iOS应用逆向与安全

刘培庆 / 著

电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书从正向开发、工具的使用、逆向实战及安全保护等方面，对 iOS 应用的逆向与安全进行了讲解。本书内容包括基本概念、逆向环境的准备、常用逆向分析工具、类的结构、App 签名、Mach-O 文件格式、hook 原理等，并通过在越狱平台和非越狱平台上的逆向分析实例，带领读者学习逆向分析的思路和方法。在应用安全及保护方面，本书内容涉及网络传输、安全检测、代码混淆等。

本书适合高校计算机相关专业的学生、iOS 开发工程师、逆向工程师、越狱开发工程师、iOS 安全工程师及应用安全审计人员阅读参考。

图书在版编目（CIP）数据

iOS 应用逆向与安全 / 刘培庆著. —北京：电子工业出版社，2018.6
（安全技术大系）
ISBN 978-7-121-34099-4
Ⅰ. ①i… Ⅱ. ①刘… Ⅲ. ①移动终端—应用程序—程序设计—安全技术 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2018）第 081246 号

策划编辑：潘　昕
责任编辑：潘　昕
印　　刷：三河市君旺印务有限公司
装　　订：三河市君旺印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　　印张：25.25　　字数：535 千字
版　　次：2018 年 6 月第 1 版
印　　次：2018 年 6 月第 1 次印刷
定　　价：85.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

推荐序

第一次和刘培庆见面是在 2017 年，西湖大道的碧桃小馆。

四月的杭州已是初夏，只穿一件单衣的我，后背仍被汗水浸湿。昏暗的餐厅里，刘培庆坐在角落，一件牛仔夹克扣到顶，看上去又闷又热。我眉头一皱，心生不悦：这人真傻。

高温败了胃口，团购的双人餐，俩大老爷们儿竟然没吃完。结账走人，我们在涌金门一带的巷弄里穿行消食，有一搭没一搭地聊着。刘培庆说，他放弃了晋升的机会，跳到了另一个更辛苦的部门，是因为新部门的业务更吸引他。放弃更舒适的环境和更高的薪水去追求兴趣？兴趣值几个钱？这人真傻。

是夜一别，我们各自忙于生活，很久没有再会。五月某天，微信上突然弹出刘培庆的名字，“狗神，你那本书有没有打算再出一版，帮你更新一下”，然后向我介绍了他的更新计划，原来是在《iOS 应用逆向工程》的基础上更深入了。我觉得他的内容定位有些超出原书目标读者的水平，于是建议他当作技术博客发出来，算是委婉的回绝。你的原创内容，放在我的书里，如果广受欢迎，名气岂不全让我赚了？这人真傻。

七月的艳阳炙烤着大地，室内活动陡然增多。我要去滨江的 HZEcers 英语角分享出书经历，地点恰好就在网易旁边，于是邀请了他。没想到这哥们儿把女朋友带了来，两人在活动中一前一后，紧张地用生涩的英语做自我介绍：“Good afternoon everyone, my name is Liu Peiqing...”他的生物专业小女友梳着齐刘海，戴着圆框眼镜，英语比他溜得多：“...recently I'm learning programming, because I want to have more common topics with my boyfriend...”单纯的出发点，真是傻得可爱。也只有这样的傻姑娘，才会爱上这样的傻小子吧？

没过多久，我的论坛上出现了一个名为 MonkeyDev 的工具，是刘培庆写的，号称“原有 iOSOpenDev 的升级，非越狱插件开发集成神器”。我大致浏览了一下，看上去挺复杂，技术含量挺高，但竟然是开源的?！从 Git 提交记录来看，刘培庆一直在花时间维护它，免费供大家使用。这个年头，还有这样的雷锋，做这样的好事？这人真傻。

金秋十月，刘培庆又在微信上找我，说他要出书了，邀请我写个序。坦白地说，以刘培庆、

James 等 90 后为代表的新一代 iOS 逆向工程师鼓捣出来的新技术，我其实看不懂，也跟不上；同时，因为对库克的失望，我早已不再往苹果系技术上投入更多的精力。我已然从原来技术舞台的主角沦为了看客。“后浪”竟然找已经“死在沙滩上”的“前浪”写序？这人真傻。

但是，傻子刘培庆就在这样的评价中朝着自己的目标一步步前进，最终写出了《iOS 应用逆向与安全》，推动了行业的发展。

他让我想起了行业内的其他傻子。

四年前，另一个傻子，不知天高地厚地出版了《iOS 应用逆向工程》，填补了市场的空白，受到了读者的欢迎。这个傻子办了个名叫 iOSRE 的论坛，为所有 iOS 逆向工程爱好者提供免费、自由、平等、纯净的交流平台，却不做广告、不收赞助，自掏腰包维护论坛。这人真傻。

一个来自加拿大的傻子，为越狱 iOS 写了个名叫 Activator 的插件，全球总下载量近 2 亿次——据说这是乔布斯最喜爱的越狱插件。如果每次下载只收 1 毛钱，这个傻子也能成为千万富翁，可是他却把 Activator 免费提供给大家使用。这人真傻。

一个大学辍学的美国傻子，为越狱 iOS 提供了一套名为 Theos 的开发工具，它的简单易用吸引了大量人才进入这个领域，为越狱开发的黄金 5 年揭开了序幕。为了维护这套免费、开源的工具，他每天熬夜到凌晨 2 点，义务解决用户的问题，优化它的体验，却分文不取。这人真傻。

一个被亲生父母遗弃的傻子，都没正经上过大学，就自不量力地想要“Think different”。他创造的产品改变了世界，却积劳成疾，英年早逝，留给后人一句“Stay hungry, stay FOOLISH”。这人真傻。

“……

我不害怕全世界就剩下我一个傻瓜

我要坚持到底，用我的方式

别在意这世界的奇妙

……”

感谢刘培庆这样的傻子们。世界因为你们，变得可爱了一些。

沙梓社

2018 年 3 月 28 日夜，于杭州

前　言

2015 年，通过校招，我以 Windows 安全方向进入网易，组内安排投入 iOS 安全方向的研究。当时，我连苹果产品都没用过，于是攒了点钱，在淘宝上买了一台可越狱的 iPad 来研究。因为之前也没有接触过 Objective-C，无法深入阅读当时在网上找到的教程，所以只能跟着敲敲代码，看看效果。后来，通过研读念茜的文章，以及国外博客上的一些教程，把基本工具实践了一番，“狗神”沙梓社的书出版后，把他的书认真看了一遍，才算是踏进了门槛。

在那一段学习过程中，我对新的知识点都是囫囵吞枣，一直停留在工具的使用上，没有形成完整的知识体系和深层的认识，一旦出现问题就要花很长的时间去解决。这一点在后面张平引荐我去做网易云课堂的教学视频时感受尤为深刻——当你要规划整个课程时，你必须从全局出发考虑问题，仅仅根据自身的经验、知道工具的使用方法是不够的，只有理解和掌握原理，才能达到举一反三的效果。虽然从准备资料、制作 PPT 到最后录制视频的过程挺累的，但在这个过程中，我加深了对知识点的理解。这是我第一次录制视频课程，由于经验不足，导致了部分视频在终端的显示字体太小等问题，但总的来说，还是要感谢那些信任我、购买了我的视频课程的人。

后来，有几个朋友建议我出本书，把掌握的东西分享出来，也让新人少踩点坑。当时我是有点犹豫的。我不仅担心写书会占用很多时间，也担心自己的水平不够、写得不好。后来，想到视频里面的一些内容需要更新，很多知识点可以补充和完善，加上书籍的学习和沉淀效果也比视频好一些，我就开始做准备，规划每一章的完成时间，每天下班后或者周末在电脑前整理资料、写书，也挺充实的。现在，这本书终于和你见面了，希望书中的内容能够帮助你扩充自己的知识面，少走弯路，成为技术大牛。

读者对象

本书介绍了 iOS 开发、逆向和安全等方面的内容，面向以下读者：

- 高校计算机相关专业的学生

- iOS 开发工程师
- 逆向工程师
- 越狱开发工程师
- iOS 安全工程师
- 应用安全审计人员

近几年，iOS 开发人员数量激增。正向开发人员应该努力提升自己的竞争力，掌握一些底层技能，为自己开发的应用保驾护航。逆向新人也不要一味追求工具的使用和功能的实现，应该静下心来，基础知识掌握得扎实一些，后面的问题自然迎刃而解。

如何阅读本书

考虑到很多逆向分析人员缺乏正向开发和安全保护方面的知识，本书将分成以下 4 个部分进行讲解。

- 第 1 章～第 3 章是快速上手部分，内容包括一些基本概念的介绍，环境的准备，以及一些常用逆向分析工具的使用和原理。
- 第 4 章～第 6 章是正向知识储备和进阶部分，内容包括逆向过程中一些理论知识的深入讲解，例如类的结构、App 签名、Mach-O 文件格式、hook 原理等。
- 第 7 章是逆向实战部分，通过在越狱平台和非越狱平台上的逆向分析实例，带领读者学习逆向分析的思路和方法。
- 第 8 章是安全保护部分，内容包括应用安全及保护方面的知识，涉及网络传输加密、动态保护、代码混淆等。

尽管不同的人感兴趣的方面可能不一样，但我还是建议读者能够从头开始阅读本书，并把书中提到的每个知识点都实践一遍，以加深理解。

本书的源代码可以在 GitHub 上面找到：

https://github.com/AloneMonkey/iOSREBook

声明

本书的写作花费了大量的时间和心血，我只是想帮助大家在学习过程中少走弯路、拓宽知识面、增加技术积累，所以，请支持正版书籍，坚决抵制盗版！另外，本书内容仅供技术学习和研究之用，请勿将本书内容用于非法商业用途。

勘误

由于知识水平有限，写作过程也比较匆忙，书中难免出现错误及不足，欢迎各位读者指正。同时，我为本书开设了一个提交 issue 的项目：

https://github.com/AloneMonkey/iOSREBook-issues

致谢

感谢我的家人，在我成长的路上一直支持我、鼓励我。

感谢我异地三年的女友，很抱歉没有陪在你的身边。即便如此，你还是不离不弃，一直支持我的选择。

感谢念茜、狗神及在我的学习路上给予我帮助的人，感谢在网易期间的所有同事，是你们让我不断成长。

感谢电子工业出版社提供的平台，感谢编辑潘昕对本书内容的把控和指导。

感谢正在阅读本书的你，谢谢你的支持和信任。

刘培庆

2018 年 4 月，于杭州

目　录

第 1 章　概述

第 2 章　越狱设备

第 3 章　逆向工具详解

第 4 章　开发储备

第 5 章　分析与调试

第 6 章 逆向进阶

第 7 章 实战演练

第 8 章　安全保护

第1章 概述

1.1 逆向工程简介

iOS 应用逆向工程（后文简称“iOS 逆向”），是指从目标应用的界面及功能表现入手，使用不同的工具和理论知识去分析其实现原理，得出应用的代码结构、整体设计、功能实现、执行流程等，然后利用 iOS 的系统知识和语言特性，借鉴或修改原有实现流程的技术。如果你从未接触过逆向或者只接触过正向开发，你都将通过本书开启新的征程！

1.1.1 iOS 逆向学习基础

如果你是 iOS 正向开发者，你可以运用丰富的开发经验及对语言的了解，从界面功能知晓程序使用的系统组件或者第三方库，通过一些系统接口的调用及代理方法快速定位代码和发现线索。除此之外，你需要了解一些底层知识，例如文件结构、汇编知识、工具的使用及技巧。

如果你是其他平台的逆向者，并且熟悉该平台的文件格式（例如 PE、ELF），了解如何通过跳转表或 inline hook 的方式修改程序逻辑，那么这些知识可以帮助你快速学习 iOS 逆向。当然，你也需要了解 iOS 系统安全机制、语言特性及分析工具的原理。

如果你两者都不是，那也没有关系。只要保持积极学习的心态、锲而不舍的探索精神，通过逐步了解 iOS 系统机制、语言特性，不断实践，解决各种难题，并且在遇到问题时去探索问题的本质和原理，不断进行总结和积累，你也将成为一个逆向高手！

1.1.2 iOS 逆向的流程

在开始一系列的学习前，我们有必要了解 iOS 逆向工程的整个流程，以便对本书的内容及自己的学习进度有更好的把控。iOS 逆向工程的流程大致如下。

①解密、导出应用程序、class-dump 导出头文件，为后续工作做准备。

②从界面表现入手，获取当前界面布局及控制器。

③hook 发现的一些相关类，记录输出调用顺序及参数。

④找到关键函数，查看调用堆栈，hook 测试效果。

⑤静态分析加动态调试分析关键函数的实现逻辑。

⑥模拟或篡改函数调用逻辑。

⑦制作插件，或者移植到非越狱机器。

整体流程就是这样，但过程可能不会太顺利，有时需要反复探索才能定位目标函数。这个过程很枯燥，也很有趣。在多次分析和实践之后，你会找到属于自己的分析方法和技巧。

1.1.3 iOS 逆向使用的工具

对任何平台进行逆向分析时都会借助很多工具，iOS 逆向也不例外。以逆向流程为例，会使用解密工具、class-dump、Cycript、Reveal、Charles、Hopper、IDA、LLDB、Xcode、Theos 等，这些工具会在后面的章节中详细讲解。当然，仅学会工具的使用是远远不够的。在掌握工具使用方法的前提下，还需要了解每一个工具的实现原理是什么，有哪些可以借鉴的地方，如果是自己去实现应该怎么做等。本书在讲解工具的使用时，都会介绍工具的原理，从而帮助你查找出错的原因和改进现有的工具。

1.1.4 iOS 逆向的应用场景

很多人起初只是觉得 iOS 逆向工程很神秘，能做很多有意思的事情，才会去学习和探索。那么，学完 iOS 逆向之后到底可以做什么？能获得哪些帮助？笔者觉得有以下几点。

- 促进正向开发，深入理解系统原理。
- 借鉴别人的设计和实现，实现自己的功能。
- 分析恶意软件，应用安全审计。
- 从逆向的角度实现安全保护。

学完逆向工程之后，你将可以从不同的角度去看待和思考问题。

1.2 应用面临的安全风险

随着 iOS 逆向的工具越来越成熟，入门门槛越来越低，加上大部分应用都没有采取保护措施，现在市面上的很多应用都面临被破解、被篡改及协议被模拟等风险，也出现了不少多开、作弊、刷榜等恶意行为，应用面临的安全问题已经非常严峻。下面从 3 个方面分别举例说明。

1.2.1 静态修改文件

虽然在 iOS 平台上直接静态修改汇编的情况较少（当然也是可以的），但还是可以通过 insert_dylib 或者 optool 等工具对可执行文件进行注入，使其加载指定的动态库文件，达到篡改的效果（非越狱插件）。另外，可以通过修改本地读取的特定文件去修改程序的功能。

1.2.2 动态篡改逻辑

在越狱设备上，可以通过 Cydia Substrate 提供的注入和 hook 模块注入指定的程序，篡改程序的功能。在非越狱设备上，也可以静态注入动态库，使用 fishhook 或者 Method Swizzle 达到修改或增加功能的效果。例如，修改 Pokeman GO 的定位的信息及移动速度。再如，在某应用收到红包时，自动调用领取红包的功能函数，实现“秒抢”的效果。此外，可以解除会员限制、破解本地验证等。

1.2.3 协议分析

除了修改应用本身，还可以逆向分析应用程序的请求协议，并通过其他程序来模拟协议，实现脱机。这种机制经常被利用进行恶意注册和刷单等。

1.3 应用保护手段

有攻便有防。为了保护应用不被篡改、分析、重签名等，需要采取一定的保护措施。虽然安全保护只是增加了逆向分析的难度，但这种方法往往是有效且必要的。

1.3.1 数据加密

应用中一般都会存储很多敏感数据，例如加密和解密的 key、用户的隐私信息、通信传输的信息。如果开发者的安全意识比较弱，可能会把一些敏感信息或者敏感字符串以明文的形式写在代码里面。此外，传输的一些关键数据也不会加密，而这往往会给分析者提供很多关键信息，增加应用被破解的风险。针对数据加密，可以从以下 3 个方面入手。

- 静态字符串加密
- 本地存储加密
- 网络传输加密

通过静态字符串加密隐藏关键信息，可以增加静态分析的难度。对本地和网络数据进行加

密，也可以避免重要信息泄露和中间人攻击。

1.3.2 程序混淆

程序中的类名和方法名定义往往是有具体含义的，攻击者根据方法名就能分析出该方法的用途。为了避免这种情况，可以把类名和方法名替换成无意义的随机字符串，让分析者无法通过名字获取有意义的信息。

除此之外，使用静态分析工具进行反编译是很容易看出程序的实现逻辑和调用关系的，所以，对关键代码的保护和混淆不可或缺。可以通过 LLVM 在编译过程中对中间代码进行控制流程的混淆、代码扁平化、插入干扰代码等操作，以干扰分析者的静态分析，增加分析的难度和成本。

1.3.3 安全监测

分析者在分析应用时，可以通过调试、注入动态库、重签名等手段修改原有应用。为保证应用运行时的安全性，可以在应用中加入反调试代码，检测调试器，或者加入动态库注入检测、函数 hook 检测、签名检测、完整性等，使得当检测到应用被侵入时，程序退出或者功能运行出现异常。

通过以上对 iOS 逆向工程的大致介绍，相信你对逆向工程的内容及学习流程都有了一定的了解。在学习过程中，要着重于理论基础和动手实践，慢慢地领会逆向的魅力。无论你是安全人员、开发者还是逆向新手，都能从中感受到乐趣。同时，正向开发者和安全人员对本章提到的风险也要认真对待，因为在学习逆向分析的方法之后，就能针对不同的分析方法提供不同的安全保护方法，从而进一步保护应用的安全了。

1.4 本书工具

为了让你更加快速地上手，也为了避免后续对一些 Mac 常用工具的重复介绍，本节将对本书中使用的一些 Mac 工具进行总体介绍。你不用马上把下面所有的工具都安装好，可以在后续使用时再安装。

1.4.1 效率工具

在工作中，一个好的开发环境可以帮助我们节约很多时间。正所谓“工欲善其事，必先利

其器”，先把利器准备好，才能高效地进行分析。下面介绍几款能够提高 Mac 开发分析效率的工具。

- iTerm2：代替默认的 Terminal，提供了很多高级设置，例如自动补全、高亮等（https://iterm2.com/）。
- oh-my-zsh：可以自定义主题、Git 显示、Tab 补全等（https://github.com/robbyrussell/oh-my-zsh）。
- Go2Shell：从 Finder 打开终端并自动切换到当前目录（http://zipzapmac.com/go2shell）。
- autojump：从终端快速进行目录跳转和切换（https://github.com/wting/autojump）。
- Alfred：快速打开软件，自定义脚本执行（https://www.alfredapp.com/）。

1.4.2　实用工具

效率提升后，需要一些实用的工具来帮助完成某些复杂的操作。下面来看几款实用的工具。

- Homebrew：Mac OS 的包管理工具，可以快速安装各种工具（https://brew.sh/）。
- Cakebrew：Homebrew 的界面管理工具（https://www.cakebrew.com/）。
- libimobiledevice：提供了很多能与 iOS 交互的工具，例如端口映射查看日志（https://github.com/libimobiledevice）。
- tree：用于查看当前目录结构树的命令行工具，通过 brew install tree 安装。
- 010 Editor：二进制编辑分析工具（https://www.sweetscape.com/010editor/）。

1.4.3　逆向工具

关于在逆向分析中使用的工具，在后面的章节里会进行详细讲解。本书没有深入讲解的工具在这里介绍如下。

- jtool：查看文件结构、代码签名（http://www.newosxbook.com/tools/jtool.html）。
- capstone：多平台、多架构支持的反汇编框架（http://www.capstone-engine.org/）。
- keystone：将汇编指令转换为 Hex 机器码（https://github.com/keystone-engine/keystone）。
- radare2：一款开放源代码的逆向工程平台（https://github.com/radare/radare2）。
- mobiledevice：安装 app 或 ipa 包（https://github.com/imkira/mobiledevice）。

第 2 章　越狱设备

2.1　什么是越狱

越狱是指通过分析 iOS 系统的代码，找出 iOS 系统的漏洞，绕过系统的安全权限检查，最终获取系统 root 权限的过程。根据越狱的程度不同，有如下 3 种定义。

- 引导式越狱：设备重启后，之前的越狱状态就会失效。这时需要连接计算机，使用越狱软件引导开机，也就是重新越狱，否则设备就无法开机使用。
- 不完美越狱：设备重启后，之前的越狱状态也会失效，不过设备可以作为非越狱设备正常使用。这时需要手动重新越狱。
- 完美越狱：越狱后重启设备，设备仍处于越狱状态，即重启不会破坏越狱环境。

越狱的目的是获取 root 权限。如果没有越狱，所有的操作就只能局限于沙盒（Sandbox），而越狱之后可以访问设备的整个文件系统、更改系统外观和功能、编写命令行工具、root App 及动态注入动态库到指定应用等。虽然越狱后可以获取 root 权限，但是原来在沙盒内的应用在越狱之后还是在沙盒内——这一点需要特别注意！在学习 iOS 逆向的过程中，将设备越狱可以帮助我们深入了解 iOS 系统的结构，分析系统的行为，更好地掌握和使用相关工具。

2.2　Cydia

在手机越狱的过程中，会安装一个叫作 Cydia 的软件。这是一个让用户在越狱的 iOS 设备上查找和安装各类软件包的工具，包括软件、系统修改、应用插件等的软件管理器，相当于越狱设备的 App Store。打开 Cydia，可以看到 5 个 Tab 选项，分别是“Cyida”“软件源”“变更”“已安装”“搜索”，如图 2-1 所示。

- Cydia：展示一些推荐插件和用户指南。
- 软件源：Cydia 可以通过添加其他软件源来安装源上面的软件（点击右上角的“编辑”选

项，然后点击左上角的“添加”选项，输入软件源的地址，最后点击“添加源”选项）。

- 变更：展示一些已安装软件可用的更新。
- 已安装：展示当前已经安装的软件，可以选择“专业人士”、“查看更多”或者“选择最近”选项来查看最近安装的软件和 Tweak。
- 搜索：可以搜索想要安装的软件的名字，然后点击“安装”选项。后面使用的很多工具都能在这里搜索和安装。

图 2-1　Cydia 首页

2.3　SSH

因为后面的很多操作都需要登录越狱设备才能进行，所以需要先配置 SSH 远程登录，以便快速访问越狱设备的文件系统，从而在上面执行命令。

SSH 是一种网络协议，用于计算机之间的加密登录，它存在多种实现。因为不同的系统使用的越狱工具不一样，所以需要根据 iOS 系统的版本进行配置。在 iOS 8 和 iOS 9 的越狱设备上，可以通过安装 OpenSSH 来登录；对于 iOS 10.0 ~ iOS 10.2 的越狱系统，越狱工具 yalu 内置了一个相对轻量级的 SSH 服务 dropbear，供用户直接使用。下面分别介绍这两种方法。

2.3.1 安装 OpenSSH

该操作只针对 iOS 8 和 iOS 9 的越狱设备，iOS 10 的用户可以直接跳过。首先，在 Cydia 中搜索并安装 OpenSSH，如图 2-2 所示。

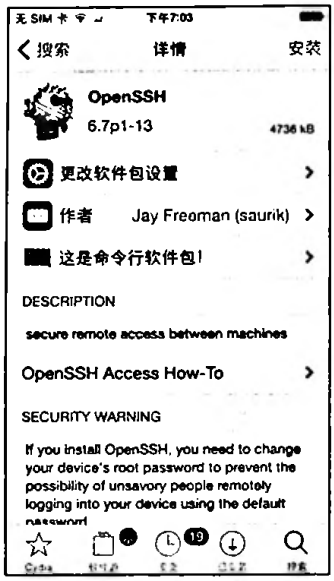


图 2-2 在 Cydia 中搜索并安装 OpenSSH

打开 Wi-Fi 设置界面，点击当前连接的 Wi-Fi，获取 IP 地址。如图 2-3 所示，这里获取的 IP 地址为 192.168.2.202。打开 Mac 终端，输入如下命令后按“Enter”键。

```
ssh root@192.168.2.202
```

稍等片刻，会出现如下提示。输入“yes”并按“Enter”键，系统会显示一句提示，表示 host 主机已经得到认可。

```
ssh root@192.168.2.202
The authenticity of host '192.168.2.202 (192.168.2.202)' can't be established.
RSA key fingerprint is SHA256:yaLlbkAnRiLCVjSou6egDssMLPCbf1q2Ii5466DQzxM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.2.202' (RSA) to the list of known hosts.
```

然后，输入默认的密码“alpine”，完成登录。

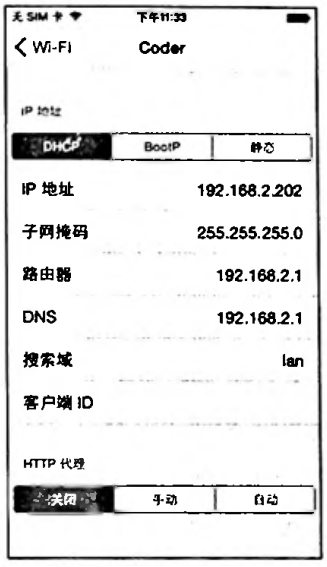


图 2-3　获取当前 Wi-Fi 连接的 IP 地址

当远程主机的公钥被接受以后，它会被保存在 $HOME/.ssh/known_hosts 文件中。再次连接这台主机时，系统就会认出它的公钥已经保存在本地，从而跳过警告部分，直接提示用户输入密码。如果连接时出现如下错误信息，说明 known_hosts 文件中保存的 IP 端口所对应的公钥和登录设备的公钥不匹配。这时需要打开 known_hosts 文件，删除对应的 IP 地址和端口记录。

```
➜  ~ ssh -p 2222 root@localhost
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
SHA256:gk78TIJMoQYgUMfNUw7KTP+G8XLZuE8AW045gBZpCLw.
Please contact your system administrator.
Add correct host key in /Users/alonemonkey/.ssh/known_hosts to get rid of this message.
Offending RSA key in /Users/alonemonkey/.ssh/known_hosts:3
RSA host key for [localhost]:2222 has changed and you have requested strict checking.
Host key verification failed.
```

2.3.2 配置 dropbear

对于 iOS 10.0 ~ iOS 10.2 的越狱系统，不用安装 OpenSSH。yalu 内置了一个相对轻量级的 SSH 服务 dropbear。首先，在 Cydia 中搜索并安装 MTerminal 和 adv-cmds，然后运行 `ps aux | grep dropbear` 命令。如果输出如图 2-4 所示的内容，表示默认支持 USB 连接。

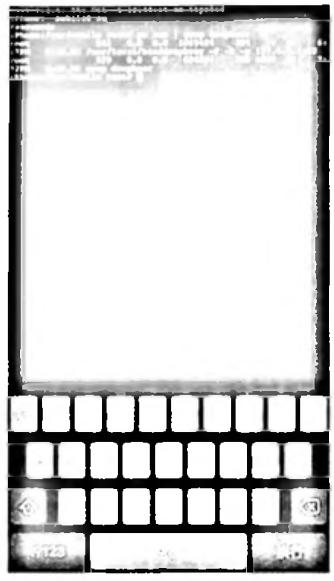


图 2-4 在使用 MTerminal 前查看 dropbear 进程

这时如果要连接，在 Mac 上面转发端口即可登录，命令如下。

```
iproxy 22 2222
ssh -p 2222 root@localhost
```

如果要通过 Wi-Fi 连接，需要在设备终端执行如下命令。

```
/usr/local/bin/dropbear -F -R -p 22
```

然后，在 Mac 上直接进行 SSH 登录，命令如下。

```
ssh root@192.168.2.202
```

如果默认通过 Wi-Fi 连接，则需要修改 yalu102.app 下面的 dropbear.plist 文件。运行 `ps aux | grep yalu` 命令，找到 yalu102.app 的目录，然后使用 iFile 将其中的“127.0.0.1:22”改为“22”，它会将该文件复制到 /Library/LaunchDaemons 目录下，使其自启动。

2.3.3　修改默认密码

在登录时默认有一个密码，我们可以将其修改成自己的密码。在修改密码前，需要了解系统中的两个用户角色 root 和 mobile。root 用户是系统中权限最高的用户，具有对系统的完全控制权。mobile 用户比 root 用户的权限低，可以操作普通文件，不能操作系统文件（关于文件权限的内容可以参考 2.4.2 节）。为了确保安全性，需要修改 root 用户和 mobile 用户的密码。分别使用 passwd 和 passwd mobile 命令修改 root 用户和 mobile 用户的密码，具体操作如下。

```
ssh root@192.168.2.202
root@192.168.2.202's password:
Monkey:~ root# passwd
Changing password for root.
New password:
Retype new password:
Monkey:~ root# passwd mobile
Changing password for mobile.
New password:
Retype new password:
Monkey:~ root#
```

2.3.4　公钥登录

如果使用密码登录，必须在每次登录时输入密码，但这样的操作在后面打包安装和调试时非常麻烦。可以使用 SSH 提供的公钥登录，以省去输入密码的步骤。

公钥登录的原理也很简单。例如，在使用 GitHub 时，本机的 SSH 公钥被保存到 GitHub 中，登录时远程设备会向用户发送一个随机字符串，登录用户用自己的私钥加密后再将其发送到远程设备，远程设备用事先存储的公钥进行解密，如果解密成功，就证明用户是可信的，可以直接登录，不再要求用户输入密码。这种方法需要用户提供自己的公钥，如果没有公钥，可以使用 ssh-keygen 生成，生成步骤如下。

```
➜  ~ ssh-keygen -t rsa -P ''
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/monkey/.ssh/id_rsa):
Your identification has been saved in /Users/monkey/.ssh/id_rsa.
Your public key has been saved in /Users/monkey/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:G7wV+pGlRCWm4ZVGeMjX2/F0iaMGxLMKZFp6VDp+7t8 monkey@bogon
The key's randomart image is:
```

```
+---[RSA 2048]----+
|      .o+=*+. . .|
|     =..+X=..o..o|
|    Bo  o+* oo.+.|
|   o.o.. + B. . .|
|    ....S *      |
|      o. * .     |
|       .o .      |
|      .   .      |
|       ... E     |
+----[SHA256]-----+
```

运行结束，会在$HOME/.ssh/目录下（终端打开命令为 open $HOME/.ssh/）生成文件 id_rsa.pub 和 id_rsa（前者是生成的公钥，后者是私钥）。然后，使用 ssh-copy-id 命令，将生成的公钥上传到远程设备中，再次登录时就不用输入密码了。相关代码如下。

```
➜  ~ ssh-copy-id -i $HOME/.ssh/id_rsa.pub root@192.168.2.202
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
"/Users/monkey/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that
are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it
is to install the new keys
root@192.168.2.202's password:

Number of key(s) added:        1

Now try logging into the machine, with:   "ssh 'root@192.168.2.202'"
and check to make sure that only the key(s) you wanted were added.
```

这一步是将本机的公钥 $HOME/.ssh/id_rsa.pub 追加到远程设备的 $HOME/.ssh/authorized_keys 文件中，读者可以自行比对。如果登录时仍然需要输入密码，可以执行如下命令打印 debug 信息来排查问题。

```
ssh -vvv root@192.168.2.202
```

注意：要确保设备中 $HOME/.ssh 文件和 $HOME/.ssh/authorized_keys 文件的组权限和其他用户权限中没有写（w）权限。因为系统认为这是不安全的，会导致授权失败。可以通过如下命令纠正权限问题。

```
chmod 700 $HOME/.ssh
```

```
chmod 600 $HOME/.ssh/authorized_keys
```

2.3.5　通过 USB 登录

在 Wi-Fi 网络不稳定的情况下，如果通过 Wi-Fi 登录，在输入命令时会出现卡顿的情况。这里提供一种通过 USB 端口转发的方法，即使用 USB SSH 登录，以保证连接的稳定和流畅。首先通过如下命令安装 libimobiledevice，然后使用里面提供的工具 iproxy 把本地端口 2222 映射到设备的 TCP 端口 22，这样就可以通过本地的 2222 端口建立连接了。

```
brew install libimobiledevice
iproxy 2222 22
ssh root@localhost -p 2222
```

如果觉得每次通过输入命令来进行端口转换比较麻烦，也可以把这个端口转换命令写到开机启动加载项中。创建文件 ~/Library/LaunchAgents/com.usbmux.iproxy.plist，将以下内容写入该文件，然后运行命令 `launchctl load ~/Library/LaunchAgents/com.usbmux.iproxy.plist`。

```
<?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>Label</key>
    <string>com.usbmux.iproxy</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/local/bin/iproxy</string>
        <string>2222</string>
        <string>22</string>
    </array>
    <key>RunAtLoad</key>
    <true/>
    <key>KeepAlive</key>
    <true/>
</dict>
</plist>
```

如果还是觉得每次输入 `ssh root@localhost -p 2222` 命令很麻烦，可以通过给它指定一个名字进行 SSH 连接。打开 $HOME/.ssh/config 文件（如果没有就新建一个），写入以下内容。完成

这些操作后，在终端输入 ssh 5s 命令即可登录。

```
Host 5s                    #自定义的设备名
Hostname localhost         #通过 USB 端口映射，写 localhost
User root                  #以 root 用户登录
Port 2222                  #指定端口号为映射的端口号 2222
```

2.4 iOS 系统结构

在手机终端执行 uname -a 命令，可以看到 iOS 系统是基于 Darwin Kernel 的，而 Darwin Kernel 是一种 UNIX-like 操作系统。了解 UNIX 的读者在后面学习 iOS 系统的相关知识时肯定会比较顺利，因为了解系统的结构对学习和理解逆向会有很大的帮助。由于 iOS 的正向开发只能在应用的沙盒里面操作，开发者关心的只有沙盒里面的 Document、Library、tmp 等。可以通过 Xcode Devices and Simulators 中间部分的 INSTALLED APPS 列出手机上的部分应用，单击下面的“+”按钮安装应用、“-”按钮删除应用，通过旁边的设置按钮可以浏览、下载和替换应用的沙盒文件，如图 2-5 所示。

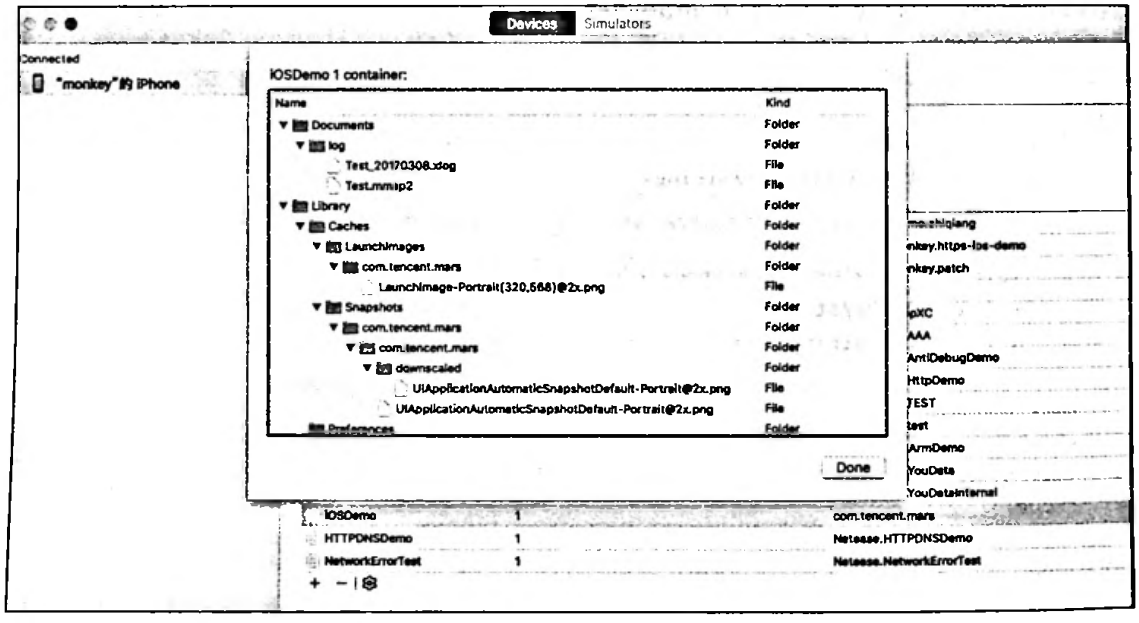


图 2-5 通过 Xcode 查看非越狱设备的沙盒

不过，通过这种方式只能查看自签名的应用，对于从 App Store 下载的其他应用束手无策。

在越狱设备上，可以通过工具访问整个 iOS 文件系统。下面我们就来逐步认识 iOS 系统的结构。

2.4.1　文件目录

在越狱设备上，可以借助文件系统查看工具浏览系统的文件目录，具体如下。

- Mac 平台
  - iTools（http://pro.itools.cn/mac/）
  - iFunBox（http://www.i-funbox.com/）
- iOS 平台
  - iFile（通过 Cydia 安装）

也可以登录手机，使用 `ls -l` 命令查看系统中的所有文件目录及属性。如果要访问越狱设备的文件系统，则要先打开 Cydia，安装 Apple File Conduit "2"，如图 2-6 所示。

图 2-6　安装 Apple File Conduit "2"

在这里使用 iFunBox 查看越狱设备的文件系统。打开 iFunBox，单击左侧的“文件系统”选项，如图 2-7 所示。

下面介绍一些比较重要的目录，以便在分析时寻找想要的内容。

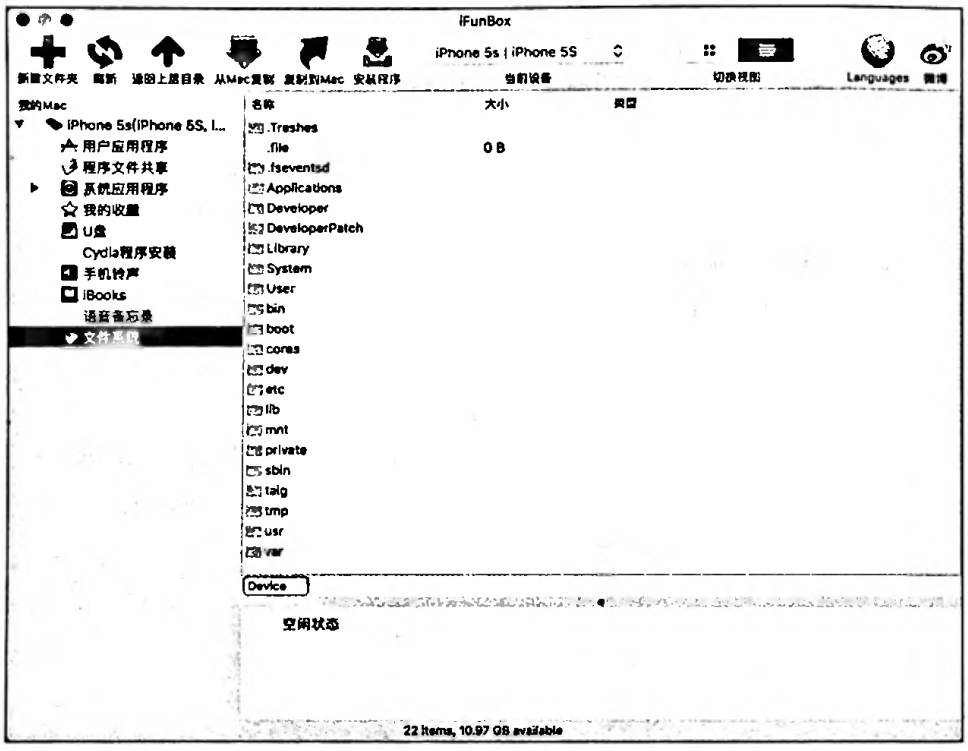


图 2-7 越狱设备的文件系统目录

- Applications：存放所有的系统 App 和来自 Cydia 的 App，不包括从 App Store 下载的 App。
- Developer：供开发者使用。
- Library：系统资源，用户设置。例如，Logs 是系统日志，Ringtones 是系统自带铃声，Launch Daemons 是启动 daemon 程序。其中比较重要的目录是 /Library/MobileSubstrate，里面存放了所有基于 Cydia Substrate 的插件（需要在 Cydia 中安装 Cydia Substrate）。
- System：系统的重要组成部分。
  - /System/Library/Carrier Bundles 里面是运营商的一些配置。
  - /System/Library/Frameworks 和 /System/Library/PrivateFrameworks 里面是系统中各种公开的和未公开的 framework。
  - /System/Library/CoreServices/SpringBoard.app 是桌面管理器，是用户和系统直接交互的部分。
  - /System/Library/PerferenceBundles 里面存放的是系统设置中的一些设置项。
- User：用户目录，实际指向 /var/mobile。

  - /User/Media/ 里面存放的是相册等。
  - /User/Library/ 里面存放的是短信、邮件等。
- bin：存放用户级二进制文件，例如 mv、ls 等。
- dev：设备文件。每个设备在 /dev 目录下都有一个对应的文件（节点）。
- etc：存放系统脚本、hosts 配置、SSH 配置文件等，实际指向 /private/etc。
- sbin：存放系统级二进制文件，例如 reboot、mount 等。
- usr：用户工具和程序。/usr/include 中存放标准 C 头文件，/usr/lib 中存放库文件。
- var：一些经常改动的文件，包括 keychains、临时文件、从 App Store 下载的应用。

2.4.2　文件权限

文件权限是 iOS 系统确保安全操作的一个重要部分。在 iOS 系统中，每个文件都具有以下属性。

- 所有者权限：决定文件的所有者可以对文件进行的操作。
- 组权限：决定属于该组的成员对他所拥有的文件能够进行的操作。
- 其他人权限：表示其他人能够对该文件进行的操作。

iOS 系统用 3 个比特（bit）表示文件的操作权限，从高位到低位分别是读（r）、写（w）和执行（x）。使用 ls -l 命令可以将与文件相关的各种权限展示出来，如下所示。

```
Monkey:~ root# cd /
Monkey:/ root# ls -l
total 22
lrwxr-xr-x  1 root admin   32 Jun 23 13:39 Applications -> /var/stash/_.VIahI5/Applications
drwxrwxr-t  8 root admin  340 Oct 14  2014 Developer
drwxrwxr-x  9 root wheel  374 Oct 22  2014 DeveloperPatch
drwxrwxr-x 15 root admin  714 Jul  5 00:24 Library
drwxr-xr-x  3 root wheel  102 Dec 26  2007 System
lrwxr-xr-x  1 root admin   11 Oct  5 20:20 User -> /var/mobile
drwxr-xr-x  2 root wheel 2006 Jul 21 13:02 bin
drwxr-xr-x  2 root wheel   68 Oct 28  2006 boot
drwxrwxr-t  2 root admin   68 Sep 13  2014 cores
dr-xr-xr-x  3 root wheel 1371 Oct  5 20:19 dev
lrwxr-xr-x  1 root wheel   12 Jan  8  1970 etc -> private/etc/
drwxr-xr-x  2 root wheel   68 Oct 28  2006 lib
drwxr-xr-x  2 root wheel   68 Oct 28  2006 mnt
drwxr-xr-x  4 root wheel  136 Nov  5  2014 private
drwxr-xr-x  2 root admin 1564 Jul 21 13:02 sbin
```

```
drwxr-xr-x  2 root admin  136 Jan  8  1970 taig
lrwxr-xr-x  1 root wheel   16 Jan  8  1970 tmp -> private/var/tmp/
drwxr-xr-x  9 root wheel  374 Jun 23 13:40 usr
lrwxr-xr-x  1 root wheel   12 Jan  8  1970 var -> private/var/
```

- 第 1 个字符表示文件的类型，“l”是符号链接文件，“d”是文件夹，“-”是普通文件。
- 第 2～4 个字符表示文件所有者的权限。例如，“lrwxr-xr-x”表示文件所有者拥有读（r）、写（w）和执行（x）权限。
- 第 5～7 个字符表示文件所属组的权限。例如，“lrwxr-xr-x”表示所属组拥有读（r）和执行（x）权限，但没有写（w）权限。
- 第 8～10 个字符表示其他人的权限。例如，“lrwxr-xr-x”表示其他人拥有读（r）和执行（x）权限，但没有写（w）权限。
- root 表示文件所属用户，wheel 和 admin 表示文件所属组。

r、w、x 中的每一个都对应于二进制 1 或 0，可以写成“111”。所以，可以用 4 代表读权限，用 2 代表写权限，用 1 代表执行权限。当然，也可以组合使用。例如，“r-x”代表读和执行权限，即 4（读）+ 1（执行）= 5。

可以使用 chmod 修改一个文件或者目录的权限，有两种模式，即符号模式和绝对模式。

- 符号模式：表示设置所有者权限为删除和执行，组权限为读和执行，其他人权限增加写和执行，示例如下。

```
chmod u-x,g=rx,o+wx testfile
```

- 绝对模式：表示设置所有者权限为读、写和执行（r、w 和 x），组权限为读和执行（r 和 x），其他人权限为读和执行（r 和 x），示例如下。

```
chmod 755 testfile
```

2.5 Cydia Substrate

Cydia Substrate（以前叫作 MobileSubstrate）是一个框架，允许第三方开发者在越狱系统的方法里打一些运行时补丁和扩展一些方法，是开发越狱插件的基石。首先，在越狱设备上面安装 Cydia Substrate，如图 2-8 所示。

Cydia Substrate 包含 3 个主要模块，分别是 MobileHooker、MobileLoader 和 Safe mode。

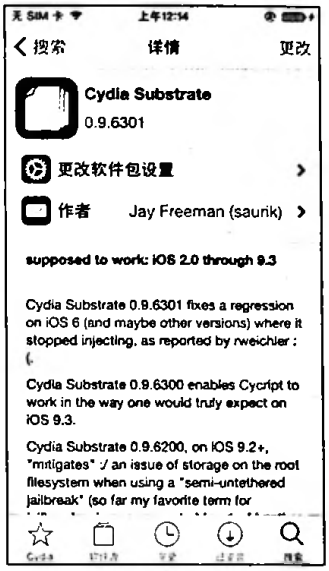


图 2-8　安装 Cydia Substrate

2.5.1　MobileHooker

MobileHooker 用于替换系统和应用的方法。它提供 MSHookMessageEx 来 hook OC 函数，提供 MSHookFunction 来 hook C 函数，具体的用法会在后面的章节讲到。

2.5.2　MobileLoader

MobileLoader 用于将第三方动态库加载到运行的目标应用里面。MobileLoader 首先通过环境变量 DYLD_INSERT_LIBRARIES 把它自己加载到目标应用里面，然后查找 /Library/MobileSubstrate/DynamicLibraries/ 目录下面的所有 plist 文件，如果 plist 文件里面的配置信息符合当前运行的应用，就会通过 dlopen 函数打开对应的 dylib 文件，如图 2-9 所示。

plist 文件中有一些过滤条件，只有满足条件时，第三方动态库才会被加载。过滤条件如下。

- CoreFoundationVersion：只有 CoreFoundation.framework 的版本高于某个值时才会加载。
- Bundles：只有应用的 Bundle ID 在该列表中时才会加载。
- Classes：只有应用实现了某些特定的 OC 类时才会加载。
- Executables：只有应用的可执行文件的名字和该列表匹配时才会加载。

例如，注入 SpringBoard，可以写成如下方式。

```
Filter = {
  Bundles = (com.apple.springboard);
};
```

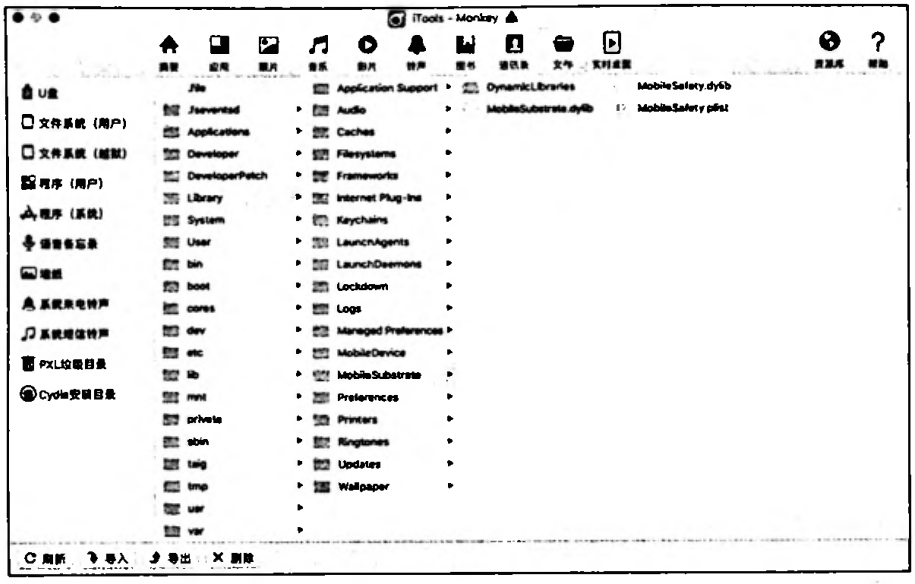


图 2-9　在 iTools 中查看 DynamicLibraries 目录

基于对安全性的考虑，当 plist 文件不存在时，dylib 不会注入所有进程。如果确定要这么做，可以使用如下方式指定。

```
{
    Filter = {
        Bundles = (
            "com.apple.Security",
        );
    };
}
```

2.5.3　Safe mode

如果插件加载导致了 SpringBoard 崩溃，MobileSafety 会捕获这个异常并让设备进入安全模式。在安全模式中，所有的第三方插件都不会加载。所以，当因为编写插件导致崩溃而进入安全模式后，可以找到最近安装的插件，然后将其卸载。

2.6　越狱必备工具

准备好越狱手机后，有很多工具都是必须要安装的。除了前面提到的 Apple File Conduit "2" 和 Cydia Substrate，还有以下工具。

2.6.1　adv-cmds

在手机上经常会使用 ps 命令来查看当前运行的进程 ID 及可执行文件的路径，但在初次使用时可能会出现“-sh: ps: command not found”这样的错误，这是因为没有安装 adv-cmds。adv-cmds 会提供 ps 命令，如图 2-10 所示。

图 2-10　安装 adv-cmds 软件包

2.6.2　appsync

直接修改一个应用的结构或文件会破解应用本身的签名信息，所以，安装修改后的应用会出现“Failed to verify code signature of XXX”这样的错误。这时需要安装 appsync，让系统不再验证应用的签名。在安装时要选择能够支持当前系统的版本。

2.6.3　iFile

iFile 是手机上的文件管理器，用于管理手机中的文件、修改文件的权限等，如图 2-11 所示。

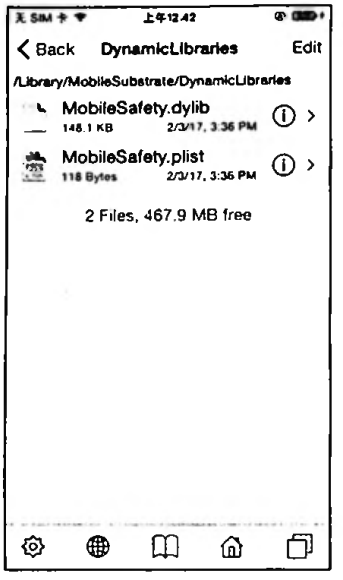


图 2-11 使用 iFile 查看文件

2.6.4 scp

对 iOS 10 以后的版本，使用 yalu 越狱后就没有 scp 这个工具了。读者可以在本书的源代码中找到 scp，使用 iFunBox 将其复制到设备的 /usr/bin/目录下，然后通过 SSH 访问设备，执行如下命令。

```
cd /usr/bin/
ldid -S scp
chmod 777 scp
```

或者，在 Cydia 中搜索并下载 rsync 来代替 scp，方法如下。

```
rsync -avze 'ssh -p 2222' root@localhost:/tmp/tmpfile ./
rsync -avze 'ssh -p 2222' ./utils.cy root@localhost:/var/root/
```

第 3 章　逆向工具详解

3.1　应用解密

应用上传至 App Store 后，苹果会对应用的代码部分进行加密，当应用运行时才会动态解密，在这样的情况下是无法直接使用后面讲到的 class-dump 和 IDA 进行分析的。所以，在分析应用之前，要把应用加密的内容解密，然后把应用和解密后的可执行文件导出到计算机中。

本节介绍如下两种解密方法。

- 使用 dumpdecrypted 动态注入后，dump 内存中解密后的代码部分。
- 使用 Clutch，通过调用 posix_spawnp 生成一个进程，然后暂停进程，dump 内存。

3.1.1　dumpdecrypted

dumpdecrypted 是一个开源的工具，它会注入可执行文件，动态地从内存中 dump 出解密后的内容。下面让我们使用 dumpdecrypted 一步步进行解密。

01　从 GitHub 下载源代码并编译

从 GitHub 下载源代码并编译，具体如下。

```
➜  iosreversecode git clone https://github.com/stefanesser/dumpdecrypted.git
Cloning into 'dumpdecrypted'...
remote: Counting objects: 31, done.
Unpacking objects: 100% (31/31), done.
remote: Total 31 (delta 0), reused 0 (delta 0), pack-reused 31
➜  iosreversecode cd dumpdecrypted
➜  dumpdecrypted git:(master) make
`xcrun --sdk iphoneos --find gcc` -Os  -Wimplicit -isysroot `xcrun --sdk iphoneos
--show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks
-F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch armv7 -arch
armv7s -arch arm64 -c -o dumpdecrypted.o dumpdecrypted.c
`xcrun --sdk iphoneos --find gcc` -Os  -Wimplicit -isysroot `xcrun --sdk iphoneos
```

```
--show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks
-F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch armv7 -arch
armv7s -arch arm64 -dynamiclib -o dumpdecrypted.dylib dumpdecrypted.o
```

编译完成，会生成一个 dumpdecrypted.dylib 文件。接下来，我们就使用这个文件进行注入解密。

02 用 ps 命令定位待解密的可执行文件

通过 SSH 访问手机，退出所有应用。然后，打开目标应用，使用 ps -e 命令查看正在运行进程的可执行文件路径，找到位于 App Store 应用目录下的目标文件路径。下面的 TargetApp 表示解密的目标应用。

```
➜  ~ ssh 5s
monkeyde-iPhone:~ root# ps -e
PID TTY          TIME CMD
1 ??        6:22.64 /sbin/launchd
18 ??        0:00.02 /usr/libexec/amfid
27 ??        2:21.44 /usr/sbin/syslogd
29 ??        0:01.86 /usr/libexec/misd
......
15710 ??        0:04.15
/var/mobile/Containers/Bundle/Application/AB23846C-B727-4F56-B62D-3AAF74DDDFB7/TargetAp
p.app/TargetApp
......
```

03 获取 TargetApp 的 Documents 目录

为什么要获取 Documents 目录呢？因为手机虽然越狱了，但是从 App Store 下载的应用还在沙盒中，而注入的动态库需要把解密后的文件写到和动态库同名的目录下。所以，只能把动态库复制到沙盒目录下，而这里选择了沙盒的 Documents 目录。每个 App 目录下都有一个 Info.plist 文件，该文件用于说明 App 的一些信息，其中就包括 App 的 Bundle ID。可以根据 Bundle ID 调用私有 API，获取应用的 Documents 目录。

获取 Bundle ID 的命令和编写的代码分别如下。

```
cat
/var/mobile/Containers/Bundle/Application/AB23846C-B727-4F56-B62D-3AAF74DDDFB7/TargetAp
p.app/Info.plist | grep CFBundleIdentifier -A 1
```

```
NSString* bundleID = @"com.toyopagroup.picaboo";  //此处为获取的 Bundle ID
```

```
NSURL* dataURL = [[NSClassFromString(@"LSApplicationProxy")
performSelector:@selector(applicationProxyForIdentifier:) withObject:bundleID]
performSelector:@selector(dataContainerURL)];
NSLog(@"%@",[dataURL.absoluteString stringByAppendingString:@"/Documents"]);
```

新建一个 iOS 项目，把获取 Bundle ID 的代码复制到 application:didFinishLaunchingWithOptions:下，然后到目标设备中运行，即可在控制台看到输出目录。还可以使用 cycript -p TargetApp 命令注入进程，调用 [[NSBundle mainBundle] bundleIdentifier] 和 NSHomeDirectory() 函数分别获取 Bundle ID 和 Home 目录（加上“/Documents”即可）。关于 Cycript 的使用，会在后面进行讲解。

这里获得的 Documents 目录为 /var/mobile/Containers/Data/Application/651168C8-D728-4B83-A365-FE8A75D67C69/Documents。

04　将 dylib 复制到 Documents 目录下

使用 scp 命令把生成的 dumpdecrypted.dylib 文件复制到 Documents 目录下，具体如下。

```
scp -P 2222 ./dumpdecrypted.dylib
root@localhost:/var/mobile/Containers/Data/Application/651168C8-D728-4B83-A365-FE8A75D6
7C69/Documents
```

05　解密

通过 DYLD_INSERT_LIBRARIES 注入就可以解密了，具体如下。

```
DYLD_INSERT_LIBRARIES=/path/to/dumpdecrypted.dylib /path/to/executable
```

针对当前目标应用的用法如下。

```
ssh 5s
cd
/var/mobile/Containers/Data/Application/651168C8-D728-4B83-A365-FE8A75D67C69/Documents
xxx/Documents root# DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/var/mobile/Containers/Bundle/Application/AB23846C-B727-4F56-B62D-3AAF74DDDFB7/TargetAp
p.app/TargetApp
mach-o decryption dumper

DISCLAIMER: This tool is only meant for security research purposes, not for application
crackers.

[+] detected 64bit ARM binary in memory.
```

```
[+] offset to cryptid found: @0x1000a0de0(from 0x1000a0000) = de0
[+] Found encrypted data at address 00004000 of length 53772288 bytes - type 1.
[+] Opening
/private/var/mobile/Containers/Bundle/Application/AB23846C-B727-4F56-B62D-3AAF74DDDFB7/
TargetApp.app/TargetApp for reading.
[+] Reading header
[+] Detecting header type
[+] Executable is a FAT image - searching for right architecture
[+] Correct arch is at offset 57802752 in the file
[+] Opening TargetApp.decrypted for writing.
[+] Copying the not encrypted start of the file
[+] Dumping the decrypted data into the file
[+] Copying the not encrypted remainder of the file
[+] Setting the LC_ENCRYPTION_INFO->cryptid to 0 at offset 3720de0
[+] Closing original file
[+] Closing dump file
```

