Broadview
www.broadview.com.cn
PEARSON

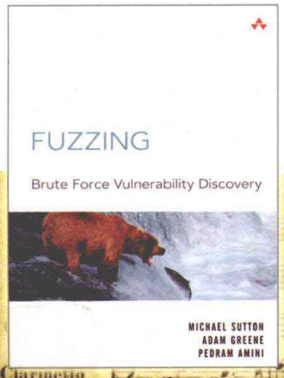
Jolt大奖精选丛书

模糊测试
强制发掘安全漏洞的利器

Fuzzing
Brute Force Vulnerability Discovery

[美] Michael Sutton
Adam Greene
Pedram Amini
著

段念 赵勇 译



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

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

模糊测试 强制发掘安全漏洞的利器

Fuzzing: Brute Force Vulnerability Discovery

掌握当今发现安全漏洞的最强大的方法之一！

模糊测试已经是当今最有效的测试软件安全性的方法。“模糊测试”方法使用随机数据作为被测程序的输入，然后系统性地找出这些输入数据导致的应用故障。模糊测试已经被黑客掌握了数年，现在，轮到你来使用了。在本书中，知名的模糊测试专家将教你如何比其他人更早发现你自己软件中的漏洞。

本书是第一本，也是到目前为止唯一的一本全面覆盖模糊测试的著作，系统性地给出了模糊测试的最佳实践。本书作者从模糊测试的概述开始，描述了模糊测试相较于其他安全性测试技术的主要优势。接着，作者介绍了在网络协议、文件格式和Web应用中发现漏洞的最先进的模糊测试技术，演示了如何使用模糊测试工具，并展示了一些实际的模糊测试案例。以下是本书覆盖的内容：

- 为什么模糊测试能够简化测试设计，能够抓住其他安全测试方法会错过的漏洞？
- 模糊测试过程：从确定模糊测试的输入到评估漏洞是否是“可利用的”
- 理解需要做什么才能进行有效的模糊测试
- 对基于变异(mutation-based)和基于生成(generation-based)的模糊测试器(fuzzer)进行比较
- 如何使用和自动化环境变量，以及如何进行参数模糊测试
- 掌握内存模糊测试技术
- 构建自定义的模糊测试框架与工具
- 如何实现智能错误检测

攻击者们已经在使用模糊测试了。显然，读者也应该使用这一技术。不管是开发者、安全性工程师、测试工程师，还是QA专员，这本书都能为你提供构建安全软件的知识。

www.awprofessional.com

www.fuzzing.org



上架建议：软件工程

ISBN 978-7-121-21083-9

9 787121 210839

定价：89.00元





PEARSON
www.pearson.com

PHEI

策划编辑：张春雨 符隆美
责任编辑：董 英
封面设计：李 玲

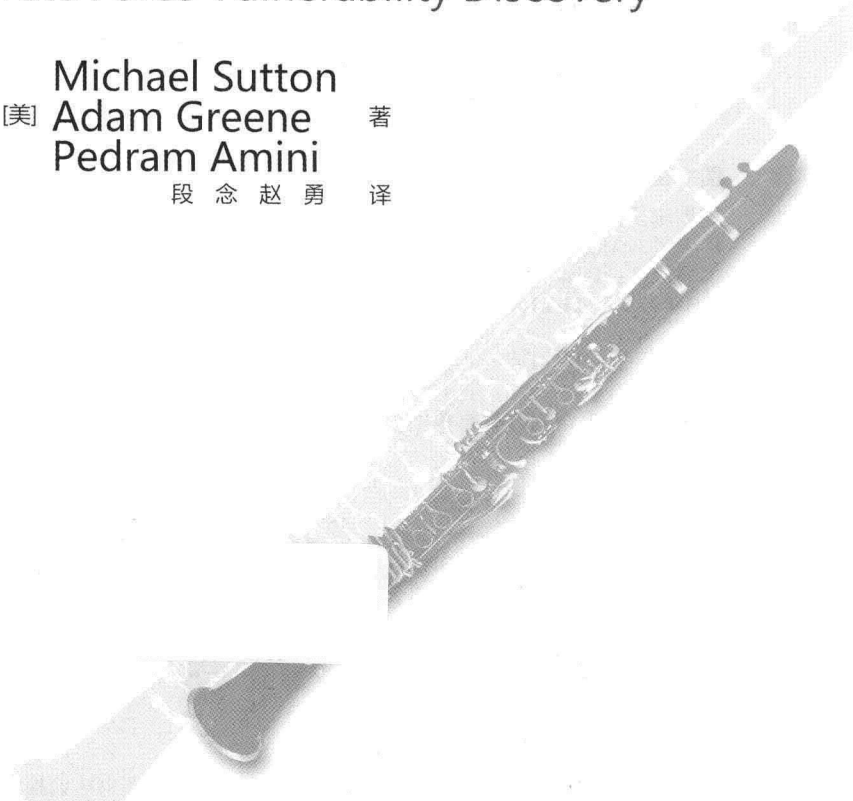
Jolt 大奖精选丛书

模糊测试
强制发掘安全漏洞的利器

Fuzzing
Brute Force Vulnerability Discovery

[美] Michael Sutton
Adam Greene
Pedram Amini
著

段念 赵勇 译



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

随着软件安全性问题变得越来越关键，传统的仅由组织内的少数安全专家负责安全的模式正受到越来越多的挑战。模糊测试是一种能够降低安全性测试门槛的方法，它通过高度自动化的手段让组织的开发和测试团队都能参与到安全性测试中，并能够通过启发式等方法不断积累安全测试的经验，帮助组织建立更有效的面向安全性的开发流程。本书是一本系统性描述模糊测试的专著，介绍了主要操作系统和主流应用类型的模糊测试方法，系统地描述了方法和工具，并使用实际案例帮助读者建立直观的认识。无论读者是否已有一定的安全性测试经验，本书都能够让你立即获得收益。

Authorized translation from the English language edition, entitled FUZZING: BRUTE FORCE VULNERABILITY DISCOVERY, 1E, 9780321446114 by Michael Sutton, Adam Greene, PedramAmini, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright© 2008 Pearson Education, Inc.
All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.
CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright©2013

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。
本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号　图字：01-2011-4223

图书在版编目（CIP）数据

模糊测试：强制发掘安全漏洞的利器 /（美）萨顿（Sutton, M.），（美）格林（Greene, A.），（美）阿米尼（Amini, P.）著；段念，赵勇译. —北京：电子工业出版社，2013.10
书名原文：Fuzzing: brute force vulnerability discovery
ISBN 978-7-121-21083-9

Ⅰ. ①模…　Ⅱ. ①萨…　②格…　③阿…　④段…　⑤赵…　Ⅲ. ①软件工具—测试　Ⅳ. ①TP311.56

中国版本图书馆 CIP 数据核字（2013）第 171019 号

策划编辑：张春雨　符隆美
责任编辑：董　英
印　　刷：北京中新伟业印刷有限公司
装　　订：河北省三河市路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编 100036
开　　本：787×980　1/16　印张：35.25　字数：711 千字
印　　次：2013 年 10 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

谨以此书献给我生命中最重要的两个女人。我的母亲，没有你的奉献，就没有这一切，愿这本书能为你的奉献提供一点补偿。Amanda，你对我坚定的爱和支持每时每刻都鼓舞着我，能够和你这样一个迷人的女人厮守真是我的幸运。

——Michael Sutton

谨以此书献给我的家庭和朋友，谢谢你们的支持和耐心。

——Adam Greene

以此书献给我的总司令：小布什先生。他虽然语言能力欠佳，却仍取得了巨大的成功。正是这一切激励着我，使我相信我也能够成为一本书的作者。

——Pedram Amini

译者序

翻译《模糊测试——强制发掘安全漏洞的利器》所花费的时间远超译者的预期，其过程也比想象得艰难很多。在开始翻译本书之前，译者曾乐观地估计可以较快地完成本书的翻译，可是真正开始之后才发现，由于本书的跨度大（从内容上，本书覆盖了 UNIX、Windows 操作系统的诸多底层知识；模糊测试的对象从本地应用跨越到网络应用和 Web 应用；在工具层面，展示了让人目不暇接的诸多工具；在扩展层面，还进入生物信息学，介绍了各种前沿的自动化协议分析的算法和工具），翻译起来让译者不免心中惴惴。翻译时遇到不能确定的地方，译者只能反复验证，尽力避免错漏；对于书中作者的小幽默，如果不能确切理解，也尽量反复参照，尽力体现作者的原意。不得不说，如果没有符隆美编辑的不断催促和鼓励，没有家人的支持，恐怕译者真不一定有勇气投入这么多的周末和假期，最终完成这本书的翻译。

不过，在“抱怨”投入了大量的时间来翻译这本书之外，翻译这本书的过程也着实让译者收获不小。模糊测试是一个并不算新的领域，但这个领域得到足够重视的历史并不算长。最早，模糊测试是安全研究者手中的秘密武器，他们通过这种强力武器发现应用中存在的问题，把自己变成像魔术师一样（这一点儿没有夸张，我原来一直仰视那些能发现 UNIX 中稀奇古怪的漏洞的安全人员，但当从本书中发现我也可以做到同样的事情时，就不再对这些曾经的“魔法师”心生敬畏了）。在那个年代，模糊测试仅出现在“美国黑帽大会”及其他以安全研究者为主的社区中，离普通的开发者和测试者甚为遥远。但随着软件产品的数量越来越多，以及越来越多的主流应用被部署在互联网上，安

全性问题成了一个越来越严重的问题。在这种背景下，模糊测试技术才被更多的人接受，并逐渐开始走入一般组织的开发和测试群体中。

模糊测试是一种通过高度自动化的手段，在产品发布之前发现产品中存在的安全漏洞的测试方法，它的高度自动化和可重复的特性使得那些重视安全的前沿软件企业早就接纳了它。虽然模糊测试并不是安全性测试的唯一手段，但这种手段的有效性和效率已经得到了诸多大的软件企业的证实。如本书所言，微软建立了自己的安全性开发生命周期 SDL，在整个产品开发周期中引入了模糊测试；而根据译者的经历，Google 也在自己的产品开发过程中大量和普遍地使用模糊测试，帮助发现可能导致服务器 DoS 问题或数据泄漏的漏洞。而且，在自动化之外，模糊测试还能够充分利用测试执行者已有的知识，通过启发式列表、智能数据集，甚至是遗传算法等方法不断提高模糊测试的有效性和效率。

本书可能不是模糊测试领域最权威或具有最高技术含量的书，但却绝对是最系统和最具实践性的书，没有之一。无论你是开发工程师还是测试工程师，无论你是否有模糊测试或安全性测试的经验，这本书都不会让你失望。相信书中对模糊测试的系统性描述、列举的立即可用的工具和精彩的案例分析，都能使读者对安全性和模糊测试有较为深入的思考和体会。作为一种能够有效降低安全性测试参与门槛的技术，模糊测试——你值得拥有。

本书的英文版成书于 2007 年（出版于 2007 年 7 月），书中作为示例的某些目标应用已经经历了多次大的版本更新，甚至退出了市场，书中描述的工具可能也已经经历了大的改进和提升，甚至书中提供的某些网站和网页都已经不复存在，但贯穿在书中的方法并未过时。即使在这个云计算已经不再新鲜、移动互联网应用已经开始崭露头角的时代，要想对现有的云应用和移动设备上的应用程序进行模糊测试，仍然可以在本书中轻易地找到对应的方法与工具。“时间是判断经典的最好方法”这句话对本书来说非常适用，尽管成书的时间离现在已有 6 年（在计算机领域，6 年可是相当长的时间！），但是，本书的核心价值丝毫没有被时间所影响。本书风格轻松，内容深入浅出（作者在序中特别提到，这本书在写作时尽量兼顾对模糊测试有实践经验和没有经验的读者），虽然覆盖了相当多的主题，但每个主题的讲述都相当系统，既有理论框架的讲解，也有工具使用的介绍，甚至还包括具体的案例分析。译者在翻译的过程中常常有豁然开朗的感觉，希望本书的各位读者在细品本书时，也同样有豁然开朗之后的喜悦之情。

当然，由于本书中涉及的内容较多，自然对读者的预备知识也有一定的要求。不过，本书的几位作者已经充分考虑到了对模糊测试领域不熟悉的读者的需求，因此，只要读者具有一定的编码基础（本书中使用了 Python、C#等多种编程语言，读者无须对本书涉及的这些语言都非常熟悉，但需要有一定的对编码的了解），对书中涉及的操作系统（Windows 和 Unix/Linux 操作系统）和计算机网络有所了解，就能够顺利地阅读本书，并从中获得不小的收益。

介绍完本书的内容，接下来对本书的翻译进行一些说明。首先，为了尽量保持本书的“原汁原味”，译者保留了原书的所有内容，包括原书的前言、序、致谢、索引等。对于书中饶有特色的每章开头的小布什的语录，本书也尽量保持原味，以原始英文的形式添加在每一章开头。小布什总统以语言能力差和浓重的德州口音著称，本书每一章开头引用的语录都带有非常有趣的错误（语法错误、辞不达意或其他问题），有兴趣的读者可以尝试自己端详和揣摩，看看能不能发现其中的有趣之处。当然，为了方便读者，本书最后的附录也给出了译者对每一条引用语录的解读，供读者参考。

本书的译者有两位，主要的译者是段念，翻译了本书的大部分内容并负责统稿，另一位译者是赵勇，承担了本书的第 9 章、第 10 章、第 14 章的翻译工作。虽然译者已经尽力使本书能够以较为完美的形态出现在读者面前，但由于译者能力所限，如在书中出现错误和疏漏，还请读者不吝指正。

最后，感谢电子工业出版社的符隆美编辑，没有她的耐心和坚定的推动，这本书的翻译是否能最终完成尚不可知；感谢各位参与稿件审阅的评委，你们的意见为本书增色不少；我个人还要感谢我的家人，感谢我的母亲对我们的照顾，感谢我的妻子和儿子，你们容忍了我在许多个周末和晚上把本应该陪你们的时间投入在本书上，在我快要失去耐心的时候继续给我动力。

段念

2013 年 4 月 8 日

前　言

漏洞是安全性研究的生命线。无论是执行渗透测试，还是评估新产品，亦或是审计关键组件的源代码——在这些活动中，漏洞都驱动着我们的决策，调整着时间分配，并影响着你接下来数年的方向。

源代码审计是一种白盒测试技术，这种技术被广泛应用于软件产品中以发现漏洞。源代码审计要求审计者了解产品所使用的每个编程概念和函数，并能深入理解产品的操作环境。而且，源代码审计存在明显的局限性——要对源代码进行审计，首先必须要有可用的源代码。

幸运的是，在没法得到源代码的情况下，我们还有黑盒测试方法。模糊测试是黑盒测试方法中的一种，已经证实，这种方法可以在产品中成功地发现不适合使用审计方法发现的核心漏洞。模糊测试的过程如下：向产品发送精心设计的非正常数据，期望这些数据触发一个错误条件或故障，错误条件能够导致可被利用的漏洞。

模糊测试并没有真正意义上的规则可以遵循。衡量模糊测试成功与否的唯一标准就是测试得到的结果。对给定的任何产品，都可能存在无数个输入。模糊测试需要预测产品中存在哪些类型的编程错误，以及哪些输入能够引发这些错误。这样看来，模糊测试更像是艺术而非科学。

模糊测试可以非常简单，甚至，在键盘上随机地敲击按键都可以是一种模糊测试。我有个朋友，他 3 岁的儿子曾经使用这种技术发现了 Mac OS X 操作系统的屏幕锁定功

能中的一个漏洞。我的这位朋友锁屏后，到厨房拿喝的。当他从厨房返回的时候，他的儿子已经设法解开了屏幕锁定，并打开了一个 Web 浏览器，而他儿子做的不过是随机地敲击键盘而已。

在过去的几年里，我使用模糊测试工具和技术在涉及多个行业的软件中发现了几百个漏洞。2003 年 12 月，我写了一个简单的工具，该工具向远程服务器发送随机的 UDP 数据包。这个简单工具发现了 Microsoft WINS 服务器中的两个新漏洞。一段时间之后，这个工具又帮助我发现了其他产品中的严重漏洞。看起来只需要一个随机 UDP 数据包流，我们就能在多个计算机相关的产品中发现漏洞，从 Norton 的 Ghost 管理服务，到 Mac OS X 操作系统提供的常用服务。

模糊测试器不仅仅对网络协议有用。2006 年的第一季度，通过混合使用三个不同的浏览器模糊测试工具，我发现了 Web 浏览器中的数十个漏洞。2006 年的第二季度，我写了一个 ActiveX 模糊测试器（AxMan），仅在微软的产品中该工具就发现了 100 个不同的漏洞。在“浏览器缺陷月”，我们对这些缺陷中的大部分进行了剖析，导致有人开发出了可以利用这些漏洞的 Metasploit 框架。即使在 AxMan 工具被开发出来一年之后，我仍然使用该工具发现了新的漏洞。模糊测试器真的是可以持续带来回报的东西。

本书是第一本将模糊测试作为一种技术来描述的书籍。本书提供了足够的知识使你可以开始模糊测试，能够构建自己的高效模糊测试工具。进行高效模糊测试的关键是知道针对什么产品该使用什么数据，以及了解需要操作、监视、管理模糊测试过程的工具。本书的几位作者都是模糊测试技术领域的先锋，本书出色地覆盖了模糊测试过程中各个复杂的部分。

H. D. Moore

原书序

"I Know the human being and fish can coexist peacefully."

——George W. Bush, Saginaw, Mich., Sept. 29, 2000

本书介绍

模糊测试的概念已经存在了将近 20 年，但直到最近，模糊测试才得到广泛的关注。2006 年，流行的客户端应用，包括微软的 Internet Explorer、Word 以及 Excel 等都受到了大量漏洞的影响，而这些漏洞中的大部分是通过模糊测试发现的。模糊测试的有效应用催生了新的工具，也导致越来越多的漏洞被发现。另外，作为第一本模糊测试方面的正式出版书籍，本书也说明了大众对模糊测试的兴趣越来越大。

由于多年来一直混迹于漏洞研究社区，我们已经在日常工作中使用过许多模糊测试技术，从不成熟的 hobby 项目到成熟的商业产品。本书的每位作者都拥有开发私有和公开的模糊测试器的经验。希望这本混合了我们几位作者的经验和研究项目的前沿书籍能够给读者带来收获。

适合阅读本书的读者

安全性书籍和文章通常由安全性研究者写成，目标读者同样是安全性研究者。我们

强烈地相信，如果安全性仍然只由安全性团队来负责，漏洞的数量和严重性将继续增长。因此，我们尽最大的努力使得本书可以面向更广泛的人群，包括模糊测试新手和在这方面有经验的读者。

指望仅仅在产品发布前由安全团队快速进行审计就能发布出安全的应用是不现实的。开发人员和 QA 团队再也不能说“安全性不是我的问题——我们有安全团队为此操心”了。安全性现在必须成为每个人的问题。安全性必须贯穿软件开发生命周期（SDLC）的各个阶段，而不是仅仅在最后阶段。

要求开发和 QA 团队关注安全性可能是一种苛求，尤其是对那些以前从未被如此要求的团队来说。我们相信模糊测试提供了一种面向更广受众的独特的漏洞发现方法，因为模糊测试可以高度自动化地运行。在期望安全研究者可以从本书中获得有价值信息的同时，我们也希望开发者和 QA 团队同样可以从本书中得到有价值的信息。模糊测试可以而且应该成为集成在软件开发生命周期中的一个部分，它不仅应该出现在测试阶段，也应该出现在开发阶段。越早发现缺陷，修复缺陷的代价就越低。

读者应该具有的预备知识（Prerequisites）

模糊测试是一个覆盖面很广的主题。在本书中我们涉及了一些并非专门面向模糊测试的主题，也假定读者具有一些预备知识。在阅读本书之前，读者至少应该对编程、计算机网络有基本的了解。模糊测试完全是自动化的安全测试，因此很自然地，本书的许多内容都与工具构建相关。我们故意为这些任务选择了多种编程语言。我们根据手头的任务选择编程语言，但编程语言的多样化也表明模糊测试可以使用多种方式达成。读者当然不必了解本书中用到的所有编程语言，但理解一两种语言将有助于读者从这些章节中收获更多。

在本书中我们描述了许多漏洞的细节，并讨论了如何通过模糊测试来发现这些漏洞。然而，定义或研究漏洞本身并不是我们的目标。目前已经有不少出色的书籍讨论了这方面的主题。如果你想要寻找软件漏洞方面的经典书籍，Greg Hoglund 的《软件剖析》（*Exploiting Software*）和《黑客大曝光》（*Hacking Exposed*）系列，以及 Jack Kiziol 和 David Litchfield 的《Shellcoder 手册》（*Shellcoder's Handbook*）等都是不错的参考。

如何使用本书

如何最好地利用本书取决于你的背景和目的。如果你是模糊测试新手，我们建议你按顺序阅读本书，因为前面章节提供的背景知识对理解后面章节的高级主题非常必要。不过，如果你已经使用过多种模糊测试工具，也不用担心直接跳到你感兴趣的主题会导致阅读困难，因为本书的多个逻辑部分和章节都具有相当的独立性。

本书的第一部分给出了后面章节中讨论的模糊测试类型的范围。如果你是模糊测试的新手，第一部分是不能错过的。模糊测试可用于发现任何目标中的漏洞，但是面向不同目标的所有方法都遵循同样的原则。在第一部分，我们将模糊测试定义为发现漏洞的方法，并详细描述了无论进行什么类型的模糊测试都需要了解的知识。

本书的第二部分聚焦于对特定类型目标进行模糊测试。我们以两章或三章为一组来描述对每种目标的模糊测试。每组的第一章提供了面向特定目标类型的背景信息，后续章节则聚焦于自动化，详细描述如何构造面向特定目标的模糊测试器。如果在 Windows 和 UNIX 平台上需要不同的工具，我们会安排两个与自动化相关的章节。例如，我们从第 11 章开始讲述“文件格式模糊测试”，该章描述了对文件处理器进行模糊测试的相关背景信息。第 12 章，“UNIX 平台上的文件格式自动化模糊测试”则详述了如何实现基于 UNIX 的文件模糊测试器，第 13 章“Windows 平台上的自动化文件格式模糊测试”详细描述了如何构造运行在 Windows 环境中的文件格式模糊测试器。

本书的第三部分讨论了模糊测试的高级主题。已经有很强模糊测试背景的读者，可以直接跳到第三部分，其他读者，我们建议先学习第一部分和第二部分，然后才到第三部分。在第三部分，我们专注于那些还处于早期阶段、但有可能在将来成为使用模糊测试方法的高级漏洞发现工具的核心部分的技术。

最后，在本书的第四部分，我们回顾通过本书学到的内容，并通过水晶球透视未来，看看未来还有什么值得研究的方向。模糊测试不是一个新概念，但它仍有许多发展空间，我们希望本书能够激励这个领域内的进一步研究。

本书中的幽默

写作一本书是一项严肃的工作，特别是写作一本关于模糊测试这样复杂的主题的书。在写作过程中，我们希望成为快乐的人（也许的确比一般人要快乐得多），并尽我们所能让写作过程充满乐趣。在这种指导思想下，我们决定用美利坚合众国第 43 任总统乔治·布什（也被戏称为“Dubya”）的语录作为本书每章的开头。无论你的政治倾向性或政治信仰如何，都不能否认小布什先生炮制了许多饶有趣味的语录，多得简直能够覆盖整个日历的 365 天[1]！我们从小布什的语录中选取了一些我们最喜欢的与你分享，希望你能和我们一样喜欢它们。从本书中你会了解到，模糊测试可以被用在许多目标上，即使是英语语言。

关于本书封面

很多时候，软件中的漏洞被比作“鱼”（例如，在来自 DailyDave 安全性邮件组的“L 单词与鱼（The L Word & Fish）”[2]这个主题中）。在讨论安全性和漏洞时，这是一种有用的比拟手段。安全研究者可以被称作渔夫。对某个应用一行行的汇编代码进行逆向工程查找漏洞的过程可以被当作“深海捕鱼”。与许多其他审计策略相比，模糊测试大多数时候只是停留在表面，能够有效地抓到“容易抓到”的鱼。此外，灰熊是著名的“毛绒动物（fuzzy）”，当然也是强有力的动物。我们选择本书封面的原因就在于此，熊代表模糊测试器，它抓住了一条鱼，而鱼代表漏洞。

本书的网站：www.fuzzing.org

fuzzing.org 网站是本书的一个组成部分，而不仅是提供附加资源的网站。除了提供本书的勘误之外，该网站还是本书提到的所有源代码和工具的中心资源仓库。随着时间的推移，我们期望 fuzzing.org 网站能够从一个以图书为中心的资源站点扩展为一个有价值的，包含与模糊测试原则相关的工具和信息的社区资源。我们欢迎读者的反馈，这些反馈能够帮助该网站成为一个有价值的、开放的知识库。

---

[1] http://tinyurl.com/33l54g

[2] http://archives.neohapsis.com/archives/dailydave/2004-q1/0023.html

致　谢

全体作者的致谢

虽然本书的封面上只出现了三个名字，但本书得以出版少不了背后的许多支持。首先是我们的朋友和家庭的支持，我们多次工作到深夜，无法在假期陪伴他们。在写作该书的过程中，我们错过了家庭中的啤酒、电影和安静的晚上，这是我们欠家人和朋友的。对那些受影响的人来说，恐怕我们是无法完全还清这些“债”了。虽然我们会怀念周四晚上定期的写作书的协作会议，但我们意识到其他人不大可能会怀念它们。

Peter DeVries 曾经说过：“我喜欢做个作家。我只是没法坚持在纸上工作。”对此，我们完全同意。有了写作的主意和想法只是战役的一半。有了草稿之后，一小队评审者加入了这场战役，希望说服世界我们能写作一本书。我们要对本书的技术编辑致以谢意，他们的工作无可挑剔。他们指出我们的错误，挑战我们的假设。特别需要提到的是 Charlie Miller，他对本书的深入审阅确保本书处于“前沿”位置。我们还要真诚地感谢 H.D. Moore 对本书的审阅，以及为本书撰写了前言。此外，我们还要感谢来自 Addison-Wesley 的团队，包括 Sheri Cain、Kristin Weinberger、Romny French、Jana Jones 和 Lori Lyons，他们在整个过程中给我们以指导。最后，我们还要特别感谢我们熟悉的编辑 Jessica Goldstein，他愿意给三个有着不靠谱想法、认为写书是件简单的事情的家伙一个机会。

Michael 的致谢

我想要利用这个机会感谢我的妻子 Amanda，感谢她在我写作本书时的耐心和理解。在写作本书时，我们正在筹划一场婚礼，有太多本应和她一起悠闲地享用红酒的晚上和周末，我都对着计算机屏幕。我还要真诚地感谢我的所有家庭成员给我的支持，他们鼓励我继续这个项目并真的相信我能行。感谢 iDefense Labs 团队和我在 SPI Dynamics 的伙伴们，他们支持我并在整个过程中不断激励我。最后，我想要谢谢和我合作的作者，激励着我从偶然为之的演讲发展到这本书，并和我一起生产出了这本远超我单凭个人力量所能产出的书籍。

Adam 的致谢

我想要感谢我的家庭（尤其是我的姐妹和父母）、JTHS 的老师和 counselors、Mark Chegwidden、Louis Collucci、ChrisBurkhart、sgo、Nadwodny、Dave Aitel、Jamie Breiten、Davis 一家、Brothers Leondi、Reynolds、Kloub 和 AE、Lusardi、Lapilla，以及最后，Richard。

Pedram 的致谢

我要感谢和我合作的作者使得我有机会参与这本书的写作，感谢他们在漫长的过程中给我的灵感。我还要感谢 Cody Pierce、Cameron Hotchkies 和 Aaron Portnoy，我在 TippingPoint 的团队，感谢他们天才的主意和他们的技术评审。感谢 Peter Silberman、Jamie Butler、Greg Hoglund、Halvar Flake 和 Ero Carrera，感谢他们提供的灵感和无尽的欢乐。特别感谢 David Endler、Ralph Schindler、Sunil James 和 Nicholas Augello，我们不是兄弟却胜似兄弟，他们总是我可以信赖的依靠。最后，衷心感谢我的家庭在我完成这本书的过程中给予我的宽容和耐心。

关于作者

Michael Sutton

Micheal Sutton 是 SPI Dynamics 的安全性专业顾问（Security Evangelist）。作为安全性专业顾问，Michael 负责识别、研究及提出 Web 应用安全性历史上出现的问题。他经常在主要的信息安全会议上演讲，写作了大量文章，并在多种信息安全主题中被引用。Michael 同时也是 Web 应用安全性协会（Web Application Security Consortium，WASC）的会员，领导着 Web 应用安全性统计项目。

在加入 SPI Dynamics 之前，Michael 是 iDefense/VeriSign 的总监，领导 iDefense Labs 这个世界级的研究团队，该团队的目标是发现和研究安全性漏洞。Michael 还在百慕大建立了为安永服务的信息系统保证和咨询服务（Information System Assurance and Advisory Services，ISAAS）。Michael 拥有阿尔伯塔大学和乔治华盛顿大学的学位。

Michael 是一个骄傲的加拿大人，在他看来，曲棍球是一种信仰而不只是一种运动。在工作之外，Micheal 还是 Fairfax 志愿救火部门的一名士官。

Adam Greene

Adam Greene 是某大金融新闻公司的工程师，公司位于纽约。在加入这家公司之前，

Adam是iDefense的工程师，iDefense是位于弗吉尼亚州雷斯顿城的一家智能公司。Adam在计算机安全性方面的兴趣主要在于可信利用（reliable exploitation）方法、模糊测试，以及开发基于UNIX系统的利用漏洞的工具。

Pedram Amini

Pedram Amini目前领导着TippingPoint的安全研究和产品安全评估团队，在此之前，他是iDefense Labes的助理总监和创始人之一。顶着“助理总监”这个奇怪的头衔，Pedram在逆向工程的基础方面投入了相当多的时间——开发自动化工具、插件及脚本。他最近的项目（也叫“Babies”）包括PaiMei逆向工程框架和Sulley模糊测试框架。

出于热情，Pedram创建了OpenRCE.org，一个专注于逆向工程的艺术和科学的社区网站。他在RECon、BlackHat、DefCon、ShmooCon和ToorCon上都进行过演讲，并教授了多次逆向工程课程。Pedram拥有杜兰大学的计算机科学学位。

目　录

译者序 ····· V
前言 ····· IX
原书序 ····· XI
致谢 ····· XV
关于作者 ····· XVII

第一部分　基础知识

第 1 章　安全漏洞发现方法学 ····· 2

1.1　白盒测试 ····· 3
1.1.1　代码评审（Source Code Review） ····· 3
1.1.2　工具与自动化 ····· 5
1.1.3　优点和缺点 ····· 7
1.2　黑盒测试 ····· 8
1.2.1　手工测试 ····· 8
1.2.2　自动化测试或模糊测试 ····· 10
1.2.3　优点和缺点 ····· 11
1.3　灰盒测试 ····· 12
1.3.1　二进制审计 ····· 12

1.3.2 自动化的二进制审计……15
1.3.3 优点和缺点……15
1.4 小结……16
第 2 章 什么是模糊测试……17
2.1 模糊测试的定义……17
2.2 模糊测试的历史……18
2.3 模糊测试各阶段……22
2.4 模糊测试的局限性和期望……24
2.4.1 访问控制漏洞……25
2.4.2 糟糕的设计逻辑……25
2.4.3 后门……26
2.4.4 破坏……26
2.4.5 多阶段安全漏洞（MuliStage Vulnerability）……27
2.5 小结……27
第 3 章 模糊测试方法与模糊测试器 类型……28
3.1 模糊测试方法……28
3.1.1 预生成测试用例……29
3.1.2 随机生成输入……29
3.1.3 手工协议变异测试……30
3.1.4 变异或强制性测试……30
3.1.5 自动协议生成测试……31
3.2 模糊测试器类型……31
3.2.1 本地模糊器……31
3.2.2 远程模糊测试器……34
3.2.3 内存模糊测试器……37
3.2.4 模糊测试框架……38
3.3 小结……39
第 4 章 数据表示和分析……40
4.1 什么是协议……40

4.2 协议中的字段……41
4.3 简单文本协议（Plain Text Protocols）……43
4.4 二进制协议……44
4.5 网络协议……47
4.6 文件格式……48
4.7 常用协议元素……51
4.7.1 名字-值对……51
4.7.2 块识别符……51
4.7.3 块大小……52
4.7.4 校验和……52
4.8 小结……52
第 5 章 有效模糊测试的需求……53
5.1 可重现性与文档……53
5.2 可重用性……54
5.3 过程状态和过程深度……55
5.4 跟踪、代码覆盖和度量……58
5.5 错误检测……58
5.6 资源约束……60
5.7 小结……60
第二部分 目标与自动化
第 6 章 自动化与数据生成……62
6.1 自动化的价值……62
6.2 有用的工具和库……63
6.2.1 ETHEREAL/WIRESHARK……64
6.2.2 LIBDASM 和 LIBDISASM……64
6.2.3 LIBNET/LIBNETNT……64
6.2.4 LIBPCAP……65
6.2.5 METRO PACKET LIBRARY……65
6.2.6 PTRACE……65

6.2.7 PYTHON 扩展……65
6.3 编程语言的选择……66
6.4 数据生成与模糊试探值（Fuzz Heuristics）……66
6.4.1 整数值……68
6.4.2 字符串重复（String Repetitions）……70
6.4.3 字段分隔符……71
6.4.4 格式字符串……73
6.4.5 字符翻译……73
6.4.6 目录遍历……74
6.4.7 命令注入……75
6.5 小结……75
第 7 章 环境变量与参数模糊测试……76
7.1 本地模糊测试介绍……76
7.1.1 命令行参数……76
7.1.2 环境变量……77
7.2 本地模糊测试原则……78
7.3 寻找测试目标……79
7.3.1 UNIX 文件权限释义……81
7.4 本地模糊测试方法……82
7.5 枚举环境变量……83
7.5.1 GNU 调试器（GNU Debug，GDB）法……83
7.6 自动化的环境变量模糊测试……84
7.6.1 库预加载（Library Preloading）……85
7.7 检测问题……86
7.8 小结……88
第 8 章 自动化的环境变量与参数模糊测试……89
8.1 iFUZZ 本地模糊测试器的功能……89
8.2 开发 iFUZZ 工具……91
8.2.1 开发方法……92

8.3 iFUZZ 使用的编程语言……95
8.4 案例研究……96
8.5 优点与改进……97
8.6 小结……98
第 9 章 Web 应用与服务器模糊测试……99
9.1 什么是 Web 应用模糊测试……99
9.2 测试目标……102
9.3 测试方法……104
9.3.1 设置目标环境……104
9.3.2 输入……105
9.4 漏洞……116
9.5 异常检测……119
9.6 小结……120
第 10 章 Web 应用和服务器的自动化模糊测试……121
10.1 Web 应用模糊测试器……122
10.2 WebFuzz 的特性……124
10.2.1 请求……124
10.2.2 模糊变量……125
10.2.3 响应……126
10.3 必备的背景信息……128
10.3.1 识别请求……128
10.3.2 检测响应……128
10.4 开发 WebFuzz……131
10.4.1 思路……131
10.4.2 选择编程语言……131
10.4.3 设计……131
10.5 案例研究……139
10.5.1 目录遍历（Directory Traversal）……139
10.5.2 溢出……140

10.5.3 SQL 注入 ····· 143
10.5.4 XSS 脚本 ····· 145
10.6 优点与可能的改进 ····· 148
10.7 小结 ····· 149
第 11 章 文件格式模糊测试 ····· 150
11.1 测试目标 ····· 151
11.2 测试方法 ····· 152
11.2.1 强制或基于变异的模糊测试 ····· 152
11.2.2 智能强制或基于生成的模糊测试 ····· 154
11.3 测试输入 ····· 154
11.4 安全漏洞 ····· 155
11.4.1 拒绝服务（Daniel of Service，DoS） ····· 155
11.4.2 整数处理问题 ····· 156
11.4.3 简单的栈和堆溢出 ····· 157
11.4.4 逻辑错误 ····· 157
11.4.5 格式字符串 ····· 158
11.4.6 竞争条件（Race Condition） ····· 158
11.5 检测错误 ····· 158
11.6 小结 ····· 159
第 12 章 UNIX 平台上的文件格式自动化模糊测试 ····· 161
12.1 notSPIKEfile 和 SPIKEfile ····· 162
12.1.1 不具有的特性 ····· 162
12.2 开发过程 ····· 162
12.2.1 异常监测引擎 ····· 163
12.2.2 异常报告（异常监测） ····· 163
12.2.3 模糊测试核心引擎 ····· 164
12.3 有意义的代码片段 ····· 165
12.3.1 UNIX 中常见的我们可能感兴趣的信号 ····· 167
12.3.2 我们不感兴趣的信号 ····· 167

12.4 僵尸进程（Zombie Process）……168
12.5 使用注意事项……170
12.5.1 Adobe Acrobat……171
12.5.2 RealNetworks RealPlayer……171
12.6 案例研究：RealPlayer RealPix 格式字符串漏洞……172
12.7 开发语言……174
12.8 小结……174
第 13 章 Windows 平台上的文件格式自动化模糊测试……175
13.1 Windows 文件格式漏洞……175
13.2 FileFuzz 工具的功能……178
13.2.1 创建文件……179
13.2.2 执行应用……180
13.2.3 异常检测……181
13.2.4 保存的审计（audit）……182
13.3 必需的背景信息……183
13.3.1 识别目标应用……183
13.4 开发 FileFuzz 工具……186
13.4.1 开发方法……187
13.4.2 选择开发语言……187
13.4.3 设计……187
13.5 案例研究……194
13.6 优势和提升空间……198
13.7 小结……199
第 14 章 网络协议的模糊测试……200
14.1 什么是网络协议的模糊测试……201
14.2 选择目标应用……203
14.2.1 第二层：数据链接层……204
14.2.2 第三层：网络层……205
14.2.3 第四层：传输层……205

14.2.4 第五层：会话层 206
14.2.5 第六层：表示层 206
14.2.6 第七层：应用层 206
14.3 测试方法 207
14.3.1 强制（基于变异的）模糊测试 207
14.3.2 智能强制（基于生成的）模糊测试 207
14.3.3 通过修改客户端进行变异模糊测试 208
14.4 错误检测 209
14.4.1 手工方式（基于调试器） 209
14.4.2 自动化方式（基于代理） 209
14.4.3 其他来源 210
14.5 小结 210
第 15 章 UNIX 平台上的自动化网络协议模糊测试 211
15.1 使用 SPIKE 进行模糊测试 212
15.1.1 选择目标 212
15.1.2 协议分析 213
15.2 SPIKE 必要知识 215
15.2.1 模糊引擎 216
15.2.2 基于行的通用 TCP 模糊测试器 216
15.3 基于块的协议模型 217
15.4 其他的 SPIKE 特性 219
15.4.1 针对协议的模糊测试器 219
15.4.2 针对协议的模糊测试脚本 219
15.4.3 基于脚本的通用模糊测试器 220
15.5 编写 SPIKE NMAP 模糊测试器脚本 220
15.6 小结 224
第 16 章 Windows 平台上网络协议的模糊测试 225
16.1 特性 226
16.1.1 数据包结构 226

16.1.2 抓取数据……227
16.1.3 解析数据……227
16.1.4 模糊测试变量……228
16.1.5 发送数据……229
16.2 必备的背景知识……229
16.2.1 检测故障……229
16.2.2 协议驱动程序……230
16.3 开发……231
16.3.1 选择编程语言……231
16.3.2 数据包抓取库……231
16.3.3 设计……232
16.4 案例研究……237
16.5 优势与可改进空间……239
16.6 小结……240
第 17 章 Web 浏览器的模糊测试……241
17.1 什么是 Web 浏览器模糊测试……242
17.2 目标……242
17.3 方法……243
17.3.1 测试方法……243
17.3.2 测试输入……244
17.4 漏洞……253
17.5 检测……254
17.6 小结……255
第 18 章 Web 浏览器的自动化模糊测试……256
18.1 组件对象模型背景……256
18.1.1 COM 简史……257
18.1.2 对象与接口……257
18.1.3 ActiveX……258
18.2 开发模糊测试器……260

18.2.1 枚举可被加载的 AcriveX 控件……261
18.2.2 属性、方法、参数与类型……266
18.2.3 模糊测试与监视……270
18.3 小结……271
第 19 章 内存模糊测试……272
19.1 为什么需要内存模糊测试？怎么进行？……273
19.2 必要的背景知识……273
19.3 究竟什么是内存模糊测试的简要解释……277
19.4 目标……278
19.5 内存模糊测试方法之变异循环插入（Mutation Loop Insertion）……279
19.6 内存模糊测试方法之快照恢复变异（Snapshot restoration mutation）……280
19.7 测试速度与处理深度……281
19.8 错误检测……281
19.9 小结……282
第 20 章 自动化内存模糊测试……284
20.1 内存模糊测试工具特性集……284
20.2 选择开发语言……286
20.3 Windows 调试 API……288
20.4 整合以上的内容……292
20.4.1 如何在目标应用中特定的点放置“钩子”……292
20.4.2 如何生成与恢复进程快照……295
20.4.3 选择在何处放置钩子……299
20.4.4 如何定位和变异目标内存空间……299
20.5 PyDbg，一个新朋友……299
20.6 一个人造的示例……301
20.7 小结……314
第三部分 高级模糊测试技术
第 21 章 模糊测试框架……318
21.1 什么是模糊测试框架……319

21.2 现有的模糊测试框架……321
21.2.1 antiparser……321
21.2.2 Dfuz……323
21.2.3 SPIKE……327
21.2.4 Peach……330
21.2.5 通用目的模糊测试器……333
21.2.6 Autodafé……335
21.3 定制模糊测试器案例研究：Shockwave Flash……338
21.3.1 为 SWF 文件建模……339
21.3.2 生成有效的数据……350
21.3.3 模糊测试环境……351
21.3.4 测试方法……351
21.4 模糊测试框架 Sulley……352
21.4.1 Sulley 的目录结构……353
21.4.2 数据表示……354
21.4.3 会话……365
21.4.4 事后分析……370
21.4.5 一个完整的实例……375
21.5 小结……382
第 22 章 自动化协议分析……383
22.1 模糊测试的痛处……383
22.2 启发式技术……385
22.2.1 代理模糊测试……385
22.2.2 改进的代理模糊测试……388
22.2.3 反汇编启发式技术……389
22.3 生物信息学……390
22.4 遗传算法……394
22.5 小结……398
第 23 章 模糊测试器跟踪……399
23.1 我们跟踪的究竟是什么……399

23.2 可视化和基础块……401
23.2.1 控制流图……402
23.2.2 控制流图示例……402
23.3 构建一个模糊测试器跟踪器……403
23.3.1 分析目标……404
23.3.2 跟踪……405
23.3.3 交叉参考……407
23.4 分析一个代码覆盖工具……410
23.4.1 PStalker 布局预览……411
23.4.2 数据源栏……412
23.4.3 数据浏览栏……413
23.4.4 数据抓取栏……414
23.4.5 局限性……414
23.4.6 数据存储……414
23.5 案例研究……417
23.5.1 测试策略……418
23.5.2 实际操作……421
23.6 优势与将来的改进……425
23.6.1 将来的改进……427
23.7 小结……428
第 24 章 智能错误检测……429
24.1 原始的错误检测技术……429
24.2 我们寻找的是什么……432
24.3 选择模糊测试值的注意事项……437
24.4 自动化的调试器监视……438
24.4.1 一个基础的调试器监视器……439
24.4.2 更高级的调试器监视器……442
24.5 首轮异常与末轮异常……446
24.6 动态二进制插装……447
24.7 小结……449

第四部分　展望

第 25 章　我们学到了什么 ····· 452

25.1　软件开发生命周期 ····· 452

25.1.1　分析阶段 ····· 455

25.1.2　设计阶段 ····· 455

25.1.3　编码阶段 ····· 456

25.1.4　测试阶段 ····· 456

25.1.5　维护阶段 ····· 457

25.1.6　在软件开发生命周期中实现模糊测试 ····· 457

25.2　开发者 ····· 457

25.3　QA 研究员 ····· 458

25.4　安全研究者 ····· 458

25.5　小结 ····· 459

第 26 章　展望 ····· 460

26.1　商业工具 ····· 460

26.1.1　Beyond Security 公司的 beSTORM ····· 461

26.1.2　BreakingPoint Systems 的 BPS-1000 ····· 461

26.1.3　Codenomicon ····· 462

26.1.4　GLEG ProtoVer 专业版 ····· 464

26.1.5　MU Security 公司的 MU-4000 ····· 465

26.1.6　Security Innovation 公司的 Holodeck ····· 466

26.2　发现漏洞的混合方法 ····· 467

26.3　集成的测试平台 ····· 468

26.4　小结 ····· 468

附录 A　本书引用的小布什语录之详细解读 ····· 469

索引 ····· 480

第一部分

基础知识

第 1 章　安全漏洞发现方法学
第 2 章　什么是模糊测试
第 3 章　模糊测试方法与模糊测试器类型
第 4 章　数据表示和分析
第 5 章　有效模糊测试的需求

第 1 章

安全漏洞发现方法学

"Will the highways of the Internet become more few?"

——George W. Bush, Concord, N.H., January 29,2000

不管向哪位小有成就的安全领域研究者请教如何发现安全漏洞，你通常都会得到一大堆的答案。为什么会这样？这是因为用于发现安全漏洞的方法多种多样，每一种方法都有各自的优缺点。既没有绝对正确的方法，也没有任何一种单一方法可以发现特定目标的所有安全漏洞。从较高的层面上说，用于发现安全漏洞的方法主要是黑盒测试、白盒测试和灰盒测试三种。不同的方法需要能够访问到被测系统的不同资源。白盒测试是这三种方法中的一个极端，该方法要求测试者能够访问到被测系统的全部资源：源代码、设计文档、甚至是实现系统的程序员；黑盒测试则是三种方法中的另一个极端，该方法几乎不要求任何被测系统的内部知识，大部分情况下只是对被测系统的“盲测”。例如，对一个远程 Web 应用进行渗透测试（Penetration Test，也被称为 Pen Test）就主要采用黑盒测试的方法。灰盒测试介于白盒测试与黑盒测试之间，它的定义因人而异。就本书所讨论的范围，我们讨论的灰盒测试方法要求被测系统编译后的二进制执行文件，以及部分基础文档。

在本章中，我们将从白盒测试开始探索多种高层和低层的安全漏洞发现方法。白盒测试有时也被称为净盒测试（clear box testing）、玻璃盒测试（glass box testing）或是半透明盒测试（translucent box testing）。探讨白盒测试之后，我们将为黑盒测试和灰盒测试下定义，同时也会为采用了灰盒和黑盒测试方法的模糊测试（fuzzing）下定义。接下

来我们将阐述每种方法的优缺点，为读者提供后续模糊测试讨论的必要背景知识。要说明的是，模糊测试只是发现安全漏洞的一种方法，了解其他可用的安全漏洞发现方法也很重要。

1.1 白盒测试

作为一套测试方法，模糊测试主要落在黑盒和灰盒测试的领域内。虽然如此，在本章中我们还是从一种被开发工程师广泛采用的安全漏洞发现方法开始我们的讨论。

1.1.1 代码评审（Source Code Review）

既可以采用手工方式，也可以使用自动化工具辅助来进行代码评审。考虑到计算机程序通常包括几万到几十万行代码，完全依赖手工进行代码评审是不现实的。自动化工具在减少枯燥的逐行仔细研读代码工作方面功效卓著，但却只能用来发现可能的漏洞或是可疑的代码片段。自动化工具发现的问题是否是有效的问题仍然需要通过人工来判断。

源代码分析工具要解决很多问题才能产生有用的分析结果，详细讨论所有这些问题已经超出了本书的范围。我们仅用一段简单的 C 语言代码来展示源代码分析工具可能遇到的问题。以下这段 C 语言代码将字符串“test”拷贝到一个 10 字节的字符数组中。

```
#include <string.h>

Int main(int argc, char ** argv)
{
  char buffer[10];
  strcpy(bufferm, "test");
}
```

接下来的这段代码与上一段代码基本相同，唯一的不同是将用户的输入拷贝到 10 字节的字符数组中。

```
#include <string.h>

Int main(int argc, char ** argv)
{
  char buffer[10];
  strcpy(bufferm, argv[1]);
}
```

以上两段代码都是用 strcpy() 函数将数据拷贝到基于栈的缓冲区中。C/C++中一般不鼓励使用 strcpy() 函数，因为该函数缺乏对拷贝的目的地址的边界检查。如果一个程序员在使用 strcpy() 函数时没有小心地加入检查代码，很可能会导致缓冲区溢出，使得缓冲区外的数据被拷贝数据所覆盖。在第一段代码中，“test”字符串的长度总是 5（包括字符串结束符），该长度小于目的缓冲区的长度，因此不会发生溢出。但对第二段代码来说，如果用户输入的命令行参数大于 10 就可能会发生缓冲区溢出。是否存在安全漏洞的关键在于，用户是否能够控制可能含有安全漏洞的方法的输入。初步的代码检查可能会将两段代码中的 strcpy() 函数都标示为可能的安全漏洞，但只有对输入数据进行检查后才能确认是否真的存在安全漏洞。代码评审是有效的安全性研究武器，一旦代码可用，就应该立即开始代码评审。但是，并非在所有情况下都能进行代码评审（例如，访问不到源代码的情况下，就无法进行代码评审）。

人们往往假设白盒测试在安全漏洞发现方面比黑盒测试的效率更高，但这种假设是不正确的。白盒测试基于对源代码的评审，难道还有什么视角能比源代码更好或是更全面地展示软件？当然没有。但请记住，几乎不可能在你看到的功能表现和你评审的源代码之间建立严格的对应关系，因为软件构建过程在将源代码编译成汇编代码时引入了巨大的改变。这是为什么不能说某种测试方法一定比另一种测试方法更好的原因之一。不同的测试方法能够发现不同的安全漏洞。要达到充分的覆盖，就需要混合应用多种测试方法。

微软源代码泄露事件

要证明代码评审在发现安全漏洞方面并不见得好于黑盒测试的效果，让我们考虑 2004 年 1 月发生的微软源代码泄露事件。2004 年 1 月，有传言说微软 Windows NT 4.0 和 Windows 2000 操作系统的许多源代码被传播到互联网上。随后微软证实了这些源代码是真实的。不少公司极其担心这次代码泄露会导致大量针对 Windows NT 4.0 和 Windows 2000 的安全漏洞被发现，但他们的担心并未成为现实。实际上，时至今日也仅有少数的安全漏洞是通过这些被泄露的源代码发现的。CVE-2004-0566 是通过源代码发现的安全漏洞中的一个，该漏洞的细节是当操作系统处理位图文件时可能会发生整数溢出[1]。有趣的是，微软否认这个漏洞是通过泄露的源代码被发现的，声称该漏洞在内部审计中早已被发现[2]。为什么这次源代码泄露并没有导致大量的安全漏洞被发现？难道通过源代码不应该发现所有的安全漏洞吗？事实是，代码评审确实是对应用和操作系

[1] Http://archives.neohapsis.com/archives/fulldisclosure/2004-02/0806.html

[2] http://news.zdnet.com/2100-1009_22-5160566.html

统安全审计中的重要一环，但由于代码规模和复杂性，代码评审难以充分的进行。此外，反汇编分析技术也能在代码分析这个级别上发挥作用。以 TinyKRNL[3]和 ReactOS[4]项目为例，这两个项目都是以提供微软 Windows 内核和操作系统兼容性为目标的。这两个项目的开发工程师都不能访问到 Windows 内核的源代码，但仍然能够创建出一定程度上与 Windows 兼容的环境，如果要审核 Windows 操作系统，在没有 Windows 源代码的情况下，TinyKRNL 和 ReactOS 项目的源代码可以作为解释反编译后的 Windows 汇编码的指南。

1.1.2　工具与自动化

安全方面的源代码分析工具通常可以分为三类：编译时检查工具、源代码浏览工具、以及自动源代码审计工具。编译时检查工具在源代码编译时查找安全漏洞，该工具通常与编译器集成在一起，主要查找安全性相关的问题而不是功能相关问题。微软的 Visual C++工具的/analyze 编译开关就是一个编译时检查工具的例子[5]。此外，微软还提供了 PREfast for Drivers 工具[6]，该工具可以帮助驱动开发工程师发现驱动程序中多种编译器无法发现的安全漏洞。

源代码浏览器通常用来帮助进行源代码的检查与评审。这类工具允许评审者在所有代码中进行搜索，枚举代码使用的函数和变量，以及在源代码中基于交叉引用进行方便的跳转。例如，评审者可以使用该类工具找到源代码中所有使用了 strcpy()的地方，以便于找到可能的安全漏洞。Cscope 工具[7]和 Linux CrossReference 工具[8]都是常用的源代码浏览工具。

自动源代码审计工具被设计用来扫描源代码以及自动发现需要关注的代码区域。市面上有许多商业和开源的这类工具可以选择。由于这类工具通常是面向特定编程语言的，因此，如果被测应用是使用多种编程语言开发的，你可能需要使用多种这类工具。

---

[3] http://www.tinykrnl.org/

[4] http://www.reactos.org/

[5] http://msdn2.microsoft.com/en-us/library/k3a3hzw7.aspx

[6] http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx

[7] http://cscope.sourceforge.net/

[8] http://lxr.linux.no/

Fortify[9]，Coverity[10]，KlockWork[11]，GrammaTech[12]和其他一些工具都提供了这类工具的商业解决方案。表 1.1 展示了一些常用的该类免费工具，并列出了它们面向的编程语言和支持的环境。

表 1.1 免费源代码审计工具

工具名称	面向的编程语言	支持的环境	下载地址
RATS (Rough Auditing Tools for Security)	C, C++, Per, PHP, Python	UNIX, Win32	http://www.fortifysoftware.com/security-resources/rats.jsp
ITS4	C, C++	UNIX, Win32	http://www.cigital.com/its4/
Splint	C	UNIX, Win32	http://lclint.cs.virginia.edu/
Flawfinder	C, C++	UNX	http://www.dwheeler.com/flawfinder/
Jlint	Java	UNIX, Win32	http://jlint.sourceforge.net/
CodeSpy	Java	Java	http://www.owasp.org/software/labs/codespy.html

要记住的是，任何自动化工具都不能替代有经验的安全研究员的经验。工具只不过能将繁重的代码检查工作简化，帮助节约时间，但工具生成的报告仍需要由有经验的分析员审核，识别出其中真正的问题，并需要开发工程师修复问题。例如，下面一段内容是 Rough Auditing Tools for Security（RATS）工具针对前文所展示的两段代码生成的安全审计报告，在报告中，RATS 指出了两处可能的安全问题。RATS 工具报告说由于使用定长的缓冲区以及使用 strcpy()都可能导致不安全。但是，RATS 工具并没有明确指出安全漏洞所在，它只是提醒工具的使用者注意到可能的不安全的代码区域，而依赖使用者的经验来判定这些区域是否真的存在安全问题。

```
Entries in Perl database: 33
Entries in Python database: 62
Entries in C database: 334
Entries in php database: 55
Analyzing userinput.c
userinput.c:4: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated on
the stack are used safely. They are prime targets for buffer overflow attacks.
```

[9] http://www.fortifysoftware.com/

[10] http://www.coverity.com/

[11] http://www.klockwork.com/

[12] http://www.grammatech.com/

```
userinput.c:5: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy more
data than can be handled, resulting in a buffer overflow.

Total line analyzed: 7
Total time 0.000131 seconds
53435 lines per second


Entries in Perl database: 33
Entries in Python database: 62
Entries in C database: 334
Entries in php database: 55
Analyzing userinput.c
userinput.c:4: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated on
the stack are used safely. They are prime targets for buffer overflow attacks.

userinput.c:5: High: strcpy
Check to be sure that argument 2 passed to this function call will not copy more
data than can be handled, resulting in a buffer overflow.

Total line analyzed: 7
Total time 0.000794 seconds
8816 lines per second
```

1.1.3　优点和缺点

如前所述，没有单一“正确”方法可以用来发现所有的安全漏洞。那么，需要如何选择合适的安全性测试方法呢？选择何种方法的决定权在于我们自己。举例来说，如果我们不具有访问被测系统的源代码的权限，我们就不能在安全测试中使用白盒测试方法（大多数安全研究员和最终用户，尤其是那些在微软的 Windows 环境中使用商业软件的人，都面临着不能获得被测系统源代码的情况）。白盒测试的优点到底在什么地方？

- **高覆盖**：代码评审是针对所有源代码进行的，因此，代码评审可以达到完全的覆盖。理论上，针对所有可能的代码路径都进行审计可以发现所有的安全漏洞。当然，由于并非所有代码路径都会在执行时被跑到，这种审计可能会导致发现一些无效的安全漏洞。

代码评审并不总是可行的。即使有条件进行代码评审，代码评审也不应该是唯一的安全漏洞发现方法，而应该和其他方法混合应用。代码分析方法存在以下这些不足。

- **过于复杂**：源代码分析工具并不完美，容易产生误报。因此，工具生成的错误报告并非完全可信赖。工具生成的报告需要有经验的程序员人工审核以发现真正的安全性问题。由于典型的软件通常包含数十万行代码，工具生成的报告可能会非常冗长，并需要花费大量时间来审核。
- **可用性**：被测系统的源代码并非总是可获得。虽然许多 UNIX 项目是开源项目，用户可获得项目的源代码并对其进行评审，但在 Win32 环境中这种状况就不多见——商业软件通常都不会提供源代码。没有源代码，白盒测试是不可能进行的。

1.2 黑盒测试

黑盒测试意味着仅通过外部观测来测试系统。被测应用就像一个黑盒子，测试者像用户一样控制进入黑盒子的输入，观测从黑盒子另一端产生的输出。测试者不需要知道被测系统内部是如何工作的。远程访问一个 Web 应用或是 Web 服务典型的就是这种状况。用户可以输入 HTML 或是 XML 形式的内容，观测服务器返回的页面或返回值，但不必知道这些页面或是返回值是如何产生的。

考虑另一个例子：假设你购买了微软的 Office 软件，通常你会得到预先编译好的二进制执行文件，而不是其源代码。你可以自行决定测试盒子的颜色（黑盒、白盒还是灰盒）：如果不动用逆向工程手段，可以用黑盒测试方法对软件进行测试；否则，可以采用下一章中讨论的灰盒测试方法对其进行测试。

1.2.1 手工测试

仍然以 Web 应用为例，安全方面的手工测试使用标准 Web 浏览器来探索 Web 站点，在探索 Web 应用结构的同时，向感兴趣的区域中插入“危险”的输入。这种工作通常非常乏味。在安全审计的早期阶段，这种技术偶尔会被采用。例如，通过为一些参数添加单引号以期望发现 SQL 注入的安全漏洞。

除非你的组织雇了一大票实习生，否则，在缺乏自动化工具帮助的情况下通过手工测试应用来找到安全漏洞是不现实的。但手工测试适合于扫除 (Sweeping) 多个应用中类似的安全漏洞。何谓扫除？扫除利用了程序员会在不同应用中犯相同错误这样一个基本事实。举例来说，如果在某个特定的 Lightweight Directory Access Protocol (LDAP) 服

务器中发现了缓冲区溢出，对其他 LDAP 服务器进行类似的测试可能会揭示同样的缓冲区溢出问题。考虑到程序员之间经常共享代码，以及同一个程序员会在多个类似项目中工作这个事实，这种情况并不少见。

扫除（Sweeping）

CreateProcess()是微软提供的 Windows 应用编程接口（API）中的一个函数，正如其名称所描述的那样，CreateProcess()函数用于启动一个新进程及其主线程[13]。该函数的原型如下：

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

根据文档中对该 API 函数的描述，如果该函数被调用时 lpApplicationName 参数为 NULL，则该函数启动 lpCommandLine 参数中空格分割的第一个部分。考虑以下的例子。

如果以这种方式调用 CreateProcess()函数：

```
CreateProcess(
    NULL,
    "c:\program files\sub dir\program.exe",
    ...
);
```

在这种情况下，CreateProcess()函数将会依次尝试启动以下这些 lpCommandLine 参数中空格分隔的值：

```
c:\program.exe
c:\program files\sub.exe
```

[13] http://msdn2.microsoft.com/en-us/library/ms682425.aspx

```
c:\program files\sub dir\program.exe
```

这种尝试一直持续到找到可启动的可执行文件，或是直到扫描完整个参数也找不到可启动的执行文件为止。因此，如果在 c:\目录下放置一个 program.exe 文件，以上的函数调用将会执行 c:\下的 program.exe 文件。如果某个应用以上述方式调用了 CreateProcess()函数，该应用就给了攻击者一个机会，让攻击者有机会执行他们指定的文件（c:\program.exe 文件），因此导致了不安全。

2005 年 11 月发布的一份安全咨询报告[14]指出有几个流行的应用程序用这种不安全的方式调用了 CreateProcess() 函数。这些安全漏洞是依靠非常简单但却很成功的扫除技术发现的。如果你想尝试利用扫除技术发现类似的安全问题，只需要简单的将你的 Windows 操作系统中的 notepad.exe 文件（或其他简单的应用）拷贝到 c:\下，重命名其为 program.exe，接下来照常使用你的计算机和其上的应用程序即可。如果 notepad.exe 突然被自动启动了，恭喜你，你很可能发现了一个对 CreateProcess() 函数的不安全的调用。

1.2.2 自动化测试或模糊测试

模糊测试很大程度上是一种强制性技术，它并不优雅，它的目标是简洁和高效。在第 2 章“什么是模糊测试”中，我们将会对模糊测试进行仔细研究。简单地说，模糊测试就是把所有你能想到的所有东西一股脑扔给被测应用，监视应用返回的结果。大部分软件仅接受特定的输入，但软件应该具有足够的健壮性并能够从错误的输入中恢复。考虑图 1.1 中给出的这个简单的 Web 表单。

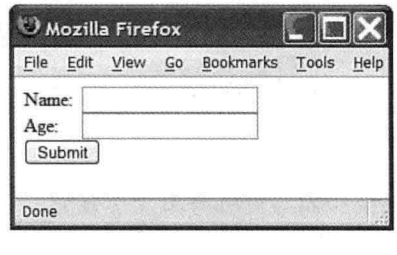


图 1.1 简单的 Web 表单

假设表单的 Name 字段仅能接受字符串值输入，而 Age 字段仅能接受整数输入。

[14] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=340

如果用户在输入时不小心弄反了输入数据，在 Age 字段输入了一个字符串会怎样？应用是会自动地将输入在 Age 域中的字符串转换成 ASCII 表中对应的整数？还是会输出一个错误信息？又或者应用会崩溃？模糊测试通过自动化过程回答这些问题。测试者不需要了解关于应用如何工作的信息，仅采用黑盒的方式就能进行模糊测试。如果我们将被测应用比喻成房子，安全漏洞比喻成房子的窗户，模糊测试就好像你背对一所房子站着，朝身后扔石头，等待听到玻璃破碎的声音。这样说起来，模糊测试似乎应该被归在黑盒测试的类别。然而，在本书中我们还会展示让强制性模糊测试更有效的方法，这些方法能够让你扔出去的石头飞得更直，更准确。这些方法借助了灰盒测试技术，因此模糊测试也可以被归在灰盒测试的类别中。

微软也做模糊测试吗？

答案是“是”。微软公司在 2005 年 3 月发布的可信计算安全开发生命周期（SDL）[15] 文档中提到，模糊测试是微软在软件产品发布前用来发现安全漏洞的关键工具。SDL 倡议在软件开发生命周期中植入安全性考虑，把安全性当成是开发过程中每一位参与者的责任。模糊测试在 SDL 中被描述为一类应该在实现阶段使用的安全性测试工具。实际上，这份文档说到“尽管 SDL 最近才开始对模糊测试进行重点关注，但到目前为止通过模糊测试取得的成效却颇为鼓舞”。

13

1.2.3　优点和缺点

虽然黑盒测试并不总是最佳选择，但在任何情况下它都是一个可行的选择。

黑盒测试的优点包括以下这些。

- **高可用性**：任何情况下，黑盒测试总是可用的。即使在有源代码的情况下，黑盒测试也仍然是有效的测试方法。
- **高可重现性**：黑盒测试不对被测系统进行任何假设。因此，对一个文件传输协议（FTP）服务器进行的黑盒测试可以很容易在其他 FTP 服务器上进行。
- **足够简单**：一些方法，如逆向代码工程（Reverse Code Engineering，RCE）需要专门的技能，而纯粹的黑盒测试由于处于比较基础的层次，因此能够在不需要应用内部知识的情况下进行。当然，黑盒测试并不能解决所有问题。例如，可以通过黑盒的自动化工具简单发现基本的拒绝服务（Denial of Service，DoS）漏

[15] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/sdl.asp

洞，但如果想要知道一次简单的应用崩溃是否会导致一些有趣的问题（例如，非期望的代码执行），就需要对被测系统的内部实现有深入的理解。

14 尽管黑盒测试具有高可用性，然而它也具有某些缺点。以下是黑盒测试方法的缺点。

- **黑盒测试的最大挑战是决定何时停止测试，以及评估测试的效果**。我们将在第23章“模糊测试跟踪器”中对此进行详尽的讨论。
- **不够“聪明”**：黑盒测试非常适于安全漏洞由单组输入引发的情况。然而，复杂的攻击通常包含多组输入，其中一些输入导致被测系统进入诱发安全漏洞的状态，另一些输入则触发安全漏洞。这类攻击的发起者需要对被测系统的逻辑实现非常了解。通常这种类型的安全漏洞只能通过手工的代码评审或是逆向代码工程（RCE）来发现。

1.3 灰盒测试

灰盒测试介于白盒和黑盒测试之间，我们定义的灰盒测试包括黑盒测试，再加上通过逆向工程（Reverse Engineering，RE）获得的系统内部知识（逆向工程有时也称作逆向代码工程，Reverse Code Engineering，RCE）。源代码是珍贵的资源，它易于阅读并提供了关于特定功能是如何实现的细节。此外，源代码还提供了特定功能需要的输入、以及该功能预期产生输出的信息。当然，没有源代码并不意味着无法进行类似的分析。虽然代价更高，对编译后得到的汇编码进行分析也能达到类似的效果。在汇编级别而不是源代码级别进行安全评估通常被称为“二进制审计（binary auditing）”。

1.3.1 二进制审计

逆向代码工程（RCE）与二进制审计（binary auditing）这两个词汇通常被当成同义词使用，但在我们的讨论中，我们把逆向代码工程看做一个方法子类，以使其区别于完全自动化的方法。逆向代码工程的目的是得到编译后的二进制应用程序中的功能实现细节。虽然不可能将一个二进制文件转换回原始的源代码，但可以通过逆向工程将二进制的汇编指令转换成介于原始的源代码和机器码之间的中间形式。通常，这种中间形态包含一些汇编语言代码和应用程序中的流程图。

一旦二进制文件被转换成可以人工读取（human readable）的格式，就可以对其进行代码评审（code review）以找到可能包含有安全漏洞的地方。与对源代码的代码评审 15 一样，找到可能有安全漏洞的地方并不是终点。在找到这些可能的地方后，仍然需要判

断用户的使用是否会影响到这些易受攻击的代码片段。因此，二进制审计技术可以被看做一种“由内及外”的方法：测试者首先通过反汇编深入发现其感兴趣的地方，然后往回追溯以判断这个可能的漏洞是否会被利用。

逆向工程类似于外科手术，该方法借助于反汇编器（disassamblers）、反编译器（decompilers）和调试器（debuggers）对被测应用进行深入剖析。反汇编器将人难以理解的机器码转换成可以人工读取的汇编语言代码。市面上有不少免费的反汇编器可供选择，但对重要的逆向工程而言，你最好是花钱购买一份图 1.2 所示的 DataRescue 公司的 Interactive Disassembler (IDA) Pro 工具。IDA 工具是一个能够运行在 Windows、UNIX 和 MacOS 上的反汇编器，非常适合对多种体系结构二进制代码进行详细解读。

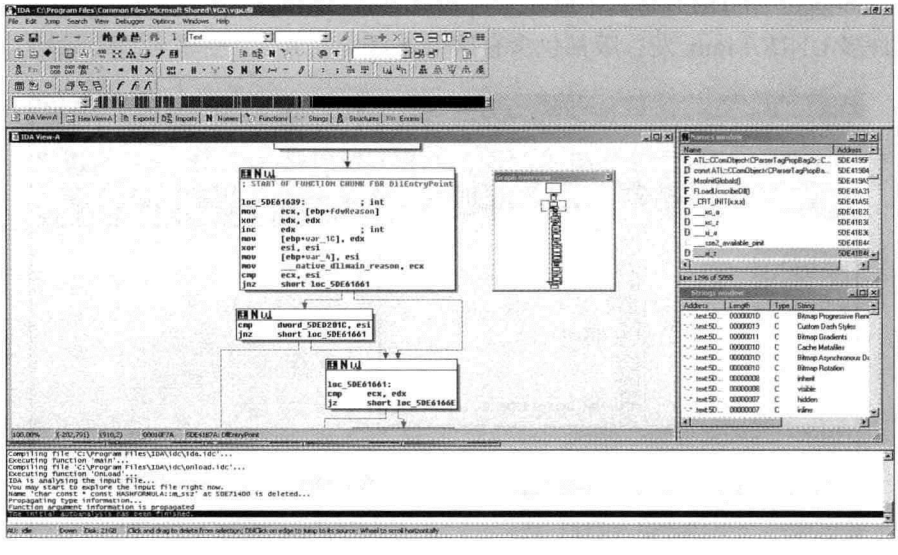


图 1.2　DataRescue IDA Pro 工具

与反汇编器类似，反编译器通过静态分析将二进制文件转换成人可读的格式。与反汇编器不同的是，反编译器并不将二进制文件直接翻译成汇编指令，而是尝试生成高级语言结构，例如分支和循环。尽管如此，反编译器还是不能恢复生成二进制代码的原始源文件，因为原始源文件中的注释、变量名称、函数名称，甚至是一些基础结构信息在编译过程中都不会被保留下来。对源代码会被编译成机器码的编程语言（例如 C 和 C++）来说，反编译器有很大的局限性，通常仅能用于试验目的。Boomerang[16]就是一个这样

16

[16] http://boomerang.sourceforge.net/

的面向机器码的反编译器。对源代码会被编译生成中间字节码的语言，例如 C#，由于编译得到的字节码中包含了更多的信息，因此反编译通常会得到更成功的结果。

与反编译工具和反编译器不同，调试器通过打开或是附加到目标程序上执行动态分析。调试器能够显示 CPU 各寄存器的内容，并能显示应用运行过程中的内存状态。Win32 平台上常用的调试器包括 OllyDbg 工具[17]（图 1.3 展示了该工具的截图）、微软的 WinDbg（读作“wind bag”）工具[18]。WinDbg 工具属于 Windows 调试工具包[19]，并能从微软网站上免费下载。OllyDbg 是一个由 Oleh Yuschuk 开发的共享软件（shareware），其用户友好性比 WinDbg 工具要稍微好一点。这两个调试器都支持用户扩展，而且 OllyDbg 上已经存在大量用于扩展其功能的第三方插件[20]。UNIX 环境下也有各式各样的调试器，GNU Project Debugger（GDB）是其中最受欢迎和最容易移植的一个。GDB 是一个命令行调试器，许多 UNIX/Linux 发行版都包含有这个工具。

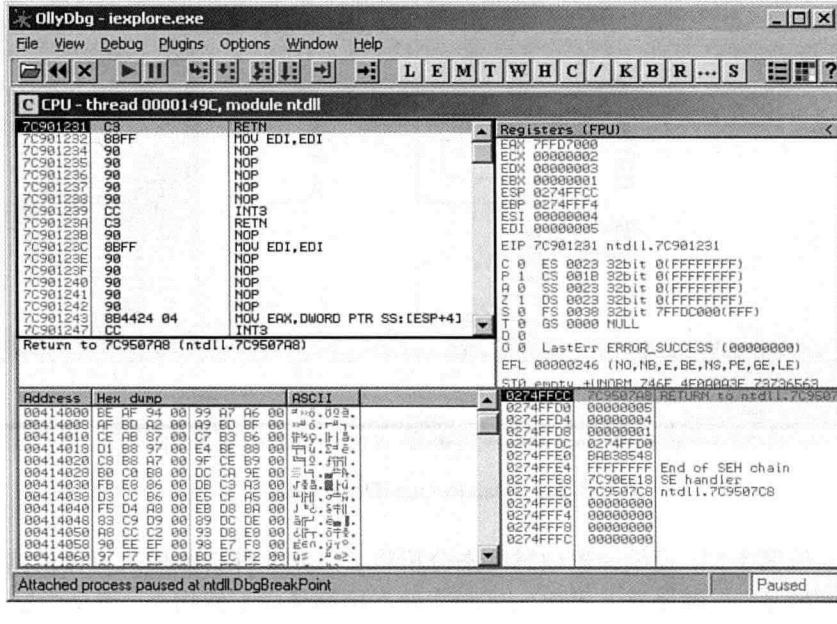


图 1.3 OllyDbg 工具

[17] http://www.ollydbg.de/

[18] http://www.openrce.org/forums/posts/4

[19] http://www.microsoft.com/whdc/devtools/debugging/default.mspx

[20] http://www.openrce.org/downloads/browse/OllyDbg_Plugins

1.3.2　自动化的二进制审计

目前有少量工具试图自动化逆向代码工程过程，从而从应用的二进制执行文件中发现可能的安全漏洞。主要的这类工具中既有商业软件，也有自由软件（Freeware）。有些工具是 IDA Pro 的插件，有些则是独立的应用。表 1.2 列出了该类工具中主要的几种。

表 1.2　自动二进制审计工具

工具名称	工具提供商	授权形式	注　释
LogiScan	LogicLibrary	商业授权	LogicLibrary 于 2004 年 9 月收购了 BugScan 公司后，为其二进制审计工具进行了重新命名，并将其集成进了 Logidex DSA 管理方案中
BugScam	Halvar Flake	自由软件	BugScam 是 IDA Pro 工具的 IDC 脚本集合，它能够枚举二进制文件中的所有函数调用，从而识别出对各种库函数的不安全调用。该应用主要在 BugScan 的基础上使用“诱骗”手段发现安全漏洞
Inspector	HB Gary	商业授权	Inspector 是一个逆向代码工程管理系统，它归一化了来自不同逆向代码工程工具，如 IDA Pro 和 OllyDbg 等的输出
SecurityReview	Veracode	商业授权	Veracode 的产品将二进制分析套件直接集成到开发环境中。该工具在二进制层面上进行分析，能够帮助开发工程师发现一些“所见非所执行的”这类问题
BinAudit	SABRE Security	商业授权	BinAudit 直到本书出版仍未发布。尽管如此，根据 SABRE Security 网站上的介绍，该工具是一个 IDA Pro 的插件，可用来发现诸如数组访问越界、重复释放内存，以及内存泄露等问题

1.3.3　优点和缺点

如前所述，灰盒测试是一种混合方法。该方法通过逆向代码工程获得应用的内部知识，并将这些知识与黑盒测试方法混合在一起。与其他方法相比，灰盒测试有优点，也有缺点。灰盒测试的优点包括。

- **较好的可用性**：除了被测系统是远端的 Web 应用和 Web 服务之外，被测软件的二进制版本几乎总是可以得到的。因此，灰盒测试总是可以派上用场。
- **较高的覆盖率**：从灰盒分析中得到的信息能够用于帮助和提高纯黑盒的模糊测试技术。

灰盒测试的缺点如下。

- **复杂性**：逆向代码工程是一项专门的技能。如果找不到具有这种技能的人，则

无法开展灰盒测试。

1.4 小结

在最高层次上，安全漏洞发现方法可以被分为白盒、黑盒和灰盒测试三种。测试者可用的资源决定了采用何种方法。白盒测试需要包括源代码在内的所有资源；而黑盒测试则只需要测试者能够进行输入和观测输出就能进行；灰盒测试介于黑盒测试和白盒测试之间，通过对二进制文件进行逆向代码工程获得比黑盒分析更多的信息。

白盒测试包括各种不同的源代码分析方法。白盒测试既可以完全依靠手工，也可以通过利用自动化工具，例如编译时检查工具、源代码浏览工具或是自动源代码审核工具来达成。

当源代码不可用时，通常采用黑盒测试方法。仅基于黑盒测试进行模糊测试基本上就是盲测，这种情况下，测试者通过产生特定输入、观测输出来测试应用，但拿不到任何被测系统的内部状态信息以供分析。基于灰盒测试的模糊测试在方法上类似于基于黑盒的模糊测试，但增加了从逆向代码工程中获得的数据以帮助提高模糊测试的效果。简言之，模糊测试采用的方法是：反复向应用程序输入非预期的数据，并在输入的同时监控输出中的异常。本书的剩余部分将会集中在模糊测试这种发现安全漏洞的方法上。

第2章

什么是模糊测试

"They misunderestimated me."

——George W. Bush, Bentonville, Ark., November 6, 2000

在主流字典中压根就找不到模糊测试（fuzzing）这个术语。这个术语有很多别名，而且对某些读者来说可能是一个全新的词汇。模糊测试是一个宽泛的研究领域，在软件安全分析方面，目前模糊测试是一种令人激动的方法。本书将会深入研究不同模糊测试方法的目标和各个方面。但在此之前，我们需要先给这个术语下个定义，探究模糊测试的历史，审视模糊测试的每个阶段，并讨论模糊测试的局限性。

2.1 模糊测试的定义

在字典中很可能根本找不到 fuzzing 这个词，更不用说根据字典的定义来在安全研究领域中定义这个术语了。模糊测试这个术语最早出现在 Wisconsin-Madison 大学的一个研究项目中，此后，该术语被用来描述软件测试的一个方法体系。在学术界，与模糊测试最为接近的术语是边界值分析[1]（Boundary value analysis, BVA）。边界值分析方法是一种黑盒测试方法，该方法根据应用的合法和非法输入区间，选择落在区间边界附近的值作为输入值来进行测试。边界值分析方法能够帮助确保应用的错误处理机制能准确地接受所有合法输入值，并能准确地过滤掉所有非法输入值。模糊测试与边界值分析方

[1] http://en.wikipedia.org/wiki/Boundary_value_analysis

法很类似，不同的是，我们在使用模糊测试方法时，不仅仅只关注边界值，同时还会关心任何能够触发未定义或是不安全行为的输入。

在本书中，我们将模糊测试定义为“通过向应用提供非预期的输入并监控输出中的异常来发现软件中的故障（faults）的方法”。典型而言，模糊测试利用自动化或是半自动化的方法重复地向应用提供输入。显然，上述定义相当宽泛，但这个定义阐明了模糊测试的基本概念。用于模糊测试的模糊测试器（fuzzer）分为两类：一类是基于变异（mutation-based）的模糊测试器，这一类测试器通过对已有的数据样本进行变异来创建测试用例；而另一类是基于生成（generation-based）的模糊测试器，该类测试器为被测系统使用的协议或是文件格式建模，基于模型生成输入并据此创建测试用例。这两种模糊测试器各有其优缺点。在第 3 章“模糊测试方法与模糊测试器类型”中，我们将会更多的谈论不同种类的模糊测试方法，以及这些方法的好处和问题。

如果你是模糊测试的新手，可以将模糊测试类比成如何闯进一所房子。假设你不幸丢了工作，不得不以犯罪为生，现在你想要破门进入一所房子。如果采用纯白盒测试的方法，你需要在破门前得到房子的所有相关信息，包括房子的蓝图（blueprints），房子的锁的生产厂家、房子的建筑材料等。显然，白盒测试放在这种情况下有独特的好处，但也并非万无一失。应用白盒测试方法，你需要对房子进行静态分析而不是进行运行时（实际进入房子时）检查。例如，通过静态分析你发现这所房子起居室的侧面窗户有一个漏洞，可以把窗户打碎进入房子。但你肯定没办法预见到你进入后的事情，也许当你进入以后，发现愤怒的房主正在屋里拿着枪等着你。另外，你可以采用黑盒测试方法来进入这所房子。采用黑盒测试方法，你可以在黑暗的掩护下接近房子，悄悄测试所有的门和窗户，向房子内窥视以决定最好的突破口。但是，如果你最终选择使用模糊测试方法进入这所房子，你可以既不用研究房子的蓝图、也不用手工尝试所有那些锁，只需要选择一种武器并让进入房子的过程完全自动化——这就是强制安全漏洞发现方法的威力！

2.2 模糊测试的历史

有据可查的最早的模糊测试概念可以追溯到 1989 年。1989 年，Barton Miller 教授（Miller 教授被大多数人认为是“模糊测试之父”）在他的高级操作系统课程中使用了一个原始的模糊测试器，用于测试 UNIX 应用的鲁棒性（robustness）[2]。该模糊测试器的目标并不是评估系统的安全性，而是用于评估系统的代码质量和可靠性。虽然该项研究

[2] http://www.cs.wisc.edu/~bart/fuzz/

中也提及了一些安全性的考虑，但在测试中并没有对 setuid 应用的安全性进行特别的关注。1995 年，在一个扩大的 UNIX 工具和操作系统集合上重复了该测试。1995 年的研究表明，应用的整体可靠性得到了提高，但仍然存在“显著的失效率（siginificant rates of failure）”。

Miller 的团队使用的模糊测试方法非常粗糙：如果某个输入导致应用崩溃或是挂起，意味着测试失败，否则就意味着测试成功。Miller 的小组使用的模糊测试器只是纯粹地遵循黑盒测试方法，向被测应用发送随机生成的字符串或是字符。以现在的观点来看，这种方式实在是过于简陋，但别忘了，那时甚至还没有模糊测试的概念。

大约在 1999 年，Oulu 大学开始创建 PROTOS 测试套件。PROTOS 测试套件首先分析协议规范，然后生成违反协议或是看上去不能被具体的协议实现正确处理的数据包。遵循这种方式，许多 PROTOS 测试套件被开发了出来。虽然开发这些测试套件的开销很大，但这些测试套件一旦被创建，就可以针对不同厂家的产品来重复运行。PROTOS 测试套件是一个混合应用白盒测试和黑盒测试的例子，通过这种方式发现了大量软件故障，因此这一事件被当成是模糊测试发展过程中的重要里程碑。

2002 年微软为 PROTOS 提供了资金支持[3]，2003 年 PROTOS 小组的成员成立了 Codenomicon 公司，该公司致力于设计和生产商业的模糊测试套件。直到今天，该公司的产品仍然基于 Oulu 测试套件，当然，还包含了图形用户界面，用户支持，通过“体检”（health-checking）方式发现软件故障的能力，以及其他一些功能[4]。关于 Codenomicon 的更多信息，以及其他的商业模糊测试解决方案将在第 26 章“展望”中进行描述。

随着 PORTOS 在 2002 年逐渐成熟，Dave Aitel 以 GPL 协议方式发布了一个开源的名为 SPIKE 的模糊测试器[5]。该模糊测试器实现了一种基于块（block-based）的方法[6]，用于测试基于网络的应用程序。SPIKE 比 Miller 的模糊测试器更先进，特别是具有描述可变长数据块的能力。除了可以像 Miller 的模糊测试器一样产生随机数据作为输入外，SPIKE 自身还带有一个库，库中包含了一些数据，这些数据有较大的可能让写得不好的应用发生故障。此外，SPIKE 内置了一些预定义的函数，这些函数可以帮助生成常见的

24

[3] http://ww.ee.oulu.fi/research/ouspg/protos/index.html

[4] http://www.codenomicon.com/products/features.shtml

[5] http://immunityinc.com/resources-freesoftware.shtml

[6] http://www.immunityinc.com/downloads/advantages_of_block_based_analysis.html

协议数据及各种格式数据。另外，SPIKE 还包括了对 Sun RPC 和 Microsoft RPC 这两种通信技术的支持，而这两种通信技术在过去一直是导致许多安全漏洞的核心原因。SPIKE 是第一个允许用户以较小代价创建自己的模糊测试器的项目，这一点使它成为了模糊测试发展史上的一个重要里程碑。在第 21 章“模糊测试框架”中我们会多次提到 SPIKE。

在发布 SPIKE 的同时，Aitel 还发布了名为 sharefuzz 的本地 UNIX 模糊测试器。与 Miller 的模糊测试器不同，sharefuzz 通过修改环境变量的值而不是修改命令行参数的值来发现安全漏洞。Sharefuzz 采用了一种有用的技巧来支持模糊测试过程。这种技巧是使用共享库（shared libraries）钩住（hook）那些返回环境变量值的函数，强制这些函数调用返回比实际环境变量值长得多的字符串值，由此发现缓冲区溢出的安全漏洞。

SPIKE 发布之后，模糊测试领域又出现了一些新的创新。这些新的创新均以某种实现了特定模糊方法的工具形式出现。2004 年，Michal Zalewski（又名 lcamtuf）[7] 将模糊测试导入 Web 浏览器测试并发布了 mangleme 工具[8]。Mangleme 工具是一个通用网关界面（Common Gateway Interface, CGI）程序，它持续产生不正常的 HTML 文件，并强制浏览器反复刷新来加载这些 HTML 文件。随后，其他一些面向 Web 浏览器的模糊测试器很快相继面世。H. D. Moore 和 Aviv Raff 开发了 Hamachi 工具[9]，该工具用来对浏览器解析动态 HTML（Dynamic HTML. DHTML）的功能进行模糊测试。后来，这两个人又与 Matt Murphy 和 Thierry Zoller 合作开发了对浏览器解析重叠样式表（cascading style sheet）功能进行模糊测试的工具，工具名为 CSSDIE[10]。

2004 年，文件模糊测试开始兴起。那一年微软发布了 MS04-28 安全公告，该公告详细描述了 JPEG 文件处理引擎中的一个缓冲区溢出漏洞[11]。虽然这不是第一个被发现的文件格式导致的安全漏洞，但由于不少流行的微软应用都使用了这段有安全性问题的代码，这个问题还是引起了广泛关注。文件格式导致的安全漏洞同样为网络保护提出了挑战。随后几年相当数量的有文件格式导致的类似安全漏洞纷纷出现，但简单地将可能存在安全漏洞的文件格式拒于网络之外是并不现实。拿互联网来说，图像和媒体文件的传输占掉了互联网传输的大部分带宽。如果 Web 上不再有图像，Web 还能有多大的吸

25

[7] http://lcamtuf.coredump.cx/

[8] http://lcmtuf.coredump.cx/mangleme/mangle.cgi

[9] http://metasploit.com/users/hdm/tools/hamachi/hamachi.html

[10] http://metasploit.com/users/hdm/tools/see-ess-ess-die/cssdie.html

[11] http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx

引力？另外，这之后不久就有人发现微软的 Office 中也存在由于文件格式导致的安全漏洞——而这些 Office 文件类型对任何企业都非常关键，因此不可能依靠将这些格式的文件拒之门外的方法保证网络安全。在这种情况下，文件格式安全漏洞成为了基于变异的模糊测试方法的主要目标，由于已经有了一些导致安全漏洞的文件格式样本，因此可以一边让快速连续发生变异，一边监控被测应用的输出以找到安全漏洞。本书作者在2005年[12]美国黑帽大会[13]（Black Hat USA Briefing）上做了一个报告，发布了一系列基于变异和基于生成的文件格式模糊测试工具，其中包括 FileFuzz，SPIKEfile 和 notSPIKEfile[14]。

2005 年，一个名为 Mu Security 的公司开始开发一种硬件模糊测试工具，该工具用于让网络中传输的协议数据发生变异[15]，这一工具的出现与日渐高涨的模糊测试潮流相吻合。不断增多的商业模糊测试方案开始出现，与此同时，自由的模糊测试技术变革也随之出现。另外，包含了对模糊测试感兴趣的开发者和安全研究员的社区也逐步扩大，一个明显的证据是社区成员们创建了一个邮件列表[16]，该列表由 Gadi Evron 维护。只有时间可以见证接下来还有多少激动人心的创新在等着我们。

ActiveX 模糊测试在 2006 年开始流行，在 2006 年 David Zimmer 发布了 COMRaider，H. D. Moore 发布了 AxMan[17]。这两个工具都面向可在 Internet Explorer 浏览器中被 Web 应用使用的 ActiveX 控件。由于使用 ActiveX 控件的 Web 应用具有庞大的用户群，因此，如果控件中存在可被远程利用的安全漏洞，这些安全漏洞将会带来巨大的风险。ActiveX 控件本身包含了接口描述、函数原型、以及控件中的成员变量，因此是极好的模糊测试对象，非常适合对其进行智能的自动化测试。ActiveX 控件的模糊测试和浏览器模糊测试作为一个整体，将在第 17 章“Web 浏览器的模糊测试”及第 18 章“Web 浏览器的自动化模糊测试”中进行详细描述。

尽管模糊测试的发展史上还存在其他一些里程碑和工具，但本章到目前为止的介绍已经足够为读者描绘出模糊测试的发展路线图了。图 2.1 展示了模糊测试的发展历史。

---

12 http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-sutton.pdf

13 译者注：黑帽大会是顶级的信息安全领域的会议，每年举办一次，吸引了大量顶尖黑客和安全领域的专家参加。

14 http://labs.idefense.com/software/fuzzing.php

15 http://www.musecurity.com/products/overview.html

16 http://ww.whitestar.linuxbox.org/mailman/listinfo/fuzzing

17 http://metaspliot.com/users/hdm/tools/axman/

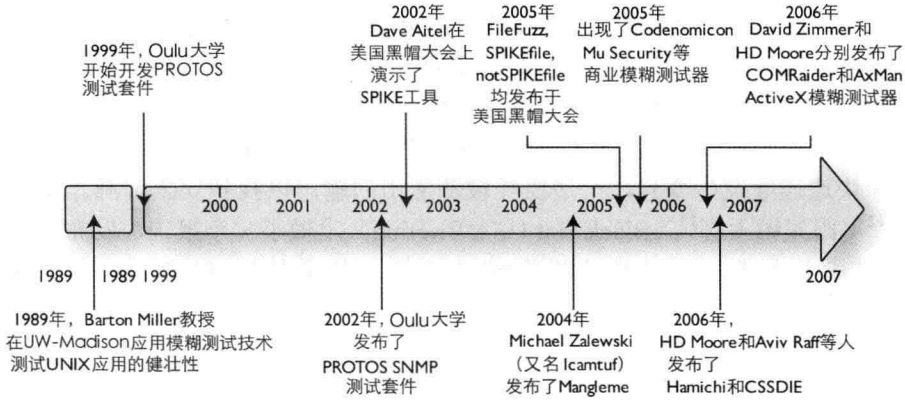


图 2.1　模糊测试的发展历史

尽管模糊测试到目前为止已经取得了一些成果，但模糊测试技术仍然处于婴儿阶段。模糊测试方面的主要工具仍然是较小的项目，由小团队、甚至是一名程序员进行维护。直到最近几年才有创业公司进入该商业领域。这些发展让人们对模糊测试成为一个独立的领域充满了信心。随着模糊测试研究方面投入的增加，模糊测试在未来若干年内将有许多革新出现，并能达到新的里程碑。

在下一节中，我们将会讨论一次完整的模糊测试审计过程中的各个阶段。

2.3　模糊测试各阶段

采用何种模糊测试方法取决于众多因素。没有所谓的一定正确的模糊测试方法，决定采用何种模糊测试方法完全依赖于被测应用、测试者拥有的技能、以及被进行模糊测试的数据的格式。但是，不论对什么应用进行模糊测试，不论采用何种模糊测试方法，模糊测试执行过程都包含相同的几个基本阶段。

1．确定测试目标

只有有了明确的测试目标后，我们才能决定使用的模糊测试工具或方法。如果要在安全审计中对一个完全由内部开发的应用进行模糊测试，测试目标的选择必须小心谨慎。但如果是要在第三方应用中找到安全漏洞，测试目标的选择就更加灵活。要决定第三方应用模糊测试的测试目标，首先需要参考该第三方应用的供应商历史上曾出现过的

安全漏洞。在一些典型的安全漏洞聚合网站如 SecurityFocus[18]和 Secunia[19]上可以查找到主要软件供应商历史上曾出现过的安全漏洞。如果某个供应商的历史记录很差，很可能意味着这个供应商的代码实践 （code practice）能力很差，他们的产品有仍有很大可能存在未被发现的安全漏洞。除应用程序外，应用包含的特定文件或库也可以是测试目标。如果需要选择应用包含的特定文件或者库作为测试目标，你可以把注意力放在多个应用程序之间共享的那些二进制代码上。因为如果这些共享的二进制代码中存在安全漏洞，将会有非常多的用户受到影响，因而风险也更大。

2．确定输入向量

几乎所有可被利用的安全漏洞都是因为应用没有对用户的输入进行校验或是进行必要的非法输入处理。是否能找到所有的输入向量（input vector）是模糊测试能否成功的关键。如果不能准确地找到输入向量，或是不能找到预期的输入值，模糊测试的作用就会受到很大的局限。有些输入向量是显而易见的，有些则不然。寻找输入向量的原则是：从客户端向目标应用发送的任何东西，包括头（headers）、文件名（file name）、环境变量（environment variables），注册表键（registry keys），以及其他信息，都应该被看做是输入向量。所有这些输入向量都可能是潜在的模糊测试变量。

3．生成模糊测试数据

一旦识别出输入向量，就可以依据输入向量产生模糊测试数据了。究竟是使用预先确定的值、使用基于存在的数据通过变异生成的值、还是使用动态生成的值依赖于被测应用及其使用的数据格式。但是，无论选择哪种方式，都应该使用自动化过程来生成数据。

4．执行模糊测试数据

该步骤紧接上一个步骤，正是在这个步骤，“模糊测试”变成了动词。在该步骤中，一般会向被测目标发送数据包、打开文件、或是执行被测应用。同上一个步骤一样，这个步骤必须是自动化的。否则，我们就不算是真正在开展模糊测试。

5．监视异常

一个重要但经常容易被忽略的步骤是对异常和错误进行监控。设想我们在进行一次模糊测试，在测试中，我们向被测的 Web 服务器发送了 10000 个数据包，最终导致了

[18] http://www.securityfocus.com/

[19] http://secunia.com/

服务器崩溃。但服务器崩溃后，我们却怎么也找不到导致服务器崩溃的数据包了。如果这种事真的发生了，我们只能说这个测试毫无价值。模糊测试需要根据被测应用和所决定采用的模糊测试类型来设置各种形式的监视。

6. 判定发现的漏洞是否可能被利用

如果在模糊测试中发现了一个错误，依据审计的目的，可能需要判定这个被发现的错误是否是一个可被利用的安全漏洞。这种判定过程是典型的手工过程，需要操作者具有特定的安全知识。这个步骤不一定要由模糊测试的执行者来进行，也可以交给其他人来进行。

图 2.2 展示了模糊测试的各个阶段。

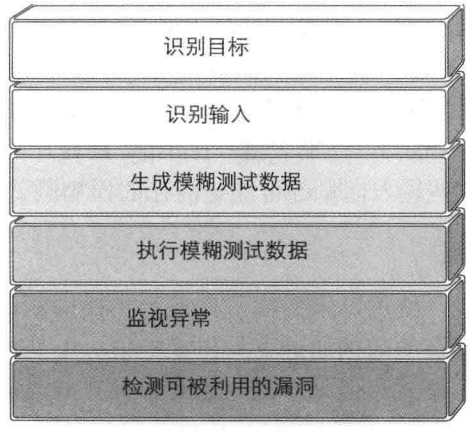


图 2.2　模糊测试阶段

无论采用什么类型的模糊测试，以上这些阶段都应该被考虑到，只有“判定发现的漏洞是否可能被利用”这个阶段可能有例外。根据测试的目的不同，以上各阶段的执行顺序、各阶段的侧重点可以进行调整。虽然模糊测试威力强大，但模糊测试仍然不能保证在被测系统中百分之百地发现所有安全相关的缺陷。下一节中我们会列出一些通过模糊测试很难发现的典型的缺陷类别。

2.4　模糊测试的局限性和期望

模糊测试适合用来找到特定被测系统中的特定弱点。因此，模糊测试能发现的安全

漏洞也是有一定局限性的。在本节中我们将展示一些模糊测试器难以发现的典型安全性漏洞。

2.4.1 访问控制漏洞

有些应用需要分级的权限模型来支持多账户或是不同级别的用户。例如，考虑一个通过 Web 前端访问的在线日程安排系统（Calendaring System）。该应用有一个管理员，管理员可以控制哪些用户能够登录到这个系统。除管理员外，该系统中还有一个特别的用户组，这组用户具有创建日程的权限。除此之外，其他用户只有读权限。在这个系统中，最基本的权限控制是保证一个普通用户无法执行管理员才能执行的任务。

模糊测试器也许能够发现该日程安排软件中的“允许攻击者取得系统的完全控制权”错误。从低级别来看，这类软件错误在不同被测应用中都可能出现，因此，可以用相同的逻辑来检测不同的测试应用中的这类错误。但是，另一类涉及应用本身逻辑的访问权限问题却很难在模糊测试中被发现。例如，在测试过程中，模糊测试器以普通用户的身份成功访问到了只有管理员才能访问到的功能，但是，这个明显的权限问题很可能不会被模糊测试器标示出来。为什么？因为模糊测试器并不理解程序逻辑，它并不知道普通用户不可以访问管理员区域。那么，是否可以在模糊测试中实现这些逻辑功能，告诉模糊测试器哪些功能是只有管理员能够使用的？这种想法听上去很美好，但实际上，这种方式的实现非常麻烦，并且，采用这种实现导致模糊测试器与应用有非常大的耦合，适用于某个应用的模糊测试器如果不经过彻底修改的话，是完全不可能在另一个应用中重用的。

2.4.2 糟糕的设计逻辑

模糊测试器也不是发现糟糕的逻辑设计的最佳工具。例如，考虑在 Veritas Backup Exec 中发现的一个安全漏洞。该安全漏洞允许攻击者远程控制 Windows 服务器并能创建、修改、删除注册表中的键值[20]。这个漏洞导致攻击者几乎可以完全控制系统。这个问题产生的原因是 Windows 服务器中一个基于 TCP 实现远程过程调用（Remote Procedure Call, RPC）的接口，该接口接受修改注册表的指令，并且完全不需要认证。实际上，这不是一个认证问题。因为 Windows 服务器就是这样设计的。这只能说是一

[20] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=269

个不合适的设计决定，该决定可能因为设计者认为攻击者更愿意花费时间来破译Microsoft 接口描述语言（Interface Description Language, IDL），并据此设计一个工具来与服务器交互，而不是花时间去寻找 RPC 服务的漏洞。

虽然模糊测试器可以发现由于被测应用对 RPC 连接传入数据的糟糕处理，导致的某些形式的较低层次错误，但模糊测试器并不适合用来确定这些错误是否导致真正的不安全。在第 18 章“自动化浏览器模糊测试”中讨论 ActiveX 的模糊测试时，我们会再次讨论这个概念，在那一章中，我们会看到许多控件暴露了某些方法，这些方法使得攻击者可以通过它们创建和执行文件。需要对这些设计缺陷给予特殊的考虑。

2.4.3 后门

如果一个模糊测试器对被测应用的结构只有很少的了解，后门看上去与其他的正常逻辑没什么不同。考虑一个应用的登录界面，无论是后门还是正常登录逻辑，都是接受输入并返回认证后的身份信息。不仅如此，除非模糊测试器有足够的信息识别成功的登录，否则，即使这个模糊测试器偶然发现了一个固定的口令，它仍然没有办法识别出这个安全漏洞。在对口令域进行模糊测试时，如果某个输入导致应用发生崩溃，这种漏洞通常可以被发现；但一个通过随机猜测可以被发现的，能进入系统的硬编码的（hard coded）口令就不会被发现。

2.4.4 破坏

内存破坏问题通常会导致被测应用的崩溃。这类问题可以根据与拒绝服务类似的症状来加以识别和发现。然而，有些内存错误问题会被被测应用掩盖，因此简单的模糊测试器永远都发现不了这类问题。考虑以下这个例子，在这个例子中，除非为被测应用附加一个调试器，否则其中的格式字符串漏洞很可能不会被发现。格式字符串安全漏洞通常是由机器代码层的违例代码导致的，例如，在 x86 Linux 机器上，如果格式字符串带有%n 这个典型特征，在执行下面这条指令时通常会发生错误：

```
mov %ecx, (%eax)
```

如果使用模糊测试器向一个充满了带%n 的格式字符串的进程发送随机数据，eax 寄存器在许多情况下将不会包含可写的地址，而是包含有一些从栈中得到的垃圾数据。在这种情况下，该指令会触发一个段违例信号 SIGSEGV。如果应用有 SIGSEGV 的信号处理例程，该处理例程可能会简单地中止进程并重启该进程。但对某些应用来说，它

们的 SIGSEGV 信号处理例程甚至可能在不重启进程的情况下尝试继续执行。对格式字符串安全漏洞来说，这样做是可行的（虽然我们强烈反对这样做），因为在 SIGSEGV 信号被触发前内存有可能实际上没有真正被破坏。那么，如果我们的模糊测试器没有发现这个问题会怎样？进程不会崩溃，而且信号处理例程屏蔽了这个错误，所以，这不是一个安全性问题，对吗？呃，记住，成功的格式字符串渗透会将特定内容写入内存区域中，并在随后通过访问这些已经被控制的内存区域来窃取进程的控制权。SIGSEGV 信号处理例程不能阻止精准的渗透，因为这种渗透根本不会触发任何失效。

根据对测试目标监视的不同程度，你的模糊测试器可能发现这类问题，也可能发现不了这类问题。

2.4.5 多阶段安全漏洞（MuliStage Vulnerability）

32

可利用性（Exploitation）不总意味着像攻击单一弱点那么简单。复杂的攻击通常通过连续利用若干个安全漏洞来获得机器的控制权：起先通过漏洞让机器接受未被授权的访问，然后利用其他漏洞缺陷获得更大的权限。模糊测试对识别单独的漏洞很有用，但通常而言，模糊测试对那些小的漏洞链构成的漏洞，或是让人不感兴趣的事件构成的多向量攻击的识别作用不大。

2.5 小结

模糊测试的发展已经有一段时间了，但在安全研究的圈子之外，模糊测试还不是一项广为人知的技术。因此，你可能会发现不同的人对模糊测试有不同的定义。在本章中，我们给出了我们对模糊测试的定义，简要介绍了模糊测试的历史，并描述了我们对模糊测试的期望。接下来，让我们开始探索模糊测试器的类别和它们使用的方法。

第3章

模糊测试方法与模糊测试器类型

"Too many good docs are getting out of the business. Too many OB/GYNs aren't able to practice their love with women all across the country."

——George W. Bush. Poplar Bluff, Mo., September 6, 2004

模糊测试定义了发现安全漏洞的总体方法。然而，在模糊测试这个大领域内，可以找到实现模糊测试的各种不同方法。从本章开始我们将会探究这些不同的方法。同时，我们将会考察各种类别的模糊测试，这些不同类别的模糊测试应用了不同的模糊测试方法，能够发现特定的安全漏洞。本书的第二部分专门用来深入探究各种模糊测试器的类别。

3.1 模糊测试方法

在上一章中，我们提到所有模糊测试器都可以归到一两个类别中。“基于变异”的模糊测试器通过在已有的数据样本上产生变异来创建测试用例，而“基于生成”的模糊测试器则对被测的协议或文件格式建模，并据此创建测试用例。在本节中，我们将会更详细地讨论这两种模糊测试器的子类。当然，目前还不存在一个业界都认可的模糊测试的分类，但鉴于本书是第一本模糊测试方面的专著，我们可以按照自己的方式将模糊测

试的方法分成五个大类。

3.1.1　预生成测试用例

在上一章中提到过，PROTOS 框架采用了预生成测试用例的方法。该方法要求首先研究特定的规约，理解该规约支持的数据结构和可接受的值的范围；然后依据这些理解生成用于测试边界条件或是违反规约的测试用例；接下来使用这些测试用例来测试该规约实现的完备性。创建所有这些测试用例需要花费很多精力，但这些用例一旦被创建，就很容易被复用，用于测试某种协议或文件格式的不同实现。

预生成测试用例的缺点是，这种模糊测试方法存在固有的局限性。由于缺乏随机生成，一旦测试用例列表中的用例被执行完，测试就只能结束。

3.1.2　随机生成输入

随机方法是最低效的方式，但是这种方式可以被用来快速地识别目标应用中是否有非常糟糕的代码。随机方法简单地向目标应用发送伪随机数据，希望得到最好或是最坏的结果（最好还是最坏取决于你看问题的角度）。使用随机方法的最简单的例子如下：

```
while[1]; do cat /dev/urandom | nc -vv target port; done
```

这个一行的命令从 Linux urandom 设备读取一个随机数，然后将该随机数发送到给定的目标地址和端口。While 循环确保该过程能一直持续，直到用户中断为止。

不管你信不信，通过这种方式还真发现了一些核心软件中的安全漏洞。真是不可思议！采用随机方式最困难的部分是如何找回导致服务器发生崩溃的数据。例如，从 500000 个随机字节中找出真正导致系统崩溃的原因会是一件痛苦的事情。你可能需要一个网络嗅探器（sniffer）记录下所有向服务器发送的随机数据，这样可以让寻找的范围变窄一点。当然，即使有网络嗅探器的帮助，你仍然需要通过调试器和反汇编器并花费大量的时间才能确定导致系统崩溃的原因。使用这种技术需要分析栈，在调用栈有可能被破坏的情况下，调试过程尤其痛苦。图 3.1 是附加在某个企业级系统和网络监控工具上的调试器的截图，该企业级系统和网络监控工具刚在一个随机模糊测试器的攻击下发生了崩溃。从图中你能找出失效的原因吗？除非你有未卜先知的能力，否则一定是不可能的。要找出这个问题的原因，需要做许多的研究工作。注意，这其中的有些数据完全被应用的保护机制给弄乱了。

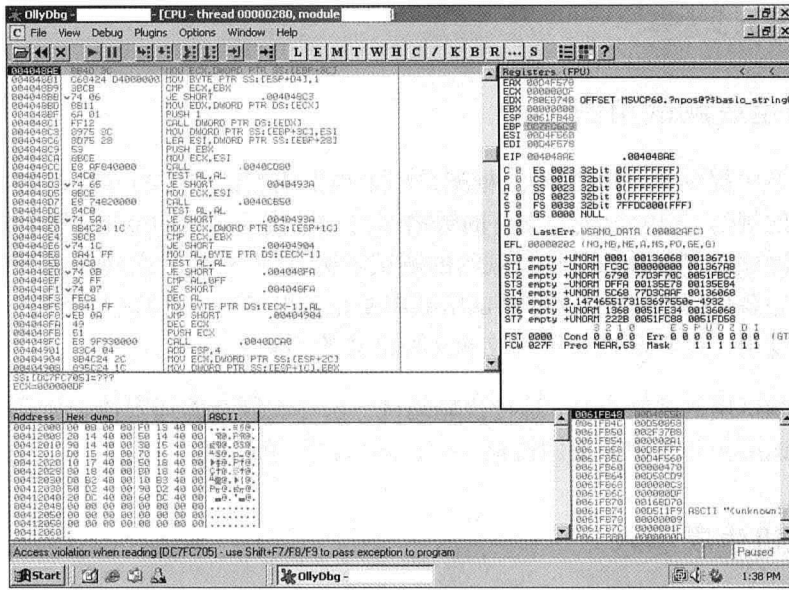


图 3.1　现在怎么办？/dev/urandom 模糊测试器进退两难

3.1.3　手工协议变异测试

按说，手工协议测试应该比随机生成输入的方法在技术方面更加初级。手工协议测试不需要自动化模糊测试器。实际上，在手工测试中，测试者就是模糊测试器。在加载了目标应用后，测试者仅仅通过输入不正确的数据，试图使服务器崩溃，或是诱发一些不正常的行为。这是“穷人”的模糊测试方法，但在过去一些年来这种方法已经被证明是有效的。这种方法的优点在于，分析者能够在安全审计中充分发挥自己过去的经验和“直觉”。这类模糊测试方法最经常用于 Web 应用的安全测试。

3.1.4　变异或强制性测试

我们这里所说的强制性测试，是指模糊测试器从一个有效的协议样本或是数据格式样本开始，持续不断的打乱数据包或是文件中的每一个字节（byte）、字（word）、双字（dword）或是字符串（string）。这是一种有效的早期模糊测试方法，因为这种方式几乎不要求对应用进行研究，而且，实现一个基础的强制性模糊测试器也相对直接。基本上，强制性模糊测试器只需要修改数据并将其发送给被测应用。当然，除了发送数据外，也可以在强制性模糊测试器中加入更多的错误检测手段、日志手段等。这样一个工具通常

能够在短时间内被创建出来。

这种方法或多或少比较低效，因为许多 CPU 周期会被浪费在生成完全不可解析的数据上。然而，以下的好处或多或少能够抵消这个不足：整个强制性模糊测试的过程可以被完全自动化。使用强制模糊测试方法达到代码覆盖依赖于已知的“合法数据包”，或是“合法文件”。大多数协议规约或是文件定义都是相对复杂的，因此即使要达到较低的覆盖率，也需要大量的样本。强制性模糊测试文件格式的模糊测试器样例包括 FileFuzz 和 notSPIKEfile 工具，分别对应 Windows 平台和 Linux 平台。

3.1.5　自动协议生成测试

自动协议生成测试是一种更高级的强制性方法。在这种方式中，首先要做的是对被测应用进行研究，理解和解释协议规约或文件定义。然而，这种方法并不基于协议规约或文件定义创建硬编码的测试用例，而是创建一个描述协议规约如何工作的文法（grammar）。采用这种方式，测试者可以识别出数据包或是文件中的静态部分和动态部分，动态部分就是可以被模糊化变量替代的部分。随后，模糊测试器动态分析包含了静态和动态部分的模板，生成模糊测试数据，将结果数据包或是文件发送给被测应用。这种方法是否能够成功取决于测试者的能力——测试者需要能够指出规约中最容易导致目标应用在解析时发生故障的部分。SPIKE 和 SPIKEfile 工具都是这类测试工具的典型样例，这两种模糊测试工具采用 SPIKE 脚本描述目标协议或文件格式，并使用一个模糊测试引擎来创建输入数据。该方法的不足之处在于它需要花较多时间在生成文法或是数据格式的定义上。

3.2　模糊测试器类型

我们已经了解了模糊测试的不同方法和手段，接下来让我们看一些具体的模糊测试器类型。模糊测试器的类型基于被测目标。不同的目标需要不同类型的模糊测试器。

在本节中，我们将会简要考察一些不同的模糊测试器类型。在本书的剩余部分，我们会详细介绍每一类模糊测试器，包括其背景资料、具体的自动化和开发样例。

3.2.1　本地模糊器

在 UNIX 世界中，setuid 应用允许普通用户临时获得更高权限。setuid 应用的任何

安全漏洞将会允许用户持久提升权限并执行其选择的任何代码，因此，setuid 应用显然是需要进行模糊测试的目标。对 setuid 应用的模糊测试可以从两个方向来进行。第一种方式是命令行参数模糊测试，该方法从命令行传递半形式化的参数给 setuid 应用。第二种方法同样是向 setuid 应用传递半形式化的参数，但却不是以命令行方式。在第二种情况下，半形式化参数通过 UNIX Shell 环境传入。下面让我们来看看命令行和环境变量模糊测试这两种不同的方式。

命令行模糊测试器

当一个应用开始运行时，通常需要处理用户传给它的命令行参数。下面这个例子在第 1 章“安全漏洞发现方法学”中也用过，我们用这个例子展示最简单的命令行参数栈溢出的形式。

```
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[10];
  strcpy(buffer, argv[1]);
}
```

如果这个程序就是 setuid 应用的话，可以用本地模糊测试器对其进行测试。问题是，在源代码不可用的情况下，安全测试者如何能够找到 setuid 应用中找到更多不那么显而易见的缺陷？如果我告诉你最简单的方法就是使用模糊测试，相信你应该不会感到惊讶。

下面是两个有用的命令行模糊测试器工具。

- **warl0ck 的 clfuzz 工具**[1]：该工具是一个命令行模糊测试工具，用于测试应用中的格式字符串和缓冲区溢出的安全漏洞。
- **Adam Greene 的 iFUZZ 工具**[2]：该工具是一个命令行模糊测试工具，用于发现应用中的格式字符串和缓冲区溢出安全漏洞。该工具包括模糊测试几个不同参数的选项，并且能够根据应用的帮助信息（Usage 信息）对应用进行模糊测试。

环境变量模糊测试器

另一类本地模糊测试包括了环境变量维度，仍然假设我们的测试对象是 setuid 应

[1] http://warl0ck.metaeye.org/cb/clfuzz.tar.gz

[2] http://fuzzing.org

用。考虑下面的程序片段，该程序对用户环境中的变量值的使用是不安全的。

```
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[10];
  strcpy(buffer, getenv("Home"));
}
```

有好几种有效的方式来对应用中环境变量的使用进行模糊测试，但尽管这个过程很简单，却没有多少自动化工具存在。许多安全研究员用他们自己拼凑的（hacked-together）脚本来完成这个任务，这是为什么很少有公开发布的这类工具的一个原因。大多数情况下，本地环境变量模糊测试器的开发并不特别复杂，不值得作为一个公共软件发布，这是很少有公开发布的这类工具的另一个原因。下面这些都是环境变量模糊测试的有用的工具。

- **Dave Aitel 的 Sharefuzz 工具**[3]：该工具是第一个公开发布的环境变量模糊测试工具。它拦截对 getenv 函数的调用并返回恶意数据。
- **Adam Greene 的 iFUZZ 工具**[4]：虽然该工具主要是一个命令行模糊测试器，但 iFUZZ 软件包也包含了基本的环境变量模糊测试能力。在环境变量模糊测试方面，iFUZZ 使用与 ShareFUZZ 相同的方法，但是更易于在使用中被定制。

在第 7 章“环境变量与参数模糊测试”和第 8 章“自动化的环境变量与参数模糊测试”中，我们将会更多地讨论本地模糊测试和 setuid 的概念。

39

文件格式模糊测试器

许多应用都需要在某个点上进行文件输入和输出，客户端应用和服务端应用都是如此。例如，反病毒网关程序通常需要分析压缩文件，诊断文件中包含的内容。而办公软件则经常需要打开文档。这两类应用在处理恶意编造的文件时，都可能会被触发其中的安全漏洞。

这些就是文件格式模糊测试的用武之地。文件格式模糊测试器动态创建半形式化文件，然后使用目标应用处理这些文件。虽然文件模糊测试中使用的模糊测试方法与其他类型的模糊测试不完全相同，但其基本思路是一致的。下面列出了可用的文件格式模糊

---

[3] http://www.immunitysec.com/resources-freesoftware.shtml

[4] http://fuzzing.org

测试工具。

- Michael Sutton 的 FileFuzz 工具[5]：该工具是一个 Windows GUI 的文件格式模糊测试工具。
- Adam Greene 的 notSPIKEfile 和 SPIKEfile 工具[6]：这两个工具是基于 UNIX 的文件模糊测试工具，前者不基于 SPIKE 工具，后者基于 SPIKE 工具。
- Cody Pierce 的 PAIMEIfilefuzz 工具[7]：与 FileFuzz 工具类似，该工具也是一个 Windows GUI 的文件格式模糊测试工具。该工具基于 PaiMei 逆向工程框架。我们将在后面的章节中详细讨论 PaiMei 框架。

我们将在第 11 章“文件格式模糊测试”、第 12 章“UNIX 平台上的文件格式自动化模糊测试”和第 13 章“Windows 平台上的文件格式自动化模糊测试”中深入研究文件格式模糊测试。

3.2.2 远程模糊测试器

远程模糊测试器的测试目标是在网络上打开端口监听的应用。网络应用可能是远程模糊测试的主要测试目标。随着因特网的发展，几乎所有企业如今都有了可被公共访问的服务器，企业使用这些服务器来发布网页、E-mail、DNS 等服务。任何一个上述服务系统中的安全漏洞都使得攻击者能够访问敏感数据，或是为攻击者提供一个对信任服务器进一步攻击的跳板。

网络协议模糊测试器

网络协议模糊测试器可以被分为两个主要类别：面向简单协议的模糊测试器和面向复杂协议的模糊测试器。接下来我们分别介绍这两种模糊测试器的特征。

简单协议

简单协议通常仅有简单的认证或是根本就没有认证。这类协议通常基于可打印的 ASCII 字符而不是二进制数据。简单协议不会包含长度或是校验域。此外，典型的使用简单协议的应用不会有很多状态。

---

[5] http://fuzzing.org

[6] http://fuzzing.org

[7] https://www.openrce.org/downloads/detail/208

FTP 协议是一个简单协议的例子。在 FTP 协议中，所有的控制信道上的通信都使用普通的 ASCII 文本。对认证来说，只需要简单文本形式的用户名和口令就可以完成认证。

复杂协议

复杂协议通常由二进制数据和偶尔包含的人可读的 ASCII 字符串构成。认证可能需要通过加密或是某种形式的混淆来实现，此外，复杂协议还包括多个复杂的状态。

微软远程过程调用（Microsoft Remote Procedure Call，MSRPC）协议是一个复杂协议的好例子。MSRPC 是一种二进制协议，在数据传输之前需要经过好几个步骤才能建立通信信道。协议需要长度描述域和分解域。总体上来说，MSRPC 协议可不是一个容易对其进行模糊测试的协议。以下是网络协议模糊测试方面有用的工具。

- **Dave Aitel 的 SPIKE 工具**[8]：SPIKE 工具是第一个公开发布的模糊测试框架。该工具包含预生成的、针对几种常用协议的模糊测试脚本，并且该工具能被以 API 方式使用。
- **Michael Eddington 的 Peach 工具**[9]：该工具是一个跨平台的模糊测试框架，使用 Python 语言开发而成。该工具非常灵活，几乎可以用来对任何需要进行模糊测试的网络应用进行测试。

在本书的第 14 章“网络协议的模糊测试”、第 15 章“UNIX 平台上的自动化网络协议模糊测试”以及第 16 章“Windows 平台上网络协议的模糊测试”中，我们将会更深入地讨论网络协议模糊测试。

Web 应用模糊测试器

Web 应用已经成为访问 E-mail、账单支付等后端服务的一种流行的便捷方式了。借助 Web 2.0（无论 Web 2.0 究竟是什么）的优势，传统的桌面应用，诸如字处理软件等都逐渐转移到了 Web 上[10]。

在对 Web 应用进行模糊测试时，测试者主要寻找 Web 应用特有的安全漏洞，如 SQL 注入、跨站脚本攻击（XSS）等。这就要求 Web 应用模糊测试器具备通过超文本传输协议（Hyper Text Transfer Protocol，HTTP）通信的能力，以及具有捕获响应以便进行

[8] Http://www.immunitysec.com/resources-freesoftware.shtml

[9] http://peachfuzz.sourceforge.net/

[10] http://www.google.com/a/

后续分析，找到安全漏洞的能力。下面这些工具可以用在 Web 应用模糊测试方面。

- **OWASP 的 WebScarab 工具**[11]：该工具是一个具有模糊测试能力的 Web 应用审计套件。
- **SPI Dynamics 的 SPI Fuzzer 工具**[12]：该工具是一个商业的 HTTP 和 Web 应用模糊测试工具，包含在 WebInspect 安全漏洞扫描器中。
- **Codenomicon 的 Codenomicon HTTP 测试工具**[13]：该工具是一个商业的 HTTP 测试套件。

我们将在第 9 章“Web 应用与服务器模糊测试”和第 10 章“Web 应用与服务器的自动化模糊测试”中对 Web 应用模糊测试中进行进一步的探索。

Web 浏览器模糊测试器

从技术上说，Web 浏览器模糊测试器只是文件格式模糊测试器中特殊的一类，但由于基于 Web 的应用为数众多，我们宁愿把 Web 浏览器模糊测试器当成一个专门的类别。Web 浏览器模糊测试器通常利用 HTML 的功能来自动化模糊测试过程。以 lcamtug 的 mangleme 工具（该工具是最早公开发布的浏览器模糊测试工具之一）为例，该工具利用了<META REFRESH>标签自动持续加载测试用例。这个独特的 Web 浏览器功能使得可以进行完全自动化的客户端模糊测试，而无需对应用进行任何包装。对于其他客户端模糊测试器来说，这种包装是不可或缺的。

Web 浏览器模糊测试并非仅限于 HTML 解析，除 HTML 解析外，还有许多适合进行模糊测试的目标。例如，模糊测试工具 See-Ess-Ess-Die 面向重叠样式表（Cascading Style Sheet，CSS）解析，COM Raider 工具面向加载到 Microsoft Internet Explorer 的组件对象模型（Component Object Model，COM）对象。其他可被模糊测试的元素包括 HTML 页面中包含的图形和服务器返回的响应头。以下这些工具可以用在 Web 浏览器模糊测试方面。

- **Lcamtuf 的 mangleme 工具**[14]：这是第一个公开发布的 HTML 模糊测试器。该工具是一个 CGI 脚本，重复地发送打乱了的 HTML 数据给浏览器。

---

11 http://www.owasp.org/index.php/Fuzzing_with_WebScarab

12 http://www.spidynamics.com/products/webinspect/index.html

13 http://www.codenomicon.com/products/internet/http/

14 http://freshmeat.net/projects/mangleme/

- H.D. Moore 和 Aviv Raff 的 DOM-Hanoi 工具[15]：该工具是一个 DHTML 模糊测试器。
- H.D. Moore 和 Aviv Raff 的 Hamachi 工具[16]：该工具是另一个 DHTML 模糊测试器。
- H.D. Moore，Aviv Raff，Matt Murphy 和 Thierry Zoller 的 CSSDIE 工具[17]：该工具是一个 CSS 模糊测试器。
- David Zimmer 的 COM Raider 工具[18]：该工具是一个易于被使用，GUI 驱动的 COM 对象（ActiveX 控件）模糊测试器。

在第 17 章“Web 浏览器的模糊测试”和第 18 章“Web 浏览器的自动化模糊测试”，我们将更深入地讨论 COM 和 ActiveX 对象模糊测试。在对 Web 浏览器模糊测试进行讨论时，我们并没有将 CSS 和图形文件模糊测试单独列出，但文件格式模糊测试中的概念在 CSS 和图形文件模糊测试中同样适用。

3.2.3　内存模糊测试器

有时在测试中的某些特定障碍阻碍了模糊测试快速和高效的进行。在这种情况下，可以考虑使用内存模糊测试。内存模糊测试的概念很简单，但是实现一个适用的内存模糊测试器并不容易。内存模糊测试的一种实现方法是对进程执行一次快照，在生成快照后迅速向该进程的输入处理子例程中注入故障数据。当执行完一个测试用例后，恢复上次的快照并注入新的数据。重复以上过程直到所有测试用例都执行完成。和其他所有模糊测试方法一样，内存模糊测试也有其优点和缺点。

内存模糊测试有下列优点。

- **快速**：这种方法不仅不需要网络带宽，甚至，在测试过程中都不需要执行从网络上接收数据包到实际解析数据包之间的代码。由此提升了测试性能。
- **捷径**：如果一个协议使用了定制的加密或是压缩算法，或是程序内到处都是数据校验代码，在这种情况下，内存模糊测试器能够在解压，解密或是数据校验完成后的某个时间点创建一个快照，这样就避免了花费时间创建一个能够处理

[15] http://metasploit.com/users/hdm/tools/domhanoi/domhanoi.html

[16] http://metasploit.com/users/hdm/tools/hamachi/hamachi.html

[17] http://metasploit.com/users/hdm/tools/see-ess-ess-die/cssdie.html

[18] http://labs.idefense.com/software/fuzzing.php#more_comraider

43

所有这些加密，压缩或是数据校验的模糊测试器，能够大大减少工作量。

内存模糊测试的缺点如下。

- **假相**：内存模糊测试是直接将数据注入进程的地址空间，因此有可能被注入的数据根本不可能通过从外部源输入的数据来产生。
- **可重现性**：虽然一个异常可能预示着一个可被利用的漏洞，但即通过内存模糊测试发现了这样的异常，测试者仍然需要从外部输入找到重现这个异常的外部数据，而找到对应的外部数据可能非常耗时。
- **复杂性**：我们将在第 19 章“内存的模糊测试”和第 20 章“内存的自动化模糊测试”中看到，内存模糊测试方法的实现非常复杂。

3.2.4 模糊测试框架

模糊测试框架可用于对众多目标进行模糊测试。简单的说，模糊测试框架就是一个通用的模糊测试器或是通用的模糊测试库，它简化了多种不同类型的测试目标需要的数据格式。我们在本章中提到的某些模糊测试器实际上就是模糊测试框架，如 SPIKE 和 Peach 工具。

典型而言，模糊测试框架包括一个库用来生成模糊测试字符串，或是通常能够导致解析子例程出现问题的模糊值。此外，典型的模糊测试框架由一套简化的网络和磁盘输入输出的子例程构成。模糊测试框架还应该包括一些类似脚本的语言，以便用来创建特定模糊测试器。模糊测试框架很流行，但框架绝不是在任何情况下都适用。模糊测试框架的使用和设计同样存在优点和缺点。优点如下所述。

- **可重用性**：如果能够创建一个真正通用的模糊测试框架，在模糊测试不同目标时，该框架可以被反复重用。
- **社区介入**（Community Involvement）：让社区介入一个大的，可以在多种场合下发挥作用的项目开发是很容易的，反之，要想找到一大群开发者来开发一个仅有非常窄的应用范围的小项目则不容易，而后者就是特定协议的模糊测试器的开发状况。

模糊测试框架存在下列这些缺点。

- **局限性**：即使是使用一个经过仔细设计的模糊测试框架，测试者也很可能会遇到不适合使用该框架的测试目标。这可能会让人非常沮丧。

44

- **复杂性**：有一个通用的模糊测试界面是好事，但学会如何驾驭该框架也需要时间。

有时候，直接从头开始创建一个针对特定测试目标的模糊测试工具比基于框架创建模糊测试工具花费的时间更少。在这种情况下，框架就没有什么作用了。

- **开发时间**：相比设计一个一次性的，针对特定协议的模糊测试工具而言，设计一个可以完全通用，可以被多次复用的东西工作量要大得多。大工作量往往意味着更长的开发时间。

有些测试者强烈认为框架是最好的方法，而另一些则坚持认为不是这样，他们认为每一个模糊测试器都应该针对特定的测试目标，一次性的创建。只能依靠读者自行实践这两种思路，并做出适合自己的决定。

3.3 小结

当进行模糊测试时，应该考虑到本章中讨论的每种方法的不同价值。通常混合使用多种方法能够得到最佳的结果。因为一种方式可能是另一种方法的补充。

模糊测试器能够关注多种不同的目标，因此，也就存在有各种不同的模糊测试器。当你开始消化不同模糊测试类型背后的基础概念时，应该设想一下如何去实现它们。这种思考将有利于你对本书后面的内容给出自己的评价。

第4章

数据表示和分析

"My job is to, like, think beyond the immediate."

——George W. Bush, Washington, DC, April 21, 2004

协议被用于计算机外部和内部通信的各个方面。协议形成了数据传输和处理的基础结构。如果我们希望进行成功的模糊测试，就必须先对模糊测试的目标应用使用的协议有所理解。了解了被测应用使用的协议后，我们可以找出该协议中最可能引发异常的部分。此外，要访问一个协议中的安全漏洞，我们也需要在开始进行模糊测试前提供合法的协议数据，这也要求我们对协议有一定的了解。对开放和专有协议的理解将会使得安全漏洞的发现更有效率。

4.1 什么是协议

协议是达成通信的必要条件。有些时候，我们所说的协议被定义成了标准，而其他时候，我们所说的协议只是被广泛接受的事实标准。交谈的方式就是一个事实标准。虽然没有正式的文档约定交谈的方式，但通常情况下，当一个人对另一个说话时，听的那个人会保持安静并准备响应说话的人。类似的，你不能仅仅走到一个人面前，就开始报出你的名字、地址和其他重要信息（在这个年代你可别这么做，当心你的身份信息被人偷走），还期望对方能够理解。个人信息需要以元数据（metadata）进行表达。例如，在传递“我的名字是梅有仁。我住在乌有街345号。”这条信息时，只有发送方和接收方对数据标准和规则达成共识，才能成功地完成通信。在计算机领域，协议非常关键。

计算机既没有智能，也没有人类的直觉，因此，严格定义的协议对数据通信和数据处理是必不可少的。

维基百科（Wikipedia）将协议定义为“在两台计算机端点之间控制或使能（enable）连接，通信和数据传输的约定或标准”[1]。这里所说的端点可以是两台独立计算机，也可以是位于同一台计算机内的两个端点（例如，一台计算机上运行的两个应用）。以读入数据为例，当计算机需要读入数据时，它首先从硬盘上读取数据，然后通过数据总线将数据传输到内存中，接着再将其移动到处理器中。在每个处理步骤内，数据一定是以特定数据格式表现的，只有这样数据的发送方和接受方才可能正确处理这些数据。本质上，数据只不过是一堆 0 和 1 的集合而已，这个集合只有在特定上下文下才有意义。如果某个处理步骤的两个端点对数据的理解不一致，那么他们之间传输的数据对双方来说都没有意义。

计算机之间的通信也依赖于协议。因特网协议（Internet）是一个广为人知的计算机间的通信协议。因特网由多种不同层次的协议组成。一台美国的家用台式机能够将数据封装在多层协议中，通过因特网传送给位于中国的另一台台式机。接收到数据的台式机用同样的协议就能解出和解释接收到的数据。当然，遵循同样的协议，中国政府也可以拦截这些传输数据，并解出和解释这些数据。当参与通信的两个端点都能够理解彼此间通信的协议时，双方就可以进行通信了。当然，如果两个端点间存在路由器的话，参与传输的路由器也必须理解协议中的某些内容，这样才能保证数据能够被正确路由。

虽然协议的作用都差不多，但不同的协议有各种不同的具体表现形式。有些协议被设计成便于人工阅读的，简单的文本格式，另一些则以难以被人直接理解的二进制形式表现。GIF 图像文件格式和 Microsoft Excel 电子表格的文件格式采用的都是二进制协议。

协议中包含的各部分通常被称为字段。协议规范定义了字段之间的分隔符和字段的顺序。下面我们来详细考察协议中的字段。

4.2　协议中的字段

设计一个协议时需要做许多决定。但其中最重要的决定之一是如何将协议中的数据划分为不同的部分。数据发送方和接收方都需要知道如何解释协议中的数据，而协议的

[1] http://en.wikipedia.org/wiki/Protocol_%28computing%29

主要作用就是定义“协议中各部分数据如何被解释”。设计协议时，有三种典型的方法：**定长字段**，**变长字段**和**分隔字段**。例如，简单文本协议（Simple Plain Text Protocol）通常以回车作为分隔符。客户端或是服务端在接收简单文本协议数据包时，如果在数据中遇到一个回车，它就知道这标识着上一个命令的结束。文件格式中也经常会使用字符来对字段进行分隔，例如逗号分隔（comma-separated value，CSV）文件就是这样。逗号分隔文件可以用来表示二维数组，并能被 Microsoft Excel 应用读取。XML 是另一个使用字符对字段进行分隔的例子，但与逗号分隔文件不同，XML 不使用单个字符作为字段结束的指示符，而是使用多个字符对字段进行分隔。例如，XML 文件中的元素都定义在两个尖括号（ < 和 > ）之间，而元素属性的名称和值使用等号（=）进行分隔。

定长字段将每个字段的长度设为一个固定值。网络协议，如以太网协议（Ethernet），Internet 协议（Internet Protocol，IP），传输控制协议（Transmission Control Protocol，TCP）和用户数据报协议（User Datagram Protocol，UDP）等都在协议头中使用定长字段。网络协议在头中使用定长字段的原因是，协议头需要高度结构化的数据。例如，一个以太网数据包的前 6 个字节是目的地址的媒体访问控制（Media Access Control，MAC），随后的 6 个字节定义了源地址的 MAC 地址。

为优化目的选择字段

能够规定某些字段的大小和位置的另一个优点是，规约的作者可以灵活地通过使用字节对齐来帮助实现处理器优化。例如，IPv4 头中许多字段按照 32 位的边界进行对齐。Intel 优化手册的第 3.41 节中提到，在奔腾处理器上，读取非对齐数据相当耗费资源。

在奔腾系列处理器上，一次对非对齐数据的访问操作需要消耗 3 个时钟周期。在奔腾 Pro 和奔腾 II 处理器上，跨越缓存边界的一次对非对齐数据的访问需要消耗 6～9 个时钟周期。数据缓存单元（Data Cache Unit，DCU）分裂（split）是指一次跨越了 32 字节边界的内存访问，对非对齐数据的访问会导致发生数据缓存单元分裂，因此会导致奔腾 Pro 和奔腾 II 性能降低。为了得到最佳性能，在这些处理器上应该确认所有大于 32 字节的数据结构和数组都是以 32 字节进行对齐的，而且对数据结构和数据元素的访问不能违背对齐原则[2]。

另外，对有些使用精简指令集计算（Reduced Instruction Set Computing，RISC）架构，如 SPARC，的处理器，在其上执行非对齐内存访问会导致失败，并抛出致命的总

[2] http://download.intel.com/design/intarch/manuals/24281601.pdf

线错误信号。

在数据并非完全结构化的情况下，更适合使用可变长字段。媒体文件通常使用可变长字段。图形文件和视频文件的格式都非常复杂，其中通常包含许多可选的头和数据组件。可变长字段为协议的开发者提供了更大的灵活性，使得他们能够创建有效使用内存的协议。可变长字段不为每个数据元素设定固定的长度，每个字段都是可选的，典型的字段由指示字段类别的标识符和指示字段长度的值作为前缀，然后是该字段的其他数据。

4.3 简单文本协议（Plain Text Protocols）

简单文本协议（Plain Text Protocols）这个术语指协议中使用的用于通信的数据值都在可打印字符范围内。可打印字符包括数字，大写和小写的字母，各种符号（例如百分号和美元符号），回车符（\r，十六进制值为 0x0D），换行符（\n，十六进制值为 0x0A），制表符（\t，十六进制值为 0x09）和空格（十六进制值为 0x20）。

简单文本协议的设计使得它可以被人读懂。和二进制协议相比，简单文本协议通常效率比较低，二进制协议能够紧凑地使用内存，而简单文本协议则需要消耗更多内存。但有些情况下，使用人工可阅读的协议会更加合适。例如，文件传输协议（FTP）的控制信道使用的就是简单文本协议。FTP 用来向远程机器上传数据或是从远程机器上下载数据，其数据传输信道用来传输二进制数据，而其控制信道使用的是简单文本协议。由于 FTP 控制信道使用的协议是人工可阅读的，因此用户可以采用人工方式，使用命令行工具与 FTP 服务器进行通信。

```
C:\>nc 192.168.1.2 21
220 Microsoft FTP Service
USER Administrator
331 Password required for Administrator.
PASS password
230 User Administrator logged in.
PWD
257 "/" is current directory.
QUIT
221
```

在上面的例子中，我们使用了 Netcat 工具[3]连接到一台 Microsoft FTP 服务器上。市

[3] http://www.vulnwatch.org/netcat/

面上有多种 FTP 客户端，但 Netcat 工具比较适合用来展示客户端和服务端之间的通信。因为该工具能够让我们完全控制发送给服务器的请求，并将通信内容完全显示出来。在上面这个例子中，粗体文本是我们发送给服务器的请求，非粗体的部分是服务器返回的信息。可以清楚的看到，交互过程中所有的请求和响应都是人工可阅读的。这种方式使得使用者可以清楚地看到发生了什么，以及方便的调试可能遇到的任何问题。

4.4 二进制协议

二进制协议很难被人所理解，因为在二进制协议数据流中你看到的只是一堆原始数据，而不是可读的文本。如果不理解协议，协议数据包就没有任何意义。对模糊测试来说，如果你希望对协议中的某个位置进行测试，那就必须理解协议的结构。

为了更好地说明如何才能获得对一个二进制协议的恰当理解并据此进行模糊测试，我们将考察一个全功能的协议：AOL 实时消息（AOL Instance Messenger，AIM）协议。该协议目前每天被数百万人用来和朋友聊天。特别地，我们主要考察 AIM 登录过程中发送和接收的数据包。

在深入研究 AIM 协议规范之前，我们先来讨论一些在开发二进制协议中常用的基本结构。AIM 是一个专有协议[4]，虽然没有官方文档，但由于不少人对协议进行过逆向工程，目前该协议结构的大量细节都可以获得。AIM 协议的官方名称叫做实时通信开放系统（Open System for CommunicAtion in Realtime，OSCAR）。逆向工程使得创建 AIM 客户端的替代品成为可能。例如 GAIM 工具[5]和 Trillian 工具[6]就是 AIM 客户端的替代品。另外，有些网络协议分析工具，如 Wireshark，也能够完全对 AIM 协议数据包进行解码。因此，如果你想要创建一个面向专有协议的模糊测试器，绝不要首先对协议进行逆向工程，而应该去寻找值得借鉴的信息——这一点非常重要。虽然有些情况下你不得不对协议进行逆向工程，但你应该假设，如果你对某个协议或文件格式的细节感兴趣，那一定也会有其他人对这些细节有兴趣。用我们的好朋友 Google 搜索相关的信息，看看有没有其他人做过类似的事情是个好主意。如果要找文件格式相关的信息，试着访问 wotsit.org，该网站收录了上百种专有和公开协议的文档描述，既有官方的也有非官方的。

[4] http://en.wikipedia.org/wiki/AOL_Instant_Messenger

[5] http://gaim.sourceforge.net/

[6] http://www.ceruleanstudios.com/

下面让我们看看 AIM 的认证或登录过程。要理解 AIM 使用的协议，首先要捕捉原始的通信数据，并将其分解为有意义的结构。对我们使用的 AIM 来说，很幸运 Wireshark 工具已经具有了对 AIM 协议的支持[7]。我们可以略过最开始的用于握手的数据包，直接跳到用户登录的位置。图 4.1 展示了 Wireshark 工具输出的初始数据包，在这些数据包中，AIM 用户的用户名被发送给了 AIM 服务器。从图中可以看到，AIM 协议由两个单独的层组成。高层数据包含 AOL 即时通信头，该头标识了发送数据的类型。在这个例子中，通信头说明这个数据包包含了 AIM 登录族（AIM Signon Family）的数据（0x0017）和 AIM 登录子族（AIM Signon Subfamily）的数据（0x0006）。

```
No.     Time        Source                Destination           Protocol Info
     15 3.225113    192.168.1.2           205.188.153.121       AIM Signon AIM Signon,
Sign-on Username: fuzzingiscool

Frame 15 (95 bytes on wire, 95 bytes captured)
Ethernet II, Src: Intel_a4:83:57 (00:0c:f1:a4:83:57), Dst: ZyxelCom_25:d5:72
(00:13:49:25:d5:72)
Internet Protocol, Src: 192.168.1.2 (192.168.1.2), Dst: 205.188.153.121
(205.188.153.121)
Transmission Control Protocol, Src Port: 4971 (4971), Dst Port: 5190 (5190), Seq: 11,
Ack: 11, Len: 41
AOL Instant Messenger
    Command Start: 0x2a
    Channel ID: SNAC Data (0x02)
    Sequence Number: 2
    Data Field Length: 35
    FNAC: Family: AIM Signon (0x0017), Subtype: Sign-on (0x0006)
AIM Signon, Sign-on
    Infotype: 0x0001
    Buddy: fuzzingiscool
        Buddyname len: 13
        Buddy Name: fuzzingiscool

0000  00 13 49 25 d5 72 00 0c f1 a4 83 57 08 00 45 00   ..I%.r.....W..E.
0010  00 51 87 c7 40 00 80 06 49 ff c0 a8 01 02 cd bc   .Q..@...I.......
0020  99 79 13 6b 14 46 1d 02 03 3b e2 5d 3c 37 50 18   .y.k.F...;.]<7P.
0030  ff f5 50 5c 00 00 2a 02 00 02 00 23 00 17 00 06   ..P\..*....#....
0040  00 00 00 00 00 00 00 01 00 0d 66 75 7a 7a 69 6e   ..........fuzzin
0050  67 69 73 63 6f 6f 6c 00 4b 00 00 00 5a 00 00      giscool.K...Z..
```

图 4.1 AIM 登录：发送用户名

在较低的层面上，我们能看到登录过程中 AIM 向服务器传递的实际登录数据。这个例子中 AIM 只发送了消息类型（0x0001），用户名称（用户名是 fuzzingiscool），用户名称的长度（13 个字节，0x0d）。这是一个使用可变长字段的例子。注意紧挨着用户名的前方是用户名的长度值。我们在前面提到，二进制协议通常使用数据块。块的最前面是一个表示块大小的值，其后是块中的实际数据（在这个例子中，块中的数据是用户名）。有时候块大小的值包含了该值本身所占的字节数，而有时候块大小的值仅表示实际数据

[7] 译者注：本书写于 2007 年，当时的 AIM 工具使用的协议与当前 AIM 工具使用的协议不同。译者使用的 AIM（版本 7.5.11.9）使用 HTTPS 协议与服务器交互，Wireshark 不能直接对其协议解包。

本身的大小。对模糊测试来说，数据块是一个重要的概念。如果你打算创建一个模糊测试器来向数据块中注入数据，那就必须仔细调整块大小的值，否则，接收到数据的应用就理解不了数据包。网络协议模糊测试器，如 SPIKE[8]工具，就是通过向数据块中注入数据实现模糊测试的，因此，SPIKE 工具在设计时必须注意调整块大小的值。

图 4.2 显示了服务端对客户端初始请求的回应数据。服务器产生了一个验证码（challenge value）用于回应客户端请求。AIM 客户端使用该验证码来生成用户口令的哈希值。再次强调，由于验证码是用数据块表示，因此在验证码前有一个表示验证码长度的值。

```
No.     Time        Source                Destination           Protocol Info
     17 3.253298    205.188.153.121       192.168.1.2           AIM Signon AIM Signon,
Sign-on Reply

Frame 17 (82 bytes on wire, 82 bytes captured)
Ethernet II, Src: ZyxelCom_25:d5:72 (00:13:49:25:d5:72), Dst: Intel_a4:83:57
(00:0c:f1:a4:83:57)
Internet Protocol, Src: 205.188.153.121 (205.188.153.121), Dst: 192.168.1.2
(192.168.1.2)
Transmission Control Protocol, Src Port: 5190 (5190), Dst Port: 4971 (4971), Seq: 11,
Ack: 52, Len: 28
AOL Instant Messenger
    Command Start: 0x2a
    Channel ID: SNAC Data (0x02)
    Sequence Number: 3005
    Data Field Length: 22
    FNAC: Family: AIM Signon (0x0017), Subtype: Sign-on Reply (0x0007)
AIM Signon, Sign-on Reply
    Signon challenge length: 10
    Signon challenge: 3740020309

0000  00 0c f1 a4 83 57 00 13 49 25 d5 72 08 00 45 20   .....W..I%.r..E
0010  00 44 aa 99 40 00 61 06 46 1a cd bc 99 79 c0 a8   .D..@.a.F....y..
0020  01 02 14 46 13 6b e2 5d 3c 37 1d 02 03 64 50 18   ...F.k.]<7...dP.
0030  40 00 b2 20 00 00 2a 02 0b bd 00 16 00 17 00 07   @.. ..*.........
0040  00 00 00 00 00 00 00 0a 33 37 34 30 30 32 30 33   ........37400203
0050  30 39                                            09
```

图 4.2 AIM 登录：服务器响应

一旦收到验证码，客户端就会将用户的显示名（screen name）和用户口令的哈希值传给服务端。此外，客户端传给服务端的数据还包括许多客户端登录时的细节，这些细节信息可以帮助服务端确定客户端具有哪些功能。细节信息是以**名字-值对**（name [nd] value pairs）的方式进行传输的，**名字-值对**也是二进制协议中常用的数据格式。**名字-值对**的一个例子是客户端 id 字符串（client id string），该字符串和 AIM 的版本号等信息（在这里是字符串“AOL InstanceMessenger, version 5.5.3591/WIN32”）一起被传给服务器。请注意，AIM 传递的值字段中包含数据块的长度值。

[8] http://www.immunitysec.com/resources-freesoftware.shtml

```
No.     Time        Source                Destination           Protocol Info
     18 3.263960    192.168.1.2           205.188.153.121       AIM Signon AIM Signon, Logon

Frame 18 (215 bytes on wire, 215 bytes captured)
Ethernet II, Src: Intel_a4:83:57 (00:0c:f1:a4:83:57), Dst: ZyxelCom_25:d5:72
(00:13:49:25:d5:72)
Internet Protocol, Src: 192.168.1.2 (192.168.1.2), Dst: 205.188.153.121 (205.188.153.121)
Transmission Control Protocol, Src Port: 4971 (4971), Dst Port: 5190 (5190), Seq: 52, Ack: 39,
Len: 161
AOL Instant Messenger
    Command Start: 0x2a
    Channel ID: SNAC Data (0x02)
    Sequence Number: 3
    Data Field Length: 155
    FNAC: Family: AIM Signon (0x0017), Subtype: Logon (0x0002)
        Family: AIM Signon (0x0017)
        Subtype: Logon (0x0002)
        FNAC Flags: 0x0000
            .... .... .... ...0 = Followed By SNAC with related information: Not set
            0... .... .... .... = Contains Version of Family this SNAC is in: Not set
        FNAC ID: 0x00000000
AIM Signon, Logon
    TLV: Screen name
        Value ID: Screen name (0x0001)
        Length: 13
        Value: fuzzingiscool
    TLV: Password Hash (MD5)
        Value ID: Password Hash (MD5) (0x0025)
        Length: 16
        Value
    TLV: Unknown
        Value ID: Unknown (0x004c)
        Length: 0
        Value
    TLV: Client id string (name, version)
        Value ID: Client id string (name, version) (0x0003)
        Length: 45
        Value: AOL Instant Messenger, version 5.5.3591/WIN32
    TLV: Client id number
    TLV: Client major version
    TLV: Client minor version
    TLV: Client lesser version
    TLV: Client build number
    TLV: Client distribution number
    TLV: Client language
    TLV: Client country
    TLV: Use SSI

0000  00 13 49 25 d5 72 00 0c f1 a4 83 57 08 00 45 00   ..I%.r.....W..E.
0010  00 c9 87 c8 40 00 80 06 49 86 c0 a8 01 02 cd bc   ....@...I.......
0020  99 79 13 6b 14 46 1d 02 03 64 e2 5d 3c 53 50 18   .y.k.F...d.]<SP.
0030  ff d9 73 de 00 00 2a 02 00 03 00 9b 00 17 00 02   ..s...*.........
```

图 4.3　AIM 登录：发送登录凭证

4.5　网络协议

FTP 和 AIM 协议都属于网络协议。因特网包含了各种各样的网络协议，例如数据传输（data transfer）协议，路由（routing）协议，电子邮件（e-mail）协议，流媒体（streaming media）协议，即时通信（instance messaging）协议，甚至还包括一些超乎你想象的协议。如智者所言，“关于标准，最好的事情就是有足够多的标准可供选择”。这句话对网络协议同样适用。

网络协议是如何被开发设计的？这个问题的答案极大依赖于该协议是专有的还是公开的。专有协议由一个公司中的封闭小组开发，协议用于特定产品，并由该公司维护和控制。专有协议有其固有的优点，因为专有协议的开发者只需要在小范围内就标准达成一致就可以了。相对的，因特网协议由于其开放性，因此需要在大量不同群体间达成一致。一般而言，因特网协议是由因特网工程任务组（Internet Engineering Task Force，IETF）[9]开发并维护的。IETF 有一个冗长的流程来发布和获取因特网标准的提案。该流程要求首先发布评论请求（Requests for Comments，RFCs），这些 RFC 文档描述协议的细节，供同行评审。经过若干轮讨论和修订后，RFC 文档才能最终被 IETF 接受为因特网标准。

54

4.6 文件格式

和网络协议一样，文件格式用来描述文件的数据结构，可以是开放的，也可以是专有的。为了进一步说明文件格式，让我们考虑 Microsoft Office 文档。历史上，Microsoft 一直在其生产率应用（如 Microsoft Office，Excel，PowerPoint）中使用专有文件格式。与其竞争的开放文档格式（OpenDocument format，ODF）[10]是一种由 OpenOffice.org 创建的文件格式，该文件格式随后被高级结构化信息标准化组织（Organization for the Advanced Structured Information Standards，OASIS[11]）所采纳。OASIS 是一个由主要 IT 厂商支持的工业组织，其目标是开发和接纳电子商务（e-business）标准。

作为对 OASIS 采纳 ODF 标准的回应，Microsoft 于 2005 年 9 月宣布 Microsoft Office 的后续版本将会使用新的 Microsoft 开放 XML 格式[12]（Microsoft Open XML）。这种新格式是一种开放格式，但与 ODF 并不相同。随后，Microsoft 宣布了开放 XML 翻译器项目[13]（Open XML Translator Project），该项目承诺为相互竞争的 XML 格式提供翻译工具。

55

接下来让我们对简单文本文件格式和专有二进制文件格式进行比较。用来进行对比的是使用 OpenOffice Writer 和 Microsoft 2003 创建的具有相同内容的文件。我们的方法是首先创建一个空文档，使用默认字体向文档中添加“fuzzing”这个词，然后保存文件。

---

[9] http://www.ietf.org

[10] http://en.wikipedia.org/wiki/OpenDocument

[11] http://www.oasis-open.org

[12] http://www.microsoft.com/office/preview/itpro/fileoverview.mspx

[13] http://sev.prnewswire.com/computer-electronics/20060705/SFTH05506072006-1.html

默认情况下，OpenOffice Writer 会将文档保存为二进制文件。如果使用十六进制编辑器（hex editor）打开文件，它看上去就是一个普通的二进制文件。然而，如果将该文件的后缀改为 zip 并使用解压缩工具将其打开，你就会发现这个文件其实是一个包含了 XML 文件，目录，图片和其他一些文本文件的压缩包。其中包含的文件和目录如下所示：

```
Directory of C:\Temp\fuzzing.odt

07/18/2006  12:07 AM       <DIR>              .
07/18/2006  12:07 AM       <DIR>              ..
07/18/2006  12:07 AM       <DIR>              Configurations2
07/18/2006  04:05 AM                  2,633   content.xml
07/18/2006  12:07 AM       <DIR>              META-INF
07/18/2006  04:05 AM                  1,149   meta.xml
07/18/2006  04:05 AM                  1,149   mimetype
07/18/2006  04:05 AM                  7,371   settings.xml
07/18/2006  04:05 AM                  8,299   styles.xml
07/18/2006  12:07 AM       <DIR>              Thumbnails
               5 File(s)             19,491 bytes
               5 Dir(s)      31,203,430,400 bytes free
```

解压后的包中包含的这些文件分别描述了文档的内容或是格式等。所有这些文件都是 XML 文件，所以我们可以通过文本编辑器，用人工的方式阅读这些文件。下文列出了 content.xml 文件的内容。可以看到，在一连串描述命名空间和格式细节的 XML 元素之后，我们能清楚的看到文件的内容（字符串“fuzzing”）。为便于识别，我们将该内容字符串以黑体字标出。

```
<?xml version="1.0" encoding="UTF-8"?>
 <office:document-content
   xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0"
   xmlns:style="urn:oasis:names:tc:opendocument:xmlns:style:1.0"
   xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
   xmlns:table="urn:oasis:names:tc:opendocument:xmlns:table:1.0"
   xmlns:draw="urn:oasis:names:tc:opendocument:xmlns:drawing:1.0"
   xmlns:fo="urn:oasis:names:tc:opendocument:xmlns:xsl-fo-compatible:1.0"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xmlns:dc="http://purl.org/dc/elements/1.1/"
   xmlns:meta="urn:oasis:names:tc:opendocument:xmlns:meta:1.0"
   xmlns:number="urn:oasis:names:tc:opendocument:xmlns:datastyle:1.0"
   xmlns:svg="urn:oasis:names:tc:opendocument:xmlns:svg-compatible:1.0"
   xmlns:chart="urn:oasis:names:tc:opendocument:xmlns:chart:1.0"
   xmlns:dr3d="urn:oasis:names:tc:opendocument:xmlns:dr3d:1.0"
   xmlns:math="http://www.w3.org/1998/Math/MathML"
```

```
xmlns:form="urn:oasis:names:tc:opendocument:xmlns:form:1.0"
xmlns:script="urn:oasis:names:tc:opendocument:xmlns:script:1.0"
xmlns:ooo="http://openoffice.org/2004/office"
xmlns:ooow="http://openoffice.org/2004/writer"
xmlns:oooc="http://openoffice.org/2004/calc"
xmlns:dom="http://www.w3c.org/2001/xml-events"
xmlns:xforms="http://www.w3.org/2002/xforms"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
office:version="1.0">
<office:scripts/>
<office:font-face-decls>
  <style:font-face style:name="Tahoma1" svg:font-family="Tahoma"/>
  <style:font-face style:name="Times New Roman"
    svg:font-family="&apos;Times New Roman&apos;"
    style:font-family-generic="roman"
    style:font-pitch="variable"/>
  <style:font-face style:name="Arial"
    svg:font-family="Arial"
    style:font-family-generic="swiss"
    style:font-pitch="variable"/>
  <style:font-face style:name="Lucida Sans Unicode"
    svg:font-family="&apos;Lucida Sans Unicode&apos;"
    style:font-family-generic="system"
    style:font-pitch="variable"/>
  <style:font-face style:name="Tahoma"
    svg:font-family="Tahoma"
    style:font-family-generic="system"
    style:font-pitch="variable"/>
</office:font-face-decls>
<office:automatic-styles/>
<office:body>
  <office:text>
    <office:forms form:automatic-focus="false" form:apply-design-mode="false"/>
    <text:sequence-decls>
      <text:sequence-decl text:display-outline-level="0" text:name="Illustration"/>
      <text:sequence-decl text:display-outline-level="0" text:name="Table"/>
      <text dequence-decl text:display-outline-level="0" text:name="Text"/>
      <text:sequence-decl text:display-outline-level="0" text:name="Drawing"/>
    </text:sequence-decls>
    <text:p text:style-name="Standard">fuzzing</text:p>
  </office:text>
</office:body>
```

```
</office:document-content>
```

如果使用 Microsoft Word 2003 创建一个具有同样内容的文件，我们得到的是一个二进制的 Word 文档（*.doc）文件。除了是二进制文件外，该文件比前面我们创建的 OpenDocument 文本文件要大（doc 文件大小接近 20KB，而 OpenOffice 文件大小约为 7KB）。在十六进制编辑器中打开该 doc 文件，可以发现 doc 文件由一些可以人工阅读的字符串和难以理解的十六进制值构成。下面展示了该 doc 文件原始内容的一个片段，从这个片段中，我们可以看到文档的作者，以及创建本文档的应用程序的一些信息。

```
00000a00h: 66 75 7A 7A 69 6E 67 0D 00 00 00 00 00 00 00 00 ; fuzzing.........
…
000024d0h: 66 75 7A 7A 69 6E 67 00 1E 00 00 00 04 00 00 00 ; fuzzing.........
000024e0h: 00 00 00 00 1E 00 00 00 1C 00 00 00 4D 69 63 68 ; ............Mich
000024f0h: 61 65 6C 2C 20 41 64 61 6D 20 61 6E 64 20 50 65 ; ael, Adam and Pe
00002500h: 64 72 61 6D 00 00 00 00 1E 00 00 00 04 00 00 00 ; dram............
…
00002560h: 4D 69 63 72 6F 73 6F 66 74 20 4F 66 66 69 63 65 ; Microsoft Office
00002570h: 20 57 6F 72 64 00 00 00 40 00 00 00 00 00 00 00 ;  Word...@.......
```

4.7 常用协议元素

为什么这部分背景知识对模糊测试很重要？因为文件格式和网络协议的结构不同，要求的模糊测试方法也就不同。对数据结构了解得越多，就越能够在模糊测试中关注那些易于引发异常的协议中的部分。下面就让我们来了解协议中的一些常用元素。

4.7.1 名字-值对

无论是二进制协议还是简单文本协议，其中的数据经常以**名字-值对**（例如，size=12）的形式表示，对简单文本协议来说尤其如此。回想我们前面展示的 content.xml 文件，整个 XML 文件中到处都是**名字-值对**。一个一般性的规则是，通过对**名字-值对**中“值”的部分进行模糊测试，通常可以发现潜在的安全漏洞。

4.7.2 块识别符

块识别符用来标识二进制数据块的数据类型。块标示符后面通常紧跟着变长或是定长的数据。在前面讨论的 AIM 的例子中，AIM 登录头包含了值为 0x0001 的块标识符。该值（0x0001）定义了后续数据的类型（在 AIM 的例子中，这个值指的是 AOL 用户名）。

模糊测试可以发现一些文档中没有记录的块识别符，这些文档中没有记录的块标识符很可能会接受额外的数据类型，并因此带来安全风险，因此通常需要对其进行模糊测试。

4.7.3 块大小

块大小在前文中提到过。块通常由诸如名字-值对这样的数据，以及紧挨着数据的一个或多个字节组成。紧挨着数据前方的这些字节用于说明块中的数据类型、块的大小。有一种模糊测试方法是修改数据包中表示块大小的值，使其比实际数据块的数据大小稍大或是稍小一些，然后观察结果输出。缓冲区溢出（buffer overflow）和缓冲区下溢（buffer underflow）主要就是通过这种模糊测试方法发现的。另外，如果要替换块中的数据进行模糊测试，一定要根据实际数据的大小调整块大小的值，只有这样，被测应用才能准确识别数据块中的数据。

4.7.4 校验和

有些文件格式在文件中嵌入了校验和，以此帮助应用发现某些原因导致的数据破坏。从安全的角度来说，这种做法并不是必需的，因为文件很多原因都会导致文件被破坏。但如果文件包含校验和的话，对于读取此文件的应用进行的模糊测试就会受到影响，因为应用程序通常会在发现校验和不正确后放弃对文件的处理。PNG 图像格式使用了校验和。在对带校验和的文件格式进行模糊测试时，务必要让模糊测试器把校验和考虑在内，模糊测试器可以自行计算校验和并将其写入文件，这样就可以保证被测应用能够正常处理用于模糊测试的文件。

4.8 小结

虽然可以使用强制性模糊测试直接对文件和网络通信进行测试，但显然，仅对可能导致安全漏洞的数据部分进行模糊测试效率更高。当然，要对协议有好的理解，就需要付出一定的前期努力，但这些努力通常都是值得的，尤其是在开放和专有协议的文档都很齐全的情况下。经验有助于识别协议中最佳的进行模糊测试的数据部分，因此，希望本章描述的这些内容能够帮助读者注意到过去那些导致安全漏洞的常见的弱点区域。在第 22 章“自动化协议分析”中，我们将会讨论一些用于剖析协议结构的自动化技术。

第5章

有效模糊测试的需求

"You teach a child to read, and he or her will be able to pass a literacy test."

