程即首先对用户名进行 MD5 散列，并将其作为 TEA 的密钥。将散列的高 64 位同低 64 位进行异或，将异或的结果存储于缓冲区 szBuffer 中，然后用 TEA 对 szBuffer 进行解密，最终输出解密结果的 ASCII 字符即为注册码。其详细源代码见本书光盘。

6.2.3 IDEA 算法

IDEA（International Data Encryption Algorithm，国际数据加密算法），于 1991 年由 XueJia Lai（来学嘉）和 L.Massey 提出。

1. 算法原理

分组密码 IDEA 明文和密文的分组长度为 64 位，密钥长度为 128 位。该算法的特点是使用了 3 种不同的代数群上的操作。

（1）子密钥生成

IDEA 共使用 52 个 16 位的子密钥，其由输入的 128 位密钥生成，过程如下。

- 首先，输入的 128 位密钥被分成 8 个 16 位的分组，并直接作为前 8 个子密钥。
- 然后，128 位密钥循环左移 25 位，生成的新的 128 位密钥被重新分成 8 个 16 位的分组，作为下面 8 个子密钥。
- 重复上一步，直至 52 个子密钥全部生成。

（2）IDEA 加密算法

IDEA 的加密过程由 8 个相同的加密步骤（称为加密轮函数）和一个输出变换组成。整体结构如图 6.3 所示。

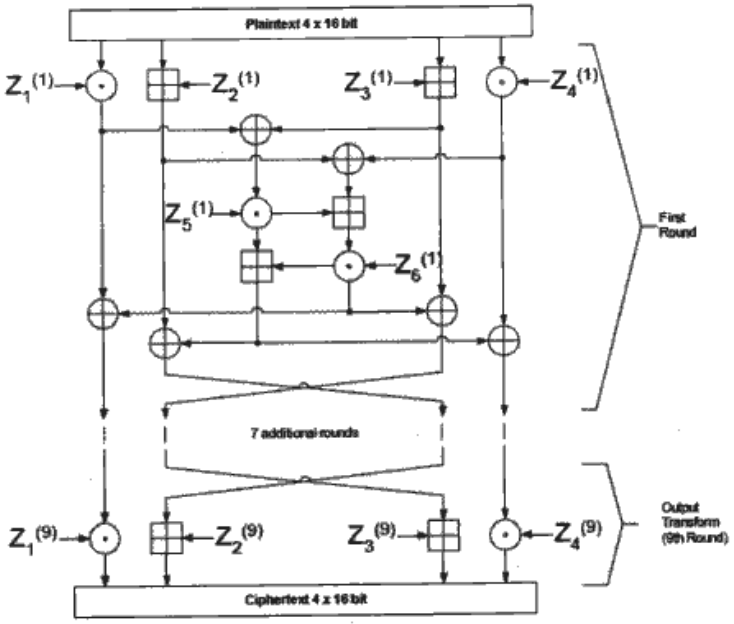


图 6.3 IDEA 加密结构

- ⊕ 表示按位异或操作；
- ⊞ 表示定义在模 2^{16}(=mod 65536)的模加法运算，其操作数都可以表示成 16 位整数；
- ⊙ 表示定义在模 $2^{16}+1$(=mod 65537) 的模乘法运算。

首先，64 位明文被分成 4 个 16 位分组。每一轮加密需要 6 个子密钥，最后的输出变换只需要 4 个子密钥，所以共需要 52 = 8×6+4 个子密钥。如图 6.3 所示，在第一轮加密中，4 个 16 位的子密钥分别通过两个模 $2^{16}+1$ 的乘法运算和两个模 2^{16} 的加法运算与明文进行混合。结果进一步处理，又用到了两个 16 位的子密钥以及按位异或操作。第一轮加密的结果，在进行部分交换后，作为第二轮加密的输入。以此再重复进行 7 轮。在接下来的输出变换（Output Transform）中，使用 52 个子密钥的最后 4 个，通过模加与模乘运算与第 8 轮的结果进行混合，产生最后的密文。

（3）IDEA 解密算法

对密文的解密计算过程同对明文的加密过程是一样的，如图 6.3 所示。解密与加密唯一不同的地方，就是使用不同的子密钥。首先，解密所用的 52 个子密钥是加密的子密钥的相应于不同操作运算的逆元。其次，解密时子密钥必须以相反的顺序使用。

IDEA 中的加法与乘法逆元的规则定义如下：

$x + \text{addinv}(x) \equiv 0 \pmod{65536}$

$x \times \text{mulinv}(x) \equiv 1 \pmod{65537}$

其中，模 $2^{16}+1$ 的乘法逆元的计算可以使用欧几里德扩展算法来求，代码如下：

```
unsigned inv(unsigned xin)
{
    long n1,n2,q,r,b1,b2,t;
    if(xin==0) b2=0;
    else
    { n1=maxim; n2=xin; b2=1; b1=0;
      do { r = (n1 % n2); q = (n1-r)/n2;
           if(r==0) { if(b2<0) b2=maxim+b2; }
           else { n1=n2; n2=r; t=b2; b2=b1-q*b2; b1=t; }
         } while (r!=0);
    }
    return (unsigned)b2;
}
```

2. 实例分析

本书光盘中的 IDEAKeyGenMe 是一个使用 IDEA 算法进行注册验证的 KeyGenMe。先用 PEiD 的 Kcrypto ANANLyzer 查看发现还有 SHA-1 算法。用 OllyDBG 加载，按 F9 键运行，输入假的用户名与序列号，用“bpx GetDlgItemTextA”下断，单击“Check”按钮中断后可来到如下代码处：

```
;前面是验证是否输入用户名，以及序列号是否为 16 个字符
00401796  push   ebp
00401797  lea    edi, [esp+328]
0040179E  lea    esi, [esp+36]
004017A2  mov    ebp, 4
004017A7  /push  esi
004017A8  |push  00408040                ;ASCII "%04X"
004017AD  |push  edi
004017AE  |call  00401900                ;sscanf 函数，将序列号转化成十六进制
004017B3  |add   esp, 0C
004017B6  |add   esi, 2
004017B9  |add   edi, 4
004017BC  |dec   ebp
004017BD  \jnz   short 004017A7
004017BF  lea    edx, [esp+4B8]
004017C6  push   edx
004017C7  call   00401000                ;跟进后，可以看出这是 SHA-1 初始化函数
004017CC  mov    edi, [esp+44]
004017D0  add    esp, 4
004017D3  xor    esi, esi
004017D5  cmp    edi, ebx
004017D7  pop    ebp
004017D8  jle    short 004017F8
004017DA  /movsx eax, byte ptr [esp+esi+3EC]
004017E2  |lea   ecx, [esp+4B4]
004017E9  |push  eax
004017EA  |push  ecx
004017EB  |call  00401040                ;对用户名字符串进行散列
004017F0  |add   esp, 8
004017F3  |inc   esi
004017F4  |cmp   esi, edi
004017F6  \jl    short 004017DA
004017F8  lea    edx, [esp+40]           ;存储散列值的缓冲区，记为 szHash
004017FC  lea    eax, [esp+4B4]          ;SHA-1 Context
00401803  push   edx
00401804  push   eax
```

```
00401805  call   004012A0                ;得到最终的160位散列值，存入szHash
0040180A  add    esp, 8
0040180D  lea    ecx, [esp+C]
00401811  push   ebx                     ;/pFileSystemNameSize
00401812  push   ebx                     ;|pFileSystemNameBuffer
00401813  push   ebx                     ;|pFileSystemFlags
00401814  push   ebx                     ;|pMaxFilenameLength
00401815  push   ecx                     ;|pVolumeSerialNumber
00401816  push   ebx                     ;|MaxVolumeNameSize
00401817  push   ebx                     ;|VolumeNameBuffer
00401818  push   0040803C                ;|RootPathName = "C:\"
0040181D  call   [<GetVolumeInformation> ;\GetVolumeInformationA
00401823  mov    edx, [esp+40]           ;上面代码得到"C:\"的卷序列号,这是散
                                         ;列值的前32位
00401827  mov    eax, [esp+44]
0040182B  mov    ecx, [esp+48]
0040182F  mov    [esp+1E], edx
00401833  mov    edx, [esp+4C]
00401837  mov    [esp+22], eax
0040183B  mov    eax, [esp+50]
0040183F  mov    [esp+2A], edx
00401843  mov    [esp+25C], eax
0040184A  lea    edx, [esp+108]
00401851  mov    [esp+26], ecx
00401855  mov    ecx, [esp+C]
00401859  lea    eax, [esp+1C]
0040185D  push   edx
0040185E  push   eax
0040185F  mov    [esp+268], ecx
-------------------------------------------------------------------
;此处加上前面的mov代码段，是将szHash的前128位转移到另外一个缓冲区，将szHash的低32
;位和卷序列号复制到一缓冲区记为szBuffer
-------------------------------------------------------------------
00401866  call   004014C0
{
    ...
    004014CD  mov    ecx, [eax]          ;将参数1移入参数2缓冲区，共128位
                                         ;(参数1的前16位并不使用)，将参数1记为
    004014CF  mov    edx, [eax+4]        ;wiDeaKey，将参数2记为wSubKey
    ...
    004014EC  mov    ecx, 9              ;循环变量初值
    004014F1  sub    eax, 0C
    004014F4  /lea   edx, [ecx+i]
    004014F7  |mov   esi, edx
    004014F9  |and   esi, 80000007        ;循环变量+1后模8
    004014FF  |jns   short 00401506
    00401501  |dec   esi
    00401502  |or    esi, FFFFFFF8
    00401505  |inc   esi
    00401506  |jnz   short 00401519       ;如果循环变量+1模8不为0，则跳
    00401508  |mov   cx, [eax+E]          ;取当前wSubKey偏移0x0E处的值送入cx
    0040150C  |mov   si, [eax]            ;取当前wSubKey处的值送入si
    0040150F  |shl   cx, 9                ;cx左移9位，即取其低7位
    00401513  |shr   si, 7                ;si右移7位，即取其高9位
    00401517  |jmp   short 00401549
```

```
        00401519  |and    ecx, 80000007        ;循环变量模 8
        0040151F  |jns    short 00401526
        00401521  |dec    ecx
        00401522  |or     ecx, FFFFFFF8
        00401525  |inc    ecx
        00401526  |jnz    short 00401539       ;如果循环变量模 8 不为 0，则跳
        00401528  |mov    cx, [eax-2]          ;取当前 wSubKey 前面的一个 16 位值送入 cx
        0040152C  |mov    si, [eax]            ; 取当前 wSubKey 处的值送入 si
        0040152F  |shl    cx, 9                ;cx 左移 9 位，即取其低 7 位
        00401533  |shr    si, 7                ;si 右移 7 位，即取其高 9 位
        00401537  |jmp    short 00401549
        00401539  |mov    cx, [eax+10]         ;取 wSubKey 相邻的两个 16 位元素，记 cx 为 a
        0040153D  |mov    si, [eax+E]          ;si 为 b
        00401541  |shr    cx, 7                ;cx 右移 7 位，即取 a 的高 9 位
        00401545  |shl    si, 9                ;si 左移 9 位，即取 b 的低 7 位
        00401549  |xor    ecx, esi             ;cx^si，得到一个 16 位数
        0040154B  |mov    [eax+1C], cx         ;将此 16 位数送入当前 wSubKey 所指向的地址处
        0040154F  |mov    ecx, edx
        00401551  |add    eax, 2               ;wSubKey 地址加 2，即移动 16 位
        00401554  |lea    edx, [ecx-1]         ;循环变量减 1
        00401557  |cmp    edx, 36              ;执行循环，直到循环变量增加为 0x36 为止
        0040155A  \jl     short 004014F4
}
```

通过对上面一段循环代码的分析，可以看出，这是将一个共 128 位的字（16 位）数组 wiDeaKey 循环左移 25 位，作为一个新的 128 位数组送入 wSubKey。共生成了 56 个 16 位字。由于 IDEA 算法使用 52 个 16 位的子密钥，且同样需要循环左移 25 位，据此可以大致猜测，此过程为 IDEA 的子密钥生成函数。在 IDEA 中，一共执行了 8 轮加密及 1 轮输出变换，共需要 9 个向量；每一轮使用 6 个密钥，输出变换使用 4 个，共需要 6 个向量。紧接着的一段代码是将此 56 个 16 位字（IDEA 只使用前 52 个）送入一个二维数组 Z[7][10]。Z[0][10]并没有使用，且对于第 i 行（0<i<7），只使用了 9 列。读者可以对照着图 6.3 来理解。

```
0040186B  lea    ecx, [esp+110]       ;上面定义的数组 z，即 IDEA 的子密钥
00401872  lea    edx, [esp+18]        ;存储 IDEA 加密结果的缓冲区
00401876  push   ecx
00401877  lea    eax, [esp+3C]        ;此处即序列号的十六进制表示形式
0040187B  push   edx
0040187C  push   eax
0040187D  call   00401390
{
  ...
  0040139B  xor    ebp, ebp
  0040139D  mov    bp, [eax+2]        ;将序列号十六进制表示的第 1 个字 x1 送入 bp
  004013A1  xor    ebx, ebx
  004013A3  mov    bx, [eax+4]        ;将序列号十六进制表示的第 2 个字 x2 送入 bx
  004013A7  xor    ecx, ecx
  004013A9  mov    cx, [eax+6]        ;将序列号十六进制表示的第 3 个字 x3 送入 cx
  004013AD  xor    edi, edi
  004013AF  mov    di, [eax+8]        ;将序列号十六进制表示的第 4 个字 x4 送入 di
  004013B3  mov    eax, [esp+24]      ;IDEA 子密钥数组
  004013B7  mov    [esp+10], ecx
  004013BB  mov    [esp+14], 8        ;循环变量
  004013C3  lea    esi, [eax+52]
  004013C6  jmp    short 004013CC
  004013C8  /mov   edi, [esp+1C]
  004013CC  |xor   ecx, ecx
```

```
004013CE  |mov   cx, [esi-3C]          ;取用于当前轮的第1个密钥
004013D2  |push  ecx
004013D3  |push  ebp                   ;x1
004013D4  |call  00401340
{
   00401340  mov    ecx, [esp+4]      ;取第1个参数
   00401344  test   ecx, ecx
   00401346  jnz    short 00401359
   00401348  mov    ecx, [esp+8]      ;若参数1为0，则取第2个参数
   0040134C  mov    eax, 10001
   00401351  sub    eax, ecx
   00401353  and    eax, 0FFFF        ;返回值为0x10001-参数2，0x10001即65537
   00401358  retn
   00401359  mov    eax, [esp+8]      ;若参数1不为0，则判断参数2是否为0
   0040135D  test   eax, eax
   0040135F  jnz    short 0040136E
   00401361  mov    eax, 10001
   00401366  sub    eax, ecx
   00401368  and    eax, 0FFFF        ;若参数2为0，则返回值为0x10001-参数1
   0040136D  retn
   0040136E  imul   ecx, eax          ;若参数1、参数2都不为0，则两数相乘，记q=a*b
   00401371  mov    eax, ecx          ;a和b分别为参数1、参数2
   00401373  shr    ecx, 10           ;q右移16位，即除以2^16=65536，也即q/65536
   00401376  and    eax, 0FFFF        ;取q的低16位，即q%65536
   0040137B  sub    eax, ecx          ;记p=(q%65536)-(q/65536)
   0040137D  test   eax, eax
   0040137F  jg     short 00401386
   00401381  add    eax, 10001   ;若p<=0，则p+=65536
   00401386  and    eax, 0FFFF
   0040138B  retn
}
```

从上面这个函数调用的代码分析来看，此函数实现了IDEA中的模$2^{16}+1$乘法运算，记为mul

```
004013D9  |xor   edx, edx
004013DB  |mov   ebp, eax            ;x1=mul(x1,Z[1][r])
004013DD  |mov   dx, [esi]           ;取用于当前轮的第4个子密钥
004013E0  |push  edx
004013E1  |push  edi
004013E2  |call  00401340            ;x4=mul(x4,Z[4][r])
004013E7  |mov   di, [esi-28]        ;取用于当前轮的第2个子密钥
004013EB  |mov   ecx, [esp+20]       ;x3
004013EF  |add   edi, ebx            ;x2+Z[2][r]
004013F1  |mov   bx, [esi-14]        ;取用于当前轮的第3个子密钥
004013F5  |add   ebx, ecx            ;x3+Z[3][r]
004013F7  |mov   [esp+2C], eax
004013FB  |and   ebx, 0FFFF          ;取低16位，即模2^16(65536)加法运算
00401401  |xor   ecx, ecx
00401403  |mov   cx, [esi+14]        ;取用于当前轮的第5个子密钥
00401407  |mov   eax, ebx
00401409  |xor   eax, ebp            ;x3同x1进行异或
0040140B  |and   edi, 0FFFF
00401411  |push  eax
00401412  |push  ecx
00401413  |call  00401340            ;mul(Z[5][r],(x1^x3)
```

```
00401418  |mov   edx, [esp+34]
0040141C  |mov   [esp+28], eax
00401420  |xor   edx, edi
00401422  |add   edx, eax
00401424  |xor   eax, eax
00401426  |mov   ax, [esi+28]    ;取用于当前轮的第 6 个子密钥
0040142A  |and   edx, 0FFFF
00401430  |push  edx
00401431  |push  eax
00401432  |call  00401340
00401437  |mov   ecx, [esp+30]   ;下面执行异或及模加运算，以及部分交换
0040143B  |mov   edx, [esp+3C]   ;以完成该轮的加密函数
0040143F  |add   ecx, eax
00401441  |xor   ebp, eax
00401443  |and   ecx, 0FFFF
00401449  |xor   eax, ebx
0040144B  |xor   edx, ecx
0040144D  |mov   ebx, eax
0040144F  |mov   eax, [esp+34]
00401453  |xor   ecx, edi
00401455  |add   esp, 20
00401458  |add   esi, 2
0040145B  |mov   edi, ecx
0040145D  |dec   eax             ;共执行 8 轮，执行完后将进行输出变换
0040145E  |mov   [esp+1C], edx
00401462  |mov   [esp+10], edi
00401466  |mov   [esp+14], eax
0040146A  \jnz   004013C8        ;下面的代码为 IDEA 的输出变换
00401470  mov    esi, [esp+24]
00401474  xor    edx, edx
00401476  mov    dx, [esi+26]    ;取用于输出变换的第 1 个子密钥
0040147A  push   edx
0040147B  push   ebp
0040147C  call   00401340        ;mul(x1,Z[1][9])
00401481  mov    ebp, [esp+28]
00401485  mov    ecx, [esp+24]
00401489  mov    [ebp+2], ax     ;将加密结果输出到缓冲区
0040148D  xor    eax, eax
0040148F  mov    ax, [esi+62]    ;取用于输出变换的第 4 个子密钥
00401493  push   eax
00401494  push   ecx
00401495  call   00401340        ;mul(x4,Z[4][9])
0040149A  mov    [ebp+8], ax
0040149E  mov    dx, [esi+3A]    ;取用于输出变换的第 2 个子密钥
004014A2  add    dx, di
004014A5  add    esp, 10
004014A8  mov    [ebp+4], dx
004014AC  mov    ax, [esi+4E]    ;取用于输出变换的第 3 个子密钥
004014B0  add    ax, bx
004014B3  pop    edi
004014B4  mov    [ebp+6], ax
004014B8  pop    esi
004014B9  pop    ebp
004014BA  pop    ebx
004014BB  add    esp, 8
```

```
        004014BE  retn
    }
    00401882    8B4C24 26       mov     ecx, [esp+26]
```

分析这段代码，结合对 IDEA 加密算法的原理，可以看出此处就是 IDEA 的加密函数，即对序列号（十六进制形式）进行加密。在上面提到，szHash 的低 32 位及“C:\”盘卷序列号被存储到了一缓冲区 szBuffer。在进行完 IDEA 加密后，将加密的结果，共 64 位，同 szBuffer 进行比较，若相同，则序列号正确，否则注册失败。

其逆过程即为使用用户名 160 位散列的前 128 位作为 IDEA 的密钥，对散列的 32 位和卷序列号进行 IDEA 的解密运算，再转化成其 ASCII 码形式，即为最终的序列号。注意 IDEA 解密密钥的生成过程，详细源代码见本书光盘。

6.2.4 BlowFish 算法

BlowFish 算法是一个 64 位分组及可变密钥长度的分组密码算法，该算法是非专利的。

1. 算法原理

BlowFish 算法基于 Feistel 网络（替换/置换网络的典型代表），加密函数迭代执行 16 轮。分组长度为 64 位（bit），密钥长度可以从 32 位到 448 位。算法由两个部分组成：密钥扩展部分和数据加密部分。密钥扩展部分将最长为 448 位的密钥转化成共 4168 字节长度的子密钥数组。其中，数据加密由一个 16 轮的 Feistel 网络来完成。每一轮由一个密钥相关置换和一个密钥与数据相关的替换组成。

（1）子密钥

BlowFish 使用大量的子密钥。这些密钥必须在进行加密前预计算产生。

- P 数组由 18 个 32 位字的子密钥组成：P1,P2,…,P18。
- 4 个 8×32 的包含总共 1024 个 32 位字的 S-box：

$$S_{1,0}, S_{1,1}, \cdots, S_{1,255}$$
$$S_{2,0}, S_{2,1}, \cdots, S_{2,255}$$
$$S_{3,0}, S_{3,1}, \cdots, S_{3,255}$$
$$S_{4,0}, S_{4,1}, \cdots, S_{4,255}$$

子密钥扩展算法如下：

① 按顺序使用常数 π 的小数部分初始化 P 数组和 S-box。

比如：

P1=0x243f6a88

P2=0x85a308d3

P3=0x13198a2e

P4=0x03707344

② 对 P 数组和密钥进行逐位异或，需要时重用密钥。

③ 使用当前的 P 数组和 S-box 对全 0 的 64 位分组使用 BlowFish 算法进行加密，用输出替代 P1、P2。

④ 使用当前的 P 和 S 对第③步的输出进行加密，并用输出替代 P3、P4。

⑤ 继续上面的过程，直到按顺序替代所有的 P 数组和 S-box 中的元素。

（2）加密

BlowFish 是由 16 轮的 Feistel 网络组成的。输入是一个 64 位的数据元素 x，将 x 分成两个 32 位部分：xL,xR。加密算法的伪 C 代码如下：

```
for(i=1;i<=16;i++)
{
```

```
    xR[i]=xL[i-1] ^ P[i];
    xL[i]=F(xR[i])^xR[i-1];
  }
  xL[17]=xR[16]^ P[18];
  xR[17]=xL[16]^ p[17];
```

其中“^”表示异或运算，函数 F 的输入是一个 32 位双字，共 4 个字节，分别作为 4 个 S-box 的索引，取出相应的 S-box 值，然后进行模 2^{32} 加运算。用等式可以描述如下：

$F(a,b,c,d)=((S_{1,a}+S_{2,b})\wedge S_{3,c})+S_{4,d}$

解密与加密完全相同，只不过 P1,P2,…,P18 以相反的顺序使用。

2．实例分析

先用 PEiD 的 Krypto ANALyzer 插件查看 BlowFishKGM.exe，可以识别出 BlowFish 的 P 数组和 S-box，猜测该 KeyGenMe 使用了 BlowFish 算法。

用 OllyDbg 打开本书光盘中的 BlowFishKGM.exe 查找参考字符串，可以找到“Success!”与“Wrong Serial!”，双击“Success!”，可以来到反汇编窗口。向上可以找到注册码判断的地方，从 00401123 处开始，在此下断点。按 F9 键运行，输入 Serial Number：

123456789ABCDEFFEDCBA987654321

可以断在 00401123 处。代码如下：

```
00401123  cmp    eax, 20                      ;注册码必须为 32 个字节
00401126  jnz    00401202
0040112C  push   esi
0040112D  xor    esi, esi
0040112F  lea    edi, [esp+1AC]
00401136  /lea   ecx, [esp+esi+1C]
0040113A  |push  ecx                          ;缓冲区 1，用来保存注册码的十六进制形式
0040113B  |push  00409050                     ;ASCII "%02X"
00401140  |push  edi
00401141  |call  00401450                     ;sscanf 函数，将注册码转化成十六进制形式
00401146  |add   esp, 0C
00401149  |inc   esi
0040114A  |add   edi, 2
0040114D  |cmp   esi, 10
00401150  \jl    short 00401136
00401152  mov    edx, [esp+1C]                ;缓冲区 1 的前 4 个字节
00401156  mov    eax, [esp+20]                ;缓冲区 1 的第 5 到 8 共 4 个字节
0040115A  mov    [esp+E4], edx
00401161  lea    ecx, [esp+E4]
00401168  push   8
0040116A  lea    edx, [esp+278]
00401171  push   ecx
00401172  push   edx
00401173  mov    [esp+F4], eax
0040117A  call   00401350
{
    ......
    00401359  mov     eax, 00407120;跟到这里，可以查看 00407120 内存地址处的数据
    ------------------------------------------------------------------
    00407120  A6 0B 31 D1 AC B5 DF 98 DB 72 FD 2F B7 DF 1A D0  ?1 熏颠楼 r?椟
    00407130  ED AF E1 B8 96 7E 26 6A 45 90 7C BA 99 7F 2C F1  瀍岣柂&jE 悠箸•,
    00407140  47 99 A1 24 F7 6C 91 B3 E2 F2 01 08 16 FC 8E 85  G樗$鰭懈怚   膥
    容易看出来，这是 BlowFish 算法的 S-box，即 π 的小数部分
```

```
0040135E  lea     ecx, [esi+48]
00401361  mov     edx, 100
00401366  mov     edi, [eax]
00401368  add     eax, 4
0040136B  mov     [ecx], edi
0040136D  add     ecx, 4
00401370  dec     edx
00401371  jnz     short 00401366
00401373  cmp     eax, 00408120
00401378  jl      short 00401361
```

很明显，上面的代码是用π的小数部分初始化 BlowFish 的 S-box，即将用于加密

```
0040137A  mov     ebp, [esp+20]        ;密钥长度
0040137E  mov     edx, [esp+1C]        ;BlowFish 密钥
00401382  mov     edi, 004070D8
```

查看内存数据可知 004070D8 处是 BlowFish 的 P 数组，而下面的代码是用此 P 数组与密钥进行循环异或，作为密钥预处理的一部分

```
004013D3  xor     eax, eax
004013D5  mov     [esp+20], eax
004013D9  mov     [esp+1C], eax        ;同上一句，初始化两个全 0 变量
004013DD  mov     esi, ebx
004013DF  mov     edi, 9               ;P 数组共 18 个元素，这初始化为 9
                                       ;每次替换两个元素
004013E4  /lea    eax, [esp+1C]
004013E8  |lea    ecx, [esp+20]
004013EC  |push   eax
004013ED  |push   ecx
004013EE  |push   ebx
004013EF  |call   00401220
```

这个函数以两个全 0 变量作为三个参数中之二，根据上面的 P 数组与密钥异或及 S-box 的初始化代码可以加以推测上面的这个函数(00401220)为 BlowFish 的加密函数，用来对两个全 0 的变量进行加密，以完成子密钥的生成过程。进一步跟踪 00401220 处的代码

```
{
    0040123E  xor   eax, [ebx]         ;将待加密的变量与初始化后的 P 数组进行异或
    00401240  push  eax
    00401241  push  edi
    00401242  mov   ebp, eax
    00401244  call  00401280
    {
        ......
        004012BF  mov     eax, [edi+eax*4+48]
        004012C3  mov     ebx, [edi+ecx*4+448]
        004012CA  mov     ecx, [edi+esi*4+848]
        004012D1  add     eax, ebx
        004012D3  xor     eax, ecx
        004012D5  mov     ecx, [edi+edx*4+C48]
```

上面一部分代码是取传入的参数的 4 个字节作为索引，分别从刚刚初始化完的 S-box 中取值，结合上面提到的一系列代码，可以断定此处是 BlowFish 算法中的 F 函数

```
    }
```

```
00401249  mov     ecx, [esp+1C]
0040124D  add     esp, 8
00401250  xor     eax, esi          ;将返回值与另外一个待加密和变量进行异或
00401252  add     ebx, 4
00401255  dec     ecx
00401256  mov     esi, ebp
00401258  mov     [esp+14], ecx
0040125C  jnz     short 0040123E
}
004013F4  |mov     edx, [esp+2C]     ;替换 P 数组
004013F8  |mov     eax, [esp+28]
004013FC  |mov     [esi], edx
004013FE  |mov     [esi+4], eax
00401401  |add     esp, 0C
00401404  |add     esi, 8
00401407  |dec     edi
00401408  \jnz     short 004013E4
--------------------------------------------------------
再往下的一段代码是利用 BlowFish 的加密函数来初始化 S-box，最终完成子密钥的生成
现在基本上可以断定是 BlowFish 算法无疑了
--------------------------------------------------------
}
0040117F  mov     eax, [esp+30]
00401183  mov     ecx, [esp+34]
00401187  mov     [esp+1C], eax
0040118B  lea     edx, [esp+20]
0040118F  mov     [esp+20], ecx
00401193  lea     eax, [esp+1C]
00401197  push    edx
00401198  lea     ecx, [esp+284]
0040119F  push    eax
004011A0  push    ecx
004011A1  call    004012F0
--------------------------------------------------------
对于 004012F0 处的函数，根据上面对 BlowFish 加密函数代码及 F 函数代码的分析，加上 P 数组的使用
顺序是反向的，可以断定是 BlowFish 的解密函数。也就是说，上面这段代码是对缓冲区 1 的后 8 个字节进
行解密运算
--------------------------------------------------------
004011A6  mov     esi, [esp+2C]
004011AA  mov     ecx, [esp+28]
004011AE  add     esp, 18
004011B1  lea     edx, [esp+18]
004011B5  xor     esi, ecx          ;将解密得到的两个双字进行异或操作
```

紧接着，KeyGenMe 得到 C 盘的硬盘序列号，与上面得到的异或的结果进行比较，如果相等，则表示注册码正确，否则注册失败。

整个序列号验证算法清楚了。即用输入的注册码的前 16 个字节的十六进制形式作为 BlowFish 算法的密钥，对其后的 16 个字节的十六进制码进行解密，将解密得到的两个双字进行异或，其结果需和硬盘序列号相等。其逆算法为，可随机生成 12 个字节，其前 8 个字节作为 BlowFish 的密钥，剩下的 4 个字节作为一个双字与硬盘序列号进行异或得到另外一个双字，然后对这两个双字（共 64 位）进行 BlowFish 加密运算，得到的 64 位密文即为注册码的后 8 个字节（十六进制形式），然后调用 sprintf 函数将 8 个字节的密钥及加密后的 8 个字节的密文进行输出，即是最后的注册码了。更加详细的过程请参考光盘中提供的注册码源代码。

6.2.5 AES 算法

AES 即 Advanced Encryption Standard，高级加密标准。是 NIST（National Institute of Standards Technology）于 1997 年开始向世界范围内征集的加密算法，用于替代 DES 成为新一代的加密标准。1997 年 9 月 NIST 发布了 AES 需要符合的标准，其中要求 AES 具有 128 比特的分组长度，并支持 128、192 和 256 比特的密钥长度，而且要求 AES 能在全世界范围内免费得到。AES 的评选工作一共进行了 3 轮。第一次共有 15 个算法入选，分别为 CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, MARS, RC6, RIJNDAEL, SAFER+, SERPENT, TWOFISH。在第二次的公开评选后，NIST 宣布共有 5 个算法进入最后决赛，分别是 MARS, RC6, Rijndael, Serpent, Twofish。最终于 2000 年 10 月，NIST 宣布 Rijndael 由于在各方面的表现都十分优秀，当选为 AES。

Rijndael 由两位国际著名的比利时密码学家 Joan Daemen 和 Vincent Rijmen 设计，读作“Rain Doll”。实际上，Rijndael 算法本身和 AES 的唯一区别在于各自所支持的分组长度和密码密钥长度的范围不同。Rijndael 是具有可变分组长度和可变密钥长度的分组密码，其分组长度和密钥长度均可独立地设定为 32 比特的任意倍数，最小值为 128 比特，最大值为 256 比特。而 AES 将分组长度固定为 128 位，而且仅支持 128、192 和 256 位的密钥长度，分别称作 AES-128、AES-192、AES-256。在本书中提到的 AES，如无特别说明，专指 FIPS-197 中规定的 AES 算法。

1．基本术语

（1）字节

AES 算法的基本处理单元叫做字节，它由 8 比特序列组成，被看作为一个整体。在 AES 中，这些字节以有限域（Finite Field）上的多项式（polynomial）来表示：

$$b_7x^7+b_6x^6+b_5x^5+b_4x^4+b_3x^3+b_2x^2+b_1x+b_0=\sum_{i=0}^{7}b_ix^i \qquad (6.1)$$

其中，b_i（$0\leqslant i\leqslant 7$）分别代表了一个字节的 8 个比特位。如字节{01100011}，即 0x63，代表了相应的有限域元素 x^6+x^5+x+1。

（2）状态（State）

AES 的所有操作都是在一个称作状态（State）的二维字节数组上进行的。状态由 4 行字节组成，每行包括 *Nb* 个字节，*Nb* 为分组长度除以 32 的值。用 *s* 来表示状态，状态数组中的每个字节有两个坐标，行号 *r* 的范围为 $0\leqslant r<4$，列号 *c* 的范围为 $0\leqslant c<Nb$。这样就可以用 $S_{r,c}$ 或者 $s[r,c]$来引用状态中的每个字节。在 AES 算法的加密和解密过程的开始，输入字节数组 in_0，in_1，…，in_{15} 复制到状态数组中，然后对状态数组中的元素进行加解密操作，最后将结果复制到输出字节数组 $out_0, out_1, \cdots, out_{15}$，如图 6.4 所示：

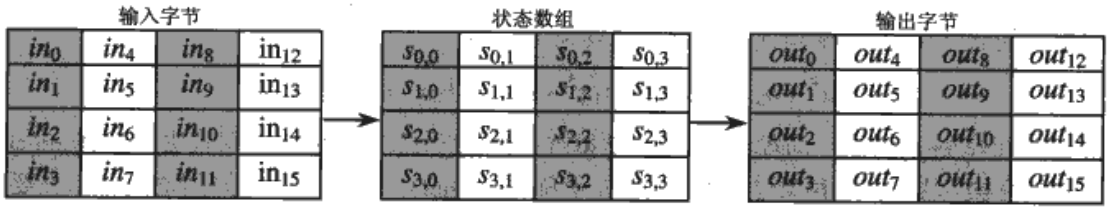


图 6.4 状态数组输入、输出

在实际的软件实现中，将每一列的 4 个字节作为一个 32 位字。如一个长度为 128 位的分组数据块在内存中的形式如下：

```
0012F770  F6 14 46 C1 A6 8C EA 53 82 48 26 A7 A4 7F 19 14  ?F力蚯S倡&Γ•
```

那么此状态数组为：

F6	A6	82	A4
14	8C	48	7F
46	EA	26	19
C1	53	A7	14

2. 数学背景

（1）加法

有限域上的两个元素的加法运算是通过对相应的多项式中的相同次幂的系数进行相加来实现的。其加法是模 2 加运算，也就是异或操作，用符号“⊕”来表示。相应的减法运算和加法是完全相同的。

例如，下面的表达式是相同的：

$(x^6+x^4+x^2+x+1)+(x^7+x+1)=x^7+x^6+x^4+x^2$　　多项式表示

{01010111} ⊕ {10000011}={11010100}　　二进制表示

{53} ⊕ {83}={d4}　　十六进制表示

（2）乘法

如果一个多项式的因子为 1 和它本身，那么称这个多项式是不可约的，此多项式为不可约多项式。以多项式形式表示的 GF(2^8)（注：GF 表示 Galois Field，伽罗瓦域，Galois 是第一位研究有限域的数学家）上的乘法运算（以·表示），对应于多项式相乘然后再模一个 8 次的不可约多项式（关于有限域更加详细的论述，请参考《Introduction to Finite Fields and Their Applications》一书）。对于 AES 算法，这个不可约多项式为：

$$m(x)=x^8+x^4+x^3+x+1 \tag{6-2}$$

或者以十六进制表示为{01}{1b}，即 0x11B。

例如，{57}· {83}={c1}，因为

$$\begin{aligned}(x^6+x^4+x^2+x+1)(x^7+x+1)&=x^{13}+x^{11}+x^9+x^8+x^7+x^7+\\&\quad x^5+x^3+x^2+x+x^6+x^4+x^2+x+1\\&=x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1\end{aligned}$$

且 $x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1 \mod x^8+x^4+x^3+x+1=x^7+x^6+1$

通过 $m(x)$模约简运算，可以保证结果为一个次数小于 8 的二进制多项式，并且可以用字节来表示。同时，AES 还涉及了有限域上求乘法逆元的运算，读者可以参考相关资料。在后面的章节中，还会涉及模 p 的乘法逆元计算问题，将在后面讲述。

（3）与 x 相乘

将上面定义的式（6-1）与多项式 x 相乘，其结果为：

$$b_7x^8+b_6x^7+b_5x^6+b_4x^5+b_3x^4+b_2x^3+b_1x^2+b_0x$$

然后将其进行模 $m(x)$约简，即如果 b_7=0，结果就是最简形式。若 b_7=1，则减去多项式 $m(x)$（减法运算即上面讲到的异或运算）。这种乘法运算在 AES 中称作 xtime()。如果乘以 x 的高阶次幂，只要重复进行 xtime()运算，并将每个中间结果进行相加即可。

例如：{57}· {13}={fe}，因为：

{57} · {02}=xtime({57})={ae}

{57} · {04}=xtime({ae})={47}

{57} · {08}=xtime({47})={8e}

{57} · {10}=xtime({8e})={07}

所以

$$\{57\} \cdot \{13\} = \{57\} \cdot (\{01\} \oplus \{02\} \oplus \{10\})$$
$$= \{57\} \oplus \{ae\} \oplus \{07\}$$
$$= \{fe\}$$

3. 算法描述

对于 AES 算法来讲，输入分组、输出分组及状态数组的长度都是 128 比特，即 *Nb*=4。密钥 K 的长度为 128、192 或者 256 比特，用 *Nk*=4、6 或者 8 来表示。加密或者解密函数所执行的轮数取决于密钥的长度。轮数用 *Nr* 来表示，则当 *Nk*=4 时，*Nr*=10；*Nk*=6 时，*Nr*=12；*Nk*=10 时，*Nr*=14，如表 6-1 所示。

表 6-1 AES 算法描述

	密钥长度 (*Nk* 个 32 位双字)	分组长度 (*Nb* 个 32 位双字)	轮数 (*Nr*)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

（1）加密过程

先将输入复制到状态数组。在进行一个初始轮密钥加（Round Key Addition）操作之后，执行 *Nr* 次轮函数（Round Function）对状态数组进行变换，其中最后一轮不同于前 *Nr*-1 轮。最终的状态数组复制到输出即为最后的密文。

轮函数由 4 部分组成，分别是 SubBytes()、ShiftRows()、MixColumns()和 AddRoundKey()。其加密过程用伪代码表示如下：

```
Cipher(byte in[4*Nb], byte out[4*Nb], dword dw[Nb*(Nr+1)])
begin
  byte state[4,Nb]  // 状态数组
  state=in
  AddRoundKey(state, dw[0,Nb-1])
  for round=1 step 1 to Nr-1
    SubBytes(state)
    ShiftRows(state)
    MixColumns(state)
    AddRoundKey(state, dw[round*Nb, (round+1)*Nb-1])
  end for
  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state, dw[Nr*Nb, (Nr+1)*Nb-1])
  out=state
end
```

（2）SubBytes()

SubBytes，字节代换，实际上就是一个简单的查表操作。AES 定义了一个 16×16 个字节的 S 盒（S-box），如表 6-2 所示。以状态数组中的每个字节元素的高 4 位作为行标，低 4 位作为列标，取出相应的元素作为 SubBytes 操作的结果。比如十六进制值{C5}，高 4 位为 C，低 4 位为 5，取 S 盒中的行标为 C 列标为 4 的元素为十六进制{A6}，则{C5}将被替换为{A6}。

按照此种操作，将状态数组中的所有元素都替换为 S 盒中的值。关于 S-box 的详细设计原理，请参考相关资料。

表 6-2　S-box(十六进制)

x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

（3）ShiftRows

ShiftRows 操作规则是状态数组的第一行保持不变，第二行循环左移一个字节，第三行循环左移两个字节，第四行循环左移三个字节。比如状态：

F6	A6	82	A4
14	8C	48	7F
46	EA	26	19
C1	53	A7	14

经过 ShiftRows 操作后的结果为：

F6	A6	82	A4
8C	48	7F	14
26	19	46	EA
14	C1	53	A7

（4）MixColumns

MixColumns 操作是以列为单位的，把状态中的每一列看作一个系数在 GF(2^8)上的四项多项式，然后乘以一个固定的多项式 $a(x)$，再模(x^4+1)。$a(x)$的定义如下：

$$a(x)=\{03\}x^3+\{01\}x^2+\{01\}x+\{02\}$$

记状态 S 经过 MixColumns 操作后为 S'，则 MixColumns 可以看作是一个矩阵乘法。

$$\begin{pmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{pmatrix}=\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}\begin{pmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{pmatrix}$$

其中$0 \le c < Nb$，上面的矩阵乘法即

$$S'_{0,c} = (\{02\} \bullet S_{0,c}) \otimes (\{03\} \bullet S_{1,c}) \otimes S_{2,c} \otimes S_{3,c}$$
$$S'_{1,c} = S_{0,c} (\{02\} \bullet S_{1,c}) \otimes (\{03\} \bullet S_{2,c}) \otimes S_{3,c}$$
$$S'_{2,c} = S_{0,c} \otimes S_{1,c} \otimes (\{02\} \bullet S_{2,c}) \otimes (\{03\} \bullet S_{3,c})$$
$$S'_{2,c} = (\{03\} \bullet S_{0,c}) \otimes S_{1,c} \otimes S_{2,c} \otimes (\{02\} \bullet S_{3,c})$$

（5）AddRoundKey()

AddRoundKey 操作是将状态中的元素同轮密钥通过简单的异或运算相加。轮密钥是由用户输入的密钥通过密钥扩展过程而生成的，同样可以看作一个状态数组。

（6）密钥扩展（Key Expansion）

密钥扩展算法通过对用户输入的 128、192 或者 256 位的密钥进行处理，共生成 *Nb*(*Nr*+1)个 32 位双字，为加解密算法的轮函数提供轮密钥。密钥扩展算法伪代码如下：

```
KeyExpansion(byte key[4*Nk], dword dw[Nb*(Nk+1)], Nk)
begin
    dword temp
    i=0
    while(i<Nk)
        dw[i]=dword(key[4*i, key[4*i+1], key[4*i+2],key[4*i+3] )
        i=i+1
    end while
    i=Nk
    while(i<Nb*(Nk+1))
        temp=dw[i-1]
        if ( i mod Nk =0 )
          temp=SubWord(RotWord(temp)) xor Rcon[i/Nk]
        else if ( Nk>6 and i mod Nk =4 )
          temp=SubWord(temp)
        end if
        dw[i] = dw[i-Nk] xor temp
        i = i + 1
    end while
end
```

其中，SubWord 函数以一个 4 个字节的双字作为输入，然后对每个字节利用 S 盒进行替换作为输出。函数 RotWord 以双字$[a_0,a_1,a_2,a_3]$作为输入，做循环左移操作，输出为$[a_1,a_2,a_3,a_0]$。轮常量数组 Rcon 中的每一个元素 Rcon[i]为一个 32 位双字，且低 24 位恒为 0。高 8 位，即一个字节按如下规则定义：Rcon[1]=1，Rcon[i]=2*Rcon[i-1]，乘法定义于 GF(2^8)上。以 10 轮为例，Rcon 的值为（十六进制）：

i	1	2	3	4	5	6	7	8	9	10
Rcon[i]	01	02	04	08	10	20	40	80	1B	36

（7）解密过程

加密算法的逆过程即为解密算法。因此解密算法的轮函数由 4 个部分组成，分别为 InvShiftRows()、InvSubBytes()、InvMixColumns()和 AddRoundKey()。

InvShiftRows 是 ShiftRows 的逆过程，即状态中的后三行执行相应的右移操作，如第二行循环右移一个字节。

InvSubBytes 是 SubBytes 的逆过程。AES 同时也定义了一个逆 S 盒（Inverse S-box）。同 SubBytes 一样，InvSubBytes 也只是简单的查表操作。

InvMixColumns 与 MixColumns 的原理是相同的，不同的是使用了不同的多项式。

$$a^{-1}(x)=\{0b\}x^3+\{0d\}x^2+\{09\}x+\{0e\}$$

InvMixColumns 使用的系数矩阵为 $\begin{pmatrix}0e & 0b & 0d & 09\\ 09 & 0e & 0b & 0d\\ 0d & 09 & 0e & 0b\\ 0b & 0d & 09 & 0e\end{pmatrix}$，它同 MixColumns 中使用的矩阵互为逆矩阵。

AddRoundKey 的逆过程就是它本身，因为异或操作是其本身的逆。

4. 在 32 位处理器上的实现

在加解密及逆向工程中，常见的 AES 算法的实现与上面所讲述的过程是不同的。实际的软件实现采用了以空间换时间的方法，将轮函数的几个步骤合并为一组简单的查表操作。

假设轮函数的输入用 a 表示，SubBytes 的输出用 b 表示，则

$$b_{i,j}=S[a_{i,j}],\quad 0\leqslant i<4,0\leqslant j<Nb \tag{6-3}$$

又设 ShiftRows 的输出用 c 表示，MixColumns 的输出用 d 表示：

$$\begin{pmatrix}c_{0,j}\\ c_{1,j}\\ c_{2,j}\\ c_{3,j}\end{pmatrix}=\begin{pmatrix}b_{0,j+0}\\ b_{1,j+1}\\ b_{2,j+2}\\ b_{3,j+3}\end{pmatrix},0\leqslant j<Nb \tag{6-4}$$

$$\begin{pmatrix}d_{0,j}\\ d_{1,j}\\ d_{2,j}\\ d_{3,j}\end{pmatrix}=\begin{pmatrix}02 & 03 & 01 & 01\\ 01 & 02 & 03 & 01\\ 01 & 01 & 02 & 03\\ 03 & 01 & 01 & 02\end{pmatrix}\bullet\begin{pmatrix}c_{0,j}\\ c_{1,j}\\ c_{2,j}\\ c_{3,j}\end{pmatrix},0\leqslant j<Nb \tag{6-5}$$

式（6-4）中下标的加法必须是模 Nb 的。式（6-3）～式（6-5）可以合并为：

$$\begin{pmatrix}d_{0,j}\\ d_{1,j}\\ d_{2,j}\\ d_{3,j}\end{pmatrix}=\begin{pmatrix}02 & 03 & 01 & 01\\ 01 & 02 & 03 & 01\\ 01 & 01 & 02 & 03\\ 03 & 01 & 01 & 02\end{pmatrix}\bullet\begin{pmatrix}S[a_{0,j+0}]\\ S[a_{0,j+1}]\\ S[a_{0,j+2}]\\ S[a_{0,j+3}]\end{pmatrix}=\left(\begin{pmatrix}02\\ 01\\ 01\\ 03\end{pmatrix}S[a_{0,j+0}]\oplus\right.$$

$$\left.\begin{pmatrix}03\\ 02\\ 01\\ 01\end{pmatrix}S[a_{1,j+1}]\oplus\begin{pmatrix}01\\ 03\\ 02\\ 03\end{pmatrix}S[a_{2,j+2}]\oplus\begin{pmatrix}01\\ 01\\ 03\\ 02\end{pmatrix}S[a_{3,j+3}]\right),0\leqslant j<Nb \tag{6-6}$$

定义 4 个表：T_0，T_1，T_2，T_3

$$T_0[a]=\begin{pmatrix}02\cdot S[a]\\ 01\cdot S[a]\\ 01\cdot S[a]\\ 03\cdot S[a]\end{pmatrix},T_1[a]=\begin{pmatrix}03\cdot S[a]\\ 02\cdot S[a]\\ 01\cdot S[a]\\ 01\cdot S[a]\end{pmatrix},T_2[a]=\begin{pmatrix}01\cdot S[a]\\ 03\cdot S[a]\\ 02\cdot S[a]\\ 01\cdot S[a]\end{pmatrix},T_3[a]=\begin{pmatrix}01\cdot S[a]\\ 01\cdot S[a]\\ 03\cdot S[a]\\ 02\cdot S[a]\end{pmatrix}$$

每个 T 表都有 256 个 4 字节的 32 位双字，从而需要 4KB 的存储空间。使用这些表，可以将式（6-6）改写成：

$$\begin{pmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{pmatrix} = T_0[a_{0,j+0}] \oplus T_1[a_{1,j+1}] \oplus T_2[a_{2,j+2}] \oplus T_3[a_{3,j+3}], 0 \leqslant j < Nb$$

同时，AddRoundKey 可以通过在每一列上执行一个额外的 32 位异或运算来实现，所以使用该 4KB 的表，对每一轮的每一列只需要 4 次查表和 4 次异或运算。而且这 4 个表只需要一个即可，其他三个可以通过循环移位得到。由于最后一轮没有 MixColumns 操作，所以最后一轮仍然需要使用常规的方法。

下面以通过查上面所讲到的 4KB 的表实现 AES，来讲解 AES 在软件保护中的应用。

5. 实例分析

先用 PEiD 的 Krypto ANALyzer 查看光盘中的 AESKeyGenMe.exe，结果如图 6.5 所示。

图 6.5 显示 PEiD 分析结果

这表示该 KeyGenMe 中有用于 MD5 算法的 64 个常量元素的 T 表及 AES 的 S 盒和逆 S 盒，可以猜测使用了 MD5 和 AES 两种算法。下面具体来分析该 KeyGenMe。首先用 OllyDbg 加载，按 F9 键运行，输入假的用户名和序列号，用“bpx GetDlgItemTextA”下断，可以来到如下代码处：

```
;前面的一段代码除了检测是否输入用户名及序列号长度是否为 32 个字节之外，还检查
;了序列号是否都是十六进制字符
004011F4  /lea    edx, [esp+esi+144]   ;缓冲区，记为 szBuffer
004011FB  |push   edx
004011FC  |push   0040B08C             ;ASCII "%02X"
00401201  |push   edi
00401202  |call   004029A0             ;sscanf 函数，将序列号转化为十六进制形式
00401207  |add    esp, 0C              ;存入缓冲区 szBuffer
0040120A  |inc    esi
0040120B  |add    edi, 2
0040120E  |cmp    esi, 10
00401211  \jl     short 004011F4
00401213  mov     ecx, 16
00401218  xor     eax, eax
0040121A  lea     edi, [esp+24]
0040121E  rep     stos dword ptr es:[edi]
00401220  lea     eax, [esp+24]
00401224  push    eax
00401225  call    004012F0             ;MD5Init
0040122A  mov     ecx, [esp+24]
0040122E  lea     edx, [esp+210]
00401235  push    ecx
00401236  lea     eax, [esp+2C]
0040123A  push    edx
0040123B  push    eax
0040123C  call    00401320             ;MD5Update
00401241  lea     ecx, [esp+34]
00401245  lea     edx, [esp+2E4]       ;szHash，用来保存 128 位的 MD5 散列
```

```
0040124C  push    ecx
0040124D  push    edx
0040124E  call    004013D0              ;MD5Final
00401253  mov     ecx, 7F
00401258  xor     eax, eax
```

上面的这段代码是对用户名使用 MD5 算法进行散列，根据前面对 MD5 的讲解，很容易判断出来。此处将产生 128 位的散列。

```
0040125A  lea     edi, [esp+3B4]
00401261  push    ebx                   ;NULL
00401262  rep     stos dword ptr es:[edi]
00401264  lea     eax, [esp+2C]
00401268  lea     ecx, [esp+3B8]
0040126F  push    eax                   ;AES 密钥
00401270  push    10                    ;密钥长度
00401272  push    ebx                   ;0
00401273  push    ecx                   ;aes_struct
00401274  call    00401EC0              ;aes_init 函数
00401279  lea     edx, [esp+170]
00401280  lea     eax, [esp+3C8]
00401287  push    edx
00401288  push    eax
00401289  call    004023A0
```

00401264 处的“lea eax,[esp+2C]”，[esp+2C]指向地址 0012F4F4，其数据如下：

```
0012F4F4  2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C  +~  (  A??蠟<
```

共 128 位，这是 AES 的密钥。可以看出这是 AES-128。aes_init 函数共有 5 个参数：aes_init(aes* a,int mode,int nk,char* key,char* iv)

- a 是一个 AES 结构，定义了 AES 的内部参数。
- mode 是 AES 的工作模式，关于对称算法的工作模式请参考相关资料。由于此处只对一个 128 位分组进行处理，所以采用 ECB 模式。
- nk 为密钥的长度，这为 16 个字节，即 128 位。
- key 为指向密钥数组的地址。
- iv 是初始化向量，用于 CBC 等模式，ECB 不需要，所以设为 NULL。

跟进 401EC0，代码如下：

```
00401EF8  push    eax
00401EF9  lea     esi, [ebx+6]          ;AES 的加密轮数，这里为 10 轮，即 AES-128
00401EFC  push    ecx
00401EFD  push    ebp
00401EFE  mov     [ebp], ebx
00401F01  mov     [ebp+4], esi
00401F04  call    00401E80              ;aes_reset 函数，设置工作模式及初始向量
00401F09  add     esp, 0C
00401F0C  lea     eax, [esi*4+4]        ;aes 所需要的子密钥数，这为 0x2C，即 44
00401F13  test    ebx, ebx
00401F15  mov     [esp+40], eax
00401F19  jle     short 00401F56
00401F1B  mov     edi, [esp+44]         ;AES 密钥
00401F1F  lea     esi, [esp+14]         ;用于存放 AES 子密钥的缓冲区
00401F23  mov     [esp+48], ebx
00401F27  /push   edi
```

```
00401F28  |call   004021A0            ;将字节数组转换为32位双字
00401F2D  |mov    [esi], eax          ;将128位的AES密钥保存为子密钥的前128位
00401F2F  |mov    eax, [esp+4C]
00401F33  |add    esp, 4
00401F36  |add    esi, 4
00401F39  |add    edi, 4
00401F3C  |dec    eax
00401F3D  |mov    [esp+48], eax
00401F41  \jnz    short 00401F27
00401F43  mov     eax, [esp+40]
00401F47  test    ebx, ebx
00401F49  jle     short 00401F56
...
00401F67  mov     dword ptr [esp+3C], 004084DC
-------------------------------------------------------------------
004084DC  01 00 00 00 02 00 00 00 04 00 00 00 08 00 00 00   ... ... ... ...
004084EC  10 00 00 00 20 00 00 00 40 00 00 00 80 00 00 00   ... ...@...€...
004084FC  1B 00 00 00 36 00 00 00 6C 00 00 00 D8 00 00 00   ...6...l...?..
0040850C  AB 00 00 00 4D 00 00 00 9A 00 00 00 2F 00 00 00  ?..M...?../...
这是用于密钥扩展的轮常量Rcon
-------------------------------------------------------------------
```

从前面介绍的AES密钥扩展算法可以知道，子密钥数组dw[i]，i>4，依赖于dw[i-1]和dw[i-Nk]。此例中Nk=4，即密钥长度为4个32位双字。所以子密钥数组只与dw[i-1]和dw[i-4]有关。对dw数组中下标为4的倍数的元素，将采用如下的方法生成子密钥：

```
if ( i mod Nk =0 )
   temp=SubWord(RotWord(temp)) xor Rcon[i/Nk]
```

下面是AES加密子密钥生成的主要代码：

```
00401F6F  sub     esi, ebp
00401F71  lea     edi, [ebp+ebx*4+8]
00401F75  mov     [esp+10], esi
00401F79  jmp     short 00401F7F
00401F7B  /mov    esi, [esp+10]
00401F7F  |mov    eax, [edi]          ;dw[i-1]
00401F81  |add    esi, edi
00401F83  |mov    edx, eax
00401F85  |mov    ebp, esi
00401F87  |shl    edx, 18
00401F8A  |shr    eax, 8
00401F8D  |or     edx, eax            ;上面代码实现循环右移8位，即RotWord操作
00401F8F  |lea    ecx, [ebx*4]
00401F96  |push   edx
00401F97  |sub    ebp, ecx
00401F99  |call   004021D0            ;SubWord
{
    ......
        004021E2  mov     edx, [esp+8]
        004021E6  mov     ecx, [esp+9]
        004021EA  and     edx, 0FF          ;取第一个字节，即低8位
        004021F0  and     ecx, 0FF          ;取第2个字节
        004021F6  mov     al, [edx+4082DC]  ;查表，即SubBytes操作
        -------------------------------------------------------------
        004082DC  63 7C 77 7B F2 6B 6F C5 30 01 67 2B FE D7 AB 76 c|w{蹇o? g+ 珣
```

```
        004082EC CA 82 C9 7D FA 59 47 F0 AD D4 A2 AF 9C A4 72 C0  蕁萑黝G氦寓瘦
        ......
        004083BC E1 F8 98 11 69 D9 8E 94 9B 1E 87 E9 CE 55 28 DF  猻?i賙銊 UU蜺(
        004083CC 8C A1 89 0D BF E6 42 68 41 99 2D 0F B0 54 BB 16  尅?挎BhA? 癟?
        从004082DC到004083DB共256个字节，为AES的S盒
        -----------------------------------------------
        004021FC  mov    dl, [ecx+4082DC]
        00402202  mov    [esp+8], al
        00402206  mov    eax, [esp+A]
        0040220A  and    eax, 0FF          ;取第3个字节
        0040220F  mov    [esp+9], dl
        00402213  mov    edx, [esp+B]
        00402217  mov    cl, [eax+4082DC]
        0040221D  and    edx, 0FF          ;取第4个字节
        00402223  mov    [esp+A], cl
        ......
}
00401F9E  |mov    edx, [esp+3C]
00401FA2  |add    esp, 4
00401FA5  |mov    ecx, [edx+ebp+C]     ;dw[i-Nk]
00401FA9  |xor    eax, ecx             ;dw[i-Nk] xor SubWord(RotWord(temp))
00401FAB  |mov    ecx, [esp+3C]
00401FAF  |mov    ebp, [ecx]           ;Rcon[i/Nk]
00401FB1  |mov    ecx, 1
00401FB6  |xor    eax, ebp             ;同Rcon异或
00401FB8  |cmp    ebx, 6
00401FBB  |mov    [edi+4], eax         ;生成的子密钥
00401FBE  |jg     short 00402003       ;Nk是否大于6，即是否大于192位
00401FC0  |cmp    ebx, ecx
00401FC2  |jle    004020B7
00401FC8  |lea    ebp, [ebx*4]
00401FCF  |lea    eax, [edi+8]
00401FD2  |sub    esi, ebp
00401FD4  |lea    edx, [esi+edx+10]
00401FD8  |/mov   esi, [esp+48]        ;这段代码为生成3个下标不是Nk倍数的子密钥
00401FDC  ||mov   ebp, [esp+40]
00401FE0  ||add   esi, ecx
00401FE2  ||cmp   esi, ebp
00401FE4  ||jge   004020B7
00401FEA  ||mov   esi, [eax-4]         ;dw[i-1]，取前一个双字
00401FED  ||mov   ebp, [edx]           ;dw[i-Nk]
00401FEF  ||xor   esi, ebp             ;dw[i]=dw[i-1] xor dw[i-Nk]
00401FF1  ||inc   ecx
00401FF2  ||mov   [eax], esi
00401FF4  ||add   edx, 4
00401FF7  ||add   eax, 4
00401FFA  ||cmp   ecx, ebx
00401FFC  |\jl    short 00401FD8
00401FFE  |jmp    004020B7
```

在本例的密钥扩展函数中，同时也生成了用于解密的子密钥，请读者自行分析。

```
00401279  lea    edx, [esp+170]       ;待加密的数据，即输入序列号的十六进制形式
00401280  lea    eax, [esp+3C8]       ;AES结构，包含了前面生成的子密钥
00401287  push   edx
00401288  push   eax
```

```
00401289  call    004023A0              ;aes_encrypt 函数
```

跟进 4023A0 处，可以发现程序先判断是何种工作模式。本例中为 ECB 模式，所以直接调用 aes_ecb_encrypt 函数。上面提到过通常 AES 的加解密过程的实现是通过查 4 个表 T_0、T_1、T_2、T_3 来实现的，本例中即是这种方法。

首先是进行一次 AddRoundKey：

```
004025FF  lea     edi, [esp+10]             ;状态数组
00402603  lea     esi, [ebp+C]              ;加密子密钥
00402606  mov     dword ptr [esp+34], 4     ;循环变量
0040260E  /push   ebx                       ;待加密的数据
0040260F  |call   004021A0                  ;取一个 32 位双字
00402614  |mov    ecx, [esi]                ;取子密钥
00402616  |add    esp, 4
00402619  |xor    eax, ecx                  ;同密钥相异或
0040261B  |add    esi, 4
0040261E  |mov    [edi], eax                ;结果送入状态数组
00402620  |mov    eax, [esp+34]
00402624  |add    edi, 4
00402627  |add    ebx, 4
0040262A  |dec    eax
0040262B  |mov    [esp+34], eax
0040262F  \jnz    short 0040260E
```

接着执行轮函数，主要代码如下：

```
00402631  mov     edi, [ebp+4]              ;轮数，共 10 轮，包括最后一轮
00402634  mov     dword ptr [esp+34], 4;循环变量
0040263C  cmp     edi, 1
0040263F  lea     eax, [esp+10]             ;状态数组
00402643  lea     ecx, [esp+20]
00402647  jle     00402783
0040264D  dec     edi
0040264E  lea     esi, [ebp+20]             ;子密钥
00402651  lea     edx, [edi*4+4]
00402658  mov     [esp+34], edx
0040265C  /mov     edx, [eax+8]             ;取状态数组第三列，记作 a2
0040265F  |mov     ebx, [eax+4]             ;取状态数组第二列，记作 a1
00402662  |shr     edx, 10                  ;a2 右移 16 位
00402665  |and     edx, 0FF                 ;取低 8 位作为索引
0040266B  |add     esi, 10
0040266E  |shr     ebx, 8                   ;a1 右移 8 位
00402671  |mov     edx, [edx*4+40951C]      ;查表 T2，记 temp0= T2[BYTE(a2>>16)]
00402678  |and     ebx, 0FF                 ;取低 8 位作为索引
0040267E  |xor     edx, [ebx*4+408D1C]      ;查表 T1 并且 temp0=temp0^T1[BYTE(a1>>8)]
00402685  |mov     ebx, [eax+C]             ;取状态数组第四列
00402688  |shr     ebx, 18                  ;右移 24 位作为索引
0040268B  |xor     edx, [ebx*4+409D1C]      ;查表 T3，并且 temp0=temp0^T3[BYTE(a3>>24)]
00402692  |xor     ebx, ebx
00402694  |mov     bl, [eax]                ;取状态数组第一列的第一个字节作为索引
00402696  |xor     edx, [ebx*4+40851C]      ;查表 T0，并且 temp0=temp0^T0[BYTE(a0)]
0040269D  |mov     ebx, [esi-14]            ;取轮密钥，即子密钥，记为 dwk[i]
004026A0  |xor     edx, ebx                 ;temp0=temp0^dwk[i]
004026A2  |mov     [ecx], edx               ;将 temp0 送入一缓冲区，记为 y[0]=temp0
004026A4  |mov     edx, [eax+8]             ;取状态数组第三列 a2
004026A7  |mov     ebx, [eax+C]             ;取状态数组第四列 a3
```

```
004026AA  |shr     edx, 8                ;a2 右移 8 位
004026AD  |and     edx, 0FF              ;取低 8 位作为索引
004026B3  |shr     ebx, 10               ;a3 右移 16 位
004026B6  |mov     edx, [edx*4+408D1C]   ;查表 T1,记 temp1=T1[BYTE(a2>>8)]
004026BD  |and     ebx, 0FF              ;取低 8 位作为索引
004026C3  |xor     edx,ebx*4+40951C]     ;查表 T2,并且 temp1=temp1^T2[BYTE(a3>>16)]
004026CA  |mov     ebx, [eax]            ;取状态数组第一列 a0
004026CC  |shr     ebx, 18               ;右移 24 位
004026CF  |xor     edx, [ebx*4+409D1C]   ;查表 T3, 并且 temp1=temp1^T3[BYTE(a0>>24)]
004026D6  |xor     ebx, ebx
004026D8  |mov     bl, [eax+4]           ;取状态数组第二列 a1 的第一个字节
004026DB  |xor     edx, [ebx*4+40851C]   ;查表 T0, 并且 temp1=temp1^T0[BYTE(a1)]
004026E2  |mov     ebx, [esi-10]         ;取轮密钥 dwk[i+1]
004026E5  |xor     edx, ebx              ; temp1=temp1^dwk[i+1]
004026E7  |mov     [ecx+4], edx          ;y1=temp1
......                                   ;省略了计算 y2,y3 过程，请读者自行分析，原理如上
00402773  |dec     edi                   ;轮数
00402774  |mov     [ecx+C], edx
00402777  |mov     edx, eax              ;下面两行连同此行，这三行代码是典型的交换代码
00402779  |mov     eax, ecx              ;即将前面得到的 y 数组作为状态数组进入下一轮变换
0040277B  |mov     ecx, edx
0040277D  \jnz     0040265C
```

上面共执行了 9 轮一样的轮函数，在前面讲过，AES 的最后一轮不同于前面的 Nr-1 轮。对于加密过程少了一个 MixColumns 操作。最后一轮的主要代码如下：

```
00402783  mov     edx, [eax+8]          ;取状态数组的第三列 a2
00402786  xor     ebx, ebx
00402788  shr     edx, 10               ;a2 右移 16 位
0040278B  and     edx, 0FF              ;取低 8 位作为索引
00402791  mov     bl, [edx+4082DC]
;在前面分析过，4082DC 处存放的是 AES 的 S 盒，即此处;执行的是 SubBytes 操作，从下面的分析根
;据取状态数组中字节的顺序可以看出同时也执行的 ShiftRows 操作
;此处记 x2=SubBytes(BYTE(a2>>16))
00402797  mov     edx, [eax+4]          ;取状态数组的第二列 a1
0040279A  shr     edx, 8                ;并右移 8 位
0040279D  and     edx, 0FF              ;取低 8 位作为索引
004027A3  mov     esi, ebx
004027A5  xor     ebx, ebx
004027A7  mov     bl, [edx+4082DC]      ;记 x1=SubBytes(BYTE(a1>>8))
004027AD  mov     edx, [eax+C]          ;取状态数组的第四列 a3
004027B0  shr     edx, 18               ;a3 右移 24 位
004027B3  mov     edi, ebx
004027B5  xor     ebx, ebx
004027B7  mov     bl, [edx+4082DC]      ;记 x3=SubBytes(BYTE(a3>>24))
004027BD  mov     edx, ebx
004027BF  shr     edx, 8
004027C2  shl     ebx, 18
004027C5  or      ebx, edx              ;x3 循环左移 24 位
004027C7  mov     edx, edi
004027C9  shr     edx, 18
004027CC  shl     edi, 8
004027CF  or      edx, edi              ;x1 循环左移 8 位
004027D1  xor     ebx, edx              ;记 temp=x1^x3
004027D3  mov     edx, esi
004027D5  shr     edx, 10
```

```
004027D8  shl     esi, 10
004027DB  or      edx, esi              ;x2 循环左移 16 位
004027DD  xor     ebx, edx              ;temp=temp^x2
004027DF  xor     edx, edx
004027E1  mov     dl, [eax]             ;取状态数组第一列 a0 的第一个字节
004027E3  mov     esi, edx
004027E5  xor     edx, edx
004027E7  mov     dl, [esi+4082DC]      ;x0=SubBytes(BYTE(a0))
004027ED  xor     ebx, edx              ;temp=temp^x0
004027EF  mov     edx, [esp+34]
004027F3  xor     ebx, [ebp+edx*4+C]    ;temp 同轮密钥进行异或
004027F7  mov     [ecx], ebx            ;y[0]=temp
```

上面的这段代码是产生一个 32 位双字的输出 y[0]，其他的三个输出用同样的方法，请读者自行分析。将 y 数组复制到输出缓冲区即为最终的密文。

然后将密文同用户名的 128 位 MD5 散列值相比较，若相同，则注册成功，否则失败。写注册机只需要对用户名的 128 位 MD5 散列值进行 AES 解密，即可得到序列号。详细的源代码请参考光盘。

6.2.6 对称加密算法小结

除了前面介绍的几种分组密码外，还有许多的分组密码没有介绍，如经典的 DES、有趣的 Twofish、安全性极高的 Safer+，以及 NESSIE 里最新提交的 MISTY1、Camellia 等。另外，关于分组密码的工作模式也没有过多的讨论，读者可以通过阅读密码学专著来进一步了解。

如果软件中使用了对称加密算法，那么一般来说，只要知道了算法的类型及密钥，那么就可以做出注册机来。可以用如下方法识别软件中所使用的对称加密算法。

（1）使用 PEiD 的 Kcrypto ANALyzer（Kanal）插件进行识别，一般的对称加密算法都可以识别出来，但也有例外（如 IDEA）。需要注意的是，不能依赖于工具。使用工具只是一种辅助，还需要进一步的跟踪以确定到底是何种算法。

（2）通过每种加密算法的独特的加解密处理过程，如是否为 Feistel 网络，加密轮数，密钥长度，子密钥生成过程，S-box 的值等一系列信息来区分和确定软件中所使用的算法。

（3）为了进一步确定是否为某种对称加密算法，以及此种算法采用何种工作模式（ECB, CBC, CFB, CTR 等），往往还需要自己写一个此种算法的加解密程序来和软件中的算法进行对比检验。

6.3 公开密钥加密算法

在上一节中介绍的对称加密算法，其加密与解密都使用同一个密钥，一旦知道了密钥，那么保护就失败了，这也是其缺点。基于这种考虑，国际著名密码学家 Diffie.W 和 Hellman.M.E 于 1976 年，在其发表的文章“New Directions in Cryptography（密码学的新方向）”中提出了公开密钥（Public Key）加密算法的概念。公钥算法加密与解密使用不同的密钥，加密所使用的叫做公钥（Public Key），而解密所使用的叫做私钥（Private Key）。故而，公钥加密算法又称为非对称加密算法（Asymmetric Key Cryptography）。任何人都可以使用密钥分配者所分发的公钥对信息进行加密，而只有私钥的所有者才可以解密。

公开密钥的设计都是基于 NP 完全问题（关于 NP 问题的详细介绍，请参考数论及密码学相关方面的资料，几乎任何一本讲解密码学的书都会涉及）。如 1978 年，麻省理工学院的三位教授 Rivest、Shamir 及 Adleman 提出的一种基于因子分解问题的公钥系统，它就是现在应用十分广泛的 RSA 公钥算法。后来所出现的背包公钥密码系统（Knapsack）、Elgamal 公钥密码、ECC 等无一不是基于 NP 问题所设计的。

如果软件作者在生成注册码时采用解密算法（私钥），而在软件中检查注册码时使用加密算法（公

钥），即使解密者能够用调试器在自己的机器上对软件进行跟踪分析从而找到公钥，他也不一定能够计算出私钥，自然也就无法得到正确的注册码，更无法写出注册机来。

6.3.1 RSA 算法

RSA 是第一个既能用于数据加密也能用于数字签名的算法，易于理解和操作，应用十分广泛。算法的名字以发明者的名字命名：Ron Rivest、Adi Shamir 和 Leonard Adleman。密码分析者既不能证明也不能否定 RSA 的安全性，但这恰恰说明该算法有一定的可信度。

1．算法原理

① 选取两个大素数：p 和 q，为了获得最大程度的安全性，两数的长度一样。（注：以下所涉及的数论知识不做过多说明，感兴趣的读者请进一步参阅相关书籍。）

② 计算 $n=p\times q$，n 称为模。

③ 计算欧拉（Euler）函数：$\phi(n)=(p-1)\times(q-1)$。

④ 选取加密密钥 e，其与 $\phi(n)$互素。如果选择合适的 e 值，RSA 加解密的速度将快得多，常用的 e 为 3、17 和 65537（$2^{16}+1$）。

⑤ 使用扩展欧几里德算法（Extended Euclid）求出 e 模 $\phi(n)$的逆元 d，即

$$ed\equiv 1 \bmod \phi(n)$$

⑥ 公钥为 e 和 n，私钥为 d，p 和 q 可以丢弃，但是必须保密。

⑦ 加密消息 m 时，将其看成一个大整数，把它分成比 n 小的数据分组，按下面的式子进行加密：

$$c_i\equiv m_i^e \bmod n$$

⑧ 对密文 c 解密时，取每一个加密后的分组 c_i 并计算：

$$m_i\equiv c_i^d \bmod n$$

RSA 加解密总结见表 6-3。

表 6-3 RSA 加解密

公钥	n：两素数 p 和 q 的乘积（q 和 p 必须保密） e：与 $(p-1)(q-1)$ 互素
私钥	d：$e^{-1} \bmod ((p-1)(q-1))$
加密	$c=m^e \bmod n$
解密	$m=c^d \bmod n$

RSA 的安全性依赖于大整数因子分解，但是否等同于大整数因子分解一直未能得到数学上的证明，也就是说，没有证明要解密 RSA 就一定要进行因子分解。目前，RSA 的一些变形算法已被证明等价于大整数因子分解问题，如 Rabin 公开密钥系统。但是目前来看，攻击 RSA 算法最有效的方法便是分解模 n。随着分解大整数方法的改进、计算机速度的提高以及计算机网络的发展（可以使用互联网上成千上万的计算资源同时进行因子分解），为了保证 RSA 系统的安全性，其密钥的位数一直在增加。目前，一般认为 RSA 需要 1024 位或更长的模数才有安全保障。常见的因子分解算法有试除法（Trial Division）、Pollard-ρ 因子分解算法、Pollard p-1 因子分解算法、椭圆曲线因子分解算法（Elliptic Curve Factoring Algorithm）、随机平方因子分解算法（Random Square Factoring Algorithm）、连分式因子分解算法（Continued Factoring Algorithm）、二次筛法（Quadratic Sieving）、数域筛法（Number Field Sieving）（包括一般（广义）数域筛法（GNFS）和特殊数域筛法（SNFS）），本书不作过多介绍，详细请参考有关资料。

2. RSA 计算

RSA 涉及的各公式的计算请参考数论等相关资料。在本书光盘里直接提供了相关计算工具。

- Gcd.exe：求最大公因子；
- MulInv.exe：求模逆元，形式为 $ed \equiv 1 \bmod \phi(n)$，其中 $\phi(n)=(p-1)(q-1)$；
- Powmod.exe：计算 $m \equiv c^d \bmod n$；
- CE.EXE：计算 d，用法： CE <p> <q> <e>；
- Factor：大数计算器，可进行因式分解；
- RSATool：一款非常强大的 RSA 辅助工具，具有图形界面，包含上述工具功能（注意，输入十六进制时，字母一定要以大写形式输入）。
- Bigcalc：大数计算器。

下面举一个例子。

设 $p=37$，$q=41$（十进制），那么：

$$n=pq=37\times 41=1517$$

$$\phi(n)=(p-1)(q-1)=36\times 40=1440$$

选取 $e=17$，则 $d=17^{-1} \bmod 1440=593$，公开 e 和 n，将 d 保密，丢弃 p 和 q（但也必须保密）。加密消息：

$$m=1234567$$

首先将其分成小的分组，在此例中，按三位数字一组就可以进行加密。

$$m_1=123$$

$$m_2=456$$

$$m_3=007\text{（不足在左边填充 0）}$$

加密：

$$123^{17} \bmod 1517 \equiv 1107 \bmod 1517 = c_1$$

$$456^{17} \bmod 1517 \equiv 1292 \bmod 1517 = c_2$$

$$007^{17} \bmod 1517 \equiv 645 \bmod 1517 = c_3$$

密文如下：

$$c=1107\ 1292\ 645$$

解密消息时需要私钥 593 进行相同的指数运算。例如：

$$1107^{593} \bmod 1517 \equiv 123 \bmod 1517 = m_1$$

$$1292^{593} \bmod 1517 \equiv 456 \bmod 1517 = m_2$$

$$645^{593} \bmod 1517 \equiv 007 \bmod 1517 = m_3$$

3. RSA 算法在加密上的应用

大多数共享软件的注册码计算设计得都不是很好，比较容易被解密做出注册机来。如果采用公钥算法作为注册保护机制，如 RSA，可以参考如下思路。

① 要求其模数 n 有一定的长度（至少为 512 位，推荐取 1024 位或更长），以防止在较短的时间内被因式分解，从而使算法被攻破。但注册码的长度也因此变长了，可能给用户带来不方便，可以采取以 License file（授权文件）的形式分发给注册用户。

② 随机生成密钥对时，要采用尽可能好的随机数生成算法，以免被轻而易举地猜到公钥或私钥。

③ 也可以在注册机中用公钥 e 对用户名进行加密得到注册码，在软件中对用户输入的注册码用私钥 d 进行解密得到用户名。此时公钥 e 就不能取常用的 3、65537 等值，否则一旦被猜出或计算出 e，也可以做出注册机。

④ 这种方法只是为了防止被解密者写出注册机，无法防止通过修改程序中跳转指令的方法来破解软件。为了防止别人修改程序文件，可以用注册码中的一部分来加密程序代码或数据。

下面举一个例子讲解 RSA 应用。该例采用大数运算库 Miracl 来实现 RSA，该库提供 C 与 C++两种接口。读者可参考其说明文档将其安装好。

（1）随机生成密钥对

可以自己编程随机搜索大素数，请参阅相关的资料。此处由于是举例，采用 RSATool 工具生成 128 位 RSA 的参数：

p = C75CB54BEDFA30ABh

q = A554665CC62120D3h

n = 80C07AFC9D25404D6555B9ACF3567CF1h

d = 651A40B9739117EF505DBC33EB8F442Dh

e = 10001h

（2）在软件中判断注册码

在软件中用公钥 e 对输入的注册码进行加密来得到密文，并与用户名比较。若相同，则认为是注册成功，否则注册失败。

```
char szName[]="pediy";                                  //用户输入的用户名
char szSerial[256]="404E85B5FEF4AE26FC2229D028BE01AD";
                                                        //用户输入的注册码
big n,e,c,m;                                            //Miracl中的大数类型
mip->IOBASE=16;                                         //设定十六进制输入输出模式
n=mirvar(0);
e=mirvar(0);
c=mirvar(0);
m=mirvar(0);                                            //以上初始化大数
cinstr(m,szSerial);                                     //将序列号的十六进制字符转换成大数m
cinstr(n," 80C07AFC9D25404D6555B9ACF3567CF1");//初始化模数n
cinstr(e,"10001");                                      //初始化公钥e
if(compare(m,n)==-1)                                    //m<n，才能对消息m加密
{
    powmod(m,e,n,c);                                    //计算密文c== m^e mod n
    big_to_bytes(0,c,&szSerial,0);                      //将c从大数转换成字节数组
    if(lstrcmp(szName,szSerial)!=0)
        //错误的注册码
    else
        //正确的注册码
}
```

（3）制作注册机

思路是：将用户名 c 用私钥 d 解密，得到的数据为注册码。代码如下：

```
char szName[]="pediy";                          //用户名
char szSerial[256]={0};                         //注册码
big n,e,c,m;
mip->IOBASE=16;
n=mirvar(0);
e=mirvar(0);
c=mirvar(0);
m=mirvar(0);
bytes_to_big(len,szName,c);                     //将用户名转换成大数 c
cinstr(n,"80C07AFC9D25404D6555B9ACF3567CF1");
cinstr(d,"651A40B9739117EF505DBC33EB8F442D");//初始化私钥 d
powmod(c,d,n,m);                                //计算 m ≡ c^d mod n
cotstr(m,szSerial);                             //m 的十六进制字符串即为注册码
```

由于 m 往往包含不可显示字符，必须将其转换成十六进制字符串，或将其编码变成可显示字符，比如采用 Base64 编码等。

4. 攻击 RSA 保护

采用 RSA 保护时，模数 n 的位数不能太少。实际中有些共享软件虽然采用了 RSA，但 n 太短，这样就失去了用 RSA 保护的意义。攻击 RSA 保护的软件，一般是通过跟踪分析得到 n，再将 n 因子分解，从而求出私钥 d，进而做出注册机。

以光盘中的 RSAKeyGenMe.exe 为例。用 OllyDbg 运行此 KeyGenMe，输入假的姓名与序列号，用“bpx GetDlgItemTextA”设断，来到如下代码处：

```
004011C5  mov   dword ptr [ebp+234], 10       ;mip->IOBASE=16,十六进制模式
004011CF  call    00401780
...
004011EF  lea     edx, [esp+20]               ;指向输入的序列号 szSerial
004011F3  mov     ebx, eax
004011F5  push    edx                         ;参数 szSerial 入栈
004011F6  push    edi                         ;参数 m
004011F7  call    004039A0                    ;cinstr(m,szSerial)
004011FC  push  0040C044                      ;n=80C07AFC9D25404D6555B9ACF3567CF1
00401201  push    esi
00401202  call    004039A0                    ;cinstr()函数初始化模 n
00401207  push    0040C03C                    ;公钥 e "10001" (65537 的十六进制)
0040120C  push    ebp
0040120D  call    004039A0                    ;cinstr(e,"10001")
00401212  push    esi                         ;n
00401213  push    edi                         ;m
00401214  call    00402680                    ;compare(m,n),m<n 时才对消息 m 加密
00401219  add     esp, 30
0040121C  cmp     eax, -1
0040121F  jnz     004012C3
00401225  push    ebx                         ;c
00401226  push    esi                         ;n
; D ESI 可以在数据窗口查看 n，第 2 组数据 88026C 指向的就是 n，其是以倒序排列形式存在的
------------------------------------------------------------------------
003C5B48  04 00 00 00 54 5B 3C 00 00 00 00 00 F1 7C 56 F3  ˜...T[<.....駑V
003C5B58  AC B9 55 65 4D 40 25 9D FC 7A C0 80 00 00 00 00    UeM@%瀚z 绂....
------------------------------------------------------------------------
```

```
00401227  push    ebp                     ;e
00401228  push    edi                     ;m
00401229  call    00403370                ;powmod(m,e,n,c), c≡m^e mod n
0040122E  lea     eax, [esp+E8]           ;szBuffer, 缓冲区，转换后的数据存放在此
00401235  push    0
00401237  push    eax
00401238  push    ebx                     ;c
00401239  push    0
0040123B  call    00403120                ;big_to_bytes(0,c,szBuffer,0);
```

从上面的代码中获得了两个重要的参数模 n 和公钥 e，其值分别为：

n = 0x80C07AFC9D25404D6555B9ACF3567CF1

e = 0x10001

因为 n 并不是很大，直接将其因式分解得到 p 和 q。因式分解可以用 RSATool ,Factor 或 PPSIQS 等工具。RSATool 工具是图形界面，操作比较方便。选择进制（Number Base）为 16，将模数 80C07AFC9D25404D6555B9ACF3567CF1 填入“modulus(N)”中，单击 Factor N 按钮进行因式分解。128 位的大数在不到一分钟的时间内就被分解了：

PRIME FACTOR: A554665CC62120D3

PRIME FACTOR: C75CB54BEDFA30AB

即：

$n=p\times q$=A554665CC62120D3×0xC75CB54BEDFA30ABh

知道 p 和 q，便能计算出 $\phi(n)=(p-1)(q-1)$，既而利用欧几里德扩展算法很容易求出 d，利用 RSATool，输入 e 后，单击“Calc.D”按钮便可计算出 d：

d = 651A40B9739117EF505DBC33EB8F442Dh

至此，这个 RSA-128 被破译。可以直接写出注册机了，源码见光盘。

这里讲解一下如何用工具解密单个数据。设输入的用户名 m 为 pediy，其 ASCII 码的十六进制值为：

m=7065646979h

生成注册码 c 的加密算法为：

$$c = m^d \bmod n$$

即

$7065646979^{651A40B9739117EF505DBC33EB8F442D} \bmod$ 80C07AFC9D25404D6555B9ACF3567CF1

这时就需要用到 Bigcalc 这个大数计算器了。先设置进制（Base）为 16，设：

X=7065646979

Y=651A40B9739117EF505DBC33EB8F442D

Z=80C07AFC9D25404D6555B9ACF3567CF1

以输入 X 为例，在 Input&Output 窗口中，输入 7065646979，接着单击“Store”按钮，然后单击“X”按钮，这时 7065646979 已经存储到变量 X 中去了。

利用相同的办法，存储 Y 和 Z，最后单击“X^Y%Z”按钮计算 $c\equiv m^d \bmod n$，这时同样在 Input&Output 窗口中得到结果：404E85B5FEF4AE26FC2229D028BE01ADh。

此例中的 RSA 模 n 的长度为 128 位，但是如果软件的模数 n 为 512 或更高，除非是特殊的一些 n，可以用特殊的因子分解算法来分解，但需要太多的时间，如果不是出于科学研究因子分解算法的目的，那么就没有必要花费如此的精力。可以采用另外一种技术：替换 n，即首先通过编程或利用 RSATool 之类的工具生成与目标软件中的 n 相同位数长度的 n（此时，其私钥 d、p 和 q 已知），然后利用逆向技术，用此 n 去替换软件中的 n，然后用自己的 d 来做出注册机。

6.3.2 ElGamal 公钥算法

1985 年，T.ELGAMAL 教授在他的一篇论文“A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms（一种基于离散对数的公钥加密系统和签名体系）”中提出了 ElGamal 公钥算法。其安全依赖于在有限域上计算离散对数的困难性。

1．算法原理

密钥对产生办法。首先生成一个大素数 p，模 p 的乘法群 z_p^* 上的一个生成元 g 和一个小于 p 的大数 x（$1 \leqslant x \leqslant p-2$），然后计算：

$$y \equiv g^x \bmod p$$

公钥为 y、g 和 p，私钥是 x。其中 g 和 p 可以由一组用户共用。

（1）ElGamal 签名

对消息 M 签名时，生成一个随机数 k，$1 \leqslant k \leqslant p-2$，且 $\gcd(k,p-1)=1$，即 k 与 $p-1$ 互素，计算：

$$a \equiv g^k \bmod p$$

再用扩展欧几里德算法对下面方程求解 b：

$$M \equiv (xa+kb) \bmod (p-1)$$

当 k 与 $p-1$ 互素时，b 有一个解。签名即为(a,b)。随机数 k 必须丢弃。

验证签名时，需要满足下式：

$$y^a a^b \equiv g^M \bmod p$$

同时要检验是否满足 $1 \leqslant a < p$，否则签名容易伪造。

（2）ElGamal 加密算法

被加密信息为 M，首先选择一个随机数 k，且 k 与 $p-1$ 互素，计算：

$$a \equiv g^k \bmod p$$

$$b \equiv y^k M \bmod p$$

其中（a, b）为密文，是明文的两倍长。解密时计算：

$$M \equiv b / a^x \ (\bmod p)$$

由此也可以看出，ElGamal 有一个不足之处就是密文的长度是明文的两倍。而另外一种签名算法，Schnorr 签名系统的密文比较短，这由其系统内的单向散列函数 h 来决定，感兴趣的读者可以参考相关资料。

2．ElGamal 算法在加密上的应用

ElGamal 提供了签名和加密两种算法。本例利用 ElGamal 签名算法生成共享软件注册码。思路如下所述。

（1）随机生成密钥对

此处采用 RSATool 工具生成 128 位大素数 p（实际操作中建议 p 选取 512 位以上）和随机数 g 和 x。各数据如下：

大素数：p = AE6F8E3B6399D3A3h

随机数：g = 92AFA3B6E8889333h（$g < p$）

随机数：x = 81BC15BBB48350D3h（$x < p$）

计算：　$y \equiv g^x \bmod p \equiv$ 2E646151C7E5A00Fh mod p

公钥为 y、g 和 p；私钥为 x，且必须保密。

（2）在软件中判断注册码

将输入的注册码分成两部分 a（签名）和 b（签名），再用 MD5 算法取得用户名的散列值，最后用算式 $y^a a^b \equiv g^M \bmod p$ 验证。若相同，则认为是注册成功，否则注册失败。

用大数运算库 MIRACL 实现的主代码如下：

```
//  szSerial1 为输入序列号的前半部，szSerial2 为输入序列号的后半部，中间以“-”分开
//  M=MD5(用户名)
cinstr(a,szSerial1);                          //初始化签名 a
cinstr(b,szSerial2);
cinstr(p,"CE892335578D3F");                   //初始化模数 p
cinstr(g,"473FE7D24CB6A6");                   //初始化生成元 g
cinstr(y,"A3CCD85BBD896");
powmod(y,a,p,result1);                        //result1=ya mod p
powmod(a,b,p,result2);                        //result2=ab mod p
mad(result1,result2,result1,p,p,result3);//result3=result1×result2 mod p
                    //模运算法则：(a×b)mod n = ((a mod n) × (b mod n)) mod n
                    //所以 result3 = y^a a^b(mod p)
powmod(g,M,p,result4);                        //result4=g^M mod p
if(compare(result3,result4)==0)               //判断签名是否正确
        正确的注册码;
else
        错误的注册码;
```

注册机编写过程就是用私钥 x 生成对数 a 和 b，并将 a 和 b 连接起来即可。

3. 攻击 ElGamal 算法保护

ElGamal 在签名验证过程中用到了参数 y、g 和 p。解密的思路就是根据 $y \equiv g^x \bmod p$，求离散对数获得 x 的值，即可攻破。

以 ElgamalKGM.exe 为例演示其过程。用 OllyDbg 载入，输入假的用户名和序列号，用“bpx GetDlgItemTextA”设断，按 F9 键运行，可来到如下代码处：

```
00401167  call    esi                 ;GetDlgItemTextA
00401169  test    eax, eax            ;判断是否输入用户名
0040116B  jnz     short 0040118B
...
0040119B  call    esi                 ;USER32.GetDlgItemTextA
0040119D  mov     esi, eax
0040119F  test    esi, esi            ;判断是否输入序列号
004011A1  jnz     short 004011C1
...
004011C1  lea     eax, [esp+2C]       ;用户输入的序列号 szSerial
004011C5  push    2D
004011C7  push    eax
004011C8  call    004078C0            ;strchr()，检查序列号中是否含有“-”字符
004011CD  add     esp, 8
004011D0  test    eax, eax
004011D2  jnz     short 004011F2
...
0040120B  sub     edi, edx            ;如果含有“-”，则计算注册码前一部分的长度
...
00401259  push    edi                 ;前面一部分代码检测注册码是否为十六进制字符
0040125A  lea     ecx, [esp+2E0]
00401261  push    eax
```

```
00401262  push    ecx
00401263  call    00407730              ;将“-”前面的字符复制到szSerial1
00401268  sub     esi, edi
0040126A  lea     edx, [esp+edi+39]
0040126E  dec     esi
0040126F  lea     eax, [esp+3B0]
00401276  push    esi
00401277  push    edx
00401278  push    eax
00401279  call    00407730              ;将“-”后面的字符复制到szSerial2
```

以上代码的主要功能是将注册码以“-”为分隔，将其分成两部分，分别复制到两个缓冲区当中，作为下面将要用到的待验证的签名 *a*, *b*。

```
0040127E  lea     ecx, [esp+10C]        ;MD5 Context 的地址
00401285  push    ecx
00401286  call    00401460
0040128B  lea     edx, [esp+44]         ;指向字符串”pediy”
0040128F  push    3
00401291  lea     eax, [esp+114]
00401298  push    edx
00401299  push    eax
0040129A  call    00401490              ;MD5Update函数，将“pediy”填充到context
0040129F  add     esp, 28
004012A2  lea     ecx, [esp+14C]
004012A9  push    ecx
004012AA  call    [<&KERNEL32.lstrlenA>]
004012B0  push    eax
004012B1  lea     edx, [esp+150]
004012B8  lea     eax, [esp+F8]
004012BF  push    edx
004012C0  push    eax
004012C1  call    00401490              ;MD5Update，将用户名填充到context中
004012C6  lea     ecx, [esp+100]
004012CD  lea     edx, [esp+220]        ;szHash，用来保存MD5散列值
004012D4  push    ecx
004012D5  push    edx
004012D6  call    00401540              ;MD5Final函数，得到最终的散列值
```

通过对前面介绍 MD5 章节的学习，不难分析出上面的代码是标准的 MD5，即对字符串“pediy+用户名”进行散列，得到 128 位的散列值。

```
004012DF  push    0
004012E1  mov     dword ptr [eax+234], 10  ;mip->IOBASE=16
004012EB  call    00402480                 ;mirvar(0)
...
00401347  push    0
00401349  mov     [esp+64], eax
0040134D  call    00402480                 ;mirvar(0)
00401352  add     esp, 40
00401355  lea     ecx, [esp+214]           ;这里指向前面得到的MD5散列值
0040135C  mov     [esp+20], eax
00401360  push    ebx
00401361  push    ecx
00401362  push    10
00401364  call    004057F0                 ;bytes_to_big函数，将散列值转化为大数变量
```

上面的这些代码主要实现了初始化一些大数变量以及将 MD5 散列值转化为大数变量的功能。程序从这里开始调用 miracl 库函数对大数进行处理。如果要看懂这些代码，需要对 miracl 库进一步熟悉。在稍后的章节中，将介绍如何识别 miracl 中的函数。

```
00401369  lea     edx, [esp+2E8]        ;szSerial1
00401370  push    edx
00401371  push    edi
00401372  call    00404E70              ;cinstr 函数，将 szSerial1 转化成大数变量 a
00401377  mov     ecx, [esp+28]
0040137B  lea     eax, [esp+3B8]        ;szSerial2
00401382  push    eax
00401383  push    ecx
00401384  call    00404E70              ;cinstr 函数，将 szSerial2 转化成大数变量 b
00401389  push    0040D09C              ;ASCII "CE892335578D3F"
0040138E  push    esi
0040138F  call    00404E70              ;cinstr,初始化 p
00401394  mov     edx, [esp+34]
00401398  push    0040D08C              ;ASCII "473FE7D24CB6A6"
0040139D  push    edx
0040139E  call    00404E70              ;cinstr，初始化 g
004013A3  mov     eax, [esp+48]
004013A7  push    0040D07C              ;ASCII "A3CCD85BBD896"
004013AC  push    eax
004013AD  call    00404E70              ;cinstr，初始化 y
```

上面这段代码初始化 ElGamal 公钥系统的参数 p、g 和 y，以及将要被验证的签名 a,b。可以猜想下面即将进行 ElGamal 的签名验证。

```
004013B2  mov     ecx, [esp+50]
004013B6  push    ebp                   ;result1
004013B7  push    esi                   ;p
004013B8  push    edi                   ;a
004013B9  push    ecx                   ;y
004013BA  call    00404840              ;powmod(y,a,p,result1)
004013BF  mov     edx, [esp+5C]
004013C3  mov     eax, [esp+58]
004013C7  add     esp, 44
004013CA  push    edx                   ;result2
004013CB  push    esi                   ;p
004013CC  push    eax                   ;b
004013CD  push    edi                   ;a
004013CE  call    00404840              ;powmod(a,b,p,result2)
004013D3  mov     edi, [esp+34]
004013D7  mov     ecx, [esp+28]
004013DB  push    edi                   ;result3
004013DC  push    esi                   ;p
004013DD  push    esi                   ;p
004013DE  push    ebp                   ;result1
004013DF  push    ecx                   ;result2
004013E0  push    ebp                   ;result1
004013E1  call    00404730              ;mad 函数
```

上面这段代码首先两次调用了 powmod 函数，分别计算：

$$result1=y^a \bmod p$$

$$result2=a^b \bmod p$$

接着调用了 miracl 中的 mad 函数，其详细说明请读者参阅 miracl 手册。这里用于实现计算：

$$result3=result1*result2 \bmod p$$

即计算：

$$result3=y^a a^b \bmod p$$

```
004013E6  mov     ebp, [esp+48]       ;result4
004013EA  mov     edx, [esp+38]       ;M
004013EE  push    ebp                 ;result4
004013EF  push    esi                 ;p
004013F0  push    ebx                 ;M
004013F1  push    edx                 ;g
004013F2  call    00404840            ;powmod 函数，powmod(g,M,p,result4)
004013F7  push    ebp                 ;result4
004013F8  push    edi                 ;result3
004013F9  call    004031C0            ;compare 函数
004013FE  add     esp, 40
00401401  test    eax, eax            ;如果不相等，则注册失败
00401403  jnz     short 0040142D
```

可以看出程序紧接着计算下式的值：

$$result4=g^M \bmod p$$

即 result3 和 result4 分别是签名验证公式的左右两边的值。然后对 result3 和 result4 进行比较，若相等，则签名是有效的，注册成功；否则，签名无效，注册失败。

从以上的分析可以看出，这是标准的 ElGamal 签名验证算法。经过跟踪分析得到的参数如下：

p = CE892335578D3F

g = 473FE7D24CB6A6

y = A3CCD85BBD896

a = szSerial1

b = szSerial2

M = MD5(pediy+用户名)

现在，最关键的是通过上面的信息来想办法得到其私钥 x，已知 x 满足下面的式子：

$$y \equiv g^x \bmod p$$

求 x 这是一个在有限域上的离散对数问题，而这也是数学中的一个 NP 完全问题。如果 p 的位数足够大，那么在现有的计算条件及算法水平上，是无法得到 x 的。但是，随着密码学家和数学工作者的不懈努力，在求有限域的离散对数问题上提出了许多算法。现在，比较常见的求离散对数的算法有 Baby-Step Giant-Step Algorithm（大步小步法）、Pollard-ρ Algorithm、Pohlig-Hellman Algorithm、Index-Calculus Algorithm，虽然这些算法仍不能完全解决有限域上的离散对数问题，但是这些算法相当有意思，需要相当的数学功底才能读懂并且通过程序来实现，感兴趣的读者可以参阅相关资料。

求离散此例中的 p 只有 56 位，可以通过本书光盘中提供的 DLPTool 来求出 x，此工具只实现了 Pollard-ρ 算法。利用此工具可以得到：

x = 264D8D82C7AAB8h

在做注册机的时候，需要计算 ElGamal 的签名(a,b)。随机生成一个 k，利用下式计算 a：

$$a \equiv g^k \bmod p$$

但是需要注意的是，k 必须与 p-1 互素，这样才能保证能得到唯一的 b。此例中：p=0xCE892335578D3F，p-1=0xCE892335578D3E，因子分解后可以得到其两个素数因子 f_1=2，f_2=0x6744919AABC69F，那么在生成 k 的时候，只要 k 不包含这两个素因子就可以了，即 gcd(k,p-1)==1。

因为 b（签名）满足：$M \equiv (xa + kb) \bmod (p-1)$

可以推算出：

$$m\text{-}xa \equiv kb \bmod p\text{-}1$$

$$(m\text{-}xa)k^{-1} \equiv b \bmod p\text{-}1 \qquad (6\text{-}7)$$

这里 k^{-1} 是 k 模 p-1 的乘法逆元，可以用扩展欧几里德算法（Extended Eculidean Algorithm）求得。式（6-7）也可以如下表示：

$$b \equiv (m\text{-}xa)k^{-1} \bmod p\text{-}1$$

也就是说，b 是$(m\text{-}xa)k^{-1} \bmod p$-1 的剩余系中的元素。将 a 和 b 通过“-”连接起来，便得到了最后的注册码，注册机源代码见光盘。

ElGamal 算法的一些注意事项：

（1）如果 m-xa 出现负值，即最后得到的签名 b 为负数，一般来讲此时需要将 b 不断地加上 p-1，直到出现一个小于 p-1 的非负值时为止，此时这个非负值便是要求的签名 b。在本例中，m 为 MD5 散列值，共 128 位，转化为 128 位的大数，私钥 x 和签名 a 都是小于等于 56 位的，其乘积小于等于 112 位，因此 m-xa 将恒得到正数，无须考虑签名出现负数的情况。

（2）签名时随机生成的 k 必须要同 p-1 互素，g 必须为乘法群 z_p^* 的一个生成元。

（3）当签名出现 b=0 的时候，一定要重新对消息进行签名，如若不然，则很容易求出私钥 x，这一点要引起重视。

（4）共享软件作者在给用户分发注册码时，一定不要用同一个 k 对不同的用户名进行签名，并将签名分发给用户，而要采用不同的 k，每一个用户对应一个 k，或者每个用户对应一对(x,k)，并且要定期更换模 p，这样才能达到最大程度上的安全。

如果没有采用第 3 个建议，而采用同一个 k 和私钥 x 对不同的用户进行签名，那么对于 ElGamal 存在这样一种攻击。讲解如下：

用户 A 和 B，使用相同的 k 对其进行签名，得到的签名分别为(a_1,b_1)，(a_2,b_2)，其用户名的散列值分别为 M_1,M_2，$m_1 \equiv M_1 \bmod p$-1，$m_2 \equiv M_2 \bmod p$-1。令：

$$a=a_1=a_2 \equiv g^k \bmod p$$

有下式：

$$m_1 \equiv xa+kb_1 \bmod p\text{-}1 \qquad (6\text{-}8)$$

$$m_2 \equiv xa+kb_2 \bmod p\text{-}1 \qquad (6\text{-}9)$$

根据模算术运算的性质，式（6-9）-式（6-8）（或者式（6-8）-式（6-9））得：

$$k(b_2\text{-}b_1) \equiv (m_2\text{-}m_1) \bmod p\text{-}1$$

设 $d = \gcd(b_2 - b_1, p-1)$，可以证明 $d \mid (m_2 - m_1)$。令

$$m' = \frac{m_2 - m_1}{d},\ b' = \frac{b_2 - b_1}{d},\ p' = \frac{p-1}{d}$$

则 $m' \equiv kb' \bmod p'$，所以 $k \equiv (b')^{-1} m' \bmod p'$，进而可以利用式（6-8）或式（6-9）求出私钥 x。

使用 miracl 库，很容易实现上述的攻击方法。可见，一定不要用同样的 k 和私钥 x 对不同的用户名进

行签名，只要解密者想办法得到两个或两个以上正确的注册信息，那么就不需要通过求离散对数，也可以求出私钥 x，进而做出注册机。这种保护就算是失败的。生成随机数 k 的算法在注册机源码中已经给出，读者可以参考。

另外，往往在实际使用中，模数 p 的位数一般都比较大，通过求离散对数解出私钥 x 一般不可行。此时，可以采取类似于上节介绍的 RSA 中的“patch n（替换 n）”的方法来达到破解的目的，即生成一个同软件中的 p 相同位数的 ElGamal 系统参数，替换掉目标软件中的参数，进而做出注册机。

6.3.3 DSA 数字签名算法

美国国家标准与技术局（NIST）于 1991 年，在借鉴了 ElGamal 及 Schnorr 签名算法的基础上，公布了数字签名标准（Digital Signature Standard）。该标准采用的算法为 DSA（Digital Signature Algorithm）。DSA 在公布之后立即产生了巨大的反响，有赞成的也有反对的。因 DSA 出自 NSA（美国国家安全局）之手，由 NIST 采用并公布，工业界没有任何插手余地，再加上公布之初，DSA 仍然存在一些考虑不周到的地方，故而几经修改，直到今天的版本 FIPS 186-2（Federal Information Processing Standards，联邦信息处理标准）。目前，对于 DSA 的攻击依然在继续，但是仍然没有充分的证据证明其安全性存在很大的弱点。DSA 的应用也越来越广泛。

算法原理

DSA 使用如下的一些参数：

p：L 位长的素数。L 是 64 的倍数，范围从 512 到 1024，$2^{L-1}<p<2^L$；

q：p-1 的素因子，取值范围为 $2^{159}<q<2^{160}$；

g：$g=h^{(p-1)/q} \bmod p$，其中 h 满足 $h<p-1$，且 $h^{(p-1)/q} \bmod p>1$；

x：$0<x<q$，其中 x 为私钥；

y：$y=g^x \bmod p$，其中（p, q, g, y）为公钥；

k：随机或伪随机数，$0<k<q$。

整数 p、q 和 g 可以公开，并且可由一组用户共享，每个用户分别具有各自的私钥 x 和公钥 y。为了保证最大限度的安全，x 和 y 需要在一段时间内进行更新。参数 x 和 k 仅用于生成签名，必须保密。对每个不同的签名，k 必须不同（这与上节提到的 ElGamal 中的 k 也必须不同的原理相同）。

签名及验证协议如下。

输入：待签名的消息 M，公钥 p,g,q，私钥 x，随机数 k

输出：签名 r，s

算法：

$$r=(g^k \bmod p) \bmod q$$

$$s=(k^{-1})((\text{SHA-1}(M)+xr)) \bmod q$$

k^{-1} 是 k 模 q 的乘法逆元，SHA-1(M)是指使用 SHA-1 散列算法对消息进行散列，进而产生 160 位的输出。

注意：如果在签名的过程中，k,r,s 三者有一个为 0，那么必须重新生成随机数 k，重新进行签名！否则解密者将很容易求出用户的私钥 x。

DSA 签名验证算法：

输入：待验证的消息 M'，公钥 p, g, q，公钥 y，签名 r'，s'

输出：若签名正确则返回 TRUE，否则返回 FALSE

算法：

（a）如果 $r'\in(0,q)$并且 $s'\in(0,q)$，则进行签名验证，否则签名无效，返回 FALSE。

（b）在满足（a）的前提下，进行如下计算：

$$w=(s')^{-1} \bmod q$$

$$u_1=((\text{SHA-1}(M')\times w) \bmod q$$

$$u_2=((r')w) \bmod q$$

$$v=((g^{u1}\times y^{u2}) \bmod p) \bmod q$$

若 $v=r'$，则签名验证成功。

在使用 DSA 数字签名系统时，程序员可以自己编程实现生成系统参数，也可以使用 DSA Tool 来生成。

本书光盘中附有一例 DSA Sample，笔者简单地实现了 DSA 的签名及验证过程。请读者参考光盘中的源码。

DSA 的安全性同样是基于有限域的离散对数问题，故而其攻击也都是尝试去解决离散对数问题，即 DLP。常见的攻击算法有 Brute Force，Pollard-pho，Pohlig-Hellman，indexcalculus。但是对于 DSA，p 一般都在 512 位以上，攻击需要花费大量的时间，而且不一定能求出私钥 x。所以，通常也可以采取“Patch p”的方法，即替换 p 及 DSA 系统的其他参数，从而达到破解的目的。

6.3.4 椭圆曲线密码编码学（Elliptic Curve Cryptography）

椭圆曲线（Elliptic Curve）作为代数几何学中一个重要问题已经有一百多年的研究历史，但直到 1985 年 N.Kobitz 和 V.Miller 才分别独立地将椭圆曲线引入密码学。由于其相比于 RSA 等公钥算法，在使用较短的密钥长度而能得到相同程序的安全性，而使得椭圆曲线密钥学的应用越来越广泛，对其的研究也如火如荼，预测未来 ECC（Elliptic Curve Cryptography）将取代 RSA 成为主流的公钥算法。

对于椭圆曲线的完整的数学描述已超出本书的范围。本书只介绍基于 GF(p)上的椭圆曲线及 ECC 在软件保护中的应用所需要的基本知识，GF(2^m)上的椭圆曲线本书不作介绍，读者可以参阅其他资料。

1．基本概念

（1）群

阿贝尔（Abelian）群 $(G,*)$ 由集合 G 和二进制操作 $*$ 组成：$G\times G\to G$ 满足如下属性：

- （结合律）$a*(b*c)=(a*b)*c$，$a,b,c\in G$
- （存在幺元）存在元素 $e\in G$，使得对所有的 $a\in G$ 都有 $a*e=e*a=a$
- （存在逆元）对每个 $a\in G$，存在 $b\in G$，使得 $a*b=b*a=e$，b 叫做 a 的逆元
- （交换律）$a*b=b*a$，$a,b\in G$

群上的操作常称为加法（+）或者乘法（•）。对于前者，相应的群叫做加法群，加法幺元常用 0 表示，a 的加法逆元用 $-a$ 表示。对于后者，相应的群叫做乘法群，乘法幺元常用 1 表示，a 的乘法逆元用 a^{-1} 表示。当 G 为一个有限集时，称群是有限的，即有限群，同时 G 中的元素个数称为 G 的阶（order of G）。

例如，令 p 为一个素数，并且令 $F_p=\{0,1,2,\cdots,p-1\}$ 表示模 p 的整数集。那么 $(F_p,+)$ 表示一个阶为 p 的有限加法群，其加法幺元为 0，+操作定义为模 p 的整数相加。又如，$(F_p^*,\cdot)$ 表示一个阶为 p-1 的有限乘法群，其乘法幺元为 1，F_p^* 表示 F_p 中的非零元，•操作定义为整数模 p 相乘。

若 G 是一个阶为 n 的有限乘法群且 $g\in G$，那么使得 $g^t=1$ 的最小正整数 t 叫做 g 的阶。通常 t 存在且是 n 的一个因子。集合 $\langle g\rangle=\{g^i,0\leqslant i\leqslant t-1\}$ 是同 G 具有相同操作的群，叫做由 g 生成的 G 的循环子群（cyclic subgroup of G generated by g）。对于加法群，g 的阶是使得 $tg=0$ 的最小正整数 t，且 t 是 n 的一个因子，集合 $\langle g\rangle=\{ig,0\leqslant i\leqslant t-1\}$。这里 tg 表示加 t 个 g。如果 G 有一个阶为 n 的元素 g，那么 G 叫做循环群且 g 叫做 G 的生成元（generator of G）。

（2）椭圆曲线群

令 p 为素数，F_p 为模 p 的整数域。F_p 上的椭圆曲线 E 用如下形式的等式来定义：

$$y^2 = x^3 + ax + b \tag{6-10}$$

其中 $a,b \in F_p$，且满足 $4a^3 + 27b^2 \neq 0 \pmod{p}$。一个有序偶$(x,y)$，若 x,y 满足式（6-10），则称(x,y)为椭圆曲线上的一个点。无穷远点，用 ∞ 表示（或者用大写的字母 O 表示），也在椭圆曲线上。椭圆曲线 E 上所有点组成的集合记作 $E(F_p)$。

例如 E 是定义于 F_7 上的椭圆曲线，其方程式为：

$$y^2 = x^3 + 2x + 4$$

那么 E 上的点为：

$$E(F_7) = \{\infty,(0,2),(0,5),(1,0),(2,3),(2,4),(3,3),(3,4),(6,1),(6,6)\}$$

（3）椭圆曲线密钥的生成

设 E 是定义于有限域 F_p 上的椭圆曲线。设 P 是 $E(F_p)$ 中的一个点，假设 P 有一个素数阶 n，那么由 P 生成的 $E(F_p)$ 的循环子群为：

$$\langle P \rangle = \{\infty, P, 2P, 3P, \cdots, (n-1)P\}$$

素数 p、椭圆曲线 E、点 P 及其阶 n 构成公共参数。私钥是随机从区间 $[1,n-1]$ 选取的一个整数 d，且其相应的公钥为 $Q = dP$。给出公共参数及公钥 Q，求解 d 的问题叫做椭圆曲线离散对数问题（Elliptic Curve Discrete Logarithm Problem，ECDLP）。

（4）简单的椭圆曲线加密体系

用椭圆曲线上的一个点 M 表示明文 m，Q 是接收者的公钥，k 为一随机选取的整数，然后通过计算 $M + kQ$ 来对 m 进行加密。发送者将 $C_1 = kP$ 和 $C_2 = M + kQ$ 传输给接收者，接收者利用私钥 d 计算：

$$dC_1 = d(kP) = k(dP) = kQ$$

然后就可以恢复 M 了，$M = C_2 - dC_1$。

（5）椭圆曲线上的运算

令 E 是定义在域 K 上的椭圆曲线。$E(K)$上的两个点相加定义为椭圆曲线上的加法操作。点集 $E(K)$及加法操作构成了一个阿贝尔群，无穷远点 ∞ 为加法幺元。这个群用来构造椭圆曲线加密系统。加法操作可以用几何图形来很好地解释。令 $P = (x_1, y_1)$，$Q = (x_2, y_2)$ 是椭圆曲线 E 上的两个不同的点。P 与 Q 的和 R 定义如下：通过 P 和 Q 作一条直线，与椭圆曲线交于第三点，那么 R 与第三点关于 x 轴对称，如图 6.6 所示。

P 的两倍 R 按如下规则定义：过点 P 作椭圆曲线 E 的切线，与椭圆曲线 E 交于第二点，那么 R 就是第二点关于 x 轴的对称点，如图 6.7 所示。

定义于 F_p（p 为大于 3 的素数）上的椭圆曲线 E 的运算规则还包括：

① 幺元：对任意的 $P \in E(F_p)$，$P + \infty = \infty + P = P$。

② 逆元：若 $P = (x, y) \in E(K)$，那么 $(x, y) + (x, -y) = \infty$。点 $(x, -y)$ 叫做点 $P(x, y)$ 的逆，记作 $-P$。实际上 $-P$ 也是椭圆曲线上的一个点。

③ 点相加。令 $P = (x_1, y_1) \in E(F_p)$，$Q = (x_2, y_2) \in E(F_p)$，且 $P \neq \pm Q$，那么 $P + Q = (x_3, y_3)$。其中：

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2, \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1$$

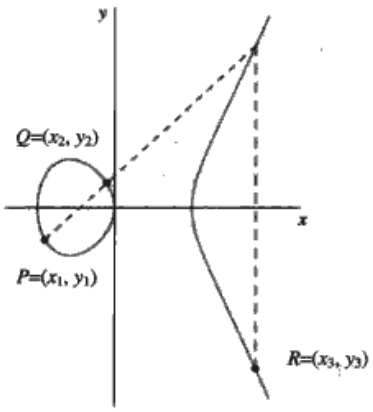


图 6.6　R=P+Q

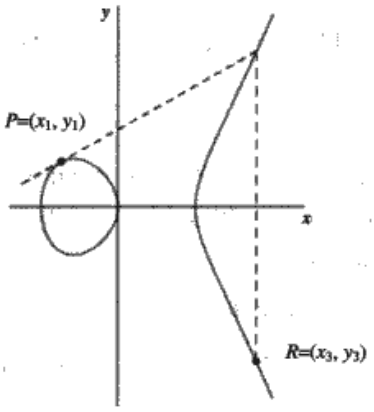


图 6.7　P+P=R

④ 点倍乘。令 $P=(x_1,y_1)\in E(F_p)$，且 $P\neq -P$，那么 $2P=(x_3,y_3)$。其中：

$$x_3=\left(\frac{3x_1^2+a}{2y_1}\right)^2-2x_1,\quad y_3=\left(\frac{3x_1^2+a}{2y_1}\right)(x_1-x_3)-y_1$$

例：（定义于有限域 F_{29} 上的椭圆曲线）令 $p=29,a=4,b=20$，则椭圆曲线方程为：

$$y^2=x^3+4x+20$$

椭圆曲线 E 上的点为：

∞	(2, 6)	(4, 19)	(8, 10)	(13, 23)	(16, 2)	(19, 16)	(27, 2)
(0, 7)	(2, 23)	(5, 7)	(8, 19)	(14, 6)	(16, 27)	(20, 3)	(27, 27)
(0, 22)	(3, 1)	(5, 22)	(10, 4)	(14, 23)	(17, 10)	(20, 26)	
(1, 5)	(3, 28)	(6, 12)	(10, 25)	(15, 2)	(17, 19)	(24, 7)	
(1, 24)	(4, 10)	(6, 17)	(13, 6)	(15, 27)	(19, 13)	(24, 22)	

例如：$(5,22)+(16,27)=(13,6),2(5,22)=(14,6)$

证明：令 $P=(5,22),Q=(16,27)$，则

$$x_3=\{[(27-22)(16-5)^{-1}]^2-5-16\}\bmod 29$$
$$=(5\times 11^{-1})^2-21\bmod 29$$
$$=(5\times 8)^2-21\bmod 29$$
$$=11^2-21\bmod 29$$
$$=100\bmod 29$$
$$=13$$
$$y_3=[(27-22)(16-5)^{-1}(5-13)-22]\bmod 29$$
$$=[5\times 8\times(-8)-22]\bmod 29$$
$$=(11\times 21-22)\bmod 29$$
$$=209\bmod 29$$
$$=6$$

上面的证明过程中涉及模 29 的乘法逆元的计算，如

$(16-5)^{-1}\bmod 29=11^{-1}\bmod 29=8$，即 $8\times 11\equiv 1\bmod 29$。

（6）椭圆曲线离散对数问题

离散对数问题的困难性是所有椭圆曲线密码体系安全性的必要保障。

椭圆曲线离散对数问题定义：给定一个定义于有限域 F_q 上的椭圆曲线 E，一个阶为 n 的点 P，且

$P\in E(F_q)$，点$Q\in\langle P\rangle$，求出使$Q=lP$成立的区间$[0,n-1]$内的整数l。整数l叫做Q以P为底的离散对数，记作$l=\log_P Q$。

用于密码学的椭圆曲线参数需要仔细地选取以抵抗对ECDLP的所有已知攻击。解决ECDLP的最基本的算法是穷举搜索，即计算点序列$P,2P,3P,4P,\cdots$直到结果为点Q。但是当n较大时，这种方法是不实际的。其他的攻击方法还有Pohlig-Hellman算法、Pollard's pho算法、index-calculus算法、Isomorphism算法。最好的已知普通意义上对ECDLP的攻击是Pohlig-Hellman和Pollard's pho算法。本书不是专门讨论椭圆曲线密码的，故不对这些攻击算法的实现进行介绍，感兴趣的读者请进一步参阅相关资料。需要注意的是，没有数学证明ECDLP问题是难于解决的，但是也没有人证明存在有效的算法解决ECDLP。

椭圆曲线在密码协议中可以用于数字签名、公钥加密和密钥交换协议。基于椭圆曲线的数字签名体系如ECDSA、EC-KCDSA，公钥加密体系如ECIES、PSEC，密钥交换协议如STS、ECMQV。

2. 椭圆曲线数字签名算法ECDSA

椭圆曲线数字签名算法ECDSA全称为The Elliptic Curve Digital Signature Algorithm，它类似于数字签名算法DSA。它是标准化最为广泛的基于椭圆曲线的签名体系，在ANSI X9.62、FIPS 186-2、IEEE 1363-2000及ISO/IEC 15946-2等标准中均有说明。

（1）算法描述

域参数$D=(q,FR,S,a,b,P,n,h)$由以下几部分组成：

① 域的阶q，即椭圆曲线上点的个数$\#E(F_q)$；

② 有限域F_q上的元素的域表示FR（field representation）；

③ 如果椭圆曲线是随机生成的，那么S代表种子；

④ 系数$a,b\in F_q$，其定义了F_q上的椭圆曲线E：$y^2=x^3+ax+b$；

⑤ 定义两个域元素x_P和y_P，其定义了点$P=(x_P,y_P)\in E(F_q)$，$P\neq\infty$，且P有一素数阶，P叫做基点（*Base Point*）；

⑥ P的阶n；

⑦ 余因子$h=\#E(F_q)/n$。

（2）ECDSA签名生成算法

在如下的表述中，H代表散列函数，通常为SHA-1。

输入：域参数$D=(q,FR,S,a,b,P,n,h)$，私钥d，待签名的消息m

输出：签名(r,s)

① 随机选择$k\in[1,n-1]$；

② 计算$kP=(x_1,y_1)$，并且将x_1转化为整数$\overline{x_1}$；

③ 计算$r=\overline{x_1}\bmod n$。如果$r=0$，则转到步骤①重新生成签名；

④ 计算$e=H(m)$。

⑤ 计算$s=k^{-1}(e+dr)\bmod n$。如果$s=0$，则转到步骤①重新生成签名；

⑥ 返回(r,s)。

（3）ECDSA签名验证算法

输入：域参数$D=(q,FR,S,a,b,P,n,h)$，公钥Q，待验证签名的消息m，签名(r,s)

输出：签名有效或者无效

① 验证签名r和s是否为区间$[1,n-1]$内的整数。如若不是，则返回“签名无效”；

② 计算$e=H(m)$；

③ 计算$w=s^{-1}\bmod n$；

④ 计算$u_1=ew\bmod n$及$u_2=rw\bmod n$；

⑤ 计算$X=u_1P+u_2Q$；

⑥ 如果 $X=\infty$，则返回“签名无效”；

⑦ 将 X 的 x 轴坐标 x_1 转化成整数 $\overline{x_1}$，并计算 $v=\overline{x_1} \bmod n$；

⑧ 如果 $v=r$，则返回“签名有效”，否则返回“签名无效”。

如果消息 m 的签名 (r,s) 是由合法的签名者生成的，那么 $s\equiv k^{-1}(e+dr)(\bmod n)$，即：

$k\equiv s^{-1}(e+dr)\equiv s^{-1}e+s^{-1}rd\equiv we+wrd\equiv u_1+u_2d \pmod n$

所以 $X=u_1P+u_2Q=(u_1+u_2d)P=kP$，因此只要 $v=r$ 即可验证签名是有效的。

3. ECDSA 在软件保护中的应用

目前在一些商业保护中使用了 ECDSA 算法来作为注册验证的主要部分，如 Safecast、Flexlm。ECDSA 的实现还涉及许多关于如何选取合适的椭圆曲线，选取安全的基点（Base Point）等一些知识，这些知识需要相关的数学知识，本书不做介绍，感兴趣的读者可以参考相关资料。

下面使用 miracl 给出一个实现如何使用 ECDSA 来作为软件的序列号生成及验证机制的例子。本例中的椭圆曲线参数采用 NIST 在“Recommended Elliptic Curves for Federal Government Use”中推荐的 GF(p) 上椭圆曲线之一 Curve P-192。

对应于域参数 $D=(q,FR,S,a,b,P,n,h)$，本例中的参数如下：

$q=0x$FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFF

$S=0x$3045ae6fc8422f64ed579528d38120eae12196d5

$a=-3$

$b=0x$64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1

$P=(P_x,P_y)$，其中：

$P_x=0x$188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012

$P_y=0x$07192b95ffc8da78631011ed6b24cdd573f977a11e794811

$n=0x$FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831

$h=1$

随机生成的 160 位的私钥 d 为：

$d=0x$97B27EA7BB8A6639601888965DC22BA3287E31D9

公钥 Q 为 $Q=(Q_x,Q_y)$，其中：

$Q_x=0x$9DD9DB8B71E342CA10652144B4FA3BFAFF854987CFBED260

$Q_y=0x$E5F30E201B0FF7F644BD6BA0313EF793A8CFA7D86CBD3DBF

（1）签名的生成

首先计算消息（用户名）的 160 位 SHA-1 散列值。

```
shs_init(&psh);
    for (i=0;i<dtLength;i++)
    {
        shs_process(&psh,(int)szName[i]);
    }
    shs_hash(&psh,szHash);
```

紧接着初始化大数，此时可以直接将椭圆曲线的参数之一 a 初始化。

```
...
big_a=mirvar(-3);
...
pt_P=epoint_init(); //初始化椭圆曲线上的点，必须要用 epoint_free 释放，否则可能会
pt_kP=epoint_init(); //造成堆栈溢出
...
bytes_to_big(20,szHash,big_e); //将 SHA-1 散列值转化为大数 e
cinstr(big_b,"64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1");
```

```
cinstr(big_py,"07192b95ffc8da78631011ed6b24cdd573f977a11e794811");
cinstr(big_n,"FFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831");
cinstr(big_d,"97B27EA7BB8A6639601888965DC22BA3287E31D9");  //私钥 d
```

然后先初始化椭圆曲线。

```
ecurve_init(big_a,big_b,big_q,MR_PROJECTIVE);
```

ecurve_init 的第 4 个参数有两个取值，定义如下：

```
#define MR_PROJECTIVE 0
#define MR_AFFINE 1
```

这两个参数主要用来指定系统内部椭圆曲线上点的坐标的表示方法，通常用 MR_PROJECTIVE，因为使用这种坐标表示方法时，速度会更快。

在生成签名的过程中，要保证随机数 *k*、签名 *r*、签名 *s* 都不能为 0，如果为 0，则需要重新生成签名，所以使用 while 循环。代码如下：

```
while(1)
    {
        decr(big_n,1,big_n); //首先将阶 n 减一
        bigrand(big_n,big_k);//生成随机数 k，范围为[0,n-1]，但当 k 为 0 时要重新生成
        incr(big_n,1,big_n);  //还原 n
        if (compare(big_k,big_zero)==0) // 随机数 k 不能为 0
        {
            continue;
        }
        else
        {
            epoint_set(big_px,big_py,1,pt_P);//设置基点 P 的坐标
            ecurve_mult(big_k,pt_P,pt_kP); // 计算点 kP
            epoint_get(pt_kP,big_r,big_r); // 得到点 kP 的 x 坐标
            divide(big_r,big_n,big_n);  //  签名 r 即为点 kP 的 x 坐标模阶 n 的值
            if (compare(big_r,big_zero)==0) // 签名 r 不能为 0
            {
                continue;
            }
            else
            {
            xgcd(big_k,big_n,big_k,big_k,big_k); // 求出 k 的模 n 逆元
            mad(big_d,big_r,big_e,big_n,big_n,big_temp); // temp=e+dr mod n
            mad(big_k,big_temp,big_temp,big_n,big_n,big_s); //s=k^-1. temp mod n
            if (compare(big_s,big_zero)==0) // 签名 s 不能为 0
            {
                continue;
            }
            else
            {
                break;
            }
        }
        break;
        }
    }
```

上面的代码段中使用了许多的 miracl 中的函数，读者可以参考 miracl 的手册查看其参数的详细说明。这里只强调函数 epoint_set，其函数原型定义如下：

```
BOOL epoint_set(x,y,lsb,p)
```

```
big x,y;
int lsb;
epoint* p
```

参数 x，y 分别为点 P 的 x 坐标和 y 坐标。第三个参数 lsb 涉及椭圆曲线上点压缩的概念。点压缩是实现椭圆曲线的一种技术，它可以降低椭圆曲线上点的存储空间。对于椭圆曲线 $y^2 = x^3 + ax + b$ 上的点(x,y)，给定一个 x，y 可能有两个值与之相对应，这可以从前面的例子中看出来。这是因为椭圆曲线上点的运算是模点的阶 n 的，当计算出来的 y 为负值时，需要重复加上 n，直到 y 为正，这样椭圆曲线上就存在两个点，具有相同的 x 坐标，其中一个是奇数，即其最低位（least significant bit）为 1，另一个是偶数，其最低位为 0。因此只要给出 x 坐标的值和 y 坐标的最低位的值 0 或 1 就可以确定一个点。

（2）签名的验证

验证的第一步是确保签名 r，s 必须位于区间[1,n-1]内，否则签名无效。

```
xgcd(big_s,big_n,big_s,big_s,big_s); //求出 s 的模 n 逆元
copy(big_s,big_w);
mad(big_e,big_w,big_w,big_n,big_n,big_u1); //u1=ew mod n
mad(big_r,big_w,big_w,big_n,big_n,big_u2); //u2=rw mod n
pt_P=epoint_init(); //初始化三个点
pt_Q=epoint_init();
pt_X=epoint_init();
ecurve_init(big_a,big_b,big_q,MR_PROJECTIVE);//初始化椭圆曲线
epoint_set(big_px,big_py,1,pt_P);  //设置点 P 的坐标
epoint_set(big_qx,big_qy,1,pt_Q);  //设置公钥点 Q 的坐标
ecurve_mult2(big_u1,pt_P,big_u2,pt_Q,pt_X); //计算 X=u1P+u2Q
```

紧接着要验证点 X 是否为无穷远点，如果是无穷远点，则签名无效。

```
if (point_at_infinity(pt_X)==TRUE)  //如果 X 为无穷远点，则签名无效
    {
        MessageBox(hWnd,"Not Valid Signature!","Verify failded",MB_OK);
    }
else
    {
        epoint_get(pt_X,big_v,big_v);  //得到点 X 的 x 坐标
        divide(big_v,big_n,big_v);  //模 n 得到 v
        if (compare(big_v,big_r)==FALSE)  //若 v=r，则签名有效，否则签名无效
        {
            MessageBox(hWnd,"Not Valid Signature!","Verify failded",MB_OK);
        }
        else
        {
            MessageBox(hWnd,"Valid Signature!","Verify Success",MB_OK);
        }
    }
```

更为详细的代码请参考光盘中的 ECDSASample。

6.4 其他算法

除了以上算法外，平时经常接触的还有 CRC32 算法、Base64 编码等算法。

6.4.1 CRC32 算法

CRC 全称为“Cyclic Redundancy Checksum”或者“Cyclic Redundancy Check”，是对数据的校验值，中文

名是“循环冗余校验码”，常用于校验数据的完整性。最常见的 CRC 是 CRC32，即数据校验值为 32 位。

首先利用 CRC32 多项式的值 04C11DB7h 或者 EDB88320h（将 CRC32 多项式的二进制表示的字符串逆向计算即可得到此值）生成一张 CRC32 表，其算法用 C 代码表示如下：

```
for(i=0;i<256;i++)
{
   crc=i;
   for(j=0;j<8;j++)
{
   if(crc&1)
      crc=(crc>>1)^0xEDB88320; // CRC32 多项式的值
   else
      crc>>=1;
}
crc32tbl[i]=crc;                    // crc32tbl 存储 CRC32 数据表
}
```

生成具有 256 个元素的 CRC32 表，如表 6-4 所示，完整的见光盘。

表 6-4 CRC32 数据表

0x00000000	0x77073096	0xee0e612c	0x990951ba	0x076dc419	0x706af48f	0xe963a535	0x9e6495a3
0x0edb8832	0x79dcb8a4	0xe0d5e91e	0x97d2d988	0x09b64c2b	0x7eb17cbd	0xe7b82d07	0x90bf1d91
0x1db71064	0x6ab020f2	0xf3b97148	0x84be41de	0x1adad47d	0x6ddde4eb	0xf4d4b551	0x83d385c7
……							
0xbdbdf21c	0xcabac28a	0x53b39330	0x24b4a3a6	0xbad03605	0xcdd70693	0x54de5729	0x23d967bf
0xb3667a2e	0xc4614ab8	0x5d681b02	0x2a6f2b94	0xb40bbe37	0xc30c8ea1	0x5a05df1b	0x2d02ef8d

然后就可以根据 CRC32 数据表来计算字符串或者文件的 CRC32 值了。算法如下所述：

```
dwCRC=0xFFFFFFFF;  // CRC 初值为-1，即 0xFFFFFFFF
for(i=0;i<Len;i++)    // Len 为要计算 CRC 的数据 Data 数组的长度（字节）
{
   dwCRC=crc32tbl[(dwCRC^Data[i])&0xFF]^(dwCRC>>8)
}
dwCRC=~dwCRC;    // 因为初值为-1,所以这里将 CRC 的值按位取反，方可得到正确的 CRC32 值
```

有关 CRC32 在文件完整性校验的应用请参考第 14 章。由于 CRC32 代码量小，容易理解，所以目前应用十分广泛。但同时，CRC32 并不是一个安全的加密算法。如果需要更安全的完整性校验算法，建议使用数字签名技术。

6.4.2 Base64 编码

Base64 编码是将二进制数据编码为可显示的字母和数字，用于传送图形、声音和传真等非文本数据。常用于 MIME 电子邮件格式中。其使用含有 65 个字符的 ASCII 字符集（第 65 个字符为“=”，用于对字符串的特殊处理过程），并用 6 个进制位表示一个可显示字符。表 6-5 列出的就是 Base64 编码表。

把数据编码为 Base64，将第一个字节放置于 24 位缓冲区的高 8 位，第二个字节放置于中间的 8 位，第三个字节放置于低 8 位。如果对少于 3 个字节的数据进行编码，相应的缓冲区位将被置 0。然后对 24 位缓冲区以 6 位为一组作为索引，高位优先，从字符串“ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/”中取出相应的元素作为输出。如果仅有一个或两个字节输入，那么只使用输出的两个或三个字符，其余的用“=”填充。

表 6-5 Base64 编码表

数值	编码	数值	编码	数值	编码	数值	编码
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

解码是编码的逆过程。首先得到 Base64 字符串的每个字符在 Base64 码表中的索引，然后将这些索引的二进制连接起来，重新以 8 位为一组进行分组，即可得到源码。

例如，表 6-6 里“转换前”栏的三个字节是原文，“转换后”栏的四个字节是 Base64 编码，其前两位均为 0。

表 6-6 Base64 值编码

源字符串（转换前）	a	b	c	
二进制	01100001	01100010	01100011	
编码后	00011000	00010110	00001001	00100011
十进制	24	22	9	35
Base64 值（转换后）	Y	W	J	j

除了 Base64 以外，还有 Base24、Base32 和 Base60。

Base24 码表：

BCDFGHJKMPQRTVWXY2346789（Windows 产品序列号就是使用这种编码）

Base32 码表：

ABCDEFGHIJKLMNOPQRSTUVWXYZ234567

Base60 码表：

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx

需要注意的是，在实际使用中，码表会和这些标准的码表不一样，但其编码原理是一样的。

6.5 常见的加密库接口及其识别

程序员在自己的程序中实现加密算法时，往往需要借助于一些加密算法库来实现。逆向分析时必须能识别出常见的加密算法库，识别加密算法库，最直接也是最精确的方法，便是掌握该算法库的使用方法。另外，也可使用 IDA 的 Flair 工具制作出算法库的 signature。

6.5.1 Miracl 大数运算库

Miracl 全称为 Multiprecision Integer and Rational Arithmetic C/C++ Library，即多精度整数和有理数算术运算 C/C++库。它是一个大数库，实现了设计使用大数的加密技术的最基本的函数。支持 RSA 公钥系统、Diffie-Hellman 密钥交换、DSA 数字签名系统及基于 GF(p)和 GF(2^m)的椭圆曲线加密系统。Miracl 提供了 C 和 C++两种接口，而且使用起来非常方便，速度相当令人满意，并且是开源的，所以用户可以根据需要扩充这一优秀的加密算法库。

官方网站为 www.shamus.ie。下载完最新版本后，参考 readme.txt 将其安装好。本节开发环境为 Visual C++ 6.0，使用 C 接口。为了方便，可以先将 miracl 目录下的 include 文件夹中的*.h 头文件统一复制到 VC6（VC98）的 include 目录中，并且将 miracl 编译好的静态链接库 ms32.lib 放到 VC6（VC98）的 lib 目录中，这样，在以后使用 miracl 的工程中，只需要添加下面两个语句即可调用 miracl 中的函数了。

```
#include <miracl/miracl.h>
#pragma comment(lib,"ms32.lib")
```

以光盘中的 MiraclStudy.exe 为例，介绍 miracl 中的大数格式及其函数的识别。

1. miracl 中的大数格式

miracl 中的大数是以 2^{32} 进制来表示大数的，即当数字大小超过 FFFFFFFFh 时，向高位进 1。在 miracl.h 中，miracl 是如下定义大数的：

```
struct bigtype
{
 mr_unsign32 len;
 mr_small *w;
};

typedef struct bigtype *big;
```

可见在程序中定义一个大数变量时，如 big bigN，那么实际上是定义了一个结构指针，结构的第一个元素是大数的长度，第二个元素是整型指针，指向存储大数数值的内存地址。

用 OllyDbg 打开光盘中的 MiraclStudy.exe，通过跟踪调试可以来到如下代码处：

```
0040122C  push   0040D03C
00401231  push   edi
00401232  call   00402BC0
```

40D03C 处指向的是一个字符串:

```
"5CF238B32E650ECA6B4D28256DFA26C9EE1B19ED1541B2AD9FE7446174A6D85"
```

EDI 是一地址，指向一大数变量，在内存中的格式如下：

```
003C6BF8  00 00 00 00 04 6C 3C 00 00 00 00 00 00 00 00 00  ....l<..........
003C6C08  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

第一个双字为 00000000h，表示大数的长度为 0，第二个双字为 003C6C04h，它指向大数的实际数值。在经过 402BC0 这个函数之后，3C6BF8 内存地址处的数据就变成了如下：

```
003C6BF8  08 00 00 00 04 6C 3C 00 00 00 00 00 85 6D 4A 17  ....l<.....mJ
003C6C08  46 74 FE D9 2A 1B 54 D1 9E B1 E1 9E 6C A2 DF 56  Ft  * T 褪贬濒⑦V
003C6C18  82 D2 B4 A6 EC 50 E6 32 8B 23 CF 05 00 00 00 00  倍处霎???....
```

第一个双字为 00000008h，表示大数长度为 8 个双字，即占用了 32 个字节。从 003C6C04 地址处开始才是大数的真正数值，可以看出 miracl 将字符串前面的字符作为低位，采用低位在前，高位在后的形式来存储大数。

2. 识别 miracl 中的函数

通过前面介绍的方法，可以通过做出 miracl 的 IDA sig 来识别函数。这里介绍另外一种方法：magic number。

在 miracl 的几乎每个函数的实现中，都有这样一条语句：

```
MR_IN(xx);
```

它是 miracl 的错误处理机制，用来表示 miracl 中的函数及退出代码。其定义在 miracl.h 中，通过它 miracl 可以知道是哪个函数出了错误。每个函数的“xx”都不同，如函数 mirvar 为 MR_IN(23)，函数 set_user_function 为 MR_IN(111)。通过反汇编，可以发现，这句代码通常都是如下形式：

```
mov    dword ptr [eax+ecx*4+20], yy
```

这里的 yy 是上面的 xx 的十六进制形式。根据 yy 就可以判断此函数究竟是 miracl 中的哪个函数了。笔者整理出了 miracl 5.01 版本中的这些函数的“magic number”，见光盘。

用 OllyDbg 打开光盘中的 MiraclStudy.exe，通过调试跟踪可以来到如下代码处：

```
004011E7  push    ebx                          ; ebx为0
004011E8  mov     dword ptr [esi+234], 10      ; mip->IOBASE=16
004011F2  call    00401730
{
   ...
   00401758  mov     dword ptr [eax+ecx*4+20], 17
   ...
}
```

通过上面的代码，17h 即十进制 23，可以判断出程序调用的是 miracl 中的 mirvar 函数。通过类似的方法，继续向下跟踪，可以分别找到如下的代码：

```
00403C7C  mov     dword ptr [ecx+eax*4+20], 8C
00402BEA  mov     dword ptr [eax+ecx*4+20], 4E
00402FEA  mov     dword ptr [eax+ecx*4+20], 12
00402F08  mov     dword ptr [eax+ecx*4+20], 4D
```

可以判断出程序接着分别调用了 bytes_to_big, cinstr, powmod, cotstr 四个函数。

6.5.2 FGInt

FGInt 全称为 Fast Gigantic Integers，是用于 Delphi 的一种可以实现常见的公钥加密系统的库。官方网站是：http://www.submanifold.be，读者可以从其下载最新版本的 FGInt。

1. FGInt 中的大数格式

FGInt 是以 2^{31} 进制来表示大数的，即当数值超过 7FFFFFFFh 时，便向高位进 1。FGInt 大数库在 FGInt.pas 中定义了大数格式，如下：

```
TFGInt = Record
     Sign : TSign;
     Number : Array Of LongWord;
End;
```

TFGInt 为一 Record 类型，第一个双字是符号位，表示大数的符号；第二个双字是一个 LongWord 数组，存储了大数的数值。

光盘中的 FGIntStudy/Project1.exe 是一个使用 FGInt 实现 ElGamal 签名的程序。

用 OllyDbg 打开 FGIntStudy.exe，通过跟踪调试可以来到如下代码处：

```
00456507  lea   edx, [ebp-30]
0045650A  mov   eax, 004565EC
0045650F  call  00454290
```

004565EC 处是一个十进制整数字符串：

```
004565EC  33 32 32 33 33 34 38 31 36 32 36 38 34 30 37 30  3223348162684070
004565FC  34 36 33 32 36 33 33 35 31 34 39 36 35 34 37 30  4632633514965470
0045660C  33 38 34 30 35 38 33 00 FF FF FF FF 26 00 00 00  3840583.□□□□&...
```

[ebp-30]内存地址处在调用 00454290 这个函数之前的数据为：

```
0012F568  99 10 01 58 00 00 00 00 00 00 00 00 00 00 00 00  ?·X............
```

在经过 454290 这个函数调用之后，内存数据变为：

```
0012F568  01 10 01 58 78 53 D6 00 00 00 00 00 00 00 00 00  ·►·XxS?........
```

第一个字节 01 表示的是大整数的符号为正，第二个双字，即 00D65378（注，以读者实际数据为准），指向的是存储大数内存地址。查看 D65378h 地址处的数据为：

```
00D65378  05 00 00 00 47 9D 4F 54 21 A9 67 0A 84 48 12 07  ♣...G渔T!⌐g.尻↕‼•
00D65388  79 6B FB 13 0F 00 00 00 16 00 00 00 94 42 D6 00  yk?
...•...攫?
```

其第一个双字表示大数的长度，占用了 5 个双字空间，从第二个双字开始才是大数的数值。

2. 识别 FGInt 的函数

第一种方法是先制作出 FGInt 的 IDA signature，然后应用 signature 即可识别 FGInt 中的函数。第二种方法是观察函数的参数个数，以及函数调用前后数据的变化来判断此函数的功能，然后对照 FGInt 的源码，对反汇编代码进行对比，来确定是何函数。第三种方法，对于 FGInt，Kanal（PEiD 的 Kcrypto ANALyzer 插件）可以识别出其函数，如本例，用 PEiD 0.94 中的 Kcrypto ANALyzer 插件可以识别出如下函数，如图 6.8 所示。

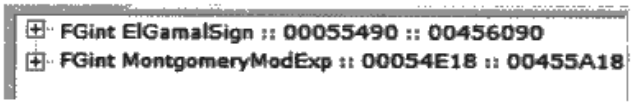


图 6.8 PEiD 插件识别出 FGInt

根据其地址即可找到该函数，这样对分析程序具有一定的帮助。

6.5.3 其他加密算法库介绍

1. freeLIP

freeLIP 最初设计是用于进行 RSA-129 挑战大数计算的大数库。其版本为 1.1。其接口的详细说明请参考 readme 及 lip.h。

freeLIP 采用 2^{30} 进制来表示大数。其速度不及 miracl。

2. Crypto++

Crypto++是一个实现了相当数量的加密算法的加密库，由华人戴伟开发。其官方网址为：http://www.cryptopp.com。Crypto++使用了 C++的高级语法，再加上文档比较少，所以不容易上手。

Crypto++的应用十分广泛。对于其函数的识别目前还没有很好的办法，常用的是做出其 IDA sig，然后应用 sig 去识别其函数。但是由于该库实现了相当数量的函数，所以在做 sig 时会有大量的冲突，需要花不少时间，但是可以通过编写脚本自动处理。而且由于不同的用户在编译该库时，采用的编译器设置不同，会造成 sig 不能完全识别其函数的情况。根据笔者的经验，识别其函数依靠于对该库的熟悉程度，依赖于

对加密算法的掌握程度，如果对加密算法的加密原理及其实现过程相当熟悉，那么不论程序采用了何种加密算法库，都可以轻而易举地识别出。

3．LibTomCrypt

LibTomCrypt 是另一款相当不错的加密算法库，其包括了常见的散列算法、对称算法及公钥加密系统。官方主页：http://libtomcrypt.org。其接口相当友好，非常适合 C 程序员使用。而且其代码书写相当规范，使用很方便。

其函数的识别，最有效的办法是，做出其 IDA sig，应用 sig 便可识别出其大部分的函数。

4．GMP

GMP 全称为 GNU Multiple Precision Arithmetic Library，其核心采用汇编实现，速度非常快，超过 miracl。常用其来实现大整数因子的分解，以提高速度。少见使用此库做软件保护。其官方主页为：http://www.swox.com/gmp。

5．OpenSSL

OpenSSL 主要用于网络安全，其中也包含了一些加密算法的实现。如对称算法中的 BlowFish、IDEA、DES、CAST，公钥中的 RSA、DSA，散列中的 MD5、RIPEMD、SHA 等。官方主页：http://www.openssl.org。

其函数的识别相对来说比较简单，一是通过反汇编，可以在使用了 OpenSSL 的程序中找到标记 OpenSSL 版本号的字符串，进而下载相应的版本进行编译，做出其 IDA sig，应用 sig 来识别。二是由于其实现的算法有限，故可以根据反汇编观察函数的参数的个数、参数的类型，然后根据这些信息到 OpennSSL 的 crypto 目录下的加密算法源代码中去找符合条件的函数，然后一一进行排除，甚至通过写几个测试程序，来检测函数的功能。

6．DCP 和 DEC

DCP 全称为 Delphi Cryptography Package，是用于 Delphi 的一个加密算法库。官方主页：http://www.cityinthesky.co.uk/cryptography.html。

DEC 全称为 Delphi Encryption Compendium，同样是用于 Delphi 的一种加密算法库。光盘中提供了其软件包。

由于这两个加密算法库实现了大部分常见的散列算法及对称算法，使用也十分方便，故而经常被 Delphi 程序员用于其软件的保护中。具体的使用请参考其文档。

对于 DCP 和 DEC 中函数的识别，最有效的方法是使用其 IDA sig。其次，取决于读者对加密算法的熟悉程序，也可以很容易地识别其加密函数。

7．Microsoft Crypto API

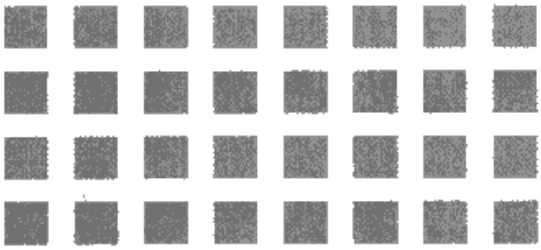
Crypto API 是微软为了方便程序员在软件中进行数字签名、数据加密的开发而提供的一套加密系统。接口也相当友好方便。其详细的说明请参考 msdn。

由于是微软提供的 API，故 IDA 及 Ollydbg、Softice 等调试软件都可以识别其函数。

8．NTL

NTL 是一个可以用于数论相关计算的库。官方主页：http://shoup.net/ntl/。它提供了非常友好的 C++接口，用于实现有符号的、算术整数的运算，以及向量、矩阵、基于有限域和整数的多项式运算。在密码学中，有限域的应用相当广泛，如 AES、twofish、ECC 等都涉及有限域。感兴趣的读者可以参考数论相关资料。

同样，只要做出该库的 IDA sig，即可达到识别其函数的目的。



第 4 篇　语言和平台篇

- 第 7 章　Delphi 程序
- 第 8 章　Visual Basic 程序
- 第 9 章　.Net 平台加解密

必须根据编译语言的特点进行加密与解密，才能达到比较理想的效果。现在所使用的语言无非是两种：一种是解释执行的语言，另一种就是编译后才能够执行的语言。解释语言的最大弱点之一就是能被反编译，因此其保护的重点应放在如何防止反编译上。

第7章 Delphi 程序

Delphi 与 C++ Builder 同为 Inprise 公司产品，共享集成开发界面（IDE），而且使用同一套 VCL（Visual Component Library）框架。事实上，Delphi 和 C++ Builder 除了使用的语言不同，其余几乎都相同，因此本节只以 Delphi 为例来讲述。

Delphi 和 C++ Builder 采用的是 VCL（可视化组件库）技术，这些环境使用特有的资源格式 RCDATA。RCDATA 资源中含有 Delphi 的窗体（Forms），所有对窗口设计的信息都包含在内。当一个典型的 Delphi 程序开始运行时，其初始化代码建立这种窗体，并从资源中读取所需要的信息。DFM 文件是 Delphi 编码的二进制文件。在编译时，这个文件就以源程序的形式存储于 RCDATA 资源中。这样，利用 eXeScope 等资源编辑工具，可以对它查看、修改，并且可改变程序的某些运行方式。

7.1 DeDe 反编译器

Delphi/C++ Builder 采用控件拖放的方式来完成界面的设计，并和事件联系起来。而这些信息以资源（RCDATA）的方式存放于可执行文件中。DeDe 便利用这个原理进行反编译，获取相关信息，将界面与事件联系关系还原，但事件的汇编代码不能还原。DeDe 公开了源代码，感兴趣的读者可以研究一下。

1．主要功能

用 DeDe 可以查看 Delphi 程序窗体的属性，可以查看按钮对应的事件，并将事件代码反汇编出来，其能识别出 Delphi 库函数，具有良好的可读性。另外，还可以把事件输出到 map 文件中供其他工具使用。

2．配置

（1）DSF 文件

① DSF 文件的含义

DSF 文件内容来自不同版本 BPL 库文件的输出符号表。DeDe 反汇编引擎使用这些符号表对生成的 ASM 代码文件添加类成员方法调用的注释，这非常类似于 IDA Pro 的 FLIRT 技术。如果没有加载任何 BPL 符号表文件，对 BPL 类的调用就无法以注释的格式说明。

② 加载 DSF 文件

经由“File/Load Symbol File”菜单，就可加载所需要的 DSF 文件。程序若能正确识别出相应版本的 Delphi 程序，会自动加载 DSF 文件。若希望每次启动 DeDe 的同时自动加载若干 DSF 文件，选中“Options/Configuration”菜单，在 Symbols 选项卡中就可完成本项工作。如果想查看包含在某个特定 DSF 文件中的输出符号表，选择“Options/Symbols”。

③ 为何需要创建 DSF 文件

处理使用自定义构件（即不是 Delphi 安装的构件）的程序时，如果有这些自定义构件的 BPL，并且为

它们创建了 DSF 文件，那么 DeDe 将会注释所有对这些自定义构件的调用。创建 DSF 的速度也是很快的。

（2）DOI 文件

DOI 意思是 Delphi Class Offset Information（Delphi 类偏移信息），该技术使用偏移信息识别类成员：方法和域（实例变量，属性）。DOI 文件包含用于识别的必要数据。DeDe 仿真指令的执行来查找使用这些偏移的引用所在。例如，在继承自 TForm 类的任何子类的偏移 0xCC 处，代表指向 ShowModal 方法的指针。当遇到类似 call [reg + $00CC]的调用时，仿真器就知道寄存器包含的对象是引用 TForm.ShowModal 方法的 TForm 子类。DOI 文件应该存放在 DSF 文件夹内。

下面是一个生成的带有 DOI 帮助信息的简单示例。

```
* Reference to control LogMemo : TMemo
004E4E7C  8B80F4020000      mov  eax, [eax+$02F4]
* Reference to field TMemo.Lines : TStrings
004E4E82  8B8004020000      mov  eax, [eax+$0204]
* Possible String Reference to: 'Loading Export Names ...'
004E4E88  BA0C584E00        mov  dx, $004E580C
004E4E8D  8B08              mov  ecx, [eax]
* Reference to method TStrings.Add(string)
004E4E8F  FF5134            call dword ptr [ecx+$34]
```

使用 DOI 文件，只需复制*.DOI 文件到 DSF 文件夹即可。DOI 数据会自动插入到生成的代码文件中。

（3）字符串参考的含义

在 DeDe 中，如果处理含有非英文字符串的程序，则选择“Option/Configuration”菜单，在 References 选项卡中可以设置 DeDe 反编译引擎查找字符串参考时使用的字符集。

注意：如果使用全部的字符集合#32～#255，则可能得到残缺的字符串参考。Delphi 程序一般不用 Unicode 字符串，这也是该选项没有包括在字符串参考配置中的原因。

3. 基本操作

DeDe 安装很简单。安装好后直接执行主程序，出现如图 7.1 所示的主界面。单击 按钮打开光盘映像文件中的 DE_Delphi 文件，然后单击“Process”按钮进行反编译。DeDe 先将被分析的文件装载到内存中，再进行反编译，因此对一些压缩加壳的程序也能反编译。

- CLasses Info：显示程序中使用的类信息；
- Units Info：显示程序中使用的单元信息；
- Forms：显示程序中的窗体信息，这部分可以用资源编辑工具修改；
- Procedures：显示程序的过程信息；
- Project：可将当前项目保存；
- Exports：导出符号文件。

图 7.1 DeDe 界面

在此显示了 Events（事件）和 Controls（控件）两方面的内容（见图 7.2）。Button1Click 事件对应的是“确定”按钮，双击可打开代码窗口。该窗口显示当前事件对应的汇编代码，右边控制条显示全局变量和局变量。双击某个表达式，可以加注释。在跳转指令、CALL 指令上双击可跳到相应代码上。

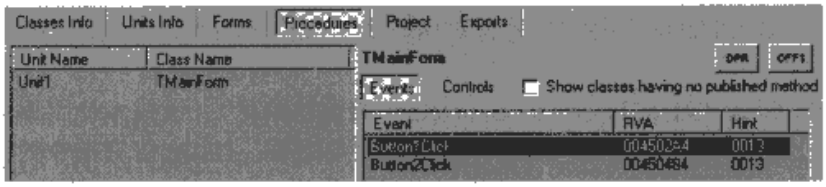


图 7.2　查看事件按钮

设目标实例 DE_Delphi 的用户名为 Name[]，具体代码如下：

