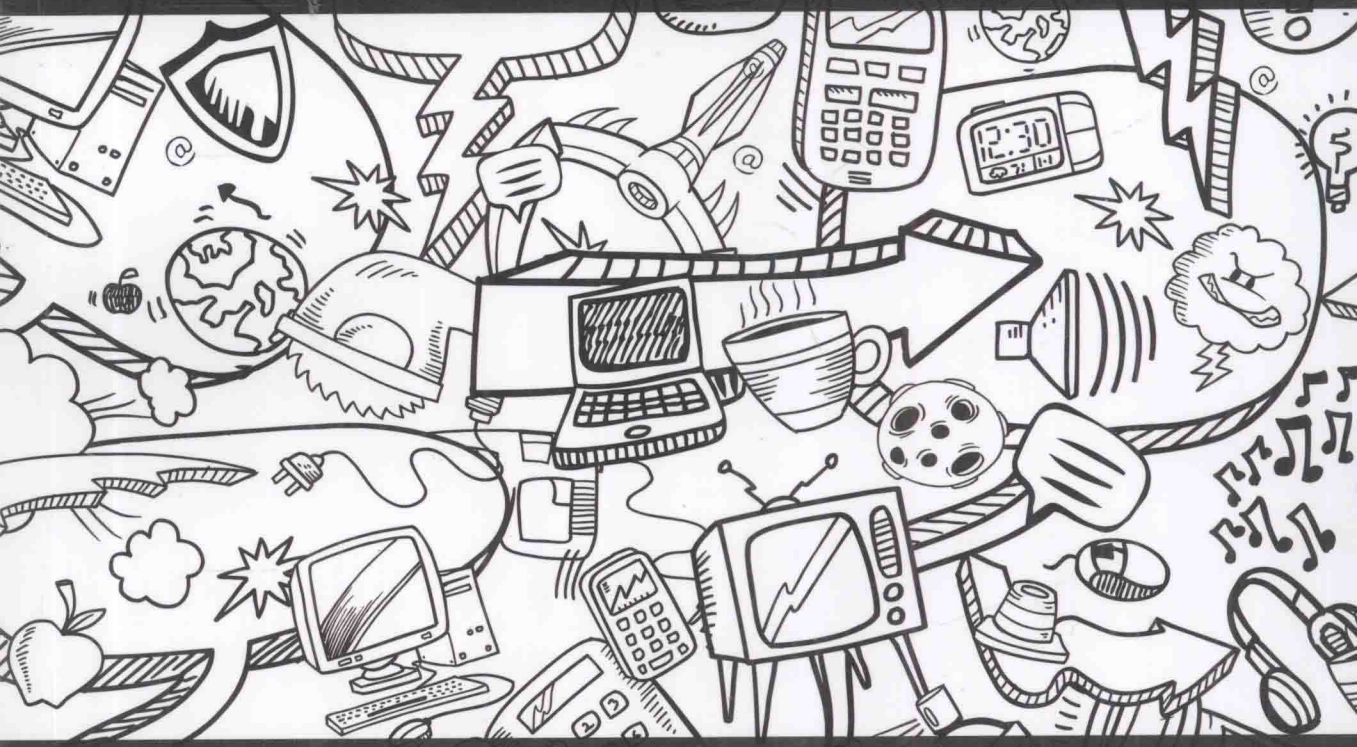
Broadview
www.broadview.com.cn

安全技术大系

技术覆盖范围全面，理论与实践相结合，技术层次循序渐进

软件保护及分析技术
—— 原理与实践



利用具体的程序代码进行演示讲解
对现今较为成熟的保护系统进行分析
全面介绍软件安全领域的各项保护和分析技术

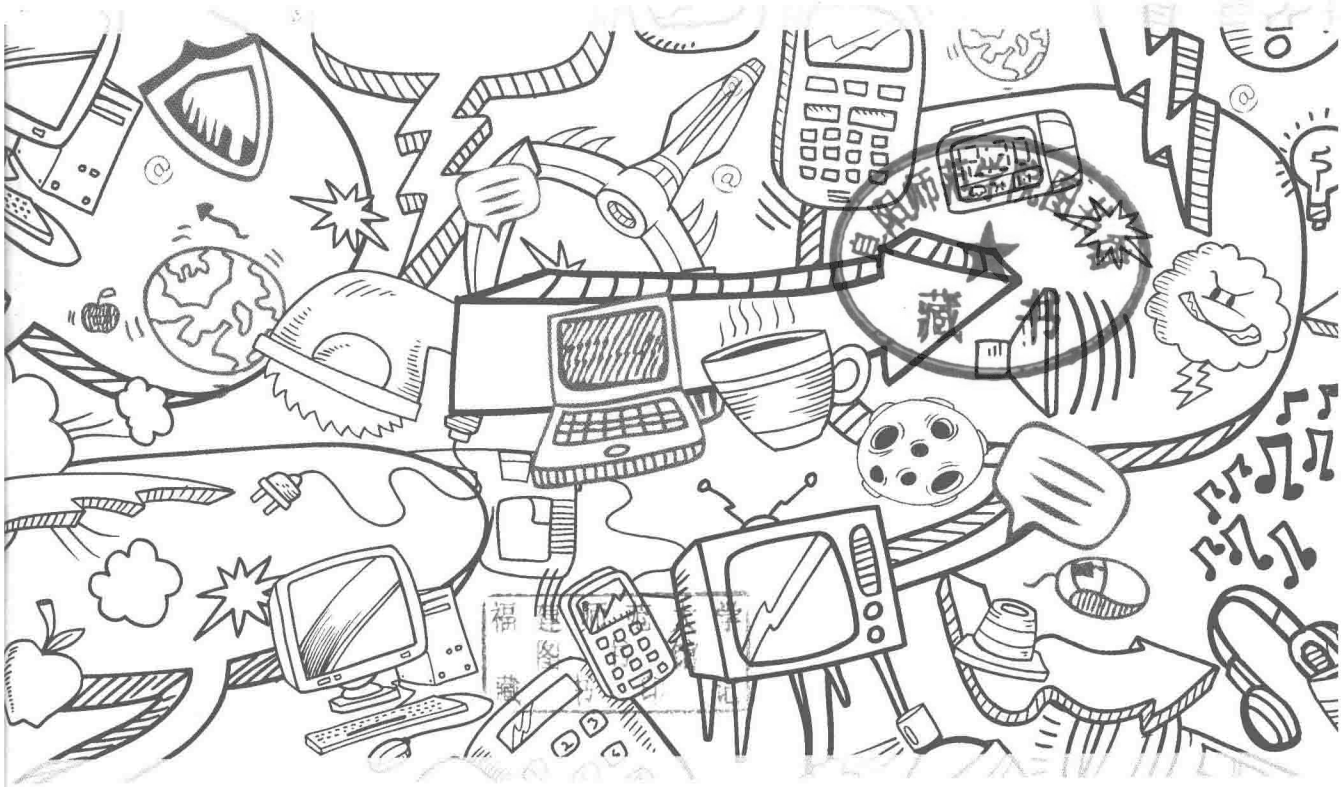
章立春　编著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

安全技术大系

软件保护及分析技术

—— 原理与实践



章立春　编著

電子工業出版社
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。

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。

前　言

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些什么，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 1 部分　软件保护

第 2 部分　软件破解

第 3 部分　自己动手

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 1DE84600	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。