解密之后会在当前目录下面生成 TargetApp.decrypted 文件。赶紧把它复制到 Mac 中备用吧。

如果将动态库复制到一个非沙盒目录下，例如 /var/tmp，运行后就会报如下错误。

```
monkeyde-iPhone:/var/tmp root# DYLD_INSERT_LIBRARIES=dumpdecrypted.dylib
/var/mobile/Containers/Bundle/Application/AB23846C-B727-4F56-B62D-3AAF74DDDFB7/Target.a
pp/Target
dyld: could not load inserted library 'dumpdecrypted.dylib' because no suitable image found.
Did find:
    dumpdecrypted.dylib: stat() failed with errno=1

Trace/BPT trap: 5
```

06 查看加密标识

加密与否是会在文件里面记录的。把刚刚解密的文件复制到 Mac 中，命令如下。

```
scp -P 2222
root@localhost:/var/mobile/Containers/Data/Application/651168C8-D728-4B83-A365-FE8A75D6
7C69/Documents/TargetApp.decrypted ./
```

然后，使用如下命令查看加密标识。

```
otool -l TargetApp.decrypted | grep crypt
TargetApp.decrypted (architecture armv7):
    cryptoff 16384
   cryptsize 38846464
```

```
     cryptid 1
TargetApp.decrypted (architecture arm64):
    cryptoff 16384
   cryptsize 43417600
     cryptid 0
```

可以看到，只有运行的架构被解密了（运行在 iPhone 5s 上，ARM64 架构），ARMv7 还是加密状态。这时需要执行 `lipo TargetApp.decrypted -thin arm64 -output TargetApp.arm64` 命令抽取指定的架构。

07　改进 dumpdecrypted

为了避免每次复制 dumpdecrypted.dylib 文件，并且对 App 里面使用的 framework 进行解密，笔者重新整理了开源的 dumpdecrypted 代码及 fork 的修改（GitHub 页面为 https://github.com/AloneMonkey/dumpdecrypted），将其编译，生成动态库，具体如下。

```
git clone https://github.com/AloneMonkey/dumpdecrypted.git
cd dumpdecrypted
make
```

在这里主要先借助 MobileLoader 进行注入，然后编写 dumpdecrypted.plist 文件，内容如下。其中，`target.bundle.id` 被替换成了目标应用的 Bundle ID。

```
{
        Filter = {
                Bundles = ("target.bundle.id");
        };
}
```

将 dumpdecrypted.plist 和 dumpdecrypted.dylib 复制到设备的 /Library/MobileSubstrate/DynamicLibraries/目录下，退出目标应用。重新打开目标应用，使用控制台（/Applications/Utilities/Console.app）查看设备的日志输入，即可看到解密后的文件在 Documents 中的保存路径，如图 3-1 所示。通过这种方式可以解密 Exteniosn 和依赖的 framework，所以，如果目标应用中有 framework，需要通过这种方式对 framework 进行解密。

dumpdecrypted 的最新版本基于 MonkeyDev（5.4 节会详细讲解）模板。直接在 dumpdecrypted.plist 中指定目标应用的 Bundle ID，然后按“Command + B”快捷键即可安装。同时，推荐使用 frida-ios-dump（https://github.com/AloneMonkey/frida-ios-dump），直接从越狱机器中提取解密后的 ipa 安装包。

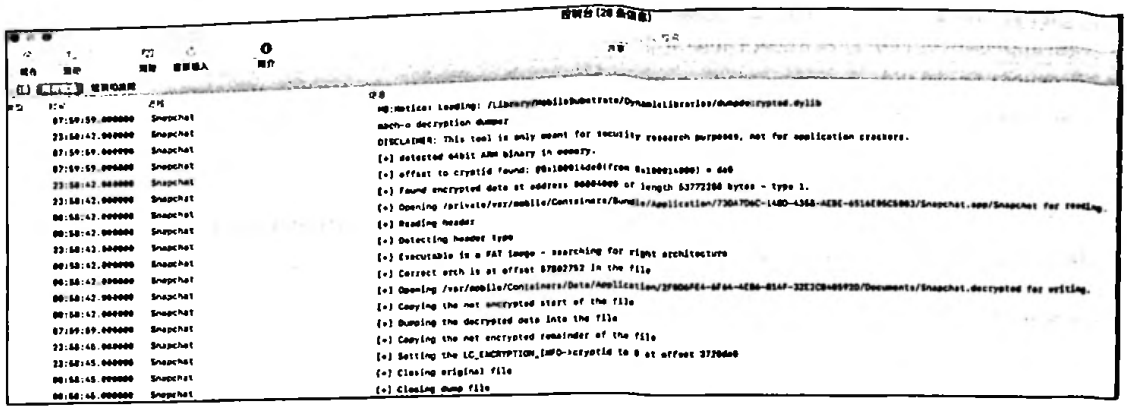


图 3-1 在控制台查看设备日志

3.1.2 Clutch

除了使用 dumpdecrypted，还可以使用开源工具 Clutch 来解密。与 dumpdecrypted 不同的是，Clutch 是生成一个新的进程，然后暂停进程并 dump 内存的。下面介绍 Clutch 解密的过程。

01 下载并编译 Clutch

首先从 Git 仓库（https://github.com/KJCracks/Clutch）中拉取项目，然后使用 Xcode 打开项目，使用 Debug 生成所有架构（Build Active Architecture Only - NO），运行目标设备来选择真机。按“Command + B”快捷键进行编译，如图 3-2 所示。

图 3-2 使用 Xcode 编译 Clutch 项目

该项目生成的是一个运行在移动设备上的命令行工具，内部脚本会把它的可执行文件复制出来，单独签名后放在 Build/Clutch 下。

02　将文件复制到设备中

使用如下命令，把生成的 Build/Clutch 文件复制到设备的/usr/bin/目录下，然后赋予其可执行权限。

```
scp -P 2222 ./Build/Clutch root@localhost:/usr/bin/
ssh 5s
chmod +x /usr/bin/Clutch
```

03　Clutch 解密

在设备终端运行 Clutch -i 命令，可以查看设备安装的应用和 Bundle ID。如果找不到目标应用，就使用上面的方法找出 Bundle ID，然后使用 Clutch -b target.bundle.id 命令解密，操作如下。

```
Monkey:~ root# Clutch -b target.bundle.id
2017-07-05 22:48:52.867 Clutch[2317:101815] command: Only dump binary files from specified
bundleID
Zipping TargetApp.app
Swapping architectures..
ASLR slide: 0x7c000
Dumping TargetApp (armv7)
Patched cryptid (32bit segment)
Writing new checksum
ASLR slide: 0x1000f4000
Dumping TargetApp (arm64)
Patched cryptid (64bit segment)
Writing new checksum
DONE: /private/var/mobile/Documents/Dumped/com.target.bundleid-iOS8.0-(Clutch-2.0.4
DEBUG).ipa
Finished dumping com.target.bundleid in 11.8 seconds
```

完成以上操作后，就可以在输出的目录下面找到 dump 出来的 ipa 文件，并将其导出到 Mac 中了。其实，ipa 就是一个压缩包。单击右键，在弹出的快捷菜单中选择“打开方式”→“归档实用工具”选项。解压后，可以在 Payload 文件夹下找到 TargetApp.app 的一个包，右键快捷菜单中显示了包的内容，可以查看应用里面的文件和资源。找到 Info.plist 文件，双击打开，其中 Bundle identifier 就是它的 Bundle ID，Executable file 就是可执行文件的名字。可以使用 lipo 和 otool 查看可执行文件的架构和加密标识。另外，Extension 也会被解密。

若在使用中出现错误，可以增加 --verbose 参数来查看详细的日志信息。

3.1.3 小结

本节介绍了两种解密方法：一种是基于 DYLD_INSERT_LIBRARIES 环境变量将动态库注入目标进程，然后 dump 内存；另一种是通过 posix_spawnp 创建一个进程，然后 dump 内存。由于使用 Clutch 解密一些应用时经常会出错，推荐使用修改后的 dumpdecrypted 进行解密。

3.2 class-dump

class-dump 是一个用于从可执行文件中获取类、方法和属性信息的工具，其代码是开源的（https://github.com/nygard/class-dump）。在分析过程中，通过该工具生成的头文件就可以快速找到想要的方法或属性。接下来，我们就一起学习 class-dump 的使用方法及原理。

3.2.1 class-dump 的使用

从 GitHub 上拉取 class-dump 的源代码，打开并编译，生成可执行文件，命令如下。

```
git clone https://github.com/nygard/class-dump.git
```

运行 class-dump，可以看到它的参数，具体如下。

```
➜  ./class-dump
class-dump 3.5 (64 bit) (Debug version compiled Jul  2 2017 23:46:40)
Usage: class-dump [options] <mach-o-file>

where options are:
        -a              show instance variable offsets
        -A              show implementation addresses
        --arch <arch>   choose a specific architecture from a universal binary (ppc, ppc64,
i386, x86_64, armv6, armv7, armv7s, arm64)
        -C <regex>      only display classes matching regular expression
        -f <str>        find string in method name
        -H              generate header files in current directory, or directory specified
with -o
        -I              sort classes, categories, and protocols by inheritance (overrides
-s)
        -o <dir>        output directory used for -H
        -r              recursively expand frameworks and fixed VM shared libraries
        -s              sort classes and categories by name
        -S              sort methods by name
        -t              suppress header in output, for testing
        --list-arches   list the arches in the file, then exit
```

```
        --sdk-ios      specify iOS SDK version (will look for
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/i
PhoneOS<version>.sdk
                                        or
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS<version>.sdk)
        --sdk-mac      specify Mac OS X version (will look for
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/Mac
OSX<version>.sdk
                                        or /Developer/SDKs/MacOSX<version>.sdk)
    --sdk-root     specify the full SDK root path (or use --sdk-ios/--sdk-mac for a shortcut)
```

使用 class-dump 导出目标应用指定架构的头文件到指定目录。目标可执行文件可以从 app 包下的 Info.plist 文件的 Executable file 中读取并获得。dump 的具体命令如下（假定 class-dump 和目标文件在同一文件夹下）。

```
./class-dump --arch arm64 TargetApp -H -o ./Headers
```

成功运行之后，就能在 Headers 目录下面看到程序的头文件了，这对逆向分析会有很大的帮助。但是，指定 ARMv7 架构时会报如下错误。下面我们就来解决这个错误。

```
2017-07-03 00:07:29.398 class-dump[6753:409389] *** Assertion failure in
-[CDObjectiveC2Processor loadMethodsAtAddress:extendedMethodTypesCursor:],
/Users/monkey/Documents/iosreversecode/class-dump/Source/CDObjectiveC2Processor.m:404
2017-07-03 00:07:29.400 class-dump[6753:409389] *** Terminating app due to uncaught
exception 'NSInternalInconsistencyException', reason: 'Invalid parameter not satisfying:
listHeader.entsize == 3 * [self.machOFile ptrSize]'
*** First throw call stack:
(
        0   CoreFoundation                      0x00007fff9682d2cb __exceptionPreprocess +
171
        1   libobjc.A.dylib                     0x00007fffab66648d objc_exception_throw +
48
        2   CoreFoundation                      0x00007fff96832042 +[NSException
raise:format:arguments:] + 98
        3   Foundation                          0x00007fff9827ac80 -[NSAssertionHandler
handleFailureInMethod:object:file:lineNumber:description:] + 195
        4   class-dump                          0x00000001016ae8ba
-[CDObjectiveC2Processor loadMethodsAtAddress:extendedMethodTypesCursor:] + 1018
        5   class-dump                          0x00000001016ae4b2
-[CDObjectiveC2Processor loadMethodsAtAddress:] + 66
        6   class-dump                          0x00000001016acb20
-[CDObjectiveC2Processor loadClassAtAddress:] + 2000
        7   class-dump                          0x00000001016a9f0c
```

```
-[CDObjectiveC2Processor loadClasses] + 284
        8   class-dump                          0x00000001016a592b -[CDObjectiveCProcessor
process] + 507
        9   class-dump                          0x000000010166eb9f -[CDClassDump
processObjectiveCData] + 399
        10  class-dump                          0x0000000101689b6f main + 4815
        11  libdyld.dylib                       0x00007fffabf4b235 start + 1
        12  ???                                 0x0000000000000007 0x0 + 7
)
libc++abi.dylib: terminating with uncaught exception of type NSException
```

要解决这个错误，可以使用 Xcode 来调试运行项目。因为运行的命令是带有参数的，所以要在 Xcode 运行程序时为它赋予启动参数。选中 class-dump Target，单击“Edit Scheme...”选项，如图 3-3 所示。

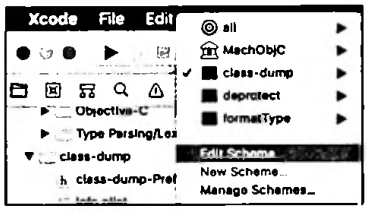


图 3-3 编辑 class-dump 的 Scheme

单击“Arguments”选项，填写启动参数，如图 3-4 所示，运行程序。

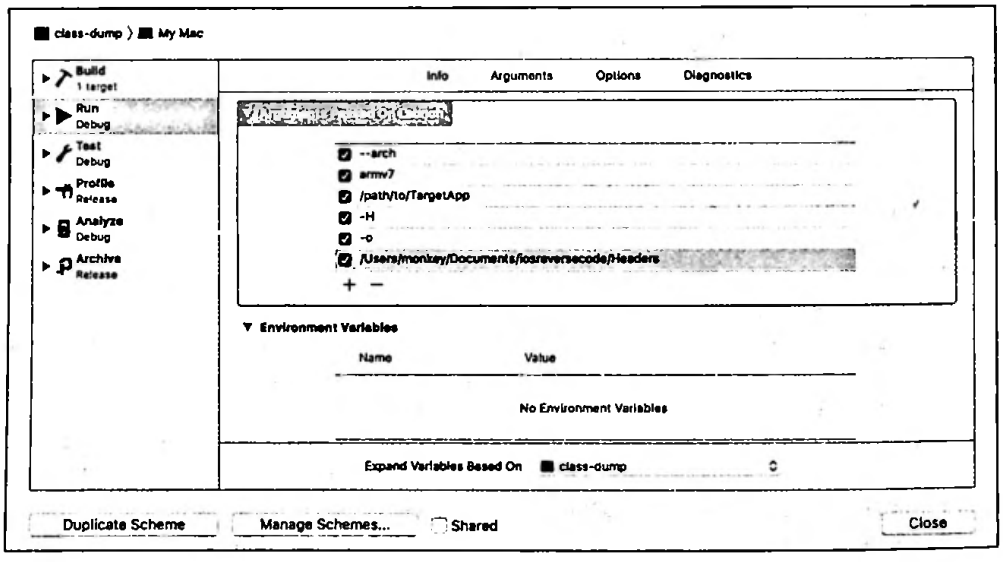


图 3-4 填写 class-dump 的启动参数

运行程序，在 CDObjectiveC2Processor.m 文件的 404 行 `NSParameterAssert(listHeader.entsize == 3 * [self.machOFile ptrSize]);` 报错。在调试中窗口看到，entsize 是一个非常大的数，是它导致了相等判断的失败。这时需要理解 class-dump 的原理，才能一步步分析原因。下面我们先分析 class-dump 的原理。

3.2.2　class-dump 的原理

分析 class-dump 的原理，可以通过一步步调试源代码来实现。因为其中涉及 Mach-O 文件结构的知识，所以读者也可以先阅读 6.2 节的相关内容。笔者对源代码做了注释，放在了 GitHub 上（https://github.com/AloneMonkey/class-dump），下面对其中几个重要部分进行讲解。为了讲解方便，笔者使用自己生成的一个 Mach-O 文件（ClassDumpApp），这个文件可以在随书源代码中找到。

入口文件是 class-dump.m。文件的前面是一些关于输入参数的处理，在这里就不多说了。我们来看下面这行代码。

```
CDFile *file = [CDFile fileWithContentsOfFile:executablePath
searchPathState:classDump.searchPathState];
```

这里是在解析 Mach-O 文件的头部，以判断是 FAT 文件还是单个架构的文件。我们主要通过如下调用去处理和解析 Mach-O 的数据。

```
[classDump processObjectiveCData];
        [processor process];
```

Mach-O 的数据里面分别是加载符号、解析协议列表、解析类列表、解析分类列表，相关代码如下。

```
if (self.machOFile.isEncrypted == NO && self.machOFile.canDecryptAllSegments) {
        //首先从 LC_SYMTAB 定位 Symbol Table，然后枚举符号表
        [self.machOFile.symbolTable loadSymbols];
        //读取 LC_DYSYMTAB
        [self.machOFile.dynamicSymbolTable loadSymbols];

        //从 __DATA,__objc_protolist 中读取解析协议列表
        [self loadProtocols];
        //合并
        [self.protocolUniquer createUniquedProtocols];
```

```
        //Load classes before categories, so we can get a dictionary of classes by address.
        //从 __DATA,__objc_classlist 中读取解析类列表
        [self loadClasses];
        //从 __DATA,__objc_catlist 中读取解析分类列表
        [self loadCategories];
}
```

笔者暂时不讲解符号表的解析，先讲一下协议、类和方法的解析。在阅读以下内容前，请下载 MachOView。这是一个可以查看 Mach-O 文件结构的界面工具，可以访问 https://sourceforge.net/projects/machoview/ 下载 dmg，也可以访问 https://github.com/gdbinit/MachOView 下载其代码并进行编译。下面从 3 个角度分别阐述其原理。

1. 协议解析

通过跟进方法 loadProtocols 可以看到，首先要读取 __objc_protolist 节的内容，然后要根据这个节里面记录的 protocol 的地址列表分别去解析它的结构。打开 MachOView，找到 __objc_protolist 节，从中可以看到它的地址列表，如图 3-5 所示。

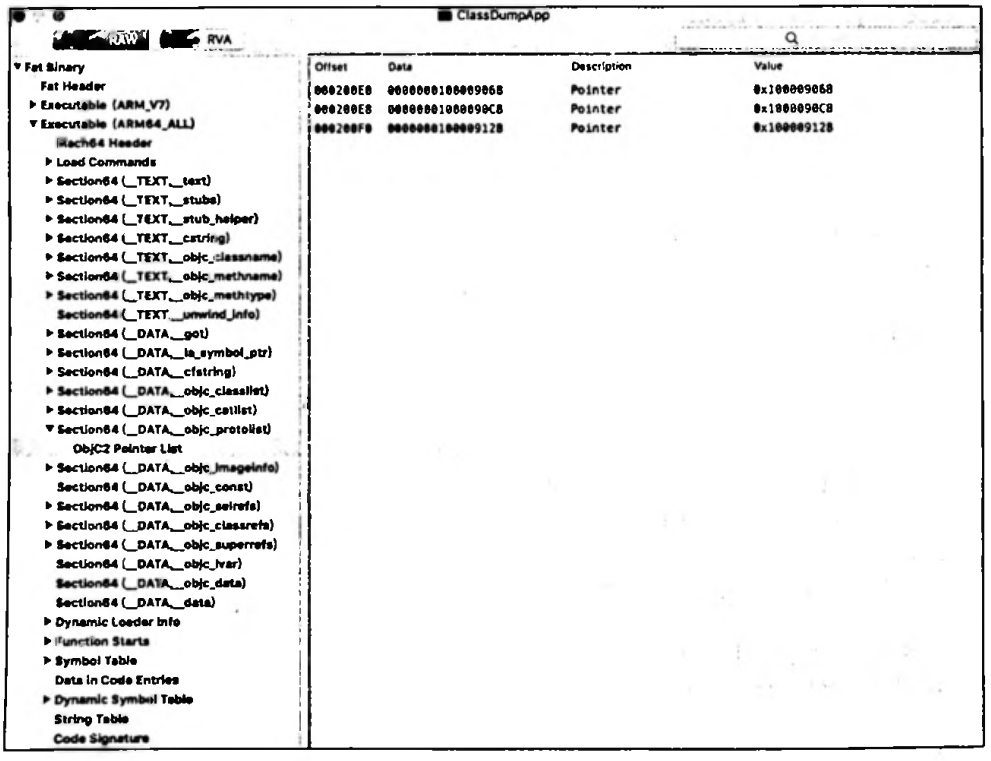


图 3-5 使用 MachOView 查看__objc_protolist 节

第 1 个地址是 0000000100009068。根据 protocolAtAddress 函数查看其结构，具体如下。

```
struct cd_objc2_protocol objc2Protocol;
objc2Protocol.isa                    = [cursor readPtr];
objc2Protocol.name                   = [cursor readPtr];
objc2Protocol.protocols               = [cursor readPtr];
objc2Protocol.instanceMethods         = [cursor readPtr];
objc2Protocol.classMethods            = [cursor readPtr];
objc2Protocol.optionalInstanceMethods = [cursor readPtr];
objc2Protocol.optionalClassMethods    = [cursor readPtr];
objc2Protocol.instanceProperties      = [cursor readPtr];
objc2Protocol.size                   = [cursor readInt32];
objc2Protocol.flags                  = [cursor readInt32];
objc2Protocol.extendedMethodTypes     = 0;
```

同时，可以在苹果开源代码中找到对应的结构（https://opensource.apple.com/source/objc4/objc4-723/runtime/objc-runtime-new.h.auto.html）。这里对应的结构如下，struct 继承的父类结构在子类的前面。

```
struct protocol_t : objc_object {
    // Class ISA;
    const char *mangledName;
    struct protocol_list_t *protocols;
    method_list_t *instanceMethods;
    method_list_t *classMethods;
    method_list_t *optionalInstanceMethods;
    method_list_t *optionalClassMethods;
    property_list_t *instanceProperties;
    uint32_t size;   // sizeof(protocol_t)
    uint32_t flags;
    ......
```

这里获取的地址 0000000100009068 是虚拟地址，需要将它转换成文件偏移地址。计算公式如下。

文件偏移地址 = 虚拟地址 − 模块在内存中地址 + 模块在文件中的偏移

模块加载在内存中的地址可以在 __TEXT 段找到，是 0000000100000000。当然，在实际内存中还要加上 ASLR（地址空间随机化）的偏移。通过 MachOView 读取当前模块（ARM64 架构）在文件中的偏移，是 0x00018000，如图 3-6 所示。

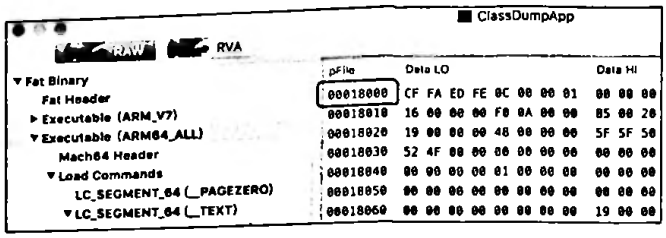


图 3-6 当前模块（ARM64 架构）在文件中的偏移

计算公式如下，结果为 0x21068。对应到文件中的偏移位置就是 (DATA,data) 的位置。从该位置读取 name 的地址，是 0x0000000100006c40。

0000000100009068 - 0000000100000000 + 0x00018000 = 0x21068

接下来，计算这个地址所对应的字符串。这也是一个虚拟地址。根据上面的算法，得到其在文件中的偏移，是 0x1EC40。使用 MachOView 查看对应位置的字符串，是 NSObject，如图 3-7 所示。

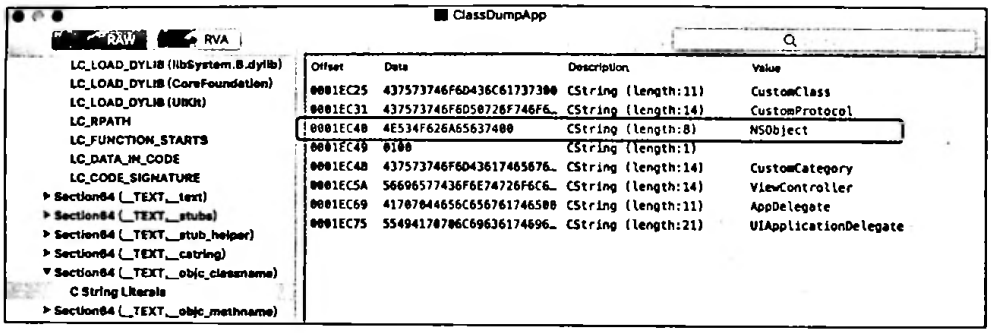


图 3-7 协议名称在 Mach-O 中的位置

协议的其他部分留给读者自己分析。

2. 类解析

类解析和协议解析类似。从 Mach-O 文件的 __objc_classlist 节中读取类的列表地址，如图 3-8 所示。虽然 Mach-O 已经将其解析出来了，但我们还是要了解解析的原理。

第 1 个地址是 0000000100008FA0。根据 loadClassAtAddress 函数查看它的结构，具体如下。

```
struct cd_objc2_class objc2Class;
objc2Class.isa        = [cursor readPtr];
objc2Class.superclass = [cursor readPtr];
objc2Class.cache      = [cursor readPtr];
```

```
objc2Class.vtable     = [cursor readPtr];

uint64_t value        = [cursor readPtr];
class.isSwiftClass    = (value & 0x1) != 0;
objc2Class.data       = value & ~7;

objc2Class.reserved1  = [cursor readPtr];
objc2Class.reserved2  = [cursor readPtr];
objc2Class.reserved3  = [cursor readPtr];

NSParameterAssert(objc2Class.data != 0);
[cursor setAddress:objc2Class.data];

struct cd_objc2_class_ro_t objc2ClassData;
objc2ClassData.flags         = [cursor readInt32];
objc2ClassData.instanceStart = [cursor readInt32];
objc2ClassData.instanceSize  = [cursor readInt32];
if ([self.machOFile uses64BitABI])
        objc2ClassData.reserved  = [cursor readInt32];
else
        objc2ClassData.reserved = 0;

objc2ClassData.ivarLayout     = [cursor readPtr];
objc2ClassData.name           = [cursor readPtr];
objc2ClassData.baseMethods    = [cursor readPtr];
objc2ClassData.baseProtocols  = [cursor readPtr];
objc2ClassData.ivars          = [cursor readPtr];
objc2ClassData.weakIvarLayout = [cursor readPtr];
objc2ClassData.baseProperties = [cursor readPtr];
```

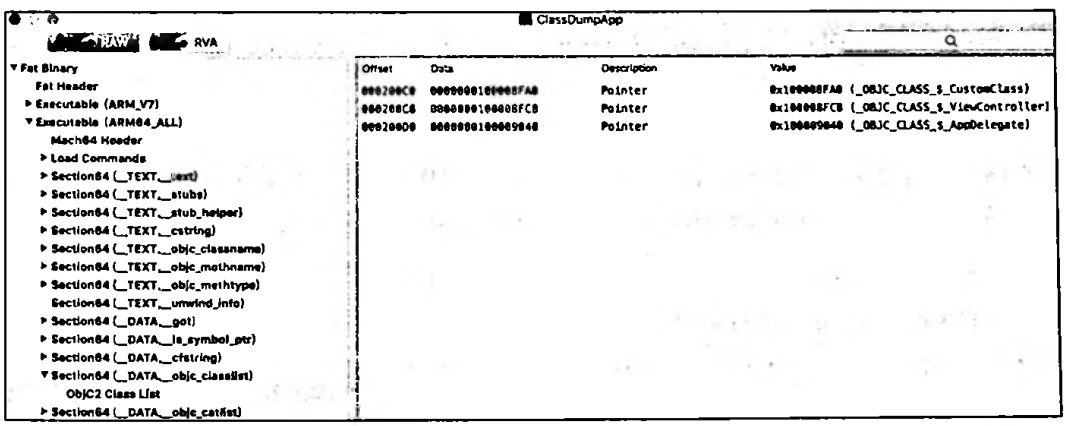


图 3-8　使用 MachOView 查看__objc_classlist 节

对应的结构如下。

```
struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
#ifdef __LP64__
    uint32_t reserved;
#endif

    const uint8_t * ivarLayout;

    const char * name;
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;

    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;

    method_list_t *baseMethods() const {
        return baseMethodList;
    }
};

struct objc_class : objc_object {
    //Class ISA;
    Class superclass;
    cache_t cache;                  //formerly cache pointer and vtable
    //struct bucket_t *_buckets;    //cache
    //mask_t _mask;                 //vtable
    //mask_t _occupied;             //vtable
    class_data_bits_t bits;         //class_rw_t * plus custom rr/alloc flags
    ......
```

根据上面的公式计算它在文件中的偏移，是 0x20FA0。然后，根据 bits 读取的地址 0x0000000100008578（该地址与 (&)~7 的计算结果还是 0x0000000100008578），转换成文件中的偏移，是 0x20578。再读取 name 的地址 0x0000000100006c25，转换成文件中的偏移，是 0x1EC25。所以，对应的字符串是 CustomClass，如图 3-9 所示。

经调试发现，在 dump ARMv7 架构时，下面这行代码会导致 data 读取出错。

```
objc2Class.data      = value & ~7;
```

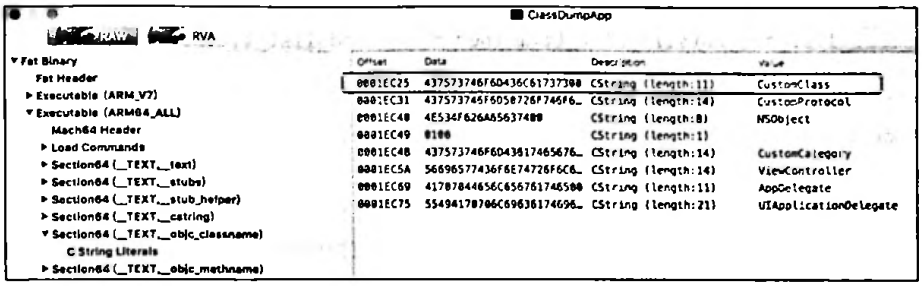


图 3-9　使用 MachOView 查看类名

而根据开源头文件中的如下代码：

```
class_rw_t* data() {
    return (class_rw_t *)(bits & FAST_DATA_MASK);
}

#if !__LP64__
......
//data pointer
#define FAST_DATA_MASK        0xfffffffcUL
#elif 1
//data pointer
#define FAST_DATA_MASK         0x00007ffffffffff8UL
......
```

在编译期间 bits 指向 class_ro_t 的结构。所以，在这里需要对 32 位和 64 位系统进行不同的操作。修改代码如下。

```
//    objc2Class.data      = value & ~7;

    if ([self.machOFile uses64BitABI])
        objc2Class.data      = value & ~7;
    else
        objc2Class.data      = value & ~3;
```

重新编译，再次 dump ARMv7 架构，结果正常。

3. 方法解析

前面在进行类解析时，有一个 baseMethods 属性，其中保存的就是类的实例方法。前面读取的类是 CustomClass，对应读取的 baseMethods 的值是 0x0000000100008478，对应的文件偏移为 0x20478。这里对应的 method_list_t 结构如下。

```
struct method_list_t : entsize_list_tt<method_t, method_list_t, 0x3> {
    //uint32_t entsizeAndFlags;
    //uint32_t count;
    //method_t first;
    ......
}

struct method_t {
    SEL name;
    const char *types;
    IMP imp;
}
```

从文件中读取的方法的个数为 5。第 1 个读取的 name 是 0x0000000100006d9c，这是方法的名字，对应的文件偏移为 0x1ED9C。从 MachOView 里面可以看到，对应的内容是“protocol DoSomething”，也就是类实现的协议的方法名称，如图 3-10 所示。

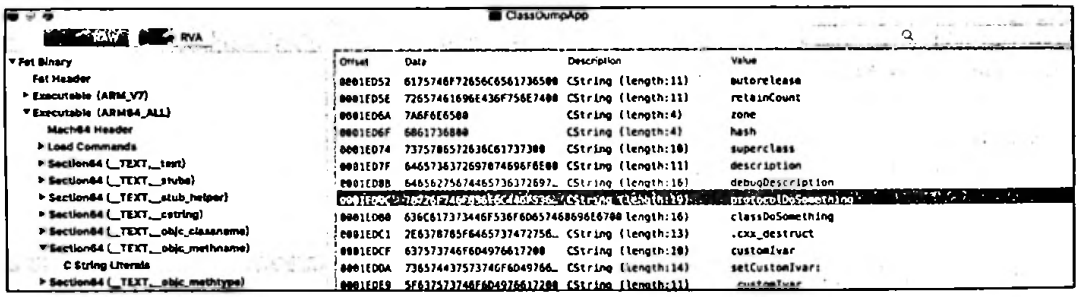


图 3-10　使用 MachOView 查看方法名

解析之后，就是关于参数 type 的处理和头文件的生成了，这里不再细说。

整个 class-dump 的原理就是这样。读者可以在学习 Mach-O 的结构之后重新阅读本节，以便对类的存储和文件结构有更深入的了解。

3.2.3　OC 和 Swift 混编

由于 Swift 的出现，部分应用已经使用 OC 和 Swift 混编的方法，而且很多新开发的应用已经全部使用 Swift 进行编写了。下面我们来看看 class-dump 对 OC 和 Swift 混编应用 dump 的结果。对单纯使用 OC 编写的 ClassDumpApp 新建如下 Swift 文件。

```
import Foundation
```

```
class SwiftClass : NSObject {
    func swiftMethod() -> String{
        return "SwiftMethod"
    }
}
```

重新使用 class-dump 获取它的头文件，具体如下。

```
./class-dump ClassDumpApp -H
```

在获取的头文件中，有一个头文件为 _TtC12ClassDumpApp10SwiftClass.h，其内容如下。

```
#import <objc/NSObject.h>

@interface _TtC12ClassDumpApp10SwiftClass : NSObject
{
}

- (id)init;
- (id)swiftMethod;

@end
```

_TtC12ClassDumpApp10SwiftClass 就是刚刚创建的 Swift 类。使用如下命令对其进行格式化。

```
xcrun swift-demangle -compact _TtC12ClassDumpApp10SwiftClass
ClassDumpApp.SwiftClass
```

因为 ClassDumpApp 是它的 namespace，所以在 Swift 中不同的库不会出现类名冲突。但是，如果要获取这个类，就要使用 objc_getClass("_TtC12ClassDumpApp10SwiftClass") 了。

3.3 Reveal

Reveal 是一个用于查看程序界面结构和调试界面的工具，其官网地址为 https://revealapp.com/。Reveal 可以在开发过程中动态调试修改程序的样式，也可以注入第三方 App 以查看应用的界面结构。本节分别讲解如何在开发过程中和在越狱设备上查看应用的结构。

3.3.1 开发集成 Reveal

开发集成 Reveal 最简单的一种方法就是通过 CocosPod 集成。在 Podfile 中加入如下内容，然后运行 pod install（configurations 设置表示只在 Debug 模式下开启）。

```
target 'RevealApp' do
    pod 'Reveal-SDK', :configurations => ['Debug']
end
```

因为新版的 Reveal 支持 USB 连接，各设备不用在同一 Wi-Fi 网络环境下工作，所以 Reveal 运行后就能看到可以选择的目标 App 了。选择运行的 App，就能看到 App 的界面结构，如图 3-11 所示。

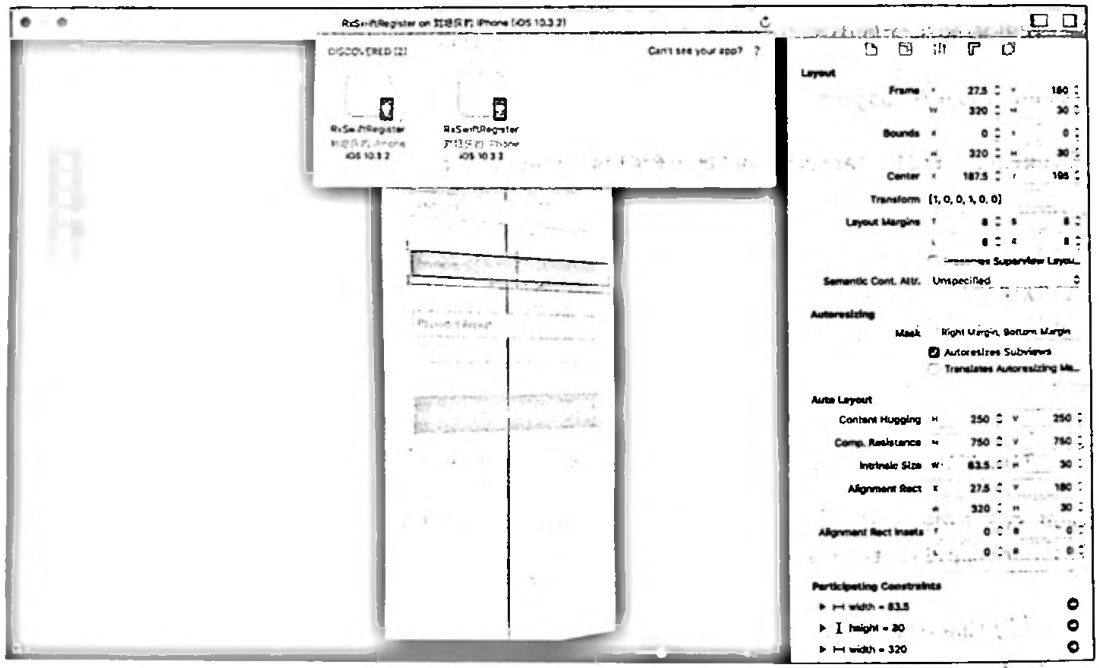


图 3-11 集成 Reveal 查看应用界面

单击“Help-Integration Guide”选项，可以看到官方文档里面还介绍了手动链接和通过 LLDB 脚本加载的方式，读者可以自行尝试。

3.3.2 越狱注入 Reveal

为了查看和调试应用的界面结构，需要让目标程序加载 RevealServer。在越狱设备上，可以通过 MobileLoader 将 RevealServer 注入目标程序。将 RevealServer 重命名为 libReveal.dylib，新建 libReveal.plist 文件，在文件中指定要注入的目标应用的 Bundle ID。指定 App Store 的 Bundle ID 为“com.apple.AppStore”，内容如下。

```
{
        Filter = {
                Bundles = (
                        "com.apple.AppStore",
                );
        };
}
```

将这两个文件复制到手机的 /Library/MobileSubstrate/DynamicLibraries/目录下，重启 App Store，成功注入后就能在 Reveal 中看到目标应用了。注入效果如图 3-12 所示。对其他应用也可以这样操作，这对分析第三方应用的界面来说意义非常大。

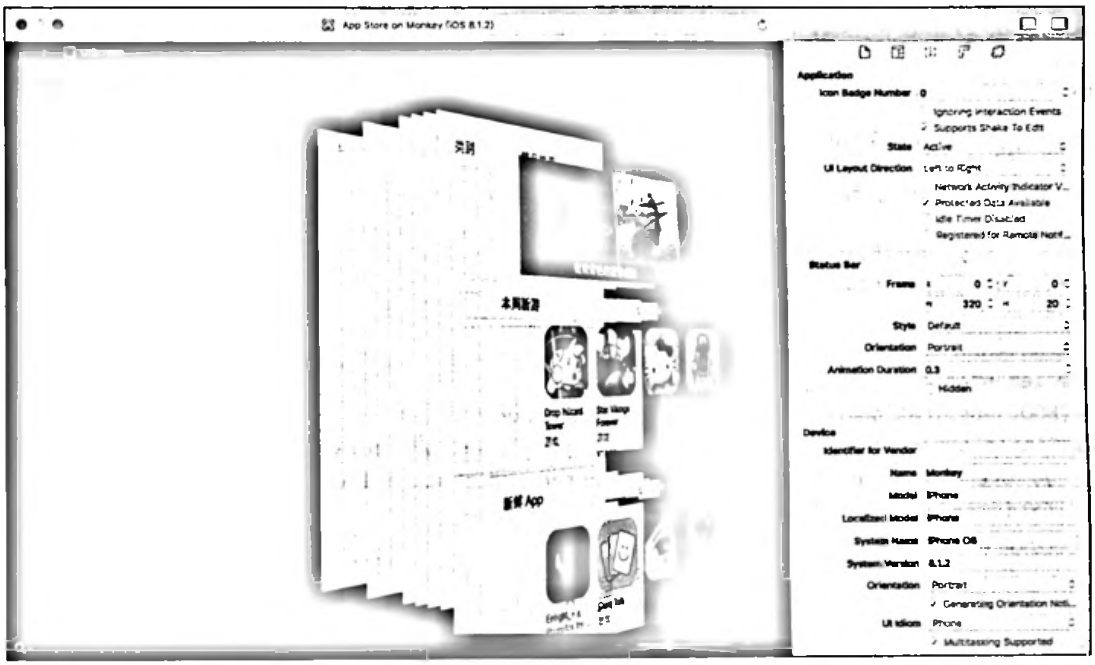


图 3-12　注入 App Store 查看界面布局

3.4　Cycript

Cycript 是一个允许开发者使用 Objective-C++ 和 JavaScript 组合语法查看及修改运行时 App 内存信息的工具，其官网（http://www.cycript.org/）提供了它的一些经典使用方法，如图 3-13 所示。

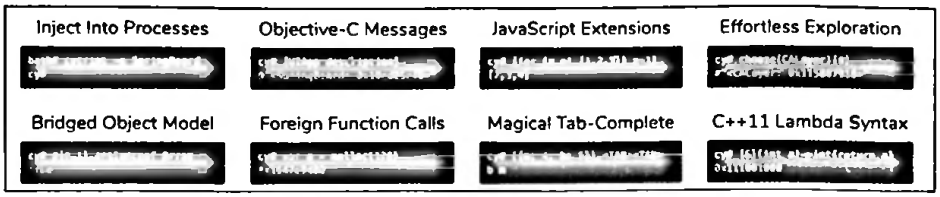


图 3-13 Cycript 的一些使用方法

Cycript 也可以在我们自己的应用中和越狱设备上集成使用。下面讲解 Cycript 的集成及使用技巧。

3.4.1 开发集成 Cycript

开发集成 Cycript 的方法比较简单。从官网下载 SDK 包并解压，将 Cycript.ios 文件夹中的 Cycript.framework 添加到项目列表中，分别添加 JavaScriptCore、libstdc++.tbd、lisqlite3.0.tbd，然后在 AppDelegate 文件中导入头文件，在 `application:didFinishLaunchingWithOptions:` 方法中调用 `CYListenServer(8888)` 启动服务，具体如下。

```
#import <Cycript/Cycript.h>
@interface AppDelegate ()
@end

@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    CYListenServer(8888);
    return YES;
}
......
@end
```

运行程序，找到最新版的 Cycript.framework（cycript_0.9.594）并运行，会报如下错误。

```
libc++abi.dylib: terminating with uncaught exception of type CYPoolError
```

笔者用一个旧版的库进行替换，就解决了问题。重新运行，执行 SDK 包下面的 Cycript 文件，就可以进入 Cycript 的交互界面了，命令如下（IP 地址为设备的 IP 地址）。

```
./cycript -r 192.168.1.109:8888
```

3.4.2　使用 Cycript 越狱

Cycript 也提供了越狱工具。在 Cydia 中搜索 Cycript 并安装，如图 3-14 所示。

图 3-14　在 Cydia 中安装 Cycript

安装完成，通过 SSH 登录手机，输入 cycript 命令，就能进入交互界面并输入简单的脚本了，具体如下。

```
➜  ~ ssh 5s
Monkey:~ root# cycript
cy# var a = 3
3
cy# var b = 4
4
cy# a + b
7
```

在越狱设备上，可以使用如下命令注入目标进程、调用目标函数。这个脚本通过注入 SpringBoard 弹出一个提示窗口。执行如下命令后，可以在手机屏幕上看到一个弹窗。

```
➜  ~ ssh 5s
Monkey:~ root# cycript -p SpringBoard
cy# var alert = [[UIAlertView alloc] initWithTitle:@"hi" message:@"hello,world!"
delegate:nil cancelButtonTitle:@"cancel" otherButtonTitles:nil]
```

```
#"<UIAlertView: 0x157a73910; frame = (0 0; 0 0); layer = <CALayer: 0x174a2f920>>"
cy# [alert show]
```

3.4.3 使用 Cycript 分析应用

在逆向过程中，我们经常会使用 Cycript 注入程序、查看程序界面、调用程序函数验证自己的想法等。下面以一个 Demo App 为例，使用 Cycript 简单分析一个应用程序，并介绍 Cycript 的一些使用技巧（Demo App 可以在随书源代码中找到）。

打开目标 App 的首页界面，如图 3-15 所示，这是一个登录界面。输入用户名和密码，点击“登录”按钮，输入正确则登录成功，否则登录失败。

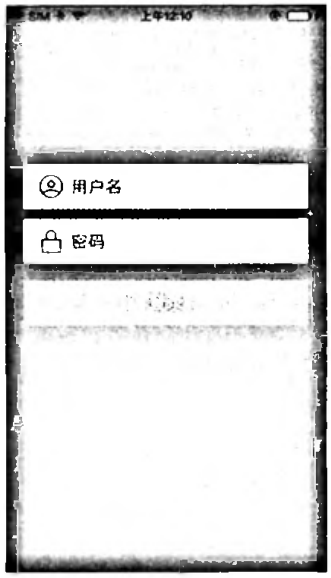


图 3-15 CycriptDemo 登录界面

通过 cycript -p CycriptDemo 命令注入目标 App，进入命令行交互模式，即可查看应用的一些基本信息，例如沙盒目录、当前界面布局、类的方法，下面逐一介绍。

1. 查看应用信息

查看应用信息，具体如下。

```
cy# NSHomeDirectory()
cy# [[NSBundle mainBundle] bundleIdentifier]
cy# ?exit
```

前两行分别是获取沙盒目录和 Bundle ID 的方法。将得到的沙盒目录加上“/Documents”或“/Library”，就能得到对应的 Documents 和 Library 目录。在执行脚本时要确保被注入的应用在前台运行。最后，使用 ?exit 命令退出。

2. 分析当前应用

可以直接在脚本中调用 OC 的函数来获取和修改应用的一些信息，具体如下。

```
cy# UIApp
#"<UIApplication: 0x15cd0b160>"
cy# UIApp.delegate
#"<AppDelegate: 0x17000cb10>"
cy# UIApp.keyWindow
#"<UIWindow: 0x15ce1aad0; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x170058900>;
layer = <UIWindowLayer: 0x170229dc0>>"
cy# UIApp.keyWindow.rootViewController
#"<LoginViewController: 0x15cd0dfd0>"
cy# #0x15cd0dfd0.view
#"<UIView: 0x170194290; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer:
0x17022a220>>"
cy# #0x15cd0dfd0.view.backgroundColor = [UIColor redColor]
cy# var loginvc = new Instance(0x15cd0dfd0)
cy# var loginvc = #0x15cd0dfd0
cy# *loginvc
cy# *#0x15cd0dfd0
```

还可以使用如下脚本来打印当前界面的结构和 ViewController 的继承关系。

```
UIApp.keyWindow.recursiveDescription().toString()
[[[UIWindow keyWindow] rootViewController] _printHierarchy].toString()
[[UIApp keyWindow] _autolayoutTrace].toString()
```

笔者安装的 Cycript 0.9.594 在打印界面结构时会出现打印不完整的情况，这时可以直接使用 Reveal 查看。还可以通过 choose() 函数来选择某个类的实例数组。例如，在界面上看到一个登录按钮，就使用 choose(UIButton) 函数来打印所有 UIButton 的实例，具体如下。

```
cy# choose(UIButton)
[#"<CDLoginInputView: 0x15cd14380; baseClass = UIButton; frame = (17.5 160; 285 45); opaque
= NO; layer = <CALayer: 0x17403f9e0>>",#"<UIButton: 0x15cd145a0; frame = (17.5 277; 285 45);
opaque = NO; layer = <CALayer: 0x174220a60>>",#"<CDLoginInputView: 0x15cd18ad0; baseClass
= UIButton; frame = (17.5 215; 285 45); opaque = NO; layer = <CALayer:
0x1742206e0>>",#"<UIButton: 0x15cd1a230; frame = (206 8; 19 19); opaque = NO; layer = <CALayer:
```

```
0x174221200>>",#"<UIButton: 0x15cd281e0; frame = (206 8; 19 19); opaque = NO; layer = <CALayer
0x1742239a0>>"]
```

从登录按钮的位置可以猜测，frame 为 (17.5 277; 285 45) 的 UIButton 就是登录按钮。这一点可以通过改变它的 hidden 属性来进一步确认——执行后，登录按钮消失，说明找对了，具体如下。

```
#0x15cd145a0.hidden=YES
```

获取修改显示文字及打印 nextResponder 响应链或者 Action 方法等，具体如下。

```
cy# [#0x15cd145a0 titleForState:UIControlStateNormal]
@"\xe7\x99\xbb\xe5\xbd\x95"
cy# [#0x15cd145a0 setTitle:@"AloneMonkey" forState:UIControlStateNormal]
cy# [#0x15cd145a0 nextResponder]
#"<UIView: 0x170194290; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer:
0x17022a220>>"
cy# [#0x170194290 nextResponder]
#"<LoginViewController: 0x15cd0dfd0>"
cy# [#0x15cd145a0 allTargets]
[NSSet setWithArray:@[#"<LoginViewController: 0x15cd0dfd0>"]]]
cy# [#0x15cd145a0 actionsForTarget:#0x15cd0dfd0
forControlEvent:UIControlEventTouchUpInside]
@["Login"]
```

Cycript 不支持直接显示中文。可以使用如下命令来查看中文内容。

```
➜  ~ echo -e "\xe7\x99\xbb\xe5\xbd\x95"
登录
➜  ~ python
Python 2.7.13 (default, Apr  4 2017, 08:47:57)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print "\xe7\x99\xbb\xe5\xbd\x95"
登录
>>> print u'\u767b\u5f55'
登录
```

3. 获取属性和方法

获取变量时，除了使用 *loginvc 方法，还可以使用 [i for (i in *UIApp)] 方法或者定义一个函数去打印一个对象的变量，具体如下。

```
cy# function tryPrintIvars(a){ var x={}; for(i in *a){ try{ x[i] = (*a)[i]; } catch(e){} }
```

```
return x; }
cy# tryPrintIvars(loginvc)
```

通过定义函数来获取类的方法如下。

```
function printMethods(className, isa) {
  var count = new new Type("I");
  var classObj = (isa != undefined) ? objc_getClass(className).constructor :
objc_getClass(className);
  var methods = class_copyMethodList(classObj, count);
  var methodsArray = [];
  for(var i = 0; i < *count; i++) {
    var method = methods[i];
    methodsArray.push({selector:method_getName(method),
implementation:method_getImplementation(method)});
  }
  free(methods);
  return methodsArray;
}

cy# printMethods("LoginViewController")
[{selector:@selector(setViews),implementation:&(extern "C" id "-[LoginViewController
setViews]"(id,
SEL, ...))}...{selector:@selector(LoginWithUsername:AndPassword:),implementation:&(exte
rn "C" id "-[LoginViewController LoginWithUsername:AndPassword:]"(id, SEL, ...))}...
```

在输出结果中会看到一个 LoginWithUsername:AndPassword: 方法，这时可以直接通过如下脚本去调用该方法来验证是否为登录的调用。调用后会出现密码不正确的提示，说明这个方法是通过传入用户名和密码进行登录验证的。

```
[#0x15cd0dfd0 LoginWithUsername:@"username" AndPassword:@"password"]
```

除此之外，可以通过如下命令调用打印属性和方法，其中 0x15cd145a0 是登录按钮对象的地址。

```
[#0x15cd145a0 _ivarDescription].toString()
[#0x15cd145a0 _methodDescription].toString()
```

3.4.4 Cycript 的高级用法

Cycript 本身支持加载自己的脚本，可以通过如下命令来加载。

```
cycript -p CycriptDemo /var/root/utils.cy
cycript -p CycriptDemo
```

这样，Cycript 就可以把自己常用的一些函数写在一个文件里面了。后续需要使用时直接调用，会方便很多。如下脚本将 3.4.3 节讲到的几个函数放在了一起。

```
// /var/root/utils.cy
function pviews(){
        return UIApp.keyWindow.recursiveDescription().toString();
};

function pvcs(){
        return [[[UIWindow keyWindow] rootViewController] _printHierarchy].toString();
};

function printIvars(a){
        var x={};
        for(i in *a){
                try{
                        x[i] = (*a)[i];
                }catch(e){

                }
        }
        return x;
};

function printMethods(className, isa) {
        var count = new new Type("I");
        var classObj = (isa != undefined) ? objc_getClass(className).constructor :
objc_getClass(className);
        var methods = class_copyMethodList(classObj, count);
        var methodsArray = [];
        for(var i = 0; i < *count; i++) {
                var method = methods[i];
                methodsArray.push({selector:method_getName(method),
implementation:method_getImplementation(method)});
        }
        free(methods);
        return methodsArray;
};
```

将该文件复制到设备中，通过以上方式加载，就可以直接调用 pviews() 函数打印当前界面的结构了。

此外，可以将加载命令简化。新建文件 /etc/profile.d/cycript.sh，写入如下内容。

```
cyc () { cycript -p $1 /var/root/utils.cy > /dev/null; cycript -p $1; }
```

执行 source /etc/profile.d/cycript.sh 命令之后，就可以直接使用 cyc CycriptDemo 这种方式来注入了。不过，这样做可能会导致多次注入，使脚本多次加载，从而衍生出一些问题。

还有一种方式可以新建脚本文件 utils.cy，具体如下。

```
(function(utils) {
        var c = utils.constants = {};

        c.pviews = function(){
                return UIApp.keyWindow.recursiveDescription().toString();
        };

        c.pvcs = function(){
                return [[[UIWindow keyWindow] rootViewController]
_printHierarchy].toString();
        };

        for(var k in utils.constants) {
                Cycript.all[k] = utils.constants[k];
        }
})(exports);
```

在 /usr/lib/cycript0.9/com 文件夹下新建文件夹 monkey，把上面的 utils.cy 文件复制到 monkey 文件夹下。以后如果需要使用该文件，就可以在 Cycript 里面通过如下命令进行导入了。

```
@import com.monkey.utils;
pviews()
```

我们在分析中经常会遇到使用 nextResponder 一直打印一个对象的响应链的情况，其实只需要写一个方法就可以解决这个问题。在分析时直接调用 rp(#对象地址)，具体如下。

```
c.rp = function(target){
    var result = "" + target.toString();
    while(target.nextResponder){
        result += "\n" + target.nextResponder.toString();
        target = target.nextResponder;
```

```
    }
    return result;
};
```

使用方式如下所示。

```
iPhone:~ root# cycript -p CycriptDemo
cy# @import com.monkey.utils;
{constants:{pviews:function (){return
UIApp.keyWindow.recursiveDescription().toString()},pvcs:function (){return
objc_msgSend(objc_msgSend(objc_msgSend(UIWindow,"keyWindow"),"rootViewController"),"_pr
intHierarchy").toString()},rp:function (t){var
e;e=""+t.toString();while(t.nextResponder){e+="\n"+t.nextResponder.toString();t=t.nextR
esponder}return e}}}
cy# choose(UIButton)
[#"<UIButton: 0x14de02830; frame = (206 8; 19 19); opaque = NO; layer = <CALayer:
0x17002ec40>>",#"<CDLoginInputView: 0x14dd26ab0; baseClass = UIButton; frame = (45 160; 285
45); opaque = NO; layer = <CALayer: 0x17402dd60>>",#"<CDLoginInputView: 0x14dd2b120;
baseClass = UIButton; frame = (45 215; 285 45); opaque = NO; layer = <CALayer:
0x17402da00>>",#"<UIButton: 0x14dd2cb20; frame = (45 277; 285 45); opaque = NO; layer =
<CALayer: 0x17402ed60>>",#"<UIButton: 0x14dd2deb0; frame = (206 8; 19 19); opaque = NO; layer
= <CALayer: 0x17402fc00>>"]
cy# rp(#0x14dd2cb20)
`<UIButton: 0x14dd2cb20; frame = (45 277; 285 45); opaque = NO; layer = <CALayer: 0x17402ed60>>
<UIView: 0x14dd265d0; frame = (0 0; 375 667); autoresize
 = W+H; layer = <CALayer: 0x17402dfc0>>
<LoginViewController: 0x14de02f70>
<UIWindow: 0x14de0c1e0; frame = (0 0; 375 667); gestureRecognizers = <NSArray: 0x170058a20>;
layer = <UIWindowLayer: 0x17002c3c0>>
<UIApplication: 0x14dd01920>
<AppDelegate: 0x17402a7a0>`
cy#
```

3.5 抓包

网络抓包是指对网络传输中发送和接收的数据包进行截获、重发、编辑、转存操作的过程。在分析应用的过程中，常常需要通过抓取网络数据包来获取一些传输信息，验证传输格式。在正向开发中，也常常会使用网络抓包来验证发送的数据是否正确或者模拟网络的返回结果等。

在不同的平台上，根据不同的原理，有不同的抓包工具，例如 Fiddler、Charles、Burp Suite、Wireshark。本节主要通过 Charles 和 Wireshark 来讲解抓包工具的使用。

3.5.1　Charles 抓包

Charles 是一个支持截取 HTTP/HTTPS 网络包的跨平台软件。除了截取网络包，Charles 还具有上面提到的重发网络请求、修改网络参数、模拟网络环境等功能。在 Charles 的官网（https://www.charlesproxy.com/）可以找到它的下载地址。在本书中，我们下载并安装 Mac 平台的 dmg 文件。

Charles 抓包的原理是将自己设为代理。代理相当于一个中间人，客户端发送的所有请求都会先经过代理，然后由代理发送给服务器。服务器收到请求后，将响应消息返回代理，然后由代理发送给客户端。对客户端来说代理就是服务器，对服务器来说代理就是客户端，所以两者都不会感知网络被拦截了（这也是中间人攻击的原理），如图 3-16 所示。

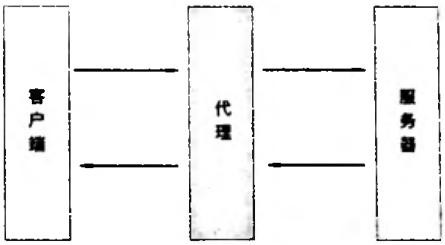


图 3-16　代理抓包原理

打开 Charles 后，首先需要用户授予设置系统代理的权限，这时单击授权并输入密码即可。进入主界面，可以在“Proxy”菜单中看到“macOS Proxy”子菜单已经被勾选了，如图 3-17 所示。设置为系统代理后，就能看到不断有请求显示出现的左侧窗口了。