——George W. Bush, Townsend, TN, February 21, 2001

在前面的章节中，我们介绍了不同的模糊测试器类型和不同的进行模糊测试的方法。在本章中我们将讨论一些可用于更有效和更高效地进行模糊测试的技巧和技术。一些明显的因素，如对测试的可重现性和可复用性的计划应该在开发模糊测试器之前就进行考虑，这样可以帮助保证未来的工作能够建立在当前工作的基础上。此外，在本章中，我们还会讨论更复杂的模糊测试器特性，例如模糊测试器的过程状态和深度，跟踪和度量，错误检测，以及约束等。在随后本书的第二部分，我们将会讨论一些模糊测试器的测试目标，以及如何构造针对这些测试目标的自动化工具。在阅读这些章节时，请记住本章介绍的这些概念，因为这些概念对于创建新的模糊测试器，甚至是像第 12 章“UNIX 平台上的文件格式自动化模糊测试”中讨论的那样创建模糊测试框架都是非常重要的。尽管目前市面上在售的商业模糊测试工具不少，但没有一个工具能够满足我们在本章中将要覆盖的所有需求。

5.1 可重现性与文档

对模糊测试工具的一个显而易见的需求是它应该能够重现单个测试和一组测试的测试结果。这个需求对于测试者与他人或是其他组沟通测试结果至关重要。模糊测试者应该能够向模糊测试工具提供一组测试用例，而这些测试用例每次运行时被测应用的行

为应该是毫无二致的。考虑下面这个假想的状况：你正在对一个 Web 服务器进行模糊测试，测试其处理不完整的 POST 数据的能力。你发送的第 50 个请求导致 Web 服务器发生了崩溃，并据此发现了一个可能被利用的内存破坏条件。随后，你重新启动了 Web 后台进程，向服务器重新发送导致服务器崩溃的第 50 个请求，但这次什么都没有发生。这是否意味着这个缺陷是偶然的吗？当然不是：计算机是确定性的，完全不会有任何真正意义上的“随机”。测试中发生的崩溃一定是由于输入的组合导致的。也许一个稍早的数据包导致 Web 服务器处于一个特殊状态，而在这个状态下第 50 个数据包触发了内存破坏。除非我们能够系统地回放整个测试过程，否则，我们就不能对此进行进一步分析，也不能发现导致崩溃的具体原因。

即使不是强制规定要编写文档，从信息共享的角度来说，对测试结果的文档记录也是很有帮助的。在国际化外包开发的增长趋势下[1]，通常情况下，安全测试者不可能跨越地域，坐在受影响的产品开发者旁边。当前情况下，外包已经普遍到计算机专业的学生都知道如何利用它了[2]。外包方式存在各种各样的沟通障碍，包括时区，语言，沟通媒介等，因此，把尽可能多的信息打包成清晰且精确的形式益发重要。组织的文档化负担不应该是一个完全手工的努力。一个好的模糊测试工具需要产生和存储易于分析和引用的日志信息。

我们需要考虑前面讨论的模糊测试器如何处理可重现性，如何实现自动的日志以及文档记录。另外，还需要考虑到如何改进其具体实现。

5.2 可重用性

从大的方面考虑，如果我们打算建立一个文件格式模糊测试工具，那我们肯定不希望每次测试新的文件格式时重写整个工具。我们可以创建一些可重用的功能，这些功能能在测试不同的文件格式时节省我们的时间。继续以文件格式的模糊测试为例，假设我们需要构造一个 JPEG 文件格式模糊测试工具来测试 Microsoft 画图软件（Microsoft Paint）中的缺陷。根据事先考虑到的可重用的部分，我们决定将工具分为图 5.1 所示的三个组件。

[1] http://www.outsourceworld.org/, http://money.cnn.com/2003/09/17/news/economy/outsourceworld/

[2] “Computer Science Students Outsource Homework”, http://developers.slashdot.org/developers/06/01/19/0026203.shtml

JPEG 文件生成器负责不停地生成变异的 JPEG 文件。目标运行器负责在 JPEG 文件生成器产生的文件上循环，每次以不同的参数启动 Microsoft 画图程序，使其加载新生成的图片文件。最后，错误检测引擎负责监视 Microsoft 画图程序的每个实例以发现可能的异常状况。将该模糊测试器分为三个部分允许只需要改动 JPEG 文件生成器就能使该模糊测试器适应其他文件格式的测试集合。

从小的方面考虑，许多小的组件应该是可在我们的模糊测试项目中移植的。例如，考虑电子邮件地址（Email）。电子邮件地址这种基本的字符串格式到处都有，在简单邮件传输协议（Simple Mail Transfer Protocol，SMTP）事务，登录屏幕，IP 语音（Voice over IP，VoIP）会话初始化协议（Session Initiation Protocol，SIP）等中都包含了 Email 域：

SIP 邀请事务片段

```
49 4e 56 49 54 45 20 73 69 70 3a 72 6f 6f 74 40   INVITE sip:root@
6f 70 65 6e 72 63 65 2e 6f 72 67 20 53 49 50 2f   openrce.org SIP/
32 2e 30 0d 0a 56 69 61 3a 20 53 49 50 2f 32 2e   2.0..Via: SIP/2.
30 2f 55 44 50 20 70 61 6d 69 6e 69 4c 2e 75 6e   0/UDP voip.openr
```

在上述的每一种情况中，Email 字段都是一个可用于模糊测试的有趣的字段，因为这个字段一定会被解析成多个部分（例如，用户名和邮件地址部分）。如果我们需要花时间来枚举可能“有害”的电子邮件地址，那么，在所有的模糊测试器中复用组件不是个好主意吗？

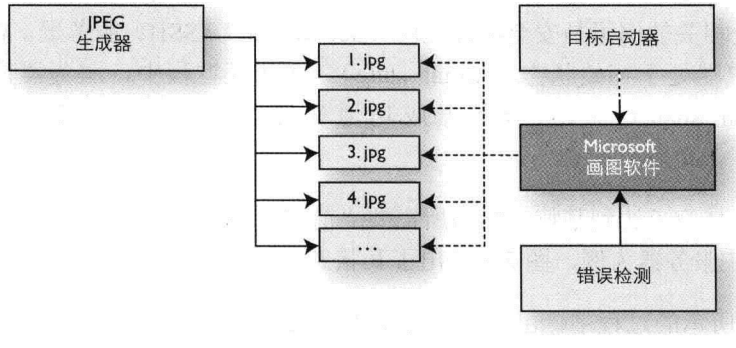


图 5.1　一个虚构的文件格式模糊测试器的结构分解与总览

5.3　过程状态和过程深度

为了使读者掌握过程状态和过程深度的概念，下面举一个大多数人都非常熟悉的例

子：ATM 取款业务。考虑图 5.2 所示的简单的状态迁移图。

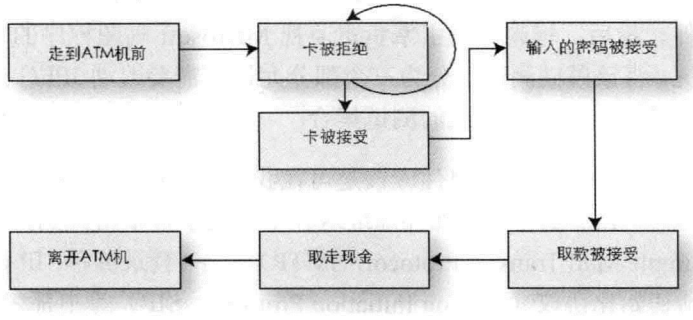


图 5.2　人为设想的 ATM 状态图例子

在一次典型的 ATM 交易中，你走到机器旁（必须小心以防有人跟踪），插入你的卡，输入密码，从屏幕菜单中选择菜单项，选择你想要取款的金额，取走你的钱，结束交易。软件的状态和状态迁移的概念和这个是一样的。稍后我们将给出另一个具体的例子。ATM 交易过程的每一步可以被看做一个状态。我们将过程状态（process state）定义为目标进程在任意给定时刻所处的具体状态。动作，例如插入一张卡或是选择一个取款金额这类动作，可以导致系统从一个状态迁移到另一个状态。当前状态相对于初始状态的距离被称为过程深度（process depth）。所以，“输入取款金额”是比“输入口令”更深的步骤。

一个安全相关的例子是安全解释程序（Secure Shell，SSH）服务器。在连接到服务器之前，客户端处于初始状态（initial state）。在认证过程中，服务器位于认证状态（authentication state）。一旦服务器成功地认证了用户，它就处于已认证状态（authenticated state）。

过程深度是为了达到某特定状态所需要“前进（forward）”的步数的度量。仍以我们提到的 SSH 服务器为例，图 5.3 给出了其状态图。

在图 5.3 所示的过程中，已认证状态比认证状态位于更“深”的位置，因为认证状态是达到已认证状态所需的一个子步骤。过程状态与过程深度的概念非常重要，因为使用这些概念可以创造非常复杂的模糊测试器。下面是一个复杂情况的例子。为了对一个 SMTP 服务器的 MAIL FROM 命令的参数进行模糊测试，就必须连接到服务器并发送一个 HELO 或是 EHLO 命令。如图 5.4 所示，无论用户使用 HELO 还是 EHLO 的初始命令，底层的 SMTP 实现可能用同一个函数处理 MAIL FROM 命令。

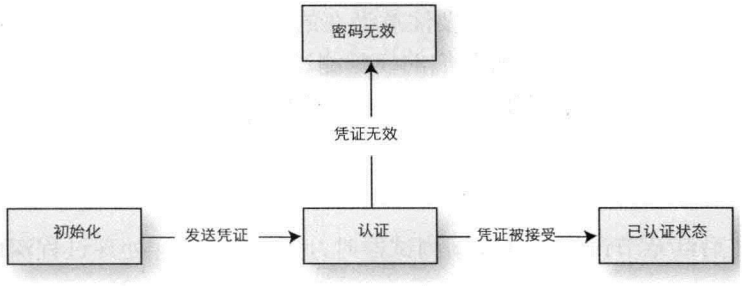


图 5.3　SSH 服务器状态图

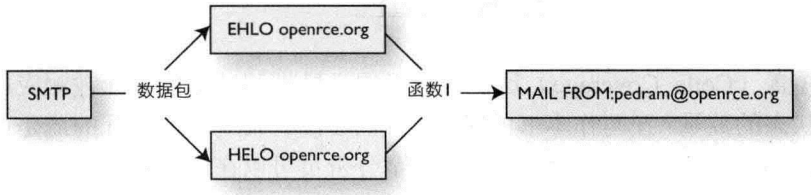


图 5.4　示例的 SMTP 状态图 1

在图 5.4 中，函数 1 是唯一的用来处理 MAIL FROM 数据的函数。另一种设计方案如图 5.5 所示，在这种设计中，根据不同的初始化命令，SMTP 可能实现了两个独立函数分别处理 MAIL FROM 命令。

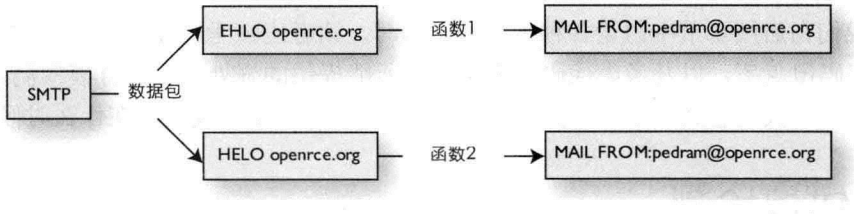


图 5.5　示例的 SMTP 状态图 2

事实上，图 5.5 展示的是一个真实的例子。2006 年 9 月 7 日，一家安全咨询机构[3]公开了一个可被利用的 SMTP 服务器中的栈溢出。该 SMTP 服务器捆绑在 Ipswitch Collaboration Suite 软件套件中。如果在输入的 email 地址的@字符和：字符间塞入一个长字符串，就会发生堆栈溢出。包含该安全漏洞的代码只有当客户端以 EHLO 开始会

[3] http://www.zerodayinitiative.com/advisories/ZDI-06-028.html

话时才可达。在设计模糊测试器时，要注意潜在的类似这样的逻辑分离。在这个例子中，要想让模糊测试获得完全的覆盖，我们的模糊测试器就不得不将所有突变产生的电子邮件地址处理两次，一次通过 EHLO，一次通过 HELO。那么，如果存在另一个逻辑分离使得过程深度路径更深呢？显然，完全覆盖所需要的迭代次数将会随着逻辑分离的数量增加而以指数级增长。

当我们在后续章节讨论各种模糊测试器时，应该考虑怎样处理过程深度的变化和逻辑分离。

5.4 跟踪、代码覆盖和度量

代码覆盖（Code Coverage）这个术语指的是模糊测试器能够让被测应用达到和执行的过程状态的数量。在写作本书的时候，我们还没有发现任何公开的或是商业的模糊测试技术能够跟踪和记录代码覆盖。这是安全分析员可以在未来对这一领域进行开拓性研究的一个重要方向。质量保证（Quality Assurance，QA）团队能够利用代码覆盖率建立对测试的信心。例如，假设你是 Web 服务器产品的 QA 团队负责人，与发布 25%的代码覆盖率基础上的零缺陷产品相比，发布一个 90%的代码覆盖率基础上的零缺陷产品一定会让你觉得更舒服。另外，安全漏洞研究员也能够从代码覆盖率分析中获益，代码覆盖率分析能够识别出少有人注意到的代码角落，从而能够帮助实现更好的代码覆盖。这个重要的概念将在第 23 章“模糊测试器跟踪”中进行深入讨论。

当我们在后续章节中讨论各种模糊测试器时，读者应该考虑使用一些创造性的方法来确定代码覆盖率，以及考虑代码覆盖率分析所能提供的好处。在进行模糊测试时，人们经常会问，“我们怎么开始？”，然而，另一个问题同样重要：“我们该在什么时候结束？”

5.5 错误检测

产生和传输可能引发异常的数据只是打赢了模糊测试战役的一半。战役的另一半是精确的识别出一个已经出现的错误。截止到写作本书时，主要的模糊测试器在这方面都是“盲目的”，它们不知道被测目标应用是如何对测试数据作出反应的。一些商业解决方案在两次请求之间加入“ping”或是存活检查（keepalive checks）来检测被测目标应用是否仍然正常工作。这里的术语“ping”指某种形式的事务，被测目标应该接受该事务并产生一个已知的正确响应。另外一些检查被测应用是否正常工作的方案基于日志输出分析。这种方式通过监视系统日志（如 Windows 事件浏览器）中被测应用产生的输

出来判断被测目标是否正常，图 5.6 展示了 Windows 事件浏览器的截图。

这些错误检测方法的优点在于他们基本上都可以很容易的在不同平台和架构上实现。然而，这些方法仅能够检测出极其有限类型的错误。例如，这些方法中没有一种方法能够发现 Microsoft Windows 应用中发生的，被结构异常处理（Structured Exception Handling）[4]例程处理的错误。

下一代的错误检测技术是使用轻量级的调试客户端来检测异常条件在被测目标中的发生。例如，将在本书第二部分中讨论的 FileFuzz 工具就包含一个可定制的，开源的 Microsoft Windows 调试客户端。利用调试客户端的工具的弊端在于你不得不为每个需要在其上运行测试的平台开发一个调试客户端。例如，如果想要在 Mac OS X，Microsoft Windows 和 Gentoo Linux 上测试三个 SMTP 服务器，你就不得不开发两个甚至是三个不同的监视客户端。此外，受测试目标的限制，有可能根本就无法，或是不能在给定时 68
间内创建出需要的调试客户端。比如，如果需要测试的目标是硬件 VoIP 电话，由于硬件系统更难于被调试，需要专用的工具，因此，可能不得不采用对测试实施控制或对日志实施监控的老办法。

如果进一步展望错误检测技术，错误检测的万能药很可能是诸如 Valgrind[5]和 Dynamo Rio[6]这样的动态二进制插装/翻译平台[7]（DBI）。在这些平台上，有可能在开发时就完成错误检测而不是在错误发生后进行检测。如果我们站在足够高的高度上看（例如，50000 英尺），基于 DBI 的调试引擎能够在底层上非常有针对性地对被测目标进行分析和插装。这种底层的控制允许内存泄漏检查，缓冲区溢出检查和越界检查等等。回到我们前面讨论可重现性时提到的内存破坏的例子，轻量级的调试客户端能够在发生内存破坏时通知我们。在那个例子中，我们将大量数据包发送给目标服务，第 50 个数据 69
包引发了服务的崩溃。在 Valgrind 平台上，我们也许能够在异常被触发前就能通过更早的测试检测到初始的内存破坏。这种方式能够节省数小时，甚至是数天的模糊测试调优和缺陷跟踪的时间。

在后续章节中我们会讨论各种不同的模糊测试器。读者应该考虑各种目标监视方法，并决定在各种情况下最适合选用的方法。

---

[4] hrrp://msdn2.microsoft.com/en-us/library/ms680657.aspx

[5] Valgrind: http://valgrind.org/

[6] Dynamo RIO: http://www.cag.lcs.mit.edu/dynamorio

[7] http://en.wikipedia.org/wiki/Binary_translation

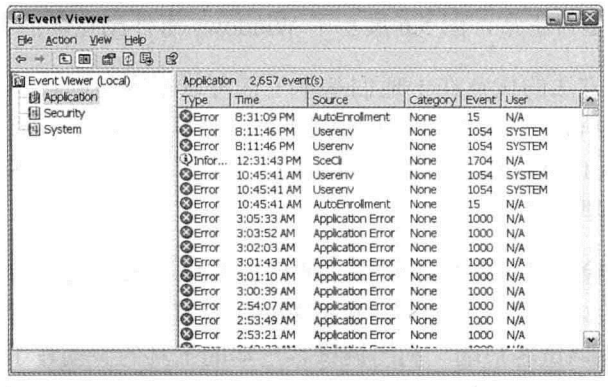


图 5.6 Microsoft Windows 事件浏览器的错误日志样例

5.6 资源约束

许多非技术因素，例如预算和时间期限等，都会给模糊测试带来限制。在模糊测试的设计和计划阶段必须牢记这些因素可能的影响。例如，在计划模糊测试时，你可能会发现自己处于一种紧急关头的恐慌状态，因为没有人哪怕是简单地检查过投资 50 万美元开发出的产品的安全性。安全性通常在软件开发生命周期（SDLC）的最后阶段才被考虑，与“添加新功能”和“符合产品期限”这类工作相比，安全性在优先级上往往忝居末席。然而，如果我们希望开发出安全的软件，就必须在整个软件开发生命周期中考虑安全性，而不仅仅做表面上的安全性工作。这就要求对软件开发生命周期进行根本性的改变，确保在开发的每个阶段都考虑安全性。也就是说，我们需要认识到软件是在真实世界，而不是一个资源充分，缺陷稀少的乌托邦中被开发出来的。因此，在阅读本书时，读者需要在心中对本书提到的各种模糊测试技术进行分类，你需要了解哪些技术适用于时间和金钱有限的情况下，也需要知道哪些技术只能在理想条件下应用。另外，读者还需要考虑在软件开发生命周期的哪些阶段实现这类工具，谁应该负责整个开发过程。

5.7 小结

本章的主要目标是介绍开发“全功能”的模糊测试器所需要的基本思想。在开发、比较和使用模糊测试技术时，回过头来阅读本章的这些概念能够让你更小心地使用各种模糊测试技术。在后续章节中我们将介绍和分析各种模糊测试解决方案，在阅读这些章节时，读者可以考虑如何在应用中实现和提高诸如度量和错误检测这样的需求目标。

第二部分

目标与自动化

第 6 章　自动化与数据生成
第 7 章　环境变量与参数模糊测试
第 8 章　自动化的环境变量与参数模糊测试
第 9 章　Web 应用与服务器模糊测试
第 10 章　Web 应用和服务器的自动化模糊测试
第 11 章　文件格式模糊测试
第 12 章　UNIX 平台上的文件格式自动化模糊测试
第 13 章　Windows 平台上的文件格式自动化模糊测试
第 14 章　网络协议的模糊测试
第 15 章　UNIX 平台上的自动化网络协议模糊测试
第 16 章　Windows 平台上网络协议的模糊测试
第 17 章　Web 浏览器的模糊测试
第 18 章　Web 浏览器的自动化模糊测试
第 19 章　内存模糊测试
第 20 章　自动化内存模糊测试

第6章

自动化与数据生成

"Our enemies are innovative and resourceful, and so we are. They never stop thinking about new ways to harm our country and our people, and neither do we."

——George W. Bush, Washington, DC, August 5, 2004

模糊测试在很大程度上依赖于自动化。相对其他软件测试方法而言，模糊测试主要的优势在于高自动化测试率。人工生成一个个测试用例耗力且乏味，事实上，这类任务很适合用计算机完成。模糊测试器的核心竞争力就在于其能够使用最小的人工干预产生有用的数据。本章聚焦于模糊测试中自动化方法的多个方面，包括选择编程语言、有用的构造块，以及在生成数据的过程中，如何选择合适的模糊值才能彻底测试被测软件，最后这一点尤其重要。

6.1 自动化的价值

自动化的好处一目了然，但为了清晰起见，我们从两方面回顾一下。首先，我们从人工计算能力的角度来强调自动化的价值，然后我们来看看自动化的可重现性。在计算机时代的早期，计算机的计算时间非常昂贵，实际上，那时的计算机计算时间远比人工计算时间昂贵。因此，那时候在大型机系统上编程基本上是非常乏味的手工过程。那时在大型机上编程需要程序员们放入磁带，拨弄开关，手工输入十六进制或是二进制的机器码。随着时光流逝，程序员智慧的价值越来越高，计算机的计算时间也越来越充裕，因此平衡人工计算时间和计算机计算时间的方式也一直在发生变化。今天，随着高级编

程语言，如 Java，.NET 和 Python 等越来越普及，这种趋势仍在继续。这些高级语言牺牲了一些计算时间，却为程序员提供了更简便的开发环境和更快的开发速度。

有鉴于此，让安全分析员连接到基于套接字（socket）的后台程序（daemon），交互式地手工输入数据，并期望由此发现软件漏洞是完全可行的，但安全分析员如果把时间花在其他任务上会更值得。在比较模糊测试和人工的代码评审和二进制审核时，可以得到相同的结论。代码评审和二进制审查需要具有高级技能的分析员的时间，而模糊测试，或多或少的可以由随便什么人执行。自动化至少可以用作模糊测试的第一个步骤，用于减少那些具有高级技能，有能力定位安全缺陷的分析员所需要花在这些任务上的时间。

接下来，我们强调可重现性的必要性。有两个主要的因素说明了可重现性的重要性：

1．如果我们能够为一个 FTP 服务器创建可重现的测试过程，我们就能够很容易地用同样的测试过程测试其他完全不同的 FTP 服务器。否则的话，我们必须花时间为每个不同的 FTP 服务器设计和实现新的模糊测试器。

2．如果一个不寻常的事件序列在被测目标中触发了故障，我们必须要能够重新产生整个序列，这样，通过不断收窄序列范围，才有可能定位到实际触发异常的那些事件上。要求分析员创建大量的测试用例并记住所有用例远称不上是科学的可重现的方法。

简而言之，这些重复耗时的数据生成，重生成，故障监控任务最好由自动化系统来完成。和大多数计算任务一样，很幸运已经有不少已存在的工具和库可以被用在模糊测试的自动化方面。

6.2 有用的工具和库

尽管已经有大量公开可用的模糊测试器脚本，但许多模糊测试开发者还是需要从头开始创建工具[1]。幸运的是，在模糊测试器的设计和实现阶段，许多工具和库都能为你提供帮助。本节列出了一些这样的工具和库（以字母顺序）。

[1] http://www.threatmind.net/secwiki/FuzzingTools

6.2.1 ETHEREAL[2]/WIRESHARK[3]

Wireshark（Ethereal 的一个分支项目）[4]是一个流行的开源网络嗅探器和协议分析工具。虽然并不一定需要依赖于该工具建立自己的模糊测试器，但在构建模糊测试器的研究和调试阶段，该工具毫无疑问能派上用场。此外，Wireshark 同时还自带有大量可作为参考的开源的协议分析器。Wireshark 捕获的所有能被协议分析器识别的数据包会以一系列的字段/值对的形式，而不是以直接的字节数据块的形式呈现。在深入进行人工协议分析之前，首先参考 Wireshark 给出的信息通常会更有效。要了解 Wireshark 中所有可用的分析器，请参考 Wireshark 的源代码库，尤其是其源代码的 epan\dissectors[5]目录。

6.2.2 LIBDASM[6]和 LIBDISASM[7]

Libdasm 和 libdisasm 都是可免费获得的、开源的反汇编库。这两个库都可以被嵌入到你自己的工具中，用于将二进制流反汇编成 AT&T 和 Intel 语法格式的反汇编代码。Libdasm 是由 C 语言开发的，并带有 Python 接口。而 libdisasm 是用 Perl 语言开发的。虽然不需要使用反汇编器来产生网络流量，但如果要实现自动化的错误检测，反汇编器就很重要了。上述两个库在本书中会被广泛使用，尤其是在第 12 章“Unix 平台上的文件格式自动化模糊测试”，第 19 章“内存的模糊测试”，第 20 章“内存的自动化模糊测试”，第 23 章“模糊测试器跟踪”以及第 24 章“智能错误检测”中。

6.2.3 LIBNET[8]/LIBNETNT[9]

Libnet 是一个可免费获取的开源工具库，提供了用于构建和向底层网络包注入数据的高层 API 方法。该工具库隐藏了生成互联网协议（Internet Protocol，IP）层和链路层（link layer）数据的复杂细节，并提供了多种平台上可移植性。如果想要实现一个网络

---

2 http://www.ethereal.com

3 http://www.wireshark.org

4 http://www.wireshark.org/faq.html#q1.2

5 http://anonsvn.wireshark.org/wireshark/trunk/epan/dissectors/

6 http://www.nologin.org/main.pl?action=codeView&codeId=49

7 http://bastard.sourceforge.net/libdisasm.html

8 http://www.packetfactory.net/libnet

9 http://www.securityfocus.com/tools/1559

栈模糊测试器，读者可能会对这个库感兴趣。

6.2.4 LIBPCAP[10]

LibPCAP 以及 Windows 平台上的 WinPCAP[11]都是可免费获取的开源工具库，为轻松创建跨 Unix 和 Windows 平台的网络捕获和分析工具提供了支持。许多网络协议分析工具，例如前面提到的 Wireshark 都是基于该库建立的。

6.2.5 METRO PACKET LIBRARY[12]

Metro Packet Library 是一个 C#库，提供了与 IPv4，TCP，UDP 和 ICMP（互联网控制管理协议）交互的抽象接口。该库能够帮助创建包嗅探器和网络分析工具。在本书的第 16 章“Windows 平台上网络协议的模糊测试”中我们会进一步讨论该库。

6.2.6 PTRACE

在 Unix 平台上进行调试，大部分时候都需要 ptrace()系统调用的帮助。Unix 平台上的进程能够使用 ptrace 控制寄存器状态、内存以及执行过程，并能够通过 ptrace 捕获其他进程发出的信号。在本书的第 8 章“自动化的环境变量与参数模糊测试”，以及第 12 章“Unix 平台上的文件格式自动化模糊测试”中，我们将会讨论使用 ptrace 机制实现的工具。

6.2.7 PYTHON 扩展

在创建模糊测试器时，许多 Python 扩展，如 Pcapy，Scapy 和 PyDbg 等都可以派上用场。Pcapy[13]是 LibPCAP/WinPCAP 的 Python 扩展，使得 Python 脚本能够捕获网络数据。Scapy[14]是一个功能强大、使用简单的数据包操作扩展，它可以用交互方式或库方式使用。Scapy 适用于构建和解码各种不同的协议的数据。PyDbg[15]是一个纯 Python 的

[10] http://www.tcpdump.org

[11] http://www.tcpdump.org/wpcap.html

[12] http://sourceforge.net/projects/dotmetro

[13] http://oss.coresecurity.com/projects/pcapy.html

[14] http://www.secdev.org/projects/scapy

[15] http://openrce.org/downloads/details/208/PaiMei

Windows 32 位调试器，允许通过进程插桩的方式调试进程。PyDbg 库是 PaiMei 逆向工程框架[16]的一个子集，PaiMei 逆向工程框架在本书的第 19、20、23 和 24 章将会涉及。

以上列出的某些库可以在多种编程语言中使用，而另一些则有特定语言的限制。决定采用何种编程语言来开发模糊测试器时，必须考虑你要实现的模糊测试器的特定需求，以及能够满足这些需求的库所支持的编程语言。

6.3 编程语言的选择

有些人把模糊测试器编程语言的选择上升到信仰层面，无论要解决的问题是什么，他们都坚定地选择某种编程语言；而另一些人则遵循“用正确的工具完成正确的任务”这一准则。我们倾向于采用后一种做法。在本书中，读者能够发现用各种编程语言编写的源代码。本书刻意包含了尽可能多的可复用的实际代码示例。不管读者喜欢什么编程语言，我们都鼓励读者在为特定任务选择编程语言之前仔细考虑特定语言的优缺点。

从最高的层次上来说，编程语言之间有一个基础的差别：解释型与编译型。编译语言例如 C 和 C++允许使用者在底层精确和直接地访问底层组件。前文提及的 Libnet 库就是一个在底层上提供接口的库。解释语言，如 Ruby 和 Python 等在比 C 和 C++更高一些的层次上，这些语言提供了更快速的开发能力和无需重新编译就能修改程序的能力。由于语言的差别，因此存在一些将底层接口转换为更高层次接口的库。模糊测试器需要灵活性甚于高质量的代码，因此会使用各种编程语言来编写。从 Shell 脚本到 Java 语言，从 PHP 到 C#，根据你要完成的任务以及你对编程语言的熟练程度，某种语言可能比其他语言更适合你。

6.4 数据生成与模糊试探值（Fuzz Heuristics）

解决“如何生成数据”的问题只是解决方案的一部分。另一个同样重要的部分是“要生成什么数据”。例如，假设我们要创建一个模糊测试器来分析 IMAP 服务器的鲁棒性。在许多 IMAP 动作和构造中我们需要检测的是命令继续请求（Command Continuation Request，CCR），以下是引自 RFC3501 的关于 CCR 的描述[17]。

---

16 http://openrce.org/downloads/details/208/PaiMei

17 http://www.faqs.org/rfcs/rfc3501.html

7.5. 服务器响应–命令继续请求

命令继续请求的响应以一个“+”而不是标签（tag）来指示。这种形式的响应表示服务器已经准备好从客户端接收一个命令的后续部分。在“+”号之后的文本行是命令继续请求响应的剩余部分。

命令继续请求响应应用在 AUTHENTICATE 命令中，用于从服务端向客户端传递数据，并请求额外的客户端数据。此外，如果任何命令的参数是文本（literal）格式的话，也需要使用命令请求响应。

在命令继续请求方式下，除非服务端指示说需要数据，否则客户端不允许向服务端发送数据。这种方式允许服务端逐行处理命令并拒绝错误。命令的剩余部分，包括命令行结尾的回车换行，跟在文本数据序列之后。如果有附加命令参数，文本数据后会有一个空格，然后是附加命令参数。

示例：
```
C: A001 LOGIN {11}
S: + Ready for additional command text
C: FRED FOOBAR {7}
S: + Ready for additional command text
C: fat man
S: A001 OK LOGIN completed
C: A044 BLURDYBLOOP {102856}
S: A044 BAD No such command as "BLURDYBLOOP"
...
```

79

根据 RFC 的描述，任何以“{数字}”形式结尾的命令(在上文中以粗体表示)中的数字指示下一行传入的命令剩余部分的字节数。因此，这个数字是模糊测试的主要目标，但我们应该用什么数字来测试这个字段？有可能用所有可能的数值测试该字段吗？假设目标后台程序（被测对象）能接受最大为 0xFFFFFFFF（4,294,967,295）的 32 位整数值，如果模糊测试器以每秒 1 个用例的速度执行测试用例，大约需要 136 年来完成整个测试！即使我们想办法把模糊测试器的执行速度提升为每秒 100 个测试用例，也需要 500 天左右的时间才能完成整个测试。当我们完成测试时，作为我们测试对象的这个 IMAP 服务器很可能已经退休了。因此，显然我们不可能测试整个数值空间，而应该聪明地选择其中有代表性的一些数值。

由存在潜在危险的模糊字符串和模糊数据组成的集合（inclusion）又被称为模糊试探值（Fuzz heuristics）。让我们来研究几个我们希望包含在智能库（intelligent library）中的数据类型的类别。

6.4.1 整数值

显然，两个边界值（0 和 0xFFFFFFFF）应该包含在我们的整数类型的测试用例列表中。此外，列表中还应该包含哪些值？也许命令继续请求中的这个数值会被用来作为内存分配函数的参数。通常，在该值的基础上增加附加空间以存储头（header），脚（footer），或是作为终止符的 NULL 字符的操作并不罕见。例如下面这段代码：

```
int size = read_ccr_size(packet);
// save space for NULL termination.
buffer = (char *) malloc(size + 1);
```

同理，也可能给定的值会被减去某个值，然后根据得到的结果进行内存分配。当被测程序不准备复制所有指定的数据到新分配的缓冲区时，就会发生这种情况。记住整数上溢（overflows，加的结果导致超出 32 位整数的最大表示范围）和下溢（underflows，无符号减法导致小于 0 的结果）都可能会导致潜在安全问题，因此，谨慎的做法是将靠近边界的数值，例如 0xFFFFFFFF-1，0xFFFFFFFF-2，0xFFFFFFFF-3，……，以及 1，2，3，4 等值都放到测试用例集合中。

80

类似的，可以对给定数值进行倍数运算。例如，如果输入的数据都会被转换成 Unicode，就需要将给定的数值乘以 2。此外，还需要包括额外的两个字节，以确保 NULL 终止符可以被包含进去。下面这段代码演示了这种情况：

```
int size = read_ccr_size(packet);
// Create space for the Unicode converted buffer
// plus Unicode NULL termination (2 bytes).
buffer = (char *) malloc((size * 2) + 2);
```

要在该情况下触发整数溢出，我们需要把以下这些边界附近的值包含在我们的测试用例中：0xFFFFFFFF/2，0xFFFFFFFF/2-1，0xFFFFFFFF/2-2。那么，需要在用例中包含 32 位整数最大值除以 3 的结果附近的值吗？需要在用例中包含最大值除以 4 的结果附近的数值吗？或者，需要在用例中包含 16 位整数的最大值（0xFFFF）附近的数值吗？让我们把所有这些用例和其他我们选定的用例都扔到一个列表中。到目前为止我们的列表包括以下这些用例：

- MAX32 – 16 <= MAX32 <= MAX32+16
- MAX32/2 – 16 <= MAX32/2 <= MAX32/2 + 16
- MAX32/3 – 16 <= MAX32/3 <= MAX32/3 + 16

- MAX32/4 – 16 <= MAX32/4 <= MAX32/4 + 16
- MAX16 – 16 <= MAX16 <= MAX16 + 16
- MAX16/2 – 16 <= MAX16/2 <= MAX16/2 + 16
- MAX16/3 – 16 <= MAX16/3 <= MAX16/3 + 16
- MAX16/4 – 16 <= MAX16/4 <= MAX16/4 + 16
- MAX8 – 16 <= MAX8 <= MAX8/2 + 16
- MAX8/2 – 16 <= MAX8/2 <= MAX8/2 + 16
- MAX8/3 – 16 <= MAX8/3 <= MAX8/3 + 16
- MAX8/4 – 16 <= MAX8/4 <= <MAX8/4 + 16

MAX32 表示 32 位整数的最大值（0xFFFFFFFF），MAX16 表示 16 位整数的最大值（0xFFFF），MAX8 表示 8 位整数的最大值（0xFF）而数值 16 是我们选定的一个我们认为合理的值。根据可用的测试时间和测试结果，读者可以扩大这个范围。当然，读者需要记住，如果被测协议中有数百个整数字段，每增加一个整数试探值（integer heuristic）就会将测试用例的数量增加数百倍。

不过，也不要太迷信这些选中的整数“试探值”（heuristics）。试探值不过是一种“有根据的推测”而已。高级用户能够在反汇编器的帮助下检查被测应用的二进制码，通过检查内存分配和数据拷贝函数的交叉引用，高级用户可以找出潜在让人感兴趣的整数值。查找这类整数值的过程能够通过自动化进行，这部分概念将在第 22 章“自动化协议分析”（Automated Protocol Dissection）中进行讨论。

81

智能数据集的必要性（The Need for a smart data set）

2005 年 9 月 1 日，一份名为“Novell Netmail IMAPD 命令继续请求堆溢出”[18]的安全报告与厂商提供的补丁包同时发布。这份报告详细描述了一个允许远程未授权的攻击者执行任意代码的漏洞，这个漏洞能够危及整个系统的安全。

该被披露的漏洞是由于命令继续请求（CCR）的处理代码导致的。代码中将用户指定的尺寸大小值直接传递给一个名为 MMalloc()的内存分配函数。MMalloc()函数的汇编代码片段如下：

```
; ebx 是被攻击者控制的
00402CA2 lea ecx, [ebx+1]
```

[18] http://pedram.redhive.com/advisories/novell_netmail_imapd/

```
00402CA5 push ecx
00402CA6 call MMalloc
```

可以看到，MMalloc()函数在分配内存前对传入的参数进行了简单的数学操作（ebx+1）。攻击者如果指定一个会导致整数溢出的恶意数据，这样导致分配得到的是一个小的内存块（大小为溢出后的值）。而原始的用于指定的那个大值将会用在随后的 memcpy()中：

```
; 目标是被攻击者分配的
00402D6E rep movsd
00402D70 movecx, edx
00402D72 and ecx, 3
00402D75 rep movsb
```

以上的指令序列会将攻击者提供的数据拷贝到分配的堆的边界之外，覆盖堆，并最终导致彻底的破坏。

Novell 的补丁程序成功解决了该特定问题，不过该补丁并没能够解决所有可能的被攻击者利用的路径。对 Novell 来说，不幸的是，该 IMAP 后台应用会把所有人工可读的数值形式转换为等价的整数。例如，“-1”会被转换成 0xFFFFFFFF，“-2”会被转换成 0xFFFFFFFE 等。因此，在打上补丁后，下面这个请求能够被正确处理：

```
x LOGIN {4294967295}
```

而这个就不能被正确处理：

```
x LOGIN {-1}
```

这些新的攻击路径后来被一个独立研究者发现，Novell 在 2006 年 12 年通过补丁修正了该问题[19]。

6.4.2 字符串重复（String Repetitions）

我们已经讨论了一些需要包含在我们的模糊测试数据集中的“明智的”整数，但哪些字符串是应该包含在我们的模糊测试集中的呢？让我们从下面这个经典的“长字符串”[20]开始：

---

[19] http://www.zerodayinitiative.com/advisoris/ZDI-06-053.html

[20] 如果你不相信，可以自行 Google:
http://www.google.com/search?hl=en&q=%22perl+-e+%27print+%22A%22*%22

```
perl -e 'print "A"*5000'
```

当然，当对字符串进行模糊测试时，除了这个全由 A 组成的长字符串外，我们还希望包含一些额外的字符序列。首先，我们希望在字符串中包含除 A 外的其他 ASCII 字符，例如“B”。这一点很重要，因为，当 Windows 操作系统上发生堆溢出时，堆结构被 A 的 ASCII 值覆盖还是被 B 的 ASCII 覆盖会导致不同的行为。需要在字符串中包含除 A 外的其他 ASCII 字符的另一个原因是，不管你信不信，我已经看到某些产品会在输入中查找或是干脆阻止纯由 A 组成的长字符串。显然，某些供应商已经见识了 AAAAAAAAAAAAAAAA 字符串的威力。

6.4.3　字段分隔符

模糊测试数据集还需要包含非字母数字字符，例如空格和制表符。这些字符经常被用作字段分隔符和终止符。将这些字符随机地放到生成的模糊字符串中能够更好地模拟我们正在模糊测试的协议，由此能让我们的模糊测试覆盖更多的代码。例如，对 HTTP 协议来说，可以考虑使用下列这些非字母数字字符：

```
!@#$%^&*()-_=+{}|\;:'",<.>/?~`
```

在上面这个列表中，你能认出多少可以作为 HTTP 字段分隔符的字符？作为参考，请看下面这个典型的 HTTP 服务器响应数据：

```
HTTP 1.1 200 OK
Date: Sun, 01 Oct 2006 22:46:57 GMT
Server: Apache
X-Powered-By: PHP/5.1.4-p10-gentoo
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, post-check=0, pre-check=0
Pragma: no-cache
Keep-Alive: timeout=15, max=93
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=ISO-8859-1
```

你能注意到的第一个模式是响应中有许多由换行分隔符分隔的行，换行分隔符由两个连续字符 0x0d0x0a 表示。在每个单独的行内会使用不同的分隔符。例如，在第一行中，空格字符（），斜线字符（/）和点字符（.）被用来分隔响应码。在第一行之后的所有行内，我们能看到一个冒号（:）分隔符被用来分隔指示词（directives）如 Content-Type，Server，Date 和它们的值。进一步检查的话，读者会发现逗号（,），等号（=），分号（;）

和短横线（-）同样也被用作分隔字段的分隔符。

对我们而言，当生成模糊字符串时，包含长短各异的，用以上列出的各种字段分隔符进行分隔的字符串非常重要。此外，增加分隔符的长度也同样重要。例如，考虑在2003 年发现的一个关键的 Sendmail 协议头处理安全漏洞[21]。通过由<>组成的长字符串可以利用该安全漏洞，触发可被利用的内存破坏。下面给出了一段代码，该段代码包含了一个处理重复的字符串分隔符时的漏洞：

```
voidparse(char *inbuf)
{
  charcpy[16];
  char *cursor;
  char *delim_index;
  int length = 0;

  for(cursor=inbuf; *cursor; cursor++)
  {
    if(*cursor == ':')
      delim_index = cursor;
    else
      length++;
  }

  // -2 是因为需要包含字符串结束符和冒号分隔符
  if(length <sizeof(cpy) - 2)
    strcpy(cpy, inbuf);
}
```

这段有漏洞的程序处理一个输入的字符串，期望找到一个分隔符（冒号）。如果找到分隔符，指向该分隔符的指针就被记录下来。否则，length 变量增加 1。通过计数得到的 length 是需要拷贝的源字符串的长度（不包括分隔符），此外，还需要为字符串终止符 NULL 和分隔符保留两个位置。因此，循环结束后需要检查目标字符缓冲区的大小以确保后续的 strcpy()函数调用有足够的空间。这段代码逻辑能够正确处理 name:pedramamini 这样的字符串。对上面这个的字符串，这段代码能够计算得到 length 的值为 16，发现为输入字符串保留的空间不足而放弃 strcpy()的调用。但如果输入的字符串是 name::::::::::::::::::::::pedram 呢？当给定以上输入时，length 的值为 10，因此 length <sizeof(cpy) – 2 的判定为 true，所以 strcpy()调用一定会发生。然而，当 strcpy()调用发

[21] http://xforce.iss.net/xforce/alerts/id/advise142

生时，strcpy()的源字符串的实际长度为 32，因此这段代码会触发一个栈溢出。

6.4.4　格式字符串

相对而言，格式字符串导致的安全问题是一类较容易发现的缺陷，模糊测试器生成的数据中应当包含这类数据。格式字符串安全漏洞可能发生在任何格式字符串标记上，例如%d 标记和%08x 标记就是两个可能的标记。%d 标记用于显示一个 10 进制的数字，而%08x 标记用于显示一个十六进制的数字。尽管如此，但在模糊测试中更好的选择是使用%s 或%n 标记（也可以两者都用）来发现漏洞，因为%s 和%n 标记更易于触发可被检测到的错误（如内存违规访问）。大多数格式字符串标记会导致从读取栈内存数据，通常这种读取不大可能在有漏洞的代码中触发故障。%s 标记会导致搜索字符串结束符 NULL，此时发生的栈解引用（dereferenced）导致大量的内存读取操作，另外，在大多数情况下%n 标记提供了触发故障的最大可能。因此，我们的模糊试探值列表应该包含带%s%n 的长序列。

利用格式化字符串漏洞的关键

所有格式化字符串标记，包括%d，%x 和%s，都会导致从栈中进行内存读取，但只有含有%n 的格式字符串会导致对内存进行写入操作。而能够执行内存写入操作是利用格式字符串安全漏洞来执行代码的关键需求。其他格式字符串标记可被用于从存在漏洞的软件中“泄露”关键信息，而%n 是唯一可被利用的，能够直接通过格式字符串进行内存写操作的。这个原因导致微软决定实现一个允许切换 printf 函数族对%n 格式化字符串支持的控制 API。负责切换控制的 API 是 is_set_printf_count_output()[22]，向该 API 传入一个非零的数值能够打开 printf 函数对%n 标记的支持，而给定零作为参数将会关闭 printf 函数对%n 标记的支持。事实上，在产品代码中很少出现%n，以至于微软将 printf 函数默认设置为不支持%n 标记。

6.4.5　字符翻译

另一个需要注意的问题是字符转换与翻译，特别是字符扩展（character expansion）。例如，十六进制值 0xFE 和 0xFF 在 UTF16 下会被扩展成 4 个字符。代码中对字符扩展的不适当处理通常会导致安全漏洞。

---

[22] http://msdn2.microsoft.com/en-us/library/ms175782.aspx

字符转换（character conversation）也可能会被不正确的实现，尤其是在处理少见或是很少被使用的边界值时。例如，微软 IE 浏览器在将 UTF-8 编码转换为 Unicode 编码时就遇到了这样的问题[23]。IE 中这个问题的核心原因是，当转换函数计算需要动态分配用来存储转换后的数据的内存块大小时，没有正确处理长度为 5 字节和 6 字节的 UTF-8 字符。然而，代码中使用的数据拷贝函数却正确处理了长度为 5 字节和 6 字节的 UTF-8 字符，因此就导致了基于堆的缓冲区溢出。所以，我们的模糊试探值列表也应该包含这些数据和另一些类似的恶意字符序列。

6.4.6 目录遍历

目录遍历安全漏洞以相似的方式影响网络后台程序和 Web 应用。一个常见的误解是认为目录遍历安全漏洞仅存在于 Web 应用中。目录遍历安全漏洞的确在 Web 应用中多发，但是，在私有网络协议和其他应用中也同样会出现针对目录遍历漏洞的攻击。根据 2006 年 Mitre CVE 的统计，尽管随着时间的推移，目录遍历安全漏洞稍有减少，但它仍然是软件应用中被发现的第 5 大类安全漏洞[24]。而 Mitre CVE 的统计数据包括了 Web 应用和传统的客户端/服务器应用。在开源安全漏洞数据库（Open Source Vulnerability Database，OSVDB）中可以看到过去多年来发生的许多目录遍历安全漏洞的例子[25]。

以 Computer Associates BrightStorARCserver 备份软件为例。BrightStor 通过 calloggerd 后台程序提供了一个 TCP 协议层上的客户登录接口。虽然这个接口没有被文档化，但通过基本的数据包分析能够发现，日志文件的名字实际上是在网络数据流中指定的。通过目录遍历修饰符（directory traversal modifiers）在输入的文件名前加上前缀就能允许攻击者将信息写入到指定的任意文件中。如果日志记录后台应用是以超级用户权限运行的，那么这个安全漏洞就会导致大问题。例如，在 UNIX 系统上，通过这个漏洞，攻击者可以向/etc/passwd 文件中增加一个新的超级用户。直到写作本书时，仍有一个还没有发布安全补丁的该类安全漏洞存留。所以，我们的模糊试探值列表应该包含诸如../../和..\..\这样的目录遍历修饰符。

---

[23] http://ww.zerodayinitiative.com/advisories/ZDI-06-017.html

[24] http://cwe.mitre.org/documents/vuln-trends.html#table1

[25] http://www.osvdb.com/searchdb.php?text=directory+travesal

6.4.7　命令注入

与目录遍历安全漏洞类似，命令注入安全漏洞典型的与 Web 应用，更具体的说，与 CGI 脚本联系在一起。再次重申，认为这类缺陷只与 Web 应用有关是一个常见的误解，这类缺陷能够通过公开的和私有的协议影响网络后台应用。任何应用，无论是 Web 应用还是网络后台程序，只要会向 exec()或是 system()等 API 传递未经过滤的，或是不恰当过滤的用户数据，都潜在的包含有命令注入安全漏洞。考虑下面这段简单的 Python 代码：

```
directory = socket.recv(1024)
listing = os.system("ls /" + directory)
socket.send(listing)
```

通常环境下服务器会接收到系统路径，该路径下的文件列表被获取到并返回给客户端。然而，由于代码中缺乏对输入的过滤，输入中包含的特定字符会导致允许额外命令执行。在 UNIX 系统上这些特定字符包括：&&, ; 和 | 。例如，如果将字符串“var/lib ; rm –rf /”作为参数传递给这段代码，执行的命令将会是 ls var/lib ; rm –rf /，显然，如果系统管理员执行了这个命令，会带来严重的后果。因此，我们的模糊测试试探值列表中也应该包含这些特定字符。

6.5　小结

在本章开头，我们讨论了自动化的必要性，给出了可以用于简化自动化模糊测试工具开发的库和工具的列表及其简要描述。我们将会在后续章节进一步解释其中某些库，并会在第二部分和第三部分的自定义工具开发中使用这些库。本章讨论的核心概念是针对数字、字符串、二进制序列的智能和高效的模糊值选择。这些知识将会在后续章节中得到应用并加以扩展。

在后续章节中，我们会讨论多种模糊测试目标，包括 Web 应用、有特权的命令行应用、网络服务等。当阅读后续这些章节时，请牢记本章讨论的概念。建议读者把本章描述的能够帮助开发模糊测试器的这些库和工具都记录下来。思考一下这里讨论的智能的模糊数据，以及将来会向模糊数据列表中添加哪些值。

第7章

环境变量与参数模糊测试

"This foreign policy stuff is a little frustrating."

——George W. Bush, as quoted by the New York Daily News, April 23, 2002

本地模糊测试或许是最简单的模糊测试类型。虽然利用远程安全漏洞和客户端安全漏洞可以获得更让人印象深刻的攻击结果，但本地特权提升（local privilege ）仍然是一个重要课题。即使通过远程攻击获得了目标机器的访问权，通常仍然需要使用本地攻击方法作为第二轮攻击手段，以此获取所需要的权限。

7.1 本地模糊测试介绍

用户可以通过两种主要方式将变量引入到程序中。除了通过标准输入设备，如键盘与程序进行交互外，命令行参数和进程环境变量同样可以作为输入。我们首先采用命令行参数作为模糊测试的输入。

7.1.1 命令行参数

除了最纯粹的 Windows 用户外，其他的用户都或多或少遇到过需要执行带命令行参数的程序。命令行参数被传入程序，并通过 C 语言的 main 函数中声明的 argv 指针来寻址。除 argv 外，变量 argc 也会被传入 main 函数，argc 的值为传入程序的参数的个数加 1，加 1 的原因是因为程序的名称也会被计入该值。让我们看几个简单的例子。

```
int main(intargc, char *argv[])
{
int ix;
for (ix = 0; ix<argc; ix++)
   printf("argv[%d] == %s\n", ix, argv[ix];
}
```

当用不同参数运行该程序若干次后，可以得到图 7.1 所示的结果。

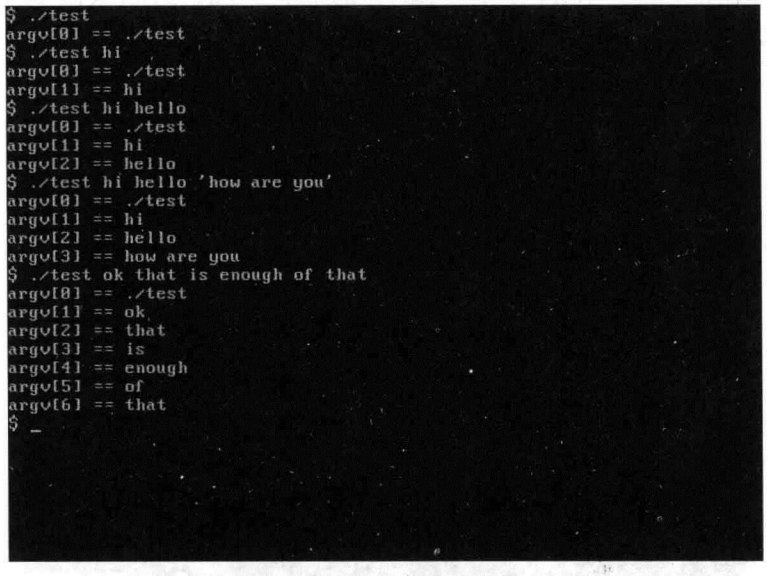


图 7.1　展示命令行参数是如何被存储的例子

7.1.2　环境变量

另一个将变量引入进程的方法是使用环境变量。每个进程都包含由环境变量组成的，被称为“环境（environ）”的东西。环境变量用来定义应用行为的全局值。用户可以设置（set）或是取消（unset）环境变量，但通常，系统中的环境变量都会被软件安装包或管理员设置为标准值。大多数命令解释器（command interpreters）会让新进程继承当前环境。Windows 下的 command.com 就是一个命令解释器，而 UNIX 系统则有多个命令处理器，例如 sh，csh，ksh，bash 等。

常用的环境变量包括 HOME，PATH，PS1 和 USER。这些环境变量的值分别表示用户的 home 目录，当前的可执行程序搜索路径，命令提示符，以及当前用户的用户名。

以上这些特定的环境变量都是 UNIX 系统的标准环境变量。除此之外，另一类常用的环境变量，包括由软件厂商提供的软件创建的环境变量，则仅在特定的应用程序中使用。当应用需要某个特定环境变量的值时，它只需简单地使用 getenv 函数就能获得该变量的值，getenv 函数的参数是需要取值的环境变量的名字。虽然 Windows 进程和 UNIX 应用一样有环境的概念，但本章的讨论主要集中在 UNIX 上，这是因为 Windows 没有 setuid 应用（setuid applications）的概念，setuid 应用是一类特殊的应用，可以由非特权用户执行，并在执行过程中获得特权。图 7.2 展示了一个典型的 UNIX 环境的样子。在 bash shell 中敲入 set 命令即可查看当前的 shell 环境。

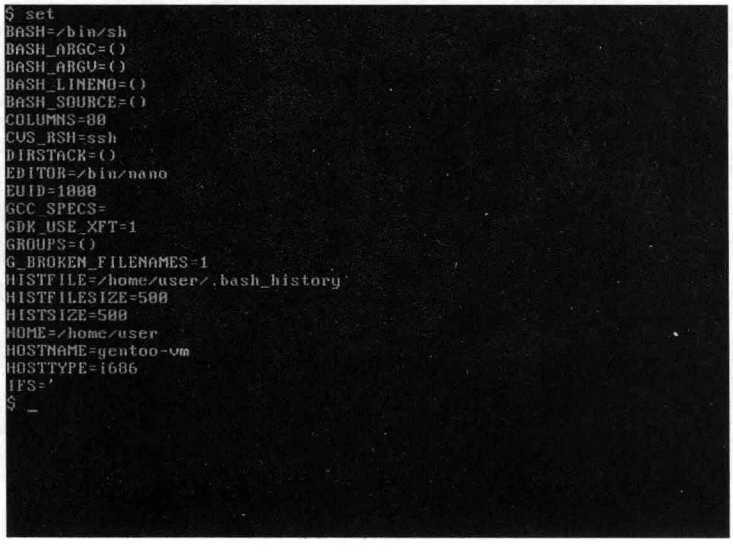


图 7.2　bash shell 使用的一些环境变量的例子

92

使用 export 命令可以操作图 7.2 中列出的每个环境变量。在了解了如何使用命令行参数和环境变量后，现在，我们可以开始讨论对环境变量和参数进行模糊测试的基本原则了。

7.2　本地模糊测试原则

环境变量模糊测试和命令行模糊测试背后的思想很简单：如果一个环境变量或是命令行选项包含一个非预期的值，应用接收到这个值后会如何响应？当然，我们仅对有异常行为的具有特权的应用感兴趣。这是因为要执行本地模糊测试，首先执行者就已经有

了访问机器的权限。在这种情况下，简单的导致一个应用崩溃并没有多大价值——就好像对本机发起拒绝服务（Denial-Of-Service，DOS）攻击一样没有意义。如果系统由多个用户共享，而某个环境变量中存在一个能够导致系统或是一个共享应用崩溃的溢出，这意味着系统有一定的风险。但我们最感兴趣的，还是在一个具有特权的应用中找到允许普通用户提升权限的缓冲区溢出漏洞。稍后的 7.3 小节“寻找测试目标”中将会讨论如何找到具有特权的，可以被用户当作目标的应用。

许多应用都采用了这样的设计：从用户处接受命令行参数，随后使用用户输入的数据来决定下一步的行动。我们以在几乎所有 UNIX 系统上都存在的‘su’应用程序来说明这个问题。当用户不带命令行参数启动这个应用时，该应用假设用户想要鉴权为 root 用户；但如果用户指定了特定用户名作为第一个参数，应用就能够理解用户想要切换到给定用户而不是 root 用户。

考虑下面的 C 语言代码，这段简化的代码展示了 su 命令如何根据不同的参数来采取不同的行为。

```
int main(intargc, char *argv[])
{
  […]
if (argc> 1)
    become(argv[1]);
else
    become(“root”);
  […]
}
```

命令行参数和环境变量是两种将变量引入到程序中的基本方法。对命令行参数和环境变量进行模糊测试的基础思想很简单，那就是：当我们从命令行传递有问题的数据给应用的时候会发生什么？由此产生的应用的行为会导致安全风险吗？

7.3　寻找测试目标

93

在进行本地模糊测试时，系统中通常只有少数值得进行模糊测试的目标。值得进行模糊测试的程序是那些在执行时具有高特权的程序。在基于 UNIX 的系统上，很容易识别出这些程序，因为它们共同的特点是程序的 setuid 或是 setgid 位被置位了。

setuid 和 setgid 位表明当一个程序运行时，程序能够要求提升权限。以 setuid 位为例，setuid 位被置位的程序在执行时将具有文件拥有者的权限，而不是运行该程序的用

户的权限。如果 setgid 位被置位，该程序的进程将会具有组拥有者的权限。例如，成功地利用设置了 setuid 位或是 setgid 位，且所有者为 root 的程序可以获得一个具有 root 权限的 shell。

使用 find 命令可以很容易地得到设置了 setuid 位的二进制程序的列表，find 是一个 UNIX 和类 UNIX 操作系统的标准命令。以下 find 命令能够列出系统中所有设置了 setuid 位的二进制程序。请注意，该命令需要以 root 权限运行，否则由于权限问题，会出现读取文件系统的错误。

```
find / -type f -perm -4000 -o -perm -2000
```

find 命令是一个强大的工具，能够用来找到特定类型的文件、设备，或是文件系统中的特定目录。在上面的例子中，我们仅使用了 find 命令所支持选项中的少数。上面命令的第一个参数表明我们需要搜索根目录（/）下的所有东西；type 选项告诉 find 命令我们只关心文件——这意味着我们不关心符号链接、目录，或是设备；-perm 选项说明了我们感兴趣的权限；-o 选项用来允许 find 命令使用“或”逻辑。如果一个二进制文件的 setgid 位或是 setuid 位被置位，逻辑表达式“-perm -4000 -o -perm -2000”的值就会为 true，这样该文件的路径就会被打印出来。简而言之，上述命令将会找到所有设置了 setuid（4）或是 setgid（2）位的常规文件。下面列出了在默认安装的 Fedora Core 4 下运行该命令的结果：

```
[root@localhost /]# find / -type f -perm -4000 -o -perm -2000
/bin/traceroute6
/bin/traceroute
/bin/mount
/bin/su
/bin/ping6
/bin/ping
/bin/umount
/usr/bin/lppassword
/usr/bin/gtali
/usr/bin/wall
/usr/bin/chsh
/usr/bin/passwd
/usr/bin/glines
/usr/bin/gnibbles
/usr/bin/at
/usr/bin/gnotravex
/usr/bin/gnobots2
/usr/bin/sudo
```

```
/usr/bin/same-gnome
/usr/bin/gataxx
/usr/bin/rcp
/usr/bin/mahjongg
/usr/bin/iagno
/usr/bin/rlogin
/usr/bin/gnotski
/usr/bin/chage
/usr/bin/lockfile
/usr/bin/write
/usr/bin/gpasswd
/usr/bin/ssh-agent
/usr/bin/crontab
/usr/bin/gnomine
/usr/bin/sudoedit
/usr/bin/chfn
/usr/bin/slocate
/usr/bin/newgrp
/usr/bin/rsh
/usr/X11R6/bin/Xorg
/usr/lib/vte/gnome-pty-helper
/usr/libexec/openssh/ssh-keysign
/usr/sbin/userhelper
/usr/sbin/userisdnctl
/usr/sbin/sendmail.sendmail
/usr/sbin/usernetctl
/usr/sbin/lockdev
/usr/sbin/utempter
/sbin/pam_timestamp_check
/sbin/netreport
/sbin/unix_chkpwd
/sbin/pwdb_chkpwd
```

7.3.1　UNIX 文件权限释义

UNIX 系统中的文件权限模型由三种不同的类型的基础访问构成：读（read），写（write）和执行（execute）。每个文件有三组权限。这三组权限分别对应于用户、组、其他。在任何时候，只有一组权限会产生效果。例如，如果你拥有某个文件，你对该文件的访问权限由用户权限决定。如果你不是这个文件的拥有者，但是你属于该文件的拥有者组，那么你对该文件的访问权限由该文件的组权限决定。在其他情况下，你对文件的访问权限将由最后一组访问权限决定。例如下面这个例子：

```
-r-x--x---   2   dude    staff        2048    Jan 2   2002 File
```

在这个例子中，用户 dude 拥有该文件。dude 拥有读和执行该文件的权限。当然，由于用户 dude 拥有这个文件，该用户可以随时修改文件的权限。

如果 staff 组中的其他用户尝试访问这个文件，他将不能读文件的内容，而只能执行这个文件。对这个文件的读操作会由于无效权限而失败。最后，除 dude 用户和包含在 staff 组的用户外，其他用户将不能访问该文件，因为其他用户没有该文件的读、写和执行权限。

在 UNIX 中，存在一种特殊的描述文件权限的方式。在这种方式中，权限以八进制形式表示。也就是说，每个权限组由一个 0 到 7 的数值表示。读权限的标记是八进制值 4，写权限的标记是八进制值 2，执行权限的标记是八进制的 1。这三个标记的值加起来表示总的权限。例如，一个允许用户，组，以及其他用户进行读、写操作的文件的权限可以表示为 666。在上面的例子中，dude 用户拥有的这个文件的权限可以被表示为 510：用户权限为 5 = 读（4）+ 执行（1），组权限为 1 = 执行（1），其他权限为 0，意味着其他用户没有任何权限操作该文件。

除了以上三组权限外，第四组权限表示一些特殊的标志位，如 setuid 和 setgid 位。setuid 位用 4 表示，setgid 位用 2 表示。因此，一个设置了 setuid 和 setgid 位的文件的权限可能是 6755。如果文件的特殊标志位没有被置位，第四组权限的值就为 0，也就意味着该文件没有扩展权限。

7.4 本地模糊测试方法

环境变量和命令行参数可以由用户很方便地提供，因为它们几乎都是简单的 ASCII 字符串，因此在实际操作中可以采用基础的手工测试方法对他们进行模糊测试。最简单的测试也许是设置 HOME 变量为一个长字符串，然后运行被测目标程序来看看会发生什么。利用 Perl 可以快速的实现这个测试，而 Perl 在大多数 Unix 系统上都是默认可用的。

96

```
HOME=`perl-e 'print "X"x10000'` /usr/bin/target
```

这是一种非常基本的测试应用的方法，目的是看应用程序是否能够处理一个长的 HOME 变量。然而，这个例子有价值的前提是你已经知道被测应用使用了 HOME 变量。如果你不知道被测应用会使用哪些变量，那该怎么办？怎样才能知道被测应用使用了哪

些变量呢？

7.5　枚举环境变量

至少可以使用两种自动化方法来确定一个程序使用的环境变量。第一种方法需要使用库预加载（library preloading）特性。如果系统支持库预加载，就可以使用钩子（hook）勾住 getenv 调用。然后提供一个新的 getenv 函数执行标准的 getenv 功能，同时将调用记入文件日志，这种方法能够有效地记录应用请求的所有环境变量。在本章 7.6 节“自动化的环境变量模糊测试”中将详细描述这种方法的扩展应用。

7.5.1　GNU 调试器（GNU Debug，GDB）法

另一种确定程序使用的环境变量的方法需要调试器的支持。通过使用 GDB，你可以在 getenv 函数中设置一个断点，并输出函数被调用时的第一个参数。在 Solaris 10 上使用 GDB 脚本来自动化该过程的示例如下所示：

```
(08:55AM) [user@unknown:~]$gdb -q /usr/bin/id
(no debugging symbols found) . . . (gdb)
(gdb) beak getenv
Function "getenv" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (getenv) pending
(gdb) commands
Type commands for when breakpoint 1 is hit, one per line
End with a line saying just "end".
>silent
>x/s $i0
>cont
>end
(gdb) r
Starting program: /usr/bin/id
[. . .]
Breakpoint 2 at 0xff2c4610
Pending breakpoint "getenv" resolved
(no debugging symbols found) . . .
0xff0a9064:       "LIBCTF_DECOMPRESSOR"
0xff0a9078:       "LIBCTF_DEBUG"
0xff24b940:       "LIBPROC_DEBUG"
0xff351940:       "LC_ALL"
```

```
0xff351948:       "LANG"
0xff3518d8:       "LC_CTYPE"
0xff3518e4:       "LC_NUMERIC"
0xff3518f0:       "LC_TIME"
0xff3518f8:       "LC_COLLATE"
0xff351904:       "LC_MONETARY"
0xff351910:       "LC_MESSAGES"
uid=100(user) gid=1(other)

Program existed normally.
(gdb)
```

如果你对 GDB 所使用的命令不熟悉，这里用几句话简单地总结之：

- **break 命令用于在特定的函数或地址上设置断点**。在上面的例子里，我们使用 break 命令让程序执行时停止到对 getenv 的调用。
- **commands 命令指定当执行到断点时所需要执行的特定操作**。在上面的例子中，我们告诉 GDB 执行 silent 命令，并通过 x/s 命令以字符串形式打印出 i0 寄存器的值。在 SPARC 上，i0 寄存器存储了被调用函数的第一个参数的值。
- **下一个命令是继续执行程序，有了这条命令，我们就不用手工告诉 GDB 在中断后继续执行**。接下来我们使用 run 命令的快捷方式（命令“r”）开始执行程序。

通过这种方法，我们立即能够看到/usr/bin/id 程序需要访问的 11 个环境变量值的列表。注意这种方法在所有系统上都适用，然而，由于不同架构上有不同的寄存器定义方式，因此，在不同平台上，可能需要改变解引用（dereference）的寄存器的名字。例如，在 x86 平台上需要打印的是$eax 的值，在 SPARC 上是$i0 寄存器，而在 MIPS 上则是$a0。

现在，我们已经掌握了确定被测应用所使用的环境变量的方法，接下来，我们将研究在更自动化的方式下对应用使用的环境变量进行模糊测试的方法。

7.6 自动化的环境变量模糊测试

在上一节“枚举环境变量”中我们曾简要提及了库预加载，库预加载可以应用在自动化的模糊测试中。程序要获得环境变量的值，就必然要调用 getenv 函数。如果我们能勾住 getenv 函数并针对每次的调用返回一个长字符串，那我们甚至都不需要知道程序使用了哪些环境变量。采用这种方法，我们能够轻松地通过拦截所有的 getenv 调用来对每一个程序进行模糊测试。当需要快速检查不安全的环境变量使用时，这种方法非

常有用。

以下是 getenv 函数的一个简化实现。它使用了全局变量 environ，该全局变量指向环境的起点。下面的代码简单地遍历环境数组，检查调用者请求的值是否存在于环境数组内。如果存在，该函数将返回一个指向该环境变量值的指针；如果不存在，该函数则返回 NULL 指针，表明该环境变量未被设置。

```
extern char **environ;
char *getenv(char *variable)
{
int ix = 0;
while (environ[ix])
  {
  if ( ! (strncmp(string, environ[ix], strlen(string))) &&
    (environ[ix][strlen(string)] == '=') )
    {
     printf("%s\n", environ[ix]+strlen(string)+1);
     return environ[ix]+strlen(string)+1;
    }
ix++;
}
```

7.6.1　库预加载（Library Preloading）

接下来我们要讨论的主题是库预加载。库预加载使用操作系统的链接器（linker），将特定函数替换为用户自定义函数，从而简单地实现“钩子”。虽然库预加载在不同系统上有不同的具体实现方式，但无论在哪种系统上，库预加载的基本概念是一致的。典型的库预加载应用方式如下：用户将一个特定环境变量的值设置为指向用户提供的库的路径，这样，当程序执行时，用户提供的库就会被加载。如果用户提供的库中包含了与程序使用的符号（symbols）相同的符号，则程序会使用用户提供库中的符号，而不是使用程序中的符号。在这里，我们提到的符号主要指函数。举例来说，如果用户构建了一个库，该库包含名为 strcpy 的函数，并在执行程序时预加载该库，被执行的程序就会调用用户版本的 strcpy，而不是调用系统版本的 strcpy。库预加载技术可以用在许多地方，例如使用该技术对调用进行包装以进行调优（profiling）和审计，另外，这种技术也可用来查找安全漏洞。在对环境变量进行模糊测试时，考虑包装或是完全替换掉 getenv 函数，因为 getenv 是用来请求环境变量值的函数。

下面的函数是一个简单的 getenv 函数的替代品，可以用来发现简单的由长字符串

引起的安全漏洞。通过使用库预加载技术，可以强制使用该函数覆盖真正的 getenv 函数。

```
#define BUFFSIZE 20000
char *getenv( char *variable)
{
  char buff[BUFFSIZE];
  memset(buff, 'A', BUFFSIZE);
  buff[BUFFSIZE-1] = 0x0;
  return buff;
}
```

很容易看出，对每一个环境变量的取值请求，该函数都返回一个长字符串。因为我们并不关心所返回数据的正确性，所以该函数完全没有使用到环境数组。

Dave Aitel 的 GPL sharefuzz 工具使用了库预加载技术，该工具已被用来在 setuid 应用中找到了大量的安全漏洞。要开始进行该类模糊测试，只需编译以上 C 语言代码到一个共享库中，并使用操作系统的预加载功能（假设你的操作系统有预加载功能）。以 Linux 为例，通过以下两个步骤可以使用预加载方法：

```
gcc -shared -fPIC -o my_getenv.so my_getenv.c
LD_PRELOAD=./my_getenv.so /usr/bin/target
```

当执行/usr/bin/下的被测应用时，所有对 getenv 的调用都会使用我们修改过的 getenv 函数。

7.7 检测问题

现在读者已经熟悉了本地模糊测试的基本方法，下一步读者需要知道当被测应用中发生我们感兴趣的不正常行为时，如何识别出这些行为。在许多情况下，被测应用中的不正常行为表现得很明显：例如，程序崩溃并打印出“Segmentation fault”或是其他的致命信号（fatal signal）信息。

100

然而，我们最终的目标是自动化，因此不能依赖用户人工识别程序的崩溃。为了达成我们的目标，我们需要一种可靠的依靠程序识别问题的方法。至少有两种方法可用于自动地识别问题。最简单的识别问题的方法是检查应用的返回值。在现代 UNIX 和 Linux 系统中，如果应用因为一个未被处理的信号而终止，shell 返回码应该等于 128 加上信号的值。例如，段错误（segmentation fault）会导致 shell 收到的返回码为十进制的 139，

因为 SIGSEGV（段错误信号）的值是 11。如果程序由于非法指令终止，shell 收到的返回值为 132，因为 SIGILL（非法指令信号）的值为 4。因此，采用这种方法自动识别错误的逻辑很简单：如果应用的 shell 返回码是 132 或是 139，就意味着一个我们可能会感兴趣的崩溃。

读者可能也会考虑中止信号。由于新版本的 glibc 引入了更严格的堆检查，因此 SIGABRT（中止信号）也是让人感兴趣的信号。SIGABRT 是一个可能导致进程终止或是转储核心（dump core）的信号。虽然在特定情况下进程会在堆破坏时中止，但有更聪明的方法来避开它。

如果你完全用 shell 脚本来做模糊测试，使用 shell 返回值来识别程序错误是可行的。然而，如果你在使用 C 语言或是其他语言写的模糊测试器，也许需要使用 wait 或是 waitpid 函数。在这种情况下，一般的本地模糊测试方法是使用 fork，然后紧接着在子进程中使用 execve，在父进程中使用 wait 或是 waitpid。如果使用方法正确，就可以通过检查 wait 或是 waitpid 返回的状态轻松地判断子进程是否崩溃。在下一章中，我们会通过一个来自 iFUZZ 工具的简化的代码片段展示该方法的应用。

如果你希望捕获被应用处理的信号（因为这些信号是被应用而不是系统处理，所以用上面的方法无法发现），除了使用钩子勾住信号子例程外，至少还有一个替代方法。这个方法就是使用系统的调试 API，将其附加到进程上，在信号处理子例程被调用前截获它。对大多数 UNIX 系统而言，可以使用 ptrace 达成这个目标。一般的方法是在父进程中依次使用 fork，ptrace 和 execve，在子进程中的循环中使用 waitpid 和 ptrace，持续监视进程的执行，并截获传入的信号。当子进程的 waitpid 返回时，就意味着程序收到了一个信号或是已经终止，此时需要检查 waitpid 返回的状态才能判断究竟是哪种情况。另外，需要明确地告知应用继续执行，并在大多数情况下继续向下传递收到的信号。这些都可以用 ptrace 完成。SPIKEfile 工具和 notSPIKEfile 工具中相关的实现能够用来作为此方法的参考。SPIKEfile 和 notSPIKEfile 工具用来进行文件模糊测试，我们将会在第 12 章“Unix 平台上的文件格式自动化模糊测试”中对其进行详细描述。此外，在下一章中，我们提供了用于展示 ptrace 方法实现地代码片段。

在多数情况下，使用 ptrace 方法进行本地模糊测试可以说是“杀鸡用牛刀”。因为只有很少的 UNIX 平台上的 setuid 应用会处理 SIGSEGV 和 SIGILL 信号。而且，一旦开始使用 ptrace，就是在引入不同操作系统和架构上不兼容的代码（因为 ptrace 在不同操作系统和不同架构上的实现很不相同）。如果你需要设计无需修改就能够在多平台上运行的应用，请考虑到这一点。

101

在下一章中，我们将会展示一个简单的命令行模糊测试器的实现，该模糊测试器被设计成可以在任何具有 C 编译器的 UNIX 系统上编译和运行。该工具同时还包含一个简单的针对 getenv 函数的共享库模糊测试器。

7.8 小结

虽然发现本地安全漏洞并不能给你带来多少荣耀，但发现一个好的权限提升漏洞仍然是有价值的。本章描述了不同的使用自动化手段发现这类安全漏洞的方法，在下一章中，我们将会具体实现这些方法中的一部分，以实际发现一些缺陷。

第8章

自动化的环境变量与参数模糊测试

"Those weapons of mass destruction have got to be somewhere!"

——George W. Bush, Washington, DC, March 24, 2004

在本章中，我们将介绍 iFUZZ 这个对本地应用进行模糊测试的工具程序。本章讨论的主要测试目标是 setuid UNIX 程序中的命令行参数和环境变量。在第 7 章“环境变量与参数模糊测试”中我们已经讨论过对命令行参数与环境变量进行模糊测试的话题。本章我们将讨论 iFUZZ 的功能，说明它的设计决策，并讨论 iFUZZ 是如何在 IBM AIX 5.3 中发现大量本地安全漏洞的。

8.1 iFUZZ 本地模糊测试器的功能

正如你猜想的那样，iFUZZ 具有几个本地模糊测试器应有的功能模块：自动处理各种目标二进制代码的引擎，能够生成 C 语言触发器的模块——触发器能够重现缺陷，以及几个不同的以模块方式实现的模糊测试方法。iFUZZ 的方便之处在于，它无需修改就能运行在几乎所有 UNIX 和类 UNIX 操作系统上。iFUZZ 工具已被用于 IRIX，HP-UX，QNX，MacOS X，AIX 等操作系统，并在这些操作系统上发现了安全漏洞。该模糊测试器的核心功能由面向不同类型模糊测试的模块组成：

104

- **argv 模糊测试器模块**：iFUZZ 工具的前两个模块非常类似，因此我们在此一并解释。iFUZZ 的前两个模块用于对可执行文件的 argv[0]和 argv[1]的值进行模糊测试。这两个模块的基础用法很简单：指定被测应用的完整路径并运行之。这两个模块会尝试使用不同长度的字符串，带有格式字符的字符串进行模糊测试。这两个模块实际使用的字符串依赖于模糊字符串数据库，最终用户可以自行补充数据库中的数据。
- **单选项/多选项模糊测试器模块**：该模块是一个“黑盒”的模糊测试器，“黑盒”的意思是说，用户不向这个模块提供任何被测应用的信息，而这个模块也不关心这些信息。该模块只是简单地针对每一个可能的选项，把字符串值传给被测应用。如果使用单选项模糊测试方式，iFUZZ 会从 a 循环到 Z 并尝试运行以下形式的命令：

```
./target -a FUZZSTRING
```

FUZZSTRING 的值从 iFUZZ 内部的模糊字符串数据库获得。单选项模糊测试是一种快速发现简单的选项相关问题的方法，但是使用这种方法不能发现复杂的问题，例如那些需要多个选项值的问题。

- **getopt 模糊测试器**：该模块需要从用户那里获得一些关于应用的信息。从该模糊测试器的名字可以看出，它需要知道应用程序通过 getopt 使用的选项字符串，因此，该模块根据它掌握的“应用使用的选项”产生作用。getopt 模糊测试是一种非常耗时的模糊测试，但是它比其他模糊测试类型更彻底。与其他模糊测试器相比，这种模块找到的复杂安全漏洞多得多。例如，假设有某个应用，该应用只有在设置了 debug，verbose 选项，且-f 参数的值为长字符串时才会发生缓冲区溢出。以下是该应用的 usage 输出：

```
$ ./sample_program
Usage:
-f <file> Input filename
-o <file> Output filename
-v Verbose output
-d Debug mode
-s Silent mode
```

105

基于应用输出的 usage 信息，我们发现这个应用的 getopt 字符串很可能是 f:o:vds。也就是说 f 和 o 选项都带有参数，而 v，d，s 选项只是开关。我们是怎么知道这些的？

答案是，我们的知识来源于 getopt 的 man 页面[1]：

> 选项参数是指明了合法的应用能接受的参数字符的字符串。如果字符串中的某个选项字符后面跟着一个冒号（“:”），说明该选项需要参数。如果某个选项字符后面跟着两个冒号（“::”），说明它的参数是可选的；这是一个 GNU 扩展。

如果我们在 getopt 模糊测试模式下运行 iFUZZ，且给定参数为 f:o:vds，用不了多长时间 iFUZZ 就会找到前面描述的安全漏洞。因为 iFUZZ 能够将已存在文件的路径用作模糊字符串，你甚至能够发现那些需要其中一个选项参数指向合法文件的安全漏洞。基于 iFUZZ 的设计，创造性地使用该工具并不困难。你可以向字符串数据库中加入其他一些合法字符串，例如用户名，主机名，合法的 XServer 目标字符串等。创造无止境——你越有创造性，你的模糊测试就越全面。

iFUZZ 还包含一个简单的可预加载的 getenv 模糊测试器。该模糊测试器包含一个环境变量列表，模糊测试器不拦截列表中的环境变量，但不在列表中的环境变量会被模糊测试器替换成其他字符串。这个模糊测试器是模仿 Dave Aitel 的 sharefuzz 工具的粗糙实现，带有一点点额外功能，允许从真实环境中返回数据。这个工具不是 iFUZZ 的核心组件，只是一个快捷工具。

iFUZZ 包含的另一个组件是个简单的 getopt 钩子，该组件能够被预加载，从目标二进制中转储 getopt 选项字符串。这个工具其实是一个仅一行的 C 语言程序，为了便于使用而被包含在 iFUZZ 工具中。如果你想使用这个组件，请谨记某些应用会用它们自定义的函数解析命令行选项。对这类应用来说，必须基于 usage 输出手工构建 getopt 字符串。

8.2　开发 iFUZZ 工具

我们已经阐述了 iFUZZ 工具的功能，现在让我们来看看 iFUZZ 工具的实现细节。如果想要全面理解 iFUZZ 是如何构建的，建议读者从 fuzzing.org 网站下载和查看该工具的源代码。在这里我们只着重指出 iFUZZ 工具的一些关键特性，并讨论该工具中的具体设计决策。

---

[1] 译者注：Linux/Unix 程序都有相应的 man 页面，通过 man [程序名称] 就能得到特定程序的 man 页面。

8.2.1 开发方法

iFUZZ 的开发思路是让系统模块化和可扩展。采用这种设计可以很容易地确定
106 iFUZZ 的基本特性，而无需花费太多时间考虑如何实现各种不同的模糊测试方法。完成主引擎的开发后，才需要考虑向其中加入模块。

开发完基础引擎和辅助功能后，随后需要开发的模块是 argv[0]模糊测试器。该模块使用一个可以在多个操作系统（包括 QNX，Linux 和 AIX）上使用的测试套件，套件包含几个已知存在 argv[0]溢出漏洞和格式字符串漏洞的二进制程序，以及一些没有安全漏洞的应用。这个测试套件可以用来测试模糊测试器，确保模糊测试器的准确性。

作为开发者，你不一定知道测试套件中的哪些程序存在已知的安全漏洞，因此你可以考虑自行创建一些有安全漏洞的应用。例如，下面是一个最简单的，带有 argv[0]格式字符串安全漏洞的程序：

```
int main(intargc, char *argv[])
{
  if(argc> 1) print(argv[1]);
  exit(0);
}
```

上面这段代码编译成二进制程序后，可以扔到一个包含某些无漏洞的二进制程序组成的测试套件中，用来检测漏报和误报。

以格式字符串作为攻击输入

格式字符串安全漏洞已经存在有一段时间了，人们一度认为它们无伤大雅。1999 年，在对 proftpd1.2.opre6 进行安全审计之后，TymmTwillman 在 BugTraq 邮件组发布了一个利用格式字符串安全漏洞进行攻击的细节[2]，该安全漏洞允许攻击者利用漏洞实现内存覆盖。以上描述的安全漏洞由 snprintf()函数导致，当向该函数传入不带格式字符串的用户输入时就会触发该漏洞[3]。

107 接下来要实现的模块是单选项模糊测试器。虽然单选项模糊测试器的思想很简单，但从应用在商业 UNIX 系统上的结果来看，这种方法是最有效的模糊测试方法之一。iFUZZ

[2] http://seclists.org/bugtraq/1999/Sep/0328.html

[3] http://cn.wikipedia.org/wiki/Format_string_attack

中这个模块的设计很简单，没有什么复杂的地方。在 iFUZZ 中，我们使用一个循环遍历所有的字母，每次使用循环中的当前字符作为程序的选项。选项参数则从模糊测试数据库中取得。

显然，有了这些简单模块后，iFUZZ 已经能够找到一些安全漏洞了，但仅依赖这些简单的模块，没法发现更复杂的安全漏洞，例如，需要多个选项才能触发的安全漏洞。因此，接下来我们在 iFUZZ 中实现的两个模块是：多选项模糊测试器和 getopt 模糊测试器。

多选项模糊测试器的设计很简单，它基本上就是在单选项模糊测试循环中嵌套单选项模糊测试器。每次测试使用的最大选项数由命令行指定。指定的选项数量越多，嵌套循环的层数就会越多。

为了得到一个更有效和更高效的多选项模糊测试器，getopt 模糊测试诞生了。除了从特定的参数列表中获取参数外，getopt 模糊测试器与多选项模糊测试器基本一致。虽然 getopt 技术减小了模糊测试的覆盖，但在这种方式下进行完全的模糊测试通常更快速，并且经常能够更快地找到安全漏洞。

我们设计了两种基本的捕获异常的方法。第一种方法是，不捕获已被应用处理的信号，但正如我们在上一章中指出的，极少的 UNIX 应用会处理我们想要捕获的信号。

Fork，Execute 和 Wait 方法

下面给出了一个简单的通过 fork，execute 和 wait 实现的异常捕获方法：

```
[…]
if((pid = fork()) != 0)
       {
           child = pid;
           waitpid(pid, &status, 0);
       if (WIFSIGNALED (status))
         {
         switch (WTERMSIG (status))
             {
             case SIGBUS:
             case SIGILL:
             case SIGABRT:
             case SIGSEGV:
                        fprintf (stderr, "CRASH ON SIGNAL #%d\n", WTERMSIG
(status));
                        break;
```

```
                default:
                            break;
              }
             }
            }
           else /* child */
            {
              execle ("/bin/program", "program", NULL, environ);
              perror ("execle");
             }
[…]
```

Fork, Ptrace/Execute,和 Wait/Ptrace 方法

下面的 C 代码片段展示了怎样复制一个进程并监视进程收到的信号，即使这些信号已经在应用内部被处理，下面的代码仍然可以监视之。下面的代码示例来自 notSPIKEfile 和 SPIKEfile 工具，是这两个工具中都包含的一部分代码的精简版，我们将会在第 12 章“Unix 平台上的文件格式自动化模糊测试”讨论 notSPIKEfile 和 SPIKEfile 这两个文件模糊测试工具。

```
[…]
       If ( !(pid = fork()) )
       { /* child */
        prace (PTRACE_TRACEME, 0, NULL, NULL);
        execve (argv[0], argv, envp);
       }
        Else
       { /* parent */
        c_pid = pid;
monitor:
            waitpid (pid, &status, 0);
            if ( WIFEXITED (status) )
            { /* program existed */
                  If ( !quiet )
                    printf ("Process %d existed with code %d\n", pid, WEXITSTATUS
(status));
                  return(ERR_OK);
            }
            else if (WIFSIGNALED (status))
            { /* program ended because of a signal */
            printf ("Process %d terminated by unhandled signal %d\n", pid, WTERMSIG
(status));
            return(ERR_OK);
```

```
                }
                else if (WIFSTOPPED (status))
                { /* program stopped because of a signal */
                     if( !quiet)
                      frpintf (stderr, "Process %d stopped due to signal %d (%s)",
pid, WSTOPSIG (status), F_signum2ascii (WSTOPSIG (status)));
                }
                 switch ( WSTOPSIG (status) )
                { /* the following signals are usually all we care about */
                     case SIGILL:
                     case SIGBUS:
                     case SIGSEGV:
                     case SIGSYS:
                     printf("Program got interesting signal… \n");
                          if ( (ptrace (PTRACE_CONT, pid, NULL,
                               (WSTOPSIG (status) == SIGTRAP) ? 0 : WSTOPSIG (status)))
== -1)
                               {
                                 perror("ptrace");
                               }
                               ptrace(PTRACE_DETACH, pid, NULL, NULL);
                               fclose(fp);
                               return(ERR_CRASH); /* it crashed */
                 }
    /* deliver the signal through and keep tracing */
                 If ( (ptrace (PTRACE_CONT, pid, NULL,
                    (WSTOPSIG (status) == SINGTRAP) ? 0 : WSTOPSIG (status))) == -1 )
                {
                     ptrace("ptrace");
                }
             goto monitor;
            }
            Return(ERR_OK);
    }
```

8.3　iFUZZ 使用的编程语言

我们选择了 C 语言来开发 iFUZZ 工具。虽然我能找出一些否定 C 语言的理由，但我得承认，C 语言是我日常使用的，感觉最舒服的语言。

当然，除了是我个人的喜好外，C 语言有一些明显的优势。譬如，大多数 UNIX 机器，甚至是老掉牙的机器，都已经预先安装了 C 编译器套件。相比之下，这些机器上

就不一定安装了 Python 或是 Ruby 等脚本语言。Perl 语言在 UNIX 机器上也相当普遍，
110 因此也是一个可选用的语言，然而，Perl 代码出了名的不好维护。

选择 Python 和 Perl 这样的语言的优势在于开发时间。脚本语言能够显著缩短开发时间。

最终，由于我们对 C 语言的偏好，以及我们希望开发出一个比那些由 bash 和各种脚本语言混合而成的简陋的小型模糊测试器更加“出色”的模糊测试器，我们最终选择了 C 语言。

8.4 案例研究

使用 iFUZZ 工具，安全测试工程师在 IBM AIX 5.3 中发现了超过 50 个本地安全漏洞，每一个发现的安全漏洞都可被渗透测试工程师和黑帽黑客（black hat hacker）[4]利用。大多数被发现的安全漏洞都隐藏在 argv[0]和 argv[1]中，因此很容易被发现。然而，有些好玩的安全漏洞则不那么容易被发现。我们说这些漏洞不那么容易被发现，并不是指测试者需要特定技能才能发现这些缺陷，而是说要发现这些漏洞，需要更耐心地准备 iFUZZ 命令行选项并等待更长的时间。下面两个存在于某 setuid 应用中的漏洞展示了 iFUZZ 的威力：

- piomkpq -A ascii -p X -d X -D x -q LONGSTRING
- piomkpq -A ascii -p LONGSTRING -d X -D X -q

由于攻击者需要拥有 printq 组访问权限才能利用这个漏洞，因此这里给出的被测应用中存在的漏洞并不会带来明显的系统安全性方面的威胁。虽然如此，我们仍以该漏洞作为示例，因为该漏洞是一个需要特殊条件才能触发的漏洞。

要使用 iFuzz 工具找到这些缺陷，首先需要阅读程序的手册页面，并据此创建 getopt 字符串。我们使用的 getopt 字符串是 a:A:d:D:p:q:Q:s:r:w:v:。

下面的 ls 命令的输出显示了该程序的权限和位置：

```
-r-sr-x---   1   root printq   32782 Dec 31 1969 /usr/lib/lpd/pio/ etc/piomkpq*
```

[4] 译者注：黑帽黑客是与白帽黑客（white hat hacker）相对应的概念，他们往往利用自身技术窃取信息，以此谋取利益。

经过一段时间的运行，iFUZZ 找出了程序中至少两个可能被利用的漏洞。在上面的例子中，LONGSTRING 指的是长度约为 20000 个字符的字符串，X 指的是任何合理的字符串，例如字符串“X”本身。X 值必须和长字符串一起使用，才能在让程序运行到存在漏洞的代码。

能够找到上面提到的，不那么容易被触发的安全漏洞是有意思的事情，看看遍历全部 iFUZZ 基础模块能找到多少简单缺陷同样是件好玩的事情。在 AIX 5.3 上，对 setuid 应用的 argv[0]，argv[1]或是单选项模式运行 iFUZZ 工具，准保能让你找到足够忙活一阵子的堆溢出，栈溢出，以及格式字符串漏洞。iFUZZ 发现了许多安全漏洞，这些漏洞被报告给了 IBM 并已被修复。AIX 5.3 中的有些漏洞是本书作者发现的，有些则是其他独立研究员发现的，但其他独立研究员很可能也是使用类似的模糊测试概念发现缺陷的。

8.5　优点与改进

读完本章后，希望你已经在 iFUZZ 中找到了一些有用的东西。同样重要的是，希望你已经发现了 iFUZZ 的一些弱点。下面列出了我们对 iFUZZ 的看法。

- 首先，iFUZZ 没有考虑系统崩溃的可能性，使用了一些硬编码的 sleep 语句让系统保持低负载。如果 iFUZZ 能够分析系统负载，合理地延长 sleep 时间，避免在重负载的情况下发生系统挂起或是误报、漏报，这将会是个很不错的特性。
- 如果 iFUZZ 提供一个选项，允许用户指定触发崩溃的字符串的大致长度，或是指定触发崩溃需要的最小的字符串长度，就更让人满意了。这个特性不那么重要，，但却能够帮助使用者节省不少时间。
- 另一个 iFUZZ 缺少的特性是自动定位操作系统中所有 setuid 和 setgid 应用的位置。该特性同样能够节省使用者的时间。
- 能够分析给定应用使用的选项会是一个极好的特性，但该特性可能不太容易实现。如果 iFUZZ 能够自动找出给定应用需要的标志和选项，我们就能得到一个需要极少用户交互的、聪明且完整的模糊测试。当然，实现该功能并不容易，因为不同的应用使用不同的选项格式。而且，即使实现了这个特性，工具也不能解析具有新格式 usage 的应用。也许可以将该特性实现为可扩展的，让使用者能够为不同的应用添加新格式的支持。
- 另一个不那么重要但却有实现价值的特性是生成 C 语言代码能力。如果发生崩溃后能够得到一个可重现攻击的 C 语言程序，那么该安全漏洞就能够立刻被重现。工具生成的 C 语言代码甚至可以作为发掘安全漏洞的基础。如果有这个特

112

性，就能大大节省漏洞开发者的时间——尤其是对一个脆弱的，总是崩溃的系统进行模糊测试时。在这种情况下，可以节省写 C 程序框架触发安全漏洞的时间。

8.6 小结

我们以往关注的重点是远程漏洞，然而，客户端安全漏洞和本地系统安全是一个充满了唾手可得的成果的领域。通过本地模糊测试器，我们可以用极小的时间代价发现这些唾手可得的成果，甚至是更复杂的安全漏洞，这证明了本地模糊测试的价值。

第9章

Web应用与服务器模糊测试

"I am the master of low expectation."

——George W. Bush, abord Air Force One, June 4, 2003

现在，我们从本地应用的模糊测试转向 C/S 架构（Client-Server architecture）应用的模糊测试，尤其是 Web 应用和 Web 服务器的模糊测试。虽然通过对 Web 应用的模糊测试也可以发现应用所在的 Web 服务器中的漏洞，但为了简便，在本书后面的部分，我们将这种类型的模糊测试仅作为 Web 应用模糊测试考虑。尽管 Web 应用与服务器模糊测试的基本概念与前面所讨论的网络模糊测试基本一致，但我们还是要指出一些区别（adjustments）。首先，Web 应用程序的输入点很多，而且这些输入点往往都不明显，因此我们需要重新定义输入向量的构成。其次，我们需要改进漏洞检测机制以便捕获和分析 Web 应用程序产生的那些可能能够揭示漏洞的错误消息。最后，为了使 Web 应用程序的模糊测试实用，我们需要建立一个合理的架构，以补偿通过网络发送输入带来的性能损失。

9.1 什么是 Web 应用模糊测试

Web 应用模糊测试是一种特殊形式的网络协议模糊测试。网络协议模糊测试（将在第 14 章“网络协议的模糊测试”中讨论）对任意类型的网络包进行变异，而 Web 应用模糊测试则专门关注遵循 HTTP 规范的网络数据包。鉴于 Web 应用在当今世界的广泛应用及其具有的重要性，我们将 Web 应用模糊测试单独拿出来当成一种独特的方法

来讨论。如今，相对于直接安装到本地计算机的传统软件产品而言，软件供应商越来越多地通过 Web 将软件作为一种服务来发布。Web 应用可以部署在其使用者提供的实体上，也可以部署在第三方上。当采用部署在第三方上的方式时，我们通常把这种模型称为应用服务提供者（Application Service Provider，ASP）模型。

Web 应用为其供应商和最终用户都带来了诸多好处。对供应商而言，Web 应用使其能够获得持续收益,并允许其进行包括安全相关更新在内的即时更新。而对最终用户而言，由于应用驻留在中心服务器上，他们无需下载和应用补丁。从最终用户的角度来说，维护工作由应用供应商完成，这意味着减少了应用程序的维护开销。然而，这些好处是建立在信任应用提供商会维持适当的安全性，保护你的私有数据免受窥视的代价之上的。因此，Web 应用的安全性应当作为一个我们首要关注的问题。

微软 LIVE

微软在 Web 应用不断增加的重要性方面下了重注。尽管微软的传统产品是 GUI 应用程序如微软 Office 等，但 2005 年 11 月，微软总裁比尔·盖茨宣布发布了两个基于 Web 的应用：Windows Live 和 Office Live[1]*。Windows Live 是针对个人的一组服务集，而 Office Live 则是面向小商业用户的应用。微软通过广告和订购模型从这两个服务中获利。Live 服务套件首次出现是在 2002 年 11 月，那时微软发布了 Xbox Live。Xbox Live 是 Xbox 视频游戏控制台的一个在线社区和市场。*

随着用于开发 Web 应用的开发语言和开发环境越来越友好，现在许多公司都在开发自己的 Web 应用，应用于企业内网，为其与商业伙伴和客户之间的商业活动和沟通提供便利。今天，无论公司规模大小，实际上都可以被看作是一个软件开发者。随着技术的发展，甚至一个人的公司也有能力开发并部署 Web 应用。然而，开发一个 Web 应用相对较为容易，但开发一个安全的 Web 应用就不那么容易了。如果一个 Web 应用由软件开发实力不是很强的公司开发，该应用在部署之前不太可能经过严格的安全性测试。不幸的是，我们并没有在第一时间发现能够跟上 Web 应用技术步伐的，易用和免费的，测试 Web 应用安全性的应用。本书的目标之一就是改变这种状况。下面，我们简要介绍 Web 应用开发过程中常用的一些技术。

115

[1] http://news.com.com/2061-10805_3-6026895.html

Web 应用相关技术

CGI

公共网关接口（Common Gateway Interface，CGI）最早是由美国国家超级计算机应用中心（National Center for Supercomputing Applications，NCSA）在 1993 年为 NCSA HTTPdWeb 服务器开发的一个标准。CGI 定义了数据应当如何在 Web 客户端和为 Web 服务器[2]处理该请求的应用程序之间进行传递。尽管可以用任何语言实现 CGI，但通常是用 Perl 来实现 CGI。

PHP

超文本预处理器（Hypertext Preprocessor，PHP）[3]是一种常用于开发 Web 应用的流行脚本语言。PHP 解释器的广泛性使得 PHP 可以应用在大多数的操作系统上。PHP 脚本可以被直接嵌入到 HTML 页中以处理 Web 页中需要动态生成的部分。

Flash

Flash 最初由 FutureWave Software 公司所开发的，1996 年 12 月 Macromedia 公司收购了该公司[4]。在这里讨论的这些技术中，从某方面来说 Flash 比较特别，因为它要求安装独立的客户端软件。而这里提到的其他大多数技术则只需要一个 Web 浏览器。

通过免费发布 Macromedia Flash 播放器，Macromedia 增加了 Macromedia Flash 的接受程度。Flash 使用一种被称为 ActionScript 的语言来处理大多数的交互特性。尽管 Flash 主要是一种客户端技术，但 Macromedia Flash 远程化（Flash Remoting）允许 Flash 播放器和 Web 服务器[5]应用之间进行交互。Macromedia 自身于 2005 年被 Adobe System 公司所收购[6]。

JavaScript

Netscape 于 1995 年的后期创建了 Javascript，作为一种允许在 Web 页面中[7]处理动态内容的手段。Javascript 可被应用为客户端或服务器端技术。当应用于客户端时，

116

[2] http://en.wikipedia.org/wiki/Common_Gateway_Interface

[3] http://www.php.net/

[4] http://en.wikipedia.org/wiki/Adobe_Flash

[5] http://www.macromedia.com/software/flashremoting/

[6] http://en.wikipedia.org/wiki/Macromedia

[7] http://en.wikipedia.org/wiki/Javascript

Javascript 被嵌入到发送给 Web 浏览器的 HTML 中，直接被浏览器解释。Javascript 也可以被用作一种服务器端技术，供 Web 服务器在创建动态内容时使用。其他的服务器端技术如微软的 ASP.Net（见下文）包含对 Javascript 的支持。

Java

Java 是 James Gosling 在 Sun Microsystems 的创造性产物。它最初的名字是 Oak，被设计用来在嵌入式系统中运行。后来被用做了基于 Web 的技术，当 Java 的发明者发现 Oak 已经被注册为商标[8]之后，它的名字就被改为了 Java 。Java 是编译型语言和解释型语言的交叉产物。Java 源代码被编译成字节码，然后字节码又被运行在目标平台上，由 Java 虚拟机进行解释。这种机制使得 Java 具备了平台无关性。Java 经常用在 Web 浏览器中，用于发布复杂的交互式应用。

ASP.Net

.Net 是一个开发平台，而不是一种开发语言。.NET 包含一个公共语言运行时（Common Language Runtime，CLR）环境，该环境可以被包括 Visual Basic 和 C#在内的许多语言使用。微软于 2002 年引入.Net，并将其作为包括 Web 应用在内的许多应用程序的开发平台。它和 Java 类似，其源代码也被编译为称作公共中间语言（Common Intermediate Language，CIL）的中间字节代码，然后中间代码再被虚拟机[9]所解释。可以使用具有语言无关性的 ASP.Net 来设计 Web 应用。而 ASP.Net 应用程序则可以使用任意与.Net 兼容的语言来编写。

117

9.2 测试目标

Web 应用模糊测试不仅能发现 Web 应用自身的漏洞，而且还能发现其底层构件中存在的漏洞（例如，Web 应用需要集成的 Web 服务器和数据库服务器）。虽然 Web 应用包含的应用程序类型非常广泛，但我们可以对其进行分类。以下列出我们的分类，对每种特定的应用类型，我们都给出一个可以通过模糊测试技术发现的漏洞的示例：

- Web 邮件

微软 Outlook Web Access Cross-Site 脚本漏洞

[8] http://en.wikipedia.org/wiki/Java_programming_language

[9] http://en.wikipedia.org/wiki/Microsoft_.Net

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=261

- 讨论板

phpBB 组 phpBB 任意文件揭露漏洞

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=204

- Wikis

Tikiwiki 用户习惯设置中的命令注入漏洞

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=335

- Web 日志

WordPress Cookie cache_lastpostdate 变量任意 PHP 代码执行

http://www.osvdb.org/18672

- 企业资源计划（ERP）

SAP Web 应用服务器 sap-exiturl Header HTTP 响应断开

http://www.osvdb.org/20714

- 日志分析

AWStats 远程命令执行漏洞

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=185

- 网络监视

IpSwitchWhatsUp Professional 2005(SPI)SQL 注入漏洞

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=268

影响多家供应商的 Cacti 远程文件包含漏洞

http://www.idefense.com/intelligence/vulnerabilities/display.php?id=265

以上给出的并不是 Web 应用类型的完整列表，但是它描述了通过 Web 发布的应用类型，并给出了各种应用类型可能具有的各类漏洞的示例。

9.3 测试方法

在开始对一个 Web 应用进行模糊测试之前，必须首先设立目标环境，然后为目标选择输入向量。Web 应用在这两方面都带来了特有的挑战。Web 应用的架构被设计为可以部署在多台网络机器上。这种设计保证了在生产环境中部署 Web 应用的扩展性，但却给模糊测试带来了性能上的下降。此外，存在多种隐蔽的 Web 应用输入方式，这些隐蔽的输入方式可能导致发生漏洞。因此，当定义 Web 应用的模糊测试输入时，必须采取更灵活的方法。

9.3.1 设置目标环境

模糊测试通常要求能够快速、连续地向目标应用发送大量输入。每个输入经历的事件包括：在本地生成输入，将输入发送到目标应用，目标应用对输入进行处理，监视输出结果。因此，模糊器运行所需的时间由所有事件中运行速度最慢的环节所决定。当对本地应用进行模糊测试时，整个测试过程的瓶颈是CPU时钟周期以及硬盘的读/写时间。考虑到现代计算机的硬件速度，这些时间可以非常小，因此模糊测试对于研究漏洞是一个可行的方法。

对 Web 应用模糊测试而言，测试过程的瓶颈往往由模糊器向目标应用发送的网络包的传输导致。考虑从远端（远程 Server）加载一个 Web 页面的过程。当浏览一个 Web 页面时，页面展现的速度由以下三个因素决定：你的计算机，该页面所在服务器，以及位于二者之间的 Internet 连接。你只能控制这三个因素中的第一个，也就是你自己的计算机。因此，当对 Web 应用进行模糊测试时，通过去除其他两个因素来提高网络通信的速度是非常重要的。可能的话，与其让目标应用在一个远程服务器上运行，不如将其部署到本地，这样数据包就无需通过网络进行传输。大多数桌面操作系统如 WindowsXP[10]或 Linux 都具有内嵌的 Web 服务器，因此，通常的选择是在本地机器上安装和配置目标应用。

119

当对一个 Web 应用进行模糊测试时，虚拟机（VM）应用如 VMWare[11]或微软的虚拟机（Microsoft Virtual Machine）[12]也是很有用的工具。当在本地运行模糊测试器时，被测目标 Web 应用可以运行在一个虚拟机实例中。这种方法提供了在本地运行目标应

---

[10] http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/iiiisin2.mspx

[11] http://www.vmware.com

[12] http://www.microsoft.com/windows/virtualpc/default.mspx

用所不具备的一些优越性。首先，通过虚拟机可以更好地管理目标应用所需要的大量资源。这确保了模糊测试进程不会消耗掉所在机器的所有资源。其次，引发系统崩溃和拒绝服务攻击的网络数据不会影响运行模糊测试器的本地机器。毕竟，如果运行模糊测试器的机器频繁地无法操作，我们将很难检测到这些我们关心的，引发系统崩溃等现象的异常。

9.3.2　输入

在开始模糊测试之前，首先必须要确定 Web 应用所提供的各种输入，因为这些输入将成为我们模糊测试活动的输入对象。这就引发了一个问题：哪些内容应该被当成输入？显然，Web 表单中的数据字段可以作为输入，URL 自身或者发送给 Web 服务器的 Cookies 能否作为输入呢？Web 请求中的请求头是否可以作为输入呢？答案是所有这些内容共同构成了输入。在我们看来，发送给 Web 服务器并被 Web 服务器解释的所有信息都应该被当成输入。

稍后我们将会对所有可能的输入进行分类，但首先，我们来看看 Web 请求的产生过程，以便更好地理解其机理。多数人使用 Web 浏览器如微软的 Internet Explorer 或 Mozilla Firefox 来访问 Web 页面。当 Web 浏览器配置好之后，只需简单的在地址栏中输入 URL 地址就可以访问 Web 页面的内容。

然而，当请求一个 Web 页面时，许多实际发生的活动隐藏在 Web 浏览器之下。为了更好地理解这些活动，我们手工发起对某个 Web 页面的请求。我们使用大多数现代操作系统都包含的 Telnet 程序来完成请求。Telnet 连接到你选择的主机和端口，然后开始发送和接收 TCP 包。

```
telnet www.fuzzing.oug 80[回车]
GET /HTTP/1.1[回车]
Host: www.fuzzing.org[回车]
[回车]
```

下面来分析一下该请求。首先，启动 Telnet 程序并向它提供两个参数：服务器名（fuzzing.org）和要连接的端口（80）。按照默认设置，Telnet 将连接到 TCP 端口 23。由于我们是手工向 Web 服务器发送一个请求，因此需要强制它使用 TCP 端口 80。接下来的代码行表示了 HTTP 协议所要求的最小化请求。首先，将使用的请求方法（GET）告知服务器（本章的后文中将详细介绍各种不同的请求方法），然后发送正在请求的路径和/或 Web 页。在这个案例中，我们请求的是服务器的默认 Web 页（/），而不是请求一

120

个特定的 Web 页面。在同一行中，我们还向 Web 服务器指明了要使用的 HTTP 协议的版本（HTTP/1.1）。下一行指定了请求的主机头（host header），该请求头在 HTTP1.0 中是可选的，但在 HTTP1.1[13]中是强制使用的。提交该请求后（注意需要使用两个回车来完成请求），服务器会返回如下所示的响应：

```
HTTP/1.1 200 OK
Cache-control: private
Content-Type: text/html
Set-Cookie: PREF=ID=56173d883ba96ae9:TM=1136763507:LM=1136763507:5=W4FkQu1vexo
Pq-:expires=Sun, 17-Jan-2038 19:14:07 GMT; path=/;domain=.google.com
Server:GWS/2.1
Transfer-Encoding: Chunked
Date: Sun, 08 Jan 2006 23:38;27 GMT
<html>
<head>
    <meta http-equiv= “content-type” content= “text/html:charset=UTF-8” >
    <title>Google</title>
    <style><!-
    Body.td.a.p..h{font-family:arial, sans-serif;}
    .h{font-size: 20px;}
    .q{color:#0000cc;}
    //-->
    </style>
    </head>
    <bodybgcolor=#ffffff text=#000000 link=#0000cc vlink=#551a8b
alink=#ff0000 topmargin=3  marginheight=3>
<center>
[snip]
<ahref=http://www.google.com/intl/en/about.html>About  Google</a>
<span id =hp style= “behavior:url(#default#homepage)” ></span>
</font><p><font size=-2>&copy;2006 Google</font></p></center>
</body>
</html>
```

接收到的响应是 Web 页的 HTML 源码，前面的一系列头信息向浏览器提供了响应有关的附加信息。如果将该响应内容的 HTML 部分保存到文件中，并用 Web 浏览器将其打开，看到的就是显示出来的页面，该页面同使用浏览器访问 URL 所得到的页面是一样的。但是，如果返回的 HTML 中使用的是相对链接而不是绝对链接，用浏览器打

[13] http://rfc.net/rfc2616.html#s14.23

开通过 Telnet 方式得到的 HTML 内容时可能会丢掉一些图像。

我们已经提到，前面这个请求仅满足了 HTTP 协议所要求的最小化格式要求。那么除此之外，我们还能向 Web 服务器发送哪些其他输入呢？为了回答这个问题，我们先来嗅探 Internet Explorer Web 浏览器所发送的请求。使用 Ethereal 工具，我们可以抓到下面的请求：

```
GET /HTTP/1.1
Accept: "/"
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0(compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR
1.1.4322; .NET CLR 2.0.50727)
Host: www.google.com
Connection: Keep-Alive
Cookie:
PREF_ID_32a1c6fa8d9c9a7a:FF_4:LD_en:NR_10:TM_1130820854:LM_1135410309:S_b9I4G
WDAtclpmXBF
```

这些附加的头（header）表示什么意思呢？HTTP 协议定义了许多头（header），而每个头（header）都有一些可接受值。HTTP/1.1 协议的详细内容参见 RFC2616 的第 176 页——超文本传输协议（Hypertext Transfer Protocol，HTTP/1.1）[14]。我们并不在这里描述整个文档，仅对前面所遇到的头（header）进行说明：

122

```
Accept:*/*
```

Accept 头（header）指明了可以在响应中使用的媒体类型。上面的例子表示所有的媒体类型（*/*）都是可以接受的。通过 Accept 头（header），我们可以限制响应中只包含特定的媒体类型如 text/html 或者 image/jpeg。

```
Accept-Language:en-us
```

Accept-Language 头（header）允许用户指定可以在响应中使用的自然语言的类型。上面的例子要求响应使用美式英语。RFC1766 中定义了语言标签[15]的正确格式，也就是各种不同语言的标识符。

```
Accept-Encoding:gzip,deflate
```

---

[14] http://rfc.net/rfc2616.html

[15] http://rfc.net/rfc1766.html

该头（header）同样是为响应指定一种可接受的格式，它定义了可接受的编码模式。在上面的例子中，我们发送的请求表明可以使用 gzip[16]或者 deflate [17]编码算法。

```
User-Agent:Moailla/4.0(compatible;MSIE 6.0;

Windows NT 5.1;SV1;.NET CLR 1.1.4322;.NET CLR 2.0.50727)
```

User-Agent 头（header）定义了发出请求的客户端（Web 浏览器）。这对服务器而言是非常重要的，因为该头允许服务器对响应进行剪裁以适应不同浏览器所支持的不同功能。此例中的User-Agent定义为在Windows XP SP2上运行的微软Internet Explorer 6.0。

```
Host:www.google.com
```

该头（header）定义了被请求的 Web 页面所在的主机和端口。如果没有包含端口信息，那么服务器将使用默认的端口。这个字段对服务器而言很重要，因为在单一的 IP 地址上允许同时绑定多个主机名。

123

```
Connection:Keep-Alive
```

Connection 头（header）允许客户端指定连接需要的多个选项。持久连接可以响应多次请求，而不用为每次请求打开一个单独的连接。Connection:close 意味着一旦响应发送完成，连接就应当立即被关闭。

```
Cookie :PREF=ID=32b1c6fa8e9e9a7a:FF=4:LD=en:

NR=10:TM=1130820854:LM=1135410309:5=b9I4GWDAtc2pmXBF
```

Cookies 可以在计算机的本地硬盘或者内存中保存一段指定的时间，当当前任务完成之后它们将被抛弃。服务器通过 Cookies 可以识别出请求者。这使得网站可以跟踪用户的上网习惯，使得为用户定制默认的 Web 页面成为可能。如果某个特定网站的 Cookie 已经存在于本地，那么浏览器将会在向这个网站发送请求时将其提交给服务器。

现在，我们已经研究了一个请求的例子，这能够帮助我们更好地定义 Web 应用模糊测试可以使用的不同输入。前面我们提到，发送给 Web 服务器的请求数据的任何部分都可被认为是一个输入。从高层次上说，发送给 Web 服务器的请求数据包括请求方法，请求的统一资源定位符（Uniform Resource Identifier，URI），HTTP 协议版本，HTTP

---

[16] http://rfc.net/rfc1952.html

[17] http://rfc.net/rfc1951.html

头以及发送的数据。在下一节，我们将分析每个组成部分，并为每个部分确定合适的模糊测试变量：

```
[方法] [请求的 URI]  HTTP/[主版本].[次版本]
[HTTP 头]

[提交的数据]
```

方法（Method）

GET 和 POST 方法是请求 Web 页最常用的两个方法。这两个方法采用名称－值对的方式向 Web 服务器请求特定的内容。可以认为它们是 Web 请求中的变量。GET 方法在请求的 URI 中向 Web 服务器提交名称-值对。例如，请求 http://www.google.com/search?as_q =security&num=10 向 Google 搜索引擎提交一个请求，告知它我们想要搜索的词为 security（as_q=security），要求在每个返回页中显示 10 个搜索结果（num=10）。当使用 GET 方法时，这些名称－值对以字符“?”开头，并被追加到 URI 的后面，每个不同的名称-值对之间以字符“&”隔开。

也可以使用 POST 方法提交名称-值对。当使用 POST 方法时，名称-值对作为 HTTP 头提交，后面跟着其他的标准 HTTP 头。使用 POST 方法的优点之一是可以发送任意大小的值。尽管 HTTP 规范并没有特别限定 URI 的总长度，但 Web 服务器和 Web 浏览器通常对此施加了一定限制。如果一个 URI 超过了预期的长度，那么 Web 服务器将返回一个 414（请求 URI 过长）状态。使用 POST 方法的缺点是不能通过转发 URI 来达成对一个动态生成的 Web 页面的共享。例如，人们经常通过转发由特定搜索生成的 Google 地图的 URI 来共享地图和方位。你能猜到下面的地图链接指向哪里吗？

```
http://maps.google.com/maps?hl=en&q=1600+Pennsylvanis+Ave&near=20500
```

正如我们所提到的，有其他一些可以用于 Web 服务器请求中的方法。下面简要介绍其他合法的方法。

- HEAD：类似于 GET 方法，但是服务器只返回响应头，并不返回被请求 Web 页的 HTML 内容。
- PUT：允许用户向 Web 服务器上传数据。尽管该方法并没有被广泛的支持，但已经发现了支持 PUT 方法，但没有正确实现导致的安全漏洞。例如，微软的安

全公告 MS05-006[18]中提到的安全问题正是这样一个漏洞。该漏洞允许未授权的用户使用 PUT 方法[19]向微软的 SharePoint 服务器上传数据。

- DELETE：允许用户请求从 Web 服务器中删除一个资源。同样，尽管该方法没有被广泛的实现，但也应当对其进行完善的测试。不正确地暴露该方法可能会导致攻击者通过删除 Web 服务器上的资源来拒绝合法用户的访问。
- TRACE：允许客户端提交一个请求并回显到发送请求的客户端。这对于调试网络连接性问题很有帮助，因为你可以查看到实际被发送给服务器的请求的结构。2003 年 1 月，WhiteHat Security 的 Jeremiah Grossman 发表了名为“跨站跟踪（Cross-Site Tracing,XST）[20]”的白皮书，该文档披露了如何利用客户端脚本，允许恶意的 Web 服务器获取对支持 TRACE 方法的第三方 Web 服务器的 Cookie 值的访问。显然，测试应当识别出这种情形，也就是什么地方隐式地支持了 TRACE 方法。

125

- CONNECT：保留以供代理使用，可以动态的转换为一个通道。
- OPTIONS：允许客户端询问 Web 服务器以确定服务器所支持的标准的和私有的方法。漏洞扫描者可以利用该方法确定 Web 服务器是否实现了潜在存在漏洞的方法。

OPTIPONS 方法也可以被用来确定一个 Web 服务器是否正在运行一个受 WebDAV 中的严重漏洞影响的 Internet 信息服务器（Internet Information Services，IIS）的特定版本（WebDAV 是为了方便 Web 内容发布而对 HTTP 协议所做的一组扩展）。WebDAV 的漏洞信息在 MS03-007[21]（Windows 组件中未检查的缓冲区可能导致服务器的安全问题）中有详细的描述。当发出一个 OPTIONS*HTTP/1.0 请求时，支持 OPTIONS 选项的服务器将返回下面的请求，标识 WebDAV 是否可用。下面列出的服务器响应表明 WebDAV[22]不可用，因为在 Public 头中没有列出任何 WebDAV 扩展：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 17 Mar 2003 21:49:00 GMT
Public: OPTIONS, TRACE, GET, HEAD, POST
```

---

18 http://www.microsoft.com/technet/security/Bulletin/MS05-006.mspx

19 http://support.microsoft.com/kb/887981

20 http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

21 http://www.microsoft.com/technet/security/bulletin/MS03-007.mspx

22 http://www.klcconsulting.net/articles/webdav/webdav_vuln.htm

```
Content-Length: 0
```

下面的响应内容包含了完整的头信息，该头信息中包含了 WebDAV 所提供的扩展。当 WebDAV 在运行微软 IIS5.0 的服务器上可用时，如果还没有打上适当的补丁，那么服务器就可能会存在 MS03-007 中的漏洞。

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 17 Mar 2003 21:49:00 GMT
Content-Length: 0
Accept-Ranges: bytes
DASL:
DAV: 1.2
Public: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, MKCOL, PROPFIND,
PROPPATCH, LOCK, UNLOCK, SEARCH
Allow:OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, PROPFIND, PROPPATCH,
LOCK, UNLOCK, SEARCH
Cache-Control: private
```

126

请求 URI（Request-URI）

在传递了方法（method）之后，客户端将向 Web 服务器传递请求 URI。请求 URI 的目的是定位被请求的资源（Web 页面）。URI 可以采用绝对 URI 格式（如 http://www.target.com/page.html），或是相对 URI 格式（如/page.html）。另外，除了定位服务器上的特定资源外，通过“*”字符 URI 也可用于定位服务器自身，OPTIONS 方法需要 URI 的这种能力。

当对请求的 URI 进行模糊测试时，URI 的每个部分都可以被模糊化。以下面的相对路径为例：

```
/dir/page.html?name1=value1&name2=value2
```

该 URI 可以被分解为以下组成部分：

```
/[路径]/[页面].[扩展名]?[名称]-[值]&[名称]-[值]
```

每个单独的部分都可以，而且应当被模糊化。其中一些部分应当使用已知的，导致前面提到的漏洞的值来模糊化，而另外一些则应当使用随机值，以确认服务器是否良好地处理了异常请求。随机值应当包括大批量的数据，以确定当服务器解析和解释数据时是否会发生缓冲区溢出。下面我们分别来看看 URI 的每个部分。

- **路径（Path）**：当对路径项进行模糊测试时，最常发现的漏洞就是缓冲区溢出和

目录遍历攻击。缓冲区溢出可以通过发送大量数据来识别，而目录遍历则可以通过发送连续的“../”字符序列来发现。我们应当使用不同的编码模式以绕过解码之前的输入验证程序。

- **缓冲区溢出（Buffer overflow）示例**：对于 Macromedia JRun 4 更新 5 发布之前的 JRun4 Web 服务器来说，接受过长路径时易出现基于栈的缓冲区溢出[23]。当路径长度超过大约 65536 个字符时，该漏洞会被触发。这说明了进行模糊测试时包含超长变量的必要性。
- **目录遍历（Directory traversal）示例**：3Com 的 Network Supervisor 应用中的一个漏洞是目录遍历漏洞的经典例子[24]。该应用启动一个在 21700 端口监听 TCP 协议的 Web 服务器。在 Network Supervisor 的 5.0.2 及其早期版本中，一个包含连续的“../”字符序列的简单 URL 将允许用户遍历 webroot 目录。需要说明的是，这样的漏洞实际上是服务器中的漏洞，而不是应用中的漏洞。
- **页面（Page）**：对公共页面名称进行模糊测试，可能会发现那些没有受到正确保护的页面，以及导致缓冲区溢出漏洞的页面。
- **缓冲区溢出（Buffer overflow）示例**：当发送超长请求访问扩展名为.htr，.stm 和.idc 的文件时，微软 IIS4.0 会发生基于栈的缓冲区溢出。微软的安全公告 MS99-019[25]中详细描述了该漏洞，并导致 IIS 4.0 成为了许多公开攻击的目标。
- **信息泄漏（Information leakage）示例**：3Com 的 OfficeConnect 无线 11g 接入点（3Com OfficeConnect Wireless 11g Access Point）包含一个漏洞，可以在无需提供适当的授权凭证的情况下，访问基于 Web 的管理界面上的敏感 Web 页[26]。这样，请求一个页如/main/config.bin 将显示该页面的内容，而不是显示登录提示，页面的内容中包括了管理员的用户名和密码等信息。Nikto[27]是搜索此类漏洞的应用程序的一个例子，它通过向 Web 服务器发送重复请求来发现公共的和可能不安全的 Web 页。
- **扩展（Extension）**：与本地文件一样，Web 页的文件扩展名通常指明了生成 Web 页所采用的技术。Web 页文件扩展名的例子包括*.html（超文本标记语言，HyperText Markup Language），*.asp（动态服务器页，Active Server Page）和*.php

127

[23] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=360

[24] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=300

[25] http://www.microsoft.com/technet/security/bulletin/MS99-019.mspx

[26] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=188

[27] http://www.cirt.net/code/nikto.shtml

（超文本预处理器，Hypertext Preprocessor）。如果能够向具有未知扩展名的 Web 页发出请求，就说明 Web 服务器中存在漏洞。

- **名称**（Name）：对公共名字组件进行模糊测试能够发现服务器接受的，未文档化的变量。换句话说，如果应用程序没有足够的错误处理能力，发送非期望的变量可能会导致应用工作不正常。
- **值**（Value）：采用何种合适的方式对值进行模糊测试，依赖于 Web 应用期望的变量类型。例如，如果请求中的名称-值对是 length=50，那么用超长值或者小的数字值来对该项进行模糊测试就是合理的。例如，如果 length 的值小于提交数据的实际长度，会发生什么事？如果 length 的值为零，甚至为负数，情况又会怎样？或许一个超出应用设计处理能力的值会导致应用崩溃或者发生整数溢出。通过模糊测试可以找到这些问题的答案。如果值项期望的是一个字符串内容，那就试着发送非期望字符串值，查看应用程序能否很好地对其进行处理。最后，同样重要的是，逐渐增多值项的数据，以查看是否会发生缓冲区溢出或崩溃。
- **分隔符**（Separator）：即使是字符串中看上去无害的各组成部分间的分隔符（/,=,&,.,:等）也应当被作为模糊测试的目标。这些字符会被 Web 服务器或应用程序解析，如果服务器不能正确处理其中的异常值，那就可能会导致发生可被利用的漏洞。

128

协议（Protocol）

可以使用数字变量来模糊化 HTTP 协议的版本号，提交支持或不支持的 HTTP 协议版本以发现某些服务器漏洞。当前最稳定的 HTTP 协议的版本是 HTTP/1.1，HTTP 协议被分为主版本和次版本（HTTP/[主版本].[次版本]）。

头（Headers）

所有的请求头都可以而且应当被模糊化。请求头应遵循如下的格式：

```
[头名称]：[头值]
```

因此，这里有三个可能被模糊化的变量：名称（name），值（value）和分隔符（:）。应该使用已知的合法值来模糊化名称，以确定 Web 应用是否支持未文档化的头信息。不同 HTTP 协议支持的头的列表可以在下面的 RFC 中找到：

- **RFC 1945-超文本传输协议-HTTP/1.0** [28]

129

[28] http://rfc.net/rfc1945.html

- RFC 2616-超文本传输协议-HTTP/1.1 [29]

模糊化值变量可用于确定应用程序是否能正确地处理异常值。

堆溢出示例

2006 年 1 月，iDefense 实验室发布了在 Novell SUSE Linux 企业服务器 9[30]中一个可被远程利用的堆溢出漏洞的细节。简单地提交一个 POST 请求就可以触发该漏洞，只需在请求的 Content-Length 头中包含一个负数即可。可以触发该漏洞的一个请求的例子如下所示：

```
POST / HTTP/1.0
Content-Length: - 900
[用于覆写堆的数据]
```

Cookies

Cookies 存储在本地，当向 Cookie 对应的 Web 服务器发送请求时，它将以 HTTP 头的形式被提交。Cookie 的格式如下所示：

```
Cookie: [名称 1]=[值 1]; [名称 2] =[值 2] …
```

同样，应当对名称（name），值（value）和分隔符（:）进行模糊化。与其他值一样，同样应该根据合法请求中提交的变量类型来决定模糊值。

130

Post 数据（Post Data）

前面已经提到，既可以使用 GET 方法在请求的 URI 中向 Web 服务器提交名称-值对，也可以使用 POST 方法以独立 HTTP 头的形式向 Web 服务器提交名称-值对。Post 数据以下面的格式发送：

```
[名称 1]-[值 1]&[名称 2]-[值 2]
```

缓冲区溢出示例

iDefense 实验室的 Greg MacManus 在流行的 Linksys WRT54G 无线路由器[31]包含的 Web 服务器中发现了一个缓冲区溢出漏洞。他发现向 apply.cgi 发送一个内容长度超过 10000 字节的 POST 请求将会导致缓冲区溢出。尽管嵌入式 Web 服务器中的这种漏洞通

---

[29] http://rfc.net/rfc2616.html

[30] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=371

[31] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=305

常很难被利用，但是设备的开源固件，以及带开发工具的修改后的固件，能够帮助开发出利用该漏洞的基础代码。

识别输入

现在我们已经知道了发给 Web 应用的请求的结构，并且了解了这些请求中可以作为单独输入变量以及潜在模糊测试目标的各个不同的组成部分。下一步就要搜索以识别所有不同的合法输入值。这可以通过手工方式，也可以用自动化方式来实现。不论采用哪一种方法，我们都要尽可能完整地覆盖输入值。当研究应用程序时，我们的目标是列出下面所有的输入：

- Web 页
- 目录
- Web 页支持的方法
- Web 表单
  - 名称－值对
  - 隐藏字段
- 头
- Cookies

131

识别上面提到的这些输入的最简单也是效率最低的方法，就是使用 Web 浏览器打开一个 Web 页，然后查看该页的源代码，在源代码中查找包含在 Web 表单中的输入。确保查找页面源代码中的隐藏字段（input type="hidden"），因为偷懒的应用开发者有时候会把隐藏字段作为实现安全性的一种手段。他们可能会假定你不会对这些字段进行测试，因为这些字段在页面上是不可见的。这显然是一种非常弱的控制方法，因为从页面的源代码中总可以查看到隐藏字段。

如果使用 Web 浏览器来识别输入，是无法看到请求/响应头信息的。例如，你看不到发送给应用服务器的 Cookies 的结构，因为它们包含在 HTTP 头中，并且会被浏览器自动地传递给浏览器。要查看原始 HTTP 请求，你可以使用网络协议分析器如 Wireshark[32]等。

即使使用 Web 浏览器和嗅探器，一旦 Web 应用的页面超过一定数量，手工识别所有可能的输入也是不可行的。幸运的是，可以使用 Web 蜘蛛（Web spider）自动跟踪和遍历 Web 应用。Web 蜘蛛（或叫 Web 爬虫）是一个应用，它能识别出一个 Web 页包含

[32] http://wireshark.org/

的所有超链接，遍历这些链接，再发现额外的超链接并遍历之，重复执行直到访问到所用可能的 Web 页。同时，它还可以获得对安全研究者而言非常重要的信息，如前面提到的那些输入。幸运的是，当前有许多功能出色的免费或是开源的 Web 蜘蛛。wget[33]工具是一个简单但功能强大的 Web 蜘蛛。尽管 wget 最初是为 UNIX 操作系统设计的，但 win32 平台上也有可用的 wget 移植版本[34]。我们要推荐的另一个免费 Web 蜘蛛包含在 WebScarab 中。WebScarab 项目是开放 Web 应用安全项目（Open Web Application Security Project，OWASP）所提供的一组工具集，它对于分析 Web 应用非常有用。在 WebScarab 提供的众多工具中，有一个 Web 蜘蛛可以用来自动化地识别 Web 应用中所有的 URL。另一个有用的 WebScarab[35]工具是一个代理服务器，可用于手工的 Web 页面审核。如果设置 Web 浏览器使用 WebScarab 代理，那么在运行应用程序时，它将记录下所有的原始请求和响应。WebScarab 甚至包含了一个非常基础的 Web 模糊测试器，但在下一章中，我们将讨论如何创建一个功能更强大的模糊测试器。

132

9.4 漏洞

Web 应用容易受到许多类型漏洞的影响，所有这些漏洞都可以通过模糊测试识别。下面的列表给出了标准的漏洞分类。

- **拒绝服务**（Denial-of-service，DoS）：Web 应用上的 DoS 攻击是一个显著的威胁。尽管 DoS 攻击并不会让攻击者获得目标应用的访问权限，但会导致你的公司的外部入口站点，甚至是收入来源的站点拒绝用户的访问，造成大的金钱损失。
- **跨站脚本**（Cross-site scripting，XSS）：根据 Mitre 的统计，2006 年新发现的漏洞中，XSS 漏洞占 21.5%[36]。这使得 XSS 漏洞成为了最常见的漏洞，不仅是 Web 应用中，而且是所有应用中最常见的漏洞。XSS 漏洞曾一度仅被当作是麻烦，而不是安全威胁，但随着钓鱼攻击（phishing attack）的爆炸性增长，这种状况已经完全改变了。XSS 漏洞使得攻击者能够控制 Web 浏览器中的客户端活动，因此成为了进行钓鱼攻击的攻击者的有用工具。

---

33 http://www.gnu.org/software/wget/

34 http://gnuwin32.sourceforge.net/packages/wget.htm

35 http://www.owasp.org/software/webscarab.html

36 http://cwe.mitre.org/documents/vuln-trends.html#table1

- **SQL注入（SQL injection）**：在Web应用漏洞中，SQL注入不仅是最常见的漏洞之一，而且是最严重的漏洞之一。2006年Mitre的统计称SQL注入排名第二，占全年新发现漏洞的14%。这种情况一方面是因为带关系型数据库支持的动态网站快速增长，另一方面则是因为大多数教科书里仍然在教授不安全的SQL编码实践。有些人误以为SQL注入只能攻击提供读取数据库记录途径的应用，只会带来纯粹的机密风险。但随着大多数关系型数据库开始支持功能强大的存储过程，SQL注入将成为更大的风险，导致对可信后端系统的全面威胁。
- **目录遍历/弱访问控制（Directory traversal/Weak access control）**：目录遍历曾经相当常见，但幸运的是现在已基本消失。当然，目录遍历仍然是一个存在的威胁，因为只要开发者忘记对一个受限的页面或目录设置正确的访问控制，就会出现该漏洞。这也是为什么应该持续不断地对Web应用进行审核的原因之一，审核不仅应当在应用发布之前进行，而且应当贯穿应用的整个生命周期。曾经安全的Web应用可能会由于服务器错误的配置变更（删除或者弱化访问控制）而突然变得不安全。 133
- **弱认证（Weak authentication）**：采用认证模式没有什么缺点，但如果没有正确实现认证，那么一切都可能会不安全。弱密码是容易被暴力攻击攻破的漏洞，而用明文传递认证凭证则使得它们容易在无线网络中被捕获。尽管通过基本的努力很容易发现这些错误，但更大的挑战来自理解应用的业务逻辑，确保开发者没有错误地在应用中放置任何可以绕过认证控制的入口。
- **弱会话管理（Weak session management）**：HTTP协议是一个无状态的协议，因此需要有一些形式的状态管理以便能够区分并发用户，使得用户不用在每个页面上都要输入认证信息。唯一会话令牌（token）究竟是通过Cookies传递，还是在URI内部或是页面数据中传递并不重要，重要的是如何组织和保护令牌。Cookies应当足够随机，足够大，使得强制的漏洞攻击不可行。另外，会话令牌需要足够频繁地过期，以防止重放攻击。
- **缓冲区溢出（Buffer overflow）**：相对桌面应用和服务器应用而言，Web应用中的缓冲区溢出攻击并不很常见。这主要是因为Web应用开发通常使用的编程语言（如C#或Java）具有内存管理控制功能，减少了缓冲区溢出的风险。但这并不是说，对Web应用进行模糊测试时就可以忽略缓冲区溢出。很可能Web应用会将用户提供的输入传递给用C或C++编写的独立应用，而该独立应用易于受到缓冲区溢出的攻击。此外还要记住，当进行模糊测试时，至少有两个测试目标：Web应用和Web服务器。Web服务器通常是用易于发生缓冲区溢出的编程语言编写和实现的。

- **未恰当支持的 HTTP 方法**（Improperly supported HTTP method）：通常情况下，Web 应用处理 GET 和 POST 请求。然而，正如前面所讨论的，除 GET 和 POST 方法外，还存在许多其他的与 RFC 兼容的，或是第三方的方法。如果没有正确地实现这些方法，攻击者就会有机会操纵服务器上的数据，或者得到后续攻击所需要的有价值的信息。因此，应当通过模糊测试识别出 Web 应用支持的所有方法，确定它们是否被恰当地实现。
- **远程命令执行**（Remote command exection）：Web 服务器可能会向其他应用或操作系统本身传递未经验证的用户输入。如果传入的输入没有经过适当的验证，攻击者就会有机会直接执行目标系统上的命令。PHP 和 Perl 应用通常特别易于受到这类攻击。
- **远程代码注入**（Remote code injection）：PHP 应用易于受到这类攻击。不良的编码习惯，包括允许向 include()或 require()等方法传入未经验证的用户输入，会允许导入本地或是远程的 PHP 代码。当用户输入被直接传给这些方法时，攻击者就可以将他们自己的 PHP 代码注入到目标应用中。据 Mitre2006 年的统计，此类漏洞占新发现漏洞的 9.5%。
- **带有漏洞的库**（Vulnerable libraries）：开发者通常会错误地信任包含在应用程序中的第三方库。Web 应用中包含的所有代码，无论是开发者从头编写的，还是从第三方获得的预编译过的代码，都可能存在潜在漏洞，都应当在安全性测试中进行相同程度的审查。至少应当检查相关文档，确保所使用的第三方库没有已知的漏洞。此外，还应当对第三方库进行与自行编写的代码相同水平的模糊测试，和其他安全性测试。
- **HTTP 响应分割**（HTTP response splitting）：随着 Sanctum 公司发布的白皮书“分割与征服”[37]，HTTP 响应分割第一次被广泛了解。如果用户能够向响应头信息中注入一个回车换行（CRLF）序，就可以发起该攻击。它使得攻击者能够篡改 Web 服务器提供的响应，并导致多种攻击，这些攻击可以破坏 Web 代理和浏览器的 cache。
- **跨站请求伪造**（Cross Site Request Forgery，CSRF）：跨站请求伪造攻击很难防护，是一种不断增长的威胁。当被攻击的用户拥有一个活跃会话时，如果攻击者能够诱使用户执行某些动作，如点击一个存在于 e-mail 消息中的链接，那就可能触发一次跨站请求伪造攻击。例如，在某个银行网站上，资金转帐请求通过提交 Web 表单完成，提交的表单包含账户信息和转帐金额。只有账户的所

134

[37] http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

有者才能进行转帐，而且只有账户的所有者才能使用他或她的凭证信息进行登录。然而，如果账户的所有者已经登录，并且在不知情的情况下点击了一个链接，使得相同的 Web 表单数据发送给银行网站，那么在被攻击者无察觉的情况下同样会发生转帐操作。为了防护跨站请求伪造攻击，Web 站点通常要求用户在执行一项敏感操作如资金转帐之前，重复确认或执行某些手工操作。某些 Web 站点也开始在 Web 表单中实现一次有效的值，以验证表单的来源。

尽管这里并没有完整列出所有可能的 Web 应用漏洞，但它说明了 Web 应用程序很容易受到多种漏洞的攻击。同时它也说明了有些影响本地应用的漏洞同样会影响 Web 应用，但有许多攻击是专门针对 Web 应用的。为了通过模糊测试发现 Web 应用的漏洞，必须通过 HTTP 协议提交输入，并且必须要使用与对本地应用模糊测试不同的机制来发现错误。要了解完整的 Web 应用漏洞信息，我们推荐读者参考 Web 应用安全协会的威胁分类项目（Web Application Security Consortium's Threat Classification project）[38]。

135

9.5　异常检测

异常检测是 Web 应用模糊测试中非常具有挑战性的一项工作。如果我们让模糊测试器连续运行整晚，并且发现应用接收到的超过 10000 个请求中的某个请求导致发生崩溃，这样的结果对我们基本没有意义。根据这个结果，我们只知道存在着一个漏洞条件，但却无法重现。因此，下面列出的这些可以用来识别潜在漏洞条件的数据非常重要。

- **HTTP 状态码**：Web 服务器的请求响应包含一个三位数字的代码以标识请求的状态。在 RFC2616－超文本传输协议－HTTP/1.1[39]的第十部分可以找到完整的状态码列表。状态码可以提供确定哪些模糊请求需要进一步研究的线索。例如，一系列 500 错误（表示“内部错误”的状态码）可能意味着前面的请求导致服务器发生错误。401 错误（表示“未授权错误”的状态码）则意味着被请求的页面存在，但是受密码保护。
- **Web 服务器错误消息**：Web 应用可能会直接在 Web 页面内容中显示错误消息。使用正则表达式来解析包含在响应中的 HTML 可以找到这类消息。
- **中断连接**：如果某个模糊测试输入导致 Web 服务器挂起或发生崩溃，后续请求

[38] http://www.webappsec.org/projects/threat/

[39] http://rfc.net/rfc2616.html

将不能成功地连接到服务器。因此，模糊测试工具应当维护日志文件，记录管道中断或是连接失败事件。当查看日志文件并发现一系列的连接失败后，就可以直接查看刚好在发生错误的日志条目之前的请求，确定故障所在。

- **日志文件**：大多数 Web 服务器可以被配置为记录多种类型的错误。这些日志同样可以提供发现哪些模糊请求会导致一个可被利用的条件发生的线索。使用日志文件所面临的难点是无法直接根据日志文件找到导致错误的请求。将请求和错误日志联系起来的一个可行方法是，将攻击计算机和目标计算机的时钟进行同步，查看请求和日志条目的时间戳，这样可以缩小需要检查的范围，但它仍然不能准确、确定地将某个模糊测试请求关联到它导致的故障。
- **事件日志**：事件日志也不能直接对应到相应的请求，但可以通过时间戳关联到特定事件，在这一点上它与 Web 服务器日志类似。微软的 Windows 操作系统使用了事件日志，可以通过事件查看器（Event Viewer）应用查看事件日志。
- **调试器**：鉴别已处理和未处理异常的最好方法是，在开始模糊测试之前将目标应用连接到调试器。错误处理机制可能会掩盖模糊测试导致的许多明显的故障表现，但通过调试器通常可以发现这些错误。寻找已处理的异常和寻找未处理的异常同样重要，因为错误处理机制会处理某些特定类型的错误，如解引用空指针以及格式字符串引发的异常，但如果给定恰当的输入，它们也可以成为可被利用的漏洞。和上面的异常检测方法一样，这里面临的最大挑战仍然是确定哪个请求导致异常的产生。在 Web 应用模糊测试方面调试器有一定的局限性，因为调试器可以发现服务器的异常，却不能发现应用层的异常。

9.6 小结

在本章中，我们定义了 Web 应用和 Web 服务器模糊测试，学习了如何使用模糊测试发现 Web 应用中的漏洞。这要求我们全面了解 Web 浏览器如何提交请求，以及 Web 服务器如何响应请求。通过这些知识，我们就能确定所有能被最终用户控制和操纵的那些“巧妙的”输入。使用这些输入进行模糊测试，监视潜在异常，就能发现漏洞。在下一章中，我们将创建一个 Web 应用模糊测试器以实现测试过程的自动化。

第10章

Web应用和服务器的自动化模糊测试

"The most important thing is for us to find Osama bin Laden. It is our number one priority and we will not rest until we find him."

