模糊测试

强制性安全漏洞发掘

FUZZING Brute Force Vulnerability Discovery



(美) Michael Sutton
Adam Greene
Pedram Amini
著

黄陇 于莉莉 李虎 译

机械工业出版社
China Machine Press

掌握揭露安全性缺陷的最强大技术！

模糊测试现在已经发展成为一种最有效的软件安全性测试方法。模糊测试是指将一个随机的数据源作为程序的输入，然后系统地找出这些输入所引起的程序失效。著名的模糊测试专家将告诉你如何抢在别人之前使用模糊测试来揭示软件的弱点。

本书是第一部也是唯一一部自始至终讨论模糊测试的专著，将以往非正式的技巧转变为训练有素的最佳实践，进而将其总结为一种技术。作者首先回顾了模糊测试的工作原理并勾勒出模糊测试相比其他安全性测试方法的关键优势。然后，介绍了在查找网络协议、文件格式及Web应用安全漏洞中的先进的模糊测试，演示了自动模糊工具的用法，并给出多个说明模糊测试强大效力的历史案例。

本书主要内容包括：

- 为什么模糊测试能够简化测试设计并捕捉利用其他方法捕捉不到的软件缺陷。
- 模糊测试过程：从识别输入到评估“可利用性”。
- 理解实施有效模糊测试所要满足的需求。
- 比较基于变异的和基于生成的模糊器。
- 在模糊测试中应用并初始化环境变量和自变量。
- 掌握内存数据的模糊测试技术。
- 构建定制的模糊测试框架和工具。
- 实现智能的故障检测。

攻击者早已经开始使用模糊测试技术。当然，你也应该使用。不论你是一位开发者、一位安全工程师还是测试人员或QA专业人员，本书都将教会你如何构建安全的软件系统。

作者简介：

Michael Sutton是SPI Dynamics公司的安全布道师。他还是Web应用安全组织(WASC)的成员，负责其中的Web应用安全统计项目。

Adam Greene目前担任纽约某大型金融新闻公司的工程师。此前他曾经是iDefense公司的工程师，这是位于Reston, VA.的一家智能技术公司。Adam Greene在计算机安全领域的主要研究兴趣是可靠挖掘方法、模糊测试和基于UNIX系统的审核和挖掘开发。

Pedram Amini是TippingPoint公司的安全研究和产品安全评估组的项目领导。此前他曾经是iDefence实验室的主任助手，同时也是该实验室的创建者之一。他的主要兴趣是研究逆向工程——开发自动支持工具、插件和脚本。

这三位作者经常出席Black Hat安全大会并在其中做主题报告。

www.Pearsonhighered.com

投稿热线：(010) 88379511
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

上架指导：计算机/程序设计
ISBN 978-7-111-25755-4
9 787111 257554
定价：59.00元

译者序

模糊测试的基本思想是自动产生和发送大量随机的或经过变异的输入值给软件，如果发生失效或异常，便可挖掘出软件系统存在的薄弱环节和安全漏洞。这种方法由威斯康星州麦迪逊大学的Barton Miller教授首先发明，后来发展成为一种对软件质量有深远影响的测试技术。近年来，模糊测试方法及其支持工具受到研究人员和工程技术人员的日益关注，逐步成为软件测试和系统安全研究领域的一个重要分支。

本书是迄今为止有关模糊测试的第一部专著性参考文献。书中首次系统地阐述了模糊测试的基本概念、分类和实现技术，总结出模糊测试和其他安全性测试方法相比所具有的优点。本书的内容全面丰富，堪称模糊测试领域的“百科全书”，内容涉及模糊测试的定义、方法、分类和不同操作系统平台下模糊器的应用和开发技术，此外还穿插了许多典型历史案例以说明模糊测试的强大威力。对于从事软件测试、网络安全和信息安全领域的研究和工程人员来说，本书具有极高的参考价值。

译者所在单位北京航空航天大学软件测评实验室（附属于北航软件工程研究所）、总参陆航研究所、二炮软件测试中心等长期从事软件测试、质量保证方法及支持工具的研究开发。作为具有全军装备软件测评资质的军用软件测评实验室，北航软件测评实验室和二炮软件测试中心承担了大量关键软件的第三方测评任务，在软件安全漏洞的分析和发掘方面积累了一定的学术成果和工程经验，为了促进国内同行在这一领域的学习和交流，特组织翻译此书。

本书的主要内容由黄陇、于莉莉翻译，李虎完成了部分翻译工作并统校全书。参加本书技术校对的还有李晓丽、许福、宋淼、刘辉、王晓博、贾荣飞等。机械工业出版社华章公司的编辑为本书付出了大量辛勤努力，在此向他们表示诚挚的感谢！

由于译者水平有限，难免存在疏漏和错译之处，欢迎广大读者批评指正。

译者简介

黄陇，男，博士，中国人民解放军总参陆航研究所高级工程师，北京航空航天大学软件工程研究所出站博士后，在国内重要期刊和国际会议上发表论文20余篇，曾获得全军科技进步奖，长期从事软件测试理论、方法和技术工具的研究开发，出版多部该领域的译著。

于莉莉，女，中国人民解放军第二炮兵软件测试中心资深软件测评工程师，北京航空航天大学软件工程研究所博士研究生，自2005年起先后负责十余项大型分布式军事系统软件测试项目，特别是在安全性测试领域积累了丰富的研究和工程经验。

李虎，男，博士，北京航空航天大学计算机学院讲师，北航软件测评实验室测评工程部负责人，近年来先后发表论文20余篇，其中大多被EI等著名检索机构收录，多个著名计算机科学类杂志的审稿人，主持包括国家自然科学基金在内的多项国家级科研项目，曾获国防科学技术三等奖，申请国家技术发明专利2件，获得发明专利1件，在软件工程、软件测试和质量保证领域出版专译著十余部。

序　言

安全漏洞是研究安全问题的生命线。无论是执行渗透测试、评价新产品还是审核关键构件的源代码，安全漏洞都驱动着我们的决策，让我们有理由花费时间，并且很多年来一直影响着我们的选择。

源代码审核是一种白盒测试技术，这是一种很长时间以来都流行的软件产品安全漏洞检测方法。这种方法需要审核者了解编程概念和产品功能的每一个细节，深入洞察产品的运行环境。除此之外，源代码审核还有一个显而易见的缺陷——必须首先要获得产品的源代码。

幸运的是，除了白盒技术外，我们还可以使用不需要访问源代码的黑盒技术。模糊测试就是黑盒技术中的一种可选方法，这种方法在发掘那些用审核方法无法发现的产品关键安全漏洞方面被证明是成功的。模糊测试是这样的一个过程：向产品有意识地输入无效数据以期望触发错误条件或引起产品的故障。这些错误条件可以指导我们找出那些可挖掘的安全漏洞。

模糊测试没有实际的执行规则。它是一种技术，测试结果是这种技术的成功性的唯一度量。任意一个给定的产品都可能接受无限的输入。模糊测试技术旨在预测产品中可能存在的编程错误以及什么样的输入可能会触发错误。正因为如此，与其说它是一门学科，不如说它是一种技术。

模糊测试可以简单到只是随意敲打键盘来输入随机数据。我的一个朋友有个3岁的儿子，他就是用这么简单的手段发现了Mac SO X操作系统的屏幕界面锁定功能中的一个漏洞。我的朋友锁定了屏幕界面然后到厨房找酒喝。当他回来的时候，他的儿子已经设法成功地解除了锁定，并且打开了浏览器，所用的方法正是随意敲打键盘。

过去的几年里，我用模糊测试技术和模糊工具在大量的软件中发现了数百个漏洞。2003年12月，我编写了一个简单的程序向一个远程服务发送随机UDP 包流。结果这个程序发现了Microsoft WINS服务器的两个新的漏洞。该程序后来又帮助我在其他产品中找出了少量的缺陷。最后的结果证明，用简单的随机UPD包流能够发现计算机协会的多个产品中的漏洞，包括Norton Ghost管理服务和OS X操作系统的一个公共服务。

模糊器对发现网络协议以及其他许多产品都有效。在2006年的第一季度，我精心设计了3个不同的浏览器模糊工具，结果发现了多种浏览器中的缺陷。2006年第二季度，我又编写了一个Active X模糊器（AxMan），仅在Microsoft的产品中就发现了超过100个缺陷。这些缺陷许多都是在“Month of Browser Bugs”项目中形成的，结果导致该项目组又进一步开发了“Metasploit”框架中的模块。在最初开发AxMan后的接近一年的时间里，我还利用模糊测试发现了AxMan本身所包含的一些漏洞。模糊器真是一个能够不断赐予我们新礼物的工具。

本书是一部真正让我们有理由相信模糊测试是一门技术的专著。书中所介绍的内容涵盖了对新产品执行模糊测试以及创建有效的模糊工具所需要的全部知识。有效模糊测试的关键在于明确对什么样的产品使用什么样的测试数据，以及需要什么工具来操纵、监控和管理模糊测试过程。本书的作者是模糊测试技术的先锋，在阐明模糊测试的复杂过程方面作出了卓越贡献。

祝各位猎捕Bug愉快！

——H. D. Moore



前　言

我知道“人类和鱼类能够和平共处”。

——George W. Bush, 2000年9月29日

简介

模糊测试的概念至少已经流传了20年，但是直到最近才引起广泛的关注。安全漏洞困扰了许多流行的客户端应用程序，包括Microsoft的Internet Explorer、Word和Excel，它们中的许多漏洞在2006年通过模糊测试技术发现。模糊测试技术的有效应用产生了许多新的工具和日益广泛的影响。本书是第一部公开发表的关于这一主题的专著，这一尴尬事实同时也预示着未来人们将会对模糊测试产生更浓厚的兴趣。

多年来，我们参与了许多有关安全漏洞的研究工作，并且在日常工作中使用了各种不同的模糊测试技术，从不成熟的、凭借个人嗜好的项目到高端的商业产品，都用到过模糊测试。每一位作者都曾参与开发过自用版本的和公开发行版本的模糊器。这本书凝聚了我们以往的实践经验和正在进行的研究项目所花费的心血，我们希望读者能够从中获益。

目标读者

安全性领域的书籍和文章通常由这一领域的研究者所撰写，以方便该领域的其他研究者参考。我们坚信，只要安全性领域的研究小组把解决安全性问题视为其唯一责任，那么安全性问题的数量和严重程度就会随着时间的推移而继续增长。因此，我们付出巨大的努力以使本书能够服务于更多的读者，既包括模糊测试的新手也包括早已对本领域有所了解的读者。

假设我们只是将开发完成的应用程序提交给一个安全小组，然后让他们在产品发布之前对其进行一个快速审核，相信这样的过程能够产生安全的应用程序显然是不现实的。当开发者或QA组的组员说：“安全根本不是问题——我们有个安全小组关心这件事呢”，如此这般，日子就会一天一天的过去。安全性必须融入软件开发生命周期（SDLC），而不是到了最后才草率处理。

让开发组和QA组把注意力集中在安全性问题上可能是个过高的要求，特别是对那些以往没有这么做的开发组和QA组来说尤其如此。我们认为模糊测试是一种独一无二的安全漏洞发掘方法学，由于它能够高度自动化，因此学习和掌握这种方法学的读者可以相当广泛。我们希望经验丰富的安全领域的研究者可从本书获得有价值的东西，同样希望开发人员和QA人员从中获益。模糊测试可以并且应该是任何完整SDLC的一部分，不仅在测试阶段需要考虑，在开发阶段也同样需要考虑。缺陷发现得越及时，修补缺陷的成本就越低。

预备知识

模糊测试是一个广泛的主题。尽管本书会介绍一些不专属于模糊测试的背景内容，但是我们仍然假设读者应该拥有这一领域的预备知识。在学习本书之前，读者至少应该对程序设计和计算机网络有一定的基本了解。模糊测试涉及自动化安全测试，这本书的内容自然要包括如何构造自动化工具。我们有目的地选择了多种编程语言来完成这个任务。语言的选择是根据具体任务的，这也说明了模糊测试可以用多种方法实现。当然，没有必要一一罗列所用到的所有编程语言的背景知识，但是介绍一两种语言无疑会帮助读者从这些章节中获益。

本书自始至终都贯穿着对各种安全漏洞的详细描述，并讨论如何通过模糊测试来识别这些漏洞。然而，定义或剖析安全漏洞本身的性质并不是本书的目标。一些优秀的书籍是专门讨论这一主题的。如果需要寻找一部关于软件安全漏洞的初级读本，可以参阅Greg Hoglund、Gray McGraw所著的《Exploiting Software》和Jack Koziol、David Litchfiel等的《Shellcoder's Handbook》，它们都是极好的参考读物。

学习方法

如何最好地利用本书，这取决于读者的背景和目的。如果你是一位模糊测试的初学者，我们推荐你按顺序逐章消化理解，因为本书的内容进行了精心编排，前面先介绍一些必要的背景信息，随后转入高级主题。反之，如果你已经在使用各种模糊测试工具方面花费了一些时间，那么请不要犹豫，可以直接进入感兴趣的主题，因为本书的不同逻辑章节的划分大致上是相互独立的。

本书的第一部分主要介绍不同的、具体的模糊测试类型，这些模糊测试类型将在随后的章节中逐一讨论。如果读者对模糊测试比较陌生，可以考虑把这一部分作为必读章节。模糊测试可以作为多种目标下的安全性测试方法，不过这些目标下的方法都遵循相同的基本原则。在第一部分，我们试图将模糊测试定义为一种安全漏洞发掘方法并详细介绍相关的知识，不考虑这种方法运用于何种目的。

第二部分关注模糊测试的各种相关应用目标。每种目标的介绍跨越了两到三章。最前面的一章介绍每类目标的背景信息，随后的各章集中介绍这些目标下的模糊测试自动化，详细阐述如何针对这种目标构造模糊器。当认为有必要分别介绍Windows平台和UNIX平台下的模糊器工具时，这两个主题分别安排在有关自动化的两章。例如，以第11章“文件格式模糊测试”为例，该章详细描述有关模糊文件分析器的内容，第12章“文件格式模糊测试：UNIX平台上的自动化测试”则深入介绍基于UNIX的文件模糊器的实用程序设计，第13章“文件格式模糊测试：Windows平台上的自动化测试”讲解运行在Windows环境中的文件格式模糊器如何构造。

第三部分讨论模糊测试领域的高级主题。对于那些已经牢固掌握模糊测试背景知识的读者，可以直接跳入第三部分，不过大部分读者很可能需要先了解第一部分和第二部分，然后再学习第三部分。第三部分关注的是近年来浮现出的新技术，这些技术刚刚得到实施，但是未来将成为安全漏洞发掘的高级工具可以利用的模糊测试技术。

最后，在第四部分，我们将总结学习过本书后的收获，然后深入洞察未来的发展方向。尽管模糊测试并不是一个新概念，但是这一领域仍然有足够的发展空间，并且我们希望本书将为未来的研究空间注入一丝灵感。

少许幽默

写书是一件严肃认真的工作，尤其是对诸如模糊测试这样的复杂主题。这就是说，我们希望尽量给随后的读者（实际上这些人可能比写书的人更重要）带来一些乐趣，同时也尽最大的努力让写作的过程更愉快。出于这样的考虑，我们决定在每一章的开头引用美国第43届总统George W. Bush（别名Dubya）的一段话。不论你的政治倾向或信仰是什么，没人能够否定Bush先生在过去几年中所炮制出的一些引文，这些引文甚至能够写满一年的日历！我们从中挑选了一些最喜欢的引文与读者分享，希望读者和我们得到同样的快乐。读完本书后，读者会发现模糊测试可以被应用于各种不同的目标，显然也可以应用到对英语的模糊测试。

关于封面

有时，安全漏洞经常被称为“鱼”（例如，可以参见DailyDave安全性邮件列表中关于“The L Word & Fish”的提示线索）。这是一个有用的类比，在讨论安全性和安全漏洞时可以被应用到这个问题的各个方面。可以把这一领域的研究者比喻为钓鱼者。对应用程序的汇编代码实施逆向工程，逐行分析查找安全漏洞，这样的人可比喻为“深海钓鱼者”。同许多其他的审核手段相比，模糊测试充其量只是海面搜索，并且通常只对“容易抓的鱼”更有效。此外，大灰熊是一个著名的“模糊动物”，当然也是强大的动物。本书的封面中有一个模糊的动物，正在捕捉一条鱼，后者代表一个安全漏洞。

配套站点

站点www.fuzzing.org绝对是本书不可分割的一部分，并不仅仅起到补充资源的作用。除了包含本书出版后的勘误表外，该站点还是书中所有源代码和工具的一个中央资源仓库。经过一段时间的努力，我们打算让这个站点从一个以图书为中心的资源站点发展成为模糊测试这一学科的资源、工具和信息的有价值的社区。我们欢迎读者提出反馈信息，以帮助我们让该站点成为一个有价值的、开放的知识库。

目　录

译者序
译者简介
序言
前言
致谢

第一部分　背　景

第1章　安全漏洞发掘方法学 ……1
1.1　白盒测试 ……1
1.1.1　源代码评审 ……1
1.1.2　工具和自动化 ……3
1.1.3　优点和缺点 ……5
1.2　黑盒测试 ……5
1.2.1　人工测试 ……6
1.2.2　自动测试或模糊测试 ……7
1.2.3　优点和缺点 ……8
1.3　灰盒测试 ……9
1.3.1　二进制审核 ……9
1.3.2　自动化的二进制审核 ……11
1.3.3　优点和缺点 ……12
1.4　小结 ……12
第2章　什么是模糊测试 ……13
2.1　模糊测试的定义 ……13
2.2　模糊测试的历史 ……14
2.3　模糊测试阶段 ……17
2.4　模糊测试的局限性和期望 ……18
2.4.1　访问控制缺陷 ……18
2.4.2　设计逻辑不良 ……19
2.4.3　后门 ……19
2.4.4　内存破坏 ……19
2.4.5　多阶段安全漏洞 ……20
2.5　小结 ……20
第3章　模糊测试方法和模糊器类型 ……21
3.1　模糊测试方法 ……21
3.1.1　预先生成测试用例 ……21
3.1.2　随机方法 ……21
3.1.3　协议变异人工测试 ……22
3.1.4　变异或强制性测试 ……23
3.1.5　自动协议生成测试 ……23
3.2　模糊器类型 ……23
3.2.1　本地模糊器 ……24
3.2.2　远程模糊器 ……25
3.2.3　内存模糊器 ……27
3.2.4　模糊器框架 ……28
3.3　小结 ……29
第4章　数据表示和分析 ……30
4.1　什么是协议 ……30
4.2　协议域 ……31
4.3　简单文本协议 ……32
4.4　二进制协议 ……32
4.5　网络协议 ……35
4.6　文件格式 ……36
4.7　常见的协议元素 ……38
4.7.1　名字-值对 ……39
4.7.2　块标识符 ……39
4.7.3　块长度 ……39
4.7.4　校验和 ……39
4.8　小结 ……39
第5章　有效模糊测试的需求 ……40
5.1　可重现性和文档记录 ……40
5.2　可重用性 ……41
5.3　过程状态和过程深度 ……42

5.4 跟踪、代码覆盖和度量 ……44
5.5 错误检测 ……44
5.6 资源约束 ……45
5.7 小结 ……46

第二部分 目标和自动化

第6章 自动化测试和测试数据生成 ……47
6.1 自动化测试的价值 ……47
6.2 有用的工具和库 ……48
6.2.1 ETHEREAL/WIRESHARK ……48
6.2.2 LIBDASM和LIBDISASM ……48
6.2.3 LIBNET /LIBNETNT ……49
6.2.4 LIBPCAP ……49
6.2.5 METRO PACKET LIBRARY ……49
6.2.6 PTRACE ……49
6.2.7 PYTHON EXTENSIONS ……49
6.3 编程语言的选择 ……50
6.4 测试数据生成和模糊启发式 ……50
6.4.1 整型值 ……51
6.4.2 字符串重复 ……53
6.4.3 字段分隔符 ……53
6.4.4 格式化字符串 ……55
6.4.5 字符翻译 ……55
6.4.6 目录遍历 ……56
6.4.7 命令注入 ……56
6.5 小结 ……57
第7章 环境变量和参数的模糊测试 ……58
7.1 本地化模糊测试介绍 ……58
7.1.1 命令行参数 ……58
7.1.2 环境变量 ……58
7.2 本地化模糊测试准则 ……60
7.3 寻找目标程序 ……60
7.4 本地化模糊测试方法 ……63
7.5 枚举环境变量 ……63
7.6 自动化的环境变量测试 ……64
7.7 检测问题 ……65
7.8 小结 ……67
第8章 环境变量和参数的模糊测试：自动化 ……68
8.1 iFUZZ本地化模糊器的特性 ……68
8.2 iFUZZ的开发 ……69
8.3 iFUZZ的开发语言 ……73
8.4 实例研究 ……73
8.5 益处和改进的余地 ……74
8.6 小结 ……74
第9章 Web应用程序和服务器模糊测试 ……75
9.1 什么是Web应用程序模糊测试 ……75
9.2 目标应用 ……77
9.3 测试方法 ……78
9.3.1 建立目标环境 ……78
9.3.2 输入 ……79
9.4 漏洞 ……88
9.5 异常检测 ……90
9.6 小结 ……91
第10章 Web应用程序和服务器的模糊测试：自动化 ……92
10.1 Web应用模糊器 ……92
10.2 WebFuzz的特性 ……94
10.2.1 请求 ……94
10.2.2 模糊变量 ……95
10.2.3 响应 ……96
10.3 必要的背景知识 ……97
10.3.1 识别请求 ……97
10.3.2 漏洞检测 ……98
10.4 WebFuzz的开发 ……100
10.4.1 开发方法 ……100
10.4.2 开发语言的选择 ……100
10.4.3 设计 ……100
10.5 实例研究 ……106
10.5.1 目录遍历 ……106

10.5.2 溢出 ……107
10.5.3 SQL注入 ……109
10.5.4 XSS脚本 ……111
10.6 益处和改进的余地 ……114
10.7 小结 ……114
第11章 文件格式模糊测试 ……115
11.1 目标应用 ……115
11.2 方法 ……116
11.2.1 强制性或基于变异的模糊测试 ……117
11.2.2 智能强制性或基于生成的模糊测试…118
11.3 输入 ……118
11.4 漏洞 ……119
11.4.1 拒绝服务 ……119
11.4.2 整数处理问题 ……119
11.4.3 简单的栈和堆溢出 ……120
11.4.4 逻辑错误 ……121
11.4.5 格式化字符串 ……121
11.4.6 竞争条件 ……121
11.5 漏洞检测 ……121
11.6 小结 ……122
第12章 文件格式模糊测试：UNIX平台上的自动化测试 ……123
12.1 NOTSPIKEFILE和SPIKEFILE ……123
12.2 开发方法 ……124
12.2.1 异常检测引擎 ……124
12.2.2 异常报告（异常检测） ……124
12.2.3 核心模糊测试引擎 ……125
12.3 有意义的代码片段 ……126
12.3.1 通常感兴趣的UNIX信号 ……127
12.3.2 不太感兴趣的UNIX信号 ……128
12.4 僵死进程 ……128
12.5 使用的注意事项 ……130
12.5.1 ADOBE ACROBAT ……131
12.5.2 REALNETWORKS REALPLAYRE…131
12.6 实例研究：REALPLAYER REALPIX格式化字符串漏洞 ……131
12.7 语言 ……133
12.8 小结 ……133
第13章 文件格式模糊测试：Windows平台上的自动化测试 ……134
13.1 Windows文件格式漏洞 ……134
13.2 FileFuzz的特性 ……136
13.2.1 创建文件 ……136
13.2.2 应用程序执行 ……138
13.2.3 异常检测 ……138
13.2.4 保存的审核 ……139
13.3 必要的背景知识 ……140
13.4 FileFuzz的开发 ……142
13.4.1 开发方法 ……143
13.4.2 开发语言的选择 ……143
13.4.3 设计 ……143
13.5 实例研究 ……149
13.6 益处和改进的余地 ……152
13.7 小结 ……152
第14章 网络协议模糊测试 ……153
14.1 什么是网络协议模糊测试 ……153
14.2 目标应用 ……155
14.2.1 数据链路层 ……156
14.2.2 网络层 ……156
14.2.3 传输层 ……157
14.2.4 会话层 ……157
14.2.5 表示层 ……157
14.2.6 应用层 ……157
14.3 测试方法 ……158
14.3.1 强制性或基于变异的模糊测试 ……158
14.3.2 智能强制性模糊测试和基于生成的模糊测试 ……158
14.3.3 修改的客户端变异模糊测试 ……159
14.4 错误检测 ……159
14.4.1 人工方法（基于调试器） ……160
14.4.2 自动化方法（基于代理） ……160
14.4.3 其他方法 ……160

14.5 小结 ……160
第15章 网络协议模糊测试：UNIX平台上的自动化测试 ……161
15.1 使用SPIKE进行模糊测试 ……161
15.1.1 选择测试目标 ……161
15.1.2 协议逆向工程 ……162
15.2 SPIKE 101 ……164
15.2.1 模糊测试引擎 ……164
15.2.2 通用的基于行的TCP模糊器 ……165
15.3 基于块的协议建模 ……166
15.4 SPIKE的额外特性 ……167
15.4.1 特定于协议的模糊器 ……167
15.4.2 特定于协议的模糊测试脚本 ……167
15.4.3 通用的基于脚本的模糊器 ……168
15.5 编写SPIKE NMAP模糊器脚本 ……168
15.6 小结 ……171
第16章 网络协议模糊测试：Windows平台上的自动化测试 ……172
16.1 ProtoFuzz的特性 ……172
16.1.1 包结构 ……172
16.1.2 捕获数据 ……173
16.1.3 解析数据 ……173
16.1.4 模糊变量 ……174
16.1.5 发送数据 ……175
16.2 必要的背景知识 ……175
16.2.1 错误检测 ……175
16.2.2 协议驱动程序 ……175
16.3 ProtoFuzz的开发 ……176
16.3.1 开发语言的选择 ……176
16.3.2 包捕获库 ……176
16.3.3 设计 ……177
16.4 实例研究 ……181
16.5 益处和改进的余地 ……182
16.6 小结 ……183
第17章 Web浏览器模糊测试 ……184
17.1 什么是Web浏览器模糊测试 ……184
17.2 目标 ……185
17.3 方法 ……185
17.3.1 测试方法 ……185
17.3.2 输入 ……186
17.4 漏洞 ……193
17.5 错误检测 ……194
17.6 小结 ……195
第18章 Web浏览器的模糊测试：自动化 …196
18.1 组件对象模型的背景知识 ……196
18.1.1 在Nutshell中的发展历史 ……196
18.1.2 对象和接口 ……197
18.1.3 ActiveX ……197
18.2 模糊器的开发 ……199
18.2.1 枚举可加载的ActiveX控件 ……200
18.2.2 属性、方法、参数和类型 ……203
18.2.3 模糊测试和监视 ……207
18.3 小结 ……208
第19章 内存数据的模糊测试 ……209
19.1 内存数据模糊测试的概念及实施该测试的原因 ……209
19.2 必需的背景知识 ……210
19.3 究竟什么是内存数据模糊测试 ……213
19.4 目标 ……214
19.5 方法：变异循环插入 ……214
19.6 方法：快照恢复变异 ……215
19.7 测试速度和处理深度 ……216
19.8 错误检测 ……217
19.9 小结 ……217
第20章 内存数据的模糊测试：自动化 …219
20.1 所需要的特性集 ……219
20.2 开发语言的选择 ……220
20.3 Windows调试API ……222
20.4 将其整合在一起 ……225
20.4.1 如何在特定点将“钩子”植入目标进程 ……226
20.4.2 如何处理进程快照和恢复 ……228

20.4.3 如何选择植入钩子的点 ……231
20.4.4 如何对目标内存空间进行定位和变异 ……231
20.5 一个最好的新工具PyDbg ……231
20.6 一个构想的示例 ……233
20.7 小结 ……244

第三部分 高级模糊测试技术

第21章 模糊测试框架 ……245
21.1 模糊测试框架的概念 ……245
21.2 现有框架 ……247
21.2.1 antiparser ……247
21.2.2 Dfuz ……249
21.2.3 SPIKE ……252
21.2.4 Peach ……255
21.2.5 通用模糊器 ……257
21.2.6 Autodafé ……259
21.3 定制模糊器的实例研究：Shockwave-Flash ……260
21.3.1 SWF文件的建模 ……261
21.3.2 生成有效的数据 ……270
21.3.3 对环境进行模糊测试 ……271
21.3.4 测试方法 ……272
21.4 模糊测试框架Sulley ……272
21.4.1 Sulley目录结构 ……272
21.4.2 数据表示 ……274
21.4.3 会话 ……282
21.4.4 事后验证阶段 ……286
21.4.5 一个完整的实例分析 ……290
21.5 小结 ……295
第22章 自动化协议解析 ……297
22.1 模糊测试存在的问题是什么 ……297
22.2 启发式技术 ……299
22.2.1 代理模糊测试 ……299
22.2.2 改进的代理模糊测试 ……300
22.2.3 反汇编启发式规则 ……302
22.3 生物信息学 ……303
22.4 遗传算法 ……305
22.5 小结 ……309
第23章 模糊器跟踪 ……310
23.1 我们究竟想要跟踪什么 ……310
23.2 二进制代码可视化和基本块 ……311
23.2.1 CFG ……312
23.2.2 CFG示例 ……312
23.3 构造一个模糊器跟踪器 ……313
23.3.1 刻画目标特征 ……314
23.3.2 跟踪 ……314
23.3.3 交叉引用 ……316
23.4 对一个代码覆盖工具的分析 ……318
23.4.1 PStalker设计概览 ……319
23.4.2 数据源 ……320
23.4.3 数据探查 ……321
23.4.4 数据捕获 ……321
23.4.5 局限性 ……321
23.4.6 数据存储 ……322
23.5 实例研究 ……324
23.5.1 测试策略 ……325
23.5.2 测试方法 ……327
23.6 益处和改进的余地 ……330
23.7 小结 ……333
第24章 智能故障检测 ……334
24.1 基本的错误检测方法 ……334
24.2 我们所要搜索的内容 ……336
24.3 选择模糊值时的注意事项 ……340
24.4 自动化的调试器监视 ……341
24.4.1 一个基本的调试器监视器 ……341
24.4.2 一个更加高级的调试器监视器 ……344
24.5 调试器在应用程序前先看到的异常和调试器再次看到程序未捕获的异常的比较 ……347
24.6 动态二进制插装 ……348

24.7 小结 ……350

第四部分 展 望

第25章 汲取的教训 ……351
25.1 软件开发生命周期 ……351
25.1.1 分析 ……353
25.1.2 设计 ……353
25.1.3 编码 ……354
25.1.4 测试 ……354
25.1.5 维护 ……354
25.1.6 在SDLC中实现模糊测试 ……355
25.2 开发者 ……355
25.3 QA研究者 ……355
25.4 安全问题研究者 ……356
25.5 小结 ……356
第26章 展望 ……357
26.1 商业工具 ……357
26.1.1 安全性测试工具beSTORM ……357
26.1.2 BreakingPoint系统BPS-1000 ……358
26.1.3 Codenomicon ……358
26.1.4 GLEG ProtoVer Professional ……360
26.1.5 安全性测试工具Mu-4000 ……361
26.1.6 Security Innovation Holodeck ……361
26.2 发现漏洞的混合方法 ……362
26.3 集成的测试平台 ……362
26.4 小结 ……363

第一部分 背 景

第1章 安全漏洞发掘方法学

"Internet高速公路将会越来越少么？"

——George W. Bush, Concord, N. H., 2004年8月5日

如果询问任何一位有成就的安全领域的研究者如何发现漏洞，很可能会得到一大堆答案。为什么？因为可用于安全性测试的方法太多，每种方法都有自己的优点和缺点。没有一种方法是绝对正确的，也没有一种方法能够揭示一个给定目标下所有可能的漏洞。在较高的层次上，有三种主要的方法用来发现安全漏洞：白盒测试、黑盒测试和灰盒测试。这些方法之间的差别是由测试者可得到的资源决定的。白盒测试是一个极端，它需要对所有资源进行充分地访问。这包括访问源代码、设计规约，甚至有可能还要访问程序员本人。黑盒测试是另一个极端，它几乎不需要知道软件的内部细节，很大程度上带有盲目测试的味道。远程Web应用的Pen测试是黑盒测试的一个好例子，它不需要访问程序源码。位于两个极端方法之间的是灰盒测试，它的定义因询问的人不同而不同。就我们的应用目的而言，灰盒测试需要访问编译后得到的二进制代码，或许还要访问一些基本的文档。

本章将考察漏洞发掘的各种不同的高层和低层方法，起点是白盒测试，你以前可能听说过这种方法也被称为玻璃、透明或半透明测试。之后我们再定义黑盒测试和灰盒测试，模糊测试可能就属于后两者。我们将阐述每种方法的利弊，这些方法的利弊将成为本书后面介绍模糊测试时所需要的背景知识。模糊测试只是漏洞发掘的一种方法，了解其他可选的实用方法也是相当重要的。

1.1 白盒测试

作为一种测试方法学，模糊测试（fuzzing）主要属于黑盒测试和灰盒测试领域。尽管如此，我们还是要考察另一种软件开发人员可选的漏洞发掘方法，即白盒测试（white box testing）。

1.1.1 源代码评审

源代码评审既可以人工完成也可以在自动化工具的辅助下完成。计算机程序通常包含数万到数十万行代码，那么单纯的人工评审一般情况下是不可行的。自动化工具是一个宝贵的资源，它能够减少长时间对着代码行阅读而带来的繁重任务，但是自动化工具只能识别出可能的漏洞或可疑的代码片段。检测出的问题是否有效，仍然需要人工分析。

让源代码分析工具产生有用的结果，必须要克服许多障碍。对这些复杂问题的剖析已经超出了本书的范围，让我们考虑如下的一个C程序代码段，这段程序只是简单地将文本“*test*”拷贝到一个10字节的字符数组。

```
#include <string.h>

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

下面，考虑同样的代码示例，该段代码示例经过了修改，允许用户的输入被拷贝到字符数组。

```
#include <string.h>
int main (int argc, char **argv)
{
   char buffer[10];
   strcpy(buffer, argv[1]);
}
```

上面的两段代码都使用了strcpy()例程将数据拷贝到一个基于栈的缓冲区。在C/C++编程中，一般不推荐使用strcpy()库函数，因为它缺少对目的缓冲区的边界检查。因此，如果程序员不小心自己编写代码而增加了边界检查代码，便可能会发生缓冲区溢出，从而将数据置于目的容器的边界之外。第一个代码例子不会发生缓冲区溢出，因为字符串“test”（包括后跟的null终止标记）的长度总是固定值5，因此小于目标缓冲区的10字节。第二个例子中的场景中可能会也可能不会发生缓冲区溢出，这要取决于用户在命令行中输入的参数数据长度。这里的关键问题是用户是否能够控制输入给有可能发生漏洞的函数的参数长度。一次彻底的代码评审可能会将strcpy()所在的一行标记为“可能产生漏洞”。尽管如此，还是需要跟踪该函数的实际输入参数，以理解可被人为利用的执行条件是否真正存在。这并不是说源代码评审对安全领域的研究者不是一个有价值的技术工具。源代码评审应该在可获得代码的任何时候进行。然而，是否要进行源代码评审取决于你的角色和观点，通常人们不需要在如此细节的层次上访问被测对象。

人们经常不正确地认为白盒测试是比黑盒测试更有效的方法。对软件的何种审视能够比访问源代码更好、更精确呢？不过无论如何都不要忘记，对源代码来说，你看到的东西并不一定是实际执行的东西。软件构建过程在从源代码到汇编代码的转换中可能会发生很大的改变。因为这个原因和其他的原因，不能说一种测试方法就一定比另一种测试方法更好。它们仅仅是不同的方法，通常会揭示不同类型的漏洞。为了达到充分覆盖的目的，有必要结合多种测试方法。

微软源代码泄露

为了说明源代码评审未必一定优于黑盒测试，让我们考虑和分析发生在2004年2月的一个

事件。在没有事先告戒的情况下，有传闻说Microsoft Windows NT 4.0和Windows 2000操作系统提供的部分源代码文档被传播到互联网上。Microsoft后来确认这些文档是真实的操作系统代码。许多公司当时感到极度紧张，因为这一泄露事件将导致上述两个操作系统版本产生大量的安全漏洞。但是他们所担心的事情没有发生。事实上，直到今天，只有很少的安全漏洞被归因于当时的源代码泄露。其中的一个安全漏洞是CVE-2004-0566，其具体细节是当操作系统在处理位图文件时会产生一个整数值溢出[㊀]。有趣的是，Microsoft否认这个安全漏洞的发现，声称其所属的软件公司在一次内部审核中早已经发现了这个问题[㊁]。为什么我们没有发现大量的安全漏洞？不是说源代码评审可以发现一切吗？事实是源代码评审尽管是应用程序或操作系统安全性审核的一个重要部分，但是由于代码的规模和复杂性，源代码评审难以充分地进行。此外，反汇编分析技术也可以发现大部分源代码中的问题。例如，让我们考察一下TinyKRNL[㊂]和ReactOS[㊃]项目，它们的目的都是为应用软件提供与Microsoft Windows内核及其操作系统的兼容性。这些项目的开发者并没有访问Microsoft的内核源代码，但是仍然能够在一定程度上创建项目，提供一个兼容Windows的环境。在审核Windows操作系统时，不大可能会直接访问Windows源码，相反，上述项目的源代码可以被作为解释Windows汇编代码的指南。

1.1.2 工具和自动化

源代码分析工具一般可以分为三类——编译时检查器、源代码浏览器或自动源代码审核工具。编译时检查器在源代码编译时查找漏洞。此类工具通常与编译器集成在一起，但是主要查找与安全性有关的问题而不是应用程序的功能问题。Microsoft Visual C++的/analyze编译选项是一个编译时检查器[㊄]的例子。Microsoft还提供了PREfast for Drivers[㊅]，它能够检测针对驱动程序开发的不同类型的漏洞，而编译器可能检测不到这些漏洞。

源代码浏览器是专门设计用于辅助人工检查和评审源代码的软件工具。这类工具允许评审者执行代码的高级搜索，枚举代码，以及在代码的交叉引用位置之间进行导航。例如，一位评审者可能会使用这样的工具来定位所有strcpy()调用的位置，力图识别可能的缓冲区溢出漏洞。Cscope[㊆]和Linux Cross-Reference[㊇]是当前流行的源代码浏览器。

源代码自动审核工具用于扫描源代码以及自动识别可能的关注区域。同大多数安全性工具一样，源代码自动审核工具既有商业工具也有开源的自由软件解决方案。除此之外，此类工具

㊀ http://archives.neohapsis.com/archives/fulldisclosure/2004-02/0806.html
㊁ http://news.zdnet.com/2100-1009_22-5160566.html
㊂ http:www.tinykrnl.org/
㊃ http:/www.reactos.org/
㊄ http://msdn2.microsoft.com/en-us/library/k3a3hzw7.aspx
㊅ http://www.microsoft.com/whdc/devtools/tools/PREfast.mspx
㊆ http://cscope.sourceforge.net/
㊇ http://lxr.linux.no/

倾向于关注具体的编程语言，因此如果目标软件使用了不同的编程语言的话，可能需要多种源代码自动审核工具。在商业软件工具方面，有来自Fortify㊀、Coverity㊁、KlockWork㊂、Gramma Tech㊃和其他供应商的产品。一些流行的自由软件工具列于表1-1，其中包括它们审核的语言以及支持的平台。

表1-1　源代码审核的自由软件工具

名　称	语　言	平　台	下　载
RATS（Rough Auditing Tool for Security）	C、C++、Perl、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++	UNIX	http://www.dwheeler.com/flawfinder/
Jlint	Java.	UNIX、Win32	http://jlint.sourceforge.net/
CodeSpy	Java	Java	http://www.owasp.org/software/labs/codespy.html

重要的是要记住，没有任何自动化工具能够完全替代有经验的安全研究员的技能。它们只不过是工具，能够将繁重的分析源代码的任务进行流水化和自动完成，有助于节省时间和解决一些问题。这些工具生成的报告仍然必须由具备经验的分析员评审，以识别出伪问题，也需要有开发人员的评审以实际修复问题。例如，下面的一段代码输出是由Rough Auditing Tool for Security（RATS）自动生成的，被分析的代码是本章前面所举的存在安全漏洞隐患的程序示例。分析结果可以表明这样一个事实：该段程序使用了固定长度缓冲区并且对`strcpy()`的使用可能是不安全的。但是，分析结果没有肯定地说确实存在一个漏洞，只是提醒用户注意存在可能不安全的代码区域，但是该区域是否真的不安全，则需要由他（她）决定。

```
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 lines analyzed: 7
```

㊀ http://www.fortifysoftware.com/

㊁ http://www.coverity.com/

㊂ http://www.klocwork.com/

㊃ http://www.grammatech.com/

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

1.1.3 优点和缺点

如同前面提到的，发现安全漏洞不存在唯一正确的方法学。那么如何选择一种合适的方法学呢？当然，有时的决策是由我们自己确定的。例如，如果我们不能访问目标软件的源代码，那么就不可能进行白盒测试。这对大部分安全研究者和用户都是一样的，尤其是那些购买了Windows环境的商业软件的人。那么白盒测试有什么优点呢？

- 覆盖能力：由于白盒测试能够获得所有源代码，因此代码评审允许完全的覆盖。所有可能的代码路径都可以被审核，以发现可能的漏洞。当然，这个过程也可能导致误报的出现，因为一些代码路径可能在代码执行的时候不可达。

代码评审并不总是可行的。即使能够执行代码评审，也应该和其他漏洞发掘方法学结合使用。源代码分析有下面一些缺点：

- 复杂性：源代码分析工具是不完善的，有可能报告出伪问题。因此，这类工具产生的错误报告并不是游戏的结束。这些结果必须经过有经验的程序员的评审，以识别出那些能够合理代表安全性漏洞的问题。重要的软件项目典型地要包含数十万行代码，工具所产生的报告可能相当冗长并且需要大量的时间阅读。
- 可用性：由于白盒测试能够获得所有源代码，因此代码评审允许完全的覆盖。尽管许多UNIX项目是开放源代码的，并且允许源代码被评审，但是这种情况在Win32环境下比较罕见，对于商业软件这种情况则通常不存在。如果不能访问源代码，那么白盒测试则根本不能作为一个测试选项。

1.2 黑盒测试

黑盒测试意味着只能了解外部观察到的东西。作为终端用户，可以控制输入，从一个黑盒子的一端提供输入，从盒子的另一端观察输出结果，但是并不知道被控目标的内部工作细节。

在远程访问Web应用和Web服务时这种情形最常见。我们可以采用超文本置标语言（HTML）或可扩展置标语言（XML）请求的形式产生输入并观察生成的Web页面或返回值，但是并不知道幕后发生了什么。

让我们考虑另一个例子。假设你在某个时候购买了一个应用软件，例如Microsoft Office，通常你只能得到一个经过编译后得到的二进制程序，并不能得到用于构建这个应用程序的源代码。在这种情况下，你的观点将决定测试方法的盒子的颜色。如果你不打算采用逆向工程技术，那么黑盒测试是有效的。反之，就应该采用灰盒测试方法，灰盒测试将在下一节讨论。

1.2.1 人工测试

仍然以Web应用为例，人工测试可能要包括使用一个标准的Web浏览器来浏览一个Web站点的层次结构，同时冗长乏味地在所观察到的感兴趣的区域输入可能导致危险的各种输入数据。在审核的最初阶段，这个技术可能只是被零星地应用，例如，通过给各种各样的参数增加引用，以期望揭示出SQL注入漏洞。

在没有自动化工具辅助的情况下通过人工测试应用程序来查找漏洞，这通常并不是一个可行的方法（除非公司雇佣了一大批实习生）。这种方法能够带来益处的场景是它被用于扫除（sweeping）多个应用程序中类似的一个漏洞时。当多个程序员在不同的项目中犯了相同的错误时，就需要利用这种扫除技术。例如，在一个专门的LDAP（Lightweight Directory Access Protocol）服务器中发现了一个缓冲区溢出，并且通过测试其他LDAP服务器发现同样的安全漏洞。如果我们假设程序之间经常共享代码或者同一个程序员参加过多个开发项目，那么这种情况就并非不经常出现。

扫除技术

CreateProcess()是Microsoft Windows 应用程序编程接口（API）提供的一个函数。正如它的名字所暗示的那样，CreateProcess()将启动一个新进程及其主线程[⊖]。该函数的原型是如下定义的：

```
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
);
```

如果lpApplicationName参数被赋予一个NULL值，那么被启动的进程将是lpCommandLine

⊖ http://msdn2.microsoft.com/en-us/library/ms682425.aspx

参数中第一个空白符号分隔之后的值，这是人们都知道并且记录在文档中的该函数的行为。考虑下面的例子中对CreateProcess()函数的调用：

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

在上述情形下，CreateProcess()函数将试图反复启动每一个空格符之后的值所代表的进程：

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

这种尝试会一直进行，直到发现了一个可执行文件的名字或者所有的选择都已经被穷尽。因此，如果一个可执行文件program.exe位于C:\路径下，带有类似上述结构中对CreateProcess()不安全调用的应用程序将会执行该program.exe程序。这为攻击者提供了一个便利的机会，他们会试图让实际要执行的程序不可达并执行另外一个程序。

2005年10月发布的一个安全性咨询[⊖]报告中列举了好几个流行的应用程序中采用了不安全的CareateProcess()调用。这些问题的发现得益于一次成功的但是却非常简单的扫除技术实践。如果你想要利用该扫除技术发现相同的安全漏洞，那么可以拷贝和重新命名一个简单的应用程序，例如notepad.exe，然后将它置于c:\路径下。然后正常使用计算机。如果先前拷贝的这个应用程序突然在预期之外执行了，那么就很可能发现了一个因不安全调用CreateProcess()而引起的漏洞。

1.2.2 自动测试或模糊测试

模糊测试在很大程度上是一种强制性的技术，因此它缺乏“优雅性”，它的目标是简单加有效。模糊测试这个术语将在第2章中详细定义和解释。简单地说，模糊测试包括把能够想到的所有东西都抛给被测目标，然后监视结果。大部分软件都应该带有具体输入并且足够健壮，在接收到异常输入的时候能够完美地恢复。考虑如下所示的一个简单的Web表单，参见图1-1。

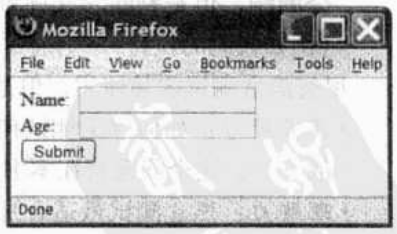


图1-1 一个简单Web表单

合理的假设是Name域应该接受一个字符串值，Age域应该接受一个整数值。如果用户偶然改变了两个域的实际输入范围并且在Age域输入了一个字符串后会发生什么呢？字符串值会自动转换为基于ASCII码的整数值吗？是否会显示一条错误报告消息？应用程序会崩溃吗？模糊测试试图通过一个自动化的过程来回答这个问题。研究

⊖ http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=340

者不需要知道应用程序的内部工作细节，因此执行的是黑盒测试。要做的就是站在后面向目标投掷石头，等待玻璃被打破的声音。就这个意义而言，模糊测试可归结为黑盒测试。然而，在本书中，我们将说明，为什么强制性的模糊测试可以变得更像“外科手术”，从而让石头每次的飞行路线更直接并且更真实。因此，模糊测试也可以应用在灰盒测试中。

Microsoft使用模糊测试吗

答案是肯定的。Microsoft 2005年3月发布的可信计算安全开发生命周期（SDL）文档[⊖]清楚地阐明了Microsoft考虑将模糊测试作为一种在软件发布之前寻找安全漏洞的关键技术。SDL是一个关于在软件开发生命周期中嵌入安全性的倡议，认为安全性是参与开发过程的每一个人的职责。模糊测试在SDL中被提为安全性测试工具，应该在软件实现阶段加以利用。实际上，这个文档中说到：“尽管对模糊测试的强调只是最近才被加入到SDL中，但是到目前为止所取得的效果却非常鼓舞人心”。

1.2.3 优点和缺点

黑盒测试尽管并不总是最好的方法，但却总是一个可选的方法。黑盒测试的优点包括以下几个方面：

- 可用性：黑盒测试总是可以应用的，甚至在源代码可用的情况下应用黑盒测试也是有益处的。
- 可重现性：由于黑盒测试的实施不需要事先对目标做出假设，以文件传输协议（FTP）服务器为例，黑盒测试可以很容易地被定制成能够测试任何其他FTP服务器。
- 简单性：一些测试方法，例如代码逆向工程（RCE）等，需要专业的技巧，而黑盒测试只是测试的一个最基本层次，它能够在不十分了解应用程序内部细节的情况下执行。尽管如此，事实上，尽管通过简单地使用测试工具就能够发现诸如拒绝服务攻击这样的安全漏洞，但是判断一次简单的应用程序崩溃是否能够被关联到某些让人更感兴趣的地方（例如代码的执行），则通常需要对软件的深入理解。

尽管黑盒测试容易使用，但是也有一些缺点。黑盒测试的缺点包括：

- 覆盖能力：如何确定测试何时结束以及测试的有效性程度，这是黑盒测试面临的最大挑战之一。这个问题将在本书的第23章“模糊器跟踪”中详细讨论。
- 理解力：黑盒测试最适合那些安全漏洞由一个单独的输入向量所引起的场景。然而，复杂的攻击需要多种攻击向量，其中的一些攻击将目标应用程序置入一种脆弱状态，其他攻击向量进一步触发漏洞。此类攻击需要深刻理解应用程序的底层逻辑，并且典型地需要通过人工代码评审和RCE才能发现漏洞。

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

1.3 灰盒测试

灰盒测试的定义是在白盒测试和黑盒分析之间浮动的，我们给它下的定义是，首先它包括了黑盒测试审核，此外还包括通过逆向工程（RE）获得的结果，逆向工程也被称为逆向代码工程（RCE）。源代码是宝贵的资源，它应该容易被阅读并且支持人们理解软件具体功能的操作细节。此外，它还隐含地提示了具体的功能所预期需要的输入，以及该具体功能的预期输出。即使缺少源代码这样的重要资源，也不意味着我们会失去一切。分析编译后得到的汇编指令能够帮助阐明类似的故事，但是要付出更多的努力。在汇编代码层次上进行安全评估而不是在源代码层次上进行安全评估，这种安全评估一般称为二进制审核（binary auditing）。

1.3.1 二进制审核

RCE通常与二进制审核这个词汇被当作同义词而使用，但就我们的目的而言，我们将RCE作为一类子方法，以使它区别于完全自动化的方法。RCE的最终目标是决定一个编译后得到的二进制应用程序的底层功能。尽管将一个二进制文件完全转换回它的源代码形式是不可能的，但是有可能对汇编指令序列应用逆向工程，从而得到一种位于源代码和构成二进制代码的机器码之间的一种表示形式。通常，这种表示是汇编语言和应用程序代码流的图形表示之间的一种中间形式。

一旦二进制文件被转换成人可读的形式，这样的代码就可以被评审以查找其中可能包含漏洞之处，这在很大程度上与源代码的评审是类似的。就源代码评审而言，找到可能存在漏洞的代码片段并不是游戏的结束。还必须决定一个终端用户是否能够影响脆弱的代码片段。遵循这样的逻辑，二进制审核也被人们称为是一种“从里向外”的技术：研究者首先识出反汇编结果中令其感兴趣的可能存在的漏洞，然后反向追溯到源代码中以确定漏洞是否可以被别人所利用。

逆向工程（reverse engineering）是一种外科手术式的技术，它利用诸如反汇编器、反编译器或调试器这样的工具。反汇编器将难以被辨认的机器码解析为某种更适合人理解的汇编码。有各种自由的反汇编器软件可供使用，但是出于重要的RCE工作的要求，我们很可能需要花费一些投资来购买DataRescue的Interactive Disassembler (IDA) Pro[⊖]，参见图1-2所示。IDA是一个运行在Windows、UNIX和MacOS平台上的商用反汇编器，能够解析很多不同机器体系架构下的二进制代码。

同反汇编器类似，反编译器可对二进制代码执行静态分析并将其转换为人可读的源代码形式。反编译器试图产生更高级的语言组件，例如条件和循环结构，而不是直接将机器码转变为汇编指令。反编译器不可能完全恢复产生最初的源代码，因为包含在源代码中的一些信息，例如注释、变量名、函数名甚至基本结构在编译之后都不再被保存。Boomerang[⊖]就是反编译器的例子。对于将源代码直接翻译成机器码的编译型语言（例如C和C++），它们的反编译器在本

⊖ http://www.datarescue.com/

⊖ http://boomerang.sourceforge.net/

质上存在一些限制。对于那些将源代码编译为某种中间形式的字节码的语言，如C#，由于其字节码中包含更多的信息，因此对这些语言的反编译通常要成功的多。

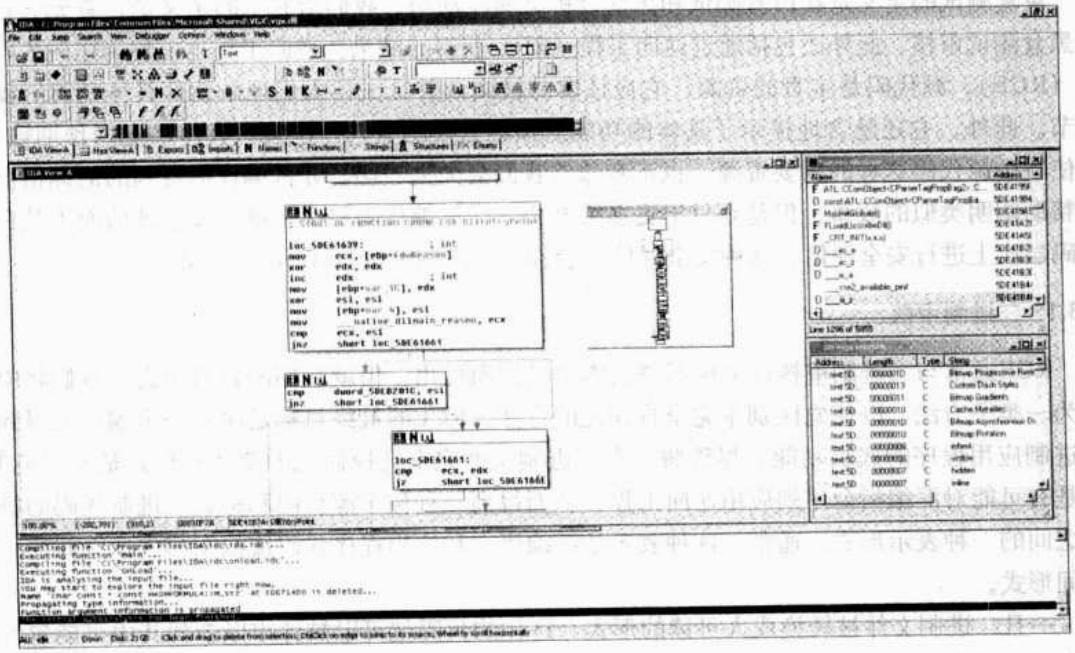


图1-2　DataRescue IDA Pro

与反汇编器和反编译器不同，调试器通过打开或附加到一个目标程序并采用动态分析技术来监控目标程序的执行。调试器能够显示应用程序正在运行时CPU寄存器的内容和内存状态。Win32平台下的流行调试器包括OllyDbg[㊀]，其运行时的一个屏幕快照如图1-3所示。此外还有Microsoft WinDbg（也被人称做“wind bag”）[㊁]。WinDbg是Windows软件调试工具包[㊂]中的一部分，可从Microsoft的网站上免费下载。OllyDbg是一个由Oleh Yuschuk开发的调试器，用户友好性稍好于WinDbg。这两个调试器都允许用户创建自定制的扩展功能组件，有许多第三方插件可用于扩展OllyDbg的功能[㊃]。UNIX环境下也有各种各样的调试器，GNU Project Debugger[㊄]（GDB）是最流行的也是最容易被移植的调试器。GDB是一个命令行调试器，许多UNIX/Linux产品中都包含这个调试器。

㊀ http://www.ollydbg.de/

㊁ http://www.openrec.org/forums/posts/4

㊂ http://www.microsoft.com/whdc/devtools/debugging/default.mspx

㊃ http://www.openrec.org/downloads/browse/OllyDbg_Plugins

㊄ http://www.gnu.org/software/gdb/gdb.html

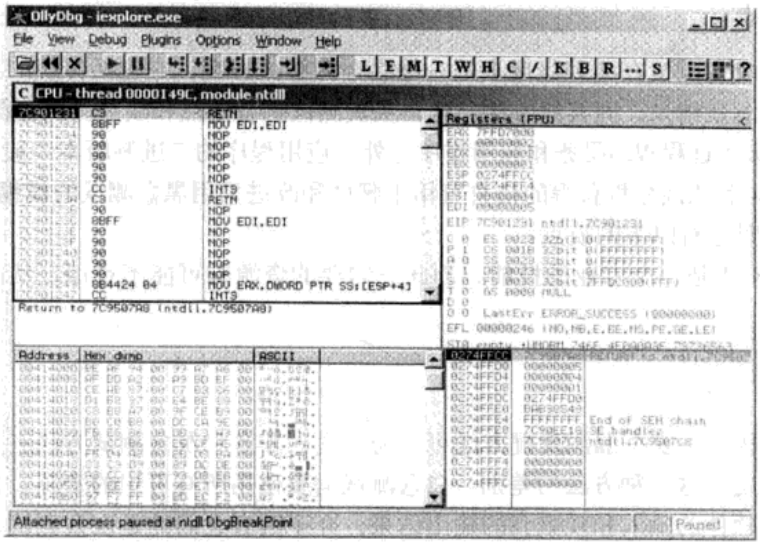


图1-3 OllyDbg

1.3.2 自动化的二进制审核

最近出现了少量的工具，这些工具试图对RCE的过程实现自动化，以识别二进制应用程序中潜在的安全漏洞。这些工具既有商业软件，也有自由软件，它们或者是IDA Pro的插件或者是单独开发的应用程序。表1-2列举了它们中的一小部分。

表1-2 自动化二进制审核工具

名 称	开发商	许可类型	说 明
LogiScan	LogicLibrary	商业软件	LogicLibrary在2004年9月收购了BugScan，此后对后者的二进制审核工具的品牌重新进行了命名，并将其包含到Logidex SDA管理方案中
BugScam	Halvar Flake	自由软件	BugScam是一个IDA Pro的IDC脚本集合，它枚举了二进制文件中的函数调用，以识别出对各种不同库函数可能的不安全调用。该应用程序主要是在BugScan的基础上采用“哄骗”的手段发现安全漏洞
Inspector	HB Gary	商业软件	Inspector是一个RCE管理系统，它统一了来自各种不同RCE工具的输出，例如IDA Pro和OllyDbg
SecurityReview	VeraCode	商业软件	VeraCode的产品将一个二进制分析套件直接集成进开发环境，类似于源代码分析套件的功能，如Coverity。在二进制代码层次上进行分析，使得VeraCode能够检测出一些问题，这些问题与“所看到的并不一定是所执行的”有关
BinAudit	SABRE Security	商业软件	BinAudit在本书出版之前还没有正式发布。尽管如此，根据SABRE Securite的Web站点上的介绍，它是一个IDA Pro的插件，用来识别一些安全漏洞，例如数组访问越界、内存破坏、内存泄露等

1.3.3 优点和缺点

如前所述，灰盒测试在某种程度上是一种混合的解决方案，它包括传统的黑盒测试，外加通过RCE获得的洞察结果。同其他方法一样，灰盒测试也有优点和缺点。灰盒测试的优点有：

- 可用性：除了远程Web服务和应用程序之外，应用程序的二进制版本一般总是可用的。
- 覆盖：从灰盒测试分析获得的信息可用于帮助和改进采用黑盒测试的模糊测试技术。

灰盒测试同时也有以下的缺点：

- 复杂性：RCE是一个专业的技能，因此它需要的资源有可能不可用，从而导致这种方法无法使用。

1.4 小结

在最高的层次上，安全漏洞发掘方法可分为白盒、黑盒和灰盒测试方法三大类。测试者可以获得的资源决定了这三种方法的差别。白盒测试需要使用所有可用的资源，包括源代码，而黑盒测试只访问软件的输入和观察到的输出结果。介于两者之间的是灰盒测试，它在黑盒测试的基础上通过对可用的二进制文件的逆向工程而获得了额外的分析信息。

白盒测试包括各种不同的源代码分析方法。可以人工完成也可以通过利用自动化工具完成，这些自动化工具包括编译时检查器、源代码浏览器或自动源代码审核工具。

在执行黑盒测试时，源代码是不可用的。仅仅通过黑盒测试来执行模糊测试，这被认为带有盲目性。尽管能够生成输入，也可以监控输入所对应的响应，但是没有办法获知和分析目标软件的内部状态。基于灰盒测试的模糊测试类似于基于黑盒测试的灰盒测试，但是增加了由RCE获得的数据。模糊测试企图反复为应用程序提供非预期的输入，同时监控可能产生的异常结果。本书的剩余部分将关注模糊测试，将其作为一种发现安全漏洞的方法。

第2章 什么是模糊测试

“他们不正确地低估了我。”

——George W. Bush，Bentonville，Ark，2000年11月6日

在主流的词典中，模糊测试（fuzzing）这个术语并不存在，同时这个概念有许多别名，对许多读者来说模糊测试很可能是一个完全陌生的名词。模糊测试是一个宽泛的研究领域，同时也是一种令人激动的软件安全性分析方法。贯穿本书，我们将深入讨论不同的模糊测试方法的细节和目标。不过在详细介绍模糊测试之前，我们先要给模糊测试下一个定义，考察模糊测试的历史，阐述模糊测试的各个阶段，最后再指出模糊测试的局限性。

2.1 模糊测试的定义

如果在你的词典中查找“fuzzing”这个词，恐怕很可能找不到任何能够为安全性领域的研究者所用的定义。Wisconsin-Madison大学的一个研究项目中首次公开提及了这个术语，现在这个术语已经被广为采纳，成为一种完整的软件测试方法学的代名词。在学术界，与模糊测试最接近的专业术语可能是边界值分析（boundary value analysis，BVA）[⊖]，在边界值分析中，需要考察给定输入的合法值的边界，然后以边界内和边界外的合法和非法值来创建测试。BVA可以帮助确保异常处理机制能够过滤掉非预期的值，同时允许最大范围的可接受的输入值。模糊测试类似于BVA，但是在执行模糊测试时，我们并不仅仅唯一关注边界值，而是需要额外关注任何可能引发未定义或不安全行为的输入。

出于本书的写作目的，我们将模糊测试定义成是一种通过提供非预期的输入并监视异常结果来发现软件故障的方法。模糊测试一般是一个自动或半自动的过程，这个过程包括反复操纵目标软件并为其提供处理数据。当然，这是一个相当泛化的定义，但是这个定义抓住了模糊测试的最基本概念。所有的模糊器都可分分为两大类：基于变异的模糊器，这种模糊器对已有数据样本应用变异技术以创建测试用例；基于生成的模糊器，这种模糊器通过对目标协议或文件格式建模的方法从头开始产生测试用例。这两大类模糊器分别有各自的优点和缺点。在本书的第3章中，我们将对模糊测试方法学做更进一步的分类并讨论每一类方法的好处和问题。

如果你是一位模糊测试的新手，那么可以将模糊测试类比为如何闯进一幢房子。假设犯罪人员要破门进入某人的家，假设采用纯白盒的方法，那么在实施破门之前应该能够得到关于这个家的所有信息。这些信息可能包括房屋设计图、各种锁的制造商列表、房屋建筑材料的详情，等等。尽管这种方法有独一无二的优点，但是也并非万无一失。在这种方法下，执行破门的过

⊖ http://en.wikipedia.org/wiki/Boundary_value_analysis

程中，你要做的不是在实际执行时去检查房屋的设计，而是要对房屋的设计执行静态分析。让我们打个比方，例如，事先研究表明起居室的侧面窗户代表了一个弱点，可以砸破这面窗户然后进入房屋，如果是这样的话，那你肯定不希望到时候有一个拿着鸟枪的愤怒房主正站在里面等着你。另一方面，如果采用一种纯的黑盒测试方法来完成破门的话，那么应该在黑夜的掩盖下逐步靠近这个房子，悄悄地尝试所有的门和窗户是否有漏洞，向房子内窥视以决定哪里可能是最好的突破口。最后，如果选择采用模糊测试来完成突破的话，便可以不必研究设计图更不用人工测试各种锁。要做的就是让破门而入的过程自动化——这就是强制性的漏洞发掘方法！

2.2 模糊测试的历史

我们所知道的对模糊测试最早的引用可以追溯到1989年。Barton Miller教授（被很多人认为是模糊测试之“父”）和他的高级操作系统课开发和使用了一个原始的模糊器，用来测试UNIX应用程序的健壮性[一]。测试的关注点并不是评价系统的安全性，而是验证全部代码的质量和可靠性。尽管在该项研究的过程中提及了安全性考虑的问题，但是在对setuid应用程序的测试中并没有专门强调这一点。1995年的研究发现模糊测试让setuid应用程序的可靠性得到了改进，但是该应用程序仍然被认为“具有较高的失效概率”。

Miller的小组所用的模糊测试方法是非常粗糙的。如果应用程序崩溃或者挂起，那么就认为测试失败，否则就认为测试通过。所用的方法是简单地向目标应用程序抛出随机字符串，这是一种纯黑盒的方法。尽管这种方法看上去可能过于简单，但是不要忘了，在那个时候甚至还没有人听说过模糊测试的概念。

大约在1999年，Oulu大学开始进行PROTOS测试集的开发工作。通过首先分析协议规约然后产生违背规约或者很有可能让协议实现无法正确处理的报文，各种各样的测试集先后开发出来。一开始时，产生这样的测试集需要耗费相当大的努力，但是一旦产生之后，这些测试集就可以用来测试多个供应商的产品。这个例子属于白盒和黑盒混合的测试方法，标志着模糊测试发展历程中的一个重要里程碑，由于后来证明大量的故障是用这样的方法发现的。

2002年Microsoft为PROTOS[二]提供了资金支持，2003年PROTOS组的成员开发了Codenomicon，这是一个专门致力于设计和产生商业用模糊测试集的公司。该公司今天的产品仍然基于最初的Oulu测试集，不过又包括了一些图形用户界面、用户支持以及通过健康特征检测而实现的故障检测能力，此外还增加了其他一些特征[三]。关于Codenomicon和其他商业模糊测试解决方案的更多信息，请阅读本书的第26章。

随着PROTOS在2002年逐渐成熟，Dave Aitel发布了一个开放源代码的模糊器，名为SPIKE[四]，采用了GNU的许可（GPL）。Aitel的模糊器实现了一种基于块的方法[五]，目的是为了

㊀ http://www.cs.wisc.edu/~bart/fuzz/

㊁ http://www.ee.oulu.fi/research/ouspg/protos/index.html

㊂ http://www.codenomicon.com/products/features.shtml

㊃ http://immunityinc.com/resources-freesoftware.shtml

㊄ http://www.immunityinc.com/downloads/advantages_of_block_based_analysis.html

测试基于网络的应用程序。SPIKE采用了一种比Miller的模糊器更高级的方法，最显著的特征是包括了描述可变长数据块的能力。此外，SPIKE不仅能够生成随机数据，而且包括了一个数据值的库，库中的数据值有较大可能性让编写较差的应用程序产生故障。SPIKE还包括一组预定义的函数，这些函数能够产生常见的协议和数据格式，其中包括Sun RPC和Microsoft RPC。它们是两个常见的通信协议，过去发现的许多安全漏洞都起源于它们。作为第一个允许用户无痛创建自定制模糊器的公开框架，SPIKE的发布标志着一个重要的里程碑。这个框架在本书中将被反复提及多次，包括本书的第21章“模糊测试框架”。

大致在SPIKE发布的同一时间，Aitel还发布了一个名为sharefuzz的本地UNIX模糊器。与Miller的模糊器不同，Sharefuzz以环境变量为目标而不是以命令行参数为模糊测试目标。Sharefuzz还采用了一个有用的技术来促进模糊测试过程。它使用共享库来挂起函数调用，这些函数返回环境变量值，它们是一些长字符串，为的是发现缓冲区溢出漏洞。

在SPIKE发布之后的大多数模糊测试技术是以不同类型的具体模糊测试工具出现的。Michal Zalewski㊀（又名lacamtuf）于2004年发布的mangleme㊁关注Web浏览器的模糊测试，mangleme是一个公共网关接口（CGI）脚本，可用来持续产生畸形的、在一个目标Web浏览器中被反复刷新的HTML文件。随后很快又出现了几个以Web浏览器为测试目标的模糊器。H.D. Moore和Aviv Raff开发了Hamachi㊂，用来对动态HTML脚本进行模糊测试以查找其实现缺陷，他们两个后来与Matt Murphy和Thierry Zoller合作又推出了CSSDIE㊃，这是一个对重叠样式表进行解析的模糊器。

文件模糊测试的兴起始于2004年。那一年Microsoft发布了MS04-28安全公告㊄，详细描述了负责处理JPEG文件的引擎中的一个缓冲区溢出漏洞。尽管这并不是第一个被发现的文件格式方面的安全漏洞，但是它还是吸引了人们的注意力，因为许多流行的Microsoft应用程序都共享这段有漏洞的代码。文件格式安全漏洞对网络的保护者也提出了挑战。尽管相当数量的类似漏洞在随后几年相继出现，但是如果我们仅仅阻止有可能存在漏洞的文件格式进入公司的网络，那样显然是不切实际的。图像和媒体文件构成了Internet流量中相当大的一部分。如果Web离开了图像的话，那将是多么令人振奋的新闻？此外，这些漏洞中的大多数在不久后就会折磨Microsoft的Office文档文件——这些文件类型对任何企业都至关重要。文件格式安全漏洞最后成为基于变异的模糊测试要发现的主要侯选漏洞，因为漏洞的样品容易被获得，并且文件格式可以在监视目标应用程序故障的同时快速地连续变异。我们在Black Hat USA Briefings in 2005㊅上做了陈述并发布了一系列同时基于变异和生成的工具，用于对文件格式进行模糊测试，包括FileFuzz、SPIKEfile和notSPIKEfile㊆。

㊀ http://lcamtuf.coredump.cx/

㊁ http://lcamtuf.coredump.cx/mangleme/mangle.cgi

㊂ http://metasploit.com/users/hdm/tools/hamachi/hamachi.html

㊃ http://metasploit.com/users/hdm/tools/see-ess-ess-die/cssdie.html

㊄ http://www.microsoft.com/technet/security/bulletin/MS04-028.mspx

㊅ http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-sutton.pdf

㊆ http://labs.idefense.com/software/fuzzing.php

2005年，一个名为Mu Security的公司开始开发一种硬件模糊装置，其目的是让网络中传输的协议数据发生变异[㊀]。这个商业产品供应商的出现，恰好吻合了当前人们关注模糊测试的潮流。我们开始看到日益增加的可用的商业模糊测试解决方案，同时伴随着模糊测试技术的自由软件解决方案的出现。另外，大量的开发者和安全领域的研究人员开始对模糊测试感兴趣，一个明显的证据是他们创建了模糊测试邮件列表[㊁]，该邮件列表由Gadi Evron维护。只有时间才能说明未来将会有什么样的创新在等待着我们。

ActiveX模糊测试在2006年开始流行，当年David Zimmer发布了COMRaider，H.D.Moore发布了AxMan[㊂]。这两个工具关注那些在访问Microsoft Internet Exploer浏览器时能够被Web应用程序实例化的ActiveX控件。此类应用程序中的可被人远程利用的漏洞是一种高风险，因为此类应用程序用户群体十分庞大。就像人们最后发现的那样，ActiveX控件是一种优秀的模糊测试目标，由于其中包括了接口、函数原型和实际的控制逻辑，从而允许对其实施自动化的智能测试。有关ActiveX和浏览器的模糊测试将在第17章和第18章中做更进一步的研究。

尽管模糊测试在历史上还有许多其他里程碑和标志性成果，不过我们认为前面介绍的内容已经足够绘制一幅综合的历史发展路线图。模糊测试的简要发展历史的图形表示参见图2-1。

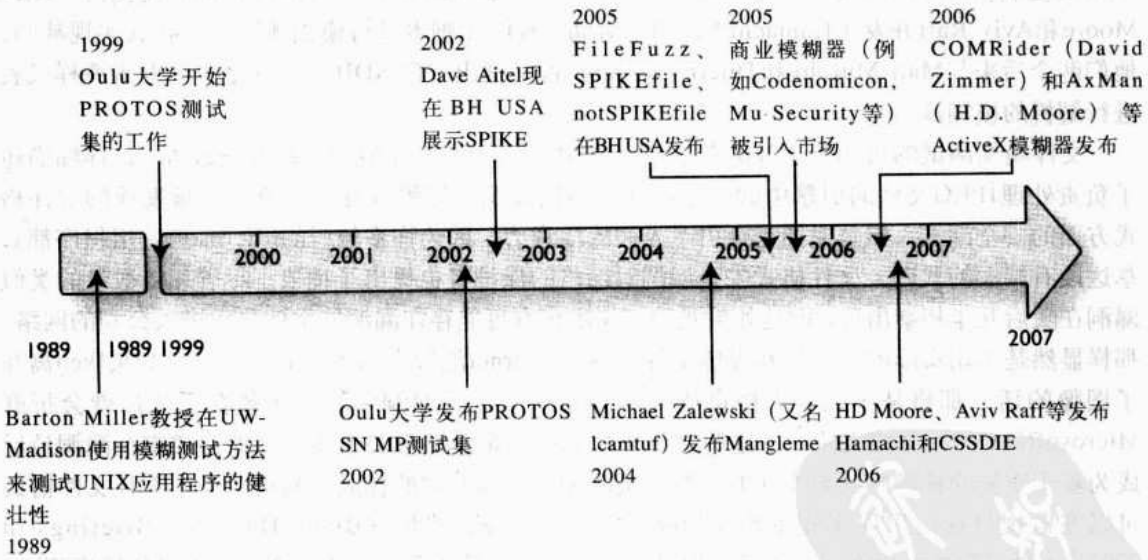


图2-1　模糊测试的历史

尽管到目前为止模糊测试取得了一定的进展，但这项技术仍然是襁褓中的婴儿。到现在为止大部分工具仍然是相对较小的项目，由几个人组成的小组甚至是一名程序员来维护。仅仅在最近几年，一些早期的研究成果才开始步入商业发展阶段。尽管这些发展让人们对模糊测试已

㊀ http://www.musecurity.com/products/overview.html

㊁ http://www.whitestar.linuxbox.org/mailman/listinfo/fuzzing

㊂ http://metasploit.com/users/hdm/tools/axman/

经成为独立的研究领域增加了信心，但是也预示着模糊测试未来将会有许多革新和发展，随着未来的几年人们对这一领域投入的日益增加，模糊测试还会达到新的里程碑。

在下一节，我们将考察执行一次完整的模糊测试审核所要经历的几个不同阶段。

2.3 模糊测试阶段

模糊测试方法的选择依赖不同的因素，可能有很大的变化。没有一种绝对正确的模糊测试方法。模糊测试方法的选择完全取决于目标应用程序、研究者的技能，以及需要测试的数据所采用的格式。然而，无论要对什么进行测试，也不论确定选择了哪种方法，模糊测试总要经历几个基本的阶段。

1. 识别目标。在没有考虑清楚目标应用程序的情况下，不可能对模糊测试工具或技术作出选择。如果是在安全审核的过程中对内部开发的应用程序进行模糊测试，目标应用程序的选择应该小心谨慎。相反，如果是对第三方应用程序进行安全漏洞的发掘研究，这种选择就有一定的灵活性。在识别目标应用程序的时候，需要考察开发商过去被发现的安全漏洞的相关历史。这可以通过浏览安全漏洞收集站点来完成，例如SecurityFocus[⊖]或Secunia[⊜]。一个开发商如果在过去的安全漏洞历史记录方面表现不佳，那么很可能有比较差的编码习惯，最终将会导致更多的安全漏洞被进一步发现。选择了目标应用程序之后，还必须选择应用程序中具体的目标文件或库。如果确要选择目标文件或库，应该选择那些被多个应用程序共享的库，因为这些库的用户群体较大，出现安全漏洞的风险也相应较高。

2. 识别输入。几乎所有可被人利用的漏洞都是因为应用程序接受了用户的输入并且在处理输入数据时没有首先清除非法数据或执行确认例程。枚举输入向量对模糊测试的成功至关重要。未能定位可能的输入源或预期的输入值对模糊测试将产生严重的限制。在查找输入向量同时应该运用水平思考。尽管有一些输入向量是很明显的，但是其他一些则难以捉摸。最后要说的是，任何从客户端发往目标应用程序的输入都应该被认为是输入向量。这些输入包括消息头、文件名、环境变量、注册键值等等。所有这些都应该被认为是输入向量，因此都应该是可能的模糊测试变量。

3. 生成模糊测试数据。一旦识别出输入向量，模糊测试就必须被生成。如何使用预先确定的值、如何变异已有的数据或动态生成数据，这些决策将取决于目标应用程序及其数据格式。不管选择了哪种方法，这个过程中都应该引入自动化。

4. 执行模糊测试数据。这一步与前一步并行进行，在这一步，模糊测试成为一个动词。执行过程可能包括发送数据包给目标应用程序、打开一个文件或发起一个目标进程。同样，这个过程中的自动化也是至关重要的。没有自动化，我们便无法执行真正的模糊测试。

5. 监视异常。在模糊测试过程中，一个至关紧要但却被经常忽视的步骤是对故障或异常的监视过程。举一个例子，如果我们没有办法准确指出是哪一个数据包引起崩溃的话，那么向目

⊖ http://www.securityfocus.com/

⊜ http://secunia.com

标Web服务器发送10000个模糊测试数据包并最终导致服务器崩溃便是一次无用的努力。监视可以采用多种形式并且应该不依赖目标应用程序和所选择的模糊测试类型。

6. 确定可利用性。一旦故障被识别，因审核的目标不同，还可能需要确定所发现的bug是否可被进一步利用。这典型地是一个人工过程，需要具备安全领域的专业知识。因此执行这一步的人可能不是最初执行模糊测试的人。

图2-2以图形方式展示了模糊测试要经历的各个阶段。

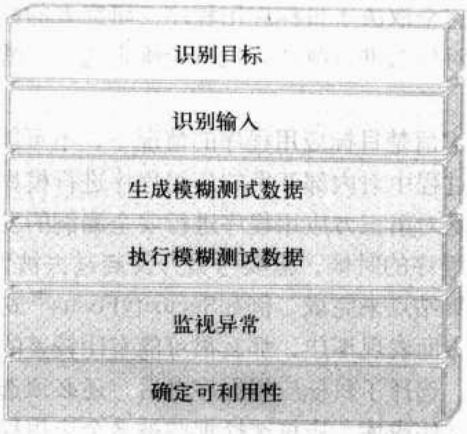


图2-2　模糊测试的阶段

不管采用什么类型的模糊测试，所有上述阶段都应该考虑到，只有确定可利用性这一步有可能例外。各个阶段的顺序和侧重点可依据研究者的目标而改变。尽管模糊测试非常强大，但绝不意味着它能发现任何被测软件百分之百的错误。在下一节，我们将罗列一些漏洞的类型，这些类型的漏洞很有可能测试不到。

2.4　模糊测试的局限性和期望

模糊测试本质上有助于发现目标软件中一些类型的漏洞。同时它对发现某些类型的漏洞也存在一定的局限性。在这一节里，我们将给出几类典型的漏洞，这些漏洞无法被模糊器发现。

2.4.1　访问控制缺陷

一些应用程序需要分层的权限来支持多个账户或用户级别。例如，考虑一个在线日历系统，它可能通过一个Web前端访问。该应用程序可能指定一名管理员来控制谁允许登录到系统。可能还存在另一个专门的用户组，该组能够创建日历。所有其他用户可能只有读的权限。最基本的访问控制形式是确保常规用户不能执行管理员的任务。

模糊器可能会发现日历软件系统中的一个故障，该故障允许一个攻击者获得对系统的完全控制。在较低的层次上，软件故障通常是跨越不同目标的，因此相同的逻辑可被用于检测不同的目标。在测试过程中模糊器可能会成功地访问只有管理员才能访问的功能，但是它却只拥有

常规用户的访问身份。这种访问控制机制被绕过的事实并不大可能会被检测出来。为什么？因为我们必须考虑到模糊器并不理解应用程序的逻辑这样一个事实。模糊器并没有办法知道管理员的访问区域是否能被常规用户所访问。在模糊器中实现应用程序逻辑感知能力并不是不可行，但是要付出极大的代价，并且在测试其他应用程序时如果不做大的修改便很可能无法被重用。

2.4.2 设计逻辑不良

模糊器不是用来识别设计逻辑不良的最好的工具。例如，考虑在Veritas Backup Exec中发现的漏洞，该漏洞允许攻击者远程访问Windows Server的注册表，包括创建、修改和删除注册表键值[⊖]。这样的访问几乎不会立即让系统的安全发生妥协。这个漏洞的发现归功于一个侦听服务，该侦听服务在一个传输控制协议（TCP）端点上实现了一个远程过程调用（RPC）接口，该接口不需要认证但是仍然接受操作注册表的命令。这并不是一个认证问题，因为根据软件设计规约，该过程根本不需要认证。但是，这却是一个不恰当的设计决策，该决策很有可能源自一个因为某个假设而引起的失效，该假设认为攻击者愿意花时间破解用来描述目标例程的Microsoft Interface Description Language （IDL）并随后编写了一个自定制的实用程序用来与服务器通信。

尽管模糊器可能会发现通过RPC连接而提供的数据确实存在解析问题，从而导致某种形式的底层故障，但是却不能确定所展示出的功能是否不安全。这个观点在本书第18章“Web浏览器模糊测试：自动化”讨论ActiveX的模糊测试时将起到重要作用，因为许多控件只为攻击者提供操作方法，例如创建和执行文件的方法。检测这些设计上的Bug还需要专门的考虑。

2.4.3 后门

对于一个模糊器来说，如果受到限制或不知道关于目标应用程序的结构信息，它的后门（backdoor）看上去和其他目标应用程序的逻辑没有什么区别，例如一个登录屏幕界面。它们都是简单的输入变量，接收身份认证秘钥。不仅如此，除非给予模糊器组的信息来让它认可成功的登录，它将没有办法通过使用一个硬编码的密码来识别一个成功的登录企图。例如，在对密码域进行模糊测试时，模糊器典型地可以发现输入的异常，但是却不能发现随机猜测的硬编码的密码。

2.4.4 内存破坏

内存破坏问题通常会导致目标进程崩溃。这种类型的问题可以根据与拒绝服务类似的症状而加以识别。然而，一些内存破坏问题被目标应用程序很好地屏蔽掉了，因此简单的模糊器永远不会发现它们。例如，一个格式字符串漏洞可能不会被发现，除非为目标应用程序附加一个调试器。格式字符串漏洞通常归结于机器代码层的违例指令。例如，在一个X86的Linux平台的机器中，如果格式字符串中带有一个%n的攻击字符特征，那么经常会在执行下列指令时发生故障：

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

⊖ http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=269

如果模糊器向带有某种格式触发字符串（例如这里或那里的%n）的进程发送随机数据，eax寄存器在许多情况下将不包含一个可写的地址，而是包含一些来自栈的垃圾字。在这种情况下，该指令将会触发一个段违例信号SIGSEGV。如果应用程序有一个SIGSEGV信号处理器，那么它将简单地杀死这个进程然后派生一个新的进程。对于某些应用程序，信号处理器甚至会企图允许过程继续执行而不需要重新启动。对于一个格式字符串漏洞，这实际上是可能的（尽管我们极力反对这么做），因在SIGSEGV信号没有发生前内存实际上并没有遭到破坏。因此，如果我们的模糊器没有检测到它又如何呢？这个进程不会崩溃，相反它会恰当通过信号处理器来处理信号，因此这里并不存在安全问题，对吗？但是，不要忘记，对格式字符串的成功利用将造成这样一个结果：在内存中写入数据并随后使用这些受到控制的内存区域获得对进程的控制权。信号处理器不能阻挡格式字符串被他人精确地利用，同样精确地利用也可能不会引起任何故障。

因此，在模糊测试目标的监视范围不同的情况下，模糊器也可能捕捉不到这种情况。

2.4.5 多阶段安全漏洞

可利用性并非总意味着简单地对一个弱点的攻击。复杂的攻击通常涉及连续利用若干个漏洞来让机器妥协。也许一个初始的缺陷让机器接受了未被授权的访问，随后的缺陷让这种未授权被进一步放大。模糊测试对识别单个的缺陷是有用的，但是通常对那些由小的漏洞链而组成的大缺陷链却并没有多少价值，对那些让人不感兴趣的多重输入向量而引起的攻击也是如此。

2.5 小结

尽管模糊测试的发展已经有一段时间，但是这种方法学仍然未被安全研究领域之外的人们所广泛理解。出于此种原因，你很可能会发现，给模糊测试所下的定义数量与你询问的人数一样多。我们就本书的目的而定义了模糊测试的概念，介绍了其简要发展历史，为模糊测试定下了未来的期望，下面将深入研究模糊测试的类型和具体方法。

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书藉杂志全免 KEY 提供,，完全无加密。

第3章 模糊测试方法和模糊器类型

"太多的好文档从企业中流失，太多的妇产医生不能与全国的妇女进行爱的实践。"

——George W. Bush, Poplar Bluff, Mo，2004年9月6日

模糊测试定义了一种发现漏洞的综合方法。然而，在模糊测试这把大伞下，有各种不同的具体方法来实现模糊测试方法学。本章开始对模糊测试进行详细剖析，考察各种不同的具体的模糊测试方法。我们还将考察不同类型的模糊测试如何实现这些方法以及如何利用这些方法来发现具体目标中的漏洞。本书的第二部分将对每种模糊器类型做进一步的专门探讨。

3.1 模糊测试方法

前一章曾提到，模糊器可分为两大类：基于变异的模糊器（mutation-based fuzzer）对已有数据样本应用变异技术来创建测试用例，基于生成的模糊器（generation-based fuzzer）通过对目标协议或文件格式建模来从头创建测试用例。在这一节，我们将对这两类模糊器进一步划分子类。当然，子类的划分并没有公认的结果，但是鉴于本书是第一部关于模糊测试的专著，我们把模糊器分为下面介绍的5类。

3.1.1 预先生成测试用例

如前一章所述，这是PROTOS框架采用的方法。测试用例的开发始于对一个专门规约的研究，其目的是为了理解所有被支持的数据结构和每种数据结构可接受的值范围。硬编码的数据包或文件随后被生成，以测试边界条件或迫使规约发生违例。这些测试用例可用于检验目标系统实现规约的精确程度。创建这些测试用例需要事先完成大量的工作，但是其优点是在测试相同协议的多个实现或文件格式时用例能够被一致地重用。

预先生成测试用例的一个缺点是，这种方式下的模糊测试存在固有的局限性。由于没有引入随机机制，一旦测试用例表中的用例被用完，模糊测试只能结束。

3.1.2 随机方法

这种方法是迄今为止最低效的，但是它可用于快速发现目标软件是否是彻头彻尾的恐怖代码。随机方法只是简单地大量产生伪随机数据给目标软件，希望得到最好的或最坏的结果，这取决于测试者所持的观点是什么。下面是这种方法的一个简单例子，它总是让人感到幽默，不过却很有效：

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

上面的一行命令从Linux urandom设备读取随机数据，然后用netcat命令将这些数据传输到

一个目标地址及其端口。While循环确保这一过程持续进行，直至用户中断该过程。

信不信由你，影响任务关键软件的漏洞在过去就是使用这样的技术发现的。这是多么令人窘迫！真正的随机模糊测试最困难之处在于如何逆向搜索服务器崩溃时软件发生异常的起因。这种搜索可能是一个令人痛苦的过程，需要回溯跟踪可能引起服务器崩溃的大约500 000个随机字节。你可能希望利用探测器捕获通信数据来帮助减少跟踪范围。此外，在调试器和反汇编器上还将花费大量的时间。使用这个技术拆分栈需要痛苦的调试过程，尤其是调用栈可能遭到破坏。图3-1是与某企业应用系统和网络监视实用程序相连的调试器窗口，被测系统刚刚在一次模糊测试攻击中发生崩溃。你能决定故障的原因吗？很可能无法确定原因，除非你是一个占星术士。精确地指出问题的起因还需要更多的研究工作。这里需要注意的是，一些数据还可能事先经过了混淆以毁灭证据。

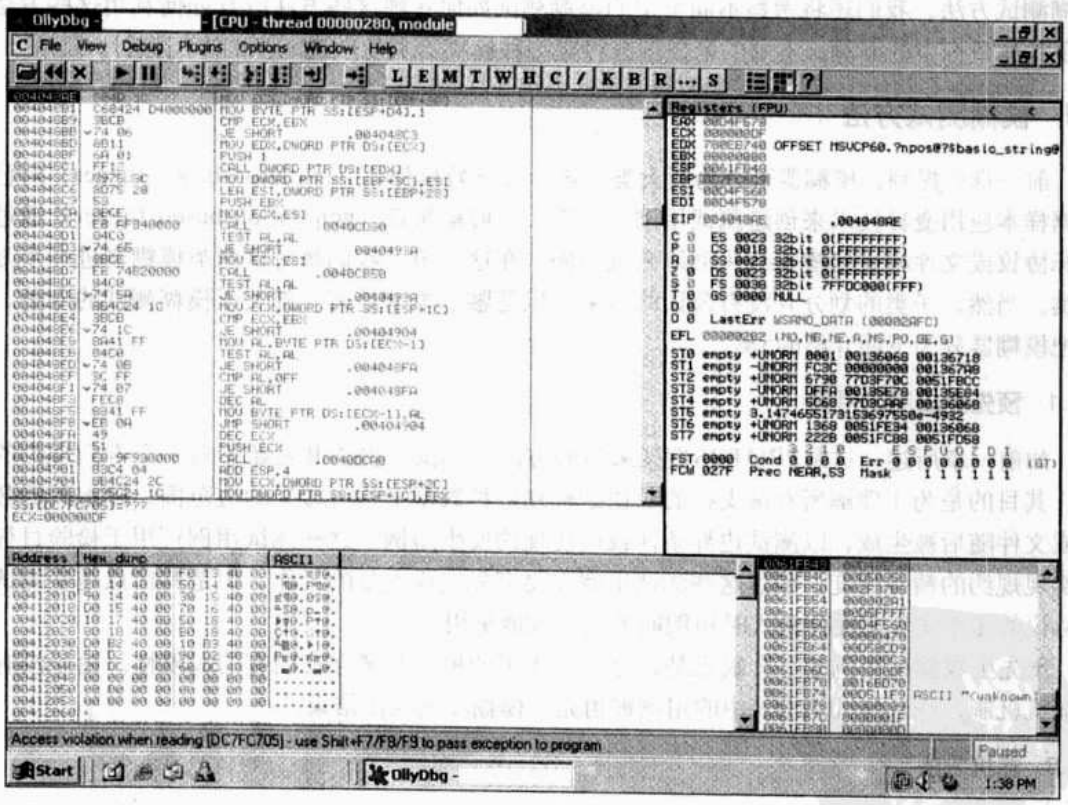


图3-1　现在发生了什么？一个dev/urandom模糊器正处在进退两难的尴尬境地

3.1.3　协议变异人工测试

按理说，人工协议测试应该比/dev/urandom方法在技术复杂性方面更加初级。人工协议测试中不需要自动化的模糊器。事实上，研究者本人就是模糊器。在加载了目标应用程序后，研究者仅仅通过输入不恰当的数据来试图让服务器崩溃或使其产生非预期的行为。这就是他的模

糊测试方法，但是在过去毫无疑问是有效的。这种方法有它的优点，优点当然是分析员本人在审核过程中可以发挥他的历史经验和“天生直觉”。此类模糊测试方法最常用来测试Web应用程序。

3.1.4 变异或强制性测试

这里所说的强制性，是指模糊器从一个有效的协议或数据格式样本开始，持续不断地打乱数据包或文件中的每一个字节、字、双字或字符串。这是一个相当早期的方法，因为这种方法几乎不需要事先对被测软件进行任何研究，实现一个基本的强制性模糊器也相对简单直接。模糊器要做的全部事情就是修改数据然后传递它。当然，在故障检测过程中可以引入许多“铃声和口哨声”，还可以引入登录功能等作为辅助工具，这种工具一般可以在很短的时间内完成开发。

此种方法带有几分低效性，因为许多CUP周期被浪费在数据生成上，并且这些数据并不能立刻得到解释。然而，这些问题所带来的挑战在一定程度上可以得到缓解，因为测试数据生成和发送的全过程都可以自动完成。使用强制性方法的代码覆盖依赖于已知的经过测试的良好的数据包或文件。大部分协议规约或文件定义都比较复杂，即使对它进行表面的测试覆盖也需要相当数量的样本。强制性文件格式模糊器的例子包括FileFuzz和notSPIKEfile，分别对应Windows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测试方法。在这种方法中，需要进行前期的研究，首先要理解和解释协议规约或文件定义。然而，与前一种方法不同，这种方法并不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。为了达到这个目的，需要识别数据包或文件中的静态部分和动态部分，后者代表了可被模糊的变量。之后，模糊器动态解析这些模板，生成模糊测试数据，然后向被测目标发送模糊后产生的包或文件。这种方法的成功依赖研究者的能力，研究者需要指出规约中最容易导致目标软件在解析时发生故障的位置。此类模糊器的例子包括SPIKE和SPIKEfile。这两个模糊器都以SPIKE脚本来描述目标协议或文件格式，并使用一个模糊引擎来创建模糊后的数据。这种方法的不足之处是需要耗费一定的时间来产生文法或数据格式的定义。

3.2 模糊器类型

前面定义了几种不同的模糊测试方法或手段，下面让我们看一些具体的模糊器类型。模糊器的不同类型取决于被测的目标程序。不同的目标需要采用不同的模糊测试方法并且每种模糊测试方法都有其适合的模糊器。

在这一节，我们将简要考察一些不同的模糊器类型。本书后面会逐一介绍每一类模糊器的细节，包括它们的背景知识、自动化技术和开发样例。

3.2.1 本地模糊器

在UNIX世界中，setuid应用程序允许一个普通用户临时获得更高的权限。这一功能使该应用程序成为一个显然的模糊测试目标，因为setuid应用程序中的任何漏洞都将使一个用户永久提升权限并执行他选择的程序。对setuid进行模糊测试有两个目标。第一个目标是命令行参数的模糊测试，关注的是在命令行中如何向setuid传递变形的参数。第二个目标也是传递变形的参数，不过采用不同的方式。第二种目标下的变形参数是通过UNIX Shell环境传递的。下面让我们分别了解命令行和环境变量的模糊测试。

命令行模糊器

当一个应用程序开始运行时，通常要处理用户传递给它的命令行参数。来看下面的例子，这个例子在第1章“安全漏洞发掘方法学”中也曾使用过，该例子说明了命令行参数栈溢出的最简单形式。

```
#include <string.h>

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

如果这个程序真的是setuid，可以用本地模糊器对其测试。这里的问题是在源代码不可获得的情况下，安全研究者如何才能在setuid应用程序中找到更难以发现的bug？如果你听到解决这个问题最好的办法是模糊测试，不应该感到吃惊。

下面是一些有用的命令行模糊器：

- warlock的clfuzz[⊖]。一个命令行模糊器，可用于测试应用程序中的格式字符串和缓冲区溢出漏洞。
- Adam Greene的iFUZZ[⊜]。一个命令行模糊器，可用于测试应用程序中的格式字符串和缓冲区溢出漏洞。包括对若干个不同参数的模糊选项，并可根据应用程序的“使用”帮助信息智能地产生模糊测试数据从而对应用程序执行测试。

环境变量模糊器

另一类本地模糊器涉及环境变量向量，它也以setuid应用程序为目标。考虑下面的应用程序，该程序不安全地使用了来自用户环境中的值：

```
#include <string.h>

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

⊖ http://warl/0ck.metaeye.org/cb/clfuzz.tar.gz

⊜ http://fuzzing.org

有好几种有效的方式可以对一个应用程序中的环境变量的使用进行模糊测试，但是并没有多少自动化的支持工具，尽管这个过程很简单。许多安全研究者试图自行拼凑脚本来完成这任务，这也是公开发布的支持工具较少的部分原因。充其量，本地环境变量模糊器并不是复杂到难以开发，因此许多这样的工具被开发并公开发行。下面是一些有用的环境变量模糊器：

- Dave Aitel的Sharefuzz[㊀]。第一个公开发布的可用的环境变量模糊器，它中途拦截对getenv函数的调用消息并且返回恶意数据。
- Adam Greene的iFUZZ[㊁]。尽管它主要是一个命令行模糊器，iFUZZ包还包括了基本的环境变量模糊能力。iFUZZ使用与Sharefuzz相同的方法，并且比后者在使用过程中更容易被定制。

在本书的第7章和第8章中，我们将详细讨论本地模糊测试和setuid的有关概念。

文件格式模糊器

大量的应用程序必须在某个时间点处理文件的输入输出，不论是客户端还是服务器端都是如此。例如，反病毒网关通常需要解析压缩了的文件以诊断其中包含的内容。另一个例子是Office Productivity软件套件，它需要打开文档。当这些类型的应用程序解析到恶意编撰的文件时，就有理由怀疑被分析的软件存在安全漏洞。

文件格式模糊测试的目的即在于此。文件格式模糊器动态创建变形了的文件，然后利用目标应用程序发起和处理这些文件。尽管文件模糊测试的具体方法严格讲与其他类型的模糊器存在不同，但是它们的总体思路是一致的。一些有用的文件格式模糊测试工具包括：

- Michael Sutton的FileFuzz[㊂]。一个基于Windows图形用户界面（GUI）的文件格式模糊测试工具。
- Adam Greene的notSPIKEfile和SPIKEfile[㊃]。基于UNIX的文件格式模糊测试工具，其中的一个基于SPIKE，另一个则不基于SPIKE。
- Cody Pierce的PAIMEIfilefuzz[㊄]。类似于FileFuzz，它是另一个基于Windows GUI的文件格式模糊测试工具，构建于PaiMei逆向工程框架的基础之上，本书后面将对PaiMei做更详细的介绍。

文件格式模糊测试的内容将在本书的第11章、第12章和第13章等详细介绍。

3.2.2 远程模糊器

远程模糊器以侦听一个网络接口的软件为测试目标。基于网络的应用程序很可能是模糊测试最重要的测试目标。随着Ineternet的发展，几乎所有的公司现在都会公开发布一些可被访问的服务器来提供Web页、e-mail、域名系统（DNS）解析服务，等等。这类软件中的漏洞为攻

㊀ http://www.immunitysec.com/resources-freesoftware.shtml
㊁ http://fuzzing.org
㊂ http://fuzzing.org
㊃ http://fuzzing.org
㊄ https://www.openrce.org/downloads/details/208

击者提供了访问敏感数据或对更可信的服务器发起攻击的机会。

网络协议模糊器

网络协议模糊器可以被分为两个主要的类：以简单协议为测试目标的模糊器和以复杂协议为测试目标的模糊器。下面将分别介绍这两类模糊器的主要特征。

简单协议

简单协议通常只有简单的认证或根本没有认证。它们通常基于可打印的ASCII文本，而不是二进制数据。简单协议不包括长度或校验和字段。此外，典型的简单协议应用程序并不包含很多的状态。

简单协议的一个例子是FTP。在FTP协议中，所有的受控信道中的通信多是以普通ASCII码文本的形式传输的。就认证而言，FTP只需要简单文本形式的用户名和密码认证。

复杂协议

复杂协议典型地要包含二进制数据，偶尔包含人可读的ASCII码字符串。认证可能需要通过加密或某中形式的混淆来实现，其中可能包括多个复杂的状态。

复杂协议的一个例子是Microsoft远程调用（MSRPC）协议。MSRPC是一个二进制协议，在数据传输之前需要执行若干个步骤才能建立起一个通信信道。协议需要长度描述域和分解域。总体上讲，它不是一个很容易实现模糊测试的协议。下面是几个网络协议模糊测试的有用工具：

- Aitel的SPIKE[㊀]。SPIKE是第一个公开发布的模糊测试框架。它包括好几种流行协议的预生成的模糊测试脚本，同时也可以作为一个API来使用。
- Michael Eddington的Peach[㊁]。一个用Python编写的跨平台的模糊测试框架。它非常灵活并且可用来对几乎所有网络目标应用程序进行模糊测试。

网络协议模糊测试的内容将在本书的第14章“网络协议模糊测试”、第12章“网络协议模糊测试：UNIX平台上的自动化测试”和第13章“网络协议模糊测试：Windows平台上的自动化测试”等章节详细介绍。

Web应用模糊器

Web应用程序已经成为访问后端服务的一种流行的便捷方式，这些后端服务包括电子邮件和计账系统等。随着Web2.0的到来（先不论它究竟是什么），诸如文字处理等传统的桌面应用程序也开始向Web上转移[㊂]。

在对Web应用程序进行模糊测试时，研究者主要考察Web应用所特有的漏洞，例如SQL注入、交叉站点脚本（XSS）等。这使得模糊器必须具备通过超文本传输协议（HTTP）进行通信以及通过捕获网络响应来分析安全漏洞的能力。下面是一些有用的Web应用模糊测试工具：

- OWASP的WebScarab[㊃]。一个开放源代码的Web应用程序审核套件，具有模糊测试能力。

㊀ http://www.immunitysec.com/resources-freesoftware.shtm

㊁ http://peachfuzz.sourceforge.net/

㊂ http://www.google.com/a/

㊃ http://www.owasp.org/index.php/Fuzzing_with_WebScarab

- SPI Dynamics的SPI Fuzzer[㊀]。一个商业的HTTP和Web应用程序模糊器，它包含在WebInspect漏洞扫描器之内。
- Codenomicon的Codenomicon HTTP Test Tools[㊁]。一个商业用HTTP测试套件。

Web应用模糊器在本书的第9章“Web应用和服务器模糊测试”和第10章“Web应用和服务器模糊测试：Windows平台上的自动化测试”等章节详细介绍。

Web浏览器模糊器

尽管Web浏览器模糊器在技术上只是一类特殊的文件格式模糊器，但是由于基于Web的应用程序的普及性，我们认为应该将它专门列为一类。Web浏览器模糊器通常利用HTML的功能来使模糊测试的过程自动化。例如，lcamtuf的mangleme实用程序是最先公开可用的浏览器模糊测试支持工具之一，它利用<META REFESH>标签以自动的方式连续加载测试用例。这种Web浏览器具有的独特特征使得客户端的模糊器能够完全被自动化，并且不需要对应用程序进行任何复杂的包装。而对其他客户端的浏览器，这种包装则是典型的需求。

Web浏览器模糊测试不仅仅限于对HTML文本进行解析，还有许多其他的测试目标。例如，See-Ess-Ess-Die模糊测试工具能够对重叠样式表进行解析，另一个模糊测试工具COM Raider则关注那些能够被加载进Microsoft Internet Explore的组件对象模型（COM）对象。其他可模糊测试的元素包括图形和服务器应答头域。下面是几个有用的Web浏览器模糊测试工具：

- lcamtuf的mangleme[㊂]。第一个公开可用的HTML模糊器。它是一个CGI脚本，可反复发送打乱了的HTML数据到一个浏览器。
- H.D. Moore和Aviv Raff的DOM-Hanoi[㊃]。一个DHTML模糊器。
- H.D. Moore和Aviv Raff的Hamachi[㊄]。另一个DHTML模糊器。
- H.D. Moore、Aviv Raff、Matt Murphy和Thierry Zoller的CSSDIE[㊅]。一个CSS模糊器。
- David Zimmer的COM Raider[㊆]。一个易于使用、GUI驱动的COM对象（ActiveX控件）模糊器。

COM和ActiveX对象的模糊测试将在第17章“Web浏览器模糊测试”和第18章“Web浏览器模糊测试：自动化”中详细介绍。CSS和图形文件的模糊测试并没有在Web浏览器的模糊测试部分专门介绍，但是其中给出的基本概念对CSS和图形文件模糊测试同样适用。

3.2.3 内存模糊器

有些时候在测试过程中存在一些困难阻碍了模糊测试快速有效地进行。这时候内存模糊测

㊀ http://www.spidynamics.com/products/webinspect/index.html
㊁ http://www.codenomicon.com/products/internet/http/
㊂ http://freshmeat.net/projects/mangleme/
㊃ http://metasploit.com/users/hdm/tools/domhanoi/domhanoi.html
㊄ http://metasploit.com/users/hdm/tools/hamachi/hamachi.html
㊅ http://metasploit.com/users/hdm/tools/see-ess-ess-die/cssdie.html
㊆ http://labs.idefense.com/software/fuzzing.php#more_comraider

试就可能会派上用场。内存模糊测试的基本思想是简单的，但是开发一个合适的实现工具却不是一件容易的事。有一种实现方法涉及到冻结进程并记录它的快照，以及快速将故障数据注入进程的输入解析例程。每个测试用例执行之后，以前所记录的快照被存储下来并且注入新的数据。上述过程反复进行直至所有测试用例都被穷尽。和其他模糊测试方法一样，内存模糊测试也有优点和缺点。它的优点包括：

- 速度。不仅没有网络带宽需求，并且任何位于接收离线数据包和实际解析数据包之间的不重要的代码都可以被删去，从而使得测试性能有所提高。
- 捷径。有时协议程序使用了定制的加密或压缩算法，或者其中充满了校验和确认代码。内存模糊器能够记录下解压、解密、或校验和确认之后某一时间点上的内存快照，而不是花费大量的时间来测试所有的过程，减少了测试的工作量。

内存模糊测试的缺点是：

- 假像。由于原始数据被注入到进程地址空间，不仅没有网络带宽需求，并且任何位于接收离线数据包和实际解析数据包之间的不重要的代码都可以被删去，从而使得测试性能有所改进。
- 重现。尽管引起一个异常可能会展示出一个可被人利用的漏洞，但是研究者仍然需要在进程之外远程创建异常。这可能会花费不少时间。
- 复杂。正如我们在第19章“内存模糊测试”和第20章“内存模糊测试：自动化”中将要看到的，这种模糊测试方法的实现极为复杂。

3.2.4 模糊器框架

模糊测试框架可用于对不同目标的模糊测试。模糊测试框架其实就是一个通用的模糊器或模糊器库，它简化了许多不同类型的测试目标的数据表示。到目前为止前面提到的许多模糊器实际上是模糊测试框架，包括SPIKE和Peach。

典型地，模糊测试框架包括一个库用来产生模糊测试字符串或者通常能够导致解析例程出现问题的数据值。此外，典型的模糊测试框架还包括一组例程以简化网络和磁盘输入输出。模糊测试框架还应该包括某种脚本类语言，以便在创建具体的模糊器时使用。模糊测试框架是流行的，但它绝对不是完全充分的解决方案。使用或设计一个模糊测试框架也有它的优点或缺点。优点包括：

- 可重用性。如果创建了一个真正通用的模糊测试框架，这个框架便可以在测试不同目标软件时反复使用。
- 群体介入。大项目的群体参与比小项目的群体参与更容易，并且大项目可以被反复使用多次，小项目只有相对狭小的使用空间，例如专门协议的模糊器。

模糊测试框架有下列缺点：

- 局限性。即使对于仔细设计的框架，研究者也很可能遇到不适合该框架的测试目标。这通常令人沮丧。
- 复杂性。获得一个通用的模糊测试接口是好事，但是却要花费时间来学习如何驱动这样

的框架。有时掉转车轮重新开辟模糊测试接口可能在时间上会更高效，但这样做可能会破坏原来的框架。

- 开发时间。对于一个具体的协议来说，设计一个完全通用和可重用的模糊测试框架要比单独为其开发一个程序所需要的工作量大得多。这些工作量的差异几乎总是会在开发时间上有所体现。

一些研究者坚持地认为框架对任何重要的模糊器来说是最好的方法，而另一些人则认为并非如此，每一个模糊器都应该在具体环境下创建。这两种思路的实验验证以及取舍显然要取决于读者自己。

3.3 小结

在执行模糊测试的时候，应该考虑到应用本章所讨论的各种方法的价值。通常情况下，组合各种不同的方法可能会收到最好的效果，因为一种方法可能是另一种方法的补充。

一个模糊器可能要关注不同的目标软件，正因为这样，可能需要多种不同类型的模糊器。在开始学习消化各种各样的模糊器类型背后隐藏的基本概念的同时，应该设想一下如何去实现它们。这将有利于在阅读本书后面要介绍的有关实现的内容时给出中肯的评价。

第4章 数据表示和分析

“我的工作是，诸如，考虑目前能马上做的之外的事情”

——George W. Bush, Poplar Bluff, Mo, 2004年9月6日

计算机在内部和外部通信的各个方面都要使用协议。这些协议构成了数据传输和数据处理的基础结构。如果要进行成功的模糊测试，必须首先对目标软件使用的协议有所理解。有了理解之后才能够针对协议中最有可能引起异常条件的部分进行测试。此外，为了访问到协议中的脆弱部分，在进行模糊测试之前我们还需要提供合法的协议数据。了解开放的和专有的协议必然会使得安全漏洞的发掘更为高效。

4.1 什么是协议

协议对通信来说是必须的。有时候一些协议被定义成标准，其他一些时候协议只是人们遵循的事实上的标准。口语就是一种事实上的标准的典型例子。尽管没有正式的文档定义说话的规则，但是人们一般都能够听懂别人在说什么，听者是沉默的，但是它可以找到机会给出回答。同样，你不会没有理由地简单地走到一个人面前滔滔不绝讲出你的名字、住址甚至其他至关重要的个人信息（更不会说出你的生日和身份证号）。相反，个人信息需要以元数据（metadata）开始。例如在“我的名字是John Simith，我住在Cambers大街345号”这些信息中，说者和听者都应该遵循一种有意义的数据通信方式。在计算领域，协议绝对至关重要。计算机本身并没有智能，更不能奢望它还有直觉。因此，严格地定义协议对通信和数据处理是必不可少的。

百科全书对协议的定义是：“在两个计算端点之间建立或控制连接、通信或数据传输的约定或标准”[⊖]。这些端点可以是两个单独的计算机，也可以是一个计算机中两个单独的点。例如，从内存读取数据时，计算机必须访问硬盘上的存储区，通过数据总线将数据转移到内存，然后再将其传送至处理器。在每个端点，数据必须具有某种形式，以便发送方和接收方都能够恰当地处理数据。在最底层，数据就是一堆比特位的集合。只有在某个上下文中理解，这些比特集合才具有含义。如果发送方和接收方端点不能就上下文达成一致，那么被传输的数据就是无意义的。

机器间的通信也依赖协议。这种通信协议的一个常见的例子就是Internet。Internet是许许多多个分层的协议集合体。位于美国的一台家用桌面计算机可以通过连续的协议对数据进行包装，然后将数据通过Internet传送到中国的另一台桌面计算机。只有理解了相同的协议，接收方计算机才能解包数据并解释它。在上面的特定场景下，中国政府很可能中途拦截了被传输的数

⊖ http://en.wikipedia.org/wiki/Protocol_%28computing%29

据，按照自己的理解解除数据的包装并对其进行了解释。这是因为参与通信的两个机器都能理解这种协议，从而使得双方的通信成为可能。在两个端点之间存在数量众多的路由器，这些路由器也必须理解协议的某些方面以便合理地对数据执行路由转发。

尽管协议起到了一个非常通用的作用，但是却可以采取各种不同的具体形式。一些协议是人可读的，并且以简单文本形式表达。其他一些协议则是以二进制形式传输数据，这些协议的数据格式不适合人们直接理解。诸如表示GIF图像或Microsoft Excel表单的文件格式便是二进制协议的典型例子。

包含一个协议的单独的成分通常叫做域（field）。协议的规约定义了这些域如何被分隔和排序。下面让我们详细考察域的概念。

4.2 协议域

在设计一个协议的时候要作出许多决策，其中最关键的决策之一是协议中的数据如何被分隔为不同的成分。发送方和接收方的机器都需要知道如何解释数据中的个体元素，这些个体元素是由协议定义的。有三种典型的方法可用来应付这个问题：定长域、变长域和分隔域。简单文本协议通常以回车等字符作为分隔符。当客户端或服务器端在接收到的数据中解析到一个回车符，就表明此处是某个命令的结尾。以字符分隔的域在一些文件格式中也常常被使用，例如以逗号分隔的值（CSV）文件，用于表示二维数组数据，这些数据可以被诸如Microsoft Execl这样的电子表单程序所读取。XML是使用字符分隔的域的另一个例子，不过它不是以简单的单字符来表明一个域的结束，XML利用多个字符来分隔文件。例如，XML元素都定义在两个尖括号（<和>）之间，元素属性的名字[nd]值用等号（=）分隔。

定长域预先定义了每个域使用的字节数。这种方法常常在网络协议的头域中被使用，例如以太网、Internet协议（IP）传输控制协议（TCP）和用户数据报文协议（UDP），因为这些协议的头域每次都需要高度结构化的数据。例如一个以太网数据包总是以标识目的媒体访问控制（MAC）地址的6个字节开始，后面跟着额外的定义源MAC地址的6个字节。

为优化目的而选择域

能够规定某些域的大小及位置的优点是规约的作者通过选择灵活的字节排列可以有助于处理器的优化。例如，IPv4头中的许多域方便地按照32位边界来排列。正如Intel Optimization Manual中的第3.41节中所述的，不按排列顺序读取数据对Pentium处理器来说相当耗费资源的。

未对准的访问对Pentium处理器家族需要耗费三个CPU周期。在Pentium Pro和Pentium II处理器上，一个跨越了缓存线的未对准的访问需要耗费6至9个CPU周期。数据缓存单元（DCU）阵列是一个内存访问装置，它跨越了32个字节的线边界。未对准的访问可能会让DCU阵列使Pentium Pro和Pentium II处理器停止运行。为了获得最佳性能，应该确保超过32字节的数据结构和数组按照32字节的数组元素对齐，并且这种对数据结构和数组元素的访问模式不能破坏对齐规则[⊖]。

⊖ http://download.intel.com/design/intarch/manuals/24281601.pdf

相反，对一些精简指令集计算（RISC）体系结构，例如SPARC，在执行不对齐的内存访问时将会彻底失效，抛出致命的总线错误异常。

当数据不太结构化时，可变长域是人们想要的。这种方法通常被用于媒体文件。一个带有很多可选的头和数据成分的图像或视频文件格式可能会相当复杂。可变长域为协议开发者提供了灵活性，同时能够创建一个有效地高效利用内存的协议。可变长域并不是为特定的数据元素设置固定的字节数，所有这些可能都不需要，这种域通常前面是一个表明域的类型的标识符，然后是一个数据值用来说明域中随后包含的数据元素的所占的字节数大小。

4.3 简单文本协议

简单文本协议（plain text protocol）这个术语指的是通信数据的数据类型大部分可被归属于可打印的ASCII字符。这些字符包括所有数字、大小写字母、百分号和美元符，还有回车（\r，十六进制数0x0D），换行符（\n，十六进制数0x0A）、制表符（\t，十六进制数0x09），以及空格（十六进制数0x20），等等。

简单文本协议的设计目的是让人们可以读懂它。简单文本协议通常不如对应的二进制协议效率高，因为它属于内存密集协议，但是有许多场合需要我们设计这种人可读的协议数据。文件传输协议（FTP）的控制信道就是简单文本协议的一个例子。另一方面，数据传输信道能够传输二进制数据。FTP用于上载和下载数据到远程机器上。FTP控制的通信数据是人可读的，这样一个事实使得通信可以通过命令行工具实现。