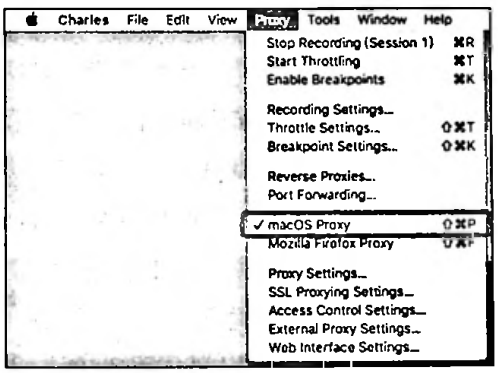


图 3-17　Charles 设置系统代理

需要注意的是，Chrome 浏览器默认不使用系统的代理设置，所以，要抓到 Chrome 发出的请

求，需要另外将 Chrome 的代理服务器设置为系统代理 localhost:8888（这可以通过 Chrome 的扩展插件 SwitchyOmega 实现）。

回到我们的正题——如何拦截 iPhone 设备上的网络请求上来。要拦截 iPhone 设备上的网络请求，首先要将 Charles 的代理功能打开。单击 Charles 窗口右上方的齿轮按钮，选择“Proxy Settings...”选项，设置代理端口并勾选“Enable transparent HTTP proxying”选项，如图 3-18 所示，完成 Charles 的设置。

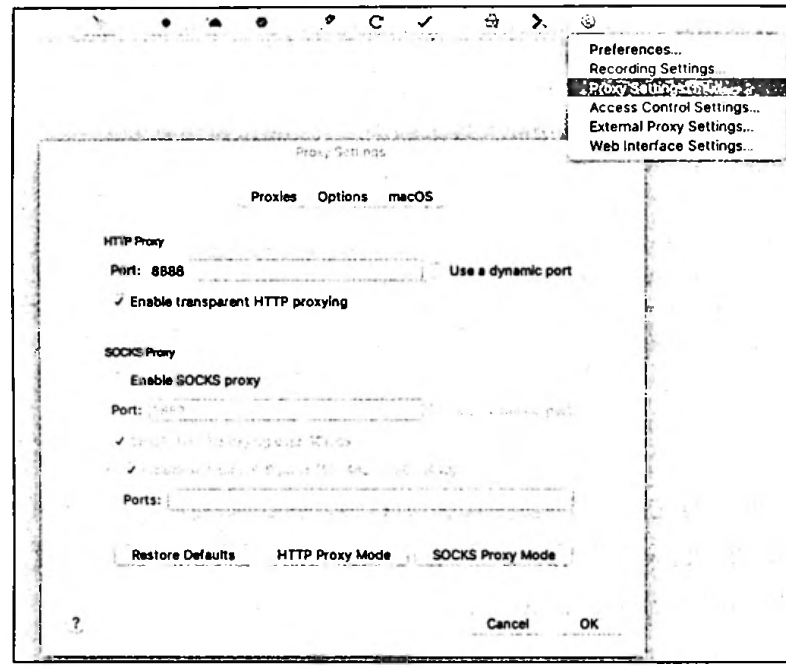


图 3-18 Charles 代理端口设置

在 iPhone 设备上，需要把网络代理设置为 Charles 启动的代理服务器。

单击“Charles”菜单下的“Help”→“Local IP Address”选项，获取本机 IP 地址，如图 3-19 所示。打开手机的 Wi-Fi 设置界面，点击当前连接的无线网络，在界面底部有 HTTP 代理的设置项。选择“手动”选项卡，在“服务器”栏填写获取的计算机 IP 地址，在“端口”栏填写 Charles 设置的端口 8888，如图 3-20 所示。设置完成后，当有网络通信发起时，Charles 就会弹出 iPhone 设备请求连接的窗口。单击“Allow”按钮，就能看到 iPhone 设备发出的网络请求了，包括请求、响应头、请求时间等。

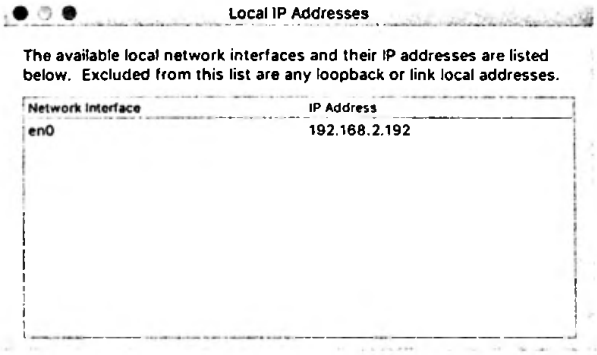


图 3-19　用 Charles 获取本机 IP 地址

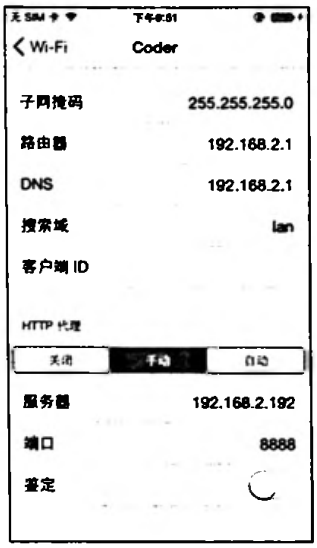


图 3-20　为 iPhone 设备设置代理服务器和端口

3.5.2　修改网络请求

如果想修改请求的参数却不想修改代码，或者想在不修改服务器代码的前提下修改服务器返回的内容，就可以使用 Charles 提供的数据重写功能。它不仅可以修改请求的数据，还能修改返回的响应消息，并支持正则匹配替换。Charles 提供了如下 3 种修改方式。

- Breakpoints：临时修改。
- Map：将本地请求重定向到一个文件或者将远程请求重定向到另一个请求。
- Rewrite：对网络请求数据进行正则匹配修改。

为了讲解方便，笔者使用了一个自己编写的 Demo，loginValidate.js 为服务器的代码。使用如下命令启动服务器（需要 node 环境）。

```
node loginValidate.js
```

打开 UserLogin 工程，修改 ULLoginManager.h 文件中的 URL_BASE 为服务器的地址，运行程序，得到一个简单的登录界面，如图 3-21 所示。输入正确的用户名“admin”和密码“123456”，会提示登录成功，否则提示登录失败。接下来，笔者就使用 Charles 对该例子里面的请求进行拦截和修改。

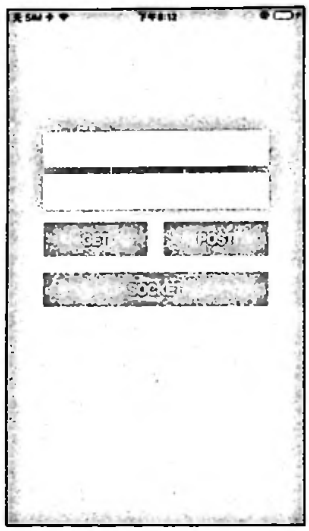


图 3-21 简单的登录请求 Demo 界面

1. Breakpoints

打开 Charles，单击请求列表中要拦截的请求，在相应的右键快捷菜单中选择“Breakpoints”选项，如图 3-22 所示。

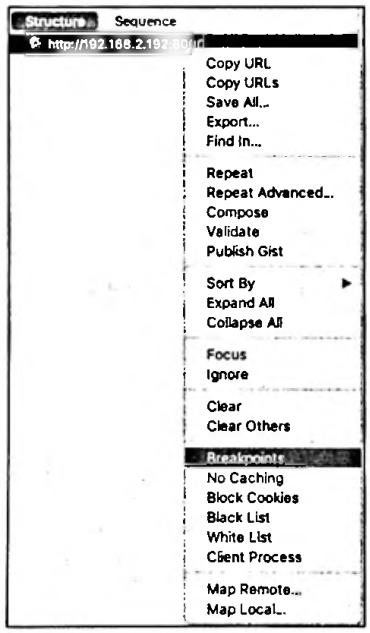


图 3-22 对指定的请求设置断点

单击 UserLogin 中的“POST”按钮，程序会在 Charles 里面中断。单击“Edit Request”标签，选择“Form”选项，如图 3-23 所示。在这里可以修改 POST 请求发送的请求参数。如果在这里将错误的用户名和密码改成正确的，然后单击“Execute”按钮，就能发起修改后的登录请求了。还可以单击“Add”和“Remove”按钮来增加和移除参数。

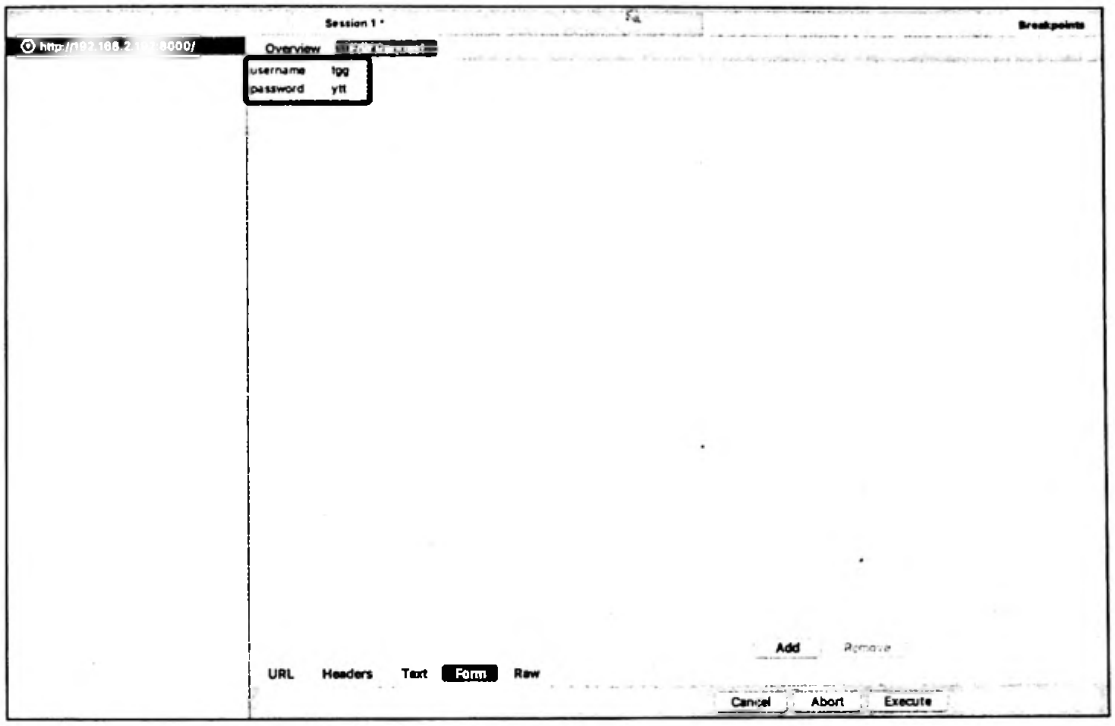


图 3-23　修改网络请求参数

2. Map

通过 Map 功能可以将请求的资源或者发向服务的查询重定向到本地文件。该功能可用于线上调试 bug 及 mock 服务器数据。在 UserLogin 中成功登录后，会返回如下的 json，可以通过 Map Local 把请求 map 到本地的文件。

```
{
        "username": "admin",
        "status": "success"
}
```

单击目标请求，然后选择“Map Local”选项，如图 3-24 所示。在弹出的对话框中选择需要映射的本地文件路径，如图 3-25 所示。设置完成后，填写任何用户名和密码并登录，都会提示登录成功。

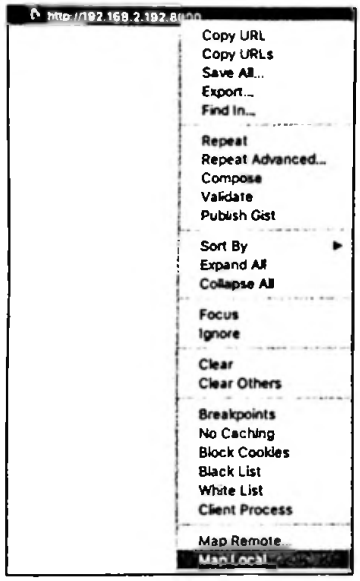


图 3-24　设置请求的 Map Local（1）

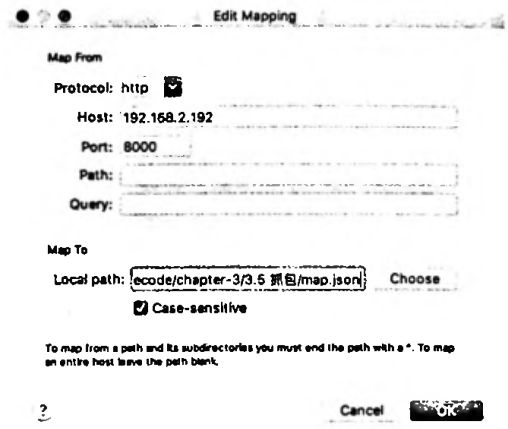


图 3-25　设置请求的 Map Local（2）

3. Rewrite

通过 Rewrite 可以进行正则匹配，替换请求路径、请求参数、请求头、请求 body、响应消息等。单击“Tools”→“Rewrite”选项，打开 Rewrite 的设置面板，如图 3-26 所示。在设置面板上方是规则的名称，接下来是匹配的请求 URL 规则，下方是对请求消息或者响应消息的替换规则。如图 3-26 所示的规则是把所有 online 的请求转向本地，可用于本地接口测试。

打开规则面板，可以看到更加详细的设置选项，如图 3-27 所示。

也可以在“Add Header”选项中增加请求验证需要的 Cookie，这样就可以跳过登录，模拟请求测试了。

除此之外，Charles 还有模拟网络环境、请求重发、反向代理等功能，这些功能就留给读者自己去探索了。

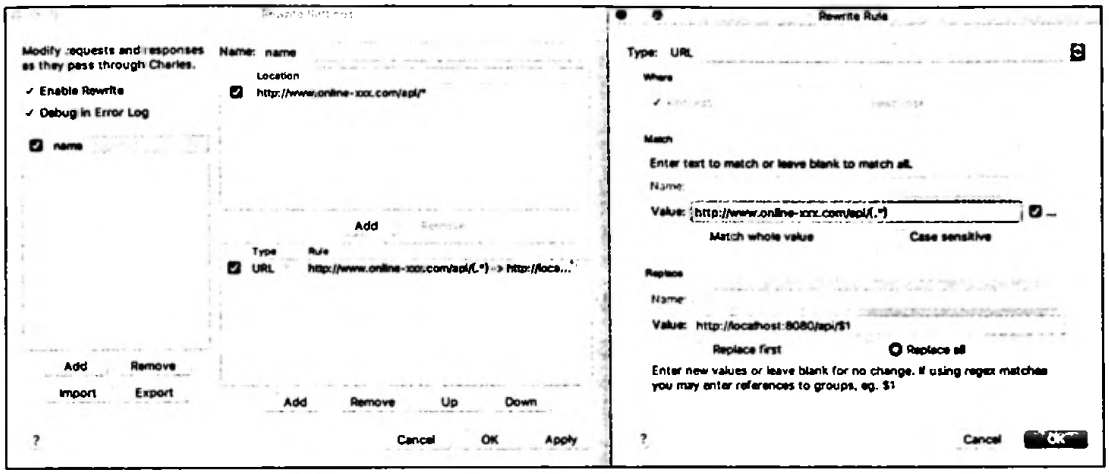


图 3-26　Rewrite 设置面板

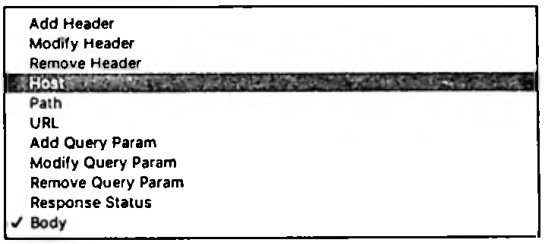


图 3-27　Rewrite 支持的选项

3.5.3　HTTPS 抓包

前面介绍的方法不能抓取 HTTPS 的网络请求。要抓取 HTTPS 的网络包，必须安装 Charles 的 CA 证书。如果要抓取 Mac 上的 HTTPS 请求，则要在 Mac 上安装证书。

单击“Help”→“SSL Proxying”→“Install Charles Root Certificate”选项，在钥匙串访问应用里面就能看到安装的证书 Charles Proxy Custom Root Certificate 了，如图 3-28 所示。右键单击安装的证书，在弹出的快捷菜单中选择“显示简介”选项，然后单击“信任”选项，更改“使用此证书时”为“始终信任”，如图 3-29 所示。最后，在想要监控的域名上单击右键，在弹出的快捷菜单中选择“Enable SSL Proxying”选项，就可以监控 HTTPS 的请求数据了。

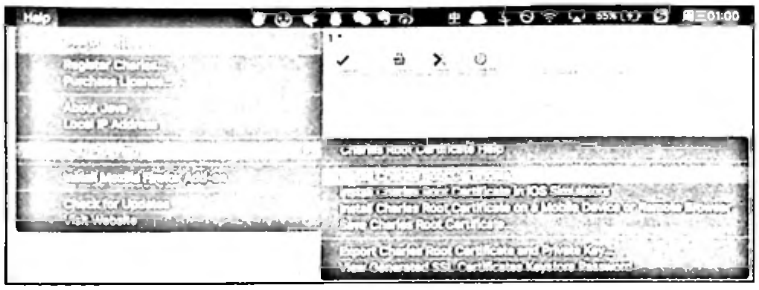


图 3-28 安装 Charles CA 证书

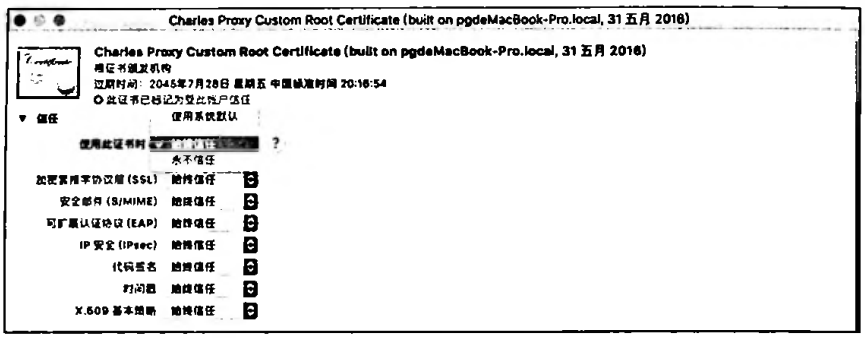


图 3-29 修改证书为始终信任

如果要监控移动设备的 HTTPS 请求，也需要在设备上面安装相应的证书。单击“Help”→“SSL Proxying”→“Install Charles Root Certificate on a Mobile Device or Remote Browse”选项，Charles 会弹出一个安装地址，如图 3-30 所示。用手机访问这个地址并安装证书，即可抓取移动设备的 HTTPS 请求。

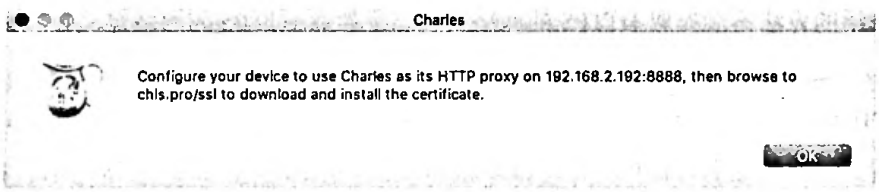


图 3-30 在真机上安装 Charles 证书

3.5.4 Wireshark 抓包

Wireshark（https://www.wireshark.org/）是一款比 Charles 更老牌也更强大的抓包工具，不仅可以抓取 HTTP 包，还可以抓取 TCP 和 UDP 包。Wireshark 的原理也和 Charles 不一样。Wireshark 将网卡设置为混杂模式：在正常情况下，网卡会过滤目标地址而不是自己的数据包；将网卡设

置为混杂模式，就会收到经过网卡的所有数据包。

本节仍使用前面的 Demo，修改 UserLogin 里面的 SOCKET_IP，使其和 server.php 运行的服务器 IP 地址一致。server.php 的监听地址为本机 IP 地址。在终端运行 `php server.php` 命令，启动服务器，输入用户名和密码，就能在终端看到服务收到的消息了，具体如下。

```
~ php server.php
client:123:456
```

本节主要以抓取 iPhone 设备上的网络请求为例来讲解 Wireshark 的使用。在官网下载并安装 Wireshark，然后将 iPhone 设备和计算机通过 USB 连接。打开 Xcode，单击菜单栏的“Window”→“Devices and Simulators”选项，找到连接设备的 Identifier，如图 3-31 所示。打开终端，运行如下命令（后面是笔者手机的设备 ID，你需要将它更换成自己的设备 ID）。

```
rvictl -s 57ff6c79061c9ab765b6463a51c30dbdbf0e36ff
Starting device 57ff6c79061c9ab765b6463a51c30dbdbf0e36ff [SUCCEEDED] with interface rvi0
```

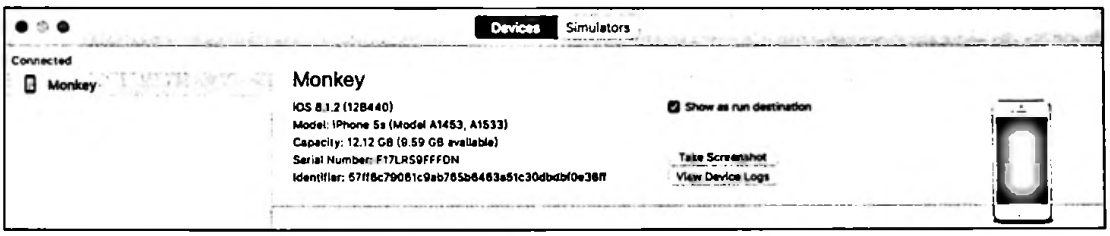


图 3-31　获取手机的设备 ID

命令执行成功后，就能在 Wireshark 中看到对应的虚拟网卡 rvi0 了。双击 rvi0，即可监控 iPhone 设备的所有流量，如图 3-32 所示。Wireshark 的主界面分为如下 4 个部分。

- 第 1 部分是工具栏菜单，可以控制开始抓包、停止抓包、重新抓包、包之间的跳转等，还可以在下面的输入框中输入自己的过滤器来查看指定条件的请求。
- 第 2 部分是所有收发数据包的列表，其中有源地址、目标地址、协议类型等信息，并且会用不同的颜色标注不同的包（“视图”→“着色规则”选项），以方便区分。
- 第 3 部分是单个数据包的详细信息。根据不同的协议类型会有不同的请求格式，通过对包信息的详细解读，可以帮助我们理解 TCP、HTTPS 握手传输的各个环节。
- 第 4 部分是数据包中原始的二进制数据。在第 3 部分选择不同的结构，下面对应的二进制数据便会高亮显示，以便我们更好地进行分析。

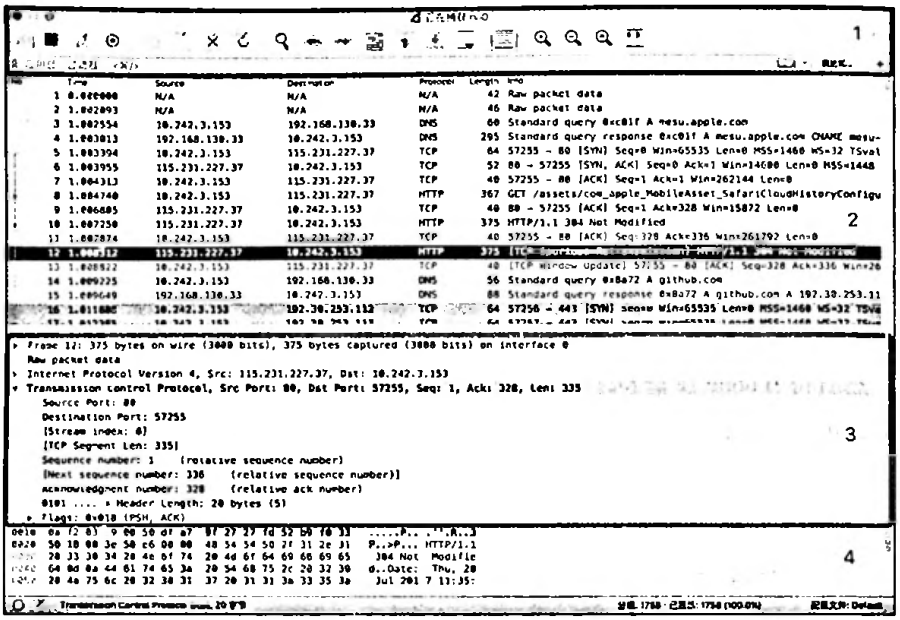


图 3-32 监控 iPhone 设备的网络流量

接下来，使用 Wireshark 观察 UserLogin 发出的数据包。为了避免过多的数据包干扰，可以在过滤输入框中输入“ip.addr == 192.168.2.192”（这个 IP 地址是笔者本机的 IP 地址）。然后，输入任意用户名和密码登录，便可以在 Wireshark 中看到如图 3-33 所示的数据包列表了。如果要查看整个 TCP 连接的发包和收包，可以选中某个数据包，在其右键快捷菜单中选择“追踪流”→“TCP 流”选项。

针对抓到的网络包，简单分析如下。

- No.1：建立连接时，客户端发送 SYN 包（SYN = x）到服务器，并进入 SYN_SEND 状态，等待服务器确认。
- No.2：服务器收到 SYN 包时，必须确认客户端的 SYN（ACK = x+1），同时自己发送一个 SYN 包（SYN = y），即 SYN+ACK 包，此时服务器进入 SYN_RECV 状态。
- No.3 ~ No.4：客户端收到服务器 B 的 SYN+ACK 包时，向服务器发送确认包 ACK（ACK = y+1），此包发送完毕，客户端和服务器进入 ESTABLISHED 状态，完成三次握手。
- No.5 ~ No.8：客户端和服务器之间的数据传输和确认。单击 PSH 类型的数据，就能在下方看到传输的实际数据。
- No.9 ~ No.10：表示传输结束。

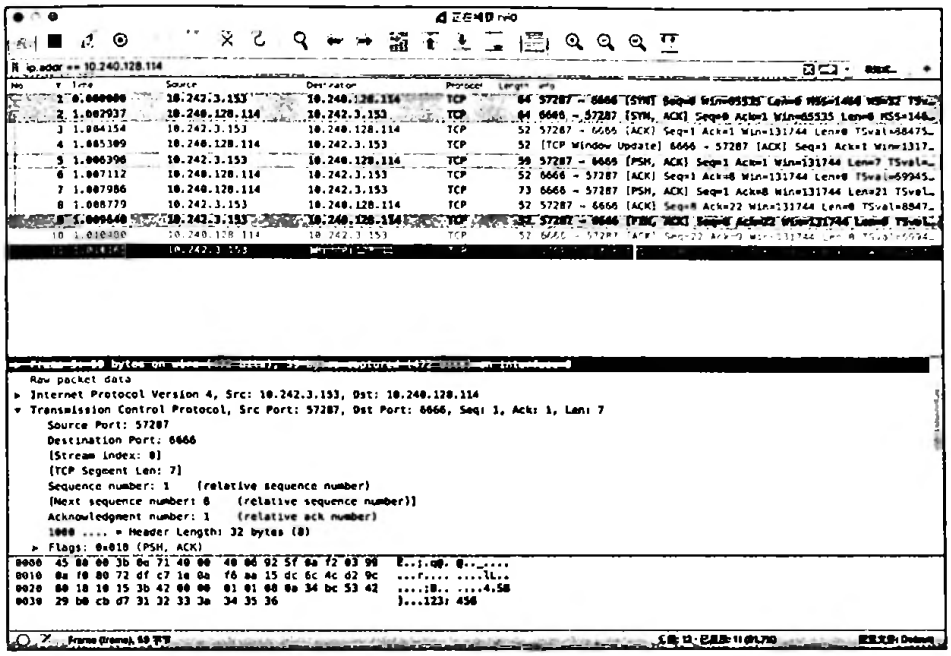


图 3-33　用 Wireshark 抓取 UserLogin 发送的数据包

在 PSH 传输中，数据包分为 3 个部分，即 IP 头、TCP 头、传输数据，其结构可以通过详细窗口看到，如图 3-34 所示。

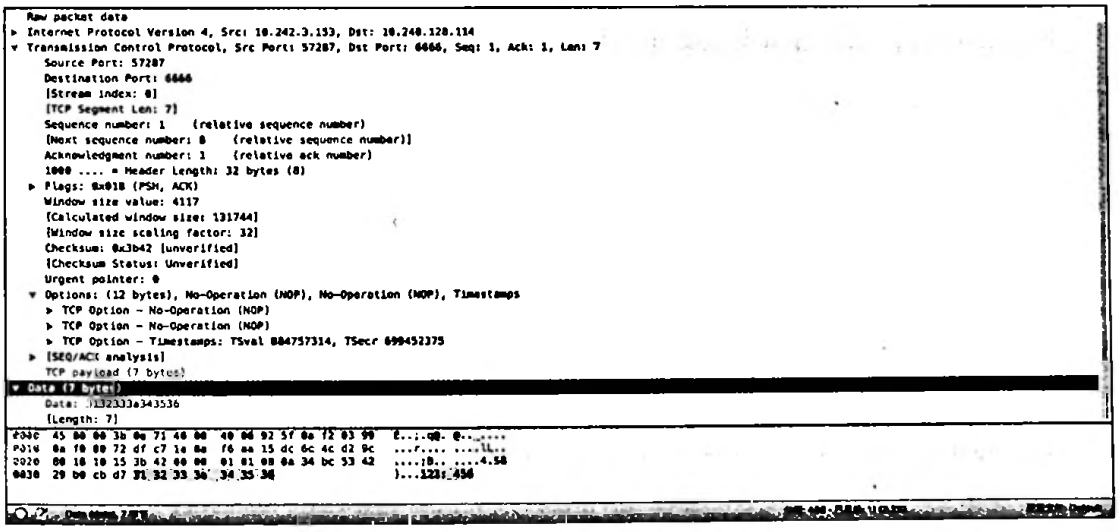


图 3-34　TCP 数据包的传输结构

IPv4 Header 和 TCP Header 所对应的结构分别如图 3-35（图片源于 https://en.wikipedia.org/wiki/IPv4#Header）和图 3-36 所示（图片源于 https://en.wikipedia.org/wiki/Transmission_Control_Protocol）。当然，在 Wireshark 里面展开 Header 也能看到二进制值和对应的含义。

Offsets	Octet	0								1								2								3			
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Version				IHL				DSCP						ECN		Total Length															
4	32	Identification																Flags				Fragment Offset											
8	64	Time To Live								Protocol								Header Checksum															
12	96	Source IP Address																															
16	128	Destination IP Address																															
20	160	Options (if IHL > 5)																															
24	192																																
28	224																																
32	256																																

图 3-35 IPv4 Header

Offsets	Octet	0								1								2								3			
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number (if ACK set)																															
12	96	Data offset				Reserved 0 0 0				N S	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N	Window Size														
16	128	Checksum																	Urgent pointer (if URG set)														
20	160	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.)																															
...	...	...																															

图 3-36 TCP Header

除了上文所说的使用 ip.addr 过滤 IP 地址，还有很多过滤条件可以设置，举例如下。

```
tcp.port == 6666
#TCP 端口为 6666 的数据包

ip.src == 10.242.3.153
#IP 源地址为 10.242.3.153 的数据包
```

如果记不住这些表达式也没关系，单击过滤框右侧的“表达式”按钮，就能直接通过界面来选择需要过滤的条件和值了，如图 3-37 所示。

此外，可以通过正则、比较、截取等操作进行过滤，具体方法请参考官方说明（https://www.wireshark.org/docs/wsug_html_chunked/ChWorkBuildDisplayFilterSection.html）。

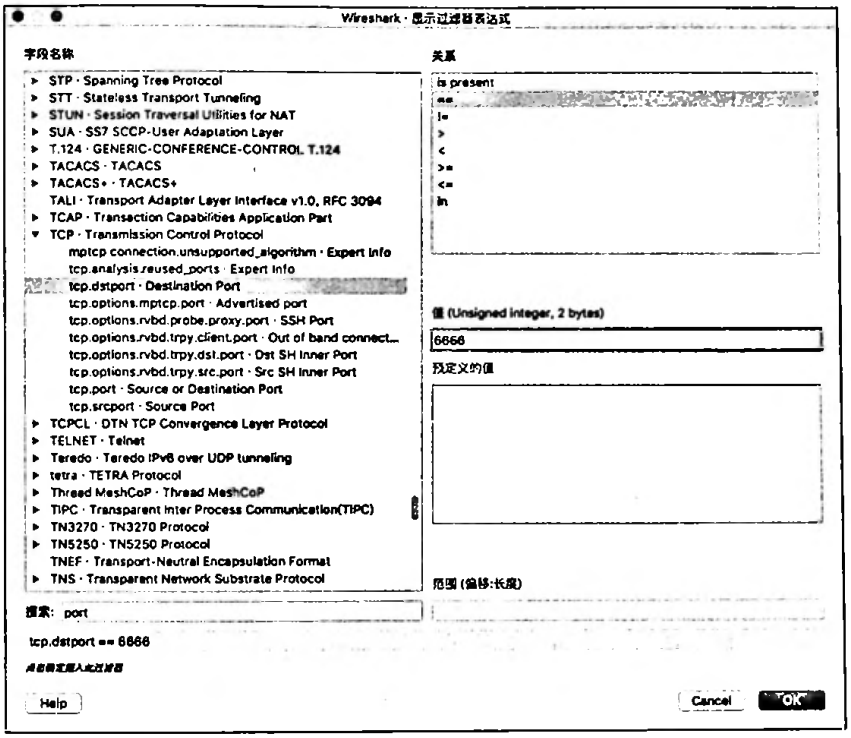


图 3-37　在 Wireshark 中选择过滤条件

第 4 章　开发储备

通过第 3 章介绍的一些常用的逆向工具，相信读者对逆向工程的认识更加全面了一些。在本章中，笔者将介绍在逆向中经常被忽略的一些理论基础。不管是逆向工具还是逆向分析技巧，其实都离不开理论基础。只有理论储备扎实，才能找到逆向分析的方法和思路，也才能在遇到问题时有办法解决，并知道可能引发问题的原因。

4.1　App 的结构及构建

在 iOS 逆向分析中，需要知道 iOS 应用的安装包如何获取、安装时是什么形式、分析的目标文件是什么及编译生成一个 app 包的细节。在本节中，将从获取应用包、分析包结构及生成应用包 3 个方面进行讲解。

4.1.1　获取应用包

获取应用包有很多种方式，通过不同方式获取的应用包可能不太一样。应用包有 .app 文件夹的，也有 .ipa 文件的，有已经解密的，也有没有解密的。下面介绍应用包的几种获取方式。

1. 从 iTunes 获取 ipa 包

iTunes 升级到 12.7.0.166 之后移除了 App Store 的功能。苹果官方提供了 iTunes 12.6.3 支持的 App Store 和 iOS 11，下载地址为 http://secure-appldnld.apple.com/itunes12/091-33628-20170922-EF8F0FE4-9FEF-11E7-B113-91CF9A97A551/iTunes12.6.3.dmg。下载后，按住“Option”键双击打开 iTunes，单击创建资料库，否则会提示 iTunes Library.itl 由高版本创建，资料库无法打开。

打开 iTunes 之后，单击“App Store”标签，在右上角的搜索框中输入想要下载的应用名称，如图 4-1 所示。

图 4-1　使用 iTunes 搜索应用

选择“建议”中的第 1 个应用，单击搜索列表中对应应用的“获取”或者“下载”按钮，即可下载该应用最新版本的 ipa 文件，如图 4-2 所示。

图 4-2　使用 iTunes 下载指定应用

下载之后，在“资料库”中找到刚刚下载的应用，单击右键，在弹出的快捷菜单中选择“在 Finder 中显示”选项，如图 4-3 所示，就会定位 Finder 所对应的 ipa 文件的位置了（这里下载的是 Snapchat 10.15.1.ipa）。

图 4-3 在 iTunes 中找到下载应用所在的文件夹

通过这种方式只能下载最新版本的应用。下面介绍一种结合 Charles 下载早期版本应用的方法。

打开 Charles，设置 Mac OS 全局代理，然后重新获取应用，即可在 Charles 中看到类似“https://p48-buy.itunes.apple.com”的请求。对该请求下断点，然后删除资源库中已经下载的应用，重新单击获取应用的按钮，就会触发断点。在 Request 发出时，单击“Execute”按钮继续发送请求。Response 会再次触发该断点。单击“Edit Response”选项，选择“XML Text”选项，可以看到类似如下的内容。

```
<key>softwareVersionBundleId</key><string>com.toyopagroup.picaboo</string>
  <key>softwareVersionExternalIdentifier</key><integer>823313080</integer>
  <key>softwareVersionExternalIdentifiers</key>
  <array>
    <integer>3880712</integer>
    <integer>3933851</integer>
    ...
    <integer>820860187</integer>
    <integer>820957756</integer>
    <integer>821184323</integer>
    <integer>821243477</integer>
    <integer>821295499</integer>
    <integer>821339677</integer>
    <integer>821373203</integer>
    ...
    <integer>823218391</integer>
    <integer>823313080</integer>
 </array>
```

- softwareVersionBundleId：当前下载应用的 Bundle ID。
- softwareVersionExternalIdentifier：其中的“823313080”和下面 array 中最新的数字对应，表示当前下载的版本。
- softwareVersionExternalIdentifiers：包含该应用所有版本所对应的数字。为了下载早期版本，需要获取这里对应的数字。

假如要下载版本为 820957756 的 ipa 文件，可以再次单击获取应用的按钮，触发断点，然后单击“Edit Request”选项，得到如下内容。

```
<?xml version="1.0" encoding="UTF-8"?>
<plist version="1.0">
<dict>
        <key>appExtVrsId</key>
        <string>823313080</string>
        <key>guid</key>
        <string>24A074F1AB2E</string>
        <key>kbsync</key>
        <data>
......
```

这里的“823313080”表示当前要下载应用的版本。将这个值修改为“820957756”，然后单击“Execute”按钮，等待下载完成。在 Finder 中显示得到了 Snapchat 10.3.0.ipa。那么，如何将这些数字和版本号对应起来呢？在 iTunes 的搜索结果中单击目标应用下面的下拉菜单，选择“复制链接”选项，如图 4-4 所示。

图 4-4　在 iTunes 中找到目标应用的链接

例如，Snapchat 复制的链接是 https://itunes.apple.com/cn/app/%E5%BF%AB%E6%8B%8D-snapchat/id447188370?mt=8。提取其中的 ID “447188370”，然后访问 https://api.unlimapps.com/v1/

apple_apps/447188370/versions。将“447188370”替换成我们复制的版本 ID，得到如下访问结果，在其中找到对应版本号的 external_identifier。否则，就只能使用上面 array 中的版本数字去尝试寻找我们想要的版本了。

```
......
{
    "id": "a1156328-03f3-4e3a-8038-08c82465b53d",
    "apple_app_id": "fed1aa69-16ea-4016-8bab-b0f41812ac05",
    "bundle_version": "10.3.0",
    "external_identifier": "820957756",
    "created_at": "2017-02-28T17:20:07.551Z",
    "updated_at": "2017-08-26T00:38:27.810Z"
},
......
```

2. 直接从越狱设备获取应用

在 iOS 8 及之前的版本中，在 iTools 中单击“应用”→“备份”选项，直接打包 ipa 文件，如图 4-5 所示。

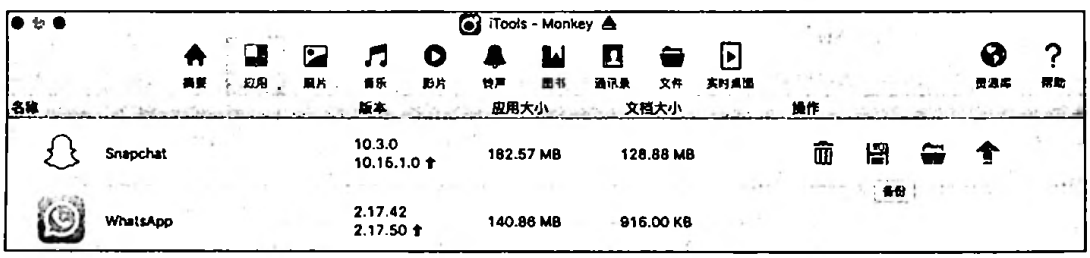


图 4-5 使用 iTools 备份应用

在 iOS 9 之后就不能通过这种方式导出了。不过，我们还是可以直接在设备中使用 scp 命令复制整个 App 文件夹，具体如下。

```
➜  ~ ssh 5s
Monkey:~ root# ps aux | grep Snapchat
mobile    5842   0.0  6.1   804092  62300   ??  Us    6:50PM   0:05.19
/var/mobile/Containers/Bundle/Application/4E322288-4AFB-4BF8-81BE-33D6081CB25C/Snapchat
.app/Snapchat
root      5981   0.0  0.0   536256    444 s000  R+   10:21PM   0:00.01 grep Snapchat

➜  iosreversebook git:(master) ✗ scp -P 2222 -r
root@localhost:/var/mobile/Containers/Bundle/Application/4E322288-4AFB-4BF8-81BE-33D608
```

```
1CB25C/Snapchat.app/ ./
Alternate-Got-No3D.otf                        100%   68KB   3.5MB/s   00:00
AppIcon20x20@2x.png                           100%  837   111.3KB/s   00:00
AppIcon20x20@3x.png                           100% 1393   310.1KB/s   00:00
AppIcon29x29@2x.png                           100% 1289   305.0KB/s   00:00
AppIcon29x29@3x.png                           100% 2091   387.3KB/s   00:00
AppIcon40x40@2x.png                           100% 1876   351.0KB/s   00:00
......
Snapchat                                      100%   74MB   4.8MB/s   00:15
......
```

上面讲到了两种获取应用包的方法，即从 iTunes 下载和从手机导出，其中有 ipa 格式的应用包，也有 app 格式的应用包。从 iTunes 下载的应用包是未解密的。此外，可以使用一些助手工具下载越狱应用，这样下载的应用是解密的。

4.1.2　应用包的格式

应用包中存在两种格式，即 ipa 和 app，可以使用 file 命令分别查看，具体如下。

```
➜  files git:(master) X file Snapchat\ 10.15.1.ipa
Snapchat 10.15.1.ipa: Zip archive data, at least v2.0 to extract
➜  files git:(master) X file Snapchat.app
Snapchat.app: directory
```

ipa 文件本身就是 zip 压缩包。单击右键，在弹出的快捷菜单中选择打开方式“归档实用工具”，即可将里面的内容解压。解压后的目录结构如下（用 brew install tree 安装 tree 工具）。

```
➜  files git:(master) X tree Snapchat\ 10.15.1 -L 2
Snapchat\ 10.15.1
├── META-INF
│   ├── com.apple.FixedZipMetadata.bin
│   └── com.apple.ZipMetadata.plist
├── Payload
│   └── Snapchat.app
├── iTunesArtwork
└── iTunesMetadata.plist
```

Payload 目录下面就是 Snapchat.app 文件夹。由于 Snapchat.app 文件夹中的内容太多，就不在此一一列举了，主要包括如下几类文件。

- Info.plist 文件：存储应用的相关设置、Bundle identifier 和 Executable file 可执行文件名。
- 可执行文件：Info.plist 中 Executable file 记录的名字所对应的文件。该文件主要用于分析。

- Frameworks：当前应用使用的第三方 Framework 或 Swift 动态库（据说苹果以后会把这些 Swift 动态库去掉）。
- PlugIns & Watch：当前应用使用的 Extension，以及和 Watch 手表一起使用的应用。
- 资源：其他文件，包括图片资源、配置文件、视频/音频资源，以及一些与本地化相关的文件。

4.1.3 应用的构建过程

了解了应用包的目录结构组成之后，我们来看看 Xcode 是如何编译，并最终生成一个应用程序的。

新建一个 Xcode iOS App 项目，按“Command + B”快捷键编译项目，单击查看编译细节，如图 4-6 所示，过程如下。

①编译源文件：使用 Clang 编译项目中所有参与编译的源文件，生成目标文件。

②链接目标文件：将源文件编译生成的目标文件链接成一个可执行文件。

③复制编译资源文件：复制和编译项目中使用的资源文件。例如，将 storyboard 文件编译成 storyboardc 文件。

④复制 embedded.mobileprovision：将本地 /Users/monkey/Library/MobileDevice/Provisioning\Profiles/ 下的描述文件复制到生成的 App 目录下面。

⑤生成 Entitlements：生成签名用的 Entitlements 文件。

⑥签名：使用生成的 Entitlements 文件对生成的 App 进行签名。

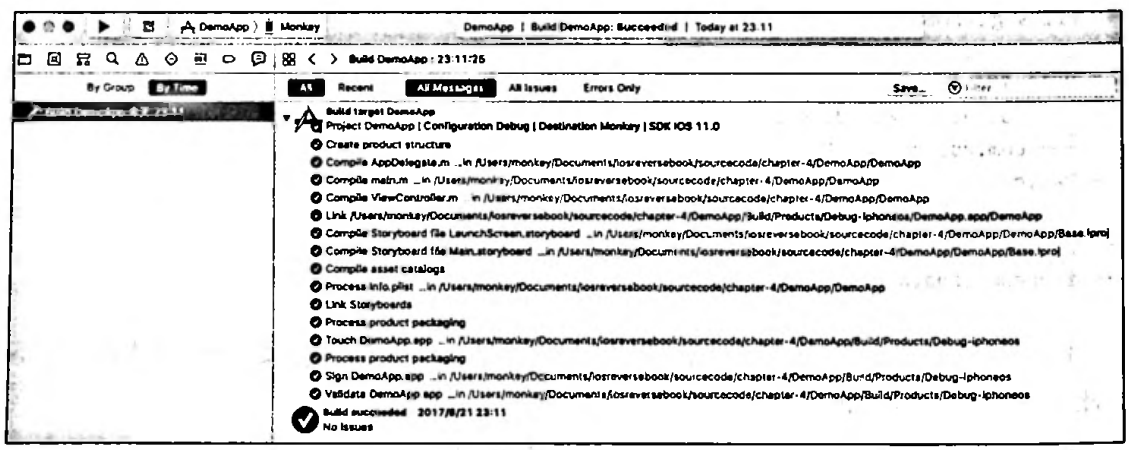


图 4-6 使用 Xcode 查看 App 的编译细节

为了帮助读者理解编译过程中的每个步骤，下面通过命令演示如何生成一个 ipa 包。从随书源代码中获取相关内容，打开 makefile 文件，具体如下。

```
makefile:

CurrentDir = "$(shell pwd)"
ResourceDirecrory = AppSource
AppName = DemoApp
TmpBuildFile = $(AppName).app
ConstIBFile = Base.lproj
Architecture = arm64
CertificateName = "iPhone Developer: peiqing liu (xxxxxxxxxx)"

compile:
        #0. 创建 BuildDemo.app 文件
        @rm -r -f $(TmpBuildFile)
        @test -d $(TmpBuildFile) || mkdir $(TmpBuildFile)

        #1. 编译 Objective-C 文件
        @#如果不用 xcrun，而直接用 Clang，需要用-isysroot 指定系统 SDK 路径，例如
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/i
PhoneOS.sdk

        @xcrun -sdk iphoneos clang \
                -arch $(Architecture) \
                -mios-version-min=8.0 \
                -fmodules \
                -fobjc-arc \
                -c $(ResourceDirecrory)/AppDelegate.m \
                -o $(TmpBuildFile)/AppDelegate.o

        @xcrun -sdk iphoneos clang -arch $(Architecture) -mios-version-min=8.0 -fmodules
-fobjc-arc -c $(ResourceDirecrory)/main.m -o $(TmpBuildFile)/main.o
        @xcrun -sdk iphoneos clang -arch $(Architecture) -mios-version-min=8.0 -fmodules
-fobjc-arc -c $(ResourceDirecrory)/ViewController.m -o $(TmpBuildFile)/ViewController.o

link:
        #2. 链接目标文件
        @xcrun -sdk iphoneos clang \
                $(TmpBuildFile)/main.o $(TmpBuildFile)/AppDelegate.o
$(TmpBuildFile)/ViewController.o \
                -arch $(Architecture) \
                -mios-version-min=8.0 \
                -fobjc-arc \
```

```
                -fmodules \
                -o $(TmpBuildFile)/$(AppName)

        @rm $(TmpBuildFile)/AppDelegate.o $(TmpBuildFile)/main.o
$(TmpBuildFile)/ViewController.o

storyboard:
        #3. 编译 storyboard 文件
        @mkdir $(TmpBuildFile)/$(ConstIBFile)

        @ibtool \
                --compilation-directory \
                $(TmpBuildFile)/$(ConstIBFile) \
                $(ResourceDirecrory)/$(ConstIBFile)/Main.storyboard

        @ibtool --compilation-directory $(TmpBuildFile)/$(ConstIBFile)
$(ResourceDirecrory)/$(ConstIBFile)/LaunchScreen.storyboard

plist:
        #4.生成 plist 文件，其中指定了 App ID、name、version 等

        @defaults write \
                $(CurrentDir)/$(TmpBuildFile)/Info \
                CFBundleDevelopmentRegion en #国际化时优先使用的语言

        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundleExecutable $(AppName)
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundleIdentifier
com.alonemonkey.$(AppName)
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundleInfoDictionaryVersion
6.0 #plist 文件结构的版本
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundleName $(AppName)
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundlePackageType APPL #APPL:
app, FMWK: frameworks, BND: loadable bundles
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundleShortVersionString 1.0
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info CFBundleVersion 1
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info LSRequiresIPhoneOS YES
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info UIMainStoryboardFile Main
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info UILaunchStoryboardName
LaunchScreen
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info MinimumOSVersion 8.0
        @defaults write $(CurrentDir)/$(TmpBuildFile)/Info DTPlatformName iphoneos

asset:
        #5.复制图片资源
```

```
        @cp -a $(ResourceDirecrory)/images/. $(TmpBuildFile)/

dsym:
        #6.生成 dSYM 文件
        @#使用 dwarfdump --uuid 可以查看 dSYM 或可执行文件的 UUID，匹配成功才能完全将 crash log
中的十六进制地址符号化
        @dsymutil \
                -arch $(Architecture) \
                $(TmpBuildFile)/$(AppName) \
                -o $(AppName).app.dSYM

codesign:
        #7.签名
        @#mobileprovision 文件包含 Team ID 和允许安装设备的 ID
        @cp -f embedded.mobileprovision $(TmpBuildFile)
        @#provision 查看命令：security cms -D -i provision_file

        @codesign \
                -fs \
                $(CertificateName) \
                --entitlements entitlements.plist \
                $(TmpBuildFile)
        @#使用 codesign -vv xx.app 命令查看 App 签名信息

package:
        #8.打包 ipa
        @mkdir -p Payload
        @cp -r -f $(TmpBuildFile) Payload
        @zip -r -q $(AppName).ipa Payload
        @rm -f -r Payload/
        @rm -f -r $(TmpBuildFile)

all: compile link storyboard plist asset dsym codesign package
```

从 makefile 文件中可以看到，整个过程大致如下。读者可以单独在命令行下指定对应的 Target，通过 make compile 编译源文件。

①compile：使用 Clang 编译源文件。xcrun 会自动找到 Clang 的位置。-fmodules 参数会自动找到需要的系统库。-fobjc-arc 参数指定由 ARC 编译。

②link：将编译生成的目标文件链接成一个可执行文件。

③storyboard：编译项目中的 storyboard 文件。

④plist：生成 plist 文件，里面会指定应用的名字、Bundle ID 等。

⑤asset：将需要的资源文件复制到目标 App 目录下。

⑥dsym：生成符号文件。

⑦codesign：对 App 进行签名，需要 embedded.mobileprovision。可以新建一个应用，在编译生成的 App 目录下找到 embedded.mobileprovision。entitlements.plist 里面的内容可以通过展开“Process product packaging”项目找到，如图 4-6 所示（注意：这两个文件都需要换成你自己的）。

⑧package：打包。将生成的 App 文件夹放到 Payload 文件夹下，通过 zip 压缩成 ipa 文件。

最后，将生成的 ipa 文件复制到 iTools 的应用中，就可以将 App 安装到手机上并运行了。

这里的演示只是为了让读者了解一个 App 的整个生成过程，真正通过 Xcode 编译生成 App 的过程会涉及其他的步骤和更多的编译参数，但这并不影响读者自己通过源代码和资源编译生成一个 App。

4.2 界面结构和事件传递

对没有接触过 iOS 正向开发的人来说，开始分析一个应用时可能会不知从哪里入手。如何根据界面的组成去寻找界面背后真正的代码？在本节中主要介绍 iOS 界面的组成、事件的传递及响应过程。

4.2.1 界面的组成

无论是 Windows 桌面应用开发、Android 开发还是 iOS 开发，都是按系统提供的大框架来进行的。开发者在框架中创建自己的界面，实现界面跳转，编写业务逻辑。所以，在分析一个应用时，首先需要了解一个正常的 iOS 应用是怎么开发的，里面有哪些组件，以及它们是如何组成界面的。下面通过一个简单的 iOS App 来分析应用的层次结构，所使用的源代码可以在随书源代码中找到。

使用 Xcode 新建一个简单的 iOS App 工程。打开 Main.storyboard 文件，向控件窗口中拖入一个 UIView。选中该控件并按住“Control”键，将其与 ViewController 的一个属性关联起来，如图 4-7 所示。

在 ViewController.m 文件的 viewDidLoad 方法中写入如下代码，获取运行时 view 的内存地址。

```
- (void)viewDidLoad {
```

```
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    NSLog(@"bgView address: %p", _bgView);
}
```

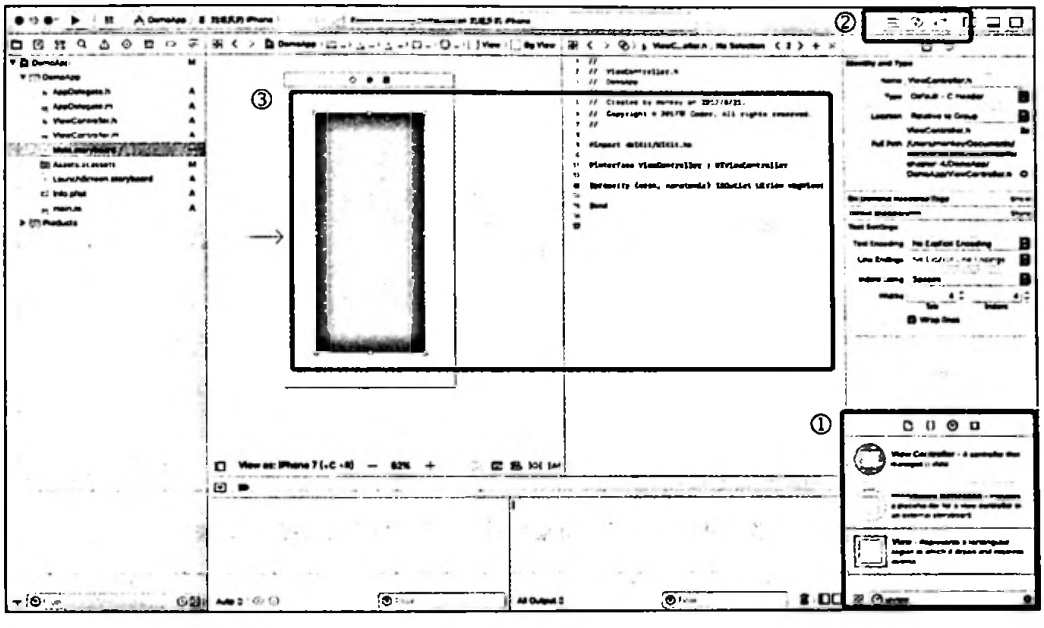


图 4-7　使用 Xcode 向 storyboard 中添加一个 UIView 控件

单击“Run”按钮，使 Demo 程序在手机上运行。单击调试工具栏的“Debug Memory Graph”按钮，可以得到如图 4-8 所示的内存引用关系图。在图 4-8 中还可以看到，bgView 在内存中的地址为 0x100308850。

复制该地址，将其粘贴到界面左下角的搜索框中，在内存中搜索该 UIView 的对象，就能够看到谁在引用该对象了，如图 4-9 所示。

根据如图 4-9 所示的内存引用关系图，可以得到这样的引用关系：顶层的 UIWindow 的 root ViewController 引用了 ViewController，ViewController 有自己的 view，该 view 中又包含 subview，而 bgView 就是它的 subview 之一。

当然这只是一个简单的例子，在复杂的例子中还有 UINavigation、ViewController 嵌套、UIView 嵌套等。在应用中，可以通过 `[UIApplication sharedApplication].keyWindow` 获取 UIWindow 对象。一个简单的 iOS 应用的层次如图 4-10 所示。