——George W. Bush, Washington, DC, September 13, 2001

"I don't know where bin Laden is. I have no idea and really don't care. It's not that important. It's not our priority."

——George W. Bush, Washington, DC, March 13, 2002

在上一章中，我们已经讨论了如何对 Web 应用进行模糊测试，该是尝试我们理论的时候了。在本章中，我们将利用在背景知识章节中学到的知识，应用这些知识来开发一个图形化的 Web 应用模糊测试器 WebFuzz。我们先确定应用的设计和找出我们面临的独特挑战，然后，我们选择一个合适的开发平台并构建模糊测试器。当然，完成开发并不表示我们完成了所有工作。在使用我们开发的工具找到漏洞之前，决不应该认为我们的开发工作已经完成了。这就好比造一辆车，造好后不可能不试驾，对不对？因此，我们会用许多不同类型的已知 Web 应用漏洞来测试我们开发的工具，看看 WebFuzz 能不能发现这些漏洞。

138

10.1 Web 应用模糊测试器

Web 应用模糊测试并不是新概念，目前已经出现了多种 Web 应用模糊测试器，而且数量还在不断增长。下面我们列出了一些流行的，免费和商业的 Web 应用模糊测试器。

- SPIKE Proxy[1]：这是个基于浏览器的 Web 应用模糊测试器，由 Dave Aitel 用 Python 开发。它以代理方式工作，捕获 Web 浏览器发出的请求，允许用户针对目标 Web 站点运行一系列预定义的审计工作，试图找出多种类型的漏洞，如 SQL 注入，缓冲区溢出，或是 XSS。由于 SPIKE Proxy 基于开源框架，因此它可以被进一步扩展，对更多类型的目标进行模糊测试。SPIKE Proxy 不只是个模糊测试器，它是漏洞扫描器和模糊测试器的集合体。图 10.1 显示了 SPIKE Proxy 的截图。

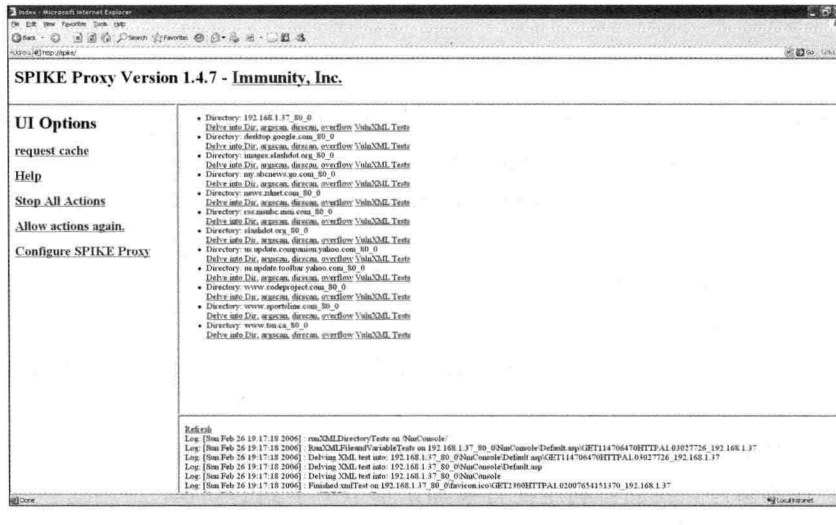


图 10.1 SPIKE Proxy

- WebScarab[2]：开放 web 应用安全项目（OWASP）开发了多种测试 Web 应用安全性的工具，其中就包括 WebScarab。尽管该工具更像是整体 Web 应用安全性

[1] http://www.immunitysec.com/resources-freesoftware.shtml

[2] http://www.immunitysec.com/resources-freesoftware.shtml

测试工具，但它包含了一个基础的模糊测试器，能够将模糊测试值注入到应用参数中。

- SPI Fuzzer[3]：SPI Fuzzer 是 SPI 工具包的组件之一，而 SPI 工具包本身又是 WebInspect 应用的一部分。WebInspect 由 SPI Dynamics 公司开发，是一款商业工具，提供了一整套全面测试 web 应用的工具。
- Codenomicon HTTP Test Tools[4]：Codenomicon 为所有你能想到的协议开发了商业模糊测试套件，其中当然包括 HTTP 协议。
- beSTORM[5]：和 Codenomicon 一样，Beyond Security 公司的业务主要是开发商业模糊测试器。beSTORM 是一款能够处理多种 Internet 协议的模糊测试器，包括 HTTP 协议。

WebFuzz 的灵感来自商业工具 SPI Fuzzer。SPI Fuzzer 是一款简单但却设计得非常好的图形化 Web 应用模糊测试器，该工具提供了完全控制进行模糊测试的原始 HTTP 请求的能力。用户需要具有一定的 HTTP 协议基础知识，才能开发出能得到结果的测试。同样，只有具有一定的 HTTP 基础才能解析响应，找出需要进一步研究的响应。图 10.2 展示了 SPI Fuzzer 工具。

SPI Fuzzer 的主要缺点是它是某个昂贵的商业工具的一部分，只能通过购买该商业工具获得。我们决定吸收 SPI Fuzzer 的经验，创建一个功能有限，但开源的替代产品 WebFuzz，以满足我们特定的需求。和大多数模糊测试器一样，WebFuzz 不是一个只需点击一下，就能自动帮你完成全部工作的安全工具。WebFuzz 仅仅是一个帮助你自动化以前不得不手工进行的操作的简单工具。最终还是要靠使用者利用它开发有意义的测试，并解释得到的结果。它应该是模糊测试的起点，而不是最终解决方案。

和所有为这本书开发的工具一样，WebFuzz 是一个开源应用。因此，它提供了一个能够使用且应当使用的框架。我们鼓励读者在框架的基础上添加特性，修复缺陷，更重要的是，我们希望读者能够将你的改进分享给其他人。本章接下来的内容详细介绍 WebFuzz 的实现细节，并通过多个案例展示它的功能和局限。从本书的 Web 站点（www.fuzzing.org）上可以下载得到 WebFuzz。

[3] http://www.spidynamics.com/products/webinspect/toolkit.html

[4] http://www.codenomicon.com/products/internet/http/

[5] http://www.beyongsecurity.com/BeStorm_Info.htm

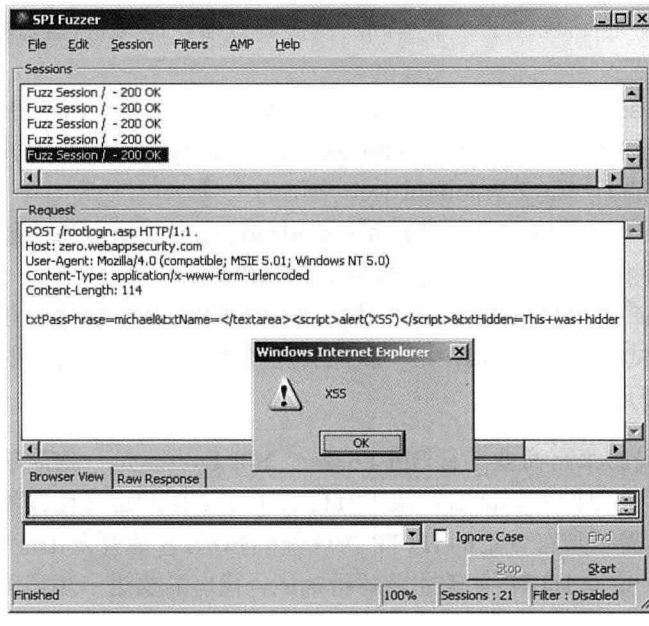


图 10.2 SPI Fuzzer

10.2 WebFuzz 的特性

在挽起袖子，开始建立自己的模糊测试器之前，我们先来回顾一下在上一章有关 HTTP 协议如何工作的章节中我们学到的知识，然后运用这些知识来确定我们的模糊测试器应该具有的特性。

10.2.1 请求

让我们从头开始。首先你需要用某种方式向 Web 应用发送请求，否则就没办法对 Web 应用进行模糊测试。通常情况下，我们使用浏览器与 Web 服务器通信，这是因为浏览器懂得“行话”（HTTP 协议），并且能够把杂七杂八的细节整合到它发出的 HTTP 请求中。然而，如果要进行模糊测试，我们就需要关心这些杂七杂八的细节了，因为我们需要能够修改请求的每个部分。因为这个原因，我们选择将原始 HTTP 请求暴露给使用者，允许使用者模糊化请求的任意部分。

图 10.3 展示了一个基本的 WebFuzz 请求。它包括以下几个部分。

- **主机**：目标机器的名字或 IP 地址，必填字段。如果不告诉 WebFuzz 将请求发到哪里，我们当然就没法对 Web 应用进行模糊测试。该字段不可模糊化。
- **端口号**：Web 应用默认运行在 TCP 的 80 端口上，当然它可以运行在任意 TCP 端口上。基于 Web 的管理控制台 Web 应用通常运行在 80 端口以外的端口上，以免干扰主服务器的运行。和主机名一样，端口号告诉 WebFuzz 把请求发送到哪里，端口号也不可模糊化。
- **超时时间**：由于我们故意发送非标准的 Web 请求，因此出现目标服务器不能及时响应的情况也很正常。所以我们有一个可由用户自定义的超时时间参数（以毫秒为单位）。当记录一个请求超时的响应时，我们需要同时记录超时发生的时间，这一点很重要，因为它表示我们的请求可能把目标应用给弄下线了，这是一个潜在的 DoS 漏洞。
- **请求头**：现在我们开始上路了。使用 Web 浏览器时，使用者可以控制目标主机名称、端口号和请求的 URI，但不能控制请求头。我们特意将请求的所有内容放在一个可直接修改的文本字段中，因为我们希望使用者能够控制请求的所有部分。简单地通过手工输入期望的请求信息到“Request Header”字段就可以手工构建请求。另外，如果你偏爱用鼠标点击操作的话，也可以通过图 10.3 所示的上下文菜单提供的标准头列表来构建请求。如果需要默认 Web 页面请求，可以从上下文菜单中选择“Default Headers”选项。

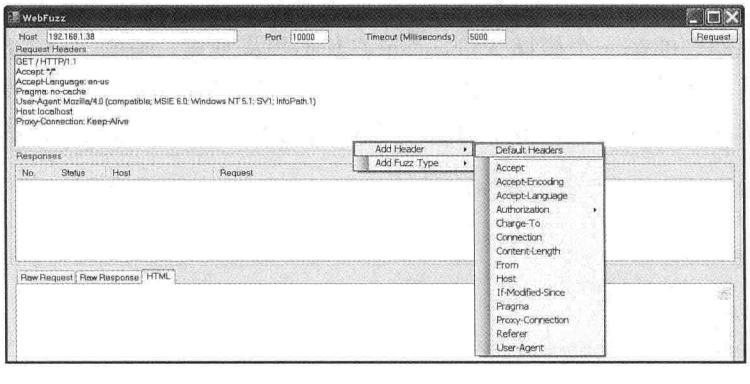


图 10.3　WebFuzz 请求

10.2.2　模糊变量

我们所说的模糊变量是指在请求中会被模糊数据所取代的区域。前面我们已经讨论过，用户能够全面控制发送至 Web 服务器的原始请求。因此，我们把模糊变量直接添

加到原始请求中，用方括号括起来的变量名进行标识（例如[Overflow]）。我们设计的创建模糊变量的函数分为两种基本类型："静态列表"类型和"生成变量"类型。"静态列表"类型从预定义的数据列表中提取模糊变量。用来发现 XSS 漏洞的模糊数据是典型的静态列表的例子。一个预定义的，由多个可能触发 XSS 漏洞的输入（例如<script>alert('XSS')</script>）组成的列表会被编译，每次注入其中的一行数据到请求当中。我们特意用外部 ASCII 文本文件维护静态列表，这样用户就可以在无需重新编译应用的情况下修改模糊变量的取值。"生成变量"类型由预定义的，或是允许接受的用户输入算法创建。溢出变量是生成变量的示例。溢出变量允许用户定义溢出时所使用的文本，文本长度，以及重复的次数。图 10.4 显示的是默认的弹出窗口，允许用户修改参数以改变溢出变量。

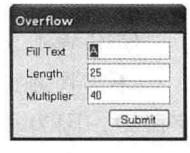


图 10.4　溢出模糊变量

为了提高效率，我们也许希望在单个请求中定义多个模糊变量。虽然有时候同时动态修改两个或更多模糊变量（如长度值以及对应的内容）比较方便，但简单起见，我们还是每次只处理一个变量。不过，在单个请求中包含多个模糊参数是完全可行的。WebFuzz 处理遇到的第一个模糊变量，而忽略其他模糊变量。当第一个模糊变量被用具体数据填充后，原始请求中的该变量就会被移除，这样 WebFuzz 就能接着处理后续的模糊变量。下面展示的 WebFuzz 请求是一个可以用来发现多个常见漏洞的请求。

```
[Methods] /file.php?var1=[XSS][SQL]&var2=[Format] HTTP/1.1
Accept: */*
Accept-Language: en-us
User-Agent: Mozilla/4.0
Host: [Overflow]
Proxy-Connection: Keep-Alive
```

10.2.3　响应

WebFuzz 捕获所有响应结果，并以原始形式记录它们。通过捕获完整的原始响应，我们能用多种方式灵活地展示响应结果。具体地说，我们让用户能够在 Web 浏览器控制台中显示原始的响应数据或是以 HTML 形式查看响应数据。图 10.5 显示的是原始响

应，图 10.6 是同样的数据显示在 Web 浏览器中的效果。能用多种形式展现响应数据十分重要，因为漏洞的线索会以多种方式表现出来。例如，响应头可能包含一个表示发生了 DoS 的状态码（比如，500 内部错误）。而 Web 页面本身可能会显示一个用户错误信息，表示发生了可能的 SQL 注入攻击。在这种情况下，用浏览器展示 HTML 时，该错误更容易被发现。

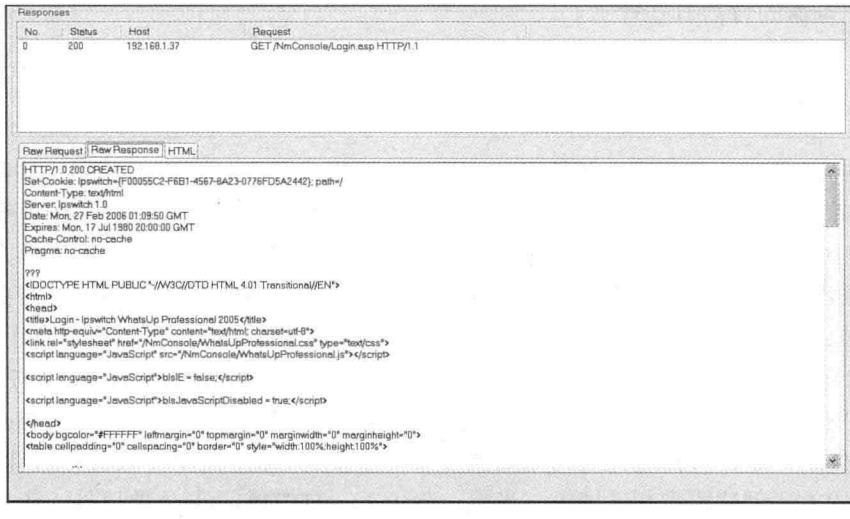


图 10.5　原始响应

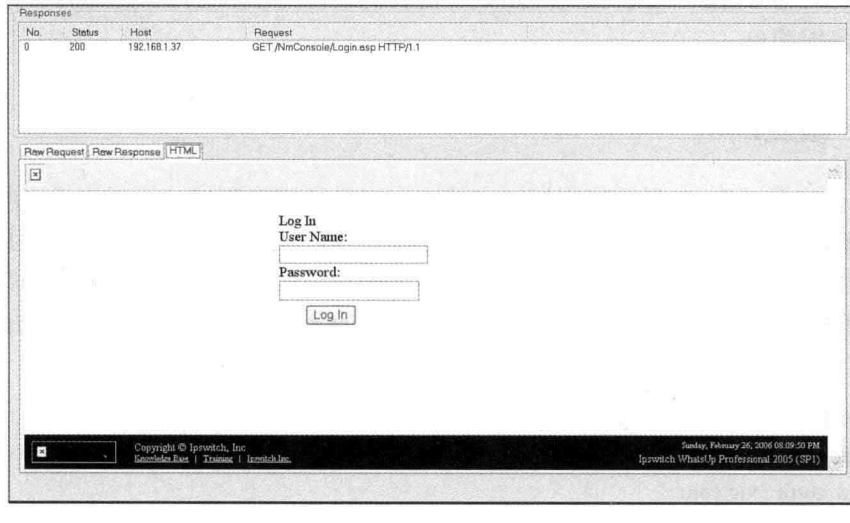


图 10.6　以 HTML 格式显示的响应

10.3 必备的背景信息

HTTP 协议为模糊测试提出了独特的挑战。这些挑战在于两方面：如何最有效地监控流量，以及更重要的，如何识别发生的异常。接下来，我们就来揭开这层窗户纸，看看 Web 浏览器的背后有些什么。

10.3.1 识别请求

WebFuzz 要求用户构建原始的 HTTP 请求，但是对于给定的 Web 应用，如何确定合适的请求？应用都有哪些页面？这些页面能够接受什么变量，怎样将变量传递给它们？在上一章中，我们讨论了如何通过手动方式，以及通过嗅探器、蜘蛛、代理等方式识别输入。在继续之前，我们希望引入一款 Web 浏览器插件，当使用 WebFuzz 并需要找出单个页面的原始请求时，该插件能够派上用场。LiveHTTPHeaders 项目[6]提供了有用的工具，帮助我们在基于 Mozilla 的 Web 浏览器中确定原始 HTTP 请求。图 10.7 展示了
145 LiveHTTPHeaders，该插件在 Firefox 侧边栏显示所有的请求以及它们对应的响应。这种方式的好处之一是，直接通过剪切粘贴就可以将捕获到的请求填入 WebFuzz，因此可以用流水线的方式创建模糊测试请求。当然，除了 LiveHTTPHeaders 项目外，还有多种可用的浏览器插件，如 Tamper Data[7]，以及同样是 Firefox 扩展的 Firebug[8]，再或者 Internet Explorer 的浏览器附加（add-on）Fiddler[9]。但我们最钟爱 LiveHTTPHeaders 工具，因为它的使用足够简单。

10.3.2 检测响应

我们在前面讨论过，目标应用发回来的响应提供了模糊测试请求所造成影响的各种线索。WebFuzz 被设计为向用户提供这些数据，但解释这些响应的工作需要用户完成。
146 手动审查所有从 Web 服务器返回来的信息不太现实，但如果只是检查响应的某些突显异常情况的部分还是可行的。如果发现了异常，用户就可以确定与异常相关的请求。在

---

6 http://livehttpheaders.mozdev.org

7 https://addons.mozilla.org/firefox/966

8 http://www.getfirebug.com

9 http://www.fiddlertool.com

WebFuzz 中，所有的响应，不管以原始形式还是 HTML 形式展现，都可以和对应的请求显示在一起，只需从 Response 窗口中选择合适的标签页即可。

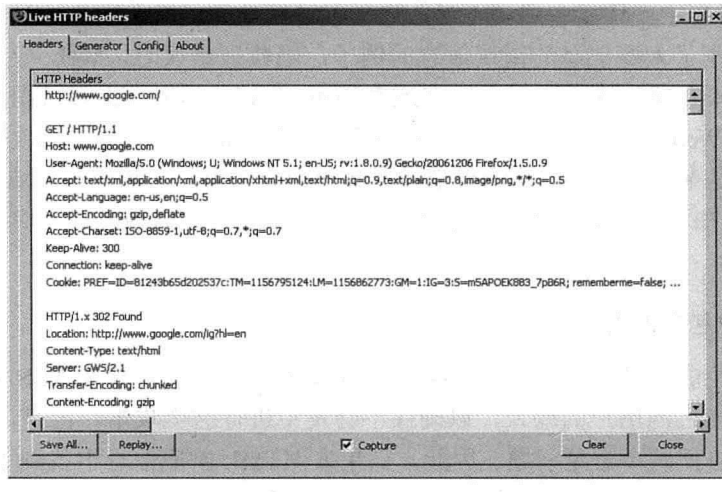


图 10.7　LiveHTTPHeaders

运行 WebFuzz 时，响应中的这些信息可以指示漏洞条件的存在：

- HTML 状态码
- 响应中的错误信息
- 响应中包含的用户输入
- 性能下降
- 请求超时
- WebFuzz 错误信息
- 处理或者未处理的异常

下面，我们分别讨论以上的每种信息，以便更好地理解为什么它们能够帮助发现漏洞。

1. HTML 状态码

我们提到过，HTML 状态码是一种重要信息，它提供了快速判断对应的请求是成功还是失败的指示。因此，WebFuzz 解析原始响应，得到状态码，然后在一个显示响应详细信息的列表中将其单独展示出来。通过 HTML 状态码信息，用户能够快速确定需要进一步详细检查的响应部分。

2. 响应中的错误信息

从设计上来说，Web 服务器一般都会在动态生成的网页中包含错误信息。如果一个 Web 服务器在生产环境中启动不当，启用了调试功能，就会发生这种情况。以下是一个典型的透露了太多信息的错误信息的例子：当验证错误时，Web 应用给出的错误信息是“密码不正确”而不是“用户名或密码不正确”。如果攻击者试图通过暴力方法强行破解某个 Web 应用的登录页面，“密码不正确”的错误消息会告诉攻击者输入的用户名存在，只是密码不正确。这使得未知参数由两个（用户名和密码）减少为一个（密码），极大地增加了攻击者进入系统的可能。在识别 SQL 注入攻击时，应用的错误信息同样特别有用。

147

3. 响应中包含的用户输入

如果动态生成的 Web 页面包含用户输入的数据，就有可能存在 XSS 漏洞。Web 应用的设计者应当过滤用户的输入，以确保不会发生进行这类攻击。但是，Web 应用没有进行合适的过滤是个常见问题。因此，如果在 HTML 响应信息中找到了 WebFuzz 提供的数据，那就表明应当测试应用中的 XSS 漏洞。

4. 性能下降

尽管通过其表现形式（直接的应用崩溃），我们很容易识别 DoS 攻击，但 DoS 漏洞则微妙得多。性能下降通常表明应用可能易于受到 DoS 攻击。请求超时是一种发现性能下降的方法，但是在模糊测试的过程中，还应当使用性能监视器检测问题，如过高的 CPU 使用率或内存使用率。

5. 请求超时

刚刚我们提到，不能忽视请求超时，因为它们可能表示存在临时或者永久的 DoS 条件。

6. WebFuzz 错误信息

WebFuzz 有它自己的错误处理方式，当某些特定的函数执行失败时，它会弹出错误信息。例如，如果目标服务器因为前一个模糊请求而下线，WebFuzz 可能会给出一个错误信息，表明无法连接到目标服务器。这意味着可能发生了 DoS 攻击。

7. 处理或未被处理的异常

当对 Web 应用进行模糊测试时，可能会在应用本身以及它所运行的服务器上都发现漏洞。因此，监视服务器的状态也很重要。尽管 Web 服务器返回的响应信息为我们

提供了发现潜在漏洞的信息，但它们并没有揭示全部问题。如果输入稍加变化，模糊测试请求极可能会导致处理或者未被处理的异常，从而导致可被利用的条件。因此，在模糊测试过程中，建议在目标 Web 服务器上连接一个单独的调试器，这样就能够识别这些异常。本书讨论过的其他模糊测试工具，如 FileFuzz 和 COMRaider，都带有内置的调试功能。Web 应用模糊测试工具并不需要调试功能，因为 WebFuzz 不需要反复地启动和中止一个应用。我们的方法是向某个 Web 应用发送一系列的模糊测试请求，服务器会持续运行并响应所有的请求，并阻止导致 DoS 的输入。

10.4　开发 WebFuzz

148

好了，理论说得够多了，下面该开始尝试构建的乐趣了。现在，让我们挽起袖子来创建一个 Web 应用模糊测试器吧。

10.4.1　思路

在设计 WebFuzz 时，我们的目标是建立一个用户友好的 Web 应用模糊测试工具。我们的目标用户是对 HTTP 有一定的理解的人，因此我们无需提供一个仅通过鼠标点击就能完成工作的方案。相反，我们想要建立一个工具，这个工具能让使用者在控制请求方面有最大的灵活性。同时，我们还希望以能够简化检测潜在漏洞的方式展现响应数据。

10.4.2　选择编程语言

为了让 WebFuzz 对用户友好，我们把它设计为图形用户界面（GUI）的应用。我们选择 C#作为开发语言，主要原因如下：首先，C#能让我们不用花多少力气，就能设计出一个看起来足够专业的 GUI；其次，C#提供了许多类帮助我们发送和接收网络流量。缺点是选择 C#就意味着我们被绑在了 Windows 平台上。然而，当对一个 Web 应用进行模糊测试时，被测试对象并不需要和模糊测试器运行在同一台机器上。因此，设计一个在 Windows 平台上运行的工具并不会限制我们只能对 Windows 平台上的目标进行模糊测试。

10.4.3　设计

如果我们将 WebFuzz 工具的所有代码都拿来分析的话，读者不打瞌睡才怪。WebFuzz 模糊测试器的主要功能包含在几个基本类中，现在我们来重点研究每个基本类

的关键功能。请记住，从本书的网站上可以下载得到所有出现在本书中的应用的完整源代码。

1. TcpClient 类

C#提供了一个用来处理 HTTP 请求和响应的类：WebClient 类。它封装了能够生成和处理网络数据的许多代码，可以极大简化应用的开发。它甚至提供了很多实现 WebFuzz 所需功能的函数，例如存取 HTTP 响应头的函数。在更底层一些的地方，C#提供了 HttpWebRequest 和 HttpWebResponse 类。尽管使用这些类需要多写一点代码，但这些类提供了更多高级特性，例如使用代理服务器。那么，在 WebFuzz 中我们用到了这些酷酷的类吗？答案是，一个都没有用。相反，我们选择使用 TcpClient 类，因为它适用于任何类型的 TCP 连接，而不仅是 HTTP。当然，正因为这样，TcpClient 也就缺少其他 Web 类封装的功能。我们为什么要这么做？难道是因为我们喜欢虐待自己，喜欢写不必要的代码吗？显然不是，这是不得已的决定。

149

编写模糊测试器时，我们面临的一个主要挑战是：我们不得不采用“非标准”的方式。因此，标准的类和函数满足不了我们的需求。我们的目标是完全控制原始的 HTTP 请求，但很不幸，C#中这么多的 Web 类却不能提供我们需要的细粒度控制。例如，考虑下面的代码：

```
WebClientwclFuzz = new WebClient();
wclFuzz.Headers.Add("blah", "blah");

Stream data = wclFuzz.OpenRead(http://www.fuzzing.org);
StreamReader reader = new StreamReader(data);

Data.Close();
Reader.Close();
```

使用 WebClient 类，只需要上面这段简单的代码就能够发出一个定制的 Web 请求。在上面这段代码中，我们创建了一个基础的 GET 请求，向其中添加了一个定制的头信息（blah : blah）。然而，在嗅探实际发送的数据时，我们发现实际发送的是下面这样的请求：

```
GET / HTTP/1.1
blah: blah
Host: www.fuzzing.org
Connection: Keep-Alive
```

读者会注意到，实际发出的请求中额外添加了两个头信息：Host 和 Connection。这就是为什么我们不能使用通常使用的 WebClient 类的原因。我们需要牺牲方便性，在更

低的层次上获得对整个过程的完全控制。因此，在 WebFuzz 的网络部分，我们选择使用 TcpClient 类。

2. 异步套接字

异步套接字或是同步套接字都可以用来实现网络功能。尽管实现异步套接字需要一些额外工作，但在 WebFuzz 中我们还是特意使用异步套接字，这是因为使用模糊测试器的时候，异步套接字方式能更好地处理可能遇到的网络问题。

同步套接字是阻塞方式的，这就意味着如果发生一个请求或者响应，主线程会暂停运行，直到通讯完成才继续运行。使用模糊测试器时，我们会故意引发一些异常情况，其中某些异常使得 Web 应用的性能下降，甚至可能使得目标应用完全下线。我们不希望 WebFuzz 为了等待某个永远不会完成的通讯而变得无法响应。异步套接字就能够帮助我们避免这个问题，因为它们是非阻塞方式的。异步套接字启动一个单独的线程处理通信，通信完成后它会调用一个回调函数来通知调用者。这样就允许其他事件不被打断地继续进行。

下面我们来看看 WebFuzz 网络部分的代码，以便更好地理解异步套接字的概念：

```
TcpClient client;
NetworkStream stream;
ClientStatecs;

try
{
    Client = new TcpClient();
    Client.Connnect(reqHost, Convert.ToInt32(tbxPort.Text));
    stream = client.GetStream();
    cs = new ClientState(stream, reqBytes);
}
catch (SocketException ex)
{
    MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK,
      MessageBoxIcon.Error);
    return;
}
catch (System.IO.Exception ex)
{
    MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK,
      MessageBoxIcon.Error);
     return;
```

```
}
IAsyncResult result = Stream.BeginWrite(cs.ByteBuffer, 0,
    csByteBuffer.Length, new AsyncCallback(OnWriteComplete), cs);

result.AyncWaitHandle.WaitOne();
```

151

在创建了典型的TCPClient和NetworkStream之后，我们调用了流方法BeginWrite()，该方法带有五个参数[10]。

- byte[] array：包含要写到网络流的数据的缓冲区；
- int offset：要发送的数据在缓冲区中的起始位置；
- intnumBytes：写入的最大字节数；
- AsynCallbackuserCallback：回调方法，通信完成后该方法会被调用；
- object stateObject：一个对象，将该异步写请求与其他请求区别开来；

AsyncWaitHandle.WaitOne() 会阻塞监听线程，直到成功发送该请求。这时，回调函数会像下面这样被调用：

```
public static void OnWriteComplete(IAsyncResultar)
{
    try
    {
        ClientStatecs = (ClientState)ar.AsyncState;
        Cs.NetStream.EndWrite(ar);
    }
    catch (System.ObjectDisposedException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
          MessageBoxIcon.Error);
    }
}
```

当我们的请求写入到网络流中以后，我们就能够收到服务器返回来的结果。

```
try
{
    Result = stream.BeginRead(cs.ByteBuffer, cs.TotalBytes,
    Cs.ByteBuffer.Length-cs.TotalBytes.
    New AsyncCallback(OnReadComplete), cs);
```

[10] http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemiofilestreamclassbeginwritetopic.asp

```
}
catch (System.IO.IOException ex)
{
    MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
      MessageBoxIcon.Error);
    ReadDone.Close();
    return;
}
```

这时，我们再次使用异步套接字，但这次我们是使用它接收从目标应用返回的响应。这次我们调用 BeginRead() 方法，它和 BeginWrite()方法带有相同的参数，这次我们使用 OnReadComplete() 作为我们的回调方法：

```
public void OnReadComplete(IAsyncResultar)
{
    readTimeout.Elapsed += new ElapsedEventHandler(OnTimedEvent);
    readTimeout.Interval = Convert.ToInt32(tbxTimeout.Text);
    readTimeOut.Enabled = true;

    ClientStatecs = (ClientState)ar.AsyncState;
    intbutesRcvd;

    try
    {
        bytesRcvd= cs.NetStream.EndRead(ar);
    }
    catch (System.IO.IOException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
          MessageBoxIcon.Error);
        return;
    }
    catch (System.ObjectDisposedException ex)
    {
        return;
    }

    cs.AppendResponse(Encoding.ASCII.GetString(cs.ByteBuffer,
      Cs.TotalBytes, bytesRcvd));
    cs.AddToTotalBytes(byteRcvd);

    if (bytesRcvd != 0)
    {
        cs.NetStream.BeginRead(cs.ByteBuffer, cs.TotalBytes,
```

```
            cs.ByteBuffer.Length - cs.TotalBytes,
                    newAsyncCallback(OnReadComplete), cs);
        }
153     else
        {
            readTimeout.Enabled = false;
            if (ReadDone.Set() == false);
                ReadDone.Set();
        }
    }
```

在 OnReadComplete() 方法的开始部分，我们创建了一个定时器（变量readTimeout），如果到了用户定义的超时时间，定时器就会调用 ReadDone.Set()方法。这使得我们能确保如果读取失败，该线程不会一直处于活跃状态，同时这种方式让终端用户可以控制超时时长。接下来的代码将收到的响应追加到我们的缓冲区。这时，需要决定是否继续等待后续数据。我们根据接收到的数据长度是否大于 0 来决定是否需要等待后续数据。如果需要等待后续数据的话，我们就再次调用 BeginRead()方法，重复上面的过程。如果不需要的话，我们就中止该线程。

生成请求

在发送请求之前，我们先要决定到底要发送什么数据。显然，发送的数据应该来自用户创建请求的 Request Header 窗口，其中的每个模糊变量[XXX]都必须被替换成真正的模糊数据。当用户点击 Request 按钮后，btnRequest_Click()方法中的代码就开始进行整个进程：

```
If (rawRequest.Contains("[") != true ])rawRequest.Contains("]") ! = true)
  rawRequest = "[None]" + rawRequest;
while (rawRequest.Contains("[") &&rawRequest.Contains(")")
{
  Fuzz = rawRequest.Substring(rawRequest.IndexOf('[') + 1, (rawRequest.IndexOf(']')
-rawRequest.IndexOf('[')) - 1);
```

我们启动一个循环来生成请求，只要在请求中遇到模糊变量，我们就不断填入用户提供的数据。然后，我们使用 case 语句，在 case 语句中决定对每个模糊变量的操作。

```
intarrayCount = 0;
intarrayEnd = 0;
Read fuzzText = null;
WebFuzz.GeneratefuzzGenerate = null;
ArrayListfuzzArray = null;
```

```
    String replaceString = "";
    String[] fuzzVariables = {"SQL", "XSS", "Methods", "Overflow", "Traversal",
"Format"};

    switch (fuzz)
    {
      case "SQL":
        fuzzText = new Read("sqlinjection.txt");
        fuzzArray = fuzzText.readFile();
        arrayEnd = fuzzArray.Count;
        replaceString = "[SQL]";
        break;
      case "XSS":
        fuzzText = new Read("xssinjection.txt");
        fuzzArray = fuzzText.readFile();
        arrayEnd = fuzzArray.Count;
        replaceString="[XSS]";
        break;
      case "Methods":
        fuzzText = new Read("methods.txt");
        fuzzArray = fuzzText.readFile();
        arrayEnd = fuzzArray.Count;
        replaceString="[Methods]";
        break;
      case "Overflow":
        fuzzGenerate = new WebFuzz.Overflow(overflowFill, overflowLength,
        overflowMultiplier);
        fuzzArray = fuzzGenerate.buildArray();
        arrayEnd = fuzzArray.Count;
        replaceString="[Overflow]";
        break;
      case "Travesal":
        fuzzGenerate = new WebFuzz.Overflow("../", 1, 10);
        fuzzArray = fuzzGenerate.buildArray();
        arrayEnd = fuzzArray.Count;
        replaceString="[Travesal]";
        break;
      case "Format":
        fuzzGenerate = new WebFuzz.Overflow("%n", 1, 10);
        fuzzArray = fuzzGenerate.buildArray();
        arrayEnd = fuzzArray.Count;
        replaceString="[Format]";
        break;
```

```
case "None":
  ArrayListnullValueArrayList = new ArrayList();
  nullValueArrayList.Add("");
  fuzzArray = nullValueArrayList;
  arrayEnd = fuzzArray.Count;
  replaceString="[None]";
  break;
default:
  arrayEnd = 1;
  break;
```

对值来自静态列表的模糊变量（“SQL”，“XSS”和“Methods”），我们创建了一个新的 Read 类的实例，并向构造函数传入包含模糊变量需要用到数据的 ASCII 文本文件的名称。另外，对生成变量（“Overflow”，“Traversal”和“Format”），我们则生成一个新的 Generate 类的实例，传入要重复的字符串，字符串总长度，以及字符串被重复的次数。

3．接收响应

接收到响应后，WebFuzz 将请求、原始响应、HTML 响应、主机名以及路径添加到各自的字符串数组中保存起来。此外，标识信息如状态码、主机名、请求等，则被添加到一个 ListView 控件中。这样，当完成模糊测试后，只需点击 ListView 控件中相应的响应条目，该响应的完整详细信息就会展示在那些标签页上的 RichTextBox 和 WebBrowser 控件中。

```
rtbRequestRaw.Text= reqString;
rtbResponseRaw.Text = dataReceived;
wbrResponse.DocumentText = html;

string path = getPath(reqString)l

lvwResponses.Items.Add(lvwResponses.Items.Count.ToString());
lvwResponses.Items[lvwResponses.Items.Count - 1].SubItems.Add(status);
lvwResponses.Items[lvwResponses.Items.Count - 1].SubItems.Add(reqHost);
lvwResponses.Items[lvwResponses.Items.Count - 1].SubItems.Add
  (requestString.Substring(0, requestString.IndexOf("\r\n")));

lvwResponses.Refresh();

requestsRaw[lvwResponses.Items.Count  - 1] = reqString;
requestsRaw[lvwResponses.Items.Count  - 1] = dataReceived;
requestsHtml[lvwResponses.Items.Count  - 1] = html;
requestsHost[lvwResponses.Items.Count  - 1] = reqHost;
```

```
requestsPath[lvwResponses.Items.Count  - 1] = path;
```

前面已经提到过，WebFuzz 不是一个只需通过鼠标点击就能工作的漏洞扫描器。它更像是一个被设计为允许懂行的人模糊化 HTTP 请求中特定部分的工具。该工具的源代码和执行文件都可以从 www.fuzzing.org 下载。

10.5　案例研究

现在我们已经基本了解了为什么要构建 WebFuzz，以及怎样构建 WebFuzz，接下来我们要转向更加重要的部分——让我们来看看 WebFuzz 是否真的管用。

10.5.1　目录遍历（Directory Traversal）

如果有用户能够突破 Web 根目录的限制，访问到不应该允许通过 Web 方式访问到的文件和文件夹时，我们就说存在目录遍历问题。这种类型的漏洞构成了保密方面的风险，同时，取决于可以被用户访问到的文件，该漏洞也可能会升级为完全的系统破坏。考虑这样的情形：目录遍历让攻击者得到了存放用户密码的文件。即使该文件是加密的，攻击者得到它之后，就有机会在线下破解它。当攻击者完成破解之后，他就可以回头使用有效的验证凭证连接到服务器。

遍历通常会涉及发送一系列的“../”字符来访问上级目录，攻击者通常会对遍历字符串进行 URL 编码，以绕开基本的过滤检测器。只要目录可浏览，就能够对所有目录进行遍历，但通常，我们还需要在目录上附加存在的文件名称。当进行目录遍历攻击测试时，建议在访问路径上添加目标服务器上默认存在的文件。例如，在 Windows 系统中，boot.ini 或者 win.ini 就是可以添加在访问路径上的好选择，一方面，它们是 ASCII 文件，当进行攻击时，如果遇到该文件，很容易辨认出来；另一方面，这两个文件存在于所有现代 Windows 操作系统中。

下面我们使用 WebFuzz 工具来发掘 Trend Micro Control Manager[11]中的目录遍历攻击问题，该目录遍历问题由 rptserver.asp 页面不正确的 IMAGE 参数输入校验引起。我们将使用 WebFuzz 向 rptserver.asp 页面发送一个 GET 请求，用模糊变量[Traversal]以及紧随其后的一个已知文件替换合法的 IMAGE 参数，在本例中，我们使用的已知文件是 win.ini。从图 10.8 所示的结果中我们看到，经过一次遍历后，确实发现了目录遍历问题。

[11] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=352

157 下面我们来试试另一个例子，这次我们来点小变化。IpswitchImail Web Calendaring[12]中的漏洞告诉我们，为了找到一个目录遍历漏洞，有时候你需要请求不存在的东西。在这个例子中，我们会发现当遍历一个不存在的 JSP 页面时存在目录遍历漏洞。再次请出 WebFuzz（详见图 10.9）。

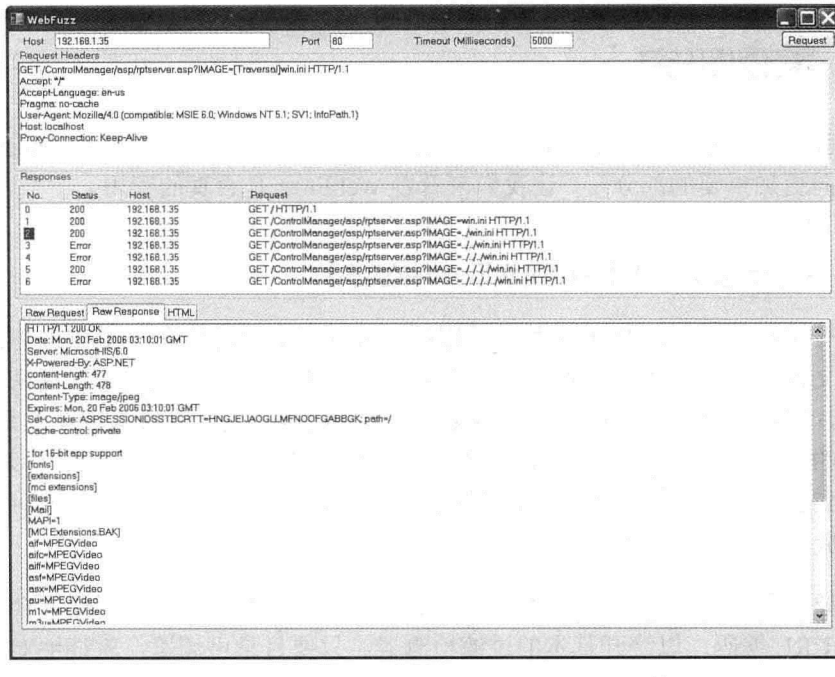


图 10.8　Trend Micro Control Manager 目录遍历

为了测试这个特定漏洞，我们向服务器发送一个 GET 方法，请求不存在的 blah.jsp Web 页面，在 URL 中紧接着 blah.jsp 的是遍历字符串和最常见的文件 boot.ini。图 10.9 显示的是经历了几次请求之后的结果，在第 5 次尝试之后……你猜对了，我们发现了漏洞。

158 ## 10.5.2　溢出

虽然在 Web 应用中很少出现缓冲区溢出，但就像控制台应用和 GUI 应用中都存在缓冲区溢出一样，Web 应用或者 Web 服务器上也可能存在缓冲区溢出，存在缓冲区溢

[12] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=242

出的原因相同：应用没有合理过滤用户提供的输入，使得非预期长度的数据溢出固定大小的缓冲区。溢出十分危险，因为它可能导致任意代码的运行。在模糊测试的过程中，如果应用接收到一个包含超长字符串的请求后发生崩溃，这就是发生溢出的最可能的标志。在模糊测试过程中，在目标应用上附加调试器，就可以确定溢出是否只限于 DoS 攻击，还是可能扩展到允许远程执行代码。

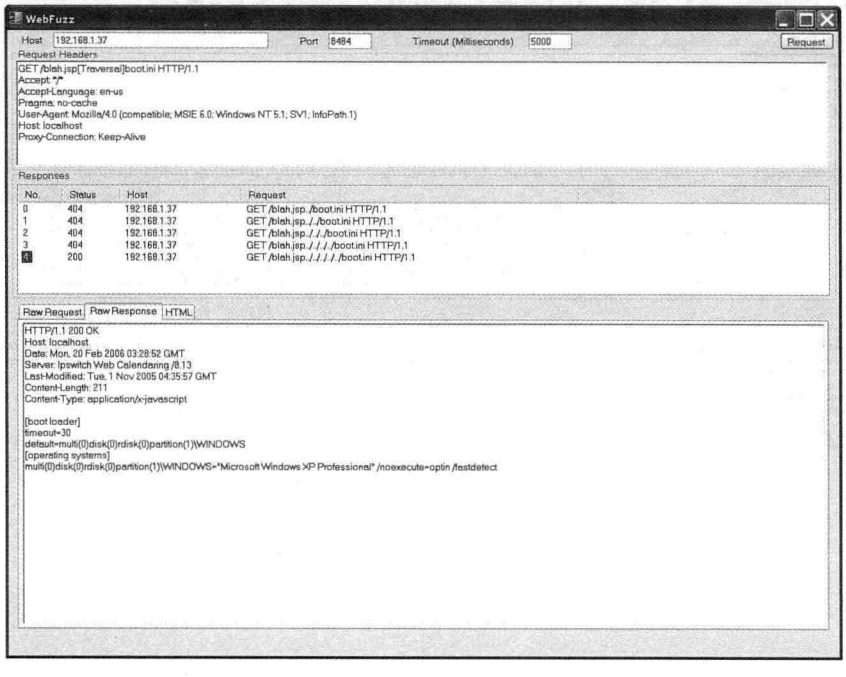


图 10.9　IpswitchImail Web Calendaring 目录遍历

我们将用一个简单的例子说明 WebFuzz 是如何检测溢出漏洞的。PMSoftware 的简单 Web 服务器（Simple Web Server[13]）中存在的溢出漏洞大概是最简单的溢出漏洞的例子了。只需要向简单 Web 服务器发送一个长 GET 请求就会导致溢出漏洞。因此我们发送下面的请求，该请求包含一个模糊测试参数 Overflow：

```
GET /[Overflow] HTTP/1.1
```

执行模糊测试时，起初，除了 404-Page Not Found（页面没找到）错误之外没有其

159

[13] http://secunia.com/advisories/15000

他任何事情发生，但从图 10.10 中我们可以看到，从第 11 个请求开始，WebFuzz 收不到任何响应了，这是为什么呢？

查看简单 Web 服务器所在的服务器时，我们很快就找到了答案，图 10.11 所示的弹窗消息显示了这个答案。

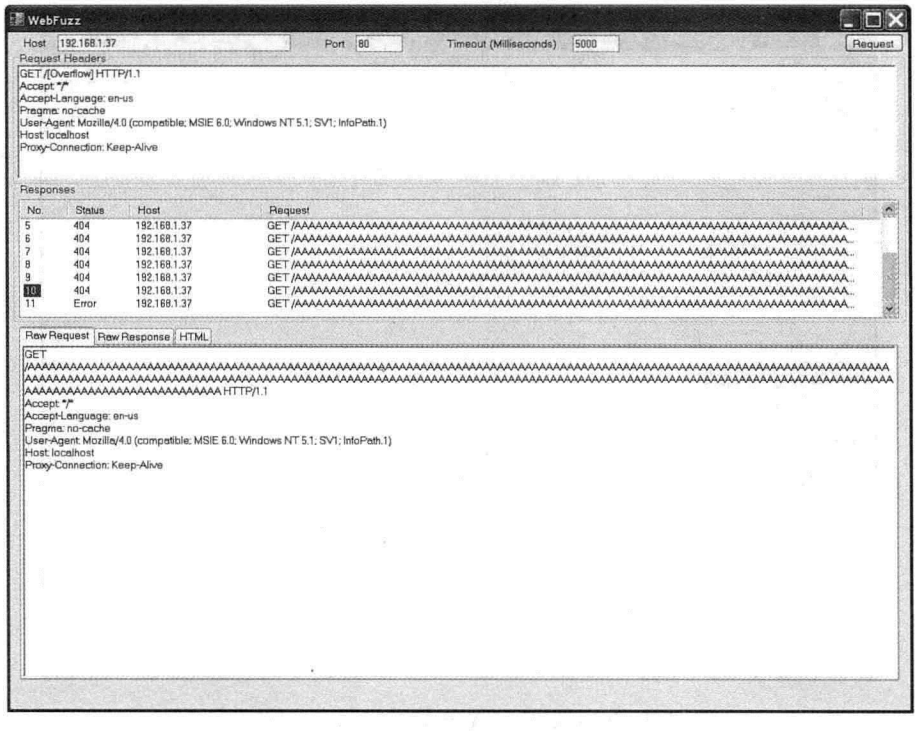


图 10.10　简单 Web 服务器的缓冲区溢出

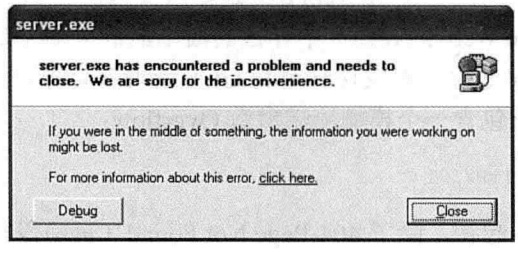


图 10.11　简单 Web 服务器的错误信息

这不是一个好现象（至少对简单 Web 服务器来说不是），关闭错误信息以后，应用

也被关闭。因此，我们至少发现了一个 DoS 攻击。这个漏洞会允许代码运行吗？答案 160
就在图 10.12，附加的调试器表明 EIP 是可以被控制的，因此有可能导致代码执行。如果在缓冲区溢出方面你是新手的话，不用过于兴奋。这个案例展示的是最幸运的情况，像这样轻而易举就能发现漏洞的情形可不多见。

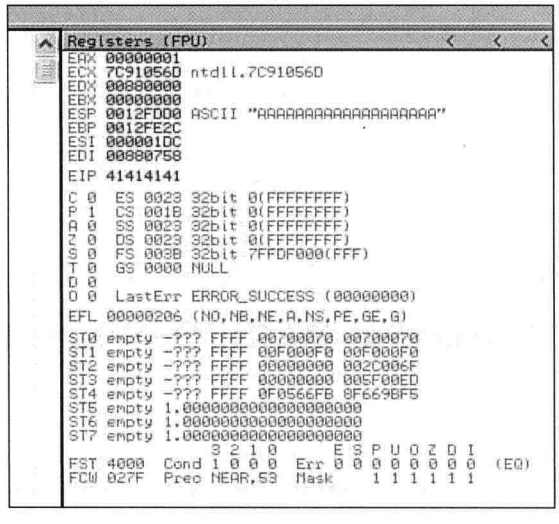


图 10.12　简单 Web 服务器溢出

10.5.3　SQL 注入

如果用户提供的数据能够影响发往后端关系型数据库的 SQL 请求，就会发生 SQL 注入攻击。这次，没有正确过滤用户数据又是罪魁祸首。在模糊测试时，应用的错误信息能够给我们提供可能存在 SQL 注入漏洞的有力线索。

通过对 IpswitchWhatsup Professional（SP1）[14]登录界面的用户名字段进行 SQL 注 161
入攻击，攻击者可以修改管理员密码，绕开所有的安全措施。我们怎样才能发现这个漏洞？使用 LiveHTTPHeaders 工具，我们能够很容易地看到用户名和口令通过 POST 请求被发送到 Login.asp 页面，如下所示：

```
POST /NmConsole/Login.asp HTTP/1.1
Host: localhost
```

[14] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=268

```
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.1)
Gecko/20060111
    FireFox/1.5.0.1
    Accept:
    text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0
.8,imager/png,*/*;q=0.5
    Accept-Language: en-us,en:q=0.5
    Accept-Encoding: gzip,deflate
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive: 300
    Connection: keep-alive
    Referer: http://localhost/NmConsole/Login.asp
    Cookie: Ipswitch={A481461B-2EC6-40AE-B362-46B31959F6D1}
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 81
```

既然我们知道了请求的正确形式，那么我们就能够拼凑出一个如下的模糊请求：

```
POST /NmConsole/Login.asp HTTP/1.1
Host: localhost

bIsJavaScriptDisabled=false&sUserName=[SQL]&sPassword=&btnLogin=Log+In
```

标准的登录失败错误信息应该是这样的："登录错误：非法用户名"。然而，当运行 WebFuzz 时，我们发现某些响应中包含了另外一些错误提示信息（如图 10.13 所示）。从错误提示信息的内容中可以清楚地看到，用户提交的数据看起来被提交到了数据库。

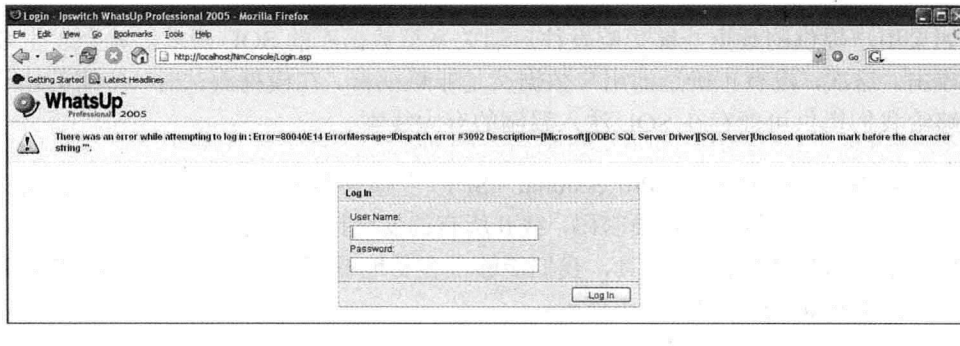


图 10.13 IpswitchWhatsUp Professional（SP1）的错误信息

然而，这并不是一个直截了当的 SQL 注入攻击。常用的绕开身份认证的技术（例

如“or 1=1”[15]）在这里并不管用，所以我们不得不编出一个合法的 UPDATE 查询来修改管理员口令。在这里我们需要数据库模式的详细信息，才能针对目标来实施量身定制的 SQL 注入攻击，但是，从哪来能获得数据库模式的信息呢？快速 Google 搜索一下就能得到答案。Google 搜索结果的第一条就是 Ipswitch 提供的它自身的数据库模式的下载链接[16]。

棒极了，但是为了得到一个有意义的查询，我们仍然还需要一些工作，而且需要构建模糊器。不过不必担心，Ipswitch 非常友好，它自己给我们提供了我们需要的查询。Ipswitch 知识库[17]给我们提供了以下命令，通过该命令我们可以将管理员密码重置为默认值。

```
osql-E -D WhatsUp-Q "UPDATE WebUser SET sPassword=DEFAULT WHERE sUserName='Admin'"
```

Ipswitch 提供的这个查询用在一个命令行工具中，用来帮助那些因为忘记密码，而将自己锁在应用之外的管理员。但是 Ipswtich 公司没想到的是，同一条请求却可以被我们用来进行 SQL 注入攻击。看起来，知识库的问答部分应该这样写。

问题（Question/Problem）：我忘记了 Web 界面里默认管理员用户的密码，怎样才能重置它？

答案（Answer/Solution）：找到一个 SQL 注入漏洞，然后运行下面的命令...

10.5.4　XSS 脚本

163

XSS 无处不在。根据 2006 年 Mitre 的调查，当年新发现的漏洞中约 21.5%都与 XSS 有关[18]，所以你可以毫不费力地找到它们。事实上，sla.ckers.org 在他们的论坛里包含了一个不断增长的关于 XSS 的“耻辱墙”[19]，让人失望的是我们看到不少大公司也出现在这份名单上。

与大多数 Web 应用漏洞一样，XSS 漏洞也是由不正确的输入校验引起的。有漏洞

---

[15] http://www.securiteam.com/securityreviews/5DP0N1P76E.html

[16] http://www.ipswitch.com/support/whatsup_professional/guides/WhatsUpDBSchema.zip

[17] http://support.ipswitch.com/kb/WP-20041122-DM01.htm

[18] http://cwe.mitre.org/documents/vuln-trends.html#table1

[19] http://sla.ckers.org/forum/read.php?3,44

的应用接收用户输入，然后不经任何过滤就将其放入动态的页面内容中。这样，攻击者就可以将客户端脚本如 JavaScript 注入到所请求的网页中。这样，攻击者就可以控制显示的网页内容，或是以受害者的身份执行一些操作。

下面我们来分析一个 XSS 模糊测试实例，我们从一个已知存在漏洞的 Web 页面开始。SPI Dynamics 在 http://zero.webappsecurity.com 上部署了一个有漏洞的 Web 应用，如图 10.14 所示，他们部署这个应用的目的是为了测试他们的 web 应用扫描器 WebInspect。默认的登录页面包含两个 Web 表单，一个是登录表单，会被提交给 login1.asp，另一个则会被提交给 rootlogin.asp 页面。第二个表单包有 XSS 漏洞。在该表单的输入字段中，显示为 Last Name 的 txtName 字段的内容会回显在 rootlogin.asp 页面中。由于用户的输入没有经过任何校验就被显示出来，因此该应用是存在 XSS 漏洞的。

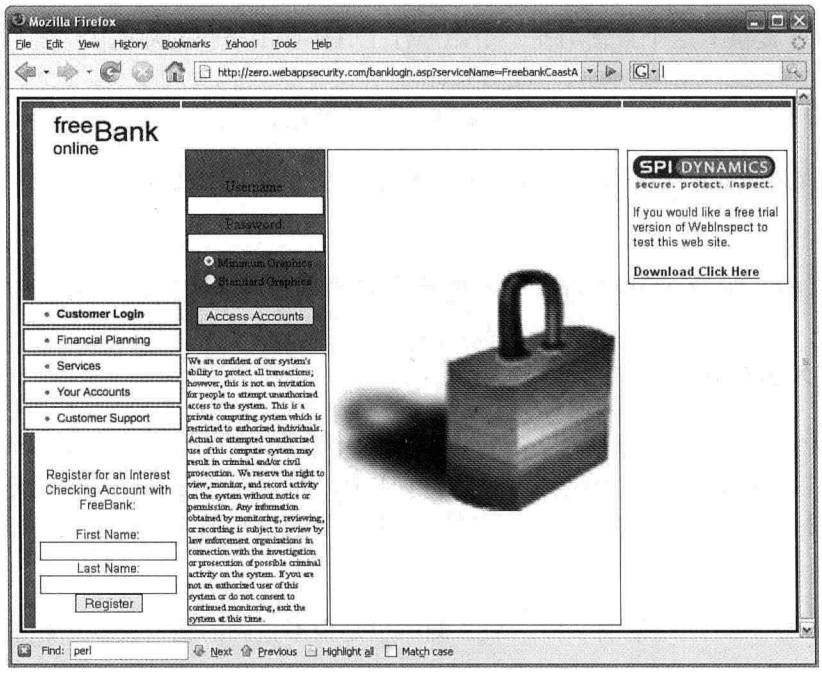


图 10.14　SPI Dynamics 的免费银行应用

一个常用的测试应用是否存在 XSS 漏洞的方法是，输入一段简单的、包含 alert 函数（会导致弹出一个窗口）的 JavaScript 片段。这是一个快捷而简陋的方法，会显示出一个简单的视觉提示，让我们知道客户端 JavaScript 可以被注入到目标网页中。因此，我们提交下面的请求测试网页是否存在 XSS 漏洞：

```
    POST /rrotlogin.asp HTTP/1.1
    Host: zero.webappsecurity.com
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.0.1)
Gecko/20060111
    FireFox/2.0.0.1
    Accept:
    text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8
,imager/png,*/*;q=0.5
    Accept-Language: en-us,en;q=0.5
    Accept-Encoding: gzip,deflate
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
    Keep-Alive: 300
    Connection: keep-alive
    Referer: http://zero.webappsecurity.com
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 72
    txtPassPhrase=first&txtName=<script>alert('Does fuzzing
    work?')</script>&txtHidden=This+was+hidden+from+the+user
```

从图 10.15 可以看到，网页上出现了一个弹出窗口。不过，当进行模糊测试时，这种方法却不是一个可行的检测机制，因为这种方法要求我们坐下来人工观察输出结果。而模糊测试最大的优点是自动化，让我们可以离开并在回来时得到结果。

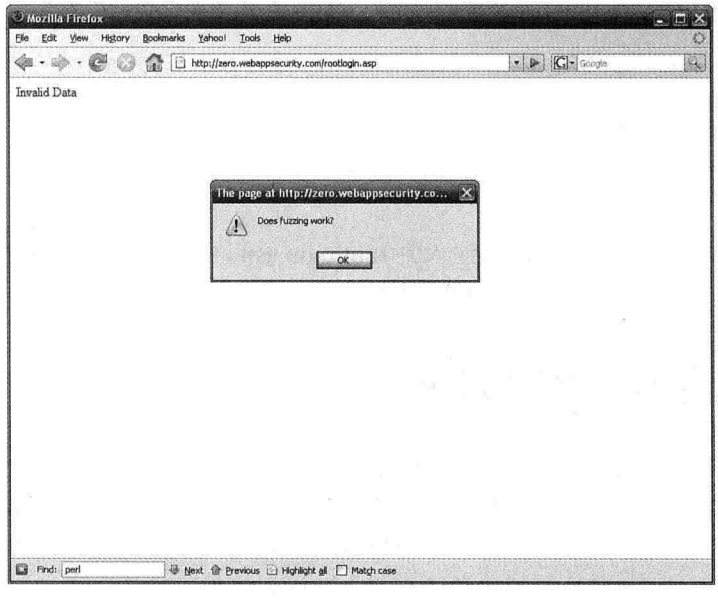


图 10.15　一个进行中的跨站脚本攻击

幸运的是，如果已知能够向网页注入代码，我们就可以有多种选择了。注入JavaScript 代码，让它在成功时给家里打电话如何？这个方案当然可行，但我们可以采用更简单的方法。客户端代码不一定非得是 JavaScript，它可以是任何浏览器可以解释的脚本文件。HTML 怎么样？HTML 是一种客户端脚本语言，比 JavaScript 更不容易被黑名单过滤掉，所以在某种意义上，它能够提供更加强大的 XSS 测试能力。一个 HTML 的 MG 标签能够让我们很容易实现“给家里打电话”的功能。我们要做的就是利用我
165 们的模糊测试器，向页面中注入一个 HTML 的 IMG 标签，标签请求一个来自本地 Web 服务器的仿页面（fake page）。完成模糊测试后，检查本地 Web 服务器的日志，如果能看到访问请求，我们就知道目标存在 XSS 漏洞了。下面，我们来试试这个方案。首先，我们需要向 WebFuzz 添加一个合适的模糊变量。鉴于 WebFuzz 被设计成从明文文本文件读取模糊变量的值，因此添加新的变量很简单，不需要重新编译模糊测试器应用。为了实施测试，我们需要将下面这行内容添加到 xssinjection.txt 文件当中：

```
%3Cimg+src%3D%27http%3A%2F%2Flocalhost%2Fblah%27%3E
```

上面这行内容是 URL 编码后的对以下图像的请求，它的目的是从本地 Web 服务器请求一个不存在的页面：

```
<imgsrc='http://localhost/blah'>
```

166 当查看我们的本地 Web 服务器日志时，我们看到了什么？

```
#Software: Microsoft Internet Information Services 5.1
#Version: 1.0
#Date: 2007-01-31 00:57:34
#Fields: time c-ipcs-method cs-uri-stem sc-status
00:57:34 127.0.0.1 GET /xss 404
```

搞定！当 WebFuzz 将图片请求注入到 rootlogin.asp 网页后，我们发现了 404-Page Not Found 日志项，证明该网页含有 XSS 漏洞。

10.6 优点与可能的改进

WebFuzz 的优点源于它提供给使用者的控制层次。即，用户可以完全控制请求的各个方面。然而，这需要用户对 HTTP 有一定程度的理解，这样才能生成有意义的，能得到结果的请求。工具的请求部分有一个可以改进的地方，那就是应该允许更复杂的模糊变量组合。例如，可以添加互相依赖的变量，这样就可以一次添加多个变量，其中一个变量的值随另一个变量的值改变。

当然，在进一步自动化 WebFuzz，让它可以覆盖整个 Web 应用方面，也有不少提升空间。添加爬虫功能可以提升 WebFuzz 的自动化能力，爬虫函数可以在初始时运行，找到所有可能的请求和结构。将 WebFuzz 的功能移植到 Web 浏览器的扩展也十分有价值，因为它允许在浏览网页时模糊化遇到的特定请求。最后，我们认为有可能能够更进一步，自动检测可以识别潜在漏洞的响应数据。例如，可以在原始响应上运行解析引擎，从响应中寻找诸如用户提供的数据这样的标志信息，它们表明有可能存在 XSS 漏洞。

好了，我们已经开始运球并将球传给你了，下面轮到你了。改进这个应用，然后将你的改进提交给我们，这样我们就能够更新版本，将你的成果分享给更多的人。

10.7　小结

167

Web 应用给我们带来了一些独特的挑战，但是毫无疑问，Web 应用是很好的模糊测试目标。希望我们的展示让你理解了将 Web 应用发布到产品环境之前，使用模糊测试发现漏洞的价值。WebFuzz 只是从概念性上演示了 Web 模糊测试，但即使还处于原始状态，WebFuzz 也能够发现漏洞。我们鼓励你下载 WebFuzz 的源代码，并且将它改进成一个更加有效的工具。

第11章

文件格式模糊测试

If this were a dictatorship, it'd be a heck of a lot easier, just so long as I'm the dictator.
——George W. Bush, Washington, DC, December 19, 2000

文件格式模糊测试是面向特定目标的特殊模糊测试方法。文件格式模糊测试的目标通常是客户端应用，如媒体播放器，Web 浏览器，Office 办公套件等。然而，文件格式模糊测试的目标也可以是服务器，例如反病毒网关扫描器，垃圾邮件过滤器，甚至是电子邮件服务器。文件格式模糊测试的最终目的是从应用解析特定类型文件的方法中找到可被利用的漏洞。

2005 至 2006 年，在现有软件中发现了大量的客户端文件格式解析漏洞，其中许多漏洞是非法组织发现的，因为在这些安全漏洞被正式公开之前，已经出现了相当多的针对这些 0day 漏洞的利用和攻击。eEye 安全研究组进行了卓有成效的工作，在他们的 Zero-DayTraker 网站发布了这些漏洞的细节[1]。大量事实表明这些安全漏洞中的大部分都是通过文件格式模糊测试发现的。这类缺陷还远没有消失，因此文件格式模糊测试仍然是一个非常有趣且“热门”的主题。

在本章中，我们将给出一些文件模糊测试的方法，并会讨论特定被测应用接受输入的各种方式。最后，我们会演示文件模糊测试器能够遇到的一些常见安全漏洞，并给出如何在实践中检测这些安全漏洞的建议。当然，我们的第一步是选择合适的目标应用。

[1] http://research.eeys.com/html/alerts/zeroday/

11.1　测试目标

和其他传统的模糊测试类型一样，通过文件格式模糊测试也能够发现许多不同类型的安全漏洞。同样的，也存在许多不同的利用安全漏洞的场景。例如，有些情况下攻击者向用户发送一个恶意文件，只有用户打开文件才会受到攻击。而另一些情况下用户只要浏览一个被攻击者控制的 Web 页就会受到攻击。最后，在有些场景下，只需要向邮件服务器或反病毒网关发送恶意电子邮件就能触发攻击。表 11.1 中提到的 Microsoft Exchange TNEF 安全漏洞就是最后一种场景。除此之外，表 11.1 还列出了其他一些文件格式安全漏洞。

表 11.1　已发现的文件格式漏洞类型和例子

应用类别	漏洞名称	报　告
Office 生产率套件	Microsoft HLINK.DLL 超链接对象库缓冲区溢出漏洞	http://www.tippingpoint.com/security/advisories/TSRT-06-10.html
反病毒扫描器	卡巴斯基反病毒引擎 CHM 文件解析缓冲区溢出漏洞	http://www.idefense.com/intelligence/vulnerabilities/display.php?id=318
媒体播放器	Winamp m3u 解析栈溢出漏洞	http://www.idefense.com/intelligence/vulnerabilities/display.php?id=377
Web 浏览器	向量标记语言中的漏洞允许执行远程代码	http://www.microsoft.com/technet/security/Bulletin/MS06-055.mspx
归档工具	WinZip MIME 解析缓冲区溢出漏洞	http://www.idefense.com/intelligence/vulnerabilities/display.php?id=377
电子邮件服务器	Microsoft Exchange TNEF 解码漏洞	http://www.microsoft.com/technet/security/Bulletin/MS06-003.mspx

读者可以发现，大多数文件格式模糊测试的目标应用会落在某个列出的类别中，有些应用甚至会落在某几个类别中。例如，许多反病毒扫描器包含解压缩文件的库，这些库使得反病毒扫描器可以像归档工具一样工作。还有一些内容扫描器可以用来分析图像文件以找到色情内容，这些程序同时可以被看作是图像查看器[2]！应用间共享通用库并不罕见，在这种情况下通用库中的安全漏洞就会影响到多个应用。例如，MS06-055 给出的安全漏洞就同时影响了 Internet Explorer 和 Outlook。

[2] http://www.clearswift.com/solutions/porn_filters.aspx

11.2 测试方法

文件格式模糊测试与其他类型的模糊测试不同，因为文件格式模糊测试通常完全运行在一台主机上。当对 Web 应用或是网络协议进行模糊测试时，可能至少需要两个系统，一个目标系统和一个运行模糊测试器的系统。在单台机器上进行模糊测试的性能更高，使得文件格式模糊测试成为一种有吸引力的发现安全漏洞的方法。

在基于网络的模糊测试中，当发生我们感兴趣的情况时，应用通常有明显的表现。在许多情况下，服务器会关闭或崩溃，无法继续提供服务。而文件格式模糊测试主要针对客户端应用进行，执行测试时，模糊测试器会持续重启和强制目标应用退出，因此，如果缺乏合适的监视器，模糊测试器无法识别发生的崩溃。在这方面，文件格式模糊测试比网络模糊测试更复杂。在进行文件格式模糊测试时，模糊测试器通常需要在每次执行时监视目标应用。一般来说，可以使用调试库动态监视目标应用中被处理的和未被处理的异常，并记录结果以供后续评审使用。概括地说，一个典型的文件模糊测试器会遵循以下步骤：

1．通过“变异”或“生成”准备一个测试用例（本章后面部分对“变异”和“生成”有更详细的介绍）。

2．启动目标应用，指示其加载测试用例。

3．监视目标应用，寻找故障（通常用调试器达成）。

4．如果发现故障，记录查找过程。如果一段时间后没有找到任何故障，终止目标应用。

5．重复上述步骤。

172

文件格式模糊测试能够通过“生成”和“变异”两种方法实现。根据我们的经验，这两种方法都很有效，但被称为“强制（Brute Force）”的“变异”方法显然更易于实现。被称为“智能强制（Intelligent Brute Force）”的“生成”方法，则需要花费更多时间来实现，不过“智能强制”方法却能够发现那些原始的强制方法所不能发现的安全漏洞。

11.2.1 强制或基于变异的模糊测试

要使用强制模糊测试方法，首先需要收集一些目标文件类型的样本。找到的文件样

本越多，测试就越全面。模糊测试器将基于这些文件运行，创建这些文件的变异文件，把它们发送给目标应用的文件解析器。取决于模糊测试器选择的方法，可以以多种形式进行变异。以字节为单位替换数据是其中一种方法。例如，遍历文件并用 0xff 替换每一个字节。另一种方法是多字节替换，例如以双字节或是 4 字节为单位进行替换。当然，向文件中插入数据而不仅仅覆写字节数据也是一种方法。然而，如果选择使用向文件中插入数据这种方法，需要注意插入的数据可能会破坏文件中的偏移数据。由于一些解析器能够快速检测无效文件并退出，破坏偏移数据会极大地降低模糊测试的覆盖率。

文件内容校验和（Checksum）也可能会破坏强制模糊测试。由于改变任意字节的值都会导致校验和失效，因此在修改文件的内容后，应用在解析文件时很可能会给出错误信息，然后在带有潜在漏洞的代码执行前自行退出。使用智能模糊测试（intelligent fuzzing）方法可以解决这个问题，我们将在 11.2.2 节中讨论智能模糊测试。当然，也可以在目标应用中停止对文件校验和进行检查。不过，取消校验和检查不是个简单的任务，通常需要对应用进行逆向工程。

为什么一旦强制模糊测试方法被实现后，就很容易被使用者所使用呢？答案很简单。因为使用者无需了解文件格式，也无需知道文件解析的工作机制。如果能够从搜索引擎或是从本地系统中找到一些样本文件，使用强制模糊测试方法进行测试的任务基本上就已经完成，接下来要做的就只是等待模糊测试器找到一些我们感兴趣的东西了。

当然，这种模糊测试方法也存在一些不足。首先，这种方法比较低效，需要花不少时间才能完成单个文件的模糊测试。以基础的 Windows Word 文档为例，一个空的 Microsoft Word 文档大约有 20KB。如果要对每个字节进行一次模糊测试，则需要创建和打开 20,480 个文件。假设处理每个文件需要 2 秒，总共需要花费 11 个小时才能完成这个文件的模糊测试。而这还只是尝试了用一个字节值进行模糊测试。如果要测试其他 254 种可能性呢？当然，使用多线程模糊测试器可以部分规避这个问题，但这个例子的确说明纯粹基于变异的模糊测试效率不高。一种简化该模糊测试方法的方式是仅聚焦在文件中可能导致预期结果的区域，例如文件头和字段头。

173

强制模糊测试的主要缺点是几乎总是会遗漏大量的功能，除非能想办法收集到一个包含所有功能和所有可能功能的样本文件集合。大多数文件格式都很复杂，需要进行大量的变换。如果考虑代码覆盖率，你会发现要彻底执行应用，需要使用者真正理解文件格式，并能够根据对文件的理解精心准备针对特定文件类型的数据，而不能简单地向应用扔一些样本文件了事。当然，通过使用文件模糊测试的“生成”方法，也即我们称为“智能强制模糊测试”的方法可以解决这个问题。

11.2.2 智能强制或基于生成的模糊测试

要使用智能强制模糊测试，必须首先研究文件规范。智能模糊测试器本质上仍是一个模糊测试引擎，因此仍然会执行强制攻击。然而，智能强制模糊测试依赖于用户提供的配置文件，这些配置文件可以使得攻击过程更加智能。配置文件通常包含描述文件类型语言的元数据。元数据可以被看作数据结构列表，它们的位置和可能的取值彼此相关。在实现层面上，可以用不同的格式表示元数据。

如果选择对一个没有任何公开文档的文件格式进行测试，测试者就不得不在构建模板前对格式规范进行进一步的研究。这些研究工作需要逆向工程的知识，但我建议读者首先从你的好伙伴 Google 开始，使用搜索方式查看是否已经有其他人做过类似工作。有些网站，例如 Wotsit's Format 网站[3]，提供了大量出色的官方和非官方文件格式文档。另一个可用的方法是比较文件类型的样本，发现其中的模式，找出文件使用的数据类型。记住，智能模糊测试器的有效性与你对文件格式的理解，以及你向所使用的模糊测试器描述文件格式的能力直接相关。在本书的第 12 章“Unix 平台上的文件格式自动化模糊测试”中，我们将在描述如何创建 SPIKEfile 工具时展示智能模糊测试实现的示例。

174

确定了测试目标和测试方法后，下一步就要针对我们选择的目标，研究合适的测试输入。

11.3 测试输入

选择了被测目标应用之后，下一步是枚举该应用支持的文件类型，文件扩展名，以及找出使文件被用不同方式解析的各种输入。同时，也应该收集和评审可用的格式规范。即使你只是想执行一次简单的强制测试，拥有文件格式的相关知识仍然很有用。由于为复杂文件类型恰当地实现解析器难度更大，在复杂文件类型中发现安全漏洞的机会无疑会更高，因此，关注复杂文件类型可能会更有收益。

让我们通过一个例子来看看怎样收集输入。归档工具 WinRAR[4]是一个可免费得到的流行的归档工具。浏览 WinRAR 的网站可以很容易地知道 WinRAR 可以处理的文件类型。在 WinRAR 网站的首页，可以找到 WinRAR 支持的文件类型列表。这个列表中列出了 zip，rar，tar，gz，ace，uue，以及其他一些文件类型。

---

[3] http://wotsit.org

[4] http://www.rarlab.com

有了 WinRAR 可以处理的文件类型列表后，需要挑选一个测试目标。有时候，最佳的挑选目标的方法是查看每个文件类型的信息，然后选择最复杂的那个。这里假设的前提是，复杂性经常会导致更多地编码错误。例如，使用大量长度标记值（length tagged value）和用户提供偏移量（user supplied offset）的文件类型会比使用静态偏移和静态长度字段的文件类型更容易出现问题。当然，当你真正掌握模糊测试之后，你会发现很多不符合这条规则的异常。理想情况下，模糊测试器最终需要面向所有可能的文件类型；因此，你选择的第一种文件类型并不一定至关重要，然而，合理选择第一种文件类型通常便于你在面向给定应用的首个模糊测试集中就能发现有意思的行为。

11.4　安全漏洞

当对不规范的文件进行解析时，一个编写得不好的应用可能会存在一大堆各种类型的安全漏洞。本节将会讨论其中的一些漏洞类别。

- DoS（崩溃或是挂起）
- 整数处理问题
- 简单栈/堆溢出
- 逻辑错误
- 格式字符串
- 竞争条件

175

11.4.1　拒绝服务（Daniel of Service，DoS）

尽管在客户端应用中拒绝服务问题并不是我们感兴趣的问题，但需要记住的是，服务端应用也可以是文件格式模糊测试地目标，而为了安全性和生产目的，这些服务器必须时刻保持可用状态。可作为文件格式模糊测试目标的服务器包括邮件服务器和内容过滤器。根据我们的经验，文件解析代码中存在 DoS 问题的最常见原因是越界读取（out of bound reads），死循环（infinite loops），空指针引用（NULL pointer references）。

导致死循环的一个常见错误是信任文件中指定其他块的位置的偏移值。如果应用不能保证这个偏移值位于当前块的前方，就有可能会导致发生死循环，使得应用不停地重复处理同一个或相同的多个块。过去在 ClamAV 中已经出现过一些这类问题的实例[5]。

[5] http://idetense.com/intelligence/vulnerabilities/display.php?id=333

11.4.2 整数处理问题

整数溢出和“符号”问题是二进制文件处理中常见的问题。我们所见到的最常见的一些问题可以用以下的伪代码表示：

```
[…]
[1] size              = read32_from_file();
[2] allocaton_size    = size+1;
[3] buffer            = malloc(allocation_size);
[4] for (ix = 0; ix < size; ix++)
[5] buffer[ix] = read8_from_file();
[…]
```

176 上面这个例子演示了一个典型的导致内存破坏的整数溢出。如果文件中 size 的值被指定为最大的无符号 32 位整数（0xFFFFFFFF），那么，第[2]行中的 allocation_size 的值由于发生整数溢出而会变成 0。这样，第[3]行的语句会导致发生一次参数值为 0 的内存分配调用，指针 buffer 会被指向进程中已被其他代码分配和使用的某个内存区块。在第[4]行和第[5]行，代码循环拷贝大小为 size 的大量数据到被 buffer 指向的缓冲区，导致发生内存破坏。

以上的特定情形不一定总是可被利用，其可利用性（exploitability）依赖于应用如何使用堆。简单地覆盖堆中的内存数据不足以获得应用的控制权。在某些情况下，类似这样的整数溢出会在堆内存被使用前触发一个与堆无关的崩溃。

当然，以上代码片段只是一个示例，用于展示解析二进制数据时可能会遇到的不正确使用整数的情况。除此之外，我们还看到过整数以许多不同方式被误用，包括经常遇到的直接比较有符号整数和无符号整数的错误。下面的代码片段展示了这类安全漏洞背后的逻辑。

```
[0] #define MAX_ITEMS 512

[…]

[1] char buff[MAX_ITEMS];
[2] int size;

[…]

[3] size = read32_from_file();
```

```
[4] if (size > MAX_ITEMS)
[5] { printf("Too many items\n"); return -1; }
[6] readx_from_file(size, buff);

[…]

/* readx_from_file: read 'size' bytes from file into buff */
[7] void readx_from_file(unsigned int size, char *buff)
{
[…]
}
```

如果 size 的值为负数，这段代码会导致一个基于栈的溢出。这是因为在第[4]行的比较中，size（在第[1]行中定义）和 MAX_ITEMS（在第[0]行中定义）都被当成有符号数处理，例如，–1 比 512 小。然而，在随后的代码中，当 size 在第[7]行的函数中被用于拷贝的边界值时，它被当成无符号整数。值“–1”会被解释成 42949672954294967295。当然，这个例子展示的问题并不一定能够被用来攻击系统，但在许多情况下，取决于 readx_from_file 函数的实现方式，通过将栈中保存的变量和保存的寄存器作为目标，这很可能会成为一个可被利用的安全漏洞。

177

11.4.3　简单的栈和堆溢出

一直以来，简单的栈和堆溢出问题已经被很好地理解，并被观察到许多次了。该问题的一个典型场景如下：在堆或是栈上分配一个固定大小的缓冲区。接下来，从文件中拷贝超过缓冲区大小的数据时没有执行边界检查。在某些情况下，可能会有一些边界检查，但边界检查实现得并不正确。当发生拷贝时，内存被破坏，往往还会导致执行某些攻击者指定的代码。关于该类漏洞更详细的信息，请参考“Shell 编程者手册：发现和利用安全漏洞”[6]。

11.4.4　逻辑错误

在某些文件格式的设计中，也可能存在可被利用的逻辑错误。虽然本书作者在研究文件格式漏洞时并没有发现逻辑类型的错误，但 MS06-001 中提到的 Microsoft WMF 安全漏洞就是一个极好的这类漏洞的例子[7]。这类漏洞不由典型的溢出导致。实际上，不

[6] ISBN-10: 0764544683

[7] http://www.microsoft.com/technet/security/Bulletin/MS06-001.mspx

需要任何类型的内存破坏，这类漏洞就允许攻击者直接执行用户提供的位于任意位置的代码。

11.4.5 格式字符串

虽然格式字符串安全漏洞几乎已经绝迹（尤其是在开源软件中），但它们仍然值得一提。我们说“几乎已经绝迹”而不是“已经绝迹”，是因为不是所有的程序员都像US-CERT的伙计们一样有安全意识，US-CERT的伙计们建议为了软件安全，应该不使用“%n”格式字符串标识符[8]。

178

但说真的，在我们的亲身经验中，当对文件格式进行模糊测试时，我们的确找到了一些格式字符串相关的问题。一部分这类问题位于Adobe[9]和RealNetworks[10]产品中。利用格式字符串问题的有趣之处在于，能够使用这些漏洞泄漏内存，帮助利用漏洞。当然，不幸的是，试图使用不规范的文件对客户端进行攻击几乎是不可能得逞的。

11.4.6 竞争条件（Race Condition）

尽管人们通常并不认为竞争条件会引起文件格式安全漏洞，但还是有一些人这么认为，而且以后可能还会有更多人这么认为。这类安全漏洞的主要目标应用是复杂的多线程应用。我们不愿意老是针对某个特定产品，但一提到竞争条件引发的安全漏洞，我们首先想到的就是微软的 Internet Explorer。Internet Explorer使用未初始化的内存，以及使用其他线程正在使用的内存导致的安全漏洞可能会不断出现。

11.5 检测错误

对文件格式进行模糊测试时，通常要启动目标应用的多个实例。在运行过程中，一些实例会挂起，最后不得不被中止；一些实例会发生崩溃；另一些实例会自行退出。我们面临的挑战在于确定何时会发生一个被处理或未被处理的异常，确定这些异常在什么情况下可被攻击者利用。模糊测试器能够利用下面这些信息找出进程相关的更多信息。

---

8 http://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/guidelines/340.html

9 http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=163

10 http://labs.idefense.com/intellegence/vulnerabilities/display.php?id=311

- **事件日志**：在 Microsoft Windows 操作系统上，可以通过事件查看器（Event Viewer）应用访问事件日志。对我们而言，事件日志并不是特别有用，因为当在模糊测试任务中启动数百个进程时，很难把某个日志条目与特定进程关联起来。
- **调试器**：最好的发现已被处理和未被处理异常的方法，是在开始模糊测试前向目标应用附加调试器。错误处理例程通常会阻止模糊测试导致的明显错误征兆出现，但通过调试器通常能够检测到这些错误。调试器除了可用于检测错误外，还有更多高级技巧，我们会在第 24 章“智能错误检测”中介绍部分高级技巧。
- **返回码**：捕获并测试应用的返回码也是一种可用的检测技术，虽然返回码不能像调试器一样给出精确信息，但返回码能够快速发现应用为什么会终止。至少在 UNIX 上，通过返回码可以发现哪个信号导致了应用终止。
- **调试 API**：与使用第三方调试器相比，在模糊测试器中实现一些基本的调试功能通常更合适和更有效。例如，对一个进程而言，我们感兴趣的是它终止的原因，终止时的当前指令是什么，此时的寄存器状态，以及诸如栈指针和一些寄存器指向的内存区域的值。通常，实现这些并不困难，但实现这些却能帮助我们在分析崩溃的可利用性时节省大量时间。在后续章节中，我们将探索这种方法并给出一个微软 Windows 平台上的调试器创建框架，该框架名为 PyDbg，是一个简单且可重用的框架。PyDbg 是 PaiMei[11]逆向工程框架的一部分。

179

一旦某个测试用例引发了故障，确认保存了你使用的各种监视错误的方法收集到的信息，以及保存了实际触发崩溃的文件。记录测试用例元数据同样重要。例如，如果模糊测试器正在使用第 42 个模糊测试值对文件的第 8 个模糊变量字段进行模糊测试，你可以将记录文件命名为 file-8-42。某些情况下，我们也许需要导出内核文件（core file）并保存。如果模糊测试器使用调试 API 捕获信号的话，就能做到这一点。在后面的两章，读者能够找到这些特定实现细节的描述。

11.6　小结

虽然文件格式模糊测试是一个较窄领域内的模糊测试方法，但该领域内仍然有大量的测试目标和大量的攻击方法。在本章中，我们不仅讨论了传统的客户端文件格式漏洞，

[11] http://www.openrce.org/downloads/details/208

还讨论了一些真实的远程场景，例如反病毒网关和邮件服务器。尽管阻止和检测基于TCP/IP 的网络攻击吸引了越来越多的关注，但文件格式漏洞仍然是一个有价值的用于渗透内部网络段的武器。

现在你已经知道要寻找哪些类型的安全漏洞，要选择哪些应用作为目标，以及如何对这些目标进行模糊测试了，接下来让我们探索如何实现文件格式模糊测试工具。

第12章

UNIX平台上的文件格式自动化模糊测试

"I am the commander -- see, I don't need to explain -- I do not need to explain why I say things. That's the interesting thing about being president."

——George W. Bush, as quoted in Bob Woodward's Bush at War

文件格式漏洞既可以在客户端被利用，比如 Web 浏览器和 Office 套件，也可以在服务端被利用，比如扫描电子邮件的反病毒网关。对于可在客户端被利用的文件格式漏洞而言，漏洞的严重性与受影响客户端的使用广泛性直接相关。例如，考虑严重程度的话，由于微软 Internet Explorer 的广泛性，影响 Internet Explorer 的 HTML 解析漏洞是最严重的文件格式漏洞。相反，仅在发生在 UNIX 平台上的客户端安全漏洞由于其传播的局限性，并不那么受到关注。

虽然如此，我们还是会在本章中介绍两个 Unix 平台上的模糊测试器：notSPIKEfile 和 SPIKEfile，这两个模糊测试器分别实现了基于变异和基于生成的文件格式模糊测试。在本章中，我们首先会讨论这两个工具的特性及其不足。然后，我们将深入开发过程，阐述这两个工具的开发方法，并提供关键的代码片段。此外，我们还会简要回顾一些基础的 UNIX 概念，例如我们感兴趣和不感兴趣的信号，以及僵尸进程（zombie processes）。最后，我们将讨论这两个工具的基本用法，解释选择开发语言的原因。

182

12.1 notSPIKEfile 和 SPIKEfile

我们开发这两个工具的目的是演示 UNIX 平台上的文件格式模糊测试，这两个工具分别被命名为 SPIKEfile 和 notSPIKEfile。正如名字描述的那样，它们分别基于 SPIKE[1] 和不基于 SPIKE。以下是它们已实现的一些关键特性：

- 集成了最小化的调试器，它能检测已被处理和未被处理的信号，并能转储（dump）内存和寄存器状态。
- 能够在中止目标进程前等待用户指定的延时，实现完全自动化的模糊测试。
- 包含两个不同的针对二进制或 ASCII 可打印数据类型的启发式模糊测试数据库。
- 有一个易扩展的启发式模糊测试数据库，该数据库存储了一些历史上发生过的漏洞。

12.1.1 不具有的特性

notSPIKEfile 和 SPIKEfile 工具中缺少一些对使用者有用的功能。

- **可移植性**：因为工具集成的调试器基于 x86 平台设计和开发，使用 Linux ptrace，因此不能在 x86 架构之外或 Linux 操作系统之外运行。然而，如果有其他可用的调试器和反汇编库，让工具兼容其他平台和操作系统并不困难。
- **智能负载监视**：虽然用户可以指定用多少个进程进行模糊测试，但目前该工具仍完全依赖用户决定系统负载是否过高。

显然，这一小节展示的这些不被包含的特性说明我们在开发过程中有一些折中的决定。工具的作者耍了一个古老的把戏，号称把这些改进作为练习留给读者。下面让我们看看构建这些工具背后的开发过程。

183

12.2 开发过程

本章关注的不是如何使用模糊测试器，而是如何实现模糊测试器。本节展示了 Linux 下文件格式模糊测试器的设计和开发细节。在开发文件模糊测试器时，我们可以放心地假设将要开发的模糊测试器会与被测目标运行在相同的系统上。我们的设计包括三个独

[1] http://www.immunityinc.com/resources-freesoftware.shtml

立的功能组件。

12.2.1　异常监测引擎

这部分代码负责检测被测目标的未定义的和潜在不安全的行为。如何实现这个功能模块？有两种基本的方法：第一种方法非常简单，监视应用收到的所有信号。这种方法可以检测到某些漏洞，例如导致非法内存引用的缓冲区溢出，但却没法检测到另一些漏洞，例如，元字符注入（metacharacter injection）或是逻辑漏洞。假设应用将攻击者输入的值直接传给 UNIX 系统函数，攻击者就可以通过使用 shell 元字符（shell metacharacter）执行任意程序，而不会导致任何类型的内存访问违例。尽管这是个危及主机安全性的漏洞，但由于这些漏洞根本不涉及内存破坏，因此，我们决定不关注这类缺陷，因为花费精力检测这些漏洞实在不划算。

对于那些愿意探索逻辑缺陷的人，一个更好的方法是使用 C 库函数钩子（LIBC），监视由模糊测试器提供的，被以非安全方式传递给 open，create，system 调用的值。

我们的设计决策是使用系统的 ptrace 调试接口实现简单的信号监测。虽然只用等待应用的响应就能知道是否发生出现了导致应用终止的信号，但如果要检测到应用内部处理的信号，就需要一些额外工作。这是因为异常监测引擎采用的方法严重依赖于系统调试接口，在我们的案例中，也就是 ptrace 调用。

12.2.2　异常报告（异常监测）

发现异常后，一个优秀的模糊测试器应该报告一些有用的信息，让使用者知道确切发生了什么。模糊测试器至少应该报告触发异常的信号。理想情况下，模糊测试器还应该报告进一步的细节，如恶意指令、CPU 寄存器状态、堆栈转储等。本章实现的这两个模糊测试器都能够产生细节报告。为了得到这些细节信息，我们必须借助 ptrace 系统调用来收集数据。此外，如果想要向用户报告恶意指令，我们还必须借助一个库 184
来反汇编指令。我们需要的库应该能够把包含数据的缓冲区转换为表示 x86 指令的字符串。在实际开发中，我们选用 libdisasm[2]库，原因是该库工作良好，接口简单，而且似乎也是 Google 搜索引擎的推荐——Google 搜索引擎把该库放在搜索结果的最前面。另外，libdisasm 自身带有一些例子，我们可以从中剪切和粘贴代码，非常方便。

---

[2] http://bastard.sourceforge.net/libdisasm.html

12.2.3 模糊测试核心引擎

文件格式模糊测试器的核心在于控制使用哪些异常数据（malformed data），以及在什么位置插入这些数据。在 notSPIKEfile 和 SPIKEfile 这两个工具中，实现这些功能的代码并不相同，因为 SPIKEfile 利用了 SPIKE 中已有的代码来实现这部分功能。除此之外，这两个工具的其余代码非常相似。

正如你猜测的那样，SPIKEfile 和 SPIKE 使用了同样的模糊测试引擎。该引擎需要一个模板，即 SPIKE 脚本，用来描述文件格式。在模板的基础上，SPIKE 组合合法和非法数据，“智能地”生成文件格式的不同变体。

在 notSPIKEfile 中，模糊测试引擎的功能则有限得多。用户必须为模糊测试器提供一个合法的目标文件。然后，模糊测试引擎使用模糊值数据库，在文件的不同部分产生变异。被变异的值分成两类：二进制值和字符串值。二进制值具有任意长度，可以取任意值，但通常用于表示通用的整数字段。字符串值，顾名思义，用来表示字符串数据，包含长字符串、短字符串、带格式描述符的字符串，以及所有类型的其他异常值，例如文件路径、URL，以及你能想到的其他类型的各种字符串。从 SPIKEfile 和 notSPIKEfile 的源代码中可以得到值类型的完整列表。但为了帮助读者入门，表 12.1 提供了一个简要列表，简单地解释了一些常用的模糊字符串。尽管表 12.1 不够详尽，但它应该能够帮助你需要考虑的输入类型。

185

表 12.1 一些通用的模糊字符串及其特征

字 符 串	特 征
“A” x 10000	长字符串，能够导致缓冲区溢出
“%n%n” x 5000	带有百分符号的长字符串，能够导致缓冲区溢出或是触发格式字符串漏洞
HTTP:// + “A” x 10000	合法的 URL 格式，能够在 URL 解析代码中触发缓冲区溢出
“A” x 5000 + “@” + “A” x 5000	合法的电子邮件地址格式，能够在电子邮件地址解析代码中触发缓冲区溢出
0x20000000,0x40000000, 0x80000000,0xffffffff	可以触发整数溢出的许多整数中的一些。在这里你可以非常有创造性。考虑使用 malloc(user_count* sizeof(struct blah))的代码，也考虑不必检查上溢和下溢就进行自增或是自减整数的代码
“../”x 5000 + “AAAA”	能够在路径或是 URL 地址解析代码中触发溢出

模糊测试模板中可以包含任意多个模糊字符串。我们要强调的是，需要在模板中包含所有可能会被特殊解析或是可能导致例外条件的值。也许在一个大模糊字符串后是否添加 HTML 后缀，就是导致模糊测试器能否找到潜在 Bug 的原因。

在 12.3 节中，我们将探索一些从 SPIKEfile 和 notSPIKEfile 工具中找到的更加有趣，与模糊测试更相关的代码片段。

12.3　有意义的代码片段

本节将研究这两个模糊测试工具共有的核心功能代码片段。让我们从复制和跟踪一个子进程的基础方法开始。下面摘录的代码片段用 C 语言编写，两个模糊测试器都会用到这段代码。

```
[…]
    if ( !(pid = fork()) )
    { /* 子进程 */
        ptrace (PTRACE_TRACEME, 0, NULL, NULL);
        execve (argv[0], argv, envp);
    }
    else
    { /* 父进程*/
        c_pid = pid;
monitor:
        waitpid (pid, &status, 0);
        if ( WIFEXITED (status) )
        { /* 退出程序 */
            if ( ! quiet )
                printf ("Process %d existed with code %d\n", pid, WEXITSTATUS
(status));
            return(ERR_OK);
        }
        else if (WIFSIGNALED (status) )
        { /* 程序接收到特定信号，退出 */
            printf("Process %d terminated by unhandled signal %d\n", pid,
WTERMSIG(status));
            return(ERR_OK);
        }
        else if (WIFSTOPPED (status) )
        { /* 程序接收到特定信号，退出 */
            if ( ! quiet )
                frpintf (stderr, "Process %d stopped due to signal %d (%d) ",
pid, WSTOPSIG (status), F_signum2ascii (WSTOPSIG (status)));
        }
        switch ( WSTOPSIG (status) )
        { /* 下面这些信号是我们通常情况下感兴趣的所有信号 */
```

```
                case SIGILL:
                case SIGBUS:
                case SIGSEGV:
                case SIGSYS:
                    printf("Program got interesting signal… \n");
                    if ( (ptrace (PTRACE_CONT, pid, NULL, (WSTOPSIG (status) ==
SIGTRAP)? 0 : WSTOPSIG (status))) == -1 )
                    {
                        perror("ptrace");
                    }
                    ptrace(PTRACE_DETACH, pid, NULL, NULL);
                    fclose(fp);
                    return(ERR_CRASH); /* it crashed */
                }
    /* 回传信号并继续跟踪 */
                if ( (ptrace (PTRACE_CONT, pid, NULL, (WSTOPSIG (status) == SIGTRAP)?
0 : WSTOPSIG (status))) == -1)
                {
                    perror("ptrace");
                }
                goto monitor;
            }
            return(ERR_OK);
        }
```

187

主进程（父进程）从复制生成目标应用的一个新进程开始。新生成的进程（子进程）调用 ptrace，向 ptrace 发送 PTRACE_TRACEME 请求，请求父进程对其进行跟踪。随后，子进程运行目标应用，而父进程则像负责任的父母一样，关注子进程，检查是否发生不合适的事情。

由于子进程使用了 PTRACE_TRACEME 请求，因此，作为父进程运行的模糊测试器能够接收到发送给子进程的所有信号。甚至，当子进程调用 exec 函数族中的函数时，父进程都能收到信号。父进程中的这个循环简单明了但作用强大。模糊测试器循环接收发送给子进程的每一个信号，然后根据信号和子进程的状态采取不同的动作。

例如，如果子进程停止运行，这表示程序没有退出，但收到了一个信号并在等待父进程决定是否允许它继续。如果该信号是内存破坏的征兆，模糊测试器就将该信号转给子进程，假设该信号将导致子进程终止并报告该结果。如果信号无害，或无关痛痒，模糊测试器就将其传给应用而不对其进行任何监视。

模糊测试器同时也检查子进程是否已经终止。如果程序发生了崩溃，你可能会问：

我们怎么知道该不该关注？因为在应用实际终止前我们已经拦截到了所有我们感兴趣的信号，因此，我们完全可以知道这个程序是由于自然行为，还是由于我们不感兴趣的信号而终止的。这也说明了理解你需要关注哪些信号何等重要。想要查找 DoS 漏洞的人会对浮点异常感兴趣，而另一些人可能只对实际发生的内存破坏问题感兴趣。还有一些人可能会关心中止（abort）信号，因为在新版本的 GLIBC 库中，中止信号是堆破坏的指示器。我们已经知道，在某些情况下，堆破坏检查可能导致执行任意代码[3]。

12.3.1　UNIX 中常见的我们可能感兴趣的信号

表 12.2 提供了安全漏洞研究者在模糊测试中可能会感兴趣的信号列表，并解释了大家为什么对这些信号感兴趣。

表 12.2　在 UNIX 下执行模糊测试时需要关注的信号

需要关注的信号	含　义
SIGSEGV	无效内存引用。成功的模糊测试最通常的结果
SIGILL	非法指令。这是内存破坏的一个可能的副作用，但是很罕见。它经常因为程序计数器被破坏，指向数据指令之间所导致（？？？）
SIGSYS	错误的系统调用。这也是一个内存破坏可能带来的副作用。但很罕见（实际上，非常罕见）。这和 SIGILL 产生的原因相同
SIGBUS	总线错误。通常由某些形式的内存破坏导致。由不正确的内存访问导致。通常在 RISC 机器上更常见一些，因为 RISC 机器的对齐需求。在大多数 RISC 实现上，非对齐存储和加载会导致 SIGBUS
SIGABRT	通常由中止的功能调用产生。这经常会让人感兴趣，因为 GLIBC 检测到堆终端后会中止

12.3.2　我们不感兴趣的信号

与表 12.2 相对应，表 12.3 描述了在模糊测试中通常会出现，但安全漏洞研究者不那么感兴趣的信号。

表 12.3 中我们提到了 SIGCHLD 信号，接下来我们来讨论一个常见的场景，该场景下我们没有对 SIGCHLD 信号进行合适的处理。在 12.4 节中，我们将解释了什么是僵尸进程，以及如何正确处理子进程才能避免出现僵尸进程。

[3] http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

表 12.3 在 UNIX 下执行模糊测试时我们不感兴趣的信号

我们不感兴趣的信号名称	含 义
SIGCHLD	一个子进程退出
SIGKILL，SIGTERM	进程被杀掉
SIGEPE	浮点异常
SIGALRM	计时器过期

189

12.4 僵尸进程（Zombie Process）

僵尸进程是父进程创建并且已经执行完成（例如，已经退出）的进程，但由于父进程没有调用 wait 或是 waitpid 获取它的状态，导致该进程变成了僵尸进程。僵尸进程的信息会一直存在于内核中，直到父进程调用 wait 和 waitpid 后，内核中该进程的信息才会被释放，这时进程才会真正结束。图 12.1 展示了我们模糊测试器创建的进程的生命周期。

当编写一个使用 fork 生成子进程的模糊测试器时，必须确认父进程使用 wait 或 waitpid 接收所有子进程的结束信息。如果发生了遗漏，子进程就会成为僵尸进程。

notSPIKEfile 的早期版本存在一些缺陷，随着测试的进展，notSPIKEfile 中的活跃进程数量会逐渐减少，直到模糊测试达到死锁状态。例如，假设用户指定使用 8 个进程进行模糊测试，随着时间流逝，活跃进程会慢慢减少到一个。这个缺陷的原因是因为粗心的工具作者犯了两个错误。notSPIKEfile 原先的设计完全依赖于 SIGCHLD 信

190

号，该信号在子进程结束后发送给进程。然而，在应用这种设计方式时，程序中有些地方会漏掉 SIGCHLD 信号。此外，当子进程执行完成后，程序中的某些地方没有成功地减少活跃进程计数，这就导致了模糊测试器实际启动的进程数逐渐减少。

当发现导致缺陷的原因后，很容易就能解决掉这些缺陷。在使用 WNOHANGE 标志将阻塞变为非阻塞时，所有的 wait 和 waitpid 调用都可能出现问题。因此，当 wait 和 waitpid 调用返回时，需要检查返回状态，确定进程是否真的执行完成。这样，当进程完成后，当前活跃进程的数量计数一定会减少 1 个。

SPIKEfile 没有提供一次启动多个应用的选项，这极大地简化了设计和实现。由于一次只会有一个 SIGCHLD 信号返回，因此我们不需要担心错失 SIGCHLD 信号。

SPIKEfile 是基于 SPIKE 的，因此，我们只需要向其中添加一两个源文件，使得它可以处理文件输入和输出，而不是只能处理网络输入和输出即可。在快速查看 SPIKE

中 TCP/IP 支持的实现后，我们发现在 SPIKEfile 中加入对文件输入和输出的支持很简单。下面的代码是我们添加的 filestuff.c 源文件的主要部分。

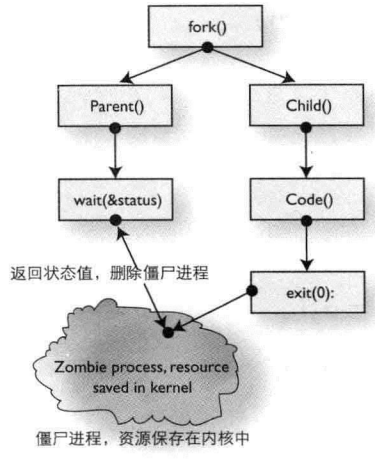


图 12.1　复制的子进程的生命周期

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include"filestuff.h"
#include"spike.h"

externstruct spike *current_spike;

int
spike_fileopen (const char *file)
{
    int fd;
    if ((fd =
         open (file, O_CREAT | O_TRUNC | O_WRONLY,
            S_IRWXU | S_IRWXG | S_IRWXO)) == -1)
      perror ("fileopen::open");
    return current_spike->fd = fd;
    current_spike->proto = 69;  /* 69 表示文件，0-68 由 ISO 模糊测试标准保留*/
    }

int
```

```
spike_filewrite (uint32 size, unsigned char *inbuffer)
{
  if (write (current_spike->fd, inbuffer, size) != size)
    {
      perror ("filewrite::write");
      return -1;
    }
  return 1;
}

void
spike_close_file ()
{
  if (current_spike->fd != -1)
    {
      close (current_spike->fd);
      current_spike->fd = -1;
    }
}
```

191

将这个源文件加入 Makefile 中，所有用过 SPIKE 的人就都能够用它来进行文件模糊测试了。如果你想知道我们向 SPIKE 中添加了哪些源文件，表 12.4 列出了这些文件的名字。

表 12.4 为了创建 SPIKEfile 引入 SPIKE 的文件列表

文 件 名	目 的
filestuff.c	包含打开文件和写文件的函数
util.c	包含许多在 notSPIKEfile 和 SPIKEfile 工具之间共享的代码。它包含了 ptrace 封装，主要的 F_execmon 函数，以及其他一些有用的函数
generic_file_fuzz.c	这是主要的 SPIKEfile 源代码。它包含了 main 函数
include/filestuff.h	filestff.c 文件的头文件
Libdisasm	当发生崩溃时用于反汇编 x86 指令的库

12.5 使用注意事项

如果要使用 SPIKEfile 或是 notSPIKEfile 对一个不能直接启动的应用进行模糊测试，就需要额外的解决方法。Adobe Acrobat Reader 和 RealNetworks RealPlayer 就是典型的例子。

通常，对 Adobe Acrobat Reader 这类应用来说，实际运行的程序是一个 shell 脚本

包装器。这些 shell 脚本设置使真正的二进制可执行文件能正确运行的环境。这么做的主要原因是让应用能够带有它们自己的共享库副本。带有自己的共享库副本使得产品具有更好的可移植性。虽然采用这种方式的出发点是让用户更易于使用，但对我们来说，它把事情搞复杂了。例如，如果我们指定 shell 脚本作为我们的模糊测试对象，我们的子进程就不能附着在实际运行的二进制程序上，而只能附着在 shell 实例上。这样一来，我们截获的信号就毫无价值。下面，我们介绍一个如何解决 Acrobat Reader 和 RealPlayer 问题的例子。其他应用的类似问题的解决思路与此相同。

192

12.5.1　Adobe Acrobat

对 Acrobat 应用，需要首先使用-DEBUG 选项来运行 acroread。使用-DEBUG 选项使得执行者能够进入一个 shell，该 shell 设置了正确的环境来调用真正的 acroread 执行文件，该执行文件通常的位置是 $PREFIX/Adobe/Acrobar7.0/Reader/intellinux/bin/acroread。虽然 Acrobat 的文档中没有包含这些信息，在 usage 中也没有对此进行说明，但阅读 acroread 的 shell 脚本可以让我们找到这些信息。有了这些信息之后，就能够毫无难度地对该执行文件进行模糊测试了。我们已经应用了该方法，在 notSPIKEfile 工具的帮助下发现了 Adobe Acrobat Reader UnixAppOpenFilePerform 缓冲区溢出安全漏洞[4]。

12.5.2　RealNetworks RealPlayer

从下面的输出可以看出，realplay 命令实际上是一个 shell 脚本。我们可以发现，真正的执行程序名为 realplay.bin。

```
user@host RealPlayer $ file realplayrealplay.bin
realplay:        Bourne shell script text executable
realplay.bin:ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.2.5, dynamically linked (uses shared libs), stripped
```

对 RealPlayer 来说，只需要将 shell 环境变量 HELIX_PATH 设置为指向 RealPlayer 的安装路径，就可以直接对包含在 RealPlayer 中的执行文件 realplay.bin 进行模糊测试了。我们已经使用该方法，在 notSPIKEfile 工具的帮助下发现了 ReadNetworks RealPlayer/HelixPlayerRealPix 格式字符串漏洞[5]。

---

[4] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=279

[5] http://www.idefense.com/intelligence/vulnerabilities/display.php?id=311

12.6 案例研究：RealPlayer RealPix 格式字符串漏洞

193

让我们看看我们是如何如何应用 SPIKEfile 工具发现 RealPlayer 中的漏洞的。当然，该漏洞已于 2005 年 9 月被修复。找到该漏洞的第一个步骤是任意选择一种 RealPlayer 支持的文件格式。在本案例中，我们选取 RealPix 格式。从 Google 搜索得到大量信息之后，我们选择了一些示例的 RealPix 文件，将其编译并用做 notSPIKEfile 模糊测试的基础文件。下面是一个迷你版的例子：

```
<imfl>
    <head title="RealPix(tm) Sample Effects"
    author="Jay Slagle"
    copyright="(c) 1998 RealNetworks, Inc."
    timeformat="dd:hh:mm:ss.xyz"
    duration="46"
    bitrate="12000"
    width="256"
    height="256"
    url="http://www.real.com"
    aspect="true"/>
</imfl>
```

[6]

这是一个非常小的 RealPlayer 文件的基本骨架。如果你在 RealPlayer 中加载这个文件，什么都不会显示出来，因为这个文件只包含一个文件头。我们将通过 notSPIKEfile 运行该文件，测试 RealPlayer 对文件头进行解析的代码以查找缺陷。我们用下面的命令开始这次模糊测试。

```
user@host $ export HELIX_PATH=/opt/RealPlayer/
user@host $ ./notSPIKEfile-t 3 -d 1 -m 3 -r 0- -S -s SIGKILL -o FUZZY-sample1.rp
sample1.rp "/opt/RealPlay/realplay.bin %FILENAME%"
[…]
user@host $
```

-t 选项让每次对 RealPlayer 的调用延迟 3 秒。同时，-d 选项告诉模糊测试器在终止一个空闲进程和启动一个新进程之间等待 1 秒。-m 选项指定启动 3 个并发的 realplayer 实例，而-r 选项则告诉工具从整个文件的第 0 个字节开始进行模糊测试。-S 指定字符串模糊测试模式，而-s 选项则指定通过 SIGKILL 信号终止空闲进程。最后，我们告诉

194

[6] http://service.real.com/help/library/guides/realpix/htmlfiles/tags.html

工具被模糊测试的文件名的格式，指定样例文件名 sample1.rp，告诉工具如何执行 RealPlayer，使它能够解析我们的文件。准备就绪并执行模糊测试后，notSPIKEfile 的输出报告了几个崩溃，告诉我们已经找到了一些可能的漏洞。

在查看当前目录下的文件时，我们发现模糊测试工具创建了一个名为 FUZZY-sample1.rp-0x28ab156b-dump.txt 的文件。打开该文件，我们可以看到 notSPIKEfile 工具创建了一个详细报告，描述了进程崩溃时的状态。从这个报告中，我们能看到导致发生崩溃的文件名。在本案例中，引发崩溃的文件被保存为 12288-FUZZY-sample1.rp，可以使用该文件重现这次崩溃。查看导致崩溃的文件内容时，从文件内容中我们能找到揭露问题的信息。导致崩溃的文件内容如下：

```
<imfl>
    <head title="RealPix(tm)Sample Effect"
    author="Jay Slagle"
    copyright="(c)1998 RealNetworks, Inc."
    timeformat="%n%n%n%n%n%n%n%n%n%n%ndd:hh:mm:ss.xyz"
    duration="46"
    bitrate="12000"
    width="256"
    height="256"
    url="http://www.real.com"
    aspect="true"/>
</imfl>
```

由于文件中出现了%n 字符，我们立即猜测是由于格式字符串漏洞导致了这次崩溃。当使用 GDB 运行 RealPlayer 执行文件的时候，这个猜测得到了证实。

```
user@host~/notSPIKEfile $ gdb -q /opt/RealPlayer/realplay.bin
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) r 12288-FUZZY-sample1.rp
Starting program: /opt/RealPlayer/realplay.bin 12288-FUZZY-sample1.rp

Program received signal SIGSEGV. Segmentation fault.
0xb7e53e67 in vfprintf() from /lib/tls/libc.so.6
(gdb) x/i $pc
0xb7e53e67 <vfprintf+13719>:        mov  %ecx, (%eax)
```

我们的确已经发现了一个 RealPlayer 中的格式字符串漏洞，该漏洞位于对 timeformat 选项进行处理的代码中。关于如何利用该漏洞进行攻击，则留给读者作为作业来思考和练习。

195

12.7 开发语言

这两个工具都是用 C 语言开发的。选择 C 语言的原因是：首先，像盐和胡椒粉一样，C 语言和 Linux 已经并且通常会配合得很好。每个现代 Linux 发行版本都有 C 编译器，而且由于 Linux 内核是由 C 语言编写的，因此可能所有 Linux 发行版本中会一直都带有 C 编译器。另外，C 语言能够使我们不需要任何特殊库就能在应用中使用 ptrace 接口，并且能够完全自由地执行需要完成的任务。

另一个选择用 C 语言开发这两个工具的原因是 SPIKE 是用 C 语言编写的。由于这两个工具中至少一个工具扩展使用了 SPIKE 的代码，又因为这两个工具需要共享某些代码，例如异常处理和报告代码，使用两种不同的编程语言实现公用功能是愚蠢的事情。

12.8 小结

在可靠的文件格式模糊测试工具的支持和配合下，发现客户端漏洞就像“选择一个目标然后耐心等待”这么简单。因此，无论是选择编写自己的模糊测试器，还是选择扩展其他的模糊测试工具，花时间去研究工具都是值得的。

第13章

Windows平台上的文件格式自动化模糊测试

"It's in our country's interests to find those who would do harm to us and get them out of harm's way."

——George W. Bush, Washington, DC, April 28, 2005

在第12章中，我们深入探讨了UNIX平台上的文件格式模糊测试。接下来我们转向研究发现Windows应用中的文件格式漏洞。虽然在Windows平台上发现文件格式安全漏洞的总体概念与UNIX平台上一致，但仍有几个需要强调的重要不同。首先，Windows程序基本上都是图形用户界面程序，因此我们在Windows平台上创建的执行脚本、发现文件格式漏洞的模糊测试器也应该具有漂亮的图形用户界面。其次，由于Windows使用给定的默认应用打开特定文件类型，因此，我们需要花一些时间来决定对何种文件格式进行模糊测试。在本章的最后，针对本章中提到的发现漏洞条件的挑战，我们给出了相应的解决方案。

13.1 Windows文件格式漏洞

虽然文件格式漏洞同样可能影响服务器应用，但其主要影响客户端应用。文件格式安全漏洞的紧急性表明客户端漏洞的重要程度正在增加。在服务端方面，网络管理员把大量

198 资源投在保护公司网络与防止网络层漏洞上，与此同时，软件供应商也在尽力降低服务端漏洞带来的威胁，这两方面的共同努力减少了流行软件和操作系统中的服务端关键漏洞，正是这些服务端关键漏洞在过去导致了蠕虫的快速传播，带来了巨大的破坏。然而，在客户端方面，情况却不是这样。过去几年，我们看到了客户端安全漏洞增长的趋势，这些安全漏洞导致了有针对性的攻击，包括钓鱼（phishing）和身份盗窃（identity theft）。

文件格式漏洞给企业带来了特殊的风险。虽然这些漏洞不会导致快速传播的蠕虫或是直接危害网络安全，但这些漏洞在很多方面更难防范。互联网的初衷是鼓励信息分享。Web 上到处都是视频、图片、音乐和文档，正是这些内容使得 Web 成为了巨大的信息和娱乐来源。这些内容的传播都依赖于开放和自由分享的各种文件。由于图片文件和电子表格文件不是可执行文件，因此，以前我们从没有考虑过这些文件可能会是恶意文件。然而，正如前面章节讨论的那样，如果解析这些文件的是存在文件格式漏洞的应用，就可能导致这类文件被攻击者利用。如何才能避免这类威胁呢？让网络管理员在防火墙上阻止所有这些文件？如果这样做，Web 上就会只剩下文本内容，而只有文本内容的 Web 一定不会招人喜欢。我们真的需要退回到使用 Lynx 这样的文本浏览器上网的时代吗？当然不需要，不过我们确实需要意识到文件格式漏洞带来的威胁。

Windows 平台提供了良好的用户友好性，因此也特别易于受到文件格式漏洞的影响。在 Windows 平台上，某个文件格式与默认处理该文件格式的应用相关联。这种关联使得用户可以双击打开特定文件，在 Windows 平台上欣赏电影或是阅读文档。在这种方式下，用户甚至不需要知道什么应用能够打开什么类型的文件。试想一下，如果 Windows 平台上的一个默认用于打开某种类型文件的应用中出现了文件格式漏洞，会有什么样的风险？显然，数以百万计的最终用户会受到该漏洞的影响。下面的引文中列出了一些影响 Microsoft Windows 的较为显著的文件格式漏洞。

影响 Microsoft Windows 的重要文件格式漏洞

MS04-028——JPEG 处理中的缓冲区超出（GDI+）能够允许代码执行

2004 年 9 月，Microsoft 发布了一个安全公告，详细描述了当 GDI+图形设备接口对包含非法注释大小的 JPEG 图片进行渲染时，会导致一个缓冲区溢出漏洞。JPEG 文
199 件中的注释以 0xFFFE 字节值开始，接下来是一个两个字节的注释大小，然后是注释内容。因为注释大小包含了尺寸值本身使用的两个字节，因此注释的最小有效值是 2。如果将该值设置为 0 或是 1，则 GDI+将会由于整数下溢而导致一个可被利用的堆溢出。该漏洞影响了大量的 Windows 应用，因为许多应用都使用到了 GDI+库（gdiplus.dll）。该安全公告发布后，很快就出现了公开的攻击代码。

MS05-009——PNG 处理中的漏洞允许远程代码执行

许多软件供应商，包括 Microsoft 在内，都曾开发过包含缓冲区溢出漏洞的应用。这些漏洞发生在处理包含大的 tRNS 块的可移植网络图像（Portable Network Graphics，PNG）文件时（tRNS 块用于图像的透明化处理）。该漏洞影响了 Windows Messenger 和 MSN Messenger，导致 Microsoft 不得不阻止有漏洞的客户端访问他们的即时信息网络，直到这些客户端打上合适的补丁为止。

MS06-001——图形渲染引擎中的漏洞允许执行远程代码

在 2005 年的旅游季，有报告称，使用 Internet Explorer 浏览器访问带有恶意 Windows 元文件（Windows Meta File，WMF）的网站时，可能导致远程代码执行。由于该问题非常严重，Microsoft 被迫在 2006 年初针对该问题发布了一个临时补丁。后来发现该漏洞由一个设计错误导致。WMF 文件包含允许参数传入特定 Windows 图形设备接口（GDI）库调用的记录。其中一个调用，Escape，及其子命令 SETABORTPROC，实际上允许调用任意可执行代码。有报告指出该漏洞的细节在黑市上卖到了 4000 美金[1]。

在 eBay 上拍卖的 Excel 漏洞

2005 年 12 月 8 日，一个名为 fearwall 的用户在 eBay 上发布了一个拍卖，拍卖的内容是 Microsoft Excel 中的一个漏洞的细节[2]。该拍卖引起了媒体极大的注意，eBay 立即从网站上取消了该拍卖，并援引了禁止非法活动的政策[3]作为取消该拍卖的原因。

FileFuzz 是一个自动识别文件格式漏洞的工具。该工具有三个目标。首先，该工具必须是用户友好的，用户仅靠直觉就能使用。其次，该工具应该能够自动创建用户模糊测试所需的数据文件，并能自动启动和运行处理这些文件的应用。第三，该工具应该集成调试功能，确保能够识别已被处理的和未被处理的异常。为了达成这些目标，我们选择使用 Microsoft .NET 平台设计该应用。Microsoft .NET 平台让我们能够使用 C# 创建图形化前端，同时还可以使用 C 语言创建后端模块如调试器等。FileFuzz 工具的图形用户界面展示在图 13.1 中。

200

[1] http://www.securityfocus.com/brief/126

[2] http://www.osvdb.org/blog/?p=71

[3] http://www.theregister.co.uk/2005/12/10/ebay_pulls_excel_vulnerability_auction/

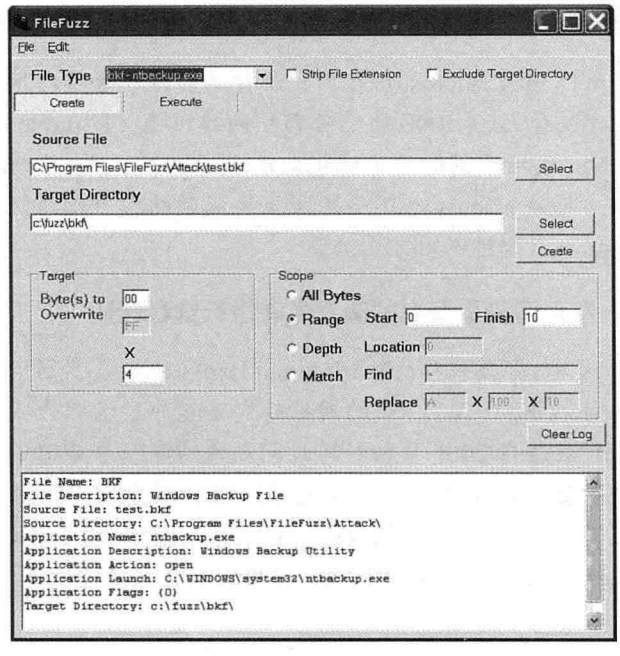


图 13.1 FileFuzz 工具的图形用户界面

13.2 FileFuzz 工具的功能

FileFuzz 工具被设计用来识别文件格式漏洞，该工具采用的是一种简单但有效的强制方法。简单地说，FileFuzz 将正确的特定格式文件的内容打乱，然后让处理这种类型文件的应用来处理它们，观察在处理过程中是否有问题发生。这种强制方法也许并不完美，但的确有效。在开发 FileFuzz 工具的过程中，我们惊讶地发现该工具能轻松地找到潜在的文件格式中可被利用的漏洞。

FileFuzz 工具在运行时执行三个独立的步骤。第一步，该工具创建用于进行模糊测试的文件。FileFuzz 工具使用用户提供的合法文件，基于用户提供的规则，对文件进行连续变异，并把结果保存为文件。第二步，目标应用会依次打开变异后的文件。工具会不断地打开文件，并在用户定义的超时时间后终止打开文件的进程。最后一步，内建的调试功能监视所有这些进程，发现可能出现的已被处理的和未被处理的异常。识别出所有的事件后，FileFuzz 工具记录结果并将结果报告给用户。下面详细描述每个步骤。

13.2.1　创建文件

FileFuzz 工具使用强制方法进行模糊测试。这意味着该工具需要能够读取有效文件，覆写文件的特定段，保存修改后的文件以供解释该文件的应用读取。由于该过程会被重复数百次乃至数千次，因此这个过程必须采用自动化方式实现。

FileFuzz 工具允许采用四种不同的方法（所有字节、范围、深度和匹配）进行文件变异，根据文件类型的不同，变异方法可以被分为以下这些类别：

- 二进制文件（Binary Files）
  - 广度（Breadth）
  - 所有字节（All bytes）
  - 范围（Range）
  - 深度（Depth）
- ASCII 文本文件（ASCII text files）
  - 匹配（Match）

可以看到，FileFuzz 既能处理二进制文件格式，也能处理 ASCII 文本文件格式。在对二进制文件进行变异时，我们使用两种不同的方法，分别称为广度方法和深度方法。让我们用“钻孔找石油”的过程来模拟说明广度和深度方法的区别。如果要在一片广阔的区域上寻找石油，肯定不能随便找个地方就开始钻孔。你需要首先使用各种技术找到最可能藏有石油的地点。你可能会研究地图，分析岩石地层，或是使用地表穿透雷达。无论采用哪种方法，只要找到了可能藏有石油的地点，你就可以开始尝试钻孔，看哪个地点最有希望发现石油。

我们在文件格式模糊测试中采用了类似方法。首先，我们使用广度方法来寻找我们感兴趣的位置；然后，我们使用深度方法来检测是否挖到了石油。广度方法对应着覆写文件内的所有字节或是文件内给定范围内的字节。通过不断将文件中给定范围内的字节值改变成预先定义的值，就可以创建用于模糊测试的文件。完成该步骤后，应用会依次打开这些创建出来的文件，通过异常检查发现修改是否会引发异常。

极少情况下，如果我们运气够好，工具能够发现清晰明了的异常，工具使用者能够控制崩溃的发生，并且从寄存器中能清晰地看到变异后的文件内容，此时我们就无须再进行其他操作。但大多数情况下，事情没这么简单。模糊测试过程中可能会发现崩溃，而且崩溃发生在我们感兴趣的位置，但从寄存器的数据来看，我们并不能确认用户是否

能够控制该崩溃的发生。在这种情况下，我们就需要转向深度方法。在文件中发现了我们感兴趣的字节位置（导致崩溃发生的变异的位置）后，我们就可以关注这些位置，使用深度方法在该位置尝试所有可能的字节值。通过查看得到的崩溃结果，我们可以知道用户能在多大程度上控制该崩溃。如果无论我们提供的字节值是什么，发生异常的位置和寄存器中的值始终如一，那就意味着我们改变的字节值对异常没有影响。然而，如果随着字节值的变化，崩溃发生的位置或是寄存器的值持续发生变化，那就清楚地说明我们在变异文件时使用的字节值至少对异常结果有一些影响。

FileFuzz 工具也可以处理 ASCII 文本文件。FileFuzz 工具允许用户首先选择一个指示文件中覆写位置的字符串，随后用三个独立的输入决定文件变异的输入。工具使用者必须提供三个输入，包括写入文件的字符串、字符串的长度、字符串重复的次数。下面我们来看一个例子。通常，ASCII 文本文件例如*.ini 文件，包含以下格式的名字-值对：

名字 = 值

典型情况下，我们需要覆写文件中的值字段。假设我们想要用连续的 10 个“A”字符覆写这个值。在这种情况下，我们首先需要将 Find 的值设置为“=”字符，因为“=”字符指示了值的开始。然后我们需要将 Replace 的值设置为“A X 10 X 10”。这将会创建 10 个变异文件，10 个变异生成的文件中的 value 值都会被覆写。生成的 10 个变异文件中的 value 的位置上将会包含 10 到 100 个“A”字符。

203

13.2.2 执行应用

创建了模糊测试文件后，我们需要在目标应用中运行它们。例如，我们通过变异生成了*.doc 文件，接下来我们就需要在 Microsoft Word 中运行这些文件。FileFuzz 工具使用 Windows API 中的 CreateProcess()函数启动应用，因此只要我们知道启动目标应用所使用的命令行（包括要传递给应用的必要参数），就能够使用 FileFuzz 来启动目标应用。在本章的后面部分，我们将详细地描述如何找到启动目标应用所需的命令行信息。

进行文件格式模糊测试时，需要重复不断地启动同一个应用数百次甚至数千次之多。如果让每次启动的进程一直运行，很快就会出现内存不足。因此，只有当进程在预定义的时间间隔后被终止后，我们才认为进程执行结束。工具允许使用者控制预定义的时间间隔，用户可以在 FileFuzz 工具的 Execute 选项卡中设置一个“Milliseconds”字段，该字段表示进程被强制终止前所能运行的时间。

13.2.3　异常检测

在前面的章节中我们提到过，异常检测是模糊测试的关键部分。如果你运行了一整夜的模糊测试，早上起床后发现模糊测试确实让应用发生了崩溃，但问题是，如果你无法确定究竟是模糊测试器发出的 10000 个输入的哪些输入导致了应用崩溃，发现崩溃又有什么意义呢？模糊测试并没有让你更接近答案。存在多种在文件格式模糊测试中检测异常的技术，我们介绍其中一种比较突出的方法。对初学者来说，你可以不使用任何技术，只需要直接观察错误窗口、应用、关心系统崩溃就好。如果你不嫌枯燥，喜欢盯着执行过程中的各种输出，直接观察的方法就挺适合你。如果你更愿意隔段时间检查一下的话，可以通过查看日志文件发现是否发生了问题。我们提到的日志文件包含应用日志文件和系统日志文件，例如 Windows 事件浏览器维护的那些日志文件。使用日志文件的问题在于，不太容易直接对应输入（模糊测试文件）与输出（日志事件）。绑定输入与输出的唯一方法是通过使用时间戳（timestamp），当然这种方法并不完美。

对大多数类型的模糊测试来说，识别异常的最好方法是使用调试器。调试器的优势在于它既能够识别出已被处理的异常，也能识别出未被处理的异常。Windows 有强大的异常处理机制，经常能够从它读取的模糊测试文件产生的异常中恢复。即便如此，识别这些已被 Windows 处理的异常仍然很重要，因为只要引入小小的变化，文件中存在问题的地方就可能产生一个不可恢复的或是可被利用的条件。

在文件格式模糊测试中使用调试器不如在其他类型的模糊测试中使用调试器那么直接。进行文件格式模糊测试时，我们不能手工将调试器附着到目标应用，然后运行模糊测试器。这是因为模糊测试器会持续启动和终止目标应用，这使得调试器进程也会被终止。因此，我们需要使用调试器 API，直接在模糊测试器中内建调试器。这样一来，模糊测试器就能够在每次启动目标应用后监视异常。

我们为 FileFuzz 工具开发了 crash.exe 程序，crash.exe 实际上是一个独立的、由 GUI 应用调用的，依次启动目标应用的调试器。该调试器是一个完全独立的命令行应用。由于 FileFuzz 工具是一个开源项目，因此你可以自由地在你自己的模糊测试项目中使用 crash.exe。

204

13.2.4 保存的审计（audit）

可以用手工方式让 FileFuzz 执行审计。选择模糊测试的目标，指明如何生成模糊测试文件，如何保存模糊测试文件就能以手工方式执行审计。另一种执行审计的方法是，用事先保存的审计填充所有必需的字段，然后进行模糊测试。FileFuzz 应用包含大量预先设置的审计，可以从主界面的“File Types”下拉菜单中访问到这些审计模板。此外，由于 FileFuzz 工具的菜单是运行时解析 targets.xml 动态创建的，因此用户还能够创建和保存自己的审计，在无须重新编译应用的情况下将其加入菜单项。下面的例子展示了一个审计的结构：

