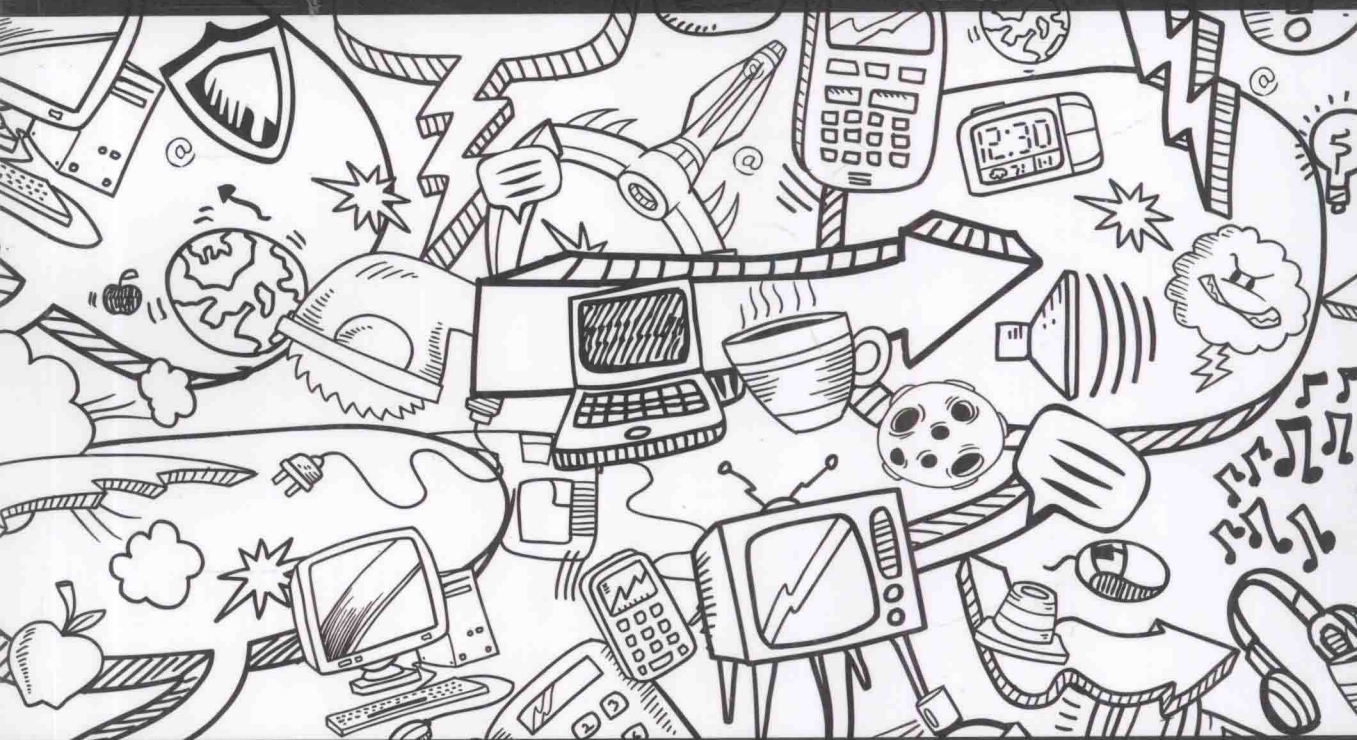


软件保护及分析技术

—— 原理与实践

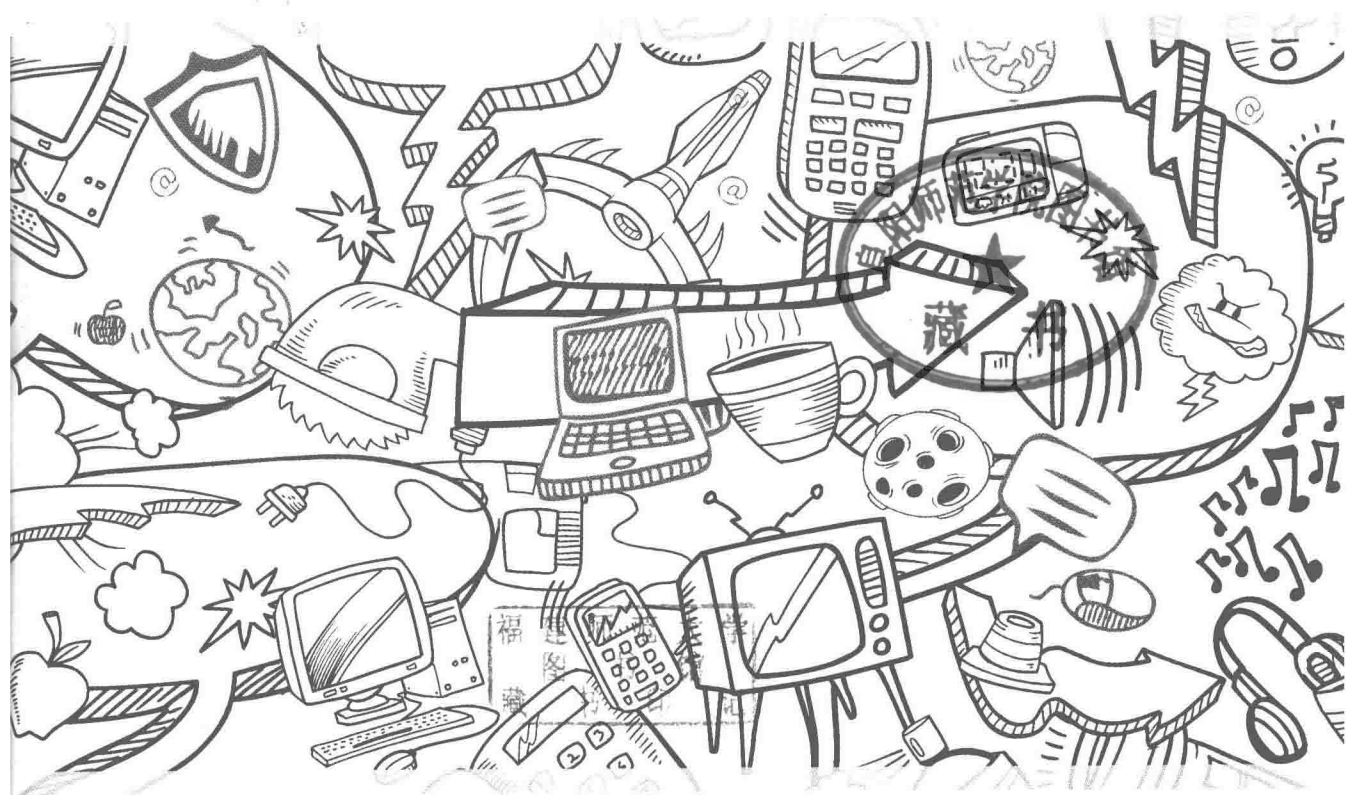


章立春 编著



软件保护及分析技术

—— 原理与实践



章立春 编著

电子工业出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书对软件安全领域的保护与分析技术进行了全面的讨论和剖析,通过保护与分析的对比讲解,结合代码和操作流程,对软件安全领域的各种技术进行了详尽的讨论,并将理论与实践相结合,利用具体的程序代码进行演示。同时,对现今较为成熟的保护系统进行了分析,全面介绍了软件安全领域的保护与分析技术。最后,结合多年从事软件软件保护与分析的经验,讲解了软件保护与分析中的各种经验和技巧。

本书适合信息安全领域相关人员、高校相关专业学生及爱好者阅读。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

软件保护及分析技术:原理与实践 / 章立春编著. —北京:电子工业出版社, 2016.8
(安全技术大系)
ISBN 978-7-121-29264-4

I. ①软… II. ①章… III. ①软件—安全技术 IV. ①TP311.56

中国版本图书馆 CIP 数据核字(2016)第 150715 号

责任编辑:潘 昕

印 刷:涿州市京南印刷厂

装 订:涿州市京南印刷厂

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:787×1092 1/16

印张:26 字数:612 千字

版 次:2016 年 8 月第 1 版

印 次:2016 年 8 月第 1 次印刷

定 价:79.00 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, 88258888。

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。

本书咨询联系方式:010-51260888-819 faq@phei.com.cn。

软件保护与分析技术领域一直以来都充满着神秘色彩。一提到这个领域，总会激发外行人的强烈好奇和异样的眼光。作为一个业内人士，我认为，尽管这个行业内充满了利益瓜葛，但是，若我们抛开利益，回归这个技术领域的本质，就会发现更加重要的其实是这个领域给我们带来的乐趣、对事物的理解，以及那些成功与挫败的体验，这些才是这个领域最为本质也是最吸引人的地方。

很难想象，如果我们涉及计算机行业，但又一直只是充当一个代码工具，在现在这个软件设计越来越趋向对象化、自动化的年代，我们真不知道除了代码之外还能有什么，更不用说长期做重复单调的工作需要多大的耐心和毅力。