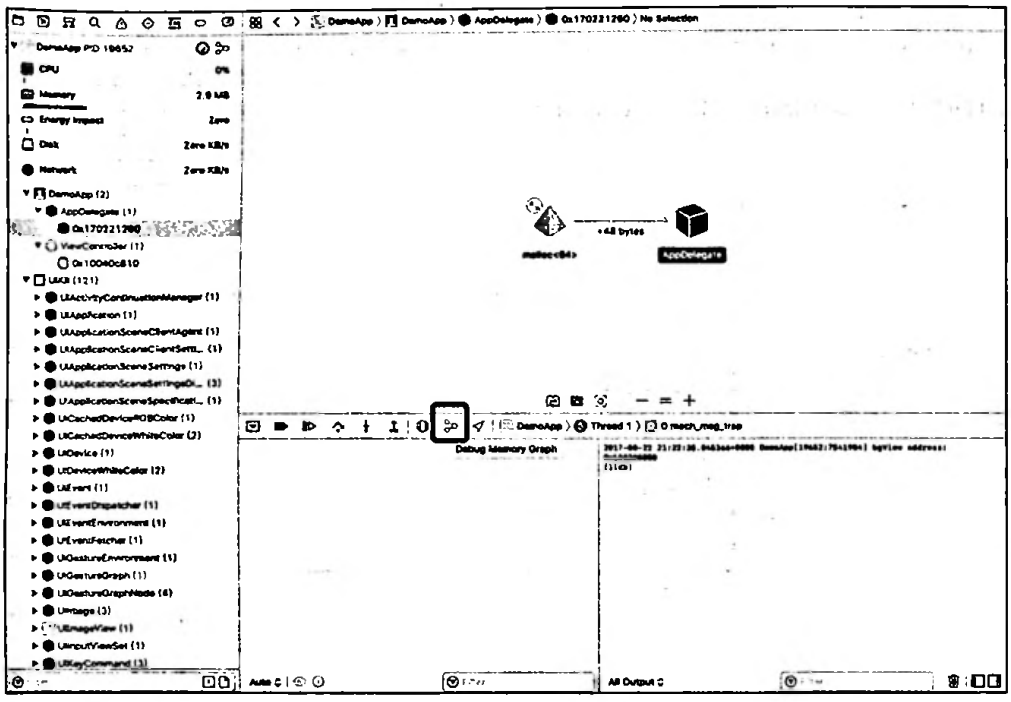


图 4-8 使用 Xcode 查看对象的内存引用关系图

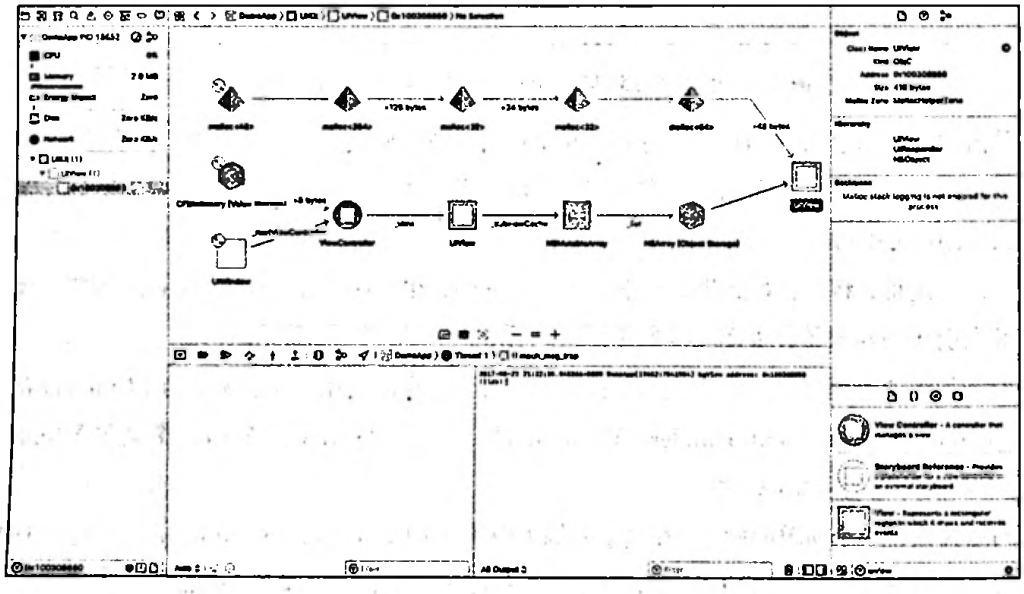


图 4-9 根据地址搜索内存中对应的对象

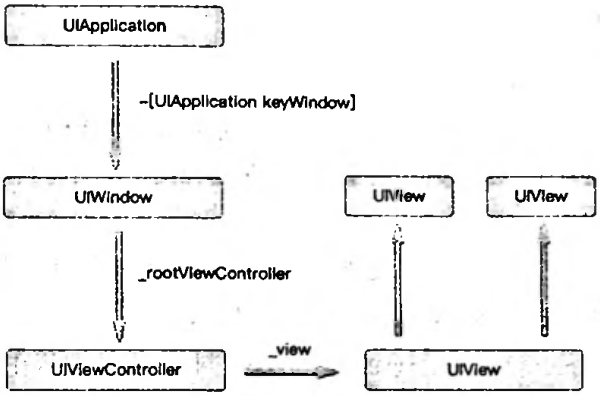


图 4-10　一个简单的 iOS 应用的层次结构

在逆向分析中，很多功能的实现都可以从 UIView 开始寻找线索，而和 UIView 相关的业务处理逻辑代码一般都会写在 UIViewController 里面，因此，需要通过 UIView 找到 UIViewController，进而找到对应的处理代码。在 4.2.2 节中将介绍如何通过 UIView 找到对应的 UIViewController。

4.2.2　界面事件的响应

iOS 中的事件有触摸事件、传感加速事件和远程控制事件，本书只讨论触摸事件。在学习触摸事件之前，需要了解一个叫作 UIResponder 的类，它是专门用来响应用户操作和处理各种事件的。UIApplication、UIView 和 UIViewController 都直接继承自 UIResponder，因此，UIWindow（继承 UIView）、自定义的 UIView 和继承 ViewController 的控制器都可以响应事件。在 iOS 中，通常将这些能够响应事件的对象称为响应者，如图 4-11 所示。

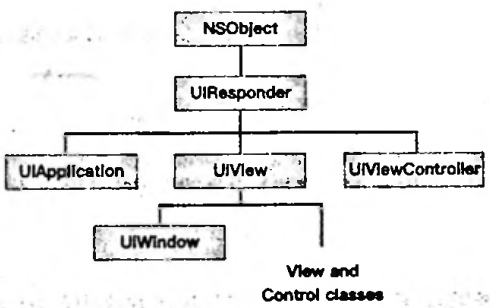


图 4-11　iOS 中继承 UIResponder 的类

当一个事件产生时，会触发寻找事件接受者和触发事件响应这两个步骤。

1. 寻找事件接受者

寻找事件接受者的流程如下。

①当一个触摸事件生成时，系统会将其加入 UIApplication 管理的事件队列中。

②UIApplication 会取出队列最前面的事件。在通常情况下，会将其先分发到应用程序的主窗口 keyWindow。

③主窗口会在当前视图层次结构中找到一个最合适的视图来处理触摸事件。

寻找最合适的视图的过程如图 4-12 所示，步骤如下。

①调用当前视图的 `pointInside:withEvent:` 方法，判断触摸点是否在当前视图内。

②如果返回 NO，那么 `hitTest:WithEvent:` 就返回 nil。如果返回 YES，就继续遍历子视图（subview），发送 `hitTest:withEvent:` 消息，直到有视图返回非空对象时返回该对象（或者在全局视图遍历完毕并都返回空对象时返回自身）。

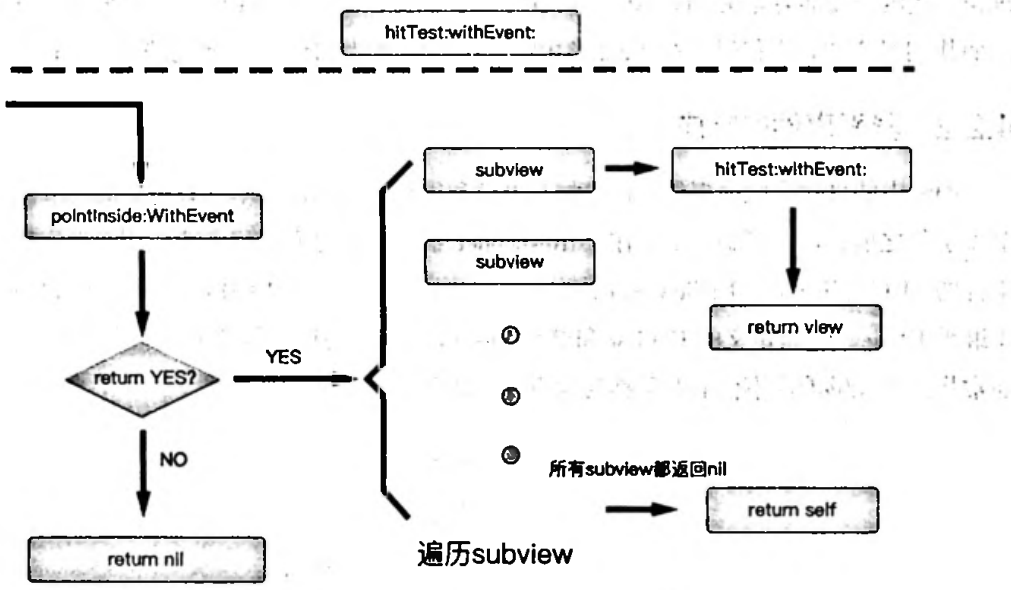


图 4-12 寻找最合适的视图的过程

2. 触发事件响应

触发事件将沿着响应者链往下传递。先看看苹果官方给出的图例，如图 4-13 所示，传递规则如下。

- 如果当前 view 是另一个 view 的子 view，那么它的父 view 就是下一个响应者。

- 如果当前 view 是控制器的 view，那么控制器就是下一个响应者。
- 如果在视图顶层还不能处理事件，那么就传给 Window 对象处理。
- 如果 Window 对象也不能处理，则将其传给 UIApplication 对象。
- 如果 UIApplication 对象也不能处理，就可能传给 UIAppDelegate 对象处理。

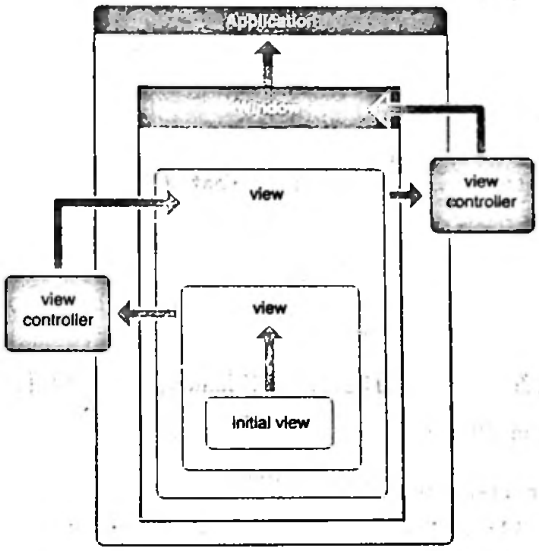


图 4-13　iOS 响应者链向下传递

在逆向过程中，经常需要根据某个控件通过 nextResponder 方法不断寻找它的下一个响应者。在 ViewController 中添加一个 button，然后给其绑定 Target-Action，测试一下。添加的代码如下。

```
#import "ViewController.h"

@interface ViewController ()

@property UIButton * btn;

@end

@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
```

```
    NSLog(@"bgView address: %p", _bgView);

    _btn = [[UIButton alloc] initWithFrame:CGRectMake(0, 0, 100, 100)];
    [_btn setBackgroundColor:[UIColor redColor]];
    [_btn addTarget:nil action:@selector(onPress)
forControlEvents:UIControlEventTouchUpInside];
    [self.view addSubview:_btn];
}


- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
    // Dispose of any resources that can be recreated.
}


@end
```

这里的 Target 故意设置为 nil。运行之后，单击 button，发现没有任何反应，这是正常的。可以通过如下方法打印 button 的响应链。

```
(lldb) po [0x111e0ca00 nextResponder]
<UIView: 0x111d0a460; frame = (0 0; 375 667); autoresize = W+H; layer = <CALayer: 0x17002f8c0>>

(lldb) po [[0x111e0ca00 nextResponder] nextResponder]
<ViewController: 0x111d08e10>

(lldb) po [[[0x111e0ca00 nextResponder] nextResponder] nextResponder]
<UIWindow: 0x111d09320; frame = (0 0; 375 667); autoresize = W+H; gestureRecognizers = <NSArray:
0x17004ad40>; layer = <UIWindowLayer: 0x17002f580>>

(lldb) po [[[[0x111e0ca00 nextResponder] nextResponder] nextResponder] nextResponder]
<UIApplication: 0x111d00920>

(lldb) po [[[[[0x111e0ca00 nextResponder] nextResponder] nextResponder] nextResponder]
nextResponder]
<AppDelegate: 0x17402e560>

(lldb) po [[[[[[0x111e0ca00 nextResponder] nextResponder] nextResponder] nextResponder]
nextResponder] nextResponder]
 nil
(lldb)
```

按照刚刚说的响应链的传递方式，在 UIApplication 上面实现这个方法，就能根据响应链的传递找到该方法了。在 UIApplication 中添加如下代码。

```
@implementation AppDelegate


- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions {
    // Override point for customization after application launch.
    return YES;
}

......

-(void)onPress{
    NSLog(@"UIApplegate onPress!!!");
}

@end
```

重新运行程序，再次单击 button，发现打印了“UIApplegate onPress!!!”（这也证实了上面的猜测）。但是，这种 Target 一般会设置成当前的 ViewController。将 Target 修改为 self 后，通过如下方法就能找到 button 的 action 了。

```
(lldb) e @import UIKit
(lldb)  po [0x1003067a0 allTargets]
{(
    <ViewController: 0x10040d9e0>
)}

(lldb) po [0x1003067a0 actionsForTarget:0x10040d9e0
forControlEvent:UIControlEventTouchUpInside]
<__NSArrayM 0x1700588d0>(
onPress
)
```

4.3　类与方法

学习了 OC 语言之后，我们会知道类该怎么定义、方法该怎么定义，也会知道该如何使用它们。但是，iOS 逆向分析人员面对的往往是已经编译好的一个文件或者大段的汇编代码，所

能获取的只有静态的二进制值和运行时的一些状态信息。所以，了解 OC 底层实现和结构组成是很有必要的。

4.3.1 类与方法的底层实现

OC 底层其实都是通过 C/C++ 实现的。为了讲解方便，我们新建一个文件，在里面定义 OC 的类、实例方法和类方法，具体如下。

```
#import <UIKit/UIKit.h>

@interface MyClass : NSObject

@property NSString *myProperty;

@end

@implementation MyClass

- (void)myMethod{
    NSLog(@"my method");
}

+ (void)myClassMethod{
    NSLog(@"my class method");
}

@end

int main(int argc, char * argv[]){
    @autoreleasepool{
        return 0;
    }
}
```

用 rewrite 将其转成 C++ 代码。执行如下命令，会在当前位置生成一个对应的 cpp 文件。由于 import UIKit，该文件的体积会特别大。

```
xcrun -sdk iphoneos clang -rewrite-objc -F UIKit -fobjc-arc -arch arm64 ClassAndMethod.m
```

来到文件的最后，直接查看我们编写的那部分代码。可以看到，这里将定义的类写到了 __DATA，__objc_classlist 这个 section 中。这里对应的就是所有类的列表，具体如下。

```
static struct _class_t *L_OBJC_LABEL_CLASS_$ [1] __attribute__((used, section ("__DATA,
__objc_classlist,regular,no_dead_strip")))= {
        &OBJC_CLASS_$_MyClass,
};
```

再往上一点就能看到 static void OBJC_CLASS_SETUP_$_MyClass(void) 方法对类的初始化了，具体如下。

```
extern "C" __declspec(dllimport) struct _class_t OBJC_METACLASS_$_NSObject;

extern "C" __declspec(dllexport) struct _class_t OBJC_METACLASS_$_MyClass __attribute__
((used, section ("__DATA,__objc_data"))) = {
        0, // &OBJC_METACLASS_$_NSObject,
        0, // &OBJC_METACLASS_$_NSObject,
        0, // (void *)&_objc_empty_cache,
        0, // unused, was (void *)&_objc_empty_vtable,
        &_OBJC_METACLASS_RO_$_MyClass,
};

extern "C" __declspec(dllimport) struct _class_t OBJC_CLASS_$_NSObject;

extern "C" __declspec(dllexport) struct _class_t OBJC_CLASS_$_MyClass __attribute__ ((used,
section ("__DATA,__objc_data"))) = {
        0, // &OBJC_METACLASS_$_MyClass,
        0, // &OBJC_CLASS_$_NSObject,
        0, // (void *)&_objc_empty_cache,
        0, // unused, was (void *)&_objc_empty_vtable,
        &_OBJC_CLASS_RO_$_MyClass,
};
static void OBJC_CLASS_SETUP_$_MyClass(void ) {
        OBJC_METACLASS_$_MyClass.isa = &OBJC_METACLASS_$_NSObject;
        OBJC_METACLASS_$_MyClass.superclass = &OBJC_METACLASS_$_NSObject;
        OBJC_METACLASS_$_MyClass.cache = &_objc_empty_cache;
        OBJC_CLASS_$_MyClass.isa = &OBJC_METACLASS_$_MyClass;
        OBJC_CLASS_$_MyClass.superclass = &OBJC_CLASS_$_NSObject;
        OBJC_CLASS_$_MyClass.cache = &_objc_empty_cache;
}
```

OBJC_CLASS_$_MyClass 是一个 _class_t 结构，其定义如下。

```
struct _class_t {
        struct _class_t *isa;
        struct _class_t *superclass;
```

```
        void *cache;
        void *vtable;
        struct _class_ro_t *ro;
};
```

它的 isa 和 superclass 都是 _class_t 类型的结构体。从 setup 方法中可以看到，MyClass 类的 isa 是它的 meta MyClass，它的 superclass 是 NSObject。而 meta MyClass 的 isa 是 meta NSObject，它的 superclass 是 meta NSObject。根据 OC 中类的继承关系，可以得到如图 4-14 所示的关系图（图片源于 http://www.sealiesoftware.com/blog/class%20diagram.pdf）。

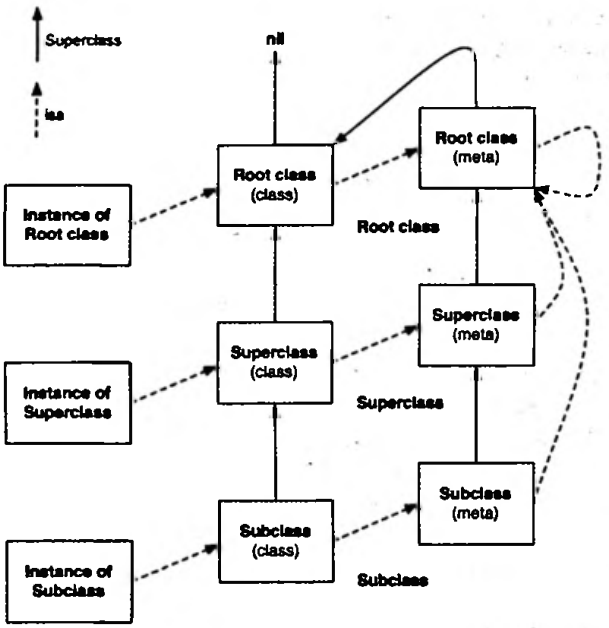


图 4-14　OC 中类的关系图

在图 4-14 中，实线表示 superclass 指针，虚线表示 isa 指针。一个类的实例的 isa 指向它的 class，类的 superclass 指向它的父类，类的 isa 指向 meta-class，类的 meta-class 的 superclass 指向父类的 meta-class。一直到 NSObject 的 meta-class，它的 meta-class 的 superclass 是它自己。NSObject 类的 superclass 为 nil。

我们接着来看 OBJC_CLASS_$_MyClass。其中有一个 _OBJC_CLASS_RO_$_MyClass，这是一个 readonly 的属性，是在编译器中确定的，并且是只读的。找到该变量的定义，具体如下。

```
static struct _class_ro_t _OBJC_CLASS_RO_$_MyClass __attribute__ ((used, section
("__DATA,__objc_const"))) = {
```

```
        0, __OFFSETOFIVAR__(struct MyClass, _myProperty), sizeof(struct MyClass_IMPL),
        0,
        "MyClass",
        (const struct _method_list_t *)&_OBJC_$_INSTANCE_METHODS_MyClass,
        0,
        (const struct _ivar_list_t *)&_OBJC_$_INSTANCE_VARIABLES_MyClass,
        0,
        (const struct _prop_list_t *)&_OBJC_$_PROP_LIST_MyClass,
};

struct _class_ro_t {
        unsigned int flags;
        unsigned int instanceStart;
        unsigned int instanceSize;
        const unsigned char *ivarLayout;
        const char *name;
        const struct _method_list_t *baseMethods;
        const struct _objc_protocol_list *baseProtocols;
        const struct _ivar_list_t *ivars;
        const unsigned char *weakIvarLayout;
        const struct _prop_list_t *properties;
};
```

该结构保存在 "__DATA, __objc_const" 中，它是一个 _class_ro_t 类型的变量，里面包含了实例的大小、方法、变量和属性。_OBJC$_INSTANCE_METHODS_MyClass 的定义如下。

```
struct _objc_method {
        struct objc_selector * _cmd;
        const char *method_type;
        void  *_imp;
};

static struct /*_method_list_t*/ {
        unsigned int entsize;  // sizeof(struct _objc_method)
        unsigned int method_count;
        struct _objc_method method_list[3];
} _OBJC_$_INSTANCE_METHODS_MyClass __attribute__ ((used, section ("__DATA,__objc_const")))
= {
        sizeof(_objc_method),
        3,
        {{(struct objc_selector *)"myMethod", "v16@0:8", (void *)_I_MyClass_myMethod},
        {(struct objc_selector *)"myProperty", "@16@0:8", (void *)_I_MyClass_myProperty},
        {(struct objc_selector *)"setMyProperty:", "v24@0:8@16", (void
*)_I_MyClass_setMyProperty_}}
```

```
};
```

这里定义了方法的个数及每个方法。需要注意的是，这里只有实例方法，而类方法定义在 meta-class 中。所以，可以看到这里的实例方法有 myMethod、myProperty 和 setMyProperty，后两个方法是编译器根据属性 myProperty 自动生成的 get 和 set 方法。方法的定义中包括方法名、方法的签名（Type Encodings）和方法的实现。关于 Type Encodings 的详细介绍，读者可以查看苹果的文档（https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Articles/ocrtTypeEncodings.html）。其中，myMethod 方法的实现如下。

```
...
static __NSConstantStringImpl
__NSConstantStringImpl__var_folders_h3_8n169g610_g69k50z7g_q4gc0000gp_T_ClassAndMethod_
a7c1ea_mi_0 __attribute__ ((section ("__DATA, __cfstring"))) =
{__CFConstantStringClassReference,0x000007c8,"my method",9};
...
static void _I_MyClass_myMethod(MyClass * self, SEL _cmd) {
    NSLog((NSString
*)&__NSConstantStringImpl__var_folders_h3_8n169g610_g69k50z7g_q4gc0000gp_T_ClassAndMeth
od_a7c1ea_mi_0);
}
```

类方法要在 meta-class 中寻找，具体如下。

```
static struct /*_method_list_t*/ {
        unsigned int entsize;  // sizeof(struct _objc_method)
        unsigned int method_count;
        struct _objc_method method_list[1];
} _OBJC_$_CLASS_METHODS_MyClass __attribute__ ((used, section ("__DATA,__objc_const"))) =
{
        sizeof(_objc_method),
        1,
        {{(struct objc_selector *)"myClassMethod", "v16@0:8", (void
*)_C_MyClass_myClassMethod}}
};
```

看一下 MyClass 的实现。它有一个 isa 指针，后跟它的属性，具体如下。

```
struct NSObject_IMPL {
        __unsafe_unretained Class isa;
};

struct MyClass_IMPL {
```

```
    struct NSObject_IMPL NSObject_IVARS;
    NSString *__strong _myProperty;
};
```

最后以一幅图来总结整个分析过程，如图 4-15 所示。

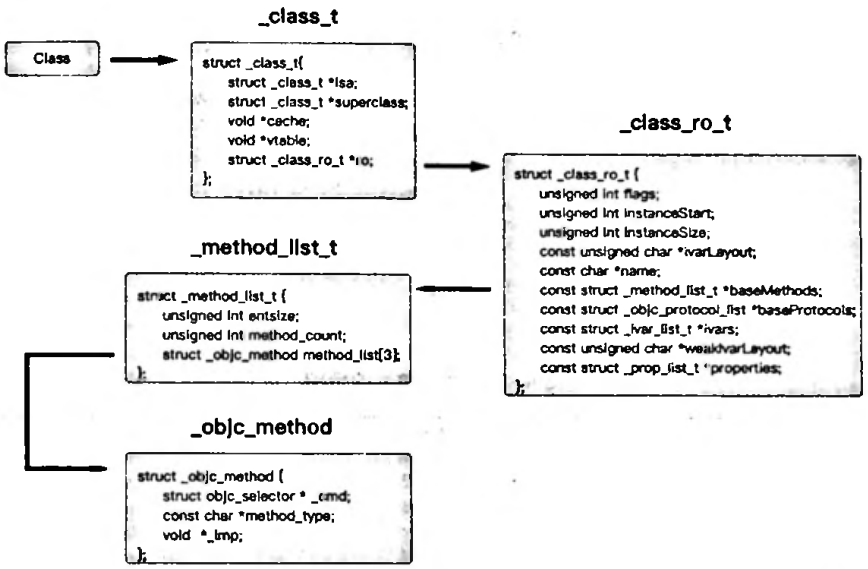


图 4-15　类的结构

4.3.2　运行时类的结构

本节主要讲解类在运行过程中的内存存储结构，以及方法调用的消息机制。苹果也开源了这部分代码（代码在 https://opensource.apple.com/source/objc4/，下载链接为 https://opensource.apple.com/tarballs/objc4/，目前的最新版本为 objc4-723，本节均以该版本代码为例进行分析）。

根据开源代码，可以将运行时类的结构用图来表示，如图 4-16 所示。运行时类的结构中多了一个 class_rw_t，它表示该结构可读、可写（Read、Write），所以在运行时添加的一些动态方法都是添加到该结构上的。

为了更加清晰地展示内存中的结构存储，笔者编写了一个简单的 Demo 程序，用来显示内存中的结构变量（该 Demo 程序可在随书源代码中找到）。打开项目，找到源文件 NSObject+Class.m，从该文件里面将开源代码中的结构定义拿出来。在 printInternalClass 方法中将 class 转换成对应的结构并获取其中的 readwrite 结构，在该处下断点，运行程序，便可以在调试窗口看到结构中对应的信息了，如图 4-17 所示。

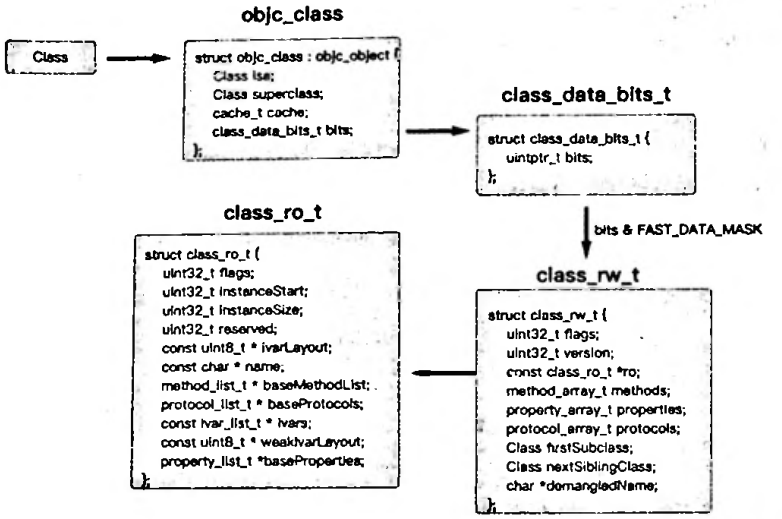


图 4–16　运行时类的结构

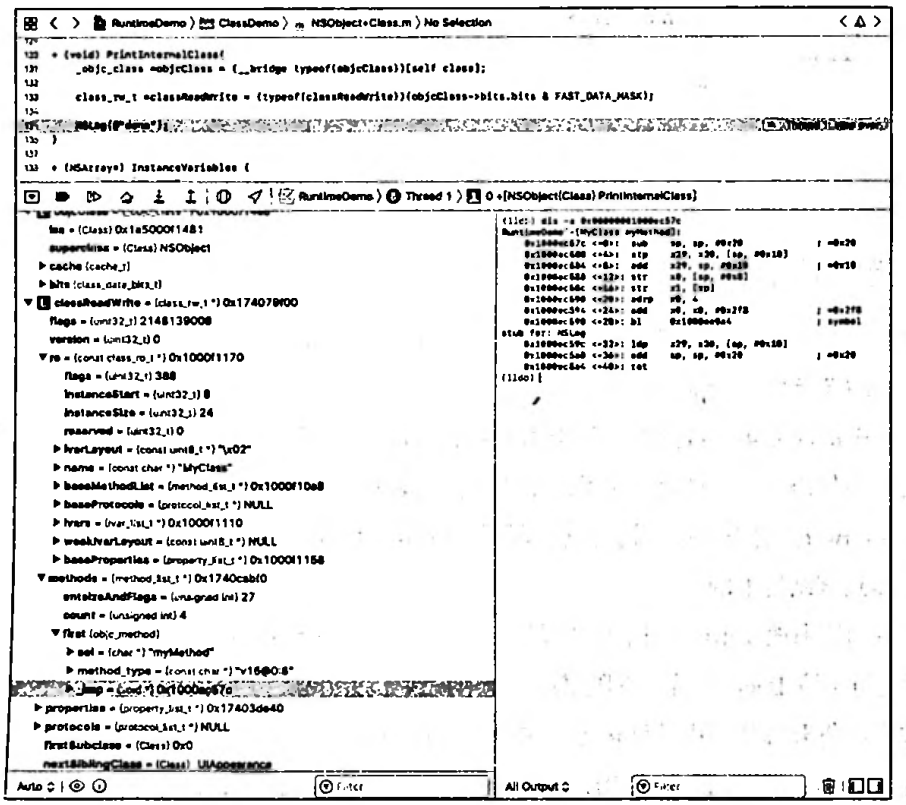


图 4–17　Xcode 调试显示类在内存中的结构信息

还可以根据打印的 class 和 meta-class 对象分别打印它的属性和方法。第 1 个打印的 class 是 MyClass 保存的实例方法，第 2 个打印的 class 是 MyClass meta-class 保存的类方法，具体如下。

```
2017-10-09 23:56:22.167239+0800 RuntimeDemo[1267:283218] Class:MyClass Address:0x1000b12f8
2017-10-09 23:56:22.167382+0800 RuntimeDemo[1267:283218] Class:MyClass Address:0x1000b1320
2017-10-09 23:56:22.167460+0800 RuntimeDemo[1267:283218] Class:NSObject
Address:0x1ad724ec8
(lldb) po [0x1000b12f8 PrintVariables]
<__NSArrayM 0x17004c7e0>(
offset: 8 name:_property type:@"NSString",
offset: 16 name:_myProperty type:@"NSString"
)

(lldb) po [0x1000b1320 PrintVariables]
<__NSArrayM 0x17004c720>(

)

(lldb) po [0x1000b12f8 PrintMethods]
<__NSArrayM 0x17004cae0>(
name:myMethod type encode:v16@0:8 IMP:0x1000ad678,
name:myProperty type encode:@16@0:8 IMP:0x1000ad6d0,
name:setMyProperty: type encode:v24@0:8@16 IMP:0x1000ad708,
name:init type encode:@16@0:8 IMP:0x1000ad5ac,
name:.cxx_destruct type encode:v16@0:8 IMP:0x1000ad74c
)

(lldb) po [0x1000b1320 PrintMethods]
<__NSArrayM 0x17004c9c0>(
name:classMethod type encode:v16@0:8 IMP:0x1000ad6a4
)

(lldb)
```

4.3.3　消息机制

在 OC 中都是通过 [MyClass classMethod] 调用一个方法的。这一点在底层是怎么实现的呢？先写一个简单的 OC 调用，具体如下。

```
#import <UIKit/UIKit.h>

@interface MyClass : NSObject
```

```
+ (void) classMethod;

@end

@implementation MyClass

+ (void) classMethod{
    NSLog(@"class method");
}

@end

int main(int argc, char * argv[]){
    @autoreleasepool {
        [MyClass classMethod];
        return 0;
    }
}
```

通过 rewrite 将其转换成 C++ 代码。我们来看看 OC 的方法在 C++ 里面是通过什么方式调用的，具体如下。

```
xcrun -sdk iphoneos clang -rewrite-objc -F UIKit -fobjc-arc -arch arm64 OCMessage.m
```

在生成的 cpp 文件中找到 main 函数对应的 C++ 代码。通过如下代码可以知道，[MyClass classMethod] 被转换成了 objc_msgSend 的调用，第 1 个参数是调用的类，第 2 个参数是调用的方法。

```
int main(int argc, char * argv[]){
    /* @autoreleasepool */ { __AtAutoreleasePool __autoreleasepool;
        ((void (*)(id, SEL))(void *)objc_msgSend)((id)objc_getClass("MyClass"),
sel_registerName("classMethod"));
        return 0;
    }
}
```

那么，objc_msgSend 又是怎么确定应该调用哪个方法的呢？它有一个动态查找过程，具体如下。

①在相应对象的缓存方法列表中（objc_class 的 cache）查找调用的方法。

②如果没找到，则在相应对象的方法列表中找调用方法。

③如果还没找到，就到父类指针指向的对象中执行①和②两步。

④如果直到根类都没找到，就进行消息转发，给自己保留处理找不到方法这一状况的机会。

⑤调用 resolveInstanceMethod，有机会让类添加这个函数的实现。

⑥调用 forwardingTargetForSelector，让其他对象执行这个函数。

⑦调用 forwardInvocation，更加灵活地处理函数调用。

⑧如果通过以上操作都没有找到，也没有进行特殊处理，就抛出 doesNotRecognizeSelector 异常。

①～④步可以用图 4-18 表示。根据类的层次，逐层往上查找方法表中有没有对应的方法。

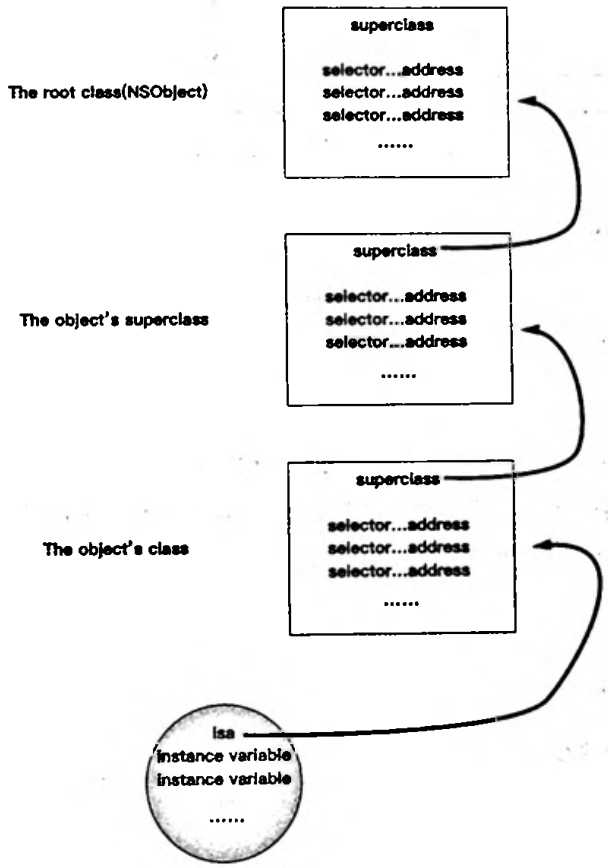


图 4-18　根据类的层级查找方法

⑤～⑦步可以用图 4-19 表示。有 3 次机会处理方法找不到的情况。利用这种方式，可以让所有方法通过消息转发跟踪到一个类的所有方法的调用。

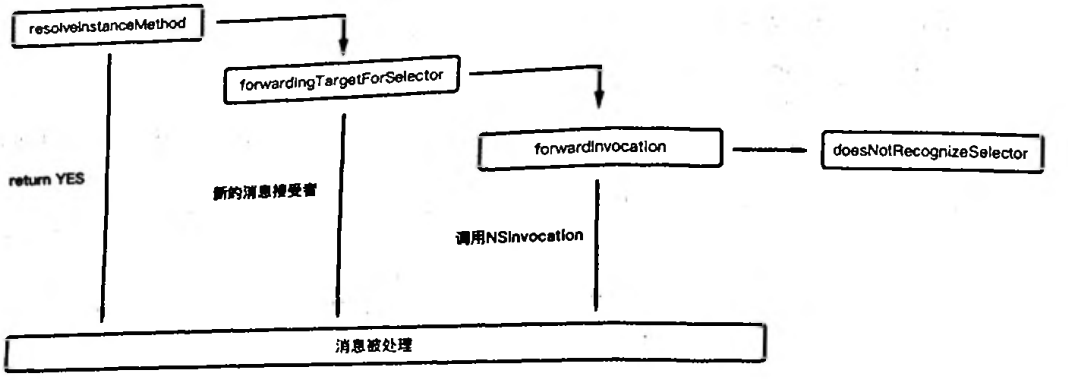


图 4-19 消息转发机制

4.3.4 runtime 的应用

基于强大的 runtime，在逆向工程中可以动态获取类和属性、绑定属性替换方法的实现等。下面介绍几种常用的方法。

1. KVC

Key Value Coding（简称“KVC”）的操作方法是由 NSKeyValueCoding 协议提供的，NSObject 实现了这个协议。几乎所有的对象都支持 KVC 操作。例如，某个类定义了一个私有的属性，即使没有导出，通过 KVC 也能直接获取，具体如下。

```
//MyClass.m

@interface MyClass ()
{
    NSString * _property;
}
@end


MyClass *myClass = [[MyClass alloc] init];

//KVC
NSString* property = [myClass valueForKey:@"_property"];
```

```
NSLog(@"property: %@", property);
```

此处的对象是直接生成的。在逆向过程中，一般会先 hook 某个类的方法，拿到该类的实例，然后通过实例对象直接获取对应的私有属性。类的属性可以通过 class-dump 获取。除了通过 KVC 获取，也可以直接用属性的偏移加对象的地址得到属性的对象，代码如下。具体的实例会在后面详细讲解。

```
Ivar ivar = class_getInstanceVariable(objc_getClass("MyClass"), "_property");
if(ivar){
    NSString* ivarProperty = (__bridge NSString *)(*(void**)((__bridge void*)myClass +
ivar_getOffset(ivar)));
    NSLog(@"ivarProperty: %@", ivarProperty);
}
```

2. AssociatedObject

对象在实例化后是不能动态添加属性的，除非在动态创建一个类时。在逆向中，如果想给一个已经存在的对象添加一个成员属性，该怎么办呢？可以通过关联对象（Associated Object）实现我们的需求。关联对象相当于把一个对象关联到另外一个对象上。在关联后可以随时获取该关联的对象，在对象被销毁时会移除所有关联的对象。

下面来看一个例子。

```
static const void *kAssociatedKey = &kAssociatedKey;

- (void)viewDidLoad {
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.

    MyClass *myClass = [[MyClass alloc] init];

    //KVC
    NSString* property = [myClass valueForKey:@"_property"];
    NSLog(@"property: %@", property);

    //AssociatedObject
    objc_setAssociatedObject(myClass, kAssociatedKey, @"AssociatedObject",
OBJC_ASSOCIATION_RETAIN_NONATOMIC);

    NSString* associatedString = objc_getAssociatedObject(myClass, kAssociatedKey);
```

```
    NSLog(@"associatedString: %@", associatedString);
}
```

先通过 objc_setAssociatedObject 给 myClass 对象关联一个字符串，再通过 objc_getAssociated Object 获取关联的值。其中，objc_setAssociatedObject 的第 2 个参数 key 要保证其唯一性，一般有以下 3 种写法。

- 声明 static void *kAssociatedKey，将 &kAssociatedKey 作为 key。
- 声明 static const void *kAssociatedKey = &kAssociatedKey，将 kAssociatedKey 作为 key。
- 将 selector 作为 key。

最后一个参数 objc_AssociationPolicy 用于指定关联策略，具体如下（其用法可以参考文档中的说明）。

```
typedef OBJC_ENUM(uintptr_t, objc_AssociationPolicy) {
    OBJC_ASSOCIATION_ASSIGN = 0,           /**< Specifies a weak reference to the associated
object. */
    OBJC_ASSOCIATION_RETAIN_NONATOMIC = 1, /**< Specifies a strong reference to the
associated object.
                                            *   The association is not made atomically. */
    OBJC_ASSOCIATION_COPY_NONATOMIC = 3,   /**< Specifies that the associated object is
copied.
                                        *   The association is not made atomically. */
    OBJC_ASSOCIATION_RETAIN = 01401,       /**< Specifies a strong reference to the
associated object.
                                            *   The association is made atomically. */
    OBJC_ASSOCIATION_COPY = 01403          /**< Specifies that the associated object is
copied.
                                            *   The association is made atomically. */
};
```

3. Method Swizzling

前面讲到，一个方法的实现是保存在 IMP 里面的。同时，runtime 提供了修改 IMP 的方法和交换两个 IMP 实现的方法。通过交换两个 selector 的实现，可以达到在调用 A 方法时实际调用 B 方法，在 B 方法里面可以继续调用 A 方法的效果。通常把这种操作称为 Method Swizzling，其原理简图如图 4-20 所示。

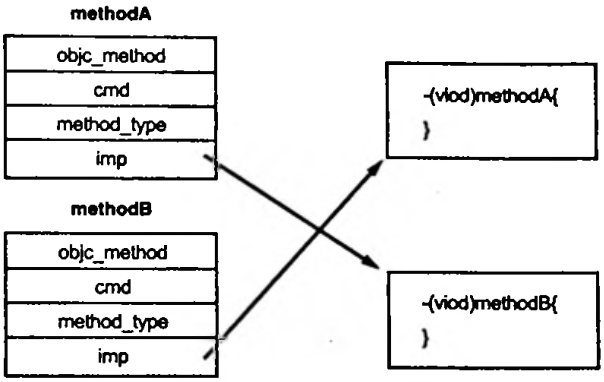


图 4-20　Method Swizzling 的原理

有了简单的认识之后，来看一个实例。给 UIViewController 加一个分类，然后在 load 方法里面交换 viewWillAppear 和 ms_viewWillAppear 的 IMP，这样在调用 -[UIViewController viewWillAppear:]时就会调用 ms_viewWillAppear:，再调用原来的 viewWillAppear 的 IMP 时，因为已经交换了 IMP，所以仍调用 [self ms_viewWillAppear:animated]。相关代码如下。

```
@implementation UIViewController(Swizzle)

+ (void)load {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        Class class = [self class];

        SEL originalSelector = @selector(viewWillAppear:);
        SEL swizzledSelector = @selector(ms_viewWillAppear:);

        Method originalMethod = class_getInstanceMethod(class, originalSelector);
        Method swizzledMethod = class_getInstanceMethod(class, swizzledSelector);

        BOOL didAddMethod =
        class_addMethod(class,
                        originalSelector,
                        method_getImplementation(swizzledMethod),
                        method_getTypeEncoding(swizzledMethod));

        if (didAddMethod) {
            class_replaceMethod(class,
                                swizzledSelector,
                                method_getImplementation(originalMethod),
```

```
                                method_getTypeEncoding(originalMethod));
        } else {
            method_exchangeImplementations(originalMethod, swizzledMethod);
        }
    });
}

#pragma mark - Method Swizzling

- (void)ms_viewWillAppear:(BOOL)animated {
    [self ms_viewWillAppear:animated];
    NSLog(@"viewWillAppear: %@", self);
}

@end
```

采用 Method Swizzling 可以在用户无感知的情况下拦截某个方法进行统计处理。Method Swizzling 在正向开发中可用于埋点、数据监控统计、防止 crash 等，在逆向工程中可以通过对某个方法进行拦截和修改达到修改程序逻辑和数据的目的，在后面的实战中会大量使用该技术。

4.4 App 签名

为了确保安装到手机上的应用是经过认证的合法应用，以及能够根据应用得知其发布者，苹果制定了一个签名机制，所有安装到设备中的应用必须是拥有合法签名的应用。在正向开发中，若是个人开发者，需要购买个人证书或者使用苹果的个人账号免费证书，若是企业开发者，则需要购买企业证书。在 App Store 上架的应用都有苹果证书签名。本节将从如何配置个人证书、签名的原理及如何重签名 3 个方面进行分析讲解。

4.4.1 配置 Xcode 签名

本节介绍个人证书的配置流程。在本机打开 Keychain，单击左上角菜单的“钥匙串访问”→“证书助理”→“从证书颁发机构请求证书”选项，选择“存储到磁盘”选项。这时会生成 CertificateSigningRequest.certSigningRequest 文件，同时会在 Keychain 的密钥中生成一对公私钥，如图 4-21 所示。

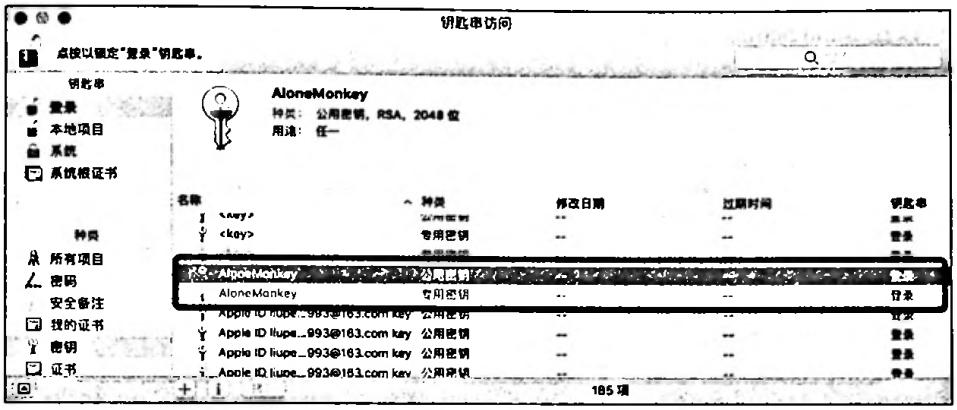


图 4-21　证书助手生成的公私钥对

在苹果开发者后台上传这个文件，生成一个 ios_development.cer 证书文件。双击该文件，将其导入 Keychain，如图 4-22 所示，里面包含了一个私钥。在开发者后台注册设备，指定 App ID，生成描述文件，将其导入本地。Xcode 会复制描述文件并使用该证书对应用进行签名，然后将应用安装到手机中。

图 4-22　查看 Keychian 证书文件

也可以在 Xcode 中登录苹果账号，然后选择自动签名。Xcode 会自动下载描述文件，如图 4-23 所示。

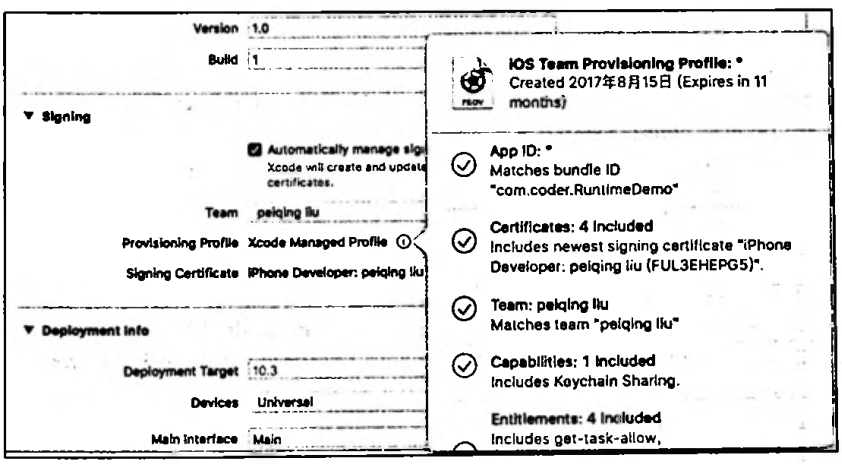


图 4-23　使用 Xcode 自动签名

还可以在 Build Settings 中手动指定签名证书，如图 4-24 所示。

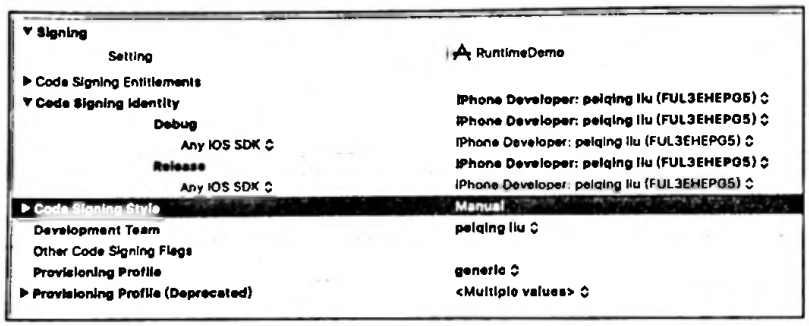


图 4-24 使用 Xcode 手动签名

4.4.2 App 签名的原理

在正向开发中，通常会直接配置 Xcode 的签名，而不会深究。但是在逆向中，常常需要对应用进行修改，这就破解了应用本身的签名。所以，需要重新签名才能将应用安装到非越狱手机上，而在越狱手机上可以使用插件绕过签名验证。因此，我们需要了解签名的原理，从而更好地实现重签名。

1. CertificateSigningRequest.certSigningRequest

这个生成的文件中包含哪些信息呢？使用如下命令查看文件的内容，其中包含：

- 申请者信息
- 申请者公钥（和申请者使用的私钥对应）
- 摘要算法和公钥加密算法

```
➜  Desktop openssl asn1parse -i -in CertificateSigningRequest.certSigningRequest
  0:d=0  hl=4 l= 649 cons: SEQUENCE
  4:d=1  hl=4 l= 369 cons:  SEQUENCE
  8:d=2  hl=2 l=   1 prim:   INTEGER           :00
 11:d=2  hl=2 l=  68 cons:   SEQUENCE
 13:d=3  hl=2 l=  37 cons:    SET
 15:d=4  hl=2 l=  35 cons:     SEQUENCE
 17:d=5  hl=2 l=   9 prim:      OBJECT            :emailAddress
 28:d=5  hl=2 l=  22 prim:      IA5STRING         :liupeiqing1993@163.com
 52:d=3  hl=2 l=  14 cons:    SET
 54:d=4  hl=2 l=  12 cons:     SEQUENCE
 56:d=5  hl=2 l=   3 prim:      OBJECT            :commonName
 61:d=5  hl=2 l=   5 prim:      UTF8STRING        :Coder
```

```
  68:d=3  hl=2 l=  11 cons:    SET
  70:d=4  hl=2 l=   9 cons:     SEQUENCE
  72:d=5  hl=2 l=   3 prim:      OBJECT            :countryName
  77:d=5  hl=2 l=   2 prim:      PRINTABLESTRING   :CN
  81:d=2  hl=4 l= 290 cons:   SEQUENCE
  85:d=3  hl=2 l=  13 cons:    SEQUENCE
  87:d=4  hl=2 l=   9 prim:     OBJECT            :rsaEncryption
  98:d=4  hl=2 l=   0 prim:     NULL
 100:d=3  hl=4 l= 271 prim:    BIT STRING
 375:d=2  hl=2 l=   0 cons:   cont [ 0 ]
 377:d=1  hl=2 l=  13 cons:  SEQUENCE
 379:d=2  hl=2 l=   9 prim:   OBJECT            :sha256WithRSAEncryption
 390:d=2  hl=2 l=   0 prim:   NULL
 392:d=1  hl=4 l= 257 prim:  BIT STRING
```

2. ios_development.cer

取出 CertificateSigningRequest.certSigningRequest 的公钥，添加账号信息，再通过哈希算法生成一个信息摘要，使用苹果的 CA 私钥进行加密，这在证书中称为数字签名。

查看证书文件信息，具体如下，其中包含：

- 申请者信息
- 申请者公钥
- 通过苹果私钥加密的数字签名

```
➜ Desktop openssl x509 -inform der -in ios_development.cer -noout -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            03:2c:51:be:79:06:7b:c7
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, O=Apple Inc., OU=Apple Worldwide Developer Relations, CN=Apple
Worldwide Developer Relations Certification Authority
        Validity
            Not Before: May  5 01:59:04 2017 GMT
            Not After : May  5 01:59:04 2018 GMT
        Subject: UID=A69STSWBQ7, CN=iPhone Developer: peiqing liu (FUL3EHEPG5),
OU=9XQEPG2J2J, O=peiqing liu, C=US
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
```

```
                    00:b8:7c:63:f2:ab:a4:7d:3e:4b:fa:c4:1a:1d:a4:
                    7d:9c:6f:1c:e3:b7:1a:b1:e1:6d:78:e1:a8:04:4c:
                    4d:23:71:92:ce:ce:87:f3:3f:ab:74:70:26:19:c9:
                    b9:c4:e5:d1:75:85:46:54:a1:b6:83:59:16:1e:11:
                    39:f7:21:46:a0:0e:84:ca:d9:ab:1f:14:8a:19:21:
                    60:1a:9b:d0:58:e1:87:e4:61:e9:29:aa:b2:bb:d4:
                    fe:ae:46:42:df:d9:c6:0c:c5:85:4a:e9:84:fd:c7:
                    25:7e:e3:7a:66:2f:15:db:8a:40:38:30:f6:25:7f:
                    da:3b:98:44:cb:c0:ca:a1:67:f3:a4:79:f7:0d:de:
                    0e:70:33:c7:63:2f:47:2a:68:e4:65:7c:01:0d:6b:
                    52:84:a2:6a:5a:a0:de:42:e5:25:e3:51:87:bc:32:
                    80:f1:af:62:68:21:8c:6f:65:e2:f1:fc:02:ec:ab:
                    4b:8e:b3:52:c1:bf:3a:10:0b:f8:a0:01:f9:f1:5b:
                    2e:50:62:65:3e:05:26:b3:e8:89:48:40:b6:9f:b2:
                    63:e2:84:a1:e8:ba:c7:75:c0:2c:fd:e4:81:4b:d2:
                    40:3f:ee:3e:3e:af:34:2c:68:44:63:5d:ff:2e:87:
                    c2:d3:4d:69:3b:04:b1:f6:c9:47:74:45:a9:30:3c:
                    77:0f
                Exponent: 65537 (0x10001)
            X509v3 extensions:
                Authority Information Access:
                    OCSP - URI:http://ocsp.apple.com/ocsp03-wwdr01

                X509v3 Subject Key Identifier:
                    B9:37:92:61:3A:00:53:05:20:1D:7F:26:A2:33:87:15:D5:11:44:23
                X509v3 Basic Constraints: critical
                    CA:FALSE
                X509v3 Authority Key Identifier:
                    keyid:88:27:17:09:A9:B6:18:60:8B:EC:EB:BA:F6:47:59:C5:52:54:A3:B7

                X509v3 Certificate Policies:
                    Policy: 1.2.840.113635.100.5.1
                      User Notice:
                        Explicit Text: Reliance on this certificate by any party assumes acceptance
of the then applicable standard terms and conditions of use, certificate policy and
certification practice statements.
                      CPS: http://www.apple.com/certificateauthority/

                X509v3 Key Usage: critical
                    Digital Signature
                X509v3 Extended Key Usage: critical
                    Code Signing
                1.2.840.113635.100.6.1.2: critical
                    ..
    Signature Algorithm: sha256WithRSAEncryption
```

```
31:3f:e5:40:4e:0d:ba:3c:d1:b1:c1:59:64:e2:82:08:bb:05:
c7:a5:4a:04:d8:39:c1:b8:e3:63:98:b5:d8:4c:54:c8:cf:49:
26:11:03:ad:00:85:8d:e2:91:fa:bc:b3:35:d0:0b:a7:69:9e:
d1:8d:7b:b5:64:7e:de:3d:6a:f1:66:7f:53:f7:bf:54:b9:3a:
35:dc:0f:06:d7:39:7c:0b:cf:f5:56:f5:97:43:4b:46:26:b5:
dc:e7:3b:98:f7:7c:64:00:24:11:df:4c:76:7a:79:ab:12:0d:
9a:9d:a7:2d:e3:60:f7:44:9b:cd:6b:4f:cd:2a:ee:52:32:8b:
fc:b1:b8:0b:1b:5b:78:b9:d9:13:a0:7a:de:58:67:8a:d6:86:
40:f1:7e:a8:c0:ab:c8:c4:25:a4:53:37:a5:af:c9:07:7b:83:
ef:75:17:cb:0e:63:57:2b:41:b6:4c:fc:5a:19:08:6e:ab:68:
82:0b:c5:42:54:a7:f2:96:5b:68:98:16:2a:41:b3:e9:8d:a5:
28:50:e1:00:f3:1c:17:d9:ef:7d:09:6a:b1:2a:77:a2:b4:72:
7d:a8:ce:f5:37:ad:67:f8:f8:b7:da:b4:e6:3a:a8:0d:c2:ca:
ea:38:6f:c3:bf:db:11:6d:23:1b:71:b6:f3:e6:69:07:82:4f:
f2:2c:d3:46
```

在这里思考一下苹果如何保证证书（公钥）是可信的。在 iOS 系统中存在一个和 CA 私钥对应的公钥，使用该公钥对数字签名进行解密，生成一个信息摘要，该信息摘要和它本身通过申请者信息和申请者公钥与经过哈希算法生成的信息摘要进行比较，只有相等才说明这个证书是可信的，如图 4-25 所示。

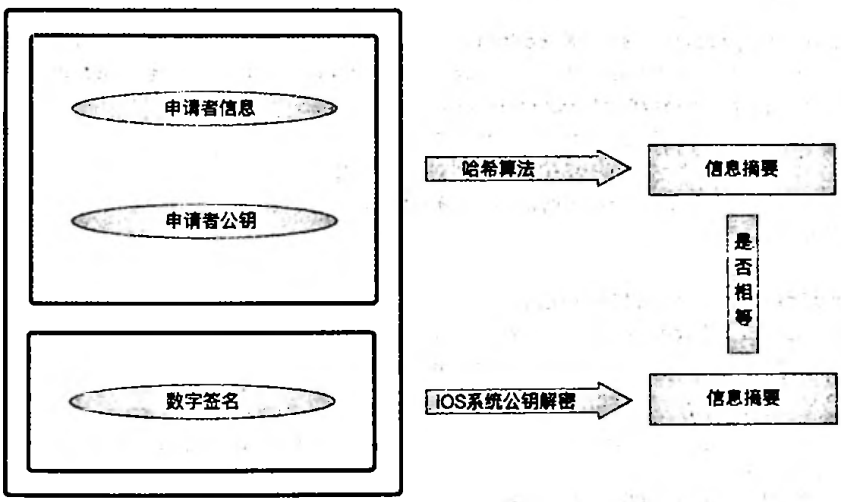


图 4-25　保证证书（公钥）是可信的

此外，可以通过如下命令来搜索本机的证书。

```
➜  Desktop security find-identity -v -p codesigning

 1) 090C4AD8DAAF6801B59C577A5D42D33370BDD09A "iPhone Distribution: peiqing liu
```

```
(9XQEPG2J2J)"
  2) 9FCDBC5F4AE2ADF5E9E64F0CD909D047D9EF8615 "iPhone Developer: peiqing liu (FUL3EHEPG5)"
  3) F3F735483B97A70C42C182E52774BB7C7ACCF6B6 "iPhone Developer: liupeiqing1993@163.com
(Z67NE84J9Z)"
  4) 6313B2ACED2A640C14C3F31A28B0EA91E42D85DA "Mac Developer: peiqing liu (FUL3EHEPG5)"
  5) 61C858A14443B54F8B56D224108A312EEC781475 "gdb-cert"
     5 valid identities found
```