```
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，使用了流行的Netcat[⊖]工具来人工连接到一个Microsoft FTP服务器。尽管还有许多可用的其他FTP客户端工具，不过Netcat允许我们完全控制发送到服务器的请求命令并显示出协议的文本性质。在这个例子中，所有黑体文本是发送到服务器的请求，常规文本则代表服务器的响应。我们可以清楚地看到这里所有的请求和响应都是人可读的。这使得人们可以准确地看到当时发生了什么，并且能够调试和解决出现的问题。

4.4 二进制协议

二进制协议对于人们来说理解起来更困难，因为这种协议传输的数据不是可读文本，你看

⊖ http://www.vulnwatch.org/netcat/

到的是原始的字节流。如果对协议不理解，协议中的数据包可能就没有专门的意义。对于模糊测试来说，如果测试目标是协议中的某个位置的话，理解协议的结构是必须的。

为了更好的说明如何才能获得对一个二进制协议的恰当的理解以产生一个模糊器，让我们考察一个特征丰富的协议，这个协议现在每天被数百万人所使用，可用来与朋友们进行AOL即时消息（AIM）对话，对话的同时可能他们都在工作。特别地，我们将考察登录会话期间发送和接收的数据包。

在详细研究AIM协议的具体内容时，有必要讨论一下开发二进制协议所用的一些基本的构造块。AIM是一个专有协议[㊀]的例子。尽管没有官方的文档，但是有关该协议的结构的大量细节我们都可以获得，这归功于其他人对该协议所做的逆向工程的努力。AIM的官方名称也叫作OSCAR（for Open System for Communication in Realtime）。逆向工程考虑到可替换的客户端，例如GAIM[㊁]和Trillian[㊂]。此外，一些网络协议分析器如Wireshark能够对包结构完全解码。这是很重要的一点。如果试图构件一个以专有协议为目标的模糊器，绝不要试图对协议实施逆向工程来从头构建模糊器。尽管这种做法在一些情况下是必要的，但是也有一些情况下你对协议或文件格式的细节感兴趣，其他人也是如此。应该利用你的好朋友Google来搜索资料，了解别人已经为你做了什么。对于文件格式模糊器，可以尝试在站点Wotsit.org获取信息，这是一个非常优秀的、集官方和非官方为一体的有关专有和开放协议文档的网站。

下面让我们看一看AIM的认证或登录过程。理解协议的第一步是捕捉一些原始数据然后将这些数据分解为更有意义的结构。对于AIM来说，我们是幸运的，因为Wireshark就已经能够理解协议的结构。现在让我们略过开始的一些有关握手的数据包，转到用户登陆有关的数据点。参见图4-1，其中显示了Wireshark输出的初始数据包，其中显示出AIM客户的用户名被发送到AIM服务器。可以看到，协议有两个单独的层次构成。高层包括AOL即时消息的头，它标识出被发送的数据的类型。在这种情况下，该消息头说明这个数据包包含来自AIM Signon家族（0x0017）和Signon子家族（0x0006）的数据。

在低层，我们可以看到位于AIM Signon之下的数据和登录的消息头。在此处，只有infotype（0x0001）、buddyname（fuzzing is cool）和buddyname的长度字节数（13）被发送。这是一个使用可变长域的例子。注意buddyname的长度后面紧跟着它本身。前面提到过，二进制协议中普遍使用数据块。数据块以一个字节数开头，说明该数据块的数据长度，后面是实际数据（buddyname）。有时字节数的值也包括这个值本身所占的字节数，其他一些时候，这个值只表示后面所跟的实际数据所占的字节数。对于模糊测试而言，数据块是一个重要的概念。如果创建一个向数据块注入数据的模糊器，那么就必须要仔细调整数据块的长度，否则接收方应用程序将无法理解数据包中其余的内容。诸如SIPKE[㊃]这样的网络协议模糊器在设计时要注意这一点。

㊀ http://en.wikipedia.org/wiki/AOL_Instant_Messenger

㊁ http://gaim.sourceforge.net/

㊂ http://www.ceruleanstudios.com/

㊃ http://www.immunitysec.com/resources-freesoftware.shtml

```
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 Signon: 发送的用户名信息

图4-2显示了响应初始请求的数据包，服务器提供了一个质疑值（challenge value, 3740020309）。AIM客户使用这个质疑值生成一个密码的杂凑值。这里要再一次注意的是，此处处理的是一个数据块，质疑值的前面还有该值所占的字节数。

```
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 Signon: 服务器的响应

一旦质疑值被接收，如图4-3所示，客户就会再一次响应，返回screen name，但是此时还包括密码的杂凑值。包括在登录密钥中的还有执行登录的客户一些细节信息，用来帮助服务器识别哪个客户有权力登录。这些值以名字[nd]值对的形式提交，这是二进制协议中又一种常见的数据结构。在这里例中使用的是client id string (name) 的形式，它与AOL即时消息同时被提交，版本号5.5.3591/WIN32。同样，前面还包括一个说明值域长度的长度值。

```
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 Signon：发送的登录密钥信息

4.5 网络协议

前面介绍的FTP和AIM协议都是网络协议的例子。Internet上到处都是网络协议，并且哪里都不缺少这种协议。网络协议包括数据传输、路由、电子邮件、流媒体、即时消息以及其他更多形式的通信协议等，种类之多超乎你的想象。正如一位智者所言“关于标准，最好的事情就是有太多的标准可供选择”。网络协议标准也不例外。

网络协议是如何被开发的呢？这个问题的答案很大程度上依赖于协议是专有的还是开放

的。专有协议是被封闭的小组开发的，这个小组属于某一个公司，协议主要用于专门的产品，其维护也是由这个公司来控制。不管怎么说，专有协议有它的固有优点，因为开发者只需要在一个小的群体范围内就标准达成一致就可以了。另一方面，Internet协议在本质上是开放的，因此需要在大量不同的群体之间达成一致。总体上讲，Internet协议是由Internet Engineering Task Force（IETF）㊀开发和维护的。IETF对Internet标准的提案有一个冗长的发布和获取反馈的过程，首先要发布“Request for Comment（RFC）”，它们是描述协议和供同行进行评审的公开文档。随后是发布评论意见和修改意见，之后RFC才能被IEFT采纳为Internet标准。

4.6 文件格式

如同网络协议，文件格式描述的数据结构也可能是开放的或专有的标准。为了说明文件格式，让我们考虑Microsoft Office文档。Microsfot历史上在它的文档处理软件中使用专有文件格式，包括Microsoft Office、Excel和PowerPoint。OpenOffice.org最初创建了一种开放的、富有竞争力的文件格式，称为OpenDocument格式（ODF）㊁，后来被高级结构信息标准化委员会（OASIS）㊂采纳。OASIS是一个由主要的IT厂商提供支持的行业组织，它的工作宗旨是开发和采纳e-business标准。

为了回应OASIS的动议，Microsoft首先在2005年从专有的Microsoft Office协议格式转向开放标准，㊃当年Microsoft宣布未来的Office版本将使用Microsoft Office Open XML格式。这种文件格式是开放的，但是却构成了对ODF的竞争。Microsoft后来宣布了开放XML翻译器项目㊄，该项目承诺将开发出能够在几种相互竞争的XML文档之间相互翻译的支持工具。

让我们比较一个开放简单文本文件格式和一个专有二进制文件格式，考察OpenOffice Writer和Microsoft Word创建的相同内容的文档。对于两种格式的文档处理工具，我们都首先创建一个空文档，然后在其中增加文本“*fuzding*”，选择默认字体。OpenOffice Writer默认将文存储为开放文档文本文件（*.odt）。用一个十六进制编辑器打开这个文件，其中内容看上去似乎是一个二进制文件。然而，如果我们将该文件的扩展名改为*.zip然后使用一个合适的解压缩软件打开它，就可以看到其中包含了多个XML文件、图像、目录以及其他文本文件，其内容如下：

```
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
```

㊀ http://www.ietf.org/

㊁ http://en.wikipedia.org/wiki/OpenDocument

㊂ http://www.oasis-open.org

㊃ http://www.microsoft.com/office/preview/itpro/fileoverview.mspx

㊄ http://sev.prnewswire.com/computer-electronics/20060705/SFTH05506072006-1.html

```
07/18/2006  04:05 AM             1,149 meta.xml
07/18/2006  04:05 AM                39 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
```

这些文件分别描述了文档的不同部分的内容。由于它们都是XML文件，因此它们都是可读的，用任何基本的文本编辑器都可以编辑它们。看一下其中的Content.xml文件，其内容如下，其中包含了一系列描述名字空间和格式细节的元素，我们可以清楚地看到其中有一个黑体字符串，它就是文档的原始内容（*fuzzing*）。

```
<?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.w3.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:sequence-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>
```

再用Microsoft Word 2003创建相同内容的文档，最终会得到一个二进制的Word文档（*.doc）文件。尽管它是一个二进制文件，但是其实际存储空间比前面创建的开放文档文件要大（20KB对7KB）。用十六进制文本编辑器打开它，你将会注意到其中包括了各种各样的可读字符串，夹杂着看不懂的二进制代码。下面选择的一部分快照是反映文档原始内容的部分，作者名列于文档属性区中，此处还包括其他用来标识创建原始内容的字符串。

```
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　常见的协议元素

为什么要介绍这一部分背景知识呢？根据文件格式或网络协议的结构不同，需要相应地调整对其进行模糊测试的方法。对数据结构了解的越多，就越容易关注那些容易引发异常条件的模糊测试对象。下面就让我们简单学习协议中的常见元素。

4.7.1 名字-值对

不论是二进制协议还是普通文本协议，其中的数据通常以名字-值对（例如size=12）的形式表示，，不过这种情况对普通文本尤其适用。回过头看一看前面提到的Content.xml文件，在整个XML文件中都可以看到名字-值对。作为一个一般规则，在处理名字-值对的时候，至多可能通过对其中的值部分进行模糊测试而识别出潜在的漏洞。

4.7.2 块标识符

块标识符是标识二进制数据类型的数字值。它的后面可能跟着可变的或固定长度的数据。在前面讨论过的AIM的例子中，AIM Sinon的头域中包含infotype（0x001）块标记。它的值定义了其后所跟的数据类型，在该例中是buddyname。模糊测试可被用于识别一些未被记录为文档的块标识符，这些标识符可能接受额外的数据类型，它们都可以被模糊测试。

4.7.3 块长度

块的长度在前面曾经提到过，它通常包含诸如名字-值对这样的数据，名字-值对的前面通常是一到多个字节用来说明该区域的数据类型以及后面所跟的变长域的数据长度。在模糊测试的时候，可以试着修改数据块的长度值，以使它大于或小于其后所跟的数据的实际长度，同时监视测试结果。这种测试策略是缓冲区溢出或不满的一个常见的来源。作为可选的另一种方案，当对块内的数据进行模糊测试时，应确保这个长度值得到了相应的调整，以确保应用程序能够恰当地识别数据。

4.7.4 校验和

一些文件格式在整个文件中嵌入了校验和，以帮助应用程序识别可能由于各种原因而遭到破坏的数据。校验和并不一定要被实现为一种安全保护机制，因为文件可能会因为各种不同的原因而被破坏，但是校验和却可以影响模糊测试的结果，因为应用程序通常在遇到不正确的校验和时会放弃文件处理过程。PNG图像格式所采用的文件类型是利用校验和的一个典型例子。在碰到校验和时，至关重要的是让模糊器考虑到校验和并重新计算和重新写入校验和，以确保目标应用程序能够恰当地处理文件。

4.8 小结

尽管对文件和网络通信可以采用强制性的模糊测试，然而更高效的做法是只对目标应用程序中有可能导致安全隐患的部分实施测试。虽然获得对目标应用程序的理解需要付出一定的前期努力，但是这些努力一般是值得的，尤其是在开放和专有协议的文档充足的情况下。经验有助于识别协议中最佳的目标位置，本章介绍的内容突出说明了历史上发现的应用程序中可能存在漏洞的某些弱点区域。在第22章“自动化协议解析”中，将详细介绍一些可用于解析协议结构的自动化技术。

第5章 有效模糊测试的需求

"你教一个孩子阅读，他将能够通过一个文化测试"

——George W. Bush, Townsend, TN, 2001年2月21日

在前一章中，我们介绍了各种不同的模糊器类型和不同的模糊测试方法。本章要讨论的是进行有效和高效的模糊测试所需要的技巧和技术。一些明显的因素，例如测试计划和模糊器的可重用性应该在开发模糊器之前就予以考虑。这样的考虑有助于保证未来的工作可以依据和构建在当前工作的基础之上。使模糊器更复杂的一些更多的特征，例如过程状态和深度、跟踪和度量、错误检测以及约束条件等，本章也进行了分析讨论。在本书的第二部分，我们还将介绍几种模糊器的测试目标以及如何开发自动化工具来支持这些目标。在阅读本章的过程中，应该尽量记住本章介绍的一些概念，因为这些概念在构建新模糊器甚至在构建本书第12章要介绍的模糊测试框架的过程中将起到重要的作用。尽管当前有许多在售的商用模糊器产品，但是它们中间没有一个能够完全满足本章所描述的有效模糊测试的所有需求。

5.1 可重现性和文档记录

对模糊测试工具的一个明显需求是它应该具备重现测试结果的能力，测试结果既包括单个的测试也包括测试序列的测试结果。这个需求对测试者与他人或小组交流测试结果至关重要。作为一个模糊测试者，应该能够为模糊测试工具提供一个引起问题的测试例数，并且心里要明白观察到的目标行为在不同的测试运行之间应该保持严格的一致。让我们考虑一个假设的场景：假设你正在测试Web服务器的能力，检验它是否能够处理非法的POST数据，在对其进行模糊测试时，向其发送的第50个测试用例引起服务崩溃，进而发现了一个可能被人利用的内存破坏条件。重新启动Web服务器并重传引起问题的那个数据给目标应用程序之后却什么也没有发现。难道这是假象吗？当然不是：计算机是确定性的计算工具，它本身没有随机的概念。前面的崩溃问题是由于输入的组合而产生的。或许前面输入的数据包首先让服务器进入了某个状态，最后第50个测试用例触发了内存破坏条件。没有进一步的分析，不具备系统的方法来重新回放整个测试过程，我们就无法进一步界定问题的起因。

即使不是强制规定要编写文档，各种各样的测试结果的记录也很有用途，这是信息共享时代的一个基本需求。在国际外包开发[⊖]的大趋势下，安全性测试者通常不可能直接走到与软件问题有关的开发者身边。外包如此普及，以至于计算机科学的学生都知道如何去利用它[⊜]。各

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

⊜ "Computer Science Students Outsource Homework," http://developers.slashdot.org/developers/06/01/19/0026203.shtml

种各样的通信障碍，包括时区、语言和通信媒体等，要求我们尽可能以清晰简洁的形式对信息打包，这一点非常重要。有组织的开发和记录文档所带来的工作负担不应该是完全的手工劳作。一个好的模糊测试工具应该产生和存储那些容易被分析和引用的日志信息。

我们应该考虑到前面讨论的模糊器如何满足软件问题的可重现性，如何自动记录日志和文档。还要考虑到如何根据这些目标改进模糊器的实现。

5.2 可重用性

从较大规模开发的角度考虑，如果我们正在构建一个文件格式模糊工具，那么我们不希望每次测试一种新的文件格式时必须要重新开发整个工具。我们可以构建出一些可重用的特征，以便未来在测试另一种不同的文件格式时能够节省时间。举一个例子，假设我们的目标是开发一个JPEG文件格式模糊测试工具，用来查找Microsoft Paint中的bug。事先我们考虑到并且知道需要重用现在的某些工作，那么很可能会决定将该工具分为三个构件，参见图5-1所示。

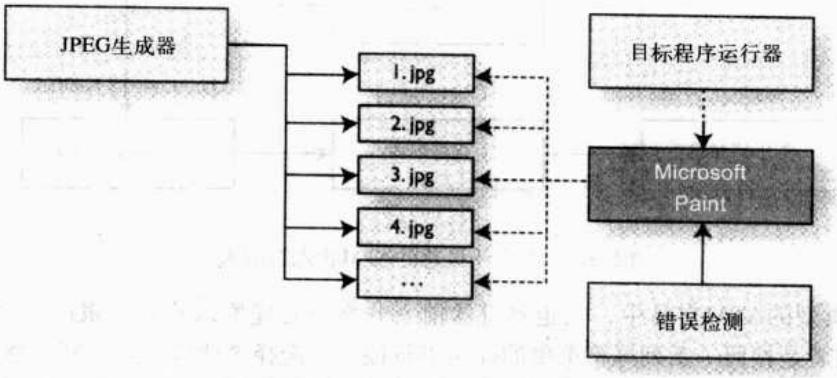


图5-1 一个虚构的文件格式模糊器的结构分解和总体结构概览

JPEG文件生成器负责不停地生成变异了的JPEG文件。目标程序运行器是工具的前端，它负责周而复始地产生图像，每次都使用不同的参数让Microsoft Paint加载新生成的图像。最后，系统结构中还包括一个错误检测引擎，它负责监视Microsoft Paint的异常条件的每个实例。将系统结构分为三个构件允许我们改变测试集以适应其他文件格式，因为只需要改动生成器即可。

如果在较小规模上考虑，我们开发的许多构造块应该可以在不同的模糊测试项目之间重用。例如，考虑一下E-mail地址。这种基本的字符串格式很常见，包括在简单邮件传输协议（SMTP）事务，登录屏幕和VoIP初始会话会话协议（SIP）中：

```
Excerpt of an SIP INVITE Transaction

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
```

在上述每种情况下，E-mail地址都是一个有用的域，因为我们确定该域将被解析并且有可能被分隔为各种各样的构件（例如用户和网域）。如果我们打算花时间列举Email地址可能引起问题的表示形式，那么如果让这些表示在所有的模糊器中被重用岂不是很好吗？

我们应该考虑如抽象或使模糊器模块化，根据前面的分析思路让使模糊器具有可重用性。

5.3 过程状态和过程深度

为了牢固掌握过程状态和过程深度的概念，下面举一个许多人都很熟悉的例子：ATM银行业务。考虑下面的一个简单的状态图，参见图5-2。

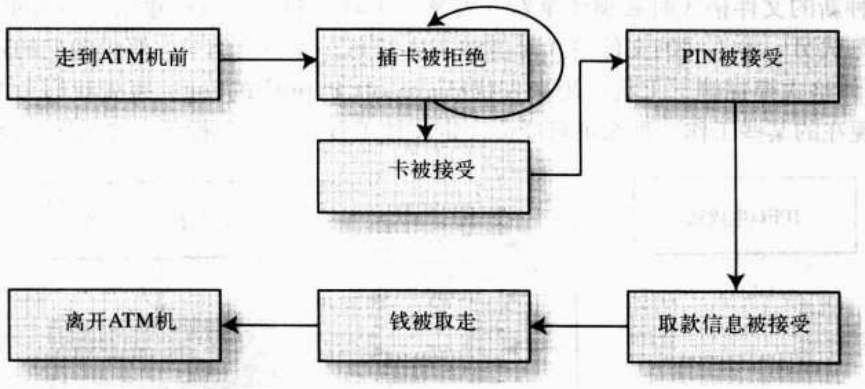


图5-2　一个人为设想的ATM状态图的例子

在一次典型的ATM交易中，人走到机器前（非常小心谨慎以防有人跟踪），插入他的卡，输入PIN，接着是按照一系列屏幕菜单的提示进行操作，选择希望提取的金额，然后是取出钱，最后是结束交易。同样的状态概念和状态转移的概念也适用于软件。稍后我们还要举一个这方面的具体例子。我们将*过程状态*（process state）定义为目标过程在任意给定的时间所处的具体状态。动作可以使状态之间发生转移，例如插卡或选择提取金额的动作。过程进行的深入程度叫做*过程深度*（process depth）。例如，描述一个取款金额发生在比输入一个PIN更深的过程深度上。

再举一个和安全更相关的例子，考虑一个secure shell（SSH）服务器。在连接到服务器之前，服务器处于*初始*状态。在认证过程中，服务器处于*认证*状态。一旦服务器成功通过了对一个用户的认证，那么服务器就处在*已认证*状态。

过程深度是达到一个具体的状态而所需要的“向前”步骤的具体度量。仍以SSH服务器为例，图5-3是它的一个状态图。

在过程中，已认证状态比认证状态“更深”，因为已认证状态需要在认证状态的基础上执行更多的子步骤。在模糊器的设计中，过程状态和过程深度是能够引起重要复杂情况的重要概念。下面是一个引起复杂情况的例子。为了对一个SMTP服务器的E-mail地址的`MAIL FROM`谓词参数进行模糊测试，就不得不连接到服务器并发出一个`HELLO`或`EHLO`命令。如图5-4所示，底层

的SMTP实现可能采用同一个函数来处理MAIL FROM命令，不论最开始使用的命令是什么。

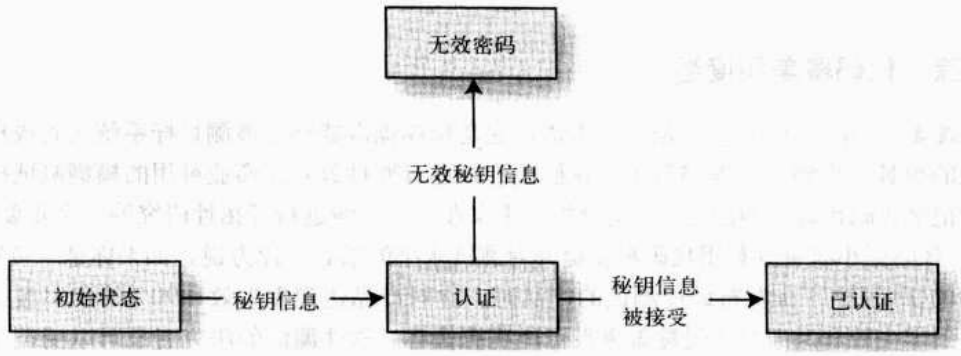


图5-3 SSH服务器

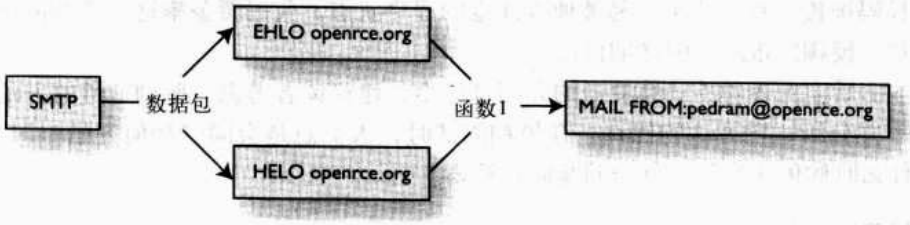


图5-4 SMTP状态图样例1

在图5-4中，函数1是唯一定义的用来处理MAIL FROM数据的函数。还可以采用另一种可选设计方案，参见图5-5，其中的SMTP实现包含了两个单独的例程用来处理MAIL FROM数据，这两个例程的选择取决于初始的命令。

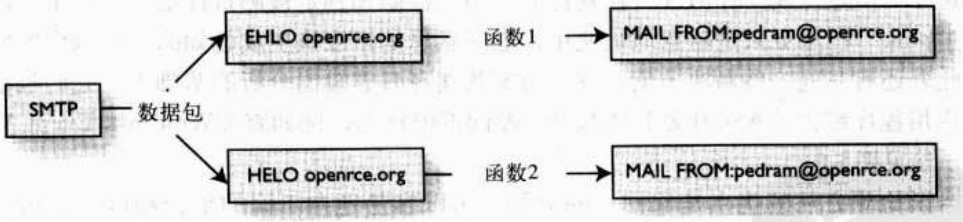


图5-5 SMTP状态图样例2

这实际上是一个真实的案例。2006年9月7日，一个有关与Ipswitch Collaboration Suite捆绑的SMTP的远程攻击栈溢出漏洞的安全公告被发布。该漏洞的产生是因为在解析E-mail地址时遇到了其中包含位于字符@和：之间的长字符串。该分析例程只有在客户发起连接时输入了以EHLO开头的会话信息时才可达。在设计模糊器时，应该留心类似的、可能的逻辑拆分。为了获得充分的覆盖，我们在设计模糊器时对E-mail地址进行了两次完全变异，第一是通过对EHLO变异，第二次是对HELO变异。如果沿这种思路获得更多的逻辑拆分并分别对其变异的话会产生什么结果呢？不要忘记完全充分的覆盖所需要的逻辑循环拆分是呈指数规模递增的。

在学习后面要介绍的各种各样的模糊器时，应该考虑到如何处理过程深度的变化和逻辑拆分。

5.4 跟踪、代码覆盖和度量

代码覆盖（code coverage）是一个术语，它是指模糊器能够让被测目标系统达到或执行的过程状态的数量。直到写作本书为止，我们还不知道有哪种公开或商业可用的模糊测试技术具有跟踪和记录代码覆盖率的能力。这是分析员未来在这一领域进行开拓性研究的一个重要概念。质量保证（QA）小组可以利用代码覆盖提升对测试水平的信心。比方说，如果你是一位Web服务器产品的QA领导，在发布经过测试的产品时，你对产品达到零失效和90%的代码覆盖率比对产品达到零失效和25%的代码覆盖率很可能更有信心。软件漏洞的研究者也可以受益于代码覆盖率分析，通过对代码覆盖率的分析他们可以识别出一些应该对软件进行的必要修改之处，以使软件的代码被覆盖到一些不容易被他人注意的程序状态。代码覆盖率这个重要的概念将在本书的第23章“模糊器跟踪”中详细讨论。

在阅读本书后面对各种不同模糊器的介绍过程中，建议读者考虑一些创造性的方法来确定代码的覆盖率并分析这种方法的优点。在模糊测试时，人们总是会问“如何开始？”，注意另一个问题“什么时候停止？”与这个问题同样重要。

5.5 错误检测

生成和传输可能引起异常的数据只是模糊测试战役的一半。战役的另一半是精确地确定错误何时出现。在写作本书的时候，大部分可用的模糊器对这个问题还处于“盲目”状态，因为这些模糊器并不知道被测目标如何对输入数据产生反应。一些商业解决方案在两次发送恶意请求之间加进了“ping”或“存活性”检查机制，用来控制和判定被测目标是否仍然正常履行功能。术语“ping”在这里只是随意地被使用，它的意思是指应该生成已知的良好响应的任何一个事务。此外还有其他一些解决方案，这些方案构建在日志输出分析的基础上。它们包括监视由单独的应用程序维护的ASCII文本日志或者查询系统日志，例如查询Windows Event Viewer中的日志，参见图5-6。

这些错误检测方法的优点是它们中的大部分可以容易地在不同的平台和体系架构之间移植。然而，这些方法在能够检测到的错误种类方面严重受限。例如，它们中没有一种方法能够检测到发生在Window应用程序内同时被结构化异常处理例程（SHE）所处理的错误。

错误检测的下一代技术是使用轻量级的调试客户端来检测目标系统中的异常条件何时发生。例如，本书第二部分将要介绍的FileFuzz工具包括了一个用户定制的、开放源代码的Microsoft Windows调试客户端。利用这种类型的工具的负面作用是必须为测试的每一个目标平台开发一个调试支持工具。例如，如果想要在Mac OX、Microsoft Windows和Gentoo Linux上测试三个SMTP服务器，就很可能需要开发两个甚至三个不同的监视客户端。此外，由于测试目标所限，有可能无法或无法及时构建这样的调试客户端。再例如，如果测试一个硬件VoIP电

话，那么就可能必须重新走到对测试实施控制或对日志实施监控的老路上去，因为硬件解决方案不支持调试，因此可能需要另外的专用工具。

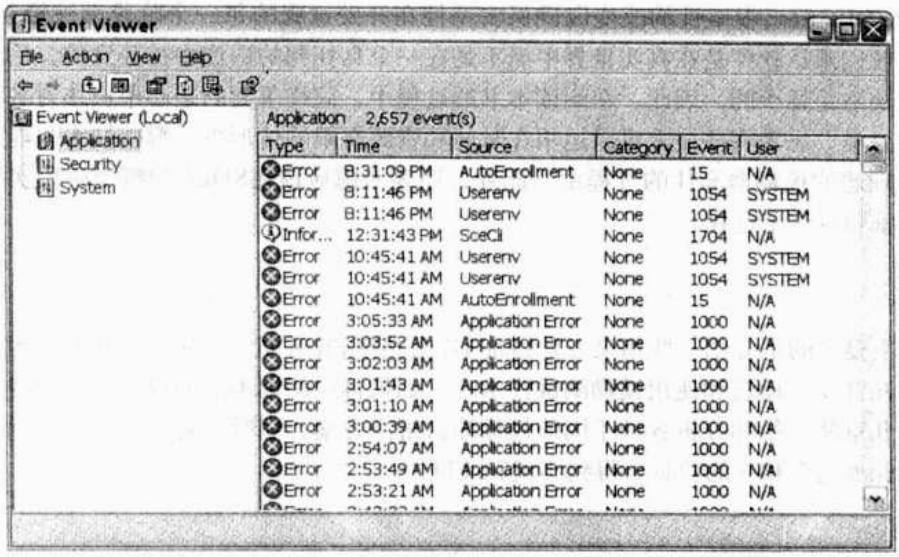


图5-6 来自Microsoft Windows Event Viewer的错误日志样例

如果进一步向前展望，错误检测的万能药很可能是诸如Valgrind[⊖]和Dynamo Rio[⊖]这样的动态二进制插装／翻译平台。在这样的平台上，有可能在软件开发时就进行错误检测而不是在错误发生后再检测它。如果朝着“5000英尺”的距离展望，基于DBI的调试引擎能够非常专业地在底层对目标软件执行分析和插装。这种底层的控制能够支持内存泄露检查、缓冲区溢出检查和非越界检查，等等。回顾前面讨论可重现性时所举的内存破坏的例子，在该例子的情形中，轻量级的调试客户端能够告诉我们内存破坏是何时被触发的。回顾那个例子我们可以知道，当时的场景是将许多数据包发送给目标服务，结果第50个测试用例引起服务崩溃。在一个诸如Valgrind这样的平台上，我们有可能检测初始的内存破坏，它发生在触发异常之前的早期的测试用例的执行中。这种方法能够节省数个小时甚至许多天的模糊过程和错误跟踪。

在随后的几章中，我们将讨论各种不同的模糊器，届时读者可以回顾这里提及的不同的目标监视方法，考虑其中的哪种方法最适用。

5.6 资源约束

各种各样的非技术因素都可能对模糊测试带来限制，例如预算和时间期限。这些因素在设计和规划阶段必须牢记。例如，你可能在模糊测试之前的最后一分钟产生试水前的恐慌感，因为没有人哪怕是很简单地检查过你投资50万美元所开发的产品的安全性。安全性在太多情况下

⊖ Valgrind:http://valgrind.org/

⊖ Dynamo RIO: http://www.cag.lcs.mit.edu/dynamorio

成为软件开发周期（SDLC）过程事后的追悔，在增加新特征或达到了产品发布期限时的追悔。如果我们希望开发安全的软件，那么安全性应该被我们“培养”而不是被我们“突击”。这种培养包括对SDLC做出根本性的改变以确保安全性在开发过程的每一个阶段都被恰当地考虑。换句话说，我们承认软件是在真实世界中而不是在一个乌托邦的国度中被开发的，后者的资源是充足的，缺陷是稀少的。因此，在阅读本书的过程中，同样重要的是心中应该对各种不同的技术做一个分类，思考哪些技术可以运用在时间和资源有限的情形下，哪些技术只能应用在梦想“最终”理想的模糊器套件的过程中。此外，还要考虑应该在SDLC的哪个阶段实现这样的工具，以及谁负责这个过程。

5.7　小结

本章的主要目的是介绍一些用来开发特征丰富的模糊器所要考虑的一些基本思想和创造性见解。当你在开发、比较和使用模糊测试技术时，如果再次阅读本章的内容，就会变得小心谨慎。未来的几章将介绍和分析各种不同的模糊测试解决方案，届时读者应该考虑本章所介绍的一些需求目标如何在相应的背景下得到实现和改进。

第二部分　目标和自动化

第6章　自动化测试和测试数据生成

“我们的敌人富有创新精神并且足智多谋，我们也一样。他们总想找到新的方法来损害我们的国家和人民，而我们不会。”

——George W. Bush, Washington, DC, 2004年8月5日

模糊测试在很大程度上可以采用自动化的方式来实施。模糊测试相对于其他的软件测试方法而言，其核心价值体现在能够将大量的手工测试转换为高度的自动化测试。生成单个测试用例是一项费力和枯燥的工作，而某些任务则非常适合于计算机来完成。模糊器的核心竞争力就是它在人工干预最小的情况下，生成有用测试数据的能力。本章关注于自动化测试的不同方面，包括语言的选择、有用的构造块、以及为了有效的测试目标软件，在测试数据生成过程中如何选择适当模糊值的重要技巧。

6.1　自动化测试的价值

尽管自动化测试的优点是显而易见的，这里为了清晰起见，再从两个方面回顾一下。首先，我们根据人工计算的能力来强调其价值，然后再分析其可重用性。针对于早期的计算机而言，计算时间是非常宝贵的，事实上，它比人工计算的时间还要宝贵。因此，在这些系统上编程是一个非常枯燥乏味的手工过程。程序员要操作磁带、转换开关、并且要手工输入十进制或者二进制的机器操作码。随着科技的发展，程序员的聪明才智不断得以体现，计算时间更加充裕，使得成本代价的平衡发生了改变。目前，这种趋势仍在继续，并且在日益流行的高级编程语言如Java，.NET和Python中得到了印证。这些编程语言牺牲了一些计算时间，而却为程序员提供了更加简便的开发环境，以及更加迅速的开发转换时间。

鉴于此，尽管对于程序分析员而言，他和基于socket的daemon程序进行交互并且手工在程序漏洞中输入数据以发现软件缺陷是十分可行的，但是最好将人工时间花费在其他任务上。当比较模糊测试与人工审核工作如源代码审查和二进制审核时，也可以得出同样的结论。人工审核方法需要高级分析员花费大量的时间，而前一种方法即模糊测试，则可以或多或少的由任何人来实施。最后，自动化应当作为减少高级程序分析员工作量的第一步，使其能够像关注其他测试方法那样来发现缺陷。

接下来，强调一下可重用的必要性。下面两个关键因素奠定了可重用性的重要性：

- 如果我们能够为一个FTP服务创建一个可重用的测试过程，那么我们就可以使用同一个测试过程很方便的测试其他版本的程序，即便是完全不同的FTP服务也可以。否则，需要浪费大量的时间来为每一个FTP服务重新设计并实现一个新的模糊器。
- 如果一个非寻常的事件序列在目标程序中触发了一个缺陷，那么我们必须要再次产生整个序列以限制造成异常的特定结果。程序分析员需要创建不同的测试用例，因为它们不具备科学的可重用性。

简而言之，测试数据生成和再生成、缺陷监测等耗时耗力的工作最适合于实现自动化。像大多数计算任务一样，对模糊测试而言，幸运的是我们可以利用许多已有的工具和库来实现自动化。

6.2 有用的工具和库

大多数模糊器的开发者将会发现他们从头开始创建工具，尽管已经有了大量的公共可用的模糊器脚本[㊀]。幸运的是，许多工具和库可以在模糊器的设计和实现阶段为你提供帮助。本节列举了一些这样的工具和库（按字母顺序排列）。

6.2.1 ETHEREAL[㊁]/WIRESHARK[㊂]

Wireshark（Ethereal项目的一个分支）[㊃]是一个流行的开源网络嗅探器和协议分析器。尽管依赖于其中的一个库并不是必要的，但毫无疑问，该工具可以在创建模糊器的研究和调试阶段提供帮助。Wireshark所提供的许多开源的分析器也经常作为参考来发挥作用。被识别为一个拥有可用分析器的被捕获通信会被显示为一系列的字段/值对，这些字段/值对是和一组原始字节相对应的。在开始人工协议分析之前，通常是首先认真地查看一下Wireshark的分析结果。对于一个可用分析器的快速列表而言，可以参阅Wireshark的版本控制库，特别是epan/dissectors[㊄]目录。

6.2.2 LIBDASM[㊅]和LIBDISASM[㊆]

Libdasm和libdisasm都是免费和开源的反汇编程序库，你可以将它们嵌入到你的工具中，以用于从二进制流开发AT&T和Intel的语法解析器。Libdasm是用C语言编写的，而libdisasm则是用Perl编写的。在libdasm中包含一个Python的接口。尽管反汇编程序对于创建网络通信是不必要的，但是当在通信的终端进行自动化的错误检测时它是很重要的。上述两个库在本书中被广泛使用，尤其是在第12章“文件格式模糊测试：UNIX平台上的自动化测试”，第19章“内存

㊀ http://www.threatmind.net/secwiki/FuzzingTools
㊁ http://www.ethereal.com
㊂ http://www.wireshark.org
㊃ http://www.wireshark.org/faq.html#q1.2
㊄ http://anonsvn.wireshark.org/wireshark/trunk/epan/dissectors/
㊅ http://www.nologin.org/main.pl?action=codeView&codeId=49
㊆ http://bastard.sourceforge.net/libdisasm.html

数据的模糊测试”，第20章“内存数据的模糊自动化测试”，第23章“模糊器跟踪”以及第24章“智能的故障检测”中。

6.2.3 LIBNET[㊀]/LIBNETNT[㊀]

Libnet是一个针对创建和注入低层网络包数据的免费、开源的高层API。该库隐藏了在生成IP和链接层通信时包含的许多复杂的细节，并且始终提供了跨不同平台的可移植性。如果你正在编写一个网络栈的模糊器，那么可能会对该库产生兴趣。

6.2.4 LIBPCAP[㊁]

LibPCAP以及和微软Windows相兼容的WinPCAP[㊂]都是免费、开源的高层库，能够简化跨UNIX和微软Windows平台的网络捕获和分析工具的创建。许多网络协议分析工具，例如前面提到的Wireshark都是依靠该库创建的。

6.2.5 METRO PACKET LIBRARY[㊄]

Metro Packet Library是一个C#库，它提供了一个与IPv4、TCP、UDP和Internet控制消息协议（ICMP）相交互的抽象接口。该库有助于创建包嗅探器和网络分析工具。对于该库的讨论和应用请见第16章“网络协议模糊测试：Windows平台上的自动化测试”。

6.2.6 PTRACE

在UNIX平台上进行调试，大部分是在系统调用ptrace()（进程跟踪）的帮助下完成的。一个进程可以利用ptrace()来控制寄存器状态、内存、执行过程，并且捕获另一个进程产生的信号。这种机制用于实现第8章“环境变量和参数的模糊测试：自动化”和第12章“文件格式模糊测试：UNIX平台上的自动化测试”中所讨论的工具。

6.2.7 PYTHON EXTENSIONS

在创建模糊器时可以使用不同的Python扩展。示例扩展包括Pcapy、Scapy和PyDbg。Pcapy[㊅]是针对LibPCAP/WinPCAP的Python扩展接口，以使得Python脚本能够捕获网络通信。Scapy[㊆]是一个功能强大的、简便的包操作扩展，它既可以通过交互方式使用也可以作为一个库使用。Scapy能够创建和解析许多不同种类的协议。PyDbg[㊇]是一个纯Python的微软

㊀ http://www.packetfactory.net/libnet
㊁ http://www.securityfocus.com/tools/1559
㊂ http://www.tcpdump.org
㊃ http://www.tcpdump.org/wpcap.html
㊄ http://sourceforge.net/projects/dotmetro
㊅ http://oss.coresecurity.com/projects/pcapy.html
㊆ http://www.secdev.org/projects/scapy
㊇ http://openrce.org/downloads/details/208/PaiMei

Windows32位的调试器，是一个方便灵活的进程工具。PyDbg库是PaiMei[⊖]逆向工程框架的一个子集，该框架将在本书的第19、20、23和24章中使用。

这里所列出的某些库可以用于多种语言。而另外一些则要受到很大的限制。当决定采用何种语言来开发模糊器时，特定的需求以及能够帮助实现这些需求的可用的库是必须要考虑在内的因素。

6.3 编程语言的选择

许多人认为模糊器编程语言的选择问题是一个笃信性的争论，他们固执的坚持其选择而不论所要解决的问题是什么，而其他人则遵循“对于正确的任务使用正确的工具”的准则。我们倾向于后一种做法。在本书中，你将会发现源代码是采用不同的语言所编写的。在本书的编写过程中做了大量努力以尽可能多的包含可重用的实际代码示例。无论你个人的喜好如何，我们鼓励你在选择一种语言用于特定任务之前，先思考所给定的任何语言的优劣之处。

在最高的层次上，你将会在开发的起点上发现编程语言存在一个根本的不同：解释型语言和编译型语言。在低层上，编译型语言如C和C++提供了准确和直接访问底层构件的功能。例如前面提到的Libnet库，就是在此低层的接口。在高层上，解释型语言如Python和Ruby提供了更快的开发时间，并且不需要重复编译就可以进行修改。针对高层语言提供低层功能的库通常是存在的。如果模糊器既要求灵活性又要注重代码质量，那么模糊器将会用所有的语言来编写。依赖于特定的任务和对各种语言的熟悉程度，从Shell脚本到Java，从PHP到C#，肯定有一种语言最适合所面临的任务。

6.4 测试数据生成和模糊启发式

如何生成测试数据的实现方法只是待解决问题的一部分，同等重要的是决定生成什么样的测试数据。例如，考虑如果我们正在创建一个模糊器以分析一个IMAP服务的健壮性。在许多IMAP请求动作和构造器中，需要加以研究的是命令继续请求（Command Continuation Request,CCR），其引自RFC3501[⊖]的描述如下：

7.5 服务器响应——命令继续请求

命令继续请求响应是通过一个“+”标记来指示的，而不是一个标签。这种形式的响应意味着服务器已做好准备以接收来自客户端的命令继续请求。响应的剩余部分是一行文本。

该响应用在AUTHENTICATE命令中，将服务器数据传递给客户端，并请求额外的客户端数据。如果命令的参数是一个文本，那么也会用到该响应。

客户端不允许发送文本的字节，除非服务器表明它希望这样做。这就允许服务器来处理命令并且逐行的抛出错误。该命令的剩余部分，包括终止命令的CRLF，后面跟着文本的字节。

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

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

如果有任何额外的命令参数，则文本字节的后面跟着一个空格和如下参数：

```
Example:    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"
...
```

根据RFC，任何以{数字}（用黑体突出显示）形式结尾的命令指明了在后续行中接续的命令的剩余字节数。这是模糊测试的主要目标，但是应该使用什么值来测试该字段呢？所有可能的数字值都能够被测试吗？目标daemon程序可能会接收最大的32位整数值范围内的所有整数，其大小为0xFFFFFFFF（4 294 967 295）。如果模糊器能够每秒钟执行一个测试用例，那么将会花费大于136年的时间来完成测试！就算我们将模糊器加足马力运行，使其每秒钟可以运行100个测试用例，那么仍然需要大约500天的时间来完成测试。按照这个速率，当我们结束测试时，IMAP协议可能将早已过时。很明显，不能检查整个范围内的所有数据，必须要选择一个智能子集或者可能整数值的一个代表集。

在模糊字符串或者模糊数据列表中所包含的特定的潜在危险值，称作模糊启发式（fuzz heuristics）。下面学习几个将要包含在智能库中的数据类型的类别。

6.4.1 整型值

在新的改进的整型测试用例列表中，选择两个极限边界（0和0xFFFFFFFF）测试用例是很明显的。还能增加其他什么值呢？可能所提供的数字被用作内存分配程序的大小参数。通常额外空间被指定的大小所包含以容纳一个头、尾或者终止空字节，例如：

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

由于相同的原因，被指定的大小可能会被之前的值减去以进行分配。当目标程序知道它不准备将所有的指定数据拷贝到新分配的缓冲区时，就会发生这种情况。记住整型值的溢出（加法运算导致结果超出32位整数的最大范围）和下溢（减法运算导致结果小于0）可能会导致在下面的代码行中出现潜在的安全问题，因此应当谨慎的包含边界附近的测试用例，例如0xFFFFFFFF-1，0xFFFFFFFF-2，0xFFFFFFFF-3...，和1，2，3，4等。

类似的，对所指定的大小同样可以应用乘法运算。例如，考虑如果提供的数据被转换为Unicode的情形。这就要求指定的大小去乘以2。另外，还应当包括额外的两个字节以确保空终止符。例如：

```
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);
```

为了在此以及类似的情形下触发一个整数的溢出，我们还应当包含下列边界附近的测试用例：0xFFFFFFFF/2,0xFFFFFFFF/2-1,0xFFFFFFFF/2-2等。那么被3除或被4除所得边界附近的测试用例是什么情形呢？在这种情况下，对16位整数（0xFFFF）而言，同样包含前面所列出的边界附近的测试用例又如何呢？那么对8位整数（0xFF）又如何呢？下面将这些问题与其他一些巧妙的选择列在一起，如下所示：

- 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 + 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

在上面的列表中，MAX32表示32位整数（0xFFFFFFFF）的最大值，MAX16表示16位整数（0xFFFF）的最大值，MAX8表示8位整数（0xFF）的最大值，随机选择的范围16作为一个合理的数量。根据不同的时间和结果，可以增加该范围值。如果在目标协议中有数百个整型字段，那么额外的启发式整数将上百倍的增加测试用例的总个数。

采用有趣的方法可以获得这些所选择的“启发式”整数。启发式只不过是“技巧性猜测”的一种风趣的说法。更加高级的用户可以在反汇编程序的作用下研究二进制代码，并且通过研究内存分配和数据复制程序中的交叉引用来抽取潜在的感兴趣的整数值。这个过程也可以实现自动化，第22章“自动化协议解析”中阐述了相关的概念。

智能数据集的必要性

2005年9月1日，一则名为Novell NetMail IMAPD Command Continuation Request Heap Overflow“Novell NetMail IMAPD 命令继续请求堆溢出”⊖的安全性公告同相应的提供商的补丁一起公开发布。这个公告详细阐述了该缺陷，它允许远程未授权的攻击者以执行任意代码，进而全面危及整个系统的安全。

所描述的缺陷存在于CCR的处理过程中，作为用户所指定的大小值被直接用做名为MMalloc()的自定义内存分配函数中的一个参数，如下面所示的汇编程序片段：

```
; ebx is attacker controlled
00402CA2 lea ecx, [ebx+1]
```

⊖ http://pedram.redhive.com/advisories/novell_netmail_imapd/

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

MMalloc()函数在分配内存之前对所提供的值执行一步小的数学操作。攻击者可以指定一个引发整数溢出的恶意数字，从而导致一个小的内存块被分配。所提供的原始的较大的值将在后面的memcpy()中使用：

```
; destination is attacker allocated
00402D6E rep movsd
00402D70 mov ecx, edx
00402D72 and ecx, 3
00402D75 rep movsb
```

该指令序列将把攻击者所提供的数据拷贝到分配堆区域的边界之外，并且随意的覆盖堆区域，最终破坏整个系统。

尽管Novell针对此问题的补丁成功地解决了这个特定的问题，但是它并没有解决所有可能的攻击路径。对于Novell而言，不幸的是，IMAP程序会把人工可读的数字形式转换为等价的整型值。例如，“−1”转换为0xFFFFFFFF，“−2”转换为0xFFFFFFFE等。因此，下面的恶意请求将被正确地处理：

```
x LOGIN {4294967295}
```

而下面的代码将不会正确处理：

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

上述问题后来被一个独立的研究者所发现，Novell于2006年12月22日发布了针对该问题的补丁[⊖]。

6.4.2 字符串重复

我们已经讨论了如何在模糊测试数据集中包含一些巧妙的整数，但是对于字符串而言，情形又如何呢？下面从经典[⊜]的“长字符串”开始讨论：

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

当我们对字符串进行模糊测试时，希望包含一些额外的字符序列。应当首先包含其他的ASCII字符，例如“B”。这是非常重要的，例如由于不同的行为，堆结构中“A”的ASCII值被“B”的ASCII值覆盖，这将在微软Windows操作系统中触发堆溢出。另外一个重要的原因是，实际上软件要特别的搜索并阻止A的长字符串。显然，某些提供商已经明白了使用长字符串AAAAAAAAAAAAAAAA的强大功能。

6.4.3 字段分隔符

同样需要包含非字母字符，包括空格和制表符。这些字符通常被用做字段分隔符和终止符。

⊖ http://www.zerodayinitiative.com/advisories/ZDI-06-053.html

⊜ 如果你不相信，可以通过Google搜索：http://www.google.com/search?hl=en&q=%22perl+-e+%27print+%22A%22*%22

将它们随机的包含到所生成的模糊测试字符串中，可以提高将被测协议进行部分复制的机会，进而增加被模糊测试的代码的数量。例如考虑相关的简单HTTP协议，下面列出了一些非字母字符：

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

其中有多少个可以被识别为HTTP字段分隔符呢？作为参考，请看下面的典型HTTP服务器的响应代码：

```
HTTP/1.1 200 OK
Date: Sun, 01 Oct 2006 22:46:57 GMT
Server: Apache
X-Powered-By: PHP/5.1.4-pl0-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
```

你可能会注意到的第一个样式就是这里有许多换行分隔符，通过序列0x0d 0x0a在字节层次上被表达。每个单独的行后面都使用不同的分隔符。例如在第一行，可以看到有字符空格（），前斜线（/）和点（.）被使用以分隔响应代码。在后续行上都使用了一个共同的分隔符冒号（:）来分隔不同的项，例如Content-Type,Server和Date和它们相对应的值。进一步研究可以发现逗号（,）、等号（=）、分号（；）和短划线（-）也被用来分隔字段。

当生成模糊字符串时，很重要的一点就是要包括像前面所列出的不同字段分隔符所分隔的很长的字符串。另外，增加分隔符的长度同样也是很重要的。例如，考虑在2003年发现的发送邮件头的处理漏洞[⊖]。此漏洞需要一个包含<>字符的长字符串来触发破坏可利用内存的行为。同样考虑下面的代码片段，它引发了重复字符串分隔符解析的漏洞：

```
void parse (char *inbuf)
{
    char cpy[16];
    char *cursor;
    char *delim_index;
    int   length = 0;

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

⊖ http://xforce.iss.net/xforce/alerts/id/advise142

```
    // -2 for null termination and the ':' delimiter
    if (length < sizeof(cpy) - 2)
        strcpy(cpy, inbuf);
}
```

该含有漏洞的解析器处理一个字符串，期望找到冒号分隔符的一个单个实例。当分隔字符被找到时，保存指向该字符串索引的指针。否则，以1来递增变量length。当循环结束时，将所计算的变量length减去1以为空终止符保留空间，然后针对找到的分隔符再减去1，将它与目标字符缓冲区的大小相比较，以确保存在足够的空间来执行后面的strcpy()函数调用。该处理逻辑字符串如name:pedram amini能被正确处理。对于此字符串，解析器将计算出length的值为16，意味着没有足够的空间，于是就跳过函数strcpy()的执行。如果字符串是name:::::::::::::::::::::pedram，情形又怎样呢？在此情况下，计算得到length的值将变为10，于是可以通过条件检查，开始执行strcpy()函数。然而，在strcpy()中所使用的该字符串的实际长度却是32，因此将触发一个栈溢出。

6.4.4 格式化字符串

格式化字符串是相对较为容易被发现的一类错误，模糊器所生成的测试数据应当包括这类字符串。格式化字符串漏洞可以用任意格式化字符串标记来揭露，例如%d，将显示一个十进制数，而%08x将显示一个十六进制数。在进行模糊测试时，格式化字符串标记最好选择%s或者%n（或者二者均选）。这些标记更有利于导致可检测错误的发生，例如内存访问违规。大多数格式化字符串标记都将导致从栈中读取内存，通常这并不会触发漏洞代码中的错误。因为栈被解除引用，%s标记将导致发生大量的内存读取操作以寻找一个表明字符串终止的空字节。在大多数情况下，%n标记提供了最大的可能性来触发一个错误。启发式模糊数据列表中应当包括%s%n的长序列。

利用格式化字符串漏洞的关键

尽管所有其他的格式化字符串标记，包括%d,%x和%s都将导致从栈中读取内存的操作的发生，但%n格式化字符串却具有唯一性，因为它还导致写内存。这是在代码执行中利用格式化字符串漏洞的关键需求。尽管其他被拒绝的格式化字符串标记可以被用来从漏洞软件中“泄露”潜在的关键信息，而利用%n标记则是通过格式化字符串直接写内存的唯一方法。正是因为这个原因，微软决定实现一个控制机制，以在printf函数类中完全支持%n格式化字符串标记。相应的API函数是_set_printf_count_output()[⊖]，若该函数被赋予一个非零值则提供对%n的支持，若赋予零则取消对%n的支持。实际上，%n在产品代码中很少使用，因此默认的设置都不提供对它的支持。

6.4.5 字符翻译

需要关注的另外一个问题是字符转换和翻译，特别是关于字符扩展。例如，十六进制数

⊖ http://msdn2.microsoft.com/en-us/library/ms175782.aspx

0xFE和0xFF在UTF16下被扩展为4个字符。在解析代码中对这样的字符扩展缺乏有效的处理，是存在漏洞的一个重要方面。字符转换同样可能会被错误地实现，尤其是在处理很少被看到或使用的边界数据时。例如微软的Internet浏览器，在从UTF-8转换为Unicode时就被该问题所影响[⊖]。该问题的症结在于，当转换程序为存储转换后的缓冲区确定动态分配存储区的大小时，没有正确地处理5字节和6字节的UTF-8字符。然而，实际的数据拷贝程序却正确处理了5字节和6字节UTF-8字符，因此就导致了基于堆的缓冲区溢出。启发式模糊数据列表中应当包含这些以及其他类似的恶意字符序列。

6.4.6 目录遍历

目录遍历漏洞会影响网络daemon程序以及Web应用程序。通常的一个误解是认为目录遍历漏洞只限于影响Web应用程序。目录遍历漏洞确实在Web应用程序中很常见，然而，这种攻击方式同样也会出现在私有网络协议和其他应用方面。根据2006年Mitre CVE的统计，尽管随着时间的发展，目录遍历漏洞在某种程度上有所减少，但其仍然是在软件应用程序中所发现的第5大类漏洞[⊜]。这些统计包括Web应用程序和传统的客户端/服务器应用程序。可以在开源漏洞数据库（OSVDB）中看到多年来所收集到的许多目录遍历漏洞的例子[⊝]。

例如，考虑Computer Associates公司的BrightStor ARCserve备份软件。BrightStor通过caloggerd程序和TCP协议为用户提供了注册接口，尽管该协议没有公开，通过基本的包分析可以发现注册文件名实际上是在网络数据流中被指定的。用目录遍历修饰符给文件名加上前缀就允许一个攻击者指定一个任意的文件，将任意的注册消息写入到该文件中。当注册程序以超级用户的身份运行时，该漏洞就可以被发现。例如在UNIX系统中，一个新的超级用户可以添加到/etc/passwd文件中。在编写本书时，仍然存留着一个在安全补丁发布前而被了解和掌握的漏洞，计划于2007年7月公开发布。启发式模糊数据列表中还应当包含如../../和..\...\等目录遍历修饰符。

6.4.7 命令注入

类似于目录遍历漏洞，命令注入漏洞通常与Web应用程序有关，特别是CGI脚本。再次重申，认为这类错误只与Web应用程序有关是一个普遍的误解，因为它们可以通过公开的私有协议来影响网络daemon程序。任何目标程序，无论Web应用程序还是网络daemon程序，向形如`exec()`或`system()`的API调用传递未经过滤或者过滤不正确的用户数据，都将潜在地暴露一个命令注入漏洞。考虑下面的简单Python代码片段：

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

⊖ http://www.zerodayinitiative.com/advisories/ZDI-06-017.html

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

⊝ http://www.osvdb.com/searchdb.php?text=directory+traversal

在正常情况下，系统路径被服务器所接收，路径下面所列出的文件是确定的，并且列表将被返回给客户端。然而由于缺乏对输入进行过滤，因此可以指定特定的字符以允许执行额外的命令。对于UNIX系统而言，这些字符包括&&,;和|。例如，参数var/lib;rm-rf/转换为ls/var/lib;rm-rf，该命令可以给被影响系统的管理者带来严重的问题。启发式模糊数据列表中同样应当包含这些字符。

6.5 小结

在本章的开始，讨论了自动化测试的必要性，同时简要列举并描述了可以用来简化开发的不同库和工具。其中，对某些库作了更为详细的解释，并应用到了本书第2和第3部分的自定义工具的开发中。针对数字、字符串和二进制序列，如何选择智能和高效的模糊数据是本章所讨论的一个核心问题。这些知识将在后续章节中得到应用并加以扩展。

在后续章节中，我们将讨论不同的模糊测试目标程序，包括Web应用程序，私有命令行应用，网络服务程序等。在阅读后续章节时要记住本章所讨论的概念。在模糊器的开发过程中可以应用所介绍的不同库和工具。思考一下本章所讨论的智能模糊数据，以及将来会向模糊数据列表中添加哪些值。

第7章 环境变量和参数的模糊测试

"这样的外交策略令人感到一丝沮丧。"

——George W. Bush，引自2002年4月23日《New York Daily News》

可以证明，本地化模糊测试是最简单的一种模糊测试。尽管许多攻击者和研究者将通过发现远程和客户端的漏洞来获取更加重要的结果，但本地私有程度的增强仍然是一个重要的研究课题。即使利用一个远程攻击来获得对目标机器的访问，通常也要使用本地化攻击作为一种辅助攻击手段来获取所需要的私有信息。

7.1 本地化模糊测试介绍

用户可以通过两种主要的方法将变量引入到程序中。除了显而易见的标准输入设备，通常是键盘之外，命令行参数和进程-环境变量也可以表示输入变量。我们首先采用命令行参数的方式来实现模糊测试。

7.1.1 命令行参数

除了最纯粹的Windows用户之外，其他人可能都或多或少遇到过需要命令行参数的程序。命令行参数被传递给一个程序，并通过在C程序中的main函数中声明的argv指针来寻址。变量argc也要传递给main函数，它表示了传递给程序的参数个数，另外还要加上1，因为它所激活的程序名被作为一个参数来计算。来看下面这个简单的例子：

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

当采用不同的参数来多次运行上述代码时，就会得到图7-1所示的结果。

7.1.2 环境变量

用户将变量引入到程序中的另外一种方法是使用环境变量。每个进程都包含由环境变量组成的所谓的环境。环境变量是全局变量，它定义了应用程序的行为。用户可以设置或取消环境变量，但通常都在软件包的安装过程中或被管理员设置为典型值。大多数的命令解释器将导致所有的新进程都继承当前的环境。command.com Shell是Windows中的命令解释器的一个例子。UNIX系统通常有多个命令解释器如sh,csh,ksh以及bash。



图7-1 命令行参数的存储方式

通常使用的一些环境变量的例子包括HOME，PATH，PSI和USER。这些值分别保存了用户的主目录，当前可执行的搜索路径，命令提示符以及当前的用户名。这些特别的变量是很标准的；然而许多其他的公共变量，包括软件提供商所创建的那些变量，只能用于其应用程序的操作中。当一个应用程序需要某个特定变量的信息时，它只需使用getenv函数，该函数将变量名指定为参数。尽管Windows进程和UNIX应用程序以相同的方式来包含环境变量，这里主要关注于UNIX系统，因为Windows没有setuid应用程序的概念，该应用程序可以被无特权的用户所启动，并在执行过程中获取私有信息。图7-2描述了一个典型的UNIX环境。可以通过键入set命令来查看bash Shell中的当前Shell环境。

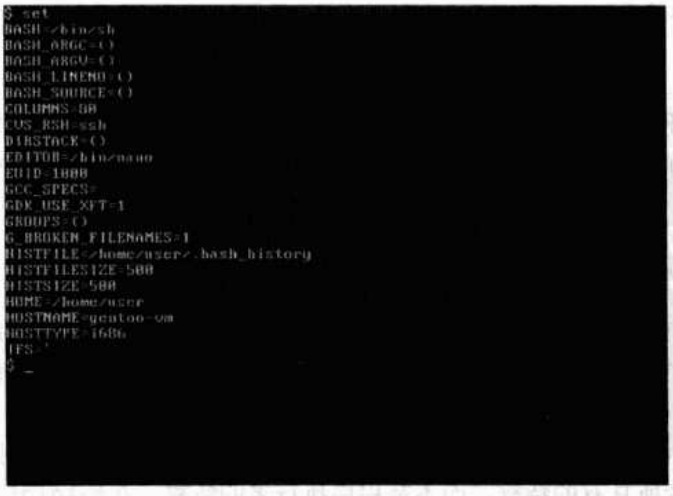


图7-2 bash Shell使用的一些环境变量示例

用户可以使用export命令来操纵该列表中的每个变量。在理解了命令行参数和环境变量的用法之后，就可以开始讨论对它们进行模糊测试的一些基本准则。

7.2 本地化模糊测试准则

环境变量模糊测试和命令行模糊测试的基本思想很简单：如果一个环境变量或者命令行选项包含一个非预期值，那么当该值被接收之后，应用程序将如何做出响应呢？当然，这里只对非正常运行的授权应用程序感兴趣。这是因为本地化模糊测试要求对机器进行本地访问。因此，只需要有限的值就可以使应用程序崩溃，你必须要自己执行拒绝服务的攻击。如果一个环境变量发生了溢出而导致多个用户共享的系统或应用程序崩溃，那么就会带来一些风险。然而，我们最感兴趣的是在授权的应用程序中寻找一个缓冲区溢出，而该溢出将允许一个受限用户来提升其特权。寻找授权目标程序将在本章后面的“寻找目标程序”一节中进行讨论。

许多应用程序都进行了这样的设计，即当其被激活时，能够从用户接收命令行参数。然后，应用程序便使用此数据来决定它应当进行的操作。几乎在所有UNIX系统中都能找到的‘su’应用程序就是一个极好的例子。当用户不使用任何参数来激活应用程序时，它将会对根用户进行鉴别；然而，如果用户指定了一个不同于第一个参数的用户名，那么它将会切换到此用户，而不是根用户。

考虑下面的C语言代码，它简要说明了su命令如何根据不同的参数来采取不同的行为：

```
int main(int argc,char *argv[])
{
   [...]
   if (argc >1)
    become(argv[1]);
   else
    become("root");
   [...]
}
```

命令行参数和环境变量是将变量引入到程序中的两种最基本的方式。对其进行模糊测试的思想很简单。当我们通过命令行向程序输入错误数据时将会怎样呢？这种行为会导致安全风险吗？

7.3 寻找目标程序

当执行本地化模糊测试时，通常在系统中只有少量的所需要的二进制目标程序。这些程序在执行时具有更高的优先级。在基于UNIX的系统中，这些程序很容易被识别，因为它们包含有setuid或者setgid位集。

setuid和setgid位表明当一个程序运行时，它可以要求提升其特权。对于setuid位，进程将拥有文件的所有者所具有的特权，而不是程序执行者的特权。在setgid位的情况下，进程将拥有文件的组所有者具有的特权。例如，成功的开发一个有setuid根用户和setgid组用户的程

序，可能会生成具有这些特权的一个Shell。

使用find命令来构建setuid二进制代码的列表是很简单的，该命令是UNIX和与UNIX类似的操作系统中的一个标准工具。下面的命令可以完整地列出系统中所有的setuid二进制代码。它应当作为根来运行，以防止文件系统的读取错误：

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

find命令是一个强有力的工具，它可以用来发现特定的文件类型、设备驱动程序以及文件系统中的目录。在上面的例子中，我们只使用了find命令所支持的一小部分选项。第一个参数指明了我们将要搜索整个系统以及在根目录/下面的所有文件。type选项告诉find命令我们只对文件感兴趣。这意味着不会返回符号链接、目录或者设备驱动程序。-perm选项描述了我们所感兴趣的特权。-o选项的使用允许find命令使用逻辑操作符或（or）。如果一个二进制文件含有setgid位或者setuid位集，那么其值将为true，并且打印出该文件的路径。总之，该命令将搜索所有含有setuid位（4）或者setgid（2）位集的规则文件。下面的例子描述了在Fedora Core 4的默认安装过程中，该命令的输出内容：

```
[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/lppasswd
/usr/bin/gtali
/usr/bin/wall
/usr/bin/chsh
/usr/bin/passwd
/usr/bin/glines
/usr/bin/gnibbles          ← everyone knows gnibbles is absolutely
                             required for a functional system...
/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
```

UNIX文件许可说明

在UNIX系统中，文件许可模型允许三种基本的不同类型的访问：读、写和执行。对每个文件而言，同样也存在着三种许可访问集，它们分别属于用户、组以及其他类型。在任意给定的条件下，实际上只有这些许可当中的一个被应用。例如，如果你拥有一个文件，则将会被用到的许可集就是该用户。如果你不拥有该文件，而组拥有该文件，那么组许可将会被应用。对于所有其他的情形，其他类型的许可将会被应用。请看下面的例子：

```
-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。那么该用户的许可包括读文件和执行文件。当然，由于该用户拥有文件，因此他或她可以任意修改这些许可。

如果用户组中的其他成员试图访问该文件，则他或她将只能够执行该文件，而不能读取该文件。试图对文件的读取将因为无效的许可而失败。最后，所有其他的用户将被拒绝访问，他们不能读、写或者执行该文件。

在UNIX系统中，存在一种特殊的方式以描述完全的文件许可。在此系统下，许可以八进制的形式被表达。也就是说，每一个许可组合都被赋予一个从0到7的值。读标志的八进制值是4，写标志的八进制值是2，执行标志的八进制值则为1。然后将这些值组合起来就得到了所有的许可。例如，如果一个文件允许用户、组和其他用户进行读和写操作，那么其值就为666。被dude所拥有的示例文件将被表达为值510——用户（5）=读（4）+执行（1），组（1）=执行（1），其他用户（0）=空。

第4列代表特殊标志如setuid和setgid位。setuid位被表达为4，而setgid位则被表达为2。因此，含有setuid和setgid位的文件可能具有值为6755的许可。如果特殊标志列被省略，则将被假定为0，因此将不再具有任何扩展许可。

7.4 本地化模糊测试方法

环境变量和命令行参数可以由用户很方便的提供，由于它们几乎总是简单的ASCII字符串，因此在实际中可以对目标程序实施某些基本的手工测试。最简单的测试可能就是将HOME变量设置为一个长字符串，然后运行目标应用程序以查看其输出。使用Perl可以非常迅速的实现该测试，而Perl在大多数UNIX系统中都是默认可用的：

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

这是一种非常基本的方法来测试应用程序，以查看其是否能够处理一个长的HOME变量。然而，这个例子假定你已经知道了应用程序使用了HOME变量。如果不知道应用程序使用了何种变量，情况又如何呢？如何来确定应用程序使用的是哪种变量呢？

7.5 枚举环境变量

至少可以采用两种自动化方法来确定应用程序使用的是什么环境变量。如果系统支持库的预加载，你就可以钩住getenv库调用。提供一个新的getenv函数以实现标准getenv函数的功能，并且也将调用注册到一个文件，那么就可以有效地记录应用程序需要的所有变量。该方法的一个扩展将在本章后面的“自动化的环境变量模糊测试”一节中做更加详细的介绍。

GNU调试器（GDB）方法

可以使用的另外一种方法需要一个调试器的支持。使用GDB，可以在getenv函数的内部设置一个断点，并且输出第一个参数。使用GDB脚本在Solaris10上进行自动化测试的例子如下：

```
(08:55AM)[user@unknown:~]$gdb -q /usr/bin/id
(no debugging symbols found)...(gdb)
(gdb) break 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 exited normally.
(gdb)
```

如果你对GDB中使用的命令还不熟悉的话，下面将简要对其进行介绍：

- break命令在一个特定的函数或地址设置一个断点。这里，使用break命令在对getenv进行调用时，中断应用程序的执行。
- command命令指定了当遇到断点时将会发生的特定的行为。在这种情况下，让程序停止执行，然后使用x/s将i0寄存器的值作为一个字符串来输出。在SPARC系统上，i0寄存器中保存了被输入的该函数的第一个参数。
- 下一个命令是继续执行程序，因此我们不必在每个断点之后都通知程序继续执行。可以使用run命令的一个快捷键来开始程序的执行。

使用这种方法，可以立即看到被/usr/bin/id程序所请求的11个环境变量的列表。注意，该方法应当在所有的系统上都能使用；然而，需要改变正在解除引用的寄存器的名字，因为不同的体系结构对它们的寄存器使用不同的记忆方法。例如，在x86架构下可能会输出$eax寄存器，在SPARC系统中输出$i0寄存器，而对于MIPS则输出$a0寄存器。

现在，我们已经掌握了目标程序将会使用什么样的变量，下面将研究采用更加自动化的方式对它们进行测试。

7.6 自动化的环境变量测试

回顾一下在上一节“枚举环境变量”中简要提到的使用库预加载的技术，这对于自动化的模糊测试也是有用的。为了检索一个环境变量的值，必须要激活getenv函数。如果我们钩住getenv函数，并且对调用它的所有函数返回长字符串，则甚至不需要知道所使用的变量的列表；只需要通过截取所有对getenv的调用来模糊化每个变量。当执行一个快速检查以发现不安全的环境变量使用时，这是很有用的。

下面的函数是getenv函数的一个简化实现。它使用了指向环境起始点的全局变量environ，该段代码简单地遍历environ数组，以查看被请求的值是否存在于environ数组中。如果存在，则返回一个指向该值的指针；如果不存在，则返回NULL以表明该变量没有被设置。

```
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++;
  }
```

库预加载

我们要讨论的下一个主题是库的预加载。库预加载本质上是通过使用操作系统链接程序来用用户提供的函数替换函数，从而提供了一种简便的方法来钩住函数。尽管该方法在不同的系统之间存在着差异，但基本思想是一致的。通常，用户为其所提供的库的路径设定一个特定的环境变量。这样，当程序执行时该库就被加载。如果该库包含程序中符号的副本，那么将用这些副本来替换原始符号。当提到符号时，这里主要是指函数。例如，如果一个用户使用strcpy函数创建了一个库，并且在运行二进制代码时将其预加载，那么二进制代码将调用用户版本的strcpy函数，而不是系统的strcpy函数。这样做是为了许多合理的目的，例如包装调用以实现切片和审核，同样对发现漏洞也具有一些作用。考虑包装getenv程序或完全替换该程序；该程序被用来请求环境变量的值。

下面的函数是getenv的一个简单替换，可以被用来发现简单的长字符串问题。可以通过使用库的预加载技术来强制该函数改写实际的getenv：

```
#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，该函数对每一个变量请求都会返回一个长字符串。这里完全没有使用environ数组，因为我们不关心所返回的正确值。

Dave Aitel的GPL sharefuzz工具使用了该方法，该工具被用来发现setuid程序中的大量漏洞。为了初始化该模糊测试，只需要将C代码编译到一个共享库中，并使用用户操作系统库的预加载功能（假定操作系统具有此功能）。对于Linux而言，可以通过如下的两步操作来完成：

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

当/usr/bin/target执行时，所有对getnev的调用将使用修改后的getnev函数。

7.7 检测问题

现在，你已经熟悉了本地化模糊测试的基本方法，下面你需要进一步了解当目标程序中发

生感兴趣的异常行为时，如何来识别所发生的问题，这在很多时候都是显而易见的。例如，程序可能会崩溃，并输出“段错误”或者其他一些严重的信号消息。

然而，由于我们的最终目标是实现自动化，因此不能依赖于用户手工识别错误。为了实现自动化，我们需要一种可靠的、可编程的方法。至少存在两种比较好的方法来实现此目的。最简单的一种方法是检查应用程序的返回代码。在现代的UNIX和Linux系统中，如果一个应用程序因为一个未处理的信号而终止，那么Shell的返回代码将等于128加上该信号数字。例如，一个段错误将导致Shell接收十进制数为139的返回代码，这是因为SIGSEGV的值等于11。如果应用程序因为一个非法指令而终止，那么Shell将接收一个值为132的返回代码，因为SIGILL的值为4。这其中的逻辑很简单：如果Shell关于应用程序的返回代码是132或者139，那么就标志着一个可能的错误。

你可能也会考虑到异常中断信号。由于在新版本的glibc中所介绍的更加严格的堆检查，使得SIGABRT更加引人注意。异常中断是这样一个信号，它可以在进程中产生以终止进程的执行并输出核心信息。尽管在特殊情况下进程可能在堆破坏处发生中断，但是有许多巧妙的方法可以将其进行扩展。

如果你使用攻击性的Shell脚本来执行模糊测试，那么使用Shell的返回代码将会变得有意义。然而，例如你正在使用一个用C或其他语言编写的适当的模糊器，那么可能就要使用wait或者waitpid函数了。在这种方式下进行模糊测试的一般方法是，在前面是由fork和一个execve函数构成的子进程，后面跟着wait或者waitpid函数构成的父进程。当使用正确时，就可以通过使用wait或者waitpid来检查所返回的状态，从而很简便地确定子进程是否崩溃。下一章将介绍从本地化模糊测试工具iFUZZ中所简化的一个代码片段，以说明该方法的使用。

如果你关注于捕获可能被应用程序处理的信号（没有被前面所讲的方法所发现），那么除了钩住signal程序之外，至少还有一种可选的方法。你将要使用系统的调试API连接到该进程，并且在信号处理器被激活之前，截获它所接收的信号。对大多数的UNIX操作系统而言，你将会使用到ptrace。通常的方法是一个fork和一个ptrace以及execve构成父进程，后面跟着waitpid和ptrace构成的子进程，形成一个循环，以持续不断的监视进程的执行情况，截获并分析可能产生的信号。每当waitpid在子进程中返回时，就意味着程序已经接收到了一个信号，或者程序已经终止运行。此时，必须要检查由waitpid所返回的状态，以确定到底发生了什么事件。同时，必须要显式的告知应用程序继续执行，并且在大多数情况下都传递信号。该操作也可以使用ptrace来完成。在SPIKEfile和notSPIKEfile中的实现可以作为此通用方法的一个参考。这两种工具用来实现文件的模糊测试，将在第12章“文件格式模糊测试：UNIX平台上的自动化测试”中对它们做详细的介绍。下一章提供了一个代码片段用以说明该方法。

在大多数情况下，ptrace方法对于本地化模糊测试而言是足够的。很少的setuid UNIX应用程序为像SIGSEGV和SIGILL等的信号大量使用信号处理器。同样，一旦你开始使用ptrace，那么就意味着这些代码没有必要在不同的操作系统和体系架构中具有兼容性。可以考虑一下如果你正在创建一个不需修改就可以运行在不同平台上的应用程序的情形。

在下一章中，提出了一个简单的命令行模糊器的实现，它被设计为只在具有C编译器的

UNIX系统中进行编译和运行。该工具也包含一个针对getenv钩子函数的共享库模糊器。

7.8 小结

尽管发现本地化漏洞不太受重视，但是对于查找一个私有化的错误而言仍然是非常有价值的。本章阐述了自动化发现这些类型漏洞的一些不同的方法，在下一章中，我们将具体实现其中的某些方法，以实际发现一些错误。



第8章 环境变量和参数的模糊测试：自动化

“大规模杀伤武器已经存在于某些地方。”

——George W. Bush, Wahington, DC, 2004年3月24日

本章介绍了针对本地化应用程序实施模糊测试的一个程序，即iFUZZ。这里的主要测试目标是在第7章“环境变量和参数的模糊测试”中所讨论的setuid UNIX程序中的命令行参数和环境变量。在本章中，将讨论iFUZZ的特性，解释其设计决策，并且讨论iFUZZ如何被用来发现IBM AIX5.3中的大量的本地化漏洞。

8.1 iFUZZ本地化模糊器的特性

iFUZZ包含你能够想象得到的关于本地化模糊器的一些特性。这些特性分别是，存在一个引擎自动化处理不同的二进制目标代码，为了简化错误再现而输出C语言触发器的功能，实现模糊测试的一些不同的方法以及模块化实现等。在iFUZZ的这些特性中，最方便的一个是它不经修改就能够运行在几乎任何UNIX或者类似于UNIX的操作系统中。它已经在IRIX，HP-UX，QNX，MacOS X和AIX系统中发现了一些漏洞。该模糊器针对每一种类型的模糊测试都具有不同的模块：

- Argv fuzzer模块。前两个模块是非常类似的，因此可以放在一起来说明。它们将可执行程序的argv[0]和argv[1]的值进行模糊化。这两个模块的基本用法很简单：指定目标应用程序的全路径并使其开始运行。然后它将试图使用不同长度的字符串以及特殊格式的字符串。实际所使用的字符串依赖于模糊字符串数据库，可以用终端用户所提供的简单字符串作为补充。
- Single-option/multioption fuzzer（单选/多选模糊器）模块。你可以将此模块认为是一个闭塞的模糊器；也就是说，用户并不向该模块提供有关目标应用程序的任何信息，并且该模块也不关心。它只需要在目标程序中为每一个可能的选项抛出一个字符串值。若使用单选模糊器，iFUZZ将从a循环到Z，并且试图运行下面的命令：

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

FUZZSTRING是从iFUZZ内部的模糊测试字符串数据库中取出的一个字符串。这是发现简单的和选项相关的问题的一种快速方法，但是不能发现更加复杂的问题，例如需要多个选项值的那些问题。

- getopt模糊器。该模块需要从用户方面得到某些信息。也就是说，它需要知道应用程序通过getopt所使用的选项字符串，这样，它就可以将应用程序将要使用的选项进行模糊化。尽管该模糊化可能是非常耗费时间的，但是它比其他的模糊化类型更加彻底。使用

像这样的一个模块，你可能会发现比使用其他的模糊器更加复杂的一些问题。例如，考虑一个应用只有被设置了调试和冗余选项，并且提供了一个长的-f参数时，该应用程序才会触发一个缓冲区溢出。下面是针对此问题编写的一个应用程序的输出，说明了该模块的用法：

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

基于所输出的使用信息，我们可以确定针对该应用程序的getopt字符串极有可能形如f:o:vds。这意味着选项f和o是作为一个参数，而选项v,d和s只是作为转换。这是如何得知的呢？根据getopt的使用手册：

> 选项参数是一个字符串，它指定了对应用程序而言是有效的选项字符。该字符串中的一个选项字符的后面可以跟着一个冒号（':'），以指明它获得了所需要的参数。如果一个选项字符的后面跟着两个冒号（'::'），则表明其参数是可选的；这就是一个GNU扩展。

如果使用f:o:vds以getopt模糊方式来运行iFUZZ，那么很快将会发现我们所描述的漏洞。由于iFUZZ也会随机的向一个现存的文件提供路径以作为模糊字符串，你甚至可以发现需要其中某个选项而成为有效文件的那些漏洞。由于iFUZZ所采用的设计方法，使得能够较容易的对它进行创造性使用。也可以向字符串数据库中添加一些其他的有效字符串，例如用户名、主机名、有效的X显示等。你所能进行的创造性工作越多，那么你的模糊器的功能就越强大。

iFUZZ的内部也包含有一个简单的可预加载的getenv模糊器。它包含你期望返回的、未修改的一个变量数组，但却模糊化其他所有的数据。它只是对Dave Aitel的共享模糊器增加一些额外的功能，如果需要的话，它允许你从实际变量中返回数据。它实际上不是iFUZZ的一个核心构件，更像是一个快速执行的工具。

iFUZZ内部同样包含一个简单的getopt钩子，它可以被预加载以从目标二进制代码中输出getopt选项字符串。尽管这是有用的，但实际上它是个只有一行代码的C程序，只是为了便于使用而被包含。当使用该钩子时，要记住某些应用程序使用它们自己的函数来解析命令行选项。对于这些应用程序而言，必须要基于使用函数的输出来手工构建getopt字符串。

8.2 iFUZZ的开发

前面列出了iFUZZ工具所具有的特性，下面将要介绍一些具体的实现细节。为了全面的理解iFUZZ的构造过程，建议读者从fuzzing.org下载并分析该工具的源代码，这里只是重点阐述一些关键功能，并分析一些特定的设计决策。

开发方法

iFUZZ所采用的开发方法是模块化和可扩展化。这使得iFUZZ能够很方便地确定应用程序的基本功能，而不用在前面所讲的各种模糊测试方法上花费大量的时间。一旦主引擎被开发完成，实际上就开始了各个模块的开发。

在主引擎和帮助函数开发完成之后，第一个被开发的模块是argv[0]模糊器。一个测试集可以在若干个不同的操作系统，包括QNX，Linux和AIX上使用，该测试集是针对argv[0]溢出漏洞的二进制代码、格式化字符串以及被安全编写的应用程序的。该测试集用来消除任何的小错误并确定准确的输出。

作为一个开发者，你并不总是具有以前存在的具有漏洞的二进制代码的知识，以将其用于你的测试集中，因此你可以创建自己的一些具有漏洞的应用程序例子。例如，下面代码采用了一个最简单的方法，创建了一个具有argv[0]格式化字符串漏洞的应用程序：

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

可以将该段代码进行编译并放进一个包含一些已知的安全二进制代码的测试集中，以发现漏报和误报漏洞的问题。

将格式化字符串作为一种攻击手段

格式化字符串漏洞已经存在一段时间了，而它们曾经被认为是没有危害的。1999年，在经过了对proftpd1.2.opre6的安全性审核之后，Tymm Twillman向BugTraq邮件列表[⊖]发布了一个研究细节，即该问题允许攻击者使用格式化字符串漏洞来改写内存。该漏洞是由snprint()函数所导致的，该函数不使用格式化字符串传递用户输入[⊜]。

接下来被实现的模块是单选项模糊器。尽管它的基本思想也很简单，但将其用于许多商用UNIX系统时是非常有效的。它的设计思想很简单，并且不具有实际的挑战性。使用一个简单的循环来遍历整个字母表，每次使用被选择的字符作为程序的一个选项。该选项的参数取自于模糊数据库。

当这些简单模块被实现之后，iFUZZ就可以发现一些漏洞了，但是却发现不了任何更加复杂的漏洞，如那些需要多个选项才能被发现的漏洞。这就引入了接下来的两个模块的开发，即多选项和getopt模糊器。

多选项模糊器的设计思想很简单，它实际上是在一个单选模糊测试循环内嵌套了另外一个单选模糊器。可以在命令行中指定在任意时间所使用的选项的最大数量。所指定的选项越多，则循环的深度就越大。

⊖ http://seclists.org/bugtraq/1999/Sep/0328.html

⊜ http://en.wikipedia.org.wiki/Format_string_attack

为了使多选模糊器更加有效，并且提高执行的效率，getopt模糊器应运而生。它与多选模糊器基本上是相同的，所不同的是它不再试图将所有值作为选项，而是给出了对应用程序而言有意义的值的子集。尽管该技术降低了模糊器功能的全面性，但却提高了其执行速度，并且能够更快的发现漏洞。

我们设计了两种简单的方法来捕获异常。第一种方法将不会捕获已处理的信号，但正如在上一章中所提到的，在UNIX下只有很少的应用程序会处理我们想要捕获的信号。

Fork,Execute和Wait方法

下面列出了fork,execute和wait方法的一个简单的实现：

```
[...]
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代码片段说明了如何来控制并监视一个进程，以找到感兴趣的信号，即使这些信号被应用程序在内部处理。这个例子是从notAPIKEfile和SPIKEfile所共享的代码中截取的一个片段，文件模糊测试工具notAPIKEfile和SPIKEfile将在第12章“文件格式模糊测试：UNIX平台上的自动化测试”中讨论。

```
[...]
     if ( !(pid = fork ()) )
     { /* child */
      ptrace (PTRACE_TRACEME, 0, NULL, NULL);
      execve (argv[0], argv, envp);
```

```
      }
       else
      { /* parent */
       c_pid = pid;
monitor:
            waitpid (pid, &status, 0);
            if ( WIFEXITED (status) )
            { /* program exited */
                 if ( !quiet )
                   printf ("Process %d exited 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 )
                   fprintf (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) == SIGTRAP) ? 0 : WSTOPSIG (status))) == -1 )
             {
                   perror("ptrace");
             }
         goto monitor;
        }
        return(ERR_OK);
 }
```

8.3 iFUZZ的开发语言

开发iFUZZ所选用的是C语言。尽管可以琢磨出一些选用C语言不恰当的理由，但是不能掩盖这样一个事实，即C语言被选中主要是因为我对它情有独钟并且经常使用它。

然而，C语言除了成为我的首选语言之外，还具有某些优点。例如，大多数的UNIX系统，即使是早期的系统，也都安装有C编译器组件。但对于Python或者Ruby等语言可能就不是这样。Perl语言可能也会作为一个选择，因为它在UNIX系统中有着非常广泛的应用。然而，Perl代码非常难以维护。

在开发的时候，为像这样的项目选择一种如Python或Perl等的语言所带来的益处是非常明显的。使用这些脚本语言，可以极大的缩短开发时间。

最终，由于我们对C语言的熟悉程度，以及想要开发一个不像用其他脚本语言构造的更小的模糊器那样的攻击性模糊器，使得选定了C作为开发语言。

8.4 实例研究

使用iFUZZ，可以识别IBM AIX5.3中超过50个的本地化漏洞，每个漏洞对于彻底测试者或黑客解密者而言都具有不同程度的有用性。这些漏洞中的绝大多数都可以很容易发现，它们只是隐藏在argv[0]和argv[1]的后面。然而，有一些漏洞更加有趣，并且发现的难度更大。当我们提到难度时，并不是意味着找到这些漏洞需要任何程度的技巧。实际上意味着当准备iFUZZ命令行选项以及等待iFUZZ的输出结果时，需要更加的耐心。下面的两个漏洞（发生在一个setuid根二进制代码中）说明了iFUZZ在系统中作为本地化模糊器所具有的作用和有效性：

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

尽管因为开发此代码需要访问printq组，使得这个例子中的目标二进制代码并没有对系统的安全性带来严重的威胁，但只是为了触发其中的漏洞，所以仍将其作为一个示例。

为了发现这些问题，阅读了该二进制代码的手册，并且为iFUZZ生成了一个getopt字符串。所使用的getopt字符串是a:A:d:D:p:q:Q:s:r:w:v:

下面列出的第一个命令的输出显示了该二进制代码的位置和许可：

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

经过一长段的运行时间之后，iFUZZ将会输出至少两个该二进制代码中有趣的和可能开发的漏洞。在前面所展示的漏洞中，LONGSTRING表示一个大约有20 000个字符的字符串，X表示任意合理的字符串，例如字符串x自身。X必须要和长字符串一起使用来到达存在漏洞的代码。

尽管发现像前面那样触发难度较大的漏洞是很有意思的，但通过遍历iFUZZ的所有基本模块来查看能够发现多少个简单的错误同样是很有趣的。在AIX5.3的setuid二进制代码中以argv[0],argv[1]或者单选模式来运行iFUZZ，那么你将会花费许多时间并发现足够多的堆溢出错

误，栈溢出错误以及格式化字符串问题。自从这些漏洞被作者和其他一些独立的研究者使用iFUZZ发现之后，其中的许多漏洞已经向IBM做了报告，并得到了修复，这些研究者都使用了类似的模糊测试的概念来发现这些漏洞。

8.5 益处和改进的余地

在阅读本章之后，希望你对iFUZZ中有用的东西产生一些自己的想法。同样重要的是你已经认识到了iFUZZ的一些弱点。下面列出的是在iFUZZ的开发之后所做的一些观察。

- 首先，iFUZZ除了使用一些硬编码的休眠值来试图控制加载之外，并没有考虑到使系统崩溃的可能性。这将是一个很好的特性，即它能够分析系统加载以查看该休眠值是否需要增加以阻止系统挂起或者出现漏报和误报的问题，而这些问题在过载的情况下可能会发生。
- 期望有一种特别的选项，它可以确定触发系统崩溃所需要的近似字符串长度或者绝对的最小值。这是一个微不足道的特性，但在很多情况下可以节省时间。
- iFUZZ所不具备的另外一个小的特性是在系统中自动定位所有的`setuid`和`setgid`二进制代码的能力。这是另外一个可以节省时间的特性。
- 从一个给定的二进制代码中解析选项的能力将是一个重要的特性，实现该特性是很重要的。如果用户可以指定一个二进制代码，并且iFUZZ可以自动确定应用程序需要哪些标志和选项，那么将会生成一个智能的、完整的模糊器，并且需要最少的用户交互。当然，这并不是一个简单的实现，因为许多应用程序使用许多不同的格式。即使该特性得到了一个相当完整的实现，那么包含一种新类型使用函数的应用程序也不能被当前的实现代码所解析。也就是说，将该特性设置为可扩展的，这样用户就可以为他们所遇到的不同应用程序来添加新的格式。
- 我们所认为的微不足道、无价值的实现就是C代码生成能力。在系统崩溃后能够生成C语言程序就意味着漏洞可以被手工及时地再生。所生成的代码甚至可能会被用做发掘漏洞的基础。这将节省漏洞发掘者的时间，特别是当为触发漏洞而编写单独的C程序的时间开始增加时，对含有许多漏洞的脆弱系统进行模糊测试非常节省时间。

8.6 小结

以前，远程漏洞被给予了太多的关注，现在，客户端漏洞和本地系统的安全问题相对来说存在着更多的研究空间。通过实现一个本地化模糊器，可以使用很短的研究时间来揭露这些研究内容以及更加复杂的漏洞，这也证明了模糊测试是非常有价值的。

第9章 Web应用程序和服务器模糊测试

"我是低期望的主人。"

——George W. Bush，于"aboard Air Force One"，2003年6月4日

现在，我们从本地化模糊测试转向对客户机-服务器体系架构进行模糊测试。特别的，我们主要关注于Web应用程序和Web服务器的模糊测试。正如我们所讨论的，对一个Web应用程序进行模糊测试也可以发现其底层的Web服务器中的漏洞，但是为了简便起见，我们将这种类型的模糊测试只作为Web应用程序的模糊测试。尽管其基本概念与前面所讨论的网络模糊测试基本上是一致的，但我们还是要做出一些改动。首先，Web应用程序的输入数据量很大，并且输入经常处于不明显的位置，因此需要对输入向量的组成进行重新定义。其次，需要改进漏洞检测机制以能够捕获并分析Web应用程序所产生的错误消息，揭露一个漏洞条件。最后，为了使Web应用程序的模糊测试实用化，需要建立一种通过网络发送输入的体系架构来弥补性能上的损失。

9.1 什么是Web应用程序模糊测试

Web应用程序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（在第14章"网络协议模糊测试"中所讨论）将任意的网络包进行变异，而Web应用模糊测试则特别关注于遵循HTTP规范的包。鉴于Web应用在当今世界的广泛应用和具有的重要性，我们将Web应用模糊测试作为一种单独的方法来论述。目前，相对于传统的软件产品直接安装到本地计算机上而言，提供商更多的是通过Web将软件作为一种服务发布。Web应用程序可以部署在使用它们的实体上，也可以部署在第三方的机器上。当包含一个第三方机器时，通常称之为应用服务提供者（Application Service Provider,ASP）模型。

Web应用为软件提供商和终端用户都提供了许多的优越性。为提供商带来了利润并允许及时的更新，包括安全性修复。终端用户不再需要下载并应用补丁，因为应用程序被驻留在一个中心服务器上。从一个终端用户的视角来看，这也意味着减少了应用程序的维护开销，因为维护工作是由ASP来完成的。即委托ASP来维护适当的安全性以保护你的私有数据免受攻击。因此，Web应用的安全性应当作为关注的一个重点问题。

微软LIVE

微软在Web应用不断增强的重要性方面下了很大的赌注。尽管其传统的产品是GUI应用程序如微软Office，但在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[⊖]。Windows Live是针对个人服务的一组集合，而Office Live则面

⊖ http://news.com.com/2061-10805_3-6026895.html

向小的商业应用。微软将通过广告和订购模型来从这两个服务中获取利益。服务的Live集最早出现在2002年的Xbox Live的发布中，Xbox Live是Xbox视频游戏控制台的一个在线社区和市场。

开发Web应用的编程语言和开发环境的用户友好性都在不断地提高。因此，现在许多公司都在开发他们自己的Web应用以在企业内部的互联网络中使用，并作为一种手段来方便与商业伙伴和客户之间的交流。现在，任何公司不论其规模大小，实际上都可以被认为是一个软件开发者。由于技术所带来的先进性，甚至一个人的公司也有能力开发并部署Web应用。然而，尽管开发一个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。当一个Web应用由软件开发实力不是很强的公司所开发时，那么该应用在部署之前很难通过严格的安全性测试。不幸的是，测试Web应用安全性的那些易于使用并且免费的程序在技术上要落后于所开发的Web程序。本书的目标之一就是要改变这种状况。下面简要介绍Web应用程序开发过程中常用的一些技术。

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 HTTPd Web服务器而开发的一个标准。CGI定义了数据应当如何在Web客户端和代表Web服务器来处理该请求的应用程序之间进行传递[㊀]。尽管CGI可以使用任何一种语言，但绝大多数都是用Perl来实现的。

PHP

超文本预处理器（Hypertext Preprocessor, PHP）[㊁]是一种经常用来开发Web应用的流行的脚本语言。解释器的应用范围很广以允许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页中以处理需要动态生成的Web页部分。

Flash

Flash最初是由FutureWave软件公司所开发的，该公司于1996年12月被Macromedia公司所收购[㊂]。在这里所谈及的技术中，Flash在某种程度上具有唯一性，因为它要求安装独立的客户端软件。而提到的其他大多数技术则只需要一个Web浏览器。

Macromedia通过将Macromedia Flash播放器变为免费可用的，从而能够扩大Macromedia Flash被接受的范围。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，而Macromedia Flash远程化则允许Flash播放器和Web服务器应用之间进行交互[㊃]。Macromedia自身于2005年被Adobe System公司收购[㊄]。

㊀ http://en.wikipedia.org/wiki/Common_Gateway_Interface

㊁ http://www.php.net

㊂ http://en.wikipedia.org/wiki/Adobe_Flash

㊃ http://www.macromedia.com/software/flashremoting/

㊄ http://en.wikipedia.org/wiki/Macromedia

JavaScript

Netscape于1995年后期创建了JavaScript，作为一种手段以允许在Web页中处理动态内容[⊖]。JavaScript可以用作一种客户端或者服务器端技术。当应用在客户端时，JavaScript嵌入到发送给Web浏览器的HTML中，并且直接被浏览器解释。JavaScript也可以被应用为一种服务器端技术，Web服务器在创建动态内容时使用该技术。其他的服务器端技术如微软的ASP.Net（见下面）对包含JavaScript提供相应的支持。

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，当发现Oak已经被注册了商标之后，它的名字改为了Java[⊜]。Java是编译型语言和解释型语言的一个交叉产物。Java的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的Java虚拟机进行解释。这种机制使得Java具备了平台无关性。它经常被Web服务器使用以发布复杂的交互式应用。

ASP.Net

.Net是一种开发平台，而不是一种语言。它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#在内的许多语言所使用。微软于2002年引入了.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具有类似性，因为其源代码也编译为称作公共中间语言（Common Intermediate Language,CIL）的一个中间字节代码，然后该中间代码再被虚拟机解释[⊝]。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。

9.2 目标应用

Web应用模糊测试不仅可以发现Web应用自身的漏洞，而且还可以发现其底层任何构件中存在的漏洞，包括可能和Web应用集成在一起的Web服务器和数据库服务器。虽然Web应用所包含的应用程序类型非常广泛，但仍然存在着一个公共的分类体系。下面就列出了这样的分类情况，并且给出了特定应用中存在的漏洞的一个示例，这些漏洞可以使用模糊测试技术被潜在地识别：

- Web邮件

微软Outlook Web Access Cross-Site脚本漏洞

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

⊖ http://en.wikipedia.org/wiki/Javascript

⊜ http://en.wikipedia.org/wiki/Java_programming_language

⊝ http://en.wikipedia.org/wiki/Microsoft_.Net

• Wikis

Tikiwiki tiki-user_preferences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

IpSwitch WhatsUp Professional 2005（SPI）SQL注入漏洞

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

多提供商Cacti远程文件包含漏洞

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

以上并不是所有Web应用类型的一个完整列表，但是它对通过Web来发布自身应用的那些应用程序类型提供了一种说明，并通过实例阐述了它们可能会具有的漏洞类型。

9.3 测试方法

在开始对一个Web应用进行模糊测试之前，必须首先要建立目标环境，然后为目标环境选择输入向量。Web应用在这两方面都具有一些特有的挑战。按照设计，Web应用的体系架构可以被部署在多个网络机器上。当在开发环境中部署这样一个应用时，尽管具备了必要的可测量性，但却带来了模糊测试中性能上的下降。除此之外，Web应用的输入可以采用多种方式来掩饰，这些都可导致漏洞的产生。因此，当定义模糊测试所需要的输入时，必须要采取更加灵活的方法。

9.3.1 建立目标环境

模糊测试通常要求能够快速、连续地向目标程序发送大量的输入。一个单一输入的事件序列包括：在本地产生输入，将输入发送到目标应用，允许目标应用对输入进行处理，监视输出结果。因此，运行一个模糊器所需要的时间也就由该序列中运行速度最慢的环节所决定。当对一个本地应用进行模糊测试时，整个测试过程的瓶颈是CPU周期以及硬件的读/写时间。在现代计算机硬件速度的支持下，这些时间可以减少到最小，因此模糊测试对于研究漏洞是一个可行的方法。

对于Web应用模糊测试而言，瓶颈点在于由模糊器向目标应用所进行的网络包的传输。考

虑一下从远程地点加载一个Web页的过程。当浏览一个Web页时，页面的显示速度是由以下三个速度来决定的：你的计算机，该页所在服务器，以及位于二者之间的Internet连接。你只能控制这三个变量中的第一个变量，即你自己的计算机。因此，当对一个Web应用进行模糊测试时，通过去除其他两个变量来提高网络通信的速度是非常重要的。在可能的情况下，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，这样包就不需要通过网络进行传输了。大多数的桌面操作系统如WindowsXP[⊖]或Linux都具有内嵌的Web服务器，因此，通常的选择是将目标应用安装并配置在本地机器上。

当对一个Web应用进行模糊测试时，一个虚拟机（VM）应用，如VMWare[⊜]或微软的虚拟机[⊜]也是一个很有用的工具。使用一个虚拟机，当模糊器在本地运行时，被测目标Web应用可以运行在一个VM实例中。这种方法提供了在本地运行目标应用所不具备的一些优越性。首先，目标应用所消耗的大量资源可以通过VM来更好地管理。这就确保了在模糊测试过程不会消耗掉所有的机器资源。其次，引起系统崩溃和拒绝服务攻击的网络通信将不会影响模糊器所运行的本地机器。最后，这些正是我们试图要发现的异常，如果机器持续加锁那么将很难做到这一点。

9.3.2 输入

在开始模糊测试之前，首先必须要确定Web应用所提供的不同输入，因为这些输入将成为模糊测试活动的目标。请看下面的问题：你认为输入应当包含什么呢？很明显，Web表单中的数据字段可以作为输入，但是URL自身或者发送给Web服务器的cookies能否作为输入呢？Web请求中的头部又如何呢？答案是所有这些内容构成了输入。按照我们的观点，一个输入认为是发送给Web服务器并被Web服务器解释的所有信息。

我们将马上对所有可能的输入进行分类，但首先让我们分析一下产生Web请求的过程，以便更好的理解其机理。多数人使用Web浏览器，如微软的Internet Explorer或Mozilla Firefox，来访问一个Web页。当一个Web浏览器配置好之后，只需简单地在地址栏中输入URL地址就可以访问Web页的内容了。

然而，Web浏览器隐藏了当请求一个Web页时所发生的许多实际活动。为了更好的理解所有这些活动，让我们手工的请求一个Web页。可以通过使用包含在大多数现代操作系统中的telnet程序来完成请求。telnet连接到你所选择的一个主机和接口，然后开始发送和接收TCP包。为了请求一个页面，打开控制台窗口并输入下面的命令：

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

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

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

⊜ http://www.vmware.com

⊜ http://www.microsoft.com/windows/virtualpc/default.mspx

(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。然而，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。接下来的代码行则表示了HTTP协议所要求的最小化请求。首先，将使用的请求方法 (GET) 告知给服务器。本章后面将详细介绍不同的请求方法。然后发送正在请求的路径和/或Web页。在这种情况下，我们请求服务器所请求的默认Web页 (/)，而不是请求一个特定的Web页。同样在该行中，向Web服务器指明了所要使用的HTTP协议的版本 (HTTP/1.1)。下一行指定了主机，这在HTTP1.0中是可选的，但在HTTP1.1中是强制使用的[⊖]。当该请求被提交时 (注意需要使用两个回车来完成请求)，服务器将会返回类似于下面所示的响应：

```
HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html
Set-Cookie: PREF=ID=56173d883ba96ae9:TM=1136763507:LM=1136763507:S=W43uFkQu1vexo
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>
<body bgcolor=#ffffff text=#000000 link=#0000cc vlink=#551a8b
alink=#ff0000  topmargin=3 marginheight=3>
 <center>
[snip]
 <a href=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>
```

所接收到的响应是Web页的HTML源代码，前面的一系列头信息向浏览器提供了有关响应的额外信息。如果你将该响应的HTML部分保存到一个文件中，并且用一个Web浏览器将其打开，那么你将会看到所显示的页面，该页面同简单地使用URL所获取的页面是一样的。然而，假定使用的是相对链接而不是绝对链接，那么其中可能会丢失一些图像。

我们已经提到，前面这个请求代表了HTTP协议所要求的最小格式化请求。那么还能向Web

⊖ http://rfc.net/rfc2616.html#s14.23

服务器提供哪些其他输入呢？为了回答这个问题，首先来分析使用Internet Explorer Web浏览器所发送的请求。使用Ethereal，我们可以发现下面的请求：

```
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=32a1c6fa8d9e9a7a:FF=4:LD=en:NR=10:TM=1130820854:LM=1135410309:S=b9I4GWDAtclpmX
BF
```

所有其他额外的头信息的含义是什么呢？HTTP协议定义了许多头信息，而每个头信息则有一些可以接受的值。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）[⊖]。这里并不试图阐述整个文档，而只对前面所遇到的一些示例头信息做一下解释：

- Accept:*/*

Accept头信息指定了可以在响应中使用的媒体类型。在上面的例子中，表示所有的媒体类型（*/*）都是可以接受的。然而，我们可以限制响应只接受特定的媒体类型如text/html或者image/jpeg。

- Accept-Language:en-us

Accept-Language允许用户指定可以在响应中使用的自然语言的类型。在上面的例子中，要求响应使用美国英语。RFC1766中定义了语言标签的正确格式，即识别不同的各种标签[⊜]。

- Accept-Encoding:gzip,deflate

该头信息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，我们所发送的请求表明可以使用gzip[⊝]或者deflate[⊞]编码算法。

- User-Agent:Mozilla/4.0(compatible;MSIE 6.0;Windows NT 5.1;SV1;.NET CLR 1.1.4322;.NET CLR 2.0.50727)

User-Agent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，因为它允许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定义了在WindowsXP SP2上运行的微软的Internet Explorer6.0。

- Host:www.google.com

该头信息定义了为被请求的Web页提供服务的主机和端口。如果没有包含端口，那么服务器将使用默认的端口。这个字段对于服务器而言是很重要的，因为单一的IP地址上可以有多个

⊖ http://rfc.net/rfc2616.html
⊜ http://rfc.net/rfc1766.html
⊝ http://rfc.net/rfc1952.html
⊞ http://rfc.net/rfc1951.html

主机名被提供。

- Connection:Keep-Alive

Connection头信息允许客户端指定连接所需要的不同选项。一个持久的连接允许多个请求被发送，而不用为每个请求打开一个单独的连接。Connection:close意味着一旦响应被发送出去，连接就应当立即被关闭。

- Cookie :PREF=ID=32b1c6fa8e9e9a7a:FF=4:LD=en:NR=10:TM=1130820854:LM=1135410309:s=b9I4GWDAtc2pmXBF

cookie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，当前任务完成之后它们将被抛弃。cookie允许服务器识别请求者。这使得一个网站可以跟踪用户的上网习惯，并且为用户潜在的定制Web页。如果针对一个特定网站的cookie存在于本地，那么浏览器将在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，这更加有利于定义对Web应用进行模糊测试可用的不同输入。正如前面提到的，发送给Web服务器的请求数据的任何部分都可以认为是一个输入。在高的层次上，这包括请求方法，请求的统一资源定位符（Uniform Resource Identifier,URI），HTTP协议版本，所有的HTTP头以及发送的数据。在下面的节中，我们将分析每个组成部分，并为每个成分确定合适的模糊测试变量：

```
[Method] [Request-URI] HTTP/[Major Version].[Minor Version]
[HTTP Headers]

[Post Data]
```

方法（Method）

GET和POST方法是请求一个Web页最常用的两个方法。这两个方法使用名－值对的方式从Web服务器请求特定的内容。可以将它们认为是一个Web请求中的变量。GET方法在请求URI中向Web服务器提交名－值对。例如，http://www.google.com/search?as_q=security&num=10将向google搜索引擎提交一个请求，以告知它我们想要搜索的词为*security*（as_q=security），并且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－值对以字符?开头被追加到URI的后面，每个不同的名－值对之间用字符&隔开。

名－值对也可以使用POST方法被提交。当使用POST方法时，名－值对作为HTTP头被提交，后面跟着其他标准HTTP头。使用POST方法的一个优点是可以发送任意大小的值。尽管HTTP规范并没有特别限制URI的总长度，但是Web服务器和Web浏览器通常对此施加一定的限制。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URI不能被共享以使其他人访问一个动态生成的Web页。例如，人们经常通过定位在一个特定搜索之后生成的Google地图URI来共享地图和方位。猜想一下下面的地图将你引导到何处？

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

正如所提到的，在Web服务器请求中还可以使用其他的一些方法。下面简要描述了其他的

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书藉杂志全免 KEY 提供,，完全无加密。

有效方法：

- 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服务器上载数据。尽管该方法并没有被广泛支持，但是当PUT方法被支持而没有被正确地实现时，就已经发现了相关的漏洞。例如，微软的安全公告MS05-006[㊀]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方法向微软的SharePoint服务器上载数据[㊁]。
- 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实现，但也应当对其进行完善的测试。不正确的使用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返回到发送请求的客户端。这对于调试网络连接性问题是很有用的，因为你可以查看到实际发送给服务器的请求的结构。在2003年1月，WhiteHat Security的Jeremiah Grossman发表了名为《跨站点跟踪》《Cross-Site Tracing,XST》的白皮书[㊂]，该书披露了客户端脚本如何被利用，以允许恶意的Web服务器获取对第三方Web服务器中cookie值的访问权，而该第三方服务器支持TRACE方法。同样，测试应当识别出这样的情形，即TRACE在何处被隐式地支持。
- CONNECT：被保留以供一个代理使用，可以动态转换为一个通道。
- 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潜在的漏洞方法是否被实现。

OPTIPONS方法也可以被用来确定一个Web服务器是否正在运行Internet信息服务器(Internet Information Services）的一个版本，该版本易受到WebDAV中的一个严重漏洞的影响，而WebDAV是为了更加便于Web内容的发布而对HTTP协议所做的一组扩展。该漏洞在MS03-007[㊃]（Windows构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，识别WebDAV是否可用[㊄]。下面列出的服务器响应表明WebDAV是不可用的，因为在Public头中没有列出任何的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
Content-Length: 0
```

接下来的响应包含了完整的头信息，在该头信息中包含了WebDAV所提供的扩展。当WebDAV在运行有微软IIS5.0的服务器上可用时，就表明了如果还没有实现适当的补丁，那么

㊀ http://www.microsoft.com/technet/security/Bulletin/MS05-006.mspx

㊁ http://support.microsoft.com/kb/887981

㊂ http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

㊃ http://www.microsoft.com/technet/security/bulletin/MS03-07.mspx

㊄ http://www.klcconsulting.net/articles/webdav/webdav_vuln.htm

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, MKCOL, PROPFIND,
PROPPATCH, LOCK, UNLOCK, SEARCH
Cache-Control: private
```