```
* Reference to control TMainForm.Edit1 : TEdit     ;用户名框控件
004502C6    mov     eax, [esi+$02F8]
* Reference to: controls.TControl.GetText(TControl):System.String;
004502CC    call    0042F4F8
004502D1    cmp     dword ptr [ebp-$0C],+$00 ;判断是否输入字符
004502D5    jnz     004502F5
......
* Reference to: controls.TControl.GetText(TControl):System.String;
004502FE    call    0042F4F8
00450303    mov     eax, [ebp-$10]           ;指向用户名
* Reference to: system.@LStrLen:Integer;
00450306    call    004044C4
0045030B    cmp     eax, +$04                ;判断是否输入了 4 个字符
0045030E    jnl     0045032E
......
* Reference to: controls.TControl.GetText(TControl):System.String;
00450345    call    0042F4F8
0045034A    mov     eax, [ebp-$14] 指向 Name[]
* Reference to: system.@LStrLen:Integer;
0045034D    call    004044C4                 ;得到用户名长度
00450352    mov     ebx, eax                 ;将长度放到 ebx 中作为计数器
00450354    test    ebx, ebx
00450356    jle     00450381
00450358    mov     edi, $00000001           ;edi=1
0045035D    /mov    eax, [ebp-$08]           ;指向 Name[]
00450360    |movzx  eax, byte ptr[eax+edi-$01]
00450365    |lea    ecx, [ebp-$18]
00450368    |mov    edx, $00000002
*Referenceto: sysutils.IntToHex(System.Integer;System.Integer)
0045036D    |call   00408310                 ;inttohex(ord(Name[i]),2)
00450372    |mov    edx, [ebp-$18]           ;[ebp-$18]指向 Name[i]16 进制
00450375    |lea    eax, [ebp-$04]           ;[ebp-$04]变量是指向 Sn 指针的指针
* Reference to: system.@LStrCat;
00450378    |call       004044CC             ;将两个串连起来，设为 Sn
0045037D    |inc    edi                      ;i+1
0045037E    |dec    ebx                      ;计数器减 1
0045037F    \jnz    0045035D                 ;循环
......
* Reference to: controls.TControl.GetText(TControl):System.String;
0045038A    call    0042F4F8                 ;取输入的序列号
0045038F    mov     eax, [ebp-$1C]           ;[ebp-$1C]是输入的序列号指针
00450392    mov     edx, [ebp-$04]           ;[ebp-$04]中存放的是正确的序列号
* Reference to: system.@LStrCmp;
00450395    call    00404608                 ;比较两个临时变量
0045039A    jnz     004503B7
```

由于 DeDe 的符号识别技术，使得上述代码可读性非常容易理解。

在图 7.2 中有两个按钮值得注意：

- DPR 按钮：是反汇编项目文件.dpr，该文件控制或记录程序中的所有文件。
- OFFS 按钮：反汇编指定地址的代码。

7.2 按钮事件代码

动态调试Delphi程序时，会发现用一般的API函数不能拦截文本框（Edit）的数据。因为Delphi通过向Edit发送WM_GETTEXT消息来获得Text的内容（直接调用WndProc，而没有使用消息函数），整个过程没有调用过任何Win32 API函数，所以常用的Hmemcpy, GetDlgItemTextA,GetWindowTextA等断点失效是当然的。那么，如何才能将用户输入的字符串拷贝到软件的缓冲区中时使调试器中断呢？可以利用DeDe来辅助，Delphi的按钮和事件对应，利用DeDe反编译得到按钮的事件地址，对此地址设断拦截。本节讨论一下如何手工定位按钮的事件代码。用eXeScope打开DE_Delphi文件，查看其RCDATA资源，找到“确定”按钮的资源。数据如下：

```
object Button1: TButton                    ;“确定”按钮
   Left = 32
   Top = 112
   Width = 75                              ;按钮的大小尺寸
   Height = 20
   Caption = #30830#23450                  ;按钮的名字
   ……
   TabOrder = 0
   OnClick = Button1Click                  ;按下按钮后调用的例程
end
```

例程（用来响应用户界面之窗体的元素）的地址是按名字绑定的。只要知道这些名字，就可知道所需要的地址，因此搜索该例程名称就能找到调用的地址。用十六进制工具打开DE_Delphi，搜索“Button1Click”，找到两处。第二处是资源本身，第一处是在地址表（Address Table）中，具体见图7.3。

图7.3 查看OnClick事件地址

现在，观察名字前那些神奇的数字，有一个字节“0C”指出了“Button1Click”的长度（12个字符）。在此字节前面的就是其事件调用地址4502A4h，然后就可用此地址设置断点了。

该方法不仅适合寻找按钮的调用地址，也适合寻找FormCreate等过程事件的地址。找到后，可以用图7.2中的“OFFS”按钮对这些指定地址进行反汇编。

7.3 模块初始化与结束化

Delphi采用的是VCL体系，它的类是由编译期间决定的，编译完成后即固定在程序中。当程序运行时，首先会将这些类完成初始化，然后开始执行代码。

用OllyDbg打开实例DE_Delphi_6.exe，以分析程序是如何初始化的。下面这段代码是Delphi 6编译器为可执行程序EXE生成的入口代码：

```
0045065C   push    ebp                              ;程序入口
0045065D   mov     ebp, esp
0045065F   add     esp, 0FFFFFFF0h
00450662   mov     eax, offset dword_4504EC         ;传入初始化表地址
00450667   call    @@InitExe                        ;调用模块初始化函数
```

```
0045066C   mov    eax, ds:off_451E24
00450671   mov    eax, [eax]
00450673   call   0044EBB4
00450678   mov    ecx, ds:off_451CB4
0045067E   mov    eax, ds:off_451E24
00450683   mov    eax, [eax]
00450685   mov    edx, off_4500A0
0045068B   call   @Forms@TApplication@CreateForm$qqrp17System@TMetaClasspv
00450690   mov    eax, ds:off_451E24
00450695   mov    eax, [eax]
00450697   call   @Forms@TApplication@Run$qqrv
0045069C   call   @System@@Halt0$qqrv          ;退出时再调用一次初始化表
```

注意这句“00450662 mov eax, 4504EC”，4504EC 负责传入单元的初始化表地址，其指向的是初始化表的个数与地址，如图 7.4 所示。初始化表的个数是 2Dh，地址是 4504F4h。

```
004504EC 2D 00 00 00 F4 04 45 00 EC 65 40 00 BC 65 40 00 -...?E.?e@.?e@.
004504FC 40 64 40 00 EC 63 40 00 64 66 40 00 34 66 40 00 @d@.?c@.df@.4f@.
0045050C C0 70 40 00 90 70 40 00 F8 70 40 00 C8 70 40 00 ?p@.?p@.?p@.?p@.
0045051C F0 08 41 00 C0 08 41 00 C0 73 40 00 90 73 40 00 ?A.?A.?s@.?s@.
```

图 7.4　初始化表大小与地址

InitExe()处理初始化表的相关代码位于 Delphi 安装目录下的 SysInit.pas ,System.pas 里，代码如下：

```
procedure _InitExe(InitTable: Pointer);
begin
  TlsIndex := 0;
  HInstance := GetModuleHandle(nil);
  Module.Instance := HInstance;
  Module.CodeInstance := 0;
  Module.DataInstance := 0;
  InitializeModule;
  _StartExe(InitTable, @Module);
end;
```

相应的汇编代码如下：

```
00406578  push   ebx
00406579  mov    ebx, eax
0040657B  xor    eax, eax
0040657D  mov    dword ptr [45270C], eax
00406582  push   0                                ; /pModule = NULL
00406584  call   <jmp.&kernel32.GetModuleHandleA> ; \GetModuleHandleA
00406589  mov    dword ptr [452714], eax
0040658E  mov    eax, dword ptr [452714]
00406593  mov    dword ptr [451090], eax
00406598  xor    eax, eax
0040659A  mov    dword ptr [451094], eax
0040659F  xor    eax, eax
004065A1  mov    dword ptr [451098], eax
004065A6  call   InitializeModule
004065AB  mov    edx, 0045108C
004065B0  mov    eax, ebx
004065B2  call   @StartExe
004065B7  pop    ebx
004065B8  retn
```

跟进 StartExe()函数，代码如下：

```
procedure      _StartExe(InitTable: PackageInfo; Module: PLibModule);
begin
  RaiseExceptionProc := @RaiseException;
```

```
  RTLUnwindProc := @RTLUnwind;
{$ENDIF}
  InitContext.InitTable := InitTable;
  InitContext.InitCount := 0;
  InitContext.Module := Module;
  MainInstance := Module.Instance;
{$IFNDEF PC_MAPPED_EXCEPTIONS}
  SetExceptionHandler;
{$ENDIF}
  IsLibrary := False;
  InitUnits;
end;
```

相应的汇编代码如下：

```
00403F78  mov    dword ptr [452014], <jmp.&kernel32.RaiseException>
00403F82  mov    dword ptr [452018], <jmp.&kernel32.RtlUnwind>
00403F8C  mov    dword ptr [452638], eax
00403F91  xor    eax, eax
00403F93  mov    dword ptr [45263C], eax
00403F98  mov    dword ptr [452640], edx
00403F9E  mov    eax, dword ptr [edx+4]
00403FA1  mov    dword ptr [45202C], eax
00403FA6  call   00403E70
00403FAB  mov    byte ptr [452034], 0
00403FB2  call   00403F18                    ; InitUnits
00403FB7  retn
```

InitUnits就是调用初始化表的代码，其主要的汇编代码如下：

```
    ......
    00403F40   mov    eax, dword ptr [edi+ebx*8] ;EDI指向初始化表地址
    00403F43   inc    ebx
    00403F44   mov    dword ptr [45263C], ebx
    00403F4A   test   eax, eax
    00403F4C   je     short 00403F50
    00403F4E   call   eax                        ;调用初始化表
    00403F50   cmp    esi, ebx
    00403F52   jg     short 00403F40
    ......
```

从上面代码分析可以得知，Delphi程序加载时，程序从前往后间隔调用初始化表，退出是从后往前间隔调用的。如图7.5所示，程序运行时，会间隔地调用4065EC, 406440, 406664…。退出时，从后往前间隔调用，即从地址00450658开始向前，如4504C4, 45048C, 450000…。所以图7.4中的初始化表个数2Dh是整个初始化表的一半，整个初始化表个数是2Dh×2。

```
004504F4  EC 65 40 00 BC 65 40 00 40 64 40 00 EC 63 40 00
00450504  54 66 40 00 34 66 40 00 00 70 40 00 90 70 40 00
00450514  F8 70 40 00 C8 70 40 00 F0 0B 41 00 C0 0B 41 00
......    ......
00450644  BC 00 45 00 00 00 45 00 BC 04 45 00 8C 04 45 00
00450654  00 00 00 00 C4 04 45 00 55 8B EC 83 C4 F0 B8 EC
```

图7.5 Delphi的初始化表

程序的入口点也就在初始化表后，在后面章节脱壳技术中，可以用这种方法确定入口点。同时一些壳（如ASProtect）加密了初始化表，需要还原。

第8章 Visual Basic 程序

VB3 和 VB4 是典型的解释语言，它们都有相应的反编译器存在。VB5 和 VB6 不再是单纯的解释程序，在继续保留 P-code 编译的同时也引入了 Native 编译方式，使得生成本地二进制代码成为可能。关于 VB.net，其生成的 EXE 程序完全可用.Net 反编译工具完成，此处不再介绍。

本章仅简单地介绍 Visual Basic 程序一般的调试方法，有关 Visual Basic 6 逆向工程更多的知识，请参考《软件加密技术内幕》此书周文雄撰写的“Visual Basic 6 逆向工程”。

8.1 基础知识

8.1.1 字符编码方式

Visual Basic 32 位版本的字符串处理采用 Unicode 编码或称宽字符（Widechars）。也就是说，字符串在 Visual Basic 内部是以 Unicode 格式存放的。在 Unicode 中，所有的字符都是 16 位的，比如 7 位 ASCII 码都被扩充为 16 位（注意：高位扩充的是零，即 0）。

Visual Basic 中的字符串因为版本不同，字符串的格式也不同。16 位 VB 4.0 和以前版本的 VB 字符串格式很复杂，使用连续指针的方式，源指针指向一个由字符串长度和字符串首地址指针组合而成的结构，其中的字符串首地址指针再指向字符串。VB 5.0 及以后版本则是单指针方式，源指针指向一个复合字符串，此字符串开始的四个字节组成一个长整数，以指示此字符串的长度，其后是字符串，而且这个字符串除了应有的字符外，结尾还有两个“00”，而最后的这两个“00”不计算在字符串长度之内。一个英文的 VB 字符串“pediy”在内存中（编译后）应该如图 8.1 所示。

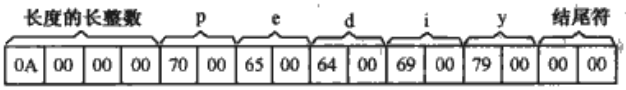


图 8.1 VB 字符串在内存中的形式

8.1.2 编译模式

编译器的编译技术可以分为 Native-Compile（自然编译）与 Pcode-Compile（伪编译）两种。自然编译是编译器将高级语言转换为汇编代码，并经链接生成 EXE 程序的过程。伪编译是编译器将高级语言转换为某种编码后，将能解释、执行此编码的一段程序一同链接，生成 EXE 程序。

所谓伪代码，其基本工作原理是编译器先把执行程序编译成比 80x86 机器码紧凑得多的中间代码形式，运行时，其基于堆栈的引擎将特殊操作码翻译成 CPU 上的操作指令。依靠 P-code 编译技术，使得编程语言不依赖于机器或系统平台成为可能。

从 Visual Basic 版本 1.0 到 4.0 都只生成 P-code。P-code 的最大问题就是执行速度比本地编译的程序慢，优点是使得程序不依赖于硬件或操作系统成为可能。从 1997 年 Visual Basic 5 问世起，微软公司在编译过程添加了允许生成本地机器码的选项。虽然目前 P-code 编译形式大多见于 Visual Basic，但 Java、PowerBuilder 等的编译实质也是一样。

因为 VB3 和 VB4 都是使用伪代码编译形式，也因为伪代码编译的代码实际只是“变形的源代码”，所以只要能理解其对应机制，就能做出反编译器来。例如 Dodi's VB 3/4 Disassembler。同样，Java、PowerBuilder 也有反编译器。

8.2 自然编译（Native）

自然编译是 VB 源代码生成汇编代码并链接的过程。因为是生成汇编代码，所以对其解读完全遵照一般的反汇编常规。

8.2.1 相关 VB 函数

掌握一些 VB 常用内部函数，将有助于 VB 程序的分析。以 VB 6.0 为例，实际上的数据计算和比较是在 msvbvm60.dll 及 oleaut32.dll 中完成的。如果要深入研究 VB，不妨用 IDA、W32Dasm 反汇编 msvbvm60.dll 及 oleaut32.dll，理解其函数的意义。msvbvm60.dll 中包含的函数名总是用__vba 或 rtc 开头，而 oleaut32.dll 函数名以 var 开头。对于函数的作用，一般可以按照函数名从右往左理解。例如：__vbaI2Str，表示字符串转换为整数。

通过研究 OLEAUT.H、OAIDL.h 文件可以了解 VB 函数中各类变量，并进而根据函数名称猜出大部分函数的用途，见表 8-1。

表 8-1　相关函数

函 数 名	含　义
MultiByteToWideChar	将 ANSI 字符串转换成 Unicode 字符（Win32 API 函数）
WideCharToMultiByte	将 Wide-Character（Unicode）字符转换成 ANSI 字符（Win32 API 函数）
rtcR8ValFromBstr	把字符串转换成浮点数
__vbaStrCmp	比较字符串
__vbaStrComp	比较字符串
__vbaStrCopy	复制字符串
__vbaStrMove	移动字符串
__vbaVarTstNe	进行变量比较
__vbaVarTstEq	进行变量比较
StrConv	转换字符
rtcMsgBox	显示对话框
rtcGetPresentDate	取得当前日期
VarBstrCmp（Oleaut32.dll）	比较字符串，例：bp Oleaut32.dll! VarBstrCmp
VarCyCmp（Oleaut32.dll）	比较字符

注意：这些函数前的下划线“__”是由两根短线“_”组成的。

8.2.2 VB 程序比较方式

本节从汇编角度来分析一下 VB 字符串的比较模式，以熟悉 VB 的一些处理方式。

1. 字符串（String）比较

在这种方法里，正确密码串（如“Correct Password”）和输入的密码串（如“Entered Password”）比较。字符串是由相邻的字符按顺序排列组成的。一个字符串包括字母、数字、空格和标点符号。

下面是一个范例代码：

```
If "Correct Password" = "Entered Password" then          ; 直接比较两个字符串
    GoTo Correct Message
Else
    GoTo Wrong Message
End if
```

这种方式是明码比较。如果程序用这种函数比较，很容易被破解。可用到的断点： __vbaStrComp 或 __vbaStrCmp。

运行光盘映像文件中用 VB 6 编译好的实例 string.exe，可以用__vbaStrCmp 或 rtcMsgBox 设置断点。汇编形式如下：

```
0040227B   push    eax                              ;输入的假序列号
0040227C   push    00401BD8                         ;查看 401BD8 地址即可看到真序列号
00402281   call    [<&MSVBVM60.__vbaStrCmp>]        ;比较函数
{
    660E8A03   push    dword ptr [esp+8]
    660E8A07   push    dword ptr [esp+8]
    660E8A0B   push    0                            ;0 为二进制比较（默认），1 为文本方式比较
    660E8A0D   call    __vbaStrComp
    660E8A12   retn    8
}
00402287   mov     esi, eax                         ;相等则返回 0 到 eax
```

OllyDbg 断下后，在数据窗口查看 401BD8 处的数据，如图 8.2 所示。其以宽字符显示，转换过来就是“pediy”，这就是真的序列号。

图 8.2 在数据窗口查看字符

2. 变量（Variant）比较

在这种方法中，两个变量（变量数据类型）互相比较。变量数据类型是一种特殊数据类型，包括数字、字符串、日期数据及一些用户定义的类型。这种类型储存的数字长度是 16 字节，而字符长度是 22 字节（加上字符串长度）。

下面是一个范例代码：

```
Dim correct As Variant, entered As Variant    ;定义 correct 和 entered 作为变量
correct = pediy                               ;设置 correct 放置“pediy”
entered = Text1.Text                          ;设置 entered 为输入的密码
If correct = entered Then                     ;用变量方法比较
    GoTo Correct Message
Else
    GoTo Wrong Message
```

在此方法中，字符串比较中的两个 API 断点将不起作用，因为程序不再用__vbaStrCmp 等函数比较字符串。其比较的方式是依赖于变量是否相等，即用__vbaVarTstEq 函数比较变量。

用 VB 6 编译好的程序为 Variant.exe。比较代码汇编形式如下：

```
004024BF   push    ecx                           ;OllyDbg 命令行里执行 D [ecx+8]
004024C0   push    edx                           ;OllyDbg 命令行里执行 D [edx+8]
```

```
004024C1   call    [<&MSVBVM60.__vbaVarTstEq>] ;变量比较函数
004024C7   test    ax, ax                      ;相等则返回-1
```

在 OllyDbg 的命令行里，执行 D [edx+8]，将显示出正确的序列号，其以宽字符显示，如图 8.3 所示。

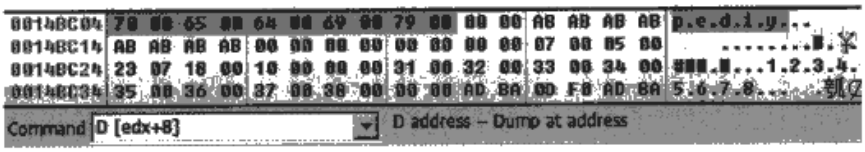


图 8.3　在数据窗口查看参数

注意：此时直接查看 ecx 和 edx 什么都看不到，这是因为 VB 用了一些特殊的寻址方式。牢记此时用 ecx+8 和 edx+8 查看内存，将显示真假序列号的偏移地址。

3. 字节（Byte）比较

这种方法是两个字节数据进行比较。Byte 变量存储为单精度型、无符号整型、8 位（1 个字节）的数值形式，范围在 0 至 255 之间。

下面是一个范例代码：

```
Dim correct As Byte, entered As Byte          ;定义 correct 和 entered 为字节型
correct =12                                   ;设置 correct 为“12”
entered = Text1.Text                          ;设置 entered 为输入的密码
If correct = entered Then                     ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

对这种类型没有专门的断点函数，因为其数据比较是在主程序里，而不是在 Visual Basic 运行库中。用 VB 6 编译好的程序为 Byte.exe。代码用十六进制数据比较，汇编形式如下：

```
004023D0   cmp     bl, al                     ;al=0xc，转换成十进制是 12
……
004023E3   jnz     short 00402444
```

4. 整型（Integer）比较

这种方法是用两个整型数据进行彼此比较。Integer 变量存储为 16 位（2 个字节）的数值形式，其范围为–32 768 到 32 767 之间。

下面是一个范例代码：

```
Dim correct As Integer, entered As Integer    ;定义 correct 和 entered 为整型
correct = 1212                                ;设置 correct 为“1212 ”
entered = Text1.Text                          ;设置 entered”为输入的密码
If correct = entered Then                     ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

对这种类型没有专门的断点函数，因为其数据比较是在主程序里，而不是在 Visual Basic 运行库中。用 VB 6 编译好的程序为 Integer.exe。代码用十六进制数据比较，汇编形式如下：

```
004023B4   cmp     si, 4BC                        ;1212 的十六进制就是 04BCh
……
004023C5   jnz     short 00402426
```

5. 长整型（Long）比较

这也是一种常见的方法，两个变量（长整型）互相比较。Long（长整型）变量存储为 32 位（4 个字节）有符号的数值形式，其范围从-2 147 483 648 到 2 147 483 647。因此该方法只能用来比较数字。

下面是一个范例代码：

```
Dim correct As Long, entered As Long          ;定义 correct 和 entered 作为长整型
correct = 1872333                             ;设置“1872333”为正确密码
entered = Text1.Text                          ;设置 entered 为输入的密码
If entered = correct Then                     ;用长整型方法比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

用 VB 6 编译好的程序为 Long.exe。代码用十六进制形式比较，汇编形式如下：

```
0040250D   cmp     esi, 1C91CD                ;1872333 的十六进制就是 1C91CDh
......
0040251F   jnz     short 00402580             ;相等则不跳转
```

6. 单精度实数（Single）比较

这种方法用两个单精度实数数据比较。Single（单精度浮点型）变量存储为 32 位 IEEE（4 个字节）浮点数值的形式，它的范围在负数的时候是从-3.402823E38 到-1.401298E-45，而在正数的时候是从 1.401298E-45 到 3.402823E38。因此这种方法仅仅能比较数字，但也可将姓名和序列号转换成实数进行比较。查看相关数据时用 DL 命令显示浮点型（Long/Real）。

下面是一个范例代码：

```
Dim correct As Single, entered As Single      ;定义 correct 和 entered 为单精度型
correct = 1872333                             ;设置 correct 为“1872333”
entered = Text1.Text                          ;设置 entered 为输入的密码
If correct = entered Then                     ;比较
     GoTo Correct Message
Else
     GoTo Wrong Message
End If
```

用 VB 6 编译好的程序为 Single.exe。代码用单精度实数数据格式比较，汇编形式如下：

```
004023B3   cmp     eax, 49E48E68              ; 1872333 的单精度浮点型是 49E48E68h
......
004023C9   jnz     short 0040242A             ;相等则不跳转
```

7. 双精度（Double）比较

这种方法用两个双精度数据比较。Double（双精度浮点型）变量存储为 64 位 IEEE（8 个字节）浮点数值的形式，它的范围在负数的时候是从-1.79769313486231E308 到-4.94065645841247E-324，而正数的时候是从 4.94065645841247E-324 到 1.79769313486232E308。与单精度类似，只能比较数字。

下面是一个范例代码：

```
Dim correct As Double, entered As Double      ;定义 correct 和 entered 为双精度型
correct = 12345678901234                      ;设置 correct 为 "12345678901234"
entered = Text1.Text                          ;设置 entered 为输入的密码
If correct = entered Then                     ;比较
     GoTo Correct Message
Else
```

```
    GoTo Wrong Message
End If
```

用 VB 6 编译好的程序为 Double.exe。由于比较的数是 64 位，因此分两次比较。第一次是比较双精度的前 32 位“9C5FE400”，第二次是比较双精度的后 32 位“42A674E7”。此例中的汇编形式如下：

```
00402952   push    eax                          ;指向输入的字符串
00402953   call    [<&MSVBVM60.__vbaR8Str>]     ;转换 string 为 Integer/Real 到 st(0)
00402959   fstp    qword ptr [ebp-24]           ;[ebp-24]=st(0)，处理输入的序列号
0040295C   lea     ecx, dword ptr [ebp-28]
0040295F   call    [<&MSVBVM60.__vbaFreeStr>]
00402965   lea     ecx, dword ptr [ebp-2C]
00402968   call    [<&MSVBVM60.__vbaFreeObj>]
0040296E   cmp     dword ptr [ebp-24], 9C5FE400    ;比较双精度的前 32 位
00402975   jnz     004029FC
0040297B   cmp     dword ptr [ebp-20], 42A674E7    ;比较双精度的后 32 位
00402982   jnz     short 004029FC
```

8. 货币（Currency）比较

这种方法是用两个 Currency 数据类型进行比较。是的，它听起来是怪，但确实存在！Currency 变量存储为 64 位（8 个字节）整型数值的形式，然后除以 10000 给出一个定点数，其小数点左边有 15 位数字，右边有 4 位数字。这种表示法的范围可以从-922 337 203 685 477.5808 到 922 337 203 685 477.5807。Currency 数据类型在货币计算与定点计算中很有用，在这些场合精度特别重要。

下面是一个范例代码：

```
Dim correct As Currency, entered As Currency ;定义 correct 和 entered 为 Currency
correct = 1234567890                         ;设置 correct 为“1234567890”
entered = Text1.Text                         ;设置 entered 为输入的密码
If correct = entered Then                    ;比较
    GoTo Correct Message
Else
    GoTo Wrong Message
End If
```

用 VB 6 编译好的程序为 Currency.exe。代码将 Currency 数据分成两段来比较，先比较前 32 位，再比较后 32 位。汇编形式如下：

```
004023DE   cmp     esi, dword ptr [ebp-1C]     ;数据前 32 位 73CE2B20h
004023E1   jnz     short 00402460
004023E3   cmp     edi, dword ptr [ebp-18]     ;数据后 32 位 00000B3Ah
004023E6   jnz     short 00402460
```

9. 小结

两个表达式都是数值数据类型（Byte, Integer, Long, Single, Double 或 Currency），进行数值比较。当一个 Single 与一个 Double 做比较时，Double 会进行舍入处理而与此 Single 有相同的精确度。如果一个 Currency 与一个 Single 或 Double 进行比较，则 Single 或 Double 转换成一个 Currency。在实际中，程序一般同时采用两种以上的方法来比较数据，如 Currency 和 String、Variant 和 Long 等。

分析 Visual Basic 程序时，对 Vbrun*.dll（VB 3 和 VB 4 版本）和 Msvbvm*.dll（VB 5 和 VB 6）强调得比较多。实际上，Visual Basic 程序的很多运算是在 Oleaut32.dll 中完成的，这个 DLL 提供了对 Visual Basic 中 Variant 类型的变量进行操作的许多函数，主要是一系列 VarXXX()。

除了用 OllyDbg 等工具分析 VB 程序，也可以用 SmartCheck 辅助分析。SmartCheck 是 NuMega 公司推出的一款针对 Visual Basic 的错误检测和调试工具，有关使用参考光盘映像文件中提供的文档。它能够自动检测和诊断 VB 运行时的错误，并将一些表达不清楚的错误信息转换为确切的错误描述。

8.3 伪编译[1]

伪编译是生成伪代码并链接的过程。运行时依赖解释引擎将伪代码翻译为汇编代码再执行。伪代码的运行完全依赖于堆栈。

如果用 IDA 或 W32Dasm 反汇编一下光盘映像文件中提供的 vbpcode.exe 程序，将会看到莫名其妙的代码，因为它不再是传统意义上的汇编代码了，只有用 P-code 反编译器才能得到正确的代码。VB 5/VB 6 的 P-code 反编译器现在有 Exdec、WKTVBDebugger、VBDE 等。

8.3.1 虚拟机与伪代码

相对 VB Native Code, P-code 编译模式要出现得更早一些。事实上，在 5.0 以前的版本中，所有的 VB 程序代码都不加选择地被编译为 P-code 形式。VB P-code 的运行速度较慢，这是由它本身的运行机制所决定的。

1. 虚拟执行的原理

P-code，即 Pseudo Code（伪代码），这一概念最早出现在 Pascal 编译器中，它是为了提供跨平台可移植性而产生的，实现这一编译机制的 Pascal 编译器被称为“Pascal P Compiler”。Sun 公司在其推出的 Java 语言上也成功地实现了这种机制。Java 程序的伪编译代码由一系列代表一定意义的字节码（byte code）组成，它们同属于一套特定的指令集，这种字节码不能由不同的 CPU 直接执行，而是要通过特殊的解释引擎翻译为 CPU 可以识别的指令才能执行，这种解释引擎就是我们常说的“虚拟机”。只要在不同的平台上提供虚拟机，把字节码翻译为对应的 CPU 指令集，也就实现了所谓的跨平台特性。

Microsoft 推出的 VB P-code，实际上也是一组自定义的指令集，必须通过基于堆栈的虚拟机翻译为 80x86 上的指令集才能执行，担任虚拟机任务的就是系统目录下的 msvbvm50.dll 和 msvbvm60.dll 这两个动态链接库文件。由于在文件执行过程中多出了这一个解释的步骤，自然要影响到其执行的速度。至少到目前为止，VB P-code 并没有实现所谓的跨平台运行特性，这对于 Pseudo 这个词的起源是不恰当的；另外，采用 P-code 形式编译生成的 VB 应用程序体积一般要小于采用 Native Code 形式编译的同样程序（这是由于 P-code 指令集的每一条指令往往对应于一组 80x86 指令所完成的任务）。

2. 虚拟机实例分析

为了理解虚拟机的运作方式，这里提供一个 P-code 编译的小程序 VBP-code-Ex.exe 来进行说明，其源码如下：

```
Private Sub Command1_Click()
X = InputBox("Please input an integer", "Input")
If X <> "" Then
    Y = X + 2
    X = Y * 3
    Out.Text = X
End If
End Sub
```

这个程序只处理 Command_Click 事件，同时用到了 InputBox，以便用 rtcInputBox 函数设断。

用 OllyDbg 加载这个程序，在命令行插件中输入 bp rtcInputBox，按下 F9 键运行。然后单击 OK 按钮，中断如下：

```
6A360CF2  PUSH EBP                          ;rtcInputBox 函数的第一句指令
6A360CF3  MOV EBP,ESP
6A360CF5  SUB ESP,54
```

[1] 本节由丁益青编写。

```
6A360CF8  MOV EAX,DWORD PTR SS:[EBP+1C]
```

按下“Ctrl+F9”键返回，当InputBox弹出以后任意输入一个整数，然后继续单步跟踪，直到下面的代码处：

```
6A37D2CD  MOVSX EAX,WORD PTR DS:[ESI]
6A37D2D0  PUSH DWORD PTR DS:[EAX+EBP]
6A37D2D3  XOR EAX,EAX
6A37D2D5  MOV AL,BYTE PTR DS:[ESI+2]
6A37D2D8  ADD ESI,3
6A37D2DB  JMP DWORD PTR DS:[EAX*4+6A37DA58]   ;注意这句
6A37D2E2  MOVSX EAX,WORD PTR DS:[ESI]         ;停在这里
6A37D2E5  ADD EAX,EBP
6A37D2E7  PUSH EAX
6A37D2E8  XOR EAX,EAX
6A37D2EA  MOV AL,BYTE PTR DS:[ESI+2]
6A37D2ED  ADD ESI,3
6A37D2F0  JMP DWORD PTR DS:[EAX*4+6A37DA58]   ;注意这句
```

注意：由于msvbvm60.dll版本不同，读者电脑显示的代码或地址会与本书不同。

读者如果不明白这些指令的意图，就会迷失在行为相似的代码之中。事实上，这几条指令正是虚拟机读取P-code伪代码的引擎部分。在这里，尤其值得关注的是它读取了什么，而不是执行了什么。为了说明这些指令的功能，首先要了解VB P-code伪代码的格式：

```
3A 6C FF 03 00
操作码   操作数
```

这是一句典型的VB P-code指令，其助记符为LitVarStr，表示将一个Variant型的字符串入栈。3A是指令的操作码，0003是操作数，是以某种形式标记的字符串地址。现在有足够的信息来解释虚拟机所做的一切了，代码如下：

```
6A37D2E2  MOVSX EAX,WORD PTR DS:[ESI]         ;esi指向待解释指令的操作数
6A37D2E5  ADD EAX,EBP                         ;取得某种形式的字符串指针
6A37D2E7  PUSH EAX                            ;压栈（这是伪代码的核心操作）
6A37D2E8  XOR EAX,EAX                         ;eax清零
6A37D2EA  MOV AL,BYTE PTR DS:[ESI+2]          ;取下一条指令的操作码
6A37D2ED  ADD ESI,3                           ;移至下一条指令的操作数
6A37D2F0  JMP DWORD PTR DS:[EAX*4+6A37DA58]   ;根据跳转地址表到下一条指令的解释单元
```

虚拟机并没有用简单的条件判断来识别操作码，而是采用了跳转地址表法，把执行流直接引导到相应的解释单元。6A37DA58h是地址跳转表的首地址，eax保存了下一条指令的操作码，由于每一个跳转地址是一个dword，所以用eax乘以4的值加上跳转表的基地址来索引下一条指令的解释单元。当然，由于每一条指令的长度是不同的，所以前面提到的读取P-code伪代码的引擎部分并不完全相同。明白了虚拟机的解释原理，跟踪P-code程序就方便多了，一旦走到读取P-code伪代码的引擎部分，就意味着虚拟机开始解释下一条指令了。尽管如此，调试P-code程序还是比调试本地机器码编译的普通可执行程序困难得多，因为在虚拟机的工作流程下无法随时回顾前面执行过的指令。

下面来看看加法在虚拟机中是如何被解释执行的。按F8键跟踪代码来到：

```
6A37D3B3  ADD EDI,EBP
6A37D3B5  PUSH EDI
6A37D3B6  XOR EAX,EAX
6A37D3B8  MOV AL,BYTE PTR DS:[ESI]
6A37D3BA  INC ESI
6A37D3BB  JMP DWORD PTR DS:[EAX*4+6A37DA58]   ;al=FBh
```

走到这里时 al 的值为 FB，这是 Variant 型变量算术运算伪指令的操作码，esi 则指向了该操作码后的一个次操作码 94，代表加法运算。跟随跳转地址表来到：

```
6A37D9BA  XOR EAX,EAX
6A37D9BC  MOV AL,BYTE PTR DS:[ESI]
6A37D9BE  INC ESI
6A37D9BF  JMP DWORD PTR DS:[EAX*4+6A37DE58]   ;al=94h
```

这里是一个二级跳转表，注意 6A37DE58 这个值和前面的跳转地址表基址不同了，这说明 VB P-code 算术运算伪指令采用了二级跳转来解释执行。由于一个字节的操作码可以表示的操作指令种类最多为 256 个，为了解决指令数不够的问题，Microsoft 对部分伪指令进行二级跳转解释执行。

```
6A384628  LEA EBX,DWORD PTR DS:[__vbaVarSub]
6A38462E  JMP SHORT MSVBVM60.6A384610
6A384630  LEA EBX,DWORD PTR DS:[__vbaVarMul]
6A384636  JMP SHORT MSVBVM60.6A384610
6A384638  LEA EBX,DWORD PTR DS:[__vbaVarDiv]
6A38463E  JMP SHORT MSVBVM60.6A384610
6A384640  LEA EBX,DWORD PTR DS:[__vbaVarIdiv]
6A384646  JMP SHORT MSVBVM60.6A384610
6A384648  LEA EBX,DWORD PTR DS:[__vbaVarMod]
6A38464E  JMP SHORT MSVBVM60.6A384610
6A384650  LEA EBX,DWORD PTR DS:[__vbaVarAdd]  ;跳转到这里执行，获得加法函数地址
6A384656  JMP SHORT MSVBVM60.6A384610
6A384658  LEA EBX,DWORD PTR DS:[__vbaVarAnd]
6A38465E  JMP SHORT MSVBVM60.6A384610
6A384660  LEA EBX,DWORD PTR DS:[__vbaVarOr]
6A384666  JMP SHORT MSVBVM60.6A384610
```

很明显，Variant 变量的算术逻辑运算伪操作都分布在这一区域，待调用函数__vbaVarAdd 的地址被装入 ebx，继续向下来到：

```
6A384610  MOVSX EDI,WORD PTR DS:[ESI]        ;取加法指令的操作数
6A384613  ADD EDI,EBP
6A384615  PUSH EDI                           ;操作数入栈
6A384616  CALL EBX                           ;这里调用函数__vbaVarAdd 执行加法操作
6A384618  PUSH EDI                           ;运算结果保存在堆栈结构中
6A384619  XOR EAX,EAX
6A38461B  MOV AL,BYTE PTR DS:[ESI+2]         ;继续取下一条伪指令的操作码
6A38461E  ADD ESI,3                          ;指向下一条伪指令的操作数
6A384621  JMP DWORD PTR DS:[EAX*4+6A37DA58];跳向下一条伪指令的解释单元
```

在 OllyDbg 朴素而强有力的逐行追击中，VB P-code 虚拟机的解释机理尽可一览无余。

8.3.2 动态分析 VB P-code 程序

毋庸置疑，OllyDbg 完全可以胜任 VB P-code 程序的调试，然而在大部分情况下，它并非是最好的选择。基于通常的调试习惯，人们总是希望能够直接跟踪可执行文件本身的每一条指令，以便直观地了解程序实现的功能。从这个角度来说，通用调试工具是分析 VB P-code 虚拟机的优秀选手，但是却无法将更多的注意力集中在 P-code 程序本身的功能上。

伴随着这样一种想法，功能强大的 VB P-code 专用调试器 WKTVBDebugger 应运而生了。这个具有划时代意义的 VB P-code 调试器，掀开了解释型语言程序调试策略的崭新一页，它不仅仅给 VB P-code 程序的调试带来了极大的方便，其编写思想和运作机制也给人以深刻启发。在本节中，笔者将为大家简要介绍这一款调试器的各种特性和使用方法。

1. 介绍

WKTVBDebugger 以单个伪代码指令为执行单元，包括跟踪、设断、修改等各个方面都完全以伪代码为基础，彻底屏蔽了虚拟机的解释细节，如同直接调试 P-code 可执行文件的功能。举例来说，当程序执行一条 Variant 型变量的加法伪指令时，读者在调试器中跟踪的将不再是冗长繁复的虚拟机解释代码，而是 AddVar 这一条指令（严格地说，AddVar 只是助记符），不仅操作码如此，每一条伪指令的操作数也可以通过堆栈来方便地观察。

在基于本地机器码的调试器实现中，通过设置单步跟踪标志位，可以在每执行一条指令以后产生单步调试断点，从而将控制权转移给调试器，调试器在执行下一条指令前可以观察寄存器的状态并修改程序的执行流程；此外，调试器还可以通过在指定的指令位置插入 CC（Int 3）设置断点，捕获异常并取得控制权。特别地，在 Win32 环境下，这类断点在 Ring 3 下形成异常，由 Windows 系统向调试器报告异常事件，调试器由此获得处理异常的优先权。然而，对于由虚拟机解释执行的伪代码，因为系统并未给程序开发人员特意留出标准的调试接口，异常的处理对用户而言也是透明的，传统的调试器原理便无法照搬过来。

WKTVBDebugger 的实现，与 VB P-code 虚拟机运作机制以及伪指令的研究是密不可分的。其作者 Mr. Silver 和 Mr. Snow 采用了一种类似于 Hook 的方法，把调试器代码插入到虚拟机和 VB P-code 伪代码之间，从而巧妙地解决了如何让调试器取得控制权的问题。观察一下 VB P-code 虚拟机伪代码读取引擎：

```
XOR EAX,EAX
MOV AL,BYTE PTR DS:[ESI+2]
ADD ESI,3
JMP DWORD PTR DS:[EAX*4+6A37DA58]
```

在这一模式中，寄存器 esi 始终指向伪指令流，虚拟机读取下一条伪指令的操作码以后，又把 esi 指向次级操作码（对于具有多级操作码的伪指令）或者操作数（对于具有单级操作码的伪指令），每一条 jmp 指令都通过跳转地址表移向下一条伪指令的解释单元……如果让这里的跳转地址指向自定义的调试代码，不就可以在下一条伪指令执行前取得控制权了吗？调试所需的初始化工作，仅仅是把 6A37DA58 所指向的跳转地址表中的所有项替换为调试循环代码的入口地址，一旦在此截获 P-code 指令，由于 al 保存着伪指令的操作码，而 esi 则指向伪指令的操作数，调试器便可以把对应的伪指令助记符在反编译窗口中显示出来，并判断当前的地址是否需要中断……无论如何，程序的执行流程已尽在掌握。一旦调试器的任务完成，就恢复现场的寄存器环境，根据原来的跳转地址表移至下一条指令的解释单元。这就是 WKTVBDebugger Hook 的基本原理。

调试器的基本框架有了，随之而来的是伪指令的解释问题。Microsoft 定义了 VB P-code 指令规范，但却没有把它们公开。这不是一个大问题，无论如何，这套指令规范是可以获得的。这里要感谢 Josephco，他的 VB P-code 反编译器 Exdec 在这方面做了许多先驱性的工作。另外，笔者推荐两款相当不错的国产 VB P-code 反编译器：万涛编写的 VBExplorer 和 ljtt 编写的 VBParser。严格地说，VBExplorer 并不是一个纯粹的反编译器，它还可以编辑修改 VB 控件的各种属性，但反编译功能做得还不精。相比之下，VBParser 是一个专业的反编译器了。

2. 安装

执行 WKTVBDebugger 文件就能完成安装。但还有几个方面要注意一下，如果不能正常装载程序，请如下尝试。

- 将目标软件复制到 WKTVBDebugger 安装目录里调试，即与 Loader.exe 同一目录；
- 将安装目录的 WKTVBDE.dll 文件复制到系统目录里；
- 将 MSVBVM60.DLL 替换成 2003 年以前版本。

3. 使用

运行 WKTVBDebugger 后，打开目标程序，单击菜单“Action/Run”装载程序。输入用户名及序列号后，单击“确定”按钮就能中断在 WKTVBDebugger 里，如图 8.4 所示。

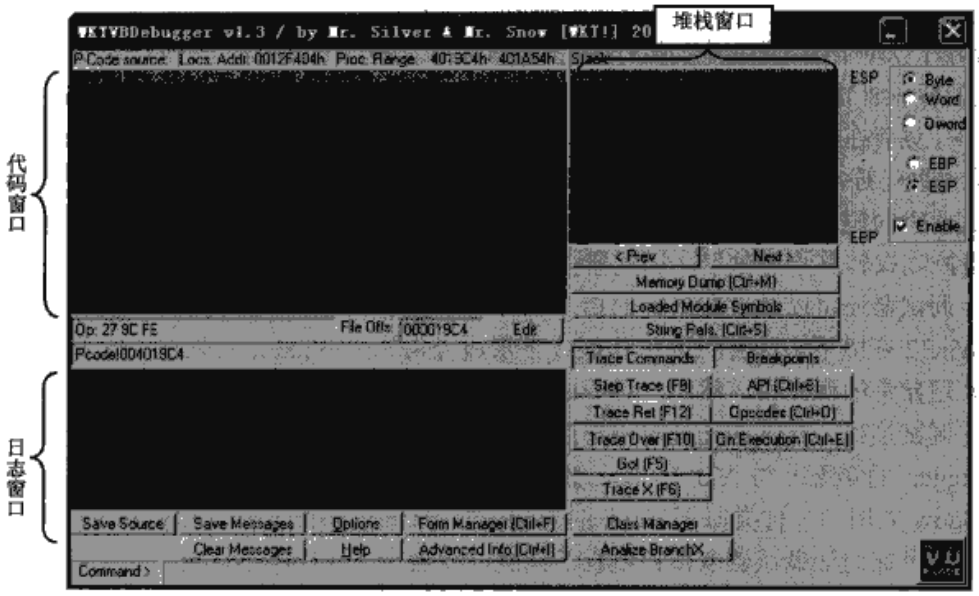


图 8.4　WKTVBDebugger 主窗口

主窗口由 3 个基本部分组成：代码（Disassembly）、日志（Log）以及堆栈（Stack）窗口。

（1）代码窗口

这个窗口显示当前执行程序的反汇编伪代码。指令格式如下：

```
XXXXXXXX:xx 指令
```

其中“XXXXXXXX”是指令的内存虚拟地址，跟着的两个数字是以十六进制格式显示的机器码。注意，只有第一个机器码被显示了，后面的是 P-code 伪指令（会根据变量和自变量的不同而不同）。另外，在该窗口中单击鼠标右键将打开命令菜单。

（2）日志窗口

当程序执行时，显示程序所有变量的信息和提示消息。所以当跟踪调试时，注意这个窗口是很重要的。它将提供一些有价值的信息，最重要的是，它将显示当前指令执行时所进行的操作。

（3）堆栈窗口

堆栈窗口有几个模式：字节、字或双字。P-code 运行有别于传统的 CPU 结构，传统的 CPU 执行依赖于寄存器和堆栈；而 P-code 只使用堆栈，所以堆栈窗口非常重要，各种指令都通过堆栈来交换数据。

4．机器码与助记命令

机器码与助记命令表（Opcode and Mnemonics Table）很重要，具体参考 WKTVBDebugger 的帮助文件。掌握 VB P-code 的关键就在这些助记命令上。这里只列出几个常用的助记命令。

- BranchF：机器码是 1Ch，3 个字节。条件跳转指令，如堆栈的值是 0 就跳转。单击“Analize BranchX”按钮可以了解当前进程中的所有条件跳转指令的位置。
- BranchT：机器码是 1Dh，3 个字节。条件跳转指令，如堆栈的值是 FFFFFFFh（-1）就跳转。
- Branch：机器码是 1Eh，3 个字节。无条件转移。
- EqVarBool：机器码是 33h，1 个字节。比较指令，比较两个变量，根据结果将-1 或 0 压进堆栈。
- LitI2_Byte：机器码是 F4h，2 个字节，将数据压入堆栈。
- ConcatStr：机器码是 2Ah，1 个字节。字符串连接指令（相当于 C 语言中的 strcat 函数）。此指令单步跟踪（Step Trace）时，会在日志窗口中留下相应字符串连接结果。所以，可以单击“Opcodes”在此伪指令处设置断点，以便了解某字符串值。
- FLdZeroAd / CVarStr：取字符串指令，特点同 ConcatStr。

8.3.3　伪代码的综合分析

尽管 Microsoft 为 VB P-code 定义了一整套伪指令助记符，但是并没有权威的技术文档解释这些伪指令执行的细节。既然 WKTVBDebugger 作为一个伪代码级的调试器已经屏蔽了 VB P-code 虚拟机的解释过程，为什么还要颇费周折地去了解这些伪指令执行的细节呢？在调试的过程中，读者一定会遇到这样的问题：假设用 WKTVBDebugger 步过了 AddVar 这一条伪指令，试问如何才能获得它的操作数和操作结果？既然 VB P-code 虚拟机是基于堆栈的，那么操作数和操作结果一定存放在堆栈中。实际情况诚然如是，然而它们究竟是以怎样的形式存在呢？是单个的操作数，是指针，还是其他复杂的数据结构？对于不同的 P-code 伪指令，其存放形式也是迥然相异的。如果在调试的过程中无法正确地获知和修改每条指令的操作数和操作结果，那么 WKTVBDebugger 的功能也就止步于静态反编译而已。

这里仍然以 VBP-code-Ex.exe 为例说明如何用现有的工具来解决上述困惑。首先使用 ljtt 的 VBParser 解析 VB P-code.exe，得到伪代码如下：

```
-----------------------------Pcode-----------------------------
[CommandButton]
Private Sub Command1_Click()
'-=-=-=-=-=-=- ProcAddr Range: [004019C4 - 00401A54] , ProcSize: 90 -=-=-=-=-=-=-
004019C4: 27 9C FE          LitVar_Missing      PushVarError 80020004 (missing)
004019C7: 27 BC FE          LitVar_Missing      PushVarError 80020004 (missing)
004019CA: 27 DC FE          LitVar_Missing      PushVarError 80020004 (missing)
004019CD: 27 FC FE          LitVar_Missing      PushVarError 80020004 (missing)
004019D0: 27 1C FF          LitVar_Missing      PushVarError 80020004 (missing)
*********** Referent String: "Input" ***********
004019D3: 3A 4C FF 00 00    LitVarStr           PushVarString Ptr_00401434
004019D8: 4E 3C FF          FStVarCopyObj       [local_C4]=vbaVarDup(Pop)
004019DB: 04 3C FF          FLdRfVar            Push local_C4
*********** Referent String: "Please input an integer" ***********
004019DE: 3A 6C FF 01 00    LitVarStr           PushVarString Ptr_00401400
004019E3: 4E 5C FF          FStVarCopyObj       [local_A4]=vbaVarDup(Pop)
004019E6: 04 5C FF          FLdRfVar            Push local_A4
```

以上就是 Command_Click 事件响应代码的开头部分，由于 VB P-code 虚拟机以流的形式顺序读入每一句伪指令，然后通过一个跳转地址表找到相应的解释代码，为了跟踪它解释伪指令的细节，就必须在伪指令的操作码上下内存访问断点。第一句伪指令 LitVar_Missing 从 004019C4（虚拟地址）开始，用 OllyDbg 加载 VBP-code-Ex.exe，在转储窗口中来到 004019C4，对第一个字节（操作码）下内存访问断点，按 F9 键执行，单击 OK 按钮，中断在这里：

```
6A37D153  MOV AL,BYTE PTR DS:[ESI]              ;开始读操作码，esi=004019C4
6A37D155  INC ESI                               ;使 esi 指向操作数
6A37D156  JMP DWORD PTR DS:[EAX*4+6A37DA58]     ;根据跳转地址表和操作码寻址解释单元
```

按 F8 键往下走一步，来到这条伪指令的解释单元：

```
6A37D39F  MOVSX EDI,WORD PTR DS:[ESI]           ;把字操作数带符号扩展到 edi
6A37D3A2  ADD ESI,2                             ;esi 指向下一句伪指令的操作码
6A37D3A5  MOV WORD PTR DS:[EDI+EBP],0A          ;ebp 是程序堆栈区某处的基址，但不是
                                                ;堆栈顶的指针，它把 0A 保存到 edi 指向
                                                ;的偏移地址处
6A37D3AB  MOV DWORD PTR DS:[EDI+EBP+8],80020004;向下偏移 8 个字节处存入 80020004，根
                                                ;据 VBParser 的说明，这个数字表示空
                                                ;参数（缺省参数），事实上在源代码中
                                                ;确实没有提供这个参数
6A37D3B3  ADD EDI,EBP                           ;使 edi 指向 0A
6A37D3B5  PUSH EDI                              ;在堆栈中压入这个虚拟地址
6A37D3B6  XOR EAX,EAX                           ;清空 eax，准备读取下一句伪指令
6A37D3B8  MOV AL,BYTE PTR DS:[ESI]              ;读取下一条伪指令的操作码
```

```
6A37D3BA  INC ESI                                  ;使 esi 指向下一句伪指令的次级操作码
                                                   ;或操作数
6A37D3BB  JMP DWORD PTR DS:[EAX*4+6A37DA58]        ;根据地址跳转表和操作码寻址解释单元
```

看一下这些指令执行完以后的堆栈：

```
0012F458   0012F494 ──┐                           ;栈顶
0012F45C   00000000   │
0012F460   00000000   │
.....                 │
.....                 │
0012F488   00000000   │
0012F48C   00000000   │
0012F490   00000000   │
0012F494   0000000A   │                           ;这就是刚才压入栈顶的数据
0012F498   00000000 ←─┘
0012F49C   80020004
0012F4A0   00000000
```

现在这个伪指令的动作明确了，LitVar_Missing 执行时，把一个虚拟地址压入堆栈，这个虚拟地址指向 0000000A, 00000000, 80020004。实际上，这句伪指令的功能就是在堆栈中提供一个空参数，其堆栈完全没有参考价值。在下面的说明中笔者将省略对虚拟机伪代码读取引擎的注释，因为这部分都是一样的。

现在来看看 4019D3 处的伪指令 LitVarStr。老规矩，在 4019D3 处设内存访问断点，按 F9 键中断在下面的地方：

```
6A37D3B8  MOV AL,BYTE PTR DS:[ESI]                 ;esi=004019D3
6A37D3BA  INC ESI
6A37D3BB  JMP DWORD PTR DS:[EAX*4+6A37DA58]
```

跟随跳转来到：

```
6A37D3C2  MOVSX EDI,WORD PTR DS:[ESI]              ;第一个字操作数 FF4C（堆栈区偏移量）
                                                   ;带符号扩展到 edi
6A37D3C5  MOVZX EAX,WORD PTR DS:[ESI+2]            ;第二个字操作数 0000（数据区偏移量）
                                                   ;无符号扩展到 eax
6A37D3C9  MOV EDX,DWORD PTR SS:[EBP-54]            ;取得 P-code 程序数据区的基址
6A37D3CC  MOV EAX,DWORD PTR DS:[EDX+EAX*4]         ;根据 eax 产生偏移量，取得数据区的数据
                                                   ;这是一个虚拟地址，指向 Unicode 字符串
                                                   ;"Input"
6A37D3CF  ADD EDI,EBP                              ;使 edi（堆栈区偏移量）指向堆栈区即将
                                                   ;保存数据的地方
6A37D3D1  MOV WORD PTR DS:[EDI],8                  ;用于识别数据类型
6A37D3D6  MOV DWORD PTR DS:[EDI+8],EAX             ;向下偏移 8 个字节处存入指向 Unicode
                                                   ;字符串“Input”的虚拟地址
6A37D3D9  PUSH EDI                                 ;最后堆栈区数据指针入栈
6A37D3DA  XOR EAX,EAX
6A37D3DC  MOV AL,BYTE PTR DS:[ESI+4]
6A37D3DF  ADD ESI,5
6A37D3E2  JMP DWORD PTR DS:[EAX*4+6A37DA58]
```

同样地，有必要观察一下堆栈：

```
0012F444   0012F544 ──┐                           ;栈顶
0012F448   0012F514   │                           ;下面是前面其他指令形成的堆栈
0012F44C   0012F4F4   │
0012F450   0012F4D4   │
0012F454   0012F4B4   │
0012F458   0012F494   │
0012F45C   00000000   │
.....                 │
.....                 │
0012F540   00000000   │
0012F544   00000008 ←─┘
0012F548   00000000
```

```
0012F54C   00401434  UNICODE "Input"
0012F550   00000000
```

结合上述跟踪分析，LitVarStr 伪指令对操作数的获取方法就清楚了：首先在转储窗口观察 12F544 处的内容，向后移动 8 个字节，得到虚拟地址 401434，观察 401434 处的内容，就是入栈的字符串参数了。

相应地，笔者在 WKTVBDebugger 中演示一下操作的过程：

① WKTVBDebugger 加载 VB P-code.exe；

② 在 Form Manager 中对 Command1 控件设断点；

③ 单击 OK 按钮；

④ WKTVBDebugger 中断在下面的地方：

```
004019C4: 27 LitVar_Missing 0012F474h
004019C7: 27 LitVar_Missing 0012F494h
004019CA: 27 LitVar_Missing 0012F4B4h
004019CD: 27 LitVar_Missing 0012F4D4h
004019D0: 27 LitVar_Missing 0012F4F4h
004019D3: 3A LitVarStr 'Input'                    ;这就是在 OllyDbg 中跟踪分析的
                                                  ;LitVarStr 伪指令
004019D8: 4E FStVarCopyObj 0012F514h
004019DB: 04 FLdRfVar 0012F514h
004019DE: 3A LitVarStr 'Please input an integer'
004019E3: 4E FStVarCopyObj 0012F534h
004019E6: 04 FLdRfVar 0012F534h
004019E9: 0B ImpAdCallI2 rtcInputBox on address 73472265h
```

⑤ 注意注释标出的这条伪指令，单步走过这条指令时，右上角堆栈窗口显示如下（为了便于观察，在右侧的单选框中选择 DWORD）：

```
0012F424: 0012F524 0012F4F4                       ;注意这里的栈顶 0012F524
0012F41C: 0012FE3C 0000004E
.....
0012F3B4: 00000000 00000000
0012F3AC: 77E6780F 0000008C
```

⑥ 按下“Ctrl+M”键打开转储窗口，在 Address to Dump 组合框中输入 0012F524，得到：

```
0012F524:08 00 00 00 00 00 00 00
0012F52C:34 14 40 00 00 00 00 00                  ;向下移动 8 个字节
                                                  ;00401434 就是对应字符串的指针了
0012F534:00 00 00 00 00 00 00 00
```

⑦ 记下这个指针，输入到 Address to Dump 组合框，现在这个 Unicode 字符串终于露出了真面目：

```
00401434:49 00 6E 00 70 00 75 00        I.n.p.u.
0040143C:74 00 00 00 00 00 00 00        t.......
00401444:00 00 00 00 E1 4E AD 33        ....酸?
```

当然，就该指令本身而言，WKTVBDebugger 已经在日志窗口中输出了其操作数，所以要观察这个字符串大可不必那么麻烦，但是对于其他没有日志记录的伪指令，这套分析方法仍是具有启发性的。

8.3.4　VB P-code 攻击实战

本节以 CyberBlade 编写的一个中等难度的 VB P-code CrackMe 作为范例来初步介绍 VB P-code 程序的调试过程，以帮助读者熟悉一些常见的 P-code 伪指令。

由于反编译器往往也提供了一些有用的信息，因此首先以 Josephco 的 Exdec 来生成该 CrackMe 的反编译代码，对照 WKTVBDebugger 的代码窗口进行调试。在 WKTVBDebugger 的 Form Manager 中对 Check

按钮下断点，填写用户名“cyclotron”和序列号“131421”，单击“Check”按钮以后，首先是关于用户名和序列号的存在性校验。

```
40E26C: 04 FLdRfVar          local_009C
40E26F: 21 FLdPrThis
40E270: 0f VCallAd           text
40E273: 19 FStAdFunc         local_0098
40E276: 08 FLdPr             local_0098
40E279: 0d VCallHresult      get__ipropTEXTEDIT   ;调用函数取得用户名
40E27E: 6c ILdRf             local_009C
40E281: 1b LitStr:
40E284: Lead0/30 EqStr                            ;用户名是否存在
40E286: 2f FFree1Str         local_009C
40E289: 1a FFree1Ad          local_0098
40E28C: 1c BranchF:          40E2C1
.................
.................
40E2CE: 0d VCallHresult      get__ipropTEXTEDIT   ;调用函数取得序列号
40E2D3: 6c ILdRf             local_009C
40E2D6: 1b LitStr:
40E2D9: Lead0/30 EqStr                            ;序列号是否存在
40E2DB: 2f FFree1Str         local_009C
40E2DE: 1a FFree1Ad          local_0098
40E2E1: 1c BranchF:          40E316
40E2E4: 27 LitVar_Missing
40E2E7: 27 LitVar_Missing
40E2EA: 3a LitVarStr:        (local_00CC) Error
40E2EF: 4e FStVarCopyObj     local_00DC
40E2F2: 04 FLdRfVar          local_00DC
40E2F5: f5 LitI4:            0x40  64  (...@)
40E2FA: 3a LitVarStr:        (local_00AC) You have to enter a key first.
```

接着是关于序列号长度合法性的校验：

```
40E323: 0d VCallHresult      get__ipropTEXTEDIT   ;调用函数取得序列号
40E328: 6c ILdRf             local_009C
40E32B: 1b LitStr:
40E32E: Lead0/30 EqStr                            ;序列号是否大于等于5位
40E330: 2f FFree1Str         local_009C
40E333: 1a FFree1Ad          local_0098
40E336: 1c BranchF:          40E36B
40E339: 27 LitVar_Missing
40E33C: 27 LitVar_Missing
40E33F: 3a LitVarStr:        ( local_00CC ) Error
40E344: 4e FStVarCopyObj     local_00DC
40E347: 04 FLdRfVar          local_00DC
40E34A: f5 LitI4:            0x40  64  (...@)
40E34F: 3a LitVarStr:         ( local_00AC ) You have to enter at least 5 chars.
40E354: 4e FStVarCopyObj     local_00BC
40E357: 04 FLdRfVar          local_00BC
```

从下面开始的两个循环是这个 CrackMe 的算法核心，其中所涉及的指令和函数对 P-code 程序的调试都具有相当的指导意义。

第一个循环（For 结构）：

```
40E36B: 28 LitVarI2:         (local_00EC)0x1 (1)  ;立即数1入栈
                                                  ;Lit 表示立即数
                                                  ;作为循环初始值
40E370: 04 FLdRfVar          local_012C           ;入栈，保存循环计数器
```

```
40E373: 04 FLdRfVar          local_009C
40E376: 21 FLdPrThis
40E377: 0f VCallAd           text
40E37A: 19 FStAdFunc         local_0098
40E37D: 08 FLdPr             local_0098
40E380: 0d VCallHresult      get__ipropTEXTEDIT    ;调用函数取得用户名
40E385: 6c ILdRf             local_009C
40E388: 4a FnLenStr                                ;取用户名的长度
40E389: Lead2/69 CVarI4      local_00CC            ;转换（C-Convert）为
                                                   ;变体型（Var-Variant）
                                                   ;以用户名长度为循环次数
40E38D: 2f FFree1Str         local_009C
40E390: 1a FFree1Ad          local_0098
40E393: Lead3/68 ForVar:     (when done) 40E3F5    ;ForVar 开始循环计算
40E399: 04 FLdRfVar          local_009C
40E39C: 21 FLdPrThis
40E39D: 0f VCallAd           text
40E3A0: 19 FStAdFunc         local_0098
40E3A3: 08 FLdPr             local_0098
40E3A6: 0d VCallHresult      get__ipropTEXTEDIT    ;取用户名的长度
40E3AB: 04 FLdRfVar          local_0094            ;用户名的指针入栈
40E3AE: 28 LitVarI2:         (local_00DC)0x1 (1)   ;立即数 1 入栈
40E3B3: 04 FLdRfVar          local_012C            ;循环计数器
40E3B6: Lead1/22 CI4Var                            ;转换为整型(I-Integer)
40E3B8: 3e FLdZeroAd         local_009C
40E3BB: 46 CVarStr           local_00BC            ;转换为变体型
40E3BE: 04 FLdRfVar          local_00FC            ;用户名字符串指针入栈
40E3C1: 0a ImpAdCallFPR4:                          ;调用 rtcMidCharVar 函数
                                                   ;每轮循环取用户名的一个字符
40E3C6: 04 FLdRfVar          local_00FC            ;取得的字符入栈
40E3C9: Lead2/fe CStrVarVal  local_0150            ;转换为字符串型(Str-String)
40E3CD: 0b ImpAdCallI2                             ;调用 rtcAnsiValueBstr 函数
                                                   ;转换字符为 ASCII 码
40E3D2: 44 CVarI2            local_00CC            ;转换为变体型
40E3D5: Lead0/ef ConcatVar                         ;将每轮循环得到的十进制数作为
                                                   ;字符串相连接
40E3D9: Lead1/f6 FstVar                            ;保存字符串（St－Save to）
40E3DD: 2f FFree1Str         local_0150
40E3E0: 1a FFree1Ad          local_0098
40E3E3: 36 FFreeVar
40E3EC: 04 FLdRfVar          local_012C
40E3EF: Lead3/7e NextStepVar:(continue) 40E399     ;继续下一轮循环
```

总结这个循环的算法如下：

（1）逐位读取用户名字符：

" cyclotron " → 'c'，'y'，'c'，'l'，'o'，'t'，'r'，'o'，'n'

（2）分别转换为十进制的数字字符串：

99，121，99，108，111，116，114，111，110

（3）顺序连接所有的数字字符串：

“991219910811111161141111110”，记为 S1

注意：这个字符串是以 Unicode 编码的形式在内存中出现的。

下面来看第二个循环（While 结构）：

```
40E3F5: 04 FLdRfVar      local_0094          ;第一个循环生成的数字字符串入栈
40E3F8: Lead0/eb FnLenVar                    ;取其长度
40E3FC: 28 LitVarI2:     (local_00AC)0x9(9)  ;立即数 9 入栈
40E401: 5d HardType
```

```
40E402: Lead0/74 GtVarBool                          ;是否大于9位
40E404: 1c BranchF:             40E425              ;小于等于9位则终止循环
40E407: 04 FLdRfVar             local_0094
40E40A: Lead3/c4 LitVarR8                           ;浮点立即数3.141592654000000
                                                    ;入栈，R8-Real number of 8 bytes
```

在这个位置WKTVBDebugger会遇到一点小小的困难，它无法现场输出这个浮点立即数的标准形式。如何获得这个浮点数呢？可以利用OllyDbg的数据窗口以标准形式输出浮点数。

```
40E416: Lead0/bc DivVar                             ;做除法
                                                    ;先入栈的是被除数，后入栈的是除数
                                                    ;其他算术运算指令也遵循这个规则
40E41A: Lead0/e1 FnFixVar                           ;对除法的结果取整
40E41E: Lead1/f6 FstVar                             ;保存结果为Variant型
40E422: 1e Branch:              40e3f5              ;回到40E3F5循环运算
40E425: 04 FLdRfVar             local_0094
40E428: Lead3/c1 LitVarI4:      (local_param_5678FF54) 0x30f85678  (821581432)
                                                    ;立即数0x30f85678入栈
40E430: Lead0/17 XorVar                             ;Xor-异或运算
40E434: Lead1/f6 FstVar                             ;保存为变体型
40E438: 04 FLdRfVar             local_0094          ;前面的运算结果入栈
40E43B: 08 FLdPr                local_param_0008
40E43E: 8a MemLdStr                                 ;调入一个内存操作数0D8B3h
40E441: Lead2/69 CVarI4         local_00AC          ;转换为变体型
40E445: Lead0/9c SubVar                             ;两数相减
40E449: Lead1/f6 FstVar                             ;保存为变体型，记为S2
```

上述循环运算过程可以总结如下：

（1）变换S1到9位以下的十进制形式：

```
while ( strlen(S1) > 9 )
S1 = Fix ( S1/3.141592654 )
```

（2）变换S1到S2：

S2 = (S1 xor 30F85678h) – 0D8B3h = 667574632

接下来进入另一个循环：

```
40E44D: 28 LitVarI2:            (local_00EC)0x1(1)  ;立即数1入栈
                                                    ;作为循环的初始值
40E452: 04 FLdRfVar             local_012C          ;循环计数器
40E455: 28 LitVarI2:            (local_00CC)0xa(10) ;循环终止值10
40E45A: Lead3/68 ForVar:        (when done) 40E495  ;进入循环
40E460: 04 FLdRfVar             local_009C
40E463: 21 FLdPrThis
40E464: 0f VCallAd              text
40E467: 19 FStAdFunc            local_0098
40E46A: 08 FLdPr                local_0098
40E46D: 0d VCallHresult         get__ipropTEXTEDIT  ;调用函数取得序列号
40E472: 6c ILdRf                local_009C
40E475: 04 FLdRfVar             local_012C
40E478: Lead1/22 CI4Var                             ;转换为整型
40E47A: 08 FLdPr                local_param_0008
40E47D: 06 MemLdRfVar           local_param_0034
40E480: 9e Ary1LdI4                                 ;从字符串数组依次取出一系列字符串
    UNICODE "373703670"
    UNICODE "684708686"
    UNICODE "698673531"
    UNICODE "391184533"
    UNICODE "329528230"
    UNICODE "654824169"
```

```
    UNICODE "557168731"
    UNICODE "387375850"
    UNICODE "212298498"
    UNICODE "851143730"
40E481: Lead0/30 EqStr                          ;上面这些字符串依次同序列号比较
40E483: 2f FFree1Str        local_009C
40E486: 1a FFree1Ad         local_0098
40E489: 1c BranchF:         40E48C              ;奇怪的跳转，比较结果为 False
                                                ;就转移到下一句执行
40E48C: 04 FLdRfVar         local_012C
40E48F: Lead3/7e NextStepVar:     (continue) 40E460    ;继续下一轮循环
```

令人意外的是，这个循环对程序的运行路线没有任何影响，鉴定为作者布下的迷魂阵。尽管如此，最后的验证也已经不远了。

```
40E4A2: 0d VCallHresult     get__ipropTEXTEDIT   ;调用函数取得序列号
40E4A7: 3e FLdZeroAd        local_009C
40E4AA: 46 CVarStr          local_00BC           ;转换为变体型
40E4AD: 04 FLdRfVar         local_0094           ;用户名的计算结果 S2 入栈
40E4B0: Lead0/9c SubVar                          ;序列号减去 S2 得到 S3
40E4B4: 04 FLdRfVar         local_0150
40E4B7: 21 FLdPrThis
40E4B8: 0f VCallAd          text
40E4BB: 19 FStAdFunc        local_0174
40E4BE: 08 FLdPr            local_0174
40E4C1: 0d VCallHresult     get__ipropTEXTEDIT   ;调用函数取得用户名
40E4C6: 6c ILdRf            local_0150
40E4C9: 4a FnLenStr                              ;取用户名长度
40E4CA: Lead2/69 CVarI4     local_00AC           ;转换为变体型
40E4CE: 5d HardType
40E4CF: Lead0/33 EqVarBool                       ;是否与 S3 相等
40E4D1: 2f FFree1Str        local_0150
40E4D4: 29 FFreeAd:
40E4DB: 35 FFree1Var        local_00BC
40E4DE: 1c BranchF:         40E55B               ;真正的关键跳转
40E4E1: 27 LitVar_Missing
40E4E4: 27 LitVar_Missing
40E4E7: 3a LitVarStr:       ( local_00CC ) Correct key
40E4EC: 4e FStVarCopyObj    local_00DC
40E4EF: 04 FLdRfVar         local_00DC
40E4F2: f5 LitI4:           0x40  64  (...@)
40E4F7: 3a LitVarStr:       ( local_00AC ) Wow, you have found a correct key!
40E4FC: 4e FStVarCopyObj    local_00BC
40E4FF: 04 FLdRfVar         local_00BC
40E502: 0a ImpAdCallFPR4:       ~
40E507: 36 FFreeVar
40E512: 27 LitVar_Missing
40E515: 27 LitVar_Missing
40E518: 3a LitVarStr:       ( local_00CC ) Correct key!
40E51D: 4e FStVarCopyObj    local_00DC
40E520: 04 FLdRfVar         local_00DC
40E523: f5 LitI4:           0x40  64  (...@)
40E528: 3a LitVarStr:       ( local_00AC ) Mail me, how you got it
40E52D: 4e FStVarCopyObj    local_00BC
```

这部分的算法总结：

strlen (用户名) = 序列号 – S2

据此得到正确的注册信息：

注册码 = strlen (" cyclotron ") + 667574632 = 667574641

第9章 .Net平台加解密[①]

本章将简要介绍微软.Net框架下的安全问题，内容基本涵盖.Net下本机Windows程序保护的各个方面，包括强名称、混淆、加壳、加密等常用保护手段，以及相应的逆向方法。

9.1 .Net概述

2002年微软正式发布了.Net的第一个版本，从此，基于.Net平台的各种软件产品逐渐增多，最新版本的Windows Vista已经直接内置了.Net Framework 3.0。由于越来越多的企业及程序开发者将产品定位在该平台上，.Net软件的保护就成为一个不可回避的课题摆在大家面前。本节主要介绍基本概念，读者应将重点放在理解.Net框架的概念和程序的运行方式上。

9.1.1 什么是.Net

.Net是微软设计的独立于操作系统之上的平台，可以将它看成一套虚拟机，无论机器运行的是什么操作系统，只要该系统安装了.Net框架，便可以运行.Net可执行程序，享受基于.Net的各类服务。上句话是从用户角度出发的观点，如果从Windows系统的角度来理解，.Net就是一系列运行于Ring 3层的DLL文件。

对于加解密的学习者，可以从以下三个方面来理解.Net。

（1）统一了编程语言：无论程序是用C#，还是C++，或是VB编写，最终都被编译为.Net中间语言IL；

（2）扩展了PE文件的格式：可执行文件中不再保存机器码，而是IL指令和元数据，部分结构也被改变用于保存.Net的相关信息；

（3）改变了程序的运行方式：Windows不再直接负责程序的运行，而由.Net框架进行管理，框架中的JIT引擎负责在运行时将IL代码即时编译为本地汇编代码再执行。

到本文写作时为止，.Net的正式版包括1.0、1.1、2.0、3.0和3.5。不同版本的.Net框架可以在同一个系统中共存，这是.Net相对传统DLL式程序兼容性的重要改进之一。本章内容以目前最流行的Windows XP加.Net 2.0平台为主，兼顾1.1平台。2.0版之后的.Net内核基本相同，只是新增了一些企业功能。

9.1.2 几个基本概念

学习新平台，总要接触一些新名词，本节就介绍几个与.Net平台加解密关系最为密切的基本概念。如果读完本节仍不是很清楚这些概念的含义，也没有关系，读者将在后面的实践中一步步掌握它们的本质。

---

① 本章由单海波编写。

MSIL：微软中间语言（Microsoft Intermediate Language），大多数时候简称IL。.Net下有很多种高级语言，常见的包括C#、C++/CLI、VB.Net，但无论哪一种语言，最终在编译后都生成IL。IL是.Net唯一能读懂的语言，也是唯一可执行的语言。大多数时候，对.Net程序进行分析和调试，就是对IL语言进行分析和跟踪。由于运行完全受.Net监控，因此IL属于托管（Managed）代码，与之对应的是本机代码如X86/64汇编，被称为非托管（Unmanaged）代码。如同Win32下要掌握汇编一样，.Net下必须掌握IL。

CLR：CLR是Common Language Runtime的简称，中文名叫通用语言运行时。CLR是.Net框架的核心内容之一，可以把它看为一套标准资源，理论上可以被任何.Net程序使用。它包括：面向对象的编程模型、安全模型、类型系统（CTS）、所有.Net基类、程序执行及代码管理等。CLR是IL语言运行的环境，就像Windows是普通PE程序的运行环境一样。在Windows中，整个CLR系统的实现其实就是几个Ring 3层的DLL，比如mscorwks.dll、mscorjit.dll，它们共同的特点是前缀均为mscor。

Metadata与token：在.Net中，元数据（Metadata）描述了一个可执行文件的所有信息，包括版本、类型的各个成员（方法、字段、属性、事件）等。一个文件要成为有效的.Net可执行程序，必须包含正确的元数据定义。由于元数据将所有的程序信息保存在文件中，并很容易被相应的反编译工具读取，所以，对元数据的加密是现有加密软件的重点之一。为了区分各项元数据，需要提供一个单独的标识，这就是token，它是同一程序中区分和定位不同元数据的依据。读者将在“PE 文件结构扩展”一节中详细了解什么是元数据。

JIT：即时编译（Just In-Time Compile），这是.Net运行可执行程序的基本方式，也就是在需要运行的时候，将对应的IL代码编译为本机指令。传入JIT中的是IL代码，导出的是本机代码，所以部分加密软件通过挂钩JIT来进行IL加密，同时又保证程序正常运行。同解释执行的代码相比，JIT的执行效率要高很多。

Assembly和Module：程序集和模块，它们是构成.Net程序的基本元素。两者之间是包含的概念：一个或多个含有可执行代码的Module，加上一些必要的控制信息，构成了一个Assembly。因此，程序集是.Net中可执行程序的基本单元，通常遇到的可执行文件就是一个程序集，包括.exe 和.dll。有时会出现以.netmodule结尾的文件，其中也包含可执行代码，但不含清单（Manifest）信息，因此只是一个模块，而不能单独成为Assembly。

Type和Method：类型和方法，这是面向对象程序设计中的概念。类型是.Net程序构成的基本元素，最常见的类型是类（class），其次还有结构（struct）和枚举（enum）。类型可以有很多成员（member），最重要的成员是方法（method）。方法是代码的基本单元，可以看作面向过程中编程语言中的函数，所有的代码都被定义在某个类型的某个方法中，定位关键方法是.Net解密中的重要步骤之一。

AppDomain：应用程序域，这是.Net独有的概念。.Net中的进程和线程与Win32平台下的不同，.Net中程序运行的基本单位是Assembly，几个Assembly可以构成一个AppDomain，通常代码只能访问本域中的数据，几个域可以运行在一个传统意义的进程下。AppDomain实现的功能有点类似进程，区分方法也很简单，前者是CLR中引入的概念，后者是操作系统（如Windows）的概念。

9.1.3 第一个.Net程序

了解一个新平台的最好方法是亲自写一个小程序，下面就请读者自己动手编写第一个.Net程序。在这之前，确定已经下载并安装了微软.Net Framework SDK，1.1和2.0版均可。C#代码如下：

```
//Code Sample 9.1.3, hello.cs
using System;
class class1
{
    public static void Main()
    {
```

```
        Console.WriteLine("hello, .net fans!");
        Console.ReadLine();
    }
};
```

代码中建立了一个类 class1（因为.Net 是面向对象的平台，所有的代码应定义在某个类中），且建立了一个公共的（public）静态（static）方法 Main 作为入口点，名称上和 Win32 下相似。首行加入了 using 语句来表示本程序引用了 System 空间的类和方法，所有的.Net 程序都会引用这个名称空间。

新建一个文本文件，输入上面的代码并保存为.cs（C#源码的默认扩展名），然后在 SDK 的命令行中键入：csc hello.cs，回车执行，一个.Net 程序便生成了。（除非另外说明，本文所有的代码均在.Net 2.0 下编译通过，工具为 Visual Studio.net 2005 和 SDK 自带的编译器。）用 SDK 自带的反汇编工具 ildasm.exe 看一下它的 IL 代码。

```
//ildasm for Code Sample 9.1.3, hello.exe

.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       19 (0x13)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "hello, .net fans!"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  call       string [mscorlib]System.Console::ReadLine()
  IL_0011:  pop
  IL_0012:  ret
} // end of method class1::Main
```

这便是 C#编译器生成的 IL 代码。看一下代码流程：ldstr 读入字符串，call 调用系统方法 WriteLine 将字符串输出到屏幕，再调用 ReadLine 使程序暂停，最后是 ret 返回指令，中间还有两个空指令 nop。可以看出，同 asm 相比，IL 语言既有相同的指令助记符（如 call、nop），又有新指令（如 ldstr）。严格地说，IL 是一种高级语言，它支持面向对象，且不直接操作内存地址，它的语言特征反映了.Net 框架的实现原理。

9.2 MSIL 与元数据

MSIL 与元数据是相辅相成的两个概念：IL 语言对元数据进行操作，而其本身又为元数据所定义，两者共同构成了.Net 程序的基本要素。熟练地掌握 IL 与元数据是进入.Net 加解密领域的敲门砖。由于两者同时被存储在 PE 文件中，且 PE 文件又反映了一个系统的基本运行框架，因此下文首先介绍.Net 平台下 PE 文件结构的扩展，其中与 Win32 下重复的内容不再赘述。

本节的重点是理解 MSIL 汇编、元数据与 PE 结构三者间的关系，读者可结合文件结构信息查看工具，边实践边理解本节内容。

9.2.1 PE 结构的扩展

回顾 Win32 平台下的 PE 结构（参考第 10 章），其中有几个 Data Directory 是未被使用的，.Net 对其进行了扩展，其中第 15 项数据目录便指向了 Common Language Runtime header。以 9.1.3 节的 hello.exe 程序为例，在文件偏移 168h 处可以找到表示 CLR 头的数据目录，如图 9.1 所示，即 RVA=2008h，Size=48h。（考虑到大多数读者已经能熟练使用工具查看 PE 结构，因此本小节在介绍结构的过程中，大多直接以十六进制数据为例，便于读者对比分析。）

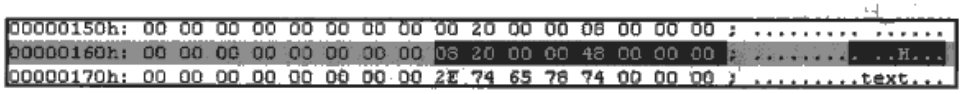


图 9.1 第 15 项数据目录指向 CLR 头

可以发现，这个偏移指向了 exe 文件的.text 区块。.text 区块是.Net 对 PE 结构改变较多的地方，传统 Win32 平台下该节通常只存储 asm 汇编指令，而.Net 中所有的元数据和 IL 代码均存储在.text 区块中，图 9.2 所示是.text 区块的大致结构。

图 9.2 .Net 程序.text 区块的构成

.text 区块中的第二项便是 CLR 头（又可称 CLI 头），该头结构的定义在 SDK 中的 CorHdr.h 里可以找到，名为 IMAGE_COR20_HEADER（无论是 Windows 还是.Net，SDK 中 include 目录里的一些头文件往往是发掘系统隐藏信息的好地方）。精简后的头结构代码如下：

```
// COM+ 2.0 header structure.
typedef struct IMAGE_COR20_HEADER
{
    // Header versioning
    DWORD                   cb;
    WORD                    MajorRuntimeVersion;
    WORD                    MinorRuntimeVersion;

    // Symbol table and startup information
    IMAGE_DATA_DIRECTORY    MetaData;
    DWORD                   Flags;

    union {
        DWORD               EntryPointToken;
        DWORD               EntryPointRVA;
    };

    // Binding information
    IMAGE_DATA_DIRECTORY    Resources;
    IMAGE_DATA_DIRECTORY    StrongNameSignature;

    // Regular fixup and binding information
    IMAGE_DATA_DIRECTORY    CodeManagerTable;
    IMAGE_DATA_DIRECTORY    VTableFixups;
    IMAGE_DATA_DIRECTORY    ExportAddressTableJumps;

    // Precompiled image info (internal use only - set to zero)
    IMAGE_DATA_DIRECTORY    ManagedNativeHeader;

} IMAGE_COR20_HEADER, *PIMAGE_COR20_HEADER;
```

IMAGE_COR20_HEADER 结构的各项说明如表 9-1 所示。

表 9-1 Common Language Header 结构与说明

偏　移	大　小	名　称	说　明
0	4	Cb	CLR 头的大小，以 byte 为单位
4	2	MajorRuntimeVersion	能运行该程序的最小.Net 版本的主版本号
6	2	MinorRuntimeVersion	能运行该程序的.Net 版本的副版本号
8	8	MetaData	元数据的 RVA 和 size
16	4	Flags	属性字段，可以在 IL 中以.corflags 进行显式设置，也可以在编译时用/FLAGS=进行设置，其中命令行设置的优先级较高
20	4	EntryPointToken /EntryPointRVA	入口方法的元数据 ID（也就是 token），作为 exe 文件必须有，dll 文件此项可以为 0（.Net 2.0 中可以是本地入口代码的 RVA 值）
24	8	Resources	托管资源的 RVA 和 size
32	8	StrongNameSignature	强名称的 RVA 和 size，通常用于程序在加载时的版本识别与完整性检验
40	8	CodeManagerTable	CodeManagerTable 的 RVA 与 size，此项暂未使用，为 0
48	8	VTableFixups	v-table 项的 RVA 和 size，主要供使用 v-table 的 C++语言对其进行重定位
56	8	ExportAddressTableJumps	用于 C++的输出跳转地址表的 RVA 和 size，大多数情况为 0
64	8	ManagedNativeHeader	仅在由 ngen 生成本地模块中该项不为 0，其余情况均为 0

其中 Flags 项定义了该 exe 文件的最基本性质，包含以下设置：

```
COMIMAGE_FLAGS_ILONLY             =0x00000001,//此程序由纯 IL 代码组成
COMIMAGE_FLAGS_32BITREQUIRED      =0x00000002,//此程序仅在 32 位系统上运行
COMIMAGE_FLAGS_IL_LIBRARY         =0x00000004,//此程序仅作为 IL 代码库（很少用）
COMIMAGE_FLAGS_STRONGNAMESIGNED   =0x00000008,//此程序有强名称（重要）
COMIMAGE_FLAGS_NATIVE_ENTRYPOINT =0x00000008,//此程序入口方法为非托管
COMIMAGE_FLAGS_TRACKDEBUGDATA     =0x00010000,//loader 和 JIT 需要追踪调试信息
```

图 9.3 所示为 hello.exe 的 CLR 头数据，请读者自行与表 9-1 中各项进行对照。

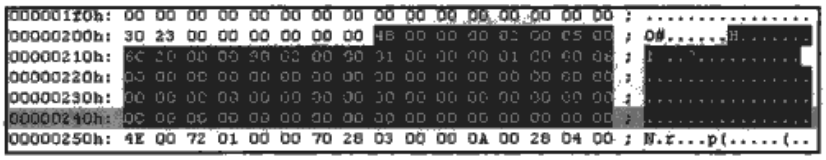


图 9.3　hello.exe 的 CLR 头数据

根据表中最重要的 MetaData 项，来查看元数据在 PE 文件中的存储格式。在示例程序中，元数据的 RVA 和大小分别是 206Ch 和 290h，转换为文件偏移就是 26Ch 处开始。注意到这个地址与刚才 CLR 头的结束地址 24Fh 之前还有一点空隙，IL 代码正是存储在这段空间里。

MetaData 结构以一个元数据头开始，代表元数据的定义由此开始。看一下官方对 MetaData Root 的定义：

```
struct STORAGESIGNATURE
{
    ULONG       lSignature;             // "Magic" signature
    USHORT      iMajorVer;              // Major file version
    USHORT      iMinorVer;              // Minor file version
    ULONG       iExtraData;             // Offset to next structure of information
    ULONG       iVersionString;         // Length of version string
};
typedef STORAGESIGNATURE UNALIGNED * PSTORAGESIGNATURE;
struct STORAGEHEADER
```

```
{
    BYTE        fFlags;                 // STGHDR_xxx flags
    BYTE        pad;
    USHORT      iStreams;               // How many streams are there
};
```

表 9-2 给出了元数据头中各项的意义。

表 9-2 MetaData Root 结构与说明

偏 移	大 小	名 称	说 明
0	4	lSignature	424A5342h，就是 4 个固定的 ASCII 码“BSJB”（BSJB 是 4 名.Net 创始人的字母缩写）
4	2	iMajorVersion	元数据的主版本，一般为 1
6	2	iMinorVersion	元数据的副版本，一般为 1
8	4	iExtraData	保留，为 0
12	4	iVersionString	接下来的版本字符串的长度，包含尾部 0，且按 4 字节对齐（例子中为 0Ch）
16	12	pVersion	UTF8 格式的编译环境版本号（例子中为“v2.0.50727”，长度 10+2）
28	1	fFlags	保留，为 0
29	1	[Padding]	此字节无意义，对齐用
30	2	iStreams	Streams 的个数

紧接着元数据头便是几个流数据的头，流按存储结构的不同分为堆（heap）和表（table），上述头信息指明了这些堆和表的位置及大小。.Net 中共有以下几种流，但不是每个文件中都包含所有的流。读者可以先学习#~、#Strings 和#US 流，因为这三种流几乎在每个.Net 程序中都会出现，并且和加解密的关系最为密切。随着学习的深入，再掌握#Blob 流的结构。

- #Strings

UTF8 格式的字符串堆，包含各种元数据的名称（比如类名、方法名、成员名、参数名等）。流的首部总有一个 0 作为空字符串，各字符串以 0 表示结尾。CLR 中这些名称的最大长度是 1024。

- #Blob

二进制数据堆，存储程序中的非字符串信息，比如常量值、方法的 signature、PublicKey 等。每个数据的长度由该数据的前 1～3 位决定：0 表示长度 1 字节，10 表示长度 2 字节，110 表示长度 4 字节。

- #GUID

存储所有的全局唯一标识（Global Unique Identifier）。

- #US

以 Unicode 格式存放的 IL 代码中使用的用户字符串（User String），比如 ldstr 调用的字符串。

- #~

元数据表流，最重要的流，几乎所有的元数据信息都以表的形式保存于此。每个.Net 程序都必须包含。

- #-

#~的未压缩（或称为未优化）存储，不常见。

每种流都具有共同的头结构，如表 9-3 所示。

表 9-3 Stream header 结构与说明

偏 移	大 小	名 称	说 明
0	4	iOffset	该流的存储位置相对 MetaData Root 的偏移
4	4	iSize	该流占多少字节
8	不定	rcName	流的名称，与 4 字节对齐（如例子中“#~”尾部应有两个字节的 0）

相应的 C++代码如下：

```
struct STORAGESTREAM
{
    ULONG       iOffset;                // Offset in file for this stream
    ULONG       iSize;                  // Size of the file
    char        rcName[32];             // Start of name, null terminated
};
```

示例 hello.exe 中共含 5 个流，没有“#-”流。以“#~”为例，从图 9.4 中看出，它的偏移是 6Ch，大小是 E8h，也就是开始地址为 26Ch+6Ch=2D8h，正好位于最后一个流“#Blob”与 4 对齐后的位置。也就是说，紧跟着 Stream 头定义的便是“#~”流的内容。由于“#~”流是最重要的元数据存储区域，因此下文仅就该流的结构继续深入，而其余各流的结构则由读者自己查阅资料进行学习。

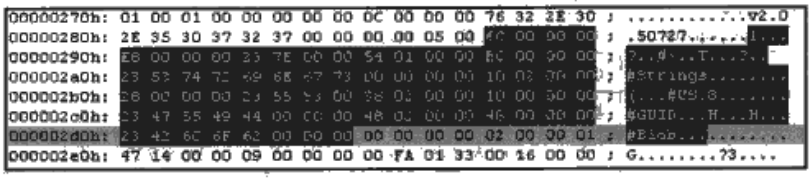


图 9.4　Stream 头的数据

示例中 2D8h 处指向的是“#~”流，也就是元数据表流，该处数据按表 9-4 中的结构进行组织，其中的内容包括了元数据中有哪些表，各个表的性质等。

表 9-4　Metadata Table Stream 结构与说明

偏　移	大　小	名　称	说　明
0	4	Reserved	保留，为 0
4	1	Major	元数据表的主版本号，与.Net 主版本号一致（例子中为 2）
5	1	Minor	元数据表的副版本号，一般为 0
6	1	Heaps	heap 中定位数据时的索引的大小，为 0 表示 16 位索引值，若堆中数据超出 16 位数据表示范围，则使用 32 位索引值。01h 代表 strings 堆，02h 代表 GUID 堆，04h 代表 blob 堆 （在#-流中可以为 20h 或 80h，前者代表流中包含在 Edit-and-Continue 的调试中修改的数据，后者表示元数据中个别项被标识为“已删除”）
7	1	Rid	所有元数据表中记录的最大索引值，在运行时由.Net 计算，文件中通常为 1
8	8	MaskValid	8 字节长度的掩码，每个位代表一个表，为 1 表示该表有效，为 0 表示无该表
16	8	Sorted	8 字节长度的掩码，每个位代表一个表，为 1 表示该表已排序，反之为 0

同样，该结构也有相应的 C++定义：

```
struct MDStreamHeader
{
    DWORD   Reserved;
    BYTE    Major;
    BYTE    Minor;
    BYTE    Heaps;
    BYTE    Rid;
    ULONGLONG   MaskValid;
    ULONGLONG   Sorted;
};
```

由于 Valid 项长度为 8 字节，因此很容易得出.Net 中最多可定义 8×8=64 个表，而实际上.Net 已定义的只有 45 个表，见表 9-5。判断 Valid 被置 1 的二进制位，便可以得出该程序中使用了哪些表。以 hello.exe 为例，由于 int64 在内存中以反字节顺序存储，可得 valid=0000000900001447h，相对应的为 Module, TypeRef, TypeDef, MethodDef, MemberRef, CustomAttribute, Assembly 和 AssemblyRef 这 8 个表。

表 9-5 元数据中所有的表（斜体为.Net 2.0 新增的）

00 – Module	01 – TypeRef	02 – TypeDef
03 – FieldPtr	04 – Field	05 – MethodPtr
06 – MethodDef	07 – ParamPtr	08 – Param
09 – InterfaceImpl	10 – MemberRef	11 – Constant
12 – CustomAttribute	13 – FieldMarshal	14 – DeclSecurity
15 – ClassLayout	16 – FieldLayout	17 – StandAloneSig
18 – EventMap	19 – EventPtr	20 – Event
21 – PropertyMap	22 – PropertyPtr	23 – Property
24 – MethodSemantics	25 – MethodImpl	26 – ModuleRef
27 – TypeSpec	28 – ImplMap	29 – FieldRVA
30 – ENCLog	31 – ENCMap	32 – Assembly
33 – AssemblyProcessor	34 – AssemblyOS	35 – AssemblyRef
36 – AssemblyRefProcessor	37 – AssemblyRefOS	38 – File
39 – ExportedType	40 – ManifestResource	41 – NestedClass
42 – *GenericParam*	43 – *MethodSpec*	44 – *GenericParamConstraint*

紧接着 Metadata Table Stream 头的是一串 4 字节数组，每个双字代表该表中有多少项记录（record），8 个表共 32 个字节。然后便开始各表的数据了，排在第一位的自然是 Module 表。44 个表各有各的结构，为节省篇幅，在这里只介绍比较典型的 Assembly 和 MethodDef 表，其余表结构请读者自行查阅资料。

Assembly 表主要定义了该程序集的基本性质，其结构见表 9-6，注意到表中最后三项索引值为 2（4），表示 2 或 4 字节，这正是由 Metadata Table Stream 头中的 Heaps 项决定的，由于示例程序中该项为 0，因此所有的索引值大小均为 2 字节。

表 9-6 Assembly Table 结构与说明

偏 移	大 小	名 称	说 明
0	4	HashAlgId	对该程序进行 hash 的算法，一般为 0x8004，表示 CALG_SHA/CALG_SHA1（CALG_* 为前缀的算法定义在 wincrypt.h 文件中）
4	2	MajorVersion	该 Assembly 的主版本号
6	2	MinorVersion	该 Assembly 的副版本号
8	2	BuildNumber	该 Assembly 的编译号
10	2	RevisionNumber	该 Assembly 的修订号
12	4	Flags	属性，主要决定 Assembly 的运行方式，包括一个强名称标识
16	2(4)	PublicKey	如有强名称，该项表示强名称数据在#Blob 中的偏移，反之该项为 0
	2(4)	Name	指向#Strings 流中的偏移，表示 Assembly 的名称，不含路径和扩展名
	2(4)	Locale	指向#Strings 流中的偏移，表示 Assembly 的地域和语言，如“en-US”、“fr-CA”

MethodDef 是另一个很重要也很有趣的表，因为它不但指出了该方法 IL 代码的位置，还限定了方法的属性，以及该方法如何被调用。MethodDef 结构见表 9-7。

表 9-7 MethodDef Table 结构与说明

偏 移	大 小	名 称	说 明
0	4	RVA	该方法体的 RVA（方法体包括：方法头、IL 代码、异常处理定义）
4	2	ImplFlags	限定了方法的执行方式（如 abstract、P/Invoke 等）
6	2	Flags	限定了方法的调用属性和其他一些性质（如 public、private、virtual 等）
8	2(4)	Name	指向#Strings 的偏移，表示该方法的名称
	2(4)	Signature	指向#Blob 的偏移，Signature 定义了方法的调用方式（如返回值类型，Calling Convention 等）
	2	ParamList	指向 Param 表的索引，指出了方法的参数

如果读者是第一次接触元数据和.Net 的扩展 PE 结构，则最好使用十六进制编辑器，与表结构一项项

对照。而在平时的分析和逆向过程中，有多种工具提供了直接浏览 PE 结构和元数据的功能，如 Spices.Net、Dotnet Explorer、Researcher.NET 和 CFF Explorer 等。一般的元数据查询与修改使用 CFF Explorer 比较方便，图 9.5 中显示了用该工具读入 hello.exe 后以树状结构显示的元数据。

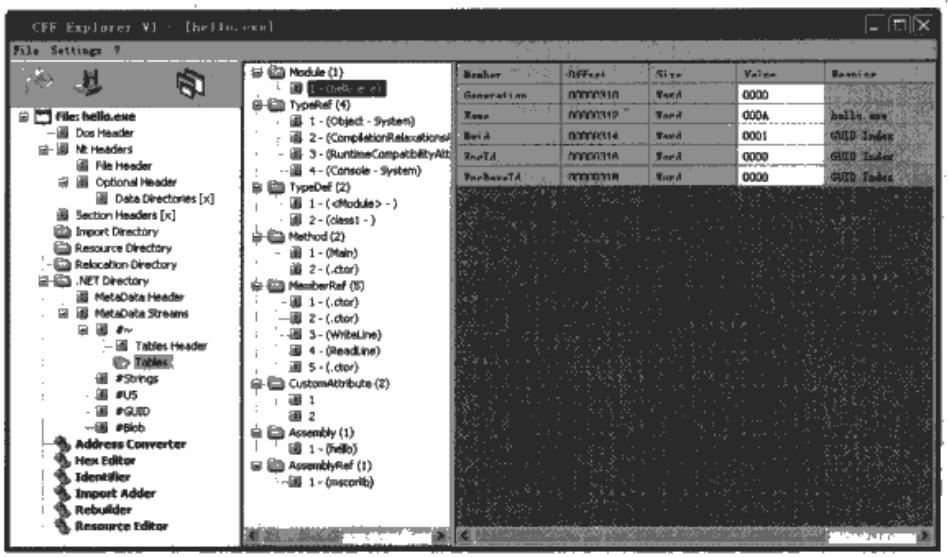


图 9.5　用 CFF Explorer 查看.Net PE 的元数据

关于元数据最权威的参考，莫过于 ilasm 的作者写的《Inside Microsoft .NET IL Assembler》，对应于 2.0 平台的第二版名叫《Expert .NET 2.0 IL Assembler》，本节未详述内容均可在该书中找到，请读者自行参考。

9.2.2　.Net 下的汇编 MSIL

.Net 平台下的程序，无论开发时使用哪种高级语言，最终都被编译为微软中间语言 MSIL。IL 与上述几种高级语言相比，显得更加底层，因此又被称为 IL 汇编。而实际上 IL 与 asm 汇编相比，又算得上高级语言，最明显的特征莫过于 IL 不直接和内存地址打交道。在.Net 平台刚出现时，各种加密手段还不成熟，所以.Net 下的破解基本上就是对 IL 反汇编代码进行阅读，随后写出注册机或直接在 IL 代码中“爆破”。现在，.Net 下的加密手段越来越复杂，但 IL 代码仍是最重要的突破口之一。对于加解密来说，IL 就像 Win32 下的 asm，不是可学可不学，而是必须掌握的知识。（C++/CLI 混合编译的程序，其中既可保存托管代码，又可保存本地代码，不属于本章讨论范围。）

先来看一个例子，新建一个文本文档，键入以下代码后，保存为.il 文件。在 SDK 的命令行里进行编译：ilasm sample922.il，最终会生成一个可执行文件。

```
//9.2.2节示例代码
.assembly extern mscorlib{}
.assembly sample1022{}
.module   sample1022.exe

.class public auto ansi class1 extends System.Object
{
   .field private static int32 sum
   .method public static void addTwoInts()
   {
      .entrypoint
      .locals init(int32 val)
      ldc.i4.1
      stloc.0     //也可写成 stloc val
      ldc.i4 10
      ldloc.0     //也可写成 ldloc val
```

```
        add
        stsfld int32 class1::sum
        ldsfld int32 class1::sum
        ldstr "1+10="
        call void [mscorlib]System.Console::WriteLine(string)
        call void [mscorlib]System.Console::WriteLine(int32)
        nop
        ret
    }
}
```

该段代码的功能是分两行输出“1+10=11”这个字符串。寥寥20多行代码，包含了IL语言的最基本要素：

（1）IL源文件扩展名为.il。

（2）IL源文件中，用“//”号表示行注释，还可以用“/* */”表示注释块。

（3）一个.exe文件必须有入口。IL源文件中，入口方法名不一定要求为Main（还记得C#吗），而用.entrypoint表示。

（4）.assembly定义本程序集，而.assembly extern则定义被引用的程序集，两者分别对应元数据表中的Assembly与AssemblyRef。mscorlib是所有.Net程序的基础，每个程序都会引用它。

（5）所有的代码必须定义在某个类的某个方法中，代码中addTwoInts方法就定义在class1类中。

（6）对于本地变量（由.locals定义），可以采用名称引用，也可以用序号表示。代码中只有一个本地变量val，因此在取val的值时，既可以用ldloc val，也可以用ldloc.0。

（7）读入常数值时，对于0～8，可以直接用简短指令，形如ldc.i4.1；而对大于8值的数值，如程序中的10，则必须用完整指令，如ldc.i4 10。

（8）调用某个方法时，必须完整地写出方法的返回值、空间名、类名，最后才是方法名，以及方法的参数。

（9）IL中也有空指令nop，不过它的十六进制编码是00h，而不是Win32 asm中的90h。

IL语言最大的特点是以堆栈为基础进行操作，通常不直接操作寄存器和内存，因此学习难度相比Win32 asm要低。下面模拟示例代码的执行（见图9.6），来解释什么叫以堆栈为基础操作。示例图中小箭头指向当前栈顶，堆栈的生长方向由下往上，堆栈的名称叫evaluation stack，这是.Net内核给出的逻辑概念。

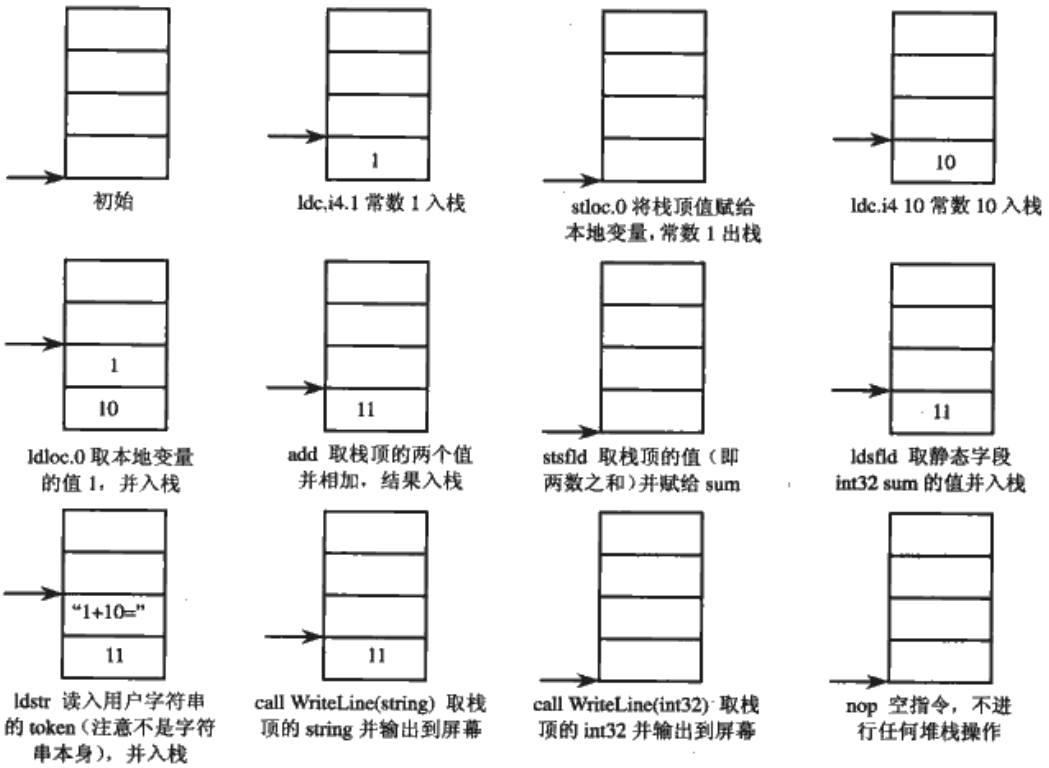


图9.6 IL代码操作堆栈流程示意图

图 9.6 给出了两个信息：一是有效的 IL 程序，必须保证堆栈的平衡，在方法开始时堆栈若为空，则在方法结束时堆栈也必须为空，这一点与 Win32 下的堆栈平衡有点类似；二是部分 IL 指令直接对堆栈进行操作，比如在堆栈中取操作数，或是将计算结果压入堆栈。其中第二点与 Win32 下 asm 指令相差比较大。以跳转指令为例，asm 中的 jz 指令判断的是 CPU 的 Zero 标志位，堆栈数据对它没有影响；而 IL 中除了 br 和 br.s 外（直接跳转），其他以字母 b 打头的条件跳转均要求将栈顶的一个或两个元素取出并进行比较，指令结束后的堆栈已经被修改。所以，喜欢“爆破”.Net 程序的读者应该注意，在进行跳转指令 patch 的时候，一定要注意堆栈的平衡。

解释完 IL 的堆栈操作原理，这门语言就已经学了一半，剩下一半是记忆各类指令助记符、关键字和代码格式。这里不再将所有的指令列出，而是在后文的分析过程中逐步介绍出现的指令及其具体功能。表 9-8 中列出了绝大部分 IL 指令所用的英文名称缩写，上半部分为操作数简称，下半部分为操作符简称，读者只需记住这些基本的英文单词，就可以对所有 IL 指令做到望文生义。

表 9-8 IL 代码英文缩写意义

	缩 写	展 开	意 义	缩 写	展 开	意 义
操作数	i1	int8	1 字节有符号整型数	i4	int32	4 字节有符号整型数
	i8	int64	8 字节有符号整型数	u1	uint8	1 字节无符号整型数
	u4	uint32	4 字节无符号整型数	f4	float32	4 字节浮点数
	f8	float64	8 字节浮点数			
操作符	ld	load	读入	st	set	赋值
	loc	local	本地变量	arg	argument	方法的参数
	s	short	短指令	c	const	常量
	inst	instance	（对象的）实例	gt	great than	大于
	lt	less than	小于	eq	equal	等于
	ne	not equal	不等于	br 和 b	branch	跳转
操作符	ind	indirect	非直接（间接，即取变量地址）	conv	convert	转换
	virt	virtual	虚的（虚函数）	fld	field	操作对象是 field
	a	address	（变量的）地址	elem	element	元素（指 array 类型）
	ovf	over flow	带溢出的	ftn	function pointer	方法（函数）的指针

很少会有开发人员直接使用 IL 编程，但对于研究.Net 底层的解密爱好者来说，使用 IL 编程的机会相对较多，比如直接利用 ildasm 反编译源程序，并使用其中的代码编写注册机。由于 ilasm 编译器提供的纠错功能很弱，因此就算程序顺利编译，也可能在运行时报错。有三种方法可避免此问题，一是尽量正确地使用 IL，不让源代码中出现错误或少出错误；二是利用 SDK 中的辅助工具 Peverify.exe，它不但可以验证 IL 代码的正确性，还可以验证程序元数据的有效性；三是利用第三方工具，比如 Spices.net 中的 Pe Verify 功能。

9.2.3 MSIL 与元数据的结合

IL 以元数据为操作对象，同时本身的执行又受到元数据的限定，两者的关系密不可分。元数据在 IL 中通过 token 引用和定位，token 是元数据项的唯一标识。下面，用 ildasm 对 9.1.3 节的 hello 示例程序进行解码，看一下元数据在 IL 中的表示。注意将 View 菜单中的“Show bytes”和“Show token values”两项选中，以便直接观察指令的十六进制数据和各个 token 值。

```
.method /*06000001*/ public hidebysig static
void Main() cil managed
```

```
// SIG: 00 00 01
{
 .entrypoint
 // Method begins at RVA 0x2050
 // Code size       19 (0x13)

 .maxstack  8
 IL_0000:  /*00|                  */ nop
 IL_0001:  /*72| (70)000001*/ ldstr      "hello, .net fans!" /* 70000001 */
 IL_0006:  /*28| (0A)000003*/ call       void
[mscorlib]/*23000001*/]System.Console/*1000004*/::WriteLine(string)/*A000003*/
 IL_000b:  /*00|                  */ nop
 IL_000c:  /*28| (0A)000004*/ call       string
[mscorlib/*23000001*/]System.Console/*01000004*/::ReadLine() /* 0A000004 */
 IL_0011:  /*26   |              */ pop
 IL_0012:  /*2A   |              */ ret
} // end of method class1::Main
```

从上面的代码看出，token 值实际上就是一个 uint32 数值 AABBBBBBh，其中前一个字节 AA 指出了它对应的表，后三个字节 BBBBBB 指出了它在表中的位置，也就是记录索引（RID，Record Index）。将本段代码出现的所有 token（被包含在“/* */”中）列出表来对比一下就很清楚了，如表 9-9 所示。

表 9-9 代码中出现的 token 值及其对应的表

token	表 值	表 类 型	索 引 值	说 明
0x06000001	0x06	MethodDef 表	0x000001	方法定义表中定义的第 1 个方法 Main()
0x23000001	0x23	AssemblyRef 表	0x000001	Assembly 引用表中定义的第 1 个 Assembly 引用：mscorlib
0x01000004	0x01	TypeRef 表	0x000004	类型引用表中定义的第 4 个引用类型：System.Console
0x0A000003	0x0A	MemberRef 表	0x000003	成员引用表中定义的第 3 个成员（方法）：WriteLine
0x0A000004	0x0A	MemberRef 表	0x000004	成员引用表中定义的第 4 个成员（方法）：ReadLine

细心的读者会注意到，这里还有一个 token 没有提到，就是 70000001h，该值比较特殊，因为 70h 没有任何对应的表。这里只要记住，以 70h 开头的 token 对应的都是用户字符串，后三个字节 000001h 对应该字符串在#US 流中的偏移（注意这里不是索引）。用户字符串流中偏移 1 处便是“hello, .net fans!”，这便是 ldstr 指令读入的值。这也说明了 IL 为什么是一种高级语言，因为它不是直接和内存地址打交道，而是采用 token 的方式（还记得 Win32 编程中的指针吗，那些指针都直接代表内存地址）。除上述已经出现的表外，其余表均按前文所列的顺序进行编码，比如 1Bh 开头的 token 值对应十进制 27 位的表 TypeSpec。值得注意的是，45 个表中只有 23 个表可以用 token 表示，剩余 22 个只用于内部使用，在 MSIL 中是不可使用的。

介绍完 token 的概念，下面再介绍一下什么是签名（signature）。回顾上面代码中的方法名下方有这样一行注释：

```
// SIG: 00 00 01
```

这行注释表示了 Main 方法的 signature 是 00 00 01h。从 PE 结构上看，signature 就是存储在#Blob 中的一段二进制数据，它的作用是描述特定元数据的性质。.Net 中共有 6 种表引用了 signature，分别是 Field、MethodDef、Property、MemberRef、StandAlongSig 和 TypeSpec。代码中关于 Main 方法的//SIG: 00 00 01 按如下方法解码：

第一个 00：任何 signature 的第一个字节都代表 calling conventions，它定义了该 sig 的类型，是 method、field 或是 property。00h 代表 IMAGE_CEE_CS_CALLCONV_DEFAULT，意思就是普通（默认）的方法，含定长的参数列表。

第二个 00：代表方法中的参数个数，Main 方法没有参数，因此为 0。

第三个 01：代表方法的返回值类型，其中 ELEMENT_TYPE_VOID=01h。

这样，一个 method 便被 token 和 sig 完全确定了：方法的代码根据 token 在相应表中查找，而方法的调用方式、参数个数及返回值类型被其相应的 signature 所限定。所有的 sig 数据保存在#Blob 流中，在通常面对一般的保护时，并不需要用到 sig 解码，但一旦深入.Net 核心（比如进行脱壳后的文件修复时），就会遇到自行解码 sig 的情况了。其他关于 sig 解码的详细帮助，请参阅 SDK 中 Tool Develop Guide 文档。

注意：.Net 中还有一种 RID 叫 Record Identifier，记录标识，该标识仅用于.Net 内部。若无特殊说明，本章中所用 RID 均指 Record Index，即记录索引。

9.3 代码分析技术

到这里，读者已经掌握了分析.Net 程序必需的基础知识，下面具体介绍几种.Net 程序代码分析技术。与 Win32 类似，.Net 下的代码分析也可分为静态和动态两种，本节就这两种方法分别加以介绍，目的是让读者掌握两种方法的原理及相应工具的使用，并能融会贯通。本节的示例程序是一个未加任何保护的 CrackMe。

9.3.1 静态分析

静态分析就是用反编译工具将程序的指令字节反编译成 IL 指令或高级语言，通过阅读反编译代码掌握程序的流程与功能。由于.Net 下的可执行文件同时保存有元数据与 IL 代码，其静态反编译出的代码具有极强的可读性，几乎等同于源代码。

.Net 平台的反编译工具较多，其中最基础也最强大的便是.Net 框架 SDK 自带的 ildasm。毕竟是微软自己的产品，反编译出的代码也是最权威的。更重要的是，由 ildasm 得到的 IL 代码，经过修改，便可以用 ilasm.exe 再编译成可执行程序，这是.Net 下 patch 的常用方式之一，微软官方对这一过程的命名叫 round-tripping。

但最好用的反编译工具并不是 ildasm，而是 Reflector，因为后者可以将反编译出的 IL 代码转换为高级语言，比如 C#、VB.Net 等，高级语言的可读性实在比 IL 代码强很多。因此，下文主要介绍 Reflector 的使用。

运行 Reflector，载入 crackme.exe 后，左边是以树状结构显示的文件结构及当前程序域中的所有 Assembly，而双击某节点则会在右边显示该项的反编译代码，上方的下拉框用以选择代码的格式。为什么这里载入的是 crackme.exe，但节点显示程序的名称却是 WindowsApplication1？还记得在 MSIL 的章节中介绍的.assembly 关键字吗，这里的名字正是由关键字定义的程序集名称，而与文件名无关。

看一下程序入口点。在 WindowsApplication1 上单击右键，选择“Go to Entry Point”，直接来到程序的入口，这里是 Main 方法。双击 Main 节点，会在右方显示它的反编译代码，切换为 C#（如图 9.7 所示）。

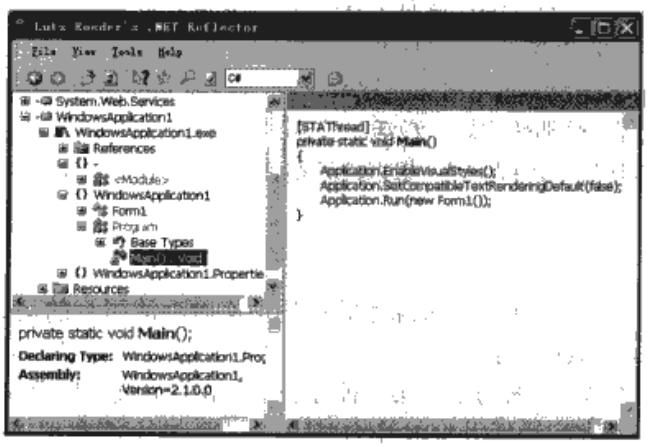


图 9.7　Reflector 的主界面

静态分析的主要任务是定位关键代码，分析程序流程。怎样在静态分析中定位计算注册码的关键方法呢？对于小程序，比如这个 CrackMe，可以在树状结构中浏览并一一查看，对于拥有成千上万个方法的大程序来说，这几乎是不可能完成的任务。这时，需要使用 Reflector 的查找功能，查找的目标包括含敏感名称的方法、类型或者用户字符串。哪些字符串含敏感信息呢？如“activate”、“register”、“user name”、“password”和“crypt”等，这些字符串都有可能成为定位关键代码的突破口。示例中的 CrackMe 在注册成功时会显示“Congratulaions”，因此就将它作为目标进行查找。（程序员也应该注意，尽量不要在程序中直接使用未经加密的字符串！）

在主界面上按 F3 键，进入查找界面，输入“congratulations”，并将查找类型切换为 String（搜索字符串），搜索结果会告诉我们哪些方法调用了这个字符串。结果显示只有一个方法，就是 crypt1。双击该结果，左边树控件中会将 crypt1 节点高亮显示，再次双击则会反编译出它的代码，如图 9.8 所示。

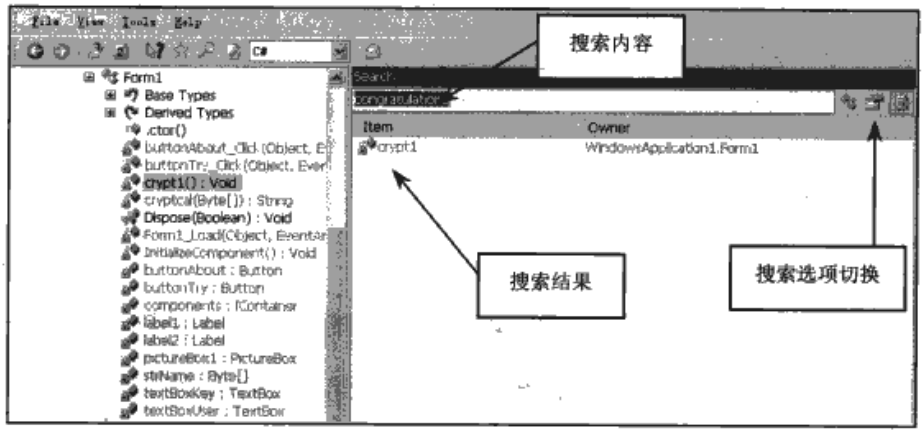


图 9.8　Reflector 的搜索功能

Reflector 另一项很有用的功能是分析（Analyse），即分析被哪些项调用，而自身又调用了哪些项。接着上面的搜索结果，来到 crypt1 方法处，在节点上单击右键，选择“Analyse”，右边便显示出分析结果，其中显示出 buttonTry 的方法中调用了 crypt1。在该项上单击右键，选择“Go to member”，来到该方法处，双击后便显示出 buttonTry 的详细代码。Analyse 功能还可用在系统节点上，比如某程序采用了网络验证，便可以在 WebRequest 上进行分析，查看程序中哪些方法调用了网络功能。

Reflector 还集成了许多插件，用以扩充它的功能，甚至包括了调试插件，这些都可以在作者的主页上（http://www.aisto.com）下载。Reflector 如此强大和普及，使它成为了最容易被 Anti 的反编译软件，后文也将演示一些 Anti 方法。在这种情况下，就需要使用其他的工具来辅助静态分析。除 Reflector 之外，反编译工具还包括 9rays Spices.net、Dis#、Decompiler .net、Xenocode Fox，这些软件都集成了除反编译外的其他许多功能，比如 Metadata 查看、反名称混淆功能等。

关于 ildasm，由于它是最基本的反编译软件，因此众多混淆软件提供了 Anti 的功能，比如 Spices.net 混淆器。看雪论坛上已经有文章探讨过该原理，在 2.0 版的.Net 中，只要代码中设置了 System.Runtime.CompilerServices.SuppressIldasmAttribute 属性，便会造成 ildasm 拒绝反编译。

一般说来，定位出关键代码（注册码比较、激活、验证）的位置，程序就分析了一半。但通常情况是算法较复杂，特别是现在的程序几乎都经过了流程混淆，想直接静态分析出整个计算流程很难。这时，便需运用动态调试。

9.3.2 动态调试

运用调试工具对程序进行调试，便可以跟踪运算过程，实时观测指令的运算结果，若是注册码完整地出现在内存中，也可以在调试工具中直接显示。

结合不同的工具，.Net 下调试有几种方法，第一种方法是用 ildasm 将程序反编译为.il 文件，再用 ilasm.exe 带上/DEBUG 信息编译回可执行程序，最后用 SDK 中自带的 GuiDbg 进行调试；第二种方法是用 WinDbg 配合.Net 调试扩展 SOS，最大的优势是可以直接利用微软的各种符号资源，但 WinDbg 的使用较复杂，不便于新手入门（WinDbg 在.Net 内核调试中显示出强大的功能，读者可自行测试）；第三种方法是使用直接调试的工具，如 PEBrowseDbg 和 OllyDbg 等。这里介绍使用 PEBrowseDbg 的方法，它的最大特点，一是方便，直接打开程序便开始调试；二是支持 inter-op 调试，IL 与 asm 代码同时显示；三是断点功能强大。

先来看看 PEBrowseDbg 的基本功能。PEBrowseDbg 提供了表 9-10 所示的断点类型。

表 9-10 PEBrowseDbg 支持的断点类型

断点类型	说　明	断点类型	说　明	断点类型	说　明
process initialization	进程初始化时	debug symbols	调试符号处	memory breakpoints	内存断点
module load	模块载入时	JITed(Just-in-Time) methods	JIT 引擎即时编译方法时	conditional breakpoint	条件断点
thread startup	线程初始化时	user specified address	用户指令地址	one-time breakpoint	一次性断点
module exports	模块导出函数				

用 PEBrowseDbg 对刚才的 CrackMe 进行调试，调试时的主界面如图 9.9 所示。直接载入 CrackMe 并运行，程序会中断在 ntdll!LdrInitializeThunk 处，单击继续，程序便中断在 CrackMe 的入口处。此时，左边树状显示区会显示程序域中的所有模块，展开 CrackMe 节点，会看到.Net 下特有的两个节点，一是.NET MetaData，二是.NET Methods。而代码窗口中同时显示出了入口方法的 IL 与 asm 代码，深色的一行为当前 EIP 指向，标题栏中则显示当前中断位置。

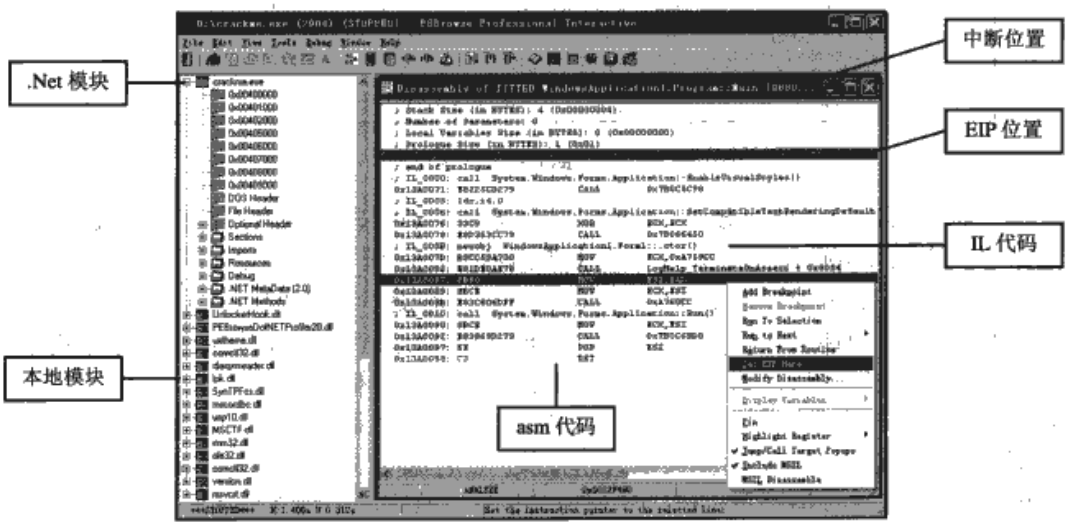


图 9.9 PEBrowseDbg 调试主界面

在代码窗口中单击右键，会显示弹出菜单并提示更多功能，如 Run to Selection，运行到所选中的指令处；Set EIP Here，设置下一条运行指令；Include MSIL，是否同屏幕显示 MSIL 代码和 asm 代码。很多选项在 OllyDbg 中也能够见到，这正是因为.Net 下调试的实质是对 JIT 生成的 asm 代码进行的，其本质和 OllyDbg 调试 Win32 下的程序相同。

将 CrackMe 的.NET Methods 展开，找到 crypt1 节点，并在该方法上下断点。对于这种简单的程序，可以直接通过浏览节点找到对应方法，但对于大型程序则一般通过查找的方法定位某节点，方法是直接在节点的右键菜单中选择“Search From”。

下断点有两种方法，一是对某方法单独下断，二是对某个类型的所有方法下断，图 9.10 分别演示这两种方法的不同操作方式。

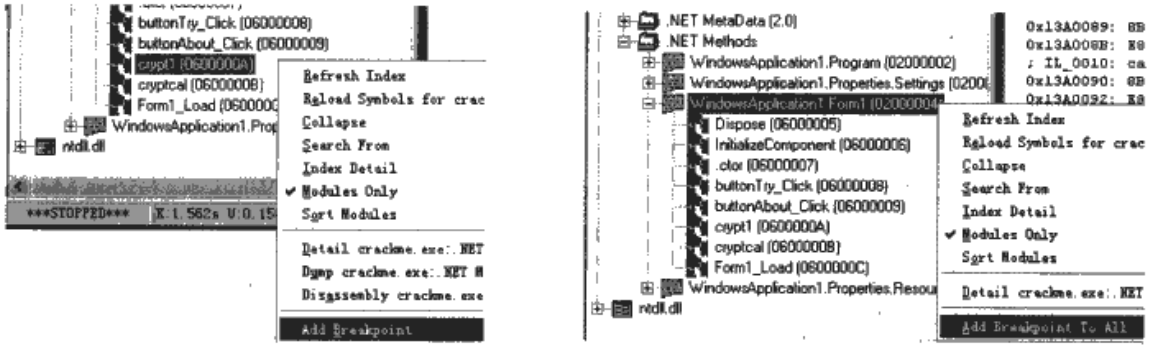


图 9.10　两种下断点的方式（左：单独下断；右：全部下断）

既然在静态分析的过程中已经知道 crypt1 的功能是注册码验证，便可直接在该方法上下单个断点。按 F5 键继续运行后，会显示 CrackMe 的界面，在用户名和注册码框中随意输入数据，单击 try 按钮，便会中断在 crypt1 的入口处。运行到字符串比较的指令 System.String :: op_Equality()处，对应的 asm 代码为 call 0x7934B8F0。

```
; IL_007F: ldloc.0
; IL_0080: ldarg.0
; IL_0081: ldfld textBoxKey
; IL_0086: callvirt  System.Windows.Forms.Control::get_Text()
; IL_008B: call  System.String::op_Equality()
; IL_0090: brfalse.s IL_00A2

0x13A0A50: 8B0424        MOV       EAX,DWORD PTR [ESP]
0x13A0A53: 8B8844010000  MOV       ECX,DWORD PTR [EAX+0x144]
0x13A0A59: 8B01          MOV       EAX,DWORD PTR [ECX]
0x13A0A5B: FF9064010000  CALL      DWORD PTR [EAX+0x164]
0x13A0A61: 8BD0          MOV       EDX,EAX
0x13A0A63: 8BCE          MOV       ECX,ESI
0x13A0A65: E886AEFA77    CALL      0x7934B8F0
0x13A0A6A: 25FF000000    AND       EAX,0xFF
0x13A0A6F: 7411          JZ        0x13A0A82          ; (*+0x13)
```

这时打开寄存器窗口（“View/Registers”），双击 ecx，便会显示 ecx 所指内存的数据。可以发现它正指向了正确的注册码“dzEbDQYDAQA=”。这样，便绕过了复杂的注册码计算过程，直接得到了答案，如图 9.11 所示。

由于注册码的明码完整地出现在内存中，所以可以直接查看。但对于没有在内存中完整出现注册码的程序，动态调试更多地是帮助分析运算过程。开发者也应该尽量避免在代码中直接进行完整注册码明码比较。

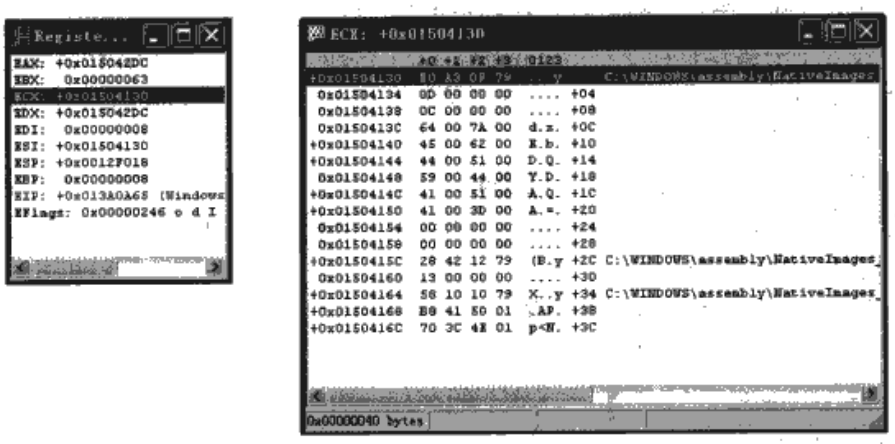


图 9.11　查看 ecx 所指内存中的注册码

在 Win32 下，user32.dll 导出的 API 函数 MessageBox 是常用断点之一，同样，在.Net 下也可以直接对系统方法下断。以 MessageBox 为例，.Net 中 MessageBox 的定义出自 System.Windows.Forms.MessageBox.Show，在 Forms.dll 中查找 MessageBox 类，然后在 Show 方法上下断。图 9.12 显示出该类中有十几个 Show 方法（面向对象程序设计的重载机制），这时便可以用 Add Breakpoints To All，给所有的方法下断，无论程序最终调用哪个，都可以被捕获。

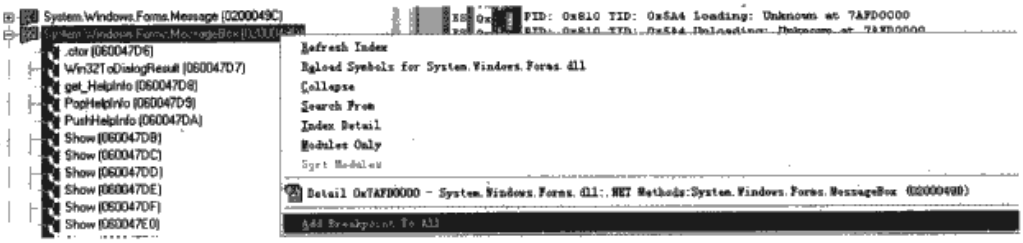


图 9.12　给系统方法 MessageBox.Show 下断点

PEBrowseDbg 还提供了很多的设置功能，不过默认的就已经足够正常使用了。程序调试是很有意思的，三言两语无法涉及所有方面，更多的经验和技巧需要读者在调试过程中自己去总结归纳。

9.3.3　代码修改

.Net 中 PE 文件的 patch 有三种方法：一是用 ildasm 将代码反编译为 IL 文件后，修改 IL 代码，再用 ilasm 编译回可执行文件；二是直接在 PE 文件中查找到 IL 代码对应的数据，在十六进制编辑工具中修改；三是利用工具将反编译代码导出为 C#工程。下面分别以前两种方法为例介绍如何给 CrackMe 打补丁，而第三种方法可用到的场合实在是太少了。

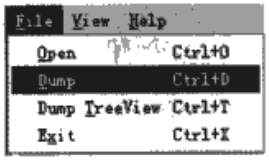


图 9.13　用 ildasm 反编译可执行文件

用 ildasm 反编译 CrackMe，如图 9.13 所示，命名生成文件为 1.il，会在相应目录下生成 4 个文件：1.il 源代码文件，1.res 资源文件，两个以 resources 为扩展名的.Net 资源文件。用编辑器打开 1.il，定位到如下代码：

```
IL_0090:  brfalse.s  IL_00a2
IL_0092:  ldstr      "you get it "
IL_0097:  ldstr      "congratulations"
```

代码的意图很明显：如果判断注册码失败，则brfalse.s跳转，成功则继续执行，并显示MessageBox。修改的方法很简单，不让brfalse跳转，注意前面提到的堆栈平衡原则，所以这里修改为pop而不是nop。

```
IL_0090:  pop   //源代码为brfalse.s  IL_00a2，修改为pop
```

在SDK命令行中进行编译，注意包含资源文件，回车后生成了1.exe。

```
ilasm /resource=1.res 1.il
```

双击运行后，无论在用户名与密码框中输入什么内容，程序直接显示“注册成功”！

第二种方法，直接在PE文件中修改。怎么定位相应的IL代码呢？需要查找IL代码的十六进制字节。在ildasm的View菜单中打开“Show bytes”选项，双击某个方法，会显示该方法每条IL代码的字节。判断注册的关键代码如下：

```
IL_0090:  /* 2C   | 10                */ brfalse.s  IL_00a2
IL_0092:  /* 72   | (70)000288        */ ldstr      "you get it "
IL_0097:  /* 72   | (70)0002A0        */ ldstr      "congratulations"
IL_009c:  /* 28   | (0A)00003F        */ call       valuetype [System.Windows.Forms]
                                        System.Windows.Forms.DialogResult[System.Windows.Forms]
                                        System.Windows.Forms.MessageBox::Show(string,string)
IL_00a1:  /* 26   |                   */ pop
IL_00a2:  /* 2A   |                   */ ret
```

可以查找2C 10 72 88 02 00 70，查找数据的多少以能唯一定位为标准。这里要注意intel字节反转顺序，(70)000288在文件中的字节顺序反转为88 02 00 70。查找到以后，便可直接将2C 10改为26（pop）00（nop），如图9.14所示。保存后运行，和第一种方法效果相同。

```
000015a0h: 32 D4 02 02 7B 09 00 00 04 28 08 00 00 06 0A 06 ; 2?.{....(......
000015b0h: 02 7B 03 00 00 04 6F 3A 00 00 0A 28 3F 00 00 0A ; .{....o:...(?...
000015c0h: 2C 10 72 88 02 00 70 72 A0 02 00 70 28 3C 00 00 ; ,.r?..pr?.p(<..
000015d0h: 0A 26 2A 00 13 30 01 00 09 00 00 00 01 00 00 11 ; .&*..0.........
000015e0h: 03 28 40 00 00 0A 0A 06 2A 06 2A 1A 7E 0A 00 00 ; .(@.....*.*.~...
```

图9.14 在UltraEdit中查找IL代码

有时，无论采取哪种补丁方式，都会造成程序运行失败，这时很有可能是程序中运用了强名称或其他一些保护方式，当文件被修改后，这些保护方式会产生异常，从而导致程序无法正常启动。

9.4 代码保护技术及其逆向

本节主要分类介绍笔者完稿时为止出现的各种.Net代码保护技术，内容包括它们的实现原理、保护效果及拆解方法。

9.4.1 强名称

强名称（StrongName）是.Net提供的一种验证机制，主要功能有两个：标识版本和标识原作者。前一个功能用来弥补Windows中DLL机制的缺陷，因为不同版本的DLL，只要强名称不同，便可在.Net中共存；后一个功能主要帮助用户验证自己得到的程序是否为原作者所写，而没有被人修改过（如添加恶意代码）。强名称的原理和Win32下的自校验有点类似，也是利用特定的算法对程序进行hash计算，但检验过程由.Net平台实施。在.Net平台刚诞生时，强名称更多地被人当作一种代码保护机制运用：保证自己的程序不被patch。

先来看一看怎么给程序签署强名称。写一个简单的 C#程序，在文本框中键入如下代码后保存为 sample1041.cs。由于 i 的值始终为 1，因此输出永远都是“i=1”。

```
//code for sample1041
using System;

namespace tankaiha.sample1041
{
    class class1
    {
        public static void Main()
        {
            int i=1;
            if(i==1)
            {
                Console.WriteLine("i="+i.ToString());
                return;
            }
            Console.WriteLine("i modified");
        }
    };
}
```

在 SDK 命令行中运行：sn –k mykey.snk，会生成名为 mykey.snk 的密匙文件，下面用它给 sample1041 签属强名称。在命令行里输入：csc /keyfile:mykey.snk sample1041.cs，编译成功。运行后仍是输出 i=1。

用 ildasm 打开 exe，双击 manifest（清单），会看到.assembly sample1041 的如下定义：

```
.assembly sample1041
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compilation
            RelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .custom instance void [mscorlib]System.Runtime.CompilerServices.
             RuntimeCompatibilityAttribute::.ctor() =
              ( 01 00 01 00 54 02 16 57 72 61 70 4E 6F 6E 45 78   // ....T..WrapNonEx
                63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )   // ceptionThrows.
  .publickey = (00 24 00 00 04 80 00 00 94 00 00 00 06 02 00 00   // .$..............
                00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00   // .$..RSA1........
                3F 8C A8 ED C0 77 E9 53 17 EC B6 6D D9 35 03 84   // ?....w.S...m.5..
                DC BC D6 FF 3F 97 96 9B 82 79 68 22 49 D0 ED 82   // ....?....yh"I...
                52 53 DB A0 28 9A FE 8A C8 1A 60 1C B4 2F 70 D1   // RS..(.....`../p.
                32 FD AA 64 E4 B4 EF 22 4E 9C C7 AA 40 DD AD DC   // 2..d..."N...@...
                DD 2A CC 93 DE 9D 92 2C D9 DE EA 3B B9 0B 00 96   // .*.....,...;....
                93 0B 3E 0B 23 8F B5 A3 19 1F 26 4E E6 84 7B 13   // ..>.#.....&N..{.
                7A 6F 0E 63 A1 8E A3 93 C7 7F 2F 50 3C F2 EA B4   // zo.c....../P<...
                A2 09 DB 50 8F 53 EA CE 9C C8 A3 21 42 BD 2F D2 ) // ...P.S.....!B./.
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
```

回顾前文描述 MSIL 时，示例代码中.assembly{}里没有包含任何内容，而这里多出了.publickey 和.hash algorithm，这两项便是该 Assembly 的强名称与所用算法。试着修改一下，将文件偏移 02e0h 处的 17h 改为 18h，即将源代码中 if(i==1)改为 if(i==2)，保存后运行，弹出报错窗口，而命令行中会显示出错信息（如图 9.15 所示）。很明显，“Strong name validation failed.”提示该文件强名称验证失败。

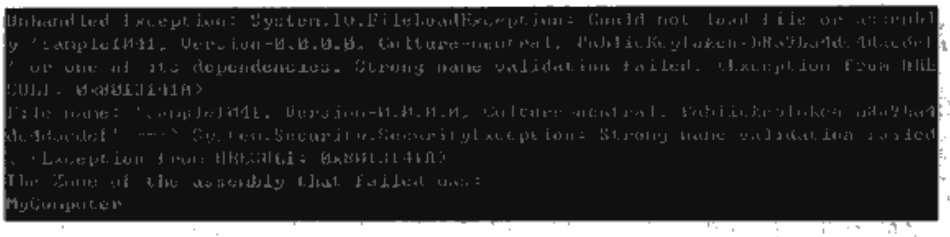


图 9.15 强名称验证失败的报错信息

要对此类文件进行打补丁，必须首先去除强名称的干扰。修改被签署强名称的程序有以下三种方法：移除强名称，重新签署强名称，给系统打补丁。

先说移除的方法，回顾前文的 PE 结构章节，一个文件中有 4 处标识指出了该文件是否有强名称：

（1）CLR 头中的 flags 位，去除 COMIMAGE_FLAGS_STRONGNAMESIGNED 标志；

（2）CLR 头中的 StrongNameSignature，RVA 与 Size 均应为 0；

（3）Assembly 表中的 Flags 项，减去 0001h（PublicKey 标识），通常改变后的标识变为 0000h（SideBySideCompatible）；

（4）Assembly 表中的 PublicKey 项，指向#Blob 的偏移，用 0 填充。

可以用工具直接移除强名称，如 Strong Name Remove，用它打开 sample1041.exe，单击 Verify 后显示如图 9.16 所示，窗口下部内容是本 Assembly 中的强名称信息，上部列表框中是所有引用的 Assembly 的强名称。这里只引用了一个系统文件 mscorlib。对于大型程序，如引用框中列出多个 Assembly 并附有强名称，则应该全部 patch，系统文件除外。

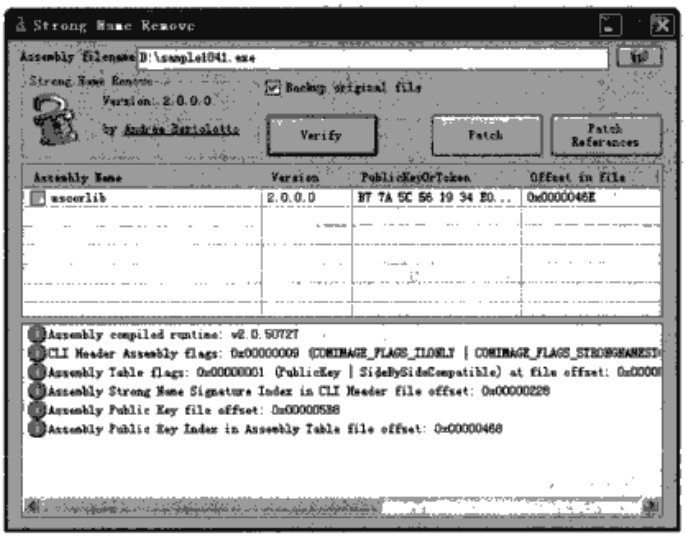


图 9.16 用 Strong Name Remove 移除强名称

对于可以用 ildasm 反编译的程序，可以在 IL 代码中将.publickey 项和.hash 项删除后再重新编译。

关于重新签署（替换）强名称，也有现成的工具，比如国外 Libx 编写的 RE-Sign，某些时候替换比直接去除的通用性更好，建议使用。而给系统打补丁的方法，在看雪论坛里已经由 lccracker 详细介绍过了，在此一并略过。最后值得注意的是，某些程序在代码中将强名称用于加解密计算（通过 GetPublicKey 或者 GetPublicKeyToken），这时如果仅仅去除它则会出错，应该选择替换强名称并修改相应代码。

由于强名称的最初目的是用于版本识别和代码完整性校验，而非程序保护，所以它的保护强度看起来不值一提。但作为程序开发者，仍应该给自己的所有文件签署强名称，并学习正确的使用方法（如 Code Group Strong Name，请参考 MSDN）。关于在程序开发中正确使用强名称的内容已超出本书范围，请有兴趣的读者自行深入。

9.4.2 名称混淆

.Net 诞生之初，有玩笑说微软在变相搞开源，因为所有的程序信息都被保存在文件中（函数名、字符串、类结构，就差注释了），并可以很方便地用反编译工具还原。为了改变这一状况，混淆器（Obfuscator）应运而生，它可以将所有（类、方法、属性等）名称变为无意义的字符。与 Visual Studio 捆绑的 DotFuscator 免费版也许是最早的商业混淆软件，现在已经出现了多种强度更大的 Obfuscator，如 9rays Spices.net、Xenocode PostBuild、{smartassembly}和 DotFuscator 正式版等。

最简单的名称混淆原理是改变 PE 文件中#Strings 流的数据，该流中保存了所有的类、方法、属性等的名称，以增加反编译代码的阅读难度。实践出真知，下面来演示一下实现这种混淆的方式。

键入如下代码，保存为 sample1042_1.cs 后编译。该段代码模仿了最简单的密码验证，判断用户输入的密码是否为“sample”，然后输出验证结果。

```
// code for sample1042_1
using System;
namespace tankaiha.sample1042_1
{
    class class1
    {
        public static void Main()
        {
            Console.WriteLine("Please input password");
            string s=Console.ReadLine();
            if(CheckValid(s)==true)
            {
                Console.WriteLine("password OK");
            }
            else
            {
                Console.WriteLine("invalid password");
            }
        }

        private static bool CheckValid(string pass)
        {
            return pass=="sample"? true:false;
        }
    };
}
```

代码中有个函数名叫 CheckValid，望文生义，看到这个名称就知道它是比较注册码的。用 Reflector 打开 sample1042_1.exe，程序所有的结构一清二楚。下面修改一下程序，用十六进制编辑工具打开 exe 文件，来到编移 440h 处，动一些手脚，将图 9.17 左边所示的数据全部改为 20h。

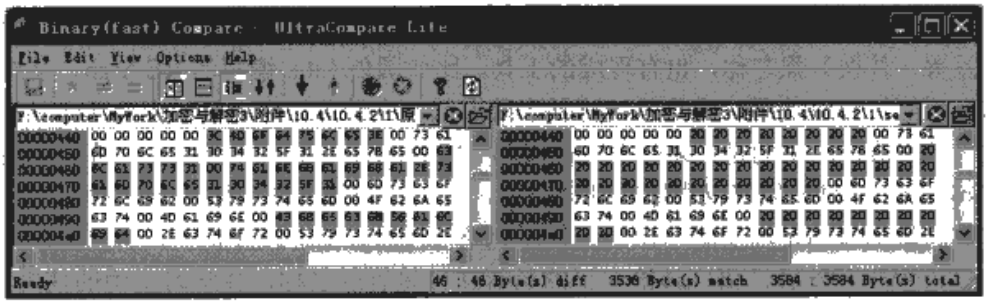


图 9.17 进行名称混淆（左图：修改前，右图：修改后）

再用 Reflector 打开 exe 文件，看一下修改前后对比。如图 9.18 所示，名称全都为空，特别是那个 CheckValid，因为它们全部被替换为了空格的 ASCII 码（20h）。这就是名称混淆的最基本原理，改变了#Strings 流中的字符串。比较成熟的混淆器各有各的修改方式，如将名称全部变为类似 x0412vaf84j21lfiavj 的无规律数值，或是直接将字符替换为不可打印的 ASCII 码。读者也可以直接将上面的字符修改为 00（即把所有的名称都删除），程序仍能正常运行。为什么呢？因为元数据 MethodDef 表中已经定义了每种方法的代码偏移和大小，只要这两项数据正确程序便可正常运行，名称是什么无所谓。

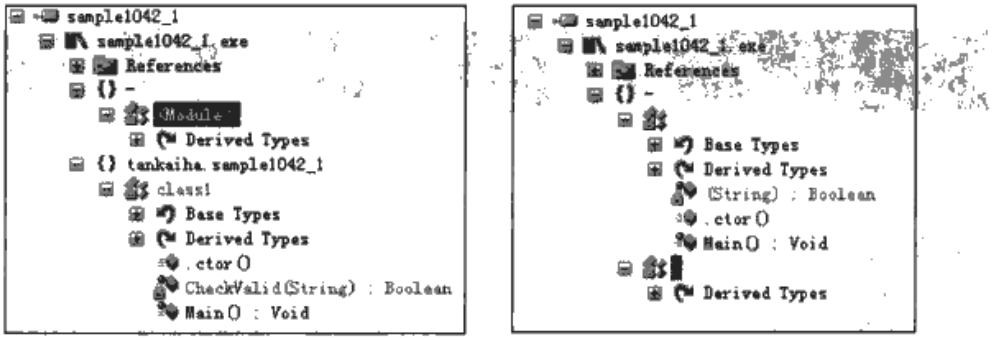


图 9.18 名称混淆前后反编译效果对比

不过，有一些函数名称是不能修改的，比如每个类的.ctor 和.cctor，系统方法如 Console.WriteLine 等。一旦这些名称被修改，程序运行将报错。

现在改变一下代码，将 CheckValid 方法放在一个 dll 中。新建 sample1042_2_lib.cs，键入下面的代码，用如下命令将其编译为一个 dll 文件：csc /target:library sample1042_2_lib.cs。

```
//code for sample1042_2_lib
using System;
namespace tankaiha.sample1042_2_lib
{
    public class class2
    {
        public static bool CheckValid(string pass)
        {
            return pass=="sample"? true:false;
        }
    };
}
```

原 sample1042_2.cs 的代码也做相应改变（粗体为主要变动部分），编译命令为：csc /r:sample1042_2_lib.dll sample1042_2.cs，即命令行中要添加对 dll 的引用。

```
using System;
using tankaiha.sample1042_2_lib;
namespace tankaiha.sample1042_2
{
    class class1
    {
        public static void Main()
        {
            Console.WriteLine("Please input password");
            string s=Console.ReadLine();
            if(class2.CheckValid(s)==true)
            {
                Console.WriteLine("password OK");
            }
            else
```

```
            {
                Console.WriteLine("invalid password");
            }
        }
    }
}
```

重复前述步骤对 exe 文件进行混淆，简单起见，只改变 CheckValid，将其全部用“20”覆盖。保存后运行，报错了，如图 9.19 所示。

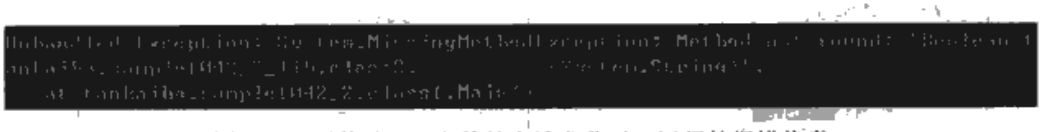
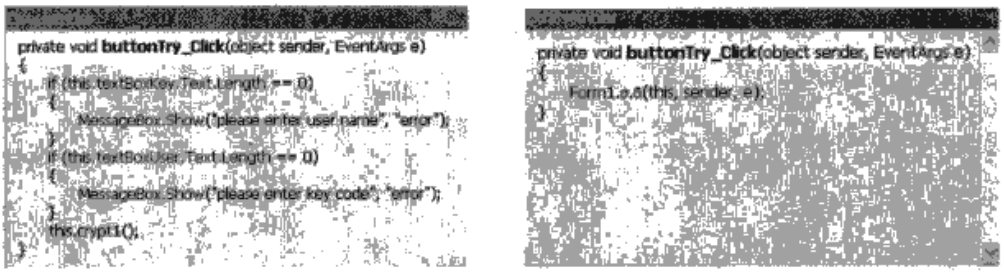


图 9.19　只修改 exe 文件的方法名称时，运行的报错信息

错误信息中提示找不到“　　”（全部是空格）这个方法。造成这个原因很简单，exe 中的方法名改变，dll 中相应的方法名也应该改变，并且要和 exe 的改法一致。用 UltraEdit 打开 dll，将其中的“CheckValid”全部用 20h 替换，再运行，一切又恢复正常。引用名称必须保持一致（欲知原因，请看 MemberRef 表的结构），这也解答了为什么混淆过的程序中凡是有关系统方法的调用一律使用原名，因为一般情况下不可能去修改系统方法的名称。因此，系统调用往往是逆向程序的突破口。

保护软件自然不会轻易让逆向者通过系统调用进行分析，比如 Spices.net 提供一个叫 Anonymization 的方法，就可以让寻找突破口的难度增加。以 9.3 节的 CrackMe 为例，被 Spices.net 混淆后的代码如图 9.20 的右图所示，而左图是原代码。



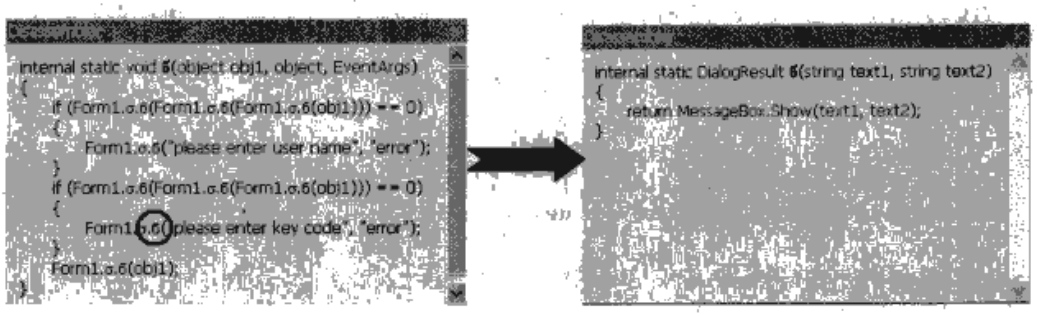
```
private void buttonTry_Click(object sender, EventArgs e)
{
    if (this.textBoxKey.Text.Length == 0)
    {
        MessageBox.Show("please enter user name", "error");
    }
    if (this.textBoxUser.Text.Length == 0)
    {
        MessageBox.Show("please enter key code", "error");
    }
    this.crypt1();
}
```

```
private void buttonTry_Click(object sender, EventArgs e)
{
    Form1.o.6(this, sender, e);
}
```

图 9.20　Anonymization 方法保护代码（1）

同样是 buttonTry_Click 事件，混淆前调用了两次 MessageBox，最后调用 crypt1，混淆后却什么代码都没有了，只剩下一个奇怪的调用“Form1.σ.6”。原来的代码呢？前面说过，形如 MessageBox 之类的系统方法名是不能被改变的，那它隐藏在哪儿呢？秘密就在这个奇怪的 σ.6 中。点击 6 跳转到该方法处，进入后见到了熟悉的用户字符串，但仍然没有见到 MessageBox 的调用（见图 9.21 左图）。再次点击字符串前的 6，终于找到了目标（见图 9.21 右图）。Anonymization 没有改变系统调用的名称，只是将其隐藏了起来（通过添加新的类和方法）。这种保护手段确实增加了分析的难度，但无论怎么变，系统调用也是无法隐藏的。

对于其他没有介绍的混淆原理，读者要记住一点：必须保证程序的完整和有效性，也必须遵守.Net 的规范。



```
internal static void 6(object obj1, object, EventArgs)
{
    if (Form1.σ.6(Form1.σ.6(Form1.σ.6(obj1))) == 0)
    {
        Form1.σ.6("please enter user name", "error");
    }
    if (Form1.σ.6(Form1.σ.6(Form1.σ.6(obj1))) == 0)
    {
        Form1.σ.6("please enter key code", "error");
    }
    Form1.σ.6(obj1);
}
```

```
internal static DialogResult 6(string text1, string text2)
{
    return MessageBox.Show(text1, text2);
}
```

图 9.21　Anonymization 方法保护代码（2）

名称混淆的原理解释清楚了，可怎么对付名称混淆呢？通常，名称混淆不会影响静态分析，但如果混淆强度已经达到影响正常分析的地步，则需要考虑修改名称。对于将名称混淆为不可打印字符的，应将这些名称替换回可打印的字符，就是上面修改 sample 的逆操作。由于替换操作本身是不可逆的，因此并没有办法将所有名称还原为初始状态，只能由读者根据分析结果来猜测名称。

对于包含数个 dll 文件的大型程序来说，单独修改主程序的名称会造成程序不可运行，原因之一就是 dll 文件中的名称也必须做相应的改变。看雪论坛 dreaman 编写的名称反混淆工具可以直接将某个目录下所有的 exe 与 dll 文件进行相对应的名称修改，rick 编写的名称编辑工具 Meta Editor 用来修改单个程序集也很方便。另外，现在的反编译软件大多提供了不可打印字符的显示，比如 Spices.net 可直接输出这些字符的十六进制名称，Decompiler .Net 则是在方法中随机对这些混淆字符串进行重命名，dis#对于单个文件更是支持所有的名称自动反混淆并输出为 C#工程文件。有一点需要了解，混淆不能保护原代码，只会增加原代码的阅读难度。

9.4.3 流程混淆

顾名思义，流程混淆就是指打乱程序流程，隐藏原作者的意图，增加代码阅读难度。按混淆的层次可分两种：方法（或类）级别的混淆和 IL 代码级别的混淆。

关于方法级的流程混淆，前一节“名称混淆”中提到的 Anonymization 实际就属于该类保护。建立新类作为 wrapper，将所有的内部调用和系统调用全部包装到新增的类里。