```
<test>
    <name>jpg - iexploer.exe </name>
    <file>
    <filename>JPG</filename>
        <fileDescription>JPEG Image</fileDescription>
    </file>
    <source>
        <sourceFile>gradient.jpg</sourceFile>
        <sourceDir>C:\WINDOWS\Help\Tours\htmlTour\</sourceDir>
    </source>
    <app>
        <appName>iexplore.exe</appName>
        <appDescription>Internet Explorer</appDescription>
            <appAction>open</appAction>
        <appLaunch>"C:\Program Files\Internet Explorer\iexplore.exe"</appLaunch>
        <appFlags>{0}</appFlags>
    </app>
    <target>
        <targetDir>C:\fuzz\jpg\</targetDir>
    </target>
</test>
```


包含动态生成的下拉菜单是我们的刻意设计。虽然 FileFuzz 是个开源应用，但大多数使用该工具的用户可能并没有编程经验，或是对编写代码扩展工具功能不感兴趣。动态菜单使得大多数用户能够以用户友好的方式扩展该工具的功能。我们设计的 XML 文件结构表明了一个重点：让你的应用尽可能直观和用户友好。虽然文档是开发过程的关键部分，但不要期望用户在遇到问题时会首先去查看文档。用户期望应用足够直观并希望这些应用能够“开箱即用”。

请看上文的 XML 文件。描述性的 XML 标记是自解释的。最终用户能够简单地增加和定制一个额外的<test>块来增加一个审计。创建直观的应用当然不是忽略文档的借口，它是一个重要的设计概念。现代编程语言允许开发者开发用户友好的应用，编程语言本身接管了底层功能，如网络和图形显示部分。开发者可以节省底层实现方面的时间，将其投入到增强应用的友好性方面，让应用具有更好的用户体验。虽然我们设计的是安全方面的专业应用，但也没必要把它设计得只有博士才能操作。

13.3　必需的背景信息

在 Windows 平台和 UNIX 平台上进行文件格式模糊测试需要的基础知识基本相同。但由于 Windows 环境中的默认应用这个重要特性能够增加这类漏洞的风险，因此，在建立我们的模糊测试器之前，让我们先花些时间来探索 Windows 环境中的缺省应用，以便更好地识别高风险目标。

13.3.1　识别目标应用

Microsoft Windows 允许用户为特定文件类型指定缺省应用。这种方式提高了操作系统的可用性，因为一旦设定了某种类型文件的缺省应用，双击文件时，Windows 就能自动打开这个文件。当确定文件格式模糊测试的目标时，记住这一点很重要。如果在某个应用中发现了缓冲区溢出漏洞，而该应用不大可能成为打开某种特定的文件类型的缺省应用，风险会比较小。但如果默认打开某种常用文件的应用中存在相同的漏洞，风险就大得多。以一个存在于 JPEG 图形文件中的溢出漏洞为例。和其他图形格式一样，存在许多可用来打开和查看图片的应用，但在给定的操作系统上，只有一个应用会与某种文件类型缺省相关联。在 Windows XP 上，缺省的 JPEG 查看器是 Windows 图片和传真查看器（Windows Picture and Fax Viewer）。因此，Windows 图片和传真查看器中发现的缓冲区溢出要比其他的，从 Download.com 上下载的，用于查看图片的自由软件中发现的缓冲区溢出风险大得多。为什么呢？因为如果 Windows 平台上的一个缺省应用存在漏洞，数百万的计算机就立刻面临漏洞风险。

206

1．Windows 资源管理器（Windows Explorer）

我们怎样才能知道，Windows 环境下哪个应用会被用来打开某个特定的文件类型？一个简单的方法是双击文件，检查 Windows 启动了哪个应用程序。这种方法可用于快速检查，但却没办法帮我们找出启动该应用的准确命令。找出启动应用的准确命令对于

模糊测试非常重要，因为我们需要一种自动和持续地执行目标应用的方法。当需要对数千个文件进行模糊测试时，快速双击打开文件显然不是解决问题的合适方法。

Windows 资源管理器是一个快速且简便的识别与文件类型相关联的应用的方法，同时该方法还能找到启动应用时的命令行参数。让我们用资源管理器证明 Windows 图片和传真查看器是 JPEG 文件的关联应用。更重要的是，我们需要确定如何才能在 FileFuzz 工具中重复启动模糊测试文件以发现漏洞。

第一步，选择资源管理器中的“文件”→“文件夹选项”菜单项。从弹出的对话框中选择“文件类型”页面。该页面显示的内容如图 13.2 所示。

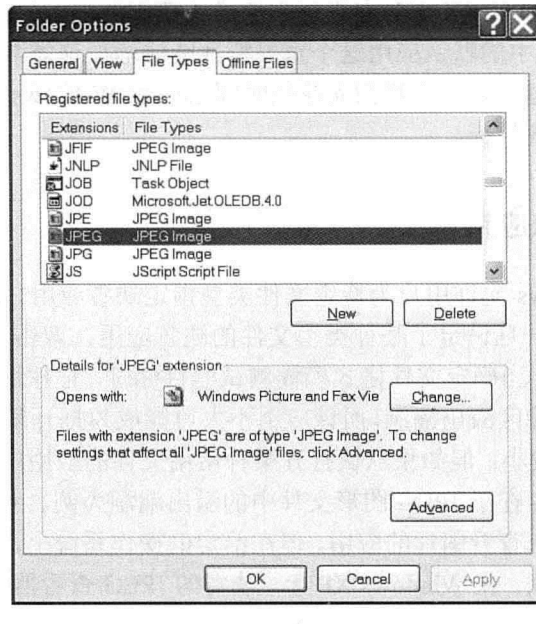


图 13.2　Windows 资源管理器的“文件夹选项”对话框

该对话框页面提供了许多信息。滚动浏览“已注册的文件类型”列表，能够看到几乎所有与特定应用关联的文件扩展名。这些文件类型是很好的模糊测试目标，因为攻击者只要简单地发送文件给受害者，诱骗他们双击文件，就能轻松地利用缺省应用中存在的漏洞。虽然这看上去不太容易，但垃圾邮件证明了如果只需要用户动动鼠标，用户是乐于点击邮件中的附件的，因此这是一个合理的可被利用的场景。

此时，我们已经找到了与 JPEG 文件类型关联的应用，但我们还不知道操作系统是

如何启动这个应用的。幸运的是，只需要再点击几次鼠标，就能知道这些信息。

在处理特定类型的文件时，Windows 有动作（action）的概念。动作允许以不同方式打开文件，或是使用不同的应用打开文件，每个动作都带有独特的命令行参数。我们的目的是找出 Windows 是如何打开 JPEG 文件的，因此我们主要关注“打开”这个动作。如图 13.3 所示，选中“打开”然后单击“编辑”按钮就能让我们找到问题的答案。

208



图 13.3　Windows 资源管理器“编辑文件类型”对话框

最终，Windows 终于揭晓了我们一直寻找的秘密。在“用于执行动作的应用”文本框中，我们看到 Windows 图片和传真查看器根本不是一个可执行程序（图 13.4）。实际上，Windows 图片和传真查看器是一个由 rundll32.exe 执行的动态链接库（Dynamic-link library，DLL）。在 Windows 图片查看器和传真查看器中打开图片文件的完整命令行如下：

```
Rundll32.exe C:\WINDOWS\system32\shimgvw.dll, ImageView_Fullscreen %1
```

不仅 Windows 图片和传真查看器不是我们期望的可执行程序，而且，Windows 还需要我们为该命令行提供一个 ImageView_Fullscreen 参数。如果在命令提示符下输入该命令并将命令行中的%1 替换为合法 JPEG 文件的文件名，我们就能看到 Windows 图片和传真查看器中正常显示给定的 JPEG 文件，如图 13.4 所示。通过命令行打开文件是一个关键概念。只要能够确定给定应用处理文件的命令行参数，我们就可以使用 FileFuzz 测试漏洞。现在，我们已经得到了需要的命令行，将其拷贝到 FileFuzz 的 Execute 页面的“Application and Arguments”字段中即可。唯一需要进行的修改是，我们必须把“%1”（该参数表示目标文件名）修改为“{0}”，因为 FileFuzz 用“{0}”的格式表示目标文件名。

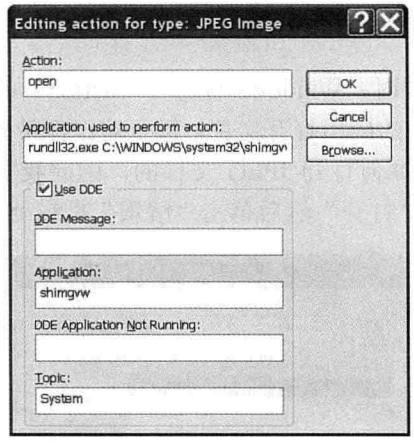


图 13.4 Windows 资源管理器“编辑文件类型的动作”对话框

2. Windows 注册表

尽管 90%的情况下都能够通过 Windows 资源管理器找出特定文件类型和应用之间的联系，但有时候，我们仍会遇到一些无法通过资源管理器找到文件类型对应应用的情况。以*.cbo 文件为例。CBO 文件是 Microsoft 交互教学（Interactive Training）文件格式，某些版本的 Windows XP 带有这个软件。例如，许多 Dell 品牌机上安装的 Windows XP 中都包含该软件。按照上一节提供的方法，你会发现 CBO 文件类型并没有包含在 Windows 资源管理器的文件类型列表中，然而，在 Windows 资源管理器中，CBO 文件以铅笔图标显示，双击这些文件时，系统会启动 Microsoft 交互教学应用。怎么会这样呢？我们如何发现那些没有出现在 Windows 资源管理器的文件类型列表中的启动命令行？为此，我们需要求助于 Windows 注册表。打开 Windows 注册表，检查注册表中\HKEY_CLASSES_ROOT\.xxx 注册表键的键值，“xxx”是文件扩展名。该注册表键的值是用于打开该类型文件的应用名。随后，找出 HKEY_CLASSES_ ROOT\ 下名字与应用名相同的注册表键，在注册表键下级的...\shell\open\command 键值中，能够找到关联这个神秘的文件扩展名的应用的启动参数。

13.4 开发 FileFuzz 工具

现在我们已经初步认识了 Windows 平台上的文件格式模糊测试的独特性，接下来我们可以开始创建一个模糊测试器了。我们将详细分析 FileFuzz 工具设计中的思考过程，然后使用 FileFuzz 工具找出被大肆宣传的 JPEG GDI 漏洞——Microsoft 安全公告

MS04-028 披露了该漏洞的细节。

13.4.1　开发方法

创建 FileFuzz 工具时，我们倾向于开发一个用户友好的图形化应用，该应用应该允许最终用户无须学习一系列的命令行参数就能开展模糊测试。我们想要产生一个仅用单击就能操作的，尽可能符合用户直觉的应用。另外，开发一个专门针对 Windows 平台的文件格式模糊测试工具也是我们的目标之一，因此，对 FileFuzz 工具来说，跨平台功能不是必需的。

13.4.2　选择开发语言

210

考虑到我们的设计目标，我们再次选择了.NET 平台作为开发平台。选用.NET 平台使得我们能够以最小的代价创建一个 GUI 前端，这样我们就能够把更多的时间用在实现功能上。FileFuzz 工具中的 GUI 和文件创建功能都是用 C#实现的，调试功能则是用 C 语言实现的，这是因为 C 语言让我们能够更轻松、更直接地和 Windows API 进行交互。.NET 平台能够支持我们这个设计决策，因为.NET 平台允许其项目包含多种编程语言，只要这些编程语言与.NET 框架兼容即可。

13.4.3　设计

在本章的前面部分，我们详细讨论了如何设计 FileFuzz。接下来我们来看看 FileFuzz 工具中特定部分的实现。在本章中描述 FileFuzz 的所有代码片段显然不可能，但我们可以重点查看重要的代码片段。如果读者想要深入理解 FileFuzz 的设计，建议从 www.fuzzing.org 下载 FileFuzz 的源代码。

1．创建文件

FileFuzz 需要能够兼容所有 Windows 文件格式，显然，对二进制文件和 ASCII 文本文件的处理方法应该不同。Read.cs 文件包含了读取合法文件的所有代码，Write.cs 文件则负责创建模糊文件。

2．读取源文件

FileFuzz 使用强制方法进行模糊测试，因此，该工具从读取已知合法的文件开始。FileFuzz 读取并存储合法文件中的数据，供创建变异的模糊测试文件使用。幸运的是，.NET 框架使得从文件中读取数据相对轻松。对二进制文件和 ASCII 文本文件，我

们采用不同的读取和创建方法。根据设计，我们使用 BinaryReader 类读取二进制文件，并将文件内容存储到字节数组中。读取 ASCII 文本文件的方法和读取二进制文件的方法类似，但对 ASCII 文本文件，我们使用 StreamReader 类读取。另外，对 ASCII 文本文件，我们将读取到的结果存储在字符串中而不是字节数组中。Reader 类的构造函数如下：

```
private BinaryReader brSourceFile;
private StreamReader arSourceFile;
public byte [] sourceArray;
public String sourceString;
private int sourceCount;
private String sourceFile;

public Read(String filename)
{
    sourceFile = filename;
    sourceArray = null;
    sourceString = null;
    sourceCount = 0;
}
```

sourceArray 用来存储读取的二进制文件的字节数组，sourceString 则用来存储读入的 ASCII 文本文件的内容。

3. 写入模糊测试文件

从合法文件中读取内容之后，我们需要对其进行变异，将变异后的结果保存为文件，并在目标应用中运行它们。如前所述，FileFuzz 采用以下四种类型的方法创建变异文件：

- 所有字节（All bytes）
- 范围（Range）
- 深度（Depth）
- 匹配（Match）

我们在 Write 类中实现了所有这四种方法，通过重载构造方法应对不同的场景。对二进制文件，我们使用 BinaryWrite 类向模糊测试文件写入新的字节数组，然后在测试执行阶段用这些模糊测试文件对目标应用进行模糊测试。对于 ASCII 文本文件，我们使用 StreamWriter 类将字符串变量写入到磁盘文件中。

4. 应用执行

负责启动进程、运行目标应用的代码在 Main.cs 文件中，接下来展示的代码段中包含了这部分代码。阅读这些代码时，读者会发现这些代码非常简单，因为这些代码并不负责直接启动目标应用，而是负责启动内建的调试器，由调试器执行目标应用。我们将在后面章节中详细讨论 Crash.exe 调试器。

我们从创建新的 Process 类实例开始。在 executeApp() 函数中，我们初始化一个循环，启动每个先前创建的模糊测试文件。在每次循环中，我们为要创建的进程设置属性，包括要执行的进程的名称，如前所述，无论模糊测试的对象是什么，进程的名称一直都是 crash.exe。命令行应用 crash.exe 会依次启动被测目标应用。这时，控制权被转交给 crash.exe，crash.exe 通过标准输出和标准错误返回最终结果，并将结果显示在 rtbLog 富文本框中，该富文本框是 FileFuzz 的主要输出窗体。

```
    Process proc = new Process();
    public Execute(int startFileInput, int finishFileInput, string targetDirectoryInput,
string fileExtensionInput, int applicationTimerInput, string executeAppNameInput, string
executeAppArgsInput)
    {
        startFile = startFileInput;
        finishFile = finishFileInput;
        targetDirectory = targetDirectoryInput;
        fileExtension = fileExtensionInput;
        applicationTimer = applicationTimerInput;
        executeAppName = executeAppNameInput;
        executeAppArgs = executeAppArgsInput;
        procCount = startFile;
    }

    public void executeApp()
    {
        boolexceptionFound = false;

        // 初始化进度条
        if (this.pbrStart != null)
        {
            this.pbrStart(startFile, finishFile);
        }

        while (procCount<= finishFile)
        {
```

```
                proc.StartInfo.CreateNoWindow = true;
                proc.StartInfo.UseShellExecute = false;
                proc.StartInfo.RedirectStandardOutput = true;
                proc.StartInfo.RedirectStandardError = true;
                proc.StartInfo.FileName = "crash.exe";
                proc.StartInfo.Arguments = executeAppName + " " + applicationTimer + " "
+ String.Format(executeAppArgs, @targetDirectory + procCount.ToString() + fileExtension);
                proc.Start();
                // 更新进度条显示
                if (this.pbrUpdate != null)
                {
                    this.pbrUpdate(procCount);
                }
                // 更新计数器
                if (this.tbxUpdate != null)
                {
                    this.tbxUpdate(procCount);
                }

                // 向富文本日志框写入输出内容
                if (this.rtbLog != null && (proc.ExitCode == -1 || proc.ExitCode == 1))
                {
                    this.rtbLog(proc.StandardOutput.ReadToEnd());
                    this.rtbLog(proc.StandardError.ReadToEnd());
                    exceptionFound = true;
                }
                procCount++;
            }
            // 清除进度条
            if (this.pbrStart != null)
            {
                this.pbrStart(0, 0);
            }
            // 清除计数器
            if (this.tbxUpdate != null)
            {
                this.tbxUpdate(0);
            }
            if (exceptionFound == false)
                this.rtbLog("No exceptions found\n\n");
            exceptionFound = false;
        }
```

5. 异常检测

如前所述，FileFuzz 以包含 crash.exe 的形式包含了调试功能，crash.exe 是一个以 C 语言写成的独立调试器，该调试器利用了 Windows API 中内建的调试功能，并使用了 libdasm 这个帮助解释反汇编码的开源库。从下面的代码可以看到，代码首先执行检查，确保调用 crash.exe 时至少传入了三个参数。在 FileFuzz 中，传入的这三个参数分别是模糊测试的目标应用所在的路径和文件名，强制终止目标应用前的等待时间，最后一个参数是应用运行时的命令行参数，加上要解析的模糊测试文件名。紧接着，代码将传入的等待时间值从字符串类型转换为整数类型，将完整的命令行参数存储在新创建的字符数组中。随后，代码通过带 DEBUG_PROCESS 标志的 CreateProcess 命令启动目标应用。


```
if(argc< 4)
{
    fprintf(stderr, "[!] Usage: crash  <path to app> <milliseconds> <arg1> [arg2
arg3 argn]\n\n");
    return -1;
}

// 将等待时间从字符串转换为整数
if ((wait_time = atoi(argv[2])) == 0)
{
    fprintf(stderr, "[!] Milliseconds argument unrecognized: %s\n\n", argv[2]);
    return -1;
}

// 创建命令行字符串，用于调用 CreateProcess()
strcpy(command_line, argv[1]);

for (i = 3; i<argc; i++)
{
    strcat(command_line, " ");
    strcat(command_line, argv[i]);
}

//
// 启动目标进程
//

ret = CreateProcess(NULL,           // 目标文件名
    command_line,                   // 命令行选项
    NULL,                           // 进程属性
```

```
        NULL,                           // 线程属性
        FALSE,                          // 不继承句柄
        DEBUG_PROCESS,          // 调试目标进程及其所有子进程
        NULL,                           // 使用当前环境
        NULL,                           // 使用当前工作目录
        &si.                            // 指向 STARTUPINFO 结构的指针
        &pi);                           // 指向 PROCESS_INFORMATION 结构的指针

printf("[*] %s\n", GetCommandLine());   //打印命令行

if (!ret)
{
    fprintf(stderr, "[!] CreateProcess() failed: %d\n\n", GetLastError());
    return -1;
}
```


此时，crash.exe 就可以监视和记录异常了。在下一段代码中，我们看到只要没有发生超时，我们就会一直监视调试事件。发生调试事件时，我们可以获得指向有问题的线程的句柄，检测所发生异常的类型。在代码中，我们使用 case 语句识别三类我们感兴趣的异常：内存访问违例、除零错，以及栈溢出。随后我们打印出能够帮助最终用户决定该异常是否值得进一步分析的调试信息。借助 libdasm 库，我们能够解码异常发生的位置，异常发生时的非法操作码，以及发生崩溃时寄存器的值。

```
While (GetTickCount() -start_time<wait_time)
{
    if (WaitForDebugEvent(&dbg, 100))
    {
        // 我们只对调试事件（EXCEPTION_DEBUG_EVENT）感兴趣
        if (dbg.dwDebugEventCode != EXCEPTION_DEBUG_EVENT)
        {
            ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId, DBG_CONTINUE);
            Continue;
        }

        // 获得指向进程的句柄
        if ((thread = OpenThread(THREAD_ALL_ACCESS, FALSE, dbg.dwThreadId)) == NULL)
        {
            frpintf(stderr, "[!] OpenThread() failed: %d\n\n", GetLastError());
            return -1;
        }

        // 获得进程的上下文信息
```

```
        context.ContextFlags = CONTEXT_FULL;

    if (GetThreadContext(thread, &context) == 0)
    {
            fprintf(stderr, "[!] GetThreadContext() failed: %d\n\n", GetLastError());
            return -1;
}

    // 检查异常码
    switch (dbg.u.Exception.ExceptionRecord.ExceptionCode)
{
    case EXCEPTION_ACCESS_VIOLATION:
        exception = TRUE;
        printf("[*] Access Violation\n");
        break;
    case EXCEPTION_INT_DIVIDE_BY_ZERO:
        exception = TRUE;
        printf("[*] Divide by Zero\n");
        break;
    case EXCEPTION_STACK_OVERFLOW:
        exception = TRUE;
        printf("[*] Stack Overflow\n");
        break;
    default:
        ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId, DBG_CONTINUE);
}

// 如果发生异常，打印更多信息
if (exception)
{
    // 得到指向目标进程的句柄
    if ((process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dbg.dwProcessId) == NULL)
    {
        fprintf(stderr, "[!] OpenProcess() failed: %d\n\n", GetLastError());
        return -1;
    }

    // 得到 EIP 指向的内存内容，进行反汇编
    ReadProcessMemory(process, (void *)context.Eip, &inst_buf, 32, NULL);

    // 反汇编成指令字符串
    get_instruction(&inst, inst_buf, MODE_32);
    get_instruction_string(&inst, FORMAT_INTEL, 0, inst_string, sizeof(inst_string));
```

```
            // 在屏幕上打印异常信息
            printf("[*] Exception caught at %08x %s\n", context.Eip, inst_string);
          printf("[*] EAX:%08x EBX:%08x ECX:%08x EDX:%08x\n", context.Eax, context.Ebx,
context.Ecx, context.Edx);
            printf("[*] ESI:%08x EDI:%08x EBP:%08x\n\n", context.Esi, context.Edi,
context.Esp, context.Ebp);
            return 1;
                }
            }
        }
```

217

crash.exe 所发现异常的细节都会被返回给 GUI，并显示在使用者面前。我们希望这些信息能够给用户提供一个快速可视化的参考，帮助识别重要的崩溃。用户应该进一步探索的异常包括在操作码指令中发生的，能导致 shellcode 获取控制权的异常，以及发生异常时，寄存器中包含用户能够控制的值或是用户能够影响的值的情况。

13.5 案例研究

目前为止，我们已经开发了一个文件格式模糊测试器，现在我们用这个工具针对一个已知的漏洞验证我们的设计。Microsoft 发布的安全公告 MS04-028 “GDI+中对 JPEG 处理的缓冲区覆盖（overrun）导致允许代码执行”[4]对我们针对的文件格式漏洞进行了描述。该公告引发了广泛关注，因为它清楚地表明了客户端漏洞能带来多大的破坏。该漏洞是一个可被利用的缓冲区溢出漏洞，存在于许多默认安装的流行的客户端应用中，并影响了一大批用户。这一漏洞导致的结果是迅速出现了对漏洞的公开利用，通过社会工程方法，用户成为了攻击、钓鱼和身份窃取的目标。

该漏洞存在于 gdiplus.dll 库，包括 MicrosoftOffice、Internet Explorer 和 Windows 资源浏览器在内的大量应用使用了该库。JPEG 文件格式允许在图片中嵌入注释。注释以字节串 0xFFFE 开头，随后是一个 16 位的双字节值，该值指示注释的长度。注释的长度值包括用于表示长度的两个字节本身，以及注释内容的长度。

FileFuzz 工具能够发现这个漏洞吗？我们来试试看。我们先从一个合法的图像文件开始，以手工方式或是使用图像编辑器为其增加一个注释。如果你想要在自己的环境中尝试重现发现漏洞的过程，请确保你使用的 Windows 版本是一个存在该安全漏洞的版

[4] http://www.microsoft.com/technet/security/Bulletin/MS04-028.mspx

本。我们的测试结果是在 Windows XP SP1 上得到的。准备测试文件时，我们使用一个非常简单的图像文件来生成测试文件，在本例子中，我们使用的图像文件仅包含一个 1×1 的白色像素。为什么使用这么简单的图像文件？前文已经指出，强制模糊测试的效率并不高。我们希望测试能够专注在图像文件头，而不是图像本身上。我们选择使用这个几乎不可能真实存在的图像，能够得到一个大小仅为 631 字节的文件，因此，我们能够在一个合理的时间范围内完成整个模糊测试。使用十六进制文件编辑器为该文件增加内容为“fuzz”的注释后，该文件的字节序列如下所示：

```
0000009eh:     FF FE 00 06 66 75 7A 7A              : …TM159fuzz
```

按照注释的格式进行分解：

```
FF FE                  注释前缀
00 06                  注释的字节长度
66 75 7A 7A            注释的内容（fuzz 的 ASCII 值）
```

开始模糊测试之前，我们需要找出究竟哪个 Windows XP 应用程序是负责渲染（rendering）JPEG 文件的缺省应用。幸运的是，从本章前面的描述中，我们已经知道 Windows 图片和传真查看器负责这个任务，且该应用使用以下命令行打开 JPEG 文件（见图 13.5）。

```
Rundll32.exe c:\windows\system32\shimgvw.dll ,ImageView_FullScreen %1
```

接下来，我们就可以运行 FileFuzz 工具，并开始模糊测试过程了。FileFuzz 有一个内建的 JPEG 文件审计，该审计可以从“File Type”下拉菜单中访问到。但是，为了演示 FileFuzz 的功能，我们将从头开始进行测试。

我们从“Create”页面开始，设置合适的选项值，基于前面创建的包含注释的合法 JPEG 文件生成一系列修改后的 JPEG 文件。我们将“Create”页面上的选项设置为以下这些值。

- “Source file”选项：设置为“C:\Program Files\FileFuzz\Attack\test.jpg”，该文件是个合法的 JPEG 文件。
- “Target directory”选项：设置为“C:\fuzz\jpg\”。该目录是存放生成的模糊测试文件的目标。
- “Byte(s) to overwrite”选项：设置为两个“00”。根据漏洞的详情[5]，如果我们

[5] http://www.securityfocus.com/archive/375204

将长度值设置为 0x00 或 0x01，就会产生溢出。注释的长度不可能为 0 或者 1，因为总长度值中已经包含了长度值本身的两个字节。因此我们将使用 0x0000 这个双字节值对该文件进行模糊测试，期望能够通过重写注释长度值触发溢出。

- **“Scope”选项**：设置为 150～170。在我们创建的测试文件中，长度值从第 160 个字节开始。为了确保发现该漏洞，我们将从第 150 个字节到第 170 个字节进行模糊测试。

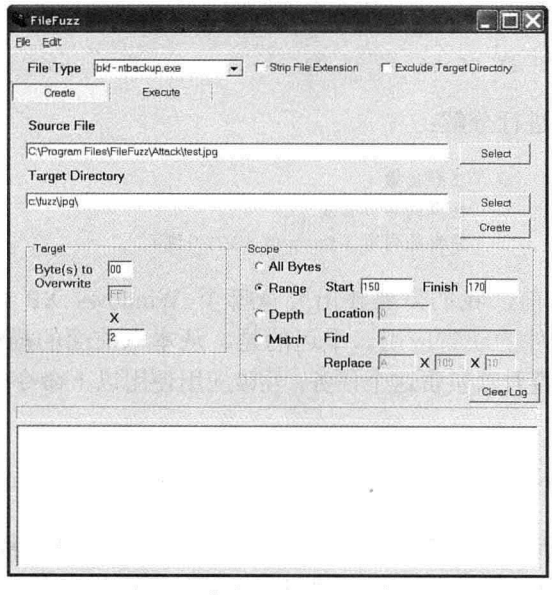


图 13.5　FileFuzz 针对 JPEG 模糊测试的 Create 页面的设置

219

最终完成后的设置见图 13.5。

完成所有的设置后，单击“Create”按钮就能生成模糊测试文件。接下来我们转向“Execute”页面。在“Execute”页面中我们需要告诉 FileFuzz 如何启动 Windows 图片和传真查看器。“Execute”页面上的各设置值如下。

- **“Application”设置**：设置为 rundll32.exe。因为 Windows 图片和传真查看器实际上是个动态链接库，我们实际上使用的应用是 rundll32.exe，该文件用于运行动态链接库文件。
- **“Arguments”设置**：设置为“C:\WINDOWS\system32\shimgvw.dll, ImageView_Fullscreen {0}”。传入的参数包括 Windows 图片和传真查看器（shimgvw.dll）的

完整位置，ImageView_FullScreen 参数，以及一个{0}，{0}是将要被打开的模糊测试文件的占位符。

- “Start File”选项：设置为 150。我们创建的第一个文件。
- “Finish File”选项：设置为 170。我们创建的最后一个文件。
- “Milliseconds”选项：设置为 2000。Windows 图片和传真查看器在被强制关闭前允许运行的时间。

各设置项的最终设置值见图 13.6。

220

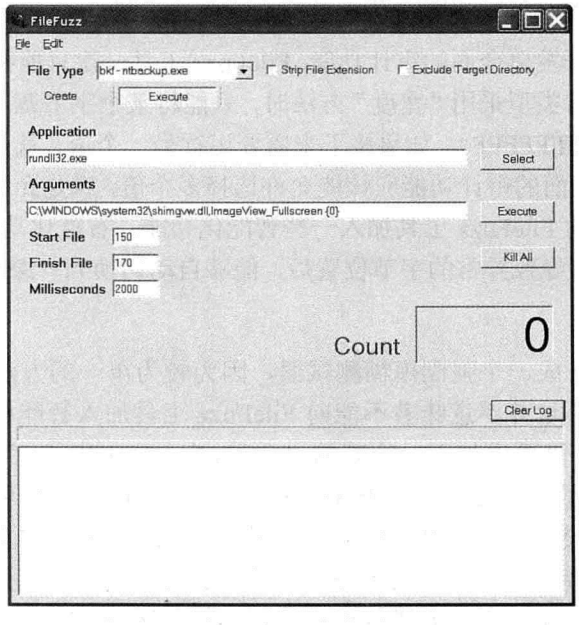


图 13.6　针对 JPEG 模糊测试的 FileFuzz 中的 Execute 页面的设置

接下来我们就可以开始运行测试了。单击“Execute”按钮，我们可以观察到 Windows 图片和传真查看器不断地启动和退出。启动和退出一共会重复 21 次，直到我们创建的 21 个模糊测试文件都被打开和处理完成。执行结束后，我们发现 FileFuzz 工具确实检测到了一些异常。然而，我们感兴趣的异常发生在处理 160.jpg 文件时，该异常的输出如下所示。第 160 个字节是 JPEG 文件中注释长度值的起始字节，在 160.jpg 文件中该值被覆写为 0x0000。

```
[*] "crash.exe" rundll32.exe 2000
C:\WINDOWS\system32\shimgvw.dll ,ImageView_Fullscreen c:\fuzz\jpg\160.jpg
[*] Access Violation
```

```
[*] Exception caught at 70e15599 rep movsd
[*] EAX:fffffffe EBX:00904560 ECX:3ffffe3c EDX:fffffffe
[*] ESI:0090b07e EDI:0090c000 ESP:00aaf428 EBP:00aaf43400
```

221

13.6 优势和提升空间

FileFuzz 提供了使用强制性方法进行模糊测试的基础功能。利用该工具的 GUI 和内建的调试能力，用户可以使用一个已有的有效文件作为起点，对文件格式进行快速的审计。在 FileFuzz 提供的基础功能之外，有很大的提升空间。

可以为初学者开发更全面的审计功能。FileFuzz 工具一次只能执行一个审计。例如，当对一个二进制文件类型采用“宽度”方法时，只能将某个字节范围内的值修改为相同字节值（例如，0xFFFFFFFF）。如果接下来需要运行另一个值，就必须使用新值来重新执行一遍进程。更全面的审计功能应该能允许选择多个值，或是允许选择需要被测试的值的范围。还可以向 FileFuzz 工具加入一些智能化功能，智能化功能首先采用“宽度”方法，然后，识别出导致异常的字节位置后，能够自动切换到“深度”方法，以发现多种类型的异常。

FileFuzz 被设计成一个强制模糊测试器，因为较为单一的方法更适合用于文件格式模糊测试。然而，这并不意味着不能向 FileFuzz 工具加入智能模糊测试的能力。也许可以向“Create”页面中加入一个新的“Create”-“Intelligent”页，这样，现有的“Create”页就变成了“Create”-“Brute Force”页。新增加的页可以包括一个全新的智能模糊测试功能集，该功能集支持用户采用模板方式定义特定文件类型的结构，而不是使用一个已经存在的文件作为起点。这种方法需要用户花费更多的精力，但可以允许用户更好地指明文件中需要进行模糊测试的特定区域，以及对其进行模糊测试的方法。用户可以在研究了目标文件格式的规格文档后创建模板，FileFuzz 工具也可以自带内建的示例模板。

智能异常处理能够帮助过滤许多不大可能导致可被利用条件的异常。向 FileFuzz 工具增加分析 crash.exe 输出的规则，可以过滤特定的异常，例如， 可以根据 crash 发生位置的操作码，发生异常时的寄存器值，或是发生异常时栈的状态实现过滤。除了过滤这些异常外，另一种可选方案是高亮显示那些更可能存在问题的地方，这种方案也许比从异常结果中过滤某些特定结果更好。

简单地说，这里有很大的提升空间。本书作者的目标只是简单地让该工具能够运转。剩下的就靠你了。

13.7　小结

在过去一些年里，文件格式漏洞一直在折磨微软公司。无论是媒体文件，还是 Office 文件中的漏洞，都已经在很多地方出现过，并正在被攻击者利用。微软并不是唯一同这类漏洞斗争的公司，之所以有这样的结果，仅仅是因为微软一直都是缺陷猎手们的最爱。随着最近越来越多的人开始关注文件漏洞，希望软件供应商们能够把模糊测试加入他们的开发生命周期中，在产品发布前发现这类漏洞。

第14章

网络协议的模糊测试

"I own a timber company? That's news to me. Need some wood?"

——George W. Bush, second presidential debate, St. Louis, MO, October 8, 2004

模糊测试这一概念诞生于威斯康辛大学使用随机参数对 setuid UNIX 命令行应用进行测试时。“模糊”这个词在诞生时仅仅指使用随机参数测试 setuid UNIX 命令行应用，但如今，出于种种合理理由，“模糊”这个词也被应用在网络协议测试上。网络协议模糊测试是安全研究者最感兴趣的模糊测试类型，因为通过这种测试方法通常能够发现最高危害级别的安全漏洞。无须有效凭证就能访问的远程漏洞，或是通过用户交互就能利用的远程漏洞等，都是这类高危漏洞的典型例子。

攻击者通常利用客户端漏洞（例如，影响微软 Internet Explorer 的漏洞）创建僵尸网络（bot nets）。利用僵尸网络进行攻击的行为类似渔夫撒网，攻击者通过广撒网的方式，尽可能捕得更多的“鱼”。大多数情况下，被捕到的“鱼”是互联网上的个人计算机。当然，攻击者同样可以利用网络后端应用中存在的服务端漏洞来构造僵尸网络，但对攻击者来说，用后端漏洞来构造僵尸网络实在是浪费。因为从攻击者的角度来说，通过后端漏洞可以掌控后端数据库或是企业级 Web 服务器，这样就能够有极好的机会窃取数据，以及以此作为跳板进行进一步的攻击。在本章中，我们将介绍网络协议模糊测试，并给出这类模糊测试独特的特点和带来的挑战。网络协议模糊测试能够帮助发现一些网络协议安全漏洞。

14.1　什么是网络协议的模糊测试

与其他类型的模糊测试一样，对网络协议进行模糊测试也需要识别出可被攻击的接口，通过变异或生成方式得到能够触发错误的模糊测试值，然后将这些模糊测试值发送给目标应用，监视目标应用的错误。简而言之，如果你的模糊测试器通过某种形式的套接字（socket）与目标应用通信，那么这个模糊测试器就可以被看作是网络协议模糊测试器。

基于网络的模糊测试使用套接字通信组件，因此与其他模糊测试相比，它有个有趣的差别，而这个差别导致了网络协议模糊测试的瓶颈。如果比较文件格式模糊测试、命令行参数模糊测试、环境变量模糊测试和网络协议模糊测试的传输速度，那就像是比较阿斯顿马丁 DB9 和 Ceo Metro 的速度[1]。

现有的网络协议模糊测试器倾向于采用两种风格：一种风格是采用通用的框架，可以对多种协议进行模糊测试。SPIKE 工具[2]和 ProtoFuzz 工具（我们将会在第 16 章中展示该工具的完整构建过程）就属于这一类。SPIKE 工具是知名度最高的模糊测试器，我们将在第 15 章中对其进行详细描述。另一种网络模糊测试器的风格是面向特定协议。这类工具包括 ircfuzz[3]，dhcpfuzz[4]，以及 InfigoFTPStressFuzzer[5]。根据这些工具的名字，读者应该猜得到它们被设计来测试何种协议。面向特定协议的模糊测试器通常只是些小脚本或小应用，而通用框架则需要更大的开发工作量。第 21 章中，我们将会对比面向特定协议的工具和模糊测试框架，讨论创建和使用模糊测试框架的价值。接下来，我们来看一些例子，它们展示了网络协议模糊测试针对的目标。

微软抓住了一个病毒

对于世界上大多数的非妄想狂和非精神分裂者来说，千禧年的到来是一个值得庆贺的时刻。我们享受了历史上最大的新年庆典，从千禧年的恐慌中幸存了下来，并高兴地看到世界并未终结。

---

1 译者注：Geo Metro 是铃木在北美的品牌，属于第三代 CULTUS，排量介于 1L～1.3L 之间；而阿斯顿马丁则是著名的大型 GT 跑车，排量为 6.0L

2 http://www.immunitysec.com/resources-freesoftware.shtml

3 http://www.digitaldwarf.be/products/ircfuzz.c

4 http://www.digitaldwarf.be/products/dhcpfuzz.pl

5 http://www.infigo.hr/en/in_focus/tools

但对微软来说，却没有那么幸运，因为在千禧年接下来的几年中，将会有不少严重
225 的服务端漏洞在微软最流行的产品中被发现。与这些漏洞同时发生的是漏洞利用代码的快速开发与发布。更严重的是，相当多的漏洞被快速扩散的恶意代码所利用，从而导致了世界范围内公司的经济损失，为世界各地的系统管理员敲响了警钟。

让我们看看一些著名的、在世界范围内对微软造成影响的蠕虫和漏洞。这些蠕虫导致的声誉损失是刺激微软在2002年启动可信赖计算计划的直接原因之一[6]，该计划改变了微软在软件开发生命周期中解决安全性问题的方式。

Code Red。IIS Web服务器应用编程接口（IIS Web Server Programming Interface，ISAPI）扩展中的一个缓冲区溢出问题在2001年6月18日被公布[7]。在该漏洞的补丁已经存在了将近一个月后，2001年7月13日发现了针对该漏洞的蠕虫程序。蠕虫程序利用了这个漏洞，感染存在安全漏洞的Web服务器，并在网站上留下“HELLO! Welcome to http://www.worm.com! Hacked by Chinese”的文字。Web服务器被感染后，蠕虫程序会潜伏20～27天，然后尝试向包括whitehouse.gov网站在内的多个特定的IP地址发动DoS攻击。

- Slammer：SQL slammer蠕虫利用了两个微软安全公告板上公布的漏洞MS02-39[8]和MS02-061[9]，这两个安全漏洞存在于微软SQL Server及其桌面引擎中。该蠕虫在2003年1月25日首次被发现，它仅用了10分钟就感染了75 000台计算机[10]。该蠕虫利用的缓冲区溢出漏洞（MS02-039）在蠕虫被发现前6个多月前就已经被发现，而且微软当时已经发布了针对该漏洞的补丁，但仍有大量的服务器存在该安全漏洞。
- Blaster：2003年8月11日，年仅18岁的Jeffery Lee Parson向世界展示了他的发明：一个蠕虫程序[11]。该程序利用了Windows XP和Windows 2000中的DCOM

---

6 http://www.microsoft.com/mscorp/twc/2007review.mspx

7 http://research.eeye.com/html/advisories/published/AD20010618.html

8 http://www.microsoft.com/technet/security/bulletin/MS02-039.mspx

9 http://www.microsoft.com/technet/security/bulletin/MS02-061.mspx

10 http://en.wikipedia.org/wiki/SQL_slammer_worm

11 http://en.wikipedia.org/wiki/Blaster_worm

远程过程调用（Remote Procedure Call，RPC）缓冲区溢出漏洞[12]。再次说明，该蠕虫出现时已经存在针对漏洞的补丁。该蠕虫通过 SYN 洪泛（SYN flood）方式分布式地向 windowsupdate.com 进行 DoS 攻击。Parse 由于其“功绩”而被判入狱 18 个月，三年的监督释放（supervised release），以及 100 小时的社区服务[13]。

以上的列表远算不上全面地展示了一些历史上由微软的服务端漏洞导致的，快速传播的，著名的蠕虫。而网络模糊测试很可能能发现以上这些网络安全漏洞。

14.2　选择目标应用

有数千种目标应用可供测试，这些应用都可能暴露出远程可被利用的网络协议解析安全漏洞。表 14.1 给出了已知网络安全漏洞的一小部分示例，以强调一些常见的目标类别。

表 14.1　存在漏洞的应用程序的常见类别与被发现的漏洞示例

应用类别	漏洞名称	报　告
邮件服务器	Sendmail 远程信号处理漏洞	http://xforce.iss.net/xforce/alerts/id/216
数据库服务器	MySQL 认证可被跳过的漏洞	http://archives.neohapsis.com/archives/vulnwatch/2004-q3/0001.html
基于远程过程调用的服务	远程过程调用 DCOM 缓冲区溢出漏洞	http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx
远程访问服务	OpenSSH 远程 Challenge 漏洞	http://bvlive01.iss.net/issEn/dilivert/xforce/alertdetail.jsp?oid=20584
媒体服务器	RealServer ../ 描述漏洞	http://www.service.real.com/help/faq/security/rootexploit082203.html
备份服务器	CA BrightStorARCserver 备份信息引擎缓冲区溢出漏洞	http://www.zerodayinitiative.com/advisories/ZDI-07-003.html

以上 6 个类别建立了一个良好的目标应用集合。然而，任何接受传入连接的应用或服务都可以是潜在的测试目标。因此，潜在测试目标包括硬件设备、网络打印机、个人数字助理、手机或其他。任何能够接受网络数据的软件都可以用网络模糊测试技术来进行审计。

[12] http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx

[13] http://weblog.infoworld.com/techwatch/archives/001035.html

为了使读者更全面地理解典型测试目标，我们为开放系统互连基础参考模型（Open System Interconnection Basic Reference Model，OSI）[14]中定义的 7 层结构的每一层上都提供了示例。OSI 七层模型如图 14.1 所示。虽然从来没有任何单一网络技术实现了 OSI 模型的全部七层，但在剖析网络技术，说明特定功能应该落在哪一层时，我们通常使用 OSI 模型作为参考。理论上，网络协议模糊测试能够针对 OSI 模型的每一层。在实践中，读者能够找到针对除第一层物理层外其他所有层的模糊测试器。在下文中展示 OSI 模型的每一层时，我们也会对与该层相关的，历史上出现过的漏洞进行进一步的解释。



图 14.1　开放系统互连基础参考模型（OSI）

14.2.1　第二层：数据链接层

数据链路层相关的技术包括以太网（Ethernet）帧和 802.11 帧。由于底层网络层的处理在操作系统内核中实现，因此第二层的漏洞看上去都很有意思。最近，Mitre 发现了一个第二层漏洞的例子，该漏洞描述在 CVE-2006-3507 中[15]。由于 Mac OS 系统中存在多个基于栈的缓冲区溢出，攻击者可以利用这个漏洞对无线上网的 Mac OS 系统进行攻击。该溢出发生在内核中，是一个严重的问题，能够导致彻底破坏被影响系统。这个漏洞有一个有趣的限制：攻击者必须和被攻击的系统在同一个无线网络中。如果在咖啡厅使用公用无线网络的时候，你发现自己的 Mac OS 系统突然挂掉了，那说明攻击者很

[14] http://en.wikipedia.org/wiki/Osi_model

[15] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3507

可能就在你的身边。

“苹果门”事件

在 2006 年拉斯维加斯的黑帽大会上，安全研究人员 Jon “Johnny Cache” Ellch 和 David Maynor 发布了一段视频，演示通过一个已知的无线驱动中的安全漏洞远程攻击一台苹果 MacBook，该视频引发了巨大的争议[16]。随后媒体疯狂地从各方挖掘和报导内幕，包括苹果公司指控这两个研究人员从未公开足够的漏洞细节以允许苹果公司验证他们的发现。苹果公司的拥趸也加入了这场论战，宣称该漏洞利用了一个第三方的驱动，而不是苹果公司自己开发的驱动。

最终，苹果公司发布了一系列的补丁，包括 CVE-2006-3507，但继续坚持该漏洞来自针对黑帽大会上的演示所进行的内部审计。发现这一漏洞的荣誉从未被授予 Maynor 或是 Ellch。在这个极度戏剧化的一系列事件中，唯一确定的事情是，我们可能永远都不会知道这件事情的真相了。

14.2.2　第三层：网络层

229

第三层即网络层，包括 IP 协议和互联网控制信息协议（ICMP）。虽然 TCP/IP 最常用的实现已经在过去的若干年间被反复测试，但在这一层上仍然能够找到漏洞。值得指出的是，Windows Vista 包含了一个完全重写的网络栈，因此 Windows Vista 可能是一个很好的进行网络模糊测试的对象。近期，微软安全报告 MS06-032 中公布了一个 TCP/IP 漏洞“TCP/IP 中的漏洞允许远程代码执行”。该漏洞存在于内核中，是由错误地解析 IPv4 的源路由选项导致的。这一错误导致了一个缓冲区溢出，而该溢出可被远程攻击者用来获取内核层访问权限。

14.2.3　第四层：传输层

OSI 模型的第四层是传输层，该层包括 TCP 和 UDP。如前所述，大多数 TCP/IP 实现都经过了良好的测试，但过去在这一层仍然发现了问题。该层的一个安全漏洞的例子是古老的利用带外（out-of-band）TCP 包的“Winnuke”攻击[17]。Winnuke 攻击也许是目前为止最简单的远程内核 DoS 攻击方式。只要发送一个设置了 TCP 紧急指针（urgent

[16] http://blog.washingtonpost.com/securityfix/2006/08/hijacking_a_macbook_in_60_seco.html

[17] http://support.microsoft.com/default.aspx?scid=kb;[LN];168747

pointer）的 TCP 包就能让一台存在漏洞的远程计算机崩溃。而只需要使用套接字 API，将数据包指定为包含带外数据就可以设置紧急指针。

14.2.4 第五层：会话层

会话层是 OSI 模型的第五层，该层包括两个特别容易存在问题的协议，这两个协议都实现了远程过程调用。这两种技术分别是 DCE/RPC（微软的 MSRPC）和 ONC RPC（也称 Sun RPC），分别实现于 Windows 和 UNIX 系统上。过去若干年中在这两种协议实现中发现了大量的关键漏洞。其中最有影响的一个当属在微软安全公告 MS04-011[18]中提到的，被大肆肆虐的 Sasser 蠕虫[19]所利用的漏洞。该漏洞存在于所有较新版本的 Windows 的 lsass.exe 文件中，该漏洞暴露了默认注册的 RPC 端点。更详细地说，该漏洞由 SdRolerUpgradeDownlevelServer 函数中的缓冲区溢出导致，并能被攻击者远程利用。

14.2.5 第六层：表示层

在第六层采用的技术中，外部数据表示（eXternal Data Representation，XDR）是适合使用模糊测试的技术之一。在过去被发现的与 XDR 相关的漏洞中，Neel Mehta 发现的 xdr_array 整数溢出漏洞是一个特别好的例子[20]。该漏洞的核心是利用整数溢出进行攻击。如果攻击者向一个数组中写入大量数据，而系统为其分配了的缓冲区不够大，写入的数据就会溢出。攻击者可以利用溢出造成的内存破坏来破坏目标系统。

14.2.6 第七层：应用层

第七层是应用层。应用层是最常被想到的和最适合应用网络协议模糊测试的 OSI 层。常用协议如 FTP，SMTP，HTTP，DNS，以及其他诸多标准化协议都位于这一层。历史上在该层发现的漏洞比任何其他层发现的漏洞都要多。由于在这一层上有常用协议的各种不同实现，因此，第七层是最常进行模糊测试的层。

本章的余下部分主要关注第七层。尽管如此，但请读者不要忘记：OSI 模型每一层

---

18 http://www.microsoft.com/technet/security/bulletin/MS04-011.mspx

19 http://en.wikipedia.org/wiki/Sasser_worm

20 http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20823

上的软件都可能存在实现问题，在对测试目标进行全面审计时，不要忽略任何一层。

14.3　测试方法

大多数可用的网络协议模糊测试方法都和第 11 章“文件格式模糊测试”中描述的方法相同。在较高的层次上，强制或智能的方法可以同时用在文件格式模糊测试和网络模糊测试中。然而，当在基于网络的模糊测试中应用这两种方法时，方法的使用上会有一些差异。

14.3.1　强制（基于变异的）模糊测试

在文件格式模糊测试中，强制模糊测试方法需要测试者获得合法的目标文件格式的样例。随后，文件格式模糊测试器用各种方法对这些文件产生变异，接着，将每个测试用例传给一个目标应用。在网络模糊测试中，测试者通常使用嗅探器，在测试之前使用静态方法或是在运行时使用动态方法抓取合法的协议数据。随后，模糊测试器对抓取到的数据进行变异，将其发送给目标应用。当然，这种方式并不总是像听起来那样有效。例如，以实现了基本回放攻击保护的协议为例。在这种情况下，简单的强制网络模糊测试除了能对初始化会话相关的代码（例如，认证过程）产生效果外，不会有其他任何作用。

另一个导致基于变异的模糊测试器失效的例子是尝试对包含校验码的协议进行模糊测试。除非模糊测试器可以动态更新校验码字段，否则，传入的数据会被目标应用在真正开始处理前就丢弃掉。在这些情况下测试用例毫无作用。基于变异的模糊测试在文件模糊测试场合工作良好，但一般来说，下一种模糊测试方法更适合网络协议模糊测试。

14.3.2　智能强制（基于生成的）模糊测试

要使用智能强制模糊测试，你首先得花些精力实际研究协议规范。智能模糊测试器只是个模糊测试引擎而已，依靠的仍然是强制性攻击。当然，与强制模糊测试器不同，智能模糊测试器可以依赖用户提供的配置文件，使得模糊测试的过程更智能。用户提供的配置通常包含描述协议语言的元数据。

当对一种常见网络协议（如 FTP 中的控制信道协议）进行模糊测试时，智能模糊测试通常是最合适的方法。使用一种简单的可以描述协议中的每个动词（USER，PASS，CWD，等等）的语言，加上描述每个动词参数的数据类型，就能够进行相当全面的测

试。可以基于一个已存在的框架（例如 PEACH[21]），也可以从头开始开发基于生成的模糊测试器。如果采用后一种方法的话，我们开发的面向特定协议的模糊测试器基本上不可能用于对其他协议进行模糊测试。当然，采用后一种方法的好处是，在设计时不需要考虑任何通用性，从而简化了设计和实现过程。

目前已有几款公开发布的面向单个协议的网络协议模糊测试工具，但许多面向单个协议的测试工具仍然处于私有状态，原因是这些工具和“上了膛的武器”一样。更详细地说，如果某个工具作者发布了一款通用的模糊测试器，无论该模糊测试器是基于变异还是基于生成的，工具的使用者仍然需要在用户端进行一些研究工作才能找到漏洞。而面向特定协议的模糊测试器仅需要用户提供测试目标就能开始测试，所以，无论谁在目标应用上运行面向特定协议的模糊测试工具，都能找到同样多的缺陷（即使工具作者也不例外），从这一点上来说，发布该工具对工具作者没什么用处。当然，即使将面向特定协议的工具发布出去，如果能找到工具尚未尝试过的、使用相同协议的新目标，这类模糊测试器仍然可以发挥作用。因此，是否愿意发布这样一个工具完全取决于个人对消除缺陷和对研究共享的所持有的价值观和信念。

14.3.3 通过修改客户端进行变异模糊测试

模糊测试中一个常见的重要问题是，随着测试复杂度的提高，开发模糊测试器的复杂度会接近开发一个完整的客户端。因此，为什么我们不回头看看呢？这里有一类我们以前没有提过的模糊测试，这种模糊测试方法通过修改客户端源代码（如果有可用的源代码的话）来创建模糊测试器。基本上，这类模糊测试器被嵌入在已经实现了用我们期望的协议与服务器进行通信的应用中，这样，就用不着在模糊测试器中实现整个协议了。如果模糊测试器能够访问到生成合法协议数据的例程（routine），这种方法就会给模糊测试器的开发者带来好处，最小化所需投入的精力。

虽然我们不知道有哪种公开发布的模糊测试工具采用了这个方法，但通过这种方法，已经有人开发出来了几个利用漏洞的程序。其中的一个例子是由 GOBBLES 写的 OpenSSH 中的 sshutuptheo 漏洞利用程序[22]，该程序利用了主流 SSH 服务器中的预认证整数溢出（preauthenticationinterger overflow）漏洞。

---

[21] http://peachfuzz.sourceforge.net/

[22] http://online.securityfocus.com/data/vulnerabilities/exploits/sshutuptheo.tar.gz

这种方法对处理复杂的协议（如 SSH 协议）尤其有优势。然而，开发者必须小心客户端可能带来的限制，而这些限制可能影响代码覆盖。例如，如果选择了 SSH 作为测试对象，但选择用来开发测试工具的基础客户端仅支持 SSH 版本 1，那开发出来的测试工具就不会覆盖 SSH 版本 2 的任何规范。当然，这种方法的另一个不足之处是，需要进行大量的修改才能让一个客户端覆盖许多测试用例。

14.4　错误检测

错误检测是模糊测试过程的核心部分，这就是为什么我们用整个 24 章“智能错误检测”讨论这个问题。虽然在本章内我们仍不会讨论高级错误检测方法，但可以看看网络模糊测试中使用的基础错误检测技术。根据测试目标的不同，在网络模糊测试中检测错误的困难之处也不相同。例如，让我们考虑一个在发生错误后崩溃并停止接受任何连接的网络后台进程。显然，从错误检测的角度来说该进程具有良好的行为。因为，当服务器停止响应之后，测试者可以简单地假定最后一个测试用例导致服务器停止响应。这种类型的检测完全可以由模糊测试器完成，而无须任何软件代理或是面向测试目标的人工监测。

让我们考虑另一个场景：假设目标应用能够自行处理错误，或是启动了一个单独进程来监听端口。在这种情况下，除非我们对目标主机进行某种形式的深入探索，否则，我们完全可以在不被检测到到得情况下触发一个可被利用的漏洞。假设在大多数模糊测试中，我们能够访问到目标主机（相信这个假设是合理的），下面，让我们来看一些可以在目标主机上检测错误的基本方法。

14.4.1　手工方式（基于调试器）

假设我们能够以本地方式访问某台机器，最简单的监视异常的方法就是在进程上附着一个调试器。发生异常后，调试器能够检测到异常并允许用户决定采取何种措施。Ollydbg，Windbg，IDA 和 GDB 工具都能胜任这个任务。这种情况下要解决的问题是，发现究竟是哪个测试用例或测试引发了异常行为。

14.4.2　自动化方式（基于代理）

可以设计一个方案来代替手工调试过程。不使用调试器，而是由模糊测试者编写一个面向目标平台的调试代理并在目标应用上运行之。调试代理要完成两个任务：第一个

任务是监视目标进程中发生的异常，另一个任务是与远程系统上的模糊测试器进行通信。这样就可以轻松地将数据和故障检测关联起来。这种方法的缺点是，对于每一个需要运行测试目标的平台，开发者都需要创建一个代理。在第 24 章中我们将对这一概念进行进一步研究。

14.4.3 其他来源

虽然调试器可能是网络模糊测试中最有价值的检测异常的工具，不过，也不要因此忽视其他可用的线索。应用日志和操作系统日志同样能够提供所发生问题的信息。与手工附加调试器到应用上一样，这里的挑战在于将发现的问题关联到模糊测试用例上。此外，谨记关注系统的性能下降。系统性能下降是发现隐藏问题的指示器。性能下降的表现包括持续上涨的 CPU 使用率，或是内存耗尽。一个导致死循环的测试用例也许不会触发异常，却会导致一个 DoS。这里，我们的观点很简单：虽然调试器是发现漏洞的出色工具，但你不该因此忽视其他有价值的线索。

234

14.5 小结

网络协议模糊测试也许是模糊测试最知名和得到最广泛应用的领域。可以用多种方法进行网络协议模糊测试。导致网络协议模糊测试变得普遍的原因多种多样，但通过模糊测试能够发现高风险、远程的、预认证（preauthentication）类的漏洞是其中的一个重要原因。此外，网络协议模糊测试也是一种成熟的模糊测试类型，原因之一是目前已经有相当多公开可用的，用于支持安全研究者工作的工具。了解了通用的网络模糊测试方法之后，是时候讨论如何实现基于网络的模糊测试器了。

第15章

UNIX平台上的自动化网络协议模糊测试

"I think we agree, the past is over."

——George W. Bush, on his meeting with John McCain,
Dallas Morning News, May 10, 2000

尽管微软 Windows 操作系统统治了桌面领域，但 UNIX 系统仍然占据着服务器平台的主要市场份额。例如，根据 NetCraft 最近的调查，主要运行在 UNIX 系统上的 Apache Web 服务器依然领先微软 IIS 服务器差不多 30 个百分点[1]。影响流行 UNIX 服务的漏洞非常严重，可能产生巨大的影响，因为 Internet 的大部分构建在基于 UNIX 的 DNS、邮件和 Web 服务上。例如，考虑这种情况：如果在 Berkeley Internet Name Domain（BIND）DNS 服务器中发现了一个漏洞，那么就有可能通过这个漏洞扰乱大部分的 Internet 通信。虽然无法确切地知道模糊测试方法找到的漏洞占已发现漏洞的比例，但我们确信模糊测试器帮助发现了不少漏洞。

在本章中，我们不打算从头开始开发一个基于 UNIX 的定制模糊测试器。相反，我们会引入 SPIKE 模糊测试框架，使用它提供的脚本接口。SPIKE 是我们在整本书中一

[1] http://news.netcraft.com/archives/2007/02/23/march_2007_wb_server_survey.html

直提到的一个开源模糊测试器。开发模糊框架的编码工作不是本章的重点，因此，我们在本章中提供了一个完整的模糊测试案例，即使用 SPIKE 从头到尾对一个闭源应用进行测试。

236

15.1 使用 SPIKE 进行模糊测试

为了展示用 SPIKE 进行模糊测试的过程，我们将从头至尾讲解一个完整例子，从开始的选择目标，到研究目标协议，再到使用 SPIKE 脚本描述协议，最后是使用模糊测试器。

15.1.1 选择目标

为了方便示例，我们选择的目标需要具有以下特性：

- 目标软件应该是被广泛应用的。
- 目标软件应该容易获得（demo 版本或评估版本）。
- 目标软件使用的协议应该有公开的文档描述，或是容易进行逆向推导。

虽然有很多选择，但根据我们的目的，在本案例中我们选择了 Novell NetMail[2]软件。这是一个企业电子邮件和日程系统，它实现了许多有文档描述的协议，并且满足我们提到的所有要求。我们具体的测试目标是 NetMail 网络消息应用协议（Networked Messaging Application Protocol，NMAP）。注意不要将这个 NMAP 协议与 Nmap[3]弄混，后者是一个应用非常广泛的网络扫描和侦测工具。那么 NetMail NMAP 到底是什么呢？我一下子也没法给出一个清晰的定义，还好 Novell 热心地提供了以下定义：

NMAP 这个缩略词表示网络信息应用协议（Networked Messaging Application Protocol）。这是一种基于文本的 IP 协议，已经在互联网数字分配机构（Internet Assigned Numbers Authority，IANA）注册了端口 689，NIMS 代理使用该端口进行通信。当与具备分布式特征的 NDS eDirectory 结合起来时，这个协议允许运行在不同服务器（甚至是不同平台）上的 NIMS 代理进行互操作，就好像它们在同一台服务器上一样。当需要对消息服务进行扩充的时候，NMAP 不是用更大的服务器替换掉原有服务器，而是允许将新的服务器加入到“集群”中。每一个版本的 NIMS 都会提供 RFC 格式的 NMAP 协

[2] http://www.novell.com/products/netmail/

[3] http://insecure.org/nmap

议文档[4]。

NMAP 看起来是一个 Novell 开发的构建于 TCP 协议之上的专有文本协议。选择这样一个专有协议作为测试目标有好处也有坏处。坏处是，除了供应商外没有人会使用我们开发的模糊测试器。好处是，自定义协议意味着没有现成的，经过长期试用表明不存在问题的解析库。供应商必须得自行编写协议解析器，很少人会审查这些代码，因此它可能会包含很多漏洞。

Novell 公司非常体贴地在官网上提供了 NetMail 的 90 天免费评估版[5]，因此我们可以很容易下载并在实验环境中安装这个软件。安装完毕后，我们用 Novell 提供的补丁将软件更新到最新版本。这个关键的步骤确保我们不会找出旧的缺陷（如果我们能找到的话）。如果你已经审查了一周，才发现忘了这个步骤，那肯定让人觉得沮丧。接下来，我们开始详细检查 NMAP 协议。

15.1.2　协议分析

在能够向 SPIKE 描述 NMAP 协议之前，我们需要先理解这个协议。有多种方法可以帮助我们理解协议。最容易想到的方法是在试验环境里监视合法的 NMAP 通信数据。但实际上，应用这个方法比你想象的要困难一些。当对复杂的企业软件进行模糊测试时，生成你需要的数据流有时候非常困难。另一种方法是利用他人已有的工作成果。Google 一下已知的关于协议的信息总是没有坏处。查看开源的 Wireshark[6]（原来的 Ethereal）嗅探器中是否有特定协议的解码器也是个好主意。找到 Wireshark 的 Subversion 库，直接跳到 epan\dissectors 目录，查看是否有你要找的协议。

另一种不常使用的方法是直接与 NMAP 的服务进程通信，看它是否能够提供一些信息。要使用这种方法，首先需要知道应用程序使用哪个端口与客户端通信。假如 NMAP 的文档没有公开说明它绑定的是 TCP 端口 689，我们可以在微软 TCPView[7]（原来是属于 SysInternals 的）程序的帮助下手工将其找出来。启动 TCPView 程序就能直接看到 nmapd.exe 正在监听 TCP 端口 689。我们使用基础 TCP 连接工具（如 netcat[8]或 Windows

[4] http：//support.novell.com/techcenter/articles/ana20000303.html

[5] http://download.novell.com/index.jsp

[6] http://anonsvn.wireshark.org/wireshark/trunk

[7] http://www.microsoft.com/technet/sysinternals/Networking/TcpView.mspx

[8] http://www.vulnwatch.org/netcat/

telnet 命令等）连接该服务进程，连接成功后大胆猜测，输入“HELP”命令，此时你会
238 惊喜地发现眼前出现了合法命令列表。

这么一会儿的工作成果还真不赖。用我们搜集到的信息继续下一步，将 nmapd.exe 二进制代码加载到我们最爱的 IDA Pro 反编译器中。按下 Shift + F12 能够看到图 15.1 显示的字符串数据库。然后我们定位到看上去像是响应“HELP”命令的字符串上，该字符串以 ASCII 文本“1000”开始的。现在我们按照字符串的内容，而不是字符串的地址对字符串数据库排序，滚动到以“1000”开头的行。在这里我们就找到了 NMAP 服务器支持的所有命令列表和它们期望的语法。

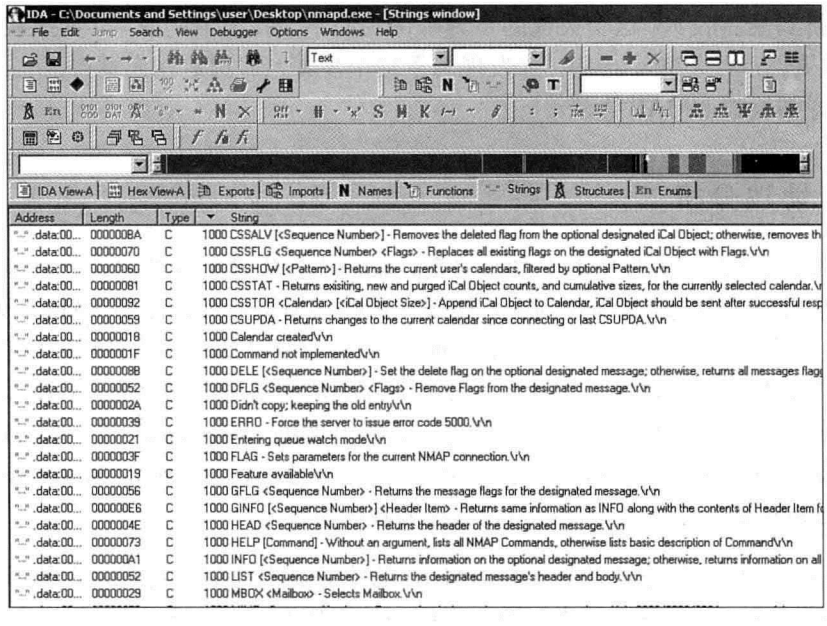


图 15.1　nmap.exe 执行中找到的字符串列表

239

浏览了各种命令和它们的描述语法后，“HELP”命令的输出含义也就清楚了。尽管有些例外，但大多数命令的语法都可以用以下原型描述：

- `<argument>`表明该参数是必需的。如果没有给出，命令将失败。
- [argument]表明该参数是可选的。
- `{CONSTANT1|CONSTANT2|CONSTANT3}`表明该参数是必需的，并且其取值必须是给定常量字符串中的一个，命令才能成功。每一个可能的值用管道字符“|”分隔。

此外，没有用尖括号括起来的字符串都被当做普通字符串，尖括号内的字符串被当做变量。同样的规则也适用于嵌套的语法规则。例如，PASS 命令可以描述如下：

```
PASS {SYS | USER <Username>} <Password>
```

该原型说明 PASS 命令后面总是跟着字符串“SYS”或“USER”中的一个。由于变量 Username 在尖括号内，并且跟在字符串文字 USER 之后，因此选择“USER”时该参数是必须的。如果选择“SYS”，Username 参数就不是必需的。变量 Password 是命令的最后一个参数，是必需的。PASS 命令可能是这个协议中最复杂的一条命令，但即使这样，它也不算太复杂。看起来我们选择了一个非常友好的协议进行模糊测试。

由于上面给出的命令（PASS 命令）处理用户认证，所以估计这个服务进程中至少有一部分功能需要认证。取决于测试的范围，模糊测试者可能会也可能不会关心这些功能。例如，对一个急要赶工的研究者小组来说，他们可能仅仅测试应用最核心的状态（未认证状态），而很可能忽略需要认证之后才能使用的命令。然而，对于涵盖所有状态的完整测试来说，模糊测试器需要确保验证完未认证状态后，还能够正确地对认证后的状态进行验证。

如果对认证后状态的模糊测试感兴趣，我们必须确定怎样成功登录。我们知道有多种方法用来进行登录。在 FTP 协议中，一个用户可以先使用 USER 命令，再使用 PASS 命令来登录。此外，用户也可以选择在 PASS 命令中给出用户名。第三种类型是允许其他 NMAP 代理使用 PASS 命令中的 SYS 指令而不是 USER 指令来认证。

在花了一些时间查看 IDA 找到的其他命令及其详细描述后，我们对协议的工作方式有了初步了解。粗略地说，每条命令都以单一的 ASCII 字符串开头，我们称之为“动词”，“动词”告诉服务进程我们要执行的动作。命令的下一部分是空格分隔符。在空格之后，要么是换行（动词不需要参数的情况），要么是跟动词相关的参数。参数的格式参见从 IDA 中找到的字符串。

有了这些关于 NMAP 协议的基本信息，就可以开始构建我们自己的 SPIKE NMAP 模糊测试器了。

15.2　SPIKE 必要知识

我们将在第 21 章介绍 SPIKE 和其他模糊测试框架。但为了更好地理解本章内容，读者需要具备一定的 SPIKE 知识。要使用 SPIKE 对以上的协议进行模糊测试，需要熟

悉 SPIKE 模糊测试引擎，以及它的通用的，脚本化的，基于命令行的模糊测试器工作方式。

15.2.1 模糊引擎

SPIKE 用模糊字符串库中的内容迭代模糊变量，达成模糊测试。可以把变量理解成协议中的字段，如用户名、口令、命令等。这些变量的值和它们在数据流中的位置取决于被测目标。模糊字符串库由多种可能引发错误的字符串和二进制数据组成，模糊字符串库中的元素是根据过去的经验仔细选择得到的特定序列，这些序列曾经引发过其他软件的问题。例如，考虑一个非常基础的模糊字符串：包含 64,000 个连续字符 A 的 ASCII 字符串，该模糊字符串可能会被用来替换协议中的 username 变量以触发一个发生在认证之前的缓冲区溢出。注意，“模糊字符串”这个词带有一点歧义。实际上，模糊字符串可以是任何数据类型，甚至是 XDR 编码的二进制数据数组。

15.2.2 基于行的通用 TCP 模糊测试器

我们要在这里构建的模糊测试器被称为基于行的通用 TCP 模糊测试器。该应用的代码位于 SPIKE 源码包的 line_send_tcp.c 文件中。这个模糊测试器是对 SPIKE 的一个简单封装，但却功能强大。它能够处理 SPIKE 脚本，模糊化脚本中的每个变量，重复连接到服务器并尝试每个模糊值。脚本中的变量会按照它们在模糊脚本中的顺序被模糊化。这意味着在大多数包含认证的协议中，SPIKE 会首先测试认证相关的信息。

241

该模糊测试器实现的脚本语言允许程序员直接调用 SPIKE 的 API 函数。为了让你对如何写脚本有基本的理解，下面列出一些可能会用到的函数。

- **`s_string(char * instring)`**：该函数将一个固定字符串添加到 SPIKE。被加入的字符串的值不会被修改。
- **`s_string_variable(unsigned char *variable)`**：该函数将一个可变字符串添加到 SPIKE。这个字符串在相应的变量被处理的时候会被模糊字符串替换掉。
- **`s_binary(char * instring)`**：该函数将二进制数据添加到 SPIKE。加入的数据值不会被修改。
- **`s_xdr_string(unsigned char *astring)`**：该函数将一个 XDR 格式的字符串添加到 SPIKE。也就是说，加入的字符串包括一个 4 字节长的标签，并

会用 0 填充至长度为 4 的倍数，加入的数据不会被修改。

- **s_int_variable(intdefaultvalue, int type)**：该函数将一个整数添加到 SPIKE。

当调用 s_int_variable()时，type 的值可以是下列几种之一。

- **Binary Big Endian**：最高有效位整型，4 字节。
- **ASCII**：ASCII 格式的有符号十进制数。
- **One Byte**：一个字节表示的整数。
- **Binary Little Endian Half Word**：最低有效位整数，2 字节。
- **Binary Big Endian Half Word**：最高有效位整型，2 字节。
- **Zero X ASCII Hex**：以 0x 开头的 ASCII 格式的十六进制数。
- **ASCII Hex**：ASCII 格式的十六进制数。
- **ASCII Unsigned**：ASCII 格式的无符号十进制数。
- **Intel Endian Word**：最低有效位整型，4 字节。

由于我们将 SPIKE 用作脚本宿主时没有使用 C 预处理器，因此脚本运行时需要知道以上各类型对应的整数值。下面列出了 SPIKE/SPIKE/include/listener.h 中定义的各类型的整数值：

```
#define BINARYBIGENDIAN 1
#define ASCII 2
#define ONEBYTE 3
#define BINARYLITTLEENDIANHALFWORD 4
#define BINARYBIGENDIANHALFWORD 5
#define ZEROXASCIIHEX 6
#define ASCIIHEX 7
#define ASCIIUNSIGNED 8
#define INTELENDIANWORD 9
```

目前为止我们讨论过的 SPIKE 功能已经足够我们开发一个 NMAP 模糊测试器了。然而，还是让我们接着看看 SPIKE 对模糊领域的主要贡献：基于块的协议模型。

15.3　基于块的协议模型

尽管我们这次选择的测试目标是一个简单的基于文本的协议，但实际上，由于 SPIKE 具有基于块的模糊能力，它还可以支持更复杂的协议。基于块的模糊测试允许根据有效的字段长度动态创建数据包。例如，考虑这样一种协议，在它的包结构中，

username 字段的前面前置有该字段的长度。我们想让模糊器在模糊化 username 的时候自动更新长度字段。自动更新长度字段能够确保模糊测试字符串能顺利通过 username 解析器的检查。可以使用 s_block_start() 函数和 s_block_end() 函数对该协议建模，这两个函数的声明如下：

```
int s_block_start(char *blockname)
int s_block_end(char *blockname)
```

只需简单地将这两个函数放在需要度量的字段之前和之后即可。这样，当需要将实际的长度值插入到数据流时，可以使用几个 blocksize 函数中的一个。与之前讨论过的整型字段类似，这几个 blocksize 函数的名称比较冗长。注意，有些 blocksize 函数是变量（variable），这意味着会对非法的 blocksize 值进行一轮模糊测试。是否对长度字段进行模糊测试取决于执行者。下面是能够在 SPIKE 中使用的 blocksize 类型的几乎完整的列表：

```
s_blocksize_signed_string_variable(char * instring, int size)
s_blocksize_unsigned_string_variable(char * instring, int size)
s_blocksize_asciihex_variable(char * blockname)
s_binary_block_size_word_bigendian(char *blockname)
s_binary_block_size_word_bigendian_varible(char *blockname)
s_binary_block_size_halfword_bigendian(char *blockname)
s_binary_block_size_halfword_bigendian_variable(char *blockname)
s_binary_block_size_byte(char *blockname)
s_binary_block_size_byte_variable(char * blockname)
s_binary_block_size_byte_plus(char * blockname, long plus)
s_binary_block_size_word_bigendian_plussome(char *blockname, long some)
s_binary_block_size_intel_halfword(char *blockname)
s_binary_block_size_intel_halfword_variable(char *blockname)
s_binary_block_size_intel_halfword_plus_variable(char *blockname, long plus)
s_binary_block_size_intel_halfword_plus(char *blockname, long plus)
s_binary_block_size_byte_mult(char * blockname, float mult)
s_binary_block_size_halfword_bigendian_mult(char * blockname, float mult)
s_binary_block_size_word_bigendian_mult(char * blockname, float mult)
s_binary_block_size_intel_word(char *blockname)
s_binary_block_size_intel_word_variable(char *blockname)
s_binary_block_size_intel_word_plus(char *blockname, long some)
s_binary_block_size_word_intel_mult_plus(char *blockname, long some, float mult)
s_binary_block_size_intel_halfword_mult(char *blockname, float mult)
s_blocksize_unsigned_string_variable(char * blockname)
s_blocksize_asciihex_variable(char * blockname)
```

15.4　其他的 SPIKE 特性

SPIKE 不仅仅是一个模糊测试器，它还是一个全功能的模糊测试框架，包含了大量有用的函数 API。这些函数能够帮助简化创建自定义模糊测试器。此外，它还包含了一大堆面向特定协议和特定应用的模糊测试器及模糊测试脚本。下面列出了一些 SPIKE 包含的，SPIKE 引擎和通信代码之外的其他东西。

15.4.1　针对协议的模糊测试器

SPIKE 包含一部分预先写好的针对具体协议的模糊测试器。以下是这些模糊测试器的列表：

- HTTP 模糊测试器
- Microsoft RPC 模糊测试器
- X11 模糊测试器
- Citrix 模糊测试器
- Sun RPC 模糊测试器

在大多数情况下，这些模糊器最好的用途仅仅是演示如何使用 SPIKE。这是因为它们已经存在了较长的一段时间，可能已经针对你感兴趣的每个目标都运行过多次。

15.4.2　针对协议的模糊测试脚本

244

SPIKE 还包含一些可以嵌入到多个 SPIKE 内含的通用模糊测试器中的脚本。脚本列表如下所示：

```
CIFS
FTP
H.323
IMAP
Oracle
Microsoft SQL
PPTP
SMTP
SSL
POP3
```

15.4.3 基于脚本的通用模糊测试器

前面我们提到过，SPIKE 中有几个通用模糊测试器，它们接受脚本作为输入。下面列出你能在 SPIKE 中找到的通用模糊测试器：

- TCP 监听模糊测试器（客户端）
- TCP/UDP 发送模糊测试器
- 行缓冲 TCP 发送模糊测试器

15.5 编写 SPIKE NMAP 模糊测试器脚本

现在回到我们的测试目标 NetMail 上来，下面示例的 SPIKE 脚本完完全全建立在我们之前通过 IDA Pro 得到的知识上，而这些知识来源于“HELP”命令输出的消息和字符串列表。该模糊脚本从对认证命令进行模糊测试开始，寻找可能的未认证前的缺陷。如果提供了正确的用户名和口令，模糊测试脚本会继续对认证后的命令进行模糊测试。

245

```
s_string_variable("PASS");
s_string(" ");
s_string_variable("USER");
s_string(" ");
s_string_variable("devel_user");
s_string(" ");
s_string_variable("secretpassword");
s_string("\r\n");

s_string("QCREA ");
s_string_variable("test");
s_string("\r\n");

s_string("CREA ");
s_string_variable("inbox");
s_string("\r\n");

s_string("MBOX ");
s_string_variable("test");
s_string("\r\n");

s_string("LIST ");
```

```
s_string_variable("0");
s_string("\r\n");

s_string("GINFO ");
s_string_variable("0");
s_string(" ");
s_string_variable("test");
s_string("\r\n");

s_string("SEARCH BODY ");
s_string_variable("test");
s_string("\r\n");

s_string("DFLG ");
s_string_variable("0");
s_string(" ");
s_string_variable("SEEN");
s_string("\r\n");

s_string("CSCREA ");
s_string_variable("test");
s_string("\r\n");

s_string("CSOPEN ");
s_string_variable("test");
s_string("\r\n");

s_string("CSFIND ");
s_string_variable("0");
s_string(" ");
s_string_variable("0");
s_string(" ");
s_string_variable("0");
s_string("\r\n");

s_string("BRAW ");
s_string_variable("0");
s_string(" ");
s_string_variable("0");
s_string(" ");
s_string_variable("0");
s_string("\r\n");
```

246

接下来，我们选择一个调试器并将其连接到目标系统上的 NMAP 进程，通过 SPIKE 运行我们开发的脚本。使用 SPIKE 的基于行的通用 TCP 模糊测试器，执行名为 nmap.spk 的脚本，命令行如下：

```
./line_send_tcp 192.168.1.2 689 nmap.spk 0 0
```

开始运行没多久，模糊器就发现了一个可利用的栈溢出。用 OllyDbg 调试器看到的处于漏洞状态下的 NMAP 服务进程状态如图 15.2 所示。

图 15.2 中右上面板显示了崩溃时寄存器的值。寄存器 EBP（堆栈帧指针），EBX，ESI，EDI 和最重要的 EIP（指令指针）都被覆写成了十六进制值 0x41，也就是 ASCII 字符“A”。右下面板显示的是堆栈帧，我们可以清楚地看到堆栈帧被一长串的“A”覆写了。左上面板通常显示的是正在执行的指令列表，但图中显示为空，因为指令指针指向的是 0x41414141。没有内存页被映射到地址 0x41414141，因此反编译窗口没什么可以显示的。接下来，我们必须追踪出是哪个动词-参数对导致了崩溃，以便我们能够重现甚至是编写一个可以利用该漏洞的应用。

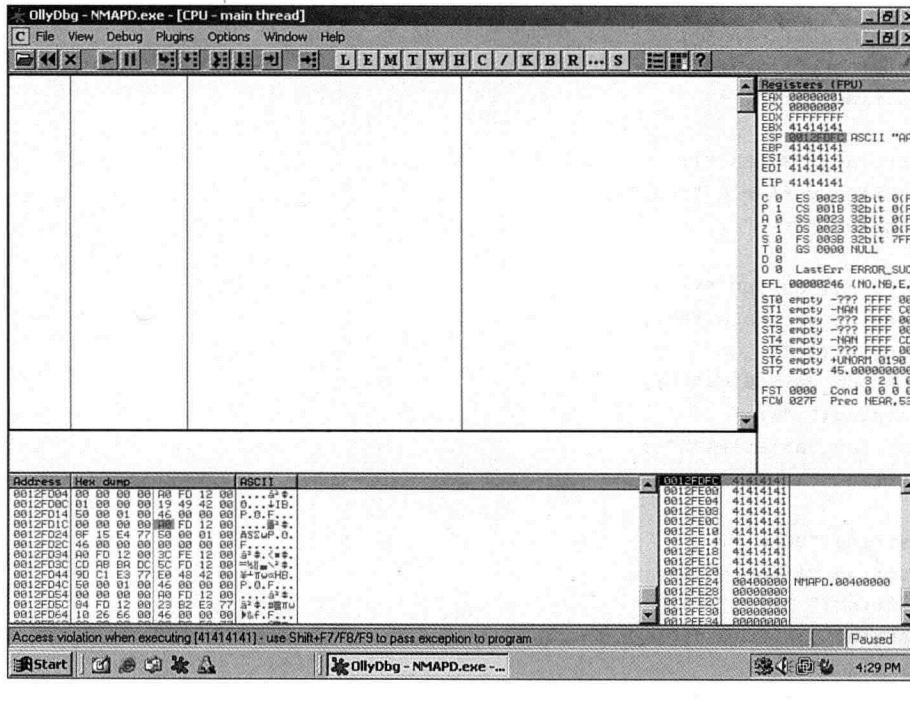


图 15.2　nmapd.exe 中的可利用堆栈溢出

有多种方法可以追踪到导致崩溃的动词-参数对。最简单的一种方法是利用 SPIKE 的一个内置特性。如果连接不到目标，SPIKE 通常会发生崩溃。通过检查 SPIKE 崩溃之前的最后输出，我们就能够知道究竟是哪个测试用例导致了 NetMail NMAP 的崩溃。当然，如果我们在目标进程上连接了调试器，目标进程就不会崩溃而是会挂起。如此一来，SPIKE 的崩溃“特性”就不会出现。因此，为了重现崩溃，我们重启进程，让它运行在非监视情况下（不连接调试器），依靠 SPIKE 因为 NMAP 服务进程失败而发生的崩溃来检测错误。再次运行测试并等待，这次我们又观察到了 NMAP 的崩溃，随后是 SPIKE 的崩溃，并伴随以下的输出：

247

```
-snip-
Fuzzing Variable 5:1
Read first line
Variablesize= 5004

Fuzzing Variable 5:2
Couldn't tcp connect to target
Segmentation fault
-snip-
```

248

在追踪导致 NMAP 出错的测试用例时，以上这种方法肯定是最初级的技术。一种稍微“科学”一点的办法是用嗅探器监视所有的网络通信。NMAP 崩溃后就再也不能响应请求了，但 SPIKE 会持续发送测试用例。通过定位有响应的最后一次传输，就能知道哪个测试用例可能触发了这个故障。

回到我们最原始的回溯技术上，我们发现 SPIKE 发出的最后一次成功的传输带有“Fuzzing Variable 5:1”这行文字。这表示最后一次成功连接使用的是模糊变量 5。发给进程的最后一个模糊字符串是模糊字符串 1。要确定哪个模糊变量是变量 5，只需打开 SPIKE 脚本，从 0 开始数包含“Variable”的行。结果是 CREA 命令，这是一个认证后才可用的动词。下一步，我们必须确定模糊字符串 1 使用的值，也就是 CREA 命令的参数。

有几种方法可以找到 CREA 的参数：一种是使用嗅探器，就像上文描述的那样。另一种方法是在 line_send_tcp.c 代码中加入 printf() 函数，让应用程序打印出当前的模糊字符串，然后简单地重新运行模糊测试器。无论使用哪种方法，我们都能找到有问题的字符串是“CREA <长字符串>”。根据这个结果，我们立即就能重现这个认证后的崩溃。用户只需登录进去，发送一个恶意的 CREA 命令参数就行了。看起来这并没有多难：花点时间就找到了一个远程漏洞。你可能会想，为什么就没有更多的供应商在发布

他们的软件之前使用这种形式的测试呢？如果我们想继续对 NMAP 进行模糊测试，简单地从 SPIKE 脚本中去掉 CREA 命令的部分即可，这样目标应用就不会因为这个已知问题不停地中断。

15.6 小结

当想要编写自己的模糊测试器时，首先评估现有的模糊测试器和模糊测试框架很重要。对于简单目标（例如 NMAP 协议）来说，从头开发一个模糊测试器意义不大。利用 SPIKE，仅需要几个小时，我们就能写出一组 SPIKE 脚本来有效地测试 NMAP 服务进程的代码。然而，需要记住的是，测试结果的质量取决于你投入的开发模糊测试器的时间。在这个具体的案例中，我们只投入了最少的精力。我们选择登录，发送几个命令请求，然后登出。在 NMAP 中，还能找到数十个其他函数，这些函数都可以加到你的脚本中进行更进一步的测试。

第16章

Windows平台上网络协议的模糊测试

"I couldn't imagine somebody like Osama bin Laden understanding the joy of Hanukkah."

——George W. Bush, White House Menorah lighting ceremony, Washington, DC, December 10, 2001

UNIX 系统也许在服务器端占据了统治地位，但全球范围内安装数量最多的仍然是微软的 Windows 操作系统，也因此，微软的 Windows 操作系统成为了易受攻击的目标。现如今，攻击者经常利用 Windows 桌面版的漏洞来建立僵尸网络（bot nets）。我们以 Slammer 蠕虫[1]为例，说明出现在 Windows 系统中的网络相关的漏洞有多可怕。Slammer 蠕虫利用了微软 SQL Server 中的一个缓冲区溢出漏洞（微软 2002 年 7 月 24 日发布的安全公告 MS02-039[2]描述了该漏洞的细节，而 Slammer 蠕虫出现于 2003 年 1 月 25 日）。Slammer 蠕虫基本上什么也不干，仅仅是在网络中扫描存在该漏洞的机器，并将自身传播到存在漏洞的机器上[3]。尽管该蠕虫自身不做什么破坏性的事情，但重复不断扫描产

[1] http://www.cert.org/advisories/CA-2003-04.html

[2] http://www.micsrosoft.com/technet/security/bulletin/MS02-039.mspx

[3] http://pedram.openrce.org/__research/slammer/slammer.txt

生的大量流量致使互联网和信用卡处理业务发生崩溃，在某些情况下甚至影响到电话网络的可用性。关于 Slammer 蠕虫，最有趣的地方在于，即使在它首次出现的 4 年之后，该蠕虫仍然名列产生流量事件的前五名之列[4]。显然，Windows 平台上暴露出来的网络漏洞影响深远。

250

在上一章中，我们利用一个已存在的模糊测试器框架 SPIKE，在 UNIX 环境上建立了一个针对 Novell NetMail NMAP 后台协议的模糊测试器。在本章中，我们将采用不同的方法，从头至尾地创建一个简单的，基于 Windows 的，具有 GUI 界面的，用户友好的模糊测试器。我们为这个模糊测试器取名为 ProtoFuzz，虽然它仅提供了基础的功能，但是作为一个扩展性良好的平台，该工具提供了创建模糊测试器的不同视角。让我们从它的基本特性开始。

16.1 特性

在开始开发该工具之前，需要首先考虑我们需要和期望的产品特性。在最基础的层面上，协议模糊测试器只需要简单地将变异后的数据包发送给目标应用。只要模糊测试器具备了生成数据包和发送数据包的能力，它基本就能工作了。然而，如果模糊测试器能够理解我们需要进行模糊测试的数据包结构，那一定更好。因此，接下来，我们对基本需求进行一些扩展。

16.1.1 数据包结构

模糊测试器需要首先理解如何构建一个数据包，然后才能发送数据包。现有的模糊测试器采用以下三种方法组装模糊测试需要的数据包。

- **精心制作测试套件**：无论是 PROTOS Test Suite[5]工具，还是它的商业版本 Codenomicon[6]，都是在程序中通过硬编码的方式定义模糊测试需要的数据包结构。创建这种测试套件需要消耗大量时间，因为这种方法需要对协议规范进行分析，以及开发出上千个硬编码方式的测试用例。
- **数据包生成方法**：在前面的章节中我们看到，SPIKE 这样的工具需要用户建立

[4] http://isc.sans.org/portreport.html?sort=targets

[5] http://www.ee.oulu.fi/research/ouspg/protos/

[6] http://www.codenonicon.com/products/

描述数据包结构的模板。然后，模糊测试器负责在运行时生成和传输测试用例。

- **数据包变异方法**：与从头开始创建一个数据包不同，一种替代方案是从一个已有的合法数据包开始，对该数据包的部分内容进行变异。虽然可以通过强制方式对数据包的每个字节进行变异，但这种方法一般来说并不适用于协议模糊测试，因为这种方法效率很低，而且可能会产生一大堆不符合网络协议要求的数据包，这些数据包甚至都不能通过网络成功发送到给定目标。我们可以借鉴基于协议模板的方法对此进行一点调整：用户基于已有的合法数据包创建一个模板，在模板中指明需要修改的部分，从而满足模糊测试的需要。这就是我们在 ProtoFuzz 工具中实现的方法。

这三种方法中没有哪种方法比其他两种更优越，只能说，每种方法都有其适合的情况。ProtoFuzz 工具选择数据包变异方法的原因仅仅是因为它实现起来最简单，只需要从网络流量数据中抓取一个合法的数据包，就可以进行模糊测试，完全不需要对协议进行研究或是创建用于生成数据的模板。

16.1.2　抓取数据

由于我们已经选取了数据包变异作为模糊测试方法，因此模糊测试工具 ProtoFuzz 应该能够像 sniffer 工具一样，具有在混杂模式（promiscuous mode）下抓取网络数据的能力。我们将在 ProtoFuzz 中创建一个网络协议分析器，使得该工具能够抓取发送给目标应用以及从目标应用发出的数据，这样我们就能够从中选择用于模糊测试的数据包了。为了实现这些功能，我们会利用一个已有的数据包抓取库。该库将在“工具开发”这一节中进行详细介绍。

16.1.3　解析数据

虽然不是绝对必要，但除了抓取数据外，我们还希望 ProtoFuzz 工具能够以易于被理解的格式展现被抓取数据的内容。这种方式能够帮助用户找到数据包中适合进行模糊测试的位置。网络上传输的数据是一系列符合某种定义的字节序列，要想将这些数据以人工可读的格式显示出来，就需要利用一个能够理解包结构，并能将数据包分解为协议头和数据段的数据抓取库。由于许多人熟悉 Wireshark 工具使用的数据表示格式，因此我们将使用 Wireshar 的数据表示格式作为我们基础的数据显示格式。图 16.1 展示了 Wireshark 工具抓取的一个简单的 TCP 包。

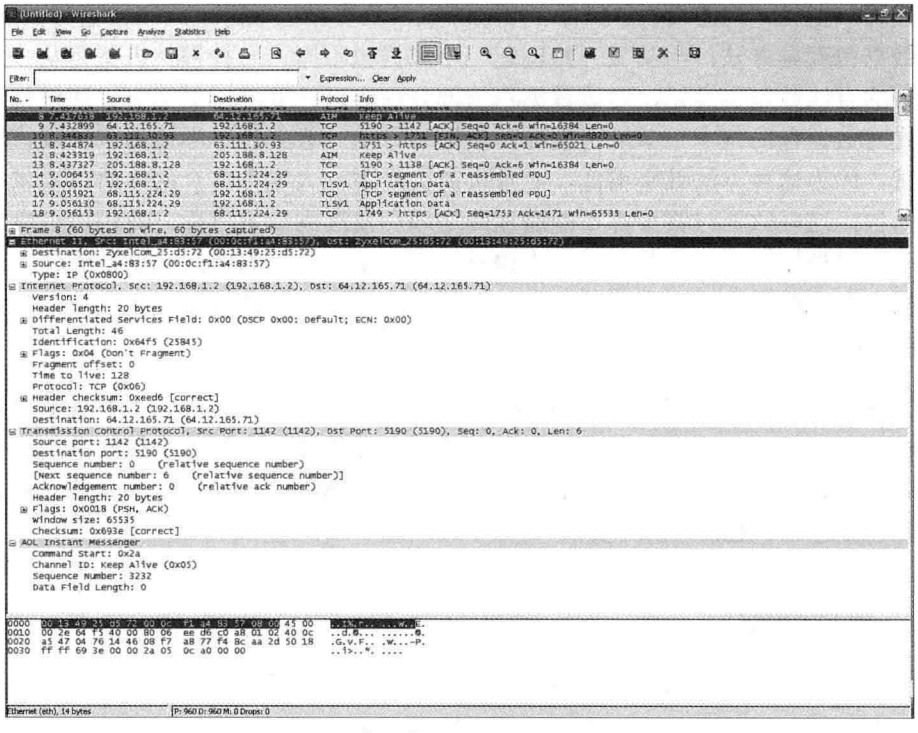


图 16.1 Wireshark 正在解析一个 AIM 的“保持活跃”数据包

来自 Wireshark 的这个屏幕截图表明其显示区域被分成三个面板。最上面的面板列出了所有抓取到的数据包。选中其中任何一个数据包，在下面两个面板中就会显示该数据包的内容。中间面板显示的是被分解成不同字段的数据包内容。在本例中，我们可以看到，Wireshark 很了解 AOL 即时通信工具的协议，它成功地解码了该 TCP 包的全部数据段。最下面的面板则直接以十六进制和 ASCII 方式显示了被选中数据包里的数据。

16.1.4 模糊测试变量

在观察和抓取到网络数据后，我们需要允许用户标识数据包中适合进行变异的位置，以便进行模糊测试。为此，我们允许用户在以十六进制方式显示的数据内容中加入简单的标签，标明进行模糊测试的部分。不同的标签表示不同类型的运行时模糊测试变量。这种简单格式不仅允许用户以可视的方式识别模糊测试变量，也使得 ProtoFuzz 工具在解析数据包结构时能够找到可以用模糊测试数据替换的区域。ProtoFuzz 工具使用

下面这些标签。

- **[XX] – 表示“强制”**：ProtoFuzz 工具将使用所有可能的字节值对方括号括起来的字节进行模糊测试。因此，方括号中的一个字节将会被模糊测试 256 次，一个字（双字节值）会被模糊测试 65,536 次。
- **<XX>- 表示“字符串”**：字符串标签允许 ProtoFuzz 工具从用户控制的文本文件（strings.txt 文件）中取得的预定义的,可变长的,以十六进制方式表示的字符串进行模糊测试。这类模糊测试通常用在数据包的数据部分，而不是用在需要定义好的结构的数据包头部分。

下面的模板展示了一个同时包含强制变量和字符串模糊测试变量的 TCP 包。

```
00 0C F1 A4 83 57 00 13 49 25 D5 72 08 00 45 00 00 28<B0 3B>00 00 FE 06
89 40 C0 A8 01 01 C0 A8 01 02 08 A6 0B 35 14 9E E1 9F 9F 33 69 E5 50 11
10 00 09 4E 00 00 01 00 5E 00 00[16]
```

16.1.5　发送数据

协议库通常采用两种方法来发送数据。第一种方法是，协议库提供一系列功能允许用户设置数据包中的特定部分，例如设定目标 IP 和 MAC 地址。但是，数据包的主体部分是根据 RFC 生成的，比如 HttpRequest 这样的.NET 类，该类允许你定义诸如方法和 URL 之类的属性，但是会自动生成大多数 HTTP 头，以及所有的 Ethernet、TCP 和 IP 头。另一种方法则是创建一个可以由用户指定数据包中任何字节的值的裸数据包，由程序员来保证数据包符合 RFC 定义的结构。虽然第二种方法需要更多的研究工作，但这种方法提供了更好的细节控制能力，使得用户可以对协议头进行模糊测试。

16.2　必备的背景知识

254

在开始讨论开发 ProtoFuzz 工具之前，需要首先给出一些重要的背景信息。

16.2.1　检测故障

对所有类型的模糊测试来说，检测目标应用中的故障对于发现漏洞都非常关键。没有完美的故障检测方法，但某些检测方法显然要更好一些。网络协议模糊测试与文件模糊测试不同，其中一个障碍是，模糊测试器和目标应用可能会分布在两个不同系统中。

当对网络协议进行模糊测试时，在目标应用上附加调试器是一个好开端，这种方法可以发现目标应用中所有已被处理和未被处理的异常——几乎也只能依靠这种方法发现这些异常。然而，即使使用调试器发现了所有异常，你仍然需要想办法找到究竟是哪些模糊数据包触发了异常。一个不是100%可靠的能解决该问题的方法是，每次向目标应用发送数据包之后，紧接着发送一个探测数据，用以确认目标应用仍然能够响应请求。例如，在每次发送完一个模糊测试数据包后，我们向目标应用发送一个 ping 数据包，确认收到响应后再送出下一个模糊测试数据包。使用 ping 方法当然称不上完美，因为即使目标应用发生了异常，也仍可能会响应 ping 数据包。然而，我们可以通过定制探测数据让这个方案变得更完备。

1. 性能下降（Performance Degradation）

除检测故障外，我们还可以监视应用的性能下降。例如，如果某个模糊测试数据包导致目标应用中发生死循环，那就不会发生实际的故障，而会导致一个拒绝服务（DoS）条件。在目标主机上运行性能监视器，例如“性能日志与警告”工具或是“系统监视器工具”（在微软管理控制台中能够找到这些工具）能够帮助发现这些情况。

2. 请求超时和非预期响应

不是所有发送给目标应用的数据包都会产生一个响应。然而，对于产生响应的数据包，监视响应数据，确保目标应用仍然能够正常工作很重要。这需要一个额外的步骤，即解析响应数据包的内容。通过对响应数据包进行解析，我们不仅能够确认是否收到了响应，还能发现包含非预期数据的响应。

255

16.2.2 协议驱动程序

许多抓取数据包的库需要使用定制的协议驱动程序。ProtoFuzz 工具选用的 Metro Pack 库[7]就包含了 ndisprot.inf 这个定制的协议驱动程序。ndisprot.inf 是微软在驱动开发套件（Driver Development Kit）中提供的网络驱动接口规范（Network Driver Interface Specification，NDIS）协议驱动程序的示例程序。NDIS 是一种高效的网络适配器 API，向应用提供了发送和接收裸以太网数据包的能力。在能够使用 ProtoFuzz 工具前，用户必须手工安装该驱动程序并通过在命令行运行“net start ndisprot”命令启动之。使用这种依赖定制的协议驱动程序的库的缺点是，定制的协议驱动程序未必能处理所有类型的

[7] http://sourceforge.net/projects/dotmetro/

网络适配器。以 ProtoFuzz 使用的 Metro 库为例，它使用的协议驱动程序就不支持 Wireshark 的网络适配器。

16.3　开发

目前已经有不少能够工作得很好的协议模糊测试器了。如果打算创建一个新工具，就要避免创建一个没有特点的项目，要提供一些其他工具不能提供的东西。考虑到许多已有的模糊测试器都以命令行方式运行，我们希望 ProtoFuzz 工具在以下方面表现得与众不同：

- **直观**。使用 ProtoFuzz 工具不需要经历很长的学习曲线。使用者应该能够在无须使用手册，也无须记住一堆烦人的命令行参数的情况下使用该工具的基本功能。
- **易用**。ProtoFuzz 工具应该在启动后就立即可以使用。用户无须创建冗繁的模板定义数据包结构，工具能够帮助用户利用抓取到的数据包建立模糊测试模板。
- **能够访问所有的协议层**。有些网络模糊测试器仅关注数据包的数据部分，而不关心协议头。ProtoFuzz 应该能够对数据包的所有位置进行模糊测试，从数据包的以太网头到 TCP/UDP 数据部分。

我们创建 ProtoFuzz 工具的目标当然不是为了替代现有的工具。反之，我们期望创建一个 Windows 环境下的网络模糊测试基础平台，期望这个平台既是一个学习模糊测试的工具，同时也是一个开源的，可被任何感兴趣的第三方进行扩展的项目。

16.3.1　选择编程语言

256

与创建 FileFuzz 工具时的选择一样，在 Windows 平台上创建网络协议模糊测试器时，我们选择了 C# 和微软的.NET 框架来创建 GUI 界面的网络协议模糊测试器。.NET 框架处理了许多 GUI 应用开发中的脏活累活，让我们可以将精力集中在应用程序的业务逻辑上（在我们的例子中，应该叫做“破坏业务的逻辑”）。要运行.NET 应用，用户需要预先安装.NET 框架，这可能会带来部分不便。然而，随着基于.NET 的应用的增长和传播，大多数 Windows 系统已经预先安装好了.NET 库，因此，这个需求带来的不便性已经不再严重。

16.3.2　数据包抓取库

在构建网络协议模糊测试器时，关键决策之一是选择合适的网络数据包抓取库。在

ProtoFuzz 工具中，我们选择的这个库需要三个核心组件：抓取、分析、传输。虽然我们可以选择从无到有地创建这些功能，但目前存在大量稳定且开源的数据包抓取工具，完全没有必要从头开始创建这些功能。

由于我们选择在 Windows 平台上开发 ProtoFuzz 工具，因此在选择数据包抓取库时必然会受到限制。通常，在微软的 Windows 上，数据包抓取库的第一选择是 WinPcap[8]。WinPcap 是一个出色的数据包抓取库，该库最早是 Piero Viano 毕业论文的一部分，由 libpcap 库移植到 Windows 平台上。WinPcap 已有多年的历史，目前该库是一个拥有强大代码基础的开源项目，支持大多数主要的协议。实际上，许多商业的和开源的需要处理数据包的应用都需要该库的支持，例如 Wireshark 工具（以前叫做 Ethereal）和 Core Impact[9]工具。WinPcap 的网站上列出了超过 100 个使用该库实现数据包抓取的工具，而这个数字还不包括 WinPcap 团队所不知道的正在使用该库的工具！

虽然 WinPcap 在 ProtoFuzz 的设计考虑之列，但我们选择 C# 作为开发语言的决定限制了我们使用 WinPcap。WinPcap 是用 C 语言编写的，虽然有不少人尝试为 WinPcap 提供 C# 语言的封装，但我们没有看到任何一个此类项目实现了 WinPcap 的全部功能。PacketX[10] 这个用 COM 类库封装的 WinPcap 也在我们的考虑之列，但由于 PacketX 是一个商业工具而作罢。在写作这本书的时候，我们希望本书创建的所有工具都使用可自由发布的库。

257

经过一段时间的寻找，我们很高兴找到了 Metro Packet 库。虽然该库不像 WinPcap 那么健壮，但该库完全用 C# 写成，附带详细的教程和文档，实在是非常适合用来创建我们期望的基础模糊测试器。Metro 的一个弱点是它的数据包解析能力不够强。Metro 包含了用来识别所有高层数据包头的类——包括 Ethernet，TCP，UDP，ICMP，IPv3 和地址解析协议（Address Resolution Protocol，ARP）——但是它不包含设计用来理解头以外的数据的类。再次说明，由于我们为 ProtoFuzz 工具设定的目标比较基础，因此，这不是一个主要的问题。并且，考虑到 Metro 是一个开源的项目，我们可以通过扩展当前类来开发任何需要功能的可能。

16.3.3 设计

好了，理论部分到此为止，是时候开始动手了。与其他工具一样，从 www.fuzzing.org

---

8 http://www.winpcap.org/

9 http://www.coresecurity.com/products/coreimpact/index.php

10 http://www,beesync.com/packetx/index.html

网站上可以找到 ProtoFuzz 的全部源代码。我们不会在本书中检视代码的每个部分，但在本章的后续小节中，我们会强调一些较为重要的代码片段。

1. 网络适配器

我们希望 ProtoFuzz 能够抓取到用于生成模糊测试模板的网络数据。要达成这个目的，使用者必须首先从设置为混合模式的网络适配器嗅探到网络通信数据。与要求用使用者手工提供适配器的名称或标识符相比，我们更愿意为他们提供一个简单的下拉列表菜单，只需要从系统中激活的网络适配器中选择一个即可。幸运的是，Metro 提供了让这个操作变得非常容易的类，使用 Metro 的类创建网络适配器下拉列表菜单的代码如下：

```
private const string DRIVER_NAME = @"\\.\ndisprot";
NdisProtocolDriverInterface driver = new NdisProtocolDriverInterface();

try
{
  driver.OpenDevice (DRIVER_NAME);
}
catch (SystemException ex)
{
  string error = ex.Message;
  error += "\n";
  error += "Please ensure that you have correctly installed the " +
    DRIVER_NAME + " device driver.";
  error += "Also, make sure it has been started.";
  error += "You can start the driver by typing \"net start " +
    DRIVER_NAME.Substring(DRIVER_NAME.LastIndexOf("\\") + 1) +
    "\" at a command prompt.";
  error += "\n";
  error += "Press 'OK' to continue…";
  MessageBox.Show(error, "Error", MessageBoxButtons.OK,
    MessageBoxIcon.Error);
  return;
}

foreach (NetworkAdapter adapter in driver.Adapters)
{
  cbxAdapters.Items.Add(adapter.AdapterName);
  if (cbxAdapters.Items.Count > 0)
    cbxAdapters.SelectedIndex = 0;
}
```

258

在这段代码中，我们首先创建一个新的 NdisProtocolDriverInterface 实例，然后，使用用户已经手工安装好的 ndisprot 网络适配器作为参数调用该实例的 OpenDevice() 函数。如果该适配器不可用，系统会抛出一个 SystemException 异常，对该异常的 catch 操作会提示用户安装和启动该适配器。如果 OpenDevice() 函数调用成功，我们就能得到一个 NetworkAdapter 数组，该数组包含了所有可用的网络适配器的信息。有了这些信息之后，我们使用 foreach 循环遍历数组并将每个网络适配器的 AdapterName 属性加入到 combo box 控件中。

2. 抓取数据

打开适配器后，我们就能够将该适配器设置为混合模式，并开始抓取网络通信数据。

```
try
{
  maxPackets = Convert.ToInt32(tbxPackets.Text);
  capturedPackets = new byte[maxPackets][];
  driver.BindAdapter(driver.Adapters[cbxAdapters.SelectedIndex]);
  ThreadStart packets = new ThreadStart(capturePacket);
  captureThread = new Thread(packets);
  captureThread.Start();
}
catch (IndexOutOfRangeException ex)
{
  MessageBox.Show(ex.Message +
  "\nYou must select a valid network adapter.",
  "Error", MessageBoxButton.OK, MessageBoxIcon.Error);
}
```

在以上代码中，我们首先创建一个二维数组 capturedPackets，该数组用来存储抓取到的数据包数组，而抓取到的数据包数组依次存储数据包中的字节数组。随后，调用 BindAdapter() 方法将选定的网络适配器绑定到前面实例化的 NdisProtocolDriverInterface 实例（对象 driver）上。接着，我们启动一个新线程，在该线程中调用 capturePacket。启动一个新线程来调用 capturePacket 很重要，因为这样可以保证抓取数据时图形用户界面不会被锁住而不能操作。

```
private void capturePacket()
{
  while (packetCount < maxPackets)
  {
    byte[] packet = driver.ReceivePacket();
    capturedPackts[packetCount] = packet;
```

```
        packetCount++;
      }
    }
```

要获得抓取到的数据包的字节数组，使用 ReceivePacket() 方法即可。接下来，只需要将这些数据加入到 capturedPackets 数组中。

3．解析数据

ProtoFuzz 工具的数据解析部分包含两个步骤，工具使用两种方式显示抓取到的数据包。与 Wireshark 的界面布局类似，当用户选中一个抓取到的数据包后，该数据包内容的汇总信息会被显示在一个树形控件中，而所有的原始数据都同时显示在一个富文本框中。这样，用户一方面能够得到清晰易理解的概要信息，另一方面仍然可以控制每个需要进行模糊测试的字节。我们尝试将两个窗口绑定在一起，当树形控件中的一行被选中时，对应的原始数据会被显示为红色。图 16.2 展示了抓取了数据包之后的 ProtoFuzz 工具的界面。

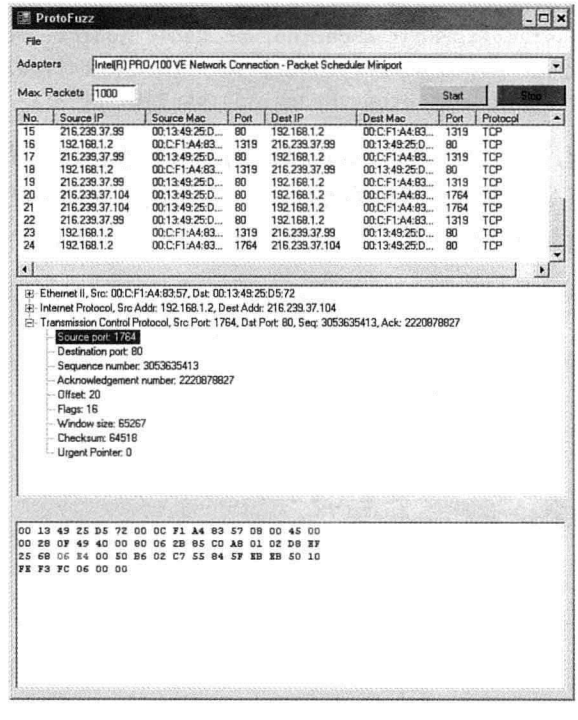


图 16.2　ProtoFuzz 抓取到的一个数据包

数据包的汇总视图显示在树形控件面板的中部。packetTvwDecode() 函数包含了创建树形控件的代码，同时，该函数还包含了多个独立的代码段用于解析以下协议的协议头：Ethernet，TCP，UDP，IP，ARP 和 ICMP。下面的代码段是一个简单的示例，展示了解析 Ethernet 头的代码片段。

```
Ethernet802_3 ethernet = new Ethernet802_3(capPacket);

srtSourceMacAddress = ethernet.SourceMACAddress.ToString();
strDestMacAddress = ethernet.DestinationMACAddress.ToString();
strEthernet = "Ethernet II, Src: " + strSourceMacAddress +
", Dst: " + strDestMacAddress;
strSrcMac = "Source: " + strSourceMacAddress;
strDstMac = "Destination: " + strDestMacAddress;
strEthernetType = "Type: " + ethernet.NetworkProtocol.ToString();
strData = "Data: " + ethernet.Data.ToString();

TreeNode nodeEthernet = tvwDecode.Nodes.Add(strEthernet);
TreeNode nodeEthernetDstMac = nodeEthernet.Nodes.Add(strDstMac);
TreeNode nodeEthernetSrcMac = nodeEthernet.Nodes.Add(strSrcMac);
TreeNode nodeType = nodeEthernet.Nodes.Add(strEthernetType);
TreeNode nodeData = nodeEthernet.Nodes.Add(strData);
```

261

从代码可以看到，我们使用来自 capPacket 的字节数组创建了类 Ethernet802_3 的一个实例，树状结构每个节点的显示内容是类的一个属性。下面的代码段是一个基本的，以网格模式显示原始数据包的循环。该循环在每行上打印 16 个字节。

```
static string PrintData(byte [] packet)
{
  string sData = null;

  int nPosition = 0, nColumns = 16;
  for (int i = 0; i< packet.Length; i++)
  {
    if (nPosition >= nColumns)
    {
      nPosition = 1;
      sData += "\n";
    }
    else
      nPosition++;

    byte nByte = (byte) packet.GetValue(i);
```

```
        if (nByte < 16)
          sData += "0";

        sData += nByte.ToString("X", oCulture.NumberFormat) + "";
      }
      sData += "\n";
      return (sData);
    }
```

5. 模糊变量

使用方括号（[]）括起来的字节表示强制模糊测试的模糊变量，使用尖括号（<>）括起来的字节表示基于字符串模糊测试的模糊变量。当向目标应用发送数据包时，ProtoFuzz 的代码简单地读取原始数据包并查找闭合的方括号和尖括号。当代码找到模糊变量后，变量中的原始数据就会被替换成模糊测试数据后再发送给目标应用。

6. 十六进制编码和解码

最后一个我们关心的内容是用十六进制方式对抓取得到的数据包内容进行编码(用于显示)，以及对以十六进制方式表示的字符串进行解码。.NET 框架提供了 ToString() 函数用于将一个字节数组转换成十六进制方式表示的字符串，但却没有提供方法将十六进制方式表示的字符串转换成字节数组[11]，由于这个原因，我们在项目中加入了 HexEncoding 类，该类的大部分代码是从 www.codeproject.com 网站上的“在 C# 中实现十六进制字符串和字节数组的双向转换”一文中借用过来的。

16.4　案例研究

现在，我们已经创建了一个网络模糊测试器，是时候检验我们的工作成果了。ProtoFuzz 是一个“单发”（每次发送一个数据包）工具，它拿到一个数据包，对其重复变异，然后将变异后的数据包发送给目标应用。这种方式不适用于某些条件。例如，如果我们需要首先向目标应用发送一系列初始数据包，使得目标应用位于漏洞状态，再发送导致目标应用发生崩溃的数据包，这种方式就不适用。以 SMTP 服务器中的一个缓冲区溢出漏洞为例，该缓冲区溢出漏洞发生在 RCPT TO 命令中。如下所示：

```
220 smtp.example.com ESMTP
HELO mail.heaven.org
```

[11] http://www.codeproject.com/csharp/hexencoding.asp

```
250 smtp.example.com Hello smtp.example.com
MAIL FROM:god@heaven.org
250 2.1.0 god@heaven.org… Sender ok
RCPT TO: [Ax1000]
```

在这个例子中，我们需要向 SMTP 服务器发送相当多请求。首先，我们需要建立 TCP 连接，然后，针对粗体字标识的每条命令（HELO，FROM，RCPT TO），我们需要向目标应用发送一个请求。为了让服务器到达准备好接受 RCPT TO 命令的状态，这些初始化请求必不可少，在这种情况下，RCPT TO 命令带的长字符串参数将最终导致溢出。类似这样的漏洞更适合使用 SPIKE 这样的工具来发现，SPIKE 这类工具允许用户创建定义数据包结构的脚本，并允许向目标应用发送多个请求。

263

ProtoFuzz 工具更适用于这样的场合：需要从传输数据中抓取一个数据包，选择数据包中的特定部分，然后立即开始进行模糊测试。如果你需要测试一个私有协议，而又没有足够的信息来创建一个能够描述协议结构的脚本，ProtoFuzz 工具就很适合。或者，你只是想要进行一次快速的测试，不想花时间为模糊测试器创建协议的描述，ProtoFuzz 也适合这种情形。

TippingPoint 的“零时差项目”（Zero Day Initiative，ZDI）发现了 HP 公司 Mercury LoadRunner 工具中的一个栈溢出漏洞，该漏洞符合 ProtoFuzz 的应用场景[12]。这个漏洞导致的溢出发生在一个代理程序中，该代理程序绑定在 TCP 端口 54345 上并监听进入的连接。该代理无须任何形式的认证就能接受传入的数据包，因此这就是我们寻找的“单发”漏洞。另外，该代理程序使用的是私有协议，而安全报告中并没有提供充分的细节让我们可以从头开始创建一个能利用该漏洞的验证性工具。安全报告唯一告诉我们的是，该溢出是被一个超长的 server_ip_name 字段值触发的。因此，我们将使用 ProtoFuzz 工具来抓取合法的数据包，找到 server_ip_name 字段，然后对数据包的这个字段进行变异。

在安装了 Mercury LoadRunner 并进行适当的配置之后，就可以开始寻找可能包含漏洞的事务了。当嗅探网络数据包时，我们发现了下面这个数据包，它同时包含了二进制和可读的 ASCII 文本，显然，其中包含有我们寻找的 server_ip_name 字段。

```
0070 2b 5b b6 00 00 05 b2 00   00 00 07 00 00 00 12 6d   +[..... .......m
0080 65 72 63 75 72 79 32 3b   31 33 30 34 3b 31 33 30   ercury2; 1304;130
0090 30 00 00 00 00 05 88 28   2d 73 65 72 76 65 72 5f   0......( -server_
00a0 69 70 5f 6e 61 6d 65 3d                             ip_name=
```

[12] http://www.zerodayinitiative.com/advisories/ZDI-07-007.html

基于我们抓取到的这个数据包，接下来，我们在 ProtoFuzz 工具中选中 server_ip_name 字段的值，点击鼠标右键，从弹出菜单中选择“Fuzz”-“Strings”。这个操作会在被选中值的两边加上尖括号（<>）。当 ProtoFuzz 工具处理这个数据包时，它会用 Strings.txt 文件中提供的每个值替换尖括号里的字节。要触发前面提到的这个栈溢出漏洞，我们需要向 Strings.txt 文件中添加超长的字符串。Strings.txt 文件中的每一行都会被包含在一个模糊测试数据包中发送给目标应用。

当模糊测试数据包被发送到目标应用后，接下来，重要的事情就是通过监视发现异常。Magentproc.exe 是绑定到 TCP 端口 54345 的应用，因此我们在这个进程上附加上调试器，并开始进行模糊测试。一旦模糊测试数据触发了应用崩溃，我们就能看到如下所示的 OllyDbg 调试器输出：

```
Registers
EAX 00000000
ECX 41414141
EDX 00C20658
EBX 00E263BC
ESP 00DCE7F0
EBP 41414141
ESI 00E2549C
EDI 00661221 two_way_.00661221
EIP 41414141

Stack
00DCE7F0 00000000
00DCE7F4 41414141
00DCE7F8 41414141
00DCE7FC 41414141
00DCE800 41414141
00DCE804 41414141
00DCE808 41414141
00DCE80C 41414141
00DCE810 41414141
00DCE814 41414141
```

类似上一章对 NetMail 的案例研究，这是一个清晰的可被远程利用的栈溢出的例子。无须认证就能使用服务这一事实，显然提高了 LoadRunner 这个流行的质量保证测试工具中出现漏洞的风险。

16.5　优势与可改进空间

目前，ProtoFuzz 工具不能处理在数据内容前附带有数据大小的数据块。因为在这

种情况下，对数据内容进行模糊测试需要在修改数据内容的同时修改数据块中数据大小的值，以保持该数据块仍然合法。向 ProtoFuzz 工具添加这个功能能够大大扩展 ProtoFuzz 工具，并且也很易于实现。

ProtoFuzz 工具目前没有提供探针（Probing）功能来监测模糊测试过程中目标主机是否仍然可用。在本章的前面部分我们提到，可以在发送模糊测试数据包后向目标应用发送探测数据包，监测目标应用的健康状况。如果目标不能响应，可能就需要中止模糊测试，因为当目标应用不能响应时，后续的测试用例只会是聋子的耳朵——摆设。此外，探针功能还能够帮助最终用户确定导致目标应用异常行为的模糊测试数据。另外，扩展 ProtoFuzz 工具使其能够处理复杂的请求-响应交互也能带来好处。例如，通过扩展 ProtoFuzz 工具，ProtoFuzz 工具除了能够发送模糊测试数据包之外，还能在开始进行模糊测试前，发送特定的有效数据包将被测应用置为某种特定状态，等待合适的响应，甚至还能够解析和解释数据包的内容。

265

我们也可以通过设计一个运行在目标机器上的，能够监视目标应用状态的远程代理来检测故障。这个代理程序应该能够与 ProtoFuzz 工具进行通信，当检测到故障时中止模糊测试。这种方式能够帮助识别哪个数据包直接导致了故障。

16.6 小结

要成为值得信赖的模糊测试工具，ProtoFuzz 工具还有很长的路要走。然而，ProtoFuzz 工具演示了一个基础的，易于被扩展到特定目标应用的网络协议模糊测试框架。创建这个网络模糊测试工具的一个关键因素是，找到一个能够抓取、解析并传输裸数据包的库。除非你在创建一个面向特定网络协议的模糊测试器，不需要修改数据包头，否则，不要采用那些不允许组装裸数据包的库。根据我们的预期目标，Metro 库为 ProtoFuzz 工具提供了强大的基础，而.NET 框架允许我们建立一个直观的 GUI 前端。我们鼓励你基于 ProtoFuzz 建立自己的模糊测试工具，并将其应用到不同的方向上。

第17章

Web浏览器的模糊测试

"One of the common denominators I have found is that expectations rise above that which is expected."

——George W. Bush, Los Angeles, September 27, 2000

由于攻击者大量利用客户端漏洞实施钓鱼攻击、身份窃取，以及创建僵尸网络，因此客户端漏洞一直是被关注的焦点。Web 浏览器中存在的漏洞为这类攻击提供了丰富的目标，存在于流行浏览器中的问题可能导致预料之外的大量犯罪。通常，客户端攻击需要使用某种形式的社交工程（social engineering），攻击者使用垃圾邮件或是利用流行的 Web 站点中的漏洞，强制潜在的受害者访问有害的 Web 页面。这种社交工程的攻击方式，加上互联网用户通常都只有少得可怜的计算机安全知识，因此，许多攻击者将关注点转移到客户端漏洞上来就不足为奇了。

从许多方面来说，2006 年都是大众开始关注浏览器缺陷的一年。这一年中，包括微软的 Internet Explorer，Mozilla 的 Firefox 在内的所有主流浏览器中都发现了漏洞。这些漏洞存在于浏览器的各个部分：解析不正常的 HTML 标签的部分；JavaScript 引擎；处理本地图片文件（如 JPG，GIF 和 PNG）的部分；解析和处理 CSS 的部分；以及 ActiveX 控件中。总共发现了数十个影响微软 Windows 操作系统和第三方软件的严重的 ActiveX 漏洞，以及上百个非关键的 ActiveX 漏洞。虽然安全研究人员过去也对 ActiveX 和 COM 审计发生过兴趣，但在 2006 年，对 ActiveX 控件的兴趣明显达到了高峰。导致安全研究人员对 ActiveX 控件漏洞关注的一个重要原因是出现了新的，用户友好的 ActiveX 审计工具。在本章中，我们将讨论如何用模糊测试发现 Web 浏览器中的漏洞。根据发现

浏览器漏洞的历史状况，我们预计 Web 浏览器中还将有大量缺陷被发现。

17.1 什么是 Web 浏览器模糊测试

Web 浏览器最初仅被设计为访问 Web，解析 HTML 页面的工具，但随着浏览器的发展，目前 Web 浏览器已经变成了瑞士军刀一样的多功能工具。现代的浏览器能够处理动态 HTML、样式表、支持执行多种脚本语言，支持 Java 小应用，支持 RSS 源，支持 FTP 连接，等等。大量的扩展和插件把 Web 浏览器变成了非常复杂的应用，甚至，Web 浏览器复杂到 Google 这样的公司能够把一些常用的桌面应用发布到互联网上。然而，对终端用户来说，不幸的是，浏览器的功能越多，被攻击的风险也就越大。随着越来越多的功能被整合到 Web 浏览器中，我们毫不惊讶地发现，Web 浏览器中发现了越来越多的漏洞。

浏览器缺陷月

2006 年 7 月， H. D. Moore 宣告了“浏览器缺陷月”[1]（Month of Browser Bugs，MoBB）。整个 7 月份，H. D.每天发布一个新的基于浏览器的漏洞，到 7 月结束，H.D. 共公布了 31 个漏洞的细节。虽然这 31 个漏洞中的大多数仅影响微软的 Internet Explorer，但所有主流浏览器，包括 Safari，Mozilla，Opera 和 Konqurer 都未能幸免[2]。这些漏洞中的大多数是由空指针引用之类的错误引起的，因此只会导致客户端 DoS，使得 Web 浏览器崩溃。虽然这些漏洞的严重性不高，但就在 MoBB 之前不久，Skywing 和 skape（与 H.D. Moore 同为 Metasploit 项目的贡献者）公开了 Internet Explorer 中未处理异常过滤器链（chainning of unhandled exception filters）中的一个设计漏洞，该漏洞可以导致特定条件下的完全代码执行[3]。这就增加了在 MoBB 中发布的一些漏洞的潜在风险。虽然有些人不同意在补丁出现之前公布漏洞的细节，但这种披露显然让人看到了 Web 浏览器漏洞的流行，以及它们带来的风险。

269

17.2 目标

每当有人发布“Firefox 比 Internet Explorer 更安全”的研究结果后不久，就会有新

[1] http://browserfun.blogspot.com/

[2] http://osvdb.org/blog/?p=127

[3] http://www.uninformed.org/?v=4&a=5&t=sumry

的，同样带有倾向性的报告给出完全相反的结论。当然，从研究者的角度来看，到底谁更安全是一个尚有争论的问题。实际上，所有的 Web 浏览器都存在严重漏洞，而模糊测试则是发现漏洞的一种有效方法。对软件来说，软件应用得越普遍，受到漏洞影响的用户就会越多。虽然浏览器市场占有率的统计结果时刻在变化，但无论从何种数据来源来看，Internet Explorer 总是雄踞榜首，而 Firefox 位列第二。显然，Internet Explorer 的流行得益于它被默认包含在全世界大多数 PC 机使用的操作系统中。尽管如此，Firefox 的势头继续对 Internet Explorer 构成压力。其他 Web 浏览器如 Netscape，Opera，Safari，以及 Konqueror，则只有一小部分用户，远远地落在后面。

17.3　方法

在对 Web 浏览器进行模糊测试时，需要做两个决定。首先，我们需要决定一种执行模糊测试的方法，然后，我们需要决定把注意力放在 Web 浏览器的哪个部分。考虑到 Web 浏览器的功能在不断增加，因此，达成完全的代码覆盖是一个艰巨的任务，更合理的期望应该是综合使用多种工具和方法来进行较为全面的审计。

17.3.1　测试方法

有一些基本方法可用于对 Web 浏览器进行模糊测试，具体使用哪种方法取决于我们想要对 Web 浏览器的哪些部分进行测试。一般而言，可以使用下面这些方法。

- **刷新 HTML 页面**：在这种情况下，我们使用 Web 服务器生成模糊测试需要使用的 Web 页面，这些生成的页面上带有让页面定时刷新的代码。有多种方式可以让页面定时刷新，例如，HTTP-EQUIV META 标记或是客户端脚本。当页面刷新时，服务器会向浏览器发送一个新的包含了模糊测试内容的页面。举例来说，Mangleme[4]是一个用于发现 Web 浏览器中的 HTML 解析漏洞的工具，该工具生成的用于模糊测试的每个 Web 页面的 HEAD 部分都包含以下内容：

```
<META HTTP-EQUIV=\"Refresh\" content=\"0;URL=mangle.cgi\">
```

该 META 标签导致 Magleme 生成的页面立即刷新并将浏览器重定向到 mangle.cgi，而 mangle.cgi 则指向下一个需要被模糊测试的页面。下面的客户端 JavaScript 脚本能达

[4] http://freshmeat.net/projects/mangleme/

成同样的目标，该脚本强制浏览器每隔 2000 毫秒刷新一次页面。

```
<HTML>
    <HEAD>
    <SCRIPT LANGUAGE="JavaScript">
        <!--
            var time = null
            function move() {
                window.location = http://localhost/fuzz
            }
        //-->
    </SCRIPT>
    </HEAD>
    <BODY ONLOAD="timer=setTimeout('move()', 2000)">
        [Page Content]
    </BODY>
</HTML>
```

- **加载页面**：可以通过让客户端直接加载模糊测试页面来完成模糊测试。使用命令行选项，可以让目标 Web 浏览器直接打开一个存在于本机上的模糊测试 Web 页面。例如，下面的命令会使用 Internet Explorer 直接打开文件 fuzz.html：

```
C:\>"C:\Program Files\Internet Explorer\iexplore.exe" C:\temp\fuzz.html
```

271

- **面向单独的浏览器对象**：特定情况下，在根本不启动浏览器的情况下也有可能进行基于浏览器的模糊测试。如果我们的测试目标是一个浏览器对象，而浏览器本身只是访问对象的媒介，在这种情况下，无须启动浏览器也能进行模糊测试。如果一个 ActiveX 控件被标记为“脚本安全”，该对象就能够被一个恶意的 Web 页面远程访问到。如果我们要对一个 ActiveX 控件进行模糊测试，无须启动浏览器就能测试该控件是否存在漏洞。David Zimmer 设计的 COMRaider[5]工具就使用这种方法直接测试 ActiveX 控件。

17.3.2 测试输入

与其他测试对象一样，对 Web 浏览器进行模糊测试时，仍然需要我们识别不同类型的用户输入。再次重申，决定什么是测试输入时必须走出我们的思维定势。对 Web 浏览器来说，攻击者是向 Web 浏览器提供内容的 Web 服务器。因此，Web 服务器发送

[5] http://labs.idefense.com/software/fuzzing.php#more_COMRaider

给浏览器的所有内容都应该被看作是输入，不应该仅仅把 HTML 代码当成输入。

1. HTML 头

浏览器收到的来自 Web 服务器的数据的第一部分是 HTML 头。虽然 HTML 头不会被浏览器直接显示出来，但 HTML 头中包含了能够影响 Web 浏览器行为，以及决定 Web 浏览器如何显示后续内容的重要信息。

CVE-2003-0113[6]是一个存在于 HTML 头中的漏洞，微软安全公告 MS03-015[7]中包含了该漏洞的信息。Jouko Pynnonen 发现这个漏洞是由于 urlmon.dll 没有正确地对 Content-type 和 Content-encoding 头[8]中的超长值进行边界检查导致的，当传入超长值时，会发生基于栈的缓冲区溢出。Internet Explorer 5.x 和 6.x 的多个版本均会受到这种类型攻击的影响。

2. HTML 标签

被发送给浏览器的页面内容由一系列 HTML 标签组成。从技术上来说，浏览器能够显示各种不同类型的标记语言。例如，HTML，XML 和 xHTML，但是出于我们的目的，我们将仅考虑由简单的 HTML 组成的 HTML 页面。

HTML 的功能一直在扩展，它正在变成一个越来越复杂的标准。此外，各种不同浏览器使用了许多私有标签，以及 Web 浏览器需要处理多种标记语言，因此，在设计 Web 浏览器时会有许多产生异常的可能。而这些编程错误通常可以被模糊测试发现。

HTML 的结构相对简单。有一些 HTML 标准确定了 Web 页面应该遵循的标准，一个具有正确格式的 Web 页面应该从文档类型声明（DTD，Document Type Declaration）开始。例如，下面的 DTD 告诉浏览器该页面遵循 HTML 4.01 标准：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
http://www.w3.org/TR/html4/strict.dtd>
```

当考虑 HTML 时，我们通常会想到定义文档内容的标签。这些标签被称为“元素”（elements），通常由起始标签和结束标签组成。除标签本身外，元素还可以有属性和内容。下面的 HTML 元素是一个字体元素：

---

[6] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0113

[7] http://www,microsoft.com/technet/security/bulletin/MS03-015.mspx

[8] http://downloads.securityfocus.com/vulnerabilities/exploits/urlmon-ex.pl

```
<font color="red">Fuzz</font>
```

在上面这个元素中，color 指明了该 font 元素的属性，而 Fuzz 这个词则是该元素的内容。虽然这个例子非常基础，但需要指出的重点是，HTML 代码中的所有部分都是模糊测试的对象。Web 浏览器被设计来解析和执行 HTML 代码，如果代码以非预期格式的形式存在，开发者可能并没有为 HTML 中的异常片段提供正确的错误处理。

Mangleme 工具通过让 Web 浏览器执行不正常的 HTML 标签来发现漏洞。通过这种方式，研究者发现了微软安全公告 MS04-040[9]中提到 CVE-2004-1050[10]漏洞。Ned 和 Berend-Jan Wever 使用 Mangleme 发现在 IFRAME，FRAME 和 EMBED 元素中使用超长的 SRC 或是 NAME 属性值会导致堆上的缓冲区溢出。他们随后公开发布了利用该漏洞的代码[11]。另一些用来模糊测试 Web 浏览器中的 HTML 结构的工具包括 H.D. Moore 和 Aviv Raff 设计的 DOM-Hanoi[12]和 Hamachi[13]工具。

273

3．XML 标签

XML 是一门继承自标准通用标记语言（SGML，Standardized General Markup Language）的通用标记语言，常用在通过 HTTP 协议传输的数据上。基于 XML 标准，人们已经定义了上千种数据格式：RSS、应用漏洞描述语言（AVDL，Application Vulnerability Description Language）、可扩展向量图形（SVG，Scalable Vector Graphics）等等。同对 HTML 的处理一样，现代浏览器能够解析和执行 XML 元素，以及其属性与内容。因此，与浏览器处理 HTML 时一样，如果浏览器在执行 XML 元素时遇到了非预期的数据，则可能导致一个未处理的异常。

向量标记语言（VML，Vector Markup Language）是一种用于定义向量图像的 XML 规范，在 Internet Explorer 和 Outlook 使用的库中已经发现了漏洞，而漏洞正是由于该库不能处理特定的，不正常的 VML 标签导致的。2006 年 9 月 16 日，微软发布了安全建议 925568[14]，在建议中详细描述了向量图像渲染引擎（vgx.dll）的栈中发生的缓冲区溢出能够导致远程代码执行。该安全建议发布后，出现了大量对该漏洞的利用，迫使微

---

9 http://www.microsoft.com/technet/security/bulletin/ms04-040.mspx

10 http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-1050

11 http://downloads.securityfocus.com/vulnerabilities/exploits/InternetExploiter.txt

12 http://metasploit.com/users/hdm/tools./domhanoi/domhanoi.html

13 http://metasploit.com/users/hdm/tools/hamachi/hamachi.html

14 http://www.microsoft.com/technet/security/advisory/925568.mspx

软在发布安全建议后 7 天就发布了一个计划外的补丁[15]。存在于同一个库中的另一个整数溢出漏洞则在稍后被发现，并于 2007 年 1 月打上了补丁[16]。

4．ActiveX 控件

ActiveX 控件是一种基于微软 COM 技术的私有技术，目的是提供可复用的软件组件[17]。Web 开发者通常在自己的应用中嵌入 ActiveX 控件，使得开发出来的 Web 应用能够具有一些典型 Web 应用不具有的扩展功能。使用 ActiveX 控件的一个主要缺点是，AtiveX 技术的私有特性限制了使用该技术的 Web 应用只能要求其用户使用 Internet Explorer 浏览器和 Windows 操作系统。尽管如此，ActiveX 技术仍然被广泛应用于许多网站。ActiveX 控件具有强大的功能，而且，一旦控件被标记为信任后，它们就具有完全访问操作系统的权限，因此，除非 ActiveX 控件的来源可信任，否则，绝不应该允许它们运行。

274

在 Windows 操作系统自带的和第三方软件中包含的 ActiveX 控件中发现过不少漏洞。如果这些有漏洞的控件被标识为“初始化安全”（safe for initialization）和“脚本安全”（safe for scripting），远程 Web 服务器就能够访问它们，因此攻击者就可以利用它们来攻击目标机器[18]。这造成了一个危险的情形：用户可能会从受信任的来源安装了一个 ActiveX 控件，但如果后来发现该 ActiveX 控件中存在诸如缓冲区溢出之类的漏洞，恶意 Web 站点就能利用这个被标记为“受信任”的控件。

COMRaider 是一个出色的向导驱动（wizard-driven）的模糊测试工具，可用于在目标系统上找出存在潜在漏洞的 ActiveX 控件。此外，COMRaider 工具能够在对 ActiveX 控件进行模糊测试时进行过滤，仅对那些能够被恶意 Web 服务器访问到的控件进行测试，而正是这些能够被恶意 Web 服务器访问到的 ActiveX 控件极大地增加了安全风险。COMRaider 可以帮助对给定系统上的所有 ActiveX 控件执行完全的审计，它能够成功地对用户选择的所有控件进行模糊测试。COMRaider 还包含了一个内建调试器，因此可以识别已被处理和未被处理的异常。另外，COMRaider 还包含了分布式审计功能，允许一个团队分共享以前的审计结果。图 17.1 显示了 COMRaider 对某个系统上的所有标记为脚本安全的 ActiveX 控件进行了完全审计之后得到的结果列表。

275

---

15 http://www.microsoft.com/technet/security/Bulletin/MS06-055.mspx

16 http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=462

17 http://en.wikipedia.org/wiki/ActiveX_Control

18 http://msdn.microsoft.com/workshop/components/activex/safety.asp

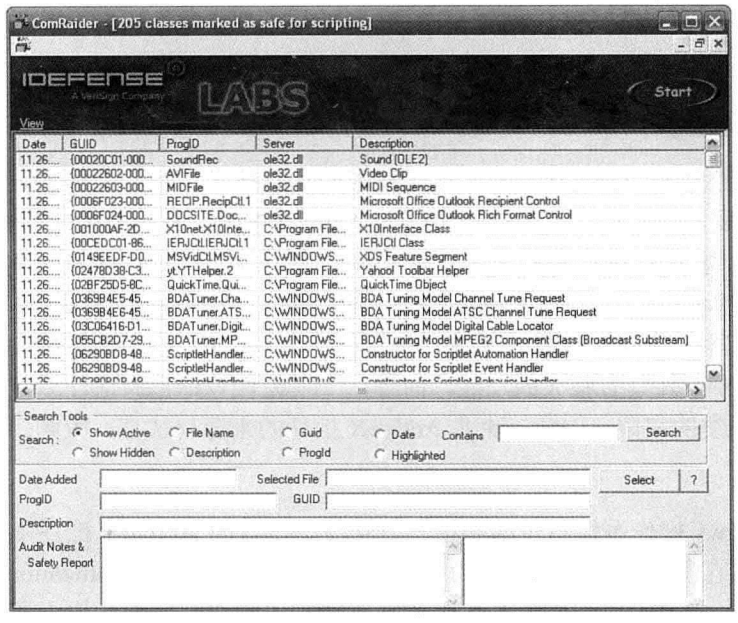


图 17.1 ComRaider

AxMan[19]是另一个可免费获得的 ActiveX 模糊测试器。AxMan 由 H. D. Moore 设计，该工具发现了“浏览器缺陷之月”中公布的 ActiveX 漏洞[20]中的大多数。

5. 层叠样式表（Cascading Style Sheets）

CSSDIE[21]是由 H. D. Moore，Matt Murphy，Aviv Raff 和 Thierry Zoller 共同创建的一个 CSS 模糊测试器，该工具发现了“浏览器缺陷之月”中公布的“Opera 浏览器中存在的内存破坏[22]”问题。在 Opera 中，当将一个 DHTML 元素的 CSS 背景属性设置为超长的 URL 时，就会引发客户端 DoS，并导致浏览器崩溃。

还有其他一系列漏洞同样是基于 CSS 的缺陷导致的。例如描述在 MS06-21[23]中的 CVE-2005-4089 缺陷，该缺陷允许攻击者跳过 Internet Explorer 中的跨域限制。该缺陷之

---

19 http://metasploit.com/users/hdm/tools/axman/

20 http://browserfun.blogspot.com/2006/08/axman-activex-fuzzer.html

21 http://metaspoilt.com/users/hdm/tools/see-ess-ess-die/cssdie.html

22 http://browserfun..blogspot.com/2006/07/mobb-26-opera-css-background.html

23 http://www.microsoft.com/technet/security/Bulletin/MS06-021.mspx

所以发生，是由于网页可以使用 CSS 中的@import 直接从其他域下载文件。来自 hack.co.il 的 Matan Gillon 演示了如何利用该漏洞查看到其他人的 Google 桌面搜索（Google Desktop Search，GDS）的结果[24]。虽然在 Google 修改了 Google 桌面搜索的实现方式后该攻击方法不再生效，但这是一个很好的例子，说明了简单的 Internet Explorer 漏洞能够与其他应用（在这个例子中是 Google 桌面搜索）的功能一起发生作用，导致复杂的攻击。

在 CSS 中，一个特定元素的格式被定义在一个花括号中，其中属性和值使用冒号隔开。例如，一个锚（anchor）标签可以被定义成 { color : white} 。当在页面中使用@import 导入非 CSS 文件时，Internet Explorer 会尝试将该文件当作 CSS 文件执行，该文件中的任何一个左花括号后的所有后续内容都会被取为 cssText 的属性值。Matan 利用这种攻击方法就能窃取到被诱使访问某特定页面的用户的 Google 桌面搜索的唯一键值。具体的方法是这样的：将带有 " }{ " 查询的 "Google 新闻" 的页面作为 CSS 文件导入。这种方法能够导致 Google 新闻的查询结果被包含在 cssText 属性中，攻击者随后就能够从 cssText 属性中获得该用户的 Google 桌面搜索的唯一键值。这样，攻击者就可以使用@import 直接从用户本地的 Google 桌面搜索中导入结果，这次就能获得本地的 Google 桌面搜索索引。

276

6. 客户端脚本

Web 浏览器能够使用多种客户端脚本语言创建 Web 页面上的动态内容。虽然 JavaScript 是应用得最普遍的脚本语言，但除了 Javascript 外，其他脚本语言，如 VBScript，Jscript[25]以及 ECMAScript[26]等也同样得到应用。脚本语言的世界是一个 "世袭家族"，许多脚本语言都是从其他脚本语言中派生出来的。例如，ECMAScript 就是 JavaScript 的一个标准化版本，而 Jscript 也只是一个微软版本的 JavaScript 实现而已。这些脚本语言为 Web 站点提供了强大的能力，但同时也导致了一些安全性问题。

攻击者通常使用脚本语言操作其他存在漏洞的组件，如 ActiveX 控件，而不是直接利用脚本引擎的漏洞。然而，这并不意味着脚本引擎中不存在漏洞。脚本引擎中经常出现内存破坏问题，例如使用 Javascript 在一个本地函数上迭代导致 Internet Explorer 中的空引用[27]这个漏洞就是个典型的脚本引擎的漏洞。在 "浏览器缺陷之月" 中，H. D. Moore

---

[24] http://www,hacker.co.il/security/ie/css_import.html

[25] http://en.wikipedia.org/wiki/Jscript

[26] http://en.wikipedia.oeg/wiki/Ecmascript

[27] http://browserfun.blogspot.com/2006/07/mobb-25-native-function-iterator.html

通过下面这行简单的代码演示了该漏洞：

```
for ( var i in window.alert) { var a = 1; }
```

Azafran 发现了 Firefox 的 JavaScript 引擎中的一个漏洞，该漏洞允许一个恶意 Web 站点远程获取任意堆内存的内容[28]，导致了信息泄漏问题。这个漏洞是由接受 lambda 表达式的 replace() 函数引发的。虽然攻击者无法利用这个漏洞修改堆内存的内容，但攻击者有可能能够在获取到的内存中找到口令之类的敏感信息。

277

堆填充（HEAP FILLS）

通常情况下，客户端脚本语言不会被攻击者当成是一个攻击手段，相反，攻击者更喜欢利用一个单独的,存在漏洞的浏览器组件发起攻击。对浏览器漏洞的利用，从内存破坏到摇摆（dangling）指针，通常都需要借助客户端脚本语言的帮助。JavaScript 可以在多个场合下帮助利用漏洞，并经常在基于堆的溢出中被利用。由于 JavaScript 的广泛应用与其出色的跨平台特性，显然它会是漏洞利用方面的好选择。假设你已经发现了一个缺陷，该缺陷能够导致发生一系列在你控制之下的解引用（dereferences）。假定在下面的例子中你能够完全控制 EAX 寄存器：

```
MOV EAX, [EAX]

CALL [EAX+4]
```

大多数时候，EAX 的值都会使这两条指令产生一个访问违例，并导致浏览器崩溃。要成功地重定向控制流，我们为 EAX 提供的地址必须是一个合法的指针，该指针指向另一个指向合法代码的指针。很难通过简单的方法在常规进程空间中找到符合我们要求的 EAX 的值。但是，我们可以使用 JavaScript 成功地分配大块堆数据，并操作内存空间使其对攻击者更友好。因此，攻击者通常使用客户端 JavaScript 向堆中填充 NOP 指令和攻击代码来增加命中攻击代码的概率。在汇编代码中 NOP 表示“无操作”，也就是“什么都不干”。当执行到 NOP 指令时，程序会继续在 NOP 指令中迭代，不会改变任何寄存器或是其他内存位置的值。Berend-Jan Wever（也就是 SkyLined）在他的 Internet Exploiter 工具[29]的代码中发明了一个流行的堆溢出技术，使用 0x0D 来对堆进行溢出。选择用来产生堆溢出的值很关键，这个值有双重含义。首先，它代表一个 5 字节的类似 NOP 的指令 OR EAX, 0D0D0D0D0D。其次，它能够成为一个指向自身的（self-referential）

[28] http://www.mozilla.org/security/announce/2005/msfa2005-33.html

[29] http://www.milw0rm.com/exploits/930

指针，指向一个合法的内存地址。它被解释的方式依赖于它指向的内容。

考虑显示图 17.2 所示的来自 OllyDbg Heap Vis[30]插件的屏幕截图[31]。该截图演示了该插件如何能在基于堆填充的利用中被用来探索和帮助可视化 Internet Explorer 内存状态。



图 17.2　OllyDbg Heap Vis 帮助发现 Internet Explorer 中的 SkyLined 堆填充

图 17.2 的右上角标题为“Heap Vis”的窗口区域显示了已分配的堆块的列表。高亮显示的条目是一个大小约为 500Kb 的从地址 0x0D0A0020 开始的块。该块包含了地址 0x0D0D0D0D。截图左上角的标题为“Heap Block – 0D0A0020..0D12101F”的窗口显示了被高亮显示的块中 0x0D 字符串结束到某内嵌的攻击代码开始部分的内容。紧接着我们上面的两行代码的解引用例子，假设我们将 EAX 的值设置为 0x0D0D0D0D。第一个

[30] http://www.openrce.org/downloads/details/1

[31] http://pedram.openrce.org/images/olly_heap_vis/skylined_ie_heap_fill.gif

解引用 MOV EAX, [EAX]导致 EAX 保持与目前地址中一致的值，都是一系列的 0x0D。下一条指令 CALL [EAX+4]导致控制权转移到了地址 0x0D0D0D11，而该地址处是一个长序列的 0x0D 的中间。这个序列被处理器当作 NOP 等价指令，直到到达截图中所示的攻击代码指令为止。之所以选择值 0x0D0D0D0D 来产生堆溢出，是因为堆通常从低地址向高地址进行填充。选择以 0 开始的地址就意味着到达目标地址前需要更少时间来填充堆，而使用一个较大的值如 0x44444444 填充堆将会导致花费较长时间才能到达攻击代码，这样就会增加在成功利用漏洞前用户强制退出浏览器的概率。

要进一步说明这个问题，考虑下面两个“坏”选择：0x01010101，该十六进制串可以被翻译成 ADD [ECX], EAX，以及 0x0A0A0A0A，该十六进制串可以被翻译成 OR CL, [EDX] 。前一条指令可能会因为寄存器 ECX 中包含了非法的写地址而失败，而后一条指令则可能由于寄存器 EDX 中包含了一个非法的读地址而失败。另一个“好”的例子是 0x05050505，该十六进制串可以被翻译成 ADD EAX, 0x05050505。一些公开发布的攻击利用中使用了这个序列。

7. Flash

尽管 Adobe Flash Player 选择作为第三方插件，而不是直接增强 Web 浏览器功能的方式对浏览器进行扩展，但到目前为止，Flash 已经得到了广泛应用。如今大多数 Web 浏览器上都安装了某个版本的 Flash Player。二进制的 Flash 文件通常带有.swf 扩展名，能够被作为独立文件打开，但 Flash 文件更经常地以 Web 页面中的内部对象形式存在，被浏览器下载并在 Flash Player 中打开。由于 Flash 文件是二进制形式的，因此可以采用两种方式来进行模糊测试。对一个已知的合法的.swf 文件，我们可以使用第 11 章到第 13 章中介绍的文件格式模糊测试的方法来对其进行模糊测试。2005 年 11 月，eEye 公布了 Macromedia Flash 版本 6 和版本 7 中的一个内存访问漏洞的细节。虽然不能确定该漏洞是如何被发现的，但模糊测试应该是可能发现该问题的一种方法。此外，可以使用 ActionScript 编写 Flash 文件，ActionScript 是一种 Flash 使用的脚本语言，用于在运行时操作数据和内容[32]。可以使用模糊测试在 ActionScript 代码编译之前对不同的 ActionScript 方法产生变异。2006 年 10 月，Rapid7 发布了一个安全建议，详细描述了如何使用 XML.addRequestHeader() 方法为 Flash 对象产生的请求添加任意的 HTTP 头[33]。通过 HTTP 请求分割技术，利用这种方法可以执行任意 HTTP 请求。

[32] http://en.wikipedia.org/wiki/Actionscript

[33] http://www.rapid7.com/advisories/R7-0026.jsp

8. URLs

有时候 URL 自身能够导致漏洞。在 MS06-042[34]发布后，eEye 发现针对这个公告的最新的补丁同样引入了一个堆溢出漏洞。他们发现当向 Internet Explorer 传入超长的 URL 时，如果目标 Web 站点指定了使用 GZIP 或是 deflate 编码，调用 lstrcpynA() 尝试复制一个长达 2084 字节的 URL 到一个 260 字节的缓冲区时就会产生一个溢出[35]。如果用户使用的是存在漏洞的 Internet Explorer，只要用户点击一个 URL 链接就会导致利用该漏洞的攻击成为可能。

17.4　漏洞

虽然攻击者或多或少都需要通过社会工程才能利用客户端漏洞实施成功的攻击，但由于客户端漏洞可能导致多种类型的攻击，因此它们仍然带来了显著的风险。

- **DoS**：许多 Web 浏览器漏洞只会触发客户端 DoS，进而导致浏览器崩溃或是变得无响应。这类漏洞通常是由无尽循环或内存破坏等不能被攻击者进一步利用的状况导致的。从重要性上来说，客户端 DoS 攻击非常次要。虽然每次 Web 浏览器崩溃后，重启浏览器这件事情很恼人，但它不会导致任何持久的损害。与服务端 DoS 攻击不同，客户端 DoS 攻击只会造成一次性的损害。
- **缓冲区溢出**：缓冲区溢出是非常普遍的 Web 浏览器漏洞，它们能够被以前提到过的几乎所有输入向量所触发。由于缓冲区溢出漏洞能够导致代码执行，因此尤其危险。
- **远程命令执行**：远程命令执行漏洞通常利用已存在的功能，该功能并没有被设计为允许远程代码执行，但却由于功能的缺陷导致允许远程代码执行。例如，Albert Puigsech Galicia 发现攻击者能够将 FTP 命令直接注入到 FTP URI 中，这意味着只要诱使 Internet Explorer 6.x 或其更早版本的用户点击一个链接就能让用户的浏览器执行 FTP 命令[36]。该漏洞能够用来向用户的计算机下载文件。进一步的研究发现，利用这个漏洞可以导致浏览器发送 email 信息。微软在 MS02-042 中描述了该漏洞。
- **旁路跨域限制**（Cross-domain restriction bypass）：Web 浏览器禁止一个 Web

281

[34] http://www.microsoft.com/technet/security/bulletin/ms06-042.mspx

[35] http://research.eeye.com/html/advisories/published/AD20060824.html

[36] http://osvdb.org/displayvuln.php?osvdb_id=12299

站点获取其他网站上的内容。这个限制非常重要，要是没有这个限制的话，任何网站都能从其他网站上获取到 cookie 之类的信息，而 cookie 信息通常包含有 session ID 等重要信息。许多现有的漏洞允许站点打破这个域限制。上文提到的 Google 桌面搜索中存在的漏洞就是这类问题的一个例子。

- **旁路安全区域（Bypassing security zones）**：Internet Explorer 根据内容来源的区域实现安全性。来源于 Internet 的文档通常被认为是不受信任的，因此存在着严格限制。与之相反，打开的本地文件则被认为是可信的，并被授予了许多特权。2005 年 2 月，Jouko Pynnonen 公布了一个安全问题，描述特殊编码的 URL 如何骗过 Internet Explorer，使其将远程文件当作打开的本地文件来处理[37]。这种漏洞允许攻击者将恶意脚本包含在下载文件中，能够执行一个攻击者提供的利用漏洞的代码。微软在安全公告 MS05-14[38]中描述了这个问题。
- **地址栏欺骗（Address bar spoofing）**：随着试图从毫无戒心的 Web 用户那里获取信用卡号之类的私人信息的犯罪的不断增加，钓鱼已经成为了一个严重的问题。虽然有些钓鱼攻击仅利用了社会手段，但也有一些复杂的钓鱼攻击利用了 Web 浏览器漏洞使得钓鱼网站看上去像是个合法的网站。地址栏欺骗漏洞对钓鱼者来说很有价值，因为这个漏洞允许一个伪造的 Web 页面看起来好像是位于合法站点上的。不幸的是，在所有主流 Web 浏览器中都存在一些这类漏洞。

282

17.5 检测

当对 Web 浏览器进行模糊测试时，同样需要在识别不明显的错误方面下工夫。不能只从单一的地方寻找错误，相反，需要探索一些不同的错误来源。

- **事件日志（Event logs）**：如果在 Windows 环境下对一个浏览器进行模糊测试，不要忽视了事件浏览器（Event Viewer）的价值。Internet Explorer 7.0 向事件浏览器中添加了一个单独的 Internet Explorer 日志组。如果你测试的是早期版本的 Internet Explorer 或是其他类型的浏览器，事件日志的条目应该包含在应用日志中。虽然日志条目远称不上全面，但日志是一个易于检查的来源，并可能包含了有用的数据。
- **性能监视器（Performance monitors）**：内存破坏问题或是无尽循环可能导致目

[37] http://jouko.iki.fi/adv/zonespoof.html

[38] http://www.microsoft.com/technet/security/bulletin/ms05-014.mspx

标 Web 浏览器的性能下降。性能监视工具可以帮助识别这些情况。然而，要利用性能监视器识别 Web 浏览器的性能下降，就需要确保在模糊测试时使用专用机器，保证没有其他因素导致性能下降。

- **调试器（Debuggers）**：目前为止，最有用的 Web 浏览器模糊测试的检测工具就是附加到目标浏览器上的第三方调试器。调试器允许你识别被处理和未被处理的异常，并能够帮助检测出内存破坏问题是否可能被利用。

17.6　小结

虽然客户端漏洞曾一度被忽视，但随着钓鱼攻击的增长，我们不得不关注它们带来的风险。你所在公司网络上的一个存在漏洞的浏览器可能会成为攻击者的网关。客户端攻击或多或少需要一些社会工程，但这不是一个主要障碍。Web 浏览器漏洞为攻击者提供了便于进行攻击的条件。在本章中，我们引入了不少已存在的 Web 浏览器模糊测试工具。接下来，我们将会运用我们在这里学到的东西来构建我们自己的面向 Web 浏览器的模糊测试器。

第18章

Web浏览器的自动化模糊测试

"Natural gas is hemispheric. I like to call it hemispheric in nature because it is a product that we can find in our neiborhoods."