黑格尔曾经说过，事物总是在对立中才能进步。软件加密与解密这个领域正好有两种永远对立的事物，这就使得这个领域不存在终点，永远都有更加深入的技术等待着我们去探索，让这个领域充满了乐趣。但是，随着计算机硬件的快速发展，这个领域的这种对立正在慢慢地发生变化并且失去平衡。因为计算机越来越快的运算速度，使得加密技术可以使用计算机的这种计算优势将安全性转移到计算机运算速度上，也就是说，现代的加密技术可以使用巨大的运算量来换取安全性的提高。一方面，这对破解来说是极不乐观的，因为在破解技术上，大多数的分析还是依靠人脑。另一方面，由于计算机领域具有累积性，慢慢地，加密与解密这个领域开始具备了排斥性，使这个领域的门槛越来越高，这让无数热爱计算机技术的人失去了体验这个领域内众多乐趣的机会。

写作背景

我相信很多准备进入软件行业的技术人员都对软件的保护和破解领域具有浓厚的兴趣，但是我也能想象，当他们兴致勃勃地从网上搜索各种加密与解密入门教程，并着手模仿和练习时，却发现现在的软件随便用 PEiD 查看都会显示“VMProtect”字样，然后他们转而查找有关 VMProtect 的资料——好不容易找到一点点关于 VMProtect 的资料后却发现，文档中的内容完全不知所云，或者文档中描述的内容和他们实际操作的代码完全不一样，这让他们感到极度挫败。更加严重的问题在于：当我们习惯使用这些成熟、强大的保护系统以后，就慢慢丧失了追寻事物本质的能力；我们看重成熟的保护系统给软件带来的安全性，就慢慢忽视了这些保护系统保护软件的技术细节，这使得我们产生一种错觉，甚至可能以为一旦拥有这样的保护系统，我们的软件就安全了，这是相当严重的问题。

作为一个成熟的技术人员，我们应当足够了解我们所开发软件的整个运行流程，这样才能在各种情况下做出相对准确和客观的判断。因此，这也包括我们对软件保护系统及软件破解技术的了解。

本书全面介绍了软件保护与分析领域的相关技术，因此涵盖很多高手看来较低级的技术，还会提及很多基础性的内容，但由于篇幅限制，我会尽量简化这些部分。在我看来，在加密与解密领域，对技术的理解必须用理论与实际结合并侧重于实际操作的方式效果才会更好。因为这个领域是一个需要培养自己动手能力和花大量时间去具体操作才能完全理解的领域，只有在实际的操作过程中才能体会到这个领域中的千变万化，所以，书中较多地采用教学的方式，并且尽量提供相应的代码和示例，最后会一步步地引导读者打造自己的工具以提升动手能力。

本书充分考虑到读者可能面临的实际情况，所以尽量不涉及和不使用内核技术来解决问题，而是将各种方法都放在 Ring3 下模拟和实现，以避免频繁的蓝屏和复杂的内核调试过程吓退读者。

本书特点

保护与分析全覆盖：本书涵盖软件保护与软件破解的大部分相关技术，不仅介绍了大量的软件保护技术，也介绍了大量的软件破解技术，通过保护与破解技术的对比，读者可以更加深刻地理解和体会各种技术的优缺点。

理论与实践结合：本书不仅通过理论来介绍各种技术，还通过实际代码和操作流程将这些理论转化为实际的程序或者工具。通过本书的“自己动手”部分，读者可以充分体会到将一种技术理论转变为实际操作过程或者程序的整个步骤，并从中体会到理论与实际应用的差别，最终对各种技术有充分、全面的理解。