*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）。它可以采用绝对URI的格式（例如http://www.target.com/page.html），或者相对URI的格式（如/page.html）。另外，除了定位服务器上一个特定的资源之外，服务器自身也可以被定位。这是通过使用字符*来实现的，OPTIONS方法也需要这种实现。

当对请求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来说明：

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

该路径可以被划分为以下的组成部分：

```
/[path]/[page].[extension]?[name]=[value]&[name]=[value]
```

每个单独的项都可以而且应当被模糊化。其中一些项应当用已知的值来模糊化，这些值会导致前面所描述的一些漏洞，而另外一些项则应当采用随机值以确定服务器是否很好的处理了非预期的请求。随机值也应当包括大批量的数据以确定当服务器解析并解释数据时，是否发生缓冲区溢出。下面将分别研究每个单独的项：

- *路径*（Path）。当对路径项进行模糊测试时，最常被发现的漏洞就是缓冲区溢出和目录遍历攻击。缓冲区溢出通常在发送大量数据的时候被识别出，而目录遍历则可以通过发送连续的../字符序列来发现。在请求被解码之前，应当使用不同的编码模式以绕过输入验证程序。
- *缓冲区溢出示例*（Buffer overflow example）。对于Macromedia JRun5发布之前的JRun4 Web服务器而言，在接收一个过长路径时易出现基于栈的缓冲区溢出[⊖]。当路径长度超过大约65 536个字符时，该漏洞将被触发。这也说明了当进行模糊测试时，包含超长变量的必要性。

⊖ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=360

- 目录遍历示例（Directory traversal example）。3Com的网络监视器应用中的一个漏洞是目录遍历漏洞的一个经典示例[㊀]。该应用使得Web服务器在21700端口监听TCP协议，发现在网络监视器5.0.2和早期版本中，一个包含连续../字符序列的简单URL将允许用户遍历webroot目录。应当注意到，类似这样的漏洞实际上是服务器中的漏洞，而不是应用程序中的漏洞。
- 页（Page）。对公共页名进行模糊测试，可能会发现那些没有受到正确保护的页，以及导致缓冲区溢出漏洞的页。
  - 缓冲区溢出示例（Buffer overflow example）。当过长的请求被发送给扩展名为.htr，.stm和.idc的文件时，微软IIS4.0将面临基于栈的缓冲区溢出的危险。该漏洞在微软的安全公告MS99-019中有详细的描述[㊁]，并引起了许多公开的研究。
  - 信息泄漏示例（Information leakage example）。“3Com OfficeConnect Wireless 11g Access Point”包含一个漏洞，即可以在基于Web的管理界面上访问敏感Web页，而不用提供适当的授权认证[㊂]。因此，请求一个页如`/main/config.bin`将显示该页的内容，而不是一个注册提示，并且包括管理者的用户名和密码等信息。Nikto[㊃]就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以及可能不安全的Web页。
- 扩展（Extension）。与本地文件一样，Web页的文件扩展名通常也指明了生成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（超文本预处理器，Hypertext Preprocessor）。当请求具有未知扩展名的Web页时，就已经在Web服务器中发现了漏洞。
- 名字（Name）。对公共名字项进行模糊测试可以发现发送给服务器的未定义的变量。同样，如果应用程序没有足够的错误处理能力，那么发送非期望的变量可能会导致应用程序失效。
- 值（Value）。对值项进行模糊测试所应采用的合适方法要依赖于Web应用所期望的变量类型。例如，如果合法请求中的名－值对是`length=50`，那么就可以用超长值或者小数字值来对值项进行模糊化。例如，如果提交的长度小于数据的实际长度，那么会怎样呢？当值为零或甚至为负数时情况又怎样呢？一个比应用程序设计的能处理的最大值还大的值可能会导致应用程序崩溃或者发生整数溢出。可以用模糊测试来找到这些问题的答案。如果值项期望的是一个字符串内容，那么就试图发送非期望字符串值以查看应用程序能否很好地进行处理。然后使用逐渐增多的大量数据，以查看是否发生缓冲区溢出或运行崩溃的情况。

㊀ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=300

㊁ http://www.microsoft.com/technet/security/bulletin/MS99-019.mspx

㊂ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=188

㊃ http://www.cirt.net/code/nikto.shtml

• 分隔符（Separator）。在不同的项之间作为分隔符的那些字符（/,=,&,.,:等）也应当被作为模糊测试的目标。这些字符被Web服务器或应用程序解析，如果服务器不能处理非预期值，那么不正确的错误处理就可以导致漏洞的发生。

协议（Protocol）

数字变量可以被用来模糊化HTTP协议的版本号，以提交支持或不支持的HTTP协议版本，以发现某些服务器漏洞。当前最常用的HTTP协议的版本是HTTP/1.1，HTTP协议被划分为主版本和次版本（HTTP/[major].[minor]）。

头（Headers）

所有请求的头都应该被模糊化。请求的头信息具有如下的格式：

```
[Header name]: [Header value]
```

因此，这里有三个可能被模糊化的变量：名字（name），值（value）和分隔符（:）。名字应当使用已知的合法值来模糊化，以确定Web应用是否支持未公开的头信息。不同HTTP协议所支持的头信息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[1]

• RFC 2616-超文本传输协议-HTTP/1.1[2]

值变量可以被模糊化以确定应用程序是否能够正确处理非预期值。

堆溢出示例

2006年1月，iDefense实验室发布了在Novell SUSE Linux企业级服务器9中被远程发现的堆溢出漏洞的细节[3]。该漏洞可以通过简单地提交一个POST请求被触发，只需在该请求的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

[Data to overwrite the heap]
```

Cookies

cookie被存储在本地，当一个请求被发送给cookie以前所存在的那个Web服务器时，cookie将作为一个HTTP头被提交。一个cookie的格式如下所示：

```
Cookie: [Name1]=[Value1]; [Name2]=[Value2] ...
```

同样，名字（name），值（value）和分隔符（:）应当被模糊化。同其他值一样，模糊值也应当受到在合法请求中提交的变量类型的影响。

---

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

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

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

发布数据（Post Data）

正如前面所提到的，名－值对既可以使用GET方法在请求URI内被提交给Web服务器，也可以使用POST方法作为独立的HTTP头提交给Web服务器。数据通过下面的格式发送：

```
[Name1]=[Value]&[Name2]=[Value2]
```

缓冲区溢出示例

iDefense实验室的Greg MacManus在与流行的Linksys WRT54G无线路由器一起使用的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10 000字节的POST请求将会导致缓冲区溢出。尽管这样的漏洞通常很难在一个嵌入的Web服务器中发现，但是设备固件的开源特性，以及使用开发工具来修改固件的有效性，可以有助于来开发漏洞发掘代码。

识别输入

现在我们已经知道了Web应用请求的结构，并且了解了这些请求中可以作为单独输入变量以及潜在的模糊测试目标的各个不同的组成部分。下一步就要执行搜索以识别所有不同的合法输入值。这可以通过手工的方式来完成，也可以用自动化的方法实现。不论采用哪一种方法，我们要尽可能完整地覆盖输入值。当研究应用程序时，我们的目标就是要列出下面所有的输入：

- Web页
- 目录
- 页所支持的方法
- Web表单
- 名－值对
- 隐藏字段
- 头信息
- cookie

识别前面所提到的输入最简单也是效率最低的方法就是使用一个Web浏览器打开一个Web页，然后查看该页的源代码，在源代码中你可以寻找包含在Web表单中的输入。同时要确保搜索隐藏字段（输入类型＝“hidden”），因为懒散的应用程序开发者有时候会将隐藏字段通过隐匿来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，因为它们在页面上是不可见的。这显然是一种非常弱的控制方法，因为隐藏字段总是可以在Web页的源代码中查看到。

当使用Web浏览器来识别输入时，你将不能看到请求/响应的头信息。例如，你将不会看到被发送给应用服务器的cookie的结构，因为它们被包含在HTTP头中，并且被自动地传递给浏览器。为了查看原始HTTP请求，你可以使用一个网络协议分析器如Wireshark[⊜]。

⊖ http://www.idfense.com/intelligence/ vulnerabilities/display.php?id=305

⊜ http://wireshark.org

即使使用一个Web浏览器和嗅探器，一旦Web应用中的页面超过一定数量，那么手工识别所有可能的输入将是不可行的。幸运的是，可以使用一个Web蜘蛛来自动化地跟踪遍历Web应用程序。Web蜘蛛（或者Web爬虫）是一个应用程序，它可以识别出一个Web页内部的所有超链接，遍历这些链接，再发现额外的超链接并遍历，重复执行此过程直到所用可能的Web页都被访问。同时，它还可以发布对安全研究者而言非常重要的信息，如前面所提到的输入信息。幸运的是，目前有许多功能卓越的Web蜘蛛，它们是免费并且开源的。一个简单而功能强大的Web蜘蛛是wget工具[㊀]。尽管它最初是为UNIX操作系统设计的，但wget的端口同样也适应于win32平台[㊁]。我们要推荐的另外一个免费Web蜘蛛包含在WebScarab中[㊂]。WebScarab项目是开放Web应用安全项目（Open Web Application Security Project,OWASP）所提供的一组工具的集合，它们对于分析Web应用非常有用。在WebScarab众多的工具中，有一个Web蜘蛛可以被用来自动化地识别Web应用中所有的URL。另外一个有用的WebScarab工具是一个代理服务器，可以用于手工的Web页的审核。如果设定你的Web浏览器使用WebScarab代理，则在你运行应用程序时，它将记录下所有的原始请求/响应。它甚至包含一个非常基本的Web模糊器，在下一章中，我们将讨论如何创建一个功能更强大的模糊器。

9.4 漏洞

Web应用易于遭受许多类型漏洞的攻击，所有这些漏洞都可以通过模糊测试来加以识别。下面列出了标准的漏洞分类：

- 拒绝服务（Denial-of-service,DoS）：DoS攻击对Web应用而言是一个很重要的威胁。尽管DoS攻击并不使攻击者访问目标程序，但是也拒绝访问你公司的外部入口站点，并且一个收入源的失去可能会导致实质上的金融损失。
- 跨站点编写脚本（Cross-site scripting,XSS）：根据Mitre的统计，XSS漏洞占2006年新发现漏洞的21.5%[㊃]。，这使得XSS漏洞成为了最普遍的漏洞，而不仅仅限于Web应用，但是在所有的应用程序中，XSS漏洞曾经被认为是比安全威胁更加严重的问题，但随着钓鱼式攻击（phishing attack）的爆炸性增长，这种情况已经完全被改变了。XSS漏洞允许一个攻击者控制Web浏览器中客户端的行为，因此成为了执行这类攻击的攻击者的有用工具。
- SQL注入（SQL injection）：在Web应用的漏洞中，SQL注入不仅是最普遍的一种漏洞之一，而且是最严重的漏洞之一。2006年的Mitre统计表明SQL注入漏洞达到了两位数，占新发现漏洞的14%。这种情况的造成，一方面是由于在关系型数据库的助推下动态网页的快速增长，另一方面则是仍然在大多数教科书里所讲述的不安全的SQL编码实践。有一些误解认为SQL注入只限于攻击提供有读取数据库记录途径的应用，因此只是一种机密风险。随着被大多数关系型数据库所支持的功能强大的存储过程的出现，SQL注入将

㊀ http://www.gnu.org/software/wget/

㊁ http://gnuwin32.sourceforge.net/packages/wget.htm

㊂ http://www.owasp.org/software/webscarab.html

㊃ http://cwe.mitre.org/documents/vuln-trends.html#table1

成为更大的风险，它可以导致对后端可信系统的全面威胁。

- 目录遍历/弱访问控制（Directory traversal/Weak access control）：目录遍历曾经是相当普遍的，但有幸的是现在已基本消除。另一方面，弱访问控制仍然是一个存在的威胁，因为它没有使开发者忘记一个受限制的页或者目录，以致不能正确地应用适当的访问控制。这也是Web应用应当被持续不断进行审核的许多原因之一，并且这种审核不仅应当在产品进入市场之前进行，而且应当贯穿于产品的整个生命周期。一个曾经安全的Web应用可能会由于对服务器所做的错误的配置变更突然变得不安全，这种变更即删除或者弱化访问控制。
- 弱认证（Weak authentication）：采用认证模式没有什么缺点，如果没有正确的实现认证，那么一切都可能会不安全。弱的密码保护是实施强有力攻击的一个漏洞，在明文本中传递信任信息使得它们在无线网络中易于被捕获。尽管这些错误通过基本的努力很容易被发现，但它可以带来更大的威胁，即当理解了应用程序的业务逻辑，以确保开发者没有错误地在应用程序中放置一个可以绕过认证控制的入口的时候。
- 弱任务管理（Weak session management）：由于HTTP协议是一个无状态协议，因此状态管理的一些形式应有必要能够区分并发用户，并且对每个及所有的页不需要输入认证信任。无论唯一任务标记是通过cookie进行传递的，还是在URI内部或者页数据中传递的，标记本身都不再重要，重要的是它们是如何被结构化的和被保护的。cookie应当是足够随机的，以使强制性漏洞攻击变得不可行。另外，它们必须要经常过期以提防重复攻击。
- 缓冲区溢出（Buffer overflow）：缓冲区溢出攻击在Web应用中的普遍性比不上它在桌面和服务器应用中的普遍性。这个原因主要归结于Web应用开发所通常使用的语言如C#或者Java，这些语言所具有的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，在对Web应用进行模糊测试时就可以忽略缓冲区溢出。很可能一个应用程序将用户提供的输入传递给用C或C++编写的一个独立的应用，而该应用易于受到缓冲区溢出的影响。除此之外要记住，当进行模糊测试时你至少有两个目标：Web应用程序和Web服务器。另一方面，Web服务器通常用易发生缓冲区溢出问题的语言编写。
- 不正确支持的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，还有许多其他的和RFC兼容的以及第三方的方法。如果这些方法没有被正确地实现，那么它们就可以允许攻击者操纵服务器上的数据，或者通过后续攻击获取有价值的信息。因此，模糊测试应当被用来识别所有支持的方法，以确定它们是否被正确实现。
- 远程命令执行（Remote command exection）：Web服务器可能简单地将用户提供的输入传递给其他的应用或者操作系统本身。如果这些输入没有经过适当的验证，那么就会使攻击者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样的攻击。
- 远程代码注入（Remote code injection）：同样，这个漏洞也易于攻击PHP应用。不良的编码习惯，包括允许未验证的用户输入传递给如`include()`或`require()`的一个方法，这将允许本地或者远程的PHP代码被包含进来。当用户输入被盲目地传递给这些方法时，它

就允许攻击者将他们自己的PHP代码注入到目标应用程序中。据Mitre 2006年的统计，此类漏洞占新发现漏洞的9.5%。

- 漏洞库（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。一个Web应用中所包含的任何代码，无论是自己编写的，还是从第三方经过预编译获取的，都可能是存在潜在漏洞的代码，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以确保所包含的库没有已知的漏洞。除此之外，应当对第三方库执行与自编库相同水平的模糊测试以及其他的安全性测试
- HTTP响应分解（HTTP response splitting）：HTTP响应分解第一次被广泛的认知，是随着Sanctum Inc.的白皮书《Divide and Conquer》《分解和征服》的发布[⊖]。当一个用户可以向响应的头信息中注入一个CRLF序列时，该攻击就变得可行。接着，它会使攻击者能够篡改Web服务器所提供的响应，并导致一系列不同的攻击，包括使Web代理和浏览器cache遭到破坏。
- 跨站点请求伪造（Cross Site Requset Forgery,CSRF）：CSRF攻击很难防护，并且是一种不断增长的威胁。当一个被攻击者有一个活动任务时，如果一个攻击者能够使被攻击者去执行某些动作，如点击一个可能存在于E-mail消息中的链接，那么一个CSRF漏洞就会存在。例如，在一个银行的网站上，一个资金转账请求可能会通过提交一个Web表单来完成，而该表单中包含有账户信息和转账金额。只有该账户的所有者才能实现转账，并且只有该账户的所有者可以使用他或她的信用信息进行注册。然而，如果此人已经注册，并且不知不觉的点击了一个链接使得相同的Web表单数据发送给银行的站点，那么在被攻击者无察觉的情况下就会发生相同的转账操作。为了防护CSRF攻击，Web站点通常要求用户在执行一项敏感操作之前，如资金转账，进行重复确认或执行某些手工操作。Web站点也开始在Web表单中实现当前（一次）值，以验证表单的来源。

尽管这里并没有完整的列出所有可能的Web应用漏洞，但它说明了Web应用程序易于受到许多漏洞的攻击。同时它也说明了尽管影响本地应用的某些漏洞同样也会影响Web应用，但有许多攻击是专门针对Web应用的。为了通过模糊测试来发现Web应用的漏洞，必须要通过HTTP协议来提交输入，并且必须要使用与对本地应用进行模糊测试不同的机制来发现错误。要了解完整的Web应用漏洞的情况，我们推荐参阅Web应用安全协会（Web Application Security Consortium's, WASC）的威胁分类（Threat, TC）项目[⊜]。

9.5 异常检测

当对Web应用进行模糊测试时，异常检测是该类模糊测试中非常具有挑战性的一项工作。如果我们让一个模糊器连夜运行，并且发现应用程序因为它所接收到的多于10 000个请求中的一个而崩溃，那么这样的结果对我们没有什么意义。我们可能会知道存在着一个漏洞条件，但

⊖ http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

⊜ http://www.webappsec.org/projects/threat/

是我们无法将它重现。因此，下面列出的数据是很重要的，可以用来识别潜在的漏洞条件。

- HTTP状态码。当一个Web服务器响应一个请求时，它包含一个三位数字码以确定请求的状态。状态码的完整列表可以在RFC2616－超文本传输协议－HTTP/1.1[⊖]的第10部分找到。这些状态码可以提供线索以确定哪些模糊请求需要进一步加以研究。例如，Internet服务器错误的状态码500可能会提示前面的模糊请求导致了服务器的错误。同样，未授权错误的状态码401则提示一个被请求的页是存在的，但是受密码保护。
- Web服务器错误消息。Web应用可能被设计成直接在Web页的内容中显示一个错误消息。使用正则表达式来解析包含在响应中的HTML可以揭露这些消息。
- 中断连接。如果其中一个模糊输入导致Web服务器冻结或崩溃，那么后续的请求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以记录毁坏通道和失效连接何时发生。当检查日志文件并识别出一个故障连接的字符串，那么就立即查看日志条目之前的请求以确定故障。
- 日志文件。大多数Web服务器都可以被配置以记录不同类型的错误。同样，这些日志可以提供线索以发现哪些模糊请求导致问题的发生，而该问题可导致一个漏洞条件的产生。日志文件所面临的挑战是它们不能被直接绑定到导致错误的请求中。将请求和错误日志相连接的一个可选方法是将攻击计算机和目标计算机的时钟进行同步，并查看请求和日志条目的时间戳，而这将会限制可能的错误数量。它不能准确地、确定地将一个模糊请求和它所导致的错误绑定在一起。
- 事件日志。事件日志类似于Web服务器日志，因为它们也不能直接绑定到请求，但是可以通过时间戳关联到一个特定的范围。事件日志被微软的Windows操作系统所使用，并且可以使用事件查看应用程序来访问。
- 调试器。鉴别已处理和未处理异常的最好方法，是在模糊测试之前将目标应用连接到一个调试器。错误处理机制将阻止由模糊测试所导致的许多错误的明显标记，但是这些错误通常可以通过使用一个调试器来发现。寻找已处理的异常同未处理的异常同等重要，因为某些特定类型的错误如空指针解除引用以及格式化字符串虽然会导致已处理的异常，但通过给定适合的输入值，它们也可以发现漏洞。这里所面临的最大挑战就是确定哪个请求导致了异常的产生。调试器在对Web应用进行模糊测试方面具有一定的局限性，因为虽然它们可以发现服务器异常，但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，并且学习了如何使用它来发现Web应用中的漏洞。这要求全面地了解Web浏览器是如何提交请求的，以及Web服务器是如何对请求作出响应的。通过这些知识，我们就可以确定所有的能够被终端用户控制和操纵的那些巧妙的输入。可以通过这些输入来进行模糊测试，并通过监视潜在的异常以发现漏洞。在下一章中，我们将创建一个Web应用模糊器以实现测试过程的自动化。

⊖ http://rfc.net/rfc2616.html

第10章 Web应用程序和服务器的模糊测试：自动化

“对我们来说最重要的事情是发现奥萨姆·本·拉登，这是我们的头等大事，在找到他之前我们不会停歇。”

——George W. Bush, Washington DC, 2001年9月13日

“我不知道本·拉登在哪里。我没有任何想法并且实际上也不关心。它并不是那么重要。它并不是我们的头等大事。”

——George W. Bush, Washington DC, 2002年3月13日

前面我们已经讨论了如何对Web应用程序进行模糊测试，现在可以讨论有关模糊测试的理论了。在本章中，我们将利用前面章节所学的知识，通过开发一个图形化的Web应用模糊器WebFuzz来应用这些知识。我们首先介绍该应用程序的设计，并确定我们所面对的独特的挑战。然后，我们就可以选择一种合适的开发平台开始构建模糊器。一旦开发完成，我们的工作并没有结束。当构建一个漏洞发掘工具时，你永远不能认为工作已经完成直到你已经使用它发现了漏洞。你不会生产一辆汽车而不对它进行驾驶测试。因此，我们将遍历一些不同类型的已知的Web应用漏洞，以确定WebFuzz是否能够发现这些漏洞。

10.1 Web应用模糊器

Web应用模糊测试并不是一个新的概念。目前存在有许多不同的模糊器并且数量还在增加。下面列出了一些主流的、免费的、以及商用的Web应用模糊器。

- SPIKE代理[⊖]。SPIKE代理是由Dave Aitel用Python语言开发的、基于浏览器的Web模糊器。它执行一个代理动作，捕获Web浏览器请求，然后允许你为一个目标Web站点运行一系列预定义的审核，以识别不同的漏洞如SQL注入、缓冲区溢出和XSS。由于建立在一个开源的架构之上，SPIKE代理可以被扩展以对不同的目标应用进行模糊测试。SPIKE代理并不是一个专门的模糊器；实际上它是漏洞扫描器和模糊器的组合体。图10-1显示了SPIKE代理的屏幕快照。
- WebScarab[⊜]。开放Web应用安全项目（Open Web Application Security Project,OWASP）为测试Web应用的安全性提供了许多不同的可用工具，这其中就包括WebScarab。尽管它是一个全面的Web应用安全测试工具，但它也包含一个基本的模糊器，可以向应用程序的参数注入模糊值。
- SPI模糊器[⊝]。SPI模糊器是SPI工具集的一个构件，而SPI工具集是WebInspect应用的一部

⊖ http://www.immunitysec.com/resources-freesoftware.shtml

⊜ http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project

⊝ http://www.spidynamics.com/products/webinspect/toolkit.html

分。WebInspect是SPI Dynamics公司开发的一个商用工具，被设计用来为Web应用程序的测试提供一套完整的工具。

- Codenomicon HTTP测试工具[⊖]。Codenomicon为包括HTTP在内的各种协议产生模糊测试集。
- beSTORM[⊜]。类似于Codenomicon，beSTORM也是一个可以处理包括HTTP在内的不同Internet协议的模糊器。

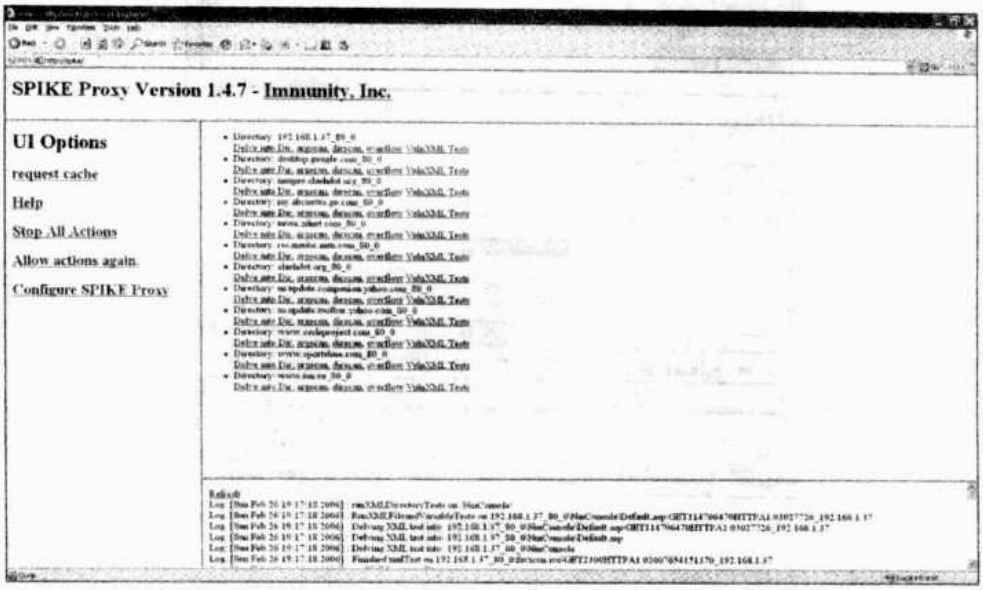


图10-1 SPIKE代理

WebFuzz的开发受到了商用工具SPI模糊器的启发。SPI模糊器是一个简单但设计很精巧的图形化Web应用模糊器，它向用户提供了对模糊测试所使用的原始HTTP请求的完整的控制。它要求用户具备基本的HTTP协议知识以开发测试并生成结果。同样，也需要使用这些知识来解释响应以识别那些需要进一步研究的问题。图10-2展示了SPI模糊器。

SPI模糊器的主要缺点是它只能作为一个昂贵的商用应用程序中的一个构件来使用。我们基于SPI模糊器的基本知识，来创建一个功能受到一些限制但是开源的工具来满足WebFuzz的特定需求。和大多数模糊器一样，WebFuzz不是一个易于使用以及完全用于安全性的一个工具。它只是一个将以前手工完成的工作进行自动化处理的工具。你作为终端用户可以利用该工具来开发有效的测试并分析测试结果。该工具应当被视为一个研究的起点，而不是最终的解决方案。

像本书中所开发的所有工具一样，WebFuzz是一个开源的应用程序。因此，它提供了一个

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

⊜ http://www.beyondsecurity.com/BeStorm_Info.htm

可以而且应当被进一步创建的架构。我们鼓励你对其添加新的功能并修复错误，但最重要的是，我们希望你能和其他人来共享你所做的改进工作。本章的剩余部分详细描述了WebFuzz的开发过程，并通过研究不同的实例来说明其所具有的功能以及受到的限制。可以从本书的Web站点www.fuzzing.org处下载WebFuzz。

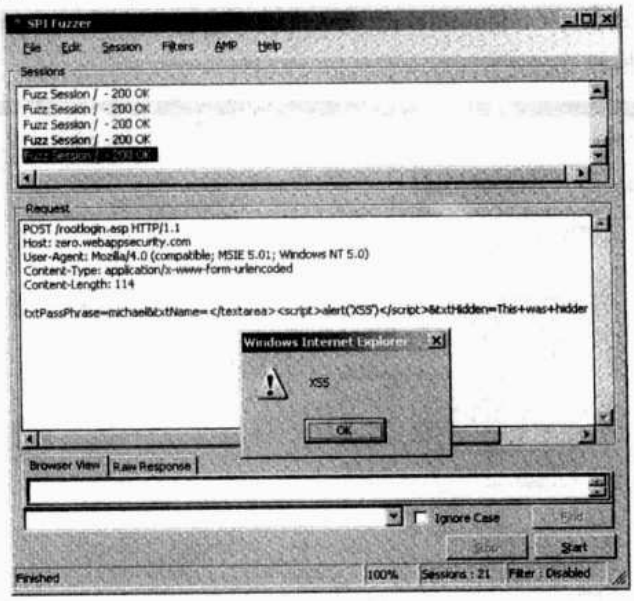


图10-2　SPI模糊器

10.2　WebFuzz的特性

在开始创建我们自己的模糊器之前，首先回顾一下上一章所学到的HTTP协议工作机理的有关知识，并利用这些知识来确定我们的模糊器所需要的特性。

10.2.1　请求

让我们从头开始讨论。你无法对一个Web应用进行模糊测试，除非你有一种方式来向它发送请求。在通常情况下，我们使用一个Web浏览器同Web服务器进行通信。毕竟它知道如何利用HTTP进行通信，并且能够将所有杂乱的细节信息组装成一个HTTP请求。然而，当进行模糊测试时，我们需要这些细节信息。我们希望能发挥主观能动性，来改变请求的所有方面，因此，我们选择了将原始请求暴露给终端用户，并允许对其中的任何部分进行模糊化。

图10-3显示了一个基本的WebFuzz请求。

该请求包括以下字段：

- 主机。目标机器的名字和IP地址是所需要的一个字段。我们不能对一个Web应用进行模糊测试，而不让WebFuzz知道向何处发送请求。这不是一个可以进行模糊测试的字段。
- 端口。尽管Web应用默认情况下运行在TCP端口80，然而它们也可以方便的运行在其他任

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书籍杂志全免 KEY 提供,，完全无加密。

意的TCP端口。实际上，通常将Web应用设计为提供一个基于Web的管理控制台，以使其运行在一个可选的端口而不干扰主要的Web服务器。同主机名一样，端口字段也是告诉WebFuzz向何处发送请求，同时，它也不是一个可以进行模糊测试的字段。

- *超时*。由于我们的目的是发送非标准的Web请求，因此，目标应用程序通常不会以一种及时的方式来作出响应。因此，我们包含了一个用户定义的、以毫秒计的超时值。当记录一个请求的响应超时时，这种记录是非常重要的，因为它可以指明我们的请求使得目标程序离线，从而导致一个潜在的DoS漏洞。
- *请求头*。这是操作执行的起点。当使用一个Web浏览器时，终端用户可以控制目标主机、端口以及请求URI，但却不能控制所有不同的头信息。我们有意将请求的所有组成部分保持在一个单独可写的文本字段中，因为我们希望终端用户能够控制请求的各个方面。可以通过简单地将所需要的请求敲进请求头字段中来手工创建一个请求。另外，如果你倾向于使用点击的方法，那么也可以使用图10-3中上下文菜单中所提供的标准头列表来拼凑一个头。最后一个选项是如果默认Web页需要一个基本请求，那么允许从上下文菜单中选择默认头。

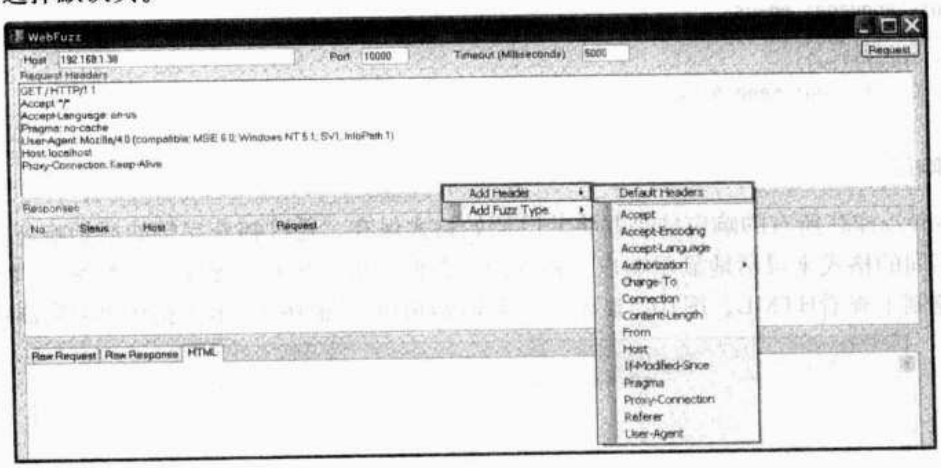


图10-3 WebFuzz请求

10.2.2 模糊变量

模糊变量是指在请求中将要被模糊数据所替代的那些区域。正如所讨论的，用户对发送给Web服务器的原始请求有着完整的控制权。因此，模糊变量被直接添加到原始请求中，并通过在方括号中的变量名（例如`[Overflow]`）来识别。当设计创建模糊变量的函数时，我们可以将它们划分为两个基本类型：静态列表或生成变量。静态列表是指从一个预定义的数据列表中取出的模糊变量。静态列表的一个示例是用来识别XSS漏洞的模糊数据。可能导致XSS漏洞的不同输入的一个预定义列表（例如`<script>alert ('XSS') </script>`）被编译，并且每次将数据注入到请求的一行中。静态列表通常被作为外部ASCII文本文件来维护，因此用户不需要再

编译应用程序就可以修改变量。另一方面，生成模糊变量是根据预定义的算法创建的，而该算法可能允许接收用户的输入。溢出变量是生成模糊变量的一个例子。一个溢出允许用户定义溢出所使用的文本，并定义文本的长度和其重复次数。图10-4说明了允许对溢出变量的输入进行更改的默认弹出窗口。

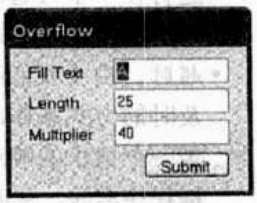


图10-4　溢出模糊变量

为了更加有效，我们可能希望在一个单一的请求中定义多个模糊变量。尽管你可能经常想要在同一时间动态的改变两个或多个变量（例如变量的长度值以及它所描述的内容），出于简化的原因，我们选择了一次只处理一个变量，但在一个单一请求中包含多个模糊变量是可能的。WebFuzz将处理所遇到的第一个模糊变量，而忽略其他的变量。一旦第一个变量被完全模糊化，那么它将从原始请求中被删除，因此WebFuzz就可以转向处理后续变量。下面的WebFuzz请求说明了一个请求，该请求被设计为通过一次攻击而识别多个公共漏洞。

```
[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捕获所有的响应结果并将其以原始格式保存。通过捕获完整的原始响应，我们就可以用不同的格式来灵活地显示响应。特别的，我们能够让用户来显示原始结果或者在Web浏览器的控制下查看HTML。图10-5显示了一个原始响应，图10-6显示了在Web浏览器中的相同



图10-5　原始响应

数据。这是非常重要的，因为现存漏洞的线索具有不同的形式。例如，头信息可能包含一个状态码（例如500－内部错误）以提示发生了一个DoS。同样，Web页本身可能也会向用户显示错误消息以提示可能发生了SQL注入。在这种情况下，当一个浏览器解释HTML时，错误将易于查看。



图10-6 HTML格式化的响应

10.3 必要的背景知识

HTTP在两个方面提出了独特的挑战，一是确定如何对通信进行最佳的监视，更重要的是确定当异常发生时如何来识别这些异常。接下来，我们将研究Web浏览器内部所执行的一些操作情况。

10.3.1 识别请求

WebFuzz要求用户创建一个原始的HTTP请求，但是对于一个给定的Web应用，如何来确定合适的请求呢？哪些Web页是存在的？这些Web页可以接收哪些变量？这些变量应当如何传递？在上一章中，我们讨论了如何手工的识别输入，以及如何通过使用嗅探器、Web蜘蛛和代理来识别输入。在继续讨论之前，需要介绍一个Web浏览器插件，该插件在使用WebFuzz时以及需要为单一Web页确定原始输入时都能够应用。LiveHTTPHeaders项目[⊖]为确定基于Mozilla的Web浏览器中的原始HTTP请求提供了一个方便的工具。图10-7展示了LiveHTTPHeaders用来显示Firefox sidebar中的所有请求及其相应的响应。使用这种方法的一个突出优点是，一个请求可以被捕获并直接剪切后粘贴到WebFuzz以简化模糊请求的创建。还有一些其他不同的浏览器

⊖ http://livehttpheaders.mozdev.org/

插件，例如作为Firefox的扩展的Tamper Data[⊖]和Firebug[⊜]，以及针对Internet Explorer的浏览器插件Fiddler[⊝]，但LiveHTTPHeaders因其简便性而成为首选。

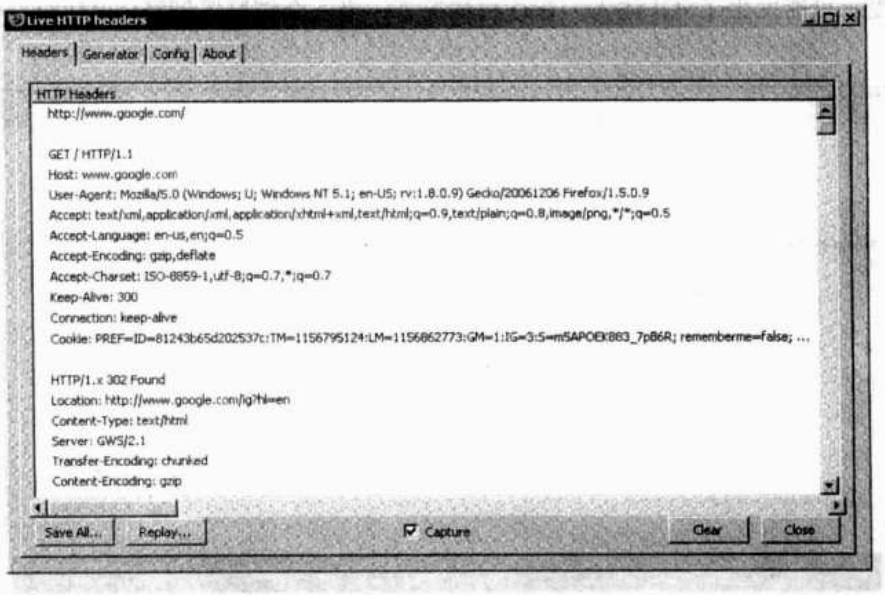


图10-7　LiveHTTPHeaders

10.3.2　漏洞检测

正如前面所讨论的，从目标应用程序所返回的响应可以提供不同的线索以指示最初模糊请求所带来的影响。WebFuzz被设计为向用户提供检测数据，但是需要用户来对响应进行解析。尽管手工评审来自于Web服务器的所有响应是不切实际的，但希望响应的特定部分能够提示出一个异常条件的发生。然后用户就可以识别相关的请求。这是可能的，因为所有的响应，不论是一个原始响应还是HTML格式的响应，都可以使用相关的请求通过从响应窗口选择合适的选项来查看。

当运行WebFuzz时，下面所列出的信息可能会提示一个漏洞条件的存在：

- HTML状态码
- 嵌入在响应中的错误消息
- 嵌入在响应中的用户输入
- 性能下降
- 请求超时
- WebFuzz错误消息

⊖ http://addons.mozilla.org/firefox/966/

⊜ http://www.getfirebug.com/

⊝ http://www.fiddletool.com

• 已处理和未处理的异常

下面分别来研究一下每个信息，以便更好的理解它们为什么能够有助于识别漏洞。

HTML状态码

我们已经提到HTML状态码是一个非常重要的信息，因为它们为初始请求的成功或失败提供了一种快捷的可视化的指示。因此，WebFuzz解析原始响应以识别状态码，然后该状态码单独显示在一个表中以详细描述所有的响应。使用此信息，用户可以迅速的识别出应当被进一步详细研究的那些响应。

嵌入在响应中的错误消息

从设计来讲，Web服务器在动态生成的Web页中可以包含错误消息。当一个Web服务器被不正确的部署到一个可以进行调试的环境时，情况尤其如此。多次提到的一个典型的错误消息是一个类似于显示“密码错误”验证错误，而不是“用户名或密码错误”。当对一个Web应用的注册界面进行强制性漏洞发掘时，第一个消息将使你知道用户名存在但密码是错误的。这将未知变量的数量从两个（用户名和密码）减少为一个（密码），并且极大的增加了获取访问的几率。当识别Web注入攻击时，应用程序错误消息也可以非常有用。

嵌入在响应中的用户输入

当动态生成的Web页包含用户提供的数据时，那么就可能后存在XSS漏洞。Web应用的设计者需要过滤用户输入以确保不会发生这样的攻击，但是不正确的过滤也是一个很普遍的问题。因此，识别WebFuzz所提供的HTML响应中的数据就表示应用程序应当被测试以检测XSS漏洞。

性能下降

尽管一个最高级别的应用程序崩溃使得一个DoS攻击易于发现，但是这类漏洞通常会更加精巧。在通常情况下，性能下降将会意味着应用程序易于受到DoS攻击。一个请求超时是识别性能下降的一种方法，但在进行模糊测试时，性能监视器也应当被使用以识别诸如CPU过载或内存使用等问题。

请求超时

正如所提到的，请求超时不应当被忽略，因为它们可能会指示一个暂时或者永久的DoS条件。

WebFuzz错误消息

WebFuzz有它自己的错误处理方法，当在特定函数执行失败时将弹出错误消息。例如，如果目标服务器由于之前的一个模糊请求而离线，则WebFuzz可能会发出一个错误消息以表明它不能连接到目标服务器。这也意味着发生了一个DoS攻击。

已处理和未处理的异常

当对Web应用进行模糊测试时，既可以发现应用程序本身的漏洞，也可以发现应用程序所运行的Web服务器的漏洞。因此，监视服务器的状态也是非常重要的。尽管Web服务器所返回的响应提供了发现潜在漏洞的机会，但是它们并没有给出完整的解决方法。模糊请求很可能会导致已处理和未处理的异常，而当输入被微小改变之后，这些异常有可能导致漏洞条件。因此，

这里推荐在模糊测试过程中为目标服务器设置一个单独的调试器，以识别这些异常。本书所讨论的其他工具如FileFuzz和COMRaider，都包含有内嵌的调试功能。然而，对于Web应用的模糊测试而言，这并不是需要的。一个WebFuzz不需要重复部署并破坏一个应用程序。这次，我们向一个单独的Web应用发送一系列模糊请求，该应用连续运行并响应所有的请求，除非遇到导致DoS条件的输入。

10.4 WebFuzz的开发

到目前为止我们已经具备了足够的理论知识，下面可以进行具体的模糊器的开发了。我们将详细介绍WebFuzz的开发过程。

10.4.1 开发方法

在进行WebFuzz的设计时，我们的目标是创建一个对Web应用进行模糊测试的用户友好的工具。由于假定我们的目标用户具有一定的HTTP知识，因此没有必要创建一个只是简单进行点击操作的工具。相反，我们期望设计一种能为终端用户提供对请求结构具有最大操纵灵活性的工具。我们也希望以一种方式来传送响应数据，而该方式可以简化潜在漏洞的发现。

10.4.2 开发语言的选择

为了使WebFuzz具有更好的用户友好性，那么最好是创建一个GUI应用程序。基于两个主要的原因选择了C#作为开发语言。首先，C#允许我们非常方便地设计一个看起来相当专业的GUI应用程序。其次，C#提供了一些类以帮助实现发送/接收的网络通信功能。使用C#的一个缺点是使我们主要依赖于Windows平台。然而，当对Web应用进行模糊测试时，我们没有必要对运行在与模糊器相同机器上的目标应用进行模糊测试。因此，设计一个基于Windows的工具并没有限制我们只能对基于Windows的目标应用进行模糊测试。

10.4.3 设计

如果将所有的代码都拿出来进行讲述，那将会使读者疲倦不堪。然而，WebFuzz模糊器的主要功能包含在一些基本的类中，我们将对每个类中的关键功能做重点说明。请记住，本书开发的所有应用程序的完整源代码在本书的网站上都可以得到。

TcpClient类

C#提供了一个WebClient类，该类中包含了处理HTTP请求和响应的函数。它将生成和处理必要的网络通信所需要的大部分代码进行了封装，能够极大的简化应用程序的开发。它甚至包含函数以处理WebFuzz所需要的许多功能，如访问HTTP响应的头。在稍微低一些的层次上，C#提供了`HttpWebRequest`和`HttpWebResponse`类。这些类需要进行稍多的编码工作，但同时也可使用更高级的功能如使用代理的能力。WebFuzz应当使用这些类中的哪些类呢？答案是都不使用。取而代之，我们选择使用为任意类型的TCP通信（不只是HTTP）而设计的`TcpClient`类。这样，它就缺少了其他Web类所封装的一些功能。为什么要这样做呢？是我们乐于编写没用的

代码吗？不是，只是必须要这样做。

当编写模糊器时，一个最大的挑战就是你试图以一种超常规的方法来编写代码。因此，那些标准的类和函数可能就不适合你的需要。我们的目的是要完整的控制原始HTTP请求，但不幸的是各种各样的Web类并没有为我们提供这样的控制粒度。例如，考虑下面的代码：

```
WebClient wclFuzz = new WebClient();
wclFuzz.Headers.Add("blah", "blah");

Stream data = wclFuzz.OpenRead("http:// www.fuzzing.org");
StreamReader reader = new StreamReader(data);

data.Close();
reader.Close();
```

这个简单的代码示例是要求使用WebClient类发送一个定制的Web请求。我们创建了一个基本的GET请求，并且只加入了一个定制头（blah:blah）。然而，当嗅探所生成的实际通信时，我们发现下面的请求被发送了：

```
GET / HTTP/1.1
blah: blah
Host: www.fuzzing.org
Connection: Keep-Alive
```

你可能会注意到在实际的请求中，两个额外的头也被加入了，即主机（Host）和连接（Connection）。因为这个原因，使得我们不能使用通常的类。我们必须要牺牲一定的使用便利性，以换取对该过程进行低层次完整的控制。在我们的例子中，WebFuzz的网络通信部分使用了TcpClient类。

异步Sockets

网络通信可以使用同步或者异步socket来实现。尽管使用异步socket需要多做一些额外工作，但在WebFuzz中使用它们是经过深思熟虑后作出的决定，因为这种方式更好地处理了在使用模糊器时可能会出现的预期的网络问题。

同步socket采用阻止化的工作方式。这意味着当遇到一个请求或响应时，主线程将停止执行，并且在继续执行之前要等待该通信执行完毕。使用一个模糊器，我们将尽力地试图去导致异常条件的发生，某些异常可能会导致性能下降或者使目标应用完全离线。我们不希望WebFuzz在等待可能永远也不会发生的通信时成为无响应的程序。异步socket可以使我们避免此问题，因为它们采用的是非阻止化的工作方式。异步socket发出一个单独的线程来处理此类通信，并且激活一个回调函数以指示通信何时完成。这就允许其他事件继续执行。

下面来分析一下WebFuzz中的网络通信代码，以便更好地理解异步socket的概念：

```
TcpClient client;
NetworkStream stream;
ClientState cs;

try
{
```

```
        client = new TcpClient();
        client.Connect(reqHost, Convert.ToInt32(tbxPort.Text));
        stream = client.GetStream();
        cs = new ClientState(stream, reqBytes);
    }
    catch (SocketException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
          MessageBoxIcon.Error);
        return;
    }
    catch (System.IO.IOException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
          MessageBoxIcon.Error);
        return;
    }
    IAsyncResult result = stream.BeginWrite(cs.ByteBuffer, 0,
        cs.ByteBuffer.Length, new AsyncCallback(OnWriteComplete), cs);

    result.AsyncWaitHandle.WaitOne();
```

在创建了一个典型的TCPClient和NetworkStream之后，我们激活了stream中的BeginWrite()方法。BeginWrite()方法使用下面5个参数[⊖]：

- byte[] array。包含写入到网络流的数据的一个缓冲区。
- int offset。缓冲区中开始发送数据的位置。
- int numBytes。写入的最大字节数。
- AsyncCallback userCallback。通信完成时将被激活的回调函数。
- object stateObject。区分此异步写请求与其他请求的一个对象。

AsyncWaitHandle.WaitOne()将导致侦听线程被阻止，直到请求被成功地发送。这时，回调函数将被调用，如下所示：

```
public static void OnWriteComplete(IAsyncResult ar)
{
    try
    {
        ClientState cs = (ClientState)ar.AsyncState;
        cs.NetStream.EndWrite(ar);
    }
    catch (System.ObjectDisposedException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
          MessageBoxIcon.Error);
    }
}
```

当我们编写完对网络流的请求之后，就可以接收到从服务器返回的结果：

```
try
{
```

⊖ http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemiofilestreamclassbeginwritetopic.asp

```
        result = stream.BeginRead(cs.ByteBuffer, cs.TotalBytes,
        cs.ByteBuffer.Length - cs.TotalBytes,
        new AsyncCallback(OnReadComplete), cs);
    }
    catch (System.IO.IOException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
          MessageBoxIcon.Error);
        ReadDone.Close();
        return;
    }
```

在这时，我们再次使用一个异步socket，但这次它被用来接收从目标应用所返回的响应。我们现在激活BeginRead()方法，该方法含有同BeginWrite()方法相同的参数，但这次我们使用OnReadComplete()作为回调函数：

```
public void OnReadComplete(IAsyncResult ar)
{
    readTimeout.Elapsed += new ElapsedEventHandler(OnTimedEvent);
    readTimeout.Interval = Convert.ToInt32(tbxTimeout.Text);
    readTimeout.Enabled = true;

    ClientState cs = (ClientState)ar.AsyncState;
    int bytesRcvd;

    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(bytesRcvd);

    if (bytesRcvd != 0)
    {
        cs.NetStream.BeginRead(cs.ByteBuffer, cs.TotalBytes,
        cs.ByteBuffer.Length - cs.TotalBytes,
              new AsyncCallback(OnReadComplete), cs);
    }
    else
    {
```

```
            readTimeout.Enabled = false;
            if (ReadDone.Set() == false)
               ReadDone.Set();
         }
      }
```

在OnReadComplete()的开头创建了一个定时器（readTimeout），当到达用户定义的超时时，该定时器将调用ReadDone.Set()。这就允许我们确保如果读取操作失败则该线程可能处于不活动状态，并且为终端用户提供一种方法以控制超时的长度。然后将所接收到的响应追加到缓冲区。这时，我们需要决定是否应当继续等待以后的数据。这项工作可以通过确定是否有字节被接收来完成。如果有字节被接收，我们通过再次激活BeginRead()来继续等待以后的数据。如果没有字节被接收，则销毁该线程并继续执行。

生成请求

在我们发送一个请求之前，必须要首先确定发送的内容。此内容显然是从用户创建请求的请求头窗口获得的，但是每个模糊变量[xxx]必须被实际的模糊数据所替代。一旦用户在btnRequest_Click()方法中点击请求按钮，那么该过程就开始执行。

```
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);
```

当我们生成请求以后，就开始执行一个循环，该循环连续地解析用户提供的数据直到在请求中遇到模糊变量。然后转到一个case语句，以确定对每个模糊变量执行何种操作。

```
int arrayCount = 0;
int arrayEnd = 0;
Read fuzzText = null;
WebFuzz.Generate fuzzGenerate = null;
ArrayList fuzzArray = 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 "Traversal":
        fuzzGenerate= new WebFuzz.Overflow("../", 1, 10);
        fuzzArray = fuzzGenerate.buildArray();
        arrayEnd = fuzzArray.Count;
        replaceString = "[Traversal]";
        break;
    case "Format":
        fuzzGenerate= new WebFuzz.Overflow("%n", 1, 10);
        fuzzArray = fuzzGenerate.buildArray();
        arrayEnd = fuzzArray.Count;
        replaceString = "[Format]";
        break;
    case "None":
        ArrayList nullValueArrayList = new ArrayList();
        nullValueArrayList.Add("");
        fuzzArray = nullValueArrayList;
        arrayEnd = fuzzArray.Count;
        replaceString = "[None]";
        break;
    default:
        arrayEnd = 1;
        break;
```

来自于静态列表（SQL，XSS和方法）的那些模糊变量，创建了一个新的Read()类，并且将包含模糊变量的ASCII文本文件的名字传递给构造器。另一方面，生成的变量（溢出，遍历和格式化）实例化一个新的Generate()类，并传递将要被重复的字符串、该字符串的总长度以及该字符串将要被传递的次数。

接收响应

当响应被接收时，WebFuzz通过将请求、原始响应、HTML响应、主机名和路径添加到单独的字符串数组来将它们进行保存。另外，识别的信息包括状态码、主机名以及被添加到一个ListView控制的请求。通过这种方法，当模糊测试执行完毕后，你可以简单地点击ListView控件中合适的响应，则详细信息将被显示在一系列被标签化的超文本框以及Web浏览器控件中。

```
rtbRequestRaw.Text = reqString;
rtbResponseRaw.Text = dataReceived;
```

```
wbrResponse.DocumentText = html;

string path = getPath(reqString);

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;
responsesRaw[lvwResponses.Items.Count - 1] = dataReceived;
responsesHtml[lvwResponses.Items.Count - 1] = html;
responsesHost[lvwResponses.Items.Count - 1] = reqHost;
responsesPath[lvwResponses.Items.Count - 1] = path;
```

正如所提到的，WebFuzz并不是一个只能执行简单点击操作的漏洞扫描器。它被设计为允许高级用户来对一个HTTP请求的目标部分进行模糊化。该工具的完整源代码和二进制代码可在网站www.fuzzing.org找到。

10.5 实例研究

现在我们已经基本了解了WebFuzz的创建原因和构造方法，下面将讨论一些更加重要的内容，来看看它在实际应用中是否能够有效地执行。

10.5.1 目录遍历

当一个用户可以进入Web根目录，并且可以访问并未想通过Web应用传递的文件和文件夹时，就会存在目录遍历。这种类型的漏洞造成了一种泄密风险，但也可能升级而导致威胁整个系统的安全，而这取决于可以被访问的文件。例如，考虑这种情况，即目录遍历将允许一个攻击者获取密码文件。即使该文件是加密的，那么当它被检索之后仍有机会获取其中的密码，因此攻击者稍后会再次登录，使用有效的认证信任连接到服务器。

遍历通常包括发送一系列的../字符以遍历更高层的目录。遍历字符通常是被编码的URL以绕过基本的检测过滤器。如果目录是可浏览的那么这就是所有需要做的工作，但是通常还需要追加一个存在文件的名字。当测试一个目录遍历攻击时，建议将服务器上默认存在的文件的名字进行追加。例如，在一个Windows系统中，boot.ini和win.ini就是很好的追加选择，因为它们是ASCII文件，同时当它们存在于所有现代的Windows操作系统时，非常容易被识别。

让我们来看在Trend Micro控制管理器[⊖]中的一个简单的目录遍历攻击的例子，该攻击是由rptserver.asp页中的IMAGE参数的不正确输入验证所引起的。我们将使用WebFuzz向rptserver.asp页发送一个GET请求，但是我们将用模糊变量[Traversal]后面跟着一个已知的

⊖ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=352

文件（在这里是win.ini）来替换正确的IMAGE参数。我们可以从图10-8所示的结果中看出，目录遍历实际上是在一个单一的遍历之后暴露的。

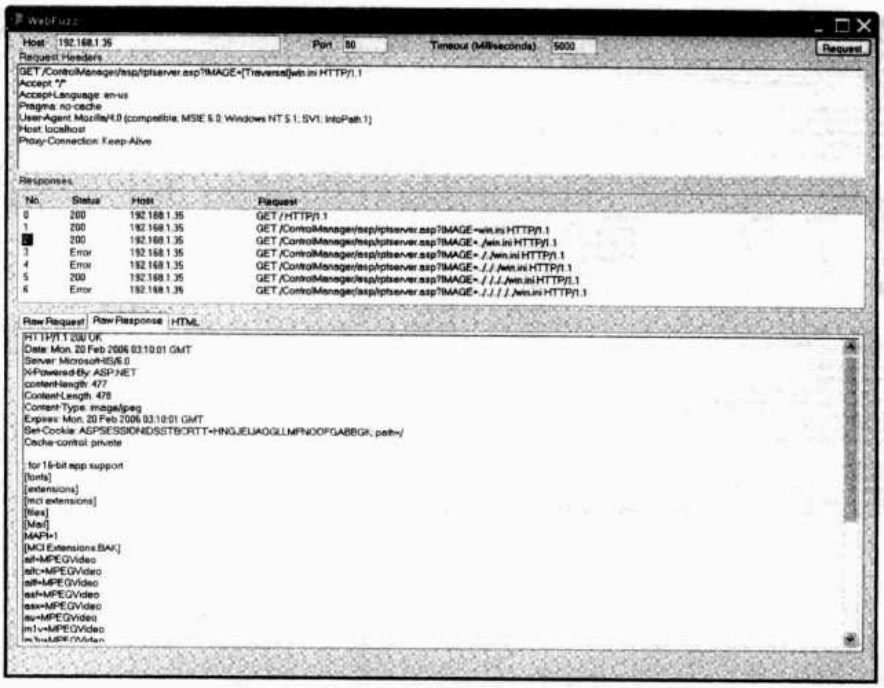


图10-8 Trend Micro控制管理器目录遍历

让我们来看另外一个例子，但这次我们将变换一种方式。Ipswitch Imail Web日历中的一个漏洞[⊖]提示我们为了发现一个目录遍历，有时必须要请求某些不存在的内容。这种情况下，我们发现当遍历被传递到一个不存在的JSP页时，目录遍历漏洞是存在的。我们再次使用WebFuzz来进行测试（如图10-9）。

为了测试该特殊的漏洞，我们将向服务器传递一个GET方法，其后面跟着遍历目录并以公共boot.ini文件作为结尾，以请求不存在的blah.asp Web页。图10-9显示它使用了一些请求，但在5个遍历序列之后，漏洞就被发现了。

10.5.2 溢出

尽管缓冲区溢出在Web应用中相对较少出现，但就像在控制台和GUI应用中一样，缓冲区溢出同样可以存在于Web应用和服务器中，并且存在的原因也是相同的：用户提供的输入没有被恰当的过滤，以致允许一个非预期长度的数据在一个固定大小的缓冲区中溢出。溢出具有很大的危险性，因为它可能导致任意的代码执行。然而，当进行模糊测试时，最可能指示发生一个溢出的情况是应用程序崩溃，并且跟着一个包含超长字符串的请求。通过在模糊测试的执行

⊖ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=242

过程中将一个调试器连接到目标应用，就有可能确定溢出是否只是一个DoS攻击，还是可以被扩展以允许远程代码执行。

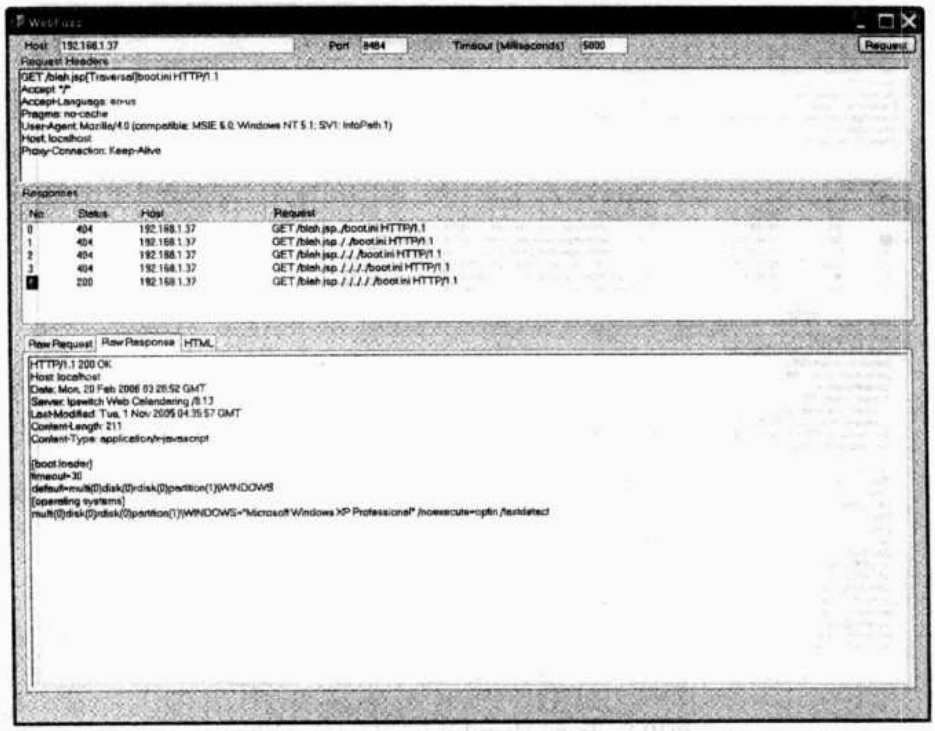


图10-9　Ipswitch Imail Web日历目录遍历

为了说明WebFuzz在检测溢出漏洞中的应用，我们将利用一个简单的例子。考虑PMSoftware的Simple Web Server中的一个溢出[⊖]。在这个例子中，我们所需要做的所有工作就是向服务器发送一个长的GET请求。我们将发送包含一个溢出模糊变量的请求如下：

最初，除了产生一个404-页没有找到的错误之外，没有发生任何其他事情。但是在图10-10中可以看到，在第10号请求之后，WebFuzz将不再接收任何响应，为什么呢？

当我们查看了Simple Web Server所运行的服务器之后，很快就得到了问题的答案，因为图10-11所示的弹出消息被显示。

这不是一个好的迹象（至少对Simple Web Server而言不是）。当该错误消息框被关闭后，应用程序也被关闭了。至少我们是受到了一个DoS攻击。那么是否还有机会使代码执行呢？从图10-12中可以得到结果，相关联的调试器显示EIP已经被控制，因此代码可能还会执行。如果你是第一次学习缓冲区溢出，那么不要太兴奋。这是最好的一种情形，实际中所遇到的情况并不总是这样乐观。

⊖ http://secunia.com/advisories/15000/

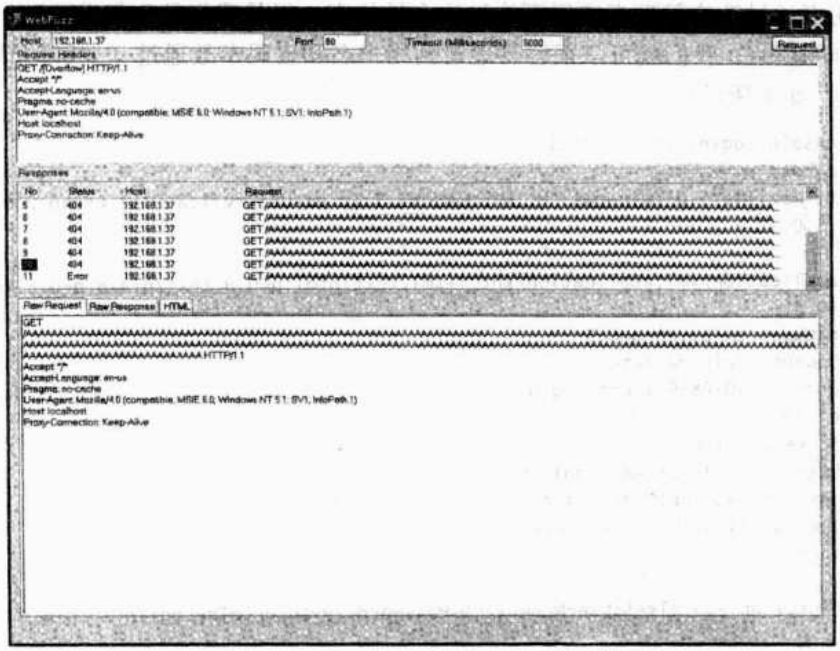


图10-10 Simple Web Server缓冲区溢出

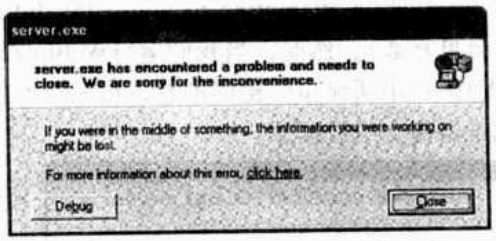


图10-11 Simple Web Server错误消息

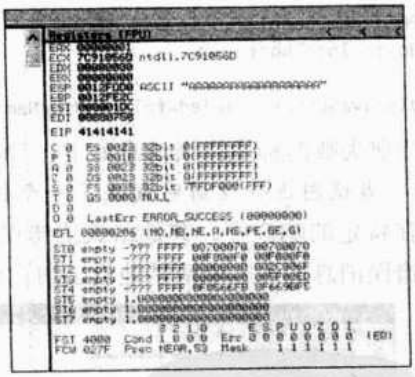


图10-12 Simple Web Server溢出

10.5.3 SQL注入

当用户提供的数据可以影响到达后端关系型数据库的SQL请求时，SQL注入攻击就会发生。再强调一次，不正确过滤的用户数据是该漏洞的罪魁祸首。当进行模糊测试时，应用程序的错误消息提供了强有力的线索，以指明可能发生了SQL注入。

Ipswitch Whatsup Professional（SPI）[⊖] 的注册界面中的用户名字段中的一个SQL注入攻

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

击，允许攻击者通过改变管理者密码而绕过安全性检查。但是我们是如何发现这一情况的？使用LiveHTTPHeaders，我们就可以很容易地看到用户名和密码参数通过一个POST请求被传递到login.asp页，如下所示：

```
POST /NmConsole/Login.asp HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.1) 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,image/
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

bIsJavaScriptDisabled=false&sUserName=xxx&sPassword=yyy&btnLogIn=Log+In
```

现在我们知道了该请求的正确格式，可以向下面这样加上一个模糊请求：

```
POST /NmConsole/Login.asp HTTP/1.1
Host: localhost

bIsJavaScriptDisabled=false&sUserName=[SQL]&sPassword=&btnLogIn=Log+In
```

注册失败的标准错误消息如下：There was an error while attempting to login: Invalid user name（当试图进行注册时发生了一个错误：非法的用户名）。然而，当我们运行WebFuzz时，发现在特定的响应中，其他格式的错误消息，如图10-13中所显示的错误消息，被显示。你可以从错误消息的内容中清楚地看出用户提供的数据准备提交给数据库。

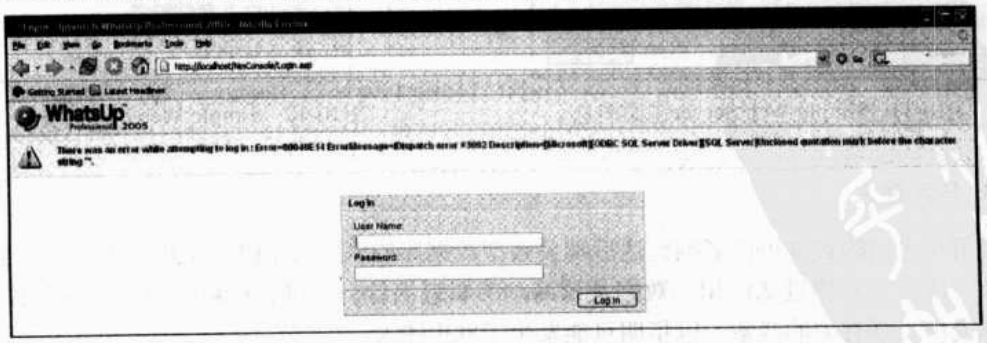


图10-13　Ipswitch Whatsup Professional（SPI）错误消息

然而，这并不是一个直接的SQL注入攻击。这里并没有使用普通的认证躲避技术如'or

1=1 ㊀，因此我们必须要实际设计一个合理的UPDATE查询来改变管理者的密码。这时，我们需要知道数据库模式的细节信息以构造特定于目标应用的被裁剪的SQL注入攻击，但是从哪里可以得到这些信息呢？通过快速的Google搜索可以找到答案。首先，Ipswitch本身就提供了可以下载的模式㊁。

这很好，但在创建模糊器的百忙之中，仍然需要做出一些努力来提出一个有意义的查询。不用担心：Ipswitch已经为我们提供了所需要的查询。Ipswitch知识库㊂实际上为我们提供了下面的命令，可以将管理者用户的密码重设为默认值：

```
osql -E -D WhatsU P-Q "UPDATE WebUser SET sPassword=DEFAULT WHERE sUserName='Admin'"
```

Ipswitch所提供的这个查询用于一个命令行工具，以帮助那些因为偶然忘记密码而不能进入应用程序的管理者。但是他们并没有认识到，该查询同样能够很好地用于SQL注入攻击。查看一下知识库的Q和A部分就可以看到如下内容：

问题/难题：我忘记了Web界面中默认管理者用户的密码。如何才能将它重设呢？

答案/解决方法：找到一个SQL注入漏洞，然后使用下面的命令...。

10.5.4 XSS脚本

XSS是无处不在的。按照Mitre在2006所做的统计，XSS占所有新漏洞的21.5%㊃，你可以很容易地发现这些漏洞。实际上，sla.ckers.org在它们的论坛上维护着一个不断增长的XSS漏洞列表，并且让人感到失望的是㊄，许多大公司也出现在列表当中。

同大多数Web应用漏洞一样，XSS也是由于不正确的输入验证所造成的。存在漏洞的应用程序接收用户输入，并且不经过任何过滤就将它们嵌入到动态页中。因此，一个攻击者就可以将客户端脚本如JavaScript注入到被请求的页中。然后，这将会允许攻击者控制所显示Web页的内容，或者代表被攻击者执行某些动作。

为了分析一个XSS模糊测试的例子，我们将从一个已知的存在漏洞的Web页开始。SPI Dynamics在http://zero.webappsecurity.com上部署了一个存在漏洞的Web应用，如图10-14所示，为了测试WebInspect，默认的注册页包含两个Web表单。一个是发送给login1.asp页的注册表单，另外一个是向rootlogin.asp页发送数据的表单。第二个表单包含一个XSS漏洞。在该表单的输入字段中，标签显示为Last Name的txtName字段的内容，将在被请求的rootlogin.asp页中返回。由于用户输入没有经过任何验证而被显示，因此应用程序将受到XSS的攻击。

测试存在XSS漏洞的一个普遍方法是输入一个简单的JavaScript代码片段和一个报警函数，该报警函数将导致产生一个弹出窗口。这是一个快速而简单的测试，它将生成一个简单的可视队列使得客户端JavaScript脚本可以被注入到目标页中。因此，我们将提交下面的请求以测试该

㊀ http://www.securiteam.com/securityreviews/5DP0N1P76E.html

㊁ http://www.ipswitch.com/support/whatsup_professional/guides/WhatsUpDBSchema.zip

㊂ http://support.ipswitch.com/kb/WP-20041122-DM01.htm

㊃ http://cwe.mitre.org/documents/vuln-trends.html#table1

㊄ http://sla.ckers.org/forum/read.php?3,44

页中的XSS漏洞：

```
POST /rootlogin.asp HTTP/1.1
Host: zero.webappsecurity.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.1.1) Gecko/20061204
Firefox/2.0.0.1
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/
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
```

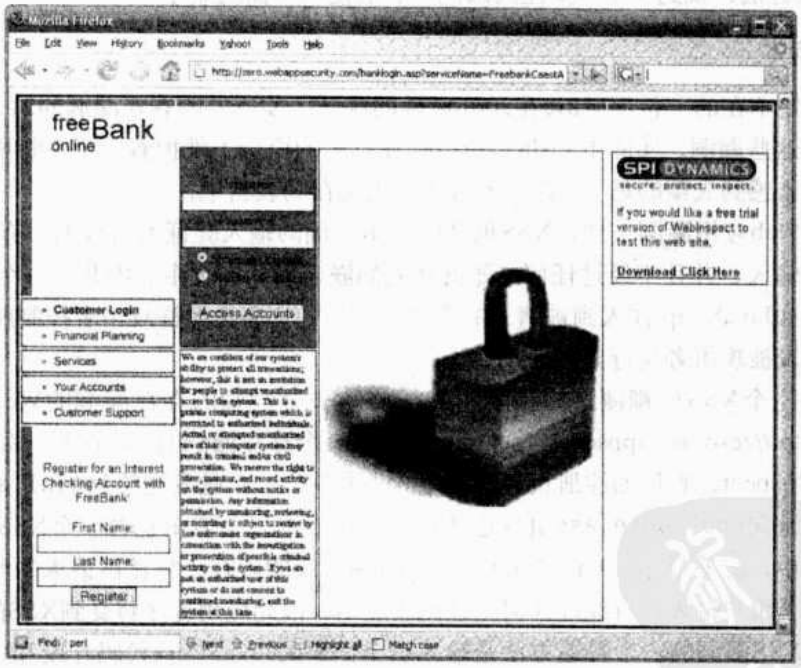


图10-14　SPI Dynamics免费银行应用

从图10-15中可以看出，在网页上显示了一个弹出窗口。然而，当开始进行模糊测试后，这并不是一个实际可用的漏洞检测机制，因为它要求我们要坐在那里观察结果。而模糊测试最大的优越性就是它可以自动化的运行，因此我们可以离开机器，而当我们返回时就可以得到测试结果。

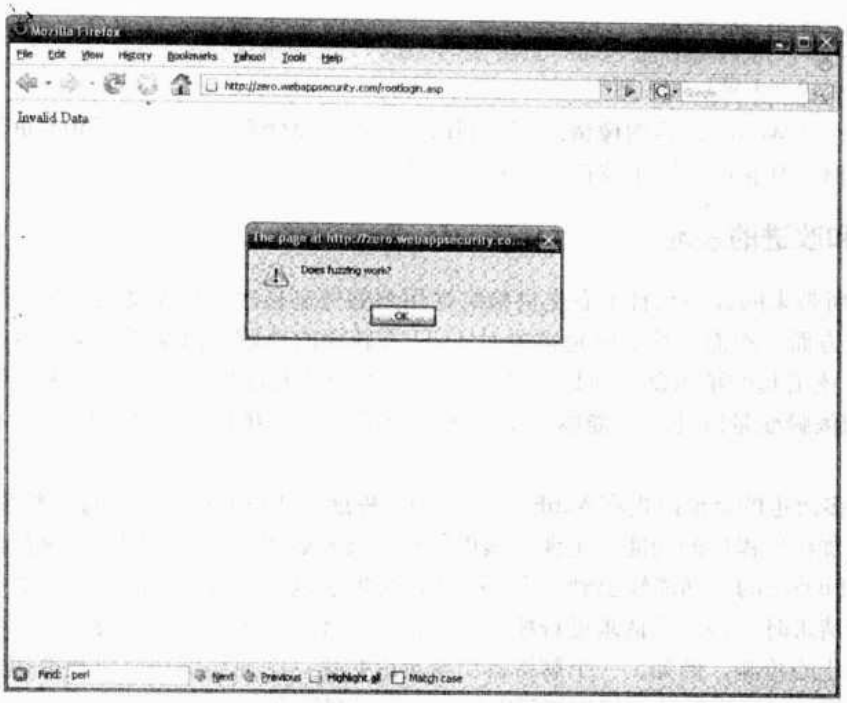


图10-15 运行中的一个跨站点脚本攻击

幸运的是，由于我们可以将代码注入到页中，因此有多种选择。那么注入JavaScript，并且当执行成功时将通过电话通知我们又如何呢？这是可以做到的，但是我们可以采取一种更加简单的方法。客户端脚本不一定必须是JavaScript，它可以是浏览器能够解释的任意脚本语言。HTML如何呢？它也是一种客户端脚本语言，并且它比JavaScript更难以通过黑名单进行过滤，因此在某种程度上它提供了一种更强大的XSS测试。一个HTML IMG标签为我们提供了非常简单的电话通知功能。我们所需要做的所有工作就是使用模糊器注入一个HTML IMG标签，从一个本地web服务器请求一个虚假页。当模糊测试完成后检查服务器日志，如果发现了该请求，那么就表明该目标应用存在XSS漏洞。我们需要立即将一个适当的模糊变量添加到WebFuzz中。这就是WebFuzz为什么被设计为从纯文本文件中抽取模糊变量的原因。添加新变量非常简单，并且不需要再次编译应用程序。为了执行测试，我们将下面的行添加到xssinjection.txt文件。

```
%3Cimg+src%3D%27http%3A%2F%2Flocalhost%2Fblah%27%3E
```

这只是下面图像请求的一个URL编码版本，该图像请求试图从我们的本地web服务器中检索一个虚假页。

```
<img src='http://localhost/blah'>
```

当检查我们的Web服务器日志文件时，将会看到什么呢？

```
#Software: Microsoft Internet Information Services 5.1
#Version: 1.0
```

```
#Date: 2007-01-31 00:57:34
#Fields: time c-ip cs-method cs-uri-stem sc-status
00:57:34 127.0.0.1 GET /xss 404
```

执行失败！当WebFuzz将图像请求注入到rootlogin.asp页时，产生了404-页没有找到的日志条目，同时也从正面证明了该页存在XSS漏洞。

10.6 益处和改进的余地

WebFuzz所带来的好处来自于它提供给终端用户的控制层次。也就是说，用户可以完全控制请求的所有方面。然而，这同时也需要对HTTP有较深的理解，以生成一个有意义的、更加奏效的请求。该工具的请求部分可以被改进的一个方面是允许更加复杂的模糊变量组合。例如，可以添加依赖变量以使每次能够添加多于一个的变量，并且一个变量值由另一个变量的值所决定。

这里有许多改进的余地以提高WebFuzz的自动化程度，使得允许完整的覆盖整个应用程序。这可以通过添加蜘蛛程序的功能来实现，该程序可以被初始化运行以识别所有可能的请求及其结构。将WebFuzz中的该功能输出到一个Web浏览器扩展也将是有价值的，因为它允许在网络中遇到特定的请求时，将这些请求进行模糊化。最后，我可以预见将来能够对识别潜在漏洞的请求进行自动化的检测。例如，一个解析器引擎可以扫描一个原始响应以寻找像用户提供的数据那样的标识符，响应中的这些标识符可以指示XSS漏洞的存在。

现在我们已经使测试工具开始运转了，下面的工作就要靠你了。提高该工具的性能，并且通过将你所做的改进提交给我们来使其他人共享此改进，这样我们就可以将该工具提升到一个新的版本。

10.7 小结

Web应用向我们展示了一些独特的挑战，但它们对模糊测试而言却是确定的合适的目标应用。我们已经阐述了在发布你的Web应用程序之前，使用模糊测试来发现其漏洞的价值。WebFuzz只是一个概念上的试验，但即使是在其原始版本状态，它也可以发现漏洞。我们希望你获取并仔细研究WebFuzz的源代码，将它创建为一个更加有效的工具。



第11章 文件格式模糊测试

"如果这是专政，可能会令人很不舒服，但我就是一个独裁者。"

——George W. Bush, Washington DC，2000年12月19日

文件格式模糊测试是一种针对特别定义的目标应用的特殊的模糊测试方法，这些目标应用通常是客户端应用。其中的例子包括媒体播放器、Web浏览器以及office办公套件。然而，目标应用也可以是服务器程序，如防病毒网关扫描器、垃圾邮件过滤器以及常用的邮件服务程序。文件模糊测试的最终目标是发现应用程序解析特定类型文件的缺陷。

大量的客户端文件格式解析漏洞在2005年和2006年被发现，其中许多是被进行恶意攻击的团体所发现的，因为在典型的漏洞发掘过程之前，发现了许多在安全补丁发布前而被了解和利用的漏洞信息。eEye安全研究组开展了很好的工作以将这些披露的漏洞详细描述在他们的Zero-Day跟踪器中[⊖]。这里有大量的事实表明这些漏洞中的绝大部分是通过文件格式模糊测试而发现的。这种类型的错误远没有灭绝的迹象，这使得文件格式模糊测试成为一个非常有趣并且热门的研究课题。

在本章中，我们提出了实现文件模糊测试的不同方法，同时也讨论了特定目标应用接收输入的不同方式。最后，我们说明了文件模糊器将会遇到的普遍漏洞，并且给出了在实际中检测这些漏洞的建议。当然，所要做的第一项工作就是选择一个合适的目标应用。

11.1 目标应用

就像传统类型的模糊测试一样，使用文件格式模糊测试可以发现许多不同类型的漏洞。同样，这里也有许多不同类型的发掘场景。例如，有些情形要求一个攻击者向用户发送一个恶意文件，并且让他或她手工将文件打开。而其他的情形可能只要求一个用户浏览被攻击者控制的Web页。最后，有些场景能够通过简单地经由邮件服务器或防病毒网关发送一个恶意邮件而触发。最后一个场景就是表11-1提到的微软的Exchange TNEF漏洞，同时该表也列出了其他文件格式漏洞的例子。

表11-1 含有已发现的文件格式漏洞的常用应用程序类型和示例

应用分类	漏洞名	查询网站
Office办公套件	微软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

⊖ http://research.eeye.com/html/alerts/zeroday/

（续）

应用分类	漏洞名	查询网站
防病毒扫描器	卡巴斯基防病毒引擎CHM文件解析器缓冲区溢出漏洞	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可以允许远程代码执行	http://www.microsoft.com/technet/security/Bulletin/MS06-055.mspx
存档工具	WinZip MIME解析器缓冲区溢出漏洞	http://www.idefense.com/intelligence/vulnerabilities/display.php?id=76
邮件服务器	微软Exchange TNEF解码器漏洞	http://www.microsoft.com/technet/security/Bulletin/MS06-003.mspx

你将会发现大多数的目标应用都属于这些分类中的某一个类别。某些应用则由于它们的次要功能而同时属于多个分类。例如，许多防病毒扫描器也包括解压缩文件的库，以允许它们作为存档工具。还有一些内容扫描器声称可以为情色内容分析图像文件。这些程序也可以被认为是图像查看器⊖。应用程序共享公共库是很普遍的，在这种情况下一个单独的漏洞可以影响多个应用。例如，考虑一下在微软的安全公告MS06－055中详细描述的漏洞，它既可以影响Internet Explorer也可以影响Outlook。

11.2 方法

文件格式模糊测试与其他类型的模糊测试是不同的，因为它通常是在一个主机上完整地执行。当执行Web应用或网络协议模糊测试时，你很可能至少会使用两个系统，即一个目标系统和一个模糊器所运行的系统。通过能够在一个单独的机器上进行模糊测试而使性能得到提高，这使得文件格式模糊测试成为一种很有吸引力的漏洞发现方法。

对于基于网络的模糊测试而言，在目标应用中何时发生一个有趣的条件通常是很明显的。在许多情况下，服务器将关闭或者立即崩溃，并且将不能够再连接。而对于文件模糊测试而言，主要是在对客户端应用进行模糊测试时，模糊器将会继续重新开始运行并且销毁目标应用，因此如果不使用适当的监视机制，那么模糊器将可能识别不出一个崩溃情形。这是文件格式模糊测试比网络模糊测试更加复杂的一个应用领域。对于文件格式模糊测试而言，模糊器通常必须要监视目标应用的每次执行以发现异常。这通常可以通过使用一个调试库来动态地监视目标应用中已处理和未处理的异常来实现，同时要将结果记为日志以便于以后的评审。从较高的层次上看，一个典型的文件模糊器的工作过程如下：

1. 通过变异或者生成来准备一个测试用例（后者使用的更多）。
2. 部署目标应用并指示其加载测试用例。
3. 监视目标应用以发现错误，通常是使用一个调试器。

⊖ http://www.clearswift.com/solutions/porn_filters.aspx

4. 当发现一个错误时，记录日志。同样，如果经过一段时间之后没有发现错误，手工销毁目标应用。

5. 重复上述过程。

文件格式模糊测试可以通过生成和变异两种方法来实现。尽管这两种方法在实践应用中都很有效，但显然变异或“强有力”的方法更加易于实现。尽管生成方法或“智能强有力”模糊测试的实现要花费更多的时间，但它们可能将发现用其他更原始的强有力方法所不能发现的一些漏洞。

11.2.1 强制性或基于变异的模糊测试

使用强制性模糊测试方法，你需要首先收集目标文件类型的一些不同的例子。你所能找到的不同文件越多，那么你的测试将越彻底。然后模糊器将作用在这些文件上，创建它们的变异文件，并通过目标应用解析器将它们进行发送。这些变异文件可以具有任意形式，这取决于在模糊器中所选择的方法。你可以使用的一个方法是用数据字节来替换字节。例如，遍历文件并用0xff替换每个字节。你也可以为多个字节执行此操作，例如两字节或四字节。你同样可以向文件中插入数据，而不只是覆盖字节。在测试字符串值时，这是一种非常有用的方法。然而，当向文件中插入数据时，需要注意可能会扰乱文件中的偏移量。这将会严重地破坏对代码的覆盖，因为某些解析器将会快速检测一个无效文件并且退出。

校验和也可以破坏强制性解析器。由于任何的字节改变都将使校验和失效，因此解析应用程序很可能将退出，并且在一个潜在的漏洞代码片段能够被到达之前提供一个错误消息。该问题的解决方法，一是转换到智能模糊测试，这将在下一节中进行讨论。或者是停止在目标软件中的校验。停止软件的校验并不是一项简单的工作，通常需要进行逆向工程的大量工作。

一旦该方法被实现之后，为什么非常便于使用呢？答案很简单。因为终端用户不需要具备文件格式的任何知识，并且也不需要了解其工作机理。如果他们可以使用常用的搜索引擎或者通过搜索他们的本地系统来找到一些示例文件的话，他们就可以通过自己的研究来进行操作，直到模糊器发现某些感兴趣的内容。

这种模糊测试方法也具有一些缺点。首先，它是一种效率不高的方法，因此需要花费较多的时间来完成对一个单一文件的模糊测试。例如，对于一个基本的word文档的测试。即使是一个空文档，其大小也大约有20KB。每次对一个字节进行模糊化，那么将需要创建并部署20 480个独立的文件。假定每个文件的处理需要2秒钟，那么总共将要花费11个小时，而这只是对一个单一字节值的处理。那么处理其他254个字节将会怎样呢？可以通过使用多线程模糊器在某种程度上回避这个问题，但它却说明了单纯的变异模糊测试具有不高的效率。简化这种模糊测试的另外一种方法是只关注文件中最有可能产生期望结果的那部分，如文件头和字段头。

强有力模糊测试的主要缺点是总是有许多功能将会被遗漏掉，除非你以某种方式收集到包含每个和所有可能特性的一个示例文件集。大多数的文件格式都是很复杂的，并且包含大量的变换。当度量代码覆盖率时，你将会发现将一些示例文件使用于一个应用中并不像另外一种情形那样彻底地执行应用，这另外的情形就是用户真正理解了文件格式，并且手工准备了一些针

对文件类型的信息。这个问题可以使用文件模糊测试中的生成方法来解决，也即我们所说的智能强有力模糊测试。

11.2.2 智能强制性或基于生成的模糊测试

使用智能强制性模糊测试，你必须首先花费一些精力来实际的研究文件规范。一个智能模糊器仍然是一个模糊测试引擎，因此仍然执行一个强制性攻击。然而，它依赖来自于用户的配置文件，以使测试过程更加智能。这些文件通常包含描述文件类型语言的元数据。可以将这些模板想象为数据结构的列表，它们的位置及其可能的取值都是相互关联的。在一个实现的层面上看，这些模板可以用许多不同的格式来表示。

如果选择一个没有任何公开文档的文件格式用于测试，那么作为一个研究者，你必须要在创建一个模板之前对格式化规范进行深入地研究。这可能需要逆向工程的知识，但首先要使用你的好朋友Google来查看一下是否已经有其他人为你做了这项工作。一些Web站点，如Wotsit的Format[⊖]，就提供了很好的官方和非官方的文件格式文档服务。一个可选的作为补充的方法是比较文件类型的这些例子以发现一些模式，并且剖析一些被使用的数据类型。记住，一个智能模糊器的有效性直接相关于你对文件格式的理解以及采用一种通用方法将其描述给所使用的模糊器的能力。在第12章“文件格式模糊测试：UNIX平台上的自动化测试”中，当创建SPIKEfile时，我们将展示智能强有力模糊器的一个示例实现。

一旦一个目标应用和测试方法被确定，下一步就是针对所选定的目标应用，研究如何生成适当的输入向量。

11.3 输入

选择了一个目标应用之后，下一步就是枚举出该应用所支持的文件类型和不同向量的扩展，以使这些文件被解析。可用的格式化规范也应当被收集并评审。即使在你只希望执行一个简单的强有力测试的情况下，具有相关的文件格式知识仍然是很有用的。关注于更加复杂的文件类型会更加有利，因为实现一个恰当的解析器将会是更加复杂的，因此发现一个漏洞的机会就会增加。

让我们通过一个例子，来看一下如何收集输入。存档工具WinRAR[⊖]是一个免费使用的流行存档工具。了解WinRAR处理何种文件的一个简单方法是浏览WinRAR网站。在该网站的主页上，你将会看到它所支持的文件类型的一个列表。这些类型包括zip,rar,tar,gz,ace,uue以及其他一些类型。

现在你已经知道了WinRAR所能处理的文件类型的列表，下一步就必须选择一个目标应用。有些时候，选择一个目标应用的最好方法就是寻找关于每个文件类型的信息，然后选择其中最复杂的一种类型。这里所假设的前提条件是复杂性通常会导致编码错误。例如，使用一些长标

⊖ http://www.wotsit.org

⊖ http://www.rarlab.com

记值和用户提供的偏移量的文件类型可能会比基于静态偏移量和静态长度字段的简单文件类型更加具有吸引力。当然，一旦你掌握了模糊测试，你将会发现该规则也有许多例外的情况。在理想情况下，模糊器将最终面向所有可能的文件类型；第一个选择并不是十分重要，然而它总是有利于在针对特定应用的第一个模糊测试集中发现有趣的行为。

11.4 漏洞

当解析一个不规则文件时，一个编写的很差的应用程序易于受到不同类别漏洞的攻击。本节讨论其中的一些漏洞类别：

- DoS（应用崩溃或挂起）
- 整数处理问题
- 简单的栈/堆溢出
- 逻辑错误
- 格式化字符串
- 竞争条件

11.4.1 拒绝服务

尽管DoS问题在客户端应用中并不是十分有趣，但你需要记住我们也可以面向服务器应用，而该应用必须要保持对安全性和生产力的可用性。当然，这包括邮件服务器和内容过滤器。在实践应用中，导致文件解析代码中的DoS问题的一些常见原因包括越界读取、无限循环和空指针解除引用。

导致无限循环的一个通常错误是相信文件中制定其他块的位置的偏移量值。如果应用程序不能保证该偏移量使当前块向前移动，那么就会发生一个无限循环并导致应用程序无限制地重复处理同一个块或多个块。以前在ClamAV中已经有许多该类问题的实例[⊖]。

11.4.2 整数处理问题

整数溢出和“有符号”问题在二进制文件解析中是非常普遍的。我们所看到的一些最常见的问题类似于下面的伪代码：

```
[...]
[1] size            = read32_from_file();
[2] allocation_size = size+1;
[3] buffer          = malloc(allocation_size);
[4] for (ix = 0; ix < size; ix++)
[5]       buffer[ix] = read8_from_file();
[...]
```

这个例子说明了一个典型的导致内存破坏的整数溢出情况。如果文件为size值指定了最大的无符号32位整数（0xFFFFFFFF），于是第[2]行中的allocation_size变量由于一个整数溢出

⊖ http://idefense.com/intelligence/vulnerabilities/display.php?id=333

而被赋值为0。在第[3]行，该代码将导致用大小为0的值来调用内存分配函数。这时，指针buffer将指向一个未分配的内存块。在第[4]行和第[5]行，应用程序执行循环并将个数等于size的初始值的大量数据复制到所分配的缓冲区中，从而导致一个内存破坏。

这种特殊情形并不总是能够被发现。它的可发现性取决于应用程序使用堆的方式。简单的覆盖内存中的堆通常不足以获得对应用程序的控制权。必须要采取一些操作使得被覆盖的堆数据能够被使用。在某些情况下，类似于此的整数溢出在堆内存被使用之前将导致一个非堆相关的崩溃的发生。

当然这只是一个例子，说明了在解析二进制数据时整数如何被错误使用。整数的错误使用还表现在其他许多方面，包括经常提到的无符号整数比较错误。下面的代码片段说明了此类漏洞的内部逻辑：

```
[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值是一个负数，那么该段代码将产生一个基于栈的溢出。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]行定义）都被作为有符号数来处理，例如，－1小于512。在后面，当size在第[7]行的函数中被用做复制的边界变量时，它是被作为无符号数来处理的。例如，此时的值－1将被解释为42949672954294967295。当然，并不能保证可以发现这类情况，但在很大情况下依赖于readx_from_file函数的实现方式，可以通过监视变量以及将寄存器保存到栈的方法来发现此情况。

11.4.3　简单的栈和堆溢出

该问题已经被很好的理解，并且在以前已经出现过许多次了。一个典型的场景如下：一个固定大小的缓冲区被分配，无论它是位于栈中还是位于堆中。稍后，当从文件复制超过缓冲区大小的数据时，没有执行任何边界检查。在某些情况下，虽然会试图进行一些边界检查，但这

些检查也是不正确的。当复制操作发生时，内存被破坏，通常还会导致任意代码执行。关于该类漏洞的更加详细的信息，请参阅The Shellcoder's Handbook: Discovering and Exploiting Security Holes《Shell编码手册：发掘并利用安全漏洞》[一]。

11.4.4 逻辑错误

依赖于文件格式的设计，可利用的逻辑错误可能会存在。尽管在对文件格式漏洞的研究过程中，我们没有亲自发现任何逻辑错误，但该类漏洞的一个极好的例子就是MS06-001所描述的微软的WMF漏洞[二]。该漏洞并不是源自于一个典型的溢出。实际上，它不要求任何类型的内存破坏，它允许一个攻击者直接执行用户提供的位置无关的代码。

11.4.5 格式化字符串

尽管大部分格式化字符串漏洞已经不存在了，特别是在开源软件中，但仍然有必要提及它们。当我们讲大部分不存在时，之所以这样说是因为并不是所有的程序员都能像US-CERT的程序员那样关注安全，US-CERT的程序员建议为了保证你的软件的安全性，不应当使用“%n”格式化字符串指定符[三]。

但情况更加严重，在我们亲自进行的实验中，当对文件进行模糊测试时确实发现了一些与格式化字符串相关的问题。其中的某些问题已经在Adobe®和RealNetworks®中发现。发现格式化字符串问题的许多乐趣来自于能够使用漏洞以泄露内存，从而帮助发现问题。不幸的是，由于使用非格式化文件的客户端攻击，你很少能够被提供这样的机会。

11.4.6 竞争条件

尽管人们通常不认为文件格式漏洞是由于竞争条件所引起的，但是有一些人是这样认为的，并且可能还会有更多的人这样认为。这种类型漏洞的主要目标应用是复杂的多线程应用。我们反对只面向一个特定的产品，但是微软的Internet Explorer是这里所想到的第一个应用程序。Internet Explorer使用非初始化内存以及其他线程正在使用的内存而导致的漏洞，将可能会被不断发现。

11.5 漏洞检测

当对文件格式进行模糊测试时，通常要产生目标应用的许多实例。其中一些将会不确定的挂起并且必须要被销毁掉，一些将会发生执行崩溃，而另外一些将会主动退出。所面对的挑战是需要确定一个已处理和未处理的异常何时发生，以及该异常何时被发现。一个模糊器可以利

[一] ISBN-10:0764544683

[二] http://www.microsoft.com/technet/security/Bulletin/MS06-001.mspx

[三] https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/guidelines/340.html

[四] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=163

[五] http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=311

用一些源信息以发现一个进程的更多信息：

- 事件日志。事件日志被微软的Windows操作系统所使用，它可以通过使用事件查看器应用程序来访问。事件日志对于我们来说并不是十分有用，因为当我们在模糊测试的过程中部署了成百上千个进程时，它很难将一个事件日志条目同一个特定的进程相关联。
- 调试器。识别未处理和已处理异常的最好方法是在进行模糊测试之前将一个调试器关联到目标应用。错误处理将阻止由模糊测试而引起的许多错误的明显征兆的出现，但是这些错误通常可以使用调试器来发现。还有许多比使用调试器来进行错误检测更加先进的技术，其中的一些技术将在第24章“智能故障检测”中介绍。
- 返回代码。捕获并测试应用程序的返回代码，虽然它不如使用调试器那样准确并提供详细的信息，但它是一种非常快速有力的确定应用程序终止原因的方法。至少在UNIX平台上，它可以通过返回代码来确定是什么信号导致了应用程序终止。
- 调试API。不使用第三方的调试器，而在模糊器中实现某些基本的调试功能通常是可行并且有效的。例如，对于一个进程而言我们所感兴趣的是它被终止的原因，当前的指令是什么，寄存器的状态以及栈指针或某些寄存器区域的内存值等。通常实现这些功能都很简单，并且在分析崩溃原因以查找漏洞的时间节省方面没有应用价值。在下一章中，我们将研究该内容，并提出一个在微软Windows平台上的名为PyDbg的简单可复用的调试器创建框架，该框架是PaiMei[⊖]逆向工程架构的一部分。

一旦一个给定的测试用例被确定用来导致一个错误，则除了保存触发错误的实际文件之外，还必须要保存被你所选择的任何错误监视方法所收集的信息。同样，在你的记录中保存测试用例的元数据也是非常重要的。例如，如果模糊器使用第42个模糊值正在对文件中的第8个变量字段进行模糊测试，那么该文件就可以被命名为文件－8－42。在某些情况下，我们可能会释放一个内核文件并且也将其保存。如果一个模糊器使用调试API来捕获信号的话就可以实现这一点，关于它的特定实现细节可以在下两章中找到。

11.6 小结

尽管文件格式模糊测试是一种定义面较窄的方法，但是仍然有许多的目标应用和大量的攻击方法。我们不仅讨论了传统的客户端文件格式漏洞，而且涉及了一些远程服务的情形，如防病毒网关和邮件服务器。强调的重点放在了阻止和发现通过TCP/IP进行的基于网络的攻击，文件格式对于深入剖析内部网络段仍然是一种有价值的手段。

⊖ http://www.openrce.org/downloads/details/208

第12章 文件格式模糊测试：UNIX平台上的自动化测试

“我是指挥官（当然，我不需要对此解释），我不需要解释为什么这么说。这就是当总统的有趣之处。”

——George W. Bush，引自Bob Woodward的“Bush at War”

文件格式漏洞既可以在客户端被利用，如Web浏览器和office套件，也可以在服务器端被利用，如邮件扫描器和防病毒网关等。关于在客户端利用的问题，受影响客户端的广泛使用与该问题的严重程度直接相关。例如，影响微软的Internet Explorer的一个HTML解析漏洞，在严重性被考虑之前是被最为广泛使用的漏洞之一。相反，局限于UNIX平台的客户端漏洞因其受限暴露而不被关注。

然而，在本章中我们将介绍两个模糊器，即notSPIKEfile和SPIKEfile，它们分别实现了基于变异的文件格式模糊测试和基于生成的文件格式模糊测试。我们将讨论这两种工具所具有的特性及其不足。然后，进一步深入到工具的开发过程中，阐述这些工具的开发方法并分析一些核心的代码片段。我们也将论述一些基本的UNIX概念，如感兴趣和不感兴趣的信号，以及僵死过程。最后，我们将介绍这两种工具的应用，并解释选择所用开发语言的原因。

12.1 NOTSPIKEFILE和SPIKEFILE

被开发用来说明在UNIX上进行文件模糊测试的两个工具是notSPIKEfile和SPIKEfile。正如它们的名字所示，它们一个是基于SPIKE，[⊖]而另一个则不基于SPIKE。下面列出了它们在实现中所提供的一些关键特性：

- 集成了最小化的调试器，能够检测已处理和未处理的信号，并清空内存和寄存器状态。
- 使用用户所指定的销毁目标进程之前的延迟来实现完全的自动化模糊测试。
- 针对目标二进制代码和ASCII可打印数据类型的两个不同的启发式模糊数据库。
- 以导致问题的历史数据值为基础，实现简单的、易扩展的启发式模糊数据库。

缺少的特性

notSPIKEfile和SPIKEfile中缺少了一些特性，而这些特性对于使用这两种工具的某些用户来说可能是有用的。下面列出了这些所缺少的特性：

- *可移植性*。由于集成调试器被创建工作在x86上，并且使用Linux `ptrace`，那么它在其他体系架构或操作系统中将不能工作。然而，增加兼容性是非常简单的，并且提供了其他调试和解析库的可用性。

⊖ http://www.immunityinc.com/resources-freesoftware.shtml

- 智能加载监视。尽管用户可以立即指定对多少进程进行模糊测试，但是当前非常需要用户来确定系统是否过载。

显然，在开发过程当中做出了一些折衷考虑，正如这些缺少的特性所描述的那样。工具的开发者通过声明将改进留作“读者的一个练习”而利用一个古老而隐蔽的工具。让我们来看一下在这些工具的创建过程中所实施的开发活动。

12.2 开发方法

本章的焦点不在于如何使用一个模糊器，而在于如何实现一个模糊器。本节描述文件格式模糊器在Linux平台上的设计和开发细节。在文件格式模糊器的开发阶段，我们可以放心的假设我们的模糊器将在与目标应用相同的系统中执行。我们的设计包括三个不同的功能构件。

12.2.1 异常检测引擎

这部分代码负责确定目标应用在何时说明某些未定义的潜在的不安全行为。如何实现这一点呢？这里有两个基本的方法。第一个方法相当简单，它只是简单地监视被应用程序所接收的任意信号。这就允许模糊器检测将导致一个无效内存引用的错误，如缓冲区溢出。当然，该方法将不会捕获诸如元字符注入或逻辑缺陷等行为。例如，考虑一个应用程序将部分由攻击者提供的值传递给UNIX系统函数。这将允许一个攻击者通过使用Shell元字符来执行任意程序代码，而不会引起任何类型的内存访问违规。尽管这里存在漏洞以允许一个攻击者危及主机的安全性，但该安全性完全不包括内存破坏。因此，最后所作出的决定是我们并不对这样的错误感兴趣，因为检测这些错误所需要付出的努力要远大于所得到的回报。

对于那些想要发现这些逻辑错误的人来说，一个好的方法是钩住C库函数（LIBC），并监视被不安全的传递给函数调用的那些由模糊器提供的值，这些函数调用包括`open`，`creat`，`system`等。

对于我们的设计而言，我们决定简单地使用系统的`ptrace`调试接口来检测信号。尽管通过简单地等待应用程序返回来确定一个信号导致了应用程序的终止是不重要的，但还要做一些工作来检测被应用程序在内部进行处理的那些信号。正因为如此，异常检测引擎所采用的开发方法要严重依赖于系统的调试接口，在这种情况下就是`ptrace`系统调用。

12.2.2 异常报告（异常检测）

在发现异常的过程中，一个好的模糊器应当对所发生情况的有用信息做出报告。至少应当报告所产生的信号。在理想情况下，更加详细的信息如恶意指令，CPU寄存器状态以及一个栈的清空也应当包含在报告中。本章所实现的这两个模糊器都能够生成这样的详细报告。为了获取这些所需要的、低层次的信息，我们必须在系统调用`ptrace`的帮助下来收集这些信息。并且如果我们希望向用户报告这些信息，那么还要利用一个库来解析指令。该库应当能够将任意数据的一个缓冲区转换为x86指令的字符串表示。因其性能的良好性我们选择了libdisasm库[⊖]，

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

它提供了一个非常简单的接口，并且被看做是Google搜索引擎的最佳选择。为了提供其实用性，libdisasm包括了一些示例，我们可以从其中剪切和粘贴代码，极大地提高了使用的便利性。

12.2.3 核心模糊测试引擎

文件格式模糊器的核心是，它控制着使用什么样的异常数据以及在哪里使用这些数据。notSPIKEfile中实现此功能的代码与SPIKEfile中实现此功能的代码是不同的，因为SPIKEfile是利用已经存在的SPIKE代码来实现此功能的。但其余部分二者是非常相似的。

毫无疑问，你可能已经猜到了，SPIKEfile使用了与SPIKE相同的模糊测试引擎。模糊测试引擎需要一个模板，即描述文件格式的一个SPIKE脚本。然后，SPIKE将使用有效数据和无效数据的组合来智能生成该格式的不同变体。

在notSPIKEfile中，模糊测试引擎的局限性更大。用户必须要向模糊器提供一个有效的目标文件。然后，引擎使用一个模糊值数据库来对该文件的不同部分进行变异。这些值划分为二进制和字符串两种类型。二进制值可以是任意长度以及任意值，但通常被用来表示公共整数字段的大小。字符串值正如其名字所示，只是表示字符串数据。这些字符串数据可以包括长或短的字符串，特定格式的字符串以及其他所用种类的异常值，如文件路径、URL和你能想到的任意其他类型的不同字符串。为了得到这些类型的一个完整列表，可以看SPIKE和notSPIKEfile的源代码，但作为一个入门，表12-1为一些有效的模糊字符串提供了一个简明的列表，并给出了简短的解释。虽然它和完整的列表相差很多，但它有助于你理解什么类型的输入应当考虑。

表12-1 一些常用的模糊字符串及其含义

字符串	含 义
“A”x10000	长字符串，可能导致缓冲区溢出
“%n%n”x5000	含有百分号的长字符串，可能导致缓冲区溢出或触发格式化字符串漏洞
HTTP://+“A”x10000	有效的URL格式，可能触发URL解析代码中的缓冲区溢出
“A”x5000+“@”+“A”5000	有效的邮件地址格式，可能导致邮件地址解析代码中的缓冲区溢出
0x20000000,0x40000000 0x80000000,0xffffffff	一些可能触发整数溢出的整数。在此可以充分发挥你的创造性。考虑一下代码`malloc(user_count*sizeof(struct blah));`也考虑一下没有对溢出和下溢进行检查就递增或递减整数的代码。
“../”x5000+“AAAA”	可能触发路径或URL地址解析代码中的溢出

实际上，这里对你可以使用的模糊字符串的数量并没有限制。然而需要注意的是，任意类型的值可能会要求进行特殊的解析，或者是产生一个应当被很好表示的异常条件。造成使用模糊器错过和发现一个错误的不同结果的原因，可能就是由于一个简单的操作，如将.html追加到一个长模糊字符串的末尾。

在下一节，我们将考察一些更有趣、更相关的代码片段，这些代码片段既来自SPIKEfile也来自notSPIKEfile。

12.3 有意义的代码片段

这两个模糊测试工具中共享的大部分核心功能都在下面进行了分析。例如，首先强调一下对子进程进行派生和跟踪的基本方法。下面所引用的代码是采用C语言编写并用于这两个模糊器的代码：

```
[...]
    if ( !(pid = fork ()) )
    { /* child */
         ptrace (PTRACE_TRACEME, 0, NULL, NULL);
         execve (argv[0], argv, envp);
    }
    else
    { /* parent */
          c_pid = pid;
monitor:
          waitpid (pid, &status, 0);
          if ( WIFEXITED (status) )
          { /* program exited */
              if ( !quiet )
          printf ("Process %d exited 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 )
               fprintf (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) == SIGTRAP) ? 0 :
```

```
WSTOPSIG (status))) == -1 )
        {
            perror("ptrace");
        }
        goto monitor;
    }
    return(ERR_OK);
}
```

主进程或父进程开始于为目标应用派生一个新的进程。该新进程或子进程使用ptrace调用来表明它将通过发出PTRACE_TRACEME请求来被其父进程跟踪。然后，该子进程继续目标应用，并知道其父进程会像任何执行良好的进程一样监视其是否会执行任何不恰当的操作。

作为父进程，模糊器能够接收发送给子进程的所有信号，因为子进程使用了PTRACE_TRACEME请求。父进程甚至会接收对执行函数家族的任何调用的一个信号。父循环是简单明了并且作用强大的。模糊器循环接收发送给子进程的所有信号。模糊器根据信号和子进程的状态而采取不同的行为。

例如，如果进程中断运行，这意味着程序还没有退出，但是它接收到了一个信号并等待父进程做出判断以允许它继续执行。如果该信号表示了一个内存破坏问题，那么模糊器将把信号传递给子进程，并假定它能够销毁子进程并报告结果。如果该信号本质上并没有什么问题或者只是一个很平常的信号，那么模糊器将把它传递给应用程序而不执行任何监视。

模糊器同时要检查子进程是否真的被中断，或者是其他的非关联情形。你可能会问，如果应用程序崩溃会发生什么情况呢？我们是否应当对此给予关注呢？好的，由于我们在应用程序实际中断运行之前，已经截取了所有感兴趣的信号，我们知道应用程序中断运行是因为一个自然的行为或者是一个不感兴趣的信号。这就强调了你理解应当对哪些信号感兴趣是非常重要的。对于那些寻找DoS漏洞的人来说，他们可能会对一个浮点数异常感兴趣。而另外一些人则可能只对真正的内存破坏问题感兴趣。那么其他人可能会等待一个异常中断信号，该信号已经成为了新版本GLIBC的堆破坏的一个指示器。已经表明，在某些情况下这些堆破坏检查可以通过暂时解决该问题来执行任意代码[⊖]。

在查看了处理特定信号的这些代码之后，可能会产生这样的疑问，即为什么对特定信号的处理方法不同于处理其他信号的方法呢？下面是在对漏洞进行研究的环境中，针对UNIX信号的一个解释。

12.3.1 通常感兴趣的UNIX信号

表12-2提供了漏洞研究者在模糊测试过程中可能感兴趣的信号的一个列表，同时给出了感兴趣的原因。

⊖ http://www.packetstormsecurity.org/papers/attack/MallocMaleficarum.txt

表12-2　在UNIX平台上执行模糊测试的一些感兴趣的信号

感兴趣的信号名	含　义
SIGSEGV	无效的内存引用。成功模糊测试的最常见结果
SIGILL	非法指令。它是内存破坏所产生的一个可能的副作用，但是相对很少发生。它通常是由于程序计数器被破坏或在数据和指令中间停止而引起的
SIGSYS	错误的系统调用。它也是内存破坏产生的一个可能的副作用，但是相对也很少发生（实际上是非常少发生）。造成产生该信号的原因与SIGILL是相同的
SIGBUS	总线错误。通常是由于某些形式的内存破坏，或者是不正确的访问内存。由于它们的排列需求，使得在RISC机器上经常产生此信号。在大多数的RISC实现中，非排列的存储和加载将产生SIGBUS信号
SIGABRT	由异常中断的函数调用所生成。这是非常有意思的，因为当GLIBC发现一个堆破坏时，它将产生异常中断

12.3.2　不太感兴趣的UNIX信号

相对于表12-2，表12-3描述了在模糊测试过程中经常产生，但对于一个漏洞研究者而言不太感兴趣的一些信号。

表12-3　在UNIX平台上执行模糊测试的一些不太感兴趣的信号

不感兴趣的信号名	含　义
SIGCHLD	一个子进程已经退出
SIGKILL,SIGTERM	进程被销毁
SIGFPE	浮点数异常，如被0除
SIGALRM	计时器过期

我们已经介绍了SIGCHLD信号，现在可以讨论如何对一个常见的场景进行处理，而该场景是由于对该信号的不正确处理所导致的。下一节解释了什么是一个僵死进程，以及如何正确处理子进程以使僵死进程不被生成。

12.4　僵死进程

一个僵死进程是这样一个进程，它是从一个父进程中分离出来并且完全执行（即已经退出）的进程，但是其父进程没有通过调用wait或waitpid来得到其状态。当这种情况发生时，整个进程的信息仍然不确定地保留在内核中，直到父进程对其发出请求。这时，这些信息被释放，进程才真正执行完成。从我们的模糊器中分离出的一个进程的通常的生命周期如图12-1所示。

当编写一个使用fork来生成子进程的模糊器时，必须要确保父进程使用wait或waitpid接收进程完成的所有状态信息。如果错过了一个进程的完成，那么将得到一个僵死进程。

早期版本的notSPIKEfile含有一些错误，会导致活动进程的数量缓慢减少，直到模糊器到达一个死锁状态。例如，假定用户一次使用8个进程来对应用程序进行模糊测试。随着时间的推移，活动进程慢慢就会减少到1个。这是由于工具开发者的两个粗心错误而造成的。最初的

设计完全依赖于SIGCHLD信号，该信号在子进程执行完毕后被发送给进程。使用此设计，就会错过某些SIGCHLD信号。同样在有些地方，当子进程执行完毕后，开发者没有成功地减少活动进程的数量，从而导致模糊测试进程的运行速度越来越慢。

曾经提到过，这些错误很好解决。当使用WNOHANG标志使阻止变为非阻止时，所有的wait和waitpid调用都可能导致问题的发生。当它们返回时，检查所返回的状态以查看一个进程是否真正执行完成。如果执行完成，那么活动进程的数量总是递减的。

使用SPIKEfile，不能立刻部署多个应用程序从而极大地简化设计和实现。我们不需要担心错过了SIGCHLD信号，因为每次都有一个信号要返回来。

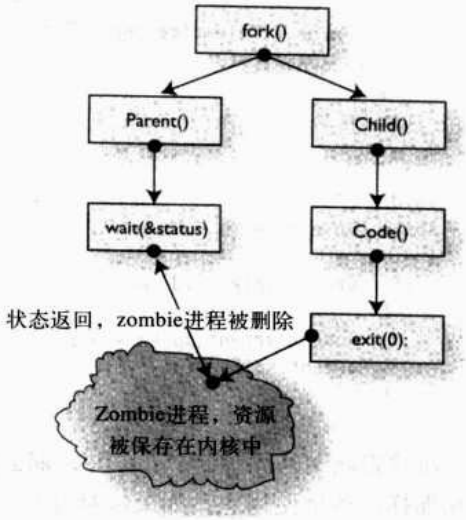


图12-1 一个被分离进程的生命周期

由于SPIKEfile是基于SPIKE的，我们只需要向其中添加一个或两个文件，以使它能够处理文件的输入和输出而不只是传输其输入和输出。通过快速查看一下SPIKE是如何针对TCP/IP进行实现的，就能够简单地提供文件支持。下面的代码是为filestuff.c添加的：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "filestuff.h"
#include "spike.h"

extern struct 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==file,0-68 are reserved by the ISO fuzzing standard
*/
}

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;
    }
}
```