在对应用进行签名时，先使用证书所对应的私钥去对代码和资源文件等进行签名，在苹果系统检验证书合法后得到对应的公钥，再使用该公钥对应用的签名合法性进行验证。

3. 授权文件

简单地说，授权文件（Entitlements）是一个沙盒的配置列表，其中列出了哪些行为会被允许、哪些行为会被拒绝。例如，在重签应用时常常需要调试器自动附加，这时就需要在授权文件里面将 get-task-allow 设置为 true。这是一个 plist 文件，在签名时 Xcode 会将这个文件作为 --entitlements 参数的内容传给 codesign。

在 Xcode 的“Capabilities”选项卡中进行相应的设置后，相关条目也会添加到授权文件中。可以使用如下命令查看一个应用的授权信息。

```
➜  Debug-iphoneos git:(master) X codesign -d --entitlements - RuntimeDemo.app
Executable=/Users/monkey/Documents/iosreversebook/sourcecode/chapter-44/RuntimeDemo/Bui
ld/Products/Debug-iphoneos/RuntimeDemo.app/RuntimeDemo
◆◆qq◆<?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>9XQEPG2J2J.com.coder.RuntimeDemo</string>
    <key>com.apple.developer.team-identifier</key>
    <string>9XQEPG2J2J</string>
    <key>get-task-allow</key>
    <true/>
    <key>keychain-access-groups</key>
    <array>
        <string>9XQEPG2J2J.com.coder.RuntimeDemo</string>
    </array>
</dict>
</plist>
```

4. 配置文件

前面提到，公钥包含在证书里面，证书又包含在配置文件（Provisioning Profiles）里面，配置文件在打包 App 时会被复制到 .app 目录下面。除了证书配置文件，配置文件还包括：

- App ID
- Entitlements 授权文件
- 使用的证书
- 可安装的设备列表

可以使用如下命令查看配置文件。

```
➜  RuntimeDemo.app git:(master) X security cms -D -i embedded.mobileprovision
security: SecPolicySetValue: One or more parameters passed to a function were not valid.
<?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>AppIDName</key>
        <string>generic</string>
        <key>ApplicationIdentifierPrefix</key>
        <array>
        <string>9XQEPG2J2J</string>
        </array>
        <key>CreationDate</key>
        <date>2017-05-05T02:14:04Z</date>
        <key>Platform</key>
        <array>
                <string>iOS</string>
        </array>
        <key>DeveloperCertificates</key>
        <array>

<data>MIIFljCCBH6gAwIBAgIIaZ3xfvJrcK0wDQYJKoZIhvcNAQELBQAwgZYxCzAJBgNVBAYTAlVTMRMwEQYDV
QQKDApBcHBsZSBJbmMuMSwwKgYDVQQLDCNBcHBsZSBXb3JsZHdpZGUgRGV2ZWxvcGVyIFJlbGF0aW9uczFEMEIG
A1UEAww7QXBwbGUgV29ybGR3aWRlIERldmVsb3BlciBSZWxhdGlvbnMgQ2VydGlmaWNhdGlvbiBBdXRob3JpdHk
wHhcNMTYwNjA2MTYxMDMyWhcNMTcwNjA2MTYxMDMyWjCBiTEaMBgGCgmSJomT8ixkAQEMCkE2OVNUU1dCUTcxMz
AxBgNVBAMMKmlQaG9uZSBEZXZlbG9wZXI6IHBlaXFpbmcgbGl1IChGVUwzRUhFUEc1KTETMBEGA1UECwwKOVhRR
VBHMkoySjEUMBIGA1UECgwLcGVpcWluZyBsaXUxCzAJBgNVBAYTAlVTMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8A
MIIBCgKCAQEAvC3sKawCxoMRWsTLWKh71rsa0MYpee3oRF5AG27ntrnhU0FPmODkuXmIOG9A3VnhVPj+

......+MIHDBggrBgEFBQcCAjCBtgyBs1JlbGlhbmNlIG9uIHRoaXMgY2VydGlmaWNhdGUgYnkgYW55IHBhcnR5
IGFzc3VtZXMgYWNjZXB0YW5jZSBvZiB0aGUgdGhlbiBhcHBsaWNhYmxlIHN0YW5kYXJkIHRlcm1zIGFuZCBjb25
```

kaXRpb25zIG9mIHVzZSwgY2VydGlmaWNhdGUgcG9saWN5IGFuZCBjZXJ0aWZpY2F0aW9uIHByYWN0aWNlIHN0YX
RlbWVudHMuMDYGCCsGAQUFBwIBFipodHRwOi8vd3d3LmFwcGxlLmNvbS9jZXJ0aWZpY2F0ZWF1dGhvcml0eS8wD
gYDVR0PAQH/BAQDAgeAMBYGA1UdJQEB/wQMMAoGCCsGAQUFBwMDMBMGCiqGSIb3Y2QGAQIBAf8EAgUAMA0GCSqG
SIb3DQEBCwUAA4IBAQAxP+VATg26PNGxwVlk4oIIuwXHpUoE2DnBuONjmLXYTFTIz0kmEQOtAIWN4pH6vLM10Au
naZ7RjXu1ZH7ebWrxZn9T979UuTo13A8G1zl8C8/1VvWXQ0tGJrXc5zuY93xkABQR30x2enmrEg2anact42D3RJ
vNa0/NKu5SMov8sbgLG1t4udlToHreWGeK1oZA8X6owKvIxCWkUzelr8kHe4PvdRfLDmNXK0G2TPxaGQhuq2iCC
8VCVKfylltoiBYqQbPpjaUoUOEA8xwX2e99CWqxKneitHJ9qM71N61n+Pi32rTmOqgNwsrqOG/Dv9sRbSMbcbbz
5mkHgk/yLNNG
	</array>
	<key>Entitlements</key>
	<dict>

		<key>keychain-access-groups</key>
		<array>
			<string>9XQEPG2J2J.*</string>
		</array>
		<key>get-task-allow</key>
		<true/>
		<key>application-identifier</key>
		<string>9XQEPG2J2J.*</string>
		<key>com.apple.developer.team-identifier</key>
		<string>9XQEPG2J2J</string>
	</dict>
	<key>ExpirationDate</key>
	<date>2018-05-05T02:14:04Z</date>

	<key>Name</key>
	<string>generic</string>
	<key>ProvisionedDevices</key>
	<array>
		<string>aea20485e7423axxxxxxxxxx3f46bb4f6f84c7d7</string>
		<string>3ff1a6c1623076xxxxxxxxxx6018a82b34daec30</string>
		<string>2044c686be5d2fxxxxxxxxxx2f546192756f92c9</string>
		<string>50365df5eb50d2xxxxxxxxxx8e82fb26499fe0e4</string>
		<string>d79195a2e12f8fxxxxxxxxxx1a30be5a73fb2285</string>
		<string>05d7d7ce97f88cxxxxxxxxxx2929eb9af46b33f9</string>
	</array>
	<key>TeamIdentifier</key>
	<array>
		<string>9XQEPG2J2J</string>
	</array>
	<key>TeamName</key>
	<string>peiqing liu</string>
	<key>TimeToLive</key>
	<integer>365</integer>

```
        <key>UUID</key>
        <string>6a1d3d59-30d6-4d5a-bd53-bea4b6678c56</string>
        <key>Version</key>
        <integer>1</integer>
</dict>
</plist>
```

应用启动时会检查配置文件和当前运行 App 的信息是否匹配，如图 4-26 所示。

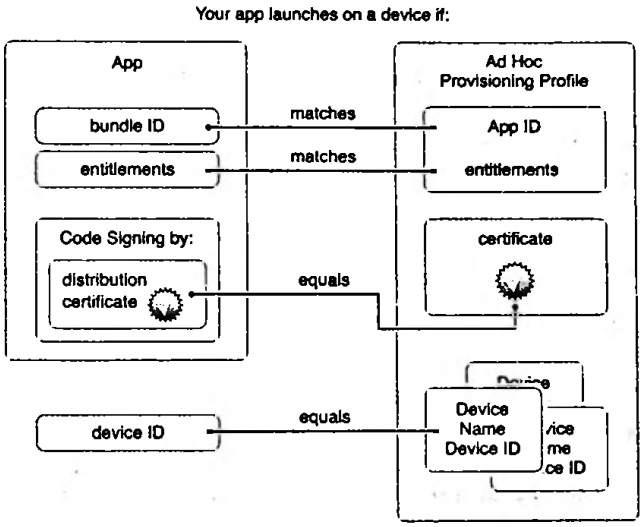


图 4-26　App 启动配置文件校验

4.4.3　重签名

一旦改变了应用的二进制文件，或者增加或修改了应用里面的资源，应用本身的签名就会被破坏。如果想将修改的文件安装到手机上，就需要对应用重新进行签名（其实就是手动完成 Xcode 签名），步骤如下。

01　获取证书列表

通过如下命令获取本机证书的列表，然后选择一个开发者证书。

```
security find-identity -p codesigning -v
  1) 090C4AD8DAAF6801B59C577A5D42D33370BDD09A "iPhone Distribution: peiqing liu
(9XQEPG2J2J)"
  2) 9FCDBC5F4AE2ADF5E9E64F0CD909D047D9EF8615 "iPhone Developer: peiqing liu (FUL3EHEPG5)"
  3) F3F735483B97A70C42C182E52774BB7C7ACCF6B6 "iPhone Developer: liupeiqing1993@163.com
(Z67NE84J9Z)"
```

```
4) 6313B2ACED2A640C14C3F31A28B0EA91E42D85DA "Mac Developer: peiqing liu (FUL3EHEPG5)"
5) 61C858A14443B54F8B56D224108A312EEC781475 "gdb-cert"
   5 valid identities found
```

第 2 个是个人开发者证书，第 3 个是免费证书。在这里选择第 2 个证书。

02 生成 entitlements.plist

新建一个 Xcode iOS App 项目，编译生成目标 App。从目标 App 目录下获取 embedded.mobileprovision，或者从开发者账号后台下载 Provisioning Profiles 文件，然后提取其中的授权文件（注意：授权文件中的 Bundle ID 要和重签名应用的 Bundle ID 匹配），具体如下。

```
security cms -D -i embedded.mobileprovision > profile.plist
/usr/libexec/PlistBuddy -x -c 'Print :Entitlements' profile.plist > entitlements.plist
```

03 复制 mobileprovision

将 xxxx.mobileprovision 文件复制到 .app 目录下。

04 签名

运行如下命令，对 .app 文件夹中的所有动态库、插件、watch 目录下的 extension 进行签名。

```
codesign -f -s 9FCDBC5F4AE2ADF5E9E64F0CD909D047D9EF8615 xxx.dylib
```

对整个 .app 文件夹进行签名，具体如下。

```
codesign -f -s 9FCDBC5F4AE2ADF5E9E64F0CD909D047D9EF8615 --entitlements entitlements.plist
Target.app
```

05 打包 ipa

打包文件，具体如下。

```
mkdir Payload
cp -r Target.app ./Payload
zip -qr Target.ipa ./Payload
```

最后，使用 iTools、Xcode 或 mobile_device 安装重新签名后的 ipa 文件。

第 5 章　分析与调试

在逆向过程中，通过前面介绍的方法，我们可以从界面、类与方法、网络等方面及一个应用的功能点入手寻找线索。但是，要想精确定位一个功能点的实现位置或者实现原理，必须结合动态调试和静态分析来分析程序的行为及实现。本章将从静态分析和动态调试两个大的方面介绍逆向过程中最重要的环节，和读者一起探索静态分析函数的执行流程、静态库的分析方法及如何调试注入的动态库和第三方应用程序。

5.1　静态分析

静态分析是指在不运行程序的前提下进行程序分析的一种方法，一般会配合已经得到的线索和分析工具来进行分析。静态分析分为以下 3 种。

- 基于 ipa 和 app 包的静态分析。
- 基于文件格式的静态分析。
- 基于二进制反汇编的静态分析。

本节主要对二进制反汇编的静态分析进行讲解。

说到反汇编，首先要介绍反汇编工具 Hopper（https://www.hopperapp.com/）和 IDA（https://www.hex-rays.com/）。Hopper 只提供 Mac 和 Linux 版本，IDA 支持 Windows、Mac 和 Linux 平台，但两者都可以显示被分析文件的反汇编代码、流程图及伪代码，也可以直接修改汇编指令，生成新的可执行文件。一般来说 Mac 上的 Hopper 已经够用了，不过从 Windows 逆向转到 iOS 逆向的开发者可能更喜欢 IDA。尽管 IDA 的功能更加强大，但是它用于翻译伪代码的插件是需要另外付费购买才能使用的。下面以笔者编写的 UserLogin 应用为例，分别讲解 Hopper 和 IDA 的使用。

5.1.1　Hopper

我们一起认识一下 Hopper。

1. Hopper 的界面

从官网下载 Hopper 并打开，单击右键快捷菜单中的“UserLogin.app”选项，选择“显示包内容”选项，然后将显示内容里面的 UserLogin 可执行文件拖入 Hopper 主界面，便会出现如图 5-1 所示的界面。

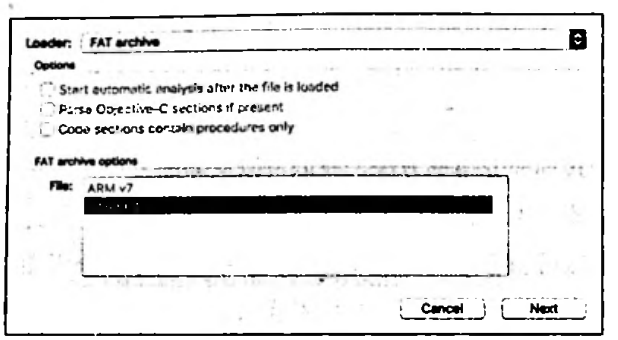


图 5-1 使用 Hopper 打开可执行文件

Hopper 会自动识别文件格式。当前识别出来的文件格式是 FAT 的 Mach-O 文件，其中包含两个架构的代码，分别是 ARMv7 和 AArch64。选择 AArch64 架构的代码并打开，可以看到 Hopper 的主界面及分析结果，如图 5-2 所示。

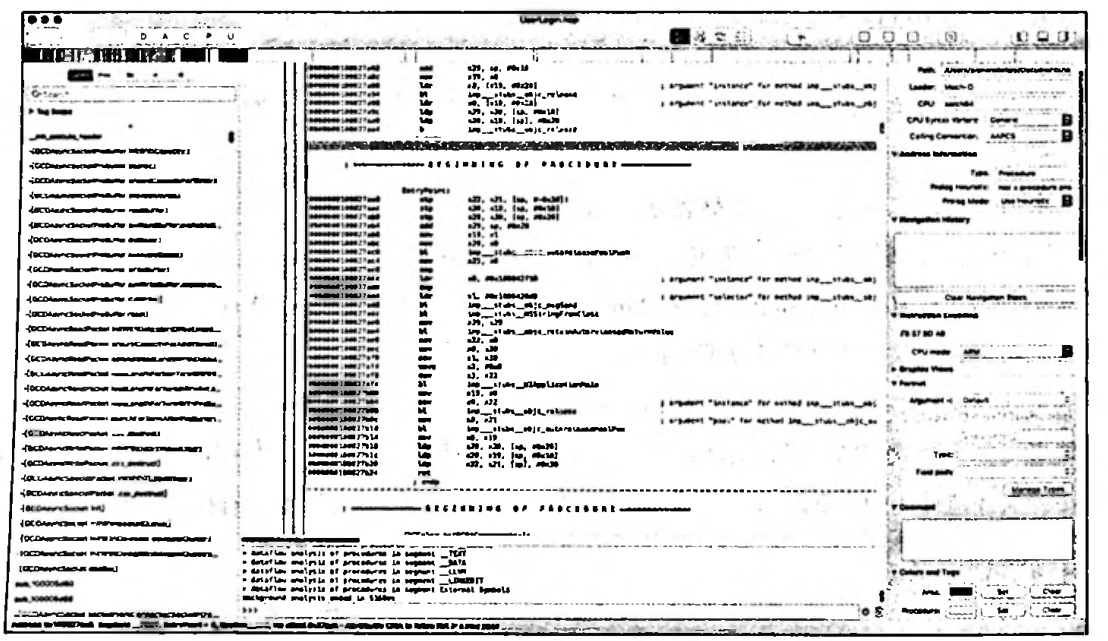


图 5-2 Hopper 主界面

Hopper 的主界面主要分为如下 5 个部分。

- 界面上方是 Hopper 的工具栏，左侧的“D”“A”“C”“P”“U”按钮用于转换当前数据的显示格式（分别是数据、ASCII 字符串、代码、函数、未定义数据）。在工具栏的中间可以切换代码的表示形式，从左到右分别是汇编代码、流程图、伪代码和二进制形式。工具栏右侧是用于控制分栏显示和界面布局展示的工具。
- 界面左侧显示了定义在文件中的所有符号和字符串，可以通过搜索来查找想要的符号。
- 界面中间部分显示的是汇编代码，包括变量、函数的解析及程序的跳转流程等。
- 界面右侧显示的是当前所选的分析代码的上下文信息，包括真实的二进制、是 ARM 还是 Thumb 汇编、引用关系等。
- 界面下方用于展示当前分析进度。分析进度也可以根据工具栏下方的彩色条形图来判断。此外，Hopper 还支持通过 Python 脚本来获取和修改当前文件的一些信息。

当然，在做静态分析之前，需要确定待分析的函数。可以通过 Cycript 打印或 class-dump 查看头文件的方法寻找线索，或者根据动态分析得到的线索做进一步的静态分析，也可以直接按猜测的关键字搜索。在这里，根据 class-dump 头文件找到 userLogin 函数，搜索后双击 `-[ULLoginViewController userLogin]`，定位具体的汇编代码，如图 5-3 所示。接下来，就可以分析该函数的实现了。

为了分析方便，以下均以地址的后 4 字节来定位代码，例如“0x10002b9ec”用“b9ec”表示。函数的头部实现了保存原有堆栈和开辟新的堆栈，这部分内容将在 6.3 节详细讲解。

下面我们从 ba1c 开始讲解。ldr 取地址上的值，并将其保存到目标寄存器中。单击地址 0x100042f20，可以看到这里保存的是 ULLoginManager 类，如图 5-4 所示，所以是在这里获取了 ULLoginManager 类。同样，在 ba24 处获取方法 sharedManager（Hopper 已经将其标注出来了）。ba28 调用了 C 函数 objc_msgSend。因为 OC 函数的调用最终都是通过调用 objc_msgSend 实现的，而且 objc_msgSend 的第 1 个参数是调用的对象或类，第 2 个参数是调用的方法，在这里可以看到，x0 寄存器保存的是 ULLoginManager 类，x1 寄存器保存的是 sharedManager 方法，所以这里调用的便是 `+[ULLoginManager sharedManager]` 函数。在 ARC 中，函数的返回值会通过 ba30 处的调用返回并保存到 x0 寄存器中。

由于 OC 函数的调用最终都会调用 objc_msgSend，下面就从 objc_msgSend 的调用点入手寻找调用方和调用的方法。来到 ba50 处的调用，看看传入的 x0 寄存器和 x1 寄存器的值是怎么得来的。x0 寄存器的值是 ba40 处通过指令 `ldr x0,[x21,x8]` 获取的，而 x21 寄存器的值是在 ba08

处由 x0 寄存器赋予的，函数入口的 x0 寄存器表示的就是调用者本身（这里表示的是 ULogin ViewController 的实例对象）。指令 `ldr x0,[x21,x8]` 表示的是实例对象地址加 x8 偏移的值，x8 寄存器的值是通过读取 0x100043214 处的值获得的。双击 0x100043214 处，得到如图 5-5 所示的结果。可以看到，Hopper 把 3210 和 3214 这两个偏移值当成了一个来解析，解析出来的是偏移 0x18 处的 loginButton 属性，而 3214 对应的是偏移 0x08 处的属性。

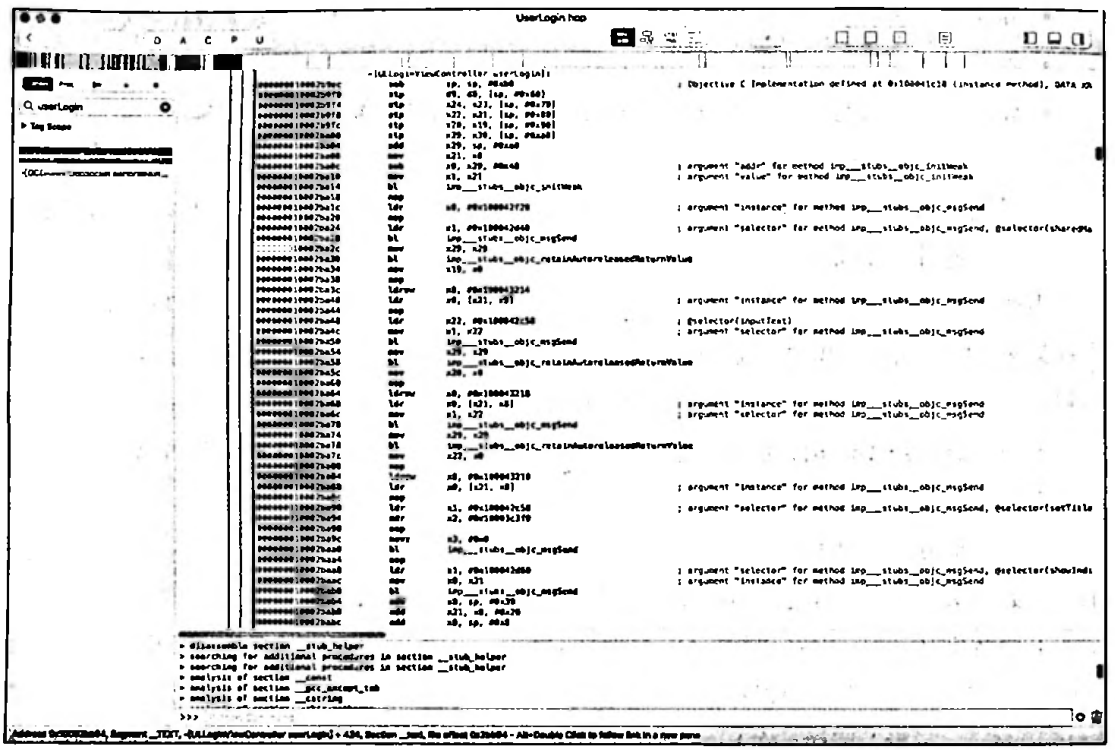


图 5-3　使用 Hopper 查看函数反汇编

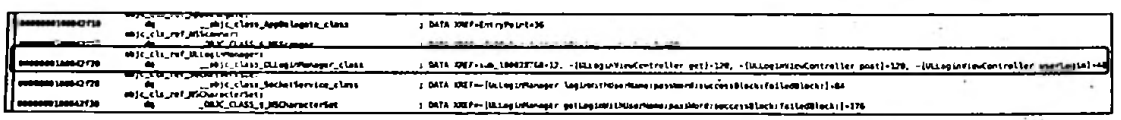


图 5-4　通过地址找到地址保存的类

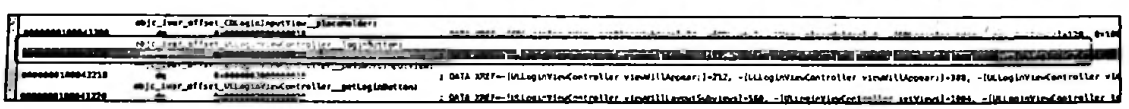


图 5-5　通过地址找到属性偏移值

在这里，需要我们自己找出偏移 0x8 所对应的属性。找到 ULLoginViewController 的结构，它在 0x00000001000435e8 处，如图 5-6 所示。

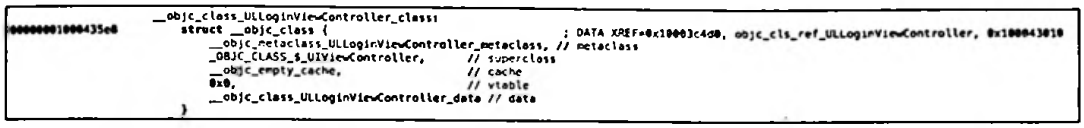


图 5-6　查看 ULLoginViewController 的结构

和分析类的结构时一样，双击 ULLoginViewController_data，来到如图 5-7 所示处，查看其结构，然后双击 ULLoginViewController_ivars，查看它有哪些变量及变量的偏移，结果如图 5-8 所示。其中，count 的值为 6，表示它有 6 个属性。在 64 位环境中每个地址占 8 字节，这 6 个属性分别是 userNameInputView、passWordInputView、loginButton、getLoginButton、postLoginButton、loginLoadingIcon，对应的偏移分别是 0x08、0x10、0x18、0x20、0x28、0x30，这样就可以知道偏移 0x08 处的属性是 _userNameInputView。而且，这里显示的 objc_ivar_offset 竟然是 objc_ivar_offset_ULLoginViewController__loginButton+4（不知道这是不是 Hopper 解析 64 位 __objc_ivar 的 bug，Hopper 对 32 位和 IDA 的解析结果都是正常的）。

回到对 userLogin 的分析。从前面的分析可知，ba50 处调用的是 ULLoginViewController 实例对象的 userNameInputView 属性的 inputText 方法，从字面上理解就是获取用户输入框的值。下一个 objc_msgSend 调用在 ba70 处，x1 表示的还是 inputText 方法，x0 用于获取当前实例对象的属性值，偏移值保存在 0x100043218 处（这一点在图 5-5 中可以直接看到，指的是 passWordInputView 属性）。接着看 baa0 处，按照以上分析方法，可以得出这里的 x0 是当前实例的 loginButton 属性，x1 表示 setTitle:forState: 方法。参数说明如下。

- 第 1 个参数保存在 x2 寄存器中，它取的是 0x10000c648 处存储的字符串“登录中”，如图 5-9 所示。
- 第 2 个参数保存在 x3 寄存器中，值为 0，对应于代码中的 UIControlStateNormal。

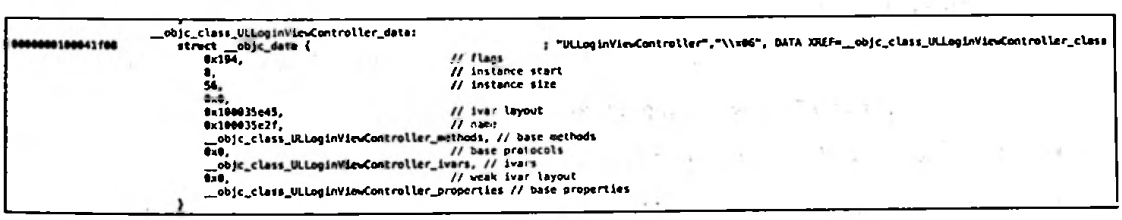


图 5-7　查看 ULLoginViewController_data 的结构

图 5-8 查看 ULLoginViewController_ivars 的结构

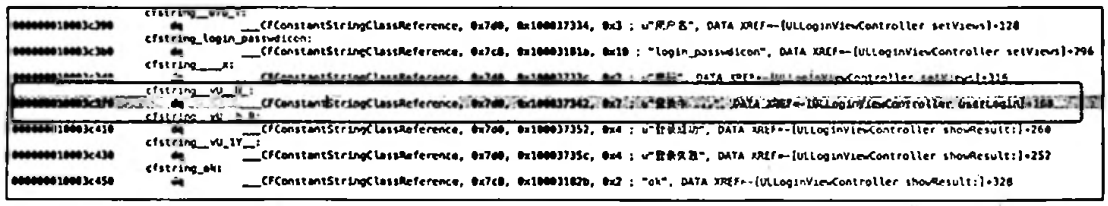


图 5-9 查看指定地址的存储内容

bab0 处调用的是当前实例的 -[ULLoginViewController showIndicatorLoading] 方法。再来看 bb48 处 x0 寄存器的值，它是在 bb3c 处从 x19 寄存器获取的。而 x19 寄存器的赋值在 ba34 处，从 x0 寄存器获取，也就是 [ULLoginManager sharedManager] 的返回值。x1 寄存器表示的方法是 loginWithUserName:passWord:successBlock:failedBlock:。参数说明如下。

- 第 1 个参数在 x2 寄存器中，它是从 x20 寄存器的值取得的，而 x20 寄存器的值是在 ba5c 处赋予的 [userNameInputView inputText] 的返回值。
- 第 2 个参数 x3 是 [passWordInputView inputText] 的返回值。
- 第 3 个参数 x4 的赋值在 bb34 处 :add x4,sp,#0x30，而 sp+0x30 处的赋值在 bacc 处。str x24,[sp,#0x30] 将寄存器 x24 的值赋给 sp+0x30 处的地址，x24 取的是 0x100038040 处的值。双击地址，得到如图 5-10 所示的结果，指向 __NSConcreteStackBlock 函数的地址。

所以，这里是一个 Block 的结构体，对应的代码如下。

```
struct Block_layout {
    void *isa;
    int flags;
    int reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor *descriptor;
    /* Imported variables. */
};
```

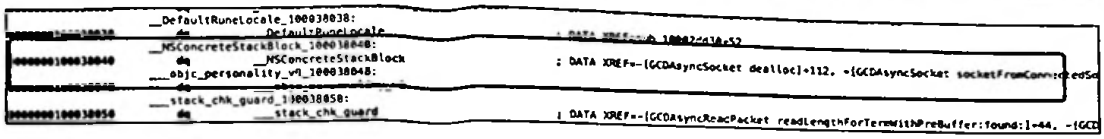


图 5-10　查看指定地址指向的函数

isa 就是 x24 寄存器，指向 __NSConcreteStackBlock 的地址。在 bad8 处，从 0x10002ee38 处取得值 0xc2000000，然后赋值给 flags。bae4 处就是 Block 内部函数指针的赋值，指向的地址为 0x10002bbcc。经分析，0x10002bbcc 处久是 Block 函数的实现。双击地址 0x10002bbcc，来到如图 5-11 所示的地方。

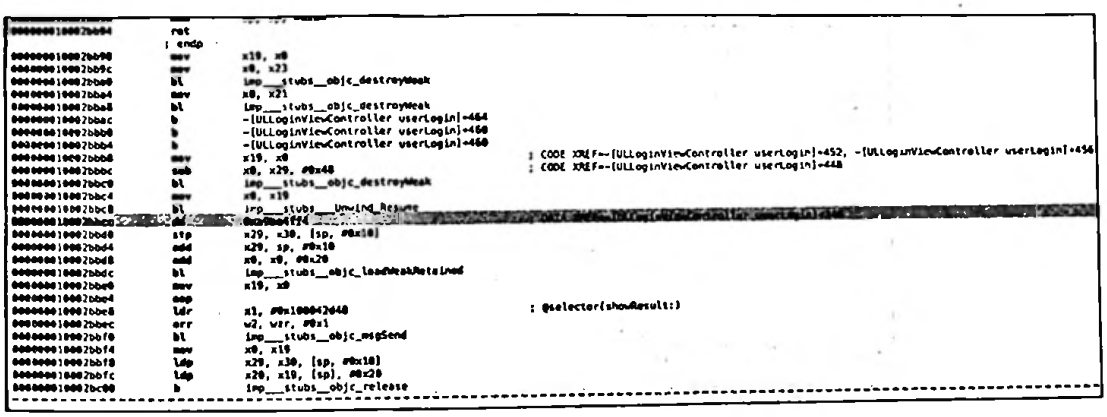


图 5-11　查看 Block 内存函数的位置

在这里，Hopper 并没有将其识别为一个函数，所以需要手动转换一下。首先将 bbcc 处的数据格式转换为代码格式。选中该行，单击工具栏左侧的“C”按钮，转换后如图 5-12 所示。继续选中 bbcc 这一行，单击“P”按钮将其转换成一个新的函数，转换结果如图 5-13 所示，这里便是 Block 内部函数的实现。

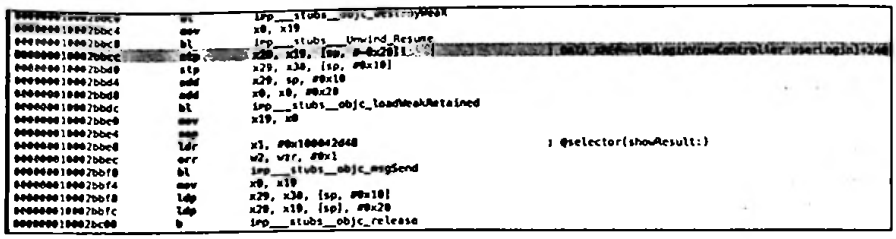


图 5-12 将数据格式转换成代码格式

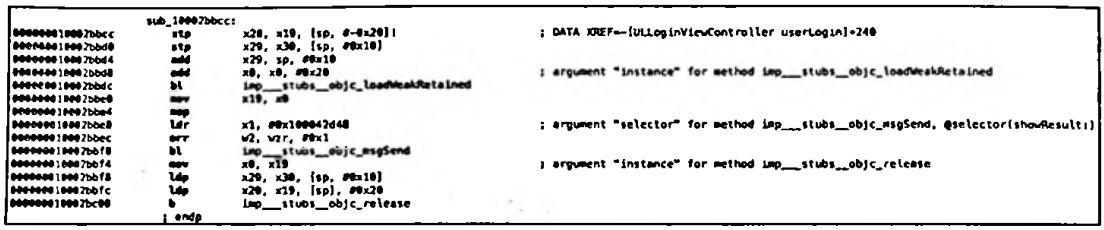


图 5-13 转换成一个新的函数

- 第 4 个参数的分析和第 3 个参数的分析一致。

到这里，这个函数就基本分析完了。除了直接分析汇编代码，也可以查看 Hopper 翻译的伪代码，如图 5-14 所示。但是，Hopper 分析出来的 64 位伪代码并非那么简单易懂。

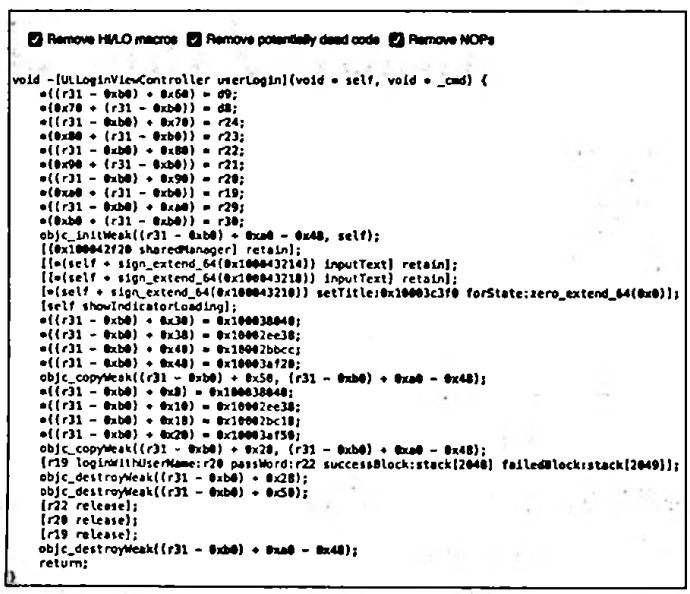


图 5-14 查看程序伪代码

2. Hopper 的常用功能

除了反汇编、伪代码转换，Hopper 还提供了很多功能。下面就对其中几个常用的功能进行讲解。

（1）查看交叉引用

选中函数或者变量，按“X”键，就能显示哪些地方引用了当前函数或者变量，这在分析调用关系时非常有用。如图 5-15 所示，在右侧的面板中也会显示“Is Referenced By”。

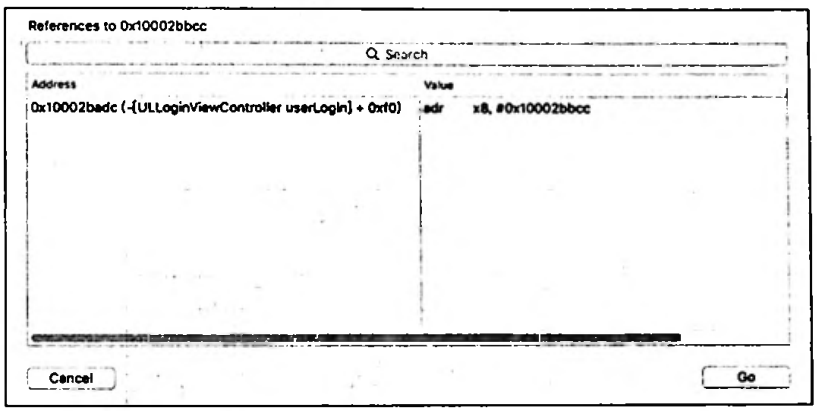


图 5-15　查看交互引用关系

（2）跳转地址

在 Hopper 中直接按“G”键，即可输入并跳转到特定的地址。如图 5-16 所示，在动态分析计算得到地址后，就可以以这种方式直接跳转到目标位置了。但是要注意，在这里可以以选择直接地址或者文件偏移的方式进行跳转，如果选择文件偏移方式，输入“0x6000”的跳转效果和图 5-16 一样。

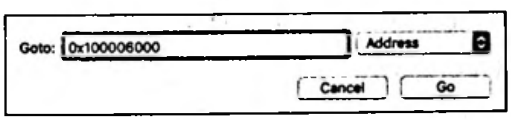


图 5-16　直接跳转到目标地址

（3）修改汇编

Hopper 支持直接修改文件汇编代码。单击要修改的位置，选择菜单项“Modify”→“Assemble Instruction”，就会弹出修改输入窗口。这时可以直接对汇编代码进行修改，如图 5-17 所示。

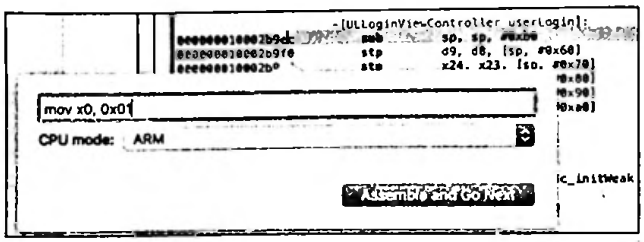


图 5-17 直接修改汇编代码

在这里要注意选择是 ARM 汇编还是 thumb 汇编。尽管这个功能在 iOS 逆向中并不常用，但在 Mac 逆向中会经常用到。除了直接修改汇编，还可以单击工具栏中间的二进制显示格式，对应到指定汇编代码的机器码。例如，在将“jne”修改为“je”时，可以直接双击对应的机器码进行操作，如图 5-18 所示。

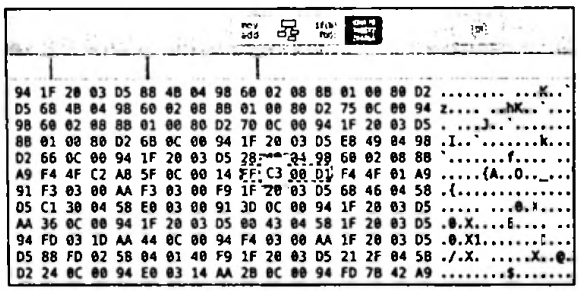


图 5-18 直接修改机器码

5.1.2 IDA

IDA 的反汇编功能比 Hopper 强大，Windows 平台和 Mac 平台都有对应的版本。

1. IDA 的界面

打开 IDA，直接将可执行文件拖入界面，便会弹出如图 5-19 所示的窗口，其中显示了当前文件包含的架构。我们可以在这里选择想要分析的架构。

选择架构之后，IDA 便会对文件进行分析。如果文件不是很大，很快就能分析完。如果文件特别大，可能需要比较长的时间才能分析完。分析完成，会显示如图 5-20 所示的界面，界面结构主要分为如下 4 个部分。

- 上方是 IDA 的菜单和工具栏，包含编辑、搜索、跳转等众多强大的功能。
- 左侧是 IDA 分析出来的函数列表，可以单击或按“Ctrl+F”快捷键进行类或方法的搜索。

- 中间是 IDA 反汇编的结果，代码分析主要在这个区域进行。可以选择不同的 Tab 页来切换不同的功能模块，例如查看反汇编、流程图、伪代码、字符串列表等。
- 下方是 IDA 分析日志。IDA 也支持 Python 脚本获取或者分析结果修改。

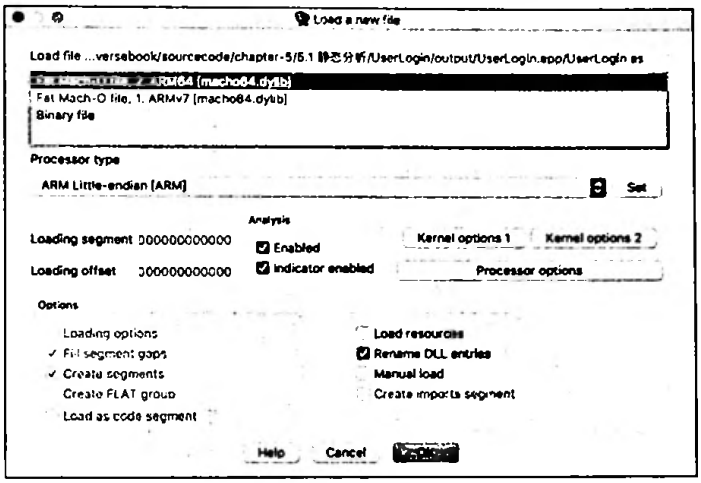


图 5-19　在 IDA 中选择想要分析的架构

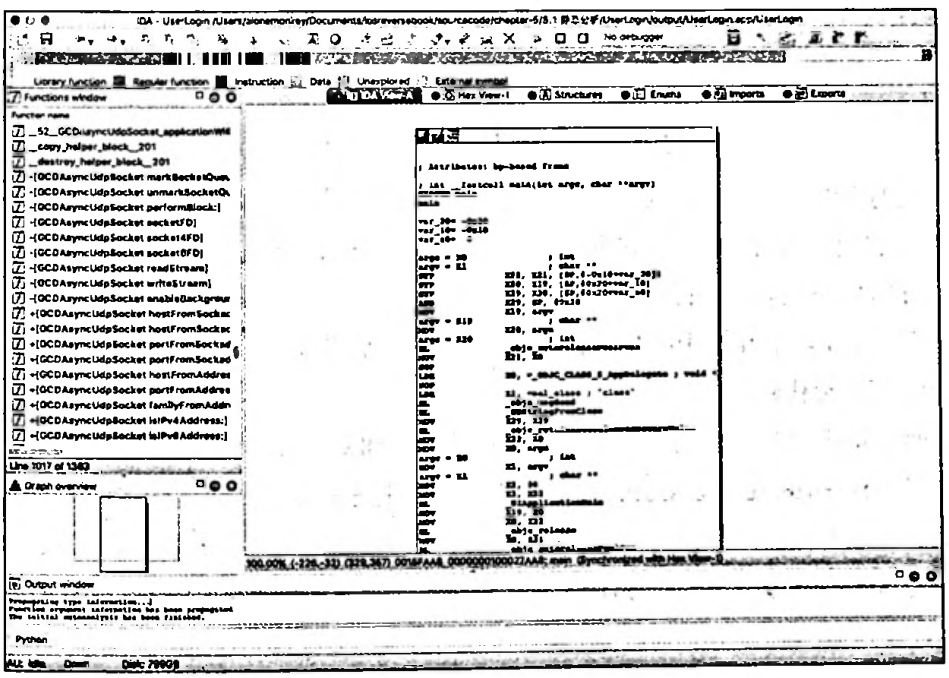


图 5-20　IDA 主界面组成分析

还是以前面分析的函数为例。搜索 userLogin，找到想要的结果并双击，即可跳转到对应的反汇编代码处。默认以流程图的形式显示，可以按“空格”键切换到反汇编文本的形式，如图 5-21 所示。

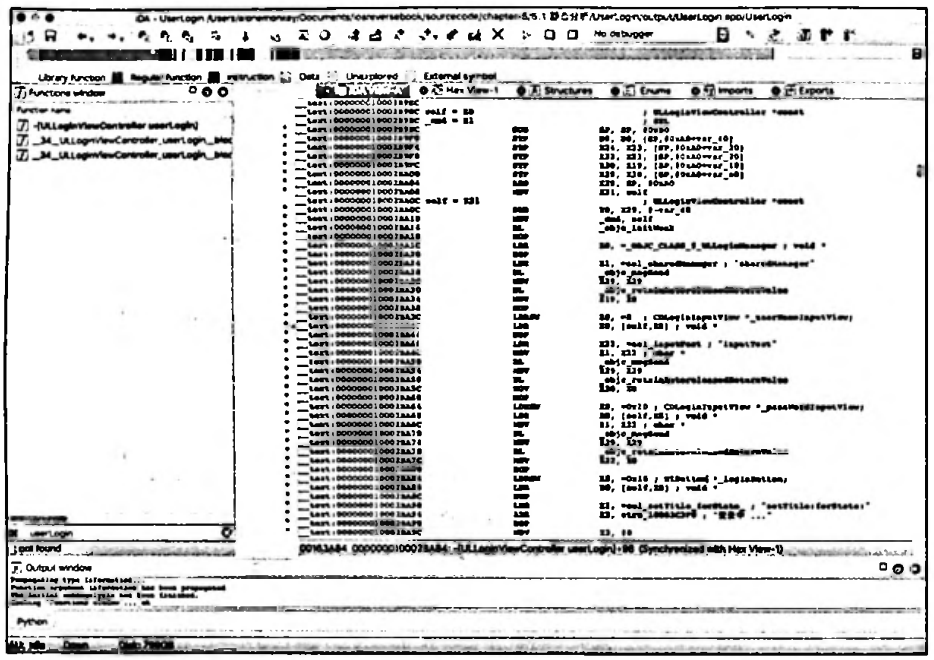


图 5-21　用 IDA 搜索特定的方法并跳转

在右侧的反汇编窗口中可以看到，对应的类名、方法名和属性都已经解析出来了，这比 Hopper 显示得更加清晰，分析起来也更加容易。所以，根据 _objc_msgSend 的调用，直接回溯对应的 x0 和 x1 寄存器，即可得到调用的方法。

再来看看 Block 在 IDA 中的存在形式是什么样的。来到 Block 调用的地方，可以看到如图 5-22 所示的反汇编代码。

同上面的分析一样，第 3 个和第 4 个参数分别是 x4 和 x5。给 x4 赋值的位置在 BACC 处，这里存放的还是 NSConcreteStackBlock 函数的地址。根据上述 Block 的结构，0xC2000000 对应的是 flags 的值，ULLoginViewController_userLoginblock_invoke 便是 Block 的函数指针。双击 ULLoginViewController_userLogin__block_invoke 处，即可跳转到该函数所对应的反汇编代码处，如图 5-23 所示。

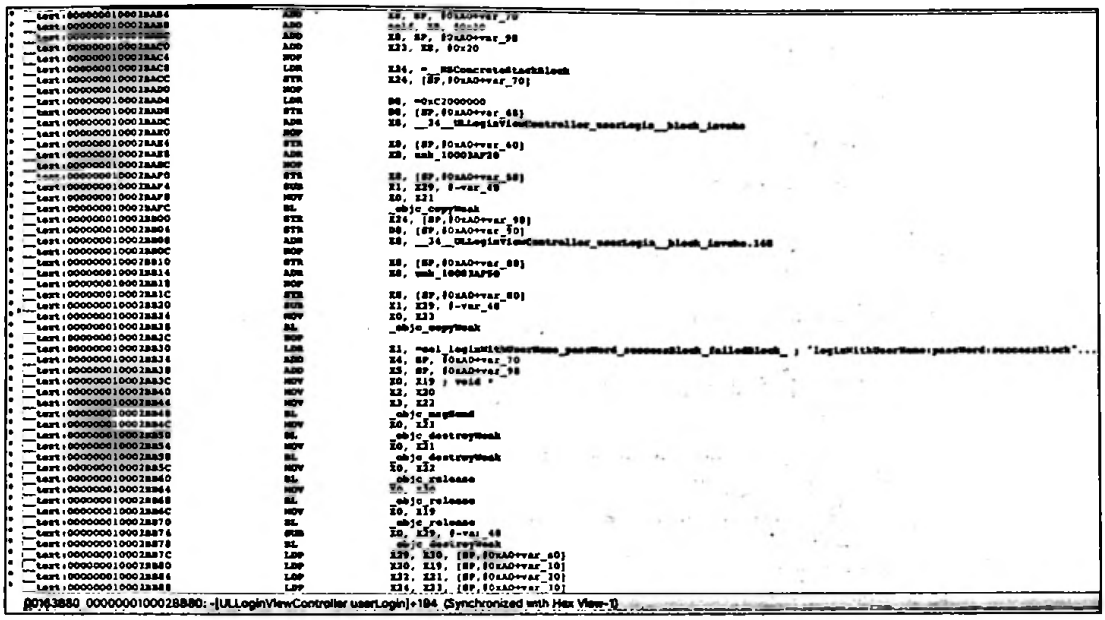


图 5-22　用 IDA 查看 Block 传参反汇编

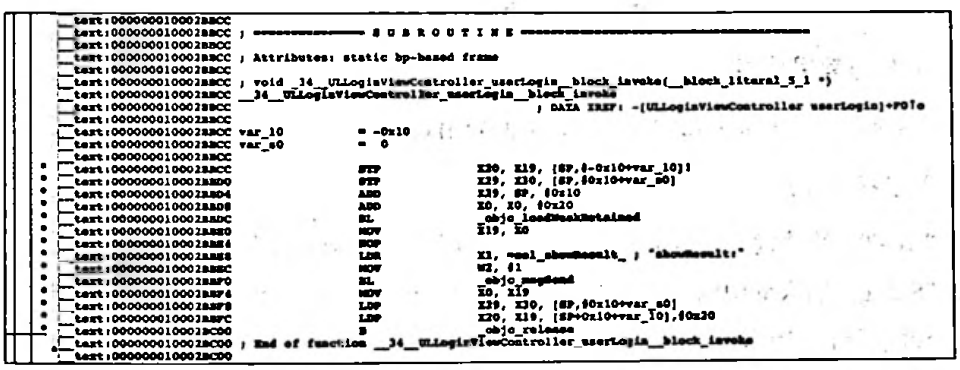
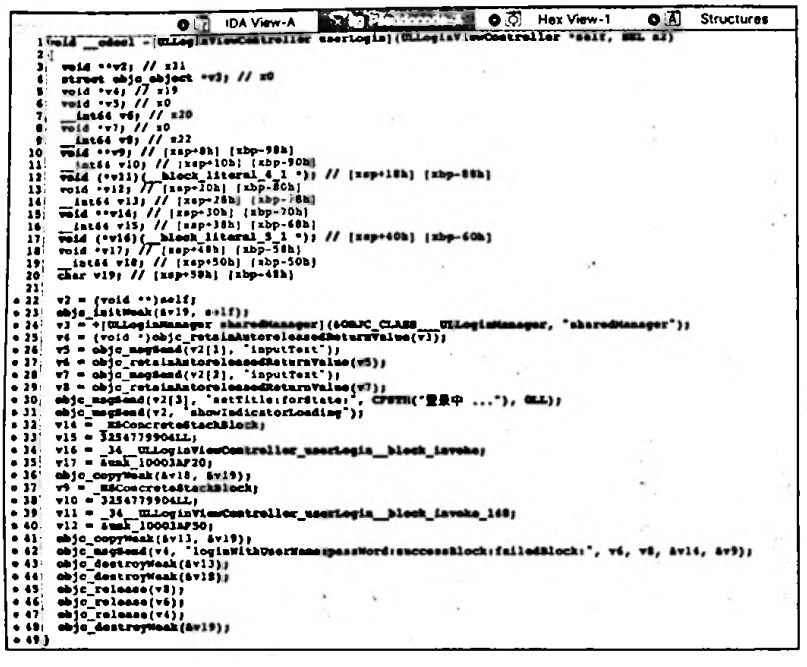


图 5-23　用 IDA 查看 Block 函数地址反汇编

IDA 也提供了显示伪代码的功能，尽管需要购买额外的插件，但显示效果要比 Hopper 好很多。在函数内存处按“F5”键，即可显示如图 5-24 所示的伪代码形式。根据伪代码可以更加容易地回溯寄存器及查看程序逻辑。在分析伪代码时，可以单击对应的伪代码行，按“Tab”键查看对应的反汇编代码，从而将伪代码和反汇编代码结合起来进行静态分析。




```
1 void __cdecl -[UILoginViewController userLogin](UILoginViewController *self, SEL a2)
2 {
3   void **v2; // x21
4   struct objc_object *v3; // x0
5   void *v4; // x19
6   void *v5; // x0
7   __int64 v6; // x20
8   void *v7; // x0
9   __int64 v8; // x22
10  void **v9; // [xsp+8h] [xbp-98h]
11  __int64 v10; // [xsp+10h] [xbp-90h]
12  void (*v11)(__block_literal_6_1 *); // [xsp+18h] [xbp-88h]
13  void **v12; // [xsp+20h] [xbp-80h]
14  __int64 v13; // [xsp+28h] [xbp-78h]
15  void **v14; // [xsp+30h] [xbp-70h]
16  __int64 v15; // [xsp+38h] [xbp-68h]
17  void (*v16)(__block_literal_5_1 *); // [xsp+40h] [xbp-60h]
18  void *v17; // [xsp+48h] [xbp-58h]
19  __int64 v18; // [xsp+50h] [xbp-50h]
20  char v19; // [xsp+58h] [xbp-48h]
21
22  v2 = (void **)self;
23  objc_initWeak(&v19, self);
24  v3 = +[UILoginManager sharedManager](&OBJC_CLASS___UILoginManager, "sharedManager");
25  v4 = (void *)objc_retainAutoreleasedReturnValue(v3);
26  v5 = objc_msgSend(v2[1], "inputText");
27  v6 = objc_retainAutoreleasedReturnValue(v5);
28  v7 = objc_msgSend(v2[2], "inputText");
29  v8 = objc_retainAutoreleasedReturnValue(v7);
30  objc_msgSend(v2[3], "setTitle:forState:", CFSTR("登录中 ..."), 0LL);
31  objc_msgSend(v2, "showIndicatorLoading");
32  v14 = _NSConcreteStackBlock;
33  v15 = 3254779904LL;
34  v16 = __34__UILoginViewController_userLogin__block_invoke;
35  v17 = &unk_10003AF20;
36  objc_copyWeak(&v18, &v19);
37  v9 = _NSConcreteStackBlock;
38  v10 = 3254779904LL;
39  v11 = __34__UILoginViewController_userLogin__block_invoke_168;
40  v12 = &unk_10003AF50;
41  objc_copyWeak(&v13, &v19);
42  objc_msgSend(v4, "loginWithUserName:passWord:successBlock:failedBlock:", v6, v8, &v14, &v9);
43  objc_destroyWeak(&v13);
44  objc_destroyWeak(&v18);
45  objc_release(v8);
46  objc_release(v6);
47  objc_release(v4);
48  objc_destroyWeak(&v19);
49 }
```

图 5-24　用 IDA 查看 32 位伪代码

2. IDA 的常用功能

IDA 本身还提供了很多强大的功能，下面介绍其中一些常用的功能。

（1）字符串搜索

按“Alt+T”快捷键（在 Mac 上是“Option+T”），就会弹出字符串搜索窗口，如图 5-25 所示。在“String”输入框中输入想要搜索的字符串，选择是否区分大小写、正则或者完整匹配，选择是否搜索所有该字符串出现的地方，单击“OK”按钮，即可进行搜索。利用这个功能，我们可以在分析时看到界面上的某个字符串，或者在进行静态分析时看到某个类。若想查看和这个类有关的所有代码，可以通过这种方式进行字符串的搜索。

（2）跳转地址

和 Hopper 一样，在 IDA 中按“G”键就会弹出如图 5-26 所示的跳转框，输入指定的地址即可进行跳转。

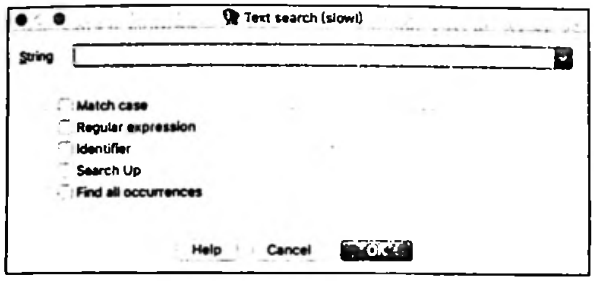


图 5-25　在 IDA 中搜索指定的字符串

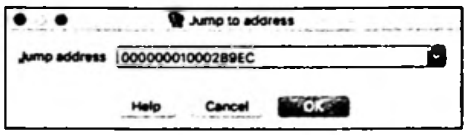


图 5-26　在 IDA 中跳转到指定地址

（3）编写注释

在分析伪代码或者反汇编代码时，经常有笔记需要记录，这时就会用到注释。在反汇编代码显示的地方，可以添加如下两种注释。

- 可重复注释：按“;”键进行注释，该注释会同时出现在其交叉引用的地方。
- 非重复注释：按“:”键进行注释，该注释不会出现在交叉引用的地方。当一处同时存在可重复注释和非重复注释时，只会显示非重复注释，不过按“;”键还是会显示可重复注释。

在伪代码处，可以直接按“/”键输入内容，为代码添加注释。

如图 5-27 所示的注释“this is a block”是笔者在 `ULLoginViewController_userLogin__block_invoke` 处添加的，但在引用它的地方这条注释也显示出来了。