高阶技术：本书不仅介绍一些常见而普通的技术，在许多方面，本书所介绍的技术即使对很多业内人士来说，也是相当有难度的。通过这些技术的示例和应用，希望能让读者明白：在计算机领域，缺乏的并不是工具，而是 idea。

读者对象

本书的内容并非是为初学者设计——尽管我希望能将我的想法表达得足够简单。在开始阅读本书之前，我假定你（读者）至少已经具备以下初步的计算机能力：

- 基本的 C/C++ 编程能力
- 基本的 x86 汇编能力
- 一定的软件运行原理知识

如果你已经具备以上条件，那么恭喜你，你将可以很好地进入本书的其他部分。由于本书涉及的技术属于两大对立的板块，所以，为了清晰地阐述本书结构，在章节安排上我采取了归纳的形式，各个章节在技术难度上没有绝对的先后关系。读者在阅读的时候，可以根据自己的实际情况调整阅读顺序。

章立春

2016 年 1 月

目 录

第 0 章 准备知识	1	0.3 Win32 进程的启动.....	5
0.1 Win32 程序	1	0.4 PE 程序的加载.....	7
0.2 PE 程序文件	3		
第 1 部分 软件保护			
第 1 章 软件保护技术	11	1.9 脚本引擎	42
1.1 反调试.....	11	1.10 网络加密	42
1.1.1 函数检测.....	12	1.11 硬件加密	42
1.1.2 数据检测.....	12	1.12 代码签名	43
1.1.3 符号检测.....	13		
1.1.4 窗口检测.....	13	第 2 章 软件保护系统	44
1.1.5 特征码检测.....	14	2.1 保护功能	44
1.1.6 行为检测.....	14	2.1.1 试用控制	44
1.1.7 断点检测.....	15	2.1.2 授权控制	45
1.1.8 功能破坏.....	16	2.1.3 功能扩展	45
1.1.9 行为占用.....	16	2.2 保护系统	45
1.2 反虚拟机.....	17	2.2.1 WinLicense 和 Themida ...	46
1.3 数据校验.....	18	2.2.2 VMProtect	47
1.4 导入表加密	19		
1.5 模块拷贝移位.....	27	第 3 章 软件保护强化	48
1.6 资源加密.....	29	3.1 设计优化	49
1.7 代码加密.....	30	3.1.1 技巧 1: 代码习惯优化....	49
1.7.1 代码变形.....	30	3.1.2 技巧 2: SDK 使用优化 ...	51
1.7.2 花指令	33	3.1.3 技巧 3: 验证保护系统....	52
1.7.3 代码乱序.....	33	3.2 加壳优化	52
1.7.4 多分支	35	3.2.1 技巧 1: 侧重选择代码	
1.7.5 call 链.....	36	加密.....	53
1.8 代码虚拟化.....	37	3.2.2 技巧 2: 精选被保护	
1.8.1 堆机	39	代码.....	54
1.8.2 栈机	40	3.2.3 技巧 3: 充分测试	55
1.8.3 状态机	42		

第2部分 软件破解

第4章 软件破解工具	58		
4.1 调试分析工具	58		
4.1.1 OllyDbg	58		
4.1.2 WinDBG	59		
4.1.3 IDA	60		
4.2 修改工具	60		
4.2.1 LordPE	60		
4.2.2 010Editor	61		
4.3 自动化工具	61		
第5章 软件破解技术	62		
5.1 静态分析	63		
5.1.1 基本信息分析	63		
5.1.2 代码静态分析	67		
5.2 软件调试	68		
5.2.1 一般调试原理	68		
5.2.2 伪调试技术	72		
5.2.3 本地调试	73		
5.2.4 远程调试	74		
5.2.5 虚拟机调试	76		
5.3 反反调试	77		
5.4 HOOK	78		
5.4.1 代码 HOOK	79		
5.4.2 函数 HOOK	86		
5.4.3 模块 HOOK	86		
5.4.4 导出表 HOOK	91		
5.5 代码注入	94		
5.6 补丁	101		
5.6.1 冷补丁	101		
5.6.2 热补丁	102		
5.6.3 SMC	102		
5.6.4 虚拟化补丁	103		
5.7 模块重定位	103		
5.8 沙箱技术	104		
5.9 虚拟化	106		
5.10 代码虚拟机	107		
5.10.1 OP 分支探测	108		
5.10.2 OP 调试	113		
5.11 自动化技术	114		
5.11.1 代码追踪	114		
5.11.2 预执行	118		
5.11.3 代码简化	120		
5.11.4 代码重建	124		
5.11.5 块执行	125		
5.11.6 多分支剔除	126		
5.11.7 小结	129		
5.12 动态分析	129		
5.12.1 着色	129		
5.12.2 黑盒测试	132		
5.13 功能模拟	132		
5.13.1 授权模拟	132		
5.13.2 网络模拟	134		
5.13.3 加密狗模拟	134		
5.14 脱壳	134		
5.14.1 导入表重建	135		
5.14.2 资源重建	149		
5.14.3 区段重建	152		
5.14.4 OEP 定位	154		
5.14.5 PE 头修复	159		
5.14.6 重定位修复	159		
5.14.7 PE 重建	163		
5.14.8 补区段	164		
5.15 进程快照技术	166		
5.15.1 进程快照技术概述	166		
5.15.2 快照脱壳	178		
5.16 代码回溯技术	180		
第6章 软件分析技巧	184		
6.1 技巧1: 精确代码范围	184		
6.2 技巧2: 多用对比参考	186		
6.3 技巧3: 逆向思考	187		
6.4 技巧4: 多利用自动化优势	187		
6.5 技巧5: 利用环境优势	187		
6.6 技巧6: 尽量避免算法分析	187		
6.7 技巧7: 够用原则	188		