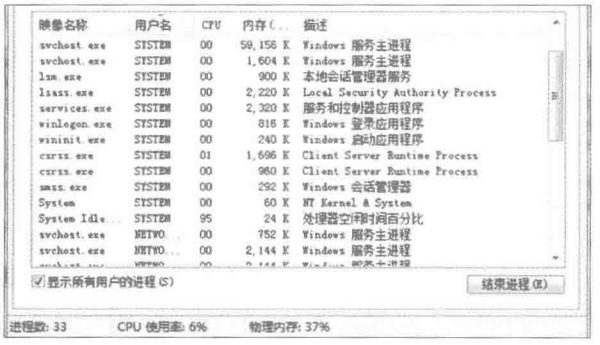


图 0.2

在图 0.2 中，计算机会同时运行多个程序，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。

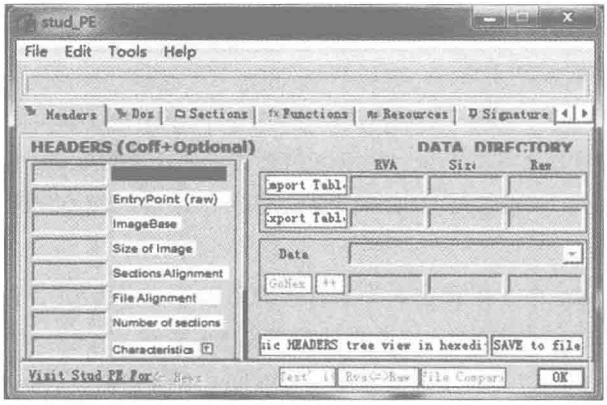


图 0.3

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。

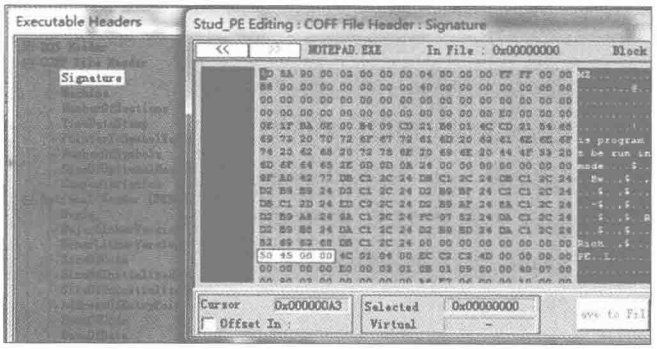


图 0.4

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。

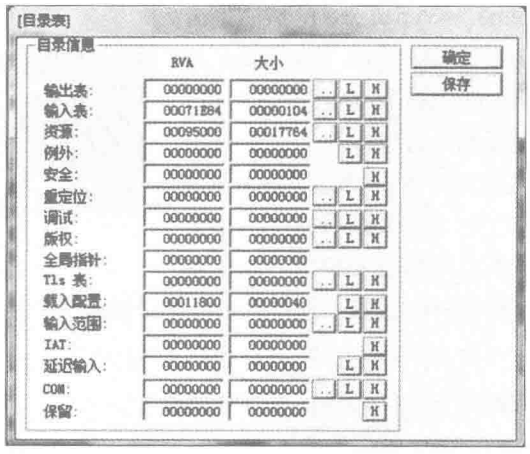


图 0.5

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）。

```
int main(void)
{
    STARTUPINFO pStartInfo = {0};
    PROCESS_INFORMATION pProcInfo = {0};
    pStartInfo.cb = sizeof(STARTUPINFO);

    //调用中断函数，直接定位到启动函数位置
    DebugBreak();
    if (!CreateProcess(_T("c:\\windows\\system32\\notepad.exe")
        ,_T("")
        ,NULL,NULL,FALSE,CREATE_SUSPENDED,NULL,
        _T("c:\\windows\\system32\\"),&pStartInfo,&pProcInfo))
    {
        return -1;
    }
    return 0;
}
```

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。

```
00E51003    68 7C39E500     push offset 00E5397C
00E51008    68 8039E500     push offset 00E53980
00E5100D    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 D0FCFFF    lea eax,[ebp-330]
76AD4349    50              push eax
76AD434A    FF15 6806AC77   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 Analysi
ProcessStartup.exe		468 K	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）。