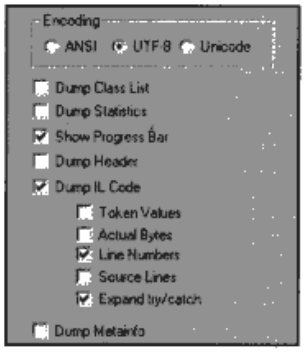


图 9.22 ildasm 导出 IL 时的选项

代码级的混淆就是将原方法的 IL 代码次序打乱，以达到增加 IL 阅读难度的效果，同时它还可以防止反编译工具直接将代码反编译为高级语言的形式。听起来很像 Win32 下的花指令，但由于 IL 语言本身的特点，.Net 中的流程混淆远没有 asm 下花指令效果好。下面请读者动手实现一个最简单的流程混淆。

用 ildasm 打开 sample1041.exe，导出为 dump.il，注意选择导出选项时将 Line Numbers 选中。其他很多选项是 dump 元数据的，暂时用不着，如图 9.22 所示。

下面修改 dump.il，由于 sample1041.exe 是被加上强名称的，这里正好演示如何在 IL 中去除强名称。下面只列出修改过的 Main 方法代码，其中斜体部分是修改过的代码，仅仅是给程序增加了两个 br.s 跳转。

```
.method public hidebysig static void  Main() cil managed
 {
  .entrypoint
  // Code size       52 (0x34)
  .maxstack  2
  .locals init (int32 V_0,bool V_1)
  IL_0000:  nop
  br.s  IL_0001

  IL_0002:  stloc.0
  IL_0003:  ldloc.0
    ……
  IL_0026:  br.s     IL_0033
  IL_0028:  ldstr      "i modified"
  IL_002d:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0032:  nop
  IL_0033:  ret
```

```
    IL_0001:  ldc.i4.1
    br.s      IL_0002
  } // end of method class1::Main
```

在 SDK 命令行中运行：ilasm /resource=dump.res /output=sample1043.exe dump.il，编译生成 sample1043.exe。试运行下，正常，说明强名称已经移除。用 Reflector 加载，以 C#方式查看 Main 方法的代码。出现什么情况？Reflector 报错了，如图 9.23 所示。

不仅是 Reflector，现有的反编译软件均不能把代码还原为 C#或其他高级语言格式，但 IL 代码仍是可以反编译的。这种混淆有什么用？由于 C#的可读性远远强于 IL，这种流程混淆避免了程序直接被还原为高级代码，从而增加了代码阅读的难度。

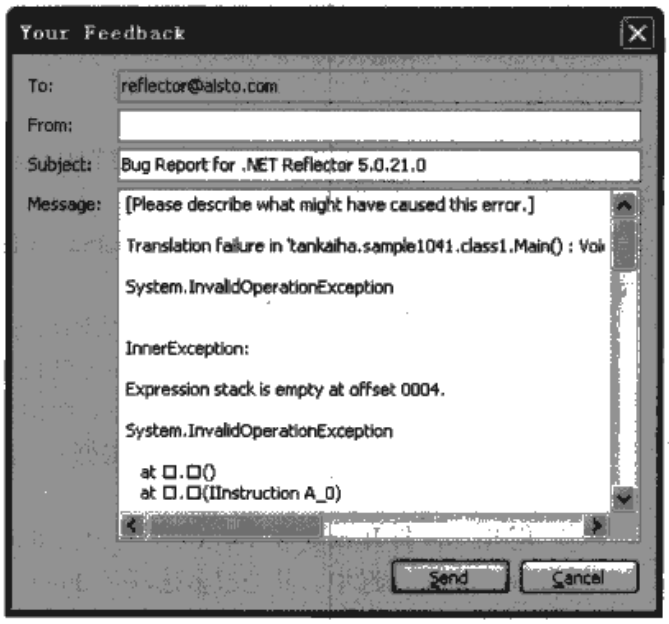


图 9.23　Reflector 对流程混淆过的程序报错

商业加密软件的流程混淆方法自然比上面复杂得多，如 Xenocode Postbuild，它会在源程序中添加许多垃圾代码，将程序分为很多代码段，运行时不停地进行跳转，并加入大量垃圾代码。最常见的如直接跳转：

```
.....
IL_075d:  br        IL_06cb
IL_0762:  br        IL_0600
IL_0767:  br.s      IL_076e
IL_0769:  br        IL_0130
.....
```

或者是加入恒成立的条件跳转：

```
IL_0006:  ldc.i4      0x80000000
IL_000b:  brtrue      IL_0298
.....
IL_029d:  ldc.i4.0
IL_029e:  brfalse.s   IL_026f
```

有什么办法反流程混淆？第一种方法，用工具帮助进行流程分析，如 IDA 提供的流程图功能。用 IDA pro 加载 exe 后，按 F12 键显示 Main 方法的流程图如图 9.24 所示。在这个简单的流程混淆中，IDA 的表现可圈可点，它将两个 br.s 跳转连成一线，清楚地显示出了程序的流程。至于对非常复杂的程序实战性强否，留待各位读者在使用中自行体会。

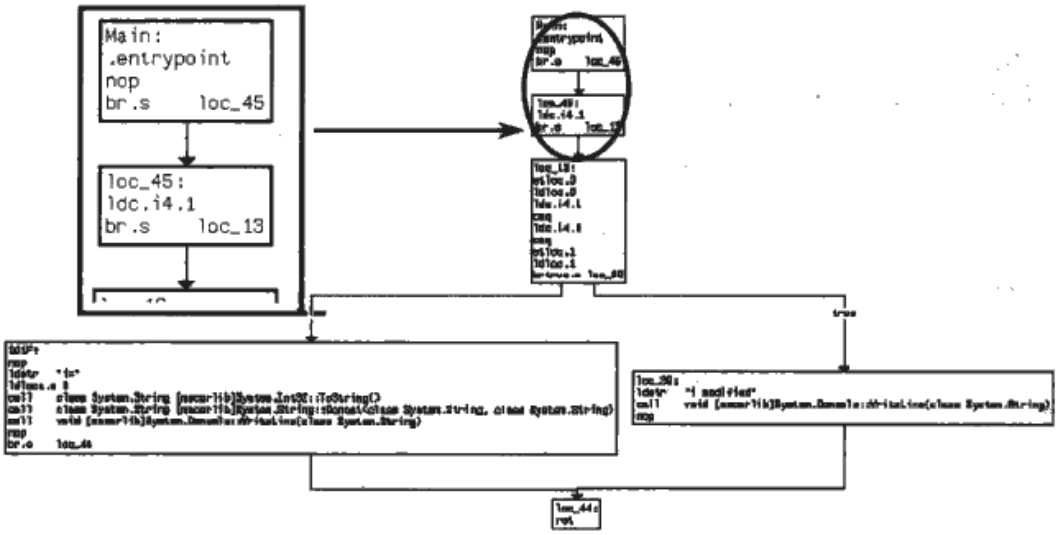


图 9.24　IDA pro 的流程图显示

第二种反流程混淆的方法要求有一定的编程功底。一般每种保护软件都有固定的流程混淆算法，如果可以分析出它的算法，便可以写出通用的反流程混淆软件，将反编译出的 IL 代码进行重组。有时，一些加密软件的混淆有固定的模式（Pattern），看似复杂，却可以通过简单的删除或替换 IL 指令来反混淆，看雪论坛 huweiqi 编写的 Simple Assembly Explorer 便具有基于模式的反混淆功能，读者可自行试用。

第三种方法，无论名称混淆还是流程混淆，代码还是摆在面前了，不过这次又多穿了一层马甲。因此，耐心的分析是对付这类保护方式的杀手锏。

9.4.4　压缩

随着.Net 可执行文件的流行，出现了一些支持.Net 的加壳软件。本节讨论的是这些加壳软件中不含元数据加密功能的那一类（也可称为压缩壳），至于含加密功能的壳的分析，放在下一节。

由于笔者所了解的几个壳对.Net 2.0 支持不是太好，所以本章程序运行在 1.1 平台下，无论在哪个版本的.Net 下，其基本原理是相同的。先看示例程序代码：

```
.assembly extern mscorlib{}
.assembly sample1044{}
.module sample1044.exe
.imagebase 0x00400000     //可省略
.subsystem 0x0003         //可省略

.namespace tankaiha.sample1044
{
.class private auto ansi beforefieldinit class1 extends [mscorlib]System.Object
{
    .method public hidebysig static void  Main() cil managed
     {
       .entrypoint
       .maxstack  1
       .locals init (class [mscorlib]System.Reflection.Assembly V_0)
       IL_0000:  nop
       IL_0001:  call   class [mscorlib]System.Reflection.Assembly [mscorlib]
                        System.Reflection.Assembly::GetEntryAssembly()
       IL_0006:  stloc.0
       IL_0007:  ldloc.0
```

```
        IL_0008:  callvirt  instance string [mscorlib]System.Object::ToString()
        IL_000d:  call      void [mscorlib]System.Console::WriteLine(string)
        IL_0012:  nop
        IL_0013:  ret
      }
      .method public hidebysig specialname rtspecialname instance void.
    ctor() cil managed
      {
        .maxstack  8
        IL_0000:  ldarg.0
        IL_0001:  call      instance void [mscorlib]System.Object::.ctor()
        IL_0006:  ret
      }
    }
}
```

代码的功能就是通过 Reflection 空间 Assembly 类的 GetEntryAssembly 方法，取得正在运行的 Assembly 全名并输出（一个程序集的全名包括名称、版本、语言和 PublicKey 标识）。

.Net 程序的压缩壳按编写方式，可分为基于.Net 平台编写和基于 Windows 平台编写两大类。纯.Net 编写的压缩壳包括 Sixxpack（现更名为 AdeptCompressor）、.NETZ 和 bsp。Sixxpack 将原程序压缩后存在新文件里，运行的时候动态解压至 byte[]数组中，最关键的是这两句。

```
Assembly assembly1 = Assembly.Load(buffer);
assembly1.EntryPoint.Invoke(null, null);
```

.Net 从诞生开始就支持这种内存中的 Assembly 载入，然后 Invoke 调用该 Assembly 的方法。解压的方法也有几种：手动 dump、用工具 dump、根据解密算法直接还原原程序等。

bsp 也利用了 Sixxpack 类似的原理，关键是压缩算法不同。这里就用 bsp 压缩 sample，然后讲一下手工 dump。用 PEBrowseDbg 载入压缩过的程序，在三个方法上全部下断，然后继续运行，如图 9.25 所示。

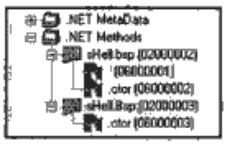


图 9.25　bsp 压缩的程序

第一次中断在 token 为 06000001 的方法处，在代码窗口中浏览一下，可以看出 bsp 用的也是 Assembly.Load，然后对入口 method 进行 Invoke，不过它对原程序进行了混淆，因此调用的入口方法不是 sample 中的 Main，而是一个名称为不可显示字符串的方法。此时的任务：等原程序全部解压完毕后进行 dump。向下看，来到 IL_02D1 处的第一句 asm 代码，然后“Run to selection”。

```
Disassembly of JITTED sHell.bsp::  (06000001) at 0x00F90058
  ; IL_02BB: ldloc.s 0x23
  ; IL_02BD: call  System.Reflection.Assembly::Load()
  ; IL_02C2: ldloc.s 0x23
  ; IL_02C4: ldc.i4.0
  ; IL_02C5: ldc.i4.1
  ; IL_02C6: stelem.i1
  ; IL_02C7: ldloc.s 0x23
  ; IL_02C9: ldc.i4.1
  ; IL_02CA: ldc.i4.2
  ; IL_02CB: stelem.i1
  ; IL_02CC: call  System.GC::Collect()
  ; IL_02D1: stloc.1
  0xF904E1: 8BCE              MOV         ECX,ESI  ;Run to selection到这里
  0xF904E3: FF151C2BBA79      CALL        DWORD PTR [0x79BA2B1C]
  0xF904E9: 8BF8              MOV         EDI,EAX
  0xF904EB: 837E0400          CMP         DWORD PTR [ESI+0x4],0x0
```

```
0xF904EF: 0F8681000000      JBE     0xF90576        ; (*+0x87)
```

注意斜体的那句指令（在不同的机器上地址可能不一样），对照 IL 代码，这句应该是将 byte[]数组地址传递给 ecx，作为 Assembly.Load 的参数。双击 esi，查看该处的内容，如图 9.26 所示。熟悉的字符出现了，这不就是 PE 头吗。

可是这并不是原程序，而是 bsp 的一个 loader，通过这个 loader 再调用原程序。是否 dump 该 loader 与本程序无关，但 dump 方法还是值得介绍下，基本流程如下：由于 PEBrowseDbg 没有 dump 的功能，因此利用 WinHex 将数据保存为文件。用 WinHex 打开进程的内存，按“Alt+G”键跳转到 01296FA0h（01296F98h+8，注意这个值不是固定的，而是随机分配的内存地址）处，然后定义块首 Begin of block，问题是块尾定义在哪。注意刚才 esi 所指的内存，并不是直接指向了“MZ”，前面还有两个数据。与 Win32 下不同，.Net 中任何托管代码都不是直接与内存地址打交道，byte[]数组也是一个系统类型，而它的第 2 个 DWORD 就指明了它的大小为 5000h。这个猜想对不对，不如保存后再验证。因此，块尾定义在 1296FA0h+5000h=129BFA0h。保存为文件后，用 Reflector 载入，原来只是个 loader 而不是原始 exe 文件，且它是一个 dll，不能直接运行。

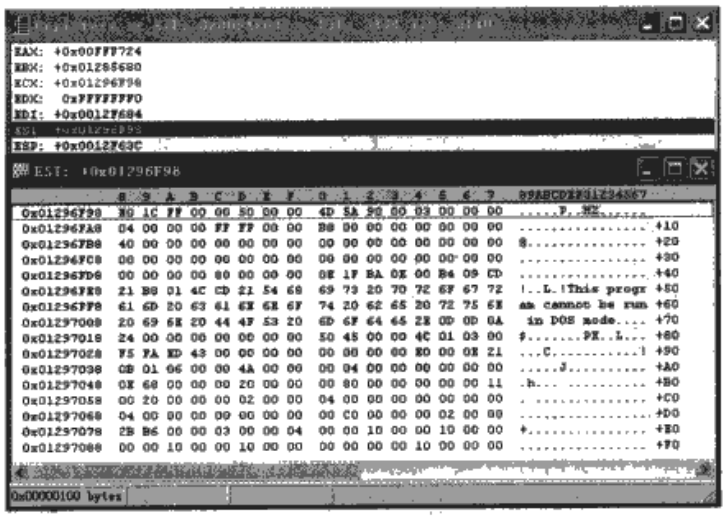


图 9.26 esi 所指内存的数据

下一步该怎么办？下断点。由于被加壳的程序是 exe，里面含有 entrypoint，可以猜想 loader 是通过调用这个入口方法执行原 exe。而要调用该方法，首先要取得该入口方法的“地址”。因此，将断点下在 mscorlib 的 System.Reflection.Assembly.get_EntryPoint。点击运行后，果然中断下来，如图 9.27 所示。

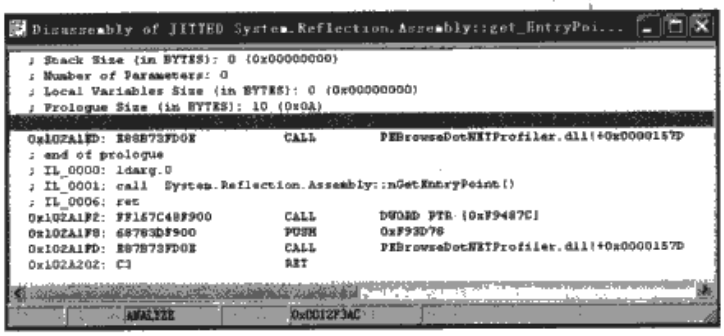


图 9.27 中断在 get_EntryPoint

按 F10 键步过 ret，返回后来到如图 9.28 所示代码处（PEBrowseDbg 中调试快捷键与 OllyDbg 不同，走的是微软路线），中断在 MOV EBX,EAX 一句，它的上一条 CALL 指令便是调用下断点的方法（get_EntryPoint）。

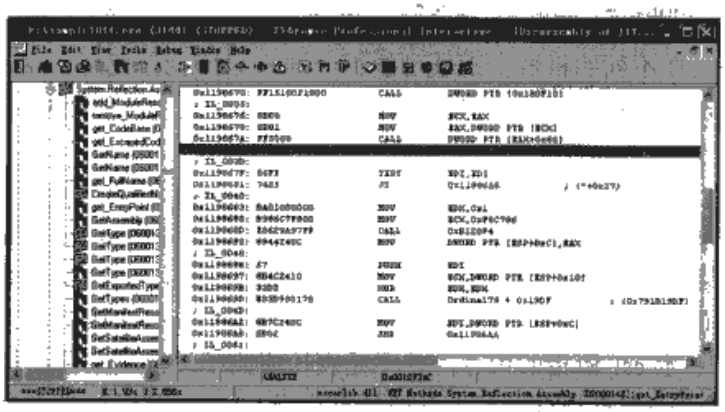


图 9.28　调用 Assembly.GetEntryPoint 返回后的代码

如果继续跟下去，还可以跟到 sample1044 中的 Assembly.ToString 方法，不过这里已经可以 dump 了，因为取得入口点的调用只能出现在完整的 Assembly 解压之后。用 WinHex 重新打开内存，查找“BSJB”（CLR 头的标志，要熟练运用哦），当来到 1221270h 处时，发现离该标志不远的字符串中出现了“sample1044”，这不就是#Strings 流吗。

```
01201270  1E 02 28 04 00 00 0A 2A  42 53 4A 42 01 00 01 00  ..(....*BSJB....
01201280  00 00 00 00 0C 00 00 00  76 31 2E 31 2E 34 33 32  ........v1.1.432
01201290  32 00 00 00 00 00 04 00  60 00 00 00 D4 00 00 00  2.......  ...?..
....
012013A0  00 00 18 00 00 00 00 00  00 00 00 00 00 3C 4D 6F  ............<Mo
012013B0  64 75 6C 65 3E 00 73 61  6D 70 6C 65 39 34 34 2E  dule>.sample944.
012013C0  45 58 45 00 6D 73 63 6F  72 6C 69 62 00 73 61 6D  EXE.mscorlib.sam
012013D0  70 6C 65 39 34 34 00 73  61 6D 70 6C 65 39 34 34  ple944.sample944
012013E0  2E 65 78 65 00 53 79 73  74 65 6D 00 4F 62 6A 65  .exe.System.Obje
```

再从“BSJB”向上寻找 PE 头，越过“.text”节，来到 01221000 处，这里便是需要 dump 的内存块的头部了。块的大小又如何选择？由于内存区域一般成片分配，在 PEBrowseDbg 中选择 Index Detail 看一下内存块详情（如图 9.29 所示），01220000h 到 01222FFFh 是一个块，就先 dump 这一段。同样的方法，用 WinHex 保存为文件，改扩展名为 exe，这就是原始程序。

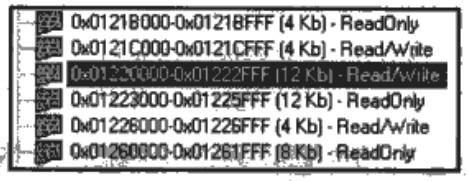


图 9.29　PEBrowseDbg 中显示的内存块

上面讨论了手工还原 bsp 压缩的 Assembly 的方法，希望大家举一反三。能够直接 dump 整个 Assembly，这也是为什么一直说 bsp 是压缩壳而不是保护壳的原因。

也有将压缩壳保护做得比较好的商业保护软件，如{smartassembly}，该壳将多个文件加密后保存，并且使用了强名称作为解密密钥，且结合强大的混淆，因此强度不错。

传统的壳也有支持.Net PE 文件的，它们加壳的最大特点是，生成的文件是 Win32 而不是.Net。压缩壳不是.Net 保护技术的主流，本书就介绍到这。像这类将原文件经过运算存储在文件中（或资源中），运行时将完整原文件在内存中展开并运行的加壳方式叫 Whole Assembly Protection。这种“整体程序集保护”的方式被证明强度是非常弱的，它可以轻易地被 NET Unpacker 之类的脱壳软件 dump。

9.4.5　加密

如果一个程序声明必须在.Net 下运行，但用 PEiD 检测却是 Win32 程序，多半这个程序已经被加密了。.Net 保护技术发展到现在，最流行的保护措施之一便是利用 Win32 的本地代码通过加密元数据、挂钩系统内核等手段保护原程序。要分析被这类程序保护的.Net 文件，读者不仅要对.Net 平台非常熟悉，还必须对.Net 内核有一定了解，比如 JIT 的编译过程，执行引擎 EE（Execute Engine）的原理等。所以在介绍加密前，有必要先简要叙述一下本章用到的内核知识。由于层次的“降低”，本章所用的调试器也从 IL 级的 PEBrowseDbg 转为汇编级的 OllyDbg。

如果用 Win32 的 PE 工具打开.Net PE，会发现整个引入表只有一个 API，exe 对应 mscoree.dll 的_CorExeMain；dll 对应 mscoree.dll 的_CorDllMain。这就是说，Windows 的 loader 载入.Net PE 后，只负责跳转到相应的 dll 中，随后该程序便运行在 ee 的监管中，Windows 本身不再负责该程序的内存分配、线程管理等工作，而将这些工作交给了.Net 框架。

用 OllyDbg 任意跟踪一个.Net PE 的加载过程，将模块加载的跟踪选中，观察 dll 的加载顺序。mscoree.dll 一直存在，随后第一个.Net 组件是 mscorwks.dll，然后是 mscorlib，再接着是 mscorjit（其间会加载一些本地 dll）。如果是窗口应用程序还会出现一些和 Forms 相关的 dll，VB 编写的程序会有 Visual Basic 的相关 dll。.Net 的核心代码在 mscorwks.dll 中，而 JIT 部分主要由 mscorjit.dll 负责。当遇到没有编译的方法时，mscorwks 调用 JIT 把代码编译为 asm 后，将控制权交给 asm 并执行，完毕后 mscorwks 再收回控制权。正因为 mscorwks 和 JIT 在整个.Net 中的核心地位，大多数加密软件都以这两个 dll 作为突破口，或进行挂钩，或进行包装，这种种操作的最终目的就是在 JIT 前将被加密的 IL 代码和元数据恢复为正常，并在方法结束后再将元数据和 IL 代码销毁，以达到保护原程序的目的。

本节分析的第一种保护方式是 CodeVeil，先查下壳，PEiD 显示是 UPolyX v0.5，是否准确无关紧要，用 OllyDbg 加载后查看可执行模块，没有 mscoree.dll。.Net 程序怎么会没有这个必需的 dll 呢？遇到这种情况，熟悉 Win32 下脱壳的读者应该很自然地想到一个经典断点：LoadLibraryA(W)。按 F9 键运行后，第一个中断便是 LoadLibraryA，参数 FileName 为“mscoree.dll”（如图 9.30 所示）。

```
地址      数值      注释
0012FB80  00408E2D  CALL 到 LoadLibraryA 来自 Crackme.00408E28
0012FB84  0040348E  FileName = "mscoree.dll"
0012FB88  00000000
```

图 9.30　在加载 mscoree.dll 时中断

跟踪至 LoadLibrary 返回并执行用户代码，不一会就看到利用 GetProcAddress 取得_CorExeMain 的地址，看来 CodeVeil 把原先.Net 程序自动完成的工作给“手动化”了。

再向下会调用 VirtualProtect，壳可能要开始解密了。

```
004091A8  - E9 23893F7C          jmp kernel32.VirtualProtect
004091AD  - 66:E9 1E28           jmp 0000B9CF
```

看一下堆栈，数据为 402000h，这是.text 区块的内存地址。

```
0012FB60    00408EAA  /CALL 到 VirtualProtect 来自 Crackme.00408EA5
0012FB64    00402000  |Address = Crackme.00402000
0012FB68    00000004  |Size = 4
0012FB6C    00000020  |NewProtect = PAGE_EXECUTE_READ
0012FB70    0012FB74  \pOldProtect = 0012FB74
```

前面介绍 PE 结构时说过，.text 区块保存了所有的元数据和 IL 代码，.text 区块的内存地址为 402000h，先看一下该处的内容。

```
00402050  EE 0F 97 2B 7D AC 59 B2 64 CA 95 3A 1F 0A D3 E2  ??}瑙瞭灏:.逾
00402060  2F B6 58 89 0B E9 FD F5 08 14 08 0C D3 8D 26 47  /襖?華?•`.訊&G
00402070  09 3D 72 8E 7B 09 09 46 DB D9 2D 23 59 09 0B 0A  .=r嶽..F圪-#Y..
00402080  21 48 09 09 63 5B 23 00 FE 0F 91 2B CA A8 59 B2  !H..c[#.??撕Y
```

和未加密的 CrackMe 比较，你会发现这明显是加过密的数据。由于壳最终肯定会在某处将 402050h 处的数据解密，此时便可以在该数据上设置断点，从而避免在无尽的花指令中跳来跳去。下内存断点会改变原数据，使还原的数据出错，因此在第一个字节处下硬件访问和写入断点。取消所有已下内存断点后运行，不一会就停在硬件断点上。

```
004095BB   F6C1 01            test cl,1//第一次硬件中断在这里
004095BE   74 15              je short Crackme.004095D5
004095C0   8A06               mov al,byte ptr ds:[esi]
004095C2   34 AA              xor al,0AA
004095C4   24 FC              and al,0FC
004095C6   0C 02              or al,2
004095C8   8806               mov byte ptr ds:[esi],al
004095CA   25 FF000000        and eax,0FF
004095CF   C0E8 02            shr al,2
004095D2   46                 inc esi
004095D3   EB 28              jmp short Crackme.004095FD
004095D5   8A06               mov al,byte ptr ds:[esi]
```

这段代码完成的就是第一次解密。第一次解密，难道还有二次解密？此时，402050h 处的数据如下，通过与原程序比较得知现在是完全解密的数据：

```
00402050  13 30 02 00 2B 00 00 00 01 00 00 11 00 03 2C 0B
00402060  02 7B 01 00 00 04 14 FE 01 2B 01 17 0A 06 2D 0E
00402070  00 02 7B 01 00 00 04 6F 10 00 00 0A 00 00 02 03
00402080  28 11 00 00 0A 00 2A 00 FE 0F 91 2B CA A8 59 B2
```

保持断点不变，继续执行，不一会就来到第二次硬件中断处：

```
00409751   0300              add eax,dword ptr ds:[eax]
00409753   000F              add byte ptr ds:[edi],cl
00409755   6F                outs dx,dword ptr es:[edi]
00409756   0B8D A4240000     or ecx,dword ptr ss:[ebp+24A4]
0040975C   0000              add byte ptr ds:[eax],al
0040975E   8BFF              mov edi,edi
00409760   0F6F06            movq mm0,qword ptr ds:[esi]
00409763   0FEFC1            pxor mm0,mm1 //第二次硬件中断在这里
00409766   0F7F07            movq qword ptr ds:[edi],mm0
00409769   83C6 08           add esi,8
0040976C   83C7 08           add edi,8
0040976F   49                dec ecx
00409770 ^ 75 EE                jnz short Crackme.00409760
```

不多见的 SSE 指令。这段代码执行完毕后，看一下 IL 代码处的数据。

```
00402050  13 30 02 00 2B 00 00 00 01 00 00 11 29 90 03 3C
00402060  CB B2 7B B1 10 00 04 54 FE E1 3B B1 17 7A A6 6D
00402070  DE E0 02 7B B1 10 00 04 6F F0 00 00 0A A0 00 02
00402080  23 38 91 10 00 0A A0 00 03 30 04 00 9C 04 00 00
```

前后两次数据有相同的也有不同的，相同的是方法头，不同的是方法体。至此，CodeVeil 保护的秘密真相大白：每次执行方法前，CodeVeil 将方法体的 IL 代码还原；每次方法执行完毕，再将 IL 动态加密，

防止 dump，但方法头保持不变。

什么是方法头，什么是方法体？原来，.Net 的方法在内存中是按一定结构存在的，主要由三个部分组成：方法头、方法体、异常处理表。

.Net 中的方法有两种，fat 和 tiny，两种方法结构不同，定义均在 CorHdr.h 中。.Net 规定，只有当方法中没有异常处理表，堆栈深度小于 8，并且代码大小为 64 字节之内时，方法才可为 tiny。

```
// tiny method header
typedef struct IMAGE_COR_ILMETHOD_TINY
{
    BYTE Flags_CodeSize;
} IMAGE_COR_ILMETHOD_TINY;

// fat method header
typedef struct IMAGE_COR_ILMETHOD_FAT
{
    unsigned Flags    : 12;  // Flags
    unsigned Size     : 4;   // size in DWords of this structure
                             // (currently 3)
    unsigned MaxStack : 16;  // maximum number of items (I4, I, I8,
                             // obj ...),
                             // on the operand stack
    DWORD   CodeSize;        // size of the code
    mdSignature  LocalVarSigTok;// token that indicates the signature of
                             // the local vars (0 means none)
} IMAGE_COR_ILMETHOD_FAT;
```

看一下未加密程序文件偏移 402050h 处第一个 method 的头结构：

```
00402050  13 30 02 00 2B 00 00 00 01 00 00 11
```

这是一个 fat header，表 9-11 说明了这些数据的具体含义。

表 9-11 Fat Method Header 结构

头参数与大小	值	说 明
flags 和 size（2 字节）	3013h （0011000000010011）二进制	最高 4 位：0011，表示头的大小为 03h 个 DWORD 中间 10 位：0000000100 代表 flags=04h 最后 2 位：11，代表 fat 03h（02h 代表 tiny）
MaxStack（2 字节）	02h	定义 MaxStack 的大小
CodeSize（4 字节）	0000002bh	该方法 IL 代码的大小是 2bh
LocalVarSigTok（4 字节）	11000001h	局部变量的 token（11h 指向 StandAloneSig 表）

读者应该形成一个条件反射，看到内存数据为 13 30h 就要想到有可能是方法头，并且是 fat 的。在实际情况中，遇到 fat 方法的几率也远大于 tiny。

回到 OllyDbg 中来，究竟什么时候可以 dump 呢？既然知道了 CodeVeil 的运行原理，时机就很好掌握，在 CodeVeil 将元数据全部解密完毕后，在加密元数据之前。错过这个时机 dump 下来的可全是乱码。

下面准备 dump 了，前面介绍过了手工操作，这里使用工具 Task Explorer（集成在 Explorer Suite 中），操作见图 9.31。在进程中选中 CrackMe，在下方 Module 框中选中 CrackMe 模块，在右键弹出菜单中选择“Dump PE”。

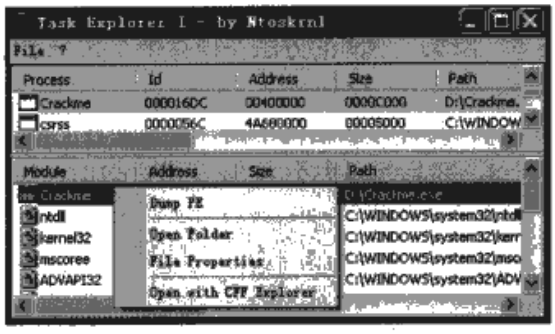


图 9.31　在加载 mscoree.dll 时中断

用 Reflector 载入 dump 下来的文件，显示没有 CLR 头，不是.Net PE 文件。剩下的工作是文件结构修复，这里介绍两种半自动的修复方法。第一种方法，用 CFF 载入后修复 PE，重新对齐文件，保存后已经可以用 ildasm 载入了，如图 9.32 所示。ildasm 反编译，ilasm 再编译，一个完好的.Net PE 文件生成了。

第二种方法，用 dis#载入脱壳后的文件，首先进行反混淆，如图 9.33 所示。有时，太多的不可打印字符会使编译工具出错。

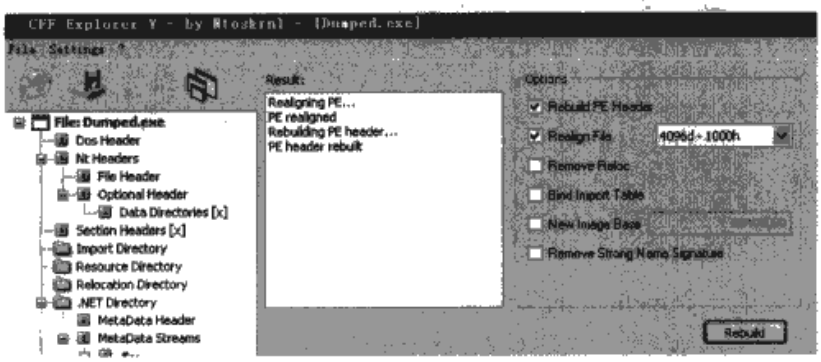


图 9.32　用 CFF 修复 PE 文件

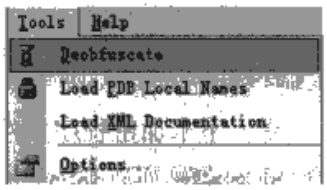


图 9.33 dis#反混淆前后对比

对比 dis#反混淆前后的效果，原先的乱码已经被替换为 Class 之类的名称，虽然只是简单的改变，但可读性已大大增加（见图 9.34）。

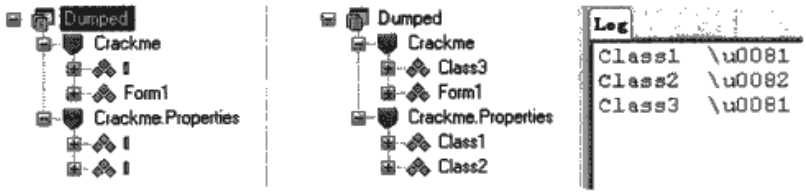


图 9.34　dis#反混淆前后对比

随后，在 dis#中将文件导出为 C#工程文件（如图 9.35 所示），便可以用 Visual Studio 载入了。将入口方法改为 Main，再次编译，于是 exe 文件便生成了，运行检测一切正常。（注：现在大多数程序均经过流程混淆，可直接导出 C#并成功编译的机会不是很多。）

可以看出 CodeVeil 是一款比较“懒”的程序，因为不论是哪个方法调用 JIT，它都会将所有的元数据进行解密，现在已经有壳做到了 per-method 解密，就是每调用一个方法时解密该方法的元数据，这样就无法轻易地完整 dump 整个程序集。

下面来看看国外另一款壳的保护方式。被该壳保护的产品可以用 Reflector 载入，但是所有的方法都为空（见图 9.36）。很明显，这类方法的 IL 代码在运行时要被动态释放。

图 9.35 dis#导出 C#工程文件

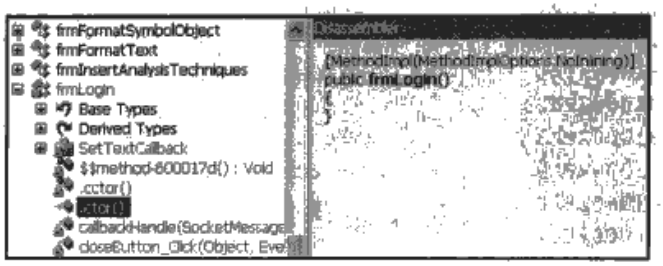


图 9.36 被保护的程序可以用反编译程序载入，但方法都为空

注意一下程序的安装目录，会发现一个 rscoree.dll，如果说 CodeVeil 的秘密稳藏在 exe 文件本身（.rsrc 节），那么该壳的秘密就稳藏在这个本地 dll 中。

Reflector 打开后，发现只有一种方法体的 IL 代码不为空，这就是静态构造函数.cctor，如图 9.37 所示。同样是 frmLogin 类，.cctor 的方法却包含一个到<PirvateImplementationDetails>的调用。静态构造方法在类被加载时执行，且只执行一次，它的执行顺序先于类中所有的代码。但就是这一次执行，已经足够壳解密该类的所有代码了（见图 9.37）。

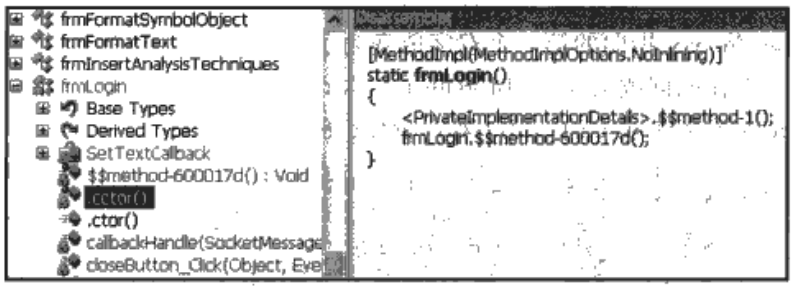


图 9.37 静态构造函数里包含调用解密的代码

点击 cctor 中的链接，来到<PrivateImplementationDetails>的内部，最有意思的代码出现在_RSEEStartup 方法中：

```
[MethodImpl(MethodImplOptions.ForwardRef),
DllImport("rscoree.dll", CharSet=CharSet.Ansi, ExactSpelling=true)]
private static extern void _RSEEStartup(int A_0);
```

这里的代码属于.Net 中的互操作，是引用本地 dll 中函数的声明，这个本地 dll 正是刚才在程序安装目录中看见的 rscoree.dll。由于调用了该 dll 的导出函数“_RSEEStartup”，下面的分析便是进入 dll 里看这个启动函数到底做了什么。

用 Win32 反汇编软件对 rscoree.dll 反汇编，进入 startup 函数后，不一会见到如下代码：

```
.text:10002CC5              push    offset aVvn8StYmW ; "vvn8~st[ym}w"
.text:10002CCA              call    sub_10001680
.text:10002CCF              mov     edi, ds:GetModuleHandleA
.text:10002CD5              add     esp, 8
.text:10002CD8              lea     ecx, [esp+420h+ModuleName]
.text:10002CDF              push    ecx     ; lpModuleName
.text:10002CE0              call    edi     ; GetModuleHandleA
```

跟踪到 GetModuleHandleA 时，可以从参数里看出“vvn8~stlym}w”的解码字符串为“mscorjit.dll”，下面看壳对 JIT 做了哪些手脚。

```
.text:10002E8D              push    offset aStOq    ; "~sT~oq"
.text:10002E92              call    sub_10001680
```

```
.text:10002E97          add     esp, 8
.text:10002E9A          lea     ecx, [esp+42Ch+ProcName]
.text:10002EA1          push    ecx             ; lpProcName
.text:10002EA2          push    esi             ; hModule
.text:10002EA3          call    ds:GetProcAddress
.text:10002EA9          test    eax, eax
.text:10002EAB          jz      loc_10002F6B
.text:10002EB1          call    eax             //调用 getJit 函数
```

这次解码出来的名称是 getJit，getJit 是 mscorjit.dll 为数不多的导出函数之一。取得该函数地址后，紧接着调用它，然后保存 getJit 的返回值。有趣的是保存 getJit 的返回值完毕后的一段代码：

```
.text:10002F36          mov     edx, [ecx]
.text:10002F38          push    offset dword_10024850 ; Value
.text:10002F3D          push    eax             ; Target,getJit 的返回值
.text:10002F3E          mov     dword_1002483C, edx
.text:10002F44          mov     dword_10024850, offset sub_100027F0
.text:10002F4E          call    ds:InterlockedExchange
.text:10002F54          pop     edi
.text:10002F55          pop     esi
```

InterlockedExchange 将 eax 指向的第一个双字值变为了 10024850h 处的值，也就是 100027F0h，其中 eax 为 getJit 的返回值。现在的问题就是 getJit 返回了什么？

用 IDA 反汇编.Net 内核文件 mscorjit.dll，来到 getJit 函数的代码处：

```
.text:7907EA7A          public __stdcall getJit()
.text:7907EA7A          __stdcall getJit() proc near
.text:7907EA7A          mov     eax, dword_790AF168
.text:7907EA7F          test    eax, eax
.text:7907EA81          jnz     short locret_7907EA97
.text:7907EA83          mov     eax, offset dword_790AF170
.text:7907EA88          mov     dword_790AF170, offset const CILJit::'vftable'
```

粗体的 mov 指令说明了 eax 的返回值是 CILJit::vftable 的偏移，很自然，想到查看 vftable 的内容是什么。双击后来到 vftable 的偏移处（就在 getJit 下方）：

```
.text:7907EA98 const CILJit::`vftable'
dd offset CILJit::compileMethod(ICorJitInfo *,CORINFO_METHOD_INFO *,… *)
.text:7907EA98                          ; DATA XREF: getJit()+E↑o
.text:7907EA9C          dd offset CILJit::clearCache(void)
.text:7907EAA0          dd offset CILJit::isCacheCleanupRequired(void)
```

vftable 偏移处是一串指针列表，看名称应该是一些系统函数的地址。到这里，rscoree 的流程就很清楚了：取得一系列系统函数的地址，并对其中的某个地址进行替换。由于它只通过 InterlockedExchange 替换了第一个指针，因此只需要弄清 vftable 的第一项是什么：

```
CILJit::compileMethod(ICorJitInfo *,CORINFO_METHOD_INFO *,uint,uchar * *,ulong *)
```

这个函数是 JIT 引擎的核心函数 compileMethod，它的输入是 IL 代码，输出为本地代码。只要是.Net 程序，就必调用该函数，因此该地址可以称为.Net 下的万能断点。rscoree 将 ICorJitInfo 中指向 compileMethod 的指针替换为 dll 中的 100027F0h 处，暂且把它称为 hookcompileMethod，这样在每个 JIT 事件时，.Net 都会调用 hookcompileMethod，从而使壳有机会解密代码，再将解密后的数据传递给真正的 compileMethod。运行 OllyDbg，下断在 hookcompileMethod，跟一会就会看到如下代码，其中最后的 call 就是调用最初在 _RSEEStartup 中保存的原始 JIT 方法地址。

```
012429B0  |.  50          push eax                          ; /Arg6
```

```
012429B1  |.  8B45 10mov eax,dword ptr ss:[ebp+10]          ; |
012429B4  |.  51      push ecx                              ; |Arg5
012429B5  |.  8B4D 0Cmov ecx,dword ptr ss:[ebp+C]           ; |
012429B8  |.  52      push edx                              ; |Arg4
012429B9  |.  8B55 08mov edx,dword ptr ss:[ebp+8]           ; |
012429BC  |.  50      push eax                              ; |Arg3
012429BD  |.  51      push ecx                              ; |Arg2
012429BE  |.  52      push edx                              ; |Arg1
012429BF  |.  FF15 3C482601call dword ptr ds:[126483C]; \mscorjit.7906E7F4
```

到这里，前一种壳 CodeVeil 的秘密也清楚了。CodeVeil 是如何挂钩 JIT 的呢？在 OllyDbg 中查看 CodeVeil 保护程序的 ICorJitInfo 接口处的数据，结果数据显示其中第一个双字指向了主程序内部，compileMethod 被挂钩了。

```
7907EA98  29 92 40 00 A7 E1 06 79 16 CE 07 79 8D 51 FF E9  )扒.n§y?y 咎?
```

到这里，读者已经初步接触了壳挂钩.Net 内核的操作原理。其实 CodeVeil 和第二种壳的挂钩方法最多只能称作 Wrapper，更强的加密方法是 Hook 内核 dll，改变其代码流程，从而在.Net 内核 JIT 的过程中进行加密与解密。这种加密方式的强度远大于 Wrapper，有兴趣的读者可以自行深入分析。

对付此类保护的方法，可以利用进程注入后反射的方法取得源代码和元数据，再重构 PE 文件，或者采用更通用的方法，即挂钩 JIT 层得到代码。可以看出，单纯使用加密壳的安全程度仍然一般，程序开发者应尽量将混淆和加密结合使用。

9.4.6 其他保护手段

上面介绍的是 5 种主流保护方式，除此之外，程序中还经常应用一些小措施来增加逆向难度，比如反调试跟踪、网络验证等。下面就介绍其中的一部分小技巧。

1. 反监测

程序保存注册信息时，最忌讳 Spy 软件的监测，因此在进行关键操作（文件和注册表）时，往往会检测是否有 Spy 软件在运行。最简单的实现是在关键代码段中枚举所有窗口，取得窗口的名称，并比较是否包含敏感字符串。最常见的是 Win32API 中的 FindWindowExW 函数。

2. 反调试跟踪

Win32 下的反调试做得比较完善，可惜.Net 是高层平台，无法直接利用寄存器实现 Anti，因此纯.Net 实现的反调试相对来说功能要弱些。程序被调试时，Win32 下有 IsDebuggerPresent 用来检测，.Net 下则是通过 System.Diagnostics.Debugger 实现，代码如下：

```
IL_002d: call      bool[mscorlib]System.Diagnostics.Debugger::get_IsAttached()
IL_0032: ldc.i4.0
IL_0033: ceq
```

OllyDbg 中有隐藏调试器标志的插件，PEBrowseDbg 中也有相应选项，如图 9.38 所示。因此，这种检测方法效果一般。

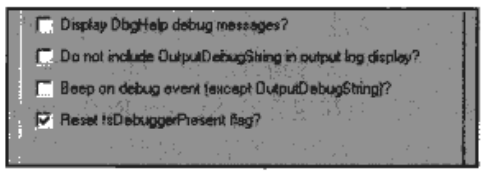


图 9.38 PEBrowseDbg 中的隐藏调试器选项

另一种反调试的方法也借鉴自 Win32，即在两段代码处分别取当前时间并相减，如果差值过大则认为程序被单步跟踪调试了，后面的流程则会调用完全不同的算法来迷惑分析者，或直接让程序退出。下面的示例代码是比较两次时间之差是否大于 3 秒：

```
IL_0ea8:  ldloca.s   V_8
IL_0eaa:  call instance int32 [mscorlib]System.TimeSpan::get_Seconds()
IL_0eaf:  ldc.i4.3
IL_0eb0:  cgt
IL_0eb2:  ldc.i4.0
IL_0eb3:  ceq
IL_0eb5:  stsfld  bool xxxxxxx.x83a6ad2c48984168::x6ae604ff7a55905e
IL_0eba:  br         IL_0e24
```

第三种方法是利用 C++/CLI 的混合编译特性，在本地代码中加入强大的反调试代码。这种方法较新颖，但本地代码也是可以被 patch 的。

3. 网络验证

网络验证现在已经比较常用，在.Net 中主要依靠 System.Net 中的几个类实现。如果程序没有提供网络功能，而在运行时防火墙不断提示程序要求联网，读者就应该注意分析是否存在网络验证了。

4. 虚拟机

VM 保护是当今很热门的一种保护方式，Win32 下已经出现多个利用 VM 的保护软件，而.Net 中也开始出现类似保护方式。虽然现在尚未普及，但绝对是将来.Net 保护方式发展的一个大方向。

5. 加密锁

这个大家很熟悉了，主要用于行业软件中。虽然保护的目标是.Net 程序，但其核心调用仍属于 Win32 范畴，与.Net 关系不大。

9.5 深入.Net

本节主要介绍.Net 中相对较深入的内容，包括反射机制、代码文档对象模型的应用、.Net 的几个核心接口，以及通过开放源代码的简化.Net 框架及框架内核 dll 来研究.Net 核心的运作方式。本节的内容看似和加密、解密没有直接关系，但掌握本章的内容会让读者的逆向分析技术更上一层楼。

9.5.1 反射与 CodeDOM

什么是反射（Reflection）？反射能做什么？

MSDN 中的解释很清楚，这是.Net 中获取运行时类型信息的方式，.Net 的应用程序由程序集（Assembly）、模块（Module）、类（Class）组成。而反射提供一种编程的方式，让程序员可以在程序运行期获得这几个组成部分的相关信息。例如：Assembly 类可以获得正在运行的程序集信息，也可以动态地加载程序集，以及在程序集中查找类型信息，并创建该类型的实例。Type 类可以获得对象的类型信息，此信息包含对象的所有要素：方法、构造器、属性等，通过 Type 类可以得到这些要素的信息，并且调用之。MethodInfo 包含方法的信息，通过这个类可以得到方法的名称、参数、返回值等，并且可以调用之。诸如此类，还有 FieldInfo、EventInfo 等，这些类都包含在 System.Reflection 命名空间下。

从逆向的角度看，通过反射，程序可以做下面的事：取得当前“进程”中加载的所有 Assembly，取得 Assembly 的各个 Module 和 Type（例如 Reflector 的树状结构），取得某个方法的 IL 代码，动态载入新的 Assembly，执行某个指定方法。

下面来看几个应用反射的例子。第一个例子是 rick 写的利用反射取得本程序所有 IL 编码的示例。程序没有加密，可直接在 Reflector 中参考源码。

首先运行未加密的版本，得到的输出片段如下：

```
Method  System.Globalization.CultureInfo get_Culture()
  MaxStackSize: 8    CodeSize: 6
7E020000042A
Method  Void set_Culture(System.Globalization.CultureInfo)
  MaxStackSize: 8    CodeSize: 7
0280020000042A
```

下面是 CodeVeil 加密过的程序，运行后 dump 到的 IL 代码：

```
Method  System.Globalization.CultureInfo '()
  MaxStackSize: 8    CodeSize: 6
7E020000042A
Method  Void '(System.Globalization.CultureInfo)
  MaxStackSize: 8    CodeSize: 7
0280020000042A
```

除了方法的名称因为混淆而改变外，IL 代码的字节并没有改变。不过 CodeVeil 加密过的程序，只会在第一次反射时取得正确的元数据，第二次则会出错。原因就是 CodeVeil 会将运行过的 IL 再次加密，第二次执行反射时，由于 JIT 后的代码已经存在，则不再次调用解密，因此只能得到被加密的元数据。

现在的关键问题是，怎么样让一个程序执行解密需要的反射代码呢？这需要用到另一个程序 injectDll，这个程序功能就是通过注入 dll，在另一个程序空间里执行用户编写的 C#代码。这个程序作为脚本演示平台非常方便，下面演示一段脚本，功能是取得 String 的所有方法的内存地址。

```
namespace ScriptNS
{
    public class Script
    {
        public void Run(Control output)
        {
            Type t=typeof(String);
            output.Text+="\r\nType:"+t.FullName+"\r\n";

            MethodInfo[] ms=t.GetMethods(BindingFlags. Static|BindingFlags.Instance|\
                BindingFlags.Public|BindingFlags .NonPublic|BindingFlags .DeclaredOnly);
            foreach(MethodInfo m in ms)
            {
              output.Text+="\r\nMethod:"+m.Name+":"+m.MethodHandle.GetFunctionPointer().\
                        ToInt32().ToString("X8");
            }
        }
    }
}
```

脚本输出片段如下：

```
Type:System.String

Method:System.IConvertible.ToBoolean:793A6980
Method:System.IConvertible.ToChar:793A69A0
```

```
Method:System.IConvertible.ToSByte:793A69C0
.....
Method:get_Chars:7A06C2E8
Method:get_Length:7A06C32D
```

这个脚本有什么用呢？试想在 OllyDbg 中对取得注册码长度的代码下断点，便可直接在 7A06C32Dh 处下断点跟踪，很方便。

利用注入原理和.Net 的反射机制，可编写出很具有实战性的工具，如 injectReflector，可以实现无须 dump 直接在内存中查看被加壳程序的 IL 代码。如图 9.39 所示，左图演示用 Reflector 直接载入被加壳的程序出错，右图演示利用 injectReflector 通过注入反射便可直接显示 IL。对于 Win32 程序，同样可以注入，之后便可以在普通 Windows 程序中调用.Net 的类和方法，是不是很有趣！更多反射的功能，由大家自己摸索，网上相关资料很多。

那么 injectDll 是怎么做到内存中动态执行 C#脚本的呢？.Net 为程序员提供了代码文档对象模型（CodeDOM）来实现该功能。关于 CodeDOM 的资料也是公开的，有兴趣的读者可以自行阅读。

有时，injectDll 注入反射脚本并不能取得所有的程序信息，特别是被加密过的程序。这是因为反射的底层是调用了非托管元数据 API 接口，此类接口直接在内存中取元数据，而不管其是加密的还是解密的。

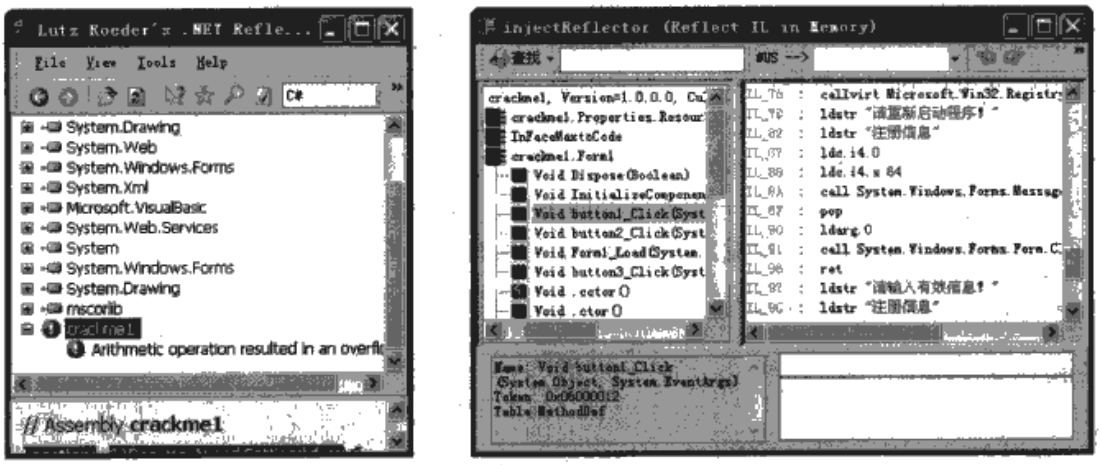


图 9.39　利用注入反射可以取得被加壳程序的源代码

9.5.2　Unmaganed API

在 1.1 版.Net 的安装目录下，有一个 Tool Develop Guide 目录，里面的 doc 子目录包含了面向.Net 工具开发者的相关文档，samples 目录则对应于文档的一些代码示例。微软对这些文档的描述是“开发人员工具指南包含生成在.NET Framework 中运行的底层开发工具（如编译器、浏览器、分析器和调试器）的文档、产品规范和示例”，由此可见这些文档和代码的含金量。2.0 版中，这些文档已经被移至 MSDN。

Unmanaged API 是.Net 平台提供的一系列非托管 API 接口，核心是运用了 COM 技术。这一系列的 API 很多，其中最重要的是三类：调试接口（ICorDebug 类）、分析器接口（ICorProfiler 类）和元数据接口（IMetaData 类），而实际使用过程中，这三类接口往往会相互调用。正因为它们的重要性，1.1 版的文档中只包含了对这三类接口的详细描述而省略了其他内容。

先从分析（Profiling）API 说起，通过 ICorProfiler 接口提供的方法，可以得到以下过程的相关信息并控制它们：Application 的开始/结束，Assembly 的载入/卸载，Methods 的开始/结束，Module 的载入/卸载，Class 的载入/卸载，与 COM 接口的互操作，托管/非托管代码的即时编译等，基本上.Net 的所有核心操作都可以通过 Profiling API 来得到信息。中文 MSDN 中提供了非常好的一篇示例文章：《在.NET Framework 2.0 中，没有任何代码能够逃避 Profiling API 的分析》，附带的源码很有参考价值。看雪论坛已有数篇文章讲

述怎样在现有代码的基础上实现自己的 Profiler，并利用该 Profiler 动态地修改内存中的 IL 代码，这里不再赘述，只介绍一些基本概念。

Profiling API 是通过编译成 dll 文件，注册为系统的 COM 服务而实现的。因此，Profiler 的例子通常都是三个文件：ProfilerCallback.h、ProfilerCallback.cpp 和 Profiler.cpp。前两个是实现 Profiler 的核心功能代码，最后一个只是实现了 COM dll 的基本框架。通常要实现自己的 Profiler，只需要在 ProfilerCallback.cpp 中添加相应代码即可。

当一个 Profiler 的 dll 编译成功后，要想正确使用必须有下列步骤：

① regsvr32 Profiler.dll（注册服务）。

② 设置进程的环境变量，主要是两个：

```
SET COR_PROFILER={18884ADE-B15B-4af8-BE6C-FE5117BA4B32}//该 Profiler 的 Guid
SET COR_ENABLE_PROFILING=1
```

③ 运行需要监测的程序，得到 Profiler 的输出。可以输出到调试器，也可保存为文件。

④ 关闭 Profiler，set Cor_Enable_Profiling=0x0。

⑤ regsvr32/u Profiler.dll（卸载服务）。

一般说来，Profiler 主要提供监测的功能，程序开发中通常用它来检测程序执行效率。若想动态修改 IL 代码，则应注意方法头和方法体的结构：对于不含异常处理表的 fat 方法与普通的 tiny 方法，直接修改即可，而含有异常处理表的方法则要注意调整异常处理表的数据。

在一个 Profiler 中，不可能不用到元数据接口。比如在代码监测过程中，需要修改程序让它动态加载一个 dll 并执行其中的方法，这就需要引用该 dll（Assembly）的某个类型（Type）的某个方法（Method），而程序集、类型和方法在.Net 中的引用都需要通过元数据定义来实现。看一段代码示例：

```
IMetaDataEmit* pMetaDataEmit = NULL;
IMetaDataAssemblyEmit* pMetaDataAssemblyEmit = NULL;
hr = m_pICorProfilerInfo->GetModuleMetaData(moduleId, ofRead | ofWrite,IID_IMetaDataEmit,\
     (IUnknown** )&pMetaDataEmit);
if (FAILED(hr)) { goto exit; }
 hr =pMetaDataEmit->QueryInterface(IID_IMetaDataAssemblyEmit,(void**)&\
     pMetaDataAssemblyEmit);
if (FAILED(hr)) { goto exit; }
```

该段代码用在 Profiler 中，首先通过 ICorProfilerInfo 接口取得 IMetaDataEmit 接口，继而通过后者获取 IMetaDataAssemblyEmit 接口，为后续代码动态定义 Assembly 引用做准备：

```
hr = pMetaDataAssemblyEmit->DefineAssemblyRef(&assemblyPublicKeyToken,sizeof\
     (assemblyPublicKeyToken),L"inject", &amd, NULL, 0, 0, &tkInsertLib);
```

DefineAssemblyRef 后，运行中的程序便被动态添加了“inject”的程序集引用，也就是新添加的程序集拥有 token 了。同样的方法，继续为类和方法构造 token 值并定义引用，便可在代码中调用该方法了：call methodToken。

光盘映像文件中提供了一个只监测方法入口参数和返回值的 Profiler，按照上述步骤执行后，监测结果保存在 output.log 文件中。MSDN 提供了更多开源的 Profiler 完整代码，值得参考。

下面的代码片段出自某 Profiler 软件的监测结果，其中注册码已经被捕捉到（粗体部分）。

```
Leave - FunctionID 0x0496E288; clientData: 0x0496E288; COR_PRF_FRAME_INFO: 0;
Leave - No Frame Info for Method Name: xxxx.x83a6ad2c48984168.x720a41bd855b5147
Function Return Value Information:
        Parameter[0] is of type (0x0000000E) ELEMENT_TYPE_STRING
        Parameter[0] Value (ELEMENT_TYPE_STRING) == CN4Ug5y8eF5fvhEiKK/vGg==
```

```
        Parameter[1] is of type (0x0000000E) ELEMENT_TYPE_STRING
```

有时带 Profiler 的程序运行会不稳定，而且所有的加密软件都提供了 Anti 功能。比如挂钩系统 dll，在 Profiler 获取数据后再解密，从而使监测数据全部是乱码；或者在进程开始时将 Profiler 必需的环境变量改为无效。因此 Profiler 并非万能，还需读者灵活使用。

第三个接口是调试器接口。调试器接口通常用于编写调试器，了解它的原理能让调试者更好地在.Net 下使用调试器。

ICorDebug 接口提供了控制.Net 调试器和被调试程序的所有方法，其中两个最重要的方法是 ICorDebug::SetManagedHandler() 和 ICorDebug::SetUnmanagedHandler()，前者用于托管程序，后者用于非托管程序。当被调试程序中发生了感兴趣的事件时，系统便会调用这两个函数注册的处理函数，调试器便有机会进行需要的工作。通常.Net 中用到的都是 SetManagedHandler，它要求提供一个 ICorDbgManagedCallback*的回调接口。其他一些接口如 ICorDebugProcess、ICorDebugAppdomain 等，均是在调试器中用于接收和控制特定对象的信息。下面的代码片段中，注册的回调函数通过 CreateAppDomain 事件（当一个 AppDomain 被创建时调用该方法）便可以得到关于进程和 AppDomain 的信息。

```
//ICorDebugManagedCallback::CreateAppDomain
HRESULT CreateAppDomain([in] ICorDebugProcess *pProcess,
                        [in] ICorDebugAppDomain *pAppDomain);
```

从传入参数中取得 ICorDebugAppDomain 接口并保存在 pAppDomain 变量中，之后便可以调用该接口提供的方法。下面的代码中，首先是将调试器 Attach 到该程序域中，然后查看该域中所有的程序集：

```
{
    ICorDebugAssemblyEnum  *pAssemblies;
    pAppDomain->Attach();
    pAppDomain-> EnumerateAssemblies(&pAssemblies);
    pProcess->Continue(FALSE);
    return S_OK;
}
```

Unmanaged API 的内容实在太多，读者学习过程中应选关键点入手，其中最实用的莫过于元数据接口，因为它提供了直接观测程序元数据的方法，而不用程序编写者纠缠于烦琐的 PE 结构。举例来说，假设要得到所有的用户字符串，如果没有元数据接口，程序编写者首先要载入 PE 文件，分析文件头，定位到 CLR 头，然后在 CLR 头中寻找#US 流的位置和大小，并手动一项项读入字符串。而在元数据接口中，调用一个 EnumUserStrings 便可以得到所有字符串，是不是很方便！

尽管 Profiler 和 Debugger 很方便，但现在的很多加密软件都有如下声明：Anti-profiler、Anti-debugger，它们实现 Anti 的原理通常是对.Net 内核做些小手脚。要想对付这些 Anti，需要对.Net 内核有一定了解。

9.5.3 Rotor、MONO 与.Net 内核

要深入.Net 的内核，通常有三种方法：查阅微软的文档，反编译.Net 内核文件，阅读源代码。第一种方法效果一般，MSDN 一般只讲 what 和 how，至于 why 提及的不多（不过不代表 MSDN 不重要，它仍是 Windows 平台开发最好的参考资料）。第二种方法对分析者的逆向功底要求很高，但得到的结论是最权威的。第三种方法最便于掌握内核运行流程，可问题是，去哪里找.Net 框架的源代码呢？

由于微软将.Net 申请了 ECMA 标准，因此公开了一些源代码，特别是为了.Net 的普及，微软公布了一份简化版的.Net 源码 Rotor，又叫 Shared Source CLI（SSCLI）。虽然不是完整的.Net 源码，但依然提供了极有价值的资料。另一个开源.Net 平台则是 MONO，主要目的是在 Linux 类的操作系统上实现.Net 平台。本小节主要介绍如何利用 SSCLI 与本机.Net 文件的逆向相结合，来探究.Net 的内核世界。

Windows 平台上的.Net 框架核心文件以 mscor 开头的 dll 文件是研究重点，其位于“\WINDOWS\Microsoft.NET\Framework\版本号\”中。用 IDA pro 对感兴趣的文件进行反汇编，IDA 会提示是否从网上下载 dll 的符号文件，这时一定要点 yes，带符号反汇编出来的汇编代码可读性大大增强。

通过将 Rotor 的源代码与相应的反汇编代码进行对比，会看到商业化.Net 框架（就是.Net Framework）与开源.Net 框架的一些区别。两者的区别不必关心，利用 SSCLI 来辅助反汇编代码的阅读才是目的。来看几个例子，下面的代码源自 IDA 对 mscorjit.dll 的反汇编代码：

```
private: virtual enum CorJitResult __stdcall
CILJit::compileMethod(class ICorJitInfo *,
                      struct CORINFO_METHOD_INFO *,
                      unsigned int,
                      unsigned char * *,
                      unsigned long *)
                      proc near
```

compileMethod 方法中的后三个参数都没有提示名称，除了动态跟踪外，很难得知它们代表了什么。再来 SSCLI 中看相应的代码：

```
CorJitResult __stdcall FJitCompiler::compileMethod (
            ICorJitInfo*            compHnd,            /* IN */
            CORINFO_METHOD_INFO*    info,               /* IN */
            unsigned                flags,              /* IN */
            BYTE **                 entryAddress,       /* OUT */
            ULONG  *                nativeSizeOfCode    /* OUT */
            )
```

也是 5 个参数，其中 flags 为输入参数，而 entryAddress 和 nativeSizeOfCode 是输出参数，SSCLI 瞬间解开了刚才的疑问。但到这里仍不能确定 SSCLI 给出的参数与反汇编中的参数是否一一对应，这需要在动态调试过程中加以确定。（实际证明，SSCLI 的答案是正确的。）

除了给出一些未公开的变量与结构外，SSCLI 还让调试者更轻松地分析.Net 内核的一些运行机制。最基本的，一个 exe 是怎样被加载的，读者可以在 SSCLI 里通过跟踪它的代码流程弄清来龙去脉。

再来看 SSCLI 的另一个应用示例。上一小节提到了某些加密软件可以轻松躲过 Profiler 的监测，那就在 SSCLI 里寻找 Profiler 的实现代码，很自然地来到“clr/src/profile”中。通过阅读代码知道，.Net 中 EE 引擎提供了 Profiler 接口，这可以通过 GetEEToProfInterface 得到。再回到 Windows 中的.Net 框架上，其中 mscordbc.dll 提供了两个导出函数，如图 9.40 所示。

Name	Address	Ordinal
GetEEToProfInterface	63E73851	1
SetProfToEEInterface	63E738AC	2
DllEntryPoint	63E7D97A	

图 9.40 mscordbc.dll 的导出表

这说明了 mscordbc.dll 负责了 EE 与 Profiler 通信的部分工作。下面就是利用 OllyDbg 调试带 Profiler 启动的程序，目的是找出系统在何处判断是否需要向 Profiler 输出信息。最终的秘密存在 mscorwks.dll 中：

```
.text:79E972AD  ;MethodDesc::MakeJitWorker(COR_ILMETHOD_DECODER *,ulong)
.text:79E972AD              test    byte ptr ProfilerStatus g_profStatus, 6
.text:79E972B4              jnz     loc_7A0FEB52
```

其中的 test 语句判断是否需要输出 JIT 信息到 Profiler，如果是则 jnz 跳转，跳转后会来到如下代码处：

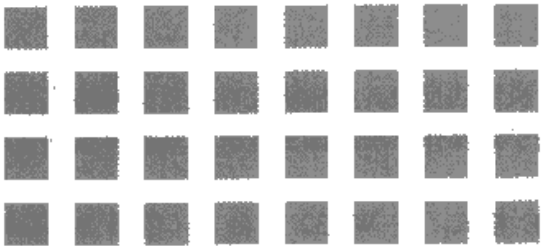
```
.text:7A0FEB7C                push    eax
.text:7A0FEB7D                mov     ecx, dword_7A382308
.text:7A0FEB83                call    dword ptr [edi+2Ch] ; call mscordbc.dll
```

其中 call 语句已经注释出了，是调用 EEToProfInterfaceImpl::JITCompilationStarted 方法，进入该方法中来到如下代码处：

```
.text:63E764E1                mov     ecx, [eax]
.text:63E764E3                push    eax
.text:63E764E4                call    dword ptr [ecx+5Ch] ; 调用我们的 profiler
```

粗体的那句 call 指令便是调用 Profiler 中的代码。到这里，一个 Profiler 是怎么输出关于 JIT 开始信息的流程就很清楚了。自然，想让 Profiler 失效也只是一个简单的内存 patch 而已：将 test 后的 jnz 指令 nop 掉即可。加密软件能让 Profiler 失效，调试者让 Profiler 恢复功能也不会太难。

本节主要介绍一些.Net 下的深入内容，从用户态的反射讲到 COM 接口非托管 API，最后是汇编及源代码级别的.Net 内核。和本章前面的内容一样，本节主要给读者提出一些分析方法，授之以鱼不如授之以渔，.Net 真正的秘密还要靠读者自己去探索。



第 5 篇　系统篇

PE 是 Windows 上可执行文件的格式，熟知 PE 文件将有助于对操作系统的深刻理解。如果你知道 EXE 和 DLL 里面的奥秘，那么你将成为一名知识更加渊博的程序员。本书用大量篇幅，图文并貌地详细讲解了 PE 格式。

SEH 的出现已非一日，但有关 SEH 的知识资料却不是很多。SEH 不仅可以简化程序的错误处理，使程序更加健壮，还被广泛应用于反跟踪和加密中。本书从调试角度讲述了 SEH 的机理，掌握这些后，调试 SEH 处理的程序，就会更加自如。

第10章 PE文件格式

从某种意义上讲，可执行文件的格式是操作系统本身执行机制的反映。虽然研究可执行文件格式并不是程序员的首要任务，但这种工作能够积累大量的知识，有助于对操作系统的深刻理解。掌握可执行文件的数据结构及其一些运行机理，也是研究软件安全的必修课。

在 Win16 平台上（如 Windows 3.x），可执行文件是 NE 格式。在 Win32 平台上（包括 Windows 9x/NT/2000/XP/2003/Vista/CE），可执行文件是 PE 格式。PE 是 Portable Executable File Format（可移植的执行体）简写，它是目前 Windows 平台上的主流可执行文件格式。

PE 文件衍生于早期建立在 VAX/VMS 上的 COFF 文件格式（Common Object File Format）。由于许多 Windows NT 的创始者来自于数字设备公司（DEC），他们很自然地使用已有的代码来快速开发新的 Windows NT 平台。采用术语“Portable Executable”是因为微软希望有一个通用于所有 Windows 平台和所有 CPU（x86, MIPS, Alpha 等）上的文件格式。当然，在 CPU 指令的二进制译码等方面会存在差异，但最重要的是系统的装载器和编程工具无需对任何一种新出现的 CPU 进行重写。从大的方面讲，这个目标已经实现。这使得 Windows NT 和它的后代，Windows 95 和它的后代以及 Windows CE 拥有了相同的格式。

为了将精力集中在 Windows NT 上，Microsoft 公司放弃了当时的 32 位工具及文件格式。例如，在 Windows NT 出现之前，16 位 Windows 的虚拟设备驱动程序使用的是 32 位的文件格式——LE 格式。更重要的是 OBJ 格式的改变：在 Windows NT 的 C 编译器之前，所有 Microsoft 编译器使用的都是 Intel 的 OMF（Object Module Format）规范。在前面讲过，基于 32 位 Windows 系统的 Microsoft 编译器所生成的是 COFF 格式的目标文件。Microsoft 的一些竞争者（如 Borland 及 Symantec）继续使用 Intel 的 OMF 格式，放弃了 COFF 格式。其结果是 OBJ 和 LIB 必须针对编译器发送不同的版本。

描述 PE 格式及 COFF 文件的主要地方是在 winnt.h，其中有一节叫“Image Format”。该节给出了 DOS MZ 格式和 Windows 3.1 的 NE 格式文件头，之后就是 PE 文件的内容。在这个头文件中，几乎能找到关于 PE 文件的每一个数据结构定义、枚举类型、常量定义。可以肯定，在别的地方也能找到相关文档，例如在 MSDN 里，但是 winnt.h 是 PE 文件定义的最终决定者。

EXE 和 DLL 文件之间的区别完全是语义上的，他们使用完全相同的 PE 格式。唯一的区别就是用一个字段标识出这个文件是 EXE 还是 DLL。还有许多 DLL 的扩展，如 OCX 控件和控制面板程序（.CPL 文件）等都是 DLL，它们有一样的实体。

另外，64 位的 Windows 只是对 PE 格式做了一些简单的修饰，新格式叫 PE32+。没有新的结构加进去，其余的改变只是简单地将以前的 32 位字段扩展成 64 位。对于 C++代码，Windows 文件头的配置使其拥有不明显的区别。

PE 文件中的数据结构一般都有 32 位和 64 位之分，如 IMAGE_NT_HEADERS32、IMAGE_NT_HEADER64 等。除了在 64 位版本中的一些扩展域以外，这些结构几乎总是一样的。在 winnt.h 都有#defines，

它可以选择适当的 32 位或 64 位结构并给它们起成与大小无关的别名（在前面的例子中，可以写成 IMAGE_NT_HEADERS）。结构选择依赖于用户正在编译的模式（尤其_WIN64 是否被定义）。若没有特别说明，本书的实例都是基于 32 位 PE 格式进行研究的。

在跳到 PE 格式细节之前，仔细看一看图 10.1，该图显示了 PE 格式的大致布局。下面将分别解释每一块的内容。学习的同时，建议用 Stud_PE 工具配合，该款工具直观地显示出 PE 各部分数据。

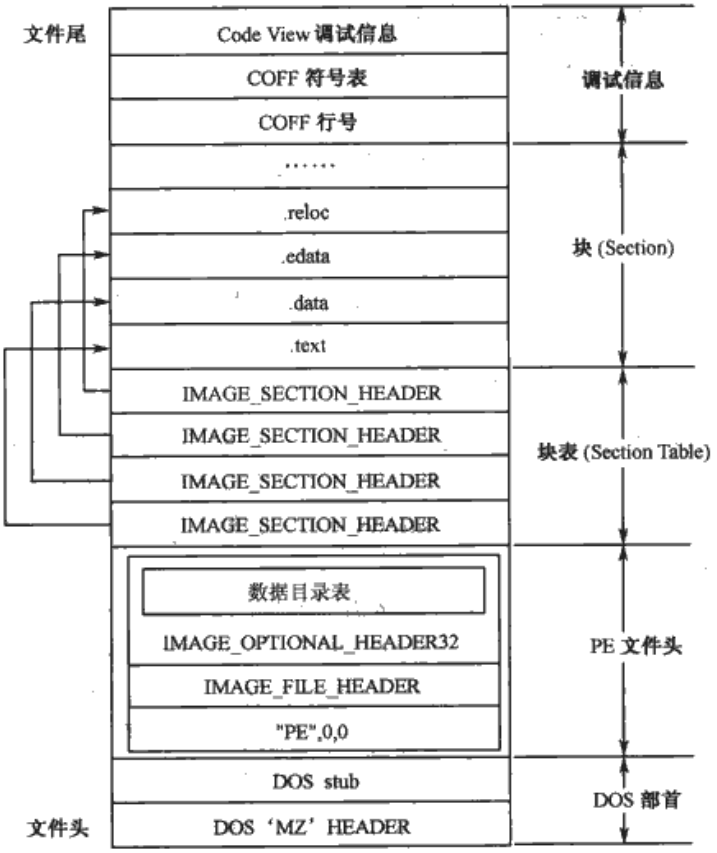


图 10.1 PE 文件的框架结构

10.1 PE 的基本概念

PE 文件使用的是一个平面地址空间，所有代码和数据都被合并在一起，组成一个很大的结构。文件的内容被分割为不同的区块（Section，又称区段、节等），区块中包含代码或数据，各个区块按页边界来对齐，区块没有大小限制，是一个连续结构。每个块都有它自己在内存中的一套属性，比如：这个块是否包含代码、是否只读或可读/写等。

认识 PE 文件不是作为单一内存映射文件被装入内存是很重要的。Windows 加载器（又称 PE 装载器）遍历 PE 文件并决定文件的哪一部分被映射，这种映射方式是将文件较高的偏移位置映射到较高的内存地址中。当磁盘文件一旦被装入内存中，磁盘上的数据结构布局和内存中的数据结构布局是一致的。这样如果知道在磁盘的数据结构中寻找一些内容，那么几乎都能在被装入到内存映射文件中找到相同的信息。但数据之间的相对位置可能改变，其某项的偏移地址可能区别于原始的偏移位置，不管怎样，所有表现出来的信息都允许从磁盘文件偏移到内存偏移的转换（如图 10.2 所示）。

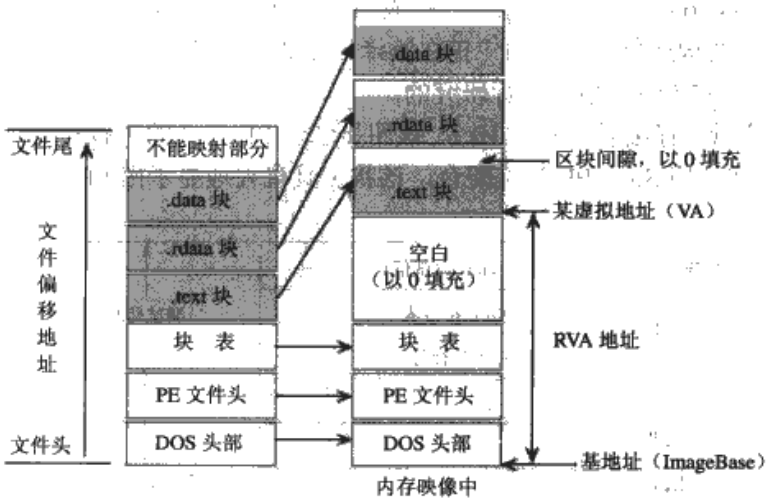


图 10.2 PE 文件磁盘与内存映像结构图

10.1.1 基地址

当 PE 文件通过 Windows 加载器被装入内存后，内存中的版本被称作模块（Module）。映射文件的起始地址被称为模块句柄（hModule），可以通过模块句柄访问内存中其他的数据结构。这个初始内存地址也称为基地址（ImageBase）。准确地说，对于 Windows CE，这是不成立的，一个模块句柄在 Windows CE 下并不等同于安装的起始地址。

内存中的模块代表着进程从这个可执行文件中所需要的代码、数据、资源、输入表、输出表及其他有用的数据结构所使用的内存都放在一个连续的内存块中，编程人员只要知道装载程序文件映像到内存后的基地址即可。PE 文件剩下的其他部分可以被读入，但是可能不映射。比如当调试信息放到文件尾部的时候，PE 的一个字段会告诉系统把文件映射到内存需要多少内存，不能被映射的数据将被放置在文件的尾部。

为了方便起见，Windows NT 或 Windows 95 将 Module 的基地址作为 Module 的实例句柄（Instance Handle，即 Hinstance）。在 32 位 Windows 系统中称基地址为 Hinstance 似乎有些混淆，因为 Instance Handle 一词来源于 16 位的 Windows 3.1，其中每个执行实例都有自己的数据段，以此来互相区分，这就是 Instance Handle 的来历。在 32 位 Windows 系统中，以为不存在共享地址空间，所以已用程序无需加以区别。当然，16 位 Windows 系统和 32 位 Windows 系统中的 Hinstance 还有些联系：在 32 位 Windows 系统中可以直接调用 GetModuleHandle 以取得指向 DLL 的指针，通过指针访问该 DLL Module 的内容。

```
HMODULE GetModuleHandle(LPCTSTR lpModuleName);
```

当调用该函数时，传递一个可执行文件或 DLL 文件名字符串，如果系统找到文件，则返回该可执行文件或 DLL 文件映像加载到的基地址。也可调用 GetModuleHandle，传递 NULL 参数，则返回调用的可执行文件的基地址。

基地址的值是由 PE 文件本身设定的。按照默认设置，用 Visual C++建立的 EXE 文件基地址是 00400000h，DLL 文件基地址是 10000000h。但是，可以在创建应用程序的 EXE 文件时改变这个地址，方法是在链接应用时使用链接程序的/BASE 选项，或者链接后通过 REBASE 应用程序进行设置。

10.1.2 相对虚拟地址

在可执行文件中，有许多地方需要指定内存中的地址。例如，引用全局变量时，需要指定它的地址。PE 文件尽管有一个首选的载入地址（基地址），但是它们可以载入到进程空间的任何地方，所以不能依赖于 PE 的载入点。由于这个原因，必须有一个方法来指定地址而不依赖于 PE 载入点的地址。

为了在PE文件中避免有确定的内存地址，出现了相对虚拟地址（Relative Virtual Address，简称RVA）概念。RVA只是内存中的一个简单的相对于PE文件装入地址的偏移位置，它是一个“相对”地址，或称为“偏移量”。例如，假设一个EXE文件从地址400000h处装入，并且它的代码区块开始于401000h，代码区块的RVA将是：

```
目标地址 401000h - 装入地址 400000h = RVA 1000h
```

将一个RVA转换成真实的地址，只是简单地翻转这个过程：将实际的装入地址加上RVA即可得到实际的内存地址。顺便说一下，在PE用语里，实际的内存地址被称作虚拟地址（Virtual Address，简称VA），另外也可以把虚拟地址想象为加上首选装入地址的RVA。不要忘了前面提到的装入地址等同于模块句柄。它们之间的关系如下：

虚拟地址（VA）= 基地址（ImageBase）+ 相对虚拟地址（RVA）

10.1.3 文件偏移地址

当PE文件储存在磁盘上时，某个数据的位置相对于文件头的偏移量，称为文件偏移地址（File Offset）或物理地址（RAW Offset）。文件偏移地址从PE文件的第一个字节开始计数，起始值为0。用十六进制工具（例如Hex Workshop、WinHex等）打开文件所显示的地址就是文件偏移地址。

10.2 MS-DOS头部

每个PE文件是以一个DOS程序开始的，有了它，一旦程序在DOS下执行，DOS就能识别出这是有效的执行体，然后运行紧随MZ header之后的DOS stub（DOS块）。DOS stub实际上是一个有效的EXE，在不支持PE文件格式的操作系统中，它将简单显示一个错误提示，类似于字符串“This program cannot be run in MS-DOS mode”。程序员也可根据自己的意图实现完整的DOS代码。用户通常对DOS stub不太感兴趣，因为大多数情况下它由汇编器/编译器自动生成。平常把DOS MZ头与DOS stub合称为DOS文件头。

PE文件的第一个字节起始于一个传统的MS-DOS头部，被称作IMAGE_DOS_HEADER。其IMAGE_DOS_HEADER结构如下所示（左边的数字是到文件头的偏移量）：

```
IMAGE_DOS_HEADER_STRUCT{
  +0h   e_magic      WORD  ?      ;DOS可执行文件标记 “MZ”
  +2h   e_cblp       WORD  ?
  +4h   e_cp         WORD  ?
  +6h   e_crlc       WORD  ?
  +8h   e_cparhdr    WORD  ?
  +0ah  e_minalloc   WORD  ?
  +0ch  e_maxalloc   WORD  ?
  +0eh  e_ss         WORD  ?
  +10h  e_sp         WORD  ?
  +12h  e_csum       WORD  ?
  +14h  e_ip         WORD  ?      ;DOS代码入口IP
  +16h  e_cs         WORD  ?      ;DOS代码的入口CS
  +18h  e_lfarlc     WORD  ?
  +1ah  e_ovno       WORD  ?
  +1ch  e_res        WORD  4 dup(?)
  +24h  e_oemid      WORD  ?
  +26h  e_oeminfo    WORD  ?
  +28h  e_res2       WORD 10 dup(?)
  +3ch  e_lfanew     DWORD ?      ;指向PE文件头 "PE",0,0
} IMAGE_DOS_HEADER_ENDS
```

其中有两个字段比较重要，分别是 e_magic 和 e_lfanew。e_magic 字段（一个字大小）需要被设置为值 5A4Dh，这个值有个#define，名为 IMAGE_DOS_SIGNATURE，ASCII 表示法里，它的 ASCII 值为“MZ”，是 MS-DOS 的最初创建者之一 Mark Zbikowski 字母的缩写。e_lfanew 字段是真正 PE 文件头的相对偏移（RVA），其指出真正 PE 头的文件偏移位置，它占用 4 个字节，位于文件开始偏移 3Ch 字节中。

用十六进制编辑器（WinHex、Hex Workshop 等带偏移量显示的尤佳）打开光盘映像文件上的 PE.exe 文件，定位在文件起始位置处，此处就是 MD-DOS 头部，如图 10.3 所示。文件第一个字符“MZ”就是 e_magic 字段。偏移 3Ch 就是 e_lfanew 的值，显示为“B0000000”。由于 Intel CPU 属于 Little-Endian 类，字符储存时低位在前，高位在后，将次序恢复后，e_lfanew 的值为 000000B0h，这个值就是真正的 PE 文件头偏移量。

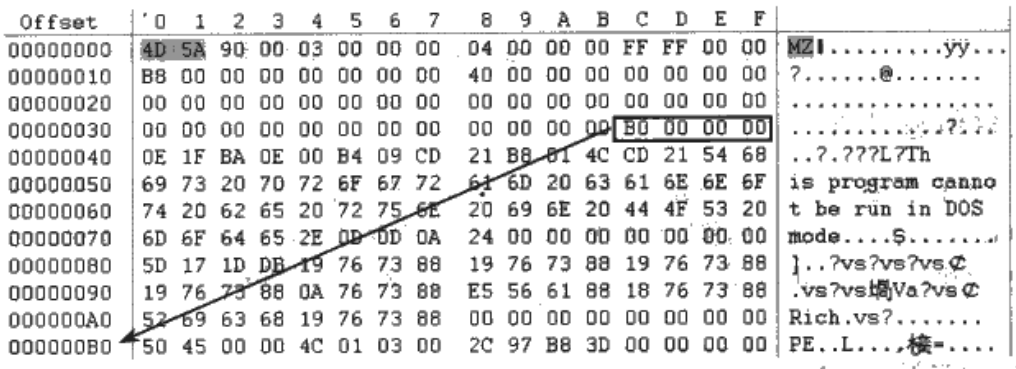


图 10.3　查看 PE 文件 MD-DOS 头部

10.3 PE 文件头

紧跟着 DOS stub 的是 PE 文件头（PE Header）。PE Header 是 PE 相关结构 NT 映像头（IMAGE_NT_HEADERS）的简称，其中包含许多 PE 装载器用到的重要字段。执行体在支持 PE 文件结构的操作系统中执行时，PE 装载器将从 IMAGE_DOS_HEADER 结构中的 e_lfanew 字段里找到 PE Header 的起始偏移量，加上基址得到 PE 文件头的指针。

PNTHeader = ImageBase + dosHeader->e_lfanew

实际上有两个版本的 IMAGE_NT_HEADER 结构，一个是为 32 位的可执行文件准备的，另一个是 64 位版本。在以后讨论中将不作考虑，它们几乎没有区别。

IMAGE_NT_HEADER 是由三个字段组成（左边的数字是到 PE 文件头的偏移量）：

```
IMAGE_NT_HEADERS STRUCT
  +0h   Signature           DWORD                       ?     ; PE 文件标识
  +4h   FileHeader          IMAGE_FILE_HEADER           <>
  +18h  OptionalHeader      IMAGE_OPTIONAL_HEADER32     <>
IMAGE_NT_HEADERS ENDS
```

10.3.1 Signature 字段

在一个有效的 PE 文件里，Signature 字段被设置为 00004550h，ASCII 码字符是“PE00”，#define MAGE_NT_SIGNATURE 定义了这个值。

```
#define IMAGE_NT_SIGNATURE              0x00004550
```

“PE\0\0”字串是 PE 文件头的开始，DOS 头部的 e_lfanew 字段正是指向“PE\0\0”，如图 10.3 所示。

10.3.2　IMAGE_FILE_HEADER 结构

IMAGE_FILE_HEADER（映像文件头）结构包含了 PE 文件的一些基本信息，最重要的是其中一个域指出了 IMAGE_OPTIONAL_HEADER 的大小。下面介绍 IMAGE_FILE_HEADER 结构的各个字段以及对这些字段的额外说明，这个结构也能在 COFF 格式的 OBJ 文件的最开始处找到，因此也称为 COFF File Header。字段前的注释标出了字段相对于 PE 文件头的偏移量。

```
IMAGE_FILE_HEADER STRUC
  +04h Machine              WORD    ? ; 运行平台
  +06h NumberOfSections     WORD    ? ; 文件的区块数目
  +08h TimeDateStamp        DWORD   ? ; 文件创建日期和时间
  +0Ch PointerToSymbolTable DWORD   ? ; 指向符号表（用于调试）
  +10h NumberOfSymbols      DWORD   ? ; 符号表中符号个数（用于调试）
  +14h SizeOfOptionalHeader WORD    ? ; IMAGE_OPTIONAL_HEADER32 结构大小
  +16h Characteristics      WORD    ? ; 文件属性
IMAGE_FILE_HEADER ENDS
```

IMAGE_FILE_HEADER 结构用十六进制工具显示情况如图 10.4 所示，图中的标号依次对应着相应的字段。

```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
                                (1)(2)    (3)       (4)
000000B0  50 45 00 00 4C 01 03 00 2C 97 B8 3D 00 00 00 00  PE..L...,棱=....
             (5)          (6)(7)
000000C0  00 00 00 00 E0 00 0F 01 0B 01 05 0C 00 02 00 00  ....?...........
```

图 10.4　IMAGE_FILE_HEADER 结构

（1）Machine：可执行文件的目标 CPU 类型。PE 文件可以在多种机器上使用，不同平台指令机器码是不同的。表 10-1 中所示是几种典型的机器类型标志。

表 10-1　机器类型标志

机　器	标　志
Intel i386	14Ch
MIPS R3000	162h
MIPS R4000	166h
Alpha AXP	184h
Power PC	1F0h

（2）NumberOfSections：区块（Section）的数目，块表紧跟在 IMAGE_NT_HEADERS 后面。

（3）TimeDateStamp：表明文件是何时被创建的。这个值是自 1970 年 1 月 1 日以来用格林威治时间（GMT）计算的秒数，这个值是一个比文件系统的日期/时间更精确的文件创建时间指示器。将这个值翻译为易读的字符串的方法是用_ctime 函数（它是时区敏感型的），另一个对此字段计算有用的函数是 gmtime。

（4）PointerToSymbolTable：COFF 符号表的文件偏移位置，描述于 Microsoft 规范的第 5.4 节。COFF 符号表在 PE 文件中较少见，因为已采用了较新的 debug 格式。在 Visual Studio .NET 之前，COFF 符号表可以通过设置链接器开关/DEBUGTYPE:COFF 来创建。COFF 符号表几乎总能在目标文件中找到，如果没有符号表存在，将此值设为 0。

（5）NumberOfSymbols：如果有 COFF 符号表，它代表其中的符号数目，COFF 符号是一个大小固定的结构，如果想找到 COFF 符号表的结束处，这个域是需要的。

（6）SizeOfOptionalHeader：紧跟着 IMAGE_FILE_HEADER 后面的数据大小。在 PE 文件中，这个数据结构叫 IMAGE_OPTIONAL_HEADER，其大小依赖于是 32 位还是 64 位文件。对于 32 位 PE 文件，这个域通常是 00E0h；对于 64 位 PE32+文件，这个域是 00F0h。不管怎么样，这些是要求的最小值，较大的值可能也会出现。

（7）Characteristics：文件属性，有选择地通过几个值的运算得到，这些标志的有效值是定义于 winnt.h 内的 IMAGE_FILE_xxx 值，具体值见表 10-2。普通的 EXE 文件这个字段的值一般是 010fh，DLL 文件这个字段的值一般是 0210h。

表 10-2　属性位字段的含义

特征值	含义
0001h	文件中不存在重定位信息
0002h	文件可执行。如果为 0，一般是链接时出问题了
0004h	行号信息被移去
0008h	符号信息被移去
0020h	应用程序可以处理超过 2GB 的地址。该功能是从 NT SP3 开始被支持，因为大部分的数据库服务器需要很大的内存，而 NT 仅提供 2GB 给应用程序。从 NT SP3 开始，通过加载/3GB 参数，可以使应用程序分配到 2～3GB 区域的地址，而该处原先’属于系统内存区
0080h	处理机的低位字节是相反的
0100h	目标平台为 32 位机器
0200h	.DBG 文件的调试信息被移去
0400h	如果映像文件是在可移动媒体中，则先复制到交换文件后再运行
0800h	如果映像文件是在网络中，则复制到交换文件后才运行
1000h	系统文件
2000h	文件是 DLL 文件
4000h	文件只能运行在单处理器上
8000h	处理机的高位字节是相反的

10.3.3　IMAGE_OPTIONAL_HEADER 结构

可选映像头（IMAGE_OPTIONAL_HEADER）是一个可选的结构，但实际上 IMAGE_FILE_HEADER 结构不足以定义 PE 文件属性，因此可选映像头中定义了更多的数据，完全不必考虑两个结构区别在哪里，两者连起来就是一个完整的“PE 文件头结构”。IMAGE_OPTIONAL_HEADER32 结构如下，字段前的注释标出了字段相对于 PE 文件头的偏移量。

```
IMAGE_OPTIONAL_HEADER32 STRUC
  +18h Magic                        WORD    ?  ; 标志字
  +1Ah MajorLinkerVersion           BYTE    ?  ; 链接器主版本号
  +1Bh MinorLinkerVersion           BYTE    ?  ; 链接器次版本号
  +1Ch SizeOfCode                   DWORD   ?  ; 所有含有代码区块的总大小
  +20h SizeOfInitializedData        DWORD   ?  ; 所有初始化数据区块总大小
  +24h SizeOfUninitializedData      DWORD   ?  ; 所有未初始化数据区块总大小
  +28h AddressOfEntryPoint          DWORD   ?  ; 程序执行入口 RVA
  +2Ch BaseOfCode                   DWORD   ?  ; 代码区块起始 RVA
  +30h BaseOfData                   DWORD   ?  ; 数据区块起始 RVA
  +34h ImageBase                    DWORD   ?  ; 程序默认装入基地址
  +38h SectionAlignment             DWORD   ?  ; 内存中区块的对齐值
  +3Ch FileAlignment                DWORD   ?  ; 文件中区块的对齐值
  +40h MajorOperatingSystemVersion  WORD    ?  ; 操作系统主版本号
  +42h MinorOperatingSystemVersion  WORD    ?  ; 操作系统次版本号
  +44h MajorImageVersion            WORD    ?  ; 用户自定义主版本号
  +46h MinorImageVersion            WORD    ?  ; 用户定义次版本号
  +48h MajorSubsystemVersion        WORD    ?  ; 所需要子系统主版本号
  +4Ah MinorSubsystemVersion        WORD    ?  ; 所需要子系统次版本号
  +4Ch Win32VersionValue            DWORD   ?  ; 保留，通常被设置为 0
  +50h SizeOfImage                  DWORD   ?  ; 映像装入内存后的总尺寸
  +54h SizeOfHeaders                DWORD   ?  ; DOS 头、PE 头部、区块表总大小
```

```
    +58h CheckSum                   DWORD    ? ; 映像校验和
    +5Ch Subsystem                  WORD     ? ; 文件子系统
    +5Eh DllCharacteristics         WORD     ? ; 显示 DLL 特性的旗标
    +60h SizeOfStackReserve         DWORD    ? ; 初始化堆栈大小
    +64h SizeOfStackCommit          DWORD    ? ; 初始化实际提交堆栈大小
    +68h SizeOfHeapReserve          DWORD    ? ; 初始化保留堆栈大小
    +6Ch SizeOfHeapCommit           DWORD    ? ; 初始化实际保留堆栈大小
    +70h LoaderFlags                DWORD    ? ; 与调试相关，默认值为 0
    +74h NumberOfRvaAndSizes        DWORD    ? ; 数据目录表的项数
    +78h DataDirectory              IMAGE_DATA_DIRECTORY 16 DUP (<0>)
IMAGE_OPTIONAL_HEADER32 ENDS
```

IMAGE_OPTIONAL_HEADER32 结构用十六进制工具显示情况如图 10.5 所示，图中的标号依次对应着相应的字段。

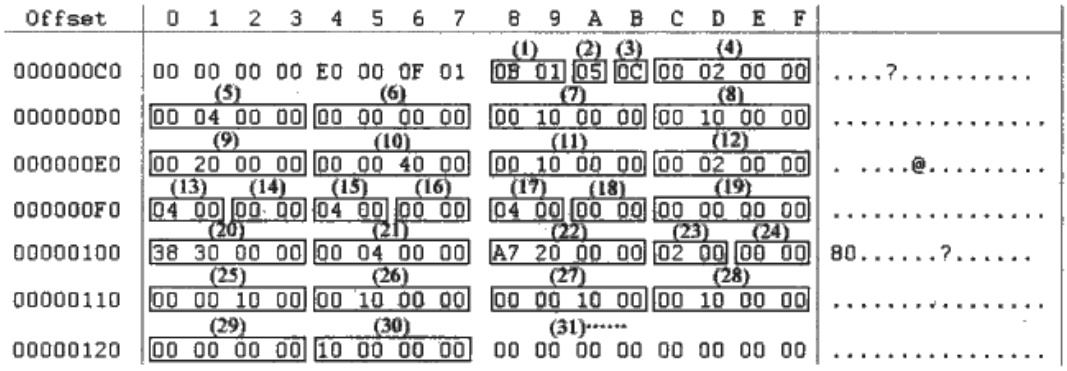


图 10.5　IMAGE_OPTIONAL_HEADER32 结构

（1）Magic：是一个标记字，说明文件是 ROM 映像（0107h），还是普通可执行的映像（010Bh），一般是 010Bh，如是 PE32+，则是 020Bh。

（2）MajorLinkerVersion：链接程序的主版本号。

（3）MinorLinkerVersion：链接程序的次版本号。

（4）SizeOfCode：所有带有 IMAGE_SCN_CNT_CODE 属性区块的总共大小（只入不舍），这个值是向上对齐某一个值的整数倍。例如，本例是 200h，即对齐的是一个磁盘扇区字节数（200h）的整数倍。通常情况下，多数文件只有一个 Code 块，所以这个字段和.text 块的大小匹配。

（5）SizeOfInitializedData：已初始化数据块的大小，即在编译时所构成的块的大小（不包括代码段）。但这个数据并不太准确。

（6）SizeOfUninitializedData：未初始化数据块的大小，装载程序要在虚拟地址空间中为这些数据约定空间。这些块在磁盘文件中不占空间，就像“UninitializedData”这一术语所暗示的一样，这些块在程序开始运行时没有指定值。未初始化数据通常在.bbs 块中。

（7）AddressOfEntryPoint：程序执行入口 RVA。对于 DLL，这个入口点是在进程初始化和关闭时以及线程创建/毁灭时被调用。在大多数可执行文件中，这个地址并不直接指向 Main、WinMain 或 DllMain，而是指向运行时库代码并由它来调用上述的函数。在 DLL 中这个域能被设置为 0，前面提到的通知消息都不能收到。链接器/NOENTRY 开关可以设置这个域为 0。

（8）BaseOfCode：代码段的起始 RVA。在内存中，代码段通常在 PE 文件头之后、数据块之前。在 Microsoft 链接器生成的执行文件中，RVA 通常是 1000h。Borland 的 Tlink32 是将 ImageBase 加上第一个 Code Section 的 RVA，并将结果存入该字段。

（9）BaseOfData：数据段的起始 RVA。数据段通常是在内存的末尾，即 PE 文件头和 Code Section 之后。可是，这个域的值对于不同版本的微软链接器是不一致的，在 64 位可执行文件中是不出现的。

（10）ImageBase：文件在内存中的首选装入地址。如果有可能（也就是说，目前如果没有其他占据这

块地址，它是正确对齐的并且是一个合法的地址，等等），加载器试图在这个地址装入 PE 文件。如果可执行文件是在这个地址装入的，那么加载器将跳过应用基址重定位的步骤。

（11）SectionAlignment：当被装入内存时的区块对齐大小。每个区块被装入的地址必定是本字段指定数值的整数倍。默认的对齐尺寸是目标 CPU 的页尺寸。对于运行在 Windows 9x/Me 下的用户模式可执行文件，最小的对齐尺寸是一页 1000h（4KB）。这个字段可以通过链接器的/ALIGN 开关来设置。在 IA-64 上，是按 8KB 来排列的。

（12）FileAlignment：磁盘上 PE 文件内的区块对齐大小，组成块的原始数据必须保证从本字段的倍数地址开始。对于 x86 可执行文件，这个值通常是 200h 或 1000h，这是为了保证块总是从磁盘的扇区开始，这个字段的功能等价于 NE 格式文件中的段/资源对齐因子。用不同版本的微软链接器默认值会改变。这个值必须是 2 的幂，其最小值为 200h，并且如果 SectionAlignment 小于 CPU 的页尺寸，这个域必须与 SectionAlignment 匹配。链接器开关/OPT:WIN98 设置 x86 可执行文件的文件对齐为 1000h，/OPT:NOWIN98 设置对齐为 200h。

（13）MajorOperatingSystemVersion：要求操作系统的最低版本号的主版本号。随着这么多版本的 Windows 的到来，这个字段明显地变得不切题了。

（14）MinorOperatingSystemVersion：要求操作系统的最低版本号的次版本号。

（15）MajorImageVersion：该可执行文件的主版本号，由程序员定义。它不被系统使用并可以设置为 0，可以通过链接器的/VERSION 开关设置它。

（16）MinorImageVersion：该可执行文件的次版本号，由程序员定义。

（17）MajorSubsystemVersion：要求最低子系统版本的主版本号。这个值与下一个字段一起，通常被设置为 4，可以通过链接器开关/SUBSYSTEM 来设置。

（18）MinorSubsystemVersion：要求最低子系统版本的次版本号。

（19）Win32VersionValue：另一个从来不用的字段，通常被设置为 0。

（20）SizeOfImage：映像装入内存后的总尺寸。它指装入文件从 Image Base 到最后一个块的大小。最后一个块根据其大小往上取整。

（21）SizeOfHeaders：是 MS-DOS 头部、PE 头部、区块表的组合尺寸。所有这些项目都出现在 PE 文件中任何代码或数据区块之前。域值四舍五入至文件对齐的倍数。

（22）CheckSum：映像的校验和。IMAGEHLP.DLL 中的 CheckSumMappedFile 函数可以计算这个值。一般的 EXE 文件可以是 0，但一些内核模式的驱动程序和系统 DLL 必须有一个检验和。当链接器的/RELEASE 开关被使用时，校验和被置于文件中。

（23）Subsystem：一个标明可执行文件所期望的子系统（用户界面类型）的枚举值。这个值只对 EXE 是重要的，具体见表 10-3。

表 10-3　界面子系统含义

值	子系统
0	未知
1	不需要子系统（如驱动程序）
2	图形接口子系统（GUI）
3	字符子系统（CUI）
5	OS/2 字符子系统
7	POSIX 字符子系统
8	保留
9	Windows CE 图形界面

（24）DllCharacteristics：DllMain()函数何时被调用，默认为 0。

（25）SizeOfStackReserve：在 EXE 文件里，为线程保留的堆栈大小。它一开始只提交其中一部分，只有在必要时，才提交剩下的部分。

（26）SizeOfStackCommit：在 EXE 文件里，一开始即被委派给堆栈的内存数量。默认值是 4KB。

（27）SizeOfHeapReserve：在 EXE 文件里，为进程的默认堆保留的内存。默认值是 1MB，但是在当前

版本的 Windows 里，堆值在用户不干涉的情况下就能增长超过这个值。

（28）SizeOfHeapCommit：在 EXE 文件里，委派给堆的内存大小。默认值是 4KB。

（29）LoaderFlags：与调试有关，默认为 0。

（30）NumberOfRvaAndSizes：数据目录的项数。这个字段从最早的 Windows NT 发布以来一直是 16。

（31）DataDirectory[16]：数据目录表，由数个相同的 IMAGE_DATA_DIRECTORY 结构组成，指向输出表、输入表、资源块等数据。IMAGE_DATA_DIRECTORY 的结构定义如下：

```
IMAGE_DATA_DIRECTORY    STRUC
  VirtualAddress    DWORD    ?        ; 数据块的起始 RVA
  Size              DWORD    ?        ; 数据块的长度
IMAGE_DATA_DIRECTORY    ENDS
```

数据目录表成员的结构如表 10-4 所示，各项成员含义后面会介绍。

表 10-4 数据目录表成员

序 号	成 员	结 构	偏 移 量
0	Export Table	IMAGE_DIRECTORY_ENTRY_EXPORT	78h
1	Import Table	IMAGE_DIRECTORY_ENTRY_IMPORT	80h
2	Resources Table	IMAGE_DIRECTORY_ENTRY_RESOURCE	88h
3	Exception Table	IMAGE_DIRECTORY_ENTRY_EXCEPTION	90h
4	Security Table	IMAGE_DIRECTORY_ENTRY_SECURITY	98h
5	Base relocation Table	IMAGE_DIRECTORY_ENTRY_BASERELOC	A0h
6	Debug	IMAGE_DIRECTORY_ENTRY_DEBUG	A8h
7	Copyright	IMAGE_DIRECTORY_ENTRY_COPYRIGHT	B0h
8	Global Ptr	IMAGE_DIRECTORY_ENTRY_GLOBALPTR	B8h
9	Thread local storage (TLS)	IMAGE_DIRECTORY_ENTRY_TLS	C0h
10	Load configuration	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	C8h
11	Bound Import	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	D0h
12	Import Address Table(IAT)	IMAGE_DIRECTORY_ENTRY_IAT	D8h
13	Delay Import	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	E0h
14	COM descriptor	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	E8h
15	保留		

PE 文件中定位输出表、输入表和资源等重要数据时，就是从 IMAGE_DATA_DIRECTORY 结构开始的。本例数据目录表位于 128h～1A7h 之间，每个成员占 8 个字节，分别指向相关的结构，如图 10.6 所示。

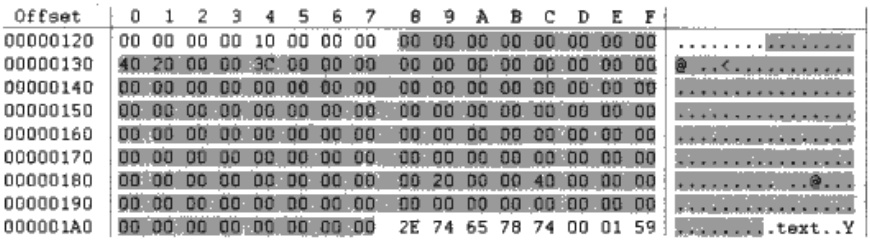


图 10.6 数据目录表

在图 10.6 中，地址 128h 就是数据目录表的第一项，其值为 0，即这个实例的输出表地址与大小皆为 0，表示无输出表。地址 130h 是第二项，该组数据表示输入表地址为 2040h（RVA），大小为 3Ch。

用 PE 编辑工具（如 LordPE）来查看实例 PE.exe 文件的 PE 结构。单击 LordPE 的“PE Editor”打开 PE_Offset 文件，面板上直接显示出 PE 结构中的主要字段，如图 10.7 所示。

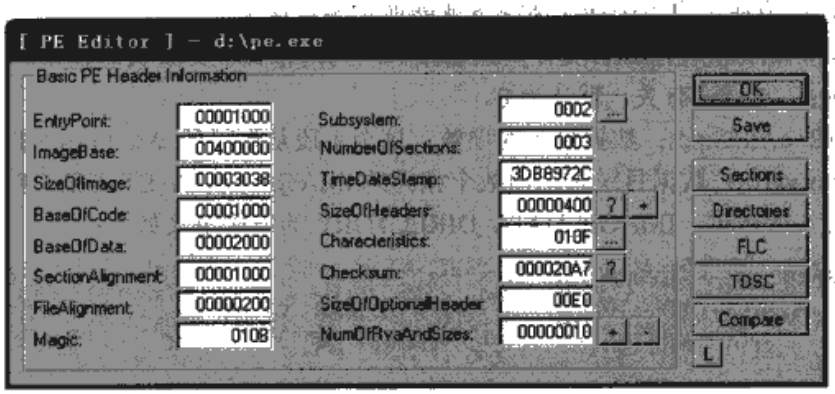


图 10.7　用 LordPE 查看文件 PE 信息

然后单击“Directories”按钮，打开数据目录表查看面板，如图 10.8 所示。

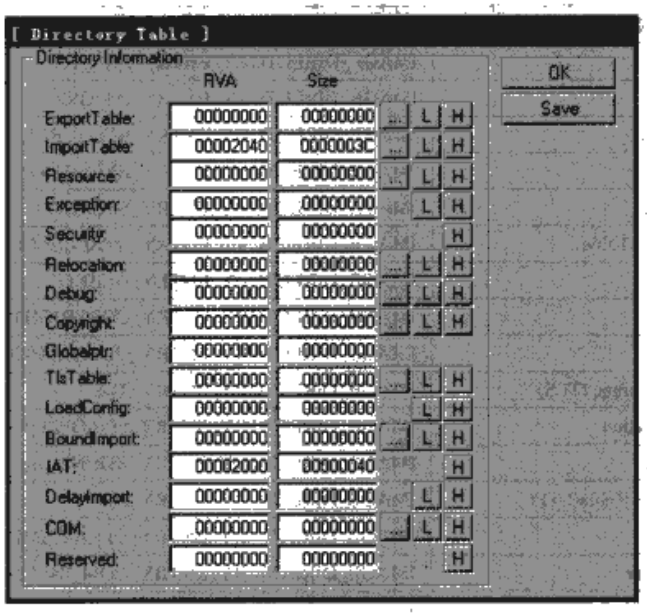


图 10.8　用 LordPE 查看 PE 文件数据目录表结构

10 区块

在 PE 文件头与原始数据之间存在一个区块表（Section Table），区块表包含每个块在映像中的信息，分别指向不同的区块实体。

10.4.1　区块表

紧跟着 IMAGE_NT_HEADERS 后的是区块表，它是一个 IMAGE_SECTION_HEADER 结构数组。每个 IMAGE_SECTION_HEADER 结构包含了它所关联区块的信息，如位置、长度、属性；该数组的数目由 IMAGE_NT_HEADERS.FileHeader.NumberOfSections 指出。

IMAGE_SECTION_HEADER 结构定义如下：

```
IMAGE_SECTION_HEADER   STRUC
  Name                      DBYTE 8 DUP (?)  ; 8 个字节的块名
  union Misc                                 ; 区块尺寸
    PhysicalAddress      DWORD  ?
```

```
    VirtualSize         DWORD  ?
  Ends
  VirtualAddress            DWORD  ?        ; 区块的 RVA 地址
  SizeOfRawData             DWORD  ?        ; 在文件中对齐后的尺寸
  PointerToRawData          DWORD  ?        ; 在文件中偏移
  PointerToRelocations      DWORD  ?        ; 在 OBJ 文件中使用，重定位的偏移
  PointerToLinenumbers      DWORD  ?        ; 行号表的偏移（供调试用）
  NumberOfRelocations       WORD   ?        ; 在 OBJ 文件中使用，重定位项数目
  NumberOfLinenumbers       WORD   ?        ; 行号表中行号的数目
  Characteristics           DWORD  ?        ; 区块的属性
IMAGE_SECTION_HEADER  ENDS
```

为了方便理解，结合实例分析一下，实例 PE.exe 的区块表含有 3 个块的描述：.text、.rdata 和.data。每个块对应一个 IMAGE_SECTION_HEADER 结构，用十六进制工具查看块表如图 10.9 所示。图中的标号依次对应着第一个 IMAGE_SECTION_HEADER 结构中的相应字段。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000001A0	00	00	00	00	00	00	00	00	2E	74	65	78	74	00	01	59	.....text..Y
000001B0	9A	01	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	C2	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	38	00	00	00	00	30	00	00	00	02	00	00	00	08	00	00	8....0.....
00000210	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00	C0	.....@..@

图 10.9 十六进制工具中的块表

在图 10.7 中单击“Sections”按钮，打开区块编辑器（见图 10.10），这是图 10.9 内容的另一种表现形式。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	0000019A	00000400	00000200	60000020
.rdata	00002000	000001C2	00000600	00000200	40000040
.data	00003000	00000038	00000800	00000200	C0000040

图 10.10 LordPE 查看的块表

（1）Name：块名。这是一个 8 位 ASCII 码名（不是 Unicode 内码），用来定义块名。多数块名以一个“.”开始（如.text），这个“.”实际上不是必需的。值得注意的是，如果块名超过 8 个字节，则没有最后的终止标志“NULL”字节。带有一个“$”的区块名字会从链接器那里得到特殊的对待，前面带有“$”的相同名字的区块被合并，在合并后的区块中，它们是按“$”后面的字符字母顺序进行合并的。

（2）VirtualSize：指出实际的、被使用的区块大小，是区块在没对齐处理前的实际大小。如果 VirtualSize 大于 SizeOfRawData，那么 SizeOfRawData 是来自可执行文件初始化数据的大小，与 VirtualSize 相差的字节用零填充。这个字段在 OBJ 文件中是被设为 0 的。

（3）VirtualAddress：该块装载到内存中的 RVA。这个地址是按照内存页对齐的，它的数值总是 SectionAlignment 的整数倍。在 Microsoft 工具中，第一个块的默认 RVA 为 1000h。在 OBJ 中，该字段没有意义，并被设为 0。

（4）SizeOfRawData：该块在磁盘文件中所占的大小。在可执行文件中，该字段包含经过 FileAlignment 调整后的块的长度。例如，指定 FileAlignment 的大小为 200h，如果 VirtualSize 中的块长度为 19Ah 个字节，这一块应保存的长度为 200h 个字节。

（5）PointerToRawData：该块在磁盘文件中的偏移。程序经编译或汇编后生成原始数据，这个字段用于给出原始数据在文件中的偏移。如果程序自装载 PE 或 COFF 文件（而不是由操作系统装入），这一字段比 VirtualAddress 还重要。在这种状态下，必须完全使用线性映像方法装入文件，所以需要在该偏移处找到块的数据，而不是 VirtualAddress 字段中的 RVA 地址。

（6）PointerToRelocations：这部分在 EXE 文件中无意义。在 OBJ 文件中，表示本块重定位信息的偏移值。在 OBJ 文件中如果不是零，它会指向一个 IMAGE_RELOCATION 结构数组。

（7）PointerToLinenumbers：行号表在文件中的偏移值。这是文件的调试信息。

（8）NumberOfRelocations：这部分在 EXE 文件中无意义。在 OBJ 文件中，是本块在重定位表中的重定位数目。

（9）NumberOfLinenumbers：该块在行号表中的行号数目。

（10）Characteristics：块属性。该字段是一组指出块属性（如代码/数据/可读/可写等）的标志。比较重要的标志如表 10-5 所示，多个标志值求或即为 Characteristics 的值。这些标志中的很多都可以通过链接器的/SECTION 选项设置。

表 10-5　字段属性

字段值		用途
IMAGE_SCN_CNT_CODE	00000020h	包含代码，常与 10000000h 一起设置
IMAGE_SCN_CNT_INITIALIZED_DATA	00000040h	该块包含已初始化的数据
IMAGE_SCN_CNT_UNINITIALIZED_DATA	00000080h	该块包含未初始化的数据
IMAGE_SCN_MEM_DISCARDABLE	02000000h	该块可被丢弃，因为它一旦被装入后，进程就不再需要它了常见的可丢弃块是.reloc（重定位块）
IMAGE_SCN_MEM_SHARED	10000000h	该块为共享块
IMAGE_SCN_MEM_EXECUTE	20000000h	该块可以执行。通常当 00000020h 标志被设置时，该标志也被设置
IMAGE_SCN_MEM_READ	40000000h	该块可读。可执行文件中的块总是设置该标志
IMAGE_SCN_MEM_WRITE	80000000h	该块可写。如果可执行文件没有设置该标志，装载程序就会将内存映像页标记为可读或可执行

例如，E0000020h=20000000h | 40000000h | 80000000h | 00000020h 表示该块包含执行代码，可读、可写并可执行。C00000040h=40000000h | 80000000h | 00000040h 表示该块可读、可写，包含已初始化的数据。60000020h=20000000h | 40000000h | 00000020h 表示该块包含执行代码，可读并可执行。

10.4.2　各种区块的描述

通常，区块中的数据逻辑上是关联的。PE 文件一般至少有两个区块：一个是代码块，另一个是数据块。每一个区块都有一个截然不同的名字，这个名字是用来传达区块的用途。例如，一个区块叫.rdata，表明它是一个只读区块。区块在映像中是按起始地址（RVA）来排列的，而不是按字母表顺序。使用区块名字只是人们为了方便，而对操作系统来说是无关紧要的。微软给这些区块取了个有特色的名字，但这不是必需的。Borland 的链接器用的是 CODE 和 DATA 这样的名字。

现在看看 EXE 和 OBJ 文件的一些常见区块（见表 10-6），除非另外声明，表中的区块名称来自于微软定义。

表 10-6　区块名称

名称	描述
.text	默认的代码区块。它的内容全是指令代码。PE 文件运行在 32 位方式下，不受 16 位段的约束，所以没有理由把代码放到不同的区块中。链接器把所有目标文件的.text 块链接成一个大的.text 块。如果使用的是 Borland C++，其编译器将产生的代码存于名为 code 的区域中，其链接器链接的结果使代码块的名称不是.text 而是 code
.data	默认的读/写数据区块。全局变量、静态变量一般放在这里
.rdata	默认的只读数据区块。但程序中很少用到该块中的数据。至少有两种情况下要用到.rdata。一是在 Microsoft 的链接器产生的 EXE 文件中，用于存放调试目录；二是用于存放说明字符串。如果程序的 DEF 文件中指定了 DESCRIPTION，字符串就会出现在.rdata 中
.idata	包含其他外来 DLL 的函数及数据信息，即输入表。将.idata 区块合并到另一个区块现在已成为了惯例，典型的是.rdata 区块。默认地，链接器仅在创建一个 Release 模式的可执行文件时才将.idata 合并到另外一个区块中

续表

.edata	输出表。当创建一个输出 API 或数据的可执行文件时，链接器会创建一个.EXP 文件，这个.EXP 文件包含一个.edata 区块，其会被加入到最后的可执行文件中。与.idata 区块一样，.edata 区块也经常被发现合并到了.text 或.tdata 区块中
.rsrc	资源，包含模块的全部资源，如图标、菜单、位图等。这个区块是只读的，无论如何它不应该命名为.rsrc 以外的其他任何名字，也不能被合并到其他的区块里
.bss	未初始化数据。很少在用，取而代之的是执行文件的.data 区块的 VirtualSize 被扩展到足够大的空间用来装下未初始化数据
.crt	用于支持 C++运行时（CRT）所添加的数据
.tls	TLS 的意思是线程局部存储器，用于支持通过__declspec(thread)声明的线程局部存储变量的数据。这包括数据的初始化值，也包括运行时所需要的额外变量
.reloc	可执行文件的基址重定位。基址重定位一般仅是 DLL 需要的，而不是 EXE。在 Release 模式，链接器并不给 EXE 文件加上基址重定位，重定位可以在链接时通过/FIXED 开关去掉
.sdata	相对于全局指针的可被定位的“短的”读/写数据。用于 IA-64 和其他使用一个全局指针寄存器的体系结构。IA-64 上的常规大小的全局变量放在这个区块里
.srdata	相对于全局指针的可被定位的“短的”只读数据。用于 IA-64 和其他使用一个全局指针寄存器的体系结构
.pdata	异常表。它包含一个 CPU 特定的 IMAGE_RUNTIME_FUNCTION_ENTRY 结构数组，DataDirectory 中的 IMAGE_DIRECTORY_ENTRY_EXCEPTION 指向它。它被用于异常处理是基于表的体系结构，像 IA-64。异常处理不使用基于表的唯一架构体系是 x86
.debug$S	OBJ 文件中 Codeview 格式的符号。这是一个变量长度的 Codeview 格式的符号记录流
.debug$T	OBJ 文件中 Codeview 格式的类型记录。这是一个变量长度的 Codeview 格式的类型记录流
.debug$P	当使用预编译的头时，它可以在 OBJ 文件中找到
.drectve	包含链接器命令，只能在 OBJ 中找到。命令是能被传递给链接器命令行的字符串例如：-defaultlib:LIBC 命令用空格字符分开
.didat	延迟装入的输入数据，只能在非 Release 模式的可执行文件中找到。在 Release 模式时，延迟装入数据被合并到另一个区块

注意：当编程从 PE 文件中读取需要的内容时，如输入表、输出表等，不能以区块名称作为参考，正确的方法是按照数据目录表中的字段进行定位。

虽然编译器自动产生一系列标准的区块，但这没有什么不可思议。读者可以创建和命名自己的区块。在 Visual C++中，用#pragma 来声明，告诉编译器插入数据到一个区块内，像下面这样：

```
#pragma data_seg( "MY_DATA" )
```

这样所有被 Visual C++处理的数据都将放到一个叫 MY_DATA 的区块内，而不是默认的.data 区块内。大部分的程序都只使用编译器产生的默认区块，但读者偶尔可能也会有一些特殊的需求，需要将代码或数据放到一个单独的区块里，例如建立一个全局共享块。

区块并不全部是在链接时形成的，更准确地说，它们一般是从 OBJ 文件开始，被编译器放置的。链接器的工作就是合并所有 OBJ 和库中需要的块，使其成为一个最终合适的区块。例如，在工程中的每一个 OBJ 至少都有一个包含代码的.text 区块，链接器把这些区块合并成单一的.text 区块。链接器遵循一套相当完整的规则，它判断哪些区块被合并以及如何被合并。OBJ 文件中的一个区块可能是为链接器而准备的，不会放入最后的可执行文件中，像这样的区块主要用于编译器向链接器传递信息。

链接器一个有趣的特征就是能够合并区块。如果两个区块有相似、一致的属性，那么它们在链接时能合并成一个单一区块。这取决于是否用/merge 开关。例如，下面的链接器选项将.rdata 与.text 区块合并为一个.text 的区块。

```
/MERGE:.rdata=.text
```

合并区块的优点是可以节省磁盘和内存的空间。每个区块至少占用一个内存页，如果能将可执行文件内的区块数从 4 个减少到 3 个，很可能少用一页内存。当然，这依赖于两个合并区块的结尾未用空间加起来是否能达到一页。

当合并区块时，事情将变得有趣，因为这没有什么硬性规定。例如，把.rdata 合并到.text 里不会有什么问题，但是不应该将.rsrc、.reloc 或.pdata 合并到其他的区块里。在 Visual Studio .NET 之前，能将.idata 合并到其他的区块里。在 Visual Studio .NET 中，这是不允许的。不过，当制作发行版本时，链接器经常将.idata 的一部分合并到其他的区块里，如.rdata。

既然部分输入数据是在被装入内存时由 Windows 加载器写入的，那么读者可能对它们是如何被放入到只读区块表示疑惑。这种情况的发生是由于在加载时，系统会临时改变那些包含输入数据的页属性为可读、可写，初始化完成后，又恢复原来的属性。

10.4.3 区块的对齐值

区块的大小是要对齐的，有两种对齐值，一种用于磁盘文件内，另一种用于内存中。PE 文件头指出了这两个值，它们可以不同。

PE 文件头里 FileAlignment 定义了磁盘区块的对齐值。每一个区块从对齐值的倍数的偏移位置开始。而区块的实际代码或数据的大小不一定刚好是这么多，所以在不足的地方一般以 00h 来填充，这就是区块间的间隙。例如，在 PE 文件中，一个典型的对齐值是 200h，这样，每个区块从 200h 之倍数的文件偏移位置开始，假设区块的第一个节在 400h 处，长度为 90h，那么从文件 400h 到 490h 为这一区块的内容，而文件对齐值是 200h，所以为了使这一节长度为 FileAlignment 的整数倍，490h 到 600h 会被用零填充，这段空间称为区块间隙，下一个区块的开始地址为 600h。

PE 文件头里 SectionAlignment 定义了内存中区块的对齐值。PE 文件被映射到内存中时，区块总是至少从一个页边界处开始，也就是说，当一个 PE 文件被映射到内存中，每个区块的第一个字节对应于某个内存页。在 x86 系列 CPU 中，页是按 4KB(1000h)来排列的；在 IA-64 上，是按 8KB(2000h)来排列的。所以在 x86 系统中，PE 文件区块的内存对齐值一般等于 1000h，每个区块按 1000h 之倍数的内存偏移位置开始。

再来回顾一下图 10.10，.text 区块在磁盘文件中的偏移位置是 400h，在内存中将是其装入地址之上的 1000h 字节处。同样，.rdata 区块在磁盘文件偏移的 600h 处，在内存中将是装入地址之上的 2000h 字节处。

Visual Studio 6.0 中的默认值是 4KB，除非使用/OPT:NOWIN98 或/ALIGN 开关。Visual Studio .NET 的链接器，依然用了默认的/OPT:WIN98，但是如果文件小于某一特定大小时，就会采用 200h 为对齐值。另一种对齐方式来自于.NET 文件的规定，它规定.NET 文件的内存对齐值为 8KB 而不是普通 x86 平台上的 4KB，这样就保证了在 x86 平台编译的程序可以在 IA-64 平台上运行。如果内存对齐值为 4KB，那么 IA-64 加载器就不能载入这个程序，因为 64 位 Windows 中的页是 8KB。

建立一个区块在文件中的偏移和在内存中的偏移相同的 PE 文件是可能的，这会使执行文件变大，但在 Windows 9x/Me 下可以加快装入速度。Visual Studio 6.0 的默认选项/OPT:WIN98 将会使 PE 文件按照这种方式来创建。在 Visual Studio .NET 中，链接器可以不使用/OPT:NOWIN98，这依赖于文件是否足够小。

10.4.4 文件偏移与虚拟地址转换

由于一些 PE 文件为减少体积，磁盘对齐值不是一个内存页 1000h，而是 200h，当这类文件被映射到内存后，同一数据相对于文件头的偏移量在内存中和磁盘文件中是不同的，这样就存在着文件偏移地址与虚拟地址的转换问题。而那些磁盘对齐值（1000h）与内存页相同的区块，同一数据在磁盘文件中的偏移和在内存中的偏移相同，不需要转换。

图 10.10 中区块显示出实例文件在磁盘与内存中各区块的地址、大小等信息。虚拟地址和虚拟大小是

指该区块在内存中的地址和大小。物理地址和物理大小是指该区块在磁盘文件中的地址和大小。由于其磁盘对齐值为 200h，与内存对齐值不同，故其磁盘映像和内存映像是不同的，如图 10.11 所示。

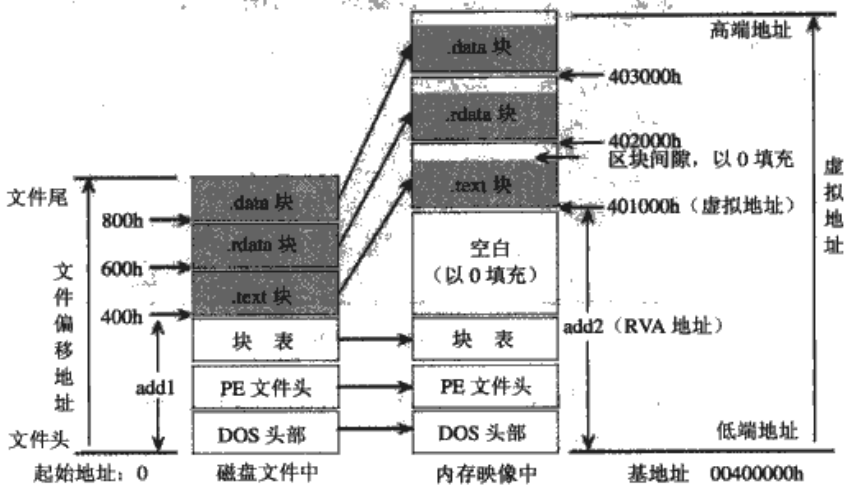


图 10.11　应用程序加载映射示意图

从图 10.11 中可看出，文件被映射到内存中，DOS 文件头、PE 文件头和块表的偏移位置与大小均没有变化。而各区块映射到内存后，其偏移位置就发生变化了。例如，磁盘文件中.text 块起始端与文件头的偏移量为 add1，映射到内存后，.text 起始端与文件头（基地址）的偏移量为 add2。同时，.text 区块与块表之间形成一大段空隙，这部分数据全是以 0 填充的。这里，add1 的值就是文件偏移地址（File Offset），add2 的值就是相对虚拟地址（RVA）。假设它们的差值为Δk，则文件偏移地址与虚拟地址关系如下：

$$\text{File Offset} = \text{RVA} - \Delta k$$
$$\text{File Offset} = \text{VA} - \text{ImageBase} - \Delta k$$

在同一区块中，各地址的偏移量是相等的，可用上面的公式对此区块中任一 File Offset 与 VA 进行转换。但请不要错误地认为在整个文件里，File Offset 与 VA 的差值是 Δk。因为各区块在内存中是以一个页边界为开始的，第一个区块结束后，一直到第二个区块起始端（1000h 对齐处）全以数据 0 填充，所以不同区块在磁盘与内存中的差值不一样。表 10-7 所示是该实例文件各区块在磁盘与内存中的起始地址差值。

表 10-7　各区块在磁盘与内存中的起始地址差值

区　块	虚拟偏移量（Virtual Offset）=RVA	文件偏移量（Raw Offset）	差值 Δk
.text	1000h	400h	0C00h
.rdata	2000h	600h	1A00h
.data	3000h	800h	2800h

例如，此实例中某一虚拟地址（VA）= 401112h，要求计算它的文件偏移地址。401112h 在.text 块中，此时Δk = 0C00h，故

$$\text{File Offset} = \text{VA} - \text{ImageBase} - \Delta k = 401112\text{h} - 400000\text{h} - \text{C00h} = 512\text{h}$$

再来看一看虚拟地址 4020D2h 的转换：
4020D2h 是在.rdata 块中，此时Δk=1800h，故

$$\text{File Offset} = \text{VA} - \text{ImageBase} - \Delta k = 4020\text{D2h} - 400000\text{h} - 1800\text{h} = 8\text{D2h}$$

在实际操作时，建议使用 RVA-Offset 之类的转换工具。LordPE 工具也有这个转换功能，单击图 10.7

中的“FLC”按钮打开“文件位址计算器（File Location Calculator）”，见图 10.12。

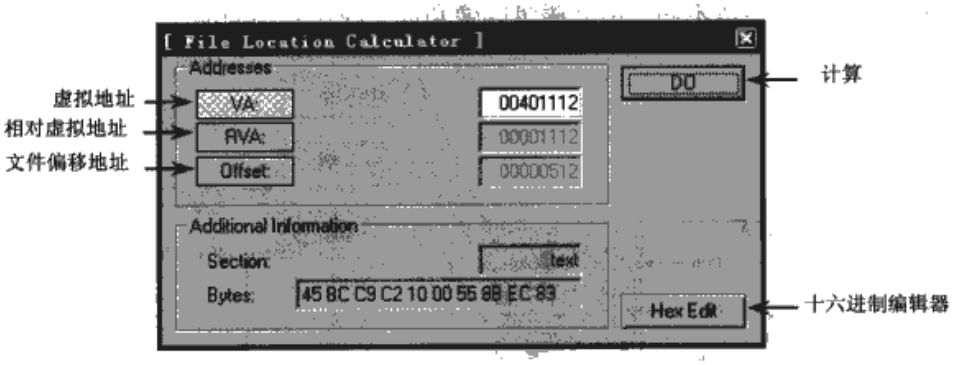


图 10.12 地址转换器

在 VA 域中输入要转换的虚拟地址 401112h，单击“DO”按钮，转换后的文件偏移地址为 512h。

10.5 输入表

可执行文件使用来自于其他 DLL 的代码或数据时，称为输入。当 PE 文件装入时，Windows 加载器的工作之一就是定位所有被输入的函数和数据，并且让正在被装入的文件可以使用那些地址。这个过程是通过 PE 文件的输入表（Import Table，简称 IT，也称为导入表）来完成的，输入表中保存的是函数名和其驻留的 DLL 名等动态链接所需的信息。输入表在软件外壳技术上的地位非常重要，因此读者在研究外壳相关技术时，一定要彻底掌握这部分知识。

10.5.1 输入函数的调用

在代码分析或编程中经常遇到“输入函数（Import functions）”。输入函数就是被程序调用但其执行代码又不在程序中的函数，这些函数的代码位于相关的 DLL 文件中，在调用者程序中只保留相关函数信息，如函数名、DLL 文件名等。对于磁盘上的 PE 文件来说，它无法得知这些输入函数在内存中的地址。只有当 PE 文件被装入内存后，Windows 加载器才将相关 DLL 装入，并将调用输入函数的指令和函数实际所处的地址联系起来。

当应用程序调用一个 DLL 的代码和数据时，那它正在隐含链接到 DLL，这个过程完全由 Windows 加载器完成。另一种是运行期的显式链接，这意味着必须确定目标 DLL 已经被加载，然后寻找 API 的地址，这几乎总是通过调用 LoadLibrary 和 GetProcAddress 来完成的。

当隐含地链接一个 API 时，类似 LoadLibrary 和 GetProcAddress 的代码始终在执行，只不过这是 Windows 装载器自动完成的。装载器还保证 PE 文件所需的任何附加的 DLL 都已被载入。例如，Windows 2000/XP 上每个由 Visual C++创建的正常程序都要链接 KERNEL32.DLL，而它又从 NTDLL.DLL 输入函数。同样地，如果链接了 GDI32.DLL，它又依赖于 USER32、ADVAPI32、NTDLL 和 KERNEL32 等 DLL 的函数，这些都是由加载器来保证装入并解决输入问题。

在 PE 文件内，有一组数据结构，它们分别对应着每个被输入的 DLL。每一个这样的结构都给出了被输入的 DLL 的名称并指向一组函数指针。这组函数指针被称为输入地址表（Import Address Table，简称 IAT）。每一个被引入的 API 在 IAT 里都有它自己保留的位置，在那里它将被 Windows 加载器写入输入函数的地址。最后一点是特别重要的：一旦模块被装入，IAT 中包含所要调用输入函数的地址。

把所有输入函数放在 IAT 中同一个地方是很有意义的，这样无论代码中多少次调用一个输入函数，都会通过 IAT 中的同一个函数指针来完成。

现在看看怎样调用一个输入函数。需要考虑两种情况：高效和低效。最好的情况是像下面这样：

```
CALL DWORD PTR [00402010]
```

直接调用[00402010]中的函数，地址 402010h 位于 IAT 里。而实际上对一个被输入的 API 低效的调用像下面这样（下面是实例 PE.exe 中的调用 LoadIconA 函数的代码）：

```
    Call 00401164
    ...
:00401164
    Jmp dword ptr [00402010]    ; 指向 USER32.LoadIconA 函数
```

这种情况，CALL 把控制权转到一个子程序，子程序中的 JMP 指令跳转到位于 IAT 中的 402010h。简单地说，它使用 5 个字节的额外代码，并且由于额外的 JMP 将花费更多的时间去执行。

读者可能会问，为什么还采用此种低效方法？有个很好的解释，编译器无法区别输入函数的调用和普通函数调用。对于每个函数调用，编译器使用同样形式的 CALL 指令：

```
CALL XXXXXXXX
```

XXXXXXXX 是一个由链接器填充的实际的地址。注意，这条指令不是从函数指针而是代码中的实际地址而来的。为了因果的平衡，链接器必须产生一块代码来取代 XXXXXXXX，简单的方法就是像上面所示调用一个 JMP stub。

JMP 指令来自于为输入函数准备的输入库。如果读者曾经检查过某个输入库，在输入函数名字的关联处就会发现与上面 JMP stub 相似的指令。这意味着在默认情况下，对被输入 API 的调用将使用低效的形式。

如何得到优化的形式？答案来自于给编译器的一个提示形式。可以用修饰函数的__declspec（dllimport）来告诉编译器，这个函数来自另一个 DLL 中，这样编译器就会产生这样的指令：

```
CALL DWORD PTR [XXXXXXXX]
```

而不是：

```
CALL XXXXXXXX
```

此外，编译器将给函数加上__imp_前缀，然后送给链接器，这样可以直接把__imp_xxx 送到 IAT，就不需要 JMP stub 了。

如果在写一个输出函数并且为它们提供一个头文件的话，别忘了在函数前加上修饰符__declspec(dllimport)，在 winnt.h 等系统头文件中就是这样做的。

```
__declspec(dllimport) void Foo(void);
```

10.5.2 输入表结构

PE 文件头的可选映像头中数据目录表的第二成员指向输入表。输入表以一个 IMAGE_IMPORT_DESCRIPTOR（简称 IID）数组开始。每个被 PE 文件隐式地链接进来的 DLL 都有一个 IID。在这个数组中，没有字段指出该结构数组的项数，但它的最后一个单元是 NULL，可以由此计算出该数组的项数。例如，某个 PE 文件从两个 DLL 文件中引入函数，就存在两个 IID 结构来描述这些 DLL 文件，并在两个 IID 结构的最后由一个内容全为 0 的 IID 结构作为结束。

IID 的结构如下：

```
IMAGE_IMPORT_DESCRIPTOR STRUC
    union                           ; 00h
        Characteristics     DWORD ?
        OriginalFirstThunk  DWORD ?
    ends
    TimeDateStamp           DWORD ? ; 04h
    ForwarderChain          DWORD ? ; 08h
    Name                    DWORD ? ; 0Ch
```

```
    FirstThunk              DWORD ? ; 10h
IMAGE_IMPORT_DESCRIPTOR ENDS
```

- OriginalFirstThunk（Characteristics）：包含指向输入名称表（简称 INT）的 RVA，INT 是一个 IMAGE_THUNK_DATA 结构的数组，数组中的每个 IMAGE_THUNK_DATA 结构指向 IMAGE_IMPORT_BY_NAME 结构，数组最后以一个内容为 0 的 IMAGE_THUNK_DATA 结构结束。
- TimeDateStamp：一个 32 位的时间标志，可以忽略。
- ForwarderChain：这是第一个被转向的 API 的索引，一般为 0。当程序引用一个 DLL 中的 API，而这个 API 又引用别的 DLL 的 API 时使用，但这样的情况很少出现。
- Name：DLL 名字的指针。是个以 00 结尾的 ASCII 字符的 RVA 地址，该字符串包含输入的 DLL 名，例如“KERNEL32.DLL”或“USER32.DLL”。
- FirstThunk：包含指向输入地址表（IAT）的 RVA。IAT 是一个 IMAGE_THUNK_DATA 结构的数组。

OriginalFirstThunk 与 FirstThunk 非常相似，它们指向两个本质上相同的数组 IMAGE_THUNK_DATA，这些数组有好几种叫法，但最常见的名字是输入名称表（Import Name Table，INT）和输入地址表（Import Address Table，IAT）。图 10.13 表示了一个可执行文件正在从 USER32.DLL 里输入一些 API。

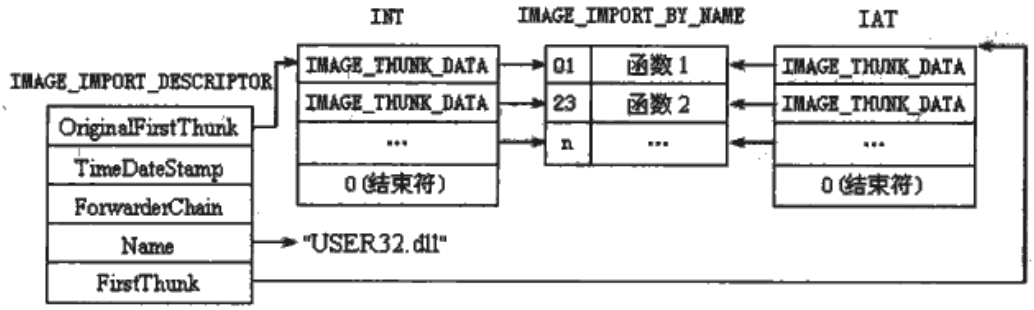


图 10.13　两个并行的指针数组

两个数组都有 IMAGE_THUNK_DATA 结构类型的元素，它是一个指针大小的联合（union）。每一个 IMAGE_THUNK_DATA 元素对应于一个从可执行文件输入的函数。两个数组的结束是通过一个值为零的 IMAGE_THUNK_DATA 元素来表示的。IMAGE_THUNK_DATA 结构实际上是一个双字，该结构在不同时刻有不同的含义。定义如下：

```
IMAGE_THUNK_DATA STRUC
    union u1
        ForwarderString     DWORD ?     ; 指向一个转向者字符串的 RVA
        Function            DWORD ?     ; 被输入的函数的内存地址
        Ordinal             DWORD ?     ; 被输入的 API 的序数值
        AddressOfData       DWORD ?     ; 指向 IMAGE_IMPORT_BY_NAME
    ends
IMAGE_THUNK_DATA ENDS
```

当 IMAGE_THUNK_DATA 值的最高位为 1 时，表示函数以序号方式输入，这时低 31 位（或者一个 64 位可执行文件的低 63 位）被看作是一个函数序号。当双字的最高位为 0 时，表示函数以字符串类型的函数名方式输入，这时双字的值是一个 RVA，指向一个 IMAGE_IMPORT_BY_NAME 结构。

IMAGE_IMPORT_BY_NAME 结构仅仅是一个字大小，存有一个输入函数的相关信息结构。其结构如下：

```
IMAGE_IMPORT_BY_NAME STRUCT
    Hint                WORD ?
    Name                BYTE ?
IMAGE_IMPORT_BY_NAME ENDS
```

- Hint：指示本函数在其所驻留 DLL 的输出表中的序号。该域被 PE 装载器用来在 DLL 的输出表里快速查询函数。该值不是必需的，一些链接器将此值设为 0。
- Name：含有输入函数的函数名，函数名是一个 ASCII 码字符串，以 NULL 结尾。注意，这里虽然将 Name 的大小定义成字节，其实它是可变尺寸域，由于没有更好的表示方法，只好在上面定义中写成 BYTE。

10.5.3 输入地址表（IAT）

为什么由两个并行的指针数组指向 IMAGE_IMPORT_BY_NAME 结构呢？第一个数组（由 OriginalFirstThunk 所指向）是单独的一项，而且不可改写，称为 INT，有时也称为提示名表（Hint-name Table）。第二个数组（由 FirstThunk 所指向）是由 PE 装载器重写的。PE 装载器首先搜索 OriginalFirstThunk，如果找到，加载程序迭代搜索数组中的每个指针，找到每个 IMAGE_IMPORT_BY_NAME 结构所指向的输入函数的地址，然后加载器用函数真正入口地址来替代由 FirstThunk 指向的 IMAGE_THUNK_DATA 数组里的元素值。Jmp dword ptr [xxxxxxxx]中的[xxxxxxxx]是指 First Thunk 数组中的一个入口，因此它称为输入地址表（Import Address Table，IAT）。因此，当 PE 文件装载内存后准备执行时，图 10.13 已转换成图 10.14 所示的状态，所有函数入口地址被排列在一起。此时，输入表中其他部分就不重要了，程序依靠 IAT 提供的函数地址就可正常运行。

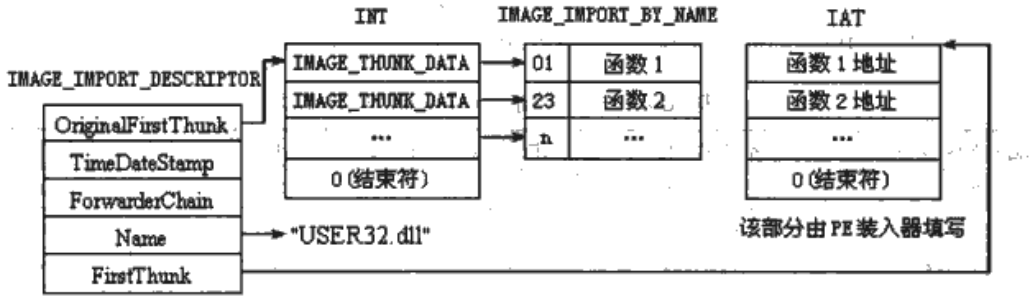


图 10.14 PE 文件加载后的 IAT 表

有些情况下，一些函数仅由序号引出，也就是说，不能用函数名来调用它们，只能用它们的位置来调用。此时，IMAGE_THUNK_DATA 值的低位字指示函数序数，而最高二进位（MSB）设为 1。Microsoft 提供了一个方便的常量来测试 DWORD 值的 MSB 位，就是 IMAGE_ORDINAL_FLAG32，其值为 80000000h（PE32+中是 IMAGE_ORDINAL_FLAG64，其值为 8000000000000000h）。

另外一种情况是程序 OrignalFirstThunk 的值为 0。在初始化时，系统根据 FirstThunk 的值找到指向函数名的地址串，由地址串找到函数名，再根据函数名得到入口地址，然后用入口地址取代 FirstThunk 指向的地址串中的原值。

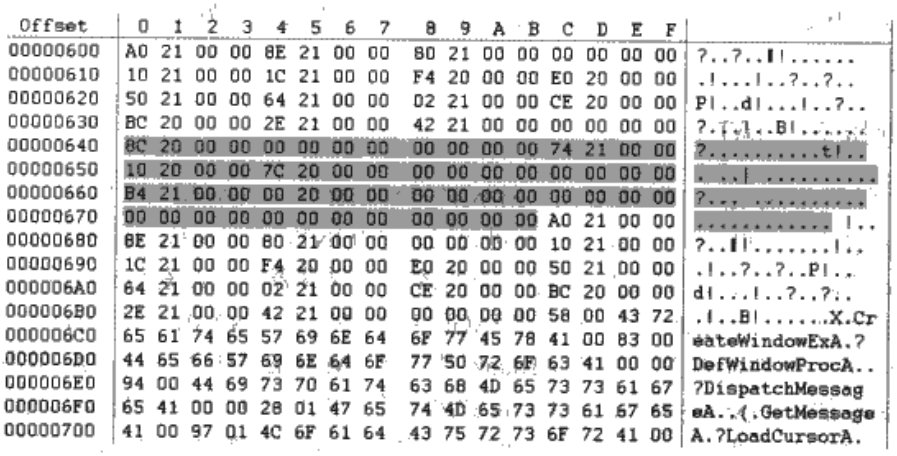
10.5.4 输入表实例分析

下面就来分析实例 PE.exe 文件的输入表。数据目录表的第二成员指向输入表，该指针具体位置是在 PE 文件头的 80h 偏移处。该文件的 PE 文件头起始位置是 B0h，输入表地址就是在整个文件的 B0h+80h=130h 处，因此在 130h 处可以发现四字节指针 40 20 00 00，倒过来就是 00002040，即输入表在内存中偏移量为 2040h 的地方。当然，这个 2040h 是 RVA 值，需要将其转换为磁盘文件的绝对偏移量，才能够在十六进制编辑器中找到输入表。

可以用 LordPE 之类的 PE 编辑工具来查看各个块的实际偏移，以便确定 2040h 到底指的是什么地方。为了加强理解，在此手动转换，从图 10.10 可知，2040h 位于.rdata 块中，.rdata 块的虚拟偏移是 2000h，其物理偏移是 600h，因此Δk = 2000h−600h = 1A00h，这个数字在后面的两种偏移量转换中需要用到，现在先

记住它。相对虚拟地址（RVA）2040h 转换成文件偏移地址：2040h－1A00h＝640h。

用十六进制工具打开文件，跳到偏移 640h 处，这里就是输入表的内容，每个 IID 包含 5 个双字，用来描述一个引入的 DLL 文件，最后以 NULL 结束，图 10.15 列出了输入表的一部分。



```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000600 A0 21 00 00 8E 21 00 00  80 21 00 00 00 00 00 00  ?..?..!.......
00000610 10 21 00 00 1C 21 00 00  F4 20 00 00 E0 20 00 00  .!...!..?..?..
00000620 50 21 00 00 64 21 00 00  02 21 00 00 CE 20 00 00  P!..d!...!..?..
00000630 BC 20 00 00 2E 21 00 00  42 21 00 00 00 00 00 00  ?.?.!..B!......
00000640 8C 20 00 00 00 00 00 00  00 00 00 00 74 21 00 00  ?.........t!..
00000650 10 20 00 00 7C 20 00 00  00 00 00 00 00 00 00 00  . ..| ..........
00000660 B4 21 00 00 00 20 00 00  00 00 00 00 00 00 00 00  ?... ..........
00000670 00 00 00 00 00 00 00 00  00 00 00 00 A0 21 00 00  ............ !..
00000680 8E 21 00 00 80 21 00 00  00 00 00 00 10 21 00 00  ?..!.....!..!..
00000690 1C 21 00 00 F4 20 00 00  E0 20 00 00 50 21 00 00  .!..?..?..P!..
000006A0 64 21 00 00 02 21 00 00  CE 20 00 00 BC 20 00 00  d!...!..?..?..
000006B0 2E 21 00 00 42 21 00 00  00 00 00 00 58 00 43 72  .!..B!......X.Cr
000006C0 65 61 74 65 57 69 6E 64  6F 77 45 78 41 00 83 00  eateWindowExA.?
000006D0 44 65 66 57 69 6E 64 6F  77 50 72 6F 63 41 00 00  DefWindowProcA..
000006E0 94 00 44 69 73 70 61 74  63 68 4D 65 73 73 61 67  ?DispatchMessag
000006F0 65 41 00 00 28 01 47 65  74 4D 65 73 73 61 67 65  eA..(.GetMessage
00000700 41 00 97 01 4C 6F 61 64  43 75 72 73 6F 72 41 00  A.?LoadCursorA.
```

图 10.15　磁盘文件中的输入表

将图 10.15 所列的输入表的 IID 数组（图中阴影部分）整理到表 10-8 中。每个 IID 包含了一个 DLL 的描述信息，现在有两个 IID，因此这里引入了两个 DLL，第三个 IID 全为 0，作为结束标志。

表 10-8　十六进制工具中显示的 IID 数组

OrignalFirstThunk	TimeDateStamp	ForwardChain	Name	First Thunk
8C20 0000	0000 0000	0000 0000	7421 0000	1020 0000
7C20 0000	0000 0000	0000 0000	B421 0000	0020 0000
0000 0000	0000 0000	0000 0000	0000 0000	0000 0000

每个 IID 中的第四个字段是指向 DLL 名称的指针。这里第一个 IID 中的第四个字段是 7421 0000，翻转过来也就是 RVA 地址 00002174h，将它减去 1A00h 得到文件偏移地址 774h，于是，查看 EXE 文件中偏移量为 774h 处的字符，是什么？原来调用的是 USER32.dll。经转换后的 IID 数组见表 10-9，表内各指针都是 RVA 地址。

表 10-9　IID 数组

DLL 名称	OrignalFirstThunk	TimeDateStamp	ForwardChain	Name	First Thunk
USER32.dll	0000 208C	0000 0000	0000 0000	0000 2174	0000 2010
KERNEL32.dll	0000 207C	0000 0000	0000 0000	0000 21B4	0000 2000

再找 USER32.dll 中被调用的函数。仍然在第一个 IID 中，查看第一个字段 OrignalFirstThunk，它指向一个数组，这个数组的元素都是指针，分别指向引入函数名的 ASCII 字符串。有些程序的 OriginalFirstThunk 值为 0，所以这时就要看 FirstThunk，它在程序运行时被初始化。

USER32.dll 所在 IID 的 OrignalFirstThunk 字段值是 208Ch，减去 1A00h 得 68Ch，在偏移 68Ch 处就是 IMAGE_THUNK_DATA 数组，它存储的是指向 IMAGE_IMPORT_BY_NAME 结构的地址，以一串 00 结束，如表 10-10 所示。

表 10-10　IMAGE_THUNK_DATA 数据

1021 0000	1C21 0000	F420 0000	E020 0000
5021 0000	6421 0000	0221 0000	CE20 0000
BC20 0000	2E21 0000	4221 0000	0000 0000

再来看看同一 IID 结构中 FirstThunk 情况，USER32.dll 所在 IID 的 FirstThunk 字段值是 2010h，减去 1A00h 得 610h，在偏移 610h 处就是 IMAGE_THUNK_DATA 数组，其数据与 OrignalFirstThunk 字段所指的完全一样（见表 10-10）。

通常，在一个完整的程序里都有这些。现在有 11 个 IMAGE_THUNK_DATA，表示有 11 个函数调用，先看看其中的两个。

1021 0000 翻转后是 2110h，减去 1A00h 后等于 710h，会发现在偏移 710h 处的字符串是 LoadIconA。

1C21 0000 翻转后是 211Ch，减去 1A00h 后等于 71Ch，会发现在偏移 71Ch 处的字符串是 PostQuitMessage。

读者也许注意到了，计算出来的偏移量并不刚好指向函数名的 ASCII 字符串，而是前面还有两个字节的空缺，这是作为函数名（Hint）引用的，可以为 0。

表 10-11 所示是第一个 IID 指向的各种 API 函数。

表 10-11 第一个 IID 指向的 API 函数

提示名表（RVA）	提示名表（File Offset）	Hint	ApiName
0000 2110	710	019B	LoadIconA
0000 211C	71C	01DD	PostQuitMessage
0000 20F4	6F4	0128	GetMessageA
0000 20E0	6E0	0094	DispatchMessageA
0000 2150	750	072D	TranslateMessage
0000 2164	764	028B	UpdateWindow
0000 2102	702	0197	LoadCursorA
0000 20CE	6CE	0083	DefWindowProcA
0000 20BC	6BC	0058	CreateWindowExA
0000 212E	72E	01EF	RegisterClassExA
0000 2142	742	0265	ShowWindow

图 10.16 是 PE.exe 文件运行前第一个 IID 的结构示意图。在程序运行前，它的 FirstThunk 字段值也是指向一个地址串，而且和 OrignalFirstThunk 字段值指向的 INT 是重复的。系统在程序初始化时根据 OrignalFirstThunk 的值找到函数名，调用 GetProcAddress 函数（或类似功能的系统代码）且根据函数名取得函数的入口地址，然后用函数入口地址取代 FirstThunk 指向的地址串中对应的值（IAT）。