——George W. Bush, Washington, DC, December 20, 2000

在第17章我们讨论了Web浏览器及其“可模糊测试性”。越来越多的人对浏览器模糊测试感兴趣，这导致产生了许多模糊测试工具，同时也导致发现了大量影响流行浏览器（如Firefox和Internet Explorer）中的漏洞。在本章中，我们将研究如何构建ActiveX模糊测试器。虽然对ActiveX漏洞的利用仅限于Internet Explorer，而且已有一些ActiveX模糊测试器，但由于ActiveX模糊测试这一主题有趣而复杂，所以我们想要进一步讨论这个测试方向。由于微软的浏览器仍然占据主要的Web使用者市场，因此，我们将讨论主题局限在Internet Explorer上也不算完全令人失望。本章从简要介绍ActiveX技术的历史开始，然后深入探讨开发一个ActiveX模糊测试工具。

18.1 组件对象模型背景

微软的COM技术诞生于20世纪90年代早期，该技术雄心勃勃地想要为软件之间的交互提供通用的交互协议。按照COM技术的设想，一旦客户端-服务器之间的通信能够被标准化，则任何支持COM的编程语言都能用来开发能够彼此交换数据的软件，无论需要交换数据的软件位于本地，位于同一个系统，还是位于两个不同的远程系统中。COM如今已被广泛应用，有过一段辉煌的时期，并在发展过程中产生了大量缩略词（同

时也导致了混淆）。

18.1.1　COM 简史

COM 最早的祖先可以追溯到动态数据交换（Dynamic Data Exchange，DDE），Windows 操作系统直到今天仍然在使用该技术。你可以在 Shell 文件扩展、剪贴板查看器（使用了 NetDDE）以及微软的红心大战游戏（也使用了 NetDDE）中找到 DDE 应用的痕迹。1991 年，微软发布了对象链接与嵌入（Object Linking and Embedding，OLE）技术。DDE 仅能用于纯粹的数据交换，OLE 却可以在一种文档类型中嵌入另一种文档类型。OLE 客户端-服务器之间的互通信发生在系统库内，通过虚函数表实现（Virtual Function Tables，VTBLs）。

COM 技术出现于 OLE 技术之后，紧接着 COM 技术的发布，微软又发布了 OLE 2。OLE 2 是新一代的 OLE，建立在 COM 技术而不是虚函数表技术上。1996 年，OLE 2 被重新命名为 ActiveX。随后，同年微软发布了 COM 技术的扩展：分布式 COM（DCOM），作为对通用对象请求代理体系结构架构[1]（Common Object Request Broker Architecture，CORBA）的回应。DCOM 背后的 RPC 机制是分布式计算环境/远程过程调用[2]（Distributed Computing Environment/Remote Procedure Call，DCE/RPC）。DCOM 进一步扩展了 COM 的灵活性，因为它允许软件开发者通过互联网这样的媒介提供功能，而无须让用户能够访问到实际的代码。

COM 历史上最近的事件是 COM+的引入，COM+随着 Windows 2000 操作系统发布而发布。COM+ 提供的额外的好处是“组件农场”（component farms），该“组件农场”由绑定在 Windows 2000 中的微软事务服务器（Microsoft Transaction Server）管理。与 DCOM 一样，COM+组件也可以是分布式的，并允许无须从内存卸载的情况下重用之。

18.1.2　对象与接口

COM 体系结构定义了对象和接口。一个对象，例如一个 ActiveX 控件，通过定义和实现一个接口来描述其功能。因此，软件就可以查询一个 COM 对象来发现该 COM 对象暴露了哪些接口和功能。每个 COM 对象都会被分配一个唯一的称为类 ID（class ID，CLSID）

285

[1] http://en.wikipedia.org/wiki/Corba

[2] http://en.wikipedia.org/wiki/DCE/RPC

的 128 位标志符。此外，每个 COM 接口也会被分配一个唯一的，128 位，称为接口 ID（Interface ID，IID）的标志符。COM 接口的例子包括 IStream，IDispatch 和 IObjectSafety。这些接口和所有其他接口都是从一个名为 IUnknown 的基础接口派生得到。

除了 CLSID 外，对象还可以指定一个 program ID（ProgID）作为标识。ProgID 一般是一个人工可读的字符串，比 CLSID 更方便用来引用一个对象。例如，考虑下面的 CLSID 和 ProgID：

- 000208D5-0000-0000-C000-000000000046
- Excel.Application

以上两个 ID 是同一个 ActiveX 定义在注册表中的 CLSID 和 ProgID，可以互相替换。然而，要注意，ProgID 不保证唯一。

18.1.3 ActiveX

ActiveX 技术是与我们当前讨论的主题“Web 浏览器模糊测试”唯一契合的 COM 技术。与 Java 小应用（Java applets）类似，ActiveX 控件能访问操作系统内更多的内容，因此可以用来开发大量扩展，扩充浏览器本身所不具备的功能。ActiveX 组件的应用很普遍，许多软件包中都包含 ActiveX 组件，同时，还有些 ActiveX 组件会直接发布在 Web 上。依赖 ActiveX 技术的产品和服务的例子包括在线病毒扫描器、网络会议和网络即时通信工具、在线游戏等等。

微软 Internet Explorer 是唯一天然支持 ActiveX，并实现了标准的文档对象模型（Document Object Model，DOM）来处理实例化，参数和方法调用的浏览器。下面的例子用于示例，供读者参考。

纯粹的 DOM：

```
<objectclassid    =    "clsid:F08DF954-8592-11D1-B16A-00C0F0283628"
        id        =    "Slider1"
        width     =    "100"
        height    =    "50">
    <param name="BorderStyle" value="1"    />
    <param name="MousePointer"value="0"    />
    <param name="Enabled"     value="1"    />
    <param name="Min"         value="0"    />
    <param name="Max"         value="10"   />
</object>
```

```
Slider1.method(arg, arg, arg)
```

过时的内嵌方式：

```
<embed type  =   "application/x-oleobject"
        name =   "foo"
        align    =   "baseline"
        border   =   "0"
        width    =   "200"
        height   =   "300"
        clsid    =   "{8E27C92B-1264-101C-8A2F-040224009C02">

foo.method(arg, arg, arg)
```

JavaScript / Jscript：

```
<script type="javascript">
    functioncall_function()
    {
        obj = new ActiveXObject('AcroPDF.PDF');
        obj.property = "value";
        obj.method(arg, arg, arg);
    }

    call_function();
</script>
```

Visual Basic：

```
<objectclassid    = 'clsid:38EE5CEE-4862-11D3-854F-00A0C9C898E7"
        id        = "target">
</object>

<scriptlanguge="vbscript">
    'Wscript.echotypename(target)
    'Sub SelectAndActivateButton( ByVallButton As Long )
    arg1=2147483647
    target.property = arg1
    target.method arg1
</script>
```

除了将控件直接加载到浏览器之外，ActiveX 控件还能够被当成标准 COM 对象加载并直接对外提供接口。对我们的模糊测试而言，直接加载控件的方法更为方便，因为

这种方式消除了生成和加载浏览器代码到 Internet Explorer 中的中间步骤。

微软COM编程及其内部机制是一个很大的话题，有好些书整本都在描述这个话题。如果想要获得进一步的 COM 技术的信息，读者可以访问微软 COM 技术的 Web 站点[3]以获得高层次的全局信息，MSDN 文章“组件对象模型：技术预览”[4]则给出了低层次的预览信息。

287

在下一节中，我们将直接深入开发一个 ActiveX 模糊测试器，沿着这个方向进一步阐述 COM 技术的相关细节。

18.2 开发模糊测试器

有许多编程语言都适合用来开发 ActiveX 模糊测试器。证据是目前已经存在了用多种编程语言实现的 ActiveX 模糊测试器。COMRaider[5]主要建立在 Visual Basic 的基础上，同时使用了一些 C++。AxMan[6]是用 C++，JavaScript 和 HTML 混合开发的。以上两种模糊测试器使用相同的基本方法进行模糊测试，因此，我们采用同样的方法来为我们的模糊测试器建模：

- 枚举所有可被加载的 ActiveX 控件。
- 枚举每个 ActiveX 控件可被访问的方法和属性。
- 解析类型库，决定方法的参数类型。
- 生成模糊测试的测试用例。
- 执行模糊测试用例，同时监视异常情况。

在编程语言选择方面，与前面两种工具的选择不同，这次我们完全使用 Python 这门编程语言实现这个模糊测试工具。我们选择 Python 语言也许会让你觉得奇怪，甚至会觉得这个选择完全行不通。然而，使用 Python 开发我们需要的模糊测试器的确是可能的，因为现代的 Python 发行版已经带有相当多的实现了底层 Windows API 的模块。我们依赖的，用于与 COM 组件进行通信的 Python 模块包括 win32api，win32com，pythoncom 和 win32con 等（如果你想了解更多 Windows 环境下的 Python 编程的细节，

---

[3] http://www.microsoft.com/com/default.mspx

[4] http://msdn2.microsoft.com/en-us/library/ms809980.aspx

[5] http://labs.idefense.com/software/fuzzing.php#more_comraider

[6] http://metasploit.com/users/hdm/tools/axman

请参考 Mark Hammond 所著的“Python Programming on Win32”[7]一书，Mark 开发了许多我们可以使用的，让 Python 可以和 COM 进行通信的组件）。图 18.1 和 18.2 均来自于 PythonWin 的 COM 与类型库浏览器，展示了从高层到低层的 COM 信息。



图 18.1　PythonWin COM 浏览器

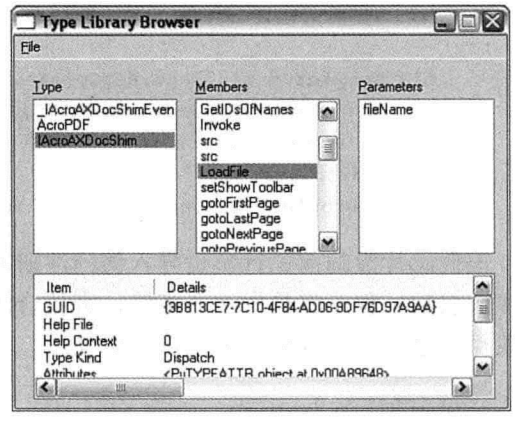


图 18.2　PythonWin 类型库浏览器

下面的代码是一个示例，该程序片段启动微软 Excel 并设置 `Visible` 属性让窗口可见。

```
import win32com.client
xl = win32com.client.Dispatch("Excel.Application")
xl.Visible = 1
```

接下来我们将研究如何枚举所有可被加载的 ActiveX 控件，并直接进入更多例程中。本章剩余部分展示的代码片段都来自可以从本书的官网（http://www.fuzzing.org）下载得到的具有完整功能的 COM 模糊测试器。

18.2.1　枚举可被加载的 AcriveX 控件

我们的第一个任务是枚举目标系统上所有可用的 COM 对象。COM 对象的完整列表存在于 Windows 注册表[8]的 HKEY_LOCAL_MACHINE（HKLM）及其子键

[7] http://www.oreilly.com/catalog/pythonwin32/

[8] http://en.wikipedia.org/wiki/Windows_registry

SOFTWARE\Classes 中。可以使用标准的 Windows 注册表访问 API 来访问这些键值[9]。

```
import win32api, win32con
importpythoncom, win32com.client
from win32com.axscript import axscript

try:
    classes_ket = win32api.RegOpenKey( \
        win32con.HKEY_LOCAL_MACHINE,   \
        "SOFTWARE\\Classes")
except win32api.error:
    print"Problem opening key HKLM\\SOFTWARE\\Classes
```

该代码片段的前三行负责导入访问注册表及与 COM 对象交互所必须的库。成功打开注册表后，程序将枚举注册表键，查找合法的类 CLSID。如果找到了一个 CLSID，该条目就会被保存到列表中，供稍后使用。

```
skey_index = 0
clsid_list = []

while True:
    try:
        skey = win32api.RegEnumKey(classes_key, skey_index)
    except win32api.error
        print "End of keys"
        break

progid = skey



try:
    skey = win32api.RegOpenKey(win32con.HKEY_LOCAL_MACHINE,  \
        "SOFTWARE\\Classes\\%s\\CLSID" % progid)
except win32api.error:
    print "Couldn't get CLSID key...skipping"
    skey_index += 1
    continue

try:
```

290

[9] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/registry_functions.asp

```
        clsid = win32api.RegQueryValueEx(skey, None)[0]
    except win32api.error:
        print 'Couldn't get CLSID value...skipping"
        skey_index += 1
        continue

    clsid_list.append(progid, clsid)
    skey_index += 1
```

以上的代码片段生成了一个元组（tuple）列表，列表列出了系统中所有可用的 COM 对象。需要进行安全性测试的 COM 对象通常限制在可以通过 Internet Explorer 访问到的控件。到目前为止，大多数流行的利用 ActiveX 漏洞的技术都是由攻击者建立一个恶意 Web 站点，诱使用户使用有漏洞的组件访问这些页面。因此，只有可以通过 Internet Explorer 访问到的 ActiveX 控件中的漏洞才可能被利用。由于只有部分控件能够通过 Internet Explorer 被访问，所以我们的第二个任务就是从我们的列表中去掉这些不能通过 Internet Explorer 访问的控件。如果下面的三个要素中的任何一个被满足的话，Internet Explorer 将会在不给出警告信息的情况下加载 ActiveX 控件[10]：

- 控件在 Windows 注册表中被标识为“脚本安全”。
- 控件在 Windows 注册表中被标识为“初始化安全”。
- 控件实现了 IObjectSafety COM 接口。

Windows 注册表包含了组件类别键，列出了已安装的组件实现的每一个类别的子键。我们查找的两个子键是 CATID_SafeForScripting 和 CATID_SafeForInitializing。下面的例程会检测一个给定的 CLSID 是否在注册表中被标识为 Internet Explorer 中可用：

```
def is_safe_for_scripting (clsid):
    try:
        key = win32api.RegOpenKey(win32con.HKEY_CLASSES_ROOT, \
            "CLSID\\%s\Implemented Categories" % clsid)
    except win32api.error:
        return False

    skey_index = 0
    while True:
        try:
            skey = win32api.RegEnumKey(key, skey_index)
```

291

10

```
            except:
                break

            # CATID_SafeForScripting
            if key == "{7DD95801-9882-11CF-9FA9-00AA006C42C4}":
                return True

            skey_index += 1

        return False

    def is_safe_for_init (clsid):
        try:
            key = win32api.RegOpenKey(win32con.HKEY_CLASSES_ROOT, \
                "CLSID\\%s\\Implemented Categories" % clsid)
        except win32api.error:
            return False;

        skey_index = 0
        while True:
            try:
                skey = win32api.RegEnumKey(key, skey_index)
            except:
                break

            #CATID_SafeForInitializing
            if key == "{7DD95802-9882-11CF-9F9A-00AA006C42C4}":
                return True

            skey_index += 1

        return False
```

292 除了在注册表中进行标记外，ActiveX 控件还能够通过实现 IObjectSafety 接口将自己标识成对 Internet Explorer 安全。要知道一个给定的 ActiveX 控件是否实现了 IObjectSafety 接口，必须将其实例化并进行查询。下面的例程能检测到一个给定的控件是否实现了 IObjectSafety 接口，以及是否可以被从 Internet Explorer 中访问。

```
    def is_iobject_safety (clsid):
        try:
            unknown = pythoncom.CoCreateInstance(clsid, \
                None,
                pythoncom.CLSCTX_INPROC_SERVER,
```

```
            pythoncom.IID_IUnknown)
    except:
        return False

    try:
        objsafe = unknown.QueryInterface(axscript.IID_IObjectSafety)
    except:
        return False

    return True
```

要确定最终版本的可加载 ActiveX 控件的列表，还必须考虑最后一件事。微软提供了一个机制销毁位（killing bitting）[11]来防止 Internet Explorer 加载某些 CLSID。该机制通过注册表键 HKLM\Software\Microsoft\Internet Explorer\ActiveX Compatibility\<AcriveX 控件的 CLSID> 实现。这个注册表位置包含的每一个 CLSID 条目都必须从我们的列表中剔除出去。下面的函数能够检测一个给定的 CLSID 是否被以这种形式禁用。

```
def is_kill_bitted (clsid):
    try:
        key = win32api.RegOpenKey(win32con.HKEY_LOCAL_MACHINE, \
            "SOFTWARE\\Microsoft\\Internet Explorer"                    \
            "\\ActiveX Compatibility\\%s" % clsid)
    except win32api.error:
        return False

    try:
        (compat_flags, type) = win32api.RegQueryValueEx(key,           \
            "Compatibility Flags")
    except:
        return False

    if typ != win32con.REG_DWORD:
        return False

    if compat_flags& 0x400:
        return True
    else:
        return False
```

293

[11] http://support.microsoft.com/kb/240797

```
        return False
```

以上的几段代码创建 ActiveX 控件列表，执行两次检查，确定哪些控件可能存在问题。接下来我们需要检查控件导出的所有属性和方法。

18.2.2 属性、方法、参数与类型

系统化地生成目标控件列表的能力很方便，我们的 ActiveX 模糊测试器正是从这个功能开始发挥作用。COM 对象的所有属性和方法的描述都直接包含在我们得到的控件列表中，此外，属性和方法的参数类型也包含在列表中。COM 的这个特性让我们这些首次得到这种特性的模糊测试器开发者觉得振奋不已。通过程序枚举出一个 ActiveX 控件可能被攻击的接口，我们可以依赖这种能力创建智能模糊测试器，智能模糊测试器知道某个特定的方法期望什么类型的数据。这样，当方法期望一个整型参数时，我们不会浪费时间提供字符串作为输入。

在 COM 的世界里，数据通过一个名为 VARIANT 的结构传递。VARIANT 数据结构支持许多数据类型，包括整数、浮点数、字符串、日期类型、布尔类型、其他 COM 对象，以及以上这些数据类型的数组。PythonCOM 提供了一个抽象层，隐藏了许多细节。表 18.1 展示了一些 Python 内置的类型及其等价的 VARIANT 类型。

294

表 18.1 PythonCOM VARIANT 翻译

Python 对象类型	VARIANT 类型
Integer	VT_14
String	VT_BSTR
Float	VT_R8
None	VT_NIL
True/False	VT_BOOL

pythoncon 模块提供了 LoadTypeLib() 函数，该函数解析直接来自二进制 COM 对象的类型库。我们需要知道的 COM 对象的属性与方法的所有信息都可以从被加载的类型库中获得。作为示例，我们来研究图 18.2 所示的 Adobe Acrobat PDF 控件的类型库。该 ActiveX 控件绑定在 Adobe Acrobat Reader 中，被标记为脚本安全与初始化安全，因此可被 Internet Explorer 访问。下面的代码片段演示了如何加载类型库，并使用了一些 Python 技巧来创建一个 VARIANT 名称的映射。

```
adobe = r"C:\Program Files\Common Files" \
        r"\Adobe\Acrobat\ActiveX\AcroPDF.dll"
```

```
tlb = pythoncom.LoadTypeLib(adobe)

VTS = {}
for vt in [x for x in pythoncom.__dict__.keys() if x.count("VT_")]:
    VTS[eval("pythoncom.%s" % vt)] = vt
```

我们后面将会看到，以上代码生成的 VARIANT 名称映射只能用于演示目的。一个类型库可以定义多个类型，调用 GetTypeInfoCount() 函数能够获取到需要循环处理的次数。在我们的例子中，图 18.2 的第一列显示了三种不同的类型。下面的代码展示了如何循环获取这些不同的类型并打印出它们的名字：

```
for pos in xrange(tlb.GetTypeInfoCount()):
    name = tlb.GetDocumentation(pos)[0]
    print name
```

Acrobat 控件定义了三种类型。我们仔细研究图 18.2 中高亮显示的类型 IAcroAXDocShim。与大多数编程中的计数方式一样，位置计数从 0 开始，这意味着我们期望的类型数量的索引是 2 而不是 3。在下面的代码块中，我们从定义好的类型库中取得类型信息和属性，然后枚举特定类型下的属性。

```
info = tlb.GetTypeInfo(2)
attr = info.GetTypeAttr()

print "properties:"
fori in xrange(attr.cVars):
    id       = info.GetVarDesc(i)[0]
    names    = info.GetNames(id)
    print "\t", names[0]
```

cVars 属性变量指定了类型的属性（或变量）的数量。该计数用于决定循环次数，在循环中我们打印出每个属性的名字。枚举得到方法、参数、参数类型的方法非常简单，参见下面的代码：

```
print "methods:"
for i in xrange(attr.cFuncs):
    desc = info.GetFuncDesc(i)

    if desc.wFuncFlags:
        continue

    id       = desc.memid
```

```
names    = info.GetNames(id)
print "\t%s()" % names[0]

i = 0
for name in names[1:]:
    print "\t%s, %s" % (name, VTS[desc.args[i][0]])
    i += 1
```

在这个例子中，cFuncs 属性变量指定了类型下定义的方法的数量。除了设置了 wFuncFlags 的方法外，所有其他方法都被枚举了出来。wFuncFlags 标志说明方法是受限的（不可访问），因此不适合对其进行模糊测试。GetNames() 例程返回方法的名称，以及方法的每个参数的名称。代码随后打印出方法的名称，并遍历列表的剩余部分（name[1:]）来访问所有的方法参数。由 GetFuncDesc() 函数返回的函数描述包含了针对每个参数的 VARIANT 类型的列表。VARIANT 类型被表示为一个整数，我们可以通过索引前面生成的 VARIANT 映射将其转换成名字。

296

在 Adobe Acrobat PDF ActiveX 控件的 IAcoAXDocDhim 接口上执行以上这段脚本的最终结果如下：

```
properties:
methods:
    src()
    LoadFile()
        fileName, VT_BSTR
    setShowToolbar()
        On, VT_BOOL
    gotoFirstPage()
    gotoLastPage()
    gotoNextPage()
    gotoPreviousPage()
    setCurrentPage()
        n, VT_I4
    goForwardStack()
    goBackwardStack()
    setPageMode()
        pageMode, VT_BSTR
    setLayoutMode()
        layoutMode, VT_BSTR
    setNamedDest()
        namedDest, VT_BSTR
    Print()
    PrintWithDialog()
```

```
setZoom()
    percent, VT_R4
setZoomScroll()
    percent, VT_R4
    left, VT_R4
    top, VT_R4
setView()
    viewMode, VT_BSTR
setViewScroll()
    viewMode, VT_BSTR
    offset, VT_R4
setViewRect()
    left, VT_R4
    top, VT_R4
    width, VT_R4
    height, VT_R4
printPages()
    from, VT_I4
    to, VT_I4
printPagesFit()
    from, VT_I4
    to, VT_I4
    shrinkToFit, VT_BOOL
printAll()
printAllFit()
    shrinkToFit, VT_BOOL
setShowScrollBars()
    On, VT_BOOL
GetVersions()
setCurrentHeightlight()
    a, VT_I4
    b, VT_I4
    c, VT_I4
    d, VT_I4
setCurrentHighlight()
    a, VT_I4
    b, VT_I4
    c, VT_I4
    d, VT_I4
postMessage()
    strArray, VT_VARIANT
messageHandler()
```

297

从打印出的结果可以看到，这个类型中定义了 39 个方法（函数），没有定义属性。代码成功地枚举并显示出了每个方法的参数列表和类型列表。对开发智能模糊测试器而言，以上这些信息非常关键。生成测试用例时需要用到这些信息，以确保每个变量都被正确地模糊化。例如，考虑对一个短整数（VT_I2）和一个长整数（VT_I4）进行模糊测试。把这两个整数当作不同类型而不是同样的通用整数来处理显然可以节省测试时间，因为一个短整数的取值范围是从 0 到 0xFFFF（65,535），而一个长整数的取值范围是从 0 到 0xFFFFFFFF（4,294,967,295）。

到目前为止，我们已经探讨了枚举所有 Internet Explorer 可访问的 ActiveX 控件的属性、方法和参数的必要步骤。下一步要做的是选择合适的模糊测试启发式准则并开始测试。

298

18.2.3 模糊测试与监视

在第 6 章中，我们描述了选择智能字符串和整数模糊值的重要性。我们选择的模糊测试启发式准则设计为“最大化发生错误的可能性”。实际上，本书描述的大部分模糊测试器和测试用例被设计用来发现较低层次上的故障（例如缓冲区溢出）而不是逻辑问题（例如访问受限资源）。目录遍历修饰符（Directory traversal modifiers）是通过启发方式发现逻辑安全漏洞的例子。对 ActiveX 控件进行模糊测试时，查找不正常行为需要格外谨慎，因为安全研究者经常会发现完全不应该允许通过 Internet Explorer 访问的控件。

例如，考虑“WinZip FileView ActiveX 控件的不安全方法漏洞”[12]。在这个漏洞中，ProgID 为 WZFILEVIEW.FileViewCtrl.61 的 ActiveX 控件被发布并设置为“脚本安全”，这就使得恶意 Web 站点能够在浏览器中加载该控件并利用它对外提供的功能。特别是，该 ActiveX 控件中提供的 ExeCmdForAllSelected 和 ExeCmdForFolder 方法允许调用者复制、移动、删除和执行位于任意位置（包括网络共享、FTP 目录和 Web 目录）的文件。这就为攻击者提供了一个简便的手段，使其可以在毫无警告的情况下下载和执行任意可执行文件，而这一切只需要攻击者简单地架设一个恶意 Web 站点即可。WinZip 针对这个问题发布了一个补丁，移除了脚本安全设置并额外为特定控件设置了销毁位（kill bit）。

除了前面使用的典型的模糊测试启发式准则之外，模糊测试数据中也应该包括合法的文件路径、命令和 URL，以帮助发现与 WinZip FileView 漏洞类似的漏洞。当然，模糊测试监视器必须具有必要的能力来确定模糊测试器是否成功地访问到了所有提供的资源。我们将在后面对这个概念进行更详细的讨论。

---

[12] http://www.zerodayinitiative.com/advisories/ZDI-06-040.html

生成合适的模糊测试数据列表后，下一步需要创建一系列的测试用例。生成测试用例的方法之一是，用前面提到的任意方法构建一个嵌入了目标 ActiveX 控件的 HTML 页面，然后使用模糊化的参数调用目标方法。这种方法需要为每个测试生成一个单独文件，然后在 Internet Explorer 中加载各个测试用例。另一种方法则是直接加载目标控件并对其进行模糊测试。继续我们前面的例子，下面这段 Python 代码将实例化 Adobe Acrobat PDF ActiveX 控件，访问它的两个外部可访问的方法 GetVersions() 和 LoadFile()：

```
adobe = win32com.client.Dispatch("AcroPDF.PDF.1")
printadobe.GetVersions()
adobe.LoadFile("c:\\test.pdf")
```

上面这段代码的第一行负责创建一个 Adobe COM 对象的实例，该实例可通过变量名 adobe 访问。COM 对象被创建后，就能够非常自然地被通过接口来访问。代码的第二行打印出 GetVersions() 的调用结果，第三行使控件加载一个 PDF 文件到 Acrobat Reader viewer 中。扩展这个例子，对其他方法、参数和控件进行模糊测试是非常简单的事情。

问题的最后一部分是模糊测试器的监视部分。除 Python COM 模块外，我们还使用了 PaiMei[13]逆向工程库实现一个基于调试的模糊测试监视器。在后面的章节中，我们将深入讨论和扩展使用 PaiMei。基本上，这个基于调试的模糊测试监视器就是一个轻量的调试器，该调试器封装了目标 ActiveX 控件的执行。当发现一个低层次的漏洞（例如，缓冲区溢出）后，模糊测试器调用调试器，记录下错误细节。PaiMei 库还提供了钩住（hooking）API 调用的功能，该功能可用于监视异常行为。通过观察调用微软提供的库函数 CreateFile()[14] 和 CreateProcess()[15]时传入的参数，我们能够确定目标 ActiveX 控件是否成功地访问到了它不应该访问到的资源。查阅 API 钩子功能文档的工作留给读者作为练习。

18.3　小结

在本章中，我们探索了微软 COM 技术的历史，列出了能够从 Internet Explorer 中访问的 ActiveX 控件需要满足的条件。本章研究了枚举可访问的 ActiveX 控件的属性、方法、参数和参数类型，探索了达成这个目标需要的 Python COM 接口以及所需的特定步骤。本章给出的代码片段都来自一个具有完整功能的 COM 模糊测试器，可以从本书官网（http://www.fuzzing.org）上下载得到该模糊测试器。

---

[13] http://www.openrce.org/downloads/details/208/PaiMei

[14] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/createfile.asp

[15] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createprocess.asp

第19章

内存模糊测试

"It's white."

——George W. Bush, after being asked by a child in Britain what the White House was like, July 19, 2001

在本章中，我们将会介绍内存模糊测试（in-memory fuzzing）的概念。内存模糊测试是一个新的、尚未得到公众注意的模糊测试方法，目前甚至还没有公开发布的具有完整功能的，针对这类模糊测试的概念型工具。从根本上来说，这种技术的目标是将模糊测试从我们熟悉的客户端-服务器（或是仅客户端）模型转向完全面向内存目标的模型。虽然从技术上说后一种技术更复杂，需要使用者具有很强的底层知识，包括汇编语言、进程内存布局、进程的底层知识等，但这种技术能够带来多种好处。在深入该技术的具体实现细节之前，我们先看看该技术能够带来的好处。

在前面的章节中，我们尝试努力从 UNIX 和 Windows 两个平台无偏见地提供模糊测试视角。而在本章中，由于内存模糊测试技术过于复杂，我们只能将本章的关注点限制在单一平台上。出于几种原因，我们选择微软的 Windows 平台作为本章我们关注的平台。做出这一决定的第一个原因是，内存模糊测试技术在闭源目标上应用得更好，而 Windows 是一个普遍采用闭源二进制软件包发布软件的平台。其次，Windows 为进程检测提供了丰富且强大的调试 API。这并不是说 UNIX 平台没有提供调试 API，但我们认为 UNIX 平台上的 API 不够健壮。当然，虽然我们在本章中仅讨论 Windows 平台上的内存模糊测试，但我们在本章中讨论的通用理念和方法都可以被用于其他平台。

19.1　为什么需要内存模糊测试？怎么进行？

302

到目前为止，我们讨论的每种模糊测试方法都需要生成数据并通过预期的“通道”传输数据。在第 11 章、第 12 章和第 13 章中，我们让被测应用直接加载变异后的文件来进行模糊测试。在第 14 章、第 15 章和第 16 章中，我们通过网络套接字接口将模糊测试数据发送给目标服务。在这两种情况下，我们都需要向目标提供完整的文件或是协议，即使我们只对文件或协议中的一个特定字段感兴趣也是如此。另外，在这两种情况下，我们完全不清楚我们输入的数据究竟会触发哪些对应底层代码的执行。

内存模糊测试是一种完全不同的模糊测试方法。在使用内存模糊测试时，我们关注的不是特定协议或是特定的文件格式，而是实际执行的代码。内存模糊测试使得我们的注意力从数据输入转移到负责解析数据输入的函数和汇编指令上。使用这种方式，我们放弃了预期的数据通信通道，转而关注于在目标应用内存中变异或是生成数据。

什么情况下这种方法能带来好处？假设我们正在对某闭源的网络后台应用进行模糊测试，而该后台应用使用了复杂的包加密或数据混淆机制对数据进行保护。按照我们以前的做法，要想对这个目标进行成功的模糊测试，首先需要对这个应用进行逆向工程，并根据逆向工程结果重建数据混淆机制。内存模糊测试非常适合对这类应用进行测试。使用内存模糊测试的话，我们不需要关心协议的封装方式，而是进入目标应用的内存空间，查找我们感兴趣的点——例如，在接收到的网络数据被解混淆之后。

另一个适合应用内存模糊测试的例子是，我们希望测试一个特定功能的健壮性。假设通过逆向工程工作，我们已经盯上了一个解析邮件的例程，该例程接受一个字符串参数，并将该参数当做 E-mail 地址进行解析。我们可以通过生成和发送包含变异的 E-mail 地址的网络数据包或测试文件来对其进行模糊测试，但如果使用内存模糊测试方法，我们便能够直接对目标例程进行模糊测试。

我们将在本章的后续内容和第 20 章中进一步澄清这些概念。但在此之前，我们需要先了解一些相关背景知识。

19.2　必要的背景知识

在继续进一步讨论之前，让我们快速回顾一下微软 Windows 内存模型，并简要讨论一个典型 Windows 进程内存空间的布局和属性。在这一节中，我们将会触及一些复

303 杂和深入的主题。我们希望读者深入研究这些主题，但由于其中某些内容很枯燥，如果读者对此不感兴趣的话，可以选择跳过这些内容，直接前进到本节结尾的图表处。

从 Windows 95 开始，Windows 操作系统开始采用平滑内存模型（flat memory model），该内存模型在 32 位平台上提供了总共 4GB 的可寻址空间。默认情况下，这 4G 地址范围被分为两部分：靠近底部的一半空间（0x00000000 ～ 0x7FFFFFFF）保留给用户空间，而靠近顶部的一半空间（0x80000000 ～ 0xFFFFFFFF）保留给内核空间。可以通过编程将分配方式改成 3：1（在 boot.ini 中设置/3GB 开关[1]），这种设置保留 3GB 内存给用户空间，保留 1GB 给内核，以此提升 Oracle 数据库等对内存敏感应用的性能。

与分段内存模型相反，平滑内存模型通常使用单一内存段引用程序代码和数据。分段内存模型则利用多个段来引用不同的内存位置。与分段内存模型相比，平滑内存模型的主要优势包括极大的性能提升，以及降低了开发者的开发复杂性。在平滑内存模型中开发者无须选择段和在段间进行切换。为进一步简化开发者的工作，Windows 操作系统通过虚拟地址管理内存。简单地说，虚拟内存模型为每个运行中的进程提供了它自己的 4GB 虚拟内存空间。借助内存管理单元（Memory Management Unit，MMU）的辅助，Windows 操作系统能够完成虚拟内存到物理内存的转换。显然，大多数系统都不可能为每个运行中的进程真正提供 4GB 的物理内存。虚拟内存空间建立在内存分页的概念上。页（page）是一片连续的内存块，在 Windows 平台上大小为 4K（0x1000）字节。使用中的虚拟内存页被存储在 RAM（主存）中，未被使用的内存页会被交换到磁盘（辅存）上，并在需要的时候被加载进入 RAM。

Windows 内存管理的另一个重要概念是内存保护（memory protections）。内存保护属性作用的最小粒度是内存页，不可能将保护属性仅作用于内存页的一部分。我们关注的 Windows 系统可用的保护属性包括以下这些。[2]

- PAGE_EXECUTE：内存页可执行，对该页的读取和写入操作会导致访问违例。
- PAGE_EXECUTE_READ：内存页可执行和可读，对该页的写操作会导致访问违例。
- PAGE_EXECUTE_READWRITE：内存页具有完全的访问权限。该页可被读取和写入。
- 304 PAGE_NOACCESS：内存页不可访问。对该内存页的执行、读取或写入都会

[1] http://support.microsoft.com/kb/q291988/

[2] http://msdn2.microsoft.com/en-us/library/aa366786.aspx

导致访问违例。

- PAGE_READONLY：内存页仅可读。尝试向内存页写入会导致访问违例。如果底层架构支持读取和执行的差别化（这种情况不常见），任何尝试从该内存页中执行的操作也会导致访问违例。
- PAGE_READWRITE：内存页可读和可写。与 PAGE_READONLY 属性一样，如果底层架构支持读取和执行的差别化，任何尝试从该内存页上执行的操作会导致访问违例。

而对于修饰器（modifier），根据我们的目的，我们仅对 PAGE_GUARD 修饰器感兴趣，根据 MSDN 的描述[3]，当发生对设置了该修饰器的页的访问行为，系统就会抛出 STATUS_GUARD_PAGE_VIOLATION 异常，然后移除该修饰器。从根本上来说，PAGE_GUARD 扮演着一次性访问警告的角色。我们可以利用这一功能监视进程对内存中特定区域的访问。此外，使用 PAGE_NOACCESS 属性也可以达到相同的目的。

对内存模糊测试而言，熟悉内存页、内存布局、保护属性非常重要，在第 20 章我们将看到这一点。内存管理是一个复杂的主题，许多出版物对其进行了大量的讨论[4]。我们鼓励读者自行参考这些文献。但就我们所关心的内存模糊测试而言，读者只需要知道下面这些知识：

- 每个 Windows 进程能“看到”自己独有的 4GB 虚拟地址空间。
- 从 0x00000000 到 0x7FFFFFFF 的内存空间中，只有靠近底部的内存可被用做用户空间，靠近顶部的 2GB 空间是保留的内核空间。
- 一个进程的虚地址空间被隐式保护，其他进程无法访问。
- 4GB 地址空间被划分为大小为 4KB 的页。
- 内存保护属性作用的最小粒度是单个内存页。
- PAGE_GUARD 内存保护修饰器可被用做一次性的页访问警告。

305

研究图 19.1 能够更好地理解典型 Windows 进程的虚拟地址空间中特定元素所在的位置。对第 20 章我们讨论的内存模糊测试自动化工具的具体开发细节而言，图 19.1 也是一个有用的参考。在图 19.1 的下方，为方便对这些内容不熟悉的读者，我们简要描述了图中列出的各种元素。

---

[3] http://msdn2.microsoft.com/en-us/library/aa366786.aspx

[4] 《Windows 内幕》，第四版，Mark E. Eussinovich, David A. Solomen：《未公开的 Windows 2000 秘密：开发者手册》，Sven Schreiber；《未公开的 Windows NT》Prasad Dabak，Sandeep Phadke，Milind Borate

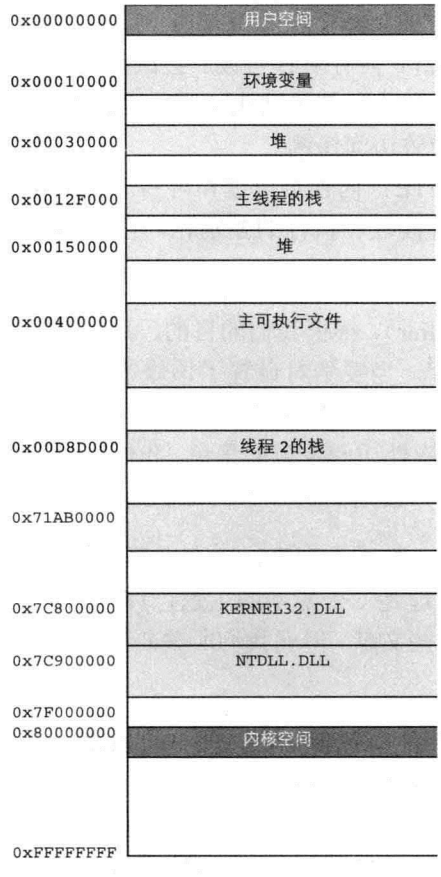


图 19.1　典型的 Windows 内存布局

306

为方便对此不熟悉的读者，我们简要描述图 19.1 中列出的各种元素。从地址范围的最低地址值开始向高地址值浏览，我们看到位于地址 0x00010000 处的是进程环境变量。在第 7 章中，我们提到过环境变量是系统级的全局变量，用于定义应用的特定行为。在更上方的位置我们能看到位于地址 0x00030000 和 0x00150000 处的两个堆。堆是内存池，通过 malloc() 函数和 HeapAlloc() 函数可以从堆中获得动态分配的内存。注意，如图 19.1 所示，任何给定的进程都能拥有不止一个堆。在两个堆之间的地址 0x0012F000 处，我们看到的是主线程的栈。栈是一种后入先出的数据结构，这里的栈用来追踪函数调用链和局部变量。注意进程中的每个线程都有它自己的栈。在我们这个示例中，线程 2 的栈位于地址 0x00D8D000 处。在地址 0x00400000 处我们看到主可执行文件（主可执行文件是个.exe 文件，通过执行该文件可以得到当前进程）被加载在这里。在用户地

址空间的上端我们可以看到一些系统 DLL：kenerl32.dll 和 ntdll.dll。DLL 是微软方式的共享库实现，是通过单一文件导出的，跨应用使用的公共代码。可执行文件与 DLL 都使用了可移植的执行（Portable Executable，PE）文件格式。注意，并非每个进程的内存布局都与图 19.1 中显示的一样，图中显示的地址值只是用作说明的一个示例而已。

我们再次触及了不少需要用专门书籍解释的复杂主题。我们鼓励读者参考更多深入的材料。然而，就我们的目的而言，我们应该已经掌握了足够的信息，可以深入目前的主题了。

19.3　究竟什么是内存模糊测试的简要解释

在本书中我们提到并实现了两种内存模糊测试方法。在这两个案例中，我们都将模糊测试器从目标外部移到了目标内部。图 19.2 的可视化方式能够帮助我们解释内存模糊测试，方便读者理解。图 19.2 描述了一个我们虚构的典型网络目标的简单控制流图。

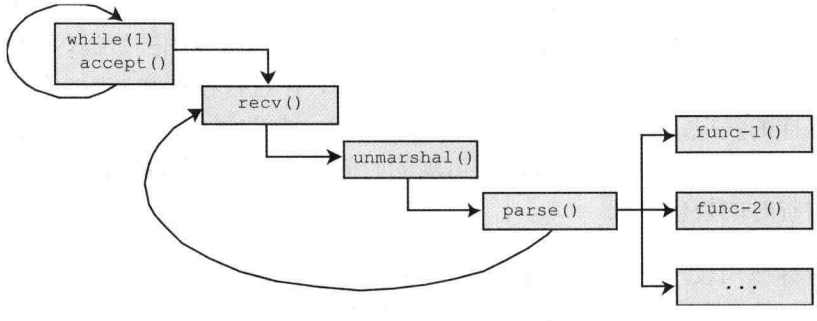


图 19.2　示例的控制流图

我们示例目标应用的主线程中有一个循环等待新客户端的连接。每接收到一个连接，进程就会创建一个新线程，该线程发起一个或多个 recv() 调用接收客户端请求。接收到的数据随后通过某种形式的数据反解或是数据处理例程被传递。unmarshal() 例程可能负责解压缩或是解密协议，但不会实际解析数据流中包含的字段。然后，处理过的数据被传递给主解析例程 parse()，该例程建立在对其他一些例程与库的调用之上。parse() 例程处理数据流中的不同字段，在循环回到从客户端接收进一步指令之前采取合适的请求动作。

在这种情况下，我们是否可能不再通过网络传送模糊测试数据，完全跳过网络传输？这种代码分离方式使我们可以直接工作在最有可能找到缺陷的例程（例如，解析例

程）上。接下来，我们可以向例程注入错误并监视修改导致的输出，这样就可以大大提高模糊测试过程的效率，因为所有操作都可以在内存中完成。通过使用某些形式的进程操作工具，我们“钩住”（hook）目标应用中位于解析例程前方的位置，修改例程接收的不同输入，让例程运行。显然，由于本书的主题是模糊测试，我们当然想要能够循环执行这些步骤，在每个迭代中自动地修改输入并监视其输出结果。

19.4 目标

要建立一个网络协议或文件格式模糊测试器，其中最耗时的需求是对协议和文件格式本身的研究。目前已有一些自动解析协议的尝试，这些尝试借鉴了生物信息学领域的知识。我们会在第 22 章中讨论这个主题。尽管如此，在大多数情况下，协议研究仍是一个依靠手工的乏味过程。

如果要处理的是 SMTP、POP 和 HTTP 等标准的文档化协议，有许多来源可供研究者参考，如 RFC、开源客户端的代码、甚至是以前写好的模糊测试器。与此同时，考虑到这些协议已经在不同应用中得到广泛使用，存在大量可复用的资源，也许基于这些在你的模糊测试器中实现这类协议是可行的。然而，在进行模糊测试时，你会经常发现自己面临这样的状况：目标协议缺乏好的描述文档，非常复杂，或者干脆是私有协议。在这些情况下，对协议的研究需要消耗可观的时间与资源，内存模糊测试也许是这种情况下可行的省时的替代方法。

308

如果目标应用使用的通信协议本身并不复杂，但应用了开放加密协议对其进行封装，怎么办？更糟的是，如果用于封装协议的是私有而不是公开的加密方法或是混淆方法呢？一度非常流行的 Skype 通信工具[5]就是一个这类难于被审计的例子。EADS/CRC 安全团队使用超长数据[6]在混淆层之上破坏了 Skype 并发现了一个关键的安全漏洞（SKYPE-SB/2005-003）[7]。如果目标应用暴露出了非加密接口，你也许能通过非加密接口发现这个问题，但如果目标应用没有提供非加密接口怎么办？加密过的协议是阻碍模糊测试的主要障碍。此外，内存模糊测试还可以使你免于对加密例程进行逆向工程的痛苦。

---

[5] http://www.skype.com

[6] http://www.ossir.org/windows/supports/2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf

[7] http://www.skype.com/security/skype-sb-2005-03.html

在使用内存模糊测试方法对你的目标应用进行模糊测试之前，你必须考虑许多因素，需要被考虑的首要且最显著的因素是，内存模糊测试需要对测试目标进行逆向工程，定位最佳的“钩子”（hook）位置，这样就可以减少进行模糊测试的开销。一般来说，内存模糊测试最适合应用于运行在你的内存模糊测试工具支持的平台上的闭源目标。

19.5　内存模糊测试方法之变异循环插入（Mutation Loop Insertion）

第一种内存模糊测试方法是变异循环插入（Mutation Loop Insertion，MLI）。变异循环插入方法需要我们首先通过逆向工程，人工定位 parse() 例程（解析例程）的开始和结束位置。一旦完成定位，我们的变异循环插入工具能够向目标应用的内存空间中插入一个 mutation() 例程（变异例程）。变异例程负责修改解析例程拿到的数据，我们将专门在第 20 章中描述几种实现变异例程的方法。接下来，我们的变异循环插入工具会向内存中插入两条无条件跳转指令，这两条指令分别是“从解析例程的结尾跳到变异例程的开始处”，以及“从变异例程的结尾跳到解析例程的开始处”。完成所有这些操作后，我们目标应用的控制流图如图 19.3 所示。

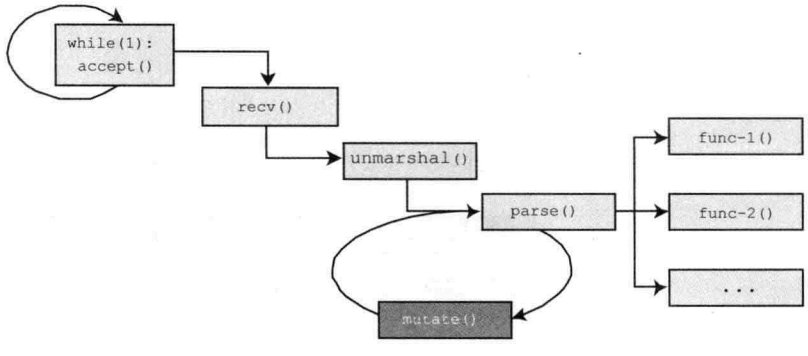


图 19.3　变异循环插入的可视化表示

读者可以看到，现在我们已经围绕解析代码和我们的目标应用创建了一个自给自足的数据变异循环。这样，我们就不再需要远程连接到我们的目标并发送数据包，自然也就能显著节省模糊测试需要的时间。数据变异循环的每一次迭代都会向 mutate() 对应的解析例程传入不同的、可能引发错误的数据。

当然，在上文中我们介绍的是一个极其简单的例子。虽然采用这个简单例子的目的是为了让读者能够对内存模糊测试有概要的理解，但需要指出的是，内存模糊测试是极其依赖经验的。我们将在第 20 章中构造一个功能性的模糊测试器，并对此进行更深入的探讨。

19.6 内存模糊测试方法之快照恢复变异（Snapshot restoration mutation）

我们提到的第二种内存模糊测试方法是快照恢复变异（Snapshot Restoration Mutation，SRM）。与变异循环插入方法一样，我们希望能够跳过目标应用中以网络为中心的部分，直接进攻我们关注的焦点，如这个例子中的 parse() 例程。同样，与变异循环插入方法一样，快照恢复变异方法也需要定位到解析代码的起始位置与结束位置。在标识出这些位置之后，我们的快照恢复变异工具会在到达解析代码的开始位置时为目标进程建立快照。在解析函数执行完成后，快照恢复变异工具恢复进程快照，对原先的数据产生变异，并使用变异得到的新数据重新执行解析代码。修改后的目标应用的控制流图如图 19.4 所示。

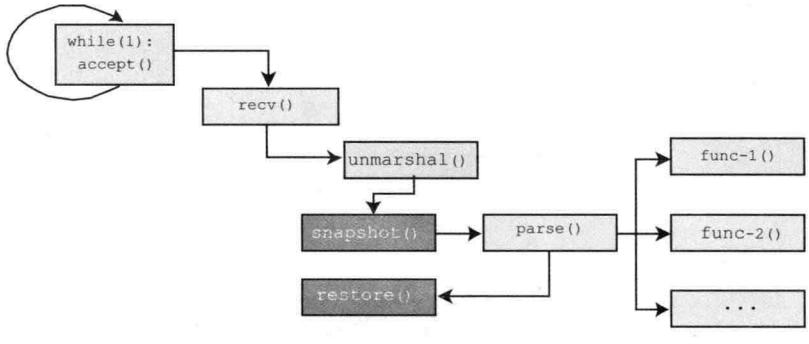


图 19.4 快照恢复变异的可视化展示

围绕目标应用的解析代码，我们再次创建了一个自给自足的数据变异循环。在第 20 章中，我们会实现一个具有基础功能的概念工具，以实践上文中描述的这两种方法。我们构建的这个原型性质的模糊测试器将帮助我们发现每种方法特定的优点和缺点。然而，在这之前，让我们首先详细讨论内存模糊测试的主要优点之一。

19.7　测试速度与处理深度

传统的以客户端方式对网络协议进行测试的方法有一个不可避免的短板。在这种方法中，测试用例必须通过网络逐个发送，依赖于模糊测试器的复杂程度，用于测试的客户端必须读取目标应用的响应并处理之。当需要对具有较深层次的协议进行模糊测试时，这个短板越发明显。例如，考虑 POP 邮件协议，该协议通常通过 TCP 端口 110 提供服务。一个合法的获取邮件的数据通信看上去类似这样（粗体显示的是用户输入）：

```
$ nc mail.example.com 110
+OK Hello there.
user pedram@openrce.org
+OK Password required
pass xxxxxxxxxxxx
+OK logged in
list
+OK
1 1673
2 19194
3 10187
... [省略一部分输出内容] ...
.
retr 1
+OK 1673 octets follow.
Return-Path: <ralph@openrce.org>
Delivered-To: pedram@openrce.org
... [省略一部分输出内容] ...
retr AAAAAAAAAAAAAAAAAAAAAAAA
-ERR invalid message number. Exiting
```

如果我们要实现一个网络模糊测试器来对 RETR 命令的参数进行模糊测试，该模糊测试器将不得不在执行每个测试用例时重连邮件服务器，以及在邮件服务器上重新认证，一步步进入到需要处理的状态。替代方法是，我们可以完全专注在 RETR 参数的处理上，通过上文介绍的其中某种内存模糊测试方法，仅在期望的“处理深度”上进行测试。对处理深度和处理状态的讨论可以在第 4 章中找到。

19.8　错误检测

内存模糊测试的长处之一在于检测模糊测试器引发的错误。既然我们已经可以在很

低的层次上对目标应用进行操作，要实现错误检测只需要简单的额外工作即可。如果我们将模糊测试器实现为调试器，使用变异循环插入或快照恢复变异方法，模糊测试器都能够在每个变异的迭代过程中监视异常条件。模糊测试器能够把错误的位置连同运行时上下文信息一同存储起来，这些被存储的信息可供研究者用来决定缺陷的精确位置。读者应该能够注意到，对于使用了反跟踪技术的目标应用（例如前面提到的 Skype），应用所使用的反跟踪技术会影响到实现方法的选择。

在错误检测方面，我们最大的困难是过滤掉那些误报。正如海森堡测不准原理[8]指出的，在内存模糊测试中使用“显微镜”对我们的目标进程进行观测会导致目标进程的行为表现发生变化。由于我们使用了目标应用未期望的方法直接修改了目标的进程状态，因此有很大的机会发生与我们的期望不相干的错误。考虑图 19.5。

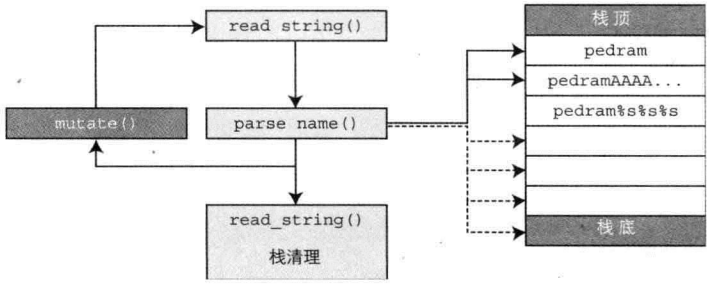


图 19.5　在变异循环插入中的栈耗尽

在图 19.5 中，我们使用变异循环插入方法来包装 parse_name() 和 read_string() 。在每个迭代中我们对 parse_name() 的输入进行变异，试图触发错误。然而，因为我们刚好把钩子设置在栈分配之后，清除之前，因此，在每个循环迭代中我们都会丢失栈空间。最终，可用的栈空间会被耗尽，并因此导致一个不可恢复的错误，该错误完全由我们的模糊测试过程引入，并非目标应用中本身存在的。

19.9　小结

内存模糊测试不是为不够坚强的人准备的。在可以开始内存模糊测试之前，必须需要具有技能的研究者对目标应用进行逆向工程，指出插入和插装的点。此外，研究人员还需要确定内存模糊测试发现的问题是否真的可以在正常情况下通过用户输入触发。多

[8] http://en.wikipedia.org/wiki/Heisenberg_principle

数情况下我们都没法发现“立即有效”的问题，而只能发现可以考虑的有可能发现问题的输入。当然，考虑到内存模糊测试能够带来的诸多好处，这些缺点也就可以容忍了。

就我们所知，内存模糊测试的概念首先由来自 HBGary 有限责任公司的 Greg Hoglund 在 2003 年的黑帽联邦[9]和黑帽美国[10]安全会议上提出。HBGary 公司提供了一个名为 Inspector[11]的商业内存模糊测试方案。当提到进程内存快照时，Hoglund 在其报告中隐约提到了一些他们专有的技术。在后续的介绍创建自动化工具的章节中，我们将会详细讨论内存快照技术及其他技术。

313

在第 20 章中我们将会逐步展示如何实现和使用一个定制的内存模糊测试框架。就我们所知，这是第一个着手处理内存模糊测试任务的开源的概念验证型工具。该框架的进展发布在本书的网站 http://www.fuzzing.org 上。

---

9 http://www.blackhat.com/presentations/bh-federal-03/bh-fed-03-hoglund.pdf

10 http://www.blackhat.com/presentations/bh-usa-02/bh-usa-03-hoglund.pdf

11 http://hbgary.com/technology.shtml

第20章

自动化内存模糊测试

"I hear there's rumors on the Internet that we're going to have a draft."

——George W. Bush, second presidential debate, St. Louis, MO, October 8, 2004

在第 19 章中，我们介绍了内存模糊测试的概念。虽然在本书前面的章节中我们尽量注意平等地覆盖 Windows 和 UNIX 平台，但由于本主题过于复杂，因此我们决定只关注 Windows 平台上的内存模糊测试。更具体地说，在本章中我们只关注 x86 架构上的 32 位 Windows 平台。在本章中，我们将详细讨论如何一步步地创建一个基础的内存模糊测试器。我们将列出期望的特性集，讨论我们采用的方法，决定合适的编程语言。最后，在一个案例研究之后，我们将给出该模糊测试方法的优势及可改进的地方。本章我们讨论的所有代码都是开源的，发布在 http://www.fuzzing.org 网站上。欢迎读者参与并贡献实现新功能的代码，或是报告发现的缺陷。

在开始之前，我们提醒读者注意，虽然在前面的章节中我们已经涉及 x86 体系结构和 Windows 平台的底层，并对此进行了一些解释，但我们并没有深入讨论这些主题，因为对这些主题的深入讨论超出了本书的范围。我们在本章中描述了多种概念，本章中我们开发的代码也会在后面章节中发挥作用，比如在第 24 章中。

20.1 内存模糊测试工具特性集

在第 19 章中我们介绍了两种内存模糊测试的方法：MLI（变异循环插入）和 SRM

（快照恢复变异）。我们希望在本章将要开发的基础工具中同时实现这两种方式。对 MLI 方法，我们需要具备修改目标应用的某条指令来创建循环，然后在循环开始的地方放置钩子，并在钩子中对目标应用的缓冲区进行变异。对 SRM 方法，我们需要能够在目标进程的两个特定点上放置钩子，在其中一个点上制作目标内存空间的一个拷贝，然后在另一个点上恢复目标进程的内存空间。为便于读者参考，我们在图 20.1 和图 20.2 中以可视化的方式表示了我们的需求，图中的粗体文字指出了我们需要对目标进程做出的修改，深灰色背景的文本框则表明我们的内存模糊测试器必须要增加的函数。

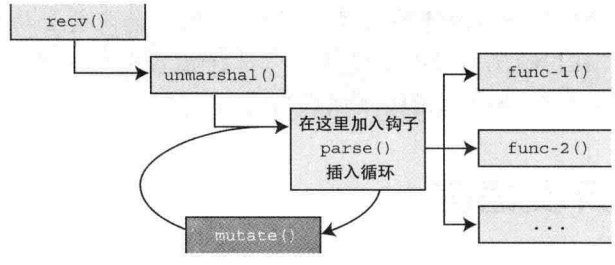


图 20.1　变异循环插入的修改需求

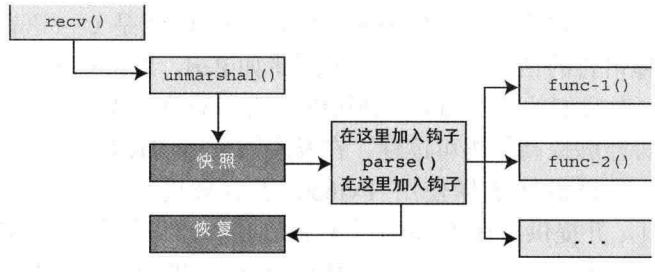


图 20.2　快照恢复变异的修改需求

为了实现以上列出的对进程进行插装的需求，我们将编写一个定制的 Windows 调试器。将我们的内存模糊测试器创建为调试器能够很好地满足我们所有的需求。在这里，我们对术语“插装”有一个宽泛的定义，期望这个术语描述尽可能多的功能。我们需要能够从目标进程的内存空间中随意读取数据，并向其中写入数据。我们也需要能够修改目标进程中任意线程的上下文。线程上下文[1]包含各种处理器特定的（processor-specific）寄存器数据，例如当前指令指针（EIP）、栈指针（ESP）、帧指针（EBP），以及其他通用目标的寄存器（EAX，EBX，ECX，EDX，ESI 和 EDI）。对线程上下文的控制允许

317

[1] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/context_str.asp

我们改变执行的状态，这是我们在 SRM 中恢复进程所需要的。庆幸的是，稍后我们将会看到 Windows 操作系统提供了可以满足我们所有需求的一个强力 API。

注意

由于 MLI 和 SRM 这两种方法在本质上来说是类似的，因此我们决定在开发模糊测试器及实现自动化的过程中仅关注 SRM。我们鼓励读者在本章例子的基础上自行练习，实现 MLI 方法。

相比 MLI，SRM 有一些独特的优势。例如，SRM 会恢复全局变量状态，而 MLI 恢复不了。另外，MLI 的使用者必须非常小心选择插装点。这方面的一个粗心的决定可能会导致内存泄漏，从而阻止模糊测试成功进行。

20.2 选择开发语言

在开始讨论特定的 Windows 调试 API 函数和我们将要使用的结构的背景信息之前，我们先决定采用何种编程语言进行开发，这样我们就可以用同一种语言书写供读者参考的代码片段。虽然根据需求，很自然地想到应该选择 C 或是 C++语言，但我们认为，如果能够使用解释语言的话，将会更好。使用诸如 Perl，Python 或是 Ruby 之类的解释语言有许多优点。对我们的目的而言，使用解释语言的最大优势是开发速度，对读者来说，使用解释语言能够提高代码可读性。在发现 ThomasHeller 开发的 Python 语言的 ctypes 模块[2]后，我们最终选择使用 Python 语言来进行开发。ctypes 模块提供了 WindowsAPI 接口，并提供了在 Python 中直接创建和操作复杂的 C 数据类型的能力。例如，下面的代码片段演示了通过 ctypes 调用 kernel32.dll 中导出的 GetCurrentProcessId() 函数是何等简单。

318

```
from ctypes import *

#创建一个方便的指向 kernel32 的捷径
kernel32 = windll.kernel32

#获得当前进程的进程 ID
current_pid = kernel32.GetCurrentProcessId()

print "ThecurrentprocessIDis%d" % current_pid
```

[2] http://starship.python.net/crew/theller/ctypes

创建 C 数据类型与传递 C 数据类型同样简单。ctypes 模块以 Python 类的形式提供了所有你需要的基本数据类型，如表 20.1 所示。

表 20.1　ctypes 中与 C 兼容的数据类型

ctypes 中的类型	C 类型	Python 类型
c_char	char	character
c_int	int	integer
c_long	long	integer
c_ulong	unsignedlong	long
c_char_p	char*	string 或是 None
c_void_p	void*	integer 或是 None

*基本数据类型的完整列表请参考 http://starship.python.net/crew/theller/ctypes/tutorial.html

可以通过传入可选的值来对所有这些可用的类型进行初始化，也可以通过设置这些数据类型对象的"value"属性达到同样的目的。在 ctypes 库中，通过使用 byref() 辅助函数可以实现以引用方式传值。要记住的一个重点是，像 c_char_p 和 c_void_p 之类的指针类型是不可变的。如果想要创建一块可变内存区域，需要使用 create_string_buffer() 辅助函数。要访问或是修改可变的块，请使用对象的 raw 属性。下面对 ReadProcessMemory() 进行调用的代码片段展示了以上我们提到的这些点：

```
read_buf = create_string_buffer(512)
count = ulong(0)

kernel32.ReadProcessMemory(h_process,\
    oxDEADBEEF,                       \
    read_buf,                         \
    512,                              \
    byref(count))

print 'Successfully read %d bytes:" % count.value
print read_buf.raw
```

ReadProcessMemory() API 带有 5 个参数：第一个参数是我们想要读取其内存空间的进程的句柄，第二个参数是我们要读取的进程内的地址，第三个参数是指向存储读取得到数据的缓冲区的指针，第四个参数是需要读取的总字节数，最后一个参数则是一个整数指针，该指针指向存储了实际读取到的字节数的整数。了解了进程中的内存读取后，接下来我们看看如何对进程中的内存进行写入操作。下面的代码片段展示了如何对进程的给定内存空间进行写操作：

```
c_data = c_char_p(data)
length = len(data)
count = c_ulong(0)

kernel32.WriteProcessMemory(h_process, \
    0xC0CAC01A,                        \
    c_data,                            \
    length,                            \
    by_ref(count))

print "Successfully wrote %d bytes:" % count.value
```

WritePorcessMemory() 函数与其姊妹函数 ReadProcessMemory() 的格式很类似。WriteProcessMemory() 同样带有 5 个参数：第一个是我们想要向其内存空间写入数据的进程句柄，第二个参数是我们要写入的数据地址，第三个参数是一个指针，指向我们要写入的数据缓冲区，第四个参数是要写入的字节数，最后一个参数同样是一个整数指针，指向储存了实际写入字节数的整数。在给定进程的内存空间中进行读/写操作是我们开发内存模糊测试器的必要条件，因此，读者将会在本章的后续部分看到这两个 API 函数发挥作用。

现在，我们已经具有了工具开发所需的基础知识，接下来，让我们简要地讨论一下为了创建一个调试器，我们还需要在 API 方面具备的知识。

320

20.3 Windows 调试 API

在第 19 章中，我们概要地介绍了典型 Windows 进程的内存布局与进程的各个组成部分。在进一步讨论内存模糊测试工具的开发之前，我们先来复习一些可能会用到，甚至可能会“滥用”的 Windows 调试 API，我们是否“滥用”了这些 API 完全取决于你的个人看法。

从 Windows NT 开始，Windows 操作系统就已经提供了一套强大的 API 函数和数据结构，利用这套 API 和数据结构，开发者能够相当容易地创建一个事件驱动的调试器。调试 API 的基础组件包括三类：函数[3]、事件[4]与结构[5]。因为我们需要实现多种需求，

[3] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_functions.asp

[4] http://msdn.miscrosoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_events.asp

[5] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_structures.asp

因此在后续的讨论中我们会涉及所有这三类组件。需要完成的第一个任务是使目标进程可以在我们调试器的控制下运行。有两种方法可以实现这一点：一种方法是在我们调试器的控制下加载目标进程；另一种方法则是启动目标进程后，再将我们的调试器附着在其上。要在调试器的控制下加载进程，可以使用下面的代码片段：

```
pi = PROCESS_INFORMATION()
si = STARTUPINFO()

si.cb = sizeof(si)

kernel32.CreateProcessA(path_to_file,
    command_line,          \
    0,                     \
    0,                     \
    0,                     \
    DEBUG_PROCESS,         \
    0,                     \
    0,                     \
    byref(si),             \
    byref(pi))

print "Start process with pid %d" % pi.dwProcessId
```

注意上面的代码中 CreateProcess 后的“A”，Windows API 通常会导出 Unicode 和 ANSI 版本的 API。不带 A 的版本仅是一个简单的包装器。根据 ctypes 的定义，我们必须使用带 A 的版本。要知道一个特定 API 是否同时导出了 ANSI 和 Unicode 的版本，最简单的方法就是在 MSDN 中查看文档。例如，MSDN 的 CreateProcess 页面[6]靠近结尾处有如下的文字：“实现为 CreateProcessW (Unicode) 和 CreateProcess (ANSI) 两种方式”。PROCESS_INFORMATION 和 STARTUP_INFO 结构以引用方式被传入 CreateProcess() API，在该 API 执行完成后，我们后续要使用的信息，例如被创建的进程标识（pi.dwProcessId）和被创建进程的句柄（pi.hProcess）会被填入这两个传入的结构。此外，如果想要将调试器附着到一个已在运行中的进程，可以调用 DebugActiveProcess()：

```
#附着到给定进程 ID
kernel32.DebugActiveProcess(pid)
```

[6] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createprocess.asp

```
#在支持解除附着的系统上允许解除附着
try:
    kernel32.DebugSetProcessKillOnExit(True)
except:
    pass
```

注意，我们在上面代码片段中调用的 API 既不需要附加 A 也不需要附加 W。DebugActiveProcess() 例程将我们的调试器附着到给定的进程上。在调用 DebugActiveProcess() 之前，也许需要提升权限等级才能正常执行，但我们以后再讨论这个话题。DebugSetProcessKillOnExit()[7] 例程从 Windows XP 开始就已经存在，该例程允许我们退出调试器而无须中止被调试的进程。我们将这个调用封装在 try/except 中，以防止在不支持该 API 的平台（例如 Windows 2000）上退出调试器时发生错误。当可以将目标进程置于调试器的控制下之后，接下来，我们需要实现调试事件处理循环。调试事件循环可以被假想成城镇中老土而爱管闲事的邻居 Agnes。Agnes 坐在窗前观察邻居家发生的所有事情。虽然 Agnes 看到了所有事情，但她看到的大部分事情都不足以让她感兴趣到给朋友打电话。偶尔，如果发生了真正有趣的事情，例如，邻居的孩子由于追赶猫爬到了树上，结果跌下来弄折了自己的胳膊，Agnes 就会立即给警察打电话。我们的调试事件循环非常像 Agnes，它能够看到许多事件。我们需要向它指明哪些事件是我们感兴趣，并希望在其上进行更多处理的。以下是一个典型的调试事件循环的框架代码：

322

```
debugger_active   = True
dbg               = DEBUG_EVENT()
continue_status   = DBG_CONTINUE

while debugger_active:
    ret = kernel32.WaitForDebugEvent(byref(dbg),100)

    #如果没有调试事件发生，继续
    if not ret:
        continue

    event_code = dbg.dwDebugEventCode

    if event_code == CREATE_PROCESS_DEBUG_EVENT:
```

[7] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugsetprocesskillonexit.asp

```
        #创建了新的进程

    if event_code == CREATE_THREAD_DEBUG_EVENT:
        #创建了新的线程

    if event_code == EXIT_PROCESS_DEBUG_EVENT:
        #进程退出

    if event_code == EXIT_THREAD_DEBUG_EVENT:
        #线程退出

    if event_code == LOAD_DLL_DEBUG_EVENT:
        #新 DLL 被加载

    if event_code == UNLOAD_DLL_DEBUG_EVENT:
        #DLL 被卸载

    if event_code == EXCEPTION_DEBUG_EVENT:
        #抓住了一个异常

    #继续处理
    kernel32.ContinueDebugEvent(dbg.dwProcessId,     \
        dbg.dwThreadId,                              \
        continue_status)
```

调试事件处理循环主要基于对 WaitFotDebugEvent()[8]的调用，该调用的第一个参数是指向 DEBUG_EVENT 结构的一个指针，第二个参数是在调试器中等待调试事件发生的毫秒数。如果发生了一个调试事件，DEBUG_EVENT 结构的 dwDebugEventCode 属性中就会含有调试事件类型。通过检查这个变量，我们可以判断调试事件的发生原因：是进程，还是线程的创建或退出导致？是加载或卸载 DLL 导致？或者，是由一个调试异常事件触发？如果发生的是调试异常事件，我们可以通过检查 u.Exception.ExceptionRecord.ExceptionCode 结构的 DEBUG_EVENT 属性值判断异常的具体原因。MSDN 给出了许多可能的异常代码[9]，但我们主要关注以下这些异常原因。

323

- EXCEPTION_ACCESS_VIOLATION：由于尝试向非法的内存地址读取或是写入导致的访问违例。

---

[8] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/waitfordebugevent.asp

[9] http://msdn.microsoft.com/librarry/default.asp?url=/library/en-us/debug/base/exception_record_str.asp

- EXCEPTION_BREAKPOINT：由于遇到断点触发的异常。
- EXCEPTION_SINGLE_STEP：单步捕获（trap）已经设置为使能并且执行了一条单独的指令。
- EXCEPTION_STACK_OVERFLOW：违规线程耗尽了其栈空间。这是一个典型的失控递归的信号，而且经常是由 DoS 引起的。

对各种调试事件和异常，我们可以编写任意逻辑对其进行处理。当处理完所有报告的事件后，我们通过调用 ContinueDebugEvent() 允许违规线程继续执行。

20.4 整合以上的内容

到目前为止，我们已经讨论了基础的 Windows 内存布局，列出了我们的需求，选定了开发语言，对 ctypes 模块进行了基本描述，介绍了 Windows 调试 API 的基础知识。现在，只剩下一些需要解决的明显问题了：

324

- 我们如何在目标应用中特定的点放置“钩子”？
- 我们如何生成与恢复进程快照？
- 我们如何在目标内存空间中进行定位和变异？
- 我们如何选择放置钩子的点？

20.4.1 如何在目标应用中特定的点放置“钩子”

前面提到过，根据我们选择的方法，可以通过调试器断点在进程中放置钩子。我们给定的 Windows 32 位平台支持两类断点：硬件断点和软件断点。80x86 兼容的处理器支持最多 4 种硬件断点。这些断点可以被设置为读取、写入，执行任何单、双，4 字节范围时被触发。为了设置硬件断点，我们必须修改目标进程上下文，设置调试寄存器 DR0，DR1，DR2，DR3 和 DR7 的值。前四个寄存器指明硬件断点的地址，而 DR7 寄存器则包含标志位，指示断点在什么范围，以及什么类型的访问时被激活。硬件断点是非侵入式的，不会修改你的代码。与此相反，软件断点则一定需要修改目标进程，通过单字节指令 INT3 实现，该指令的十六进制形式为 0xCC。

我们既可以使用硬件断点，也可以使用软件断点来完成给定的任务。如果我们在某些点上需要设置的钩子超出了硬件断点允许的四种类型，就需要使用软件断点。为了更好地理解这一处理过程，假设需要在一个假想程序的内存地址 0xDEADBEEF 处设置一

个软件断点，我们来看看所需的操作步骤。首先，我们的调试器必须调用 ReadProcessMemory() API 读取目标地址的原始字节值并保存下来，如图 20.3 所示。

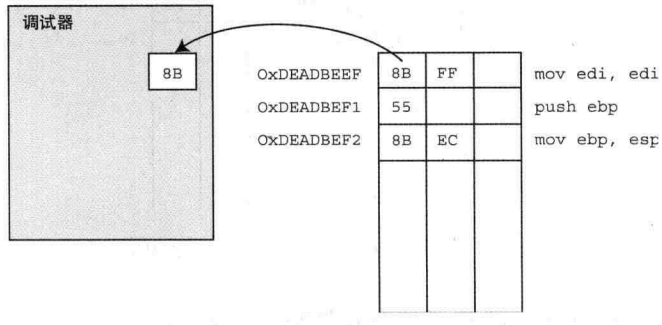


图 20.3　保存断点地址处的原始字节

注意，目标地址的第一条指令实际上是双字节指令。下一步，我们使用 WriteProcessMemory() API 将 INT3 指令写入目标地址，如图 20.4 所示。

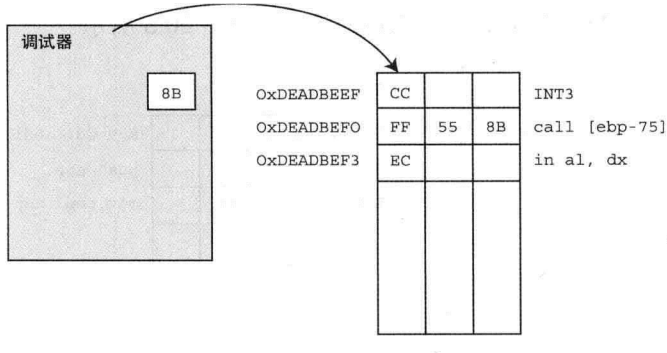


图 20.4　写入 INT3 操作码

完成这个操作后，目标地址处的指令变成什么样了？嗯，插入的 0xCC 被反汇编成单字节的 INT3 指令。mov edi,edi 指令的第二个字节（0xFF），连同 push ebp 指令（0x55）和 mov ebp,esp 指令的第一个字节（0x8B）一起被反汇编成 call [ebp-75]指令。剩下的 0xEC 字节，则被反汇编成单字节指令 in al,dx。现在当执行到地址 0xDEADBEEF 时，INT3 指令会触发一个带 EXCEPTION_BREAK_POINT 异常码的 EXCEPTION_DEBUG_EVENT 调试事件，而我们的调试器会在调试事件循环中捕捉到这个调试事件。该点上的处理如图 20.5 所示。

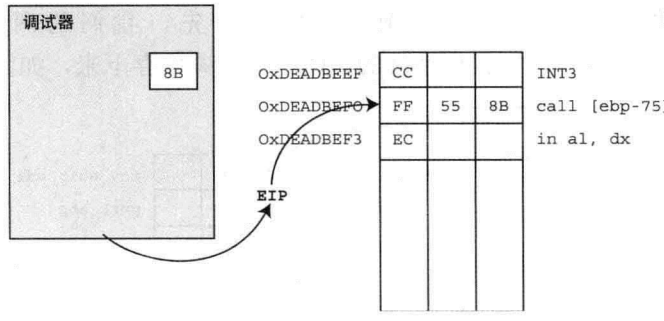


图 20.5　捕获 EXCEPTION_BREAKPOINT

现在，我们成功地插入并捕获了一个软件断点，但这只是完成了一半的工作。注意，在上面描述的情况下，我们的原始指令并没有被执行。除此之外，指令指针（EIP，该指针让 CPU 知道从哪里获取，解码，并执行下一条指令）指向 0xDEADBEF0 而不是 0xDEADBEEF。这是因为我们在地址 0xDEADBEEF 处插入的 INT3 指令被成功地执行后，EIP 被更新为 0xDEADBEEF+1。在进程可以被继续执行之前，我们将必须修正 EIP
326 的值，并恢复地址 0xDEADBEEF 处的原始指令，如图 20.6 所示。

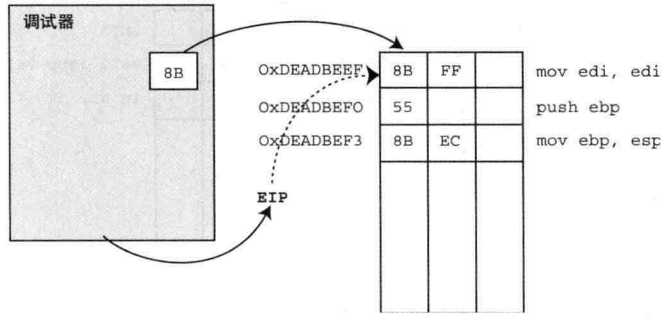


图 20.6　调整 EIP

恢复内存地址 0xDEADBEEF 处的原始指令这个任务我们并不陌生。然而，改变指令指针 EIP 的值就比较困难了。在本章的前面部分我们提到过，线程上下文包含了各种处理器相关的寄存器数据，我们现在关注的指令指针 EIP 就包含在内。通过调用
327 GetThreadContext()[10]API，我们能够获得任意给定线程的上下文，向该 API 传入当前线程句柄与指向 CONTEXT 结构的指针，然后，我们就可以修改返回的 CONTEXT 结构

[10] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/getthreadcontext.asp

的内容，并调用 SetThreadContext()[11]API，传入当前线程句柄来修改上下文：

```
context = CONTEXT()
context.ContextFlags = CONTEXT_FULL

kernel32.GetThreadConext(h_thread,byref(context))

context.Eip -= 1

kernel32.SetThreadContext(h_thread,byref(context))
```

此时，原先的执行上下文被恢复，接下来我们就可以继续执行进程了。

20.4.2　如何生成与恢复进程快照

为了回答这个问题，我们需要首先考虑另一个问题：当进程运行的时候，哪些东西会发生改变？答案是，很多东西都会发生改变。进程运行时会创建新线程，终止老线程。进程运行时还会打开和关闭文件、套接字、窗口句柄及其他组件。进程中会发生内存的分配与释放，读取与写入。单个线程上下文中的各种寄存器高度不稳定，其值一直在发生改变。我们可以使用虚拟机技术完成生成与恢复进程快照的任务，例如，VMWare[12]允许我们生成完全的系统快照或是从完全的系统快照中恢复。然而，使用 VMWare 生成完全的系统快照或是从完全的系统快照中恢复需要特别长的时间，而且，还需要虚拟机的客户机与虚拟机的主机上的连接器之间采用某种形式进行通信。 我们决定不采用虚拟机方法，而是借鉴一个以前讨论过的技术[13]，只考虑线程上下文和内存的变化实行“欺骗”。我们生成快照的过程包含两个步骤。

第一个步骤，我们保存目标进程中每个线程的上下文。我们已经看到，获取与设置单个线程的上下文是非常简单的。现在我们需要封装负责枚举属于我们目标进程的系统线程的逻辑代码。通过工具辅助函数[14]可以达成这一目标。首先，我们通过指定 TH32CS_SNAPTHREAD 标志获取一个包含所有系统线程的列表。

```
thread_entry =THREADENTRY32()
```

328

[11] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/setthreadcontext.asp

[12] http://www.vmware.com

[13] GregHoglund, “运行时反编译”，黑帽大会

[14] http://msdn2.microsoft.com/en-us/library/ms686832.aspx

```
contexts     =[]

snapshot=kernel32.CreateToolhelp32Snapshot(          \
    TH32CS_SNAPTHREAD,                               \
    0)
```

接下来，我们从列表中取得第一个线程项。然而，在进行这个操作之前，我们必须要使用 Thread32First() API 初始化线程项结构中的 dwSize 变量。接着，我们向 Thread32First() API 传入前面得到的快照，以及一个指向我们的线程项结构的指针：

```
thread_entry.dwSize=sizeof(thread_entry)

success=kernel32.Thread32First(         \
    snapshot,                           \
    byref(thread_entry))
```

最后，我们循环查找属于目标进程的进程 ID（pid）的这些线程。如果找到了，我们就使用 OpenThread() API 取得线程句柄，用本章前面部分描述的方法获得线程上下文，并将其添加到一个列表中：

```
while success:
    if thread_entry.th32OwnerProcessID == pid:
        context = CONTEXT()
        context.ContextFlags = CONTEXT_FULL

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

        kernel32.GetThreadContext(                  \
            h_thread,                               \
            byref(context))

        contexts.append(context)

        kernel32.CloseHandle(h_thread)

    success = kernel32.Thread32Next(                \
        snapshot,
        byref(thread_entry))
```


保存了属于我们进程的每个线程的上下文之后，可以在以后通过再次循环遍历所有

系统线程，恢复保存的线程上下文来恢复我们的进程快照。

第二个步骤，我们保存每个可能发生变化的内存块的内容。回忆第 19 章的描述，32 位 x86 的 Windows 平台上的每个进程能够“看到”它自己的 4GB 内存空间。这 4G 内存空间中底部的一部分被保留给我们的进程使用（0x0000000～0x7FFFFFFF）。该内存空间进一步被分成内存页（内存页的大小通常为 4096 字节）。最后，内存访问权限以内存页为最小单位被应用。我们不需要保存所有被用到的内存页，仅需要保存可能发生变化的内存页的快照，以节省时间和资源。我们忽略所有设置为“阻止写入”的页，也就是说，忽略所有具有以下属性的页：

- PAGE_READONLY
- PAGE_EXECUTE_READ
- PAGE_GUARD
- PAGE_NOACCESS

另外，我们也需要忽略可执行代码所在的页，因为他们不大可能改变。简单地循环调用 VirtualQueryEx()[15] API 例程即可逐一访问所有可用的内存页，通过这种方式，我们能够获得特定虚拟地址范围内所有页面的信息：

```
cursor          =0
memory_blocks=[]
read_buf        =create_string_buffer(length)
count           =c_ulong(0)
mbi             =MEMORY_BASIC_INFORMATION()

while cursor < 0xFFFFFFFFF:
    save_block = True

    bytes_read = kernel32.VirtualQueryEx(       \
        h_process,                              \
        cursor,                                 \
        byref(mbi),                             \
        sizeof(mbi))

    if bytes_read < sizeof(mbi):
        break
```

330

[15] http://msdn2.microsoft.com/en-us/library/aa366907.aspx

如果调用 VirtualQueryEx() 失败，我们就认为已经遍历了所有可用的用户空间，并让代码跳出循环。对循环中发现的每一个内存块，我们检查它是否设置了我们感兴趣的页面访问权限：

```
if mbi.State != MEM_COMMITor   \
    mbi.Type == MEM_IMAGE:
        save_block = False

if mbi.Protect & PAGE_READONLY:
    save_block = False

if mbi.Protect & PAGE_EXECUTE_READ:
    save_block = False

if mbi.Protect & PAGE_GUARD:
    save_block = False

if mbi.Protect & PAGE_NOACCESS:
    save_block = False
```

如果发现了我们想要包含到快照中的内存块，我们就使用 ReadProcessMemory()[16]API 读取该内存块的内容，并将该内存块中的数据连同内存块的信息存入快照列表中。然后，我们递增 cursor 的值，继续扫描下一个内存块：

```
if save_block:
    kernel32.ReadProcessMemory(        \
        h_process,                     \
        mbi.BaseAddress,               \
        read_buf,                      \
        mbi.RegionSize,                \
        byref(count))

    memory_blocks.append((mbi,read_buf.raw))

cursor += mbi.RegionSize
```


细心的读者也许立刻就能注意到以上的方法并非完美。例如，如果我们快照中的某个给定页在这个时刻被标记为 PAGE_READONLY，但稍后该页被修改成“可写”了，那怎么办？这是个好问题。对这个问题的答案是，我们简单地忽略这种情况。毕

16 http://msdn2.microsoft.com/en-us/library/ms680553.aspx

竟，我们从未承诺这种方法是完美的！实际上，我们用这个机会再次强调本章主题的实验色彩。提醒那些对本章主题之外的东西有兴趣的读者，一个潜在的克服该缺点的方案是使用钩子“钩住”各种调整内存权限的函数，基于发现的修改来调整我们的监视方法。

以上的两个步骤合在一起提供了生成和恢复进程相关快照所必须的基础。

20.4.3　选择在何处放置钩子

由于不存在确定的方法来选择放置钩子的地点，因此，这个问题是我们从科学转向艺术的转折点。决定在哪里放置钩子需要一个身经百战的逆向工程师的经验。简单的说，选择合适的放置钩子的地点，就是要找到负责解析用户输入数据的代码的起始位置和结束位置。假如你事先并不清楚目标软件的工作方式，那么，应用调试器进行跟踪会很好地帮助缩小选择范围。我们将在第 23 章中深入讨论调试器跟踪的细节。

为了让读者更好地理解这部分内容，我们会在下文中列举一个例子。

20.4.4　如何定位和变异目标内存空间

为了进行内存模糊测试，我们需要诸多的准备工作。这些准备工作中的最后一项是选择进行变异的内存位置。与上一个问题一样，选择进行变异的内存位置的过程与其说是科学，不如说是艺术。同样，使用调试器进行跟踪能够帮助我们找到需要进行变异的目标内存空间。然而，一般来说，我们应该给定一个初始的钩子位置，例如指向我们的目标数据或是指向目标数据邻近区域的指针。

20.5　PyDbg，一个新朋友

332

正如你知道的那样，写一个调试器毫无疑问需要花费相当的精力。幸运的是，一个叫做 PyDbg[17]的 Python 类已经实现了本章中到目前为止我们讨论过的话题（甚至更多）。你可能会在心里嘀咕，为什么我们不在本章的开头部分介绍这个 Python 类？好吧，我们故意的。基础知识对于理解本章的主题很重要，如果我们在一开始就告诉你捷径的话，你可能就不会关注我们前面提到的那些基础内容了。

---

[17] http://oepnrce.org/downloads/details/208/PaiMei

对 PyDbg 来说，通过 Windows 调试 API 进行进程插装是小菜一碟。PyDbg 可以帮助你很容易地做到下面这些：

- 读取，写入与查询内存
- 枚举进程，附着在进程上，解除附着以及中止进程
- 枚举线程，暂停与继续执行线程
- 设置断点，移除与处理断点
- 生成快照与恢复进程状态（SRM 中的 S 和 R）
- 解析函数地址
- 其他……

下面是一个简单示例，在该示例中我们实例化一个 PyDbg 对象，将其附着到 PID 为 123 的目标进程上，并进入调试循环：

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

dbg = newPydbg()
dbg.attach(123)
dbg.debug_event_loop()
```

这算不上什么了不得的东西，接下来，我们让这个例子多些功能。基于上面的示例代码，在下面的代码片段中，我们将断点设置在 Winsock 的 recv() API 上，并注册了一个回调处理函数，当断点触发时，回调处理函数会被调用。

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

ws2_recv = None

def handler_bp(pydbg,dbg,context):
    global ws2_recv

    exception_address=\
        dbg.u.Exception.ExceptionRecord.ExceptionAddress

    if exception_address == ws2_recv:
        print "ws2.recv() called!"

    return DBG_CONTINUE
```

```
dbg = newpydbg()
dbg.set_callback(EXCEPTION_BREAKPOINT,handler_bp)
dbg.attach(123)

ws2_recv = dbg.func_resolve("ws2_32","recv")
dbg.bp_set(ws2_recv)
```

代码中的粗体部分是我们新增的内容。新增内容的第一部分是我们定义的 handler_bp() 函数，该函数带三个参数：第一个参数接收到的是我们创建的 PyDbg 实例。第二个参数接收到的是来自调试事件循环的 DEBUG_EVENT[18]结构，包含刚刚发生的调试事件的各种信息。第三个参数接收到的是调试事件所在线程的上下文。我们的断点处理函数只是检查异常发生的地址是否与 Winsock 的 recv() API 的地址相同，如果是，就打印一条信息。断点处理函数返回 DBG_CONTINUE，通知 PyDbg 我们的异常处理已经完成，PyDbg 就会继续目标进程的执行。如果查看我们的调试器脚本的主体部分，你会发现我们增加了一个对 PyDbg 例程 set_callback()的调用。该例程用来为 PyDbg 注册一个处理特定调试事件或异常的回调函数。在这个案例中，我们希望在断点被触发时调用 handler_bp()。新增代码的最后两行是对 func_resolve() 和 bp_set() 的调用。前一个调用解析出 recv() API 在 Windows 的 ws2_32.dll 模块中的地址，并将其保存到一个全局变量中。后一个调用在解析得到的地址处设置了一个断点。当该调试器被附着到一个目标进程后，任何对 Winsock 的 recv() API 的调用都会使得调试器显示信息“ws2.recv() called”，并继续正常执行。这同样并不值得让我们特别兴奋，但，现在我们能够更进一步，开始创建我们的第一个概念级别的内存模糊测试器了。

20.6　一个人造的示例

前面我们已经讨论了大量的背景和预备知识，是时候来创建一个初始的概念级别的工具来向你展示这些理论上的东西真的可操作。读者能够从 fuzzing.org 网站上找到 fuzz_client.exe 和 fuzz_server.exe，以及这两个工具的源代码。现在我们不必对源代码进行研究。为了重现更接近真实世界的场景，我们假设必须通过逆向工程才能获得被测目标的信息。这一对客户端-服务器（fuzz_client.exe 和 fuzz_server.exe）是非常简单的模糊测试目标。服务器被启动后，会绑定在 TCP 端口 11427 上等待客户端连接。客户端连上服务器，并向服务器发送数据，随后服务端解析收到的数据。服务器是如何解析数

18

据的？我们并不确切地知道服务器的解析方法，而且目前我们也并不真正关心这个问题，因为我们的目标是对目标进行模糊测试，而不是评审目标的源代码或是二进制码。我们首先运行服务器：

```
$./fuzz_server.exe
Listening and waiting for client to connect...
```

接下来，我们启动客户端，客户端带两个命令行参数：第一个参数是服务器的 IP 地址，第二个参数是要发送给服务器的数据：

```
$./fuzz_client.exe 192.168.197.1 'sending some data'
connecting...
sending...
sent...
```

客户端连接到了位于 192.168.197.1 的服务器，并向其成功地传输了字符串“sending some data”。在服务器端我们能看到服务器输出了以下信息：

```
client connected.
received 17bytes
parsing:sending some data
exiting...
```

服务器成功地接收到了我们传过去的 17 个字节，对其进行解析，然后退出。然而，
335 当我们检查在网络上传输的数据时，我们定位不到这些数据。紧跟在 TCP 的三次握手之后的数据包应该包含我们的数据，但实际上，该数据包包含的数据不是我们期望的“sending some data”，而是图 20.7 中的 Ethereal[19]屏幕截图中高亮显示的内容。

显然，客户端一定在将数据包发送到网络之前，对数据进行了干扰（mangle）、加密（encrypt）、压缩（compress）或是混淆（obfuscate）处理。而服务器一定在开始解析数据包之前对其进行解混淆处理，因为我们可以在服务端输出的信息日志中看到正确的字符串。如果使用传统方法对我们的示例服务器进行模糊测试，就需要对混淆例程进行逆向工程。

一旦揭开混淆方法的秘密，我们就能够生成和发送任意数据。但当你看到后面的解决方案时，会意识到在这个案例中这些工作并不重要。然而，出于继续这个例子的目的，我们假设需要消耗显著的资源才能对这个混淆方法进行逆向工程。这是一个展示内存模

[19] http://www.ethreal.com/, Ethereal 是一个网络协议分析器

糊测试价值的极好的例子（这毕竟是一个人为的示例）。我们将会在 fuzz_server.exe 中传入数据被解混淆之后的某处放置钩子，避免需要分析和解密混淆例程。

我们需要在 fuzz_server.exe 中找出两个位置来完成我们的任务：第一个是快照点，也就是，我们需要在执行到哪个点时保存目标进程的状态？第二个是恢复点，也就是，我们需要在执行到哪个点时回退进程状态，变异我们的输入，继续我们插装的执行循环？为了回答这两个问题，我们需要使用调试器对输入进行追踪。下面我们会使用 OllyDbg[20]，一个自由可用的强大的 Windows 平台上的用户模式调试器。

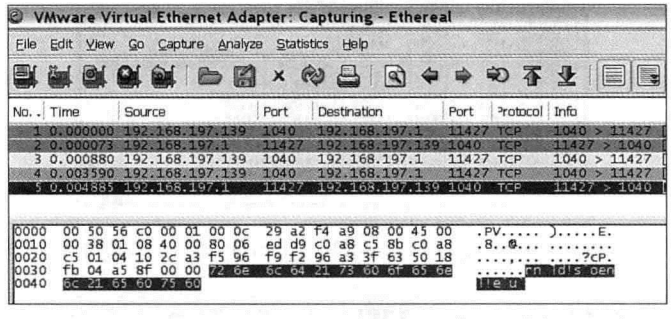


图 20.7　Ethereal 抓取的客户端-服务器之间的通信数据

Ethereal 网络嗅探器项目已经转移到 Wireshark 项目，读者可以从网站 http://www.wireshark.org 上下载得到。

几乎需要一整本书才能完整描述 OllyDbg 的功能及其使用方法，因此我们假设你已经熟悉了该工具的使用。我们首先要做的事情是找到 fuzz_server.exe 中接收数据的位置。我们知道 fuzz_server.exe 通过 TCP 接收数据，因此将 fuzz_server.exe 加载到 OllyDbg 中，并在 WS2_32.dll 的 recv() API 上设置断点。在 OllyDbg 中打开模块列表，选中 WS2_32.dll，按下 Ctrl+N 键打开模块内的名字列表（如图 20.8 所示）。然后滚动到 recv() 函数并按下 F2 键设置断点。

设置好断点后，按下 F9 键继续执行进程。然后，运行 fuzz_client.exe。当 fuzz_client.exe 发送数据后，OllyDbg 会因为到达了我们设置的断点而立即暂停 fuzz_server.exe 的执行。然后我们按下 Alt+F9 键来“执行直到遇到用户代码”。现在可以看到从 fuzz_server 发出的对 WS2_32 的调用。一直按 F8 键，单步执行到 print() 调用，

336

[20] http://www.ollydbg.de

该调用会显示服务端收到了多少个字节数据的信息。接下来，如图 20.9 所示，我们看到一个调用，该调用的对象是 fuzz_server.exe 中位于地址 0x0040100F 处的一个未命名子例程。在 OllyDbg 的 dump 窗口中检查该函数的第一个参数，可以发现该参数是一个指针，指向我们在 Ethereal 中看到的模糊化后的数据。那么，0x0040100F 处的例程是否就是负责对数据包数据进行解混淆的例程？

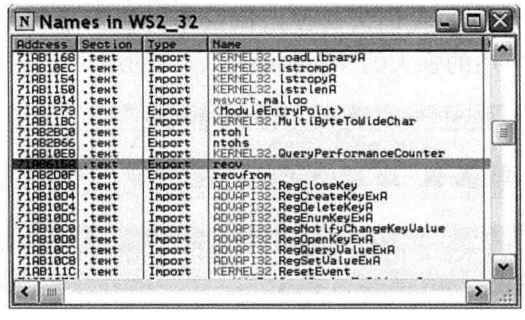


图 20.8　OllyDbg 工具，WS2_32.recv() 函数上设置了断点

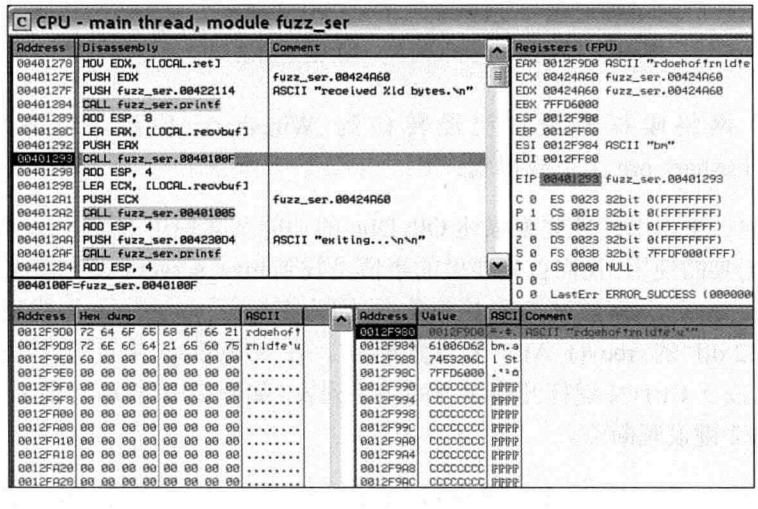


图 20.9　OllyDbg 显示的解混淆前的内容

这是找出混淆例程一个简单的方法：执行并看看会发生什么。再次按下 F8 键执行该函数调用，我们立马会看到，dump 窗口中显示的数据发生了变化，如图 20.10 所示。

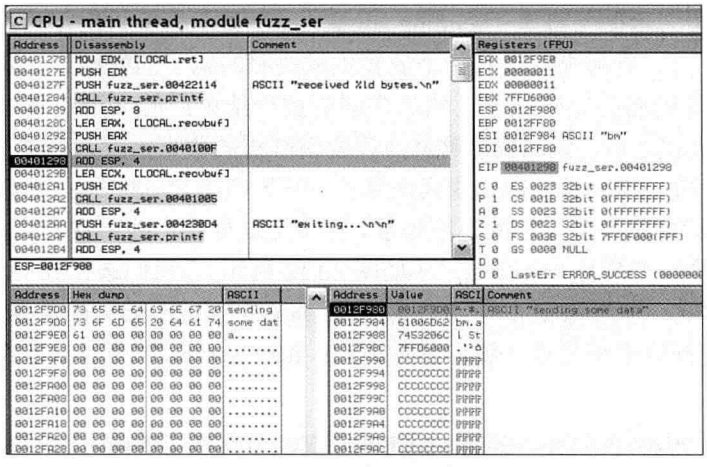


图 20.10　OllyDbg 显示的解密后的内容

太好了！现在我们知道了我们的快照点应该在这个例程之后的某个点。接着向下执行，我们会看到一个对内存地址 0x00401005 处的未命名例程的调用，然后是一个对 printf() 的调用。在对 printf() 的调用处，我们看到传入的字符串是“exiting...”。结合前面观察到的 fuzz_server 的行为，我们知道 fuzz_server 马上要退出了。因此，0x00401005 处的例程一定是我们要找的数据处理例程。使用 F7 键单步进入该例程，我们能看到一条跳转至 0x00401450 的无条件跳转指令，我们所怀疑为数据处理例程的最开头内容显示在图 20.11 中。

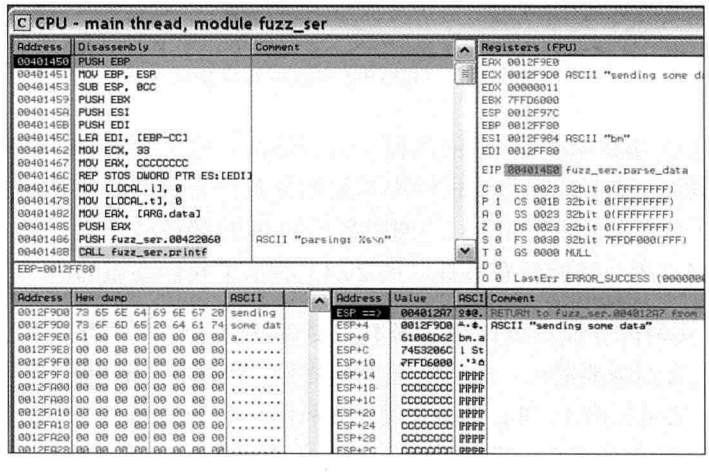


图 20.11　OllyDbg 中显示的解析例程开头的内容

注意解混淆之后的字符串“sending some data”显示为解析例程的第一个参数，位于 ESP+4 的位置。看起来这里是一个放置生成快照的钩子的好地点。我们能够在解析例程的开头保存进程的完整状态，然后在解析例程结束后的某个点将进程状态恢复回来。一旦恢复进程之后，我们就能够修改需要被解析的数据内容，继续执行，并一直重复这个过程。但是首先，我们需要找到恢复点。按下 Ctrl+F9 键来“执行直到返回为止”，然后按下 F7 或 F8 键直到我们的返回地址，在该地址处我们能再次看到对 printf() 的调用，且调用的传入参数是“exiting...”（如图 20.12 所示）。我们选择 printf() 调用之后的地址 0x004012b7 作为恢复点，这样我们能够看到 fuzz_server 在恢复之前打印出“exiting...”。选择这个恢复点让我们感觉很好：fuzz_server 想要退出但我们就是不让它退出。

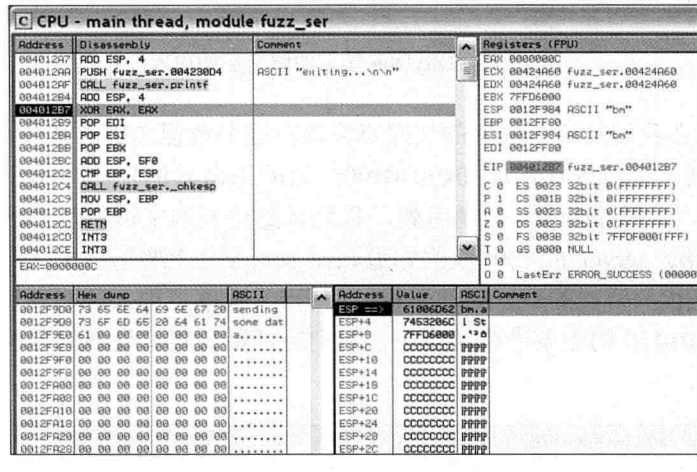


图 20.12　OllyDbg 中显示的恢复点

我们已经知道 0x0040100F 处的例程负责解混淆，位于 0x00401450 处的例程负责解析解码后的数据。我们选择了解析例程的起始位置作为我们的快照和变异点。我们略带随意地选择了位于 printf("exiting...")调用之后的 0x004012b7 作为恢复点。现在我们已经拥有了开始编码所必需的所有条件，图 20.13 展示了我们要做的事情的概念图。

用 PyDbg 实现内存模糊测试测试器不需要在我们已讨论过的工作上增加太多额外的工作。先导入我们需要的库，然后定义全局变量，保存我们选择的快照点和恢复点之类的信息。接下来是标准 PyDbg 骨架代码，实例化一个 PyDbg 对象，注册回调函数（回调函数是这里最重要的部分，我们将在后面定义该函数），定位我们的目标进程，将模糊测试器附着到目标进程上，在快照点和恢复点上设置断点，最后进入调试事件循环：

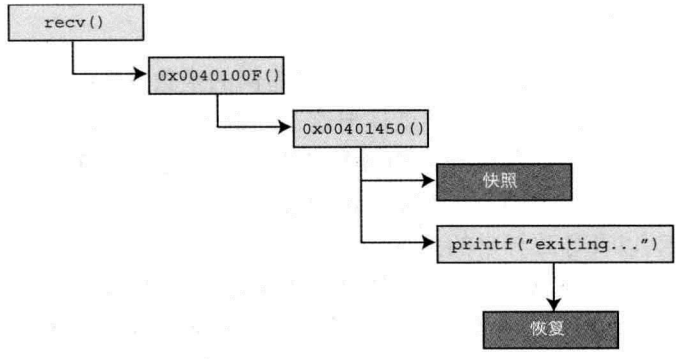


图 20.13　概念图

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

import time
import random

snapshot_hook     = 0x00401450
restore_hook      = 0x004012B7
snapshot_taken    =False
hit_count         =0
address           =0

dbg = pydbg()
dbg.set_callback(EXCEPTION_BREAKPOINT, handle_bp)
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, handle_av)

found_target = False
for (pid, proc_name) in dbg.enumerate_processes():
    if proc_name.lower() == "fuzz_server.exe"
        found_target = True
        break

if found_target:
    dbg.attach(pid)
    dbg.bp_set(snapshot_hook)
    dbg.bp_set(restore_hook)
    print "enteringdebugeventloop"
    dbg.debug_event_loop()
else:
```

```
    print "target not found."
```

在上面定义的两个回调函数中，处理访问违例的函数（handle_av）更易于理解，因此我们首先讨论它。我们在一开始就注册这个回调函数，以检测一个潜在的可被利用的状况。该回调函数的代码很简单，很容易被用在其他的 PyDbg 应用中。该回调函数首先从异常记录中获取一些有用的信息，例如触发异常的指令，标识违例是由读还是写导致的一个标记，以及导致异常的内存地址。然后，该函数尝试取得违法指令的反汇编码，打印出异常的实质信息。最后，在终止被调试对象之前，该函数尝试打印发生异常时目标进程的执行上下文。执行上下文包含各个寄存器的值，寄存器值所指向数据的内容（如果寄存器值是指针的话），以及一个可变的栈解引用（dereferences）的数量（在本案例中我们指定为 5）。关于 PyDbg API 规范的更多细节，请参考位于 http://pedram.redhi ve. com/PaiMei/docs/PyDbg/ 处的深入文档。

341

```
def handle_av(pydbg, dbg, context)
    exception_record      =dbg.u.Exception.ExceptionRecord
    exception_address     =exception_record.ExceptionRecord
    write_violation       =exception_record.ExceptionInformation[0]
    violation_address     =exception_record.ExceptionInformation[1]

try:
    disasm = pydbg.disasm(exception_address)
except:
    disasm = "[UNRESOLVED]"
    pass

print "***ACCESS VIOLATION@%08x %s***" %\
    (exception_address,disasm)

if write_violation:
    print "write violation on"
else:
    print "read violation on",

print "%08x" % violation_address

try:
    print pydbg.dump_context(context, 5, False)
except:
    pass

print "terminating debuggee"
```

```
pydbg.terminate_process()
```

另外，我们也可以在其他调试器如 OllyDbg 中捕捉访问违例。为了达成这一目的，必须将你选择的调试器配置为操作系统的“即时”（just-in-time, JIT[21]）调试器。然后，访问违例处理函数的函数体就可以被替换为下面的代码：

342

```
def handle_av(pydbg, dbg, context):
    pydbg.detach()
    return DBG_CONTINUE
```

当真的发生了一个访问违例后，我们熟悉的对话框就会出现，如图 20.14 所示。

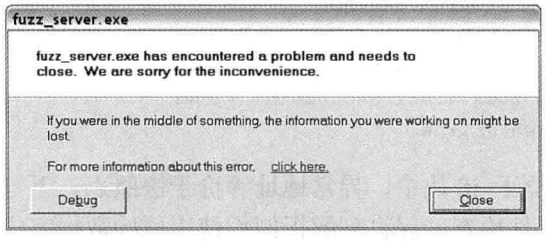


图 20.14　Fuzz_server 挂了

单击对话框的“Debug”按钮会弹出你注册的即时调试器，以供近距离查看目标中究竟出了什么问题。我们内存模糊测试器的最后一个组件是模糊测试器中最重要的部分，也就是断点处理函数。当 fuzz_server 运行到我们以前在快照点和恢复点设置的断点时，断点处理函数就会被调用。接下来的断点处理函数的代码片段是我们目前见到的最难对付的 PyDbg 代码，因此让我们一点一点地对它进行详细分析。在函数的开头我们定义了被访问的全局变量，并将断点发生时的地址保存在 exception_address 中。

```
def handle_bp(pydbg, dbg, context):
    global snapshot_hook, restore_hook
    global snapshot_taken, hit_count, address

    exception_address = \
        dbg.u.Exception.ExceptionRecord.ExceptionAddress
```

343

接下来检查我们是否位于快照点。如果是，递增 hit_count 变量的值并打印出一条信息指示我们当前的位置：

---

21 http://msdn.microsoft.com/library/default.asp?url=/library/en-us/base/debugging_terminology.asp

```
if exception_address == snapshot_hook:
    hit_count += 1
    print "snapshot hook hit#%d\n" % hit_count
```

接下来，我们检查 boolean 标志 snapshot_taken。如果 fuzz_server 以前没有生成过快照，那就通过调用 PyDbg 的 process_snapshot() 例程来生成。为了提供计时信息，我们在快照操作外面包裹了一个定时器，然后更新 snapshot_taken 标志的值为 True。

```
#如果还没有进程快照，生成一个
if not snapshot_taken:
    start = time.time()
    print "taking process snapshot..."
    pydbg.process_snapshot()
    end = time.time() - start
    print "done. took %.03f seconds\n" % end
    snapshot_taken = True
```

注意，我们仍然位于 if 块中，异常地址等价于快照点。下一个代码块负责数据变异。hit_count 上的条件检查确保直到解析例程使用原始数据完成了第一个迭代之后才发生变异。当然，这不是必需的。如果发现以前已经分配过地址（一会儿我们将清楚地看到 address 是从哪儿来的），我们就使用 PyDbg 封装的 virtual_free() 释放它。

```
if hit_count >= 1
    if address:
        print "freeing last chunk at",
        print "%08x" % address
        pydbg.virtual_free(        \
            address,               \
            1000,                  \
            MEM_DECOMMIT)
```

344

我们仍处在 hit_count>=1 的 if 块中，接下来我们使用 PyDbg 封装的 virtual_alloc() 在 fuzz_server 的进程空间中分配一块内存。该内存分配对应刚才看到的内存释放。为什么我们要首先分配内存？因为比起在合适的位置修改传入解析例程的原始数据相比，将变异后的数据放置到 fuzz_server 进程空间的其他地方，然后将指向原始数据的指针修改为指向我们变异后的数据块，要容易得多。这里需要注意的是，在这种方式下，潜在的栈破坏可能会表现为堆破坏，因为存在漏洞的缓冲区可能会被从栈中移出去了。

```
print "allocating memory for mutation"
address = pydbg.virtual_alloc(     \
    None,                          \
```

```
        1000,                          \
        MEM_COMMIT,                \
        PAGE_READWRITE)
    print "allocating at %08x\n" % address
```

我们假定服务器只能解析 ASCII 数据并使用一个简单的数据生成“算法”向分配得到的变异块（变量 mutant）中填充模糊测试数据。从一个全由 A 字符组成的长字符串开始，我们选择一个随机索引值，向字符串的索引位置插入一个随机 ASCII 字符。这非常简单：

```
print "generating mutant..."
fuzz = "A"*750
random_index = random.randint(0, 750)
mutant   = fuzz[0:random_index]
mutant   += chr(random.randint(32, 126))
mutant   += fuzz[random_index:]
mutant   += "\x00"
print "done.\n"
```

接下来，我们使用 PyDbg 的 write_process_memory() 例程将模糊测试数据写入前面分配的内存块中：

```
print "writing mutant to target memory"
pydbg.write_process_memory(address, mutant)
print
```

最后，我们修改函数参数指针，使其指向我们新分配的内存块。回到图 20.11，指向包含了原始解混淆后的数据的缓冲区的指针位于我们当前栈指针偏移为 4 的位置。接下来我们继续执行：

```
print "modifying function argument"
pydbg.write_process_memory(        \
    context.Esp+4,                 \
    pydbg.flip_endian(address))
print

print "continuing execution...\n"
```

回头查看断点处理函数定义的剩余部分，最后是一个到达恢复点时处理快照恢复的 if 语句块。在这个 if 语句块中，通过调用 PyDbg 的 process_restore() API 例程实现快照恢复。为了得到恢复过程执行的时间信息，我们再次把快照恢复过程包裹在定时器中：

```
if exception_address == restore_hook:
    start = time.time()
    print "restoring process snapshot..."
    pydbg.process_restore()
    end = time.time() - start
    print "done. took %0.3f seconds\n" % end
    pydbg.bp_set(restore_hook)

return DBG_CONTINUE
```

好了……现在我们已经准备好进行模糊测试了，首先启动服务器：

```
$./fuzz_server.exe
Listening and waiting for client to connect...
```

接下来，启动内存模糊测试器：

```
$./chapter_20_srm_poc.py
entering debug event loop
```


启动客户端：

```
$./fuzz_client.exe 192.168.197.1 'sending some data'
connecting...
sending...
sent...
```

客户端发送完数据之后，服务器立刻就会运行到我们设置了钩子的快照点，我们的内存模糊测试器就会开始工作：

```
snapshot/mutate hook point hit #1
taking process snapshot...done. took 0.015 seconds
```

我们的内存模糊测试器生成一个快照并继续服务器的执行。fuzz_server 完成数据解析，打印出输出信息，然后想要退出：

```
received 17 bytes
parsing:sending some data
exiting...
```

然而，在 fuzz_server 有机会退出之前，fuzz_server 运行到了我们设置了钩子的恢复点，于是我们的内存模糊测试器再次开工：

```
restoring process snapshot...done. took 0.000 seconds
```

fuzz_server 进程的状态成功地退回到了我们的快照点并继续执行。这一次，由于 hit_count 现在大于 1，断点处理函数中的数据变异块被执行：

```
snapshot/mutate hook point hit#2

allocating chunk of memory to hold mutation
memory allocated at 003c0000

generating mutant...done.
writing mutant into target memory space

modifying function argument to point to mutant

continuting execution...
```

从输出信息中可以看到，模糊测试器分配了内存，生成了一个模糊测试变异，将该变异数据写入了 fuzz_server，并修改了参数指针。当 fuzz_server 恢复执行后，通过它的输出信息，我们能确认第一个快照恢复变异是成功的：

```
parsing:
AAAA
```

```
freeing last chunk at 01930000
allocating chunk of memory to hold mutation
memory allocated at 01940000

generating mutant... done.

writing mutant into target memory space

modifying function argument to point to mutant

continuing execution...