第3部分 自己动手

第7章 打造函数监视器.....	190	9.4.4 对比内存数据	244
7.1 制定功能.....	190	9.4.5 重建重定位区段	245
7.2 确定技术方案.....	191	9.5 效果演示	248
7.3 开发筹备.....	192	9.5.1 DLL 模块重定位修复..	249
7.4 具体实现.....	193	9.5.2 主模块重定位修复	251
7.4.1 启动目标进程并注入 xVMRuntime 模块.....	196		
7.4.2 通信协议.....	199		
7.4.3 事件设定.....	201		
7.4.4 辅助调试功能.....	216		
7.4.5 技术问题.....	217		
7.5 效果演示.....	217		
第8章 打造资源重建工具	220	第10章 打造进程拍照机	253
8.1 制定功能.....	220	10.1 制定功能	253
8.2 确定技术方案.....	220	10.2 确定技术方案	253
8.3 开发筹备.....	222	10.3 开发筹备	255
8.4 具体实现.....	222	10.4 具体实现	255
8.4.1 数据结构及通信协议.....	222	10.4.1 先期模块注入	255
8.4.2 获取内存段资源数据.....	223	10.4.2 接管进程内存管理	261
8.4.3 监控资源函数获取 数据.....	225	10.4.3 建立函数调用中间层.....	268
8.4.4 强制搜索内存穷举获 取数据	227	10.4.4 实现场景载入功能	269
8.4.5 重建资源区段.....	230	10.4.5 转储并修正映像及 相关数据.....	270
8.4.6 技术问题.....	234	10.4.6 增加 TIB 转储.....	274
8.5 效果演示.....	235	10.5 效果演示	276
第9章 打造重定位修复工具	238	10.5.1 WinLicense 测试	276
9.1 制定功能.....	238	10.5.2 VMProtect 测试.....	279
9.2 确定技术方案.....	238	第11章 打造函数通用追踪器	281
9.3 开发筹备.....	239	11.1 制定功能	281
9.4 具体实现.....	239	11.2 确定技术方案.....	281
9.4.1 通信协议.....	239	11.3 开发筹备	282
9.4.2 注入模块.....	240	11.4 具体实现	283
9.4.3 抓取内存快照.....	240	11.4.1 建立插件框架	283
		11.4.2 分层式虚拟机	284
		11.4.3 调用代码查找识别	287
		11.5 追踪函数	294
		11.6 重建导入表.....	299
		11.7 修复调用代码.....	304
		11.7.1 内存式修复	305
		11.7.2 文件式修复	307
		11.8 效果演示	309

第 12 章 打造预执行调试器312

12.1 制定功能312

12.2 确定技术方案312

12.3 开发筹备313

12.4 具体实现313

12.4.1 预执行功能313

12.4.2 代码追踪记录功能317

12.4.3 块执行功能321

12.4.4 OP 记录调试功能327

12.5 效果演示331

第 13 章 打造伪调试器335

13.1 制定功能335

13.2 确定技术方案335

13.3 开发筹备336

13.4 具体实现336

13.4.1 数据结构与通信协议337

13.4.2 第 1 步：界面相关工作338

13.4.3 第 2 步：在调试端启用和禁用伪调试技术339

13.4.4 第 3 步：创建调试目标341

13.4.5 第 4 步：等待调试事件主循环344

13.4.6 第 5 步：被调试端的初始化346

13.4.7 第 6 步：中转异常349

13.4.8 第 7 步：辅助调试函数实现350

13.4.9 小结354

13.5 效果演示354

第 4 部分 实例分析

第 14 章 VMProtect 虚拟机分析358

第 15 章 WinLicense 虚拟机分析 ... 375

第 5 部分 脱壳实例

第 16 章 VMProtect 脱壳382

第 17 章 WinLicense 脱壳 394

写在最后 405

第 0 章 准备知识

如果你跳过了前言直接阅读本章，强烈建议你还是返回仔细阅读前言。在这里我们不会详尽地讨论计算机的运行原理，只是关心与本书内容相关的知识，以及 Windows 平台下的程序运行基本流程和必要条件。

为了照顾一些基础比较差而又对软件加密与解密特别好奇的读者，我们特别安排了本章。如果已经具备相关知识，请跳过本章。

0.1 Win32 程序

首先我们引出一个问题：什么是程序？

顾名思义，程序就是一系列的流程和动作。在计算机中，我们所说的程序，一般是指能够操控计算机帮助我们完成一件或多件事情的一种事物，这种事物以数字的方式存在。

回顾一下计算机原理。计算机最重要的组件就是 CPU（中央处理器），CPU 最重要的功能就是进行逻辑运算，比如加、减、乘、除这些常用的运算。为了让 CPU 能够真正为我们所用，我们必定要有控制 CPU 进行运算的能力。在计算机中，因为 CPU 只能进行纯数字的运算，所以 CPU 是数字指令模式的，也就是说，我们使用 CPU 的过程是：向 CPU 输入一段数字和处理这些数字的方式，CPU 经过运算后返回一堆数字结果。另外，计算机处理我们所输入的数字的方式，我们也是以数字的形式告诉 CPU 的，如今大部分 CPU 只接受二进制（只需要用 0 和 1 表示）数字，任何其他形式的数字最终都会转换为二进制输入到 CPU 中。我们将输入 CPU 的这些数据（包括需要被处理的数字和处理方式）称为指令或代码，将 CPU 的运算结果称为指令结果。为了实现不同的目的，需要让 CPU 处理不同数目和类别的指令，这类指令称为指令集。你可能听说过 x86、x87、mmx、sse、MIPS 等，它们都是不同的指令集。一块 CPU 可能具备处理多种指令集的能力，比如我们目前常用的 Intel 通用处理器一般都具备处理 x86、x87、mmx 指令集的能力。

如图 0.1 所示是一段 x86 指令示例。

地址	十六进制数据	汇编代码
0046E7C4	68 10E84600	push 0046E81D
0046E7C9	64:FF35 00000000	push dword ptr fs:[0]
0046E7D0	8B4424 10	mov eax,dword ptr [esp+10]
0046E7D4	896C24 10	mov dword ptr [esp+10],ebp
0046E7D8	8D6C24 10	lea ebp,[esp+10]
0046E7DC	2BE0	sub esp,eax
0046E7DE	53	push ebx
0046E7DF	56	push esi

图 0.1

图 0.1 中共包含 8 条指令。第 2 列的十六进制数据就是输入 CPU 的指令数据，可以发现，指令始终是数字形式的（在十六进制中，A、B、C、D、E、F 代表数字而非字母）。第 1 列的地址表示这些指令在内存中的存放位置。第 3 列的汇编代码就是这些数字指令所对应的指令翻译，我们称之为汇编代码。CPU 的计算与这里看到的第 3 列内容无关，这些代码形式只是为了方便我们阅读和理解这些数字指令而翻译出来的。