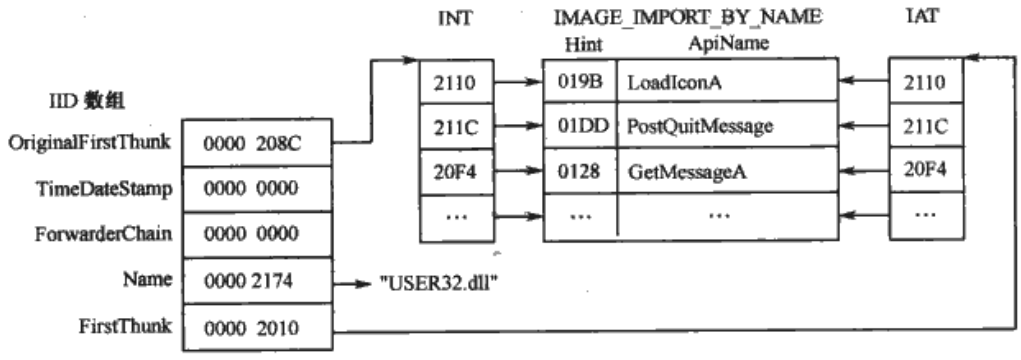


图 10.16 第一个 IID 在磁盘文件里的结构

实例 dumped.exe 是从内存中抓取出来的，因此其结构就是 PE 文件映射到内存的状态。打开映像文件，由于在内存中区块的对齐值与内存页相同，因此此时其文件偏移地址与相对虚拟地址（RVA）的值相等。输入表的 RVA 地址是 2040h，具体见图 10.17。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00002000  D9 AC E5 77 AB E2 E5 77  63 9B E5 77 00 00 00 00
00002010  DD 16 D2 77 77 F2 D1 77  18 91 D1 77 05 91 D1 77
00002020  BC 8A D1 77 05 AB D1 77  07 CA D1 77 EC AB D1 77
00002030  F4 19 D1 77 72 EC D1 77  DF C1 D1 77 00 00 00 00
00002040  8C 20 00 00 00 00 00 00  00 00 00 00 74 21 00 00
00002050  10 20 00 00 7C 20 00 00  00 00 00 00 00 00 00 00
00002060  B4 21 00 00 00 20 00 00  00 00 00 00 00 00 00 00
00002070  00 00 00 00 00 00 00 00  00 00 00 00 A0 21 00 00
00002080  8E 21 00 00 80 21 00 00  00 00 00 00 10 21 00 00
00002090  1C 21 00 00 F4 20 00 00  E0 20 00 00 50 21 00 00
000020A0  64 21 00 00 02 21 00 00  CE 20 00 00 BC 20 00 00
000020B0  2E 21 00 00 42 21 00 00  00 00 00 00 58 00 43 72
000020C0  65 61 74 65 57 69 6E 64  6F 77 45 78 41 00 83 00
```

图 10.17　内存中的部分输入表

从图 10.17 中看出，OrignalFirstThunk 字段指向的数据没变，但 FirstThunk 字段指向的数据已改变（图中阴影部分为 IAT）。第一个 IID 的 FirstThunk 字段为 2010h，该 RVA（2010h）指向输入地址表（IAT），将这张表的数据整理进表 10-12（笔者当时的系统是 Windows XP SP1，不同系统其值不同）。

表 10-12　内存中第一个 IID 结构的输入地址表（IAT）

77D2 16DD	77D1 F277	77D1 9118	77D1 9105	77D1 8ABC	77D1 AB05
77D1 CA07	77D1 ABE0	77D1 19F4	77D1 EC72	77D1 C1DF	0000 0000

表 10-12 中各地址都是 USER32.dll 链接库的相关输出函数，先反汇编 USER32.dll（反汇编技术参考静态分析一章），跳到 77D216DDh 地址处，显示代码如下：

```
Exported fn(): LoadIconA - Ord:01BCh
:77D216DD 8BC0                  mov eax, eax
:77D216DF 55                    push ebp
:77D216E0 8BEC                  mov ebp, esp
:77D216E2 66F7450EFFFF          test [ebp+0E], FFFF
:77D216E8 0F8529170200          jne 77D42E17
:77D216EE 5D                    pop ebp
:77D216EF EBB6                  jmp 77D216A7
:77D216F1 90                    nop
:77D216F2 90                    nop
```

原来，77D216DD 指向的是 USER32.dll 中 LoadIconA 函数代码处。图 10.18 是 PE.exe 文件装载到内存里的第一个 IID 的结构示意图。

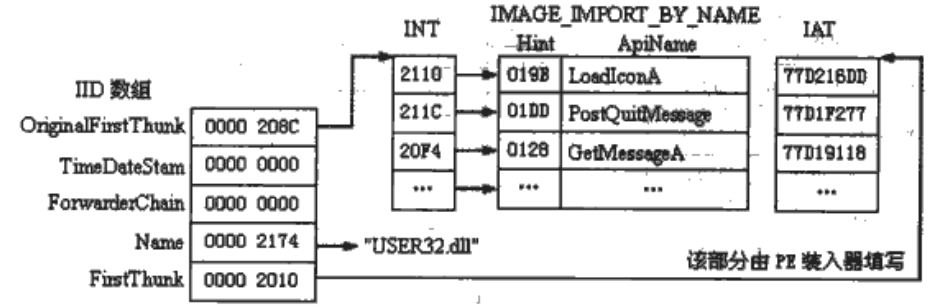


图 10.18　第一个 IID 装载到内存里的结构

程序装载进内存中后，只与 IAT 交换信息，输入表的其他部分不需要了。例如，程序调用 LoadIconA 函数的指针是指向 IAT 的，而 IAT 已指向系统 USER32.dll 的 LoadIconA 函数代码里。调用 LoadIconA 函数的相关代码如下：

```
    Call 00401164
00401164
    Jmp dword ptr [00402010] ; 跳到77D216DD，此处是USER32.dll指向的LoadIconA
```

这个程序有两个DLL，读者可参考上面的过程分析第二个IID。

10.4 绑定输入

当PE装载器装入PE文件时，检查输入表并将相关DLL映射到进程地址空间。然后遍历IAT里的IMAGE_THUNK_DATA数组并用输入函数的真实地址替换它，这一步需要很多时间。如果程序员事先能正确预测函数地址，PE装载器就不用每次装入PE文件时都去修正IMAGE_THUNK_DATA值了，绑定输入（Bound Import）就是这种思想的产物。

当一个可执行文件被绑定（例如通过绑定程序Visual Studio的Bind.exe）时，IAT中的IMAGE_THUNK_DATA结构被输入函数的实际地址改写了。磁盘中的可执行文件，它们的IAT里有的存放的是相关DLL输出函数的实际内存地址。这样可以使应用程序更快地进行初始化，并且使用较少的存储器。

在执行整个进程期间，Bind程序做了两个重要假设：

- 当进程初始化时，需要的DLL实际上加载到了它们的首选基地址中。
- 自从绑定操作执行以来，DLL输出表中引用的符号位置一直没有改变。

当然，如果上面的两个假设中有一个是假的，IAT中所有地址均是无效的，加载器会检查这种情况并做出相应反应，加载器从INT表里获得所需要的信息来解决输入API的地址问题。对于一个可执行文件的装入，INT是不需要的。但是，如果没有，可执行文件是不能被绑定的。微软的链接器似乎总是生成一个INT，但是在很长一段时间里，Borland的链接器（TLINK）没有这样做，由Borland生成的文件是不能被绑定的。

由于不知用户运行的是Windows 2000还是Windows XP，无法将系统DLL绑定起来，因此程序安装时是绑定程序的最佳时机。Windows安装器的BindImage将做这些工作。另外，IMAGEHLP.DLL提供了BindImageEx函数。不管用什么方式，绑定都是一个好主意。如果加载器确定绑定信息是当前的，可执行文件的装入会更快，如果绑定信息已经变得陈旧了，也不会影响程序的运行。

对于加载器来说，使绑定变得有效的一个关键步骤是确定在IAT表中的绑定信息是否是当前的。当一个可执行文件被绑定时，被参考的DLL信息放入了文件中，加载器检查这些信息来做一个快速的绑定有效性验证。

数据目录表(DataDirectory)的第11个成员指向绑定输入。绑定输入以一个IMAGE_BOUND_IMPORT_DESCRIPTOR结构的数组开始，一个绑定可执行文件包含一系列这样的结构，每个IBID结构都指出了一个已经被绑定输入DLL的时间/日期戳。IBID的结构如下：

```
IMAGE_BOUND_IMPORT_DESCRIPTOR STRUC
  TimeDateStamp                 DWORD   ?
  OffsetModuleName              WORD    ?
  NumberOfModuleForwarderRefs   WORD    ? ;
IMAGE_BOUND_IMPORT_DESCRIPTOR ENDS
```

- TimeDateStamp：一个双字，包含一个被输入DLL的时间/日期戳；允许加载器快速判断绑定是不是新的。

- OffsetModuleName：一个字，包含一个指向被输入 DLL 的名称的偏移。这个字段是与第一个 IBID 结构之间的偏移（不是 RVA）。
- NumberOfModuleForwarderRefs：一个字，它包含紧跟在这个结构后面的 IMAGE_BOUND_FORWARDER_REF 结构的数目。这些结构是和 IBID 相同的，除了最后的一个字（NumberOfModuleForwarderRef）被保留。

当绑定一个 API 被转向到另一个 DLL 时，被转向到的 DLL 的有效性也要被检查。这样，IMAGE_BOUND_FORWARDER_REF 和 IMAGE_BOUND_IMPORT_DESCRIPTOR 结构是交叉存取的。

例如链接到 HeapAlloc，它被转向到 NTDLL 中的 RtlAllocateHeap，然后对可执行文件运行 BIND。在 EXE 里，已经有一个针对 KERNEL32.DLL 的 IBID，它后面跟着一个针对 NTDLL.DLL 的 IMAGE_BOUND_FORWARDER_REF。紧跟在后面可能是另外的你输入并绑定的针对其他 DLL 的 IBID。

Windows 目录里的应用程序就是典型的绑定输入结构程序，其 IAT 已指向相关 DLL 的函数。图 10.19 是 Windows XP 记事本程序（Notepad.exe）的绑定输入结构，此时记事本程序的 IAT 全是指向了系统 DLL 相关函数的入口地址。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000250	00	00	00	00	40	00	00	40	A0	16	90	3B	58	00	00	00	....@..@ .?;X...c
00000260	96	16	90	3B	65	00	00	00	B9	16	90	3B	71	00	00	00	??e...??q...X...c
00000270	32	FE	7D	3B	7E	00	00	00	96	16	90	3B	8B	00	00	00	2?;~...???.....c
00000280	95	16	90	3B	96	00	00	00	94	16	90	3B	A3	00	01	00	???..???..?.....c
00000290	1E	E0	7D	3B	B0	00	00	00	96	16	90	3B	BA	00	00	00	.?;?..???.....c
000002A0	95	16	90	3B	C4	00	00	00	00	00	00	00	00	00	00	00	???.............c
000002B0	63	6F	6D	64	6C	67	33	32	2E	64	6C	6C	00	53	48	45	comdlg32.dll.SHE
000002C0	4C	4C	33	32	2E	64	6C	6C	00	57	49	4E	53	50	4F	4F	LL32.dll.WINSPOO
000002D0	4C	2E	44	52	56	00	43	4F	4D	43	54	4C	33	32	2E	64	L.DRV.COMCTL32.d
000002E0	6C	6C	00	6D	73	76	63	72	74	2E	64	6C	6C	00	41	44	ll.msvcrt.dll.AD
000002F0	56	41	50	49	33	32	2E	64	6C	6C	00	4B	45	52	4E	45	VAPI32.dll.KERNE
00000300	4C	33	32	2E	64	6C	6C	00	4E	54	44	4C	4C	2E	44	4C	L32.dll.NTDLL.DL
00000310	4C	00	47	44	49	33	32	2E	64	6C	6C	00	55	53	45	52	L.GDI32.dll.USER
00000320	33	32	2E	64	6C	6C	00	00	00	00	00	00	00	00	00	00	32.dll..........c

图 10.19　绑定输入的结构

为了方便实现，Microsoft 的一些编译器（如 Visual Studio）都提供了 bind.exe 这样的工具，由它检查 PE 文件的输入表，并用输入函数的真实地址替换 IAT 里的 IMAGE_THUNK_DATA 值。当文件装入时，PE 装载器必定检查地址的有效性。如果 DLL 版本不同于 PE 文件存放的相关信息，或 DLL 需要重定位，那么装载器认为原先计算的地址是无效的，它必定遍历 OriginalFirstThunk 指向的数组以获取输入函数新地址，产生一个新的 IAT。

绑定输入表去除是不会影响程序正常运行的，去除方法是将图 10.19 中的绑定数据清零，然后再将目录表中的 Bound import 的 RVA 与大小清零即可。

10.5 输出表

当创建一个 DLL 时，实际上创建了一组能让 EXE 或其他 DLL 调用的一组函数，此时 PE 装载器根据 DLL 文件中输出信息修正被执行文件的 IAT。当一个 DLL 函数能被 EXE 或另一个 DLL 文件使用时，它被称为输出了（exported）。其中输出信息被保存在输出表中，DLL 文件通过输出表向系统提供输出函数名、序号和入口地址等信息。

EXE 文件一般不存在输出表，而大部分 DLL 文件中存在输出表。当然，这也不是绝对的，有些 EXE 文件也会存在输出函数。

10.7.1 输出表结构

输出表（Export Table）中的主要成分是一个表格，内含函数名称、输出序数等。序数是指定DLL中某个函数的16位数字，在所指向的DLL里是独一无二的。在此不提倡仅仅通过序数引出函数这种方法，这会带来DLL维护上的问题。一旦DLL升级或修改，调用该DLL的程序将无法工作。

输出表是数据目录表的第一个成员，指向IMAGE_EXPORT_DIRECTORY（简称IED）结构。IED结构定义如下：

```
IMAGE_EXPORT_DIRECTORY STRUC
    Characteristics         DWORD   ?   ; 未使用，总是为0
    TimeDateStamp           DWORD   ?   ; 文件生成时间
    MajorVersion            WORD    ?   ; 主版本号，一般为0
    MinorVersion            WORD    ?   ; 次版本号，一般为0
    Name                    DWORD   ?   ; 模块的真实名称
    Base                    DWORD   ?   ; 基数，加上序数就是函数地址数组的索引值
    NumberOfFunctions       DWORD   ?   ; AddressOfFunctions阵列中的元素个数
    NumberOfNames           DWORD   ?   ; AddressOfNames阵列中的元素个数
    AddressOfFunctions      DWORD   ?   ; 指向函数地址数组
    AddressOfNames          DWORD   ?   ; 函数名字的指针地址
    AddressOfNameOrdinals   DWORD   ?   ; 指向输出序列号数组
IMAGE_EXPORT_DIRECTORY ENDS
```

说明：

- Characteristics：表示输出属性的旗标。目前还没有定义，总是为0。
- TimeDateStamp：输出表创建的时间（GMT时间）。
- MajorVersion：输出表的主版本号。未使用，设置为0。
- MinorVersion：输出表的次版本号。未使用，设置为0。
- Name：指向一个ASCII字符串的RVA，这个字符串是与这些输出函数关联的DLL的名字（例如KERNEL32.DLL）。
- Base：这个字段包含用于这个可执行文件输出表的起始序数值（基数）。正常地，这个值是1，但是并不需要非得这样。当通过序数来查询一个输出函数时，这个值从序数里被减去，结果被用作进入输出地址表（EAT）的索引。
- NumberOfFunctions：EAT中的条目数量。注意，一些条目可能是0，表明用这个序数值没有代码或数据被输出。
- NumberOfNames：输出函数名称表（ENT）里的条目数量。这个值总是小于或等于NumberOfFunctions域值。小于的情况发生在符号只通过序数来输出，另外当被赋值的序数里有数字间距时也会是小于的，这个值也是输出序数表的长度。
- AddressOfFunctions：EAT的RVA。EAT是一个RVA数组，数组中的每一个非零的RVA都对应于一个被输出的符号。
- AddressOfNames：ENT的RVA。ENT是一个指向ASCII字符串的RVA数组。每一个ASCII字符串对应于一个通过名字输出的符号。这个表是排序的，所以ASCII字符串也是按顺序排列的。这允许加载器在查询一个被输出的符号时用二进制查找方式，名称的排序是二进制的（像C++ RTL中strcmp函数提供的一样），而不是一个环境特定的字母顺序。
- AddressOfNameOrdinals：输出序数表的RVA。这个表是字的数组。这个表将ENT中的数组索引映射到相应的输出地址表条目。

输出表的设计是为了方便PE装载器工作。首先，模块必须保存所有输出函数的地址，供PE装载器查询。模块将这些信息保存在AddressOfFunctions域所指向的数组中，而数组元素数目存放在

NumberOfFunctions 域中。如果模块引出 40 个函数，则 AddressOfFunctions 指向的数组必定有 40 个元素，而 NumberOfFunctions 值为 40。如果有些函数是通过名字引出的，那么模块必定也在文件中保留了这些信息。这些名字的 RVA 存放在一个数组中，供 PE 装载器查询。该数组由 AddressOfNames 指向，NumberOfNames 包含名字数目。PE 装载器知道函数名，并想以此获取这些函数的地址。至今为止，已有两个模块：名字数组和地址数组，但两者之间还没有联系的纽带，还需要一些联系函数名及其地址的东西。PE 参考指出使用到地址数组的索引作为连接，因此 PE 装载器在名字数组中找到匹配名字的同时，它也获取指向地址表中对应元素的索引。这些索引保存在由 AddressOfNameOrdinals 域所指向的另一个数组（最后一个）中。由于该数组起到联系名字和地址的作用，所以其元素数目必定和名字数组相同。例如，每个名字有且仅有一个相关地址，反过来则不一定：每个地址可有好几个名字来对应。因此，给同一个地址取“别名”。为了起到连接作用，名字数组和索引数组必须并行成对使用，比如索引数组的第一个元素必定含有第一个名字的索引，依次类推。图 10.20 显示了 Export Table 的格式及其中的 3 个阵列。

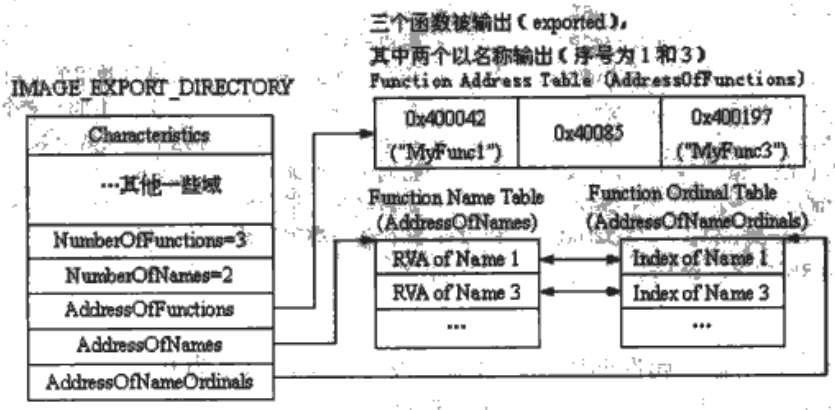


图 10.20　一个典型的输出表（Export Table）

10.7.2　输出表结构实例分析

在这里以光盘映像文件中提供的 DllDemo.DLL 这个实例了解输出表。数据目录表的第一个成员指向输出表，该指针具体位置是在 PE 文件头的 78h 偏移处。该文件的 PE 文件头起始位置是 100h，输出表就是在整个文件的 100h+78h=178h 处，因此在 178h 处可以发现四字节指针 00 40 00 00，倒过来就是 00004000，即输入表在内存中偏移 4000h 的地方。当然，这个 4000h 指的是内存中的偏移量，转成文件偏移地址就是 0C00h。文件偏移 0C00h 处是输出表内容，具体如图 10.21 所示。

```
Offset   0  1  2  3  4  5  6  7   8  9 10 11 12 13 14 15
00003072 00 00 00 00 00 00 00 00  00 00 00 00 32 40 00 00  ............2@..
00003088 01 00 00 00 01 00 00 00  01 00 00 00 28 40 00 00  ............(@..
00003104 2C 40 00 00 30 40 00 00  08 10 00 00 3E 40 00 00  ,@..0@......>@..
00003120 00 00 44 6C 6C 44 65 6D  6F 2E 44 4C 4C 00 4D 73  ..DllDemo.DLL.Ms
00003136 67 42 6F 78 00 00 00 00  00 00 00 00 00 00 00 00  gBox............
```

图 10.21　输出表

这个 DLL 只有一个输出函数 MsgBox，其 IMAGE_EXPORT_DIRECTORY 结构如表 10-13 所示。

表 10-13　IMAGE_EXPORT_DIRECTORY 结构

Characteristics	TimeDateStamp	MajorVersion	MinorVersion	Name	Base
0000 0000	0000 0000	0000	0000	3240 0000	0100 0000
NumberOfFunctions	NumberOfNames	AddressOfFunctions	AddressOfNames	AddressOfNameOrdinals	
0100 0000	0100 0000	2840 0000	2C40 0000	3040 0000	

（1）Name：4032h-3400h=C32h，指向 DLL 名字 DllDemo.DLL。

（2）AddressOfNames：402Ch-3400h=C2Ch，指向函数名的指针 403Eh-3400h=C3Eh，C3Eh 再指向函数名 MsgBox。

（3）AddressOfNameOrdinals：4030h-3400h=C30h，指向输出序号数组。

再来看看输出是如何实现的。PE 装载器调用 GetProcAddress 来查找 DllDemo.DLL 里的 API 函数 MsgBox，系统通过定位 DllDemo.DLL 的 IMAGE_EXPORT_DIRECTORY 结构开始工作，从这个结构中，它获得输出函数名称表（Export Names Table，简称 ENT）起始地址，进而知道这个数组里一共有 1 个条目，它对名字进行二进制查找直到发现字符串“MsgBox”。

PE 装载器发现 MsgBox 是数组的第一个条目，加载器然后从输出序数表读取相应的第一个值，这个值是 MsgBox 的输出序数。使用输出序数作为进入 EAT 的索引（并且也要考虑 Base 域值），它得到 MsgBox 的 RVA 是 1008h，1008h 加上 DllDemo.DLL 的装入地址得到 MsgBox 的实际地址。

10.8 基址重定位

当链接器生成一个 PE 文件时，它假设这个文件执行时会被装载到默认的基地址处，并且把 code 和 data 的相关地址都写入 PE 文件中。如果装入时按默认的值作为基地址装入，则不需要重定位。但如果可执行文件被装载到虚拟内存的另一个地址，链接器所登记的那个地址就是错误的，这时就需要用重定位表来调整。在 PE 文件中，它往往单独分为一块，用“.reloc”表示。

10.8.1 基址重定位概念

和 NE 格式的重定位方式不同，PE 的做法十分简单。它们并不参考外部 DLL 或模块中的其他 sections，而是把文件中所有可能需要修改的地址放在一个数组里。如果可执行文件不在首选的地址装入，那么文件中每一个定位都需要被修正。对加载器来说，它不需要知道关于地址如何使用的任何细节，它只需知道有一系列的数据需要以某种一致的方式来修正就可以了。

下面以实例 DllDemo.DLL 为例讲述其重定位过程。下面代码中两个加粗的地址指针是需要重定位的数据。

```
Exported fn(): MsgBox - Ord:0001h
:00401008 C8000000              enter 0000, 00
:0040100C 6A00                  push 00000000
* Possible StringData Ref from Data Obj ->"动态链接库"
:0040100E 6800204000            push 00402000
:00401013 FF7508                push [ebp+08]
:00401016 6A00                  push 00000000
:00401018 E804000000            Call 00401021
:0040101D C9                    leave
:0040101E C20400                ret 0004
* Reference To: USER32.MessageBoxA,  Ord:0000h
:00401021 FF2530304000          Jmp dword ptr [00403030]
```

来分析一下 0040100Eh 这句，其作用将一个指针压进栈，402000h 是某一字符串的指针。这句指令 5 字节长，前 1 个字节（68h）是指令的操作码，后 4 个字节用来保存一个 DWORD 大小的地址（00402000h）。在这个例子中，指令是来自一个基址为 00400000h 的 DLL 文件，因此这个字符串的 RVA 是 2000h。

如果可执行文件确实在 00400000h 处装入，那么指令能够按照现在的样子正确执行。但是当 DLL 执行时，Windows 加载器决定将其映射到 870000h 处（映射基址由系统决定）。加载器会比较基址和实际的

装入地址，计算出一个差值。在这个例子中，差值是 470000h。这个差值能被加到 DWORD 大小的地址值里以形成新地址。在前面的例子中，地址 0040100Fh 是指令中双字的定位，对它将有一个基址重定位，实际上字符串新的地址就是 872000h。为了让 Windows 有能力这样调整，可执行文件中内含许多个“基址重定位数据”。本例中的装载器应把 470000h 加给 402000h，并将 872000h 结果写回原处。图 10.22 演示了这个过程。

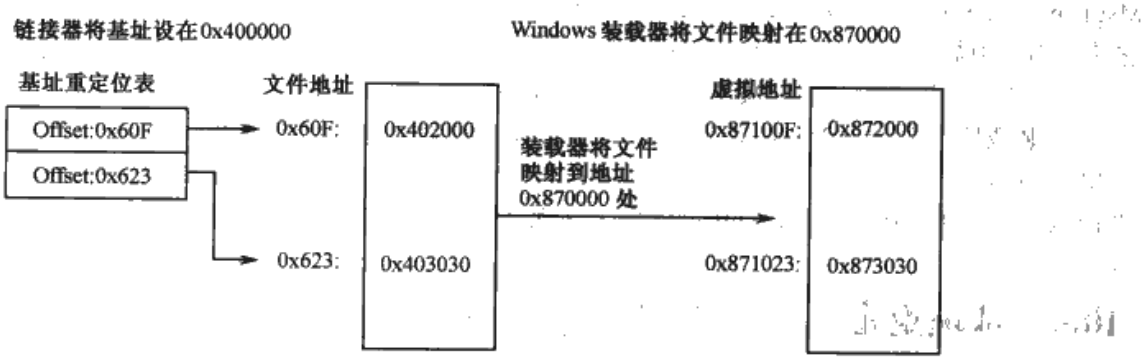


图 10.22　PE 文件重定位过程

DllDemo.DLL 在内存中重定位处理过的代码如下：

```
:00871008 C8000000              enter 0000, 00
:0087100C 6A00                  push 00000000
:0087100E 6800204000            push 00872000
:00871013 FF7508                push [ebp+08]
:00871016 6A00                  push 00000000
:00871018 E804000000            Call 00871021
:0087101D C9                    leave
:0087101E C20400                ret 0004
:00871021 FF2530304000          Jmp dword ptr [00873030]
```

对于 EXE 文件来说，每个文件总是使用独立的虚拟地址空间，所以 EXE 总是能够按照这个地址装入，这意味着 EXE 文件不再需要重定位信息。对于 DLL 来说，由于多个 DLL 文件全部使用宿主 EXE 文件的地址空间，不能保证装入地址没有被其他的 DLL 使用，所以 DLL 文件中必须包含重定位信息，除非用一个/FIXED 开关来省略它们。在 Visual Studio .NET 中，链接器会为 Debug 和 Release 模式的 EXE 文件省略掉基址重定位。因此在不同系统上跟踪同一个 DLL 文件时，其虚拟地址都是不相同的。也就是说，在读者的机器里运行 DllDemo.DLL，装载器映射的基址可能不是 00870000h，而是其他的值。

10.8.2　基址重定位结构定义

基址重定位表（Base Relocation Table）位于一个叫.reloc 的区块内，但是找到它们的正确方式是通过数据目录表的 IMAGE_DIRECTORY_ENTRY_BASERELOC 条目。基址重定位数据组织方法采用类似按页分割的方法，其由许多重定位块串接成的，每个块存放着 4KB（1000h）大小的重定位信息，每个重定位数据块的大小必须以 DWORD（4 字节）对齐。它们以一个 IMAGE_BASE_RELOCATION 结构开始，格式如下：

```
IMAGE_BASE_RELOCATION STRUC
  VirtualAddress     DWORD    ? ; 重定位数据开始 RVA 地址
  SizeOfBlock        DWORD    ? ; 重定位块的长度
TypeOffset           WORD     ? ; 重定项位数组
IMAGE_BASE_RELOCATION ENDS
```

- VirtualAddress：是这一组重定位数据的开始 RVA 地址。各重定位项的地址加上这个值才是该重定位项完整的 RVA 地址。
- SizeOfBlock：是当前重定位结构的大小。因为 VirtualAddress 和 SizeOfBlock 大小都是固定的 4 个字节，因此这个项减去 8，则是 TypeOffset 大小。
- TypeOffset：是一个数组。数组每项大小为两个字节，共 16 位。它又分为高 4 位与低 12 位，高 4 位代表重定位类型；低 12 位是重定位地址，它与 VirtualAddress 相加即是指向 PE 映像中需要修改的地址数据的指针。

常见的重定位类型见表 10-14。虽然有多种重定位类型，对于 x86 可执行文件，所有的基址重定位类型都是 IMAGE_REL_BASED_HIGHLOW。在一组重定位结束的地方会出现一个类型是 IMAGE_REL_BASED_ABSOLUTE 的重定位，这些重定位什么都不做，在那里只用于填充，以便下一个 IMAGE_BASE_RELOCATION 是以 4 字节分界线来对齐。所有重定位块最终以一个 VirtualAddress 字段为 0 的 IMAGE_BASE_RELOCATION 结构作为结束。

表 10-14　常见的重定位类型

类　型	在 winnt.h 里的预定义值	含　义
0	IMAGE_REL_BASED_ABSOLUTE	没有具体含义，仅是为了让每个段 4 字节对齐
3	IMAGE_REL_BASED_HIGHLOW	重定位指向的整个地址都需要被修正，实际基本都是这情况
10	IMAGE_REL_BASED_DIR64	出现在 64 位 PE 文件中，对指向的整个地址修正

重定位表的结构如图 10.23 所示，由数个 IMAGE_BASE_RELOCATION 结构组成，每个结构由 VirtualAddress、SizeOfBlock 和 TypeOffset 三部分组成。

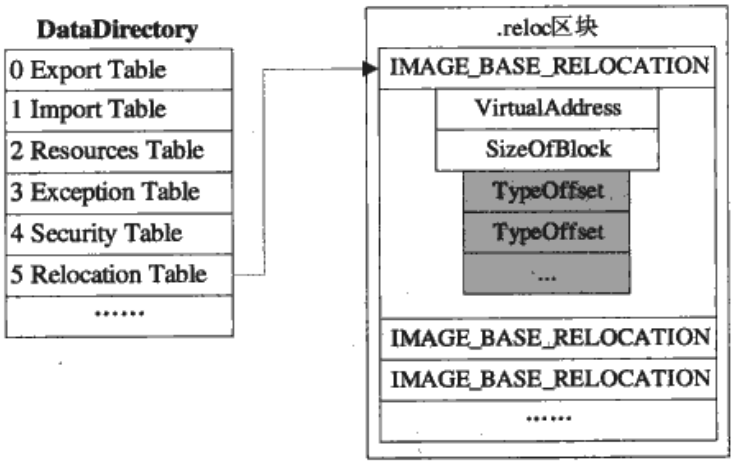


图 10.23　重定位表示意图

对于 IA-64 可执行文件，重定位似乎总是 IMAGE_REL_BASED_DIR64 类型。就像 x86 重定位，它们也用 IMAGE_REL_BASED_ABSOLUTE 重定位类型来进行填充。有趣的是，尽管 IA-64 的 EXE 页大小是 8KB，但基址重定位仍旧是 4KB 的块。

10.8.3　基址重定位结构实例分析

下面以 DllDemo.DLL 为例来讲解。数据目录表指向重定位表的指针是 5000h，换算成文件偏移地址就是 0E00h。其 IMAGE_BASE_RELOCATION 结构如图 10.24 所示。

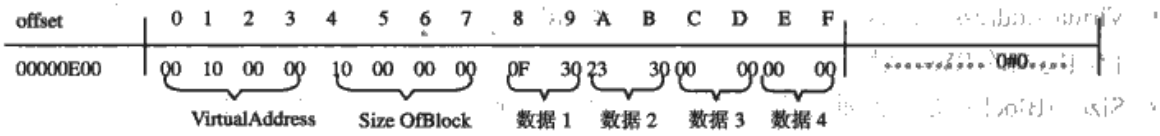


图 10.24　基址重定位表

VirtualAddress：00001000h

SizeOfBlock：00000010h（有 4 个重定位数据，(10h-8h)/2h=4h）

- 重定位数据 1：300Fh
- 重定位数据 2：3023h
- 重定位数据 3：0000（用于对齐）
- 重定位数据 4：0000（用于对齐）

重定位数据计算过程见表 10-15。

表 10-15　重定位数据转换

项　目	重定位数据 1	重定位数据 2	重定位数据 3	重定位数据 4
原始数据	0F 30	23 30	00 00	00 00
TypeOffset 值	300F	3023	—	—
TypeOffset 高 4 位（类型）	3	3	—	—
TypeOffset 低 12 位（地址）	00F	023	—	—
低 12 位加上 VirtualAddress	100F(RVA)	1023(RVA)	—	—
转换成文件偏移地址	60F	623	—	—

用十六进制工具查看实例文件，其中 060Fh 和 623h 分别指向 402000h 和 403030h，图 10.25 阴影部分即为所需要重定位的数据。

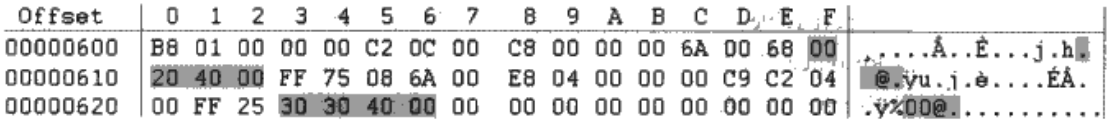


图 10.25　需要重定位的数据

执行 PE 文件前，加载程序在进行重定位的时候，会将 PE 文件在内存中的实际映像地址减去 PE 文件所要求的映像地址，得到一个差值，再将这一差值根据重定位类型的不同添加到地址数据中。

10.9 资源

Windows 程序的各种界面称为资源，包括加速键（Accelerator）、位图（Bitmap）、光标（Cursor）、对话框（Dialog Box）、图标（Icon）、菜单（Menu）、串表（String Table）、工具栏（Toolbar）、版本信息（Version Information）等。在 PE 文件所有结构中，资源部分是最复杂的。

10.9.1　资源结构

资源用类似于磁盘目录结构的方式保存，目录通常包含 3 层。最上面的目录很类似于一个文件系统的根目录。每一个在根部下的目录条目总是在它自己权利下的一个目录。这些二级目录中的每一个对应于一个资源类型（字符串表、菜单、对话框、菜单等）。在每一个二级资源类型目录下，是第三级子目录。目录结构如图 10.26 所示。

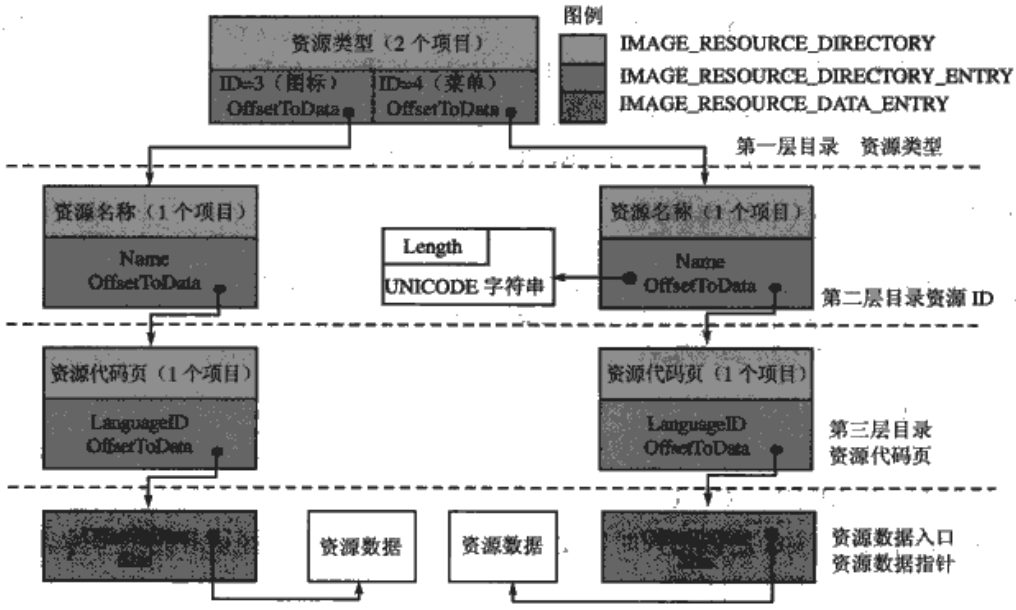


图 10.26　资源的树形结构

1. 资源目录结构

数据目录表中的 IMAGE_DIRECTORY_ENTRY_RESOURCE 条目包含资源的 RVA 和大小。资源目录结构中的每一个节点都是由 IMAGE_RESOURCE_DIRECTORY 结构和紧跟其后的数个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构组成的，这两种结构组成了一个目录块。

IMAGE_RESOURCE_DIRECTORY 结构长度为 16 字节，共有 6 个字段。其定义如下所示：

```
IMAGE_RESOURCE_DIRECTORY STRUCT
  Characteristics          DWORD   ?    ; 理论上是资源的属性标志，但是通常为 0
  TimeDateStamp            DWORD   ?    ; 资源建立的时间
  MajorVersion             WORD    ?    ; 理论上放置资源的版本，但是通常为 0
  MinorVersion             WORD    ?
  NumberOfNamedEntries     WORD    ?    ; 使用名字的资源条目的个数
  NumberOfIdEntries        WORD    ?    ; 使用 ID 数字资源条目的个数
IMAGE_RESOURCE_DIRECTORY ENDS
```

在这个结构中唯一令人感兴趣的字段是 NumberOfNamedEntries 和 NumberOfIdEntries，它们说明了本目录中目录项的数量。NumberOfNamedEntries 字段是以字符串命名的资源数量，NumberOfIdEntries 是以整型数字来命名的资源数量，两者加起来是本目录中的目录项总和，即为后面紧跟的 IMAGE_RESOURCE_DIRECTORY_ENTRY 数目。

2. 资源目录入口结构

紧跟着资源目录结构后的就是资源目录入口（Resource Dir Entries）结构，此结构长度为 8 个字节，包含 2 个字段。IMAGE_RESOURCE_DIRECTORY_ENTRY 结构定义如下：

```
IMAGE_RESOURCE_DIRECTORY_ENTRY STRUCT
    Name                 DWORD   ?    ; 目录项的名称字符串指针或 ID
    OffsetToData         DWORD   ?    ; 资源数据偏移地址或子目录偏移地址
IMAGE_RESOURCE_DIRECTORY_ENTRY ENDS
```

根据不同情况，这两个字段的含义不一样。

（1）Name 字段

该字段定义目录项的名称或 ID。当结构用于第一层目录时，定义的是资源类型；当结构用于第二层目录时，定义的是资源的名称；当结构用于第三层目录时，定义的是代码页编号。当最高位是 0 时，表示字段的值作为 ID 使用；而最高位为 1 时，字段的低位作为指针使用，资源名称字符串是使用 UNICODE 编码，这个指针并不直接指向字符串，而是指向一个 IMAGE_RESOURCE_DIR_STRING_U 结构。其定义如下：

```
IMAGE_RESOURCE_DIR_STRING_U STRUCT
    Length                  WORD    ?       ; 字符串的长度
    NameString              WCHAR   ?       ; UNICODE 字符串，字对齐的，长度是可变的，由
                                            ; Length 指明 Unicode 字符串的长度
IMAGE_RESOURCE_DIR_STRING_U ENDS
```

（2）OffsetToData 字段

该字段是一个指针。当最高位（位 31）为 1 时，低位数据指向下一层目录块的起始地址；当最高位为 0 时，指针指向 IMAGE_RESOURCE_DATA_ENTRY 结构。

当将 Name 和 OffsetToData 用做指针时需要注意，该指针是从资源区块开始的地方算起的偏移量，不是 RVA，即根目录的起始位置的偏移量。

有一点要说明的是，当 IMAGE_RESOURCE_DIRECTORY_ENTRY 用在第一层时，它的 Name 字段作为资源类型使用。当资源类似以 ID 定义并且数值在 1 到 16 之间时，表示是系统预定义的类型，具体见表 10-16。

表 10-16 系统预定义资源类型

类型 ID 值	资源类型	类型 ID 值	资源类型
01h	光标（Cursor）	08h	字体（Font）
02h	位图（Bitmap）	09h	加速键（Accelerators）
03h	图标（Icon）	0Ah	未格式资源（Unformatted）
04h	菜单（Menu）	0Bh	消息表（MessageTable）
05h	对话框（Dialog）	0Ch	光标组（Group Cursor）
06h	字符串（String）	0Eh	图标组（Group Icon）
07h	字体目录（Font Directory）	10h	版本信息（Version Information）

3. 资源数据入口

经过三层 IMAGE_RESOURCE_DIRECTORY_ENTRY（一般是 3 层，也有可能更少些。第一层是资源类型，第二层是资源名，第三层是资源的 Language），第三层目录结构中的 OffsetToData 指向 IMAGE_RESOURCE_DATA_ENTRY 结构。该结构描述了资源数据的位置和大小，其结构定义如下：

```
IMAGE_RESOURCE_DATA_ENTRY STRUCT
    OffsetToData            DWORD ? ; 资源数据的 RVA
    Size                    DWORD ? ; 资源数据的长度
    CodePage                DWORD ? ; 代码页，一般为 0
    Reserved                DWORD ? ; 保留字段
} IMAGE_RESOURCE_DATA_ENTRY ENDS
```

经过多层结构后，此处的 IMAGE_RESOURCE_DATA_ENTRY 结构就是真正的资源数据了。结构中的 OffsetToData 指向资源数据的指针，其为 RVA 值。

10.9.2　资源结构实例分析

在这里以光盘映像文件中提供的实例 pediy.exe 分析资源。数据目录表的第三个成员指向资源结构，该指针具体位置是在 PE 文件头的 88h 偏移处。用十六进制工具查看实例文件的 PE 文件头起始位置是 0C0h，则资源结构在整个文件的 0C0h+88h=148h 处，因此在 148h 处可以发现资源的 RVA 为 4000h。由于这个实例文件磁盘文件中的区块对齐值等于 1000h，与内存页对齐值相同，因此 RVA 与文件偏移地址不用转换。图 10.27 所示就是该程序资源的一部分，文件偏移 4000h 是资源起始地址。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00004000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	03	00	.....
00004010	03	00	00	00	28	00	00	80	04	00	00	00	40	00	00	80	....(...@..
00004020	0E	00	00	00	58	00	00	80	00	00	00	00	00	00	00	00	....X.....
00004030	00	00	00	00	00	00	01	00	01	00	00	00	70	00	00	80	..........p..
00004040	00	00	00	00	00	00	00	00	00	00	00	00	01	00	00	00	.....
00004050	E8	00	00	80	86	00	00	80	00	00	00	00	00	00	00	00	?.?.....
00004060	00	00	00	00	00	00	01	00	67	00	00	00	A0	00	00	80	........g...?.
00004070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	.....
00004080	04	08	00	00	B8	00	00	00	00	00	00	00	00	00	00	00	....?.....
00004090	00	00	00	00	00	00	01	00	09	04	00	00	C8	00	00	00	.....?..
000040A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	01	00	.....
000040B0	04	08	00	00	D8	00	00	00	00	41	00	00	E8	02	00	00	....?...A..?..
000040C0	00	00	00	00	00	00	00	00	00	44	00	00	5A	00	00	00	.....D..Z...
000040D0	00	00	00	00	00	00	00	00	E8	43	00	00	14	00	00	00	.....鑷.....
000040E0	00	00	00	00	00	00	00	00	05	00	50	00	45	00	44	00	.....P.E.D.
000040F0	49	00	59	00	00	00	00	00	00	00	00	00	00	00	00	00	I.Y.....
00004100	28	00	00	00	20	00	00	00	40	00	00	00	01	00	04	00	(... ...@.....

图 10.27　资源的十六进制形式

1．根目录

文件偏移 4000h 处指向的是根目录，第一行数据就是 IMAGE_RESOURCE_DIRECTORY 结构，其各项的值如图 10.28 所示。

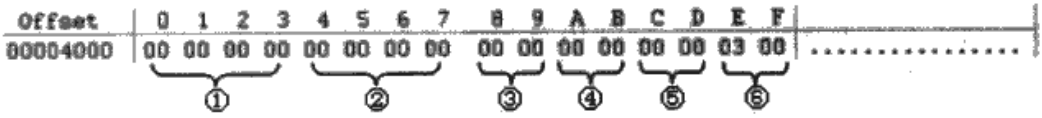


图 10.28　根目录的 IMAGE_RESOURCE_DIRECTORY 结构

从图 10.28 读出根目录的 IMAGE_RESOURCE_DIRECTORY 各结构成员值：Characteristics 为 00000000，TimeDateStamp 为 00000000，MajorVersion 为 0000，MinorVersion 为 0000，NumberOfNamedEntries 为 0000，NumberOfIdEntries 为 0003。NumberOfNamedEntries 与 NumberOfIdEntries 的和是 3，表明这个程序有 3 个资源项目。也就是说，其后面紧跟着 3 个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构，根据图 10.27 中的数据将这 3 个结构整理见表 10-17。

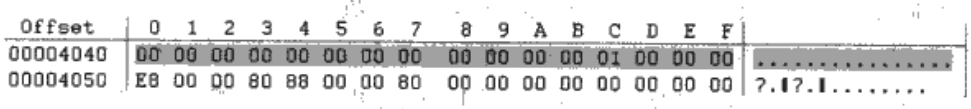
表 10-17　根目录下的 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构数据

	第一个 Directory_entry 结构	第二个 Directory_entry 结构	第三个 Directory_entry 结构
偏移地址	4010h	4018h	4020h
Name/Id	00000003h（ICON）	00000004h（MENU）	0000000E h（GROUP ICON）
OffsetToData	80000028h	80000040h	80000058h

以表 10-17 中的第二个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构为例分析资源的下一层。第一层目录，Name 字段是定义资源类型，目前其 ID 值为 04h，表明这是一个菜单资源。另外，OffsetToData 字段为 80000040h，第一个字节 80h 的二进制为 10000000，最高位为 1，说明还有下一层。所以 OffsetToData 的低位数据 40h 指向第二层目录块。第二层目录块的地址为资源块首地址加上 40h，即为 4000h+40h = 4040h。

2. 第二层目录

偏移 4040h 的数据即为第二层，见图 10.29。

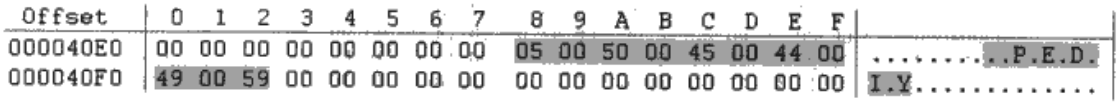


```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00004040  00 00 00 00 00 00 00 00  00 00 00 00 01 00 00 00  ................
00004050  E8 00 00 80 88 00 00 80  00 00 00 00 00 00 00 00  ?.?.?.?.........
```

图 10.29 第二层目录的 IMAGE_RESOURCE_DIRECTORY 结构

图 10.29 中阴影部分是第二层的 IMAGE_RESOURCE_DIRECTORY 结构成员：Characteristics 为 0，TimeDateStamp 为 0，MajorVersion 为 0，MinorVersion 为 0，NumberOfNamedEntries 为 0，NumberOfIdEntries 为 1。NumberOfNamedEntries 与 NumberOfIdEntries 的和是 1，表明这层有一个资源项目。也就是说，其后面紧跟着 1 个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构，即在文件偏移 4050h 处，Name 是 800000E8h，OffsetToData 是 80000088h。

当在第二层目录时，Name 字段定义的是资源名称，Name 字段第一个字节 80h 的二进制为 10000000，最高位为 1，表明这是一个指针，指向 IMAGE_RESOURCE_ DIR_STRING_U 结构，其地址为资源块首地址加上 Name 字段低位数据 0E8h，即 4000h+0E8h = 40E8h。具体见图 10.30 中阴影部分，图中显示 Length 是 05，NameString 是 Unicode 字符“PEDIY”，即这个资源名为“PEDIY”。



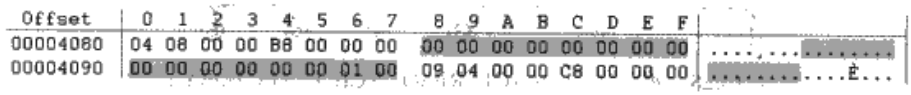
```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
000040E0  00 00 00 00 00 00 00 00  05 00 50 00 45 00 44 00  ..........P.E.D.
000040F0  49 00 59 00 00 00 00 00  00 00 00 00 00 00 00 00  I.Y.............
```

图 10.30 IMAGE_RESOURCE_DIR_STRING_U 结构

OffsetToData 字段是 80000088h，第一个字节 80h 的二进制为 10000000，最高位为 1，说明还有下一层。所以 OffsetToData 的低位数据 88h 指向第三层目录块。第三层目录块的地址为资源块首地址加上 88h，即为 4000h+88h = 4088h。

3. 第三层目录

文件偏移 4088h 的数据指向第三层，如图 10.31 所示。

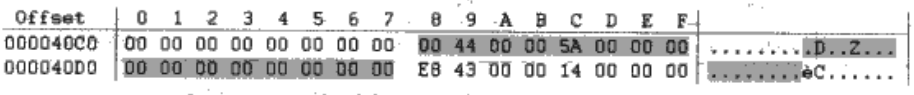


```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00004080  04 08 00 00 B8 00 00 00  00 00 00 00 00 00 00 00  ................
00004090  00 00 00 00 00 00 01 00  09 04 00 00 C8 00 00 00  ............È...
```

图 10.31 第三层目录的 IMAGE_RESOURCE_DIRECTORY 结构

从图 10.31 中可以看到第三层的 IMAGE_RESOURCE_DIRECTORY 结构成员：Characteristics 为 0，TimeDateStamp 为 0，MajorVersion 为 0，MinorVersion 为 0，NumberOfNamedEntries 为 0，NumberOfIdEntries 为 1。NumberOfNamedEntries 与 NumberOfIdEntries 的和是 1，表明这层有一个资源项目。也就是说，其后面紧跟着 1 个 IMAGE_RESOURCE_DIRECTORY_ENTRY 结构，即在文件偏移 4098h 处，Name 是 00000409h，OffsetToData 是 000000C8h。

当在第三层目录时，Name 字段定义的是代码页编号，00000409h 表示代码页是英语。OffsetToData 高位现在是 0，所以其低位数据 0C8h 指向 IMAGE_RESOURCE_DATA_ENTRY 结构，0C8h 加上资源块首地址，即 4000h+0C8h = 40C8h，见图 10.32 中阴影部分。



```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
000040C0  00 00 00 00 00 00 00 00  00 44 00 00 5A 00 00 00  .........D..Z...
000040D0  00 00 00 00 00 00 00 00  E8 43 00 00 14 00 00 00  ........èC......
```

图 10.32 IMAGE_RESOURCE_DATA_ENTRY 结构

在这里就能查看到 IMAGE_RESOURCE_DATA_ENTRY 结构成员值，OffsetToData 是 00004400，Size 是 0000005Ah，CodePage 是 00000000，Reserved 是 00000000h。此时图标的真正资源数据 RVA 为 4400h，大小为 5Ah。

10.9.3　资源编辑工具

资源数据，它们一般被存储在 PE 文件的.rsrc 区块中，并且不能通过由程序源代码定义的变量直接访问，Windows 提供函数直接或间接地把它们加载到内存中以备使用。

资源类型主要有以下几种：

- VC 类标准资源（包括菜单、对话框、串表等资源）；
- Delphi 类标准资源（Rcdata 资源）；
- 非标准的 Unicode 字符（主要是一些 VB 编译程序等）。

这些资源可以定制和修改，如更改字体、对话框，增加按钮、菜单等。Visual C ++等编译器可以直接编辑修改 PE 文件的资源。另外常用的资源修改工具有 Resource Hacker 和 eXeScope 等，它们也可直接编辑修改用 VC++及 Delphi 编制的程序资源，包括 EXE、DLL 和 OCX 等，并且是功能强大的汉化和调试辅助工具。

10.10　TLS 初始化

使用线程本地存储器（TLS）可以将数据与执行的特定线程联系起来。当使用__declspec(thread)声明的 TLS 变量时，编译器将它们放入一个叫.tls 的区块里。当应用程序加载到内存中时，系统要寻找可执行文件中的.tls 区块，并且动态地分配一个足够大的内存块，以便存放所有的 TLS 变量。系统也将一个指向已分配的内存的指针放到 TLS 数组里，这个数组由 FS:[2Ch]指向（在 x86 架构上）。

在一个可执行文件中的线程局部存储（TLS）数据是由数据目录表中的 IMAGE_DIRECTORY_ENTRY_TLS 条目指出的。如果是非零的，这个字段指向一个 IMAGE_TLS_DIRECTORY 结构。其结构如下：

```
IMAGE_TLS_DIRECTORY32  STRUC
    StartAddressOfRawData  DWORD    ? ; 内存起始地址，用于初始化一个新线程的 TLS
    EndAddressOfRawData    DWORD    ? ; 内存终止地址，用于初始化一个新线程的 TLS
    AddressOfIndex         DWORD    ? ; 运行库使用这个索引来定位线程局部数据
    AddressOfCallBacks     DWORD    ? ; PIMAGE_TLS_CALLBACK 函数指针数组的地址
    SizeOfZeroFill         DWORD    ? ; 后面跟零的个数
    Characteristics        DWORD    ? ; 保留，目前设为 0
IMAGE_TLS_DIRECTORY32  ENDS
```

AddressOfCallBacks 是线程建立和退出时的回调函数，包括主线程和其他线程。当一个线程被创建或销毁时，在列表中的每一个函数被调用。一般程序都没有回调函数，这个列表是空的。有一点特别注意，程序运行时，TLS 数据初始化和 TLS 回调函数都在入口点（AddressOfEntryPoint）之前执行，也就是说，TLS 是程序最开始运行的地方，许多病毒或外壳程序就利用这点执行一些特殊操作。程序退出时，TLS 回调函数再被执行一次。

在 IMAGE_TLS_DIRECTORY 结构中的地址是虚拟地址，而不是 RVA。这样，如果可执行文件不是从基地址装入，那么这些地址就会通过基址重定位来修正，而且，IMAGE_TLS_DIRECTORY 本身并不在.tls 区块中，它存在于.rdata 区块里。

10.11　调试目录

当用调试信息构建一个可执行文件时，按照惯例应该包括这种信息的格式及位置细节。操作系统并不需要这个来运行可执行文件，但这对开发工具是有用的。

数据目录表的第 7 个条目（IMAGE_DIRECTORY_ENTRY_DEBUG）指向调试目录，它由一个 IMAGE_DEBUG_DIRECTORY 结构数组组成。这些结构保持存储在文件中的变量的类型、尺寸和位置的调试信息。debug 目录里的元素数量可以通过 DataDirectory 内的 Size 字段来计算。其结构定义如下：

```
IMAGE_DEBUG_DIRECTORY  STRUC
  Characteristics       DWORD   ? ;未使用，设为 0
  TimeDateStamp         DWORD   ? ;debug 信息的时间/日期戳
  MajorVersion          WORD    ? ;debug 信息的主版本，未使用
  MinorVersion          WORD    ? ;debug 信息的次版本，未使用
  Type                  DWORD   ? ;debug 信息的类型
  SizeOfData            DWORD   ? ;debug 数据的大小
  AddressOfRawData      DWORD   ? ;当被映射到内存时，debug 数据的 RVA，为 0 不被映射
  PointerToRawData      DWORD   ? ;debug 数据的文件偏移（不是 RVA）
IMAGE_DEBUG_DIRECTORY   ENDS
```

到目前为止，debug 信息的最普遍形式是 PDB 文件。PDB 文件基本上是 CodeView 样式的 debug 信息的演变。PDB 信息的存在是由一个 IMAGE_DEBUG_TYPE_CODEVIEW 类型的调试目录字段指出的。如果检查由这个条目指向的数据，将发现一个简短的 CodeView 样式的头部。这个 debug 数据的多半是指向外部 PDB 文件的路径。在 Visual Studio 6.0 中，debug 头部以 NB10 标识开始，在 Visual Studio .NET 中，头部是以 RSDS 开始的。

在 Visual Studio 6.0 中，COFF 格式的 debug 信息能用/DEBUGTYPE:COFF 链接器开关生成。在 Visual Studio .NET 中这个选项没有了。

10.1 延迟装入数据

延迟装入一个 DLL 是一种混合方式，其是通过 LoadLibrary 和 GetProcAddress 获得延迟加载函数的地址，然后直接转向对延迟加载函数的调用。

记住延迟装入不是操作系统的一个特征，它完全是通过链接器和运行库加入额外的代码和数据来实现的。同样地，无法在 winnt.h 里找到关于延迟装入的更多参考，不过，可以在延迟装入数据和常规的输入数据之间看到一定的相似之处。

数据目录表中的 IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT 条目指向延迟装入的数据，这是一个指向 ImgDelayDescr 结构数组的 RVA，这个结构定义在 Visual C++的 DelayImp.H 中，表 10-18 说明了它的内容。每一个被延迟装入的 DLL 都对应一个 ImgDelayDescr 结构。

表 10-18 ImgDelayDescr 结构

大 小	成 员	描 述
DWORD	grAttrs	这个结构的属性。目前唯一被定义的旗标是 dlattrRva，表明这个结构中的字段应该被认为是 RVA，而不是虚拟地址
RVA	rvaDLLName	指向一个被输入的 DLL 名称的 RVA。这个字符串被传递给 LoadLibrary
RVA	rvaHmod	指向一个 HMODULE 大小的内存位置的 RVA，当延迟装入的 DLL 被装入内存后，它的模块句柄（hModule）被保存在这个地方
RVA	rvaIAT	指向这个 DLL 的输入地址表的 RVA，它与常规的 IAT 表的格式相同
RVA	rvaINT	指向这个 DLL 的输入名称表的 RVA。它与常规的 INT 表的格式相同
RVA	rvaBoundIAT	可选的绑定 IAT 的 RVA，指向这个 DLL 的输入地址表的绑定拷贝。它与常规的 IAT 表的格式相同。目前，这个 IAT 的拷贝并不是实际的绑定，但是这个特征可能会加到绑定程序的未来版本中
RVA	rvaUnloadIAT	原始 IAT 的可选拷贝的 RVA，它指向这个 DLL 的输入地址表的未绑定拷贝。它与常规的 IAT 表的格式相同。目前总是设为 0
DWORD	dwTimeStamp	延迟装入的输入 DLL 的时间/日期戳。通常设为 0

ImgDelayDescr 结构的关键在于它包括了对应 DLL 的 IAT 和 INT 的地址，这些表在格式上与常规的输入表是相同的，唯一的区别是它们由运行库代码而不是操作系统进行写入和读出的。当第一次从一个延迟装入的 DLL 中调用一个 API 函数时，运行库调用 LoadLibrary（如果需要），然后是 GetProcAddress，最后得到的地址被存在延迟装入的 IAT 表中，这样以后每次调用这个 API 都会直接到这里来。

在 Visual C++ 6.0 中有延迟装入数据最初的原型，ImgDelayDescr 中所有包含地址的域均是虚拟地址，而不是 RVA。换句话说，它们包含延迟装入数据所在位置的实际地址。这些域是双字，是 x86 上一个指针的大小。现在向 IA-64 的快速移植正在被支持。很显然 4 字节不足以装下一个完整的地址，在这点上，微软做了件正确的事情，已将包含地址的字段改为了 RVA。表 10-18 中已经用了修正的结构定义和名称。

关于在 ImgDelayDescr 结构中是使用 RVA 还是虚拟地址，现在仍有争论，在这个结构中有一个字段是旗标值。当 grAttrs 字段被设为 1 时，结构成员被当成是 RVA。这是唯一与 Visual Studio.NET 和 64 位编译器一起出现的选项。如果在 grAttrs 中的这个位关掉，ImgDelayDescr 字段将是虚拟地址。

10.1 程序异常数据

一些体系结构（包括 IA-64）不使用基于框架的异常处理，像在 x86 上就是这样。取而代之的是，它们使用基于表的异常处理。这种方式下，里面有一个表，它包含了每一个有可能受异常展开影响的函数信息。为每个函数准备的数据包括起始地址、结束地址以及关于异常应该如何处理并在什么地方被处理的信息。当一个异常发生时，系统通过遍历这个表来定位合适的入口并处理它。异常表是一个 IMAGE_RUNTIME_FUNCTION_ENTRY 结构数组，数组是由数据目录表中的 IMAGE_DIRECTORY_ENTRY_EXCEPTION 条目来指向的。IMAGE_RUNTIME_FUNCTION_ENTRY 结构的格式随体系结构的不同而不同。对于 IA-64，布局看上去像这个样子：

```
DWORD BeginAddress;
DWORD EndAddress;
DWORD UnwindInfoAddress;
```

UnwindInfoAddress 数据的格式没有在 winnt.h 中给出，但是，可以从 Intel 的“IA-64 Software Conventions and Runtime Architecture Guide”这份资料的第 11 章中找到。

10.1 .Net 头部

.Net 文件为 Microsoft .Net 环境生成的可执行文件。.Net 环境由公共语言运行环境（CLR）和.Net 框架类库组成。可以把 CLR 看作是一台虚拟机，.Net 应用程序就在这台机器中运行。.Net 可执行文件的主要目的是获得.Net 特定的装入内存的信息，像元数据（Metadata）和中间语言（Intermediate Language，简称 IL）。另外，.Net 可执行文件依靠 MSCOREE.DLL 进行链接，这个 DLL 对于一个.Net 进程是起始点。当一个.Net 可执行文件装入时，它的入口点通常是一小块残余代码，这小块代码只不过是跳到 MSCOREE.DLL 中的一个输出函数（_CorExeMain 或_CorDllMain）。从那里，MSCOREE 接管并开始使用来自可执行文件的元数据和中间语言。这种运行类似于 Visual Basic 程序使用 MSVBVM60.DLL 的方式。

.Net 环境下的 PE 文件，在整体结构上与传统 PE 一致，所不同的，就是利用了数据目录表中的 IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 条目扩充了其结构。这个条目原本是设计用于 COM，但一直没有被使用，现在用于保存.Net 的信息结构，指向 IMAGE_COR20_HEADER。有关.Net 的具体内容，请参考第 9 章。

10.15 编写 PE 分析工具

常用的 PE 分析工具有 LordPE、Stud_PE 等，熟悉 PE 格式后，这些工具使用都比较简单，读者可自行摸索。本节将从学习角度，讲解如何编写一个简单的 PE 结构分析程序，这对理解 PE 格式和相关 PE 编程非常有帮助。

PE 格式分析工具的编写并不难，主要就是对 PE 格式的各个结构进行定位。本节定义了一个 MAP_FILE_STRUCT 结构来存放有关信息。结构如下：

```
typedef struct _MAP_FILE_STRUCT
{
    HANDLE hFile;          //文件句柄
    HANDLE hMapping;       //映射文件句柄
    LPVOID ImageBase;      //映像基址
} MAP_FILE_STRUCT;
```

10.15.1 文件格式检查

文件格式可以通过 PE header 开始的标志 Signature 来检测。也许读者会说，检测 DOS Header 的 Magic Mark 不是也可以检测此 PE 文件是否合法吗？这个想法没有错，但是检测 Magic Mark 不一定能确定就是 PE 文件，如果某文本文件正好在开始就是“MZ”字符串，就会误判断。

（1）判断文件开始的第一个字段是否为 IMAGE_DOS_SIGNATURE，即 5A4Dh。

（2）再通过 e_lfanew 找到 IMAGE_NT_HEADERS，判断 Signature 字段的值是否为 IMAGE_NT_SIGNATURE，即 00004550h，如果是 IMAGE_NT_SIGNATURE，就可以认为该文件是 PE 格式。

具体实现的代码如下：

```
BOOL IsPEFile(LPVOID ImageBase)
{
    PIMAGE_DOS_HEADER  pDH=NULL;
    PIMAGE_NT_HEADERS  pNtH=NULL;

    if(!ImageBase)                                    //判断映像基址
        return FALSE;
    pDH=(PIMAGE_DOS_HEADER)ImageBase;
    if(pDH->e_magic!=IMAGE_DOS_SIGNATURE)             //判断是否为 MZ
        return FALSE;
    pNtH=(PIMAGE_NT_HEADERS32)((DWORD)pDH+pDH->e_lfanew);
    if (pNtH->Signature != IMAGE_NT_SIGNATURE )       //判断是否为 PE 格式
        return FALSE;

    return TRUE;
}
```

10.15.2 FileHeader 和 OptionalHeader 内容的读取

只要得到了 IMAGE_NT_HEADERS，根据 IMAGE_NT_HEADERS 的定义，就可以找到 IMAGE_FILE_HEADER 和 PIMAGE_OPTIONAL_HEADER。

（1）得到 IMAGE_NT_HEADERS 结构指针的函数：

```
PIMAGE_NT_HEADERS GetNtHeaders(LPVOID ImageBase)
{
   PIMAGE_DOS_HEADER  pDH=NULL;
   PIMAGE_NT_HEADERS  pNtH=NULL;
```

```
    if(!IsPEFile(ImageBase))
          return NULL;

    pDH=(PIMAGE_DOS_HEADER)ImageBase;
    pNtH=(PIMAGE_NT_HEADERS)((DWORD)pDH+pDH->e_lfanew);
    return pNtH;
}
```

（2）得到 IMAGE_FILE_HEADER 结构指针的函数：

```
PIMAGE_FILE_HEADER WINAPI  GetFileHeader(LPVOID ImageBase)
{
    PIMAGE_NT_HEADERS  pNtH=NULL;
    pNtH=GetNtHeaders(ImageBase);
    if(!pNtH)
      return NULL;
    pFH=&pNtH->FileHeader;
    return pFH;
}
```

（3）得到 IMAGE_OPTIONAL_HEADER 结构指针的函数：

```
PIMAGE_OPTIONAL_HEADER WINAPI  GetOptionalHeader(LPVOID ImageBase)
{
    PIMAGE_OPTIONAL_HEADER pOH=NULL;
    pNtH=GetNtHeaders(ImageBase);
   if(!pNtH)
     return NULL;
    pOH=&pNtH->OptionalHeader;
    return pOH;
}
```

得到指向 IMAGE_NT_HEADERS 结构和 IMAGE_OPTIONAL_HEADER 结构指针的函数以后，接下来就是要把 FileHeader 和 OptionalHeader 的信息显示出来。例如，要把 FileHeader 和 OptionalHeader 的信息以十六进值方式显示在编辑控件上，此时先用函数 wsprintf()将欲显示的值进行格式化，然后再调用 API 函数 SetDlgItemText 即可。具体代码如下：

```
void    ShowFileHeaderInfo(HWND hWnd)
{
    char   cBuff[10];
    PIMAGE_FILE_HEADER pFH=NULL;
    pFH=GetFileHeader(stMapFile.ImageBase);   //得到文件头指针
    if(!pFH)
    {
      MessageBox(hWnd,"Get File Header faild! :(","PEInfo_Example",MB_OK);
       return;
     }
    //下面的代码是将有关信息按十六进制格式化，并显示在编辑控件上
    wsprintf(cBuff, "%04lX", pFH->Machine);
    SetDlgItemText(hWnd,IDC_EDIT_FH_MACHINE,cBuff);
    ……（省略部分代码）
}
void    ShowOptionHeaderInfo(HWND hWnd)
{
    char   cBuff[10];
    PIMAGE_OPTIONAL_HEADER pOH=NULL;
    pOH=GetOptionalHeader(stMapFile.ImageBase); //得到可选文件头指针
   if(!pOH)
   {
     MessageBox(hWnd,"Get Optional Header faild! :(","PEInfo_Example",MB_OK);
     return;
```

```
    }
//下面的代码是将有关信息按十六制格式化，并显示在编辑控件上
   wsprintf(cBuff, "%08lX", pOH->AddressOfEntryPoint);
SetDlgItemText(hWnd,IDC_EDIT_OH_EP,cBuff);
……（省略部分代码）
}
```

10.15.3 得到数据目录表信息

数据目录表（DataDirectory）由一组数组构成，每组项目包括执行文件的重要部分的起始 RVA 和长度。因为数据目录有 16 项，如果不嫌麻烦，代码可以一行行的写，在这里定义了一个编辑控件 ID 的结构数组，用一个循环就可以了。具体代码如下：

```
typedef struct
{
    UINT   ID_RVA;
    UINT   ID_SIZE;
} DataDir_EditID;

DataDir_EditID EditID_Array[]=
{
    {IDC_EDIT_DD_RVA_EXPORT,        IDC_EDIT_DD_SIZE_EXPORT},
    {IDC_EDIT_DD_RVA_IMPORT,        IDC_EDIT_DD_SIZE_IMPORT},
    {IDC_EDIT_DD_RVA_RES,           IDC_EDIT_DD_SZIE_RES},
    {IDC_EDIT_DD_RVA_EXCEPTION,     IDC_EDIT_DD_SZIE_EXCEPTION},
……（省略部分代码）
};
void ShowDataDirInfo(HWND hDlg)
{
    char   cBuff[9];
    PIMAGE_OPTIONAL_HEADER pOH=NULL;
    pOH=GetOptionalHeader(stMapFile.ImageBase);
    if(!pOH)
       return;
   for(int i=0;i<16;i++) //利用 for 循环将信息显示在编辑控件上
   {
     wsprintf(cBuff, "%08lX", pOH->DataDirectory[i].VirtualAddress);
     SetDlgItemText(hDlg,EditID_Array[i].ID_RVA,cBuff);

     wsprintf(cBuff, "%08lX", pOH->DataDirectory[i].Size);
     SetDlgItemText(hDlg,EditID_Array[i].ID_SIZE,cBuff);
}
}
```

10.15.4 得到区块表信息

紧接 IMAGE_NT_HEADERS 以后就是区块表（Section Table）了，Section Table 则是由 IMAGE_SECTION_HEADER 组成的数组。如何得到 Section Table 的位置呢？换句话说，也就是如何得到第一个 IMAGE_SECTION_HEADER 的位置。在 Visual C++中，可以利用 IMAGE_FIRST_SECTION 宏来轻松地得到第一个 IMAGE_SECTION_HEADER 的位置。

又因为区块的个数已经在文件头中指明了，所以只要得到第一个区块的位置，然后再利用一个循环语句就可以得到所有区块的信息了。

下面的 GetFirstSectionHeader 函数是利用 IMAGE_FIRST_SECTION 宏得到区块表的起始位置。

```
PIMAGE_SECTION_HEADER  GetFirstSectionHeader(PIMAGE_NT_HEADERS  pNtH)
{
    PIMAGE_SECTION_HEADER pSH;
```

```
    pSH =IMAGE_FIRST_SECTION(pNtH);
    return  pSH;
}
```

这里必须要强调一下，在一个 PE 文件中，OptionalHeader 的大小是可以变化的，虽然它的大小通常为 E0h，但是总有例外。原因是可选文件头的大小是由文件头中的 SizeOfOptionalHeader 字段指定的，并不是个固定值。这也是 IMAGE_FIRST_SECTION 宏对于可选文件头的大小为什么不直接用固定值的原因。系统的 PE 加载器在加载 PE 文件的时候，也是利用了文件头中的 SizeOfOptionalHeader 字段的值来定位区块表的，而不是用固定值。能否正确地定位到区块表，取决于 SizeOfOptionalHeader 字段的值的正确性。这是个很容易被忽略的问题，因此会导致一些程序的 BUG，如 Peditor v1.7 和 PEiD v0.9 都有这样的 BUG。本例中，用 ListView 控件来显示 PE 文件中的区段信息。具体代码如下：

```
void ShowSectionHeaderInfo(HWND hDlg)
{
LVITEM              lvItem;
char                cBuff[9],cName[9];
WORD                i;
  PIMAGE_FILE_HEADER    pFH=NULL;
PIMAGE_SECTION_HEADER   pSH=NULL;

pFH=GetFileHeader(stMapFile.ImageBase); //得到文件头指针
if(!pFH)
      return;
pSH=GetFirstSectionHeader(stMapFile.ImageBase); //得到第一个块表的指针
for( i=0;i<pFH->NumberOfSections;i++)   //在列表控件中依次显示各个区块的信息
{
    memset(&lvItem, 0, sizeof(lvItem));
    lvItem.mask    = LVIF_TEXT;
    lvItem.iItem   = i;
    memset(cName,0,sizeof(cName));
    memcpy(cName, pSH->Name, 8);
    lvItem.pszText = cName;
    SendDlgItemMessage(hDlg,1006,LVM_INSERTITEM,0,(LPARAM)&lvItem);
    ……（省略部分类似代码）
    ++pSH;
}
}
```

10.15.5　得到输出表信息

输出表（Export Table）中的主要成分是一个表格，内含函数名称、输出序数等。输出表是数据目录表的第一个成员，其指向 IMAGE_EXPORT_DIRECTORY 结构。输出函数的个数由结构 IMAGE_EXPORT_DIRECTORY 的字段 NumberOfFunctions 来说明。实际上，也有例外。例如，在写一个 DLL 的时候，可以用 DEF 文件来制定输出函数的名称、序号等。请看下面这个 DEF 文件的内容：

```
LIBRARY TESTEXPORTFUNCS
EXPORTS
        func1    @1
        func2    @3
        func3    @5
        func4    @8
        func5    @12
        func6    @13
        func7    @15
        func8    @17
        func9    @20
        func10   @23
```

```
        func11    @31
```

在这个文件中，共输出了 11 个函数（func1～func11），而输出函数的序号却是从 1 至 31，如果没有考虑这一点的话，很有可能会在这里出错。因为这时 IMAGE_EXPORT_DIRECTORY 的字段 NumberOfFunctions 值为 0x1F，即 31。如果认为 NumberOfFunctions 值就为输出函数个数的话，那就错了。图 10.33 所示的就是程序出错时的界面。

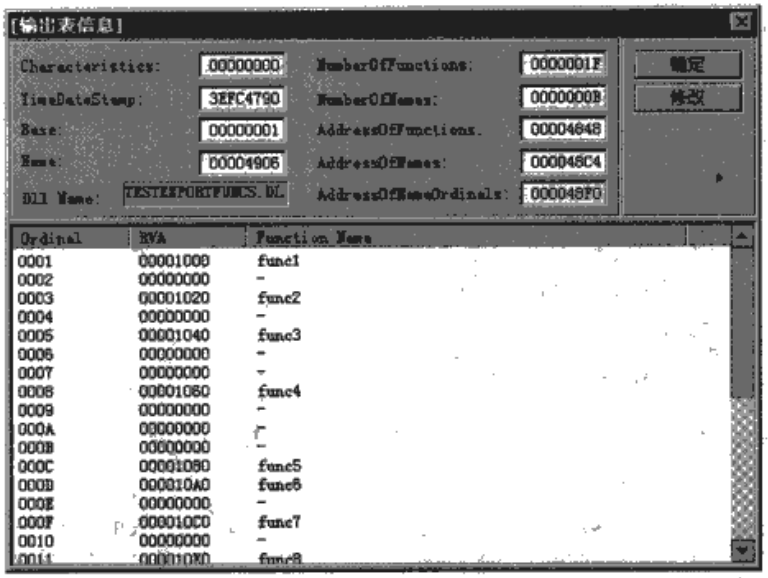


图 10.33　显示输出函数信息

在这里使用十六进制工具来分析一下这个 DLL，如图 10.34 所示。11 个小框中的数据才是真正的输出函数的 RVA，其余的所谓的“输出函数的 RVA”则用 0 填充了。

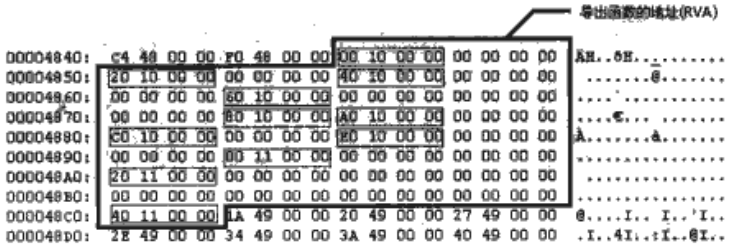


图 10.34　用十六进制工具分析

因此编程时，必须将这些特殊情况考虑进去，正确显示输出表和输出函数信息的程序代码见光盘映像文件中的 ShowExportFuncsInfo(HWND hDlg)。

10.15.6　得到输入表信息

数据目录表第二成员指向输入表。输入表以一个 IMAGE_IMPORT_DESCRIPTOR 结构开始，以一个空的 IMAGE_IMPORT_DESCRIPTOR 结构结束。在这里可以通过 GetFirstImportDesc 函数得到 Import Table 在文件中的位置。GetFirstImportDesc 函数的定义如下：

```
PIMAGE_IMPORT_DESCRIPTOR  GetFirstImportDesc(LPVOID ImageBase)
{
   PIMAGE_IMPORT_DESCRIPTOR pImportDesc;
   pImportDesc=(PIMAGE_IMPORT_DESCRIPTOR)GetDirectoryEntryToData \
(ImageBase,IMAGE_DIRECTORY_ENTRY_IMPORT);
   if(!pImportDesc)
```

```
        return NULL;

    return  pImportDesc;
}
```

找到了输入表的位置，可以通过一个循环来得到整个输入表，循环终止的条件是 IMAGE_IMPORT_DESCRIPTOR 结构为空。请看 ShowImportDescInfo 函数的定义：

```
void  ShowImportDescInfo(HWND hDlg)
{
    HWND      hList;
    LVITEM    lvItem;
    char      cBuff[10], * szDllName;
    PIMAGE_NT_HEADERS       pNtH=NULL;
    PIMAGE_IMPORT_DESCRIPTOR  pImportDesc=NULL;

    memset(&lvItem, 0, sizeof(lvItem));
    hList=GetDlgItem(hDlg,IDC_IMPORT_LIST);
    SendMessage(hList,LVM_SETEXTENDEDLISTVIEWSTYLE,0,(LPARAM)0x20);

    pNtH=GetNtHeaders(stMapFile.ImageBase);
    pImportDesc=GetFirstImportDesc(stMapFile.ImageBase);
    if(!pImportDesc)
     {
        MessageBox(hDlg,"Can't get ImportDesc:(","PEInfo_Example",MB_OK);
        return;
     }

    int i=0;
while(pImportDesc->FirstThunk)
    {
      memset(&lvItem, 0, sizeof(lvItem));
      lvItem.mask    = LVIF_TEXT;
      lvItem.iItem   = i;

     szDllName=(char*)RvaToPtr(pNtH,stMapFile.ImageBase,pImportDesc->Name);
      lvItem.pszText = szDllName;
     SendDlgItemMessage(hDlg,IDC_IMPORT_LIST,LVM_INSERTITEM,0,(LPARAM)&lvItem);
      ……（省略部分类似代码）

      ++i;
      ++pImportDesc;
    }
```

在 ShowImportDescInfo 函数中，首先用 GetFirstImportDesc 函数得到指向第一个 IMAGE_IMPORT_DESCRIPTOR 结构的指针 pImportDesc，以 pImportDesc->FirstThunk 为真来作为循环的条件，循环得到 Import Table 的各项信息。

通过上面的 ShowImportDescInfo 函数，可以得到 PE 文件所引入的 DLL 的信息，接下来的任务就是如何分析得到通过 DLL 所输入的函数的信息，这里必须通过 IMAGE_IMPORT_DESCRIPTOR 所提供的信息来得到输入的函数信息。具体代码见光盘映像文件中的 ShowImportFuncsByDllIndex(HWND hDlg,int index) 函数。可以通过名字和序号来引入所用的函数，怎么来区分一个函数是如何引入的呢？在于 IMAGE_THUNK_ DATA 值的高位，如果被置位了，低 31 位被看作是一个序数值。如果高位没被置位，IMAGE_THUNK_DATA 值是一个指向 IMAGE_IMPORT_BY_NAME 的 RVA。如果两者都不是，则可以认为 IMAGE_THUNK_DATA 值为函数的内存地址。

另外，微软的 ImageHlp 库中提供了大量的有关对 PE Image 操作的 API，直接调用就可，当然这些 API 的功能读者也可以自己来实现。

第11章 结构化异常处理

结构化异常处理（Structured Exception Handling，简称 SEH）是 Windows 操作系统处理程序错误或异常的技术。SEH 是 Windows 操作系统的一种系统机制，与特定的程序设计语言无关。本章仅简单地从调试角度认识一下 SEH，有关 SEH 更深层的知识，请参考《软件加密技术内幕》一书中温玉杰撰写的“第 4 章 Windows 下的异常处理”或其他相关资料。

11.1 基本概念

Intel 公司在从 386 开始的 IA-32 家族处理器中引入了中断（Interrupt）和异常（Exception）的概念。中断是由外部硬件设备或异步事件产生的。异常是由内部事件产生的，异常又可分为故障、陷阱和终止三类。故障和陷阱，正如其名称所暗示的，是可恢复的；终止类异常是不可恢复的，如果发生了这种异常，系统必须重启。

11.1.1 异常列表

所谓异常就是在应用程序的正常执行过程中发生的不正常事件。CPU 引发的异常称为硬件异常，例如访问一个无效的内存地址。操作系统或应用程序引发的异常称为软件异常。表 11-1 是常见异常的列表。

表 11-1 异常列表

中断类型号	类　型	相关指令
0	除数为 0 中断	DIV、IDIV
1	调试异常	任何指令
3	断点中断	INT 3 指令
4	溢出中断	INTO
5	边界检查	BOUND
6	非法指令故障	非法指令编码或操作数
7	设备不可用	浮点指令或 WAIT
8	双重故障	任何指令
0Ah	无效 TSS 中断	JMP、CALL、IRET、中断
0Bh	段不存在异常	装载段寄存器
0Ch	堆栈段异常	装载 SS 寄存器或 SS 段寻址
0Dh	通用保护异常	任何特权指令、任何访问存储器的指令
0Eh	页异常	任何访问存储器的指令

除了 CPU 捕获一个事件并引发一个硬件异常外，在代码中也可以强制引发一个软件异常。只需调用 RaiseException 函数：

```
VOID RaiseException(
    DWORD dwExceptionCode,              // 标识所引发异常的代码
    DWORD dwExceptionFlags,             // 异常继续是否执行的标识
    DWORD nNumberOfArguments,           // 附加信息
    CONST DWORD *lpArguments            // 附加信息
  );
```

程序捕获软件异常的方法与捕获硬件异常的完全相同。

11.1.2　异常处理的基本过程

首先来看看一个应用程序发生错误后，Windows 是如何结合 SEH 机制进行处理的。

（1）因为有多种异常，系统首先判断异常是否应发送给目标程序，如果应该发送，并且目标程序正处于被调试状态，则系统挂起程序，填写如下结构：

```
typedef _EXCEPTION_DEBUG_INFO{
    EXCEPTION_RECORD ExceptionRecord;
    DWORD dwFirstchance;
} EXCEPTION_DEBUG_INFO;
```

将成员 dwFirstchance 置为 1，并向调试器发送 EXCEPTION_DEBUG_EVENT 消息。剩下的事情就由调试器全权负责了，调试器可能处理这个异常，也可能无法处理。

（2）如果调试器未能处理异常或程序根本没有被调试，系统就会查找是否存在与线程相关的异常处理过程，如果目标程序中存在与线程相关的异常处理过程，系统就调用程序的线程相关的 SEH 异常处理例程，交由其处理。

（3）与线程相关的异常处理过程可以有一个或多个，每个可以选择处理或者不处理异常，如果它不处理并且存在多个线程相关的异常处理过程，可交由链起来的其他异常处理过程进行处理，依此类推。

（4）如果程序线程的异常处理均选择不处理异常，如果程序处于被调试状态，操作系统仍会再次挂起程序通知调试器，这次 EXCEPTION_DEBUG_INFO 结构的 dwFirstchance 成员置为 0。

（5）如果程序未处于被调试状态或者调试器仍然未能够处理，并且程序调用了 API 函数 SetUnhandledExceptionFilter 设置了与进程相关的异常处理过程的话，系统转向对它的调用。

（6）如果程序没有设置进程相关的异常处理过程或者进程相关的异常处理过程也未能处理这个异常，系统会调用默认的系统异常处理程序，通常显示一个对话框（如图 11.1 所示），可以选择“关闭”或者最后将其附加到调试器上的“调试”按钮。如果没有调试器能被附加于其上或调试器还是处理不了异常，系统就调用 ExitProcess 终结程序。

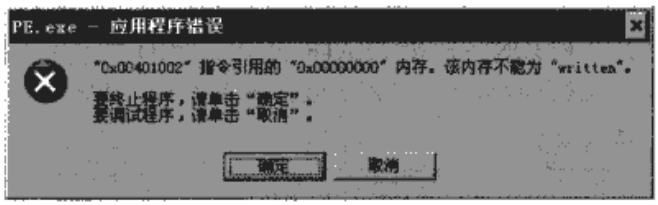


图 11.1　Windows XP 下的出错对话框

（7）不过在终结之前，系统再次调用发生异常的线程中所有的异常处理过程，这是线程异常处理过程获得的最后清理未释放资源的机会，其后程序就终结了。

学习 SEH 要树立一个最基本的观念：SEH 是系统发现异常或错误时，在终结应用程序之前给应用程

序的一个最后改正错误的机会，从程序设计的角度来说，就是系统在终结程序之前给程序的一个执行其预设定的回调函数的机会。

11.1.3 SEH 的分类

一般地，按作用域（即其监视范围）分，SEH 可分为两类：一类是监视某线程中某段代码是否发生异常的异常处理过程，一般称为线程相关的异常处理过程，或称为 Per_Thread 类型的异常处理过程，有时也简称为线程异常处理。另一类是监视整个进程中所有线程是否发生异常的异常处理过程，称为进程相关的异常处理过程，或称为“Final”型异常处理过程。有人也称之为筛选器，源于其对应于设置其的 API 函数 SetUnhandledExceptionFilter 中的 Filter 一词，在 Win32 API 文档中，称之为顶层（top-level）异常处理。

11.2 SEH 相关数据结构

SEH 涉及了几个关键的数据结构。

11.2.1 TEB 结构

TEB（Thread Environment Block，线程环境块）在 Windows 9x 系列中称为 TIB（Thread Information Block，线程信息块），它记录着线程的重要信息。每一个线程对应一个 TEB 结构，在 Windows 2000 DDK 中定义为：

```
typedef struct _NT_TIB {
    struct _EXCEPTION_REGISTRATION_RECORD *ExceptionList;
    PVOID StackBase;
    PVOID StackLimit;
    PVOID SubSystemTib;
    union {
        PVOID FiberData;
        ULONG Version;
    };
    PVOID ArbitraryUserPointer;
    struct _NT_TIB *Self;
} NT_TIB;
```

与异常处理相关的项是指向 EXCEPTION_REGISTRATION 结构的指针 ExceptionList，正好位于 TEB 的偏移 0 处。Windows 在创建线程时，操作系统均会为每个线程分配 TEB，而且都将 FS 段选择器指向当前线程的 TEB 数据（单 CPU 机器上的系统中在任何时刻只有一个线程在执行），这就为程序提供了存取 TEB 数据的途径，即 TEB 总是由[FS:0]指向的。

11.2.2 EXCEPTION_REGISTRATION 结构

TEB 偏移量为 00h 的_EXCEPTION_REGISTRATION_RECORD 主要用于描述线程异常处理过程的地址，多个该结构的链表描述多个线程异常处理过程的嵌套层次关系。其定义如下：

```
EXCEPTION_REGISTRATION struc
    prev            dd      ?           ; 指向前一个 EXCEPTION_REGISTRATION 结构的指针
    handler         dd      ?           ; 当前异常处理回调函数地址
EXCEPTION_REGISTRATION ends
```

其中，prev 是指向前一个 EXCEPTION_REGISTRATION（简称 ERR）的指针，形成一链状结构；handler 指向异常处理代码，如图 11.2 所示。当系统遇到一个它不知如何处理的异常时，它就查找异常处理链表。每个线程都有它自己的异常处理链表。

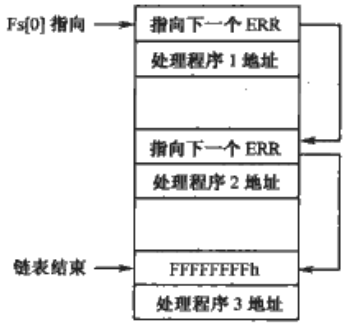


图 11.2　异常处理链表的示意图

本书实例 seh.exe 程序演示了一个结构异常处理的例子，用 OllyDbg 打开，注意 OllyDbg 的“调试选项/异常”设置，所有选项不勾选，按 F9 键让 seh.exe 跑起来，程序将在 401017 这行停住。

```
00401017     mov     eax, dword ptr [esi]      ;程序会停在这行，此时 esi=0
```

如果程序试图读取或写入 0 地址空间，那么 CPU 就会引发一个访问违规。在 Win98 下，这个地址空间是 0～0FFFh，在 NT 架构下是 0～FFFFh，这个地址空间是禁止进入的，如果读取就会发生内存访问违规现象。由于访问违规，OllyDbg 状态栏就有如一行字符提示，“访问违规：读取[00000000] -使用 Shift+F7/F8/F9 忽略程序异常”，如图 11.3 所示。按 Shift+F9 键即可忽略异常继续执行。

图 11.3　OllyDbg 状态栏里提示是否忽略异常

如果在 OllyDbg 的“调试选项/异常”设置忽略“非法访问内存”异常，OllyDbg 遇到此类异常将不暂停。OllyDbg 这种对异常处理的设置，非常有利于程序的跟踪，平时要擅于使用。

接下来，从调试器的角度来理解一下 SEH 的机制。在 OllyDbg 选项里忽略所有异常，重新加载实例 seh.exe 调试。相关汇编代码如下：

```
00401000    lea     eax, dword ptr [esp-8]
                                ;在栈中分配 8 字节存放 ERR 结构，即局部变量
                                ;并把栈顶指针保存到 eax
00401004    xchg    dword ptr fs:[0], eax
                                ;记住 fs:[0]永远是指向当前 ERR 结构的指针
                                ;fs:[0]<->eax，fs:[0]指向栈顶，eax 等于原 fs[0]
                                ;fs:[0]指向栈顶，准备在堆栈中构造 1 个 ERR 结构
                                ;eax 等于原 fs[0]，即指向原来的 ERR 结构的指针

0040100B    mov     ebx, 0040102E
00401010    push    ebx             ;压入堆栈，即填充 ERR 结构的 handler
00401011    push    eax             ;压入原 fs[0]，填充 ERR 结构的 prev
00401012    mov     esi, 0
00401017    mov     eax, dword ptr [esi]
                                ;读取线性地址 0，产生异常
                                ;异常后，控制权交给操作系统的异常处理代码，系统处理完
                                ;将跳到当前的 handler，即 0040102E 处继续执行

00401019    push    0               ;Style = MB_OK|MB_APPLMODAL
0040101B    push    00403000        ;Title = "OK"
00401020    push    00403010        ;Text = "SEH Fail"
00401025    push    0               ;hOwner = NULL
00401027    call    <jmp.&USER32.MessageBoxA>
0040102C    jmp     short 00401041
0040102E    push    0               ;Style = MB_OK|MB_APPLMODAL
```

```
00401030   push   00403000        ;Title = "OK"
00401035   push   00403003        ;Text = "SEH Succeed "
0040103A   push   0               ;hOwner = NULL
0040103C   call   <jmp.&USER32.MessageBoxA>
00401041   push   0
00401043   call   <jmp.&KERNEL32.ExitProcess>
00401048   jmp    dword ptr [<&USER32.MessageBoxA>]
0040104E   jmp    dword ptr [<&KERNEL32.ExitProcess>]
```

跟踪 SEH 程序一定要注意堆栈窗口，以查看 handler 的值，然后对此地址设断。用 OllyDbg 跟踪到 00401017 地址处时，查看堆栈窗口的数据，如图 11.4 所示。

```
0012FFBC  0012FFE0  Pointer to next SEH record
0012FFC0  0040102E  SE handler
```

图 11.4　查看 ERR 结构

此时，handler 值为 40102E，对 40102E 设断，然后按 F9 键让程序运行。当执行 00401017 一句读取线性地址 0 时，产生异常，将跳到 40102E（handler）处继续执行。因此，只有提前在 handler 地址处设置断点，才能正常跟踪 SEH 代码的运行，否则代码就有可能跟“飞”。

11.2.3　EXCEPTION_POINTERS、EXCEPTION_RECORD、CONTEXT

当一个异常发生时，操作系统向引起异常的线程的堆栈里压入 EXCEPTION_POINTERS 结构，EXCEPTION_POINTERS 结构包含两个指针，一个指向 EXCEPTION_RECORD 结构，一个指向 CONTEXT 结构。

```
typedef struct _EXCEPTION_POINTERS {
  +0  pEXCEPTION_RECORD ExceptionRecord  DWORD  ?  //指向 EXCEPTION_RECORD 结构
  +4  pCONTEXT ContextRecord             DWORD  ?  //指向 CONTEXT 结构
}
EXCEPTION_POINTERS ends
```

上例跟踪到 00401017 地址处，按 F7 键或 Shift+F7 键就能进入异常刚发生时的系统代码处，EXCEPTION_POINTERS 结构就在栈顶，即 ESP -> ptEXCEPTION_POINTERS。当前堆栈的数据就是 EXCEPTION_POINTERS 结构的两个数据成员 EXCEPTION_RECORD 和 CONTEXT，如图 11.5 所示。

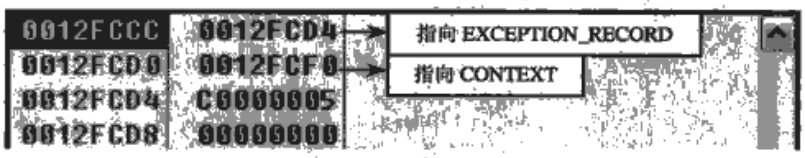


图 11.5　EXCEPTION_POINTERS 结构

1. EXCEPTION_RECORD 结构

EXCEPTION_RECORD 结构包含有关最近发生异常的详细信息，这些信息独立于 CPU。其结构如下：

```
EXCEPTION_RECORD STRUCT {
   +0  DWORD    ExceptionCode       ;异常代码
   +4  DWORD    ExceptionFlags      ;异常标志
   +8  struct   EXCEPTION_RECORD    ;指向另外一个 EXCEPTION_RECORD 的指针
   +C  PVOID    ExceptionAddress    ;异常发生的地址
   +10 DWORD    NumberParameters    ;下面 ExceptionInformation 所含有的 dword 数目
   +14 ULONG_PTR    ExceptionInformation [EXCEPTION_MAXIMUM_PARAMETERS]
} EXCEPTION_RECORD ENDS
```

其中 ExceptionCode 字段定义了产生异常的原因，表 11-2 列出了一些常见的异常原因。当然也可以定义自己的 ExceptionCode。

表 11-2　常见的异常原因代码列表

异常原因	对 应 值	说　明
STATUS_GUARD_PAGE_VIOLATION	080000001h	读写属性为 PAGE_GUARD 的页面
EXCEPTION_BREAKPOINT	080000003h	断点异常
EXCEPTION_SINGLE_STEP	080000004h	单步中断
EXCEPTION_INVALID_HANDLE	0C0000008h	向一个函数传递了一个无效句柄
EXCEPTION_ACCESS_VIOLATION	0C0000005h	读写内存冲突
EXCEPTION_ILLEGAL_INSTRUCTION	0C000001Dh	遇到无效指令
EXCEPTION_IN_PAGE_ERROR	0C0000006h	存取不存在的页面
EXCEPTION_INT_DIVIDE_BY_ZERO	0C0000094h	除零错误
EXCEPTION_STACK_OVERFLOW	0C00000FDh	堆栈溢出

上例中的“mov eax, [esi]”指令读取 0 地址，所以会引发一个 EXCEPTION_ACCESS_ VIOLATION 异常，异常代码是 0C0000005h。

在图 11.5 所示处，查看 OllyDbg 数据窗口 12FCD4 处的数据，即获得 EXCEPTION_RECORD 结构的数据成员（见图 11.6），显示异常代码是 0C0000005h，异常发生地址是 401017h。为了直观显示，也可在 OllyDbg 数据窗口执行右键菜单中的“long/Hex”（长型/十六进制）将数据切换成 DWORD 显示。

图 11.6　EXCEPTION_RECORD 结构

2. CONTEXT 结构

CONTEXT 结构是 Win32 API 一个几乎唯一与处理器相关的结构，包括了线程运行时处理器各主要寄存器的完整镜像，用于保存线程运行时环境。目前，已经存在为 Intel、MIPS、Alpha 和 PowerPC 处理器定义的 CONTEXT 结构。若要了解这些结构，可参考 VC 的头文件 WinNT.h。

x86 CPU 的 CONTEXT 结构如下：

```
MAXIMUM_SUPPORTED_EXTENSION equ 512

CONTEXT STRUCT
    ContextFlags                    DWORD     ?                    +00
    iDr0                            DWORD     ?                    +04
    iDr1                            DWORD     ?                    +08
    iDr2                            DWORD     ?                    +0C
    iDr3                            DWORD     ?                    +10
    iDr6                            DWORD     ?                    +14
    iDr7                            DWORD     ?                    +18
    FloatSave     FLOATING_SAVE_AREA  ;浮点寄存器区+1C~+88
    regGs                           DWORD     ?                    +8C
    regFs                           DWORD     ?                    +90
    regEs                           DWORD     ?                    +94
    regDs                           DWORD     ?                    +98
    regEdi                          DWORD     ?                    +9C
    regEsi                          DWORD     ?                    +A0
    regEbx                          DWORD     ?                    +A4
    regEdx                          DWORD     ?                    +A8
    regEcx                          DWORD     ?                    +AC
    regEax                          DWORD     ?                    +B0
    regEbp                          DWORD     ?                    +B4
    regEip                          DWORD     ?                    +B8
    regCs                           DWORD     ?                    +BC
    regFlag                         DWORD     ?                    +C0
    regEsp                          DWORD     ?                    +C4
    regSs                           DWORD     ?                    +C8
    ExtendedRegisters db MAXIMUM_SUPPORTED_EXTENSION dup(?)
CONTEXT ENDS
```

该结构的大部分域是不言自明的，值得解释的是其第一个域 ContextFlags，它表示该结构中的哪些域有效，当需要在用 CONTEXT 结构保存的信息恢复执行时可对应更新，这为有选择地更新部分而非全部提供了手段。

以下是摘自 Windows.inc 中的定义：

```
CONTEXT_i386                    EQU 000010000H
CONTEXT_i486                    EQU 000010000H
CONTEXT_CONTROL                 EQU CONTEXT_i386+00000001H  ;控制寄存器
CONTEXT_INTEGER                 EQU CONTEXT_i386+00000002H  ;（整数）通用寄存器
CONTEXT_SEGMENTS                EQU CONTEXT_i386+00000004H  ;段寄存器
CONTEXT_FLOATING_POINT          EQU CONTEXT_i386+00000008H  ;浮点寄存器
CONTEXT_DEBUG_REGISTERS         EQU CONTEXT_i386+00000010H  ;调试寄存器
```

上述结构就是在处理 SEH 时用到的大部分结构，通常作为参数传递给异常处理回调函数，可以通过这些结构了解相关信息或改变线程的状态。

在图 11.5 所示处，查看 OllyDbg 数据窗口 12FCF0 处的数据，即获得 CONTEXT 结构的数据成员（见图 11.7），该结构储存的是图 11.4 所示刚触发异常时各种寄存器信息。

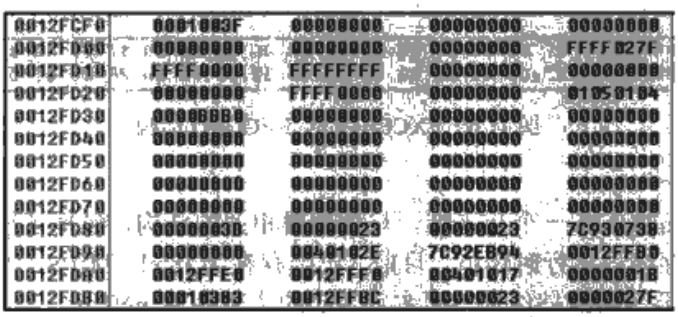


图 11.7　CONTEXT 结构

例如，“D 12FCF0+0B8”显示刚发生异常时的地址（EIP）为 401017。

CONTEXT 结构非常重要，程序通过修改 CONTEXT 结构中的成员，可以干出许多“出格”的事情，以达到反跟踪的目的。

11.4 异常处理回调函数

SEH 异常处理回调函数的参数定义如下：

```
_Handlerproc(
    _lpExceptionRecord,         // 指向一个 EXCEPTION_RECORD 结构
    _lpSEH,                     // 指向 EXCEPTION_REGISTRATION 结构的地址
    _lpContext,                 // 指向 CONTEXT 结构
    _lpDispatcherContext
    )
```

返回值：

ExceptionContinueExection（等于 0）：回调函数返回后，系统将线程环境设置为_lpContext 参数指定的 CONTEXT 结构并继续执行；

ExceptionContinueSearch（等于 1）：回调函数拒绝处理这个异常，系统将通过 EXCEPTION_REGISTRATION 结构的 prev 字段得到前一个回调函数的地址并调用它；

ExceptionNestedException（等于 2）：回调函数在执行中又发生了新的异常，即嵌套异常；

ExceptionCollidedUnwind （等于 3）：发生了嵌套的展开操作。

CONTEXT 结构成员可以改变，这意味着程序可以用改变的 CONTEXT 内容去执行程序，如清除断点，改变代码运行路线等。

例如，本书实例中 seh2.exe 实例的部分代码如下：

```
.code
_start:
;-------------------------------------------------
; 在堆栈中构造一个 EXCEPTION_REGISTRATION 结构
        push  offset perThread_Handler
        push  fs:[0]
        mov   fs:[0],esp
;-------------------------------------------------
; 引发异常的指令
        mov  esi,0
        mov  eax,[esi]          ; 读 0 地址的内存异常
 WouldBeOmit:
        invoke  MessageBox,0,addr Text,addr Caption,MB_OK  ; 这一句永远无法被执行
;-------------------------------------------------
; 异常处理完毕后，从这里开始执行
ExecuteHere:
        invoke   MessageBox,0,addr TextSEH,addr Caption,MB_OK
;-------------------------------------------------
; 恢复原来的 SEH 链
        pop     fs:[0]
        add     esp,4
        invoke  ExitProcess,NULL
;-------------------------------------------------
; 异常回调处理函数
perThread_Handler proc uses ebx pExcept:DWORD,pFrame:DWORD,pContext:DWORD,pDispatch:DWORD
        mov     eax,pContext
        Assume  eax:ptr CONTEXT
        lea     ebx, ExecuteHere   ; 异常后准备从 ExecuteHere 后开始执行
        mov     [eax].regEip,ebx   ; 修改 CONTEXT.EIP，准备改变代码运行路线
        xor     ebx,ebx
        mov     [eax].iDr0,ebx     ; 对 Drx 调试寄存器清零，使断点失效（反跟踪）
        mov     [eax].iDr1,ebx
        mov     [eax].iDr2,ebx
        mov     [eax].iDr3,ebx
        mov     [eax].iDr7,341
        mov     eax,0              ; 返回值=ExceptionContinueExecution，表示已经修复
        ret
perThread_Handler endp
end _start
```

该实例通过修改 CONTEXT 结构中的成员，将调试寄存器 Drx 清零，使断点失效，以达到反跟踪的目的。

跟踪时可按上一节的方法从 ERR 结构入手，操作过程如下。

① 执行到 00401018 一行，记下堆栈中的 handler 值 00401051。

② 按 F7 键进入系统代码，查看 CONTEXT.EIP 的值，命令行为“D [esp+4]+0B8”，因为软件保护中一般会通过 CONTEXT.EIP 的值改变程序流程。

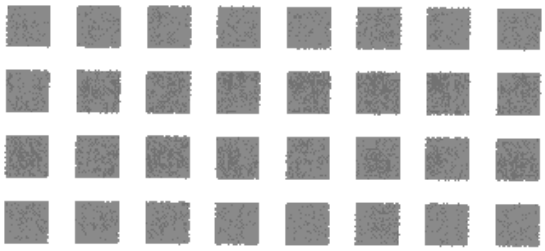
③ 一步步执行，直到 CONTEXT.EIP 值改变。40105E 这行命令“mov dword ptr [eax+0B8], ebx”就是对 CONTEXT.EIP 赋值。

④ 程序处理完后，将跳到新的 CONTEXT.EIP（0040102D）代码处执行。

⑤ 程序在处理 CONTEXT 结构的同时，将 Dr0，Dr1，Dr2 和 Dr3 调试寄存器清零，使调试器的断点失效等，以达到反跟踪的目的。

跟踪时的具体汇编代码如下：

```
00401000   push   00401051                  ;Handler,发生异常后到这里执行
00401005   push   dword ptr fs:[0]
0040100C   mov    dword ptr fs:[0], esp     ;构造1个ERR结构
00401013   mov    esi, 0
00401018   mov    eax, dword ptr [esi]      ;读取线性地址0，产生异常
0040101A   push   0                         ;/Style = MB_OK|MB_APPLMODAL
0040101C   push   00403000                  ;|Title = "SEH"
00401021   push   0040300F                  ;|Text = "SEH程序?,BB,"有运行"
00401026   push   0                         ;|hOwner = NULL
00401028   call   <jmp.&USER32.MessageBoxA> ;\MessageBoxA
0040102D   push   0                         ;/Style = MB_OK|MB_APPLMODAL
0040102F   push   00403000                  ;|Title = "SEH"
00401034   push   00403004                  ;|Text = "Hello,SEH!"
00401039   push   0                         ;|hOwner = NULL
0040103B   call   <jmp.&USER32.MessageBoxA> ;\MessageBoxA
00401040   pop    dword ptr fs:[0]
00401047   add    esp, 4
0040104A   push   0
0040104C   call   <jmp.&KERNEL32.ExitProcess>
00401051   push   ebp                       ;结构异常处理程序
00401052   mov    ebp, esp
00401054   push   ebx
00401055   mov    eax, dword ptr [ebp+10]   ;EAX此时即为 CONTEXT 的指针
00401058   lea    ebx, dword ptr [40102D]
0040105E   mov    dword ptr [eax+B8], ebx   ;修改CONTEXT.EIP，希望到这里执行
                                            ;EIP为40102D，对此地址设置断点
00401064   xor    ebx, ebx
00401066   mov    dword ptr [eax+4], ebx    ;CONTEXT.Dr0=0，使断点失效
00401069   mov    dword ptr [eax+8], ebx    ;CONTEXT.Dr1=0，使断点失效
0040106C   mov    dword ptr [eax+C], ebx    ;CONTEXT.Dr2=0，使断点失效
0040106F   mov    dword ptr [eax+10], ebx   ;CONTEXT.Dr3=0，使断点失效
00401072   mov    dword ptr [eax+18], 155   ;修改 CONTEXT.Dr7 = 0x155
00401079   mov    eax, 0                    ;0表示已经修复，可从异常处继续执行
0040107E   pop    ebx
0040107F   leave
00401080   retn   10                        ;返回系统代码里，跟入将迷失在系统代码里
                                            ;系统处理完，将从CONTEXT.EIP开始执行
```



第 6 篇　脱壳篇

现在越来越多的软件都采用了加壳保护。在软件分析和汉化过程中，脱壳是必不可少的一步，本章详细介绍了基本的脱壳技术，完全为脱壳新手量身定做。

第12章 专用加密软件

软件加密的资料相对来说比较匮乏，它是一种对抗性的技术，需要开发者对解密技术有一定了解。对于大多数开发者来说，由于不熟悉加密与解密这个领域，导致花费了大量人力和物力设计的保护方案不堪一击。术业有专攻，为了让软件开发者从软件保护措施中脱离出来，专心致力于自己的软件开发，出现了一个新的事物——专用加密软件。

12.1 认识壳

壳[①]是最早出现的一种专用加密软件技术，现在越来越多的软件都加壳保护，那到底什么是壳呢？

12.1.1 壳的概念

在自然界中，植物用壳来保护种子，动物用壳来保护身体等。同样，在一些计算机软件里也有一段专门负责保护软件不被非法修改或反编译的程序。它们附加在原程序上通过 Windows 加载器载入内存后，先于原始程序执行，得到控制权，执行过程中对原始程序进行解密、还原，还原完成后再把控制权交还给原始程序，执行原来的代码的部分。加上外壳后，原始程序代码在磁盘文件中一般是以加密后的形式存在的，只在执行时在内存中还原，这样就可以比较有效地防止破解者对程序文件的非法修改，同时也可防止程序被静态反编译。由于这段程序和自然界的壳在功能上有很多相同的地方，基于命名的规则，就把这样的程序称为“壳”了，如图 12.1 所示。

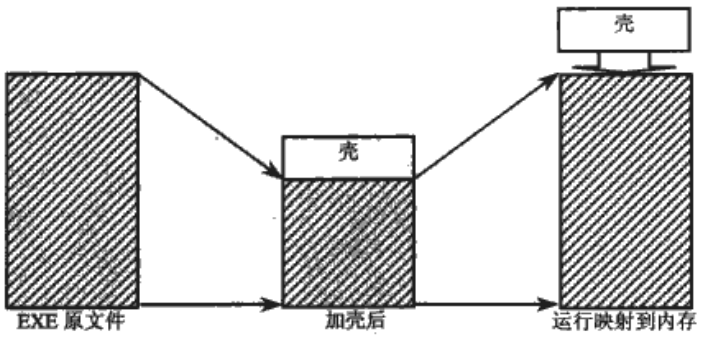


图 12.1 描述壳的示意图

[①] 音有 ké 和 qiào，这里一般发音 ké。

最早提出“壳”这个概念的，是当年推出脱壳软件 RCOPY 3 的作者熊焰先生。在 DOS 时代，“壳”一般都是指磁盘加密软件的段加密程序，可能是那时侯的加密软件还刚起步不久，所以大多数的加密软件（加壳软件）所生成的“成品”在“壳”和需要加密的程序之间总有一条比较明显的“分界线”。有经验的人可以在跟踪软件的运行以后找出这条分界线来。

脱壳技术的进步，促进且推动了当时的加壳技术的发展。LOCK95 和 BITLOK 等所谓的“壳中带籽”加密程序纷纷出笼，真是各出奇谋，把小小的软盘也折腾得够辛苦的了。国内的加壳软件和脱壳软件正在较量得火红的时候，国外的“壳”类软件早已经发展到像 LZEXE 之类的压缩壳了。这类软件其实就是一个标准的加壳软件，它把 EXE 文件压缩了以后，再在文件上加上一层在软件执行时自动把文件解压缩的“壳”来达到压缩 EXE 文件的目的。接着，这类软件越来越多，比如 PKEXE、AINEXE、UCEXE 和 WWPACK 都属于这类软件。奇怪的是，当时看不到一个国产的同类软件。

过了一段时间，可能是国外淘汰了磁盘加密，转向使用软件序列号加密方法，保护 EXE 文件不被动态跟踪和静态反编译就显得非常重要了。所以专门实现这样功能的加壳程序便诞生了。MESS、CRACKSTOP、HACKSTOP、TRAP、UPS 等都是比较有名气的本类软件代表。这样的软件才能算是正宗的加壳软件。

由于 Microsoft 保留了 Windows 95 的很多技术上的秘密，即便 Windows 95 已经推出三年多的时间，也没见过在其上面运行的“壳”类软件。直到 1998 年的中期，这样的软件才迟迟出现，而这个时候 Windows 98 也发表了一段日子。这类的软件不发表尚可，一发表就大批地涌现出来。先是加壳类的软件，如 BJFNT、PELOCKNT 等，它们的出现，使暴露三年多的 Windows 下的 PE 格式 EXE 文件得到很好的保护。接着出现的就是压缩壳（Packers），因为 Windows 下运行的 EXE 文件“体积”一般都比较大，所以它的实用价值比起 DOS 下的压缩软件要大很多。这类软件也很多，UPX、ASPack、PECompact 等都是其中的佼佼者。随着软件保护的需要，出现了加密壳（Protectors），它用上了各种反跟踪技术保护程序不被调试、脱壳等，其加壳后的体积大小不是其考虑的主要因素，如 ASProtect、Armadillo、EXECryptor 等。随着加壳技术的发展，这两类软件之间的界线越来越模糊，很多加壳软件除具有较强的压缩性能，同时也有了较强的保护性能。

加壳软件一般都有良好的操作界面，使用也比较简单。除了一些商业壳，还有一些个人开发的壳，种类较多。壳对软件提供了良好保护的同时，也带来了兼容性的问题，选择一款壳保护软件后，要在不同硬件和系统上多测试。由于壳能保护自身代码，因此许多木马或病毒都喜欢用壳来保护和隐藏自己。对于一些流行的壳，杀毒引擎能对目标软件脱壳，再进行病毒检查。而大多数私人壳，杀毒软件不会专门开发解压引擎，而是直接把壳当成木马或病毒处理。

有加壳就一定会有脱壳。一般的脱壳软件多是专门针对某加壳软件而编的，虽然针对性强、效果好，但收集麻烦。因此掌握手动脱壳技术十分必要。

12.1.2　压缩引擎

一些加壳软件能将文件压缩，大多数情况下，压缩算法是调用现成的压缩引擎。目前压缩引擎种类比较多，不同的压缩引擎有不同特点，如一些对图像压缩效果好，一些对数据压缩效果好。而加壳软件选择压缩引擎有一个特点，在保证压缩比的条件下，压缩速度慢些关系不是太大，但解压速度一定要快，这样加了壳的 EXE 文件运行起来速度才不会受太大的影响。例如下面几个压缩引擎就能满足这样要求：aPLib、JCALG1、LZMA。

12.2　压缩壳

不同的外壳所侧重的方面也不一样，有的侧重于压缩，有的则侧重于加密。压缩壳的特点就是减小软件体积大小，加密保护不是其重点。目前兼容性和稳定性比较好的压缩壳有 UPX、ASPack、PECompact 等。

12.2.1 UPX

UPX 是一个以命令行方式操作的可执行文件免费压缩程序，兼容性和稳定性很好。UPX 包含 DOS、Linux 和 Windows 等版本，并且开源。官方主页：http://upx.sourceforge.net。

UPX 的命令格式为：upx [-123456789dlthVL] [-qvfk] [-o file] file..

UPX 早期版本压缩引擎是自己实现的，3.x 版本也支持 LZMA 第三方压缩引擎。UPX 除了对目标程序进行压缩外，也可解压缩。UPX 的开发近乎完美，它不包含任何反调试或保护策略。另外，UPX 保护工具 UPXPR、UPX-Scrambler 等可修改 UPX 加壳标志，使 UPX 自解压缩功能失效。

12.2.2 ASPack

ASPack 是一款 Win32 可执行文件压缩软件，可压缩 Win32 位可执行文件 EXE、DLL、OCX，具有很好的兼容性和稳定性。官方主页：http://www.aspack.com。

双击 ASPack 运行软件，出现了程序界面（见图 12.2）。如果压缩过程中出错，请将选项中的“压缩资源”去除选中。其他加壳软件也会出现类似问题，解决办法一样。

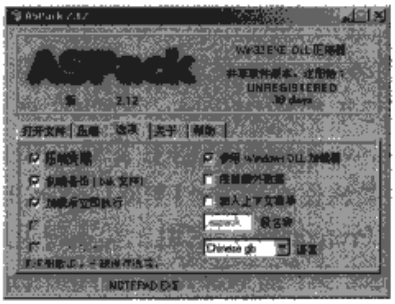


图 12.2 ASPack 程序界面

12.3 加密壳

加密壳种类比较多，不同的壳侧重点不同，一些壳单纯保护程序，另一些壳还提供额外的功能，如提供注册机制、使用次数、时间限制等。加密壳还有一个特点，越是有名的加密壳，研究的人也越多，其被脱壳或破解的可能性也越大，所以不要太依赖壳的保护。加密壳在强度与兼容性上做得好的并不多，这里向大家介绍几款常见的。

12.3.1 ASProtect

ASProtect 是一款非常强大的 Win32 位保护工具，这款壳开创了壳的新时代。它拥有压缩、加密、反跟踪代码、CRC 校验和花指令等保护措施。它使用 Blowfish、Twofish、TEA 等强劲的加密算法，还用 RSA1024 作为注册密钥生成器。它还通过 API 钩子与加壳的程序进行通信，并且 ASProtect 为软件开发人员提供 SDK，实现加密程序内外结合。SDK 支持 VC、VB、Delphi 等。

ASProtect 创建者是俄国人 Alexey Solodovnikov，其将 ASPack 中的一些开发经验运用到 ASProtect，该壳的编写简单而精巧，是款经典之作。ASProtect 很注重于兼容性和稳定性，因此没有采用过多的反调试策略。

ASProtect 目前有两个系列，一个系列是 ASProtect 1.3x，另一个系列是 ASProtect SKE 2.x。ASProtect SKE 系列已采用了部分虚拟机技术，主要是在 Protect Original EntryPoint 与 SDK 上。保护过程中建议大量使用 SDK，SDK 使用请参考其帮助文档及安装目录下的样例，在使用时注意 SDK 不要嵌套，并且同一组标签用在同一个子程序段里。

运行 ASProtect，主界面如图 12.3 所示。首先单击“File To Protect”的 按钮选择一个需要保护的文件，在“Output To FileName”中填上输出的文件名。再单击“Modes”标签，单击“Add Mode”按钮，将“Is this Mode Avtive”选上。最后，单击“Protection”标签，对软件进行保护即可。

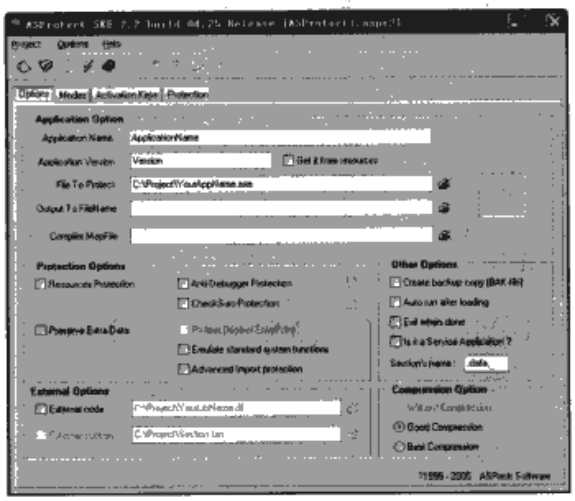


图 12.3　ASProtect 主界面

ASProtect 加壳过程中也可挂接用户自己写的 DLL 文件，方法是在图 12.3 中的“External Options”选项中加上目标 DLL 即可。这样，用户可以在 DLL 中加入自己的反跟踪代码，以提高软件的反跟踪能力。

由于 ASProtect 在共享软件里使用得相当广，因此大家研究得也多，其各类保护机制已被研究得很透了，甚至可能都有脱壳机的存在。

12.3.2　Armadillo

Armadillo 也称穿山甲，是一款应用面较广的商业保护软件（其界面如图 12.4 所示）。可以运用各种手段来保护你的软件，同时也可以为软件加上种种限制，包括时间、次数，启动画面等。其官方主页：http://www.siliconrealms.com。Armadillo 对外发行时有 Public、Custom 两个版本。Public 是公开演示的版本，Custom 是注册用户拿到的版本。只有 Custom 才有完整的功能，如强大的 Nanomites 保护。Public 版有功能限制，没什么强度，不建议采用。

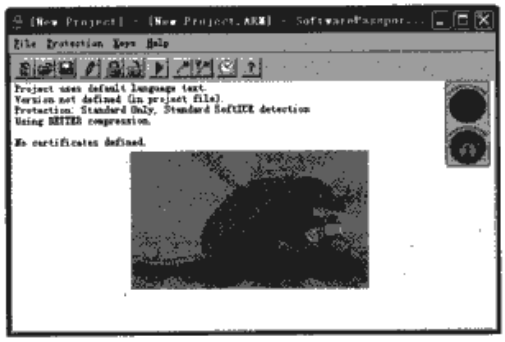


图 12.4　Armadillo 界面

Armadillo 有如下保护功能：Nanomites、Import Table Elimination、Strategic Code Splicing、Memory-Patching Protections 等。其中 Nanomites 功能最为强大，使用时，需要在程序里加入 Nanomites 标签，如下面这段 VC 编译器里定义的标签：

```
#define NANOBEGIN   __asm _emit 0xEB __asm _emit 0x03 __asm _emit 0xD6 __asm _emit 0xD7 __asm _emit 0x01
#define NANOEND     __asm _emit 0xEB __asm _emit 0x03 __asm _emit 0xD6 __asm _emit 0xD7 __asm _emit 0x00
```

用 NANOBEGIN 和 NANOEND 标签将需要保护的代码括住，Armadillo 加壳时，会扫描程序，处理标签里的跳转指令，将所有跳转指令换成 INT 3 指令，其机器码是 CC。此时 Armadillo 运行时，是双进程，子进程遇到 CC 异常，由父进程截获这个 INT 3 异常，计算出跳转指令的目标地址并反馈给子进程，子进程继续运行。由于 INT 3 机器码是 CC，因此也称这种保护是 CC 保护。

12.3.3 EXECryptor

EXECryptor 是一款商业保护软件（其主界面如图 12.5 所示），官方主页：http://www.strongbit.com。其可以为目标软件加上注册机制、时间限制、使用次数等附加功能。这款壳的特点是 Anti-Debug 比较强大，同时做得比较隐蔽，另外就是采用了虚拟机保护一些关键代码。要使这款壳有强大的保护，必须合理使用 SDK 功能，将关键的功能代码用虚拟机保护起来。

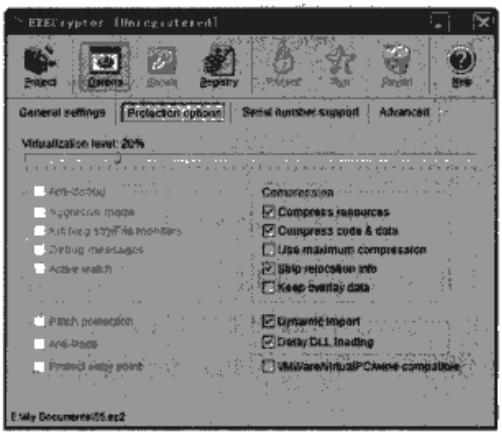


图 12.5 EXECryptor 主界面

12.3.4 Themida

Themida 是 Oreans 的一款商业保护软件（其主界面如图 12.6 所示），官方链接：www.oreans.com。Themida 1.1 以前版本带驱动，稳定性有些影响。Themida 最大特点就是其虚拟机保护技术，因此在程序中擅用 SDK，将关键的代码让 Themida 用虚拟机保护起来。Themida 最大的缺点就是生成的软件有些大。WinLicense 这款壳和 Themida 是同一个公司的一个系列产品，WinLicense 主要多了一个协议，可以设定使用时间、运行次数等功能，两者核心保护是一样的。

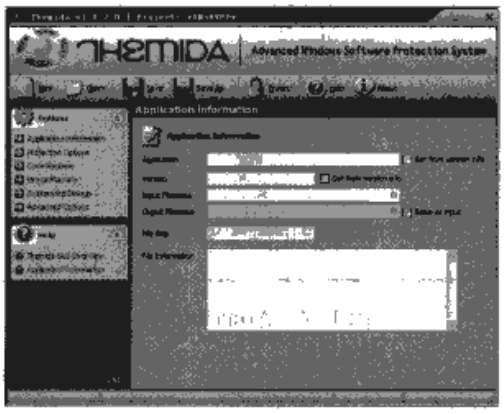


图 12.6 Themida 主界面

12.4 虚拟机保护软件

虚拟机（Virtual Machine，简称 VM），现在这个概念已经越来越广泛了。比如 VMWare 能够在软件环境下模拟出一台单独的机器，VMWare 就是一种虚拟机。许多解释性语言，如 Visual Basic 的 P-code 程序也是一种虚拟机。这里讨论的虚拟机和 VMWare 不同，比较类似于 P-code，它将一系列的指令解释成 bytecode（字节码）放在一个解释引擎中执行，以对软件进行保护。

12.4.1 虚拟机介绍

一个虚拟机引擎由编译器、解释器和 VPU Context（虚拟 CPU 环境）组成，再配上一个或多个指令系统。虚拟机运作的时候，先把已知的 x86 指令根据自定义的指令系统解释成字节码，放在 PE 文件中，然后将原处代码删掉，改成如下类似的代码进入虚拟机执行循环。

```
push bytecode
jmp  VstartVM
```

从这里可看出，虚拟机保护与加壳保护还是不同的。调试者跟踪进入到虚拟机后，是非常难于理解原指令的。就好像把一篇文章从英文翻译到中文，结果发现文章的很多段落是用孟加拉语写的。

如果分析者要跟踪虚拟机内的代码执行，这将是一件非常繁重的工作。要想理解程序流程，就必须对虚拟机引擎进行深入分析。完整地得到原始代码和 P-code 的对应关系，复杂性可想而知了。也就是说，虚拟机加密策略，是建立在提高解密者的分析成本上的一种加密策略。正是由于这个原因，虚拟机已经成为最流行的保护趋势。

虚拟机技术是以效率换安全的，往往一条原始汇编指令经过 VM 处理后会膨胀几十倍甚至几百倍，执行速度会大大降低。正是由于这个原因，VM 保护一般提供 SDK 方式，使用者一般只需要把较为重要的代码用 VM 保护起来，这样在一定程度上确保了程序的执行效率。由于当前 CPU 速度足够快，对于一般程序用虚拟机处理后，不会太多影响程序性能。如果对于一些对速度要求高的代码，就不适合用虚拟机保护了。

现今这一技术逐渐被用于软件保护技术中，如 EXECryptor、Themida、VMProtect 等商业保护软件。前两者是将壳与虚拟机结合起来了，由于设计原因，其虚拟机强度没有 VMProtect 强大，VMProtect 是一款纯虚拟机保护软件。

12.4.2 VMProtect 简介

VMProtect 适合 Visual Basic(native)、Visual C、Delphi、ASM 等本地编译的目标程序，支持 EXE、DLL、SYS。VMProtect 是由俄国的 PolyTech 开发，一个利用伪指令虚拟机的保护软件。VMProtect 并不是一款壳，它将指定的代码变形（Mutation）和虚拟化（Virtualization），处理后能很好地隐藏代码算法，防止算法被逆向。

1. 保护指定代码

VMProtect 可以精确地保护指定地址的代码，用调试器（如 OllyDbg）跟踪目标程序，找到要保留的核心代码，获得相关地址。然后用 VMProtect 打开待保护的文件，单击“Project”菜单下的“New procedure”或者单击工具栏中的 按钮，在出现的对话框中填入这个地址。也可在 Dump 窗口，找到待保护的代码，再执行菜单“New procedure”，如图 12.7 所示。重复这个过程，将其他要保护的地址添加上。每添加一个要保护的地址，VMprotect 会根据代码的执行流程判断最可能结束的地址。如果要指定结束地址，在相应地址上，单击鼠标右键，执行“End of procedure”功能。

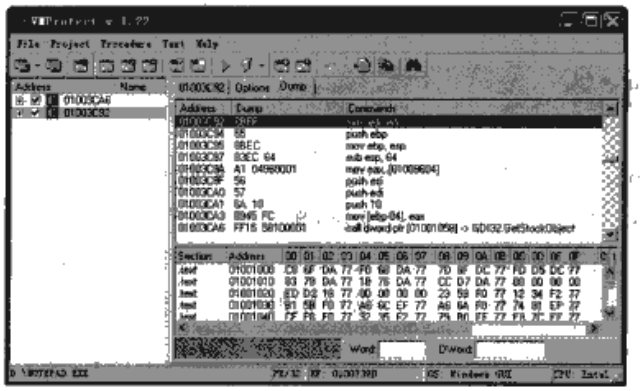


图 12.7　VMProtect 主界面

用调试器获得地址的操作过程比较专业，不适合大众所使用。因此，VMProtect 自 1.2 版本以后，支持 SDK。在编程时，用 VMProtect 提供一对标记将需要保护的代码括住，然后用 VMProtect 打开编译后的 EXE 文件时，VMProtect 就会认出这些标记，并在有标记的地方进行保护。

在 Delphi 程序中的 SDK。标记开始：

```
asm
  db $EB,$10,'VMProtect begin',0
end;
```

标记结束：

```
asm
  db $EB,$0E,'VMProtect end',0
end;
```

在 VB 程序中的标记。标记开始：

```
Call VarPtr("VMProtect begin")
```

标记结束：

```
Call VarPtr("VMProtect end")
```

在 VC 中可以利用_emit 语句插入十六进制字节代码来实现，具体见本书光盘映像文件中样例。

运行 VMProtect，打开带有 SDK 的目标文件，单击工具栏中的 按钮，在弹出的添加地址窗口中会自动将 SDK 定义代码的地址填上，如图 12.8 所示。

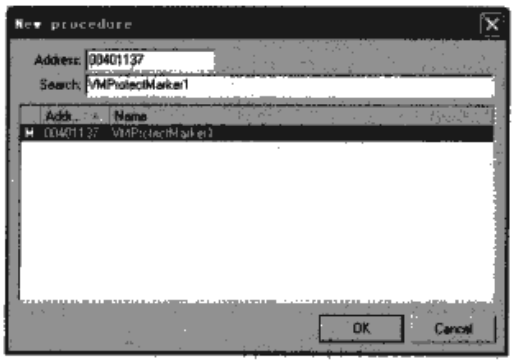


图 12.8　显示 SDK 标签处的代码

2. 保护函数

VMProtect 支持 Map 文件来定位函数，设置编译器，让其生成 Map 文件。这个 Map 文件含有程序中自定义函数和引用外部函数的名称和地址，VMProtect 会用这个 Map 文件来定位函数的地址。将目标文件和 Map 文件放在一起（文件名要相同），用 VMProtect 打开文件后，执行菜单“New procedure”时能够列出很多内部函数，这时只需要选择想加密的函数进行后续处理即可，如图 12.9 所示。如果是目标文件有输出函数，VMProtect 还将列出各输出函数的地址。

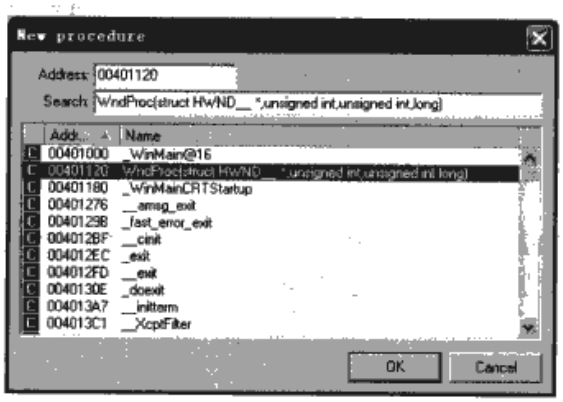


图 12.9　打开有 Map 程序的情况

3. 加密保护

在 VMProtect 里确定了将待加密的地址后，单击“Options”标签，根据需要设置相应的选项，如图 12.10 所示。最后单击菜单“Project/Compilation”（F9），便可对目标软件进行保护。经 VMProtect 处理好的文件，要多测试，并建议用调试器 OllyDbg 再检查一下目标代码。经处理过的软件，可以继续用 ASProtect、Themida 等加壳软件进一步保护，不过笔者认为没有太多的必要了，VMProtect 保护的代码安全性已经很好了。

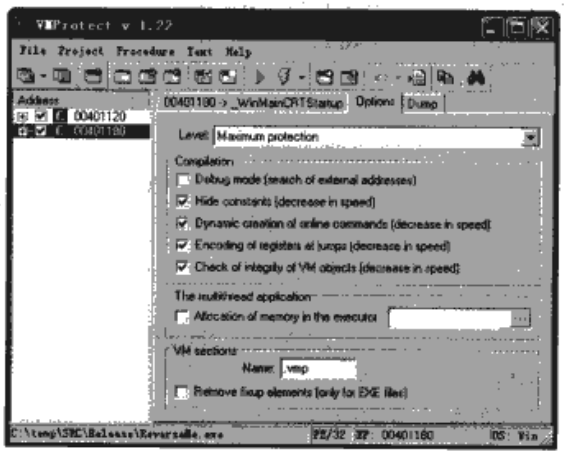


图 12.10　设置 VMProtect 的选项

VMProtect 低版本对多线程程序处理不是太好，因此不要同时处理多个不同线程的代码。处理时，可以先处理一个线程内的代码，编译后，再用 VMProtect 处理另一个线程内的代码。测试时，可以到超线程的 CPU 或双 CPU 硬件上测试。VMProtect 高版本在这方面有所完善。

VMProtect 是当前最强的虚拟机保护软件之一，经过 VMProtect 处理的软件大大增加了逆向人员的分析困难性，关键是用好。另外，经虚拟机处理代码效率会降低，因此一些对效率要求比较高的代码就不适合用 VMProtect 进行处理。

第13章 脱壳技术

任何事物都有两面性，有加壳，就必有脱壳。加壳与脱壳有着紧密的联系，一些脱壳技术是针对加壳而产生的，脱壳的进步，又迫使加壳软件不断创新发展。现在越来越多的软件都加壳保护，脱壳有时是分析一个软件不可缺少的步骤。

13.1 基础知识

现阶段加壳软件种类比较多，各种反跟踪技术和保护技术都被应用上了，如要脱壳，必须对壳的一些共性原理有所掌握。

13.1.1 壳的加载过程

壳和病毒在某些方面比较类似，都需要比原程序代码更早地获得控制权。壳修改了原程序的执行文件的组织结构，从而能够比原程序的代码提前获得控制权，并且不会影响原程序的正常运行。这里简单说说壳的常见加载过程。

1. 保存入口参数

加壳程序初始化时保存各寄存器的值，外壳执行完毕，再恢复各寄存器内容，最后再跳到原程序执行。通常用 pushad/popad、pushfd/popfd 指令对来保存与恢复现场环境。

2. 获取壳自己所需要使用的 API 地址

一般外壳的输入表中只有 GetProcAddress、GetModuleHandle 和 LoadLibrary 这几个 API 函数，甚至只有 Kernel32.dll 以及 GetProcAddress。如果需要其他的 API 函数，则通过 LoadLibraryA(W) 或 LoadLibraryExA(W)将 DLL 文件映像映射到调用进程的地址空间中，函数返回的 HINSTANCE 值用于标识文件映像映射到的虚拟内存地址。

LoadLibrary 函数的原型如下：

```
HINSTANCE LoadLibrary(
    LPCTSTR lpLibFileName          // DLL 文件名的地址
  );
返回值：成功返回模块的句柄，失败返回 NULL。
```

如果 DLL 文件已被映射到调用进程的地址空间里，可以调用 GetModuleHandleA(W)函数获得 DLL 模块句柄。函数的原型如下：

```
HMODULE GetModuleHandle(
```

```
    LPCTSTR lpModuleName      // DLL 文件名地址
);
```

一旦 DLL 模块被加载，线程就可以调用 GetProcAddress 函数获取输入函数的地址。函数原型如下：

```
FARPROC GetProcAddress(
    HMODULE hModule,          // DLL 模块句柄
    LPCSTR lpProcName         // 函数名
);
```

参数 hModule 是调用 LoadLibrary(Ex)或 GetModuleHandle 函数的返回值。参数 lpProcName 可以采用两种形式：第一种是以 0 结尾的字符串地址；第二种形式是调用地址的符号的序号（微软公司非常反对使用序号）。

读者必须熟练掌握这三个函数的用法，外壳中用到其他函数就是用这三个函数来调用的。现在有些壳，为了提高强度，甚至连系统提供的 GetProcAddress 函数都不用，而是自己写个相同功能的函数代替 GetProcAddress，以提高函数调用的隐蔽性。

3. 解密原程序的各个区块的数据

壳出于保护原程序代码和数据的目的，一般都会加密原程序文件的各个区块。在程序执行时外壳将会对这些区块数据解密，以让程序能正常运行。壳一般是按区块加密的，那么在解密时也按区块解密，并且把解密的区块数据按照区块的定义放在合适的内存位置。

4. IAT 的初始化

IAT 的填写，本来应该由 PE 装载器实现。但由于加壳时，自己构造了一个输入表，并让 PE 头中的输入表指针指向了自建的输入表。所以，PE 装载器就将对自建的输入表进行了填写。那么原来 PE 的输入表的填写，只好由外壳程序实现了。外壳要做的就是将这个新输入表结构从头到尾扫描一遍，对每一个 DLL 引入的所有函数重新获取地址，并填写在 IAT 表中。

5. 重定位项的处理

文件执行时将被映像到指定内存地址中，这个初始内存地址称为基址。当然这只是程序文件中声明的，程序运行时能够保证系统一定满足其要求吗？

对于 EXE 的程序文件来说，Windows 系统会尽量满足。例如某 EXE 文件的基地址为 400000h，而运行时 Windows 系统提供给程序的基地址也同样是 400000h。在这种情况下就不需要进行地址“重定位”了。由于不需要对 EXE 文件进行“重定位”，所以加壳软件把原程序文件中用于保存重定位信息的区块干脆也删除了，这样使得加壳后的文件更加小巧。有些工具提供“Wipe Reloc”的功能，其实就是这个作用。

不过对于 DLL 的动态链接库文件来说，Windows 系统没有办法保证每一次 DLL 运行时提供相同的基地址。这样“重定位”就很重要了，此时壳中也需要提供进行“重定位”的代码，否则原程序中的代码是无法正常运行起来的。从这点来说，加壳的 DLL 比加壳的 EXE 修正时多了一个重定位表。

6. HOOK-API

程序文件中的输入表的作用是让 Windows 系统在程序运行时提供 API 的实际地址给程序使用。在程序的第一行代码执行之前，Windows 系统就完成了这个工作。

壳一般都修改了原程序文件的输入表，然后自己模仿 Windows 系统的工作来填充输入表中相关的数据。在填充过程中，外壳就可填充 HOOK-API 的代码的地址，这样就可间接地获得程序的控制权。

7. 跳转到程序原入口点（OEP）

从这个时候起壳就把控制权交还给原程序了，一般的壳在这里会有明显的一个“分界线”。当然现在越来越多的加密壳将 OEP 一段代码搬到外壳的地址空间里，然后将这段代码清除掉。这种技术称为 Stolen

Bytes。这样，OEP 与外壳间就没那条明显的分界线了，增加了脱壳难度。

13.1.2 脱壳机

针对特定的壳，开发出来的脱壳软件，称为脱壳机。脱壳就是将加壳后的程序恢复原来状态，脱壳成功的标志是文件能正常运行。由于脱壳有时没有将壳本身的代码去除，脱壳后的程序一般会比原始程序大。

脱壳机一般分为专用脱壳机和通用脱壳机。专用脱壳机是针对某种壳专门编写的，只能脱特定的壳，使用范围小，但效果好。通用脱壳机则具有通用性，可以脱多种不同类型的壳，主要是压缩壳，例如 Quick Unpack、File Scanner 等。

分析一个软件前，用 PEiD 确定一下壳的种类，然后选用合适的脱壳机。脱壳机使用都比较简单，本节就不介绍了。

13.1.3 手动脱壳

对于一些加密壳或修改的壳，没有脱壳机，此时必须要分析外壳，手动脱壳了。手动脱壳可以加深对 PE 格式的理解，并能从一些外壳里吸取先进的加密技术。手动脱壳过程一般分为三步：一是查找程序的真正入口点；二是抓取内存映像文件；三是 PE 文件重建。

手动脱壳原理请参考 13.1.1 节“壳的加载过程”。程序执行时，外壳代码首先获得控制权，模拟 Windows 加载器，将原来的程序恢复到内存中。这时，内存中的数据就是加壳前的镜像文件了，适机抓取出来，再修正一下即可还原到加壳前的状态。

13.2 寻找 OEP

外壳保护的程序运行时，首先执行外壳程序，外壳程序负责把用户原来的程序在内存中解压还原，并把控制权交还给解开后的真正程序，再跳到原来程序的入口点，一般的壳在这里会有明显的一个“分界线”，这个解压后程序真正的入口点称为 OEP，即 Original Entry Point。

13.2.1 根据跨段指令寻找 OEP

决大多数 PE 加壳程序在被加密的程序中加上一个或多个区块，当外壳代码处理完毕后，会跳到程序本身的代码上，所以依据跨段的转移指令就可找到真正的入口点。

用第 16 章编写的加壳工具对实例进行加壳处理，加壳后的程序称为 RebPE。本节用这个实例来演示如何依据跨段指令寻找 OEP。

为了学习方便，先查看一下加壳前程序的一些信息。用 LordPE 打开加壳前的实例，获得其入口点 RVA 为 1130h。再查看区块，如图 13.1 所示。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	000036DE	00001000	00004000	60000020
.rdata	00005000	0000084E	00005000	00001000	40000040
.data	00006000	000029FC	00006000	00003000	C0000040
.rsrc	00009000	00009A70	00009000	0000A000	40000040

图 13.1 查看加壳前的区块

加壳后 RebPE 的入口点 RVA 为 13000h。再查看区块，如图 13.2 所示。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00004000	00000400	00002400	E0000020
.rdata	00005000	00001000	00002800	00000200	C0000040
.data	00006000	00003000	00002A00	00000200	C0000040
.rsrc	00009000	0000A000	00002C00	00001200	C0000040
.pediy	00013000	00007000	00003E00	00006A00	E0000040

图 13.2　查看加壳后的区块

加壳后，RebPE 多了一个区块.pediy，这个区块就是外壳部分，其相当于一个文件加载器（Loader）。RebPE 运行时，各区块被 Windows 系统映射到内存中，现在的入口点是 13000h，指向外壳部分。外壳拿到控制权后，通过各种方式获得自己所需要的 API 地址，解密原程序各区块的数据，填充 IAT，做完这些工作后，就准备跳到原程序入口点（OEP），即 401130h 处执行，如图 13.3 所示。

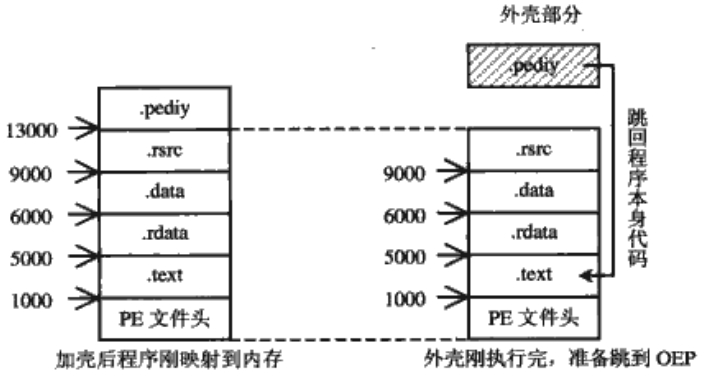


图 13.3　外壳加载运行

运行 OllyDbg 后，在调试选项里，将暂停点设置在主模块的入口点（Entry point of main module）。用 OllyDbg 打开加壳后的实例 RebPE，可能会提示所加载的程序入口点超出代码范围，如图 13.4 所示。这是因为现在入口点指向外壳部分，即区块.pediy 部分，不是常见的代码段（本例是.text 区块），所以 OllyDbg 会给出提示。可以在选项里将这个提示关闭。

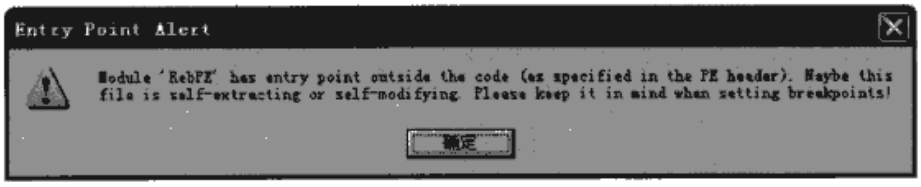


图 13.4　提示入口点超出代码范围

OllyDbg 暂停后，加壳程序的入口点代码如下（对各句汇编代码的理解，可以参考第 16 章中的 16.3.4 节“外壳引导段”）：

```
00413000  pushad                          ;保存现场环境
00413001  call    004130C8                ;这种取地址的做法在外壳、病毒中使用非常普遍
                                          ;按 F8 键可能会跑“飞”，因此这步请按 F7 键
......
004130C8  pop     ebp
004130C9  sub     ebp, 6                  ;取得外壳入口点地址
......
004130FF  call    dword ptr [ebp+2E]      ;GetProcAddress(eax,"VirtualAlloc")
00413102  mov     dword ptr [ebp+B8], eax
00413108  push    4
```

```
0041310A  push    1000
0041310F  push    dword ptr [ebp+8F]
00413115  push    0
00413117  call    dword ptr [ebp+B8]   ;调用 VirtualAlloc 分配外壳第二段所需内存
0041311D  push    eax
0041311E  mov     dword ptr [ebp+C4], eax
00413124  mov     ebx, dword ptr [ebp+8B]
0041312A  add     ebx, ebp
0041312C  push    eax
0041312D  push    ebx
0041312E  call    00413137            ;调用 Aplib 解压函数 aP_depack_asm
00413133  pop     edx
00413134  push    ebp
00413135  jmp     edx                 ;转到外壳的第二段继续执行
```

第二段的主要工作是还原各区块数据，初始化原程序，如填充 IAT 表。由于外壳的第二段空间是调用 VirtualAlloc 函数随机分配的，因此读者系统显示的地址和笔者可能会不同，阅读时，请以汇编代码来参考。

```
00370000  call    00370005
00370005  pop     edx
00370006  sub     edx, 5                    ;取外壳第二段的基址，本例是 370000
00370009  pop     ebp
0037000A  mov     eax, dword ptr [edx+359]
00370010  or      eax, eax
00370012  je      short 0037001A            ;用来处理 DLL 文件
……
;************* 解压缩各区块***************
003700A0  mov     ebx, 2A1
003700A5  cmp     dword ptr [ebx+ebp], 0
003700A9  je      short 003700F2
003700AB  push    ebx
003700AC  push    4
003700AE  push    1000
003700B3  push    dword ptr [ebx+ebp]
003700B6  push    0
003700B8  call    dword ptr [ebp+34D]       ;VirtualAlloc 申请内存进行读写
003700BE  pop     ebx
003700BF  mov     esi, eax
003700C1  mov     eax, ebx
003700C3  add     eax, ebp
003700C5  mov     edi, dword ptr [eax+4]    ;取得欲解压区块的 RVA
003700C8  add     edi, dword ptr [ebp+351]
003700CE  push    esi
003700CF  push    edi
003700D0  call    dword ptr [ebp+355]       ;Aplib 解压函数 aP_depack_asm
003700D6  mov     ecx, dword ptr [ebx+ebp] ;原区块大小，即需写回的解压数据的大小
003700D9  push    esi
003700DA  rep     movs byte ptr es:[edi], byte ptr [esi] ;将解压后的数据写回
003700DC  pop     esi
003700DD  push    ebx
003700DE  push    8000
003700E3  push    0
003700E5  push    esi
003700E6  call    dword ptr [ebp+35D]       ;VirtualFree 释放内存
003700EC  pop     ebx
003700ED  add     ebx, 0C
```

```
003700F0  jmp     short 003700A5          ;循环
......
;************* 填充 IAT***************
003700F2  mov     eax, dword ptr [ebp+28D]
003700F8  or      eax, eax                ;为 0 表示输入表没加密
003700FA  jnz     00370181
......
00370181  mov     edx, dword ptr [ebp+291] ;此处为对转储后的输入表的初始化代码
00370187  add     edx, ebp
00370189  mov     edi, dword ptr [edx]
0037018B  or      edi, edi
......
003701F2  loopd   short 003701BE          ;循环
003701F4  jmp     short 00370189          ;准备处理下一 DLL

;*************修正重定位数据*************
003701F6  mov     esi, dword ptr [ebp+299]
003701FC  or      esi, esi
003701FE  je      short 00370233          ;本例 EXE 没有重定位表，故不处理
......
;*******Anti Dump*****************
00370233  push    dword ptr fs:[30]
0037023A  pop     eax
0037023B  test    eax, eax
0037023D  js      short 0037024E          ;判断操作系统
0037023F  mov     eax, dword ptr [eax+C]  ;Windows NT/2000/XP 时的处理代码
00370242  mov     eax, dword ptr [eax+C]
00370245  mov     dword ptr [eax+20], 1000
0037024C  jmp     short 0037026A
0037024E  push    0                       ;Windows 9x 时的处理代码
00370250  call    dword ptr [ebp+345]
......
;*************准备返回 OEP***************
00370270  mov     eax, dword ptr [ebp+289] ;取出原来的入口 RVA，此为 1130
00370276  add     eax, dword ptr [ebp+351] ;加上映像基地址，此为 401130
0037027C  add     dword ptr [ebp+284], eax ;将“401130”放到 370284 这句处
00370282  popad
00370283  push    401130                  ;401130 即为 OEP，由 37027C 这句动态生成
00370288  retn
```

至此，外壳代码处理完毕，其已将目标程序初始化完毕，然后从外壳段直接跳到代码段执行，即用跨段的转移指令跳到真正的入口点执行解压后的程序。转移指令如下：

```
00370283  push    401130
00370288  retn
```

这句指令相当于 jmp 401130。来到 401130 处，读者可能会看到如下代码：

```
00401130      55      db      55                              ;  CHAR 'U'
00401131      8B      db      8B
00401132      EC      db      EC
00401133      6A      db      6A                              ;  CHAR 'j'
```

此时按“Ctrl+A”键强迫 OllyDbg 重新分析一下代码即可。

```
00401130   55            push    ebp
00401131   8BEC          mov     ebp, esp
```

再来完整地回顾一下外壳初始化过程。外壳部分用了两次跨段转移指令：一次是从.pediy 区块跳到外壳第二段（调用 VirtualAlloc 随机分配）；第二次是从外壳第二段，跳到程序本身代码所处的区块，本例是.text 区块，这也是一般判断是否 OEP 的关键。停在 OEP 时，在 OllyDbg 里按“Alt+M”键，可以看到各模板情况，如图 13.5 所示。

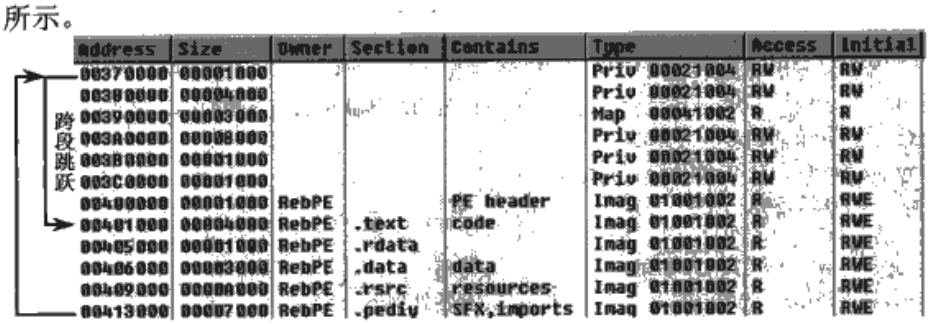


图 13.5　查看实例的内存模块

这个方法没什么技巧，从壳的开始一直跟踪，直到来到代码段本身，从而确定 OEP。

13.2.2　用内存访问断点找 OEP

外壳首先将原来压缩的代码解压，并放到对应的区块上，处理完毕，将跳到代码段执行。上一节是手动跟踪这个过程，一直来到代码段上。很高兴，OllyDbg 有对代码段设断的功能，可以避免手动一步步地跟踪了。当对代码段设置内存访问断点时，一定会中断在外壳对代码写入的那句上面。

按“Alt+M”键打开内存模板，对代码段（本例是.text 区块）按 F2 键下内存访问断点，如图 13.6 所示。这个断点是一次性断点，当所在段被读取或执行时就中断，中断发生以后，断点将被自动删除。

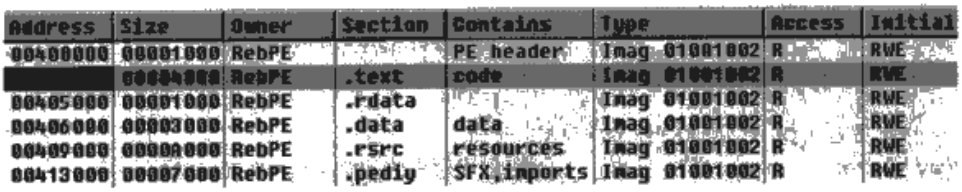


图 13.6　对代码段下内存访问断点

对.text 区块设置内存访问断点后，按 F9 键执行程序，程序将中断如下代码处：

```
00413145  movs    byte ptr es:[edi], byte ptr [esi]  ;将停在此处
00413146  mov     bl, 2
00413148  call    004131BA
0041314D  jnb     short 00413145
......
004131D6  sub     edi, dword ptr [esp+28]
004131DA  mov     dword ptr [esp+1C], edi
004131DE  popad
004131DF  retn    8
```

上面这段代码是 Aplib 解压函数 aP_depack_asm，走出这个函数，将来到外壳代码处。具体如下：

```
003700D0  call    dword ptr [ebp+355]      ;Aplib解压函数aP_depack_asm
003700D6  mov     ecx, dword ptr [ebx+ebp] ;原区块大小，即需写回的解压数据的大小
003700D9  push    esi
003700DA  rep     movs byte ptr es:[edi], byte ptr [esi] ;将解压后的数据写回
```

这段代码依次将.text、.rdata、.data、.rsrc 区块解压，并放到正确的位置上。此时代码段全部解压完毕

后，再对代码段（.text 区块）设置内存访问断点，按 F9 键执行程序，当外壳跳到 OEP 返回代码段时，即可触发内存访问断点而中断。

这个方法的关键是要等代码段解压完毕，再对代码段设置内存访问断点。如果之前设置断点，会不停地中断在对代码写入的指令上。解决这个问题还有一个更好的方法，即下两次内存断点办法。一般的壳，会依次对.text、.rdata、.data、.rsrc 区块进行解压处理，所以，可以先在.rdata、.data 等区块下内存访问断点，中断后，此时代码段已解压，接着再对代码段（.text 块）下内存访问断点，即可达到 OEP。

13.2.3 根据堆栈平衡原理找 OEP

编写加壳软件时，必须保证外壳初始化的现场环境（各寄存器值）与原程序的现场环境是相同的（主要是 ESP、EBP 等重要的寄存器值）。加壳程序初始化时保存各寄存器的值，外壳执行完毕，再恢复各寄存器内容，最后再跳到原程序执行。通常用 pushad/popad、pushfd/popfd 指令对来保存与恢复现场环境，其中 EFLAGS（标志寄存器）影响不是太重要，一般不处理。也就是说，编写加壳软件时，必须遵守堆栈平衡的原理，具体如下：