*** ACCESS VIOLATION @41414141 [UNRESOLVED] ***
read violation on 41414141
terminating debuggee
```

348

当我们这个基于 SRM 的基本的概念性模糊测试器执行到第 265 次迭代时，我们在示例的目标应用中发现了一个显然可被利用的漏洞。“ACCESS VIOLATION @41414141”表明进程试图从虚拟地址 0x41414141 读取和执行指令时发生了失败，因为该内存位置是不可读的。0x41 是十六进制表示的字符 A 的 ASCII 值（我想你应该已经猜到了吧）。来自我们的模糊测试器的数据导致了一个溢出并覆写了栈中的返回地址。当导致溢出的函数返回其调用者时，就会发生访问违例，而该访问违例会被我们的模糊测试器捕获。攻击者能够轻松地利用这个漏洞，但不要忘了，攻击者必须使用模糊化例程来对解密数据发起对服务器的攻击（在真实环境中，你可能没法轻易地让客户端发送任意数据）。读者可以对 fuzz_server 的源代码或是二进制文件进行分析，找到这个漏洞的准确原因。这个工作就作为练习留给读者吧。

20.7 小结

在本章中，我们花费了很长的篇幅讨论了一个崭新的、理论性很强的模糊测试方法。该方法的理论非常有趣，值得我们仔细研究该理论的具体应用。然而，我们更鼓励读者下载示例文件，自行运行示例的模糊测试。仅通过书本这种非交互式媒体，难以理解这些例子。

希望最近两章的内容能够激起读者的兴趣，甚至于能够在某些点上帮助读者解决面临的具体问题。PyDbg 平台和本章用到的示例应用都是开源的，读者可以自由地使用。读者可以从 http://www.fuzzing.org 下载这些应用，享受探索之旅。对于我们所有的项目，

我们都希望从读者那里得到项目改进建议，缺陷报告，补丁程序，以及使用情况，这样可以使得这些工具不断得到更新和发展。

虽然本章有一点偏离本书的主要关注点，然而，当你尝试用自动化的方式进行异常检测时，本章包含的调试方面的知识能够帮到你。在第 24 章讨论高级异常检测时，我们将会更深入地讨论进行自动化的异常检测的话题。

第三部分

高级模糊测试技术

第 21 章　模糊测试框架
第 22 章　自动化协议分析
第 23 章　模糊测试器跟踪
第 24 章　智能错误检测

第21章

模糊测试框架

"There's an old saying in Tennessee——I know it's in Taxas, probably in Tennessee——that says, fool me once, shame on——shame on you. Fool me——you can't get fooled again."

——George W. Bush, Nashville, TN, September 17, 2002

目前已有不少专用的模糊测试器能够对常用的，已有文档的网络协议和文件格式进行模糊测试。这些模糊测试器能够彻底测试一个特定协议，并能对支持这种协议的诸多应用进行压力测试。例如，专用于 SMTP 协议的模糊测试器就可以对 E-mail 传输程序，如微软的 Exchange，以及 Sendmail、qmail 等程序进行测试。而另一些“哑”模糊测试器则使用了更通用的方法，能够对任意协议和文件格式进行模糊测试，对其执行简单的、与协议无关的变异，例如位翻转和字节转置。

以上这两类模糊测试器能有效地应用于许多常用应用上，尽管如此，我们还是经常需要对专有协议和未测试过的协议进行更全面彻底的模糊测试。这就是模糊测试框架能够发挥极大作用的领域。

在本章中，我们将探索一些目前可用的开源模糊测试框架，例如 SPIKE，这个一度流行的框架如今已成为了家喻户晓的名字（取决于你的家庭有多极客）。此外，我们还会关注一些令人兴奋的新工具，例如 AutoDafe 和 GFP。在结束对这些现有技术的探究之后，我们将会看到，尽管已有许多通用的模糊测试框架提供了各种支持，有时候我们仍有必要从头开始创建一个模糊测试器。在本章的后半部分，我们将以一个真实的模糊测试问题及其解决方案的开发为例来解释这一点。在本章的最后，我们会介绍一个由本

书作者开发的新框架，并探索该框架的优势。

21.1　什么是模糊测试框架

部分现有的模糊测试框架是用 C 语言开发的，而其他模糊测试框架则使用 Python 和 Ruby。有些框架提供的特性基于其开发语言，而另一些框架使用某种定制语言。例如，Peach 模糊测试框架提供了基于 Python 语言的模糊测试构造器，而 dfuz 工具则实现了它自己的模糊测试对象集（本章后面将对这两个框架进行更详细的讨论）。有些框架对数据生成进行了抽象，而另一些则不然。有些框架采用了面向对象的设计方式，具有良好的文档，而另一些则可能在大多数情况下仅对其创建者有用。尽管存在这些区别，但所有模糊测试框架都具有相同的目标：为模糊测试开发者提供一个快速、灵活、可重用、一致的开发环境。

好的模糊测试框架应该抽象许多枯燥的任务，最小化使用者在枯燥任务上的工作。为了帮助进行模糊测试开始阶段的协议建模，有些框架提供了工具，将抓取到的网络数据转换为框架可理解的格式。这种工具使得研究者可以用工具导入大量的经验数据，而研究者本人可以更专注于适合人工处理的任务上（例如，确定协议中的字段边界）。

对一个全面的框架来说，自动的长度计算绝对是必需的。许多协议使用类似 ASN.1[1] 标准的类型、长度、值（TLV，即 Type、Length、Value）风格的语法。例如，假设传输数据的第一个字节定义了后续数据的类型：0x01 表示纯文本，0x02 表示原始二进制数据格式。随后的两个字节定义了后续数据的长度。剩余部分则定义了值。传输数据的规范如下图所示：

01	00　07	F　U　Z　Z　I　N　G
类型	长度	值

当对该协议的值字段进行模糊测试时，在每个测试用例中我们必须计算并更新这个两字节的长度字段。否则，如果通信数据被检测为违反协议规范，我们的测试用例就面临着被立即扔掉的危险。一个有用的框架应该包含循环冗余校验（Calculating Cyclic Redundancy Check，CRC）[2]计算方法和其他校验算法。CRC 值通常用于在文件规范和

[1] http://en.wikipedia.org/wiki/Asn.1

[2] http://en.wikipedia.org/wiki/Cyclic_redundancy_check

协议规范中识别可能被破坏的数据。例如，PNG 图像文件使用了 CRC 值，当程序发现图像文件的 CRC 值与计算得到的值不一致时，程序就避免对图像文件进行处理。CRC 是一个重要的安全相关的功能，但如果对一个协议进行变异时没有正确地更新 CRC 值，CRC 就会阻止模糊测试的顺利进行。分布网络协议（Distributed Network Protocol，DNP3）[3]规范是一个更极端的例子，该规范用于数据采集与监视控制（Supervisory Control and Data Acquisition，SCADA）系统中。在该协议中，数据流被切分成 250 字节大小的块，每一块都前置了一个 CRC-16 校验值！最后，传输数据中经常会包含客户端、服务器的 IP 地址，而这两个 IP 地址在进行模糊测试的过程中经常会发生变化。如果框架提供了方法来自动确定 CRC 值，并将这些值包含在生成的模糊测试用例中，将会大大方便框架使用者。

大多数（如果不是全部的话）框架提供了生成伪随机数据的方法。好的框架甚至会更进一步，包含一个启发式攻击的列表。启发式攻击是已知的会导致软件错误的数据序列。例如，格式字符串（%n%n%n）和目录遍历（../../../）序列就是简单的启发式攻击的例子。在许多场景中，在使用随机数据进行测试之前，先遍历一个包含启发式攻击测试用例的有限长度的列表能够节省时间，而且，这是个值得继续深入研究的方向。

错误检测在模糊测试中扮演着重要角色，在第 24 章“智能错误检测”中我们将对此进行详细讨论。在最简单的层面上，如果目标应用不能接受新连接，模糊测试器能够检测到它的目标可能已经出错了。更高级的错误检测通常需要借助调试器的帮助。高级的模糊测试框架应该允许模糊测试器直接与附着在目标应用上的调试器通信，甚至是自带定制的调试器。

根据你个人的偏好，可能会有一个长长的次要特性列表，列表中的功能能够显著提升你的模糊测试器开发体验。例如，有些框架支持对各种数据格式进行解析。另外，当将原始的字节数据复制粘贴到模糊测试脚本中时，可以将十六进制数据粘贴为下面这些格式：0x41 0x42、\x41 \x42、4142，这会带来很大的便利。

模糊测试度量（参见第 23 章“模糊测试器跟踪”）目前也受到了一些小小的关注。高级模糊测试框架可以包含一个与度量数据收集工具（如代码覆盖监视器）通信的接口。

354

最后，理想的模糊测试框架将会提供最大化代码复用的基础设施，这些基础设施允许开发出来的组件能被将来的项目所使用。如果这些基础设施得到正确实现，模糊测试

[3] http://www.dnp.org/

器就能够得到“进化”，被使用得越多就越“聪明”。接下来，在探究设计与创建一个同时面向任务与通用目的的定制框架之前，我们先来探索一些已有的模糊测试框架。在探索这些框架时，请读者记住以上描述的概念。

21.2 现有的模糊测试框架

在本节中，我们将仔细研究一些模糊测试框架，理解这些已存在的框架。当然，我们无法覆盖所有现存的模糊测试框架，但我们会检查已有框架的一个样本集，样本集中的框架展示了各种不同方法。下面按照成熟度和功能的丰富程度列出了这些框架，从最原始的框架开始。

21.2.1 antiparser[4]

antiparser 框架以 Python 语言编写，是一个专门帮助模糊测试器创建随机数据的 API。该框架可以用来开发能够跨平台运行的模糊测试器，因为这个框架仅要求可用的 Python 解释器。使用该框架的方式非常直接：首先创建一个 antiparser 类的实例，antiparser 类起到容器的作用。然后，antiparser 提供了一些模糊测试类型，这些类型能够被实例化并添加到容器中。以下这些是可用的模糊测试类型。

- apChar()：8 位的 C 语言字符。
- apCString()：C 风格的字符串；也即，一个以空字符结尾的字符数组。
- apKeywords()：一个值列表。每个值都包含分隔符、数据块和终止符。
- apLong()：32 位 C 语言整型。
- apShort()：16 位 C 语言整型。
- apString()：自由格式的字符串。

在所有这些可用的数据类型中，apKeywords() 是最有趣的一个。这个类定义了一个关键字列表、一个数据块、一个关键字和数据之间的分隔符，以及一个可选的数据块终止符。类所生成数据的格式是“[关键字] [分隔符] [数据块] [终止符]”。

355

antiparser 的发布版包含一个示例脚本 evilftpclient.py，这个脚本使用了 apKeywords() 数据类型。我们可以检查脚本中相关的部分，以更好地理解如何在这个

[4] http://antiparser.sourceforge.net/

框架上进行开发。下面展示的代码来自 evilftpclient.py 脚本，这部分代码在脚本中负责测试一个 FTP 后台，检查解析 FTP 动词参数时是否存在格式字符串漏洞。

```
from antiparser import *

CMDLIST = ['ABOR',   'ALLO',     'APPLE',     'CDUP',     'XCUP',    'CWD',
           'XCWD',   'DELE',     'HELP',      'LIST',     'MKD',     'XMKD',
           'MACB',   'MODE',     'MTMD',      'NLST',     'NOOP',    'PASS',
           'PASV',   'PORT',     'PWD',       'XPWD',     'QUIT',    'REIN',
           'RETE',   'RMD',      'XRMD',      'REST',     'RNFR',    'RNTO',
           'SITE',   'SIZE',     'STAT',      'STOR',     'STRU',    'STOU',
           'SYST',   'TYPE',     'USER']

SEPARATOR    = " "
TERMINATOR   = "\r\n"

for cmd in CMDLIST:
    ap = antiparser()
    cmdkw = apKeywords()
    cmdkw.setKeywords([cmd])
    cmdkw.setSeparator(SEPARATOR)
    cmdkw.setTernimator(TERMINATOR)

    cmdkw.setContent(r"%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n")
    cmdkw.setMode('incremental')
    cmdkw.setMaxSize(65536)
    ap.append(cmdkw)

    sock = apSoket()
    sock.connect(HOST, PORT)

    # 打印 FTP 后台 banner
    print sock.recv(1024)

    # 发送模糊测试
    sock.sendTCP(ap.getPayload())

    # 打印 FTP 后台响应
    print sock.recv(1024)
    sock.close()
```


在代码的开头部分，我们导入了 antiparser 框架中所有可用的数据类型和类，定义

了一个 FTP 动词列表，定义了动词参数分隔字符（空格）和命令终止序列（回车换行）。接下来的代码迭代测试每个 FTP 动词，在每个迭代中，我们首先实例化一个新的 antiparser 容器类。接下来，apKeywords() 数据类型开始发挥作用。接下来的代码指定包含当前被测试的命令动词的列表作为关键字集合（在本例中关键字集合中只包含一个关键字）。随后的代码定义了恰当的动词参数分隔符和命令终止符。然后，apKeywords() 对象（代码中的变量 cmdkw）的数据内容被设置成一个格式字符串标志的序列。如果目标 FTP 服务器在解析动词参数时存在漏洞，设置在 apKeyword() 对象中的数据内容一定会触发该漏洞。

接下来是两个函数调用：setMode('incremental') 和 setMaxSize(65536)，指明在数据变异时数据块应该逐步增长到最大值 65,536。然而，在这个具体的例子中，这两个调用没有什么作用，因为模糊测试器并没有通过调用 ap.permute() 来循环遍历测试用例或排列。相反，对每个动词，这个模糊测试器使用一个单独的数据块进行测试。

剩余部分的代码行基本上是自解释的。剩下部分的代码将数据类型 apKeywords() 添加到 antiparser 容器，并创建了一个套接字。建立连接后，调用 ap.getPayload() 生成测试用例，并通过 sock.sendTCP() 向测试目标发送测试用例。

显然，antiparser 存在不少局限。实际上，即使不借助 antiparser 框架的帮助，直接使用 Python 也能够轻易地创建出以上展示的这个 FTP 模糊测试器。与使用其他框架相比，antiparser 框架相关的代码所占模糊测试器的比例明显要小一些。另外，antiparser 框架缺乏许多在上一节中列出的我们期望的自动化功能，例如自动计算和表示普通 TLV 协议格式的能力。最后，这个框架的文档很少，而且不幸的是，发布于 2005 年 8 月的该框架的 2.0 版本中只有一个可用的例子。虽然这个框架简单，而且确实能够帮助生成简单的模糊测试器，但它并不适合用来处理复杂的任务。

21.2.2　Dfuz[5]

357

Dfuz 模糊测试框架是由 Diego Bauche 用 C 语言开发的，处于活跃的维护中，经常发布更新。该框架已经帮助发现了多种漏洞，这些漏洞覆盖微软、IPswitch 和 RealNetworks 等公司的产品。Dfuz 框架是开源的，可以从互联网下载得到。但该框架

[5] http://www.genexx.org/dfuz/

的源代码采用了一种严格的开源许可，在未得到作者允许的情况下，任何人不得复制或修改该框架的源代码。根据你的需求，这个受限的许可可能会让你不想使用这个框架。框架作者之所以采用这个许可背后的原因似乎是对他自己的代码质量不满意（根据README文件中的描述）。如果你想要复用他的一部分代码，可能只能直接联系作者。Dfuz被设计为运行在UNIX/Linux平台上，提供了一种定制的编程语言来开发新的模糊测试器。Dfuz框架不是我们本章研究的最高级的模糊测试框架，然而，它的设计方式简单且直观，这使得它成为了一个好的研究框架设计的案例，因此下面我们将对其进行更加深入的研究。

构成Dfuz的基础组件包括数据、函数、列表、选项、协议及变量。这些不同的组件被用来定义成规则集，模糊测试引擎可以解析这些规则集，并根据规则集生成和传输数据。在规则文件中，Dfuz框架使用下面这种读者熟悉的简单语法来定义变量：

```
var my_variable   = my_data
var ref_other     = "1234", $my_variable,0x00
```

变量通过前缀var定义，通过在变量名前加上美元符号（$符号，与Perl或PHP中相同），可以在代码中的其他位置引用该变量。模糊测试器创建是完全自包含的。这意味着与antiparser不同，完全基于它提供的定制语言就可以在该框架之上建立一个模糊测试器。

Dfuz定义了多个函数来完成需要经常完成的任务。这些函数很容易被识别出来，因为它们的名字都以百分号字符（%）开头。Dfuz框架定义了以下这些函数。

- %attach()：在按下回车键之前一直等待。该函数用于暂停模糊测试器以完成其他任务。例如，如果你的模糊测试目标启动新线程来处理进入的连接，而你希望在新线程上附着一个调试器，那么，在初始连接建立后插入一个%attach()调用，然后定位到新线程，并将调试器附着到目标线程上。
- %length()或%calc()：计算所提供参数的大小，并以二进制数据格式插入该值。例如，%length("AAAA")将会把二进制值0x04插入二进制流中。默认情况下该函数的输出为32位长度，但可以通过调用%length:unit8()将其设置为8位，或调用%length:uint16()将其设置为16位。
- %put:<size>(number)：将给定的数插入二进制流中。size可以被设置为byte（字节）、word（字）或dword（双字）之一。这三个值分别是uint8、uint16和uint32的别名。

- %random:<size>() ：生成并插入一个随机的给定长度的二进制值。类似于%put()，size可以被指定为byte、word或dword之一。
- %random:data(<length>,<type>)：生成并插入随机数据。length指定要生成数据的字节数。type指定要生成的随机数据的种类，可以是ascii（ASCⅡ字符）、alphanum（数字）、plain（文本）或graph（图像）中的一种。
- %dec2str(num)：将一个十进制数据转换成字符串，并将其插入二进制流中。例如，%dec2str(123) 生成字符串“123”。
- %fry()：随机修改以前定义的数据。例如，规则“AAAA”,%fry() 使得字符串“AAAA”中随机多个字符被替换成随机的字节值。
- %str2bin()：将多种表示十六进制的字符串解析为对应的十六进制数据。例如，4141、41 41及41-41都会被翻译成“AA”。

Dfuz中可以用多种方式表示数据。Dfuz中的定制脚本语言支持指定字符串、原始字节、地址、重复数据及基础数据循环。使用逗号分隔符可以将多种数据定义串接在一起，组成一个简单列表。下面的例子演示了多种定义数据的方法（更多细节请参考文档）：

```
var my_variable1 = "a string"
var my_variable2 = 0x41, |0xdeadbeef| , [Px50] , [\x41*200] , 100
```

使用list关键字可以定义列表。列表的声明格式如下：首先是关键字list，然后是列表名称，接下来是关键字begin，接着是以新行分隔的数据值列表，最后以关键字end结束。列表用来定义和索引一个数据序列。例如：

```
list my_list:
begin
    some_data
    more_data
    even_more_data
end
```

359

与变量一样，在列表名前加上美元符号（$）后，就可以从其他位置引用该列表。和Perl、PHP一样，通过方括号（[ ]）可以索引列表成员。rand关键字用于支持在列表中进行随机索引，用法为：$my_list[rand]。

以下选项用于控制Dfuz引擎的全局行为。

- keep_connecting：即使无法连接到目标也继续进行模糊测试。
- big_endian：修改生成数据的字节顺序为大端（big endian）模式，默认情况下

是小端（little endian）模式

- little_endian：修改生成数据的字节顺序为小端（little endian）模式，小端模式是默认值。
- tcp：指定通过 TCP 建立套接字连接。
- udp：指定通过 UDP 建立套接字连接。
- client_side：指定引擎以客户端方式工作，对服务器进行模糊测试。
- server_side：指定引擎以服务端方式工作，对客户端进行模糊测试。
- use_stdout：生成数据到标准输出（控制台），而不是生成到一个通过套接字连接的对端。该选项要求与 host 值为“stdout”的选项一起使用。

为了便于对常见的协议进行模糊测试，Dfuz 提供了对 FTP、POP3、Telnet 及服务器信息块（Server Message Block，SMB）协议的支持。Dfuz 通过 ftp:user() 、ftp:pass() 、ftp:mkd() 、pop3:user() 、pop3:pass() 、pop3:dele() 、telnet:user() 、telnet:pass() 等函数支持这些协议。从 Dfuz 的文档中可以找到 Dfuz 提供的支持协议的完整函数列表。

Dfuz 提供的基础组件必须与一些附加的指令组合起来创建规则文件。下面是一个简单的例子，该例子是一个用于对 FTP 服务器进行模糊测试的规则文件（框架自带的例子）：

360

```
port=21/tcp

peer write: @ftp:user("user")
peer read
peer write: @ftp:pass("pass")
peer read
peer write: "CWD /", %random:data(1024,alphanum), 0x0a
peer read
peer write: @ftp:quit()
peer read

repeat=1024
wait=1
# No Options
```

规则文件中的第一条指令指示引擎通过 TCP 协议连接端口 21。由于没有为该指令指定选项，因此它默认以客户端模式工作。peer read 和 peer write 指令分别指示引擎等待读取数据和向模糊测试目标发送数据。在上面的规则文件中，前两对 peer read 和 peer write 指令用于完成在目标 FTP 服务器上进行认证。接下来，规则文件构建出

CWD 命令（更改工作目录命令），并将其发送给服务器。CWD 命令带有一个由 1024 个随机字母组成、以换行符（0x0a）结尾的字符串参数。然后，规则文件指示关闭连接。规则文件最后的 repeat 指令指示将该指令上方的 peer read 和 peer write 指令块执行 1024 次。在每个测试用例中，Dfuz 会建立一个到 FTP 服务器的认证后的连接，发送一个 CWD 命令，该命令带一个长 1024 字节的随机字母字符串作为参数，然后断开连接。

Dfuz 是一个简单且功能强大的模糊测试框架，可以用来表示和对许多协议及文件格式进行模糊测试。该框架具有 stdout（标准输出）支持，提供一些基本的命令行脚本，因此可以变身为文件格式、环境变量或命令行参数模糊测试器。Dfuz 的学习曲线比较平坦，开发效率也较高。Dfuz 要求使用者完全使用框架提供的脚本语言来开发模糊测试器是一把双刃剑：其好处在于非程序员也能使用框架提供的脚本语言来描述协议，对其进行模糊测试；不利之处则是有经验的程序员没法利用成熟的编程语言固有的功能与威力。Dfuz 框架提供了一些代码复用，但是并不像其他框架，如 Peach 那样多。目前 Dfuz 框架缺少的一个关键的功能是可用的启发式攻击智能列表。总体来说，Dfuz 是一个良好设计的模糊测试框架的实例，是一个可以随身携带的好工具。

21.2.3　SPIKE[6]

361

SPIKE 由 Dave Aitel 编写，也许是被最广泛使用和最知名的模糊测试框架。SPIKE 用 C 语言编写，提供了一套允许快速和高效开发网络协议模糊测试器的 API。SPIKE 在灵活的 GNU 的 GPL[7]许可下开源。正是因为 GPL 这个良好的许可，才有了 SPIKEfile 工具，该工具修改了 SPIKE 工具，使其成为了一个专门用来进行文件格式模糊测试的 SPIKE 版本（参见第 12 章“UNIX 平台上的文件格式自动化模糊测试”）。SPIKE 使用了一种新技术表示网络协议，并对其进行模糊测试。在 SPIKE 中，协议数据结构被分解并表示为块，也叫做 SPIKE，这些块同时包含了二进制数据和块大小。基于块的协议表示方法允许在各种协议层上进行抽象，并能自动计算数据尺寸。要更好地理解块的概念，请参考下面这个来自白皮书“安全性测试中基于块的协议分析的优势”[8]中的简单例子：

```
s_block_size_binary_bigendian_word("somepacketdata");
```

[6] http://www.immunitysec.com/resoruces-freesoftware.shtml

[7] http://www.gnu.org/copyleft/gpl.html

[8] http://www.immunitysec.com/downloads/advantage_of_block_based_analysis.pdf

```
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

这个基础的SPIKE脚本（SPIKE脚本用C语言编写）定义了一个名为somepacketdata的块，将4个字节的数据0x01020304压入块中，并将块的长度放置在块的前部。在这个例子中，somepacketdata块的大小为4，并保存在一个以大端方式保存的字中。注意，大多数SPIKE API都带有s_或spike_前缀。s_binary() API用于向块中加入二进制数据，该API具有非常灵活的参数格式，能够处理各种通过复制粘贴方式得到的输入，该API能够处理各种字符串格式，如4141 \x41 0x41 41 00 41 00等。这个例子虽然简单，但其演示了基础而全面的构建一个SPIKE的方法。SPIKE允许在块中嵌套其他的块，因此在SPIKE中，即使是复杂协议，也可以轻松地通过块分解为最小的原子。下面是对上一个例子的扩展：

362

```
s_block_size_binary_begendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_blocksize_halfword_bigendian("innerdata");
s_block_start("innerdata")
s_binary("00 01");
s_binary_bigendian_word_variable(0x02);
s_string_variable("SELECT");
s_block_end("innerdata");
s_block_end("somepacketdata");
```

在这个例子中，定义了两个块：somepacketdata和innerdata。后一个块包含在前一个块中，每个块都有一个描述块尺寸的前缀。新增的innerdata块以一个静态的双字节值0x0001开始，然后是一个默认值为0x02的四字节可变整数，最后是一个默认值为SELECT的字符串变量。s_binary_bigendian_word_variable()和s_string_variable() API会分别在一个预定义的整数集合和字符串变量集合（启发式攻击）中循环，启发式攻击的列表已经在本书前面的章节中进行了讨论。在处理这个例子的脚本时，SPIKE会首先进行可能的字变量变异，然后进行字符串变量变异。SPIKE框架的真正强大之处在于，SPIKE会在每次变量发生变异时自动地更新尺寸字段的值。如果读者想要查看或扩展模糊测试变量的当前列表，可以查看SPIKE/src/spike.c文件。SPIKE框架的版本2.9包含了一个大约有700个可能触发错误的启发式列表。

在通过前面的例子展示了如何使用SPIKE的基本概念后，我们来看看如何在这个框架中对复杂的协议进行建模。除了上面介绍的基本概念外，SPIKE中还有许多其他的

API 和自带的例子。SPIKE 的文档描述了这些更进一步的信息。继续以 FTP 协议的模糊测试为例，下面的这段代码来自随 SPIKE 一起发布的一个 FTP 模糊测试器，是一个可运行的例子。从展示 SPIKE 能力的角度来说，这个例子并不是最好的，因为这个例子中并没有使用块，但这个例子适合用来帮助比较 SPIKE 框架和其他框架。

```
s_string("HOST ");
s_string_variable("10.20.30.40");
s_string("\r\n");

s_string_varibale("USER");
s_string(" v");
s_string_varibale("bob");
s_string("\r\n");
s_string("PASS ");
s_string_variable("bob");
s_string("\r\n");
s_string("SITE ");
s_string_variable("SEDV");
s_string("\r\n");

s_string("ACCT ");
s_string_variable("bob");
s_string("\r\n");

s_string("CWD ");
s_string_variable(".");
s_string("\r\n");

s_string("SMNT  ");
s_string_variable(".");
s_string("\r\n");

s_string("PORT ");
s_string_variable("1");
s_string(",");
s_string_variable("2");
s_string(",");
s_string_variable("3");
s_string(",");
s_string_variable("4");
s_string(",");
s_string_variable("5");
```

```
s_string(",");
s_string_variable("6");
s_string(",");
```

SPIKE 只有些零星的文档，而且其发布包中包含一些已废弃的，可能会让用户感到困惑的组件。然而，我们仍然可以找到许多能够工作的样例，这些样例能够帮助我们熟悉这个功能强大的模糊测试框架。SPIKE 框架缺乏全面的文档，发布包的组织方式也很糟糕，以至于有些研究者猜测 SPIKE 是故意把这些方面弄得不完整，以防止其他人发现仅由作者本人发现的漏洞。当然，这个猜测的真实性仍然没有得到证实。

对需要在 Windows 平台上进行模糊测试的用户来说，缺乏对微软的 Windows 的支持是 SPIKE 框架的一个主要问题。虽然有些报告说通过 Cygwin[9]可以让 SPIKE 工作在 Windows 平台上，但 SPIKE 是为在 UNIX 环境中运行而设计的。另一个需要考虑的事实是，即使只在 SPIKE 框架中进行很小的修改，例如增加新的模糊字符串，也需要重新编译该框架。最后一个问题是，如果想要在使用该框架开发的模糊测试器之间进行代码重用，只能通过手工复制粘贴代码来完成。在 SPIKE 框架中不能简单地定义一个新元素（如面向 E-mail 地址的模糊测试器）并跨框架地对其进行全局引用。

364

总体来说，SPIKE 是一个有效的模糊测试框架，已被它的作者和其他一些研究者用来发现了多个影响巨大的漏洞。SPIKE 框架还带有例如代理服务器之类的工具，允许研究者监视浏览器和 Web 应用之间的通信，并对其进行模糊测试。SPIKE 诱导故障发生的能力大大帮助创建了模糊测试的新价值。从 SPIKE 公开发布之日起，其采用的基于块的模糊测试方法就得到了广泛的理解，许多模糊测试框架都已经采用了这种技术。

21.2.4 Peach[10]

Peach 工具由 IOACTIVE 发布，是一个用 Python 编写的跨平台模糊测试框架，最早发布于 2004 年。Peach 是一个开源的框架，采用了开放许可。与其他模糊测试框架相比，Peach 可能具有最灵活的架构和最大的代码复用的可能。而且，在作者看来，这个框架拥有一个最有趣的名字（peach，fuzz ——懂了吗？[11]）。Peach 框架提供了一些基本组件用于构建新的模糊测试器，这些基本组件包括生成器（generators）、转换器（transformers）、协议（protocols）、发布器（publishers）和组（groups）。

---

[9] http://www.cygwin.com/

[10] http://peachfuzz.sourceforge.net

[11] 译者注：peach 指桃子，fuzz 有桃子上的绒毛的意思。

生成器负责生成数据，从简单字符串到复杂的分层二进制信息。通过把生成器链接在一起，能够简化复杂数据类型的生成。将数据生成抽象到生成器对象中允许在已实现的模糊测试器之间进行代码复用。例如，在对 SMTP 服务器进行审计时，我们开发了一个 E-mail 地址生成器。那么，这个生成器可以在另一个需要生成 E-mail 地址的模糊测试器中被透明地复用。

转换器以特定方式改变数据。转换器的例子包括 base64 编码器、gzip 编码器和 HTML 编码器。转换器也可以被链接起来，并能绑定到生成器上。例如，生成的 E-mail 地址可以被传给 URL 编码转换器，转换得到的结果可以被再次传给一个 gzip 转换器。将数据转换抽象到转换器对象中允许在已实现的模糊测试器之间进行代码复用。一旦实现了某个给定的转换器，该转换器就能够被所有将来所开发的模糊测试器透明地复用。

发布器实现通过协议传输已生成数据的传输形式。发布器的例子包括文件发布器和 TCP 发布器。同样，将发布这个概念抽象到发布器的对象提升了代码复用。虽然在 Peach 的当前版本中还不支持这类复用，但最终发布器要能够允许与其他发布器进行透明的交互。例如，当你创建了一个 GIF 图像生成器后，这个生成器就应该能够通过简单地更换发布器，将生成的内容发布到文件或发送到一个 Web 表单中。

组包含一个或多个生成器，是逐个访问生成器生成内容的机制。Peach 中包含了几个常用的组的实现。Script 对象是一个附加的组件，该组件实现了简单的抽象，提供了 group.next() 和 protocol.step() 方法，能够减少实现数据遍历所需的冗余代码。

下面是一个全面但简单的例子，这个例子基于 Peach 框架，用来从一个字典文件中暴力破解某 FTP 用户的口令：

```
from Peach                  import *
from Peach.Transformers     import *
from Peach.Generators       import *
from Peach.Protocols        import *
from Peach.Publishers       import *

loginGroup = group.Group()
loginBlock = block.Block()
loginBlock.setGenerators((
    static.Static("USER username\r\nPASS "),
    dictionary.Dictionary(loginGroup, "dict.txt"),
    static.Static("\r\nQUIT\r\n")
    ))
```

```
loginProt = null.NullStdout(ftp.BasicFtp('127.0.0.1', 21), loginBlock)

script.Script(loginProt, loginGroup, 0.25).go()
```

这个模糊测试器首先导入了 Peach 框架的各种组件。接下来，代码生成了一个块和组的实例。块 loginBlock 的定义是首先向 FTP 服务器发送用户名和 PASS 命令，接下来，块导入一个包含可能的密码的字典，块中的这个元素会在模糊测试中迭代作用。块的最后一个元素完成了 PASS 命令，并向 FTP 服务器发送一个 FTP 退出命令，断开与服务器的连接。接下来的代码基于已存在的 FTP 协议，扩展定义了一个新的 FTP 协议。最后一行代码创建了一个脚本对象来执行连接，并在字典中循环迭代。在浏览了脚本之后，你的第一个体会可能是这个框架的交互方式不是那么直观。“不直观”可能是 Peach 框架最大的不足。对测试者来说，选择使用 Peach 开发第一个模糊测试器显然会比使用 Autodafe 或 Dfuz 要花更多时间。

366

Peach 体系结构允许研究者聚焦于一个个的特定协议的子组件，然后组合它们来创建完整的模糊测试器。这种开发方法可能不如基于块的开发方法快速，但其对代码复用的支持却比其他模糊测试框架都要好。例如，如果在测试一个反病毒解决方案时我们开发了一个 gzip 转换器，那么当我们稍后测试一个 HTTP 服务器时，就可以直接使用这个 gzip 转换器处理压缩后的数据。这是 Peach 框架出色的方面。对 Peach 来说，你用得越多，它就会变得越聪明。由于 Peach 是纯粹基于 Python 实现的，因此基于 Peach 创建的模糊测试器能够在任何支持 Python 的环境下运行。此外，利用已存在的 Python 接口、微软的 COM[12]接口或微软的.NET 包，可以使用 Peach 对 ActiveX 控件和托管代码直接进行模糊测试。Peach 的发布包提供了直接对微软 Windows DLL 进行模糊测试，以及将 Peach 嵌入 C/C++代码来创建插装客户端和服务器的例子。

Peach 处于活跃的开发中，到本书出版时它的最新版本是 0.5（发布于 2006 年 4 月）。不过，虽然 Peach 在理论上非常领先，不幸的是它缺乏全面的文档，因此没有得到广泛的应用。缺乏参考资料导致了 Peach 的学习曲线陡峭，这可能会导致你不愿意考虑这个框架。Peach 的作者在这个框架中引入了一些新奇的想法，并创建了一个可被扩充的强壮的基础。最后要提到的是，已经有人宣布了一个 Peach 框架的 Ruby 移植版本，不过，到本书写作时还没有关于这个新版本的进一步消息。

---

12 http://en.wikipedia.org/wiki/Component_Object_Model

21.2.5 通用目的模糊测试器[13]

通用目的模糊测试器（General Purpose Fuzzer，GPF）由 Applied Security 的 Jared DeMott 发布，其名称是常见术语“一般保护错误（General Protection Fault）”的双关语。通用目的模糊测试器项目处于活跃的维护中，基于 GPL 许可方式开源，设计为在 UNIX 平台上运行。正如其名字所暗示的，通用目的模糊测试器被设计为一个通用的模糊测试器。与 SPIKE 不同，通用目的模糊测试器能够生成无数个变异。我们并不是说生成式的模糊测试器比启发式模糊测试器先进，无论是生成式方法还是启发式方法都各有优缺点。从本节我们对通用目的模糊测试器的描述可以看到，相比其他框架，该框架的主要优点是可以用很低的成本建立并运行一个模糊测试器。通用目的模糊测试器通过多种模式对外提供功能，包括纯随机模式（PureFuzz）、转换模式（Convert）、通用模糊（GPF，也是该框架的主要模式）、模式模糊（Pattern Fuzz）及超级通用模糊（SuperGFP）。

纯随机模式是一个易用的纯随机的模糊测试器，其实现方式类似于在一个套接字上使用/dev/urandom。虽然生成的输入空间是一个非智能且无限的空间，但这种技术确实曾经发现过漏洞，甚至在常见的企业软件中也发现过漏洞。与简单地使用 netcat 和 /dev/urandom 的组合相比，纯随机模式的主要优点是其提供了一个“随机种子”选项，该选项允许实现伪随机的数据流的重放。

此外，在成功应用纯随机模式的场合中，range 选项是找到导致异常的数据包的有效方法。

随机也可以高效！

许多人想当然地认为像 GPF 的 PreFuzz 这种只是生成随机数据的模糊测试器过于简单，无法实际发挥作用。为了消除这个通常的错误观念，让我们看看下面这个来自 Computer Associates 的 BrightStor ARCserver 备份方案中的实际案例。2005 年 8 月，在负责处理微软 SQL Server 备份的 agent 中发现了一个严重的可被利用的栈缓冲区溢出[14]。被影响的后台进程在 TCP 端口 6070 上监听，只要向该端口发送长度超过 3168 个字节的数据，就会触发这个漏洞。

基于随机方式的模糊测试可以在几乎不需要设置和无须进行协议分析的情况下轻

[13] http://www.appliedsec.com/resources.html

[14] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=287

易发现该漏洞。这个实例证明，在构建一个智能模糊测试器的同时，运行类似 PureFuzz 这样的模糊测试器是值得的。

转换模式是通用目的模糊测试器自带的一个工具，用来将 Ethereal[15]和 Wireshark[16]等工具生成的 libpcap 文件翻译成 GPF 文件。在协议建模的初始阶段，可以使用这个工具将二进制 pcap（包抓取）格式转换成人工可读和便于修改的基于文本的格式，从而减少一些枯燥的手工工作。

通用目的模糊测试器的主要模式（即通用模糊）提供了通过 GPF 文件和不同的命令行选项来控制多种基础的协议攻击的方式。通用模糊以各种方式对抓取到的数据的某些部分进行变异，然后回放变异后的数据。通用模糊采用的变异方法包括插入逐渐增大的字符串序列和格式字符串、字节循环及随机变异等。这种模糊测试模式是手工密集型的，其大部分逻辑建立在分析者的直觉之上。

368

模式模糊（Partern Fuzz，PF）是通用目的模糊测试器中最著名的模式，因为该模式能够自动地分析协议中的纯文本部分，并自动地对其进行模糊测试。模式模糊检查目标协议中常用的 ASCII 边界和字段终止符，并依据内置的规则对找到的字段自动地进行模糊测试。模式模糊的内部规则以 C 语言代码定义，称为 tokAids。模式模糊包含一个 ASCII 变异引擎，该引擎被定义为 tokAid（名为 normal_ascii）。此外，模式模糊中还包含一些其他的 tokAid（例如 DNS）。为了精确、智能地对定制的二进制协议进行建模和模糊测试，我们必须编写和编译相应的 tokAid。

超级通用模糊（SuperGFP）是通用模糊的 Perl 脚本封装，用于对付这种情况：给定了通过套接字访问的某个服务器作为模糊测试目标，但研究者完全不清楚该从哪里开始。超级通用模糊能够通过利用通用模糊抓取到的内容和一个包含了有效协议命令的文本文件，生成数千个新的抓取内容文件。随后，超级通用模糊脚本启动多个通用模糊的实例，以多种模糊测试模式，使用这些大量的生成数据轰击被测目标。超级通用模糊仅能用于对 ASCII 协议进行模糊测试。

为了与前面展示的 FTP 模糊测试器进行比较，我们提供了一个通用目的模糊测试器的脚本：

---

15 http://www.ethereal.com

16 http://www.wireshark.org

```
Source:S size:20 Data:220 (vSFTPd 1.1.3)
Source:C size:12 Data:USER fuzzy
Source:S size:34 Data:331 Please specofy the password.
Source:C size:12 Data:PASS wuzzy
Source:S size:33 Data:230 Login successful. Have fun.
Source:C size:6 Data:QUIT
Source:S Size:14 Data:221 Goodbye.
```

使用通用目的模糊测试器的最大缺点是复杂。使用通用目的模糊测试器的学习曲线较为陡峭。使用 tokAids 对私有的二进制协议进行建模不像其他方式（例如 SPIKE 的基于块的方式）那么简单，而且，创建的 tokAids 模型需要经过编译后才能使用。最后，严重依赖于命令行选项使得实际需要运行的命令行非常笨重（见下面的命令行例子）：

```
GPF ftp.gpf client localhost 21 ? TCP 8973987234 100000 0 + 6 6 100 100 5000 43
finish 0 3 auto none -G b
```

总体而言，通用目的模糊测试器是一个有价值的工具，它既灵活，又可以被扩展。它提供的各种模式允许研究者在设置更智能的攻击之前快速开始模糊测试。通用目的模糊测试器具有强大自动处理 ASCⅡ协议并对其进行模糊测试的能力，这一点在我们研究的所有框架中表现尤为突出。在写作本书时，通用目的模糊测试器的作者正在为该框架开发一个新的模糊测试模式，该模式利用革命性的计算基础来自动侦测未知协议，并能智能地对其进行模糊测试。在第 22 章“自动化协议分析”中我们将深入讨论这种高级的模糊测试方法。

369

21.2.6 Autodafé[17]

我们可以把 Autodafé 简单地描述为下一代的 SPIKE，该框架能够用于对网络协议和文件格式进行模糊测试。Autodafe 由 Martin Vuagnoux 发布，基于灵活的 GNU GPL 许可，被设计为运行在 UNIX 平台上。与 SPIKE 类似，Autodafé 的核心模糊测试引擎采用了基于块的方法为协议建模。下面引自白皮书“Autodafé，对软件的折磨”[18]的代码片段演示了 Autodafé 使用的基于块的语言，SPIKE 的用户应该会觉得这段代码看上去很眼熟。

```
string("dummy");                    /* 定义一个常量字符串 */
string_uni("dummy");                /* 定义一个常量 unicode 字符串 */
```

[17] http://autodafe.sourceforge.net

[18] http://autodafe.sourceforge.net/docs/autodafe.pdf

```
hex(0x0a 0a \x0a);              /* 定义一个十六进制的常量 */
block_begin("block");           /* 定义块的开头 */
block_end("blick");             /* 定义块的结尾 */
block_size_b32("block");        /* 32 位大端块大小 */
block_size_l32("block");        /* 32 位小端块大小 */
block_size_b16("block");        /* 16 位大端块大小 */
block_size_l16("block");        /* 16 位小端块大小 */
block_size_8("block");          /* 8 位大端块大小 */
block_size_8("block");          /* 8 位大端块大小 */
block_size_hex_string("a");     /* 块的十六进制字符串大小 */
block_size_dec_string("a");     /* 块的 10 进制字符串大小 */
block_crc32_b("block");         /* 大端块的 crc32 */
block_crc32_l("block);          /* 小端块的 crc32 */
send("block");                  /* 发送块 */
recv("block");                  /* 接收块 */
fuzz_string("dummy");           /* 对字符串 "dummy" 进行模糊测试 */
fuzz_string_uni("dummy");       /* 对 unicode 字符串 "dummy" 进行模糊测试 */
fuzz_hex(0xff ff \xff);         /* 对十六进制值进行模糊测试 */
```

Autodafé 提供的块功能极其简单，使用该功能可以表示大部分的二进制和文本协议格式。

370

Autodafé 框架的主要目标是减少整个模糊测试输入空间的大小和复杂度，使得使用者可以更有效地专注于协议中可能导致出现安全漏洞的区域。对一次完全的模糊测试审计而言，计算输入空间或复杂度并不复杂。考虑下面这个简单的 Autodafé 脚本：

```
fuzz_string("GET");
string(" /");
fuzz_string("index.html");
string(" HTTP/1.1");
hex(0d 0a);
```

当针对一个目标 Web 服务器启动该脚本之后，这个脚本会首先在许多 HTTP 动词的变异中迭代，然后在许多动词参数的变异中迭代。假如 Autodafé 带有一个包含 500 个条目的模糊测试字符串替换库，那么，完成审计所需的测试用例的总数是 500 乘以需要模糊测试的动词，总共 1000 个测试用例。在大多数真实的场景中，字符串替换库的大小至少是我们假定大小的两倍，而且同时会有数百个需要进行模糊测试的变量。Autodafé 使用了一种名为 Markers 技术，这种有趣的技术为每个模糊测试变量设置权重。Autodafé 中的 Marker 可以被定义为数据、字符串或数值，由用户（或是模糊测试器）决定。这个权重用于决定处理模糊测试变量的顺序，使得我们可以聚焦在那些更可能导

致发现安全漏洞的变量上。

为完成模糊测试的任务，Autodafé 包含一个名为 adbg 的调试器组件。调试技术一直都与模糊测试器一同使用，甚至某些特定的模糊测试工具，如 FileFuzz（见第 13 章，“Windows 平台上的文件格式自动化模糊测试”）还包含了调试器，但 Autodafé 是第一个明确地包含调试器的模糊测试框架。Autodafé 使用调试器组件在通常的危险 API（如 strcpy() 和 fprintf()）上设置断点，strcpy() API 是已知的导致缓冲区溢出的原因之一，而 fprintf() 是已知的导致格式字符串漏洞的原因之一。模糊测试器将测试用例同时发送给被测目标和调试器。然后，调试器监视对危险 API 的调用，查找模糊测试器发送的字符串。

每个模糊测试变量都被当成一个 Marker。如果检测到某个 Marker 被传给危险 API，则该 Marker 的权重会增加。 在首次测试中我们不对未曾传给危险 API 的 Marker 进行模糊测试。具有高权重的 Marker 具有高优先级，会被首先进行模糊测试。调试器检测到一个访问违例时，模糊测试器会自动地收到通知，并将相应的测试用例记录下来。通过为模糊测试变量设置优先级，忽略那些不会被危险 API 使用的变量，可以明显减小模糊测试的输入空间。

Autodafé 带有几个附加工具，这些工具可以帮助快速和高效地开发出模糊测试器。第一个工具是 PDML2AD，该工具能够解析 Ethereal 和 Wireshark 导出的包数据 XML（pack data XML，PDML）文件，将其转换为基于块的 Autodafé 语言。如果你的目标协议包含在 Ethereal 和 WireShark 这些流行的网络嗅探器所能识别的超过 750 种的协议中，那么大部分枯燥的基于块的建模工作就可以被自动处理了。即使你的目标协议不能被这些网络嗅探工具所识别，PDML2AD 也能够提供一些捷径，因为它能自动地检测纯文本字段并生成恰当的对 hex()、string() 等函数的调用。第二个工具，TXT2AD，是一个简单的 shell 脚本，该脚本能够将文本文件转换成 Autodafé 脚本。最后一个工具是 ADC，是一个 Autodafé 编译器。ADC 可以在开发复杂的 Autodafé 脚本时发挥作用，它能够检测常见错误，例如不正确的函数名称和未关闭的块。

Autodafé 是一个经过完善考虑的高级模糊测试框架，基于 SPIKE 框架已有的工作进行了扩充。也因此，Autodafé 与 SPIKE 具有诸多一致的优缺点。Autodafé 框架最吸引人的功能是调试组件，这是在所有框架中表现突出的一点。由于该框架缺少对微软 Windows 的支持，对某些用户来说，这可能会导致该框架立即被排除在可用名单之外。另外，由于修改该框架需要重新编译，在模糊测试脚本间重用代码并不像它应该做到的那样透明和容易。

21.3 定制模糊测试器案例研究：Shockwave Flash

在本章前面的各节中，我们展示了目前可用的模糊测试框架。本书的网站 http://www.fuzzing.org 上提供了更完整的可用模糊测试框架的列表。我们鼓励读者分别下载这些测试框架，浏览其文档并尝试框架包附带的例子。这些框架中没有哪个是所有情况下都能比其他框架更好地处理任务的“最佳框架”。事实是，根据不同的被测目标、测试目的、时间限制、预算等因素，某个框架可能比其他框架表现更好。对一个测试团队来说，熟悉多种框架能够帮助团队在为给定任务选择最合适的工具时提供更大的灵活性。

虽然这些公开可用的模糊测试框架能够轻易地描述大部分协议和文件格式，但仍然存在一些所有工具都不适用的情况。当对这些特别的软件或协议进行审计时，创建一个新的、专为特定目标服务的模糊测试器能够产生更好的结果，并能在审计过程中节省时间。在本节中，我们以 Adobe 的 Macromedia Shockware Flash[19]（SWF）文件格式为例，讨论建立一个定制的测试方案。Shockwave Flash 中的安全漏洞通常都具有很大的影响，因为这个星球上几乎每台桌面计算机上都安装有某个版本的 Flash 播放器。事实上，现代的 Windows 操作系统就默认带有某个版本的 Flash 播放器。仔细和全面地研究 SWF 文件的审计大大超出了本章的范围，所以在本章中我们只讨论和本章的主题最相关和最有意思的部分。读者可以访问本书的官网 http://www.fuzzing.org 获得更多的信息，最新的结果及代码清单[20]。

372

我们选择 SWF 作为我们定制的模糊测试器的测试对象有多方面的原因。首先，Adobe 开发者参考手册中的“Macromedia Flash（SWF）和 Flash Video（FLV）文件格式规范”[21]完整地描述了 SWF 文件格式的结构。本书参考的是该文档的版本 8。通常情况下，除非你审计的对象是开源的或自行开发的文件格式，否则你找不到像 SWF 文件格式规范这样全面的文档，因此也就只能采用通用的方法。而对本案例而言，我们可以尽量利用 Adobe 和 Macromedia 提供的文档中的每个细节。通过检查规范文档，我们发现 SWF 文件格式主要被以比特位流方式解析，而不是像大多数文件和协议那样被以字节方式解析。SWF 文件通常在互联网上传播，互联网的流式传输方式可能是 SWF 文件选择使用比特级解析背后的原因。SWF 选择的这种比特级的解析方式增加了模糊测试

---

19 http://www.macromedia.com/software/flash/about/

20 该项研究主要由 TippingPoint 的 Aaron Portnoy 主导。

21 http://www.adobe.com/licensing/developer/

器的复杂度，因此我们更有理由创建一个定制的方案，因为我们在本章所探索过的所有模糊测试框架都不支持比特类型。

21.3.1　为 SWF 文件建模

为了对 SWF 文件格式进行成功的模糊测试，变异和生成测试用例是模糊测试器必须完成的第一个任务。SWF 文件由一个文件头（header）和一系列的标签组成，其基本结构如下所示：

```
[Header]
    <magic>
    <version>
    <length>
    <rect>
        [nbits]
        [xmin]
        [xmax]
        [ymin]
        [ymax]
    <framerate>
    <framecount>

[FileAttributes 标签]
[Tag]
    <header>
    <data>
        <datatypes>
        <structs>
        ...
...
[ShowFrame 标签]
[End 标签]
```

以下是对各个字段的描述。

- magic：3 个字节。所有合法的 SWF 文件中这个字段的值都应该是“FWS”。
- version：1 个字节。生成该 SWF 文件的 Flash 的数字版本号。
- length：4 个字节。指示整个 SWF 文件的大小。
- rect：该字段包含以下 5 个部分。
  - nbits：长度为 5 比特。表示该字段的后 4 个部分中每个部分的比特长度。

373

- xmin, xmax, ymin, ymax：这 4 个字段指示渲染 Flash 内容的屏幕区域的大小。注意这些字段值的单位不是像素，而是 twip，twip 是 Flash 中的度量单位，一个 twip 为一个“逻辑像素”的 1/20。

从 rect 字段的定义中我们意识到 SWF 文件格式确实比较复杂。如果 nbits 字段的值为 3，那么文件头中的 rect 部分的长度为 5+3+3+3+3 = 17 比特。如果 nbits 字段的值为 4，那么文件头的 rect 部分的长度为 5+4+4+4+4 = 21 比特。基于我们前面讨论过的可用的模糊测试框架来表示这个结构并不容易。文件头的最后两个字段表示了帧的回放速度和 SWF 文件中的总帧数。

374

文件头定义之后是一系列定义 Flash 文件的内容和行为的标签。第一个标签是 FileAttribute，该标签在 Flash 版本 8 中引入，有两个作用。第一个作用是指明 SWF 文件是否包含了标题和描述之类的属性。标题和描述等信息通常被像搜索引擎一样的外部进程引用。另一个作用是告诉 Flash 播放器是否应该授予该 SWF 文件本地文件访问或网络文件访问的权限。FileAttibute 标签之后是 SWF 文件中用于定义和控制 Flash 展现内容的标签，这些标签分为两类：定义标签和控制标签。定义标签定义形状、按钮、声音之类的内容。控制标签则定义位置和移动。遇到 ShowFrame 标签前，Flash 播放器仅对标签进行处理而不将其显示出来，遇到 ShowFrame 标签后，Flash 播放器将需要显示的内容渲染到屏幕上。SWF 文件以 End 标签结束。

SWF 文件中有许多可用的标签，每个标签都包含一个两字节的标签头，以及紧跟在标签头后的数据。根据数据的长度，标签会被定义为长标签或短标签。如果数据长度小于 63 比特，相应的标签就会被当作“短标签”，其定义为[10 比特标签 id] [6 比特表示的数据长度] [数据]。如果数据块大于 63 比特，相应的标签就会被当作 “长标签”，其定义为[10 比特标签 id] [111111] [用 4 字节表示的数据长度] [数据]。从这个例子中我们可以再次看到，由于 SWF 格式的复杂性，我们不能使用前面提及的模糊测试框架对它进行良好的建模。

从现在开始，我们要处理更困难的事情。在 SWF 文件中，每个标签都有一套它自己的字段。有些字段包含原始数据，而另一些字段则包含在 SWF 中被称为结构（struct 或 structs）的基础类型。每个结构由一套定义好的字段组成，字段可以包含原始数据，还可以包含其他的结构！还有更难处理的情况：标签和结构中的字段组可以依赖同一个标签或结构中的其他字段的值。即便是上面的解释文字都容易让人混乱，因此让我们从头开始建立模糊测试器，然后通过一些例子更好地对其进行解释。

首先，我们在 Python 中定义一个比特字段类，用于表示各种数值字段的不同比特长度。然后，通过扩展这个比特字段类，我们可以轻松地定义 8 比特、16 比特、32 比特和 64 比特，从而将这个比特字段类扩展为可用于定义字节（字符型）、字（短整型）、双字（长整型）的类。Python 的 sulley 包中已经定义了这些基本的数据结构（请读者注意，这里提到的 sulley 包和 21.3.2 节中提到的 Sully 模糊测试框架不是一回事）。

```
BIG_ENDIAN    = ">"
LITTLE_ENDIAN = "<"

class bit_field (object):
    def __init__ (self, width, value=0, max_num=None):
        assert(type(value) is int or long)

        self.width       = width
        self.max_num     = max_num
        self.value       = value
        self.endian      = LITTLE_ENDIAN
        self.static      = False
        self.s_index     = 0

        if self.max_num == None:
            self.max_num = self.to_decimal("1" * width)

    def flatten (self):
        '''
        @rtype:  Raw Bytes
        @return: Raw byte representation
        '''

        # 将比特流对齐字节边界
        bit_stream = ""

        if self.width % 8 == 0:
            bit_stream += self.to_binary()
        else:
            bit_stream = "0" * (8 - (self.width % 8))
            bit_stream += self.to_binary()

        flattened = ""

        # 将比特流从比特字符串转化为字节数据
        for i in xrange(len(bit_stream) /8 ):
```

375

```
            chunk = bit_stream[8*i : 8*i+8]
            flattened += struct.pack("B", self.to_decimal(chunk))

        # 如果需要，将得到的字节数据进行端转换（从小端转为大端或是相反）
        if self.endian == LITTLE_ENDIAN:
            flattened = list(flattend)
            flattened.reverse()
            flattened = "".join(flattened)

        return flattened

    def to_binary (self, number=None, bit_count=None):
        '''
        @type number:     Integer
        @param number:    (Opt., def=self.value)
        @type bit_count:  Integer
        @param bit_count: (Opt., def=self.width) width of bit string

        @rtype:  String
        @return: Bit string
        '''

        if number == None:
            number = self.value

        if bit_count == None:
            bit_count = self.width

        return "".join(map(lambda x:str((number >> x) & 1),  \
            range(bit_count -1, -1, -1)))

    def to_decimal (self, binary):
        '''
        Convert a binary string into a decimal number and return.
        '''

        return int(binary, 2)

    def randomize (self):
        '''
        Randomize the value of this bitfield
        '''
```

```
        self.value = random.randint(0, self.max_num)

    def smart (self):
        '''
        Step the value of this bitfield through a list of smart values.
        '''

        smart_cases =\
        [
            0,
            self.max_num,
            self.max_num / 2,
            self.max_num / 4,
            # etc ....
        ]

        self.value    = smart_cases[self.s_index]
        self.s_index += 1

class byte (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "8", value)[0]

        bit_field.__init__(self, 8, value, max_num)

class word (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "H", value)[0]

        bit_field.__init__(self, 16, value, max_num)

class dword (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
            value = struct.unpack(endian + "L", value)[0]

        bit_field.__init__(self, 32, value, max_num)

class qword (bit_field):
    def __init__ (self, value=0, max_num=None):
        if type(value) not in [int, long]:
```

```
            value = struct.unpack(endian + "Q", value)[0]

        bit_field.__init__(self, 64, value, max_num)

# 定义这些类的别名
bits    = bit_field
char    = byte
short   = word
long    = dword
double  = qword
```

bit_field 基础类定义了字段的宽度（属性 width）、该字段可表示的最大值（属性 max_num）、字段的值（属性 value）、字段比特位的顺序（属性 endian）、指示字段值是否允许修改的标志（属性 static），以及一个内部使用的索引值（属性 s_index）。此外，bit_field 类还定义了一些有用的函数：

378

- flatten() 函数将字段的值转换为一个字节序列并返回之。
- to_binary() 函数能够将一个数值转换为字符串形式的比特。
- to_decimal() 函数将字符串形式的比特转换成数值。
- randomize() 函数将字段值修改为该字段有效范围内的一个随机值。
- smart() 函数通过一个“智能”边界序列改变字段的值，上面的代码仅显示了“智能边界”的一部分内容。

当以 bit_field 类为基础创建复杂类型时，可以调用最后两个函数对生成的数据进行变异，然后递归调用每个单独组件的 flatten() 函数遍历数据，并将数据写入文件。

现在，我们可以使用以上这些基础类型来定义较为简单的 SWF 结构，如 RECT 和 RGB 结构。下面的代码片段展示了 RECT 和 RGB 类的定义，这两个类都派生自 base 类（这里没有列出 base 类的代码，base 类的定义参见 http://www.fuzzing.org）。

```
class RECT (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.fields =\
        [
            ("Nbits" , sulley.numbers.bits(5, value=31, static=True)),
            ("Xmin"  , sulley.numbers.bits(31)),
            ("Xmax"  , sulley.numbers.bits(31)),
            ("Ymin"  , sulley.numbers.bits(31)),
```

```
            ("Ymax"  , sulley.numbers.bits(31)),
        ]

class RGB (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.fields =\
        [
            ("Red"   , sulley.numbers.byte()),
            ("Green" , sulley.numbers.byte()),
            ("Blue"  , sulley.numbers.byte()),
        ]
```


这段代码展示了开发过程中我们使用的一个简化方式。为了简化模糊测试，我们将所有不定长的字段定义为其允许的最大长度。为了表示带有字段依赖的结构，我们定义了一个全新的、派生自 bit_field 类的原始类（primitive）dependent_bit_field，该类的定义如下：

```
class dependent_bit_field (sulley.numbers.bit_field):
    def __init__ (self, width, max=0, max_num=None, static=False, \
                  parent=None, dep=None, vals=[]):
        self.parent  = parent
        self.dep     = dep
        self.vals    = vals

        sulley.numbers.bit_field.__init__(self, width, value, \
                                          max_num, static)

    def flatten (self):
        # 如果将该结构转换成字节值时存在依赖，则检查之
        if self.parent:
            #
            if self.parent.fields[self.dep][1].value not in self.vals:
                # 没有找到依赖的字段，返回空字符串
                return ""

        return sulley.numbers.bit_field.flatten(self)
```

这个扩展得到的新类指定了一个字段索引和值，在生成数据和返回数据前进行检查。如果属性 dep 引用的字段并没有包含列表 vals 中的值，则返回空字符串。MATRIX 数据结构演示了如何使用这个新定义的原始类：

```
class MATRIX (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.fields =\
        [
            ("HasScale"       , sulley.numbers.bits(1)),
            ("NScaleBits", dependent_bits(5, 31, parent=self, \
                              dep=0, vals=[1])),
            ("ScaleX"         , dependent_bits(5, 31, parent=self, \
                              dep=0, vals=[1])),
            ("ScaleY"         , dependent_bits(5, 31, parent=self, \
                              dep=0, vals=[1])),

            ("HasRotate"      , sulley.numbers.bits(1)),


            ("NRotateBits"    , dependent_bits(5, 31, parent=self, \
                              dep=4, vals = [1])),
            ("skew1"          , dependent_bits(31, parent=self, \
                              dep=0, vals = [1])),
            ("skew2"          , dependent_bits(31, parent=self, \
                              dep=0, vals = [1])),

            ("NTranslateBits", sully.numbers.bits(5, value=31)),
            ("TranslateX"     , sully.numbers.bits(31)),
            ("TranslateY"     , sully.numbers.bits(31)),
        ]
```

从这段代码可以看到，MATRIX 结构中的 NScaleBits 字段被定义为 5 比特宽，默认值为 31。只有当索引 0 处的字段（HasScale）的值为 1 时，NScaleBits 字段才会被包含在 MATRIX 中。ScaleX、ScaleY、skew1、skew2 字段也依赖于 HasScale 字段。换句话说，只有当 HasScale 字段的值为 1 时，以上这些字段才有意义。否则，以上这些字段就不会出现在 MATRIX 结构的定义中。同样，NRotateBits 字段依赖于 MATRIX 结构中索引为 4 的字段（HasRotate）的值。在写作本书的时候，我们已经使用这种标记方式精确地定义了超过 200 个 SWF 结构[22]。

有了所有这些必需的原始类和结构之后，现在，我们可以开始定义 SWF 中的全部标签了。首先，我们定义所有标签类的公共基类（类 base）：

[22] 同样，访问 http://www,fuzzing.org 可以得到定义的源代码。

```
class base (structs.base):
    def __init__ (self, parent=None, dep=None, vals=[]):
        self.tag_id = None
        (structs.base).__init__(self, parent, dep, vals)

    def flatten (self):
        bit_stream = structs.base.flatten(self)

        # 对齐比特流到字节
        if len(bit_stream) % 8 != 0:
            bit_stream = "0" * (8-(len(bit_stream)%8)) + bit_stream

        raw = ""

        # 将字符串形式的比特流转换成字节值
        for i in xrange(len(bit_stream) / 8):
            chunk = bit_stream[8*i : 8*i+8]
            raw += pack("B", self.to_decimal(chunk))

        raw_length = len(raw)

        if raw_length >= 63:
            # 长整形（record_header是一个word+dword）
            record_header = self.tag_id
            record_header <<= 6
            record_header |= 0x3f
            flattened = pack('H', record_header)

            record_header <<= 32
            record_header |= raw_length
            flattened += pack('Q', record_header)
            flattened += raw

        else:
            # 短整形（record_header是一个word）
            record_header = self.tag_id
            record_header <<= 6
            record_header |= raw_length
            flattened = pack('H', record_header)
            flattened += raw

        return flattened
```

base 类的 flatten() 例程自动计算数据部分的长度，生成正确的短标签头或是长标签头。到写作本书时，我们已经用这个框架精确定义了超过 50 个 SWF 标签。下面是几个简单的和中等复杂度的示例：

```
class PlaceObject (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.__init__(self, *args, **kwargs)

        self.tag_id = 4

        self.fields = \
        [
            ("CharacterId"        , sulley.numbers.word(value=0x01)),
            ("Depth"              , sulley.numbers.word()),
            ("Matrix"             , structs.MATRIX()),
            ("ColorTransform"     , structs.CXFORM()),
        ]

class DefineBitsLossless (base):
    def __init__ (self, *args, **kwargs):
        base.__init__(self, *args, **kwargs)

        self.tag_id = 20
        self.fields = \
        [
            ("CharacterId"            , sulley.numbers.word()),
            ("BitmapFormat"           , sulley.numbers.byte()),
            ("BitmapWidth"            , sulley.numbers.word()),
            ("BitmapHeight"           , sulley.numbers.word()),
            ("BitmapColorTableSize"   , structs.dependent_byte( \
                                          parent=self, dep=1, vals=[3])),
            ("ZlibBitmapData"         , structs.COLORMAPDATA( \
                                          parent=self, dep=1, vals=[3]]),
            ("ZlibBitMapData_a"       , structs.BITMAPDATA( \
                                          parent=self, dep=1, vals=[4, 5])),
        ]

class DefineMorphShape (base):
    def __init__ (self, *args, **kwargs):
        base.__init__ (self, *args, **kwargs)
```

```
self.tag_id = 46
self.fields = \
[
    ("CharacterId"          , sulley.numbers.word()),
    ("StartBounds"          , structs.RECT()),
    ("EndBounds"            , structs.RECT()),
    ("Offset"               , sulley.numbers.word()),
    ("MorphFillStyles"      , structs.MORPHFILLSTYLE()),
    ("MorphLineStyles"      , structs.MORPHLINESTYLES()),
    ("StartEdges"           , structs.SHAPE()),
    ("EndEdges"             , structs.SHAPE()),
]
```

现在，让我们回顾上文描述的诸多内容。为了给 SWF 文件建立合适的模型，先从 383
最基本的 bit_field 原始类（primitive）开始。基于 bit_field 类，通过派生得到了表示字节、字、双字和四字的新类。另外，还从 bit_field 类派生得到了 dependent_bit_field 类，然后，又从 dependent_bit_field 类派生得到了带依赖关系的字节、字、双字和四字类。以上这些类型，连同一个新的 base 类，共同组成了表示 SWF 结构的基础。SWF 标签的基类从结构的 base 类派生得到，而结构的 base 类，连同 SWF 结构和我们定义的原始类一起组成了 SWF 标签。图 21.1 描绘了这些类、结构和标签之间的关系。

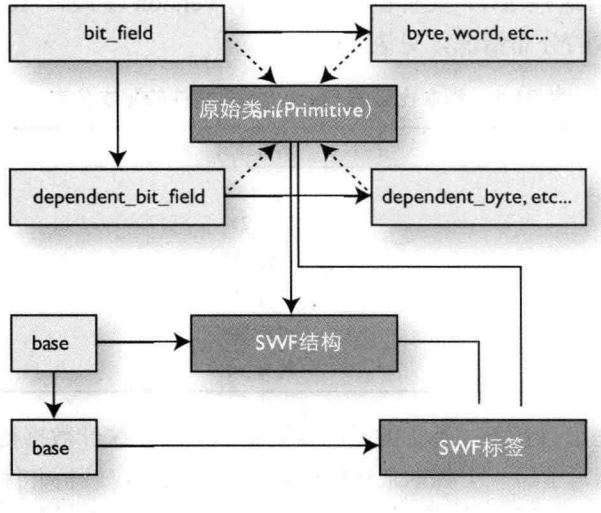


图 21.1　SWF 模糊测试器组件之间的关系

一些其他的构件，如字符串原始类等，也是完成 SWF 建模所必需的。通过源代码可

以看到这些构件的定义。具备了所有这些类之后，我们构建一个全局的 SWF 容器，将其实例化，并开始向其中加入标签。然后，就可以遍历得到的数据结构，使用各组件的 randomize() 和 smart() 例程，通过人工或自动的方式轻松地修改这个数据结构。最后，再次遍历这个复杂的数据结构，将各组件的 flatten() 例程的返回结果连接起来，我们就能将这个数据结构写入文件。这种设计允许在单独的标签或结构内部透明地处理特殊情况。

384

21.3.2 生成有效的数据

在成功解决如何表示基础的 SWF 结构这个挑战后不久，我们又遇到了另一个挑战。SWF 规定标签只能依赖于以前已定义的标签。当对一个依赖多个其他标签的标签进行模糊测试时，必须先成功解析其依赖的所有标签；否则，就无法访问目标标签。猜测字段值的有效范围是个痛苦的过程，但我们有聪明的替代方法。利用 Google SOAP API[23]，使用 filetype:swf 关键字就可以在互联网上搜索 SWF 文件。我们写了一个简单的脚本，该脚本以关键字“a filetype:swf”、“b filetype:swf”……“z filetype:swf”进行搜索，然后解析返回结果，下载搜索到的 swf 文件。在将搜索到的文件下载到本地时，脚本会以文件内容的 MD5 散列值命名该文件，这样就可以避免重复保存具有相同内容的文件。

通过这个脚本，我们找到了大约 10,000 个不同的 SWF 文件，文件总大小超过 3G 字节。通过检查所有这些 SWF 文件的文件头中的 version 字段，我们统计出了这些文件中各种 Flash 版本的分布情况，见表 21.1。

表 21.1 脚本找到的 Flash SWF 文件的版本分布

SWF 版本	所占比例（%）
Flash 8	< 1%
Flash 7	约为 2%
Flash 6	约为 11%
Flash 5	约为 55%
Flash 4	约为 28%
Flash 1 - Flash 3	约为 3%

这个结果看上去很有意思。不幸的是，我们无法从这些统计数据得到 Flash 播放器的版本分布情况。我们写了另一个脚本来解析抓到的每个文件，解析并保存每个文件中含有的 SWF 标签。接下来，我们可以用这个包含有效标签的标签库来帮助创建 SWF 测试用例。

[23] http://www.google.com/apis/index.html

21.3.3　模糊测试环境

385

PaiMei[24]逆向工程框架用于支持实例化与监视单个的测试用例。PaiMei 是一套纯 Python 类，设计目标是帮助开发自动化的逆向工程工具。第 23 章深入描述了如何使用 PaiMei 框架。使用 PaiMei 框架进行测试的过程同 PaiMei FileFuzz 工具的使用方式类似，具体步骤如下：

1．在 PaiMei PyDbg 脚本化调试器的控制下，在 Flash 播放器中加载一个 SWF 测试用例。

2．开始播放并监视执行 5 秒。选择“5 秒”并没有什么特别的理由，我们假定如果 SWF 文件会导致解析出错，播放过程的前 5 秒就会出现错误。

3．如果在播放器 5 秒的播放过程中检测到了错误，记录下这个测试用例及相应的调试信息。

4．如果在 5 秒的播放过程中没有检测到错误，就关闭 Flash 播放器。

5．继续下一个测试用例。

为了更高效地进行模糊测试，我们可以修改 SWF 文件头中 framerate 属性的值，将其从默认的 0x000C 修改为允许的最大值 0xFFFF。通过这一改动，每个测试用例在 5 秒的执行时间内可以处理 SWF 文件中更多的内容。默认查看 SWF 文件的方法是在微软的 Internet Explorer 或 Mozilla 的 Firefox 等浏览器中加载 SWF 文件。对本案例来说，这种查看 SWF 文件的方法尚能接受，但如果我们要对 ActionScript[25]进行模糊测试，应该使用 SAFlashPlayer.exe 这个单机播放器来查看 SWF 文件。SAFlashPlayer.exe 包含在 Macromedia Studio 的发行版中[26]。

21.3.4　测试方法

对 SWF 文件进行模糊测试的最通用的方法是位翻转方法。在前面的章节中我们已经看到了这种基础测试技术在字节层面的应用。然而，在更细粒度的比特层上对 SWF

---

[24] http://openrce.org/downloads/details/208/PaiMei

[25] http://en.wikipedia.org/wiki/ActionScript

[26] http://www.adobe.com/products/studio/

文件应用位翻转方法需要谨慎小心，因为 SWF 文件中的单个字节可能会跨字段。基于位翻转的思路进一步考虑，我们可以使用一个定制的 SWF 标签解析脚本修改单个 SWF 文件内标签的顺序，或在两个不同的 SWF 文件之间交换标签。最后，最彻底的对 SWF 文件进行模糊测试的方法是对每个结构和标签中的每个字段单独进行压力测试。可以参考 21.3.3 节中得到的标签和结构库来剖析一个基础的 SWF 文件，对其进行变异。

386

使用这里提到的三种方法，我们可以用相同的方式将生成的文件通过模糊测试环境发送给被测目标。在 21.4 节中，我们将探索一个新的模糊测试框架，该框架由本书的作者提供。

21.4 模糊测试框架 Sulley

Sulley 是一个模糊测试器开发框架，也是一个模糊测试框架，由多个可扩展的组件组成。以我们的愚见，Sulley 在能力方面超越了之前在商业领域和公开领域发布的大多数模糊测试技术。Sulley 框架的目标不仅要简化表示数据的方法，还希望能够简化数据传输，以及简化对目标的监视。Sulley 这个亲切的名字来自电影“怪物公司”[27]。因为怪物公司中的角色 Sulley 同样是毛茸茸（fuzzy）的小东西（英文中 fuzzy 的意思是“毛茸茸的”，而模糊测试的英文是 fuzz，两个词非常相似）。读者可以从 http://www.fuzzing.org/sulley 下载得到 Sulley 的最新版本。

大多数现代的模糊测试器完全专注于数据生成。Sulley 则不仅有吸引人的数据生成能力，而且还更进一步，含有许多现代模糊测试器应该提供的其他重要能力。Sulley 对网络进行监视并有条不紊地维护监视记录。Sulley 插装并监视目标的健康状态，能够使用多种方法将目标恢复到“好”状态。Sulley 检测和跟踪错误，并能够将检测到的错误归类。Sulley 能够并行地进行模糊测试，显著提升测试速度。Sulley 还可以自动检测错误是由哪些测试用例序列触发的。Sulley 可以在无须人工干预的情况下自动完成上述所有工作，甚至更多。Sulley 的总体用法如下。

- **数据表示**：这是应用所有模糊测试器的第一步。运行目标应用，通过抓取数据包找出数据界面，将协议分解为单个请求（request）并在 Sulley 中以块来表示。
- **会话**：将你开发的请求（request）链接在一起组成一个会话，将 Sulley 提供的各种可用的监视代理（socket、debugger 等）附着在其上，开始进行模糊测试。

---

[27] http://www.pixar.com/featurefilms/inc/chars_pop1.html

- **事后工作**：审查生成的数据和得到的监视结果。重放单个测试用例。

从 http://www.fuzzing.org 下载得到最新的 Sulley 发布包后，将其解压到任意目录。387 解包后得到的目录结构相对复杂，因此我们先来看看 Sulley 是如何通过目录组织所有内容的。

21.4.1 Sulley 的目录结构

Sulley 使用的目录结构“颇有韵味”。保持这个“有韵味的”目录结构能够确保当通过积木（Legos）、请求（requests）和工具（utilities）对模糊测试器进行扩展时，模糊测试器中的所有内容仍能保持良好的组织。下面对分级目录的描述给出了读者需要了解的信息。

- archived_fuzzies：这个目录的格式较为自由，以模糊测试目标的名字来组织，用于存储归档后的模糊测试器和模糊测试会话生成的数据。
  - trend_server_protect_5168：这个目录下存放的是一个执行完毕的模糊测试，在本章后面单步演练中我们会以其作为参考。
  - trillian_jabber：这个目录下存放的是另一个可供参考的执行完毕的模糊测试。
- audits：该目录存放录制得到的 PCAP 数据、崩溃报告、代码覆盖率及活动的模糊测试会话的分析图。一旦模糊测试执行完毕，这些记录数据就会被移动到 archived_fuzzies 目录下。
- docs：该目录下存放的是文档和工具生成的 EpydocAPI 参考手册。
- requests：该目录下存放的是 Sulley 的请求库。每个目标应该有它自己的文件，文件可存储多种请求。
  - __REQUESTS__.html：该文件包含保存的请求类别的描述，并单独列出了每个类型。类别与类型按照字母顺序排列。
  - http.py：各种 Web 服务器模糊测试请求。
  - trend.py：包含与本章后面讨论的完整的模糊测试演练相关的全部请求。
- sulley：模糊测试器框架。除非要扩展该框架，否则不需要动该目录下的文件。
  - legos：用户定义的复杂原始类型。
    - ber.py：ASN.1/BER 协议的原始类型。
    - dcerpc.py：微软 RPC NDR 协议的原始类型。
    - misc.py：各种未归类的复杂原始类型，如 E-mail 地址和主机名等。
    - xdr.py：XDR 类型。

- pgraph：Python 的图表库。在构建会话时使用。
- utils：各种辅助例程
  - dcerpc.py：微软的 RPC 辅助例程，例如支持绑定到一个接口，或生成请求的例程。
  - misc.py：各种未归类的例程，例如 CRC-16 和操作 UUID 的例程。
  - scada.py：面向 SCADA 的辅助例程，包括 DNP3 块编码器。
- __init__.py：定义了各种用于创建请求的以 s_开头的别名。
- blocks.py：定义了一些块和块辅助函数。
- pedrpc.py：定义了 Sulley 使用的客户端和服务器类，这些类用于支持各种代理与主模糊测试器之间的通信。
- primitives.py：定义了模糊测试器的各种原始类型，包括静态数据、随机数据、字符串数据及整数数据。
- sessions.py：该文件包含建立和执行会话的功能。
- sex.py：Sulley 的定制异常处理类。

- unit_tests：Sulley 的单元测试夹具。
- utils：各种单机工具。
  - crashbin_explorer.py：命令行工具，用于浏览保存在序列化的崩溃日志文件中的结果。
  - pcap_cleaner.py：命令行工具，用于从一个 PCAP 目录中清除所有与给定错误无关的数据。
  - network_monitor.py：PedRPC 驱动的网络监视代理。
  - process_monitor.py：PedRPC 驱动的基于调试器的目标监视代理。
  - unit_test.py：Sulley 的单元测试夹具。
  - vmcontrol.py：PedRPC 驱动的 VMWare 控制代理。

现在读者应该对这个目录结构有了一些了解，接下来让我们看看 Sulley 如何处理数据表示。数据表示是构建模糊测试器的第一步。

21.4.2 数据表示

Aitel 已经在 SPIKE 工具中展示了正确的数据表示方法：在研究了我们所能拿到的每个模糊测试器之后，我们发现，对大多数协议来说，基于块的表示方法在简单性和灵活性方面都比其他方法更胜一筹。Sulley 利用基于块的方法生成单个请求，然后将这些请求链接在一起形成会话。首先，用一个新名称初始化请求：

```
s_initialize("new request")
```

389

现在可以开始向请求中加入原始类型（primitive）、块及带嵌套的块。每个原始类型都能够被单独地呈现（rendered）与变异。原始类型的呈现操作会以原始的数据格式返回其内容。而原始类型的变异操作则会改变其内容。在很大程度上，呈现操作和变异操作的概念来自模糊测试器开发者的实践，所以不用操心这两个概念的来源。然而，读者需要知道，当用完可用的模糊测试值后，每个可被变异的原始类型会使用一个默认值。

1. 静态数据和随机数据的原始类型

我们从最简单的原始类型 s_static()开始，该类型向请求中添加一个静态的、任意长度的、不可变异的值。为了方便使用，Sulley 为该类型提供了多个别名，如 s_dunno() 、s_raw() 及 s_unknown() 。

```
# 下面给出的每行代码都是等价的
s_static("pedram\x00was\x01here\x02")
s_raw("pedram\x00was\x01here\x02")
s_dunno("pedram\x00was\x01here\x02")
s_unknown("pedram\x00was\x01here\x02")
```

原始类型、块等组件都带有一个可选的关键字参数“名称”（name）。如果为组件指定了名称，使用 request.names["name"] 就可以直接访问到请求中的这个命名组件，而不需要遍历块结构。 s_binary() 类型同样用来表示不可变异的值，但与 s_staic() 类型并不等价， s_binary() 类型接受以多种格式表示的二进制数据。SPIKE 用户应该认得出这个 API，因为它的用法与 SPIKE 用户熟悉的方式相同（或应该相同）：

```
# s_binary()函数能够处理下面所有这些二进制数据的表示方式
s_binary("0xde 0xad be ef \xca fe 00 01 02 0xba0xdd f0 0d")
```

大多数 Sulley 的原始类型由模糊测试启发式规则驱动，只包含有限数量的变异。s_random() 原始类型是一个例外，该类型可用于生成任意长度的随机数据。s_random() 包含两个必需的参数，分别是 min_length 和 max_length ，用来指定每个迭代中生成的随机数据的最小和最大长度。此外，该原始类型还接受下面这些可选的关键字参数。

- num_mutations (整型，默认值为 25) ：恢复到默认值之前要进行变异的次数。
- fuzzable（布尔型，默认值为 True） ：将此原始类型的模糊测试设置为使能或不使能。

390

- name（字符串，默认值为 None）：与其他 Sulley 对象一样，指定名称允许使用者能够通过名称从请求中直接访问该原始类型。

num_mutations 关键字参数指定了原始类型在用完所有的变异值之前应该被呈现的次数。要用随机数据填充一个固定长度的字段，只需要将 min_length 和 max_length 设为相同值即可。

2. 整型数据

二进制协议和 ASCII 协议都包含有许多不同大小的整数值，例如 HTTP 协议中的 Content-Length 字段就是一个整数值字段。与大多数模糊测试框架一样，Sulley 中也有一些用于表示各种整数的类型。

- 单字节：s_byte()，s_char()
- 两个字节：s_word()，s_short()
- 四个字节：s_dword()，s_long()，s_int()
- 八个字节：s_qword()，s_double()

以上每个整数类型都带有一个必需的参数，即默认的整数值。除了必需的参数外，以下是可选的关键字参数列表。

- endian（字符，默认值为“<”）：位字段的存放顺序。“<”表示用小端方式存储，“>”表示用大端方式存储。
- format（字符串，默认值为“binary”）：整数的输出格式，“binary”或是“ascii”，控制整数类型的显示格式。例如，整数值 100 在 ASCII 模式下会被显示成“100”，而在 binary 模式下则会被显示为“\x64”。
- signed（布尔类型，默认值为 Flase）：将整数设置为有符号的或无符号的，仅当 format 的值为“ascii”时有效。
- full_range（布尔类型，默认值为 False）：如果该参数的值为 True，对该原始类型进行变异时会遍历所有可能的值（稍后会有更详细的说明）。
- fuzzable（布尔类型，默认值为 True）：允许或不允许对该原始类型进行模糊测试。
- name（字符串，默认值为 None）：和其他 Sulley 对象一样，指定名称允许使用者能够通过名称从请求中直接访问该原始类型。

在这些可选的参数中，full_range 是特别有趣的一个。如果我们要对一个双字（DWORD）值进行完全的模糊测试，总共需要 4,294,967,295 个测试用例。就算我们每

秒能完成 10 个用例，单单完成对这一个原始类型的模糊测试也需要 13 年！为了减小输入空间，Sulley 在默认情况下仅尝试那些“智能”值。“智能”值包括我们选定的每个边界值左右的各 10 个值，而我们选定的边界包括：0、最大的整数值（MAX_VAL）、最大整数值的一半（MAX_VAL/2）、最大整数值的 1/3（MAX_VAL/3）、最大整数值的 1/4（MAX_VAL/4）、最大整数值的 1/8（MAX_VAL/8）、最大整数值的 1/16（MAX_VAL/16）及最大整数值的 1/32（MAX_VAL/32）。缩小后的输入空间仅包含 141 个测试用例，数秒内就能完成对这个输入空间的模糊测试。

3. 字符串与分隔符

字符串无处不在。毫无疑问，在进行模糊测试时，你一定会遇到类似 E-mail 地址、主机名、用户名、口令等字符串。Sullet 提供了 s_string() 原始类型用于表示字符串字段。s_string() 原始类型带有一个必需的参数，该参数指定了原始类型的默认值。除了这个必需的参数外，s_string() 原始类型还可以带有以下这些可选的关键字参数。

- **size（整型，默认值为-1）**：字符串的长度。–1 表示长度可变，大于–1 的数据表明字符串具有固定长度，且长度为 size 的值。
- **padding（字符，默认值为 0x00）**：如果生成的字符串长度小于 size 给定的长度，则用 padding 指定的字符来填充字符串。
- **encoding（字符串，默认值为“ascii”）**：字符串使用的编码方式。可接受的选项与 Python 的 str.encode() 函数相同。选项“utf_16_le”对应微软的 Unicode 字符串。
- **fuzzable（布尔值，默认值为 True）**：允许或不允许对该原始类型进行模糊测试。
- **name（字符串，默认值为 None）**：和其他 Sulley 对象一样，指定名称允许使用者能够通过名称从请求中直接访问该原始类型。

通过使用分隔符，字符串经常会被解析为子字段。例如，在 HTTP 请求字符串“GET /index.html HTTP/1.0”中，空格、斜线（/）和点（.）都是字符串中的分隔符。当在 Sulley 中定义协议时，确保通过 s_delim() 原始类型定义分隔符。和其他原始类型一样，s_delim() 原始类型的第一个参数是必需的，用于指定该类型的默认值。此外，s_delim() 原始类型也接受“fuzzable”和“name”这两个关键字参数。针对分隔符的变异方法包括重复、替换和删除。下面这个例子定义了一个对 HTML 的主体（body）部分的标签进行模糊测试的原始类型序列，完整地展示了 Sulley 中的字符串和分隔符：

```
# 模糊测试字符串: <BODY bgcolor="black" >
s_delim("<")
s_string("BODY")
```

```
s_delim(" ")
s_string("bgcolor")
s_delim("=")
s_delim("\"")
s_string("black")
s_delim("\"")
s_delim(">")
```

392

4. 块

掌握了原始类型之后，接下来我们看看如何在块中组织和嵌套这些原始类型。使用s_block_start() 和 s_block_end() 可以定义一个块， s_block_start() 表示块的开始，s_block_ end() 表示块的结束。定义块时必须给块定义一个名字，块的名字作为s_block_start() 的第一个参数传入。除名字外， s_block_start() 还可以接受下面这些可选的关键字参数。

- **group（字符串，默认值为 None）** ：该块要关联的组名（本章稍后我们会更详细地介绍组的概念）。
- **encoder（函数指针，默认值为 None）** ：指向一个函数的指针，该指针指向的函数负责在数据返回前对其进行处理。
- **dep（字符串，默认值为 None）** ：参数值为一个可选的原始类型，该块依赖于特定的原始类型的值。
- dep_value（混合类型，默认值为 None） ：为了能够呈现该块，dep 字段必须要包含的值。
- **dep_values（混合类型列表，默认值为[]）** ：为了能够呈现该块，dep 字段可以包含的值。
- **dep_compare（字符串，默认值为“==”）** ：应用于依赖的比较方法。有效的选项包括==、!=、>、>=、<和<=。

分组（grouping）、编码（encoding）和依赖（dependency）都是其他大多数模糊测试框架所不具有的特性，因此我们将对这些概念进行进一步的研究和分析。

（1）分组

分组操作允许将一个块绑定到一组原始类型，表明这个块应该为组中的每个值循环遍历所有可能的变异值。例如，在需要表示一个带相似参数结构的有效操作或动作的列表时，组这种原始类型就很有用。s_group() 原始类型定义一个组，带有两个必需的参数。第一个参数指明组的名称，第二个参数指明将要在其中遍历的可能的原始值列表。

以下是一个简单的例子，该例子是一个对 Web 服务器进行模糊测试的完整请求：

```
# 导入 Sulley 的所有模块
from sulley import *

# 用于模糊测试的请求为: {GET,HEAD,POST,TRACE} /index.html HTTP/1.1

# 定义一个名为 "HTTP BASIC" 的新块
s_initialize("HTTP BASIC")

# 定义一个列出我们希望模糊测试的各种 HTTP 动词的组原始类型
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])

# 定义一个名为 "body" 的新块并将其与上一个组进行关联
if s_block_start("body", group="verbs"):
        # 将 HTTP 请求的剩余部分分解为单个的原始类型
        s_delim(" ")
        s_delim("/")
        s_string("index.html")
        s_delim(" ")
        s_string("HTTP")
        s_delim("/")
        s_string("1")
        s_delim(".")
        s_string("1")
        # 以 HTTP 要求的结束符结束请求
        s_static("\r\n\r\n")
# 关闭块，name 参数可选
s_block_end("body")
```

脚本首先导入 Sulley 的所有模块。接下来，脚本初始化了一个新请求并将其命名为“HTTP BASIC”。通过这个名字就能直接引用该请求。接下来，脚本定义了一个名为 verbs 的组，该组包含 4 个可能的字符串值：GET、HEAD、POST 和 TRACE。然后，脚本定义了一个新块，该块名为 body，并与组 verbs 关联。注意，s_block_start() 函数总是返回 True，这样，使用一个简单的 if 语句就能将块中包含的所有原始类型都用 TAB 对齐。注意，s_block_end() 同样带有一个可选的 name 参数。采用这样的设计方式纯粹是出于美学目的。再然后，脚本在 body 块内定义了一系列的原始类型并结束这个块。当 Sulley 中的会话加载这个定义好的请求时，对组 verbs 内定义的每个动词，模糊测试器都将为 body 块生成和传输所有的可能值。

394

（2）编码器

编码器是个简单但功能强大的块修饰符。编码器是一个可以附加在块上的函数，该函数在块返回并传输数据之前修改块的内容。用一个实际的例子能够更好地解释编码器。Trend Micro Control Manager 的 DcsProcessor.exe 后台进程在 TCP 端口 20901 上监听，接受用私有的 XOR 编码例程格式化后的数据。通过对解码器进行逆向工程，我们开发出了下面这个编码器：

```
def trend_xor_encode (str):
        key = 0xA8534344
        ret = ""

        # 对齐到 4 字节
        pad = 4 - (len(str) % 4)

        if pad == 4:
            pad = 0

        str += "\x00" * pad

        while str:
            dword    = struct.unpack("<L", str[:4])[0]
            str      = str[4:]
            dword    ^= key
            ret      += struct.pack("<L", dword)
            key      = dword

        return ret
```

Sulley 编码器带有一个参数，即要编码的数据，返回值是编码后的数据。现在可以将我们定义的编码器附加到包含可被模糊测试的原始类型的块上，这样模糊测试器开发者就能绕开这个小障碍继续工作了。

（3）依赖

依赖允许我们将条件应用在呈现整个块的过程中。首先需要用可选的 dep 关键字将块与原始类型关联起来。当 Sulley 开始呈现这个具有依赖的块时，Sulley 会检查它所关联的原始类型并据此行动。使用 dep_value 关键字参数可以指定依赖值。通过 dep_values 关键字参数可以指定一个依赖值列表。

395

最后，通过 dep_compare 关键字参数可以修改条件比较的方式。例如，考虑下面这

种情况。在下面描述的情况中，根据所依赖的整数值的不同，我们期望不同的数据格式：

```
s_short("opcode", full_range=True)

# 操作码（opcode） 10 期望一个认证序列
if s_block_start("auth", dep="opcode", dep_value=10):
        s_string("USER")
        s_delim(" ")
        s_string("pedram")
        s_static("\r\n")
        s_string("PASS")
        s_delim(" ")
        s_delim("fuzzywuzzy")
s_block_end()

# 操作码（opcode） 15 和 16 期望一个单个的字符串主机名
if s_block_start("hostname", dep="opcode", dep_value=[15,16]):
        s_string("pedram.openrce.org")
s_block_end()

# 其他的操作码（opcode）接受以两个下画线开头的字符串
if s_block_start("something", dep="opcode", dep_value=[10,15,16],
dep_compare="!="):
        s_static("__")
        s_string("some string")
s_block_end()
```

通过以多种方式链接块依赖，可以实现强大的（不幸的是，强大也意味着复杂）组合。

5. 块辅助函数

为了有效地使用 Sulley，Sulley 的块辅助函数是读者必须熟悉的，与数据生成相关的重要内容。Sullley 的块辅助函数包括块大小辅助函数、校验和辅助函数及重复辅助函数。

（1）块大小辅助函数

SPIKE 用户应该很熟悉 s_sizer()（或 s_size()）块辅助函数。这个辅助函数用于度量给定块的大小，其第一个参数是块名称，除块名称外，该函数还可以带有下面这些关键字参数。

- **length（整型，默认值为 4）**：块大小辅助函数对应字段的长度。
- **endian（字符，默认值为“<”）**：位字段的顺序。“<”表示小端，“>”表示大端。

396

- format（字符串，默认值为“binary”）：输出格式，参数值可以是“binary”或“ascii”，控制整数原始类型呈现时的输出格式。
- inclusive（布尔类型，默认值为 False）：函数的返回结果是否应该包含它自身的长度？
- signed（布尔类型，默认值为 False）：将 size 设置为有符号或无符号的，仅当 format 为“ascii”时有效。
- fuzzable（boolean 类型，默认值为 False）：允许或不允许对原始类型进行模糊测试。
- name（字符串，默认值为 None）：和其他 Sulley 对象一样，指定名称允许使用者能够通过名称从请求中直接访问该原始类型。

块大小辅助函数是数据生成中的关键组件，需要使用该组件才能表示诸如 XDR、ASN.1 之类的复杂协议。Sulley 会在执行该函数时动态地计算相关块的长度。默认情况下，Sulley 不会对块大小辅助函数对应的字段进行模糊测试。在许多情况下，预期行为都是不对该函数对应的字段进行模糊测试，但如果确实需要对其进行模糊测试，只需将 fuzzable 标志设置为 True。

（2）校验和辅助函数

与块大小辅助函数类似，s_chechsum() 辅助函数针对的也是一个给定的块，该函数计算给定块的校验和，接受的第一个参数是给定块的名称，此外，该函数还接受下列可选的关键字参数。

- algorithm（字符串或函数指针，默认值为“crc32”）：指明应用在目标块上的校验和算法（该参数的有效值包括“crc32”、“adler32”、“md5”、“sha1”）。
- endian（字符，默认值为“<”）：位字段的顺序。“<”表示小端，“>”表示大端。
- length（整型，默认值为 0）：校验和的长度，0 表示自动计算长度。
- name（字符串，默认值为 None）：和其他 Sulley 对象一样，指定名称允许使用者能够通过名称从请求中直接访问该原始类型。

algorithm 参数的值可以是 crc32、adler32、md5 或 sha1 中的一个。或者，你也可以将该参数的值指定为一个函数指针，这样可以在指定的块上应用自定义的校验和算法。

（3）重复器辅助函数

s_repeat()（或 s_repeater()）辅助函数用于将一个块重复多次。例如，当对带有多个

元素的表进行解析以测试可能的溢出时，这个函数是很有用的。s_repeat() 函数带有 3 个必需的参数：被重复的块的名称、最小重复次数、最大重复次数。此外，该函数还可以使用以下这些可选的关键字参数。

- **step（整型，默认值为 1）**：最小和最大重复次数之间的步长。
- **fuzzable（布尔类型，默认值为 False）**：允许或不允许对原始类型进行模糊测试。
- **name（字符串，默认值为 None）**：和其他 Sulley 对象一样，指定名称允许使用者能够通过名称从请求中直接访问该原始类型。

下面是一个使用了以上三个辅助函数的例子，该例子用于对某个协议的一部分进行模糊测试，被测试的这部分协议带有由字符串组成的表。表中的每个条目由 4 个字段组成：两字节的表示类型的字符串字段、两字节的长度字段、字符串字段，以及 CRC-32 校验和字段。由于不知道表示类型的字段的有效值是哪些，因此我们采用随机值对该字段进行模糊测试。在 Sully 中被测试协议的表示看起来类似这样：

```
# 表条目：[类型][长度][字符串][校验和]
if s_block_start("table entry"):
        # 我们不知道有效的类型值是什么，因此使用随机数对其进行模糊测试
        s_random("\x00\x00", 2, 2)

        # 接下来，我们加入一个字段长度为 2 的块大小辅助函数
        s_size("string field", length=2)

        # 块辅助函数仅能作用于块上，因此将字符串原始类型封装在一个块中
        if s_block_start("string field"):
            # 字符串的默认值是一个简单的字符 C 的序列
            s_string("C" * 10)
        s_block_end()

        # 加上字符串的 CRC-32 校验和
        s_checksum("string field")
s_block_end()

# 以步长 50 重复表中的条目，从 100 到 1000 重复次数
s_repeat("table entry", min_reps=100, max_reps=1000, step=50)
```

这个 Sulley 脚本不仅可以模糊测试表条目的解析是否存在问题，还可以发现处理特别长的表时可能发生的错误。

398

6. 积木（Legos）

Sulley 利用积木表示用户定义的组件，如 E-mail 地址、主机名，以及在微软 RPC、XDR、ASN.1 等其他协议中用到的协议原始类型。在 ASN.1/BER 中，字符串表示为[0x04][0x84][双字的字符串长度][字符串]。对基于 ASN.1 的协议进行模糊测试时，在每个字符串前加上长度、类型等前缀非常麻烦。因此，我们可以定义一个积木类型并引用之：

```
s_lego("ber_string", "anonymous")
```

每个积木都遵循类似的格式，但不同的积木可以带有不同的可选（options）关键字参数。下面这个例子是一个名为 tag 的积木的定义，该积木用于对 XMLish 协议进行模糊测试：

```
class tag (blocks.block):
        def __init__ (self, name, request, value, options=[]):
            blocks.block.__init__(self, name, requests, None, None, None, None)

            self.value   = value
            self.options = options

            if not self.value:
                raise sex.error("MISSING LEGO.tag DEFAULT VALUE")

            #
            # [分隔符][字符串][分隔符]

            self.push(primitives.delim("<"))
            self.push(primitives.string(self.value))
            self.push(primitives.delim(">"))
```

这个例子中的积木接受一个字符串形式的标签，并使用恰当的分隔符对其进行封装。该积木扩展了 block 类，通过 self.push() 方法向块中加入标签分隔符及用户提供的字符串。

下面是另一个例子，这个例子中的积木在 Sulley 中用来表示 ASN.1/BER[28]整数。我们选择所有整数的最小公分母 4，将所有整数都表示成 4 字节整数，格式为[0x02][0x04][dword]，其中 0x02 表示整数类型，0x04 表示整数的长度为 4 字节，最后
399 的 dword 表示真正要传入的整数。下面这段代码来自 sulley\legos\ber.py 文件的定义：

---

[28] http://luca.ntop.org/Teaching/Appunti/asn1.html

```
class integer (blocks.block):
        def __init__ (self, name, request, value, options=[]):
            blocks.block.__init__(self, name, request, None, None, None, None)

            self.value   = value
            self.options = options

            if not self.value:
                raise sex.error("MISSING LEGO.ber_integer DEFAULT VALUE")

            self.push(primitives.dword(self, value, endian=">"))

        def render (self):
            #使用父类来 render 方法初始化
            blocks.block.render(self)

            self.rendered = "\x02\x04" + seld.rendered
            return self.rendered
```

与前一个例子类似，我们使用 self.push() 函数将整数加入块中。与前一个例子不同的是，在这个例子中我们重载了 render() 方法，在显示出来的内容前增加了“\x02\x04”前缀，以符合前面提到的整数表示方法。Sulley 能够随着新模糊测试器的创建而成长，开发出来的新**块**和新**请求**能够扩展请求库，并能够很好地帮助构建未来的模糊测试器。接下来，我们该来来看看如何建立一个会话了。

21.4.3 会话

定义了一些请求之后，就可以把这些请求放入会话中了。与其他模糊测试框架相比，Sulley 的主要优势之一是它能够对一个协议进行深入的模糊测试。将请求链接成一个图就可以达成这个目标。在下面的例子中，我们会把一系列请求绑定在一起，并利用 pgraph 库以 uDraw 格式将其显示出来，会话类和请求类都扩展自 pgraph 库。图 21.2 展示了最终的显示结果。

```
from sulley import *

s_initialize("helo")
s_static("helo")

s_initialize("ehlo")
s_static("ehlo")
```

```
s_initialize("mail from")
s_static("mail from")

s_initialize("rcpt to")
s_static("rcpt to")

s_initialize("data")
s_static("data")

sess = sessoins.session()
sess.connect(s_get("hello"))
sess.connect(s_get("ehlo"))
sess.connect(s_get("helo"),        s_get("mail from"))
sess.connect(s_get("ehlo"),        s_get("mail from"))
sess.connect(s_get("mail from"),   s_get("rcpt to"))
sess.connect(s_get("rcpt to"),     s_get("data"))

fh = open("sessions_test.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()
```

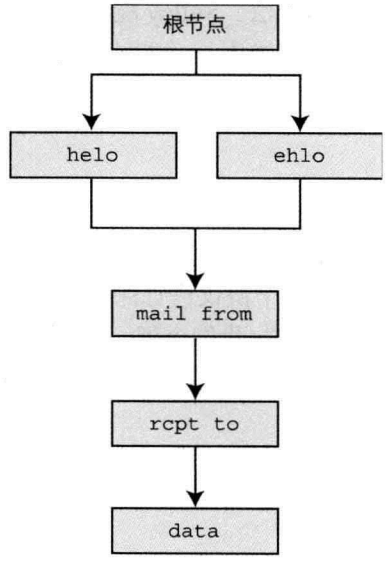


图 21.2　示例的 SMTP 会话的图形化结构

401

当开始模糊测试时，Sulley 从这个图形结构的根节点开始，依次对每一个组件进行

模糊测试。在这个例子中，Sulley 从 helo 请求开始。完成对 helo 请求的模糊测试后，Sulley 会接着对 mail from 请求进行模糊测试。为了对 mail from 请求进行模糊测试，Sulley 会在每个 mail from 的测试用例前加上一个有效的 helo 请求。接下来，Sulley 继续对 rcpt to 请求进行模糊测试。同样地，对 rcpt to 请求的模糊测试也需要在每个测试用例前加上有效的 helo 和 mail from 请求。随后，处理完 data 请求之后，Sulley 会重新回到开头，执行 ehlo 这条路径。这就是 Sulley 的将协议分解为单个请求，并基于协议图对所有可能路径进行模糊测试的能力，这种能力非常强大。以 2006 年 9 月在 Ipswtich Collaboration Suite 中发现的一个问题[29]为例。这个例子中发现的错误是：软件在解析包含字符“@”和“:”的长字符串时出现了栈溢出。这个案例有意思的地方在于，漏洞仅出现在包含 EHLO 请求的路径中，而不出现在包含 HELO 的路径中。如果我们的模糊测试器不能遍历所有可能的协议路径，就可能无法发现这类问题。

创建会话对象时，可以指定下面这些可选的关键字参数。

- **session_filename（字符串，默认值为 None）**：用于保存被序列化的持久数据的文件。如果为该参数指定了一个文件名，就可以在执行模糊测试的过程中暂停测试并在稍后恢复执行。
- **skip（整型，默认值为 0）**：要忽略执行的测试用例的数量。
- **sleep_time（浮点数，默认值为 1.0）**：向目标发送测试用例数据之间等待的时间。
- **log_level（整型，默认值为 2）**：设置日志级别。值越大，输出的日志越多。
- **proto（字符串，默认值为“tcp”）**：用于模糊测试的通信协议。
- **timeout（浮点型，默认值为 5.0）**：等待 send() 和 recv() 返回数据前的等待超时时间（秒）。

Sulley 引入的另一个高级功能是，它允许在协议图结构中定义的每条边上注册回调函数。这个功能允许我们在节点的传输之间注册函数，用于实现挑战-应答系统之类的功能。Sulley 允许定义的回调函数必须遵循以下原型：

```
def callback(node, edge, last_recv, sock)
```

node 是将要发送的节点，edge 是指向 node 的当前模糊测试路径的最后一条边，last_recv 包含上一次套接字传输返回的数据，sock 是当前活动的套接字。举例来说，如果下一个包的大小是在上一个数据包中指定的，在这种情况下回调函数就很有用。下面

[29] http://www.zerodayinitiative.com/advisories/ZDI-06-028.html

402 是另一个回调函数能发挥作用的例子：如果目标服务器具有动态 IP 地址，那么，可以通过注册一个回调函数解决这个问题，回调函数通过 sock.getpeername()[0] 获得服务器的 IP 地址，并将其填入测试用例。除了使用 callback 函数外，通过向 session.connect() 方法中传入可选的关键字参数 callback 也能在协议图的边上注册调函数。

1. 目标和代理

我们的下一步工作是定义目标，将它们同代理连接起来，再将目标加入会话。在下面的例子中，我们实例化一个运行在 VMWare 中的新目标，并将其与三个代理连接起来：

```
target = sessions.target("10.0.0.1", 5168)

target.netmon     = pedrpc.client("10.0.0.1", 26001)
target.procmon    = pedrpc.client("10.0.0.1", 26002)
target.vmcontrol  = pedrpc.client("127.0.0.1", 26003)

target.procmon_options = \
{
        "proc_name"       : "SpntSvc.exe",
        "stop_commands"   : ['net stop "trend serverprotect"'],
        "start_commands"  : ['net start "trend serverprotect"'],
}

sess.add_target(target)
sess.fuzz()
```

实例化的目标（target）绑定在主机 10.0.0.1 的 TCP 端口 5168 上。目标系统上运行了一个网络监视代理，默认在端口 26001 上监听。网络监视器会记录所有的端口通信数据，将其保存在以测试用例号命名的 PCAP 文件中。目标系统上还运行了进程监视器代理，默认在端口 26002 上监听。这个代理需要一个额外的参数，参数描述需要附加监视器的进程的名称、停止目标进程的命令及启动目标进程的命令。最后，VMWare 控制代理运行在本地，默认在本地的 26003 端口上监听。然后，我们将 target 加入会话，开始进行模糊测试。Sulley 能够同时对多个目标进行模糊测试，每个被测试的目标都可以带有独立的连接在一起的代理集合。通过这种方式，Sulley 能够将整个测试空间分布在多个测试目标上，从而节省测试时间。

（1）网络监视器代理（network_monitor.py）

网络监视器代理负责监视网络通信，并将其记录到磁盘上的 PCAP 文件中。该代理硬编码为绑定在 TCP 端口 26001 上，接受来自 Sulley 会话的连接。Sulley 会话通过自

定义的 PedRPC 协议与网络监视器进行通信。将测试用例发送到目标前，Sulley 会向代理发送请求，要求它开始记录网络传输数据。当成功传输测试用例后，Sulley 会再次向代理发送请求，要求它将录制到的数据写入本地磁盘的 PCAP 文件中。为简单起见，代理使用测试用例编号命名保存数据的 PCAP 文件。网络监视器代理不需要与目标软件运行在同一个系统中，然而，它必须能“看到”发送和接收到的网络通信数据。网络监视器代理接受以下这些命令行参数：

```
ERR> Usage: network_monitor.py
        <-d|-device DEVICE #>       嗅探的设备（可用设备见下面的列表）
        [-f|-filter PCAP FILTER]    BPF 过滤字符串
        [-p|-log_path PATH]         存放 pcap 文件的目录
        [-l|-log_level LEVEL]       日志级别（默认值为 1），数字越大，日志越详细

Network Device List:
        [0] \Device\NPF_GenericDialupAdapter
        [1] {2D938150-4270-445F-93D6-A913B4EA20C0} 192.168.181.1
        [2] {9AF9AAEC-C362-4642-9A3F-0768CDA60942} 0.0.0.0
        [3] {9ADCDA98-A452-4956-9408-0968ACC1F482} 192.168.81.193
        ...
```

（2）进程监视器代理（process_monitor.py）

进程监视器代理负责检测模糊测试过程中可能发生在目标进程中的错误。该代理被硬编码为绑定在 TCP 端口 26002 上，接受来自 Sulley 会话的连接。Sulley 使用自定义的 PedRPC 协议与进程监视器代理通信。将单个测试用例传输到目标之后，Sulley 会联系进程监视器代理，要求其检测测试用例是否触发了测试目标上的错误。如果进程监视器代理发现了错误，就会将描述错误本质的高层（high-level）信息传回给 Sulley 会话，这些信息可以通过 Sulley 的内部 Web 服务器对外呈现（我们将会在稍后更详细地介绍 Sulley 的内部 Web 服务器）。同时，被测试用例触发的错误还会被记录到序列化的“崩溃日志”中，以便进行事后分析。我们将会在后文中对该功能进行进一步的介绍。进程监视器代理接受以下这些命令行参数：

```
ERR> USAGE: process_monitor.py
        <-c|-crash_bin FILENAME>    序列化后的崩溃日志保存的文件名
        [-p|-proc_name NAME]        查找和附加在其上的进程名称
        [-i|-ignore_pid PID]        当查找目标进程时忽略掉的 PID
        [-l|-log_level LEVEL]       日志级别（默认为 1），数字越大，日志越详细
```

（3）VMWare 控制代理（wmcontrol.py）

VMWare 控制代理被硬编码为绑定在 TCP 端口 26003 上，接受来自 Sulley 会话的

连接。Sulley 会话通过自定义的 PedRPC 协议与该代理通信。VMWare 控制代理提供了
404 一套与虚拟机镜像进行交互的 API，这套 API 能够启动、停止、暂停或重置虚拟机镜像，还能够生成、删除和恢复快照。当检测到错误或者目标应用无法访问时，Sulley 可以联系这个代理并将虚拟机返回到一个已知的“好”状态。Sulley 严重依赖这个代理来完成“平滑的”测试，识别引发特定复杂错误的准确的测试用例序列。VMWare 控制代理接受以下这些命令行参数：

```
ERR> USAGE: vmcontrol.py
        <-x|-vmx FILENAME>          指向 VMX 控制的路径
        <-r|-vmrun FILENAME>        指向 vmrun.exe 的路径
        [-s|-snapshot NAME>         设置快照名称
        [-l|-log_level LEVEL]       日志级别（默认为 1），数字越大，日志越详细
```

2. Web 监视接口

Sulley 会话类带有一个内建的最小化的 Web 服务器，该服务器被硬编码为绑定在端口 26000 上。当会话类的 `fuzz()` 方法被调用时，会话类就会创建一个 Web 服务器线程，用户可以通过这个 Web 服务器看到模糊测试的进度，以及测试的中间结果。图 21.3 展示了 Web 服务器提供信息的屏幕截图。

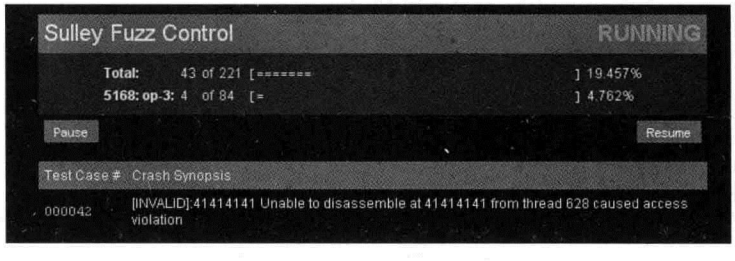


图 21.3 Sulley 的 Web 监视接口

点击“pause”和“resume”按钮可以暂停和继续模糊测试器的运行。检测到的错误会被放入一个列表中，该列表包含导致错误的测试用例序号，以及错误的“症状”。点击列表中的测试用例序号，系统会加载发生错误时输出的详细崩溃结果。当然，崩溃日志文件中同样保存了发生错误时的详细信息，通过程序可以访问到。会话执行结束之后，该进入事后分析阶段，对结果进行分析了。

405 21.4.4 事后分析

当 Sulley 模糊测试会话完成后，应该对结果进行审查，进入事后分析阶段。虽然会

话内建的 Web 服务器提供了一些信息，能够帮助你初步判断可能存在的问题，但在这个阶段，你需要分析出真正的问题所在。Sulley 提供了一些在这个阶段中可以为你提供帮助的工具。第一个工具是 crashbin_explorer.py，该工具接受以下这些命令行参数：

```
$ ./utiils/crashbin_explorer.py
    USAGE:   crashbin_explorer.py <xxx.crashbin>
        [-t|-test #]          导出指定编号测试用例的崩溃症状
        [-g|-graph name]      生成所有崩溃路径的图形，保存到'name'.udg 文件中
```

例如，我们可以使用这个工具查看所有发生错误的位置，列出在这个地址上触发错误的测试用例的编号。下面的内容来自一个真实的对 Trillian Jabber 协议解析器的测试结果分析：

```
$ ./utils/crashbin_explorer.py audits/trillian_jabber.crashbin
    [3] ntdll.dll : 7c910f29 mov ecx, [ecx] from thread 664 caused access violation
          1415, 1416, 1417,
    [2] ntdll.dll : 7c910e03 mov [edx], eax from thread 664 caused access violation
          3780, 9215,
    [24] redenzvous.dll : 4900c4f1 rep movsd from thread 664 caused access
          violation 1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 3443, 3781,
          3782, 3783, 3784, 3785, 3786, 3787, 9216, 9217, 9218, 9219, 9220, 9221,
          9222, 9223,
    [1] ntdll.dll : 7c911639 mov cl, [eax+0x5] from thread 664 caused access
          violation 3442,
```

以上列出的所有错误中，没有哪个是明显的可被利用的问题。因此，我们可以进一步深入到每个错误的细节，使用-t 命令行开关指定测试用例编号。让我们来看看编号为 1416 的测试用例：

```
$ ./utils/crashbin_explorer.py audits/trillian_jabber.crashbin -t 1416
    ntdll.dll : 7c910f29 mov ecx, [ecx] from thread 664 caused access violation
when attempting to read from 0x263b7467
    CONTEXT DUMP
        EIP: 7c910f29 mov ecx, [ecx]
        EAX: 039a0318 ( 60424984) -> gt;&gt;&gt;...&gt;&gt;&gt;&gt;&gt;(heap)
        EBX: 02f40000 ( 49545216) ->       PP@ (heap)
        ECX: 263b7467 ( 641430631) -> N/A
        EDX: 263b7467 ( 641430631) -> N/A
        EDI: 0399fed0 ( 60423888) -> #e<root><message>&gt;&gt;&gt;...&gt;&gt;& (heap)
        ESI: 039a0310 ( 60424976) -> gt;&gt;&gt;...&gt;&gt;&gt;&gt;&gt;(heap)
        EBP: 03989c38 ( 60333112) -> \|gt;&t]IP"Ix;IXIox@ @x@PP8|p|Hg9I P (stack)
        ESP: 03989c2c ( 60333100) -> \|gt;&t]IP"Ix;IXIox@ @x@PP8|p|Hg9I (stack)
```

```
          +00: 02f40000 ( 49545216) ->        PP@ (heap)
          +04: 0399fed0 ( 60423888) -> #e<root><message>&gt;&gt;&gt;...&gt;&gt;& (heap)
          +08: 00000000 ( 0) -> N/A
          +0c: 03989d0c ( 60333324) -> Hg9I Pt]I@"ImI,IIpHsoIPnIX{ (stack)
          +10: 7c910d5c (2089880924) -> N/A
          +14: 02f40000 ( 49545216) ->        PP@ (heap)
disasm around:
              0x7c910f18 jnz 0x7c910fb0
              0x7c910f1e mov ecx, [esi+0xc]
              0x7c910f21 lea eax, [esi+0x8]
              0x7c910f24 mov edx, [eax]
              0x7c910f26 mov [ebp+0xc], ecx
              0x7c910f29 mov ecx, [ecx]
              0x7c910f2b cmp ecx. [edx+0x4]
              0x7c910f2e mov [ebp+0x14], edx
              0x7c910f31 jnz 0x7c911f21

stack unwind:
              ntdll.dll : 7c910d5c
              rendezvous.dll : 49023967
              rendezvous.dll : 4900c56d
              kernel32.dll : 7c80b50b
SEH unwind:
              03989d38 ->ntdll.dll : 7c90ee18
              0398ffdc ->rendezvous.dll : 49025d74
              ffffffff -> kernel32.dll : 7c8399f3
```

还是没有发现明显的可被利用的问题，但从显示的内容中，我们知道测试用例确实影响了这个被发现的访问违例，因为被非法解引用的寄存器 ECX 包含 ASCII 字符串“&;tg”。或许是字符串扩展问题导致的？接下来我们可以用图的方式查看崩溃位置，通过向命令行传入-g 参数，可以得到增加了一个额外维度（已知的执行路径）的图。下面的图（图 21.4）同样来自对 Trellian Jabber 解析器进行测试的真实案例。

可以看到，虽然我们已经发现了 4 个不同的崩溃位置，但看上去这些问题的来源相同。进一步的研究表明我们这个推断完全正确。这个具体的漏洞存在于同步/可扩展通信和表示协议（Redenzvous/Extensiblex Messageing and Presence Protocol，XMPP）通信子系统中。Trillian 通过 UDP 端口 5353 上的_presencemDNS（多播 DNS）服务来定位附近的用户。用户通过多播 DNS 注册，完成注册后通过 TCP 端口 5298 上的 XMPP 进行通信。从 plugins\rendezvous.dll 的实现可以知道，Trillian 接收信息的逻辑如下：

```
4900C470 str_len:
```

```
4900C470      mov cl. [eax]      ; *eax = message+1
4900C472      inc eax
4900C473      test cl, cl
4900C475      jnz short str_len

4900C477      sub eax, edx
4900C479      add eax, 128       ; strlen(message+1) + 128
4900C47E      push eax
4900C47F      call _malloc
```

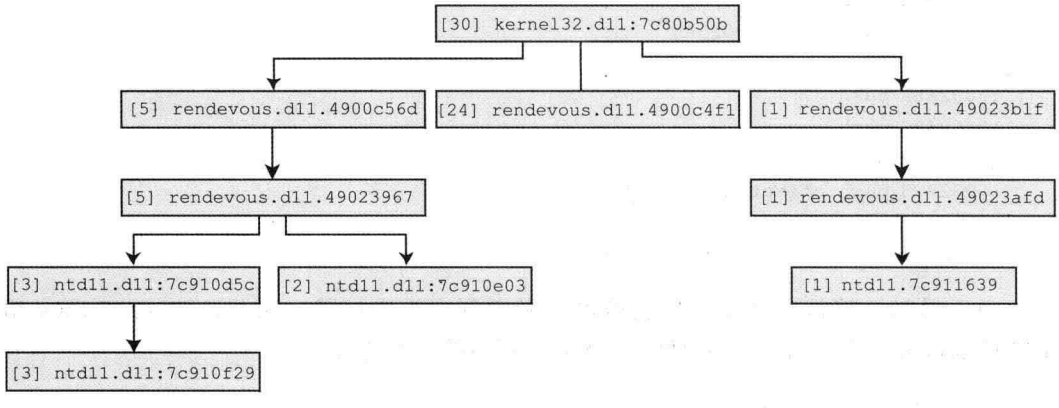


图 21.4　Sully 崩溃二进制日志中调用路径的的图形化展示

这段代码首先计算所接收信息的字符串长度，在堆中分配一个字符串长+128 字节的缓冲区来存储信息的副本，然后将该缓冲区传给 expatxml.xmlComposeString()，调用该函数的方式如下：

```
plugin_send(MYGUID, "xmlComposeString", struct xml_string_t *)

struct xml_string_t {
        unsigned int      struct_size;
        char              *string_buffer;
        struct xml_tree_t *xml_tree;
};
```

这个 xmlComposeString() 例程最终调用了 expatxml.19002420()，该函数负责将字符&、>、<分别编码为&amp、&lt、&gt。从下面的反汇编片段中可以看到这个函数的行为：

```
19002492 push 0
19002494 push 0
```

```
19002496 push offset str_Amp        ; "&amp;"
1900249B push offset ampersand      ; "&"
190024A0 push eax
190024A1 call sub_190023A0

190024A6 push 0
190024A8 push 0
190024AA push offset str_Lt         ; "&lt;"
190024AF push offset less_than      ; "<"
190024B4 push eax
190024B5 call sub_190023A0

190024BA push
190024BC push
190024BE push offset str_Gt         ; "&gt;"
190024C3 push offset greater_than   ; ">"
190024C8 push eax
190024C9 call sub_190023A0
```

由于最初计算得到的字符串长度没有考虑到字符串扩展，因此，在 redezvous.dll 中随后的内存复制操作就可能触发一个可被利用的内存破坏：

```
4900C4EC mov ecx, eax
4900C4EE shr ecx, 2
4900C4F1 rep movsd
4900C4F3 mov ecx, eax
4900C4F5 and ecx, 3
4900C4F8 rep movsb
```

409*

在本例中，Sulley 发现的所有错误都由这个逻辑错误导致。跟踪错误位置和路径使得我们能够快速地假定所有的错误来自同一个源头。我们要执行的最后一个步骤是移除所有不包含错误相关信息的 PCAP 文件。pcap_cleaner.py 工具就是为这个目的创建的。

```
# ./utils/pcap_cleaner.py
        USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

pcap_cleaner.py 工具打开指定的崩溃日志文件，读入所有触发了错误的测试用例的编号，然后从指定的目录下删除所有不以触发了错误的测试用例编号命名的 PCAP 文件。为了帮助读者更好地理解所有这些工具和模块是如何在一起工作的，我们将从头到尾遍历一个完整的真实例子。

21.4.5 一个完整的实例

本案例涉及许多 Sulley 中的中高级概念，应该能够帮助读者更好地理解 Sulley 框架。本节的主要目的是演示如何使用 Sulley 的高级特性，因此，在描述案例的时候我们有意跳过了对被测目标的细节描述。在本案例中，我们选择的被测目标是 Trend Micro Server Protect 软件，该软件通过服务 SpntSvc.exe 绑定位于 TCP 端口 5168 的一个微软 DCE/RPC 端点。被绑定的 RPC 端点从 TmRpcSrv.dll 中导出，其接口定义语言（Interface Definition Language，IDL）桩信息如下：

```
// opcode: 0x00,    address: 0x65741030
// uuid: 25288888-bd5b-11d1-9d53-0080c83a5c2c
// version: 1.0

error_status_t rpc_opnum_0 (
 [in] handle_t arg_1,                    // 不通过网络进行传输
 [in] long trend_req_num,
 [in] [size_is(arg_4)] byte some_string[],
 [in] long arg_4,
 [out] [size_is(arg_6)] byte arg_5[],    // 不通过网络进行传输
 [in] long arg_6
);
```

实际上，arg_1 和 arg_6 参数都不会通过网络传输。当开发实际的模糊测试请求时，这是一个需要考虑的重要因素。进一步的研究表明，参数 trend_req_num 具有特殊的含义。trend_req_num 参数的高低两个部分（高位的一半和低位的一半）控制一对跳转表（jump tables），这对跳转表提供了大量可通过这个 RPC 函数到达的子例程。对跳转表进行逆向工程可以发现下面这些组合：

410

- 当高半部分的值为 0x0001 时，低半部分的有效值是 1 到 21。
- 当高半部分的值为 0x0002 时，低半部分的有效值是 1 到 18。
- 当高半部分的值为 0x0003 时，低半部分的有效值是 1 到 84。
- 当高半部分的值为 0x0005 时，低半部分的有效值是 1 到 24。
- 当高半部分的值为 0x000A 时，低半部分的有效值是 1 到 48。
- 当高半部分的值为 0x001F 时，低半部分的有效值是 1 到 24。

接下来我们必须创建一个定制的编码例程负责将已定义的块封装成有效的 DCE/RPC 请求。需要额外封装的只有一个函数编号，因此很简单。我们基于 utils.dcerpc.request() 定义一个基础封装器，将操作码硬编码为 0：

```
# dce rpc request encoder used for trend server protect 5168 RPC service.
# opnum is always zero.
def rpc_request_encoder (data):
        return utils.dcerpc.request(0, data)
```

1. 构建请求

有了以上信息和我们定义的编码器之后，接下来我们可以开始定义需要的 Sulley 请求了。为此，我们创建一个 requests\trend.py 文件，该文件包含所有 Trend 相关的请求和辅助函数的定义。在创建 trend.py 文件之后，我们开始编码。这个例子很好地说明了为什么用一种真正的编程语言(与使用定制语言相比较)构建模糊测试器会更有优势，因为使用 Python 这种真正的编程语言可以充分利用 Python 的循环特性，自动地为每个有效的 trend_req_num 的高半部分的值生成单独的请求。

```
for op, submax in [(0x1, 22), (0x2, 19), (0x3, 85), (0x5, 25), (0xa, 49), (0x1f, 25)]:
        s_initialize("5168: op-%x" % op)
        if s_block_start("everything", encoder=rpc_request_encoder):
            # [in] long trend_req_num
            s_group("subs", values=map(chr, range(1, submax)))
            s_static("\x00")                    # subs is actually a little
median word
            s_static(struct.pack("<H", op))     # opcode

            # [in][size_is(arg_4)] byte some_string[],
            s_size("some_string")

            if s_block_start("some_string", group="subs"):
                s_static("A" * 0x5000, name="arg3")
            s_block_end()

            # [in] long arg_4
            s_size("some_string")

            # [in] long arg_6
            s_static(struct.pack("<L", 0x5000))     # output buffer size
        s_block_end()
```

在生成的每个请求内部，我们首先初始化一个新的块，将块传给之前定义的定制的编码器。接下来，我们使用 s_group() 原始类型定义一个名为 subs 的序列，该序列用来表示我们前面看到的 trend_req_num 的有效的低半部分的值。然后，我们以静态值方式将高半部分的值加到请求中。由于我们已经通过逆向工程知道了 trend_req_num 的有效

值，因此不需要对它进行模糊测试；如果事先不知道 trend_req_num 的有效值，我们可以设置对这些字段进行模糊测试。接下来，我们将 some_string 的 NDR 大小前缀加入请求中。这里我们可以选择使用 Sulley 的 DCE/RPC NDR 积木原始类型，但因为这个 RPC 请求足够简单，因此我们决定手工表示 NDR 格式。接下来，我们向请求中加入 some_string 值。我们将 some_string 字符串值封装在一个块中，因为只有这样才能度量它的长度。在本案例中我们使用一个长度固定的、由字符 A 组成的字符串（长度约为 20K）。通常我们应该在这儿插入一个 s_string() 原始类型，但由于我们知道 Trend 会在给定任意长字符串的情况下发生崩溃，因此我们仅使用一个静态值，以减小测试集合。接下来，我们向请求中再次加入 some_string 的长度，以填充 arg_4 需要的 size_is 字段。最后，我们为输出缓冲区随意指定一个大小，然后关闭这个块。现在，我们已经准备好了所需的请求，接下来可以创建会话了。

2. 创建会话

我们在 Sulley 的顶级目录下创建了一个新文件 fuzz_trend_server_protect_5168.py 用于表示会话。在 Sulley 的发布包中，这个文件位于 archived_fuzzies 目录中，因为当它完成自己的生命周期后，就会被移动到 archived_fuzzies 目录中。在 fuzz_trend_server_protect_5168.py 文件开头，我们首先导入 Sulley，并从请求库中导入我们创建的 Trend 请求：

```
from sulley import *
from requests import trend
```

接下来，我们定义一个预发送函数，该函数负责在传输测试用例前建立 DCE/RPC 连接。这个函数接受一个参数，也就是传输数据所用的套接字。这个函数基于 utils.dcerpc.bind() ，是一个简单的 Sulley 工具例程：

```
def rpc_bind (sock):
        bind = utils.dcerpc.bind("25288888-bd5b-11d1-9d53-0080c83a5c2c", "1.0")
        sock.send(bind)

        utils.dcerpc.bind_ack(sock.recv(1000))
```

现在可以初始化会话并定义目标了。模糊测试的目标只有一个，就是安装在 VMWare 虚拟机中的一个 Trend Server Protect 实例，该虚拟机的 IP 地址是 10.0.0.1。我们将遵照 Sulley 框架的使用说明，将序列化后的会话信息保存到 audits 目录。最后，我们会注册一个网络监视器、一个进程监视器和一个针对所定义目标的虚拟机控制代理：

```
    sess = sessions.session(session_filename="audits/trend_server_protect_5168.ses
sion")
    target = sessions.target("10.0.0.1", 5168)

    target.netmon     = pedrpc.client("10.0.0.1",    26001)
    target.procmon    = pedrpc.client("10.0.0.1",    26002)
    target.vmcontrol  = pedrpc.client("127.0.0.1",   26003)
```

由于我们提供了一个 VMWare 控制代理，因此，当检测到错误或访问不到目标时，Sulley 会默认回滚到一个已知的“好”快照。如果 VMWare 控制代理不可用，但进程监视代理可用，Sully 就会尝试重启目标进程以继续模糊测试。通过为进程监视代理指定 stop_commands 和 start_commands 选项，可以让进程监视代理重新启动目标进程：

```
    target.procmon_options = \
    {
            "proc_name"       : "SpntSvc.exe"
            "stop_commands"   : ['net stop "trend serverprotect"'],
            "start_commands"  : ['net start "trend serverprotect"'],
    }
```

使用进程监视代理时，proc_name 是强制参数；该参数指定一个进程的名称，调试器应当附加在给定的进程上并在其中寻找错误。如果 VMWare 控制代理和进程监视代理都不可用，那么，当数据传输不成功的时候，Sulley 就只能简单地给目标一段时间，期望目标自行恢复了。

413 接下来，我们调用 VMWare 控制代理的 restart_target() 例程以启动目标进程。当目标进程运行起来之后，我们将目标加入会话中，指定预发送函数，把我们定义的每个请求都连接到模糊测试的根节点。最后，调用会话类的 fuzz() 例程开始模糊测试。

```
    #start up the target
    target.vmcontol.restart_target()

    print "virtual machine up and running"

    sess.add_target(target)
    sess.pre_send = rpc_bind
    sess.connect(s_get("5168: op-1"))
    sess.connect(s_get("5168: op-2"))
    sess.connect(s_get("5168: op-3"))
```

```
sess.connect(s_get("5168: op-5"))
sess.connect(s_get("5168: op-a"))
sess.connect(s_get("5168: op-1f"))
sess.fuzz()
```

3. 设置环境

启动模糊测试会话前的最后一个步骤是设置环境。我们通过以下操作设置环境：启动目标虚拟机镜像，在测试镜像中通过命令行直接启动网络代理和进程监视代理：

```
network_monitor.py -d 1 \
                        -f "src or dst port 5168" \
                        -p audits\trend_server_protect_5168

process_monitor.py -c audits\trend_server_protect_5168.crashbin \
                       -p SpntSvc.exe
```

这两个代理和会话脚本都在 Sulley 顶级目录对应的映射共享中执行。我们向网络监视器传入一个伯克利包过滤器（Berkeley Packet Filter，BPF）过滤字符串，以确保监视器只记录我们感兴趣的数据包。同时传入的还有 audits 目录下的一个子目录，网络监视器会把为每个测试用例创建的 PCAPS 文件保存在这个子目录下。在代理和目标进程运行起来的同时，我们会生成一个快照，并将其命名为“named sulley ready and waiting”。

414

接下来，我们关闭 VMWare 并在运行虚拟机的系统上启动 VMWare 控制代理。VMWare 控制代理需要 vmrun.exe 可执行文件的路径，我们使用的实际镜像路径，以及当发生数据传输错误时需要回退到的快照的名称。

```
vmcontrol.py -r "c:\\VMware\vmrun.exe"
                 -x "v:\\vmfarm\Trend\win_2000_pro.vmx"
                 -snapshot "sulley ready and waiting"
```

4. 预备，开始！以及事后分析

终于一切就绪了。运行 fuzz_trend_server_protect_5168.py，启动一个 Web 浏览器访问 http://127.0.0.1:26000 就可以监视模糊测试器的进度，现在我们什么都不用做，只需要端杯茶坐下，观察输出就行。

当模糊测试器执行完列表中的 221 个测试用例后，我们发现其中 19 个测试用例触发了错误。可以使用 crashbin_explorer.py 工具浏览以异常地址归类的错误：

```
$ ./utils/crashbin_explorer.py audits/trend_server_protect_5168.crashbin
```

```
        [6] [INVALID]:41414141 Unable to disassemble at 41414141 from the thread 568
caused access violation
            42, 109, 156, 164, 170, 198,
        [3] LogMaster.dll:63272106 push ebx from thread 568 caused access violation
            53, 56, 151,
        [1] ntdll.dll:77fbb267 push dword [ebp+0xc] from thread 568 caused access
violation
            195,
        [1] Eng50.dll:6118954e rep movsd from thread 568 caused access violation
            181,
        [1] ntdll.dll:77facbbd push edi from thread 568 caused access violation
            118,
        [1] Eng50.dll:61187671 cmp word [eax],0x3b from thread 568 caused access violation
            116,
        [1] [INVALID]:0058002e Unable to disassemble at 0058002e from thread 568 caused
access violation
            70,
        [2] Eng50.dll:611896d1 rep movsd from thread 568 caused access violation
            152, 182,
        [1] StRpcSrv.dll:6567603c push esi from thread thread 568 caused access violation
            106,
        [1] KERNEL32.dll:7c57993a cmp ax,[edi] from thread 568 caused access violation
            165,
        [1] Eng50.dll:61182415 mov edx.[edi+0x20c] from thread 568 caused access
violation
            50,
```

415

有些问题是明显的可被利用的问题，例如，导致 EIP 为 0x41414141 的测试用例就一个。测试用例 70 看上去是一个无意中发现的可能的代码执行问题，该问题是一个 Unicode 溢出（实际上，如果进行深入研究，你会发现这个问题可能是个直接溢出）。崩溃日志浏览工具能够生成一个所有发现错误的图状视图，也能够基于观测到的栈回溯画出路径。崩溃日志浏览工具可以帮助指出特定问题的根本原因。该工具接受以下命令行参数：

```
$ ./utils/crashbin_explorer.py
    USAGE: crashbin_explorer.py <xxx.crashbin>
        [-t|-test #]          dump the crash synopsis for a specific test case number
        [-g|-graph name]  generate a graph of all crash paths, save to 'name'.udg
```

使用这个工具，我们可以进一步检查当执行测试用例 70 并检测到错误时的 CPU 状态：

```
$ ./utils/crashbin_explorer.py audits/trend_server_protect_5168.crashbin -t 70
    [INVALID]:0058002e Unable to disassemble at 0058002e from thread 568 caused
access violation
```

```
when attempting to read from 0x0058002e
CONTEXT DUMP
    EIP:     0058002e Unable to disassemble at 0058002e
    EAX: 00000001 (            1) -> N/A
    EBX: 0259e118 (     39444760) -> A.....AAAAA (stack)
    ECX: 00000000 (            0) -> N/A
    EDX: ffffffff (   4294967295) -> N/A
    EDI: 00000000 (            0) -> N/A
    ESI: 0259e33e (     39445310) -> A.....AAAAA (stack)
    EBP: 00000000 (            0) -> N/A
    ESP: 0259d594 (     29441812) -> LA.XLT.......MPT.MSG.OFT.PPS.RT (stack)
    +00: 0041004c (      4259916) -> N/A
    +04: 0058002e (      5767214) -> N/A
    +08: 0054004c (      5505100) -> N/A
    +0c: 0056002e (      5636142) -> N/A
    +10: 00530042 (      5439554) -> N/A
    +14: 004a002e (      4849710) -> N/A
disasm around:
        0x0058002e Unable to disassamble

SEH unwind:
        0259fc58 -> StRpcSrv.dll:656784e3
        0259fd70 -> TmRpcSrv.dll:65741820
        0259fda8 -> TmRpcSrv.dll:65741820
        0259ffdc -> RPCRT4.dll:77d87000
        ffffffff -> KERNEL32.dll:7c5c216c
```

从这里可以看到，栈被看上去像是文件扩展名的 Unicode 字符串所清空。你也可以找出给定测试用例对应的 PCAP 文件来查看。图 21.5 显示了 Wireshark 工具的截图，该工具正在检查一个 PCAP 文件的内容。

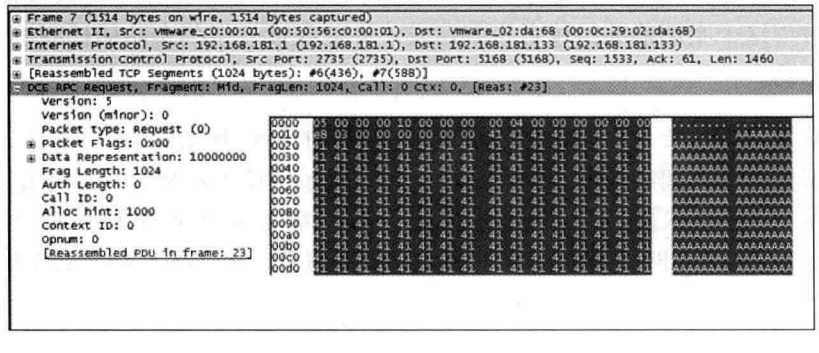


图 21.5　Wireshark DCE/RPC 分析

我们要采取的最后一个步骤是移除所有不包含错误相关信息的 PCAP 文件。pcap_cleaner.py 工具最适合用来完成这个任务：

```
$ ./utils/pcap_cleaner.py
    USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

pcap_cleaner.py 工具会打开给定的崩溃日志文件，读入触发错误的测试用例编号列表，从指定的目录下删除所有文件名不在列表中的 PCAP 文件。我们已经将在本次模糊测试中发现的代码执行漏洞都汇报给了 Trend 公司，下面这些安全报告就源于我们发现的漏洞：

417

```
TSRT-07-01: Trend Micro ServerProtect StCommon.dll Stack Overflow
TSRT-07-02: Trend Micro ServerProtect eng50.dll Stack Overflow Vulnerabilities
```

当然，这并不是说我们已经发现了这个接口中所有可能的漏洞。实际上，我们的案例可能只是针对这个接口最基础的模糊测试。如果再进行一次模糊测试，使用 `s_string()` 原始类型而不是简单地使用长字符串会更有优势。

21.5 小结

模糊测试框架为缺陷检查者和 QA 团队提供了一种灵活、可复用、同构的开发环境。在本章中，对所有可用的模糊测试框架进行探索之后，我们应该很清楚没有任何一种框架是所谓的“最佳框架”。所有这些框架都提供了合理的构建模糊测试的基础，但选择何种框架通常被测试目标及使用者对特定框架使用的编程语言的熟悉程度所左右。Adobe Macromedia Shockware Flash 的 SWF 文件格式案例分析表明框架仍然是一种不成熟的技术，读者仍可能会遇到没有任何一种现存框架能满足你的需求的情况。正如演示的那样，你可能偶尔会需要用到为手头上的任务开发的可重用组件，开发定制的模糊测试方案。希望这个案例能够给读者带来启发，并能在读者遇到奇葩的模糊测试任务时，提供一些帮助。

在本章的最后一节，我们介绍并探索了一个新的模糊测试框架：Sulley。通过深入研究这个模糊测试框架的多个方面，我们清楚地展示了这个框架的优越之处。Sulley 是不断扩大的模糊测试框架家庭中的最新成员，本书的作者在积极地维护它。从 http://www.fuzzing.org 网站可以获得该框架更新的信息、更多的文档及最新的例子。

第22章

自动化协议分析

"I know how hard it is for you to put food on your family."

——George W. Bush, Greater Nashua, NH, January 27, 2000

在前面的章节中，我们详细研究了多种模糊测试器。从简单的字节变异式的通用文件模糊测试器，到最新的模糊测试框架，模糊测试技术在进化的过程中克服了许多自动化软件测试的障碍。在本章中，我们将介绍模糊测试的最高级形态，并尝试解决一个所有模糊测试技术面临的共同困境。具体地说，我们要解决的这个棘手的问题是，如何将协议分解为基本构件。

本章首先讨论自动化协议分析的基本技术，然后深入讨论应用生物信息学和基因算法进行协议分析。这些都是自动化软件测试的最前沿领域，理论性极强，不少人正在积极地研究这些课题。

22.1 模糊测试的痛处

模糊测试中最痛苦的地方是进入门槛高，尤其是当需要对缺乏文档的、复杂的二进制协议进行模糊测试时，我们需要投入大量的研究工作才能理解协议。想象这个场景：我们需要在 Samba[1]的第一个版本发布前对微软的 SMB 协议进行模糊测试。Samba 项目

[1] http://www.samba.org

420 的第一个版本发布于 1992 年，在数不清的志愿者投入大量时间的帮助下，Samba 项目得以逐渐成长为一个成熟的，功能完整的，兼容 Windows 的 SMB 客户端-服务器。感谢这些志愿者的工作，现在我们可以对 SMB 协议进行良好的模糊测试。但如果没有这些志愿者，你是否有足够大的团队和超过 10 年的时间来对一个类似的没有文档的私有协议进行模糊测试？

Samba

Samba 始于 1992 年 1 月 10 日。在这一天，来自澳大利亚国立大学的 Andrew Tridgell 向 vmsnet.networks.desktop.pathworks 新闻组发布了一条消息[2]。在消息中 Andrew 宣布发布了一个与 DOS 下的 Pathworks 兼容的 UNIX 文件共享服务。这个项目最初叫作 nbserver，但 1994 年 4 月，Tridgell 接到了来自 Syntax 公司的邮件，邮件中提到 nbserver 这个名字可能构成了商标侵权[3]。随后，该项目改名为 Samba。15 年后的今天，Samba 已经成为目前应用最广泛的开源项目之一。

对你自己的协议进行模糊测试很容易。因为你已经对这类协议的格式、协议中的细微差别、协议的复杂性，甚至是那些可能你自己不怎么有把握、需要格外注意的地方都了如指掌。对具有良好文档的协议（如 HTTP 协议）进行模糊测试也相对容易。因为你可以参考 RPC[4]描述的细节内容，以及其他的公开文档。你要做的不过是提供有效且全面的测试用例数据。当然，需要注意的是，即使你要测试的协议是具有良好文档的常用协议，但这也不意味着软件开发商一定会严格遵循已发布的标准。下面是发生在 Ipswitch IMail 软件中的实际案例。2006 年 9 月，有人发现了一个影响 Ipswitch 的 SMTP 后台进程的远程溢出漏洞[5]。导致这个漏洞的原因是软件在处理包含字符“@”和“:”的长字符串时没有进行边界检查。查看二进制代码的相关部分发现很难通过审计方式发
421 现这个问题。然而，由于包含漏洞的特性是 Ipswitch 的专有特性，没有任何标准文档对此有描述，因此，模糊测试器也不大可能发现这个漏洞。

对专有的第三方协议进行模糊测试，不管协议多么简单，都可能是一个充满挑战的过程。纯粹随机的模糊测试实现起来极快，但得到的结果却不怎么好。而其他的通用技术，如字节翻转等，能提供更好的测试，但却需要一个额外步骤：必须首先观察客户端

---

2 http://groups.google.com/group/vmsnet.networks.desktop.pathworks/msg/7d939a9e7e419b9c

3 http://www.samba.org/samba/docs/10years.html

4 http://www.w3.org/Protocols/rfc2616/rfc2616.html

5 http://www.zerodayinitiative.com/advisories/ZDI-06-028.html

和服务器之间的有效通信数据。幸好，大多数情况下，在给定的客户端-服务器对中触发通信还算容易。但是，拿到通信数据后，我们却无法确切地知道该协议还有多大比例的部分没有被观察到。以 HTTP 协议为例，假设我们不具有任何 HTTP 协议的知识，只需要短时间地观察客户端和服务器之间的通信，我们可能就能发现 GET 和 POST 动作。但是，其他那些不怎么明显的动作，如 HEAD、OPTIONS 以及 TRACE 等则不一定会被观察到。怎样才能发现协议中使用频率较低的那些部分？为了更好地分析目标协议，我们可以对客户端和服务器的二进制代码开展逆向工程。逆向工程的代价非常高，因为只有具有高超技巧的，有经验的工程师才能执行逆向工程，而这种工程师很难找到。某些情况下，特别是要进行模糊测试的协议是纯文本协议时，我们可以通过猜的方式猜测这些出现频率低的特性。但对二进制协议来说，猜的成本可就太高了。

在雇用昂贵的逆向工程师之前，我们建议确保已经充分利用了他人的工作成果。这个世界很大，如果你需要去了解一个专有协议的细节，很可能其他人也和你有同样的需求。在深入探索专有协议之前，确保你已经通过 Google 彻底地检索过相关信息。可能你会惊喜地发现，某个和你面临同样挑战的人已经为你感兴趣的协议撰写了非官方文档。另外，Wotsit.org 网站是一个文件格式信息的好来源，包含大量已被文档化和未被文档化的文件的格式描述。Wireshark 和 Ethereal 的源代码则为网络协议分析提供了一个好的起点，这两个工具的源代码中已经包含了许多被彻底研究过的、较为知名的私有协议的定义。

当解开某种专有协议秘密的重任完全落在你的肩头时，你一定强烈渴望一种需要较少人工参与的协议分析方法。本章将专门介绍各种方法，这些方法有助于自动地在网络协议和文件格式的噪声中检测信号，在数据中检测结构。

22.2　启发式技术

本节介绍的两种技术都不是真正意义上的自动化协议分析技术。但是，这两种基于启发式规则的技术能够用于提升模糊测试的自动化程度和性能。把它们放在这里是为了在深入研究更复杂的技术之前做个铺垫。

22.2.1　代理模糊测试

我们讨论的第一种基于启发式规则的分析技术由一个私人开发[6]的模糊测试引擎所

[6] 读者可以搜索由 TippingPoint 的 Cody Pierce 发布的这个工具。

实现，该引擎名为 ProxyFuzzer。在典型的基于网络的客户端-服务器模型中，客户端和服务器之间会直接进行通信，如图 22.1 所示。

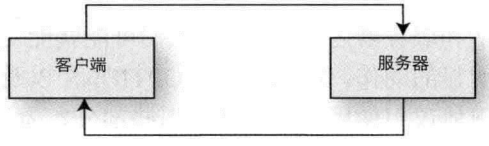


图 22.1　典型的客户端-服务器通信路径

正如其名称所暗示的，代理模糊测试器在客户端和服务器之间的连接上充当中继。为了有效地完成这一任务，客户端和服务器必须被手工配置[7]为指向代理服务器。换句话说，客户端把代理当作服务器，而服务器把代理当作客户端。包括代理模糊测试器的新网络结构如图 22.2 所示。

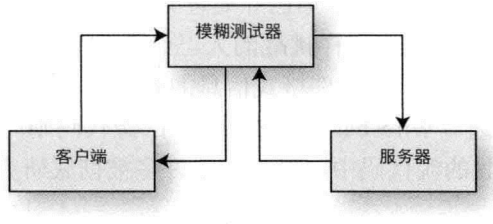


图 22.2　透明的在线代理

图 22.2 中节点之间的实线表示原始未修改的数据，这些数据通过代理转发。默认模式下，ProxyFuzzer 将接收到的数据透明地中继到对端，同时用便于程序修改或数据重放的格式记录通信数据。这个特性非常方便，使得研究者可以跳过生成数据和修改抓取到的 PCAP 文件的中间步骤。顺便说一句，修改和重放录制得到的数据是一种很好的对未知协议进行逆向工程的人工方法，Matasano 的协议调试器（PDB）[8]中已经很好地实现了该方法。

成功配置好 ProxyFuzzer 并确认它能工作后，可以打开它的自动变异引擎开关。在自动变异模式下，ProxyFuzzer 会在通过它转发的通信数据中找出纯文本的组件。ProxyFuzzer 找出纯文本组件的方法是从数据中寻找落在有效 ASCII 范围内的、超过给定长度的字符序列。然后，模糊测试器会用 ASCII 有效范围外的字符和格式字符串随

[7] 实际上这个过程可以通过 IP 欺骗自动地进行，为了简单起见在这里我们不讨论这种方式。

[8] http://www.matasano.com/log/399/pub-blackhat-talk-materials-as-promised/

机拉长或修改这些纯文本组件。图 22.3 中以虚线方式显示了变异流。

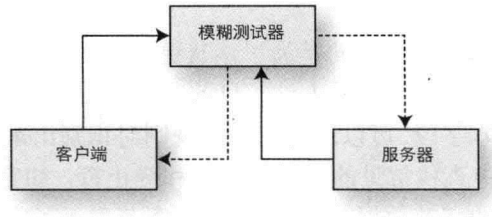


图 22.3　自主的内联变异

ProxyFuzzer 非常简单且特别易于使用，它能够辅助手工分析，甚至，仅凭其当前的形态就已经发现过安全问题！也许，这种内联（代理）模糊测试的最大好处是能够原封不动地保持协议的大部分内容，而仅对协议的某些字段进行变异。在保持大多数传输数据原封不动的情况下，只需很少的分析就能成功地进行模糊测试。例如，考虑带数据层序列号的复杂协议。如果数据包包含无效的序列号，处理程序立即就会检测到协议破坏，停止解析数据包中剩下的数据（包括被模糊测试的字段）。使用内联（代理）方式，模糊测试器就能够利用合法的交易数据，生成有效的变异数据使得数据顺利到达解析器。

424

ProxyFuzzer 发现的安全问题

即使与前面章节中我们介绍过的最基础的模糊测试器相比，ProxyFuzzer 也显得初级（rudimentary）。然而，它已经在流行的企业软件（显然，是编码有问题的那些）上证明了自己的有效性。在完全无须设置和使用其他技术辅助的情况下，ProxyFuzzer 在 Computer Associates 的 Brightstore 备份软件中发现了两个可被远程利用的漏洞。

第一个问题发生在后认证事务（post authentication transacation）中，因此影响不算太严重。在通过 TCP 端口 6050 的标准客户端-服务器通信中，长文件名会触发 UnivAgent.exe 中的一个可被利用的栈溢出漏洞。

第二个问题存在于文件 igateway.exe 这个 HTTP 后台程序中，该后台程序通过 TCP 端口 5250 通信。这个问题只在打开调试开关的情况下才会出现，因此也不算太严重。igateway.exe 会将用户请求的文件名不安全地传入到 fprintf() 调用中，从而导致一个可被利用的格式字符串漏洞。

如果和那些非常严重的漏洞相比，ProxyFuzzer 发现的这两个漏洞都不显得特别“性感”。但是，考虑到只需要极少的付出就能发现这些缺陷，也许 ProxyFuzzer 仍然值得一试。

22.2.2 改进的代理模糊测试

目前对 ProxyFuzzer 的改进方向是为了让其更“聪明”，使用启发式规则辅助进行自动字段检测和数据变异是其中一种方法。目前 ProxyFuzzer 中的数据解析器能够从二进制协议中分离出纯文本字段。可以应用启发式规则对识别出来的纯文本字段进行进一步处理，从纯文本字段中查找常见的字段分隔符和终止符，如空格、逗号等。还可以搜索检测到的字符串前方的字节数据，检查是否某些字节表示了字符串长度和类型（例如，TLV 字段），并通过在多个数据包中进行交叉检查来验证找到的长度前缀是否正确。位于字符串后方的字节则可能是用于填充静态字段的数据。数据流中经常会出现客户端和服务器的 IP 地址（在 TCP/IP 头之外）。基于这个已知信息，分析引擎能够扫描网络数据，查找以字节格式和 ASCⅡ格式表示的 IP 地址。

假定大多数简单协议都可以被描述为一个字段序列，组成这个序列的字段类别列于图 22.4 中。基于这个假定，我们就可以开始使用一些猜测和验证方法来对未知协议的层次结构进行建模。

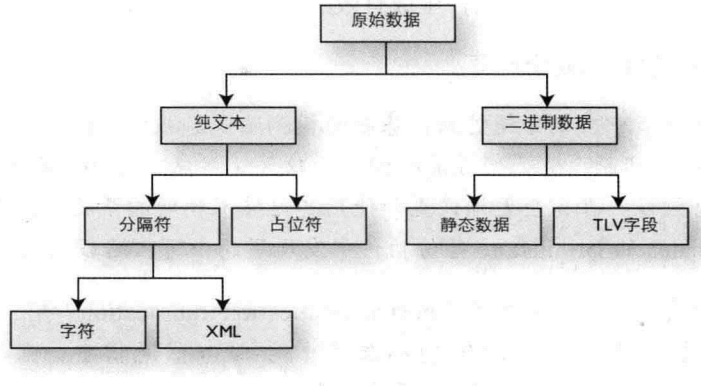


图 22.4 分层的协议分解

在面对复杂的二进制协议如 SMB 时，这种初级的技术可能起不到什么作用。然而，我们应用这种方法的目的并不是进行完整的协议分析，而是扩展适合进行自动化模糊测试的简单协议的范围。考虑下面这个例子中人为制造的网络数据，每一行数据是一个事务，由以竖线分隔的十六进制字节数据组成。

```
|00 04|user|00 06|pedram|0a 0a 00 01|code rev 254|00 00 0000 be ef|END
|00 04|user|00 04|cody|0a 0a 00 02|code rev 11|00 00 000000 de ad|END
|00 04|user|00 05|aaron|0a 0a 00 03|code rev 31337|00 00 00 c0 1a|END
```

一个人（也许一只海豚都能做到）只需要扫一眼这三行数据，就能快速地对协议的规范做出合理的猜测。让我们看看改进后的自动分析方法在彻底检查第一个事务后能发现什么。首先，自动分析例程会迅速发现一个 4 字节的 IP 地址字段（0a 0a 00 01 = 私有 IP 地址 10.10.0.1）。接下来，自动分析例程会发现 4 个 ASCⅡ字符串：user、pedram、code rev 254 及 END。每个字符串前面的字节值会被当成长度为 1 字节、2 字节、3 字节、4 字节的有效长度值来扫描。分析例程现在会猜测这个协议以两个可变长度的字符串开始，这两个字符串的前缀是 2 字节的长度值数据，接下来是连接中的客户端 IP 地址。在为第 3 个字符串扫描前缀的长度字段时不会得到任何结果。因此，分析例程会假设第 3 个字符串是静态大小的 ASCⅡ字段，在发现字符串后面的 4 个 null 字节后，分析例程会猜测这个字段是一个 10 字节定长的 ASCⅡ字段。最后一个字符串 END，既不包含长度前缀，后面也没有填充字符。分析例程会假定这个字段是一个 3 字节定长的 ASCⅡ字段。对于剩余的两个字节 0xbeef，分析例程无法做出猜测。这两个字节可能是定长的字段，也可能是变长的字段，需要进一步分析才能明确。结合我们以上所有的分析，分析引擎应该能从第一个事务数据中归纳得到下面的协议结构：

- 字符串长度描述（2 字节），然后是一个字符串。
- 字符串长度描述（2 字节），然后是一个字符串。
- 4 字节客户端 IP 地址。
- 10 字节定长的以 null 字符作为填充符的 ASCII 字符串。
- 未知（定长或不定长的二进制字段）。
- 3 字节定长的 ASCⅡ字符串。

分析引擎会继续处理后续事务，以测试第一次分析中得到的猜测。通过测试，我们能够确认某些猜测，发现另一些不正确的猜测，所有这些操作都会继续提升我们所建立的协议模型的准确性。不同于其他简化的例子，这种基于启发式的方法对于更为复杂的协议会失败。记住，我们的目的不是完全地理解被测协议，而是改进自动化的模糊测试。

22.2.3　反汇编启发式技术

应用汇编/反汇编层面的启发式技术来辅助进行更有效的模糊测试是一个可行的概念，但目前还没有任何公开可用的模糊测试工具或框架应用这个概念。这个概念本身很简单：在进行模糊测试时，使用运行时插装工具（例如调试器）监视被测目标上的代码执行。在调试器中，寻找静态字符串和整数比较操作。然后，将这些信息传回给模糊测试器，为以后生成测试用例提供参考。对每个案例而言，找到的结果都会不同；然而，

毫无疑问，模糊测试器能够合理地利用反馈循环生成更聪明的数据。考虑下面的来自某实际产品服务器软件的代码片段：

```
0040206C call ds:__imp_sscanf
00402072 moveax, [esp+5DA4h+var_5CDC]
00402079 add esp, 0Ch
0040207C cmpeax, 3857106359        ; string prefix check
00402081 jzshor loc_40208D
00402083 push offset 'string'      ; "access protocol error"
00402088 jmp loc_401D61
```


在这段代码中，原始网络数据通过 sscanf() API 被转换为整数。转换得到的整数会和静态整数值 3857106359 进行比较，如果两个值不同，协议解析器返回一个“协议访问错误”。当模糊测试器首次遍历这块代码时，调试器组件会发现这个静态整数值并将其传入反馈循环。随后，模糊测试器会把各种形式的这个值放入测试用例中，以期望能尽可能覆盖更多的目标应用的代码。如果没有反馈循环，这个模糊测试器只会是一潭死水。

使用 PaiMei[9]逆向工程框架的 PyDbg 组件，只需数小时就能开发出一个实现了基本概念的目标监视模糊测试器的反馈调试器。从 http://www.fuzzing.org 上可以下载得到相关的源代码。

在描述了以上这些基础的技术后，接下来让我们看一些更高级的，应用生物信息学（bioinfomatics）的方法自动分析协议的技术。

22.3 生物信息学

在 Wikipedia 上，生物信息学（又称计算生物学）是一个宽泛的术语，该术语指“利用应用数学、信息学、统计学和计算机科学的方法研究生物学的问题（通常在分子层次上）[10]”。从本质上来说，生物信息学给出了一些用来发现复杂但结构化的数据序列（例如基因序列）中的模式的技术。网络协议也可以被看作是由结构化数据组成的长序列。那么，软件测试人员是否可以借鉴生物学来简化模糊测试？

生物信息学中最基础的分析是排列两个序列（不管长度是否相同），并找出它们的

---

[9] http://openrce.org/downloads/details/208/PaiMei

[10] http://en.wikipedia.org/wiki/Bioinfomatics

最大相似度。在排列两个序列时，可以向序列中插入空格。考虑下面两个氨基酸序列：

```
序列 1：ACAT    TACAGGA
序列 2：ACATTCCTACAGGA
```

向第一个序列中插入三个空格可以实现长度对齐，并能使它们具有最大的相似度。有不少算法可以完成这类以及其他对齐序列的任务。Needleman-Wunsch（NW）算法[11]就是这类算法之一，该算法由 Saul Needleman[12]和 Christian Wunsch[13]于 1970 年提出。NW 算法通常应用在生物信息领域，用来对齐两个蛋白质或核苷酸序列。该算法属于“动态规划”领域，是一种将大问题分解为一系列小问题进行解决的方法。

世界上第一个应用生物信息理论进行网络协议分析的公开演示也许是 2004 年年底加尼福利亚州圣地亚哥召开的 ToorCon[14]黑客大会上的那次。在那次会议上，Marshall Beddoe 发布了一个试验性的名为 Protocol Informatics（PI）的 Python 框架，该框架引起了轰动，甚至引来了 *Wired* 网站的关注[15]。PI 的原始网站已经不能访问了，该框架也不再有人维护了。目前 PI 的作者已经被一家名为 Mu Security[16]的安全分析公司雇用，据猜测，该技术由于要提供给商业使用，因此不再公开了。幸运的是，目前仍然可以从 Packet Storm[17]网站下载得到一份经过 beta 测试的 PI 框架的副本。

PI 框架的目标是通过分析大量观察到的数据，自动推断协议的字段边界。PI 已经成功地从 HTTP、TCMP 和 SMB 协议中识别出了协议字段。PI 框架应用了 Smith-Waterman[18]（SW）本地序列对齐算法、NW 全局序列对齐算法、相似矩阵和进化树方法。虽然 PI 框架所使用的这些生物信息学技术的具体细节已经超出了本书的范围，但我们仍然会给出这些技术的简介，或至少列出算法的名字。

网络协议能够包含许多截然不同的消息类型。在不同消息类型之间尝试序列对齐是得不到结果的。为了解决这个问题，首先会在一对序列上应用本地序列对齐的 SW 算法，

428

[11] http://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm

[12] http://en.wikipedia.org/wiki/Saul_Needleman

[13] http://en.wikipedia.org/wiki/Christian_Wunsch

[14] http://www.toorcon.org

[15] http://www.wired.com/news/infostructure/0,1377,65191,00.html

[16] http://www.musecurity.com

[17] http://packetstormsecurity.org/sniffers/PI.tgz

[18] http://en.wikipedia.org/wiki/Smith-Waterman

定位和匹配相似的子序列。这些匹配到的位置和匹配到的序列对用于辅助 NW 全局序
429 列对齐。相似矩阵用于辅助 NW 算法，优化序列对齐。两个常用矩阵，接受度变异矩阵（Percent Accepted Mutation，PAM）和块置换矩阵（Blocks Substitution Matrix，BLOSUM），被 PI 框架用来进行更合适的、基于数据类型的序列对齐操作。在实际应用中，这种方法可以归纳为“将二进制数据与其他二进制数据对齐，将 ASCⅡ数据与其他 ASCII 数据对齐”。这种区分允许在协议结构中更精确地识别可变长字段。在此基础上，PI 框架进一步进行多种序列对齐，以更好地理解目标协议。为了规避直接应用 NW 的不可计算性，PI 框架使用无加权成对算术平均数（Unweight Pairwise Mean by Arithmetic Averages，UPGMA）算法生成进化树，然后将该进化树作为启发式向导来执行多种对齐操作。

下面，我们通过检查 PI 框架对一个简单的定长协议 ICMP[19]的分析过程，来了解该框架是如何工作的。这里列出了使用 PI 框架分析 ICMP 的基本步骤。首先需要收集一些 ICMP 包以供分析：

```
# tcpdump -s 42 -c 100 -n1 -w icmp.dumpicmp
```

接下来，将抓取到的数据传入 PI 框架：

```
# ./main.py -g -p ./icmp.dump
Protocol Informatics Prototype (v0.01 beta)
Written by Marshall Beddoe<mbeddoe@baselineresearch.net>
Copyright (c) 2004 Baseline Research
Found 100 unique sequences in '../dumps/icmp.out'
Creating distance matrix .. Complete
Creating phylogenetic tree .. Complete
Discovered 1 clusters using a weight of 1.00
Performing multiple alignment on Cluster 1 ..complete
Output of cluster 1
0097 x08 x00 xad x4b x05 xbe x00 x60
0039 x08 x00 x30 x54 x05 xbe x00 x26
0026 x08 x00 xf7 xb2 x05 xbe x00 x19
0015 x08 x00 x01 xdb x05 xbe x00 x0e
0048 x08 x00 x4f xdf x05 xbe x00 x2f
0040 x08 x00 xf8 xa4 x05 xbe x00 x27
0077 x08 x00 xe8 x28 x05 xbe x00 x4c
0027 x08 x00 xc3 xa9 x05 xbe x00 x1a
```

[19] http://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

```
0087 x08 x00 xdd xc1 x05 xbe x00 x56
0081 x08 x00 x88 x42 x05 xbe x00 x50
0058 x08 x00 xb0 x42 x05 xbe x00 x39
0013 x08 x00 x3e x38 x05 xbe x00
0067 x08 x00 x99 x36 x05 xbe x00 x42
0055 x08 x00 x0f x56 x05 xbe x00 x36
0004 x08 x00 xe6 xda x05 xbe x00 x03
0028 x08 x00 x83 xd9 x05 xbe x00 x1b
0095 x08 x00 xc1 xd9 x05 xbe x00 x5e
0093 x08 x00    xb6 x05 xbe x00 x5c
[ 省去一部分输出 ]
0010 x08 x00 xd1 xb6 x05 xbe x00
0024 x08 x00 x11 x8f x05 xbe x00 x17
0063 x08 x00 x11 x04 x05 xbe x00 x3e
0038 x08 x00 x37 x3b x05 xbe x00 x25
DT  BBB ZZZ BBB BBB BBB BBB ZZZ AAA
MT  000 000 081 089 000 000 000 100
Ungapped Consensus:
CONS x08 x00 x3f x18 x05 xbex00 ???
DT   BBB ZZZ BBB BBBBBB ZZZ AAA
MT   000 000 081 089 000 000000 100
Step 4: Anapyze Consensus Sequence
Pay attention to datatype composition and mutation rate
Offset 0: Binary data, 0% mutation rate
Offset 1: Zeroed data, 0% mutation rate
Offset 2: Binary data, 81% mutation rate
Offset 3: Binary data, 89% mutation rate
Offset 4: Binary data, 0% mutation rate
Offset 5: Binary data, 0% mutation rate
Offset 6: Zeroed data, 0% mutation rate
Offset 7: ASCII data, 100% mutation rate
```

虽然输出的结果不是那么直观，但 PI 框架生成的分析结果可以被翻译成下面的协议结构：

```
[1 字节] [1 字节] [2 字节] [2 字节] [1 字节] [1 字节]
```

这个结果和真正的协议结构之间已经相去不远了：

```
[1 字节] [1 字节] [2 字节] [2 字节] [2 字节]
```

PI 框架在识别最后一个字段时犯了错误，因为我们抓取到的供分析用的 ICMP 包数量有限。PI 识别错误的字段实际上是一个 16 位的序列号。因为我们只用了 100 个数

431 据包，PI 识别错误的这个字段的最显著的字节从未增长。如果向 PI 传入更多的数据，PI 应该可以正确地识别出最后一个字段。

应用生物信息学进行自动协议分析是非常有趣的高级方法。目前这方面取得的成果有限，相当多的研究者对于应用这些技术能够得到的收益持怀疑态度[20]。但无论如何，PI 框架已经提供了这方面的成功证据，我们期望看到这个方法将来能够有所发展。

22.4 遗传算法

遗传算法（Genetic Algorithm，GA）是模拟进化软件使用的一种近似搜索技术。遗传算法在初始种群上进行变异，应用自然选择来选择更能“适应”环境的样本，在后代中保留样本中优秀的遗传特性。被选择的样本通过交配和变异产生后代。对遗传算法而言，通常需要定义以下三方面的内容。

- 表示方法：解决方案（个体）的表示方法。
- 适应度函数：用来评估得到的方案（个体）对环境适应度的函数。
- 生产函数：负责变异和让两个方案（个体）交配的函数。

为了更好地说明这些概念，我们来用遗传算法解决一个简单的问题。我们要解决的问题是得到一个长度为 10 的，包含最多的“1”的二进制数字串。这个问题的表示方法和适应度函数显而易见：可能的解决方案会被表示为一个二进制数字串，而适应度函数会被定义为计算二进制数字串中“1”出现的次数。对生产函数（或交配函数）而言，我们随意地将其定义为“在二进制数字串的位置 3 和位置 6 处交换两个串，并在后代中随机地改变（翻转）一个位的值”。这个生产函数也许不是最有效的，但它能够满足我们的需要。生产函数中的交换规则允许父母传入遗传信息，而其中的位翻转负责随机变异。这个例子中的遗传算法的进化过程如下：

1. 从随机生成的一群方案（个体）开始。
2. 在每个方案（个体）上应用适应度函数进行计算，选择适应度最好的。

432 3. 在选定的方案（个体）上应用生产（交配）函数。

4. 用生成的后代取代父母，并继续这个过程。

---

[20] http://www.matasano.com/log/294/protocol-informatic/

为了看到这个例子是如何工作的，我们从 4 个随机生成的二进制数字串（个体）开始并计算每个个体的适应度：

```
0100100000        2
1000001010        3
1110100111        7
0000001000        1
```

中间的这两个二进制数字串（粗体显示）具有最高的适应度，因此会被选择用于产生下一代（幸运的家伙）。在二进制数字串的位置 3 处进行交换，生成一对后代，在二进制数据的位置 6 处交换生成另一对后代：

```
1000001010        3        100 + 0100111 -> 1000100111
1110100111        7        111 + 0001010 -> 1110001010

1000001010        3        100000 + 0111 -> 1000000111
1110100111        7        111010 + 1010 -> 1110101010
```

在生成的后代上应用随机变异（以粗体显示发生变异的位），并再次应用适应度函数计算其适应度：

```
1000100111   ->       1010100111        6
1110001010   ->       1110000010        4
1000000111   ->       1000001111        5
1110101010   ->       1110101110        7
```

我们可以看到新生代的平均适应度已经提高了，遗传算法给出了成功的结果。在这个例子中，我们使用的是固定的变异率。在更高级的例子中，如果生成的后代在一段时间内没有进化，可以选择自动提高变异率。注意，遗传算法被认为是带有随机性的全局优化器。换句话说，该算法中存在随机的因素，因此输出会持续发生变化。然而，虽然遗传算法会持续寻找更好的方案，但无论让它执行多长时间，它也不一定能找到最佳的方案（在我们的例子中，最佳方案是一个全为 1 的二进制数字串）。

佛罗里达中央大学（University of Central Florida，UCF）的一个小组最近在研究用遗传算法改进模糊测试，并在黑帽美国 2006 的美国安全大会[21]上展示了它们在这方面的研究。佛罗里达中央大学的这个小组展示了一个名为 Sidewinder 的概念性工具，该工具能够自动地设计输入，强制被测应用执行设计好的路径。这个案例中的遗传算法方

433

[21] http://www.blackhat.com/html/bh-usa-06/bh-06-speakers.html#Embleton

案（个体）是生成的模糊测试数据，以上下文无关的文法表示[22]。该工具以非传统的模糊测试方法定义适应度函数，这种非传统的模糊测试方法不生成测试数据和对服务进行监视，而是首先静态地定位可能的漏洞代码位置，例如，发生不安全的 API 调用（例如 strcpy）的位置。这个步骤类似于我们在前一章讨论的 Autodafé定位代码点，在标记（markers）上增加权重的方法。接下来，Sidewinder 工具检查整个目标二进制数据的控制流图，提取出套接字数据入口点（recv 调用的位置）之间的子图和可能的漏洞代码的位置。图 22.5 给出了一个包含这些点的控制流图的示例。在第 23 章“模糊测试器跟踪”中我们会给出关于控制流图的更详细的定义和解释。

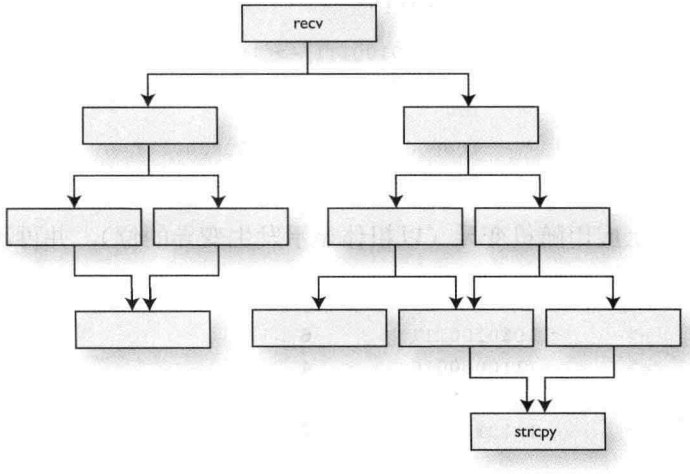


图 22.5　包含了潜在漏洞的控制流图

下一步，Sidewinder 工具识别出从数据入口点到目标漏洞代码的所有路径上的所有节点。图 22.6 展示的是与图 22.5 相同的控制流图，其中位于连接路径上的节点被以黑色突出显示。

434

接下来，Sidewinder 识别出控制流图中的出口节点。出口节点是刚好处于连接路径之外的边界节点。如果执行到了出口节点，则任何指向有漏洞代码的可能路径都不会被执行到。图 22.7 显示的控制流图与图 22.6 相同，其中位于连接路径的节点以黑色突出显示，出口节点以白色突出显示。

[22] http://en.wikipedia.org/wiki/Context-free_grammar

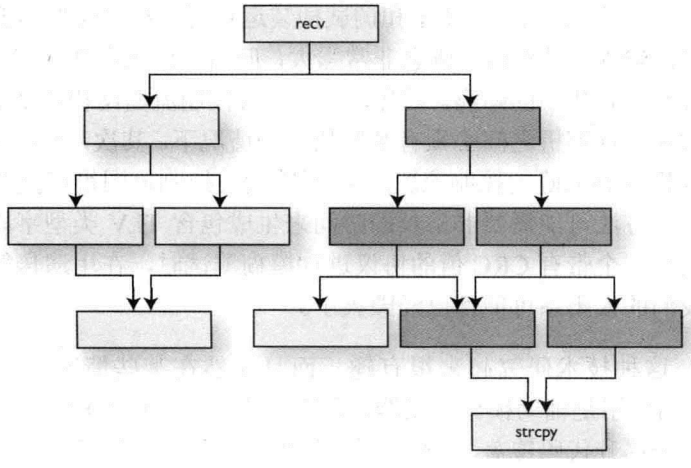


图 22.6　突出显示了连接路径的控制流图

有了最后这个高亮的控制流图后，就可以定义适应度函数了。佛罗里达中央大学的这个小组使用了马尔科夫过程，基于遍历特定路径的可能性来计算适应度。从应用角度简单地说，适应度函数的计算包含下面这些步骤。将一个调试器附着在目标进程上，并在入口节点、目标节点、出口节点及连接路径上的所有节点上设置断点。到达这些断点时，监视执行情况，评估给定输入（方案）在运行时覆盖我们期望的路径的进度。这种跟踪一直进行到到达出口节点。模糊测试器生成输入，将其发送给目标进程。选择具有最大“适应度”的输入进行生产，并继续这个过程，直到成功到达目标节点或发现指定的潜在漏洞。

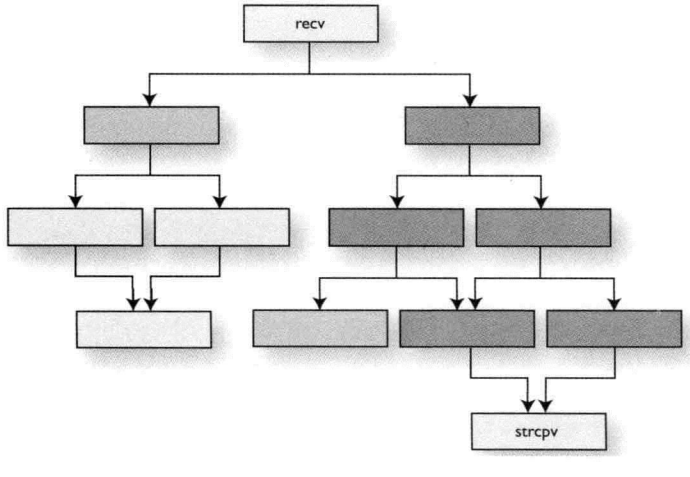


图 22.7　突出显示了出口节点的控制流图

Sidewinder 结合了静态分析、图论和调试插装运行时分析来强制得到能够到达目标进程中任意位置的输入。虽然这个概念非常强大，但当前仍然存在许多限制，使得这个概念离完美的方案仍有相当的距离。首先，不是所有控制流图结构都适合遗传算法，特
435 别是在图形结构和数据解析之间必须有某些依赖的情况下。其次，不是在所有情况下都可以通过静态分析得到准确的控制流图。而不准确的控制流图很可能会导致整个过程失败。最后，这种方法可能需要非常长的时间来生成包含 TLV 类型字段的协议的合法数据。例如，当对一个带有 CRC 值的协议进行模糊测试时，在用遗传算法生成任何有用的结果前，我们的太阳系可能都已经毁灭了。

总而言之，这种技术研究起来很有趣，而且显然在某些情况下也有价值。也许 Sidewinder 最大的作用是辅助模糊测试器，在需要的时候通过对输入进行变异，以遍历一个小的子路径来提升代码覆盖，而不是用于发现一个能够遍历从入口到目标的整个连接路径的输入。

22.5 小结

在本章的开始，我们提出模糊测试中最困难的方面是克服理解目标协议或文件格式和对其建模的障碍。接下来，我们给出了三种方法，用于辅助进行手工和自动化的协议
436 分析。在本章中，我们对启发式方法、生物信息学方法及遗传算法都进行了描述，并将其作为可能的方案进行了评价。本章展示的这些概念都是模糊测试研究的前沿概念，每天都有新的进展，读者可以访问 http://www.fuzzing.org 来获得更实时的信息与资源。

第23章

模糊测试器跟踪

"Well, I think if you say you're going to do something and don't do it, that's trust-worthiness."