从图 0.1 中我们还可以看出，每条指令可以具有不同的长度。在 x86 指令集的 CPU 中，指令是顺序执行的，也就是说，除非遇到流程控制指令，否则 CPU 会自上而下地顺序执行指令。

所以，程序就是一系列 CPU 指令的集合，我们甚至可以说，一条指令就是一个程序。但是，随着计算机技术的发展，对程序的这种范围上的定义越来越不适合表述现在的情况。现在，一台普通计算机每秒所运行的指令都是上百万级别的，因此我们对程序的这种定义就显得很笼统。

现在的各种多任务操作系统，如 Windows、Linux、Mac OS，都有一种按照功能将一族代码归类的设计，称之为进程。在这种设计理念下，一个进程将为了实现一个目的而设计的大量指令归类到同一个集合中进行管理，所以我们常把进程称为程序，如图 0.2 所示。

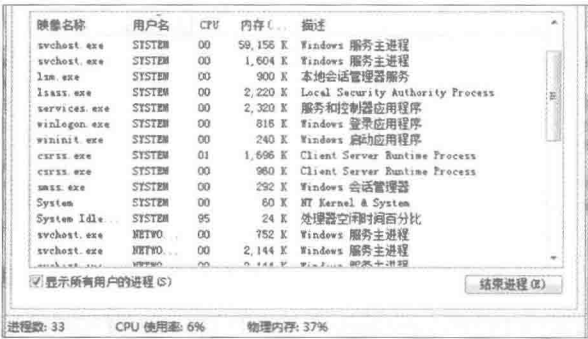


图 0.2

在图 0.2 中，计算机同时运行多个程序，每一个程序都负责特定的工作，也因此拥有特定的指令集合。根据冯·诺依曼体系（一种计算机设计体系，核心思想是将代码复制到内存后再交由 CPU 执行），每个程序运行时，这些指令都会被复制到内存中提供给 CPU 执行，在程序没有执行时，这些代码就存储在文件中。第 1 列的映像名称就表示这个程序的入口代码数据所在的文件名称。根据系统设计的不同，各系统所要求的文件存储程序代码的方式也不同。在 Linux 下程序文件格式为 ELF，在 Windows 下程序文件格式为 PE。

0.2 PE 程序文件

PE 文件格式是 Windows 下程序代码在文件中的存储规范,任何 Win32 子系统的程序都要以这种规范存储才能被系统正确识别并加载运行。PE 程序的详细格式,请读者自行阅读相关文档。当然,这里我们也只提供一种直观地观察 PE 文件程序格式的方法,希望帮助读者更加快速地了解 PE 文件格式。

要理解 PE 文件格式,除了埋头阅读介绍文档,更好的方法是通过 PE 查看或编辑工具来观察实际的 PE 文件,目前使用比较多的 PE 文件工具有 PEiD、LordPE、stud_pe、Explorer Suite、PE-Explorer 等。在这里我们选用 stud_pe,因为它有一个直观的十六进制和 PE 文件头同步查看器。

启动 stud_pe,软件运行界面如图 0.3 所示。

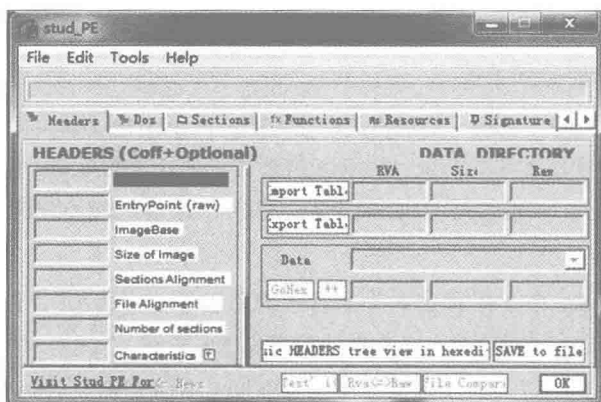


图 0.3

依次单击“File”→“Open PE File”菜单项,打开一个 PE 程序文件(例如 c:\windows\system32\notepad.exe),然后单击“Basic HEADERS tree view in hexeditor”按钮,打开如图 0.4 所示的界面。通过该界面显示的内容和相关文档,我们可以详细地观察 PE 文件头的各个成员在文件中的对应数据和位置。相信通过这样的直观观察,读者一定能够很快对 PE 文件格式有所了解。

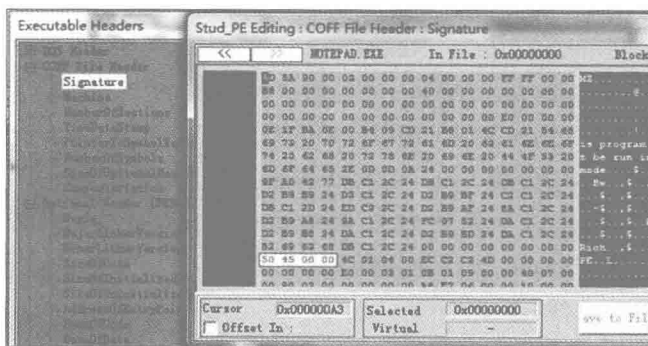


图 0.4

在充分了解 PE 文件格式后,我们将简单讨论 PE 文件格式中各成员的重要程度,这对于我们理解软件破解是很有帮助的。

一个 PE 程序要被 Windows 正确识别并启动运行,需要 PE 文件的各个成员数据设定

正确，但这并非对所有成员都是必要的。

实际上，在 PE 文件头中，DOS Header（详见 Windows SDK winnt.h 头文件中的 IMAGE_DOS_HEADER）部分对于 32 位与 64 位 PE 程序来说，只有其中的 Magic Number 和 e_lfanew（File address of new exe header）成员是重要的，其他成员都只有 16 位程序才会使用。在这个头中，Magic Number 固定为 4D 和 5A(MZ)，所以，判断一个 PE 文件最简单的方法就是观察文件是否由字符“MZ”开始，如果不是，则该文件一定不是可以直接运行的 PE 文件。