通过将此文件添加到Makefile，那么曾经使用过SPIKE的任何人现在都可以像把文件作为终端那样简单地使用它。如果你对哪些文件被添加到SPIKE中感兴趣，那么表12-4就提供了相关的信息。

表12-4　为生成SPIKEfile而对SPIKE所做的一些更改

文件名	目　的
filestuff.c	包含打开和写文件的程序
util.c	包含在notSPIKEfile和SPIKEfile中所共享的许多代码。包含ptrace函数，主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main函数
include/filestuff.h	filestuff.c的一个头文件
Libdsasm	当发生崩溃的情况时，被用来解析x86指令的库

12.5　使用的注意事项

如果你计划对一个不能直接部署的应用程序使用SPIKEfile或notSPIKEfile，那么需要以某种方式来暂时解决这个问题。这些类型应用程序的典型例子包括Adobe Acrobat Reader和RealNetworks RealPlayer。

通常情况下，对于这些应用程序而言，你实际所运行的程序是一个Shell脚本函数。Shell脚本建立起环境以使真正的二进制代码可以正确运行。因此，应用程序就可以包含它们自身共享库的副本。由于包含了它们自身对公共共享库的副本，就增加了该产品的可移植性。尽管这样做是为了给用户使用带来方便，但对于我们要实现的目的而言，它反而使情况变得更加糟糕。例如，如果我们指定Shell脚本作为模糊测试的文件，那么当其运行时我们将不会被关联到实际的二进制文件，而被关联到Shell的一个实例。这将会使我们的信号捕获变得没有意义。这有一个示例，说明了在Acrobat Reader和RealPlayer中的情况。对于诸如此类的其他应用程序而言，

思想都是类似的。

12.5.1 ADOBE ACROBAT

对Acrobat而言，你必须首先要使用-DEBUG选项来运行acroread。这将会进入到一个被设置了正确环境的Shell脚本，以直接激活实际的acroread二进制代码，该代码通常存在于$PREFIX/Adobe/Acrobat7.0/Reader/intellinux/bin/acroread中。尽管它没有被公开化，并且不存在有用的函数，但是可以通过简单的阅读acroread Shell脚本来确定此信息。你现在可以对此二进制代码进行模糊测试而不会发生任何问题。该方法被notSPIKEfile所使用以发现Adobe Acrobat Reader UnixAppOpenFilePerform的缓冲区溢出漏洞[⊖]。

12.5.2 REALNETWORKS REALPLAYRE

正如下面的输出所显示的，realplay命令实际上是一个Shell脚本。我们也看到实际的二进制应用程序被称作realplay.bin。

```
user@host RealPlayer $ file realplay realplay.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
```

对于RealPlayer而言，你只需要将Shell环境变量HELIX_PATH设置为RealPlayer的安装路径。然后就可以直接对包含在RealPlayer中的二进制文件realplay.bin进行模糊测试了。通过简单地阅读realplay Shell脚本就可以确定此信息。在notSPIKEfile中使用此方法，就可以发现RealNetworks RealPlayer/HelixPlayer RealPix中的格式化字符串漏洞[⊜]。

12.6 实例研究：REALPLAYER REALPIX格式化字符串漏洞

我们现在将要讲述如何使用notSPIKEfile来发现RealPlayer中的一个漏洞，该漏洞于2005年9月被修补。第一步是任意选择RealPlayer所支持的一种文件格式。当然在这个例子中，选择了RealPix格式。在经过大量的搜索之后，一些示例RealPix文件被编译，并被用于notSPIKEfile模糊测试的基础文件。下面给出了一个示例：

```
<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"
```

⊖ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=279

⊜ http://www.idefense.com/intelligence/vulnerabilities/display.php?id=311

```
        aspect="true"/>
</imfl>⊖
```

这只是一个很小的RealPlayer文件的一个框架示例。如果在RealPlayer中加载该文件，那么将不会显示任何内容，因为它只包含一个头。我们将通过notSPIKEfile来运行该文件以测试头解析代码。使用下面的命令来启动模糊测试：

```
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 $
```

我们使用－t选项告诉工具对每个RealPlayer的激活要延续3秒钟。同时，使用－d选项告诉该工具在它销毁一个空闲进程和启动一个新进程之间要等待1秒钟。使用－m选项来指定部署RealPlayer的3个并发实例，并使用－r选项告诉工具对从0字节开始的整个文件进行模糊测试。我们还使用－S选项来指定字符串模糊测试的模式，并使用－s选项指定SIGKILL信号来终止空闲进程。最后，告诉工具为模糊测试文件名所采取的一种格式，并指定示例文件的文件名sample1.rp，并告诉工具如何执行RealPlayer以使它解析我们的文件。一切准备就绪。notSPIKEfile的输出通过报告一些崩溃信息来表明发现了某些类型的漏洞。

我们将当前目录中的文件列出，可以看到文件FUZZY-sample1.rp-0x28ab156b-dump.txt被创建。当我们查看该文件时，就可以看到一个生成的描述崩溃时进程状态的详细报告。同时也可以看到导致崩溃的文件的名字。在这个例子中，它被保存为12288-FUZZY-sample1.rp。该文件可以用来再现崩溃。当查看该文件时，我们得到了该问题可能是什么的一个很好的提示。该文件的内容如下：

```
<imfl>
    <head title="RealPix(tm) Sample Effects"
      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%n字符的出现，我们立刻怀疑是一个格式化字符串漏洞导致了崩溃的发生。当在GDB中部署RealPlayer二进制文件时，我们的怀疑得到了证实。

```
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
```

⊖ http://service.real.com/help/library/guides/realpix/htmfiles/tags.htm

```
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)
```

我们真的在RealPlayer对timeformat选项的处理中发现了一个格式化字符串漏洞。关于如何利用该漏洞进行攻击的问题给读者留作练习思考。

12.7 语言

这些工具之所以采用C语言编写，是由于以下一些逻辑原因。首先，就像盐和胡椒粉的关系一样，C和Linux一直能够并且还将能够相互配合很好地进行工作。每个先进的Linux分布式系统都有一个C编译器，并且可能将会一直拥有，因为Linux的内核就是用C语言编写的。这里不需要任何特殊的库来将ptrace接口暴露给我们的应用程序，我们完全有自由来执行所需要的任务。

这些工具使用C语言编写的另外一个原因是由于SPIKE就是用C编写的。由于至少有一个工具扩展使用了SPIKE代码，同时这两个工具需要共享某些代码如异常处理和报告代码，因此采用不同的语言来实现相同的功能是不明智的。

12.8 小结

在可靠的文件模糊化测试的工具的支持下，客户端漏洞发掘工作变成了只是选择一个目标应用并且辅以耐心。不论你是选择编写自己的模糊器还是从其他的模糊器进行扩展，那么所花费的时间都是非常值得的。

第13章 文件格式模糊测试：Windows平台上的自动化测试

"发现那些伤害我们的人并阻止他们，这是我们国家的利益所在。"

——George W. Bush, Washington DC，2005年4月28日

在上一章中，我们讨论了在UNIX平台上实现文件格式模糊测试。现在我们要转向发现Windows应用程序中的文件格式漏洞。尽管所有的概念都是一致的，但我们还是要强调一些重要的不同点。首先，Windows编程在本质上都是进行图形化的工具设计，因此我们将脱离上一章所提到的命令行应用，为脚本程序创建一个独特的GUI应用。同时还要花时间来识别作为模糊测试目标的适当的文件格式，当工作在Windows环境时这是一个非常重要的决定，提供了针对给定文件类型对Windows中默认应用的严重依赖性。最后，我们为曾经提到的发现漏洞条件的挑战提供一种解决方法。

13.1 Windows文件格式漏洞

尽管文件格式漏洞也可以影响服务器，但它们更多的是影响客户端应用。文件格式漏洞的出现表明客户端漏洞的重要性正在不断地增加。网络管理者关注于资源以使协同网络免受网络层漏洞的攻击。同时，软件提供商也意识到了由服务器端漏洞所导致的威胁，这些努力联合起来减少了在流行应用程序和操作系统中的严重的服务器端漏洞数量，这些漏洞过去曾导致快速扩张的蠕虫并造成非常严重的损害。然而，对于客户端的情况却大不一样。客户端漏洞在过去的几年中一直在增长，并使它们成为了攻击的目标，如钓鱼式攻击和身份窃取。

文件格式漏洞给企业带来独特的风险。尽管这些漏洞并不会使它们自己变为快速增长的蠕虫或直接危及网络安全，但是它们在很多方面是难以被防护的。Internet网络从设计初衷上讲，是为了促进信息的共享。Web站点上充满了电影、图像、音乐以及文档，这使它变成了信息和娱乐的一个巨大的资源库。所有这些内容都要求文件公开的、免费的共享。我们从来没有将图片和电子表格这样的文件认为是恶意的，因为它们是不能自己执行的。然而正如前面所讨论的，当它们被一个漏洞应用所解释时就可能导致漏洞的发生。我们如何来避免此类威胁呢？网络管理者是否应当在防火墙阻止所有的内容呢？如果Web站点上只是基于文本的内容，那么它将会使人感到厌烦。我们真的想回到使用基于文本的浏览器如Lynx来浏览Web站点的时代吗？当然不是，但我们需要认识到由文件格式漏洞所带来的威胁。

Windows平台更加易于受到文件格式漏洞的影响，因为它具有用户友好的特性。文件类型是与默认将要处理它们的应用程序关联在一起的。这就允许一个用户通过简单的双击文件就可以观看电影或阅读文档。甚至没有必要知道何种应用程序能够提供特定的文件类型以查看或聆听它们。想象一下由应用程序中所发现的一个文件格式漏洞带来的风险，该漏洞默认运行在

Windows操作系统中。那么成百上千的终端用户将会立刻受到影响。下面列出了影响微软Windows的一些严重的文件格式漏洞。

影响微软Windows的严重的文件格式漏洞

MS04-028，JPEG处理（GDI+）中的缓冲区溢出，可以导致代码执行

2004年9月，微软发布了一个安全公告，详细描述了一个缓冲区溢出漏洞，该漏洞发生在当GDI+图形设备接口所提供的JPEG图像包含一个无效大小的内容时。JPEG文件中的内容的前缀是0xFFFE字节值，后面跟着针对后续内容的一个2字节大小。内容的最小有效大小是2字节，因为内容大小包含被大小值自身所使用的2个字节。我们发现一个0或1的大小值将由于一个整数下溢而导致一个堆溢出。因为GDI+库（gdiplus.dll）被多个应用程序所使用，因此该漏洞影响了大量的Windows应用程序。在该安全公告发布之后，很快就出现了公开的漏洞利用代码。

MS05-009，PNG处理中的漏洞，可以允许远程代码执行

许多软件提供商，包括微软都曾开发了包含缓冲区溢出漏洞的应用程序，这些漏洞发生在当渲染包含有大块tRNS的可移植网络图形（Portable Network Graphics,PNG）图像时，tRNS用于图像的透明化处理。该漏洞影响了Windows消息处理器和MSN消息处理器，并且导致微软阻止含有漏洞的客户端访问它们的即时消息网络，直到适当的补丁被应用为止。

MS06－001，图形渲染引擎中的漏洞，可以允许远程代码执行

在2005年的假日期间，有报道称Web站点对恶意的Windows元文件（Windows Meta File，WMF）文件进行攻击将会导致一个问题，即当使用Internet Explorer查看页面时将发生代码的执行。随后微软根据所发生漏洞的级别，在2006年初被迫发布了针对该问题的一个生命周期之外的补丁。该漏洞后来证明是由于一个设计错误而导致的。WMF文件包含记录以允许参数被传递给特定的Windows图形设备接口（Graphics Device Interface,GDI）库的调用。其中的一个调用Escape及其子命令SETABORTPROC，实际上允许任意可执行代码被调用。稍后将说明该漏洞的细节信息已经被秘密地卖到了4000美元[㊀]。

eBay网中的Excel漏洞拍卖

2005年12月8日，一个由自称为“fearwall”的黑客发起的拍卖在eBay网上进行，该拍卖提供了微软Excel中的一个漏洞的详细信息[㊁]。该拍卖引起了媒体的重点关注，并且很快被eBay网取消，删除该拍卖的原因是eBay网引用了一条禁止非法活动[㊂]的政策。

FileFuzz被创建用来将识别文件格式漏洞的过程实现自动化。FileFuzz要实现的目标包含三个方面。首先，应用程序应当是直观且用户友好的。其次，它应当既可以实现模糊化文件创建的自动化，也能够实现设计用来解释这些文件的应用程序执行过程的自动化。第三，它应当组合有调试功能，以确保已处理和未处理的异常都能被识别。为了实现这些目标，我们选用微软

㊀ http://www.securityfocus.com/brief/126

㊁ http://www.osvdb.org/blog/?p=71

㊂ http://www.theregister.co.uk/2005/12/10/ebay_pulls_excel_vulnerability_auction/

的.NET平台来设计该应用程序。这就允许我们使用C#来创建图形化的前端，而其他的一些后端构件如调试器仍然用C语言编写。图13-1显示了该应用程序GUI的一个快照。

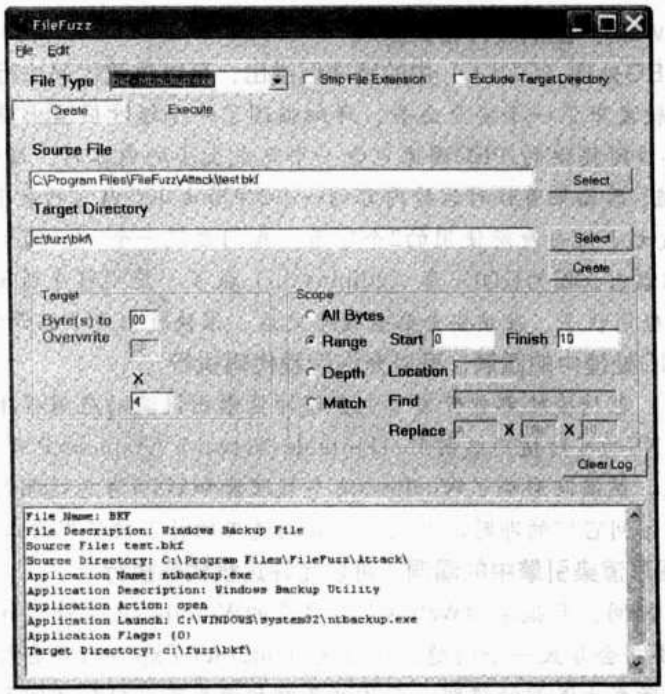


图13-1　FileFuzz界面

13.2　FileFuzz的特性

FileFuzz被设计为通过使用一种简单而有效的强有力漏洞发掘方法来识别文件格式漏洞。FileFuzz破坏正确格式化的文件，并将它们部署到设计用来解释这些文件的应用程序中，然后查看是否发生任何问题。这种方法可能不是非常好，但它可以正确地工作。当开发FileFuzz时，我们对它非常容易地发现文件格式中潜在的漏洞条件感到非常的惊讶。

FileFuzz的开发经历了3个不同的阶段。首先，它创建将要被模糊化的文件。它通过获得一个合法的用户提供的文件并基于所提供的目录，来进行有计划的连续变异，并保存结果文件的方法来实现此功能。其次，变异后的文件一个接一个的被部署到目标应用中。这些文件被重复部署，并且结果进程最终基于用户定义的一个超时而被销毁。最后，内嵌的调试功能监视进程以识别可能发生的已处理和未处理异常。当这些事件被识别时，它们被记录下来并报告给终端用户。下面对每个阶段进行更加详细的介绍。

13.2.1　创建文件

FileFuzz采用了一种强制性方法来进行模糊测试。这意味着我们需要具有以下功能，即可

以从一个有效的文件中进行读取，覆盖该文件的特定部分以及保存改变后的文件，这样它就可以被负责解释该文件的应用程序来读取。因为这个过程将被重复执行成百上千次，因此必须将它实现自动化。

FileFuzz允许使用4个单独的方法（所有字节，范围，深度和匹配）来进行文件的变异，这些方法可以划分为如下的类别：

- 二进制文件
- 宽度
- 所有字节
- 范围
- 深度
- ASCII文本文件
- 匹配

可以看到，FileFuzz可以处理二进制文件格式和ASCII文本文件格式。在二进制文件中，采用了两个单独的方法，名为宽度和深度。为了区分宽度和深度，我们将使用钻井取油来进行类比。如果你正在一个广阔的区域内寻找石油，那么你不可能随意就开始钻井。你必须要首先使用一些不同的技术来确定那些最有可能包含石油的位置。可能你会研究地图，分析岩石构造或者使用地面探测雷达。不管使用什么方法，一旦你发现这些感兴趣的位置，你就可以进行试探性的钻探以发现最佳的位置。

对于文件格式模糊测试，我们也采用了一种类似的方法。首先，我们使用宽度方法来发现感兴趣的位置，然后使用深度方法来确定是否找到了需要的内容。宽度方法意味着覆盖文件中的所有字节或者一个特定的区域。通过连续的将区域内的字节值改变为一个预定义的值来生成单独的文件，直到整个区域都被覆盖。一旦该操作完成，这些文件被一个接一个的部署以确定这些变化是否导致了任何异常。

有时候我们会比较幸运，不再需要进行其他操作。有些时候，异常结果是很有趣的，并且终端用户可以控制崩溃，因为被引入的变异值在寄存器中是显而易见的。但在大多数情况下，并不会这么简单。一个崩溃将会发生并且位置可能是有趣的，但如果终端用户拥有控制崩溃的任何方法，那么寄存器的值将不是清晰可见的。在这种情况下，就该转向使用深度方法了。一旦感兴趣的字节位置在文件中被识别，我们将关注这些位置并使用深度方法来为该位置试验所有可能的字节值。当我们查看所生成的崩溃时，就可以了解能够控制该崩溃的程度。如果不论提供什么字节值，异常发生的位置和寄存器值都保持一致的话，那么就表明没有对该异常的控制权。然而，如果崩溃发生在不同的位置或寄存器的值连续地改变，那么很显然我们对所产生的异常至少造成了一些影响，而这种影响基于对文件进行变异时所使用的值。

FileFuzz也可以处理ASCII文本文件。它是采用如下的方法来进行处理的，即允许终端用户首先选择一个标识将要被覆盖位置的字符串，然后请求三个单独的输入来确定用来变异文件的输入。终端用户必须提供一个字符串值及其长度，以及它将要被增加的次数。下面来看一个例子。在通常情况下，ASCII文本文件，如*.ini，都包含如下格式的名－值对：

```
name = vlue
```

典型的，我们想要将值覆盖。假定我们要用连续的10个A字符来覆盖值。在这种情况下，我们首先将所发现的值设置为＝字符，它标识了值的开始。然后将替换值设置为AX10X10。这将生成10个变异文件，即当一个＝字符被发现时为每个实例使用覆盖值。所生成的10个文件将在值的位置包含从10到100个A字符。

13.2.2 应用程序执行

一旦模糊文件被创建，我们需要将它们部署到应用程序中。例如，如果我们已经将*.doc文件变异，想要将它们部署到微软的Word程序中。FileFuzz使用来自于Windows API的CreateProcess()函数来实现此功能，因此我们可以使用FileFuzz来部署应用程序，只要能够确定相同的应用程序如何从命令行部署，并且包含可能传递给应用程序的必要标志。我们将在本章后面详细介绍如何识别这些信息。

在文件格式模糊测试中，我们需要将同一个应用程序部署成百上千次。如果我们让所有前面已经执行过的进程开始运行，那么将会很快用完可用内存。因此，执行阶段并没有完成，直到在一个预定义的时间间隔之后该进程也被销毁为止。我们允许终端用户对此进行控制，方法是通过在执行标签下包含一个毫秒字段，以标识在需要时被强制销毁之前进程所允许运行的总次数。

13.2.3 异常检测

正如在上一章中所提到的，检测部分是模糊测试的一个核心构件。如果整夜运行一个模糊器，当醒来时发现真正可以使应用程序崩溃，但是你却不知道是这1000个输入中的哪个导致了崩溃的发生，那么这样有什么好处呢？在开始测试之前你是不会得到答案的。文件格式模糊测试提供了许多检测异常的选项，但有一个是例外的。对初学者而言，可以从较低的技术层面入手，只是简单的观察模糊测试过程以识别错误窗口或应用程序和系统的崩溃。如果你喜欢“*Watching Paint Dry*（看着涂料风干）”，那么就可以采用这种方法。如果你倾向于将它提高一步，那么可以检查日志文件以确定是否发生了一个问题。这些日志文件既包括应用程序的日志文件，也包括系统日志文件，如被Windows事件查看器所维护的那些文件。该方法存在的问题是输入（模糊文件）和输出（日志事件）没有相互关联。将二者进行关联的唯一方法就是使用时间戳，但它并不是十分完美。

对于大多数类型的模糊测试而言，识别异常的最好方法是使用一个调试器。调试器的优点是它既可以识别出已处理的异常，也能够识别出未处理的异常。Windows具有强大的异常处理功能，并且通常可以从它所读取的模糊文件中进行恢复。然而，识别这些异常情形是非常重要的，因为在文件异常处的一个小的改变就可以结束创建一个不可恢复或可利用的条件。

调试器在文件格式模糊测试过程中的使用，并不像它在其他类型模糊测试中那样直接。在这种情况下，我们无法手工将一个调试器关联到目标应用程序并使模糊器运行。这是因为模糊器要经常部署并销毁目标应用程序。因此，它同样也会销毁调试器。为此，我们需要利用一个

调试API并且在模糊器中直接创建调试器。采用这种方法，模糊器就可以执行目标应用，并且自己关联到一个调试器，然后销毁目标应用。因此，模糊器在每次目标应用被部署时来监视异常。

对于FileFuzz而言，我们创建了crash.exe，它实际上是一个被GUI应用执行然后部署目标应用的单机版调试器。它是一个完全的单机命令行应用程序，同时由于FileFuzz是一个开源项目，因此你可以在你自己的模糊测试项目中免费使用crash.exe。

13.2.4 保存的审核

在FileFuzz中，可以通过手工方式来进行审核，方法是选择要进行模糊测试的目标应用并确定模糊文件应当如何生成和保存。另外一种方法是，可以使用以前所保存的审核来填充所有需要的字段以立即启动模糊测试。带有一些已保存的审核信息的应用程序，可以通过主界面中的文件类型下拉菜单进行访问。然而，终端用户也可以创建他们自己的已保存的审核并将它们添加到下拉菜单中，而不用重新编译应用程序。这可能是由于菜单是在运行时通过解析target.xml文件而动态生成的。一个单独审核的结构如下所示：

```
<test>
   <name>jpg - iexplore.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>
```

包含一个动态生成的下拉菜单是特意做出的一个设计决定。尽管FileFuzz是一个开源应用，但它假定大多数用户不具有编程经验，或通过额外编码来扩展功能的兴趣。使用动态菜单的方法就允许大多数终端用户以相对更加友好的方式来扩展功能。XML文件的结构说明了很重要的一点：尽可能使你的应用程序更加直观和用户友好。尽管文档是开发过程中的一个关键构件，但不要期望用户首先去查看文档。用户所期望的是直观的应用程序，可以简单直接的进行使用。

请看前面所示的XML文件。描述性的XML标签是自解释的。终端用户可以通过简单的添加并定制一个额外的<test>块来添加一个审核。尽管创建直观的应用程序并不是忽略文档的一个合理理由，但它是一个重要的设计概念。现代编程语言允许实现用户友好的应用程序，因为

它们负责控制底层代码，如网络通信和图形显示。创建一个为终端用户提供良好体验的应用程序可以节省大量的时间。这是因为假如你正在设计一个安全的应用程序，但这并不意味着你需要具备博士学位来操作该程序。

13.3 必要的背景知识

Windows和UNIX平台上的文件格式模糊测试具有相同的基础，但是Windows环境中默认应用的重要性可以增加该类漏洞所造成的风险的级别。因此在创建模糊器之前，我们需要花费一些时间来研究一下该问题，以更好地理解如何识别具有高风险的目标应用。

识别目标应用

微软Windows允许用户将默认的应用程序指定为特定的文件类型。这就增强了操作系统的可用性，因为当一个文件被双击时，它允许应用程序自动开始运行。当为目标应用识别文件格式时，这是应当被牢记的很重要的一点。如果在应用程序中发现了一个缓冲区溢出，但该应用程序不会被用做查看一个特定的文件类型，那么此时的风险比较小。但如果在应用程序中发现了一个相同的漏洞，但该应用默认要打开一个常用的交易文件，那么此时就具有很大的风险。例如，考虑JPEG图像格式中的一个溢出。像所有的图形格式一样，存在许多可用的应用程序可以来展现图像，但在一个给定的操作系统中默认只有一个与该文件类型相关联的应用程序。在Windows XP中，默认的JPEG图像查看器是Windows Picture和Fax Viewer。因此，在Windows Picture和Fax Viewer中发现溢出所带来的风险要大于使用从Download.com下载的免费图像查看器发现相同溢出的风险。为什么呢？如果一个默认的Windows应用程序被发现含有漏洞，那么立刻就会有数百万的机器受到该漏洞的影响。

Windows Explorer

我们如何确定在Windows环境中哪些应用程序将展现一个特定的文件类型呢？一个简单的方法是双击文件，然后查看是哪个应用程序被启动。这对于快速检查是很好的，但它不能帮助我们确定被执行以启动应用程序的准确的指令。但进行模糊测试时，这些指令对我们来说是非常重要的，因为我们需要一种方法来使目标应用自动化连续运行。当你对几千个文件进行模糊测试时，只是快速双击是不实际的。

Windows Explorer是识别与文件类型相关联的应用程序的一种快速而简便的方法，并且可以识别用来启动应用程序的命令行参数。让我们使用Explorer来证明JPEG文件是与Windows Picture和Fax Viewer相关联的。更重要的是，让我们确定如何能够在FileFuzz中重复启动模糊化的JPEG文件以识别漏洞。

第一步需要从Windows Explorer的菜单中选择Tools[el]文件夹选项，然后，选择文件类型标签。所显示的界面如图13-2所示。

该界面本身包含有大量的信息。拖动已注册文件类型列表框中的滑块，可以直观地看到与特定应用程序相关联的所有文件扩展名。这些文件类型可以成为很好的模糊测试目标应用，因为在默认应用中所识别的漏洞可以通过下面的方法被利用，即将文件发送给被攻击者然后确认

他们双击该文件。尽管这是很重要的，但垃圾邮件证明终端用户已经习惯于点击这些文件，因此它代表了一种合理利用漏洞的场景。



图13-2 Windows Explorer的文件夹选项对话框

在这时，根据所列出的标签，我们已经知道了与JPEG文件类型相关联的应用程序，但是我们不知道操作系统实际上是如何启动该应用程序的。幸运的是，我们可以通过较简单的操作就可以得到这些信息。

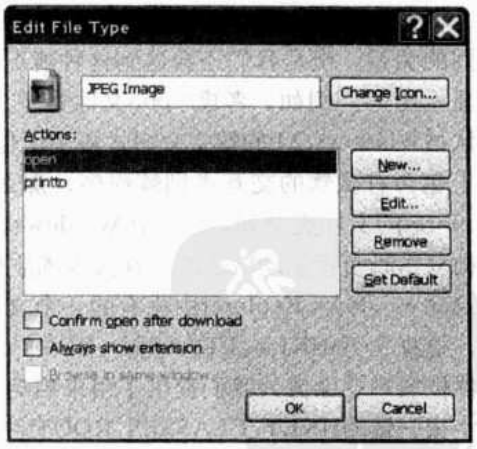


图13-3 Windows Explorer的编辑文件类型对话框

当处理文件类型时，Windows包含了动作的概念。动作允许在给定唯一的命令行选项的情况下，采用一种不同的方法或使用不同的应用程序来打开文件。我们的目的是想要知道Windows是如何打开JPEG文件的，因此我们关注于打开动作。如图13-3所示，选中打开动作并点击编辑按钮将会得到问题的答案。

Windows最终揭晓了我们所寻找的秘密。在用来执行动作域的应用程序中，如图13-4所示，我们看到Windows Picture和Fax Viewer并不是一个可执行的应用程序。实际上，它是使用rundll32.exe来启动的一个动态链接库（dynamic-

link library,DLL)。在Windows Picture和Fax Viewer中启动一个图像文件的完整命令行参数如下所示。

```
rundll32.exe C:/WINDOWS/sysem32/shimgvw.dll,ImageView_Fullscreen %1
```

不仅Windows Picture和Fax Viewer出乎我们的意料不是一个可执行的应用，同时我们可以看到Windows也期望提供ImageView_Fullscreen参数。如果你在命令行提示符下运行该行代码，并用一个合法的JPEG文件名来替换%1，那么你就会看到该文件象预期的那样在Windows Picture和Fax Viewer中被展现。这是一个关键的概念。如果我们能够确定在一个给定的应用程序中展现文件的适当的命令行参数的话，那么就可以使用FileFuzz来测试漏洞。我们现在将同一个命令行拷贝到FileFuzz中执行标签的应用程序和参数字段中。唯一需要做的就是将代表目标文件的%1变为一个{0}，因为这是FIleFuzz所期望的格式。有一个建议：即当遇到诸如命令行中的空格和引号时，Windows过于挑剔。因此要保证准确地复制命令，以避免以后繁琐的调试工作。

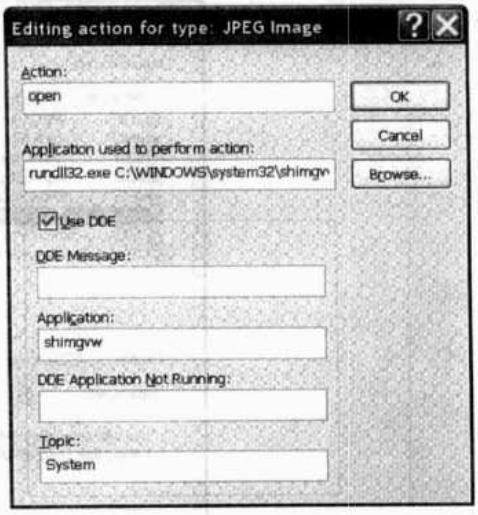


图13-4　Windows Explorer的编辑动作类型对话框

Windows注册表

尽管Windows Explorer在90%的情况下都可以展示特定文件类型及与之相关联的应用程序之间的关系，但是我们还会遇到这种情况，即关联关系是存在的但在Windows Explorer中并没有显示出来。例如，考虑*.cbo文件类型。CBO文件被微软的交互式训练应用程序所使用，并默认被Windows XP的特定分布式应用所包含，例如许多Dell机器上所带有的应用。如果你的机器上包含有微软的交互式训练程序，那么你将会注意到CBO文件类型并没有包含在Windows Explorer的文件类型列表中，在Windows Explorer中，一个CBO文件显示为一个铅笔图标，当它被双击时负责启动微软的交互式训练应用程序。这表明什么呢？我们如何来确定命令行参数，以启动Windows Explorer中没有的一个文件类型呢？为此，我们需要使用Windows注册表。首先，检查一下\HKEY_CLASSED_ROOT\.XXX注册表键的默认字段的值，其中xxx是相关联的文件扩展名。该值为我们提供了用来启动文件类型的应用程序名。接着，确定与同一个描述性名字相一致的HKEY_CLASSES_ROOT\注册表键。在...\shell\open\command键中，你将会发现用来启动与该文件扩展名相关联的应用程序的命令行参数的详细信息。

13.4　FileFuzz的开发

现在我们已经初步了解了在Windows平台上进行文件格式模糊测试的一些独特方面，下面

就可以开始创建一个模糊器了。我们将FileFuzz的设计过程进行详细地分解与分析，然后，通过使用FileFuzz来识别微软安全公告MS04－028中详细描述的被广泛宣传的JPEG GDI＋漏洞，来对FileFuzz的开发做出总结。

13.4.1 开发方法

对于创建FileFuzz而言，我们想要开发一个用户友好的且图形化的应用程序，该程序允许终端用户不需要学习一系列的命令行参数知识就可以执行模糊测试。即尽可能的开发一个简单易用的、直观的应用程序。我们的目标还包括要开发一个能够特别适应Windows平台上文件格式模糊测试特点的工具，因此，跨平台功能并不是一个必需的特性。

13.4.2 开发语言的选择

针对给定的目标，我们再次选择了.Net作为开发平台。这使得我们可以非常容易地创建一个GUI前端，以节省出时间集中关注于项目的功能方面。GUI界面和所有的文件创建功能都是使用C＃语言实现的。对于调试功能的实现，我们转向使用了C语言，因为它允许我们更加简便和直接地与Windows API进行交互。.Net平台能够适应此设计决策，因为它允许项目包含多个编程语言，只要这些语言与.Net框架相兼容即可。

13.4.3 设计

我们已经对如何来设计FileFuzz进行了足够的讨论。现在可以深入到具体实现的一些特定部分了。在这里对所有的代码进行描述是不切实际的，但我们要强调一些更加重要的代码段。为了全面理解FileFuzz是如何被设计的，我们建议读者从www.fuzzing.org处下载源代码进行分析。

创建文件

FileFuzz需要能够适应任何Windows文件格式，因此对于二进制文件和ASCII文本文件需要采用不同的方法。Read.cs文件包含阅读有效文件的所有代码，而Write.cs文件则处理模糊文件的创建。

从源文件中读取

FileFuzz采用了一种强制性方法来进行模糊测试，因此我们从读取已知的源文件开始。该合理文件中的数据被读取并保存以用于随后创建模糊文件时进行变异。幸运的是，.Net框架相对来说简化了诸如从文件读取这类任务。我们在读取并创建二进制文件和ASCII文本文件时采用了不同的方法。为了实现此目的，我们利用BinaryReader类来读取二进制文件并将数据保存到一个字节数组中。读取ASCII文本文件非常类似于读取二进制文件，只是使用的是StreamReader类。另外，我们将结果保存到一个字符串变量，而不是一个字节数组中。Read类的构造函数如下所示。

```
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将用于支持所读取的二进制文件的字节数组，而sourceString将被用做保存一个ASCII文本文件的内容。

写入模糊文件

现在文件已经被读取了，我们需要对其进行变异并保存结果文件，以在目标应用程序中启动它们。正如所提到的，FileFuzz基于文件变异所采用的方法将文件的创建划分为如下4种类型：

- 所有字节
- 范围
- 深度
- 匹配

我们在一个单独的Write类中对所有不同的方法进行处理，但是对构造器进行重载以处理每一个不同的场景。对于二进制文件类型，我们使用BinaryWrite类将新字节数组写入一个文件中，该文件将在执行阶段用来对目标应用进行模糊测试。另一方面，对于ASCII文本文件，利用StreamWrite类将字符串变量写到磁盘上。

应用程序执行

负责启动执行目标应用程序的代码存在于Main.cs文件中，如下一个代码段所示。然而，当你查看该段代码时，会发现它相对来说很简单，这是因为该段代码本身实际上并不负责启动目标应用程序；而它实际上是负责启动内嵌的调试器，该调试器接着再处理目标应用程序的执行。稍后将对crash.exe调试器进行详细的讨论。

首先，对Process类的一个新的实例进行初始化。在executeApp()函数中，我们利用一个循环启动前面所创建的各个模糊文件。在循环的每次执行过程中，为要创建的进程设置属性，包括要执行的进程的名字。正如所提到的，该名字将会一直是crash.exe而不论什么被模糊化，因为命令行应用程序crash.exe接着将要启动目标应用。这时，控制权被转交给crash.exe，并且结果最终由crash.exe通过标准输出和标准错误来返回，并显示在rtbLog超文本框中，该文本框是FileFuzz的主输出窗口。

```
Process proc = new Process();

public Execute(int startFileInput, int finishFileInput, string targetDirectoryInput,
string fileExtensionInput, int applicationTimerInput, string executeAppNameInput,
string executeAppArgsInput)
```

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书藉杂志全免 KEY 提供,，完全无加密。

```
{
   startFile = startFileInput;
   finishFile = finishFileInput;
   targetDirectory = targetDirectoryInput;
   fileExtension = fileExtensionInput;
   applicationTimer = applicationTimerInput;
   executeAppName = executeAppNameInput;
   executeAppArgs = executeAppArgsInput;
   procCount = startFile;
}

public void executeApp()
{
   bool exceptionFound = false;

   //Initialize progress bar
   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();
      //Update progress bar
      if (this.pbrUpdate != null)
      {
         this.pbrUpdate(procCount);
      }
      //Update counter
      if (this.tbxUpdate != null)
      {
         this.tbxUpdate(procCount);
      }
      proc.WaitForExit();

      //Write std output to rich text box log
      if (this.rtbLog != null && (proc.ExitCode == -1 || proc.ExitCode == 1))
      {
         this.rtbLog(proc.StandardOutput.ReadToEnd());
         this.rtbLog(proc.StandardError.ReadToEnd());
         exceptionFound = true;
      }
      procCount++;
   }
```

```
    //Clear the progress bar
    if (this.pbrStart != null)
    {
       this.pbrStart(0, 0);
    }
    //Clear the counter
    if (this.tbxUpdate != null)
    {
       this.tbxUpdate(0);
    }
    if (exceptionFound == false)
       this.rtbLog("No excpetions found\n\n");
    exceptionFound = false;
}
```

异常检测

正如所提到的，FileFuzz包括以crash.exe形式存在的调试功能，它是用C语言编写的一个单机调试器，利用了内嵌在Windows API中的调试功能。同时它也利用了帮助解析代码的开源库libdasm。从列出的下一个代码段中可以看到，首先进行了一个检查以确保为crash.exe至少传递了3个参数。FileFuzz所使用的这些参数是进行模糊测试的应用程序的名字及其路径、在强制销毁目标应用程序之前的等待时间以及额外增加的一个命令行参数，即将被解释的模糊文件的名字。在该检查之后，将等待时间值从一个字符串类型转换为一个整型，完整的命令行参数被创建并保存到一个字符数组中。然后，使用CreateProcess命令并设置DEBUG_PROCESS标志来启动目标应用程序。

```
if (argc < 4)
{
   fprintf(stderr, "[!] Usage: crash <path to app> <milliseconds> <arg1>
     [arg2 arg3 argn]\n\n");
   return -1;
}

// convert wait time from string to integer.
if ((wait_time = atoi(argv[2])) == 0)
{
   fprintf(stderr, "[!] Milliseconds argument unrecognized: %s\n\n", argv[2]);
   return -1;
}

// create the command line string for the call to CreateProcess().
strcpy(command_line, argv[1]);

for (i = 3; i < argc; i++)
{
   strcat(command_line, " ");
   strcat(command_line, argv[i]);
}

//
// launch the target process.
```

```
//

ret = CreateProcess(NULL,      // target file name.
  command_line,                // command line options.
  NULL,                        // process attributes.
  NULL,                        // thread attributes.
  FALSE,                       // handles are not inherited.
  DEBUG_PROCESS,               // debug the target process and all spawned children.
  NULL,                        // use our current environment.
  NULL,                        // use our current working directory.
  &si,                         // pointer to STARTUPINFO structure.
  &pi);                        // pointer to PROCESS_INFORMATION structure.

   printf("[*] %s\n", GetCommandLine());  //Print the command line
if (!ret)
{
   fprintf(stderr, "[!] CreateProcess() failed: %d\n\n", GetLastError());
   return -1;
}
```

这时，crash.exe就可以监视并记录异常了。在下一个代码片段中，我们可以看到只要超时值没有过期，我们就可以一直监视调试事件。当遇到一个异常时，我们获得存在问题的线程的句柄并且确定所发生异常的类型。使用一个case语句来判断我们所感兴趣的三种类型的异常：内存访问违规，被0除的错误以及栈溢出。然后输出相关的调试信息以帮助终端用户确定该异常是否值得进行深入分析。通过利用libdasm库，我们得到了发生异常的位置，恶意的操作码以及程序崩溃时寄存器的值。

```
while (GetTickCount() - start_time < wait_time)
{
   if (WaitForDebugEvent(&dbg, 100))
   {
      // we are only interested in debug events.
      if (dbg.dwDebugEventCode != EXCEPTION_DEBUG_EVENT)
      {
         ContinueDebugEvent(dbg.dwProcessId, dbg.dwThreadId, DBG_CONTINUE);
         continue;
      }

      // get a handle to the offending thread.
      if ((thread = OpenThread(THREAD_ALL_ACCESS, FALSE, dbg.dwThreadId)) == NULL)
      {
         fprintf(stderr, "[!] OpenThread() failed: %d\n\n", GetLastError());
         return -1;
      }

      // get the context of the offending thread.
      context.ContextFlags = CONTEXT_FULL;

      if (GetThreadContext(thread, &context) == 0)
      {
         fprintf(stderr, "[!] GetThreadContext() failed: %d\n\n", GetLastError());
```

```
            return -1;
         }
         // examine the exception code.
         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);
         }

         // if an exception occurred, print more information.
         if (exception)
         {
            // open a handle to the target process.
            if ((process = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dbg.dwProcessId)) ==
NULL)
            {
               fprintf(stderr, "[!] OpenProcess() failed: %d\n\n", GetLastError());
               return -1;
            }

            // grab some memory at EIP for disassembly.
            ReadProcessMemory(process, (void *)context.Eip, &inst_buf, 32, NULL);

            // decode the instruction into a string.
            get_instruction(&inst, inst_buf, MODE_32);
            get_instruction_string(&inst, FORMAT_INTEL, 0,
inst_string,sizeof(inst_string));

            // print the exception to screen.
            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 ESP:%08x EBP:%08x\n\n", context.Esi,
context.Edi,
              context.Esp, context.Ebp);
            return 1;
         }

      }
   }
```

被crash.exe所识别的异常的细节信息被返回到GUI显示给终端用户。我们希望该信息将为终端用户提供一个快速的可视化的队列以帮助识别重要的崩溃。需要进行深入研究的崩溃包括发生在恶意指令中的异常，这时寄存器包括用户控制的或用户影响的输入，并且该异常将允许shellcode获得代码执行流的控制权。

13.5 实例研究

现在我们已经开发了一个文件格式模糊器，我们将使用它来测试一个已知的漏洞以验证我们的设计。我们对文件格式漏洞的兴趣主要来自于微软所发布的安全公告MS04-028-"Buffer Overrun in JPEG Processing (GDI+) Could Allow Code Execution"（"JPEG处理中的缓冲区溢出可能导致代码执行"）[⊖]。该公告引起了广泛的关注，因为它清晰地表明了客户端漏洞的破坏性是多么的大。这里有一个在许多流行的客户端应用中可以利用的缓冲区溢出，它被默认安装并且影响了大量的用户。其结果是公开利用漏洞的迅速出现，尽管依赖于某些社会工程，但这使它们自己成为了攻击目标，以及不加选择的钓鱼式攻击和身份窃取。

漏洞本身存在于gdiplus.dll库中，该库被许多应用程序所使用，包括微软的Office，Internet Explorer和Windows Explorer。JPEG格式允许内容被嵌入到图像自身中。内容的前面是0xFFFE字节序列，后面跟着一个16位的字值，该值表明了内容的总大小。该大小包括为大小所使用的两个字节和以内容自身作为结尾的头。

FileFuzz能够识别这个漏洞吗？让我们来探查一下。我们将使用一个合理的图像文件，然后手工的或使用一个图像编辑器来添加一个注释。如果你选择为自己来复制这个例子，那么要确保你正在使用Windows的一个漏洞版本。我们的结果是使用Windows XP SP1来生成的。当创建测试文件时，我们将使用一个非常简单的图像，在这个例子中是一个1 x 1的白色像素。为什么用这么简单的图像呢？正如所提到的，强制性模糊测试的效率较低。我们想要关注于图像头，而不是图像本身，因此我们坚持使用一个几乎是不存在的图像。作为结果，我们以一个大小只有631字节的文件作为结尾，因此就可以在一个合理的时间帧内对所有的字节进行强制性测试。在添加了"fuzz"注释之后，当我们使用一个十六进制编辑器查看该文件时，就可以看到如下的字节序列。

```
0000009eh: FF FE 00 06 66 75 7A 7A                    ; ÿþ..fuzz

Breakdown:
FF FE               Comment preface
00 06               Length of comments in bytes
66 75 7A 7A         ASCII value of 'fuzz'
```

在能够开始模糊测试之前，我们想要知道在默认情况下是哪个Windows XP应用程序负责展现JPEG图像。幸运的是，我们在本章的前面已经确定了是Windows Picture和Fax Viewer负责完成此任务，它们使用下面的命令行参数来启动JPEG图像（如图13-5所示）。

⊖ http://www.microsoft.com/technet/security/Bulletin/MS04-028.mspx

```
rundll32.exe C:\WINDOWS\system32\shimgvw.dll,ImageView_Fullscreen %1
```

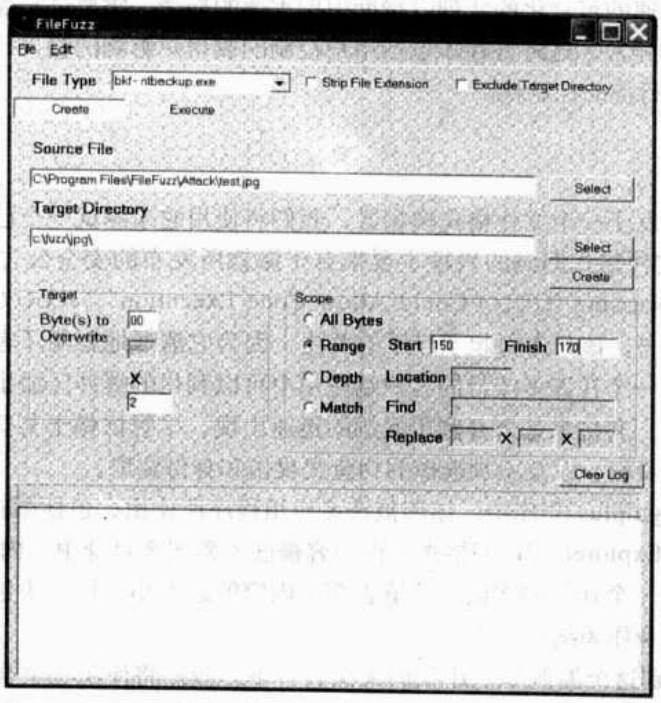


图13-5　FileFuzz针对JPEG模糊测试的Create选项卡设置

这时，我们就可以启动FileFuzz并开始模糊测试过程。FileFuzz针对JPEG文件有一个内嵌的审核器，它可以通过文件类型下拉菜单进行访问。但为了说明FileFuzz的功能特性，我们将从头开始介绍。

我们从Create选项卡开始，设定适当的选项以生成一系列变化了的JPEG文件，而生成的基础是一个合理的并内嵌有前面生成的注释的JPEG文件。我们将Create选项卡中的所有选项设置为如下的值：

- 源文件。C:\Program Files\FileFuzz\Attack\test.jpg。是一个合理的JPEG文件。
- 目标目录。C:\fuzz\jpg\。目标目录是所生成的模糊文件将要被保存到的目录。
- 要覆盖的字节。00 x 2。根据漏洞的详细信息[⊖]，如果我们将漏洞的长度值设置为0x00或0x01，那么就会发生溢出。注释的长度为0或1个字节是不可能的，因为两个字节的大小被包含在整个长度中。因此我们使用一个字值0x0000来模糊化该文件，以期望通过覆盖注释的大小值来触发溢出。
- 范围。范围＝150－170。在我们所创建的测试文件中，大小值起始于160字节.为了安全起见，我们将150到170字节之间的范围进行模糊化。

⊖ http://www.securityfocus.com/archive/1/375204

最终的设置如图13.5所示。

一旦设定好所有的选项，就可以点击Create按钮来生成文件。下面来研究一下Execute选项卡。在该选项卡中我们需要告诉FileFuzz如何来启动Windows Picture和Fax Viewer。Execute选项卡中的设置值如下所示：

- 应用程序。Rundll32.exe。由于Windows Picture和Fax Viewer实际上是一个DLL，因此我们所使用的应用程序是run32.exe3，它将用来启动DLL文件。
- 参数。C:\WINDOWS\system32\shimgvw.dll,ImageView_Fullscreen{0}。所传递的参数包括Windows Picture和Fax Viewer的完整的路径（shimgvw.dll），ImageView_Fullscreen参数，{0}是针对将要打开的模糊文件的名字的一个占位符。
- 起始文件。150.我们所创建的第一个文件。
- 结束文件。170.我们所创建的最后一个文件。
- 毫秒数。2000。Windows Picture和Fax Viewer被强制关闭之前所允许运行的时间长度。

Execute选项卡的设置情况如图13-6所示。

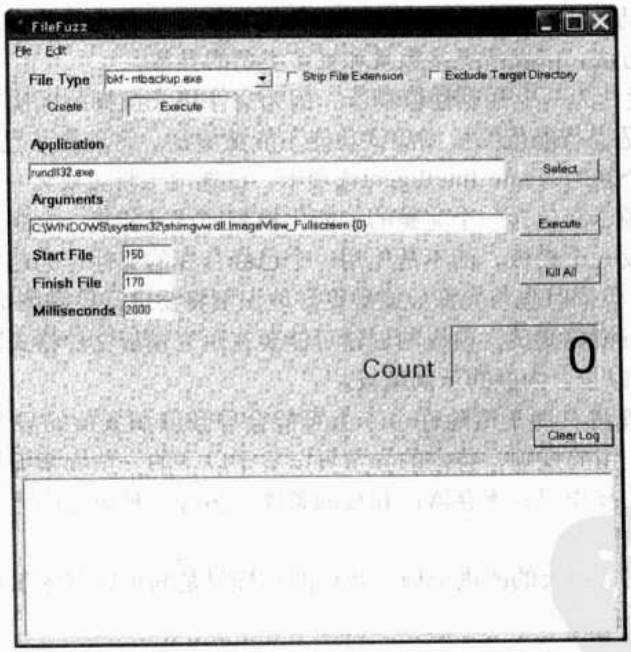


图13-6　针对JPEG模糊测试的FileFuzz中 Execute选项卡的设置

这时，我们就准备就绪了。当点击Execute按钮时，我们可以看到Windows Picture和Fax Viewer被重复地启动和退出。这种情况将会发生21次，因为这里有21个模糊文件被打开和解释。当执行结束时，我们看到FileFuzz已经发现了一些异常。然而，我们所感兴趣的异常发生在160.jpg文件被启动时，如下面的示例输出所示。该异常之所引起我们的兴趣，是因为160字节是JPEG注释大小的起始位置，并且文件160.jpg已经用0x0000将初始值覆盖。

```
[*] "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
```

13.6　益处和改进的余地

FileFuzz为采用一种强制性方法来对文件格式进行模糊测试提供了基本的功能。使用其GUI界面和内嵌的调试功能，就可以通过使用一个已知的文件进行快速的审核。也就是说，存在着很大的改进余地。

对于初学者而言，可以开发更加全面的审核功能。正如所指出的，每次只能执行一个审核。例如，当对一个二进制文件类型执行宽度方法时，每一次只能使用一个单一的字节值（如0xFFFFFFFF）对字节范围执行一次单一的扫描。如果另外一个值被使用，那么进程就必须使用新的值来重新运行。一个更加全面的审核功能将允许选择多个值，或将要被测试的值的范围。可能有一些智能化的功能将被增加，一旦字节位置被识别，它首先执行一个宽度方法，然后自动转向执行一个深度方法，以发现更多数量和更多类型的异常。

FileFuzz有意被设计为一个强制性模糊器，因为文件格式模糊测试使其成为了更加简单的方法。然而，这并不是说智能化的模糊测试功能不能被增加。可能除了已经存在的Create选项卡以外，还会增加一个新的Create-Intelligent选项卡，该选项卡将会成为一个Create-Brute Force选项卡。这个新的选项卡将包含一个完整的智能化模糊测试功能集，它将需要终端用户来为特定文件类型的结构开发一个模板，而不是使用一个已经存在的文件作为起点。该方法需要用户进行更多的前端工作，但同时也将允许他或她更好地识别将要进行模糊测试的文件的特定区域，并确定如何对它们进行模糊测试。模板的创建可能要在研究模板文件格式的规范文档之后来进行，但是FileFuzz也可以包含内嵌的示例模板。

智能化的异常处理将有助于清除许多不太可能会导致漏洞条件的异常。通过嵌入在分析crash.exe的输出时所应用的规则，特定的异常可以基于以下的一些因素而被忽略，这些因素包括崩溃的内存位置处的操作码，寄存器值和栈的条件。另外一种可选的方法是，与其忽略特定的结果，不如将它们加以强调。

简而言之，这里有着很大的改进余地。我们的目标只是简单地使该工具可以运转，而剩下的完善工作就要靠你自己了。

13.7　小结

文件格式漏洞在过去的几年中给微软带来了很多的麻烦。无论该漏洞是出现在多媒体文件还是文档文件中，它们都是大量存在的并且被攻击者所利用。这并不是说微软是唯一一个与该类漏洞进行斗争的提供商，只是表明微软一直是错误发掘者所感兴趣的漏洞发掘目标。我们希望随着近来对文件格式漏洞的关注程度的提高，软件提供商将会在他们的开发周期中加入模糊测试，以在生成产品之前捕获这些漏洞。

第14章 网络协议模糊测试

"我拥有一个木材公司吗？这对我来说是一个新消息。是否需要一些木材？"

——George W. Bush，第二届总统论坛，St.Louis,MO,2004年10月8日

模糊测试是威斯康星大学为UNIX工具中的命令行setuid引入随机参数值时提出的。尽管它最初是用于命令行参数，但模糊测试这个术语现在通常被认为是应用于网络协议领域，这是有合理理由的。网络协议模糊测试对安全研究者来说是最感兴趣的模糊测试类型，因为所发现的漏洞通常具有最高级别的危险程度。不需要有效的身份认证就可以到达，或者不需要目标用户的任何交互就可以利用的一个远程的可利用漏洞，是所发现漏洞的一个缩影，如果你同它们进行交互那将是最好的。

客户端漏洞，例如影响微软的Internet Explorer的漏洞，通过在创建被恶意代码感染并控制的与Internet相连的计算机网络时被利用。贪婪的渔夫撒下一张大网，期望能够捕获尽可能多的鱼。在大多数时候，这些被捕获的鱼是宽度网络上所连接的个人桌面计算机。一个网络daemon程序中的服务器端漏洞对于创建一个被恶意代码感染并控制的与Internet相连的计算机网络同样是非常有用的，但是以这种方式来利用这些漏洞是对潜在资源的浪费。从一个攻击者的角度来看，拥有像后端数据库或企业级Web服务器这样的软件，那么就给数据的窃取提供了很大的机会，就像从一个可信的平台上执行进一步的攻击一样。本章中，我们将介绍网络协议模糊测试，并提出一些由这种流行的模糊测试类型所带来的独特的特性和挑战。同时，我们还将研究一些网络协议漏洞。

14.1 什么是网络协议模糊测试

相对于其他类型的模糊测试而言，网络协议模糊测试要求识别攻击的界面，变异或生成包含错误的模糊值，然后将这些模糊值传递给一个目标应用，并监视目标应用以发现错误。非常简单，如果你的模糊器通过某种形式的socket与其目标应用通信，那么该模糊器就是一个网络协议模糊器。

基于网络的模糊测试中的socket通信构件带来了一个有趣的变化，因为它对测试的吞吐量造成了一个瓶颈。将文件格式模糊测试、命令行参数模糊测试以及环境变量模糊测试的传输速度与网络协议模糊测试的传输速度进行比较，那么就像是一辆阿斯顿·马丁DB9和谢奥·米托罗进行比赛一样。

现有的网络协议模糊器倾向于采用两种风格。其中一些采用了通用体系结构以能够对不同的协议进行模糊测试。这类工具的代表如SPIKE[⊖]和ProtoFuzz，我们在第16章"网络协议模糊

⊖ http://www.immunitysec.com/resources-freesoftware.shtml

测试：Windows平台上的自动化测试”中从头创建的一个模糊器也是属于这种类型。SPIKE是最广为人知的一个模糊器，在下一章中将对它进行更加详细的介绍。另外一种类型的网络协议模糊器包括哪些被设计为面向特定协议的模糊器。这种类型模糊器的例子包括ircfuzz㊀，dhcpfuzz㊁以及Infigo FTPStress模糊器㊂。直接从它们的名字你就可以猜出每个所提到的模糊器被设计为面向哪种协议。特定于协议的模糊器通常面向于较小的目标脚本和应用程序，但其体系结构开发的工作量却较大。在第21章“模糊测试的体系结构”中，我们将讨论相对于特定的单机工具而言，创建和使用模糊测试体系结构的价值。下面，让我们来看一些网络协议模糊测试目标的例子。

微软感染了一个病毒

对于世界上大多数非妄想狂和精神分裂症的人来说，新千年的到来是一个值得庆祝的时刻。我们享受了历史上最大的新年庆祝宴会，从2000年问题的惊恐中恢复了过来，并且看到世界并没有灭亡。但对于微软来说却很不幸，自从新千年到来后的几年中，在大多数主流的微软产品中发现了大量服务器端漏洞。正如所证明的，多数问题将带来漏洞利用代码的快速开发和公开发布。另外，其中的一些漏洞将被快速扩张的恶意代码所利用，从而导致给世界范围内的公司带来实质性的金融损失，同时也给世界各地的系统管理者敲响了警钟。

让我们来看一些著名的影响微软的蠕虫以及在世界范围内被广泛传播的漏洞。由这些蠕虫所造成的名誉上的损害是促使微软于2002年开始建立可信计算组织的直接原因之一㊃，该措施从根本上改变了微软在软件开发生命周期中解决安全问题的方法。

- Code Red。在一个IIS Web服务器，即Internet服务器应用程序编程接口（Internet Server Application Programming Interface, ISAPI）扩展中的一个缓冲区溢出，于2001年6月18日被发现㊄。尽管针对该问题的补丁的可用性持续了将近一个月，但在同年的7月13日又发现了一个蠕虫，它利用该缺陷并使用消息“HELLO! Welcome to http://www.worm.com! Hacked By Chinese!”来破坏含有漏洞的服务器上的Web站点。在感染之后，该蠕虫将休眠20到27天，然后将试图对不同的固定的IP地址发动一个DoS攻击，这其中包括whitehouse.gov的IP地址。
- Slammer。SQL Slammer蠕虫利用了两个单独的漏洞，它们分别被微软的安全公告MS02-039㊅和MS02-061㊆在微软的SQL Server和桌面引擎中所识别。当该蠕虫在2003年1月25日第一次出现时，它用了大概10分钟的时间感染了最初的75 000个受攻击者㊇。该蠕虫所

㊀ http://www.digitaldwarf.be/products/ircfuzz.c
㊁ http://www.digitaldwarf.be/products/dhcpfuzz.pl
㊂ http://www.infigo.hr/en/in_focus/tools
㊃ http://www.microsoft.com/mscorp/twc/2007reviesw.mspx
㊄ http://research.eeye.com/html/advisories/published/AD20010618.html
㊅ http://www.microsoft.com/technet/security/bulletin/MS02-039.mspx
㊆ http://www.microsoft.com/technet/security/bulletin/MS02-061.mspx
㊇ http://en.wikipedia.org/wiki/SQL_slammer_worm

利用的缓冲区溢出漏洞（MS02-039）已被微软在6个月之前发现并添加了补丁，但仍然有大量的含有漏洞的服务器在使用。

- Blaster。2003年8月11日，18岁的Jeffrey Lee Parson将他所制造的蠕虫传播给了全世界，该蠕虫[㊀]利用了在Windows XP和Windows 2000中的一个DCOM远程过程调用（Remote Procedure Call,RPC）的缓冲区溢出[㊁]。同样，该问题的补丁也是早已经就可用的。该蠕虫可以通过一个SYN的溢出对windowsupdate.com执行一个分布式的DoS攻击。由于制造并传播了该蠕虫，Parson被判了18个月的监禁，3年保释下的监管以及100小时的社区服务[㊂]。

以上所列出的漏洞还很不全面，但它说明了历史上一些重要的、存在于微软的服务器端的漏洞，而该漏洞可能导致生成快速传播的蠕虫。任何网络漏洞都可以通过网络模糊测试来潜在地发现。

14.2 目标应用

这里有成百上千的目标应用可以选择，它们都可以暴露出远程可利用的网络协议解析漏洞。在表14-1中，我们列出了已知的网络漏洞的一小部分示例，以强调一些常用的目标分类。

表14-1 含有漏洞的应用程序的常用类型和以前所发现的漏洞的示例

应用程序分类	漏洞名称	参考信息
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绕过验证漏洞	http://archives.neohapsis.com/archives/vulnwatch/2004-q3/0001.html
基于RPC的服务	RPC 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远程挑战漏洞	http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20584
多媒体服务器	RealServer../DESCRIBE漏洞	http://www.service.real.com/help/faq/security/rootexploit082203.html
备份服务器	CA BrightStor ARCserve备份消息引擎缓冲区溢出漏洞	http://www.zerodayinitiative.com/advisories/ZDI-07-003.html

这6个类别构成了目标应用的很好的集合。然而，任何接受进入连接的应用程序或服务都可以作为一个潜在的目标应用。这包括硬件设备、网络打印机、PDA以及蜂窝电话等。任何具备接受网络通信功能的软件都可以使用网络模糊测试技术来进行审核。

为了对典型的目标应用有一个更加深入的了解，我们对开放系统互连基础参考模型（Open Systems Interconnection,OSI模型）[㊃]中所定义的7层结构的每一层都提供了示例，OSI模型如图14-1所示。尽管在任何单一的网络技术中没有被直接地实现，但当剖析网络技术以说明特定的功能应当被放在哪一层时，OSI模型通常被作为一个参考点来使用。从理论上来讲，网络协议

㊀ http://en.wikipedia.org/wiki/Blaster_worm

㊁ http://www.microsoft.com/technet/security/bulletin/MS03-026.mspx

㊂ http://weblog.infoworld.com/techwatch/archives/001035.html

㊃ http://en.wikipedia.org/wiki/Osi_model

模糊测试可以面向7层OSI模型中的任何一层。在实际测试中，你将会发现模糊器面向除了第一层物理层之外的所有层。由于每一层都被提出，因此下面就提供了与各层相关的一个历史上的漏洞。

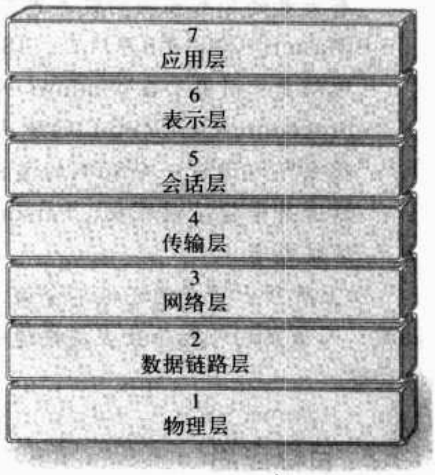


图14-1　开放系统互连（OSI）模型

14.2.1　数据链路层

与数据链路层相关的技术包括以太网框架和802.11框架。该层的漏洞是很有趣的，因为对此低层次网络层的处理通常是在操作系统内核中实现的。最近，该层中的一个漏洞是被Mitre在CVE-2006-3507中识别的[⊖]。在该漏洞中，一个攻击者可以危及一个机场中支持无线的Mac OS系统的安全，因为存在有多个基于栈的缓冲区溢出漏洞。该溢出发生在内核的环境中，可以被利用以彻底破坏被影响的系统。这里存在的一个有趣的需求是，攻击者必须要位于他或她所面向的无线网络的范围之内。如果你曾经在咖啡店中连接过公共无线网，并且你的Mac OS系统曾经发生过故障，那么你可能会谨慎地查看房间并确认网络环境。

APPLEGATE

2006年，在拉斯维加斯的Black Hat简报中，当安全问题研究者Jon“Johnny Cache”Ellch和David Maynor发布了一段录像，以说明通过一个设想的无线网络驱动器中的漏洞对Apple Macbook所造成的远程破坏时，引起了很大的争论[⊖]。接下来所发生的媒体的狂暴行为包括从所有卷入的团体中大量地揭发隐私，也包括Apple的指控即研究者从没有将该漏洞足够多的详细信息进行共享，以允许他们来证实该声明。Apple的拥护者也加入到了这场争论中，有情形表明该漏洞利用了一个第三方的驱动程序，而不是Apple的开发者所编写的程序。

最后，Apple发布了一系列的补丁，包括CVE-2006-3507，但继续主张该漏洞来自于Black Hat说明所引发的内部审核。发现此缺陷的荣誉始终没有被授予Maynor和Ellch。在这个过度戏剧化的一系列连锁事件中，最有可能的是我们可能永远都不会知道事情的真相。

14.2.2　网络层

第3层即网络层，它包括IP协议和Internet控制消息协议（Internet Control Message Protocol，ICMP）。尽管TCP/IP协议最为常见的实现已经在前些年经过了很好的测试，但是漏洞仍然可以在该层被发现。同时值得一提的是，Windows Vista包含一个完全被重写的网络栈，这表明它可能是模糊测试的一个很好的目标。最近被发现的一个TCP/IP协议中的漏洞被描述在MS06-032

⊖ http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-3507

⊖ http://blog.washingtonpost.com/securityfix/2006/08/hijacking_a_macbook_in_60_seco.html

中，即“TCP/IP中的漏洞可以允许远程代码执行”㊀。内核中出现的漏洞是由于错误解析IPv4源路由选项而引起的。这个不正确的处理导致了一个缓冲区溢出，而该溢出可以被远程攻击所利用以获得对内核级的访问。

14.2.3 传输层

再上面的一层是第4层即传输层，该层包括TCP和UDP。正如前面所提到的，大多数的TCP/IP实现已经经过了很好的测试，但过去该层仍然存在有问题。该层漏洞的一个突出的例子是陈旧的利用带外TCP包的“Winnuke”攻击㊁。可以证明，Winnuke攻击是到目前为止最简单的远程内核DoS攻击。远程的使一个受影响的系统崩溃所需要做的所有工作就是使用TCP紧急指针集来传输一个TCP包。该指针可以使用任意socket API来简单地设置，方法是通过简单地将包指定为包含带外数据。

14.2.4 会话层

OSI模型中的第5层即会话层，该层包含两个非常特别的协议，这两个协议都实现了远程过程调用。这些技术是DCE/RPC（微软的MSRPC）和ONC RPC，后者也被称为Sun RPC。这两个协议分别是针对Windows和UNIX系统而实现的。过去，通过使用这些技术发现了大量严重的漏洞。其中最为严重的一个漏洞是在微软的安全公告MS04－011㊂中被添加了补丁，该漏洞被Sasser㊃蠕虫使用而得到了疯狂的传播。该漏洞存在于二进制文件lsass.exe中，该文件暴露了所有当前版本的Windows操作系统中已被默认注册的RPC端点。特别的，该漏洞的产生是由于DaRolerUpgradeDownLevelServer函数中的一个缓冲区溢出，并且该漏洞可以被远程利用。

14.2.5 表示层

在第6层所采用的技术中，使其更加有利于进行模糊测试的一个很好的例子是在Sun RPC中所使用的外部数据表示XDR（eXternal Data Representation）。在以前所发现的与XDR相关的漏洞中，一个非常好的例子是Neel Mehta所发现的xdr_array整数溢出㊄。该发现的核心是对一个整数溢出的滥用。如果一个攻击者为一个数组指定了大量的实体，那么可能分配了一个不够大的缓冲区，然后随着数据的写入而发生了溢出。攻击者可以利用这个内存破坏来危及底层系统的安全。

14.2.6 应用层

第7层是应用层，它对于网络协议模糊测试而言，是最常被想起和作为测试目标的OSI层。该层包括许多常用的协议，如FTP,SMTP,HTTP,DNS以及许多其他标准的私有的协议。历史上，

㊀ http://www.microsoft.com/technet/security/Bulletin/MS06-032.mspx
㊁ http://support.microsoft.com/default.aspx?scid=kb;[LN];168747
㊂ http://www.microsoft.com/technet/security/Bulletin/MS04-011.mspx
㊃ http://en.wikipedia.org/wiki/Sasser_worm
㊄ http://bvlive01.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=20823

在该层被发现的漏洞要多于在其他层被发现的漏洞。第7层是经常被模糊测试的层，因为对常用的协议而言，这里有许多不同的实现。

本章的剩余部分重点关注于第7层，但是应当记住，任何层上的软件都会存在实现问题，并且在对一个目标执行彻底的审核时不应当忽略这些软件。

14.3 测试方法

对大多数层而言，可用的模糊测试方法与第11章“文件格式模糊测试”所提出的方法是相同的。从高层次而言，我们可以使用强制性或智能的方法来进行文件格式模糊测试和网络模糊测试。然而，当应用于基于网络的模糊测试时，这些方法的使用上存在着一些不同。

14.3.1 强制性或基于变异的模糊测试

在文件格式模糊测试的环境下，强制性模糊测试要求模糊测试者获得目标文件格式的有效示例。然后模糊器采用不同的方法将这些文件进行变异，并将每个测试用例提供给一个目标应用。在网络模糊测试的环境下，模糊测试者通常使用一个嗅探器来捕获有效的协议通信，既可以在运行之前静态的捕获也可以在运行时动态的捕获。然后模糊器将所捕获的数据进行变异，并在目标应用上进行使用。当然，情况并不总是这么简单。例如，考虑一下实现了对基本的重复攻击进行防护的任何协议。在这种情况下，简单的强有力模糊测试除了对诸如验证进程的会话初始化代码之外，将不会对其他任何代码有效。

一个简单的基于变异的模糊器应用失败的另外一个例子是当其试图对嵌入到校验和中的协议进行模糊测试时。除非校验和字段被模糊器动态更新，否则所传输的数据在执行任何深度解析之前将被目标软件所丢弃。在这种情况下，测试用例就被浪费了。基于变异的模糊测试在文件模糊测试领域应用得很好，但在通常情况下，下一个模糊测试方法在网络协议模糊测试中应用得更好。

14.3.2 智能强制性模糊测试和基于生成的模糊测试

对于智能强制性模糊测试而言，你必须首先要花费一些时间来实际研究一下协议的规范。一个智能模糊器仍然是一个模糊测试引擎，因此仍然会执行一个强制性攻击。然而，这要依赖于用户的配置文件，以使测试过程更加智能化。这些文件通常包含描述协议语言的元数据。

当对一个不重要的网络协议如FTP中的控制通道进行模糊测试时，智能模糊测试通常是最佳的方法。描述每个协议动词（USER,PASS,CWD等）的一个简单的语法以及对每个动词参数的数据类型的描述，将允许进行一个非常彻底的测试。基于生成的模糊器的开发可以构建在一个已经存在的框架之上，如PEACH[⊖]，或者从头开始创建。后一种情况下，所开发的特定于某种协议的模糊器在大多数情况下不能被应用于测试其他协议。使用这种方法，通常的设计思想被抛弃，从而简化了测试过程的设计和实现。

⊖ http://peachfuzz.sourceforge.net/

这里有许多公开的、可用的面向一个单一协议的测试工具，但是许多测试工具仍然是私有的，因为它们实质上可以看做是一个被加载的工具。这就是说，当一个开发者发布一个通用的模糊器时，无论它是基于变异的还是基于生成的，终端用户都需要进行一些研究来发现漏洞。使用一个特定于某种协议的模糊器，那么需要做的所有工作就是选择一个测试目标。如果终端用户使用该模糊器来测试与开发者所测试的相同的目标，那么将会有相同的错误被发现，这时，工具对于最初的开发者而言将具有不高的价值。如果发现了从来没有被该模糊器测试过的目标，并且该目标使用相同的协议，那么情况将不会是这样。发布这样一个工具的决定要归结于对规范的个人理解以及错误消除和共享研究的想法。

14.3.3 修改的客户端变异模糊测试

对模糊测试常见的一个批评是随着所开发的模糊器的复杂性的增加，其复杂性开始接近于开发一个完整客户端的复杂性。因此为什么不降低其复杂性呢？有一个前面没有提到过的特殊的模糊测试类型，该方法是这样一种技术，即客户端－服务器通信中的客户端源代码（当可用时）被修改以创建一个模糊器。实际上，该方法并不是在一个模糊器中实现一个完整的协议，而是该模糊器被嵌入到一个已经使用了所希望的语言的应用程序中。这样做是具有优越性的，因为模糊器可以访问已经存在的、生成有效的协议数据所需要的程序，并且将模糊器开发者的工作量减少到最小。

尽管我们不知道有采用这种方法的任何公开发布的模糊测试工具，但是已经按照这种方式进行了一些开发工作。其中的一个例子是由GOBBLES所编写的sshutuptheo[⊖]OpenSSH漏洞利用工具，它利用了主流SSH服务器中的一个验证前整数溢出。

这种方法特别适应于类似于SSH这样的复杂协议。然而，开发者必须要意识到客户端可能会具有的、将会影响代码覆盖的任何限制条件。例如，如果选择了SSH，并且所选择的客户端只支持版本1的SSH，那么对版本2的SSH的细节将不会有代码覆盖。当然，它所具有的另外一个缺点是一个合理的客户端可能会被进行大量修改以覆盖许多测试用例。

14.4 错误检测

错误检测是模糊测试过程中的一个关键组成部分，因此第24章“智能故障检测”专门对此进行了讨论。尽管在该章之前不会涉及到高级的错误检测方法，但我们可以来看一些应用于网络协议模糊测试的基本技术。在网络模糊测试过程中检测错误的难度完全依赖于测试目标。例如，考虑一个网络daemon程序发生崩溃，并且在任意错误发生之后停止接收连接。很明显，从基本的错误检测的观点来看，这是一个很重要的行为。当服务器发生崩溃时，简单的假定是最后一个测试用例导致了它崩溃。这可以完全由模糊器来完成，而不需要在目标机器上的软件代理或人工评审的任何帮助。

让我们考虑目标应用能够处理错误、或者生成一个单独进程以及独立维护监听socket的另

⊖ http://online.securityfocus.com/data/vulnerabilities/exploits/sshutuptheo.tar.gz

外一个场景。理论上，我们可以触发一个可利用的漏洞，当然并不知道我们在目标机器上应用了某种形式的深度评审。可以安全地假设在大多数模糊测试过程中，我们可以访问目标机器。下面来看一些我们可以用来在目标机器上检测错误的基本方法。

14.4.1 人工方法（基于调试器）

在可以对一个机器进行本地访问时，监视异常最简单的方法就是将一个调试器关联到一个进程。该调试器可以检测到何时发生了一个异常，并允许用户决定采取什么动作。使用Ollydbg,Windgb,IDA和GDB都可以实现这种方法。这里所存在的问题是，哪个测试用例或测试引起的序列导致了该行为。

14.4.2 自动化方法（基于代理）

考虑设计一种方法，以替代人工的调试过程。不是使用一个调试器应用，模糊测试者而是编写了一个特定于运行在目标机器上的目标平台的调试代理。该代理有两项任务。第一项工作就是在目标进程中监视异常。第二项工作是同远程系统中的模糊器进行通信。这允许将数据和错误检测关联起来。该方法的缺点是开发者需要为被测试的每个平台创建一个代理。同样，这个概念将在第24章中进行详细讨论。

14.4.3 其他方法

尽管当执行网络模糊测试时，调试器可能会成为检测异常的最有价值的工具，但是不要忽略其他可用的方法。应用程序和操作系统日志可能会提供有关已发生的问题的信息，这里的挑战是将问题与可靠的模糊测试用例进行关联。同时，要确保对系统的性能降低进行监视，因为这可能表明了某些隐藏的问题。性能下降可以包括增加的CPU利用率或者内存的耗尽。导致一个无限循环的测试用例可能永远不会触发一个异常，但同时会导致一个DoS。其中的要点很简单：尽管调试器对于发现漏洞而言是奇特的工具，但是你也不能忽略其他的方法。

14.5 小结

网络协议模糊测试可能是最广为人知和利用最广泛的模糊测试类别，并且有许多不同的方法来实现网络协议模糊测试。它的流行受一些因素的共同影响，并不只是因为它可以发现高风险、远程的验证前漏洞。除此之外，它是一种成熟的模糊测试类型，因为存在许多公开的、可用的工具可以帮助安全问题研究者。在了解了网络模糊测试的一些常用方法之后，就可以讨论基于网络的模糊器的具体实现了。

第15章 网络协议模糊测试：UNIX平台上的自动化测试

"我认为我们已经达成一致，过去已经结束。"

——George W. Bush,会见John McCain时的讲话,达拉斯早间新闻，2000年5月10日

尽管微软的Windows在桌面操作系统领域居于统治地位，但UNIX系统仍然在服务器平台的选择中占据着大部分市场。例如Web服务器Apache，广泛运行于UNIX系统上，并且根据最新的NetCraft调查[⊖]，它在微软所有的IIS产品中保持着将近30分的领先优势。影响主流UNIX系统的漏洞都非常重要，并且具有深远的影响。Internet上有许多产品运行于基于UNIX的DNS，邮件和Web服务。例如，考虑一个在伯克利的Internet域名（Berkeley Internet Name Domain，BIND）服务器中所发现的缺陷，它可以被利用以破坏大部分的Internet通信。尽管无法确切知道以前所发现的漏洞有多少是通过模糊测试发现的，但可以肯定的是许多漏洞是在一个模糊器的帮助下才发现的。

本章中，我们并不从头开发一个定制的基于UNIX的模糊器。相反，我们将提出并利用由SPIKE模糊测试框架提供的可用的脚本编写接口，SPIKE是本书中经常提到的一个开源模糊器。由于在本章中，我们不关注于模糊测试框架开发的实际编码问题，我们将自始至终的使用SPIKE来对一个非开放源代码的应用的模糊测试进行一次走查。

15.1 使用SPIKE进行模糊测试

为了说明使用SPIKE进行模糊测试的过程，让我们从头到尾分析一个完整的例子，首先要选择一个测试目标，研究目标协议，在一个SPIKE脚本中描述该协议，然后设置该模糊器开始工作。

15.1.1 选择测试目标

为了我们要达到的目的，这里选择一个具有下列所描述特性的目标软件：

- 目标软件应当被普遍使用。
- 目标软件应当可以通过演示或评估来方便地获得。
- 目标软件应当使用一个公开的或易于被逆向工程的协议。

这里存在着许多可能的目标软件，但是出于此例子的目的，我们选择了Novell NetMail[⊖]，它是一个实现了许多公开协议的企业级邮件和日历系统，并且满足我们需要的所有准则。特别

⊖ http://news.netcraft.com/archives/2007/02/23/march_2007_web_server_survey.html

⊖ http://www.novell.com/products/netmail/

的，我们面向的是NetMail网络消息应用协议（Networked Messaging Application Protocol，NMAP）。NMAP被没有被普遍存在的网络扫描和侦测工具Nmap[⊖]所扰乱。那么NetMail NMAPM究竟是什么意思呢？我们并不知道准确的定义，因此要感谢Novell热心的共享了下面的定义：

> 缩写NMAP代表网络消息应用协议。它是一个基于文本的IP协议，由Internet Assigned Numbers Authority（IANA）在端口689所注册，并使用NIMS代理来进行通信。当与NDS eDirectory的分布式特性相结合时，该协议就允许NIMS代理运行在不同的服务器上（甚至是不同的平台）来进行操作，就好像它们是在同一个服务器上一样。当消息服务的需求增加时，不是用一个更大的服务器来替换该服务器，NMAP允许额外的服务器被添加到这个“簇”中。每一个版本的NIMS都提供了NMAP协议的RFC类型的文档[⊖]。

NMAP看起来像是一个构建在TCP之上的、Novell所开发的基于文本的私有协议。将一个私有协议作为测试目标既有优点也有缺点。其不足的一面是，我们所开发的模糊器在测试该协议之外似乎没有其他用处。然而有利的一面是，一个自定义的协议意味着长期存在的、经过试验的真实的解析库是不可用的。协议提供商必须要自己编写协议解析器，较少的人评审过相关代码，因此它可能会包含大量的漏洞。

Novell是很热心的，它通过其Web站点提供了NetMail的免费90天的试用版[⊜]，因此在实验室环境中下载并安装目标软件非常容易。在安装之后，我们使用Novell提供的补丁来更新软件。这是一个关键的步骤以确保我们不会重复发现旧的错误（如果发现了一些）。指出你在一次审核中将该步骤错过了一个星期，那么将是相当沮丧的。下一步就是要研究NMAP协议的相关细节。

15.1.2 协议逆向工程

在我们可以描述针对SPIKE的NMAP协议之前，显然需要先将其理解透彻。有许多方法可以处理该步骤。最明显的方法可能就是在我们的实验室环境中监视合理的NMAP通信。该方法实际上要比你所想象的要复杂一些。当处理一些复杂的企业级软件时，有时候生成你所寻找的通信是很困难的。另外一种方法是基于其他的工作进行构建。可以通过Google查询一下其他人对任意给定的协议了解的如何。查看在开源Wireshark（以前的Ethereal）嗅探器中是否存在ygie协议解码器也是一个不错的方法。跳转到Wireshark版本控制库，特别是epan\dissectors[⊛]目录中，并且查看是否能够找到你的协议。

另外一个也是最少被使用的方法就是简单地与NMAP daemon程序本身进行通信，以查看它是否为我们提供任何信息。为此，我们需要首先确定应用程序使用哪个端口与客户端进行通

⊖ http://insecure.org/nmap/

⊖ http://support.novell.com/techcenter/articles/ana20000303.html

⊜ http://download.novell.com/index.jsp

⊛ http://anonsvn.wireshark.org/wireshark/trunk/

信。假定文档中并没有公开的说明NMAP绑定到TCP的689端口，那么我们可以在微软TCPView[⊖]（以前的Sys Internals）工具的帮助下手工得到此信息。简单的启动TCPView程序，就会立即发现nmapd.exe在TCP的689端口进行监听。我们可以使用一个基本的TCP连接工具如netcat[⊜]或者甚至是Windows telnet命令来连接到daemon程序。采用一个大胆的猜想，我们发出HELP命令，然后欣喜而又惊讶地看到，它为我们提供了daemon将接受的命令的一个列表。

前一段所进行的工作并不差。利用我们所收集的信息再往前进一步，即把nmapd.exe的二进制文件加载到我们的解析器IDA Pro中。按Shift+F12键以浏览图15-1所示的字符串数据库。然后就可以定位在该图中所显示的有用响应的位置。即以ASCII文本“1000”开始的字符串。现在，我们依据字符串的内容而不是其地址对字符串数据库进行分类，然后滑动到以“1000”开始的行。在这里，我们发现了NMAP服务器所支持的所有命令及其期望语法。

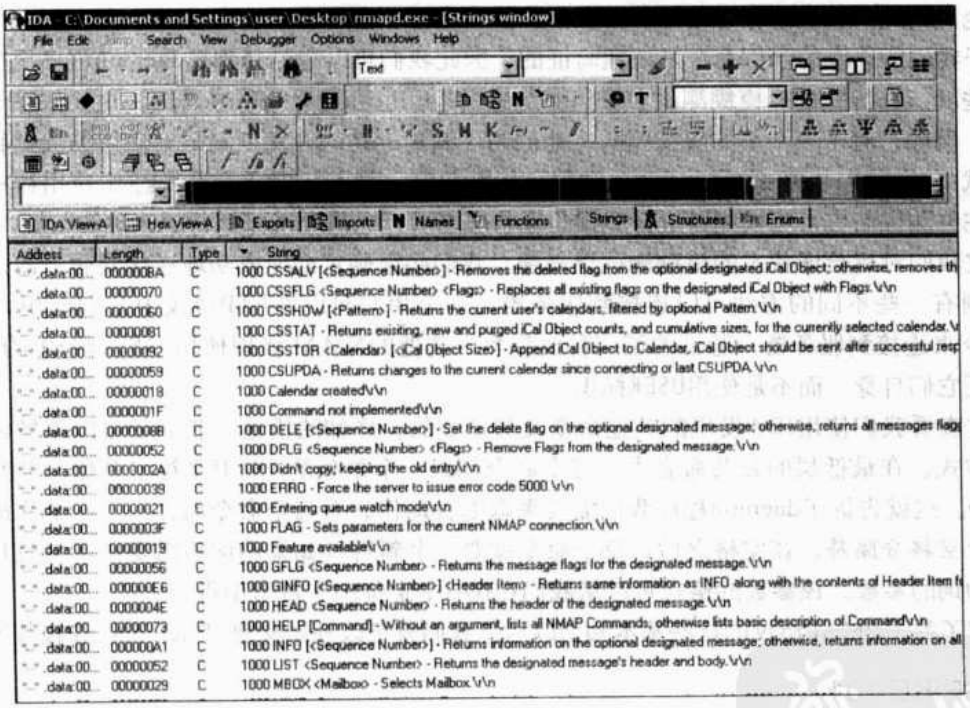


图15-1 在可执行文件nmapd.exe中发现的字符串列表

在评审了不同的命令及其所描述的语法之后，针对有用输出的符号变得清晰起来。尽管命令的某些参数与其方法是不一致的，但在大多数情况下使用了如下的原型：

- `<argument>`。这种类型的参数是必需的。如果它没有被指定，那么命令将执行失败。
- `[argument]`。这种类型的参数是可选的。

⊖ http://www.microsoft.com/technet/sysinternals/Networking/TcpView.mspx

⊜ http://www.vulnwatch.org/netcat/

- {CONSTANT1|CONSTANT2|CONSTANT3}。这种类型的参数是必需的，并且必须为后面的命令从一组常量字符串中选取。每个可能的取值用字符|分隔。

同时，任何没有包含在角括号中的字符串应当被作为一个文字字符串来处理。包含在角括号中的任何字符串应当被作为一个变量来处理。这些规则对于存在语法描述嵌套的命令同样适用。例如，PASS命令被描述如下：

```
PASS [SYS | USER <Username>] <Password>
```

该原型指定了这个命令的后面将总是跟着字符串值SYS或USER。由于变量Username存在于字符串USER后面的角括号中，因此它是必需的，但只有当字符串USER被选择时才如此。如果字符串SYS被选择，那么就不需要Username参数。最后，作为该命令的最后一个参数，变量Password总是必需的。PASS命令的协议可能是这个协议中最复杂的命令，但并没有太多的关于此的评论。因此，我们选择了一个非常友好的协议进行模糊测试。

由于我们所选取的示例命令是处理验证的，因此我们可以立刻猜到该daemon程序至少需要为其某些功能进行验证。模糊测试者可能对此感兴趣，也可能对此不感兴趣，这要依赖于测试的范围。例如，对于一个小的研究团队，他们为了满足最终的开发期限只对应用的关键状态进行了测试（预验证），那么后验证命令就很可能被忽略。然而，对于一个关注于应用程序所有状态的完整测试而言，当模糊器进行完预验证状态之后，它需要确保执行了正确的验证。

假定我们对后验证状态的模糊测试感兴趣，我们必须确定如何成功地连接到服务器。我们可以看到有一些不同的方法可以连接到服务器。一个用户可以在FTP协议中分别使用USER和PASS命令来连接到服务器。验证的第三种类型允许其他的NMAP代理使用PASS命令中的SYS标识来验证它们自身，而不是使用USER标识。

通过查看我们使用IDA发现的其他命令及其便捷的、详细的描述，我们就可以理解该协议的工作方式。在最低层的公共命名中，每个命令都以一个单独的ASCII字符串开始，即我们所指的*verb*。这就告诉了daemon程序我们需要采取哪种动作。对每个命令而言，下一个构成部分就是一个空格分隔符。在空格之后，每个命令需要一个新行（如果动词不需要任何参数）或者特定于动词的参数。该参数的格式可以从我们在IDA中定位的字符串中得到。

了解了关于NMAP协议的这些基本知识以后，我们就可以开始构建SPIKE NMAP模糊器了。

15.2 SPIKE 101

在第21章“模糊测试框架”中所介绍的其他可用的模糊测试体系结构中，简要涉及了SPIKE。但在这里，我们将对其进行深入的分析。由于我们将使用SPIKE来对该协议进行模糊测试，因此下面的学习是很重要的，即熟悉SPIKE的模糊测试引擎的工作机理及其通用的、可编脚本的、基于行的TCP模糊器的工作机理。

15.2.1 模糊测试引擎

SPIKE通过迭代变量和模糊字符串来进行工作。考虑一下作为协议当中的字段的变量，如

用户名、密码和命令等。这些变量及其在数据流中的位置将是特定于每个目标的。模糊字符串是那些能够潜在触发一个错误的不同的字符串和二进制数据的库，基于以往导致其他软件发生问题的特定序列的经验来仔细选择模糊字符串库中的元素。例如，考虑一个非常基本的模糊字符串，如一个包含64 000个连续A字符的ASCII字符串。该模糊字符串可能会替换协议中的一个用户名变量，从而触发一个预验证缓冲区溢出。记住术语模糊“字符串”有一些令人误解。一个模糊字符串可以是任意的数据类型，甚至是类似于二进制数据的一个XDR编码的数组。

15.2.2 通用的基于行的TCP模糊器

我们这里所要创建的模糊器被称为通用的基于行的TCP模糊器。该应用的代码存在于SPIKE源代码包中的line_send_tcp.c文件中。该模糊器是关于SPIKE的一个非常简单而又功能强大的模糊器。它将处理一个SPIKE脚本，将该脚本中的每个变量进行模糊化，并试图为每一次新的模糊测试重复连接到主机。这些变量根据它们在模糊脚本中的位置来进行模糊化。这意味着在大多数带有验证的协议中，与验证有关的信息将首先被SPIKE测试。

脚本语言所采用的实现方法允许用户编写脚本以直接访问SPIKE API函数。为了提供关于如何编写脚本的基础知识，下面首先列出了可能会调用的一些函数：

- `s_string(char * instring)`。该函数将向SPIKE添加一个固定的字符串。该字符串将永远不会改变。
- `s_string_variable(unsigned char *variable)`。该函数将向SPIKE添加一个变量字符串。当该变量被处理时，这个字符串将被模糊字符串所替代。
- `s_binary(char * instring)`。向SPIKE添加二进制数据。该数据将永远不会被改变。
- `s_xdr_string(unsigned cahr *astring)`。该函数将向SPIKE添加一个XDR类型的字符串。也就是说，它将包含一个4字节长度的标签，并将用0来扩展4倍长度。该字符串将永远不会改变。
- `s_int_variable(int defaultvalue, int type)`。该函数将向SPIKE添加一个整数。为了调用s_int_variable()，类型值应取如下之一：
- *Binary Big Endian*。最高有效位元（Most significant bit,MSB）整数，4字节。
- *ASCII*。标记十进制数的一个ASCII格式。
- *One Byte*。一个字节整数。
- *Binary Little Endian HalfWord*。最低有效位元（Least significant bit, LSB）位整数，2字节。
- *Binary Big Endian HalfWord*。MSB整数，2字节。
- *Zero X ASCII Hex*。以0x开头的ASCII格式的十六进制数。
- *ASCII Hex*。一个ASCII格式的十六进制数。
- *ASCII Unsigned*。一个ASCII格式的无符号十进制数。
- *Intel Endian Word*。LSB整数，4字节。

由于当我们将SPIKE作为一个编写脚本的主机使用时，并没有使用一个C预处理器，因此

我们需要知道该类型的整数值。下面摘自于SPIKE/SPIKE/include/listener.h的代码片段显示了其定义值：

```
#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
```

到现在为止所讨论的SPIKE的功能对于开发一个NMAP模糊器而言已经足够了。然而，让我们再来研究一下SPIKE在模糊测试领域的主要功能，基于块的协议表达。

15.3 基于块的协议建模

尽管我们选择了一个简单的基于文本的协议来进行测试，但是SPIKE通过使用其基于块的模糊测试功能可以支持更加复杂的协议。使用基于块的模糊测试允许动态创建具有有效字段长度的包。例如，考虑带有包结构的一个协议，该结构的前缀是一个用户名字段和所指定的用户名的长度。我们希望当对用户名进行模糊测试时，模糊器可以自动更新该长度字段。这就确保了模糊字符串正确传递给了用户名解析器。s_block_start()和s_block_end()函数允许你对这样的协议进行建模，并声明为如下的形式：

```
int s_block_start(char *blockname)
int s_block_end(char * blockname)
```

为了有效地利用这些函数，你只需要在字段被度量之前和之后进行声明。然后，当实际长度必须要被插入到数据流中时，使用若干个blocksize函数中的一个。它们类似于前面所讨论的整数字段，并且被详细地定义。注意到这些blocksizes中的一些是变量，这意味着将使用无效的blocksizes来执行一系列的模糊测试。对长度字段是否进行模糊测试取决于你所做的决定。下面列出了在SPIKE中可以使用的所有不同类型的blocksizes的一个较完整列表。

```
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_variable(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 * instring, int size)
s_blocksize_asciihex_variable(char * blockname)
```

15.4 SPIKE的额外特性

SPIKE不只是一个模糊器，它实际上是一个完整的模糊测试框架，该框架包含了大量有用函数的API，可以帮助我们简化定制模糊器的创建。它还包含许多特定于协议以及特定于应用程序的模糊器和模糊测试脚本。下面是除了SPIKE引擎和通信代码之外，SPIKE所提供的一些其他功能。

15.4.1 特定于协议的模糊器

SPIKE包含一小部分以前所编写的特定于协议的模糊器，如下所示：

- HTTP模糊器
- 微软RPC模糊器
- X11模糊器
- Citrix模糊器
- Sun RPC模糊器

在大多数情况下，这些模糊器只是作为如何使用SPIKE的示例来使用。这是因为它们已经存在了一段时间，并且在你可能感兴趣的每个目标中都运行良好。

15.4.2 特定于协议的模糊测试脚本

同时包含在SPIKE中的还有一些脚本，这些脚本可以插入到SPIKE所包含的许多通用模糊器的某个当中。这些脚本如下所示：

- CIFS
- FTP
- H.323
- IMAP
- Oracle
- Microsoft SQL
- PPTP
- SMTP

- SSL
- POP3

15.4.3 通用的基于脚本的模糊器

正如前面所提到的，在SPIKE中有许多采用脚本作为输入的通用模糊器。下面的通用模糊器可以在SPIKE中被找到：

- TCP监听（客户端）模糊器
- TCP/UDP发送模糊器
- 行缓冲的TCP发送模糊器

15.5 编写SPIKE NMAP模糊器脚本

再回到我们的NetMail目标应用，下面所列出的SPIKE示例脚本完全是根据HELP输出消息以及来自于IDA Pro的字符串列表创建的。该脚本首先对实际验证命令本身进行了模糊测试，以寻找可能的预验证错误。如果提供了一个正确的用户名和密码，那么该模糊脚本将继续对后验证命令进行模糊测试。

```
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");
```

接着，我们将所选择的一个调试器关联到目标系统的NMAP进程，并通过SPIKE运行我们所开发的脚本。我们使用SPIKE的通用的基于行的TCP模糊器，并使用下面的命令行选项来执行名为nmap.spk的脚本：

```
./line_send_tcp 192.168.1.2 689 nmap.spk 0 0
```

在执行一小段时间之后，我们的模糊器就发现了一个可利用的栈溢出。图15-2显示了通过OllyDbg得到的NMAP daemon程序处于漏洞时的状态。

在图15-2中，右上角的窗格显示了程序崩溃时的寄存器值。寄存器EBP（栈结构指针），EBX,ESI,EDI以及更重要的EIP（指令指针）都被十六进制值0x41或者ASCII字符A所改写。右下角的窗格显示了栈结构，我们可以清楚地看到它已经被一个由A组成的长字符串所改写。左上角的窗格中通常显示被执行的指令列表，但由于当前指令指针处于地址0x41414141，因此该窗格是空的。没有内存页被映射到该地址，因此在解析窗口中没有显示任何内容。接下来，我们必须要查到是哪个动词－参数对导致了该崩溃，这样我们就可以将它再现，甚至可以利用它。

有许多方法可以做到这一点。其中最简单的一个方法是利用SPIKE中内嵌的一个特性。如果SPIKE不能连接到目标，那么它通常会崩溃。通过检查SPIKE崩溃之前的最后一个输出，我们就可以确定是哪个测试用例导致了NetMail NMAP中的崩溃。然而，由于我们和一个调试器

相关联，因此目标进程将不会崩溃，而是挂起，并且SPIKE的崩溃特性将不会出现。因此，为了能够重现崩溃，我们重新启动该进程，允许它在没有监视的情况下运行（不关联到一个调试器），并依赖于SPIKE崩溃以及受影响的NMAP daemon程序。我们再次运行测试并等待，再次观察到了NMAP中的一个崩溃，接着是SPIKE中带有下面输出的一个崩溃：

```
-snip-
Fuzzing Variable 5:1
Read first line
Variablesize= 5004

Fuzzing Variable 5:2
Couldn't tcp connect to target
Segmentation fault
-snip-
```

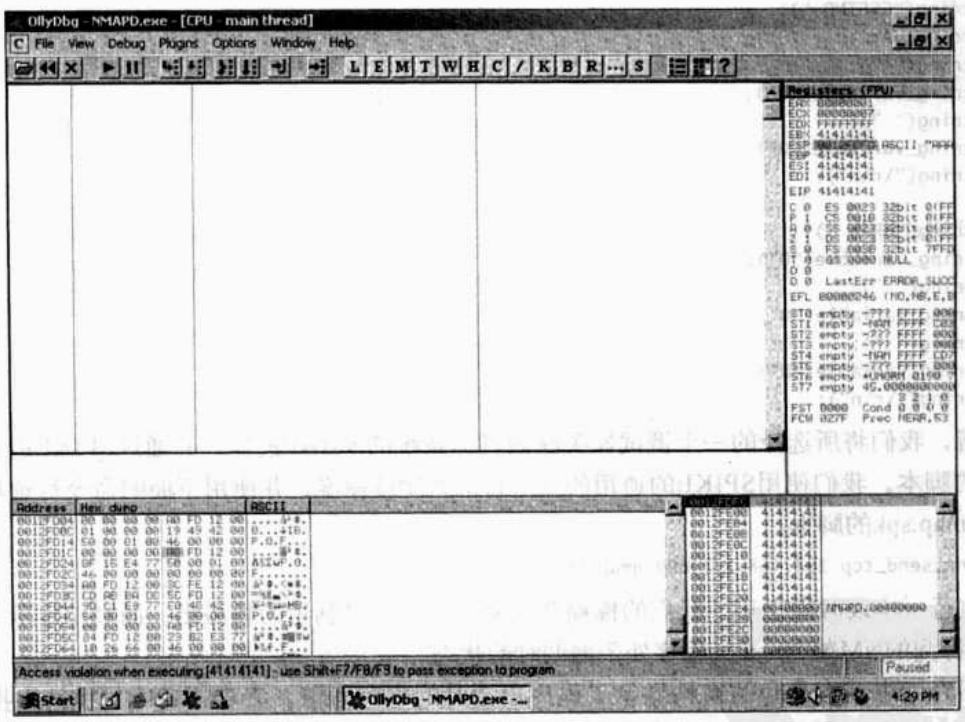


图15-2　nmapd.exe中的一个可利用栈溢出

该方法对于追查触发NMAP中错误的测试用例而言，无疑是一种最基本的技术。更加科学的方法包括使用一个嗅探器来监视所有的网络通信。NMAP将崩溃，并且不能够再对请求做出响应。但在另一方面，SPIKE将继续传输测试用例。通过定位最后一次请求响应的传输，我们就可以确定可能是哪个测试用例触发了该错误。

回到最初所描述的向后追查的技术，来自于SPIKE的最后一次成功的传输标记在“模糊变量5：1”行中。这告诉我们最后一次成功的连接是关于模糊变量5。发送给该进程的最后一个

模糊字符串是1。为了确定哪一个模糊变量是模糊变量5，我们只要简单地加载我们的SPIKE脚本，然后从第0行开始统计包含有“变量”这一词的行的个数。在遇到CREA命令中的只能用于后验证的一个动词参数时停止计数。接下来，我们必须要确定为模糊字符串1，即CREA命令的参数，使用了什么值。

有许多方法可以实现这一点。一种方法是如前面所提到的使用一个嗅探器。另外一种可以采用的方法是向模糊测试程序line_send_tcp.c中添加一个printf()函数，以命令应用程序输出当前的模糊字符串，然后再重新运行模糊器。不论使用哪一种方法，我们都发现导致错误的字符串就是“CREA<longstring>”。我们立刻就可以重新创建这个后验证崩溃。需要做的所有工作就是让用户连接到服务器并发送一个恶意的CREA命令参数。这并不很困难：为发现远程漏洞只要花费很少的时间。这可能会使你产生这样的疑问，为什么许多提供商在正式发布他们的软件之前不应用这种类型的测试呢？如果我们想要继续对NMAP进行模糊测试，那么可以在关于CREA命令的SPIKE脚本中删除此部分，以使我们不再用一个已知的问题来破坏测试目标。

15.6 小结

当考虑编写你自己的模糊器时，首先对已经存在的模糊器以及模糊测试的体系结构进行评估是非常重要的。对于一个简单的测试目标如NMAP协议而言，从头开发一个模糊器得到的收获将会很小。只要花费几个小时的时间，我们就可以掌握一些SPIKE脚本，以有效地执行NMAP daemon代码。但是要记住，你所得到的结果的质量将反映出开发模糊器所花费的时间。在这种特殊的情况下，我们追求的是绝对的最小化。我们的选择是连接到服务器，发出一些命令请求，然后退出。在NMAP中还可以发现许多其他的函数可以被添加到脚本中，以进行更多的测试。



第16章 网络协议模糊测试：Windows平台上的自动化测试

"我无法想像奥萨马·本·拉登等人如何理解光明节的乐趣。"

——George W. Bush，白宫烛台点火仪式，Washington DC，2001年12月10日

尽管UNIX系统在服务器领域可能居于统治地位，但在全世界范围内所安装的更多的操作系统是微软的Windows操作系统，这也使它成为了易受攻击的目标。影响Windows桌面的漏洞经常在创建现有的许多被恶意代码感染并控制的计算机网络时被利用。考虑一下Slammer蠕虫[㊀]，它利用了微软SQL Server中的一个缓冲区溢出，并被作为说明通过网络导致Windows漏洞的威力的一个示例。该漏洞被描述在2002年7月24微软发布的安全公告MS02-039中[㊁]，而Slammer蠕虫在2003年1月25日被肤浅地加以描述。该蠕虫实际上并没有有效载荷，它只是简单地利用被感染的主机来扫描并传播到其他被感染的机器[㊂]。尽管该蠕虫缺乏有效载荷，但这种入侵式的扫描生成了足够的通信以导致对Internet、信用卡的处理的破坏，以及在某些情况下造成了对蜂窝电话网络可用性的破坏。令人感兴趣的是甚至在4年之后，Slammer蠕虫仍然位于最常见的5种通信生成事件之内[㊃㊄]。显然，在Windows中所暴露的一个网络漏洞具有深远的含义。

在前面的章节中，我们利用了一个已经存在的模糊器框架SPIKE，来构建了在UNIX环境中面向Novell NetMail NMAP daemon程序的一个协议模糊器。本章中，我们采用一种不同的方法，从头至尾构建一个简单的基于Windows的、具有GUI界面以及用户友好的模糊器。尽管名为ProtoFuzz的最终产品只提供最基本的功能，但是它提供了一个很好的扩展平台，并且为模糊器的创建提供了一些不同的视角。下面首先来讨论一下该模糊器所具有的特性。

16.1 ProtoFuzz的特性

在我们开始开发该工具之前，首先必须要考虑一下所需要的以及所期望的特性。在最基本的层面上，一个协议模糊器只是简单地在目标上传输变异的包。因此，只要模糊器能够生成并发送数据包，那么它就满足要求了。然而，如果ProtoFuzz具备理解我们想要模糊的包的结构的功能，那么将会更好。下面让我们对这个基本的需求进行扩展。

16.1.1 包结构

在我们的模糊器可以发送一个数据包之前，需要了解如何来创建一个包。对于现有的模糊

㊀ http://www.cert.org/advisories/CA-2003-04.html

㊁ http://www.microsoft.com/technet/security/bulletin/MS02-039.mspx

㊂ http://pedram.openrce.org/_research/slammer/slammer.txt

㊃ http://isc.sans.org/portreport.html?sort=targets

㊄ http://atlas.arbor.net/

器而言，在模糊测试中采用下面的三种基本方法之一来组装包：

- 技巧性的测试集。PROTOS测试集[⊖]及其配套的商业工具Codenomicon[⊜]，都是将用于模糊测试的数据包的结构进行硬编码。像这样来创建测试集很浪费时间，因为它需要分析一个协议的规范并开发成百上千个硬编码的测试用例。
- 生成模糊器。正如我们在前面的章节中所看到的，像SPIKE这样的工具需要用户创建一个描述数据包结构的模板。然后，模糊器就负责在运行时生成并传输单个的测试用例。
- 变异模糊器。如果不是从头创建一个包，那么一个可选的方法是从一个已知的好的包开始，然后连续对该包中的部分进行变异。尽管在强制性测试模式下，该包中的每一个字节都可以被变异，但该方法通常不适用于协议模糊测试，因为该方法的效率不高，并且会生成与网络协议不一致的包，甚至会生成无法到达目标的包。可以通过借鉴协议模板方法的一些技巧来修改该方法。该模糊器从一个已知的好包开始工作，然后用户通过识别应当被作为测试目标的那部分数据来创建一个模板。该方法将被ProtoFuzz所利用。

没有一个单独的方法要优越于其他方法，每个方法都有其各自的适用情形。ProtoFuzz之所以选择包变异方法，就是因为它所具有的简洁性。以一个可以从被观察的网络通信中获取的已知的好包开始，就允许用户可以立即开始模糊测试，而不用花费大量的时间来研究协议并创建生成模板。

16.1.2 捕获数据

由于选择了包变异作为模糊测试的方法，那么ProtoFuzz应当能够在混乱模式下捕获网络数据，就像一个嗅探器一样。我们将在ProtoFuzz中创建一个网络协议分析器，这样，它就可以捕获从目标应用进出的通信，然后我们就可以选择单独的包来进行模糊测试。为了实现此功能，我们将利用一个已存在的包捕获库，该库将在“ProtoFuzz的开发”一节中做详细介绍。

16.1.3 解析数据

尽管这不是绝对需要的，但我们想要ProtoFuzz除了捕获数据之外，还能够以一种易于理解的形式将所捕获的包的内容展现出来。这将有助于用户识别包中适于进行模糊测试的部分。网络包数据只是一系列遵循特定模式的字节。以一种用户易读的格式来显示数据要求利用一个捕获库，而该库能够理解包结构并将其分解为组成数据流的不同的协议头和数据段。许多人对Wireshark所使用的表现格式很熟悉，因此我们将其作为显示格式的基础。图16-1显示了被Wireshark所捕获的一个简单的TCP包。

这个来自于Wireshark的特定的屏幕快照将段在三个窗格中进行了显示。最上面的窗格列出了被单个捕获的包。选择其中的一个包将把该包的内容加载到下面的两个窗格中。中间的窗格显示了将该包分解为单独的字段。在这个例子中，我们可以看到Wireshark比较熟悉AOL实例消息协议，因为它成功地将TCP包的整个数据段进行了解析。最后，底层的窗格显示了被选择

⊖ http://www.ee.oulu.fi/research/ouspg/protos/

⊜ http://www.codenomicon.com/products/

包的原始十六进制和ASCII字节。

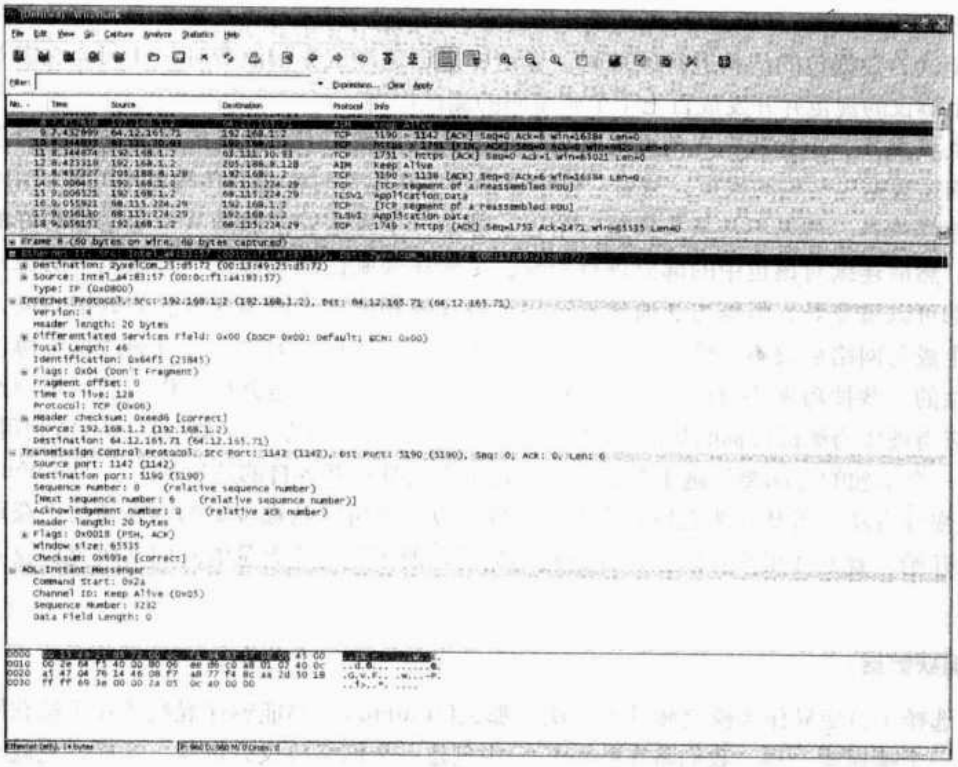


图16-1　Wireshark解析一个使AIM生效的包

16.1.4　模糊变量

一旦网络数据被观测并捕获，我们想要允许用户来确定其中将为模糊测试而进行变异的部分。为此，我们将使用一种非常简单的格式，即使用开始和结束标签将包的十六进制表示部分包含起来。不同的标签用来表示运行时模糊变量的不同类型。这种简单的格式将允许用户可视化的识别模糊变量，并且允许ProtoFuzz在解析包结构时来识别将要被模糊数据所替代的区域。我们将使用下面的标签：