```
        PUSHAD          ;PUSHAD 相当于 push eax, ecx, edx, ebx, esp, ebp, esi, edi
        ……              ;外壳代码部分
        POPAD           ;POPAD 相当于 pop edi, esi, ebp, esp, ebx, edx, ecx, eax
        JMP OEP         ;准备跳到入口点处
OEP:    ……              ;解压后程序的原代码
```

脱壳时，可以根据这个堆栈平衡的原理对 ESP 设断，能很快找到 OEP。

用 OllyDbg 加载 RebPE，当执行 pushad 指令后，各寄存器的值将被压入 0012FFA4h～0012FFC0h 的堆栈中，如图 13.7 所示为堆栈窗口。

图 13.7 查看堆栈窗口

此时 ESP 指向 12FFA4h，对这个地址下硬件访问断点，如图 13.8 所示。

图 13.8 下硬件访问断点

按 F9 键运行程序，外壳代码处理结束后，调用 popad 指令恢复现场环境时，访问这些堆栈，OllyDbg 将会中断，此处离 OEP 也不远了。

```
00370282  popad
00370283  push    401130              ;将会中断在这里
00370288  retn                        ;返回 OEP
```

可以把整个外壳当做一个函数或子程序来理解，这个过程遵守堆栈平衡的原理，当其跳到 OEP 时，ESP 的值不会变。了解这个现象后，就可以用另一个方法直接来到 OEP 了。当 PE 文件开始运行时，记下当时的 ESP 值，假设是 12FFC4h，而大多数程序第一句都是压栈（push 指令），而这句就是对 12FFC0h 进行写入操作，因此对其设置硬件写断点（如图 13.9 所示），就可方便地来到 OEP 附近。

图 13.9　下硬件写断点

注意：程序刚开始运行时，ESP 的值是由操作系统决定的。

13.2.4　根据编译语言特点找 OEP

各类语言编译的文件入口点都有自己的特点，同一种编译器，其入口代码都很类似，都有一段启动代码，编译器编译程序时，自动连接到程序中去。其完成必需的初始化工作后，再调用 WinMain 函数，执行完后，启动代码再次获得控制权，进行一次初始化清除工作。

例如，对于所有的 Visual C++ 6.0 程序来说，其默认的入口点代码大概有 4 个版本，处理 ANSI 字符的 GUI 版本，其启动函数是 WinMainCRTStartup；处理 ANSI 字符的 CUI 版本，其启动函数是 mainCRTStartup；还有两个 Unicode 版本。具体描述参考第 4 章的“4.1 启动函数”这一节。由于 VC 6 启动部分都有这几个函数，如 GetCommandLineA(W)、GetModuleHandleA(W)、GetVersion、GetStartupInfoA(W)，因此可以用来设置断点，很方便定位程序的 OEP。

用 OllyDbg 加载 RebPE，对 GetVersion 函数设置断点。中断两次后，就能返回到 OEP 附近。

```
00401130  push    ebp
00401131  mov     ebp, esp
00401133  push    -1
00401135  push    004050B8
0040113A  push    00401DFC
0040113F  mov     eax, dword ptr fs:[0]
00401145  push    eax
00401146  mov     dword ptr fs:[0], esp
0040114D  sub     esp, 58
00401150  push    ebx
00401151  push    esi
00401152  push    edi
00401153  mov     dword ptr [ebp-18], esp
00401156  call    dword ptr [405028]          ;kernel32.GetVersion
```

读者对常见语言的入口代码都要熟悉，在脱壳修复或定位 OEP 时，将带来很大的方便。

采用默认的启动代码对软件加壳保护不是很有利，一些开发人员于是对启动源代码进行修改，这时程序的入口点与默认的完全不同。

13.3　抓取内存映像

抓取内存映像，也称为转存，英文称为 Dump。其是把内存指定地址的映像文件读取出来，然后用文件等形式保存下来。

脱壳时，在何时 Dump 文件是有一定技巧的。一般情况下，外壳来到 OEP 处 Dump 是正确的。如果程序运行起来再 Dump，一些变量已初始化了，不适合再 Dump 了。在外壳处理过程中，它要把压缩了的全部的代码数据释放到内存中，初始化一些项目，此时也可以选择合适的点 Dump。

13.3.1　Dump 原理

常用的 Dump 软件有 LordPE、ProcDump、PETools 等。这类工具一般是利用 Module32Next 来获取欲

Dump 进程的基本信息的。Module32Next 函数的原型如下：

```
BOOLModule32Next (HANDLE  hSnapshot , LPMODULEENTRY32  lpme)
```

参数：

- hSnapshot：由先前的 CreateToolhelp32Snapshot 函数返回的快照；
- lpme：指向 MODULEENTRY32 结构的指针。

每次执行函数后，它都将把一个进程的信息填入 MODULEENTRY32 结构中。MODULEENTRY32 结构的定义如下：

```
typedef struct tagMODULEENTRY32 {
  DWORD      dwSize;              //此结构的大小
  DWORD      th32ModuleID;
  DWORD      th32ProcessID;       //进程的标识符
  DWORD      GlblcntUsage;
  DWORD      ProccntUsage;
  BYTE*      modBaseAddr;         //进程的映像基址
  DWORD      modBaseSize;         //进程的映像大小
  HMODULE    hModule;             //进程句柄
  TCHAR      szModule[MAX_MODULE_NAME32 + 1];
  TCHAR      szExePath[MAX_PATH];//进程的完整路径
} MODULEENTRY32;
typedef MODULEENTRY32 *PMODULEENTRY32; Members
```

ProcDump 和 LordPE 都是根据此结构中的 modBaseSize 和 modBaseAddr 字段得到进程的映像大小和基址，再调用 ReadProcessMemory 来读取进程内的数据。读取成功以后，ProcDump 会检测 IMAGE_DOS_SIGNATURE 和 IMAGE_NT_SIGNATURE 是否完整。如果完整它就基本不再对其他大多数字段做检验了，如果不完整它会根据 szExePath 字段打开进程的原始文件，读取它的文件头以取代进程的文件头。LordPE 则更简单，它根本不用进程的文件头，直接读取原始文件的文件头来用。在读取内存数据后，再把进程中的数据保存到硬盘的文件里。

在这里以 LordPE 讲述一下此类工具用法。在如图 13.10 所示的 Options 里，默认选上“Full dump:paste header from disk”，也就是说，PE 头的信息是直接从磁盘文件获得的。

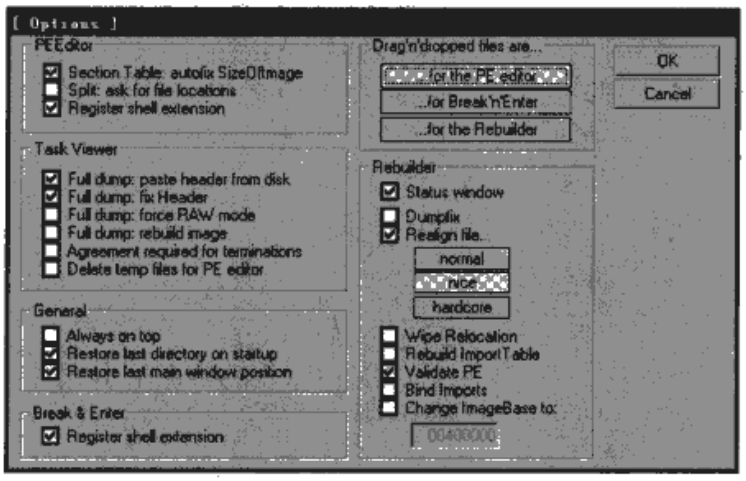


图 13.10　LordPE 选项设置

设置好后，在 LordPE 的进程窗口选择相关进程，单击右键，执行“dump full”菜单命令，即可抓取文件并保存到文件里，如图 13.11 所示。

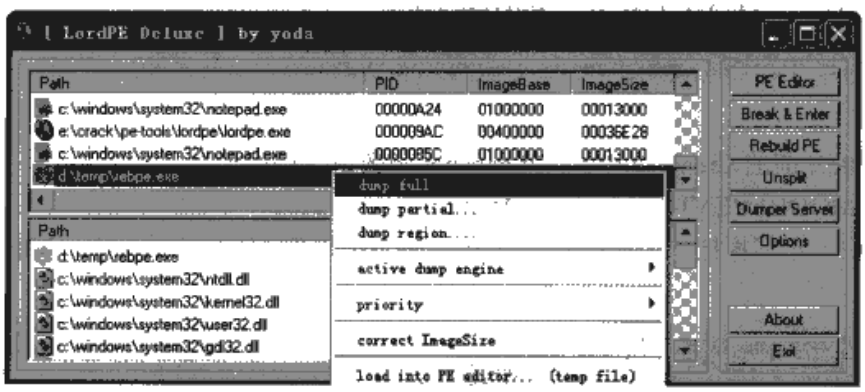


图 13.11　LordPE 里 Dump 镜像文件

OllyDbg 的一个插件 OllyDump，也支持 Dump 功能，使用也比较方便，缺点是不方便 Dump 取 DLL 文件的镜像。

13.3.2　反 Dump 技术（Anti-Dump）

Dump 是脱壳过程中的一个关键步骤，部分加密外壳会采取 Anti-Dump 技术来防止被脱壳，为了能顺利脱壳，就必须绕过这些 Anti-Dump。

1．纠正 SizeOfImage

Dump 文件时，一些关键参数是通过 MODULEENTRY32 结构快照获得的，因此可以在 modBaseSize 和 modBaseAddr 字段中填入错误的值，让 Dump 软件无法正确读取进程的数据。经测试发现，如果修改系统中 modBaseAddr 的值会让系统出现问题，所以只能修改 modBaseSize 的值，方法如下：

```
    ;code by Hying
    Mov eax, fs:[30h]                    ;获得 PEB 的首地址
    testeax, eax
    js  fuapfdw_is9x                     ;9x 内核和 NT 内核不同，要分别进行不同处理

fuapfdw_isNT:
    mov eax, [eax+0Ch]                   ;+00c struct   _PEB_LDR_DATA *Ldr
    mov eax, [eax+0Ch]                   ;LDR_MODULE 的首地址
    mov dword ptr [eax+20h], 1000h       ;[EAX+20H]中保存有进程映像的大小
    jmp fuapfdw_finished

fuapfdw_is9x:
    invoke  GetModuleHandleA, 0
    testedx, edx
    jns fuapfdw_finished
    cmp dword ptr [edx+8], -1
    jne fuapfdw_finished
    mov edx, [edx+4]                     ;EDX 指向系统保存的另一份 PE 头数据
    mov dword ptr [edx+50h], 1000h       ;修改此份 PE 头的 SizeOfImage 字段

fuapfdw_finished:
```

在程序中插入这样一段代码后，用 Module32Next 函数得到的该进程的映像大小就是 1000h 字节，Dump 软件也就只会读出 1000h 字节的程序数据。对于 ProcDump，很可能在进行后续工作时产生非法操作，而用 LordPE 得到的将只是一个大小为 4KB 的无用文件。

如何来防止它呢？当然就是找出类似的代码，然后跳过它。LordPE 的“correct ImageSize”功能也可

以处理这个 Anti，其原理是打开磁盘文件，直接读取 PE 头的 SizeOfImage 来纠正这个错误。实例 RebPE 就带有这个 Anti-Dump，运行后，在相关进程的右键菜单中执行“correct ImageSize”功能，然后再 dump full，可以得到完整的文件，如图 13.12 所示。

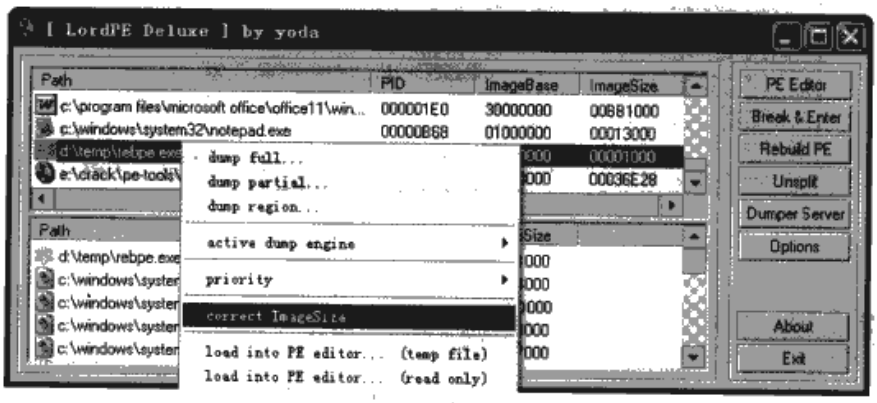


图 13.12 LordPE 里“correct ImageSize”功能

2. 修改内存属性

当 PE 文件被加载到内存中时，其所有段的属性都是可读的，这样 Dump 工具打开进程时，就可读取内存数据并转存到磁盘文件中。如果进程的某个地址不可读，某些 Dump 工具可能不能正确读取相关数据。

本书光盘映像文件中提供的实例 Modify_the_read_right 使用 VirtualProtect 函数将 PE 头设为不可读，运行后，用 LordPE 来 Dump，会出现如图 13.13 所示的错误框。

需要查看内存指定区域属性的时候，可以使用 LordPE 的“dump region”功能。选中进程，执行右键菜单中的“dump region”，如图 13.14 所示。地址 400000h 处 PE 头部显示为 NOACCESS（不可读写）属性，这样用 LordPE 工具读取不了数据。

图 13.13 不能读取内存数据出错

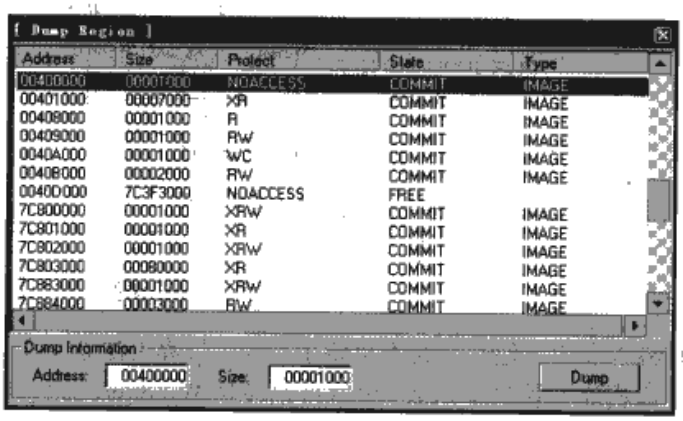


图 13.14 查看内存属性

在这种情况下，如何 Dump 内存映像呢？可以用 OllyDbg 加载目标程序，运行后，按“Alt+M”键打开内存镜像，在该进程的 PE 头单击右键，执行“Set access/Full access”，将 PE 头设置为完整权限，如图 13.15 所示。这样，再运行 LordPE，就可 Dump 内存境像了。

另一款 PE 工具——PETools 1.5.8 却能成功 Dump 该实例进程，但是查看抓取出的映像文件，会发现 PE 头部分全是 0。跟踪一下 PETools 的 Dump 过程，会发现它遇到 NOACCESS 页面并没有 Dump，直接放弃，它只 Dump 能读的页面了。也就是说，在这种情况下 PETools 抓取的内存映像是不完整的。

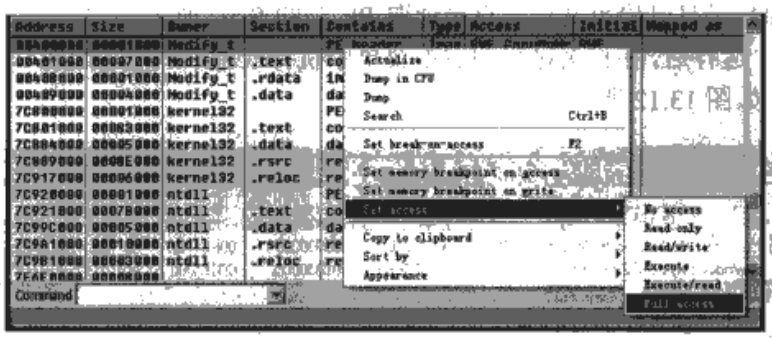


图 13.15　设置区块属性

13.4 重建输入表

在加密外壳中，破坏原程序的输入表是必有的功能，在脱壳中输入表处理是很关键的一个环节，因此要求脱壳者对 PE 格式中的输入表概念非常清楚。

13.4.1　输入表重建的原理

输入表结构中与实际运行相关的主要是 IAT 结构，这个结构用于保存 API 的实际地址。PE 文件运行时，在初始化输入表这块时，Windows 装载器首先搜索 OriginalFirstThunk，如果存在，加载程序迭代搜索数组中的每个指针，找到每个 IMAGE_IMPORT_BY_NAME 结构所指向的输入函数的地址，然后加载器用函数真正入口地址来替代由 FirstThunk 指向的 IMAGE_THUNK_DATA 数组里的元素值。当初始化结束时，输入表情况如图 13.16 所示。

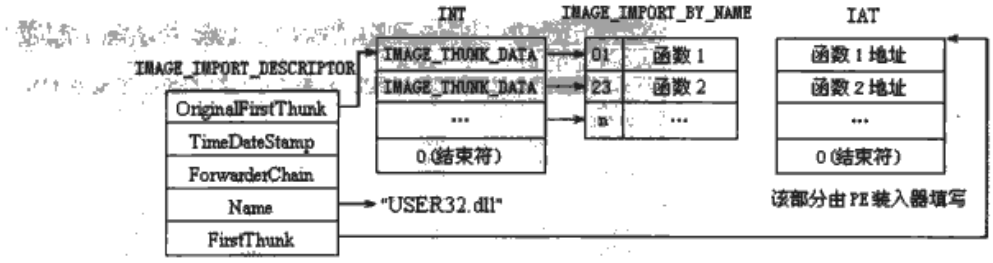


图 13.16　PE 文件加载后的 IAT 表

此时输入表中其他部分就不重要了，程序依靠 IAT 提供的函数地址就可正常运行。对于外壳程序，一般都修改了原程序文件的输入表，然后自己模仿 Windows 装载器的工作来填充 IAT 中相关的数据，也就是说，内存中就一张 IAT 表，原程序的输入表是不存的。输入表重建就是根据这个 IAT 还原整个输入表的结构（即图 13.16 所示的这个结构），如 IID 结构及其各成员指向的数据等。

一些加密软件为了防止输入表被还原，就在 IAT 加密上大作文章，此时外壳填充 IAT 里的不是实际的 API 地址，而是填充壳中用来 HOOK-API 的外壳代码的地址。这样外壳中的代码一旦完成了加载工作，在进入原程序的代码之后，仍然能够间接地获得程序的控制权。因为程序总是需要与系统打交道，与系统打交道的途径是 API，而 API 的地址已经替换成了外壳的 HOOK-API 的地址，那每一次程序与系统打交道，都会让外壳的代码获得一次控制权，这样外壳可以进行反跟踪继续保护软件，同时也可完成某些特殊的任务。所以重建输入表的关键是获得没加密的 IAT，一般的做法是跟踪加壳程序对 IAT 处理过程，修改相关指令，不让外壳加密 IAT。

13.4.2 确定 IAT 的地址和大小

输入表重建的关键就在于 IAT 的获得，一般程序的 IAT 是连续排列的，以一个 DWORD 字的 0 作为结束，因此只要确定 IAT 某个点，就能获得整个 IAT 地址和大小。

程序中每一个 API 函数在 IAT 里都有它自己的位置，这样无论代码中多少次调用一个输入函数，都会通过 IAT 中的同一个函数指针来完成。要确定 IAT 的地址，先看看程序是怎样调用一个输入函数的。一种情况是像下面这样：

```
00401156  FF15 28504000 call    dword ptr [405028];kernel32.GetVersion
```

直接调用[405028]中的函数，地址 405028h 位于 IAT 里，指向 GetVersion 函数。另一种 API 调用像下面这样：

```
0040109D  E8 F4DF0A00   call    004AF096
.....
004AF096  FF25 48D35000 jmp     dword ptr [50D330];KERNEL32.GetProcessHeap
```

在这种情况下，CALL 把控制权转到一个子程序，子程序中的 JMP 指令跳转到位于 IAT 中的 50D330h。

本节通过实例演示一下如何确定 IAT 位置。用 OllyDbg 打开没有加壳的 RebPE.exe，随便找一句调用 API 函数的语句，如图 13.17 所示。

```
00401152  .  57          push    edi
00401153  .  8965 E8     mov     dword ptr [ebp-18], esp
00401156  .  FF15 28504000 call  dword ptr [405028]      kernel32.GetVersion
0040115C  .  33D2        xor     edx, edx
```

图 13.17 调用 API 函数的语句

地址 405028 就是 IAT 中的一个部分，在数据窗口查看其内容，如图 13.18 所示。

```
00405028 FA 11 81 7C 82 CA 81 7C 16 1E 80 7C 15 DE 80 7C
00405038 72 30 86 7C EF B4 80 7C 5F D6 81 7C 07 4B 81 7C
00405048 F4 A0 80 7C 83 CC 81 7C 28 2F 81 7C B7 CC 80 7C
00405058 59 2F 81 7C 12 4B 81 7C FE 2A 81 7C 18 0F 81 7C
Command: d 405028
```

图 13.18 在数据窗口查看 IAT

图 13.18 所示的每一个 DWORD 数据都是指向一个 API 函数，如 FA 11 81 7C 就是地址：7C8111FA，在 OllyDbg 里按“Ctrl+G”键，输入 7C8111FA 跳到这个地址就会发现是 GetVersion 函数，如图 13.19 所示。

```
7C8111F9                90              nop
7C8111FA kernel32.GetVersion  64:A1 18000000  mov     eax, dword ptr fs:[18]
7C811200                8B48 30         mov     ecx, dword ptr [eax+30]
7C811203                8B81 B0000000   mov     eax, dword ptr [ecx+B0]
```

图 13.19 查看 API 函数

IAT 是一块连续排列的数据，因此在数据窗口向上翻屏，直到出现 00 数据，寻找 IAT 起始地址，本例翻屏后直到 405000h，此处也是.rdata 区块的起始端，如图 13.20 所示。

```
00405000 71 8E 81 7C B0 A4 80 7C 94 89 83 7C C8 CC 80 7C
00405010 70 8D 83 7C 18 9C 80 7C C4 3F 80 7C C1 B6 80 7C
```

图 13.20 确定 IAT 起始地址

再向下翻屏确定 IAT 末端，输入表每个 DLL 对应一个 IAT，一般这些 IAT 以一个数据为 0 的 DWORD 间隔。IAT 最后是以 0 结尾的，如图 13.21 所示，4050B8h 处就是 IAT 结尾。也就是说，IAT 的地址是 405000h，大小为 450B8h-405000h=B8h。

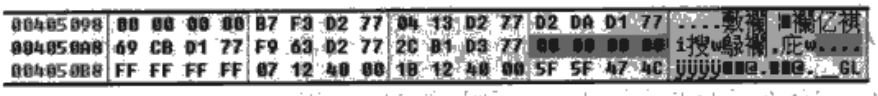


图 13.21　确定 IAT 结束地址

为了直观些，也可以这样让数据窗口直接显示这些 API 函数，以确定 IAT 是否正确，在数据窗口单击右键，执行菜单“Long/Address”命令，如图 13.22 所示。

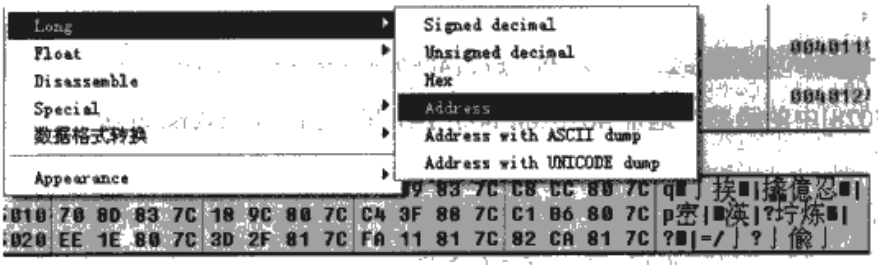


图 13.22　切换数据窗口显示模式

调整后的数据窗口就直观了，直接显示调用的 API 函数名，如图 13.23 所示。若要恢复原有模式，执行菜单“Hex/ASCII(16 bytes)”命令。

图 13.23　以 Address 模式显示

13.4.3　根据 IAT 重建输入表

本节演示一下如何利用 IAT 重建一份完整的输入表，以加深对输入表的理解。当然实际工作中不需要手工构造输入表，可以用 ImportREC 等专业的输入表重建工具来完成这些工作。

实例 Reb_IT.exe 输入表比较简单，容易手工构建输入表，用 UPX 对实例进行加壳处理。用 OllyDbg 加载已加壳的目标后，在代码窗口中一直往下翻屏，能发现如下代码：

```
004052BE   61            popad
004052BF   E9 3CEDFFFF   jmp     00401000    ;跳到OEP
```

处理完数据后，UPX 外壳用了一次跨段的转移指令（JMP）跳到 OEP，会发现 OEP 为 401000h。因此，只需要在 4052BFh 处设断，中断后，运行 LordPE 将内存数据 Dump 出来，保存为 dumped.exe。

用上一节方法确定 IAT 的位置，如图 13.24 所示。

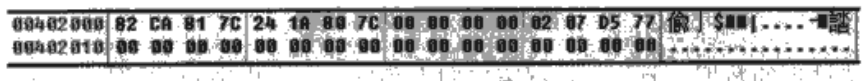


图 13.24　查看 IAT

为了分析方便，用 Address 模式查看 IAT，如图 13.25 所示。

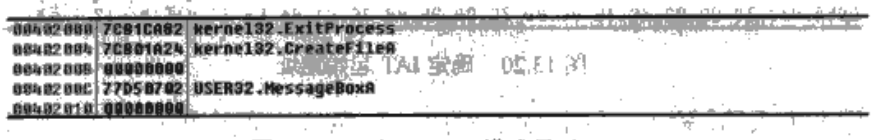


图 13.25　以 Address 模式显示 IAT

这个程序输入表里输入了两个 DLL：一个是 kernel32.dll，另一个是 user32.dll，分别对应了一份 IAT，

两份 IAT 以一个 DWORD 的 0 隔开。将 IAT 结果总结在表 13-1 中。

表 13-1 IAT 成员指向的函数名

KERNEL32.dll	ExitProcess	CreateFileA
USER32.dll	MessageBoxA	

用十六进制工具在 dumped.exe 文件中找一块空间，在这里选择 2100h，将表 13-1 中的 DLL 名和函数名写进去（见图 13.26）。DLL 与函数名的位置可以任意。每个函数名前要留两个字节放函数的序号，序号可以为 0；每个函数名后的一个字节为 0；每个函数名或 DLL 名起始地址必须按偶数对齐，空隙填充 0。

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00002100 4B 45 52 4E 45 4C 33 32  2E 44 4C 4C 00 55 53 45  KERNEL32.DLL.USE
00002110 52 33 32 2E 64 6C 6C 00  00 00 45 78 69 74 50 72  R32.dll...ExitPr
00002120 6F 63 65 73 73 00 00 00  43 72 65 61 74 65 46 69  ocess...CreateFi
00002130 6C 65 41 00 00 00 4D 65  73 73 61 67 65 42 6F 78  leA...MessageBox
00002140 41 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  A...............
```

图 13.26 填充 IMAGE_IMPORT_BY_NAME 结构

由于是内存映像文件，因此文件偏移地址与相对虚拟地址（RVA）值是相等的。将图 13.26 中各 DLL 名和函数名所在的偏移地址归纳到表 13-2 中。

注意：下面会忽略所有的文件偏移地址和 RVA 注释。

表 13-2 各字符串的地址

DLL 地址		函数名地址			
KERNEL32.dll	00002100h	ExitProcess	00002118h	CreateFileA	00002126h
USER32.dll	0000210Dh	MessageBoxA	00002134h		

根据表 13-2 构造指向函数名地址的 IMAGE_THUNK_DATA 数组，具体见表 13-3。

表 13-3 IMAGE_THUNK_DATA 数组

第一个 IID 的 IMAGE_THUNK_DATA 数组	1821 0000（20E0h 处）	2621 0000
第二个 IID 的 IMAGE_THUNK_DATA 数组	3421 0000（20ECh 处）	

选择 20E0h 放 IMAGE_THUNK_DATA 数组，两个数组之间空 2 个字节，填充 0，如图 13.27 所示。

```
Offset   0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
000020E0 18 21 00 00 26 21 00 00  00 00 00 00 34 21 00 00  .!..&!......4!..
000020F0 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00002100 4B 45 52 4E 45 4C 33 32  2E 44 4C 4C 00 55 53 45  KERNEL32.DLL.USE
00002110 52 33 32 2E 64 6C 6C 00  00 00 45 78 69 74 50 72  R32.dll...ExitPr
00002120 6F 63 65 73 73 00 00 00  43 72 65 61 74 65 46 69  ocess...CreateFi
00002130 6C 65 41 00 00 00 4D 65  73 73 61 67 65 42 6F 78  leA...MessageBox
00002140 41 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  A...............
```

图 13.27 填充 IMAGE_THUNK_DATA 数组

最后构建其 IID 数组，如表 13-4 所示。

表 13-4 IID 数组

DLL 名称	OrignalFirstThunk	TimeDateStamp	ForwardChain	Name	First Thunk
KERNEL32.dll	E020 0000	0000 0000	0000 0000	0021 0000	0020 0000
USER32.dll	EC20 0000	0000 0000	0000 0000	0D21 0000	0C20 0000
结束标志	0000 0000	0000 0000	0000 0000	0000 0000	0000 0000

IID 数组位置也可以任意，在这里放在 2010h 地址处。其中 IAT 位置很重要，不能改变，否则相关指令就找不到函数调用地址（除非再修正这些函数调用的地址），IAT 中的内容可以不重新构造，PE 文件加载时，Windows 系统会填充。在这里将 FirstThunk 指向原来的 IAT，并将 OrignalFirstThunk 指向的数据复制到 FirstThunk 指向的空间。图 13.28 所示是第一个 IID 结构示意图。

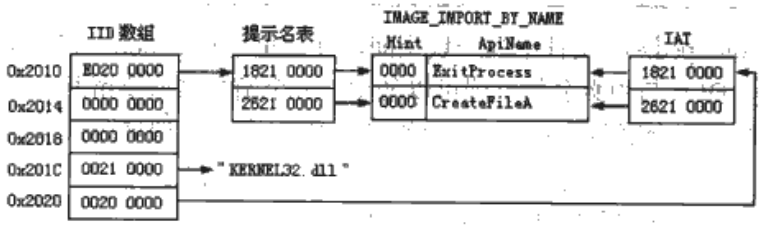


图 13.28　第一个 IID 结构示意图

图 13.29 所示是手动构建的完整 IID，其中输入表的地址是 2010h，大小是 28h。

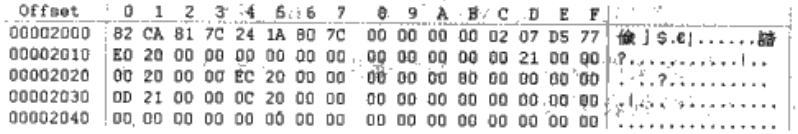


图 13.29　手动构建的 IID

输入表的相对虚拟地址（RVA）存储在 PE 文件头部的目录表中（它的偏移量为[PE 文件头偏移量+80h]）。

PE 文件头偏移量+80h=B0h+80h=130h

在 130h 处写入输入表的地址和大小：1020 0000 2800 0000；或直接用 PE 编辑工具修改目录表里的输入表选项，如图 13.30 所示。

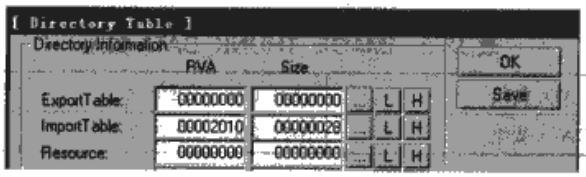


图 13.30　LordPE 中修改输入表的地址

13.4.4　ImportREC 重建输入表

ImportREC 是目前最好用的输入表重建专业工具，它可从杂乱的 IAT 中重建一个新的输入表，原理参考上一节的手动构造输入表。

1．基本用法

在运行 ImportREC 之前，必须满足如下条件：

- 目标文件已完全被 Dump，另存为一个文件；
- 目标文件必须正在运行中；
- 事先找到目标程序真正的入口点（OEP）或 IAT 的偏移与大小。

这里以 13.2.1 节的 RebPE.exe 为例，讲述 ImportREC 的使用。让 OllyDbg 暂停在 OEP 401130h 处，运行 LordPE，由于这个实例有 Anti-Dump，执行“correct ImageSize”功能，然后再“dump full”，保存为 dumped.exe（此时，一些杀毒软件会提示 dumped.exe 为病毒，不必理会）。

准备工作做好，然后运行 ImportREC，操作如下：

① 运行 ImportREC，在下拉列表框中选择 RebPE.exe 进程，如图 13.31 所示。

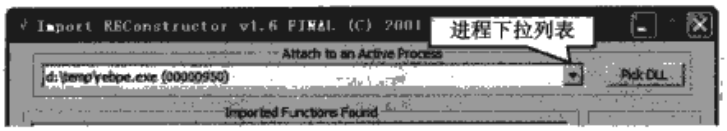


图 13.31 ImportREC 进程下拉列表

② 如有正确的 OEP 值，则在左下角 OEP 处填入 OEP 的 RVA 值，这里填上 1130。默认时，ImportREC 重建输入表时会同时用此值修正入口点（选项里可设置），同时提供正确的 OEP 有助于分析 IAT 的准确位置。单击“IAT AutoSearch”按钮，让其自动检测 IAT 偏移和大小，如果出现“Found address which may be in the Original IAT.Try 'Get Import'”对话框，如图 13.32 所示，这表示输入的 OEP 发挥作用了。接下来跳到第④步。

③ 如没正确的 OEP 或 ImportREC 没找到 IAT 偏移，则手工填入 IAT 的 RVA 和大小，如图 13.33 所示。

图 13.32 提示发现 IAT 的地址和大小

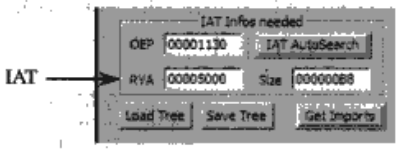


图 13.33 IAT 的 RVA 与 Size 域

④ 单击“Get Imports”按钮，让其分析 IAT 结构得到基本信息，如图 13.34 所示。

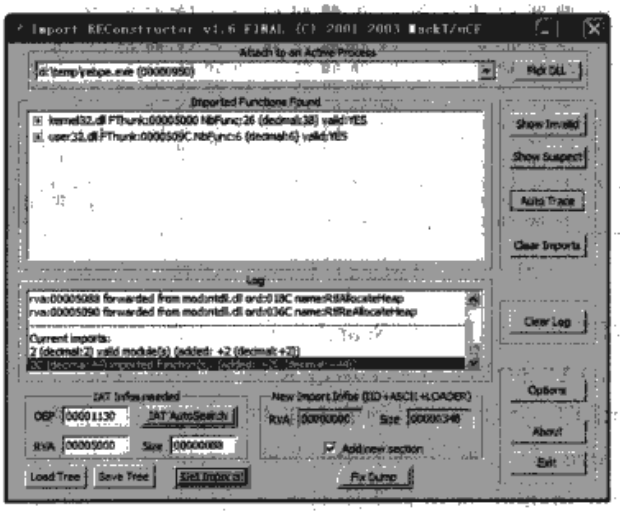


图 13.34 分析 IAT 获取输入表相关信息

⑤ 本例所有 API 函数都能正确识别。如果不能识别，会显示“valid :NO”，单击“Show Invalid”按钮分析所有的无效信息，在“Imported Functions Found”栏中单击鼠标右键，选择“Trace Level1(Disasm)”，再单击“Show Invalid”按钮。如果成功，可以看到所有的 DLL 都为“valid:YES”。如果仍有无效的地址，可以尝试右键菜单中的“Trace Level 2(HOOK)”或“Trace Level 2(Trap Flag)”命令修复（尽量关闭其他程序）。“Auto Trace”按钮用来自动执行 Trace Level 1、Trace Level 2 等。

⑥ 最后修复已脱壳的程序 Dump.exe。选择“Add new section”（默认为选中）为 Dump.exe 文件加一个区块，区块名为.mackt（虽然文件变大，但避免了许多不必要的麻烦）。单击“Fix Dump”按钮，并选择刚抓取的映像文件 Dump.exe，在此不必备份。如果修复的文件名是“Dump.exe”，它将创建一个“Dump_.exe”，此外 OEP 也被自动修正。

⑦ 第⑥步脱壳后，输入表放在新增的.mackt 区块上。也可以将新的输入表放到程序空白处，本例可

以在.rdata 选一空白，此处为 40545Ch，在 ImportREC 里填上 RVA545Ch（如图 13.35 所示），再单击“Fix Dump”按钮即可。

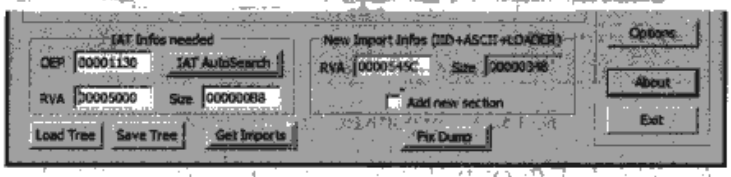


图 13.35　指定输入表的存放地址

2．处理不连续的 IAT

输入表里，一个 DLL 对应一份 IAT，多个 IAT 之间一般以一个 DWORD 的 0 隔开。也有些程序其 IAT 被分割成几部分，如 Borland C++ 1999、Borland C++ Builder 等程序。

OllyDbg 打开实例 TestWin.exe，分析获得 kernel32.dll 所在的 IAT 为 40E0ECh～40E188h，大小为 9Ch；gdi32.dll 所在的 IAT 为 40E190～40E198h，大小为 8h；user32.dll 所在的 IAT 为 40E1ECh～40E230h，大小为 44h。这三份 IAT 不连续，各有一段间隙，如图 13.36 所示。

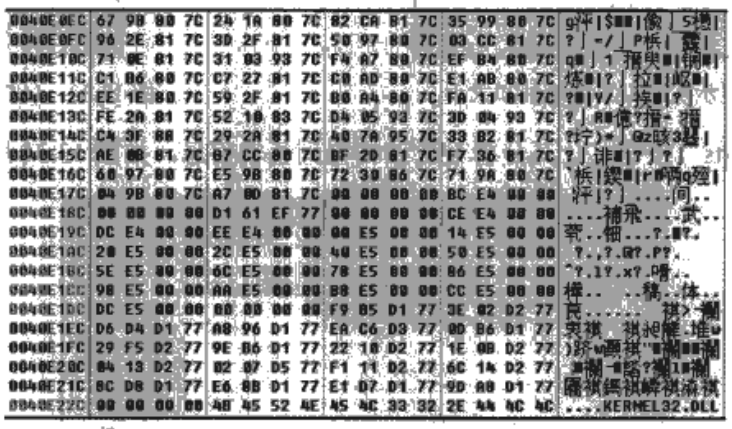


图 13.36　Borland C 程序的 IAT 排列不连续

遇到这种情况，ImportREC 的“IAT AutoSearch”只能自动检测第一份 IAT 偏移和大小。要得到正确结果，必须依次将其他各小块 IAT 的地址和大小填进 RVA 和 Size 域中，单击“Get Imports”按钮分析 IAT 结构。重复这个过程，ImportREC 会自动将得到的 IAT 数据组合成一个整体。但这种方法不是很好，也可以直接将整个 IAT 大小填进 Size 域中，此处填 144h（保证填入的值大于真实 Size 即可），单击“Get Imports”按钮分析 IAT 结构。单击“Show Invalid”按钮分析所有的无效信息，在无效信息中单击右键，执行“Cut thunk”功能将无用信息清除，即可得到一份正确的输入表，如图 13.37 所示。

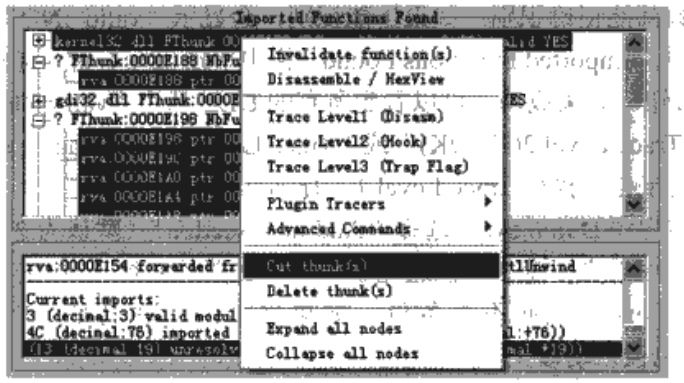


图 13.37　消除无用的信息

3. 修复函数

ImportREC 会对个别函数识别出错，这时必须通过跟踪原程序，获得正确的函数，并进行修复。实例 apitest.exe 是一个简单的程序，在 Windows XP 系统上，运行 ImportREC 获得输入表，如图 13.38 所示。单击“Fix Dump”按钮生成新的 apitest_.exe 程序。

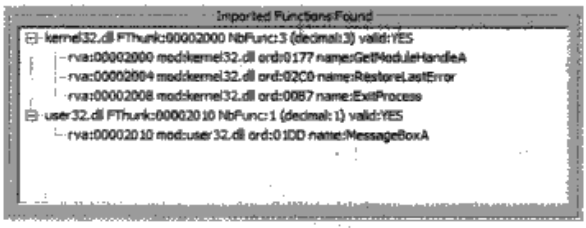


图 13.38 查看树结构的输入表

修复后的 apitest_.exe 在 Windows XP 下能运行，但在 Windows 2000 下运行会出现如图 13.39 所示的错误对话框。

图 13.39 Windows 2000 下运行出错

原来在 Windows XP 下修复输入表的时候，ImportREC 会将 ntdll!RtlRestoreLastWin32Error 重定位到 kernel32!RestoreLastError，这个函数在以前版本的 Windows 系统下是不存在的，这样会造成修复的程序不能跨平台运行。而 RestoreLastError 和 SetLastError 实际上是同一个函数，ImportREC 将 SetLastError 识别成 RestoreLastError 了。

在图 13.38 所示的 kernel32.dll 的子节点 RestoreLastError 上双击鼠标左键，在弹出的输入函数编辑框中对这个输入函数的名字进行修正，改成 SetLastError 函数，如图 13.40 所示。这个问题是 ImportREC 的 bug，一些修改版的 ImportREC 已修正了这个问题。

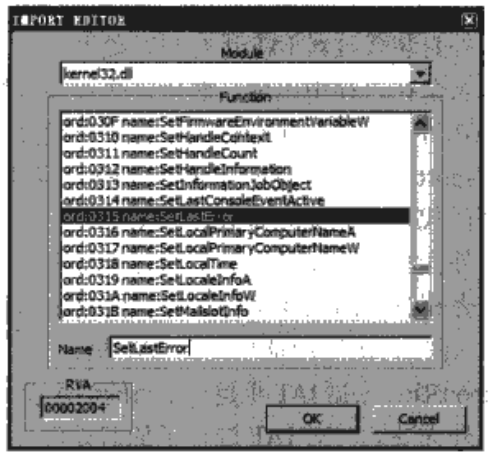


图 13.40 输入函数编辑框

一般遇到函数识别出错的问题时，解决方法就是在出错函数节点上单击右键，执行“Disassemble/HexView”查看对应的反汇编代码，将其与没脱壳的程序对比，分析出真实的 API，然后再重新构建输入表。

4. 其他说明

- 在“Imported Functions Found”栏中单击鼠标右键，可以选择“Expand all nodes”及“Collapse all nodes”来打开/关闭所有节点。
- “Save Tree”将当前输入表以文本文件保存，“Load Tree”从磁盘中导入输入表文本文件。
- 如果 IAT 自动搜索失败，请尝试如下两种方法：
  - 调整“Options（选项）”里的“Max Recursion（最大循环）”和“Buffer Size（缓冲区大小）”。
  - 跟踪程序，获得 IAT 的地址和大小。
- 如果IAT被分割成几部分，依次将各小块IAT的地址和大小填进RVA和Size域中，单击“Get Imports”按钮分析 IAT 结构。重复这个过程，ImportREC 会自动将得到的 IAT 数据组合成一个整体。
- 处理某些壳时，最好是在 OEP 处将被加壳的进程 Suspend（挂起），然后用 ImportREC 分析。因为某些外壳程序在进入 OEP 之后会修改 IAT 的某些项。
- Options 的“New Imports”几个选项可以控制新建输入表的一些结构：
  - “Rebuild OriginalFT”选项就是重建 OriginalFirstThunk，否则以 0 填充，工作时靠 FirstThunk。
  - “Create New IAT”选项是为 IAT 选定一个新的地址，同时修正代码中对 API 函数的调用，所以一般不建议勾选。
  - “Import all by Ordinal”选项表示按序号构建输入表，按序号构建在不同平台容易出兼容性问题，不推荐使用。

13.4.5 输入表加密概括

脱壳过程，输入表修复是个重点。修复的关键是得到没加密的 IAT，这可以对 IAT 相关的点设断，以找到外壳处理 IAT 的代码，然后找对策。加壳程序处理输入表有以下几种情况。

（1）完整地保留了原输入表，外壳加载时没对 IAT 加密

外壳解压数据时，完整的输入表会在内存中出现，然后外壳用显式装载 DLL 方式获得各函数地址（如 GetProcAddress 函数），并将该地址填充到 IAT 中。

脱壳时可以在内存映像文件刚生成时抓取，此时，外壳还没来得及破坏原始的输入表。此类壳有 ASPack、PECompact 等。

（2）完整地保留了原输入表，外壳装载时对 IAT 进行加密处理

外壳解压数据时，完整的输入表会在内存中出现，然后外壳用显式装载 DLL 方式获得各函数地址，并对该地址进行处理（即 HOOK-API），最后将 HOOK-API 的外壳代码的地址填充到 IAT。

由于 IAT 已加密，如直接用 ImportREC 是重建不了输入表的。但可以在外壳还没来得及加密 IAT 时，抓取输入表；或者跳过对 IAT 加密的代码。此类壳有 tElock 等。

（3）加壳时破坏了原输入表，外壳装载时没对 IAT 进行加密处理

外壳已完全破坏原输入表，外壳刚解压的映像文件中存在的是输入函数的字符串，然后外壳用显示装载 DLL 方式获得这些函数的地址，直接将函数地址填充到 IAT 里。

由于 IAT 没加密，脱壳时用 ImportREC 根据 IAT 重建一个新的输入表。此类壳有 UPX 等。

（4）加壳时破坏了原输入表，外壳装载时对 IAT 进行了加密处理

外壳已完全破坏了原输入表，然后外壳用显式装载 DLL 方式获得各函数地址，并对该地址进行处理（即 HOOK-API），最后将 HOOK-API 的外壳代码的地址填充到 IAT。

脱壳时可以利用 ImportREC 的一些插件对付这些加密的 IAT；也可修改外壳处理输入函数地址的代码，让其生成的 IAT 没加密，然后再用 ImportREC 重建。此类壳有 ASProtect 等。

13.5 DLL 文件脱壳

DLL 是 Dynamic Link Library（动态链接库）的缩写形式，是一个共享函数库的可执行文件。DLL 文件的脱壳与 EXE 文件步骤差不多，DLL 文件多了个基址重定位表等要考虑。

13.5.1 寻找 OEP

当 DLL 被初次映射到进程的地址空间中时，系统将调用 DllMain 函数，当卸载 DLL 时也会再次调用 DllMain 函数。也就是说，DLL 文件相比 EXE 文件运行有一些特殊性，EXE 的入口点只在开始时执行一次，而 DLL 的入口点在整个执行过程中至少要执行两次。一次是在开始时，用来对 DLL 做一些初始化。至少还有一次是在退出时，用来清理 DLL 再退出。

外壳编写时也必须考虑这个因素，初次载入时，做些初始化工作，如 IAT 初始化等。退出时再次进入入口点时，外壳将跳过相关的初始化代码，这时程序代码流程会短些。所以找 OEP 也有两条路可以走，一是载入时找，二是在退出时找，退出时流程短些，相对来说更容易找到 OEP。

用第 16 章编写的加壳工具对实例 EdrLib.dll 进行加壳处理。用 LordPE 查看其 PE 信息，获得 EntryPoint 为 D000h，ImageBase 为 400000h，区块的信息如图 13.41 所示。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00006000	00001000	00004000	E0000020
.rdata	00007000	00001000	00005000	00001000	C0000040
.data	00008000	00004000	00006000	00001000	C0000040
.reloc	0000C000	00001000	00007000	00001000	C2000040
.pediy	0000D000	00001000	00008000	00001000	E0000040

图 13.41 查看区块信息

DLL 本身不能直接执行，但可以调用 LoadLibrary 将 DLL 的文件映像映射到调用进程的地址空间中，退出时调用 FreeLibrary 卸载 DLL。为了调试 DLL，OllyDbg 提供了一个类似原理的辅助程序 loaddll.exe，这个程序被压缩存放在资源段里，如果 OllyDbg 所在文件夹内没有 loaddll.exe，则会释放这个文件。用 OllyDbg 打开 DLL，将会询问启动 loaddll.exe，如图 13.42 所示。然后链接库被加载并停在程序的入口（ModuleEntryPoint），现在就可以正常调试 DLL 程序了。

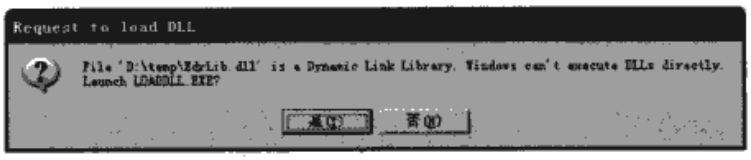


图 13.42 提示是否启动 loaddll.exe

OllyDbg 加载 EdrLib.dll 将停在外壳代码第一行。细心的读者会发现，此时 EdrLib.dll 并没有被映射到默认的 400000h 内存地址中，而是选择了另一个地址。按“Alt+M”键打开内存映像窗口，将会发现 EdrLib.dll 被映射到地址 E20000h，如图 13.43 所示。

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00E20000	00001000	EdrLib		PE header	Imag	R	RWE	
00E21000	00006000	EdrLib	.text	code	Imag	R	RWE	
00E27000	00001000	EdrLib	.rdata	exports	Imag	R	RWE	
00E28000	00004000	EdrLib	.data	data	Imag	R	RWE	
00E2C000	00001000	EdrLib	.reloc		Imag	R	RWE	
00E2D000	00001000	EdrLib	.pediy	SFX,imports	Imag	R	RWE	

图 13.43 查看 DLL 被映射的地址

注意：由于 DLL 被映射的地址是由系统动态分配的，因此在读者系统中显示的地址可能与本书不同，操作时以当前系统基址为准。下面将忽略所有基地址注释，读者操作时以实际的映像地址来操作。

外壳的入口代码如下：

```
00E2D000  pushad
00E2D001  call    00E2D0C8
......
00E2D0C9  sub     ebp, 6
00E2D0CF  mov     eax, dword ptr [ebp+C0]  ;[ebp+C0]是一个计数器变量，此时为 0
00E2D0D5  or      eax, eax                 ;如果是 0,表示首次加载，将执行初始化代码
00E2D0D7  je      short 00E2D0E0
00E2D0D9  push    ebp                      ;dll 文件退出时走这里
00E2D0DA  jmp     dword ptr [ebp+C4]
00E2D0E0  inc     dword ptr [ebp+C0]       ;[ebp+C0]计数器变量加 1
```

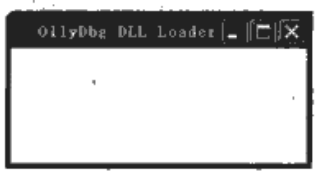


图 13.44　加载 DLL 成功后的界面

此时可以按正常的思路跟踪代码寻找 OEP，由于 DLL 退出时也会来到入口一次，本例演示一下 DLL 退出时寻找 OEP 的过程。先在外壳的入口 00E2D000 处按 F2 键设置一个断点，按 F9 键数次让 DLL 跑起来，DLL 装载成功后，loaddll.exe 将出现如图 13.44 所示的界面。

然后将如图 13.44 所示的窗口关闭，DLL 将被卸载，将再次中断在外壳的入口点处。

```
00E2D000  pushad                           ;退出时，会再次来到这里
00E2D001  call    00E2D0C8
......
00E2D0C9  sub     ebp, 6
00E2D0CF  mov     eax, dword ptr [ebp+C0]  ;[ebp+C0]的值此时是 1
00E2D0D5  or      eax, eax
00E2D0D7  je      short 00E2D0E0
00E2D0D9  push    ebp
00E2D0DA  jmp     dword ptr [ebp+C4]
00E2D0E0  inc     dword ptr [ebp+C0]       ;dll 文件退出时将走这条线路
```

来到外壳代码的第二段，这段是解压还原、初始化原程序，为避免重复初始化，第二次进入入口点后将会跳过这些初始化代码。

```
003E0000  call    003E0005
003E0005  pop     edx
003E0006  sub     edx, 5
003E0009  pop     ebp
003E000A  mov     eax, dword ptr [edx+359]  ;计数器变量
003E0010  or      eax, eax
003E0012  je      short 003E001A
003E0014  popad
003E0015  jmp     0E21240                  ;dll 退出时从这里进入 OEP
```

跳过外壳初始化代码后，将直接到 OEP 处。

```
003E0283  push    0E21240
003E0288  retn                             ;返回到 OEP，即 1240h（RVA 值）
```

此时，OEP 的 RVA 值 = E21240 h-映像地址=E21240 h-E20000h = 1240h。

技巧：一般加壳软件用同一套外壳代码处理 EXE 和 DLL，因此在处理 EXE 文件时，外壳里也有判断是不是多次加载的代码，找到这些跳转，强行改变，这样程序虽不能运行，但能很快定位到 OEP 相关的代码。

OllyDbg 加载某些外壳的 DLL 文件，不能正常地暂停在外壳的入口点，会直接运行起来。碰到这种情况，可以将外壳的入口点改成死循环，机器码是 EBFEh。一个示例如下：

```
00401000 EB FE          jmp     short 00401000
```

OllyDbg 加载修改后的 DLL，由于入口死循环，OllyDbg 的 CPU 窗口显示白屏，按 F12 键让 OllyDbg 暂停即可看到代码，再将入口代码恢复成原指令，又可单步调试了。详细操作见光盘映像文件中动画演示。

13.5.2 Dump 映像文件

停在 OEP 后，运行 LordPE，在进程窗口中选择 loaddll.exe 进程，在下方窗口中的 EdrLib.dll 模块上单击右键，执行“dump full”菜单命令，将文件抓取并保存到文件里，如图 13.45 所示。

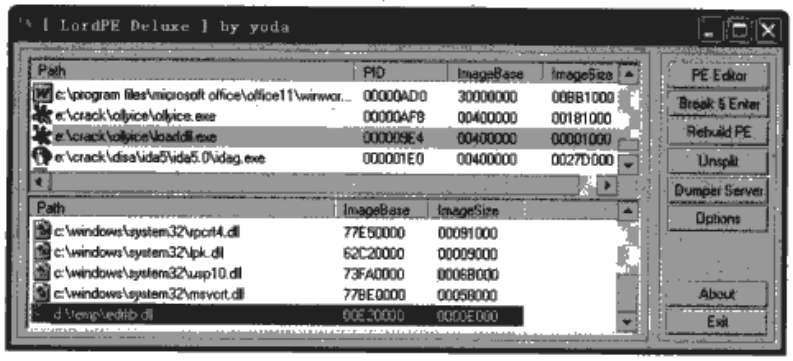


图 13.45 抓取 DLL 内存映像

1. 重定位后抓取映像

对于 DLL 文件来说，Windows 系统没有办法保证每一次运行时提供相同的基地址。如果 DLL 基址所在内存空间被占用或该区域不够大，系统会寻找另一个地址空间的区域来映射 DLL，此时外壳将对 DLL 执行某些重定位操作。从图 13.43 得知，此时 DLL 被映射到内存的地址是 E20000h，与 EdrLib.dll 默认的基址 400000h 不同，被重定位项所指向的地方是已经重定位了的代码数据。

如果没有被重定位，一条访问内存的语句应是这样的：

```
00401253  833D 68AD4000 00    cmp     dword ptr [40AD68], 0
```

重定位后，直接访问内存的地址被修正了，语句变成如下：

```
00E21253  833D 68ADE200 00    cmp     dword ptr [E2AD68], 0
```

为了保证重定位后脱壳后的程序能正常运行，必须修正基址为当前环境的值，本例就是 E20000h（见图 13.46）。如果被重定位的基址小于默认基址，不可用此法。

2. 重定位前抓取映像

上一节修改脱壳文件的基址虽也能解决问题，但问题解决得不是很完美，如果脱壳文件的代码和加壳前一样就完美了。为了得到与加壳前一样的文件，可以在 DLL 载入时跟进外壳，找到重定位的代码，跳过它，让 DLL 不被重定位。

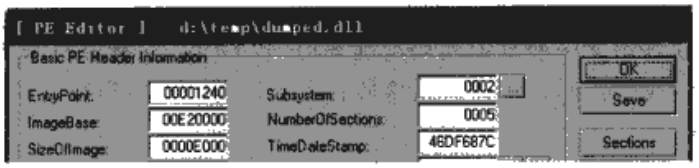


图 13.46　修正 ImageBase

```
003E01F6  mov     esi, dword ptr [ebp+299]      ;取原重定位表的 RVA
003E01FC  or      esi, esi
003E01FE  je      short 003E0233                ;判断需不需要重定位
003E0200  add     esi, dword ptr [ebp+351]
003E0206  mov     edi, dword ptr [ebp+351]
```

在 3E01FEh 这句强行跳转，不让外壳对代码进行重定位，来到 OEP 后，就可直接抓取内存映像了。本例中，由于此时代码段数据已完全恢复，也可以在此处直接抓取内存映像。

本例外壳代码较短，单步跟踪能很快找到重定位处理代码，如果外壳比较复杂，就得花点技巧找到重定位处理代码。首先得选取一个重定位的数据，直接对内存地址操作的指令肯定需要重定位。来到 OEP 后，分析一下相关代码，选取一句会被重定位的语句。例如：

```
00E21253   833D 68ADE200 00   cmp     dword ptr [E2AD68], 0
```

E21255h 这个地址处的数据 E2AD68h 是被重定位了的。因此，重新加载 EdrLib.dll，等代码段解压完毕，在数据窗口对 E21253h 设内存写断点，即可中断在重定位表处理的代码上。

还可用下两次内存访问断点来寻找重定位处理代码。本实例会依次对.text、rdata、.data、.rsrc 区块进行解压处理，所以，可以先在.rdata 等区块下内存访问断点，中断后，此时代码段已解压，接着再对代码段（.text 块）下内存访问断点，当外壳对代码点进行重定位操作时，即可中断。

13.5.3　重建 DLL 的输入表

ImportREC 能很好地支持 DLL 的输入表的重建，首先，在 Options 里将“Use PE Header From Disk”默认的选项去除选中。这是因为 ImportREC 需要获得基址计算 RVA 值，DLL 加载的地址不是默认基址时，从磁盘取默认基址计算会导致结果错误。

- 在 ImportREC 下拉列表框中选择 DLL 装载器的进程，此处为 loaddll.exe 进程。
- 单击“Pick DLL”按钮，在 DLL 进程列表中选择 EdrLib.dll 进程（见图 13.47）。

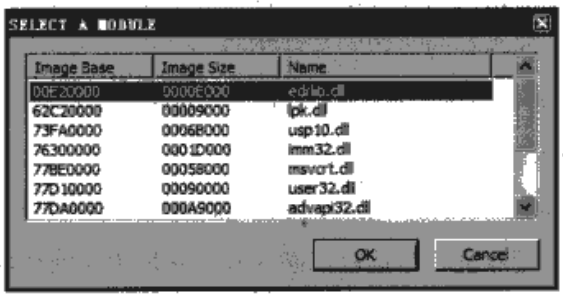


图 13.47　选择 DLL 进程

- 在 OEP 处，填上 DLL 入口的 RVA 值 1240h，单击“IAT AutoSearch”按钮获取 IAT 地址。如果失败，必须手工判断 DLL 的 IAT 位置和大小，其 RVA 为 7000h，Size 为 E8h。
- 单击“Get Imports”按钮，让其分析 IAT 结构重建输入表。
- 勾选“Add new section”，单击“Fix Dump”按钮，并选择刚抓取的映像文件 dumped.dll，它将创建一个 dumped_.dll 文件。

13.5.4 构造重定位表

对于DLL的动态链接库文件来说，重定位数据一般是必需的。外壳很可能破坏了原始的重定位表，将原始的重定位表换个形式存储，运行时模拟PE加载器的重定位功能，对相关代码重定位。脱壳必须根据PE文档的重定位表的定义，重新构造一份新的重定位表。

先来回顾一下重定位表的结构：

```
IMAGE_BASE_RELOCATION STRUCT
    VirtualAddress   dd  0
    SizeOfBlock      dd  0
    Type1            dw  0; 其中：Bit15~Bit12 为类型 type，Bit11~Bit0 为 ItemOffset
IMAGE_RELOCATION ENDS
```

重定位表以1000h大小为一个段，因为ItemOffset最长为12位，即刚好为1000h。如果还有更多段，将重复上面数据结构，直到VirtualAddress为NULL，表示结束。如图13.48所示是重定位表结构的一个示意图。

VirtualAddress	SizeOfBlock	TypeOffset			
00001000	00000010	300F	3023	0000	0000

图13.48 重定位表结构示意图

外壳重定位相关数据时，会根据外壳转储的重定位表确定要重定位的RVA，再加上当前的基址，完成代码重定位工作。本节构造重定位原理就是将这些要重定位的RVA提取出来，再将这些RVA根据重定位表的定义重新生成一份新的重定位表。根据这个原理，笔者写了一款工具来完成这个重建功能，详见光盘映像文件中的ReloREC。

OllyDbg加载EdrLib.dll，来到重定位初始化的地方，代码如下：

```
003E01F6  mov     esi, dword ptr [ebp+299]     ;取原重定位表 RVA
003E01FC  or      esi, esi
003E01FE  je      short 003E0233
003E0200  add     esi, dword ptr [ebp+351]     ;加上载入基址
003E0206  mov     edi, dword ptr [ebp+351]     ;取当前基址
003E020C  mov     ebx, edi
003E020E  sub     edi, dword ptr [ebp+29D]     ;当前基址减去默认基址
003E0214  movzx   eax, byte ptr [esi]          ;从外壳转储的重定位表结构取数据
003E0217  jmp     short 003E022F
003E0219  cmp     al, 3                        ;是本段重定位数据第一项吗
003E021B  jnz     short 003E0227
003E021D  inc     esi                          ;指向下一个数据
003E021E  add     ebx, dword ptr [esi]         ;得到需要重定位项目的地址
003E0220  add     dword ptr [ebx], edi         ;进行重定位
003E0222  add     esi, 4
003E0225  jmp     short 003E022C
003E0227  inc     esi
003E0228  add     ebx, eax                     ;得到需要重定位项目的地址
003E022A  add     dword ptr [ebx], edi         ;进行重定位
003E022C  movzx   eax, byte ptr [esi]          ;从外壳转储的重定位表结构取数据
003E022F  or      al, al
003E0231  jnz     short 003E0219
003E0233  push    dword ptr fs:[30]
```

接下来在上面代码中找到一个点，将需要重定位的RVA取出来，经分析，3E022F这个点比较合适，

执行到这句，EBX 寄存器保存的就是需要重定位的地址。补丁的思路是找块代码空间，跳过去执行补丁代码，补丁代码可能是将重定位的地址转成 RVA，并保存下来。本例选取 3E0289 这个地址放补丁，注意此处并不是代码空白处，是存放了外壳的一些参数，当执行 3E01FC 这句后，这段数据外壳已不用了。因此当 OllyDbg 第一次来到 3E022F 这句时，键入如下的补丁指令：

```
003E022C  movzx   eax, byte ptr [esi]
003E022F  jmp     short 003E0289              ;补丁此处
003E0231  jnz     short 003E0219
```

然后在 3E0289h 处键入如下补丁代码：

```
003E0289  pushad                              ;保存各寄存器的值
003E028A  mov     edx, dword ptr [3F0000]     ;从全局变量 3F0000h 取一地址指针
003E0290  sub     ebx, E20000                 ;减外壳基址，将 ebx 中的地址转成 RVA
003E0296  mov     dword ptr [edx], ebx        ;将获得的 RVA 保存下来
003E0298  add     edx, 4                      ;指向下一个 DWORD 地址
003E029B  mov     dword ptr [3F0000], edx     ;将指针保存到全局变量中
003E02A1  popad                               ;恢复各寄存器的值
003E02A2  or      al, al                      ;原外壳的指令
003E02A4  jmp     short 003E0231              ;跳回外壳代码
```

这段补丁代码读者必须根据本机情况调整一些参数，例如 E20000h 这是外壳被加载后的基址，3F0000h 这个地址是 OllyDbg 的插件 HideOD 分配的，如图 13.49 所示。

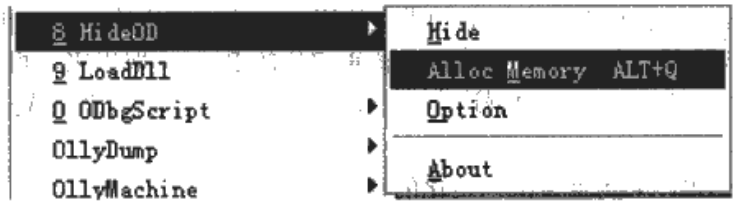


图 13.49　利用 OllyDbg 插件分配临时空间

这个分配空间的地址是随机的，读者系统环境可能会分配到其他值。在 3F0000h 地址处键入 3F0010h，这个地址用来存放获得的重定位 RVA，如图 13.50 所示。

```
003F0000 10 00 3F 00 00 00 00 00 00 00 00 00 00 00 00 00 ..?.............
003F0010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

图 13.50　利用一个全局变量当地址指针

补丁代码键入完成后，外壳在处理重定位相关代码时，这段补丁代码将需要重定位的 RVA 全部提取出来，执行后效果如图 13.51 所示。

```
003F0000 F0 00 3F 00 00 00 00 00 00 00 00 00 00 00 00 00 ??..............
003F0010 00 00 00 00 1D 10 00 00 31 10 00 00 6E 10 00 00 ....■■..1■..n■..
003F0020 8D 10 00 00 A1 10 00 00 DE 10 00 00 F8 10 00 00 ?..?..?..?..?..
003F0030 09 11 00 00 0F 11 00 00 13 11 00 00 18 11 00 00 .■..■■..■■..■■..
```

图 13.51　获得需要重定位地址的 RVA

从 3F0014h 开始就是需要重定位代码的 RVA，每个地址占用一个 DWORD 字节。切换到 OllyDbg 的数据窗口，将 3F0014h~3F0AF8h 这段需要重定位的 RVA 复制出来（选取数据时，最后一个 DWORD 数据是 0），操作时单击鼠标右键，执行菜单“Binary/Binary copy”（二进制复制）功能，再运行 WinHex，新建一文档，将这段二进制数据粘贴进去，粘贴时，选择“ASCII Hex”模式（见图 13.52），然后将提取的数据保存为 Relo.bin。Relo.bin 中就是需要重定位的地址，以 RVA 表示。部分数据如下：

```
0000101D
00001031
0000106E
0000108D
000010A1
.....
```

ReloREC 这款工具，就是根据这些 RVA 重新生成一份新的重定位表。准备工作完成后，运行 ReloREC，将 Relo.bin 拖放到 ReloREC 主界面上可打开此文件，如图 13.53 所示。在“Relocation’ s RVA”域里填上原始重定位表的 RVA 地址，本例为 C000h，最后单击“Fix Dump”按钮，打开上节刚修复输入表的 dumped_.dll 文件，即可完成重定位表的修复。

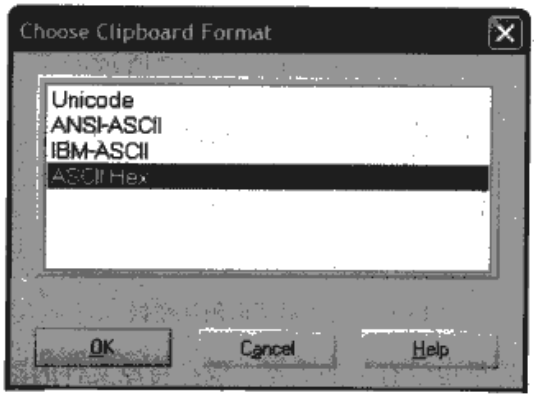


图 13.52 WinHex 里以 ASCII Hex 粘贴

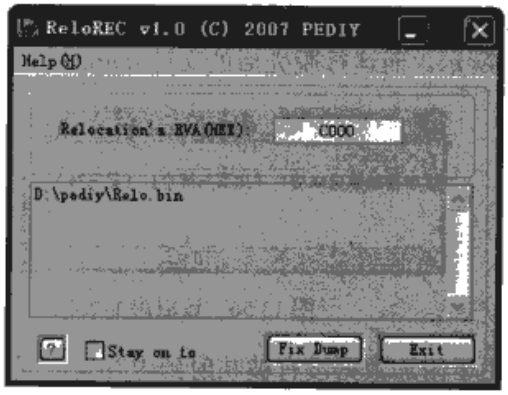


图 13.53 ReloREC 工具界面

13 附加数据

某些特殊的 PE 文件在各个区块的正式数据之后还有一些数据，这些数据不属于任何区块。由于 PE 文件被映射到内存是按区块映射的，因此这些数据是不能被映射到内存中的，这些额外的数据称为附加数据（overlay）。

附加数据的起点可以认为是最后一个区块的末尾，终点是文件末尾。用 LordPE 查看实例 overlay.exe 的区块，如图 13.54 所示。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
UPX0	00001000	00008000	00000400	00000000	E0000080
UPX1	00009000	00003000	00000400	00002E00	E0000040
.rsrc	0000C000	00001000	00003200	00000600	C0000040

图 13.54 查看区块信息

从图 13.54 可以计算出最后一个区块末尾的文件偏移值为 3200h+600h=3800h。用十六进制工具打开目标文件，跳到 3800h，会发现后面还有一段数据，这就是附加数据，如图 13.55 所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000037E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000037F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00003800	B6	C1	C8	A1	B8	BD	BC	D3	CA	FD	BE	DD	B3	C9	B9	A6	读取附加数据成功
00003810	A3	A1	A1	B6	BC	D3	C3	DC	D3	EB	BD	E2	C3	DC	A1	B7	！《加密与解密》
00003820	A3	A8	B5	DA	C8	FD	B0	E6	A3	A9	20	77	77	77	2E	70	（第三版） www.p
00003830	65	64	69	79	2E	63	6F	6D	00	00	00	00	00	00	00	00	ediy.com.....

图 13.55 附加数据

用 PEiD 分析实例 overlay.exe，会给出结果“Nothing found [Overlay] *”，其中 Overlay 就表明有附加数据存在。带有附加数据的文件脱壳时，必须将附加数据粘贴回去，如果文件有访问附加数据的指针，也要修正。

本节实例 overlay.exe 实际是用 UPX 加壳了，由于附加数据的存在，干扰了 PEiD 分析。用 OllyDbg 打开实例，来到 OEP 处。

```
00401436    55              push    ebp
00401437    8BEC            mov     ebp, esp
00401439    6A FF           push    -1
```

此时，抓取内存镜像保存到磁盘中，然后用 ImportREC 重建输入表，最终文件为 dumped_.exe。

运行实例原文件，然后单击菜单“File/Open”，程序将会读取附加数据并在编辑框中显示出来，如图 13.56 所示。而运行脱壳后的文件 dumped_.exe，不能将原来的文字显示出来，如图 13.57 所示。

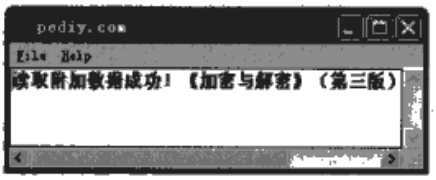


图 13.56　读取附加数据

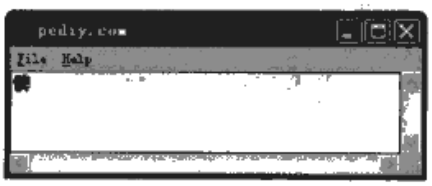


图 13.57　脱壳后读取附加数据

由于附加数据没有被映射到内存里，因此抓取的映像文件里也没有附加数据。现在将原文件的附加数据移到脱壳后的文件里。用十六进制工具打开 overlay.exe，将 3800h 后的附加数据追加到 dumped_.exe 文件末尾 E000h 处。

运行已有附加数据的 dumped_.exe，但执行“File/Open”仍不能正确读取数据。用 OllyDbg 分析一下实例是如何读取自身附加数据的。用 CreateFileA 设断，执行“File/Open”功能后，会中断到这段代码处。

```
00401040  push    ecx
00401041  push    0
00401043  call    dword ptr [<GetModuleFileNameA>]     ;取自身文件名
00401049  push    0
......
00401062  call    dword ptr [<&CreateFileA>]           ;打开自身
00401068  mov     dword ptr [ebp-11C], eax
......
004010DC  push    0
004010DE  push    0
004010E0  push    3800                                 ;注意这个值
004010E5  mov     edx, dword ptr [ebp-11C]
004010EB  push    edx
004010EC  call    dword ptr [<SetFilePointer>]         ;移动读写指针
......
0040110E  call    dword ptr [<&ReadFile>]
00401127  mov     ecx, dword ptr [ebp-108]
0040112D  push    ecx
0040112E  mov     edx, dword ptr [ebp-10C]
00401134  push    edx
00401135  call    dword ptr [<SetWindowTextA>]         ;将附加数据显示到文本框里
```

用 CreateFileA 打开一个文件后，文件指针默认是指向文件的第一个字节的。程序用 SetFilePointer 设置指针，指向附加数据，然后用 ReadFile 将附加数据读取出来。这里 SetFilePointer 函数比较关键，其原型如下：

```
DWORD SetFilePointer(
 HANDLE hFile,                    // 文件句柄
 LONG lDistanceToMove,            // 移动的距离，这个是低 32 位
 PLONG lpDistanceToMoveHigh,      // 移动的距离，这个是高 32 位
 DWORD dwMoveMethod               // 移动方式
);
```

由于脱壳后，文件大小发生变化，追加后的附加数据地址已改变，此处变为 E000h，因此需要修正 SetFilePointer 的参数，将其指向附加数据，。

```
004010DC  push    0
004010DE  push    0
004010E0  push    0E000                          ;将此处指向附加数据 E000h
004010E5  mov     edx, dword ptr [ebp-11C]
004010EB  push    edx
004010EC  call    dword ptr [<&kernel32.SetFilePointer>]
```

也就是说，对于带有附加数据的程序，抓取内存映像后，必须将附加数据追加到脱壳文件的最后，同时修正读取附加数据的相应指针。

13.7 PE 文件的优化

一般脱壳没有将外壳本身的代码去除，资源也没有完全释放，此时脱壳后的程序可能会比原始程序大。虽然脱壳后的文件能正常运行，但在汉化一些场合下可能会遇到问题，比如一些汉化工具不识别脱壳后的文件，或某些功能无效等。本节继续通过实例 RebPE 来讲解手工优化文件。

1．优化输入表存放位置

OllyDbg 加载实例 RebPE，来到 OEP 后，运行 LordPE 将内存映像抓取出来，另存为 dumped.exe。运行 LordPE 查看 dumped.exe 的区块信息，如图 13.58 所示。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00004000	00001000	00004000	E0000020
.rdata	00005000	00001000	00005000	00001000	C0000040
.data	00006000	00003000	00006000	00003000	C0000040
.rsrc	00009000	0000A000	00009000	0000A000	C0000040
.pediv	00013000	00007000	00013000	00007000	E0000040

图 13.58　查看区块

接下来的步骤一般是用 ImportREC 重建输入表，默认是将生成的输入表存放在新增的区块上。要使脱壳完美些，尽可能将新输入表存放在原输入表地址处，这需要读者熟悉常见编译器的输入表存放位置，本例是 Visual C++ 6.0 SDK 编译的程序，输入表一般存放在.rdata 区块上。用十六进制工具查看 dumped.exe 文件的.rdata 内容，选取一地址以存放输入表，这个地址必须以 DWORD 对齐。经查看，发现 40545Ch 开始有一大段空白，其空间大小可以存放新的输入表，并且 VC 编译器生成的.rdata 区块内容是只读，因此不用担心当程序运行时，这段空白会有数据写入。运行 ImportREC，获取输入表后，在新建输入表信息中 RVA 填 545C，如图 13.59 所示。最后单击“Fix Dump”按钮，选择 dumped.exe 进行修复，得到 dumped_.exe。

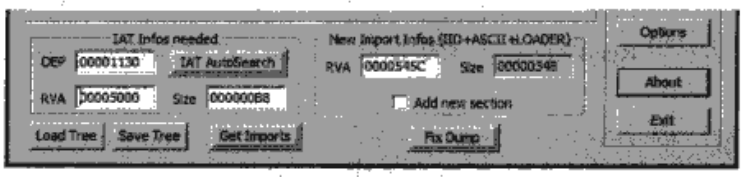


图 13.59　指定输入表的存放地址

2．资源的重建

有些软件脱壳后的资源不可查看、编辑或能编辑但保存不了，这是因为脱壳后，资源没完全释放。在“16.2.7 资源数据处理”这节中描述了外壳如何处理特殊资源，如 Icon（图标）、Group icon（组图标）等，这些图标在程序没有被执行时仍然会被系统读取，它们一般是不能压缩的，所以被存放在外壳本身的代码空间里。正常脱壳后，资源段的其他数据已恢复，但图标等资源还留在外壳里。资源重建，就是把这些资源移回.rsrc 区块里。

实例 RebPE.exe 脱壳后并修复输入表后的文件为 dumped_.exe，用 ResFixer 打开 dumped_.exe 文件，如图 13.60 所示。图中显示为红的资源部分位于外壳代码里，如 Icon、Group icon 都在.pediy 区块里。现在，把这些资源移回到.rsrc 区块里，这类资源修复工具有多种，常用的有 DT_ResFix、freeRes 等。

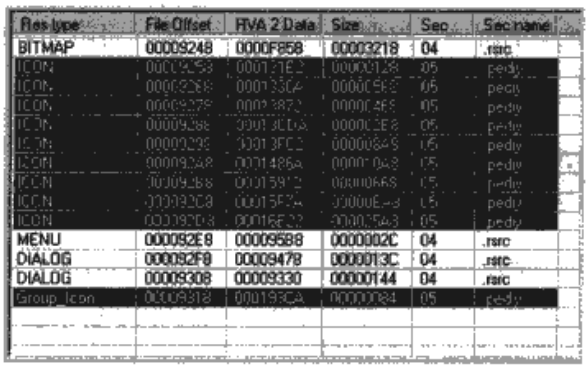


图 13.60　查看资源

运行 DT_ResFix，打开 dumped_.exe，单击“Fix Resource”按钮，它将分布在多个节里的资源重新移到一个资源节里，并且对资源进行修复优化。重新建立后的资源，用其他资源编辑工具（eXeScope 等）就可正常处理了。也可单击“Dump”标签，将重建的资源模块提取出来，如图 13.61 所示。在“Res File”框里设置好生成文件的路径及文件名，在“NewRVA”框中设置新资源段的 RVA，本例是取.rsrc 区块的 RVA，“FileAligment”对齐值填 1000h。设置好后单击“Dump Resource”按钮来 Dump 资源，文件被保存为 rsrc.bin。

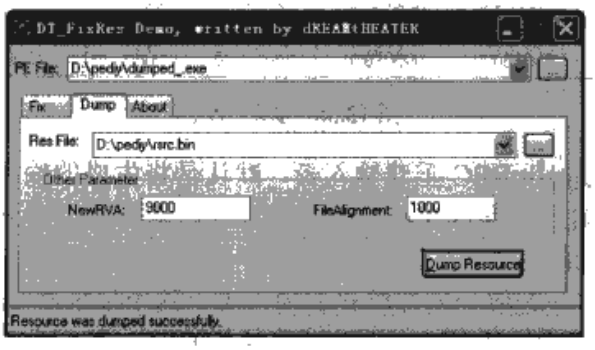


图 13.61　重建资源

3．装配文件

现在进行区块调整，PE 编辑工具选择 LordPE，使用前设置一下 LordPE 的 Options，如图 13.62 所示。

将“Section Table:autofix SizeOfImage”选上，这个功能是自动修正 SizeOfImage 大小，在增减区块时比较方便。但其自动纠正功能，偶尔也会给使用带来不便，如用其打开某些驱动文件时，LordPE 会擅自纠正其 SizeOfImage 值，这会导致文件的 Checksum 校验和不正确，从而系统认为驱动文件损坏。

设置好后，用 LordPE 的 PE 编辑器打开 dumped_.exe，单击“Sections”按钮，进入区块编辑功能中。.pediy 区块是外壳的代码，脱壳后不需要，可以删除。.rsrc 区块是资源所在区块，上一节已重建新的资源数据，可以删除。在要删除的区块上单击右键，执行“Wipe section header”功能删除区块，但这种删除仅是删除 PE 头中的数据，对区块的具体内容再用相关的十六进制工具删除。另一款 PE 工具 CFF Explorer 能自动删除相关数据，比较方便。

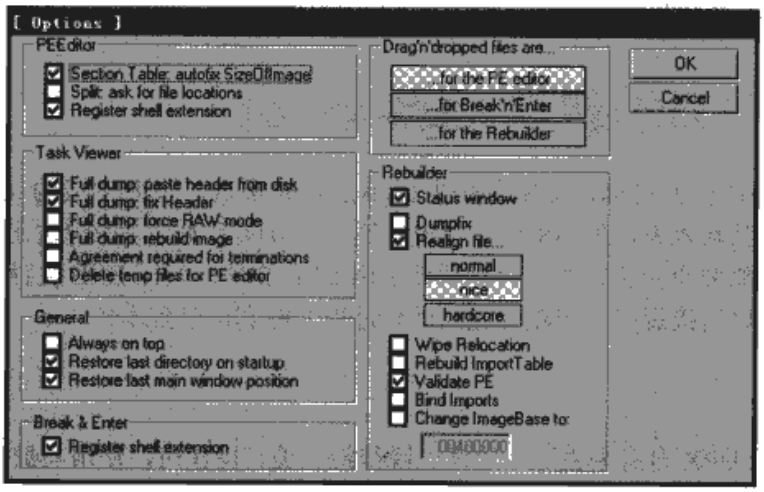
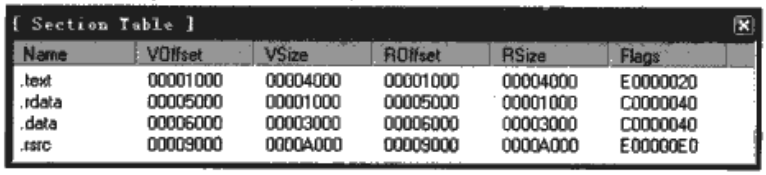


图 13.62　LordPE 的选项设置

删除.pediy 和.rsrc 区块及相应数据后，单击右键，执行“Load section from disk”功能，选中新的资源文件 rsrc.bin，将其导入，优化后的区块如图 13.63 所示。



Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00004000	00001000	00004000	E0000020
.rdata	00005000	00001000	00005000	00001000	C0000040
.data	00006000	00003000	00006000	00003000	C0000040
.rsrc	00009000	0000A000	00009000	0000A000	E00000E0

图 13.63　装配后的区块信息

4. 修正 PE 文件头

用 LordPE 查看 PE 头，几个重要的 PE 字段要修复，主要是 EntryPoint、BaseOfCode 和 BaseOfData，如图 13.64 所示。

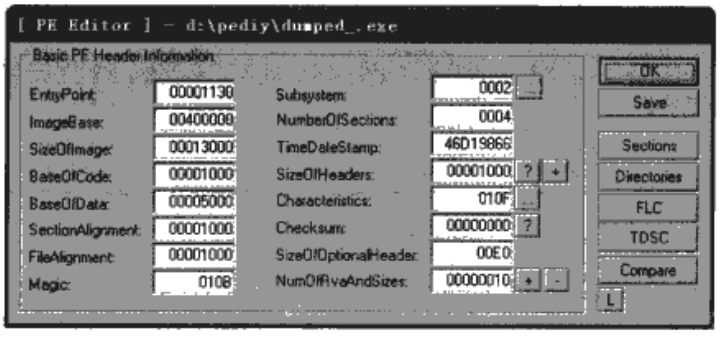


图 13.64　修正 PE 头信息

- EntryPoint：即脱壳时的 OEP，一般 ImportREC 会自动修正。
- BaseOfCode：代码段的起始 RVA。一般是第一个区段（本例是.text 段）的 RVA，所以这里应该填 1000h。
- BaseOfData：数据段的起始 RVA。一般指除了代码外的部分开始的 RVA，本例就是.rdata 区块的 RVA5000h。
- SizeOfImage：指装入文件从基址到最后一个块的大小，最后一个块根据其大小往上取整。一般工具会自动纠正这个值。

也可用 LordPE 的 Rebuild PE 功能重建程序，偶尔情况下，重建后用来汉化可能会出错。另外，一些优化工具，如 PE Optimizer 也可选用。

13.8 压缩壳

压缩壳以减小文件体积为目标，加密保护方面不是它的重点，因此生成的 IAT 都是没加密的，用 ImportREC 可轻易重建其输入表，如 ASPack、UPX 等。本节以手动方式探讨这几种壳的调试技巧，如获得 OEP、修复输入表、修复重定位表等。目标软件采用 DLL 文件，相比 EXE 文件多了一个重定位表需要处理。虽有许多工具可以直接脱壳，但跟踪壳的处理过程才能真正学到本领。

13.8.1 UPX 外壳

UPX 外壳可以使用 UPX 自身来去除，这样壳脱得最完美。操作时，使用与加壳所用版本相同的 UPX 脱壳，或选用更高版本的 UPX。

脱壳命令是：

```
UPX -d 文件名
```

为了阻止 UPX 脱其本身的壳，一些保护工具 UPXPR 和 UPX-Scrambler 对加壳文件进行处理，使得 UPX -d 命令失效。解决方法是恢复被破坏的文件或手动脱壳。

1. UPXPR 保护

UPXPR 修改了一些 UPX 加壳的标志，修复这些标识即可重新用 UPX -d 参数脱壳。光盘映像文件中提供的 UPXPR_notepad.exe 是被 UPXPR 处理过的记事本程序，用 UPX -d 命令脱壳提示“CantUnpack Exception: file is modified/hacked/protected;take care!!!”。

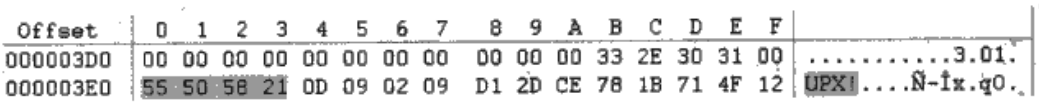
用 LordPE 打开该软件，查看区块信息。一般被 UPXPR 处理过的，其第一、二个块的块名不是 UPX0 和 UPX1，而是其他字符。所以第一步就是恢复两个块名，将第一个块名改为 UPX0，第二个块名改为 UPX1。修改方法是：先选中块，单击鼠标右键，选择“edit section header”项目，在“Name”框中输入新的块名（如 UPX0），如图 13.65 所示。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
UPX0	00001000	0000A000	00000200	00000000	E0000080
UPX0	0000B000	00004000	00000200	00003C00	E0000040
.rsrc	0000F000	00002000	00003E00	00001200	C0000040

图 13.65 查看区块表

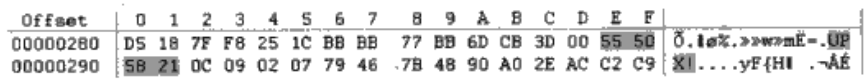
如果用十六进制工具打开没处理过的 UPX 外壳，查看加壳 UPX 的版本号，在版本后有一 UPX 加壳标志“UPX!”，如图 13.66 所示。UPXPR 会将此标志删除，导致 UPX 不能解压。



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000003D0	00	00	00	00	00	00	00	00	00	00	00	33	2E	30	31	00	.....3.01.
000003E0	55	50	58	21	0D	09	02	09	D1	2D	CE	78	1B	71	4F	12	UPX!....Ñ-Îx.qO.

图 13.66 查看 UPX 版本号

UPX 0.9x～1.2x 各版本在标志 UPX!后面的 4 个字节均为 0C 09 ?? ??，更高版本是 0D 09 ?? ??，可以用这个特殊字节来定位 UPX 标志位置，找到后再把它前面的 4 个字节改为 UPX!（见图 13.67）。



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000280	D5	18	7F	F8	25	1C	BB	BB	77	BB	6D	CB	3D	00	55	50	Õ.ø%.»»w»mË=.UP
00000290	58	21	0C	09	02	07	79	46	7B	48	90	A0	2E	AC	C2	C9	X!....yF{H .¬ÂÉ

图 13.67 恢复 UPX 加壳标志

这样，恢复 UPX!、UPX0 和 UPX1 标志后，就可用 UPX –d 命令对此文件脱壳了。

在 0C 09 ?? ??这 4 个字节之后还有 24 个字节，修改其中的任何一个字节都会使 UPX 无法解压，所以如果不知道这些正确的数值就会无法恢复。

另一款工具 UPXFIX_by_DiKeN 可以很好地修复处理过的 UPX 外壳，它可以重构 UPX 外壳，处理后，用原版 UPX –d 命令脱壳。

2. 手动脱 UPX 的壳

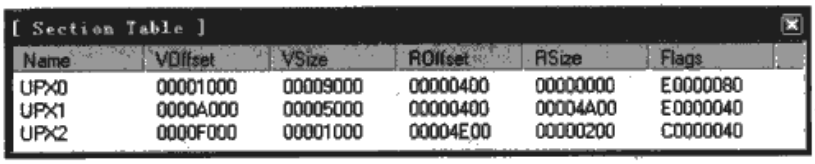
UPX 壳既破坏了输入表也破坏了重定位表，尽量使用其自身命令脱壳，实在没办法了再尝试手动脱壳。

用 UPX v3.01 将 EdrLib.dll 文件加壳，用 PE 工具查看其 PE 信息。

EntryPoint：E640h

ImageBase：400000h

再查看其区块信息，如图 13.68 所示。UPX 加壳后已将区块重新组织，分别是 UPX0、UPX1、UPX2 等。其中 UPX0 的 Raw Size 是 0，UPX 是将解压缩后的原始文件数据映射到此区块中。UPX 的解压缩执行代码在 UPX1 里，被压缩的原始数据放在 UPX1 和 UPX2 中。



[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
UPX0	00001000	00009000	00000400	00000000	E0000080
UPX1	0000A000	00005000	00000400	00004A00	E0000040
UPX2	0000F000	00001000	00004E00	00000200	C0000040

图 13.68 查看区块表

针对 UPX 的壳，装载后，可以不跟踪，在代码窗口里一直往下翻屏，就能发现类似下面的跳转代码，一个跨段指令转到 OEP，如图 13.69 所示。

```
003DE7F6   .  61            popad
003DE7F7   .  8D4424 80     lea     eax, dword ptr [esp-80]
003DE7FB   >  6A 00         push    0
003DE7FD   .  39C4          cmp     esp, eax
003DE7FF   .^ 75 FA         jnz     short 003DE7FB
003DE801   .  83EC 80       sub     esp, -80
003DE804   >-E9 372AFFFF    jmp     003D1240                    OEP的RVA=1240H
```

图 13.69 跳到 OEP

由于 DLL 重定位，此时对内存操作的指令被修改了。例如这句：

```
003D1266    A1 58B43D00        mov     eax, dword ptr [3DB458]
```

为了得到与加壳前一样的文件，必须找到重定位的代码，跳过它，让其不被重定位。重新加载 DLL，对上句重定位的地址 3D1267h 下内存写断点，中断几下，就可来到重定位的处理代码处。

```
003DE79E  mov     al, byte ptr [edi]        ;指向UPX自己加密过的重定位表
```

```
003DE7A0  inc     edi                          ;指针移向下一位
003DE7A1  or      eax, eax                     ;EAX=0? 结束标志
003DE7A3  je      short 003DE7C7
003DE7A5  cmp     al, 0EF
003DE7A7  ja      short 003DE7BA
003DE7A9  add     ebx, eax                     ;EBX 的初值为（0xFFC+基址）
003DE7AB  mov     eax, dword ptr [ebx]         ;EBX 指向需要重定位的数据，取出放到 EAX
003DE7AD  xchg    ah, al
003DE7AF  rol     eax, 10
003DE7B2  xchg    ah, al
003DE7B4  add     eax, esi                     ;ESI 指向 UPX0 区块的 VA，本例=3D1000
003DE7B6  mov     dword ptr [ebx], eax         ;重定位
003DE7B8  jmp     short 003DE79C
003DE7BA  and     al, 0F
003DE7BC  shl     eax, 10
003DE7BF  mov     ax, word ptr [edi]
003DE7C2  add     edi, 2
003DE7C5  jmp     short 003DE7A9
003DE7C7  mov     ebp, dword ptr [esi+E044 ;改好 ESI 为 401000 后，按 F4 键到这里
```

UPX 壳已将原基址重定位表清零，重定位操作时，使用其自己的重定位表。地址 3DE7B4h 处 ESI 指向 UPX0 区块的 VA，本例为 3D1000h，为了让代码以默认 ImageBase 的值 400000h 重定位代码，可以在这句强制将 ESI 的值改为 401000h。来到这句后，双击 ESI 寄存器，改成 401000h，然后按 F4 键来到 3DE7C7h 这里。此时代码段的数据没被重定位：

```
003D1253    833D 68AD4000 00    cmp    dword ptr [40AD68], 0
```

此时可以 Dump 了。运行 LordPE 抓取 DLL 映像，并保存为 upx_dumped.dll。

运行 ImportREC，手工判断 DLL 的 IAT 位置和大小并填上，最后单击“Get Imports”按钮，重建输入表后的文件保存为 upx_dumped_.dll，如图 13.70 所示。

图 13.70　ImportREC 重建输入表

接下来用 ReloREC 工具构造一份新的重定位表，首先将 UPX 外壳这些要重定位的 RVA 提取出来。在处理重定位代码语句中，下面这句就是对代码重定位，其中 EBX 保存的就是要重定位的地址。

```
003DE7B6    mov    dword ptr [ebx], eax        ;EBX 指向要重定位的 RVA
```

补丁的思路是找块代码空间，跳过去执行补丁代码，将重定位的地址转成 RVA，并保存下来。如下语句跳到补丁代码处：

```
003DE7B8  jmp     short 003DE80A
```

补丁代码：

```
003DE80A  pushad
003DE80B  mov     edx, dword ptr [3E0000]     ;从全局变量 3E0000h 取一地址指针
003DE811  sub     ebx, 3D0000                 ;减外壳基址，将 ebx 中的地址转成 RVA
003DE817  mov     dword ptr [edx], ebx        ;将获得的 RVA 保存下来
003DE819  add     edx, 4                      ;指向下一个 DWORD 地址
003DE81C  mov     dword ptr [3E0000], edx     ;将指针保存到全局变量中
```

```
003DE822   popad
003DE823   jmp     003DE79C                          ;跳回外壳代码
```

3E0000h 这个地址是 OllyDbg 的插件 HideOD 临时分配的，其初始值设为 3E0010h，如图 13.71 所示。

图 13.71 分配空间保存重定位的 RVA

执行完补丁代码，数据窗口将保存需要重定位的 RVA，执行菜单“Binary/Binary copy”功能将数据复制出来，并用 WinHex 将提取的数据保存为 Relo.bin。然后在 dumped_.dll 里找一块空白代码处保存重定位表（一般在 UPX1 或 UPX2 区块里找），在这里选择 C000h 处。最后，参照 13.5.4 节，用 ReloREC 完成重定位表的修复。

13.8.2 ASPack 外壳

ASPack 外壳运行时，曾有一段时间将程序完全解密，此时内存映像是加壳前的状态，输入表、重定位表都是完整存在的，没有被破坏。正是由于 ASPack 保留了加壳前程序完整的状态，因此其兼容性极好。ASPack 脱壳很简单，适机抓取内存映像，再修正 PE 头的输入表、重定位表的地址即可。

用 ASPack 2.12 将 EdrLib.dll 文件加壳，查看其 PE 信息。

EntryPoint：D001h

ImageBase：400000h

1. 寻找 OEP

寻找 DLL 的 OEP 有两条路可以走：一是载入时找，二是在退出时找。本节采用第二种方法。

OllyDbg 加载 EdrLib.dll，中断在外壳代码第一行。代码如下：

```
003DD001  pushad                                    ;在此处设断
003DD002  call    EdrLib.003DD00A
```

在外壳的入口点按 F2 键设一个断点，按 F9 键让 EdrLib.dll 运行，DLL 装载成功后，关闭 loaddll.exe 界面，即会卸载 DLL 文件，将再次中断在外壳的入口点处。

```
003DD001  pushad
003DD002  call    003DD00A                          ;请按 F7 键
003DD007  nop                                       ;原来是花指令，此处 NOP 方便显示
003DD008  jmp     short 003DD00E
003DD00A  pop     ebp
003DD00B  inc     ebp
003DD00C  push    ebp
003DD00D  retn                                      ;此处返回到 003DD008
003DD00E  call    003DD014                          ;请按 F7 键
003DD013  nop                                       ;原来是花指令，此处 NOP 方便显示
003DD014  pop     ebp
003DD015  mov     ebx, -13
003DD01A  add     ebx, ebp
003DD01C  sub     ebx, 0D000
003DD022  cmp     dword ptr [ebp+422], 0
003DD029  mov     dword ptr [ebp+422], ebx          ;保存当前基址
003DD02F  jnz     003DD39A                          ;第二次进入入口后，就跳转
```

注意，在“CALL 003DD00A”语句处要按 F7 键跟进，不然按 F8 键程序就会运行起来。这里的 call 语句不是什么真正的过程调用，无非是变形的 jmp 跳转语句而已。要识别它也不难，看看它跳转的地址是

否就在附近。如果是就按 F7 键，而不要按 F8 键。

由于 DLL 已解压，因此外壳不会再次对 DLL 文件解压缩，3DD02Fh 一行将跳过外壳解压代码直接到 OEP 处。跟踪 EXE 文件时，也可直接在 DD02Fh 这一行跳转来定位到 OEP 的处理代码处。

```
003DD39A  mov     eax, 1240                          ;此值为 OEP 的 RVA
003DD39F  push    eax
003DD3A0  add     eax, dword ptr [ebp+422]
003DD3A6  pop     ecx
003DD3A7  or      ecx, ecx
003DD3A9  mov     dword ptr [ebp+3A8], eax           ;计算出的 OEP 放到 3DD3BA
003DD3AF  popad                                      ;恢复现场环境
003DD3B0  jnz     short 003DD3BA
003DD3B2  mov     eax, 1
003DD3B7  retn    0C
003DD3BA  push    003D1240
003DD3BF  retn                                       ;跳到 OEP
```

EdrLib.dll 装载后，基地址不是默认的 400000h，新的基地址是 3D0000h。此时的 OEP 的 RVA 值为 1240h。

2. 解压分析

用 LordPE 查看 EdrLib.dll 的区块信息（见图 13.72）。ASPack 加壳时没合并区块，各区块的 RVA 仍与加壳前一样。.aspack 与.adata 是外壳的执行程序和数据，脱壳后可以去除。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00006000	00000600	00003800	C0000040
.rdata	00007000	00001000	00003E00	00001000	C0000040
.data	00008000	00004000	00004E00	00000600	C0000040
.reloc	0000C000	00001000	00005400	00000600	C0000040
.aspack	0000D000	00002000	00005A00	00001200	C0000040
.adata	0000F000	00001000	00006C00	00000000	C0000040

图 13.72　区块信息

ASPack 外壳依次将.text、.rdata、.data、.reloc 区块解压，并放到正确的位置上。当这些区块解压结束后，内存映像是加壳前的原始状态，外壳还没有来得及进行进一步处理，这个时候正是得到完整映像文件的好时机。

由于外壳会向区块里写数据，可以对区块地址设内存断点。.text 区块的 RVA 为 1000h，加上映像基址，本例结果为 3D1000h。在数据窗口中，对此地址下内存写断点，同时监视 3D1000h 内存数据变化。中断几次，会发现 3D1000h 内存数据已还原。此时代码如下：

```
003DD16F  mov     edi, dword ptr [esi]
003DD171  add     edi, dword ptr [ebp+422]           ;EDI 指向区块地址(VOffset)
003DD177  mov     esi, dword ptr [ebp+152]           ;ESI 指向已还原的数据
003DD17D  sar     ecx, 2                             ;ECX 是区块数据大小(VSize)
003DD180  rep     movs dword es:[edi], dword [esi] ;将 ESI 指向数据复制到 EDI
003DD182  mov     ecx, eax
003DD184  and     ecx, 3
003DD187  rep     movs byte ptr es:[edi], byte ptr [esi]
003DD189  pop     esi
003DD18A  push    8000
003DD18F  push    0
003DD191  push    dword ptr [ebp+152]
003DD197  call    dword ptr [ebp+551]
003DD19D  add     esi, 8
003DD1A0  cmp     dword ptr [esi], 0
```

```
003DD1A3  jnz     003DD0C7
003DD1A9  push    8000                                ;可在此设断，抓取映像文件
```

上面代码的作用是 ASPack 外壳将已还原的区块数据放回将要执行的区块空间里，会循环执行数次，直到所有的块都还原后，内存中是完整的原程序，此时可将内存映像抓取下来。操作时，只需要在 3DD1A9h 一行设断，中断后抓取内存映像文件，保存为 dumped.dll。

3. 输入表

由于 ASPack 外壳还没来得及破坏输入表，只要找到输入表的地址即可。根据 API 函数的调用，确定 IAT 的 RVA 是 7000h～70E4h。重新加载 EdrLib.dll，在 IAT 里任意选一地址设置内存写断点。中断如下：

```
003DD376  mov     dword ptr [edi], eax          ;填充 IAT
003DD378  add     dword ptr [ebp+549], 4
003DD37F  jmp     003DD2B6
```

根据跳转，向上来到输入表处理的代码处：

```
003DD278  mov     esi, 7694                     ;输入表的 RVA
003DD27D  mov     edx, dword ptr [ebp+422]      ;映像基址
003DD283  add     esi, edx                      ;转成虚拟地址
003DD285  mov     eax, dword ptr [esi+C]        ;取 IID 中 Name 的 RVA
003DD288  test    eax, eax
003DD28A  je      003DD39A
003DD290  add     eax, edx                      ;加上基址
003DD292  mov     ebx, eax
003DD294  push    eax
003DD295  call    dword ptr [ebp+F4D]           ;GetModuleHandleA
```

从 3DD278h 这句可得知输入表 RVA 为 7694，大小可以用十六进制工具查看 dumped.dll。根据 IID 结构，很容易得知大小为 3Ch。

为了进一步巩固输入表知识，讲述另一种方法确定输入表的地址。dumped.dll 文件里输入表是完整存在的，因此以 kernel32.dll 为突破口，反推 IID 结构的地址。用十六进制工具打开 dumped.dll 文件，查找“KERNEL32.dll”字符串（因为一般程序输入表中肯定存在此字符）。会发现有三处，第一处是原程序的输入表（见图 13.73），第二、三处是在外壳的代码里（即在.aspack 块里）。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000077D0	4B	45	52	4E	45	4C	33	32	2E	64	6C	6C	00	00	55	53	KERNEL32.dll..US
000077E0	45	52	33	32	2E	64	6C	6C	00	00	05	02	54	65	78	74	ER32.dll....Text

图 13.73 显示 KERNEL32.dll 字符串

因为 dumped.dll 文件是内存映像，所以其 RVA 值与文件偏移值相等。图 13.73 显示的 KERNEL32.dll 的地址为 77D0h，该地址是 IID 结构中 Name 项的值，而 Name 值存在形式为“D0770000”。然后以 Hex Values 模式查找十六进制，结果如图 11.74 所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00007690	00	00	00	00	E4	76	00	00	00	00	00	00	00	00	00	00	....鋲..........
000076A0	D0	77	00	00	14	70	00	00	D0	76	00	00	00	00	00	00	Ðw..p..Ðv......
000076B0	00	00	00	00	32	78	00	00	00	70	00	00	00	00	00	00	....2x...p......
000076C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000076D0	F6	77	00	00	1A	78	00	00	0E	78	00	00	EA	77	00	00	鯷...x...x..豚..

图 13.74 IID 结构

上面显示的就是输入表 IID 数组，共有两个数组，如表 13-5 所示。

表 13-5　十六进制工具中显示的 IID 数组

OrignalFirstThunk	TimeDateStamp	ForwardChain	Name	FirstThunk
E476 0000	0000 0000	0000 0000	D077 0000	1470 0000
D076 0000	0000 0000	0000 0000	3278 0000	0070 0000

第一个 IID 数组地址就是输入表的地址，结果为 7694h。

4. 基址重定位表

EXE 文件一般不需要重定位表，这步可略过。ASPack 没破坏重定位表，因此只需要确定重定位表的地址和大小即可。

进入 OEP 后，寻找一句需要重定位的地址。

```
003D1266  A1 58843D00    mov     eax, dword ptr [3DB458]
```

地址 3D1267h 会被重定位，重新加载 DLL 后，对 3D1267h 设置内存写断点。中断如下：

```
003DD1E5  mov     esi, dword ptr [ebp+539]      ;取重定位表的 RVA
003DD1EB  add     esi, dword ptr [ebp+422]
003DD1F1  cmp     dword ptr [esi], 0
003DD1F4  je      short 003DD257
……
003DD247  add     dword ptr [edi+ebx], edx      ;重定位
003DD24A  jmp     short 003DD24C
003DD24C  or      word ptr [esi], 0FFFF
003DD250  add     esi, 2
003DD253  loopd   short 003DD209                ;循环
003DD255  jmp     short 003DD1F1
003DD257  mov     edx, dword ptr [ebp+422]      ;重定位处理结束，此时 ESI 是其结束地址
```

外壳程序在 3DD1E5h 开始模拟 Windows 系统重定位代码，此时 ESI 的值就是重定位表的起始 RVA，本例为 C000h。这段初始化代码以 ESI 为指针，取重定位表的数据，当执行结束后，来到 3DD257h，此时的 ESI 中的值是重定位表的结束地址，本例为 3DC5C0h，转成 RVA 为 C5C0h，因此重定位表的大小为 5C0h。

如果熟悉重定位表，可以用十六进制工具直接查看 dumped.dll 文件，以确定重定位表的地址和大小。重定位表一般以“00100000”开始，并且在十六进制工具中右边的字符栏显示的是可见的 ASCII 字符，因此很好辨认。

5. PE 文件修正

（1）OEP 修正

用 PE 编辑工具 LordPE 打开 Dump.dll 程序。将 1240h 填进 EntryPoint 域中，单击“Save”按钮保存。

（2）输入表修正

用 LordPE 打开 Dump.dll，单击“Directories”按钮打开目录表，在 ImportTable 的 RVA 域中填入 7694h，大小为 3Ch（此值无关紧要，可填个比 0 大的数字），单击“Save”按钮保存修改，如图 13.75 所示。

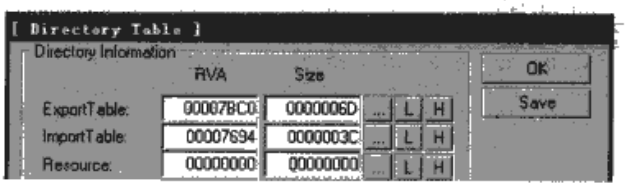


图 13.75　修正输入表的地址

（3）基址重定位表修正

用 LordPE 打开 Dump.dll，单击“Directories”按钮打开目录表，在 Relocation 的 RVA 域中填写 C000h，

Size 域中填写 5C0h。

（4）删除无用区块

程序中还有两个外壳使用的块已不需要，可以删除它们。用 LordPE 打开 dumped.dll，单击 Sections 打开区块表窗口，用鼠标右键菜单中的“Wipe section header”命令删除.aspack（ROffset=D000h）和.adata（ROffset=F000h）块。再用十六进制工具打开 dumped.dll，删除 D000h 后的数据。

13.9 加密壳

此类壳以加密保护为重点，用尽了各种反跟踪技术，保护重点是在 OEP 隐藏和 IAT 加密上，甚至是虚拟机加密技术。由于看雪论坛精华集里对加密壳进行分析的文章比较丰富，故本书不再重复了。

13.9.1 ASProtect

ASProtect 是一款经典的加密壳，它曾一度代表了加密壳的发展方向，其特长在于加密算法的运用，在保证强度的前提下，有着极好的兼容性和稳定性。

由于 ASProtect 在软件加密领域名气太大，用其保护的软件也多，从而导致许多爱好者研究其保护机制，并找出拆解的方案。因此，ASProtect 脱壳资料已很多，具体请参考看雪论坛中相关帖子。VoIX 的 Aspr2.XX_unpacker 脚本能支持目前 ASProtect 所有壳的版本，相当于一款脱壳机了。本节不是讲解如何去脱 ASProtect 壳，而是与大家讨论一下 ASProtect 的一些保护技术点。

1. Emulate standard system functions

ASProtect 可以将 API 入口一段代码抽到外壳里，执行完毕后，从中间调用系统 API，这样对 API 入口地址设断的传统方法将失效。来看一个样例，加壳前的 API 调用语句：

```
00401015  call    dword ptr [<&USER32.DialogBoxParamA>]
```

用 ASProtect 对目标程序加壳，勾选上“Emulate standard system functions”。OllyDbg 加载目标文件，来到同样的 API 调用处，其已将 DialogBoxParamA 函数的开头部分代码抽到外壳空间，执行完毕后，再跳回 DialogBoxParamA 继续执行。

```
00401015    call    00D10004
{
    00D10004    inc     dword ptr [esp]
    00D10007    jmp     00D00000

    00D00000    mov     edi, edi            ;DialogBoxParamA 函数入口代码搬到此处
    00D00002    push    ebp
    00D00003    mov     ebp, esp
    00D00005    push    ebx
    00D00006    push    esi
    00D00007    mov     esi, [ebp+8]
    00D0000A    push    0
    00D0000C    push    [ebp+C]
    00D0000F    or      ebx, FFFFFFFF
    00D00012    push    5
    00D00014    push    esi
    00D00015    call    [77D714C4]          ; kernel32.FindResourceExA
    00D0001B    test    eax, eax
    00D0001D    push    77D3B149            ;跳回 DialogBoxParamA 这个 API 函数里
    00D00022    retn
```

```
}
```

调试这类保护的程序，可以将断点设到 API 函数结尾返回处。

脱壳时，必须让外壳将正确的 API 调用写回去。先来查看一下目标文件的区块信息，如图 13.76 所示。

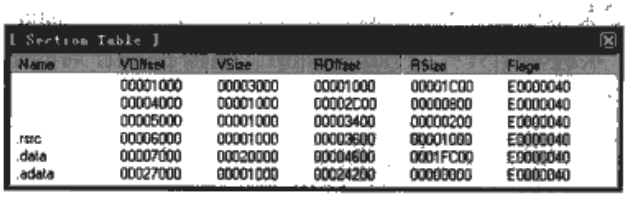


图 13.76　查看区块信息

外壳的入口代码在第一个区块上，外壳初始化完毕，必定会将解压出来的原程序数据填回第一个区块。因此，用 OllyDbg 重新加载目标程序，在数据窗口中，对第一个区块设内存写断点，本例是 401000h。

设断后，按 F9 键运行程序，第一次中断忽略，第二次中断如下：

```
00A6266B  rep     movs dword ptr es:[edi], dword ptr [esi]
00A6266D  mov     ecx, eax
00A6266F  and     ecx, 3
00A62672  rep     movs byte ptr es:[edi], byte ptr [esi]
00A62674  pop     edi
00A62675  pop     esi
00A62676  retn                          ;来到这后，按“Alt+M”键对第一区块下内存访问断点
```

取消内存断点，来到 A62676h 这行时，按“Alt+M”键打开内存窗口，对第一区块下内存访问断点。中断后：

```
00A6EE64  add     edi, dword ptr [edx]
00A6EE66  add     ecx, edi
00A6EE68  mov     edi, ecx
00A6EE6A  shl     edi, 3
```

这是一段对外壳进行校验的 Hash 函数，往下翻屏，一直来到函数的结尾处：

```
00A6F4BA  add     dword ptr [edx+54], eax
00A6F4BD  pop     edx
00A6F4BE  pop     ebp
00A6F4BF  pop     edi
00A6F4C0  pop     esi
00A6F4C1  pop     ebx
00A6F4C2  retn                          ;此行按 F4 键
```

走出这段校验函数，来到：

```
00A6F626  call    dword ptr [ecx+14]    ;会从这出来
00A6F629  add     edi, 40
00A6F62C  sub     ebx, 40
00A6F62F  cmp     ebx, 40
00A6F632  jge     short 00A6F620
00A6F634  mov     eax, dword ptr [esp]      ;在这按 F4 键
```

外壳校验完毕，将开始处理“Emulate standard system functions”功能，因此，按“Alt+M”键打开内存窗口，对代码段再次下内存写断点，当对代码段改写时，OllyDbg 就会再次中断。

```
00A8BA97  push    edx                   ;EAX 是 API 函数的地址，EBP 是马上要填写的地址
00A8BA98  push    0
00A8BA9A  lea     ecx, dword ptr [esp+18]
```

```
00A8BA9E  mov     edx, eax
00A8BAA0  mov     eax, dword ptr [ebx+3C]
00A8BAA3  call    00A87174              ;外壳抽API代码，感兴趣的可以跟进去研究一下
00A8BAA8  mov     ecx, eax
00A8BAAA  mov     dl, byte ptr [esp+18]
00A8BAAE  mov     eax, ebx
00A8BAB0  call    00A8BCA4
00A8BAB5  sub     eax, ebp
00A8BAB7  sub     eax, 5
00A8BABA  inc     ebp
00A8BABB  mov     dword ptr [ebp], eax;将值填到代码里
00A8BABE  mov     eax, dword ptr [esp+10]
00A8BAC2  mov     eax, dword ptr [eax]
00A8BAC4  mov     dword ptr [esp+14], eax
00A8BAC8  jmp     short 00A8BAD6
```

这段外壳代码，是将需要模拟的API函数头部提取出来，并将程序调用API的指令改为调用外壳指令。

修复的思路是，对这段外壳代码进行补丁，改写其原来功能。首先是在IAT里搜索获得的API地址，得到函数在IAT中的调用地址，然后将这个调用地址写回原程序里。

先在A8BA97h设个硬件断点，重新加载目标程序，会中断此处。用插件HideOD临时分配一些空间写补丁代码，在临时空间键入如下补丁代码：

```
00D00000  pushad
00D00001  mov     esi, 404000           ;404000是IAT起始地址
00D00006  cmp     dword ptr [esi], eax  ;eax中是API函数的地址
00D00008  je      short 00D00017        ;如果在IAT找到了匹配的API就跳
00D0000A  add     esi, 4                ;指向下一个DWORD
00D0000D  cmp     esi, 4040D0           ;4040D0是IAT结束地址
00D00013  ja      short 00D0002B        ;如果在IAT里都没找到，结束搜索
00D00015  jmp     short 00D00006        ;循环，继续搜索
00D00017  mov     cx, 15FF
00D0001B  mov     word ptr [ebp], cx    ;将原语句改写成类似的：call [4040C8]
00D0001F  add     ebp, 2
00D00022  mov     dword ptr [ebp], esi
00D00025  popad
00D00026  jmp     00A8BAC8              ;跳回外壳代码
00D0002B  jmp     short 00D0002B
```

可以在A8BA97h这句，键入一个转移指令jmp D00000来执行补丁，但由于这样改变了外壳代码，外壳的校验会导致程序异常出错。解决这个问题可以用OllyScript脚本来改变程序流程，保留A8BA97h这处的硬件断点，执行如下的脚本。这样每当程序中断在A8BA97h这句时，脚本将把补丁的地址D00000h写进EIP，并继续运行。

```
LABEL:
 cmp eip,00A8BA97        //判断中断是不是来自00A8BA97这行
jne END
 mov eip,00D00000
 run
 jmp LABEL
END:
 pause
```

执行脚本前，在A8BB01处设一个断点，当脚本执行完毕后，就会来到这里。

```
00A8BAF7  cmp     dword ptr [esp], 0
```

```
00A8BAFB  ja      00A8B956                    ;循环，继续修复其他 API
00A8BB01  push    ebx                         ;修补完所有 API 的调用，来到此处
```

补丁修复结束后，按“Alt+M”键打开内存窗口，在代码段设断，可以直接来到 OEP。

2. stolen bytes

stolen bytes 是指外壳将程序部分代码变形，并搬到外壳段。ASProtect 不仅能将 OEP 代码搬到外壳里，还能将程序中的代码搬到外壳里（需要编程使用 SDK）。用 ASProtect 对目标程序加壳，勾选上“Protect Original EntryPoint”。OllyDbg 加载目标文件，用上一节的方法，可以来到此处。

```
00A8BABA  inc     ebp
00A8BABB  mov     dword ptr [ebp], eax        ;会中断在这
00A8BABE  mov     eax, dword ptr [esp+10]
……
00A8BAF1  add     esi, dword ptr [ebx+E4]
00A8BAF7  cmp     dword ptr [esp], 0
00A8BAFB  ja      00A8B956
00A8BB01  push    ebx                         ;这里按 F4 键，跳过这段代码
```

这段是 Advanced Import protection 保护，具体可参考看雪论坛精华集。在 A8BB01h 一行，按 F4 键跳过这段，按“Alt+M”键打开内存窗口，对代码段下内存访问断点。中断如下：

```
00A82C6F  mov     byte ptr [ebx], 0E9
00A82C72  lea     edx, dword ptr [ebx+1]
00A82C75  mov     dword ptr [edx], eax
00A82C77  mov     eax, dword ptr [ebp+8]
00A82C7A  mov     dword ptr [eax], edx
00A82C7C  mov     eax, 5
00A82C81  pop     ebx
00A82C82  pop     ebp
00A82C83  retn    4
```

走出这段代码，来到：

```
00A8BF2A  add     ebx, 8                      ;此处 d ebx
00A8BF2D  mov     eax, dword ptr [ebx]
00A8BF2F  test    eax, eax
00A8BF31  jnz     short 00A8BF12
00A8BF33  pop     ebx
00A8BF34  pop     ecx
00A8BF35  pop     ebp
00A8BF36  retn    0C
```

在 A8BF2Ah 一行，在数据窗口中查看 EBX 指向的数据，如图 13.77 所示。

图 13.77　查看指向 stolen bytes 的数据

这些数据就是外壳指向 stolen bytes 的数据，其值为 RVA，例如其中的一个数据 13A0h，加上基址，其地址为 4013A0h。当外壳执行到 A8BF33h 这句时，查看 4013A0h 的代码。

```
004013A0    E9 CFEE8900    jmp     00CA0274
```

ASProtect 外壳为了兼容性，在原来抽掉的代码处，保留了一个转移指令，其指向 stolen bytes 代码。stolen bytes 代码是由一些变形代码和花指令组成的，例如一个典型变形语句：

```
add     edi, 4
```

变形后：

```
lea     edi, [edi+ecx+4]          //edi=edi+ecx+4
sub     edi, ecx                  //edi=edi-ecx=edi+ecx+4-ecx=edi+4
```

重建 stolen bytes 是很困难的，如果是 OEP 处的代码，可以根据编译语言的特征进行恢复，或采取补区段的方法恢复。有关 stolen bytes 修复，读者可以参考看雪论坛精华集的相关资料。

13.9.2　Themidia 的 SDK 分析[①]

Themida 是一款优秀的保护壳，在放弃使用驱动反调试后，强度主要靠 SDK 的虚拟机技术来保证。SDK 之外的脱壳过程，和其他的壳没有太大区别。下面主要讨论 SDK 保护代码的修复。

光盘映像文件中提供的演示程序 VMTest，使用了 SDK 的 Encode、Clear、CodeReplace 和 VM 宏，用 Themida1910 加壳，关闭 Protection Options 中的全部选项，虚拟机保护设置为最低，处理器类型选择 CISC-2，如图 13.78 所示。

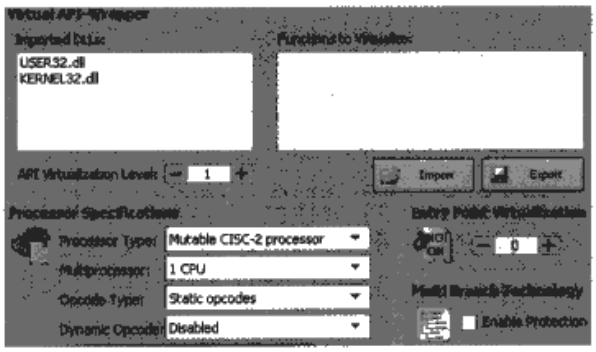


图 13.78　范例的 VM 加壳设置

1. Encode 与 Clear 宏保护代码的修复

下面是使用 ENCODE_START/END 宏保护的原始代码。

```
.text:00401001            jmp     short loc_401013
.text:00401003            dd 20204C57h, 4, 0, 20204C57h
.text:00401013 loc_401013:
.text:00401013            mov     esi, ds:MessageBoxA
.text:00401019            push    0                   ;uType
.text:0040101B            push    offset Caption      ;"Debug1"
.text:00401020            push    offset Text         ;"这段代码用 ENCODE 宏保护!"
.text:00401025            push    0                   ;hWnd
.text:00401027            call    esi                 ;MessageBoxA
.text:00401029            jmp     short loc_40103B
.text:0040102B            dd 20204C57h, 5, 0, 20204C57h
```

加壳后的代码为：

```
00401001    E8 61970B00     call    004BA767
00401006    0000            add     byte ptr [eax], al
```

[①] 本节由 softworm 编写

```
00401008   0000          add    byte ptr [eax], al
0040100A   0000          add    byte ptr [eax], al
0040100C   0000          add    byte ptr [eax], al
0040100E   1E            push   ds
0040100F   0000          add    byte ptr [eax], al
00401011   0020          add    byte ptr [eax], ah
00401013   71 55         jno    short 0040106A        ;这里下断
```

401001 处变成了 call 指令。Encode 宏保护实际是 SMC，即执行时会解出明文代码。ENCODE_START 占 18 字节，在第 19 字节即原始代码处（401013）下硬件执行断点。

```
00401001   E8 61970B00   call   004BA767
00401006   0000          add    byte ptr [eax], al
00401008   0000          add    byte ptr [eax], al
0040100A   0000          add    byte ptr [eax], al
0040100C   0000          add    byte ptr [eax], al
0040100E   1E            push   ds
0040100F   0000          add    byte ptr [eax], al
00401011   0020          add    byte ptr [eax], ah
00401013   8B35 9C504000 mov    esi, dword ptr [40509C] ; USER32.MessageBoxA
00401019   6A 00         push   0
0040101B   68 AC604000   push   004060AC         ;ASCII "Debug1"
00401020   68 94604000   push   00406094
00401025   6A 00         push   0
00401027   FFD6          call   esi
00401029   E8 62CF0A00   call   004ADF90          ;重新加密代码
0040102E   0000          add    byte ptr [eax], al
```

不仅解出了明文代码，在原 ENCODE_END 的位置（401029）还有个 call，执行后会将解出的代码重新加密。只要保存解码结果，NOP 掉 ENCODE_START/END 宏对应的 36 字节即可。

Clear 宏与 Encode 相似，唯一的区别是 CLEAR_END 为 23 字节，作用为擦除解出的代码，而不是重新加密。

```
0040105D   60            pushad
0040105E   E8 00000000   call   00401063
00401063   5F            pop    edi
00401064   81EF 23000000 sub    edi, 23
0040106A   B9 23000000   mov    ecx, 23
0040106F   33C0          xor    eax, eax
00401071   F3:AA         rep    stos byte ptr es:[edi]
00401073   61            popad
```

可以在 Dump 之前编写 OllyDbg 脚本来修复这两种宏保护的代码。

2. CodeReplace 与 VM 宏保护代码的修复

对于 Themida 当前版本，这两个宏是相同的，都使用虚拟机技术。加壳时将原始机器码反汇编转换为伪码，执行时由虚拟机引擎解释执行。如果想还原代码，需要分析 VM 解释引擎的工作方式，编写 pcode 解码器。Themida 目前支持 4 种 VM 处理器类型，分为 CISC 和 RISC 两类，后者提供的保护强度更高，也更复杂。

出于自我保护的目的，VM 解释引擎是被混淆过的变形代码。原始代码按预先定义的模式膨胀，生成的结果被划分为若干代码块，随机置换各代码块的物理位置，再用 JMP 指令链接起来，如下面的例子。

```
01596EEF   push   edx
01596EF0   add    ebx, ecx
01596EF2   mov    edx, 0
```

```
01596EF7    mov     bh, 35h
01596EF9    add     edx, edi
01596EFB    shr     al, 3
01596EFE    push    ecx
01596EFF    sub     eax, ebx
01596F01    jmp     loc_14F562A
014F562A    mov     ecx, 7445h
014F562F    neg     ecx
014F5631    not     ecx
014F5633    not     ecx
014F5635    or      ecx, 629Ah
014F563B    jmp     loc_14FA94D
014FA94D    sub     al, dl
014FA94F    or      ecx, 205Ch
014FA955    xor     ecx, 0FFFFDBFFh
014FA95B    mov     bl, 8Ah
014FA95D    add     ecx, edx
014FA95F    add     ah, dh
014FA961    jmp     loc_1594A3C
01594A3C    mov     esi, [ecx]
01594A3E    mov     bl, ch
01594A40    pop     ecx
01594A41    mov     ebx, 7570h
01594A46    mov     edx, [esp]
01594A49    add     esp, 4
```

这段代码逻辑上是连续的，物理上被分成了4块，用3个JMP链接起来。eax和ebx为空闲寄存器，用来生成干扰指令。以粗体显示的是有用的代码。从014F562A开始对ecx的连续变换结果为0。去掉JMP和干扰代码后为：

```
01596EEF    push    edx
01596EF2    mov     edx, 0
01596EF9    add     edx, edi
01596EFE    push    ecx
014F562A    mov     ecx, 0
014FA95D    add     ecx, edx
01594A3C    mov     esi, [ecx]
01594A40    pop     ecx
01594A46    mov     edx, [esp]
01594A49    add     esp, 4
```

这实际上只等于1条指令：

```
01596EEF    mov   esi, [edi]
```

如果不对变形代码进行清理，很难理解handler的真正目的。遗憾的是，笔者没有完美可靠的办法，目前的做法是根据具体的代码变形模式进行匹配压缩的，得到的结果容易出错，只能用来辅助分析。希望有人能开发出更好的方法。

下面分析范例中被CodeReplace宏保护代码的执行过程。VM宏与此类似，留给读者练习。光盘映像文件中有个简单的解码程序。被保护的原始代码为：

```
.text:00401086    push    0
.text:00401088    push    offset aDebug3  ; "Debug3"
.text:0040108D    push    offset aTICodereplaceG ; "这段代码用 CODEREPLACE 宏保护!"
.text:00401092    push    0
.text:00401094    call    esi
```

加壳后的代码，以 JMP 开始执行 VM 保护代码。

```
00401074    E9 F7FE0B00   jmp     004C0F70      ; CODEREPLACE_START 处为 JMP
004C0F70    68 FC5C2607   push    7265CFC
004C0F75    E9 701FF5FF   jmp     00412EEA
```

压栈的 imm32 代表了 pcode 数据的地址，同时被用做 pcode 数据解码 key。下面是 VM 入口代码。

```
00412EEA    pushad                          ;保存进入 VM 时的执行环境
00412EEB    pushfd
00412EEC    cld
00412EED    call    00412EF2
00412EF2    pop     edi
00412EF3    sub     edi, 71B7EDE            ;delta=F925B014
00412EF9    mov     eax, edi
00412EFB    add     edi, 71B7BF6            ;edi=412C0A,指向 ctx(上下文结构)
00412F01    cmp     eax, dword ptr [edi+2C];delta 与 ctx 内的值(初值为 0)
00412F04    jnz     short 00412F08          ;是否相等
00412F06    jmp     short 00412F1B
00412F08    mov     dword ptr [edi+2C], eax
00412F0B    mov     ecx, 0A7                ;167 个 handler
00412F10    jmp     short 00412F17
00412F12    add     [edi+ecx*4+40], eax     ;用 delta 计算 handler
00412F16    dec     ecx                     ;的实际地址填到地址表，这是第 1 次
00412F17    or      ecx, ecx                ;执行 VM 保护代码时完成的
00412F19    jnz     short 00412F12
00412F1B    mov     esi, dword ptr [esp+24];进入 VM 前 push 的 imm32
00412F1F    mov     ebx, esi                ;pcode 解码 key
00412F21    add     esi, eax                ;pcode 地址
00412F23    mov     ecx, 1
00412F28    xor     eax, eax
00412F2A    lock    cmpxchg [edi+30], ecx   ;检测设置 busy 标记
00412F2F    jnz     short 00412F28          ;等待 VM 空闲
00412F31    lods    byte ptr [esi]          ;取指令
00412F32    sub     al, 67
00412F34    jmp     0041B147
```

执行初始化代码后，检测 VM 是否忙，忙则等待，VM 不支持多线程访问。如果 VM 空闲，开始取 pcode 解释执行。清理后的取指令代码为：

```
l_FetchOpcode:
  lodsb                                     ;取 opcode
  add       al, bl
  sub       al, 7
  xor       al, 82h                         ;解码 opcode
  add       bl, al                          ;用解码结果变换 key
  movzx     eax, al
  jmp       dword ptr [edi+eax*4]           ;跳到对应的 handler
```

handler 的最后 1 条指令为 JMP，跳到 l_FetchOpcode 继续取指令/执行指令循环，直到遇到特定的 opcode（如 ExitVm）。

在调试代码前，先了解一下 VM 的基本特征。最重要的是上下文结构 VMCtx，保存了原程序寄存器组及 VM 内部使用的变量。其成员的含义要根据 handler 如何使用来确定。演示程序的 VMCtx 结构在 412C0A 处。

```
00000000 VMCtx          struc     ;(sizeof=0x44)
00000000 edx            dd ?      ;原程序的寄存器组
00000004 ecx            dd ?
```

```
00000008 ebp            dd ?
0000000C eax            dd ?
00000010 esi            dd ?
00000014 edi            dd ?
00000018 ebx            dd ?
0000001C eflag          dd ?
00000020 jxxFlag        dd ?    ;是否执行控制转移的标记
00000024 counter        dd ?    ;模仿控制转移的 handler 时用
00000028 indexOfEcx     dd ?    ;ecx 在 VMCtx 内的索引,模仿 jcxz,jecxz 时用
0000002C delta          dd ?    ;VM 加载地址
00000030 busy           dd ?    ;VM 忙标记
00000034 field_34       dd ?
00000038 field_38       dd ?
0000003C relocDelta     dd ?    ;重定位 delta,对 dll 加壳时使用
00000040 field_40       dd ?
00000044 VMCtx          ends
```

在 VMCtx 结构后面就是 handler 地址表，共 167 项。这里列出的数据已替换为清理变形代码后的地址，注释中是原始的 handler 地址。

```
Themida_:00412C4E Opcode11      dd offset loc_4CF000    ; 0041BE13
Themida_:00412C52 Opcode12      dd offset loc_4CF011    ; 00413AF4
Themida_:00412C56 Opcode13      dd offset loc_4CF021    ; 0041A819
Themida_:00412C5A Opcode14      dd offset loc_4CF03F    ; 0041D14E
```

Themida CISC-2 处理器的指令长度为 1 字节，每条指令可带有 0/1/2/4 字节的操作数。是否带有操作数，可以从相应的 handler 代码看出来，若有操作数，也需要解码。VMCtx 结构成员及 handler 地址表用 1 个字节作为索引即可寻址，opcode 编码直接从 0x11 开始。

进入 VM 后，有 3 个寄存器有特殊含义，在 handler 执行过程中保持不变：

```
ebx ->  解码 key
esi ->  指向 pcode 数据
edi ->  指向 VMCtx
```

handler 的实现使用了 3 个寄存器 eax/ecx/edx，未使用 ebp。可以认为这 3 个寄存器是 VM 内部的寄存器。为避免引起混淆，解码时将这 3 个寄存器另外命名。

```
eax ->  R0
ecx ->  R1
edx ->  R2
```

pcode 与被保护的原始机器码并非是一一对应关系，一条机器指令一般需要执行几条 pcode 才能模仿，如果再加上 pcode 变形，数量更多，性能损耗相当大。另外，有少量 opcode 由 VM 内部使用（如实现 pcode 变形），并不用于模仿原代码。

```
PUSH32      00000000
PUSH32      112ADEE4
POP32       R2
ADD32       R2,ctx.relocDelta ; 注意这里对重定位的处理
POP32       [R2]
```

这 5 行 pcode 代码模仿原来的一条指令“mov ds:[112ADEE4],0”。

Themida VM 提供的是“平面”式的保护，即被虚拟机保护的代码，如果其中有调用其他函数（或 API）的代码，被调用代码不会被纳入保护。当执行到 call 时会退出 VM，调用完成后重新进入 VM，这样原始代码对应的 pcode 数据被明显地分为几段。另一种情况是被保护代码中含有 VM 不支持的指令（如浮点指令），也会到 VM 外来执行。

下面是范例中被 CodeReplace 保护代码的 pcode 数据，被一个 call 分为两部分。pcode 数据的地址为进入 VM 时 push 的 imm32 加 delta。

```
Themida_:004C0D06    push    7265E67h        ;第 2 个 push imm32
Themida_:004C0D0B    jmp     l_Vm_Entry      ;下面为 pcode 数据
Themida_:004C0D10    dd 35C218E2h, 0B5429460h, 36C313E0h, 0B9469261h, 38C519E4h
Themida_:004C0D10    dd 16E12A8Fh, 8F6035C2h, 15E5BA47h, 9B693ECBh, 9A05C04Dh
......
Themida_:004C0D10    dd 648970BAh, 3F644B93h, 14392074h, 0E90EF549h, 0C2E7CE1Ah
Themida_:004C0D10    dd 99BEA5F5h, 8F947BCDh
Themida_:004C0F70
Themida_:004C0F70 loc_4C0F70:                          ; CODE XREF: start-74•j
Themida_:004C0F70    push    7265CFCh        ;第 1 个 push imm32
Themida_:004C0F75    jmp     l_Vm_Entry
```

第 1 段 pcode 数据地址= 7265CFC + F925B014 = 004C0D10

第 2 段 pcode 数据地址= 7265E67 + F925B014 = 004C0E7B

在实际情况中可能并不容易迅速看出 pcode 数据究竟由几段组成，但在对 pcode 解码的过程中最终会显示出来。

限于篇幅，有关范例中被 CodeReplace 保护代码的 pcode 分析的结果，请参考光盘映像文件中提供的文档。

13.1 静态脱壳

脱壳机编写分成两类，一类是静态脱壳，另一类是动态脱壳。静态脱壳需要完全分析出壳的引导过程及解压算法，把要脱壳的程序作为数据文件输入，然后自己实现壳的数据解压过程，修正 PE 结构，完成脱壳。这样可以避免调试不慎程序跑飞，很适合病毒、木马的脱壳分析，因此一些杀毒软件会集成一些静态脱壳引擎。动态脱壳机可以用调试 API 或虚拟机技术来实现，其加载目标文件，可以控制外壳的运行，利用壳自身解密数据再修复 PE 结构，相对来说比较容易实现。

13.10.1 外壳 Loader 的分析

编写静态脱壳机需要弄清楚壳的 Loader 工作过程，本文以 ASPack 1.08.0 版本为例讲解静态脱壳机编写的基本过程。

1. 外壳第一部分

该外壳的 Loader 分两部分，第一部分以非压缩的方式存在，第二部分以压缩的方式存在。外壳执行时先执行第一部分，这部分将外壳的第二部分在内存中解压缩，并初始化一些数据。用 IDA 打开目标实例，来到外壳的入口处。分析如下：

```
.aspack:01025000    start proc near
.aspack:01025000    pusha
.aspack:01025001    call loc_1025647
{
  .aspack:01025647    mov ebp, [esp]          ;将 CALL 的返回地址 1025006h 放 ebp
  .aspack:0102564A    sub ebp, 44291Ah
  .aspack:01025650    retn
}
.aspack:01025006    jmp short loc_1025049
......
.aspack:01025049    mov ebx, 442914h        ;ebx=442914h
.aspack:0102504E    add ebx, ebp            ;ebx=1025000h
.aspack:01025050    sub ebx, [ebp+44293Dh]  ;减去入口点偏移，获得当前映像基址
```

这段代码功能就是取当前映像的基址。接着外壳调用 aPLib 解压引擎，将外壳的代码第二部分解压出来。相关代码如下：

```
.aspack:01025056    cmp [ebp+dwImageBase2], 0;是否多次进入（处理 DLL 文件）
.aspack:0102505D    mov [ebp+442DFFh], ebx
.aspack:01025063    jnz _End_Import
.aspack:01025069    lea eax, [ebp+szKERNEL32.dll]
.aspack:0102506F    push eax                    ;ASCII "kernel32.dll"
.aspack:01025070    call [ebp+4431BCh]          ;call  GetModuleHandleA
.aspack:01025076    mov [ebp+hModKernel32], eax
.aspack:0102507C    mov edi, eax
.aspack:0102507E    lea ebx, [ebp+4430BDh]
.aspack:01025084    push ebx                    ;ASCII "VirtualAlloc"
.aspack:01025085    push eax
.aspack:01025086    call dword ptr [ebp+fnGetProcAddress]
.aspack:0102508C    mov [ebp+fnVirtualAlloc], eax
.aspack:01025092    lea ebx, [ebp+4430CAh]
.aspack:01025098    push ebx                    ;ASCII "VirtualFree"
.aspack:01025099    push edi
.aspack:0102509A    call dword ptr [ebp+fnGetProcAddress]
.aspack:010250A0    mov [ebp+442949h], eax
.aspack:010250A6    mov eax, [ebp+dwImageBase]
.aspack:010250AC    mov [ebp+dwImageBase2], eax
;前面这些代码就是为了获得 VirtualAlloc 函数的地址
.aspack:010250B2    push 4
.aspack:010250B4    push 1000h
.aspack:010250B9    push 54Ah                   ;分配 0x54A 的内存空间
.aspack:010250BE    push 0
.aspack:010250C0    call[ebp+fnVirtualAlloc]    ;call VirtualAlloc
.aspack:010250C6    mov [ebp+WorkMem1], eax
.aspack:010250CC    lea ebx, [ebp+442A11h]
.aspack:010250D2    push eax                    ;*destination
.aspack:010250D3    push ebx                    ;*source
.aspack:010250D4    call aPLib_Decode           ;解压外壳的第二段
; size_t aP_depack( const void *source,void *destination );
.aspack:010250D9    mov ecx, eax                ;0000054A
.aspack:010250DB    lea edi, [ebp+442A11h]      ;EDI=010250FD
.aspack:010250E1    mov esi, [ebp+WorkMem1]
.aspack:010250E7    rep movsb                   ;将刚解压的数据，复制到 10250FD 处
```

这段代码中最关键的就是解压函数 10250D4 call aPLib_Decode 的分析，如果不能识别何种算法，需要将其算法逆向，或将解压函数的汇编代码直接提取出来，程序里内嵌汇编调用即可。本例 ASPack 是调用了 aPLib v0.22b 压缩引擎，因此写脱壳机的时候，可以寻找相应版本的 aPLib SDK 参考。

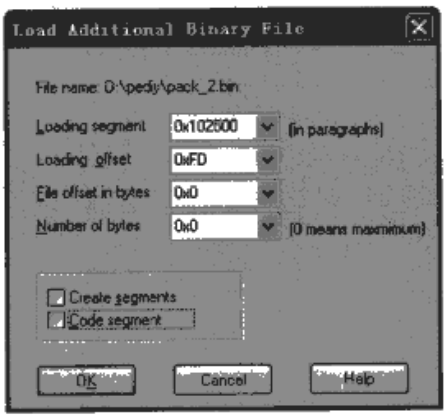


图 13.79　将二进制文件导入到 IDA 中

外壳将第二段代码解压后，覆盖到 10250FDh 地址处。由于外壳第二段是加密的，在 IDA 中查看是乱码，必须将其解压。可以用 IDC 脚本来实现，也可以用 OllyDbg 将解密后的数据提取出来，导进 IDA。后者相对来说更容易操作些，用 OllyDbg 加载实例，停在 10250E7h 这行，将 ESI 指向的 54Ah 大小的数据提取出来另存为 pack_2.bin。执行 IDA 的菜单“File/Load file/Additional binary file”，打开 pack_2.bin 文件，如图 13.79 所示。在“Loading segment”中填上 102500h，“Loading offset”中填上 0xFD，执行后，pack_2.bin 数据就会更新到 IDA 中的 10250FDh 地址处。

2. 外壳第二部分

第二部分开始的代码是外壳的真正部分，主要功能是还原原始程序，并初始化 IAT，如果需要重定位，则重定位，完成后就跳到真正的程序处执行。这部分比较重要，必须分析出外壳的一些重要数据结构，如区块的数据、输入表、入口点等信息。

```
.aspack:01025105    dd 0
.aspack:01025109    dd 1000000h
.aspack:0102510D    dd 0
.aspack:01025111    dd 0
.aspack:01025115 _RvaReloc dd 0
.aspack:01025119 _RvaImport dd 12B80h
.aspack:0102511D _RvaEntrypoint dd 12475h
.aspack:01025121    dd 0
.aspack:01025125    dd 0
.aspack:01025129    dd 1000h                    ; .text 的 RVA
.aspack:0102512D    dd 12800h                   ; .text 的 VSize
.aspack:01025131    dd 14000h                   ; .data 的 RVA
.aspack:01025135    dd 0A00h                    ; .data 的 VSize
.aspack:01025139    db 1D0h dup(0)
```

上面是外壳的一些重要数据结构，这些数据在内存中的偏移是固定的，如入口点 Entrypoint 的偏移为 11Dh。脱壳机工作时，需要从这些偏移中获取相关的数据及结构。

外壳的第二部分代码如下：

```
.aspack:01025309    mov ebx, [ebp+442A21h]    ;即: mov ebx,[102510D]
.aspack:0102530F    or  ebx, ebx
.aspack:01025311    jz  short _Begin_DoSection
.aspack:01025313    mov eax, [ebx]
.aspack:01025315    xchg eax, [ebp+442A25h]
.aspack:0102531B    mov [ebx], eax
.aspack:0102531D _Begin_DoSection:
.aspack:0102531D    lea esi, [ebp+442A3Dh]    ;取区块信息
.aspack:01025323    cmp dword ptr [esi], 0
.aspack:01025326    jz  _End_Decode_Section
.aspack:0102532C    lea esi, [ebp+442A3Dh]
.aspack:01025332 _Loop_Decode_Section:
.aspack:01025332    mov eax, [esi+4]
.aspack:01025335    push 4
.aspack:01025337    push 1000h
.aspack:0102533C    push eax
.aspack:0102533D    push 0
.aspack:0102533F    call [ebp+fnVirtualAlloc];call kernel32.VirtualAlloc
.aspack:01025345    mov [ebp+442941h], eax
```

```
.aspack:0102534B    push esi
.aspack:0102534C    mov ebx, [esi]
.aspack:0102534E    add ebx, [ebp+4430A8h]
.aspack:01025354    push eax                    ;pDst
.aspack:01025355    push ebx                    ;pSrc
.aspack:01025356    call aPLib_Decode           ;解压区块数据
; size_t aP_depack(const void *source,void *destination);
.aspack:0102535B    cmp byte ptr [ebp+44293Ch], 0
.aspack:01025362    jnz short _CopyMem
{
   ......                                       ;如果是第一次需要修复 E8E9
}
.aspack:010253B0 _CopyMem:
{
   ......                                       ;将解压后的区块数据复制到指定地址处
}
.aspack:010253E4    cmp dword ptr [esi], 0
.aspack:010253E7    jnz _Loop_Decode_Section
.aspack:010253ED    mov ebx, [ebp+442A21h]
.aspack:010253F3    or  ebx, ebx
.aspack:010253F5    jz  short _End_Decode_Section
.aspack:010253F7    mov eax, [ebx]
.aspack:010253F9    xchg eax, [ebp+442A25h]
.aspack:010253FF _End_Decode_Section:
```

这段代码是将各区块的数据解压，并复制到内存映像指定的位置上。其中一段代码是修复代码段 E8E9 的功能，需要注意一下。相关汇编代码如下：

```
.aspack:0102538C _Loop_Fix_E8E0:
.aspack:0102538C    or  ecx, ecx
.aspack:0102538E    jz  short loc_10253AC
.aspack:01025390    js  short loc_10253AC
.aspack:01025392    lodsb
.aspack:01025393    cmp al, 0E8h
.aspack:01025395    jz  short loc_102539F
.aspack:01025397    cmp al, 0E9h
.aspack:01025399    jz  short loc_102539F
.aspack:0102539B    inc ebx
.aspack:0102539C    dec ecx
.aspack:0102539D    jmp short _Loop_Fix_E8E0
```

外壳为了提高压缩率，将 CALL、JMP 指令修正了一下。下面两句指令是压缩前的指令，都是 CALL 到同一地址，但其机器码不同。代码如下：

```
0100832A  E8 5FF9FFFF   call    01007C8E
0100898C  E8 FDF2FFFF   call    01007C8E
```

现在外壳改为用相对基址的偏移来改写指令，100832Ah 的 RVA 为 832Ah，相对基址 1000h 的偏移为 732Ah，1007C8Eh+732Ah=100EFB8h，因此改写的指令如下：

```
0100832A E8 896C0000    call    0100EFB8
0100898C E8 896C0000    call    0100F61A
```

改写后，CALL 调用改成相对于 Base 的偏移，而不是当前指令的偏移，这样提高了机器码的重复率，

压缩率也就提高了。

外壳运行时，按这个相反过程将原指令恢复。用高级语言来描述就是：

```
//如果是第一次解压区块数据，则是代码段，需要修复 E8E9
if (isfirst)
{
    isfirst = false;
    BYTE *p = (BYTE *)Image + pSectionInfo->Rva;
    int nSize = dwRealSize - 6;
    DWORD off = 0;
    while (nSize > 0)
    {
        if ((*p == 0xE8)||(*p == 0xE9))
        {
            *(DWORD *)(p + 1) -= off;
            off += 4;
            p += 4;
            nSize -= 4;
        };
        off++;
        p++;
        nSize--;
    }
};
```

原始数据解压恢复后，如果需要重定位，外壳就会对代码进行重定位。由于 ASPack 没破坏原始重定位数据，因此只需要得到重定位表的地址就可修复了。

```
.aspack:01025405    mov eax, [ebp+dwOriginalImageBase]
.aspack:0102540B    sub edx, eax                    ;判断需不需要重定位
.aspack:0102540D    jz  short __End_Reloc
.aspack:0102540F    mov eax, edx
.aspack:01025411    shr eax, 10h
.aspack:01025414    xor ebx, ebx
.aspack:01025416    mov esi, [ebp+442A29h]          ;取重定位表的 RVA  1025115
```

从上面的代码可知，重定位表的 RVA 外壳保存在偏移 115h 处。

原始数据解压恢复后，外壳就会模拟 Windows 系统加载器，填充 IAT，由于 ASPack 没破坏原始输入表的结构，因此此时只需要得到输入表的地址就可修复了。

```
.aspack:01025488    mov esi, [ebp+442A2Dh]          ;1025119，输入输入表的 RVA
.aspack:0102548E    mov edx, [ebp+dwImageBase2]     ;这段代码是模拟加载器，填充 IAT
.aspack:01025494    add esi, edx
.aspack:01025496    mov eax, [esi+0Ch]
.aspack:01025499    test eax, eax
.aspack:0102549B    jz  _End_Import
```

从上面的代码可知，输入表的 RVA 外壳保存在偏移 119h 处。

现在已经分析完了 Loader。用伪代码描述一下它的核心流程：

```
//DoSection
While (pSectionInfo->Rva)
{
```

```
    //
};
//DoReloc
If (ImageBase != dwOriginalImageBase)
{
    //
};
//DoImport
While (pImportInfo->Rva)
{
    //
}
//DoEntrypoint;
```

13.10.2　编写静态脱壳器

通过分析，已经知道壳的加载过程，现在来分析如何取得解压缩时的重要数据。写静态脱壳的关键就是找到正确的数据，把相关信息还原或者补回去。其中必需的步骤如下：

① 数据位置的确定；

② 算法的还原；

③ 输入表、重定位表以及资源的修复。

要想正确脱壳，首先得对壳进行识别，对根本不认识的版本进行脱壳的结果是不可预知的。因此，首先来检查程序的入口，判断是否为目标。

```
DWORD dwEntrypoint = ph->OptionalHeader.AddressOfEntryPoint;
BYTE *pEntrypoint = (BYTE *)RvaToPointer(pMap, dwEntrypoint);

//根据入口指令判断是否是该壳
if ((*pEntrypoint != 0x60)
    ||(*((DWORD*)(pEntrypoint + 0x49)) != 0x442914BB)//mov ebx, 442914
    ||(*((DWORD*)(pEntrypoint + 0xBA)) != 0x54A) //push 54Ah
    ||(*((DWORD*)(pEntrypoint + 0x29)) != dwEntrypoint)
)
{
    halt3("not aspack 1.08.04", -11);
};
```

然后，将解压函数 call aPLib_Decode 中的代码逆向或直接将汇编代码提取出来。具体代码详见源码中的 aPLibDePack()函数。

接着调用 aPLibDePack()将各区块的数据解压恢复出来，代码如下：

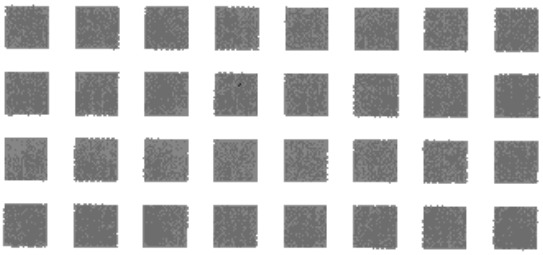
```
stSectionInfo *pSectionInfo = (stSectionInfo *)(pLoaderCore + offSectionInfo);
bool isfirst = true;
while (pSectionInfo->Rva != 0)
{
    //直接解压到 Image
    DWORD dwRealSize = aPLibDePack(RvaToPointer(pMap, pSectionInfo->Rva), \
                       Image + pSectionInfo->Rva);