紧接着 DOS 头的是 FileHeader（详见 winnt.h 中的 IMAGE_FILE_HEADER）。在这个头中，Machine（如图 0.4 所示的 Signature）、NumberOfSections、SizeOfOptionalHeader、Characteristics 成员都是必需的，Machine 成员固定为“50 45 00 00”字节序列。因此，在判断 DOS 头后，进一步验证 PE 文件是否正确的方法就是验证这里的字节序列。NumberOfSections 指明了该 PE 文件头所拥有的区段数目，对于一个能够正确运行的 PE 程序文件来说，区段映射必须正确。

SizeOfOptionalHeader 指明了紧接着 FileHeader 的 Optional Header 的大小，操作系统会验证这里的值来判断 PE 文件的特征，所以也是必需且要正确设定的。Characteristics 指定了 PE 文件所具备的信息，如是否是 DLL 模块等，这对于操作系统是否能够正确识别 PE 程序文件是必要的。

在 Optional Header 中（尽管这个头的名字的意思是“可选头”，但是这个文件头在 32 位及以上的 PE 文件中是必需的），这个文件头根据 PE 文件（32 位/64 位）的不同有不同的结构，分别在 winnt.h 中定义为 IMAGE_OPTIONAL_HEADER32 和 IMAGE_OPTIONAL_HEADER64。两者的大体结构是一样的，只是在各成员中的大小不一样。在 Optional Header 中，很多成员（如 Magic、AddressOfEntryPoint、ImageBase、SectionAlignment、FileAlignment、DllCharacteristics、NumberOfRvaAndSizes）都是必需的，其中 MajorSubsystemVersion、MinorSubsystemVersion 成员指定了该 PE 程序文件运行所需要的最小子系统版本。从 Visual Studio 2012 开始，编译器默认将这个版本设定为 6.0，因此 Windows 2003 及以下系统默认无法运行 Visual Studio 2012 编译器编译的 PE 程序文件。

在 Optional Header 中，还有一个 DataDirectory 数组成员，这个成员指定了 PE 程序运行时的另外一些信息，如图 0.5 所示。这个目录表指定了该 PE 程序运行时的必要信息，如导入表等。



图 0.5

0.3 Win32 进程的启动

进入 Windows 系统操作界面以后，绝大部分进程都是由其他进程启动的。所以，为了理解 Windows 进程的启动过程，我们最好从“在一个程序中启动另外一个进程”开始。下面我们就在实际操作中一步步了解这个过程。

在一个程序中启动另外一个进程比较常用的 Win32 函数是 CreateProcess（这是函数别名，对应于 CreateProcessA 或 CreateProcessW，本书其他部分也只引用函数别名）。我们可以构建如图 0.6 所示的程序代码（读者可以在本书相应的代码目录下找到示例代码）。

```
5 int main(void)
6 {
7     STARTUPINFO pStartInfo = {0};
8     PROCESS_INFORMATION pProcInfo = {0};
9     pStartInfo.cb = sizeof(STARTUPINFO);
10
11     //调用中断函数，直接定位到启动函数位置
12     DebugBreak();
13     if (!CreateProcess(_T("c:\\windows\\system32\\notepad.exe"),
14         _T(""),
15         NULL, NULL, FALSE, CREATE_SUSPENDED, NULL,
16         _T("c:\\windows\\system32\\"), &pStartInfo, &pProcInfo))
17     {
18         return -1;
19     }
20     return 0;
21 }
```

图 0.6

编译完成后，用调试器 OllyDbg 启动程序并直接运行，等待程序中断，然后步过，如图 0.7 所示。

00E51063	60 7C39E500	push offset 00E5997C
00E51068	60 8039E500	push offset 00E59980
00E5106D	FF15 9C61E500	call dword ptr [<KERNEL32.CreateProcessW>]

图 0.7

一路跟踪，发现 CreateProcessW 函数最终调用 NtCreateuserProcess 函数，如图 0.8 所示。

76AD4342	50	push eax
76AD4343	8D85 D0FCFF	lea eax,[ebp-320]
76AD4349	50	push eax
76AD434B	FF15 6806AC7	call dword ptr [<ntdll.NtCreateUserProcess>]

图 0.8

当我们再次步过该函数，可以发现新的进程已经被创建，如图 0.9 所示。这说明，真正的进程初始化和创建工作都是由系统内核完成的。由于本书不涉及 Windows 内核方面的内容，所以这里只简单说明原理。

ollydbg.exe	0.17	29,396 K	35,244 K	2108 Free	32-bit Analy
ProcessStartup.exe		468 X	2,816 K	4064	
	Sus...	424 K	128 K		

图 0.9

Windows 内核在调用 NtCreateUserProcess 函数后，根据传递过来的参数，先检查参数中指定的程序文件状态和文件格式（本例中为 c:\windows\system32\notepad.exe），如果是合

格的 PE 程序文件且允许载入，就开启一个新进程，并将 PE 程序数据按照文件格式要求映射到内存空间中。然后，进行进程的首要初始化，包括进程的 PEB 等。

接下来，内核会通过新建的进程空间中 ntdll 模块的 LdrInitializeThunk 函数，将代码执行转移到新进程中进行进一步初始化。使用进程工具观察，我们可以发现新进程被创建后部分模块已经自动载入了，如图 0.10 所示。

Path	Description	Company Name
C:\Windows\System32\apisetschema.dll	ApiSet Sche...	Microsoft Corporation
C:\Windows\System32\ntdll.dll	NT 层 DLL	Microsoft Corporation
C:\Windows\SysWOW64\notepad.exe	记事本	Microsoft Corporation
C:\Windows\SysWOW64\ntdll.dll	NT 层 DLL	Microsoft Corporation

图 0.10

实际上，只有 ntdll.dll 模块是完全在内核中载入的，其他模块的载入（包括 kernel32.dll 的载入）都是在新进程本身的空间中初始化的。我们可以通过下面的方法了解这个过程。

LdrInitializeThunk 函数是新进程创建后进程空间中运行的第一条指令入口，我们可以使调试器在这个入口点中断，从而分析之后的整个初始化过程。普通调试器并不具备中断到 LdrInitializeThunk 函数的功能，使用 OllyDbg 的插件 StrongOD 并打开“Break On Ldr”功能就可以中断到 LdrInitializeThunk 函数入口。