——George W. Bush, in a CNN online chat, August 30, 2000

在前面的章节中，我们定义了模糊测试，讨论了模糊测试目标，列出了各种模糊测试类型，讨论了各种生成数据的方法。但是，我们还没有讨论如何跟踪各种模糊测试器的执行情况。目前，模糊测试器跟踪的观念还没有在安全社区中得到足够重视。据我们了解，当前还没有可用的商业或免费的模糊测试器解决了这个问题。

在本章中我们将定义模糊测试器跟踪，该技术也被称为代码覆盖。我们将在本章中讨论模糊测试器跟踪的益处，探索各种实现模糊测试器跟踪的方法。在本书的后面部分我们将会创建一个实现了模糊测试器跟踪功能的原型，该原型能够结合我们前面开发的模糊测试器，提供更为强大的分析能力。

23.1 我们跟踪的究竟是什么

在程序的最低层次上，CPU 执行的是汇编指令，或其他它所能理解的命令。不同的计算机体系结构使用不同的指令集。例如，我们熟悉的 IA-32 / x86 体系结构是一种复杂指令集计算机（Complex Instruction Set Computer，CISC）[1]体系结构，该体系结构

[1] http://www.intel.com/intelpress/chapter-scientific.pdf

提供了超过 500 条指令，这些指令为 CPU 提供了从执行基础的数学运算到内存操作的能力。与此不同，SPARC 采用精简指令集计算机（Reduced Instruction Set Computer，RISC）体系结构，仅提供不到 200 条指令[2]。

CISC 与 RISC

CISC 由 IBM 的研究员在 20 世纪 70 年代开发，CISC 的定义是相对 RISC 而言的，用以和 RISC 的设计理念进行区分。CISC 和 RISC 体系结构在设计理念上的区别是：RISC 速度更快，生产成本更低；而 CISC 处理器较少调用主存，用较少的代码就能完成 RISC 处理器上同样的任务。随着狂热的 Mac 用户数量的增加，关于这两种架构谁更出色的争论最近达到了一个新高峰，直到最近，PowerPC RISC 处理器仍然是 Mac 用户心目中的神。

有趣的是，尽管 x86 处理器上有大量的复杂指令，但只有大约一打指令的执行时间超过一半指令的平均执行时间[3]。

不管你的工作环境是 CISC 还是 RISC，CPU 上实际执行的指令大多数时候都不是由程序员手写的，而是从高层语言，如 C 和 C++语言，通过编译或翻译过程变成底层语言的。通常情况下，你操作的一般规模的程序都会包含数万条或数百万条指令。与软件的每次交互都会导致指令的执行。点击鼠标的动作、处理网络请求、打开一个文件等都需要执行数条指令。

如果要对一个 IP 语音（Voice over IP，VoIP）软件电话（softphone）进行模糊测试审计，你可以采用某些方法导致其崩溃，那么，我们怎样才能知道哪个指令集导致了软件错误？目前，研究者使用的典型方法是事后分析方法，通过检查日志，以及使用调试器来找到错误的位置。另一种方法是使用事前方法，主动监视我们生成的模糊测试请求引发了哪些指令的执行。有些调试器（例如 OllyDbg）实现了跟踪功能，允许用户跟踪所有执行的指令。不幸的是，这种跟踪功能通常需要消耗非常多的系统资源，而且需要大量的人力，因此实用性不强。以我们的 VoIP 软件电话为例，如果软件电话发生崩溃，我们可以回顾已被执行的指令，检查错误发生在什么位置。

439

编写软件时，一种最基础的调试形式是在代码中插入打印语句。在代码执行时，这

---

[2] 这里给出的体系结构提供的指令的确切数量不一定准确，但提到的概念是成立的。CISC 体系结构实现的指令数多于 RISC 体系结构。

[3] http://www.openrce.org/blog/view/575

种方法使得开发人员能够知道执行到了什么位置，以及代码的各个逻辑块的执行顺序。我们想要完成的跟踪任务从根本上来说与此相同，只不过是在更低的层次（CPU 指令层次）上。在 CPU 指令层次上，我们无法在指令中随意插入打印语句。

我们将**在处理各种请求时监视和记录请求真正执行的指令的过程**定义为“模糊测试器跟踪”。描述这个过程的一个更通用的术语是你可能早已熟悉的“代码覆盖”。我们将在本章中交替使用这两个术语。读者马上就会看到，模糊测试器跟踪是一个非常简单的过程，这个过程为我们提供了一些机会来改进我们的模糊测试分析。

23.2　可视化和基础块

在我们讨论代码覆盖时，一个基本概念是理解用可视化方式展现二进制数据，在设计和开发模糊测试器跟踪器时，我们将会用到这一概念。可执行文件反汇编得到的结果能被可视化为图，名为调用图（call graph）。调用图将每个函数表示为一个节点，将函数间的调用表示为节点之间的边。在查看可执行文件的结构和各个函数之间的关系时，调用图是一种有用的抽象方式。考虑图 23.1 所示的这个人为设计的例子。

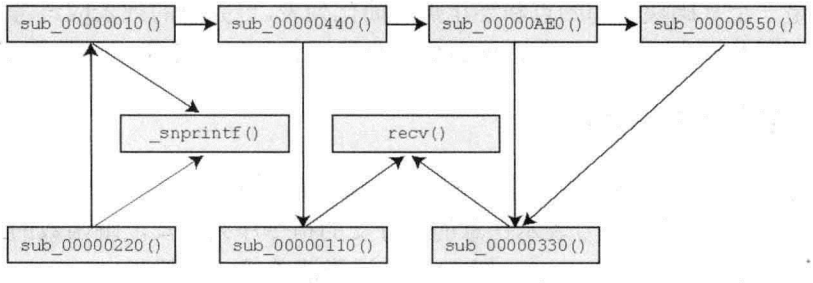


图 23.1　简单的调用图

图 23.1 中的大多数节点包含一个以 sub_开头的标签，sub_的后面跟着 8 个字符的十六进制数字，这些数字表示子例程在内存中的地址。DataRescue Interactive Disassembler Pro（IDA Pro）[4]使用的就是这种名称转换方式，无须符号信息就能自动生成函数名。其他反汇编引擎也使用类似的命名模式。当分析闭源二进制应用时，由于缺少符号信息，反汇编得到的大部分函数都会以这种方式命名。图 23.1 中有两个节点 sprintf() 和 recv() 带有符号名称，所以被打上了正确的标签。只要扫一眼这个调用图，根据未命名的子例程 sub_00000110() 以及 sub_00000330() 与 recv() 例程间的调用关

[4] http://www.datarescue.com

系，我们立即就可以知道这两个例程是负责处理网络数据的。

23.2.1 控制流图

反汇编的函数也可以被可视化为控制流图（Control Flow Graph，CFG）。控制流图将每个基础块表示为节点，基础块之间的分支表示为节点间的边。这自然引出了一个问题：什么是基础块？我们将基础块定义为一个指令序列，一旦执行到了基础块的第一条指令，则块中的每条指令都一定会被顺序运行。更详细地说，基础块的起点通常符合下面这些条件：

- 函数的开始位置。
- 分支指令的目标位置。
- 一条分支指令之后的指令。

基础块通常以下面的条件终止：

- 一条分支指令。
- 一条返回指令。

以上是一个基础块的起始点和终止点的简化列表。更完整的列表应该考虑对无返回值函数的调用及异常处理。然而，就我们的目的而言，以上定义已经足够了。

441

23.2.2 控制流图示例

为了透彻地理解控制流图，考虑下面这个来自假想的子例程 `sub_00000010` 的指令片段。图 23.2 中的省略号表示任意的无分支的指令序列。反汇编函数的这种视图也被称为反汇编清单（dead listing），图 23.2 是这类清单的一个例子。

```
00000010 sub_00000010
00000010 push ebp
00000011 mov ebp, esp
00000013 sub esp, 128h
…
00000025 jz'00000050
0000002B mov eax, 0Ah
00000030 mov ebx, 0Ah
…
00000050 xor eax, eax
00000052 xor ebx, ebx
…
```

图 23.2　sub_00000010 的反汇编清单

根据我们前面给出的规则，将反汇编清单分解为基础块是一个简单的任务。第一个基础块从函数的起始处（地址 0x00000010）开始，到第一条分支指令（地址 0x00000025）结束。分支指令的下一条指令（地址 0x0000002B），以及分支指令指向的目标地址 0x00000050 均标识着新基础块的开始。被分解为基础块后，图 23.2 的这个子例程的控制流图如图 23.3 所示。

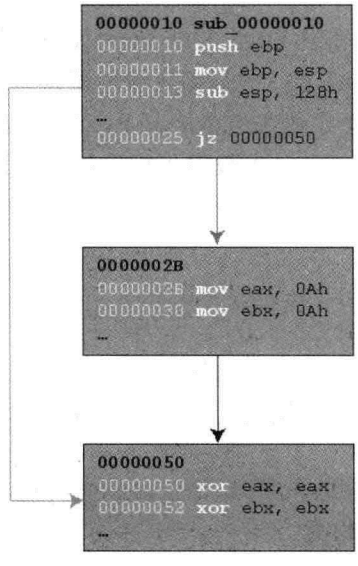


图 23.3　sub_00000010 的控制流图

控制流图这种有用的抽象能够查看函数内的各种控制流路径，并能够指出函数中的逻辑循环。关于控制流图，要记住的是每个指令块中的指令确定地会一起执行。换句话说，如果块中的第一条指令被执行了，块中的每条指令都会被执行。稍后，我们会看到这个知识能够帮助我们开发模糊测试器跟踪器时采用必要的捷径。

23.3　构建一个模糊测试器跟踪器

442

我们可以采用多种方法来实现一个定制的代码覆盖工具。如果你已经读过了前面的章节，或许第一个跳入你的脑海中的方法是使用带跟踪功能的调试器，监视和记录每条被执行的指令。这种方法正是 OllyDbg 用来实现它的“调试/跟入”（debug\trace into）和“调试/跟出”（debug\trace over）的方法。按照下面的逻辑，利用第 20 章“自动化内存模糊测试”中用过的 PyDbg 库，我们也能够实现这个功能：

1．加载我们需要进行模糊测试的目标进程或是附着在已运行的测试目标上。

2．注册一个单步处理函数，设置单步标志。

3．在单步处理函数中，记录当前指令的地址并重新设置单步标志。

4．继续执行，直到该进程退出或发生崩溃。

443

在 PyDbg 的基础上，用不到 50 行的 Python 代码就能实现这个简化的跟踪方法。然而，这个方法并不是一个很好的方法，因为通过调试器为进程中的每条指令进行插装会大大增加延迟，这些延迟会导致在许多情况下，该方法不可用。如果不信，读者可以自行尝试从 http://www.fuzzing.org 下载得到相关代码并自行尝试。另外，单单确定和记录被执行的指令仅能覆盖我们总体目标的一个方面，为了生成更好的报告，需要关于目标的更多信息。

我们将使用一个三阶段方法实现模糊测试器跟踪器：

- 首先，对我们的测试目标进行静态分析。
- 第二步，实时跟踪和记录我们测试目标中的那些为响应我们的模糊测试而执行的指令。
- 第三步，也即最后一步，交叉参考生成的实时数据和来自第一个步骤的分析数据，生成有用的跟踪报告。

23.3.1 分析目标

假设我们已经有办法知道给定的应用中有哪些指令被执行，那么，利用手头上已经得到的信息，我们还会想要知道哪些关于应用的其他信息？例如，我们至少会想要知道我们的测试目标中总共有多少条指令。这样的话，我们就能够知道负责处理给定任务的代码占多大比例。理想情况下，我们想要有一个对指令集的更详细的分解：

- 目标中有多少条指令？
- 目标中有多少个函数？
- 目标中有多少个基础块？
- 目标中的哪些指令与标准支持库一致？
- 目标中的哪些指令是由我们的开发人员编写的？
- 每条指令是从哪些源代码翻译得到的？
- 什么指令和数据是可达的，并且能够被网络和文件数据影响？

可以使用多种工具和技术来回答这些问题。大多数开发环境有能力生成符号调试器信息，能够帮助我们更精确地回答这些问题。就我们的目标而言，我们假设源代码并不处于可用状态。

为了回答这些关键问题，我们基于 IDA Pro 的反汇编器建立了分析器（profiler）。我们可以从 IDA 轻松地收集指令和函数的统计数据。通过实现一个简单的基于基础块规则的算法，我们还可以轻松地获得每个函数中所包含的基础块列表。

23.3.2　跟踪

彻底地单步跟踪一个程序很消耗时间，而且也非必要，除了单步跟踪之外，我们还可以采用什么其他方法来跟踪指令的覆盖情况？在前面部分的定义中，一个基础块就是一个指令序列，序列中的每条指令保证一起按顺序执行。如果我们能够跟踪基础块的执行情况，实际上就能够跟踪到指令的执行情况。结合从调试器和从分析步骤收集得到的信息，我们可以完成这个任务。利用 PyDbg 库，我们能够用下面的逻辑跟踪基础块的执行情况：

1．加载模糊测试目标进程，或附着到运行中的目标进程上。

2．注册一个断点处理函数。

3．在每个基础块的开始处设置断点。

4．在断点处理函数中，记录当前指令地址，并恢复断点（可选）。

5．继续执行，直到进程退出或发生崩溃。

通过记录基础块，而不是单独指令的执行，我们能够大大地提升传统跟踪方法（例如用前文提到的 OllyDbg 实现的受限方法）的性能。如果我们只关心哪些基础块被执行到，那就不需要在第 4 步恢复断点。一旦某个基础块被执行到，我们就会在处理函数中记录下来，而不关心这个基础块在将来是否会被再次执行到，以及何时会被执行到。然而，如果我们对基础块的执行顺序感兴趣，就必须在第 4 步恢复断点。例如，如果在一个循环中一次又一次地执行到相同的基础块，那么，只有通过恢复断点的方式，我们才能记录每次的迭代执行。恢复断点为我们提供了更多上下文相关的信息。使断点失效能够让我们执行得更快。具体采用哪种方法依赖于需要完成的任务，所以在我们的工具中这应该是一个可配置的选项。

在跟踪大量基础块的情况下，跟踪速度也许不如我们期望得那么快。在这种情况下，
445 我们可以在更高的层面进行跟踪，从跟踪基础块的执行变为跟踪函数的执行。这只需要对我们的逻辑进行小小的改造即可：将断点设置在函数的开始位置，而不是设置在基础块上。同样，究竟应该采取何种方法可能随着每次运行条件的不同而不同，因此在我们的工具中，这也应该是一个可配置的选项。

简化的 PyDbg 单步代码覆盖工具

对感兴趣的读者，下面的 PyDbg 脚本演示了一个单步执行器的实现。在初始的附着和断点设置阶段，脚本将所有线程都设置为单步模式。随后，脚本为创建新线程注册了一个处理函数，处理函数将所有新创建的线程设置为单步模式。

在这个 PyDbg 单步执行器的开发过程中，唯一“技巧性”的地方是处理切换到内核时的上下文。在执行每个单步时，脚本检查当前指令是否为 sysenter。现代的微软 Windows 操作系统就使用 sysenter 指令作为从用户态切换到内核的“网关”。当遇到 sysenter 指令后，脚本会在内核返回的地址处设置一个断点，允许线程正常执行（也就是说，非单步模式）。

当线程从内核中返回后，将会恢复单步执行。感兴趣的读者可以参考以下的代码：

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

# 断点处理函数
def on_bp (dbg):
    ea      = dbg.exception_address
    disasm  = dbg.disasm(ea)

    # 将所有线程设置为单步模式
    if dbg.first_breakpoint:
        for tid in dbg.enumerate_threads():
            handler = dbg.open_thread(tid)
            dbg.single_step(True, handle)
            dbg.close_handle(handle)

    print "%08x: %s" % (ea, disasm)
    dbg.single_step(True)
    return DBG_CONTINUE
```

446

```
# 单步处理函数
def on_ss (dbg):
    ea       = dbg.exception_address
    disasm   = dbg.disasm(ea)

    print "%08x: %s" % (ea, disasm)

    # 我们不能单步进入内核空间
    # 因此在内核的返回位置上设置断点并继续
    if disasm == "sysenter":
        ret_addr = dbg.get_arg(0)
        dbg.bp_set(ret_addr)
    else:
        dbg.single_step(True)

    return DBG_CONTINUE

# 创建线程处理函数
def on_ct (dbg):
    # 将新创建的线程设置为单步模式
    dbg.single_step(True)
    return DBG_CONTINUE

dbg = pydbg()
dbg.set_callback(EXCEPTION_BREAKPOINT,      on_bp)
dbg.set_callback(EXCEPTION_SINGLE_STEP,     on_ss)
dbg.set_callback(CREATE_THREAD_DEBUG_EVENT, on_ct)

try:
    dbg.attach(123)
    dbg.debug_event_loop()
except pdx, x:
    dbg.cleanup().detach()
    print x.__str__()
```

另一个要在我们的模糊测试器跟踪工具中实现的便捷的特性是“能够从新的记录数据中过滤掉以前记录到的代码覆盖”。可以利用该特性轻松地过滤掉处理我们不感兴趣任务的函数、基础块及指令。负责处理 GUI 事件（鼠标点击、菜单选择等）的代码是典型的我们不感兴趣的代码。稍后，我们将会看到这个特性是如何在代码覆盖上发挥作用,发现漏洞的。

23.3.3　交叉参考

在第三个，也就是最后一个步骤中，我们结合来自静态分析的信息和来自运行时跟

踪器的输出，生成准确和有用的报告，以回答下面这些问题：

- 哪些区域的代码负责处理我们的输入？
- 目标的百分之多少已经被覆盖了？
- 导致某个特定错误的执行路径是什么？
- 目标中的哪些逻辑判定没有被执行到？
- 目标中的哪些逻辑判定直接依赖于用户提供的数据？

列表中的最后两点是为了改进我们的模糊测试器，而不是为了在实际目标应用中找出特定的漏洞。我们会在本章后面讨论模糊测试器跟踪能带来的各种好处时进一步讨论这两点。

通过代码覆盖找到漏洞

2005 年 6 月 14 日，微软发布了一个安全漏洞。该漏洞影响 Outlook Express，在 Outlook Express 解析网络新闻传输协议（Network News Transfer Protocol，NNTP）的响应时可能导致远程可被利用的缓冲区溢出[5]。我们来看看在这个问题上，代码覆盖分析能够给漏洞研究者带来的好处。微软给出的安全报告的核心内容如下：

当 Outlook Express 被用作新闻组阅读器时，存在一个远程代码执行漏洞。攻击者可以通过建立一个恶意的新闻组服务器利用这个漏洞，如果用户向该恶意服务器请求新闻，这个恶意的新闻组服务器可能会允许远程代码执行。攻击者可以通过成功利用该漏洞来获得对用户系统的完全控制。但是，利用该漏洞的过程需要用户参与交互。

448

除此之外，报告中没有提供更多的信息。显然，对诸如入侵检测系统（IDS）或入侵防护系统（IPS）的开发者来说这些信息明显不够。根据这份报告，我们所知道的只是：缺陷发生在 Outlook Express 连接到一个恶意的 NNTP 服务器之后。让我们试试看是否能利用一个开源的代码覆盖工具，Process Stalker[6]，将这个问题的范围缩小一些。Process Stalker 的命令行选项和的用法描述超出了本书的范围，如果读者想要了解这个例子背后的具体细节，请参考 OpenRCE 上的文章[7]。

通过安装微软提供的补丁，并监视 Outlook Express 的安装目录中哪些文件发生了变化，我们可以知道漏洞代码一定存在于 MSOE.DLL 中。使用 Process Stalker IDA Pro

[5] http://www.microsoft.com/technet/security/bulletin/MS05-030.mspx

[6] https://www.openrce.org/downloads/details/171/Process_Stalker

[7] http://www.openrce.org/article/full_view/12

插件对这个模块进行分析，发现该模块总共包含大约 4800 个函数和 58000 个基础块。如果要手工地从这些数据中进行筛选，这肯定是个让人望而生畏的任务。为了缩小范围，我们可以利用 Process Stalker 来定位 Outlook Express 连接到 NNTP 服务器的相关代码路径。显然，这种方法能够有效地减小分析空间，但实际上我们还能做得更好。我们在前面描述过，每次与目标的交互都会导致指令的执行。在当前情况下，这意味着当我们通过 Internet Explorer 中的 "news://" URI 处理程序访问一个恶意的 NNTP 服务器时，我们的代码覆盖工具记录了一些不必记录的代码，例如负责启动 Outlook Express 的代码、负责绘制屏幕上显示元件（如窗口和对话框）的代码，以及负责处理用户驱动的 GUI 事件（如点击按钮）的代码。比如，当 Outlook Express 第一次连接到一个 NNTP 服务器时，会弹出如图 23.4 所示的对话框。

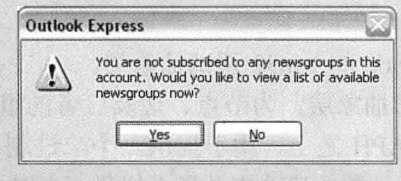


图 23.4　降低可被利用的漏洞的严重性等级的对话框

就我们的目标而言，创建、处理及销毁这个对话框根本不在我们的兴趣范围内。为了进一步缩小我们的分析范围，我们可以让 Process Stalker 仅记录负责处理各种 GUI 任务的代码。通过附着在 Outlook Express 上并与其交互，而不是实际连接到 NNTP 服务器的方式来可以实现这一目标。接下来，我们移除面向 GUI 的代码覆盖和连接服务器的代码覆盖之间的重叠部分，就能找出专门负责解析 NNTP 服务器请求的代码。执行完这些操作后，我们可以将关注范围减小到 91 个函数。在这 91 个函数包含的所有 1337（我们发誓这个数据是真实的）个基础块中，我们的服务器连接记录表明只有 747 个基础块会被真正地访问到。与开始时的 58000 个基础块相比，这极大缩小了分析范围，大大帮助了我们发现漏洞！

使用一个简单的脚本来定位负责移动服务器提供数据的基础块，可以进一步将基础块减少到极少的 26 个，这 26 个基础块中的第二个揭示了漏洞的准确细节：以空格分隔的任意长度的，服务器提供的数据被复制到一个静态的 16 字节栈缓冲区中。很好，现在我们可以利用这个漏洞……呃……不对，是防护这个漏洞，并准时回家欣赏最新一集的 "黑道家族"[8] 了。

[8] http://www.hbo.com/sopranos/

23.4 分析一个代码覆盖工具

我们前面用过 PyDbg 库，它实际上是一个名为 PaiMei 的大的逆向工程框架的一部分。PaiMei 是一个活跃的，开源的，可被自由下载的 Python 逆向工程框架[9]。PaiMei 基本上可以说是逆向工程师的瑞士军刀，它能够帮助进行多种高级的模糊测试任务，例如智能检测和异常处理，我们会在第 24 章“智能错误检测”中深入探讨这个主题。PaiMei 框架中带有一个名为 PAIMEIpstalker（也叫 Process Stalker 或 PStalker）的代码覆盖工具。虽然这个工具不能生成与你的期望完全一致的报告，但由于这个框架及其附带的应用都是开源的，因此我们可以通过修改代码来满足给定项目的特定需求。PaiMei 框架可以被分解为下面这些核心组件。

- PyDbg：一个纯 Python 的 win32 调试抽象类。
- pGRAPH：一个图形抽象层，为节点、边及二者的组合提供了单独的类。
- PIDA：建立在 pGRAPH 之上，用于提供一个二进制（DLL 和 EXE）层上的抽象和持久的接口，为函数、基础块及指令提供了单独的类。PIDA 工具的输出结果是一个可移植文件，加载该文件后，使用者可以任意遍历整个原始二进制文件。

在前面的章节中我们已经熟悉了 PyDbg 提供的接口。和 PyDbg 相比，在 PIDA 的基础上编写脚本更加简单。PIDA 接口允许我们将二进制文件当作图访问，枚举节点（函数和基础块）及它们之间的边。考虑下面的例子：

```
import pida

module = pida.load("target.exe.pida")

# 进入模块中的每个函数
for func in module.functions.values():
    print "%08x - %s" % (func.ea_start, func.name)

    # 进入函数中的每个基础块
    for bb in func.basic_blocks.values():
        print "\t%08x" % bb.ea_start
```

[9] http://openrce.org/downloads/details/208/PaiMei

```
# 遍历基础块中的每条指令
for ins in bb.instructions.values():
    print "\t\t%08x %s" % (ins.ea, ins.disasm)
```

在这段代码中，我们首先加载一个通过前面分析得到的 PIDA 文件，将其作为脚本中的一个模块。随后我们进入模块中的每个函数，依次打印出函数的开始地址和符号名称。接下来，我们进入当前函数中的每个基础块。对每个基础块，我们进一步遍历其包含的指令，并打印出指令地址和反汇编码。基于我们前面提到的内容，对读者来说，如何结合 PyDbg 和 PIDA 来创建一个基础块级别的代码覆盖工具原型已经很清楚了。在核心组件的上层，PaiMei 框架的剩余部分可以被分解为下面这些组件。

- **工具**：一套完成各种重复任务的工具。其中的 process_stalker.py 是一个我们感兴趣的特定的工具类，该类用于提供代码覆盖功能的抽象接口。
- **控制台**：一个带插件的 WxPython 应用，用于快速和高效地为新创建的工具提供定制的 GUI 界面。我们就是通过这个组件与 PStalker 代码覆盖模块交互的。
- **脚本**：完成各种任务的单独的脚本，pida_dump.py IDA Python 脚本是其中一个非常重要的例子，该脚本从 IDA Pro 中运行，生成.PIDA 文件。

PStalker 代码覆盖工具通过前面提到的 WxPython GUI 提供界面，而且，我们马上将会看到依赖大量 PaiMei 框架提供的组件。在探索了 PStalker 代码覆盖工具的各个子组件后，让我们通过一个案例研究看看如何使用它。

23.4.1　PStalker 布局预览

PStalker 模块是 PaiMei 图形控制台下的许多可用模块中的一个。在如图 23.5 所示的 PStalker 初始界面中，读者可以看到这个界面被分为三栏。

数据源（Data Source）栏提供了创建和管理被跟踪目标所必需的功能，同时，负责加载合适的 PIDA 模块。数据浏览（Data Exploration）栏显示任何单独的代码覆盖运行或跟踪的结果。最后，数据抓取（Data Capture）栏是指定运行时配置选项和可执行目标的地方。PStalker 工具的一般工作流程如下（后文中有具体描述）：

- 创建一个新目标（target）、标签（tag）或同时创建二者。目标可以包含多个标签，每个单独标签包含它自己保存的代码覆盖数据。
- 选择一个标签用于跟踪。
- 为每个需要获取覆盖信息的.DLL 和.EXE 代码加载 PIDA 模块。
- 启动目标进程。

- 在数据抓取面板下刷新进程列表，选择需要附着的目标进程，或浏览到将要加载的目标文件。
- 配置代码覆盖选项并附着到选定目标上。
- 操作目标应用，同时记录代码覆盖。
- 在恰当的时候，从目标上解除附着，等待 PStalker 将抓取到的内容导出到数据库。
- 加载记录的执行信息并开始探索记录到的数据。

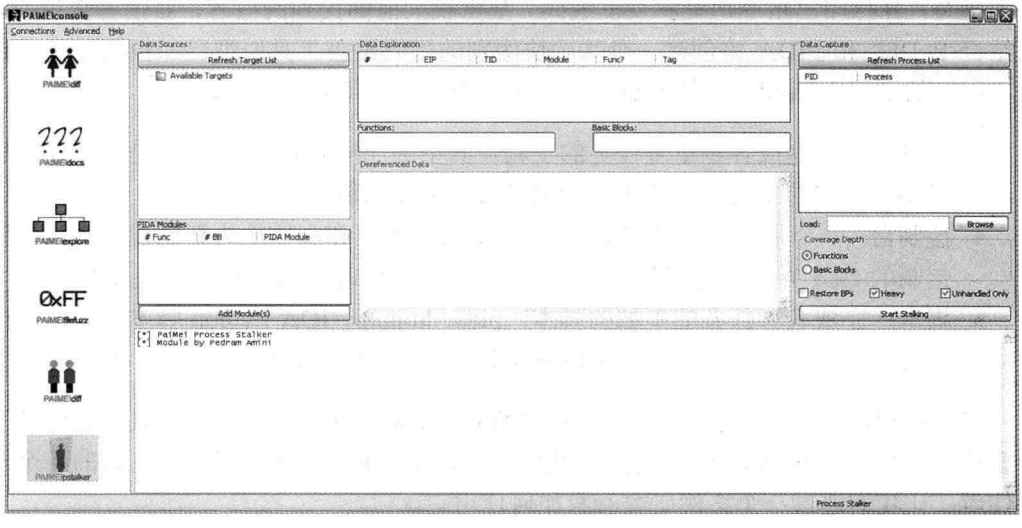


图 23.5 PaiMeiPStalker 模块

PaiMei 的在线文档[10]包括演示 PStalker 模块使用的一段完整视频。如果你以前没有看过这个视频，在继续讨论之前最好先去看看那个在线视频。

23.4.2 数据源栏

当通过“Connection”菜单建立到 MySQL 的控制台范围的连接后，选择“Retrive Target List”按钮来展开数据源浏览树。应用第一次运行的时候这个列表应该是空的。可以从标签为“Available Targets”的根节点的上下文菜单中创建新目标。使用目标节点的上下文菜单可以删除目标，向目标下加入一个标签，以及从目标下的所有标签中加载访问数据到数据浏览面板。针对单个标签的上下文菜单提供了许多特性。

[10] http://pedram.openrce.org/PaiMei/docs/PAIMEIPStalker_flash_demo/

453

- **加载访问数据**（Load hits）：清除数据浏览面板，加载与选中标签关联的访问数据。
- **追加访问数据**（Append hits）：在当前的数据浏览面板中已有的数据之后扩展与选中标签关联的访问数据。
- **导出到 IDA**（Export to IDA）：将选中标签下的访问数据导出成适合在 IDA 中导入的脚本，用来突出显示被访问到的函数和块。
- **与 uDraw 同步**（Sync with uDraw）：在控制台范围的 uDraw 连接上同步数据浏览面板。对这个特性来说，使用双显示器很有帮助。
- **用于跟踪**（User for stalking）：将所有在数据抓取阶段记录到的访问数据存储到选中的标签。在附着到一个进程前必须选中一个标签用于跟踪。
- **过滤标签**（Filter tag）：在以后的跟踪中，不要记录被选中标签下的任何节点和基础块的访问数据。
- **清除标签**（Clear tag）：保留标签，但移除标签下所有保存的访问数据。PStalker 不能将跟踪数据写入已经存在数据的标签中。如果要覆写标签内容，必须首先清除其内容。
- **扩展标签**（Expand tag）：对标签中每个被访问的函数，为函数中包含的每个基础块生成并添加一个入口，即使该基础块没有显式地被访问过。
- **目标/标签属性**：修改标签名称，查看或编辑标签备注，或修改目标名称。
- **删除标签**：移除标签和标签下所有保存的点击。

PIDA 模块列表控件显示了加载的 PIDA 模块的列表，以及它们各自所包含的函数和基础块的数量。在附着到目标进程之前，必须加载至少一个 PIDA 模块。在代码覆盖阶段，需要参考 PIDA 模块来在函数或基础块上设置断点。可以通过上下文菜单从列表中移除 PIDA 模块。

23.4.3　数据浏览栏

数据浏览面板分为三个部分。当访问数据被从数据源栏中加载或追加时，第一个部分被展开。每个访问数据都显示在带滚动条的列表框中。当选中滚动条中的一行时，如果该数据可用，与该数据相关的上下文数据会被显示在“Dereferenced Data”文本控件中。在列表框和文本控件之间有两个进度条，这两个进度条都用于表示总代码覆盖，分别基于被独立访问的基础块与总基础块的比值，以及基于数据浏览面板中列出的所有函数与加载的全部 PIDA 模块中包含的函数的比值。

454

23.4.4 数据抓取栏

当选择了一个标签用于跟踪之后，PStalker 会应用选中的过滤器，加载目标 PIDA 模块，然后，实际附着到目标进程上并对其进行跟踪。

点击“Retrive List”按钮显获取最新的进程列表，较新的进程置于列表的底部。选择要附着的目标进程，或直接浏览到要加载的目标应用，设置期望的覆盖深度。选择“Functions”选项来监视和跟踪目标进程执行了哪些函数，或选择“Basic Blocks”选项来进一步查看目标进程执行了哪些单独的基础块。

“Resource BPs”检查框控制在访问到断点之后是否恢复断点。如果仅需要知道目标中的哪些代码被执行到，不用选中这个选项。如果要知道哪些代码被执行了，以及代码的执行顺序，那就需要选中这个选项。

“Heavy”检查框控制 PStalker 是否会保存每个被记录的访问的上下文数据。不选中这个选项可以提高速度。清除“Unhandled Only”检查框，这样即使调试器能够正确地处理调试异常事件，也可以接收到这些事件的通知。

最后，设置好所有选项后，选择目标进程，点击“Attach and Start Tracking”按钮开始进行代码覆盖跟踪。日志窗口会显示运行时信息。

23.4.5 局限性

读者应该小心这个工具的一个主要局限，那就是，这个工具依赖于 DataRscue 的 IDA Pro 工具对目标二进制的静态分析；因此，如果 IDA Pro 犯了错，这个工具就会失败。例如，如果数据被错误地表示为代码，当 PStalker 在这些数据上设置断点时就会出现问题。IDA Pro 出现错误是 PStalker 最常见的失败原因。在进行自动分析之前，不选中 IDA Pro 的“Make final pass analysis kernel”选项能够禁用导致这类错误的激进逻辑。然而，不选中这个选项会影响到总体的分析结果。另外，PStalker 工具无法对打包的数据或运行中修改自身的代码进行跟踪。

23.4.6 数据存储

一般用户不需要知道 PStalker 代码覆盖工具是如何组织和存储数据的。大多数用户只需要利用工具提供的抽象层与工具交互。然而，如果我们要创建自定义的扩展或提升

生成报告的质量，就需要知道关于 PStalker 的数据库模式的更详细的信息。因此，本节将致力于分析 PStalker 背后的数据存储机制。

PStalker 中的所有目标和代码覆盖数据都保存在一个 MySQL 数据库服务器中。信息保存在三张表中：cc_hits、cc_tags、cc_targets。正如下面的 SQL 表结构所示，cc_target 数据表包含一个简单的目标名称列表、一个自动生成的唯一数据标识，以及一个用于存储与目标相关的任意注释的文本字段：

```
CREATE TABLE 'paimei'.'cc_targets' (
    'id'int(10) unsigned NOT NULL auto_increment,
    'target'varchar(255) NOT NULL default '',
    'notes' text NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=MyISAM;
```

接下来的 SQL 表结构是 cc_tags 数据表，包含一个自动生成的唯一数字标识、标签关联目标的标识、标签名称，以及一个用于存储与目标相关的任意注释的文本字段。

```
CREATE TABLE 'paimei'.'cc_tags" (
    'id'int(10) unsigned NOT NULL auto_increment,
    'target_id' int(10) unsigned NOT NULL default '0',
    'tag'varchar(255) NOT NULL default '',
    'notes' text NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=MyISAM;
```

最后一张表的表结构如下面的 SQL 所示，cc_hits 数据表包含大量被保存下来的运行时代码覆盖信息：

```
CREATE TABLE 'paimei'.'cc_hits' (
    'target_id' int(10) unsigned NOT NULL default '0',
    'tag_id' int(10) unsigned NOT NULL default '0',
    'num'int(10) unsigned NOT NULL default '0',
    'timestamp'int(10) unsigned NOT NULL default '0',
    'eip'int(10) unsigned NOT NULL default '0',
    'tid'int(10) unsigned NOT NULL default '0',
    'eax'int(10) unsigned NOT NULL default '0',
    'ebx'int(10) unsigned NOT NULL default '0',
    'ecx'int(10) unsigned NOT NULL default '0',
    'edx'int(10) unsigned NOT NULL default '0',
    'edi'int(10) unsigned NOT NULL default '0',
    'esi'int(10) unsigned NOT NULL default '0',
```

455

456

```
    'ebp'int(10) unsigned NOT NULL default '0',
    'esp'int(10) unsigned NOT NULL default '0',
    'esp_4' int(10) unsigned NOT NULL default '0',
    'esp_8' int(10) unsigned NOT NULL default '0',
    'esp_c' int(10) unsigned NOT NULL default '0',
    'esp_10' int(10) unsigned NOT NULL default '0',
    'eax_deref' text NOT NULL,
    'ebx_deref' text NOT NULL,
    'ecx_deref' text NOT NULL,
    'edx_deref' text NOT NULL,
    'edi_deref' text NOT NULL,
    'esi_deref' text NOT NULL,
    'ebp_deref' text NOT NULL,
    'esp_deref' text NOT NULL,
    'esp_4_deref' text NOT NULL,
    'esp_8_deref' text NOT NULL,
    'esp_c_deref' text NOT NULL,
    'esp_10_deref' text NOT NULL,
    'is_function' int(1) unsigned NOT NULL default '0',
    'module'varchar(255) NOT NULL default '',
    'base'int(10) unsigned NOT NULL default '0',
    PRIMARY KEY ('target_id', 'tag_id', 'num'),
    KEY 'tag_id' ('tag_id'),
    KEY 'target_id' ('target_id')
) ENGINE=MyISAM;
```

cc_hits 表包含以下这些字段。

- target_id 和 tag_id：这两个字段将这个表中的每个单独的行绑定到特定的目标和标签组合上。
- num：行表示的代码块在给定的“目标-标签”组合中以这个字段的数字顺序执行。
- timestamp：该字段存储了从 UNIX 起始时间（1970 年 1 月 1 日，00:00:00 格林尼治标准事件）以来的秒数，该格式很容易被转换成其他表示方式。
- eip：该字段存储行表示的被执行的代码块的绝对地址，通常以十六进制数字格式表示。
- tid：该字段存储了负责在 eip 位置执行代码块的线程的标识符。线程标识符由 Windows 操作系统分配，用来在多线程应用中分辨被不同线程执行的代码块。
- eax、ebx、ecx、edx、edi、esi、ebp 和 esp：这些字段含有代码块执行时这 8 个通用寄存器的数字值。对于每个通用寄存器和保存的栈偏移，存在一个 deref 字段，该字段包含了给定寄存器的值指向的 ASCII 数据。在 ASCII 数据字符串

的结尾是一个标签，标签值可以是{stack}、{heap}或{global}中的一个，表示解引用的数据来自哪里。如果由于相关的寄存器不含有一个可被解引用的地址，而不能进行解引用操作，deref 字段的值为 N/A。

- esp_4、esp_8、esp_c、esp_10：这些字段包含 esp 寄存器对应的栈偏移位置的数值（esp_4 = [esp+4]，esp_8 = [esp+8]，依此类推）。
- is_function：该字段是一个布尔值字段，值 1 表示被访问的数据块（eip 指向的数据块）是函数的起始位置。
- module：该字段存储了被访问的模块的名称。
- base：该字段包含了上一个字段中指定的模块的数字基地址。该信息可与 eip 中的值一起使用，计算被访问到的数据块在模块中的偏移。

PStalker 的这种开放数据存储方式为高级用户提供了一个发挥的空间，让高级用户可以定制和生成比工具提供的默认报告更高级的报告。下面我们通过一个完整的案例研究来测试这个跟踪器。

23.5　案例研究

我们在本节引入一个实际的案例研究，应用 PStalker 代码覆盖工具来确定一个模糊测试审计的相对完整性。模糊测试的目标是 Gizmo 项目[11]，该项目是一个包含 VoIP、即时消息和电话通信的软件套件。图 23.6 是一个来自 Gizmo 项目网站的带注释的屏幕截图，该截图详细说明了 Gizmo 工具的主要特性。

Gizmo 实现了许多特性，这也是它能够流行起来的原因之一。这个工具被许多人看作可能的 Skype[12]杀手，与被广泛应用的 Skype 相比，Gizmo 的主要区别是它建立在开放的 VoIP 标准，例如会话初始协议（Session Initiation Protocol，SIP - RFC 3261）上，而没有将自己实现为闭源私有系统。使用开放的 VoIP 标准使得 Gizmo 能够实现与其他 SIP 兼容方案的互通。同时，这个决定也使得我们可以使用标准的 VoIP 安全测试工具对 Gizmo 进行测试。现在我们已经找准了需要进行模糊测试的目标软件，下一步我们需要选择目标协议，并选择一个合适的模糊测试器。

---

[11] http://www.gizmoproject.com/

[12] http://www.skype.com/

458

23.5.1 测试策略

我们将测试 Gizmo 处理非正常 SIP 数据包的能力。虽然可以作为测试目标的 VoIP 协议不止一个，但选择 SIP 协议进行模糊测试会比较合适，因为大多数 VoIP 电话通信都会首先使用该协议进行信号协商。目前已经有一些可用的免费模糊测试器。对我们的测试而言，我们将会使用由芬兰奥卢大学的电子与信息工程系发布的，面向 SIP 协议的模糊测试器，该模糊测试器名为“PROTOS 测试套件：c07-sip”[13]，可以免费获得。PROTOS 测试套件包含 4527 个独立的测试用例，发布在一个单一的 Java JAR 文件中。虽然这个测试套件仅能够测试 INVITE 消息，但它非常适合我们的需要。另外，我们选择这个模糊测试器的部分原因是负责开发这个工具的小组已经通过 Codenomicon 公司[14]发布了该工具的商业版本。商业版本的 SIP 测试套件对 SIP 协议进行了更加深入的探索，并包含了 31971 个独立的测试用例。

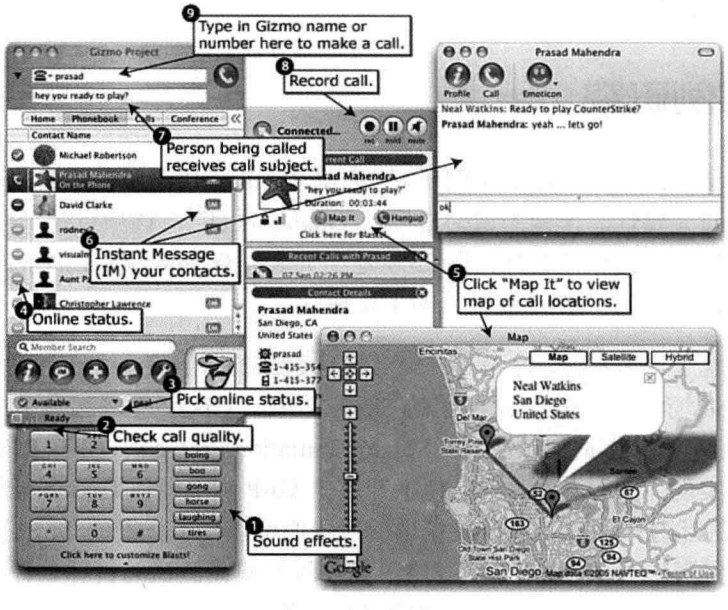


图 23.6 Gizmo 项目特性列表

[13] http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/

[14] http://www.codenomicon.com/

我们的测试目标是：利用 PStalker 彻底运行全部的 PROTOS 测试用例集合，度量被执行代码的覆盖率。我们可以扩展本案例，使其更加实用，例如，比较模糊测试方案或确定一个给定审计的可信度。在本章的后面，我们将进一步讨论使用代码覆盖度量的好处。

459

PROTOS 测试套件以命令行方式运行，提供了大约 20 个选项用以控制其运行时的行为，以-h 选项（帮助开关）运行 PROTOS 测试套件得到的输出中列出了所有的命令行参数：

```
$ java -jar c07-sip-r2.jar -h

Usage java -jar <jarfile>.jar [ [OPTIONS] | -touri<SIP-URI> ]
    -touri        <addr>        Recipient of the request
                                Example: <addr> : you@there.com
    -fromuri      <addr>        Initiator of the request
                                Default: user@pamini.unity.local
    -sendto       <domain>      Send packets to <domain? instead of
                                domainname of -touri
    -callid       <callid>      Call id to start test-case call ids from
                                Default: 0
    -dport        <port>        Portnumber to send packets on host
                                Default: 5060
    -lport        <port>        Local portnumber to send packets from
                                Default: 5060
    -delay        <ms>          Time to wait before sending new test-case
                                Defaults to 100 ms (milliseconds)
    -replywait    <ms>          Maxinum time to wait for host to reply
                                Defaults to 100 ms (milliseconds)
    -file         <file>        Send file <file> instead of test-case(s)
    -help                       Display this help
    -jarfile      <file>        Get data from an alternate bugcat
                                JAR-file <file>
    -showreply                  Show received packets
    -showsent                   Show send packets
    -teardown                   Send CANCEL/ACK
    -single       <index>       Inject a single test-case <index>
    -start        <index>       Inject test-cases starting from <index>
    -stop         <index>       Stop test-case injections to <index>
    -maxpdusize   <int>         Maxinum PDU size
                                Default to 65507 bytes
    -validcase                  Send valid case (case #0) after each
                                test-case and wait for a response. May
                                be used to check if the target is still
                                responding. Default: off
```

你的场景也许和我们的不同，但我们发现下面的选项工作得最好（touri 选项是唯一强制需要的选项）：

```
java -jar c07-sip-r2.jar -touri 17476624642@10.20.30.40  \
                         -teardown                       \
                         -sendto 10.20.30.40             \
                         -dport 64064                    \
                         -delay 2000                     \
                         -validcase
```

delay 选项设置为 2 秒，是为了给 Gizmo 一点时间释放资源，更新 GUI，或从测试用例中恢复。validcase 参数告诉 PROTOS 应该确保执行每个测试用例之后，继续下一个测试用例之前，向目标发送一个有效的通信数据。这个基础技术允许模糊测试器检测到由于目标跑飞导致的测试中断。在我们的测试中，validcase 选项发挥了重要作用，因为在测试中 Gizmo 真的发生了崩溃！幸运的是，Gizmo 中的这个漏洞是由空指针解引用导致的，因此这个漏洞并不是一个可被利用的漏洞。但无论如何，我们确实发现了一个真实存在的问题（提示：它出现在最开始的 250 个测试用例中）。读者可以自行下载测试套件，并自己把这个问题找出来。

1. Gizmo 发生崩溃时导出的上下文

```
[*] 0x004fd5d6 moveax,[esi+0x38] from thread 196 caused access violation when
attempting to read from 0x00000038

CONTEXT DUMP
    EIP: 004fd5d6 moveax, [esi+0x38]
    EAX: 0419fdfc (  68812284) -><CCallMgr::IgnorCall() (stack)
    EBX: 006ca788 (   7120776) -> e(elllllllllllllllllll (PCPlsp.dll.data)
    ECX: 00000000 (         0) -> N/A
    EDX: 00be0003 (  12451843) -> N/A
    EDI: 00000000 (         0) -> N/A
    ESI: 00000000 (         0) -> N/A
    EBP: 00000000 (         0) -> N/A
    ESP: 0419fdd8 (  68812248) -> NR (stack)
    +00: 861c524e (2250003022) -> N/A
    +04: 0065d7fa (   6674426) -> N/A
    +08: 00000001 (         1) -> N/A
    +0c: 0419fe4c (  68812364) ->xN (stack)
    +10: 0419ff9c (  68812700) ->raOo|hoho||@ho0@*@b0zp (stack)
    +14: 0061cb99 (   6409113) -> N/A

disasm around:
```

```
        0x004fd5c7 xoreax, esp
        0x004fd5c9 push eax
        0x004fd5ca lea eax, [esp+0x24]
        0x004fd5ce movfs:[0x0], eax
        0x004fd5d4 movesi, ecx
        0x004fd5d6 moveax, [esi+0x38]
        0x004fd5d9 push eax
        0x004fd5da lea eax, [esp+0xc]
        0x004fd5de push eax
        0x004fd5df call 0x52cc60
        0x004fd5e4 add esp, 0x8

SEH unwind:
        0419ff9c -> 006171e8: movedx, [esp+0x8]
        0419ffdc -> 006172d7: movedx, [esp+0x8]
        ffffffff -> 7c839aa8: push ebp
```

在 Gizmo 启动阶段检查出入端口 5060（标准的 SIP 端口）的数据，可以找出需要在测试中传递给 touri 和 dport 参数的值。图 23.7 显示了相关的片段，其中高亮显示的数字是我们感兴趣的值。

值 17476624642 是分配给我们的 Gizmo 账号的实际电话号码，而端口 64064 看上去像是标准的 Gizmo 客户端 SIP 端口。好了，我们的策略已经准备好了。我们已经选择了目标软件、协议和模糊测试器，也已经决定了模糊测试器的运行选项。现在，让我们转向这个案例的实际操作部分。

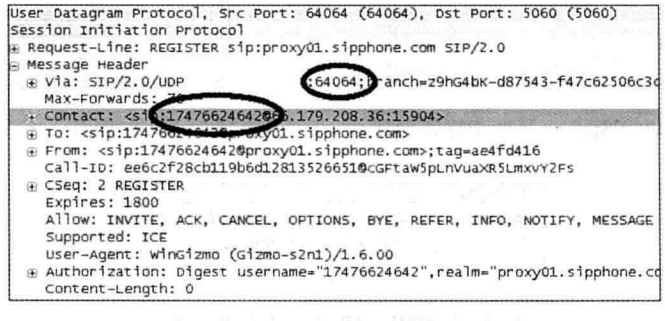


图 23.7　Gizmo 启动 SIP 包抓取解码

23.5.2　实际操作

由于我们的模糊测试对象是 SIP 协议，因此我们对 SIP 处理代码特别感兴趣。对我

们来说，幸运的是，SIP 处理代码很容易被从我们的测试对象中分离出来，Gizmo 软件所带的名为 SIPPhoneAPI.dll 的动态库就是我们的目标。我们首先将动态库加载到 IDA Pro 中，运行 pida_dump.py 脚本，生成对应的 PIDA 文件 SIPPhoneAPI.pida。因为我们仅对监视代码覆盖感兴趣，将 pida_dump.py 脚本的分析深度设定为“基础块”而不是“指令”能够提升跟踪性能。接下来，我们请出 PaiMei，检查所有必需的前提是否得到满足，确保获得代码覆盖是可操作的。要获得更多关于 PaiMei 的具体使用方法的信息，请参考在线文档[15]。图 23.8 显示了我们的第一步操作：创建名称为“Gizmo”的目标，向目标中加入一个名为“Idle”的标签。我们使用“Idle”标签记录 Gizmo 中“保持不变”的代码覆盖。这个步骤并不是必需的步骤，但由于你可以指定多个标签用于记录，然后选择使用标签来过滤数据，因此这个步骤不会带来任何不便。“Idle”标签必须被选为“用于跟踪”，如图 23.9 所示。同时，SIPPhoneAPI.pida 文件必须被加载到 PStalker 中，如图 23.10 所示。



图 23.8　PaiMeiPStalker 目标和标签创建



图 23.9　选择“Idle”标签用于“Stalking”

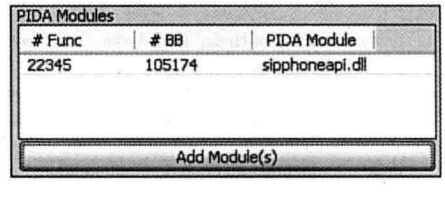


图 23.10　SIPPhoneAPI PIDA 模块被加载进来

[15] http://pedram.openrce.org/PaiMei/docs/

设置好数据源并选择标签之后，接下来，我们选择目标进程，设置数据抓取选项并开始记录代码覆盖。图 23.11 显示了我们用到的配置选项。点击“Refresh Process List”按钮获取当前运行的进程列表。在本案例中，我们选择附着到一个已运行的 Gizmo 实例，而不是启动一个新的 Gizmo 实例。



图 23.11　代码覆盖抓取选项

在“Coverage Depth”选项中，我们选择“Basic Blocks”以获得最小粒度的代码覆盖。从图 23.10 可以看到，作为模糊测试目标的动态库包含接近 25000 个函数，超过 100000 个基础块。因此，如果担心运行速度，可以提高代码覆盖度量的粒度，将覆盖深度选项设置为“Functions”以提高性能。

“Restore BPs”检查框指示是否应该在第一次访问基础块后仍然继续跟踪后续访问。由于我们只对执行了哪些代码感兴趣，因此我们可以将该检查框设置为“不选中”以提高性能。

最后，我们取消“Heavy”检查框的选中状态。“Heavy”检查框指示代码覆盖工具是否应该在每个断点上执行运行时上下文检查，并保存找到的内容。同样，由于我们只对代码覆盖感兴趣，没有记录这些额外数据的需要，因此，取消该选项的选中状态，能够帮助我们避免不必要的抓取导致的性能损失。

点击“Start Stalking”后，代码覆盖工具附着到 Gizmo 进程并开始监视其执行。如果你对这种场景下进程内部工作的具体细节感兴趣，可以参考 PaiMei 文档。激活 PStalker 之后，描述 PStalker 行动和进度的日志信息将被写入日志面板。下面的摘要输出展示了一次成功的 PStalker 执行的开头部分：

```
[*] Stalking module sipphoneapi.dll
[*] Loading 0x7c900000 \WINDOWS\system32\ntdll.dll
[*] Loading 0x7c800000 \WINDOWS\system32\kernel32.dll
[*] Loading 0x76b40000 \WINDOWS\system32\winmm.dll
[*] Loading 0x77d40000 \WINDOWS\system32\user32.dll
[*] Loading 0x77f10000 \WINDOWS\system32\gdi32.dll
[*] Loading 0x77dd0000 \WINDOWS\system32\advapi32.dll
[*] Loading 0x77e70000 \WINDOWS\system32\rpcrt4.dll
[*] Loading 0x76d60000 \WINDOWS\system32\iphlpapi.dll
[*] Loading 0x77c10000 \WINDOWS\system32\msvcrt.dll
[*] Loading 0x71ab0000 \WINDOWS\system32\ws2_32.dll
[*] Loading 0x71aa0000 \WINDOWS\system32\ws2help.dll
[*] Loading 0x10000000 \Internet\Gizmo Projects\SipphoneAPI.dll
[*] Setting 105174 breakpoints on basic blocks in SipphoneAPI.dll
[*] Loading 0x16000000 \Internet\Gizmo Projects\dnssd.dll
[*] Loading 0x006f0000 \Internet\Gizmo Projects\libeay32.dll
[*] Loading 0x71ad0000 \WINDOWS\system32\wsock32.dll
[*] Loading 0x7c340000 \Internet\Gizmo Projects\MSVCR71.dll
[*] Loading 0x00340000 \Internet\Gizmo Projects\ssleay32.dll
[*] Loading 0x774e0000 \WINDOWS\system32\ole32.dll
[*] Loading 0x77120000 \WINDOWS\system32\oleaut32.dll
[*] Loading 0x00370000 \Internet\Gizmo Projects\IdleHook.dll
[*] Loading 0x61410000 \WINDOWS\system32\urlmon.dll
...
[*] debugger hit 10221d31 cc #1
[*] debugger hit 10221d4b cc #2
[*] debugger hit 10221d67 cc #3
[*] debugger hit 10221e20 cc #4
[*] debugger hit 10221e58 cc #5
[*] debugger hit 10221e5c cc #6
[*] debugger hit 10221e6a cc #7
[*] debugger hit 10221e6e cc #8
[*] debugger hit 10221e7e cc #9
[*] debugger hit 10221ea4 cc #10
[*] debugger hit 1028c2d0 cc #11
[*] debugger hit 1028c30d cc #12
[*] debugger hit 1028c369 cc #13
[*] debugger hit 1028c37b cc #14
...
```


现在我们可以激活模糊测试器，并跟踪 Gizmo 在空闲时间内执行的代码。等待一段时间后，PStalker 跟踪得到的被访问到的断点列表不再增加，这意味着我们已经访问到了当 Gizmo 处于空闲状态时 SIP 库中全部会被执行的代码。接下来，我们创建一个

新标签，通过右键菜单选择“Use for Stalking”将其选为活动标签。最后，我们使用右键点击前面创建的“Idle”标签并选择“Filter Tag”。通过这些操作，我们有效地忽略了所有 Gizmo 在空闲（idle）状态时执行的基础块。接下来，我们启动模糊测试器并观察其运行。图 23.12 展示了执行 PROTOS 的全部 45237 个测试用例时显示的代码覆盖情况。

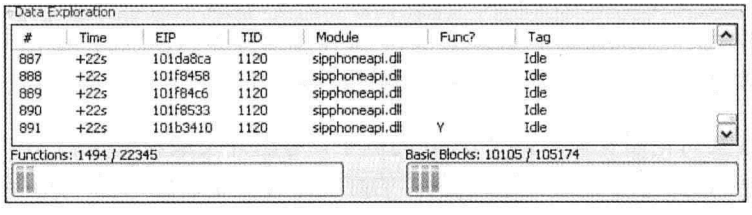


图 23.12　代码覆盖结果

总而言之，PROTOS 测试套件仅能够访问到 SIPPhoneAPI 库中大约 6%的函数和 9%的基础块。对一个免费的测试套件来说，这个结果也还不错，但这个结果解释了为什么运行一个单独的测试套件与进行综合的审计在效果上相差甚远。运行测试套件后，我们并没有能够测试到大部分代码，而且也不大可能全面测试了我们访问到的代码的功能。然而，通过在测试用例的短暂间隔内增加一个进程重启机制，我们就能够提升覆盖结果。原因是即使 Gizmo 成功地响应了有效测试用例，而且在错误的测试之间插入了正常的请求，一段时间后 Gizmo 仍然会停止正常运行，从而影响代码覆盖。重启进程能够重置 Gizmo 的内部状态，使其恢复完整功能，从而允许我们更精确、更全面地检查目标。

PROTOS 的商业版本包含的测试用例数量是 PROTOS 的 7 倍，那么，我们是否能够期望应用 PROTOS 的商业版本可以达到 7 倍的覆盖率？也许不能。虽然应用 PROTOS 的商业版本可以提高覆盖率，但测试用例数量与代码覆盖之间并不存在一致的或可预测的比例关系。唯一能够获得精确结果的方法是通过观测。

那么，用这种方式跟踪模糊测试器的执行能带给我们什么好处呢？我们能够如何改进该技术？在下一节，我们将为这些问题提供答案。

23.6　优势与将来的改进

传统意义上，我们并不会以非常科学的方式进行模糊测试。经常出现的模糊测试场景是：安全研究者和 QA 工程师一起写出一些基本的脚本，用脚本生成非正常的数据，用这些数据运行被测应用一段时间，然后退出被测应用。随着最近商业模糊测试方案的

增长，我们可以利用由 Codenomicon 等商业公司开发的完整测试用例列表，把模糊测试向科学的测试方面推进一步。对那些只需要找到软件漏洞并报告或出售的安全研究员而言，他们通常没有更进一步的需求。考虑到软件安全的当前状况，通过目前的测试方法，我们已经能够得到足够多的缺陷和安全漏洞。然而，随着安全开发和测试成熟度的提升，以及业界对安全开发和测试关注度的提升，我们需要更加科学的方法。软件开发者和 QA 团队关心的是发现所有漏洞，而不仅仅是那些容易发现的漏洞。他们将会更加关注采用科学的方法进行模糊测试。模糊测试器跟踪是这个方向上必须的一步。

我们继续来讨论贯穿本章的 Gizmo 的例子，考虑开发最新版本的 VoIP 软件电话的各方，以及他们能从模糊测试跟踪中获得的好处。对产品经理来说，知道测试覆盖了产品哪些部分的代码使他能够更准确地做出决定，以及对已经进行的各层次的产品测试更有信心。例如，考虑下面的陈述："我们最新的 VoIP 软件电话版本通过了 45000 个恶意的测试用例。"这个声明什么都不能证明。也许在执行的 45000 个恶意用例中，只有 5000 个用例与目标软件电话提供的特性相关。也可能这全部 45000 个测试用例只不过覆盖软件电话提供的可用特性的 10%。又或许，这些测试用例是由"真正聪明"的工程师编写的，覆盖了全部的代码。后一种情况当然很棒，但是，不通过代码覆盖分析就没有办法知道前面那个声明的有效性。如果结合模糊测试器跟踪器和这 45000 个测试用例组成的测试集，我们可以得到下面这个更有信息量的陈述："我们最新版本的 VoIP 软件电话通过了 45000 个恶意测试用例，测试覆盖了代码的 90%以上。"

对 QA 工程师而言，了解软件电话的哪些区域未被测试使得他能够为下一轮的模糊测试创建更智能的测试用例。例如，在跟踪软件电话的模糊测试器执行时，工程师也许会注意到模糊测试器生成的测试用例持续地触发一个名为 parse_sip()的例程。在深入分析这个例程之后，工程师观察到虽然这个函数被执行了多次，但并不是函数中的所有分支都被覆盖了。为了改进测试，工程师应该检查没有被覆盖的分支来决定如何修改模糊测试器的数据生成例程，覆盖那些没有被覆盖到的路径。

对开发者来说，知道发生错误前所执行的具体指令集和路径使得他能够快速和准确地定位并修复受影响的代码区域。目前，QA 工程师和开发者之间只能在较高层次上沟通，因此可能导致双方都把时间浪费在重复研究上。结合模糊测试器跟踪技术，开发者能够收到 QA 团队给出的，详细描述测试用例发现的可能的错误位置的报告，而不是一份仅列出触发了各种失败的测试用例的列表。

记住，作为一个独立的度量方法，代码覆盖并不意味着彻底的测试。考虑这个例子：我们在某个可执行文件中测试了一个字符串复制操作，但是仅使用了较短长度的字符串

进行测试。当向这段代码传入一个特别长或具有不恰当格式的字符串时，这段代码可能会存在缓冲区溢出漏洞，但我们的测试工具无法发现这个漏洞。知道我们已经测试了什么只是一半的胜利，理想情况下我们还需要提供对测试效果的度量。

23.6.1　将来的改进

对这个代码覆盖工具而言，一个可以考虑加入的便利的特性是增加与其他模糊测试工具的通信能力。在发送每个单独的测试用例之前和之后，模糊测试器可以向代码覆盖工具发送通知，因此可以将每个单独的测试用例与记录的代码覆盖关联起来。另一种实现关联的方法是后处理方式，通过对齐数据传输的时间戳和代码传输的时间戳也可以实现两者的关联。由于存在延迟和很难完美地进行时钟同步，第二种方法通常不那么有效，但比较易于实现。在这两种情况下，得到的结果都能提供信息，让分析者能够深入到与单独测试用例对应的特定的代码块中。

468

枚举目标内所有基础块和函数是一个痛苦和容易出错的过程。解决这个问题的方案之一是在运行时枚举基础块。根据 Intel IA-32 体系结构软件开发者手册[16] 3B[17]卷第 18.5.2 节，奔腾 4 和至强（Xeon）处理器提供了硬件层次的“分支单步”支持。

> **分支单步（single-step on branches，BTF）标志（第一位）**
>
> 设置该位后，处理器将 EFLAGS 寄存器中的 TF 标志看作“单步分支”标志，而不是“单步指令”标志。该机制允许在分支、中断和异常时对处理器进行单步运行。参见 18.5.5 节“在分支、异常和中断上单步运行”。

利用这个特性，我们可以开发一个无须进行事先分析，就能在基础块级别跟踪目标进程的工具。同时，由于不再需要在每个单独基础块的开始位置设置软件断点，这种方法能够提高运行速度。然而，需要记住，为了得到代码的完整范围和覆盖率数据，我们仍然需要静态分析和枚举基础块。此外，虽然在开始进行跟踪时，这种方法会比较快，但对长期运行而言，它可能并不比前面介绍的方法快。这是因为这种方法不能停止监视已经执行过的块，也不能仅跟踪特定模型。但不管怎么说，这种方法是一种有趣且强大的技术。OpenRCE 上有一篇博客文章“用 Intel MSR 寄存器进行分支跟踪”[18]，该文基

---

[16] http://www.intel.com/products/processor/manuals/index.htm

[17] ftp://download.intel.com/design/Pentium4/manuals/25366919.pdf

[18] http://www.openrce.org/blog/view/535

于本章前面提到的 PyDbg 单步跟踪器进行了扩展，给出了一个可以工作的分支级别的单步跟踪器。对该方法的进一步研究作为练习留给读者。

改进的最后一个方面是考虑代码覆盖和路径覆盖（或进程状态覆盖）之间的区别。在本章中我们的讨论集中于代码覆盖；换句话说，我们讨论的是实际执行了哪些指令或代码块。而路径覆盖进一步考虑了一些代码块中的各种可能路径，例如，考虑如图 23.13 所示的代码块。

469

代码覆盖分析表明 4 个块 A、B、C 和 D 都已经被一系列的测试用例访问到了。但是，是否所有可能的路径都已经被执行了？如果是，每个单独的测试用例执行了哪些路径？要回答这些问题，就需要路径覆盖技术。路径覆盖技术能够给出结论，说明下面的哪些路径组合已被覆盖：

- A→B→D
- A→B→C→D
- A→C→D

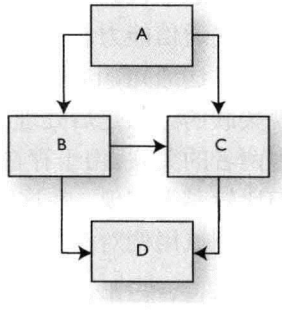


图 23.13　控制流图示例

例如，如果以上三条路径中，仅有第一条和第三条路径被执行，代码覆盖分析会给出 100%代码覆盖的结论。而路径覆盖则能够告诉我们只覆盖了 66%的可能路径。路径覆盖提供的这个附加信息使得分析者能够进一步调整和改进测试用例列表。

23.7　小结

在本章中我们介绍了代码覆盖的概念，并从一个仅由数行 Python 代码实现的简化方案开始，在模糊测试中应用这种技术。通过研究二进制可执行代码的构件，我们演示了一种更高级的实现代码覆盖的方法。虽然以前公众对模糊测试的关注大部分都集中在数据生成上，但数据生成实际上仅解决了一半问题。我们不应该仅仅提出“我们该如何开发模糊测试”这个问题，同样，我们也需要问“我们什么时候停止模糊测试”。如果不监视代码覆盖，我们就无法智能地决定该在什么时候停止模糊测试的投入。

470

在本章中，我们介绍并展示了一个用户友好的、开源的代码覆盖分析工具 PaiMei PStalker。我们探索了在模糊测试中应用这个工具，同时还讨论了这个工具可以改进的地方。代码覆盖工具和概念（或“模糊测试跟踪”）带领我们朝更科学的模糊测试方法迈进了一步。在下一章中，我们将讨论智能异常检测技术，该技术可以结合到我们的测试套件中，进一步将模糊测试带到下一个阶段。

第24章

智能错误检测

"Never again in the halls of Washington, DC, do I want to have to make explanations that I can't explain."

——George W. Bush, Portland, OR, October 31, 2000

从前面的章节中，我们知道了怎样选择测试目标，知道了怎样生成测试数据。通过上一章，我们还知道了如何对模糊测试进行跟踪，以了解我们的数据在哪里，以及如何被处理。下一个我们要掌握的重要内容是：如何成功地判断我们的模糊测试器何时触发了问题。这个问题的答案并不是显而易见的，因为也许从目标系统的外部并不能轻易检测到目标系统内发生的错误。在目前可用的模糊测试器中，这个方向并没有得到大的关注。然而，与模糊测试器跟踪这个方向不同，我们已经开始看到智能错误检测在商业领域和开源领域内都已经前进了一大步。

在本章中，我们从探索最原始的错误检测方案（例如，简单地对响应的解析）开始。然后，我们借助调试工具和调试技术，转向更高级别的错误检测。最后，通过探索动态二进制插装（Dynamic Binary Instrumentation，DBI），我们会简单地介绍最高级的错误检测技术。总而言之，本章展示的这些工具、技术和概念将通过发现错误的方式，帮助我们判断我们的模糊测试器何时成功地发现了漏洞。

24.1 原始的错误检测技术

假设你向被测目标发出了 50000 个用于模糊测试的 IMAP 认证请求，其中一个请求

导致 IMAP 服务器发生了崩溃，但是，你并没有随时检查 IMAP 服务器的状态，因此直到发送完最后一个测试用例，你才知道这些请求中的某些触发了漏洞。在这种情况下，你得到了什么有价值的信息？简单地说，在这个案例中，你什么都没有得到。如果你连到底哪个测试用例导致了 IMAP 服务器的崩溃都不清楚，在执行完模糊测试之后，你并没有得到更多的信息。一个不关心测试目标健康状态的模糊测试器是完全无价值的模糊测试器。

模糊测试器可以利用多种技术来判断一个独立的测试用例是否在目标中引发了不正常，像 PROTOS 那样在测试用例之间加入一个连接检查也许是最简单的方法。仍然以我们的 IMAP 服务器的测试为例，假设我们应用模糊测试器生成了下面两个测试用例：

```
x001 LOGIN AAAAAAAAAAAAAAAAAAAAAA...AAAA
x001 LOGIN %s%s%s%s%s%s%s%s%s%s%s%s...%s%s%s%s%s
```

在执行每个生成的测试用例之后，模糊测试器可以尝试在被测系统的 143 端口（IMAP 服务器端口）上建立一个 TCP 连接。如果建立连接失败，我们就可以假设最后一个执行的测试用例导致服务器发生了崩溃。Codenomicon 公司提供的商业测试套件就采用了这种方法，该套件提供了一个功能，使用者可以选择在执行每个测试用例后发送一个合法的或已知的“好”的用例。实现这个功能的伪代码非常简单：

```
for case in test_cases:
    fuzzer.send(case)
    if not fuzzer.tcp_connect(143):
        fuzzer.log_fault(case)
```

通过将“检查连接”替换为“检查响应是否有效”，我们可以对这个逻辑进行小小的改进。例如，如果我们知道用户名 paimei 和口令 whiteeyebrow 是 IMAP 服务器上的有效认证凭证，那么我们就可以通过尝试以用户名 paimei 进行认证，而不是仅仅通过服务器是否可访问来判断其行为是否正常。如果认证失败，我们同样假设我们执行的最后一个测试用例导致了服务器的不正常。实现该任务的伪代码也非常简单：

```
for case in test_cases:
    fuzzer.send(case)
    if not fuzzer.imap_login("paimei", "whiteeyebrow"):
        fuzzer.log_fault(case)
```

473

前文中，我们提到了在连接失败或对期望响应的检查失败的情况下，我们都假设最后一个执行的测试用例导致了目标工作不正常。但这个描述并不完全准确。例如，在对 IMAP 服务器进行模糊测试时，在向被测目标发送了第 500 个测试用例后，我们发现连接测试失败了。太棒了！看来我们发现了一个能够导致服务器崩溃的输入。然后，我们

抑制住跑下楼报告我们伟大发现的冲动，尝试重现这个错误。重启 IMAP 服务器，重新单独运行测试用例 500……然而，让我们失望的是，什么都没有发生。怎么回事？可能的情况是这样：也许是前面一些测试用例的组合将 IMAP 服务器置于某种特定的状态，而在这个状态下第 500 个测试用例导致服务器挂掉。然而，如果服务器不处于特定的状态，看起来被测系统完全能够恰当地处理测试用例 500。在这种情况下，我们可以利用一种叫作“模糊测试器步进（Fuzzer Stepping）”的简单技术，帮助缩小可能导致错误发生的测试用例范围。

上面这个问题可以用更一般的方式来描述：我们知道测试用例 1 到测试用例 499 之间的某些用例将 IMAP 服务器置为特定状态，导致测试用例 500 引发了服务器崩溃，但我们不清楚哪些测试用例的组合导致了这种特定状态。有不少可用的步进算法可以解决这个问题。我们从下面这个简单的例子开始：

```
# 找到上边界
for in in xrange(1, 500):
    for j in xrange(1, i + 1):
        fuzzer.send(j)

    fuzzer.send(500)

    if not fuzzer.tcp_connect(143):
        upper_bound = i
        break

    fuzzer.restart_target()

# 找到下边界
fori in xrange(upper_bound, 0, -1):
    for j in xrange(i, upper_bound + 1):
        fuzzer.send(j)

    fuzzer.send(500)

    iffuzzer.tcp_connect(143):
        lower_bound = i
        break

    fuzzer.restart_target()
```

以上代码片段的前半部分定位导致期望状态的测试用例序列的上边界。这部分代码

先向目标发送第 1 个到第 n 个用例，然后发送第 500 个用例。n 的值从 2 开始，每次递增 1。测试完一个序列后，代码重新启动模糊测试目标来重置内部状态。用来定位导致期望状态的测试用例序列的下边界的代码与此相似。通过查找上边界和下边界，我们的例程能够将导致目标崩溃的测试用例序列找出来。现在，我们能够再次触发服务器崩溃了，真是让人兴奋。当然，这种算法相对简单，不会考虑这种情况：实际将服务器置于漏洞状态的是一对我们识别范围内的测试用例，而不是一个测试用例序列。例如，假设实际导致崩溃的测试用例序列是第 15 个用例、第 20 个用例和第 500 个用例。在这种情况下，我们的算法会认为第 15 个用例到第 20 个用例的序列及第 500 个测试用例导致了崩溃。但无论如何，这个算法都能够帮助我们缩小问题的范围。

我们刚才讨论的简单方案是监视错误的基本方法。为了更好地理解我们的模糊测试目标中究竟发生了什么，可以使用调试器进行更加深入的挖掘，获得底层信息。然而，在开始使用调试器之前，重要的是，需要明白我们真正寻找的究竟是什么。

24.2 我们寻找的是什么

在计算机的最底层，当发生失败、异常或中断（这里统称为事件或异常）时，例如内存访问违例或除零错时，CPU 会通知操作系统。许多这类事件的发生过程非常自然。例如，当一个进程尝试访问已经交换到磁盘的内存页时，就会产生一个页面失效，操作系统会处理该事件，从磁盘缓存中将合适的页面加载到主存中。该事务对请求访问内存的进程完全透明。未被操作系统处理的事件都会被上升到进程级别。这才是我们感兴趣的地方。

当我们的模糊测试器在目标进程或设备内东游西荡时，可能会触发大量的事件。如果将调试器附着到模糊测试目标上，我们就能在事件出现时将其截获。截获到事件后，我们就停止目标进程的执行，让目标进程等待交互指令。将这种方法与前面提及的检查方法结合起来，当调试器截获到事件后，就暂停模糊测试目标的运行，使得随后的连接检查或对有效输入产生输出的检查发生失败。结合使用这两种方式，我们可能就能在这个示例的 IMAP 服务器中发现更多问题。由于在汇编级别上会出现各种异常事件，因此我们需要“智能”地确定这些事件究竟是什么原因导致的，以及，更重要的是，从安全性的角度来确定事件是否值得关注。下面给出的列表列出了软件内存破坏的三个分类，所有软件内存破坏漏洞都可以被归在这三个类别中。接下来，我们将查看每个类别的一些例子，并探讨每类错误可能会导致哪些类型的异常。这三类软件内存破坏的分类是：

475

- 将执行控制权传给不应该执行的地址。

- 从不应该读取数据的地址读取数据。
- 向不应该写入数据的地址写入数据。

经典的栈溢出就是一个“将执行控制权传给不应该执行的地址”的例子，下面的代码片段来自一个带有栈帧（Stack Frame）的函数。该函数的栈布局如图 24.1 所示。

```
void taboo (int arg1, char *args)
{
    int x;
    charbuf[16];
    int y;

    strcpy(buf, arg2);
}
```

这个函数带有一个整型参数和一个字符串参数，声明了 3 个局部变量，函数将传入的字符串复制到声明的栈缓冲区中，而没有考虑源字符串和目标缓冲区的大小。当调用函数 taboo() 时，该函数的参数被以逆序压入栈中。所产生的汇编 CALL 指令会隐式地将当前指令的指针地址（保存在寄存器 EIP 中）压入栈中。这样，当函数执行完后，CPU 就知道应该把控制权转到什么地方。通常情况下，系统自动生成的函数开头部分的汇编指令会将当前的帧指针（保存在寄存器 EBP 中）压入栈中。最后，系统将会修改栈指针（保存在寄存器 ESP 中）的值，为声明的 3 个局部变量分配空间。形成的栈帧如图 24.1 所示。

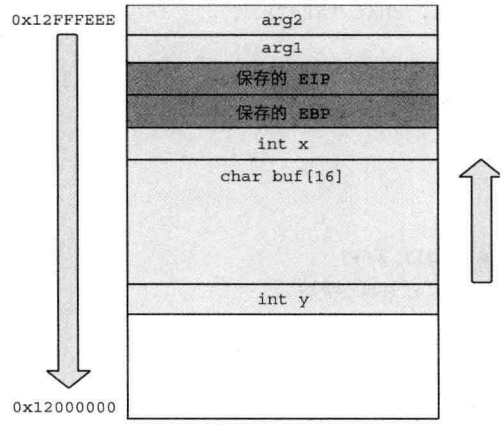


图 24.1　示例的栈布局

如图 24.1 所示，栈从高地址向低地址生长。而数据写入则正好相反，方向是从低

地址到高地址。如果字符串 arg2 的长度长于 16 个字节（16 字节是 buf 声明的大小），strcpy() 函数将越过 buf 分配的存储空间的尾部继续复制，覆盖局部变量 x，然后覆盖保存的帧指针，接着覆盖保存的函数返回地址，再往后是第一个参数等。假设我们传入的 arg2 是一个由字符“A”组成的长字符串，那么保存的 EIP（函数执行完后返回的地址）的值就会被覆写为 0x41414141（0x41 是 ASCII 字符 A 的十六进制值）。

476 当函数 taboo() 执行结束后，RETN 汇编指令将已被覆写的保存的 EIP 值回复到 EIP 中，将控制传给地址 0x41414141 处的指令。根据地址 0x41414141 处的内存的内容不同，可能会导致不同的情况。多数情况下，没有有效的内存页面映射到这个地址上，因此当 CPU 试图从不可读的地址 0x41414141 处读取下一条指令时，就会发生 ACCESS_VIOLATION 异常。注意，在这个例子中，如果传入的 arg2 的值长于 16 个字节，但短于 20 个字节，不会发生异常。

如果地址 0x41414141 处存在有效的数据，但不是可执行的代码，而且处理器支持非可执行（notexecutable，NX[1]）页面许可，当 CPU 试图从非可执行地址 0x41414141 处执行代码时，会发生 ACCESS_VIOLATION 异常。这里最有意思的地方是，如果地址 0x41414141 处真的有合法的可执行代码，就不会发生异常。这是本章后面要讲到的一个关键点。这个例子演示了最直接的一种劫持控制流的形式。

下面这段代码来自一个带栈帧的函数，是一个“从不应该读取数据的地址读取数据”的例子。图 24.1 同样描述了该函数的栈布局：

477
```
voidtaboo_two (int arg1, char *arg2)
{
    int *x;
    charbuf[] = "quick brown dog.";
    int y = 10;

    x = &y;

    for (inti=0; i< arg1; i++)
        printf("%02x\n", buf[i]);

    strcpy(buf, arg2);

    printf("%d\n", *x);
}
```

[1] http://en.wikipedia.org/wiki/NX_bit

这个函数带有两个参数，一个整型参数和一个字符串参数。声明了 3 个局部变量，将整型变量 x 的值设置为指向 y 的指针。接下来，这个函数打印由 arg1 指定数量的包含在 buf 中的字节的十六进制值。由于代码中没有检查传入的循环次数 arg1，因此，如果传入的 arg1 的值大于 16，printf() 例程就会访问并显示它本来不应该访问的数据。

同样，数据读取是从低地址到高地址的，与栈增长的方向相反。因此函数中的循环会打印出局部变量 x，然后是被保存的帧指针、被保存的返回地址的值，以及第一个参数的值等。当 arg1 的值不够大的时候，也许读取时不会产生 ACCESS_VIOLATION 异常，而只会导致访问程序员逻辑上不希望暴露出来的数据，使得意识不到这些数据敏感性的 CPU 和操作系统将其显示在执行者面前。

接下来，和前一个例子一样，这个函数进行了一次不考虑边界的字符串复制。假设传入的 arg2 的值包含 20 个字符“A”（比字符串 buf 的长度多 4 个字节），那就只有局部变量 x 会被覆写。在这种情况下，栈溢出不会像上个例子一样导致不合法的执行转移。然而，当在函数中调用 printf() 的时候，被覆写的变量 x 会作为指针被解引用，CPU 尝试从地址 0x41414141 读取数据，这时就可能会产生一个 ACCESS_VIOLATION。

下面的代码片段给出了“从不应该读取数据的地址读取数据”的真实例子，这段代码展示了经典的格式字符串漏洞是如何发生的：

```
void syslog_wrapper (char *message)
{
    syslog(message);
}
```

根据 syslog() API 的原型，该 API 接受一个格式字符串和一个可变的格式字符串参数列表。代码中将用户传入的 message 参数的值（字符串）直接传递给 syslog()，因此，syslog() 会解析用户输入参数中包含的所有字符串标记（%s、%d 等），并将对应的格式字符串参数解引用。如果传入的某些格式标记没有对应的格式字符串参数，那么 syslog() 就会使用目前存在于函数栈上的值。例如，如果用户传入参数的内容中包含“%s%s%s%s%s”，那么 syslog()就会从函数栈中读取 5 个地址，将其解引用为字符串指针，从指针一直向后访问，直到找到 NULL 字节。如果用户在传入的 message 参数值中包含足够多的%s 格式字符串标记，通常都会导致不合法的指针被当作字符串，最终导致 ACCESS_VIOLATION 异常。在某些情况下，指针可能指向有效的内存地址，但是其执行的内存地址中存储的并不是以 NULL 结尾的字符串，就会导致函数在查找

NULL 字节时一直读到超出内存页的结尾。注意，根据环境和用户输入参数中包含的格式字符串标记的数量，这个函数在执行时并不一定会发生违例情况。在本章的后面，我们会再次讨论这个关键概念。

下面来自带栈帧函数的 C 代码是一个“向不应该写入数据的地址中写入数据”的例子。这个函数带有栈帧，栈布局如图 24.1 所示。

```
voidtaboo_three (int arg1, char *arg2)
{
    int x;
    charbuf[] = "quick brown dog.";
    int y;

    buf[arg1] = '\0';
}
```

该函数有两个参数，一个整型参数和一个字符串参数，声明了 3 个局部变量，在参数 arg1 指定的位置截断了保存在 buf 中的字符串。由于这个函数没有对调用者输入的 arg1 进行检查，因此，如果传入的 arg1 的值超过 16，将会导致一个错误的空字节被写入不该被写入的地址。和上一个例子中看到的一样，根据空字节写入位置的不同，不一定会实际产生一个异常。但是，即使测试过程中没有生成异常，在进行模糊测试的过程中，当发生这类事情时我们还是希望能够知道。

下面这段代码展示了一个更真实的例子。这段简化的代码展示了“向不应该写入数据的地址写入”的情况，演示了典型的堆溢出漏洞[2]是如何引起的：

```
char *A = malloc(8);
char *B = malloc(16);
char *C = malloc(24);

strcpy(A, "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP");
free(B);
```

这段代码声明了 3 个字符指针，指针初始化为指向来自堆的动态分配的内存。在操作系统中，每个堆块会包含前向和后向指针，构成了如图 24.2 所示的双向链表。

当执行 strcpy() 调用后，A 引用的缓冲区会发生溢出，因为这个缓冲区无法存储传入的由 P 字符组成的长字符串。发生溢出的缓冲区不存储在栈中（存储在堆中），因此

[2] http://doc.bughunter.net/buffer-overflow/free.html

这次溢出不会修改保存的返回地址。那么，溢出会导致问题吗？在下一行代码中，free(B)会从堆块的双向链表中去掉 B 块，将 A 块和 C 块连接在一起。连接关系的更改通过如下逻辑实现：

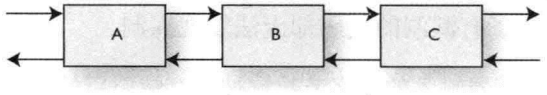


图 24.2　堆块双向链表

```
B->backward->forward = B->forward
B->forward->backward = B->backward
```

由于后向和前向指针保存在每个堆块的起始位置，strcpy() 导致的溢出会将 B 块包含的前向和后向指针的值覆写为 0x50505050（0x50 是 ASCII 字符 P 的十六进制值）。这样，当执行 free(B) 操作时，在汇编级别上，对 B 的前向和后向指针的操作会成为非法的读写操作。

480

一次悲剧的安全方案尝试

尽管与现在讨论的主题不直接相关，但我们还是希望利用这个机会描述某个可笑的“预编译安全方案”的尝试。前段时间德雷塞尔大学计算机系发表了一篇肤浅的名为“使用程序变换来防止缓冲区溢出，提高 C 程序的安全性”[3]的研究论文。

该论文要求将所有栈缓冲区替换成从堆中分配的缓冲区，认为这样就可以消除所有可被利用的缓冲区溢出漏洞。

这篇 10 页的论文描述了动态变换，讨论了变换的效率，列出了大量的参考文献，甚至演示了一个例子，说明他们如何将一个认为设计的可被利用的代码执行漏洞转变成一个简单的 DoS。然而，对作者而言，不幸的是，他在研究中从未注意到堆溢出也是可被利用的这一事实。

24.3　选择模糊测试值的注意事项

到目前为止，我们举例说明了许多高级语言代码片段和高层次上的软件漏洞概念是

[3] http://www.cs.drexel.edu/~spiros/research/papers/WCRE03a.pdf

如何被“翻译”成低层次事件的，接下来，我们来解决一个关键问题：如何选择模糊测试值。读者可能已经注意到，根据不同的条件，将模糊测试值作为内存地址引用可能并不会导致实际的访问违例，因此也就不会产生异常。当选择模糊测试请求中使用的模糊测试值时，必须牢记这一点。选择不合适的模糊测试值会导致错误的假象，比如，我们的模糊测试器成功遍历了有漏洞的代码却无法发现漏洞。

例如，考虑模糊测试器输入的 4 字节数据被内存解引用的多种情形。由字符“A”组成的经典长字符串会导致对内存地址 0x41414141 的解引用。虽然大多数情况下，没有有效内存页面映射到这个特定地址上，因此解引用操作会导致我们期望的 ACCESS_VIOLATION 异常。但是，在某些情况下，地址 0x41414141 却是个有效的地址，对该地址的解引用不会触发异常。虽然在这种情况下，从触发异常的角度来说，我们的模糊测试器很可能已经给被测对象造成了足够大的破坏，甚至会导致被测目标在执行接下来的语句后立刻就触发异常，但从定位问题的角度来说，我们希望当被测系统偏离正轨时就能立即发现。那么，如果由你来选择作为非法内存地址的 4 个字节的值，你会选择什么值？显然，选择内核地址空间（通常是 0x80000000 - 0xFFFFFFFF）中的地址值作为模糊测试的输入值是个不错的主意，因为在用户模式下，内核空间总是不可被访问的。

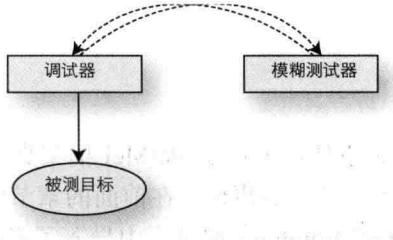
这里还有另一个例子，这次我们考虑格式字符串漏洞。我们已经看到，如果在格式字符串中提供的 %s 字符串标记少于给出的参数，后续对栈指针解引用的操作就可能会触发对无效内存的解引用。当遇到这类问题时，你的第一反应可能是向模糊测试请求中添加 %s 标记来解决问题。然而，经过进一步分析，你可能会发现某些因素，例如字符串长度的限制，可能会导致你的方案不可用。如何才能优雅地解决这个问题？在本书前面我们提到过，利用格式字符串漏洞的关键是能够利用格式字符串标记 %n 和它的派生标记写入栈中的能力。%n 格式字符串标记将应该被格式字符串输出的字符的数量写入相应的栈指针中。通过提供%n 标记而不是%s 标记，我们有更大的可能性在存在输入字符串长度限制的情况下触发错误。考虑到%n 标记可能会被应用过滤或禁用，组合应用%n 和%s 标记可能是最聪明的选择（第 6 章“自动化与数据生成”中提供了这方面的更多信息）。

简而言之，关注如何选择的模糊测试数据，以及如何使用这些数据能够带来巨大的好处。请参考第 6 章以获得更多关于如何选择合适的模糊测试数据的信息。

24.4 自动化的调试器监视

由于大多数调试器被设计为交互使用，用程序的方式自动控制调试器或从调试器中

抽取信息超出了它们的设计初衷，因此在调试器的辅助下进行目标监视不怎么可行。所以，我们要进行的最后一项工作是在模糊测试上手工登记调试器，当调试器识别到异常后重启进程。幸运的是，我们能够再次利用前面介绍的 PaiMei[4]逆向工程框架。利用这个框架，我们可以实现一个如图 24.3 所示的模糊测试与监视系统。请注意，PaiMei 逆向工程框架目前只能用于 Windows 平台。另外，在本节和本书的剩余部分，我们关注的都是 Intel IA-32 体系结构。



482

图 24.3　模糊测试器/监视架构

24.4.1　一个基础的调试器监视器

仍然考虑我们本章的例子，假设我们的测试目标是一个 IMAP 服务器。这个目标可以被附加到我们即将创建的调试器，或者在调试器的控制下加载。我们要实现的调试器和我们的模糊测试器之间有一个双向的通信信道。可以用多种方式实现这个信道回路。最方便的选择可能是使用套接字，因为套接字使得我们可以用不同的编程语言编写模糊测试器和调试器，而且可以在不同的系统，甚至不同的平台上运行模糊测试器和调试器。为了进行深入研究，让我们考虑以下基于调试组件的 Python 代码：

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

import utils

def av_handler (dbg):
    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)

    # 向模糊测试器发信号
```

[4] http://www.openrce.org/downloads/details/208/PaiMei

```
        print crash_bin.crash_synopsis()
        dbg.terminate_process()

    while 1:
        dbg = pydbg()
        dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, av_handler)
        dbg.load(target_program, arguments)

        # 向模糊测试器发信号

        dbg.run()
```


这段代码看上去没做什么事情，但由于 PaiMei 框架真正实现了其宣称的“简明扼要”，这段 Python 片段实际上做了不少事情。在前面的章节中，我们介绍过以上代码的前两行，这两行代码从 PyDbg[5] Windows 调试库中导入了必需的功能和定义。然而，我们在前面却没有提过第三条语句中导入的模块，这条语句导入了 PaiMei.utils[6]模块，该模块包含了多种逆向工程的支持库。在本案例中，我们使用 PaiMei.utils 模块包含的 crash_binning 模块（我们将在后文解释该模块）。跳过几行代码，接下来是一个无限 while 循环。这个 while 循环首先创建一个 PyDbg 类的实例，然后将函数 av_handler() 注册为处理 ACCESS_VIOLATION 异常事件的回调处理函数。当发生一个无效的内存解引用时（例如前面提到的例子），这个注册的函数就会被调用。接下来，调试器使用可选参数加载目标程序。

这时，我们需要通过信道回路向模糊测试器发送信号，让它知道我们的目标已经准备好接受模糊测试请求。在循环的最后一行，调试器通过 run() 成员函数允许被调试对象（也叫作 debug_event_loop()）执行。

程序的剩余部分是 av_handler() 函数的定义。和所有注册事件的回调函数一样，av_handler() 函数仅带一个参数，也就是 PyDbg 调试器对象的当前实例。该函数首先创建一个我们在上文中提到的 PaiMei crash_binning 工具的实例。这个工具可以简化对我们有帮助的、关于当前线程状态的属性的收集过程，在本案例中，当前线程就是我们正在处理的触发 ACCESS_VIOLATION 异常的线程。下一条语句调用 record_crash() ，透明地记录了以下信息。

---

[5] http://pedram.redhive.com/PaiMei/docs/PyDbg/

[6] http://pedram.redhive.com/PaiMei/docs/Utilities/

- **异常地址**：异常发生的地址，在本案例中，是导致了 ACCESS_VIOLATION 异常的地址。
- **写入违例**：一个标志，当发生了内存位置的写入异常时，该标志被设置。如果该标志没有被设置，说明发生的异常是读取内存的异常。
- **违例地址**：导致异常的读取或写入操作的地址。
- **违例的线程 ID**：发生异常的线程的数字标识。结合前 4 个属性，我们可以做出这样的描述："在线程 1234 内，0xDEADBEEF 处的汇编指令不能读取地址 0xC0CAC01A 的内容，导致应用终止。"
- **上下文**：发生异常时保存的 CPU 寄存器值。
- **反汇编**：导致异常的指令的反汇编形式。
- **周围指令的反汇编**：在特定地址处导致异常的指令前后各 5 条指令的反汇编形式。通过对周围的指令和上下文信息一起检查，分析者可以更好地了解为什么会发生异常。
- **栈展开**：产生异常的线程的调用栈。虽然在 32 位 Windows 平台上该信息不总是可用，但当该信息可用时，它可以向分析者显示程序是如何运行到当前地址的。另一种检查栈展开的方法是动态记录目标的行为，我们在第 23 章"模糊测试器跟踪"中对此进行了讨论。
- **SEH 展开**：发生异常时的结构化异常处理器（Structured Exception Handler，SEH）链。根据我们是否正在处理调试器在应用之前看到的异常（稍后将对此），分析者能够检查该链条来查看在目标中哪些函数被注册为异常处理函数。

此时，调试器应该再次向模糊测试器发送信号，让它知道上一个测试用例导致了一个需要进一步检查的异常。模糊测试器可以记录下这个事实，接着等待来自调试器的、指明目标已经被重新加载、准备好接受下一个模糊测试请求的信号。当调试器向模糊测试器发送信号后，它会打印出一个人工可读的、调用 record_crash() 时获取到的所有元数据的概要信息，并终止被调试对象。while 循环负责不断加载目标。关于 PyDbg 功能的更多具体信息，请参考第 20 章"内存的自动化模糊测试"。

64 位版本 Windows 上的栈展开

在 32 位 Windows 平台上进行调试时，不幸的是，许多情况下栈展开信息会丢失。展开栈的一般方法很简单。当前帧从栈偏移量 EBP 开始。下一帧的地址从 EBP 指向的指针读取。任意给定帧的返回地址存储在 EBP+4 的位置。从线程环境块（Thread

Environment Block，TEB）中可以通过FS寄存器[7]获取栈的范围。FS[4]指向栈顶的地址，FS[8]指向栈底的地址。当一个存在于调用链中的函数不使用基于EBP的栈帧时，这个过程就会终止。

不使用帧指针能够避免使用通用目的寄存器EBP，是一种编译器常用的优化技巧。然而，拥有基于EBP的帧的函数可以使用静态EBP寄存器加上偏移的方式来引用局部变量：

```
MOV EAX, [EBP-0xC]              ; EBP-based framing
MOV EAX, [SEP+0x44=0xC]         ; frame pointer omitted
```

当在展开的调用链中的任何地方使用上面的编译器优化技巧时，真实的链信息就会丢失。在新的64位平台上，微软通过将各种调用规范替换成一种调用规范，减轻了调用链信息丢失的痛苦，关于调用规范的全部细节都可以在MSDN上找到[8]。此外，在新的64位Windows平台上，每个非叶子函数（叶子函数指不调用任何函数的函数）都有存储在可移植可执行（Portable Executable，PE）文件[9]中的静态展开信息，允许在任何时候及时、全面、准确地进行栈展开。

现在你应该理解上面的示例代码了。拥有了上面的PyDbg脚本，你也就拥有了领先于当前商业模糊测试工具带有的模糊测试监视软件的技术。然而，我们还可以做得更好。当对软件，尤其是写得不好的软件进行模糊测试时，通常会面临要对数千个可能触发异常的测试用例进行分类和排序的任务，这一任务让人望而生畏。而经过对这些测试用例的深入分析，你可能会发现许多用例发现的是同一个问题，或者说，这个模糊测试器发现了同一个漏洞的多种触发方式。

486

24.4.2 更高级的调试器监视器

回到本章我们讨论的IMAP的例子，让我们在真实环境下重新考虑这个问题。考虑如下场景：我们已有一个定制的模糊测试器，该模糊测试器能够生成50000个不重复的测试用例。结合运行这个模糊测试器和创建的调试器脚本，我们发现有1000个测试用例导致了异常。作为研究者，我们现在的任务是手工地仔细检查每个用例，确定问题出

---

[7] http://openrce.org/reference_library/files/refrence/Windows%20Memory%20Layout,%20UserKernel%20Address%20Spaces.pdf

[8] http://msdn2.microsoft.com/en-us/library/7kcdt6fy.aspx

[9] http://www.uninformed.org/?v=4&a=1&t=sumry

在哪里。我们来看看第一组触发异常的用例。

```
Test case 00005:  x01 LOGIN %s%s%s%s%s%s%s%s%s%s%s%s...%s%s%s
                      EAX=11223300 ECX=FFFF7248...
                      EIP=0x00112233: REP SCASB
Test case 00017:  x01 AUTHENTICATE %s%s%s%s%s%s%s%s%s%s%s%s...%s%s%s
                      EAX=00000000 ECX=FFFFFF70
                      EIP=0x00112233: REP SCASB

Test case 00023:  x02 SELECT %s%s%s%s%s%s%s%s%s%s%s%s...%s%s%s
                      EAX=47392700 ECX=FFFEEF44...
                      EIP=0x00112233: REP SCASB
```

为了保持简洁，上面给出的片段中只包含了导出上下文的一部分信息。这 3 个用例中的异常都是因为尝试从非法内存地址进行读取的操作导致的(这部分信息没有显示出来)。完全基于这 3 个用例的输入，我们可以合理地猜测导致问题的原因要么是不正确地解析长字符串，要么是格式字符串漏洞。我们注意到这些访问违例都发生于执行同一地址的指令 REP SCASB 时，所以怀疑该模式可能存在于这 1000 个用例中的许多用例中。在进行了一些粗略的检查之后，该怀疑得到了证实。我们来详细分析一下。IA-32 体系结构上的 SCASB 汇编指令扫描一个字符串，找到第一个存储在 AL 寄存器（EAX 的首字节）中的字节。在以上显示的 3 个用例中，AL 的值都是 0。REP 指令循环执行该指令后方的指令，当 ECX 不为 0 时递减 ECX 寄存器的值直到其为 0。在显示的 3 个用例中，ECX 的值都很大。REP SCASB 是典型的用于确定字符串长度的汇编模式。看起来许多崩溃都是在扫描字符串、找出字符串终止符时发生的。

真让人沮丧。我们有 1000 个已知的会导致问题的测试用例，但它们中的大部分看起来触发的都是在同一个地方或接近同一个地方发生的问题。从 1000 个测试用例的完整列表中进行手工筛选完全不是人干的活（虽然你可以把这个活派给实习生干），那么，有更好的筛选办法吗？当然有，否则的话，我们一开始就不会把你引到这条道上来。

正如其名，PaiMei crash binning 工具能做的远远超过显示当前上下文相关的细节。crash binning 工具的设计目标主要是成为持久化的容器，能够保存许多崩溃相关的上下文信息。调用 record_crash() 成员函数会在 crash binning 内部维护的列表中创建一个新条目。每个新创建的条目包含它独特的相关上下文元数据。这种组织方式使得我们可以将前面给出的描述“在 50000 个测试用例中，其中 1000 个用例导致了异常”转化成更有用的描述：“在 50000 个测试用例中，其中 650 个用例导致了 0x00112233 处的异常，300 个用例导致了 0x11335577 处的异常，20 个用例导致了 0x22446688 处的异常”等等。

要达成这个目标，只需要对我们原先的脚本进行简单的修改即可。修改内容在下面代码中以粗体显示：

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

impot utils
crash_bin = utils.crash_binning.crash_binning()

def av_handler (dbg):
    globalcrash_bin
    crash_bin.record_crash(dbg)

    # 向模糊测试器发信号

    for ea in crash_bin.bins.keys():
        print "%d recorded crashes at %08x" % \
            (len(crash_bin.bins[ea]), ea)

    print crash_bin.crash_synopsis()
    dbg.terminate_process()

while 1:
    dbg = pydbg()
    dbg.set_callback(EXCEPTION_ACCESS_VIOLATION, av_handler)

    dbg.load(target_program, arguments)

    # 向模糊测试器发信号

    dbg.run()
```


为了实现持久化，我们将 crash_bin 的声明移到了全局命名空间中。当每次记录一个新异常时，我们增加了一部分逻辑，用于在已知的错误地址中进行迭代，显示在每个地址监视到的异常的数量。显示这些信息并非最有用的做法，这里仅作为例子呈现。作为一个练习，读者可以试着对这个脚本进行必要的修改，为每个发生错误的地址创建一个目录；在每个目录中，为每个记录到的崩溃创建一个文本文件，文件内容需要包含 crash_synopsis() 输出的、人工可读的内容。

为了进一步提高自动化和对优化崩溃的分类，我们可以扩展 crash binning 工具，使其引用我们记录到的崩溃的栈展开信息。在这种情况下，我们可以将存储的数据结构从

“列表的列表”扩展为“树列表”。树列表的数据结构使得我们可以通过路径及异常地址抓取测试用例，以及对其进行分组。请看图 24.4。

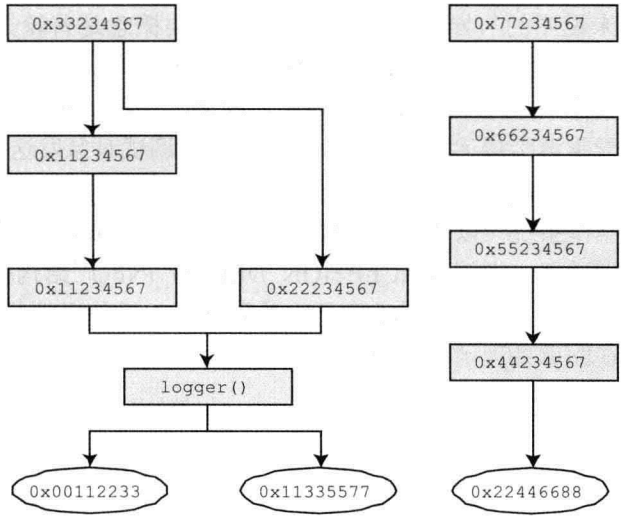


图 24.4　crash binning 栈展开分类的概念

图 24.4 中的椭圆框内的数据表示以前的异常地址类别。方框内的数据表示来自栈展开的地址。通过将栈展开数据加入我们的考虑范围，就可以开始查看导致不同错误地址的路径。从根本上说，我们可以将上一个描述“在 50000 个测试用例中，650 个用例导致了 0x00112233 处的异常，300 个用例导致了 0x11335577 处的异常，20 个用例导致了 0x22446688 处的异常”改进为这样的描述：“在 50000 个测试用例中，650 个用例导致了 0x00112233 处的异常，其中 400 个用例经过路径 x、y、z，250 个用例经过路径 a、b、z。”更重要的是，以这样的视角进行分析使得我们可能确定重复错误的来源。在这个例子中，我们捕获到的大量异常都发生在地址 0x00112233 和地址 0x11335577 处。到达这两个地址的路径成百上千，为了简便起见，我们仅列出了其中一些路径。图 24.4 中到达这两个点的路径都经过了 logger() 例程。logger() 例程会是这些问题的根源吗？或许 logger() 例程中的确存在格式字符串漏洞。由于我们的 IMAP 服务器中的许多位置都有对 logger() 例程的调用，因此我们有理由相信，向多个 IMAP 动词传入格式字符串都能够导致同样的问题。虽然只有通过深入分析才能确定这些问题是否由 logger() 例程引起，但从上面的描述中，我们可以立即看到在人工分析前进行自动化分析的好处。同样，我们还是把实现该附加功能所需的修改作为练习留给读者。

489

24.5 首轮异常与末轮异常

目前，我们仍未说明的与异常处理有关的一个重要概念是首轮异常（First-Chance Exception）和末轮异常（Last-chance Exception）。当由调试器加载的目标进程发生异常后，微软 Windows 操作系统会向调试器发送“首轮提醒（first-chance notification）”[10]。调试器决定是否将异常继续传递给被调试对象。当向被调试对象发送异常时，有两种可能：如果被调试对象能够处理异常，它会处理异常并继续正常的执行；如果被调试对象不能处理该异常，操作系统会以“末轮提醒（last-chance notification）”的方式再次将这个异常信息发送给调试器。根据 EXCEPTION_DEBUG_INFO[11]结构中的某个值，我们可以知道调试器收到的异常是首轮异常还是末轮异常。如果使用的是 PyDbg 库，我们可以用下面的方式在回调处理函数内检查这个值：

490

```
def access_violation_handler (dbg):
    if dbg.dbg.u.Exception.dwFirstChance:
        # first chance
    else:
        # last chance
```

如果 dwFirstChance 的值不为零，就说明这是调试器第一次遇到这个异常，被调试对象还没有看到和处理该异常。

对我们而言，这意味着什么？这意味着我们要做一个决定，决定是否要忽略首轮异常。例如，假定我们再次对示例 IMAP 服务器进行审计，下面的代码片段是 `logger()` 例程中造成格式字符串漏洞的原因：

```
void logger (char *message)
{
    ...
    try
    {
        // 存在格式字符串漏洞的调用
        fprintf(log_file, message);
    }
```

---

[10] http://msdn.microsoft.com/msdnmag/issues/01/09/hood/

[11] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50lrfexceptiondebuginfo.asp

```
    except
    {
        fprintf(log_file, "Log entry failed!\n)";
    }
    ...
}
```

包围在存在漏洞的 fprintf() 调用周围的 try/except 语句块创建了一个异常处理器。如果第一个 fprintf() 语句执行失败，第二个 fprintf() 语句就会被调用。在这个例子中，如果我们选择忽略首轮异常，就不可能发现这个格式字符串漏洞，因为 IMAP 服务器通过 try/except 方式对它自己的错误进行了优雅的处理。但这并不意味着该漏洞不存在或不可以被利用！简单地说，在这种情况下我们漏掉了一个可能的问题（漏报）。另一方面，在许多情况下，应用本身会对它自身的大部分异常进行良好的处理，自行消除由异常导致的安全问题。如果我们选择处理首轮异常，我们将不得不处理大量的无须处理的异常（误报）。如果你是一个独立的安全研究者，选择处理首轮异常或选择不处理首轮异常都不是绝对正确的决定。考虑到这一点，你可以依据经验，自行决定在给定测试目标上选择哪个方向：容忍漏报还是容忍误报。然而，如果你是一个以保证自己产品的安全为目的的软件开发者或 QA 工程师，更安全的做法是监视首轮异常，并在每个触发异常的测试用例中花些时间来研究异常的原因。

24.6　动态二进制插装

调试器辅助的监视技术对我们大有帮助。然而，正如第 5 章“有效模糊测试的需求”中提到的那样，错误检测的最佳方案是动态二进制插装（Dynamic Binary Instrumentation，DBI）。深入讨论这个主题需要开发原型工具，其内容已经超出了本书的范围。但是，本节提供的信息能够帮助你更清楚地理解某些当前可用的高级调试工具特性的原理。动态二进制插装引擎可以在最小的逻辑片段（也就是基础块）上对一个可执行目标进行插装。

在第 23 章中，我们将基础块定义为一个指令序列，当该序列中的第一条指令被执行之后，序列中的每条指令都一定会按顺序依次执行。动态二进制插装系统允许你通过增加、修改或变换指令逻辑来对每个基础块中的每条指令进行插装。通常通过将执行控制流从原先的指令集移动到修改后的代码缓存中来实现插装。在有些系统上，可以使用更高层次的类 RISC 的伪汇编语言完成指令级的插装。这使得开发者能够更容易地开发跨体系结构运行的动态二进制插装工具。动态二进制插装系统提供的 API 使得我们可

以在更广阔的范围内应用该技术，从程序分析和优化到机器翻译和安全监视。

如今有不少可用的动态二进制插装系统，例如 DynamoRIO[12]、DynInst[13]和 Pin[14]。以 DynamoRIO 系统为例，该系统是麻省理工大学和惠普公司的合作项目。DynamoRIO 在 IA-32 体系结构上实现，支持微软 Windows 和 Linux 系统。DynamoRIO 非常稳定，具有高性能，其能力已经被 Determina[15]的 Memory Firewall[16]之类的工具证明。关于 Determina 的 DynamiRIO 工具，其使用信息可以从麻省理工大学发表的名为“基于程序监视的安全执行”[17]的研究论文中找到。Pin DBI 是一个非常有趣的项目，因为和大多数动态二进制插装系统不同，该工具允许开发者开发一个能够附着到目标进程上的工具，而不局限于在动态二进制插装系统的控制下加载目标进程。

492

应用动态二进制插装技术开发监视工具，使得我们可以将检测错误的能力从检测错误发生推进到检测可能的错误源。以前面我们用过的堆溢出的代码为例：

```
char *A = malloc(8);
char *B = malloc(16);
char *C = malloc(24);