    //如果是第一次需要修复 E8E9
    if (isfirst)
```

```
        {
          ......
        };
        pSectionInfo++;
    };
```

由于壳没有破坏输入表、重定位表，可以直接将这些数据写回 PE 头，否则必须重建输入表和重定位表。

外壳压缩时，通常壳会把 MAINICON 和 VERSION 等资源提取出来放在壳外面（为了能够正常看到程序图标和版本信息）。如果需要将外壳的代码区块删除，则脱壳时必须将这些资源还原到.rsrc 资源区块里或重建资源。这部分代码本节没有提供，读者参考相关资料自己加上。

最后，修正入口点，优化区块信息，将修正后的文件保存到文件，完成脱壳。



第 7 篇　保护篇

市场上虽有大量现成的保护方案可选用，如基于软件的加密壳保护和基于硬件的加密锁保护，但这些优秀的保护方案由于太流行，造成大家对其研究的透彻，反而容易被破解。所以，有必要自己实现相关的保护方法。

第14章 软件保护技术

市场上虽有大量现成的保护方案可选用，如基于软件的加密壳保护和基于硬件的加密锁保护，但这些优秀的保护方案由于太流行，造成大家对其研究的透彻，反而容易被破解。所以，有必要自己实现相关的保护方法。

14.1 防范算法求逆

设计一套合理的注册算法，应该有必要的抗分析手段，可以有效限制解密者分析注册算法，从而阻止注册机或者破解补丁的出现。

14.1.1 基本概念

软件保护的目的是向合法用户提供完整的功能，所以软件保护必然要包括验证用户合法性的环节，而这一环节通常采用注册码验证的方式实现。

（1）用户向软件作者提交用户码 U，申请注册。

（2）软件作者计算出注册码 $R=f(U)$，返回给合法用户。

（3）用户在软件注册界面输入 U 和 R。

（4）软件验证 $F(U,R)$的值是否合法来判定用户的合法性。

其中一些常用术语说明如下：

- 用户码 U：用于区别用户身份。
- 注册码 R：用于验证用户身份。
- 注册机：把 $R=f(U)$ 中的小 f 称为注册机，掌握了注册机就有能力针对任何用户码计算出相应的注册码。
- 验证函数：把 $F(U,R)$ 中的大 F 称为验证函数，软件使用验证函数验证注册码的合法性，即，当且仅当 $R=f(U)$ 成立时，$F(U,R)$ 取合法值。
- 算法求逆：把解密者通过验证函数 F 推导注册机 f 的过程称为算法求逆，所以验证函数 F 的构造非常关键。

在软件注册保护的“初级阶段”，验证函数与注册机没有本质区别，即：$F(U,R)=f(U)-R$。这样做很危险，验证函数自身就包含注册机，解密者只需要跟踪软件的运行，直接将软件验证函数中计算 $f(U)$ 的汇编代码拷贝下来就可以当注册机用，甚至根本不需要了解 f 的算法。

改进的做法是先求出 f 的反函数 f^{-1}，使：$U=f^{-1}(R)$，然后令 $F(U,R)=f^{-1}(R)-U$。这样做安全了许多，

软件本身不包含注册机 f，解密者必须在充分了解 f^{-1} 算法过程的基础上才能分析推导出注册机 f。可能会有读者感到疑惑：假如解密者先指定 R，然后直接利用验证函数计算出 $U=f^{-1}(R)$，不就可以使用 U、R 来注册了吗，何必一定要推导 f 呢？在实际应用中，由于 U、R 通常以字符串的形式给出，而验证函数通常采用数值运算，所以一般会将 U、R 转换成数值形式 U'、R'，则注册机 f 及注册机的反函数 f^{-1} 实际上都是复合函数，验证函数由于只需要检验 U'、R' 的合法性，因而可以不完全等于 f^{-1}。

假设		$U'=f_1(U)$，$R'=f_2(U')$，$R=f_3(R')$
则	f	$R=f(U)=f_3(f_2(f_1(U)))$
	f^{-1}	$U=f^{-1}(R)=f_1^{-1}(f_2^{-1}(f_3^{-1}(R)))$
	F	$F(U,R)=f_2^{-1}(f_3^{-1}(R))-f_1(U)$

看起来解密者通过 $F(U,R)=f_2^{-1}(f_3^{-1}(R))-f_1(U)$ 推导 f^{-1} 比推导 f 要容易，关键在于 f^{-1} 通常是建立在 ASCII 编码表之上的一个变换，所以即使推导出 f^{-1} 也没有多大价值。

14.1.2 堡垒战术

事实上，在通信领域人们很早就开始了身份验证的研究，并发展出了散列加密和非对称加密等优秀的密码学算法，其中的 MD5 算法和 RSA 算法非常适合在软件注册算法中运用。

MD5 算法通常并不被直接用来对消息进行加密，因为不存在逆算法，所以加密之后无法解密，这样的加密是没有应用价值的，MD5 算法的用途在于数字签名。例如甲和乙进行通信，甲仅仅将明文 A 加密为 B 传送给乙是不够的，因为即使密文 B 的加密强度再高也只能防止在传输的过程中被解密而泄露内容，却不能防止破坏者直接篡改密文 B。乙收到 B 后解密得到 A，也无法确定 A 所包含的内容的真实性。所以甲在发送 B 的同时通常会在 A 之后署名，然后计算 C = MD5(A)，将 B、C 一起发送给乙。乙收到 B、C 后，首先解密 B 得到 A，然后同样计算 C = MD5(A)，若计算出的 C 与收到的 C 相同，则可确定 A 真实可靠。

同样 MD5 算法也不适合直接被用来做注册机，假如使用 $R=\text{MD5}(U)$ 做注册机，由于 MD5 不存在反函数，验证函数将不得不包含注册机。但是用以下办法使用 MD5 算法：

（1）设注册机 f 为：$R=f(U)$。

（2）设 $\text{MD5}(a)=b$。

（3）令验证函数 F 为：$F(U,R)=\text{MD5}\,[f^{-1}(R)-U+a\,]$。

显然 F 的合法值应该为 b，只要 U、R 满足 $R=f(U)$，F 一定等于 b。由于 MD5 不可逆，解密者无法通过 b 获知 $f^{-1}(R)-U+a$ 应该等于 a，也就无法获得 f^{-1} 的准确表达式，更无法获得注册机 f。至于 $f^{-1}(R)-U+a$ 表达示中虽然含有 a，但解密者无法判断 a 和 f^{-1} 的关系。例如：$U=f^{-1}(R)=R*5+19$，$a=7$，则 $F(U,R)=\text{MD5}(R*5+26-U)$，$a$ 和 f^{-1} 融为一体，叫解密者如何分得清？

实际上利用 MD5 算法还有很多方法可以构造 F，让解密者根本看不出 F 和 f、f^{-1} 的关系，就更谈不上求逆了。这里只是举了一个简单的例子而已，相信读者完全能够进行精彩的发挥。

当然，MD5 算法也有它的缺陷，正因为 MD5 完全不可逆，所以 a 必须为常数，一旦解密者获得了一对合法的 U、R，他们就可以跟踪到合法的 a、b 值，从而获得 f^{-1}，并进一步推导出注册机 f。当然，前提是他们必须先设法获得一对合法的 U、R。

RSA 算法很容易在软件保护中进行应用：

（1）软件作者使用：$R=U^d \bmod n$ 作为注册机。

（2）软件使用：$U=R^e \bmod n$ 作为验证函数。

解密者即使跟踪软件运行的全过程，也得不到 d，无法写出注册机。基本上进行 RSA 算法保护的软件可以说是建立了一座坚不可摧的堡垒，但是由于 RSA 算法众所周知，再加上软件作者通常会采用公共函数库来实现 RSA 算法，所以使用 RSA 算法也存在一些风险。

① RSA 算法本身虽然足够坚固，但使用者往往直接采用第三方公用代码。这些代码可能含有漏洞，

而世界上有大量的爱好者在研究这些代码的漏洞，一旦发现漏洞，软件的安全性就可能成为陪葬品。

② RSA 算法的使用者往往并不了解算法细节，可能因错误使用 RSA 而在不知不觉之中遭遇非常规手段的攻击。例如，在不同的软件作品中，使用不同的 e、d 但使用相同的 n 而遭到“公共模组攻击”等。

③ RSA 算法在通信领域由于密文的解密过程并不暴露给窃听者，所以其加解密过程虽然同为模幂运算，但实际实现过程往往并不一致，在解密端通常会采用“中国剩余定理”进行加速，而中国剩余定理中包含对原始数据 p、q 的引用。这一类函数库在软件保护中使用时要非常小心，一旦软件作者选择了错误的 RSA 库函数，在验证函数中使用了中国剩余定理，则会导致 RSA 防线如同虚设。

④ 某些函数库在生成随机素数时，采用“伪随机数产生器”，即在完全相同的初始条件下会产生完全相同的“随机数”序列。解密者如果得到该函数库，就可以根据 n 值推断 p、q 生成过程，从而攻破 RSA 防线。

⑤ RSA 算法中存在若干由某些特殊素数构造而成的“弱密钥”，某些函数库在生成随机素数时，没有淘汰这些特殊素数，导致 RSA 防线在数论高手面前虚弱无力。

14.1.3 游击战术

软件保护中的游击战术就是将验证函数 F 分解成多个互不相同的 F_i，然后将这些 F_i 尽可能地隐藏到程序里去。

通过任意一个 F_i 的验证都只是注册码合法的必要条件，而非充分条件，真正合法的注册码能够通过所有的 F_i 的验证。解密者找到 F_i 其中的任一个或任意几个，只要不能将所有的 F_i 一网打尽，他就无法一睹 F 的全貌，无法进行算法求逆。

当然，将 F 分解成一系列必要非充分的 F_i 需要较专业的数学知识，但可以使用分段函数来简单地实现这一目标。

（1）将 R 切分成多段 R_i。

（2）构造不同的 f 算法，使得：$R_i = f_i(U)$。

（3）令 $F_i = f_i^{-1}$。

这样做虽然有点麻烦，但绝对是值得的。例如可以让 F_1 使用 MD5 算法，F_2 使用 RSA 算法，F_3 使用自定义算法。在用户输入注册码后仅仅使用 F_1 进行验证，并将注册码以密文形式写入自定义格式的数据文件中，如果验证通过就显示注册成功的提示。另外两个验证函数藏起来，只有使用者执行特定的操作时才被调用，例如，在用户进行存档操作或使用某些高级功能的时候将注册码读出来再次验证。一旦任何一个验证函数发现注册码非法，就清除注册码并将软件恢复为未注册状态。

游击战第二个宗旨是虚虚实实。对于解密者来说，遇到游击战术会非常被动，除非他找到的验证函数已经能够将 U、R 形成一对一的对应关系，否则永远不能确定软件中是否还埋藏着其他的验证函数，而事实上软件作者根本没有必要让 U、R 形成一对一的对应关系，验证函数个数的不确定性的确很容易让试图制作注册机的解密者懊恼不已。

假如运用一点简单的线性代数的知识，可以将 R_i 的其中几个（注意只是其中几个，而不是全部）和 F_i 关联起来：

设：$R_a = 3U$，$R_b = 5U$，$R_c = 7U$，则：

$$F_a = 7R_a + 11R_b + 5R_c - 111U$$

$$F_b = 11R_a + 7R_b + 3R_c - 89U$$

$$F_c = 5R_a + 3R_b + 11R_c - 107U$$

这样解密者找到 F_a、F_b、F_c 中的任意一个，甚至无法求出 R 哪怕是小小的一段。一个更好的主意是让参与到线性方程组中的 R_i 的个数稍稍大于使用线性方程的验证函数的个数，软件作者手里持有线性方程的某一组特定解作为注册机，而解密者则无法了解验证函数到底有几个。

如果将一对 U、R 作为纵横坐标，看做平面上的一点，将注册机 f 看做由合法 U、R 连成的一段平面曲线，还可以构造多个空间曲面方程作为验证函数 F，条件是 f 落在这些空间曲面之上，如果稍稍了解空间解析几何的知识，相信各位可以构造出无数个曲面方程作为验证函数，甚至还可以考虑使用参数方程，这样即使解密者获得了所有的 F_i，也要有精深的数学水平才能求出 f。

这里必须反复强调数学知识的重要性，不管是数论、代数、线性代数、几何、解析几何，还是微积分、概率论，都可以拿来作为软件保护的武器。

游击战第三个宗旨是战略转移。游击战术的致命弱点在于，每一个验证函数都必须访问注册码，而注册码的源头只有一个。解密者会跟踪程序从注册界面读入注册码的过程，并监控存放注册码的内存地址，一旦验证函数访问这一地址就会泄露行踪，这样注册码实际上成为了解密者寻找验证函数的一把钥匙，理论上解密者只要牢牢地抓着这把钥匙不放，就一定会找到所有的验证函数。应对的办法就是大规模的转移，软件必须不停地将注册码“搬家”，搬家的方法要多样化。

（1）内存拷贝，这种常规做法容易被解密者用内存监视断点识破。

（2）写入注册表或文件，然后在另一处代码中再读入到另一个内存地址，这种办法会被解密者的注册表、文件监视工具识破。

（3）一次将注册码拷贝到多个地址，让解密者无法确定哪一个地址是注册码的新家。

（4）在反复使用同一个函数搬家后，突然使用另一个前半部分代码相同而后半部分不同的函数进行搬家。

（5）将以上方法反复使用。

事实上，主动权永远掌握在程序员的手里，还可以用更多的方法来对付解密者。

14.2 抵御静态分析

静态分析是指从反汇编出来的程序清单上分析程序流程。反静态分析技术主要是从扰乱汇编代码可读性入手，如使用大量的花指令、将提示信息隐藏等。

14.2.1 花指令

在反汇编的过程中，存在着几个关键的问题，其中之一是数据与代码的区分问题。汇编指令长度、多种多样的间接跳转实现形式，反汇编算法必须对这些情况做出恰当的处理，保证反汇编结果的正确性。

目前主要的两类反汇编算法是线性扫描算法（Linear Sweep）和递归行进算法（Recursive traversal），它们都有着广泛的应用。常见的反汇编工具所用的反汇编算法见表 14-1。

表 14-1 常见的反汇编器实现的技术

工 具 名	反汇编算法
OllyDbg	Linear Sweep/Recursive traversal(按“Ctrl+A”键时)
SoftICE	Linear Sweep
WinDBG	Linear Sweep
W32Dasm	Linear Sweep
IDA Pro	Recursive traversal

线性扫描算法这种方法的技术含量并不高，反汇编器只是依次逐个地将整个模块中的每一条指令都反汇编成汇编指令。没有对所反汇编的内容进行任何判断，而是将遇到的机器码都作为代码来处理。因此无法正确地将代码和数据区分开，数据也将被作为代码来进行解码，从而导致反汇编出现错误。而这种错误将影响下一条指令正确识别，会使得整个反汇编都错误。

递归行进算法按照代码可能的执行顺序来反汇编程序，对每条可能的路径都进行扫描。当解码出分支指令后，反汇编器就将把这个地址记录下来，并分别反汇编各个分支中的指令。采用这种算法可以避免将代码中的数据作为指令来解码，比较灵活。

巧妙构造代码和数据，在指令流中插入很多“数据垃圾”，干扰反汇编软件的判断，从而使得它错误地确定指令的起始位置，这类代码数据称之为“花指令”。用花指令来进行静态加密是很有效的，这会使解密者无法一眼看到全部指令，杜绝了先把程序代码列出来再慢慢分析的做法。

不同的机器指令包含的字节数并不相同，有的是单字节指令，有的是多字节指令。对于多字节指令来说，反汇编软件需要确定指令的第一个字节的起始位置，也就是操作码的位置，这样才能正确地反汇编这条指令，否则它就可能反汇编成另外一条指令了。

以下是一段汇编源程序：

```
start_:
    xor  eax,eax
    test eax,eax
    jz   label1
    jnz  label1
    db   0E8h              ; 注意这里
label1:
    xor  eax,3
    add  eax,4
    xor  eax,5
    ret
```

对源程序进行编译，然后用 W32Dasm 进行反汇编，来看一下反汇编后的结果。

```
:00401000 33C0              xor  eax, eax
:00401002 85C0              test eax, eax
:00401004 7403              je   00401009
:00401006 7501              jne  00401009
:00401008 E883F00383        call 83440090
:0040100D C00483F0          rol byte ptr [ebx+4*eax], F0
:00401011 05C3000000        add eax, 000000C3
```

由于 Linear Sweep 式反汇编软件是逐行反汇编，代码中的垃圾数据 E8h 干扰了其工作，结果错误地确定了指令的起始位置，导致反汇编的一些跳转指令跳转的位置无效，就像这里 40100Bh 地址不再是一条指令的起始处，而是出现在指令的内部了。因此，如果反汇编一个程序时发现这样的特征，就可以断定该程序中使用了花指令。当然还有很多方法可使反汇编软件落入陷阱之中。

OllyDbg 打开文件用的是 Linear Sweep，分析代码功能（按“Ctrl+A”键）用的是 Recursive traversal 算法。用 OllyDbg 打开实例分析后，其生成的反汇编代码完全正确，那个 0E8h 字节并没有被反汇编，此类花指令不能迷惑 OllyDbg。如下所示：

```
00401000   33C0          xor    eax, eax
00401002   85C0          test   eax, eax
00401004   74 03         je     short 00401009
00401006   75 01         jnz    short 00401009
00401008   E8            db     E8
00401009   83F0 03       xor    eax, 3
0040100C   83C0 04       add    eax, 4
0040100F   83F0 05       xor    eax, 5
00401012   C3            retn
```

在 Recursive traversal 算法中，一个十分重要的假设是对于任一条控制转移指令，其后继即转移的目的

地址都能够确定。要迷惑这类反汇编器，只要让其难以确定跳转的目的地址即可。在这创建一个指向无效数据的跳转指令代码。代码如下：

```
start_:
    xor     eax,eax
    test    eax,eax
    jz      label1
    jnz     label0                  ;指向无效的跳转指令
label0:
        db  0E8h
label1:
    xor     eax,3
    add     eax,4
    xor     eax,5
    ret
end start_
```

用 OllyDbg 打开编译好了的实例，这次汇编代码识别出错了，其认为垃圾数据 0E8h 所在的地址 401008 是有效的，故将其当指令起始地址，结果导致后面指令识别出错。

```
00401000   33C0          xor     eax, eax
00401002   85C0          test    eax, eax
00401004   74 03         je      short 00401009
00401006   75 00         jnz     short 00401008
00401008   E8 83F00383   call    83440090
0040100D   C00483 F0     rol     byte ptr [ebx+eax*4], 0F0
00401011   05 C3000000   add     eax, 0C3
```

通过前面的介绍，知道由于“无用的字节”干扰了反汇编器对指令起始位置的判断，从而导致反汇编的错误结果。如果能让反汇编正确地识别指令的起始位置，也就达到了去除花指令的目的了。例如，可把那些无用的字节都替换成单字节指令，最常见的一种替换方法是把无用的字节替换成 NOP 指令，即十六进制数 90。

```
00401000   33C0          xor     eax, eax
00401002   85C0          test    eax, eax
00401004   74 03         je      short 00401009
00401006   75 00         jnz     short 00401008
00401008   90            nop                     ;0E8h 被替换成 090h
00401009   83F0 03       xor     eax, 3
0040100C   83C0 04       add     eax, 4
0040100F   83F0 05       xor     eax, 5
00401012   C3            retn
```

OllyDbg 的一个花指令去除插件，就是利用这个原理来去除花指令的，根据收集的花指令特征码，将垃圾数据替换为 NOP 指令，从而使得反汇编工具正常工作。

14.2.2　SMC 技术实现

编写 SMC 代码不是汇编语言的专利，但是使用汇编语言编写可以更充分并且方便地控制每一个细节，即便如此，通常调试一段 SMC 代码也比调试普通的程序要花费更多的时间和耐心。不过一旦掌握了这项技术，那么就可以设计出更加完善的加密方案。

在这先看一个实例：

```
;-=-=-=-解密数据-=-=-=-=-
push    DataLen
```

```
    push    offset EnData
    call    DecryptFunc

EnData  BYTE0ECh,01Bh,054h,076h,064h,064h,066h,074h,074h,001h
        BYTE04Ah,021h,06Dh,070h,077h,066h,021h,075h,069h,06Ah
        BYTE074h,021h,068h,062h,06Eh,066h,022h,001h,060h,0E9h
        BYTE001h,001h,001h,001h,05Eh,082h,0EEh,023h,011h,041h
        BYTE001h,06Bh,001h,08Eh,086h,003h,011h,041h,001h,051h
        BYTE08Eh,086h,00Bh,011h,041h,001h,051h,06Bh,001h,000h
        BYTE0D8h,08Eh,0BEh,001h,011h,041h,001h,0BAh,04Eh,001h
        BYTE001h,001h,034h,0C1h,0FDh,0F4h,0ABh

DataLen     EQU $ - offset EnData       ;$
lm_exit:    invoke  ExitProcess,0
```

或许一开始读者就注意到了 EnData 地址处的一段莫名奇妙的数据，它是干什么用的？稍作思考，不难想到那段数据是一段被加密了的代码。现在看一下 DecryptFunc 函数，就可以发现这个函数会对 EnData 地址处的 77 个字节进行异或，从而解密出加密的代码，异或完成之后，程序将从 DecryptFunc 函数返回，并且将执行已经解密后的 EnData 代码。这段代码的真实面目是：

```
CodeBegin:  jmp     loc_begin
szTitle     BYTE'Success',0
szMsg       BYTE'I love this game!',0
loc_begin:
    ;-=-=-=-从堆栈中得到MessageBoxA函数的地址-=-=-=-=-
    pop     edi
    ;-=-=-=-计算运行时的地址与编译地址之差-=-=-=-=-
    call    loc_next
loc_next:
    pop     ebp
    sub     ebp,offset loc_next
    ;-=-=-=-调用MessageBoxA函数显示信息-=-=-=-=-
    push    MB_OK
    lea     eax,[ebp + szTitle]
    push    eax
    lea     eax,[ebp + szMsg]
    push    eax
    push    NULL
    call    edi
    ;-=-=-=-把自身的代码清零-=-=-=-=-
    lea     edi,[ebp + CodeBegin]
    mov     ecx,CodeLen
    xor     eax,eax
    cld
    rep     stosb
CodeEnd:
```

原来这段代码的功能是调用 MessageBoxA 函数显示一段信息，然后把自身的代码清零，最后程序结束。

现在是否觉得 SMC 并不难理解。下面来认识一下 SMC，SMC 是英文 Self-Modifying Code 的缩写形式，也就是说，可以在一段代码执行之前先对它进行修改。利用 SMC 技术的这个特点，在设计加密方案时，可以把代码以加密形式保存在可执行文件中，然后在程序执行时再动态解密，这样可以有效地对付静态分析。因此如果要了解被加密的代码的功能，那么只有动态跟踪或者分析出解密函数的位置编写程序来解密这些代码。

对于前面的例子，利用单步跟踪可以很容易得到解密的代码，即便是对于静态分析，它的解密函数也显

得过于简单。现在来思考一下怎样才能在加密中更好地利用 SMC 技术呢？

首先，可以在解密函数中把代码的解密与自身保护以及反单步跟踪、反断点跟踪结合起来。

其次，还可以利用 SMC 技术设计出多层嵌套加密的代码。如图 14.1 所示，在第一层代码中解密出第二层的代码，而第二层代码则解密出第三层的代码，依此类推。

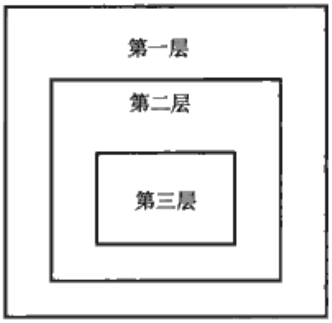


图 14.1　多层嵌套加密的代码示意图

另外，可以设计出一个比较复杂的解密函数。针对在最外层的解密函数，还可以把它分散在程序中的多处，使其隐蔽性更强。

经过以上的考虑，就可以初步实现 SMC 技术的反跟踪实例了。在介绍第二个实例之前，首先来了解一下它的实现方法，如图 14.2 所示。

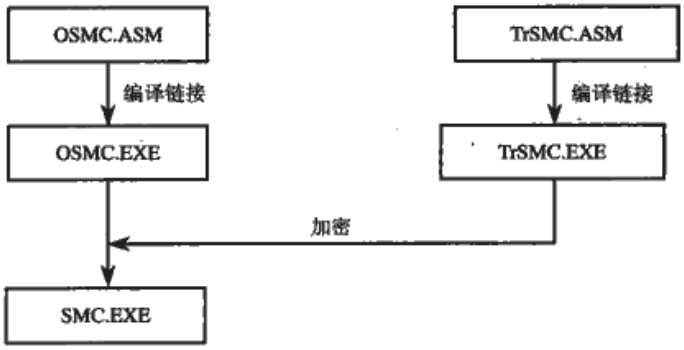


图 14.2　SMC 的实现方法

在前面的介绍中，已经了解到 SMC 代码是以加密形式保存在文件中的，为此必须有一段代码实现加密功能，对未加密形式的 SMC 代码进行加密。在这个例子中，OSMC.ASM 文件包含有完整的未加密形式的 SMC 代码，经过编译后可以得到 OSMC.EXE 文件，而 TrSMC.ASM 文件中包含了加密函数以实现对 OSMC.EXE 的加密。运行编译后的 TrSMC.EXE，即可以对 OSMC.EXE 文件中的 SMC 代码进行加密，从而得到最终的 SMC.EXE 文件。

OSMC.ASM 代码片段：

```
Main:
        assume  fs:nothing
        ;-=-=建立结构化异常处理=-=-
        invoke  SetUnhandledExceptionFilter,addr Final_Handler
        mov     [OldFinalHandler],eax
        mov     [OldEsp],esp
        ;-=-=解密以后代码块=-=-
        push    SIZE_OF_ALLBLOCK
        push    offset Block1
        call    DecryptFunc
        jmp     Block1
```

```
;设置这些数据的目的是为了方便程序加密
FlagData DWORD    0C3C3C3C3h
         DWORD    offset Block1
         DWORD    SIZE_OF_BLOCK1
         DWORD    offset Block2
         DWORD    SIZE_OF_BLOCK2
         DWORD    offset Block3
         DWORD    SIZE_OF_BLOCK3
         DWORD    offset Block4
                                        ; 数据 A
;-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
;第一层代码，也是最外层代码
Block1:
        callloc_next

        ;-=-=这里是异常处理函数的起始地址=-=-
        mov       esi,[esp+4]
        assume    esi:ptr EXCEPTION_RECORD
        mov       edi,[esp+0Ch]
        assume    edi:ptr CONTEXT
        cmp       [esi].ExceptionCode,EXCEPTION_SINGLE_STEP
        jz        @F
        mov       eax,ExceptionContinueSearch
        ret
@@:     mov       eax,[edi].regEax
        xchg      eax,[edi].regEdx      ; 代码 B
        mov       [edi].regEax,eax
        xor       eax,eax
        mov       [edi].iDr0,eax
        and       [edi].iDr1,eax
        and       [edi].iDr2,eax
        and       [edi].iDr3,eax
        and       [edi].iDr6,0FFFF0FF0h
        and       [edi].iDr7,eax
        mov       [edi].ContextFlags,CONTEXT_ALL
        mov       eax,ExceptionContinueExecution
        ret

        ;-=-=建立 SEH 机制=-=-
loc_next:         pushfs:[0]
        mov       fs:[0],esp
        ;-=-=以下的大循环用于解密下一个代码块的数据=-=-
        mov       edi,offset Block2
        mov       esi,edi
        xor       ecx,ecx
        mov       edx,INIT_KEY
loc_loop1:  push     esi
        push      ecx            ;入栈保存寄存器值
;-=-=以下小循环用于计算解密用的密钥=-=-
        mov       esi,offset Block1
        add       ecx,SIZE_OF_BLOCK1
        ;-=-=让当前代码块的字节参与密钥计算，实现自身保护=-=-   ; 代码 A
@@:     lodsd
        xor       eax,edx
```

```
            xor      eax,ecx

    ;===把反跟踪与计算相结合===                              这里将产生单步异常
            pushf
            or       byte ptr [esp+1],01h
            popf
            nop
            loop     @B
            ;========小循环结束========
            pop      ecx
            pop      esi
            ;===反跟踪代码===
            ;代码 C
            pushad
            mov      edi,offset sContext
            mov      ecx,sizeof CONTEXT
            xor      eax,eax
            cld
            rep      stosb
            invoke   GetCurrentThread
            mov      edi,eax
            mov      [sContext].ContextFlags,CONTEXT_ALL
            invoke   GetThreadContext,edi,addr sContext
            mov      [sContext].iDr7,11h
            invoke   SetThreadContext,edi,addr sContext
            popad
            ;===解密下一代码块的一个字节===
            lodsd
            xor      eax,edx
            stosd
            ;===变换密钥===
            xor      edx,HASH_NUM1
            xor      edx,ecx
            inc      ecx
            cmp      ecx,SIZE_OF_BLOCK2
            jnz      loc_Loop1
            ;========大循环结束========
            pop      fs:[0]
            add      esp,4
    ;第一层代码结束
    ;=========================================
    ;第二层代码开始

    Block2:     ....(省略,见完整源代码)

    Block3:     ....(省略,见完整源代码)
```

读者可以看到在程序中设置了一个标志块（数据 A 部分），设置这些数据的目的是为了 TrSMC.EXE 加密 SMC 代码的方便，利用这些数据 TrSMC.EXE 可以很容易地定位出 SMC 块的信息，当然 TrSMC.EXE 在加密完成之后，这些数据会被清除，以免被跟踪者利用。

在这个程序中共有三层 SMC 代码，从外到内分别是 Block1 块、Block2 块和 Block3 块。每一块的解密方法大体相同，同时加入一些反跟踪的技术。这里主要分析一下 Block1 块的代码。为了防止动态跟踪，在 Block1 块中设计了如下的方法：

① 每次只解密出加密代码块的一个字节，利用循环的方式，解密出所有的加密代码。这样循环中的

反跟踪代码会多次执行，以防止跟踪者不修改代码直接通过修改寄存器值的方法跳过反跟踪代码。

② 在循环体中，对当前解密函数的代码进行自身保护，防止被断点跟踪，也就是对解密函数的代码进行校验（代码 A 部分），并把校验值参与解密过程。这样一旦解密函数中的代码被修改，将无法正确地进行解密。

③ 计算校验值的同时在代码中还加入了利用 SEH 技术的反跟踪方法。在程序中利用了修改标志寄存器的方法产生一个单步异常，在 SEH 的异常处理函数中将会交换 EAX 和 EDX 寄存器的值（代码 B 部分），同时还清除了线程上下文（CONTEXT 结构）中 DRx 成员的值，防止 BPM 一类的断点，这样如果异常处理程序不能正常执行，那么解密函数也无法正确解密。

④ 另外，在循环中也加入了另一种清除调试寄存器断点的方法（代码 C 部分），来防止动态跟踪。这样对按 F7 键进行跟踪的方法也进行了有效的防护。

在以上的例子中，尽量在设计中把使用 SMC 技术与其他反跟踪技术结合起来，这样才能充分发挥 SMC 的威力。另外，用 SMC 技术进行多层加密时，可能存在一定的调试难度。

14.2.3 信息隐藏

目前，大多数软件在设计时，都采用了人机对话方式。所谓人机对话，即在软件运行过程中，需要由用户选择的地方，软件即显示相应的提示信息，并等待用户按键选择。而在执行完某一段程序之后，便显示一串提示信息，以反映该段程序运行后的状态，是正常运行，还是出现错误，或者提示用户进行下一步工作的帮助信息。因此，解密者可根据这些提示信息迅速找到核心代码。为了安全，就要对这些敏感文字进行隐藏处理。

现在假设有如下的逻辑：

```
if condition then
   showmessage(0, 'You see me!', 'You see me!', 0);
```

编译后的程序用反汇编工具 W32Dasm 反汇编得到如下的形式：

```
:00401110 85C0              test eax, eax          ; 判断
:00401112 755B              jne 0040116F
:00401114 50                push eax
* Possible StringData Ref from Data Obj ->"You see me!"
:00401115 6838504000        push 00405038
* Possible StringData Ref from Data Obj ->"You see me!"
:0040111A 6838504000        push 00405038
:0040111F 50                push eax
* Reference To: USER32.MessageBoxA, Ord:01BEh
:00401120 FF15A0404000      Call dword ptr [004040A0]
```

在对该段代码进行破解时，很容易由静态的反汇编文本中对文本“You see me!”的引用，快速定位到条件的判断位置，将条件转移条件改掉。

同样逻辑，首先将文字内容做了隐藏变化，然后在程序中使用到该文字的地方首先对文字内容进行还原。使用如下：

```
:00401110 85C0              test eax, eax          ; 判断
:00401112 755B              jne 0040116F
:00401114 50                push eax
* Possible StringData Ref from Data Obj ->"Tbx-"
:00401115 6838504000        push 00405038
* Possible StringData Ref from Data Obj ->" Tbx-"
:0040111A 6838504000        push 00405038
```

```
:0040111F 50                    push eax
* Reference To: USER32.MessageBoxA, Ord:01BEh
:00401120 FF15A0404000          Call dword ptr [004040A0]
```

对一些关键数据进行了隐藏处理后，在一定程度上可以增加静态反编译的破解难度。如将“软件已经过期，请购买”等数据进行隐藏处理，就可以有效防止那些利用静态反汇编，根据这些信息快速找到程序判断点进行快速破解。信息隐藏实现思路有多种，比如将要显示的字符加密存放，需要时，解密后再显示。

14.2.4 简单的多态变形技术

病毒的世界里总是充满了多态（Polymorphic）变形（Metamorphic）和混乱（Obfucscation），从 ASProtect 开始，外壳中也流行这些东西了。除了虚拟化（Virtualization）外，这些技术的确拥有非常好的抗分析效果。

多态和变形（也称为变态）两个术语不是特别容易区分开，事实上也没有太多的必要区分，不过从病毒制造者的角度来看，多态引擎往往意味着它会把病毒用某种算法编码，算法可能是即时生成的，也可能是密码学算法，然后引擎会生成一个充满了干扰指令的解码器，以便在运行时对病毒代码解码，这样可能会使依靠静态扫描特征码的杀毒软件失去作用。不过，现在的杀毒软件可以在运行期扫描特征码，所以多态的效果大不如前，并且因为多态要还原出明文代码，因此对于反调试也没有多大作用。多态就像代码的一层壳，过去就没什么了。

Z0MBiE 的 Kewl Mutation Engine（KME552）引擎变化很复杂，用堆栈解码，静态分析还是比较困难的。不过杀毒似乎不喜欢这个引擎，遇到了直接认为是 Win32 Crypt 病毒。B0z0 的 Expressway To My Skull(ETMS) 能选择算法，生成的代码看起来也很复杂，可惜如果做壳只能支持 EXE，因为生成的 Decryptor 只能在固定的 EIP 运行。tElock 0.98 用到了 Benny’s Polymorphic Engine（BPE32），可以生成无用的 SEH 干扰调试，可惜它加密数据仅仅是 XOR，很容易被已知明文攻击，因此只推荐作为编写 polymorph 的一个例子，不要应用到实际当中。

变形则是把一段代码重新编码，虽然仍然使用 x86 指令集，但是例如“add eax, 5”可能会被换成等价的 5 个 inc eax，并且可以用 jmp 打乱代码的顺序。诸如此类的变换使代码迅速膨胀，因此注重小体积的病毒并不过多地用变形，而在壳中可以不去关心体积，Themida 保护一个几 KB 的小文件会膨胀到将近 2MB，尽量地将代码变形，可以很好地干扰分析人员。ExeCryptor 有不错的强度，就在于它有一个很好的变形混乱引擎，可以把壳代码变得非常难看，当然也非常难以跟踪分析。不过，作者没有对变形引擎自身保护，forgot 曾经修改了它的主程序，迫使它产生了一个没有变形过的外壳体，很轻松就分析完了它的整个过程。因此，如果在壳中使用变形引擎，不要忘记保护主程序的引擎代码。

变形引擎编写比较困难，因为它需要一个反汇编引擎，还要能对代码块进行分析，似乎没有见到适合用来保护代码的开源作品，不过 Z0MBiE 的 Mistfall 2.0 实现了类似的功能，可以作为参考。

混乱通常以多态变形引擎的垃圾生成器出现，一般是指一些花指令和无用的代码。有时候变形也被叫做混乱，并且伴随着扩散，这里所谓的扩散跟香农的信息论不是一回事，只是对混乱过的代码再次混乱，更进一步地消除原始代码的特征。

要想还原变形，就必须写一个相应的收缩程序（Shrinker，因为变形也被称为膨胀），这也同样需要一个反汇编引擎，在代码中插入的数据一直是所有反汇编引擎头疼的东西，即使是 IDA 也没有把握区分出谁是数据谁是代码，因此推荐在引擎中插入一些难辨真伪的数据，使反汇编掉入陷阱，这样就很难编写变形代码的清除程序了。

变形引擎主要来自病毒，VX Heaven（http://vx.netlux.org）上有非常多的病毒资料，几乎所有的病毒引擎都可以下载。

14.3 文件完整性检验

在软件保护方案中，建议增加对软件自身的完整性检查，以防止解密者修改程序以达到破解的目的。这包括对磁盘文件和内存映像的检查，DLL 和 EXE 之间也可以互相检查完整性。

文件完整性检验的原理就是文件发布时用散列函数计算文件的散列值，并将此值放在某处，以后文件每次运行时，重新计算文件的散列值，并与原散列值比较，以判断文件是否被修改。

14.3.1 磁盘文件校验实现

由于 CRC-32 算法的代码比较简短，故本节的自校验实例用 CRC 来实现。CRC 算法可以对一段字符串进行 CRC-32 转换，最后可以得到一个 4 个字节长的 CRC-32 值。当要实现完整性校验时，首先将需要校验的文件当做一段字符串，计算该字符串的 CRC-32 值，然后在文件的某个地方储存这个 CRC-32 值。当文件运行的时候，对文件重新进行 CRC-32 计算，再与原储存的 CRC-32 值进行比较，如果文件有改动，则 CRC-32 值就变化了，这样就实现了文件完整性检查的目的。

为了简单起见，本文将计算文件 CRC-32 值的起始地址选在 PE 文件头开始处，结束位置定在整个文件的末尾，如图 14.3 所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
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	mode....$.......
00000080	2C	46	F0	F6	68	27	9E	A5	68	27	9E	A5	68	27	9E	A5	,F拨h'濒h'濒h'濒
00000090	80	38	94	A5	7E	27	9E	A5	EB	3B	90	A5	61	27	9E	A5	€8敫~'濒?愉a'濒
000000A0	0A	38	8D	A5	6D	27	9E	A5	68	27	9F	A5	42	27	9E	A5	.8岔m'濒h'嫂B'濒
000000B0	80	38	95	A5	6C	27	9E	A5	52	69	63	68	68	27	9E	A5	€8瞳l'濒Richh'濒
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....
000000D0	50	45	00	00	4C	01	03	00	9B	A6	C0	3F	00	00	00	00	PE..L...泱?....

PE 文件头（计算 CRC-32 值的起始地址）

此处放 CRC-32 值

图 14.3 PE 文件头

储存 CRC-32 值的位置可以放在 PE 文件头前那段空间处，也可把 CRC-32 值写入一个单独的文件中，然后在运行的时候读取该文件中储存好的 CRC-32 值，进行比较。

具体实现方法是：先写一个第三方的程序“add2crc32.exe”。可以用这个程序，打开一个目标文件，然后计算出目标文件（从 PE 头开始到文件结束这段数据）的 CRC-32 值，并把这个 CRC-32 值写入到 PE 文件头前那个空白处。

编程写好需要进行保护的程序，在这个程序里面加入 CRC-32 的校验模块。这个校验模块的工作过程是这样的：

① 先读取自身文件从 PE 文件头开始的所有内容，储存在一个字符串中；对这个字符串进行 CRC-32 转换，这时就得到了一个“原始”文件的 CRC-32 值。

② 读取自身文件先前储存的 CRC-32 值（PE 文件头前一个字段），这个值是通过“add2crc32.exe”写进去的。

③ 把步骤①和②中得到的两个 CRC-32 值进行比较，如果相等，说明文件没有被修改过；反之就说明文件已经被修改了。

实现代码如下：

```
BOOL IsFileModified()
{
PIMAGE_DOS_HEADER        pDosHeader=NULL;
```

```
PIMAGE_NT_HEADERS       pNtHeader=NULL;
DWORD fileSize,OriginalCRC32,NumberOfBytesRW;
TCHAR  *pBuffer ,szFileName[MAX_PATH];

GetModuleFileName(NULL,szFileName,MAX_PATH);//获得文件名
HANDLE hFile = CreateFile(szFileName,GENERIC_READ,1,NULL,3,FILE_ATTRIBUTE_NORMAL,NULL);
if ( hFile == INVALID_HANDLE_VALUE ) return FALSE;

fileSize = GetFileSize(hFile,NULL); //获得文件长度
if (fileSize == 0xFFFFFFFF) return FALSE;
pBuffer = new TCHAR [fileSize];
ReadFile(hFile,pBuffer, fileSize, &NumberOfBytesRW, NULL);
CloseHandle(hFile);  //关闭文件
pDosHeader=(PIMAGE_DOS_HEADER)pBuffer;
pNtHeader=(PIMAGE_NT_HEADERS32)((DWORD)pDosHeader+pDosHeader->e_lfanew);
 //定位到 PE 文件头（即字符串“PE\0\0”处）前 4 个字节处，并读出储存在这里的 CRC-32 值
OriginalCRC32 =*((DWORD *)((DWORD)pNtHeader-4));
fileSize=fileSize-DWORD(pDosHeader->e_lfanew);//将 PE 文件头前那部分数据去除
if (CRC32((BYTE*)(pBuffer+pDosHeader->e_lfanew),fileSize) == OriginalCRC32 )
    return TRUE;
else
    return FALSE;
}
```

经编译后文件为 crc32.exe，再用光盘映像文件中提供的 add2crc32.exe 打开 crc32.exe，计算文件的 CRC-32 值，并写进 PE 文件头前面一个字段处。今后 crc32.exe 文件一旦被修改，就会自行发现。

解密者可能会自己计算 CRC-32 值，重新写入文件里。解决方法是在计算 CRC-32 值之前，对需要进行转换的字符串做点手脚。例如对这个字符串进行移位、XOR 等操作，然后在最后比较的时候，也用同样的方法反计算出 CRC-32 值。或对计算出来的 CRC-32 值可逆变换处理一下，再写进文件里。

14.3.2　校验和（Checksum）

PE 的可选映像头（IMAGE_OPTIONAL_HEADER）里面，有一个 Checksum 字段，是该文件的校验和。一般 EXE 文件可以是 0，但一些重要的和系统 DLL 及驱动文件必须有一个校验和。

Windows 提供了一个 API 函数 MapFileAndCheckSumA 测试文件的 Checksum，它位于 IMAGEHLP.DLL 链接库里。其原型如下：

```
ULONG MapFileAndCheckSumA(
  IN LPSTR Filename,                          //文件名
  OUT LPDWORD HeaderSum,                      //指向 PE 文件头 Checksum
  OUT LPDWORD new_checksum                    //指向新计算出的 Checksum
);
```

程序一旦运行后，new_checksum 地址处将放当前文件的校验和，old_checksum 地址指向 PE 文件的 checksum 字段。

此保护很脆弱，攻击者很容易用相关工具修正 PE 文件的校验和。一个比较好的办法，是将正确的校验和放在别处（如注册表里），需要时，用这个值与 new_checksum 比较。

14.3.3　内存映像校验

磁盘文件完整性校验可以抵抗解密者直接修改磁盘文件，但对于内存补丁却没效果，因此必须对内存关键代码数据也实行校验。

1. 对整个代码数据校验

每个程序至少有一个代码区块和数据区块。数据区块属性可读写，程序运行时，全局变量通常会放在这里，这些变量数据会动态变化，因此校验这部分是没有意义的。而代码区块属性只读，存放的是程序代码，在程序运行过程中数据是不会变化的，因此用这部分进行内存校验是可行的。

具体实现的思路如下：

① 从内存映像得到 PE 相关数据，如代码区块的 RVA 值和内存大小等。

② 根据得到的代码区块的 RVA 值和内存大小，计算其内存数据的 CRC-32 值。

③ 读取自身文件先前储存的 CRC-32 值（PE 文件头前一个字段），这个值是通过光盘映像文件中提供的 add2memcrc32.exe 写进去的。

④ 比较两个 CRC-32 值。

这样就实现了内存映像的代码区块校验，只要内存数据被修改，都能被发现。这个方法还能有效地抵抗调试器的普通断点，因为调试器一般通过给应用程序代码硬加 INT 3 指令（机器码 CCh）来实现中断，这样就改变了代码区块的数据，计算 CRC-32 值就会与原来的不同。当然用硬件断点不会影响校验值，因为其用了 DR3～DR0 寄存器，没改变原程序代码数据。详细实现代码见本书光盘映像文件。

```
BOOL CodeSectionCRC32( )
{
    PIMAGE_DOS_HEADER       pDosHeader=NULL;
    PIMAGE_NT_HEADERS       pNtHeader=NULL;
    PIMAGE_SECTION_HEADER   pSecHeader=NULL;
    DWORD                   ImageBase,OriginalCRC32;

    ImageBase=(DWORD)GetModuleHandle(NULL);//取基址
    pDosHeader=(PIMAGE_DOS_HEADER)ImageBase;
    pNtHeader=(PIMAGE_NT_HEADERS32)((DWORD)pDosHeader+pDosHeader->e_lfanew);
     //定位到 PE 文件头（即字符串“PE\0\0”处）前 4 个字节处，并读出储存在这里的 CRC-32 值
     OriginalCRC32 =*((DWORD *)((DWORD)pNtHeader-4));
     pSecHeader=IMAGE_FIRST_SECTION(pNtHeader);   //得到第一个区块的起始地址
     //假设第一个区块就是代码区块
     if(OriginalCRC32==CRC32((BYTE*) (ImageBase+pSecHeader->VirtualAddress),
     pSecHeader->Misc.VirtualSize))
        return TRUE;
    else
        return FALSE;
}
```

第 10 章已讲过，PE 文件在磁盘中的数据结构布局和内存中的数据结构布局是一样的，代码区块在磁盘中的数据与内存映像数据是相同的。add2memcrc32.exe 就是根据这个原理计算磁盘文件的代码区块 CRC-32 值，并写入目标文件里的。

如果程序不加壳这样就可直接发行了，但如果用加壳程序来进一步保护时，可能会出错。因为刚才是直接从磁盘文件中读取代码区块的 RVA 值和大小，加壳后，程序读取的是外壳的代码区块 RVA 值和大小，这样计算出来的 CRC-32 校验值当然就不对了。解决办法是编程时直接用代码区块的 RVA 具体值参与计算，这些具体的值可以用 PE 工具（如 LordPE）查看。由图 14.4 可知，代码区块（.text）的 RVA 值为 1000h，大小为 36AEh，将这些值填进源程序中再编译即可。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	000036AE	00001000	00004000	60000020
.rdata	00005000	000007DE	00005000	00001000	40000040
.data	00006000	00002A1C	00006000	00003000	C0000040

图 14.4　查看区块信息

虽然源程序一样，但在不同系统编译，代码区块的大小可能会不同，以当时编译的具体值为准。为了方便加壳，改进后的代码如下：

```
if(OriginalCRC32==CRC32((BYTE*) 0x401000,0x36AE))
    return TRUE;
else
    return FALSE;
```

2．校验内存代码片段

在实际过程中，有时只需对一小段代码进行内存校验，以防止调试工具的 INT 3 断点。实现代码如下：

```
DWORD address1,address2,size;
_asm mov address1,offset begindecrypt;
_asm mov address2,offset enddecrypt;

begindecrypt :   //标记代码的起始地址
MessageBox(NULL,TEXT ("Hello world!"),TEXT ("OK"),MB_ICONEXCLAMATION);
enddecrypt :     //标记代码的结束地址

size=address2-address1;
if(CRC32((BYTE*)address1,size)==0x78E888AE)
  return TRUE;
else
  return FALSE;
```

上述代码中 CRC32()函数的返回值可通过调试器跟踪得到，再填进源代码里重新编译即可。具体的汇编代码如下：

```
00401006  mov     dword ptr [ebp-8], 401014
0040100D  mov     dword ptr [ebp-4], 40102B
// 校验代码的起始处
00401014  push    esi
00401015  mov     esi, dword ptr [404094]
0040101B  push    30
0040101D  push    4050A4
00401022  push    405094
00401027  push    0
00401029  call    esi                          ;MessageBoxA
// 校验代码的结束处
0040102B  mov     ecx, dword ptr [ebp-4]
```

跟踪调试时，如对 401014h～40102Bh 之间代码设 INT 3 断点时，CRC 校验将发生变化，从而发现程序被跟踪。实际操作时，可以不提示断点被发现，而是悄悄退出，使得校验更隐蔽。

14.4　代码与数据结合技术

一般程序的代码与数据是分开的，本文提出了一个将序列号与程序代码结合的防护方案设想，在未知正确注册码的情况下，很难被破解。其实现原理是将特征数据（如序列号）与程序的某些关键代码或数据联系起来，比如用序列号或其散列值对程序的关键代码或数据进行解密（当然这些关键代码或数据事先是在软件作者那里进行加密后才发行的）。这样，即使解密者通过修改判断跳转指令可以得到一个看似注册的版本，但是不正确的注册码只会使得解密出来的那段代码或数据全是垃圾，根本无法使用。具体说来可采用如下的方法。

① 在软件程序中有一段加密过的密文 C，这个 C 既可以是注册版本中的一段关键代码，也可以是使用注册版程序的某个功能所必需的数据。

② 当用户输入用户名和序列号之后计算解密用的密钥：密钥 =F（用户名，序列号）。

③ 对密文进行解密：明文 M= Decrypt（密文 C，密钥）。

④ 对解密出来的代码加上异常处理代码，如序列号不正确，产生的垃圾代码定会导致异常发生；如序列号正确，则生成代码正常，不会导致异常产生。这一步也可采用第⑤步。

⑤ 利用某种散列算法计算解出来的明文 M 的校验值：校验值 =Hash（明文 M）。散列算法可以采用 MD5、SHA 等。检查校验值是否正确，如果校验值不正确，说明序列号不正确，就拒绝执行。

14.4.1 准备工作

先选择一段代码作为要加密的数据，可以选择软件某个功能的核心代码。本文将用一实例 Codedata.exe 来演示，此例选择菜单“File/Open”功能代码。为了编程时能方便处理这段代码，用 begindecrypt 和 enddecrypt 两个标签将关键数据括住。代码如下：

```
begindecrypt:                    //需要加密的代码起始标签，即 address1 地址
if(GetOpenFileName (&ofn))
{
    hFile = CreateFile( szFileName,
                        GENERIC_READ ,
                        NULL,
                        NULL,
                        OPEN_EXISTING,
                        FILE_ATTRIBUTE_NORMAL ,
                        NULL);
    if( hFile != INVALID_HANDLE_VALUE )
    ……（省略若干代码，详见光盘）
}
delete pBuffer;
bSucceed=true;
enddecrypt:                      //需要加密的代码结束标签，即 address2 地址
```

编译后 begindecrypt 和 enddecrypt 之间的数据称为明文 M。程序编译好后必须对明文 M 加密处理（可以用光盘映像文件中提供的 Encrypter.exe 工具进行处理），得到的数据就是密文 C，这样才能发行。

软件执行时，调用软件自定义某种算法计算解密用的密钥：

密钥 k = F（用户名，序列号）

然后对密文 C 进行解密：

明文 D= Decrypt（密文 C，密钥）

再利用 SEH 来处理加密代码，如解密出的是乱码，则会触发异常，这样就可利用 SEH 告知用户注册失败的提示消息。

源码如下：

```
_asm mov address1,offset begindecrypt     // 取待加密代码首地址
_asm mov address2,offset enddecrypt       // 取待加密代码末地址
//对输入的注册码进行一定的变换，得到密钥 k，k = F（注册码）
k=1;
for (unsigned int i=0;i<strlen(cCode);i++)
{
    k = k*6 + cCode[i];
```

```
}
Size=address2-address1;
ptr=(DWORD*)address1;
if(!bSucceed)
Decrypt (ptr,Size,k); //执行解密函数
//如 Decrypt()函数没解出正确的代码，则会异常
try//异常处理
{
 // 在十六进制工具中用下面两行代码定位加密代码起始处
 _asm inc eax // 在十六进制工具中对应 0x40
 _asm dec eax // 在十六进制工具中对应 0x48
begindecrypt:
……(此段为密文 C)
enddecrypt:
// 在十六进制工具中用下面两行代码定位加密代码结束处
_asm inc eax // 在十六进制工具中对应 0x40
_asm dec eax // 在十六进制工具中对应 0x48
return TRUE;
   }
//如果密文 C 解密不成功，则执行这些代码时必定异常，异常后就会跳到如下代码处
catch (...)
{
    MessageBoxA (NULL, TEXT ("请注册，以获得完整的功能 !"), TEXT ("提示"), 0) ;
    Decrypt (ptr,Size,k);
    return FALSE;
 }
```

14.4.2　加密算法选用

解密算法应该是整个设计的关键。确保算法应该无法让人逆推，因为程序代码并不是真正的随机数据，比如某些指令出现的几率较高，因此存在着被攻击的可能性。所以 Decrypt 算法使用不对称算法很重要，否则攻击者可以假设明文，反推密钥进行攻击。

此例便于讲解方便，故选用了最简单的 XOR 来加密数据。

注意：由于程序代码的数据并不是真正随机数，攻击此类 XOR 加密只是几分钟的事，实际应用时请选用合适的密码学算法。

具体代码如下：

```
void Decrypt (DWORD* pData,DWORD Size,DWORD value)
{
Size=Size/0x4; //对数据共需要异或的次数
//解密 begindecrypt 与 enddecrypt 标签处的数据
while(Size--)
{
    *pData=(*pData)^value;
    pData++;
}
}
```

14.4.3　手动加密代码

Codedata.exe 程序编译好后，必须进一步处理明文 M 这段代码。为了能在十六进制工具中方便找到明

文 M 这段代码，待加密的源代码用了下面两行汇编代码括住：

```
_asm inc eax     机器码是 40h
_asm dec eax     机器码是 48h
```

这样，上面的代码变成了：

```
// 待加密的代码
_asm inc eax // 在十六进制工具中对应 40
_asm dec eax // 在十六进制工具中对应 48
begindecrypt:

        ……（密文 C）

enddecrypt:
_asm inc eax // 在十六进制工具中对应 40
_asm dec eax // 在十六进制工具中对应 48
```

文件编译好后，用十六进制工具打开文件，搜索十六进制数据“4048”，就能找到待加密的代码明文 M。其中开始地址 address1 值为 143Bh，结束地址 address2 值为 14Feh，如图 14.5 黑影部分所示。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00001430	C4	0C	C7	45	FC	00	00	00	00	40	48	68	F0	65	40	00	Ä.ÇEü....@Hhðe@.
00001440	E8	8B	01	00	00	85	C0	0F	84	9E	00	00	00	6A	00	68	鑁...呷.?...j.h
00001450	80	00	00	00	6A	03	6A	00	6A	00	8D	95	DC	FE	FF	FF	€...j.j.j.嗇蔑
00001460	6B	00	00	00	80	52	FF	15	0B	50	40	00	8B	F0	83	FE	k...€R ..P@.嬪澀
00001470	FF	74	64	8D	45	EC	50	56	FF	15	14	50	40	00	3D	00	td岷霧V ..P@.=
00001480	00	01	00	7D	4B	8D	4D	EC	6A	00	51	50	53	56	FF	15	...}K峄鹬.QPSV .
00001490	18	50	40	00	85	C0	74	36	8B	55	08	53	52	FF	15	D0	.P@.呷t6嬪.SR .Ð
000014A0	50	40	00	56	FF	15	88	50	40	00	53	E8	26	01	00	00	P@.V .壳@.S?...
000014B0	B8	01	00	00	00	83	C4	04	A3	44	66	40	00	8B	4D	F4	7...颓. f@.猱Iô
000014C0	64	89	0D	00	00	00	00	5F	5E	5B	8B	E5	5D	C2	04	00	d?....._^[嫫]?..
000014D0	56	FF	15	88	50	40	00	6A	00	68	74	61	40	00	68	54	V .壳@.j.hta@.hT
000014E0	61	40	00	6A	00	FF	15	D4	51	40	00	53	E8	E5	00	00	a@.j. ..Q@.S桢..
000014F0	00	83	C4	04	C7	05	44	66	40	00	01	00	00	00	40	48	.鈧Ä.Ç.Df@....,@H

图 14.5　待加密的代码块

接着必须编写一个工具 Encrypter.exe（程序及源码见光盘映像文件），以执行如下函数功能：

密文 C= Encrypt（明文 M，密钥）

然后运行 Encrypter.exe 工具打开待加密的文件，填写通过十六进制工具得到的 address1 与 address2 的值（十六进制格式），再填上所需要的注册码，此处假设为“pediy”，如图 14.6 所示。

单击“Encrypt!”按钮，即可将明文 M 加密成密文 C。

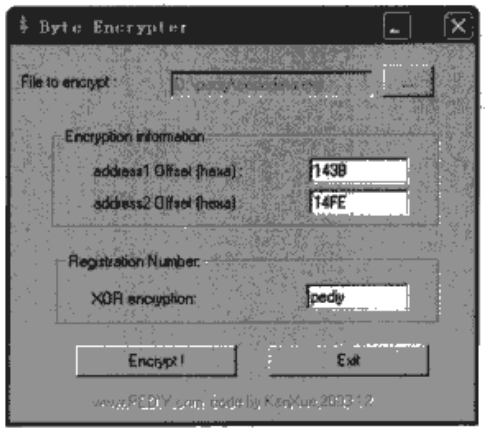


图 14.6　Encrypter.exe 工具运行界面

14.4.4　使.text 区块可写

在 Win32 平台上，文件编译后，.text 区块的属性是只读的。但是，本节实例会向.text 区块写入新的数据（“xor byte ptr[esi],al”这句指令），由于.text 区块只读，这样会导致程序崩溃。

因此必须用 PE 工具，如 LordPE 或 Prodump 等改变.text 区块的属性（characterics）为 E0000020h，表示可读、可写、可执行，如图 14.7 所示。

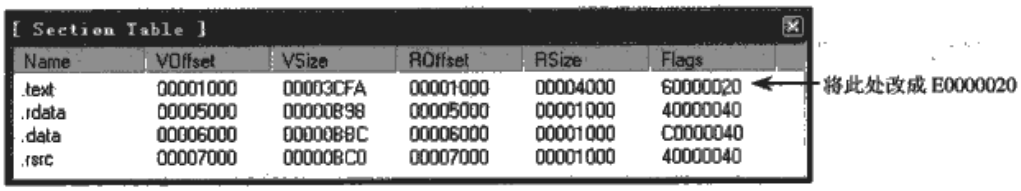


图 14.7　改写代码区块的属性

还有一种方法不用手工修改区块属性，而是编程时用 VirtualProtect 等函数修改内存的读写属性，这样就可直接向.text 等区块写数据了。本例采用的代码如下：

```
void Decrypt (DWORD* pData,DWORD Size,DWORD value)
{
 //首先要做的是改变这一块虚拟内存的内存保护状态，以便可以自由存取代码
MEMORY_BASIC_INFORMATION mbi_thunk;
 //查询页信息
VirtualQuery(pData, &mbi_thunk, sizeof(MEMORY_BASIC_INFORMATION));
//改变页保护属性为读写
VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize,PAGE_READWRITE, \
                                                    &mbi_thunk.Protect);

    Size=Size/0x4; //对数据共需要异或的次数
    //解密 begindecrypt 与 enddecrypt 标签处的数据
    while(Size--)
    {
        *pData=(*pData)^value;
        pData++;
}

//恢复页的原保护属性
DWORD dwOldProtect;
VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize, mbi_thunk.Protect,&dwOldProtect);
}
```

由于此例为 EXE，故没有提到重定位问题的解决。对于 DLL 文件，把代码作为加密对象而言，如果代码中有重定位数据，则加密后的密文解密后还需要对其进行重定位，否则这段代码就算是正确解密也无法运行。希望读者注意这点。

14.5　软件保护的若干忠告

本节将给出关于软件保护的一般性建议，这些都是无数人的经验总结。程序员在设计自己的保护方式时最好遵守这里给出的准则，这样会提高软件的保护强度。

（1）尽量开发自己的保护机制，不要过分依赖不是自己开发的任何代码。在不影响效率的情况下，保护的核心代码用虚拟机保护软件处理一下，如 VMProtect 等。

（2）不要太依赖壳的保护，加密壳都能被解开或脱壳，现在许多壳转向虚拟机加密方向，多利用这方面的功能。如果时间允许且有相应的技术能力，可以设计自己的加壳/压缩方法。如果采用现成的加壳工具，最好不要选择流行的工具，保护强度与流行性成反比，因为越是流行的工具越是可能已被广泛深入地研究过，即有了通用的脱壳/解密办法。

（3）增加对软件自身的完整性检查。这包括对磁盘文件和内存映像的检查，以防止有人未经允许就修改程序以达到破解的目的。DLL 和 EXE 之间可以互相检查完整性。

（4）不要采用一目了然的名字来命名函数和文件，比如 IsLicensedVersion()、key.dat 等。所有与软件加密相关的字符串都不能以明文形式直接存放在可执行文件中，这些字符串最好是动态生成的。

（5）尽可能少地给用户提示信息，因为这些蛛丝马迹都可能导致解密者直接深入到加密的核心。比如，当检测到破解企图之后，不要立即给用户提示信息，而是在系统的某个地方做一个记号，随机地过一段时间后使软件停止工作，或者装作正常工作但实际上却在所处理的数据中加入了一些垃圾。

（6）将注册码和安装时间记录在多个不同的地方。

（7）检查注册信息和时间的代码越分散越好。不要调用同一个函数或判断同一个全局标志，因为这样做，只要修改了一个地方则全部就都被破解了。

（8）不要依赖 GetLocalTime()和 GetSystemTime()这种众所周知的函数获取系统时间，可以通过读取关键的系统文件的修改时间来得到系统时间的信息。

（9）如果有可能，可以采用联网检查注册码的方法，并且数据在网上传输时要加密。

（10）编程时在软件中嵌入反跟踪的代码，以增加安全性。

（11）在检查注册信息的时候插入大量无用的运算以误导解密者，并在查出错误的注册信息之后加入延时。

（12）给软件保护加入一定的随机性，比如除了启动时检查注册码之外，还可以在软件运行的某个时刻随机地检查注册码。随机值还可以很好地防止那些模拟工具，如软件狗模拟程序。

（13）如果采用注册码的保护方式，最好是一机一码，即注册码与机器特征相关。这样一台机器上的注册码就无法在另外一台机器上使用，可以防止有人散播注册码；并且机器号的算法不要太迷信硬盘序列号，因为用相关工具可以修改其值。

（14）如果试用版与正式版是分开的两个版本，且试用版的软件没有某项功能，则不要仅仅使相关的菜单变灰，而是彻底删除相关的代码，使得编译后的程序中根本没有相关的功能代码。

（15）如果软件中包含驱动程序，则最好将保护判断加在驱动程序中。因为驱动程序在访问系统资源时受到的限制比普通应用程序少得多，这也给了软件设计者发挥的余地。

（16）如果采用 keyfile 的保护方式，则 keyfile 的尺寸不能太小，可将其结构设计得比较复杂，在程序中不同的地方对 keyfile 的不同部分进行复杂的运算和检查。

（17）自己设计的检查注册信息的算法不能过于简单，最好采用比较成熟的密码学算法。可以在网上找到大量的源码。

对于加密方案的设计读者应该多从解密的角度考虑，这样才可能比较合理地运用各种技术。当然任何加密方案都很难做到完美，因此，在设计时要注意其他方面的平衡考虑。加密方案的好坏或许用 IT 界一句名言来描述很合适："一个木桶能够装多少水，是由最短的那块木板决定的"。

第15章 反跟踪技术[①]

好的软件保护都要与反跟踪技术结合在一起。如果没有反跟踪技术，软件等于直接裸露在解密者的面前。这里所说的反跟踪是泛指，包括防调试器、防监视工具等内容。本章将讨论一些常用的反跟踪方法，读者可以根据实际情况在自己的软件中采用相关的技术和代码。

15.1 由 BeingDebugged 引发的蝴蝶效应

一个坏的微小的机制，如果不加以及时地引导、调节，会给社会带来非常大的危害，戏称为“龙卷风”或“风暴”；一个好的微小的机制，只要正确指引，经过一段时间的努力，将会产生轰动效应，或称为“革命”。

15.1.1 BeingDebugged

Win32 API 为程序提供了 IsDebuggerPresent 判断自己是否处于调试状态，懒惰的程序员总是用它。来看一下实现代码：

```
// debug.c
BOOL
APIENTRY
IsDebuggerPresent(VOID)
{
    return NtCurrentPeb()->BeingDebugged;
}
```

这个函数读取了当前进程PEB中的BeingDebugged标志，每个运行中的进程拥有一个名为PEB（Process Environment Block，进程环境块）的结构，对它有少许了解会有助于理解后面的内容。PEB 结构的内容：

```
Offset    Elements name              Type
+0x000    InheritedAddressSpace      : UChar
+0x001    ReadImageFileExecOptions   : UChar
+0x002    BeingDebugged              : UChar
+0x003    SpareBool                  : UChar
+0x004    Mutant                     : Ptr32 Void
+0x008    ImageBaseAddress           : Ptr32 Void
+0x00c    Ldr                        : Ptr32 _PEB_LDR_DATA
```

[①] 本章由林子深编写。

```
+0x010   ProcessParameters              : Ptr32 _RTL_USER_PROCESS_PARAMETERS
+0x014   SubSystemData                  : Ptr32 Void
+0x018   ProcessHeap                    : Ptr32 Void
+0x01c   FastPebLock                    : Ptr32 _RTL_CRITICAL_SECTION
+0x020   FastPebLockRoutine             : Ptr32 Void
+0x024   FastPebUnlockRoutine           : Ptr32 Void
+0x028   EnvironmentUpdateCount         : Uint4B
+0x02c   KernelCallbackTable            : Ptr32 Void
+0x030   SystemReserved                 : [1] Uint4B
+0x034   ExecuteOptions                 : Pos 0, 2 Bits
+0x034   SpareBits                      : Pos 2, 30 Bits
+0x038   FreeList                       : Ptr32 _PEB_FREE_BLOCK
+0x03c   TlsExpansionCounter            : Uint4B
+0x040   TlsBitmap                      : Ptr32 Void
+0x044   TlsBitmapBits                  : [2] Uint4B
+0x04c   ReadOnlySharedMemoryBase       : Ptr32 Void
+0x050   ReadOnlySharedMemoryHeap       : Ptr32 Void
+0x054   ReadOnlyStaticServerData       : Ptr32 Ptr32 Void
+0x058   AnsiCodePageData               : Ptr32 Void
+0x05c   OemCodePageData                : Ptr32 Void
+0x060   UnicodeCaseTableData           : Ptr32 Void
+0x064   NumberOfProcessors             : Uint4B
+0x068   NtGlobalFlag                   : Uint4B
+0x070   CriticalSectionTimeout         : _LARGE_INTEGER
+0x078   HeapSegmentReserve             : Uint4B
```

接下来的问题当然是如何找到 PEB 地址。它储存在另一个名为线程环境块（Thread Environment Block，TEB）的结构之内。

Windows 在调入进程，创建线程时，操作系统均会为每个线程分配 TEB，而且 FS 段寄存器总是被设置成使得地址 FS:0 指向当前线程的 TEB 数据（单 CPU 机器在任何时刻系统中只有一个线程在执行），这就为存取 TEB 数据提供了途径，如图 15.1 所示。

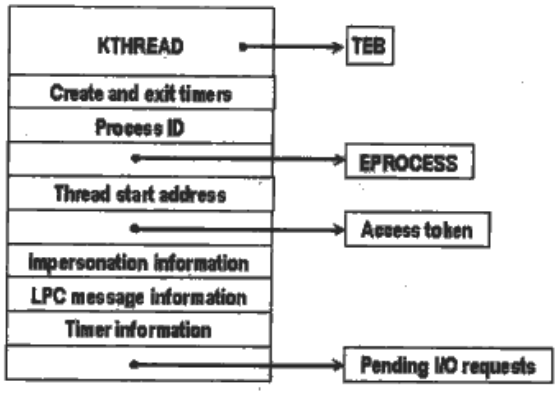


图 15.1　执行线程块的结构

再来了解一下 TEB 的结构，请注意+30h 处的偏移字段。

```
+0x000 NtTib    :                       :_NT_TIB
+0x01c EnvironmentPointer               : Ptr32 Void
+0x020 ClientId                         : _CLIENT_ID
+0x028 ActiveRpcHandle                  : Ptr32 Void
+0x02c ThreadLocalStoragePointer        : Ptr32 Void
+0x030 ProcessEnvironmentBlock          : Ptr32 _PEB
```

```
+0x034 LastErrorValue              : Uint4B
+0x038 CountOfOwnedCriticalSections : Uint4B
+0x03c CsrClientThread             : Ptr32 Void
+0x040 Win32ThreadInfo             : Ptr32 Void
+0x044 User32Reserved              : [26] Uint4B
+0x0ac UserReserved                : [5] Uint4B
+0x0c0 WOW32Reserved               : Ptr32 Void
+0x0c4 CurrentLocale               : Uint4B
+0x0c8 FpSoftwareStatusRegister    : Uint4B
+0x0cc SystemReserved1             : [54] Ptr32 Void
+0x1a4 ExceptionCode               : Int4B
+0x1a8 ActivationContextStack      : _ACTIVATION_CONTEXT_STACK
+0x1bc SpareBytes1                 : [24] UChar
+0x1d4 GdiTebBatch                 : _GDI_TEB_BATCH
+0x6b4 RealClientId                : _CLIENT_ID
+0x6bc GdiCachedProcessHandle      : Ptr32 Void
+0x6c0 GdiClientPID                : Uint4B
+0x6c4 GdiClientTID                : Uint4B
+0x6c8 GdiThreadLocalInfo          : Ptr32 Void
+0x6cc Win32ClientInfo             : [62] Uint4B
+0x7c4 glDispatchTable             : [233] Ptr32 Void
+0xb68 glReserved1                 : [29] Uint4B
+0xbdc glReserved2                 : Ptr32 Void
+0xbe0 glSectionInfo               : Ptr32 Void
+0xbe4 glSection                   : Ptr32 Void
+0xbe8 glTable                     : Ptr32 Void
+0xbec glCurrentRC                 : Ptr32 Void
+0xbf0 glContext                   : Ptr32 Void
+0xbf4 LastStatusValue             : Uint4B
+0xbf8 StaticUnicodeString         : _UNICODE_STRING
+0xc00 StaticUnicodeBuffer         : [261] Uint2B
+0xe0c DeallocationStack           : Ptr32 Void
+0xe10 TlsSlots                    : [64] Ptr32 Void
+0xf10 TlsLinks                    : _LIST_ENTRY
+0xf18 Vdm                         : Ptr32 Void
+0xf1c ReservedForNtRpc            : Ptr32 Void
+0xf20 DbgSsReserved               : [2] Ptr32 Void
+0xf28 HardErrorsAreDisabled       : Uint4B
+0xf2c Instrumentation             : [16] Ptr32 Void
+0xf6c WinSockData                 : Ptr32 Void
+0xf70 GdiBatchCount               : Uint4B
+0xf74 InDbgPrint                  : UChar
+0xf75 FreeStackOnTermination      : UChar
+0xf76 HasFiberData                : UChar
+0xf77 IdealProcessor              : UChar
+0xf78 Spare3                      : Uint4B
+0xf7c ReservedForPerf             : Ptr32 Void
+0xf80 ReservedForOle              : Ptr32 Void
+0xf84 WaitingOnLoaderLock         : Uint4B
+0xf88 Wx86Thread                  : _Wx86ThreadState
+0xf94 TlsExpansionSlots           : Ptr32 Ptr32 Void
+0xf98 ImpersonationLocale         : Uint4B
+0xf9c IsImpersonating             : Uint4B
+0xfa0 NlsCache                    : Ptr32 Void
+0xfa4 pShimData                   : Ptr32 Void
```

```
+0xfa8 HeapVirtualAffinity          : Uint4B
+0xfac CurrentTransactionHandle     : Ptr32 Void
+0xfb0 ActiveFrame                  : Ptr32 _TEB_ACTIVE_FRAME
+0xfb4 SafeThunkCall                : UChar
+0xfb5 BooleanSpare                 : [3] UChar
```

00h 处的 TIB（Thread Information Block，线程信息块）结构为：

```
+0x000 ExceptionList                : Ptr32 _EXCEPTION_REGISTRATION_RECORD
+0x004 StackBase                    : Ptr32 Void
+0x008 StackLimit                   : Ptr32 Void
+0x00c SubSystemTib                 : Ptr32 Void
+0x010 FiberData                    : Ptr32 Void
+0x010 Version                      : Uint4B
+0x014 ArbitraryUserPointer         : Ptr32 Void
+0x018 Self                         : Ptr32 _NT_TIB，指向 TEB 结构的指针
```

每个进程都有自己的 PEB，Windows 一般通过 TEB 间接得到 PEB 的地址。即通过以下语句获得：

```
mov eax,fs:[18h]          //获得当前线程的 TEB 地址
mov eax,[eax+30h]         //在 TEB 偏移 30h 处获得 PEB 地址
```

TIB+18h 处为 Self，是 TIB 的反身指针，指向 PEB 首地址，因此可以省略而直接使用 fs:[30h]得到自己进程的 PEB。

为了免去繁冗的定义，这里给出一个内联汇编代码的简化版 IsDebuggerPresent：

```
BOOL MyIsDebuggerPresent(VOID)
{
    __asm {
        mov eax, fs:[0x30]              //在位于 TEB 偏移 30h 处获得 PEB 地址
        movzx eax, byte ptr [eax+2]     //获得 PEB 偏移 2h 处 BeingDebugged 的值
    }
}
```

根据这个原理，OllyDbg 可以用插件清除 BeingDebugged 以隐藏调试器。

虽然在 Windows 2000/NT 系统中 PEB 本身在大多数情况下被映射到 7FFDF000h 处，不过值得注意的是，从 Windows XP SP2 后系统引入了一个特性：PEB 地址随机化。每个进程的 PEB 地址不固定，大概有 14 种可能。

系统创建进程时设置 PEB 的地址，调用 NtCreateProcess/NtCreateProcessEx，依次转向 PspCreateProcess/MmCreatePeb/MiCreatePebOrTeb，在 MiCreatePebOrTeb 函数中根据当前时间计算随机值：

```
PVOID HighestVadAddress;
LARGE_INTEGER CurrentTime;
HighestVadAddress = (PVOID) ((PCHAR)MM_HIGHEST_VAD_ADDRESS + 1);
KeQueryTickCount (&CurrentTime);
CurrentTime.LowPart &= ((X64K >> PAGE_SHIFT) - 1);
if (CurrentTime.LowPart <= 1) {
     CurrentTime.LowPart = 2;
   }
HighestVadAddress = (PVOID) ((PCHAR)HighestVadAddress - (CurrentTime.LowPart<< PAGE_SHIFT));
```

所以不能认为 PEB 就在 7FFDF000h，不同的进程 PEB 地址会不一样。当然也就不能用本进程 FS:[18h]的指针去读写其他进程的内容。正确的方法是使用下面的函数，取得某个线程段选择子的线性地址：

```
BOOL GetThreadSelectorEntry(
  HANDLE hThread,
  DWORD dwSelector,
```

```
    LPLDT_ENTRY lpSelectorEntry
);
```

如果喜欢 Native API，也可以通过 NtQueryInformationProcess 获得 PEB：

```
ULONG GetPebBase(ULONG ProcessId)
{
    HANDLE hProcess = NULL;
    PROCESS_BASIC_INFORMATION pbi = {0}
    ULONG peb = 0;
    ULONG cnt = 0;
    hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, ProcessId);
    if (hProcess != NULL) {
        if (NtQueryInformationProcess(
                hProcess,
                ProcessBasicInformation,
                &pbi,
                sizeof(PROCESS_BASIC_INFORMATION),
                &cnt) == 0) {
            PebBase = (ULONG)pbi.PebBaseAddress;
        }
        CloseHandle(hProcess);
    }
}
```

这里采用 GetThreadSelectorEntry，下面这段代码就可以清除 BeingDebugged 标记了：

```
BOOL HideDebugger( HANDLE hThread,HANDLE hProcess)
{
    CONTEXT ctx;
    LDT_ENTRY sel;
    DWORD fs;
    DWORD peb;
    SIZE_T bytesrw;
    WORD flag;
    ctx.ContextFlags = CONTEXT_SEGMENTS;
    if (!GetThreadContext(hThread, &ctx))
        return FALSE;
    if (!GetThreadSelectorEntry(hThread, ctx.SegFs, &sel))
        return FALSE;
    fs = (sel.HighWord.Bytes.BaseHi << 8 | sel.HighWord.Bytes.BaseMid) << 16 | sel.BaseLow;
    if (!ReadProcessMemory(hProcess, (LPCVOID)(fs + 0x30), &peb, 4, &bytesrw) || bytesrw != 4)
        return FALSE;
    if (!ReadProcessMemory(hProcess, (LPCVOID)(peb + 0x2), &flag, 2, &bytesrw) || bytesrw != 2)
        return FALSE;
    flag = 0;
    if (!WriteProcessMemory(hProcess, (LPCVOID)(peb + 0x2), &flag, 2, &bytesrw) || bytesrw != 2)
        return FALSE;
    return TRUE;
}
```

现在读者一定认为这个标志太愚蠢了，事实上它比你想象的要复杂一些。BeingDebugged 虽然被消灭了，但是问题并不是这么简单。

15.1.2　NtGlobalFlag

先查看一下 Windows 2000 中的源码，BeingDebugged 被清除之前发生了什么事情。相关代码如下：

```
VOID LdrpInitialize (
      IN PCONTEXT Context,
      IN PVOID SystemArgument1,
      IN PVOID SystemArgument2
    )
  //  Routine Description:
......
//#if DBG
      if (TRUE)
//#else
//     if (Peb->BeingDebugged || Peb->ReadImageFileExecOptions)
//#endif
      {
       PWSTR pw;
       pw = (PWSTR)Peb->ProcessParameters->ImagePathName.Buffer;
       if (!(Peb->ProcessParameters->Flags & RTL_USER_PROC_PARAMS_NORMALIZED)) {
            pw = (PWSTR)((PCHAR)pw + (ULONG_PTR)(Peb->ProcessParameters));
            }
        UnicodeImageName.Buffer = pw;
        UnicodeImageName.Length = Peb->ProcessParameters->ImagePathName.Length;
        UnicodeImageName.MaximumLength = UnicodeImageName.Length;

         LdrQueryImageFileExecutionOptions( &UnicodeImageName,
                                            L"DisableHeapLookaside",
                                            REG_DWORD,
                                            &RtlpDisableHeapLookaside,
                                            sizeof( RtlpDisableHeapLookaside ),
                                            NULL
                                          );

         st = LdrQueryImageFileExecutionOptions( &UnicodeImageName,
                                                 L"GlobalFlag",
                                                 REG_DWORD,
                                                 &Peb->NtGlobalFlag,
                                                 sizeof( Peb->NtGlobalFlag ),
                                                 NULL
                                               );
        if (!NT_SUCCESS( st )) {
           if (Peb->BeingDebugged) {// 这里改写了 NtGlobalFlag
            Peb->NtGlobalFlag |= FLG_HEAP_ENABLE_FREE_CHECK |
                                 FLG_HEAP_ENABLE_TAIL_CHECK |
                                 FLG_HEAP_VALIDATE_PARAMETERS;
              }
           }
```

可以看到如果 BeingDebugged 被设为 TRUE，NtGlobalFlag 中也会因此被设置这些标志：

```
FLG_HEAP_ENABLE_FREE_CHECK,
FLG_HEAP_ENABLE_TAIL_CHECK,
FLG_HEAP_VALIDATE_PARAMETERS
...
```

回顾一下 PEB 的结构，+68h 处就是 NtGlobalFlag，用 WinHex 比较内存可以发现，被调试时程序的 NtGlobalFlag=70h，正常情况下却不是。因此得到一个改进的 IsDebuggerPresent 函数：

```
BOOL MyIsDebuggerPresentEx(VOID)
{
    __asm {
```

```
        mov eax, fs:[0x30]
        mov eax, [eax+0x68]
        and eax, 0x70
    }
}
```

ExeCryptor 比较早用到 NtGlobalFlag 做检测，刚开始让许多人莫名其妙了一段时间。注意源码中那个 LdrQueryImageFileExecutionOptions 函数如果成功的话，就不会改写 NtGlobalFlag 了，这个函数事实上读取注册表了。注册表内容：

```
HKLM\Software\Microsoft\Windows Nt\CurrentVersion\Image File Execution Options
```

如果在这里建一个名为进程名、值为空的子键，那么 NtGlobalFlag（及其引发的）的检测都变得无效了。

现在深呼吸一下，逐渐清醒了，这个所谓的新发现也不过是一个标志而已，还是可以像 BeingDebugged 一样被清除掉，可惜就如引发它的 BeingDebugged 一样，虽然被毁灭，它留下的痕迹仍然存在。

15.1.3 Heap Magic

为了掌握 NtGlobalFlag 所留下的痕迹，在 WRK 中找找线索。相关代码如下：

```
PVOID
RtlCreateHeap (
    IN ULONG Flags,
    IN PVOID HeapBase OPTIONAL,
    IN SIZE_T ReserveSize OPTIONAL,
    IN SIZE_T CommitSize OPTIONAL,
    IN PVOID Lock OPTIONAL,
    IN PRTL_HEAP_PARAMETERS Parameters OPTIONAL
    )
    ......
if (NtGlobalFlag & FLG_HEAP_ENABLE_TAIL_CHECK) {
    Flags |= HEAP_TAIL_CHECKING_ENABLED;
    }
if (NtGlobalFlag & FLG_HEAP_ENABLE_FREE_CHECK) {
    Flags |= HEAP_FREE_CHECKING_ENABLED;
    }
if (NtGlobalFlag & FLG_HEAP_DISABLE_COALESCING) {
    Flags |= HEAP_DISABLE_COALESCE_ON_FREE;
}
Peb = NtCurrentPeb();
if (NtGlobalFlag & FLG_HEAP_VALIDATE_PARAMETERS) {
    Flags |= HEAP_VALIDATE_PARAMETERS_ENABLED;
}
if (NtGlobalFlag & FLG_HEAP_VALIDATE_ALL) {
    Flags |= HEAP_VALIDATE_ALL_ENABLED;
}
if (NtGlobalFlag & FLG_USER_STACK_TRACE_DB) {
    Flags |= HEAP_CAPTURE_STACK_BACKTRACES;
}
    ....
#ifndef NTOS_KERNEL_RUNTIME
//
// In the non kernel case check if we are creating a debug heap
// the test checks that skip validation checks is false.
```

```
if (DEBUG_HEAP( Flags )) {
    return RtlDebugCreateHeap( Flags,
                               HeapBase,
                               ReserveSize,
                               CommitSize,
                               Lock,
                               Parameters );
}
#endif // NTOS_KERNEL_RUNTIME
```

其中用到了 DEBUG_HEAP 宏：

```
//heappriv.h
#define HEAP_DEBUG_FLAGS   (HEAP_VALIDATE_PARAMETERS_ENABLED | \
                HEAP_VALIDATE_ALL_ENABLED       | \
                HEAP_CAPTURE_STACK_BACKTRACES   | \
                HEAP_CREATE_ENABLE_TRACING      | \
                HEAP_FLAG_PAGE_ALLOCS)
#define DEBUG_HEAP(F)     ((F & HEAP_DEBUG_FLAGS) && !(F & \
                           HEAP_SKIP_VALIDATION_CHECKS))
```

NtGlobalFlag 因为 BeingDebugged 为 TRUE 的缘故设置了 FLG_HEAP_VALIDATE_ PARAMETERS，因此 RtlCreateHeap 选择用 RtlDebugCreateHeap 创建调试堆。再去翻一翻 RtlDebugCreateHeap 函数中的片段：

```
PVOID
RtlDebugCreateHeap (
    IN ULONG Flags,
    IN PVOID HeapBase OPTIONAL,
    IN SIZE_T ReserveSize OPTIONAL,
    IN SIZE_T CommitSize OPTIONAL,
    IN PVOID Lock OPTIONAL,
    IN PRTL_HEAP_PARAMETERS Parameters
    )
.....
    Heap = RtlCreateHeap( Flags |
                          HEAP_SKIP_VALIDATION_CHECKS |
                          HEAP_TAIL_CHECKING_ENABLED |
                          HEAP_FREE_CHECKING_ENABLED,
                          HeapBase,
                          ReserveSize,
                          CommitSize,
                          Lock,
                          Parameters );
```

原来还是调用 RtlCreateHeap，看来关键起作用的标记是：

```
HEAP_SKIP_VALIDATION_CHECKS
HEAP_TAIL_CHECKING_ENABLED
HEAP_FREE_CHECKING_ENABLED
```

回到 RtlCreateHeap 搜索这些文字，第一个标记看起来是为了防止从 RtlCreateHeap 到 RtlDebugCreate Heap 又回到 RtlCreateHeap 而做过多的重复工作。不过后面两个周围却有一些有趣的内容：

```
//  Otherwise if the flags indicate that we should fill heap then it it now.
} else if (Heap->Flags & HEAP_FREE_CHECKING_ENABLED) {
    RtlFillMemoryUlong( (PCHAR)(BusyBlock + 1), Size & ~0x3, ALLOC_HEAP_FILL );
}
//  If the flags indicate that we should do tail checking then copy
```

```
// the fill pattern right after the heap block.
if (Heap->Flags & HEAP_TAIL_CHECKING_ENABLED) {
    RtlFillMemory( (PCHAR)ReturnValue + Size,
                   CHECK_HEAP_TAIL_SIZE,
                   CHECK_HEAP_TAIL_FILL );
    BusyBlock->Flags |= HEAP_ENTRY_FILL_PATTERN;
}
```

没想到调试堆里面还会被填充一些奇怪的东西，相关的定义也很容易找到：

```
//heap.h
#define CHECK_HEAP_TAIL_SIZE HEAP_GRANULARITY
#define CHECK_HEAP_TAIL_FILL 0xAB
#define FREE_HEAP_FILL 0xFEEEFEEE
#define ALLOC_HEAP_FILL 0xBAADF00D
```

既然堆里面被填充了东西，就可以编写以下代码测试一下：

```
LPVOID GetHeap(VOID)
{
    return HeapAlloc(GetProcessHeap(), NULL, 0x10);
}
```

然后用 OllyDbg 看看返回的指针里面都是什么：

```
00153660  0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
00153670  0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
00153680  0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
00153690  0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
001536A0  0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
......
00153770  0D F0 AD BA 0D F0 AD BA 0D F0 AD BA 0D F0 AD BA
00153780  EE FE EE AB AB AB AB AB AB AB FE EE FE EE FE
00153790  00 00 00 00 00 00 00 00 0D 01 28 00 EE 14 EE 00
001537A0  78 01 15 00 78 01 15 00 EE FE EE FE EE FE EE FE
001537B0  EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
001537C0  EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
001537D0  EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
001537E0  EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
001537F0  EE FE EE FE EE FE EE FE EE FE EE FE EE FE EE FE
```

可以看到果然有很多的 BAADF00D 和 FEEEFEEE，还有 ABABABAB。参考前面的内容，在 Image File Execution Options 中创建一个键，看看正常情况下的堆内容：

```
00153CA8  F0 03 15 00 F0 03 15 00 43 00 45 00 5C 00 3B 00  ? .? .C.E.\.;.
00153CB8  4C 00 03 00 5C 00 57 00 D8 03 15 00 D8 03 15 00  L. .\.W.? .? .
00153CC8  57 00 53 00 5C 00 4D 00 69 00 63 00 72 00 6F 00  W.S.\.M.i.c.r.o.
00153CD8  73 00 6F 00 66 00 74 00 2E 00 4E 00 45 00 54 00  s.o.f.t..N.E.T.
00153CE8  5C 00 46 00 72 00 61 00 6D 00 65 00 77 00 6F 00  \.F.r.a.m.e.w.o.
......
00153D78  63 00 72 00 6F 00 73 00 6F 00 66 00 74 00 20 00  c.r.o.s.o.f.t. .
00153D88  56 00 69 00 73 00 75 00 61 00 6C 00 20 00 53 00  V.i.s.u.a.l. .S.
00153D98  74 00 75 00 64 00 69 00 6F 00 5C 00 43 00 6F 00  t.u.d.i.o.\.C.o.
00153DA8  6D 00 6D 00 6F 00 6E 00 5C 00 54 00 6F 00 6F 00  m.m.o.n.\.T.o.o.
00153DB8  6C 00 73 00 5C 00 57 00 69 00 6E 00 4E 00 54 00  l.s.\.W.i.n.N.T.
00153DC8  3B 00 44 00 3A 00 5C 00 50 00 72 00 6F 00 67 00  ;.D.:.\.P.r.o.g.
```

只是一堆没有初始化的数据而已，没有那些特殊的 Magic 标记。写一个程序，在堆中搜索那些奇怪的

标记，如果出现了很多的话（比如 10 次以上），那么程序就被调试了：

```
LPVOID GetHeap(SIZE_T nSize)
{
    return HeapAlloc(GetProcessHeap(), NULL, nSize);
}

BOOL IsDebugHeap(VOID)
{
    LPVOID HeapPtr;
    PDWORD ScanPtr;
    ULONG nMagic = 0;

    HeapPtr = GetHeap(0x100);

    ScanPtr = (PDWORD)HeapPtr;
    try {
        for(;;) {
            switch (*ScanPtr++) {
                case 0xABABABAB:
                case 0xBAADF00D:
                case 0xFEEEFEEE:
                    nMagic++;
                    break;
            }
        }
    }
    catch(...) {
        return (nMagic > 10) ? TRUE : FALSE;
    }
}
```

这里用到了一个小伎俩，为了尽量完整地对堆进行扫描，此处用死循环一直向下搜索内存，直到内存地址已经无效时，SEH 会捕获错误，此时返回统计的结果。那些奇怪的数字习惯上被称为 Magic，所以这个检测技巧也被称为 HeapMagic。

除了自己从堆申请内存，ap0x 在他的站点公布了一段代码，从被调试程序 PEB 的 LDR_MODULE 中也能搜索到那些用来填坑的标记，事实上，Themida 中发现了一字不差的检测代码。

```
; MASM32 antiRing3Debugger example
; coded by ap0x
; Reversing Labs: http://ap0x.headcoders.net

ASSUME FS:NOTHING
PUSH offset _SehExit
PUSH DWORD PTR FS:[0]
MOV FS:[0],ESP

; Get NtGlobalFlag 口 这里ap0x出现bug，这里获取的是PEB
MOV EAX,DWORD PTR FS:[30h]

; Get LDR_MODULE
MOV EAX,DWORD PTR[EAX+12]

; Note: This code works only on NT systems!
```

```
_loop:
INC EAX
CMP DWORD PTR[EAX],0FEEEFEEEh
JNE _loop
DEC [Tries]
JNE _loop

PUSH 30h
PUSH offset DbgFoundTitle
PUSH offset DbgFoundText
PUSH 0
CALL MessageBox
PUSH 0
CALL ExitProcess
RET
_Exit:
PUSH 40h
PUSH offset DbgNotFoundTitle
PUSH offset DbgNotFoundText
PUSH 0
CALL MessageBox
PUSH 0
CALL ExitProcess
RET

_SehExit:
POP FS:[0]
ADD ESP,4
JMP _Exit
```

抛开那些 BAADF00D，原先发现的 Flags 其实还有利用价值，往下翻翻，RtlCreateHeap 下面还有一些不起眼的操作。代码如下：

```
    // Fill in the heap header fields
    //
    Heap->Entry.Size = (USHORT)(SizeOfHeapHeader >> HEAP_GRANULARITY_SHIFT);
    Heap->Entry.Flags = HEAP_ENTRY_BUSY;
    Heap->Signature = HEAP_SIGNATURE;
    Heap->Flags = Flags;
    Heap->ForceFlags = (Flags & (HEAP_NO_SERIALIZE |
                                 HEAP_GENERATE_EXCEPTIONS |
                                 HEAP_ZERO_MEMORY |
                                 HEAP_REALLOC_IN_PLACE_ONLY |
                                 HEAP_VALIDATE_PARAMETERS_ENABLED |
                                 HEAP_VALIDATE_ALL_ENABLED |
                                 HEAP_TAIL_CHECKING_ENABLED |
                                 HEAP_CREATE_ALIGN_16 |
                                 HEAP_FREE_CHECKING_ENABLED));
```

这里的 Flags 在前面已经被 NtGlobalFlag 影响了，看来进程堆的 Flags 和 ForceFlags 也不会幸免，它们也会感染上那些标记。

事实上，正常情况下系统为进程创建第一个堆时，会将它的 Flags 和 ForceFlags 分别设为 2（HEAP_GROWABLE）和 0，在调试状态下，这两个标志通常被设成 50000062h（取决于 NtGlobalFlag）和 40000060h。下面提供的是 HEAP 结构，一会儿就会需要它。

```
+0x000 Entry                            : _HEAP_ENTRY
+0x008 Signature                        : Uint4B
+0x00c Flags                            : Uint4B
+0x010 ForceFlags                       : Uint4B
+0x014 VirtualMemoryThreshold           : Uint4B
+0x018 SegmentReserve                   : Uint4B
+0x01c SegmentCommit                    : Uint4B
+0x020 DeCommitFreeBlockThreshold       : Uint4B
+0x024 DeCommitTotalFreeThreshold       : Uint4B
+0x028 TotalFreeSize                    : Uint4B
+0x02c MaximumAllocationSize            : Uint4B
+0x030 ProcessHeapsListIndex            : Uint2B
+0x032 HeaderValidateLength             : Uint2B
+0x034 HeaderValidateCopy               : Ptr32 Void
+0x038 NextAvailableTagIndex            : Uint2B
+0x03a MaximumTagIndex                  : Uint2B
+0x03c TagEntries                       : Ptr32 _HEAP_TAG_ENTRY
+0x040 UCRSegments                      : Ptr32 _HEAP_UCR_SEGMENT
+0x044 UnusedUnCommittedRanges          : Ptr32 _HEAP_UNCOMMMTTED_RANGE
+0x048 AlignRound                       : Uint4B
+0x04c AlignMask                        : Uint4B
+0x050 VirtualAllocdBlocks              : _LIST_ENTRY
+0x058 Segments                         : [64] Ptr32 _HEAP_SEGMENT
+0x158 u                                : __unnamed
+0x168 u2                               : __unnamed
+0x16a AllocatorBackTraceIndex          : Uint2B
+0x16c NonDedicatedListLength           : Uint4B
+0x170 LargeBlocksIndex                 : Ptr32 Void
+0x174 PseudoTagEntries                 : Ptr32 _HEAP_PSEUDO_TAG_ENTRY
+0x178 FreeLists                        : [128] _LIST_ENTRY
+0x578 LockVariable                     : Ptr32 _HEAP_LOCK
+0x57c CommitRoutine                    : Ptr32     long
+0x580 FrontEndHeap                     : Ptr32 Void
+0x584 FrontHeapLockCount               : Uint2B
+0x586 FrontEndHeapType                 : UChar
+0x587 LastSegmentIndex                 : UChar
```

请再次回顾 PEB 结构，并且注意+18h 处的 ProcessHeap，又写出一个似乎很隐蔽的标记检测代码：

```
BOOL CheckHeapFlags(VOID)
{
    __asm {
        mov eax, fs:[0x30]
        mov eax, [eax+0x18]
        cmp dword ptr [eax+0x0C], 2
        jne __debugger_detected
        cmp dword ptr [eax+0x10], 0
        jne __debugger_detected
        xor eax, eax
      __debugger_detected:
    }
}
```

15.1.4 从源头消灭 BeingDebugged

首先系统创建进程的时候设置 BeingDebugged = TRUE，后来 NtGlobalFlag 根据这个标记设置

FLG_HEAP_VALIDATE_PARAMETERS 等标记。在为进程创建堆时，又由于 NtGlobalFlag 的作用，堆的 Flags 被设置了一些标记，这个 Flags 随即被填充到 ProcessHeap 的 Flags 和 ForceFlags 当中，同时堆的内存也因而被填充了很多 BAADF00D 之类的东西。这样调试器就会被检测到。

分析这个流程会发现，一切祸根只不过是很久以前系统设置了一个 BeingDebugged。犹如某地上空一只小小的蝴蝶扇动翅膀而扰动了空气，长时间后可能导致遥远的彼地发生一场暴风。因此要从源头制止这一切，在后面的事情没有发生之前改写这个值，那么所有的历史都会改写了。

系统确实给了这个时机，在编写调试器（或调试器插件）时，创建进程并调用 WaitForDebugEvent 后，在第一次 LOAD_DLL_DEBUG_EVENT 发生的时候置 BeingDebugged = FALSE 就可以了。但是会发现这样没法中断在系统断点了，所以在第二次 LOAD_DLL_DEBUG_EVENT 的时候要将 BeingDebugged 置为 TRUE，之后会停在系统断点，此时可以安全地清除 BeingDebugged 了。

笔者画了一个小表格来总结这一段代码，代码如下：

DebugEventCode	Count	PEB.BeingDebugged	Note
LOAD_DLL_DEBUG_EVENT	0	FALSE	
LOAD_DLL_DEBUG_EVENT	1	TRUE	
EXCEPTION_DEBUG_EVENT	0	FALSE	EXCEPTION_BREAKPOINT

至此，一次性地解决了 BeingDebugged、NtGlobalFlag、HeapFlags、HeapForceFlags 和 HeapMagic，这种感觉真好。

15.2 回归 Native：用户态的梦魇

在 Windows 这个隐藏了许多秘密的系统里，该相信调用的函数吗？调用的函数还是原来的那个吗？或者确定到底调用了谁？

15.2.1 CheckRemoteDebuggerPresent

打开 MSDN，除了 IsDebuggerPresent 之外，还有一个检测调试器的函数（本节不作特别说明的，都是指用 DebugAPI 实现的用户态调试器）。

```
BOOL CheckRemoteDebuggerPresent(
  HANDLE hProcess,
  PBOOL pbDebuggerPresent
);
```

看上去用法很简单，现在写一个测试程序看看效果。由于笔者的 SDK 版本太旧，这里用 GetProcAddress 获取函数地址后再调用。代码如下：

```
typedef BOOL (WINAPI *CHECK_REMOTE_DEBUGGER_PRESENT)(HANDLE, PBOOL);
BOOL CheckDebugger(VOID)
{
    HANDLE      hProcess;
    HINSTANCE   hModule;
    BOOL        bDebuggerPresent = FALSE;
    CHECK_REMOTE_DEBUGGER_PRESENT CheckRemoteDebuggerPresent;
    hModule = GetModuleHandleA("Kernel32");
    CheckRemoteDebuggerPresent = (CHECK_REMOTE_DEBUGGER_PRESENT) GetProcAddress(
                                 hModule,"CheckRemoteDebuggerPresent");
    hProcess = GetCurrentProcess();
    return CheckRemoteDebuggerPresent(
```

```
            hProcess,
            &bDebuggerPresent) ? bDebuggerPresent : FALSE;
}
```

分别在调试器下和正常情况下测试，判断结果都很准确，令人满意。读者可能怀疑它是否也是读取BeingDebugged判断调试器？那就清除掉这个标记试试。

用OllyDbg加载程序，如果有IsDebuggerPresent插件，可以直接使用它来解决。也可以手工来解决这个问题，在OllyDbg的数据窗口按“Ctrl+G”键，输入：fs:[30] + 2，或在命令行里输入。若是对30和2这些偏移量仍感迷惑的话，请复习一下TIB和PEB结构，按回车键可以看到BeingDebugged的值，如图15.2所示。

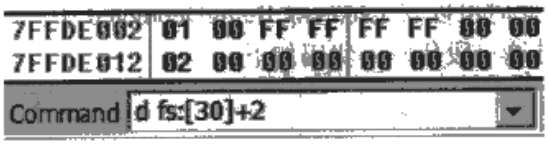


图15.2 查看BeingDebugged

7FFDE002h处的01就是BeingDebugged，它的值为TRUE，在01上单击后按“Ctrl+E”键，把它修改为00，结果就是IsDebuggerPresent这个函数检测不了调试器了。测试后会发现CheckRemoteDebuggerPresent给出的判断一如平常，这不禁令人好奇它内部的构造了。

15.2.2 ProcessDebugPort

在OllyDbg的CPU窗口按“Ctrl+G”键，输入CheckRemoteDebuggerPresent后按回车键，查看这个函数的汇编代码。代码如下：

```
7C859B1E  mov     edi, edi
7C859B20  push    ebp
7C859B21  mov     ebp, esp
7C859B23  cmp     dword ptr [ebp+8], 0
7C859B27  push    esi
7C859B28  je      short kernel32.7C859B5F
7C859B2A  mov     esi, dword ptr [ebp+C]
7C859B2D  test    esi, esi
7C859B2F  je      short kernel32.7C859B5F
7C859B31  push    0
7C859B33  push    4
7C859B35  lea     eax, dword ptr [ebp+8]
7C859B38  push    eax
7C859B39  push    7
7C859B3B  push    dword ptr [ebp+8]
7C859B3E  call    dword ptr [<&ntdll.ZwQueryInformationProcess >;
7C859B44  test    eax, eax
7C859B46  jge     short kernel32.7C859B50
7C859B48  push    eax
7C859B49  call    kernel32.7C80936B
7C859B4E  jmp     short kernel32.7C859B66
7C859B50  xor     eax, eax
7C859B52  cmp     dword ptr [ebp+8], eax
7C859B55  setne   al
7C859B58  mov     dword ptr [esi], eax
7C859B5A  xor     eax, eax
7C859B5C  nop
```

```
7C859B5D  jmp     short kernel32.7C859B68
7C859B5F  push    57
7C859B61  call    kernel32.7C8092B0
7C859B66  xor     eax, eax
7C859B68  pop     esi
7C859B69  pop     ebp
7C859B6A  retn    8
```

如果感觉汇编代码不太直观，把它翻译成 C 语言，是这个样子：

```
BOOL CheckRemoteDebuggerPresent(HANDLE hProcess,PBOOL pbDebuggerPresent)
{
    DWORD rv;
    if (hProcess & pbDebuggerPresent) {
        rv = NtQueryInformationProcess(hProcess, 7, &hProcess, 4, 0);
        if (rv < 0) {
            BaseSetLastNTError(rv); //事实上，这个函数名是用 IDA 打开 kernel32.dll 得到的
            return FALSE;
        } else {
            pbDebuggerPresent = hProcess;
            return TRUE;
        }
    } else {
        SetLastError(ERROR_INVALID_PARAMETER);
        return FALSE;
    }
}
```

抛开那些不需要过分关心的错误检查语句，会发现起作用的只有一行代码，就是对 NtQueryInformationProcess 的调用。NtQueryInformationProcess 不是 Win32 API，而是 Native API。至于 Native API 又是什么？很快就会了解到，但现在可以暂且认为它们只是一些普通的 API 函数，不要为此伤脑筋。

大多 Native API 是 Microsoft 尚未文档化的（Undocumented），但 Gary Nebbett 写了一本非常酷的参考手册《Windows NT 2000 Native API Reference》，一切可以从书中找到答案。先看看这个函数的原型：

```
ZwQueryInformationProcess(
  IN HANDLE ProcessHandle,
  IN PROCESSINFOCLASS ProcessInformationClass,
  OUT PVOID ProcessInformation,
  IN ULONG ProcessInformationLength,
  OUT PULONG ReturnLength OPTIONAL
);
```

读者大概会发现函数开头变成 Zw 了，而不是 Nt 了，事实上在用户态中它们是同一个函数的两个名字。ZwQueryInformationProcess 根据不同的 ProcessInformationClass 查询关于一个进程对象的信息，上面代码显示 CheckRemoteDebuggerPresent 查询了 7 号信息。可以从下面这个列表中找到它的意义：

```
                                        Query   Set
typedef enum _PROCESSINFOCLASS {
  ProcessBasicInformation,              // 0     Y       N
  ProcessQuotaLimits,                   // 1     Y       Y
  ProcessIoCounters,                    // 2     Y       N
  ProcessVmCounters,                    // 3     Y       N
  ProcessTimes,                         // 4     Y       N
  ProcessBasePriority,                  // 5     N       Y
  ProcessRaisePriority,                 // 6     N       Y
  ProcessDebugPort,                     // 7     Y       Y
```

```
    ProcessExceptionPort,              // 8    N      Y
    ProcessAccessToken,                // 9    N      Y
    ProcessLdtInformation,             // 10   Y      Y
    ProcessLdtSize,                    // 11   N      Y
    ProcessDefaultHardErrorMode,       // 12   Y      Y
    ProcessIoPortHandlers,             // 13   N      Y
    ProcessPooledUsageAndLimits,       // 14   Y      N
    ProcessWorkingSetWatch,            // 15   Y      Y
    ProcessUserModeIOPL,               // 16   N      Y
    ProcessEnableAlignmentFaultFixup,  // 17   N      Y
    ProcessPriorityClass,              // 18   N      Y
    ProcessWx86Information,            // 19   Y      N
    ProcessHandleCount,                // 20   Y      N
    ProcessAffinityMask,               // 21   N      Y
    ProcessPriorityBoost,              // 22   Y      Y
    ProcessDeviceMap,                  // 23   Y      Y
    ProcessSessionInformation,         // 24   Y      Y
    ProcessForegroundInformation,      // 25   N      Y
    ProcessWow64Information            // 26   Y      N
} PROCESSINFOCLASS;
```

数字 7 代表 ProcessDebugPort，具体含义为：

```
ProcessDebugPort
HANDLE DebugPort; // Information Class 7
When querying this information class, the value is interpreted as a Boolean
Indicating whether a debug port has been set or not.The debug port can be set only
if it was previously zero (in Windows NT 4.0, once set the port can also be reset
to zero).The handle which is set must be a handle to a port object. (Zero is also
allowed in Windows NT 4.0.)
```

终于了解到，CheckRemoteDebuggerPresent 实际上调用了 NtQueryInformationProcess，查询了某个进程的 ProcessDebugPort，这个值是系统用来与调试器通信的端口句柄。NtCurrentPeb()->BeingDebugged 可以被随意地清除掉而不影响调试，但若将调试端口设置为 0，系统就不会向用户态调试器发送调试事件通知，调试器当然就无法正常工作了。

如果注意到 ProcessInformationClass 列表中，ProcessDebugPort 既支持 Query 也支持 Set 操作，自然会联想到通过与 NtQueryInformationProcess 相应的 NtSetInformationProcess 函数将 DebugPort 设为 0，导致调试器无法与被调试进程通信来使之失效，这是个好想法。不过请注意看上面的一段关于 ProcessDebugPort 的说明，“The debug port can be set only if it was previously zero”，由于程序被调试时 DebugPort 已经被系统设为非零值，之后就无法再设置了，因此这个想法不能转化为现实了。

别灰心，我们已经想到破坏调试器与被调试程序之间的通信，只是具体实施上遇到了一些困难。

15.2.3 ThreadHideFromDebugger

既然 NtSetInformationProcess 这条路走不通，就换个思路。翻开《Windows NT 2000 Native API Reference》，搜索 Debugger 一词，会发现一些新的线索，如 ZwSetInformationThread：

```
NTSTATUS
ZwSetInformationThread(
    IN HANDLE  ThreadHandle,
    IN THREADINFOCLASS  ThreadInformationClass,
    IN PVOID  ThreadInformation,
    IN ULONG  ThreadInformationLength
    );
```

这个函数可以设置一个线程相关的信息，看看 ThreadInformationClass 列表能发现什么？

```
                                                    Query    Set
typedef enum _THREADINFOCLASS {
    ThreadBasicInformation,                 // 0     Y        N
    ThreadTimes,                            // 1     Y        N
    ThreadPriority,                         // 2     N        Y
    ThreadBasePriority,                     // 3     N        Y
    ThreadAffinityMask,                     // 4     N        Y
    ThreadImpersonationToken,               // 5     N        Y
    ThreadDescriptorTableEntry,             // 6     Y        N
    ThreadEnableAlignmentFaultFixup,        // 7     N        Y
    ThreadEventPair,                        // 8     N        Y
    ThreadQuerySetWin32StartAddress,        // 9     Y        Y
    ThreadZeroTlsCell,                      // 10    N        Y
    ThreadPerformanceCount,                 // 11    Y        N
    ThreadAmILastThread,                    // 12    Y        N
    ThreadIdealProcessor,                   // 13    N        Y
    ThreadPriorityBoost,                    // 14    Y        Y
    ThreadSetTlsArrayAddress,               // 15    N        Y
    ThreadIsIoPending,                      // 16    Y        N
    ThreadHideFromDebugger                  // 17    N        Y
} THREADINFOCLASS;
```

最后一行的 ThreadHideFromDebugger 非常显眼：

```
    This information class can only be set. It disables the generation of debug events for the
thread.This information class requires no data, and so ThreadInformation may be a null
pointer .ThreadInformationLength should be zero.
```

为线程设置 ThreadHideFromDebugger 可以禁止某一个线程产生调试事件，这里有一个小程序，调试它看看到底产生了怎样的效果。

```
typedef DWORD (WINAPI *ZW_SET_INFORMATION_THREAD)(HANDLE, DWORD, PVOID, ULONG);
#define ThreadHideFromDebugger 17
VOID DisableDebugEvent(VOID)
{
HINSTANCE hModule;
ZW_SET_INFORMATION_THREAD ZwSetInformationThread;

hModule = GetModuleHandleA("Ntdll");
ZwSetInformationThread = (ZW_SET_INFORMATION_THREAD)GetProcAddress
                         (hModule, "ZwSetInformation Thread");
ZwSetInformationThread(GetCurrentThread(),ThreadHideFromDebugger,0,0);
}
```

编译好之后用 OllyDbg 打开运行，会发现 OllyDbg 中什么也看不到了，如图 15.3 所示。

这是由于程序已经退出了，调试器打开的进程句柄不再有效。OllyDbg 没有收到退出通知，或者说系统根本就不通知 OllyDbg，它仍在试图反汇编进程的内存数据，可惜都是徒劳。

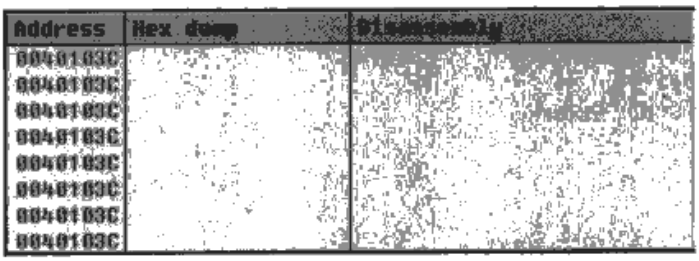


图 15.3　OllyDbg 打开后的结果

这个 ThreadHideFromDebugger 是不是把调试端口设置为 0 了？在 Windows 2000 的代码中可以找到 ZwSetInformationThread 是如何对 ThreadHideFromDebugger 进行处理的。代码如下：

```
case ThreadHideFromDebugger:
      if ( ThreadInformationLength != 0 ) {
         return STATUS_INFO_LENGTH_MISMATCH;
      }

      st = ObReferenceObjectByHandle(
            ThreadHandle,
            THREAD_SET_INFORMATION,
            PsThreadType,
            PreviousMode,
            (PVOID *)&Thread,
            NULL
            );

      if ( !NT_SUCCESS(st) ) {
         return st;
      }
      Thread->HideFromDebugger = TRUE;
      ObDereferenceObject(Thread);
      return st;
      break;
```

可以看到，这里只是将 Thread 对象的 HideFromDebugger 成员设置为 TRUE 了，那么再搜索一下引用了 HideFromDebugger 的代码，有很多处，随便选一个看看。比如：

```
VOID
DbgkMapViewOfSection(
   IN HANDLE SectionHandle,
   IN PVOID BaseAddress,
   IN ULONG SectionOffset,
   IN ULONG_PTR ViewSize
   )
/*++
Routine Description:
   This function is called when the current process successfully
   maps a view of an image section. If the process has an associated
   debug port, then a load dll message is sent.
......
--*/

{
   PVOID Port;
   DBGKM_APIMSG m;
   PDBGKM_LOAD_DLL LoadDllArgs;
   PEPROCESS Process;
   PIMAGE_NT_HEADERS NtHeaders;

   PAGED_CODE();

   Process = PsGetCurrentProcess();

   Port = PsGetCurrentThread()->HideFromDebugger ? NULL : Process->DebugPort;
   if ( !Port || KeGetPreviousMode() == KernelMode ) {
```

```
        return;
    }

    LoadDllArgs = &m.u.LoadDll;
.....
    LoadDllArgs->DebugInfoSize = 0;
    try {
        NtHeaders = RtlImageNtHeader(BaseAddress);
        if ( NtHeaders ) {
            LoadDllArgs->DebugInfoFileOffset =
                                NtHeaders->FileHeader.PointerToSymbolTable;
            LoadDllArgs->DebugInfoSize = NtHeaders->FileHeader.NumberOfSymbols;
            }
        }
    except(EXCEPTION_EXECUTE_HANDLER) {
        LoadDllArgs->DebugInfoFileOffset = 0;
        LoadDllArgs->DebugInfoSize = 0;
    }
    DBGKM_FORMAT_API_MSG(m,DbgKmLoadDllApi,sizeof(*LoadDllArgs));
    DbgkpSendApiMessage(&m,Port,TRUE);
    ZwClose(LoadDllArgs->FileHandle);
}
```

注释里说，如果当前进程成功映射了一个映像，就会调用这个函数。如果进程跟一个调试端口关联起来，就会通知调试器发生了 LOAD_DLL_DEBUG_EVENT 事件。若线程的 HideFromDebugger 为 TRUE，代码在中间就已返回，调试器对发生的事情就一无所知了。

SoBeIt 在《Windows 异常处理流程》中也提到过这个 HideFromDebugger，部分内容如下：

> 用户模式异常处理流程：若 KiDebugRoutine 不为空，则不为空就将 Context、陷阱帧、异常记录、异常帧、发生异常的模式等压入栈并将控制交给 KiDebugRoutine。当处理完毕用 Context 设置陷阱帧并返回到上一级例程。（第一次机会）否则把异常记录压栈并调用 DbgkForwardException，在 DbgkForwardException 里判断当前线程 ETHREAD 结构的 HideFromDebugger 成员如果为 FALSE（为 TRUE 表示该异常对用户调试器不可见）则向当前进程的调试端口（DebugPort）发送 LPC 消息。

ThreadHideFromDebugger 与直接将 DebugPort 清零的方法异曲同工，是个效果不错的反调试技巧。一会儿将会看到，它的价值不只于此。

15.2.4　Debug Object

一直以来，都紧盯着被调试的进程不放，现在来看看调试器。如果想要了解一点 Windows 调试器的一些内幕，推荐 Alex Ionescu 的系列文章《Windows User Mode Debugging Internals》、《Windows Native Debugging Internals》以及《Kernel User-Mode Debugging Support (Dbgk)》，它们包含了 Windows 调试机制所有的谜底，在 OpenRCE.org 上可以在线阅读。

尽管用 OllyDbg 看汇编代码也不是很复杂，不过大部分代码都可以从 ReactOS 项目中找到相应的、模仿得非常逼真的 C 语言源代码，笔者打算直接引用它们，可以让眼睛休息一下，不必为大量的 push、call 指令烦恼。

调试器与被调试程序建立关系有两种途径：在创建进程时设置 DEBUG_PROCESS，或者调用 DebugActiveProcess 附加到某个已运行的进程上。建立新进程时有太多与调试无关的操作，因此以后者为入手点研究。代码如下：

```
BOOL WINAPI DebugActiveProcess(IN DWORD dwProcessId)
{
    NTSTATUS Status;
```

```
    HANDLE Handle;

    /* Connect to the debugger */
    Status = DbgUiConnectToDbg();
    if (!NT_SUCCESS(Status))
    {
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Get the process handle */
    Handle = ProcessIdToHandle(dwProcessId);
    if (!Handle) return FALSE;

    /* Now debug the process */
    Status = DbgUiDebugActiveProcess(Handle);
    NtClose(Handle);

    /* Check if debugging worked */
    if (!NT_SUCCESS(Status))
    {
        /* Fail */
        SetLastErrorByStatus(Status);
        return FALSE;
    }

    /* Success */
    return TRUE;
}
```

Windows 的很多函数都是价值不高的包装函数（Wrapper），一层层地向下调用，所以只能一层层地向下看。DbgUiConnectToDbg 函数：

```
NTSTATUS NTAPI DbgUiConnectToDbg(VOID)
{
    OBJECT_ATTRIBUTES ObjectAttributes;

    /* Don't connect twice */
    if (NtCurrentTeb()->DbgSsReserved[1]) return STATUS_SUCCESS;

    /* Setup the Attributes */
    InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, 0);

    /* Create the object */
    return ZwCreateDebugObject(&NtCurrentTeb()->DbgSsReserved[1],
                               DEBUG_OBJECT_ALL_ACCESS,
                               &ObjectAttributes,
                               TRUE);
}
```

显然调试器创建了一个 DebugObject，并且存储在了 NtCurrentTeb()->DbgSs Reserved[1]。这个域虽然名字古怪，事实上它就是调试器用来保存 DebugObject 句柄的地方，普通的进程中 DbgSsReserved[1]应该为 NULL，反过来若不为 NULL，则它是一个用户态的调试器的进程。

Ntdll 中有导出函数可以操作这个域：

```
HANDLE NTAPI DbgUiGetThreadDebugObject(VOID)
```

```
{
    /* Just return the handle from the TEB */
    return NtCurrentTeb()->DbgSsReserved[1];
}

VOID
NTAPI
DbgUiSetThreadDebugObject(HANDLE DebugObject)
{
    /* Just set the handle in the TEB */
    NtCurrentTeb()->DbgSsReserved[1] = DebugObject;
}
```

之所以把这两个函数拿出来，是因为如果想针对 DebugObject 来判断某个进程是不是一个调试器，对 TEB 中的 DbgSsReserved[1]偏移量硬编码也许不是一个特别好的方案，但可以从 DbgUiGetThread DebugObject 函数体中得到准确的偏移量：

```
7C9706DE    mov eax, dword ptr fs:[18]   ; DbgUiGetThreadDebugObject
7C9706E4    mov  eax, dword ptr [eax+F24]; Get F24
7C9706EA    retn
```

再回过头来看 DebugObject 是如何被创建的：

```
NTSTATUS NTAPI NtCreateDebugObject(OUT PHANDLE DebugHandle,
                IN ACCESS_MASK DesiredAccess,
                IN POBJECT_ATTRIBUTES ObjectAttributes,
                IN BOOLEAN KillProcessOnExit)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    PDEBUG_OBJECT DebugObject;
    HANDLE hDebug;
    NTSTATUS Status = STATUS_SUCCESS;
    PAGED_CODE();

    /* Check if we were called from user mode*/
    if (PreviousMode != KernelMode)
    {
        /* Enter SEH for probing */
        _SEH_TRY
        {
            /* Probe the handle */
            ProbeForWriteHandle(DebugHandle);
        }
        _SEH_HANDLE
        {
            /* Get exception error */
            Status = _SEH_GetExceptionCode();
        } _SEH_END;
        if (!NT_SUCCESS(Status)) return Status;
    }

    /* Create the Object */
    Status = ObCreateObject(PreviousMode,
                            DbgkDebugObjectType,
                            ObjectAttributes,
                            PreviousMode,
```

```
                        NULL,
                        sizeof(DEBUG_OBJECT),
                        0,
                        0,
                        (PVOID*)&DebugObject);
.....
```

ZwCreateDebugObject 事实上调用了 ObCreateObject 创建对象，因此可以用 ZwQueryObject 查询所有对象的类型，若发现名为“DebugObject”的数目不为 0，那么系统中就存在有调试器。

```
ZwQueryObject(
  IN HANDLE ObjectHandle,
  IN OBJECT_INFORMATION_CLASS ObjectInformationClass,
  OUT PVOID ObjectInformation,
  IN ULONG ObjectInformationLength,
  OUT PULONG ReturnLength OPTIONAL
);
```

ObjectInformationClass 取值见下面列表：

```
                                                Query     Set
typedef enum _OBJECT_INFORMATION_CLASS {
  ObjectBasicInformation,                  // 0    Y         N
  ObjectNameInformation,                   // 1    Y         N
  ObjectTypeInformation,                   // 2    Y         N
  ObjectAllTypesInformation,               // 3    Y         N
  ObjectHandleInformation                  // 4    Y         Y
} OBJECT_INFORMATION_CLASS;
```

设置 ObjectAllTypesInformation 就可以获得全部对象类型的信息，结构如下：

```
typedef struct _OBJECT_ALL_TYPES_INFORMATION {    // Information Class 3
  ULONG NumberOfTypes;
  OBJECT_TYPE_INFORMATION TypeInformation;
} OBJECT_ALL_TYPES_INFORMATION, *POBJECT_ALL_TYPES_INFORMATION;
```

OBJECT_TYPE_INFORMATION：

```
typedef struct _OBJECT_TYPE_INFORMATION { // Information Class 2
  UNICODE_STRING Name;
  ULONG ObjectCount;
  ULONG HandleCount;
  ULONG Reserved1[4];
  ULONG PeakObjectCount;
  ULONG PeakHandleCount;
  ULONG Reserved2[4];
  ULONG InvalidAttributes;
  GENERIC_MAPPING GenericMapping;
  ULONG ValidAccess;
  UCHAR Unknown;
  BOOLEAN MaintainHandleDatabase;
  POOL_TYPE PoolType;
  ULONG PagedPoolUsage;
  ULONG NonPagedPoolUsage;
} OBJECT_TYPE_INFORMATION, *POBJECT_TYPE_INFORMATION;
```

注意 ObjectAllTypesInformation 的 Remark“The ObjectHandle parameter need not contain a valid handle to query this information class”。因此需要将 ObjectHandle 设为 NULL，Exetools 上 Peter[Pan]写了这一检测的

完整实现，详细代码见光盘映像文件。恢复那些被注释掉的行，在有无调试器的环境下运行，可以清楚地看出对象类型结构在不同情况下的区别。

无调试器时：

```
TypeName: DebugObject
DefaultNonPagedPoolCharge: 30
DefaultPagedPoolCharge: 0
GenericMapping: 20001
HighWaterNumberOfHandles: 5
HighWaterNumberOfObjects: 6
InvalidAttributes: 0
MaintainHandleCount: 0
PoolType: 0
SecurityRequired: 1
TotalNumberOfHandles: 0
TotalNumberOfObjects: 0
```

有调试器时：

```
TypeName: DebugObject
DefaultNonPagedPoolCharge: 30
DefaultPagedPoolCharge: 0
GenericMapping: 20001
HighWaterNumberOfHandles: 5
HighWaterNumberOfObjects: 6
InvalidAttributes: 0
MaintainHandleCount: 0
PoolType: 0
SecurityRequired: 1
TotalNumberOfHandles: 1
TotalNumberOfObjects: 1
```

不过如此检测只能说明系统中存在一个调试器，却不能确定这个调试器正在调试当前的程序。如果想给 Cracker 一点惩戒，却无心伤害了普通用户就不合适了。这有点“宁可错杀一千，不可放过一人”的意味，过于苛刻。

不过，在实际应用中，正常用户运行一个 3D 游戏还开着调试器确实不太合理。检测到 DebugObject 就去重新启动系统的做法自然不可取，给予提示要求用户关闭调试器的方式亦容易暴露出弱点，默默地退出是比较好的做法。

15.2.5 SystemKernelDebuggerInformation

前面在《Windows NT 2000 Native API Reference》中搜索关于 debugger 的篇章，事实上忽略了一个函数 ZwQuerySystemInformation ：

```
ZwQuerySystemInformation(
  IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
  IN OUT PVOID SystemInformation,
  IN ULONG SystemInformationLength,
  OUT PULONG ReturnLength OPTIONAL
);
```

当 SystemInformation = SystemKernelDebuggerInformation 的时候可以判断是否有系统调试器存在：

```
typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION { // Information Class 35
  BOOLEAN DebuggerEnabled;
```

```
  BOOLEAN DebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;
Members
DebuggerEnabled :A boolean indicating whether kernel debugging has been enabled or not.
DebuggerNotPresent:A boolean indicating whether contact with a remote debugger has been
established
or not.
```

依循惯例，下面还是一个实例：

```
#include <windows.h>
#include <stdio.h>
#define SystemKernelDebuggerInformation 35
#pragma pack(4)

typedef struct _SYSTEM_KERNEL_DEBUGGER_INFORMATION
{
    BOOLEAN DebuggerEnabled;
    BOOLEAN DebuggerNotPresent;
} SYSTEM_KERNEL_DEBUGGER_INFORMATION, *PSYSTEM_KERNEL_DEBUGGER_INFORMATION;

typedef DWORD(WINAPI *ZW_QUERY_SYSTEM_INFORMATION)(DWORD,PVOID,ULONG,PULONG);

BOOL
CheckKernelDbgr(VOID)
{
    HINSTANCE hModule = GetModuleHandleA("Ntdll");
    ZW_QUERY_SYSTEM_INFORMATION ZwQuerySystemInformation =
                (ZW_QUERY_SYSTEM_INFORMATION)GetProcAddress(hModule,
                "ZwQuerySystemInformation");
    SYSTEM_KERNEL_DEBUGGER_INFORMATION Info = {0};

    ZwQuerySystemInformation(
        SystemKernelDebuggerInformation,
        &Info,
        sizeof(Info),
        NULL);

    return (Info.DebuggerEnabled && !Info.DebuggerNotPresent);
}
```

跟 ZwQueryProcessInformation 没有多大区别，不过请看 SystemKernelDebugger Information 的说明，其中提到这个功能检测的是“kernel debugger”，直译就是系统调试器，SoftICE 是系统调试器，但这里的“kernel debugger”却不是 SoftICE 这样的调试器。这里有必要说一些调试器之间的差异了。

硬件调试器不在本文的讨论范畴内，在软件实现的调试器中，最明显的分界线莫过于用户级调试器和系统调试器了。前面已经提到过用户级调试器，例如 VC 和 OllyDbg 是使用 DebugAPI 来开发的，自身也只是一个 Ring 3 级应用程序，故而能做的事情有限，只能调试应用程序，无法中断内核，自然也就无法调试驱动程序。

而系统调试器则拥有更大的权利，比较流行的系统调试器就是 SoftICE、Syser Debugger 等，这一类调试器实现方法比较底层，调试起来速度也比较快。

比较奇特的就是 WinDbg 了，WinDbg 可以用于 Kernel 模式调试，也可以进行用户模式调试，还可以调试 Dump 文件。推荐 SoBeIt 的《Windows 内核调试器原理浅析》，对 WinDbg 和 SoftICE 的实现原理有比较详细的分析，文中提到 WinDbg 双机调试时会以 Debug 方式启动系统。摘自部分内容：“内核调试器

在一台机器上启动，通过串口调试另一个相联系的以 Debug 方式启动的系统，这个系统可以是虚拟机上的系统，也可以是另一台机器上的系统(这只是微软推荐和实现的方法，其实像 SoftICE 这类内核调试器可以实现单机调试)。很多人认为主要功能都是在 WinDbg 里实现的，事实上并不是那么一回事，Windows 已经把内核调试的机制集成进了内核，WinDbg、kd 之类的内核调试器要做的仅仅是通过串行发送特定格式数据包来进行联系，比如中断系统、下断点、显示内存数据等。然后把收到的数据包经过 WinDbg 处理显示出来。”

事实上，也只有以 Debug 方式启动系统时，系统中才会留下特殊的标记，上面的例子代码才能检测到“kernel debugger”的存在，如果只用 WinDbg 的 lkd 来观察内核，是无法发现的。至于 SoftICE，虽然它也是系统级调试器，却不需要太多操作系统的支持，因此 SystemKernelDebuggerInformation 是无法探测到它的。

15.2.6 Native API

了解 Native API 能对 Windows 的内部机制有个很好的认识。值得注意的是，这些内容都是针对 NT 内核系统的，对 Windows 9x 来说不适用了。

1. 认识 Native API

什么是 Native API？《Слежение за вызовом функций Native API》(《跟踪 Native API 函数调用》)已经解释得非常清楚，笔者还是要重提一下，先看图 15.4。

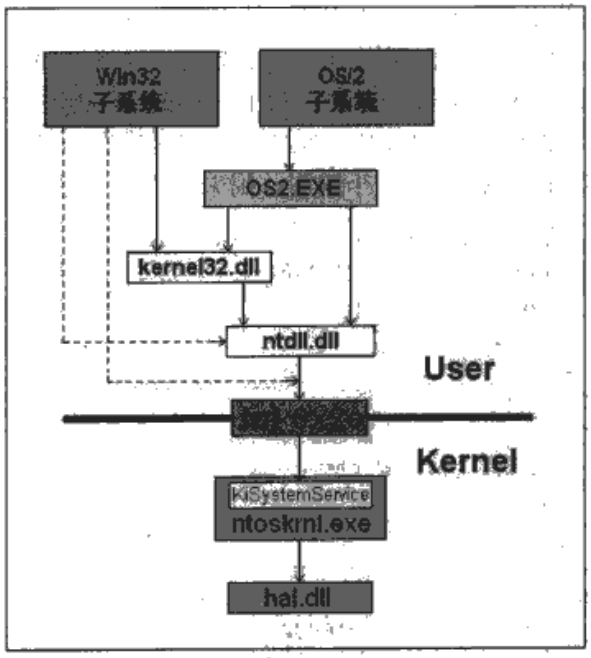


图 15.4 API 的调用过程

Windows NT 不但能运行 Win32 程序，旧的 Win16、MSDOS、OS/2 等系统下的应用程序也有相应的子系统提供支持。各个子系统转入 ntdll 或者直接转入内核中的 ntoskrnl，最终到硬件抽象层 hal 来实现。这样，设计子系统时，只需要实现对 ntdll 提供的接口的封装即可。各个子系统最终会转入 ntdll.dll 或直接调用 0x2e 中断进入 ntoskrnl.exe 内核态。

作为一个例子，来看一下常见的 kernel32!CreateFileA 函数最终是如何转入系统层的。做了一些转换工作后，又转向了 CreateFileW。

```
7C801A24   8BFF            mov     edi, edi
```

```
7C801A26    55              push    ebp
...
7C801A4A    E8 11ED0000     call    kernel32.CreateFileW
```

CreateFileW 又转发到 ntdll!NtCreateFile。

```
7C810760 >  8BFF            mov     edi, edi
7C810762    55              push    ebp
...
7C81090F    50              push    eax
7C810910    FF15 0810807C   call    dword ptr [<&ntdll.NtCreateFile>]
```

ntdll!NtCreateFile 如下：

```
7C92D682 >  B8 25000000     mov     eax, 25
7C92D687    BA 0003FE7F     mov     edx, 7FFE0300
7C92D68C    FF12            call    dword ptr [edx]
7C92D68E    C2 2C00         retn    2C
```

在 Windows 2000 以后的系统里，ntdll!NtCreateFile 是如下样式：

```
mov eax, xxx
lea edx, [esp+4]
int 2Eh
ret xxx
```

在 Windows XP 之后，系统用 Sysenter 进入内核，不会主动使用 int 2e 作为调用内核函数的方法，不过 int 2e 仍然被保留下来，还可以使用它们调用 Native 函数而无须通过 ntdll。

不只是 ZwCreateFile，ntdll!NtXxx 大部分函数（除了 NtCurrentTeb）都是由这样一个被称为 Stub 的函数转向系统层中 ntoskrnl 中真正的函数的。ZwXxx/NtXxx 这些函数在 ntdll 中看起来只有第一行指令“mov eax,XXX”的 XXX 不相同，这里的 XXX 是一个索引，用来在内核中定位真正的函数。例如：

```
7C92D586 >  B8 19000000  mov     eax, 19            ; ZwClose
7C92D58B    BA 0003FE7F  mov     edx, 7FFE0300
7C92D590    FF12         call    dword ptr [edx]
7C92D592    C2 0400      retn    4
7C92D595    90           nop
7C92D59B >  B8 1A000000  mov     eax, 1A            ; ZwCloseObjectAuditAlarm
7C92D5A0    BA 0003FE7F  mov     edx, 7FFE0300
7C92D5A5    FF12         call    dword ptr [edx]
7C92D5A7    C2 0C00      retn    0C
7C92D5AA    90           nop
7C92D5AB    90           nop
7C92D5B0 >  B8 1B000000  mov     eax, 1B            ; ZwCompactKeys
7C92D5B5    BA 0003FE7F  mov     edx, 7FFE0300
7C92D5BA    FF12         call    dword ptr [edx]
7C92D5BC    C2 0800      retn    8
```

稍后研究 Stub 与索引的问题。继续分析 NtCreateFile，它将 eax 设置为 25h 后又调用了 KiFastSystemCall。代码如下：：

```
7C92EB8B    8BD4         mov     edx, esp
7C92EB8D    0F34         sysenter
7C92EB8F    90           nop
7C92EB90    90           nop
7C92EB91    90           nop
7C92EB92    90           nop
```

```
7C92EB93    90              nop
7C92EB94    C3              retn
```

这条 Sysenter 指令 OllyDbg 跟踪不下去了，CPU 执行这条指令之后就会进入内核态，而 OllyDbg 只是一个用户级调试器。这条指令是 Windows XP 之后的系统才使用的，关于 Sysenter 和对应的 Sysexit 以及将要提到的 rdmsr 和 wrmsr 指令，请看《P4_IA32 Intel Architecture Software Developer's Manual》以及 wowocock 写的《SYSENTER 简介及相关例子》。简单地说，Sysenter 指令会设置一系列环境转入 MSR 的 SYSENTER_EIP_MSR 寄存器中的地址执行，这个寄存器号为 176，可以用 WinDbg 输入“rdmsr 176”得到，更详细的资料请参考《XP 下 sysenter hook-RDMSR-WRMSR》。在 WinDbg 中输入命令：

```
lkd> rdmsr 176
msr[176] = 00000000`80541770
```

可见转入了地址 80541770，用 ln 命令可以看出这个地址对应什么函数：

```
lkd> ln 80541770
(80541770)   nt!KiFastCallEntry   |  (8054187e)   nt!KiServiceExit
Exact matches:
   nt!KiFastCallEntry = <no type information>
```

这样那些 Stub 函数最终进入了 KiFastCallEntry。刚才说到 Windows XP 下仍然可以使用 int 2e，现在就来看看为什么。中断 2e 的处理函数（ISR）是 KiSystemService，用 lkd 命令“!idt -a 2e”可以看到：

```
lkd> !idt -a 2e
Dumping IDT:
2e: 805416a1 nt!KiSystemService
```

也就是说，int 2e 指令会执行 KiSystemService，这个函数做了一些准备工作又会转到 KiFastCallEntry 中，所以 Sysenter 和 int 2e 是异曲同工的。

```
nt!KiSystemService:
805416a1 6a00            push    0
805416a3 55              push    ebp
805416a4 53              push    ebx
805416a5 56              push    esi
805416a6 57              push    edi
805416a7 0fa0            push    fs
805416a9 bb30000000      mov     ebx,30h
805416ae 668ee3          mov     fs,bx
80541708 f6462cff        test    byte ptr [esi+2Ch],0FFh
8054170c 0f858afefffe    jne     nt!Dr_kss_a (8054159c)
80541712 fb              sti
80541713 e9e7000000      jmp     nt!KiFastCallEntry+0x8f (805417ff)
```

至于 KiFastCallEntry，这个函数比较长，详细一些的分析可参考《Undocumented Windows 2000 Secrets》，ReactOS 中也有仿真代码。这个函数主要做了一些参数检测的工作，最后使用 Stub 函数中设置的 eax 作为索引，查找系统中的一个表，表项对应的地址就是内核函数真正的地址，找到之后就可以设置参数调用了。

2. SDT

系统服务表（System Service Table，SST）是被存储在服务描述表（Service Descriptor Table，SDT）中的一个表项，它的结构官方没有公开，不过《Undocument Windows 2000 Secretes》出版了 C 语言定义：

```
typedef struct _SYSTEM_SERVICE_TABLE
{
    PNTPROC  ServiceTable;    // array of entry points to the calls
    PDWORD   CounterTable;    // array of usage counters
```

```
    DWORD   ServiceLimit;   // number of table entries
    PBYTE   ArgumentTable;  // array of arguments
}
    SYSTEM_SERVICE_TABLE,
    *PSYSTEM_SERVICE_TABLE,
    **PP_SYSTEM_SERVICE_TABLE;
```

一些 AntiRootkit 软件中也称 SDT 为系统服务描述表（System Service Descriptor Table，SSDT），以后本书不区分 SSDT 与 SDT。SDT 结构的定义是这样的：

```
typedef struct _SERVICE_DESCRIPTOR_TABLE
{
SYSTEM_SERVICE_TABLE ntoskrnl;    // ntoskrnl.exe ( native api )
SYSTEM_SERVICE_TABLE win32k;      // win32k.sys (gdi/user support)
SYSTEM_SERVICE_TABLE Table3;      // not used
SYSTEM_SERVICE_TABLE Table4;      // not used
}
SYSTEM_DESCRIPTOR_TABLE,
*PSYSTEM_DESCRIPTOR_TABLE,
**PPSYSTEM_DESCRIPTOR_TABLE;
```

一个 SDT 可以包含 4 个 SST，但不是所有的 SST 都被使用到。ntoskrnl 导出了一个名为“KeServiceDescriptorTable”的 SDT，它里面只有一个 SST 对应着 ntoskrnl 中的函数。用 lkd 可以看到：

```
lkd> dd KeServiceDescriptorTable
8055c6e0  80504940 00000000 0000011c 80504db4  //ntoskrnl.exe
8055c6f0  00000000 00000000 00000000 00000000
8055c700  00000000 00000000 00000000 00000000  //其他三个 SST 未使用
8055c710  00000000 00000000 00000000 00000000
```

不过系统还维护着另一个未导出的 SDT，叫做“KeServiceDescriptorTableShadow”，这个 SDT 会指派给 GUI 线程使用。因为 user32.dll 和 gdi32.dll 一些函数也会通过类似于 ntdll 中的那些 Stub 函数来调用内核中的代码。

```
lkd> dd KeServiceDescriptorTableShadow
8055c6a0  80504940 00000000 0000011c 80504db4  //ntoskrnl.exe
8055c6b0  bf999280 00000000 0000029b bf999f90  //win32k.sys
8055c6c0  00000000 00000000 00000000 00000000  //其余两个 SST 未使用
8055c6d0  00000000 00000000 00000000 00000000
```

SST 的 ServiceTable 就是函数的地址表了，与 ntoskrnl.exe 对应的 SST 中 ServiceTable 还可以通过导出的地址 KiServiceTable 来访问：

```
lkd> dd KiServiceTable
//注意这里的 80504940 就是上面 SDT 中的 ServiceTable
80504940  805a4104 805f037e 805f3bcc 805f03b0
80504950  805f3c06 805f03e6 805f3c4a 805f3c8e
80504960  80614b98 806158da 805eb72e 805eb386
80504970  805d43c2 805d4372 806151be 805b59ea
```

前面跟踪 ntdll!ZwCreateFile 设置的索引是 25，通过这个表，就可以知道它的庐山真面目了。

```
lkd> ln poi(KiServiceTable+25*4)
(80578ed2)   nt!NtCreateFile   |  (80578f0c)   nt!NtCreateNamedPipeFile
Exact matches:
    nt!NtCreateFile = <no type information>
```

正是 nt!NtCreateFile，地址是 80578ed2h，这个地址大于 7fffffffh 位于内核，是货真价实的函数，向下就会调用 IO 函数了。

```
80578ed2 8bff        mov     edi,edi
80578ed4 55          push    ebp
80578ed5 8bec        mov     ebp,esp
80578ed7 33c0        xor     eax,eax
80578ed9 50          push    eax
80578eda 50          push    eax
80578edb 50          push    eax
...
80578efa ff7508      push    dword ptr [ebp+8]
80578efd e8a8d8ffff  call    nt!IoCreateFile (805767aa)
80578f02 5d          pop     ebp
80578f03 c22c00      ret     2Ch
```

在 ntoskrnl 中，也有 ZwXxx 和 NtXxx 两个版本的函数，不过不像 ntdll 中是同一个函数的不同名字。

3. Zw 和 Nt

Zw 和 Nt 开头的函数有什么区别？OSR Online 上的《Nt vs. Zw - Clearing Confusion On The Native API》将这些问题讲得很清楚，读者可以花点时间去注册一个 Member。

ntdll.dll 中 Zw 和 Nt 两种开头的名字，以 ZwWriteFile 和 NtWriteFile 为例，用 LordPE 查看这两个函数的 RVA，会发现这两个名字事实上指向同一段 Stub 函数，这个 Stub 函数再通过 Sysenter 调用 ntoskrnl.exe 中的函数。得出的结论就是 ntdll 中 ZwXxx 与 NtXxx 是同一个 Stub 函数的不同名字。

相应的，用 WinDbg 的 lkd 观察 ntoskrnl 中的函数，看看有什么不同：

```
lkd> u nt!ZwWriteFile
nt!ZwWriteFile:
804dec24 b8i2010000  mov     eax,112h
804dec29 8d542404    lea     edx,[esp+4]
804dec2d 9c          pushfd
804dec2e 6a08        push    8
804dec30 e8fc090000  call    nt!KiSystemService (804df631)
804dec35 c22400      ret     24h

lkd> u nt!NtWriteFile l50
nt!NtWriteFile:
80578145 6a74        push    74h
80578147 68d82c4f80  push    offset nt!GUID_DOCK_INTERFACE+0x3bc (804f2cd8)
8057814c e8eab2f6ff  call    nt!_SEH_prolog (804e343b)
......
805781ae a134005680  mov     eax,dword ptr [nt!MmUserProbeAddress (80560034)]
805781b3 3bc8        cmp     ecx,eax
805781b5 0f83e8ee0600    jae     nt!NtWriteFile+0x72 (805e70a3)
805781bb 8b01        mov     eax,dword ptr [ecx]
805781bd 8901        mov     dword ptr [ecx],eax
```

可以看到，nt!ZwWriteFile 也是一个 Stub 函数，而 nt!NtWriteFile 却不像 ntdll!NtWriteFile，它是真正的内核函数。内核中的 NtXxx 是各个 Stub 函数最终转向的、实现具体功能的函数。如果调用来自用户态，即 PreviousMode 为 UserMode，就会进行一系列参数检测；反之如果调用来自内核态，即 PreviousMode 为 KernelMode，则不会检测参数。

再回顾一下：

（1）ntdll.dll 中 ZwXxx 与 NtXxx 函数都是导向 ntoskrnl.exe 的 Stub 函数。

（2）ntoskrnl.exe 中 ZwXxx 函数也是 Stub 函数，导向 ntoskrnl.exe 的 NtXxx 函数。

（3）ntoskrnl.exe 中 NtXxx 函数是所有 Stub 的目的地，是真正做事情的函数。

15.2.7 Hook 和 AntiHook

看似强大的保护通常都有致命的弱点，一些著名的游戏反外挂程序，只要找到一个小小的开关，所有保护特性都会被关闭。

1. Hook

本节所使用的反调试技巧都是使用 Native API 进行的，如果调用的 Native API 被人动了手脚，那么一切检测都像是别人手里的提线木偶。例如，OllyDbg 调试器插件 HideOD 可以自动挂钩这些 Native API，于是教科书式的反调试都无效了。

下面来看一下 HideOD 是如何让相应的 Native API 不能正常工作的。加载目标程序后，在 OllyDBG 的 CPU 窗口查看 ZwSetInformationThread 函数汇编代码。正常的代码如下：

```
7C92E642   B8 E5000000   mov    eax, 0E5
7C92E647   BA 0003FE7F   mov    edx, 7FFE0300
7C92E64C   FF12          call   dword ptr [edx]
7C92E64E   C2 1000       retn   10
```

执行 HideOD 插件隐藏功能，ZwSetInformationThread 将被修改成类似下面这个样子：

```
7C92E642   B8 E5000000   mov    eax, 0E5
7C92E647   BA 08109100   mov    edx, 911008//这个地址是插件申请的
7C92E64C   FF12          call   [edx]
7C92E64E   C2 1000       retn   10
```

查看插件申请的地址，即[edx]所指向的代码是这样的：

```
00910250   5A            pop    edx                    ; kernel32.7C816FD7
00910251   E8 CAFFFFFF   call   00910220
00910256   B8 E5000000   mov    eax, 0E5
0091025B   BA 0003FE7F   mov    edx, 7FFE0300
00910260   FF12          call   [edx]
00910262   C2 1000       retn   10
```

中间调用了 910220，用来过滤掉 ThreadHideFromDebugger 这个动作：

```
00910220   33C0          xor    eax, eax
00910222   837C24 0C 11  cmp    dword ptr [esp+C], 11      ; ThreadHideFromDebugger
00910227   74 01         je     short 0091022A
00910229   C3            retn
0091022A   5A            pop    edx                        ; kernel32.7C816FD7
0091022B   C2 1000       retn   10
```

通过这种 Ring 3 级别的简单钩子，调用 ThreadHideFromDebugger 检测调试器的诡计就无声无息地幻灭了。

2. AntiHook/Splicing

Hook 是万能的，不过所有公开的 Hook 都是无用的。就像前面的例子一样，HideOD 插件钩住了 ZwSetInformationThread、ZwQueryInformationProcess 等。简单的 Ring 3 钩子 Hook 了 ZwXxx，所有依靠 Native API 的检测都失效了。

HideOD 中的代码是经过改良的，最早期插件直接将 ZwXxx 函数入口写入一个 jmp 指令，跳到另一个地方，过滤掉一个危险的操作，或者执行原来的函数，最后返回。Hying 的 PE-Armor 外壳中很早用到了

ThreadHideFromDebugger 技巧，很多人写了简单的插件甚至手工干脆把 ZwSetInformationThread 改成了“retn 10”来防止调试器与被调试程序脱钩，于是 Hying 改进了他的壳，当发现所要调用的函数代码不是以下两种形式时，就引发错误。

```
mov eax, xxx
lea edx, [esp+4]
int 2Eh
ret xxx

mov eax, xxx
mov edx, xxxxxxxx
call [edx]
retn xxx
```

这两种形式正是 Stub 函数在 Windows 2000 和 Windows XP 下的样子，没有哪个导出的 Native API 代码是其他的形式，除非它被下了断点或者被 Hook 了。这样一来直接写 jmp 指令来 Hook 或者直接“retn 10”的做法行不通了，不过 PE-Armor 并不对操作系统版本进行严格的检测。如果在 Windows 2000 下出现 Windows XP 的 Stub 也不会有异议，而 Windows XP 下的 Stub 函数保持格式的情况下还有修改第 2 行 xxxxxxxx 的余地，于是改进型的 Hook 产生了，就是 HideOD 中的 Hook：

```
//这段 hook 是 heXer 构造的
7C92E642    B8 E5000000    mov     eax, 0E5
7C92E647    BA 08109100    mov     edx, 911008  //这个地址是插件申请的
7C92E64C    FF12           call    [edx]
7C92E64E    C2 1000        retn    10
```

针对这个改进，自然而然想到的对策就是检测第 2 行的地址是否属于 ntdll，很明显 911008 不符合要求，不过这样还是不太奏效，因为在 ntdll 中找一个没有被使用的空间来存放 Hook 代码很容易，PE Header 里、区块间隙中有充足的空间。

这种无休止的小机关战争令人厌倦，俄罗斯程序员 PSI_H 的《Простой способ противодействия сплайсингу API》（对应的中文翻译《对付 API-splicing 的一种简单方法》）介绍了对付 API Hook 更高明一些的技巧。

Splicing 这个术语的含义是把一段代码抽走，放到壳动态申请的内存中，然后生成一个跳转指令跳过去执行，这样抓取内存镜像时就会丢掉这一部分，从而使脱壳变得麻烦一些。而现在的 Hook 程序，比如微软的 Detours 库，也是将函数的开头部分复制到另一段内存，再把原来的地方写上一个 jmp 指令跳到一个地方拦截对这个函数的调用，最后再执行复制出来的代码，继而转入原始的函数，所谓的“inline hook”指的也是类似的手段。

这里结合文中的代码介绍一下击败 Splicing 实现的 Hook：

```
// 将 NTDLL.DLL 文件拷入 TEMP 文件夹
 char szTemp[MAX_PATH];
 GetTempPath(MAX_PATH, szTemp);
 strcat(szTemp, "ntdll2.dll");
 CopyFile("C:\\Windows\\System32\\ntdll.dll", szTemp, TRUE);
 // 取得指向原始函数的指针
 HMODULE hMod = LoadLibrary(szTemp);
 void* ptr_orig = GetProcAddress(hMod, "ZwWriteVirtualMemory");
 // 取得指向当前函数的指针
     void* ptr_new=GetProcAddress (LoadLibrary ("ntdll.dll"),"ZwWriteVirtualMemory");
 // 设置内存访问权限
 DWORD dwOldProtect;
 VirtualProtect(ptr_new, 10, PAGE_EXECUTE_READWRITE, &dwOldProtect);
```

```
    // 替换函数的前 10 个（为保险起见）字节
    memcpy(ptr_new, ptr_orig, 10);
    FreeLibrary(hMod);
    DeleteFile(szTemp);
```

这段代码恢复了 Rizng 3 级对 ntdll.dll 中 ZwWriteVirtualMemory 这个函数的挂钩，因为 LoadLibrary 去加载一个已经加载过的 DLL 会直接返回它的 HINSTANCE，这里要的是原始的未经修改的函数代码，因此要把 DLL 复制到另一个地方改头换面进行加载。新加载的 DLL 没有被修改，因此就可以得到对应函数原始的代码，再把它们写回原来的地址，这样就摘除了 Hook。PSI_H 的文章还指出："尽管这里给出的摘除 Hook 的方法完全奏效，但需要加载新的 DLL 模块，这可能会引起防火墙的暴怒。所认为的更为优雅的办法就是只需从文件中读取所需要的字节，并且给出了一段例子代码，有兴趣的读者可以参考"。

Themida 也是通过读取文件来防止被 Hook，不过它做得更"过火"，干脆不恢复钩子直接去调用读出来的代码。而系统 DLL，例如 user32.dll，总是被加载到同一个地址，因此读出来的代码可以不需要重定位就可以调用，代码所访问的数据地址都是真实 DLL 中的有效地址，可以跟踪一下 softworm《一个小花招》提供的例子程序来体会一下。对付这种完全自己读取 DLL 代码的防止 Hook 的防御措施，一般是在程序读取 DLL 代码用的 Buffer 中搜索函数的特征码来下断点，可以参考 kanxue 老师的《如何中断 Themida 的 MessageBox 对话框》。

3. 小结

像所有的故事一样，事情应该有个结局了。自己读取 DLL 代码的方法对 Hook 有一定阻挡作用，可是毕竟 LoadLibrary(Ex)甚至 Themida 用的 ReadFile 这些函数还是会被 Hook（比如 deroko 写过 Themida-Spy）。回忆前面的内容，从 Windows 2000 到 Windows XP 都支持使用 int 2e 来调用 Native API，只需要用这样的代码：

```
push argn
...
push arg0
mov edx, esp
mov eax, index
int 2e
```

下面这种方式比较隐蔽地调用了 ThreadHideFromDebugger：

```
//code by shoooo
push    0
push    0
push    11
push    -2
mov     eax, 0C7
mov     edx, esp
int     2E
mov     eax, 0E5
mov     edx, esp
int     2E
mov     eax, 0EE
mov     edx, esp
int     2E
mov     eax, 136
mov     edx, esp
int     2E
add     esp, 10
```

这段代码直接在堆栈中设置好参数就通过 int 2e 进入内核了，所以不管如何修改 ntdll.dll 中的

ZwSetInformationThread 都不会对这个反调试代码有任何影响。为什么这段代码调用了 4 个不同的索引？shoooo 解释说，在不同的 NT 系统上 ZwSetInformationThread 的 SDT 索引不一样，调用 4 个让这个 Anti 技巧在不同的操作系统上都有效，可能有人认为用 GetProcAddress 得到 ZwSetInformationThread 再读取“mov eax, xxx”的立即数会得到准确的索引。而事实上，从外界获取信息越少越安全，因为如果有人把索引改成另一个函数的索引，会因为参数不合法而不产生任何操作。因此这里我们得到一点不同以往的认识，那就是硬编码并不总是坏的。反过来说，因为 4 个索引只有一个会“中标”，其他三次调用也会因为参数不合法而不产生操作，因此也不需要太担心这样的蛮力调用会有什么副作用。

这个 Anti 可以无视所有 Ring 3 的 Hook，将它用虚拟机保护起来，会让许多调试者挠头，却也不是完美的，因为 int 2e 还是会调用 KiSystemService，查找 SDT 中 ntoskrnl 中的函数。因此只要写一个驱动，通过修改 SDT，或者 inline hook ZwSetInformationThread，把过滤器放在内核中，这个努力了一整节的 Anti 还是被无情地淘汰掉。更简单的隐藏调试器的方案是运行 HideToolz，或者为 OllyDbg 安装一个 Phantom 插件，它们已经写好了这种驱动程序。

15.3 真正的奥秘：小技巧一览

把任何一个反调试手段单独摆出来，要化解它就如捏死一只小蚂蚁一样，不过请回忆一下童年的画册，成千上万的军团蚁聚在一起，黑压压的一片，所到之处遍地白骨，是多么可怕的景象。事实上，就像 hying 在《软件加密内幕中》说的，真正令破解者头疼的，并不是设计得多么精妙的一个新奇装置，往往就是那些司空见惯、单调乏味的反调试手段堆砌起来的东西，让天生不喜欢重复劳动的 Cracker 不得不陷入无趣的体力劳动中，拖垮他们的耐心，才是真正成功的加密。

在本节里，走马观花似地浏览一些五花八门的反调试小技巧，笔者还会介绍一些曾经流行一时却已失去生命力的东西，献给奋斗在那个年代的人。Ap0x 用 MASM 写了很多这方面的例子，笔者会借用。

15.3.1 SoftICE 检测方法

SoftICE 作为最著名的内核级调试器，早在 Windows 9x 时代就已成为最流行的调试工具。所以对抗 SoftICE 的办法早就被大家研究透彻。

1. 句柄检测

句柄方法检测原理是用 CreateFileA 或_lopen 函数试图获得 SoftICE 的驱动程序“\\.\SICE”(Windows 9x 版本)、“\\.\NTICE”（Windows NT 版本）等的句柄，如果成功，则说明 SoftICE 驻留在内存中。这种方法也称为 MeltICE 子类型。

但 DriverStudio 2.x 以后的版本用这种方法不能检测到。该系列的 Symbol Loader 检测 SoftICE 是否激活方法是先将 DriverStudio 安装序列号取出，经过简单的运算，得到 4 个字符“xxxx”，然后将“\\NTICE”与“xxxx”连接成“\\NTICExxxx”，最后用 CreateFile("\\NTICE xxxx ",…)来检测到 SoftICE 是否激活。详见 nmtrans.dll 中 NmSymIsSoftICELoaded 函数。

这种方法还可以用来检测 Filemon、Regmon 等工具。

2. BoundsChecker 后门

Numega 公司除了出品了 SoftICE 外，还给开发者提供了一个内存检测工具，那就是 BoundsChecker，利用它可以检测内存泄露、资源泄露等程序开发中常见的错误，SoftICE 为 BoundsChecker 留了一个后门接口。

```
mov     ebp, 'BCHK'
mov     ax, 4
int     3
```

```
cmp     al, 4                   ;SoftICE 存在时 AL 会发生变化
je      __no_debugger
```

3. SoftICE 后门指令

后门指令（Back Door Commands）通过中断 INT 03 来进行。DOS 时代，用后门指令可以获得 SoftICE 版本信息、设置断点和执行命令等。当然这些技术都已过时了，现在这些后门中可能只有一条 RET 指令了，什么都不做。

执行 SoftICE 命令的 INT 03 子功能描述如下：

```
入口参数：
-AX = 0911h
-SI = 4647h ('FG')
-DI = 4A4Dh ('JM')
-DS:DX -> ASCII 命令字符串(最多 100 个字节，0Dh 表示 OK，例如："HBOOT"，ODH，0)
返回值：无
```

每个 INT 03 子功能的入口参数都有相同的部分，即 SI = 4647h ('FG')；DI = 4A4Dh ('JM')。这两个入口值在 SoftICE 中叫做“魔法值（Magic values）”，凡是 SoftICE 的后门指令都必须以这两个数为标志。

后门指令现在主要是被用做检测 SoftICE，这种方法要结合 SEH 来实现，否则当 SoftICE 不存在时就会引起一个断点异常。

4. 判断 NTICE 服务是否运行

在 Windows NT/2000/XP 系统中，SoftICE 是一个内核设备驱动类型的服务，服务名称是 NTICE，因此可通过判断 NTICE 服务是否运行来检测 SoftICE。

5. 利用 UnhandledExceptionFilter 检测

SoftICE 作为系统级调试器，会把自己置为系统默认调试器并捕获系统异常，其做法是在载入时，会把 kernel32!UnhandledExceptionFilter 的第一字节“55”用“CC”来代替。因此就可以根据这个“CC”机器码，判断 SoftICE 是否加载。

6. int 2d

int 2d 本来是被内核 ntroskrnl.exe 运行 DebugServices 用的，但是也可以在 Ring 3 模式下使用它。如果在一个正常应用程序中使用 int 2d，将会发生异常，然而如果这个程序被附加了调试器，就不会产生异常。