- [XX]强有力模糊变量。被方括号包围起来的字节将使用所有可能的字节值来模糊化。因此，一个单独的字节将被模糊256次，而一个字值（两个字节）将被模糊65 536次。
- <XX>字符串模糊变量。也可以使用用户控制的文本文件中（String.txt）预定义的变量长度字符串的十六进制表示来模糊化字节。这种类型的模糊测试通常在包的数据部分进行，而不是在需要一个已定义结构的头字段中。

下面的模板说明了同时具有强有力模糊变量和字符串模糊变量的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

协议库在发送数据时通常采用两种不同的方法。首先，它们具有一系列的函数允许你设定数据的特定部分，如目标IP和MAC地址。然而，包结构的大部分内容都是基于适当的RFC已经给你构建好的。一个.NET框架类如`HttpRequest`，就是这种类的一个示例。该类允许你定义诸如方法和URL等属性，但大多数的HTTP头和所有的以太网，TCP以及IP头都是已经给你创建好的。另外一种可选的方法是创建一个原始数据包以使单个字节可以被指定，并且由程序员来确保包要遵循定义在适当RFC中的结构。尽管第二种方法需要研究者多做一些工作，但是它提供了更细的控制粒度，将使模糊协议头具有同数据段一样的功能。

16.2 必要的背景知识

在开始讨论ProtoFuzz的开发过程之前，我们首先应当强调一些重要的背景知识。

16.2.1 错误检测

对任何类型的模糊测试而言，在目标应用中检测错误等同于发现漏洞。没有一个完美的方法来实现这一点，但某些方法要优于其他的方法。例如，在协议模糊测试中存在的不同于文件模糊测试的一个障碍是，模糊器和目标应用可能将被分隔为两个系统。当对网络协议进行模糊测试时，将一个调试器关联到目标应用是一个好的开端，它将允许你查明已处理和未处理的异常，否则这些异常可能不会被识别。当使用调试器时，你仍然要面临将模糊包与它们所生成的异常进行关联的挑战。尽管不是十分的安全，但可做的工作包括在每个包之后发送某种形式的探针，以确保目标应用仍然是可响应的。例如，你可以向目标应用发送一个ping请求，并确保在发送下一个模糊包之前接收到一个回应。这不是非常的完美，因为一个异常可能已经发生，但并不影响目标应用响应ping的能力。然而，你可以针对目标应用来定制探针。

性能下降

除了监视错误之外，我们也可以监视目标应用所发生的性能下降问题。例如，考虑一个在目标应用的处理逻辑中导致一个无限循环的病态数据包。在这种情况下，没有实际的错误生成，但是我们却具备了一个DoS条件。在目标机器上运行性能监视器，如在微软的管理控制台中可用的Performance Logs and Alerts或者System Monitor snap-ins，可以帮助识别这些条件。

请求超时和非预期响应

并不是所有被传输的模糊包都将引起一个响应。然而，对大多数包而言都会产生一个响应，监视响应以确保目标应用仍然正常运转是非常重要的。再进一步，也可以对响应包的内容进行解析。采用这种方法，你不仅能够识别何时响应没有被接收，而且还可以识别包含非预期数据的响应。

16.2.2 协议驱动程序

许多包捕获库需要使用一个定制的协议驱动程序。为ProtoFuzz所选择的Metro Packet

Library[⊖]包括ndisprot.inf，它是微软作为其驱动程序开发工具包的一部分而提供的网络驱动程序接口规范（Network Driver Interface Specification,NDIS）协议驱动程序的一个示例。NDIS对于网络适配器而言是一个有效的API，并且为应用程序提供了发送和接收原始以太网包的能力。在ProtoFuzz可以使用之前，该驱动程序必须被手工安装和启动，这可以通过在命令行运行net start ndisprot来实现。使用这样一个需要定制的驱动程序的库的缺点是，该驱动程序可能无法处理所有类型的网络适配器。例如对于Metro而言，该驱动程序将不能处理一个无线适配器。

16.3 ProtoFuzz的开发

现在已经有一些运行良好的协议模糊器。如果我们准备创建一个新的模糊器，那么为了避免开发一个重复的项目，就需要提供其他模糊器所不具备的一些功能。由于许多现有的模糊器都是从命令行开始运行的，因此我们面向下面的目标来使ProtoFuzz不同于其他模糊器：

- *直观性*。掌握ProtoFuzz的使用应当不需要一个长的学习曲线。一个终端用户应当可以掌握该工具的基础用法，而不用手工输入或记住令人费解的命令行选项。
- *易于使用*。一旦ProtoFuzz被启动，它就应当立即开始运转。我们将利用前面所捕获的数据包的结构来创建模糊测试模板，而不需要用户来创建繁琐的模板以定义数据包的结构。
- *可以访问所有的协议层*。一些网络模糊器关注于包内的数据，而不是协议头本身。ProtoFuzz应当能对一个包的任何以及所有部分进行模糊化，包括从以太网头到TCP/UDP数据。

我们创建ProtoFuzz的目标肯定不是要替代现有的工具。它是为在Windows环境中进行网络模糊测试来构建一个基础平台，该平台既可以作为一个学习工具，也可以作为一个开源项目被任何感兴趣的团体来进行扩展。

16.3.1 开发语言的选择

对于我们的基于Windows的文件格式模糊测试工具FileFuzz而言，我们选择了C#和微软的.NET框架来创建GUI驱动的网络协议模糊器。当开发GUI应用程序时，.NET平台可以处理许多繁琐的事务，以使你可以关注于应用程序的业务逻辑，对于我们的情形而言，就是关注于破坏业务的逻辑。.NET应用要求用户首先要安装.NET框架。然而，基于.NET的应用的不断流行也正在使这一点变得有些不方便，因为大多数系统已经具有可用的库。

16.3.2 包捕获库

在创建网络协议模糊器的过程中，所做的一个关键决定就是选择一个合适的包捕获库。该库需要具备三个核心功能：捕获，解析以及传输包。尽管可以从头来创建此功能，但通常并没有建议指出，已经存在一些健壮的，开源的包捕获库。

我们选择在Windows平台上开发ProtoFuzz，就自动限制了我们在该领域的选择。典型的，

⊖ http://sourceforge.net/projects/dotmetro/

针对微软的Windows平台所选择的库将是WinPcap[⊖]，它是一个非常优秀的包捕获库，最初是Piero Viano将libpcap移植到Windows平台中作为其毕业论文的一部分。WinPcap已经存在许多年了，现在它作为开源项目已经发展成一个支持大多数主流协议的强大的代码库。实际上，它对于需要同数据包，如Wireshark（前面所提到的Ethereal）和Core Impact[⊜]，一起工作的许多商业的和开源的应用来说是一个必备条件。WinPcap的Web站点上列出了超过100种利用该库实现包捕获功能的工具，而这只是他们所知道的一些工具。

尽管针对ProtoFuzz而言，WinPcap已经被考虑在内，但使用C#作为开发语言的决定也限制了使用主流库的便利性。WinPcap是用C语言编写的，尽管有一小部分工作试图编写包装程序以简化WinPcap在C#应用中的使用，我们还是看到了一个合并了WinPcap所有功能的完整项目。为WinPcap提供一个包装程序的COM类库PacketX[⊝]也曾经被该项目所考虑过，但最终还是被否决了，因为它是一个商用的库。当撰写此书时，我们希望尽所有的努力来利用免费发布的库。

在经过了一段时间的搜索之后，我们很高兴偶然间发现了Metro包库。尽管它的健壮性不如WinPcap，但该库全部是用C#来编写的，并且具有完整的使用指南和文档，并且对于我们寻找要创建的基础模糊器是足够用的。Metro所具有的一个弱点是受限制的解析能力。它包含有一些类，可以识别出所有高层的包的头——以太网，TCP,UDP,ICMP,IPv4以及地址转换协议（Address Resolution Protocol,ARP）——但它不包含那些可以理解在这些头下面的包中的数据的类。同样，根据ProtoFuzz所要实现的适当的目标，这不是一个主要的考虑问题，由于Metro是一个开源项目，我们总是可以对当前的类进行扩展以开发任何所需要的功能。

16.3.3 设计

我们已经了解了足够的理论知识，下面就可以进行编码了。同样，ProtoFuzz的所有源代码可以从www.fuzzing.org网站上获取。我们不准备分析代码的所有部分，但是在下面的几节中，我们将强调一些非常重要的代码段。

网络适配器

我们希望ProtoFuzz能够捕获将被用来生成模糊模板的通信。为此，用户必须要首先选择将被置为混合模式以嗅探通信的网络适配器。我们不要求用户提供一个适配器名或者他们手工发现的标识符，我们将向用户提供一个简单的下拉菜单，通过这个菜单，他们可以选择系统中任何活动的网络适配器。幸运的是，Metro提供了用以简化该过程的类，如下面的代码片段所示：

```
private const string DRIVER_NAME = @"\\.\ndisprot";
NdisProtocolDriverInterface driver = new NdisProtocolDriverInterface();

try
{
   driver.OpenDevice (DRIVER_NAME);
}
```

⊖ http://www.winpcap.org/

⊜ http://www.coresecurity.com/products/coreimpact/index.php

⊝ http://www.beesync.com/packetx/index.html

```
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 += "To stop it again, type \"net stop " +
      DRIVER_NAME.Substring(DRIVER_NAME.LastIndexOf("\\") + 1) +
      "\" in 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;
}
```

首先，我们初始化了一个新的NdisProtocolDriverInterface，然后使用用户应当已经手工安装的ndisprot网络适配器来调用OpenDevice()函数。如果该适配器不可用，一个SystemException异常将被捕获，用户将被提示安装并启动该适配器。一旦该调用成功地执行完，我们就得到了一个NetworkAdapter数组，该数组包含了所有可用的网络适配器的信息。利用该数组，我们使用一个简单的foreach循环来遍历该数组，并将AdapterName属性添加到组合列表框控件中。

捕获数据

一旦我们打开了一个适配器，就可以将它置于混合模式并开始捕获通信：

```
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", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
```

我们一开始创建了一个二维数组（capturePackets），它包含一个被捕获包的数组以及一个在该包中的字节的数组。然后调用BindAdapter()函数将所选择的网络适配器绑定到前面所初始化的NdisProtocolDriverInterface(driver)。这时，我们在一个单独的线程中调用capturePacket。为此部署一个单独的线程是非常重要的，这样当包被捕获时GUI就不会加锁。

```
private void capturePacket()
{
   while (packetCount < maxPackets)
   {
      byte[] packet = driver.RecievePacket();
      capturedPackets[packetCount] = packet;
      packetCount++;
   }

}
```

为了得到一个被捕获包的字节数组，只需要简单地调用ReceivePacket()函数。然后只要将它们添加到capturePacket数组中即可。

解析数据

因为我们采用两种独立的方法来显示被捕获的包，因此对数据的解析是一个包含两个步骤的过程。为了与Wireshark所使用的设计保持一致，当一个用户选择了一个被捕获的包时，该包内容的一个概要将显示在一个树形控件中，同时，原始字节被显示在一个超文本框控件中。采用这种方法，用户就具有了一个清晰的易于理解的总体概览，但仍然控制着单个字节以用于模糊测试。我们试图通过当树形控件中的一行选项被选中时，在红色文本框中突出显示相关联的原始字节来将两个窗口关联起来。图16-2显示了ProtoFuzz带有一个被捕获包的完整布局。

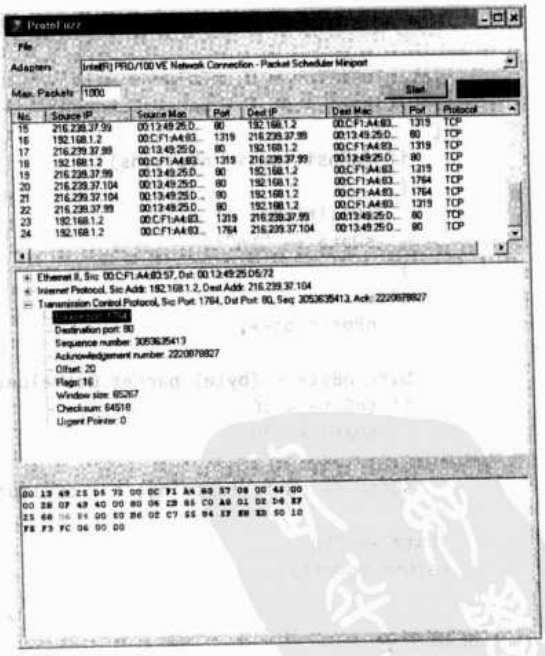


图16-2　ProtoFuzz显示一个被捕获的包

被捕获包的概要视图显示在中间的树形窗格中。创建树形控件的代码包含在packetTvwDecode()函数中，该函数将代码分隔为不同的块以解析下面每个头：以太网，TCP, UDP, IP, ARP和ICMP。为了简单起见，下面的代码片段只说明了对以太网头的解析：

```
Ethernet802_3 ethernet = new Ethernet802_3(capPacket);

strSourceMacAddress = 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);
```

可以看到，Ethernet802_3类使用提供给构造器的capPacket中的字节数组来进行了初始化，并且树形控件中显示的每个节点就是该类的一个属性。下面的代码片段是一个基本的循环，以在一个网格窗格中显示原始包字节。该循环每行打印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);
}
```

模糊变量

对于强制性模糊测试而言，模糊变量通过将字节包围在方括号中（[]）来表达，而对于基于字符串的模糊测试而言，模糊变量则通过将字节包围在尖括号中（<>）来表达。当发送包时，代码简单地读取原始包并寻找成对的括号。当遇到这些模糊变量时，在发送包之前模糊数据将替代原始数据。

十六进制编码和解码

最后一个关心的问题是对被用于显示捕获数据包内容的字节表示进行编码和解码。.NET框架提供了ToString（"X"）函数将一个字节数组转换为一个十六进制字符串，但是它并没有提供类似的函数将一个字符串转换为一个字节数组[⊖]。因为这个原因，所以添加了HexEncoding类，该类的大部分内容借鉴自www.codeproject.com站点中的"Converting Hexadecimal String to/from Byte Array in C#"（用C#将十六进制字符串转换为字节数组，或者从字节数组转换为十六进制字符串）。

16.4 实例研究

现在我们已经构建了一个网络模糊器，下面应该确保它能够正常工作。ProtoFuzz是一个一次执行单个任务的工具，它获取一个包并在将它发送给目标之前重复对它进行变异。例如，在发送导致崩溃的包之前，它不能发送一系列被设计的初始包以使目标进入到一个漏洞状态。例如，考虑在一个SMTP服务器中发生在RCPT TO命令中的一个缓冲区溢出，如下所示：

```
220 smtp.example.com ESMTP
HELO mail.heaven.org
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服务器发送一些请求。我们需要建立起初始的TCP连接，并且为粗体所示的每个HELO，MAIL FROM和RCPT TO命令发送一个请求。初始请求必须要将服务器设置到准备接收RCPT TO命令，并且长字符串最终将导致溢出的状态。类似于此的一个漏洞更加适合于采用如SPIKE这样的工具，允许创建脚本来定义包结构，并且允许发送多个请求。

ProtoFuzz更加适用于这样的情形，即你只想在传输中捕获一个包，选择数据的一个特定部分，然后立即开始模糊测试。可能你正在处理一个私有协议，无法获得充足的信息以生成详细描述协议结构的脚本。另外一种情况是，你可能只想执行一次快速测试，而不是花费时间来描述针对模糊器的协议。

TippingPoint的Zero Day Initiative（ZDI）发现了在HP的Mercury LoadRunner性能测试工具中的一个栈溢出，该漏洞适合于我们的需求[⊖]。该溢出发生在一个代理中，该代理绑定到TCP端口54345并监听引入的连接。该代理在接收包之前不需要任何形式的验证，因此我们就得到所寻找的单一漏洞。另外，该代理所适用的协议是私有协议，并且在安全建议中没有提供足够的详细信息以允许我们从头创建一个对概念的证明。建议所告诉我们的全部信息就是溢出是由server_ip_name字段中的一个超长值触发的。因此，我们将使用ProtoFuzz来捕获合理的通信，识别server_ip_name字段，并据此来变异包。

⊖ http://www.codeproject.com/csharp/hexencoding.asp

⊖ http://www.zerodayinitiative.com/advisories/ZDI-07-007.html

一旦我们安装了Mercury LoadRunner，并正确地进行配置之后，我们就可以开始搜索一个可能包含漏洞的事务。当对通信进行嗅探时，我们识别了下面的包，该包包含二进制和ASCII可读文本的组合，并且明显地包含我们所寻找的erver_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=
```

在包被捕获之后，下面将强调包含在server_ip_name字段中的值，点击鼠标右键，然后选择模糊——字符串。这将所选择的字节包含在一个尖括号中（<>）。当ProtoFuzz解析被捕获的包时，它将用Strings.txt文件中所提供的每个值来替换所识别的字节。因此，为了触发溢出，我们需要向文件中添加连续的更大的字符串。单独的模糊包将会为String.txt文件中的每一行而发送。

当模糊数据被发送到我们的目标时，我们监视异常的发生是非常重要的。Magentproc.exe是绑定到TCP端口54345的应用程序，因此我们需要将一个调试器关联到该进程并开始进行模糊测试。一旦崩溃被触发，那么可以在下面的代码片段中看到来自于OllyDbg的输出：

```
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
```

类似于上一章中所提到的NetMail示例，这也是一个远程可利用的栈溢出的典型例子。由于服务不需要进行任何验证，因此也就增加了由该漏洞引起的在一个流行的质量保证测试工具中的风险。

16.5　益处和改进的余地

目前，ProtoFuzz还不能处理一系列字节前面带有数据大小的数据块。这种情况下，对数据

字节进行模糊测试需要对数据块的大小进行并行的更新，以使包能够保持有效性。这种能力将会极大地扩展ProtoFuzz的功能，并且实现起来相当简便。

ProtoFuzz目前还没有提供探查功能，以来确定在模糊测试过程中，目标主机是否一直可用。正如本章前面所提到的，探针包可以在一个模糊包之前发送，以确定目标应用程序的状态。如果目标应用没有响应，那么模糊测试就会停止，因为后续的测试用例将没有任何用处。当试图将导致在目标应用中发生异常行为的模糊包进行关联时，这个特性也可以对终端用户提供帮助。ProtoFuzz也可以通过进行扩展来受益，即允许它处理复杂的请求——响应对。除了能够简单地发送模糊包之外，ProtoFuzz还可以在模糊测试之前将目标应用置于一个特殊的状态，方法是通过首先发送已知的好包，然后等待适当的响应，并解析和解释其内容。

可以通过设计一个位于目标机器上的远程代理来监视目标应用的状态，从而进一步提高错误检测功能。该代理可以和ProtoFuzz进行通信，并且在发现一个错误时停止模糊测试。这有助于直接确定是哪个包导致了错误的发生。

16.6 小结

在ProtoFuzz成为一个健壮的模糊器之前，还需要进行长期的开发完善过程。然而，它起到了说明网络协议模糊测试的基础框架的作用，并且易于被扩展到特定的目标应用。创建这个网络模糊器的关键因素是发现一个可以处理原始包的捕获，解析以及传输的库。避免使用那些不允许原始包被组装的库，除非你正在为一个特定的、不需要访问任何包的头的网络协议创建一个模糊器。出于我们的考虑，Metro库为ProtoFuzz提供了一个强大的基础库，并且.NET框架允许我们使用一个直观的GUI前端来创建它们。我们鼓励你来创建ProtoFuzz，并且将它应用于不同的方面。

第17章 Web浏览器模糊测试

"我经常发现的一个分母是期望值比期望的还要大"

——George W. Bush, Los Angeles, 2000年9月27日

客户端漏洞很快成为了关注的焦点，因为它们被攻击者广泛利用以进行钓鱼式攻击、身份的窃取并且创建大量的被恶意代码感染并控制的计算机网络（botnets）。Web浏览器中的漏洞为这样的攻击提供了丰富的目标环境，因为在主流浏览器中的一个漏洞将导致数百万人受其影响。客户端攻击通常要求运用某种形式的社会科学，因为攻击者必须首先要强制一个潜在的被攻击者去访问一个恶意的Web页。通常可以通过垃圾邮件或利用主流Web站点中的一个额外漏洞来帮助实施该过程。再加上Internet上的单个用户通常具有很少的计算机安全方面的知识，因此许多攻击者将关注的重点转向客户端漏洞也就不足为奇了。

在许多方面，浏览器错误成为最引人注目的漏洞是在2006年。所发现的漏洞影响了所有主流的Web浏览器，包括微软Internet Explorer和Mozilla Firefox。缺陷在解析病态的HTML标签、JavaScript、本地图像文件如JPG、GIF和PNG、CSS以及ActiveX控件时被发现。许多被发现的严重的ActiveX漏洞既影响了微软Windows操作系统的默认安装，也影响了第三方的应用程序。另外，人们还发现了成百个不严重的ActiveX漏洞。尽管安全研究者在过去对ActiveX和COM的审核表现出了一定的兴趣，但在2006年这一兴趣达到了最高程度。存在于大量ActiveX控件中的漏洞的一个重要因素是新的、用户友好的ActiveX控件审核工具的公共可用性。本章中，我们将讨论如何使用模糊测试来发现Web浏览器漏洞。如果将历史结论作为指导的话，那么还将有大量的浏览器错误被发现。

17.1 什么是Web浏览器模糊测试

Web浏览器最初只是被设计为浏览Web页以及解析HTML页，现在Web浏览器已经发展成为和瑞士军刀的多功能性相等价的计算机设备。现代的Web浏览器可以处理动态HTML页、类型表单、多种脚本语言、Java applet、在线共享内容的一种简易方式（RSS）以及FTP连接等多种目标。可以进行许多扩展以及插件的安装，以将Web浏览器转换为一个更加灵活的应用程序。这种灵活性允许像Google这样的公司再次将一些通用的桌面应用程序转向Web应用。但对于终端用户来说，不幸的是，在增加功能的同时也增加了攻击的范围。毋庸置疑，随着越来越多的功能被增加到Web浏览器中，将会有越来越多的漏洞被发现。

浏览器错误的发布月

2006年7月，H.D.Moore发布了"浏览器错误的发布月"（Month of Browser Bugs,

MoBB）[⊖]。在整个7月当中，H.D.每天发布一个新的基于浏览器的漏洞，直到该月结束他一共发布了31个漏洞的详细信息。尽管这些漏洞的绝大部分都影响微软的Internet Explorer，但所有主流的浏览器都被提出，包括Safari、Mozilla、Opera和Konqurer[⊜]。由于存在像空指针解除引用这样的问题，因此大多数的漏洞只是通过使Web浏览器崩溃来导致一个DoS攻击。尽管这些漏洞的严重性不是很高，而就在MoBB之前不久，Skywing和skape（与H.D.Moore一道为Metasploit项目做出了贡献）发现了在Internet Explorer中未处理异常过滤器中的一个设计缺陷，该缺陷可以利用未开发的漏洞以导致在特定的条件下发生代码执行[⊜]。这就潜在地增加了在MoBB过程中发现某些漏洞的风险。尽管某些人不同意在发布补丁之前揭露漏洞详细信息的方法，但它使人们看到了Web浏览器漏洞的盛行，以及由这些漏洞所带来的风险。

17.2 目标

每当一项研究指出Firefox是一个比Internet Explorer更加安全的浏览器，那么就会有一个新的带有偏见的报告被发布并提出相反的结论。从一个研究者的视角来看，它是一个尚处于争论中的观点。所有的Web浏览器都有严重的漏洞，而模糊测试则是识别这些漏洞的一个有效方法。对于任何类型的软件来说，该应用程序的应用越广泛，就会有越多的人直接受到漏洞的影响。尽管统计数据一直在发生变化，也不论你使用的数据来源是什么，你可能将会发现Internet Explorer的使用占据了终端用户的最大份额，而Firefox仍然是强有力的第二位的竞争者。自然的，Internet Explorer的流行源自于它被默认包含在一个操作系统中，而该操作系统居于全世界PC操作系统的统治地位。尽管如此，Firefox继续让Internet Explorer运行。其他的Web浏览器如Netscape, Opera, Safari和Konqueror，通常占据终端用户的很小一部分份额，并且联合起来处于第三的位置。

17.3 方法

我们有两种方法可以对Web浏览器进行模糊测试。首先我们必须要确定一种方法来控制模糊测试，然后我们还要确定所要重点关注的浏览器的那些部分。由于Web浏览器的功能在不断增加，因此获得完整的代码覆盖可能会成为一项艰巨的工作。我们期望使用多种工具和方法来执行一个完整的审核。

17.3.1 测试方法

当对Web浏览器进行模糊测试时，有一些基本的方法可供我们选用，并且所用的单个方法要依赖于被进行模糊测试的浏览器的部分。通常情况下，下面的方法是可用的：

- 刷新HTML页。在这种场景下，被模糊的Web页由一个Web服务器所生成，它也包含一种

⊖ http://browserfun.blogspot.com/

⊜ http://osvdb.org/blog/?p=127

⊜ http://www.uninformed.org/?v=4&a=5&t=sumry

方法使得Web页在正则区间内被刷新。可以通过使用不同的方法来实现这一点，如刷新HTTP-EQUIVMETA标签或某些客户端脚本。当Web页被刷新之后，一个包含有新的模糊内容的Web页的新版本将会出现。Mangleme[⊖]是一个模糊测试工具，被设计为用来发现Web浏览器中的HTML解析缺陷，例如，包括如下的在每个模糊化Web页的HEAD部分中的行：

```
<META HTTP-EQUIV=\"Refresh\" content=\"0;URL=mangle.cgi\">
```

这个标签将导致mangleme页立即被更新，并重定向到mangle.cgi，在那里有一个新的被破坏的页正在等待Web浏览器。下面的客户端JavaScript可以用来实现一个类似的目标，即在2 000毫秒之后刷新一个Web页：

```
<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Web页。模糊测试可以只在客户端进行，方法是通过直接将一个模糊页加载到Web浏览器中。使用命令行选项，可以指示一个目标Web浏览器打开存在于本地机器上的一个模糊化Web页，下面的命令将直接在Internet Explorer中打开文件fuzz.html：

```
C:\>"C:\Program Files\Internet Explorer\iexplore.exe" C:\temp\fuzz.html
```

- 目标单个浏览器对象。在特定条件下，根本不部署浏览器而执行基于浏览器的模糊测试是可能的。当我们的目标是一个浏览器对象，并且浏览器只是访问该对象的一个工具时，就可以实现这一点。例如，我们可以对一个ActiveX控件进行模糊测试，如果该对象被标记为“脚本安全”，那么就可以通过一个恶意的Web站点来远程访问该控件。在这种情况下，就不需要从浏览器内部启动ActiveX控件来确定它是否包含漏洞。David Zimmer设计的COMRaider[⊜]就是使用这个方法来对ActiveX控件进行模糊测试的。

17.3.2 输入

相对于其他的测试目标，对Web浏览器进行模糊测试要求我们识别不同类型的用户输入。

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

⊜ http://labs.idefense.com/software/fuzzing.php#more_COMRaider

同样，我们必须要跳出以前关于输入的组成部分的思考圈子。在Web浏览器的情形下，攻击者是Web服务器，它所传输的内容接着被Web浏览器所解释。因此，Web服务器所发送的所有内容构成了输入，而不只是HTML代码。

HTML头

Web浏览器所接收的第一个数据片段就是HTML头。尽管它不被浏览器直接显示，但是该头包含有重要的信息可以影响Web浏览器的行为，以及它显示后续内容的方式。

这种类型漏洞的一个例子可以在微软的安全公告MS03-015[⊖]所包含的CVE-2003-0113[⊜]中看到。Jouko PynnÖnen发现它是由于urlmon.dll中的不正确边界检查而引起的，内容类型以及内容编码头中的长值[⊜]可以导致一个基于栈的缓冲区溢出。已经发现Internet Explorer5.x和6.x的许多版本都易于受到这种类型的攻击。

HTML标签

Web页本身的内容以一系列HTML标签的形式发送给Web浏览器。从技术上来说，一个浏览器可以显示不同类型的标记语言如HTML、XML以及XHTML，但为了实现我们的目的，这里将Web页的内容简单地作为HTML。

HTML由于其不断扩展的功能，使其变成了一个日渐复杂的标准。另外，许多私有标签被特定的Web浏览器所使用，并且Web浏览器可以解释多种类型的标记语言，当设计一个Web浏览器时你开始看到有许多产生异常的机会。这些编程错误进而可以通过模糊测试来加以识别。

HTML具有相对简单的结构。有一些HTML标准确定了Web页应当遵循的适当的标准，一个被正确格式化的Web页应当以一个文档类型声明（Document Type Declaration,DTD）作为开始。例如，下面的DTD将告知浏览器Web页应当遵循HTML 4.01标准：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

当我们考虑HTML时，我们通常认为标签定义了文档的内容。这些标签被认为是元素，并且通常包含起始和结束标签。除了这些标签本身之外，元素也可以包含属性和内容。下面的HTML元素说明了一个字体元素：

```
<font color="red">Fuzz</font>
```

在这个特殊的元素中，color代表了font元素的一个属性，而Fuzz是该元素的内容。尽管这是一个非常基础的例子，但是需要指出的很重要的一点是HTML代码的所有部分都可以成为模糊测试的合理目标。Web浏览器被设计用来解析和解释HTML代码，如果代码处于一种非预期的格式，那么开发者可能不会针对一个异常的HTML代码片段提供适当的错误处理机制。

mangleme被设计为通过Web浏览器解释病态HTML标签来发现漏洞，它导致了在微软安全

⊖ http://www.microsoft.com/technet/security/bulletin/MS03-015.mspx

⊜ http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0113

⊜ http://downloads.securityfocus.com/vulnerabilities/exploits/urlmon-ex.pl

公告MS04-040[㊀]中所提到的CVE-2004-1050[㊁]的发现。Ned和Berend-Jan Wever使用mangleme发现了在IFRAME,FRAME和EMBED中的超长SRC或NAME属性，而这些超长属性可能会导致一个基于堆的缓冲区溢出。然后，他们就发布了针对该问题的完整的公共利用代码[㊂]。其他被设计为在Web浏览器中对HTML结构进行模糊测试的工具包括由H.D.Moore和Aviv Raff所开发的DOM-Hanoi[㊃]和Hamachi[㊄]工具。

XML标签

XML是源自于标准通用标记语言（Standardized General Markup Language,SGML）的一个通用标记语言，并且通常被用来通过HTTP协议来传输数据。XML的标准实现被开发为定义了成百上千个数据格式，如RSS、应用程序漏洞描述语言（Application Vulnerability Description Language,AVDL）以及可升级的向量图形（Scalable Vector Graphics,SVG）等等。通过使用HTML，现代的Web浏览器可以解析并解释XML元素及其相关的属性和内容。同样，如果当浏览器在解释元素时遇到了非预期的数据，那么可能就会导致一个未处理的异常。向量标记语言（Vector Markup Language,VML）是一个用来定义向量图形的XML规范。已经在Internet Explorer和Outlook所使用的库中发现了漏洞，该漏洞是由于无法处理特定的病态VML标签而导致的。2006年9月19日，微软发布了925568号安全建议[㊅]，详细描述了在向量图形展现引擎（vgx.dll）中的一个基于栈的缓冲区溢出，该溢出可以导致远程代码的执行。该建议发布之后，就出现了对该漏洞的大量利用，这就迫使微软在7天之后发布了一个周期之外的补丁[㊆]。在同一个库中的整数溢出漏洞于2007年被发现并打上了补丁[㊇]。

ActiveX控件

ActiveX控件代表了构建在微软COM上的一个私有的微软技术，用来提供可重用的软件构件[㊈]。Web开发者通常嵌入ActiveX控件以允许对功能进行扩展，而这对于一个典型的Web应用来说可能是不允许的。使用ActiveX控件的一个主要缺点是它们所具有的私有特性限制了它们在Internet Explorer Web浏览器和Windows操作系统中的使用。然而，它们在许多Web站点上被广泛使用。ActiveX控件的功能非常强大，并且一旦它们得到信任，就可以对操作系统进行完全的访问，因此除非它们来自于一个可信的渠道，否则不要运行ActiveX控件。

㊀ http://www.microsoft.com/technet/security/bulletin/ms04-040.mspx

㊁ http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-1050

㊂ http://downloads.securityfocus.com/vulnerabilities/exploits/InternetExploiter.txt

㊃ http://metasploit.com/users/hdm/tools/domhanoi/domhanoi.html

㊄ http://metasploit.com/users/hdm/tools/hamachi/hamachi.html

㊅ http://www.microsoft.com/technet/security/advisory/925568.mspx

㊆ http://www.microsoft.com/technet/security/Bulletin/MS06-055.mspx

㊇ http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=462

㊈ http://en.wikipedia.org.wiki/ActiveX_Control

在包含在Windows操作系统或第三方软件包中的ActiveX控件中已经发现了大量的漏洞。如果这些控件被标识为“初始化安全”和“脚本安全”，那么它们就可以通过一个远程的Web服务器来访问，因此就可以被利用来危及目标机器的安全[⊖]。这就造成了一个危险的情形，因为用户可能会从一个可信的渠道来初始化安装一个ActiveX控件，但如果它后来被发现包含有一个漏洞如缓冲区溢出，那么同样可信的控件将被一个恶意的Web站点所利用。

COMRaider是一个非常优秀的示例驱动的模糊测试工具，可以用来识别目标系统上ActiveX控件中的潜在漏洞。另外，它还可以过滤模糊化的ActiveX控件，而只针对那些可以被一个恶意Web服务器访问的ActiveX控件，因为这些控件极大地增加了一个漏洞所带来的风险。COMRaider对于在一个给定的系统中的所有ActiveX控件执行完整的审核是非常有用的。它将会首先识别所有的ActiveX控件及其相关的方法和输入。然后，它将会连续的对所选择的ActiveX控件进行模糊化。COMRaider包含一个内嵌的调试器，因此它可以识别已处理和未处理的异常。COMRaider同时还具有分布式审核的功能，允许一个团队将其以前的审核结果进行共享。图17-1显示了COMRaider在对一个完整系统中标记为脚本安全的所有ActiveX控件执行完审核之后所生成的一个列表。

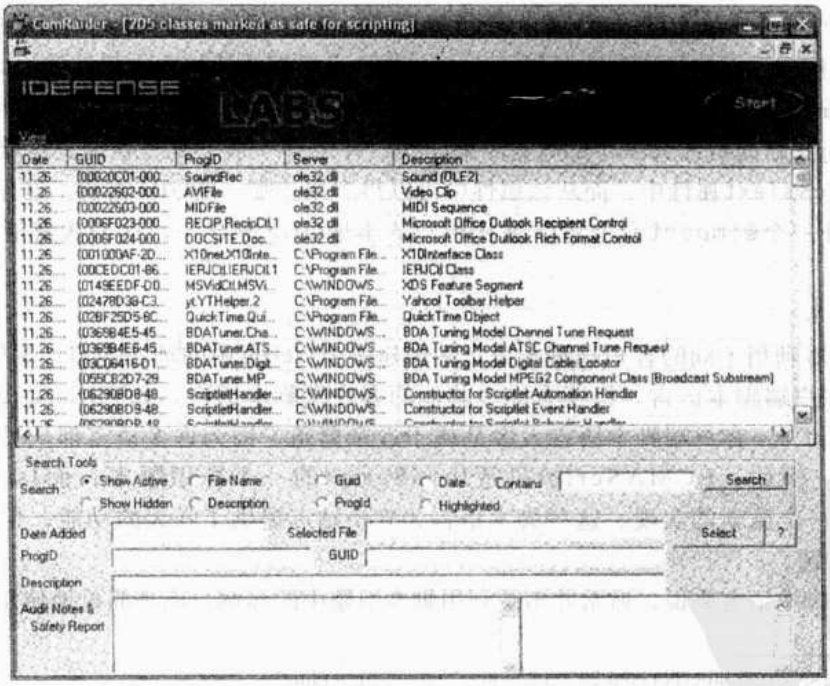


图17-1 COMRaider

AxMan[⊜]是另外一个免费可用的ActiveX模糊器。AxMan是由H.D.Moore设计的，被用来发

⊖ http://msdn.microsoft.com/workshop/componets/activex/safety.asp

⊜ http://metasploit.com/users/hdm/tools/axman/

现在2006年7月浏览器错误发布月所发布的大部分ActiveX漏洞[㊀]。

层叠排列类型表单

CSSDIE[㊁]是H.D.Moore,Matt Murphy,Aviv Raff和Thierry Zoller所创建的一个CSS模糊器，被用来在浏览器错误发布月中发现Opera浏览器中的一个内存破坏问题[㊂]。它发现将一个DHTML元素的CSS背景属性设置为一个超长的URL将会导致一个客户端DoS，进而导致浏览器崩溃。

其他基于CSS的问题都导致了不同的漏洞，如MS06－021[㊃]中的CVE-2005-4089，该漏洞允许一个攻击者绕过Internet Explorer中的跨域限制。这种情况是可能发生的，因为站点可以使用@import命令从其他不正确的CSS文件域中下载文件。hacker.co.il的Matan Gillon说明了如何利用该漏洞来窥探某些人的Google桌面搜索（Google Desktop Search,GDS）结果[㊄]。尽管Google针对这个问题做了更改，使得该攻击不再有效，但它仍然是一个非常好的例子，说明了一个简单的Internet Explorer漏洞如何能够同其他应用程序（在这里是GDS）的功能相结合，来执行一个相对来说更为复杂的攻击。在一个CSS中，一个特定元素的格式被定义在一组花括号中，其中的属性和值用冒号隔开。例如，一个anchor标签可以被定义为{color:white}。如果使用@import命令将一个CSS之外的一个文件导入，那么Internet Explorer将试图把该文件作为一个CSS来解释，并且该文件中位于一个左开花括号后面的任何内容将被使用cssText属性来检索。Matan可以利用这个攻击来为一个访问Web页的用户窃取唯一的GDS密钥，方法是通过首先将Google新闻页作为一个CSS文件来导入，并带有一个“}{”查询。这就导致了查询结果被包含在cssText属性中，而从该属性中可以得到用户唯一的GDS密钥。这样，攻击者就可能使用另外一个@import命令将结果从用户的本地GDS中导入，这一次就能够窥探本地GDS索引。

客户端脚本

Web浏览器利用不同的客户端脚本语言来创建一个Web页中的动态内容。尽管JavaScript是最常用的客户端脚本语言，但还有许多其他的客户端脚本语言，如VBScript，Jscript[㊅]以及ECMAScript[㊆]。客户端脚本语言大多是属于近亲繁殖，因为许多语言都从另外一种语言发展而来的。例如，ECMAScript只是JavaScript的一个标识版本，而Jscript则只是JavaScript的一个微软的实现。这些脚本语言为Web站点提供了强大的功能，但同时也导致了许多安全问题。

对客户端脚本语言来说，通常并不是利用脚本引擎中的漏洞，而是简单地使用脚本语言来

㊀ http://browserfun.blogspot.com/2006/08/axman-activex-fuzzer.html
㊁ http://metasploit.com/users/hdm/tools/see-ess-ess-die/cssdie.html
㊂ http://browserfun.blogspot.com/2006/07/mobb-26-opera-css-backgroung.html
㊃ http://www.microsoft.com/technet/security/Bulletin/MS06-021.mspx
㊄ http://www.hacker.co.il/security/ie/css_import.html
㊅ http://en.wikipedia.org/wiki/Jscript
㊆ http://en.wikipedia.org/wiki/Ecmascript

访问其他包含漏洞的构件如一个ActiveX控件。这并不是说在脚本引擎中不存在漏洞。内存破坏问题相对来说更为普遍，例如在Internet Explorer中使用JavaScript来迭代一个本地函数所导致的空引用[㊀]。浏览器错误发布月使用下面这一行JavaScript代码说明了该漏洞：

```
for (var i in window.alert) {var a = 1;}
```

由Azafran所发现的在Firefox JavaScript引擎中的一个漏洞导致了一个信息泄露问题，这将允许一个恶意的Web站点远程检索任意堆内存的内容[㊁]。该漏洞是由处理λ表达式的replace()函数所导致的。尽管攻击者不能够控制所接收的内存，但它可能包含一些敏感的信息，如密码等。

堆的填充

通常情况下，客户端脚本语言并不被用做一个攻击手段；而是用于促进一个单独的包含漏洞的浏览器构件的开发。对浏览器漏洞的利用是由内存破坏或悬摆指针的使用而引起的，例如，通常是在一个客户端脚本语言的帮助下来进行利用。在漏洞利用的过程中，JavaScript可以用于不同的任务，并且通常利用在基于堆的溢出中。由于其具有应用的广泛性和跨平台的可用性等特性，使其成为了一个显而易见的选择。考虑你发现了一个错误，它导致了你能够控制的一系列解除引用。假定在下面的例子中，你可以完整地控制寄存器EAX：

```
MOV EAX, [EAX]
CALL [EAX+4]
```

针对EAX的大多数值将导致这两条指令引发一个访问违规，并使浏览器崩溃。为了成功地将控制流进行重定向，我们所提供的地址必须是一个指向有效代码的有效指针，它只是我们可以容易并可靠地在一个正常的进程空间中所发现的某些内容。然而，我们可以使用JavaScript来连续的分配大块的堆数据，并且操纵内存空间使其处于更加易于受到攻击的状态。因此，客户端JavaScript经常被用做使用一个NOP sled和shellcode来填充堆，以增加当进入堆时对shellcode进行破坏的机会。NOP表示在组装代码时的一个无操作，它只是不执行有效动作的一个操作或一系列操作。当执行被转换到一个NOP sled时，执行将会继续迭代NOP sled，而不改变寄存器或其他内存位置。Berend-Jan Wever（a.k.a.SkyLined）使用其Internet Explorer代码开发了一个流行的堆溢出技术[㊂]，该技术使用0x0D值来对堆进行溢出。这个值的选择是非常重要的，因为它包含有双重含义。首先，它代表一个5字节的类NOP指令，等价于OR EAX 0D0D0D0D。另外，它还可以是一个自引用的指针，指向一个有效的内存地址。它被解释的方式依赖于它所引用的内容。

考虑图17-2所示的来自于OllyDbg Heap Vis[㊃]插件的屏幕快照[㊄]。该屏幕快照说明了如何使

㊀ http://browserfun.blogspot.com/2006/07/mobb-25-native-function-iterator.html

㊁ http://www.mozilla.org/security/announce/2005/mfsa2005-33.html

㊂ http://www.milw0rm.com/exploits/930

㊃ http://www.openrce.org/downloads/details/1

㊄ http://pedram.openrce.org/images/olly_heap_vis/skylined_ie_heap_fill.gif

用该插件来帮助把在基于堆填充利用时的Internet Explorer的内存状态进行可视化。

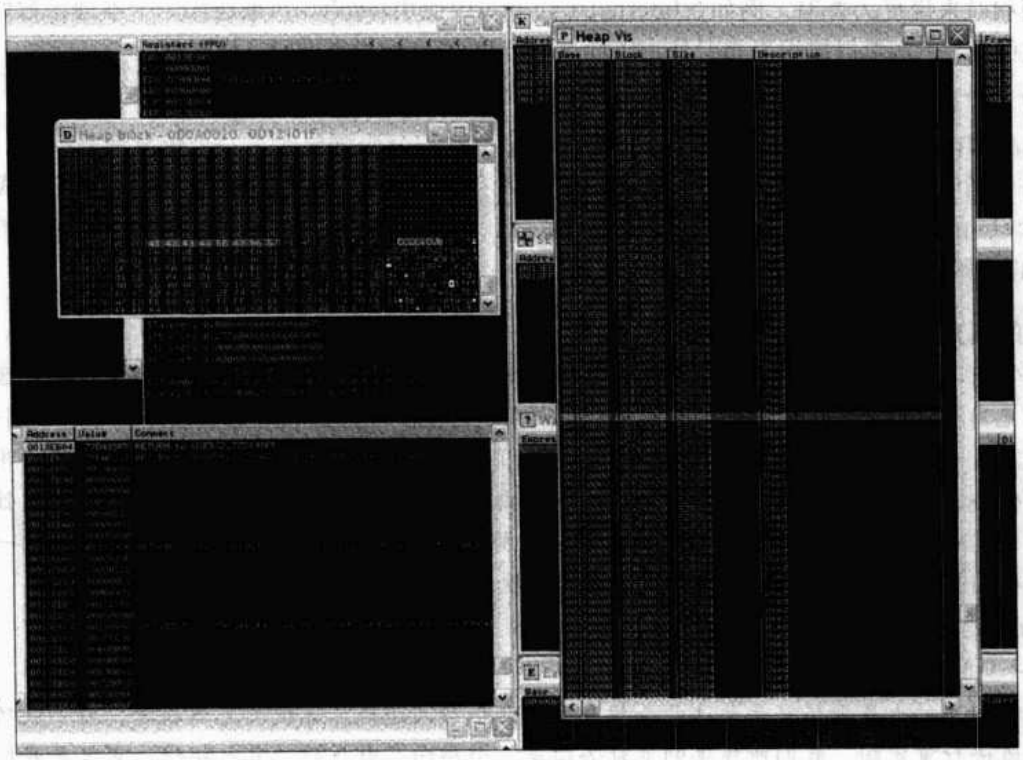


图17-2　SkyLined的OllyDbg Heap Vis的堆填充运行图

屏幕快照右上角标记为“Heap Vis”的窗格显示了所分配的堆块的列表。高亮显示的条目表示了从地址0x0D0A0020开始的大约500Kb大小的一个块。这就是包含地址0x0D0D0D0D的堆块。屏幕快照左上角标记为“Heap Block-0D0A0020..0D12101F”的窗格显示了高亮块的一部分特殊内容，即从某些嵌入的shellcode开始到0x0D字符串副本结束之间的内容。根据前面所示的解除引用的两行代码的例子，假定我们将EAX寄存器的值设置为0x0D0D0D0D。第一次解除引用MOV EAX,[EAX]将使EAX保持相同的值，因为包含在目标地址的字节全部是0x0D。下一条指令CALL [EAX+4]将使控制权转换到地址0x0D0D0D11，这就进入了一个0x0D长序列的中间。该序列将被处理器作为等价的NOP指令来执行，直到达到了shellcode，如屏幕快照所示。为堆溢出所选择的值0x0D0D0D0D利用了这样一个事实，即堆填充通常是从较低的地址开始，然后逐渐向较高的地址填充。选择一个以0开始的地址意味着在到达目标地址之前，需要花费较少的时间来填充堆。而选择一个较高的地址值如0x44444444则需要花费较长的时间来填充堆以到达目标地址，并且还会增加目标用户在成功利用之前对浏览器进行破坏的机会。

进一步考虑下面的两个“坏”选择：一个是被翻译成ADD [ECX],EAX的0x01010101，另一个是被翻译成OR CL,[EDX]的0x0A0A0A0A。第一个例子可能会执行失败，因为它对包含在

ECX寄存器中的地址进行了无效的写操作，而第二个例子也会执行失败，因为它对包含在EDX寄存器中的地址进行了无效的读操作。作为另外一个好的例子，考虑将被翻译成ADD EAX, 0x05050505的0x05050505。该序列在某些公开可见的开发中被使用。

Flash

尽管Adobe Flash Player代表了一个第三方Web浏览器所附加的一个构件，而不是浏览器所内嵌的一个功能构件，但它已经被如此广泛地所接受，并且当前的大部分Web浏览器都安装有某些版本的Flash Player。二进制的Flash文件通常具有一个.swf扩展名，并且可以作为单机文件来打开，但更普遍的情况是作为浏览器所下载的Web页中的对象，然后在Flash Player中被启动。由于Flash文件所具有的二进制特性，当执行模糊测试时可以采用两种可选方法中的一种。可以像第11章“文件格式模糊测试”，第12章“文件格式模糊测试：UNIX平台上的自动化测试”和第13章“文件格式模糊测试：Windows平台上的自动化测试”中所描述的那样，对一个已知的好的.swf文件进行模糊测试，这几章中详细描述了文件格式模糊测试技术。2005年11月，eEye发布了在Macromedia Flash6和Flash7中的一个内存访问漏洞的详细信息。尽管不知道该漏洞是如何被发现的，但模糊测试是一种可能会被采用的方法。另外，Flash文件可以使用ActionScript来进行编码，而ActionScript是Flash用来在运行时操纵数据和内容的一种脚本语言[⊖]。然后就可以使用模糊测试在编译二进制Flash文件之前来变异不同的ActionScript方法。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()方法来向一个由Flash对象做出的请求添加任意的HTTP头[⊜]。通过一个HTTP请求分割技术，就可以使用该方法来执行任意的HTTP请求。

URLs

在某些时候，URL本身也可以导致漏洞。在MS06-042[⊛]发布之后，eEye发现针对该公告新发布的补丁也导致了一个堆溢出漏洞。他们发现，当一个超长的URL传递给Internet Explorer时，如果目标Web站点被表示为GZIP或者缩减编码，那么当一个LstrcpynA()调用试图将一个2 048字节的URL拷贝到一个260字节的缓冲区时，就会发生一个溢出[⊜]。使用该漏洞的一个攻击将只要求用户点击一个提供的URL，并且假定他们正在使用Internet Explorer的一个包含漏洞的版本。

17.4 漏洞

尽管客户端漏洞要求至少具备某些社会学知识以有助于执行一个成功的攻击，但由于它们可以导致的攻击的类型，使得它们仍然带来了非常严重的风险。

- DoS。许多Web浏览器漏洞只是简单地导致一个DoS攻击，以使浏览器崩溃或变得无法响应。导致这种情况的发生，通常是由于一个无限循环或不能被进一步利用的内存破坏。

⊖ http://en.wikipediaorg/wiki/Actionscript

⊜ http://www.rapid7.com/advisories/R7-0026.jsp

⊛ http://www.microsoft.com/technet/security/bulletin/ms06-042.mspx

⊜ http://research.eeye.com/html/advisories/published/AD20060824.html

从主要的方面来看，一个客户端DoS攻击是非常次要的。尽管在每次Web浏览器崩溃之后它都会令人讨厌的去重启浏览器，但它并不会导致任何永久的损害。不像服务器端DoS攻击，客户端DoS攻击只有一个单一的攻击目标。

- 缓冲区溢出。缓冲区溢出是相对来说更为普遍的Web浏览器漏洞。事实上，它可以被前面所提到的任何输入向量所触发，并且具有很特别的损害，因为它可以导致代码执行。
- 远程命令执行。命令执行漏洞通常利用了已经存在的功能，即此功能并不是被设计为允许远程代码执行，但实际上它允许代码执行。例如，Albert Puigsech Galicia发现一个攻击者可以直接将FTP命令注入到一个FTP URI中，通过简单地迫使用户点击该链接，而导致Internet Explorer6.x及其以前的版本去执行FTP命令[⊖]。例如，可以利用该漏洞下载文件到用户的计算机上。后来进一步发现，同一个漏洞还可以被利用以导致浏览器发送邮件消息。微软在MS06－042中描述了该漏洞。
- 绕过跨域限制。Web浏览器具有这样的控制功能，即阻止一个特定的Web站点访问其他站点的内容。这是一个非常重要的限制，否则的话，例如任何站点将能够从其他站点获取通常包含任务ID的cookie。目前出现了大量的漏洞，这些漏洞允许一个站点摆脱这个域限制。前面所提到的GDS漏洞就是此类问题的一个例子。
- 绕过安全区。Internet Explorer根据内容的起始区域来加强安全性。来源于Internet的文档通常被认为是不可信的，因此对它们施加了更为严格的限制。另外，在本地打开的文件被认为是可信的，并被授予了更大的特权。2005年2月，Jouko PynnÖnen发布了一个建议，详细描述了如何使用经过特殊编码的URL来诱使Internet Explorer解释远程文件，就好像是从本地区域中所打开的一样[⊜]。接下来，这就允许一个攻击者在下载的文件中包含恶意脚本，以执行一个攻击者所提供的利用。微软在安全公告MS05－14中描述了此漏洞[⊝]。
- 地址栏欺骗。钓鱼式攻击已经变成了一个严重的问题，因为有越来越多的攻击者试图从可信的Web站点上获取私人信息，如信用卡号等。尽管大多数的钓鱼式攻击只是简单利用一些社会学知识，但是一些更加高级的攻击利用Web浏览器的漏洞帮助来使进行钓鱼式攻击的站点看上去是合法的。允许地址栏被欺骗的漏洞对于钓鱼式攻击来说是具有价值的，因为它们允许一个伪造的Web页看上去好像是位于一个合法站点上。不幸的是，在所有主流的Web浏览器中已经存在一些这样的漏洞了。

17.5　错误检测

当对Web浏览器进行模糊测试时，开展并行的工作以识别不明显的错误也是非常重要的。不能只在一个单一的地方寻找错误。相反，研究一些不同的错误来源是值得的。

⊖ http://osvdb.org/displayvuln.php?osvdb_id=12299

⊜ http://jouko.iki.fi/adv/zonespoof.html

⊝ http://www.microsoft.com/technet/security/bulletin/ms05-014.mspx

- 事件日志。如果你正在Windows环境中对Web浏览器进行模糊测试，那么不要忽略事件查看器。实际上，Internet Explorer7.0将一个单独的Internet Explorer日志添加到了事件查看器。如果你正在对一个早期版本的Internet Explorer或另外的浏览器进行模糊测试，那么条目将会被包含到应用程序的日志中。尽管它不是很完整，但它是一个便于检查的来源，并且可能包含有用的数据。
- 性能监视器。内存破坏问题或无限循环都将可能导致目标Web浏览器性能的下降。性能监视工具可以帮助识别这样的情形。然而，当进行模糊测试时，要确保使用一个专用机器，以保证其他因素不会影响性能下降。
- 调试器。到目前为止，对Web浏览器的模糊测试而言，最有用的错误检测工具就是一个关联到被测试浏览器的第三方调试器。这将允许你识别已处理和未处理的异常，同时也将有助于确定内存破坏问题是否可能会被利用。

17.6 小结

尽管客户端漏洞曾一度被忽略，但钓鱼式攻击的出现迫使我们根据它们所带来的风险重新对其进行关注。在协作网络上的一个单一的含有漏洞的浏览器可以成为攻击者的一个网关。客户端攻击至少需要具备一些社会学知识，但相对来说它是一个次要的障碍。Web浏览器漏洞也导致了在以浏览器为目标的攻击中的很高明的攻击手段。在本章中，我们介绍了一些现有的Web浏览器模糊测试工具。在下一章中，我们将为Web浏览器构建我们自己的模糊器。

第18章 Web浏览器的模糊测试：自动化

“天然气存在于半球中。我喜欢将其称之为半球是因为它是一种我们可以在邻国发现的产品。”

——George W. Bush，华盛顿，2000年12月20日

在第17章“Web浏览器模糊测试”中，我们讨论了Web浏览器的一些方面及其“可模糊测试性”。对Web浏览器模糊测试关注的不断增加也导致产生了一些模糊测试工具，甚至产生了更多的影响目前所使用的主流浏览器如Mozilla Firefox和微软的Internet Expolorer的漏洞。本章中，我们将讨论构建一个ActiveX模糊器的需求。尽管开发被限制于Internet Explorer并且一些ActiveX模糊器已经存在，但因为它是最令人感兴趣和最复杂的测试，所以选择该测试技术以进行进一步的讨论。限制在Internet Explorer并不是完全打消积极性，因为微软的浏览器仍然占据着Web用户市场的绝大部分份额。本章首先概要介绍了ActiveX技术的历史，然后直接深入到一个ActiveX模糊测试工具的开发中。

18.1 组件对象模型的背景知识

微软的COM是一个功能强大的软件技术，最初是在1990年早期被引入，其目标是为软件的互操作性提供一个通用协议。标准的客户机——服务器通信允许软件采用不同的语言来编写，而这些语言要能支持COM以在本地的同一个系统中相互交换数据，或者在不同的系统之间远程交换数据。COM目前被广泛地使用，并且它的发展历史非常重要，同时在发展过程中产生了一些缩写词（以及混淆）。

18.1.1 在Nutshell中的发展历史

COM最早的前身可以被追溯到动态数据交换（Dynamic Data Exchange,DDE），目前DDE技术仍然使用在Windows操作系统中。你可以通过Shell文件扩展、剪贴板查看器（NetDDE）以及微软的Hearts（也是NetDDE）来发现执行中的DDE。在1991年，微软发布了对象链接和嵌入（Object Linking and Embedding,OLE）技术。但是DDE只限于纯数据交换，而OLE可以在相互之间嵌入文档类型。OLE客户机——服务器互通信通过使用虚函数表（Virtual Function Table,VTBLs）而发生在系统库的内部。

在OLE之后引入了COM，然后又发布了OLE2，它是构建在COM而不是VTBLs之上的一个新版本的OLE。该技术在1996年被重命名为ActiveX。在1996年的后期，微软又发布了分布式COM（Distributed COM,DCOM）作为对COM的扩展以及对公共对象请求代理体系结构

(Common Object Request Broker Architecture,CORBA）的回应[⊖]。分布式计算环境/远程过程调用[⊜]（Distributed Computing Environment/Remote Procedure Call,DCE/RPC）是DCOM后面潜在的RPC机制。DCOM进一步扩展了COM的灵活性，因为它允许软件开发者通过像Internet这样的媒介来暴露功能，而不用提供对潜在代码的访问。

COM的历史上最近发生的事件是COM＋的引入，它是同Windows2000操作系统一起发布的。COM＋提供的额外好处是同Windows2000绑定在一起的微软事务服务器所管理的“构件工厂”。像DCOM一样，COM＋构件也可以是分布式的，并且不需要从内存中卸载该构件就可以实现重用。

18.1.2 对象和接口

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。于是软件就可以查询一个COM对象以确定它可能会暴露哪个接口和功能。每个COM对象被分配了一个唯一的128位标识符，该标识符被称为一个类ID（CLSID）。另外，每个COM接口也被分配了一个唯一的被称为接口ID（IID）的标识符。COM接口的例子包括IStream,IDispatch和IObjectSafety。这些接口以及其他的接口都是从一个被称为IUnknown的基接口派生而来的。

除了CLSID之外，对象能够可选地为验证指定一个程序ID（ProgID）。ProgID是一个人工可读的字符串，它非常便于引用一个对象。例如，考虑下面的别名：

- 000208D5-0000-0000-C000-000000000046
- Excel应用程序

CLSID和ProgID都被定义在注册表中，并且可以互相替换使用。然而，要注意ProID并不能被保证是唯一的。

18.1.3 ActiveX

ActiveX是目前可用的切合我们所讨论主题——Web浏览器模糊测试的唯一一种COM技术。类似于Java applets，ActiveX控件对操作系统的访问范围更广，并被用来开发许多对浏览器来说不可用的扩展。ActiveX控件的应用非常广泛，并且就像是直接分布在Web上一样同许多软件包绑定在一起。可能会依赖于ActiveX技术的产品和服务的例子包括在线病毒扫描器，Web会议和通话，即时消息传输器以及在线游戏等等。

微软的Internet Explorer是唯一从本质上支持ActiveX，并实现了标准的文档对象模型(Document Object Model, DOM）以处理实例化、参数和方法调用的浏览器。下面的例子为此问题提供了参考和说明：

```
Pure DOM
<object classid = "clsid:F08DF954-8592-11D1-B16A-00C0F0283628"
```

⊖ http://en.wikipedia.org/wiki/Corba

⊜ http://en.wikipedia.org/wiki/DCE/RPC

```
        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)
```

Outdated Embed

```
<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">
  function call_function()
  {
    obj = new ActiveXObject('AcroPDF.PDF');
    obj.property = "value";
    obj.method(arg, arg, arg);
  }

  call_function();
</script>
```

Visual Basic

```
<object classid = 'clsid:38EE5CEE-4B62-11D3-854F-00A0C9C898E7'
        id      = 'target' >
</object>

<script language='vbscript'>
  'Wscript.echo typename(target)
  'Sub SelectAndActivateButton ( ByVal lButton As Long )
  arg1=2147483647
  target.property = arg1
  target.method arg1
</script>
```

除了直接将控件加载到浏览器之外，一个ActiveX控件也可以直接作为标准COM对象来被加载并提供接口。直接加载一个控件对我们的模糊器而言是有利的，因为它省略了生成浏览器代码并将其加载到Internet Explorer的中间步骤。

微软COM的编程及其内部机理是一个很大的研究课题，有许多书都专门讲述该课题。为了

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书藉杂志全免 KEY 提供,，完全无加密。

了解关于COM的更多信息，可以访问微软的COM Web站点[㊀]以获得更高层次的一个概览，同时可以通过MSDN的文章《组件对象模型：技术概览》（The Component Object Model: A Technical Overview）[㊁]，来得到低层的信息。

在下一节中，我们将深入到一个ActiveX控件的开发中，同时也将进一步讨论有关COM技术的一些细节。

18.2 模糊器的开发

有许多语言适合于开发ActiveX模糊器。这是显而易见的，因为目前在不同ActiveX模糊器的实现中使用了大量的编程语言。COMRaider[㊂]大部分是用Visual Basic编写的，还有一小部分是用C++语言编写的。AxMan[㊃]是混合采用C++, JavaScript和HTML等多种语言来编写的。我们采用与这些模糊器所使用的相同的通用方法来对我们的模糊器进行建模：

- 枚举所有的可加载的ActiveX控件。
- 枚举针对每个ActiveX控件可访问的方法和属性。
- 解析类型库以确定方法参数的类型。
- 生成模糊测试用例。
- 执行模糊测试用例并监视异常情况。

与前面所开发的模糊器不同，我们完全采用一种单一的语言Python来实现我们的模糊器。你可能一开始会对所选择的语言感到吃惊，因为你会认为它完全不可行。然而，采用Python来开发实际上是可能的，因为现代的分布式应用包含暴露有潜在的Windows API的模块。我们所依赖的与COM进行接口的不同模块包括win32api, win32com, pythoncom和win32con。（如果你有兴趣进一步了解关于Windows环境下Python编程的一些详细信息，那么可以参阅由Mark Hammond编写的名为《Win32中的Python编程》（Python Programming on Win32）的文章[㊄]，Mark Hammond开发了一些模块，我们以这些模块作为桥梁将Python引入到了COM领域。）图18-1和图18-2是分别取自于PythonWin COM浏览器和PythonWin类型库浏览器的屏幕截图，显示了从高层对低层COM信息的访问。

作为一个程序例子，考虑下面的代码片段，它将启动微软Excel的一个实例，并通过设置布尔属性Visible将其变为可见的：

```
import win32com.client
xl = win32com.client.Dispatch("Excel.Application")
xl.Visible = 1
```

下面我们将通过研究如何枚举所有的可加载的ActiveX控件来深入分析更多的例子。本章

㊀ http://www.microsoft.com/com/default.mspx

㊁ http://msdn2.microsoft.com/en-us/library/ms809980.aspx

㊂ http://labs.idefense.com/software/fuzzing.php#more_comraider

㊃ http://metasploit.com/users/hdm/tools/axman/

㊄ http://www.oreilly.com/catalog/pythonwin32/

剩余部分所列出的代码片段引自于一个具有完整特性的COM模糊器，该模糊器可以从本书的官方Web站点（http://www.fuzzing.org）上下载。

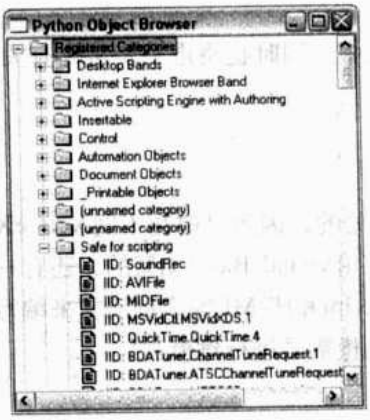


图18-1　PythonWin COM浏览器

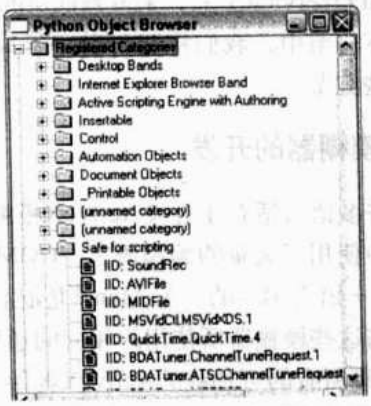


图18-2　PythonWin类型库浏览器

18.2.1　枚举可加载的ActiveX控件

我们的首要开发任务是枚举出在目标系统中所有可用的COM对象。COM对象的完整列表显示在Windows注册表[⊖]中HKEY_LOCAL_MACHINE（HKLM）键值和子键SOFTWARE\Classes的下面。我们可以使用标准的Windows注册表访问API[⊜]来访问该键值：

```
import win32api, win32con
import pythoncom, win32com.client
from win32com.axscript import axscript

try:
    classes_key = win32api.RegOpenKey( \
        win32con.HKEY_LOCAL_MACHINE,   \
        "SOFTWARE\\Classes")
except win32api.error:
    print "Problem opening key HKLM\\SOFTWARE\\Classes"
```

上述代码片段的前三行负责导入访问注册表以及与COM对象进行接口所需要的功能。一旦接口打开，键值就通过搜索有效的类CLSID来被枚举。如果发现了一个CLSID，那么条目就被保存到以后所使用的一个列表中。

```
skey_index = 0
clsid_list = []

while True:
    try:
```

⊖ http://en.wikipedia.org/wiki/Windows_registry

⊜ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/sysinfo/base/registry_functions.asp

```
            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:
       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
```

这些代码片段组合起来生成了描述系统中所有可用COM对象的一个元组列表。对COM对象的安全测试通常局限于那些可以访问Internet Explorer的控件。这是缘于这样一个事实，即到目前为止最主流的ActiveX开发技术就是一个攻击者可以控制一个恶意的Web站点，并强迫用户使用含有漏洞的构件来访问Web页。由于只有其中的某些控件可以在Internet Explorer中访问，所以我们的第二个任务就是将那些不能访问的构件从我们的列表中剔除出去。Internet Explorer在加载一个ActiveX控件时并不会提示是否满足下面三个准则中的任何一个[⊖]：

- 控件在Windows注册表中被标记为“脚本安全”。
- 控件在Windows注册表中被标记为“初始化安全”。
- 控件实现了IObjectSafety COM接口。

Windows注册表包含一个构件类别键，该键为已安装构件所实现的每个类别列出了子键。我们所寻找的两个子键是CATID_SafeForScripting和CATID_SafeForIntializing。下面的程序将确定一个给定的CLSID是否在注册表中被标记为在Internet Explorer中可访问。

```
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:
```

⊖ http://msdn.microsoft.com/workshop/components/artivex/safety.asp

```
            try:
                skey = win32api.RegEnumKey(key, skey_index)
            except:
                break

            # CATID_SafeForScripting
            if skey == "{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 skey == "{7DD95802-9882-11CF-9FA9-00AA006C42C4}":
                return True

            skey_index += 1

        return False
```

除了在注册表中进行标记之外，一个ActiveX控件还可以通过实现IObjectSafety接口来在Internet Explorer中将其自身标记为安全的。为了确定一个给定的ActiveX控件是否实现了IObjectSafety接口，首先必须要将其实例化然后进行查询。下面的程序将确定一个给定的控件是否实现了IObjectSafety接口，以及是否可以访问Internet Explorer：

```
    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
```

为了使可加载ActiveX控件的列表更加完整，我们必须要考虑最后一个方面。微软提供了这样一种机制，即通过注册表键HKLM\Software\Microsoft\Internet Explorer\ActiveX Compatibility\<CLSID of ActiveX Control>来销毁位[⊖]或者阻止在Internet Explorer中加载单个的CLSID。每个包含位于此注册表位置的一个条目的CLSID，都必须从列表中剔除出去。可以使用下面的函数来确定一个给定的CLSID是否以这种方式被禁用：

```
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, typ) = win32api.RegQueryValueEx(key, \
        "Compatibility Flags")
except win32api.error:
    return False

if typ != win32con.REG_DWORD:
    return False

if compat_flags & 0x400:
    return True
else:
    return False

return False
```

对我们所创建的ActiveX控件列表进行两遍检查，以确定哪个控件可能是不恰当的，接下来我们必须要研究控件所暴露的属性和方法。

18.2.2 属性、方法、参数和类型

具备系统的生成目标控件的一个列表的能力是非常方便的，这也正是我们的ActiveX模糊测试发挥优势之处。对一个COM对象所暴露的所有属性和方法的描述直接嵌入到其中。另外，属性和方法参数的类型也被加以了描述。这个COM特性对作为模糊器开发者的我们来说是很令人兴奋的，就像是第一次听说该特性的软件开发者那样兴奋。可以通过编程来枚举出一个ActiveX控件的攻击界面的这种能力允许我们创建更加智能化的模糊器，它可以知道一个特定的方法可能会期望何种类型的数据。这样，当期望一个整数时，我们就不会浪费时间来提供一个字符串了。

⊖ http://support.microsoft.com/kb/240797

在COM领域，数据是通过一个被称为VARIANT的结构来传递的。VARIANT数据结构支持许多数据类型，包括整数，浮点数，字符串，日期型，布尔型，其他的COM对象以及这些类型的任意数组。PythonCOM提供了一个抽象层以为我们隐藏许多的细节信息。表18-1显示了一些内部Python类型以及它们的等价VARIANT类型之间的映射。

表18-1　PythonCOMVARIANT转换

Python对象类型	VARIANT类型
整数	VT_I4
字符串	VT_BSTR
浮点数	VT_R8
空	VT_NLL
真/假	VT_BOOL

pythoncom模块提供了LoadTypeLib()函数以解析直接来自于一个二进制COM对象的类型库。我们所需要了解的关于一个COM对象属性和方法的所有信息都可以通过被加载的类型库来获得。作为一个例子，让我们来研究一下图18-2中所示的针对Adobe Acrobat PDF控件的类型库。ActiveX控件同Adobe的Acrobat Reader绑定在一起并且可以访问Internet Explorer，因为它被同时标记为脚本安全以及初始化安全。下面的代码片段说明了如何针对该代码来加载类型库，同时它还应用了一些Python编程技巧创建了一个VARIANT名字的映射：

```
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
```

所生成的VARIANT名字映射只是用于说明的目的，我们在后面将会看到这一点。一个类型库可以定义多种类型，并且循环的次数可以通过调用GetTypeInfoCount()来重新得到。在我们的例子中，有三种类型被显示在图18-2的第一列中。下面的例子说明了如何循环不同的类型并输出它们的名字：

```
for pos in xrange(tlb.GetTypeInfoCount()):
    name = tlb.GetDocumentation(pos)[0]
    print name
```

在Acrobat控件中定义了三种类型。下面对图18-2中高亮显示的IAcroAXDocShim类型进行仔细研究。同大多数编程中的计算方法一样，计数的位置是从0开始的，这意味着我们所期望类型的索引是2而不是3。在下面的代码块中，类型信息和属性从前面定义的类型库中被抽取出来，并被用于枚举此特定类型所包含的属性：

```
info = tlb.GetTypeInfo(2)
attr = info.GetTypeAttr()

print "properties:"
for i in xrange(attr.cVars):
    id    = info.GetVarDesc(i)[0]
    names = info.GetNames(id)
    print "\t", names[0]
```

cVars属性变量指定了在该类型下所定义的属性（或变量）的数量。该数量用于执行循环并输出每个属性的名字。将方法、参数和参数类型进行枚举非常简单，如下面的代码片段所示：

```
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属性变量指定了在该类型下所定义的方法的数量。这些方法被枚举出来，但忽略了那些被设置了wFuncFlags标志的方法。该标志指明了这个方法是受限制的（不可访问的），因此不适合于进行模糊测试。GetNames()函数返回方法的名字，以及方法中每个参数的名字。方法的名字被输出，然后列表的剩余部分names[1:]被传递以访问参数。该函数的描述最初是通过调用GetFuncDesc()来返回的，并且包含针对每个参数的VARIANT类型的一个列表。VARIANT类型被表示为一个整数，该整数通过索引前面生成的VARIANT映射被转换为一个名字。

该脚本在Adobe Acrobat PDF ActiveX控件的IAcroAXDocShim接口上的最终执行结果如下所示：

```
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()
    setCurrentHightlight()
        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()
```

根据所输出的结果，我们可以看到在该类型中定义了39个方法（函数），但是并没有定义任何属性。每个方法的参数和类型列表被成功地枚举并显示。这个信息在智能模糊器的开发过程中是非常重要的，它应当被用于测试用例的生成以确保每个变量被正确地模糊化。例如，考虑对一个短整数（VT_I2）和一个长整数（VT_I4）进行模糊测试的比较。不将这些类型看作是普通的整数，那么就可以节省宝贵的时间，方法是通过在从0到0xFFFF（65 535）的有效区间内对短整数进行模糊测试，而不是在从扩展的0到0xFFFFFFFF（4 294 967 295）的全长整数（长整数）的有效区间内对其进行模糊测试。

到目前为止，我们已经讨论了为一个给定系统中的、可以访问Internet Explorer的每个ActiveX控件枚举其每个属性、方法和参数所必需的步骤。下一步就是要选择合适的模糊测试启发式准则并开始测试。

18.2.3 模糊测试和监视

第6章“自动化的测试数据生成”讨论了选择智能字符串和整数模糊值的重要性。所选择的模糊测试启发式准则被设计为使错误发生的可能性最大化。本书中所描述的绝大部分模糊器和测试用例实际上都是被设计来发现低层错误的，如缓冲区溢出，而不是发现诸如访问受限资源那样的逻辑错误。目录遍历修饰符就是启发式准则的一个例子，它被用来发现类似的安全漏洞。当对ActiveX控件进行模糊测试时，对行为违规的搜索要特别谨慎，因为研究者经常发现控件不应当通过Internet Explorer来被访问。

例如，考虑WinZip FileView ActiceX控件的不安全方法暴露漏洞[⊖]。在这个例子中，ActiveX控件及其ProgID WZFILEVIEW.FileViewCtrl.61作为脚本安全而发布，允许一个恶意Web站点将控件加载到浏览器并利用其暴露的功能。更加特别的是，方法`ExeCmdForAllSelected`和`ExeCmdForFolder`允许调用者从任意位置拷贝、移动、删除并执行文件，这些位置包括网络共享库、FTP目录以及Web目录。这就为攻击者提供了一种简便的手段，即通过控制一个恶意的Web站点来下载并执行一个任意的可执行文件，而不发出任何警告信息。WinZip针对此问题提出了一种处理方法，即删除脚本安全设置并为特定的控件设置一个销毁位。

除了前面所使用的典型的模糊测试启发式准则之外，有效文件路径、命令以及URL应当被包含在模糊数据中以帮助发现类似于WinZip FileView漏洞的错误。当然，模糊监视器必须要包含所需要的功能以确定所提供的任何资源是否被成功访问。我们将在后面对这个概念进行深入讨论。

一旦生成了模糊数据的一个合适的列表，下一步就是创建一系列的测试用例。完成此任务的一种方法是使用前面所提到的任意一种方法来创建嵌入在目标ActiveX控件中的一个HTML页，然后使用模糊化的参数来调用一个目标方法。可以为每个测试生成一个单独的文件，于是单独的测试用例就可以被加载到Internet Explorer中。另外，目标控件也可以被直接加载和测试。根据我们前面提到的例子，下面的Python代码片段将实例化一个Adobe Acrobat PDF ActiveX控

⊖ http://www.zerodayinitiative.com/advisories/ZDI-06-040.html

件，并访问它所暴露的方法中的两个：

```
adobe = win32com.client.Dispatch("AcroPDF.PDF.1")
print adobe.GetVersions()
adobe.LoadFile("c:\\test.pdf")
```

该代码片段的第一行负责实际创建Adobe COM对象的一个实例，该实例可以通过名为adobe的变量来访问。一旦该实例被创建，该对象就可以很自然地通过接口来访问。第二行输出了调用GetVersions()函数的结果，第三行使该控件将一个PDF文件加载到Adobe Reader查看器。将这个例子进行扩展以对其他的方法、参数和控件进行模糊测试就是非常简单的事情了。

该问题的剩余环节就是模糊器中的监视器部分。除了Python COM模块之外，我们利用PaiMei[⊖]逆向工程库来实现一个基于调试器的模糊测试监视器。该库将在后续章节中被扩展使用，在那里我们将讨论一些更深层次的内容。从本质上来说，轻量级调试器的创建就是为了封装目标ActiveX控件的执行。当一个低层次漏洞如缓冲区溢出被发现时，调试器被应用，并且错误的特定细节信息被记录下来。PaiMei库也提供了钩住API调用的功能。该特性被用来监视异常行为问题。通过对微软库调用函数如CreateFile()[⊜]和CreateProcess()[⊝]的参数进行观察，我们就可以确定目标ActiveX控件是否成功访问了一个它不应当访问的资源。对API钩子功能的细节进行评审，作为一个练习留给读者来完成。

最后需要注意的一点是，使用简单的正则表达式搜索来检查所暴露属性和方法的名字是值得进行的一项工作。如果遇到了名为GetURL(),DownLoadFile()或Execute()等的方法，那么你可能想要立即给予它们某些特别的关注。

18.3 小结

在本章中，我们讨论了微软的COM技术的发展历史，并且指出了从Internet Explorer Web浏览器对一个ActiceX控件进行访问的需求。研究了所需要的Python COM接口，同时研究了枚举可访问的ActiveX控件及其属性、方法、参数和参数类型所需要的特定步骤。本章中所研究的代码片段引自于一个具有完整特性的COM模糊器，该模糊器可以从本书的官方Web站点（http://www.fuzzing.org）上下载。



⊖ https://www.openrce.org/downloads/details/208/PaiMei

⊜ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/fs/createfile.asp

⊝ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createprocess.asp

第19章 内存数据的模糊测试

"它是白的。"

——George W. Bush，在英国被一个小孩问白宫像什么之后的回答，2001年7月19日

在本章中，我们引入内存数据模糊测试的概念，它是一种以前较少被人们所关注的新的模糊测试方法，目前还没有针对该方法的具有完整特性的概念验证工具被公开发布。从本质上来说，该技术的目标是将模糊测试从所熟悉的客户机——服务器（或客户机——目标机）模型，转换到只面向内存目标的模型中。尽管从技术上来看，它是一种更加复杂的方法，要求熟练掌握汇编语言、进程内存规划以及进程工具的低层知识，但是在我们进入任何特定的实现细节之前，我们会指出使用该技术同样会带来许多的益处。

在上一章中，我们采用一种协商的方法从UNIX和Windows的视角都提供了一个无偏见的观点。由于该模糊测试技术所具有的复杂性，本章中我们将关注的焦点限制在一个单一的平台上。我们选择微软的Windows作为关注的平台，其原因如下：首先，这种模糊测试技术在不开源的目标应用中应用的更好，而二进制软件在Windows平台上的分布要比其在UNIX平台上的分布更加广泛。其次，Windows为进程工具提供了丰富而功能强大的调试API。这并不是说调试API在UNIX平台上是不可用的，但我们认为UNIX平台上的调试API的健壮性不强。当然，可以采用这里所讨论的通用概念和方法以应用到不同的平台中。

19.1 内存数据模糊测试的概念及实施该测试的原因

到目前为止我们所讨论的每一种模糊测试方法都要求通过所期望的通道来生成和传输数据。在第11章"文件格式模糊测试"，第12章"文件格式模糊测试：UNIX平台上的自动化测试"和第13章"文件格式测试：Windows平台上的自动化测试"中，被变异的文件直接被应用程序所加载。在第14章"网络协议模糊测试"，第15章"网络协议模糊测试：UNIX平台上的自动化测试"和第16章"网络协议模糊测试：Windows平台上的自动化测试"中，模糊数据通过网络socket来传递到目标服务。这两种情形都要求我们向目标提供整个文件或一个完整的协议，即使我们只对一个特定域的模糊测试感兴趣。同时在这两种情况下，我们完全没有察觉到也完全不关心是哪些底层代码被执行以响应我们的输入。

内存数据模糊测试是一种完全不同的测试方法。它不是关注于一个特定的协议或者文件格式，它所关注的是实际的底层代码。我们将测试的关注焦点从数据输入转换到负责解析数据输入的函数以及单个的汇编指令。采用这种方式，我们忽略了所期望的数据通信通道，而是关注于在目标应用程序的内存中变异或生成数据。

这种方法在什么时候可以给我们带来益处呢？考虑这样一个情形，即你正在对一个实现了

复杂包加密或模糊模式的不开源网络daemon程序进行模糊测试。为了成功地对该目标应用进行模糊测试，首先需要实现一个完整的逆向工程，并重新生成模糊模式。这是内存数据模糊测试的一个主要应用场景。不是关注于不必要的协议封装，我们将分析目标应用的内存空间以寻找感兴趣的时间点，例如，当所接受的网络数据被模糊化之后。作为另外一个例子，考虑下面的情形，即我们期望对一个特定函数的健壮性进行测试。假定通过进行某些逆向工程工作，我们阻止了含有单一字符串参数（即邮件地址）的一个邮件解析例程，并以某种方式将其进行解析。我们能够生成并传输包含变异后的邮件地址的网络包或测试文件，但是在内存数据模糊测试中，我们可以通过直接对目标例程进行模糊测试来关注所要做的工作。

这些概念将在本章的后续部分以及下一章中给予进一步的详细阐述。然而，我们需要首先来学习一些相关的背景知识。

19.2　必需的背景知识

在我们开始进一步讨论之前，让我们先来快速浏览一下微软Windows的内存模型，并简要讨论一下一个典型Windows进程的内存空间的通用布局和属性。在这一节中，我们将涉及到一些复杂的细节内容，我们鼓励读者对这些问题进行深入钻研。由于其中某些内容可能会非常枯燥，你可以选择跳跃阅读概要内容，并直接前进到本节结尾的图进行分析。

从Windows95开始，Windows操作系统就开始构建在平坦内存模型之上，该模型在32位的平台上提供了总共4GB的可寻址内存。4GB的地址范围默认被划分为两部分。底层的一半（0x00000000-ox7FFFFFFF）被保留用做用户空间，顶层的一半（0x80000000-0xFFFFFFFF）被保留用做内核空间。这种划分也可以被改为3：1（通过/3GB boot.ini设置[⊖]），其中3GB用做用户空间，1GB用做内核空间，用于像Oracle数据库这样的内存密集型应用程序改善性能。平坦内存模型与分段内存模型不同，它通常是使用一个单一的内存段来引用程序代码和数据。在另一方面，分段内存模型利用多个段来引用不同的内存位置。平坦内存模型与分段内存模型相比所具有的最大优点是获得了极高的性能并降低了程序员编程复杂性，因为在这种情况下不需要选择和转换段。为了进一步简化程序员的编程工作，Windows操作系统通过虚拟寻址来管理内存。从本质上来说，虚拟内存模型为每个运行中的进程提供了它自己的4GB虚拟地址空间。这项工作是通过一个从虚拟地址到物理地址的转换来完成的，并且是在一个内存管理单元（Memory Management Unit,MMU）的帮助下完成的。显然，并不是有很多的系统可以在物理上支持为每个运行中的进程提供4GB的内存空间。相反，虚拟内存空间是建立在内存分页的概念之上的。一个页是一个连续的内存块，在Windows操作系统中页的典型大小是4 096（0x1000）字节。当前正在使用的虚拟内存页是存储在RAM中的（主存储器）。没有使用的内存页可以被交换到磁盘（辅助存储器），并且当以后再次需要时重新存储到RAM中。

Windows内存管理的另外一个重要概念是内存保护。内存保护属性应用在最细粒度的内存页上。将内存保护属性只分配到页的一部分是不可能的。我们所关心的一些可用的内存保护属

⊖ http://support.microsoft.com/kb/q291988/

性如下所示[⊖]：

- 页可执行（PAGE_EXECUTE）。内存页是可执行的，试图从页中读取或向页中写入将导致一个访问违规。
- 页可执行可读（PAGE_EXECUTE_READ）。内存页是可执行的并且是可读的，试图向该页中写入将导致一个访问违规。
- 页可执行可读可写（PAGE_EXECUTE_READWRITE）。内存页是完全可以被访问的。它是可执行的，也可以被读取并且写入。
- 页不可访问（PAGE_NOACCESS）。内存页是不可访问的。任何试图执行该页，对该页进行读取或向该页中写入的操作都将导致一个访问违规。
- 页只读（PAGE_READONLY）。内存页只能被读取。任何试图向该页中写入的操作都将导致一个访问违规。如果底层的体系结构支持读取与执行之间的差异（实际并不总是这样），那么任何试图从该内存页进行执行的操作都将导致一个访问违规。
- 页可读可写（PAGE_READWRITE）。内存页是可读的并且是可写的。同PAGE_READONLY属性一样，如果底层的体系结构支持读取与执行之间的差异，那么任何试图从该内存页进行执行的操作都将导致一个访问违规。

出于我们的目的，这里只重点关注于PAGE_GUARD修饰符，根据MSDN[⊜]的描述，任何试图对被保护的内存页进行访问以及随后删除保护条件的操作都将导致一个STATUS_GUARD_PAGE_VIOLATION违规。实际上，PAGE_GUARD充当着一次访问报警的角色。我们可能会使用该特性来监视对内存特定区域的访问。除此之外，我们也可以使用PAGE_NOACCESS属性。

熟悉内存页，内存空间布局以及保护属性对于内存数据模糊测试而言非常重要，其重要性我们将在下一章中可以看到。内存管理是一个非常复杂的问题，在其他许多出版物当中已经对该问题进行了大量的讨论[⊝]。我们鼓励读者去参考其他人所做的工作。然而，出于本书的考虑，你需要知道的所有内容如下所示：

- 每个Windows进程“可以看到”它自己的4GB虚拟地址空间。
- 只有从0x00000000到0x7FFFFFFF的底层2GB空间可以用于用户空间，上层的2GB空间被保留用做内核空间。
- 一个进程的虚拟地址空间被隐式地保护，以不被其他进程所访问。
- 4GB的地址空间被划分为单独的页，该页的典型大小是4 096字节（0x1000）。
- 在最细的粒度上，内存保护属性被应用于单个的内存页。

⊖ http://msdn2.microsoft.com/en-us/library/aa366786.aspx

⊜ http://msdn2.microsoft.com/en-us/library/aa366786.aspx

⊝ 《微软Windows内幕（第4版）》（Microsoft Windows Internals, Fourth Edition），Mark E.Russinovich,David A.Solomon；《未公开的Windows2000内幕：程序员指南》（Undocumented Windows 2000 Secrets: A Programmer's Cookbook），Sven Schreiber；《未公开的WindowsNT》（Undocumented Windows NT），Prasad Dabak,Sandeep Phadke,Milind Borate。

• PAGE_GUARD内存保护修饰符可以被用做一次页访问报警器。

仔细研究一下图19-1，以便更好地理解在典型的Windows进程的虚拟地址空间中特定元素所处的位置。同样，当我们在下一章讨论一个内存数据自动化模糊测试工具的特定开发细节时，图19-1也会作为一个有用的参考。在该图之后是对所列出的不同元素的一个简要描述，如果你对它们不熟悉的话可以参阅。

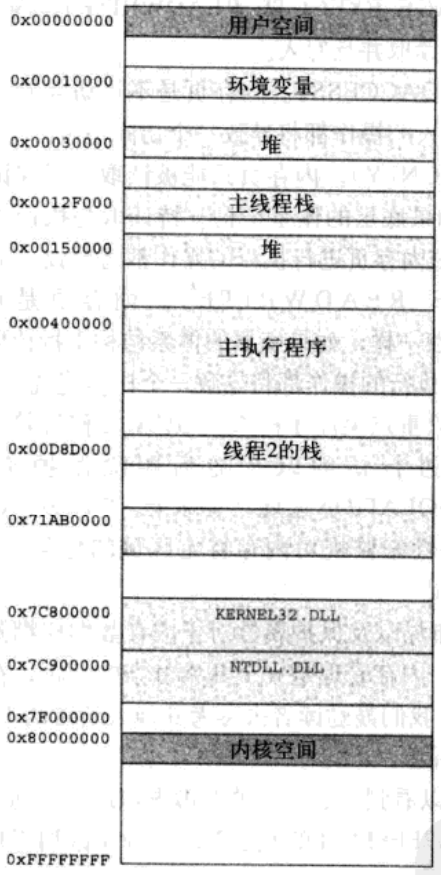


图19-1　典型的Windows内存空间布局（没有根据大小刻度来绘制）

为了帮助那些对此不太熟悉的人更好地理解有关内容，下面对图19-1中所显示的不同元素进行一个简要的描述。从最低的地址区域开始，我们在地址0x00010000处发现了进程环境变量。回忆一下第7章“环境变量和参数的模糊测试”，可以知道环境变量是系统范围的全局变量，被用来定义应用程序的特定行为。在更高一些的地址区域中，我们发现了用于堆的两个内存区域，分别在地址0x00030000和0x00150000。堆是一个内存池，在这里可以通过调用诸如`malloc()`和`HeapAlloc()`函数来进行动态内存分配。注意对任何给定的进程，这里都可以存在多于一个的堆，正如图19-1中所示有两个堆。在这两个堆区域之间，我们在地址0x0012F000处发现了主线程的栈。该栈是一个后进先出的数据结构，用来跟踪函数调用链和局部变量。注意在一个进

程中的每个线程都有其自己的栈。在图19-1的示例中，线程2的栈位于地址0x00D8D000。在地址0x00400000，我们发现了主执行程序的内存区域，我们所执行的.exe文件将进程加载到了内存中。在用户空间中靠近顶层的区域，我们发现了一些系统DLL，如kernel32.dll和ntdll.dll。DLL是微软针对共享库的实现，它所包含的公共代码被跨应用程序使用，并且通过一个单一的文件被导出。可执行文件和DLL都利用了可移植的可执行（Portable Executable, PE）文件格式。注意图19-1所示的地址对于不同的进程将会是不同的，图中的地址只是作为一个例子来进行说明。

我们再一次涉及到了一些复杂的主题，而这些主题都应当在专门的书中来讲述。我们鼓励读者去参考一些讨论更加深入的材料。然而，对于我们要实现的目标而言，我们应当已经掌握了足够的信息以深入研究目前的主题。

19.3 究竟什么是内存数据模糊测试

我们在本书中提出并实现了两种内存数据模糊测试的方法。在这两种方法中，我们都将模糊器从目标之外移到了目标自身内部。这个过程可以通过可视化的方式来得到最好的解释和理解。作为开始，考虑一下图19-2，它描述了针对一个假象但很典型的网络目标的一个简单控制流图。

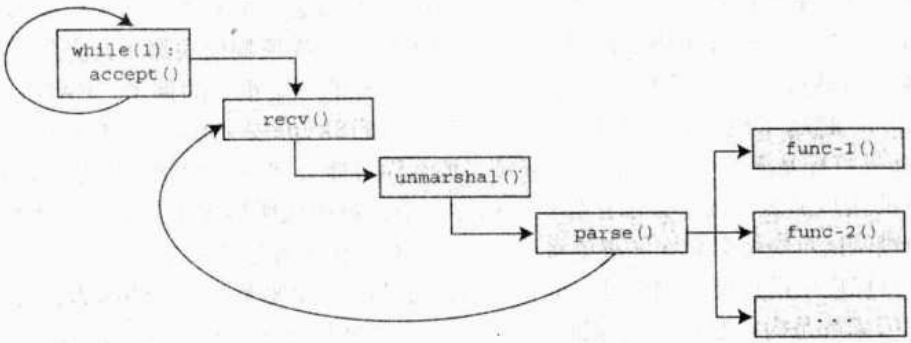


图19-2 控制流图示例

在示例目标应用程序主线程中的循环正在等待新的客户端连接。当接收到一个连接时，就产生一个新线程以处理客户端的请求，该请求通过一个或多个对recv()的调用而被接收。然后所收集的数据就通过某种形式的反汇编或处理例程被传输。Unmarshal()例程可能负责协议的解压或解密，但实际上并不解析数据流内部的单个字段。处理后的数据接着传递到主解析例程parse()，该例程是建立在其他一些例程和库调用基础之上的。parse()例程处理数据流内部不同的单个字段，并且在最终往回执行循环以从客户端接收更多的指令之前进行适当的请求行为。

在这种情况下，如果不通过网络来传输模糊数据并且我们可以忽略网络通信的整个步骤，那么情况将会怎么样呢？这样的代码分离允许我们直接使用最想要发现错误的例程来进行工作，比如像解析例程那样的例程。于是，我们就可以将错误注入到例程中并监视这些变化的结

果，这样可以极大提高模糊测试过程的效率，因为所有的处理活动都在内存中进行。通过使用某种形式的进程工具，我们想要在解析例程之前“钩住”我们的目标，修改例程所接受的不同输入，并且使例程允许它们的过程。由于本书的主题是模糊测试，我们当然希望能够循环执行这些步骤，在每次循环中自动修改输入并且监视输出结果。

19.4 目标

对于构建一个网络协议或文件格式模糊器而言，最耗费时间的需求就是对协议和文件格式本身的解析。已经进行了一些工作以试图进行自动化的协议解析，这些工作借鉴了生物信息学领域的一些思想，该主题我们将在第22章“自动化的协议解析”中进行介绍。然而，在大多数情形下，协议解析仍然是一个手工并且枯燥单调的过程。

当对标准的、公开的协议如SMTP、POP和HTTP等进行处理时，你得到的好处是可以有大量的参考来源，例如RFC、开源的客户端甚至是以前所编写的模糊器。另外，考虑到这些协议被不同的应用程序所广泛使用，那么在你的模糊器中实现该协议可能是值得的，因为它可以被大量重用。然而，你会经常发现自己处于这样一种情形，即目标协议是未公开的、复杂的或者是私有的。在这种情况下，协议的解析可能需要相当的时间和资源，那么内存数据模糊测试可能会成为一个可行的节省时间的选择。

如果目标通信是在这样一个协议之上，即该协议被封装进一个开放的加密标准那么将会怎么样呢？情况更糟一些，如果协议被封装进一个私有的加密或模糊模式那么又会怎样呢？曾经一度非常流行的Skype[⊖]通信套件就是要审核的复杂目标的一个很好的例子。EADS/CRC安全团队竭尽全力[⊜]超越了模糊的层次并发现了一个严重的Skype安全缺陷（SKYPE_SB/2005-003）[⊝]。如果目标暴露出一个非加密的接口，那么你可能会回避该问题，但是如果目标没有暴露出这样的接口那么情况又会怎样呢？加密协议是模糊测试所显现出来的一个主要的障碍。内存数据模糊测试可能也会允许你避免将加密例程进行逆向工程这样艰难的过程。

当你对目标应用进行模糊测试时，准备应用该方法时必须要考虑许多的因素，首先并且最重要的考虑因素就是内存数据模糊测试要求将目标应用进行逆向工程，以定位最佳的“钩住”点，这样就可以减少进入的开销。通常情况下，内存数据模糊测试应用的最佳目标是运行在被你的内存数据模糊测试工具所支持的平台上的不开源的目标应用。

19.5 方法：变异循环插入

在内存数据模糊测试中我们所提到的第一个方法就是变异循环插入（MLI）。MLI要求我们首先通过手工的逆向工程来定位`parse()`例程的起始点和结束点。一旦将其定位，我们的MLI客户端就可以将一个`mutate()`例程插入到目标应用的内存空间中。变异例程负责传递被解析例

⊖ http://www.skype.com

⊜ http://www.ossir.org/windows/supports/2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf

⊝ http://www.skype.com/security/skype-sb-2005-03.html

程所处理的数据，该例程可以通过许多方法来实现，下一章中我们将对这些方法进行特别的介绍。接下来，我们的MLI客户端将插入从解析例程末尾到变异例程起始点的一个无条件跳转，以及从变异例程末尾到解析例程起始点的一个无条件跳转。在这些操作执行完毕后，针对我们目标的控制流图如图19-3所示。

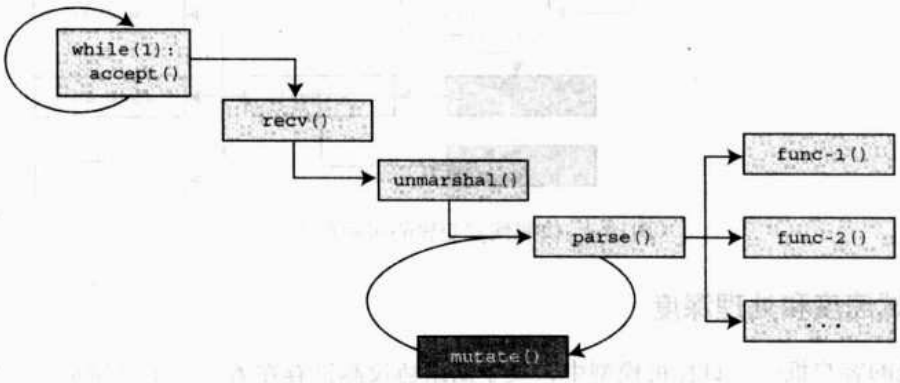


图19-3 变异循环插入的可视化表示

那么现在发生了什么情况呢？我们已经在目标应用的解析代码周围创建了一个自满足的数据变异循环。这样就不再需要远程连接到我们的目标应用并发送数据包，自然也就节省了大量的时间。每次循环迭代将传递不同的、潜在导致错误的数据到mutate()所规定的解析例程中。

到目前为止，并没有说我们所设计的例子过于简单。尽管这样做的目的是作为一个可以理解的入门性介绍，但注意到内存数据的模糊测试是一种实践性很强的方法非常重要。我们将在第20章“内存数据的模糊测试：自动化”构建一个功能化模糊器时对此进行更加详细的介绍。

19.6 方法：快照恢复变异

在内存数据模糊测试中我们所提到的第二种方法就是快照恢复变异（SRM）。像MLI一样，我们将要略过目标应用中以网络为中心的部分，而只攻击关注的焦点，在这个例子中就是parse()例程。同样像MLI一样，SRM也要求定位解析代码的起始点和结束点。当这些位置被标记后，当到达解析代码的起始点时，我们的SRM客户端将利用目标进程的一个快照。一旦解析过程完成，我们的SRM客户端将会把进程快照进行恢复，变异初始数据，并且再次执行解析代码。修改之后的目标应用的控制流图如图19-4所示。

我们再一次在目标应用的解析代码周围创建了一个自满足的数据变异循环。在第20章中，我们实现了一个概念上的功能验证工具，在该工具中对所描述的这些方法进行了实际的应用。我们所构建的原型性质的模糊器将有助于我们依靠实验来确定每一种方法的优缺点。然而，还是首先让我们来更加详细地讨论一下内存数据模糊测试所具有的主要优点之一。

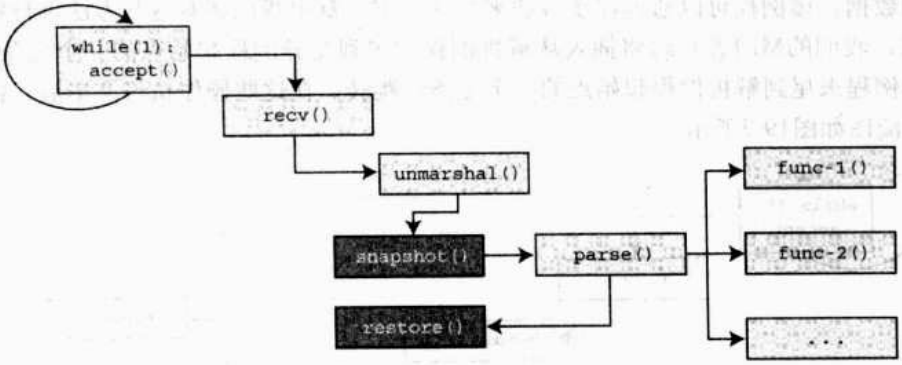


图19-4　快照恢复变异的可视化表示

19.7　测试速度和处理深度

在传统的客户机——目标机模型中，关于网络协议测试存在着一个不可避免的延迟。测试用例必须要单个通过网络来进行传递，并且依赖于模糊器的复杂程度，来自于目标的响应必须要被重复读取和处理。当对更深层次的协议状态进行模糊测试时，该延迟将会进一步增加。例如，考虑通常经过TCP端口110来提高服务的POP邮件协议。一个有效的检索邮件的交换（用粗体显示了用户输入）可能会如下所示：

```
$ nc mail.example.com 110
+OK Hello there.
user pedram@openrce.org
+OK Password required.
pass xxxxxxxxxxx
+OK logged in.
list
+OK
1 1673
2 19194
3 10187
... [output truncated]...
.
retr 1
+OK 1673 octets follow.
Return-Path: <ralph@openrce.org>
Delivered-To: pedram@openrce.org
… [output truncated]…
retr AAAAAAAAAAAAAAAAAAAAAA
-ERR Invalid message number. Exiting.
```

由于我们的网络模糊器的测试目标是RETR动词的参数，因此必须要通过为每个测试用例来重新连接并重新验证邮件服务器来遍历所需要的进程状态。相反，我们可以通过使用前面所讨论的某个内存数据模糊测试方法在所需要的进程深度进行测试，从而使我们能够专注于

RETR参数的解析。进程深度和进程状态在第4章“数据表示和分析”中进行过讨论。

19.8 错误检测

内存数据模糊测试的优点之一是对模糊器所引发错误的检测。由于我们已经在这样一个低层次上对目标应用实施了测试，因此可以很容易地扩展到其余部分。使用MLI和SRM方法，如果我们的模糊器被作为一个调试器实现（它将会是一个调试器），那么它就可以在每次变异迭代的过程中监视异常条件。错误发生的位置可以同例程的上下文信息一起被保存，这样，一个研究者就可以在以后对它进行检查从而确定错误的精确位置。你应当注意到目标利用了反调试方法，如前面提到的Sype的目标将会限制这个所选择的实现方法。

错误检测的最大困难是过滤掉那些误报情况。类似的问题被认为是海森堡的不确定性原理[⊖]，在内存数据模糊测试的条件下彻底观察我们的目标进程，就会导致其行为发生变化。由于我们采用了被确定为不是最初想要的方式对目标的进程状态进行了直接的修改，因此存在许多机会会产生不相关的错误。考虑图19-5。

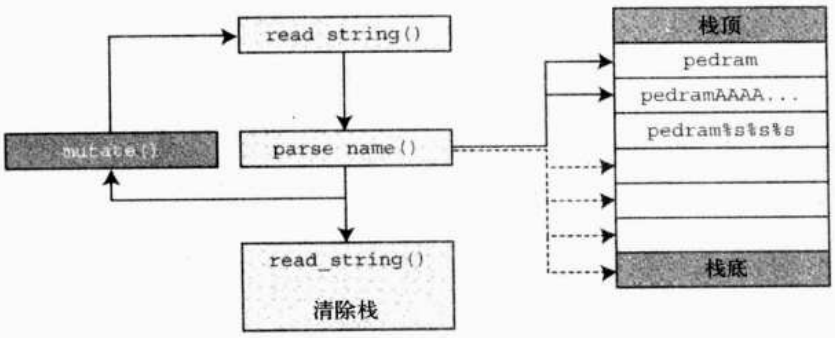


图19-5 在变异循环插入过程中的栈空间耗尽

我们应用MLI来封装了parse_name()和read_string()。在每次循环迭代过程中，我们将parse_name()的输入进行变异以试图引发一个错误。然而，由于我们是在栈被分配之后以及栈修改之前执行的钩住函数，因此在每次循环迭代中我们将连续地丢失栈空间。最后，我们将耗尽可用的栈空间从而导致一个不可再现的错误，该错误完全是由我们的模糊测试过程所引发的，而不是目标应用中已经存在的一个错误。

19.9 小结

内存数据模糊测试不是拯救困境的一种努力。在任何测试可以发生之前，一个熟练的研究者必须要将目标进行逆向工程以查明插入和使用工具的位置。同时还要求研究者确定通过内存数据模糊测试所发现的问题在正常的条件下是否可以真正访问用户输入。在大多数情况下，将不存在“立刻的”发现，但应当考虑一个潜在的输入。这些缺陷被该技术所提供的许多优点所

⊖ http://en.wikipedia.org/wiki/Heisenberg_principle

覆盖。

根据我们所掌握的情况，内存数据模糊测试的概念是由HBGary LLC的Greg Hoglund在2003年的BlackHat Federal[⊖]和Blackhat USA[⊜]安全会议上第一次提出来的。HBGary提供了一个称为Inspector[⊜]的商业内存数据模糊测试解决方案。在该论文介绍中，当提到进程内存快照时，Hoglund隐含地表达了他的一些专利技术。内存快照是我们在一个自动化工具的创建过程中将要详细讨论的技术之一。

在下一章中，我们也将讨论一个定制的内存数据模糊测试框架的实现及其使用。根据我们所掌握的情况，这是处理该任务的第一个开源的概念验证工具。关于该框架的所有改进信息将在本书的Web站点http://www.fuzzing.org上给予发布。

⊖ http://www.blackhat.com/presentations/bh-federal-03/bh-fed-03-hoglund.pdf

⊜ http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-hoglund.pdf

⊜ http://hbgary.com/technology.shtml

第20章 内存数据的模糊测试：自动化

"我听说在Internet上有一些谣言，说我们正在进行一个设计。"

——George W. Bush，第二届总统辩论，St Louis，MO，2004年10月8日

在上一章中，我们引入了内存数据模糊测试的概念。尽管我们在以前试图对Windows平台和UNIX平台上的内存数据模糊测试保持一个平衡，但是由于目前所讨论的主题的复杂性，我们决定将只关注于这个模糊测试方法在Winodws平台上的应用。更加特别的，我们将主要讨论该方法在32位x86 Windows平台上的应用。本章中，我们将详细地一步一步介绍创建一个内存数据模糊测试概念验证工具的步骤。我们将枚举出所希望具有的特性集，讨论所采用的开发方法，以及开发语言的选择。最后，在对一个实例研究进行了分析之后，我们总结了该模糊测试方法所带来的益处以及进一步改进的地方。我们在本章中所讨论的所有代码都是开源的，并且可以从http://www.fuzzing.org网站上公开下载。我们希望读者对相关特性进行改进、扩充并提供错误报告。

作为开始讨论之前的一个提示，请注意尽管我们已经涉及并解释了x86体系结构和Windows平台不同的低层次的方面，但是我们并没有非常深入地对它们进行剖析。因为这些内容已经超出了本书的研究范围。但是一个积极的方面是，我们在本章中所讨论的不同概念以及所开发的代码将在后续章节中发挥作用，如第24章"智能故障检测"。

20.1 所需要的特性集

回忆一下我们在第19章"内存数据的模糊测试"中所引入的针对内存数据模糊测试的两种方法，即MLI和SRM，我们希望将这两种方法都包含到概念验证的实现中。对于MLI而言，我们需要具备修改一条指令的能力，以创建循环，在目标应用中循环的起始位置放入钩子，并且在该处变异目标应用的缓冲区。对于SRM而言，我们需要具备在两个特定的点将目标进程钩住的能力，在一个钩子点复制目标应用的内存空间，并且在其他的钩子点恢复目标应用的内存空间。为了提供进一步的参考，图20-1和图20-2可视化地描述了我们的需求，图中的粗体文本指出了我们需要对目标进程所做的修改，灰色文本框指出了我们的内存数据模糊器必须要增加的功能。

为了实现所列出的*instrumentation*进程的需求，我们准备编写一个定制的Windows调试器。将内存数据模糊器作为一个调试器来创建，将能够很好地满足我们所有的需求。我们很宽泛地定义了术语*instrument*以覆盖尽可能多的功能。我们需要能够从目标进程的内存空间中进行读取并写入到目标进程的内存空间中。我们也需要能够修改目标进程内部任意线程的上下文。一

个线程的上下文[⊖]包括不同的特定处理器的寄存器数据，比如当前指令指针（EIP），栈指针（ESP）以及在其余通用目标寄存器（EAX,EBX,ECX,EDX,ESI和EDI）中的帧指针。对线程的上下文进行控制将允许我们改变执行的状态，这是在SRM的恢复过程中需要进行的工作。幸运的是，稍后我们将会看到Windows操作系统提供了可以满足我们的所有需求的一个功能强大的API。

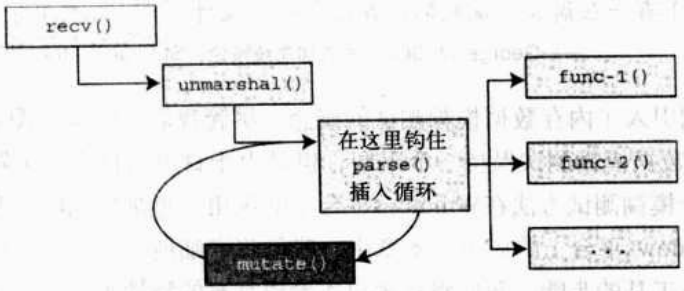


图20-1　变异循环插入的修改需求

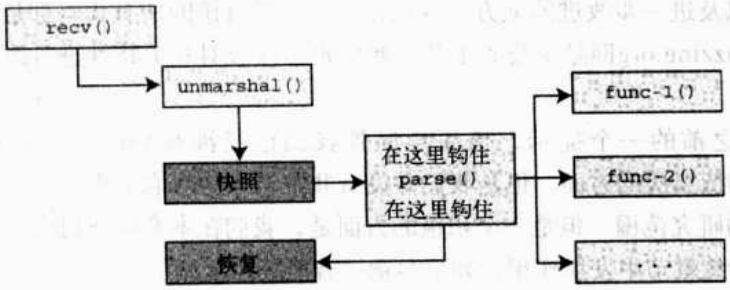


图20-2　快照恢复变异的修改需求

注意

尽管MLI和SRM这两个方法在本质上是类似的，但是我们仍然要特别关注于SRM方法在内存数据模糊器的开发及实现自动化过程中的应用。我们鼓励读者作为一个练习，使用本章中所讨论的例子来实现MLI方法。

SRM具有不同于MLI的一些独特的优点。例如，在SRM中对全局变量的状态进行了恢复，但在MLI却没有。同时，使用MLI你必须要非常小心选择instrumentation点。在这个方面的一个粗心决定将导致一个内存泄漏，从而妨碍模糊测试的成功进行。

20.2　开发语言的选择

在开始讨论有关特定的Windows调试API函数以及我们将要使用的结构的背景知识之前，

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/context_str.asp

让我们首先来确定一下将要使用的开发语言，这样我们就可以采用这种语言来展现一些具有参考价值的代码片段。尽管针对我们的需求而言，我们可能会立即很自然地选择C或者C++，如果我们可以采用一种解释性的语言来编写该工具，那么将是一个完美的选择。采用一种解释性的语言如Perl,Python或Ruby来开发该工具具有许多的优越性。出于我们的目的考虑，它所带来的最大优点就体现在开发速度上，对于读者而言，它将会提高代码的可读性。在发现了由Thomas Heller开发的Python ctypes模块[⊖]之后，我们最终选择了Python作为开发语言。ctypes模块为我们提供了针对Windows API的一个接口，这样就可以非常简单地直接创建并操纵Python内部已编译的C数据类型。例如，下面的代码片段说明了调用被kernel32.dll所导出的GetCurrentProcessId()函数是非常简单的：