strcpy(A, "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP");
free(B);
```

使用基于调试器的监视方法，当 strcpy() 调用导致实际发生溢出时，我们无法检测到溢出。对于前面我们所演示的栈溢出，使用基于调试器的监视方法是可以检测到的。在前面演示的栈溢出的情况下，当受影响的函数返回时，系统可能会生成一个异常，而调试器能够检测到该异常。然而，在这个例子和其他堆溢出的情况下，只有当对堆进行后续操作，例如发生以上代码片段中的 free() 函数调用，才会发生异常。基于动态二进制插装，可以在运行时跟踪和记录（fence）所有的内存分配。名词 fence 指的是标记和记住每个堆块的开始位置和结束位置。通过插装所有进行内存写操作的指令，我们可以向应用中加入检查，确保没有发生越过堆块边界的内存写操作。当对某个被分配的堆块进行操作时发生了超出（overrun），哪怕超出一个字节，我们都会发出通知。成功地实

---

12 http://www.cag.lcs.mit.edu/dynamorio

13 http://www.dyninst.org/

14 http://rogue.colorado.edu/pin/

15 http://www.determina.com

16 http://www.determina.com/products/memory_firewall.asp

17 http://static.usenix.org/events/sec02/full_papers/kiriansky/kiriansky_html/

现这种技术能够帮你节约花在分析上的大量时间、精力及金钱。

学习开发定制的动态二进制插装工具并不那么容易。幸运的是，在你之前，有不少人已经经历了创建可以集成到模糊测试器中的工具的麻烦，因此目前已经有一些这方面的可用工具。商业工具有 IBM Rational Purify[18]、Compuware DevPartnerBoundsChecker[19]、OC Systems RootCause[20]及 Parasoft Insure++[21]等。实际上，Purify 建立在静态二进制插装（Static Binary Instrumentation，SBI）的基础上，而 BoundsChecker 则结合使用了静态二进制插装和动态二进制插装技术。这两个工具都能够提供我们所需要的错误检测和性能分析特性。开源世界中最著名的方案应该是 Valgrind[22]（读作/vælgr□nd/）。Valgrind 提供了一个动态二进制插装系统，以及相当多的事先开发好的工具，如 Memcheck，该工具可用于定位内存泄漏和堆访问越界漏洞。Valgrind 有相当多的第三方插件。对我们来说，Annelid[23]这个进行边界检查的插件是最重要的。

在大多数情况下，使用调试器辅助的自动化技术已经使我们在当前模糊测试技术的基础上前进了一步。但是，我们仍然鼓励读者探索一些更高级的错误检测方法。

24.7　小结

在本章中，我们讨论了监视模糊测试目标的方法，覆盖了从最基本的高层技术到更高级的调试器辅助技术。此外，我们还介绍了动态二进制插装技术，介绍了各种动态二进制插装系统和一些立即可用的方案，读者可以以此为基础探索更高级的监视目标的方法。

有了本章介绍的知识，你应该具有了能够融合定制的调试器监视工具与高层模糊测试器单步执行工具的能力。利用这两类工具的组合，读者可以准确地知道哪些单独的测试用例或测试用例组导致了你的目标偏离了正轨。有了这些自动化带来的好处，你可以把你的实习生从模糊测试结果的事后分析中解放出来，让他们能够回来帮你煮咖啡。

---

18 http://www-306.ibm.com/software/awdtools/purify

19 http://ww.compuware.com/products/devpartner/visualc.htm

20 http://www.ocsystem.com/prod_rootcause.html

21 http://www.parasoft.com/jsp/products/home.jsp?product=Insure

22 http://valgrind.org/

23 http://valgrind.org/downloads/variants.html?njn

第四部分

展　望

第 25 章　我们学到了什么
第 26 章　展望

第25章

我们学到了什么

"Rarely is the question asked: Is our children learning?"

——George W. Bush, Florence, SG, January 11, 2000

我们希望读者已经清楚地知道什么是模糊测试，模糊测试为什么有效，以及如何应用模糊测试发现程序代码中隐藏的错误。在本书的前言中，我们提到过这本书面向三类可以从模糊测试中获益的受众：开发者、QA 工程师和安全研究者。在本章中，我们将对分解和分析软件开发生命周期，明确每类读者分别可以在开发的哪些阶段应用模糊测试以构建安全的软件。

25.1 软件开发生命周期

模糊测试曾一度是一种几乎仅由安全研究者在产品发布后应用的技术，但如今软件开发者已经开始学习使用模糊测试，在软件开发生命周期的早期发现漏洞。微软已经将模糊测试作为他们的可信计算安全开发生命周期（Trustworthy Computing Security Development Lifecycle）[1]的关键部分。他们的方法鼓励开发者在被称为安全开发生命周期（Security Development Lifecycle，SDL）的实现阶段“应用包含模糊测试工具在内的安全测试工具”。微软定义了“安全开发生命周期”的概念，该概念以及它与软件开发

[1] http://www.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/sdl.asp

生命周期的各个阶段的对应关系如图 25.1 所示。

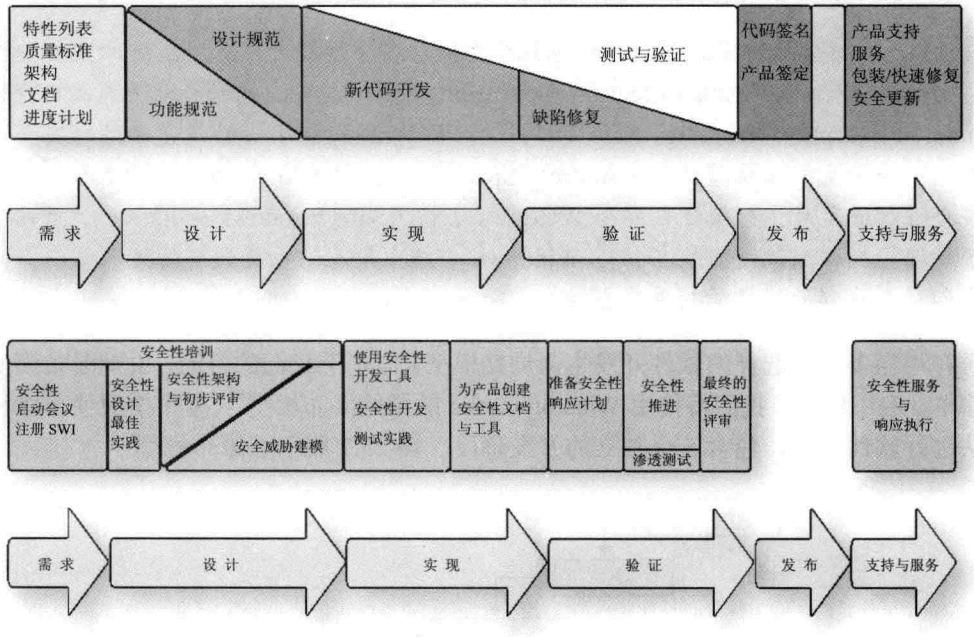


图 25.1　微软的软件开发生命周期与微软的安全开发生命周期

从图中这些并列的过程中，读者可以看到，微软在软件开发生命周期的每个阶段都识别出了适当的安全问题。这种“必须在整个软件开发生命周期中集成安全性”的认识非常重要。另一个必须实现的，但却没有在图中反映出来的是安全性必须渗透在整个软件开发生命周期中，而不是仅作为一个并行过程存在。只有依靠开发者、QA 团队、安全研究者的协作才能生产出安全的代码。

499

微软关注安全性

微软将安全性紧密地集成在了它自己的软件开发生命周期中，这一点丝毫不令人感到意外。由于微软占有统治性的市场份额，因此它的技术是安全研究者们的主要目标。虽然还存在争论，例如有人认为微软在安全性方面还有很长的路要走，但毫无疑问，微软已经在提高软件安全性方面迈出了一大步。

作为一个例子，考虑微软互联网信息服务软件（Internet Information Sevices，IIS）

[2]，也就是微软的 Web 服务器。IIS 5.x 有 14 个被公开的漏洞[3]。而在 2003 年初发布的 IIS 6.x 版本中，则仅有 3 个已知的漏洞[4]，其中没有一个是关键级别的。

IIS 产品安全性的提高可以部分地归结为低层安全性提升，如/GS 缓冲区安全性检查[5]、数据执行预防（Data Execution Prevention，DEP）、安全的结构化异常处理（Safe Structured Exception Handling，SafeSEH）[6]，以及 Windows Vista 的最受期待的安全性改进之一：地址空间布局随机化（Address Space Layout Randomization，ASLR）[7]。在这些安全性改进之外，微软还在安全 Windows 小组（Safe Windows Initiative）[8]项目中投入了资源。模糊测试是微软在安全 Windows 小组中投入的人员使用的技术之一，提高了我们今天看到的微软的软件安全性。

有相当多可供选择的软件开发生命周期模型，但基于我们的目的，由于瀑布模型足够简单，应用广泛，我们将使用 Winston Royce 的原始瀑布模型[9]。瀑布模型使用串行的方法进行软件开发，包含 5 个独立的开发阶段。图 25.2 展示了瀑布模型。

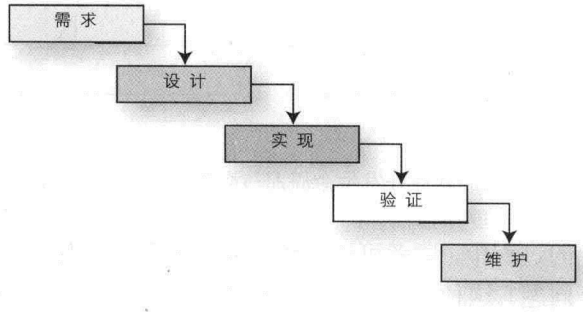


图 25.2　Royce 的原始瀑布模型

接下来，我们考虑如何在瀑布模型的各个阶段实现模糊测试。

---

[2] http://www.microsoft.com/WindowsServer2003/iis/default.mspx

[3] http://secunia.com/product/39/?task=advisories

[4] http://secunia.com/product/1438/?task=advisories

[5] http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx

[6] http://en.wikipedia.org/wiki/Data_Execution_Prevention

[7] http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf

[8] http://www.microsoft.com/technet/Security/bestprac/secwinin.mspx

[9] http://en.wikipedia.org/wiki/Waterfall_process

25.1.1 分析阶段

我们在分析阶段收集一个项目开始之前所需要的所有信息。在这个阶段，我们和最终用户一起工作，确定他们真正的需求。虽然在软件开发生命周期的这个阶段中不会直接用到模糊测试，但这是个重要的阶段，因为开发者、QA 团队、安全专家会在这个阶段考虑是否将模糊测试当作适合后续过程的测试工具。

25.1.2 设计阶段

设计一个应用包括生成项目的抽象表示。根据被建模对象的不同，我们需要针对软件的不同部分使用不同的建模技术，如硬件建模、软件建模、数据库建模等。编码阶段使用的编程语言和软件库也是在这个阶段决定的。

做完设计决定后，我们开始关注模糊测试的可行性及可能的模糊测试方法。在这个阶段，我们需要定义两个细节，这两个细节会影响整个模糊测试方法。第一个要定义的细节是在何种软件和硬件平台上实现模糊测试。这个决定会影响可使用的模糊测试工具和可应用的模糊测试类型。例如，如果当前的项目是一个在 Linux 操作系统上运行的控制台应用，环境变量模糊测试就是合适的模糊测试方法。如果该项目被设计为在 Windows 操作系统上使用，那么，它是否有被标记为“脚本安全”的 ActiveX 控件可以被外部网站所使用？如果答案是“是”，我们就应该使用 COM 对象模糊测试方法。

在软件开发生命周期的这个阶段，我们可能还需要决定进程间的网络通信协议。例如，如果某特定项目需要一个内建的实时通信解决方案，我们可能会选择可扩展通信和表示协议（Extensible Messaging and Presence Protocol，XMPP）。XMPP 协议是一种开放的 XML 通信技术，由 Jabber[10]开源社区于 1999 年开发。在软件开发生命周期中，设计阶段是适合将协议规范传递给模糊测试团队的合适阶段。同时，也适合在该阶段分配一些资源来搜索和枚举当前可用的 XMPP 测试工具和曾经被发现过的 XMPP 解析漏洞。

另一个在设计阶段需要做的重要决定与将要开发的应用中的输入向量相关。记住，模糊测试只不过是向应用发送非常规的数据，观察所发生情况的一种技术。因此，识别应用中所有的输入向量并决定对其进行最佳的模糊测试才是关键。在准确地确定模糊测试使用的输入向量时，创造力很重要。有些问题仅在后台发生，不暴露在最终用户面前

501

[10] http://www.jabber.org/

并不意味着安全。因为攻击者能够定制代码来访问接口。进行模糊测试时考虑的输入向量越完整，通过模糊测试达到的总代码覆盖率就越高。

25.1.3 编码阶段

在编码阶段，开发团队创建项目所需的组件，逐步将它们集成为应用。这里的要点是，开发者在开发中可以且应当使用模糊测试。为发现错误而进行测试的责任不应该仅由 QA 团队和安全专家承担。

代码被编译并进行了恰当的功能性测试后，也应该对其进行足够的安全测试。对设计阶段识别的任何可能的输入向量，都可以进行模糊测试。如果应用提供的 ActiveX 控件已经就绪，就可以开始对其进行模糊测试。如果应用的网络服务部分已经完成，也可以开始对其进行模糊测试。在第 19 章“内存的模糊测试”和第 20 章“内存的自动化模糊测试”中，我们看到可以在函数级别应用内存模糊测试。如果一个函数被设计为从用户输入的字符串中提取 E-mail 地址，就可以使用大量随机的和精心选择的字符串值，重复调用这个函数数千次。

不完善的输入验证通常是导致安全漏洞的原因，模糊测试能够帮助开发者更好地理解为什么输入验证如此关键。模糊测试被设计来识别开发者没有考虑到的，应用不能正确处理的输入。当开发者看到模糊测试发现的错误后，他们会在最恰当的位置建立合适的输入验证控制，并确保在后续代码中不会犯同样的错误。更重要的是，在软件开发生命周期中越早发现漏洞，修复漏洞的费用就越低。如果开发者能够自己找到自己的错误，修复错误就会容易得多。

25.1.4 测试阶段

当软件开发进入测试阶段时，QA 团队就开始介入了。在项目发布前，安全团队也应该参与到测试阶段。如果能在软件发布前发现缺陷，缺陷修复的费用总是会更低。另外，确保漏洞在内部发现，而不是被发布在公开的邮件列表中，对保护公司的声誉也是有好处的。

对 QA 团队和安全研究者而言，决定如何在测试阶段最有效地利用对方的资源、如何一起工作很重要。对有些软件开发商来说，在软件开发生命周期中引入安全专家也许是新方法，但增长的公开发布的漏洞数量说明了引入安全专家的必要性。

QA 团队和安全研究者都需要关注可在测试阶段使用的商业和开源的模糊测试工

具。可以使用同行培训（peer training）来交换团队成员之间的这类知识。相对来说，安全研究者更可能了解最新的和最好的模糊测试工具，他们可以利用这些知识培训 QA 团队和开发者，使得 QA 团队和开发者了解这些可以帮助提高安全性的工具。作为培训的额外好处，开发者可以了解如何避免的常见编码错误，获得对安全编程实践的强有力的理解。

25.1.5 维护阶段

尽管开发者、QA 团队和安全研究者在产品开发过程中尽了最大的努力，但在最终发布的产品中，我们仍然会发现导致异常和关键安全漏洞的错误。主要的软件开发商在安全性上投入了数十亿美金，虽然产品的安全性确实得到了提升，但我们仍然需要面对有漏洞的代码。因此，即使代码已经变成了被发布的产品，我们仍然需要积极地研究安全漏洞。新的模糊测试工具、具有越来越渊博知识的研究者和改进的模糊测试技术能够从新的视角检查以前已经测试过的代码，发现新的漏洞。甚至，安全性可以依赖于具体的实现方式，因此在修改配置后，发布的代码也可能会出现漏洞。举例来说，某些问题可能不会在 32 位平台上出现，但会在 64 位平台上出现。

除了在内部识别有漏洞的代码外，建立与独立研究者协同工作的流程很重要。已经公开发布的大部分软件安全漏洞都是由独立研究者发现的。虽然安全研究者们发现漏洞的动机各不相同，但无论如何，他们的努力对于改进产品代码的安全性很重要。软件开发商应该确保建立流程，允许独立研究者向负责连接研究者和开发者之间的团队报告他们的发现，而开发者最终负责修复这些缺陷。

25.1.6 在软件开发生命周期中实现模糊测试

瀑布模型提供了简化的开发流程，但其对开发过程的描述过于简单。大多数项目需要多团队协作，需要经历循环的软件开发生命周期阶段，而不是瀑布模型描述的顺序阶段。但无论如何，任何开发模型都是瀑布模型描述的基本阶段的某种组合形式。我们无须聚焦于模型本身，而是要利用我们本章讨论的内容，在软件开发生命周期中实现模糊测试，在代码发布为产品之前更好地提高代码的安全性。

25.2 开发者

在一个新项目中，开发者是实现安全性的第一道防线，开发者应该具有对新开发的

函数、类和库进行模糊测试的能力。通过培养开发者识别基本安全性问题的技能，许多代价高昂的安全性问题可以立即被发现，根本就不会到达 QA 或安全团队，更不要说被发布给用户。

通常，开发者会在一个他们感觉舒服的集成开发环境（IDE）中完成大部分编程工作。例如，微软的 Visual Studio 是在 Windows 平台上进行 C#和 Visual Basic 开发的标准 IDE，而 Eclipse 则是 Java 和其他一些编程语言的流行 IDE 环境。我们建议的一种激励开发者将模糊测试集成到他们的开发实践中的方法是，为他们使用的 IDE 开发安全测试插件，而不是要求开发者学习一套全新的安全测试工具。尽管 SPI Dynamics 的 DevInspect[11]不是一个专门的模糊测试工具，但它展示了如何使用该方法。DevInspect 是一个扩展 Visual Studio 和 Eclipse 功能的工具，为开发者提供源代码分析功能，以及为 ASP.NET 和 Java Web 应用提供黑盒测试功能。

25.3 QA 研究员

传统上 QA 专注于功能测试而不是安全性测试。幸运的是，随着最终用户越来越关注不安全软件导致的风险，他们对软件安全性的要求越来越高，因此，QA 的关注点正在发生改变。虽然我们不应该期望 QA 团队成为安全专家，但 QA 团队应该具有基础的安全知识，以及更重要的，知道何时该引入安全专家。理论上，模糊测试具有高度的自动化，适合 QA 团队来进行。但逆向工程并不是需要 QA 工程师掌握的合适技能，因为它是一种高度专业的技能，需要持续的培训。反而，要求 QA 团队运行模糊测试工具更加现实。在测试阶段，虽然也许需要安全团队的协作才能确定一个特定的异常是否会导致可被利用的条件，但团队仍然值得对任何类型的崩溃进行审核，并确保开发者解决问题。从这个角度说，安全性问题和功能性问题之间的差异并不关键。关键的是立即着手修复这个问题。

25.4 安全研究者

对安全研究者来说，模糊测试并不是什么新东西。由于模糊测试相对简单并能带来高回报，该方法已经被广泛应用于发现安全漏洞。过去，软件开发生命周期中根本不会包含安全研究者。直到现在，许多组织中仍然是这种情况。如果你的软件开发生命周期

[11] http://www.spidynamics.com/products/devinspect/

中没有包含安全专家，现在是时候重新回顾你的软件开发过程了。例如，微软通过举办蓝帽子安全简报（BlueHat Security Briefings）[12]将安全知识从著名的安全研究者传递给它的开发团队。

安全研究者面临的挑战是需要将他们的注意力转移到更早地发现安全漏洞上。不管在什么时候发现漏洞，为漏洞打补丁总是有价值的，但是随着项目的进行，为漏洞打补丁的成本会不断上升，如图 25.1 所示。安全团队可以使用他们的专门技能，调整测试过程，持续地改进模糊测试工具和使用的模糊测试技术。虽然 QA 团队负责运行标准的模糊测试工具和脚本，但安全研究者可以负责创建第三方工具，识别新的安全测试技术并持续地教育 QA 团队和开发者如何应用这些技术。

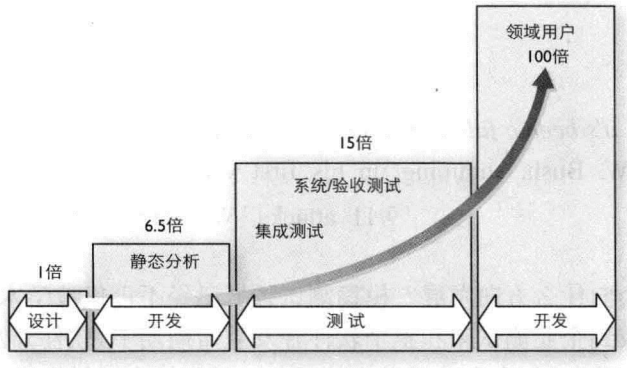


图 25.3　贯穿软件开发生命周期的软件缺陷的修复成本

25.5　小结

现在已经不再是软件开发团队中的每个人都可以说“软件安全性不是我的问题，我们有专门的团队对安全性负责”的时代了。安全性是每个人的责任，无论你是开发者，QA 团队成员，还是安全研究者。作为产品经理，简单地无视漏洞并期望它们会魔术般消失是不可能的。你的人必须利用过程和工具的能力将安全性嵌入软件开发生命周期中，而模糊测试则恰好是可以被以上三类人员使用的，可被自动化的过程。模糊测试不是万能的，但你可以用这种技术开发用户友好的，能够发现大范围内的漏洞的安全性工具。

---

[12] http://www.microsoft.com/technet/security/bluehat/sessions/default.mspx

第26章

展　望

"But all in all, it's been a fabulous year for Laura and me."

——George W. Bush, summing up his first year in office, three months after the 9/11 attacks, Washington, DC, December 20, 2001

模糊测试将会向什么方向发展？模糊测试领域已经不再仅被学术界和黑客界了解，它正在开始被集成到企业测试床和整个软件开发生命周期中。软件开发者对模糊测试的接受度日渐增加，因此，如今有些开发中的商业工具开始应用模糊测试这种强大的方法也不足为奇。在本章中我们来看看模糊测试的现状，并试图预测不久的将来模糊测试会如何发展。

26.1　商业工具

商业玩家进入某个领域，通常就标志着一个行业开始成熟。而这正是目前我们在模糊测试技术领域见到的趋势。大的软件开发商，例如微软，已经接纳模糊测试并将其作为在软件开发生命周期的早期识别安全漏洞的方法，同时，对稳定的、用户友好的模糊测试器的需求还会催生新的公司和产品。在本节中，我们来看看一些已经存在的商业工具，这些工具在下文中以字母顺序列出。

26.1.1 Beyond Security 公司的 beSTORM[1]

508

beSTORM 是一个黑盒测试工具，该工具利用模糊测试在各种基于网络的协议中查找漏洞。Beyond Security 公司的创始人最初参与创立了 SecuriTeam[2]门户网站，该网站发布安全相关的新闻和资源。beSTORM 能够测试的协议包括下面这些：

- HTTP 协议——超文本传输协议
- FrontPage 扩展
- DNS 协议——域名系统
- FTP 协议——文件传输协议
- TFTP 协议——一般文件传输协议
- POP3 协议——邮局协议 v3
- SIP 协议——会话初始化协议
- SMB 协议——服务器消息块
- SMTP 协议——简单邮件传输协议
- SSLv3——安全套接字层 v3
- STUN—（通过网络地址转换（NATs）简单遍历用户数据报协议（UDP））
- DHCP 协议——动态主机配置协议

beSTORM 能够运行在 Windows、UNIX 和 Linux 平台上，有独立的测试和监视组件[3]。在下一代的商业模糊测试技术中我们期望看到包含在模糊测试工具内的目标监视组件。beSTORM 中的监视组件是一个定制的调试器，目前仅支持 UNIX 和 Linux 平台。图 26.1 显示了 beSTORM 的屏幕截图。

26.1.2 BreakingPoint Systems 的 BPS-1000[4]

509

作为模糊测试领域的最新进入者，BreakingPoint 公司以其在单纯的流量生成方面的统治地位在市场上占有了一席之地。BPS-1000 是一个定制的硬件解决方案，具有每秒在 50 万个 TCP 会话上生成超过 500 万个 TCP 数据包的能力。虽然 BPS-1000 不是传统

---

1 http://www.beyondsecurity.com/

2 http://www.securiteam.com/

3 http://www.beyondsecurity.com/beSTORM_FAQ.pdf

4 http://www.breakingpointsystem.com/

意义上的模糊测试工具，但该工具能够对所生成的数据进行变异，从而达到检测错误的目的。

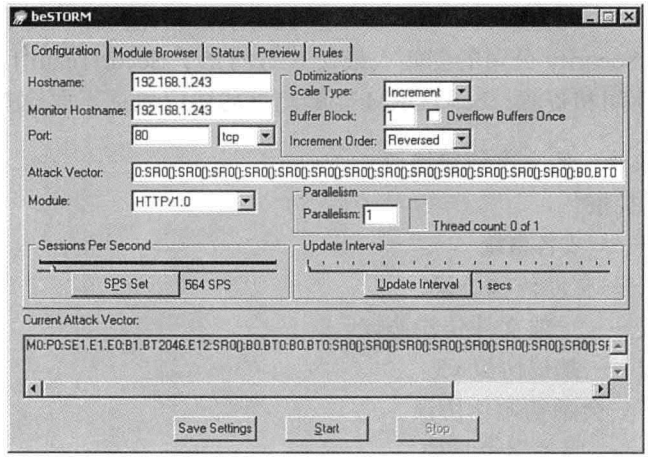


图 26.1　Beyond Security 的 beSTORM

510

前面板上的交流电源适配器是 BPS-1000 的一个有趣的设计（如图 26.2 所示），在测试设备不能继续测试时，该设计用于重启设备。

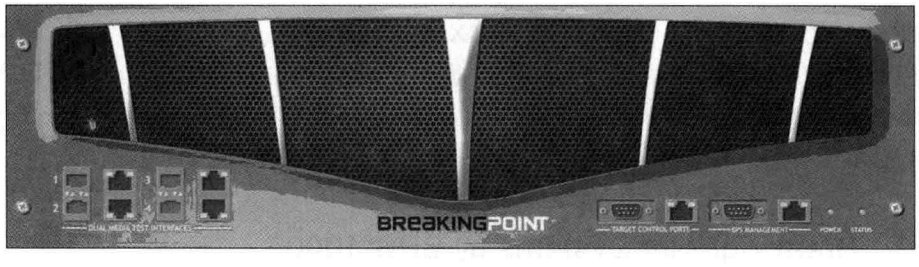


图 26.2　BreakingPoint Systems 的 BPS-1000

26.1.3　Codenomicon[5]

Codenomicon 工具提供的软件模糊测试解决方案也许是最广为人知的。Codenomicon 公司的创始人来自 PROTOS[6]测试套件项目，该项目起初由位于芬兰

[5] http://www.codenomicon.com/

[6] http://www.ee.oulu.fi/research/ouspg/protos/

Linanmaa 的奥卢大学赞助。PROTOS 首次获得公众的关注是在 2002 年，当时该项目宣布在各种 SNMPv1 的实现中发现了大量的安全漏洞。PROTOS 的开发者研究了 SNMPv1 协议并枚举了请求和捕获得到的数据包的所有可能的变异。然后他们开发了一系列的事务集合，引入了异常元素（exceptional elements）的概念，将该概念定义为“在实现软件的时候没有仔细考虑的输入”[7]。在某些情况下，这些异常元素违反了协议标准，而在另一些情况下，异常元素遵循了协议但却包含某些会破坏实现得不好的解析器的内容。PROTOS SNMPv1 测试套件得到的结果吸引了大量关注，因为该套件识别出了大量的漏洞，导致几十家受影响的硬件和软件厂商发布补丁[8]。从进入公众的视野开始，PROTOS 项目一直不断地发展，如今该项目已经包含了针对大量网络协议和文件格式的测试套件，包括下面这些协议：

- WAP——无线应用协议
- HTTP——超文本传输协议
- LDAPv3——轻量目录访问协议（版本 3）
- SNMPv1——简单网络管理协议（版本 1）
- SIP——会话初始化协议
- H.323——视频会议中常用协议的集合
- ISAKMP——互联网安全组织和密钥管理协议
- DNS——域名系统

Codenomicon 商业产品所使用的测试方法主要来自 PROTOS。值得一提的是，该商业测试套件覆盖了大量的网络协议和文件格式，同时提供了如图 26.3 所示的友好的图形用户界面。

Codenomicon 的许可方式是为每个独立的协议提供单独的报价。在写作本书时，每个协议的零售价格约为 30000 美金。这种定价方式说明这家公司面向的是仅需要针对自身产品用到少数几种协议套件的开发者，而不是对大范围的产品和协议感兴趣的安全研究员。该产品最大的局限性是缺乏监视目标的能力。Codenomicon 利用已知的“好”的测试用例在测试中检查目标的健康状况。我们在第 24 章“智能错误检测”中提到过，这种方法仅仅是监视技术中最初级的技术。

---

[7] http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/index.html#h-ref2

[8] http://www.cert.org/advisories/CA-2002-03.html

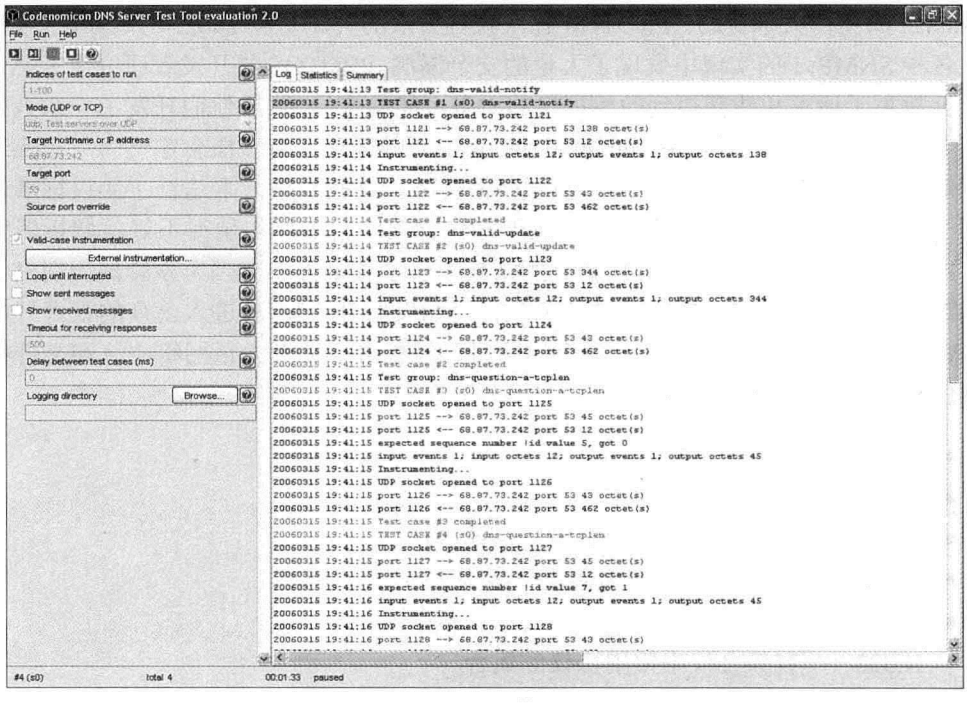


图 26.3 Codenomicon DNS 服务器测试工具

26.1.4 GLEG ProtoVer 专业版[9]

GLEG 是一家俄罗斯公司，由 VulnDisco[10]发展而来，VulnDisco 是一个每月发布新发现的漏洞的服务，该服务以 Immunity 公司的 **CANVAS**[11]**可利用框架**模块的形式发布漏洞。GLEG 后来拓展了自己的产品，开发了自己的模糊测试器，该模糊测试器可被用于发现许多通过 VulnDisco 服务发布的漏洞。GLEG 公司最终的产品是 ProtoVer 专业版，采用 Python 语言开发。在写作本书时，该工具支持以下协议：

- IMAP——互联网消息访问协议
- LDAP——轻量目录访问协议

[9] http://www.gleg.net/protocer_pro.shtml

[10] http://www.gleg.net/products.shtml

[11] http://www.immunitysec.com/products-canvas.shtml

- SSL——安全套接字协议
- NFS——网络文件系统

ProtoVer 测试套件包括多种操作界面，如命令行操作界面、图形用户界面，以及如图 26.4 所示的基于 Web 的操作界面。ProtoVer 由安全研究员开发，看起来也是面向安全研究员的，目前的价格是 4500 美金（一年内所有支持的协议的许可费用）。邮件支持的费用也包含在这个许可费用内。

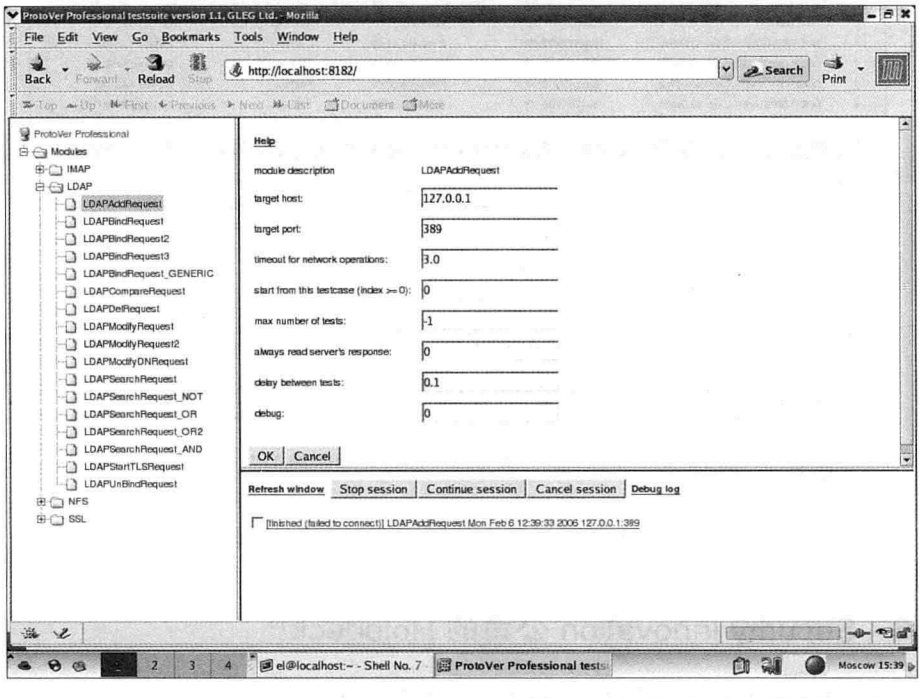


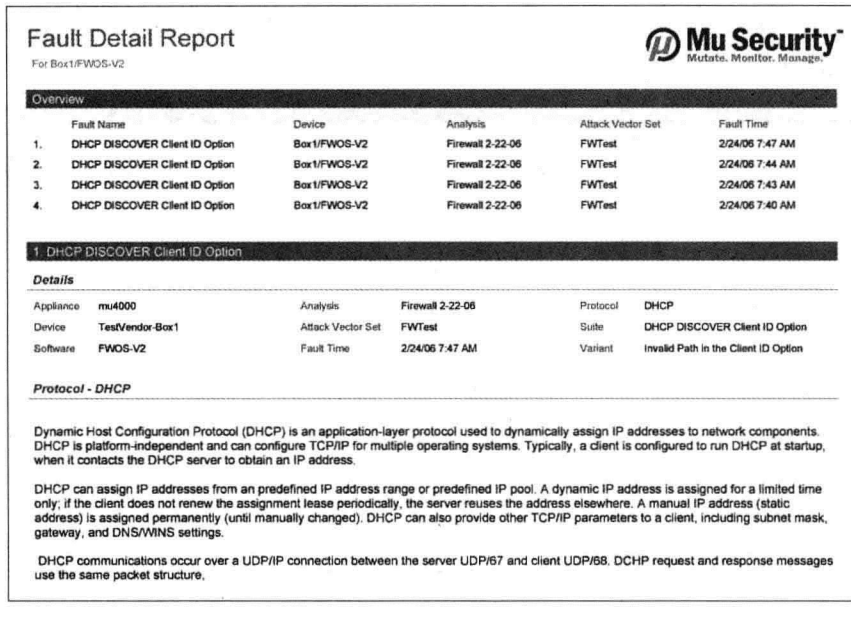
图 26.4　GLEG ProtoVer 专业版测试套件的 Web 界面

26.1.5　MU Security 公司的 MU-4000[12]

MU Security 公司提供的 MU-4000 是一套硬件设备，该工具既能对网络数据进行变异，也能监视目标以发现错误。MU-4000 主要设计为测试其他网络设备，而且，与 BreakingPoint 的方案一样，它也具有重启目标设备、从检测到的错误中恢复的能力。

[12] http://www.musecurity/products/mu-4000.html

MU-4000 的另一个有趣的功能是：当在基于 Linux 的可执行文件中发现错误时，它能够作为外部安全性触发器自动生成针对该漏洞的利用代码（概念验证性的）。MU 公司宣称 MU-4000 能够处理任何基于 IP 的协议。图 26.5 展示了该工具在某个 DHCP 实现中发现异常的细节报告。



Fault Detail Report

For Box1/FWOS-V2

Mu Security
Mutate. Monitor. Manage.

Overview

	Fault Name	Device	Analysis	Attack Vector Set	Fault Time
1.	DHCP DISCOVER Client ID Option	Box1/FWOS-V2	Firewall 2-22-06	FWTest	2/24/06 7:47 AM
2.	DHCP DISCOVER Client ID Option	Box1/FWOS-V2	Firewall 2-22-06	FWTest	2/24/06 7:44 AM
3.	DHCP DISCOVER Client ID Option	Box1/FWOS-V2	Firewall 2-22-06	FWTest	2/24/06 7:43 AM
4.	DHCP DISCOVER Client ID Option	Box1/FWOS-V2	Firewall 2-22-06	FWTest	2/24/06 7:40 AM

1. DHCP DISCOVER Client ID Option

Details

Appliance	mu4000	Analysis	Firewall 2-22-06	Protocol	DHCP
Device	TestVendor-Box1	Attack Vector Set	FWTest	Suite	DHCP DISCOVER Client ID Option
Software	FWOS-V2	Fault Time	2/24/06 7:47 AM	Variant	Invalid Path in the Client ID Option

Protocol - DHCP

Dynamic Host Configuration Protocol (DHCP) is an application-layer protocol used to dynamically assign IP addresses to network components. DHCP is platform-independent and can configure TCP/IP for multiple operating systems. Typically, a client is configured to run DHCP at startup, when it contacts the DHCP server to obtain an IP address.

DHCP can assign IP addresses from an predefined IP address range or predefined IP pool. A dynamic IP address is assigned for a limited time only; if the client does not renew the assignment lease periodically, the server reuses the address elsewhere. A manual IP address (static address) is assigned permanently (until manually changed). DHCP can also provide other TCP/IP parameters to a client, including subnet mask, gateway, and DNS/WINS settings.

DHCP communications occur over a UDP/IP connection between the server UDP/67 and client UDP/68. DCHP request and response messages use the same packet structure.

图 26.5　MU Security 公司的 MU-4000 生成的报告

26.1.6　Security Innovation 公司的 Holodeck[13]

Security Innovation 公司的 Holodeck 是一个独特的软件工具，该工具允许开发者模拟非预期的失效，研究 Windows 平台上开发的软件实现的安全性和错误处理器的稳定性。Holodeck 具有典型的模糊测试行为，例如对文件和网络数据产生变异的能力。Holodeck 的独特功能包括能够触发以下错误：

- 当尝试访问特定文件、注册表键和值和 COM 对象时的资源错误。
- 在受限的磁盘空间、内存和网络带宽情况下发生的系统错误。

Holodeck 的价格还算合理：每个用户许可为 1495 美金，并且提供了开放的 API，

[13] http://www.securityinnovation.com/holodeck/

允许高级用户围绕这个工具创建自动化工具及扩展。图 26.6 展示了 Holodeck 主界面的屏幕截图。

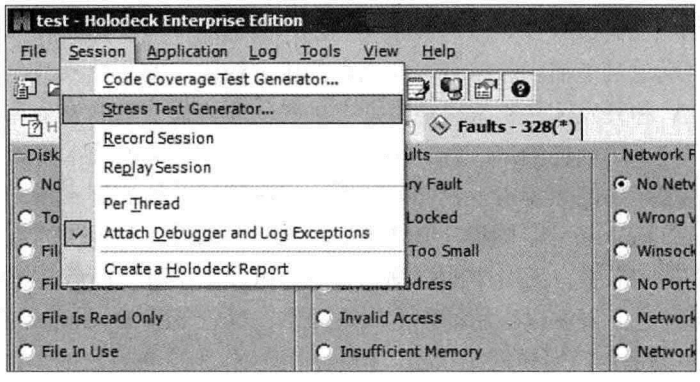


图 26.6　Security Innovation 的 Holodeck

上面给出的这些公司和产品的列表当然不是可用的商业模糊测试方案的完整列表。然而，这个列表表明了模糊测试领域的当前状况。在技术上，以上列出的大部分工具都还处于“初始版本”阶段。模糊测试技术领域虽然很重要，但仍然处于相对不成熟的阶段。

26.2　发现漏洞的混合方法

我们所讨论的商业模糊测试器大多数面向较窄范围的产品，只专注于某种特定的发现漏洞的方法。如果说我们已经从漏洞研究中得到了什么启发，那就是没有任何一种方法是银弹。所有的方法都有优点，也都有局限性。

了解了这一点，我们就期望安全性工具转向混合分析方式，能够结合应用多种方法。我们的目标是找到一个方案，使得部分之和大于整体。例如，考虑混合源代码分析技术与模糊测试技术。确保有足够的代码覆盖是模糊测试的挑战之一。在一个纯黑盒的测试场景中，如何才能确认已经对所有的输入进行了模糊测试？在第 23 章“模糊测试器跟踪”中，我们讨论了将代码覆盖作为度量模糊测试完整性的指标。下面我们来考虑一个替代的混合分析方法。首先，进行自动的源代码评审。这种传统的白盒测试方式通常能够发现一些“潜在”的漏洞，但可能会有误报，因为这种测试方法无法确认应用运行时最终的输出。利用静态扫描辅助生成模糊测试用例，而在动态模糊测试阶段证明或证伪这些潜在的漏洞，这样我们就可以解决白盒测试的局限性。我们期望模糊测试变成混合性安全保证技术的一个集成部分。

26.3 集成的测试平台

安全专家们将会持续发明、改进和采用新的，改进的，一次性的和单机版本的模糊测试器。然而，与满足功能需求和符合时间期限的压力相比，安全性测试会一直是第二位的需求，因此，QA 和开发团队不太可能有那么多时间研究最前沿的安全工具。要想在 QA 和开发社区中广泛采用模糊测试，就必须找到方法，将模糊测试技术集成到 QA 和开发人员熟悉的，每天都使用的平台中。我们期望看到模糊测试功能被集成到开发环境，如微软的 Visual Studio 和 Eclipse 中。同样，IBM 或 Mercury 等软件厂商提供的流行 QA 平台也可以从模糊测试功能中获益。工具中的模糊测试能力可以由工具厂商或其维护者提供，但开始时更可能由第三方开发并以应用插件的形式提供。尽管 SPI Dynamics 的 DevInspect 和 QAInpect 产品都不是专门的模糊测试器，但它们可以作为这种方法的早期例子。

26.4 小结

模糊测试这个行业将向何处发展？有些早期信号表明模糊测试技术应用正在蓬勃发展。我们欢迎这个趋势，由于多种原因，我们将模糊测试看作安全社区的一项独特技术。最重要的是，二进制逆向工程和深入的源代码分析等替代技术需要专门的技能，而对 QA 和开发者来说，具有这些技能是不现实的。另一方面，模糊测试可以被实现为自动化的过程，因此 QA 和开发团队都可以使用。

517

尽管独立的安全研究者会持续推动封装和开发有趣的模糊测试技术，但首先还是需要通过商业软件厂商的努力来创建易于使用的模糊测试环境，并能平滑地集成到开发环境中。在第 24 章我们讨论过，最核心的需求是先进的错误检测技术。现有的模糊测试器极少绑定使用调试技术。即使某些模糊测试器绑定了调试器，效果也远远谈不上最佳。必须应用更高级的技术，以准确指明并自动分类所发现的错误。错误检测厂商，例如我们在第 24 章中提到的包括 IBM、Compuware、Parasoft 和 OC System 等，都在各自不同的方向上来发展这方面的能力和达成这个目标。

回顾比模糊测试更成熟的技术的发展历史，我们可以预计到几个发展中的关键事件。首先，更多、更大的公司会进入模糊测试领域中，与先行者竞争。厂商可以从头开发他们自己的方案，但在许多情况下，他们会需要先行者的一些商业技术。有哪些公司可能会进入这个领域？这个问题很难确定地回答，但是在安全性和 QA 领域的软件厂商中应该会出现通过采用模糊测试技术实现与竞争对手差异化的厂商。

附录 A

本书引用的小布什语录之详细解读

小布什总统大约是美国历史上制造语录最多的总统。这位有着浓重德州口音（小布什有个绰号“Dubya”，就是嘲笑他会把 W 这个字母发音为“Dubya”的口音）且经常犯些低级语法错误的总统是当之无愧的语录之王，有人为他的各种搞笑语录编书，有人把他的各种语录变成了歌曲……，甚至，本书的作者之一 Pedram 都在本书中向小布什致谢，感谢他的故事激励作者本人树立了自己能成为作者的信心。

小布什在任期间，虽然没有耀眼的成就，在外交方面的成果也乏善可陈，但在国内经济方面，还是达成了一些不错的成果的。另外，对任内的 9/11 事件的处理，虽然在初期被许多美国人置疑，但纵观其后对威胁美国安全的恐怖势力的清剿，建立全球性的反恐联合战线，客观地说，还是有不少有见地的举措。本来，掌管美国这个全球第一大经济实体的工作就不好做，小布什既没有拿“国情困难”和“天灾人祸”为自己的失误搪塞，也没有成天光喊口号不干实事，在位期间基本还是很好地实现了“谋其政”。

其实，作为一个经常需要在各种场合下口若悬河的角色，常常犯些小错误本来也没什么大不了的，只怪小布什生在这个几乎无隐私的互联网时代，就任的是人人都可以骂的美国总统的角色，因此，也就只能让大家拿他的语录来娱乐了。

本附录给出了书中每章引用的小布什的语录的出现位置、原文、中文译文和译者本

人给出的解读。虽然译者已经尽力追本溯源，确认每句引语中的问题，但毕竟只是一家之言（在论坛上有时候也能看到美国人自己对这些引语的理解都不太一致），如有疏漏之处，还请各位读者斧正。

出现位置：原书序

原文："I Know the human being and fish can coexist peacefully."

——George W. Bush, Saginaw, Mich., Sept. 29, 2000

译文：我知道人类和鱼可以和平共存。

——乔治·布什，2000 年 9 月 29 日，于密歇根州萨吉诺

解读：这句话如果出自一位动物爱好者之口可能还不会让人觉得那么奇怪，但出自小布什总统之口就有点不伦不类了。大约小布什是想要突出人类和自然的和平相处吧（小布什总统从未对这句话进行过解释），真实情况如何就不得而知了。另外，“coexist peacefully”是一个在冷战时期常用的词，一般用来形容美苏两个超级大国的和平共处，用在“人类和鱼”的身上也的确让人感觉蛮怪异的。

出现位置：第 1 章

原文："Will the highways of the Internet become more few?"

——George W. Bush, Concord, N.H., January 29,2000

译文：“互联网高速公路会不会越来越少？”

——乔治·布什，2000 年 1 月 29 日，于新罕布什尔州康科德

解读：英文水平一向被美国人诟病的小布什总统这次犯了个中国的中学生都能发现的低级错误，“more few”显然应该是“fewer”。

出现位置：第 2 章

"They misunderestimated me."

——George W. Bush, Bentonville, Ark., November 6, 2000

译文：“他们错误地低估 (misunderestimate) 了我。”

——乔治·布什，2000 年 11 月 6 日，于阿肯色州本顿维尔

解读：这次，天才的小布什总统生造了一个词：misunderestimate。underestimate 的意思是“低估”，小布什大约是急于说明对他的低估是个严重的错误，因此干脆将“mis”这个表示否定的前缀加在 underestimate 之前，我猜测他要表达的意思是“错误地低估”。

出现位置：第 3 章

原文："Too many good docs are getting out of the business. Too many OB/GYNs aren't able to practice their love with women all across the country."

——George W. Bush, Poplar Bluff, Mo., September 6, 2004

译文：很多好的医生现在都失业了。很多妇产科医师不能向全国的妇女传递他们的爱心。

——乔治·布什，2004 年 9 月 6 日，于密苏里州波普勒布拉夫

解读：这句话没有什么语法上的问题。不过用词很暧昧。“practice their love with women”这句话可以直译为“与妇女们实践爱”，这个实在是很让人想入非非啊……

出现位置：第 4 章

原文："My job is to, like, think beyond the immediate."

——George W. Bush, Washington, DC, April 21, 2004

译文：我的工作是，诸如，考虑超出当前的事情。

——乔治·布什，2004 年 4 月 21 日，于华盛顿特区

解读：小布什总统说他的工作是“考虑超出当前的事情”，不过看起来这个例子举得不怎么好。事实上，小布什总统给人的印象一直是没有前瞻性，缺乏章法，总是被动地处理到来的问题。所以，小布什总统的这个举例（like）看起来完全是对他自己的讽刺。

出现位置：第 5 章

原文："You teach a child to read, and he or her will be able to pass a literacy test."

——George W. Bush, Townsend, TN, February 21, 2001

译文：你教孩子阅读，他们才能通过识字测试。

——乔治·布什，2001 年 2 月 21 日，于田纳西州汤森

解读：这又是一个可以作为中国的中学生英语考题的例子。能看出来这句话的语法问题在哪里吗？对了，“he or her”看上去怎么那么别扭呢？作为主语，这里应该是“he or she”。

出现位置：第 6 章

原文："Our enemies are innovative and resourceful, and so we are. They never stop thinking about new ways to harm our country and our people, and neither do we."

——George W. Bush, Washington, DC, August 5, 2004

译文：我们的敌人富有创新精神且足智多谋，我们也一样。敌人总在不停地想找新办法来伤害我们的国家和人民，我们也一样。

——乔治·布什，2004 年 8 月，于华盛顿特区

解读：和上一个引语相比，这里的错误不那么明显。“neither do we”的意思是“我们也一样”，而不是“我们不同”。小布什总统大约是想说“我们和他们不一样”，可惜他的英文水平实在是令人堪忧，直接用了一个“neither do we”。

出现位置：第 7 章

原文："This foreign policy stuff is a little frustrating."

——George W. Bush, as quoted by the New York Daily News, April 23, 2002

译文：这个外交政策的内容有些让人沮丧。

——乔治·布什，纽约日报新闻 2002 年 4 月 23 日引用

解读：“foreign policy stuff”的用法比较奇怪。当然，这句话之所以被广泛引用的重要原因还在于小布什的外交政策一直被人诟病。由他说出这句话来明显是个讽刺。

出现位置：第 8 章

原文："Those weapons of mass destruction have got to be somewhere!"

——George W. Bush, Washington, DC, March 24, 2004

译文：这些大规模的杀伤性武器一定在某个地方！

——乔治·布什，2004 年 3 月 24 日，于华盛顿特区

解读：细心的读者恐怕很快就能找出这句话中的错误之处。“have got to be”应该是“have gotten to be”。不过这句话被广泛引用的另一个原因在于小布什先生当年信誓旦旦地说伊拉克有大规模杀伤性武器，最后却被证实是摆了个大乌龙。

出现位置：第 9 章

原文："I am the master of low expectation."

——George W. Bush, aboard Air Force One, June 4, 2003

译文：我是低期望的主人。

——乔治·布什，于空军一号，2003 年 6 月 4 日

解读：这句话大约是小布什总统的自嘲吧，不过还真挺符合他在大家心目中的印象的。总之，不要对小布什期望过高。

出现位置：第 10 章

原文："The most important thing is for us to find Osama bin Laden. It is our number one priority and we will not rest until we find him."

——George W. Bush, Washington, DC, September 13, 2001

"I don't know where bin Laden is. I have no idea and really don't care. It's not that important. It's not our priority."

——George W. Bush, Washington, DC, March 13, 2002

译文：对我们来说，当前最重要的事是找到奥萨马·本·拉登，这是我们头等优先的任务，我们不找到他绝不罢休。

——乔治·布什 2001 年 9 月 13 日，于华盛顿

我不知道本·拉登在哪。我真的不知道，也不关心。这没有那么重要，不是我们优先要考虑的。

——乔治·布什 2002 年 3 月 13 日，于华盛顿

解读：前言不搭后语也是小布什总统的特色。2001 年 9 月 13 日刚说完“找到本·拉登是我们的头等大事”，4 个月后立刻就表明“找到本·拉登不是我们优先要考虑的”。变化之快让人有啼笑皆非的感觉。其实，这类错误小布什总统经常会犯，例如，2002

在东京小布什说“半个多世纪以来，美日两国筑成了现代历史上最伟大、最持久的同盟关系之一”，全然忘了 50 多年前美国在日本投下的那两颗原子弹。

出现位置：第 11 章

原文："If this were a dictatorship, it'd be a heck of a lot easier, just so long as I'm the dictator."

——George W. Bush, Washington, DC, December 19, 2000

译文：如果这是个独裁政权，如果我是那个独裁者，那就太容易了。

——乔治·布什，2000 年 12 月 19 日，于华盛顿特区

解读：小布什总统还有个分不清场合的毛病。例如，“a heck of”这种俚语的表达方式就不太适合出现在讨论这种严肃问题的场合中。这方面小布什总统没少犯错。例如，2008 年在华盛顿，与罗马教皇本尼迪克特十六对话时，小布什恭维教皇说“你的演讲棒极了（Awesome speech）”，用 awesome 这种俚语恭维教皇，明显有失尊敬。

出现位置：第 12 章

原文："I am the commander--see, I don't need to explain--I do not need to explain why I say things. That's the interesting thing about being president."

——George W. Bush, as quoted in Bob Woodward's Bush at War

译文：我是指挥官——看，我不需要解释——我不需要解释为什么要说这些。这是当总统的有趣之处。

——乔治·布什，在 Bob Woodward 的《战争中的布什》中被引用

解读：小布什总统对“总统”职位的理解总是与众不同。“不需要解释”成了“当总统的有趣之处”，真不知让小布什先生在美国总统的位置上呆上两个任期是幸运还是不幸。

出现位置：第 13 章

原文："It's in our country's interests to find those who would do harm to us and get them out of harm's way."

——George W. Bush, Washington, DC, April 28, 2005

译文：找出那些想要伤害我们的人，让他们无法伤害我们——这是我们的国家利益所在。

——乔治·布什，2005 年 4 月 28 日，于华盛顿特区

解读：小布什总统的这段话超过了法律的范畴。如何界定“想要伤害我们的人”？法律只能对已有的行为做出惩罚，但却不应该把有“伤害我们的想法”的人绳之以法。小布什政府在应对 9/11 恐怖袭击的过程中扩大了国家安全部门的权力，允许美国政府在其他国家绑架恐怖嫌疑人，实际上开了危险的先例。

出现位置：第 14 章

原文："I own a timber company? That's news to me. Need some wood?"

——George W. Bush, second presidential debate, St. Louis, MO, October 8, 2004

译文：我有一个锯木厂？我才知道。要不要来点木头？

——乔治·布什，第二次总统选举辩论，2004 年 10 月 8 日，于密苏里州圣路易斯

解读：小布什在 2004 年的第二次总统选举辩论中矢口否认自己拥有一个锯木厂，然而，小布什在 2001 年的退税表中列出了 84 美金的退税，标明这笔收入来自他拥有的一个锯木厂。

出现位置：第 15 章

原文："I think we agree, the past is over."

——George W. Bush, on his meeting with John MacCain, Dallas Morning News, May 10, 2000

译文：我想我们同意，过去已经结束了。

——乔治·布什，与约翰·麦凯恩会谈，2000 年 5 月 10 日，达拉斯早报

解读：“the past is over”是很奇怪的说法。要表示“过去的就过去了”，常用的说法是“the past is gone”。这是小布什总统的英语水平不过关的又一个例子。

出现位置：第 16 章

原文："I couldn't imagine somebody like Osama bin Laden understanding the joy of Hanukkah."

——George W. Bush, White House Menorah lighting ceremony, Washington, DC, December 10, 2001

译文：我不能想象像本·拉登这样的家伙能懂得光明节的乐趣。

——乔治·布什，2001 年 12 月 10 日，白宫灯台亮灯仪式上，于华盛顿特区

解读：Hanukkah 是犹太光明节，是犹太人的节日。而本·拉登作为一个极端伊斯兰教徒，自然不可能会去过光明节，也不会懂得光明节的乐趣。单看这句话，不免让人觉得小布什总统够无知的。不过，其实小布什在这句话上是有点被“冤枉”的，这句话是在白宫光明节的亮灯仪式上说的，当时记者问他：“在这个安宁与庆祝的时刻，你能告诉我们你如何看待本·拉登的录像带吗？（Sir, on this occasion of peace and celebration, can you tell us how you were struck by this bin Laden videotape?）”小布什的回答是：“这只能提醒我本·拉登是一个怎样的刽子手，我们要抓他的理由足够正确。我不能想象本·拉登这样的家伙能够懂得光明节的乐趣，懂得圣诞节的乐趣，或是会为和平与希望进行庆祝（It just reminded me of what a murderer he is and how right and just our cause is. I couldn't imagine somebody like Osama bin Laden understanding the joy of Hanukkah, or the joy of Christmas, or celebrating peace and hope.）。”

在我看来，如果不被断章取义的话，这次小布什的话其实相当正常。

出现位置：第 17 章

原文："One of the common denominators I have found is that expectations rise above that which is expected."

——George W. Bush, Los Angeles, September 27, 2000

译文：我发现一件共性的事情是超越期望的期望。

——乔治·布什，2000 年 9 月 27 日，于洛杉矶

解读：又是“小布什式”的英语。“expectations rise above which is expected”——你能理解他到底想说什么吗？

出现位置：第 18 章

原文："Natural gas is hemispheric. I like to call it hemispheric in nature because it is a product that we can find in our neighborhoods."

——George W. Bush, Washington, DC, December 20, 2000

译文：天然气是半球性（hemispheric）的，我喜欢把它称作半球性的，因为我们可以在我们的邻国找到它们。

——乔治·布什，2000 年 12 月 20 日，于华盛顿

解读：小布什总统的能源政策也是经常让人看不懂。这番话是小布什总统在和墨西哥达成能源协议的时候解释自己和墨西哥签署能源协议的原因时说的。墨西哥有丰富的油气存储，但由于缺乏资金和技术无法开采。小布什政府希望通过投资的方式帮助墨西哥开采石油和天然气并供应美国，但看起来这只是他的一厢情愿而已。

出现位置：第 19 章

原文："It's white."

——George W. Bush, after being asked by a child in Britain what the White House was like, July 19, 2001

译文：它是白色的。

——乔治·布什，2001 年 7 月 19 日，在一个英国小孩问到白宫是什么样子的时候回答说

解读：这个大约不需要译者来解读了。小布什总统对一个从未见过白宫的小朋友解释白宫的时候，居然只能用“它是白色的”这样的话语，实在是让人大跌眼镜。

出现位置：第 20 章

原文："I hear there's rumors on the Internet that we're going to have a draft."

——George W. Bush, second presidential debate, St. Louis, MO, October 8, 2004

译文：我听说互联网上谣传我们将会有一个草案。

——乔治·布什，第二次总统选举辩论，2004 年 10 月 8 日，于密苏里州圣路易斯

解读：显然，“there's rumors”是有问题的。应该是“there are rumors”。

出现位置：第 21 章

原文："There's an old saying in Tennessee--I know it's in Texas, probably in Tennessee--that says, fool me once, shame on--shame on you. Fool me--you can't get fooled again."

——George W. Bush, Nashville, TN, September 17, 2002

译文：田纳西有一句古老的格言——我知道是得克萨斯的格言，呃，也许是田纳西的——格言说，愚弄我一次，你——你应该感到羞愧，愚弄我——呃，你不会被多次愚弄。

——乔治·布什，2002 年 9 月 17 日，于田纳西州纳什维尔

解读：这段话经典地体现了什么叫前言不搭后语。实际上，这句格言应该是“Fool me once, shame on you; fool me twice, shame on me”。意思是“骗我一次是你狠，骗我两次是我蠢”。可是我们的小布什总统完全弄不清“你”和“我”。可怜的小布什总统，估计是被人骗的次数多了，已经不知道 shame on you 还是 shame on me 了。

出现位置：第 22 章

原文："I know how hard it is for you to put food on your family."

——George W. Bush, Greater Nashua, NH, January 27, 2000

译文：我知道对你来说，让全家吃饱有多难。

——乔治.W.布什，2000 年 1 月 27 日，于新罕布什尔州大纳施华区

解读：有句英语习语“put food on the family table”，直译为“把食物放在家庭的餐桌上”，意思是指为全家提供吃的（养活全家），小布什总统大概是想用这句习语，可是他漏掉了“table”这个词，就闹出笑话来了。

出现位置：第 23 章

原文："Well, I think if you say you're going to do something and don't do it, that's trustworthiness."

——George W. Bush, in a CNN online chat, August 30, 2000

译文：嗯，我认为如果你说了你要做某事而没有去做，那就值得信任。

——乔治·布什，2000 年 8 月 30 日，美国有线新闻的在线访谈

解读：小布什总统在这里犯的错误是错误地使用了“trustworthiness”这个词，小布什想要表达的是这种行为是不可信的，但这句话的意思却变成了这种行为是“值得信任的”。

出现位置：第 24 章

原文："Never again in the halls of Washington, DC, do I want to have to make explanations that I can't explain."

——George W. Bush, Portland, OR, October 31, 2000

译文：我再也不想在华盛顿的这个大厅里解释那些我没法解释的事情了。

——乔治.布什，2000 年 10 月 31 日，于俄勒冈州波特兰

解读：这里也有一个初级的语法错误。正确的说法应该是“make explanations about that I can't explain”。

出现位置：第 25 章

原文："Rarely is the question asked: Is our children learning?"

——George W. Bush, Florence, SG, January 11, 2000

译文：我们很少问这个问题：我们的孩子正在学习吗？

——乔治·布什，2000 年 1 月 11 日，于南卡罗来那州佛罗伦萨

解读：“children”是复数形式的名词，因此这里不应该说“Is our children……”，而应该用“Are our children……”，又是一个可以上我国中小学英语考试的题目。

出现位置：第 26 章

原文："But all in all, it's been a fabulous year for Laura and me."

——George W. Bush, summing up his first year in office, three months after the 9/11 attacks, Washington, DC, December 20, 2001

译文：总而言之，今年对劳拉和我来说是出色的一年。

——乔治·布什，一周年庆典，9/11 攻击后的 3 个月，华盛顿特区，2001 年 12 月 20 日

解读：小布什总统在 9/11 攻击中的表现受到了许多美国人的质疑，甚至还有一部以此为题材的影片《9/11 中的小布什》，极尽挖苦之能事。在这种情况下，小布什总统憾然在年末的一周年庆典上总结说这一年是出色（fabulous）的一年，真不知道他的勇气从何而来。

索　引

A

abort signals　中止信号，100
Accept header,Accept　头，122
Accept-Encoding header,Accept-Encoding　头，122
Accept-Language header,Accept-Language　头，122
access　访问
  control　控制，29
  violation handler　违例处理例程，340-342
ActiveX controls,ActiveX　控件，284
  Adobe Acrobat PDF control, Adobe Acrobat PDF　控件，294-296
  fuzzer development　模糊测试器开发，287-289
    heuristics　启发式，298
    loadable controls,enumerating　可加载控件，枚举，289-293
    monitoring　监视，299
    properties,methods,parameters,and types　属性，方法，参数，类型，293-297
    test cases　测试用例，298
  history　历史，25
  overvoew　概述，285-287
  vulnerabilities　安全漏洞，273-275
  WinZip FileView，298
adbg，370
address bar spoofing vulnerabilities　地址栏欺骗漏洞，281
address　地址
  reading　读取，476-478
  writing to　写入，478-479
Adobe
  Acrobat
    PDF control,PDF　控件，294-296
    shell script workaround,shell　脚本解决方案，192
  Macromedia Shockwave Flash file format.*See*

SWF agents Macromedia Shockwave Flash 文件格式，见 SWF 代理
fault detection 错误检测，233
network monitor 网络监视器，402
process monitor 进程监视器，403
Sulley sessions Sully 会话，402-404
VMWare control VMWare 控件，403
AIM(AOL Instant Messenger)protocol AIM（AOL 即时通信软件）协议，49-52
logon credentials sent 登录身份发送，52
server reply 服务器回应，51
username 用户名，51
Aitel，Dave，23
algorithms 算法
GAs 遗传算法，431-435
CFG with connecting path 带连接路径的控制流图，433
CFG with exit nodes 带退出节点的控制流图，434
CFG with potential vulnerability 含潜在漏洞的控制流图，433
fitness function 适应度函数，432
reproduction function 繁殖函数，431
Sidewinder，433-434
as stochastic global optimizers 带随机性的全局优化器，432
Needleman-Wunsch，428
Smith-Waterman local sequence alignment Smith-Waterman 本地序列对齐算法，428
Averages 无加权成对算术平均数，429
alignment of sequences 序列对齐，427
amino acid sequence alignment 氨基酸序列对齐，427
antiparser 反解析器，354-356
AOL Instant Messenger.*See* AIM protocol AOL 即时通信工具，参见 AIM 协议
APIs
antiparser 反解析器，354-356
CreateProcess，321
debugging APIs 调试 API，179
Windows debugging Windows 调试，320-323，332-333
apKeywords()data type apKeywords() 数据类型，355
Apple Macbook vulnerability 苹果 Macbook 安全漏洞，228
application layer vulnerabilies 应用层漏洞，230
application logs,fault detection 应用日志，错误检测，233
applications 应用程序
manually testing 手工测试，10
service provider 服务提供者（ASP），114
setuid，37
sweeping 清扫，10
web application fuzzers web 应用模糊测试器，41
buffer overflow vulnerability 缓冲区溢出漏洞，130
configuring 配置，118-119
exception,detecting 异常，检测，135-136
heap overflow vulnerability 堆溢出漏洞，129
inputs.*See* web application fuzzers inputs, 输入。参见 web 应用模糊测试器，【缺少页数信息】
overview 概述，113-115

targets 目标，117-118
technologies 技术，115
vulnerabilities 漏洞，132-135
archiving utility vulnerabilities 归档工具漏洞，170
argv module argv 模块，104-106
ASCII text files ASCII 文本文件，202
ASP(application service provider) ASP（应用福服务提供者），114
ASP.NET，116
asynchronous sockets(WebFuzz) 异步套接字（WebFuzz）150-153
attack heuristics 启发式攻击，353
audits,saving 审计，保存，204-205
authentication(weak) 认证（弱认证），133
Autodafe[as]，369-371
automation 自动化
benefits 益处，73
human computation power 人工计算威力，73
reproducibility 可再现性，74
binary testing 二进制测试，17-18
debugger monitoring 调试监视，481
advanced example 高级示例，486-489
architecture 架构，481
basic example 基础示例，482-484
PaiMei crash binning utility PaiMei 崩溃日志二进制分析工具，487-489
环境变量模糊测试
GDB method GDB 方法，96-97
getenv function getenv 函数，98
library preloading 库预加载，98-99
协议分析
bioinformatics 生物信息学，427-431
genetic algorithms 遗传算法，431-435
heuristic-based 基于启发的方法，421-427
source code auditing tools 源代码审计工具，7
length calculation 长度计算，352
protocol generation testing 协议生成测试，36
white box testing 白盒测试，6-8
可用性
availability 黑盒测试，13
gray box testing 灰盒测试，18
white box testing 白盒测试，9
av_handler()routine av_handler()例程，483
AWStats Remote Command Execution Vulnerability AWStats 远程命令执行漏洞
website 网站，118
AxMan，25，275

B

backdoor fuzzing limitations 后门模糊测试限制，30
basic blocks 基础块
defined 定义，440
start/stop points 起始点/停止点，440
tracking 跟踪，444
Beddoe，Marshall，428
BeginRead()method BeginRead() 方法，152
BeginWrite()method BeginWrite() 方法，151
beSTROM，138，508
binary auditing 二进制审计
automated 自动审计，17-18
manual 手工审计，14-16
binary files(FileFuzz) 二进制文件（FileFuzz），201

binary protocols 二进制协议，49-52
binary visualization 二进制可视化，439-441
BinAudit，18
BindAdapter()function BindAdapter() 函数，259
bioinformatics 生物信息学，427-431
bits 位，93，385
bit_field class 位字段类，377
black box testing 黑盒测试，9
  fuzzing 模糊测试，12
  manual 手工测试，10
  pros/cons 优点/缺点，13-14
  tool(beSTORM) 工具（beSTORM），138，508
Blaster worm Blaster 蠕虫，225
blocks 块
  basic 基础块
    defined 定义，440
    start/stop points 起始点/结束点，440
    tracking 跟踪，444
  helpers 辅助块，395
    checksums 校验块，396
    example 示例，397
    repeaters 重复器，396
    sizers 尺寸，395-396
  identifiers 标识符，58
  protocol 协议
    modeling 建模，242
    representation 表示，361-362
    sizes 尺寸，395-396
  Sulley framework Sulley 框架，392
    dependencies 依赖，394-395
    encoders 编码器，394
    grouping 分组，392-393
    helpers 辅助器，395-397
BLOSUM(Blocks Substitution Matrix) 块置换矩阵（BLOSUM），429
boundary value analysis(BNA) 边界值分析（BVA），21
BoundsChecker，493
bp_set() routine bp_set() 例程，333
break command break 命令，97
BreakingPoints，509
breakpoints 断点
  handler 断点处理器，342
  hardware 硬件断点，324
  software 软件断点，324
  WS2_32.dll recv() WS2_32.dll recv() 函数，336
Brightstor backup software vulnerabilities Brightstor 备份软件漏洞，424
browser fuzzers 浏览器模糊测试器，41-42
  ActiveX controls ActiveX 控件，287-289
    heuristics 启发式，298
    loadable controls,enumerating 可加载控件，枚举，289-293
    monitoring 监视，299
    properties,methods,parameters,and types 属性，方法，参数和类型，293-297
    test cases 测试用例，298
  approaches 测试方法，269-271
  fault detection 错误检测，282
  heap overflows 堆溢出，277-279
  history 历史，24
  inputs 输入，271
    ActiveX controls ActiveX 控件，273-275

client-side scripting 客户端脚本，276
CSS，275-276
Flash，279
HTML headers HTML 头，271
HTML tags HTML 标签，271-273
URLs，280
XML tags XML 标签，273
methods 方法，269，275-276，279-280
Month of Browser Bugs 浏览器缺陷月，268
overview 概述，268
targets 目标，269
vulnerabilities 漏洞，280-282
brute force fuzzing 强制模糊测试
file formats 文件格式模糊测试，172-173
network protocols 网络协议模糊测试，231
brute force testing 强制测试，36
btnRequest_Click() method btnRequest_Click() 方法，153
buffer overflows 缓冲区溢出
case study(Web applications) 案例研究（Web 应用），158-160
vulnerabilities 漏洞，130
web applications Web 应用，133
web browsers Web 浏览器，281
BugScam，17
BVA(boundary value analysis) 边界值分析（BVA），21
byref() function byref() 函数，318

C

call graphs 调用图，439
callbacks 回调
access violation handler 访问违例处理器，340-342
breakpoint handler 断点处理器，342
registering 注册回调，401-402
Cascading Style Sheet.*See* CSS 层叠样式表。参见 CSS
cc_hits database cc_hits 数据库，455-457
CCR(Command Continuation Request)example 命令继续请求（CCR）示例，78-79
cc_tags database cc_tags 数据库，455
CFGs(control-flow graphs) 控制流图（CFG），433，440-441
disassembled dead listing 反汇编的不可达代码清单，441
GAs 遗传算法
connecting path 连接路径，433
exit nodes 退出节点，434
potential vulnerability 潜在漏洞，433
CGI(Common Gateway Interface) 公共网关接口（CGI），115
character translations 字符翻译，85
character-delimited field 以字符分隔的字段，47
checksums 校验和
as block helpers 块辅助器形式的校验和，396
protocols 协议，58
child processes,forking offand tracing 子进程，进程复制与跟踪，185-187
choosing 选择
fuzz values 模糊值，480-481
hook points 钩子点，331

memory mutation locations 内存变异位置，331
packet capture libraries 数据包抓取库，256-257
protocol fields 协议字段，47
web application inputs web 应用输入，119-121
cookies，129
headers 头，128
method 方法，123-125
post data post 数据，130
protocol 协议，128
request-URI 请求 URI，126-128
CISC (Complex Instruction Set Computer) 复杂指令集计算机（CISC）
architecture 架构，438
class IDs (CLSIDs) 类 ID（CLSIDs），285
classes 类
apKeywords()，355
bit_field 位字段，377
PyDbg，332-334，340-345
TcpClient，148-149
clfuzz，38
clients 客户端
Ethreal，335
launching 发布，334
client-side vulnerabilities 客户端漏洞，223，276，280-282
CLSIDs (class IDs) 类 ID（CLSIDs），285
code 代码
coverage. See tracking 代码覆盖率，参见“跟踪”
reusability 可重用性
frameworks 框架，354
Peach，366
Code Red worm Code Red 蠕虫，225
Codenomicon，41，510-511
foundation 基础，23
HTTP test tools HTTP 测试工具，41，138
CodeSpy website CodeSpy 网站，7
coding phase (SDLC) 编码阶段（SDLC），501-502
COM (Component Object Model) 组件对象模型（COM），284
ActiveX controls ActiveX 控件
Adobe Acrobat PDF control Adobe Adobe PDF 控件，294-296
fuzzer development. See ActiveX controls fuzzer development 模糊测试器开发，参见“ActiveX 控件”部分的“模糊测试器开发”
overview 概述，285-287
Winzip FileView，298
history 历史，284
interfaces 接口，284
objects 对象，284
Raider，25，42，275
VARIANT data structure VARIANT 数据结构，293
Command Continuation Request (CCR) example 命令继续请求（CCR）示例，78-79
command-line arguments 命令行参数，89
command-line fuzzers 命令行模糊测试器，37-28
commands 命令
break break 命令，97
CREA CREA 命令，248

commands commands 命令，97
execution vulnerabilities 执行漏洞，281
find find 命令，93
injection vulnerabilities 注入漏洞，87
commands command commands 命令，97
commercial tools 商业工具，507
beSTORM，138，508
BreakingPoint，509
Codenomicon，41，510-511
foundation 基础，23
HTTP test tools HTTP 测试工具，41，138
Holodeck，514-515
Mu-4000，512
ProtoVer Professional ProtoVer 专业版，512
Common Gateway Interface (CGI) 公共网关接口（CGI），115
community involvement (frameworks) 相关社区（框架），43
compile time checkers 编译时间检查器，6
Complex Instruction Set Computer (CISC) 复杂指令集计算机（CISC）
architecture 架构，438
complex protocols 复杂协议，40
complexity 复杂性
frameworks 框架，44
gray box testing 灰盒测试，18
in-memory fuzzing 内存模糊测试，43
white box testing 白盒测试，9
Component Object Model. *See* COM 组件对象模型，参见“COM”
Computer Associates'Brightstor backup software
vulnerabilities,漏洞，424
Compuware Devpartner BoundsChecker，493
configuring web applications 配置 web 应用，118-119
CONNECT method CONNECT 方法，125
Connection header Connection 头，123
connections 连接
dropped 中断连接，135
testing 测试连接，472
ContineuDebugEvent() routine ContinueDebugEvent() 例程，323
control-flow graphs. *See* CFGs 控制流图，参见“CFG”
convert 转换，367
Cookie header cookie 头，123
cookies，129
coverage 覆盖
black box testing 黑盒测试，14
gray box testing 灰盒测试，18
white box testing 白盒测试，9
crash binning tool 崩溃数据分析工具，487-489
crash locations,viewing 崩溃位置，查看，405-406
crashbin_explorer.py tool crashbin_explorer.py 工具，405
CRC (Cyclic Redundancy Check) values 循环冗余校验（CRC）值，353
CREA command CREA 命令，248
CreateFile() method CreateFile() 函数，299
CreateProcess() function CreateProcess() 函数，203，10，203，299，321

create_string_buffer() function create_string_buffer() 函数，318
cross-domain restriction bypass vulnerabilities 能够旁路跨域限制的漏洞，281
cross-site scripting (XSS) 跨站脚本（XSS），132，163-166
CSRF (cross site request forgery) 跨站请求伪造（CSRF），134
CSS (Cascading Style Sheet) 层叠样式表（CSS）
  CSSDIE fuzzer CSSDIE 模糊测试器，24，42，275-276
    vulnerabilities 漏洞，275-276
  CSSDIE，24，275，42，276
ctypes module ctypes 模块，318
Cyclic Redundancy Check (CRC) values 循环冗余校验（CRC）值，353

D

data 数据
  capture 抓取
    networks 网络，251
    ProtoFuzz design ProtoFuzz 设计，258-259
    PStalker，454
  exploration 探索，453
  generating 生成
    CCR example CCR 示例，78-79
    character translations 字符翻译，85
    command injection 命令注入，87
    directory traversal 字典遍历，86
    field delimiters 字段分隔符，83-84
    format strings 格式字符串，85
    generators 生成器，364
    integer values 整数值，79-81
    pseudo-random data 伪随机数，353
    string repetitions 字符串重复，82
    SWF fuzzing test case SWF 模糊测试用例，384
  link layer vulnerabilities 链路层漏洞，228
  packets,assembling 数据包，组包，250-251
  parsing 解析
    networks 网络，251-252
    ProtoFuzz design ProtoFuzz 设计，259-261
  sources 源
    DFuz，358
    PStalker，452-453
  storage 存储，454-457
  types 类型
    apKeywords()，355
    block helpers 辅助块，395-397
    blocks 块，392-395
    character translations 字符翻译，85
    command injection 命令注入，87
    delimiters 分隔符，391
    directory traversal 目录遍历，86
    field delimiters 字段分隔符，83-84
    format strings 格式字符串，85
    integers 整数，79-81，390-391
    legos 积木（lego），398-399
    passing 传递（passing），318
    smart data sets 智能数据集，81
    strings 字符串，82，391
    Sulley framework Sulley 框架，

388-390
数据库
cc_hits，455-457
cc_tags，455
DataRescue Interactive Disassembler Pro（IDA Pro），439
DBI (Dynamic Binary Instrumentation) 动态二进制插装（DBI），68，491-493
DynamoRIO，491
error detection 错误检测，68
fault detection 故障检测，492
Pin，492
tool development 工具开发，492
DCOM (Distributed COM) 分布式 COM（DCOM），284
DDE (Dynamic Data Exchange) 动态数据交换（DDE），284
dead listing 反汇编清单（Dead listing），441
debug event loops 调试事件循环，322-323
调试异常事件，323
DebugActiveProcess() 例程，321
debuggers 调试器，16
adbg，370
automated monitoring 自动化监视，481
advanced example 高级示例，486-489
architecture 架构，481
basic example 基础示例，482-484
PaiMei crash binning utility PaiMei 崩溃二进制分析工具，487-489
DBI，491-493
DynamoRIO，491
fault detection 故障检测，492
Pin，492
fault detection 故障检测，233
GDB，16
OllyDbg，16，335
before/after demangle 解修饰前/后,336
conceptual diagram 概念图，339
parse routine 解析例程，338
restore point 恢复点，338
WS2_32.dll recv() breakpoint WS2_32. dll recv() 断点，336
process information 进程信息，178
web application fuzzing web 应用模糊测试，136
Web browser errors web 浏览器错误，282
WinDbg，16
debugging API (Windows) 调试 API（Windows），320-323
process information 进程信息，179
process instrumentation with PyDbg 用 PyDbg 进行进程插装，332-333
DebugSetProcessKillOnExit() routine DebugSet ProcessKillOnExit() 例程，321
DEBUG_EVENT structure DEBUG_EVENT 结构，323
decompilers 反编译器，16
DELETE method DELETE 方法，124
delimiters (Sulley framework) 分隔符（Sulley 框架），391
DeMott，Jared，366
denial-of-service. *See* DoS attacks 拒绝服务，参见“DoS 攻击”
dependencies (blocks) 依赖（块），394-395
depth (process) 深度（进程），64-66

design 设计
FileFuzz
applications,launching 应用，启动，211-213
exception detection 异常检测，213-217
files,creating 文件，创建，210
source files,reading 源文件，读取，210-211
writing to files 写入文件，211
logic 逻辑，30
ProtoFuzz，257，262
data capture 数据抓取，258-259
fuzz variables 模糊变量，261
hexadecimal encoding/decoding 16进制编码/解码，262
network adapters 网络适配器，257-258
parsing data 解析数据，259-261
SDLC，500-501
WebFuzz
asynchronous sockets 异步套接字，150-153
requests,generating 请求，生成，153-155
responses,receiving 响应，接收，155-156
TcpClient class TcpClient 类，148-149
detecting 检测
errors 错误，67-69
DBI platforms DBI 平台，68
debug clients 调试客户端，67
ping checks ping 检查，67
exceptions 异常
detection engines 检测引擎，183
FileFuzz，203-204，213-217
faults 故障
crash locations,viewing 崩溃位置，查看，405-406
DBI，492
execution control,transferring 执行控制，转移，475-476
first-chance versus last-chance exceptions 首轮异常与末轮异常，489-491
format string vulnerability 格式字符串漏洞，477
frameworks 框架，353
graphical representation 图形化表示，406
in-memory fuzzing 内存模糊测试，311-312
network protocol fuzzing 网络协议模糊测试，232-233，254
page faults 页故障，474
primitive 原始类型，472-474
reading addresses 读取地址，476-478
software memory corruption vulnerability 软件内存破坏漏洞
categories 类别，475
stack layout example 栈布局示例，477
Web browsers Web 浏览器，282
writing to addresses 写入地址，478-479
file format fuzzing 文件格式模糊测试，178-179

local fuzzing problems 本地模糊测试问题，99-101
web application exceptions web 应用异常，135-136
developers 开发者，503-504
development 开发
ActiveX fuzzer ActiveX 模糊测试器，287-289
heuristics 启发式，298
loadable controls,enumerating 可加载控件，枚举，289-293
monitoring 监视，299
properties,methods,parameters,and types 属性，方法，参数，类型，293-297
test cases 测试用例，298
DBI tools DBI 工具，492
environment integration offuzzing 模糊测试的环境集成，516
file format fuzzing 文件格式模糊测试
core fuzzing engine 核心模糊测试引擎，184-185
exception detection engine 异常检测引擎，183
exception reporting 异常报告，183-184
forking off/tracing child processes 创建/跟踪子进程，185-187
interesting UNIX signals 我们感兴趣的 UNIX 信号，187
not interesting UNIX signals 我们不感兴趣的 UNIX 信号，188
FileFuzz，209
applications,launching 应用，启动，211-213
approach 方法，209
design 设计，210-217
exception detection 异常检测，213-217
files,creating 文件，创建，210
language,choosing 编程语言，选择，210
source files,reading 源代码，读取，210-211
writing to files 向文件写入，211
iFuzz，105-107
ProtoFuzz
design 设计，258-262
goals 目标，255
packet capture library, choosing 包抓取库，选择，256-257
programming language 编程语言，256
software development lifecycles. *See* SDLC 软件开发生命周期。参见 SDLC
tracking tool 跟踪工具，442-443
basic blocks,tracking 基础块，跟踪，444
cross-referencing 交叉应用，447
instruction execution,tracing 指令执行，跟踪，444-445
recordings,filtering 录制，过滤，446
target profiling 目标描绘，443-444
web application fuzzers web 应用模糊测试器
approach 方法，148
asynchronous sockets 异步套接字，150-153
programming language,choosing 编程语言，选择，148

requests,generating 请求，生成，153-155
responses,receiving 响应，接收，155-156
TcpClient class TcpClient 类，148-149
DevInspect tool DevInspect 工具，504
Dfuz，357-360
data sources 数据源，358
functions 函数，357-258
lists,declaring 列表，声明，358
protocols,emulating 协议，仿真，359
rule files 规则文件，359-360
variables,defining 变量，定义，357
DHTML (Dynamic HTML) DHTML（动态HTML），24
directory traversal 目录遍历
case study 案例研究，156-157
Ipswitch Imail Web Calendaring，157
Trend Micro Control Manager，156
vulnerabilities 漏洞，86，132
disassembled binaries,visualizing 反汇编的二进制代码，可视化，439-441
disassemblers 反汇编器，15
disassembly-level heuristics 反汇编层次的启发式，426-427
discussion board vulnerabilities 讨论板漏洞，117
Distributed COM (DCOM) 分布式 COM（DCOM），284
documentation 文档，62
DOM (Document Object Model) DOM（文档结构模型），285
DOM-Hanio，42
DoS (denial-of-service) attacks DoS（拒绝服务）攻击
file formats 文件格式，175
web applications web 应用，132
web browsers web 浏览器，280
DownloadFile() method DownloadFile() 方法，299
drivers 驱动，255
Dynamic Binary Instrumentation. *See* DBI 动态二进制插装，见 DBI
Dynamic Data Exchange (DDE) 动态数据交换（DDE），284
Dynamic HTML (DHTML) 动态 HTML（DHTML），24
DynamoRIO DBI system DynamoRIO DBI 系统，491

E

ECMAScript,ECMAScript，276
Eddington，Michael，40
elements ofprotocols 协议元素
block identifiers/sizes 块标识符/大小，58
checksums 校验值，58
name–value pairs 名称-值对，57
encoders (blocks) 编码器（块），394
Enterprise Resource Planning (ERP) 企业资源计划（ERP），117
enumerating 枚举
environment variables 环境变量
GDB method GDB 方法，96-97
getenv function getenv 函数，98
library preloading 库预加载，98-99

loadable ActiveX controls 可加载的ActiveX控件，289-293
environment variables 环境变量，38-39，90-92
GDB method GDB方法，96-97
getenv function getenv函数，98
library preloading 库预加载，98-99
ERP (Enterprise Resource Planning) ERP（企业资源计划），117
error detection 错误检测，67-69
DBI platforms DBI平台，68
debug clients 调试客户端，67
ping checks ping检查，67
Ethereal client,server data capture Ethereal客户端，服务器数据抓取，335
event logs 事件日志
debugging 调试，322-323
logs 日志
process information 处理信息，178
web application fuzzing web应用模糊测试，136
Web browser errors Web浏览器错误，282
Evron，Gadi，25
Excel eBay vulnerability Excel eBay漏洞，199
exceptional elements 异常元素，510
exceptions 异常
debug exception events 调试异常事件，323
detecting 检测，183，203-204，213-217
first-chance versus last-chance 首轮与末轮，489-491
monitoring 监视，28
reporting 报告，183-184
web applications web应用，135-136，147
Execute() method Execute()方法，299
executing 执行
fuzz data 模糊数据，28
instructions 指令，438。*See* also tracking
execution control,transferring 参见“跟踪”
execution control,transferring 执行控制，转移，475-476
exit nodes (GAs) 退出节点（GAs），434
exploitability 可利用性，28
External Data Representation (XDR) 外部数据表示（XDR），230

F

fault detection 故障检测
crash locations,viewing 崩溃位置，查看，405-406
DBI，492
execution control,transferring 执行控制，转移，475-476
first-chance versus last-chance exceptions 首轮和末轮异常，489-491
format string vulnerability 格式字符串漏洞，477
frameworks 框架，353
graphical representation 图形化表示，406
in-memory fuzzing 内存模糊测试，311-312
network protocol fuzzing 网络协议模糊测试，232-233
network protocol fuzzing 网络协议模糊测试，254

page faults 页故障，474
primitive 原始类型，472-474
reading addresses 读取地址，476-478
software memory corruption vulnerability 软件内存破坏漏洞
categories 类别，475
stack layout example 栈布局示例，477
web browsers web 浏览器，282
writing to addresses 写入地址，478-479
fencing memory allocation 带边界内存分配，492
field delimiters 字段分隔符，83-84
file format fuzzers 文件格式模糊测试器，39，54-57
benefits 好处，221
case study 案例研究，217-220
detection 检测，178-179
development 开发，209
applications,launching 应用，启动，211-213
approach 方法，209
core fuzzing engines 核心模糊测试引擎，184-185
design 设计，210-217
exception detection 异常检测，183-184，213-217
files,creating 文件，创建，210
language,choosing 编程语言，选择，210
source files,reading 源代码，读取，210-211
writing to files 写入文件，211
FileFuzz
applications,launching 应用，启动，203
ASCII text files ASCII 文本文件，202
audits,saving 审计，保存，204-205
binary files 二进制文件，201
exception detection 异常检测，203-204
features 特性，201
goals 目标，200
forking off/tracing child processes 创建/跟踪子进程，185-187
future improvements 未来的改进，221
interesting UNIX signals 我们感兴趣的 UNIX 信号，187
methods 方法，171-172
brute force 强制，172-173
inputs 输入，174
intelligent brute force 智能强制，173-174
not interesting UNIX signals 我们不感兴趣的 UNIX 信号，188
notSPIKEfile
features 特性，182
missing features 缺少的特性，182
programming language 编程语言，195
RealPix format string vulnerabilityRealPix 格式字符串漏洞，193-195
shell script workarounds shell 脚本解决方案，192
ODF，54
Open XML，54
SPIKEfile
features 特性，182

missing features 缺少的特性，182
programming language 编程语言，195
shell script workarounds shell 脚本解决方案，192
targets 目标，170-171，205，208-209
Windows Explorer Windows 文件浏览器，206-209
Windows Registry Windows 注册表，209
vulnerabilities 漏洞
DoS，175
examples 示例，170
format strings 格式字符串，177
heap overflows 堆溢出，177
integer handling 整数处理，175-177
logic errors 逻辑错误，177
race conditions 竞争条件，178
race conditions 简单栈，177
Windows，197-198
zombie processes 僵尸进程，189-191
File Transfer Protocol (FTP) 文件传输协议（FTP），48，362-363
FileFuzz，39
applications,launching 应用，启动，203
ASCII text files ASCII 文本文件，202
audits,saving 审计，保存，204-205
benefits 好处，221
binary files 二进制文件，201
case study 案例研究，217-220
development 开发，209
applications,launching 应用，启动，211-213
approach 方法，209
design 设计，210-217
exception detection 异常检测，213-217
files,creating 文件，创建，210
language,choosing 编程语言，选择，210
source files,reading 源代码，读取，210-211
writing to files 写入文件，211
error detection 错误检测，67
exception detection 异常检测，203-204
features 特性，201
future improvements 未来的改进，221
goals 目标，200
files. *See* also file format fuzzers 文件。参见“文件格式模糊测试器”
fuzzing 模糊测试，24
permissions 权限，95
rule 规则，359-360
SWF，372
bit_field class bit_field 类，378
component relationships 组件关系，383
data generation 数据生成，384
data structures 数据结构，374-377
dependent_bit_field class dependent_bit_field 类，379
environment 环境，385
header 头，372-373
MATRIX struct MATRIX 结构，379-380
methods 方法，385

RECT/RGB structs RECT/RGB 结构，378
string primitives 字符串原始类型，383
SWF files,modeling SWF 文件，建模，374
tags 标签，374
tags,defining 标签，定义，380-382
FilesAttributes tag (Flash) FilesAttributes 标签（Flash），374
filtering tracking recordings 过滤跟踪记录，446
find command 查找命令，93
first-chance exceptions 首轮异常，489-491
fixed length fields (protocols) 定长字段（协议），47
Flash，115，279
flatten() routine flatten()例程，378
Flawfinder website Flawfinder 网站，7
forking offchild processes 创建子进程，185-187
format string vulnerabilities 格式字符串漏洞，85，106
example 示例，477
exploiting 利用，85
file formats 文件格式，177
RealPlayer RealPix format string RealPlayer RealPix 格式字符串，193-195
frameworks 框架，43-44
antiparser 反解析器，354-356
attack heuristics 攻击启发式列表，353
Audodafe[as]，369-371
automatic length calculation 自动长度计算，352
code reusability 代码可复用性，354
community involvement 社区介入，43
complexity 复杂性，44
CRC values CRC 值，353
data parsing 数据解析，353
development time 开发时间，44
Dfuz，357-360
data sources 数据源，358
functions 函数，357-358
lists,declaring 列表，声明，358
protocols,emulating 协议，仿真，359
rule files 规则文件，359-360
variables,defining 变量，定义，357
fault detection 故障检测，353
features 特性，352-353
fuzzing metrics，353
GPF，366-369
limitations 局限，43
overview 概述，352-354
PaiMei，449
advanced target monitoring 高级目标监视，486-489
basic target monitoring 基础目标监视，482-484
components 组件，450
crash binning utility 崩溃二进制日志工具，487-489
PIDA interface PIDA 接口，450
PStalker module. *See* Pstalker PStalker 模块。见 PStalker
SWF fuzzing SWF 模糊测试，385
Peach，364-366

code reusability 代码可复用性，366
disadvantages 缺点，366
generators 生成器，364
groups 组，365
publishers 发布者，364
transformers 转化者，364
PI，428
programming languages 编程语言，352
protocol modeling 协议建模，352
pseudo-random data,generating 伪随机数，生成，353
reusability 可复用性，43
SPIKE，361-364
block-based protocol representation 基于块的协议表示，361-362
disadvantage 缺点，363
FTP fuzzer FTP 模糊测试器，362-363
Sulley，386-387
blocks 块，392-397
data types 数据类型，388-390
delimiters 分隔符，391
directory structure 目录结构，387-388
download website 下载网站，386
environment,setting up 环境，设置，413
features 特性，386
fuzzers,launching 模糊测试器，启动，414-416
integers 整数，390-391
legos 积木，398-399
postmortem phase 事后阶段，405-409
requests,building 请求，构建，410-411
RPC endpoint walkthrough. *See* Sulley framework RPC 端点介绍。见“Sulley 框架”
RPC endpoint walkthrough RPC 端点介绍
session 会话，399-404，411-413
strings 字符串，391
FTP (File Transfer Protocol) FTP（文件传输协议），48，362-363
functions. *See* also routines 函数。参见“例程”
BindAdapter()，259
byref()，318
CreateProcess()，10，203
create_string_buffer()，318
Dfuz，357-358
GetCurrentProcessId()，318
getenv，98
printf()，248
ReadProcessMemory()，318
ReceivePacket()，259
s_block_end()，242，392
s_block_start()，242，392
taboo()，475
WriteProcessMemory()，319
func_resolve() routine func_resolve()例程，333
future offuzzing 模糊测试的未来
development environment integration 开发环境集成，516
hybrid analysis ofvulnerabilities 混合漏洞分析，515
fuzz_client.exe，334-346
fuzz_server.exe，334

conceptual diagram 概念图，339
launching 启动，345
memory allocation 内存分配，344-345
OllyDbg，336-338
restore point 恢复点，335
snapshot 快照，335，343
WS2_32.dll recv() breakpoint WS2_32.dll recv() 断点，336
fuzz_trend_server_protect_5168.py，414-416
fuzz values,choosing 模糊值，选择，480-481
fuzzer stepping 模糊测试器单步，473
fuzzing 模糊测试
as black box testing 作为黑盒测试，12
defined 定义，21
history 历史，22-27
ActiveX，25
Codenomicon，23
files 文件，24
Miller,Professor Barton,Miller Barton 教授，22
PROTOS test suites PROTOS 测试套件，23
sharefuzz，24
SPIKE，23
limitations 局限
access control 访问控制，29
backdoors 后门，30
design logic 设计逻辑，30
memory corruption 内存破坏，31
multistage vulnerabilities 多步漏洞，32
phases 阶段，27-29
Fuzzing mailing list 模糊测试邮件列表，25
fuzzing metrics 模糊测试度量，353

G

GAs (genetic algorithms) GAs（遗传算法），431-435
CFG
connecting path 连接路径，433
exit nodes 退出节点，434
potential vulnerability 潜在漏洞，433
fitness function 适应度函数，432
reproduction function 繁殖函数，431
Sidewinder tool Sidewinder 工具，433-434
as stochastic global optimizers 作为带随机性的全局优化器，432
GDB (GNU debugger) GDB（GNU 调试器），16，96-97
GDI+ buffer overflow vulnerability, 198 GDI+ 缓冲区溢出漏洞，198，217-220
General Purpose Fuzzer (GPF) 通用目的模糊测试器（GPF），366-369
generating data 生成数据
CCR example CCR 示例，78-79
data types 数据类型
character translations 字符翻译，85
command injection 命令注入，87
directory traversal 目录遍历，86
field delimiters 字段分隔符，83-84
format strings 格式字符串，85
integer values 整数值，79-81
string repetitions 字符串重复，82
generators 生成器，364
pseudo-random data 伪随机数，353

SWF fuzzing test case SWF 模糊测试用例，384
generation-based fuzzers 基于生成的模糊测试器，22，231
generators 生成器，364
generic line-based TCP fuzzer 基于行的通用TCP 模糊测试器，240-243
generic script-based fuzzers 基于脚本的通用模糊测试器，244
genetic algorithms. *See* GAs 遗传算法，见“GAs”
GET method GET 方法，123
GetCurrentProcessId() function GetCurrentProcessId()函数，318
getenv function getenv 函数，98
GetFuncDesc() routine GetFuncDesc()例程，296
GetNames() routine GetNames()例程，295
getopt module getopt 模块，104，107
getPayload() routine getPayload()例程，356
GetThreadContext() API，327
GetTypeInfoCount() routine GetTypeInfoCount()例程，294
GetURL() method GetURL()方法，299
Gizmo Project case study Gizmo 项目案例研究
data sources 数据源，462
execution,monitoring 执行，监视，464-465
features list 特性列表，457
results 结果，465
SIP
packets,testing 数据包，测试，458-461
processing code 处理代码，462
strategy 策略，458-461
tags,choosing 标签，选择，462
tracking capture options 跟踪抓取选项，462-463
GNU debugger (GDB) GNU 调试器（GDB），16，96-97
Gosling，James，116
GPF (General Purpose Fuzzer) GPF（通用目的模糊测试器），366-369
graphs 图形
call graphs 调用图，439
CFGs，440-441
disassembled dead listing 反汇编得到的指令列表，441
GAs，433-434
example SMTP session 示例的 SMTP 会话，399-401
Sulley crash bin graphical representationSulley 崩溃二进制日志的图形化表示，406
gray box testing 灰盒测试，14
binary auditing 二进制审计
automated 自动审计，17-18
manual 手工审计，14-16
pros/cons 优点/缺点，18
Greene，Adam，38
groups 组
blocks 块，392-393
Peach framework Peach 框架，365

H

Hamachi，24，42

handler_bp() routine handler_bp()例程，333
hardware breakpoints 硬件断点，324
HEAD method HEAD 方法，124
headers 头
Accept，122
Accept-encoding，122
Accept-Language，122
Connection，123
Cookie，123
Host，122
HTML，271
HTTP protocol HTTP 协议，122
SWF，372-373
User-Agent，122
web applications input web 应用输入，128
heap overflow vulnerabilities 堆溢出漏洞，129
browsers 浏览器，277-279
file formats 文件格式，177
Heller，Thomas，318
heuristics 启发方法，79
ActiveX，298
attack 攻击，353
character translations 字符翻译，85
command injection 命令注入，87
directory traversal 目录遍历，86
field delimiters 字段分隔符，83-84
format strings 格式字符串，85
integer values 整数值，79-81
protocol dissection 协议分析，421
disassembly-level 反汇编层次，426-427
hierarchical breakdown 逐层分解，424
improved analysis 改进分析，425-426
proxy fuzzing 代理模糊测试，422-423
string repetition 字符串重复，82
Hewlett-Packard Mercury LoadRunner vulnerability HP Mercury LoadRunner 漏洞，263
hexadecimal encoding/decoding 16 进制编码/解码，262
hierarchy 层次
protocols 协议，424
Sulley directory structure Sulley 目录结构，387-388
history 历史
COM，284
fuzzing 模糊测试，22-27
ActiveX，25
Codenomicon，23
files 文件，24
Miller,Professor Barton Miller，Barton 教授，22
PROTOS test suites PROTOS 测试套件，23
sharefuzz，24
SPIKE，23
Samba，420
Hoglund，Greg，312
Holodeck，514-515
hook points 钩子点，331
hooking processes 钩住进程，324-327
context modifications 上下文修改，327
EIP,adjusting EIP，调整，326
INT3 opcode,writing INT3 操作码，写入，324
original bytes at breakpoint address,saving

断点地址处的原字节，保存，324
software breakpoint,catching 软件断点，抓住，325
Host header Host 头，122
HTML (Hypertext Markup Language) HTML（超文本标记语言）
header vulnerabilities 头漏洞，271
status codes 状态码，146
tag vulnerabilities 标签漏洞，271-273
HTTP (Hypertext Transfer Protocol) HTTP(超文本传输协议）
field delimiters 字段分隔符，83
methods 方法，133
protocol 协议，122
requests,identifying 请求，识别，144
response splitting 响应拆分，134
status codes 状态码，135
human computation power 人工计算能力，73
hybrid analysis ofvulnerabilities 混合漏洞分析，515
Hypertext Preprocessor (PHP) 超文本预处理器（PHP），115

I

IAcroAXDocShim interface IAcroAXDocShim 接口，295
IBM
AIX 5.3 local vulnerabilities AIX 5.3 本地漏洞，110
Rational Purify，492
ICMP,dissecting ICMP，分析，429-430
IDA (Interactive Disassembler) IDA（交互式反汇编器），15
IDA Pro (DataRescue Interactive Disassembler) IDA Pro（DataRescue 交互式反汇编器），439
IDE (integrated development environment) IDE（集成开发环境），503
IETF (Internet Engineering Task Force) IETF（互联网工程任务组），54
iFuzz，38
case study 案例研究，110-111
development approach 开发方法，105-107
features 特性，103-105
fork,execute,and wait approach 复制，创建和等待方法，107
fork,ptrace/execute,and wait/ptrace fork,ptrace/execute, 和 wait/ptrace
approach 方法，108-109
getopt hooker getopt 钩子，105
language 编程语言，109
modules 模块，104
argv，104-106
getopt，104，107
preloadable getenv fuzzer,可预加载的 getenv 模糊测试器，105
single-option/multi-option 单选项/多选项，104，107
postdevelopment observations 开发后的观察，111-112
IIDs (interface IDs) IIDs（接口 ID），285
improperly supported HTTP methods 不恰当支持的 HTTP 方法，133
in-memory fuzzers 内存模糊测试器，42-43
advantages 优势，42，302
complexity 复杂性，43

control flow diagram example 控制流图示例，306
example 示例
access violation handler 访问违例处理器，340-342
breakpoint handler 断点处理器，342
client,launching 客户端，启动，334
conceptual diagram 概念图，339
data mutation 数据变异，343
fuzz_client,launching,fuzz_client 启动，346
fuzz_server memory allocation fuzz_server 内存分配，344-345
fuzz_server snapshot fuzz_server 快照，343
fuzz_server,launching fuzz_server，启动，345
false positives 正确报告，43
fault detection 故障检测，311-312
hook points,choosing 钩子点，选择，331
language,choosing 编程语言，选择，317-319
memory mutation locations, choosing 内存变异位置，选择，331
methods 方法
mutation loop insertion 变异循环插入，308-309，316
snapshot restoration mutation 快照恢复变异，309-310，316
overview 概述，302-307
process depth 进程深度，310-311
process hooking 进程钩入，324-327
context modifications 上下文修改，327
EIP,adjusting EIP，调整，326
INT3 opcode,writing INT3 操作码，写入，324
original bytes at breakpoint address, saving 断点地址处的原字节，保存，324
software breakpoint,catching 软件断点，抓住，325
process snapshots/restores 进程快照/恢复，327-331
memory block contents,saving 内存块内容，保存，329-330
thread context,saving 线程内容，保存，328-329
reproduction 重现，43
required feature sets 需要的特性集，316-317
restore point 恢复点，335
speed,testing 速度，测试，310-311
shortcuts 捷径，42
snapshot point 快照点，335
speed 速度，42
targets 目标，307-308
WS2_32.dll recv() breakpoint WS2_32.dll recv()断点，336
implementing with PyDbg 用 PyDbg 实现，340-345
obfuscation method 加密方法，335
OllyDbg，336-338
server data capture 服务器数据抓取，335
server,launching 服务器，启动，334
inputs,输入
file format fuzzing 文件格式模糊测试，174

identifying 识别，27
web applications web 应用
choosing 选择，119-121
cookies，129
headers 头，128
identifying 识别，130-131
method 方法，123-125
post data post 数据，130
protocol 协议，128
request-URI 请求 URI，126-128
web browser fuzzing web 浏览器模糊测试
ActiveX controls ActiveX 控件，273-275
client-side scripting 客户端脚本，276
CSS，275-276
Flash，279
HTML headers HTML 头，271
HTML tags HTML 标签，271-273
URLs，280
XML tags XML 标签，273
Inspector，18，312
instructions,tracing 指令，跟踪，444-445.*See also* tracking 见“跟踪”
INT3 opcode,writing INT3 操作码，写入，324
integers 整数
handling vulnerabilities 处理整数的漏洞，175-177
Sulley framework Sulley 框架，390-391
values 值，79-81
integrated development environment (IDE) 集成开发环境（IDE），503
intelligence 智能
black box testing 黑盒测试，14
brute force fuzzing 强制模糊测试
file formats 文件格式，173-174
network protocols 网络协议，231
fault detection 故障检测
first-chance versus last-chance exceptions 首轮和末轮异常，489-491
page faults 页故障，474
software memory corruption vulnerability categories 软件内存破坏漏洞类别，475-479
load monitoring 加载监视，182
Interactive Disassembler (IDA) 交互式反汇编器（IDA），15
interface IDs (IIDs) 接口 IDs（IIDs），285
interfaces 接口
COM，284
IAcroAXDocShim，295
IObjectSafety，292
PIDA，450
Sulley Web monitoring Sulley Web 监视，404
Internet protocols 互联网协议，46
Internet Engineering Task Force (IETF) 互联网工程任务组（IETF），54
IObjectSafety interface IObjectSafety 接口，292
Ipswitch
I-Mail
vulnerability 漏洞，420
Web Calendaring directory traversal Web Calendaring 目录遍历，157
Whatsup Professional SQL injection attack 1Whatsup Professional SQL 注入攻击，118，160-162
ITS4 download website ITS4 下载网站，7

J

Java，116
JavaScript，116
Jlint website　Jlint 网站，7

K

kill bitting　kill bitting（Windows 禁止 IE 加载指定 CLSID 加载的机制）

L

languages (programming)　语言（编程）
choosing　选择，77
ECMAScript，276
FileFuzz，210
frameworks　框架，352
iFuzz，109
in-memory fuzzing　内存模糊测试，317-319
ProtoFuzz，256
Python. *See* PythonPython　见“Python”
SPIKEfile/notSPIKEfile tools　SPIKEfile/notSPIKEfile 工具，195
WebFuzz，148
last-chance exceptions　末轮异常，489-491
Layer 2 vulnerabilities　第二层漏洞，228
Layer 3 vulnerabilities　第三层漏洞，229
Layer 4 vulnerabilities　第四层漏洞，229
Layer 5 vulnerabilities　第五层漏洞，229
Layer 6 vulnerabilities　第六层漏洞，230
Layer 7 vulnerabilities　第七层漏洞，230
legos　积木，398-399
libdasm library　libdasm 库，75
libdisasm library　libdisasm 库，75
Libnet library　Libnet 库，76
LibPCAP library　LibPCAP 库，76
libraries　库
data types,including　数据类型，包括
character translations　字符翻译，85
command injection　命令注入，87
directory traversal　目录遍历，86
field delimiters　字段分隔符，83-84
format strings　格式字符串，85
integer values　整数值，79-81
string repetitions　字符串重复，82
libdasm，75
libdisasm，75
Libnet，76
LibPCAP，76
Metro Packet Library　Metro Packet 库，76
packet capture　数据包抓取，256-257
PaiMei，299
preloading　预加载，98-99
PyDbg，445-446
SIPPhoneAPI，462
vulnerable web applications　有漏洞的 web 应用，134
WinPcap，256
limitations　限制
frameworks　框架，43
fuzzing　模糊测试
access control　访问控制，29
backdoors　后门，30
design logic　设计逻辑，30

memory corruption 内存破坏，31
multistage vulnerabilities 多步漏洞，32
resource constraints 资源限制，69
PStalker，454
LoadTypeLib() routine LoadTypeLibrary()例程，294
local fuzzers 本地模糊测试器，37
abort signals 中止信号，100
command-line 命令行，37-38，89
environment variable 环境变量，38-39，90-92
GDB method GDB 函数，96-97
getenv function getenv 函数，98
library preloading 库预加载，98-99
file format 文件格式，39
iFuzz
case study 案例研究，110-111
development approach 开发方法，105-107
features 特性，103-105
fork,execute,and wait approach 复制，创建和等待方法，107
fork, ptrace/execute, 和 wait/ptrace approach 方法，108-109
getopt hooker getopt 钩子，105
language 语言，109
modules 模块，104-106
postdevelopment observations 开发后观察，111-112
preloadable getenv fuzzer 可预加载的 getenv 模糊测试器，105
methods 方法，95
principles 原则，92
problems,detecting 问题，检测，99-101
ptrace method ptrace 方法，100
su application example su 应用示例，92
targets 目标，93-95
log files 日志文件
analysis vulnerabilities 漏洞分析，118
web application fuzzing web 应用模糊测试，136
logic error vulnerabilities 逻辑错误漏洞，177
LogiScan，17
loops,debug event 循环，调试事件，322-323

M

Macbook vulnerability Macbook 漏洞，228
Macromedia Flash，115，279
Macromedia Shockwave Flash. *See* SWF
Macromedia Shochwave Flash，见“SWF”
malformed arguments 变形的参数，37
mangleme 加密，24，42
manual testing 手工测试
applications 应用，10
protocol mutation 协议变异，35
RCE，14-16
Matasano’s Protocol Debugger 4Matasano 的 Protocol Debuger 工具，423
MATRIX struct MATRIX 结构，379-380
media server vulnerability 媒体服务器漏洞，226
memory 内存
allocating 分配，344-345，492
block contents,saving 块内容，保存，329-330

corruption 破坏，31
in-memory fuzzing 内存模糊测试
benefits 好处，302
control flow diagram example 控制流图示例，306
example. *See* in-memory fuzzers, example 示例。见“内存模糊测试器”的“示例”
fault detection 故障检测，311-312
hook points,choosing 钩子点，选择，331
language,choosing 编程语言，选择，317-319
memory mutation locations, choosing 内存变异位置，选择，331
mutation loop insertion 变异循环插入，308-309，316
overview 概述，302，306-307
process depth 进程深度，310-311
process hooking 进程钩着，324-327
process snapshots/restores 进程快照/恢复，327-331
required feature sets 需要的特性集，316-317
snapshot restoration mutation 快照恢复变异，309-310，316
speed,testing 速度，测试，310-311
targets 目标，307-308
mutable blocks,creating 可变块，创建，318
mutation locations,choosing 变异位置，选择，331
process memory reading/writing 进程内存读取/写入，318-319
protection attributes 保护属性，303
software memory corruption vulnerabilities 软件内存破坏漏洞，475
execution control，transferring 执行控制，转移，475-476
reading addresses 读取地址，476-478
writing to addresses 写入地址，478-479
Windows memory model Windows 内存模型，302-306
metadata test case 元数据测试用例，179
method input (web applications) 函数输入（web 应用），123-125
methods . *See* also functions;routines 方法，见“函数”和“例程”
ActiveX，293-297
automatic protocol generation testing 自动协议生成测试，36
BeginRead()，152
BeginWrite()，151
brute force testing 强制测试，36
btnRequest_Click()，153
CONNECT，125
CreateFile()，299
CreateProcess()，299
DELETE，124
DownloadFile()，299
Execute()，299
file format fuzzing 文件格式模糊测试，171-174
brute force 强制测试，172-173
inputs 输入，174
intelligent brute force 智能强制测试，173-174

GET，123
GetURL()，299
HEAD，124
HTTP，133
in-memory fuzzing　内存模糊测试
mutation loop insertion　变异循环插入，308-309，316
process depth　进程深度，310-311
snapshot restoration mutation　快照恢复变异，309-310，316
local fuzzing　本地模糊测试，95
manual protocol mutation testing　手工协议变异测试，35
network protocol fuzzing　网络协议模糊测试，230
brute force　强制，231
generation-based　基于生成的，231
intelligence brute force　智能强制，231
modified client mutation　修改客户端变异，232
mutation-based　基于变异，231
OnReadComplete()，152
OPTIONS，125
POST，124
pregenerated test cases　预生成的测试用例，34
ptrace，100
PUT，124
random　随机，34
SWF fuzzing　SWF 模糊测试，385
TRACE，124
web browser fuzzing　web 浏览器模糊测试，269
approaches　方法，269-271
inputs　输入，275-280
Metro Packet Library　Metro 数据包库，76
Microsoft　微软
COM　见“COM”
fuzzing　模糊测试，13
NDIS protocol driver　NDIS 协议驱动，255
Open XML format　开放 XML 格式，54
Remote Procedure Call（MSRPC）　远程过程调用（MSRPC），40
Samba，420
security　安全性，498-499
source code leak　源代码泄漏，5
vulnerabilities　漏洞
Excel eBay vulnerability　Excel eBay 漏洞，199
GDI+ buffer overflow　GDI+缓冲区溢出，198，217-220
Outlook Express NNTP vulnerabilityOutlook Express NNTP 漏洞
discovery　发现，447-449
Outlook Web Access Cross- Site Scripting　Outlook Web 访问的跨站脚本问题，117
PNG，199
WMF，199
Windows
Live/Office Live，114
memory model　内存模型，302-306
worms　蠕虫，224-226
Miller, Professor Barton　Miller，Barton 教授，22
MLI (mutation loop insertion)　变异循环插入（MLI），308-309，316

MMalloc() routine MMalloc()例程，81
MoBB (Month of Browser Bugs) 浏览器缺陷月（MoBB），268
modified client mutation fuzzing 修改客户端变异模糊测试，232
modules 模块
ctypes，318
iFuzz，104-107
监视
ActiveX fuzzer ActiveX 模糊测试器，299
automated debugger 自动化调试器，481
advanced example 高级示例，486-489
architecture 架构，481
basic example 基础示例，482-484
PaiMei crash binning utility PaiMei二进制崩溃日志工具，487-489
exceptions 异常，28
instruction execution 指令执行，见“跟踪”
network vulnerabilities 网络漏洞，118
Month of Browser Bugs (MoBB) 浏览器缺陷月（MoBB），268
Moore, H.D.，24-25，42
MSRPC (Microsoft Remote Procedure Call) 微软远程过程调用（MSRPC），40
Mu Security，25
Mu-4000，512
multi-option module (iFuzz) 多选项模块（iFuzz），104，107
Multiple Vendor Cacti Remote File Inclusion Vulnerability 影响多个软件供应商的Cacti远程文件包含漏洞
website 网站，118
multistage vulnerabilities 多步漏洞，32
Murphy，Matt，24
mutable memory blocks，creating 可变内存块，创建，318
mutation loop insertion (MLI) 变异循环插入（MLI），309，308-309，316
mutation-based fuzzers 基于变异的模糊测试器，22，231

N

name-value pairs 名称-值对，57
naming schemes 命名方案，439
NDIS (Network Driver Interface Specification) protocol 网络驱动接口规范（NDIS）协议
driver 驱动，255
Needleman-Wunsch algorithm Needleman-Wunsch 算法，428
Netcat，49
NetMail Networked Messaging Application Protocol NetMail 网络消息应用协议，见“NMAP”
Network Driver Interface Specification (NDIS) protocol 网络驱动接口规范（NDIS）协议
driver 驱动，255
Network News Transfer Protocol (NNTP) 网路新闻传输协议（NNTP），447
networks 网络
adapters 适配器，257-258
client-side vulnerabilities 客户端漏洞，223
layer vulnerabilities 网络层漏洞，229
monitoring 监视，118，402
protocol fuzzer 协议模糊测试器，40

complex protocols 复杂协议，40
defined 定义，224
fault detection 故障检测，232-233，254
methods 方法，230-232
protocol drivers 协议驱动，255
ProtoFuzz 见“PtotoFuzz”
requirements 需求，250-253
simple protocol 简单协议，40
server-side vulnerabilities 服务端漏洞，223
socket communication component 套接字通信组件，224
targets 目标，226-230
application layer 应用层，230
categories 分类，226
data link layer 数据链路层，228
network layer 网络层，229
presentation layer 表示层，230
session layer 会话层，229
transport layer 传输层，229
vulnerabilites，226
UNIX system UNIX 系统，236
SPIKE NMAP fuzzer script SPIKE NMAP 模糊测试器脚本，244-248
目标，236-237
Windows 见“ProtoFuzz”
NMAP (NetMail Networked Messaging Application Protocol) 网络信息应用协议（NMAP），236
overview 概述，237-240
SPIKE NMAP fuzzer script SPIKE NMAP 模糊测试器脚本，245-248
NNTP (Network News Transfer Protocol) 网络新闻传输协议（NNTP），447
noalphanumeric characters 非字母数字字符，83-84
nonexecutable (NX) page permissions 非可执行（NX）页面权限，476
notSPIKEfile，39
core fuzzing engine 核心模糊测试引擎，184-185
exception handling 异常处理，183-184
features 特性，182
forking off/tracing child processes 创建/跟踪子进程，185-187
interesting UNIX signals 我们感兴趣的 UNIX 信号，188
programming language 编程语言，195
RealPix format string vulnerabilities RealPix 格式字符串漏洞，193-195
shell script workarounds shell 脚本解决方案，192
zombie processes 僵尸进程，189-191
Novell NetMail IMAPD Command Continuation Request Novell NetMail IMAPD 命令继续请求
Heap Overflow security advisory 堆溢出安全性报告，81
NW (Needleman-Wunsch) algorithm NW 算法，428
NX (nonexecutable) page permissions 非可执行页面权限，476

O

OASIS (Organization for the Advancement of Structured Information Standers) 结构化信息标准高级组织（OASIS），54

ODF (OpenDocument format) 开放文档格式（ODF），54
Office Live，114
OLE (Object Linking and Embedding) 对象链接与嵌入（OLE），284
OllyDbg，16，335
before/after demangel 加密之前/之后，336
conceptual diagram 概念图，339
parse routine parse 例程，338
restore routine restore 例程，338
WS2_32.dll recv() breakpoint WS2_32.dll recv()断点，336
OnReadComplete() methods OnReadComplete()方法，152
Open Document format (ODF) 开放文档格式（ODF），54
open protocol 开放协议，54
open source disassemble libraries 开源反汇编库，75
Open System for Communication in Realtime (OSCAR) 实时通信开放系统（OSCAR），49
open XML format 开放 XML 格式，54
OpenSSH Remote Challenge vulnerabilityOpenSSH 远程挑战漏洞，226
operation system logs, fault detection 操作系统日志，故障检测，233
OPTIONS method OPTIONS 方法，125
Organization for the Advancement of Structured Information Standard (OASIS) 结构化信息标准高级组织（OASIS），54
OSCAR (Open System for communication in Realtime) OSCAR（实时通信开放系统），49
Outlook Express NNTP vulnerability discoveryOutlook Express NNTP 漏洞发现，447-449
Overflow fuzz variable 溢出模糊变量，142
overflows (stack) 溢出（栈），246
OWASP (WebScarab) OWASP（WebScarab），41

P

packets 数据包
assembling 汇编，250-251
capture libraries，choosing 抓取库，选择，256-257
page faults 页故障，474
PAGE_EXECUTE attribute PAGE_EXECUTE 属性，303
PAGE_EXECUTE _READ attribute PAGE_EXECUTE_READ 属性，303
PAGE_EXECUTE _READWRITE attributePAGE_EXECUTE _READWRITE 属性，303
PAGE_NOACCESS_READ attribute PAGE_NOACCESS_READ 属性，304
PAGE_READONLY attribute PAGE_READONLY 属性，304
PAGE_ READWRITE attribute PAGE_ READWRITE 属性，304
PaiMei framework PaiMei 框架
ActiveX fuzzer ActiveX 模糊测试器，299
crash binning utility 崩溃二进制日志工具，487-489
SWF fuzzing SWF 模糊测试，385
target monitoring 目标监视
advanced example 高级示例，486-489

basic example 基础示例，482-484
PAIMEI filefuzz，39
PAIMEIpstallker 见“PStalker”
PAM(Percent Accepted Mutation) 百分比接受变异，429
parameters(ActiveX) 参数(ActiveX)，293-297
parse()routine parse()例程，306
parsing 解析
data 数据
networks 网络，151-152
ProtoFuzz design ProtoFuzz 设计，259-261
framework features 框架特性，353
passing data types 传入数据类型，318
Pattern Fuzz (PF) 模式模糊测试（PF），368
PDB (Protocol Debugger) 协议调试器（PDB），423
PDML2AD，371
Peach framework Peach 框架，40, 364-366
code reusability 代码可复用性，366
disadvantages 缺点 366
generators 生成器，364
groups 组，365
publishers 发布者，364
transformers 变化者，364
Percent Accepted Mutation (PAM) 百分比接受变异，429
performance 性能
degradation 下降，147
monitors, Web browser errors 监视器，Web 浏览器错误，282
PF (Pattern Fuzz) 模式模糊测试（PF），368
phases of fuzzing 模糊测试的阶段，29
exceptions, monitoring 异常，监视，28
exploitability 利用，28
fuzz data, executing/generating 模糊数据，执行/生成，28
inputs, identifying 输入，识别，27
targets, identifying 目标，识别，27
phishing vulnerabilities 钓鱼漏洞，281
PHP (Hypertext Preprocessor) 超文本预处理器（PHP），115
phpBB Group phpBB Arbitrary File Disclosure Vulnerability website phpBB 组 phpBB 任意文件暴露漏洞网站，117
PI (Protocol Informatics) 协议信息学，428
PIDA interface PIDA 接口，450
Pierce, Cody, 39
Pin DBI system Pin DBI 系统，492
pinging, error detection pinging 错误检测，67
plain text protocols 普通文本协议，48-49
PNG (Portable Network Graphics) vulnerability 可移植网络图形（PNG）漏洞，199
post data, web application input post 数据，web 应用输入，130
POST method POST 方法，124
postmorterm phrase (Sulley) 事后分析阶段（Sulley），405-409
crash bin graphical exploration 崩溃二进制日志图形化研究，406
crash locations, viewing 崩溃位置，查看，405-406
precompile security solution 预编译安全解决方案，480
pregenerated test cases 预生成的测试用例，34
preloading libraries 预加载库，98-99

presentation layer vulnerabilities 表示层漏洞，230
primitive fault detection 原始故障检测，472-474
principles of local fuzzing 本地模糊测试的原则，92
printf() function printf()方法，248
Pricess Stalker 进程 Stalker，见“PStalker”
processes 进程
child, forking off and tracing 子进程，创建与跟踪，185-187
depth 深度，64-66
hooking 钩子，324-327
context modifications 内容修改，327
EIP, adjusting, EIP 调整，326
INT3 opcode, writing INT3 操作码，写入，324
original bytes at breakpoint address, saving 断电处原先的字节，保存，324
points, choosing 点，选择，331
software breakpoint, catching 软件断点，抓获，325
memory 内存
reading 读取，318
writing 写入，319
monitor agent 监视代理，403
snapshots/restores 快照/恢复
handling 处理，327-331
memory block contents, saving 内存块内容，保存，329-330
thread context, saving 线程上下文，保存，328-329
state 状态，64-66
zombie 僵尸进程，189-191
process_restore() routine process_ restore() 例程，345
process_snapshot() routine process_ snapshot() 例程，343
profiling targets 剖析目标，443-444
ProgIDs (program IDs) 程序 ID（ProgIDs），285
programming languages 编程语言
choosing 选择，77
ECMAScript，276
FileFuzz，210
frameworks 框架，352
iFuzz，109
in-memory fuzzing 内存模糊测试，317-319
ProtoFuzz，256
Python 见“Python”
SPIKEfile/notSPIKEfile tools SPIKEfile/not SPIKEfile 工具，195
WebFuzz，148
properties (ActiveX) 属性（AcitiveX），293-297
proprietary protocols 私有协议，54, 421
Protocol Debugger (PDB) 协议调试器（PDB），423
Protocol Informatics (PI) 协议信息学（PI），428
protocol-specific fuzz scripts 面向协议的模糊测试脚本，244
protocol-specific fuzzers (SPIKE) 面向协议的模糊测试器（SPIKE），243
protocols 协议

AIM，49-52
logon credentials sent 发送登录凭证，52
server reply 服务器响应，51
username 用户名，51
automated dissection 自动化研究
bioinformatics 生物信息学，427-431
genetic algorithm 遗传算法，431-435
heuristic-based 基于启发式，421-427
binary 二进制，49-52
block-based representation 基于块的表示，361-362
complex 复杂，40
defined 定义，46
documentation 文档化，421
drivers 驱动，255
elements 元素
block identifiers 块标识符，58
block sizes 块大小，58
checksums 校验和，58
name-value pairs 名称-值对，57
emulating with Dfuz 使用 Dfuz 进行仿真，359
fields 字段，46-48
FTP，48
fuzzing 模糊测试，419-421
hierarchical breakdown 逐层分解，424
HTTP，122
ICMP，429-430
Internet，46
modeling 建模，352
network fuzzers 网络模糊测试器，40，53-54
application layer vulnerabilities 应用层漏洞，230
complex protocols 复杂协议，40
data capture 数据抓取，251
data link layer vulnerabilities 数据链路层漏洞，228
defined 定义，224
fault detection 故障检测，232-233,254
fuzz variables 模糊变量，253
methods 方法，230-231
modified client mutation 修改客户端变异，232
network layer vulnerabilities 网络层漏洞，229
packet assembling 组成数据包，250-251
parsing data 解析数据，251-252
presentation layer vulnerabilities 表示层漏洞，230
protocol drivers 协议驱动，255
protoFuzz 见“protoFuzz”
sending data 发送数据，253
session layer vulnerabilities 会话层漏洞，229
simple protocols 简单协议，40
socket communication component 套接字通信组件，224
targets 目标，226-230
transport layer vulnerabilities 传输层漏洞，229
UNIX systems UNIX 系统，236-237,244-247
Windows systems Windows 系统，见“proroFuzz”

NMAP，236
overview 概述，237-240
SPIKE NMAP fuzzer script SPIKE NMAP 模糊测试脚本，244-247
NNTP，447
open 打开，54
overview 概述，45
plain text 普通文本，48-49
proprietary 私有的，54,421
simple 简单，40
SIP
processing code 处理代码，462
testing 测试，458-461
testing 测试
automatic protocol generation testing 自动协议生成测试，36
manual protocol mutation testing 手工协议变异测试，35
third-party 第三方，421
TLV style syntax TLV 类型语法，352
web applications input web 应用输入，128
ProtoFuzz
case study 案例研究，262-264
data capture 数据抓取，251
design 设计，257,262
data capture 数据抓取，258-259
fuzz variables 模糊变量，261
hexadecimal encoding/decoding 16 进制编码/解码，262
network adapters 网络适配器，257-258
parsing data 解析数据，259-261
disadvantages 缺点，264-265
fuzz variables 模糊变量，253
goals 目的，255
NDIS protocol driver NDIS 协议驱动，255
packets 数据包
assembling 组成，250-251
capture library,choosing 抓取库，选择，256-257
parsing data 解析数据，251-252
programming language 编程语言，256
sending data 发送数据，253
PROTOS,23,458-460,510
ProtoVer Professional ProtoVer 专业版，512
proxy fuzzing 代理模糊测试，422-423
proxyFuzzer，422-424
pseudo-random data generation 伪随机数据生成，353
PStalker
data 数据
capture 抓取，454
exploration 探索，453
sources 来源，452-453
storage 存储，454-457
Gizmo project case study Gizmo 项目案例研究
data sources 数据源，462
execution,monitoring 执行，监视，464-465
feaures list 特性列表，457
results 结果，465
SIP packet,testing SIP 数据包，测试，458-461
SIP processing code SIP 处理代码，462
strategy 策略，458-461

tags,choosing 标签，选择，462
tracking capture options 跟踪抓取选项，462-463
layout overview 布局概述，452-452
limitations 局限，454
ptrace method ptrace 方法，76,100
PTRACE_TRACEMENT requestPTRACE_TRACEMENT 请求，187
pureFuzz，367
PUT method PUT 方法，124
pyDbg class pyDbg 类，332-334
in-memory fuzzer implementation 内存模糊测试器实现，340-345
single stepper tracking tool implementation 单步跟踪工具实现，445-446
python
ctypes module ctypes 模块，318
extensions 扩展，77
interfacing withCOM COM 接口，287
PaiMer framework PaiMei 框架，449
advanced target monitoring 高级目标监视，486-489
basic target monitoring 基础目标监视，482-484
components 组件，450
crash binning utility 崩溃二进制日志工具，487-489
PIDA interface PIDA 接口，450
PStalker module PStalker 模块，见“PStalker”
SWF fuzzing SWF 模糊测试，385
Protocol Informatics(PI) 协议信息学（PI），428
PyDbg class PyDbg 类，332, 334, 332
in-memory fuzzer implementation 内存模糊测试器实现，340-345
single stepper tracking tool 单步跟踪工具
implementation 实现，445-446
PythonWin COM browser PythonWin COM 浏览器，288
PythonWin type library browser PythonWin 类型库浏览器，288

Q

QA Researcher QA 研究者，504

R

race condition vulnerabilities 竞争条件漏洞，178
Raff, Aviv, 24, 42
random fuzzing 随机模糊测试，367
random primitives 随机原始类型，389-390
random testing 随机测试，34
randomize() routine randomize()例程，378
Rational Purify, 492
RATS (Rough Auditing tool for Security), 7-8
RCE (reverse code engineering), binary auditing 逆向代码工程（RCE），二进制审计
automated 自动方式，17-18
manual 手工方式，14-16
reading 读取
addresses 地址，476-478
process memory 进程内存，318

ReadProcessMemory() functionReadProcessMemory()函数，318
RealPlayer
RealPix format string vulnerability RealPix 格式字符串漏洞，193-195
shell script workaround shell 脚本解决方案，192
RealServer ../ DESCRIBEvulnerabilityRealServer ../ DESCRIBE 漏洞，226
ReceivePacket() function RecievePacket()函数，259
recordings (tracking), filtering 记录（跟踪），过滤，446
record_crash() routine record_crash()例程，483, 487
RECT struct RECT 结构，378
Reduced Instruction Set Computer (RISC)architecture 精简指令集计算机（RISC）架构，438
registering callbacks 注册回调，401-402
remote access services vulnerability 远程访问服务漏洞，226
remote code injection 远程代码注入，134
remote fuzzers 远程模糊测试器，39
network protocol 网络协议，40
web application web 应用，41
web browser web 浏览器，41-42
repeaters as block helper 辅助块的重复器 396
reporting exceptions 报告异常，183-184
reproduction, in-memory fuzzing 重现，内存模糊测试，43
request-URI input (web applications) 请求 URI 输入（web 应用），126-128
requests 请求
HTTP, 144
PTRACE_TRACEME, 187
Sulley, 410-411
timeouts 超时，147
Web application fuzzing Web 应用模糊测试，140-141
WebFuzz, 153-155
Requests for Comment (RFCs) 请求评议（RFCs），54
researchers 研究者
QA, 504
security 安全性，504-505
resource constrains 资源限制，69
responses 响应
error messages 错误信息，146-147
user input 用户输入，147
Web application fuzzing Web 应用模糊测试，143
WebFuzz, 155-156
restores 恢复
points 恢复点
fuzz_server.exe, 335
OllyDbg, 338
processes 进程
handling 处理器，327-331
memory block contents, saving 内存块内容，保存，329-330
thread context, saving 线程上下文，保存，328-329
return code 返回码，179
reusability 可复用性，43, 62-64
reverse code engineering (RCE) 逆向代码工

程（RCE），14
reverse engineering frameworks (PaiMei) 逆向工程框架（PaiMei），449-450. 也见“PStlker”
RFCs (Requests for Comment) 请求评议（RFCs），54
RGB struct RGB 结构，378
RISC (Reduce Instruction Set Computer) architecture 精简指令集计算机（RISC）架构，438
Rough Auditing Tool for Security (RATS), 8
routines. 例程。见“functions”，“methods”
ap.getPayload(), 356
av_handler(), 483
bp_set(), 333
ContinueDebugEvent(), 323
DebugActiveProcess(), 321
DebugSetProcessKillOnExit(), 321
flatten(), 378
func_resolve(), 333
GetFuncDesc(), 296
GetNames(), 295
GetThreadContext(), 327
GetTypeInfoCount(), 294
handler_bp(), 333
LoadTypeLib(), 294
MMalloc(), 81
parse(), 306
process_restore(), 345
process_snapshot(), 343
randomize(), 378
record_crash(), 483, 487
s_checksum(), 396
self.push(), 398
setMaxSize(), 356
setMode(), 356
SetThreadContext(), 327
set_callback(), 333
smart(), 378
s_repeat(), 396
sscanf(), 427
s_sizer(), 395
strcpy(), 5
syslog(), 478
Thread32First(), 328
to_binary(), 378
to_decimal(), 378
unmarshal(), 306
VirtualQueryEx(), 329
write_process_memory(), 344
xmlComponentString(), 407
Royce, Winston, see waterfall model Royce，Winston，见“瀑布模型”
RPC DCOM Buffer Overflow vulnerabilityRPC DCOM 缓冲区溢出漏洞，226
RPC-based services vulnerabilities 基于 RPC 的服务漏洞，226
rule file (DFuz) 规则文件（DFuz），359-360

S

Samba, 420
SAP Web Application Server sap-exiturl Header HTTP Response Splitting website SAP Web 应用服务器 sap-exiturl 头 HTTP 响应分解网站，117
saving 保存
audits 审计，204-205

memory block contents 内存块内容，329-330
original bytes at breakpoint address 断点地址处原先的字节，324
test case metadata 测试用例元数据，179
thread context 线程上下文，328-329
s_block_end() function s_block_end()函数，242, 392
s_block_start() function s_block_start()函数，242, 392
s_checksum() routine s_checksum()例程，396
scripts 脚本
client-side scripting vulnerabilities 客户端脚本漏洞，276
protocol-specific fuzz 面向协议的模糊测试，244
SPIKE NMAP fuzzer SPIKE NMAP 模糊测试器，244-248
XSS scripting case study XSS 脚本案例研究，163-166
SDL (Security Development LifeCycle) 安全开发生命周期（SDL），498
SDLC (software development lifecycle) 软件开发生命周去（SDLC），69
Microsoft SDL 微软安全开发生命周期，498
security 安全性，69
waterfall model 瀑布模型，499
analysis 分析，500
coding 编码，501-502
design 设计，500-501
maintenance 维护，502-503
testing 测试，502
secure shell (SSH) servers 安全 shell（SSH）服务器，64
security 安全性
development lifecycle 开发生命周期，498
Microsoft 微软，499
researcher 研究者，504-505
software development lifecycle. see SDLC 软件开发生命周期，见“SDLC”
zone vulnerabilities 区域漏洞，281
SecurityReview 安全性评审，18
SEH (Structured Exception Handler) 结构化异常处理器（SEH），484
self.push() routine self.push()例程，398
sequences, aligning 序列，对齐，427
servers 服务器
data capture example 数据抓取示例，335
launching 启动，334
media vulnerability 媒体漏洞，226
Microsoft worms vulnerabilities 微软蠕虫漏洞，224-226
sessions 会话
layer vulnerabilities 会话层漏洞，229
Sulley framework Sulley 框架，399
callbacks, registering 回调，注册，401-402
creating 创建，411-413
instantiating 实例化，401
linking requests into graphs 将请求链接到图形，399-401
targets and agents 目标与代理，402-404
Web monitoring interface Web 监视界面，404
setgid bits setgid 位，93

setMaxSize() routine setMaxSize()例程，356
setMode() routine setMode()例程，356
SetThreadContext() API SetThread Context() API，327
setuid applications setuid 应用，37
setuid, bits setuid，位，93
set_callback() routine set_callback()例程，333
SGML (Standardized General Markup Language) 标准化通用标记语言（SGML），273
Sharefuzz, 24, 38
Shockwave Flash 见“SWF”
Sidewinder (GAs) SideWinder（遗传算法），433-434
SIGABRT signal SIGABRT 信号，188
SIGALRM signal SIGALRM 信号，188
SIGBUS signal SIGBUS 信号，188
SIGCHLD signal SIGCHLD 信号，188
SIGFPE signal SIGFPE 信号，188
SIGILL signal SIGILL 信号，188
SIGKILL signal SIGKILL 信号，188
signals 信号
handlers 处理器，31
UNIX, 187-188
SIGSEGV signal SIGSEGV 信号，31, 188
SIGSYS signal SIGSYS 信号，188
SIGTERM signal SIGTERM 信号，188
simple protocol 简单协议，40
simple stack vulnerabilities 简单的栈漏洞，177
Simple Web Server buffer overflow Simple Web Server 中的缓冲区溢出，159
single-option modules (iFuzz) 单选项模块（iFuzz），104, 107
SIP
processing code 处理代码，462
testing 测试，458-461
SIPPhoneAPI library SIPPhoneAPI 库，462
sizers as block helpers 辅助块的大小计算器，395-396
Slammer worm Slammer 蠕虫，225
smart data sets 智能数据集，81
smart() routine smart()例程，378
Smith-Waterman (SW) local sequence alignment algorithm SW 本地序列对齐算法，428
snapshots 快照
fuzz_server.exe, 343
points 快照点，335
processes 进程
handling 处理，328-331
memory block contents, saving 内存块内容，保存，329-330
thread context, saving 线程上下文，保存，328-329
restoration mutation, (SRM) 恢复变异（SRM），309-310, 316
socket communication 套接字通信，224
software 软件
breakpoints 断点，324
development lifecycles 开发生命周期，见“SDLC”
memory corruption vulnerabilities 内存破坏漏洞，475
execution control, transferring 执行控制，转移，475-476
reading addresses 读取地址，476-478
writing to addresses 写入地址，478-479

source code 源代码
browsers 浏览器，7
white box testing analysis 白盒测试分析，4-9
SPI Dynamics, 41
SPI Dynamics Free Bank application vulnerabilitySPI Dynamics Free Bank 应用漏洞，163-164
SPI Fuzzer, 41, 138
SPIKE, 23, 40, 361-364
block-based protocol 基于块的协议
modeling 建模，242
representation 表示，361-362
disadvantage 不足，363
FTP fuzzer FTP 模糊测试器，362-363
fuzz engine 模糊测试引擎，240
generic line-based TCP fuzzer 基于行的通用 TCP 模糊测试器，240-243
generic script-based fuzzer 基于脚本的通用模糊测试器，244
protocol-specific fuzzers 面向协议的模糊测试器，243-244
Proxy 代理，138
UNIX fuzzing UNIX 模糊测试，236
SPIKE NMAP fuzzer script SPIKE NMAP 模糊测试器脚本，244-248
targets 目标，236-237
SPIKEfile
core fuzzing engine 核心模糊测试引擎，184-185
exceptions 异常
detection engine 检测引擎，183
reporting 报告，183-184
features 特性，182
forking off/tracing child processes 创建/跟踪子进程，185-187
interesting UNIX signals 我们感兴趣的 UNIX 信号，187
missing features 缺少的特性，182
not interesting UNIX signals 我们不感兴趣的 UNIX 信号，188
programming language 编程语言，195
shell script workaround shell 脚本解决方案，192
zombie processes 僵尸进程，189-191
Splint website Splint 网站，7
SQL injections SQL 注入
case study (Web applications) 案例研究（Web 应用），160-162
vulnerabilities 漏洞，132
s_repeat() routine s_repeat()例程，296
SRM (snapshot restoration mutation) 快照恢复变异（SRM），309-310, 316
sscanf() routine sscanf()例程，427
SSH (secure shell) servers 安全 shell（SSH）服务器，64
s_sizer() routines s_sizer()例程，395
stacks 栈
fault detection example 故障检测示例，477
overflow vulnerabilities 溢出漏洞
Hewlett-Packard Mercury LoadRunner HP Mercury LoadRunner，263
NMAP protocol NMAP 协议，246
unwinding 展开，485
Standardized General Markup Language (SGML) 标准化通用标记语言，273

start points (basic blocks) 起始点（基本块），440
state (process) 状态（进程），64-66
static primitives 静态原始类型，389-390
stepping 步进，473
stop points (basic blocks) 终止点（基本块），440
storage (data) 存储（数据），455-457
strcpy() routine strcpy()例程，5
strings 字符串
format 格式，85
file format vulnerabilities 文件格式漏洞，177
RealPlay RealPix format string vulnerability RealPlay RealPix 格式字符串漏洞，193-195
vulnerabilities 漏洞，85, 477
primitives 原始类型，383
repetitions 重复，82
Sulley framework Sulley 框架，391
Structured Exception Handler (SEH) 结构化异常（SEH）处理器，484
su application example su 应用示例，92
Sulley framework Sulley 框架，391
block helpers 块辅助函数，395
checksums 校验和，396
example 示例，397
repeaters 重复器，396
sizers 计算尺寸器，395-396
blocks 块，392
dependencies 依赖，394-395
encoders 编码器，394
grouping 分组，392-393
data types 数据类型，388-390
delimiters 分隔符，391
directory structure 目录结构，387-388
download website 下载网站，386
environment, setting up 环境，设置，413
features 特性，386
fuzzers, launching 模糊测试器，启动，414-416
integers 整数，390-391
legos 积木，398-399
postmortem phase 事后分析阶段，405-409
requests, building 请求，构建，410-411
RPC endpoint walkthrough RPC 端点研究，409-410
environment, setting up 环境，设置，413
launching 启动，414-416
requests, building 请求，构建，410-411
sessions, creating 会话，创建，411-413
sessions 会话，399
callbacks, registering 回调，注册，401-402
creating 创建，411-413
instantiating 实例化，401
linking requests into graphs 将请求链接到图形，399-401
targets and agents 目标和代理，402-404
Web monitoring interface Web 监视界面，404
strings 字符串，391
SuperGPF, 368
Sutton, Michael, 39
SW (Smith-Waterman) local sequence alignment algorithm SW 本地序列对齐算法，428

sweeping applications 扫描应用，10
SWF (Shockwave Flash), 372
bit_field class bit_field 类，378
component relationships 组件关系，383
data 数据
generation 生成，384
structure 结构，374-377
dependent_bit_field class dependent_bit_field 类，379
environment 环境，385
header 头，372-373
MATRIX struct MATRIX 结构，379-380
methods 方法，385
RECT/RGB struct RECT/RGB 结构，378
string primitives 字符串原始类型，383
SWF files, modeling SWF 文件，建模，374
tags 标签，374, 380-382
syslog() routine syslog()例程，478

T

taboo() function taboo()函数，475
tag vulnerabilities 标签漏洞
HTML, 271-273
XML, 273
targets 目标
audiences for fuzzing 模糊测试审计，503-504
debugger-assisted monitoring 调试器辅助的监视，481
advanced example 高级示例，486-489
architecture 架构，481
basic example 基础示例，482-484
PaiMei crash binning utility PaiMei 崩溃二进制日志工具，487-489
file format fuzzing 文件格式模糊测试，170-171
identifying 识别，27
in-memory fuzzing 内存模糊测试，307-308
local fuzzing 本地模糊测试，93-95
network protocol fuzzing 网络协议模糊测试，226-230
application layer 应用层，230
categories 分类，226
data link layer 数据链路层，228
network layer 网络层，229
presentation layer 表示层，230
session layer 会话层，229
transport layer 传输层，229
profiling 剖析，443-444
setuid applications setuid 应用，37
Sulley sessions Sulley 会话，402-404
UNIX, 236-237
web application fuzzing web 应用模糊测试
environment, configuring 环境，配置，118-119
examples 示例，117-118
inputs. see web application fuzzing, inputs 输入，见“web 应用模糊测试”之“输入”
web browsers web 浏览器，269
Windows file formats Windows 文件格式
identifying 识别，205-209
Windows Explorer Windows 资源管理器，206-209

Windows Registry Windows 注册表，209
TCP/IP vulnerabilities TCP/IP 漏洞，229
TcpClient class TcpClient 类，148-149
TEBs (thread environment blocks) 线程环境块（TEBs），485
test cases 测试用例
connectivity checks 连接检查，472
metadata, saving 元数据，保存，179
testing 测试
black box 黑盒，9
fuzzing 模糊测试，12
manual 手工 10
pros/cons 优点/缺点，13-14
gray box 灰盒，14
binary auditing 二进制审计，14-18
pros/cons 优点/缺点，18
SDLC, 502
white box 白盒
pros/cons 优点/缺点，9
source code analysis 源代码分析，4-5
tools 工具，6-8
third-party protocols 第三方协议，421
threads 线程
context, saving 上下文，保存，328-329
environment blocks (TEBs) 环境块（TEBs），485
Thread32First() API, 328
Tikiwiki tiki-user_preference Command Injection Vulnerability website Tikiwiki tiki-user_preference 命令行注入漏洞网站，117
TLV (Type, Length, Value) style syntax TLV（类别，长度，值）类型语法，352
tools. see also debuggers 工具，见“调试器”
Autodafe[as], 369-371
AxMan, 275
beSTORM, 138, 508
BinAudit, 18
BoundsChecker, 493
BreakingPoint, 509
BugScam, 17
clfuzz, 38
Codenomicon, 510-511
foundation 基础，23
HTTP test tools, 41 HTTP 测试工具，138
COM Raider, 25, 42, 275
Convert, 367
crash binning, 487-489
crashbin_explorer.py, 405
CSSDIE, 276, 275, 42
DevInspect, 504
Dfuz, 357-360
data sources 数据源，358
functions 函数，357-358
lists, declaring 列表，声明，358
protocols, emulating 协议，仿真，359
rule files 规则文件，359-360
variables,defining 变量，定义，357
DOM-Hanoi,42
FileFuzz,39
applications,launching 应用，启动，203
ASCII text files ASCII 文本文件，202
audits,saving 审计，保存，204-205

benefits 好处，221
binary files 二进制文件，201
case study 案例研究，217-220
development.See FileFuzz development，开发。见“FileFuzz”之“开发”
error detection 错误检测，203-204
features 特性，201
future improvements 将来的改进，221
goals 目的，200
fuzz_trend-server-protect-5268.py,414-416
GPF,366-369
Hamachi,42
Holodeck,514-515
iFuzz,38
case study 案例研究，110-111
development approach 开发方法，105-107
features 特性，103-105
fork,execute,and wait approach fork, execute 和 wait 方法，107
fork,ptrace/execute,and wait/ptrace approach fork, ptrace/execeute 和 wait/ptrace 方法，108-109
getopt hooker getopt 钩子，105
language 编程语言，109
modules 模块，104-107
postdevelopment observations 开发后的观察，111-112
Inspector,18,312
LogiScan,17
Mangleme,42
Mu-4000,512
Netcat,49
notSPIKEfile,39
core fuzzing engine 核心模糊测试引擎，184-185
exception handling 异常处理，183-184
features 特性，182
forking off/tracing child processes 创建/跟踪子进程，185-187
interesting UNIX signals 我们感兴趣的 UNIX 信号，187
missing features 缺少的特性，182
not interesting UNIX signals 我们不感兴趣的 UNIX 信号，188
programming language 编程语言，195
RealPix format string vulnerabilityReal Pix 格式字符串漏洞，193-195
Shell script workarounds Shell 脚本解决方案，192
Zombie processes 僵尸进程，189-191
PAIMEfilefuzz,39
Pattern Fuzz,368
PDML2AD,371
Peachi,40,364-366
code reusability 代码可复用性，366
disadvantages 缺点，366
generators 生成器，364
groups 组，365
publishers 发布者，364
transformers 变化者，364
Process Stalker.See Pstalker Stalker 进程，见“Pstalker”
ProtoFuzz
case study 案例研究，262-264

data capture 数据抓取，251,258-259
disadvantages 缺点，264-265
fuzz variables 模糊变量，253,261
goals 目的，255
hexadecimal encoding/decoding 16进制编码/解码，262
NDIS protocol driver NDIS 协议驱动，255
network adapters 网络适配器，257-258
packet capture library, choosing 数据包抓取库，选择，256-257
packets,assembling 数据包，组包，250-251
parsing data 解析数据，259-261
programming language 编程语言，256
sending data 发送数据，253
PROTOS, 458-460,510
ProtoVer Professional ProtoVer 专业版，512
ProxyFuzzer,422-424
Pstalker
data capture 数据抓取，454
data explotation 数据探索，453
data sources 数据源，452-453
data storage 数据存储，455-457
Gizmo Project case study. See Gizmo Project case study Gizmo 项目案例研究，见“Gizmo 项目案例研究”
layout overview 布局概述，451-452
limitations 局限，454
ptrace(),76
PureFuzz,367
Python extensions Python 扩展，77
Rational Purify,492
SecurityReview,18
Sharefuzz,38
Sidewinder,433-434
Source code 源代码，7
SPI Fuzzer,41,138
SPIKE,361,362,40,364
block-based protocol modeling 基于块的协议建模，242
block-based protocol representation 基于块的协议表示，361-362
disadvantage 缺点，363
FTP fuzzer FTP 模糊测试器，362-363
fuzz engine 模糊测试引擎，240
generic line-based TCP fuzzer 基于行的通用 TCP 模糊测试器，240-243
generic script-based fuzzer 基于脚本的通用模糊测试器，244
protocol-specific fuzzers 面向协议的模糊测试器，243
Proxy 代理，138
UNIX fuzzing UNIX 模糊测试，236-237,244-248
SPIKE Proxy,138
SPIKEfile
core fuzzing engine 核心模糊测试引擎，184-185
exception detection engine 异常检测引擎，183
exception reporting 异常报告，183-184

features 特性，182
forking off/tracing child processes 创建/跟踪子进程，185-187
interesting UNIX signals 我们感兴趣的 UNIX 信号，187
missing features 缺少的特性，182
not interesting UNIX signals 我们不感兴趣的 UNIX 信号，188
programming language 编程语言，195
shell script workarounds shell 脚本解决方案，192
zombie processes 僵尸进程，189-191
Sulley framework,387,386
block helpers 块辅助函数，395-397
blocks 块，392-395
data types 数据类型，388-390
delimiters 分隔符，391
directory structure 目录结构，387-388
download website 下载站点，386
environment,setting up 环境，设置，413
features 特性，386
fuzzers,launching 模糊测试器，启动，414-416
integers 整数，390-391
legos 积木，398-399
postmortem phase 事后分析阶段，405-409
requests,building 请求，构建，410-411
RPC endpoint walkthrough RPC 端点分析，409-416
sessions.See Sulley framework sessions，会话，见“Sulley 框架”之“会话”
strings 字符串，391
SuperGPF,368
tracking 跟踪
benefits 好处，466-467
binary visualization 二进制可视化，439-441
future improvements 将来的改进，467-469
as metrics 作为度量工具使用，66
Outlook Express NNTP vulnerabilityOutlook Express NNTP 漏洞，447-449
overview 概述，437-439
PaiMei framework PaiMei 框架，449-450
Pstalker.SeePStalker Pstalker，见“PStalker”
PyDbg single stepper tool PyDbg 单步工具，445-446
tool development 工具开发，442-447
TXT2AD,371
Valgrind,493
WebFuzz
approach 方法，148
asynchronous sockets 异步套接字，150-153
benefits 好处，166
buffer overflow case study 缓冲区溢出案例分析，158-160
directory traversal case study 目录遍历案例研究，156-157
error messages 错误信息，146-147

future improvements 将来的改进，166
fuzz variables 模糊变量，142
generating requests 生成请求，153-155
handled/unhandled exceptions 已处理的/未处理的异常，147
HTML status codes HTML 状态码，146
HTTP requests,identifying HTTP 请求，识别，144
performance degradation 性能下降，147
programming language,choosing 编程语言，选择，148
request timeouts 请求超时，147
requests 请求，140-147
responses 响应，143,155-156
SQL injection case study SQL注入案例研究，160-162
TcpClient class TcpClient 类，148-149
user input 用户输入，147
vulnerabilities,identifying 漏洞，识别，145-147
XSS scripting case study XSS 跨站脚本案例研究，163-166
WebScarab,41,131,138
white box testing 白盒测试，6-8
Wireshark,75
to_binary()routine to_binary()例程，378
to_decimal()routine to_decimal()例程，378
TRACE method TRACE 方法，124
tracing 跟踪
child processes 子进程，185-187
instruction execution 指令执行，444-445
tracking 跟踪
benefits 好处，466-467
binary visualization 二进制可视化，439
call graphs 调用图，439
CFGs,440-441
future improvements 将来的改进，467-469
as metrics 作为度量工具，66
Outlook Express NNTP vulnerabilityOutlook Express NNTP 漏洞，447-449
overview 概述，437-439
PaiMei framework PaiMei 框架，449-450
PStalker tool PStalker 工具
data capture 数据抓取，454
data exploration 数据探索，453
data sources 数据源，452-453
data storage 数据存储，454-457
Gizmo Project case study.See Gizmo Project casestudy Gizmo 项目案例研究，见“Gizmo 项目研究”
layout overview 布局概述，451-452
limitations 局限，454
PyDbg single stepper tool PyDbg 单步工具，445-446
tool development 工具开发，442
basic blocks,tracking 基本块，跟踪，444
cross-referencing 交叉引用，447
instruction execution,tracing 指令执行，跟踪，444-445
recordings,filtering 记录，过滤，446
target profiling 目标剖析，443-444
traffic generation 流量生成，509
transformers 变化者，364

transport layer vulnerabilities 传输层漏洞，229
Trend Micro Control Manager directory traversal Trend Micro Control Manager 目录遍历，156
Tridgell,Andrew,420
Trustworthy Computing Security Development Lifecycle 可信计算安全开发生命周期
document (Microsoft) 文档（微软），13
TXT2AD,371
Type,length,Value(TLV)style syntax 类别，长度，值（TLV）类型语法，352

U

UNIX
file format fuzzing.See SPIKEfile;notSPIKEfile 文件格式模糊测试，见“SPIKEfile”和“NotSPIKEfile”
file permissions 文件权限，95
interesting/not interesting signals 我们感兴趣/不感兴趣的信号，187-188
targets 目标，236-237
unmarshal()routine unmarshall()例程，306
unwinding stacks 栈展开，485
UPGMA(Unweighted Pairwise Mean by ArithmeticAverages) algorithm UPGMA（无加权成对数据平均数）算法，429
URL vulnerabilities URL 漏洞，280
User-Agent header User-Agent 头，122
utilities.See tools 工具，见“工具”

V

Valgrind,493
values(fuzz) 值（模糊值），480-481
variable length fields(protocols) 可变长字段（协议），48
variables 变量
defining 定义，357
environment 环境，38-39,90-92
GDB method GDB 方法，96-97
getenv function getenv 函数，98
library preloading 库预加载，98-99
network protocols 网络协议，253
protoFuzz design protoFuzz 设计，261
Web application fuzzing Web 应用模糊测试，142
VARIANT data structure VIARANT 数据结构，293
Vector Markup Language(VML) 向量标记语言（VML），273
viewing 查看
crash locations 崩溃位置，405-406
fault detections 故障检测，406
VirtualQueryEx()routine VirtualQueryEx() 例程，329
VML(Vector Markup Language) VML（向量标记语言），273
VMs(virtual machines) VMs（虚拟机），119
VMWare control agent VMWare 控制代理，403
vulnerabilities 漏洞
ActiveX controls ActiveX 控件，273-275
address bar spoofing 地址栏欺骗，281
Apple Macbook,228
application layer 应用层，230
Brightstore backup software Brightstore 备

份软件，424
buffer overflows 缓冲区溢出，130,281
client-side 客户端，223,280-282
client-side scripting 客户端脚本，276
commands 命令
execution 执行，281
injection 注入，87
cross-domain restriction bypass 规避跨域限制，281
CSS,275-276
data link layer 数据链路层，228
data session link layer 数据会话链路层，229
directory traversal 目录遍历，86
discussion boards 讨论板，117
DoS,280
ERP,117
Excel eBay,199
file formats 文件格式
DoS,175
examples 示例，170
format strings 格式字符串，177
heap overflows 堆溢出，177
integer handing 整数处理，175-177
logic errors 逻辑错误，177
race conditions 竞争条件，178
simple stack 简单栈，177
Flash,279
format strings 格式字符串，85,106,477
GAs,433
GDI+buffer overflow GDI+缓冲区溢出，198,217-220
heap overflow 堆溢出，129
HTML
headers 头，271
tags 标签，271-273
hybrid analysis approach 混合分析方法，515
Ipswitch I-Mail,420
libraries 库，134
log analysis 日志分析，118
media servers 媒体服务器，226
Microsoft source code leak 微软源代码泄漏，5
network 网络
layer 层，229
monitoring 监视，118
target categories 目标类别，226
NMAP stack overflow NMAP 栈溢出，246
Outlook Express NNTP,447-449
phishing 钓鱼，281
PNG,199
precompile security solution 预编译安全性方案，480
presentation layer 表示层，230
RealPlayer RealPix format string RealPlay RealPix 格式字符串，193-195
remote access services 远程访问服务，226
RPC-based services 基于 RPC 的服务，226
security zones 安全性区域，281
sever-side 服务端，223-226
software memory corruption 软件内存破坏，475
execution control,transferring 执行控制，转移，475-475
reading addresses 读取地址，476-478

writing to addresses 写入地址，478-479
SPI Dynamics Free Bank application SPI Dynamics Free Bank 应用，163-164
stack overflow 栈溢出
Hewlett-Packard Mercury LoadRunner, HP Mercury LoadRunner，263
NMAP protocol NMAP 协议，246
TCP/IP,229
transport layer 传输层，229
URLs,280
Web Maill,117
weblogs,117
Wikis,117
Windows file formatting Windows 文件格式化，197-198
winnuke attack winnuke 攻击，229
WinZip FileView,298
WMF,199
XML tags XML 标签，273

W

Wterfall model 瀑布模型，499
Analysis 分析，500
coding 编码，501-502
design 设计，500-501
maintenance 维护，502-503
testing 测试，502
weak access control 弱访问控制，132
weak authentication 弱认证，133
weak session management 弱会话管理，133
web application fuzzing web 应用模糊测试，41,138
benefits 好处，166
beSTORM,138,508
buffer overflows 缓冲区溢出，130,158-160
Codenomicon,138
configuring 配置，118-119
development 开发
approach 方法，148
asynchronous sockets 异步套接字，150-153
programming language,choosing 编程语言，选择，148
requests,generating 请求，生成，153-155
responses,receiving 响应，接收，155-156
TcpClient class TcpClient 类，148-149
directory traversal case study 目录遍历案例研究，156-157
Ipswitch Imail Web Calendaring,157
Trend Micro Control Manager,156
error messages 错误信息，146-147
exceptions,detecting 异常，检测，135-136
future improvements 将来的改进，166
fuzz variables 模糊变量，142
handled/unhandled exceptions 已处理的/未处理的异常，147
heap overflow vulnerability 堆溢出漏洞，129
HTML status codes HTML 状态码，146
HTTP requests,identifying HTTP 请求，识别，144
inputs 输入
choosing 选择，119-121
cookies,129
headers 头，128
identfying 识别，130-131

method 方法，123-125
post data post 数据，130
protocol 协议，128
requests-URI 请求 URI，126-128
overview 概述，113-115
performance degradation 性能下降，147
request timeouts 请求超时，147
requests 请求，140-141
responses 响应，143
SPI Fuzzer SPI 模糊测试器，138
SPIKE Proxy SPIKE 代理，138
SQL injection case study SQL 注入案例研究，160-162
targets 目标，117-118
technologies 技术，115
user input 用户输入，147
vulnerabilities 漏洞，132-135,145-147
WebScarab,138
XSS cripting case study XSS 跨站脚本案例研究，163-166
web browser fuzzing web 浏览器模糊测试，41-42
ActiveX,287-289
heuristics 启发式，298
loadable controls,enumerating 可加载控件，枚举，289-293
monitoring 监视，299
properties,methods,paramenters,and types 属性，方法，参数与类型，294-297
test cases 测试用例，298
approaches 方法，269-271
fault detection 故障检测，282
heap overflows 堆溢出，277-279
history 历史，24
inputs 输入，271
ActiveX controls ActiveX 控件，273-275
client-side scripting 客户端脚本，276
CSS,275-276
Flash,279
HTML headers HTML 头，271
HTML tags HTML 标签，271-273
URLs,280
Month of Browser Bugs 浏览器缺陷月，268
overview 概述，268
targets 目标，269
vulnerabilities 漏洞，280-282
Web Mail vulnerabilities Web Mail 漏洞，117
Web monitoring interface(Sulley) Web 监视界面（Sulley），404
web server error messages web 服务器错误信息，135
web sites 网站
AWStats Remote Command Execution Vulnerability AWStats 远程命令执行漏洞，118
CodeSpy,7
Flawfinder,7
ITS4,7
Jlint,7
Splint,7
Sulley download 下载 Sulley，386
Vulnerabilities 漏洞
IpSwitch Whats Up Professional 2005(SP1)SQL Injection IpSwitch Whats Up 专业版 2005（SP1）SQL 注入，118
Microsoft Outlook Web Access Cross Site Scripting 微软 Outlook Web 访问跨站脚本，117

Multiple Vendor Cacti Remote File Inclusion 影响多个供应商的 Cacti 远程文件包含，118

OpenSSH Remote Challenge OpenSSH 远程挑战，226

phpBB Group phpBB Arbitrary File Disclosure phpBB 组 PhpBB 任意文件暴露，117

RATS download RATS 下载，7

RealServer../DESCRIBE,226

RPC DCOM Buffer Overflow RPC DCOM 缓冲区溢出，226

SAP Web Application Server sap-exiturl Header SAP Web 应用服务器 sap-exiturl 头

HTTP Response Splitting HTTP 响应分解，117

Tikiwiki tiki-user_preferences CommandInjection Tikiwiki tiki-user_preferences 命令注入，117

WinZip MIME Parsing Buffer OverflowAdvisory WinZip MIME 解析缓冲区溢出报告，170

WordPress Cookie cache-lastpostdate VariableArbitrary PHP Code Execution WordPress Cookie cache-lastpostdate 可变任意 PHP 代码执行，117

Wireshark,335

Wotsit,421

WebFuzz

benefits 好处，166

buffer overflow case study 缓冲区溢出案例分析，158-160

development 开发

approach 方法，148

asynchronous sockets 异步套接字，158-160

programming language,choosing 编程语言，选择，148

requests,generating 请求，生成，153-155

responses,receiving 响应，接收，155-156

TcpClient class TcpClient 类，148-149

directory traversal case study 目录遍历案例研究，156-157

error messages 错误信息，146-147

future improvments 将来的改进，166

fuzz variables 模糊变量，142

handled/unhandled exceptions 已处理的/未处理的异常，147

HTML status codes HTML 状态码，146

HTTP requests,identifying HTTP 请求，识别，144

performance degradation 性能下降，147

request timeouts 请求超时，147

responses 响应，143

SQL injection case study SQL 注入案例研究，160-162

user input 用户输入，147

vulnerabilities,identifying 漏洞，识别，145-147

XSS scripting case study 跨站脚本案例研究，163-166

weblog vulnerabilities weblog 漏洞，117

WebScarab,41,131,138

White box testing 白盒测试

pros/cons 优点/缺点，9

source code analysis 源代码分析，4-5

tools 工具，6-8

Wikis vulnerabilities Wikis 漏洞，117

WinDbg,16
Windows
debugging API 调试 API，320-323,332-333
Explorer file format targets 探索文件格式目标，206,209
file format fuzzing 文件格式模糊测试，205-209
File format vulnerabilities,197-198.See also FileFuzz 文件格式漏洞，197-198，见“FileFuzz”
Live,114
memory model 内存模型，302-303,306
Meta File(WMF)vulnerability 元文件（WMF）漏洞，199
Registry,file format targets 注册表，文件格式目标，209
winnuke attack winnuke 攻击，229
WinPcap library WinPcap 库，256
WINRAR,174
WinZip vulnerabilities WinZip 漏洞
FileView ActiveX Control Unsafe Method Exposure FileView ActiveX 控件不安全的方法暴露，298
MIME Parsing Buffer Overflow MIME 解析缓冲区溢出，170
Wireshark,75
sniffer 嗅探，237
web site 网站，335
WMF(Windows Meta File)vulnerability WMF（Windows 元文件）漏洞，199
WordPress Cookie cache_lastpostdate Variable Arbitrary PHP Code Execution web site WordP ress Cookie cache_lastpostdate 可变任意 PHP 代码执行网站，117
worms(Microsoft) 蠕虫（微软），224-226
Wotsit web site Wotsit 网站，421
WriteProcessMemory()functionWriteProcessMemory()函数，319
write_process_memory()routinewrite_process_memory()例程，344
writing 写入
addresses 地址，478-479
INT3 opcode INT3 操作码，324
process memory 进程内存空间，319
WS2_32.dll recv()breakpoint WS2_32. dll recv()断点，336

X

XDR(External Data Representation) XDR（外部数据表示），230
XML tag vulnerabilities XML 标签漏洞，273
xmlComposeString()routinexmlComposeString()例程，407
XSS(cross-site scripting) XSS（跨站脚本），132,163-166

Y-Z

Zalewski,Michal,24
Zimmer,Dacid,25,42
Zoller,Thierry,24
zombie processes 僵尸进程，189-191