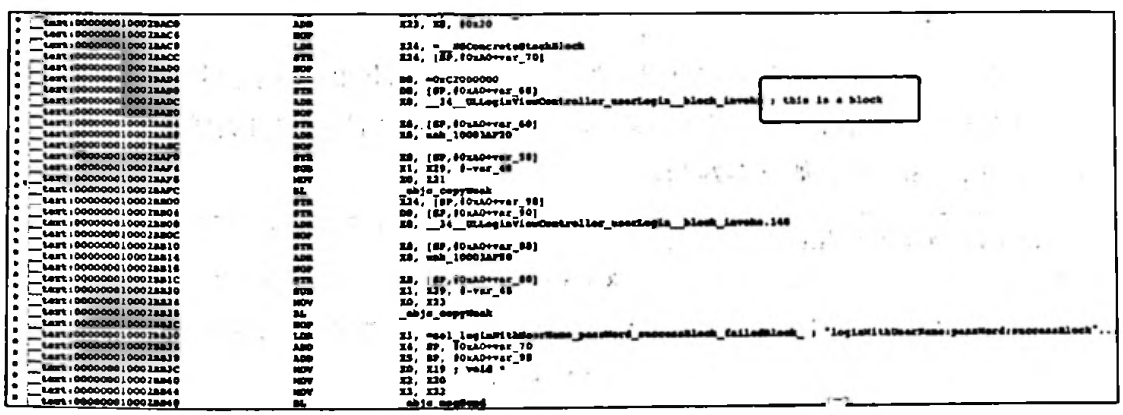


图 5-27　在 IDA 中使用可重复注释

（4）变量重命名

IDA 伪代码中的变量名称都是无意义的，在根据上下文分析出某个变量的含义之后，就可以通过变量重命名的方式来给变量赋予有意义的名字了。单击目标变量，按“N”键，就会弹

出重命名窗口，在其中输入名字并确认，如图 5-28 所示。为变量重命名可以帮助我们基于伪代码快速进行分析。

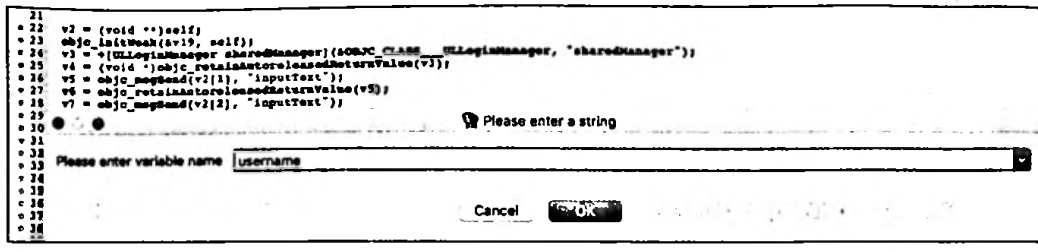


图 5-28　在 IDA 中对变量进行重命名

（5）查看交叉引用

选中函数或者变量，按“X”键，就能显示哪些地方引用了当前函数或者变量。这一点和 Hopper 是一样的。

（6）进制转换

在伪代码分析中会存在一些十进制数字，我们只有将这些数字转换成十六进制才能分析它的意义。单击该数字并按“H”键即可进行进制的转换。读者可以自己尝试将图 5-24 中的 flags 值转换成十六进制值 0xC200000。

（7）类型定义

在分析时，如果变量的类型或者方法的类型（例如 IDA 分析出来的方法的参数个数和类型）不是人工分析出来的结果，可以单击目标方法或者变量，按“Y”键打开类型定义窗口，输入正常的参数个数和类型，如图 5-29 所示。

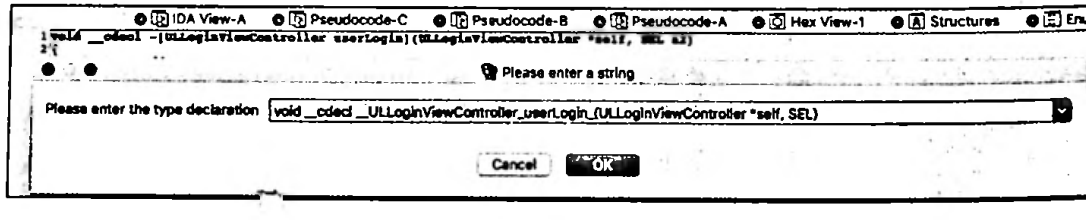


图 5-29　IDA 对方法类型重定义

（8）格式转换

IDA 也提供将数据解析成代码和将机器码解析成数据的功能。在被解析成数据的代码处按“C”键，即可将其还原成代码；在未被解析成数据的地方按“D”键，即可将机器码或者其他

内容转换成数据。

（9）显示机器码

在默认情况下，IDA 显示的反汇编代码的前面是没有显示机器码的。如果想显示机器码，需要选择“Options”→“General”→“Disassembly”选项进行设置。如图 5-30 所示，设置显示 10 个机器码的信息。

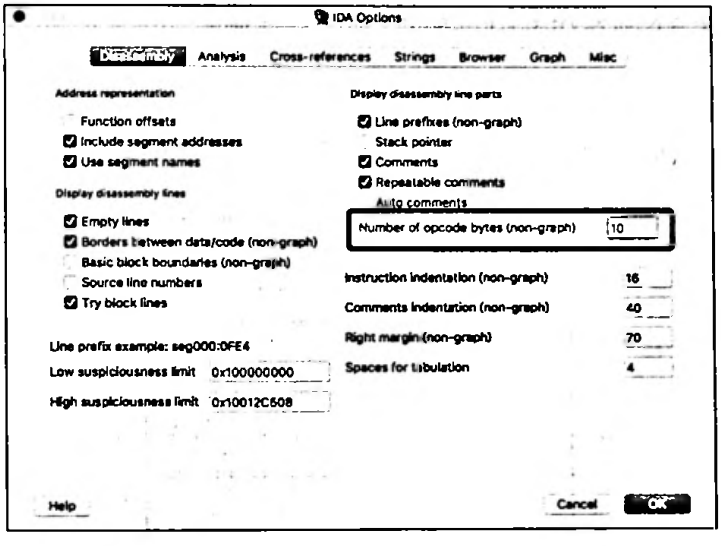


图 5-30　IDA 反汇编代码显示机器码信息

IDA 还有很多强大的功能，例如通过脚本自动分析当前的静态代码、使用 IDA 插件增强 IDA 的功能（例如 keypatch，https://github.com/keystone-engine/keypatch），这些就留给读者自己慢慢探索了。

5.1.3　静态库分析

除了使用 Hopper 和 IDA 对 App 进行分析，有时还需要对静态库进行单独的分析。下面将介绍对静态库进行分析的要点和技巧。

一般的静态库文件里面会包含模拟器和真机的架构。以 Crashlytics 的静态库为例，可以通过如下命令来查看静态库文件包含的架构信息。

```
lipo -info Crashlytics
Architectures in the fat file: Crashlytics are: armv7 armv7s i386 x86_64 arm64
```

在分析时，只需要分析其中的一个架构。在这里先将 FAT 文件“瘦身”，然后获取其中 ARM64 的架构，代码如下。

```
lipo Crashlytics -thin arm64 -output Crashlytics_arm64
```

其实，静态库就相当于一个压缩包，里面包含了静态库中编译的所有 OBJECT 文件。为了方便分析，可以通过 ar 命令将静态库中的所有 OBJECT 文件解压出来，代码如下。解压后就能看到静态库里面包含的所有 OBJECT 文件了，如图 5-31 所示。

```
mkdir Objects
cd Objects
ar -x ../Crashlytics_arm64
```

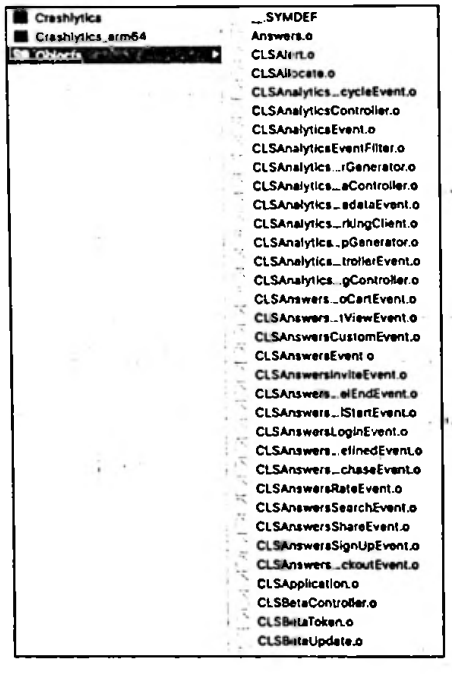


图 5-31 使用 ar 命令解压获取 OBJECT 文件

解压之后就可以对 OBJECT 文件进行分析了。在逆向静态库的过程中，当从静态库的初始化函数开始分析时，若其中调用了其他函数，但不知道这个函数是在哪里实现或者出现的，可以通过对所有 OBJECT 文件进行字符串搜索来查看某个字符串在哪些 OBJECT 文件中出现，从而确定需要分析的 OBJECT 文件。通过如下命令可以查找所有出现“Upload”字符的 OBJECT 文件。

```
➜  Objects git:(master) X grep "Upload" -rn ./
Binary file .//CLSCrashReportingController.o matches
Binary file .//CLSNetworkClient.o matches
Binary file .//CLSReportsController.o matches
```

为了方便分析，还可以将所有 OBJECT 文件链接成一个 OBJECT 文件。首先，通过如下命令判断 OBJECT 文件是否包含 bitcode，如果不包含 bitcode，将不会有内容输出。

```
➜  Objects git:(master) X otool -l Crashlytics.o | grep bitcode
  sectname __bitcode
```

然后，执行 ld 命令，将所有 OBJECT 文件链接成一个 OBJECT 文件，具体如下。

```
ld -r -arch arm64 -syslibroot
/Applications/Xcode-beta.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/S
DKs/iPhoneOS.sdk -bitcode_bundle ./*.o -o ../output
```

若包含 bitcode，则添加参数 -bitcode_bundle；若不包含，则不用添加。执行之后，会在上级目录中看到一个 output 文件。把该文件直接拖到 Hopper 插件中，得到单个的 OBJECT Mach-O 文件，如图 5-32 所示。接下来，就可以直接对该 OBJECT 文件进行分析，而不用考虑之前的一大堆 OBJECT 文件了。

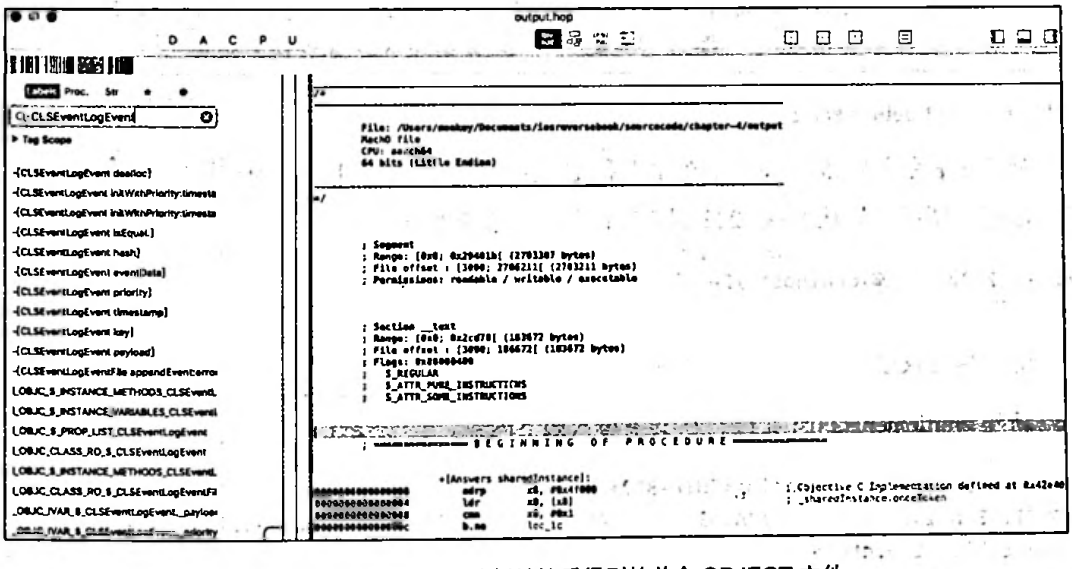


图 5-32　使用 Hopper 分析链接后得到的单个 OBJECT 文件

5.2 动态调试

动态调试与静态分析是相辅相成的。静态分析只能分析静态的函数内部执行。要想动态获取程序在运行时的参数传递、执行流程及寄存器内存等信息，就需要使用动态调试的方法。动态调试可以让我们对程序运行的整个流程有更加清晰的认识。在逆向分析过程中，动态调试也能够帮助我们分析程序的某个行为调用的方法及调用的上下文。本节将从如何通过 LLDB 命令行调试应用及如何直接使用 Xcode 调试第三方应用两方面来讲解。

5.2.1 LLDB 调试

LLDB 是 Xcode 自带的调试工具，既可以在本地调试 Mac 应用程序，也可以远程调试 iPhone 应用程序。当使用 Xcode 调试手机 App 时，Xcode 会将 debugserver 文件复制到手机中，以便在手机上启动一个服务，等待 Xcode 进行远程连接调试。所以，只有当设备连接计算机真机调试 App 后，debugserver 文件才会安装到设备的 /Developer/usr/bin 目录下。但是，debugserver 文件默认只能调试自己开发的应用，在调试从 App Store 下载的应用时会出现“unable to start the exception thread”错误。

1. 准备工作

为了调试其他应用，需要给 debugserver 文件赋予 task_for_pid 权限。在调试前需要做如下准备工作。

01 复制 debugserver 文件

使用 scp 命令将 debugserver 文件从手机复制到 Mac 计算机上。因为这里使用 iproxy 进行了端口映射，所以直接从本地 2222 端口进行复制，命令如下。

```
scp -P 2222 root@localhost:/Developer/usr/bin/debugserver ./
```

02 签名权限

新建 entitlements.plist 文件，在其中写入如下内容。

```
<?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.springboard.debugapplications</key> <true/>
    <key>run-unsigned-code</key>
    <true/>
    <key>get-task-allow</key>
    <true/>
    <key>task_for_pid-allow</key>
    <true/>
</dict>
</plist>
```

使用 codesign 进行签名，命令如下。

```
codesign -s - --entitlements entitlements.plist -f debugserver
```

03　将 debugserver 文件复制回手机

因为 /Developer/ 是只读的，所以需要执行如下命令，将签好权限的 debugserver 文件复制到手机的 bin 目录下。

```
scp -P 2222 ./debugserver root@localhost:/usr/bin/debugserver
```

2. 进行调试

准备好有 task_for_pid 权限的 debugserver 文件后，就能够对从 AppStore 下载的应用进行调试了。debugserver 支持通过进程名和进程 ID 进行调试，下面以 App Store 上的应用 Snapchat 为例进行讲解。因为 Snapchat 的最新版本已经不支持 iOS 8 了，所以笔者使用版本较低的 Snapchat 10.3.0.ipa。使用 iTools 安装并打开应用，运行 ps aux 命令，找到启动的进程，具体如下。

```
mobile     731   1.6  9.0   781340  91972   ??  Us   10:33PM   0:08.91
/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/Snapchat
.app/Snapchat
```

找到目标进程后，就可以通过进程名进行调试了，命令如下。

```
debugserver *:1234 -a Snapchat
```

也可以通过进程 ID 进行调试，命令如下。

```
Monkey:~ root# debugserver *:1234 -a 731
debugserver-@(#)PROGRAM:debugserver  PROJECT:debugserver-320.2.89
 for arm64.
```

```
Attaching to process 731...
Listening to port 1234 for a connection from *...
```

命令执行之后，进入等待连接的状态，被调试的进程将会卡住。如果调试的是 SpringBoard，那么整个桌面都会卡住，这是正常的情况。接下来，需要通过 Mac 远程连接目标的监听服务。在 Mac 计算机上打开终端，输入“lldb”并按“Enter”键，然后输入如下命令，连接远程 debugserver 启动的服务。在这里，需要将“192.168.2.202”替换为你的手机的 IP 地址（手机的 IP 地址可以在手机的 Wi-Fi 设置页面获取）。因为这里是通过远程 IP 地址连接的，所以建立连接的过程会比较慢，请耐心等待。连接建立后，会显示如下信息提示。通过 Wi-Fi 连接的速度大都比较慢，因此，笔者推荐先使用 iproxy 做端口转发（命令为 `iproxy 1234 1234`），再通过 `process connect connect://localhost:1234` 命令进行连接。

```
➜  ~ lldb
(lldb) process connect connect://192.168.2.202:1234
Process 731 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x0000000192b28e7c libsystem_kernel.dylib`mach_msg_trap + 8
libsystem_kernel.dylib`mach_msg_trap:
->  0x192b28e7c <+8>: ret

libsystem_kernel.dylib`mach_msg_overwrite_trap:
    0x192b28e80 <+0>: mov    x16, #-0x20
    0x192b28e84 <+4>: svc    #0x80
    0x192b28e88 <+8>: ret
Target 0: (Snapchat) stopped.
(lldb)
```

此时，目标进程还是无法响应事件。输入“c”，然后按“Enter”键，才能让程序继续运行并响应外部事件，具体如下。

```
(lldb) c
Process 731 resuming
```

在程序运行过程中，可以随时按“Control + C”快捷键来暂停程序，输入命令，查看程序内存中的信息，相关代码如下。如果只需要获取指定模块加载的基地址，就在后面加上模块名。

```
Process 731 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x0000000192b28e7c libsystem_kernel.dylib`mach_msg_trap + 8
```

```
libsystem_kernel.dylib`mach_msg_trap:
->  0x192b28e7c <+8>: ret

libsystem_kernel.dylib`mach_msg_overwrite_trap:
    0x192b28e80 <+0>: mov    x16, #-0x20
    0x192b28e84 <+4>: svc    #0x80
    0x192b28e88 <+8>: ret
Target 0: (Snapchat) stopped.
(lldb)
(lldb) image list -o -f
[  0] 0x0000000000038000
/private/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/
Snapchat.app/Snapchat(0x0000000100038000)
[  1] 0x000000000004c000 /usr/lib/dyld(0x000000012004c000)
[  2] 0x00000001058b0000
/Library/MobileSubstrate/MobileSubstrate.dylib(0x00000001058b0000)
[  3] 0x0000000000000000 /usr/lib/libc++.1.dylib(0x0000000191b48000)
[  4] 0x0000000000000000 /usr/lib/libiconv.2.dylib(0x0000000191c74000)
[  5] 0x0000000000000000 /usr/lib/libsqlite3.dylib(0x00000001926e8000)
......

(lldb) image list -o -f "Snapchat"
[  0] 0x0000000000038000
/private/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/
Snapchat.app/Snapchat(0x0000000100038000)
```

对内存信息每一列的解释如下。

- 第 1 列是加载的模块的序号。
- 第 2 列是目标模块在内存中加载的基地址。
- 第 3 列是加载模块的全路径和模块真正加载开始的地址（加载的起始地址 + 模块在虚拟地址的内存偏移）。

知道了程序在内存中加载的起始地址，就可以通过该地址和文件中的偏移地址对某个方法下断点了。将解密后的 Snapchat 文件拖入 Hopper 或 IDA 中查看。注意选择当前设备的架构。笔者的设备是 iPhone 5s，运行的是 ARM64，解密后也是 ARM64，所以要选择 ARM64 架构进行分析。如果是 32 位的设备（例如 iPhone 5c），就要选择 ARMv7 架构。

在进行动态调试之前，一般都获得了一些分析结论。结合动态分析来确定分析的正确性，才更容易获取真正运行的调用。分析 Snapchat 的登录界面，可以看到，点击“登录”按钮时调用的函数为 `+[Manager performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirm`

Reactivation:rememberDevice:fromDeepLink:onComplete:]。在这里先不讨论分析的细节。也可以通过 Cycript，找到登录界面的 ViewController 为 SCLoginV2View Controller，然后找到该类的 performLogin 方法。反编译该方法，就可以知道调用了 Manager 的哪个函数（在后面的章节中会用具体的案例来讲解相关的分析技巧）。

找到对应的方法后，就需要对该方法下断点进行调试了。通过 Hopper 或 IDA，可以分析出该方法对应的地址为 0x000000010095ECCC，如图 5-33 所示。将该地址加上模块加载的基地址，计算得到内存中的真实地址，代码如下。

```
(lldb) p/x 0x000000010095ECCC+0x0000000000038000
(long) $0 = 0x0000000100996ccc
```

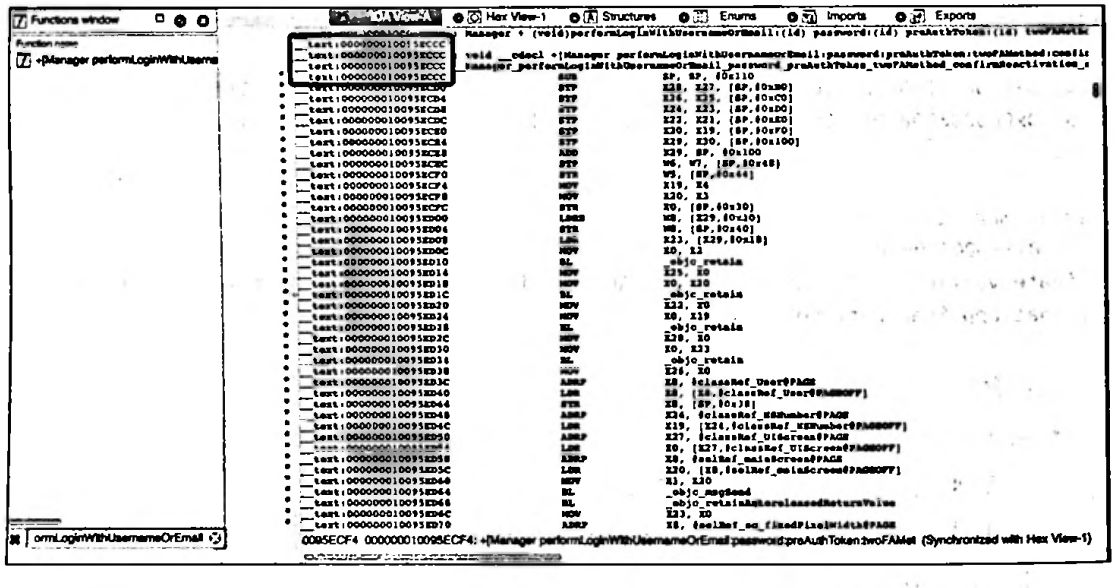


图 5-33 根据 IDA 找到对应函数的地址

直接对该地址使用如下方法下断点。下完断点，先按“C”键，再按“Enter”键，让程序继续运行。输入任意用户名和密码，点击“登录”按钮，如图 5-34 所示。

```
b 0x0000000100996ccc
或者
br s -a 0x000000010095ECCC+0x0000000000038000

(lldb) b 0x0000000100996ccc
Breakpoint 1: where = Snapchat`void
```

```
std::__1::atomic_store<std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener>
__weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >(std::__1::shar
ed_ptr<std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener> __weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >*,
std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener> __weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > >) + 314848, address
= 0x0000000100996ccc
(lldb) c
Process 731 resuming
(lldb)
```

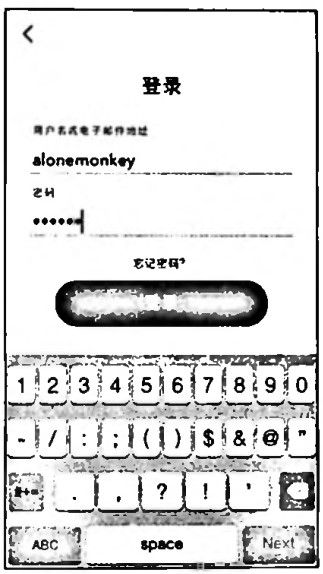


图 5-34　输入用户名和密码并点击登录按钮

登录程序后，便会触发断点，使程序暂停。这时就可以通过寄存器或者内存来分析程序运行过程中的参数传递、函数调用等信息了，相关代码如下。

```
Process 731 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100996ccc Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener>
__weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >(std::__1::shar
ed_ptr<std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener> __weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >*,
```

```
std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener> __weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > >) + 314848
Snapchat`std::__1::atomic_store<std::__1::vector<id<SCImageProcessVideoPlaybackSessionL
istener> __weak, std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >:
->  0x100996ccc <+314848>: sub    sp, sp, #0x110            ; =0x110
    0x100996cd0 <+314852>: stp    x28, x27, [sp, #0xb0]
    0x100996cd4 <+314856>: stp    x26, x25, [sp, #0xc0]
    0x100996cd8 <+314860>: stp    x24, x23, [sp, #0xd0]
Target 0: (Snapchat) stopped.
(lldb)
```

因为该断点处的函数是一个 OC 函数，所以可以通过 x0 寄存器获取调用的类，通过 x1 寄存器获取调用的方法，通过其他寄存器获取调用的参数。在 LLDB 命令交互模式下分别打印输出它们的值，具体如下。

```
(lldb) po $x0
Manager
(lldb) po [$x0 class]
Manager
(lldb) x/s $x1
0x1030d6e46:
"performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:
rememberDevice:fromDeepLink:onComplete:"
(lldb) po $x2
alonemonkey
(lldb) po $x3
123456
(lldb) po $x4
<object returned empty description>
(lldb) p $x5
(unsigned long) $9 = 0
(lldb) p $x6
(unsigned long) $10 = 0
(lldb) p $x7
(unsigned long) $11 = 1
```

“po”是“print object”的缩写，用于打印 OC 对象。在这里直接打印了该 OC 类的类名。如果程序重载了 description 方法，这里打印的就是 description 返回的值。在这种情况下，若想获取 OC 的类名，可以执行 `po [$x0 class]` 命令来打印类名。第 2 个参数是一个方法名，它是一个字符串，所以可以通过 `x/s $x1` 命令将寄存器 x1 处的内存以字符串的形式显示出来，其对应的就是调用的方法名。后面的参数类型可以通过 class-dump 导出头文件里面的类型进行判断。

当前函数在 class-dump 头文件中的形式如下。

```
+ (void)performLoginWithUsernameOrEmail:(id)arg1 password:(id)arg2 preAuthToken:(id)arg3
twoFAMethod:(int)arg4 confirmReactivation:(_Bool)arg5 rememberDevice:(_Bool)arg6
fromDeepLink:(_Bool)arg7 onComplete:(CDUnknownBlockType)arg8;
```

根据 dump 出来的头文件中的类型，可以判断前 3 个参数都是 id 类型，并可以通过 po 命令直接打印。上面打印出来第 1 个参数是 alonemonkey，第 2 个参数是 123456，分别对应于笔者输入的用户名和密码；第 3 个参数是 <object returned empty description>，表示该参数是一个 nil 空值；第 4 个参数是 int 类型，值为 0；第 5 个参数是 BOOL 类型，值为 NO；第 6 个参数是 BOOL 类型，值为 YES。需要注意的是，在 ARM64 中，参数只会通过 x0 ~ x7 寄存器传递，如果参数的个数多于可用于传递的寄存器的个数，多余的参数就只能通过栈传递了。

对最后两个参数，该如何获取它们所对应的值呢？因为其他参数是通过栈传递的，所以可以使用如下命令，通过打印当前调用的函数栈获取这两个参数。sp 及 fp 的含义会在 6.3 节详细讲解，该命令的作用就是打印内存中的栈的值。可以看到，前面两个值都是 0，也就是说，第 7 个和第 8 个参数都没有值。读者可以自己尝试编写一个简单的多参数程序，通过这种方式去了解多余的参数是如何传递的。

```
(lldb) memory read -force -f A $sp $fp
0x16fdc5570: 0x0000000000000000
0x16fdc5578: 0x0000000000000000
0x16fdc5580: 0x0000000000000001
0x16fdc5588: 0x0000000174621ee0
0x16fdc5590: 0x0000000124630a10
0x16fdc5598: 0x00000001030b2f37 "continueButtonClicked"
0x16fdc55a0: 0x0000000124646b30
0x16fdc55a8: 0x0000000000000000
```

通过动态调试还可以获取程序调用的堆栈信息。在 LLDB 命令交互模式下输入“bt”并按“Enter”键，可以看到如下结果。

```
(lldb) bt
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
  * frame #0: 0x0000000100996ccc Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener>
__weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >(std::__1::shar
ed_ptr<std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener> __weak,
```

```
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > > >*,
std::__1::vector<id<SCImageProcessVideoPlaybackSessionListener> __weak,
std::__1::allocator<id<SCImageProcessVideoPlaybackSessionListener> > >) + 314848
    frame #1: 0x0000000100e595b4 Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCMergedGalleryDataSourceListener> __weak,
std::__1::allocator<id<SCMergedGalleryDataSourceListener> > > >(std::__1::shared_ptr<st
d::__1::vector<id<SCMergedGalleryDataSourceListener> __weak,
std::__1::allocator<id<SCMergedGalleryDataSourceListener> > > >*,
std::__1::vector<id<SCMergedGalleryDataSourceListener> __weak,
std::__1::allocator<id<SCMergedGalleryDataSourceListener> > >) + 101676
    frame #2: 0x0000000100d4db0c Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCGalleryLagunaWifiControllerEventListener>
__weak,
std::__1::allocator<id<SCGalleryLagunaWifiControllerEventListener> > > >(std::__1::shar
ed_ptr<std::__1::vector<id<SCGalleryLagunaWifiControllerEventListener> __weak,
std::__1::allocator<id<SCGalleryLagunaWifiControllerEventListener> > > >*,
std::__1::vector<id<SCGalleryLagunaWifiControllerEventListener> __weak,
std::__1::allocator<id<SCGalleryLagunaWifiControllerEventListener> > >) + 463496
    frame #3: 0x0000000186498d34 UIKit`<redacted> + 96
    frame #4: 0x0000000186481e48 UIKit`<redacted> + 612
    frame #5: 0x00000001864986d0 UIKit`<redacted> + 592
    frame #6: 0x000000018649835c UIKit`<redacted> + 700
    frame #7: 0x00000001864918b0 UIKit`<redacted> + 684
    frame #8: 0x0000000100d64ba4 Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCGalleryLagunaWifiControllerEventListener>
__weak,
std::__1::allocator<id<SCGalleryLagunaWifiControllerEventListener> > > >(std::__1::shar
ed_ptr<std::__1::vector<id<SCGalleryLagunaWifiControllerEventListener> __weak,
std::__1::allocator<id<SCGalleryLagunaWifiControllerEventListener> > > >*,
std::__1::vector<id<SCGalleryLagunaWifiControllerEventListener> __weak,
std::__1::allocator<id<SCGalleryLagunaWifiControllerEventListener> > >) + 557856
    frame #9: 0x0000000186464fa8 UIKit`<redacted> + 264
    frame #10: 0x0000000186703f58 UIKit`<redacted> + 14992
    frame #11: 0x0000000186463510 UIKit`<redacted> + 1616
    frame #12: 0x0000000181c6a9ec CoreFoundation`<redacted> + 24
    frame #13: 0x0000000181c69c90 CoreFoundation`<redacted> + 264
    frame #14: 0x0000000181c67d40 CoreFoundation`<redacted> + 712
    frame #15: 0x0000000181b950a4 CoreFoundation`CFRunLoopRunSpecific + 396
    frame #16: 0x000000018ad3f5a4 GraphicsServices`GSEventRunModal + 168
    frame #17: 0x00000001864ca3c0 UIKit`UIApplicationMain + 1488
    frame #18: 0x0000000100a175c0 Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCChatViewLifeCycleListener> __weak,
std::__1::allocator<id<SCChatViewLifeCycleListener> > > >(std::__1::shared_ptr<std::__1
::vector<id<SCChatViewLifeCycleListener> __weak,
```

```
std::__1::allocator<id<SCChatViewLifeCycleListener> > > >*,
std::__1::vector<id<SCChatViewLifeCycleListener> __weak,
std::__1::allocator<id<SCChatViewLifeCycleListener> > >) + 5092
    frame #19: 0x0000000192a2aa08 libdyld.dylib`<redacted> + 4
(lldb)
```

通过调用堆栈可以看到，frame #1 对应于地址 0x0000000100e595b4。用内存地址减去加载的基地址，即可得到文件中对应的偏移地址，相关代码如下。

```
(lldb) p/x 0x0000000100e595b4-0x0000000000038000
(long) $18 = 0x0000000100e215b4
或者使用
 (lldb) image lookup -a 0x0000000100e595b4
      Address: Snapchat[0x0000000100e215b4] (Snapchat.__TEXT.__text + 14790372)
      Summary: Snapchat`void
std::__1::atomic_store<std::__1::vector<id<SCMergedGalleryDataSourceListener> __weak,
std::__1::allocator<id<SCMergedGalleryDataSourceListener> > > >(std::__1::shared_ptr<st
d::__1::vector<id<SCMergedGalleryDataSourceListener> __weak,
std::__1::allocator<id<SCMergedGalleryDataSourceListener> > > >*,
std::__1::vector<id<SCMergedGalleryDataSourceListener> __weak,
std::__1::allocator<id<SCMergedGalleryDataSourceListener> > >) + 101676
(lldb)
```

在 IDA 中按“G”键跳转到该地址，到该地址是从哪个函数调用的。通过这种将堆栈地址转换成文件偏移的方式，可以快速恢复当前的调用堆栈，进而找到当前函数的上级调用函数，这对分析一个函数是在哪里调用的及跟踪函数的底层实现都是非常有用的（具体的应用会在第 7 章讲解，这里只做了解）。

除了上面提到的这些指令，LLDB 还提供了很多指令操作，列举如下。

- register read：读取所有寄存器的值。
- register read $x0：读取某个寄存器的值。
- register write $x5 1：修改某个寄存器的值。
- si：跳到当前指令的内部。
- ni：跳过当前指令。
- finish：返回上层调用栈。
- thread return：不再执行下面的代码，直接从当前调用栈返回一个值。
- br list：查看当前断点列表。
- br del：删除当前的所有断点。

- br del 1.1.1：删除指定编号的断点。
- br dis 2.1：使断点 2.1 失效。
- br enable 2.1：使断点 2.1 生效。
- watchpoint set expression -w write -- 0x101801a48：给某个地址设置观察断点，当对该地址的内存进行写操作时就会触发断点。
- x/10xg 0x101801a48：读取目标地址的内存指令。这里的“x”代表用十六进制来显示结果，“g”代表 giant word（8 字节）大小。所以，“x/10xg”就是用十六进制显示 0x101801a48 所指空间的 10 个 64 位的元素内容。常见的大小格式为“b - byte”（1 字节）“h - half word”（2 字节）“w - word”（4 字节）“g - giant word”（8 字节）。
- dis -a $pc：反汇编指定地址。这里是 pc 寄存器所对应的地址。
- f 2：切换到当前调用栈为 2 的位置，也就是 bt 中的 frame #2。
- thread info：输出当前线程的信息。
- b ptrace -c xxx：满足某个条件之后程序才会中断。

LLDB 还有很多命令，我们不可能全都记下来，但是要记住其中的两个查找命令，即 help 和 apropos。直接在 LLDB 命令交互模式下输入“help”，可以查看所有 LLDB 命令。如果想查看某个命令的详细用法，可以使用“help 命令名”的形式。查看 thread 命令的用法，具体如下。

```
(lldb) help thread
    Commands for operating on one or more threads in the current process.

Syntax: thread

The following subcommands are supported:

     backtrace      -- Show thread call stacks.  Defaults to the current thread, thread
indexes can be
                       specified as arguments.  Use the thread-index "all" to see all threads.
     continue       -- Continue execution of the current target process.  One or more threads
may be
                       specified, by default all threads continue.
     info           -- Show an extended summary of one or more threads.  Defaults to the
current
                       thread.
     jump           -- Sets the program counter to a new address.
     list           -- Show a summary of each thread in the current target process.
     plan           -- Commands for managing thread plans that control execution.
     return         -- Prematurely return from a stack frame, short-circuiting execution of
```

```
newer
                    frames and optionally yielding a specified value.  Defaults to the
exiting the
                    current stack frame.  Expects 'raw' input (see 'help raw-input'.)
     select         -- Change the currently selected thread.
     step-in        -- Source level single step, stepping into calls.  Defaults to current
thread
                    unless specified.
     step-inst      -- Instruction level single step, stepping into calls.  Defaults to
current thread
                    unless specified.
     step-inst-over -- Instruction level single step, stepping over calls.  Defaults to
current thread
                    unless specified.
     step-out       -- Finish executing the current stack frame and stop after returning.
Defaults to
                    current thread unless specified.
     step-over      -- Source level single step, stepping over calls.  Defaults to current
thread
                    unless specified.
     step-scripted  -- Step as instructed by the script class passed in the -C option.
     until          -- Continue until a line number or address is reached by the current or
specified
                    thread.  Stops when returning from the current function as a safety
measure.
                    The target line number(s) are given as arguments, and if more than one
is
                    provided, stepping will stop when the first one is hit.

For more help on any particular subcommand, type 'help <command> <subcommand>'.
```

如果没有记住某个命令，只记得其中的关键字，或者想根据关键字去查找相关的 LLDB 命令，可以使用 apropos 来搜索相关的命令信息。搜索“watch”的相关信息，具体如下。

```
(lldb) apropos watch
The following commands may relate to 'watch':
  watchpoint               -- Commands for operating on watchpoints.
  watchpoint command       -- Commands for adding, removing and examining LLDB commands
executed when
                              the watchpoint is hit (watchpoint 'commmands').
  watchpoint command add   -- Add a set of LLDB commands to a watchpoint, to be executed
whenever the
                              watchpoint is hit.
  watchpoint command delete -- Delete the set of commands from a watchpoint.
```

```
  watchpoint command list   -- List the script or set of commands to be executed when the
watchpoint is
                            hit.
  watchpoint delete         -- Delete the specified watchpoint(s).  If no watchpoints are
specified,
                            delete them all.
  watchpoint disable        -- Disable the specified watchpoint(s) without removing it/them.
If no
                            watchpoints are specified, disable them all.
  watchpoint enable         -- Enable the specified disabled watchpoint(s). If no watchpoints
are
                            specified, enable all of them.
  watchpoint ignore         -- Set ignore count on the specified watchpoint(s).  If no
watchpoints are
                            specified, set them all.
  watchpoint list           -- List all watchpoints at configurable levels of detail.
  watchpoint modify         -- Modify the options on a watchpoint or set of watchpoints in
the
                            executable.  If no watchpoint is specified, act on the last
created
                            watchpoint.  Passing an empty argument clears the modification.
  watchpoint set            -- Commands for setting a watchpoint.
  watchpoint set expression -- Set a watchpoint on an address by supplying an expression.
Use the '-w'
                            option to specify the type of watchpoint and the '-s' option
to specify
                            the byte size to watch for. If no '-w' option is specified, it
defaults
                            to write. If no '-s' option is specified, it defaults to the
target's
                            pointer byte size. Note that there are limited hardware
resources for
                            watchpoints. If watchpoint setting fails, consider
disable/delete
                            existing ones to free up resources.
  watchpoint set variable   -- Set a watchpoint on a variable. Use the '-w' option to specify
the type
                            of watchpoint and the '-s' option to specify the byte size to
watch for.
                            If no '-w' option is specified, it defaults to write. If no '-s'
option
                            is specified, it defaults to the variable's byte size. Note that
there
                            are limited hardware resources for watchpoints. If watchpoint
```

```
setting
                          fails, consider disable/delete existing ones to free up
resources.
  wivar                  -- Set a watchpoint for an object's instance variable.
```

另外，可以为一些常用命令设置别名，然后将其写到 LLDB 的初始化文件 ~/.lldbinit 中，具体如下。

```
command alias -H "Reload ~/.lldbinit" -h "Reload ~/.lldbinit" -- reload_lldbinit command
source ~/.lldbinit
command alias pcc process connect connect://localhost:1234
command alias iheap command script import lldb.macosx.heap
```

5.2.2　LLDB 解密

除了使用 dumpdecrypted 解密从 App Store 下载的应用的可执行文件和 framework，也可以使用 LLDB 直接从内存中 dump 出解密后的模块。下面以解密 Snapchat 中的 Cronet.framework 为例描述如何使用 LLDB 对其动态 dump 进行解密。

1. 准备工作

在解密之前，需要获取加密的可执行文件，查看一些对应的偏移，为解密做准备。为了获取 App 里面的 Cronet.framework 文件，可以先通过 ps 命令找到运行 App 的可执行文件路径，然后通过该路径获取 Frameworks 文件的位置，以找到 Cronet.framework 文件，最后通过 scp 命令将其复制到计算机中，具体如下。

```
#设备执行的命令
Monkey:~ root# ps aux | grep Snapchat
mobile    1681   1.6  4.0   784748  40980   ??  Ss   11:54PM   0:04.69
/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/Snapchat
.app/Snapchat
root      2194   0.0  0.0   536256    464 s000  R+   11:26PM   0:00.01 grep Snapchat
Monkey:~ root# cd
/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/Snapchat
.app/
Monkey:/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/S
napchat.app root# ls Frameworks/
Cronet.framework      libswiftDarwin.dylib      libswiftObjectiveC.dylib
libswiftCore.dylib          libswiftDispatch.dylib
libswiftCoreGraphics.dylib  libswiftFoundation.dylib
Monkey:/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/S
```

```
napchat.app root#

#计算机执行的命令
➜  sourcecode git:(master) X scp -P 2222 -r
root@localhost:/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C49
2DF205/Snapchat.app/Frameworks/Cronet.framework ./
Cronet                                        100% 6232KB   6.1MB/s   00:01
Info.plist                                    100%  977   113.0KB/s   00:00
Cronet.sinf                                   100% 1056   224.4KB/s   00:00
Cronet.supf                                   100%   45KB   3.8MB/s   00:00
Cronet.supp                                   100%   47KB   3.6MB/s   00:00
Cronet_armv7.supf                             100%   45KB   3.0MB/s   00:00
CodeResources
......
```

得到目标的可执行文件之后，可以通过如下命令获取文件头部的一些信息。这些信息将为后面的解密和写入解密内容所用。

```
➜  Cronet.framework git:(master) X otool -hf Cronet
Fat headers
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
    cputype 12
    cpusubtype 9
    capabilities 0x0
    offset 16384
    size 2749664
    align 2^14 (16384)
architecture 1
    cputype 16777228
    cpusubtype 0
    capabilities 0x0
    offset 2768896
    size 3612224
    align 2^14 (16384)
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
0xfeedface      12          9  0x00           6    27       3328 0x00118085
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
0xfeedfacf 16777228          0  0x00           6    27       3816 0x00118085

➜  Cronet.framework git:(master) X otool -arch arm64 -l Cronet | grep crypt
```

```
    cryptoff 16384
   cryptsize 3309568
     cryptid 1
```

由于笔者的机器是 64 位的，在这里获取的是 ARM64 的加密信息。如果是 32 位的机器，将“ARM64”改为“ARMv7”即可。

2. LLDB 附加

首先使用 LLDB 进行附加。附加成功，即可通过如下命令获取需要解密的目标文件加载模块的基地址。

```
(lldb) im li Cronet
[  0] 188F5BF7-B4C4-36EF-BB9A-976FA870F9D7 0x0000000105920000
/private/var/mobile/Containers/Bundle/Application/3A4C68EB-4059-47D4-ACE6-BE9C492DF205/
Snapchat.app/Frameworks/Cronet.framework/Cronet (0x0000000105920000)
```

然后，通过如下命令从内存中 dump 出解密后的二进制数据。

```
(lldb) memory read --force --outfile ~/Desktop/dumpoutput --binary --count 3309568
16384+0x0000000105920000
3309568 bytes written to '/Users/monkey/Desktop/dumpoutput'
```

~/Desktop/dumpoutput 是保存到本地计算机的目标文件路径。3309568 是通过 otool 读取的 cryptsize 加密数据的大小。16384 + 0x0000000105920000 是当前模块加载基地址加上加密数据的偏移地址，它是加密数据在内存中真正开始的位置。完成此步骤，会在 ~/Desktop/ 目录下面得到一个 dumpoutput 文件，这个文件就包含了是解密后的数据。

3. 修复文件

因为 dump 出来的文件都没有 Mach-O 文件头，所以在这里要先把 dump 出来的数据写回原来加密的文件，以替换原来加密的部分，相关代码如下。

```
➜  Cronet.framework git:(master) ✗ dd seek=2785280 bs=1 conv=notrunc
if=/Users/monkey/Desktop/dumpoutput of=./Cronet
3309568+0 records in
3309568+0 records out
3309568 bytes transferred in 4.698067 secs (704453 bytes/sec)

➜  Cronet.framework git:(master) ✗ lipo Cronet -thin arm64 -output Cronet_arm64
```

2785280 是 2768896 + 16384 的计算结果。2768896 是之前获取的 ARM64 架构的偏移值 offset，16384 是加密数据的偏移值 cryptoff，二者相加，得到了写入的加密数据在文件中的偏移值。替换之后，使用 lipo 从 FAT 文件中提取 ARM64 架构的文件。将 Cronet_arm64 拖到 MachOView 中，修改 cryptid 为 00000000，如图 5-35 所示，就完成了 Framework 解密。整个过程可以使用 Python 脚本调用 LLDB 提供的接口自动完成，GitHub 上有其实现代码，读者也可以自己去实现。

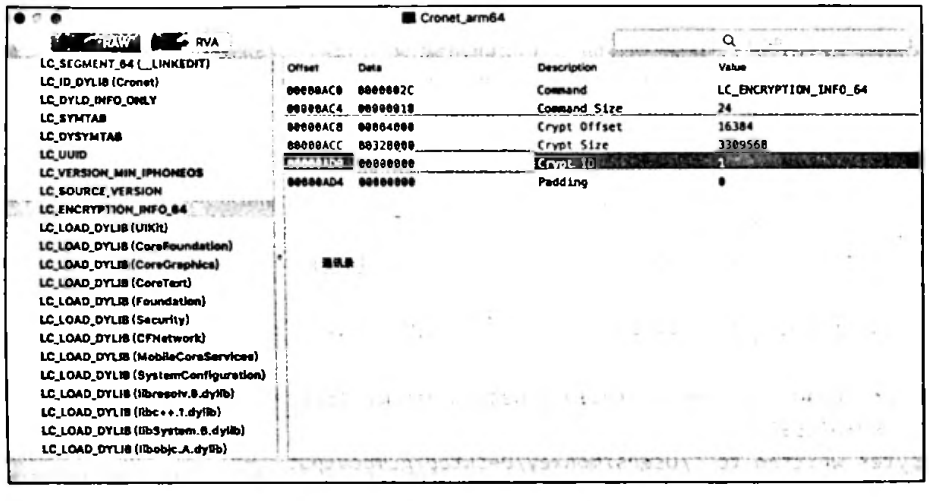


图 5-35 修改加密标识 cryptid 为 00000000

5.2.3 用 Xcode 调试第三方应用

在使用命令行进行调试时，查看堆栈、内存等都需要使用命令，而习惯使用 Xcode 进行调试的开发者可能不太习惯对着终端命令窗口进行调试，本节就将介绍如何在非越狱设备上直接使用 Xcode 来调试第三方应用、进行符号的还原、查看带符号的堆栈调用，从而让调试第三方应用和调试自己的应用一样简单。在调试之前，需要准备一个解密的 app 包（里面的 Framework 和 Extension 都需要解密，这部分操作可以参考 3.1 节）。

01 新建 iOS App 项目

使用 Xcode 新建一个 iOS App 项目。单击菜单项“File”→“New”→“Project”，在弹出的窗口选择“iOS”→“Single View App”选项，如图 5-36 所示。

02 增加自定义脚本

Xcode 提供了一个强大的功能，开发者可以在编译流程中增加自己的脚本来修改工程的一

些配置或者文件。单击“Build Phases”标签，然后单击左上角的加号按钮，选择“New Run Script Phase”选项，如图 5-37 所示。

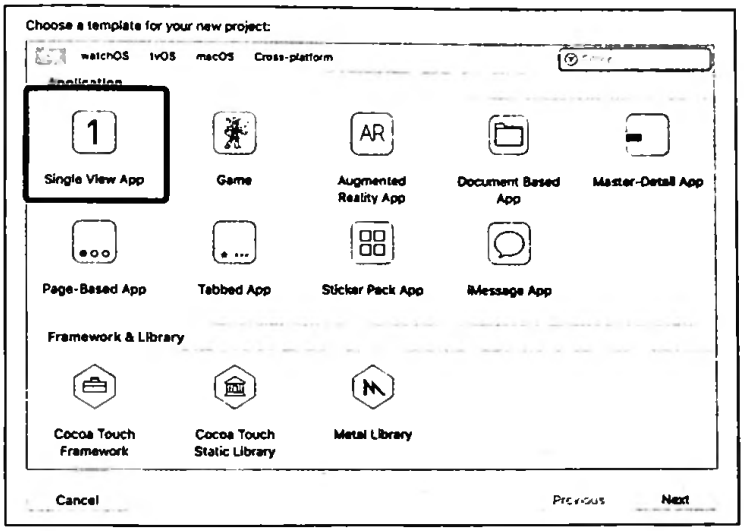


图 5-36　使用 Xcode 新建一个 Single View App 工程

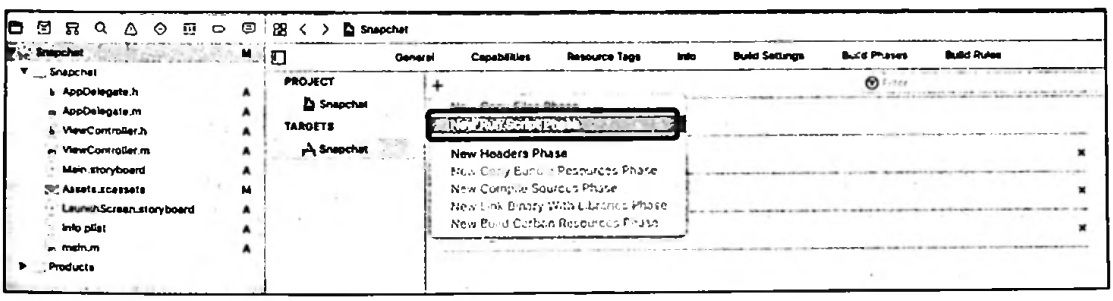


图 5-37　创建一个自定义脚本

在项目中新建一个 pack.sh 文件，将其保存在源文件目录下面，并使用“chmod +x pack.sh”命令对文件赋予可执行权限，如图 5-38 所示。

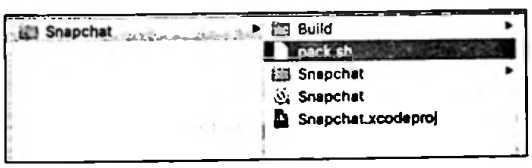


图 5-38　创建 pack.sh 文件

在自定义脚本处填入如下内容，如图 5-39 所示。

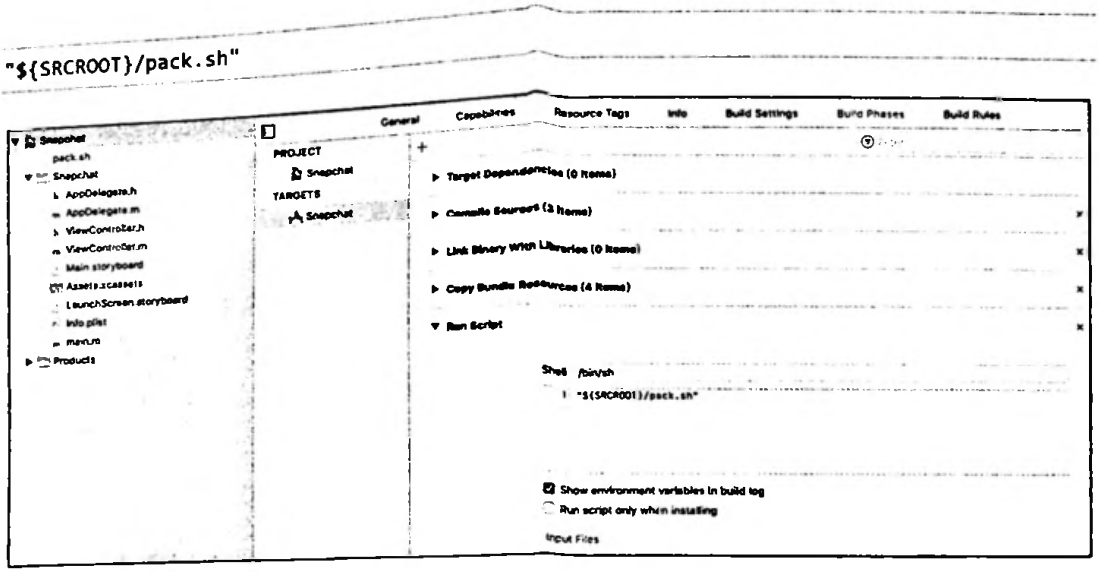


图 5-39 填入自定义的 Shell 脚本

03 编写 Shell 脚本

为了让一个签名被破坏了的程序运行在非越狱的设备上面，需要在脚本里面给应用重新签名，还需要替换该项目本身生成的 app 包，这样 Xcode 才会运行复制过去的被替换的应用。将准备好的解密后的应用和 pack.sh 文件放在同一文件夹中，编写如下 Shell 脚本。

```
TARGET_APP_PATH="${SRCROOT}/Snapchat.app"                    #需要复制的目标应用
BUILD_APP_PATH="$BUILT_PRODUCTS_DIR/$TARGET_NAME.app"        #build 生成的应用

#签名所有 framework 和动态库
function codesign(){
   for file in `ls $1`;
   do
              extension="${file#*.}"
      if [[ -d "$1/$file" ]]; then
                    if [[ "$extension" == "framework" ]]; then
                           /usr/bin/codesign --force --sign
"$EXPANDED_CODE_SIGN_IDENTITY" "$1/$file"
                    else
                           codesign "$1/$file"
                    fi
             elif [[ -f "$1/$file" ]]; then
                    if [[ "$extension" == "dylib" ]]; then
                           /usr/bin/codesign --force --sign
```

```
"$EXPANDED_CODE_SIGN_IDENTITY" "$1/$file"
                fi
        fi
    done
}

#删除原目录
rm -rf "$BUILD_APP_PATH" || true
mkdir -p "$BUILD_APP_PATH" || true

#复制目标应用，替换 build 生成的应用
cp -rf "$TARGET_APP_PATH/" "$BUILD_APP_PATH/"

#删除 PlugIns 和 Watch
rm -rf "$BUILD_APP_PATH/PlugIns" || true
rm -rf "$BUILD_APP_PATH/Watch" || true

codesign "$BUILD_APP_PATH"
```

写好脚本，单击 Xcode 的“Run”按钮，就能将应用运行在非越狱设备上了。

04　在 Xcode 中下断点

接下来，就能使用 Xcode 直接调试第三方应用了。单击 Xcode 的暂停按钮，如图 5-40 所示，在 LLDB 命令行中输入命令下断点。断点还是下在 `+[Manager performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:rememberDevice:fromDeepLink:onComplete:]` 函数处，地址为 0x000000010095ECCC。根据模块加载基地址下断点，然后输入“c”，继续运行。相关代码如下。

```
(lldb) im li -o -f Snapchat
[  0] 0x000000000000b4000
/Users/monkey/Documents/iosreversebook/sourcecode/chapter-4/xxx/Snapchat/Build/Products
/Debug-iphoneos/Snapchat.app/Snapchat
(lldb) br s -a '0x000000010095ECCC+0x00000000000b4000'
Breakpoint 1: where = Snapchat`___lldb_unnamed_symbol47133$$Snapchat, address =
0x0000000100a12ccc
(lldb) c
Process 15841 resuming
```

打开 Snapchat，输入用户名和密码，点击“登录”按钮，Xcode 便会中断，如图 5-41 所示。此时还能看到当前程序的调用堆栈。如果堆栈没有展开，单击如图 5-41 所示界面左栏下方左边的按钮即可。

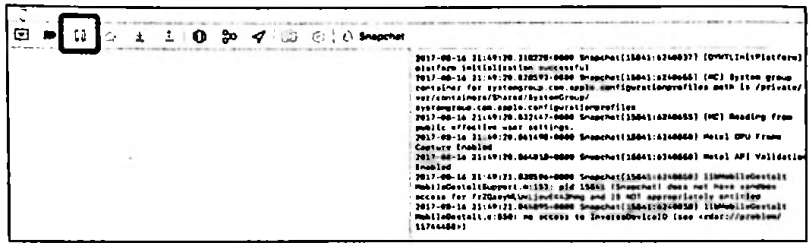


图 5–40　暂停 Xcode 进入 LLDB 命令行交互模式

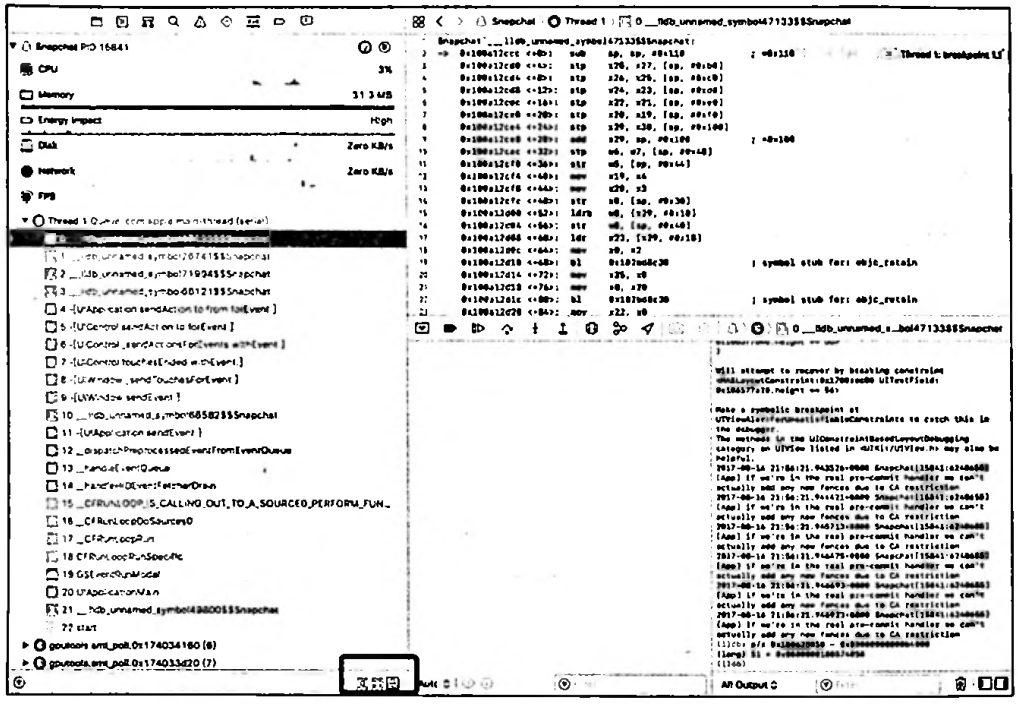


图 5–41　Xcode 触发断点后中断查看调用堆栈

切换到上一个调用栈，可以看到调用地址为 0x100628050，如图 5–42 所示。用该地址减去当前偏移基地址，得到文件偏移 0x0000000100574050，代码如下。使用 IDA 查看对应偏移地址所对应的代码，可知是从 `-[SCLoginV2ViewController performLogin]` 函数调用该函数的，如图 5–43 所示。

```
(lldb) p/x 0x100628050 - 0x00000000000b4000
(long) $1 = 0x0000000100574050
```