如果读者希望使用其他调试器（例如 IDA）中断到 LdrInitializeThunk 函数，这里提供另外一种方法。构建如图 0.11 所示的代码（该段代码可以在本书的代码目录下找到）。

```

5  int main(void)
6  {
7      STARTUPINFO pStartInfo = {0};
8      PROCESS_INFORMATION pProcInfo = {0};
9      pStartInfo.cb = sizeof(STARTUPINFO);
10
11     if (!CreateProcess(_T("c:\\windows\\system32\\notepad.exe"),
12                      _T(""),
13                      NULL, NULL, FALSE, CREATE_SUSPENDED | DEBUG_ONLY_THIS_PROCESS, NULL,
14                      _T("c:\\windows\\system32\\"), &pStartInfo, &pProcInfo))
15     {
16         return -1;
17     }
18
19     const void* lpLdrInitializeThunk = (const void*)GetProcAddress(
20         GetModuleHandle(_T("NTDLL")),
21         "LdrInitializeThunk");
22     //双int 3因为LdrInitializeThunk函数为mov edi,edi
23     //和某些调试器的步过
24     BYTE loopBytes[] = {0xCC, 0xCC};
25     SIZE_T szWritten = 0;
26     WriteProcessMemory(pProcInfo.hProcess,
27                        (LPVOID)lpLdrInitializeThunk,
28                        loopBytes, sizeof(loopBytes), &szWritten);
29     DebugActiveProcessStop(pProcInfo.dwProcessId);
30     return 0;
31 }

```

图 0.11

编译并运行该程序，就能够启动一个处于暂停状态的进程，此时用任何调试器附加到该进程上，就能中断于 LdrInitializeThunk 函数处（别忘了清除此处的“int3”），如图 0.12 所示。

316 →	ntdll.dll:774A9E4A ;	
	ntdll.dll:774A9E4B nop	
*	ntdll.dll:774A9E48 push	ebp
*	ntdll.dll:774A9E4C mov	ebp, esp
*	ntdll.dll:774A9E4E push	dword ptr [ebp+0Ch]
*	ntdll.dll:774A9E51 push	dword ptr [ebp+8]
*	ntdll.dll:774A9E54 call	near ptr unk_774A9F56
*	ntdll.dll:774A9E59 push	1
*	ntdll.dll:774A9E5B push	dword ptr [ebp+8]
*	ntdll.dll:774A9E5E call	near ptr ntdll_NtContinue
*	ntdll.dll:774A9E63 push	eax
*	ntdll.dll:774A9E64 call	near ptr ntdll_RtlRaiseStatus

图 0.12

根据 774A9E5E（平台不同，该地址会不同）处的 NtContinue 可以发现，LdrInitializeThunk 是由一个断点异常回调的，真正的线程入口在 ntdll.RtlUserThreadStart 处。在函数 774A9F56 中，系统将对进程进行初始化。我们可以在 LdrInitializeThunk 函数处查看此时进程的模块列表：除了 ntdll 外，连 kernel32 都没有载入，而以后可以知道，这对我们来说实在是一个好消息。此时，通过查看 PEB 中的 LoaderData 数据可以发现，进程的模块链表也没有初始化，所以，在 unk_774A9F56 中会首先初始化进程正常运行所必需的环境信息（如模块链表），然后通过 LdrLoadDll 载入 kernel32.dll，最后初始化主模块的 PE 文件头中所要求的初始化信息，正式进入 PE 程序文件的加载过程。

0.4 PE 程序的加载

通过 0.3 节我们可以了解，代码执行进入 LdrInitializeThunk 后，系统将开始对主模块进行初始化装载。在这个过程中，由于进程的主模块内存地址在进程创建时就已经由内核映射到内存空间了，所以，要想了解整个 PE 程序的加载，我们可以通过更加方便的手段——加载 DLL 来分析。系统加载 DLL 和加载 EXE 主模块之间没有太大的区别，关键在于加载 DLL 时，整个程序从文件读取数据开始就是在进程本身的空间中完成的，所以能很好地被我们调试和观察。下面我们就观察一下 DLL 的加载流程。

构建如图 0.13 所示的代码，编译在调试器中启动后中断于 LoadLibrary 的调用。

```

5 int main(void)
6 {
7     //调用中断函数，直接定位到启动函数位置
8     DebugBreak();
9     LoadLibrary(_T("c:\\windows\\system32\\shell32.dll"));
10    return 0;
11 }

```

图 0.13

此时，我们忽略其他无伤大雅的代码，直接跟踪 ntdll 模块中 LdrLoadDll 的位置，如图 0.14 所示。

75261D22	51	push ecx
75261D23	50	push eax
	FF15 F012257	call dword ptr [&ntdll.LdrLoadDll]
75261D2A	8BF8	mov edi, eax
75261D2C	FF75 10	push dword ptr [ebp+10]
75261D2F	FF75 FC	push dword ptr [ebp-4]

图 0.14

根据函数名称可以确定，该函数的作用就是加载 DLL 模块。进入该函数，发现该函数首先对模块地址做进一步处理和判断，并查找当前模块列表是否已经载入该模块。如果模块已经载入，那么 LdrLoadDll 就扮演 GetModuleHandle 的角色，只是将模块的引用计数调整好就直接返回了。在 NtOpenSection 处下断点，如图 0.15 所示。