```
push    offset _seh     ;\
push    fs:[0]          ;| 设置 SEH
mov     fs:[0], esp     ;/
int     2dh             ;如果有调试器，正常运行；否则，会触发异常
nop
pop     fs:[0]          ;\  消除 SEH
add     esp, 4          ;/
检测到调试器
_seh:
    未发现调试器
```

int 2d 不仅能够用来检测 Ring 3 下的调试器，同时也能检测 DbgMsg 驱动，这意味着，它可以用来检测 Ring 0 下的 SoftICE。

除此以外，int 2d 还有个妙用，因为附加调试器的程序在运行完 int 2d 后，会跳过此指令后的一个字节。

```
int     2dh
nop                     ;会被跳过
```

如果在调试器中步进或步过 int 2d，不同的调试器会有不同的执行方式，所以可以用 int 2d 来完成一些

代码混淆工作。这部分不是本章内容，有兴趣的朋友可以研究一下。

15.3.2　OllyDbg 检测方法

如果让所有的解密者都只能选择一个工具，大部分的人都会选择 OllyDbg。OllyDbg 非常的强大，对于它的检测变得很重要。

1．查找特征码

想想在生活中，怎样识别两张近似的脸孔？是的，根据特征！同样的，特征检测在计算机领域也被广泛应用。比如，杀毒软件就是根据特征码来辨识病毒的，庞大的病毒库里，包含的最主要内容就是形形色色的病毒特征码，这些特征码就是从病毒体内不同位置提取出来的一系列字节。对于 OllyDbg 的检测，也可以采用类似方法。

例如，对 OllyDbg 1.1 版本提取特征码：

（1）地址：401126h　特征码：83 3D 1B 01

（2）地址：43AA7Ch　特征码：8D 8E 83 21

当程序在运行时，对当前运行的所有进程做一次枚举，如果发现某进程的目标地址有这个特征码，就可以认定是监测到 OllyDbg 了。

```
While(Process32Next(…) != FALSE)
{
 OpenProcess(…);
 ReadProcessMemory(…, 0x401126, &buf1,4,….);
 ReadProcessMemory(…, 0x43AA7C, &buf2,4,….);

 If(buf1 == 0x833d1b01)&&(buf2 == 0x8d8e8321)
    ……                                    //找到OllyDbg
}
```

为了降低误报率，还可以多检测几个特征码。值得注意的是，这种方法只能检测某一个或某几个版本的 OllyDbg。由于不知道对手使用的是什么版本的 OllyDbg，所以这种方法并不能确保检测结果。

2．检测 DBGHELP 模块

调试器一般用 Microsoft 提供的 DBGHELP 库来装载调试符号，如果一个进程加载了 DBGHELP.DLL，那么它很可能是一个调试器。以用 CreateToolhelp32Snapshot 创建进程的模块快照，通过 Module32First 和 Module32Next 来枚举模块，看看是否有不良之辈。

但是这种检测是脆弱的，只需要把 DBGHELP 改名，再修改 OllyDbg 中对应的名字字符串，它就失效了。

3．查找窗口

不要忘记，Ring 3 调试器，也是一个普通的 Windows 程序而已。所以，可以用两种常见的方式去检测 OllyDbg：查找窗口和查找进程。

查找窗口，可以用三种方法：

（1）FindWindow

像所有的对话框一样，OllyDbg 的主窗口，也有它的标题和类名。使用这个 API 函数可以判断 OllyDbg 的主窗口是否打开。既可以通过类名来查找窗口，也可以通过标题来查找，如果要搜索子窗口，需要使用 FindWindowEx。

（2）EnumWindow

这个函数枚举所有顶级窗口，并调用指定的回调函数，可以在回调函数中用 GetWindowText 得到窗口

的标题，比较是否包含“OllyDbg”字样。

（3）GetForeGroundWindow

这个函数与前两种方法略有不同，它并不枚举窗口，而是返回前台窗口（用户当前工作的窗口）。系统分配给产生前台窗口的线程一个稍高一点的优先级。

当程序正在被调试时，调用这个函数将获得前台窗口，也就是 OllyDbg 的窗口句柄。

4. 查找进程

枚举进程检测是否有 OllyDbg.exe 进程存在。这个方法和查找窗口方法一样，都很好被跳过，调试者只要把 OllyDbg 稍作修改，改进程名字和标题名字就可以了。

5. SeDebugPrivilege 方法

在默认情况下，进程是没有 SeDebugPrivilege 权限的。然而进程通过 OllyDbg 和 WinDbg 之类的调试器载入的时候，SeDebugPrivilege 权限被启用了。这种情况是由于调试器本身会调整并启用 SeDebugPrivilege 权限，当被调试进程加载时 SeDebugPrivilege 权限也被继承了。

可以通过打开 CSRSS.EXE 进程间接地使用 SeDebugPrivilege 确定进程是否被调试。普通程序的默认权限，是无法对 CSRSS.exe 执行 OpenProcess 的，如果能够打开 CSRSS.EXE，则意味着进程启用了 SeDebugPrivilege 权限，由此可以推断进程正在被调试。这个检查能起作用是因为 CSRSS.EXE 进程安全描述符只允许 SYSTEM 访问，但是一旦进程拥有了 SeDebugPrivilege 权限，就可以忽视安全描述符 9 而访问其他进程。注意：在默认情况下，这一权限仅仅授予了 Administrators 组的成员，也就是说，如果用户以非管理员身份登录，该检测方法将失效。

下面是 SeDebugPrivilege 检查的例子：

```
call     [CsrGetProcessId]

;打开 CSRSS.EXE 进程
push     eax
push     FALSE
push     PROCESS_QUERY_INFORMATION
call     [OpenProcess]

;如果成功，被调试
test     eax,eax
jnz    .debugger_found
```

如果 OpenProcess()成功，则意味着 SeDebugPrivilege 权限被启用，这也意味着进程很可能被调试。

6. SetUnhandledExceptionFilter 方法

这个方法和异常处理有关，当一个异常未被任何 SEH 处理而到达 Unhandled Exception Filter（kernel32!UnhandledExceptionFilter）并且程序没有被调试时，Unhandled Exception Filter 将会调用 kernel32!SetUnhandledExceptionFilter 作为参数指定的高层 exception Filter。如果被调试，则把异常发往调试器。可以利用这一点，通过设置 exception Filter 然后抛出异常，如果程序被调试，那么这个异常将会被调试器接收；否则，控制被移交到 exception Filter 运行得以继续。利用 SetUnhandledExceptionFilter 检测调试器原理和 ProcessDebugPor 类似，看雪论坛 simonzh2000 写过一篇很详细、很完整的文章。

下面的示例中通过 SetUnhandledExceptionFilter 设置了一个高层的 exception Filter，然后抛出一个违规访问异常。如果进程被调试，调试器将收到两次异常通知；否则，exception Filter 将修改 CONTEXT.EIP 并继续执行。

```
;set the exception filter
push     .exception_filter
```

```
call     [SetUnhandledExceptionFilter]
mov      [.original_filter],eax

;throw an exception
xor      eax,eax
mov      dword [eax],0

;restore exception filter
push     dword [.original_filter]
call     [SetUnhandledExceptionFilter]

.exception_filter:
;EAX = ExceptionInfo.ContextRecord
mov      eax,[esp+4]
mov      eax,[eax+4]

;set return EIP upon return
add      dword [eax+0xb8],6

;return EXCEPTION_CONTINUE_EXECUTION
mov      eax,0xffffffff
retn
```

有些程序是直接通过 kernel32!_BasepCurrentTopLevelFilter 手工设置 exception Filter 的，这样做更加隐蔽，以防破解者在那个 API 上下断。

7. EnableWindow 方法

这是一个小小的伎俩，却往往可以让 Ring 3 调试器中招。调用这个 API 可以暂时锁定前台的窗口，让用户休息一下，顺便也让调试器无法工作。

```
EnableWindow(GetForegroudWindow(),FALSE);
```

处理完之后，当然要恢复用户的窗口了，要做得更“狠毒”一些，可以创建线程不断锁住所有的窗口。

8. BlockInput 方法

跟 EnableWindow 大同小异的小伎俩，调用 BlockInput(TRUE)可以锁住键盘，完成工作后用 BlockInput(FALSE) 恢复。这种锁定可以按“Ctrl+Alt+Del”键强制解除。

15.3.3 调试器漏洞

软件在与破解者的对抗中，并不是总处于被动防守的姿态，有的时候，攻击才是最好的防守方式。攻击调试器成为现在 Anti-Debug 技术中的重要一环。

只要是软件，就存在漏洞，调试器是软件中的一种，自然也不能例外。发现和利用它们自身的漏洞来攻击往往是非常有效的。下面以 OllyDbg 为例，列举一些已知的漏洞。

1. OutputDebugStringA

OutputDebugStringA 函数用于向调试器发送一个格式化的串，OllyDbg 会在底端显示相应的信息。OutputDebugString 漏洞本质上是一个格式化串的溢出漏洞。格式化串其实也是很严重的漏洞，轻则崩溃，重则可以导致执行任意代码。OllyDbg 的问题就是对格式化串过滤不严间接导致了缓冲区溢出的发生，保存在栈中的返回地址被覆盖。其实，以下库函数都存在格式化串溢出漏洞 printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf。

先来看一个简单的例子：

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
if( argc != 2 )
 {
   printf("输入一个字符串\n");
   return 1;
  }
 printf( argv[1] );
 printf( "\n" );
 return 0;
}
```

程序打印出自己的参数。如果输入“hello world”，则输出“hello world”。但是如果输入“%d”，会发现程序输出“4198693”，十六进制就是 401125。正常地打印一个十进制数值应该是带参数的，比如printf("%d",i)。i 就是一个整型变量。这里略了后者，当所有参数压栈完毕调用 printf 函数的时候，printf 并不能检查参数的正确性，只是机械式地从栈中取值作为参数，也就是看到的 4198693，这个时候堆栈就被破坏了，栈中的信息就泄露了。

尽管 OllyDbg 已经对 OutputDebugString 输出的字符串进行了长度检查，只接受 255 个字节，但是没有对提供的参数进行检查，所以间接导致了缓冲区的溢出。可以将参数设置为"%s%s%s "，调用 OutputDebugStringA，会让 OllyDbg 崩溃。

当然，经过精心构造的输出串，使 OllyDbg 溢出后执行任意代码也是完全可以做到的，这里不再详述。一些修改版本的 OllyDbg 已对这个漏洞进行了修补。

2. DRx 清理 BUG

OllyDbg 只要捕获到被调试程序的异常，就毫不留情地把所有 DRx(DR0-DR7)清零。这是一个不能称之为 bug 的 bug，但是却可以被利用。在异常中设置 DRx 的值，再利用这些值进行解码，已经成为了很多壳的手段。运用最成熟的，当属 Hying's PEArmor 壳。

PE-Armor 壳设置了很多 SEH，并在 SEH 中改写 DRx 的值，利用这些值进行解码，才能使程序正确地运行下去。如果使用 OllyDbg 遇到异常会清掉 DRx，用 DRx 值来解码就会出错。

15.3.4 防止调试器附加

防止调试器附加（Anti-Attach）是非常重要的，因为在不方便使用调试器启动程序的时候，调试者往往先运行目标程序，再使用调试器附加到目标进程（例如调试游戏时）。Ring 3 调试器的附加使用的是 DebugActiveProcess 函数，在附加相关进程时，会首先执行到 ntdll.dll 下的 ZwContinue 函数，最后停留在 ntdll.dll 的 DbgBreakPoint 处。熟悉 OllyDbg 的朋友一定对这个函数不陌生。事实上是调试器在这里设了一个 int 3 断点，由调试器自己捕捉。当调试者按 F9 键继续后，调试器再恢复这里的代码继续运行。

只要在这两处调试器 Attach 过程中设置一点点障碍，就能有效地阻止程序被附加调试。先看看下面这个例子：

```
    @get_api_addr    "NTDLL.DLL","ZwContinue"
    xchg  ebx,eax

    ;得到ntdll.dll的ZwContinue地址
    call  a1
    dd 0
a1: push  PAGE_READWRITE
```

```
    push 5
    push ebx
    call VirtualProtect
    @check  0,"Error: cannot deprotect the region!"

    ;申请内存读写权限
    lea edi,_ZwContinue_b
    mov ecx,0Fh
    mov esi,ebx
    rep movsb

    ;edi 寄存器指向我们自定义的一块大小为 0F 的内存区域
    ;这正是 Ntdll.ZwContinue 函数的大小。rep movsb 指令把
    ;原始 ZwContinue 函数复制到我们指定的 ZwContinue_b 处

    lea eax,_ZwContinue
    mov edi,ebx
    call  make_jump

    ; _ZwContinue 处地址放入 eax，原函数地址放入 edi，调用
    ;make_jump 在原函数开头构造一个跳转指令（常用伎俩）

    @debug  "attach debugger to me now!",
    MB_ICONINFORMATION

    exit:mov byte ptr [flag],1

    ;正常调用，flag 为 1
    push    0
    @callx  ExitProcess

make_jump:
    pushad
    mov byte ptr [edi],0E9h
    sub eax,edi
    sub eax,5
    mov dword ptr [edi+1],eax
    popad
    ret

    ;保留所有寄存器，构造跳转，使 ZwContinue 原函数跳入
    ;我们的_ZwContinue 执行

    flagdb  0
    ;定义 flag，用来判断是否被附加调试

_ZwContinue:pushad
    cmp byte ptr [flag],0
    jne we_q
    @debug  "Debugger found!",MB_ICONERROR
we_q:   popad

    ;判断 flag 是否为 0，如果为 0，检测到调试器，否则继续
    ;执行下面的代码，正是我们复制的 ZwContinue 原始代码
```

```
_ZwContinue_b:   db  0Fh dup (0)

comment $
    77F5B638   B8 20000000    MOV EAX,20
    77F5B63D   BA 0003FE7F    MOV EDX,7FFE0300
    77F5B642   FFD2           CALL EDX
    77F5B644   C2 0800        RETN 8
    $
    ;复制完成后，这里看起来应该是上边的样子
    end start
```

这段代码挂接了 Ntdll.ZwContinue，如果经过这里，则报告发现调试器；如果未经过这里，则说明程序没被附加调试。而对 DbgBreakPoint 函数，也可以采用同样的方式检测，或者可以直接在程序中检测这里是否有 int 3，也可以达到同样的目的。

15.3.5 父进程检测

理论上来说，当一个程序被正常启动时，它的父进程应该是 Exploer.exe（资源管理器启动）、cmd.exe（命令行启动）或者 Services.exe（系统服务）中的一种，当某进程的父进程并非上述三个进程之一时，一般可以认为它被调试了（或者被内存补丁之类的 loader 加载了）。

下面是实现这种检查的一种方法：

（1）通过 TEB（TEB.ClientId）或者使用 GetCurrentProcessId 来检索当前进程的 PID；

（2）用 Process32First/Process32Next 得到所有进程的列表，判断 explorer.exe 的 PID（通过 PROCESSENTRY32.szExeFile）和通过 PROCESSENTRY32.th32ParentProcessID 获得的当前进程的父进程 PID；

（3）如果父进程的 PID 不是 explorer.exe、cmd.exe 或 Services.exe 的 PID，则目标进程很可能被调试。

需要注意的是，在一些非正常启动进程情况下，例如某进程启动另一个进程时，这个调试器检查会引起误报。

15.3.6 时间差

一个程序被断点打断，CPU 捕获异常发给调试器，调试器处理后继续运行，这个时间显然比程序直接执行要慢很多，计算一个操作开始和结束运行的时间，如果慢得不合理，那么可以判断被跟踪了。

RDTSC 指令（Read Time-Stamp Counter）用来获得 CPU 自开机运行的时钟周期数。它的结果是 64 位的，保存到 eax 和 edx 两个寄存器中，本来设计为用来精确测量算法开销，不过现在利用两次使用该指令来计算时间差。

```
    rdtsc
    mov ecx,eax
    mov ebx,edx

    ; … 一些代码
    ;计算两次 RDTSC 指令之间的偏移
    rdtsc

    cmp edx,ebx ;检测高位
    ja  __debugger_found
    sub eax,ecx ;检测低位
    cmp eax,0x200
    ja  __debugger_found
```

当然，也可以采用 kernel32!GetTickCount() 实现类似功能的检测。

15.3.7　通过 Trap Flag 检测

CPU 符号位 EFLAGS 中一位叫做 TF，即 TRAP FLAG。当这个 TF=1 的时候 CPU 执行完 EIP 的指令就会触发一个单步异常。例如这段代码：

```
pushfd ;
push eflags
or dword ptr [esp], 100h ;TF=1
popfd                    ;这条指令之后 TF=1 了
nop                      ;这一条执行后会触发异常，应该提前安装 SEH，跳到其他地方执行
jmp die                  ;如果顺序执行下来了，说明被跟踪
```

15.3.8　双进程保护

Windows 下 Ring 3 调试器对被调试程序的关系是“一个萝卜一个坑”，一个进程只能有一个调试器。熟悉脱壳的朋友一定不会对 Armadillo 双进程保护功能陌生，它所使用的就是这种技术。

只需要对 Windows 提供的调试 API 稍加了解，就能够对自己的软件产品实行双进程保护了。一个简单的调试器应该实现以下一些功能：

（1）加载一个进程或附加到一个正在运行的进程上

使用 CreateProcess 创建进程时，指定 DEBUG_PROCESS 标志，来启动被调试进程，或者使用 DebugActiveProcess 函数绑定到某个正在运行的进程上。

（2）获得被调试程序的底层信息，包括进程 ID、映像基址等

使用 WaitForDebugEvent 等待调试事件发生，该函数阻塞调用线程直到等待到调试信息。

（3）接收被调试进程发来的调试事件并进行处理

当 WaitForDebugEvent 返回时，意味着在被调试进程中发生了调试事件，响应并处理该调试事件，继续执行被调试程序。

关于 Windows 调试 API 并非本章重点，这里只是略加阐述以便于理解双进程保护的机制，有兴趣的朋友可以自己实现一套自己的双进程保护系统。

第16章 外壳编写基础[①]

对一个程序文件加密有一个方面是必须要考虑的，这就是防止解密者对程序文件的非法修改和反编译。要实现这种保护最常用的方法之一就是给编译好的程序文件加上一个外壳。

对程序的加壳一般是选用一款现成的加壳软件。使用现成的加壳软件虽然很方便，但却存在着一些不可避免的缺点：越是先进、优秀的加壳软件有时反而会越不安全。为什么呢？因为加壳软件越优秀，用它加密的软件越多，研究它的人也会越多，其中必定有一些高手会分析出这些外壳所用的关键技术，并将其公开，有时甚至会针对这些加壳软件而写出专门的脱壳机。一旦一个加壳软件被写出脱壳机，那用它加密的软件的保密性就可想而知了。所以写一个自己专用的加壳软件还是有一定意义的。

16.1 外壳的结构

本章所讲述的加壳工具由两部分组成，第一部分是主体程序，主要是将原 PE 文件读入到内存，然后对 PE 文件各个部分加工，主要是各区块数据压缩，将输入表、重定位变形，最后将外壳部分与处理好的主体文件拼合。第二部分就是外壳部分，这部分主要是加壳后程序执行时候的引导段，它模拟 PE 加载器处理输入表、重定位表，最后跳到原程序执行。

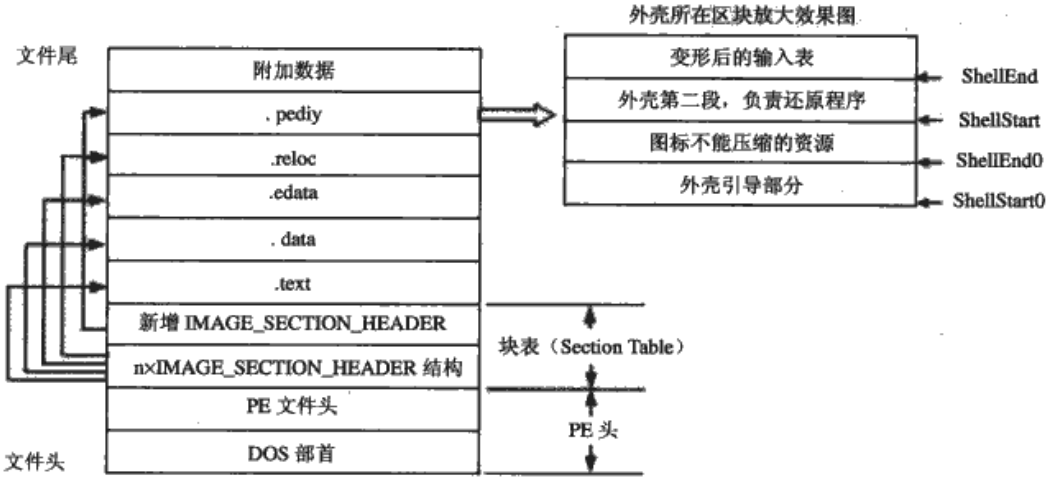


图 16.1 加壳后程序的结构图

最终装配成一个完整壳的程序结构如图 16.1 所示。目标程序新增了一个区块“.pediy”，这部分就是

[①] 本章由印豪编写。

“壳”。区块.text、.data 等是原始程序的代码数据，不过其现在是以压缩的形式保存的。另外，为了简化，本例没处理输出表，其所在区块.edata 仍以明文形式存在。在“.pediy”区块里，以 ShellStart 为分界，之前的部分以非压缩的方式存在，之后的部分是以压缩的方式存在的。新程序的入口点指向外壳 ShellStart0 开始的部分，外壳执行时先执行这部分，这部分的主要功能是将 ShellStart 开始的真正的外壳代码在内存中解压缩，并初始化一些数据。初始化完成后转移到 ShellStart 继续执行。ShellStart 开始的代码是外壳的真正部分，它的主要功能是还原原始程序（.text、.data 等区块数据），另一个重要功能则是阻止破解者的跟踪、脱壳。所以一般来说这段代码会比较长，里面还有各种反调试器、反 Dump 的代码。将它以压缩的方式存储一方面可以减小文件尺寸，另一方面也有利于程序的安全性。

16.2 加壳主程序

本章将通过一个实例，说明如何编写一个简单的加壳程序。在这个加壳程序中要实现的主要功能有：对程序的压缩、对资源的处理、对输入表的处理、对重定位表的处理、区块的融合和额外数据的保留等。为了突出重点，在此实例中基本不涉及对各种常用调试工具的防范，需要此类功能的读者，可以参考本书中的其他相关章节，添加各种相应的代码。

为了使源程序更易于讲解方便，主程序采用 Microsoft Visual C++ 6.0 编写，外壳部分采用汇编，因此需要读者有一定的汇编编程基础。完整的源码在配套光盘映像文件中。

16.2.1 判断文件是否为 PE 格式

本节程序处理的加密对象是 EXE 和 DLL 文件，所以在对文件进行处理前必须先判断目标文件是否为正确的 PE 格式。在本例中，文件格式的判断是通过使用一个自定义名为 IsPEFile()的函数来进行的。如果格式符合，函数返回 1，同时在消息框中输出格式正确的消息；否则返回 0，并输出相应的错误消息。

校验的方法是先检验文件头部第一个字的值是否等于 IMAGE_DOS_SIGNATURE，也就是字符串“MZ”，如果是则表示 DOS MZ header 有效。其次根据 e_lfanew 字段找到 PE header，校验比较 PE header 的第一个字的值是否等于 IMAGE_NT_SIGNATURE，也就是字符串“PE”。如果前后两个值都匹配，那就认为该文件是一个有效的 PE 文件。

通过校验 FileHeader 结构中 Characteristics 字段的值，判断是 EXE 文件还是 DLL 文件。特征值 IMAGE_FILE_DLL 表示该文件是 DLL。在 WINNT.H 中已经被定义：

```
#define IMAGE_FILE_DLL                  0x2000
```

实现检测的流程如下：

（1）判断文件开始的第一个字段是否为 IMAGE_DOS_SIGNATURE，即 5A4Dh。

（2）通过 e_lfanew 找到 IMAGE_NT_HEADERS，判断 Signature 字段的值是否为 00004550h，即 IMAGE_NT_SIGNATURE，如果是 IMAGE_NT_SIGNATURE，就可以认为该文件是 PE 格式。

（3）判断该 PE 文件是否可加壳。如果该文件只有一个区块就认为是被加壳了，或其入口点的值大于第二个区块虚拟地址值也认为其被加壳了。

（4）最后，校验一下 FileHeader 结构中 Characteristics 字段的值，判断是 EXE 文件还是 DLL 文件。

16.2.2 文件基本数据读入

文件格式判断正确后就可以将文件读入内存等待处理。文件的读入方式一般有两种：一种是直接按文件偏移的方式读入；另一种是依据 PE 文件的结构，仿照 Windows 装载器载入 PE 的方式，根据各个区块的 RVA 分别读入。

第一种读入方式编程时非常简单，只需利用 GetFileSize 函数取得欲处理文件的大小，然后申请一块同样大小的内存，再一次性读入整个文件即可，或者利用 CreateFileMapping 和 MapViewOfFile 将文件映射到内存也可以。读入后要使用文件中的某个数据时，只需要根据它的文件偏移（Offset）加上读入基址就可以找到这个数据，上一节的 IsPEFile()函数就是采用这种方式。这种方式虽然读入操作简单，但是在以后的处理中却有很大的弊端。因为在 PE 文件中，所有的数据都是根据 RVA 或 VA 来定位的，如果要使用一个根据 RVA 来定位的数据，那就先得把它的 RVA 转换为 Offset，然后才能找到并使用它。如果要处理的数据比较多且分布在不同区块中的话，这种转换就会显得非常麻烦，而且稍有疏忽就容易引起错误。所以笔者建议使用第二种读入方式。

第二种读入方式是先取得 PE 文件头的 SizeOfImage 字段的值，然后申请相应大小的内存，再根据文件的 Section Table 中的 VirtualAddress 和 VirtualSize 的值逐个区块分别读入。如此操作，要使用数据时只需根据数据的 RVA 加上读入基址就可找到并操作它了。

先定义几个全局变量，将读入内存的映像相关参数放入，方便以后随时读取。

```
UINT                    m_nImageSize = 0;    // 映像大小
PIMAGE_NT_HEADERS       m_pntHeaders = 0;    // PE 结构指针
PIMAGE_SECTION_HEADER   m_psecHeader = 0;    // 第一个 SECTION 结构指针
PCHAR                   m_pImageBase = 0;    // 映像基址
```

下面就是文件读取所使用的代码：

```
HANDLE hFile = CreateFile(szFilePath,GENERIC_READ|GENERIC_WRITE,FILE_SHARE _READ|\
               sFILE_SHARE_WRITE, NULL,OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
if ( hFile == INVALID_HANDLE_VALUE ) {
     AddLine(hDlg,"错误!文件打开失败!");
     return  FALSE;
}
// 读 DOS 头
ReadFile(hFile,&dosHeader, sizeof(dosHeader), &NumberOfBytesRW, NULL);//定位到到 e_lfanew
SetFilePointer(hFile,dosHeader.e_lfanew, NULL, FILE_BEGIN); // 读出 PE 头
ReadFile(hFile,&ntHeaders, sizeof(ntHeaders), &NumberOfBytesRW, NULL);//获取文件大小等信息
nFileSize       = GetFileSize(hFile,NULL);                    //文件大小
nSectionNum     = ntHeaders.FileHeader.NumberOfSections;      //区块数
nImageSize      = ntHeaders.OptionalHeader.SizeOfImage;       //映像尺寸
nFileAlign      = ntHeaders.OptionalHeader.FileAlignment;     //文件中区块对齐值
nSectionAlign   = ntHeaders.OptionalHeader.SectionAlignment;  //内存中区块对齐值
nHeaderSize     = ntHeaders.OptionalHeader.SizeOfHeaders;     //文件头大小

m_nImageSize= AlignSize(nImageSize, nSectionAlign);           //修正映像大小没有对齐的情况
m_pImageBase= new char[m_nImageSize];                         //申请内存用于保存映像
memset(m_pImageBase, 0, m_nImageSize);                        //将申请的内存区域初始化 0
SetFilePointer(hFile, 0, NULL, FILE_BEGIN);                   //首先定位并读 PE 文件头到内存中
ReadFile(hFile, m_pImageBase, nHeaderSize,&NumberOfBytesRW, NULL);
m_pntHeaders = (PIMAGE_NT_HEADERS)((DWORD)m_pImageBase + dosHeader.e_lfanew);
// 注：由于程序文件的 IMAGE_DATA_DIRECTORY 个数可以自定义(不一定非得定义 16 个)
// 因此这里通过计算来得到准确的 IMAGE_NT_HEADERS 的大小
nNtHeaderSize= sizeof(ntHeaders.FileHeader)+sizeof(ntHeaders.Signature)
                        + ntHeaders.FileHeader.SizeOfOptionalHeader;
m_psecHeader = (PIMAGE_SECTION_HEADER)((DWORD)m_pntHeaders + nNtHeaderSize);
// 循环依次读出 SECTION 数据到映像中的虚拟地址处
for (nIndex=0,psecHeader=m_psecHeader;nIndex<nSectionNum;++nIndex,++ psecHeader)
{
  nRawDataSize   = psecHeader->SizeOfRawData;
  nRawDataOffset      = psecHeader->PointerToRawData;
```

```
    nVirtualAddress    = psecHeader->VirtualAddress;
    nVirtualSize       = psecHeader->Misc.VirtualSize;
    SetFilePointer(hFile, nRawDataOffset,NULL,FILE_BEGIN); //定位到SECTION起始处
    // 读SECTION数据到映像中
    ReadFile(hFile,&m_pImageBase[nVirtualAddress],nRawDataSize,&NumberOfBytesRW,NULL);
}
```

16.2.3 附加数据读取

某些特殊的 PE 文件在各个区块的正式数据之后还有一些额外数据。这些额外数据不属于任何区段，所以当程序被 Windows 装载器载入时它们不会被直接读入内存，而是事后由程序在需要使用时自行读取。这些额外数据对于程序的运行一般是至关重要的，但是按照上一小节的方法将文件读入内存时，这些数据不会被读入。所以当加密完成重写文件时它们可能会丢失，造成程序无法运行。对于这类程序，必须在读入文件时将这些额外数据单独读取、保留，等待加密完成后再追加到文件的最后。

额外数据的起点可以认为是最后一个区块的末尾，终点是文件末尾，所以额外数据的大小就是文件大小减去文件头到最后一个区块的末尾的大小。

读取额外数据所使用的代码如下：

```
/*-------------------------------------------------------------*/
/*保存额外数据地址：MapOfSData                                 */
/*额外数据大小：nMapOfSDataSize                                */
/*-------------------------------------------------------------*/
if(IsSaveSData) // 保存额外数据吗
{
    nMapOfSDataSize=nFileSize-(psecHeader->PointerToRawData+psecHeader-> SizeOfRawData);
    if(nMapOfSDataSize>0) // 额外数据大小大于0则保存之
    {
        MapOfSData = new char[nMapOfSDataSize];   // 申请内存以保存额外数据
        memset(MapOfSData , 0, nMapOfSDataSize);  // 将刚申请的内存清零
        ReadFile(hFile, MapOfSData, nMapOfSDataSize,&NumberOfBytesRW, NULL);
        AddLine(hDlg,"额外数据读取完毕.");
    }
    else
        AddLine(hDlg,"没有额外数据.");
}
```

16.2.4 输入表处理

在如今常见的那些带加密功能的外壳中，破坏原程序的输入表几乎是必有的功能，而且是各出奇招，尽可能让脱壳者无法修复。

要破坏一个程序的输入表一般要有两步。第一步是在加密时做的，它破坏原程序中的输入表，将其换一个形式存储。但是仅此还不够，如果外壳初始化原程序时仍将各个函数的正确地址写回输入表，那么借助某些工具就可以轻易地重建一个可用的输入表，所以破坏输入表还需要第二步。

破坏输入表的第二步就是外壳对原程序进行初始化时不把真正的函数入口地址写回输入表，写回的是外壳中一段程序的入口，通过那段程序的变换后，再转到正确的函数入口去执行。这样通过中间的一些变换，就可以欺骗一些重建输入表的软件。如果不能重建输入表的话，脱壳也就失败了。在本例中只研究破坏的第一步，至于第二步读者可以参考其他一些源码。在看程序之前，先来分析一下程序正常载入时输入表的初始化过程。

首先系统根据输入表项中的 Name 字段找到 DLL 名，根据 DLL 名获取 DLL 在内存中的句柄，然后再根据 OriginalFirstThunk 字段找到 IMAGE_THUNK_DATA 结构，它一般是指向 IMAGE_IMPORT_BY_

NAME 的指针数组，或者也可能是函数在 DLL 中的序列。根据函数序列或 IMAGE_IMPORT_BY_NAME，就可以得到函数的入口地址，再将获取的这些入口地址写回到 FirstThunk 指向的 IMAGE_THUNK_DATA 结构数组就可以了。如果 OriginalFirstThunk 为零，则用 FirstThunk 代替。

由此可知，在转储后的输入表结构中只要包含了 FirstThunk 也就知道了要初始化的数据的地址，知道了 DLL 名和函数名或函数序号也就知道了要填这些地址的函数入口，知道这些就可以完成对原程序输入函数的初始化了。据此，笔者设计了如图 16.2 所示的新输入表结构。

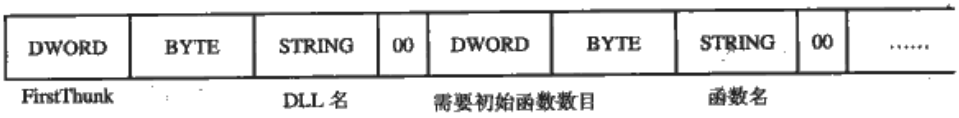


图 16.2　新输入表结构

在此结构中每个字符串前的一个字节中保存了该字符串的长度，其后为“00”的字节表示了字符串的结束，在函数名字符串前的那个字节中如果存储的是“00”，则字符串中不是函数名而是函数序号。当然，读者也可以设计自己的新输入表结构，甚至可以比笔者的更简单，只要能正确完成原输入表的初始化就可以了。

转储输入表所用的代码如下：

```
UINT MoveImpTable(PCHAR m_pImportTable)
{

PIMAGE_IMPORT_DESCRIPTOR pImportDescriptor = NULL, pDescriptor = NULL;
PIMAGE_DATA_DIRECTORY    pImportDir = NULL;
PCHAR                    pszDllName = NULL;
UINT                     nSize = 0;
PCHAR                    pData = NULL;
PCHAR                    pFunNum =NULL;
PIMAGE_THUNK_DATA32      pFirstThunk = NULL;
PIMAGE_IMPORT_BY_NAME    pImportName = NULL;

pImportDir=&m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
pImportDescriptor =(PIMAGE_IMPORT_DESCRIPTOR)RVAToPtr(pImportDir->VirtualAddress);
// 遍历原始输入表
for(pData=m_pImportTable,pDescriptor=pImportDescriptor;pDescriptor->Name!=0; pDescriptor ++)
{
    *(DWORD *)pData = pDescriptor->FirstThunk;          //保存首 THUNK 数据的 RVA 地址
    pData += sizeof(DWORD);
    pszDllName = (PCHAR)RVAToPtr(pDescriptor->Name); //保存 DLL 名称长度(WORD)
    *(BYTE *)(pData) = (BYTE)(strlen(pszDllName) );   //DLL 字符串长度
    pData += sizeof(BYTE);
    memcpy(pData,pszDllName,strlen(pszDllName)+ 1);   // 保存 DLL 字符串
    pData += strlen(pszDllName) + 1;
    pFunNum = pData;              // pFunNum 指向需要初始函数的数目
    *(DWORD *)pFunNum =0;
    pData += sizeof(DWORD);   //指向“BYTE |STRING|00|…”
    // 在 OrginalFirstThunk 无效时，才取 FirstThunk 作为函数名称等信息的存放位置
    if (pDescriptor->OriginalFirstThunk != 0){
        pFirstThun =(PIMAGE_THUNK_DATA32)RVAToPtr (pDescriptor-> Original FirstThunk);
    }
    else{
        pFirstThunk = (PIMAGE_THUNK_DATA32)RVAToPtr(pDescriptor->FirstThunk);
    }
    while (pFirstThunk->u1.AddressOfData != NULL)
```

```
    {
        if (IMAGE_SNAP_BY_ORDINAL32(pFirstThunk->u1.Ordinal)) // 函数以序号方式
        {
            *(BYTE *)pData = 0;
            pData += sizeof(BYTE);
            // 如果该元素值的最高二进位为 1，那么是序数
            *(DWORD *)pData = (DWORD)(pFirstThunk->u1.Ordinal & 0x7FFFFFFF;
            pData += sizeof(DWORD)+1;
           (*(DWORD *)pFunNum) ++;// 计数器，函数个数加 1
        }
        else   // 函数以字符串方式
        {
            pImportName = (PIMAGE_IMPORT_BY_NAME)RVAToPtr((DWORD)(pFirstThunk->\
                        u1.AddressOfData));
           *(BYTE *)pData = (BYTE)(strlen((char *)pImportName->Name));//函数名长度
            pData += sizeof(BYTE);
            memcpy(pData, pImportName->Name, strlen((char *)pImportName->Name) + 1);
           (*(DWORD *)pFunNum) ++;//计数器，函数个数加 1
            pData += strlen((char *)pImportName->Name) + 1;
        }
        pFirstThunk ++;
    }
}
*(DWORD *)pData = (DWORD)0;  // DLL 结构的结束符
pData += sizeof(DWORD);  // 计算实际大小
return(pData - m_pImportTable);
}
```

程序的输入表转储完成后，应该将原来的输入表清除，以防止被脱壳者利用。使用的代码如下：

```
void ClsImpTable( )
{
PIMAGE_IMPORT_DESCRIPTOR pImportDescriptor = NULL, pDescriptor = NULL;
PIMAGE_DATA_DIRECTORY        pImportDir = NULL;
PCHAR                        pszDllName = NULL;
PIMAGE_THUNK_DATA32          pFirstThunk = NULL;
PIMAGE_IMPORT_BY_NAME        pImportName = NULL;

pImportDir=&m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT];
pImportDescriptor=(PIMAGE_IMPORT_DESCRIPTOR)RVAToPtr(pImportDir->VirtualAddress);
// 遍历原始输入表，每循环一次，清除一个 DLL 的信息
for (pDescriptor = pImportDescriptor; pDescriptor->Name != 0; pDescriptor ++)
{
  pszDllName = (PCHAR)RVAToPtr(pDescriptor->Name);
  memset(pszDllName, 0, strlen(pszDllName)); // 擦除 DLL 字符串信息
  // 擦除原始 THUNK 数据
  if (pDescriptor->OriginalFirstThunk != 0)
  {
    pFirstThunk=(PIMAGE_THUNK_DATA32)RVAToPtr(pDescriptor->OriginalFirstThunk);
    while (pFirstThunk->u1.AddressOfData != NULL)//清除 OriginalFirstThunk
    {
    if (IMAGE_SNAP_BY_ORDINAL32(pFirstThunk->u1.Ordinal))
          memset(pFirstThunk, 0 , sizeof(DWORD));
    else {
         pImportName = (PIMAGE_IMPORT_BY_NAME)RVAToPtr((DWORD) (pFirstThunk->\
                  u1.AddressOfData));
```

```
            memset(pImportName,0,strlen((char *)pImportName->Name) +sizeof (WORD));
            memset(pFirstThunk, 0 ,sizeof(DWORD));
          }
        pFirstThunk ++;
      }
    }
    pFirstThunk = (PIMAGE_THUNK_DATA32)RVAToPtr(pDescriptor->FirstThunk);
    //清除 FirstThunk
    while (pFirstThunk->u1.AddressOfData != NULL)
    {
      memset(pFirstThunk, 0 ,sizeof(DWORD));
      pFirstThunk ++;
    }
    memset(pDescriptor, 0 ,sizeof(IMAGE_IMPORT_DESCRIPTOR));
  }
}
```

16.2.5 重定位表处理

在本例中重定位数据的去除是利用一个自定义函数 ClsRelocData()来完成的。它的功能是找到重定位数据所对应的区块，将区块的文件大小改为 0，同时将该区块所有数据清零。

下面是程序代码：

```
void ClsRelocData( )
{
    PIMAGE_BASE_RELOCATION   pBaseReloc = NULL;
    PIMAGE_DATA_DIRECTORY    pRelocDir = NULL;
    UINT                     nSize = 0;

pRelocDir=&m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
pBaseReloc =(PIMAGE_BASE_RELOCATION)RVAToPtr(pRelocDir->VirtualAddress);
if (pRelocDir->VirtualAddress == 0) // 如果没有重定位数据，则直接返回
    return ;
// 擦除所有的重定位数据
while (pBaseReloc->VirtualAddress != 0)
{
    nSize = pBaseReloc->SizeOfBlock;
    memset(pBaseReloc, 0 , nSize);
    pBaseReloc = (PIMAGE_BASE_RELOCATION)((DWORD)pBaseReloc + nSize);
}
m_pntHeaders->OptionalHeader.DataDirectory[5].VirtualAddress = 0;
m_pntHeaders->OptionalHeader.DataDirectory[5].Size = 0;
}
```

对于一般的 EXE 文件，它的实际载入基址与优先载入基址一般是相同的，所以它的重定位数据一般是无用的，可以把它去除，这可以使文件更小。但是对于 DLL 的动态链接库文件来说，Windows 系统没有办法保证每一次 DLL 运行时提供相同的基地址，这样“重定位”就很重要了，重定位数据一般是必需的。为了保证程序能运行起来，外壳必须模拟 PE 加载器的重定位功能，对相关代码重定位，否则原程序中的代码是无法正常运行起来的。

为了提高壳的强度，将原始的重定位表换个形式存储，外壳程序运行时会根据这个结构重定位相关代码。转储的新重定位结构如下：

```
typedef struct _NEWIMAGE_BASE_RELOCATION {
    BYTE   type;
```

```
    DWORD  FirstTypeRVA;
    BYTE   nNewItemOffset[1];
}
```

- type：重定位表的类型，由于是讨论 i386 架构情况，本例仅考虑了 TypeOffset 数组的类型为 IMAGE_REL_BASED_HIGHLOW 的情况。
- FirstTypeRVA：这一组重定位数据的开始 RVA 加上 TypeOffset 数组第 1 项的低 12 位（ItemOffset 值）。
- nNewItemOffset：是一个数组。数组大小每项为 1 个字节。每一项的值是当前的 ItemOffset 值与上一项的 ItemOffset 差值。

在转储后的重定位表结构中，FirstTypeRVA 指出了第一项重定位的地址，以后每项在这个基础上加上差值 nNewItemOffset，依次定位到所有的重定位地址。如图 16.3 所示是处理前后的重定位表结构的一个样例，这样的结构不光提高了壳的强度，还减少了重定位表数据，缩小了文件体积。

处理前

VirtualAddress	SizeOfBlock	TypeOffset					
0x00001000	0x00000170	0x302E	0x3043	0x3054	0x305F	0x3069	…

处理后

type	FirstTypeRVA	nNewItemOffset					
0x3	0x0000102E	0x15	0x11	0x0B	0x0A	0x06	…

图 16.3　处理前后的重定位表结构

当然，读者也可以设计自己的重定位表结构，只要能正确重定位指定代码就可以了。转储重定位表所用的代码如下：

```
BOOL SaveReloc()
{
//由于 WINNT.H 中，IMAGE_BASE_RELOCATION 结构没用 TypeOffset，故重新定义
typedef struct _IMAGE_BASE_RELOCATION2 {
    DWORD   VirtualAddress;
    DWORD   SizeOfBlock;
    WORD    TypeOffset[1];
} IMAGE_BASE_RELOCATION2;
typedef IMAGE_BASE_RELOCATION2 UNALIGNED * PIMAGE_BASE_RELOCATION2;

PIMAGE_DATA_DIRECTORY    pRelocDir = NULL;
PIMAGE_BASE_RELOCATION2  pBaseReloc = NULL;
PCHAR                    pRelocBufferMap =NULL;
PCHAR                    pData = NULL;
UINT                     nRelocSize = NULL;
UINT                     nSize = 0;
UINT                     nType = 0;
UINT                     nIndex = 0;
UINT                     nTemp = 0;
UINT                     nNewItemOffset =0;
UINT                     nNewItemSize =0;

pRelocDir=&m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
nRelocSize = pRelocDir->Size;
pBaseReloc =(PIMAGE_BASE_RELOCATION2)RVAToPtr(pRelocDir->VirtualAddress);
if (pRelocDir->VirtualAddress == 0)
    return TRUE;
pRelocBufferMap = new char[nRelocSize];
```

```
ZeroMemory(pRelocBufferMap, nRelocSize);
pData =pRelocBufferMap;
while (pBaseReloc->VirtualAddress != 0)
{
    nNewItemSize = (pBaseReloc->SizeOfBlock-8)/2;//保存新数据需要的字节长
    while (nNewItemSize != 0)
    {
        nType = pBaseReloc->TypeOffset[nIndex] >> 0x0c;//取 type
        if(nType ==0x3 )
        { // 取出 ItemOffset，加上本段重定位起始地址，减去 nTemp
            nNewItemOffset = ((pBaseReloc->TypeOffset[nIndex] & 0x0fff)\
                            + pBaseReloc->VirtualAddress) - nTemp;
            if(nNewItemOffset > 0xff)//如果是本段重定位数据第一项
            {
                *(BYTE *)(pData) = 3;
                pData += sizeof(BYTE);
                *(DWORD *)pData = (DWORD)(nNewItemOffset);
                pData += sizeof(DWORD);
            }
            else
            {
                *(BYTE *)(pData) =(BYTE)(nNewItemOffset);
                pData += sizeof(BYTE);
            }
            nTemp += nNewItemOffset;
        }
        nNewItemSize--;
        nIndex++;
        }
        nIndex    = 0;
    pBaseReloc=(PIMAGE_BASE_RELOCATION2)((DWORD)pBaseReloc+ pBaseReloc ->SizeOfBlock);
}
memset((PCHAR)RVAToPtr(pRelocDir->VirtualAddress),0,nRelocSize);
memcpy((PCHAR)RVAToPtr(pRelocDir->VirtualAddress),pRelocBufferMap,nRelocSize);
delete pRelocBufferMap;
return TRUE;
}
```

这段代码将目标文件的重定位处理好后，再放回重定位表所在的地方。一些加壳软件没有重定位表转储这一步，脱壳后，完整的重定位表就在文件里，如 tElock、ASProtect 等。

16.2.6 文件的压缩

如果加壳时用到了压缩技术，那么在解密之前还有一道工序，当然是解压缩。这也是一些壳的特色之一，比如说原来的程序文件未加壳时为 1～2MB 大小，加壳后反而只有几百 KB。

一般现在的加壳软件都具有压缩功能，压缩后的程序比较小，便于交流，而且保密性也比较好。

压缩算法是采用现有的压缩引擎，压缩引擎在选择上除了考虑压缩率外，更关键的是其解压速度要快。因为，解压速度快，加壳程序加载速度才不至于受太大的影响。加壳软件一般采用较多的压缩引擎有 aPlib、JCALG1、LZMA 等。aPlib 引擎对于小文件压缩效果较好；JCALG1 具有更为强劲的压缩效果，对大文件的压缩效果好；LZMA 是 7-Zip 程序中 7z 格式的默认压缩算法，具有很高的压缩比。

在本例中压缩引擎使用的是公开的 aPlib 压缩函数库（www.ibsensoftware.com），读者可以到此下载到包含有多种编程语言使用资料的完整压缩包。VC 中调用 aPlib 很简单，只需包含 aplib.h 库即可调用 aPlib

相关函数了。压缩前调用函数 m_nSpaceSize 估算出需要的内存空间大小，然后调用 aP_pack()函数压缩数据，该函数最后两个参数是回调函数，主要用于配合显示压缩进度条。压缩代码如下：

```
/*------------------------------------------------------------*/
/*压缩后的地址:m_pPackData 大小: m_nPackSize (全局变量)        */
/*------------------------------------------------------------*/
BOOL PackData(PCHAR pData, UINT nSize)
{
PCHAR       pCloneData = NULL;
UINT        m_nSpaceSize=NULL;

m_nSpaceSize= aP_workmem_size(nSize);                    // 计算工作空间大小
m_pWorkSpace= new CHAR[m_nSpaceSize];                    // 申请工作空间
m_pPackData = new CHAR[nSize * 2];                       // 申请保存压缩数据的空间
pCloneData  = (PCHAR)GlobalAlloc(GMEM_FIXED, nSize);    // 申请空间
memcpy(pCloneData, pData, nSize);                        // 复制原始数据到新空间
// 对原始数据进行压缩
m_nPackSize=aP_pack((PBYTE)pCloneData,(PBYTE)m_pPackData,nSize,(PBYTE)m_pWorkSpace,0,0);
GlobalFree(pCloneData);                                  // 释放空间
pCloneData = NULL;
if (m_nPackSize == 0) return FALSE;                      //压缩过程中发现错误
return TRUE;
}
```

PE 文件的压缩一般是按区块进行的，这样可以较好地保持原始程序文件的结构，方便程序的载入与还原。但是有些区块由于其功能的特殊性，当程序被系统载入时里面的数据会被系统所使用，所以是不能压缩的。有些则要做一些特殊处理后才能压缩。常见的此类区块有资源块（.rsrc）、输出块（.edata）等。其中由于资源区块处理方法的特殊性，因此将它作为一小节，在下面单独讲解。

在本例中程序的压缩是通过一个自定义函数来做的。具体函数如下：

```
/*------------------------------------------------------------*/
/*自定义压缩函数                                              */
/*------------------------------------------------------------*/
BOOL PackFile(TCHAR *szFilePath,UINT FirstResADDR)
{
PIMAGE_SECTION_HEADER    psecHeader = m_psecHeader;
UINT                     nSectionNum = 0;
UINT                     nFileAlign = 0;
UINT                     nSectionAlign = 0;
UINT                     nSize = 0;
DWORD                    nbWritten;
UINT                     nIndex = 0;
PCHAR                    pData = NULL;
UINT                     nNewSize = 0;
UINT                     nRawSize = 0;

hPackFile = CreateFile(szFilePath,GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ|\
            FILE_SHARE_WRITE, NULL,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
if ( hPackFile == INVALID_HANDLE_VALUE ) {
      return  FALSE;
    }
nSectionNum   = m_pntHeaders->FileHeader.NumberOfSections;
nFileAlign    = m_pntHeaders->OptionalHeader.FileAlignment;
nSectionAlign = m_pntHeaders->OptionalHeader.SectionAlignment;
```

```
// 计算新的文件头的大小(已考虑增加一个区段)
nSize = (PCHAR)(&psecHeader[nSectionNum + 1]) - (PCHAR)m_pImageBase;
nSize = AlignSize(nSize,nFileAlign);//对齐
m_pntHeaders->OptionalHeader.SizeOfHeaders = nSize; // 要修正文件头中的 SizeOfHeaders 大小
psecHeader->PointerToRawData = nSize;// 同时还要修正第一个区块的 RAW 地址
WriteFile(hPackFile,(PCHAR)m_pImageBase,nSize,&nbWritten,NULL);// 写入原各区块数据
for (nIndex = 0; nIndex < nSectionNum; nIndex ++, psecHeader ++)
{
    pData = RVAToPtr(psecHeader->VirtualAddress);
    nSize=psecHeader->Misc.VirtualSize;
    // 注：由于某些区块之前的一些擦除操作已经清除了部分数据，导致区块可能变小
    // 因此这里通过搜索并去掉尾部无用的零字节，重新计算区块的大小
    nNewSize = CalcMinSizeOfData(pData, nSize);
    // 如果整个区块已经只剩零字节，则可以不需要保存此区块的数据
    if (nNewSize == 0)
    {
        psecHeader->SizeOfRawData = 0;
        psecHeader->Characteristics |= IMAGE_SCN_MEM_WRITE;
        // 由于压缩的原因，因此必须每次都修正下一个区块的起始偏移地址
        if (nIndex != nSectionNum - 1)
        {
            psecHeader[1].PointerToRawData = psecHeader->PointerToRawData +\
             psecHeader->SizeOfRawData;
        }
        continue;
    }
    if (IsSectionCanPacked(psecHeader)) // 判断当前区块数据能否被压缩
    {
        PackData(pData, nNewSize);
        nRawSize  = AlignSize(m_nPackSize, nFileAlign);
        // 写入压缩后的数据
       WriteFile(hPackFile,(PCHAR)m_pPackData,m_nPackSize,&nbWritten,NULL);
        // 写入为对齐而填充的零数据
        if (nRawSize - m_nPackSize > 0)
            FillZero(hPackFile, nRawSize - m_nPackSize); //该函数见光盘
        psecHeader->SizeOfRawData = nRawSize;// 修正区块的大小
        // 记录压缩过的区块信息，用于外壳在运行时解压缩，该函数源码见光盘
      AddPackInfo(psecHeader->VirtualAddress,psecHeader ->Misc.VirtualSize,\
                psecHeader->SizeOfRawData);
    }
    else
    {
        //对资源区段特殊处理
        if ((strcmp((char *)psecHeader->Name, ".rsrc") == 0)&& isPackRes)
        {
            ……  //省略资源段压缩处理部分，在下一节中详细介绍
        }
        else// 不能压缩的区块，则直接保存区块的数据
        {
            nRawSize  = AlignSize(nNewSize, nFileAlign);
            WriteFile(hPackFile,(PCHAR)pData,nRawSize,&nbWritten,NULL);
            psecHeader->SizeOfRawData = nRawSize;
        }
    }
    // 由于压缩的原因，因此必须每次都修正下一个区块的起始偏移地址
```

```
    if (nIndex != nSectionNum - 1)
    {
        psecHeader[1].PointerToRawData = psecHeader->PointerToRawData +\
                                         psecHeader->SizeOfRawData;
    }
    psecHeader->Characteristics |= IMAGE_SCN_MEM_WRITE;
}
return TRUE;
}
```

16.2.7　资源数据处理

一般程序的资源都集中在资源区块（.rsrc）中，对于那些资源格式比较特殊的程序文件，由于它们不具有普遍性，在此就不去讨论了。

资源区块的大致结构先是资源目录，紧接在后面的才是资源数据。系统必须根据资源目录才能找到正确的资源数据。一般的资源只有在程序真正运行时才会被用到，但是有些特殊类型的资源却不是这样，即使程序没有运行，它们也可能会被系统读取、使用。例如，当使用“我的电脑”查看某个文件夹的内容时，该目录下的所有应用程序的图标都会被显示出来。这些图标就是程序资源的一部分，它们在程序没有被执行时仍然会被系统读取。这种特殊的资源类型通常包括：Icon（图标）、Group icon（组图标）、Version information（版本信息）等，它们一般是不能压缩的，因为它们一旦被错误处理，轻则产生程序异常如丢失图标，重则可能根本无法运行。另外，由于系统必须根据资源目录才能找到它们，所以资源目录也不能压缩。因此，对于资源区块的压缩一般需要分成三步来做：

（1）找到资源目录和资源数据的分割点，也就是资源数据的起点，在这之前的资源目录部分不能被压缩。

（2）把那些不能压缩的资源类型如图标等从资源数据中提取出来，转移到一个不被压缩的部分，一般是在外壳数据中找一段空间存放它们。同时还要修改资源目录项中相应的指针，以保证当它们被移动后系统仍然可以找到它们。

（3）压缩资源区块中的资源数据。

下面就这三步分别来分析。

第一步使用一个自定义函数来完成，通过这个函数将找到的资源数据起点 RVA 地址作为返回值放入 EAX。函数所依据的原理是遍历所有的资源数据项，比较它们的起始 RVA，找出最小的那个作为整个资源数据的起点。函数中涉及资源目录结构的部分，读者可以参考前面章节或 MSDN 里的资料。函数代码如下：

```
/*-------------------------------------------------------------*/
/* 函数功能：查找程序资源数据的起点                            */
/*-------------------------------------------------------------*/
UINT FindFirstResADDR()
{
UINT                                FirstResAddr = NULL;
PIMAGE_DATA_DIRECTORY               pResourceDir = NULL;
PIMAGE_RESOURCE_DIRECTORY           pResource = NULL;
PIMAGE_RESOURCE_DIRECTORY           pTypeRes = NULL;
PIMAGE_RESOURCE_DIRECTORY           pNameIdRes = NULL;
PIMAGE_RESOURCE_DIRECTORY           pLanguageRes = NULL;
PIMAGE_RESOURCE_DIRECTORY_ENTRY     pTypeEntry = NULL;
PIMAGE_RESOURCE_DIRECTORY_ENTRY     pNameIdEntry = NULL;
PIMAGE_RESOURCE_DIRECTORY_ENTRY     pLanguageEntry = NULL;
PIMAGE_RESOURCE_DATA_ENTRY          pResData = NULL;
……
//FirstResAddr 里现放置一个较大值（这里取映像尺寸），然后根据比较逐渐减小
FirstResAddr = m_pntHeaders->OptionalHeader.SizeOfImage;
```

```
pResourceDir = &m_pntHeaders->OptionalHeader.DataDirectory[2];
if (pResourceDir->VirtualAddress == NULL) //如果没资源则返回 0
    return FALSE;
//资源起点地址
pResource    = (PIMAGE_RESOURCE_DIRECTORY RVAToPtr(pResourceDir-> Virtual Address);
pTypeRes     = pResource;
nTypeNum     = pTypeRes->NumberOfIdEntries + pTypeRes->NumberOfNamedEntries;
pTypeEntry   = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)pTypeRes + \
               sizeof(IMAGE_RESOURCE_DIRECTORY));
for (nTypeIndex = 0; nTypeIndex < nTypeNum; nTypeIndex ++, pTypeEntry ++)
{
    //该类型目录地址
    pNameIdRes = (PIMAGE_RESOURCE_DIRECTORY)((DWORD)pResource + \
                 (DWORD)pTypeEntry->OffsetToDirectory);
    //该类型中有几个项目
    nNameIdNum=pNameIdRes->NumberOfIdEntries+pNameIdRes->\NumberOfNamedEntries;
    pNameIdEntry = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)pNameIdRes\
                   + sizeof(IMAGE_RESOURCE_DIRECTORY));

    for(nNameIdIndex=0;nNameIdIndex<nNameIdNum;nNameIdIndex++,pNameIdEntry++)
    {
        //该项目目录地址
        pLanguageRes = (PIMAGE_RESOURCE_DIRECTORY)((DWORD)pResource +\
                       (DWORD)pNameIdEntry->OffsetToDirectory);
        nLanguageNum = pLanguageRes->NumberOfIdEntries+ pLanguageRes->NumberOfNamedEntries;
        pLanguageEntry = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)\
                         pLanguageRes + sizeof(IMAGE_RESOURCE_DIRECTORY));

        for (nLanguageIndex = 0; nLanguageIndex < nLanguageNum;nLanguageIndex ++,
            pLanguageEntry ++)
        {
            pResData = (PIMAGE_RESOURCE_DATA_ENTRY)((DWORD)pResource + \
                       (DWORD)pLanguageEntry->OffsetToData);
            if((pResData->OffsetToData < FirstResAddr) && (pResData->\
                OffsetToData>pResourceDir->VirtualAddress))
            {
                FirstResAddr = pResData->OffsetToData;
            }
        }
    }
}
return FirstResAddr;
}
```

第二步也使用一个自定义函数来完成。通过这个函数将特定类型的资源移动到指定的位置。资源类型的指定是根据资源 ID 号来确定的，Icon 的 ID 号为 03h，Group Icon 为 0Eh，Version Information 为 10h。只要在入口参数中分别指定不同的 ID 号，就可移动不同类型的资源。为移动三种不同的资源，此函数将被执行三次。函数代码中涉及外壳结构的部分请参考图 16.1。函数代码如下：

```
/*-------------------------------------------------------------*/
/* 将特殊类型的资源移动到指定的位置                            */
/* ResType:资源的 ID 号                                        */
/* MoveADDR: 为目标地址,如为 0，函数不移动数据，只返回数据大小 */
/* MoveResSize:为上次移动资源的大小                            */
/*-------------------------------------------------------------*/
```

```
BOOL MoveRes(UINT ResType,PCHAR MoveADDR,UINT MoveResSize)
{
PIMAGE_DATA_DIRECTORY              pResourceDir = NULL;
PIMAGE_RESOURCE_DIRECTORY          pResource = NULL;
PIMAGE_RESOURCE_DIRECTORY          pTypeRes = NULL;
PIMAGE_RESOURCE_DIRECTORY          pNameIdRes = NULL;
PIMAGE_RESOURCE_DIRECTORY          pLanguageRes = NULL;
PIMAGE_RESOURCE_DATA_ENTRY         pResData = NULL;
DWORD                              mShell0_nSize = NULL;//外壳引导段的尺寸
PCHAR                              pOffsetToDataPtr;
PIMAGE_RESOURCE_DIRECTORY_ENTRY    pTypeEntry = NULL;
PIMAGE_RESOURCE_DIRECTORY_ENTRY    pNameIdEntry = NULL;
PIMAGE_RESOURCE_DIRECTORY_ENTRY    pLanguageEntry = NULL;

.....
//计算外壳引导段尺寸
mShell0_nSize =(DWORD) (&ShellEnd0) - (DWORD)(&ShellStart0) ;
pResourceDir = &m_pntHeaders->OptionalHeader.DataDirectory[2];
if (pResourceDir->VirtualAddress == NULL)
    return FALSE;
pResource    = (PIMAGE_RESOURCE_DIRECTORY)RVAToPtr(pResourceDir-> VirtualAddress);
pTypeRes     = pResource;
nTypeNum     = pTypeRes->NumberOfIdEntries + pTypeRes-> NumberOfNamedEntries;
pTypeEntry   = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)pTypeRes + \
               sizeof(IMAGE_RESOURCE_DIRECTORY));
for (nTypeIndex = 0; nTypeIndex < nTypeNum; nTypeIndex ++, pTypeEntry ++)
{
 if(pTypeEntry->NameIsString==0)
 {
    if((DWORD)pTypeEntry->NameOffset ==ResType)
    {
        //该类型目录地址
        pNameIdRes = (PIMAGE_RESOURCE_DIRECTORY)((DWORD)pResource + \
                     (DWORD)pTypeEntry->OffsetToDirectory);
        //该类型中有几个项目
        nNameIdNum = pNameIdRes->NumberOfIdEntries+pNameIdRes->NumberOfNamedEntries;
        pNameIdEntry = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD) \
                       pNameIdRes + sizeof(IMAGE_RESOURCE_DIRECTORY));
        for (nNameIdIndex = 0; nNameIdIndex < nNameIdNum; nNameIdIndex ++,pNameIdEntry ++)
        {
            pLanguageRes = (PIMAGE_RESOURCE_DIRECTORY)((DWORD)pResource\
                           + (DWORD)pNameIdEntry->OffsetToDirectory);
            nLanguageNum = pLanguageRes->NumberOfIdEntries+pLanguageRes\
                           ->NumberOfNamedEntries;
            pLanguageEntry = (PIMAGE_RESOURCE_DIRECTORY_ENTRY)((DWORD)\
                             pLanguageRes + sizeof(IMAGE_RESOURCE_DIRECTORY));
            for (nLanguageIndex = 0; nLanguageIndex < nLanguageNum;nLanguageIndex ++,\
                 pLanguageEntry ++)
            {
              pResData = (PIMAGE_RESOURCE_DATA_ENTRY)((DWORD)pResource\
                         + (DWORD)pLanguageEntry->OffsetToData);
                if(MoveADDR)
                {
                  pOffsetToDataPtr = RVAToPtr(pResData->OffsetToData);
                  //将 OffsetToData 字段指向外壳引导的新资源处
```

```
                    pResData->OffsetToData=m_nImageSize+ mShell0_nSize+ MoveResSize;
                    memcpy(MoveADDR+MoveResSize,pOffsetToDataPtr,pResData->Size);
                    ZeroMemory(pOffsetToDataPtr, pResData->Size);
                }
                MoveResSize+=pResData->Size;
            }
        }
        return MoveResSize;
       }
      }
  }
 return 0;
}
```

在第三步中资源区块的压缩还需要分两步来做。先是将区块中不能压缩的资源目录部分写入文件，然后从资源数据的起点开始压缩资源区块的剩余部分。压缩完后将压缩得到的数据也写入文件。最后再将两部分写入的数据作为一个完整的区块，根据它的大小、偏移等情况修正文件头区块表中的相应信息。完整的代码如下，使用时只需将它们插入上一小节中省略的部分处即可。

```
/*-------------------------------------------------------------*/
/*PackFile()里处理资源的代码                                   */
/*-------------------------------------------------------------*/
PIMAGE_DATA_DIRECTORY           pResourceDir = NULL;
UINT                            nResourceDirSize = NULL;
PCHAR                           pResourcePtr = NULL;
UINT                            nResourceSize;
pResourceDir = &m_pntHeaders->OptionalHeader.DataDirectory[2];
if (pResourceDir->VirtualAddress != NULL)
{
 pResourcePtr = (PCHAR)RVAToPtr(pResourceDir->VirtualAddress);
 nResourceSize = pResourceDir->Size;

 //写入资源段不被压缩的部分
 UINT                  nFirstResSize;
 PCHAR              pFirstResADDR = NULL;
 //减区块基址后，得到不压缩部分长度
 nResourceDirSize =  FirstResADDR - pResourceDir->VirtualAddress ;
 WriteFile(hPackFile,(PCHAR)pResourcePtr,nResourceDirSize,&nbWritten,0;
 pFirstResADDR = RVAToPtr(FirstResADDR);  //待压缩资源地址
 nFirstResSize = nResourceSize-nResourceDirSize; //需要压缩资源的大小

 //压缩后的地址 m_pPackData
 //压缩后数据大小 m_nPackSize
 PackData(pFirstResADDR,nFirstResSize);
 nRawSize  = AlignSize(m_nPackSize+nResourceDirSize, nFileAlign);//对齐后的资源段大小
 WriteFile(hPackFile,(PCHAR)m_pPackData,m_nPackSize,&nbWritten,NULL); // 写入压缩后的数据
 // 写入为对齐而填充的零数据
 if (nRawSize - m_nPackSize -nResourceDirSize > 0)
    FillZero(hPackFile, nRawSize - m_nPackSize -nResourceDirSize);
 psecHeader->SizeOfRawData = nRawSize;// 修正区块的大小
 // 记录压缩过的区块信息，用于外壳在运行时解压缩
 AddPackInfo(FirstResADDR,nFirstResSize,m_nPackSize);
}
```

16.2.8　区块的融合

在 PE 文件中的各个区块的长度都必须是 FileAlignment 值的倍数，不足部分则以“00”填充，所以各个区块之间必定都有一定长度的间隙。所谓区块融合就是将一些可以压缩的区块合并为一个区块，然后进行统一处理。这样可以减少压缩后区块的个数，也就减少了区块间隙的个数，这可以减小加壳后文件的体积。要注意的是此处的融合并非是仅仅将各个区块间的空隙挤掉，将数据连接起来，那样做会导致程序中用来定位数据的地址信息变得无效，最终导致程序被破坏。要做的仅仅是将多个区块在逻辑上作为一个区块来考虑，而它们在内存中未压缩时的存储方式（各数据的 RVA）并没有丝毫的改变。此处的融合和编译、连接程序时用“/MERGE”参数进行的区块合并是完全不同的概念。

由于是按文件区块的 RVA 读入文件的，与程序运行时由系统装载器载入的情况基本相同，所以只需要修改文件头中区块表的数据就可以了。为了简化，在本例中仅仅融合最初的几个区块，具体做法是根据区块名从第一个区块（一般是代码块）开始，找到第一个不能压缩的区块，将它们之间的区块合并（包含第一个区块，但不包含最后那个不能压缩的区块），作为一个区块。合并后的区块的 RVA 为参与融合的第一个区块的 RVA，VirtualSize 为参与融合的各个区块 VirtualSize 之和。融合完成后还需将后面没有融合的区块的区块表向前移动，紧接第一个区块的区块表排列。具体代码如下：

```
/*-------------------------------------------------------------------*/
/* 将开始的一些可以压缩的区块合并,可以缩小一些压缩后文件的大小经过此函数后融  */
/* 合生成的区块,只有映像大小和映像偏移有用,文件大小和文件偏移在压缩回写时修正 */
/*-------------------------------------------------------------------*/
BOOL MergeSection()
{
  UINT                  nSectionNum = 0;
  PIMAGE_SECTION_HEADER  psecHeader  =  m_psecHeader;
  UINT                  nspareSize  = NULL;
  UINT                  nMergeVirtualSize = 0;
  nSectionNum   = m_pntHeaders->FileHeader.NumberOfSections;
  for (UINT nIndex = 0; nIndex < nSectionNum; nIndex ++, psecHeader ++)
  {
   if ((m_psecHeader->Characteristics & IMAGE_SCN_MEM_SHARED) != 0)
        break;   // 共享区块不融合
   if ((strcmp((char *)psecHeader->Name, ".edata") == 0))
        break;   // 输出表所在区块不融合
   if ((strcmp((char *)psecHeader->Name, ".rsrc") == 0))
        break;   // 资源所在区块不融合
   nMergeVirtualSize += psecHeader->Misc.VirtualSize;
  }
  m_psecHeader->Misc.VirtualSize = nMergeVirtualSize;
  m_pntHeaders->FileHeader.NumberOfSections = nSectionNum - nIndex+1;// 现在的区块数
  //将剩余的区块移前
  memcpy(m_psecHeader+1,psecHeader,(nSectionNum-nIndex)*sizeof(IMAGE_SECTION_HEADER));
  nspareSize=(nSectionNum - m_pntHeaders->FileHeader.NumberOfSections)*\
             sizeof(IMAGE_SECTION_HEADER);//多余区块长度
  memset(m_psecHeader+nSectionNum - nIndex+1,0,nspareSize);
}
```

16.3　外壳部分编写

壳和病毒在某些方面比较类似，都需要比原程序代码更早地获得控制权。壳修改了原程序的执行文件

的组织结构，从而能够比原程序的代码提前获得控制权，并且不会影响原程序的正常运行。外壳除了还原原始程序外，另一个重要功能则是阻止破解者的跟踪、脱壳。所以一般来说这段代码会比较长，里面还有各种花指令、反调试器、反 Dump 的代码。因此外壳部分是壳开发的重中之重。

16.3.1 外壳的加载过程

Windows 的 PE 加载器加载可执行程序时，首先根据输入表获取所有 API 调用的地址，并填写到 IAT 中，再重定位所有的重定位项，最后调用 WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,PSTR szCmdLine, int iCmdShow)执行；如果是 DLL，则调用 DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)。加壳后，这个加载过程就由外壳来模拟了。

这里简单说说一般壳的装载过程。EXE 和 DLL 两种方式加壳处理的基本原理是一样的，但也有区别。DLL 涉及多次进入、入口参数和返回地址的保存、重定位项处理。

1. 保存入口参数

加壳程序初始化时保存各寄存器的值，外壳执行完毕，再恢复各寄存器内容，最后再跳到原程序执行。通常用 pushad/popad、pushfd/popfd 指令对来保存与恢复现场环境。

2. 处理多次进入

由于 DLL 加载过程中，会多次进入 DllMain()，在外壳程序中的一些变量在第一次进入时，就已经初始化。所以，还要有一个记录这些变量是否已经被初始化的标志，以防止重复初始化。举个例子说明如下：假设外壳程序中有一个内存变量“test dd SomeAddr”，而 SomeAddr 是一个需要外壳重定位的地址，那么在第一次进入时，外壳将对 test 进行重定位“add test,ebp”，在该 DLL 并未卸载的情况下二次进入时，又要执行一遍“add test,ebp”，很显然，这时候的 test 的内容将是错误的。例程中使用“is_data_initialized”这个变量作为标志。

另外，再定义一个变量“S_FileIsDll”来记录当前加载的 PE 是否为一个 DLL，只有是 DLL 文件时，才进行一些特殊处理。

3. 模拟 PE 加载器完成相应的功能

外壳首先将原始数据解压出来，然后模拟 Windows 系统的 PE 加载器，恢复输入表。如果有重定位表，则对代码进行“重定位”操作。处理完跳到原始入口点，从这个时候起壳就把控制权交还给原程序了。

16.3.2 自建输入表

在正常情况下，程序通过 PE 输入表可以得到 API 函数的地址，从而使程序能够正常运行。那么当程序加壳后，外壳程序中所调用的 API 函数在执行时又是如何取得其在系统中的地址的呢？

通常加壳程序都是通过自建输入表、壳程序本身再将原输入表导入的方法使壳程序能够正常运行。

首先回顾一下关于输入表的部分内容，它的结构如下：

```
IMAGE_IMPORT_DESCRIPTOR STRUCT
    OriginalFirstThunk   dd   ?    ; 指向 IMAGE_THUNK_DATA 数组的 RVA
    TimeDateStamp        dd   ?    ; 时间日期标志
    ForwarderChain       dd   ?    ; 正向链接索引
    Name                 dd   ?    ; 指向 DLL 文件名的 RVA
    FirstThunk           dd   ?    ; 指向输入函数真实地址单元处的 RVA
IMAGE_IMPORT_DESCRIPTOR ENDS
```

自建输入表就是要将在外壳程序中用到的 API 调用，以这种结构构造出来并将所构造的输入表的 RVA，作为加壳后的 PE 输入表 RVA，写入 PE 头。下面以 KERNEL32 为例，说明实现方法如下：

```
ImportTableBegin  LABEL  DWORD
```

```
;-------------------------------------------------------------------------
; IMAGE_IMPORT_DESCRIPTOR 结构
ImportTable DD  AddressFirst-ImportTable     ; OriginalFirstThunk
            DD  0                            ; TimeDataStamp
            DD  0                            ; ForwardChain
AppImpRVA1  DD  DllName-ImportTable          ; Name
AppImpRVA2  DD  AddressFirst-ImportTable     ; FirstThunk
            DD  0,0,0,0,0                    ; 输入表结束符
;-------  ---------------------------------------------------------------
; 输入名称表（INT）
AddressFirst DD  FirstFunc-ImportTable       ; IMAGE_THUNK_DATA
AddressSecond   DD  SecondFunc-ImportTable   ; IMAGE_THUNK_DATA
AddressThird DD  ThirdFunc-ImportTable       ; IMAGE_THUNK_DATA
             DD  0                           ; 结束符
;-------------------------------------------------------------------------
DllName     DB  'KERNEL32.dll'               ; Kernel32.dll 的文件名
            DW  0                            ; 结束符
;-------------------------------------------------------------------------
; IMAGE_IMPORT_BY_NAME 结构
FirstFunc   DW  0                            ; Hint
            DB  'GetProcAddress',0           ; 函数名的 ASCII 码字符串，以 NULL 结尾
SecondFunc  DW  0                            ; Hint
            DB  'GetModuleHandleA',0         ; 函数名的 ASCII 码字符串，以 NULL 结尾
ThirdFunc   DW  0                            ; Hint
            DB  'LoadLibraryA',0             ; 函数名的 ASCII 码字符串，以 NULL 结尾
;-------------------------------------------------------------------------
ImportTableEnd  LABEL    DWORD
```

可以看出，除了 IMAGE_IMPORT_DESCRIPTOR 结构外，其他部分也是按照输入表结构中的各项构造的，如 IMAGE_IMPORT_BY_NAME 结构、输入表名称表（INT）等。

由于自建的输入表很多项目都是相对于 ImportTable 的偏移的。所以，加壳程序在加壳时，要重定位各项目是以 ImportTable 为基准的。

外壳输入表 RVA = (ImportTableBegin - ShellStart0) + 外壳程序入口的 RVA。

对自建的输入表中以 ImportTableBegin 为基准的项目都加上(ImportTableBegin-ShellStart0)，这样就完成了重定位。当 PE 装载器装入加壳后的 PE 文件时，就会根据自建的输入表，将 Kernel32 模块的相应函数地址填入上面的输入地址表处。这样在外壳程序中，通过使用 CALL[外壳入口点 VA+(AddressFirst-ShellStart0)]的方式调用 Kernel32.dll 的 GetProcAddress。

16.3.3 外壳引导段

本章加壳的主程序是用 Visual C++来编写的，但外壳部分（shell.asm）是用 32 位汇编来写的。因此在加壳时就存在一个主程序如何调用 shell.asm 的变量问题。

具体实现如下：

（1）在 32 位汇编中对供高级语言调用的变量或子程序，使用 PUBLIC 声明。

（2）在 Visual C++6.0 中调用时，使用“extern "C" DWORD para”声明。

例程中的相关程序如下：

```
//在 32 位汇编中
; 变量
PUBLIC      ShellStart0
PUBLIC      ShellEnd0
```

```
//在VC++中
extern "C" DWORD ShellStart0;
extern "C" DWORD ShellEnd0;
```

这样可以直接在 Visual C++里调用 shell.asm 变量了。

接着是编译问题，具体参考附录 B 中的“四、在 Visual C++ 工程中使用独立汇编”。

首先将 shell.asm 填加到 Visual C++工程的 Source files 中，在 Source files 中的 shell.asm 上单击右键->选择 Setting->选中 Custom Build 页。

命令行：c:\masm32\bin\ml /c /coff /Fo$(IntDir)\$(InputName).obj $(InputPath)

输出：$(IntDir)\$(InputName).obj

如果要生成调试信息，可以在命令行中加入“/Zi”参数，还可以根据需要生成 .lst 和 .sbr 文件。如果没有把 MASM 安装在 C 盘，则要做相应的修改。

在外壳代码设计时还会遇到一个数据寻址的问题。先来简单看一下程序编译时是如何进行数据寻址的。假设源程序中有一个变量名为“sum”，有一句代码为“mov eax,sum”，则编译后为“mov eax,[????????]”，其中的“????????”为变量 sum 在内存中的地址，由于程序每次载入的地址都是固定的，所以执行时不会有什么问题，但是外壳代码遇到的却不是这种情况。外壳的代码是事前设计好的，当对不同的程序加密后，执行时外壳代码所在的内存地址都是不同的，如果也像一般程序那样编写，那么在运行时就无法正确找到变量，运行就会出问题。那如何解决呢？可以先设法得到程序中某个位置的地址，然后根据那些变量相对于此位置的偏移找到变量。

举例如下：

```
call@@1
 @@1:
pop edx          ;取得@@1的地址
mov ebx,dword ptr [edx+(sum-@@1)]
.....
sum dd  0
```

在本例中通过一个 CALL 指令自动将此 CALL 下一句（@@1）的地址入栈，然后用 POP 指令取出地址放入 EDX，要取变量 sum 的值时通过“sum-@@1”取得变量相对于“@@1”的偏移，加上 EDX 就得到了变量的真实地址。这种做法在外壳中使用非常普遍，读者需要熟练掌握。

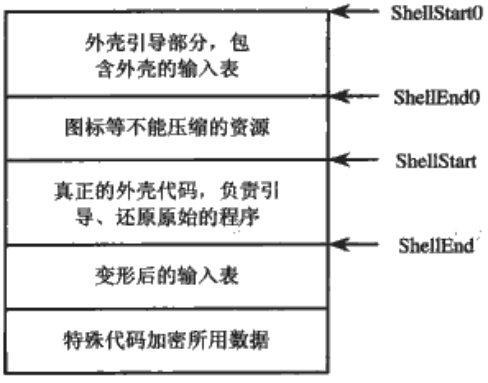


图 16.4 外壳结构图

外壳部分的结构如 16.4 所示，其中以 ShellStart 为分界，之前的部分在压缩后的程序中是以非压缩的方式存在的，之后的部分是以压缩的方式存在的。外壳执行时先执行 ShellStart0 开始的部分，这部分的主要功能是将 ShellStart 开始的真正的外壳代码在内存中解压缩，并初始化一些数据。初始化完成后转移到 ShellStart 继续执行。ShellStart 开始的代码是外壳的真正部分，它的主要功能是还原原始程序，另一个重要功能则是阻止破解者的跟踪、脱壳。所以一般来说这段代码会比较长，里面还有各种反调试器、反 Dump

的代码。将它以压缩的方式存储一方面可以减小文件尺寸，另一方面也有利于程序的安全性。下面我们就两段代码分别进行分析。

外壳第一段代码如下：

```
/*-------------------------------------------------------------*/
/*外壳第一段                                                   */
/*-------------------------------------------------------------*/
ShellStart0 LABEL   DWORD
pushad              ; 保存当前环境变量
call    next0
; 以下是自构造的外壳输入表
ImportTableBegin LABEL   DWORD
ImportTable     DD  AddressFirst-ImportTable    ;OriginalFirstThunk
                DD  0,0                         ;TimeDataStamp,ForwardChain
AppImpRVA1      DD  DllName-ImportTable         ;Name
AppImpRVA2      DD  AddressFirst-ImportTable    ;FirstThunk
                DD  0,0,0,0,0
AddressFirst    DD  FirstFunc-ImportTable       ;指向 IMAGE_tHUNK_DATA
AddressSecond   DD  SecondFunc-ImportTable      ;指向 IMAGE_tHUNK_DATA
AddressThird    DD  ThirdFunc-ImportTable       ;指向 IMAGE_tHUNK_DATA
                DD  0
DllName         DB  'KERNEL32.dll'
                DW  0
FirstFunc       DW  0
                DB  'GetProcAddress',0
SecondFunc      DW  0
                DB  'GetModuleHandleA',0
ThirdFunc       DW  0
                DB  'LoadLibraryA',0
ImportTableEnd  LABEL   DWORD
;外壳自己的重定位表（全为 0）
RelocBaseBegin  LABEL   DWORD
RelocBase       DD  0
                DD  08h
                DD  0
;以下是需要由加壳程序修正的变量
SHELL_DATA_0    LABEL   DWORD
ShellBase       DD  0           ;保存外壳压缩部分相对于 ShellStart0 的偏移
ShellPackSize   DD  0           ;保存外壳压缩部分的原始大小
TlsTable        DB  18h dup (?) ;保存原始程序的 Tls 表
;外壳引导段使用的变量空间
Virtualalloc    DB  'VirtualAlloc',0
VirtualallocADDR DD  0
Imagebase       DD  0           ;外壳基址（DLL 文件用）
ShellStep       DD  0           ;是否多次进入（DLL 文件用）
ShellBase2      DD  0           ;外壳运行后，外壳第二段的虚拟地址
;************************
next0:
pop     ebp
sub     ebp,(ImportTable-ShellStart0)     ; 取得外壳入口点地址
; 下面 6 句主要是针对 Dll 文件加壳，DLL 退出时会再走一遍的
mov     eax, dword ptr [ebp+(ShellStep-ShellStart0)]
.if eax != 0                            ; dll 文件退出时走这里
    pushebp
    jmp     dword ptr [ebp+(ShellBase2-ShellStart0)]
.endif
```

```
inc     dword ptr [ebp+(ShellStep-ShellStart0)]
; 如果是 DLL 文件，取当前映像基址；如果是 EXE，在后面会用 Getmulehandle 取基址的
mov     eax, dword ptr [esp+24h]
mov     dword ptr [ebp+(imagebase-ShellStart0)], eax
lea     esi,[ebp+(DllName-ShellStart0)] ; 指向 KERNEL32.dll 字符串
; 下面 2 句就是 GetModuleHandleA("KERNEL32.dll")
push    esi
call    dword ptr [ebp+(AddressSecond-ShellStart0)]
lea     esi,[ebp+(Virtualalloc-ShellStart0)] ; 指向 Virtualalloc 字符串
; 下面 3 句就是 GetProcAddress (eax,"VirtualAlloc ")
push    esi
push    eax
call    dword ptr [ebp+(AddressFirst-ShellStart0)] ; GetProcAddress 函数
mov     dword ptr [ebp+(VirtualallocADDR-ShellStart0)],eax
; 下面几句就是 VirtualAlloc(0, ShellPackSize,MEM_COMMIT,PAGE_READWRITE)
push    PAGE_READWRITE
push    MEM_COMMIT
push    dword ptr [ebp+(ShellPackSize-ShellStart0)]
push    0
call    dword ptr [ebp+(VirtualallocADDR-ShellStart0)] ; VirtualAlloc
push    eax     ; 调用 VirtualAlloc 分配内存的地址入栈，此值即为外壳第二段地址
;将外壳第二段地址放到 ShellBase2，dll 退出时会用到
mov     dword ptr [ebp+(ShellBase2-ShellStart0)], eax
mov     ebx,dword ptr [ebp+(ShellBase-ShellStart0)]  ; 取待解压的外壳数据
add     ebx,ebp
push    eax
push    ebx
call    _aP_depack_asm  ; 调用 Aplib 提供的解压函数，将外壳第二段解压出来
pop     edx
push    ebp             ; 保存第一段基址，以便第二段中使用
jmp     edx             ; 转到第二段继续执行
_aP_depack_asm:
…… 省略解压代码，对于不同版本的 Aplib 库，解压代码稍有不同
ShellEnd0   LABEL   DWORD
```

16.3.4 外壳第二段

第二段的主要工作是还原、初始化原程序，由于开始未对程序做特殊的处理，所以还原的工作主要就是解压缩。初始化工作主要指的就是初始化输入表，一般来说 EXE 文件载入基地址与程序的 ImageBase 都是相同的，所以重定位一般可以省略，如果要设计一个对 DLL 加壳软件的话，重定位这一步则是必不可少的。本来在第二段中应该有很多的代码用来抵抗脱壳者的攻击，这是加密的重中之重，也是判断一个外壳性能的重要依据。但是为了突出本节的重点，在本例中这些代码都没有采用，在第二段只包含了最基本的用来还原原程序的代码。如果要设计一个有一定加密强度的加壳软件，反调试、反 Dump、反监视等的代码是必不可少的，读者可以根据需要查看本书中其他章节的内容，在外壳第二段中加入相应的代码。不要怕麻烦，这种代码是多多益善的，哪怕只是消耗脱壳者的耐心，浪费他的时间也是好的。如果一个脱壳者在调试过程中不断遇到各种 ANTI 代码，稍有不慎就会导致前功尽弃，经过了数小时仍然不能看到希望的话，他多数会放弃的，那加密的目的也就达到了。不要希望通过某种简单的方法彻底阻止脱壳者的前进，任何防止调试、脱壳的方法对于有经验的解密者来说都是可以绕过的，或许用无数的代码破坏脱壳者的信心，最终拖垮脱壳者才是最彻底，也是最容易达到目的的解决方法。但是另一方面，过多 ANTI 代码的加入也会引起程序执行效率的降低。所以要编制一个优秀的加壳软件的话，应该在加密强度和执行效率之间找到一个比较好的平衡点。幸运的是，电脑硬件的飞速发展已经使得软件的执行效率不像以前那么重要了。

第二段代码如下：

```
/*-------------------------------------------------------------*/
/* 外壳的第二段代码                                            */
/*-------------------------------------------------------------*/
ShellStart LABEL DWORD
  Call   $+5
  pop    edx
  sub    edx,5h
  pop    ebp

  mov    eax, dword ptr [edx+(ShellStep_2-ShellStart)]
  .if    eax != 0     ;dll 退出时从这里进入 OEP
         popad
         jmp ReturnOEP
 .endif

  mov    ecx,3h
  lea    esi,[ebp+(AddressFirst-ShellStart0)]
  lea    edi,[edx+(GetprocaddressADDR-ShellStart)]
MoveThreeFuncAddr:
  mov    eax,dword ptr [esi]
  mov    dword ptr [edi],eax
  add    esi,4h
  add    edi,4h
  loop   MoveThreeFuncAddr ;保存外壳输入表的 3 个函数入口地址备用
  lea    eax,[ebp+(_aP_depack_asm-ShellStart0)]
  mov    dword ptr [edx+(aP_depackAddr-ShellStart)],eax;保存解压函数入口
  mov    eax,dword ptr [ebp+(VirtualallocADDR-ShellStart0)]
  mov    dword ptr [edx+(S_VirtualallocADDR-ShellStart)],eax

  mov    eax,[ebp+(imagebase-ShellStart0)]     ;将 DLL 基址读出
  mov    ebp,edx
  mov    dword ptr [ebp+(FileHandle-ShellStart)],eax
  mov    eax, dword ptr [ebp+(S_FileIsDll-ShellStart)]
  .if    eax == 0;如果是 EXE 文件，则用 GetModuleHandleA 取得其当前文件句柄
         push0
         calldword ptr [ebp+(GetmulehandleADDR-ShellStart)]
         mov dword ptr [ebp+(FileHandle-ShellStart)],eax
  .endif

  ;*******取一些函数入口
  lea    esi,dword ptr [ebp+(Ker32DllName-ShellStart)]
  push   esi
  call   dword ptr [ebp+(GetmulehandleADDR-ShellStart)]
  .if    eax==0
         pushesi
         calldword ptr [ebp+(LoadlibraryADDR-ShellStart)]
  .endif
  mov    esi,eax
  lea    ebx,dword ptr [ebp+(S_Virtualfree-ShellStart)]
  push   ebx
  push   esi
  call   dword ptr [ebp+(GetprocaddressADDR-ShellStart)]
  mov    dword ptr [ebp+(S_VirtualfreeADDR-ShellStart)],eax
  ;*******解压缩各段********
  mov    ebx,S_PackSection-ShellStart
```

```
DePackNextSection:
cmp    dword ptr [ebp+ebx],0h
jz     AllSectionDePacked
push   ebx
push   PAGE_READWRITE
push   MEM_COMMIT
push   dword ptr [ebp+ebx]
push   0
call   dword ptr [ebp+(S_VirtualallocADDR-ShellStart)];申请内存进行读写
pop    ebx
mov    esi,eax ;申请得到内存空间的基地址
mov    eax,ebx
add    eax,ebp
mov    edi,dword ptr [eax+4h] ;取得欲解压区块的 RVA
add    edi,dword ptr [ebp+(FileHandle-ShellStart)]
push   esi
push   edi
call   dword ptr [ebp+(aP_depackAddr-ShellStart)];解压
mov    ecx,dword ptr [ebp+ebx];原区块大小，即需写回的解压数据的大小
push   esi
rep    movsb ;将解压后的数据写回
pop    esi
push   ebx
push   MEM_RELEASE
push   0
push   esi
call   dword ptr [ebp+(S_VirtualfreeADDR-ShellStart)]    ;释放内存
pop    ebx
add    ebx,0ch
jmp    DePackNextSection
AllSectionDePacked:
;******初始化原始程序的输入表******
mov    eax,dword ptr [ebp+(S_IsProtImpTable-ShellStart)]
.if    eax == 0 ;为 0 表示输入表没加密，此时模拟 PE 加载器处理输入表
       mov     edi,dword ptr [ebp+(ImpTableAddr-ShellStart)] ;取输入表地址
       add     edi,dword ptr [ebp+(FileHandle-ShellStart)] ;加上载入基址
  GetNextDllFuncAddr:
       mov     esi,dword ptr [edi+0ch]          ; 指向 DLL 名字符串
       .if     esi == 0     ; 为空表示所有 DLL 初始化完成
               jmp AllDllFuncAddrGeted
       .endif
       add     esi,dword ptr [ebp+(FileHandle-ShellStart)]
       push    esi
       call    dword ptr [ebp+(GetmulehandleADDR-ShellStart)]
       .if     eax==0   ; 为 0 表示 DLL 未被载入
           Push    esi
           Call    dword ptr [ebp+(LoadlibraryADDR-ShellStart)];载入 DLL
       .endif
       mov     esi,eax  ; 保存 DLL 句柄
       mov     edx,dword ptr [edi] ; 取 OriginalFirstThunk 的值
       .if     edx == 0
               mov edx,dword ptr [edi+10h]; 为 0 则用 FirstThunk 来代替
       .endif
       add     edx,dword ptr [ebp+(FileHandle-ShellStart)]
       mov     ebx,dword ptr [edi+10h]
```

```
        add     ebx,dword ptr [ebp+(FileHandle-ShellStart)]
    GetNextFuncAddr:
        mov     eax,dword ptr [edx] ;取得一个 IMAGE_THUNK_DATA
        .if     eax == 0 ;如为 0 表示所有此 DLL 中所有函数初始化完成
                jmp     AllFuncAddrGeted
        .endif
        push    ebx
        push    edx
        cdq
        .if     edx == 0;判断 IMAGE_THUNK_DATA 最高位是否为 0
                add     eax,2h
                ;EAX 指向函数名
                add     eax,dword ptr [ebp+(FileHandle-ShellStart)]
        .else
                and     eax,7fffffffh ;EAX 为函数序号
        .endif
        push    eax ;以函数名指针或函数序号为输入
        push    esi
        ;取得函数入口
        call    dword ptr [ebp+(GetprocaddressADDR-ShellStart)]
        ;将入口地址写回 FirstThunk 指向的 IMAGE_THUNK_DATA
        mov     dword ptr [ebx],eax
        pop     edx
        pop     ebx
        add     edx,4h
        add     ebx,4h
        jmp     GetNextFuncAddr  ;准备处理下一函数
    AllFuncAddrGeted:
        add     edi,14h
        jmp     GetNextDllFuncAddr ;准备处理下一 DLL
    AllDllFuncAddrGeted:
.else
;此处为对转储后的输入表的初始化代码
    mov     edx,dword ptr [ebp+(ImpTableAddr-ShellStart)] ;取输入地址
    add     edx,ebp  ; 加上载入基址
GetNextDllFuncAddr2:
    mov     edi,dword ptr [edx] ; 指向要初始化的 IAT
    .if     edi == 0   ; 为空表示所有 DLL 初始化完成
            jmp     AllDllFuncAddrGeted2
    .endif
    add     edi,dword ptr [ebp+(FileHandle-ShellStart)] ;加上载入基址
    add     edx,5h; 指向 DLL 名字符串
    mov     esi,edx
    push    esi
    call    dword ptr [ebp+(GetmulehandleADDR-ShellStart)] ;取 DLL 句柄
    .if     eax==0 ; 为 0 表示 DLL 未载入
            push    esi
            call    dword ptr [ebp+(LoadlibraryADDR-ShellStart)];载入 DLL
    .endif
    movzx   ecx,byte ptr [esi-1]      ;取 DLL 名字符串长度
    add     esi,ecx
    mov     edx,esi
    mov     esi,eax
    inc     edx                       ;指向转储输入表中需要的初始函数数目
    mov     ecx,dword ptr [edx]       ;需要初始函数数目
```

```
    add     edx,4h                          ; 指向第一个函数名
GetNextFuncAddr2:
    push    ecx
    movzx   eax,byte ptr [edx]              ;函数名字符长
    .if     eax == 0                        ;为0表示存储的函数序号
        inc     edx                         ;指向函数序号
        push    edx
        mov     eax,dword ptr [edx]
        push    eax
        push    esi
        call    dword ptr [ebp+(GetprocaddressADDR-ShellStart)]; 取得函数入口
        mov     dword ptr [edi],eax;将函数地址填入IAT中
        pop     edx
        add     edx,4h
    .else
        inc     edx ; 指向函数名
        push    edx
        push    edx
        push    esi
        call    dword ptr [ebp+(GetprocaddressADDR-ShellStart)]; 取得函数入口
        mov     dword ptr [edi],eax  ;将函数地址填入IAT中
        pop     edx
        movzx   eax,byte ptr [edx-1] ;取该函数名长度
        add     edx,eax
    .endif
    inc     edx                             ;指向下一函数名
    add     edi,4h
    pop     ecx
    loop    GetNextFuncAddr2                ;准备处理下一函数
    jmp     GetNextDllFuncAddr2             ;准备处理下一DLL
AllDllFuncAddrGeted2:
.endif
;*************修正重定位数据*************
mov     esi, dword ptr [ebp+ (S_RelocADDR-ShellStart)] ;取原重定位表RVA
.if     esi != 0
        add     esi, dword ptr [ebp+(FileHandle-ShellStart)];加上载入基址
        mov     edi, dword ptr [ebp+(FileHandle-ShellStart)];取当前基址
        mov     ebx, edi
        ; 当前基址减去编译器链接产生的基址（记录在PE头中的ImageBase）
        sub     edi, dword ptr [ebp+(S_PeImageBase-ShellStart)]
        movzx   eax, byte ptr [esi]             ;从新重定位表结构中取数据
        .while  al
            .if al == 3h                        ;是本段重定位数据第一项吗
                inc     esi                     ;指向下一数据
                add     ebx, dword ptr [esi] ;得到需要重定位项目的地址
                add     dword ptr [ebx], edi ;进行重定位
                add     esi, 4h
            .else
                inc     esi                     ;指向下一数据
                add     ebx, eax                ;得到需要重定位项目的地址
                add     dword ptr [ebx], edi ;进行重定位
            .endif
            movzx   eax, byte ptr [esi]
        .endw
.endif
```

```
……
;*************准备返回 OEP***************
  inc         dword ptr [ebp+(ShellStep_2-ShellStart)]        ;ShellStep_2 标志加 1
  mov         eax,dword ptr [ebp+(OEP-ShellStart)]            ;取出原来的入口 RVA
  add         eax,dword ptr [ebp+(FileHandle-ShellStart)]     ;加上映像基地址
  add         dword ptr [ebp+(ReturnOEP-ShellStart)+1],eax    ;将结果放到 ReturnOEP
  popad
ReturnOEP:
  push   dword ptr[0]
  ret
; 以下是需要由加壳程序修正的变量
SHELL_DATA_1 LABEL   DWORD                    ; 标签
OEP                  DD  0                    ; 存放原始程序的入口 RVA
S_IsProtImpTable     DD  0                    ; 存放输入表是否转储标志
ImpTableAddr         DD  0                    ; 存放原始程序的输入表地址
S_FileIsDll          DD  0                    ; 存放原始是拿为 DLL 的标志
S_RelocADDR          DD  0                    ; 原始重定位地址
S_PeImageBase        DD  0                    ; 原始映像基址
S_PackSection        DB  0a0h dup (?)         ; 放入被压缩的区块信息
; 以下是外壳第二段使用的变量
GetprocaddressADDR   DD  0
GetmulehandleADDR    DD  0
LoadlibraryADDR      DD  0
S_VirtualallocADDR   DD  0
FileHandle           DD  0                    ; 存放文件句柄
aP_depackAddr        DD  0                    ; 存放 Aplib 解压函数入口地址
ShellStep_2          DD  0                    ; 存放多次进入的值（处理 DLL 外壳用）
S_VirtualfreeADDR    DD  0
Ker32DllName         DB  'KERNEL32.dll',0
S_Virtualfree        DB  'VirtualFree',0
ShellEnd LABEL       DWORD
```

16.4 将外壳部分添加至原程序

对程序处理的最后一步就是将外壳部分代码添加到原程序，一般的做法是给原程序增加一个独立的区块，将外壳的所有代码放入这个区块中。根据 16.4 可知，外壳由好几部分组成，所以在写入文件之前得先将它们装配起来。另外，原程序的一些重要数据也得保存在外壳的适当位置，比如程序原始的入口、程序的 TLS 表等。还有一点，一般程序的区块有很多在载入时是规定只能读不能写的，但是外壳要解压代码、还原程序就必须对这部分内存有可写的权限，这一般可以通过修改程序区块表中区块的属性来做到。下面就来分析程序。

```
/*-------------------------------------------------------------------*/
/* pMapOfPackRes:图标等不能压缩的资源新地址                          */
/* nNoPackResSize:图标等资源大小                                     */
/* m_pImportTable:变形输入表的地址                                   */
/* m_pImportTableSize:变形输入表的大小                               */
/*-------------------------------------------------------------------*/
BOOL DisposeShell(PCHAR pMapOfPackRes,UINT nNoPackResSize,PCHAR m_pImportTable,
UINT m_pImportTableSize ,HWND hDlg)
{
  PIMAGE_SECTION_HEADER  plastsecHeader = NULL;
  PIMAGE_DATA_DIRECTORY  pImportDir = NULL;
  PIMAGE_SECTION_HEADER  psecHeader = m_psecHeader;
```

```
    PIMAGE_DATA_DIRECTORY  pTlsDir = NULL;
    PIMAGE_TLS_DIRECTORY   pTlsDirectory = NULL;

    PCHAR                  mShell_pData = NULL;      //整个外壳申请的地址
    UINT                   mShell_nSize = NULL;      //外壳尺寸
    PCHAR                  mShell1_pData = NULL;     //外壳引导段空间
    UINT                   mShell1_nSize = NULL;     //外壳引导段尺寸
    PDWORD                 pShell0Data = NULL;
    PDWORD                 pShellData = NULL;
    ……
  // 先处理外壳第二段
    mShell1_nSize =(DWORD)(&ShellEnd) - (DWORD)(&ShellStart);//外壳第二段尺寸
    //申请大小=外壳第二段+变形输入表大小
    mShell1_pData = new CHAR[mShell1_nSize + m_pImportTableSize];
    ZeroMemory(mShell1_pData, mShell1_nSize+ m_pImportTableSize);
    memcpy(mShell1_pData, (&ShellStart), mShell1_nSize);//将外壳第二段读入缓冲
    //指向缓冲区中 shell.asm 的变量
    pShellData=(PDWORD)((DWORD)(&SHELL_DATA_1)-(DWORD)(&ShellStart)+ mShell1_pData);
    pShellData[0] =m_pntHeaders->OptionalHeader.AddressOfEntryPoint;
    pShellData[1] =IsProtImpTable;//将是否处理输入表的标志放到外壳里
    if(IsProtImpTable){
      memcpy(mShell1_pData+mShell1_nSize,m_pImportTable,m_pImportTableSize);
      pShellData[2]= mShell1_nSize;//将变形输入表的地址保存（相对外壳第二段偏移地址）
    }
    else//将原始输入表地址保存到外壳中
    {
      pImportDir=&m_pntHeaders->OptionalHeader.DataDirectory[1];
      pShellData[2]    = pImportDir->VirtualAddress;
    }
    //将是否为 DLL 的标志放进外壳中
    if(m_pntHeaders->FileHeader.Characteristics & IMAGE_FILE_DLL)
      pShellData[3] = 1;
    //将重定位地址放进外壳中
    pShellData[4] =m_pntHeaders->OptionalHeader.DataDirectory [5].Virtual Address;
    //保存原始映像基址到外壳中
    pShellData[5] = m_pntHeaders->OptionalHeader.ImageBase;
    //保存压缩块表信息到外壳中
    memcpy((PCHAR)((PBYTE)(&pShellData[6])), m_pInfoData, m_pInfoSize);
    //对外壳第二段及变形输入表进行压缩
    PackData(mShell1_pData, (mShell1_nSize+ m_pImportTableSize),hDlg);
    /******处理整个外壳段*******/
    //计算整个外壳需要尺寸
    mShell_nSize = (DWORD) (&ShellEnd0) - (DWORD)(&ShellStart0) + m_nPackSize + nNoPackResSize;
    mShell_pData = new CHAR[mShell_nSize];
    ZeroMemory(mShell_pData, mShell_nSize);
    mShell0_nSize    = (DWORD) (&ShellEnd0) - (DWORD)(&ShellStart0);
    memcpy(mShell_pData, &ShellStart0, mShell0_nSize);//将外壳引导段读入缓冲
    if(isPackRes)
    {   //将不能压缩的资源（光标等）读入缓冲
      memcpy((mShell_pData+mShell0_nSize),pMapOfPackRes,nNoPackResSize);
    }
    //将压缩后的数据读入缓冲(即将外壳的引导段与第二段合并)
    memcpy((mShell_pData+mShell0_nSize +nNoPackResSize), m_pPackData, m_nPackSize);
    /***修正外壳输入表***/
    PIMAGE_IMPORT_DESCRIPTOR   pImportDescriptor = NULL;
```

```
PIMAGE_IMPORT_DESCRIPTOR    pDescriptor = NULL;
PIMAGE_THUNK_DATA32         pFirstThunk = NULL;
psecHeader=psecHeader+ m_pntHeaders->FileHeader.NumberOfSections;//指向区块最尾端
plastsecHeader = psecHeader - 1;
nBasePoint=plastsecHeader->VirtualAddress+plastsecHeader->Misc.VirtualSize;//新区块
//ImportTableBegin 在外壳引导段偏移值
nImportTableOffset=(DWORD)(&ImportTableBegin)-(DWORD)(&ShellStart0);
pImportDescriptor = (PIMAGE_IMPORT_DESCRIPTOR)(mShell_pData+ nImportTable Offset);
nITRVA = nBasePoint +nImportTableOffset;//校正的参数
for (pDescriptor = pImportDescriptor; pDescriptor->FirstThunk != 0; pDescriptor ++)
{
 pDescriptor->OriginalFirstThunk += nITRVA;
 pDescriptor->Name += nITRVA;
 nFirstThunk = pDescriptor->FirstThunk;
 pDescriptor->FirstThunk = nFirstThunk + nITRVA;
 pFirstThunk=( PIMAGE_THUNK_DATA32)(&(mShell_pData+ nImportTableOffset)[nFirstThunk]);
 while (pFirstThunk->u1.AddressOfData != 0)
  {
      nFirstThunk = pFirstThunk->u1.Ordinal;
      pFirstThunk->u1.Ordinal = nFirstThunk + nITRVA;
      pFirstThunk ++;
  }
}
//指向缓冲区中 shell.asm 的变量
pShell0Data=(PDWORD)((DWORD)(&SHELL_DATA_0)-(DWORD)(&ShellStart0)+ mShell_pData);
pShell0Data[0] = mShell0_nSize + nNoPackResSize;
pShell0Data[1] = mShell1_nSize + m_pImportTableSize;
// 如果原来有 TLS 数据，则修正
pTlsDir=&m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS];
if (pTlsDir->VirtualAddress != NULL)
{
   PDWORD       pShellTlsTable = NULL;
   pTlsDirectory = (PIMAGE_TLS_DIRECTORY32)RVAToPtr (pTlsDir-> VirtualAddress);
   memcpy((PCHAR)(&pShell0Data[2]),pTlsDirectory, sizeof IMAGE_TLS _DIRECTORY));
   m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_TLS].VirtualAddress=\
                                    nBasePoint+(DWORD)(&TlsTable)-(DWORD)(&ShellStart0);
   m_pntHeaders->OptionalHeader.DataDirectory[9].Size = sizeof(IMAGE_TLS_DIRECTORY32);
}
nFileAlign      = m_pntHeaders->OptionalHeader.FileAlignment;
nSectionAlign   = m_pntHeaders->OptionalHeader.SectionAlignment;
nRawSize        = AlignSize(mShell_nSize, nFileAlign);
nVirtualSize    = AlignSize(mShell_nSize, nSectionAlign);
// 修正新区块信息
memset(psecHeader, 0, sizeof(IMAGE_SECTION_HEADER));
memcpy(psecHeader->Name, ".pediy", 6);
psecHeader->PointerToRawData =plastsecHeader->PointerToRawData +plastsecHeader->\
                                                          SizeOfRawData;
psecHeader->SizeOfRawData    = nRawSize;
psecHeader->VirtualAddress   = plastsecHeader->VirtualAddress +plastsecHeader->\
                                                        Misc.VirtualSize;
psecHeader->Misc.VirtualSize = nVirtualSize;
psecHeader->Characteristics  = 0xE0000040;
// 修改文件头
m_pntHeaders->FileHeader.NumberOfSections ++;
m_pntHeaders->OptionalHeader.CheckSum = 0;
```

```
    m_pntHeaders->OptionalHeader.SizeOfImage = psecHeader->VirtualAddress+psecHeader\
                                                          ->Misc.VirtualSize;
    m_pntHeaders->OptionalHeader.AddressOfEntryPoint = nBasePoint;
    m_pntHeaders->OptionalHeader.DataDirectory[1].VirtualAddress=nBasePoint + \
                                                          nImportTableOffset;
    m_pntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_ IMPORT].Size =\
                                (DWORD)(&ImportTableEnd)-(DWORD)(&ImportTableBegin);

    //如果是 DLL，将外壳的重定位指向其虚构的重定位表处
    if(m_pntHeaders->FileHeader.Characteristics & IMAGE_FILE_DLL)
    {
       m_pntHeaders->OptionalHeader.DataDirectory[5].VirtualAddress = nBasePoint + \
                                    (DWORD)(&RelocBaseBegin) -(DWORD)(&ShellStart0);
       m_pntHeaders->OptionalHeader.DataDirectory[5].Size = 0x08;
    }
    // 将外壳部分写入文件
    WriteFile(hPackFile,(PCHAR)mShell_pData,mShell_nSize,&nbWritten,NULL);
    // 写入为对齐而填充的零数据
    if (nRawSize - mShell_nSize > 0)
       FillZero(hPackFile, nRawSize - mShell_nSize);
    delete[] mShell1_pData;
    delete[] mShell_pData;
    return TRUE;
}
```

至此，一个简单的加壳软件所需的各种功能的实现方法已经基本分析完了。至于这些功能如何完整地结合起来，读者可以根据附带的源码自行学习，此处仅讲一些要注意的地方。对 PE 文件的各项处理必须要注意其先后顺序，如果不注意的话可能会影响程序的执行效果，甚至使处理后的程序无法运行。比如资源处理的三步，如果将其顺序改为二、一、三，读者可以发现加壳后的文件会比原来的大，这就是第二步先处理后，对第一步的结果产生了不应有的影响。另外，防止程序被脱壳的方法除了尽量防止程序被脱壳者跟踪、调试外，尽可能地加强原程序和外壳的联系，也是一个不错的入手点。比如上面讲到的修改输入表，在初始化时不把正确的函数入口写入输入表，而让其指向外壳中的某段程序，必须经过这段程序的“翻译”，才能找到原始的入口。这就是设置了一条外壳与原程序联系的途径。有了这条途径，每执行一个函数外壳代码就得到一次系统的控制权，如果在其中加入 ANTI 代码的话，就可以更有效地保护程序不被攻击。此外，如果所写的外壳只是针对某一个程序，还可以把原程序的一部分功能代码放到外壳中。具体做法是，可以在程序本身做一个类似于引用外部 DLL 的调用，外壳在解密过程中先将功能代码解密到临时内存中，在初始化输入表时根据特定的函数名找出这些调用，并将它们初始化为指向临时内存中的功能代码，这样只有加壳后正常运行时这些功能才能使用，一旦脱壳就会使程序功能不全。当然将外壳与原程序联系起来的方法远不止这几种，两者之间联系得越多，完整、完美的脱壳也就会越困难。此处笔者只是提出一个思路，读者可以根据 PE 文件的结构特征、欲加密文件的功能特点设计各种不同的方法，自行发挥。

第17章 虚拟机的设计[①]

虚拟机是近几年刚刚兴起的名词，这里谈到的虚拟机和 VMWare 之类的虚拟机是不同的东西，它是一种基于虚拟机的代码保护技术。准确地说，这里谈的虚拟机其实是一种解释执行系统，例如 Visual Basic 6 中的 Pcode 编译方式，并且现在的一些动态语言如 Ruby、Python、Lua 和.Net 等从某种角度讲也是解释执行的。要理解本章内容，需要对汇编指令和操作系统有一定的了解才行。

17.1 原理

虚拟机保护技术就是将基于 x86 汇编系统的可执行代码转换为字节码指令系统的代码，以达到保护原有指令不被轻易逆向和篡改，这种指令执行系统和 Intel 的 x86 指令系统并不在同一个层次上。比如说 80x86 汇编指令是在 CPU 里执行的，而字节码指令系统是通过解释指令来执行的，并且这里谈到的字节码指令执行系统是建立在 x86 指令系统上的。

字节码(Bytecode)其实就是指令执行系统定义的一套指令和数据组成的一串数据流。Java 的 JVM、.Net 或者其他动态语言的虚拟机都是靠解释字节码来执行的，但它们的字节码之间并不通用，因为每一个系统设计的字节码都是为自己使用的，并不兼容其他的系统。

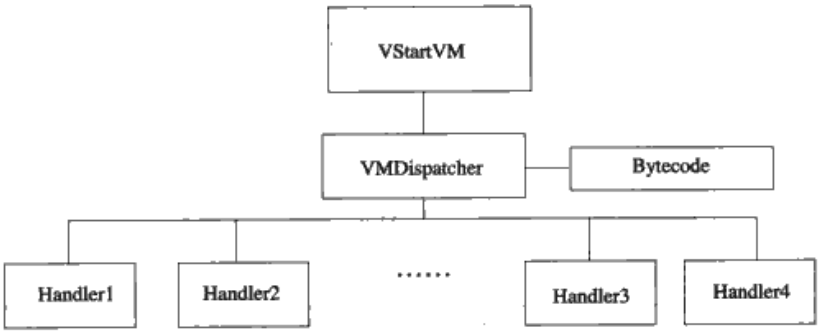


图 17.1 虚拟机图示

如图 17.1 所示是一个虚拟机执行时的整图概述，VStartVM 部分初始化虚拟机，VMDispatcher 来调度这些 Handler，如果将其看成一个 CPU 的话，Bytecode 就是 CPU 中所执行的二进制代码，VMDispatcher 就是 CPU 执行调度器，Handler 就是 CPU 中所支持的每一条指令。

---

① 本章由冯典参与编写。

17.1.1 反汇编引擎

既然要将 80x86 指令转换为字节码，那么在做其他事情之前，将 80x86 指令反汇编为可读的结构是必然的工作。

反汇编引擎网上有现成的资源供参考，本节实例采用的反汇编引擎是 OllyDbg 提供的源代码，作者是 Oleh Yuschuk，非常感谢他所做的工作。Oleh Yuschuk 所提供的源代码的函数和结构笔者已稍做添加和修改，所以请尽量查看本书光盘映像文件中的源代码。

17.1.2 指令分类

这里将需要描述的 x86 指令进行分类，按功能可以分为 4 类：普通指令、堆栈栈指令、流指令和不可模拟指令。

（1）普通指令包括算术指令、数据传输指令等；

（2）栈指令主要是 push 和 pop 等进行栈操作的指令；

（3）流指令是如 jmp, jmpc, call, retn 等会更改程序执行流程的指令；

（4）不可模拟指令，顾名思义，就是无法再次模拟的指令了，如 int3, sysenter, in, out 等，这类指令只能用其他方式处理。

按操作数可以分为：无操作数指令、单操作数指令、双操作数指令、多操作数指令。表 17-1 是分类指令表（不一定完全）。

表 17-1 指令分类

无操作数指令		单操作数指令	双操作数指令		多操作数指令
AAA	STC	CALL	ADC	XADD	IMUL
AAD	STD	JMP	ADD	XCHG	SHLD
AAM	STI	Jcc	AND	CMP	SHRD
AAS	INSB	LOOP	MOV	CMPS	
CBW	INSW	LOOPE	OR	LEA	
CDQ	INSD	LOOPNE	RCL	MOVSX	
CLD	LAHF	INC	RCR	MOVZX	
CLI	LODSB	DEC	ROL		
CLTS	LODSW	MUL	SAL		
CMC	SCASB	DIV	SAR		
CPUID	SCASW	IDIV	SHL		
CWD	SCASD	LODS	SHR		
CWDE	STOSB	NEG	SBB		
DAA	STOSW	NOT	SUB		
DAS	STOSD	SETcc	TEST		
NOP			XOR		

其中，多操作数其实可以写为双操作数的形式，所以也可以在实现时将其归类为双操作数中。

当前主流动态语言的虚拟机是基于两种模式的：Stack-Based 模式和 Regiser-Based 模式。除了 Lua5 现在已经改为 Register-Based 模式，其他语言现在还是 Stack-Based 模式。Stack-Based 模式的优点在于字节码指令长度短，用到的指令比 Register-Based 模式更少。关于它们的差别，更多的是在编译器上。本节所讲述的虚拟机稍有一些不同，其是将汇编指令编译为字节码，不过仍然可以借鉴 Stack-Based 模式的思想。

17.2 启动框架和调用约定

在讲解 Handler 之前，有必要先说一下启动框架，因为它们之间是互相调用的，并且是相辅相成的，所以它们之间需要有一种代码约定。请先看启动框架是如何设计的。

17.2.1 调度器 VStartVM

VStartVM 过程将真实环境压入后有一个 VMDispatcher 标签，当 Handler 执行完毕后会跳回到这里形成了一个循环，所以 VStartVM 过程也可以叫做 dispatcher（调度器）。

VStartVM 首先将所有寄存器的符号压入堆栈，然后 esi 指向字节码起始地址，ebp 指向真实堆栈，edi 指向 VMContext，esp 再减去 40h（这个值是可以变化的）就是 VM 用的堆栈地址了。换句话说，这里将 VM 的环境结构和堆栈都放在了当前堆栈之下 200h 处的位置上了。因为堆栈是变化的，在执行完跟堆栈有关的指令时总应该检查一下真实堆栈是否已经接近自己存放的数据了，如果是，那么再将自己的结构往更底下移动。然后从“movzx eax,byte ptr [esi]”这句开始，读字节码，读出一个字节，然后在 JUMP 表中寻找相应的 Handler，并跳转过去继续执行。具体代码如下：

```
VStartVM:
    push eax
    push ebx
    push ecx
    push edx
    push esi
    push edi
    push ebp
    pushfd
    mov     esi,[esp+0x20]                  ;参数，字节码开始的地址
    mov     ebp,esp                         ;ebp 就是堆栈了
    sub     esp,0x200
    mov     edi,esp                         ;edi 就是 VMContext
    sub     esp,0x40                        ;这时的 esp 就是 VM 用的堆栈了
VMDispatcher:
    movzx   eax,byte ptr [esi]              ;获得 Bytecode
    lea     esi,[esi+1]                     ;跳过这个字节
    jmp     dword ptr [eax*4+JUMPADDR]      ;跳到 Handler 执行处
```

调用方法：

```
push 指向字节码的起始地址
jmp  VStartVM
```

在这里看到了几个约定：

- edi　= VMContext
- esi　= 当前字节码地址
- ebp　= 真实堆栈

这是整个执行循环都要遵守的一个事实，一般情况下谁也不应该将这些寄存器另做他用。另外，edi 指向的 VMContext 存放在栈中而没有存放在其他固定地址或者申请的堆空间中，是因为考虑到多线程程序的兼容。假如有一个需要虚拟化的函数可能会被多个线程调用，而这时存放在固定的地址上就会出错，因为只能保存一个线程的环境结构。当然使用分配堆空间的 API 来为每一个线程都创建一个存放环境结构的空间也未尝不可，但虚拟机只是将汇编指令转换成虚拟指令来执行而已，使用 API 的话就会使其依赖操作系统而失去了兼容性，所以这里选择了堆栈。

17.2.2 虚拟环境：VMContext

VMContext 即虚拟环境结构，存放了一些需要用到的值：

```
struct VMContext
{
    DWORD v_eax;
    DWORD v_ebx;
    DWORD v_ecx;
    DWORD v_edx;
    DWORD v_esi;
    DWORD v_edi;
    DWORD v_ebp;
    DWORD v_efl;//符号寄存器
}
```

为什么这个环境唯独缺少了 esp 寄存器呢？因为 esp 寄存器的值已经被放在真实的 ebp 寄存器中了，VStartVM 将所有的寄存器都压入了堆栈。所以，首先应该使堆栈平衡才能开始执行真正的代码，为此，设计了一个 Handler VBegin 来做这项工作。

17.2.3 平衡堆栈：VBegin 和 VCheckEsp

平衡堆栈：VBegin 和 VCheckEsp 的代码如下：

```
vBegin:
    mov eax,dword ptr [ebp]
    mov [edi+0x1C],eax          ; v_efl
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi+0x18],eax          ; v_ebp
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi+0x14],eax          ; v_edi
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi+0x10],eax          ; v_esi
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi+0x0C],eax          ; v_edx
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi+0x08],eax          ; v_ecx
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi+0x04],eax          ; v_ebx
    add esp,4
    mov eax,dword ptr [ebp]
    mov [edi],eax               ; v_eax
    add esp,4
    add esp,4                   ; 释放参数
    jmp VMDispatcher
```

执行这个 Handler 之后，堆栈就平衡了，就可以开始继续执行真正的代码了。但是，因为将 VMContext 结构存放在当前使用的堆栈更靠下面的一部分，所以应该避免出现 VMContext 结构被覆盖的情况。

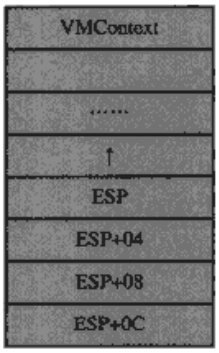


图 17.2　VMContext 环境结构在堆栈中的分布

如图 17.2 所示，当堆栈被压入数据时，总会在某条指令之后改写 VMContext 的内容，因为这个原因设计了 VCheckESP Handler。代码如下：

```
VCheckESP:
    lea     eax, dword ptr [edi+0x100]
    cmp     eax,ebp                         ;比较
    jl      VMDispatcher                    ;小于则继续执行
    mov     edx, edi                        ;否则
    mov     ecx, esp
    sub     ecx, edx
    push    esi                             ;保存 IP 指针
    mov      esi,esp
    sub      esp, 0x60
    mov      edi,esp
    push     edi                            ;保存新的 edi 基地址
    sub      esp, 0x40
    cld
    rep movsb                               ;复制
    pop edi
    pop esi
jmp VMDispatcher
```

一些可能会涉及堆栈的 Handler 在执行后跳转到 VCheckESP 判断 esp 是否接近 VMContext 所在的位置，如果是就将 VMContext 结构复制到更远的位置存放。

17.3 Handler 的设计

这里说的 Handler，并不是 Windows 中的句柄，而是一段小程序，或者说是一段过程，是由 VM 中的调度器来进行调用的。

Handler 分两大类：一类是辅助 Handler，另一类是普通 Handler。辅助 Handler 是一些更重要的、更基本的指令；普通 Handler 的功能是用来执行普通的 x86 指令的。

17.3.1　辅助 Handler

辅助 Handler 除了 VBegin 这些维护虚拟机不会导致崩溃的 Handler 之外，就是专门用来处理堆栈的 Handler 了。请看下面几个 Handler：

```
vPushReg32:
    mov     eax,dword ptr [esi]             ;从字节码中得到 VMContext 中的寄存器偏移
```

```
    add     esi,4
    mov     eax,dword ptr [edi+eax]      ;得到寄存器的值
    push    eax                          ;压入寄存器
    jmp     VMDispatcher

vPushImm32:
    mov     eax,dword ptr [esi]
    add     esi,4
    push    eax
    jmp     VMDispatcher

vPushMem32:
    mov     edx,0
    mov     ecx,0
    mov     eax,dword ptr [esp]          ;第1个寄存器偏移
    test    eax,eax
    cmovge  edx,dword ptr [edi+eax]      ;如果不是负数则赋值
    mov     eax,dword ptr [esp+4]        ;第2个寄存器偏移
    test    eax,eax
    cmovge  ecx,dword ptr [edi+eax]      ;如果不是负数则赋值
    imul    ecx,dword ptr [esp+8]        ;第2个寄存器的乘积
    add     ecx,dword ptr [esp+0x0C]     ;第三个为内存地址常量
    add     edx,ecx
    add     esp,0x10                     ;释放参数
    push    edx                          ;插入参数
    jmp     VMDispatcher

vPopReg32:
    mov     eax,dword ptr [esi]          ;得到reg偏移
    add     esi,4
    pop     dword ptr [edi+eax]          ;弹回寄存器
    jmp     VMDispatcher

vFree:
    add esp,4
    jmp     VMDispatcher
```

有了上述专门处理堆栈的 Handler 之后，就可以像图 17.3 一样设计普通 x86 指令的 Handler 了。

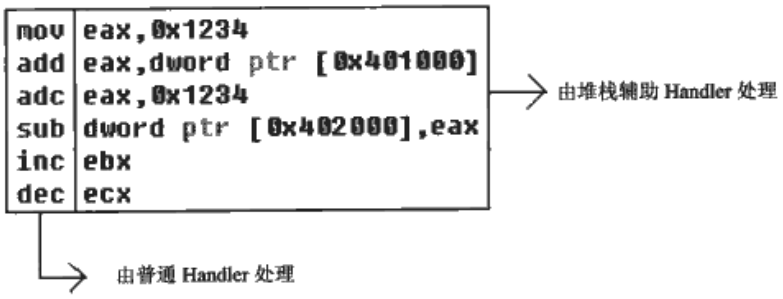


图 17.3　普通 x86 指令的 Handler

17.3.2　普通 Handler 和指令拆解

图 17.3 表达的意思是指令由普通 Handler 来处理，而源操作数和目的操作数都由堆栈 Handler 来处理。这样做的好处是，不必为指令的每一种形式都写一个模拟的 Handler。

例如，add 指令的形式通常有“add reg,imm”、“add reg,reg”、“add reg,mem”、“add mem,reg”等写法。如果将操作数都先交给堆栈 Handler 处理，那么执行到 vadd Handler 时，已经是一个立即数存放在堆栈中了，vadd Handler 不必去管它从哪里来，只需要用这个立即数做加法操作即可。

先来实现一个 vadd 的指令：

```
vadd:
    mov  eax,[esp+4]                    ;取源操作数
    mov  ebx,[esp]                      ;取目的操作数
    add  ebx,eax
    add  esp,8                          ;平衡堆栈
    push ebx                            ;压入堆栈
```

请看下面的指令是如何转换为伪代码的：

```
add esi,eax
```

转换为：

```
vPushReg32 eax_index                    ;eax 在 VMContext 中的偏移，下同
vPushReg32 esi_index
vadd
vPopReg32  esi_index
```

再来看这句：

```
add esi,1234
```

转换为：

```
vPushImm32 1234
vPushReg32 esi_index
vadd
vPopReg32  esi_index
```

下面这句转换稍微有点复杂，因为源操作数是一个内存数，而内存数的真实结构是“[imm + reg * scale + reg2]”这样的。

```
add esi,dword ptr [401000]
```

本节对 Oleh Yuschuk 反汇编引擎做了修改，使其可以得到这些信息，具体信息请参考源代码。

这行指令可以转换为：

```
vPushImm32 401000
vPushImm32 -1          ;scale
vPushImm32 -1          ;reg_index
vPushImm32 -1          ;reg2_index
vPushMem32             ;压入内存地址的值
vPushReg32 esi_index
vadd
vPopReg32  esi_index
```

这就是 add 指令的多种实现，读者可以发现无论是哪一种形式，都可以使用 vadd 来执行，只是使用了不同的堆栈 Handler，这就是 Stack-Based 虚拟机的方便之处。

17.3.3　标志位问题

标志位是一个麻烦的问题，稍有不慎就可能导致程序崩溃，并且难以调试。在 x86 中，涉及标志运算的指令有很多，如 adc,add,and,bsf,bsr,bt,btc,btr,bts,cld,cli,cmc,cmovcc,cmp, cmps,cmpxchg,cmpxchg8b,daa,das,

dec,div,idiv,imul,inc,jcc,mul,neg,not,or,rcl,rcr,rol,ror,sahf,sal,sar,shl,shr,sbb,scas,setcc,shld,shrd,stc,std,sti,sub,test,xadd 等。其中有的指令是设置标志，有的指令是判断标志，所以在相关 Handler 执行前恢复标志位，执行后保存标志位。举个简单的例子，stc 指令是将标志的 CF 位置为 1：

```
VStc:
    Push [edi+0x1C]
    Popfd
    Stc
    Pushfd
    Pop [edi+0x1C]
    jmp VMDispatcher
```

这样操作之后就能保证代码中的标志不会被虚拟机引擎所执行的代码所改变。

17.3.4 相同作用的指令

在 x86 指令集中，为了提升性能或者其他原因，可以看到一些不同的指令其实可以用同一种指令去实现。比如如下两条指令：

```
inc esi
add esi,1
```

虽然它们使用了不同的指令，但是它们的目的是相同的，这样的指令还有 sub 和 dec。另外一些位运算指令也可以相互变换，不过位运算的变换可能会涉及标志位，使标志位的结果不同，因此有的地方指令变换时需要谨慎，但不是大问题。如果将这些 x86 指令这样化简之后，那么便不用对每个指令都做一个 Handler 来描述了。

17.3.5 转移指令

转移指令有条件转移、无条件转移、call 和 retn。这里先讲解前两类转移指令。

实现时可以将 esi 指向当前字节码的地址，esi 指针就好比真实 CPU 中的 eip 寄存器，可以通过改写 esi 寄存器的值来更改流程。无条件跳转 jmp 的 Handler 比较简单：

```
vJmp:
    mov esi,dword ptr [esp]  ;[esp]指向要转到的地址
    add esp,4
    jmp VMDispatcher
```

条件转移 jcc 指令稍微有一点麻烦，因为它要通过测试标志位来判断是否需要更改流程，不过这里其实可以采取一些技巧。

读者会发现转移指令 jcc 和条件传输指令 cmovcc 高度匹配，请看表 17-2 中的一些比较。

表 17-2 同等功能指令

条件转移指令	条件传输指令
jne	cmovne
ja	cmova
jae	cmovae
jb	cmovb
jbe	cmovbe
je	cmove
jg	cmovg

基本上所有条件跳转指令都有相应的 CMOV 指令来匹配，感谢 Intel 设计出了这样一个指令，这样就好办了。这些指令可以这样设计：

```
vjne:
    cmovne esi,[esp]
add   esp,4
    jmp   VMDispatcher
vja:
    cmova  esi,[esp]
    add   esp,4
    jmp   VMDispatcher
vjae:
    cmovae esi,[esp]
    add   esp,4
    jmp   VMDispatcher
vjb:
    cmovb  esi,[esp]
    add   esp,4
jmp   VMDispatcher
    vjbe:
    cmovbe esi,[esp]
    add   esp,4
    jmp   VMDispatcher
je:
    cmove, esi,[esp]
    add   esp,4
    jmp   VMDispatcher
jg:
    cmovg  esi,[esp]
    add   esp,4
    jmp   VMDispatcher
```

字面上稍有不同的一个条件跳转指令是 jecxz，对应的指令是 cmovz。可以这样设计：

```
jecxz:
    mov   ecx,[edi+0x0c]
    test  ecx,ecx
    cmovz esi,eax
    add   esp,4
    jmp   VMDispatcher
```

17.3.6　转移跳转指令的另一种实现

上节的条件转移是利用了 cmovcc 和 jcc 指令的相似性来模拟跳转，这样做太过简单，并且太过暴露，成了虚拟机的一个鸡肋。

这里有另外一种方法来模拟跳转，请看第 4 章表 4-4 中关于 Jcc 指令的描述。条件转移是根据标志位来判断是否需要跳转，那么模拟转移指令时判断标志位就行了。如图 17.4 所示是 80x86 的标志寄存器。

图 17.4　80x86 的标志寄存器

图 17.4 描述了标志位在标志寄存器中的位置，现在需要测试的标志位和所在位置都已经知道，就可以

模拟条件跳转了。下面以 jae 指令为例：

```
vJAE:
  push [edi+0x1C]
  pop  eax
  and  eax, 1
  cmove esi,[esp]
  add  esp,4
  jmp VMDispatcher
```

这样调用这条指令：

```
vPush jumptoaddr              ;要跳转的地址
vJae
```

这个指令首先得到标志位，然后和 1 做 and 运算（取 CF 位），cmove 指令是判断 ZF 标志是否为 0，为 0 就改变 esi 指向的地址。jae 只是判断 CF 位，这里再以 jbe 做一个例子：

```
vJBE:
    push [edi+0x1C]
    pop  eax
    and  eax,0x41                ;1001B
    cmp  eax,0x41                ;Jump short if below or equal (CF=1 or ZF=1)
    cmove esi,[esp]
    add  esp,4
    jmp VMDispatcher
```

其他的跳转指令的实现也只是检测其他不同的标志位，没有太多的不同。

17.3.7 call 指令

call 和 retn 指令虽然也是转移指令，但是因为它们的功能不一样，所以被分开讲解。首先，虚拟机设计为只在一个堆栈层次上运行。请看如下代码：

```
    mov eax,1234
    push 1234
    call anotherfunc
theNext:
    add  esp,4
```

其中第 1、2、4 条指令都是在当前堆栈层次上执行的，而 call anotherfunc 是调用子函数，会将控制权移交给另外的代码，这些代码是不受虚拟机控制的。所以碰到 call 指令，必须退出虚拟机，让子函数在真实 CPU 中执行完毕后再交回给虚拟机执行下一句指令。看起来，vcall 这个 Handler 设计起来稍微有点麻烦，其实不然，call 指令先压入下一句汇编代码的地址，然后跳到目标函数。下面的代码和它等同：

```
push theNext
jmp  anotherfunc
```

如果想在退出虚拟机后让 anotherfunc 这个函数返回后再次拿回控制权，可以更改返回地址，来达到继续接管代码的操作。在一个地址写上这样的代码：

```
theNextVM:
    push theNextByteCode
    Jmp  VStartVM
```

这是一个重新进入虚拟机的代码，theNextByteCode 代表 theNext 之后的代码字节码。只需要将 theNext 的地址改为 theNextVM 的地址，即可完美地模拟 call 指令了。vcall 的伪代码如下：

```
vcall:
```

```
push  all vreg                ;所有虚拟寄存器…
pop   all reg                 ;弹出到真实寄存器中…
push  返回地址
push  要调用的函数的地址
retn
```

17.3.8　retn 指令

retn 指令和其他普通指令不一样，retn 在这里被虚拟机认为是一个退出函数。retn 有两种写法：一种是不带操作数的；另一种是带操作数的。比如：

```
retn
retn 4
```

第一种 retn 形式先得到当前 esp 中存放的返回地址，然后再释放返回地址的堆栈并跳转到返回地址；第二种比前一种多了一个步骤，即在释放返回地址的堆栈时再释放操作数的空间。vRetn 的 Handler 设计如下：

```
vRetn:
xor eax,eax
mov ax,word ptr [esi]          ;retn 的操作数是 WORD 型的，所以最大只有 0xFFFF
add esi,2
mov ebx,dword ptr [ebp]        ;得到要返回的地址
add ebp,4          ;释放空间
add ebp,eax                    ;如果有操作数，同样释放
push ebx                       ;压入返回地址
push ebp                       ;压入堆栈指针
push [edi+0x1c]
push [edi+0x18]
push [edi+0x14]
push [edi+0x10]
push [edi+0x0c]
push [edi+0x08]
push [edi+0x04]
push [edi]
pop  eax
pop  ebx
pop  ecx
pop  edx
pop  esi
pop  edi
pop  ebp
popfd
pop  esp                       ;还原堆栈指针到 esp 中，而 VM_Context 也算是自动销毁了
retn
```

17.3.9　不可模拟指令

不可模拟的指令前面也有提及，在这里任何不能识别的指令都可将其划分为不可模拟指令，碰到这类指令时，只能与 vcall 使用一种方法，即先退出虚拟机，执行这个指令，然后再压入下一个字节码的地址，重新进入虚拟机。

17.4 托管代码的异常处理

如果只是通过模拟跳转指令就想控制程序的流程是不够的，因为还有一种会打乱流程的情况，那就是异常处理。因此必须挟持原有的异常处理，才能绝对地控制程序的流程执行。关于编译器级的 SEH 更详细的资料，请参考其他文献。异常处理是不太可能完美解决的，只能针对编译器来进行模拟。

17.4.1 VC++的异常处理

VC 编译器已经将 Win32 异常处理封装。如图 17.5 所示是 VC 7 编译器生成的栈帧布局。Scopetable 是一个记录（record）的数组，每个 record 描述了一个__try 块，以及块之间的关系。

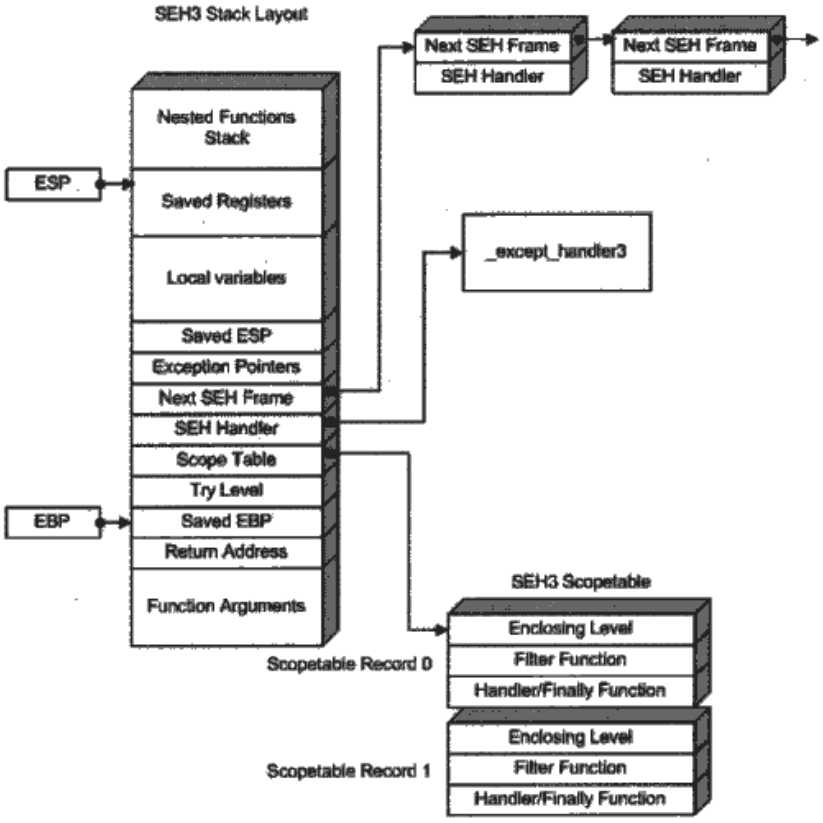


图 17.5 SEH3 Stack Layout

Scopetable 的结构如下：

```
struct _SCOPETABLE_ENTRY
{
  DWORD EnclosingLevel;
  void* FilterFunc;
  void* HandlerFunc;
}
```

MSVC 2005 的编译器为 SEH 帧增加了一些缓冲区溢出保护。完整的栈帧布局如图 17.6 所示。

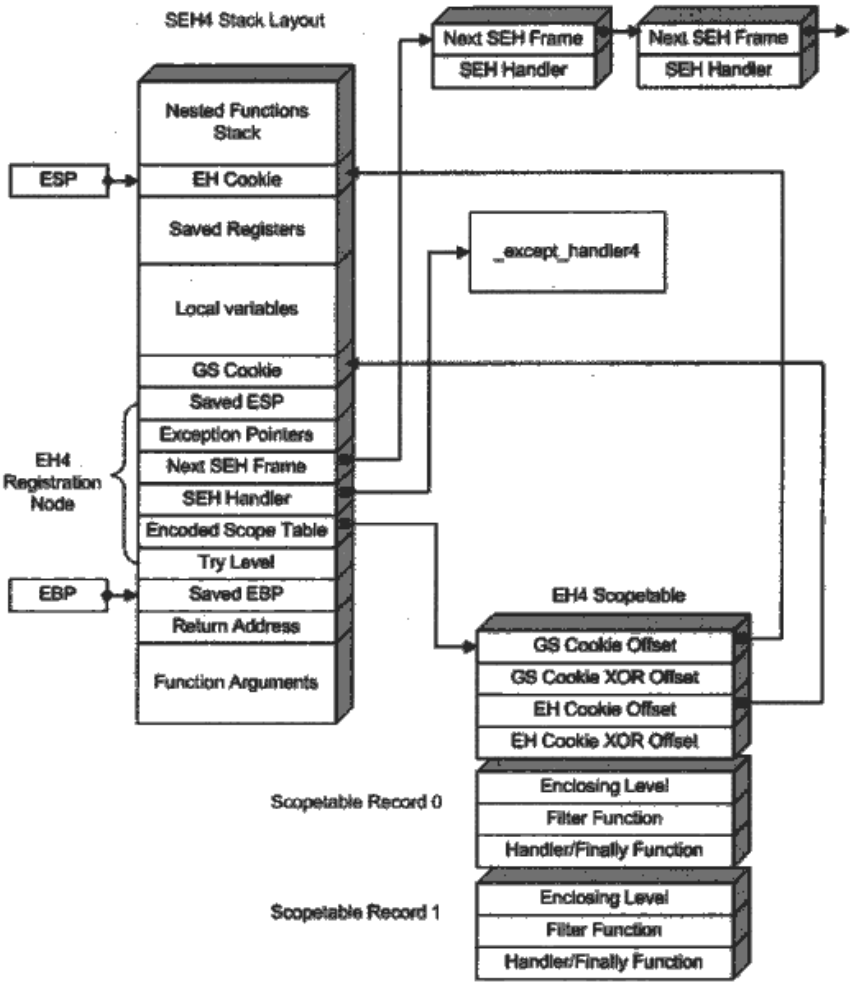


图 17.6　SEH4 Stack Layout

SEH4 Scopetable 基本上和 SEH3 一样，只是加了一个 Cookie 头。没有什么不同，只要找对 Scopetable 的偏移就行了。

```
struct _EH4_SCOPETABLE
{
  DWORD GSCookieOffset;
  DWORD GSCookieXOROffset;
  DWORD EHCookieOffset;
  DWORD EHCookieXOROffset;
  _EH4_SCOPETABLE_RECORD ScopeRecord[1];
};
struct _EH4_SCOPETABLE_RECORD
{
  DWORD EnclosingLevel;
  long (*FilterFunc)();
  union
  {
    void (*HandlerFunc)();
    void (*FinallyFunc)();
  };
};
```

请看下面这样一个代码例子：

```
void func1(char* str)
{
  char buf[12];
  __try // try block 0
  {
    __try // try block 1
    {
      *(int*)123=456;
    }
    __except(GetExceptCode() == EXCEPTION_ACCESS_VIOLATION)
    {
     printf("Access violation");
    }
    strcpy(buf,str);
  }
  __finally
  {
    puts("in finally");
  }
}
```

下面是对应的反汇编代码：

```
func1          proc near

_excCode         = dword ptr -28h
buf              = byte ptr -24h
_saved_esp       = dword ptr -18h
_exception_info  = dword ptr -14h
_next            = dword ptr -10h
_handler         = dword ptr -0Ch
_scopetable      = dword ptr -8
_trylevel        = dword ptr -4
str              = dword ptr  8

  push    ebp
  mov     ebp, esp
  push    -1
  push    offset _func1_scopetable
  push    offset _except_handler3
  mov     eax, large fs:0
  push    eax
  mov     large fs:0, esp
  add     esp, -18h
  push    ebx
  push    esi
  push    edi

  ; --- end of prolog ---
```

```
    mov     [ebp+_trylevel], 0      ;trylevel -1 -> 0: beginning of try block 0
    mov     [ebp+_trylevel], 1  ;trylevel 0 -> 1: beginning of try block 1
    mov     large dword ptr ds:123, 456
    mov     [ebp+_trylevel], 0      ;trylevel 1 -> 0: end of try block 1
    jmp     short _endoftry1

_func1_filter1:                     ; __except() filter of try block 1
    mov     ecx, [ebp+_exception_info]
    mov     edx, [ecx+EXCEPTION_POINTERS.ExceptionRecord]
    mov     eax, [edx+EXCEPTION_RECORD.ExceptionCode]
    mov     [ebp+_excCode], eax
    mov     ecx, [ebp+_excCode]
    xor     eax, eax
    cmp     ecx, EXCEPTION_ACCESS_VIOLATION
    setz    al
    retn

_func1_handler1:                ; beginning of handler for try block 1
    mov     esp, [ebp+_saved_esp]
    push    offset aAccessViolatio      ; "Access violation"
    call    _printf
    add     esp, 4
    mov     [ebp+_trylevel], 0      ;trylevel 1 -> 0: end of try block 1

_endoftry1:
    mov     edx, [ebp+str]
    push    edx
    lea     eax, [ebp+buf]
    push    eax
    call    _strcpy
    add     esp, 8
    mov     [ebp+_trylevel], -1      ; trylevel 0 -> -1: end of try block 0
    call    _func1_handler0          ; execute __finally of try block 0
    jmp     short _endoftry0

_func1_handler0:                     ; __finally handler of try block 0
    push    offset aInFinally ; "in finally"
    call    _puts
    add     esp, 4
    retn

_endoftry0:
    ; --- epilog ---
    mov     ecx, [ebp+_next]
    mov     large fs:0, ecx
    pop     edi
    pop     esi
    pop     ebx
    mov     esp, ebp
    pop     ebp
    retn
```

```
func1          endp

_func1_scopetable
  ;try block 0
  dd -1                          ;EnclosingLevel
  dd 0                           ;FilterFunc
  dd offset _func1_handler0      ;HandlerFunc

  ;try block 1
  dd 0                           ;EnclosingLevel
  dd offset _func1_filter1           ;FilterFunc
  dd offset _func1_handler1      ;HandlerFunc
```

func1 的汇编代码的第一句到“mov large fs:0,esp”这一句组成了这样一个结构：

```
typedef struct
{
    _EH3_EXCEPTION_REGISTRATION* pPrev;    //上一级节点
    EXCE_HANDLER ExceptionHandler;         //各种版本的_except_handler
    _SCOPETABLE_ENTRY* pScopeTable;        //指向一个_SCOPETABLE_ENTRY 数组
    DWORD TryLevel;                        //指示当前指令在 Try 的层次级别
} _EH3_EXCEPTION_REGISTRATION;
```

其中，执行“push -1”之后，[ebp-4] = –1，也就是 TryLevel = –1，代表未进入 try 块，这里有一个问题，那就是 Scopetable 这个数组没有数量，即不知道到底有多少个__try 块。

```
mov     [ebp+_trylevel], 0 ;trylevel -1 -> 0: beginning of try block 0
mov     [ebp+_trylevel], 1 ;trylevel 0 -> 1: beginning of try block 1
```

上面两句则代表进入 try block 0 后，又进入了 try block 1，当出现异常后，ExceptionHandler 处理程序被执行，然后 ExceptionHandler 处理程序通过 trylevel 找到指向的 try 块在 pScopeTable 数组中搜索异常处理程序，即 pScopeTable[trylevel].FilterFunc 或 pScopeTable[trylevel].HandlerFunc。现在好办了，知道了 pScopeTable 数组之后，就可以得到每一个异常处理程序的真实地址了。但还有一个小问题，现在并不知道 pScopeTable 数组有多少项，或者说并不知道有多少个 try block。有两种方法来得到数组大小。第一种：暴力搜索 pScopeTable，一直找到后面有一项的 FilterFunc 和 HandlerFunc 都为错误的地址时，就可以确定数组大小了；第二种：使用_trylevel 的某种特征，比如通常情况下为–1（SEH3），通常所在的位置在 ebp-4 处，也可以通过计算异常代码和堆栈位置相互的关系来确定_trylevel 的堆栈位置，找出所有对其赋值的常数，最大的那个常数应该就是数组的大小了。第一种简单有效，第二种比较复杂，且都不一定可靠，也不一定只有这两种办法，所谓八仙过海，各显神通了。

找到了_func1_scopetable 数组中所有异常处理函数的地址后，就可以为每一个异常处理函数生成一个托管过程的代码，比如为 func1 FilterFunc 生成一个托管代码如下：

```
func1_FilterFunc_Stub:
  push FilterFunc_ByteCode_Addr          ;过滤函数的字节码地址
  Jmp  StartVM
```

然后将 scopetable->FilterFunc 的地址替换为 func1_FilterFunc_Stub 的地址，当出现异常时就会被调用，这时就再次进入了虚拟机，执行 FilterFunc 的字节码了。HandlerFunc 也是一样。

17.4.2 Delphi 的异常处理

Delphi 的异常处理也经过了封装，描述其内部构造的文献很少，本节所述内容并未找到权威文献加以

证明，请读者用怀疑的眼光看本节。

请看下面一段 Pascal 代码：

```
try
  asm
    xor ecx,ecx
    idiv ecx
  end;
  MessageBox(0,'这里过了异常,但这里永远不会到','1111',0);
except
  MessageBox(0,'这里是异常处理函数','2222',0);
end;
```

相应的汇编代码：

```
        push    ebp                     ; TForm1.Button1Click
        mov     ebp, esp
        push    ebx
        push    esi
        push    edi
        xor     eax, eax
        push    ebp
        push    044FBD5h                ;044FBD5h : jmp    @HandleAnyException
        push    dword ptr fs:[eax]
        mov     dword ptr fs:[eax], esp
        xor     ecx, ecx
        idiv    ecx
        push    0
        push    044FBF8h                ; ASCII "1111"
        push    044FC00h
        push    0
        call    MessageBox                  ;<= Jump/Call Address Not Resolved
        xor     eax, eax
        pop     edx
        pop     ecx
        pop     ecx
        mov     dword ptr fs:[eax], edx
        jmp     @Project1_0044FBF2
        jmp     @HandleAnyException     ;<= Jump/Call Address Not Resolved
        push    0
        push    044FC20h                ; ASCII "2222"
        push    044FC28h
        push    0
        call    MessageBox                  ;<= Jump/Call Address Not Resolved
        call    @DoneExcept                 ;<= Jump/Call Address Not Resolved
@Project1_0044FBF2:
        pop     edi
        pop     esi
        pop     ebx
        pop     ebp
        retn
```

可以发现，Delphi 所封装的异常处理不像 VC 那样带有一个数据结构，而是一句跳转指令跳转到了@HandleAnyException。

请看@HandleAnyException 的代码：

```
@HandleAnyException:
        mov     eax, dword ptr [esp+4]      ; @HandleAnyException
        test    dword ptr [eax+4], 6
        jnz     CurrencyFormat
        cmp     dword ptr [eax], 0EEDFADEh
        mov     edx, dword ptr [eax+018h]
        mov     ecx, dword ptr [eax+014h]
        je      @Project1_004035ED
        cld
        call    RaiseExceptionProc          ;<= Jump/Call Address Not Resolved
        mov     edx, dword ptr [0452010h]   ; <Project1.GetExceptionObject>
        test    edx, edx
        je      CurrencyFormat
        call    edx
        test    eax, eax
        je      CurrencyFormat
        mov     edx, dword ptr [esp+0Ch]
        mov     ecx, dword ptr [esp+4]
        cmp     dword ptr [ecx], 0EEFFACEh
        je      blockDescFreeList
        call    NotifyNonDelphiException    ;<= Jump/Call Address Not Resolved
        cmp     byte ptr [045002Ch], 0
        jbe     blockDescFreeList
        cmp     byte ptr [0450028h], 0
        ja      blockDescFreeList
        lea     ecx, dword ptr [esp+4]      ; heapErrorCode
        push    eax                         ; heapLock
        push    ecx
        call    UnhandledExceptionFilter    ;<= Jump/Call Address Not Resolved
        cmp     eax, 0
        pop     eax
        je      CurrencyFormat
        mov     edx, eax
        mov     eax, dword ptr [esp+4]
        mov     ecx, dword ptr [eax+0Ch]
        jmp     rover

blockDescFreeList:
        mov     edx, eax                    ; blockDescFreeList
        mov     eax, dword ptr [esp+4]
        mov     ecx, dword ptr [eax+0Ch]

@Project1_004035ED:
        cmp     byte ptr [045002Ch], 1
        jbe     rover
        cmp     byte ptr [0450028h], 0
        ja      rover
```

```
        push    eax
        lea     eax, dword ptr [esp+8]
        push    edx
        push    ecx
        push    eax
        call    UnhandledExceptionFilter    ;<= Jump/Call Address Not Resolved
        cmp     eax, 0
        pop     ecx
        pop     edx
        pop     eax
        je      CurrencyFormat

rover:
        or      dword ptr [eax+4], 2        ; rover
        push    ebx                         ; remBytes
        xor     ebx, ebx
        push    esi
        push    edi                         ; curAlloc
        push    ebp
        mov     ebx, dword ptr fs:[ebx]
        push    ebx
        push    eax
        push    edx
        push    ecx                         ; committedRoot
        mov     edx, dword ptr [esp+028h]
        push    0
        push    eax
        push    0403638h
        push    edx
        call    dword ptr [0452018h]        ; <Project1.DefLongDayNames>
        mov     edi, dword ptr [esp+028h]
        call    @GetTls                     ;<= Jump/Call Address Not Resolved
        push    dword ptr [eax]
        mov     dword ptr [eax], esp
        mov     ebp, dword ptr [edi+8]
        mov     ebx, dword ptr [edi+4]
        mov     dword ptr [edi+4], HInstance
        add     ebx, 5
        call    NotifyAnyExcept             ;<= Jump/Call Address Not Resolved
        jmp     ebx                         ;注意这里
        jmp     ChangeAnyProc               ;<= Jump/Call Address Not Resolved
        call    @GetTls                     ;<= Jump/Call Address Not Resolved
        mov     ecx, dword ptr [eax]
        mov     edx, dword ptr [ecx]        ; .3
        mov     dword ptr [eax], edx
        mov     eax, dword ptr [ecx+8]      ; .1
        jmp     Free                        ;<= Jump/Call Address Not Resolved

CurrencyFormat:
        mov     eax, 1                      ; CurrencyFormat
        retn                                ; DateSeparator
```

一般情况下会执行到这里：

```
mov     ebx, dword ptr [edi+4]              ;取出 Win32 异常处理程序地址，即 044FBD5h
mov     dword ptr [edi+4], HInstance
add     ebx, 5                              ;跳过这条语句
call    NotifyAnyExcept                     ;<= Jump/Call Address Not Resolved
jmp     ebx                                 ;跳转到 Delphi 异常处理程序
```

由此可以发现，Delphi 的 SEH 异常封装似乎与 VC 的异常封装相比要简单得多，但是为了虚拟机代码要绕过这个封装却比 VC 稍微烦琐一点。既然 HandleAnyException 执行后跳转到 Win32 异常处理程序地址加 5 的位置（即下一句指令），那么可以生成一个下面这样的托管代码。