```
int main(void)
{
    STARTUPINFO pStartInfo = {0};
    PROCESS_INFORMATION pProcInfo = {0};
    pStartInfo.cb = sizeof(STARTUPINFO);

    if (!CreateProcess(_T("c:\\windows\\system32\\notepad.exe")
        ,_T("")
        ,NULL,NULL,FALSE,CREATE_SUSPENDED |DEBUG_ONLY_THIS_PROCESS,NULL,
        _T("c:\\windows\\system32\\"),&pStartInfo,&pProcInfo))
    {
        return -1;
    }

    const void* lpLdrInitializeThunk = (const void*)GetProcAddress(
                GetModuleHandle(_T("NTDLL")),
                "LdrInitializeThunk");
    //双int 3因为LdrInitializeThunk函数为mov edi,edi
    //和某些调试器的步过
    BYTE loopBytes[] = {0xCC,0xCC};
    SIZE_T szWriten = 0;
    WriteProcessMemory(pProcInfo.hProcess,
                       (LPVOID)lpLdrInitializeThunk,
                       loopBytes,sizeof(loopBytes),&szWriten);
    DebugActiveProcessStop(pProcInfo.dwProcessId);
    return 0;
}
```

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。

```
EIP→ ntdll.dll:774A9E4A ;
     ntdll.dll:774A9E4A nop
     ntdll.dll:774A9E4B 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。

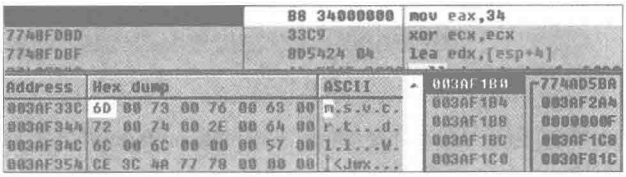


图 0.15

通过查看 NtOpenSection 的参数可知，LdrLoadDll 首先尝试用模块名称直接测试是否有该名称模块的内存映射。如果有，就直接通过 NtMapViewOfSection 函数尝试映射模块的内存空间。如果没有找到区段，那么我们系统首先通过 NtOpenFile 打开模块文件，再使用 NtMapViewOfSection 函数。NtMapViewOfSection 函数的定义如下。

```
typedef NTSTATUS  NTAPI LPNtMapViewOfSection(
IN HANDLE hSection,
IN HANDLE hProcess,
   IN OUT PVOID *BaseAddress,
   IN ULONG ZeroBits,
   IN ULONG CommitSize,
   IN OUT PLARGE_INTEGER SectionOffset OPTIONAL,
   IN OUT PULONG ViewSize,
   IN SECTION_INHERIT InheritDisposition,
   IN ULONG AllocationType,
IN ULONG Protect)
```

其中，参数 InheritDisposition 指明了映射类型。当该参数指定为 SEC_IMAGE 时，该函数将校验映射区块的文件格式。当内存映射完成，系统就开始向模块列表添加模块信息，然后转入 PE 的相关加载。为了观察 Windows 是如何加载 PE 程序的，我们可以观察 RtlImageDirectoryEntryToData 函数。我们要庆幸，PE 程序中大多数需要初始化的数据都由 PE 头中的 DataDirectory 目录指定，而 Windows 正好利用 RtlImageDirectoryEntryToData 函数定位数据目录并导出了该函数。该函数定义如下。

```
void WINAPI RtlImageDirectoryEntryToData(HMODULE hmod,void* a2,DWORD type,
LPDWORD lpSize)
```

其中，参数 type 定义了需要获取数据目录的索引，在系统完成映像内存映射后，我们就可以在该函数上下断点并进行观察。通过观察可以发现，在系统所有的数据目录加载过程中，对主模块来说导入表是最为重要的，对 DLL 模块来说重定位目录也是至关重要的，其他数据目录可以根据程序的不同进行选择。

当系统加载各目录以后，就进入模块入口代码的调用过程，控制权就移交给被加载的程序代码了。