```
from ctypes import *

# create a convenience shortcut to kernel32.
kernel32 = windll.kernel32

# determine the current process ID.
current_pid = kernel32.GetCurrentProcessId()

print "The current process ID is %d" % current_pid
```

创建和传递C数据类型同样也非常简单。ctypes模块提供了你所需要的所有基本数据类型，并将其作为内部Python类，如表20-1所示。

表20-1 ctypes中与C兼容的数据类型

ctypes类型	C类型	Python类型
c_char	Char	character
c_int	Int	Integer
c_long	Long	Integer
c_ulong	unsigned long	Long
c_char_p	Char*	string 或者None
c_void_p	Void*	integer或者None

*可用的基本数据类型的完整列表可参阅http://starship.python.net/crew/theller/ctypes/tutorial.html

所有的可用类型可以作为实例的可选初始值来传递。另外，可以通过向value属性赋值来设定值。通过使用byref()帮助函数可以简化通过引用来传递值的实现。需要记住的一个关键点是像c_char_p和c_void_p这样的指针类型是不可变的。为了创建一个可变的内存块，可以使用create_string_buffer()帮助函数。为了访问或者修改可变内存块，可以使用raw属性。下面的代码片段在对ReadProcessMemory()的调用过程中说明了这些内容：

⊖ http://starship.python.net/crew/theller/ctypes/

```
read_buf = create_string_buffer(512)
count    = c_ulong(0)

kernel32.ReadProcessMemory(h_process, \
    0xDEADBEEF, \
    read_buf,   \
    512,        \
    byref(count))

print "Successfully read %d bytes: " % count.value
print read_buf.raw
```

ReadProcessMemory() API的参数包括一个我们想要读取的内存空间进程的句柄、要读取的地址、一个指向被读取数据所保存的缓冲区的指针、想要读取的字节数、最后是指向一个整数的指针，该整数保存了我们能够读取的实际字节的数量。当我们正在进行进程内存空间的读取操作时，我们应当讨论一下进程内存空间的写入问题，如下面的代码片段所示：

```
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 "Sucessfully wrote %d bytes: " % count.value
```

WriteProcessMemory()函数具有类似于其姊妹函数ReadProcessMemory()的格式。它所需要的参数包括一个关于我们想要写入的内存空间进程的句柄，要写入的地址，一个指向包含我们想要写入的数据缓冲区的指针，我们想要写入的字节数量，最后是一个指向一个整数的指针，该整数保存了我们可以写入的实际字节数。由于我们需要具备这样的能力，即可以从进程的内存空间进行读取并写入到该空间以满足我们的一些需求，因此你会看到这些例程将在后续章节中发挥作用。

现在我们已经掌握了工具开发所需要的相关知识，因此让我们简要讨论一下为了创建一个调试器，还需要在API方面了解哪些知识。

20.3 Windows调试API

在第19章中，我们简要讨论了内存的空间布局以及典型的Windows进程的组成结构。在我们开始进一步阐述之前，让我们先来复习关于Windows调试API的一些背景知识，根据你个人的观点，我们将使用或滥用这些知识。

自从Windows NT开始，操作系统就提供了一个功能强大的API函数和结构的集合，以允许开发者相对较容易地来创建一个事件驱动的调试器。调试API的基本组成部分可以划分为如下

的三个类别：函数[㊀]、事件[㊁]和结构[㊂]。由于我们关注于不同需求的特定实现，因此我们将涉及到所有这三个元素。我们必须要完成的第一项工作就是要让目标应用在我们的调试器控制下运行。有两种方法可以实现这一点。一种方法是我们可以在调试器的控制下加载目标进程，另外一种方法是让目标进程自己开始运行，然后再将我们的调试器与其相关联。为了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 "Started process with pid %d" pi.dwProcessId
```

注意在CreateProcess后面所追加的一个A，Windows API通常以Unicode和ASCII的格式被导出。在这种情况下，API的非追加格式是作为一个简单的包装器来使用的。但是对于我们使用ctypes的目的而言，我们必须要调用带有追加的格式。可以通过简单地查看MSDN的Web页来确定一个特定的API是否被作为ANSI和Unicode而导出。例如，关于CreateProcess的MSDN页[㊃]在底部附件有如下的描述："被实现为CreateProcessW（Unicode）以及被实现为CreateProcessA（ANSI）。"PROCESS_INFORMATION和STARTUP_INFO结构被通过引用传递给CreateProcess API，然后被我们后面将需要的信息所填充，比如被创建进程的标识符（pi.dwProcessId）以及被创建进程的一个句柄（pi.hProcess）填充。另外，如果我们想要关联到一个已经正在运行的进程，那么可以调用DebugActiveProcess()：

```
# attach to the specified process ID.
kernel32. DebugActiveProcess(pid)

# allow detaching on systems that support it.
try:
    kernel32.DebugSetProcessKillOnExit(True)
except:
    pass
```

㊀ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_functions.asp

㊁ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_events.asp

㊂ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_structures.asp

㊃ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/createprocess.asp

注意我们在上述代码片段中使用的所有API调用都不需要一个追加的A或者W。DebugActiveProcess()例程将我们的调试器关联到一个特定的进程标识符。在调用DebugActiveProcess()之前，可能必须要提升我们的特权级别，但是让我们稍后再关注此问题。自从Windows XP之后，DebugSetProcessKillOnExit()例程[⊖]就是可用的了，它允许我们不用销毁被调试进程（我们所关联到的那个进程）就可以退出调试器。我们在该调用中加入了一个try/except语句，使得如果我们正在一个不支持被请求的API的平台，例如Windows2000上运行调试器时，可以阻止我们的调试器错误地退出。一旦我们拥有了在调试器控制下的目标进程，我们需要来实现处理调试事件的循环。调试事件循环可以被认为是城镇中守旧的爱管闲事的邻居艾格尼丝（Agnes）。艾格尼丝坐在她家的窗户前面，观看着附近所发生的一切事情。尽管艾格尼丝可以看到任何事情，但是大多数所发生的事情都不是那么有趣以激发她打电话告诉朋友。但是偶尔将会发生一些有趣的事情，例如邻居的小孩将一只猫追赶到了一棵树上，结果摔伤了胳膊，那么艾格尼丝将会立即打电话通知警察。与艾格尼丝非常相像，我们的调试事件循环也将会看到许多的事件。我们需要来指定哪些事件是我们所感兴趣并希望进行更多处理的。下面是一个典型的调试事件循环的框架：

```
debugger_active = True
dbg             = DEBUG_EVENT()
continue_status = DBG_CONTINUE

while debugger_active:
    ret = kernel32. WaitForDebugEvent(byref(dbg), 100)

   # if no debug event occurred, continue.
    if not ret:
        continue

    event_code = dbg.dwDebugEventCode

    if event_code == CREATE_PROCESS_DEBUG_EVENT:
        # new process created

    if event_code == CREATE_THREAD_DEBUG_EVENT:
        # new thread created

    if event_code == EXIT_PROCESS_DEBUG_EVENT:
        # process exited

    if event_code == EXIT_THREAD_DEBUG_EVENT:
        # thread exited

    if event_code == LOAD_DLL_DEBUG_EVENT:
        # new DLL loaded

    if event_code == UNLOAD_DLL_DEBUG_EVENT:
```

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugsetprocesskillonexit.asp

```
        # DLL unloaded

    if event_code == EXCEPTION_DEBUG_EVENT:
        # an exception was caught

    # continue processing
    kernel32.ContinueDebugEvent(dbg.dwProcessId, \
        dbg.dwThreadId, \
        continue_status)
```

调试事件处理循环主要是基于对WaitForDebugEvent()[⊖]的调用，WaitForDebugEvent()函数的第一个参数是指向一个DEBUG_EVENT结构的一个指针，第二个参数是在被调试进程中等待一个调试事件发生所需要的毫秒数。如果发生了一个调试事件，那么DEBUG_EVENT结构将在dwDebugEventCode属性中包含调试事件类型。我们通过检查该变量来确定调试事件是否由于以下原因而被触发：一个进程的创建或退出，一个线程的创建或退出，一个DLL的加载或卸载或者是一个调试异常事件。在调试异常事件发生的情况下，我们可以通过检查u.Exception.ExceptionRecord.ExceptionCode DEBUG_EVENT结构的属性来确定导致该异常的原因是什么。在MSDN上列出了许多可能的异常代码[⊖]，但出于这里的考虑，我们主要是关注以下的异常代码：

- 访问违规异常（EXCEPTION_ACCESS_VIOLATION）。一个访问违规的发生通常是由于试图从一个无效的内存地址进行读取或写入而导致的。
- 断点异常（EXCEPTION_BREAKPOINT）。由于遇到了一个断点而触发了一个异常。
- 单步执行异常（EXCEPTION_SINGLE_STEP）。单步陷阱是可用的，并且一个单步指令被执行。
- 栈溢出异常（EXCEPTION_STACK_OVERFLOW）。违规的线程耗尽了其栈空间。这是一个典型的失控递归的指示符，并且通常只局限于一个DoS。

我们可以推断出希望采用什么样的逻辑来处理不同的调试事件和异常。一旦我们对所报告的事件进行了处理，那么通过调用ContinueDebugEvent()就可以允许违规的线程继续运行。

20.4 将其整合在一起

到此阶段为止，我们已经讨论了Windows内存空间布局的基本知识，列出了我们的需求，选择了所需要的开发语言，阐述了ctypes模块的基本概念，并且涉足了Windows调试API的基础知识。在这其中，有几个突出的问题如下：

- 如何在特定点将“钩子”植入目标进程的需求？
- 如何处理进程快照和恢复？
- 如何选择植入钩子的点？
- 如何对目标内存空间进行定位和变异？

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/waitfordebugenent.asp

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/exception_record_str.asp

20.4.1 如何在特定点将“钩子”植入目标进程

我们在本章前面已经提及，可以通过使用调试器断点来在我们所选择的方法中实现对进程的钩住。在我们所给定的平台上有两种支持的断点类型，即硬件断点和软件断点。与80x86相兼容的处理器支持4个硬件断点。这些断点可以分别被设定以触发读、写、或者执行任意的1字节，2字节或4字节区域。为了设置硬件断点，我们必须要修改目标进程的上下文环境，用从DR3到DR7的调试寄存器来修改DR0寄存器。前4个寄存器包含硬件断点的地址。DR7寄存器包含若干标志，用以指定哪些断点是活动的，在什么区域是活动的以及对何种类型的访问（读，写或者执行）是活动的。硬件断点是非侵入的，并且不会改变你的代码。另一方面，为了实现软件断点，我们必须要修改目标进程，并且使用单字节指令INT3来实现，该指令在十六进制中被表示为0xCC。

我们既可以使用硬件断点也可以使用软件断点来实现给定的任务。如果我们可能会在某些地方需要多于4个的钩子点的话，那么就要决定利用软件断点了。为了更好的理解该处理过程，让我们来分析一个在地址0xDEADBEEF处的一个构想的程序中设置一个软件断点所需要的步骤。首先，我们的调试器必须要使用前面提到的ReadProcessMemory API调用来读取并保存存储在目标地址中的原始字节，如图20-3所示。

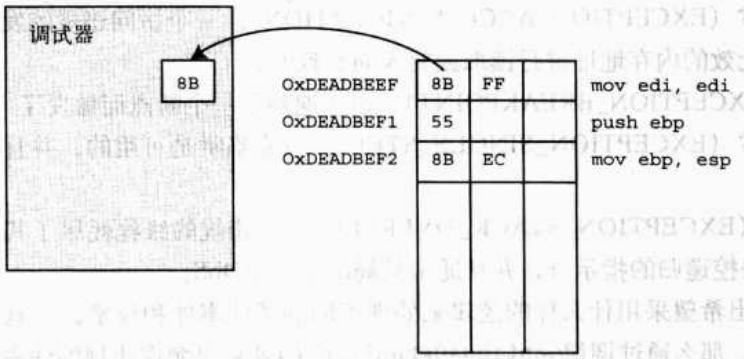


图20-3　保存断点地址处的原始字节

注意目标地址处的第一条指令实际上有两个字节。下一步是使用也是在前面提到的WriteProcessMemory API在目标地址处写入INT3指令，如图20-4所示。

但对前一条指令将会发生什么情况呢？插入的0xCC被作为单字节INT3指令来解析。而mov edi,edi指令的第二个字节（0xFF），与前面所展示的push ebp指令的字节（0x55）以及mov ebp,esp指令的第一个字节（0x8B）组合在一起被解析为call[ebp-75]指令。剩余的字节0xEC则被解析为单字节指令in al,dx。这时，当执行到地址0xDEADBEEF时，INT3指令将触发一个带有一个EXCEPTION_BREAKPOINT异常代码的EXCEPTION_DEBUG_EVENT调试事件，我们的调试器将在调试事件循环中捕获该代码（还记得艾格尼丝的故事吗？）。图20-5显现了此时的进程状态。

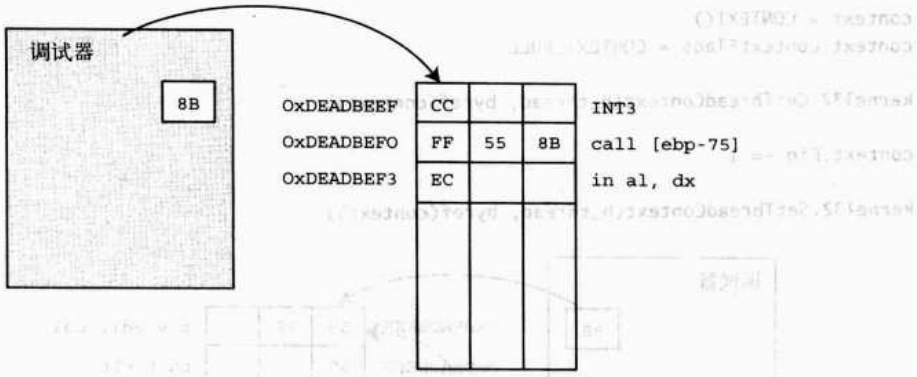


图20-4 写入INT3操作码

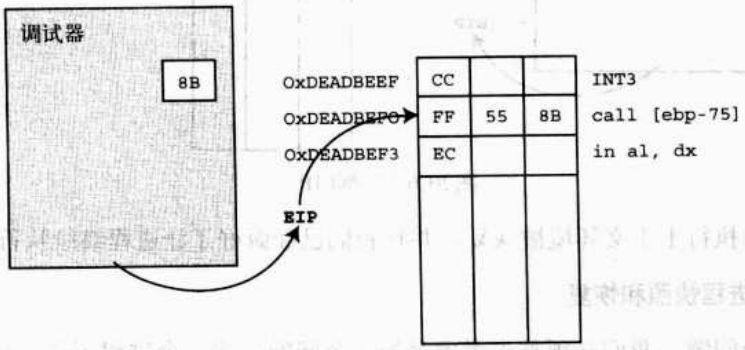


图20-5 捕获的EXCEPTION_BREAKPOINT代码

因此我们现在已经成功地插入并捕获了一个软件断点，但这只完成了一半的工作。注意我们的原始指令仍然没有发挥作用。另外，指令指针（即EIP，它使CPU知道从哪里获取，解析并执行下一条指令）位于0XDEADBEF0而不是0XDEADBEEF。这是由于我们在地址0XDEADBEEF插入的单字节INT3指令被成功执行，从而导致EIP被更新为0XDEADBEEF+1。在我们可以继续执行之前，必须要纠正EIP的值，并且恢复地址0XDEADBEEF处的原始值，如图20-6所示。

恢复地址0XDEADBEEF处的字节是我们已经熟悉的一项工作。然而，改变指令指针的值，即寄存器EIP的值则是一项不同的任务。我们在本章前面曾提到过线程的上下文环境包含不同的特定于处理器的寄存器数据，例如我们现在所关注的指令指针（EIP）。我们可以通过调用GetThreadContext() API[⊖]来为任意给定的线程检索上下文，并向其传递当前线程的句柄以及一个指向CONTEXT结构的指针。然后，我们就可以修改CONTEXT结构的内容，并调用SetThreadContext() API[⊜]，再一次传递当前线程的句柄以修改上下文：

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/getthreadcontext.asp

⊜ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/setthreadcontext.asp

```
context = CONTEXT()
context.ContextFlags = CONTEXT_FULL

kernel32.GetThreadContext(h_thread, byref(context))

context.Eip -= 1

kernel32.SetThreadContext(h_thread, byref(context))
```

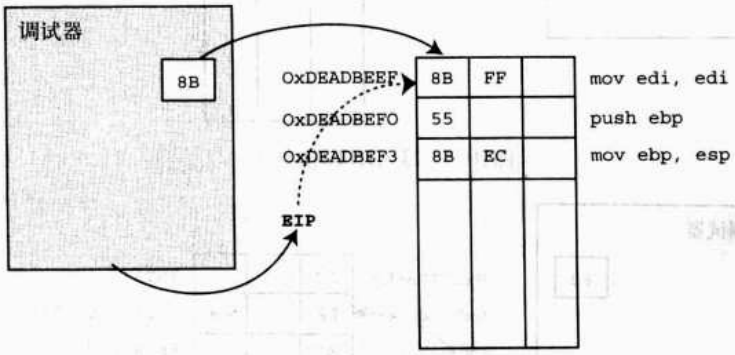


图20-6　调整EIP

这时，原始的执行上下文环境被恢复，并且我们已经做好了让进程继续执行的准备。

20.4.2　如何处理进程快照和恢复

为了回答这个问题，我们必须首先考虑另外一个问题：当一个进程正在运行时，什么会发生改变呢？答案是有许多内容都会发生变化。新线程被创建并销毁。针对文件、socket、窗口以及其他元素的句柄被打开和关闭。内存被分配和释放，被读取和写入。在单个线程上下文环境中的不同寄存器具有高度的不稳定性，并且经常被改变。我们可以使用一个虚拟机技术如VMWare[⊖]来完成我们的任务，VMWare允许我们获得并恢复完整的系统快照。然而，该处理过程非常缓慢，并且要求虚拟机客户和虚拟主机上某种形式的仲裁器之间进行通信。相反，我们借鉴于前面所讨论的一种技术[⊜]，并且只考虑线程上下文环境和内存的变化。我们的快照过程将需要两个步骤。

在第一步中，我们将目标进程内部的每个线程的上下文加以保存。我们已经看到对单个线程的上下文环境进行获取和设置是非常简单的。现在我们将在该代码中加入处理逻辑，以枚举出属于目标进程的系统线程。为此，我们要依赖于工具帮助函数[⊝]。首先，我们通过指定TH32CS_SNAPTHREAD标志来获取所有系统线程的一个列表：

```
thread_entry = THREADENTRY32()
contexts     = []
```

⊖ http://www.vmware.com

⊜ Greg Hoglund,运行时反汇编,BlackHat会议

⊝ http://msdn2.microsoft.com/en-us/library/ms686832.aspx

```
snapshot = kernel32.CreateToolhelp32Snapshot( \
    TH32CS_SNAPTHREAD,                        \
    0)
```

接下来，我们从该列表中检索第一个线程项。然而在开始这项操作之前，Thread32First() API有一个强制性的要求，即要将线程项结构内部的dwSize变量进行初始化。我们将前面获得的快照以及指向线程项结构的一个指针作为参数传递给Thread32First() API：

```
thread_entry.dwSize = sizeof(thread_entry)

success = kernel32.Thread32First( \
    snapshot,                     \
    byref(thread_entry))
```

最后，我们循环遍历线程列表，以寻找属于目标进程的进程ID（pid）的那些线程。如果存在一些这样的线程，我们就使用OpenThread() API来检索一个线程句柄，并且像前面那样检索上下文环境，然后将其追加到列表中：

```
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))
```

在保存了属于我们的进程的每个线程的上下文环境之后，就可以通过再次循环遍历所有的系统线程，并恢复所保存的在前面所看到的任意线程的上下文，为了以后恢复我们的进程快照。

在第二步中，我们保存了每个可变内存块的内容。回忆一下上一章中所讲的，每个在32位x86 Windows平台上的进程可以“看到”它自己的4GB内存空间。在这4GB空间中，通常是底层的一半空间（0x00000000-0x7FFFFFFF）被留作我们的进程来使用。该内存空间进一步被划分为单独的页，通常每个页的大小是4 096字节。最后，内存许可被应用在这些单独页的每个页的最细粒度上。不是将每个被单独使用的内存页的内容加以存储，而是通过将快照限制于那些我们认为是可变的内存页来把时间和资源同时加以保存。我们将忽略那些具有限制性的许可，即阻止写入的所有页。这包括带有如下标记的页：

- 只读页（PAGE_READONLY）
- 可执行可读页（PAGE_EXECUTE_READ）
- 保护页（PAGE_GUARD）
- 不允许访问的页（PAGE_NOACCESS）

我们也想要忽略属于可执行映射的那些页，因为它们不太可能发生变化。遍历所有可用的内存页需要在VirtualQueryEx()[⊖]API例程外面加入一个简单的循环，以提供关于在指定虚拟地址范围内的页的信息：

```
cursor        = 0
memory_blocks = []
read_buf      = create_string_buffer(length)
count         = c_ulong(0)
mbi           = MEMORY_BASIC_INFORMATION()
while cursor < 0xFFFFFFFF:
    save_block = True

    bytes_read = kernel32.VirtualQueryEx( \
        h_process,                        \
        cursor,                           \
        byref(mbi),                       \
        sizeof(mbi))

    if bytes_read < sizeof(mbi):
        break
```

如果对VirtualQueryEx()的调用执行失败，那么就假定我们已经耗尽了可用的用户空间，然后退出读取循环。对于在循环中所发现的每个内存块，我们检查有利的页许可：

```
if mbi.State != MEM_COMMIT or \
   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
```

如果我们偶然遇到了想要包含在快照中的一个内存块，那么我们就使用ReadProcessMemory() API[⊜]来读取该内存块的内容，并且将原始数据同内存信息一样保存到快照列表中。

⊖ http://msdn2.microsoft.com/en-us/library/aa366907.aspx

⊜ http://msdn2.microsoft.com/en-us/library/aa680553.aspx

最后，我们递增内存扫描指针并继续执行：

```
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
```

你可能会立刻注意到我们采用的方法是有缺陷的。如果一个给定页在快照的时候被标记为只读页（PAGE_READONLY）而后来又被更新为可写的和可修改的，那么情况将会是怎样呢？答案是我们遗漏了这种情形。我们从来没有许诺说该方法是完美的！实际上，让我们利用这次机会来再次强调一下该课题所具有的实验本质。对于具有求知欲的读者来说，对此缺陷的一个潜在的解决方法是“钩住”那些调整内存许可的函数，并且基于观察到的变化来修改我们的监视。

将我们所提出的两个步骤结合在一起，就提供了获取和恢复特定内存快照所需要的元素。

20.4.3 如何选择植入钩子的点

这是我们从科学方法转向艺术技巧的转换点，因为目前没有确定的方法能够帮助我们选择植入钩子的点。做出这个决定需要具备逆向工程的丰富经验。从本质上来说，我们正在寻找负责解析用户控制数据的代码起始点和结束点。假定你对目标应用没有任何先验知识，使用一个调试器来进行跟踪是缩减决策范围的一个好方法。我们将在第23章“模糊器跟踪”中对调试器跟踪进行更加详细的介绍。为了更好的理解这一点，我们稍后将分析一个例子。

20.4.4 如何对目标内存空间进行定位和变异

在需要具备的所有先决条件中，最后一个步骤就是选择实施变异的内存位置。同样，完成这个过程更多的是需要艺术技巧，而不是太多的科学知识。使用一个调试器来进行跟踪将有助于我们的搜索。然而在通常情况下，我们应当选择一个初始的植入钩子的点，比如指向目标数据或者邻近的下一个内存空间的一个指针。

20.5 一个最好的新工具PyDbg

到现在为止，你无疑已经推断出编写一个调试器需要进行大量的工作。但幸运的是，到目前为止我们所遇到的所有内容（或者更多的内容）都已经有开发者为你编写好了，他们将其写入一个被称为PyDbg[⊖]的方便的Python类中。你可能会非常困惑地问自己，为什么我们在前面

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

没有提到该类呢？我们是在有意隐瞒它。因为理解这些基础知识是非常重要的，如果我们在一开始就告诉你这个捷径，那么你将不会对这些知识给予足够的关注。

使用PyDbg，可以非常容易通过Windows调试API来执行进程instrumentation。使用PyDbg可以非常简单地完成如下功能：

- 读内存，写内存以及查询内存。
- 枚举进程，关联到进程，与进程相分离，终止进程。
- 枚举线程，暂停线程的执行，恢复线程的执行。
- 设置断点，删除断点，处理断点。
- 快照以及恢复内存状态（SRM中的SR）。
- 确定函数地址。
- 其他功能。

考虑如下的简单例子，我们初始化了一个PyDbg对象，将其在PID123关联到目标进程，并且进入了一个调试循环：

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

dbg = new pydbg()
dbg.attach(123)
dbg.debug_event_loop()
```

这并不是那么的令人兴奋，因此让我们再添加一些更多的功能。以上一个例子为基础，在下一个代码片段中，我们在Winsock recv() API的调用中设置了一个断点，并且当遇到一个断点时，记录一个回调函数的句柄以被调用：

```
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 = new pydbg()
dbg.set_callback(EXCEPTION_BREAKPOINT, handler_bp)
dbg.attach(123)

ws2_recv = dbg.func_resolve("ws2_32", "recv")
dbg.bp_set(ws2_recv)

dbg.debug_event_loop()
```

上面代码片段中的粗体文本强调了我们所增加的内容。你看到的第一个所增加的内容是我们定义了一个函数handler_bp()，该函数包含三个参数。第一个参数接收我们所创建的PyDbg的实例。第二个参数接收来自于调试事件循环的DEBUG_EVENT结构[⊖]，并且包含刚刚发生的关于调试事件的一些不同信息。第三个参数接收发生调试事件的线程的上下文环境。我们的断点处理器简单地检查发生异常的地址与Winsock recv() API的地址是否是相同的地址。如果是同一个地址，那么就输出一条消息。断点处理器返回DBG_CONTINUE以告知PyDbg我们已经完成了对异常的处理，并且PyDbg应当让目标进程继续执行。回过头来看一下调试器脚本的主体部分，你将会看到增加了对PyDbg例程set_callback()的一个调用。该例程用做为PyDbg记录一个回调函数，以处理一个特定的调试事件或者异常。在这种情况下，只要遇到一个断点，那么就将调用handler_bp()。最后，我们看到增加了对func_resolve()和bp_set()的调用。前面一个调用用来确定recv() API在Windows模块ws2_32.dll中的地址，并且将其保存到一个全局变量中。后一个调用用来在所确定的地址处设置一个断点。当关联到一个目标进程时，任何对Winsock recv() API的调用都将导致调试器显示“ws2.recv()被调用”的消息，然后继续正常执行。同样，这也不是非常令人兴奋，但是现在我们就可以进一步向前推进，并且创建第一个内存数据模糊器的概念验证工具。

20.6 一个构想的示例

在已经了解了一些背景知识以及必备的先决条件之后，现在我们就可以建立一个最初的概念验证工具，以告诉你所有这些理论构想实际上都是可行的。在fuzzing.org Web站点上，你会看到包含源代码的两个文件fuzz_client.exe和fuzz_server.exe。先不要马上研究源代码。为了创建一个更加接近于实际情况的场景，让我们假定逆向工程是必需的。客户机——服务器模式是非常简单的目标应用。当服务器部署之后，它绑定到TCP端口11427并且等待一个客户端的连接。客户端进行连接并向服务器发送数据，稍后该数据被解析。那么数据是如何被解析的呢？实际上我们并不知道，但是我们对此并不关心，因为我们的目的是对目标应用进行模糊测试，而不是对源代码或二进制代码进行评审。让我们首先来部署服务器：

```
$ ./fuzz_server.exe
Listening and waiting for client to connect...
```

接下来，我们要部署客户端，该操作需要两个参数，第一个参数是服务器的IP地址，第二个参数是发送给服务器以被解析的数据：

```
$ ./fuzz_client.exe 192.168.197.1 'sending some data'
connecting....
sending...
sent...
```

客户端在IP地址192.168.197.1处成功地连接到了服务器，并且传递了字符串“sending some data”。在服务器端，我们看到了如下的附加消息：

⊖ http://msdn2.microsoft.com/en-us/library/aa686832.aspx

```
client connected.
received 17 bytes.
parsing: sending some data
exiting...
```

服务器成功接收到了我们所发送的17个字节，并对其进行解析，然后退出。然而，当对网络上所传输的包进行检查时，却无法定位我们的数据。该包应当在TCP的三次握手过程之后包含我们的数据，但是它实际包含了来自于Ethereal[⊖]的屏幕快照中突出显示的那些字节，如图20-7所示。

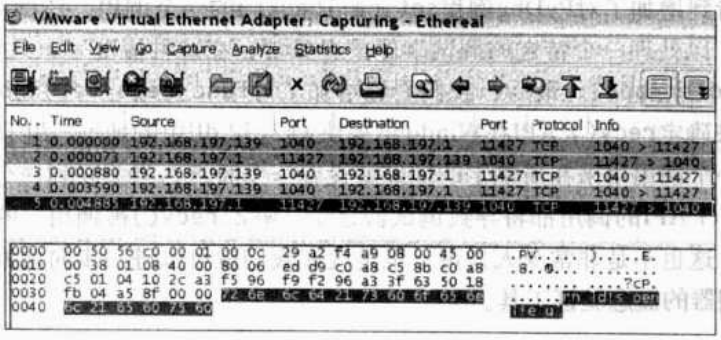


图20-7　Ethereal的客户端——服务器数据捕获

Ethereal网络嗅探器项目已经转向了Wireshark，该项目可以从http://www.wireshark.org处下载

客户端必须要在将包数据写入到网络之前，对数据进行干扰、加密、压缩或者以别的方式进行模糊化处理。在服务器端，必须要在解析包数据之前对该数据进行反模糊化处理，因为我们可以在输出的消息日志中看到正确的字符串。以传统的方式对我们的示例服务器进行模糊测试要求我们对模糊例程实施逆向工程。一旦模糊方法中的秘密被揭露，我们就可以生成并发送任意的数据。当你在后面看到解决方案时，你将会意识到在这种情况下完成这项操作将会非常容易。然而，为了继续使用我们的例子，我们假定所采用的模糊化方法将利用相当多的资源来进行逆向工程。这是一个极好的例子来说明在什么时候执行内存数据模糊测试是具有实际价值的（毕竟这是一个构想的例子）。我们将通过当所传输的数据被反模糊化之后在fuzz_server.exe中植入钩子的方法，来避免对模糊例程进行解析。

我们需要在fuzz_server.exe中查明两个位置以完成我们的任务。第一个位置是快照点。我们想要在哪一个执行点来保存目标进程的状态呢？第二个位置是恢复点。我们想要在哪一个执行点来恢复目标进程的状态，变异我们的输入并且继续我们的instrumentation执行循环呢？对这两个问题的回答都需要使用一个调试器来对输入进行一些跟踪。我们将使用一个免费可用的、功能强大的Windows用户模式的调试器OllyDbg[⊜]。关于OllyDbg的用法和功能，已经有许多专门书籍进行了描述，因此我们假定你已经熟悉了这些内容。我们所需要做的第一件事就是确定

⊖ http://www.ethereal.com/,Ethereal:A Network Protocol Analyzer

⊜ http://www.ollydbg.de

fuzz_server.exe在哪里接收数据。我们通过TCP来实现这一点，因此我们将fuzz_server.exe加载进OllyDbg，并且在WS2_32.dll recv() API中设置一个断点。我们是通过下述操作来实现这一点的，即提出模块列表，选择WS_32.dll，并且按下Ctrl+N键来显示出在该模块内部的名字的列表，如图20-8所示。然后我们拖动滑块到recv()，并且按下F2键来使该断点可用。



图20-8　OllyDbg在WS_32.recv()的断点

在设置了断点之后，我们按下F9键来继续执行，然后就像以前所做的那样运行我们的fuzz_client。一旦数据被发送，由于遇到了我们所设定的断点，因此OllyDbg就暂停fuzz_server。于是我们就按下Alt+F9键来“执行直到用户代码为止”。这时，从fuzz_server对WS2_32的调用就是可见的了。我们按下几次F8键并且略过对printf()的调用，该函数显示了服务器消息以说明有多少字节被接收。接下来，如图20-9所示，我们看到在fuzz_server内部地址为0x0040100F的地方有一个对未命名子例程的调用。在OllyDbg的输出窗口中观察该函数的第一个参数，我们就会发现它是一个指向我们在Ethereal中所看到的数据的模糊形式的指针。处于地址0x0040100F的例程能够将包数据反模糊化吗？

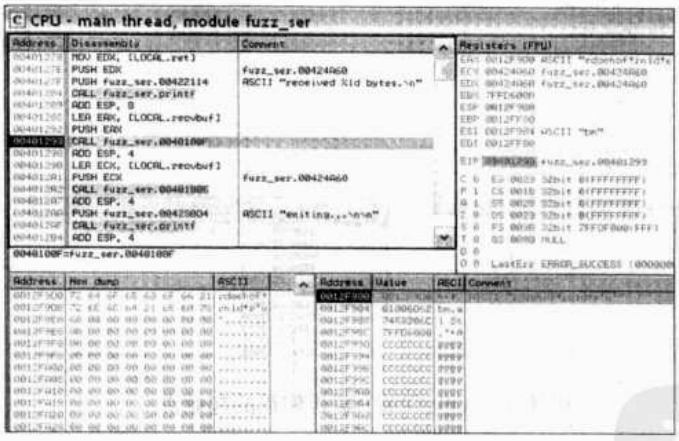


图20-9　在反模糊化之前的OllyDbg

有一个简单的方法可以得到该问题的答案：让fuzz_server继续执行并观察所发生的情况。我们再次按下F8键以略过对printf()的调用。我们可以立刻看到如图20-10所示，在OllyDbg输出窗口中的模糊化数据已经被转换了。

这很好。我们由此就知道了快照点必须要放在该例程之后的某个点。再接着往下看，我们在地址0x00401005处发现了一个对未命名例程的调用，然后是一个对printf()的调用。我们可以看到字符串“exiting...”作为一个参数传递给了printf()，再回忆一下前面所观察到的fuzz_server的行为，我们就知道它很快将要退出。处于地址0x00401005的例程必定是我们的解

析例程。使用F7键来单步执行它，就会发现一个转向地址0x00401450的无条件跳转，图20-11显示了我们所猜测的解析例程的起始点。

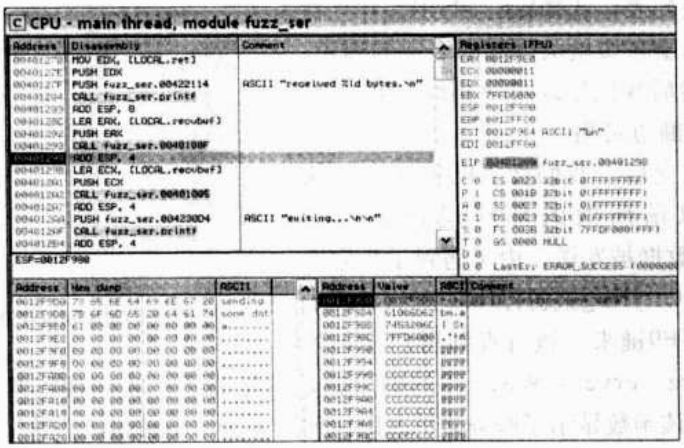


图20-10　在反模糊化之后的OllyDbg

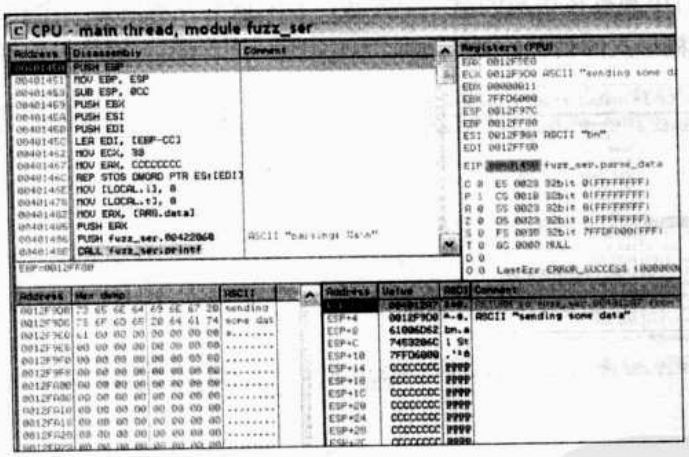


图20-11　OllyDbg中解析例程的起始点

注意我们的反模糊化字符串“sending some data”是作为在ESP＋4地址处的解析例程的第一个参数。这对于我们的快照钩子来说，似乎是一个很好的植入点。我们可以将进程的整个状态保存到解析例程的起始点，然后在解析例程执行完之后的某个点再将进程的状态进行恢复。一旦我们恢复了该进程，我们就可以修改将要被解析的数据的内容，然后继续执行，并且不确定地重复该过程。但是首先我们需要定位恢复点。按下Ctrl+F9键来“执行直到返回为止”，然后按下F7或者F8键以到达我们的返回地址，如图20-12所示。在该地址处我们再次看到了使用字符串“exiting[el]”对printf()的调用。让我们在位于地址0x004012b7处的对printf()的调用之后选择一个恢复点，这样我们就可以看到在恢复点之前fuzz_server输出了“exiting[el]”。我们之所以这样做，是因为它能够让我们更好地理解fuzz_server想要退出，但是我们不让它退

出这样一个思想。

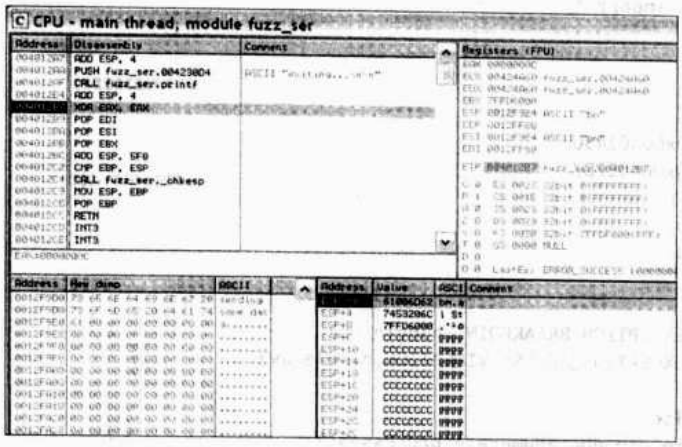


图20-12 OllyDbg中的恢复点

我们知道处于地址0x0040100F处的例程是负责解码的。而位于地址0x00401450处的例程是负责解析被解码的数据的。我们已经选择了解析器的起始点作为快照和变异点。我们略微随意地选择地址0x004012b7作为对printf("exiting[el]")的调用之后的恢复点。我们已经掌握了所有必需的内容，现在可以开始具体编码了。图20-13展示了我们将要进行的工作的一个概念视图。

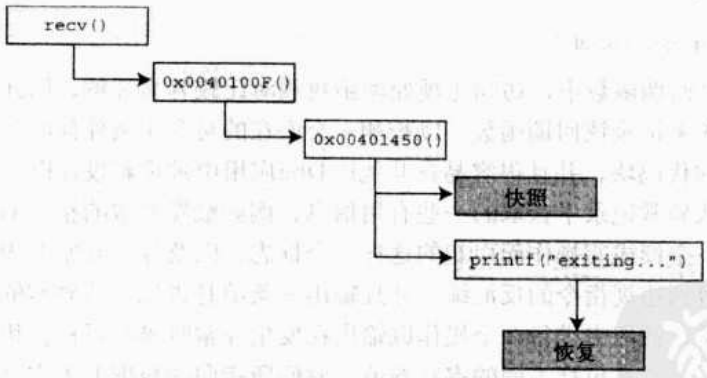


图20-13 概念视图

使用PyDbg来实现内存数据模糊器所需要进行的工作，并不比我们已经讨论的这些工作要多。我们从所需要的输入开始做起，通过定义全局变量来存储一些信息，比如我们所选择的快照点和恢复点。然后我们将进入标准的PyDbg的框架代码，初始化一个PyDbg对象，保存回调函数（我们稍后将定义两个回调函数，它们之间类似于肉和马铃薯的关系），定位我们的目标进程，然后关联到目标进程，在快照点和恢复点设置断点，并且最后进入调试事件循环：

```
from pydbg import *
from 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 "entering debug event loop"
    dbg.debug_event_loop()
else:
    print "target not found."
```

在所定义的两个回调函数中，访问违规处理函数相对比较容易理解，因此我们首先将对它进行讨论。我们首先来记录该回调函数，以检测一个潜在的可利用条件何时发生。这是一个相当简单的易于理解的代码块，并且很容易在其他PyDbg应用中被重新设计以用做其他目的。在例程的起始处，是从异常记录中获取的一些有用信息，例如触发异常的指令的地址，表明了访问违规是否是由于一个读或写操作而引起的这样一个标志，以及导致异常的内存地址。试图进行一个操作以重新得到违规指令的反汇编，并且输出一条消息告知你该异常的类别。最后，在终止被调试进程之前，试图来进行一个操作以输出在发生异常时候的目标应用的当前执行上下文信息。该执行上下文信息包括不同的寄存器值，它们所指向的数据的内容（如果它是一个指针的话），以及栈解除引用的一个可变数量（在这个例子中，我们指定为5）。关于PyDbg API的特定方面的更加详细的内容，可以参阅位于http://pedram.redhive.com/PaiMei/docs/PyDbg处的一些进行深入分析的文档。

```
def handle_av (pydbg, dbg, context):
    exception_record  = dbg.u.Exception.ExceptionRecord
    exception_address = exception_record.ExceptionAddress
    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，我们可以使用其他的调试器如OllyDbg来捕获访问违规。为此，你所选择的调试器必须要被操作系统所配置，以作为运行时编译执行（JIT）[⊖]的调试器。于是，访问违规处理函数的主体部分就可以用如下代码来替换：

```
def handle_av (pydbg, dbg, context):
    pydbg.detach()
    return DBG_CONTINUE
```

当发生一个访问违规时，将显示一个我们所熟悉的对话框，如图20-14所示。

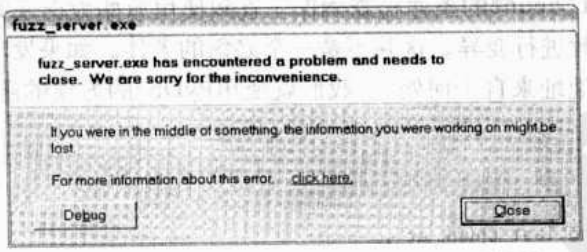


图20-14 fuzz_server关闭

如果选择该对话框中的Debug按钮，那么将会弹出你已注册的JIT调试器，以便于更详细地查看在目标应用中究竟是什么出现了错误。在我们的内存数据模糊器中，所要介绍的最后一个组件是该工具的核心，即断点处理器。只要fuzz_server的执行到达了快照点和恢复点，那么断

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/debugging_terminology.asp

点处理器就将在我们以前设置断点的位置被调用。这是我们所看到的最复杂的PyDbg代码块，因此让我们逐步来解析该代码。在该函数的开头我们定义了一些将要访问的全局变量，并且将获取的断点发生的地址作为exception_address。

```
def handle_bp (pydbg, dbg, context):
    global snapshot_hook, restore_hook
    global snapshot_taken, hit_count, address
exception_address = \
  dbg.u.Exception.ExceptionRecord.ExceptionAddress
```

然后，我们进行检查以确定是否处于快照钩子点。如果是，我们将递增hit_count变量，并且输出一条关于我们当前位置的信息。

```
if exception_address == snapshot_hook:
    hit_count += 1
    print "snapshot hook hit #%d\n" % hit_count
```

接下来，我们要检查布尔标志snapshot_taken。如果fuzz_server的一个快照在以前没有被获取过，那么就通过调用PyDbg的process_snapshot()例程来获取该快照。我们在该操作中加入了一个定时器以确定执行时间等信息，并且将snapshot_taken标志更新为True。

```
# if a process snapshot has not yet been
# taken, take one now.
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
```

要注意我们仍然在if块中，在这里异常地址与钩子点的地址是相同的。下一个代码块将考虑数据的变异。针对hit_count的条件检查确保了直到使用原始数据来第一次循环遍历解析例程之后，我们才对该参数进行变异。这并不是一个必需的条件。如果发现了一个以前所分配的地址（很快就会清楚该地址来自于何处），我们就使用PyDbg的方便的函数virtual_free()来释放它。

```
if hit_count >= 1:
    if address:
        print "freeing last chunk at",
        print "%08x" % address
        pydbg.virtual_free( \
            address,        \
            1000,           \
            MEM_DECOMMIT)
```

由于我们仍然处于hit_count>=1的if块中，因此使用PyDbg的方便的函数virtual_alloc()来在fuzz_server的进程空间中分配一个内存块。这是我们刚看到的一个空闲空间的分

配。为什么我们首先要分配内存呢？因为比起在适当的位置修改传递到解析例程的原始数据，将我们的变异数据放置到fuzz_server的进程空间的其他地方要容易的多，并且可以简单地将指向原始数据的指针修改为指向变异块的指针。在这里需要注意的一件事情是潜在的栈破坏可能会向堆破坏那样发生，因为漏洞缓冲区已经潜在的从栈中被移出了。

```
print "allocating memory for mutation"
address = pydbg.virtual_alloc( \
    None,                      \
    1000,                      \
    MEM_COMMIT,                \
    PAGE_READWRITE)
print "allocation at %08x\n" % address
```

我们假定服务器只能够解析ASCII数据，并且使用一个简单的数据生成算法来生成模糊数据以填充到被分配的变异块中。以一个包含A字符的长字符串开始，我们为该字符串选择一个随机的索引并插入一个随机的ASCII字符。这是个足够简单的例子，如下所示：

```
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的方便的write_process_memory()例程将模糊数据写入到前面所分配的内存块中：

```
print "writing mutant to target memory"
pydbg.write_process_memory(address, mutant)
print
```

最后，我们修改函数参数的指针，以使其指向我们新分配的内存块。回忆一下图20-11所示的情形，指向包含我们原始的反模糊化数据缓冲区的指针，位于我们当前的栈指针的4个正向偏移量的位置处。然后我们继续执行。

```
print "modifying function argument"
pydbg.write_process_memory( \
    context.Esp + 4,        \
    pydbg.flip_endian(address))
print

print "continuing execution...\n"
```

再回过头来看一下断点回调函数处理器定义中的剩余部分，我们看到还有最后一个if块，其作用是当到达恢复钩子点时处理快照的恢复。这一点可以通过调用PyDbg的process_restore() API来简单地实现。同样，我们在该块中增加了一个计时器以获取有关时

间信息：

```
if exception_address == restore_hook:
    start = time.time()
    print "restoring process snapshot...",
    pydbg.process_restore()
    end = time.time() - start
    print "done. took %.03f seconds\n" % end
    pydbg.bp_set(restore_hook)

return DBG_CONTINUE
```

下面我们就可以进行实际的测试了，首先来启动服务器：

```
$ ./fuzz_server.exe
Listening and waiting for client to connect...
```

然后启动内存数据模糊器：

```
$ ./chapter_20_srm_poc.py
entering debug event loop
```

接着启动客户端程序：

```
$ ./fuzz_client.exe 192.168.197.1 'sending some data'
connecting....
sending...
sent...
```

只要客户端的传输量达到了有效载荷，那么就到达了快照钩子点，并且我们的内存数据模糊器开始运转：

```
snapshot / mutate hook point hit #1
taking process snapshot... done. took 0.015 seconds
```

在获取了一个快照之后，执行将继续进行。解析过程完成，fuzz_server打印其输出消息，然后等待退出：

```
received 17 bytes.
parsing: sending some data
exiting...
```

然而在它得到退出的机会之前，将到达恢复钩子点，并且我们的内存数据模糊器将再次开始运转：

```
restoring process snapshot... done. took 0.000 seconds
```

fuzz_server的进程状态已经成功回退到我们的快照点，并且执行将继续进行。这一次，由于hit_count的值现在要大于1，因此处于断点处理器内部的变异块将被执行：

```
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

continuing execution...
```

内存将被分配，一个模糊变异体生成并被写入fuzz_server，并且参数指针被修改。当继续执行时，fuzz_server的输出消息将证实我们的第一个快照成功恢复了变异：

```
parsing:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA)AAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
exiting...
```

注意上面用粗体突出显示的、插入在A字符中间的）字符。同样，fuzz_server想要退出，但是从没有得到退出的机会，因为进程状态被回退到恢复点。在下一次迭代过程中，以前所分配的内存块将被释放，一个新的内存块将被创建并分配，然后进程继续执行。尽管在不同的机器上运行这个例子将会产生不同的结果，但是在本书中运行该例子的目的是为了得到如下的结果：

```
continuing execution...

restoring process snapshot... done. took 0.016 seconds

snapshot / mutate hook point hit #265

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
```

在基于SRM的概念验证模糊器的第256次迭代过程中，在我们的例子目标应用中发现了一个明显的可利用漏洞。访问违规@41414141意味着进程试图从虚拟地址0x41414141处读取并执行一条指令，结果该操作失败，因为该地址处的内存是不可读的。如果你现在还没有猜到的话，

0x41其实就是ASCII字符A的十六进制表示。来自于我们的模糊器的数据导致了一个溢出，并且覆盖了栈上的一个返回地址。当发生溢出的函数将执行权返回给调用者时，就会发生访问违规，并且被我们的模糊器所获取。攻击者可以非常容易地利用这个漏洞，但是不要忘记，模糊例程将必定要被解释以攻击服务器（在现实情况中，你可能不会很容易地命令客户端发送任意的数据）。分析相关的源代码或二进制代码，以查明该漏洞的本质以及导致该漏洞原因，留作一个练习供读者完成。

20.7　小结

我们已经花费了很长的篇幅来讨论一个崭新的、理论性很强的模糊测试方法。该理论是非常有趣的，并且值得我们去阅读分析该理论的一些具体应用。然而，我们鼓励读者去下载示例文件，并且实际去执行示例测试。只通过书本所提供的这种非交互式的媒介方式，不能够很好地理解这些例子。

最后两章的内容很好地激发了读者的阅读兴趣，它所提供的方法可能会应用于你所面对的一个特定问题的某些方面。PyDbg平台和示例应用程序都是开源的，并且可以自由地进行修改。可以从http://www.fuzzing.org处下载它们并进行分析研究。同我们所有的项目一样，我们希望读者为我们提供有关该工具的改进意见、错误报告、补丁程序以及实例应用等信息，共同努力来使该工具不断地得到更新并获得动态的发展。



第三部分 高级模糊测试技术

第21章 模糊测试框架

“在田纳西州有一个古老的谚语（我知道它是在得克萨斯州，也可能是在田纳西州），是这样说的：一旦你欺骗我，那么为你感到羞愧。欺骗我——你不会再次受到欺骗。”

——George W. Bush，Nashville，TN，2002年9月17日

这里有一些可用的、专门化的模糊测试工具，它们面向许多普通的并且公开的网络协议和文件格式。这些模糊器可以对一个指定的协议进行完整的测试，也可以被扩展用来对支持该协议的不同应用程序进行压力测试。例如，同样一个专门的SMTP模糊器可以被用来对一些不同的邮件传输程序进行模糊测试，如微软的Exchange、Sendmail、qmail等等。其他的模糊器采用一种更加通用的方法以允许对任意的协议和文件格式进行模糊测试，并且执行简单的、不被协议所意识到的变异，如位翻转以及字节置换。

尽管这些模糊器对范围广阔的普通应用而言是有效的，但是我们经常需要针对私有的、以前没有被测试过的协议实施更加专用化的、更加彻底的模糊测试。在这个时候，模糊测试框架就变得非常有用了。

在本章中，我们将研究一些目前可用的开源的模糊测试框架，其中包括SPIKE，它是一个曾经非常流行的框架，现在已经变成了家族名（这要取决于你的家族的族谱）。同时，我们还要讨论一些在该领域中新出现的、令人感兴趣的框架，如Autodafé和GPF。在对现有的技术进行分析之后我们将会看到，尽管许多通用的模糊测试框架已经提供了强大的功能，但是我们仍然需要在某些时候重新创建一个模糊器。我们稍后将通过一个模糊测试问题以及针对该问题的解决方案的实际例子来说明这一点。最后，我们将介绍一个由本书作者所开发的新的模糊测试框架，并研究该框架所具有的特点。

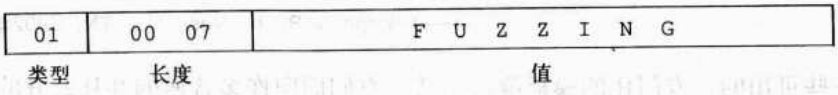
21.1 模糊测试框架的概念

目前可用的一些模糊测试框架是用C语言编写的，而其他的框架则是用Python或者Ruby来编写的。有一些框架是通过开发语言自身来实现的，而其他框架则是利用了一个定制的语言。例如，Peach模糊测试框架是使用Python来构建的，而dfuz则实现了它自己的模糊测试对象集（本章的后面将对这两种框架进行更加详细的讨论）。有一些框架将数据的生成进行了抽象，而另外一些则没有。还有一些框架是面向对象并且公开的，而其他的框架在大多数时候只能被创

建者所使用。然而，所有模糊测试框架的共同目标都是相同的，即为模糊器的开发者提供一个快速的、灵活的、可重用的以及同构的开发环境。

一个好的模糊测试框架应当将一些单调的工作进行抽象化，并且将这些工作减少到最少的程度。为了帮助实现协议建模的第一个阶段，有一些框架包含这样的工具，即将被捕获的网络通信转换为该框架能够理解的一种格式。这样以来，研究者就可以导入大量的经验数据，并且将其关注的重点放在更加适合于人工完成的工作上，比如确定协议字段的边界。

对于一个成熟的框架来说，自动化的长度计算是绝对必需的。许多协议是使用类似于ASN.1[⊖]标准的一个TLV（类型、长度、值）风格的语法来实现的。考虑这样一个例子，数据通信的第一个字节定义了后面跟着的数据的类型：0x01表示纯文本，0x02表示原始二进制数据。接下来的两个字节定义了后面数据的长度。最后，剩余的字节定义了特定于通信的值或者数据，如下所示：



当对这个协议的值字段进行模糊测试时，我们必须要在每个测试用例中计算并更新该协议中两个字节的长度字段。否则的话，当违背协议规范而进行通信时，我们的测试用例将面临无法被执行的风险。CRC（Calculating Cyclic Redundancy Check）[⊜]计算以及其他校验和算法作为完成其他任务的工具，应当被一个有用的框架所包含。CRC值通常嵌入到文件和协议规范中以识别潜在的被破坏的数据。例如，如果接收到的CRC与所计算的CRC值不匹配，那么PNG图像文件可以利用CRC值以允许程序避免处理一个图像。尽管这对于安全性和功能性来说是一个非常重要的特性，但是如果当协议变异时CRC值没有被正确的更新，那么就会阻碍模糊测试的进行。作为一个更加极端的例子，可以考虑分布式网络协议（Distributed Network Protocol, DNP3）[⊜]规范，该规范在监督控制和数据获取（Supervisory Control and Data Acquisition, SCADA）通信中被利用。数据流被单个的划分为250字节的块，并且每个块的前面增加了一个CRC—16的校验和。最后，考虑一下客户端和服务器的IP地址，它们中的一者或两者经常是处于所传输的数据中，并且在模糊测试的过程中，这两者的地址可能会经常发生变化。如果一个模糊测试框架能够提供这样一个方法，即在所生成的模糊测试用例中自动化的确定并包含这些值，那么将是非常方便的。

即使不是所有的，那么也是大多数的框架都提供了生成伪随机数的方法。一个好的框架将通过包含一个启发式攻击的完整列表来提供更强大的功能。一个启发式攻击实际上就是一个被存储的数据序列，并且已知该数据序列将会导致一个软件错误。格式化字符串（%n%n%n%n）和目录遍历序列是简单的启发式攻击的常用例子。在重新回到随机测试数据生成之前，遍历包含这样的测试用例的一个有限列表，将会在许多情形下节省时间，同时它还是一个值得进一步研

⊖ http://en.wikipedia.org/wiki/Asn.1

⊜ http://en.wikipedia.org/wiki/Cyclic_redundancy_check

⊜ http://www.dnp.org/

究的课题。

错误检测在模糊测试中发挥着重要的作用，我们将在第24章“智能故障检测”中对它进行深入的探讨。一个模糊器可以发现如果其测试目标不能接收一个新的连接的话，那么它可能会在一个非常简单的层次上执行失败。更加高级的错误检测通常是借助于一个调试器来完成的。一个高级的模糊测试框架应当允许模糊器与关联到目标应用的一个调试器直接进行通信，或是在其框架内绑定一个定制的调试器技术。

依赖于你个人的喜好，这里可能会有一个很长的包含各种细微特性的清单，可以显著提高你的模糊器开发经验。例如，有一些框架包含对解析范围很广的一类格式化数据的支持。例如，当将原始字节复制并粘贴到一个模糊测试脚本时，能够以如下格式0x41 0x42,\x41\x42,4142等中的任意一种来粘贴十六进制字节将会是非常方便的。

到目前为止，人们对模糊测试的度量（查看第23章“模糊器跟踪”）的关注也相对较少。一个高级的模糊测试框架应当包含一个接口，以同一个度量收集工具如代码覆盖监视器进行通信。

最后，理想的模糊测试框架将提供便利条件以最大程度的实现代码重用，方法是通过使已开发的构件能够很容易地用于后续项目。如果被正确地加以实现，那么这个概念将使得一个模糊器变得更加智能化。在深入讨论特定于任务的以及面向通用目的这两种框架的设计和创建过程之前，当我们对一些模糊测试框架进行研究时，要牢记所讨论的这些概念。

21.2 现有框架

在本节中，我们将通过剖析一些模糊测试框架来了解目前已经存在的一些情况。我们并不会涉及所有可用的模糊测试框架，但是相反，我们将研究一个代表了许多不同方法学的框架示例。下面所列出的框架是根据其成熟度和包含特性的丰富程度来排列的，首先介绍的是最原始的框架。

21.2.1 antiparser[⊖]

Antiparser是一个用Python语言编写的API，被设计用来帮助生成随机数据，特别是针对模糊器的创建。这个框架可以被用来开发能够跨多种平台运行的模糊器，因为该框架只依赖于一个Python解释器的可用性。该框架的使用是非常方便的。首先必须要创建antiparser类的一个实例，该类是作为一个容器使用的。接下来，antiparser将提供一些可以被实例化并追加到该容器中的模糊类型。可用的模糊类型如下所示：

- `apChar()`:一个8位的C字符。
- `apCString()`:一个C类型的字符串，即以一个空字节结尾的字符数组。
- `apKeywords()`:一个值的列表，每个值关联着一个分隔符，数据块以及终结符。
- `apLong()`:一个32位的C整数。

⊖ http://antiparser.sourceforge.net/

- apShort():一个16位的C整数。
- apString():一个自由形式的字符串。

在所有可用的数据类型中，apKeywords()是最有趣的一种。使用这个类，你可以定义关键字的一个列表，一个数据块，关键字和数据之间的一个分隔符，以及一个可选的数据块终结符。这个类将以如下的格式来生成数据：[keyword][separator][data block][terlminator]。

Antiparser发布了一个示例脚本evilftpclient.py，该脚本利用了apKeywords()数据类型。让我们通过研究该脚本的一部分来更好地理解该框架的开发过程。下面摘录的Python代码显示了evilftpclient.py脚本中的相关部分，该部分的功能是在解析FTP动词参数的过程中，对一个FTP daemon程序进行测试以发现格式化字符串漏洞。例如，该代码片段并没有显示使用目标FTP daemon程序进行验证的功能。为了得到完整的代码，可以参考相关的源代码。

```
from antiparser import *

CMDLIST = ['ABOR', 'ALLO', 'APPE', 'CDUP', 'XCUP', 'CWD',
           'XCWD', 'DELE', 'HELP', 'LIST', 'MKD',  'XMKD',
           'MACB', 'MODE', 'MTMD', 'NLST', 'NOOP', 'PASS',
           'PASV', 'PORT', 'PWD',  'XPWD', 'QUIT', 'REIN',
           'RETR', '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.setTerminator(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 = apSocket()
    sock.connect(HOST, PORT)

    # print FTP daemon banner
    print sock.recv(1024)
    # send fuzz test
    sock.sendTCP(ap.getPayload())

    # print FTP daemon response
    print sock.recv(1024)
    sock.close()
```

在上述代码的开头，首先从antiparser框架中导入了所有可用的数据类型，定义了一个列表，该列表包含FTP动词，动词参数分隔符字符（空格）以及命令终止序列（回车后面跟着一个新行）。然后，以一个新的antiparser容器类的实例作为开始，每一个所列出的FTP动词被单独循环迭代并测试。接下来，就该apKeywords()数据类型发挥作用了。有一个列表定义了一个单独的条目，当前正在被测试的动词被指定为关键词（在这个例子中是keyword）。然后，定义适当的动词——参数分隔符以及命令终止字符。所创建的apKeywords()对象的数据内容被设置为格式化字符串标记的一个序列。如果在解析动词参数的过程中，目标FTP服务器暴露了一个格式化字符串漏洞，那么该数据内容必定将触发它。

接下来的两个调用setMode ('incremental') 和setMaxSize (65536)，指定了在变异过程中，数据块应当逐渐递增到最大值65 536。然而，在这个特定的例子中，这两个调用却是不相关的，因为模糊器并不会通过调用ap.permute()来循环遍历一些测试用例或者变异。相反，每个动词使用一个单独的数据块来测试。

上述代码的剩余行都是能够自解释的。单个数据类型apKeywords()被追加到antiparser容器中，并且创建了一个socket。一旦建立起一个连接，那么就在对ap.getPayload()的调用中生成测试用例，并且通过sock.sendTCP()来传输测试用例。

显然，antiparser也具有一些局限性。可以使用原始Python来很容易地重新生成示例FTP模糊器，而不需要框架提供帮助。当在antiparser框架下开发模糊器时，特定于框架的代码与通用代码之间的比率与其他的框架相比是非常低的。该框架也缺乏在上一节所列出、所需要的许多自动化功能，比如自动化的计算并表示通用TLV协议格式的能力。最后一点局限性是，该框架的文档的参考价值不高，并且不幸的是，在2005年8月所发布的该框架2.0版本中，只有一个单独的例子是可用的。尽管该框架非常简单，并且为创建简单模糊器提供了一些帮助，但是它不足以处理更加复杂的问题。

21.2.2 Dfuz[⊖]

Dfuz是Diego Bauche用C语言编写的，它是一个被动态维护并且经常被更新的模糊测试框架。该框架用来发现影响提供商如微软、Ipswitch和RealNetwork的一些不同的漏洞。该框架的源代码是开放的并且可以下载，但是一个限制性的许可将禁止任何复制或者修改操作，除非经过作者的特殊许可。取决于你的需要，这个受限制的许可可能会阻止你利用该框架。这个严格的许可背后的激发因素可能是作者对他自己的代码的质量感到不满意（根据README文件）。如果你想要重用他的代码的某些部分，那么就应当直接与他联系。Dfuz被设计为运行在UNIX/Linux平台上，并且提供了一个定制语言以开发新的模糊器。这个框架并不是本章所讨论的最高级的模糊测试框架。然而，其设计思想的简单性和直观性使其成为了学习框架设计的一个很好的实例，因此下面将对它进行更加深入的研究。

构成Dfuz的基本组成部分包括数据、函数、列表、选项、协议以及变量。这些不同的组成

⊖ http://www.genexx.org/dfuz/

部分被用来定义了一组规则集，模糊测试引擎可以对这些规则集进行解析以生成和传输数据。可以使用下面熟悉而又简单的语法来在规则文件中定义变量：

```
var my_variable = my_data
var ref_other   = "1234",$my_variable,0x00
```

变量通过简单的前缀*var*来定义，并且可以通过在变量前面加上一个美元符号（即$，同Perl和PHP中一样）的前缀来从其他位置进行引用。模糊器的整个创建过程是自包含的。这意味着与antiparser不同，例如，在此框架上创建一个模糊器完全是使用其自己定制的脚本语言。

Dfuz定义了不同的函数来实现经常需要的一些功能。这些函数非常易于识别，因为它们的名字前面具有百分号字符（%）的前缀。所定义的函数如下所示：

- %attach()：在按下Enter或者Return键之前一直处于等待状态。当想要将模糊器暂停以完成其他任务时，该函数就是有用的。例如，如果你的模糊测试目标生成了一个新的线程以处理引入的连接，并且你希望将一个调试器关联到这个新线程，那么就在初始连接被建立之后插入一个对%attach()的调用，然后定位并关联到目标线程。
- %length()或%calc()：计算所提供参数的大小，并且将其以二进制格式插入二进制流中。例如，%length（"AAAA"）将会把二进制值0x04插入到二进制流中。这些函数的默认输出大小是32位，但是可以通过调用%length:uint8()将大小改为8位，也可以通过调用%length:uint16()将大小改为16位。
- %put:<size>（number）：将所指定的数字插入到二进制流中。该大小可以被指定为一个字节，一个字或者是两个字，它们所对应的别名分别是uint8,uint16和uint32。
- %random:<size>()：将针对指定大小，生成并插入一个随机二进制值。同%put()相类似，该大小也可以被指定为一个字节，一个字或者是两个字，它们所对应的别名分别是uint8,uint16和uint32。
- %random:data（<length>,<type>）：生成并插入随机数据。长度指明了将要生成的字节数量。类型指定了将要生成的随机数据的种类，它可以被指定为一个ASCII码，一个字母和数字的组合体，一个无格式数以及图形。
- %dec2str（num）：将一个十进制数转换为字符串，并且将其插入到二进制流中。例如，%dec2str（123）将生成123。
- %fry()：随机的修改前面所定义的数据。例如，规则"AAAA"，%fry()将会导致字符串"AAAA"中的字符的一个随机数被一个随机的字节值所替代。
- %str2bin()：对不同的十六进制字符串表示进行解析，将其转换为原始值。例如，4141，41 41和41－41都将会转换为AA。

可以采用许多不同的方法来表示数据。定制的脚本语言支持指定字符串、原始字节、地址、数据重复以及基本数据循环的语法。通过使用逗号分隔符，可以将多个数据定义串接起来以形成一个简单的列表。下面的例子说明了可以采用不同的方法来定义数据（可以参阅相关的文档，以获得更加详细的信息）：

```
var my_variable1 = "a string"
var my_variable2 = 0x41,|0xdeadbeef|,[Px50],[\x41*200],100
```

列表的声明格式如下：首先是关键字list，其后面是列表的名字，然后是关键字begin，再后面是以新行来分隔的数据值的列表，最后是以关键字end作为终结符。可以使用一个列表来定义并索引一个数据序列。例如下面的代码：

```
list my_list:
begin
    some_data
    more_data
    even_more_data
end
```

类似于变量，也可以通过在列表的前面加上一个美元符号（$）前缀来从其他位置引用该列表。同其他脚本语言如Perl和PHP的语法相类似，一个列表的元素可以通过方括号`$my_list[1]`来进行索引。通过使用关键字*rand*，用如下的形式`$my_list[rand]`就可以实现列表中的随机索引。

有一些选项可以用来控制整个引擎的行为，这些选项如下所示：

- keep_connecting:继续执行模糊测试，即使是在一个连接不能够建立的情况下。
- big_endian:将数据生成的字节顺序改变为将高位字节先存放在低地址处的顺序（默认是将低位字节先存放在低地址处的顺序）。
- little_endian：将数据生成的字节顺序改变为将低位字节先存放在低地址处的顺序（默认情况）。
- tcp:指定应当通过TCP来建立socket连接。
- udp:指定应当通过UDP来建立socket连接。
- client_side:指定引擎将作为一个客户端来对服务器进行模糊测试。
- server_side:指定引擎将作为等待连接的一个服务器来对客户端进行模糊测试。
- use_stdout:将数据生成到标准的输出（控制台），而不是一个socket连接的输出点。该选项必须被关联上一个主机值“stdout”。

为了将重新生成经常被模糊测试的协议的工作进行简化，Dfuz可以模拟FTP，POP3，Telnet以及服务器消息块（SMB）等协议。该功能的实现通过提供以下函数来完成：`ftp:user()`,`ftp:pass()`,`ftp:mkd()`,`pop3:user()`,`pop3:pass()`,`pop3:dele()`,`telnet:user()`,`telnet:pass()`以及`smb:setup()`等等。可以参阅Dfuz的相关文档以得到一个完整的函数列表。

这些基本的组成部分必须要同某些附加指令结合在一起才能创建规则文件。作为一个简单而又完整的例子，考虑如下的对一个FTP服务器进行模糊测试的规则文件（同该框架绑定在一起）：

```
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
```

第一条指令指明了该引擎必须要通过TCP端口21来进行连接。由于没有选项被指定，因此在默认情况下它是作为一个客户端来进行操作的。peer read和peer write指令分别为引擎指示了应当何时从模糊测试目标中读取数据，以及何时将数据写入模糊测试目标中。在这个特定的规则文件中，使用了FTP协议功能来对目标FTP服务器进行验证。接下来，改变工作区目录（Change Working Directory,CWD）命令被手工创建并传输给服务器。CWD命令的内容是1 024字节的字母与数字相组合的数据，其后面跟着一个表示终结的新行（0x0a）。最后连接被关闭。最后的repeat指令指明了peer read和peer write代码块应当被执行1 024次。对于每一个测试用例，Dfuz将建立起一个同FTP服务器的经过验证的连接，发出一个以随机的1 024字节的字母数字的混合字符串为参数的CWD命令，然后销毁该连接。

Dfuz是一个简单而又功能强大的模糊测试框架，可以使用它对许多协议和文件格式进行复制和模糊测试。在某些基本的命令行脚本支持下，stdout（标准输出）的组合就可以将该框架转化为一个针对文件格式、环境变量以及命令行参数的模糊器。Dfuz具有一个相对较快的学习曲线，并且以较快的开发速度来节省时间。完全依靠自己的脚本语言来实现模糊器的开发这样一个事实是一把双刃剑。其有利的方面是，没有经验的程序员可以在该框架上对协议进行描述并实施模糊测试，而其负面影响是，有经验的程序员不能够利用成熟的编程语言所提供的内在功能和特性。Dfuz在某种程度上促进了代码的重用，但是不如其他框架如Peach那样全面。目前该框架所欠缺的一个关键特性是一个智能化启发式攻击集的可用性。总之，对于具有良好设计的模糊测试框架来说，Dfuz是一个有趣的研究实例，并且可以作为一个好的工具供日常使用。

21.2.3 SPIKE[⊖]

Dave Aitel所编写的SPIKE可能是使用最为广泛并且最被人所熟知的模糊测试框架。SPIKE是采用C语言来编写的，它提供了一个API以用于快速有效地开发网络协议模糊器。SPIKE是一个开源的工具，并且在具有灵活性的通用公共许可（GPL）的控制下发布[⊜]。这个备受欢迎的许可允许创建该框架的另外一个版本SPIKEfile，它是特别被设计用于对文件格式进行模糊测试的（可以参见第12章“文件格式模糊测试：UNIX平台上的自动化测试”）。SPIKE利用了一种新技术以对网络协议进行表示，并且对其实施模糊测试。协议数据结构被分割并被表示为块，其中包含二进制数据和该块的大小。基于块的协议表示就允许抽象地创建不同的协议层，并且

⊖ http://www.immunitysec.com/resources-freesoftware.shtml

⊜ http://www.gnu.org/copyleft/gpl.html

可以自动计算大小。为了更好的理解基于块的概念，考虑下面这个简单的例子，该例子来自于白皮书《基于块的协议分析在安全测试中的优越性》(The Advantages of Block-Based Protocol Analysis for Security Testing) ⊖。

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata")
s_binary("01020304");
s_block_end("somepacketdata");
```

这个基本的SPIKE脚本（SPIKE脚本是采用C语言编写的）定义了一个名为somepacketdata的块，将4个字节值0x01020304放入该块中，并且将块的长度作为前缀加到块的前面。在这个例子中，块的长度将被计算为4，并以将高位字节先存放在低地址处的顺序而被保存为一个字。注意大多数的SPIKE API都具有前缀s_或者spike_。s_binary() API被用来将二进制数据添加到一个块中，其参数的格式是非常自由的，以允许该API来处理范围很广的不同的被复制以及粘贴的输入，例如字符串4141\x41 0x41 41 00 41 00。尽管这个例子非常简单，但是它却说明了构建SPIKE的基本的以及全部的方法。由于SPIKE允许将块嵌入到其他块中，因此任意复杂的协议能够被非常简单地分解为最小的原子组成。下面将前面的例子进行扩展：

```
s_block_size_binary_bigendian_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somepacketdata和innerdata两个块。第二个块包含在第一个块内部，并且每一个块都以一个大小值作为前缀。新定义的innerdata块以一个静态的两字节值0x0001作为开始，后面跟着一个默认值为0x02的4字节变量整数，最后是一个默认值为SELECT的字符串变量。s_binary_bigendian_word_vari-able()和s_string_variable() API将分别循环遍历一个预定义的整数和字符串变量集（启发式攻击），其目的我们在前面已经知道是为了发现安全漏洞。SPIKE开始执行时，将首先循环遍历可能的字变量变异，然后就开始对字符串变量实施变异。该框架的真正威力在于SPIKE将会自动化的更新每个大小字段的值，就好像实施了不同的变异操作一样。为了研究并扩展当前的模糊变量列表，可以查看SPIKE/src/spike.c。该框架的2.9版包含一个大约有700个可诱发错误的启发式攻击列表。

使用上一个例子中所描述的基本概念，你就可以看到在这个框架中，如何对任意复杂的协议进行建模。这里还有一些额外的API和例子，可以查阅SPIKE的相关文档以获取更多的信息。让我们仍然来关注运行中的例子，下面的代码片段引自于同SPIKE一起发布的一个FTP模糊器。因为没有块被实际定义，因此这并不是对SPIKE功能最好的展现，但是它可以有助于进行同类

⊖ http://www.immunitysec.com/downloads/advantages_of_block_based_analysis.pdf

工具之间的比较。

```
s_string("HOST ");
s_string_variable("10.20.30.40");
s_string("\r\n");

s_string_variable("USER");
s_string(" v);
s_string_variable("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("\r\n");
```

SPIKE被部分文档化，并且它所发布的包都包含许多可能会导致混淆的构件。然而，目前有许多可用的例子，这些例子对于熟悉这个功能强大的模糊测试框架来说是极好的参考。由于缺乏完整的文档以及对所发布的包的破坏，因此使得一些研究者推断SPIKE是有意识地在一些领域阻止其他人发现漏洞，而这些漏洞是由该工具的开发者私密地发现的。这种说法的真实性仍然没有得到证实。

取决于你个人的需要，SPIKE框架的一个主要缺陷是缺乏对微软Winodws的支持，因为SPIKE被设计为运行在UNIX环境中，尽管目前有一些混合的报道称可以通过使用Cygwin[⊖]来

⊖ http://www.cygwin.com/

使SPIKE运行于Windows平台上。另外考虑到的一个不足因素是即使对该框架做一个非常小的改变，例如增加新的模糊字符串，那么也需要重新进行编译。最后一个不足的方面是需要通过手工的复制——粘贴操作来实现所开发的模糊器之间的代码重用。新元素例如针对邮件地址的一个模糊器不能被简单地定义，也不能稍后在框架内被全局引用。

总之，SPIKE已经被证明是有效的，并且已经被其作者和其他人所使用而发现了多种具有鲜明特征的漏洞。SPIKE也包括一些诸如代理这样的工具，它允许一个研究者对Web浏览器和Web应用程序之间的通信进行监视，并实施模糊测试。SPIKE诱发错误的功能经历了一个很长的阶段才体现出了模糊测试的全部价值。自从SPIKE最初被公开发布以来，针对模糊测试的基于块的方法就获得了流行，目前许多的模糊测试框架都已经采用了该技术。

21.2.4 Peach[⊖]

由IOACTIVE所发布的Peach是一个采用Python语言编写的跨平台的模糊测试框架，它最初是于2004年发布的。Peach是一个开源框架，并且具有开放式的许可。同其他可用的模糊测试框架相比，Peach是可证明的最为灵活的一个框架，并且最大程度地促进了代码的重用。另外，以作者的观点来看，它具有一个最有趣的名字（桃子，模糊——理解了吗？）。该框架提供了一些基本的构件以创建新的模糊器，包括生成器、转换器、协议、发行器以及群组。

生成器负责生成从简单的字符串到复杂的分层的二进制消息范围内的数据。可以将生成器串接起来以简化复杂数据类型的生成。将数据生成抽象到其自己的对象中就可以很容易地在所实现的模糊器中进行代码重用。例如，考虑在一个SMTP服务器的审核过程中开发了一个邮件地址生成器。该生成器可以明显地在其他需要生成邮件地址的模糊器中被重用。

转换器以一种特定方式来改变数据。示例转换器可能包括base64编码器、gzip以及HTML编码器。转换器也可以被串接起来使用，也可以将其绑定到一个生成器。例如，一个所生成的邮件地址可以通过一个URL编码器的转换器来进行传递，然后再通过一个gzip转换器传递。将数据转换抽象到其自己的对象中就可以很容易地在所实现的模糊器中进行代码重用。一旦一个给定的转换被完成，那么它就可以明显地被所有以后所开发的模糊器来重用。

发行器通过一个协议实现了针对所生成数据的一种传输形式。示例发行器包括文件发行器和TCP发行器。同样，将此概念抽象到其自己的对象中也促进了代码的重用。尽管在当前的Peach版本中还是不可行的，但是发行器的最终目标是为任意的发行器提供透明的接口。例如，考虑你创建了一个GIF图像生成器。该生成器应当能够通过简单地产生一个特定的发行器来发布一个文件或者传递到一个Web表单。

群组包含一个或者多个生成器，它是对生成器可以产生的值进行遍历的一种机制。Peach包含一些常用的群组实现。另一个额外的构件是脚本对象，它是一个简单的抽象以减少实现下面这些循环所需要的多余代码的数量，即数据循环以及对`group.next()`和`protocolo.step()`的调用的循环。

⊖ http://peachfuzz.sourceforge.net

作为一个完整而又简单的例子，考虑下面的Peach模糊器，它被设计为从一个字典文件中强力发掘一个FTP用户的密码：

```
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()
```

在模糊器的开头，首先导入了Peach框架的多个不同的构件。接下来，实例化了一个新的块和群组。该块被定义为传输用户名以及密码命令的动词。该块的下一个元素导入了潜在密码的一个字典。我们将在模糊测试的过程中对这个块元素进行循环迭代。该块的最后一个元素终止了密码命令，并且发出了FTP退出命令以从服务器断开连接。然后，对已经可用的FTP协议进行扩展来定义了一个新协议。最后，创建了一个脚本对象以有效地组织连接的循环，并且对字典进行循环遍历。在对上述脚本进行研究之后，脑海中所浮现出的第一个想法就是与该框架进行接口并不是非常直观。这是一个正确的批评，并且该问题可能是Peach框架所存在的一个最大的缺陷。在Peach中开发你的第一个模糊器肯定要比在Autodaf é 或者Dfuz中花费更长的时间。

Peach框架允许研究者关注于一个给定协议的单独的子部分，然后再将它们结合在一起创建一个完整的模糊器。这种开发模糊器的方法，尽管可以证明其开发速度不如基于块的方法那样快，但是它肯定能够促进在其他任意模糊测试框架中的代码重用。例如，如果必须要开发一个gzip转换器以测试一个反病毒解决方案，然后它在库中变为可用的，并且稍后可以被透明地用于测试一个HTTP服务器处理压缩数据的能力。这是Peach所显示的很好的一面。你使用它越多，那么它就会变得越智能。由于其完全是采用Python语言来实现的，Peach模糊器可以在正确安装有Python的任何环境中运行。另外，通过利用现有的一些接口，如Peach的接口、微软的COM[⊖]或者微软的.NET包，Peach可以直接对ActiveX控件和可控制的代码实施模糊测试。目前也有使用Peach直接对微软的Windows DLL进行模糊测试的例子，同时也可以将Peach嵌入到C/C++代码中以生成被操纵的客户端和服务器。

Peach处于动态发展之中，到本书出版时其最新的可用版本是0.5（在2006年4月发布）。尽

⊖ http://en.wikipedia.org/wiki/Component_Object_Model

管Peach在理论上来说是非常先进的，但是不幸的是它没有被完整地文档化，并且没有被广泛的应用。由于缺乏相关的参考资料就导致了一个比较困难的学习曲线，这可能会阻碍你对该框架的应用。Peach的开发者介绍了一些新颖的思想，并且创建了一个坚实的基础以供扩展。最后需要指出的一点是，尽管在本书写作时还没有更多的详细信息被提供，但是Peach框架的一个Ruby端口已经被声明。

21.2.5 通用模糊器[⊖]

由Applied Security的Jared DeMott所发布的通用模糊器（General Purpose Fuzzer, GPF），被命名为关于普遍公认的术语——通用保护错误的一个双关语。GPF被动态地进行维护，并且在GPL许可下是可用的开源工具，被开发以运行在UNIX平台上。正如其名字所暗示的，GPF被设计为一个通用的模糊器，同SPIKE不一样，它可以生成无限数量的变异。这并不是说生成式的模糊器就优越于启发式的模糊器，因为这两种方法各具优点和局限性。本节中列出的GPF与其他框架相比所具有的主要优点是，使用GPF来建立和运行一个模糊器的初始成本比较低。GPF通过一些模式来提供相关功能，这些模式包括PureFuzz，Convert，GPF（主模式），Pattern Fuzz以及SuperGPF。

PureFuzz是一个易于使用的纯随机模糊器，类似于将/dev/urandom关联到一个socket。尽管所生成的输入空间是非智能化的和无限的，但是该技术在过去已经发现了一些漏洞，甚至是在公共的企业应用软件中也发现了漏洞。PureFuzz与netcat和/dev/erandom的组合相比，所具有的主要优点是PureFuzz提供了一个种子选项以允许重放伪随机流。另外，如果PureFuzz执行成功，那么可以通过使用范围选项来查明导致异常的特定包。

随机方法也可以很有效

许多人认为诸如GPF的PureFuzz的模糊器生成纯粹的随机数，对于实际应用来说过于简单。为了消除这个普遍的误解，考虑下面这个关于计算机协会的BrightStor ARCserve文件备份的解决方案的实际例子。2005年8月，在负责处理微软SQL Server文件备份的代理中发现了一个细微的、可利用的基于栈的缓冲区溢出漏洞[⊜]。触发该漏洞所需要进行的操作就是向受影响的、正在TCP端口6070监听的daemon程序传输超过3 168个字节的数据。

可以通过用最少的设置并且不用进行协议分析的随机模糊测试来很容易地发现该漏洞。将该例子作为一个证明，即当用手工方法正在创建一个智能模糊器时，运行一个诸如PureFuzz的模糊器是非常值得的。

Convert是一个可以将libpcap文件，比如那些被Ethereal[⊝]和Wireshark[⊞]所生成的文件，翻译成一个GPF文件的GPF实用工具。该工具通过将二进制pcap（捕获的包）格式转换为一种人

⊖ http://www.appliedsec.com/resources.html

⊜ http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=287

⊝ http://www.ethereal.com

⊞ http://www.wireshark.org

工可读的、准备修改的并且基于文本的格式，从而减少了在协议建模的初始阶段所进行的一些单调的工作。

在GPF的主模式中，提供了一个GPF文件以及多种命令行选项以控制一些不同的基本协议攻击。捕获的通信被重放，并且采用了不同的方法对其中的某些部分进行了变异。这些变异包括插入逐渐增长的字符串序列以及格式化字符串标记、字节循环以及随机变异方法。这种模糊测试模式需要进行大量的手工操作，因为其逻辑部分是构建在分析者的直觉之上的。

Pattern Fuzz（PF）是最为著名的GPF模式，因为它能够自动化的对所检测到的协议的纯文本部分进行标记和模糊测试。PF检查目标协议以发现公共的ASCII边界和字段终结符，然后根据内部规则集自动化的对它们进行模糊测试。这些规则被定义为名为tokAids的C代码。ASCII变异引擎被定义为一个tokAid（normal_ascii），这里还有其他一些引擎（如DNS）。为了准确地、智能化地对一个定制的二进制协议进行建模和模糊测试，必须要编写并编译tokAid。

SuperGPF是一个用Perl脚本语言编写的GPF包装器，它被编写以解决这样的情形，即当一个特定的socket端点被设为模糊测试的目标，但是研究者却不知道从何处入手。SuperGPF与一个捕获了包含有效协议命令的一个文本文件的GPF相结合，并且生成了上千个新的捕获文件。然后，该脚本采用不同的模糊测试模式启动多个GPF的实例，以所生成的多种不同的数据对目标进行攻击。SuperGPF被局限于只能对ASCII协议进行模糊测试。

同样，我们提供一个示例GPF脚本以同前面所讨论的FTP模糊器进行比较和对比：

```
Source:S Size:20 Data:220 (vsFTPd 1.1.3)
Source:C Size:12 Data:USER fuzzy
Source:S Size:34 Data:331 Please specify 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GPF的最大障碍在于其所具有的复杂性。掌握GPF所需要的学习曲线是非常漫长的。使用tokAids来对私有的二进制协议进行建模并不像使用诸如SPIKE的基于块的方法等方法那样简单，并且还需要进行编译才能使用。最后，由于对命令行选项的过分依赖导致了一些不实用的命令，如下所示：

```
GPF ftp.gpf client localhost 21 ? TCP 8973987234 100000 0 + 6 6 100 100 5000 43 finish
0 3 auto none -G b
```

总之，由于GPF具有灵活性和可扩展性等特性，因此它仍然是一个有价值的工具。不同的模式允许当研究者正在建立另一个更加智能的攻击时，可以立刻开始进行模糊测试。GPF所具有的自动化的对ASCII协议进行处理和模糊测试的能力是非常强大的，这一点在其他框架中也是非常突出的。本书正在编写的时候，GPF的开发者正在开发一种新的模糊测试模式，该模式利用了进化计算的基本原理对未知协议进行自动化解析和智能化模糊测试。我们将在第24章“自动化的协议解析”中对这个高级的模糊测试方法进行更加详细的讨论。

21.2.6 Autodafé[⊖]

非常简单，Autodafé可以被描述为SPIKE的换代产品，并且可以用来对网络协议和文件格式进行模糊测试。Autodafé是由Martin Vuagnoux在灵活的GNU GPL的许可下发布的，并且被设计为运行在UNIX平台上。同SPIKE相类似，Autodafé的核心模糊测试引擎也采用了基于块的方法来进行协议建模。下面摘自于白皮书《Autodafé，一个对软件进行攻击的工具》(Autodafé, an Act of Software Torture)[⊜]代码说明了Autodafé所使用的基于块的语言，SPIKE用户对这些代码可能会比较熟悉：

```
string("dummy");              /* define a constant string */
string_uni("dummy");          /* define a constant unicode string */
hex(0x0a 0a \x0a);            /* define a hexadecimal constant value */
block_begin("block");         /* define the beginning of a block */
block_end("block");           /* define the end of a block */
block_size_b32("block");      /* 32 bits big-endian size of a block */
block_size_l32("block");      /* 32 bits little-endian size of a block */
block_size_b16("block");      /* 16 bits big-endian size of a block */
block_size_l16("block");      /* 16 bits little-endian size of a block */
block_size_8("block");        /* 8 bits size of a block */
block_size_8("block");        /* 8 bits size of a block */
block_size_hex_string("a"); /* hexadecimal string size of a block
block_size_dec_string("b"); /* decimal string size of a block */
block_crc32_b("block");       /* crc32 of a block in big-endian */
block_crc32_l("block");       /* crc32 of a block in little-endian */
send("block");                /* send the block */
recv("block");                /* receive the block */
fuzz_string("dummy");         /* fuzz the string "dummy" */
fuzz_string_uni("dummy");     /* fuzz the unicode string "dummy" */
fuzz_hex(0xff ff \xff);       /* fuzz the hexadecimal value */
```

这个功能非常简单，因为它可以表示大部分的二进制以及纯文本协议格式。

Autodafé框架的主要目标是缩减整个模糊测试的输入空间，并降低其复杂性，以更加有效地关注可能会导致发现安全漏洞的那些协议的领域。对于一个完整的模糊测试审核来说，计算输入空间及其复杂性是非常简单的。考虑下面的例子：

```
fuzz_string("GET");
string(" /");
fuzz_string("index.html");
string(" HTTP/1.1");
hex(0d 0a);
```

一旦将上述脚本部署到一个目标Web服务器上，那么它将会首先循环遍历许多HTTP动词变异，其后面跟着许多的动词参数变异。假定Autodafé包含一个具有500个条目的模糊字符串替换库。完成这个审核所需要的测试用例的总数是被模糊化变量数的500倍，因此总共有1 000个测试用例。在最接近于现实的情况下，替换库至少会将其库的大小翻倍，这样就会有更多的被

⊖ http://autodafe.sourceforge.net

⊜ http://autodafe.sourceforge.net/docs/autodafe.pdf

模糊化的变量。Autodafé应用了一种非常有趣的被称为Markers的技术，来为每个模糊化变量提供一个权重。Autodafé将一个marker定义为被用户（或者模糊器）所控制的数据、字符串或者数字。所指定的权重被用于确定对模糊变量进行处理的顺序，将关注的焦点集中于那些最有可能导致发现安全漏洞的变量。

为了完成这个任务，Autodafé包含进了一个名为adbg的调试器构件。调试技术在传统意义上一直是和模糊器一起使用的，甚至被包含在特定的模糊测试工具中如FileFuzz（参见第13章“文件格式模糊测试：Windows平台上的自动化测试”），但是Autodafé是第一个显式包含一个调试器的模糊测试框架。Autodafé使用调试器构件在公共的具有危险的API中设置断点，这些API的例子包括被认为是导致缓冲区溢出来源的`strcpy()`，以及被认为是导致格式化字符串漏洞来源的`fprintf()`。模糊器同时将测试用例传递给目标应用和调试器。然后，调试器就对具有危险的API调用进行监视以寻找始自于模糊器的字符串。

每个模糊变量被认为是一个marker。然后将被检测到通过具有危险的API传递的每个单独的marker的权重进行增加。没有触及危险API的那些marker在第一次传递过程中将不被模糊化。具有较高权重的marker将得到较高的优先级，并且将首先被模糊化。如果调试器检测到了一个访问违规，那么就会自动通知模糊器并且记录可用的测试用例。通过对模糊变量划分优先级，并且忽略那些从未经过危险API的变量，就可以极大地缩减整个输入空间。

Autodafé还包含一些辅助的工具以帮助快速而有效地进行模糊器的开发。第一个工具是PDML2AD，可以将Ethereal和Wireshark所导入的包数据XML（PDML）文件解析为基于块的Autodafé语言。如果你的目标协议处于主流网络嗅探器所识别的超过750个协议的范围中，那么基于块的建模所需要进行的大部分单调的工作都可以实现自动化的处理。即使是你的目标协议没有被识别，PDML2AD仍然可以提供一些捷径，因为它能够自动检测纯文本字段并且生成适当的对`hex()`和`string()`等的调用。第二个工具是TXT2AD，它是一个简单的Shell脚本，能够将一个文本文件转换为一个Autodafé脚本。第三个也是最后一个工具是ADC，它是Autodafé的编译器。当开发复杂的Autodafé脚本时，ADC是非常有用的，因为它能够发现常见的错误如不正确的函数名以及未关闭的块等。

Autodafé是在SPIKE所奠定的基础上进行扩展得到的、一个经过深思熟虑的高级模糊测试框架。Autodafé具有同SPIKE一样的许多优点以及局限性。该框架最突出的一个特性就是其调试构件，这使得它在其他框架中显得非常突出。同样，取决于你自己的需要，缺乏对微软Windows的支持可能会限制该框架的某些应用。对该框架的修改需要经过重新编译才能生效，并且模糊脚本之间的代码重用并不像想象中的那样直接和简便。

21.3 定制模糊器的实例研究：Shockwave Flash

在上一节中我们讨论了当前可用的模糊测试框架的典型代表。为了得到更加完整的框架列表，可以参阅本书的配套Web站点http://www.fuzzing.org。我们鼓励你下载单个的框架包，并且研究其完整的文档以及各种不同的示例应用。没有一个单独的框架可以称之为最好的框架，因为某个框架只是对某些问题的处理要优于其他框架。相反，依赖于特定的目标应用、测试目

的、时间线、预算以及其他一些因素，可能会有一种框架是最为适合的。多熟悉一些框架将会为测试团队提供更大的灵活性，以将最好的工具应用于给定的任务。

尽管大多数的协议和文件格式都可以简单地在至少一种公开可用的模糊测试框架中进行描述，但是仍然存在这样的情形，即所有的框架都不可用。当处理专门软件或协议审核时，做出构建一个新的面向特定目标的模糊器的决定，将会产生更丰富的结果并且节省审核周期中的大量时间。在本节中，我们将讨论Adobe的Macromedia Shockwave Flash[⊖]（SWF）文件格式，将其作为一个现实世界中的模糊测试目标，并且提供测部分定制的测试解决方案。安全漏洞在Shockwave Flash中具有非常典型的影响，因为Flash Player的某些版本几乎被安装到全世界的每个桌面计算机上。实际上，现代的微软Windows操作系统默认都带有一个Flash Player的功能版本。解析一个成熟的SWF审核的所有细节已经远远超出了本章的讨论范围，因此这里只讨论最相关的以及我们所关注的那些方面。为了得到最新的结论以及代码列表的更多信息[⊜]，可以访问本书的官方网站http://www.fuzzing.org。

有许多因素使SWF成为了定制模糊测试解决方案的一个很好的选择。首先，SWF文件格式的结构在一个Adobe开发者参考规范中被完整地文档化，该规范名为“Macromedia Flash（SWF）和Flash Video（FLV）文件格式规范”（Macromedia Flash (SWF) and Flash Video (FLV) File Format Specification）[⊝]。本书在写作中所参考的是该规范的8.0版本。除非你正在审核一个开源的或内部所开发的文件格式，否则如此完整的文档并不是经常可用的，并且可能会限制你采用更加通用的方法来进行模糊测试。在这个特定例子中，我们将最大程度地利用Adobe和Macromedia所提供的每个细节。通过研究该规范，我们发现SWF文件格式主要被解析为一个位流，而不是大多数文件和协议被解析为的典型的字节层次的粒度。由于SWF典型地是通过Web来进行传递的，因此传输流的大小可能是决定在位的层次上进行解析的驱动力。这就增加了模糊器的创建难度，并且为创建一个定制的模糊器提供了另外一个原因，因为在本章所讨论的模糊测试框架中，没有一种能够支持位解析类型。

21.3.1 SWF文件的建模

为了成功地对SWF文件格式进行模糊测试，所需要到达的第一个里程碑就是，要求我们的模糊器能够变异并且生成测试用例。SWF包含一个头，其后面跟着一系列的标签，类似于如下所示的高层结构：

```
[Header]
  <magic>
  <version>
  <length>
  <rect>
    [nbits]
    [xmin]
```

⊖ http://www.macromedia.com/software/flash/about

⊜ 这项研究是主要是由TippingPoint的Aaron Portnoy所进行的。

⊝ http://www.adobe.com/licensing/developer/

```
      [xmax]
      [ymin]
      [ymax]
    <framerate>
    <framecount>

  [FileAttributes Tag]
  [Tag]
    <header>
    <data>
      <datatypes>
      <structs>
      ...
  [Tag]
    <header>
    <data>
      <datatypes>
      <structs>
      ...
  ...
  [ShowFrame Tag]
  [End Tag]
```

下面是对各种不同字段的具体解析：

- magic。包含三个字节，对于任何有效的SWF文件必须是“FWS”。
- version。包含一个单独的字节，表示生成文件的Flash的数字版本。
- length。是一个4字节值，指明了整个SWF文件的大小。
- rect。该字段被划分为5个组成部分：
- nbits。这部分有5个位的宽度，以位为单位指明了后续4个字段每个字段的长度。
- xmin,xmax,ymin,ymax。这些字段表示了展现Flash内容所处位置的屏幕坐标。注意这些字段并不表示像素，而是Flash的一个像素度量单元缇（twip），它表示一个逻辑像素的1/20。

从rect字段我们可以看到该文件格式的复杂性。如果nbits字段的值等于3，则该头的rect部分的总长度将是5＋3＋3＋3＋3＝17位。如果nbits字段的值等于4，则该头的rect部分的总长度将是5＋4＋4＋4＋4＝21位。使用目前可用的一种模糊测试框架来表示该结构并不是一件简单的事情。该头中的最后两个字段表示了回放帧的速度以及SWF文件中的总帧数。

该头的后面跟着一系列的标签，这些标签定义了Flash文件的内容以及行为。第一个标签FileAttributes是在Flash8.0版本中引入的一个必需的标签，该标签有两方面的作用。首先，该标签指定了SWF文件是否包含不同的属性如一个标题和描述。这个信息并不在内部使用，相反它可以被诸如搜索引擎等外部进程引用。其次，该标签指定了Flash Player是否应当给予SWF文件本地或者网络文件访问的特权。每个SWF文件都含有数量不等的其余标签，这些标签可以划分为两类：定义标签和控制标签。定义标签是定义诸如形状、按钮和声音等内容。控制标签定义了诸如位置和动作等概念。Flash Player将一直处理标签直到它遇到一个ShowFrame标签，在这时，所包含的内容呈现到屏幕中。SWF文件以强制性的End标签作为终结。

这里有许多可用的标签可以选择，每个都包含一个两字节的标签头以及后面的数据。取决于数据的长度，标签可以被定义为长型或短型值。如果数据的长度小于63位，那么标签就被认为是短型，其定义格式如下：[10位标签id][6位数据长度][数据]。如果数据块更大一些，那么标签将被认为是长型，并且在头中使用一个额外的、用以指明数据长度的4字节字段来进行定义，格式如下：[10位标签id][111111][4字节数据长度][数据]。同样，我们可以看到由于SWF格式所具有的复杂性，使得它并没有在前面所提及的任何模糊测试框架中被很好地建模。

情况从此开始变得糟糕。每个标签有它自己的字段集。其中有一些包含原始数据，而另一些则包含名为结构的SWF基本类型（struct或者structs）。每个struct包含一组被定义的字段，既可以包含原始数据也可以包含其他structs。情况可能还会变得更糟。标签和structs内部的字段群组可能被定义也可能不被定义，这取决于同一个标签或struc内部的另外一个字段的值。只做一般的解释可能会使你更加迷惑，因此让我们开始从头来创建SWF模糊器，并通过一些例子来更好地加以解释。

首先，我们使用Python定义了一个位字段类，以表示变量的任意数字字段的位长度。我们将这个位字段类进行扩展以定义字节（chars）、字（shorts）、双字（longs）以及四字（doubles），它们可以被简单地定义为位字段类的8位、16位、32位以及64位的实例。这些基本数据结构在包Sulley下是可以访问的（不要与下一节所讨论的Sulley模糊测试框架相混淆）：

```
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
        '''

        # pad the bit stream to the next byte boundary.
        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 = ""

        # convert the bit stream from a string of bits into raw bytes.
        for i in xrange(len(bit_stream) / 8):
            chunk = bit_stream[8*i:8*i+8]
            flattened += struct.pack("B", self.to_decimal(chunk))

        # if necessary, convert the endianess of the raw bytes.
        if self.endian == LITTLE_ENDIAN:
            flattened = list(flattened)
            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) Number to convert
    @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 + "B", 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)


    # class aliases
    bits   = bit_field
    char   = byte
    short  = word
    long   = dword
    double = qword
```

bit_field基类定义了如下内容：字段的宽度（width），该字段能够表示的最大数

(max_num)，该字段的值（value），该字段的位顺序（endian），指明了该字段的值是否可以被修改的一个标志（static），最后是在内部所使用的一个索引（s_index）。然后，bit_field类进一步定义了一些有用的函数：

- flatten()例程从该字段转换并返回一个以字节分界的原始字节序列。
- to_binary()例程可以将一个数字值转换为一个包含位的字符串。
- to_decimal()例程正好相反，它将一个包含位的字符串转换为一个数字值。
- randomize()例程将字段值更新为字段有效区间内的一个随机值。
- smart()例程通过一个智能边界序列来更新字段的值，代码中只显示了这些数字的一部分。

当从bit_field构造块创建一个复杂类型时，可以调用最后两个例程来对所生成的数据结构进行变异。然后通过递归调用每个单独构件的flatten()例程，就可以对数据结构进行遍历并将其写入到一个文件中。

使用这些基本类型，我们现在就可以开始定义更加简单的SWF结构，例如RECT和RGB。下面所摘录的代码显示了对这些类的定义，它们是从一个未显示出来的基类继承而来的（可以参阅http://www.fuzzing.org以得到该基类的定义）：

```
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()),
      ]
```

这段代码也显示了开发过程中被发现且达成一致意见的一个简单的捷径。为了简化模糊测试，变量宽度字段都被定义为其最大长度。为了用所依赖的字段来表示结构，从bit_field类又引出了一个新的原始类dependent_bit_field，如下所示：

```
class dependent_bit_field (sulley.numbers.bit_field):
    def __init__ (self, width, value=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):
        # if there is a dependency for flattening (including) this
        # structure, then check it.
        if self.parent:
            #                                VVV - object value
            if self.parent.fields[self.dep][1].value not in self.vals:
                # dependency not met, don't include this object.
                return ""

        return sulley.numbers.bit_field.flatten(self)
```

这个被扩展的类指定了在生成和返回数据之前要检查的一个字段索引和值。如果被dep所引用的适当的字段不包含列表vals中的一个值，那么将没有数据被返回。下面的MATRIX结构说明了这个新定义的原始类的用法：

```
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(31, parent=self, \
                          dep=0, vals=[1])),
        ("ScaleY"       , dependent_bits(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" , sulley.numbers.bits(5, value=31)),
        ("TranslateX"     , sulley.numbers.bits(31)),
        ("TranslateY"     , sulley.numbers.bits(31)),
      ]
```

从这个代码段中可以看到，MATRIX结构中的NScaleBits字段被定义为一个5位宽的字段，只有当该字段索引为0（HasScale）的位置处的值等于1时，该结构才包含一个默认值31。

ScaleX,ScaleY,skew1和skew2字段也都依赖于HasScale字段。换句话说，如果HasScale是1，那么这些字段就是有效的。否则，就不应当定义这些字段。类似的，NRotateBits字段依赖于该字段在索引位置4处（HasRotate）的值。在本书编写的时候，有多于200个的SWF结构采用这种标记被准确地定义[⊖]。

在具备了所有需要的原始类和结构之后，我们现在就可以开始定义完整的标签了。首先，要定义一个基类，所有的标签都将从该基类中引出：

```
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)

        # pad the bit stream to the next byte boundary.
        if len(bit_stream) % 8 != 0:
            bit_stream  = "0" * (8-(len(bit_stream)%8)) + bit_stream

        raw = ""

        # convert the bit stream from a string of bits into raw bytes.
        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:
            # long (record header is a 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:
            # short (record_header is a word)
            record_header = self.tag_id
            record_header <<= 6
            record_header |= raw_length
            flattened = pack('H', record_header)
            flattened += raw
```

⊖ 同样，可以访问http://www.fuzzing.org以得到代码。

```
        return flattened
```

针对基标签类的flatten()例程自动化计算数据部分的长度，并且生成正确的短型或长型头。在编写本书的时候，有多于50个的SWF标签在该框架中被准确地定义。下面是一些简单的以及具有中等复杂性的例子：

```
class PlaceObject (base):
    def __init__ (self, *args, **kwargs):
        base.__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()),
    ]
```

到目前为止，我们已经讨论了许多的内容，因此让我们来做一个回顾。为了对一个任意的SWF进行正确建模，我们首先使用了最基本的原始类bit_field。通过该类，我们引出了针对字节、字、双字以及四字的原始类型。同样是根据bit_field，我们引出了dependent_bit_field，通过该类我们又进一步引出了可依赖的字节、字、双字以及四字。这些类型连同一个新的基类，共同构成了SWF结构的基础。从结构的基类中引出了一个针对标签的基类，该基类连同结构和原始类共同构成了SWF标签。图21-1显示了这些构件之间的相互关系，有助于我们更好地理解SWF模糊器。

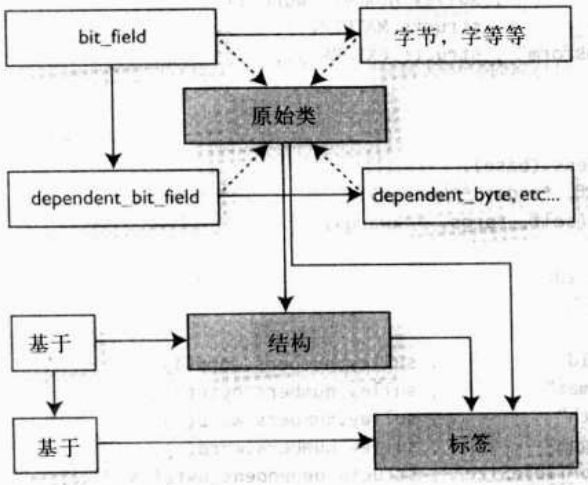


图21-1　SWF模糊器构件之间的相互关系

其他一些构造块如字符串原始类，对于完成SWF建模也是必需的。可以查阅相关资料得到它们的定义。将这些类放置于适当的位置，我们就可以创建一个完整的SWF容器，并且开始对其进行实例化以及追加标签。然后，就可以很容易地遍历所生成的数据结构，并且可以手工或使用每个单独构件的randomize()或smart()例程来自动化对其进行修改。最后，可以通过再次遍历复杂的数据结构并且将flatten()例程的运行结果进行连接，来将该结构写入到一个文件中。这种设计允许在单独的标签或结构内部对一些特殊情形进行透明的处理。

21.3.2　生成有效的数据

在我们成功地表示了基本的SWF结构之后，很快又遇到了另外一个挑战。SWF规范中声明了标签只能依赖于以前所定义的标签。当对一个带有多个依赖的标签进行模糊测试时，所有的依赖必须要首先被成功地解析，否则的话，就永远不能到达目标标签。猜测有效的字段值是一

个非常费力的过程，但是这里有一个更加智能化的方法。利用Google的SOAP API[⊖]，我们可以在Web上搜索带有关键字filetype:swf的SWF文件。已经编写了一个简单的脚本从a filetype:swf到b filetype:swf一直递增搜索到z filetype:swf。该脚本通过对返回结果进行解析以寻找并下载所发现的SWF文件。使用MD5散列算法来命名文件，以防止重复存储。

有大约10 000个唯一的SWF文件占据了超过3GB的已遍历空间。通过检查SWF头中的版本字段可以发现其分布的统计抽样结果，如表21-1所示。

表21-1 已遍历的Flash SWF版本的分布情况

SWF版本	占总数的百分比
Flash 8	<1%
Flash 7	~2%
Flash 6	~11%
Flash 5	~55%
Flash 4	~28%
Flash1-Flash3	~3%

这些结果是非常有趣的。但不幸的是，从这些统计中并不能得出有关运行的Flash Player的分布情况的结论。于是又编写了另外一个脚本来解析每个文件，将单独的标签进行抽取并存储。现在，这个有效的标签库就可以被用来帮助创建SWF测试用例了。

21.3.3 对环境进行模糊测试

PaiMei[⊜]逆向工程框架被用来简化单个测试用例的实例化，并且对单个测试用例进行监视。PaiMei是一组纯Python类的集合，被设计用来帮助自动化逆向工程工具的开发。该框架的具体使用将在第23章中进行更加详细的介绍。其实际测试过程类似于普通的PaiMei FileFuzz所使用的测试过程，如下所示：

1. 在编写有PaiMei PyDbg类的脚本的调试器下，将一个SWF测试用例加载到一个Flash Player中。
2. 启动player并监视执行5秒钟。5秒钟的延迟时间是任意选择的，这里假定如果一个错误将在SWF文件的解析过程中发生，那么它就会在这个时间区间内发生。
3. 如果在这个时间区间内检测到了一个错误，那么就将测试用例和相关的调试信息进行存储。
4. 如果在这个时间区间内没有检测到错误，那么就关闭Flash Player实例。
5. 继续执行下一个测试用例。

为了得到更加有效的测试，我们可以在SWF头中修改帧率，其范围从默认值0x000C到最大值0xFFFF。通过该操作，我们就可以允许SWF文件的更多部分在分配给每个测试用例的5秒钟时间内被处理。查看SWF文件的默认方法是将它们加载到一个Web浏览器中，如微软的Internet Explorer或者Mozilla Firefox。尽管针对这个特定的例子可以采用这种方法，但是还有另外一个方法也是可用的，那就是使用SAFlashPlayer.exe，该方法特别适用于对ActionScript进行模糊测试[⊝]。这个小的单机版播放器是同Macromedia Studio一起发布的[⊞]。

⊖ http://www.google.com/apis/index.html
⊜ http://openrce.org/downloads/details/208/PaiMei
⊝ http://en.wikipedia.org/wiki/ActionScript
⊞ http://www.adobe.com/products/studio/

21.3.4 测试方法

对SWF进行模糊测试的最普通的方法就是使用位翻转。我们在前面的章节中已经看到了这个基本的技术在字节层次上的应用。针对SWF在更细粒度的位层次上应用此技术要非常谨慎，因为一个单一的字节可能会跨越多于一个的字段。除了使用位翻转方法之外，可以使用一个定制编写的SWF标签解析脚本来改变一个单独SWF文件内部的标签顺序，并且在两个不同的SWF文件之间交换标签。最后，对SWF进行模糊测试的最彻底的方法是对每个结构和标签内部的每个字段进行单独的压力测试。可以引用前面所生成的标签和结构库来剖析一个基本的SWF以进行变异。

使用这三种方法，所生成的文件被同样地加载到了模糊测试环境中。在下一节中，我们将研究一种新的模糊测试框架，该框架是本书作者连同本书一起引入并发布的。

21.4 模糊测试框架Sulley

Sulley是一个模糊器开发和模糊测试框架，它由多个可扩展的构件组成。Sulley（以作者的观点来看）所具有的能力超过了以前所发布的大多数模糊测试技术，包括商业工具和那些在公开领域中可用的工具。该框架的目标是不仅要简化数据的表示，而且还要简化数据的传输以及对目标的监视。Sulley从Monsters,Inc创建之后就被亲切地命名[⊖]，因为它是模糊化的工具。你可以从站点http://www.fuzzing.org/sulley处下载Sulley的最新版本。

现在的大多数模糊器只是关注于数据的生成。而Sulley不仅关注于数据的生成，而且将此功能更进了一步，即包含了一个现代的模糊器应当提供的许多重要功能。Sulley观察着网络通信并系统地维护着相关的记录。Sulley监视着目标应用的状态，并且能够使用多种方法将其恢复到一个好的状态。Sulley还可以将所发现的错误进行检测、跟踪和分类。Sulley还可以并行进行模糊测试，这就极大的提高了测试速度。Sulley能够自动确定是哪个唯一的测试用例序列触发了错误。Sulley不需要人工干预就可以自动化完成上述工作以及更多的工作。Sulley的所有用法如下所示：

1. *数据表示*：这是使用任何模糊器所需要执行的第一个步骤。运行目标应用，并且当拦截包时处理某些接口。将协议分解为单独的请求，并使用Sulley将它们表示为块。

2. *会话*：将所开发的请求进行连接以形成一个会话，并将其关联到不同的可用Sulley监视代理（socket，调试器等），然后开始执行模糊测试。

3. *事后工作*：对生成的数据以及所监视的结果进行评审。重放单独的测试用例。

一旦你从站点http://www.fuzzing.org/sulley处下载了Sulley的最新包，那么将它解压到你所选择的一个目录中。由于该目录的结构是相对比较复杂的，因此让我们来看一下其具体的组织构成情况。