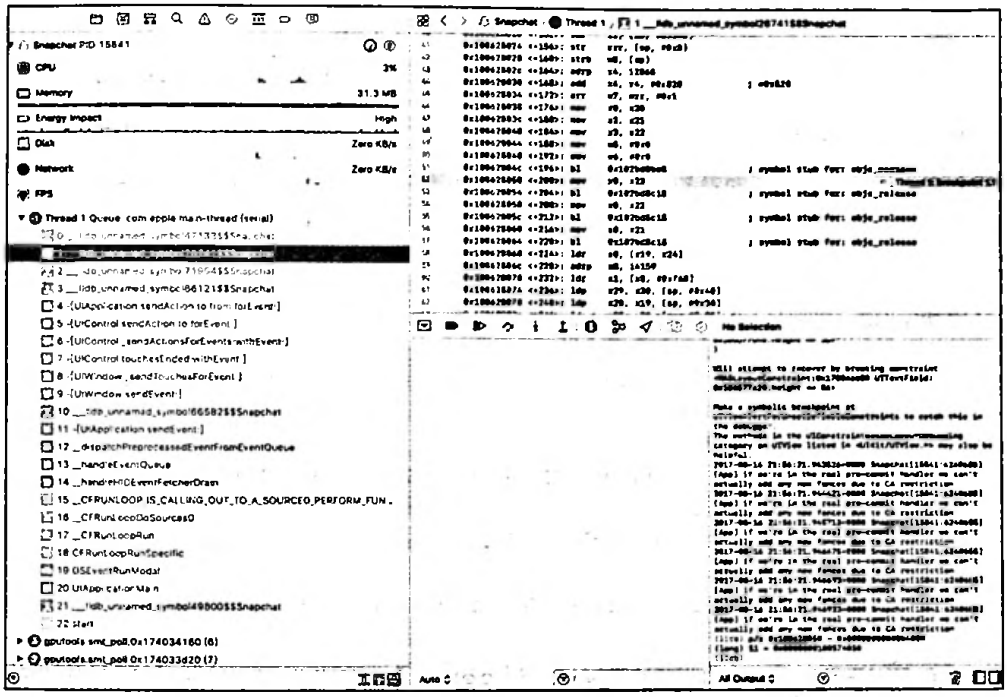


图 5-42　使用 Xcode 查看上一个调用栏

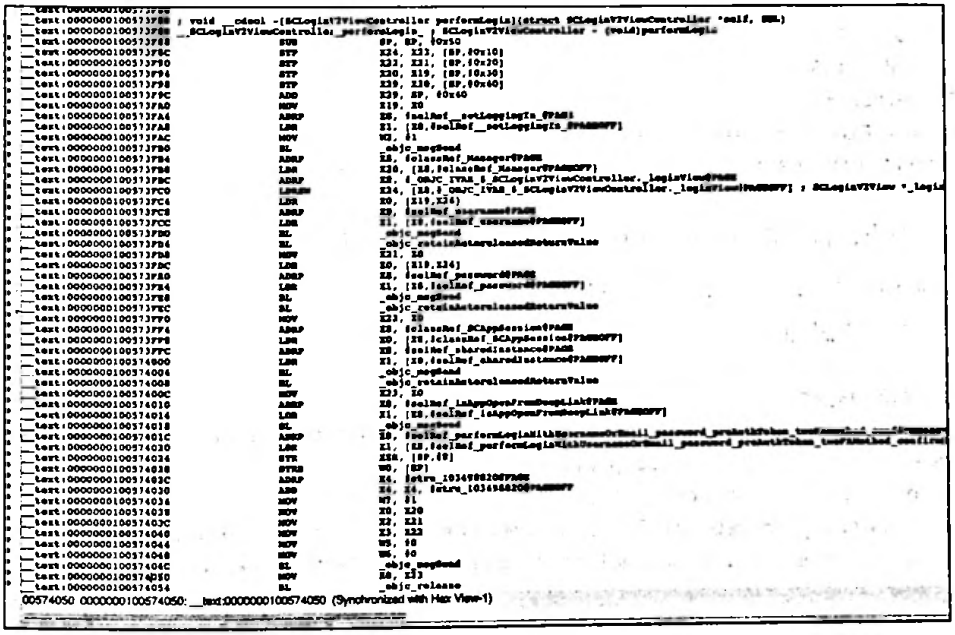


图 5-43　使用 IDA 定位对应偏移地址

05 符号还原

每次手动计算符号偏移很麻烦，二进制本身已经包含了这些符号，且这些和符号和偏移都是一一对应的，而 Xcode 找不到这些符号的原因是在符号表里面没有保存其中的对应关系。因此，可以根据对应二进制的分析将这些 OC 符号和偏移对应起来并写入符号表，这样就能让 Xcode 识别对应的符号了。GitHub 上已经有开源工具实现了这个功能，页面地址为 https://github.com/tobefuturer/restore-symbol。下载该开源工具，直接运行 make 命令，即可生成 restore-symbol 工具。使用 restore-symbol 工具对 Snapchat 的二进制文件进行符号还原，具体如下。

```
➜  Snapchat git:(master) X ./restore-symbol Snapchat -o Snapchat_with_symbol
=========== Start =============
Scan OC method in mach-o-file.
2017-08-16 22:19:03.047 restore-symbol[79780:3169986] Warning: Parsing instance variable
type failed, _isPersistedLocally
2017-08-16 22:19:04.555 restore-symbol[79780:3169986] Warning: Parsing instance variable
type failed, _outgoingCount
2017-08-16 22:19:04.555 restore-symbol[79780:3169986] Warning: Parsing instance variable
type failed, _incomingCount
2017-08-16 22:19:04.577 restore-symbol[79780:3169986] Warning: Parsing instance variable
type failed, _mediaScenePath
2017-08-16 22:19:04.585 restore-symbol[79780:3169986] Warning: Parsing instance variable
type failed, _isAvailableLocally
2017-08-16 22:19:04.586 restore-symbol[79780:3169986] Warning: Parsing instance variable
type failed, _isSynced
Scan OC method finish.
=========== Finish ============
➜  Snapchat git:(master) X mv Snapchat_with_symbol Snapchat
```

完成操作，将处理后的 Snapchat 复制回 Snapchat.app 目录，替换原来的可执行文件。再次使用 Xcode 运行项目，暂停并进入 LLDB 交互模式，由于已经进行了符号还原，可以直接使用符号下断点，代码如下。

```
(lldb) b +[Manager
performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:r
ememberDevice:fromDeepLink:onComplete:]
Breakpoint 1: where = Snapchat`+[Manager
performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:r
ememberDevice:fromDeepLink:onComplete:], address = 0x000000010095eccc
```

触发断点后也能够看到完整的带符号的调用堆栈，如图 5-44 所示。在 LLDB 交互模式下，同样可以使用各种 LLDB 命令查看内存、OC 对象等。

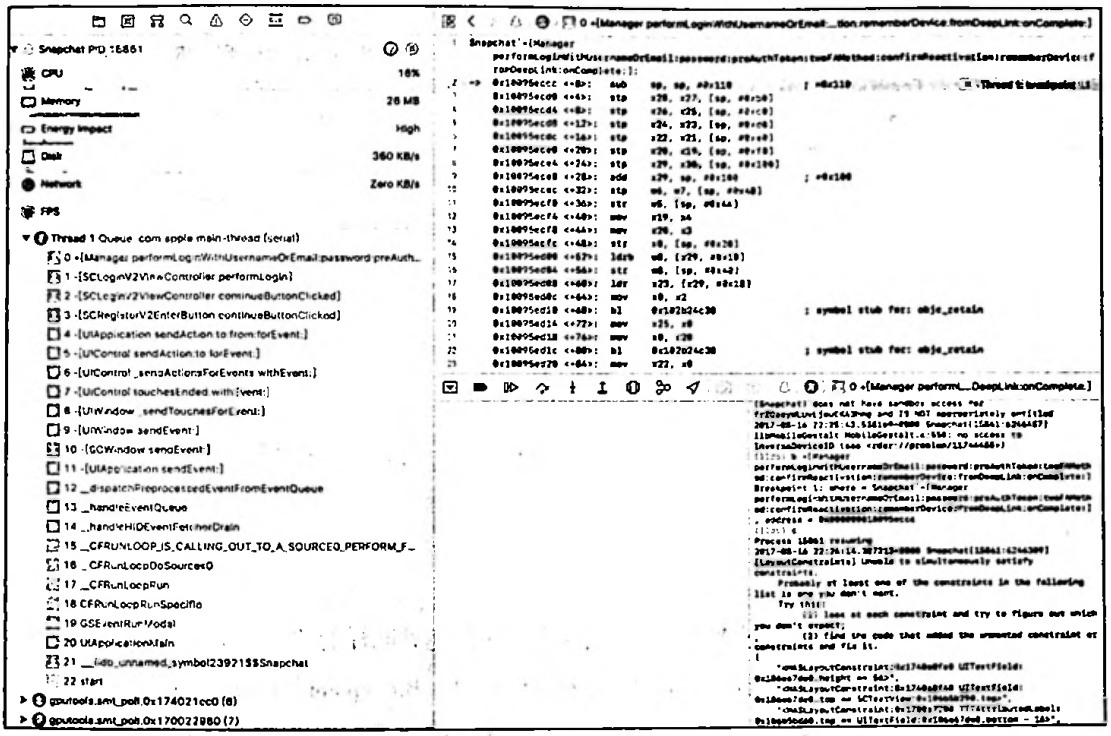


图 5-44　使用 Xcode 查看带符号的堆栈信息

5.2.4　LLDB 的高级调试技巧

除了基本的调试，我们还需要学习和掌握 LLDB 的高级调试技巧，这样才能在动态分析中达到事半功倍的效果。接下来，笔者将介绍几种在实际分析中使用较多且能极大提高分析速度的技巧。

1. 给断点添加命令

在动态调试过程中，可以通过给某个断点添加一些命令，在每次断点被触发时打印某些寄存器的值，然后让程序继续运行。使用如下命令就可以达到这样的效果。在输入自定义的命令之后，输入“DONE”作为结束，这样在每次触发该断点时就会自动打印调用的类和方法，然后让程序继续运行。

```
(lldb) b +[Manager
performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:r
ememberDevice:fromDeepLink:onComplete:]
Breakpoint 2: where = Snapchat`+[Manager
performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:r
ememberDevice:fromDeepLink:onComplete:], address = 0x00000001009e6ccc
(lldb) br com add 2
Enter your debugger command(s).  Type 'DONE' to end.
> po $x2
> po $x3
> c
> DONE
(lldb) c
Process 17733 resuming
 po $x2
alonemonkey
 po $x3
123456
 c
Process 17733 resuming
Command #3 'c' continued the target.
```

除了使用命令行，直接通过 Xcode 添加符号断点也能达到同样的效果。单击 Xcode 界面左栏的箭头按钮，然后单击左下角的加号按钮，选择“Symbolic Breakpoint”选项，如图 5–45 所示。在“Symbol”文本框中填入想要跟踪的函数，单击“Action”列表，选择“Debugger Command”选项，在下方的输入框中填入断点触发后执行的调试命令，为了在触发断点后让程序继续运行，需要勾选“Automatically continue after evaluating actions”选项，如图 5–46 所示。当然，还可以设置触发条件、忽略次数及其他 Action 选项。

2. 使用 Python 脚本

LLDB 提供了很多 API 来帮助我们获取和修改调试信息、进程信息及内存信息。这些接口可以直接通过 Python 脚本调用，然后通过 LLDB 加载使用。我们可以阅读 LLDB 官方提供的 API 文档 http://lldb.llvm.org/python_reference/index.html。在阅读文档前，建议你看看别人是怎么编写代码的，例如接下来要介绍的两个开源库。

- chisel：Facebook 开源的 LLDB 命令工具（https://github.com/facebook/chisel）。
- LLDB：Derek Selander 开源的工具。在他的著作 *Advanced Apple Debugging and Reverse Engineering* 中也讲到了一些高级调试技巧（https://github.com/DerekSelander/LLDB）。

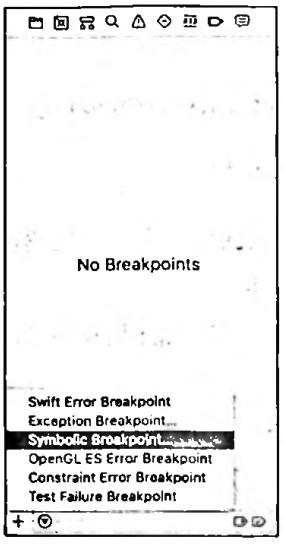


图 5-45　使用 Xcode 添加符号断点

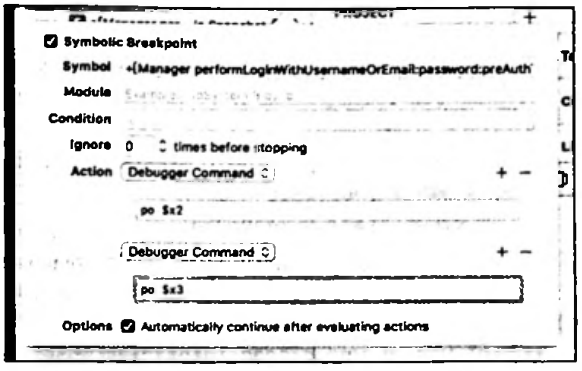


图 5-46　使用符号断点设置 Debugger Command

首先，安装开源脚本。LLDB 会默认从 ~/.lldbinit 加载自定义脚本。如果没有这个文件，可以先使用 `touch ~/.lldbinit` 命令新建一个，再使用 `brew install chisel` 命令安装 chisel，具体如下，或者把 chisel 和 LLDB 的代码克隆下来。

```
touch ~/.lldbinit
brew install chisel
或者克隆代码:
➜  LLDB git clone https://github.com/facebook/chisel.git
Cloning into 'chisel'...
remote: Counting objects: 1419, done.
remote: Total 1419 (delta 0), reused 0 (delta 0), pack-reused 1419
Receiving objects: 100% (1419/1419), 3.26 MiB | 1.05 MiB/s, done.
Resolving deltas: 100% (846/846), done.
➜  LLDB git clone https://github.com/DerekSelander/LLDB.git
Cloning into 'LLDB'...
remote: Counting objects: 781, done.
remote: Compressing objects: 100% (52/52), done.
remote: Total 781 (delta 39), reused 52 (delta 22), pack-reused 707
Receiving objects: 100% (781/781), 18.41 MiB | 119.00 KiB/s, done.
Resolving deltas: 100% (473/473), done.
```

然后，在 ~/.lldbinit 文件中加载下载的脚本，具体如下。这里是笔者本地的代码路径，你需要将其换成自己下载的脚本所对应的路径。

```
# ~/.lldbinit
...
command script import /Users/monkey/Documents/iosreversecode/LLDB/chisel/fblldb.py
command script import
/Users/monkey/Documents/iosreversecode/LLDB/LLDB/lldb_commands/dslldb.py
```

重启 Xcode，或者在 LLDB 交互模式下输入 command source ~/.lldbinit 命令，加载脚本。加载成功后，LLDB 会提供 reload_lldbinit 来重新加载 ~/.lldbinit 文件。

在 GitHub 中，对 Xcode 和 LLDB 命令都有相应的介绍，下面以其中几个常用的命令为例来讲解。

打印当前界面结构和 ViewController，如果 pviews 出错，就要先导入 UIKit，具体如下。

```
(lldb) pviews
error: error: use of undeclared identifier 'UIApplication'
None
(lldb) expression @import UIKit
(lldb) pviews
<SCWindow: 0x113d304d0; baseClass = UIWindow; frame = (0 0; 375 667); gestureRecognizers
= <NSArray: 0x170245940>; layer = <UIWindowLayer: 0x17003a340>>
   | <UILayoutContainerView: 0x113e18470; frame = (0 0; 375 667); autoresize = W+H;
gestureRecognizers = <NSArray: 0x17425f050>; layer = <CALayer: 0x174037f40>>
   |    | <UIView: 0x113d40940; frame = (0 0; 375 667); layer = <CALayer: 0x17003da80>>
   |    |    | <SCManagedCapturePreviewView: 0x113d43550; frame = (0 0; 375 667);
clipsToBounds = YES; layer = <CALayer: 0x17003e280>>
   |    |    |    | <CALayer: 0x17003e2a0> (layer)
   |    |    | <UIView: 0x113d40ae0; frame = (0 0; 375 667); alpha = 0; layer = <CALayer:
0x17003dae0>>
   |    |    | <UIToolbar: 0x113d40c80; frame = (0 0; 375 667); alpha = 0.791373; layer =
<CALayer: 0x174026740>>
   |    |    |    | <_UIBarBackground: 0x113e36010; frame = (0 0; 375 667);
userInteractionEnabled = NO; layer = <CALayer: 0x174220500>>
   |    |    |    |    | <UIImageView: 0x113d40e80; frame = (0 -0.5; 375 0.5);
userInteractionEnabled = NO; layer = <CALayer: 0x17003db80>>
   |    |    |    |    | <UIVisualEffectView: 0x113d41070; frame = (0 0; 375 667); layer =
<CALayer: 0x17003dba0>>
   |    |    |    |    |    | <_UIVisualEffectBackdropView: 0x113d41630; frame = (0 0; 375
667); autoresize = W+H; userInteractionEnabled = NO; layer = <UICABackdropLayer: 0x17003dc00>>
   |    |    |    |    |    | <_UIVisualEffectFilterView: 0x113e565f0; frame = (0 0; 375 667);
autoresize = W+H; userInteractionEnabled = NO; layer = <CALayer: 0x174225a60>>
   |    |    | <UIView: 0x113e36ac0; frame = (0 0; 375 667); alpha = 0.846875; layer = <CALayer:
0x174221280>>
```

```
   |    | <UINavigationTransitionView: 0x113d31310; frame = (0 0; 375 667); clipsToBounds
= YES; autoresize = W+H; layer = <CALayer: 0x17003ac60>>
   |    |    | <UIViewControllerWrapperView: 0x113d84860; frame = (0 0; 375 667); layer =
<CALayer: 0x17022b320>>
   |    |    |    | <UIView: 0x113e007e0; frame = (0 0; 375 667); layer = <CALayer:
0x17403eae0>>
   |    |    |    |    | <SCButton: 0x113d43b30; baseClass = UIButton; frame = (0 507; 375
80); clipsToBounds = YES; opaque = NO; layer = <CAShapeLayer: 0x1742212e0>>
   |    |    |    |    |    | <UIButtonLabel: 0x113d451e0; frame = (161.5 23; 52 34.5); text
= '登录'; opaque = NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x17009f810>>
   |    |    |    |    |    |    | <_UILabelContentLayer: 0x174225980> (layer)
   |    |    |    |    | <SCButton: 0x113e39b20; baseClass = UIButton; frame = (0 587; 375
80); clipsToBounds = YES; opaque = NO; layer = <CAShapeLayer: 0x1742223e0>>
   |    |    |    |    |    | <UIButtonLabel: 0x113e3a040; frame = (161.5 23; 52 34.5); text
= '注册'; opaque = NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x17428a780>>
   |    |    |    |    |    |    | <_UILabelContentLayer: 0x174225820> (layer)
   |    |    |    |    | <UIImageView: 0x113e3b3d0; frame = (177 207; 21 20); hidden = YES;
opaque = NO; userInteractionEnabled = NO; layer = <CALayer: 0x1742229e0>>
   |    | <UIImageView: 0x113e3a8f0; frame = (0 0; 6 6); opaque = NO; autoresize = RM+BM;
userInteractionEnabled = NO; layer = <CALayer: 0x174222640>>
   |    | <UIImageView: 0x113e3baa0; frame = (369 0; 6 6); transform = [-1, 0, 0, 1, 0, 0];
opaque = NO; autoresize = LM+BM; userInteractionEnabled = NO; layer = <CALayer: 0x174222540>>
   |    | <UIImageView: 0x113d47aa0; frame = (0 661; 6 6); transform = [1, 0, 0, -1, 0, 0];
opaque = NO; autoresize = RM+TM; userInteractionEnabled = NO; layer = <CALayer: 0x17003f8a0>>
   |    | <UIImageView: 0x113d47c90; frame = (369 661; 6 6); transform = [-1, 0, 0, -1, 0,
0]; opaque = NO; autoresize = LM+TM; userInteractionEnabled = NO; layer = <CALayer:
0x17003f920>>
(lldb) pvc
<NavigationController 0x114038400>, state: appeared, view: <UILayoutContainerView
0x113e18470>
   | <LoginRegisterViewController 0x113d30dd0>, state: appeared, view: <UIView 0x113e007e0>
```

查看“登录”按钮 UIButtonLabel 的响应链，具体如下。

```
(lldb) presponder 0x113d451e0
<UIButtonLabel: 0x113d451e0; frame = (161.5 23; 52 34.5); text = '登录'; opaque = NO;
userInteractionEnabled = NO; layer = <_UILabelLayer: 0x17009f810>>
   | <SCButton: 0x113d43b30; baseClass = UIButton; frame = (0 507; 375 80); clipsToBounds
= YES; opaque = NO; layer = <CAShapeLayer: 0x1742212e0>>
   |    | <UIView: 0x113e007e0; frame = (0 0; 375 667); layer = <CALayer: 0x17403eae0>>
   |    |    | <LoginRegisterViewController: 0x113d30dd0>
   |    |    |    | <UIViewControllerWrapperView: 0x113d84860; frame = (0 0; 375 667); layer
= <CALayer: 0x17022b320>>
   |    |    |    |    | <UINavigationTransitionView: 0x113d31310; frame = (0 0; 375 667);
```

```
clipsToBounds = YES; autoresize = W+H; layer = <CALayer: 0x17003ac60>>
   |    |    |    |    |    | <UILayoutContainerView: 0x113e18470; frame = (0 0; 375 667);
autoresize = W+H; gestureRecognizers = <NSArray: 0x17425f050>; layer = <CALayer: 0x174037f40>>
   |    |    |    |    |    |    | <NavigationController: 0x114038400>
   |    |    |    |    |    |    |    | <SCWindow: 0x113d304d0; baseClass = UIWindow; frame
= (0 0; 375 667); gestureRecognizers = <NSArray: 0x170245940>; layer = <UIWindowLayer:
0x17003a340>>
   |    |    |    |    |    |    |    |    | <SCApplication: 0x113e02620>
   |    |    |    |    |    |    |    |    |    | <SCAppDelegate: 0x174104380>
(lldb)
```

查看“登录”按钮的 Action 事件，具体如下。

```
(lldb) pactions 0x113d43b30
<LoginRegisterViewController: 0x113d30dd0>: snapchatterButtonClicked,
touchDownInsideButton, touchUpOutside
```

为了方便地查看界面布局和控件，也可以直接使用 Xcode 自带的工具。单击调试栏中的“Debug View Hierarchy”按钮，如图 5-47 所示。界面布局如图 5-48 所示。单击“登录”按钮后，还能在界面右侧看到该按钮的一些详细信息和 target-action。

查看注册用户时，需要检测用户名有效性函数的 Block 参数，相关代码如下。

```
(lldb) b [Manager checkRequestedUsername:onComplete:]
Breakpoint 10: where = Snapchat`-[Manager checkRequestedUsername:onComplete:], address =
0x000000010056eb24
(lldb) c
Process 17733 resuming
(lldb) po $x3
<__NSMallocBlock__: 0x17444b6d0>

(lldb) pblock 0x17444b6d0
Imp: 0x1011c633c    Signature: void ^(NSString *, NSString *, NSArray *);
(lldb)
```

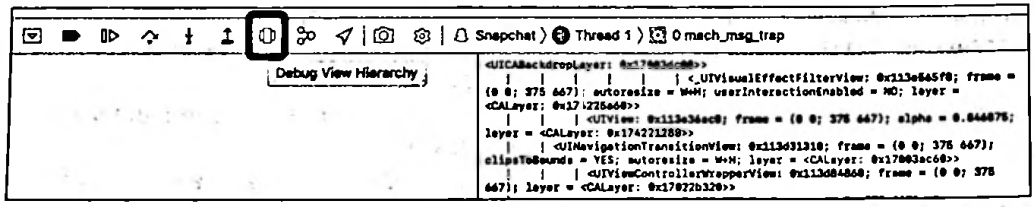


图 5-47　单击 Xcode 工具栏的 Debug View Hierarchy 按钮

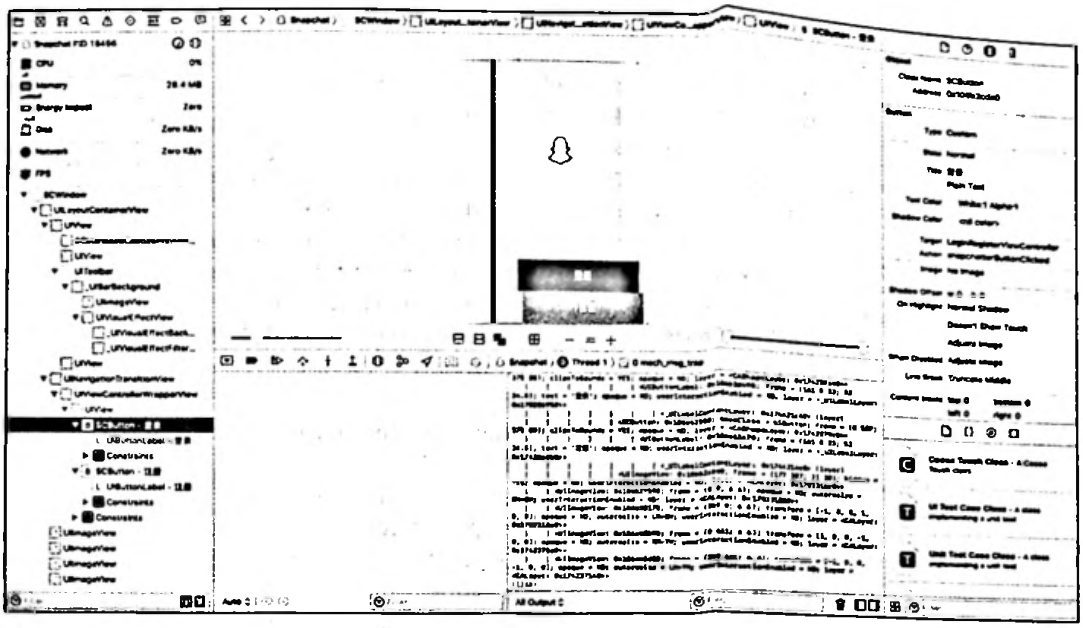


图 5-48　使用 Xcode 查看界面布局

查看 LoginRegisterViewController 的所有方法及对应的内存地址。找到内存地址，就可以直接对方法的地址下断点了。相关代码如下。

```
(lldb) methods 0x113de6270
<LoginRegisterViewController: 0x113de6270>:
in LoginRegisterViewController:
        Properties:
                @property (retain, nonatomic) UIColor* backgroundPatternColor;
(@synthesize backgroundPatternColor = _backgroundPatternColor;)
                @property (retain, nonatomic) UIImageView* ghostFaceImageView;
(@synthesize ghostFaceImageView = _ghostFaceImageView;)
                @property (readonly) unsigned long hash;
                @property (readonly) Class superclass;
                @property (readonly, copy) NSString* description;
                @property (readonly, copy) NSString* debugDescription;
        Instance Methods:
                - (id) getPageViewName; (0x1009fb104)
                - (BOOL) canPerformNavigation; (0x1006bf828)
                - (void) newMemberButtonClicked; (0x100b818c0)
                - (void) removeObservers; (0x100f94010)
                - (void) handleRegistration; (0x10107b65c)
                - (void) addObservers; (0x100c7b500)
```

```
                - (id) ghostFaceImageView; (0x100d913ac)
                - (id) backgroundPatternColor; (0x100907b7c)
                - (void) touchDownInsideButton; (0x100e81e34)
                - (void) touchUpOutside; (0x1007817d4)
                - (void) addTweaksGestures; (0x10084f890)
                - (void) verifyPhoneFromDeepLink:(id)arg1; (0x100478e3c)
                - (void) tripleDoubleTap:(id)arg1; (0x1003e5be0)
                - (void) setBackgroundPatternColor:(id)arg1; (0x100e495f4)
                - (void) tweakViewControllerPressedDone:(id)arg1; (0x1005544c4)
                - (void) snapchatterButtonClicked; (0x100d61eac)
                - (void) setGhostFaceImageView:(id)arg1; (0x100ef1b84)
                - (void) .cxx_destruct; (0x100501ed0)
                - (void) dealloc; (0x100bf0718)
                - (unsigned long) supportedInterfaceOrientations; (0x100bd0698)
                - (void) loadView; (0x100895e8c)
                - (void) viewWillAppear:(BOOL)arg1; (0x100c730bc)
                - (void) viewDidAppear:(BOOL)arg1; (0x10039e2f4)
                - (void) viewDidDisappear:(BOOL)arg1; (0x10044dd14)
                - (void) viewDidLoad; (0x1002c386c)
                - (void) willEnterForeground:(id)arg1; (0x10097c538)
in SCCameraBackgroundViewController:
        Properties:
                @property (retain, nonatomic) UIToolbar* blurOverlay;  (@synthesize
blurOverlay = _blurOverlay;)
                @property (retain, nonatomic) UIView* colorOverlay;  (@synthesize
colorOverlay = _colorOverlay;)
                @property (retain, nonatomic) UIView* cameraView;  (@synthesize
cameraView = _cameraView;)
                @property (retain, nonatomic) UIToolbar* extraBlur;  (@synthesize
extraBlur = _extraBlur;)
                @property (retain, nonatomic) UIView* whiteTransitionOverlay;
(@synthesize whiteTransitionOverlay = _whiteTransitionOverlay;)
        Instance Methods:
                - (void) initCameraBlur; (0x100bc6e3c)
                - (id) colorOverlay; (0x10086f218)
                - (void) fadeOutBlurAndTopViewFromView:(id)arg1
withCompleteBlock:(^block)arg2; (0x100460320)
                - (void) setExtraBlur:(id)arg1; (0x10096eb54)
                - (void) setupCaptureVideoPreviewView; (0x100c96dd4)
                - (void) setBlurOverlay:(id)arg1; (0x1006c970c)
                - (void) setWhiteTransitionOverlay:(id)arg1; (0x1004fdffc)
                - (id) whiteTransitionOverlay; (0x100c317b8)
                - (void) setColorOverlay:(id)arg1; (0x100ebd6fc)
                - (void) animateFromWhiteToCamera; (0x10104d524)
```

```
                - (id) blurOverlay; (0x100bd3c74)
                - (id) extraBlur; (0x100311f34)
                - (void) initCamera; (0x1004a90dc)
                - (void) .cxx_destruct; (0x100c623d0)
                - (void) viewWillAppear:(BOOL)arg1; (0x10052b134)
                - (void) viewWillDisappear:(BOOL)arg1; (0x100e05a70)
                - (void) viewDidDisappear:(BOOL)arg1; (0x1009dd72c)
                - (void) didEnterBackground:(id)arg1; (0x100548cec)
                - (void) willEnterForeground:(id)arg1; (0x100451a20)
                - (id) cameraView; (0x1008a232c)
                - (void) setCameraView:(id)arg1; (0x100cf0338)
(UIViewController ...)

(lldb)
```

搜索 UITextField 的实例对象和 Cycript 中的 choose 一样的，具体如下。

```
(lldb) search UITextField
<UITextField: 0x12fe669b0; frame = (3 0; 267 44); text = '123456'; clipsToBounds = YES; opaque
= NO; gestureRecognizers = <NSArray: 0x1704484f0>; layer = <CALayer: 0x17402e720>>

<UITextField: 0x12fd63280; frame = (3 0; 267 44); text = 'alonemonkey '; clipsToBounds =
YES; opaque = NO; gestureRecognizers = <NSArray: 0x17424f180>; layer = <CALayer:
0x1702291a0>>
```

对于其他命令，读者可以自己阅读文档，逐一尝试。

3. 查看对象内存关系

LLDB 提供了一个强大的 Python 脚本，脚本的位置为 /Applications/Xcode.app/Contents/Shared Frameworks/ LLDB.framework/Versions/A/Resources/Python/lldb/macosx/heap.py。虽然路径中标明了是供 Mac OS X 使用的，但经测试，在 iOS 上也可以使用。该脚本的功能如下。

- 查看一个对象分配内存的堆栈，需要配合 MallocStackLogging 环境变量使用（`malloc_info -s`）。
- 得到内存指定类的实例对象（`obj_refs`）。
- 得到一个内存地址所有被引用的地方（`ptr_refs`）。
- 搜索内存中指定的 C 字符串（`cstr_ref`）。

可以通过如下命令来加载该脚本，或者给其取个别名（`command alias loadosx command script import lldb.macosx.heap` 命令）并写入 ~/.lldbinit 文件，以后就可以直接通过 loadosx 加载了。

```
(lldb) command script import lldb.macosx.heap
"malloc_info", "ptr_refs", "cstr_refs", "find_variable", and "objc_refs" commands have been
installed, use the "--help" options on these commands for detailed help.
```

为了查看某个对象内存分配的调用堆栈，需要在程序启动的环境变量中设置 MallocStack Logging。5.2.3 节讲到了如何使用 Xcode 调试第三方应用，在这里可以直接使用该例子在 Xcode 中添加环境变量。单击“Edit Scheme...”选项，如图 5-49 所示。

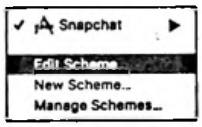


图 5-49 单击 Edit Scheme 选项

在环境变量中增加 MallocStackLogging 的值 1，如图 5-50 所示。或者先单击“Edit Scheme...”选项，再单击“Diagnostics”选项，在界面上勾选“Malloc Stack”选项，最后运行应用。

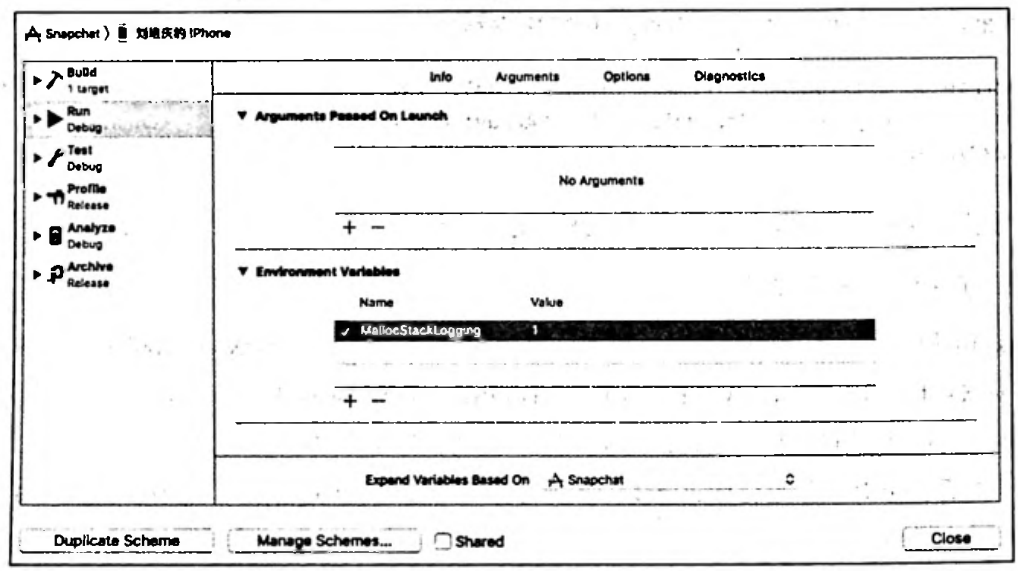


图 5-50 增加环境变量

进入 Snapchat 登录界面，单击 Xcode 调试工具栏上的“Debug Momery Graph”按钮，如图 5-51 所示，获取内存引用关系图。

在 Xcode 中可以看到如图 5-52 所示的界面，各部分说明如下。

- 界面左侧显示的是当前程序中所有在内存中的对象，在左下方的搜索框中可以搜索自己

感兴趣的对象。

- 界面的中间部分是当前选择的对象的引用关系图（只显示上层应用当前对象的结构），在图中可以看到对应的属性及名称。
- 界面右侧显示了该对象的详细信息，包括类名、地址、对象大小及继承关系。由于开启了 MallocStackLogging，还可以看到该对象内存申请的调用堆栈。

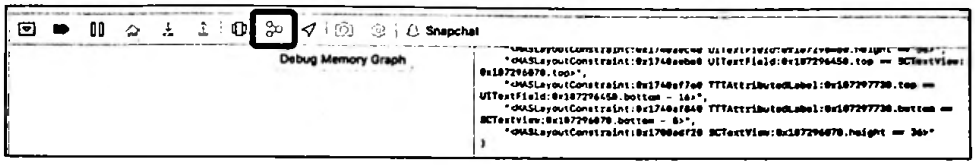


图 5-51　单击 Xcode 工具栏中的 Debug Memory Graph 按钮

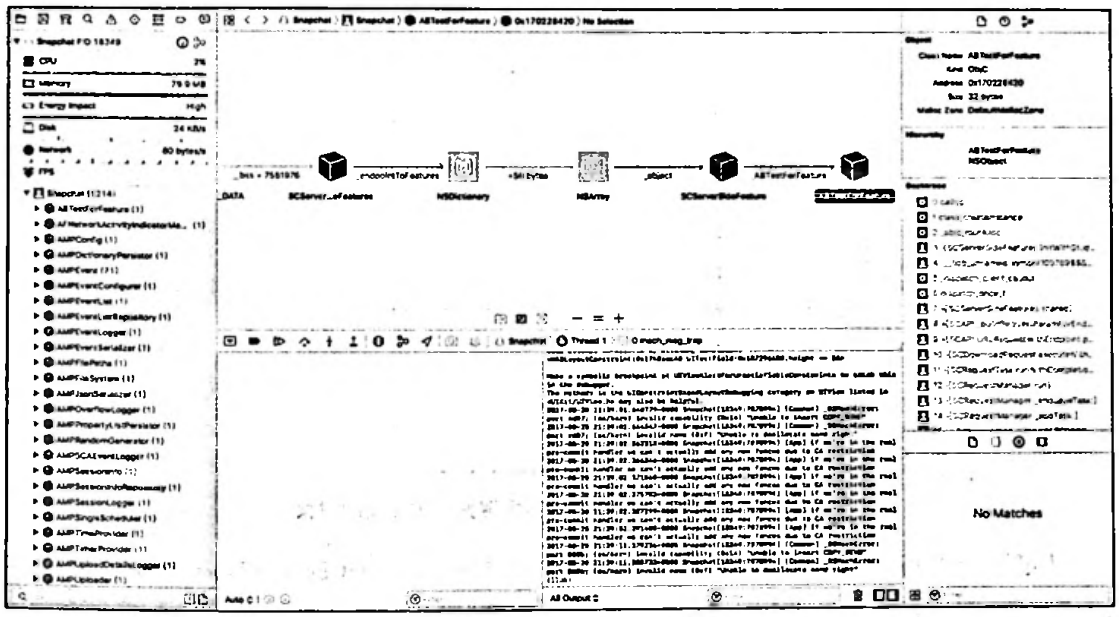


图 5-52　Xcode 显示的内存引用关系图

这个功能能够帮助我们快速分析当前程序的对象、引用关系及内存申请的堆栈，甚至获得更多有用的信息。例如，要查看当前内存中所有 UITextField 类型的对象实例，可以在左下方的搜索框中输入“UITextField”进行搜索。如图 5-53 所示，发现内存中有两个实例对象，应该就是当前界面的用户名和密码的输入框。单击某个对象，即可在内存引用关系图中看到引用该对象的其他对象。

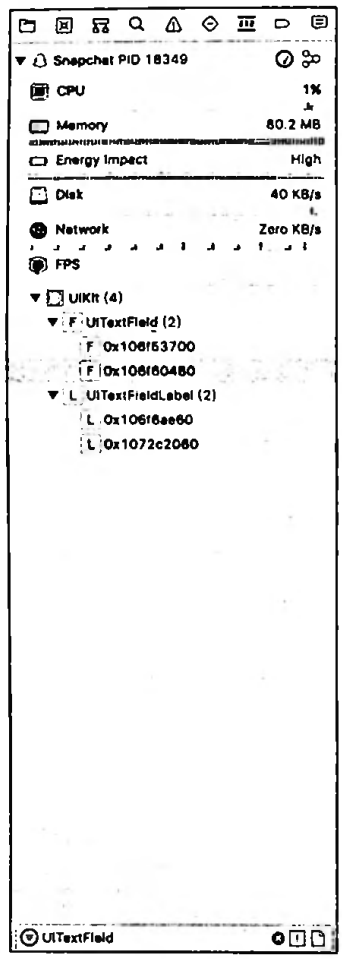


图 5-53 使用 Xcode 在内存中搜索指定类型的对象

如图 5-54 所示，不仅可以看到 SCLoginV2ViewController→SCLoginV2View→SCTextView→UITextField 的引用关系，还可以看到哪个属性引用了该对象———目了然。

在界面右侧还可以看到该对象内存分配的调用堆栈。如图 5-55 所示，可以看到该对象是由在 `-[SCLoginV2View initPasswordTextField]` 中调用 `-[SCTextView initWithFrame:]` 初始化的。这样，程序的逻辑和对象的初始化就展示得很明确了。

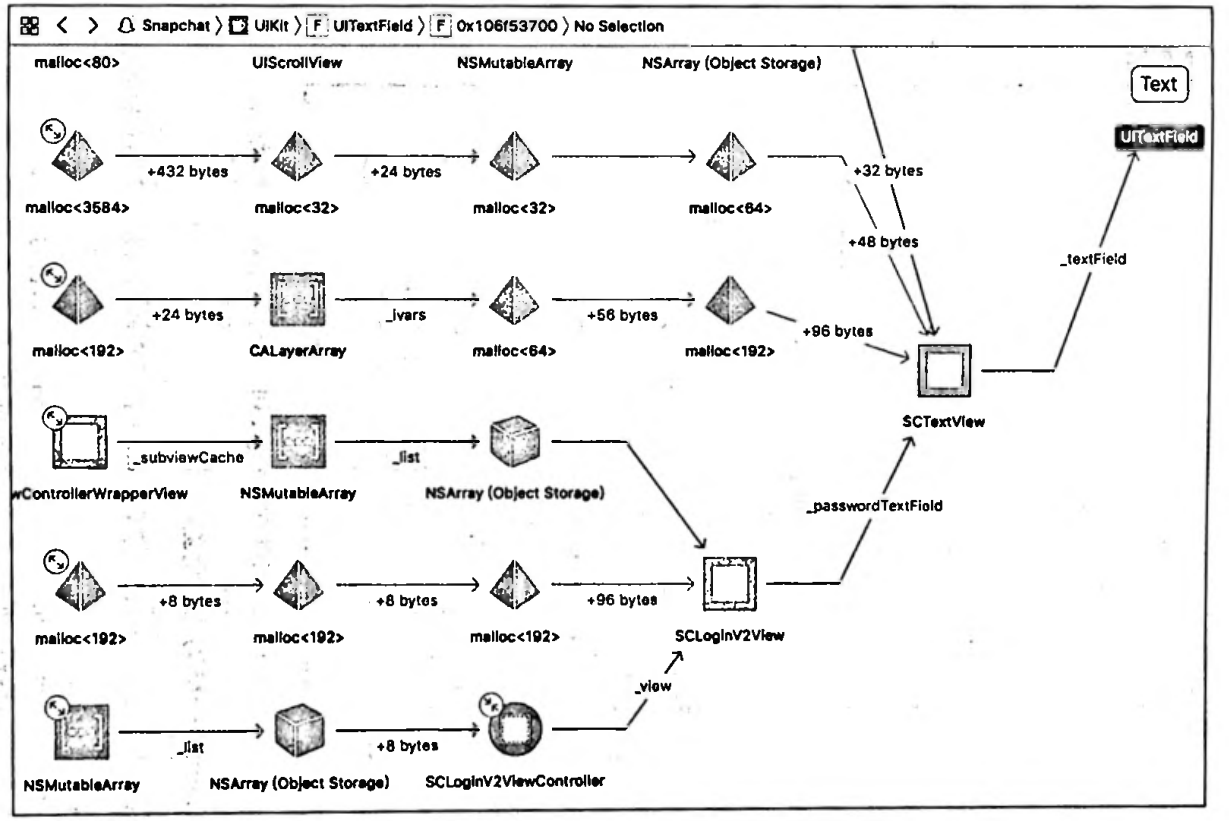


图 5-54　使用 Xcode 查看 UITextField 对象内存引用关系图

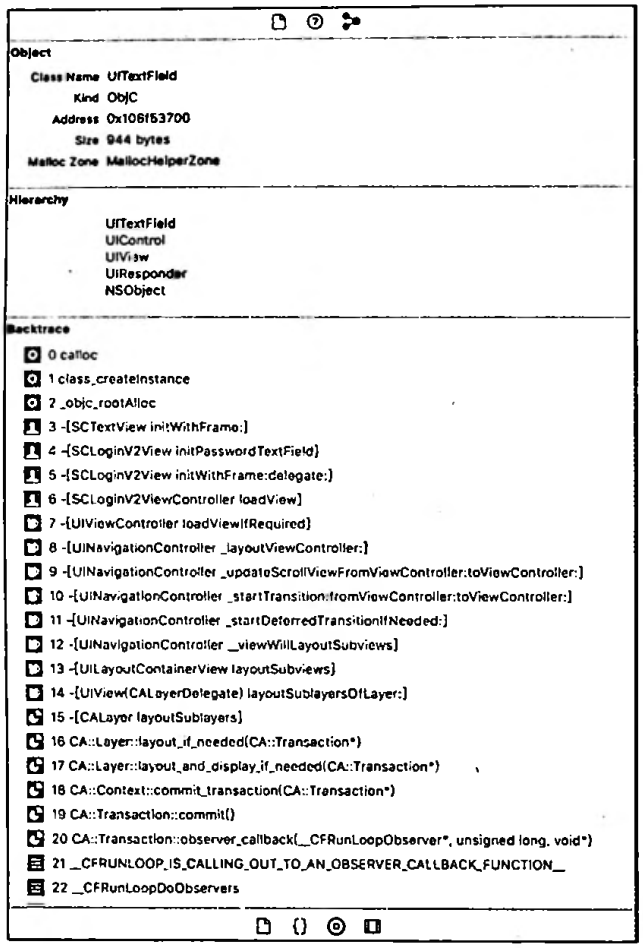


图 5-55 使用 Xcode 查看 UITextField 对象内存分配堆栈

4. 像 Cycript 一样动态调用函数

LLDB 也支持动态执行函数，像 Cycript 一样动态调用函数、打印对象等，示例如下。

```
(lldb) e @import UIKit
(lldb) e UIApplication *$app = [UIApplication sharedApplication];
(lldb) e UIWindow *$keyWindow = $app.keyWindow
(lldb) e UIViewController *$root = $keyWindow.rootViewController
(lldb) po $root
<NavigationController: 0x12c03d200>

(lldb) e [(SCButton *)0x12bd4b760 setTitle:@"AloneMonkey" forState:UIControlStateNormal]
```

```
(lldb) e (void)[CATransaction flush]
(lldb) c
Process 18474 resuming
```

这里的 0x12bd4b760 是从 Snapchat 首页获取的“登录”按钮对象的地址。

5. 其他技巧

Xcode 提供查看当前加载模块的功能。单击“Debug”→“Debug Workflow”→“Shared Libraries”选项，即可打开如图 5-56 所示的界面。

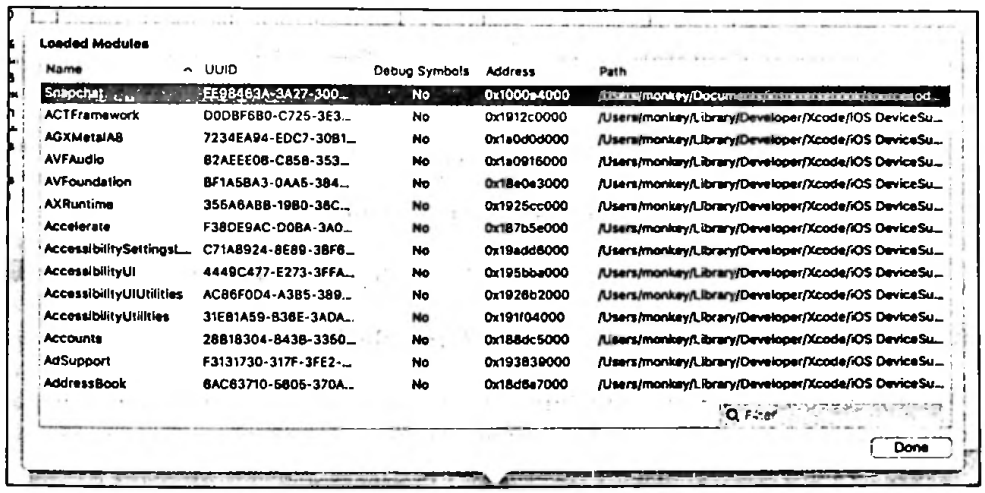


图 5-56　使用 Xcode 查看当前进程加载的模块

如果要查看某个地址指向的内存信息，只要单击“Debug”→“Debug Workflow”→“View Memory”选项，输入想要查看的内存地址，即可看到如图 5-57 所示的界面。

查看某个地址所在模块的信息。0x100dbdeac 是 [LoginRegisterViewController snapchatterButtonClicked] 函数在内存中的地址，对应文件中的偏移为 0x0000000100cd9eac，具体如下。

```
(lldb) b [LoginRegisterViewController snapchatterButtonClicked]
Breakpoint 1: where = Snapchat`-[LoginRegisterViewController snapchatterButtonClicked],
address = 0x0000000100dbdeac
(lldb) c
Process 18474 resuming
(lldb) image lookup -a 0x100dbdeac
      Address: Snapchat[0x0000000100cd9eac] (Snapchat.__TEXT.__text + 13449180)
      Summary: Snapchat`-[LoginRegisterViewController snapchatterButtonClicked]
```

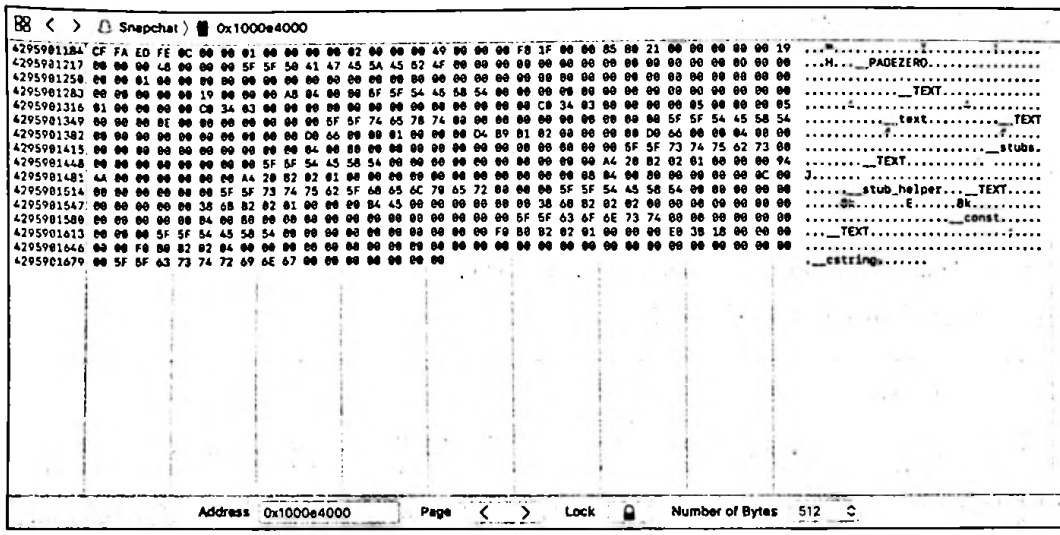


图 5-57 使用 Xcode 查看指定地址的内存信息

使用正则对某个类的所有方法下断点并跟踪打印调用参数，具体如下。

```
将 command regex bclass 's/(.+)/rb \[%1 /'写入文件~/.lldbinit

(lldb) bclass LoginRegisterViewController
Breakpoint 1: 26 locations.
(lldb) br command add 1
Enter your debugger command(s).  Type 'DONE' to end.
> po $x0.
> x/s $x1
> c
> DONE
```

在 Snapchat 模块中查看与 login 有关的符号信息，具体如下。

```
(lldb)  image lookup -rn login Snapchat
17 matches found in
/Users/monkey/Documents/iosreversebook/sourcecode/chapter-4/Snapchat/Build/Products/Deb
ug-iphoneos/Snapchat.app/Snapchat:
        Address: Snapchat[0x00000001004650f4] (Snapchat.__TEXT.__text + 4581924)
        Summary: Snapchat`-[SCRegisterV2BaseView loginTitleTopPadding]      Address:
Snapchat[0x000000010047f32c] (Snapchat.__TEXT.__text + 4688988)
        Summary: Snapchat`-[SCLoginV2ViewController _loginReactivationStatus:]
Address: Snapchat[0x0000000100819ea0] (Snapchat.__TEXT.__text + 8468432)
        Summary: Snapchat`-[SCLoginV2ViewController _loginReactivationConfirmationNeeded:]
```

```
Address: Snapchat[0x0000000100b90004] (Snapchat.__TEXT.__text + 12097844)
        Summary: Snapchat`-[SCLoginV2ViewController _loginDidFail:]        Address:
Snapchat[0x0000000100cde018] (Snapchat.__TEXT.__text + 13465928)
        Summary: Snapchat`-[SCLoginV2ViewController _loginVerificationNeeded:]
Address: Snapchat[0x0000000100e679c0] (Snapchat.__TEXT.__text + 15078128)
        Summary: Snapchat`-[SCLoginV2ViewController _loginTwoFANeeded:]        Address:
Snapchat[0x000000010051f7ac] (Snapchat.__TEXT.__text + 5345500)
        Summary: Snapchat`-[SCLoginV2ViewController _loginDidSucceed:]        Address:
Snapchat[0x0000000100e14d64] (Snapchat.__TEXT.__text + 14739092)
        Summary: Snapchat`+[User
loginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:remember
Device:fromDeepLink:width:height:successBlock:failureBlock:]        Address:
Snapchat[0x0000000101156394] (Snapchat.__TEXT.__text + 18152644)
        Summary: Snapchat`-[User loginInSession]        Address:
Snapchat[0x00000001006d8018] (Snapchat.__TEXT.__text + 7149896)
        Summary: Snapchat`-[User loginUsername]        Address: Snapchat[0x00000001007fb43c]
(Snapchat.__TEXT.__text + 8342892)
        Summary: Snapchat`-[User loginSuccessWithDictionary:]        Address:
Snapchat[0x0000000100d87f5c] (Snapchat.__TEXT.__text + 14162060)
        Summary: Snapchat`-[SCPhoneVerifyDeepLinkProcessor
initWithNavigationDelegate:loginRegisterNavigationDelegate:user:]        Address:
Snapchat[0x0000000100d66b64] (Snapchat.__TEXT.__text + 14025876)
        Summary: Snapchat`-[SCDeepLinkManager
initWithUser:withStories:navigationDelegate:loginRegisterNavigationDelegate:]
Address: Snapchat[0x00000001022d28c0] (Snapchat.__TEXT.__text + 36487664)
        Summary: Snapchat`-[SOJUAuthHumpbackCheckPasswordRequest loginIdentifierValue]
Address: Snapchat[0x00000001022d28b0] (Snapchat.__TEXT.__text + 36487648)
        Summary: Snapchat`-[SOJUAuthHumpbackCheckPasswordRequest loginIdentifierType]
Address: Snapchat[0x00000001022d1f68] (Snapchat.__TEXT.__text + 36485272)
        Summary: Snapchat`-[SOJUAuthHumpbackCheckPasswordRequest
initWithUserId:password:loginIdentifierType:loginIdentifierValue:]        Address:
Snapchat[0x00000001022d1ee4] (Snapchat.__TEXT.__text + 36485140)
        Summary: Snapchat`-[SOJUAuthHumpbackCheckPasswordRequest loginIdentifierTypeEnum]
```

更多的技巧有待读者自己探索和尝试。

5.3 Theos

通过前面的各种分析和调试，我们基本上可以定位某个功能点的实现函数和调用参数信息了，下一步就是修改对应的参数和函数内部的实现。幸运的是，已经有大牛为我们编写了注入、拦截函数的工具 Theos。Theos 最初是由 DHowett 开发的。DHowett 不再维护后，Adam Demasi（kirb）接手了该项目并完善了很多功能。

5.3.1 Theos 的安装

首先，在 Mac 上安装 dpkg、ldid 和 fakeroot，具体如下。

```
$ brew install ldid fakeroot
$ brew install --from-bottle
https://raw.githubusercontent.com/Homebrew/homebrew-core/7a4dabfc1a2acd9f01a1670fde4f00
94c4fb6ffa/Formula/dpkg.rb
$ brew switch dpkg 1.18.10   //如果当前已经安装了新的版本，使用该命令切换到 1.18.10
$ brew pin dpkg
```

- dpkg：用于管理 deb 包。因为新版的 dpkg 中会出现“error: obsolete compression type 'lzma'; use xz instead”错误，所以在这里需要指定一个较老的版本。
- ldid：用于签名。
- fakeroot：用于模拟 root 权限。

然后，从 GitHub 上拉取 Theos 代码，将代码安装到 /opt/theos 目录下，增加 recursive 参数以确保拉取了所有的子模块，并修改 /opt/theos 目录的所有者，具体如下。

```
sudo clone --recursive https://github.com/theos/theos.git /opt/theos
sudo chown -R $(id -u):$(id -g) /opt/theos
```

写入 $THEOS 的环境变量，编辑 ~/.zshrc 文件，具体如下。

```
export THEOS=/opt/theos
export PATH=$THEOS/bin:$PATH
```