		88 34000000	mov eax,34
7748F00D		33C9	xor ecx,ecx
7748F0BF		8D5424 04	lea edx,[esp+4]
Address	Hex dump	ASCII	
003AF33C	6D 00 73 00 76 00 63 00	n.s.v.c.	003AF180
003AF344	72 00 74 00 2E 00 64 00	r.k...d.	003AF184
003AF34C	6C 00 6C 00 00 00 57 00	l.l...W.	003AF188
003AF354	CE 3C 4A 77 78 00 00 00	<Jm...	003AF18C
			003AF1C0
			003AF1C4
			003AF1C8
			003AF1CC
			003AF1D0
			003AF1D4
			003AF1D8
			003AF1DC
			003AF1E0
			003AF1E4
			003AF1E8
			003AF1EC
			003AF1F0
			003AF1F4
			003AF1F8
			003AF1FC
			003AF200
			003AF204
			003AF208
			003AF20C
			003AF210
			003AF214
			003AF218
			003AF21C
			003AF220
			003AF224
			003AF228
			003AF22C
			003AF230
			003AF234
			003AF238
			003AF23C
			003AF240
			003AF244
			003AF248
			003AF24C
			003AF250
			003AF254
			003AF258
			003AF25C
			003AF260
			003AF264
			003AF268
			003AF26C
			003AF270
			003AF274
			003AF278
			003AF27C
			003AF280
			003AF284
			003AF288
			003AF28C
			003AF290
			003AF294
			003AF298
			003AF29C
			003AF2A0
			003AF2A4
			003AF2A8
			003AF2AC
			003AF2B0
			003AF2B4
			003AF2B8
			003AF2BC
			003AF2C0
			003AF2C4
			003AF2C8
			003AF2CC
			003AF2D0
			003AF2D4
			003AF2D8
			003AF2DC
			003AF2E0
			003AF2E4
			003AF2E8
			003AF2EC
			003AF2F0
			003AF2F4
			003AF2F8
			003AF2FC
			003AF300
			003AF304
			003AF308
			003AF30C
			003AF310
			003AF314
			003AF318
			003AF31C
			003AF320
			003AF324
			003AF328
			003AF32C
			003AF330
			003AF334
			003AF338
			003AF33C
			003AF340
			003AF344
			003AF348
			003AF34C
			003AF350
			003AF354
			003AF358
			003AF35C
			003AF360
			003AF364
			003AF368
			003AF36C
			003AF370
			003AF374
			003AF378
			003AF37C
			003AF380
			003AF384
			003AF388
			003AF38C
			003AF390
			003AF394
			003AF398
			003AF39C
			003AF3A0
			003AF3A4
			003AF3A8
			003AF3AC
			003AF3B0
			003AF3B4
			003AF3B8
			003AF3BC
			003AF3C0
			003AF3C4
			003AF3C8
			003AF3CC
			003AF3D0
			003AF3D4
			003AF3D8
			003AF3DC
			003AF3E0
			003AF3E4
			003AF3E8
			003AF3EC
			003AF3F0
			003AF3F4
			003AF3F8
			003AF3FC
			003AF400
			003AF404
			003AF408
			003AF40C
			003AF410
			003AF414
			003AF418
			003AF41C
			003AF420
			003AF424
			003AF428
			003AF42C
			003AF430
			003AF434
			003AF438
			003AF43C
			003AF440
			003AF444
			003AF448
			003AF44C
			003AF450
			003AF454
			003AF458
			003AF45C
			003AF460
			003AF464
			003AF468
			003AF46C
			003AF470
			003AF474
			003AF478
			003AF47C
			003AF480
			003AF484
			003AF488
			003AF48C
			003AF490
			003AF494
			003AF498
			003AF49C
			003AF4A0
			003AF4A4
			003AF4A8
			003AF4AC
			003AF4B0
			003AF4B4
			003AF4B8
			003AF4BC
			003AF4C0
			003AF4C4
			003AF4C8
			003AF4CC
			003AF4D0
			003AF4D4
			003AF4D8
			003AF4DC
			003AF4E0
			003AF4E4
			003AF4E8
			003AF4EC
			003AF4F0
			003AF4F4
			003AF4F8
			003AF4FC
			003AF500
			003AF504
			003AF508
			003AF50C
			003AF510
			003AF514
			003AF518
			003AF51C
			003AF520
			003AF524
			003AF528
			003AF52C
			003AF530
			003AF534
			003AF538
			003AF53C
			003AF540
			003AF544
			003AF548
			003AF54C
			003AF550
			003AF554
			003AF558
			003AF55C
			003AF560
			003AF564
			003AF568
			003AF56C
			003AF570
			003AF574
			003AF578
			003AF57C
			003AF580
			003AF584
			003AF588
			003AF58C
			003AF590
			003AF594
			003AF598
			003AF59C
			003AF5A0
			003AF5A4
			003AF5A8
			003AF5AC
			003AF5B0
			003AF5B4
			003AF5B8
			003AF5BC
			003AF5C0
			003AF5C4
			003AF5C8
			003AF5CC
			003AF5D0
			003AF5D4
			003AF5D8
			003AF5DC
			003AF5E0
			003AF5E4
			003AF5E8
			003AF5EC
			003AF5F0
			003AF5F4
			003AF5F8
			003AF5FC
			003AF600
			003AF604
			003AF608
			003AF60C
			003AF610
			003AF614
			003AF618
			003AF61C
			003AF620
			003AF624
			003AF628
			003AF62C
			003AF630
			003AF634
			003AF638
			003AF63C
			003AF640
			003AF644
			003AF648
			003AF64C
			003AF650
			003AF654
			003AF658
			003AF65C
			003AF660
			003AF664
			003AF668
			003AF66C
			003AF670
			003AF674
			003AF678
			003AF67C
			003AF680
			003AF684
			003AF688
			003AF68C
			003AF690
			003AF694
			003AF698
			003AF69C
			003AF6A0
			003AF6A4
			003AF6A8
			003AF6AC
			003AF6B0
			003AF6B4
			003AF6B8
			003AF6BC
			003AF6C0
			003AF6C4
			003AF6C8
			003AF6CC
			003AF6D0
			003AF6D4
			003AF6D8
			003AF6DC
			003AF6E0
			003AF6E4
			003AF6E8
			003AF6EC
			003AF6F0
			003AF6F4
			003AF6F8
			003AF6FC
			003AF700
			003AF704
			003AF708
			003AF70C
			003AF710
			003AF714
			003AF718
			003AF71C
			003AF720
			003AF724
			003AF728
			003AF72C
			003AF730
			003AF734
			003AF738
			003AF73C
			003AF740
			003AF744
			003AF748
			003AF74C
			003AF750
			003AF754
			003AF758
			003AF75C
			003AF760
			003AF764
			003AF768
			003AF76C
			003AF770
			003AF774
			003AF778
			003AF77C
			003AF780
			003AF784
			003AF788
			003AF78C
			003AF790
			003AF794
			003AF798
			003AF79C
			003AF7A0
			003AF7A4
			003AF7A8
			003AF7AC
			003AF7B0
			003AF7B4
			003AF7B8
			003AF7BC
			003AF7C0
			003AF7C4
			003AF7C8
			003AF7CC
			003AF7D0
			003AF7D4
			003AF7D8
			003AF7DC
			003AF7E