21.4.1 Sulley目录结构

在Sulley的目录结构中存在着一些技巧。对目录结构进行维护将可以确保当你使用用户定

⊖ http://www.pixar.com/featurefilms/inc/chars_pop1.html

义的复杂类型、请求以及工具对模糊器进行扩展时，目录中的所有内容仍然是有组织的。下面的层次关系指出了你需要了解的有关目录结构的一些内容：

- archived_fuzzies：这是一个自由格式的目录，由模糊测试目标应用的名字进行组织，以存储已存档的模糊器以及从模糊测试会话中所生成的数据。
  - trend_server_protect_5168：包含将在本文档后面的单步遍历中被引用的已执行完毕的模糊测试。
  - trillian_jabber：包含文档将引用的另外一个已执行完毕的模糊测试。
- audits：针对活动的模糊测试会话所记录的PCAP、崩溃的二进制文件、代码覆盖以及分析图等应当被保存到这个目录中。一旦模糊测试执行完毕，应当将所记录的数据转移到archived_fuzzies目录中。
- docs：该目录包含相关文档以及所生成的Epydoc API引用。
- requests：该目录是Sulley请求库。每个目标应用都应当获得其自己的文件，该文件可以被用来存储多个请求。
  - _REQUESTS_.html：这个文件包含针对所存储的请求分类的描述，并且列出了单个的类型。它是按照字母顺序来进行维护的。
  - http.py：表示不同的Web服务器模糊测试请求。
  - trend.py：包含与本文档后面所讨论的完整的模糊测试走查相关联的请求。
- sulley：该目录是模糊器的框架。除非你想要扩展该框架，否则不要触及这些文件。
  - legos：包含用户所定义的复杂原始类型。
    - ber.py：包含ASN.1/BER原始类型。
    - dcerpc.py：包含微软的RPC NDR原始类型。
    - misc.py：包含多种不同的未分类的复杂原始类型，如邮件地址和主机名等。
    - xdr.py：包含XDR类型。
  - pgraph：该目录包含Python图形抽象库。在构建会话时使用。
  - utils：包含不同的帮助例程。
    - dcerpc.py：微软的RPC帮助例程，例如实现绑定到一个接口或者生成一个请求。
    - misc.py：包含不同的未分类例程，比如CRC-16以及UUID操纵例程。
    - scada.py：包含一个DNP3块编码器特定于SCADA的帮助例程。
  - _init_.py：这里定义了在创建请求时所使用的不同的s_别名。
  - blocks.py：这里定义了块以及块的帮助例程。
  - pedrpc.py：这个文件定义了Sulley所使用的在不同的代理与主模糊器之间进行通信的客户端和服务器类。
  - primitives.py：该文件定义了不同的模糊器原始类型，包括静态类型、随机类型、字符串类型以及整数类型等。
  - sessions.py：这个文件包含构建以及执行一个会话所需要的功能。
  - sex.py：包含Sulley的定制异常处理类。

- unit_tests：该目录包含Sulley的单元测试套件。
- utils：包含多种不同的单机版工具。
  - crashbin_explorer.py：包含用于研究存储在序列化的崩溃bin文件中的结果的命令行工具。
  - pcap_cleaner.py：包含用于清除一个PCAP目录中与错误不相关的所有条目的命令行工具。
  - network_monitor.py：包含PedRPC驱动的网络监视代理。
  - process_monitor.py：包含PedRPC驱动的、基于调试器的目标监视代理。
  - unit_test.py：Sulley的单元测试套件。
  - vmcontrol.py：包含PedRPC驱动的VMWare控制代理。

经过上面的分析，我们现在对该目录结构有了进一步的了解。下面让我们来看一下Sulley是如何来进行数据表示的。这是构建模糊器所需要进行的第一个步骤。

21.4.2 数据表示

Aitel在SPIKE中已经讨论了该问题：我们对可以得到的每个模糊器都进行了分析，基于块的协议表示方法要优于其他方法，它集简便性和灵活性于一身，可以表示大多数的协议。Sulley也利用了基于块的方法来生成单独的请求，这些请求稍后将组合在一起来构成一个会话。一开始，使用一个新的名字来初始化你的请求：

```
s_initialize("new request")
```

现在你可以向请求中添加原始类型、块以及嵌套的块了。每个原始类型都可以被单独显示和变异。显示一个原始类型将导致以原始数据格式来返回其内容。对原始类型进行变异将导致转换其内部内容。显示和变异的概念在很大程度上是从模糊器开发者的实践中抽象得来的，因此不必担心其正确性。然而，当可模糊化的值用完时，每个可变异的原始类型将接收一个默认的恢复值。

静态和随机的原始类型

让我们首先从最简单的原始类型s_static()开始讨论，它将向请求中添加一个静态的、任意长度的未变异的值。为了便于使用，Sulley为它提供了许多不同的别名，例如s_dunno()，s_raw()以及s_unknown()都是s_static()的别名。

```
# these are all equivalent:
s_static("pedram\x00was\x01here\x02")
s_raw("pedram\x00was\x01here\x02")
s_dunno("pedram\x00was\x01here\x02")
s_unknown("pedram\x00was\x01here\x02")
```

原始类型和块等都包含一个可选的名字关键字参数。指定一个名字就允许你在请求中通过request.names["name"]来直接访问被命名的条目，而不用遍历块的结构以到达所需要的元素。与前面的类型有些关联但并不等同的是s_binary()原始类型，它可以接收以多种格式来表示的二进制数据。SPIKE的用户将会认识这个API，因为其功能与以前所熟知的功能是相同的（应当与那些功能相同）：

```
# yeah, it can handle all these formats.
s_binary("0xde 0xad be ef \xca fe 00 01 02 0xba0xdd f0 0d")
```

由于大多数的Sulley的原始类型都是由模糊测试的启发式规则来驱动的，因此它只包含有限数量的变异。而其中的一个例外情形是s_random()原始类型，它可以被用来生成不同长度的随机数据。这个原始类型包含两个强制使用的参数min_length和max_length，它们分别指明了在每次循环迭代过程中所生成的随机数据的最大长度和最小长度。该原始类型同时也接受下面所列出的可选关键字参数：

- num_mutations（整数值，默认值是25）：恢复到默认值之前要进行变异的数量。
- fuzzable（布尔值，默认值是True）：将对此原始类型的模糊测试设置为使能或者不使能。
- name（字符串值，默认值是None）：在所有Sulley对象中，指定一个名字就可以在整个请求中直接访问该原始类型。

num_mutations关键字参数指定了在该原始类型被认为用完之前，应当被重新显示多少次。为了用随机数据填充一个静态大小的字段，可以将min_length和max_length的值设置为同一个值。

整数类型

二进制协议的ASCII协议同样都包含有许多不同大小的整数值，例如HTTP中的内容长度字段。同大多数模糊测试框架一样，Sulley中的一部分负责表示以下这些类型：

- 1个字节：s_byte(),s_char()
- 2个字节：s_word(),s_short()
- 4个字节：s_dword(),s_long(),s_int()
- 8个字节：s_qword(),s_double()

每个整数类型至少接受一个单一的参数，即默认的整数值。另外，下面可选的关键字参数也可以被指定：

- endian（字符型，默认值为<）：位字段的存放顺序。将低位字节先存放在低地址处的顺序指定为<，将高位字节先存放在低地址处的顺序指定为>。
- format（字符串型，默认值是binary）：将输出格式指定为“binary”或者“ascii”，用以控制整数原始类型的显示格式。例如，值100以ASCII格式显示为“100”，而以二进制格式则显示为“\x64”。
- signed（布尔型，默认值是False）：将大小设置为有符号的或者无符号的，只有当输出格式设定为“ascii”时才可应用该类型。
- full_range（布尔型，默认值是False）：如果其值为真，那么该原始类型将变异所有可能的值（稍后将对该类型进行更加详细的介绍）。
- fuzzable（布尔型，默认值是True）：将对该原始类型的模糊测试设置为使能或者不使能。
- name（字符串型，默认值是None）：在所有的Sulley对象中，指定一个名字就可以在整个请求中直接访问该原始类型。

full_range修饰符在上述这些类型中是特别有趣的一个。考虑你想要对一个DWORD值进行模糊测试，总共有4 294 967 295个可能的值。以每秒钟执行10个测试用例的速度来计算，它将会耗费13年的时间完成对这个单一的原始类型的模糊测试。为了减少巨大的输入空间，Sulley

在默认情况下只尝试那些“智能”值。这包括对0周围的10个边界测试用例加上或减去适当的值，最大的整数值（MAX_VAL），MAX_VAL/2，MAX_VAL/3，MAX_VAL/4，MAX_VAL/8，MAX_VAL/16，MAX_VAL/32。缩减后的输入空间只包含141个测试用例，而执行这些测试用例只需要几秒钟的时间。

字符串和分隔符

随处都可以发现字符串。进行模糊测试时，你必定会遇到许多的字符串构件，例如邮件地址、主机名、用户名、密码等都是字符串的例子。Sulley提供了原始类型s_string()来表示这些字段。该原始类型含有一个单一的强制性的参数，这个参数为该类型指明了默认的有效值。也可以指定下面所列出的额外的关键字参数：

- size（整数值，默认值是－1）：该字符串的静态大小。为了指定动态大小，将该值设置为－1。
- padding（字符型，默认值是\x00）：如果一个显式大小被指定，并且所生成的字符串小于该大小，那么就使用该值将字段变为等于大小值。
- encoding（字符串型，默认值是ascii）：编码以为字符串所用。有效选项包括Python str.encode()例程可以接受的任何内容。对于微软的Unicode字符串而言，将其指定为“utf_16_le”。
- fuzzable（布尔值，默认值是True）：将对该原始类型的模糊测试设置为使能或者不使能。
- name（字符串型，默认值是None）：在所有的Sulley对象中，指定一个名字就可以在整个请求中直接访问该原始类型。

通过使用分隔符，字符串经常被解析为子字段。例如，空格字符在HTTP请求GET/index.html HTTP/1.0中就被用做一个分隔符。在这个请求当中，前面的斜线（/）和点（.）字符同样也是分隔符。当在Sulley中定义一个协议时，要确保使用s_delim()原始类型来表示分隔符。同其他原始类型一样，第一个参数是强制使用的，并且被用来指定默认值。同样，与其他原始类型一样，s_delim()也接受可选的fuzzable和name关键字参数。针对分隔符的变异包括重复、替换和删除。作为一个完整的例子，考虑下面针对HTML主体标签进行模糊测试的原始类型序列：

```
# fuzzes the string: <BODY bgcolor="black">
s_delim("<")
s_string("BODY")
s_delim(" ")
s_string("bgcolor")
s_delim("=")
s_delim("\"")
s_string("black")
s_delim("\"")
s_delim(">")
```

块

在掌握了原始类型的相关知识后，下面让我们来看一下如何将它们进行组织，并且在块的

内部进行嵌套。通过使用s_block_start()来定义并打开一个新块，使用s_block_end()来关闭该块。每个块必须要给定一个名字，并将其作为s_block_start()的第一个参数。该例程同时也接受下面列出的可选关键字参数：

- group（字符串型，默认值是None）：该块所关联的群组的名字（稍后将对此进行更加详细的介绍）。
- encoder（函数指针，默认值是None）：指向一个函数的指针，该函数在返回所显示的数据之前显示该数据。
- dep（字符串型，默认值是None）：可选的原始类型，该块依赖于该类型的特定值。
- dep_value（混合类型，默认值是None）：为了能够显示块，dep字段必须要包含的值。
- dep_values（混合类型的列表，默认值是[]）：为了能够显示块，dep字段可能将包含的值。
- dep_compare（字符串型，默认值是==）：应用于依赖的比较方法。有效的选项包括==,!=,>,>=,<以及<=。

分组、编码和依赖是在其他大多数框架中所没有的功能强大的特性，因此值得对它们进行深入研究。

群组

分组允许你将一个块关联到一个群组原始类型，以指定该块应当为群组内部的每个值来循环遍历所有可能的变异。例如，群组原始类型对于表示带有相似参数结构的有效操作码或动词的一个列表是非常有用的。原始类型s_group()定义了一个群组，并且接受两个强制使用的参数。第一个参数指定了群组的名字，第二个参数指定了将要遍历的、可能的原始值的列表。作为一个简单的例子，考虑下面这个完整的Sulley请求，该请求被设计为对一个Web服务器进行模糊测试：

```
# import all of Sulley's functionality.
from sulley import *

# this request is for fuzzing: {GET,HEAD,POST,TRACE} /index.html HTTP/1.1

# define a new block named "HTTP BASIC".
s_initialize("HTTP BASIC")

# define a group primitive listing the various HTTP verbs we wish to fuzz.
s_group("verbs", values=["GET", "HEAD", "POST", "TRACE"])

# define a new block named "body" and associate with the above group.
if s_block_start("body", group="verbs"):
    # break the remainder of the HTTP request into individual primitives.
    s_delim(" ")
    s_delim("/")
    s_string("index.html")
    s_delim(" ")
    s_string("HTTP")
    s_delim("/")
    s_string("1")
    s_delim(".")
```

```
    s_string("1")
    # end the request with the mandatory static sequence.
    s_static("\r\n\r\n")
# close the open block, the name argument is optional here.
s_block_end("body")
```

在该脚本的开始，首先导入了Sulley的所有构件。接下来，初始化了一个新的请求，并且将其命名为HTTP BASIC。在后面可以通过引用这个名字来直接访问该请求。然后定义了一个群组，其名字为verbs，所包含的可能的字符串值为GET,HEAD,POST以及TRACE。接着启动了一个新块，其名字为body，并且通过可选的group关键字参数将其关联到前面所定义的群组原始类型。注意s_block_start()将一直返回True，这就允许你可以使用一个简单的if语句来识别出其所包含的原始类型。同时注意到s_block_end()的name参数也是可选的。这些框架的设计决定纯粹是出于美观的目的而做出的。接着，在body块的内部定义了一系列基本的分隔符和字符串原始类型，然后将该块关闭。当将这个被定义的请求加载到一个Sulley会话中时，模糊器将为该块生成并传递所有可能的值，每次为群组中定义的每个动词生成并传递值。

编码器

编码器是一个简单而又功能强大的块修饰符。可以指定一个函数并将其关联到一个块，以在通过网络返回和传输之前，来修改所显示的该块的内容。使用一个现实世界中的例子可以更好地对此进行解释。来自于Trend微控制管理器的DcsProcessor.exe daemon程序在TCP端口20901进行监听，并且期望接收被私有的XOR编码例程所格式化的数据。通过对解码器实施逆向工程，开发了如下的XOR编码例程：

```
def trend_xor_encode (str):
    key = 0xA8534344
    ret = ""

    # pad to 4 byte boundary.
    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编码器包含一个单一的参数，即将要编码的数据，并且返回被编码的数据。现在可以将这个定义的编码器关联到包含可模糊化原始类型的一个块，并且允许模糊器的开发者继续工作就好像这个小的障碍不存在一样。

依赖

依赖允许你将一个条件应用于一个完整块的显示中。其实现过程如下，首先使用可选的dep关键字参数将一个块与它将要依赖的原始类型相连接。当Sulley准备显示依赖块时，它将会检查所连接的原始类型的值并相应的采取行为。可以使用dep_value关键字参数来指定一个依赖值。另外，可以使用dep_values关键字参数来指定依赖值的一个列表。最后，可以通过使用dep_compare关键字参数来修改实际的条件比较。例如，考虑这样一个情形，取决于一个整数的值而期望得到不同的数据：

```
s_short("opcode", full_range=True)

# opcode 10 expects an authentication sequence.
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()

# opcodes 15 and 16 expect a single string hostname.
if s_block_start("hostname", dep="opcode", dep_values=[15, 16]):
    s_string("pedram.openrce.org")
s_block_end()

# the rest of the opcodes take a string prefixed with two underscores.
if s_block_start("something", dep="opcode", dep_values=[10, 15, 16],
dep_compare="!="):
    s_static("__")
    s_string("some string")
s_block_end()
```

可以采用多种方法将块依赖串接起来，以构成功能更加强大（但也更为复杂）的组合。

块帮助函数

为了有效的利用Sulley，你必须要熟悉的关于数据生成的一个重要方面就是块帮助函数。这些函数包括大小计算函数（sizers）、校验和函数（checksums）和重复函数（repeaters）等。

大小计算函数

SPIKE的用户将会对块帮助函数s_sizer()（或s_size()）非常熟悉。该帮助函数使用块的名字来度量第一个参数的大小，并且接受如下的额外关键字参数：

- length（整数值，默认值是4）容量字段的长度。
- endian（字符型，默认值为“<”）：位字段的存放顺序。将低位字节先存放在低地址处的顺序指定为“<”，将高位字节先存放在低地址处的顺序指定为“>”。
- format（字符串型，默认值是“binary”）：将输出格式指定为“binary”或者“ascii”，用以控制整数原始类型的显示格式。

- inclusive（布尔型，默认值是False）：大小计算函数应当计算其自己的长度吗？
- signed（布尔型，默认值是False）：将大小设置为有符号的或者无符号的，只有当输出格式设定为“ascii”时才可应用该类型。
- fuzzable（布尔型，默认值是False）：该原始类型的模糊测试设置为使能或者不使能。
- name（字符串型，默认值是None）：在所有的Sulley对象中，指定一个名字就可以在整个请求中直接访问该原始类型。

大小计算函数是数据生成中的一个重要构件，它允许表示诸如XDR符号和ASN.1等的复杂协议。当显示大小计算函数时，Sulley将会动态计算所关联块的长度。默认情况下，Sulley将不会对size字段进行模糊测试。在许多情况下我们需要进行这样的处理，然而，如果不想这样处理，那么就将可模糊化标志设置为使能。

校验和

类似于大小计算函数s_checksum()帮助函数使用块的名字来计算第一个参数的校验和。同时也可以指定下面可选的关键字参数：

- algorithm（字符串型或者是函数指针，默认值是“crc32”）：应用于目标块的校验和算法（crc32,adler32,md5,sha1）。
- endian（字符型，默认值为<）：位字段的存放顺序。将低位字节先存放在低地址处的顺序指定为<，将高位字节先存放在低地址处的顺序指定为>。
- length（整数值，默认值是0）：校验和的长度，其值为0时表示自动计算。
- name（字符串型，默认值是None）：在所有的Sulley对象中，指定一个名字就可以在整个请求中直接访问该原始类型。

该算法的参数可以是crc32,adler32,md5或者sha1中的一个。另外，你还可以为该参数指定一个函数指针，以应用一个定制的校验和算法。

重复函数

可以使用s_repeat()（或者s_repeater()函数）对一个块重复可变数量的次数。例如，当对带有多个元素的表中的溢出进行测试时，该函数是非常有用的。该帮助函数包含三个强制使用的参数：将要被重复的块的名字，重复的最小次数以及重复的最大次数。另外，也可以指定下面可选的关键字参数：

- step（整数值，默认值是1）：在最小重复次数和最大重复次数之间的单步计数。
- fuzzable（布尔型，默认值是False）：将对该原始类型的模糊测试设置为使能或者不使能。
- name（字符串型，默认值是None）：在所有的Sulley对象中，指定一个名字就可以在整个请求中直接访问该原始类型。

考虑下面的例子，它将所介绍的三个帮助函数连接到了一起。我们正在对包含一个字符串表的协议的一部分进行模糊测试。该表中的每个条目包含以下内容：一个两字节字符串类型的字段，一个两字节长度字段，一个字符串字段，最后是一个对字符串字段进行计算的CRC-32校验和字段。由于我们不知道这些类型字段的有效值是什么，因此我们将使用随机数据对它们进行模糊测试。下面是协议的该部分在Sulley中的表现形式：

```
# table entry: [type][len][string][checksum]
if s_block_start("table entry"):
    # we don't know what the valid types are, so we'll fill this in with random data.
    s_random("\x00\x00", 2, 2)

    # next, we insert a sizer of length 2 for the string field to follow.
    s_size("string field", length=2)

    # block helpers only apply to blocks, so encapsulate the string primitive in one.
    if s_block_start("string field"):
        # the default string will simply be a short sequence of Cs.
        s_string("C" * 10)
    s_block_end()

    # append the CRC-32 checksum of the string to the table entry.
    s_checksum("string field")
s_block_end()

# repeat the table entry from 100 to 1,000 reps stepping 50 elements on each
iteration.
s_repeat("table entry", min_reps=100, max_reps=1000, step=50)
```

这个Sulley脚本将不仅对表条目的解析过程进行模糊测试，而且还可能会在对超长表的处理中发现一个错误。

Legos

Sulley利用legos来表示用户定义的构件，如邮件地址、主机名以及在微软的RPC，XDR，ASN.1以及其他标准中所使用的协议原始类型。在ASN.1/BER中，字符串被表示为[0x04][0x84][双字长度][字符串]的序列。当对一个基于ASN.1的协议进行模糊测试时，在每个字符串的前面加上长度和类型前缀是非常繁琐的。取而代之，我们可以定义一个lego，并对其进行引用：

```
s_lego("ber_string", "anonymous")
```

几乎所有的lego都遵循着一个类似的格式，其中的一个例外情形是可选的options关键字参数，它是特定于单个的lego的。作为一个简单的例子，考虑如下的对tag lego的定义，它有助于对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, request, None, None, None, None)

        self.value   = value
        self.options = options

        if not self.value:
            raise sex.error("MISSING LEGO.tag DEFAULT VALUE")

        #
        # [delim][string][delim]

        self.push(primitives.delim("<"))
```

```
        self.push(primitives.string(self.value))
        self.push(primitives.delim(">"))
```

这个示例lego简单地将所需要的标签作为一个字符串来接受，并且将其封装到适当的分隔符中。它是通过下面的操作来实现这一点的，即将块类进行扩展，并且通过self.push()手工将标签分隔符和用户提供的字符串添加到块中。

这里还有另外一个例子，它生成一个简单的lego以在Sulley中表示ASN.1/BER[⊖]整数。采用最低层次的通用命名符将所有的整数表示为4字节整数，其格式如下：[0x02][0x04][双字]，其中0x02指明了整数的类型，0x04指定了该整数的长度为4字节，双字表示我们正在解析的实际整数值。下面是摘自于sulley\legos\ber.py的示例定义：

```
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):
        # let the parent do the initial render.
        blocks.block.render(self)

        self.rendered = "\x02\x04" + self.rendered
        return self.rendered
```

类似于前面的例子，使用self.push()将所提供的整数值添加到了一个块的栈中。但与前面的例子不同的是，render()例程被重载以在所显示内容的前面加上了\x02\x04的静态序列作为前缀，从而用于满足前面所描述的整数表示需求。伴随着每个新模糊器的创建，Sulley也处于不断的发展完善中。所开发的块和请求扩展了请求库，并且可以在后续模糊器的创建过程中被方便地引用和使用。下面我们来看一下如何创建一个会话。

21.4.3 会话

一旦你已经定义了许多的请求，那么就可以将它们在一个会话中连接起来。同其他模糊测试框架相比，Sulley所具有的一个主要的优越性在于，它具备对协议进行深层次模糊测试的能力。这一功能是通过在一个图中将请求连接在一起来实现的。在下面的例子中，一系列的请求被连接到了一起，并且利用pgraph库以uDraw格式将图形加以显示，而会话和请求类都是从pgraph库扩展而来的，如图21-2所示：

⊖ http://luca.ntop.org/Teaching/Appunti/asn1.html

```
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 = sessions.session()
sess.connect(s_get("helo"))
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("session_test.udg", "w+")
fh.write(sess.render_graph_udraw())
fh.close()
```

当开始进行模糊测试时，Sulley将从根节点开始遍历该图的结构，并且同时对每一个构件进行模糊测试。在这个例子中，它是从helo请求开始进行遍历的。一旦完成对该请求的模糊测试，Sulley将开始对mail from请求进行模糊测试。它通过给每个测试用例添加一个有效的helo请求的前缀来实现这一功能。接下来，Sulley将转向对rcpt to请求进行模糊测试。同样，这也是通过给每个测试用例添加一个有效的helo请求和mail from请求的前缀来实现的。该过程继续对data进行处理，然后从ehlo路径重新开始向下执行模糊测试过程。这种将协议划分为单个的请求，并且通过所构建的协议图来对所有可能的路径进行模糊测试的能力是非常强大的。例如，考虑Ipswitch协作组于2006年9月所披露的一个问题[⊖]。在这个例子中，所发现的软件错误是在解析包含在字符@和:内部的长字符串时产生的一个栈溢出。使这个例子令人感兴趣的是该漏洞只通过EHLO路径被暴露，而通过HELO路径则没有被暴露。如果模糊器不能够遍历所有可能的协议路径，那么诸如此类的问题就可能被漏过。

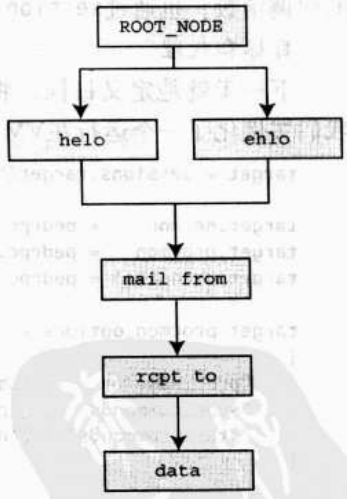


图21-2 SMTP会话图结构的示例

⊖ http://www.zerodayinitiative.com/advisories/ZDI-06-028.html

当实例化一个会话时，可以指定如下所示的可选的关键字参数：

- sessio_filename（字符串型，默认值是None）：序列化持久数据的文件名。指定一个文件名将允许你停止并重新启动模糊器。
- skip（整数值，默认值是0）：将要被略过的测试用例数。
- sleep_time（浮点数，默认值是1.0）：在传输测试用例之间的休眠时间。
- log_level（整数值，默认值是2）：设置日志级别，一个更大的数字表示将包含更多的日志消息。
- proto（字符串型，默认值是“tcp”）：通信协议。
- timeout（浮点数，默认值是5.0）：在超时之前等待send()或recv()返回的秒数。

Sulley所具备的另外一个高级特性是，它可以在协议图结构中所定义的每条边上记录回调函数。这就允许我们在节点传输之间记录一个将调用的函数，以实现诸如挑战响应系统这样的功能。回调方法必须采用下面的原型加以定义：

```
def callback(node, edge, last_recv, sock)
```

在上述原型定义中，node表示将要被发送的节点，edge是针对node的模糊测试路径的最后一条边，last_recv包含从最后一次socket传输所返回的数据，sock是当前活动的socket。回调函数在下面的情形中也是有用的，例如下一个包的大小被指定在第一个包中。作为另外一个例子，如果你需要填充目标应用的动态IP地址，那么就记录一个从sock.getpeername()[0]拦截IP的回调函数。也通过session.connect()方法可选的关键字参数callback来记录边的回调函数。

目标和代理

下一步就是定义目标，将它们同代理相连接，并且将目标添加到会话中。下面的例子中，我们实例化了一个运行在VMWare虚拟机中的新目标，并且将其与三个代理相连接：

```
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()
```

实例化的目标被绑定在IP地址为10.0.0.1的主机上的TCP端口5168。在目标系统中运行着一个网络监视代理，该代理默认在端口26001进行监听。网络监视器将所有的socket通信记录到单独的PCAP文件中，这些文件通过测试用例号来进行标识。进程监视代理也运行在目标系统中，默认在端口26002处进行监听。该代理接受额外的参数以指明将要关联到的进程名，停止目标

进程执行的命令以及启动目标进程执行的命令。最后，VMWare控制代理是运行在本地系统中的，默认在端口26003处进行监听。将目标添加到会话中，然后就可以开始进行模糊测试了。Sulley可以对多种目标进行模糊测试，对每个目标使用连接代理的唯一集合进行测试。这样，就可以通过将总的测试空间划分到不同的目标来节省测试时间。

下面让我们来详细讨论一下每个单独代理的功能。

代理：网络监视器（network_monitor.py）

网络监视器代理负责监视网络通信，并且将它们作为日志记录到磁盘上的PCAP文件中。该代理被硬编码并被绑定到TCP端口26001，并且通过PedRPC定制的二进制协议接受来自于Sulley会话的连接。在将一个测试用例传递到目标之前，Sulley将关联到该代理，并且要求它开始记录网络通信。一旦测试用例被成功传递，Sulley将再次关联到此代理，并且要求它将所记录的通信导出到磁盘上的一个PCAP文件中。用测试用例号来命名PCAP文件，以便于检索。该代理不需要被部署到与目标软件相同的系统中。然而，它必须可见地发送和接收网络通信。该代理接受如下所示的命令行参数：

```
ERR> USAGE: network_monitor.py
    <-d|-device DEVICE #>    device to sniff on (see list below)
    [-f|-filter PCAP FILTER] BPF filter string
    [-p|-log_path PATH]      log directory to store pcaps to
    [-l|-log_level LEVEL]    log level (default 1), increase for more verbosity

Network Device List:
    [0] \Device\NPF_GenericDialupAdapter
    [1] {2D938150-427D-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
    ...
```

代理：进程监视器（process_monitor.py）

进程监视器代理负责检测模糊测试目标进程中可能发生的错误。该代理被硬编码并绑定到TCP端口26002，并且通过PedRPC定制的二进制协议接受来自于Sulley会话的连接。成功地将每个单独的测试用例传递到目标之后，Sulley将关联到该代理以确定是否有一个错误被触发。如果有错误被触发，那么有关错误本质的高层信息将被传递回Sulley会话，以在Web服务器内部显示（稍后将对此进行更加详细的讨论）。所触发的错误也被记录到一个序列化的“崩溃二进制文件”中，以用于后续分析。我们在后面将对该功能进行更加详细的研究。该代理接受如下所示的命令行参数：

```
ERR> USAGE: process_monitor.py
    <-c|-crash_bin FILENAME> filename to serialize crash bin class to
    [-p|-proc_name NAME]     process name to search for and attach to
    [-i|-ignore_pid PID]     ignore this PID when searching for the target process
    [-l|-log_level LEVEL]    log level (default 1), increase for more verbosity
```

代理：VMWare控制（vmcontrol.py）

VMWare控制代理被硬编码并绑定到TCP端口26003，并且通过PedRPC定制的二进制协议

接受来自于Sulley会话的连接。该代理提供了一个API以同一个虚拟机映像进行通信，包括具备启动、停止、挂起或者重设映像的能力，同时也可以获取、删除和恢复快照。如果一个错误已经被发现或者目标不能到达，Sulley可以连接到该代理并且将虚拟机恢复到一个已知的好的状态。测试序列处理工具将主要依赖该代理来实现其功能，即识别触发任何给定的复杂错误的准确的测试用例序列。该代理接受如下所示的命令行参数：

```
ERR> USAGE: vmcontrol.py
    <-x|-vmx FILENAME>    path to VMX to control
    <-r|-vmrun FILENAME>  path to vmrun.exe
    [-s|-snapshot NAME>   set the snapshot name
    [-l|-log_level LEVEL] log level (default 1), increase for more verbosity
```

Web监视接口

Sulley会话类含有一个内嵌的最小化Web服务器，该服务器被硬编码并绑定到端口26000。一旦会话类的fuzz()方法被调用，那么该Web服务器线程将解锁，并且模糊器的执行进度包括中间结果都可以被显示出来。图21-3显示了一个示例屏幕快照。



图21-3　Sulley Web监视接口

可以通过点击适当的按钮来暂停和恢复模糊器的执行。每个所检测到错误的概要信息将被作为一个列表显示，该列表的第一列是导致错误发生的测试用例的编号。点击测试用例编号将会加载错误发生时的详细崩溃信息。当然，该信息在崩溃二进制文件中也是可用的，并且可以通过编程来获取。一旦会话被执行完毕，就进入了事后验证阶段并开始分析所生成的结果。

21.4.4　事后验证阶段

一旦一个Sulley模糊测试会话过程执行完毕，下面就开始对结果进行评审并且进入到事后验证阶段。会话的内嵌Web服务器将会向你提供有关潜在的未发现问题的早期提示信息，但实际上这时你将会对结果进行分隔。在这个处理过程中，有一些可用的工具可以为你提供帮助。第一个是crash-bin_explorer.py工具，该工具接受如下的命令行参数：

```
$ ./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
```

例如，我们可以使用该工具来查看检测到错误的每个位置，并且进一步列出在该地址触发一个错误的单个测试用例编号。下面的结果来自于一个现实世界中的审核，即对Trillian Jabber协议解析器的审核：

```
$ ./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] rendezvous.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,
```

在上面所列出的错误点中，没有一个错误可能会作为一个明显的可利用问题而显得突出。我们可以通过使用-t命令行开关来指定一个测试用例编号，从而深入到一个单个错误的特定方面中。下面让我们来看一下编号为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
```

同样，这里也没有非常明显的内容，但是我们知道我们正在影响这个特定的访问违规，因为包含ASCII字符串“&;tg”的ECX寄存器被非法解除引用。可能会发生字符串扩展问题吗？我们可以采用图形化的方式来查看发生崩溃的位置，这种方法增加了一个额外的维度，可以使用-g命令行开关来在该维度上显示已知的执行路径。下面所生成的图形（图21-4）仍然来自于对Trillian Jaber解析器的一个真实的审核。

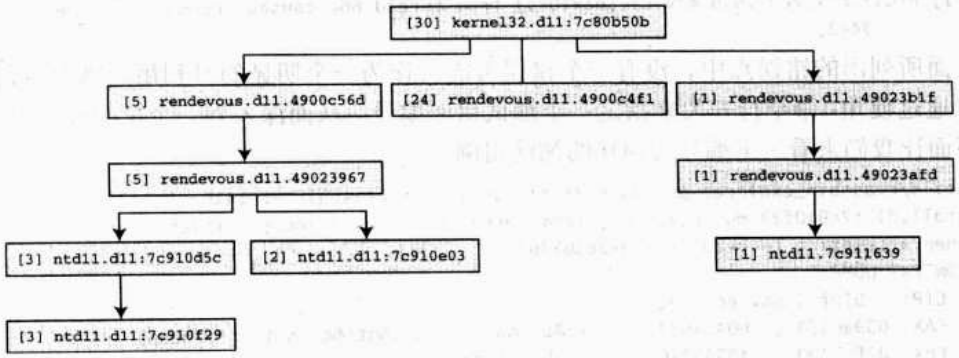


图21-4　Sulley崩溃二进制文件的图形化研究

可以看到尽管我们已经发现了4个不同的崩溃位置，但是问题的根源似乎是相同的。进一步的研究表明情况确实如此。该特定的缺陷存在于可扩展消息处理现场协议（Rendezvous/Extensible Messaging and Presence Protocol,XMPP）通信子系统中。Trillian通过UDP端口5353上的_presence mDNS（多点传送DNS）服务来定位邻近的用户。一旦一个用户通过mDNS进行了注册，那么就通过TCP端口5298上的XMPP来完成通信。在plugins\rendezvous.dll内部，应用了如下逻辑代码来接收消息：

```
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
```

该代码计算了所提供消息的字符串长度，并且分配了一个长度为+128的堆缓冲区来保存消息的副本，然后该缓冲区将通过expatxml.xmlComposeString()来进行传递，该函数的调用原型如下所示：

```
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;
};
```

xmlComposeString()例程将一直调用到expatxml.19002420()，在该例程中，HTML分别将字符&>和<编码为&>和<。在如下的汇编语言代码片段中，我们可以看到该操作的相关情况：

```
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
```

由于最初所计算的字符串长度没有考虑这个字符串的扩展问题，因此在rendezvous.dll内部如下后续的内嵌的内存复制操作将会触发一个可利用的内存破坏漏洞：

```
4900C4EC mov ecx, eax
4900C4EE shr ecx, 2
4900C4F1 rep movsd
4900C4F3 mov ecx, eax
4900C4F5 and ecx, 3
4900C4F8 rep movsb
```

Sulley所检测到的每个错误都是同这个逻辑错误直接相关的。对错误的发生位置和路径进行追踪将允许我们快速做出假定是某个单一根源导致了该错误。我们可能想要执行的最后一个步骤就是删除所有不包含错误相关信息的PCAP文件。可以编写pcap_cleaner.py构件来很好地完成这个操作：

```
$ ./utils/pcap_cleaner.py
    USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

该构件将打开特定的崩溃二进制文件，在触发错误的测试用例编号列表中进行读取，并且从特定的目录中删除所有其他的PCAP文件。为了从头到尾更好地理解整个过程的运行机理及

其内部的相互关系，我们将对一个完整的现实世界中的审核例子进行详细分析。

21.4.5　一个完整的实例分析

这个例子涉及了许多高级的Sulley概念，并且非常有助于加深对该框架的理解。在对这个例子的分析过程中，我们省略了许多关于目标特定方面的细节，因为这一部分的主要目的是说明Sulley的一些高级特性的用法。我们所选择的目标是Trend Micro服务器保护，它是通过SpntSvc.exe服务被绑定到TCP端口5168的一个微软DCE/PRC端点。PRC端点是从TmRpcSrv.dll中被暴露的，并且带有如下的接口定义语言（IDL）所描述的桩信息：

```
// opcode: 0x00, address: 0x65741030
// uuid: 25288888-bd5b-11d1-9d53-0080c83a5c2c
// version: 1.0

error_status_t rpc_opnum_0 (
[in] handle_t arg_1,                          // not sent on wire
[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[],           // not sent on wire
[in] long arg_6
);
```

实际上，参数arg_1和arg_6都没有通过网络被传递。当我们在后面编写实际的模糊测试需求时，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。该参数的上下两部分控制着一对跳转表，该表通过这个单一的PRC函数暴露出了多余的可达子例程。对该跳转表进行逆向工程可以发现如下的组合：

- 当上半部分的值是0x0001时，1到21是有效的下半部分值。
- 当上半部分的值是0x0002时，1到18是有效的下半部分值。
- 当上半部分的值是0x0003时，1到84是有效的下半部分值。
- 当上半部分的值是0x0005时，1到24是有效的下半部分值。
- 当上半部分的值是0x000A时，1到48是有效的下半部分值。
- 当上半部分的值是0x001F时，1到24是有效的下半部分值。

接下来，我们必须要创建一个定制的编码例程，该例程的作用是将所定义的块封装为一个有效的DCE/RPC请求。这里只有一个单独的函数编号，因此是比较简单的。我们在utisl.dcerpc.request()的外面定义了一个基本的封装函数，它将操作码参数硬编码为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)
```

创建请求

在这些信息和我们创建的编码器的帮助下，我们就可以开始定义Sulley请求了。我们创建一个requests\trend.py文件以包含所有的同Trend相关的请求以及帮助函数的定义，并且开始进

行编码。这是一个极好的例子，它说明了在一种语言（相对于一种定制语言）的内部创建一个模糊器请求是多么的有益，因为我们利用了一些Python循环来为trend_req_num参数的每个有效的上半部分值自动生成一个单独的请求：

```
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 endian 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()
```

在每个所生成的请求内部，一个新的块被初始化，并且被传递到我们以前定义的定制编码器中。接下来，使用s_group()来定义一个名为subs的序列以代表我们早先看到的trend_req_num参数的下半部分值。然后，上半部分字值被作为一个静态值添加到请求流中。我们将不会对trend_req_num参数进行模糊测试，因为我们已经对其有效值进行了逆向工程；如果没有进行逆向工程，那么我们同样可以对这些字段进行模糊测试。接下来，some_string的NDR大小前缀被添加到请求。我们可以在这里可选地使用Sulley DCE/RPC NDR lego原始类型，但是由于RPC请求是如此简单，因此我们决定手工来表示NDR格式。接下来，some_string值被添加到请求中。该字符串的值被封装到一个块中，因此其长度可以被度量。在这个例子中，我们使用一个静态大小的、由字符A构成的字符串（大约20k）。通常情况下，我们将在这里插入一个s_string()，但是由于我们知道Trend将因为任意的长字符串而发生崩溃，因此通过使用一个静态值来减小测试集的大小。字符串的长度再次被追加到请求中，以实现针对arg_4参数的size_is需求。最后，我们为输出缓冲区大小指定一个任意的静态大小，并且关闭该块。现在，我们的请求已经做好了准备，我们可以开始创建一个会话。

创建会话

我们在Sulley的最顶层文件夹中创建一个名为fuzz_trend_server_protect_5168.py的新文件，用于我们的会话。该文件后来被移动到archived_fuzzies文件夹中，因为它已经完成了其生命周期。首先进行最重要的操作，即导入Sulley并且从请求库中导入所创建的Trend请求：

```
from sulley   import *
from requests import trend
```

接下来，我们将要定义一个Presend函数，以在传输任何单独的测试用例之前建立DCE/RPC连接。Presend例程接受一个单一的参数，即将要传输数据的socket端口。由于Sulley构件例程utils.dcerpc.bind()的可用性，使得该例程的编写非常简单，如下所示：

```
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))
```

这时候，就可以初始化会话并定义一个目标了。我们将对一个单一目标进行模糊测试，即在VMWare虚拟机内部地址10.0.0.1处安装的Trend服务器保护程序。我们将通过把序列化的会话信息保存到审核目录来遵循该框架的指南。最后，我们使用所定义的目标来注册一个网络监视器，进程监视器以及虚拟机控制代理：

```
sess   = sessions.session(session_filename="audits/trend_server_protect_5168.session")
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)
```

由于提供了一个VMWare控制代理，所以只有一个错误被检测到或者不能到达目标，那么Sulley在默认情况下将转换为一个已知的好的快照。如果一个VMWare控制代理不可用但一个进程监视器代理是可用的，那么Sulley将试图重新启动目标进程以恢复模糊测试。可以通过将stop_commands和start_commands选项指定为进程监视器代理来实现此功能：

```
target.procmon_options = \
{
    "proc_name"      : "SpntSvc.exe",
    "stop_commands"  : ['net stop "trend serverprotect"'],
    "start_commands" : ['net start "trend serverprotect"'],
}
```

无论何时使用进程监视器代理，proc_name参数都是强制使用的，它指定了调试器应当被关联到的进程的名字，以及在哪个进程中寻找错误。如果VMWare控制代理和进程监视器代理都不可用，那么Sulley将没有其他选择，如果数据传输没有成功完成，那么它只能简单地提供目标时间以供恢复。

接下来，我们通过调用VMWare控制代理restart_target()来使目标应用开始运行。一旦目标应用开始运行，目标就被添加到会话中，presend例程被定义，并且每个被定义的请求都被连接到模糊测试的根节点。最后，通过对会话类的fuzz()例程的调用来开始执行模糊测试。

```
# start up the target.
target.vmcontrol.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()
```

建立环境

在启动模糊会话之前需要做的最后一个步骤就是建立环境。我们是通过如下操作来实现这一功能的，即建立目标虚拟机映像，并且使用如下的命令行参数在测试映像内直接启动网络和进程监视器代理：

```
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
```

网络和进程监视器代理都在与会话脚本所运行的Sulley的顶层目录相应的一个映射共享库中执行。一个伯克利包过滤器（Berkeley Packet Filter，BPF）的过滤字符串传递到网络监视器以确保只有我们所感兴趣的包被记录。在审核文件夹内部的一个目录也同样被选择，网络监视器将在该目录中为每个测试用例创建PCAP。当网络和进程监视器代理以及目标进程都开始运行后，当命名的sulley准备好并处于等待状态时，将生成一个现场快照。

接下来，我们将关闭VMWare，并且在主机系统上（模糊测试系统）启动VMWare控制代理。该代理要求到vmrun.exe的路径是可执行的，到实际映像的路径是可控制的，并且如果当数据传输的错误检测失败时，快照的名字可以进行转换：

```
vmcontrol.py -r "c:\\VMware\vmrun.exe"
             -x "v:\vmfarm\Trend\win_2000_pro.vmx"
             -snapshot "sulley ready and waiting"
```

准备，设置，采取行动！以及事后验证工作

最后，我们一切准备就绪了。只需要简单地启动fuzz_server_protect_5168.py，将一个Web浏览器连接到http://127.0.0.1:26000以监视模糊器的执行进度，只是进行观察而不采取任何操作。

当模糊器执行完所有221个测试用例之后，我们发现其中的19个测试用例触发了错误。使用crashbin_explorer.py工具，我们就可以对异常地址所分类的错误进行研究：

```
$ ./utils/crashbin_explorer.py audits/trend_server_protect_5168.crashbin
    [6] [INVALID]:41414141 Unable to disassemble at 41414141 from 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 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,
```

其中某些错误是非常明显的可利用问题，例如，导致0x41414141的EIP的测试用例。测试用例70则似乎是在无意中发现了一个可能的代码执行问题，即Unicode溢出（实际上，通过更加深入的研究就可以发现这是一个很直接的溢出）。崩溃二进制文件资源管理器工具可以生成针对所检测错误的一个图形化视图，基于所观察到的栈回路来绘制路径。这可以有助于查明特定问题的根源所在。该工具接受如下的命令行参数：

```
$ ./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
```

例如，我们可以进一步研究与测试用例70相对应的、在错误被检测时的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 (  39441812) -> 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 disassemble
SEH unwind:
        0259fc58 -> StRpcSrv.dll:656784e3
        0259fd70 -> TmRpcSrv.dll:65741820
        0259fda8 -> TmRpcSrv.dll:65741820
        0259ffdc -> RPCRT4.dll:77d87000
        ffffffff -> KERNEL32.dll:7c5c216c
```

你在这里可以看到栈已经被文件扩展名的Unicode字符串清空。你同样可以为给定的测试用例输出已存档的PCAP文件。图21-5显示了来自于Wireshark的一个屏幕快照截图，显示了其中一个被捕获的PCAP文件的内容：

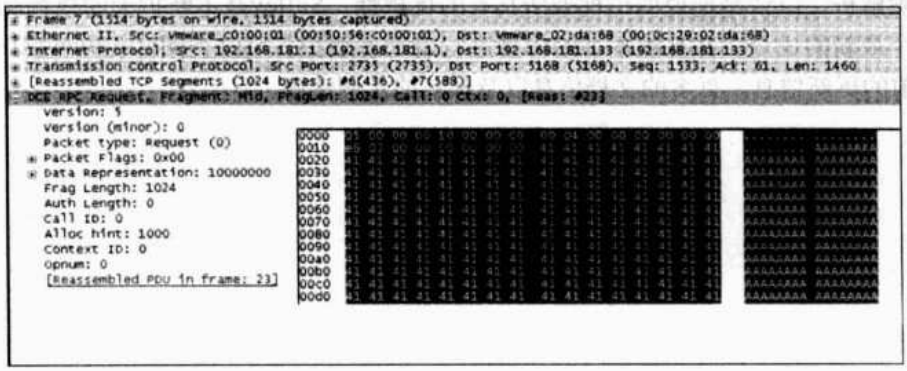


图21-5 Wireshark DCE/RPC 解析器

我们想要进行的最后一个步骤就是删除不包含与错误相关信息的所有PCAP文件。所编写的pcap_cleaner.py的工具可以实现这一功能，如下所示：

```
$ ./utils/pcap_cleaner.py
   USAGE: pcap_cleaner.py <xxx.crashbin> <path to pcaps>
```

该工具将打开特定的崩溃二进制文件，对触发错误的测试用例列表进行读取，并且从特定的目录中删除所有其他的PCAP文件。在此模糊测试中所发现的代码执行漏洞都被报告给了Trend，并且生成了如下的建议文档：

- TSRT-07-01：Trend Micro ServerProtect StCommon.dll栈溢出漏洞
- TSRT-07-02: Trend Micro ServerProtect eng50.dll栈溢出漏洞

但是，这并不是说所有可能的漏洞都在这个接口中被发现。实际上，这是针对该接口的最基本的模糊测试可能。相对于简单地使用一个长字符串而言，实际使用s_string()的辅助模糊测试现在可以给我们带来好处。

21.5 小结

模糊测试框架为错误检测者和QA团队提供了一个灵活的、可重用的、同构的开发环境。在本章中对可用的模糊测试框架列表进行分析之后，可以得出明确的结论即没有任何一种框架

是最好的。所有这些框架都提供了一种合理的框架结构以便于在其基础上进行构造，但是对于特定框架的选择通常是由模糊测试的目标应用以及对特定框架所使用的编程语言的熟悉程度来决定的。Adobe Macromedia Shockwave Flash的SWF文件格式实例研究说明了该框架仍然采用了一种不成熟的技术，并且你仍然可能会遇到这样的情形，即没有任何现有的框架能够满足你的需求。正如前面所说明的，有时候可能会需要开发一个定制的模糊测试解决方案，利用为手头工作所开发的可重用的构件。我们希望当你遇到你自己独特的模糊测试任务时，该实例研究可以提供一些有用的帮助。

本章的最后一部分，我们介绍并研究了一种新的模糊测试框架Sulley。该模糊测试框架的不同方面被解析，以清楚地说明该框架所具有的优越性。Sulley是不断扩大的模糊测试框架家族中的一个最新的成员，并且被本书的作者进行着动态维护。可以查阅http://www.fuzzing.org来获得当前的更新信息、更多的文档信息以及最新的示例资源。





第22章 自动化协议解析

“我知道你为你的家庭提供穿衣吃饭是多么的困难。”

——George W. Bush,Greater Nashua，NH，2000年1月27日

我们在上一章中对多种类型的模糊器进行了较为详细的讨论。这包括从简单的对字节实施变异的通用文件模糊器到当前使用最广泛的模糊测试框架，模糊测试技术克服了自动化软件测试的许多障碍。在本章中，我们将介绍模糊测试的最高级形式，并且试图去解决所有模糊测试技术中的一个难题。特别的，我们将解决一个棘手的问题，即将一个协议分解为其基本的构造块。

在本章的开始部分，首先将讨论面向自动化协议解析的一些基本技术，然后将深入到生物信息学和通用算法的应用问题。在自动化软件测试的这些研究领域，这些课题具有很强的理论性，当前正被许多研究者进行广泛的研究。

22.1 模糊测试存在的问题是什么

对于模糊测试而言，最困难的一个方面就是入门时的障碍，特别是当处理一些非公开的、复杂的二进制协议时，需要进行大量的研究以理解其内涵。想像一下1992年以前，即Samba[⊖]的第一个版本被发布之前对微软的SMB协议进行模糊测试的情形。Samba项目后来发展成为一个功能完整的、与Windows相兼容的SMB客户机－服务器项目。这可能要归功于无数志愿者的工时数。你目前拥有如此大的一个团队，并且在十几年后对类似的一个非公开的私有协议进行模糊测试吗？

SAMBA

SAMBA是于1992年1月10日被发布的，当时来自于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[⊖]。在那条消息中，他宣布了与Pathworks相兼容的一个UNIX文件共享服务针对DOS的可用性。该项目后来成为人们所熟知的nbserver，但是其名字于1994年4月被变更为Samba，因为当时Tridgell接收到了来自于Syntax公司的一封信件，该信件中表明了可能存在的商标侵权问题[⊖]。十五年之后，Samba已经成为了最广泛使用的开源项目之一。

对你自己的协议进行模糊测试是非常简单的。因为你对该协议的格式、细微差别以及复杂性都非常熟悉，甚至对那些你在打瞌睡的时候编写出来的代码区，假如事后要重新关注的话，

⊖ http://www.samba.org

⊖ http://groups.google.com/group/vmsnet.networks.desktop.pathworks/msg/7d939a9e7e419b9c

⊖ http://www.samba.org/samba/docs/10years.html

也很熟悉。同样，对一个公开的协议如HTTP进行模糊测试相对来说也较为简单。详细的RFC[⊖]以及其他的公开文档都可以被用做参考。对于数据条目而言，可以简单地生成一系列有效而完整的测试用例。但是需要注意的是，即使一个协议既是常用的又是公开的，也不能保证一个软件提供商会严格遵守已发布的标准。考虑现实世界中Ipswitch IMail的例子。在2006年9月，一个远程可利用的栈溢出漏洞被公开揭露[⊜]，该漏洞影响了Ipswitch的SMTP daemon应用。该缺陷的存在是由于以下原因，即在对包含在字符@和:内部的长字符串进行解析时，缺乏相应的边界检查。当查看二进制代码中的相关部分时，发现该问题很难被发现。然而，由于该漏洞功能是Ipswitch的一个私有特性，并且没有在任何标准中被公开发布，因此模糊器很少有机会能够发现该漏洞。

对一个私有的第三方协议进行模糊测试，无论该协议多么简单，都存在着巨大的挑战性。纯粹的随机模糊测试虽然是最快的实现方法，但其很难产生智能化的结果。其他通用技术，例如字节翻转，虽然提供了更好的测试但它需要进行一个额外的步骤，即首先要观测客户端和服务器之间的有效通信。幸运的是，在大多数情况下触发任意客户机—服务器之间的通信是一项较为简单的工作。然而，无法肯定地指出该协议有百分之多少的部分仍然没有被观测。例如，在没有任何先验知识的假设下使用HTTP协议。对通信观测一小段时间将可能会发现GET和POST动作。然而，更加隐蔽的一些动作如HEAD,OPTIONS以及TRACE将不太可能被观测到。对任意一个协议而言，我们如何来发现其中很少被使用的部分呢？为了更好的对目标协议进行解析，可以对客户机和服务器的二进制代码应用逆向工程。这是一个需要花费很高的代价才能完成的步骤，因为有经验的逆向工程师都具有很强的技能并且很难找到。在某些情况下，特别是对于纯文本协议，有时候可能会猜想要具备额外的功能。这个高代价的操作对于二进制协议而言在很少情况下是可用的。

在花费高昂的代价以雇佣或分派一个具有一定技能的逆向工程师之前，首先要确保已经利用了其他人的工作成果。世界是一个大舞台，其中所存在的机会就是如果你想要详细了解一个私有协议，那么也会有其他人也会面临同样的困境。在进行深入研究之前，要确保通过Google来搜索一下针对该问题的相关解决方案。你可能会惊讶地发现由曾经面临同样挑战的其他人所开发的非官方文档。Wotsit.org同样也为文件格式提供了丰富的参考信息，其中既有公开的信息也有非公开的信息。对于网络协议而言，Wireshark和Ethereal的源代码也提供了很好的学习起点，因为这些代码包含了对许多较为熟悉的、已知的私有协议的协议定义，而这些协议已经被彻底地解析过。

当解析一个私有协议的重任单独落在你自己的身上时，非常希望能够采用一种需要较少人工劳动的方法。本章致力于研究可以有助于实现在噪声中自动化检测信号或结构的不同方法，以及在网络协议和文件格式内部自动化检测数据的不同方法。

⊖ http://www.w3.org/Protocols/rfc2616/rfc2616.html

⊜ http://www.zerodayinitiative.com/advisories/ZDI-06-028.html

22.2 启发式技术

在本节中所提出的两种技术并不是自动化协议解析的真正形式。相反，这些基于启发式规则的技术可以用来提高模糊测试的自动化程度和性能。在这里提出这两种技术只是为了在深入研究更加复杂的方法之前做一个铺垫。

22.2.1 代理模糊测试

我们所讨论的第一个基于启发式规则的协议解析技术是被一个私下所开发[⊖]的模糊测试引擎——ProxyFuzzer来加以实现的。在典型的基于网络的客户机—服务器模式中，在客户机和服务器之间存在着一个直接的通信，如图22-1所示。

正如其名字所暗示的那样，一个代理模糊器必须要服务于客户机和服务器之间的连接，以将其自己定位为一个中继器。为了更加有效地完成这一任务，必须要手工对客户机和服务器进行配置[⊜]以在代理的地址处相互搜索。换句话说，客户机将代理看做是服务器，而服务器则将代理看做是客户机。包含模糊器连接的新的网络结构图如图22-2所示。

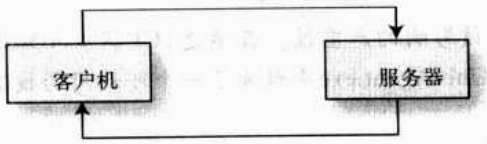


图22-1 典型的客户机－服务器通信路径

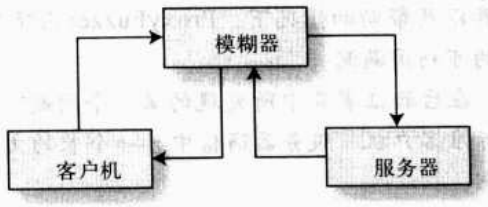


图22-2 透明的内嵌代理

图22-2中所示的位于节点之间的实连接线代表在此时，最初没有被修改的数据只是通过代理被简单地重定向。在其默认的模式下，当ProxyFuzzer采用一种适合于编程修改和回放的格式来记录事务时，它只是将客户机和服务器端所接收到的通信进行盲目的转发。这对于它自身来说是一个很方便的特性，因为这允许研究者忽略与生成和修改原始包捕获（PCAP）文件相关的中间步骤。同时，修改和重复所记录的事务对于转换未知协议而言是一个很好的人工方法，并且该技术已经被Matasano的协议调试器（PDB）很好地实现了[⊝]。

一旦ProxyFuzzer成功加以配置，并且通过了验证以准备执行时，那么其自动化的变异引擎就是可用的了。在这种模式下，ProxyFuzzer对中转的通信进行处理以寻找纯文本的构件。它是通过识别属于有效的ASCII范围并且在序列上长于一个所指定的门限的字节来实现这一功能的。一旦识别出这样的字节，那么所发现的字符串要么被随机延长，要么被无效的字符和格式化字符串标记所修改。图22-3中的虚线描述了被变异的字节流。

ProxyFuzzer的使用非常的简单，它提供了辅助的人工分析手段，使用其当前形式甚至已经

⊖ 通过TippingPoint的Cody Pierce可以查阅该工具的发布信息。

⊜ 这个过程实际上可以通过某些IP欺骗来自动完成，由于其过于简单所以这里没有加以讨论。

⊝ http://www.matasano.com/log/399/pdb-blackhat-talk-materials-as-promised/

发现了安全漏洞！内嵌（代理）模糊测试所带来的最大好处可能就是当个别字段被变异时，协议的大部分内容仍然保持不变。使大量数据的传输不被改变，这就允许通过较少的分析就可以成功地进行模糊测试。例如，可以考虑一下在数据层次上实现序列号的复杂协议。一个无效的序列号将导致立刻发现协议的破坏情况以及数据的剩余部分，包括没有被解析的模糊化字段。通过内嵌变异，一个模糊器就可以利用有效的事务并且生成足够有效的变异以到达解析器。

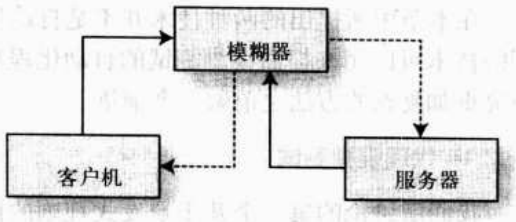


图22-3　自治的内嵌变异

ProxyFuzzer的结果

同前面章节所讨论过的最基本的模糊器相比，ProxyFuzzer仍然是非常基本的。然而，它早已经被证明对于主流的（明显没有被很好地进行编码的）商业软件是非常有效的。在没有任何工具以及帮助的情况下，ProxyFuzzer已经在计算机协会的Brightstor备份软件中发现了两个远程的可利用漏洞。

在后验证事务中所发现的第一个问题，削弱了该影响的严重性。在通过TCP端口6050进行的标准客户机－服务器通信中，一个长的文件名在UnivAgent.exe中触发了一个可利用的栈溢出漏洞。

所发现的第二个问题存在于igateway.exe HTTP daemon程序内部，该程序通过TCP端口5250来进行通信。该问题只有当调试标志是可用的时候才会出现，这也削弱了该影响的严重性。被请求的文件名通过对fprintf()的一个调用而被不安全地传递，从而暴露出了一个可利用的格式化字符串漏洞。

所发现的这两个问题在那些具有一些技巧性的漏洞集合中并不是特别突出。但是，如果考虑到发现这些错误所需要做的很少的工作，那么ProxyFuzzer可能就是首先应当使用的工具。

22.2.2　改进的代理模糊测试

目前正在进行的改进工作是为了使ProxyFuzzer更加智能化。额外的一些启发式方法也被加以考虑以有助于进行自动化字段检测，以及后续的变异操作。当前的通信解析器能够从一个二进制协议的剩余部分中分离出纯文本字段。被识别出的纯文本字段应当进一步加以处理，以寻找常用的字段分隔符和终结符，如空格和冒号等。另外，可以对被检测的字符串前面的原始字节进行搜索，以确定字符串长度和类型规范（也就是TLV字段）。于是，通过对一系列包的交叉检查就可以验证可能的长度前缀。同样，也可以对字符串后面的原始字节进行研究，以发现静态字段的内容。客户机和服务器的IP地址通常都出现在数据流的内部（在TCP/IP头的外面）。由于这些信息是已知的，因此分析引擎就可以扫描网络数据以搜索原始和ASCII表示。

通过假定最简单的协议可以被描述为若干字段的一个序列，而这些字段分布属于图22-4中

所显示的类别之一，那么我们就可以通过一系列的猜想和验证来开始对一个未知协议的层次结构进行建模。

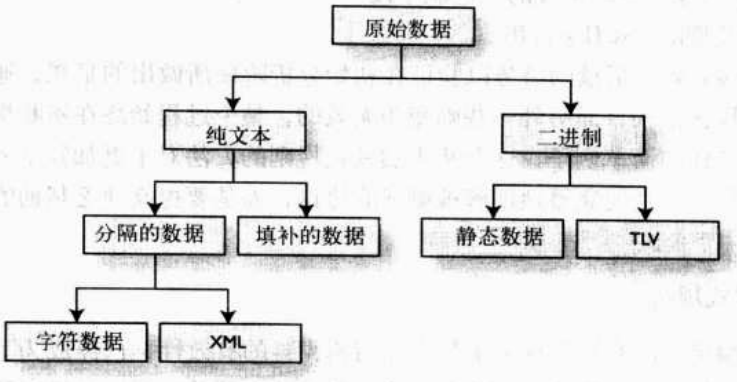


图22-4 层次化协议分类

这个基本的技术对于解析复杂的二进制协议如SMB而言，可能只会提供很少的帮助。然而，这里的最终目标并不是执行一个完整的解析，而是将简单协议的范围进行扩展以适合于进行自动化模糊测试。作为一个应用例子，考虑下面人为生成的网络数据，这里的每一行作为一个事务，并且使用字符|将原始十六进制字节进行分隔：

```
|00 04|user|00 06|pedram|0a 0a 00 01|code rev 254|00 00 00 00 be ef|END
|00 04|user|00 04|cody|0a 0a 00 02|code rev 11|00 00 00 00 00 de ad|END
|00 04|user|00 05|aaron|0a 0a 00 03|code rev 31337|00 00 00 c0 1a|END
```

每个人（可能甚至是一只海豚）都可以很简单的对这三个事务进行浏览，并且迅速对协议的特定部分做出有根据的推测。让我们通过只研究第一个事务来看一下，我们所改进的自动分析工具能够发现什么。首先，它检测到了一个4字节的IP地址字段，并且定位于（0a 0a 00 01=私有IP地址10.10.0.1）。接下来，它又发现了4个ASCII字符串，即user，pedram，code rev 254以及END。然后扫描每个字符串前面的字节值，以发现一个一字节的有效长度值，然后是一个两字节的有效长度值，三字节的有效长度值，直到最后是一个4字节的有效长度值。现在，分析例程就可以猜测该协议的起始部分是两个可变长度的字符串，其前缀是一个两字节的长度值，其后面跟着所连接客户端的IP地址。扫描下一个字符串以发现一个长度前缀将不会产生任何结果。假定此处的下一个ASCII字段是一个静态大小的字段，引擎进行搜索并且在后面发现了4个空字节。于是猜想下一个字段应当是一个10字节的静态大小的ASCII字段。最后的字符串END不包含长度前缀，其后面也没有跟着任何内容。该字段被猜想是一个三字节的静态大小的ASCII字段。对于包含剩余两个字节的字段，我们无法做出任何假定。它可以是静态大小的也可以是可变长度的，因此需要进行进一步的分析。将所有这些字段连接在一起，引擎就可以从第一个事务中推断出如下的协议结构：

- 2字节的字符串长度值，其后面跟着字符串。
- 2字节的字符串长度值，其后面跟着字符串。

- 4字节的客户端IP地址。
- 10字节的、静态长度的、空填充的ASCII字符串。
- 静态长度或者可变长度的未知的二进制字段。
- 3字节的静态长度的ASCII字符串。

现在，引擎将接着处理后续的事务以验证在初始分析阶段所做出的猜想。通过执行这一操作，某些特定的假设被证实，而另外一些则变为无效的，整个过程始终在不断提高所建模协议的准确性。不同于其他的简单例子，这个基于启发式规则的方法对于更加复杂的协议将会执行失败。记住我们的目标不是要完整地理解被测试的协议，而是要提高缺乏帮助的模糊测试的有效性。

22.2.3 反汇编启发式规则

应用（反）汇编层次的启发式规则来帮助提高模糊器的有效性，已经成为了所有公开可用的模糊测试工具以及框架所使用的一个概念。这个概念非常简单：当进行模糊测试时，通过使用一个运行时诊断工具如调试器来监视代码在目标机上的运行情况。在调试器中，寻找静态字符串和整数比较。将这些信息传递给模糊器，以供其在以后生成测试用例时进行参考。所产生的结果将由于测试用例的不同而有所不同，然而确定无疑的是，模糊器可以通过恰当地利用这些反馈信息来生成更加智能化的数据。考虑如下的代码片段，该片段摘自于现实世界的一个服务器软件：

```
0040206C call ds:__imp__sscanf
00402072 mov eax, [esp+5DA4h+var_5CDC]
00402079 add esp, 0Ch
0040207C cmp eax, 3857106359        ; string prefix check
00402081 jz  short loc_40208D
00402083 push offset `string'       ; "access protocol error"
00402088 jmp loc_401D61
```

在sscanf() API的帮助下，原始网络数据转换为一个整数。然后，所生成的该整数将与静态整数值3857106359进行比较，如果这两个值不匹配，那么协议解析器就会由于一个“访问协议错误”而返回。当模糊器第一次遍历该代码块时，调试器构件将会发现“奇特”的值并且通过反馈回路将其传递给模糊器。然后，模糊器就可以将该值包含在一些格式中，以期望将目标应用置于一个正确的模式而暴露更多的代码覆盖。如果不使用反馈回路，那么模糊器就会失去应有的作用。

可以使用PaiMei[⊖]的逆向工程框架中的PyDbd构件，较为迅速地开发一个监视目标应用的模糊器反馈调试器的概念验证实现。可以参阅http://www.fuzzing.org网站，以获取相关的源代码。

在掌握了更加基本的一些技术之后，接下来我们将通过介绍生物信息学的概念，来讨论更加高级的自动化协议解析的方法。

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

22.3 生物信息学

Wikipedia定义了生物信息学，或者说是计算生物学的概念，它作为一个宽泛的术语概括了如下思想和技术的使用，即“应用数学、信息学、统计学、计算机科学、人工智能、化学以及生物化学通常在分子层次上来解决生物学的问题”㊀。从本质上来说，生物信息学包含不同的技术可以被用来在复杂的但具有结构化特征的长数据序列，如基因序列中发现相关的模式。网络协议同样也可以被看做是复杂的但具有结构化特征的长数据序列。因此，软件测试人员能否从生物学家那里借鉴相关的技术来简化模糊测试呢？

在生物信息学中，最基本的分析就是排列对齐两个序列的能力，不管其长度如何而只是使相似的程度最大化。在排列对齐的过程中可以在任何一个序列中插入空格以有助于实现该目标。考虑如下的氨基酸序列排列：

```
Sequence One: ACAT    TACAGGA
Sequence Two: ACATTCCTACAGGA
```

在第一个序列中插入了一些空格，强制使两个序列在长度上对齐，并且使其相似程度最大化。已经有了一些算法能够实现上述以及其他序列对齐任务。其中的一个算法如Needleman-Wunsch（NW）算法㊁，最初是由Saul Neeleman㊂和Christian Wunsch㊃在1970年提出的。NW算法通常被应用于生物信息学领域以排列对齐两个蛋白质和核苷序列。该算法是“动态编程”的一个示例，动态编程是这样一种方法，即它通过将大的问题分解为一系列小的问题来解决这个大问题。

第一次公开将生物信息学的相关理论应用于网络协议分析领域，是在2004年下半年于加利福尼亚州圣地亚哥召开的ToorCon㊄电脑黑客会议上。在该会议上，Marshall Beddoe发布了一个名为协议信息学（PI）的实验性的Python框架，引发了很大的反响，甚至发表了一篇网络文章㊅。但是从此，最初的Web站点就日益衰败，并且该框架也没有被积极地加以维护。由于PI的发布者被安全分析公司Mu Security㊆所雇佣，因此关于该技术不再公开可用的一些推测使得该技术成为了一个商业化的工具。幸运的是，经过β测试保证的PI代码的一个副本仍然可以从Packet Storm㊇处下载。

PI框架的目标是通过对大量所观测数据进行分析，来自动化推断出任意协议的域边界。已经阐明PI成功地从HTTP，ICMP和SMB协议中识别出了域。它是通过使用Smith－Waterman（SW）㊈本地序列对齐算法、NW全局序列对齐算法以及相似矩阵和进化树等方法来实现这一

㊀ http://en.wikipedia.org/wiki/Bioinformatics
㊁ http://en.wikipedia.org/wiki/Needleman-Wunsch_algorithm
㊂ http://en.wikipedia.org/wiki/Saul_Needleman
㊃ http://en.wikipedia.org/wiki/Christian_Wunsch
㊄ http://www.toorcon.org
㊅ http://www.wired.com/news/infostructure/0,1377,65191,00.html
㊆ http://www.musecurity.com
㊇ http://packetstormsecurity.org/sniffers/PI.tgz
㊈ http://en.wikipedia.org/wiki/Smith-Waterman

功能的。尽管PI框架所应用的生物信息科学的细节已经超出了本书的讨论范围，但是这里仍然对其进行了高层次的概要介绍，并且对该算法的典型应用进行了说明。

网络协议可以包含许多截然不同的消息类型。试图在不同的消息类型之间执行一个序列的对齐操作将无功而返。为了解决这个问题，首先将针对本地序列对齐的SW算法应用于一对序列以定位和匹配相似的后续序列。稍后，这些所确定的位置和所匹配的序列对将被使用以对NW全局序列对齐算法提供帮助。同时，使用相似矩阵来帮助NW算法以优化序列之间的对齐操作。PI框架使用两个常用的矩阵即接受率变异矩阵（PAM）和块置换矩阵（BLOSUM），来基于数据类型对序列进行更加恰当的对齐操作。在实际应用中，可以将该操作推广为对二进制数据与其他的二进制数据进行对齐，将ASCII数据与其他的ASCII数据进行对齐。这种特性就允许更加准确地确定协议结构内部的可变长度字段。PI框架通过执行多种序列对齐算法从而将该操作又向前推进了一步，以便于更好地理解目标协议。为了直接避开所应用的NW算法在计算中的不可行性，使用了未加权的成对算术平均数算法（UPGMA）来生成一棵进化树，然后将该进化树用于启发式规则来执行多种对齐操作。

让我们通过研究对一个简单的固定长度的协议ICMP[⊖]进行解析的功能，来分析一下该框架的实际运转情况。下面是使用PI框架来解析ICMP协议的一些基本步骤。首先，必须要收集一些ICMP包以用于分析：

```
# tcpdump -s 42 -c 100 -nl -w icmp.dump icmp
```

然后，被捕获的通信就可以通过PI框架来进行运行：

```
# ./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
0017 x08 x00 xe8 x6c x05 xbe x00 x10
0027 x08 x00 xc3 xa9 x05 xbe x00 x1a
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
```

⊖ http://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

```
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
[ output trimmed for sake of brevity ]
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 xbe x00 ???
DT   BBB ZZZ BBB BBB BBB BBB ZZZ AAA
MT   000 000 081 089 000 000 000 100
Step 3: Analyze 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
```

尽管不是立刻很明显地就可以观察到，但所生成的分析结果被转换成了如下的协议结构：

```
[ 1 byte ] [ 1 byte ] [ 2 byte ] [ 2 byte ] [ 1 byte ] [ 1 byte ]

[ 1 byte ] [ 1 byte ] [ 2 byte ] [ 2 byte ] [ 2 byte ]
```

识别最后一个字段时所发生的错误是由于被捕获以用于分析的ICMP包的受限制的编号所致。这个被错误识别的字段实际上是一个16位的序列号。由于只有100个包被使用，因此该字段的最重要的字节从未被增加过。当更多的数据被传递到PI时，该字段的大小将被正确地确定。

将生物信息学应用于自动化的协议解析是一个非常有趣并且技巧性很强的方法。该方法的应用结果在当前受到了一些限制，并且有许多研究者对应用这些技术所带来的整体收益产生了怀疑[⊖]。然而，由于PI框架已经为我们提供了成功的案例，因此我们可以期望看到该方法的进一步发展。

22.4 遗传算法

遗传算法（Genetic Algorithm,GA）是软件所使用的的一种近似搜索技术，以用来模拟进化过程。为了在后续种群中保存优秀的遗传特性，遗传算法对初始种群实施变异。它应用自然选择的规则来选择所生成的那些更加适合于环境的个体。然后，将所选择的个体进行配对和变

⊖ http://www.matasano.com/log/294/protocol-informatics/

异，接着整个处理过程再继续进行。为了定义一个遗传算法，典型地需要下面这些组成部分：

- 针对所生成个体的一种*表示方法*。
- 评价一个单独个体的优劣程度的*适应性函数*。
- 对单个个体实施变异，以及对两个个体实施交叉操作的一个*繁殖函数*。

为了更好地说明这些概念，让我们考虑一个简单的问题及其相应的遗传算法解决方案。我们将要解决的基本问题是在一个长度为10的二进制字符串中，使其中1的数量最大化。针对该问题的个体表示方法和适应性函数都是非常显而易见的。可能的个体被表示为一个由二进制数字所构成的字符串，而适应性函数则被定义为计算该字符串中1的个数的函数。对于我们的繁殖和配对函数，我们任意选择在两个字符串的位置3和位置6处交换相应的数字，并且在所生成的子代中随机改变一个位上的数字。该配对函数可能并不是最有效的配对函数，但是它能够满足我们的需求。所采用的交换规则允许父代向子代传递遗传信息，并且位值的改变也说明了随机变异操作。然后，遗传算法就依据如下步骤进行进化：

1. 首先，随机生成一个初始种群。
2. 对种群中的每个个体应用适应性函数，并且选择适应度最高的个体。
3. 对所选择的个体应用配对函数。
4. 用所生成的后代替换初始种群，然后再重复执行上述操作。

为了查看该算法的实际执行过程，让我们首先随机生成由4个二进制字符串所构成的初始种群，并且计算每个个体的适应度：

```
0100100000      2
1000001010      3
1110100111      7
0000001000      1
```

经过计算，中间的一对个体（用黑体高亮显示的）的适应度最高，因此选择这两个个体进行配对（它们是非常幸运的）。在位置3处将两个字符串进行交换，就生成了一对后代个体，在位置6处再将两个字符串进行交换，又可以得到另外一对后代个体，如下所示：

```
1000001010      3     100 + 0100111 -> 1000100111
1110100111      7     111 + 0001010 -> 1110001010

1000001010      3     100000 + 0111 -> 1000000111
1110100111      7     111010 + 1010 -> 1110101010
```

然后，对所生成的后代个体实施一个随机的变异操作（用黑体高亮显示），并且再次应用适应性函数：

```
1000100111  ->  1010100111      6
1110001010  ->  1110000010      4
1000000111  ->  1000001111      5
1110101010  ->  1110101110      7
```

我们可以看到遗传算法的执行获得了成功，因为新生成种群的平均适应度得到了提高。在这个特定的例子中，我们使用了一个静态的变异率。而在一个更加高级的例子中，如果所生成

的后代在一定的时间限制内没有完成进化，我们可能将会选择自动化来提高变异率。注意，遗传算法被认为是一种随机的全局最优化算法。换句话说，在该算法中存在着一定的随机因素，因此它所输出的结果经常会发生变化。然而，尽管遗传算法将会持续执行以找到最佳的个体，但是无论执行多少时间，它都可能不会找到最佳的可能个体（在我们的例子中就是一个全1的字符串）。

应用遗传算法以提高模糊测试的性能，是佛罗里达中央大学（UCF）的一个团队近期所研究的一个课题，相关的研究成果被发表在2006年的美国安全会议BlackHat上[⊖]。UCF团队发布了一个名为Sidewinder的概念验证工具，该工具能够自动生成输入以强制执行一个设计好的执行路径。在这种情形下，遗传算法的作用是生成被表示为上下文无关文法的模糊数据[⊜]。一个非传统的模糊测试方法被应用以定义一个适应性函数。该工具并不是首先生成数据并监视错误的发生，而是静态地定位潜在的漏洞代码位置如不安全的API调用（例如strcpy）。此过程中的这一步骤类似于前面章节所讨论的Autodafé定位代码点以将增加的权重应用于标记的过程。接下来，对整个目标二进制代码的控制流程图（CFG，不要同上下文无关的文法相混淆）进行检查，并且抽取出包数据入口点（对recv的调用）和潜在的漏洞代码位置之间的子流程图。图22-5描述了一个包含所有这些点的CFG示例。关于CFG更加详细的定义和解释，请参见第23章“模糊器跟踪”。

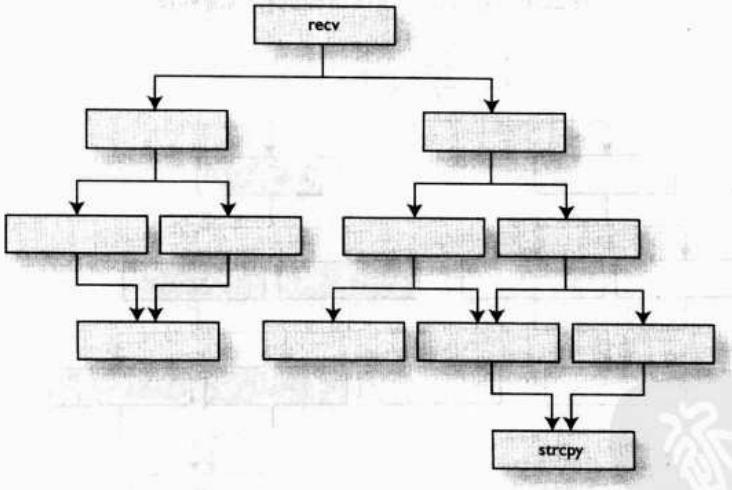


图22-5 包含潜在漏洞的控制流程图

在下一步骤中，必须要识别出连接入口点和目标漏洞代码的所有路径上的节点。图22-6显示了相同的CFG，并且用黑色框突出显示了连接路径上的节点。

接下来，同一个CFG中的退出节点被确定。一个退出节点被定义为在连接路径外面的边界

⊖ http://www.blackhat.com/html/bh-usa-06/bh-usa-06-speakers.html#Embleton

⊜ http://en.wikipedia.org/wiki/Context-free_grammar

节点。如果到达了一个退出节点，那么执行就会从任何可能的路径转向漏洞代码。图22-7显示了同前面相同的一个CFG，用黑色框突出显示了连接路径上的节点。另外，用白色框突出显示了退出节点。

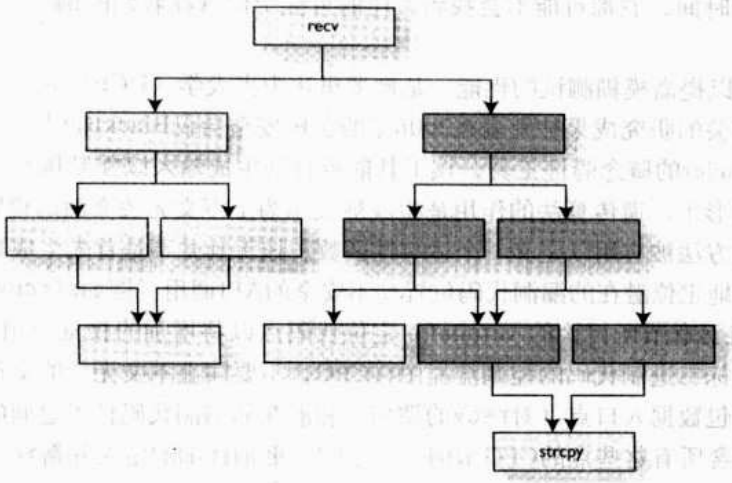


图22-6　突出显示连接路径的控制流程图

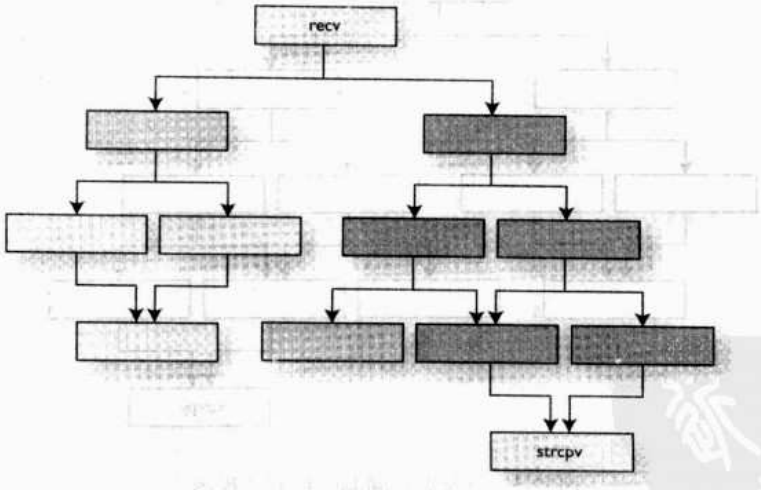


图22-7　突出显示退出节点的控制流程图

根据目前突出显示的CFG，我们就可以定义一个适应性函数。UCF研究团队对所生成的CFG应用了一个马尔可夫过程，以基于遍历特定路径的概率来计算适应度。在实际应用以及更加简单的一些条件下，可以采用如下的步骤。使用一个调试器关联到目标进程并且在入口节点、目标节点、退出节点以及连接路径上所有的节点上设置断点。当到达这些断点时对它们进行监视，以评估针对一个给定输入（个体）的运行时执行路径的进度。然后跟踪执行过

程，直到到达一个退出节点。模糊器将生成输入，并且将其提供给目标进程。选择那些适应度最高的输入来进行繁殖，然后继续执行整个过程，直到成功到达一个目标节点，即表明一个潜在漏洞的节点。

将Sidewinder与静态分析技术、图论理论以及调试器运行时的分析方法相结合，就能够从本质上强制提供一个输入从而可以到达目标进程中的任意位置。尽管这是一个功能非常强大的概念，但是目前仍然存在着许多局限性使其还不是一个很完美的解决方案。首先，并不是所有的CFG结构都适合于采用遗传算法来进行处理。特别的，在图形结构和数据解析之间必须要存在某些依赖关系。其次，通过静态分析的方法来抽取出准确的CFG并不总是可行的。一个不准确抽取出的CFG可能会导致整个处理过程执行失败。最后，对于包含TVL类型字段的协议而言，该方法将会花费过长的时间来生成有效的数据。例如，当对一个包含所计算的CRC值的协议进行模糊测试时，在遗传算法生成任何有用的结果之前，我们的太阳将可能会成为超新星。

总而言之，对该技术进行研究是非常有趣的，并且在某些情况下也会给我们带来一些帮助。如果需要的话，Sidewinder的最佳使用场合可能就是帮助一个模糊器来对输入进行变异，以遍历一个小的子路径来增加代码覆盖率，而不是发现一个能够对从入口到目标节点的整个连接路径进行遍历的输入。

22.5 小结

在本章的开始部分，首先说明了模糊测试最困难的方面就是要克服对目标协议或文件格式进行理解和建模的一些入门障碍。本章提出了三种常规的方法，以有助于手工或者自动化进行协议解析。分别描述了启发式技术、生物信息学方法以及遗传算法，并将它们作为了可能的解决方案。本章中所提出的这些概念是模糊测试研究领域的前沿问题，并且每天都会有一些新的发展。为了得到更多的新信息和资源，可以访问http://www.fuzzing.org站点。

第23章 模糊器跟踪

“好的，如果你说你准备去做某事但并没有做，那么我认为这是值得信赖的。”

——George W. Bush,在美国有线新闻的在线访谈，2000年8月30日

在前面的章节中，我们已经定义了模糊测试，指明了模糊测试的目标，枚举了不同的模糊测试类别，并且讨论了数据生成的不同方法。然而，我们还没有讨论对模糊器技术的执行进度进行跟踪的相关内容。在有关安全的研究领域中，模糊器跟踪仍然没有受到足够的关注。据我们所知，当前所有可用的商业或者免费模糊器都没有解决这一问题。

在本章中，我们定义模糊器跟踪技术，该技术也被称为代码覆盖。我们将讨论其所具有的优越性，研究不同的实现方法，当然，在本书剩余部分中，我们将构造一个功能性的原型，该原型可以同我们前面所开发的模糊器结合在一起，从而提供更加强大的分析功能。

23.1 我们究竟想要跟踪什么

在最低的编程层次上，是CPU负责执行汇编指令，或者是执行它可以理解的其他命令。可用的指令集将依据目标机器的底层体系结构的不同而有所不同。例如，我们所熟知的IA-32/86体系结构，就是一个复杂指令集计算机体系结构（Complex Instruction Set Computer,CISC）[⊖]，它包含多于500条的指令，当然也具备执行基本的数学运算和内存操纵的能力。可以将这种体系结构与采用精简指令集计算机体系结构（Reduced Instruction Set Computer,RISC）的微处理器如SPARC相比较，SPARC所包含的指令要少于200条[⊜]。

CISC同RISC的比较

术语CISC实际上是一个逆向的定义，用来与RISC的概念设计相区别，RISC是IBM的研究人员于1970后期所开发的一种体系结构。两者在设计理念的主要区别是，RISC处理器可以被证明运行的更快，并且生产该处理器的花销要更少，而CISC处理器则会更少的调用主内存，并且使用相对更少的代码就可以完成同样的任务。关于哪种体系结构更加优秀的争论随着最近狂热的苹果机Mac用户的增加而达到了一个新的高度，直到现在他们的应用程序只能在PowerPC的RISC处理器上运行。

有趣的是，尽管在x86体系结构上有大量可用的复杂指令，差不多有一打指令的运行时间要多于平均的执行时间[⊝]。

⊖ http://www.intel.com/intelpress/chapter-scientific.pdf

⊜ 关于这些体系框架所提供的准确的指令数目前还存在着争议，但有一个通常的看法。CISC体系框架所实现的指令数要多于RISC。

⊝ http://www.openrce.org/blog/view/575

不论你是基于CISC还是RISC体系结构，实际的单独指令已经很少被程序员来人工编写，而是从一个高级语言如C或者C++被编译或者解释到低级语言。通常情况下，你所编写的普通规模的应用程序将包含从几万条到上百万条指令。同软件所进行的每一次交互都将导致一条指令的执行。下面这些简单的动作，如点击鼠标、处理一个网络请求以及打开一个文件都需要大量的指令被执行。

如果我们能够让一个Voice over IP（VoIP）软件电话（softphone）在模糊审核的过程中发生崩溃，那么如何来知道是哪些指令集导致了该错误的发生呢？目前研究者所采用的典型方法就是进行回溯操作，通过检查日志文件并且关联到一个调试器来定位错误发生的位置。另外一个可选的方法是向前操作，通过动态监视哪些指令被执行以响应所生成的模糊请求。有一些调试器如OllyDbg，实现了一种跟踪的功能以允许用户来跟踪所有已经执行的指令。不幸的是，这些跟踪功能通常要花费很多的资源，并且需要大量的人力，因此其实用性不强。继续讨论我们的例子，当VoIP软件崩溃时，我们可以评审已经执行过的指令以确定错误所发生的位置。

当编写软件时，对其进行调试的一种最基本的形式就是在代码中插入输出语句。采用这种方法，当程序运行时，开发者将关注于执行所发生的位置以及不同逻辑块的执行顺序。从本质上来说，我们想要实现同一个功能，但是在一个更低的层次上，我们将无法在代码中任意插入输出语句。

我们把在处理不同请求的过程中，对所执行的指令进行监视和记录的过程定义为“模糊跟踪”。你可能早已经所熟知的针对此过程的一个更加通用的术语称为代码覆盖，这两个术语将在本章中被交替使用。我们很快将会看到模糊器跟踪是一个相当简单的过程，它为我们提供了许多机会以提高模糊分析的水平。

23.2 二进制代码可视化和基本块

在我们关于代码覆盖的讨论中，一个核心概念就是对二进制代码可视化的理解，这一功能将在模糊器跟踪器原型的设计和开发中被实现。反汇编的二进制代码可以通过图形来可视化表示，该图形称之为调用图。这一功能是通过将每个函数表示为一个节点，而将函数之间的调用表示为节点之间的边来实现的。这是一种非常有用的抽象方法，用以查看整个二进制代码的结构以及不同函数之间的调用关系。考虑一个假想的例子，如图23-1所示。

图23-1中的大多数节点都包含一个标签，该标签的前缀是sub_，其后面跟着一个由8位字符构成的16进制数字，该数字表示子例程在内存中的地址。这种命名规则被反汇编器DataRescue Interactive Disassembler Pro（IDA Pro）[⊖]所使用，以在不知道符号信息的情况下自动为函数生成名字。其他反汇编引擎也使用相类似的命名模式。当对一个不开源的二进制应用程序进行分析时，缺乏符号信息将导致大多数的函数采用这种形式来命名。图23-1中有两个节点具有符号名，并且分别被恰当地命名为`snprintf()`和`recv()`。通过对此调用图进行快速浏览，我们就可以根据未命名的子例程`sub_00000110()`和`sub_00000330()`关于`recv()`之间的关系，来立刻确

⊖ http://www.datarescue.com

定这两子例程的功能是负责处理网络数据。

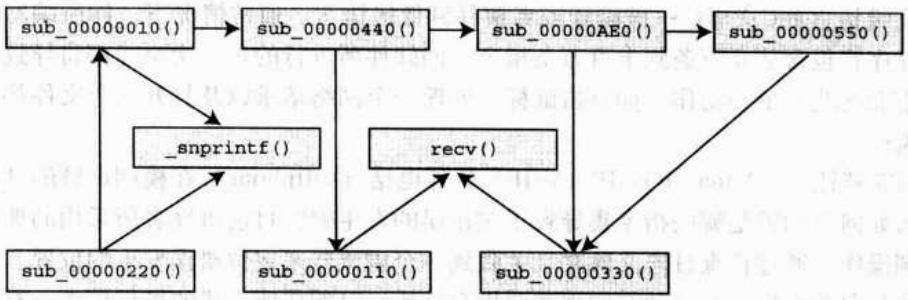


图23-1　简单的调用图示例

23.2.1　CFG

反汇编函数也可以通过图形来可视化表示，该图形称之为CFG。这是通过将每个基本块表示为一个节点，而将基本块之间的边表示为节点之间的边来实现的。这很自然地引出了一个问题，即什么是基本块？一个基本块被定义为指令的一个序列，并且一旦到达了基本块的第一条指令，那么该块内部的每条指令都确保能够按次序运行。为了进一步加以说明，一个基本块通常以下列条件作为起始点：

- 一个函数的开始点
- 一条分支指令的目标
- 一条分支指令的下一条指令

基本块通常以下列条件作为终止点：

- 一条分支指令
- 一条返回指令

以上只是关于基本块的起始点和终止点的一个简单列表。一个完整的列表将会考虑到对无返回值函数的调用以及对异常处理函数的调用。然而对于我们这里要到达的目标而言，上述定义和列表也就足够了。

23.2.2　CFG示例

为了更好地理解这一点，可以考虑下面这个来自于一个假想的子例程sub_00000010的代码片段。图23-2中的[el]序列代表任意数量的非分支指令。一个反汇编函数的这个视图也被称之为字串列表，图23-2表示了字串列表的一个例子。



图23-2　子例程sub_00000010的反汇编字串列表

按照前面所描述的规则，将字串列表划分为基本块是一项非常简单的工作。第一个基本块的起始点位于0x00000010的最顶层函数，其终止点位于0x00000025的第一个分支指令。位于0x0000002B的分支指令的下一条指令，同分支指令的跳

转目标0x00000050一样，各自都标识了一个新的基本块的起始点。一旦被分解为其基本块，图23-2中所显示的子例程的CFG将变为图23-3的样子。

这是一种非常有用的抽象方法，可以查看经过一个函数的不同的控制流路径，同时也可以识别出逻辑循环。关于CFG需要记住的一个关键点是每一个指令块都被确保一起执行。换句话说，如果该块中的第一条指令被执行，那么该块中的所有指令都将被执行。稍后，我们将会看到该方法如何能够帮助我们在模糊器跟踪器的开发过程中采取一些必要的捷径。

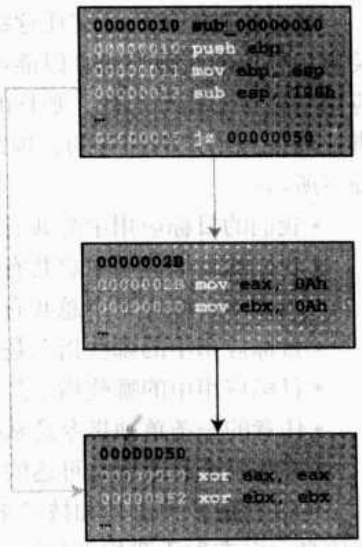


图23-3 子例程sub_00000010的控制流程图

23.3 构造一个模糊器跟踪器

我们可以采用许多方法来实现一个定制的代码覆盖工具。如果你已经阅读了本书的前面的章节，可能脑海中所浮现的第一个方法就是使用一个带有跟踪功能的调试器以对所执行的每一条指令进行监视和记录日志。这也正是OllyDbg所采用的方法，用来实现其“调试/跟踪进入”和“调试/跟踪退出”功能。我们可以依赖于在第20章“内存数据的模糊测试：自动化”中所使用的相同的PyDbg库，并且使用如下的逻辑来再现此功能：

1. 关联到或者加载我们正在进行模糊测试的目标进程。
2. 注册一个单独的单步处理器，并且设置单独的单步标志。
3. 在单独的单步处理器内部，对当前指令的地址进行记录日志，并且再次设置单独的单步标志。
4. 继续执行直到该进程退出或者发生崩溃。

在PyDbg的基础上，可以只用不到50行的Python指令就可以实现这个简单的方法。然而，它并不是一个令我们满意的方法，因为通过一个调试器在进程内部对每一条指令所进行的监视将会增加大量的延迟，而该延迟在许多情况下都将使该方法变为不可用。如果你不相信这一点，那么可以使用工具条上所列出的相关代码进行一下试验，该代码在网站http://www.fuzzing.org上是可用的。另外，简单确定并记录所执行的指令只是覆盖了整个目标的一个方面。为了生成更好的报告，我们需要同目标相关的更多的信息。

我们使用一个三阶段的方法来实现模糊器跟踪器：

- 首先，我们要静态地对目标特征进行刻画。
- 第二个步骤是对目标内部的、为响应模糊器而执行的那些指令进行动态地跟踪和记录。
- 第三个也是最后一个步骤是对所生成的动态数据连同在步骤1中所生成的特征数据进行交叉引用，以生成一个实用的跟踪报告。

23.3.1 刻画目标特征

假定我们能够确定在任意给定的应用程序内部有哪些指令被执行，那么我们还可以知道有关应用程序的其他信息，以能够更快捷、更好地理解数据集呢？例如，至少我们想要知道目标应用所包含的指令总数。那样的话，我们就可以确定在目标应用内部有百分之多少的代码是负责处理任意给定的任务的。理想的情况下，我们也想知道关于指令集的更加详细的一些信息，如下所示：

- 我们的目标应用中总共有多少条指令？
- 我们的目标应用中总共有多少个函数？
- 我们的目标应用中总共有多少个基本块？
- 目标应用中的哪些指令是与标准的支持库相一致的？
- 目标应用中的哪些指令是我们的开发者所编写的？
- 任意的一条单独指令是从哪些源代码行中被转换而来的？
- 哪些指令和数据是可达的，并且可以通过网络和文件数据而被影响？

可以利用多种工具和技术来回答上面这些问题。大多数的开发环境都能够生成符号的调试器信息，以有助于我们更加准确地解决这些问题。对于我们这里要实现的目标而言，我们假定不能够很轻易地获得源代码。

为了提供针对这些核心问题的答案，我们基于IDA Pro的反汇编器来构造我们的特征描述器。通过IDA，我们可以非常容易地收集有关指令和函数的统计信息。通过基于基本块的规则来实现一个简单的算法，我们同样可以非常容易地抽取出包含在每个函数内部的所有基本块的一个列表。

23.3.2 跟踪

由于对一个完整的程序进行单步执行是非常耗费时间的，并且也是不必要的，那么还可以采用其他的什么方法能够跟踪指令的覆盖情况呢？回忆一下我们在前面对基本块所做的定义，即基本块就是若干条指令的一个序列，该块中的每一条指令都被保证按顺序执行。如果我们可以跟踪基本块的执行，那么我们实际上就是跟踪了指令的执行。我们可以借助于一个调试器以及在刻画特征步骤中所收集到的信息来完成这一功能。利用PyDbg库，我们就可以通过如下的逻辑来实现此概念：

1. 关联到或者加载我们正在进行模糊测试的目标进程。
2. 注册一个断点处理器。
3. 在每个基本块的起始位置设置断点标志。
4. 在断点处理器内部，将当前指令的地址进行记录日志，并且可选的恢复断点。
5. 继续执行直到该进程退出或者发生崩溃。

通过记录基本块的执行而不是单独指令的执行，我们就可以极大地提高传统的跟踪功能如前面所提到的OllyDbg所使用的受限制的方法的性能。如果我们只对哪些基本块被执行感兴趣，那么就不需要在上面的步骤4中对断点进行恢复。一旦执行到了一个基本块，我们将在处理器

中记录下这个事实，并且我们不再关心知道以后是否还能再次到达该基本块，以及何时到达该基本块。然而，如果我们同样也对基本块的执行次序感兴趣，那么就必须在上面的步骤4中对断点进行恢复。如果反复到达了同一个断点，例如可能是在一个循环内部，那么我们将会记录下每一次循环到达该断点的情况。恢复断点为我们提供了更多的上下文信息。使断点失效就可以提高执行的速度。究竟采用哪一种方法主要是取决于你所面临的任务，因此应当在我们的工具中将其作为一个选项来设置。

在这样的情形下，我们正在对数量很大的基本块进行跟踪，跟踪速度可能会同预想的不一致。在这种情况下，可以通过对所执行的函数进行跟踪，而不是对所执行的基本块进行跟踪来从指令层次上进一步抽取出更多的信息。这就需要对原有的处理逻辑进行一个细微改动。只对启动函数的那些基本块设置断点，而不是对所有的基本块都设置断点。同样，究竟采用哪一种方法主要是取决于你所要完成的功能，因此应当在我们的工具中也将其作为一个选项来设置。

简化的PyDbg单步代码覆盖工具

针对那些感兴趣的读者，下面的PyDbg脚本说明了一个单步代码执行器的实现。在初始的关联以及断点设置阶段，该脚本将每一个已经存在的线程置于单步执行模式。注册一个处理器以创建新的线程，并且同样将这些线程也设置为单步模式。

在PyDbg的单步代码执行器的开发过程中所使用的唯一一个技巧就是在内核中对上下文转换进行处理。在每一个单步执行过程中，该脚本要检查当前的指令是否是sysenter。该指令是现代微软的Windows操作系统所使用的进入内核的通路。当遇到sysenter指令时，在内核将要返回的地址处设置一个断点，并且线程被允许正常执行（也就是说，不是在单步模式下执行）。一旦线程从内核中返回，那么将恢复单步执行机制。相关的代码如下所示。

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

# breakpoint handler.
def on_bp (dbg):
    ea      = dbg.exception_address
    disasm = dbg.disasm(ea)

    # put every thread in single step mode.
    if dbg.first_breakpoint:
        for tid in dbg.enumerate_threads():
            handle = 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

# single step handler.
def on_ss (dbg):
    ea      = dbg.exception_address
    disasm = dbg.disasm(ea)
```

```
        print "%08x: %s" % (ea, disasm)

        # we can't single step into kernel space
        # so set a breakpoint on the return and continue
        if disasm == "sysenter":
            ret_addr = dbg.get_arg(0)
            dbg.bp_set(ret_addr)
        else:
            dbg.single_step(True)

        return DBG_CONTINUE

    # create thread handler.
    def on_ct (dbg):
        # put newly created threads in single step mode.
        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__()
```

在我们的模糊器跟踪工具中所要实现的另外一个便捷的特性就是能够从新的记录中过滤出以前所记录的代码覆盖信息。可以利用该特性来非常容易地过滤出函数、基本块以及处理我们不感兴趣的任务的那些指令。不感兴趣代码的一个例子是那些负责处理GUI事件（点击鼠标以及选择菜单等）的代码。稍后，我们将会看到这个特性如何在代码覆盖中发挥作用以发现漏洞。

23.3.3 交叉引用

在第三个即最后一个步骤中，我们将静态特征刻画所收集到的信息同运行时跟踪器的输出信息相结合，来生成准确的有用的报告，以解决如下的一些问题：

- 哪些区域的代码是负责处理我们的输入的？
- 我们覆盖了百分之多少的目标代码？
- 导致一个特定错误发生的执行路径是什么？
- 在目标代码内部，有哪些逻辑决策没有执行？
- 在目标代码内部，有哪些逻辑决策直接依赖于所提供的数据？

上面列表中的后两个问题实际上是要做出更好的决策，以提高我们的模糊器的性能，而不是要在实际的目标应用中定位任何的特定缺陷。当我们在本章的后面讨论模糊器跟踪技术可以

为我们提供的多种优越性时，将会对上面这些问题进行深入分析。

针对漏洞检测的代码覆盖

2005年6月14日，发布了一个可以远程利用的缓冲区溢出，该溢出在对网络新闻传输协议(NNTP)的响应[⊖]进行解析时，将会影响微软的Outlook Express。当一个漏洞攻击的研究者应用代码覆盖分析时，让我们来研究一下它所提供的帮助。微软所发布的建议的核心内容如下：

当Outlook Express被用做一个新闻组的阅读器时，就会存在一个远程代码执行漏洞。一个攻击者可以通过如下的方法来利用该漏洞，即创建一个恶意的新闻组服务器，并且当用户查询服务器以获取新闻时潜在地允许远程代码执行。成功利用该漏洞的攻击者可以对一个受影响的系统进行完整的控制。然而，对该漏洞的利用需要进行用户的交互。

并没有太多的信息以供参考，例如对于IDS或者IPS签名的开发者而言，肯定也是非常不够用的。我们所知道的就是当将Outlook Express连接到一个恶意的NNTP服务器时，出现了这些错误。利用前面所提到的一个可用的、开源的代码覆盖工具Process Stalker[⊜]，让我们来进行实验并且查看它是否能够在某种程度上解决该问题。特定的命令行选项以及Process Stalker的使用已经超出了本书的讨论范围。为了查看使用该例子之后的一些特定的细节信息，可以参阅原版的OpenRCE文章[⊜]。

通过安装可用的补丁，并且监视Outlook Express安装路径内部的哪些文件被修改了，我们就可以确定肯定是MSOE.DLL包含漏洞代码。使用Process Stalker IDA Pro插件对该模块进行分析，就可以发现大约有4800个函数以及58 000个基本块。手工对这些数据进行审查过滤将会是非常繁琐的一项工作。为了缩减我们感兴趣的区域，可以利用Process Stalker来定位与将Outlook Express连接到一个任意的NNTP服务器相关的代码路径。尽管这样肯定会有效减少问题的分析空间，但是还可以采用更好的方法。回忆一下前面所讨论的，同我们的目标应用依次所进行的每一次交互，都将导致指令的执行。在目前的情形下，这就意味着当我们在Internet Explorer中通过'news://'URI句柄来访问一个恶意的NNTP服务器时，代码覆盖工具将没有必要记录启动Outlook Express执行的那些代码，绘制屏幕控件如窗口和对话框的那些代码，以及负责处理用户驱动GUI事件如点击按钮的那些代码。例如，当第一次连接到一个NNTP服务器时，将会显示图23-4所示的对话框。

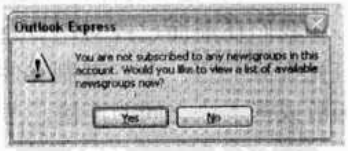


图23-4 对话框

对该对话框进行创建、处理以及销毁都远远超出了我们所关心的范围。为了进一步缩减我们的分析空间，可以使用Process Stalker来只记录那些负责处理不同GUI事件的代码。我们通过关联到并且同Outlook Express进行交互来实现这一功能，而不是实际连接到一个NNTP服务器。接下来，我们将使用只面向GUI事件以及连接到服务器这两种方法所得到的记录信息中那些重叠的覆盖记

⊖ http://www.microsoft.com/technet/security/bulletin/MS05-030.mspx

⊜ http://www.openrce.org/downloads/details/171/Process_Stalker

⊜ http://www.openrce.org/articles/full_view/12

录进行删除，以特别定位那些负责解析NNTP服务器请求的代码。当执行所有这些操作后，我们就可以将分析范围减少到91个函数。在涉及这91个函数的1 337个基本块中（我们保证这个数字并不是虚构的），实际上只有747个被服务器连接这种记录方法所访问。和开始时的58 000个基本块相比，这种分析范围的缩减是非常显著的，具有很大的帮助作用。

应用一个简单的脚本以定位那些负责移动服务器所提供的数据的基本块，将会进一步将其数量减少到极少的26个基本块，其中的第二个基本块发现了该漏洞准确的细节信息：通过空格来分隔的任意数量的服务器所提供的数据，被复制到了一个静态的16字节的栈缓冲区中。非常好，我们现在就可以对漏洞保护过滤器进行利用，并且及时回到家里以聆听最新的女高音的歌曲[⊖]。

23.4　对一个代码覆盖工具的分析

我们前面所使用的PyDbg库，实际上是一个更大的逆向工程框架的一个子构件，该框架的名称是PaiMei。PaiMei是一个动态开发的、开源的Python逆向工程框架，并且可以免费从网站上下载[⊜]。PaiMei在本质上可以被认为是逆向工程领域的“瑞士军刀”，它可以用来为多种高级的模糊测试任务提供帮助，例如我们在第24章“智能化错误检测”所深入讨论的智能化异常检测和处理问题。已经存在一个同该框架绑定在一起的代码覆盖工具，其名称为PAIMEIpstakler（也被称为Process Stakler或者PStalker）。尽管该工具可能不会生成你想要的准确的报告，但是由于该框架和所绑定的应用都是开源的，因此它们可以被修改以满足针对一个给定项目的任何特定的需求。可以将该框架分解为如下的核心组成部分：

- *PyDbg*　一个纯Python编写的win32调试抽象类。
- *pGRAPH*　一个图形抽象层，并且针对节点、边以及二者的组合都具有单独的类。
- *PIDA*　PIDA基于pGRAPH，其目标是针对二进制文件（DLL和EXE）提供一个抽象的持久化的接口，并且具有单独的类以表示函数、基本块以及指令。其最终的结果是创建一个可移植的文件，并且当你加载该文件时，可以任意遍历整个原始的二进制文件。

我们在前面已经对PyDbg的接口有所了解。基于PIDA来编写脚本将会更加容易。PIDA接口允许我们将一个二进制文件作为图形来进行遍历，将节点枚举为函数或者基本块，将边作为函数或基本块之间的连接。考虑下面的例子：

```
import pida

module = pida.load("target.exe.pida")

# step through each function in the module.
for func module.functions.values():
    print "%08x - %s" % (func.ea_start, func.name)

    # step through each basic block in the function.
```

⊖ http://www.hbo.com/sopranos/

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

```
        for bb in func.basic_blocks.values():
            print "\t%08x" % bb.ea_start