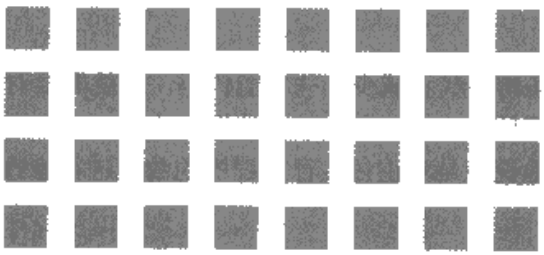
```
Except_Handler_Stub:
Jmp  @HandleAnyException
push Except_HandlerByteCode                 ; Except_Handler 的字节码地址
Jmp  StartVM
```

这一节主要描述了如何模拟 VC 和 Delphi 的异常处理机制，其他高级语言对异常处理的挟持也大同小异。

17 小结

这个虚拟机是将汇编指令转换成字节码来模拟执行的，因为汇编指令和字节码之间的特性不同，使得并不能完美地模拟汇编指令，这也是为什么无法将直接用汇编写的比较具有技巧性的代码成功转换为字节码执行的原因。另外，还有一些指令，比如 jmp eax，这种代码比较模糊，并不确定要跳转到哪个地址。碰到这种指令，的确没有什么好办法，但是好在高级语言编译器似乎也不会编译出这种代码。

本章主要对虚拟机的大概框架、Handler 设计、指令拆解和异常处理挟持作了详细的描述，而对于如何使用高级语言去实现并未作过多讲解，因为关于代码的设计不属于本章的范畴。本章附带源代码是笔者为参加看雪论坛 2007 年的 CrackMe&ReserveMe 大赛而匆忙设计的一个虚拟机，它还缺少很多东西，如异常处理机制等，只能算是一个业余的作品，读者可以研究这个源代码以设计出一个更灵活的、更强大的虚拟机。



第 8 篇 PEDIY 篇

“补丁技术”介绍了文件补丁和内存补丁技术，同时重点讲解了 SMC 技术在补丁方面的应用。学习补丁是一件很有意思的事情。

“代码的二次开发”主要是讲述如何在没有源码和无接口的情况下扩充可执行文件的功能，这一技术非常的实用。

第18章 补丁技术

在逆向工程中，经常需要改变程序原有的执行流程，使其增加或者减少一些功能代码，这就需要对原文件进行补丁。补丁分文件补丁和内存补丁两种。文件补丁就是修改文件本身某个数据，达到一劳永逸的效果。顾名思义，内存补丁是在内存中打补丁、修改，确切地说，是对正在运行的程序的数据进行修改，以达到某种效果。

18.1 文件补丁

文件补丁直接修改可执行文件或其功能模块的二进制数据，使其满足需求。实现起来很容易，对修改前和修改后的目标程序代码进行比较，把不同之处记录下来，然后写一个程序实现补丁功能。其优点如下：

（1）理论简单，容易实施。只需通过基本的十六进制数的操作就可完成任务。

（2）有很多现成的补丁工具供选择，填入几个参数，就可以做出自己的补丁器。

以第5章的EnableMenu为例讲述文件补丁的制作，需要修改代码如下：

原指令	004011E3 6A01	push 00000001
修改后指令	004011E3 6A00	push 00000000

程序的流程如下：

- 使用CreateFileA函数打开目标程序。
- 检查打开的文件是不是目标文件。可以检查文件大小，或者随机检查一些字节。
- 使用SetFilePointer函数把文件指针移动到指定位置。
- 使用WriteFile函数将数据写进磁盘文件中。

补丁源代码主体部分如下：

```
BOOL Patch()
{
    HANDLE hFile;
    DWORD szTemp;
    DWORD FileSize;
    DWORD lFileSize=40960;              //文件大小
    DWORD lFileOffset=0x11E3;           //需要修改的文件偏移地址
    DWORD lChanges=2;                   //需要修补的字节数
    BYTE BytesToWrite[]={0x6A,00};      //写的数据

    hFile = CreateFile( szFileName,0xC0000000,3,NULL,3,FILE_ATTRIBUTE_NORMAL,NULL);
```

```
    if( hFile != INVALID_HANDLE_VALUE )
    {
        FileSize=GetFileSize(hFile,&szTemp);  // 获取文件大小
        if (FileSize == 0xFFFFFFFF) {
            CloseHandle(hFile);
            return FALSE;
        }
        if(FileSize!=lFileSize){
            CloseHandle(hFile);
            return FALSE;
        }
        SetFilePointer(hFile,lFileOffset,NULL,FILE_BEGIN);//设置文件指针
        if(!WriteFile(hFile,&BytesToWrite,lChanges,&szTemp,NULL))
        {
            CloseHandle(hFile);
            return FALSE;
        }
        CloseHandle(hFile);
        return TRUE;
    }
    return FALSE;

}
```

18.4 内存补丁

文件补丁虽然实现简单，但也有其局限性，如果待补丁文件被加壳，压缩或有完整性校验，则补丁不能顺利应用。正因为有了如此的一些不便之处，所以，在文件补丁技术以外，还需要一种更加高阶、隐蔽的方法，也就是内存补丁。

内存补丁的总体思想就是在某个时刻（解压、校验或某种情况发生以后），在目标程序的地址空间中修改数据，因此也被称为 Loader，每次使用时都需要调用程序运行。

下面将分别详细论述这几种内存补丁的制作方法，并提供尽可能多的源代码供读者参考。

18.2.1 跨进程内存存取机制

就操作系统的设计原则而言，运行于同一个系统内的进程 A 和进程 B 之间应该是“绝缘”的，也就是说，一个进程的地址、指令、内存使用等信息对于另一个进程而言应该是完全透明的。说得更高级一点，操作系统应该对运行于其内部所有的进程进行“封装”。可是在软件的实践过程中，“封装”和“灵活”从来都是一对不可调和的矛盾体。在 C++语言中，为了达到这两者的平衡，引入了 friend 关键字，引入了 public、private、protected 等存取层级概念。同样的，对于操作系统而言，在对进程进行封装的同时，也需要提供一种在一定条件下可以实现的进程间互访的机制。作为最简单的“进程互访”动作，内存的存取是一个最基本也是最重要的功能。所以 Windows 提供了两个用于进程间互访内存的函数 ReadProcessMemory、WriteProcessMemory，只要进程的足够权限打开以后，就可读写进程的地址空间，权限可以用 VirtualProtect 函数设置。

在整个运行过程中，Loader 载入待补丁程序可以由 CreateProcess 函数完成，资源释放、清理工作可以由 ExitProcess 函数完成。整个流程的核心之处在于流程中间的 Loader 对创建出来的待补丁进程的控制和修改，其将要补丁的代码通过 WriteProcessMemory 函数写入目标进程适当的位置，其运行过程如图 18.1 所示。

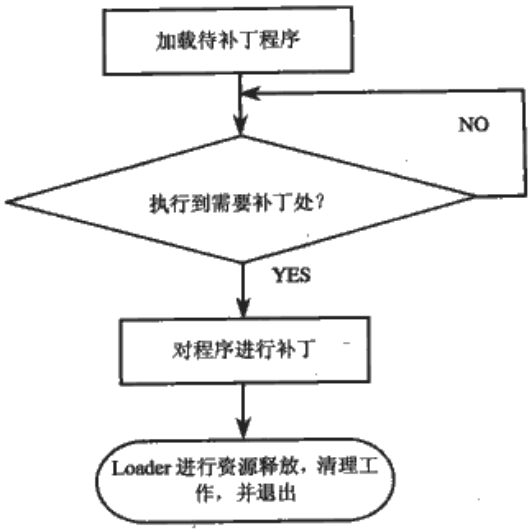


图 18.1　内存补丁执行流程图

以第 2 章的 TraceMe 演示内存补丁制作，用 ASProtect 为 TraceMe 加壳，加壳后的程序命名为“TraceMe_asp.exe”。通过调试得知，只要将 4011F5h 的判断改为 NOP 指令，该实例输入任何姓名及序列号皆可注册成功。

```
004011F5   74 37        je      short 0040122E
```

修改为：

```
004011F5     90         nop
004011F6     90         nop
```

由于 TraceMe_asp 加壳保护，不能直接修改。此处给出一段补丁程序的代码，首先创建一个进程，然后定时把目标进程挂起，读取目标进程的代码是到了补丁时机；否则继续让目标程序运行，运行一段时间再挂起判断，直到目标进程解码。读者也可以用 FindWindow 查找窗口名来判断是否解码。整个代码框架如下：

```
#define PATCH_ADDRESS    0x4011F5              //目标进程要补丁的地址
char   szFileName[20]={".\\TraceMe_asp.exe"}; //目标文件名
BYTE   TarGetData[]={0x74,0x37};               //补丁前的代码数据
BYTE   WriteData[]={0x90,0x90};                //需要补丁的数据
BYTE   ReadBuffer[128]={0};
BOOL   bContinueRun=TRUE;
DWORD Oldpp;
STARTUPINFOA          si;
PROCESS_INFORMATION  pi;

//创建一个挂起进程
if( !CreateProcess(szFileName,0,0,0,0,CREATE_SUSPENDED,0,0,&si,&pi )){
      MessageBox(NULL, "CreateProcess Failed.", "ERROR", MB_OK);
      return FALSE;
   }
while (bContinueRun) {
    ResumeThread(pi.hThread);
    Sleep(10);                          //让目标程序运行 10ms
    SuspendThread(pi.hThread);          //再挂起目标程序的进程，查看是否已解码
    ReadProcessMemory(pi.hProcess,(LPVOID)PATCH_ADDRESS,&ReadBuffer,2, NULL);
    //判断是不是完全解码出来了
```

```
        if( !memcmp(TarGetData,ReadBuffer, PATCH_SIZE) ){
            VirtualProtectEx(pi.hProcess, (LPVOID)PATCH_ADDRESS,2, 0x40, &Oldpp);
            WriteProcessMemory(pi.hProcess, (LPVOID)PATCH_ADDRESS, &WriteData,2,0);
            ResumeThread(pi.hThread);
            bContinueRun=FALSE;
        }
    }

    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
    return 0;
```

在上面的代码段中，ReadProcessMemory 函数的作用是读取目标进程特定地址的内容，有了这些内容，就可以对目标进程做一些校验性的工作。因为补丁这种程序是与程序的代码精确对应的，所以，WriteProcessMemory 之前，对目标进程进行校验是非常必要的！

读出来的内容通过校验后，就可以进入下一步的正式补丁工作了，在这把补丁数据放到 WriteData 指向的内存中，然后调用 WriteProcessMemory 函数将这些数据写入指定的地址。

这个方法的缺点很明显，在这个例子中，要补丁的地址是 4011F5h，可是在这似乎没有比较完善的方法能够让目标进程在运行到这个地址之前完成补丁工作。只有一个“勉强而且丑陋”的方法就是 CreateProcess 后，马上 Suspend 目标进程，然后等补丁完成后，再进行 Resume 的操作。

以上的补丁方法显然不能满足要求。理想的状况是需要一种“通知机制”，也就是让程序当 EIP = 4011F5h 时，能够给调试进程发送一个消息，调试进程收到消息后，再进行补丁操作。更美好的状态是：被调试进程在发送过来的消息中，不仅包含了 EIP 的内容，还能包含其他一些感兴趣的内容。很显然，要实现这种“通知机制”光凭自己的努力是不够的，还需要操作系统提供一些更底层的支持。下面就看看 Windows 提供给的 Debug API 能够在这个上面发挥什么样的功用。

18.2.2 Debug API 机制

本节需要一些 Debug API 基础，读者可以通过其他渠道获取这方面知识。要用 Debug API 实现上节所说的“通知机制”，需要利用 Debug API 的 EXCEPTION_DEBUG_EVENT 调试事件。该事件的运作机理是这样的：

① 进程 A 内部产生异常；

② 操作系统监测到 A 的异常情况，并将该异常的相关情况包装到 EXCEPTION_RECORD 结构中；

③ 查找正在对进程 A 进行调试的进程，并将第②步包装好的结构通过 EXCEPTION_DEBUG_EVENT 消息发送到查找到的进程。

④ 陷入循环的调试进程收到进程 A 的 EXCEPTION_DEBUG_EVENT 调试信息，调试进程解析调试信号所包含的信息，并进行相应的操作。

补丁时，需要在目标进程执行到该地址的时候，发送一个 EXCEPTION_DEBUG_EVENT 调试消息给调试进程，以使其能够对此做出相应的处理。那么如何才能够让目标进程给调试进程发送调试消息呢？方法有两个：

（1）CreateProcess 后，将目标进程设置成 Single Step 运行方式。由于目标进程每执行一条指令，都要和调试进程进行一次通信，所以该方法可以取得对目标进程的完全控制权。单步控制程序的弊端也是显而易见的：大幅度降低目标进程的运行效率。

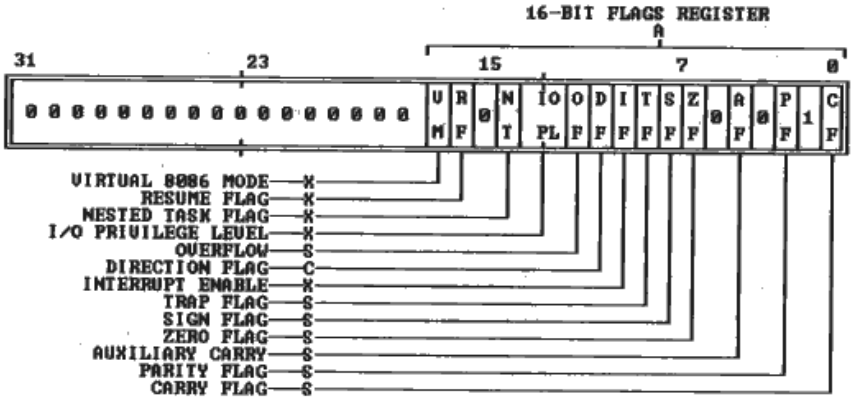
（2）修改目标进程的代码，用类似于“文件补丁”的方法，将需要发送调试信息的地址处的原指令改为 INT 3，这样当目标进程执行到特定地址时，就会因为 INT 3 的缘故，给调试进程发送相应的调试信息。

两种方法相比较，优劣很明显，当然是方法 2 比较高效和灵活。但是出于全面性的考虑，下面还是分

别将这两种方法的实现介绍给读者。毕竟，在运行时，将目标进程的执行方式设置成 Single Step 给调试进程所带来的强大威力是其他所有方法都不具备的。很多高阶的 Loader 程序，如大部分的脱壳机都要用到这项技术，所以掌握这项技术也是非常必要的。

1. Single Step 辅助机制

要使进程在 Single Step 模式下运行，需要利用 Intel CPU 的 EFLAGS 中的一个 Single Step 标志位。如图 18.2 所示是 Intel CPU EFLAGS 寄存器的示意图。



S = 状态标志（Status flag）　C = 控制标志（Control flag）　X = 系统标志（System flag）

注意：0 或 1 保留，未定义

图 18.2　Intel CPU EFLAGS 寄存器的示意图

图 18.2 中的 SF 标志位就是 Single Step 标志了。只要将其设置为 1，就能保证目标进程每执行一条指令就能与调试进程进行一次通信。将该位置位的代码可以如下所示：

```
CONTEXT          Regs ;
GetThreadContext(pi.hThread, &Regs) ;
Regs.EFlags |= 0x100 ;
SetThreadContext(pi.hThread, &Regs) ;
```

由于 VC 编译器默认的结构对齐机制已经满足了 4 字节对齐的要求，所以在 Regs 的声明处笔者并未加上多余的 align 说明。而如果读者是用 MASM 这种非常低阶的程序设计语言来编写代码的话，那么就需要在变量声明的时候加上这样的地址对齐说明伪指令。比如：

```
align   dword                                          ;设置 4 字节对齐
Regs    CONTEXT   <CONTEXT_FULL OR CONTEXT_DEBUG_REGISTERS>    ;这个结构的地址要对齐
```

目标进程处于 Single Step 状态时，每执行一条指令就会给调试进程发送 EXCEPTION_DEBUG_EVENT 调试信息，该信息的 ExceptionRecord.ExceptionCode 部分为 EXCEPTION_SINGLE_STEP。在 WaitForDebugEvent 循环中，可以监测该消息，然后进行相应的操作。需要注意的是，收到这个消息后，如果还想继续让程序 Single Step 下去，是需要重新设置一次 SF 位的。

2. INT 3 中断

INT 3 是 Intel 系列 CPU 专门引入的一个用于表示中断的指令，目标进程只要一执行 INT 3 指令，就代表目标进程发生了异常，Windows 会将 ExceptionRecord.ExceptionCode 部分设为 EXCEPTION_BREAKPOINT，然后发送给调试进程。

有了如此方便的触发异常的方法，只需要将需要补丁的地址处的指令改成 INT 3，剩下的一切事情，交给调试进程吧！而调试进程是可以用任何语言编制的，可以实现任意功能的程序。很明显，一旦控制权交到调试进程手中，就可以对目标进程“为所欲为”了。而且该方法可以应付多个 INT 3 的情况，只需要收到

调试信息后，利用 GetThreadContext 函数得到目标进程的 EIP，就可以知道当前正处于什么样的位置上了。

本节利用 Debug API 完成一个类似于 keymake 的内存注册机，目标软件是没加壳的 TraceMe，该实例的序列号是明码比较，调用 lstrcmpA 函数比较序列号，此时 EBP 寄存器指向真序列号。代码如下：

```
0040138D  55              push    ebp                              ;ebp 指向真序列号
0040138E  50              push    eax
0040138F  FF15 04404000   call    dword ptr [KERNEL32.lstrcmpA>]
```

Loader 的执行方式可以分为三段：

（1）将目标地址 40138Dh 指令改写为 CCh（也就是 INT 3 的机器码）。

（2）中断触发异常后，读取 EBP 所指向的数据，并调用 MessageBox 显示出来。

（3）目标进程退出时，将其被修改的指令复原，以保证原软件功能的不变性。

本节提供的 Loader 没有考虑加壳的情况，其实现主体代码如下：

```
/*-------------------------------------------------------------*/
/*利用 Debug API 制作补丁                                      */
/*-------------------------------------------------------------*/
#define BREAK_POINT1     0x040138D          //需要中断的地址
#define SZFILENAME       ".\\TraceMe.exe"   //目标文件名
STARTUPINFO              si ;
PROCESS_INFORMATION      pi ;
BOOLWhileDoFlag=TRUE;
BYTE    ReadBuffer[MAX_PATH]={0};
BYTE    dwINT3code[1]={0xCC};
BYTE    dwOldbyte[1]={0};

if( !CreateProcess(SZFILENAME, NULL, NULL, NULL, FALSE,
      DEBUG_PROCESS|DEBUG_ONLY_THIS_PROCESS, NULL, NULL, &si, &pi)) {
         MessageBox(NULL, "CreateProcess Failed.", "ERROR", MB_OK);
         return FALSE;
   }

DEBUG_EVENT      DBEvent ;
CONTEXT          Regs ;
DWORD            dwState,Oldpp;

Regs.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS ;
while (WhileDoFlag) {
    WaitForDebugEvent (&DBEvent, INFINITE);
    dwState = DBG_EXCEPTION_NOT_HANDLED ;
    switch (DBEvent.dwDebugEventCode)
    {
      case CREATE_PROCESS_DEBUG_EVENT:
        //如果进程开始运行，则将断点地址的代码改为 INT 3 中断，同时备份原机器码
        ReadProcessMemory(pi.hProcess, (LPCVOID)(BREAK_POINT1), &dwOldbyte,1, NULL) ;
        WriteProcessMemory(pi.hProcess, (LPVOID)BREAK_POINT1,&dwINT3code,1,NULL);
        dwState = DBG_CONTINUE ;
        break;

      case  EXIT_PROCESS_DEBUG_EVENT :
        WhileDoFlag=FALSE;
        break ;

      case  EXCEPTION_DEBUG_EVENT:
        switch (DBEvent.u.Exception.ExceptionRecord.ExceptionCode)
```

```
        {
            caseEXCEPTION_BREAKPOINT:
            {
                GetThreadContext(pi.hThread, &Regs) ;
                if(Regs.Eip==BREAK_POINT1+1){
                    //中断触发异常事件，恢复原机器码，并读出数据
                    Regs.Eip--;
                    WriteProcessMemory(pi.hProcess,(LPVOID)BREAK_POINT1,&dwOldbyte, 1,0);
                    ReadProcessMemory(pi.hProcess, (LPCVOID)(Regs.Ebp),&ReadBuffer, 1, 0) ;
                    MessageBox (0, (char *)ReadBuffer, "pediy", MB_OK);
                    SetThreadContext(pi.hThread, &Regs) ;
                }
                dwState = DBG_CONTINUE ;
                break;
            }
        }
        break;
    }
    ContinueDebugEvent(pi.dwProcessId, pi.dwThreadId, dwState) ;
} //.end while

CloseHandle(pi.hProcess) ;
CloseHandle(pi.hThread)  ;
```

利用 INT 3 与前面几种方法相比，作为调试进程能做的事情更多，使用起来也更方便。但是仍然需要直接修改目标进程的代码的能力，也就是第 1 步，目标地址指令改写这一步。但是在很多情况下，软件被外壳保护，或软件自己的 CRC 校验过分强大（最突出的如 WinHex 等），或者软件中有多层 SMC 或其他各种“代码动态生成”机制时，该方法都会不适用。看来对于前面提出的两大难题：

（1）在适当的地址中断，控制权交给调试进程；

（2）控制权交付后，允许调试进程获取目标进程中断时的环境属性。

第 2 个难题可以通过 Debug API 完美解决；而前者，需要更底层、更隐蔽的方法，也就是让 CPU 辅助来调试！

18.2.3 利用调试寄存器机制

Windows 操作系统提供了两种层次的进程控制和修改机制：

- 跨进程内存存取机制；
- Debug API 监控目标进程运行信息。

但是这两种层次的进程监控机制都是运行在“操作系统”层次之上的，对于一些校验特别强悍的程序来说，这两种机制均不能满足要求。幸运的是，自 386 以后，Intel 公司已经在其 CPU 内部集成了 Dr0～Dr7 一共 8 个调试寄存器，并且对 EFLAGS 标志寄存器的功能也进行了扩展，使其也具有一部分调试的能力。所以最隐蔽和最通用的内存补丁制作方式应该是利用 CPU 内建的 DrX 调试寄存器的调试能力来对目标进程进行补丁。

从 Intel CPU 体系架构手册中，可以找到 Dr X 调试寄存器的介绍，参考第 2 章的图 2.25。当要对 401000h 设置的时候，将 Dr 0～Dr 3 其中的一个设置为 401000h，然后在 Dr 7 中设定相应的控制位。这样当被调试进程运行到 401000h 时，CPU 就会给调试器发送异常信息，调试器可以捕获该信息进行相应的操作。

下面来看看这个威力强大的“调试信息通知机制”如何应用。作为接收方，不能直接接收 DrX 调试寄存器发出来的中断/异常信息，Windows 已经将这个调试信息包装到了 Debug API 体系中，每当 DrX 调试信息被触发时，ExceptionRecord.ExceptionCode 部分都被设置成 EXCEPTION_SINGLE_STEP，只需要在

Debug API 循环中接受这个消息就可以达到目的。

自 Windows 2000 起，CreateProcess 后，没有办法在目标进程的入口点地址处中断，常见的解决办法有两种。

1. 利用 Single Step 机制

使进程在 Single Step 模式下运行，每执行一条指令就会给调试进程发送 EXCEPTION_SINGLE_STEP，当收到第一个 EXCEPTION_SINGLE_STEP 异常信号，表示中断在程序的第一条指令，即入口点。从而达到中断在入口点的目的。

本节用 DrX 调试目标程序 TraceMe，中断到指定地址，并将序列号显示出来。由于是利用 DrX 寄存器中断，因此目标程序可以加壳。CreateProcess 后的部分代码如下：

```
DEBUG_EVENT      DBEvent ;
CONTEXT          Regs ;
DWORD             dwSSCnt ;
dwSSCnt = 0 ;
Regs.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS ;

//使进程在 Single Step 模式下运行
//本例只需要设置一次，如果还想继续让程序 Single Step 下去，必须重新设置 SF 位
GetThreadContext(pi.hThread,&Regs);
Regs.EFlags|=0x100;
SetThreadContext(pi.hThread,&Regs);
ResumeThread(pi.hThread);

while (WhileDoFlag) {
    WaitForDebugEvent (&DBEvent, INFINITE);
    switch (DBEvent.dwDebugEventCode)
    {
    case EXCEPTION_DEBUG_EVENT:
        switch (DBEvent.u.Exception.ExceptionRecord.ExceptionCode)
        {
        case EXCEPTION_SINGLE_STEP :
            {
                ++dwSSCnt ;
                if (dwSSCnt == 1)
                {
                    //当收到第一个异常信号，表示中断在程序的第一条指令，即入口点
                    //把 Dr0 设置成程序的入口地址
                    GetThreadContext(pi.hThread,&Regs);
                    Regs.Dr0=Regs.Eax;
                    Regs.Dr7=0x101;
                    SetThreadContext(pi.hThread,&Regs);
                }
                else if (dwSSCnt == 2)
                {
                    //第二次中断在起先设置的入口点，在 BREAK_POINT1 处设置硬件断点
                    GetThreadContext(pi.hThread, &Regs) ;
                    Regs.Dr0 = BREAK_POINT1;
                    Regs.Dr7 = 0x101 ;
                    SetThreadContext(pi.hThread, &Regs) ;
                }
                else if (dwSSCnt == 3)
                {
                    //第三次中断，已到指定的地址，读取 EBP 寄存器指向的内存数据
```

```
                    GetThreadContext(pi.hThread, &Regs) ;
                    Regs.Dr0 = Regs.Dr7 = 0 ;
                    ReadProcessMemory(pi.hProcess, (LPCVOID)(Regs.Ebp), &ReadBuffer,\
                    sizeof(ReadBuffer), NULL) ;
                    MessageBox (0, (char *)ReadBuffer, "test", MB_OK);
                    SetThreadContext(pi.hThread, &Regs) ;
                }
                break ;
            }
        }
        break ;
    caseEXIT_PROCESS_DEBUG_EVENT :
            WhileDoFlag=FALSE;
            break ;
    }
    ContinueDebugEvent(pi.dwProcessId, pi.dwThreadId, DBG_CONTINUE) ;
} //.end while
```

上面的代码的流程大致是这样子的：

① 调用 CreateProcess 函数创建目标进程 TraceMe_asp.exe，在创建进程的时候，传递 DEBUG_PROCESS|DEBUG_ONLY_THIS_PROCESS 调试参数。

② 由于是以 DEBUG_PROCESS|DEBUG_ONLY_THIS_PROCESS 参数创建的目标进程，所以拥有对目标进程完全的调试权和控制权。调试程序的主体就是一个调用 WaitForDebugEvent (&DBEvent, INFINITE) 函数构成的循环，一直到目标进程退出后，调试进程才会退出该循环。

③ 将目录进程设置在 Single Step 模式下运行

④ 当程序运行第一条指令时，就会第一次发送 EXCEPTION_SINGLE_STEP 异常信号时，表示中断在程序入口点处。并且设置 Dr0 的值等于入口点地址，Dr7 = 101h，表示 CPU 执行到 Dr0 中的地址时，发出异常信号。

⑤ 第二次收到 EXCEPTION_SINGLE_STEP 异常信号时，表示已经到达了程序的入口，这时，设置 Dr0 的值等于地址 40138Dh。

⑥ 第三次收到 EXCEPTION_SINGLE_STEP 异常信号时，表示已中断在 40138Dh，此时清除断点，并将 EBP 所指向的内容读出。

整个程序的运行过程都没有对目标文件进行任何改动，却实现了接近完美的“消息通知”机制，而这一切，都得利于 CPU 上面的 DrX 寄存器的强大功能！

当然，DrX 寄存器作为寄存器级别的调试工具，其应用范围绝不止内存补丁这么一种，结合 Windows 操作系统的 Debug API 功能和一些底层的驱动设施，DrX 可以在很多与底层紧密交互的场合发挥出巨大的作用。而这一切，由于与本书主旨相去甚远，就留待给读者朋友日后工作中自己探索。

2．利用 ntdll.ntcontinue 作为跳板

CreateProcess 后，当程序执行到 ntdll.ntcontinue 时，再对程序入口地址进行设断操作。EliCZ（API Hook Server 的作者）最先发现了这个跳板，然后经由 yoda（PeEditor 及 LordPE 的作者）的 bpm example code 才流传开。

完整的代码见光盘映像文件，代码的流程大致是这样子的：

① 调用 CreateProcess 函数创建目标进程 TraceMe_asp.exe。

② 目标进程正式运行前，会给调试进程发送一个 EXCEPTION_BREAKPOINT 调试信息，在调试进程内，通过 dwBPCnt 计数器判断出接收到的是第一个调试信息。然后是用 GetProcessAddress 和 GetModuleHandle 得到 ntdll.ntcontinue 的地址，并且设置 Dr0 的值等于该地址，Dr7 = 101h，表示 CPU 执行到 Dr0 中的地址时，发出异常信号。

```
case EXCEPTION_BREAKPOINT:
    {
        ++dwBpCnt ;
        if (dwBpCnt == 1)
        {
            GetThreadContext(pi.hThread, &Regs) ;
            Regs.Dr0 = (DWORD)(GetProcAddress(GetModuleHandle("ntdll.dll"), "NtContinue") );
            Regs.Dr7 = 0x101 ;
            SetThreadContext(pi.hThread, &Regs) ;
            dwState = DBG_CONTINUE ;
        }
        break ;
    }
```

③ 当收到第一个 EXCEPTION_SINGLE_STEP 异常信号时，表示中断在 ntdll.ntcontinue 函数，在这里，要把 Dr0 设置成程序的入口地址，可是这时不能用 SetContextThread，而要用 ESP 的地址作为跳板，来设置调试寄存器的内容。具体的算法，程序中已经很清楚。

```
case EXCEPTION_SINGLE_STEP :
    {
        ++dwSSCnt ;
        if (dwSSCnt == 1)
        {
            GetThreadContext(pi.hThread, &Regs) ;
            Regs.Dr0 = Regs.Dr7 = 0 ;
            SetThreadContext(pi.hThread, &Regs) ;
            ReadProcessMemory(pi.hProcess,(LPCVOID)(Regs.Esp+4),&dwAddrProc, 4,0);
            ReadProcessMemory(pi.hProcess, (LPCVOID)dwAddrProc, &Regs, sizeof(CONTEXT),0);
            Regs.Dr0 = BREAK_ENTRYPOINT ;
            Regs.Dr7 = 0x101 ;
            WriteProcessMemory(pi.hProcess, (LPVOID)dwAddrProc, &Regs, sizeof(CONTEXT),0);
            dwState = DBG_CONTINUE ;
        }
    ......
    }
```

④ 第二次收到 EXCEPTION_SINGLE_STEP 异常信号时，表示已经到达了程序的入口，这时，设置 Dr0 的值等于地址 40138Dh。

⑤ 第三次收到 EXCEPTION_SINGLE_STEP 异常信号时，表示已中断在 40138Dh，此时清除断点，并将 EBP 所指向的内容读出。

18.2.4　DLL 劫持技术

当一个可执行文件运行时，Windows 加载器将可执行模块映射到进程的地址空间中，加载器分析可执行模块的输入表，并设法找出任何需要的 DLL，并将它们映射到进程的地址空间中。

由于输入表中只包含 DLL 名而没有它的路径名，因此加载程序必须在磁盘上搜索 DLL 文件。首先会尝试从当前程序所在的目录加载 DLL，如果没找到，则在 Windows 系统目录中查找，最后是在环境变量中列出的各个目录下查找。利用这个特点，先伪造一个系统同名的 DLL，提供同样的输出表，每个输出函数转向真正的系统 DLL。程序调用系统 DLL 时会先调用当前目录下伪造的 DLL，完成相关功能后，再跳到系统 DLL 同名函数里执行，如图 18.3 所示。这个过程用个形象的词来描述就是系统 DLL 被劫持（hijack）了。

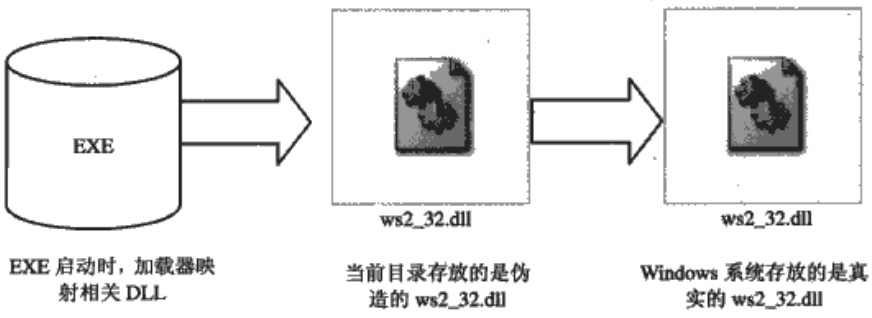


图 18.3　DLL 劫持技术演示

利用这种方法取得控制权后，可以对主程序进行补丁。此种方法只对除 kernel32.dll、ntdll.dll 等核心系统库以外的 DLL 有效，如网络应用程序的 ws2_32.dll、游戏程序中的 d3d8.dll，还有大部分应用程序都调用的 lpk.dll，这些 DLL 都可被劫持。

利用第 5 章 5.6.2 节提供的 CrackMeNet.exe 来演示一下如何利用劫持技术制作补丁，目标文件用 Themida v1.9.2.0 加壳保护。

1．补丁地址

去除这个 CrackMe 网络验证方法参考第 5 章 5.6.2 节，将相关补丁代码存放到函数 PatchProcess()里。例如将 401496h 改成：

```
00401496   EB 29    jmp     short 004014C1
```

补丁编程实现就是：

```
    unsigned char p401496[2] = {0xEB, 0x29};
    WriteProcessMemory(hProcess,(LPVOID)0x401496, p401496, 2, NULL);
```

p401496 这个数组的数据格式，可以用 OllyDbg 插件获得，或十六进制工具转换。例如 Hex Workshop 打开文件，执行菜单“Edit/Copy As/Source”即可得到相应的代码格式。

2．构建输出函数

查看实例 CrackMeNet.exe 输入表，会发现名称为“ws2_32.dll”的 DLL，因此构造一个同名的 DLL 来完成补丁任务。伪造的 ws2_32.dll 有着真实 ws2_32.dll 一样的输出函数，完整源码见光盘映像文件。实现时，可以利用 DLL 模块中的函数转发器来实现这个目标，其会将对一个函数的调用转至另一个 DLL 中的另一个函数。可以这样使用一个 pragma 指令：

```
#pragma comment(linker, "/EXPORT:SomeFunc=DllWork.someOtherFunc")
```

这个 pragma 告诉链接程序，被编译的 DLL 应该输出一个名叫 SomeFunc 的函数。但是 SomeFunc 函数的实现实际上位于另一个名叫 SomeOtherFunc 的函数中，该函数包含在称为 DllWork. dll 的模块中。

如果要达到劫持 DLL 的目的，生成的 DLL 输出函数必须与目标 DLL 输出函数名一样。本例可以这样构造 pragma 指令：

```
#pragma comment(linker, "/EXPORT:WSAStartup=_MemCode_WSAStartup,@115")
```

编译后的 DLL，会有与 ws2_32.dll 同名的一个输出函数 WSAStartup，实际操作时，必须为想要转发的每个函数创建一个单独的 pragma 代码行，读者可以用工具 AheadLib 或用其他办法，将 ws2_32.dll 输出函数转换成相应的 pragma 指令。

当应用程序调用伪装 ws2_32.dll 的输出函数时，必须将其转到系统 ws2_32.dll 中，这部分的代码自己

实现。例如，WSAStartup 输出函数构造如下：

```
ALCDECL MemCode_WSAStartup(void)
{
    GetAddress("WSAStartup");
    __asm JMP EAX;//转到系统 ws2_32.dll 的 WSAStartup 输出函数
}
```

其中，GetAddress()函数的代码如下：

```
// MemCode 命名空间
namespace MemCode
{
HMODULE m_hModule = NULL;                    //原始模块句柄
DWORD m_dwReturn[500] = {0};                 //原始函数返回地址
// 加载原始模块
inline BOOL WINAPI Load()
{
    TCHAR tzPath[MAX_PATH]={0};
    TCHAR tzTemp[MAX_PATH]={0};
    GetSystemDirectory(tzPath, sizeof(tzPath));
    strcat(tzPath,"\\ws2_32.dll");
    m_hModule = LoadLibrary(tzPath);//加载系统系统目录下 ws2_32.dll
    if (m_hModule == NULL)
    {
        wsprintf(tzTemp, TEXT("无法加载 %s，程序无法正常运行。"), tzPath);
        MessageBox(NULL, tzTemp, TEXT("MemCode"), MB_ICONSTOP);
    }
    return (m_hModule != NULL);
}

// 释放原始模块
inline VOID WINAPI Free()
{
    if (m_hModule)
        FreeLibrary(m_hModule);
}
// 获取原始函数地址
FARPROC WINAPI GetAddress(PCSTR pszProcName)
{
    FARPROC fpAddress;
    TCHAR szProcName[16]={0};
    TCHAR tzTemp[MAX_PATH]={0};

    if (m_hModule == NULL)
    {
        if (Load() == FALSE)
            ExitProcess(-1);
    }

    fpAddress = GetProcAddress(m_hModule, pszProcName);
    if (fpAddress == NULL)
    {
        if (HIWORD(pszProcName) == 0)
        {
            wsprintf(szProcName, "%d", pszProcName);
```

```
            pszProcName = szProcName;
        }
        wsprintf(tzTemp, TEXT("无法找到函数 %hs，程序无法正常运行。"), pszProcName);
        MessageBox(NULL, tzTemp, TEXT("MemCode"), MB_ICONSTOP);
        ExitProcess(-2);
    }
    return fpAddress;
}
}
using namespace MemCode;
```

编译后，用 LordPE 查看伪造的 ws2_32.dll 输出函数，和真实 ws2_32.dll 完全一样，如图 18.4 所示。

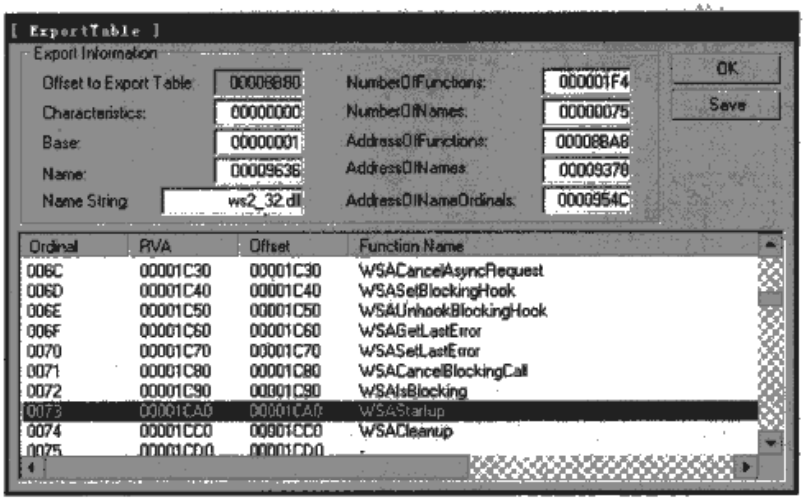


图 18.4　伪造 ws2_32.dll 的输出表

查看伪造的 ws2_32.dll 中任意一个输出函数，例如 WSACleanup：

```
.text:10001CC0 ; int __stdcall WSACleanup()
.text:10001CC0 WSACleanup   proc near
.text:10001CC0              push    offset aWsacleanup ;"WSACleanup"
.text:10001CC5              call    sub_10001000   ;GetAddress(WSACleanup)
.text:10001CCA              jmp     eax
.text:10001CCA WSACleanup   endp
```

会发现输出函数 WSACleanup()首先调用 GetAddress(WSACleanup)，获得真实 ws2_32.dll 中 WSACleanup 的地址，然后跳过去执行，也就是说，ws2_32.dll 各输出函数被 Hook 了。

3. 劫持输出函数

ws2_32.dll 有许多输出函数，经分析，程序发包或接包时，WSAStartup 输出函数调用的比较早，因此在这个输出函数放上补丁的代码。代码如下：

```
ALCDECL MemCode_WSAStartup(void)
{
    hijack();
    GetAddress("WSAStartup");
    __asm JMP EAX;
}
```

hijack()函数主要是判断是不是目标程序，如果是就调用 PatchProcess()进行补丁。

```
void hijack()
{
```

```
    if (isTarget(GetCurrentProcess()))   //判断主程序是不是目标程序，是则补丁之
    {
        PatchProcess(GetCurrentProcess());
    }
}
```

伪造的 ws2_32.dll 制作好后，放到程序当前目录下，这样当原程序调用 WSASTartup 函数时就调用了伪造的 ws2_32.dll 的 WSASTartup 函数，此时 hijack()函数负责核对目标程序校验，并将相关数据补丁好，处理完毕后，转到系统目录下的 ws2_32.dll 执行。

这种补丁技术，对加壳保护的软件很有效，选择挂接的函数最好是在壳中没有被调用的，当挂接函数被执行时，相关的代码已被解压，可以直接补丁了。在有些情况下，必须用计数器统计挂接的函数的调用次数来接近 OEP。此方法巧妙地绕过了壳的复杂检测，很适合加壳程序的补丁制作。

一些木马或病毒也会利用 DLL 劫持技术搞破坏，因此当在应用程序目录下发现系统一些 DLL 文件存在时，如 lpk.dll，应引起注意。

18.3 SMC 补丁技术

利用 SMC 能修改自身代码这个特点，可以对加壳程序直接进行补丁，效果相当于内存补丁。加壳程序执行时，都有一个将数据解压并将其写入原始映像基址处的过程，在代码刚恢复还没运行前，在外壳里插入一段补丁代码，对刚解压出来的数据补丁。

18.3.1 单层 SMC 补丁技术

本节以 UPX 外壳演示一下单层 SMC 补丁技术。用 UPX 对第 5 章的 EnableMenu 加壳，保存为 MenuUPX.exe。恢复程序功能需要修改代码如下：

原指令	:004011E3 6A01	push 00000001
修改后指令	:004011E3 6A00	push 00000000

现在不脱壳，直接在原文件里增加代码去除补丁限制。思路就是外壳代码将程序在内存中解压完毕时，让其先跳到补丁代码处执行补丁，再回到原指令处继续正常工作，具体过程如图 18.5 所示。

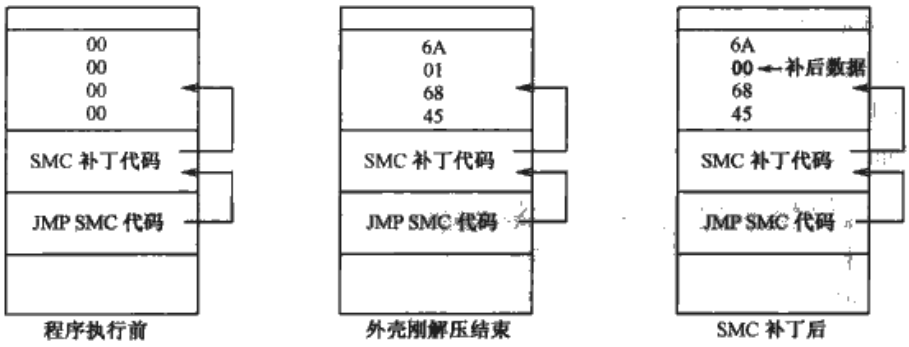


图 18.5 SMC 补丁示意图

用 OllyDbg 打开 MenuUPX，发现 UPX 外壳跳到入口点（OEP）处的代码形式如下：

```
0040BCCE   61            popad
0040BCCF   E9 3C55FFFF   jmp     00401210          ;让其跳到 SMC 代码处
0040BCD4   00            db      00
```

这段外壳代码是以未压缩的形式存在于外壳代码里的，外壳执行到这里时，原始的代码数据已还原，因此可以用此处作为 SMC 起点。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
UPX0	00001000	00007000	00000400	00000000	E0000080
UPX1	00008000	00004000	00000400	00003E00	E0000040
.rsrc	0000C000	00001000	00004200	00000C00	C0000040

图 18.6　查看区块信息

在加壳的 MenuUPX.exe 里，找一空间存放补丁代码。要求文件解压后，这个空间不能被原文件覆盖。同时程序运行时，不能被调用，如全局变量的调用。空间可以在各个区块间隙中获得，也可对原文件增加一个区块。此例选择前一种方案。用 LordPE 查看其区块信息，如图 18.6 所示。

UPX 加壳后已将区块重新组织，分别是 UPX0、UPX1 等。UPX 外壳代码在 UPX1 里，被压缩的原始数据放在 UPX1 中。运行时，外壳将解压缩后的原始代码映射到 UPX0。PE 文件各区块之间总有些间隙，在 UPX1 区块与.rsrc 区块之间有一段空白代码空间（填充的 00），其文件偏移为 40E0h～4200h。由于这段空隙是在外壳代码空间里，加壳程序运行到 OEP 时，外壳代码部分将不会被执行了，因此这段空隙很适合放补丁代码。

被修改的 UPX1 区块属性必须可读写。幸运的是，一般加壳软件的区块是可读可写的。SMC 补丁代码如下：

```
0040BCCF   E9 0C000000          jmp     0040BCE0 ;解压结束后跳到 SMC 修补代码处
0040BCD4   00                   db      00
0040BCD5   00                   db      00
.....
;SMC 补丁代码
0040BCE0   66:C705 E3114000 6A00 mov     word ptr [4011E3], 6A ;执行修补指令
0040BCE9   E9 2255FFFF          jmp     00401210              ;返回 OEP
```

最后，用 OllyDbg 将修改后的结果保存到文件。

18.3.2　多层 SMC 补丁技术

一些外壳多层嵌套加密压缩，即在第一层代码中解密还原第二层代码，而第二层代码则解密第三层代码，依此类推。这就必须用多层 SMC 代码补丁才能达到效果。

用 ASPack 对 EnableMenu.exe 加壳，命名为“Menu_ASPack.exe”。用 LordPE 查看其区块信息，如图 18.7 所示。.aspack 区块是外壳部分，其末尾的空白部分可以存放补丁代码。OllyDbg 加载，查看.aspack 区块尾部，将 BB40hh（RVA）作为补丁空间。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	00003000	00001000	00001A00	C0000040
.rdata	00004000	00001000	00002A00	00000600	C0000040
.data	00005000	00001000	00003000	00000200	C0000040
.rsrc	00006000	00004000	00003200	00001C00	C0000040
.aspack	0000A000	00002000	00004E00	00001C00	C0000040
.data	0000C000	00001000	00006A00	00000000	C0000040

图 18.7　查看区块信息

如要正确 SMC，必须确定需修改的语句“4011E3 6A01 push 1”在内存何处被解压。用 OllyDbg 加载 Menu_ASPack 后，在数据窗口对 4011E3h 下硬件写断点。

同时查看数据窗口 4011E3h，直到代码被还原。此时程序外壳程序的代码如下

```
0040A179  rep     movs dword ptr es:[edi], dword ptr [esi]      ;中断在此
0040A17B  mov     ecx, eax
0040A17D  and     ecx, 3
0040A180  rep     movs byte ptr es:[edi], byte ptr [esi]
0040A182  pop     esi
0040A183  push    8000
0040A188  push    0
0040A18A  push    dword ptr [ebp+443FF4]
0040A190  call    dword ptr [ebp+444000]
0040A196  add     esi, 8
0040A199  cmp     dword ptr [esi], 0
0040A19C  jnz     0040A0C8                          ;循环解压恢复各区块
0040A1A2  push    8000
```

程序会中断在 repz movsd 语句处，向下不远处是个判断是否解压完毕的语句，在此之后让它跳到空白代码处。即做如下修改：

```
0040A19C   0F85 26FFFFFF   jnz    0040A0C8
0040A1A2   E9 99190000     jmp    0040BB40          ;跳到空白处，进行补丁
```

40BB40h 处的补丁代码如下：

```
0040BB40  mov     word ptr [4011E3], 006A      ;补丁代码
0040BB49  push    8000                         ;原补丁处的指令搬到此处
0040BB4E  push    0040A1A7                        ;40A1A7 入栈
0040BB53  retn                                 ;跳到 40A1A7 处继续执行，等同 jmp
```

因为 40A1A2h 处的代码是动态生成的，磁盘文件中此处数据是加密的，所以必须再次用 SMC 技术补丁此处。首先确定这行代码在内存何处被解压。在数据窗口对 40A1A2h 地址下内存写断点或硬件写断点，重新用 OllyDbg 加载实例运行。会中断如下指令处：

```
0040A57B   0FBFDB           movsx  ebx, bx       ;运行到此处 40A1A2h 代码还原了
0040A57E   BB B13B5466      mov    ebx, 66543BB1
0040A583   41               inc    ecx
0040A584   0FBFFD           movsx  edi, bp
0040A587   81ED 02000000    sub    ebp, 2
0040A58D   81ED 02000000    sub    ebp, 2
0040A593   81EA 4F8DC947    sub    edx, 47C98D4F
0040A599   E9 1C000000      jmp    0040A5BA
……
0040A5BA   0FBFF6           movsx  esi, si
0040A5BD   81F9 B78BBBB6    cmp    ecx, B6BB8BB7
0040A5C3   0F85 68FFFFFF    jnz    0040A531
0040A5C9   BB F1264F36      mov    ebx, 364F26F1
0040A5CE   E9 68FEFFFF      jmp    0040A43B       ;选择此处 SMC
```

用十六进制工具可以在加壳的 MenuASPack 文件里找到上述代码的机器码，意味着可以直接修改程序了。选择何处代码执行 SMC 操作很关键，经分析，在 40A5CEh 一行时，已跳出了循环，40A1A2h 代码被还原。所以选择 40A5CEh 处跳到空白代码处。代码修改如下：

```
0040A5C9   BB F1264F36    mov    ebx, 364F26F1
0040A5CE   E9 86150000    jmp    0040BB59        ;修改此处
```

40BB59h 这段 SMC 代码是补丁 40A1A2h 处的指令，使 40A1A2h 处指令如下：

```
0040A1A2   E999190000     jmp       0040BB40
```

在 OllyDbg 中键入补丁代码：

```
0040BB59  mov     byte ptr [40A1A2], 0E9
0040BB60  mov     dword ptr [40A1A3], 1999      ;将40A1A2h指令改成jmp 40BB40
0040BB6A  jmp     0040A43B                      ;原40A5CEh的指令为jmp 40A43B
```

将上述修改保存到磁盘文件里，就可完成此次的补丁了。一般外壳的代码区段属性是可读写的，如果被修补的地址不可写，此时必须调用 VirtualProtectEx 函数将其属性设置为可写。本例是两层 SMC，有些外壳可能要多层 SMC 才能成功，操作方法类似两层，一层层地修补，一直到外壳的可见代码为止。由于补丁时，改变了原程序的指令，一些外壳会对代码进行完整性检查，因此遇到这种情况，必须找到校验处，将原始的校验值返回给外壳。

18.4 补丁工具

如果不喜欢编程，也可用补丁制作工具制作同样效果的补丁文件。补丁制作工具的种类较多，比如经典的文件补丁 CodeFusion 等，这些工具使用比较简单，参考一下帮助文档就能掌握。本节以 dUP 这款优秀的补丁制作软件为例，简单讲述一下补丁工具的使用。

dUP 支持偏移补丁、查找与替换补丁、注册表补丁等，并可将这些方式混合，支持文件补丁和内存补丁，功能灵活强大，是目前一款流行的补丁制作工具。

dUP 支持自定义界面，单击 Settings 标签，设置 dUP 的运行环境和补丁界面，如设置图标，定制皮肤，定制窗口形状等。

1．制作偏移补丁

补丁具体制作步骤在“Patch Data”选项卡里，单击“New Project”按钮，设置补丁程序的一些说明信息。单击“Add”按钮，添加补丁方式，dUP 的各类补丁就在这选择。选择“Offset Patch”方式，然后在主界面里选择刚添加的“Offset Patch”补丁方式，单击“Edit”按钮，对其编辑，如图 18.8 所示。

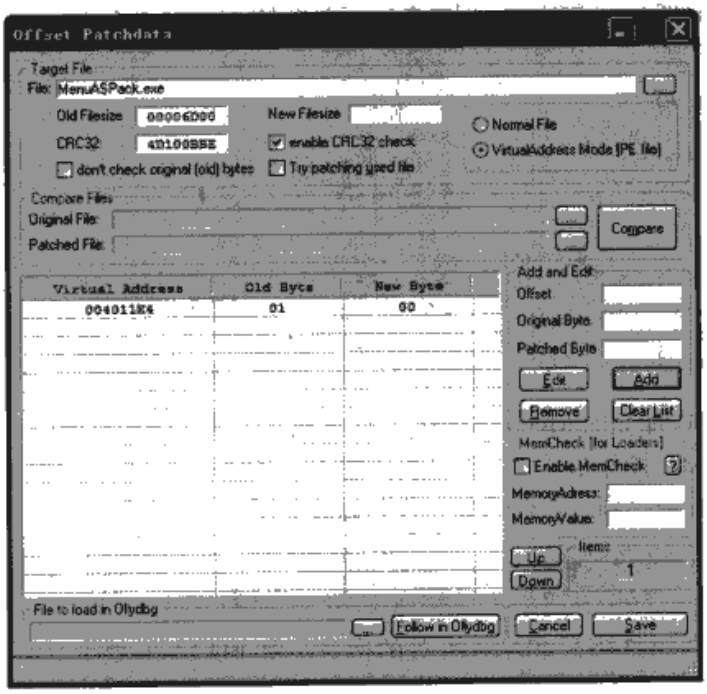


图 18.8　偏移补丁设置

对未加壳的程序制作文件补丁，选择“Normal File”模式，将补丁的文件偏移地址通过 Add 按钮填进来，也可比较（Compare Files）修改前后的文件来获得这些地址。

如果是要对加壳程序制作补丁，选择“VirtualAddress Mode”模式，以虚拟地址方式进行补丁。

添加补丁地址结束后，返回主界面，单击“Creater Loader”按钮创建内存补丁或单击“Creater Patch”钮创建文件补丁。其中文件补丁支持加壳程序，其原理是在原程序添加一段代码，运行时创建一个线程，监视指定地址数据，然后适机补丁。读者可以用本节的 MenuASPack 做个试验。

2．查找与替换补丁

在补丁方式里，选择“Search & RePlace Patch”，其可以在被补丁的程序中搜索指定的机器码，并替换成所需要的值，如图 18.9 所示。

所查找的机器码可能在程序里有重复，可以选择修改第几次出现的机器码进行补丁，建议查找的机器码尽可能长些，以降低重复的几率。如果没加壳，可以单击“check occurrence”按钮，其可查找重复机器码重复的频次。如果目标加壳，请将选项“Target is a compressed PE file”勾上。

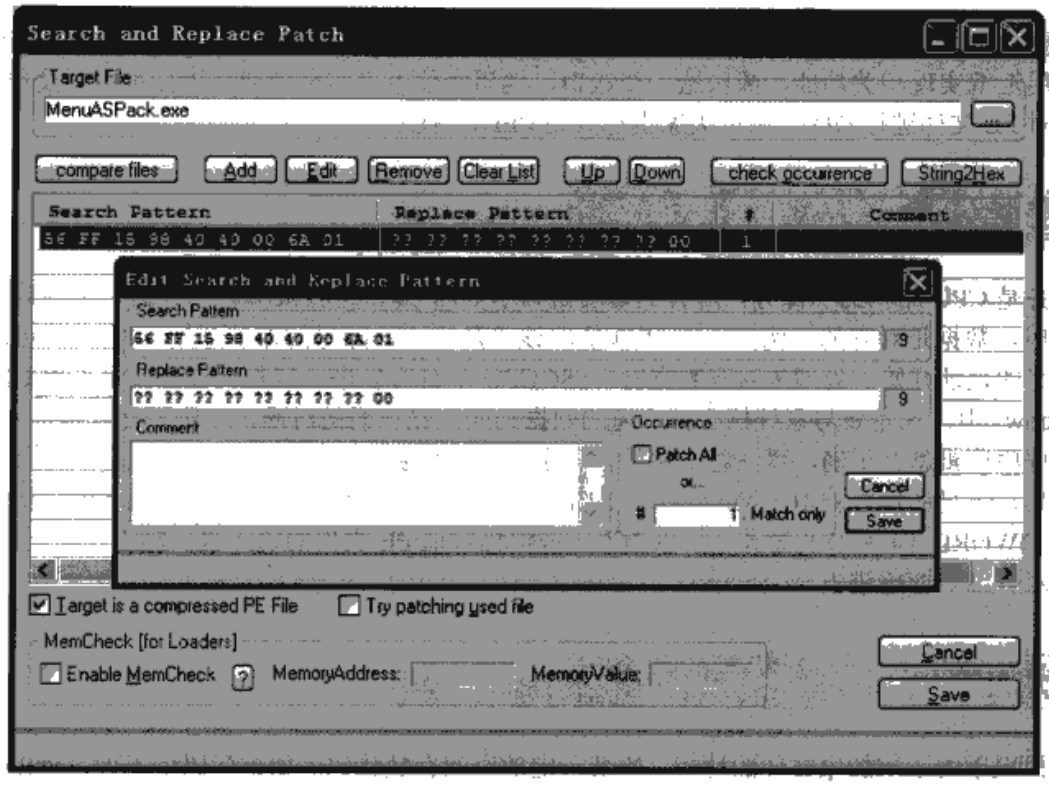


图 18.9　查找与替换补丁

3．注册表和附件补丁

如果需要往注册表里写相关数据，可以用 dUP 直接完成，将注册表文件*.reg 导入即可。发布补丁时，可能需要附带一些文件，如 DLL 等，可以通过 dUP 的“Attached File”功能来完成。

第19章 代码的二次开发

本章主要是讨论在没有源码和无接口的情况下扩充可执行文件的功能，目标是二进制的 EXE 或 DLL 文件，需要用汇编实现相关功能，或构造一个接口，调用其他语言实现功能。该技术主要是修改扩充 PE 结构功能，对 PE 文件进行 DIY，所以大家也称其为 PEDIY 技术。

19.1 数据对齐

数据对齐是 CPU 结构的一部分，对齐的目的是为了提高 CPU 运行效率。当数据大小的数据模数的内存地址是 0 时，数据是对齐的。例如，WORD 值应该总是从被 2 除尽的地址开始，而 DWORD 值应该总是从被 4 除尽的地址开始。处理未对齐数据时，x86 CPU 本身可以进行调整，代价就是占用 CPU 资源。

在 Windows 中，凡是要与系统核心打交道的数据结构，在声明的时候，一定要让其地址 4 字节对齐；否则，任凭程序逻辑怎么正确，算法怎么精妙，一定得不到想要的结果。微软文档规定映像页面相关数据必须对齐排列，不足的地方补 0。Windows 上的 PE 文件里的数据都是按这个要求对齐的。例如，输入表里，RVA 以 DWORD（4 字节）对齐。所以当手工或编程构造 PE 文件的输入表、输出表和重定位表等时，必须将数据对齐细节考虑进去。

19.2 增加空间

在某些情况下为了增加原文件的功能，需要一定的空间存放代码。如果代码量不大，可以放到区块间隙里；否则必须增加一个区块。

19.2.1 区块间隙

由于 PE 文件每个区块的大小必定等于磁盘对齐值的整数倍，而区块的实际代码或数据的大小不一定刚好是这么多，所以在不足的地方一般以 00 来填充，这就是区块间的间隙,具体参考第 10 章 10.4.3 节。

实例 pediy.exe 的 FileAlignment 值为 1000h，其磁盘文件中区块的大小就是 1000h 的整数倍，由于每个区块数据实际大小（见 VSize 值）不足这个值，因此各区块末尾都有一段空白的空间可以用，如图 19.1 所示。可以用十六进制工具打开文件查看，会发现 VSize 值后的数据是 00，这些就是区块间隙。

[Section Table]

Name	VOffset	VSize	ROffset	RSize	Flags
.text	00001000	0000024C	00001000	00001000	60000020
.rdata	00002000	00000226	00002000	00001000	40000040
.data	00003000	00000064	00003000	00001000	C0000040
.rsrc	00004000	00000460	00004000	00001000	40000040

图 19.1 查看区块信息

利用区块间隙时，必须注意区块属性（Characteristics），其指出了该区块是否可读写的问题。例如，查看本实例的.data 区块属性，在相应区块名上单击右键，执行“edit section header”命令，再单击“Flags”按钮打开属性状态图，其状态为可写（Writeable），如图 19.2 所示。

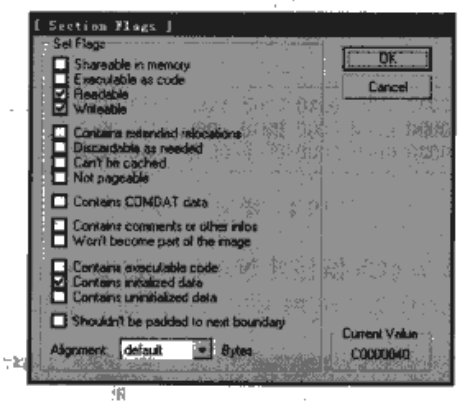


图 19.2 查看区块属性

由于.data 区块是可写的，其区块中的一些间隙可能会被程序中的变量所使用，如全局变量或变量的缓冲区等。因此使用可写属性区块的间隙时，必须注意这些问题。如果区块属性是只读的，一般来说相对安全。

19.2.2 手工构造区块

在补丁的代码量比较大的情况下，可以新增一个区块。本节将在 pediy.exe 文件尾部（5000h）处）增加一个大小为 1000h 的数据段。

手工构造区块能熟悉 PE 格式，实际操作时一般可用工具辅助。增加区块有三个工作要做：一是增加一个块头；二是增加块头指向的数据段；三是调整文件映像尺寸。

构造区块时，必须注意区块的对齐。如果区块不对齐，在 Windows 9x 系统上程序运行可能没问题，但在 Windows 2000/XP 下程序会出错，如图 19.3 所示。

图 19.3 区块不对齐而出错

1. 修正块表

块表（Section Table）位于 PE 文件头之后，块表由一系列的 IMAGE_SECTION_HEADER 结构排列而成，每个结构描述的是一个块。结构的排列顺序和它们描述的块在文件中的排列顺序是一致的。全部有效结构的最后以一个空的 IMAGE_SECTION_HEADER 结构作为结束。

现在增加一个块头，具体内容如下：

```
IMAGE_SECTION_HEADER   STRUC
  Name= 'pediy'                    ; 8 bytes, 块名
  VirtualSize=1000h                ; 4 bytes, 该块真实长度
  VirtualAddress=5000h             ; 4 bytes, RVA 地址
  SizeOfRawData =1000h             ; 4 bytes, 在文件中对齐后的尺寸
  PointerToRawData=5000h           ; 4 bytes, 在文件中偏移
  PointerToRelocations=0           ; 4 bytes, 在 OBJ 文件中使用, 重定位的偏移
  PointerToLinenumbers=0           ; 4 bytes, 行号表的偏移（供调试用）
  NumberOfRelocations =0           ; 2 bytes, 在 OBJ 文件中使用, 重定位项数目
```

```
  NumberOfLinenumbers =0          ; 2 bytes，行号表中行号的数目
  Characteristics=E0000020h       ; 4 bytes，块属性，表示包含执行代码，可读写并可执行
IMAGE_SECTION_HEADER  ENDS
```

构造区块头时，一定要注意块对齐的问题。用十六进制工具在原块表后，将上面的数据填入，具体如图 19.4 所示。

```
00000250 0000 0000 4000 0040 7065 6469 7900 0000 ....@..@pediy...
00000260 0010 0000 0050 0000 0010 0000 0050 0000 .....P.......P..
00000270 0000 0000 0000 0000 0000 0000 2000 0060 ............ ...
```

图 19.4　增加一个块头

增加了一个块头后，就要修正 PE 头 6Ch 偏移处 NumberOfSections 的值，将其由原来的 4 改成 5，表示当前区块数为 5。用 LordPE 查看修正过的块表会发现多了一个区块（见图 19.5）。

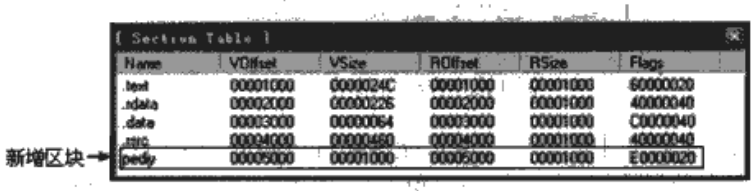


图 19.5　增加后的块头

2．增加数据段

有了区块头，但区块中无数据，程序还不能运行。用十六进制工具在文件尾部 5000h 处插入 1000h 大小的数据块，数据块内容为 0。这样，pediy 区块就指向了大小为 1000h 的数据块，可以在这增加所需要的代码了。

3．修正映像文件尺寸

因为扩大了原文件的尺寸，所以必须修正 SizeOfImage 的值，将其由原来的 5000h 改成 6000h。

19.2.3　工具辅助构造区块

实际增加区块操作时，一般用工具辅助完成。使用 LordPE 打开文件，在区块的列表上，执行右键菜单中的“add section header”功能就可增加一个区块。如果 LordPE 选项里勾选了“autofix SizeOfImage”，则自动修正 SizeOfImage 的值。但数据段的内容，还需要十六进制工具完成。另一款 PE 工具 CFF Explorer，这方面功能比较强大，能自动增加区块及所对应的数据内容。专门增加 PE 文件区块的工具还有 ZeroAdd、Topo 等，操作都很简单。

一般的软件 PE 头部分的区块表（Section Table）后是一段全 0 的空间，因此新增区块不会破坏文件。但也有一些软件的块表后的空间没有富余，就不能增加区块了。例如 Windows 自带的一些程序，在块表后面紧跟着就是 BoundImport 数据。用 ZeroAdd 处理具有 BoundImport 结构的程序时，会报出“PE 头空间不足”的提示。解决办法是在数据目录表里将 BoundImport 的 RVA 和 Size 赋零，同时将 BoundImport 数据区清零，处理后就可正常增加区块了。

19.3　获得函数的调用

在扩充程序功能时，经常遇到调用的 API 函数不在输入表中。解决方法：一种方法是修改输入表结构，增加相应的 API 函数；另一种方法是用显式链接方式调用 DLL 相关函数。

19.3.1　增加输入函数

可以用十六进制工具修改输入表中 IID 的成员，增加新的输入函数。也可以用相关的 PE 工具增加输入函数，如 LordPE 等工具。

本例将对实例文件增加 MessageBoxA 函数调用。用 LordPE 工具打开光盘映像文件中的实例 addapi.exe，单击“Directories”按钮打开目录表窗口，在“Import Table”域中单击 ... 按钮打开输入表编辑窗口，然后在任意一个 DLL 文件上单击鼠标右键，执行“Add Import”命令打开增加函数窗口（见图 19.6）。输入 DLL 文件名和函数名，单击“OK”按钮即可为原文件增加一个函数。LordPE 将新增一个区块，来存放新增加的 IID 数据。

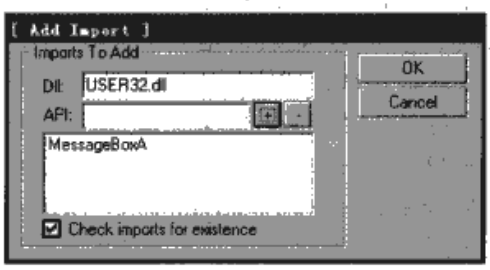


图 19.6　LordPE 增加输入函数

当汇编代码调用新增加的函数时，在 LordPE 的输入表编辑窗口选择相应的 DLL 文件，勾上“View always FirstThunk”选项（见图 19.7），则 ThunkRVA 项的值就是该函数在 IAT 中的 RVA 地址。调用时代码为：CALL [基址+ThunkRVA]。

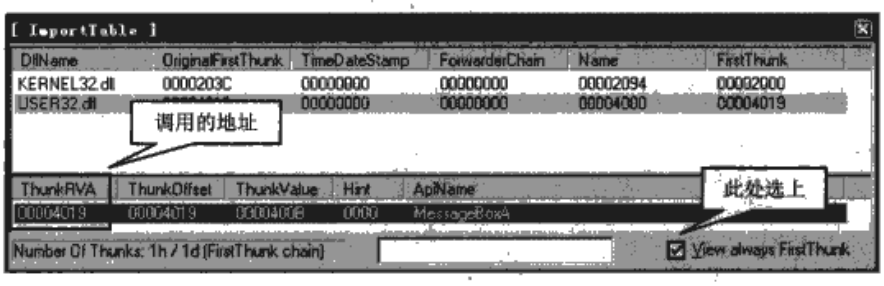


图 19.7　查看调用地址

MessageBox 在 USER32.DLL 用户模块中，其 ANSI 版是 MessageBoxA。根据参数调用的需要，在实例的.rdata 区块选择一空白处，构造两个参数所需要的字符串，如图 19.8 所示。

```
004020B0  48 65 6C 6C 6F 00 00 00  00 00 00 00  00 00 00 00  Hello...........
004020C0  50 45 44 49 59 00 00 00  00 00 00 00  00 00 00 00  PEDIY...........
```

图 19.8　构造字符串

构造好 MessageBoxA 的相关函数，然后用 call [404019]方式就可直接调用：

```
00401000  push    0                     ; /Style = MB_OK|MB_APPLMODAL
00401002  push    4020C0                ; |Title = "PEDIY"
00401007  push    4020B0                ; |Text = "Hello"
0040100C  push    0                     ; |hOwner = NULL
0040100E  call    dword ptr [404019]    ; \MessageBoxA
```

如果在 OllyDbg 里直接键入“call MessageBoxA”语句，程序也能运行，看起来没什么问题似的。代码如下：

```
0040100E  call    77D50702              ; USER32.MessageBoxA
```

此时直接调用了 MessageBoxA 函数的入口地址，称为硬编码。如果操作系统不同，或 USER32.DLL 版本不同，MessageBoxA 的地址就不一定是当前值，程序就会出错。所以，正确的做法必须在输入表里加入 MessageBoxA 函数，由 PE 加载器来获得函数的地址，再调用。

19.3.2 显式链接调用 DLL

在应用程序调用 DLL 中的函数之前，DLL 文件映像必须被映射到调用进程的地址空间中。有两种方法可以达到这一要求：一是加载时的隐含链接；二是运行期的显式链接。

“增加输入函数”一节讲述的就是隐含链接。显式链接是通过 LoadLibraryA(W)或 LoadLibraryExA(W)将 DLL 文件映像映射到调用进程的地址空间中，函数返回的 HINSTANCE 值用于标识文件映像映射到的虚拟内存地址。如果 DLL 文件已被映射到调用进程的地址空间里，则可以调用 GetModuleHandleA(W)函数获得 DLL 模块句柄。一旦 DLL 模块被加载，线程可以调用 GetProcAddress 函数获取输入函数的地址。LoadLibrary、GetProcAddress 等本身也是 API 函数，如果原输入表没有，就得在输入表里增加这些 API 函数；或参考一些病毒技术，在 kernel32.dll 里暴力搜索出这两个函数地址。

光盘映像文件中提供的实例 addapi.exe，已有 LoadLibraryA、GetProcAddress 函数，现利用这两个函数调用 MessageBox 函数显示一个对话框。根据这几个函数所需要的参数，在数据区构造出所需要的字符串，如图 19.9 所示。

```
004020B0 55 53 45 52 33 32 2E 64 6C 6C 00 00 00 00 00 00 USER32.dll........
004020C0 4D 65 73 73 61 67 65 42 6F 78 41 00 00 00 00 00 MessageBoxA.....
004020D0 48 65 6C 6C 6F 00 00 00 00 00 00 00 00 00 00 00 Hello...........
004020E0 50 45 44 49 59 00 00 00 00 00 00 00 00 00 00 00 PEDIY...........
```

图 19.9 构造字符串

首先调用 LoadLibraryA 函数获得 USER32.dll 基址，其结果返回到 eax 寄存器，再将结果放进堆栈，然后调用 GetProcAddress 函数获得 MessageBoxA 的地址，最后直接调用。

```
00401000  push    4020B0                ; /FileName = "USER32.dll"
00401005  call    dword ptr [402008]    ; \LoadLibraryA
0040100B  push    4020C0                ; /ProcNameOrOrdinal = "MessageBoxA"
00401010  push    eax                   ; |hModule
00401011  call    dword ptr [402004]    ; \GetProcAddress
00401017  push    0
00401019  push    4020E0                ; ASCII "PEDIY"
0040101E  push    4020D0                ; ASCII "Hello"
00401023  push    0
00401025  call    eax                   ; eax中是USER32.MessageBoxA地址，直接调用
```

本方法需要输入表中存在 LoadLibraryA、GetProcAddress 等函数；否则，必须在输入表中增加这些函数。

19.4 代码的重定位

在 PE 文件里，涉及直接寻址的指令都是需要重定位的。Windows 系统会尽量保证 EXE 程序加载到所需要的基址，因此重定位基本可以不考虑。不过对于 DLL 的动态链接库文件来说，Windows 系统没有办法保证每一次 DLL 运行时提供相同的基地址。这样修补 DLL 文件时，也必须提供“重定位”的代码，否则原程序中的代码可能无法正常运行。

19.4.1 修复重定位表

重定位信息是编译器生成的并保留在 PE 文件的重定位表里，如果程序里新增需要重定位的代码，则需要在重定位表里增加新的项。

在实例 CodeRloc.dll 的输出函数_DisplayTextA@0 里增加一段代码，当被调用时显示 MessageBoxA 窗口。用 LordPE 在 CodeRloc.dll 输入表里增加 MessageBoxA 的调用，其 ThunkRVA 为 C019h。用 Hiew 将如下代码数据修改到 CodeRloc.dll 文件里。代码如下：

```
00401010 CodeRloc.DisplayTextA
00401010  60              pushad                          ;保存现场寄存器
00401011  6A00            push    0
00401013  68105C4000      push    000405C10               ;指向字符串 PEDIY
00401018  68005C4000      push    000405C00               ;指向字符串 Hello!
0040101D  6A00            push    0
0040101F  FF1519C04000    call    dword ptr [0040C019]    ;调用 MessageBoxA 函数
00401025  61              popad                           ;恢复现场寄存器
00401026 33C0             xor     eax,eax
00401028 C3               retn
```

在扩充功能新增补丁代码时，必须保证一些重要的寄存器的值不能被破坏，如 ESI、EBP 等，简单的做法是用 pushad、popad 指令保存现场所有寄存器。

在没修复重定位表之前，运行加载 DLL，在笔者的系统中其被加载到基址 370000h 上，此时新增的代码情况如图 19.10 所示，由于代码没有重定位，因此继续运行程序将会崩溃。

```
00371010  60              pushad
00371011  6A 00           push    0
00371013  68 105C4000     push    405C10          需要重定位,指向375C10
00371018  68 005C4000     push    405C00          需要重定位,指向375C00
0037101D  6A 00           push    0
0037101F  FF15 19C04000   call    dword ptr [40C019]  需要重定位,指向37C019
00371025  61              popad
```

图 19.10　代码没重定位的情况

这段补丁代码有三处需要重定位，其中地址为 401013h 处就是其中一句需要重定位的语句，当基址是默认的 400000h 时，这句代码是正确的。当被加载的是其他基址时，401014h 所指向的数据需要重定位。归纳起来，需要重定位的三处地址（RVA）是 1014h、1019h、1021h。

重定位表以 1000h 大小为一个段，在 IMAGE_BASE_RELOCATION 结构中，VirtualAddress 的值就是 1000h 的整数倍。本例需要重定位的 1014h、1019h、1021h，可以放到 VirtualAddress 为 1000h 的组里，整理后的 TypeOffset 数据如表 19-1 所示。

表 19-1　计算重定位数据

项　目	重定位数据 1	重定位数据 2	重定位数据 3
需要重定位地址	1014h（RVA）	1019h（RVA）	1021h（RVA）
重定位项目值	1014h−1000h=14h	1019h−1000h=19h	1021h−1000h=21h
TypeOffset	14h or 3000h=3014h	19h or 3000h=3019h	3021h or 3000h=3021h

有两种方法将新增的 TypeOffset 放进重定位表结构中。第一种方法是在重定位表中，在 RVA 为 1000h 的索引中插入 TypeOffset，这种方法比较麻烦，插入新的数据后，其他数据要往后移动。第二种方法是新构造一个重定位表段（见图 19.11），追加到原重定位表后面。

VirtualAddress	SizeOfBlock	TypeOffset		
00001000	0000000E	3014	3019	3021

图 19.11　新增加的重定位数组

用十六进制工具，在原重定位后面追加新的重定位表数据上去，构造时，注意各数据按照内存存放规律放置——低地址放低字节，高地址放高字节，如图 19.12 所示。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
0000A500  DC 38 E0 38 E4 38 E8 38  EC 38 F0 38 F4 38 00 39  ???????.9
0000A510  9C 39 A0 39 00 10 00 00  0E 00 00 00 14 30 19 30  ?9 9........0.0
0000A520  21 30 00 00 00 00 00 00  00 00 00 00 00 00 00 00  !0..............
```

图 19.12　追加新的重定位表数据

最后，用 LordPE 将 Directories 中 Relocation 项的 Size 调整为 522h。单击 ... 按钮，以可视化方式显示新增的重定表数据，在图 19.13 中 Index 为 9 的就是新增的结构。

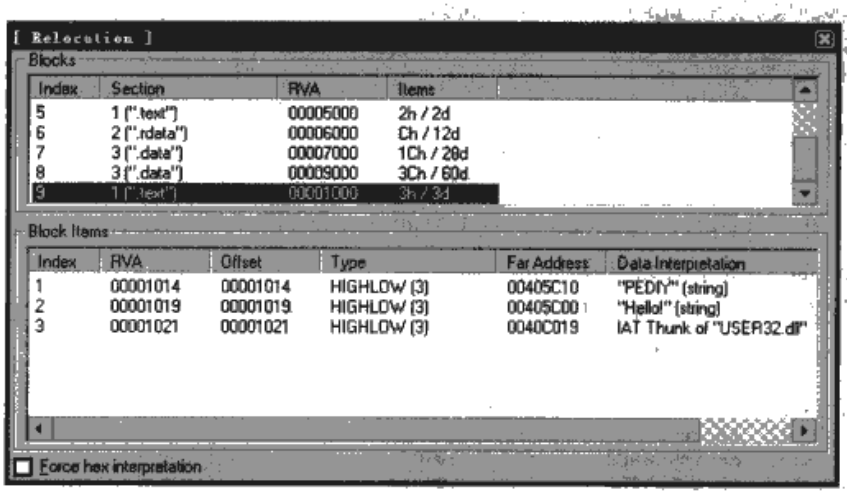


图 19.13　增加的重定位表数据

运行加载修复后的 CodeRloc.dll 文件，此时 DLL 被映射到内存 370000h 地址上，相关代码这次被重定位了，如图 19.14 所示。

```
00371010  60              pushad
00371011  6A 00           push   0
00371013  68 105C3700     push   375C10                ASCII "PEDIY"
00371018  68 005C3700     push   375C00                ASCII "Hello!"
0037101D  6A 00           push   0
0037101F  FF15 19C03700   call   dword ptr [37C019]    USER32.MessageBoxA
00371025  61              popad
```

图 19.14　已重定位后的代码

19.4.2　代码的自定位技术

不通过重定位表，也可实现代码的重定位，其原理是利用 CALL 指令执行时会将返回地址压入堆栈，然后用 POP 指令将这个返回地址取出，就可实现代码的自定位了。观察下面这段代码：

```
00401000    E8 00000000          call    00401005
00401005    5A                   pop     edx          ;执行这句后，ebp=401005h
00401006    83EA 05              sub     edx, 5       ;执行这句后，ebp=401000h
```

这段代码可以获得自身的地址，运行时 CALL 指令将返回地址 401005h 压入堆栈，下一句 POP 指令将返回地址取出放入 EDX，再接下去减去 CALL 指令长度 5，得到当前代码的地址。

在 16.3.4 节已讲解了这一技术在外壳上的利用，可以很巧妙地解决直接寻址的重定位问题。观察下面这段代码：

```
00401000    E8 00000000          call    00401005
00401005    5A                   pop     edx
00401006    81EA 05104000        sub     edx, 401005
0040100C    8B82 00104000        mov     eax, dword ptr [edx+401000]
```

用 POP 指令取出地址放入 EDX，通过 SUB 指令取得代码重定位的偏移，加上 EDX 就得到了变量的真实地址。

用 LordPE 在 CodeRloc.dll 输入表里增加 MessageBoxA 的调用，其 ThunkRVA 为 C019h。将如下代码数据输入到 CodeRloc.dll 文件里。代码如下：

```
00401010  60                    pushad
00401011  6A 00                 push    0                       ;/Style
00401013  E8 06000000           call    0040101E                ;|Title = "PEDIY"
00401018  50 45 44 49 59 00     ascii   "PEDIY",0
0040101E  E8 07000000           call    0040102A                ;|Text = "Hello"
00401023  48 65 6C 6C 6F 21 00  ascii   "Hello!",0
0040102A  6A 00                 push    0                       ;|hOwner = NULL
0040102C  E8 00000000           call    00401031
00401031  5A                    pop     edx
00401032  81EA 31104000         sub     edx, 401031
00401038  FF92 19C04000         call    dword ptr[edx+40C019] ;\MessageBoxA
0040103E  61                    popad
```

这段代码可以不需要重定位表，自身完成重定位功能。401013h 这句的“call 40101E”执行后，紧跟其下方的字符串“PEDIY”地址被压入堆栈了，相当于语句“push 401018”。MessageBoxA 用语句“call [edx+40C019]”调用，EDX 中保存的是重定位的偏移。

19.5 增加输出函数

一般情况下，若需要实现某个函数功能，可以重新写一个 DLL 文件，然后在 EXE 文件里调用该 DLL 的输出函数。但在特殊情况下，需要在现有 DLL 文件里增加输出函数。

在输出表中，各输出函数名之间必须按字母升序排列，否则 Windows 会报告“无法定位到相关 DLL 文件上”的错误。在新增输出函数时，必须注意这点。例如，有两个输出函数，第一个函数的第一个字母应该比第二个函数的第一个字母小；如果第一个字母一样，第一个函数的第二个字母应该比第二个函数的第二个字母小。

本例为 MenuLib.dll 增加一个 zounter()输出函数，假设将增加的函数执行代码放在.data 的 5B00h 处。

增加输出函数首先要了解输出表的结构，具体参考 PE 一章。原输出表 RVA 为 4920h，大小为 5Ah。输出表的 IMAGE_EXPORT_DIRECTORY 结构内容见表 19-2。

表 19-2　IMAGE_EXPORT_DIRECTORY 结构

Characteristics	TimeDateStamp	MajorVersion	MinorVersion	Name	Base
0000 0000	C9C6 EC3D	0000	0000	6049 0000	0100 0000
NumberOfFunctions	NumberOfNames	AddressOfFunctions	AddressOfNames	AddressOfNameOrdinals	
0200 0000	0200 0000	4849 0000	5049 0000	5849 0000	

现在准备追加一个输出函数上去，若在原 IED 结构上修改，很不方便。这里采用先增加输出函数名，然后再重新构造一个 IED 结构。用十六进制工具打开 MenuLib.dll，在原输出函数末尾处追加 zounter 字符串。将 4990h 处作为输出表的新地址，根据 IED 结构定义，依次构造各数据区，具体如图 19.15 所示。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F  .
00004950  68 49 00 00 71 49 00 00  00 00 01 00 4D 65 6E 75  hI..qI......Menu
00004960  4C 69 62 2E 64 6C 6C 00  4D 65 6E 75 4F 70 65 6E  Lib.dll.MenuOpen
00004970  00 4D 65 6E 75 53 61 76  65 00 43 6F 75 6E 74 65  .MenuSave.Counte
00004980  72 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  r...............
00004990  00 00 00 00 C9 C6 EC 3D  00 00 00 00 5C 49 00 00  ....蓉?....\I..
000049A0  01 00 00 00 03 00 00 00  03 00 00 00 C0 49 00 00  ............鐙..
000049B0  D0 49 00 00 E0 49 00 00  00 00 00 00 00 00 00 00  瑙..鄂..........
000049C0  10 10 00 00 70 11 00 00  00 5B 00 00 00 00 00 00  ....p....[......
000049D0  68 49 00 00 71 49 00 00  7A 49 00 00 00 00 00 00  hI..qI..zI......
000049E0  00 00 01 00 02 00 00 00  00 00 00 00 00 00 00 00  ................
```

图 19.15　重新构造输出表

其中，指向函数地址数组指针 AddressOfFunctions 为 49C0h，这里是 3 个输出函数的调用地址，在这里再补上新增的 zounter()输出函数调用地址 5B00h。指向函数名字的地址表的指针 AddressOfNames 为 49D0h，并将 zounter 字符串的地址 497Ah 填上。指向输出序列号数组的指针 AddressOfNameOrdinals 放在 49E0h 中，并增加一项序号“02”。新构建输出表的 IMAGE_EXPORT_DIRECTORY 结构内容见表 19-3。

表 19-3 IMAGE_EXPORT_DIRECTORY 结构

Characteristics	TimeDateStamp	MajorVersion	MinorVersion	Name	Base
0000 0000	C9C6 EC3D	0000	0000	5C49 0000	0100 0000
NumberOfFunctions	NumberOfNames	AddressOfFunctions	AddressOfNames	AddressOfNameOrdinals	
0300 0000	0300 0000	C049 0000	D049 0000	E049 0000	

用 LordPE 打开 MenuLib.dll，修正数据目录表里输出表的地址为 4990h。最后要做的就是在 5B00h 处增加 zounter 函数的代码。若数据需要重定位，则要修复重定位表，并将.data 属性设置为 E0000040h，表示该块可读、可写、可执行并包含已初始化的数据。

19.6 消息循环

Windows 应用程序的运行以消息为核心，每个窗口具有一个窗口函数，窗口函数从 Windows 接收消息，并检查每一条消息，根据这些消息完成特定的操作。也就是说，这里是程序的控制中心。本节主要是与大家探讨 Win32 SDK 写的程序，其他如 MFC 等消息处理比较复杂，不做深入探讨。

19.6.1 WndProc 函数

窗口函数被程序员们称为 WndProc，是一个消息处理回调函数，对消息进行判断和处理，一旦有消息产生，就会调用该函数。MFC 采用消息映射实现对消息的响应，将各种消息拐弯抹角地送到各个对应的 WndProc 函数，使得消息的处理更加隐蔽。

Windows 调用 WndProc 时，传递的参数有 4 个：hWnd 参数为窗口句柄，message 参数定义消息的类型，wParam 和 lParam 参数包含消息的附加信息。

光盘映像文件中提供的实例 pediy.exe 文件的窗口函数（WndProc）源码如下：

```
LRESULT CALLBACK WndProc (HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_CREATE :                // WINUSER.H 里定义：#define WM_CREATE 0001h
    ……
        case WM_COMMAND:                // #define WM_COMMAND  0x0111
        switch (LOWORD (wParam))
        {
            case IDM_APP_ABOUT: // 关于菜单的 ID，用 eXeScope 查看为 9C53h
                MessageBox (hwnd, TEXT ("逆向分析技术\n") ……) ;
                return 0 ;
        }
        break ;
    case WM_DESTROY:                // #define WM_DESTROY    0x0002h
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, message, wParam, lParam) ; // 系统默认的处理过程
}
```

WndProc 的主体是由一系列 case 语句组成的消息处理程序段，程序员只需根据窗口可能收到的消息在 case 语句中编写相应的处理代码即可。如果想为原程序增加功能，如增加菜单、按钮等功能，就必须在消息循环 WndProc 里插入相应的处理代码。

19.6.2 寻找消息循环

根据 Windows 程序处理方式，用合适的 API 下断，就能方便地定位到消息循环（WndProc）的处理代码。

1. 利用 RegisterClassA(W)或 RegisterClassEx A(W)函数

程序创建窗口之前，必须首先调用 RegisterClass 注册一个窗口类。该函数的参数是一个指向类型为 WNDCLASS 的结构指针，WNDCLASS 结构的第二个成员 lpfnWndProc 指向 WndProc。

RegisterClass 函数原型如下：

```
ATOM RegisterClass (
    CONST WNDCLASS *lpWndClass
    );
```

WNDCLASS 结构如下：

```
typedef struct _WNDCLASS {
    UINT      style;
    WNDPROC   lpfnWndProc;    //指向消息循环
    int       cbClsExtra;
    int       cbWndExtra;
    HANDLE    hInstance;
    HICON     hIcon;
    HCURSOR   hCursor;
    HBRUSH    hbrBackground;
    LPCTSTR   lpszMenuName;
    LPCTSTR   lpszClassName;
} WNDCLASS;
```

OllyDbg 加载实例 pediy.exe，用 RegisterClassA 设断，中断如下：

```
00401083  push    edx                              ; /pWndClass = 0012FF7C
00401084  call    dword ptr [402048]               ; \RegisterClassA
```

在数据窗口查看 RegisterClassA 的参数指向的 WNDCLASS 结构，如图 19.16 所示。WNDCLASS 结构中 lpfnWndProc 指向的值为 40112Eh，这就是 WndProc 地址。

图 19.16 查看 WNDCLASS 结构

如果用 IDA 反汇编分析，其能自动分析出 WNDCLASS 结构初始化过程。代码如下：

```
.text:00401023   sub     esp, 48h
.text:00401026   mov     [ebp+WndClass.style], 3
.text:0040102D   mov     [ebp+WndClass.lpfnWndProc], offset sub_40112E
.text:00401034   mov     [ebp+WndClass.cbClsExtra], 0
.text:0040103B   mov     [ebp+WndClass.cbWndExtra], 0
……
.text:00401080   lea     edx, [ebp+WndClass]
.text:00401083   push    edx                      ; lpWndClass
.text:00401084   call    ds:RegisterClassA
```

2. 利用 SendMessageA 函数

在某些程序中单击其菜单或按钮时，Windows 会调用 SendMessageA(W)函数发送一个 WM_COMMAND 消息给应用程序，该消息的 wParam 参数就是按钮或菜单的 ID 号。中断后，跟踪程序的处理过程就可发现 WndProc 处。

3. 利用菜单或按钮对话框

当单击菜单或按钮时，会弹出一些对话框，如果能设断拦截，也可找到 WndProc。

例如，单击 pediy.exe 关于窗口，会调用 MessageBoxA 函数打开版权窗口，用其设断也能来到 WndProc 消息循环里。

4. 利用系统消息循环

当 Windows 处理完 WndProc 后，会重新返回到系统中，此时跟进，会来到系统消息循环处理代码处。不同系统，这个地址不同，笔者的 Windows XP SP2 的代码如下：

```
77D1871A   or      byte ptr [eax+FB4], 1
77D18721   call    dword ptr [ebp+8]          ;调用 WndProc
77D18724   mov     ecx, dword ptr fs:[18]
77D1872B   and     byte ptr [ecx+FB4], 0
```

直接在系统消息循环处理中设置断点，如本例在 77D18721h 地址处下断，可以快速定位到 WndProc，很适合 MFC 多个消息循环的程序。如果程序比较大，消息循环比较多，可能会频繁中断，此时可以用 OllyDbg 的 log 记录功能，将所有的 WndProc 记录下来分析。

5. 利用 GetWindowLongA(W)函数

编程获取本进程内窗口的窗口过程用 GetWindowLong 实现很简单，直接调用：

```
lpfnWndProc = (WNDPROC)GetWindowLong(hWnd,GWL_WNDPROC)
```

19.6.3 WndProc 汇编形式

若在 WndProc 里设断，会发现 Windows 不停地调用该段代码，以处理程序的各类消息。Windows 调用 WndProc 传递的参数有 4 个：hWnd、message、wParam 和 lParam。WndProc 部分代码如下：

```
0040112E  push    ebp
0040112F  mov     ebp, esp
00401131  sub     esp, 8
00401134  mov     eax, dword ptr [ebp+C]
00401137  mov     dword ptr [ebp-4], eax
0040113A  cmp     dword ptr [ebp-4], 5
0040113E  ja      short 0040115B
00401140  cmp     dword ptr [ebp-4], 5        ;caseWM_SIZE
00401144  je      short 004011BE
00401146  cmp     dword ptr [ebp-4], 1        ;case WM_CREATE
0040114A  je      short 00401173
0040114C  cmp     dword ptr [ebp-4], 2        ;case WM_DESTROY
00401150  je      00401224
00401156  jmp     00401230
0040115B  cmp     dword ptr [ebp-4], 7        ;case WM_SETFOCUS
0040115F  je      short 004011AA
00401161  cmp     dword ptr [ebp-4], 111      ;case WM_COMMAND
   ......
004011F1  mov     edx, dword ptr [ebp+10]     ;wParam 值，即菜单的 ID 值
004011F4  and     edx, 0FFFF
004011FA  mov     dword ptr [ebp-8], edx
```

```
004011FD  cmp     dword ptr [ebp-8], 9C53       ;关于菜单的 ID
00401204  je      short 00401208                ;如果单击了关于菜单就跳过去处理
```

本例中，在 WndProc 代码里（40112Fh 以后）可以用下面的变量调用其参数：

```
LRESULT CALLBACK WndProc (
  HWND hwnd,                // [EBP+08], 窗口句柄
  UINT message,             // [EBP+0C], 消息的类型
  WPARAM wParam,            // [EBP+10], 消息的附加信息
  LPARAM lParam             // [EBP+14], 消息的附加信息
);
```

注意：由于各类程序传递参数的方式不一样，因此具体问题具体分析。例如，有些程序用 ESP 来传递参数。

程序编译方式不同，CASE 语句实现的代码略有不同。例如，按文件体积最小优化编译时一段样例代码如下：

```
00401106    push ebp                    ;WndProc 开始处
00401107    mov    ebp, esp
00401109    mov     eax,[ebp+0C]        ;EAX 中是 message 值
0040110C    dec     eax                 ;message-1
0040110D    dec     eax                 ;message-1
0040110E    jz      0040114A
00401110    sub     eax,0000010F        ;case WM_COMMAND (0x10F =0x111-0x1-0x1)
```

19.7 修改 WndProc 扩充功能

本节将为 pediy.exe 程序增加三个功能：一是使菜单“File/Exit”命令生效；二是具有打开文本文件功能；三是具有保存文本文件功能。

19.7.1 扩充 WndProc

如果想增加程序的菜单、按钮等功能，就在 WndProc 里加入新的消息判断和事件代码。修改时不得破坏原有的消息循环和堆栈平衡。

用 Visual C++、eXeScope 或资源黑客等资源编辑工具为 pediy.exe 增加 Open、Save 等菜单，各菜单的 ID 自定义值见表 19-4。

表 19-4　菜单 ID 值

菜　单	Exit	Open	Save	Help
ID 号	40005（9C45h）	40002（9C42h）	40003（9C43h）	40019(9C53h)

增加的菜单不得有重复 ID，否则这些菜单功能会一样，因为 Windows 是靠 ID 来判断用户单击的是哪个菜单。菜单项目建好后，但由于还没事件代码，因此执行菜单不会有反应。单击菜单后，Windows 会发送一个 WM_COMMAND 消息给应用程序，WM_COMMAND 消息的 wParam 参数就是菜单的 ID 值，应用程序判断消息的地方就是在窗口函数（WndProc）里。

WndProc 判断菜单 ID 的代码如下：

```
004011FA  mov     dword ptr [ebp-8], edx      ;[EBP-08]是菜单的 ID
004011FD  cmp     dword ptr [ebp-8], 9C53     ;9C53h 是“关于”菜单的 ID
00401204  je      short 00401208
00401206  jmp     short 00401232
```

这段代码就是判断“关于”菜单的代码。现在必须增加一段代码判断 Exit、Open 和 Save 菜单的 ID。程序在 401250h 后的空间没有被代码占用，因此将增加的代码放在此处。键入如下代码：

```
00401206   jmp 00401250                          ;跳到空白代码处
......
00401250   cmp dword ptr [ebp-08], 00009C45      ;Exit 菜单
00401257   je ????????                           ;Exit 事件处理代码
00401259   cmp dword ptr [ebp-08], 00009C42      ;Open 菜单
00401260   je 0040128D                           ;Open 事件处理代码
00401262   cmp dword ptr [ebp-08], 00009C43      ;Save 菜单
00401269   je 00401357                           ;Save 事件处理代码
0040126F   jmp 00401222                          ;返回系统默认处理过程
```

19.7.2 扩充 Exit 菜单功能

程序可调用 PostQuitMessage (0)函数退出，即 WM_DESTROY 消息的事件代码。程序中原有的退出代码如下：

```
00401224  push    0                              ;/ExitCode = 0
00401226  call    dword ptr [402038]             ;\PostQuitMessage
```

方案确定后，可以修改程序，让其直接跳到 PostQuitMessage (0)函数处。修改的代码如下：

```
00401250  cmp     dword ptr [ebp-8], 9C45        ;Exit 菜单
00401257  je      short 00401224                 ;调用 PostQuitMessage
```

19.7.3 扩充 Open 菜单功能

打开文件需要调用 VirtualAlloc 函数申请内存空间。Edit 控件编辑的最大文本是 64KB，因此本例中分配的缓冲区大小定为 64KB。

```
LPVOID VirtualAlloc(
   LPVOID lpAddress,                    // 待分配空间的起始地址，如为 NULL 则系统来分配
   DWORD dwSize,                        // 定义分配空间的大小，此处为 64 KB
   DWORD flAllocationType,              // 定义分配类型，此处为 MEM_COMMIT (0x1000)
   DWORD flProtect                      // 保护属性，此处为 PAGE_READWRITE (0x4)
  );
```

如果不需要内存，调用 VirtualFree 函数即可释放内存。

```
BOOL VirtualFree(
   LPVOID lpAddress,                    // 释放的地址
   DWORD dwSize,                        // 必须为 0
   DWORD dwFreeType                     // MEM_RELEASE (0x8000)
  );
```

分配内存后就调用打开文件的对话框，这就需要定义 OPENFILENAME 结构。现在需要一点空间存放 OPENFILENAME 结构。再次调用 VirtualAlloc 函数分配内存，大小为(sizeof(OPENFILENAME) = 76 bytes)。

因为申请的内存块的数据都是 0，所以只需初始化 OPENFILENAME 结构里的相关字段，其他字段用默认的 0 填充。OPENFILENAME 结构中需要初始化的项目如下：

```
typedef struct tagOFN {
  DWORD       lStructSize;     // 0x0  ;结构的长度，76 字节
  LPCTSTR     lpstrFilter;     // 0xC  ;过滤器，  *.txt.*.txt.*.*.*.*
  LPTSTR      lpstrFile;       // 0x1C ;重要！全路径的文件名缓冲区
  DWORD       nMaxFile;        // 0x20 ;文件名缓冲区大小，这里是 512 bytes (200h)
  DWORD       Flags;           // 0x34 ;标志，在这设 OFN_FILEMUSTEXIST (0x1000)
}
```

首先选定一个空间来存放 OPENFILENAME 结构的文件筛选字符串："*.txt.*.txt.*.*.*.*"。一般建议在各块的尾部找空隙，各块的起始部分有时好像是空的，但存在全局变量使用的可能性。本例选择在.rdata 块的 2D00h 处存放筛选字符串。为了保证程序能正常运行，用 LordPE 编辑.rdata 块，使 VSize=RSize=1000h。

初始化 OPENFILENAME 结构后，调用 GetOpenFileName 函数获得文件名。再调用 CreateFileA 函数打开文件，并用 ReadFile 将数据读出，接着调用 SetWindowTextA 函数将内存中的数据显示到文本框中。文本框的句柄可通过 CreateWindow 创建 Edit 控件的返回值获得。创建 Edit 编辑框的代码如下：

```
00401182  push    0                               ; |Height = 0
00401184  push    0                               ; |Width = 0
00401186  push    0                               ;  Y = 0
00401188  push    0                               ; |X = 0
0040118A  push    50B000C4                        ; |Style
0040118F  push    0                               ; |WindowName = NULL
00401191  push    0040303C                        ; |Class = "edit"
00401196  push    0                               ; |ExtStyle = 0
00401198  call    dword ptr [40201C]              ; \CreateWindowExA
0040119E  mov     dword ptr [403060], eax         ;返回句柄放到 [00403060]
```

本例用隐含链接方式调用 API 函数，因此用 LordPE 修改输入表以增加所需的输入函数，各 API 函数的具体调用地址如表 19-5 所示。

表 19-5　新增的输入函数

新增的函数名	调用地址	Hiew 里输入形式
comdlg32.dll!GetOpenFileNameA	405020h	call d,[405020]
kernel32.dll!VirtualAlloc	405099h	call d,[405099]
kernel32.dll!CreateFileA	40509Dh	call d,[40509D]
kernel32.dll!ReadFile	4050A1h	call d,[4050A1]
kernel32.dll!CloseHandle	4050A5h	call d,[4050A5]
kernel32.dll!VirtualFree	4050A9h	call d,[4050A9]
kernel32.dll!LoadLibraryA	4050ADh	call d,[4050AD]
kernel32.dll!GetProcAddress	4050B1h	call d,[4050B1]
user32.dll!SetWindowTextA	4050D5h	call d,[4050D5]

在 40128Dh 处键入如下代码：

```
0040128D pushad                          ; 保存现场寄存器，很重要
;-------------------------------------------------
; 申请内存空间存放打开的文件名
0040128E push 00000004                   ; PAGE_READWRITE
00401290 push 00001000                   ; MEM_COMMIT
00401295 push 00010000                   ; 64KB
0040129A push 00000000                   ; 由系统分配内存
0040129C call dword ptr [00405099]       ; VirtualAlloc 函数
004012A2 mov edi, eax                    ; 将申请空间的地址放在 edi 中
;-------------------------------------------------
; 申请内存放 OPENFILENAME 结构
004012A4 push 00000004                   ; PAGE_READWRITE
004012A6 push 00001000                   ; MEM_COMMIT
004012AB push 0000004C                   ; 76 bytes
004012AD push 00000000                   ; 由系统分配内存
004012AF call dword ptr [00405099]       ; VirtualAlloc 函数
004012B5 mov esi, eax                    ; 将申请空间的地址放在 esi 中
```

```
;-------------------------------------------------
; 初始化 OPENFILENAME 结构
004012B7 mov dword ptr [esi], 4C      ; lStructSize，结构大小
004012BD mov [esi+0C], 00402D00       ; lpstrFilter，筛选字符串
004012C4 mov dword ptr [esi+1C], edi  ; lpstrFile，全路径文件名缓冲区
004012C7 mov [esi+20], 00000200       ; nMaxFile，文件名缓冲区大小
004012CE mov [esi+34], 00001000       ; 标志，OFN_FILEMUSTEXIST
;-------------------------------------------------
; 用 GetOpenFileNameA 获得文件名
004012B5 push esi                     ; OPENFILENAME 结构地址
004012D6 call dword ptr [00405020]    ; GetOpenFileNameA 打开对话框
004012DC or eax, eax                  ; 单击对话框取消按钮，则 eax=0
004012DE je 00401332                  ; 跳到结束处理代码
;-------------------------------------------------
; CreateFileA 函数打开文件
004012E0 push 00000000                ; hTemplateFile
004012E2 push 00000080                ; FILE_ATTRIBUTE_NORMAL
004012E7 push 00000003                ; OPEN_EXISTING
004012E9 push 00000000                ; 默认的安全属性
004012EB push 00000001                ; FILE_SHARE_READ
004012ED push 80000000                ; GENERIC_READ
004012F2 push edi                     ; 打开的文件名
004012F3 call dword ptr [0040509D]    ; CreateFileA 函数
004012F9 cmp eax, -1                  ; 打开文件失败
004012FC je 00401332                  ; 跳到结束处理代码
004012FE mov dword ptr [esi], eax     ; 将句柄保存在 OPENFILENAME 空间
;-------------------------------------------------
; 用 ReadFile 读取数据到缓冲区中
00401300 push 00000000                ; overlapped 结构
00401302 mov ecx, esi                 ; 将 OPENFILENAME 的指针传给 ecx
00401304 add ecx, 00000004            ; 将 OPENFILENAME+4 处当缓冲区用
00401307 push ecx                     ; 此缓冲区作为存放读入的字节数
00401308 push 0000EA00                ; 要读入字节数，在这定为 60000
0040130D push edi                     ; 用于保存读入数据的缓冲区
0040130E push dword ptr [esi]         ; CreateFileA 打开文件的句柄
00401310 call dword ptr [004050A1]    ; ReadFile 函数
;-------------------------------------------------
; 关闭文件
00401316 push dword ptr [esi]         ; CreateFileA 打开文件的句柄
00401318 call dword ptr [004050A5]    ; CloseHandle 函数
0040131E mov eax, dword ptr [esi+04]; 读入的字节数
00401321 or eax, eax                  ; 字节数为 0
00401323 je 00401332                  ; 跳到结束处理代码
;-------------------------------------------------
; 将内存数据送到控件编辑框中
00401325 push edi                     ; 已读取文件的缓冲区地址
00401326 push dword ptr [00403060]    ; 控件编辑框句柄
0040132C call dword ptr [004050D5]    ; SetWindowTextA 函数
;-------------------------------------------------
; 结束处理代码
00401332 push 00008000                ; MEM_RELEASE
00401337 push 00000000
00401339 push esi                     ; OPENFILENAME 结构
0040133A call dword ptr [004050A9]       ; VirtualFree 函数
00401340 push 00008000                ; MEM_RELEASE
```

```
00401345  push 00000000
00401347  push edi                        ; 存放打开文件缓冲区
00401348  call dword ptr [004050A9]       ; VirtualFree 函数
;-------------------------------------------------
0040134E  popad                           ; 恢复现场，重要
0040134F  jmp 00401246                    ; 跳到 DefWindowProcA 下一行交系统代码处理
```

修改完毕后，建议在不同系统或硬件上测试一下程序的功能是否正常。因为在修改代码过程中，很可能会在某些地方考虑不周全，在不同平台上测试会将隐藏的问题暴露出来。

增加 Save 菜单的原理类似，关键是要懂得基本的 Win32 编程，在此就不重复讲述了。

19.8 增加接口

上一节给 pediy.exe 增加菜单功能的例子固然巧妙，但工作量太大，并且很容易出错。为什么不写个 DLL 来实现打开、保存的功能呢？如果将消息循环的代码扩展到 DLL 文件来处理，这样功能扩展性更好。如果有兴趣，还可实现打印功能。这样，只需要用少量汇编代码提供一个接口，功能的实现只需调用 DLL，非常的方便。

19.8.1 用 DLL 增加功能

写一个 DLL，增加 MenuOpen 和 MenuSave 两个输出函数，在 pediy.exe 里直接调用相关输出函数完成功能。

1. 创建 DLL 文件

读者可以用自己熟悉的语言创建 DLL 文件，如 ASM、C、Delphi 等。本例用 Visual C++创建 DLL，创建的 DLL 输出两个函数：MenuOpen 和 MenuSave（程序及源码见光盘映像文件）。

当 Visual C++以 stdacll 方式将 C 函数输出时，Microsoft 的编译器会改变函数的名字，设置一个前导下画线，再加上一个@符号做前缀，后随一个数字，表示作为参数传递给函数的字节数。例如，输出函数：

```
EXPORT BOOL CALLBACK  MenuOpen(HWND hWnd);
```

经编译后，会变成_MenuOpen@4。当然，此时可直接利用_MenuOpen@4 函数进行输出操作，但也可让编译器输出没有经过改名的函数。方法是为编程项目建立一个.def 文件，并在该.def 文件中加上类似于下面的 EXPORTS 节：

```
EXPORTS
        MenuOpen
```

当链接程序分析.def 文件时，发现_MenuOpen@4 和 MenuOpen 均被输出。由于这两个函数名是互相匹配的，因此链接程序使用 MenuOpen 的.def 文件名输出该函数。

2. 创建 DLL 文件

可以用隐含链接或显式链接方法调用输出函数。调用时需注意函数的调用约定，如 StdCall、C 调用等。本例是 StdCall 调用，函数内平衡堆栈。

用 LordPE 打开 pediy.exe 文件，增加 MenuOpen 和 MenuSave 两个输入函数，具体见表 19-6。

表 19-6 输入函数

功 能	函 数 名	调 用 地 址
打开文件	MenuOpen(HWND hWnd)	405022h
保存文件	MenuSave(HWND hWnd)	405026h

这两个函数的参数是 pediy.exe 的句柄。由上一节已知，句柄是通过[EBP+8]来传递的。因此键入如下代码：

```
00401259  cmp dword ptr [ebp-08], 9C42    ;Open 菜单
00401260  je 00401273
00401262  cmp dword ptr [ebp-08], 9C43    ;Save 菜单
00401269  je 00401282
0040126B  jmp 00401222
;-------------------------------------------------------
; 调用 MenuOpen(hWnd)函数
00401273  pushad                          ;保存现场环境
00401274  push [ebp+08]                   ;句柄入栈
00401277  call dword ptr [00405022]       ;调用 DLL 的 MenuOpen
0040127D  popad                           ;恢复现场环境
0040127E  jmp 00401222
;-------------------------------------------------------
; 调用 MenuSave(hWnd)函数
00401282  pushad                          ;保存现场环境
00401283  push [ebp+08]                   ;句柄入栈
00401286  call dword ptr [00405026]       ;调用 DLL 的 MenuSave
0040128C  popad                           ;恢复现场环境
0040128D  jmp 00401222
```

19.8.2 扩展消息循环

消息循环 WndProc 是程序的控制中心，上节是在汇编状态下扩展 WndProc 对消息的判断处理，更好的办法是将消息循环延伸到 DLL 里来处理，这样灵活性更高。

思路是：在调用原 WndProc 前，先转到 DLL 处理相关的事件，处理完再转到原 WndProc 执行。现在对 pediy.exe 程序 WndProc 开始处设断，代码如下：

```
0040112E  push    ebp                     ; WndProc，在此设断
0040112F  mov     ebp, esp
```

刚进入 WndProc 时的堆栈，如图 19.17 所示。

图 19.17 WndProc 入口堆栈情况

此时堆栈中有一个返回地址和 WndProc 的 4 个参数。由于是在此处扩展消息循环（MyWndProc），可以将堆栈中这 5 个值作为 MyWndProc 的参数。设计的 MyWndProc 原型如下：

```
void _cdecl MyWndProc (const DWORD reversed,HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

扩展的 MyWndProc 和 WndProc 相比，多了一个参数 reversed，这是为了直接用图 19.17 所示堆栈中的各参数。修改程序时，只需要在汇编状态下直接调用 MyWndProc()即可，不需要另传参数，比较简单。

MyWndProc 接管消息后，就可用高级语言来扩充各功能了。代码如下：

```
void _cdecl MyWndProc (const DWORD reversed,HWND hwnd, UINT message,
                       WPARAM wParam,LPARAM lParam)
{
    switch (message)
    {
        case WM_COMMAND:
            switch (LOWORD (wParam))
```

```
            {
              case 40002://Open
                 MenuOpen(hwnd);
                  break;
              case 40003://Save
                 MenuSave(hwnd);
                  break;
              case 40005://Exit
                 SendMessage (hwnd, WM_CLOSE, 0, 0) ;
                  break;
            }
       break ;
      }
}
```

将上述代码编写到一个 DLL 文件 peplug.dll 里，输出 MyWndProc 函数。让 pediy.exe 在调用自己的 WndProc 消息前，先调用 MyWndProc，完成后继续原消息处理。

用 LordPE 在 peplug.dll 输入表里增加 MyWndProc 的调用，其 ThunkRVA 为 5017h。修改 pediy.exe 程序如下：

```
0040112E  jmp     0040124E                ;原 WndProc 开始处，跳去执行 MyWndProc
00401133  nop
00401134  mov     eax, dword ptr [ebp+C]
```

跳到空白处，执行 MyWndProc()，然后再跳回原消息循环处理。

```
0040124E  call    dword ptr [405017]      ;扩展消息循环 peplug.MyWndProc
00401254  push    ebp                     ;这几行是原 WndProc 的代码，移到此处
00401255  mov     ebp, esp
00401257  sub     esp, 8
0040125A  jmp     00401134                ;跳回 WndProc 代码
```

读者也可直接修改 WndProc 起始地址，将其指向 MyWndProc()，再跳回 WndProc，这样补丁代码比较简洁。

```
0040102D  mov     [ebp+WndClass.lpfnWndProc], 40112E   ;改为 0040124E
```

MyWndProc()修改如下：

```
0040124E  call    dword ptr [405017]      ;扩展消息循环 peplug.MyWndProc
00401254  jmp     00401134                ;跳回 WndProc 代码
```

为程序打造接口，用 DLL 来扩展程序功能，灵活性大，维护十分方便。进行接口设计时，一定要注意堆栈平衡和不破坏原有寄存器。

附录 A 浮点指令

一般的汇编书里对浮点指令介绍得很少，因此本节简单地介绍一下浮点数。

1．浮点数据格式

在计算机中表达实数时要采用浮点数格式，Intel 80x87 遵循 IEEE 浮点格式标准。IEEE 浮点数的格式由符号位 S、指数部分 E 和尾数部分 M 三部分组成（见图 A.1）。

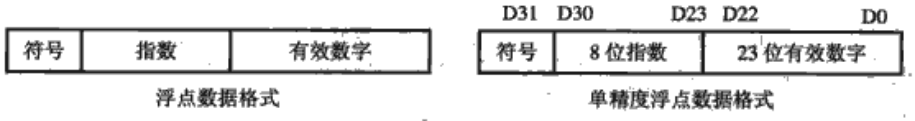


图 A.1　浮点数据格式

- 符号（Sign）：表示数据的正负，在最高有效位，负数的符号是 1，正数的符号是 0。
- 指数（Exponent）：或称阶码，表示数据以 2 为底的幂。指数采用偏移码表示，恒为整数。例如，单精度格式 8 位指数的偏移基数为 127，除去全 0、全 1 两个编码外，其余的 1～254 编码表示阶码数值-126～+127。
- 有效数字（Significand）：表示数据的有效位数，反映数据的精度。
  - 单精度浮点数据格式是 32 位，符号位占 1 位，E 占 8 位，M 占 23 位。
  - 扩展单精度格式，符号位占 1 位，E 大于或等于 11 位，M 占 31 位。
  - 双精度浮点数据格式是 64 位，符号位占 1 位，E 占 11 位，M 占 52 位。
  - 扩展精度浮点数据格式是 80 位，符号位占 1 位，E 大于或等于 15 位，M 大于 63 位。

（1）把浮点格式数据转换成实数表达式

设单精度浮点数的符号位是 s，指数是 e，有效数字是 x，则该浮点数的值用十进制表示为：

$= (-1)^{s} \times (1 + x) \times 2^{(e - 127)}$

例如：49E48E68 h=0100 1001 1110 0100 1000 1110 0110 1000 b

将它分成符号、指数和有效数字 3 部分后的结果为：

49E48E68 h=0　100 1001 1　110 0100 1000 1110 0110 1000 b

其中：

- 符号位是 0，即 s = 0。
- 指数部分是 100 1001 1，读成十进制就是 147，即 e = 147。
- 有效数字部分是 110 0100 1000 1110 0110 1000，也就是二进制的纯小数 0.110 0100 1000 1110 0110 1000，其十进制形式为 0.78559589385986328125，即 x = 0.78559589385986328125。

这样，该浮点数的十进制表示如下所示：

$=(-1)^{s}\times(1+x)\times2^{(e-127)}$

$=(-1)^{0}\times(1+0.78559589385986328125)\times2^{(147-127)}$

$=1872333$

（2）把实数转换成浮点格式

例如：100.25 = 0110 0100.01 b = 1.10010001 b×2^{6}

其中：

- 符号位= 0；
- 指数部分= 127+6= 133 = 10000101 b；
- 有效数字部分 = 10010001000000000000000 b；

这样，100.25 表示成单精度浮点数为：

0 10000101 10010001000000000000000 b = 42C88000 h

2. 浮点寄存器

组成浮点执行环境的寄存器主要是 8 个通用数据寄存器和几个专用寄存器，它们是状态寄存器、控制寄存器和标记寄存器。

（1）浮点数据寄存器

浮点处理单元有 8 个浮点数据寄存器（FPU），编号为 ST0～ST7，图 A.2 所示的是 OllyDbg 寄存器面板显示的浮点寄存器。每个浮点寄存器都是 80 位，并以扩展精度格式存储数据。8 个浮点寄存器组成首尾相接的堆栈，不采用随机存取，而是按照“后进先出”的堆栈原则工作，并且首尾循环，所以浮点寄存器常常称为浮点数据栈。

```
ST0 Valid 4.000000000000000000
ST1 empty 0.000000000045353280e-4933
ST2 empty 0.000000000000023290e-4933
ST3 empty 7.077798931489875331Oe-2505
ST4 empty -NAN FFFF 804E6760 804E3490
ST5 empty -UNORM 8A68 00000038 A820FC38
ST6 empty +UNORM 003B 0012FDC0 00000000
ST7 empty 16.000000000000000000
```

图 A.2 OllyDbg 寄存器面板显示的浮点寄存器

（2）浮点状态寄存器

浮点状态寄存器表明 FPU 当前的各种操作状态，每条浮点指令都对它进行修改以反映执行结果，其作用与整数处理单元的标志寄存器 EFLAGS 相当，如图 A.3 所示。

15	14	13~11	10	9	8	7	6	5	4	3	2	1	0
B	C3	TOP	C2	C1	C0	ES	SF	PE	UE	OE	ZE	DE	IE

图 A.3 浮点状态寄存器

C3～C0 是 4 位条件码标志，其中 C1 表示数据寄存器栈出现上溢或下溢；C3/C2/C0 是保存浮点比较指令的结果。

3. 浮点操作

这里介绍几个常用的浮点指令。

（1）取数指令

取数指令从存储器或浮点数据寄存器中取得数据，压入寄存器栈顶 st(0)。“压栈”操作致使栈顶指针

减 1，数据进入新的栈顶 st(0)，原来的 st(0)成为现在的 st(1)，原来的 st(1)为现在的 st(2)，依此类推。数据进入寄存器栈前由浮点处理单元自动转换成扩展精度浮点数。

- FLD m32r/m64r/m80r/ st(i)：取存储器或 st(i)中的浮点数，压入栈顶 st(0)；
- FILD m16i/m32i/m64i：取存储器的整数，压入栈顶 st(0)。

（2）存数指令

该组指令除执行存数功能外，还要弹出（pop）栈顶。“出栈”操作是将栈顶 st(0)清空，并使栈顶指针加 1，原来的 st(1)成为现在的 st(0)，原来的 st(2)为现在的 st(1)，依此类推。出栈指令的助记符都是用 P 结尾。

- FSTP m32r/m64r/m80r/ st(i)：st(0)按浮点格式存入存储器或 st(i)，然后出栈；
- FISTP m16i/m32i/m64i：st(0)按整数格式存入存储器，然后出栈。

（3）比较指令

浮点比较指令比较栈顶数据与栈顶的源操作数，比较结果通过浮点状态寄存器反映，见表 A-1。

表 A-1　比较指令的结果

比较结果	C3	C2	C0
st(0) > 源操作数	0	0	0
st(0) < 源操作数	0	0	1
st(0) = 源操作数	1	0	0
不可排序	1	1	1

- FCOM：浮点数比较，st(0)和 st(1)比较；
- FCOMP：浮点数比较，st(0)和 st(1)比较，并出栈；
- FICOM m16i/m32i：整数比较，st(0)与 m16i/m32i 比较。

在比较结果中，“不可排序”是指两个浮点格式数不能按照相对值进行比较排序的一种关系。由于浮点指令没有转移指令，所以需将比较结果 C3/C2/C0 转换到整数状态寄存器中，然后利用整数转移指令进行跳转。具体过程如下：

① 首先用“FSTSW AX”指令将浮点状态字存入整数寄存器 AX；

② 再通过“SAHF”指令将条件码传送到整数状态寄存器的低 4 位。SAHF 指令将包含 C3/C2/C0 的高字节送入整数状态字的低字节，正好对应 ZF/PF/CF；

③ 最后利用 jxx 指令进行条件判断转移（用无符号条件转移指令）。

汇编形式如下：

```
FCOM          ; 比较指令
FSTSW AX      ; 浮点状态寄存器送 AX
SAHF          ; AX 高字节转送到整数标志寄存器的低字节
JB ...        ; 小于，转移
```

4. 浮点指令汇总表

首先对下面的指令做一些说明：

- st(i)代表浮点寄存器，所说的出栈、入栈操作都是对 st(i)的影响；
- src, dst, dest, op 等表示指令操作数，src 表示源操作数，dst/dest 表示目的操作数；
- mem8, mem16, mem32, mem64, mem80 等表示内存操作数，后面的数值表示该操作数的内存位数（8 位为一字节）；
- x ← y 表示将 y 的值放入 x，例如，st(0) ← st(0)-st(1)表示将 st(0)-st(1)的值放入浮点寄存器 st(0)。

（1）数据传递和对常量的操作指令

指令格式	指令含义	执行的操作
FLD src	装入实数到 st(0)	st(0) ← src (mem32/mem64/mem80)
FILD src	装入整数到 st(0)	st(0) ← src (mem16/mem32/mem64)
FBLD src	装入 BCD 数到 st(0)	st(0) ← src (mem80)
FLDZ	将 0.0 装入 st(0)	st(0) ← 0.0
FLD1	将 1.0 装入 st(0)	st(0) ← 1.0
FLDPI	将π装入 st(0)	st(0) ← π
FLDL2T	将 $\log_2 10$ 装入 st(0)	st(0) ← $\log_2 10$
FLDL2E	将 $\log_2 e$ 装入 st(0)	st(0) ← $\log_2 e$
FLDLG2	将 $\log_{10} 2$ 装入 st(0)	st(0) ← $\log_{10} 2$
FLDLN2	将 $\log_e 2$ 装入 st(0)	st(0) ← $\log_e 2$
FST dest	保存实数 st(0)到 dest	dest ← st(0) (mem32/mem64)
FSTP dest		dest ← st(0) (mem32/mem64/mem80)，并出栈
FIST dest	将 st(0)以整数保存到 dest	dest ← st(0) (mem32/mem64)
FISTP dest		dest ← st(0) (mem16/mem32/mem64)，并出栈
FBST dest	将 st(0)以 BCD 保存到 dest	dest ← st(0) (mem80)
FBSTP dest		Dest ← st(0) (mem80)，并出栈

（2）比较指令

指令格式	指令含义	执行的操作
FCOM	浮点数比较：st(0)与 st(1)	将标志位设置为 st(0) - st(1)的结果标志位
FCOM op	浮点数比较	将标志位设置为 st(0) - op (mem32/mem64)的结果标志位
FICOM op	和整数比较	将 Flags 值设置为 st(0)-op 的结果 op (mem16/mem32)
FICOMP op	和整数比较	将 st(0)和 op 比较 op(mem16/mem32)后，并出栈
FTST	零检测	将 st(0) 和 0.0 比较
FUCOM st(i)	不可排序比较指令	比较 st(0) 和 st(i)
FUCOMP st(i)	不可排序比较指令	比较 st(0) 和 st(i)，并且执行一次出栈操作
FUCOMPP st(i)	不可排序比较指令	比较 st(0) 和 st(i)，并且执行两次出栈操作
FXAM	检测栈顶数据 st(0)	条件码 C1=0 为正数，C1=1 为负数

（3）运算指令

指令格式	指令含义	执行的操作
加法		
FADD	加实数	st(0) ← st(0) + st(1)
FADD src		st(0) ← st(0) + src (mem32/mem64)
FADD st(i),st		st(i) ← st(i) + st(0)
FADDP st(i),st		st(i) ← st(i) + st(0)，并出栈
FIADD src	加上一个整数	st(0) ← st(0) + src (mem16/mem32)
减法		
FSUB	减去一个实数	st(0) ← st(0) - st(1)
FSUB src		st(0) ← st(0) - src (reg/mem)

续表

指令格式	指令含义	执行的操作
FSUB st(i),st		st(i) ← st(i) - st(0)
FSUBP st(i),st		st(i) ← st(i) - st(0)，并出栈作
FSUBR st(i),st	用一个实数来减	st(i) ← st(0) - st(i)
FSUBRP st(i),st		st(i) ← st(0) - st(i)，并出栈
FISUB src	减去一个整数	st(0) ← st(0) - src (mem16/mem32)
FISUBR src	用一个整数来减	st(0) ← src - st(0) (mem16/mem32)
乘法		
FMUL	乘以一个实数	st(0) ← st(0)×st(1)
FMUL st(i)		st(0) ← st(0)×st(i)
FMUL st(i),st		st(i) ← st(0)×st(i)
FMULP st(i),st		st(i) ← st(0)×st(i)，并出栈
FIMUL src	乘以一个整数	st(0) ← st(0)×src (mem16/mem32)
除法		
FDIV	除以一个实数	st(0) ← st(0)÷st(1)
FDIV st(i),st(0)		st(i) ← st(0)÷st(i)
FDIVP st(i),st(0)		st(i) ← st(i)÷st(0)，并出栈
FIDIV src	除以一个整数	st(0) ← st(0)÷src (mem16/mem32)
FDIVR st(i),st	用实数除	st(0) ← st(0)÷st(i)
FDIVRP st(i),st		st(i) ← st(0)÷st(i)，并出栈
FIDIVR src	用整数除	st(0) ← src÷st(0) (mem16/mem32)
FSQRT	平方根	st(0) ← sqrt st(0)
FSCALE	2 的 st(0)次方	st(0) ← st(0)×$2^{st(1)}$
FXTRACT	取指数和有效数	将 st(0)的指数部分存于原数据寄存器，将原 st(0)有效数字压入栈顶
FPREM	取余数	st(0) ←st(0) MOD st(1)
FPREM1	取余数（IEEE 标准），st(0)←st(1)÷st(0)的余数，余数的符号与原来栈顶数据的符号一样	
FRNDINT	取整（四舍五入）	st(0) ← INT(st(0))
FABS	求绝对值	st(0) ←|st(0)|
FCHS	改变符号位（求负数）	st(0) ←st(0)
F2XM1	计算 2^x-1	st(0) ← $2^{st(0)}-1$
FYL2X	计算以 2 为基数的对数	st(1) ←st(1)×$\log_2$(st(0))，并出栈
FCOS	余弦函数 cos	st(0) ← COS(st(0))
FPTAN	正切函数 tan	st(0) ← TAN(st(0))，并将 1.0 压入栈顶
FPATAN	反正切函数 arctan	st(1) ← ARCTAG(st(1) / st(0))
FSIN	正弦函数 sin	st(0) ← SIN(st(0))
FSINCOS	sincos 函数	st(0) ← SIN(st(0))，并且压入 st(1) st(0) ← COS(st(0))

续表

指令格式	指令含义	执行的操作
FYL2XP1	计算 st(0)接近 0 的对数值	st(1) ← st(1)×$\log_2$(st(0)+1.0)，并出栈
处理器控制指令		
FINIT	初始化 FPU	
FSTSW AX	保存状态字的值到 AX	AX← MSW
FSTSW dest	保存状态字的值到 dest	dest←MSW (mem16)
FLDCW src	从 src 装入 FPU 的控制字	FPU CW ←src (mem16)
FSTCW dest	将 FPU 的控制字保存到 dest	dest← FPU CW
FCLEX	清除异常	
FSTENV dest	保存环境到内存地址 dest 处，保存状态字、控制字、标志字和异常指针的值	
FLDENV src	从主存取出 14/28 个字节数据，设置 FPU 的环境	
FSAVE dest	检测和处理未屏蔽的错误，将全部状态存入主存，然后初始化 FPU	
FRSTOR src	从 src 处装入由 FSAVE 保存的 FPU 状态	
FINCSTP	堆栈指针 ST 加 1；如果 TOP=7，则增量后为 0	st(6) ←st(5); st(5) ←st(4),...,st(0) ←?
FDECSTP	堆栈指针 ST 减 1；如果 TOP=0，则减量为 7	st(0) ←st(1); st(1) ←st(2),...,st(7) ←?
FFREE st(i)	标志寄存器 st(i)未被使用	
FNOP	空操作，等同 CPU 的 nop	st(0) ←st(0)
WAIT/FWAIT	同步 FPU 与 CPU；停止 CPU 的运行，直到 FPU 完成当前操作码	
FXCH	交换指令，交换 st(0)和 st(1)的值	st(0) ←st(1) st(1) ← st(0)

附录 B

在 Visual C++中使用内联汇编

使用内联汇编可以在 C/C++代码中嵌入汇编语言指令，而且不需要额外的汇编和连接步骤。在 Visual C++中，内联汇编是内置的编译器，因此不需要配置诸如 MASM 一类的独立汇编工具。这里以 Visual Studio .NET 2003 为背景，介绍在 Visual C++中使用内联汇编的相关知识（如果是早期的版本，可能会有些出入）。

内联汇编代码可以使用 C/C++变量和函数，因此它能非常容易地整合到 C/C++代码中。它能做一些对于单独使用 C/C++来说非常笨重或不可能完成的任务。

内联汇编的用途包括：

- 使用汇编语言编写特定的函数；
- 编写对速度要求非常较高的代码；
- 在设备驱动程序中直接访问硬件；
- 编写 naked 函数的初始化和结束代码。

1. 关键字

使用内联汇编要用到__asm 关键字，它可以出现在任何允许 C/C++语句出现的地方。先来看一些例子。

- 简单的__asm 块：

```
__asm
{
    MOV AL, 2
    MOV DX, 0xD007
    OUT AL, DX
}
```

- 在每条汇编指令之前加__asm 关键字：

```
__asm MOV AL, 2
__asm MOV DX, 0xD007
__asm OUT AL, DX
```

- 因为__asm 关键字是语句分隔符，所以可以把多条汇编指令放在同一行：

```
__asm MOV AL, 2  __asm MOV DX, 0XD007  __asm OUT AL, DX
```

显然，第一种方法与 C/C++的风格很一致，并且把汇编代码和 C/C++代码清楚地分开，还避免了重复输入__asm 关键字，因此推荐使用第一种方法。

不像在 C/C++中的“{}”，__asm 块的“{}”不会影响 C/C++变量的作用范围。同时，__asm 块可以嵌套，而且嵌套也不会影响变量的作用范围。

为了与低版本的 Visual C++兼容，_asm 和__asm 具有相同的意义。另外，Visual C++支持标准 C++的 asm 关键字，但是它不会生成任何指令，它的作用仅限于使编译器不会出现编译错误。要使用内联汇编，必须使用__asm 而不是 asm 关键字。

2. 汇编语言

（1）指令集

内联汇编支持 Intel Pentium 4 和 AMD Athlon 的所有指令。更多其他处理器的指令可以通过 EMIT 伪指令来创建（_EMIT 伪指令说明见下文）。

（2）MASM 表达式

在内联汇编代码中，可以使用所有的 MASM 表达式（MASM 表达式是指用来计算一个数值或一个地址的操作符和操作数的组合）。

（3）数据指示符和操作符

虽然__asm 块中允许使用 C/C++的数据类型和对象，但它不能使用 MASM 指示符和操作符来定义数据对象。这里特别指出，__asm 块中不允许 MASM 中的定义指示符（DB、DW、DD、DQ、DT 和 DF），也不允许使用 DUP 和 THIS 操作符。MASM 中的结构和记录也不再有效，内联汇编不接受 STRUC、RECORD、WIDTH 或者 MASK。

（4）EVEN 和 ALIGN 指示符

尽管内联汇编不支持大多数 MASM 指示符，但它支持 EVEN 和 ALIGN。当需要的时候，这些指示符在汇编代码里面加入 NOP 指令（空操作），使标号对齐到特定边界。这样可以使某些处理器取指令时具有更高的效率。

（5）MASM 宏指示符

内联汇编不是宏汇编，不能使用 MASM 宏指示符（MACRO、REPT、IRC、IRP 和 ENDM）和宏操作符（<>、!、&、%和.TYPE）。

（6）段

必须使用寄存器而不是名称来指明段（段名称“_TEXT”是无效的），并且，段跨越必须显式地说明，如 ES:[EBX]。

（7）类型和变量大小

在内联汇编中，可以用 LENGTH、SIZE 和 TYPE 来获取 C/C++类型和变量的大小。

- LENGTH 操作符用来取得 C/C++中数组的元素个数（如果不是一个数组，则结果为 1）。
- SIZE 操作符可以获取 C/C++变量的大小（一个变量的大小是 LENGTH 和 TYPE 的乘积）。
- TYPE 操作符可以返回 C/C++类型和变量的大小（如果变量是一个数组，它得到的是数组中单个元素的大小）。

例如，程序中定义了一个 8 维的整数型变量：

```
int iArray[8];
```

下面是 C 和汇编表达式中得到的 iArray 及其元素的相关值。

__asm	C	Size
LENGTH iArray	sizeof(iArray)/sizeof(iArray[0])	8
SIZE iArray	sizeof(iArray)	32
TYPE iArray	sizeof(iArray[0])	4

（8）注释

内联汇编中可以使用汇编语言的注释，即“;”。例如：

```
__asm MOV EAX, OFFSET pbBuff          ; 将 pbBuff 的地址载入到 EAX
```

因为C/C++宏将会展开到一个逻辑行中，为了避免在宏中使用汇编语言注释带来的混乱，内联汇编也允许使用C/C++风格的注释。

（9）_EMIT伪指令

_EMIT伪指令相当于MASM中的DB，但是_EMIT一次只能在当前代码段（.text段）中定义一个字节。例如：

```
__asm
{
    JMP _CodeLabel

    _EMIT 0x00                          ; 定义混合在代码段的数据
    _EMIT 0x01

_CodeLabel:                             ; 这里是代码
    _EMIT 0x90                          ; NOP 指令
}
```

（10）寄存器使用

一般来说，不能假定某个寄存器在__asm块开始的时候有已知的值。寄存器的值将不能保证会从__asm块保留到另外一个__asm块中。

如果一个函数声明为__fastcall调用方式，则其参数将通过寄存器而不是堆栈来传递。这将会使__asm块产生问题，因为函数无法被告知哪个参数在哪个寄存器中。如果函数接收了EAX中的参数并立即储存一个值到EAX中的话，原来的参数将丢失掉。另外，在所有声明为__fastcall的函数中，ECX寄存器是必须一直保留的。为了避免以上的冲突，包含__asm块的函数不要声明为__fastcall调用方式。

提示：如果使用EAX、EBX、ECX、EDX、ESI和EDI寄存器，则不需要保存它。但如果用到了DS、SS、SP、BP和标志寄存器，那就应该用PUSH保存这些寄存器。如果程序中改变了用于STD和CLD的方向标志，必须将其恢复到原来的值。

3. 使用C/C++元素

（1）可用的C/C++元素

C/C++与汇编语言可以混合使用，在内联汇编中可以使用C/C++变量以及很多其他的C/C++元素，包括：

- 符号，包括标号、变量和函数名；
- 常量，包括符号常量和枚举型成员；
- 宏定义和预处理指示符；
- 注释，包括“/**/”和“//”；
- 类型名，包括所有MASM中合法的类型；
- typedef名称，通常使用PTR和TYPE操作符，或者使用指定的结构或枚举成员。

在内联汇编中，可以使用C/C++或汇编语言的基数计数法。例如，0x100和100H是相等的。

（2）操作符使用

内联汇编中不能使用诸如“<<”一类的C/C++操作符。但是，C/C++和MASM共有的操作符（比如“*”和“[]”操作符），都被认为是汇编语言的操作符，是可以使用的。举个例子：

```
int iArray[10];
__asm MOV iArray[6], BX             ; 把 BX 的值传送到 iArray + 6 的地址处
iArray[6] = 0;                      // 把 iArray + 12 地址处的值赋 0
```

在内联汇编中，可以使用TYPE操作符使其与C/C++一致。比如，下面两条语句是一样的：

```
__asm MOV iArray[6 * TYPE int], 0     ; 把 iArray + 12 地址处的值赋 0
iArray[6] = 0;                        // 把 iArray + 12 地址处的值赋 0
```

（3）C/C++符号使用

在__asm 块中可以引用所有在作用范围内的 C/C++符号，包括变量名称、函数名称和标号。但是不能访问 C++类的成员函数。

下面是在内联汇编中使用 C/C++符号的一些限制。

- 每条汇编语句只能包含一个 C/C++符号。在一条汇编指令中，多个符号只能出现在 LENGTH、TYPE 或 SIZE 表达式中。
- 在__asm 块中引用函数必须先声明；否则，编译器将不能区别__asm 块中的函数名和标号。
- 在__asm 块中不能使用对于 MASM 来说是保留字的 C/C++符号（不区分大小写）。MASM 保留字包含指令名称（如 PUSH）和寄存器名称（如 ESI）等。
- 在__asm 块中不能识别结构和联合标签。

（4）访问 C/C++中的数据

内联汇编的一个非常大的方便之处是它可以使用名称来引用 C/C++变量。例如，如果 C/C++变量 iVar 在作用范围内：

```
__asm MOV EAX, iVar                   ; 将 iVar 的值传送到 EAX
```

如果 C/C++中的类、结构或者枚举成员具有唯一的名称，则在__asm 块中可以只通过成员名称来访问（省略“.”操作符之前的变量名或 typedef 名称）。然而，如果成员不是唯一的，你必须在“.”操作符之前加上变量名或 typedef 名称。例如，下面的两个结构都具有 SameName 这个成员变量：

```
struct FIRST_TYPE
   {
     char *pszWeasel;
     int SameName;
   };
```

```
struct SECOND_TYPE
{
    int iWonton;
    long SameName;
};
```

如果按下面方式声明变量：

```
struct FIRST_TYPE ftTest;
struct SECOND_TYPE stTemp;
```

那么，所有引用 SameName 成员的地方都必须使用变量名，因为 SameName 不是唯一的。另外，由于上面的 pszWeasel 变量具有唯一的名称，你可以仅仅使用它的成员名称来引用它。

```
__asm
{
    MOV EBX, OFFSET ftTest
    MOV ECX, [EBX]ftTest.SameName     ; 必须使用“ftTest”
    MOV ESI, [EBX]. pszWeasel         ; 可以省略“ftTest”
}
```

提示：省略变量名仅仅是为了书写代码方便，生成的汇编指令还是一样的。

（5）用内联汇编写函数

如果用内联汇编写函数的话，要传递参数和返回一个值都是非常容易的。看下面的例子，比较一下用独立汇编和内联汇编写的函数。

```
; PowerAsm.asm
; Compute the power of an integer
 PUBLIC     GetPowerAsm
 _TEXT      SEGMENT WORD PUBLIC 'CODE'
 GetPowerAsm PROC
    PUSH    EBP                     ; 保存 EBP
    MOV     EBP, ESP                ; 把 ESP 的值传给 EBP，以方便我们引用堆栈参数
    MOV     EAX, [EBP+4]            ; 取第一个参数
    MOV     ECX, [EBP+6]            ; 取第二个参数
    SHL     EAX, CL                 ; EAX = EAX * (2 ^ CL)
    POP     EBP                     ; 恢复 EBP
    RET                             ; 返回
 GetPowerAsm  ENDP
 _TEXT      ENDS
 END
```

C/C++函数一般用堆栈来传递参数，所以上面的函数中需要通过堆栈位置来访问它的参数（在MASM或其他一些汇编工具中，也允许通过名称来访问堆栈参数和局部堆栈变量）。

下面的程序是使用内联汇编写的。

```
// PowerC.c
#include <Stdio.h>
int GetPowerC(int iNum, int iPower);
int main()
{
   printf("3 times 2 to the power of 5 is %d\n", GetPowerC( 3, 5));
}
   int GetPowerC(int iNum, int iPower)
{
    __asm
    {
        MOV EAX, iNum               ; 取第一个参数
        MOV ECX, iPower             ; 取第二个参数
        SHL EAX, CL                 ; EAX = EAX * (2 to the power of CL)
    }
     // 返回值在 EAX 中
}
```

使用内联汇编写的GetPowerC函数可以通过参数名称来引用它的参数。由于GetPowerC函数没有执行C的return语句，所以编译器会给出一个警告信息，可以通过#pragma warning禁止生成这个警告。

内联汇编的其中一个用途是编写naked函数的初始化和结束代码。对于一般的函数，编译器会自动帮我们生成函数的初始化（构建参数指针和分配局部变量等）和结束代码（平衡堆栈和返回一个值等）。使用内联汇编，可以自己编写干干净净的函数。当然，此时必须自己动手做一些有关函数初始化和扫尾的工作。例如：

```
 void __declspec(naked) MyNakedFunction()
{
  // 为 naked 函数提供初始化代码
   __asm
      {
        PUSH EBP
        MOV ESP, EBP
```

```
            SUB ESP, __LOCAL_SIZE
        }
        ...
        // 函数结束代码
    __asm
        {
            POP EBP
            RET
        }
}
```

（6）调用 C/C++函数

内联汇编中调用声明为__cdecl 方式（默认）的 C/C++函数必须由调用者清除参数堆栈。下面是一个调用 C/C++函数的例子。

```
#include <Stdio.h>
char szFormat[] = "%s %s\n";
char szHello[] = "Hello";
char szWorld[] = " world";

void main()
{
    __asm
        {
            MOV     EAX, OFFSET szWorld
            PUSH    EAX
            MOV     EAX, OFFSET szHello
            PUSH    EAX
            MOV     EAX, OFFSET szFormat
            PUSH    EAX
            CALL    printf
            // 压入了 3 个参数在堆栈中，调用函数之后要调整堆栈
            ADD     ESP, 12
        }
}
```

提示：参数是按从右往左的顺序压入堆栈的。

如果调用__stdcall 方式的函数，则不需要自己清除堆栈。因为这种函数的返回指令是 RETn，会自动清除堆栈。大多数 Windows API 函数均为__stdcall 调用方式（仅除 wsprintf 等几个之外）。下面是一个调用 MessageBox 函数的例子。

```
#include <Windows.h>
TCHAR g_tszAppName[] = TEXT("API Test");

void main()
{
    TCHAR tszHello[] = TEXT("Hello, world!");
    __asm
        {
            PUSH    MB_OK OR MB_ICONINFORMATION
            PUSH    OFFSET g_tszAppName      ; 全局变量用 OFFSET
            LEA     EAX, tszHello            ; 局部变量用 LEA
            PUSH    EAX
            PUSH    0
```

```
        ; 注意这里不是 CALL MessageBox，而是调用重定位过的函数地址
        CALL    DWORD PTR [MessageBox]
    }
}
```

提示： 参数可以不受限制地访问 C++ 成员变量，但是不能访问 C++ 的成员函数。

（7）定义__asm 块为 C/C++宏

使用 C/C++宏可以方便地把汇编代码插入到源代码中。但是这其中需要额外地注意，因为宏将会扩展到一个逻辑行中。为了不会出现问题，请按以下规则编写宏。

- 使用花括号把__asm 块包围住；
- 把__asm 关键字放在每条汇编指令之前；
- 使用经典 C 风格的注释（“/* comment */”），不要使用汇编风格的注释（“; comment”）或单行的 C/C++注释（“// comment”）。

举个例子，下面定义了一个简单的宏。

```
#define PORTIO __asm        \
/* Port output */           \
{                           \
    __asm MOV AL, 2         \
    __asm MOV DX, 0xD007    \
    __asm OUT DX, AL        \
}
```

乍一看，后面的 3 个__asm 关键字好像是多余的。其实它们是需要的，因为宏将被扩展到一个单行中：

```
__asm /* Port output */ {__asm MOV AL, 2  __asm MOV DX, 0xD007  __asm OUT DX, AL}
```

从扩展后的代码中可以看出，第 3 个和第 4 个__asm 关键字是必需的（作为语句分隔符）。在__asm 块中，只有__asm 关键字和换行符会被认为是语句分隔符，又因为定义为宏的一个语句块会被认为是一个逻辑行，所以必须在每条指令之前使用__asm 关键字。

括号也是需要的，如果省略了它，编译器将不知道汇编代码在哪里结束，__asm 块后面的 C/C++语句看起来会被认为是汇编指令。

同样是由于宏展开的原因，汇编风格的注释（“; comment”）和单行的 C/C++注释（“// commen”）也可能会出现错误。为了避免这些错误，在定义__asm 块为宏时请使用经典 C 风格的注释（“/* comment */”）。

和 C/C++宏一样，__asm 块写的宏也可以拥有参数。和 C/C++宏不一样的是，__asm 宏不能返回一个值，因此，不能使用这种宏作为 C/C++表达式。

不要不加选择地调用这种类型的宏。比如，在声明为__fastcall 的函数中调用汇编语言宏可能会导致不可预料的结果（请参看前文的说明）。

（8）跳转

可以在 C/C++里面使用 goto 转跳到__asm 块中的标号处，也可以在__asm 块中转跳到__asm 块里面或外面的标号处。__asm 块内的标号是不区分大小写的（指令、指示符等也是不区分大小写的）。例如：

```
void MyFunction()
{
  goto C_Dest;      /* 正确 */
  goto c_dest;      /* 错误 */
  goto A_Dest;      /* 正确 */
  goto a_dest;      /* 正确 */
  __asm
  {
```

```
        JMP C_Dest      ; 正确
        JMP c_dest      ; 错误
        JMP A_Dest      ; 正确
        JMP a_dest      ; 正确
      a_dest:           ; __asm 标号
    }
  C_Dest:               /* C/C++ 标号 */
    return;
}
```

不要使用函数名称当作标号，否则将转跳到函数中执行，而不是标号处。例如，由于 exit 是 C/C++的函数，下面的转跳将不会到 exit 标号处：

```
; 错误：使用函数名作为标号
    JNE exit
    …
exit:
    …
```

美元符号“$”用于指定当前指令位置，常用于条件跳转中。例如：

```
    JNE $+5      ; 下面这条指令的长度是 5 个字节
    JMP _Label
    NOP          ; $+5，转跳到了这里
    …
  _Label:
    …
```

4. 在 Visual C++工程中使用独立汇编

内联汇编代码不易于移植，如果程序打算在不同类型的机器（比如 x86 和 Alpha）上运行，可能需要在不同的模块中使用特定的机器代码。这时候可以使用 MASM（Microsoft Macro Assembler），因为 MASM 支持更多方便的宏指令和数据指示符。

这里简单介绍一下在 Visual Studio .NET 2003 中调用 MASM 编译独立汇编文件的步骤。

在 Visual C++工程中，添加按 MASM 的要求编写的.asm 文件。在解决方案资源管理器中，右键单击这个文件，选择“属性”菜单项，在属性对话框中，点击“自定义生成步骤”，设置如下项目：

- 命令行：ML.exe /nologo /c /coff "-Fo$(IntDir)\$(InputName).obj" "$(InputPath)"
- 输出：$(IntDir)\$(InputName).obj

如果要生成调试信息，可以在命令行中加入“/Zi”参数，还可以根据需要生成.lst 和.sbr 文件。

如果要在汇编文件中调用 Windows API，可以从网上下载 MASM32 包（包含了 MASM 汇编工具、非常完整的 Windows API 头文件/库文件、实用宏以及大量的 Win32 汇编例子等）。相应地，应该在命令行中加入“/I X:\MASM32\INCLUDE”参数指定 Windows API 汇编头文件（.inc）的路径。MASM32 的主页是：http://www.masm32.com，里面可以下载最新版本的 MASM32 包。

术 语 表

Anti_Debug	反调试
Anti_Dump	反转存
Anti_Loader	反载入
Anti_Trace	反跟踪
API	Application Programming Interface，应用程序编程接口
ASCII	美国信息交换标准码，ASCII 码
Buffer Overflows	缓冲区溢出
CLR	Common Language Runtime，通用语言运行时
CrackMe	一些公开给他人调试解密的小程序
CRC	Cyclic Redundancy Checksum，循环冗余校验码
DDK	Device Drivers Kit，设备驱动程序开发包
Decompiler	反编译，将程序还原成高级语言的原始结构
Disassembler	反汇编，将机器语言转化成汇编语言
DPL	描述符特权级别
Dump	抓取内存镜像数据保存到磁盘
EP	Entry Point，文件执行时的入口点
Exploit	漏洞利用程序
ExploitMe	一种有漏洞的小程序，用来练习 Exploit 技术
Export Table	输出表，导出表，引出表
File Offset	磁盘文件偏移地址
GDT	Global Descriptor Table，全局描述符表
GUI	图形视窗程序
Handle	句柄
Hash	哈希算法，单向散列函数算法
Heap	堆
IAT	Import Address Table，输入地址表，导入地址表
IDT	中断描述符表
IID	输入表的 IMAGE_IMPORT_DESCRIPTOR 结构
ImageBase	基址，基地址
Import Table	简称 IT，输入表，导入表，引入表
Inline Patch	同 SMC
INT	Import Name Table，输入名称表
KeygenMe	一些供给别人尝试解密的小程序，要求做出它的 keygen（序号产生器）
Loader	内存补丁
Module	模块
MSIL	微软中间语言
Nag 窗口	警告提示窗口
Native-Compile	自然编译，编译器将高级语言转换为汇编代码
Obfuscation	混淆
OD	OllyDbg 的简称

OEP	Original Entry Point，原始入口点
Overflow	溢出
pack	壳（音 ké），一种专用加密软件
patch	为文件打补丁
Pcode-Compile	伪编译，编译器将高级语言转换为某种编码后，解释执行
PE	Portable Executable，Windows 系统可执行文件格式
PEB	Process Environment Block，进程环境块
PEDIY	修改扩充 PE 结构功能，对 PE 文件进行 DIY
Reflect	反射
ReverseMe	为练习逆向技术而写的（或特别构造的）小程序
Reversing	逆向，或称逆向工程（Reverse Engineering）
Ring0	操作系统内核模式
Ring3	操作系统用户模式
Rootkit	一种被设计的能得到最基本（高）权限的程序
RPL	请求特权级别
RVA	Relative Virtual Address，相对虚拟地址
SDK	Software Development Kit，软件开发工具包
SDT	ServiceDescriptorTable，服务描述表
Section	区块，区段，节
SEH	Structured Exception Handling，结构化异常处理
ShellCode	通称缓冲区溢出攻击中植入进程的代码
SMC	Self-Modifying Code 的缩写形式，在一段代码执行之前先对它进行修改
Stack	栈
stolen bytes	指外壳将程序部分代码变形，并搬到外壳段
String Encoding	字符串编码
StrongName	强名称
TEB	Thread Environment Block，线程环境块
TLS	Thread Local Storage，线程局部存储
Unmanaged Code	非托管代码
unpack	脱壳
Virtual Address	简称 VA，内存虚拟地址
VM	Virtual Machine，虚拟机
领空	指在某一时刻，CPU 的 CS:EIP 所指向的某段代码的所有者

参 考 文 献

[1] Charles Petzold. Programming Windows. Microsoft Press，1998
[2] John Robbins. Debugging Applications. Microsoft Press，1999
[3] Gary Nebbett.Windows NT 2000 Native API Reference，2001
[4] Everett N. McKay,Mike Woodring. Debugging Windows Programs. Addison-Wesley
[5] Bruce Schneier . Applied Cryptography. John Wiley & Sons，1996
[6] Jeffrey Richter. Programming Applications for Microsoft Windows .Microsoft Press，2000
[7] Matt Pietrek . Windows 95 System Programming SECRETS. IDG Books，1995
[8] Eldad Eilam.Secrets of Reverse Engineering. Wiley，2005
[9] Matt Pietrek.An In-Depth Look into the Win32 Portable Executable File Format
[10] Randy Kath.The Portable Executable File Format from Top to Bottom
[11] David Solomom.Inside Windows NT,2nd Edition
[12] FIPS 186-2.Digital Signatrue Standard. NIST 2000
[13] FIPS 197.Announcing the Advanced Encryption Standard. NIST 2001
[14] A public Key CryptoSystem and a Signature Scheme Based on Discrete Logarithms.TAHER Elgamal，1985
[15] Media Crypt .Internation Data Encryption Algorithm Technical Description
[16] RFC1321.The MD5 Message-Digest Algorithm. R.Rivest，1992
[17] David J. Wheeler, Roger M. Needham. "TEA, A Tiny Encryption Algorithm",
[18] "Guide to Elliptic Curve Cryptography", Darrel Hankerson,Alfred Menezes, Scott Vanstone, Springer 2003
[19] Cryptography and Network Security Principles and Practices, Third Edition William Stallings，2003
[20] Making, Breaking Codes An Introduction to Cryptology, Paul Garrett，2003
[21] Cryptography Theory and Practice (Second Edition), Douglas R. Stinson，2002
[22] Professional C#(3rd),Simon Robinson,Wiley Publishing，2004
[23] Expert .NET 2.0 IL Assembler,Serge Lidin,Apress，2006
[24] Essential .NET Vollume 1: The Common Language Runtime，Don Box，Addison Wesley，2003
[25] C# to IL,Vijay Mukhi，BPB Publications，2003
[26] Distributed Virtual Machines: Inside the Rotor CLI,Gray Nutt，Addison Wesley，2005
[27] ECMA-334，C# Language Specification；ECMA-335，Common Language Infrastructure (CLI)
[28] Applied Microsoft .NET Framework Programming，Jeffrey Richter，Microsoft Press，2002
[29] Common Language Infrastructure Annotated Standard，Jim Miller，Addison Wesley，2003
[30] 看雪学院. 软件加密技术内幕. 北京：电子工业出版社，2004
[31] Kris Kaspersky. 谭明金译. 黑客反汇编揭秘. 北京：电子工业出版社，2004
[32] 罗云彬. Windows 环境下 32 位汇编语言程序设计. 北京：电子工业出版社
[33] 周明德. 保护方式下的 80386 及其编程. 北京：清华大学出版社
[34] 扬季文. 80X86 汇编语言程序设计教程. 北京：清华大学出版社
[35] 尤晋元，史美林等. Windows 操作系统原理. 北京：机械工业出版社，2001
[36] 侯杰. Windows 系统编程奥秘
[37] 邹丹. www.zoudan.com. 关于 Windows 95 下的可执行文件的加密研究
[38] Lenus. 浅谈脱壳中的 Dump 技术. 看雪软件安全论坛
[39] Peansen. 记事本功能增加方案. 看雪软件安全论坛
[40] CCDebuger. OllyDBG 入门系列教程. 看雪软件安全论坛，2006
[41] CCDebuger.浅谈程序脱壳后的优化. 看雪软件安全论坛，2006