            # step through each instruction in the
            # basic block.
            for ins in bb.instructions.values():
                print "\t\t%08x %s" % (ins.ea, ins.disasm)
```

在该代码片段的开始部分，我们首先将前面所分析并保存的一个PIDA文件作为脚本中的一个模块来进行加载。接下来，我们对该模块中的每一个函数进行遍历，打印出函数的起始地址，并且同时输出符号名。接着，我们对包含在当前函数中的每一个基本块进行遍历。对于每一个基本块而言，我们进一步遍历它所包含的指令，并且输出指令地址以及反汇编信息。基于我们前面所指出的，你可能会非常明显地想到，如何将PyDbg和PIDA组合起来以简单地实现一个基本的块层次的代码覆盖工具的原型。在核心构件的上一个层次中，你将会发现PaiMei框架的剩余部分被分解为如下的组成部分：

- 工具。完成多种重复性工作的一组工具集。这里有一个我们所感兴趣的特殊的工具类process_stalker.py，该类用于提供针对代码覆盖功能的一个抽象的接口。
- 控制台。一个具有插件性质的WxPython应用，以快速有效地创建针对新创建工具的定制的GUI接口。我们就是通过控制台与pstalker代码覆盖模块进行交互的。
- 脚本。完成不同任务的单独的脚本，其中的一个非常重要的例子是IDA Python 脚本pida_dump.py，该脚本运行于IDA Pro以生成.PIDA文件。

pstalker代码覆盖工具是通过前面所提及的WxPython GUI来提供接口的，同时我们很快将会看到，该工具依赖于PaiMei框架所提供的许多构件。当对该代码覆盖工具不同的子构件进行了研究之后，我们将通过一个实例来研究其用法。

23.4.1 PStalker设计概览

PStalker模块是在PaiMei的图形化控制台中可用的许多模块之一。在图23-5所示的PStalker的初始界面中，你可以看到该接口被分解为三个截然不同的栏。

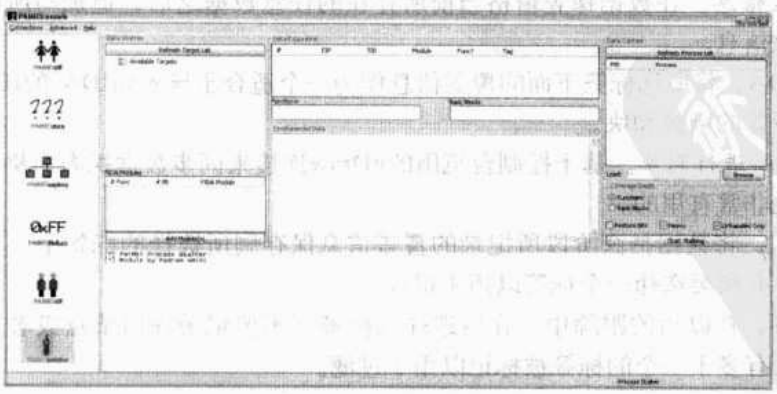


图23-5 PaiMei PStalker模块

数据源这一栏提供了必要的功能以创建和管理将要被分析的目标，并且同时加载适当的PIDA模块。数据探查栏中所显示的是任意的单独代码覆盖运行的结果。最后，数据捕获栏是用于指定运行时配置选项以及目标可执行信息的。同该工具相关联的通常的工作流如下所示(注意理解其中的细节信息)：

- 创建一个新的目标、标签或者两者都创建。目标可以包含多个标签，并且每一个单独的标签都包含其自己所保存的代码覆盖数据。
- 选择一个标签以用于跟踪。
- 针对那些需要覆盖信息的每一个.DLL和.EXE代码加载PIDA模块。
- 启动目标进程。
- 在数据捕获窗格中刷新进程列表，并且选择将要关联到的目标进程，或者浏览将要加载的目标进程。
- 配置代码覆盖选项，并且关联到所选择的目标进程。
- 当代码覆盖信息被记录时，同目标进程进行交互，
- 在适当的时机，同目标进程相分离并且等待PStalker将所捕获的条目导出到数据库。
- 加载所记录的覆盖信息以开始研究所记录的数据。

针对PaiMei的在线文档[⊖]包括一段完整的录像，以说明PStalker模块的用法。如果你在这之前从来没有看过这一说明，那么在继续讨论之前最好先对其有所了解。

23.4.2 数据源

一旦通过连接菜单建立起控制台范围内的MySQL连接，那么就点击检索目标列表按钮来展开数据源浏览树。应用程序第一次运行时，该列表将会是空的。可以从标记为可用目标的根节点的上下文菜单中创建新的目标。使用针对单个目标的上下文菜单来删除单个的目标，在目标下面增加一个标签，并且从目标下的所有标签中将覆盖信息加载到数据探查窗格中。针对单个标签的上下文菜单所提供的特性如下所示：

- 加载覆盖信息。清空数据探查窗格，并且将与所选择的标签相关联的覆盖信息进行加载。
- 追加覆盖信息。在数据探查窗格当前所存在的任意数据之后，追加与所选择的标签相关联的覆盖信息。
- 导出到IDA。将所选标签下面的覆盖信息作为一个适合于导入到IDA 的IDC脚本导出，以强调所覆盖的函数和块。
- 使用uDraw进行同步。基于控制台范围的uDraw连接来同步数据探查窗格。双监视器对于此特性是非常有用的。
- 用于跟踪。将数据捕获阶段所记录的覆盖信息保存到所选择的标签中。在关联到一个进程之前，必须要选择一个标签以用于跟踪。
- 过滤标签。在以后的跟踪中，在所选择的标签下不要记录任何节点或者基本块的覆盖信息。可以有多于一个的标签被标记以用于过滤。

⊖ http://pedram.openrce.org/PaiMei/docs/PAIMEIpstalker_flash_demo/

- 清除标签。保留标签，而删除该标签下面所存储的所有覆盖信息。PStalker不能使用以前所存在的任何数据来跟踪一个标签。在重写一个标签的内容之前，必须首先将其清空。
- 扩展标签。对于标签中的每一个碰击函数（hit function），为每一个所包含的基本块生成并添加一个入口，即使是该基本块并没有被显式地覆盖。
- 目标/标签属性。修改标签的名字，查看或者编辑标签的注释，或者是修改目标的名字。
- 删除标签。删除标签以及存储在标签下面的所有的覆盖信息。

PIDA模块列表控件显示了所加载的PIDA列表，同时也显示了其单个的函数以及基本块的个数。在关联到目标进程之前，至少有一个PIDA模块必须要被加载。在代码覆盖的过程中，PIDA模块被引用以在函数或者基本块中设置断点。可以使用上下文菜单来将一个PIDA模块从列表中删除。

23.4.3 数据探查

数据探查窗格被划分为三个部分。当覆盖信息从数据源窗格中被加载或者追加时，第一个部分将被展开。每一个覆盖信息被显示在一个可以滚动的列表框中。选择其中的一行条目，如果该条目可用的话，那么就会在解除引用数据的文本框中显示与该覆盖信息相关联的上下文数据。在列表框和文本框控件之间有两个进度条，它基于对数据探查窗格中所列出的基本块和函数的单独覆盖次数，而不是被加载的所有PIDA模块所包含的基本块和函数的总数，来表示总的代码覆盖。

23.4.4 数据捕获

一旦一个标签被选择以用于跟踪，所选择的过滤器被应用，并且目标PIDA模块也被加载，那么下一个步骤就是实际关联并且跟踪目标进程。

点击检索列表按钮以显示一个更新的进程列表，在其底部列出的是最新的进程。选择将要关联到的目标进程，或者浏览目标进程以加载和选择所需要的覆盖深度。选择在目标进程内部执行的函数以用于监视和跟踪，或者选择基本块以进一步仔细查看所执行的基本块。

Restore BPs复选框控制着在覆盖了断点之后，是否要将其进行恢复。为了确定在目标内部有哪些代码被执行，可以将该选项设置为不可用。另外，为了同时确定哪些代码被执行以及这些代码的执行次序，那么应当将此选项设置为可用的。

Heavy复选框控制着PStalker是否将要保存每一个所记录的覆盖信息的上下文数据。可以将该选项设置为不可用以提高执行速度。不选中Unhandler Only复选框以接收调试异常事件的通知，即使是调试器能够对这些事件进行正确处理。

最后，当设置了所有的选项并且选择了目标进程之后，点击关联并且启动跟踪按钮以开始执行代码覆盖。可以查看日志窗口以获取运行时信息。

23.4.5 局限性

你应当意识到该工具存在着一个主要的局限性。特别的，该工具依赖于对DataRescue的IDA Pro所提供的目标二进制代码的静态分析，因此当IDA Pro出现错误时，该工具将会执行失

败。例如，如果数据被错误地表示为代码，那么当PStalker针对该数据设置断点时，那么该工具的执行将会出现问题。这是该工具执行失败的一个最常见的原因。在自动分析之前将IDA Pro中的核心选项——进行最后的通过性分析选项设置为不可用，将会使造成这些类型错误的恶意代码变为不可用。然而，这也会影响到整个的分析。另外，目前还不能对包装的或自修改的代码进行跟踪。

23.4.6 数据存储

一般的用户可能并不需要知道在PStalker代码覆盖工具内部，数据是如何被组织以及存储的。这些用户可以直接利用该工具所提供的抽象功能。然而，如果需要构建定制的扩展或者提供内容更加丰富的报告，那么就需要知道关于数据库模式更加详细的知识。因此，这一部分将致力于对该工具内部所使用的数据存储机制进行分析。

所有的目标和代码覆盖数据都被保存到一个MySQL数据库服务器中。这些信息是通过三个表来存储的：cc_hits,cc_tags以及cc_targets。如下面的SQL表结构所示，cc_targets数据库包含一个目标名字的简单列表，一个自动生成的唯一的数字标识符，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;
```

下面所列出的是cc_tags数据库的SQL表结构，它包含一个自动生成的唯一的数字标识符，该标签所关联到的目标的标识符，该标签的名字，以及一个用于存储与该标签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，下面所列出的是cc_hits数据库的SQL表结构，它包含大量的所记录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',
  '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表所包含的字段如下所示：

- target_id和tag_id　这两个字段联接该表内部的任意单独行，以形成一个特定目标和标签的组合。
- num　该字段中保存了被单独行所表示的代码块在所指定的目标——标签组合内部的执行顺序。
- timestamp　该字段保存了自UNIX起始时间（1970年1月1日，00：00：00格林尼治标准时间）以来的秒数，这种格式非常易于转换到其他表示方式。
- eip　该字段保存了被单独行所表示的、已执行的代码块的绝对地址；该字段的值通常被格式化为一个十六进制的数字。
- tid　该字段保存了负责执行位于*eip*中地址值的代码块线程的标识符。这个线程标识符被Windows操作系统所分配，并且被存储以用于区分在多线程应用程序中，被不同的线程所执行的代码块。
- eax,ebx,ecx,edx,edi,esi,ebp以及esp　这些字段分别包含了8个通用寄存器在运行时的数字值。对于每一个通用寄存器和所存储的栈偏移量而言，存在着一个*deref*字段，它包含针对特定寄存器的、作为一个指针的ASCII数据。在ASCII数据字符串的末尾是一个标签，其值是（栈），（堆）和（全局）这三者之一，用于表示发现解除引用数据的位置。当由于相关的寄存器不包含一个解除引用地址，而造成不可能进行解除引用的情况下，使用

N/A值。

- esp_4,esp_8,esp_c以及esp_10　这些字段包含在其相关的栈偏移量处的数字值(esp_4=[esp+4],esp_8=[esp+8]等)。
- is_function　这是一个布尔型的字段，值1表示所覆盖的数据块（位于*eip*中的地址处）是一个函数的起始点。
- module　该字段存储发生覆盖的模块的名字。
- base　该字段包含了在上一个字段中所指明的模块的数字基地址。这个信息可以同*eip*一起被使用以计算在模块内部的覆盖数据块所发生的偏移量。

这种开放的数据存储机制为高级用户提供了一个发挥的余地，即生成定制的、比PStalker工具默认提供的报告更加全面的报告。下面让我们通过一个完整的实例分析来对该跟踪器进行一下测试。

23.5　实例研究

对于一个实际的实例研究而言，我们将应用PStalker代码覆盖工具来确定一个模糊审核的相对完整性。我们的目标软件是Gizmo项目[⊖]，它是一个包含VoIP、即时消息（IM）以及电话通信等功能的组件。图23-6是来自于Gizmo项目的Web站点的、被增加了注释的一个屏幕截图，它详细说明了该工具的主要特性。

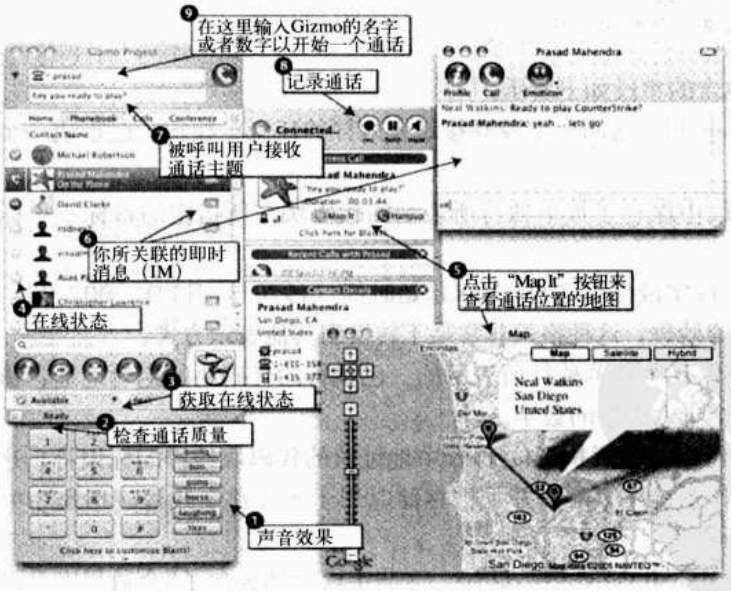


图23-6　Gizmo项目的特性列表

Gizmo实现了许多非常重要的特性，这也是它较为流行的一个主要原因。其中有许多特性

⊖ http://www.gizmoproject.com/

要优越于Skype[⊖]，Gizmo通过基于开放的VoIP标准如会话初始化协议（SIP－RFC3261）来构建，而不是实现一个封闭的私有化系统，从而将其与普遍使用的Skype相区别。这种设计决策就允许Gizmo能够同其他与SIP相兼容的解决方案实现互操作。同时，该设计决策也允许我们能够使用标准的VoIP安全测试工具来对Gizmo进行测试。现在我们的焦点已经放在了目标软件上，下一步必须要挑选一个目标协议并且选择一个适当的模糊器。

23.5.1 测试策略

我们将要对Gizmo处理错误SIP包的能力进行测试。尽管这里有若干个我们可以作为测试对象的VoIP协议，而SIP作为一个很好的选择，是因为大多数的VoIP技术都通过该协议实现了基于信号的会话。同样，现在已经有了一些免费可用的SIP模糊器。对于这里的测试而言，我们将使用一个由Oulu大学电子和信息工程系所发布的、免费可用的SIP模糊器，其名字为PROTOS测试套件：c07-sip[⊜]。PROTOS测试套件包含4527个分布在一个单独java JAR文件中的单个的测试用例。尽管该测试套件仅限于对INVITE消息进行测试，但是它非常适合我们这里的需求。选择该模糊器的一个次要动机是其开发团队已经通过Codenomicon公司创建了该模糊器的一个商业版本[⊜]。SIP测试套件的商业版本对SIP协议进行了更加深入的研究，并且绑定了31 971个单个的测试用例。

我们所要实现的目标如下：当执行完整的PROTOS测试用例集时利用PStalker，以度量所执行的代码覆盖。可以将这个实例进行扩展以完成一个实际的任务，例如比较不同的模糊测试解决方案或者确定一个给定的审核的可信性。本章的后面将进一步讨论使用代码覆盖作为度量手段所带来的好处。

PROTOS测试套件是从命令行开始运行的，并且提供了将近20个选项以调控其运行时行为，下面所示的是来自于帮助函数的输出信息：

```
$ 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
```

⊖ http://www.skype.com/

⊜ http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/

⊜ http://www.codenomicon.com/

```
                        Defaults to 100 ms (milliseconds)
  -replywait <ms>       Maximum 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 sent 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 injection to <index>
  -maxpdusize <int>     Maximum 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
```

你所要实现的目标可能是不同的，但是我们发现下面这些选项工作的最好（touri选项是唯一的一个强制使用的选项）：

```
java -jar c07-sip-r2.jar -touri 17476624642@10.20.30.40 \
                         -teardown                      \
                         -sendto 10.20.30.40            \
                         -dport 64064                   \
                         -delay 2000                    \
                         -validcase
```

将delay选项设置为两秒，将会在有些时候使Gizmo来释放资源，更新其GUI，而在另外一些时候将在测试用例之间进行恢复。validcase参数指定了PROTOS应当确保一个有效的通信在每个测试用例之后以及在继续执行之前能够发生。这个基本的技术就允许模糊器来检测目标在何时出现问题，并且停止测试。在这种情形下它是一个非常重要的因素，因为Gizmo实际上已经发生了崩溃。对Gizmo而言幸运的是，该漏洞是由于对一个空指针的解除引用而被触发的，因此它不能够被加以利用。然而，问题却是存在的（提示：问题发生在最初的250个测试用例中）。你可以自己下载该测试套件，然后找出存在的问题。

Gizmo在发生崩溃时的上下文输出：

```
[*] 0x004fd5d6 mov eax,[esi+0x38] from thread 196 caused access violation when
attempting to read from 0x00000038

CONTEXT DUMP
  EIP: 004fd5d6 mov eax,[esi+0x38]
  EAX: 0419fdfc (  68812284) -> <CCallMgr::IgnoreCall() (stack)
  EBX: 006ca788 (   7120776) -> e(elllllllllllllllllllll (PGPlsp.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||@hoO@*@bOzp (stack)
+14: 0061cb99 (   6409113) -> N/A

disasm around:
        0x004fd5c7 xor eax,esp
        0x004fd5c9 push eax
        0x004fd5ca lea eax,[esp+0x24]
        0x004fd5ce mov fs:[0x0],eax
        0x004fd5d4 mov esi,ecx
        0x004fd5d6 mov eax,[esi+0x38]
        0x004fd5d9 push eax
        0x004fd5da lea eax,[esp+0xc]
        0x004fd5de push eax
        0x004fd5df call 0x52cc60
        0x004fd5e4 add esp,0x8

SEH unwind:
        0419ff9c -> 006171e8: mov edx,[esp+0x8]
        0419ffdc -> 006172d7: mov edx,[esp+0x8]
        ffffffff -> 7c839aa8: push ebp
```

在Gizmo的启动过程中，通过对从端口5060（标准的SIP端口）传进和传出的包进行检查，收集了touri和dport两个参数。图23-7显示了相关的屏幕截图，并且在其中突出显示了我们所感兴趣的值。

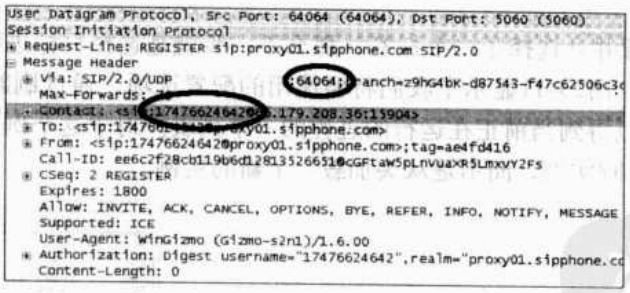


图23-7 Gizmo启动时捕获的SIP包的解析图

数字17476624642是分配给Gizmo账户的实际的电话号码，而端口64064是作为标准的Gizmo客户端SIP端口而显示的。我们的策略已经准备好了。目标软件、协议和模糊器都已经选择好了，并且已经设置了针对模糊器的各个选项。现在，就可以讨论该实例中所采用的具体方法了。

23.5.2 测试方法

因为我们正在对SIP进行模糊测试，因此这里将特别对SIP处理代码感兴趣。对于我们而言

幸运的是，该代码能够被很容易地分离，因为名为SIPPhoneAPI.dll的库立刻成为了我们的目标。首先来解决最重要的事情：将该库加载到IDA Pro中，并且运行pida_dump.py脚本来生成SIPPhoneAPI.pida。由于我们只关注于监视代码覆盖情况，因此将“基本块”指定为针对pida_dump.py脚本的分析深度将会带来性能上的改进。接下来，我们启动PaiMei，并且遍历必要的先决条件以开始代码覆盖操作。为了得到有关PaiMei的特定用法的更多信息，可以参阅相关的在线文档[⊖]。图23-8所描述的第一个步骤是创建一个名为“Gizmo”的目标，并且添加一个名为“Idle”的标签。我们将使用“Idle”标签来记录在“保持不变”的Gizmo内部的代码覆盖情况。这并不是一个必须的步骤，但是由于你可以指定多个标签用于记录，并且可以进一步选择单个的标签用于过滤，因此这一步骤也从未被忽略过。必须要选择“Idle”标签以用于“跟踪”，如图23-9所示，并且必须要加载SIPPhoneAPI PIDA文件，如图23-10所示。



图23-8　PaiMei PStalker目标和标签的创建



图23-9　选择“Idle”标签以用于跟踪

一旦建立了数据源并且选择了一个标签，那么下一步就是选择目标进程以及捕获选项，并且开始记录代码覆盖。图23-11显示了我们将要使用的配置选项。通过刷新进程列表（Refresh Process List）按钮可以得到当前正在运行的进程的一个列表。对于这个例子而言，我们选择关联到Gizmo的一个运行时实例，而不是从头加载一个新的实例。

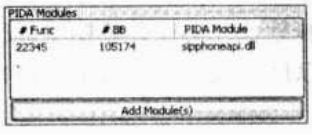


图23-10　加载SIPPhoneAPI PIDA模块



图23-11　代码覆盖捕获选项

⊖ http://pedram.openrce.org/PaiMei/docs/

关于覆盖深度，我们选择基本块以得到最小粒度的代码覆盖。正如你在图23-10中所看到的，我们的目标库只有不到25 000个函数，而基本块的个数是函数个数的4倍还多。因此，如果执行速度是我们所考虑的一个因素的话，我们可以通过选择函数作为覆盖深度，从而以牺牲覆盖粒度来提高性能。

Restore BPs复选框标志着基本块是否应当在其第一次被覆盖之外再被跟踪。由于我们只对代码的执行位置感兴趣，因此可以将其选项设置为不可用来提高性能。

最后，我们将清空Heavy复选框。该复选框标记着代码覆盖工具是否应当在每个断点处执行运行时上下文审查，并且保存所发现的内容。同样，由于我们只对代码覆盖感兴趣，因此额外的数据是不必要的，这样就避免了执行不必要的数据捕获而造成的性能损失。

一旦我们按下了“Start Stalking”按钮，代码覆盖工具将关联到Gizmo，并且开始对执行情况进行监视。如果你对该过程内部的特定处理细节感兴趣，可以参阅PaiMei文档。一旦目标被激活，那么指明PStalker的动作和进度的日志消息将被写入到日志窗格中。下面的代码片段说明了一个启动成功的运行：

```
[*] 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 Project\SipphoneAPI.dll
[*] Setting 105174 breakpoints on basic blocks in SipphoneAPI.dll
[*] Loading 0x16000000 \Internet\Gizmo Project\dnssd.dll
[*] Loading 0x006f0000 \Internet\Gizmo Project\libeay32.dll
[*] Loading 0x71ad0000 \WINDOWS\system32\wsock32.dll
[*] Loading 0x7c340000 \Internet\Gizmo Project\MSVCR71.DLL
[*] Loading 0x00340000 \Internet\Gizmo Project\ssleay32.dll
[*] Loading 0x774e0000 \WINDOWS\system32\ole32.dll
[*] Loading 0x77120000 \WINDOWS\system32\oleaut32.dll
[*] Loading 0x00370000 \Internet\Gizmo Project\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
...
```

我们现在可以激活模糊器，并且对在空闲时间内所执行的代码进行跟踪。接下来，我们要等待覆盖断点的列表逐渐减少以表明当Gizmo的空闲时间已经被耗尽时，在SIP库内部的代码是执行的。接下来，我们要创建一个新的标签，并且通过右键点击菜单选择Use for Stalking选项来将其选作为活动的标签。最后，我们右键点击前面所创建的“Idle”标签，并且选择过滤器标签。这将有效地忽略在空闲状态期间所执行的基本块。现在，我们可以启动模糊器，并且查看其运行情况。图23-12显示了同所有4 527个PROTOS测试用例相对应的代码覆盖情况。

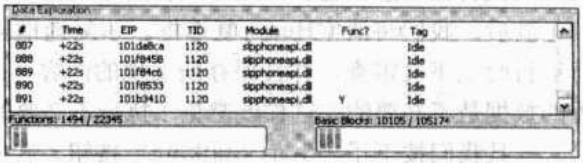


图23-12　代码覆盖结果

总之，PROTOS测试套件只能够到达SIPPhoneAPI库内部大约6%的函数以及9%的基本块。这对于一个免费的测试套件而言已经是不错的了，但是它说明了为什么运行一个单独的测试套件所产生的效果不能够等同于执行一个完整的审核。在运行该测试套件之后，我们对大部分代码的测试将会失败，并且也不太可能对所遇到的代码内部的功能进行完整的测试。然而，可以通过在测试用例之间的短暂间隙内增加一个进程重启机制来改善这些覆盖结果。这其中的原因是，即使Gizmo成功地对有效的测试用例做出了响应，我们仍然要在错误的测试之间插入PROTOS，而在一段时间之后Gizmo仍然停止正确的操作，并且影响代码覆盖。重启进程以重新设置内部状态并且恢复完整的功能，允许我们更加准确和全面地对目标进行研究。

考虑PROTOS只包含其商业版本测试用例数的1/7，那么我们能够期望应用其商业版本以获得比用PROTOS多于7倍的代码覆盖率吗？可能不会。尽管能够使代码覆盖率有所提高，但是在测试用例数以及所执行的代码总数之间并不存在一个一致的或者可预计的关系。只有经验结果才是准确的。

那么，使用这种方式来跟踪模糊器能够获得什么好处呢？我们如何对该技术进行改进呢？下面的部分将对这些问题提供某些答案。

23.6　益处和改进的余地

在传统意义上，模糊测试并没有以一种非常科学的方式来执行。安全研究者和QA工程师经常编写一些基本的脚本以生成病态数据，并且允许在一段任意的时间内执行数据的创建过程，然后命令该过程停止。随着最近商业化模糊测试解决方案的增多，你将会发现通过利用由商业工具提供商如Codenomicon所开发的完整的测试用例列表，使得模糊测试向着更加科学化的方向前进了一大步。对于那些只是寻找漏洞以发布或者售卖报告的安全研究者而言，通常不需要对该过程做出任何改进。对于目前软件安全性的状态而言，现有的方法已经足以生成大量的关

于错误和漏洞的一个详细列表了。然而，随着安全性开发和测试的发展以及对其重要性认识的提高，就迫切需要有一种更加科学化的方法。软件开发者和QA团队所关心的是发现所有的漏洞，而不只是一些低层次的问题，他们将更加关注于采用一种更加科学化的方法来进行模糊测试。模糊器跟踪技术就是实现此功能的一个必备步骤。

仍然考虑贯穿本章所使用的同一个例子，考虑在最近发布的VoIP softphone的开发中所包含的不同的方面，以及从模糊器跟踪中可以得到的好处。对于项目经理而言，了解被测试代码的特定区域将允许他更加准确地做出决定，并且对他们的产品所经历的测试层次更加有信心。例如，考虑下面的情形："我们最新发布的VoIP softphone产品通过了45 000个精心设计的测试用例。"没有提供任何证据以证明该结论。可能在这45 000个精心设计的测试用例中，只有5 000个覆盖了目标软件softphone所提供的特性。也可能是整个测试集只执行了10%的目标软件softphone所提供的可用特性。也可能这些测试用例是由真正具有才能的工程师所编写的，并且覆盖了全部的代码。后一种情形将是非常好的，但是存在同样的问题，即如果没有代码覆盖分析，那么就无法确定最初结论的有效性。将一个模糊器跟踪器与同一个测试集相关联，可能会产生如下更加详细的结论："我们最新发布的VoIP softphone产品通过了45 000个精心设计的测试用例，并且完整覆盖了超过90%的代码。"

对于QA工程师而言，确定softphone软件的哪些部分没有被测试将允许他生成更加智能化的测试用例以用于下一轮的模糊测试。例如，在对softphone进行模糊器跟踪的过程中，工程师可能会注意到由模糊器所生成的测试用例将会一直触发一个名为parse_sip()的例程。对该例程进行分析，将会发现尽管该函数自身被执行了多次，但是并没有覆盖该函数内部的所有分支。该模糊器并没有遍历完该解析例程内部所有可能的逻辑路径。为了提高测试的效果，工程师应当对未覆盖的分支进行研究以确定应当对模糊器的测试数据生成例程做出哪些改动，以执行这些被漏掉的分支。

对于开发者而言，了解在任意被发现的错误之前所执行的特定指令集和路径，将允许他迅速并且准确地定位以及修复受影响的代码区域。目前，QA工程师和开发者之间的沟通可能仍然处于一个较高的层次上，这就使得二者进行重复研究而浪费了时间。可以不从QA接收一份列出了触发不同错误的测试用例的高层报告，开发者可以接收一份同测试用例相关的、详细描述了可能的错误位置的报告。

记住代码覆盖作为一个单机版的度量方法，并不意味着执行了一个完整的测试过程。考虑在一个可执行文件内部遇到了一个字符串复制操作，但是只使用很小的字符串长度进行了测试。该代码可能会存在一个我们的测试工具没有发现的缓冲区溢出漏洞，因为一个不够长的或者未正确格式化的字符串被传递给了它。确定我们已经测试了什么只完成了一半的工作，理想的情况下我们还应当提供对测试效果的度量。

进一步的改进

所考虑增加的一个方便的特性是，代码覆盖工具同模糊器进行通信的能力。在传输每一个单独的测试用例之前和之后，模糊器都可以向代码覆盖工具通知相关的事务，因此就可以关联

并且记录与每一个单独的测试用例相关联的覆盖信息。另外一个可选的实现方法就是通过后处理，即通过对齐与数据传输相关联的时间戳以及与代码覆盖相关联的时间戳。由于延迟以及未很好同步的时钟，使得该方法的有效性比较低，但是非常易于实现。在上述两种情况下，最终的结果都允许一个分析者对与一个单独的测试用例相关的、所执行的特定代码块进行深入分析。

目前，枚举出在目标软件内部的所有基本块和函数的位置的需求是一个非常繁琐并且易于出错的过程。针对此问题的一个解决方案是在运行时对基本块进行枚举。根据Intel IA－32体系架构的软件开发者手册[⊖]，第3B卷[⊜]第18.5.2节中的描述，Pentium4和Xeon处理器具有对“分支级的单步执行”的硬件层次上的支持：

BTF（针对分支的单步执行）标志（1位）

当设置该标志时，处理器将在EFLAGS寄存器中对TF标志作为一个“分支级的单步执行”来对待，而不是作为一个“指令级的单步执行”标志来看待。这种机制允许处理器单步执行分支、中断以及异常。可以参阅上述手册的第18.5.5节“针对分支、异常和中断的单步执行”(Single-Stepping on Branches, Exception and Interrupts)。

利用这种特性所提供的优越性可以使所开发的工具能够在基本块的层次上跟踪一个目标进程，而不需要进行预先分析。同时，该特性也使运行时的速度得到了提高，因为在启动时不再需要为每一个单独的基本块设置软件断点。然而需要记住的是，静态分析和对基本块的枚举仍然是必须的，它们被用来确定代码的全部的范围，并且指明代码覆盖的百分比。另外，尽管在开始时使用这种方法来进行跟踪可能会更快些，但到最后它可能将不会比前面所详细讨论的方法运行的快。这是因为它无法停止对已经执行过的块的监视，以及它无法只对一个特定的模型进行跟踪。尽管如此，它仍然是一种非常有趣并且功能强大的技术。打开本章前面所描述的PyDbg的单步执行器，可以在针对运行中的分支级单步执行器的OpenRCE中看到名为“使用Intel MSR寄存器的分支跟踪”[⊝]的博客条目。对于该方法的深入研究作为一个练习留给读者来完成。

关于改进的最后一个方面是，考虑代码覆盖和路径覆盖（或者是进程状态覆盖）之间的差异。我们在本章中已经讨论过的内容都与代码覆盖相关；换句话说，也就是确定哪些单独的指令或者代码块被实际执行过。路径覆盖则要进一步考虑在代码块组成的集合中，可能会产生的各种不同的路径。例如，考虑图23-13所示的代码块的一个集合。

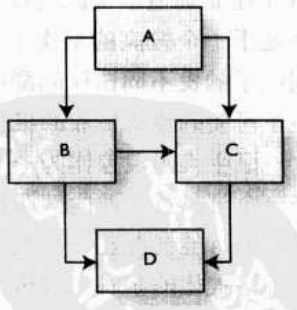


图23-13　控制流程图示例

代码覆盖分析可能会发现所有的4个块A,B,C和D都被一系列的测试用例执行过。那么所有的路径是否都被执行过呢？如

⊖ http://www.intel.com/products/processor/manuals/index.htm

⊜ ftp://download.intel.com/design/Pentium4/manuals/25366919.pdf

⊝ http://www.openrce.org/blog/view/535

果都被执行过，那么每一个单独的测试用例又是执行的哪些路径呢？要回答这些问题就需要进行路径分析，这种技术能够进一步确定下面哪一种可行路径的组合被覆盖：

- A → B → D
- A → B → C → D
- A → C → D

例如，如果上述列表中只有第一条和第三条路径被执行过，那么代码覆盖分析将会发现达到了百分之百的代码覆盖。而在另一方面，路径覆盖分析将发现只有66%的可行路径被覆盖。整个额外的信息就使得分析者能够进一步对测试用例进行调整和改进。

23.7 小结

在本章中，我们介绍了应用于模糊测试领域的代码覆盖的概念，并且在一开始引入了一个只有很少Python代码的非常简单的解决方案。通过研究二进制可执行代码的构造块，我们说明了一个更加高级的、在前面所实现的一个方法。尽管对于模糊测试而言，大部分的关注焦点都在于测试数据生成领域，但是该方法只解决了一半的问题。我们不应当只是问“如何开始模糊测试？”我们同样需要问“何时来停止模糊测试？”如果不对代码覆盖进行监视，那么我们就无法智能化确定应当在何时来停止测试。

在本章中，引入并描述了一个用户友好的、开源的代码覆盖分析工具PaiMei PStalker。研究了该工具在模糊测试领域中的应用，同时也讨论了针对该工具的改进内容。该代码覆盖工具的应用以及概念，或者说是“模糊器跟踪”技术，使得我们能够进一步采用一种更加科学化的方法来进行模糊测试。在下一章中，我们将讨论智能化的异常检测和处理技术，通过将这些技术加入到我们的测试套件中，就可以将模糊测试水平推向一个更高的层次。

第24章　智能故障检测

“再一次在华盛顿特区的礼堂中，我从未想要对那些无法解释的事情做出任何解释。”

——George W. Bush，Portland，OR，2000年10月31日

我们已经知道了如何来选择一个目标应用，我们也知道了如何来生成测试数据。通过前面章节的讨论，我们也知道了如何来对处理数据的位置和方式进行跟踪。需要掌握的下一个重要的概念就是确定我们的模糊器在何时成功导致了某些问题的发生。这可能并不总是能够不证自明的，因为从目标系统的外部可能很难检测到错误所产生的结果。同样，在当前可用的所有模糊器中，这个问题也没有被引起足够的重视。然而，与模糊器跟踪不同，我们将对商业的和开源的工具都进行介绍。

在本章的开始，我们将首先研究最基本的错误检测方法，例如简单的响应解析方法。然后，我们通过利用调试工具和技术来进入到一个更加高级的层次。最后，我们将通过对动态二进制插装（DBI）的研究，简要涉及一些最高级的错误检测技术。总而言之，本章所讨论的工具、技术和概念将有助于我们确定模糊器在何时已经完成了其任务。

24.1　基本的错误检测方法

当你盲目地发出5 000个病态的IMAP验证请求，而其中的一个请求导致IMAP服务器发生崩溃，并且直到最后一个测试用例才检查IMAP服务器的状态，那么你将会得到什么呢？答案很简单：你得不到任何东西。如果连哪一个测试用例实际导致了IMAP服务器发生崩溃都不知道的话，那么你所拥有的也就是在一开始运行模糊器之前那样少的信息。如果一个模糊器不知道其目标的运行状况，那么它将没有任何用处。

这里有一些技术可供模糊器来利用以确定一个单独的测试用例是否在目标中发现了任何的异常情况，其中最简单的一种方法可能就是在测试用例之间增加一个连接性检查，就像上一章的PROTOS那样。继续我们关于IMAP的例子，假定我们正在运行一个模糊器，它生成了如下的两个最初的测试用例：

```
x001 LOGIN AAAAAAAAAAAAAAAAAAAAAAAAAAAA...AAAA
x001 LOGIN %s%s%s%s%s%s%s%s%s%s...%s%s%s%s%s
```

在每一个所生成的测试用例之间，模糊器可以试图在端口143（IMAP服务器端口）建立一个TCP连接。如果建立连接失败，那么就可以做出假定是最后一个测试用例导致服务器发生了崩溃。由Codenomicon所开发的商业测试套件采用的就是这种方法，它可以在每个测试用例之后来可选的发送一个合理的或者是已知的好用例。实现此功能的伪代码非常简单，如下所示：

```
for case in test_cases:
    fuzzer.send(case)
```

```
    if not fuzzer.tcp_connect(143):
        fuzzer.log_fault(case)
```

我们可以通过用一个已知的有效响应测试来替代连接性检查，从而对此处理逻辑做出细微改进。例如，如果我们知道用户名paimei和密码whiteeyebrow是IMAP服务器上有效的注册信息，那么我们就可以确保该服务器不只是可用的，而且可以通过试图作为用户paimei来进行成功的验证，从而使其能够正常运行。如果有效响应测试执行失败，那么就可以做出假定是最后一个测试用例导致了服务器发生崩溃。同样，实现此功能的伪代码也非常简单，如下所示：

```
for case in test_cases:
    fuzzer.send(case)
    if not fuzzer.imap_login("paimei", "whiteeyebrow"):
        fuzzer.log_fault(case)
```

在前面，我们曾指出，建立连接失败或者已知的有效响应测试执行失败都可以使我们做出这样的假定，即是最后一个测试用例在目标内部导致了一个错误。但这并不是一直正确的。例如，考虑在一个IMAP服务器模糊测试的过程中，在执行了第500个测试用例之后，连接性测试的执行失败。我们已经发现了有一个输入可以导致服务器发生崩溃。然而，在总结相关信息报告我们的重大发现之前，我们将重启IMAP服务器，并且单独运行第500个测试用例，然而使我们沮丧的是没有任何事情发生。究竟是怎么回事呢？可能的场景就是一个或者是若干个先前测试用例的组合，使IMAP服务器进入到了这样的一个状态，即允许第500个测试用例使服务器发生崩溃。然而，第500个测试用例自身似乎是被进行了正确的处理。在这种情况下，我们可以利用另外一个简单的、我们称之为*模糊器单步执行*的技术，来帮助我们减少出现这种情形的可能性。

该问题的一般性描述如下所示：我们知道在测试用例1到499之间的某个地方，IMAP服务器被置于了这样一种状态，即它允许第500个测试用例使服务器发生崩溃，但是我们并不知道是哪些测试用例的组合导致了崩溃的发生。这里有许多我们可以利用的步进算法。下面首先列出了一个简单的算法：

```
# find the upper bound:
for i 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()

# find the lower bound:
for i in xrange(upper_bound, 0, -1):
    for j in xrange(i, upper_bound + 1):
        fuzzer.send(j)
```

```
fuzzer.send(500)

if fuzzer.tcp_connect(143):
    lower_bound = i
    break

fuzzer.restart_target()
```

上述处理逻辑的前半部分功能是负责定位生成所期望状态的序列的上界。这是通过从第1个测试用例递增到第n个测试用例来实现的，后面跟着第500个测试用例，而n则是持续的以1来进行递增。当每一个序列都被测试之后，模糊测试目标将会被重新启动以重设内部状态。然后，应用类似的处理逻辑来定位序列的下界。当后面跟着的第500个测试用例导致目标发生崩溃时，我们的组合例程已经将测试用例的序列进行了分离。由于我们可以再一次使目标发生崩溃，因此仍然可以得到令人兴奋的结果。当然，这是一个相对简单的算法，它并没有考虑这样的一种实际情况，即在我们所确定的区域中可能有几个测试用例将服务器置于了一种漏洞状态。例如，我们可能会发现后面跟着测试用例500的测试用例15到20导致了崩溃的发生，而实际上是测试用例15、20以及500这样一个连续的序列将会导致崩溃。尽管如此，该算法也将缩减可能导致崩溃的测试用例的范围。

上面所讨论的简单的解决方案是监视错误的一种最基本的方法。为了更好的理解我们的模糊器目标内所发生的情形，我们可以使用一个调试器来进行深入研究，并且从更低的层次上来进行探查。在开始研究之前，对于我们而言，准确确定所要搜索的内容是非常重要的。

24.2　我们所要搜索的内容

在最低的层次上，当不同的错误、异常或者中断（统称为事件或者异常）如内存访问违规或被0除等问题发生时，CPU将会通知操作系统。许多这些事件都是非常自然地发生的。例如，考虑当一个进程试图访问当前在磁盘上被分页的内存时，将会发生什么情况。一个页错误将会产生，并且操作系统通过将适当的页从磁盘缓存中加载到主内存来处理该事件。整个事务处理过程对于做出最初的内存访问请求的进程而言都是透明的。没有被操作系统所处理的事件将会被冒泡排序到进程的层次上。这也是我们开始感兴趣的地方。

当我们的模糊器遍历一个目标进程或者设备时，它可能会触发许多事件。将一个调试器关联到模糊器目标将允许我们在这些事件发生时截取它们。当与前面所描述的简单方法结合在一起时，一旦调试器捕获了事件并且中断了模糊器目标，那么后续的连接性检查或者已知的有效输入检查将会执行失败。这种组合方法允许我们潜在地发现更多存在于示例IMAP服务器中的问题。当不同的异常事件发生在汇编层次时，需要我们智能化确定该事件在更高层次上的触发源，并且更重要的是要确定从安全性的角度来看，该事件是否是非常严重的问题。下面将所有的软件内存破坏漏洞划分为三个基本的类别。我们将分别针对每一个类别来研究相关的实例，同时要确定每一个漏洞可能会产生的异常类型：

- 错误地将执行的控制权转移到了一个地址
- 错误地从一个地址读取数据

• 错误地向一个地址写入数据

作为错误地将执行的控制权转移到了一个地址的例子，考虑典型的栈溢出问题，下面的C代码片段摘自于一个栈框架中的函数，该栈框架如图24-1所示。

```
void taboo (int arg1, char *arg2)
{
    int x;
    char buf[16];
    int y;

    strcpy(buf, arg2);
}
```

这个所设计的函数包含一个整数以及一个字符串参数，并且声明了三个局部变量，然后将所提供的字符串复制到被定义的栈缓冲区中，而不考虑源字符串的大小以及目标缓冲区的大小。当函数taboo()被调用时，其参数将以相反的次序被压入栈中。所产生的汇编CALL指令将会隐式地把当前指令的指针地址（存储在寄存器EIP中）压入到栈中。这样的话，一旦被调用的函数执行完毕，CPU就可以知道将控制权转移回何处。在通常情况下，该函数序言将会把当前的结构指针（存储在寄存器EBP中）压入到栈中。最后，通过从栈指针（存储在寄存器ESP中）中去除总的变量大小来为所声明的三个局部变量留出空间。所生成的栈结构如图24-1所示。

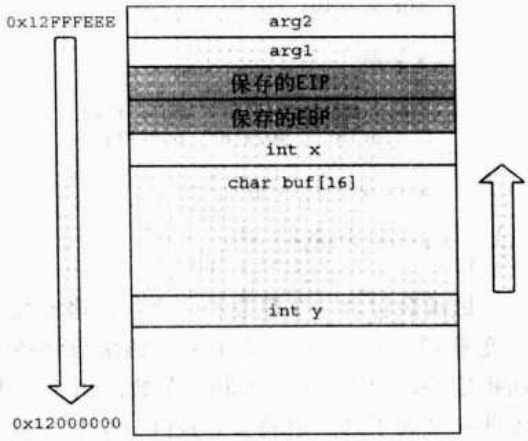


图24-1 示例栈结构

正如图24-1所描述的，该栈是从高层地址逐渐向下变到低层地址的。然而，数据写入操作则是发生在相反的向上的方向。如果字符串arg2的长度大于16字节（所声明的buf的大小），那么strcpy()将会持续地进行复制，超出了为buf所分配的空间大小，并且重写了局部整数变量x，接着重写了所保存的结构指针、所保存的返回地址以及第一个参数等。假定arg2是一个由字符A构成的长字符串，那么所保存的EIP将被0x41414141所重写（0x41是ASCII字符A的十六进制值）。

当函数taboo()执行完毕时，RETN汇编指令将恢复被重写的、所保存的EIP，并且将执行的控制权转移给处于地址0x41414141的指令。依赖于位于地址0x41414141处的内存的内容，可能会产生几个不同的场景。在大多数情况下，没有有效的内存页被映射到这个特别的地址，并且当CPU试图从不可读的地址0x41414141处读取下一条指令时，将会生成一个ACCESS_VIOLATION异常。注意在这个特定的例子中，如果arg2的长度大于16字节但是小于或者等于20字节，那么将不会发生任何异常。

如果存在有效的数据，但是在该地址处不存在可执行的代码，并且基本的进程对不可执行

（NX[⊖]）页的许可已经提供了支持，那么当CPU试图从不可执行的地址0x41414141处开始执行时，同样将会生成一个ACCESS_VIOLATION异常。最有趣的是，如果在该地址处存在有效的可执行代码，那么将不会生成任何的异常。这是本章后面将要讨论的一个关键问题。这个例子说明了控制流转移的一种最直接的形式。

作为错误的从一个地址读取数据的例子，考虑下面的C代码片段，它摘自于一个栈框架中的函数，该栈框架如图24-1所示。

```
void taboo_two (int arg1, char *arg2)
{
    int *x;
    char buf[] = "quick brown dog.";
    int y = 10;

    x = &y;

    for (int i = 0; i < arg1; i++)
        printf("%02x\n", buf[i]);

    strcpy(buf, arg2);

    printf("%d\n", *x)
}
```

该函数包含一个整数和一个字符串参数，声明了三个局部变量，并且将整数变量x设置为指向变量y的一个指针。接下来，该函数将持续输出一个可变的数量，该数量是由整数参数arg1所指定的buf中的十六进制字节数。由于对用做循环终止值的该参数没有施加任何有效性检查，因此一个大于16的值将会导致printf()例程去访问并且显示它本不应当访问和显示的数据。

数据的读取是向着更高的地址方向进行的，这与栈增长的方向是相反的。因此该循环将会发现局部整数变量x的值，接着将发现所保存的结构指针、所保存的返回地址以及第一个参数等等。依赖于arg1的大小，可能不会产生一个针对读取的ACCESS_VIOLATION异常，当足够短的值简单地驱使该循环去访问那些程序员在逻辑上不想被访问的数据时，这些数据被CPU和操作系统很好地共享，因为它们并没有意识到这些变化。

接下来，正如在上一个例子中那样，执行了一个没有边界的字符串复制。然而，假定arg2只包含20个A字符（比buf的大小多出4个字节），那么将只有局部指针变量x被重写。在这种情形下，栈溢出将不会导致像前面那样一个无效的执行流的转移。然而，在对printf()的调用过程中，当被重写的变量作为一个指针被解除引用时，那么当试图从地址0x41414141处读取时，将可能会产生一个ACCESS_VIOLATION异常。

关于错误地从一个地址读取数据的更加现实的例子，可以考虑下面的代码片段，它说明了典型的格式化字符串漏洞的发现过程：

```
void syslog_wrapper (char *message)
{
    syslog(message);
}
```

⊖ http://en.wikipedia.org/wiki/NX_bit

根据syslog() API的原型，期望有一个格式化字符串，并且其后面跟着可变数量的格式化字符串参数。通过直接将包含在参数message中的用户所提供的字符串传递给syslog()，那么包含在该参数内部的任意格式化字符串标记都将被解析，并且相关的格式化字符串参数也将被解除引用。由于不存在这样的格式化字符串，因此当前存在于函数栈中的任何值都将被利用。例如，如果该参数包含%s%s%s%s%s，那么将会有5个地址从栈中被读取，并且将被作为一个字符串指针而被解除引用，直到发现一个空字节为止。提供足够的%s格式化字符串标记通常会导致一个无效指针作为一个字符串被处理，这将会发生一个ACCE_VIOLATION异常。在某些情况下，该指针将是有效的、可以设定地址的内存，但是该内存的内容将不是以空作为终结，这将会导致当函数正在搜索一个空字节时，很快读取到一个页的末尾。注意依赖于环境以及所提供的格式化字符串标记的数量，可能不会发生任何的违规操作。同样，本章后面将要对这个关键概念进行更加详细的讨论。

作为错误地向一个地址写入数据的例子，考虑下面的C代码片段，它摘自于一个栈框架中的函数，该栈框架如图24-1所示。

```
void taboo_three (int arg1, char *arg2)
{
    int x;
    char buf[] = "quick brown dog.";
    int y;

    buf[arg1] = '\0';
}
```

该函数包含一个整数和一个字符串参数，声明了三个局部变量，然后在由参数arg1所指定的索引处，将存储在buf中的字符串进行截取操作。由于对所提供的索引没有进行任何的有效性检查，因此一个大于16的值将会导致一个错误的空字节被写入到一个不应当被写入的地址中。正如在上一个例子中所看到的，依赖于该地址所处的位置，一个实际的异常可能会也可能不会产生。即使不会产生一个异常，当在进行模糊测试时，我们也应当知道何时会发生异常。

关于错误的向一个地址写入数据的更加现实的例子，可以考虑下面的代码片段及其简化的例子，它说明了一个典型的堆溢出漏洞的发现过程[⊖]：

```
char *A = malloc(8);
char *B = malloc(16);
char *C = malloc(24);

strcpy(A, "PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP");
free(B);
```

在上述代码片段中，声明了三个字符指针，并且将其初始化为指向来自于堆的动态分配的内存。在其他所存储的内嵌信息中，每一个堆包含一个向前和向后的链接以创建双向链接列表，如图24-2所示。

当执行到对strcpy()的调用时，所分配的被A所引用的缓冲区将会发生溢出，因为它不能

⊖ http://doc.bughunter.net/buffer-overflow/free.html

够存储由字符P所构成的长字符串。由于溢出的缓冲区没有被保存在栈中，因此就无法获得一个所保存的返回地址。那么问题究竟出在什么地方呢？在下一行代码中，对free(B)的调用必须要从列表中解除对B的链接，并且将A和C相互链接起来。可以通过下面的基本逻辑来实现这一功能：

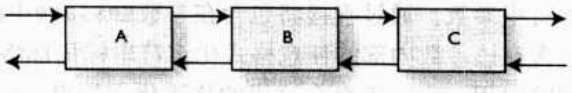


图24-2　堆的双向链接列表

```
B->backward->forward = B->forward
B->forward->backward = B->backward
```

由于向前和向后的指针都存储在每一个堆的起始位置处，因此由调用strcpy()所产生的溢出将会在B中用0x50505050（0x50是ASCII字符P的十六进制值）来重写这些指针值。在汇编层次上，这将会被解释为无效的读和写操作。

在安全解决方案上的一个失败的尝试

尽管与我们现在所讨论的主题并不直接相关，我们还是要在此指出为创建一个预编译的安全性解决方案而做出的一次可笑的尝试。在之前的某个时候，Drexel大学计算机科学系发表了一篇名为《使用程序变换来防止缓冲区溢出以保证C程序的安全性》（Using Program Transformation to Secure C Programs Against Buffer Overflows）的论文[⊖]。

该论文通过简单地使用从堆中所分配的缓冲区来替代所有栈缓冲区的使用，从而提出了保证减少所有的可利用缓冲区漏洞的能力。

这篇10页长的论文讨论了变换的动态方面以及变换的有效性，引用了大量的参考文献，甚至列举了一个例子以说明该变换如何能够将一个假设的、代码执行利用漏洞转换为一个简单的DoS。但是对于其作者而言不幸的是，在其整个研究过程中从未注意到这样一个事实，即堆溢出同样也是可利用的。

24.3　选择模糊值时的注意事项

到目前为止，我们已经说明了一些高层次的代码片段和软件漏洞概念如何能够转换为一个所生成的低层次的事件。下面让我们来解决模糊值选择过程中的一个关键问题。你可能早已经注意到依赖于不同的条件，作为内存地址而被引用的模糊值实际上可能并不会导致一个访问违规，因此也就不会产生一个异常。当选择在模糊请求中所使用的值时，必须要记住这一点。没有被很好地选择的值可以导致漏报的情形，如我们的模糊器成功遍历了漏洞代码但是却不能对其加以识别。

例如，考虑这样的情形，即在内存的解除引用中利用了4个字节的模糊器所提供的数据。典型的由字符A构成的字符串导致了对内存地址0x41414141进行解除引用。尽管在大多数情况下，在这个特定的地址处没有有效的内存页被映射，因此会导致我们所期望的ACCESS_VIOLATION异常，那么也会存在这样的情形即该地址确实是有效的，这样将不会有

⊖ http://www.cs.drexel.edu/~spiros/research/papers/WCRE03a.pdf

任何异常事件生成。尽管在所有可能出现的情况中，我们的模糊器都已经进行了足够的破坏性操作以触发一个异常，但是我们仍然需要尽可能快知道在何时产生了异常情况。如果给定这样一个机会，即选择被用做一个无效内存地址的4个字节，那么你将会选择什么值呢？选择处于内核地址空间中（通常是0x80000000-0xFFFFFFFF）的一个地址将会是很好的，因为内核空间对于用户模式而言是永远不可访问的。

下面来看另外一个例子，考虑格式化字符串漏洞的情形。我们看到如果没有提供足够的%s标记，那么后续的栈指针解除引用就可能不会触发任何的无效内存引用。作为最先想到的方法，你可能会试图通过简单增加在不同的模糊请求中所使用的%s标记的数量来解决这个问题。然而，进一步的分析将会发现其他的一些因素如所施加的字符串长度限制，将会使该解决方案变得不起作用。如何能够更好地解决这个问题呢？可以回忆一下，利用格式化字符串漏洞的关键就是使用格式化字符串标记%s及其派生标记向栈中写入的能力。%n格式化字符串标记将应当被格式化字符串输出的字符的数量写入到相关的栈指针中。通过提供一系列的%n标记来替代%s标记，就使我们能够在缩减输入字符串长度的限制下来触发一个错误。对于%n标记可以被过滤或者设置为不可使用的情形而言，分散的%n%s标记的一个组合可能是最好的选择（可以参阅第6章“自动化的测试以及数据生成”，以获得更多的信息）。

简而言之，对所选择的模糊数据在何处结束以及它们如何被使用给予关注，将会给我们带来很大的好处。可以参阅第6章以获得关于选择合适的模糊值的更多的信息。

24.4 自动化的调试器监视

在调试器帮助下进行目标监视的方法之所以不可行，是因为大多数的调试器都是被设计为以交互的方式来使用的，而从中自动化抽取出有用的信息以及通过程序来控制实际的调试器是不可能的。我们所要进行的最后一项操作就是手工在模糊器中进行登记，并且当调试器识别出一个异常时重启进程。幸运的是，我们可以再一次利用前面所介绍的逆向工程框架PaiMei[⊖]所提供的功能。利用该框架，我们可以构造一个包含模糊测试以及监视两项功能的系统，如图24-3所示。这里应当注意的是，该逆向工程框架目前只能用于Windows平台，而我们在本节以及本书的剩余部分中所关注的都是Intel IA-32体系结构。

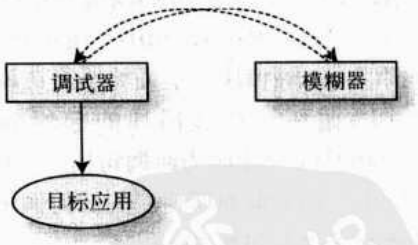


图24-3 模糊器/监视器体系结构

24.4.1 一个基本的调试器监视器

仍然考虑在本章中所设计的例子，而我们的目标是一个IMAP服务器。在我们马上将要创建的调试器的控制下，可以直接关联到该目标或者将其加载。该调试器与我们的模糊器之间存

⊖ https://www.openrce.org/downloads/details/208/PaiMei

在着一个双向的通信通路。可以采用许多方法来实现该通信回路。最方便的选择可能就是使用socket，因为这样我们不仅可以采用不同的语言来编写模糊器和调试器，并且可以潜在地将它们运行于单独的系统中，甚至是运行于不同的平台上。为了进行深入的研究，考虑下面针对调试器组件的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)

    # signal the fuzzer.

    print crash_bin.crash_synopsis()
    dbg.terminate_process()

while 1:
    dbg = pydbg()
    dbg.set_callback(EXCEPTION_ACCESS_VIOLATON, av_handler)
    dbg.load(target_program, arguments)

    # signal the fuzzer.

    dbg.run()
```

这段代码看起来似乎并没有什么，但是PaiMei框架所宣称的“简短扼要”的特点却是真的，上述的Python代码片段实际上实现了许多功能。你可能会从前面的章节中识别出前两行代码，即从PyDbg[⊖]的Windows调试器库中导入必要的功能和定义。然而，下一个所导入的库我们还没有看到过。PaiMei.utils[⊜]模块包含不同的逆向工程支持库。在这个例子中，我们将要使用crash_binning模块，后面将要对该模块进行详细解释。跳过几行代码，你将会发现一个无限的`while`循环。在该循环的起始部分，首先创建了PyDbg类的一个实例，并且将函数`av_handler()`注册为回调句柄以处理ACCESS_VIOLATION异常事件。只要一个无效的内存引用发生，例如前面所提到的任何例子，那么此注册函数将被调用。接下来，该调试器使用任意可选的参数来加载目标程序。

这时，必须要通过回路来向模糊器发送信号，以使其知道我们的目标应用将要接收一个模糊请求。在该循环的最后一行，调试器允许被调试对象通过成员函数`run()`（`debug_event_loop()`的别名）来执行。

上述代码中的其余部分是对`av_handler()`例程的定义。同任何注册的事件回调函数一样，该函数包含一个单一的参数，即当前的PyDbg调试器对象的实例。在该函数的起始部分，首先

⊖ http://pedram.redhive.com/PaiMei/docs/PyDbg/

⊜ http://pedram.redhive.com/PaiMei/docs/Utilities/

创建了一个前面所提到的PaiMei crash_binning构件的实例。使用该构件可以简化对某些属性的收集过程，这些属性是与当前的线程状态相关的不同的有用属性，在这个例子中，该线程就是触发我们正在处理的ACCESS_VIOLATION异常的线程。在下一行对record_crash()进行调用的代码中，直接记录了下面的这些信息：

- 异常地址。即异常所发生的地址。在这个例子中，就是导致ACCESS_VIOLATION异常发生的指令的地址。
- 写入违规。当一个标志被唤醒时，它表示在试图向一个内存位置写入的过程中，产生了一个异常。否则的话，它表明在试图从一个内存位置读取的过程中产生了一个异常。
- 违规地址。试图进行读取或者写入操作并且导致异常产生的地址。
- 违规线程ID。发生异常的线程的数字标识符。前4个参数组合在一起就允许我们做出如下的结论，如“在线程1234内部，位于地址0xDEADBEEF处的汇编指令将不能够从地址0xC0CAC01A处进行读取，并且将导致应用程序终止执行”。
- 上下文。发生异常时所保存的CPU寄存器的值。
- 反汇编。对导致产生异常的指令执行反汇编操作。
- 周围指令的反汇编。将在特定异常地址处导致异常发生的指令之前和之后的5条指令执行反汇编操作。通过对其周围的指令组合以及上下文信息进行研究，就能够使分析者更好地理解异常所产生的原因。
- 栈展开。异常发生时违规线程的调用栈。尽管这个信息在32位的Windows平台上并不总是可用的，但是当其可用时，它就向分析者显示了程序是如何到达目标地址的。检查栈展开的另外一种方法是动态地记录下目标的动作，具体见第23章“模糊器跟踪”中的相关讨论。
- SEH展开。异常发生时的结构化异常处理器（SEH）列表。依赖于我们是否正在处理调试器在应用程序前先看到的异常（稍后将对此进行详细讨论），一个分析者可以对此列表进行检查以发现哪些函数在目标自身内部被注册以处理异常。

这时，调试器应当再一次向模糊器发送信号，以使其知道是最后一个测试用例导致了一个异常，并且需要对其进行深入研究。模糊器可以记录下这个事实，然后等待来自于调试器的下一个信号，以表明目标已经被重新加载并且准备好接收下一个模糊请求。一旦调试器向模糊器发送了信号，它将输出一个人工可读的、在对record_crash()的调用中所捕获的所有元数据的概要信息，并且终止调试器的执行。while循环将负责重新加载目标应用。关于PyDbg的基本功能的更加特定的信息，可以参阅第20章“内存数据的模糊测试：自动化”。

64位Windows版本中的栈展开

当在不同的32位Windows平台上调试应用程序时，所面临的一个不幸的事实是在许多情况下，丢失了栈展开的信息。对栈进行展开所使用的一般方法非常简单。当前的帧起始于栈偏移量EBP。下一帧的地址是从指向EBP的指针中读取的。任意给定帧的返回地址被保存在EBP+4

中。可以从线程环境块（TEB）中重新获得栈的范围，用FS寄存器[⊖]FS[4]来引用以表示栈的顶层地址，并且用FS[8]来引用以表示栈的底层地址。一旦一个函数存在于不使用基于EBP的栈构成的调用链内部，那么该过程将会终止。

忽略帧指针的使用是编译器常用的一种优化方法，因为它释放了EBP寄存器，以使其可以被用做一个通用寄存器。然而，使用基于EBP构成的函数可以将局部变量作为静态EBP寄存器的偏移量而引用：

```
MOV EAX, [EBP-0xC]          ; EBP-based framing
MOV EAX, [ESP+0x44-0xC]     ; frame pointer omitted
```

如果这种优化方法在被展开的调用链内部所使用，那么关于真实调用链的信息将会丢失。微软通过用一个单一的调用规范来替代多种不同的调用规范，从而在新的64位的平台上部分解决了该问题，关于此调用规范的完整信息可以在MSDN上找到[⊜]。另外，每一个非叶子函数（一个叶子函数就是不调用任何其他函数的函数）现在都将静态地展开信息保存在可移植的可执行（PE）文件中[⊝]，这就使得任何时候都可以及时执行完整而准确的栈展开。

现在你已经得到了所需要的代码。将这个PyDbg脚本添加到你的指令集中，那么将会得到最先进的与当前的商业模糊器相绑定的模糊器监视软件。然而，我们还可以做的更好。当对软件进行模糊测试时，特别是对那些没有被很好地编写的软件进行测试时，你将会面临一个非常繁琐的工作，即在触发一个异常的潜在的成百上千个测试用例中进行分类和存储。通过深入研究，你将会发现许多测试用例所暴露的是相同的问题。换句话说，即模糊器发现了针对同一个漏洞的多种因素。

24.4.2　一个更加高级的调试器监视器

让我们再回到本章所讨论的IMAP例子，使用更加现实的情形来重新研究此问题。这里的场景如下：我们有一个能够生成50 000个唯一的测试用例的定制的模糊器。我们使用最近生成的调试器脚本来运行模糊器，并且发现有1 000个测试用例导致了一个异常的发生。现在，需要我们人工对这些测试用例中的每一个进行仔细观察，并且确定所发生的问题是什么。让我们首先来查看最初的几个测试用例，如下所示：

```
Test case 00005: x01 LOGIN %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
```

⊖ http://openrce.org/reference_library/files/reference/Windows%20Memory%20Layout,%20UserKernel%20-Address%20Spaces.pdf

⊜ http://msdn2.microsoft.com/en-us/library/7kcdt6fy.aspx

⊝ http://www.uninformed.org/?v=4&a=1&t=sumry

```
Test case 00023: x02 SELECT %s%s%s%s%s%s%s%s%s%s%s...%s%s%s
                 EAX=47392700 ECX=FFFEEF44...
                 EIP=0x00112233: REP SCASB
```

出于简化的原因，在上面的代码片段中只包含了一部分的上下文输出信息。在每一个所显示的测试用例中，由于试图从一个无效的内存地址（并没有显示出来）读取或者向其写入而导致产生了一个异常。只基于模糊输入而做出的一个合理假设使我们相信这个问题是由于解析长字符串的不正确的能力所造成的，或者是由于一个格式化字符串漏洞而造成的。注意当执行REP SCASB指令时，违规将发生在同一个地址，我们怀疑这种模式可能会存在于1 000个测试用例中的许多用例中。在进行了几个粗略的检查之后，该怀疑得到了证实。下面让我们来详细分析一下。SCASB IA-32汇编指令将扫描一个字符串，以搜索存储在第一个AL寄存器中的字节(EAX的第一个字节)。在上面所列出的每一个测试用例中，AL都是0。REP指令前缀将循环带有前缀的指令，并且当 ECX不等于0时逐渐清空ECX。在上面所列出的每一个测试用例中，ECX都具有一个较高的值。这是一种典型的汇编模式，以用于确定字符串的长度。许多的崩溃都是发生在扫描一个字符串以获得其终止空字节的过程中。

这是多么的令人沮丧啊。我们已经知道有1 000个测试用例导致了一个问题的产生，但是有许多的测试用例是在同一个位置或者是近似同一个位置发生的。手工从测试用例列表中进行选择是不切实际的（尽管可能适合于一个初学者）。还有更好的方法吗？当然有，否则的话我们也不会首先给你提出这个问题。

正如其名字所描述的那样，PaiMei crash binning构件将不只是简单地显示当前上下文的详细信息。该构件主要被用做一个持久化的容器以存储与许多崩溃相关的上下文信息。对record_crash()成员函数的调用将在一个内部维护和组织的列表中创建一个新的条目。每一个所创建的条目都包含其自己唯一的、相关的上下文元数据。这种组织方法将允许我们把前面的描述“50 000个测试用例中的1 000个导致了一个异常的发生”转换为更加有用的描述，如“在50 000个测试用例中，有650个在地址0x00112233处产生了一个异常，有300个在地址0x11335577处产生了一个异常，有20个在地址0x22446688处产生了一个异常”等等。对我们最初的脚本所做的修改非常简单，如下面代码中的黑体所示：

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

import utils
crash_bin = utils.crash_binning.crash_binning()

def av_handler (dbg):
    global crash_bin
    crash_bin.record_crash(dbg)

    # signal the fuzzer.

    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_VIOLATON, av_handler)

        dbg.load(target_program, arguments)

        # signal the fuzzer.

        dbg.run()
```

为了实现持久化，崩溃二进制文件的声明被移动到了全局命名空间中。每当一个新的异常被记录时也会添加相应的处理逻辑，来迭代已知的错误地址，并且显示在每一个地址处所监视的异常的数量。但这并不是显示这些信息最实用的方法，它在这里只是作为一个例子来说明问题。作为一个练习，可以试着对该脚本进行某些修改，以使得针对每一个违规地址来创建一个目录。在每一个目录内部，应当为每一个所记录的崩溃来创建一个文本文件，而该文件包含来自于crash_synopsis()的、人工可读的输出信息。

为了进一步提高其自动化程度以及分类水平，我们可以对crash binning构件进行扩展，以使其引用来自于每一个所记录崩溃的、可用的栈展开信息。在这种情形下，我们可以将数据存储结构从列表结构扩展为一个树结构。这就允许我们通过路径以及异常地址来捕获并且组合单个测试用例，考虑图24-4。

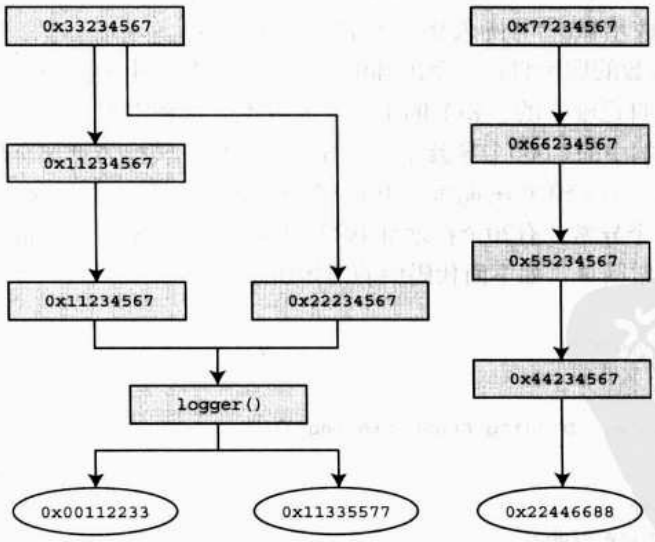


图24-4　Crash binning栈展开分类概念

在图24-4中，椭圆形框中的数据表示以前的异常地址类别。矩形框中的数据则表示来自于栈展开的地址。通过将栈展开数据添加到我们所考虑的范围内，我们就可以开始查看导致不同

错误地址的路径。从本质上来说，我们可以将最近一个结论如“在所有50 000个测试用例中，有650个在地址0x00112233处产生了一个异常，有300个在地址0x11335577处产生了一个异常”等等加以改进，从而形成“在所有50 000个测试用例中，有650个在地址0x00112233处产生了一个异常，而其中有400个测试用例覆盖了路径x,y,z，有250个测试用例覆盖了路径a,b,c”这样的结论。更为重要的是，根据此观点做出的分析将使我们有可能确定重复错误的根本来源。在我们目前的情形中，所捕获的异常大多数都发生在地址0x00112233和0x11335577处。到达这些地址的路径有成百上千条，但是出于简化的原因，这里只显示出了一部分路径。图24-4中到达上述两个地址的所有路径都执行了logger()例程。那么它是造成大多数问题的来源吗？可能在此例程中存在有一个格式化字符串漏洞。由于该例程在IMAP服务器内部许多地方都被调用过，因此它预示着将一个格式化字符串传递给多个IMAP动词将会导致出现同样的问题。尽管只有通过更加深入的分析才能真正探明其原因，但是我们可以立刻看到自动化分析所带来的好处。同样，实现这个增加的功能所需要进行的修改工作，作为一个练习留给读者来完成。

24.5 调试器在应用程序前先看到的异常和调试器再次看到程序未捕获的异常的比较

到目前为止，我们仍然没有讨论的关于异常处理的一个重要的概念就是，调试器在应用程序前先看到的异常和调试器再次看到程序未捕获的异常之间的比较。当在调试器下所加载的目标进程中发生一个异常时，微软的Windows操作系统会把异常作为调试器在应用程序前先看到的异常的通知来提交给调试器[⊖]。然后由调试器来选择是否将异常传递给被调试对象。如果异常被传递给了调试对象，那么可能会发生以下两种情形中的一种。如果被调试对象能够对异常进行处理，那么它将会处理异常并且执行将会正常继续进行。如果被调试对象不能够对异常进行处理，那么操作系统将再一次把异常提交给调试器，而这一次是作为调试器再次看到程序未捕获的异常的通知。所发送的异常是调试器在应用程序前先看到的异常还是调试器再次看到程序未捕获的异常，要取决于EXCEPTION_DEBUG_INFO[⊖]结构中的一个值。当使用PyDbg时，我们可以采用如下的方式在一个回调句柄中对该值进行检查：

```
def access_violation_handler (dbg):
    if dbg.dbg.u.Exception.dwFirstChance:
        # first chance
    else:
        # last chance
```

dwFirstChance的一个非0值意味着这是调试器第一次遇到该异常，并且被调试对象还没有获得机会来查看并处理该异常。

这对于我们要实现的目的意味着什么呢？它意味着我们需要做出一个决定，即是否将要忽略调试器再次看到程序未捕获的异常。例如，我们再一次对示例IMAP服务器进行审核，并且

⊖ -http://msdn.microsoft.com/msdnmag/issues/01/09/hood/

⊖ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50lrfexceptiondebug-info.asp

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书藉杂志全免 KEY 提供,，完全无加密。

如下的代码片段发现了logger()例程中的格式化字符串漏洞：

```
void logger (char *message)
{
    …
    try
    {
        // format string vulnerability.
        fprintf(log_file, message);
    }
    except
    {
        fprintf(log_file, "Log entry failed!\n");
    }
    …
}
```

在对易受攻击的fprintf()进行调用时，try/except语句创建了一个异常句柄。如果第一个fprintf()调用执行失败，那么第二个fprintf()将会被调用。如果在这种情况下，我们选择了忽略调试器再次看到程序未捕获的异常，那么我们将不能够检测到该格式化字符串漏洞，因为IMAP服务器很好地处理了它自己的错误。但这并不是说该漏洞不存在或者是不可利用的。简而言之，我们得到了一个漏报的信息。另一方面，也将会发现我们自身处于了这样一种情形，即我们所发现的大多数异常都被很好地加以了处理，并且都没有暴露出任何的安全问题。如果我们选择对调试器再次看到程序未捕获的异常进行处理，那么我们将要应对许多的漏报情况。作为一个独立的安全问题研究者，你不可能做出完全正确的决定。一旦你意识到该问题，可以对任何给定的目标应用经验测试以确定你将要采用漏报和误报的哪一种方法。作为通过测试来保证你自己产品的安全性的软件开发者或QA工程师而言，所做出的可靠决定应当是对调试器再次看到程序未捕获的异常进行监视，并且花费时间来研究每一种情形下异常的来源。

24.6　动态二进制插装

借助于调试器的监视技术可以为我们提供很大的帮助。然而，正如第5章“有效的模糊测试的需求”中所提到的，错误检测的最佳解决方案存在于DBI中。更加深入地讨论这个问题需要开发相应的原型工具，而这已经超出了本书的讨论范围。然而，本节中所提供的信息仍然有助于你更加清晰地理解目前可用的一些高级调试工具的内部执行机理。DBI引擎可以在最小的逻辑层次即基本块上插装一个目标可执行文件。

回忆一下第23章所描述的，一个基本块被定义为指令的一个序列，而一旦到达该基本块的第一条指令时，该块内部的每一条指令都将被确保按次序执行。DBI系统允许你通过添加、修改或者变换指令逻辑来在每一个基本块内部插装单独的指令。通常可以通过将执行控制流从初始指令集转换到一个修改过的代码缓存来实现这一功能。在某些系统中，可以使用更高层次的、类RISC的伪汇编语言来实现指令级的插入。这就使得开发者能够更加简单地开发DBI工具，并且试着使该工具能够跨平台使用。DBI系统所提供的API可以为我们提供范围很广的应用，从切片和最优化到机器翻译以及安全性监视。

目前存在有许多的DBI系统，其中的例子包括DynamoRIO㊀,DynInst㊁以及Pin㊂。例如DynamoRIO系统，它是由马萨诸塞技术研究所和惠普公司联合开发的项目。DynamoRIO是在IA-32框架中实现的，支持微软的Windows操作系统以及Linux操作系统。DynamoRIO非常稳定，并且具有很高的性能，这已经通过其在商业软件工具如Determina㊃的内存防火墙㊄中的应用得到了证实。有关DynamoRIO在Determina中的使用的更多信息可以参阅名为《基于程序监视的安全执行》（Secure Execution Via Program Shepherding）的MIT研究论文㊅。Pin DBI非常有趣，因为与大多数的DBI系统不同，它允许开发一个可以关联到目标进程的工具，而不是被局限于在DBI控制下加载目标进程。

利用DBI使得我们可以开发监视工具以进一步提高错误检测能力，即从检测一个错误的发生到潜在地发现错误的来源。再重新回到前面所使用的堆溢出的例子：

```
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，我们将无法检测到实际的溢出，因为它发生在对`strcpy()`的调用过程中。对于前面所讨论的栈溢出而言，情况也是如此。在栈溢出的情形下，当受影响的函数返回时，可能会生成并检测到一个异常。对于这种情形以及其他的堆溢出而言，将不会生成一个异常，直到后续的堆处理操作如对`free()`的调用出现在前面的代码片段中。基于DBI，我们就能够在运行时跟踪并记录所有的内存分配。术语*fence*表示标记或者记录每一个堆块的起始和结束位置的能力。通过注入所有的内存写入指令，就可以进行相应的检查以确保任何单独堆块的边界没有被超越。一旦一个单一的字节超越了一个所分配的堆块边界，那么就会发出一个通告。成功的实现这样一种技术可以为你节省大量的时间、分析精力以及财力。

开发一个定制的DBI工具的学习曲线并不非常容易。幸运的是，在你开始创建该工具之前，可以利用其他许多工具来实现模糊器集成。可用的商业工具包括IBM Ratinal Purify㊆，Compuware DevPartner BoundsChecker㊇,OC Systems RootCause㊈以及Parasoft Insure++㊉。例如Purify工具，它实际上是构建于静态二进制注入工具（SBI），而BoundsChecker则是建立在SBI和DBI组合的基础之上。这两种工具都可以提供你所期望的错误检测以及性能分析特性集。

㊀ http://www.cag.lcs.mit.edu/dynamorio/
㊁ http://www.dyninst.org/
㊂ http://rogue.colorado.edu/pin/
㊃ http://www.determina.com
㊄ http://www.determina.com/products/memory_firewall.asp
㊅ http://www.determina.com/products/memory_firewall.asp
㊆ http://www-306.ibm.com/software/awdtools/purify/
㊇ http://www.compuware.com/products/devpartner/visualc.htm
㊈ http://www.ocsystems.com/prod_rootcause.html
㊉ http://www.parasoft.com/jsp/products/home.jsp?product=Insure

在开源工具领域中，你可能会遇到的最著名的解决方案就是Valgrind[⊖]（其中“val”表示“value”，“grind”表示“grinned”）。Valgrind为你提供了DBI系统以及许多前面所开发的工具如Memcheck，该工具可以被用来定位内存泄漏和堆溢出漏洞。Valgrind也可以使用许多第三方控件。对于我们这里的特定目的而言，边界检查版本的实现Annelid[⊜]是最为重要的。

对于大多数情况而言，借助于调试器自动化错误监视的使用已经是对其当前的模糊测试方法进行了改进。但是，我们仍然鼓励你对这些高级的错误检测方进行更加深入的研究。

24.7　小结

在本章中，我们讨论了模糊器目标监视方法，包括从最基本的高层技术到更加高级的借助于调试器的技术。同时，我们也涉及到了DBI的概念，并且介绍了不同的DBI系统，你可以利用一些立即可用的技术以实现更加高级的目标监视。

通过对本章所介绍知识的学习，你应当可以将定制的调试器监视工具与高层的模糊器单步执行工具所具有的功能加以结合。通常情况下，这种组合可以准确确定是哪一个单独的测试用例或者是测试用例的组合导致了目标应用发生问题。并且在相应的自动化工具支持下，将会为你减少模糊器执行之后的分析工作量。

⊖ http://valgrind.org/

⊜ http://valgrind.org/downloads/variants.html?njn

第四部分 展 望

第25章 汲取的教训

"很少问到这样的问题：孩子们正在学习吗？。"

——George W. Bush，Florence，SC，2000年1月11日

我们期望此时可以清晰地描绘出模糊测试究竟是什么，为什么它是一种有效的测试方法以及如何使用它在程序代码中发现隐藏的错误。我们在前面提到过，本书主要面向于可以从模糊测试中得到收益的三类不同的目标读者群：开发者、QA团队成员以及安全问题研究者。在本章中，我们将对软件开发生命周期（Software Development Lifecycle,SDLC）进行分析，以确定这三类读者分别可以在软件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

模糊测试曾经一度是安全问题研究者在软件产品开发完成之后所专用的一种技术，但是现在软件开发者也已经开始应用模糊测试技术，以在软件开发生命周期的早期发现漏洞。微软已经在其可信计算安全开发生命周期[⊖]中将模糊测试作为一个关键的组成部分。这种方法鼓励开发者在他们称之为安全开发生命周期（Security Development Lifecycle,SDL）的实现阶段中"应用安全性测试工具以及模糊测试工具"。微软已经定义了SDL的相关概念，图25-1显示了它和SDLC各阶段之间的对应关系。

你从上述这些并行的过程中可以看到，微软已经开始在其SDLC的每一个阶段中考虑识别适当的安全性问题。这是在集成代码中很重要的一个方面，即安全性必须要贯穿于SDLC的整个过程中。在上图中没有反映出来但必须要实现的就是将安全性渗透到SDLC中，但不只是作为一个并行过程。开发者、QA团队以及安全问题研究者必须要协同工作以实现开发安全代码的共同目标。

微软对安全性的关注

微软正在将安全性紧密结合到其软件开发生命周期中，这并不奇怪。由于在市场份额中居于主导地位，微软的许多技术成为了安全问题研究者的研究目标。尽管还存在着许多争论认为微软要达到安全研究领域的前沿还有很长的路要走，但毋庸置疑的是，微软已经在加强其软件产品的安全性方面迈出了很大的步伐。

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

作为一个例子，考虑微软的Internet信息服务（Internet Information Services,IIS）㊀，即微软的Web服务器。目前有14个公开发现的漏洞影响了IIS的5.x版本㊁。在2003年早期所发布的6.x版本，只被3个已知的错误所影响㊂，并且没有一个被认为是严重的错误。安全性的改进要部分归功于低层的安全性保证方法，如/GS缓冲区安全检查㊃、数据执行预防（DEP）、安全的结构化异常处理（SafeSEH）㊄以及受到最广泛期待的一个Windows Vista的安全性保证技术，即地址空间的随机规划（Address Space Layout Randomization,ASLR）㊅。除了上述这些安全性保证技术之外，微软还通过象微软产品安全小组㊆这样的计划来投入人力资源以保证软件的安全性。模糊测试就是这部分开发者所使用的多种技术之一，用以加强所开发软件的安全性。

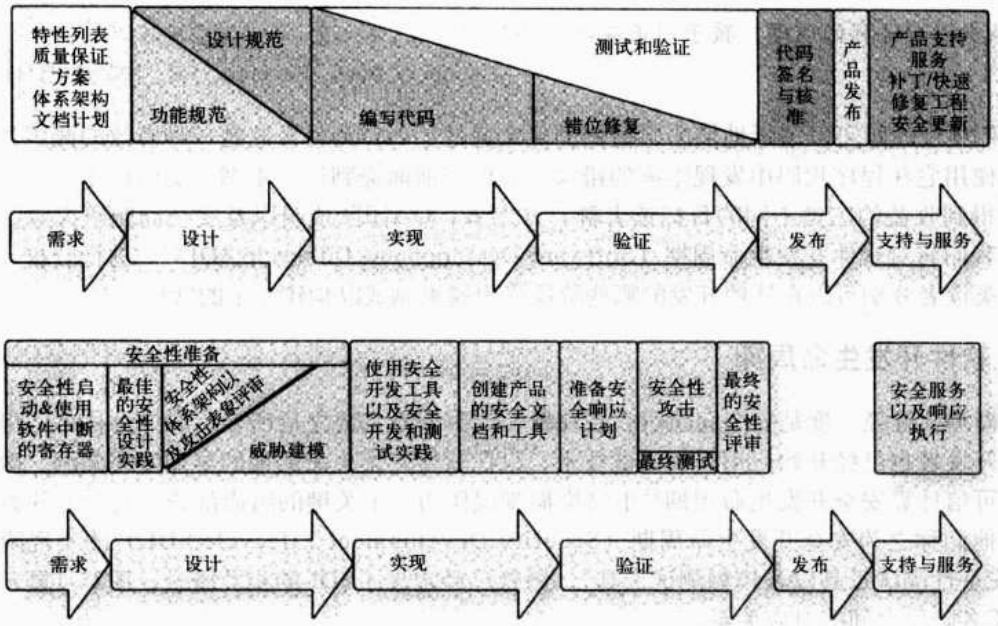


图25-1　微软的SDLC和SDL之间的对应关系

SDLC方法不存在什么不足，但是出于我们要到达的目的，我们在这里将使用Winston Royce初始的瀑布模型㊇，因为它非常简单并且被广泛接受。瀑布模型将一种连续的方法应用于软件开发，在软件的整个生命周期中一共经历了5个不同的阶段。图25-2说明了该方法。

㊀ http://www.microsoft.com/WindowsServer2003/iis/default.mspx
㊁ http://secunia.com/product/39/?task=advisories
㊂ http://secunia.com/product/1438/?task=advisories
㊃ http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx
㊄ http://en.wikipedia.org/wiki/Data_Execution_Prevention
㊅ http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf
㊆ http://www.microsoft.com/technet/Security/bestprac/secwinin.mspx
㊇ http://en.wikipedia.org/wiki/Waterfall_process

下面，我们将讨论在上述每个阶段中如何实施模糊测试。

25.1.1 分析

分析阶段是在开始开发一个项目之前，收集所有需要的信息的过程。这个过程包括同终端用户一起来讨论，以准确确定用户的需求。尽管模糊测试在SDLC的这个阶段可能不会发挥一个直接的作用，但是它对于开发者、QA团队和安全问题研究者开始考虑是否将模糊测试作为后续过程的一个适当的测试工具是非常重要的。

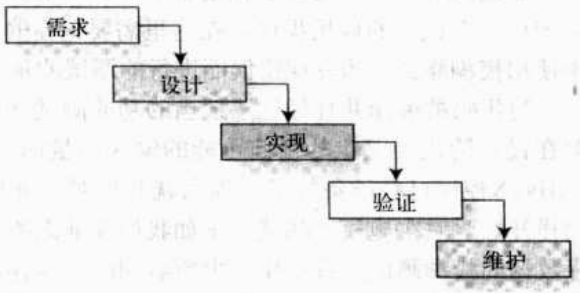


图25-2 Royce的初始瀑布模型

25.1.2 设计

对一个应用程序进行设计包括生成该项目的一个抽象的表示。依赖于被建模的整个解决方案的不同部分，如硬件、软件或者数据库构件，在这个阶段中可以使用多种不同的建模技术。这个阶段将要确定在编码阶段所使用的编程语言和软件库。

当做出设计决策之后，关于模糊测试的可行以及可能使用的方法开始成为关注的焦点。在这个阶段有两个重要的细节将被定义，它们会影响到整个的模糊测试方法。首先，要确定将被实现的软件和硬件平台。这将会影响能够被使用的模糊测试工具，并且影响将被应用的模糊测试类。例如，如果该项目是一个被设计为运行在Linux操作系统中的控制台应用程序，那么采用环境变量模糊测试将更加合适。如果该项目被设计为运行在Windows操作系统中，那么标记为“编写脚本安全”的ActiveX控件是否将向外部的Web站点提供函数呢？如果是这种情况，那么就应当应用COM对象的模糊测试。

在SDLC的这一阶段中，可能将会选择处理通信的网络协议。例如，如果在该特定项目的需求中，要求采用一种内嵌的即时消息解决方案，那么可扩展的消息处理现场协议(Extensible Messaging and Presence Protocl,XMPP) 可能就会被选中。这个协议是一种开放的XML通信技术，它是1999年由开源组织在Jabber[⊖]项目中开发的。这是在SDLC中将协议规范传递给模糊测试团队的一个很好的阶段。同时也可以在这个阶段中分配一些资源，以搜索并枚举出当前可用的XMPP测试工具以及前面所发现的XMPP解析漏洞。

在设计阶段中所做出的另外一个重要的决定与将要在应用程序中被开发的输入向量相关，需要记住的是，模糊测试是这样一种简单的技术，即它向一个应用程序提供非常规输入，并且观察所发生的情况。因此确定所有这样的输入向量以及如何最佳地利用它们进行模糊测试是一个非常关键的问题。在准确确定输入向量的构成时，进行创造性的思考也非常重要。仅仅因为某些操作是在内部进行的而没有直接暴露给一个终端用户，这并不能表明一个攻击者不能创建定制代码以访问接口。所列举的输入向量越完整，那么使用模糊测试的代码覆盖就可以完成的更好。

⊖ http://www.jabber.org/

25.1.3 编码

在编码阶段，开发者要构建整个项目所需要的各个组成部分，并且逐步将它们集成在一起以形成一个统一的应用程序。在这里需要记住的重要一点是，开发者可以并且应当在开发过程中使用模糊测试。为发现错误而进行的测试的重担不应当只由QA团队和安全专业人士来承担。

当代码被编译并且经过了适当的功能测试之后，也应当对它们进行足够的安全性测试。针对在设计阶段所确定的任何可能的输入向量而言，都可以应用模糊测试。如果所提供的一个ActiveX控件已经准备好了，那么就开始对它进行模糊测试。如果完成了一个网络服务，那么同样开始对它实施模糊测试。正如我们在前面的第19章“内存数据的模糊测试”和第20章“内存数据的模糊测试：自动化”中所看到的，内存数据模糊测试同样可以被应用于函数层次。如果一个给定的函数被设计为从一个用户提供的字符串中抽取出一个邮件地址，那么就使用许多随机生成以及智能选择的字符串值来对该函数进行上千次的调用。

不完善的输入验证往往是导致安全漏洞的原因，而模糊测试将有助于开发者更好地理解为什么输入验证如此重要。模糊测试被设计为要识别这样的输入，即开发者没有考虑到以及应用程序不能够处理的输入。当开发者发现这种情形时，他们将在最佳的位置来构建适当的输入验证控件，并且确保在后续代码中不再出现同样的错误。更为重要的是，漏洞在SDLC中被发现的越早，那么修改这些错误的开销也就会越少。当一个开发者发现其自己代码中的错误时，就会更早的修复它们。

25.1.4 测试

当软件开发进入到测试阶段时，QA团队就开始发挥作用了。在项目发布之前，安全保证团队也应当参加到测试阶段中。如果缺陷在软件发布之前被发现，那么整个开销就会越少。另外，你可以通过确保漏洞都是在内部被发现，而不是首先出现在一个公开的邮件列表中，来维护公司的声誉。

对于QA团队和安全研究者而言，确定如何在测试阶段最佳地协同工作以更好地相互利用资源是非常重要的。对于某些软件提供商而言，在SDLC中包含安全问题专家可能是一个新的方法，但逐渐增多的被公开发现的漏洞也说明了包含安全专家是非常必要的。

QA团队和安全研究者都需要认识到在测试阶段可以应用的商业的和开源的模糊测试工具。这是可以实现对等培训任务的地方。安全研究者更有可能掌握最新最好并且可用的模糊测试工具的知识，他们可以利用这些知识来对QA团队和开发者进行模糊测试工具的培训，而这些工具可以被用来提高安全性。作为该培训的一个副产品，开发者能够认识到他们可以避免常见的编程错误，并且能够更好地理解安全编程实践。

25.1.5 维护

尽管开发者、QA团队和安全研究者已经做出了最大的努力，但是仍然可以在最终的产品代码中发现导致简单异常以及重要安全漏洞的错误。主要的软件提供商都已经在安全领域中投入了数十亿美元，尽管情形有所改善，但是我们仍将面对包含有漏洞的代码。因此，即使生成了产品代码，继续对安全漏洞进行研究是非常重要的。新的模糊测试工具、经验丰富的研究者以

及改进的模糊测试技术都可能对前面测试过的代码进行更新的检测，并且发现新的漏洞。另外，漏洞可能会具有依赖性的被实现，因此在代码作为产品发布之后由于配置的改变使得漏洞就会出现。作为一个例子，某些问题可能不会在32位平台上出现，但是将会在64位平台上出现。

除了在开发团队内部通过努力积极识别漏洞代码之外，在适当的时候对独立的研究者做出响应以及同其协同工作也是非常重要的。独立的研究者负责发现软件中的大多数公开的安全漏洞。尽管这些研究者发现漏洞的目的各不相同，但是他们所做出的努力对于生成更加安全的产品代码是非常重要的。软件提供商应当在适当的时候提供这样的过程，即允许独立的研究者向开发团队报告他们的错误发现情况，以作为连接研究者和开发者之间的桥梁，从而最终实现对错误的修复。

25.1.6 在SDLC中实现模糊测试

我们认识到瀑布模型所展现的是一种过度简化的软件开发过程。大多数的项目都需要多个开发团队来协同工作，并且SDLC的各个阶段是循环迭代的出现，而不是瀑布模型中所描述的纯粹的顺序执行过程。无论如何，任何的开发模型都是将这些基本的开发阶段进行适当的组合形成的。不要只关注于模型本身，而是要利用本章所讨论的知识在你自己的SDLC模型中实现模糊测试，以在产品发布之前更好地保证代码的安全。

25.2 开发者

当要在新的项目中实现安全性要求时，开发者是第一道防线，并且当他们开发了新的函数、类以及库时，应当能够对这些对象进行模糊测试。通过培养开发者认识到基本的安全问题的技能，就可以立刻解决许多开销很大的安全问题，而不再将问题留给QA或者安全保证团队，因此留给公众用户去发现的问题也就更少。

通常情况下，开发者是在他们所熟悉的一个特定的集成开发环境（IDE）中来完成其大部分编程工作的。例如微软的Visual Studio，是在Windows平台上进行C#和Visual Basic编程的标准的IDE，而Eclipse则是使用Java和多种其他语言进行编程的一个主流的IDE。鼓励开发者在其开发过程实践中结合模糊测试的一个推荐方法是开发IDE插件，而不是要求开发者去学习一个全新的工具集以进行安全测试。尽管SPI Dynamics的DevInspect[⊖]工具不是一个特定的模糊测试工具，但是它却提供了此方法的一个例子。DevInspect是扩展了Visual Studio和Eclipse功能的一个工具，使开发者能够对ASP.Net和Java Web应用程序进行源代码分析以及黑盒测试。

25.3 QA研究者

在传统意义上，QA只关注于功能测试而不是安全测试。幸运的是，由于终端用户逐渐认识到了不安全的软件所带来的危害，并且向其软件提供商提出了更多的安全方面的要求，因此使得这种情况得到了改变。尽管不应当期望QA团队成为安全问题的专家，但是应当期望他们对安全性具有一个基本的了解，更为重要的是，他们要知道何时需要引入安全问题专家。模糊

⊖ http://www.spidynamics.com/products/devinspect/

测试是一种非常适合于QA团队的测试过程，因为它可以实现高度的自动化。例如，逆向工程对于QA研究者而言将不是一个适于掌握的技能，因为它是一种高度专业化的技能，并且需要经过大量的培训。另一方面，执行模糊测试工具对于QA团队来说是一项非常现实的工作。尽管这可能需要同安全保证团队进行协同工作以确定一个特定的异常是否会导致一个可利用的条件，在测试阶段中，任何类型的崩溃都需要进行评审，并且应当被开发者所解决。这时，安全性和功能性之间的区别已经不再重要。而重要的是解决目前所遇到的问题。

25.4　安全问题研究者

模糊测试对于安全问题研究者而言并不是一个新的概念。由于模糊测试所具有的相对简单性以及高回报率，使其成为了一种被广泛使用的用于发现安全漏洞的方法。在过去的几年中，安全问题研究者根本就没有被包含在SDLC中。并且目前在许多开发组织中仍然是这种情形。如果你在SDLC中没有使用安全问题专家，那么就需要对你的软件开发过程重新进行评审了。例如微软，它通过主办BlueHat安全简报[⊖]，从著名的安全问题研究者那里获得相关的知识，以武装其员工的头脑。

安全问题研究者所面临的挑战是，将其所关注的焦点转移到在SDLC的早期发现漏洞。不论漏洞是何时被发现的，只要对其进行修复就是有意义的，但是随着项目开发阶段的不断进行，修复漏洞的开销也在不断增加，如图25-3所示。安全性团队可以利用特殊的技能以调整测试过程，从而持续地改进所使用的模糊测试工具和技术。尽管QA团队应当负责执行标准的模糊测试工具和脚本，但是实际上是安全问题研究者来构建定制的工具或者识别新的技术，并且持续的对QA团队和开发者进行相关的培训。

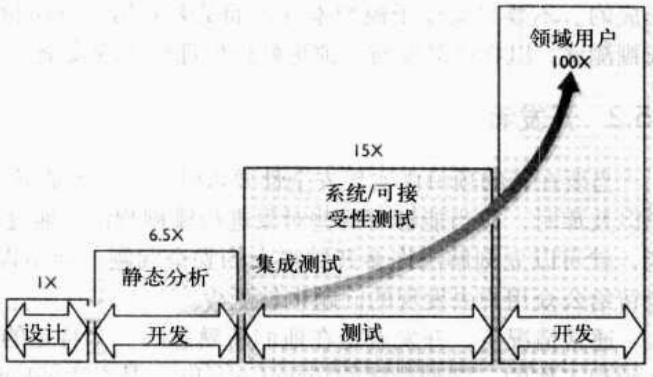


图25-3　在SDLC中修复软件缺陷的开销

25.5　小结

任何一个软件开发者可以说“安全性不是我的问题，我们有一个团队来考虑安全问题”的时代已经过去了。安全性是每一个人的责任，包括开发者、QA团队以及安全问题研究者。作为一个产品经理，不能够满足于简单的忽略漏洞，并且希望这些漏洞将会神奇地消失。你的员工也必须要使用相应的过程和工具，以将安全性嵌入到SDLC中，并且模糊测试代表了一个可以被上述三类人所利用的自动化的过程。模糊测试并不是一个万能钥匙，但是它是一种可以开发用户友好的安全性工具的方法，从而可以发现大量的漏洞。

⊖ http://www.microsoft.com/technet/security/bluehat/sessions/default.mspx

溜客资源网 WwW.176Ku.CoM，1000G 无加密版教程书籍杂志全免 KEY 提供,，完全无加密。

第26章 展 望

"但是总的来说，对我和劳拉这是难以置信的一年。"

——George W. Bush,在发生9·11攻击三个月之后，
对其上任第一年的工作进行总结，Washington，DC，2001年12月20日

模糊测试将向什么方向发展呢？其应用领域已经开始从学术研究转向了企业级的测试床，并且贯穿于SDLC的整个过程中。由于有更多的软件开发者接受了模糊测试，因此有许多商业工具被开发以利用这种功能强大的方法也就不足为奇了。在本章中，我们将讨论模糊测试的现状，并且预测其进一步的发展方向。

26.1 商业工具

可以证明，表明一种工业成熟度的强有力的标志就是商业工具在该领域中的应用。这也是我们当前对模糊测试技术所要做出的证明。大量的软件开发者如微软都已经接受了模糊测试，并且将其作为在SDLC的早期发现安全漏洞的一种方法。同时还将会出现新的产品和公司以满足开发健壮的、用户友好的模糊器的需求。在本节中，我们将介绍目前存在的一些商业化工具，这些工具是以字母顺序来排列的。

26.1.1 安全性测试工具beSTORM[⊖]

beSTORM是利用模糊测试技术的一种黑盒测试工具，用于在多种不同的基于网络的协议中发现漏洞。beSTORM安全性最初是来自于所创建的公开的Web站点SecuriTeam[⊜]，其目的是发布与安全相关的新闻和资源。beSTORM所能够测试的协议如下所示：

- HTTP－超文本传输协议
- FrontPage扩展名
- DNS－域名系统
- FTP－文件传输协议
- TFTP－一般的文件传输协议
- POP3－用于电子邮件接收的邮局协议，版本3
- SIP－会话启动协议
- SMB－服务器消息块
- SMTP－简单的邮件传输协议
- SSLv3－安全Socket层，版本3

⊖ http://www.beyondsecurity.com/

⊜ http://www.securiteam.com/

• STUN－通过网络地址解析器（NAT）进行的简单的用户数据报协议（UDP）的传输

• DHCP－动态主机配置协议

beSTORM可以运行在Windows、UNIX以及Linux平台上，并且具有单独的测试和监视组件[⊖]。我们希望在下一代的商业模糊测试工具中包含目标监视组件。对于beSTORM而言，其中的监视组件只是作为一个定制的调试器而创建的，并且局限于UNIX和Linux平台。图26-1显示了beSTORM的一个屏幕快照。

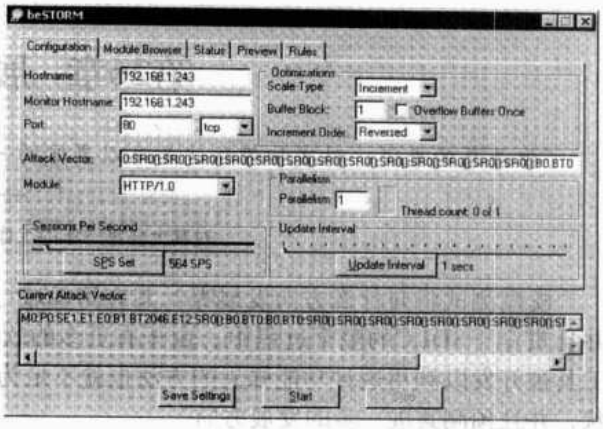


图26-1　安全性测试工具beSTORM

26.1.2　BreakingPoint系统BPS-1000[⊜]

作为该领域最新的一个工具，BreakPoint在纯通信的市场中占有明显的统治地位。BPS-1000是一个定制的硬件解决方案，它能够以每秒钟生成500 000个TCP会话的速度生成超过5百万个TCP会话。尽管从传统意义上来讲它并不是一个模糊器，但是BPS－1000能够对所生成的通信进行变异，从而实现检测错误的目的。

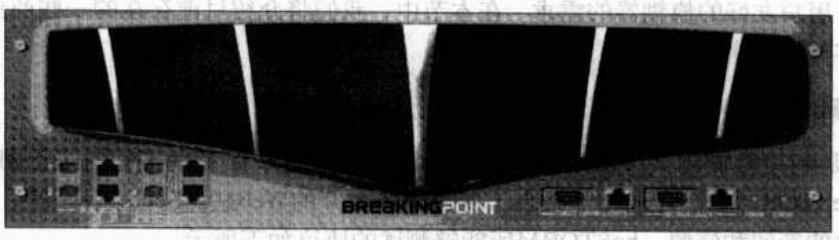


图26-2　BreakPoint系统BPS－1000

BPS－1000最有趣的特性之一是前端的AC适配器，其作用是当目标测试设备无法继续进行测试时，它负责向其提供动力循环。

26.1.3　Codenomicon[⊝]

可以证明，Codenomicon提供了最广为人知的软件模糊测试解决方案。Codenomicon最初来源于PROTOS[⊛]测试套件，该套件最初是由芬兰Linnanmaa的Oulu大学所发起的一个项目所开发的。PROTOS首次获得广泛的公开是在2002年后期，当时该项目发布了一个令人惊讶的、

⊖ http://www.beyondsecurity.com/beSTORM_FAQ.pdf

⊜ http://www.breakingpointsystems.com/

⊝ http://www.codenomicon.com/

⊛ http://www.ee.oulu.fi/research/ouspg/protos/

关于在多种不同的SNMPv1实现中的漏洞列表。PROTOS的开发者对SNMPv1协议进行了解析，并且列举出了针对请求和捕获包的所有可能的变换。然后，他们开发了一个实际的事务集合并引入了异常元素的概念，他们将该术语定义为“在实现软件时可能不会被认为是一个正确的输入”[⊖]。在某些情况下，这些异常元素违反了协议标准，而在其他情形中，它们与协议规范是一致的但是却包含破坏解析器的内容。PROTOS SNMPv1测试套件的执行结果中保存了非常重要的信息，因为它们识别出了大量的漏洞，而这将会导致许多受影响的硬件和软件提供商发布补丁程序[⊖]。PROTOS项目从此之后就不断地发展，目前它包含针对多种不同的网络协议和文件格式的测试套件，其中所涉及到的协议如下所示：

- WAP－无线应用协议
- HTTP－超文本传输协议
- LDAPv3－轻量级目录访问协议，版本3
- SNMPv1－简单网络管理协议，版本1
- SIP－会话启动协议
- H.323－视频会议中通常使用的一组协议的集合
- ISAKMP－Internet安全联盟密钥管理协议
- DNS－域名系统

该商业产品内部所使用的测试方法在很大程度上并没有偏离最初的PROTOS。最为特别的是，该商业测试套件覆盖了大量的网络协议和文件格式，并且还提供了一个友好的GUI界面，如图26-3所示。



图26-3 Codenomicon域名服务器测试工具

⊖ http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/index.html#h-ref2

⊖ http://www.cert.org/advisories/CA-2002-03.html

Codenomicon以溢价为每一个单独的协议提供了一个独立的产品。在撰写本书时，每一个协议测试产品的零售报价大约是30 000美元。这种定价策略表明该公司所面向的是那些只需要少数几个针对其所研发产品类型的协议测试套件的软件开发者，而不是对测试范围很广的许多产品和协议都感兴趣的那些纯粹的安全性研究机构。该产品最大的局限就是缺乏目标监视功能。Codenomicon在测试过程中利用已知的测试用例来获取目标的状态。正如在第24章“智能化错误检测”中所讨论的，这是一种最基本的监视技术。

26.1.4　GLEG ProtoVer Professional[⊖]

GLEG是一家由于提供了VulnDisco[⊜]而建立起来的俄罗斯公司，VulnDisco是一个包含每月发布新发现的漏洞的服务，而发布的形式是针对Immunity公司的CANVAS[⊜]可利用框架的开发模块。后来GLEG将该工具进行了扩展，以包含其自主开发的、被用来发现由VulnDisco产品所提供的许多漏洞的模糊器。扩展后的产品被称为ProtoVer Professional，它是采用Python编程语言来开发的。在编写本书的时候，该模糊器可以支持如下的协议：

- IMAP－Internet消息访问协议
- LDAP－轻量级目录访问协议
- SSL－加密套接字协议层
- NFS－网络文件系统

该测试套件包含多种不同的界面，如命令行、GUI以及基于Web的界面，如图26-4所示。ProtoVer是由安全问题研究者编写的，并且面向与Codenomicon相同的市场，其目前的定价为

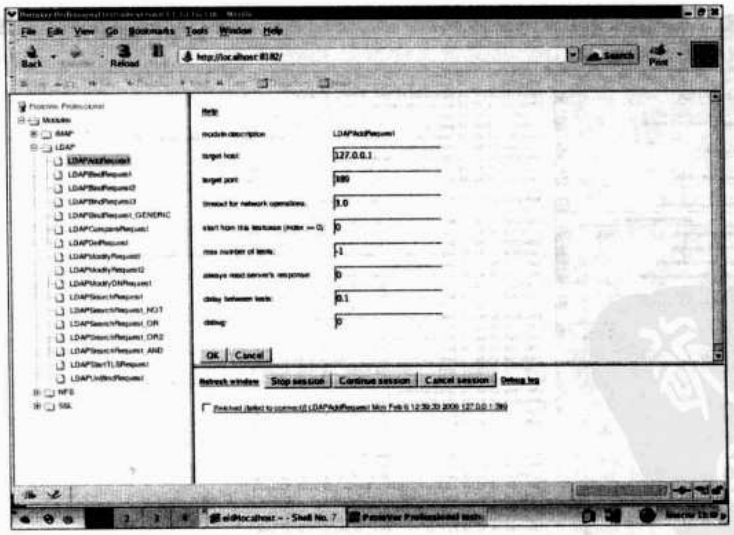


图26-4　GLEG ProtoVer专业级测试套件的Web界面

⊖ http://www.gleg.net/protover_pro.shtml

⊜ http://www.gleg.net/products.shtml

⊜ http://www.immunitysec.com/products-canvas.shtml

4 500美元，目的是为所有支持的协议获得一个年度许可。

26.1.5 安全性测试工具Mu-4000[⊖]

Mu Security所提供的Mu－4000是一个硬件工具，它能够对网络通信进行变异并且可以监视目标以发现错误。Mu-4000主要被设计为测试其他的网络设备，类似于BreakingPoint解决方案，它能够为目标设备提供动力循环以从所检测到的错误中进行恢复。Mu-4000的另外一个有趣的特性是当一个错误被以基于Linux的可执行文件的形式被Mu认为是一个外部漏洞触发器发现时，Mu4000能够自动生成一个可利用的概念验证工具。Mu声称其设备可以处理任意的基于IP的协议。图26-5显示了详细描述在一个DHCP实现中所发现异常的一个报告示例。

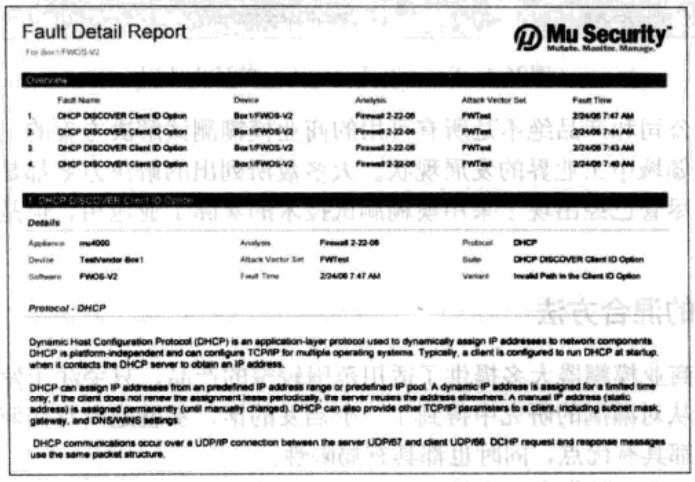


图26-5 Mu-4000所生成的详细错误报告

26.1.6 Security Innovation Holodeck[⊖]

Security Innovation的Holodeck是这样一种唯一的软件解决方案，它允许开发者通过模拟非预期的错误，从而研究针对微软Windows平台所开发的软件中所实现的安全性和错误处理器的健壮性。这个产品提供了典型的模糊测试功能，如对文件和网络流进行变异的能力。在其所具备的许多唯一特性中，Holodeck能够触发下列错误：

- 当试图访问特定的文件、注册表健和值以及COM对象时所产生的资源错误。
- 由于受限制的磁盘空间、内存以及网络带宽而引起的系统错误。

Holodeck具有一个比较合理的价格标签，针对单个用户许可为1 495美元，并且提供了一个开放的API，允许高级用户在该产品的基础上创建自动化工具和扩展。图26-6显示了Holodeck主界面的一个屏幕快照。

⊖ http://www.musecurity.com/products/mu-4000.html

⊖ http://www.securityinnovation.com/holodeck/

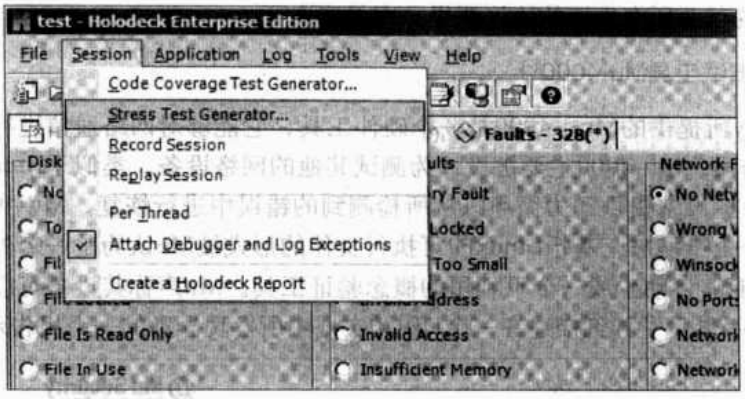


图26-6　Security Innovation的Holodeck

上面所列出的公司和产品绝不是所有可用的商业模糊测试解决方案的一个完整的列表。然而，它表明了在该领域中工业界的发展现状。大多数所列出的解决方案都是相对较新的“第一个版本”的技术。尽管已经出现了采用模糊测试技术的实际工业应用，但是其发展相对来说还很不成熟。

26.2　发现漏洞的混合方法

我们所讨论的商业模糊器大多提供了适用范围较窄的产品，只关注于发现漏洞的一个特定的方法。如果我们从对漏洞的研究中得到了一个启发的话，那就是没有任何一个单独的方法是银弹。所有的方法都具有优点，同时也都具有局限性。

了解了这一点，我们就期望安全性工具的开发要结合多种分析方法。这里所要实现的目标是要创建这样一个解决方案，即各部分的功能性能之和要大于整体的功能性能。例如，可以考虑将源代码分析同模糊测试相结合。模糊测试所面临的一个挑战就是要确保足够的代码覆盖。在一个纯黑盒测试场景中，如何来确保已经对所有可能的输入都进行了模糊测试呢？在第23章中，我们讨论了将代码覆盖作为一种度量手段以确定模糊测试的完成程度。考虑一下另外一种可能的方法，即使用混合分析的方法。这种分析方法首先要进行一个自动化的源代码评审。这种典型的白盒测试方法通常将会发现大量的潜在的漏洞，但是要受到误报的影响，因为它在运行时不能确认结果的最终形式。这个局限性可以通过下面的方法来解决，即利用静态扫描的结果来帮助生成模糊测试用例，从而有助于在审核的动态模糊测试阶段中证明或驳斥那些潜在的漏洞。我们希望模糊测试成为混合型安全保证技术中的一个集成部分。

26.3　集成的测试平台

安全专家将会一直创建并且接受新的改进的、一次性的以及单机版的模糊器。然而，QA和开发团队将很少有时间去研究那些复杂的、前沿的安全性工具，因为安全测试总是在扮演一个辅助的角色，而首先要满足的是开发需求和基线。为了使模糊测试技术在QA和开发团队中被广泛接受，它必须要面向这些团队已经熟悉并且经常使用的现有平台。我们期望看到模糊测

试功能被集成到开发环境中，如微软的Visual Studio和Eclipse。同样，由IBM和Mercury这样的提供商所提供的主流的QA平台也可以从模糊测试功能中获得好处。这些功能可以由提供商提供，也可以由这些应用程序的维护者来提供，但最初通常是由第三方所开发并且作为一个应用程序插件来提供的。尽管SPI Dynamics的DevInspect和QAInpect产品不是一个专用的模糊器，但它可以作为此方法的一个早期的例子。

26.4 小结

工业界发展的前沿是什么呢？前面已经暗示过，对模糊测试技术更为广泛的接受正处于发展的起点。我们鼓励这种发展趋势，并且基于许多原因将模糊测试看做是安全领域中唯一的方法。更为重要的是，其他可选的方法（如二进制逆向过程和深度的源代码分析）都需要具备特殊的技能，而这对于开发者和QA团队而言是不现实的。另一方面，模糊测试可以被实现为一个自动化的过程，因此开发者和QA团队都使用这种方法。

尽管独立的安全问题研究者将会持续推动该问题向前发展，并且开发有趣的模糊测试技术，但主要还是要靠商业提供商的努力以创建第一个易于使用的模糊测试环境，并且能够顺利地集成到开发环境中。正如第24章所讨论的，其中最重要的一个需求就是错误检测的先进性。调试技术很少与目前可用的模糊器绑定在一起。即使将调试器与模糊器绑定在一起，那么也并不是最佳的选择。必须要应用更加先进的技术以实现对所发现的错误进行准确定位和自动化分类。错误检测工具提供商如第24章所提到的，包括IBM、Compuware、Parasoft和OC Systems，都具有唯一的目的即扩展并完善其工具的功能。

通过对类似的、但更加成熟的技术的发展历史进行研究，我们可以期望看到一些关键事件的发生。首先，将会有更多的公司参与到同早期开发模糊测试工具的公司的竞争中来。提供商可以从头开发他们自己的工具，但是在许多情况下将需要某些已有的商业技术。有哪些公司可能会参与到这个竞争中来呢？很难给出确定性的回答，且看一下处于安全性和QA竞争空间中的软件提供商可以通过接受模糊测试技术，从而利用这个机会使其与竞争者区别开来。