到此为止，安装就完成了。

5.3.2 Theos 的基本应用

本节以拦截 Snapchat 的一个函数为例进行讲解。打开终端，输入“nic.pl”。若是刚设置的环境变量，需要先执行 source ~/.zshrc 命令。

```
➜  chapter-5 git:(master) X nic.pl
NIC 2.0 - New Instance Creator
------------------------------
  [1.] iphone/activator_event
  [2.] iphone/application_modern
  [3.] iphone/cydget
  [4.] iphone/flipswitch_switch
```

```
  [5.] iphone/framework
  [6.] iphone/ios7_notification_center_widget
  [7.] iphone/library
  [8.] iphone/notification_center_widget
  [9.] iphone/preference_bundle_modern
  [10.] iphone/tool
  [11.] iphone/tweak
  [12.] iphone/xpc_service
Choose a Template (required):
```

Theos 提供了很多模块来创建不同类型的项目。在这里选择 Tweak，输入“11”并按“Enter”键，进行一些初始化设置，就会在当前目录下生成一个项目，具体如下。

```
Choose a Template (required): 11
Project Name (required): TweakDemo
Package Name [com.yourcompany.tweakdemo]: com.alonemonkey.tweak
Author/Maintainer Name [monkey]: AloneMonkey
[iphone/tweak] MobileSubstrate Bundle filter [com.apple.springboard]:
com.toyopagroup.picaboo
[iphone/tweak] List of applications to terminate upon installation (space-separated, '-'
for none) [SpringBoard]: Snapchat
Instantiating iphone/tweak in tweakdemo/...
Done.
```

- Project Name：创建项目的名字。
- Package Name：创建包的唯一 ID。
- Author：创建者的名字。
- MobileSubstrate Bundle filter：需要注入的目标应用的 Bundle ID。
- applications to terminate：安装成功后杀掉指定的进程。

再来看一下生成的文件夹目录结构，具体如下。

```
➜  tweakdemo git:(master) ✗ tree
.
├── Makefile
├── Tweak.xm
├── TweakDemo.plist
└── control

0 directories, 4 files
```

- Makefile：makefile 文件。

- Tweak.xm：编写代码的文件。
- TweakDemo.plist：plist 文件，用于指定需要注入的目标文件的 Bundle ID。
- control：指定 deb 包的一些信息，包括名字、描述、版本号等。

例如，拦截 Snapchat class-dump 出来的 Manager 类中的 +[Manager performLoginWithUsernameOrEmail:password:preAuthToken:twoFAMethod:confirmReactivation:rememberDevice:fromDeepLink:onComplete:] 函数，可以编辑 Tweak.xm 文件，向其中写入如下内容。

```
%hook Manager

+ (void)performLoginWithUsernameOrEmail:(id)arg1 password:(id)arg2 preAuthToken:(id)arg3
twoFAMethod:(int)arg4 confirmReactivation:(_Bool)arg5 rememberDevice:(_Bool)arg6
fromDeepLink:(_Bool)arg7 onComplete:(id)arg8{
    %log;
    %orig;
}

%end
```

这是 Theos 为了方便开发而提供的语法。在编译过程中，会通过 logos.pl 转换成真正的代码，通过 %hook ClassName 指定需要 hook 的类，然后在里面写入需要 hook 的方法，并通过 %log 在该函数被调用时打印日志，使用 %orig 调用原来的方法。第 8 个参数是一个 block 类型的 CDUnknownBlockType，如果直接编译就会出错，所以将其改成了 id 类型。关于 Logs 的语法，读者可以参考 http://iphonedevwiki.net/index.php/Logos。

编写完成后，在终端输入“make”并按“Enter”键，编译代码，使用 make messages=yes 命令可以看到 Theos 内部处理的细节，具体如下。

```
➜  tweakdemo git:(master) X make
> Making all for tweak TweakDemo…
==> Preprocessing Tweak.xm…
==> Compiling Tweak.xm (armv7)…
==> Linking tweak TweakDemo (armv7)…
clang: warning: libstdc++ is deprecated; move to libc++ with a minimum deployment target
of iOS 7 [-Wdeprecated]
==> Preprocessing Tweak.xm…
==> Compiling Tweak.xm (arm64)…
==> Linking tweak TweakDemo (arm64)…
clang: warning: libstdc++ is deprecated; move to libc++ with a minimum deployment target
of iOS 7 [-Wdeprecated]
```

```
==> Merging tweak TweakDemo…
==> Signing TweakDemo…
```

然后，将其打包成 deb，具体如下。

```
➜  tweakdemo git:(master) X make package
> Making all for tweak TweakDemo…
make[2]: Nothing to be done for `internal-library-compile'.
> Making stage for tweak TweakDemo…
dm.pl: building package `com.alonemonkey.tweak:iphoneos-arm' in
`./packages/com.alonemonkey.tweak_0.0.1-1+debug_iphoneos-arm.deb'
```

打包之后，会在当前目录的 packages 文件夹中生成一个 deb 文件。最后，将 deb 文件安装到手机中。因为在执行如下命令后会出错，所以需要在环境变量中设置 THEOS_DEVICE_IP 和 THEOS_DEVICE_PORT。可以在 makefile 文件中设置，也可以直接将设置写入 ~./zshrc 文件。

```
➜  tweakdemo git:(master) X make install
==> Error: /Applications/Xcode.app/Contents/Developer/usr/bin/make install requires that
you set THEOS_DEVICE_IP in your environment.
==> Notice: It is also recommended that you have public-key authentication set up for root
over SSH, or you will be entering your password a lot.
make: *** [internal-install] Error 1

#向 makefile 文件写入环境变量，此处通过 USB 做了端口转发，也可以直接写 IP 地址和端口：
export THEOS_DEVICE_IP=localhost
export THEOS_DEVICE_PORT=2222

include $(THEOS)/makefiles/common.mk

TWEAK_NAME = TweakDemo
TweakDemo_FILES = Tweak.xm

include $(THEOS_MAKE_PATH)/tweak.mk

after-install::
    install.exec "killall -9 Snapchat"

➜  tweakdemo git:(master) X make install
==> Installing…
Selecting previously unselected package com.alonemonkey.tweak.
(Reading database ... 3542 files and directories currently installed.)
Preparing to unpack /tmp/_theos_install.deb ...
Unpacking com.alonemonkey.tweak (0.0.1-1+debug) ...
```

```
Setting up com.alonemonkey.tweak (0.0.1-1+debug) ...
install.exec "killall -9 Snapchat"
```

安装之后，就可以在设备的 /Library/MobileSubstrate/DynamicLibraries 目录下看到新安装的 dylib 和 plist 文件了。为了查看输出的日志信息，可以采用如下方法。

- 单击“Xcode”→“Window”→“Devices and Simulators”选项，选择对应的设置。单击左下角的三角形按钮，展开日志输出面板。
- 打开应用 /Applications/Utilities/Console.app，选择设备，在搜索框中输入进程名或关键字。
- 运行 `brew install libimobiledevice` 命令，安装 libimobiledevice。运行 `idevicesyslog | grep Snapchat` 命令，查看日志。

使用 `idevicesyslog` 命令查看日志的输出结果，具体如下。在第 2 行中可以看到加载动态库的信息。在出现问题时，首先要确认动态库有没有被目标进程加载，如果动态库本身有问题，在日志里面是可以看到加载的错误信息的。最后一行是 hook 输出的日志信息，可以看到目标方法的所有参数都打印出来了。

```
Aug 27 18:07:55 Monkey Snapchat[8279] <Notice>: MS:Notice: Injecting:
com.toyopagroup.picaboo [Snapchat] (1141.16)
Aug 27 18:07:55 Monkey Snapchat[8279] <Notice>: MS:Notice: Loading:
/Library/MobileSubstrate/DynamicLibraries/TweakDemo.dylib
Aug 27 18:07:55 Monkey Snapchat[8279] <Error>: assertion failed: 12B440: libxpc.dylib + 71820
[A4F17798-F3DE-3FBC-85E3-F569762F0EB9]: 0x7d
Aug 27 18:07:55 Monkey SpringBoard[5822] <Warning>: No valid 'aps-environment' entitlement
string found for application 'Snapchat': (null). Notifications will not be delivered.
Aug 27 18:08:01 Monkey Snapchat[8279] <Warning>: Unable to simultaneously satisfy constraints.
Aug 27 18:08:01 Monkey Snapchat[8279] <Warning>: Unable to simultaneously satisfy constraints.
Aug 27 18:08:17 Monkey Snapchat[8279] <Notice>: [TweakDemo] Tweak.xm:4 DEBUG: +[<Manager:
0x103dfb848> performLoginWithUsernameOrEmail:alonemonkey password:123456 preAuthToken:
twoFAMethod:0 confirmReactivation:0 rememberDevice:1 fromDeepLink:0 onComplete:(null)]
```

5.3.3 Theos 的高级应用

由于不同的需求会涉及不同的 Theos 高级使用方法，下面笔者将对插件开发中常见的高级 Theos 技巧一一进行讲解。

1. %c

如果要生成一个目标应用中某个类的对象，直接按下面这种方式写代码是会报错的。

```
@class SCLoginV2ViewController;

SCLoginV2ViewController* loginVC = [[SCLoginV2ViewController alloc] init];
NSLog(@"%@",loginVC);
```

在这里需要使用 OC 的 runtime 特性，通过 objc_getClass 动态获取类的方式。

Theos 提供了一个简单的方法，即直接写 %c(SCLoginV2ViewController)（其实就是调用 objc_getClass），具体如下。

```
@class SCLoginV2ViewController;

SCLoginV2ViewController* loginVC = [[%c(SCLoginV2ViewController) alloc] init];
NSLog(@"%@",loginVC);
```

2. arc

一般在写 Tweak 时，都会在 makefile 中开启 ARC，这样就不用自己手动进行内存管理了。相关命令如下，TweakDemo 是创建的项目的名字。

```
TweakDemo_CFLAGS = -fobjc-arc
```

但是，如果源文件只能使用 MRC，该怎么办呢？那就需要单独为该文件指定 MRC 的方式，编译如下。

```
TWEAK_NAME = TweakDemo
TweakDemo_FILES = Tweak.xm ZKSwizzle/ZKSwizzle.m
TweakDemo_CFLAGS = -fobjc-arc
ZKSwizzle/ZKSwizzle.m_CFLAGS = -fno-objc-arc
```

3. %new

如果想给一个类增加一个方法，可以通过 %new 实现，具体如下。

```
%new(v@:@)
-(void)printMessage:(NSString*) message{
    NSLog(@"[TestApp]: %@",message);
}
```

括号后面的内容是方法的签名，如果不写，Theos 会自动生成。“v” 表示 void 返回类型；第 1 个 “@” 表示调用者；“:” 表示 SEL；最后一个 “@” 表示参数类型。如果不知道该怎么写，

可以写一个 Demo 工程，然后调用如下函数来输出签名。

```
@implementation ViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    NSLog(@"%s", method_getTypeEncoding(class_getInstanceMethod([self class],
@selector(printMessage:))));
}

-(void)printMessage:(NSString*) message{
    NSLog(@"[TestApp]: %@",message);
}

@end
```

例如，在上面的例子中增加一个方法，具体如下。

```
@interface Manager

-(void)printMessage:(NSString*) message;

@end

%hook Manager

%new(v@:@)
-(void)printMessage:(NSString*) message{
    NSLog(@"[TestApp]: %@",message);
}

+ (void)performLoginWithUsernameOrEmail:(id)arg1 password:(id)arg2 preAuthToken:(id)arg3
twoFAMethod:(int)arg4 confirmReactivation:(_Bool)arg5 rememberDevice:(_Bool)arg6
fromDeepLink:(_Bool)arg7 onComplete:(id)arg8{
    SCLoginV2ViewController* loginVC = [[objc_getClass("SCLoginV2ViewController") alloc]
init];
    NSLog(@"%@",loginVC);

    [[self new] printMessage:@"test new method"];

   %log;
   %orig;
}
```

```
%end
```

需要注意的是，这里通过 %new 增加的是实例方法，而调用时是在类方法中调用的，对应的 self 就是 Manager 类本身而不是一个类的实例，所以，需要在调用 [self new] 生成一个类的实例之后再去调用。

4. layout

在很多情况下，编写 Tweak 时需要加入图片或者文件，然后在代码中使用它们，而这需要使用 layout 将文件放到指定的路径中。

在项目的根目录下新建一个 layout 文件夹，然后将需要指定的文件目录放在 layout 文件夹下新建的子文件夹中。例如，要将文件目录放到 /Library/Application\Support/TweakDemo/ 文件夹下面，可以通过如下方式新建文件夹。

```
➜  tweakdemo git:(master) X mkdir -p layout/Library/Application\ Support/TweakDemo/
```

新建一个 plist 文件，代码如下。

```
touch resource.plist

<?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>name</key>
    <string>AloneMonkey</string>
</dict>
</plist>
```

安装时会将 layout 文件夹下面的内容按照指定的目录放置。通过如下代码即可在项目中获取该文件的内容。

```
NSString* resourcePath = @"/Library/Application Support/TweakDemo/";

NSDictionary* resourceDict = [[NSDictionary alloc] initWithContentsOfFile:[resourcePath
stringByAppendingPathComponent:@"resource.plist"]];

NSLog(@"resourceDict:%@", resourceDict);
```

5. 集成第三方库

如果想在 Tweak 中集成第三方库，需要另外指定一些参数和搜索路径。以集成笔者编写的简单的静态库为例，一个是 .a 形式的，一个是 framework 形式的，两者都是静态库。将静态库文件、头文件和 framework 复制到根目录的 BookLib 和 BookFramework 下面，然后在 makefile 文件中指定编译参数，具体如下。

```
TweakDemo_CFLAGS = -fobjc-arc -I./BookLib/include -F./BookFramework

TweakDemo_LDFLAGS = -L./BookLib -F./BookFramework
TweakDemo_LIBRARIES = BookLib
TweakDemo_FRAMEWORKS = BookFramework
```

在代码中导入头文件，直接调用，具体如下。

```
#import <BookLib/BookLib.h>
#import <BookFramework/BookFramework.h>


BookLib* bookLib = [[BookLib alloc] init];
[bookLib printLib];

BookFramework* bookFramework = [[BookFramework alloc] init];
[bookFramework printFramwork];
```

6. logify.pl

有时我们想要监控某个方法或者某个类的函数调用及调用参数，但如果根据 class-dump 出来的头文件里面的方法和参数类型自己逐一输入的话，就太麻烦了。幸好 Theos 提供了根据头文件快速生成代码的工具 logify.pl，这个工具可以在 Theos 安装目录的 bin 目录下找到。例如，要监控从 Snapchat dump 出来的头文件里面的 Manager.h 文件中的所有方法，可以在 dump 出来的头文件目录下执行如下命令。

```
logify.pl Manager.h >> ../Manager.xm
```

命令执行之后，会生成一个 Manager.xm 文件，里面已经自动生成了所有方法的 hook 函数。将该文件复制到 TweakDemo 根目录下，并在 makefile 文件中将其加入编译文件，具体如下。

```
TweakDemo_FILES = Tweak.xm Manager.xm ZKSwizzle/ZKSwizzle.m
```

运行 make 指令，发现出现了很多错误，按如下方法即可一一解决。

- unknown type name 'CDUnknownBlockType'：不识别的类型。将 CDUnknownBlockType 换成 id，使用 Sublime Text，单击“Find”→“Replease”→“Replease All”选项，即可解决。
- unknown type name 'SCChatLoader'：这是一个类，使用 `@class SCChatLoader` 声明即可解决。
- expected selector for Objective-C method：删除 `.cxx_destruct` 方法即可解决。
- cast from pointer to smaller type 'unsigned int' loses information：64 位类型转换错误。将 `@" = 0x%x", (unsigned int)r` 替换成 `@" = 0x%lx", (uintptr_t)r` 即可解决。

运行 `make install` 指令，就能在日志中看到 Manager 方法的调用信息和参数了，类似如下输出。

```
Aug 28 22:43:56 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:72 DEBUG: -[<Manager:
0x174182ff0> user]
Aug 28 22:43:56 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:72 DEBUG:  = <User:
0x155e3fe80>
Aug 28 22:44:01 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:82 DEBUG: -[<Manager:
0x174182ff0> clearExpiredCacheInBackground]
Aug 28 22:44:06 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:32 DEBUG: -[<Manager:
0x174182ff0> sessionCount]
Aug 28 22:44:06 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:32 DEBUG:  = 2
Aug 28 22:44:21 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:17 DEBUG: +[<Manager:
0x103e07848> shared]
Aug 28 22:44:21 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:169 DEBUG: -[<Manager:
0x174182ff0> description]
Aug 28 22:44:21 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:169 DEBUG:  = <Manager:
0x174182ff0>
Aug 28 22:44:21 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:17 DEBUG:  = <Manager:
0x174182ff0>
Aug 28 22:44:21 Monkey Snapchat[8868] <Notice>: [TweakDemo] Manager.xm:162 DEBUG: -[<Manager:
0x174182ff0> compareSeqno:0]
```

5.4 MonkeyDev

因为 Theos 是跨平台的，所以以上采用的编写方式都是通过生成命令行参数来编译生成 Tweak 的。然而，在 Mac 平台习惯使用 Xcode 的开发者来可能不太适应这种方式。尽管 Spencer W.S. James 基于 Xcode 模块开发了 iOSOpenDev，但他已经很多年没有维护该项目了，因此，在新的 Xcode 和 Theos 中集成时都会有很多问题。笔者基于 iOSOpenDev 的思路开发了 MonkeyDev

（https://github.com/AloneMonkey/MonkeyDev），它不仅能完美地支持新版 Xcode 和新版 Theos，而且可以使用 Xcode 开发 CaptainHook Tweak、Logos Tweak 和 Command-line Tool。下面分别通过实例加以说明。

5.4.1 安装 MonkeyDev

在安装 MonkeyDev 之前，需要准备如下环境。

- 安装最新版的 Theos。
- 安装 ldid。
- 配置越狱设备免密码登录（或者安装 sshpass）。

这些内容在之前的章节都讲过，这里就不再细说了。然后就是安装 MonkeyDev，读者可以通过如下命令选择指定的 Xcode 进行安装。

```
sudo xcode-select -s /Applications/Xcode-beta.app
```

如果不指定，默认安装的 Xcode 如下。

```
xcode-select -p
```

运行安装脚本，命令如下。

```
git clone https://github.com/AloneMonkey/MonkeyDev.git
cd MonkeyDev/bin
sudo ./md-install
```

或者

```
sudo /bin/sh -c "$(curl -fsSL
https://raw.githubusercontent.com/AloneMonkey/MonkeyDev/master/bin/md-install)"
```

如果想要卸载 MonkeyDev，在 MonkeyDev/bin 目录下执行 `sudo ./md-uninstall` 命令即可。如果在 GitHub 上没有发布特殊说明，使用如下命令更新即可。

```
sudo ./md-update
```

安装 MonkeyDev 不会影响 Xcode 的编译和打包。安装之后，打开 Xcode，单击“File”→“New - Project...”选项，选择“iOS”选项，在窗口下方可以看到 MonkeyDev 提供的模块，如图 5-58 所示。选择项目类型，就可以创建对应的项目了。

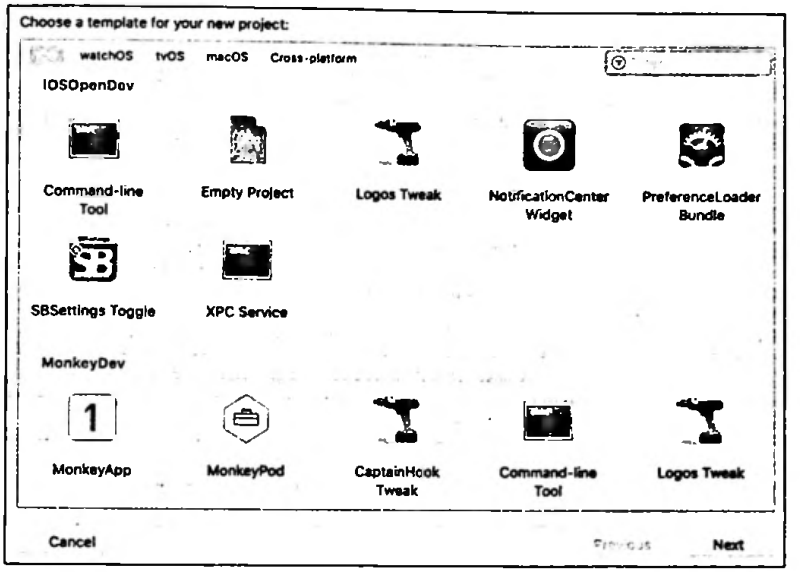


图 5-58　使用 Xcode 新建 iOS App 项目并选择 MonkeyDev 模板

5.4.2　Logos Tweak

选择 Logos Tweak 模板之后，新建的工程目录结构如图 5-59 所示，在这里要注意，Xcode 无法识别 .xm 文件格式，需要在 Xcode 界面右侧设置“Type”为“Objective-C++ Source”，然后重新打开工程。

- Logs.xm：和 Theos 中的文件一样，直接写 Logs 语法即可，在编译时会通过 logos.pl 转换成 Logs.mm。如果想了解其原理，可以查看这个文件。
- control：和 Theos 中的 control 文件一样，用于指定 deb 包的一些信息。
- Logs.plist：和 Theos 中的 plist 文件一样，用于指定需要注入的目标应用的过滤器。
- CydiaSubstrate：Logos Tweak 依赖于 CydiaSubstrate 库，项目会自动链接。

另外，在 Build Settings 列表的下面会提供一些选项，如图 5-60 所示。对这些设置的说明如表 5-1 所示。

表 5-1　Build Settings 设置项的意义

设 置 项	意　义
MonkeyDevBuildPackageOnAnyBuild	每次 build 都生成 deb 包
MonkeyDevClearUiCacheOnInstall	安装时清除 uicache

续表

设 置 项	意　义
MonkeyDevCopyOnBuild	build 时将 deb 包复制到设备的 /var/root/MonkeyDevBuilds/ 目录下
MonkeyDevDeviceIP	目标设备的 IP 地址，默认为 USB 连接，localhost
MonkeyDevDevicePort	目标设备的端口，默认为 22
MonkeyDevDevicePassword	目标设备的 ssh 登录密码，默认为空使用免密码登录
MonkeyDevInstallOnAnyBuild	每次 build 都将 deb 安装到设备
MonkeyDevInstallOnProfiling	点击 Profile 才将 deb 安装到设备
MonkeyDevPath	MonkeyDev 的安装路径，默认不用修改
MonkeyDevTheosPath	Theos 的安装路径
MonkeyDevKillProcessOnInstall	安装时杀掉指定的进程，填写进程名

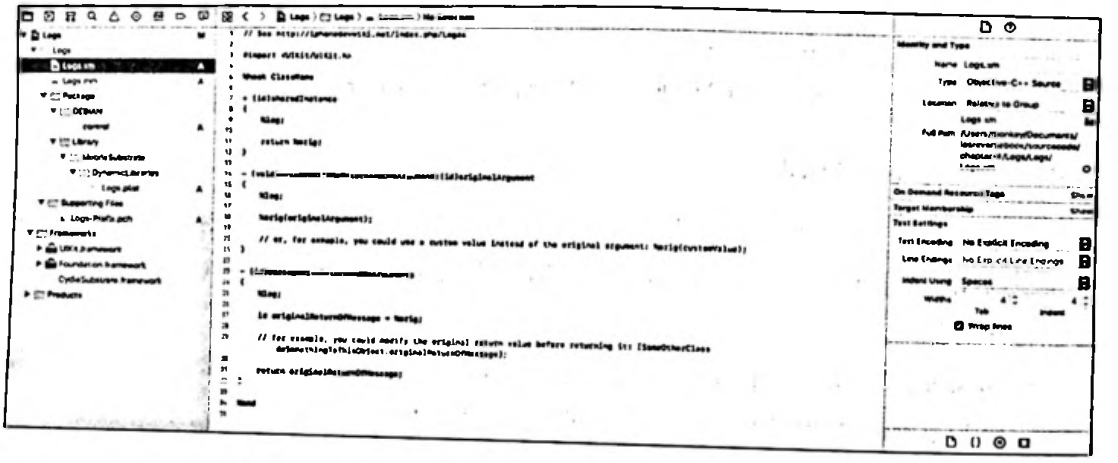


图 5-59　使用 Xcode 新建 Logos Tweak 模板

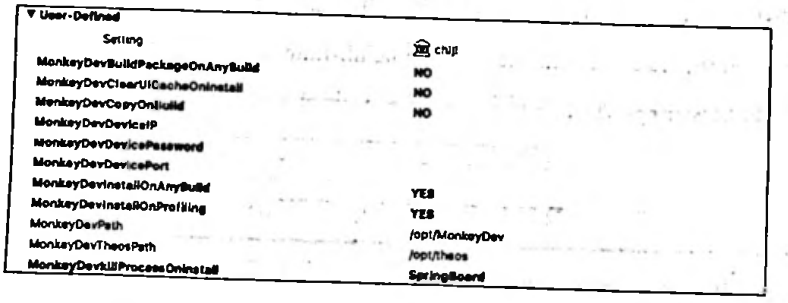


图 5-60　Xcode Build Settings 设置

其中，需要设置的是 MonkeyDevDeviceIP 和 MonkeyDevDevicePort。如果没有设置默认值，则需要读取环境变量中的 MonkeyDevDeviceIP 和 MonkeyDevDevicePort。如果免密码配置失败，需要配置 MonkeyDevDevicePassword 为远程设备的密码。

写好代码，按“Commonand + B”快捷键，就会将程序以 Debug 模式安装到手机上面。按“Command + Shift + I”快捷键，程序会以 Release 模式安装（但是在这种模式下不会输出 Log 信息）。

5.4.3　CaptainHook Tweak

CaptainHook Tweak 的项目目录结构如图 5-61 所示。与 Logos Tweak 不同的是，它直接通过导入 CaptainHook.h 头文件，然后使用里面的宏来进行 hook。关于 CaptainHook.h 的使用，读者可以参考 https://github.com/rpetrich/CaptainHook/wiki。它既不需要语法转换，也不依赖 CydiaSubstrate 动态库，但可能需要开发者熟悉它的 hook 写法。

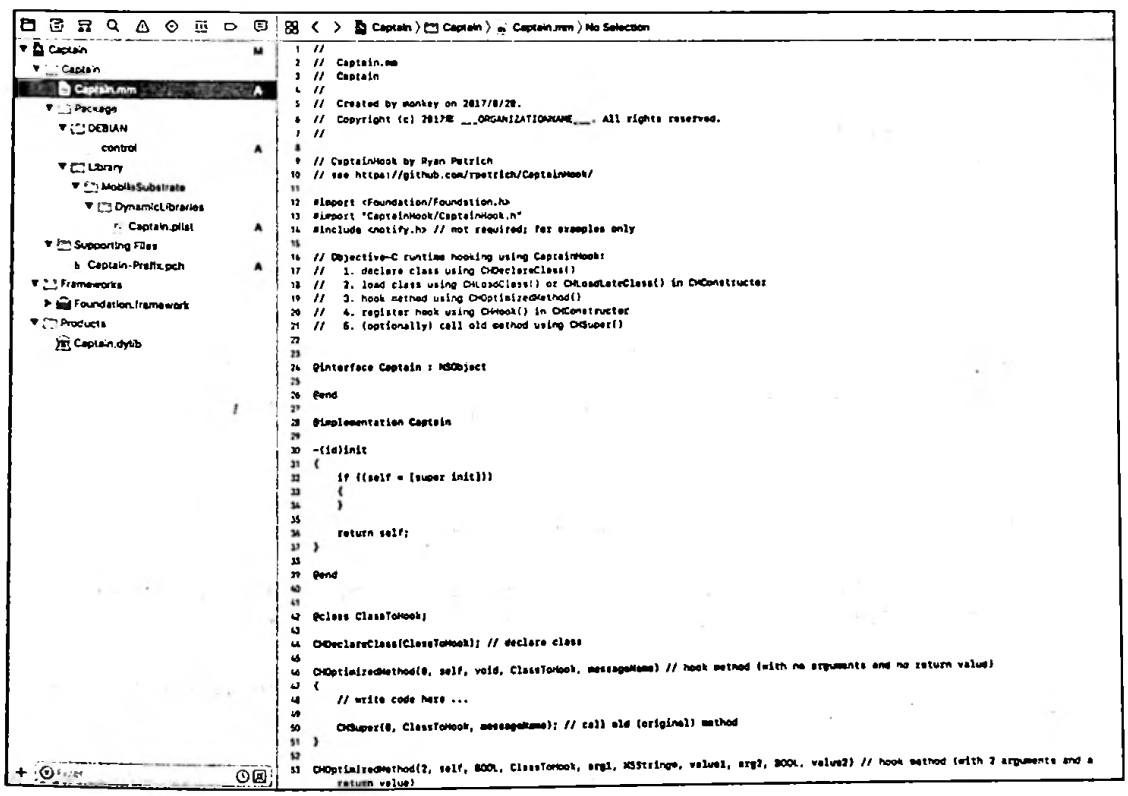


图 5-61　使用 Xcode 新建 CaptainHook Tweak 模板

如果要 hook 一个方法，可以按以下步骤实现。

①使用 CHDeclareClass 声明一个类。

②使用 CHLoadClass 或者 CHLoadLateClass 在 CHConstructor 中加载类。

③使用 CHOptimizedMethod hook 目标函数。

④使用 CHHook 在 CHConstructor 中注册 hook。

⑤使用 CHSuper() 函数调用 hook 的原函数实现。

为了讲解方便，这里以笔者编写的 TargetApp 作为 hook 的目标对象，根据上面的步骤来编写一个 hook 实例方法，代码如下。

```
CHDeclareClass(ViewController)

CHOptimizedMethod2(self, void, ViewController, instanceMethodUsername, NSString*, username,
password, NSString*, password){
    NSLog(@"hook instance method username: %@, password: %@", username, password);
    CHSuper2(ViewController, instanceMethodUsername, username, password, password);
}

CHConstructor{
    CHLoadLateClass(ViewController);
    CHHook2(ViewController, instanceMethodUsername, password);
}
```

这些宏都是什么意思呢？单击它们的定义，一层套一层，看都看晕了。这时，先用 Xcode 进行预处理，再将宏展开，就一目了然了。单击 “Product” → “Perform Action” → “Preprocess xxxx” 选项，可以对文件进行预处理，还可以将代码转换成汇编代码。查看预处理后的文件即可知道这些宏生成的代码，这可以帮助我们理解这些宏的作用。对上面例子的代码进行预处理，关键代码如下。

```
@class ViewController; static CHClassDeclaration_ ViewController$;

static void (*$ViewController_instanceMethodUsername$password$_super)(ViewController *
self, SEL _cmd, NSString* username, NSString* password);

static void $ViewController_instanceMethodUsername$password$_method(ViewController * self,
SEL _cmd, NSString* username, NSString* password);

__attribute__((always_inline)) static inline void
$ViewController_instanceMethodUsername$password$_register() {
```

```
        Method method = class_getInstanceMethod(ViewController$.class_,
@selector(instanceMethodUsername:password:));

        if (method) {

                $ViewController_instanceMethodUsername$password$_super =
(__typeof__($ViewController_instanceMethodUsername$password$_super))method_getImplement
ation(method);

                method_setImplementation(method,
(IMP)&$ViewController_instanceMethodUsername$password$_method);
    }
}

static void $ViewController_instanceMethodUsername$password$_method(ViewController * self,
SEL _cmd, NSString* username, NSString* password){

    CHDebugLog(@"hook instance method username: %@, password: %@", username, password);

    $ViewController_instanceMethodUsername$password$_super(self,
@selector(instanceMethodUsername:password:), username, password);
}

static __attribute__((constructor)) void CHConstructor22(){
    CHLoadClass_(&ViewController$, objc_getClass("ViewController"));
    $ViewController_instanceMethodUsername$password$_register();
}
```

对预处理后的代码进行分析，其实利用的是 OC 的 runtime 特性。下面介绍 CaptainHook 中常用的一些宏。

（1）hook 类方法

若使用 hook 类方法，只需要把 CHOptimizedMethod 改成 CHOptimizedClassMethod，相关代码如下。

```
CHOptimizedClassMethod2(self, void, ViewController, classMethodUsername, NSString*,
username, password, NSString*, password){
    CHDebugLog(@"hook class method username: %@, password: %@", username, password);
    CHSuper2(ViewController, classMethodUsername, username, password, password);
}
```

（2）获取私有属性

私有属性可以使用 CHIvar 获取，也可以通过 KVC 获取，相关代码如下。

```
NSString* ppassword = CHIvar(self,_password,__strong NSString*);
CHDebugLog(@"private password: %@", ppassword);
```

（3）增加属性

使用 CHPropertyRetainNonatomic 给类增加一个属性也是比较简单的，相关代码如下。

```
@interface ViewController : NSObject

@property (nonatomic, retain) NSString* newProperty;

@end

CHDeclareClass(ViewController)

CHPropertyRetainNonatomic(ViewController, NSString*, newProperty, setNewProperty);

CHConstructor{
    CHLoadLateClass(ViewController);
    CHHook0(ViewController, newProperty);
    CHHook1(ViewController, setNewProperty);
}
```

（4）增加方法

使用 CHDeclareMethod 可以给类增加一个方法，相关代码如下。

```
@interface ViewController : NSObject

@property (nonatomic, retain) NSString* newProperty;

- (void)addMethod:(NSString*) output;

@end

CHDeclareMethod1(void, ViewController, addMethod, NSString*, output){
    NSLog(@"add method %@", output);
}

CHOptimizedMethod2(self, void, ViewController, instanceMethodUsername, NSString*, username,
```

```
password, NSString*, password){
    [self addMethod:@"output"];
    CHSuper2(ViewController, instanceMethodUsername, username, password, password);
}
```

读者可以自己看一下这些宏，然后尝试做一遍，做完后就基本没有问题了。

5.4.4　Command-line Tool

使用 Command-line Tool 模板新建的工程的目录结构如图 5-62 所示，其中主要的是一个 main.c 源文件。安装成功后，在手机终端输入对应的命令即可，具体如下。

```
Monkey:~ root# Commandline
Hello, World!
```

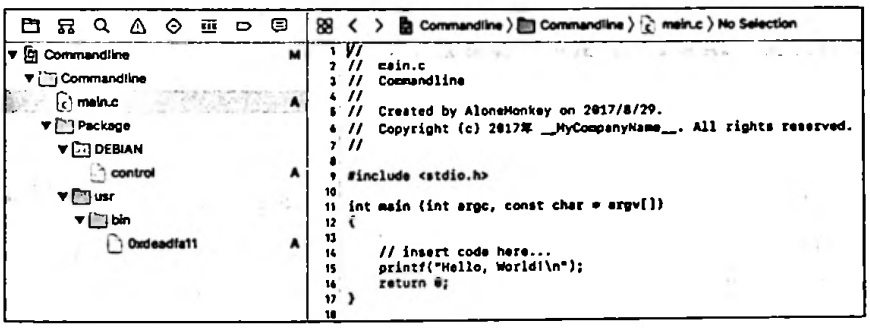


图 5-62　使用 Xcode 新建 Command-line Tool 模板

第 6 章　逆向进阶

6.1　程序加载

在编写一个应用程序时，我们看到的入口函数都是 main.m 里面的 main 函数，因此我们一般会以为程序是从这里开始执行的。其实不然，程序在执行 main 函数之前已经执行了 +load 和 constructor 构造函数。接下来，让我们一起看看在 main 函数执行之前都发生了什么。

6.1.1　dyld 简介

程序在运行时会依赖很多系统动态库。系统动态库会通过动态库加载器（默认是 /usr/lib/dyld）加载到内存中，系统内核在做好启动程序的准备工作之后就会将工作交给 dyld。由于很多程序都需要使用系统动态库，不可能在每个程序加载时都去加载所有的系统动态库，为了优化程序启动速度和利用动态库缓存，iOS 系统采用了共享缓存技术。dyld 缓存在 iOS 系统中，位于 /System/Library/Caches/com.apple.dyld/ 目录下，按不同的架构保存了不同的文件。在笔者的 iPhone 5s 中就有 dyld_shared_cache_arm64 和 dyld_shared_cache_armv7s 两个文件。要分析系统原始的二进制文件，就需要从 dyld 缓存文件中将系统库的原始二进制文件提取出来，方法如下。

1. dsc_extractor

访问 https://opensource.apple.com/tarballs/dyld/，下载源代码。笔者下载的是 433.5 版本的代码（当前为 dyld2，不包括对 dyld3 的分析）。在 launch-cache 文件夹下有 dsc_extractor.cpp 文件，修改该文件 651 行中的“#if 0”为“#if 1”。使用如下命令编译生成 dsc_extractor 文件，使用生成的文件对 cache 文件进行解压。

```
clang++ -o dsc_extractor dsc_extractor.cpp dsc_iterator.cpp
./dsc_extractor ../dyld_shared_cache_arm64 ../arm64
```

2. jtool

使用如下命令即可导出对应的模块。

```
./jtool -extract UIKit ../dyld_shared_cache_arm64
```

3. dyld_cache_extract

Mac 界面导出工具的下载地址是 https://github.com/macmade/dyld_cache_extract。

除了手动提取，新设备在连接到 Xcode 时也会自动提取系统库到 ~/Library/Developer/Xcode/iOS DeviceSupport 目录下。当连接多个设备之后，这个目录会变得特别大，因此可以定期清理这个目录。共享缓存在系统启动后被加载到内存中，当有新的程序加载时会先到共享缓存里面寻找。如果找到，就直接将共享缓存中的地址映射到目标进程的内存地址空间，极大地提高了加载效率。

6.1.2　dyld 加载流程

要想知道 +load 和 constructor 是在什么时候调用的，就需要分析 dyld 加载 Mach-O 文件的流程。dyld 的代码可以从苹果开源网站下载（http://opensource.apple.com/tarballs/dyld，笔者下载的是 433.5 版本）。从 dyldStartup.s 文件开始执行，其中用汇编实现的 __dyld_start 方法里面调用了 dyldbootstrap::start()方法，然后调用了 dyld 的 main 函数。直接转到 dyld.cpp 文件的 main 函数进行分析，具体如下。

```
uintptr_t
_main(const macho_header* mainExecutableMH, uintptr_t mainExecutableSlide,
                int argc, const char* argv[], const char* envp[], const char* apple[],
                uintptr_t* startGlue)
{
    uintptr_t result = 0;
        //保存执行文件头部，后续可以根据头部访问其他信息
        sMainExecutableMachHeader = mainExecutableMH;
    ......
    //设置上下文信息
        setContext(mainExecutableMH, argc, argv, envp, apple);

        //Pickup the pointer to the exec path.
        //获取可执行文件的路径
        sExecPath = _simple_getenv(apple, "executable_path");
```

```
    //<rdar://problem/13868260> Remove interim apple[0] transition code from dyld
    if (!sExecPath) sExecPath = apple[0];

    //将相对路径转换成绝对路径
    if ( sExecPath[0] != '/' ) {
            //have relative path, use cwd to make absolute
            char cwdbuff[MAXPATHLEN];
        if ( getcwd(cwdbuff, MAXPATHLEN) != NULL ) {
                    //maybe use static buffer to avoid calling malloc so early...
                    char* s = new char[strlen(cwdbuff) + strlen(sExecPath) + 2];
                    strcpy(s, cwdbuff);
                    strcat(s, "/");
                    strcat(s, sExecPath);
                    sExecPath = s;
            }
    }
    //Remember short name of process for later logging
    //获取文件的名字
    sExecShortName = ::strrchr(sExecPath, '/');
    if ( sExecShortName != NULL )
            ++sExecShortName;
    else
            sExecShortName = sExecPath;

    //配置进程是否受限
 configureProcessRestrictions(mainExecutableMH);

 ......
 {
            //检查设置环境变量
            checkEnvironmentVariables(envp);
            //如果 DYLD_FALLBACK 为 nil，将其设置为默认值
            defaultUninitializedFallbackPaths(envp);
    }
    //如果设置了 DYLD_PRINT_OPTS 环境变量，则打印参数
    if ( sEnv.DYLD_PRINT_OPTS )
            printOptions(argv);
    //如果设置了 DYLD_PRINT_ENV 环境变量，则打印环境变量
    if ( sEnv.DYLD_PRINT_ENV )
            printEnvironmentVariables(envp);
   //获取当前运行架构的信息
   getHostInfo(mainExecutableMH, mainExecutableSlide);
......
//instantiate ImageLoader for main executable
```

```
    //加载可执行文件并生成一个 ImageLoader 实例对象
    sMainExecutable = instantiateFromLoadedImage(mainExecutableMH, mainExecutableSlide,
sExecPath);
    gLinkContext.mainExecutable = sMainExecutable;
    gLinkContext.mainExecutableCodeSigned = hasCodeSignatureLoadCommand(mainExecutableMH);
    //load shared cache
    //检查共享缓存是否开启，在 iOS 中必须开启
    checkSharedRegionDisable();
    #if DYLD_SHARED_CACHE_SUPPORT
    if ( gLinkContext.sharedRegionMode != ImageLoader::kDontUseSharedRegion ) {
        //检查共享缓存是否映射到了共享区域
        mapSharedCache();
    }
    ......
    //Now that shared cache is loaded, setup an versioned dylib overrides
    #if SUPPORT_VERSIONED_PATHS
    //检查库的版本是否有更新，如果有则覆盖原有的
    checkVersionedPaths();
    #endif
    ......
    //加载所有 DYLD_INSERT_LIBRARIES 指定的库
    //load any inserted libraries
    if    ( sEnv.DYLD_INSERT_LIBRARIES != NULL ) {
        for (const char* const* lib = sEnv.DYLD_INSERT_LIBRARIES; *lib != NULL; ++lib)
            loadInsertedDylib(*lib);
    }
    //record count of inserted libraries so that a flat search will look at
    //inserted libraries, then main, then others.
    sInsertedDylibCount = sAllImages.size()-1;
    ......
    //链接主程序
    link(sMainExecutable, sEnv.DYLD_BIND_AT_LAUNCH, true, ImageLoader::RPathChain(NULL,
NULL), -1);
    sMainExecutable->setNeverUnloadRecursive();
    if ( sMainExecutable->forceFlat() ) {
        gLinkContext.bindFlat = true;
        gLinkContext.prebindUsage = ImageLoader::kUseNoPrebinding;
    }

    //链接插入的动态库
    //link any inserted libraries
    //do this after linking main executable so that any dylibs pulled in by inserted
    //dylibs (e.g. libSystem) will not be in front of dylibs the program uses
    if ( sInsertedDylibCount > 0 ) {
```

```
        for(unsigned int i=0; i < sInsertedDylibCount; ++i) {
            ImageLoader* image = sAllImages[i+1];
            link(image, sEnv.DYLD_BIND_AT_LAUNCH, true, ImageLoader::RPathChain(NULL, NULL),
-1);
            image->setNeverUnloadRecursive();
        }
        //only INSERTED libraries can interpose
        //register interposing info after all inserted libraries are bound so chaining works
        for(unsigned int i=0; i < sInsertedDylibCount; ++i) {
            ImageLoader* image = sAllImages[i+1];
            //注册符号插入
            image->registerInterposing();
        }
    }

    //<rdar://problem/19315404> dyld should support interposition even without
DYLD_INSERT_LIBRARIES
    for (long i=sInsertedDylibCount+1; i < sAllImages.size(); ++i) {
        ImageLoader* image = sAllImages[i];
        if ( image->inSharedCache() )
            continue;
        image->registerInterposing();
    }
    ......
    //apply interposing to initial set of images
    for(int i=0; i < sImageRoots.size(); ++i) {
        //应用符号插入
        sImageRoots[i]->applyInterposing(gLinkContext);
    }
    gLinkContext.linkingMainExecutable = false;

    //<rdar://problem/12186933> do weak binding only after all inserted images linked
    //弱符号绑定
    sMainExecutable->weakBind(gLinkContext);
    ......
    //notify any montoring proccesses that this process is about to enter main()
    notifyMonitoringDyldMain();

    //find entry point for main executable
    //寻找入口并执行
    //LC_MAIN
    result = (uintptr_t)sMainExecutable->getThreadPC();
    if ( result != 0 ) {
        //main executable uses LC_MAIN, needs to return to glue in libdyld.dylib
```

```
        if ( (gLibSystemHelpers != NULL) && (gLibSystemHelpers->version >= 9) )
            *startGlue = (uintptr_t)gLibSystemHelpers->startGlueToCallExit;
        else
            halt("libdyld.dylib support not present for LC_MAIN");
    }
    else {
        //main executable uses LC_UNIXTHREAD, dyld needs to let "start" in program set up
for main()
        //LC_UNIXTHREAD
        result = (uintptr_t)sMainExecutable->getMain();
        *startGlue = 0;
    }
    .....
}
```

通过代码可以知道，dyld 的加载流程主要包括如下 9 个步骤。

01　设置上下文信息，配置进程是否受限

首先，调用 setContext，设置上下文信息，包括后面需要调用的函数及传入参数。然后，调用 configureProcessRestrictions，设置进程是否受限。这个函数里面有 __IPHONE_OS_VERSION_MIN_REQUIRED 和 MAC_OS_X_VERSION_MIN_REQUIRED 两个宏，后者在 iOS 中未定义（测试版本为 iOS 10.3.2），所以只需要分析 __IPHONE_OS_VERSION_MIN_REQUIRED 里面的代码。在其中为 sEnvMode 设置不同的 Mode，默认是 envNone（受限模式，忽略环境变量）。在设置 get_task_allow 权限和开发内核时会设置为 envAll，但只要设置了 uid 或 gid，就会变成受限模式。相关代码如下。

```
sEnvMode = envNone;
gLinkContext.requireCodeSignature = true;
if ( csops(0, CS_OPS_STATUS, &flags, sizeof(flags)) != -1 ) {
    if ( flags & CS_ENFORCEMENT ) {
        //设置 get_task_allow 权限，在 Debug 模式下不受限
        if ( flags & CS_GET_TASK_ALLOW ) {
            //Xcode built app for Debug allowed to use DYLD_* variables
            sEnvMode = envAll;
        }
        else {
            //Development kernel can use DYLD_PRINT_* variables on any FairPlay encrypted app
            uint32_t secureValue = 0;
            size_t   secureValueSize = sizeof(secureValue);
            if ( (sysctlbyname("kern.secure_kernel", &secureValue, &secureValueSize, NULL,
0) == 0) && (secureValue == 0) && isFairPlayEncrypted(mainExecutableMH) ) {
                sEnvMode = envPrintOnly;
```

```
            }
        }
    }
    else {
        //Development kernel can run unsigned code
        sEnvMode = envAll;
        gLinkContext.requireCodeSignature = false;
    }
}
//设置 uid 或 gid 受限
if ( issetugid() ) {
    sEnvMode = envNone;
}
```

因此，在新版的 iOS 系统（iOS 10.3.2 及以上版本）中设置 Other Linker Flags 为 -Wl,-sectcreate,__RESTRICT,__restrict,/dev/null，即使增加 __RESTRICT,__restrict section，也不能设置成受限模式，也就是说，阻止不了 DYLD_INSERT_LIBRARIES 注入。

02 配置环境变量，获取当前运行架构

调用 checkEnvironmentVariables，根据环境变量设置相应的值。如果 sEnvMode 为 envNone 就直接跳过，否则调用 processDyldEnvironmentVariable 处理并设置环境变量，代码如下。

```
static void checkEnvironmentVariables(const char* envp[])
{
    if ( sEnvMode == envNone )
        return;
    const char** p;
    for(p = envp; *p != NULL; p++) {
        const char* keyEqualsValue = *p;
        if ( strncmp(keyEqualsValue, "DYLD_", 5) == 0 ) {
            const char* equals = strchr(keyEqualsValue, '=');
            if ( equals != NULL ) {
                strlcat(sLoadingCrashMessage, "\n", sizeof(sLoadingCrashMessage));
                strlcat(sLoadingCrashMessage, keyEqualsValue,
sizeof(sLoadingCrashMessage));
                const char* value = &equals[1];
                const size_t keyLen = equals-keyEqualsValue;
                char key[keyLen+1];
                strncpy(key, keyEqualsValue, keyLen);
                key[keyLen] = '\0';
                if ( (sEnvMode == envPrintOnly) && (strncmp(key, "DYLD_PRINT_", 11) !=
0) )
```

```
                continue;
            processDyldEnvironmentVariable(key, value, NULL);
        }
    }
    else if ( strncmp(keyEqualsValue, "LD_LIBRARY_PATH=", 16) == 0 ) {
        const char* path = &keyEqualsValue[16];
        sEnv.LD_LIBRARY_PATH = parseColonList(path, NULL);
    }
}

#if SUPPORT_LC_DYLD_ENVIRONMENT
    checkLoadCommandEnvironmentVariables();
#endif // SUPPORT_LC_DYLD_ENVIRONMENT

#if SUPPORT_ROOT_PATH
    // <rdar://problem/11281064> DYLD_IMAGE_SUFFIX and DYLD_ROOT_PATH cannot be used
together
    if ( (gLinkContext.imageSuffix != NULL) && (gLinkContext.rootPaths != NULL) ) {
        dyld::warn("Ignoring DYLD_IMAGE_SUFFIX because DYLD_ROOT_PATH is used.\n");
        gLinkContext.imageSuffix = NULL;
    }
#endif
}
```

调用 getHostInfo，获取当前运行的架构的信息。从下面的代码中可以看到，如果设置了 DYLD_PRINT_OPTS 和 DYLD_PRINT_ENV，将打印参数和当前的环境变量。

```
//如果设置了 DYLD_PRINT_OPTS 环境变量，则打印参数
if ( sEnv.DYLD_PRINT_OPTS )
    printOptions(argv);
//如果设置了 DYLD_PRINT_ENV 环境变量，则打印环境变量
if ( sEnv.DYLD_PRINT_ENV )
    printEnvironmentVariables(envp);
```

新建一个 iOS App 项目，单击“Edit Scheme”选项，在环境变量中增加这两个环境变量，如图 6-1 所示。运行项目，就能得到如下的日志输出。

```
opt[0] =
"/var/containers/Bundle/Application/3E7A1EAE-0E84-447F-B89D-0D279152CF61/AppStart.app/A
ppStart"
TMPDIR=/private/var/mobile/Containers/Data/Application/CF2160BE-69E3-4B06-B468-5B711373
66A6/tmp
__CF_USER_TEXT_ENCODING=0x1F5:0:0
```

```
OS_ACTIVITY_DT_MODE=YES
HOME=/private/var/mobile/Containers/Data/Application/CF2160BE-69E3-4B06-B468-5B71137366
A6
SHELL=/bin/sh
DYLD_PRINT_TO_STDERR=YES
CFFIXED_USER_HOME=/private/var/mobile/Containers/Data/Application/CF2160BE-69E3-4B06-B4
68-5B71137366A6
PATH=/usr/bin:/bin:/usr/sbin:/sbin
LOGNAME=mobile
NSUnbufferedIO=YES
XPC_SERVICE_NAME=UIKitApplication:com.alonemonkey.AppStart[0xde8b][63]
CLASSIC=0
DYLD_INSERT_LIBRARIES=/Developer/usr/lib/libBacktraceRecording.dylib:/Developer/Library
/PrivateFrameworks/DTDDISupport.framework/libViewDebuggerSupport.dylib
DYLD_PRINT_ENV=
USER=mobile
DYLD_PRINT_OPTS=
XPC_FLAGS=0x1
DYLD_LIBRARY_PATH=/usr/lib/system/introspection
```

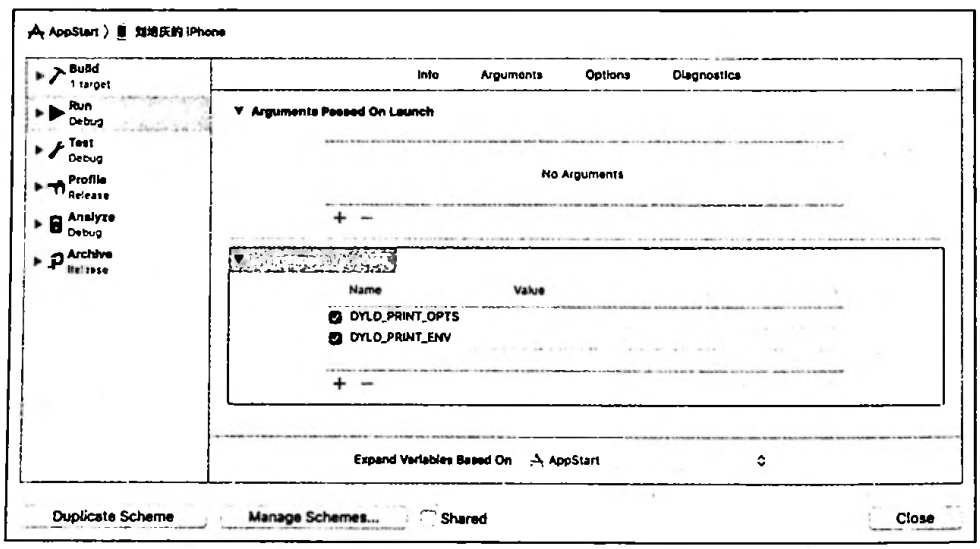


图 6-1 使用 Xcode 设置环境变量

03 加载可执行文件，生成一个 ImageLoader 实例对象

可以调用 instantiateFromLoadedImage 函数来实例化一个 ImageLoader 对象。该函数先调用 isCompatibleMachO 来判断文件的架构是否和当前的架构兼容，然后调用 ImageLoaderMachO::

instantiateMainExecutable 来加载文件生成实例，并将 image 添加到全局 sAllImages 中，更新 mapped range，同时在 addImage 函数的后面可以看到，如果设置了 DYLD_PRINT_LIBRARIES，将输出当前加载的 image。

instantiateMainExecutable 函数首先调用 sniffLoadCommands 获取 Load Command 的相关信息，并对其进行各种校验。根据是否存在 LC_DYLD_INFO 和 LC_DYLD_INFO_ONLY，可以判断当前 macho 是不是压缩的。然后，根据当前 macho 是普通类型还是压缩的，使用不同的 ImageLoaderMachO 子类进行初始化。相关代码如下。

```
//从可执行文件创建 image
ImageLoader* ImageLoaderMachO::instantiateMainExecutable(const macho_header* mh, uintptr_t
slide, const char* path, const LinkContext& context)
{
    //dyld::log("ImageLoader=%ld, ImageLoaderMachO=%ld, ImageLoaderMachOClassic=%ld,
ImageLoaderMachOCompressed=%ld\n",
    //  sizeof(ImageLoader), sizeof(ImageLoaderMachO), sizeof(ImageLoaderMachOClassic),
sizeof(ImageLoaderMachOCompressed));
    bool compressed;
    unsigned int segCount;
    unsigned int libCount;
    const linkedit_data_command* codeSigCmd;
    const encryption_info_command* encryptCmd;
    //获取 Load Command 的相关信息，并对其进行各种校验
    sniffLoadCommands(mh, path, false, &compressed, &segCount, &libCount, context,
&codeSigCmd, &encryptCmd);
    // instantiate concrete class based on content of load commands
    //根据不同的文件类型调用不同的 ImageLoaderMachO 进行初始化
    if ( compressed )
        return ImageLoaderMachOCompressed::instantiateMainExecutable(mh, slide, path,
segCount, libCount, context);
    else
#if SUPPORT_CLASSIC_MACHO
        return ImageLoaderMachOClassic::instantiateMainExecutable(mh, slide, path,
segCount, libCount, context);
#else
        throw "missing LC_DYLD_INFO load command";
#endif
}
```

以 ImageLoaderMachOCompressed::instantiateMainExecutable 为例，先用 instantiateStart 进行实例化，然后调用 instantiateFinish 来处理其他的 Load Command。如果设置了 DYLD_PRINT_

SEGMENTS，最后将会打印 segment 加载的详细信息。打印信息如下。

```
dyld: Main executable mapped
/var/containers/Bundle/Application/71DE881B-1564-4A7E-AFE2-407EE4241898/AppStart.app/Ap
pStart
        __PAGEZERO at 0x00000000->0x100000000
            __TEXT at 0x1000D0000->0x1000D8000
            __DATA at 0x1000D8000->0x1000DC000
        __LINKEDIT at 0x1000DC000->0x1000E4000
......
```

04 检查共享缓存是否映射到了共享区域

首先，使用 checkSharedRegionDisable 检查是否开启了共享缓存（在 iOS 中必须开启）。然后，调用 mapSharedCache() 函数，将共享缓存映射到共享区域。后者先通过 shared_region_check_np 检查缓存是否已经映射。如果已经在共享区域中，则更新 sharedCacheSlide 和 sharedCache UUID，否则就调用 openSharedCacheFile 打开共享缓存文件，一般是 /System/Library/Caches/com.apple.dyld/dyld_shared_cache_arm64 或其他架构的文件，检验缓存文件。最后，使用 shared_region_map_and_slide_np 完成映射。后面还会调用 checkVersionedPaths 函数检查是否有库的版本需要更新，如果有就覆盖原有的版本进行升级。

05 加载所有插入的库

遍历 DYLD_INSERT_LIBRARIES 环境变量，然后调用 loadInsertedDylib 加载，代码如下。

```
if  ( sEnv.DYLD_INSERT_LIBRARIES != NULL ) {
    for (const char* const* lib = sEnv.DYLD_INSERT_LIBRARIES; *lib != NULL; ++lib)
        loadInsertedDylib(*lib);
}
```

在讲解前，笔者演示一个通过环境变量 DYLD_INSERT_LIBRARIES 注入的例子。编写一个简单的动态库代码，具体如下。

```
#import <Foundation/Foundation.h>

__attribute__((constructor)) static void entry(){
    NSLog(@"InsertDylib Loaded!!!\n");
}
```

编译命令如下。命令的参数暂时不用理解，在 6.5 节会详细分析。

```
xcrun --sdk iphoneos clang -dynamiclib -arch arm64 -framework Foundation InsertDylib.mm -o
InsertDylib.dylib -compatibility_version 1 -current_version 1 -install_name
@executable_path/InsertDylib.dylib
```

这里注入的是非越狱的机器，所以还需要签名，具体如下。

```
codesign -fs "iPhone Developer: peiqing liu (FUL3EHEPG5)" ./InsertDylib.dylib
```

将生成的 InsertDylib.dylib 文件作为资源复制到应用包里面，如图 6-2 所示。在 Edit Scheme 环境变量中设置 DYLD_INSERT_LIBRARIES，值为 @executable_path/InsertDylib.dylib。运行应用，看到如下日志，说明注入成功（删除环境变量之后就不会出现了）。

```
2017-09-01 10:43:18.052 AppStart[1182:257437] InsertDylib Loaded!!!
```

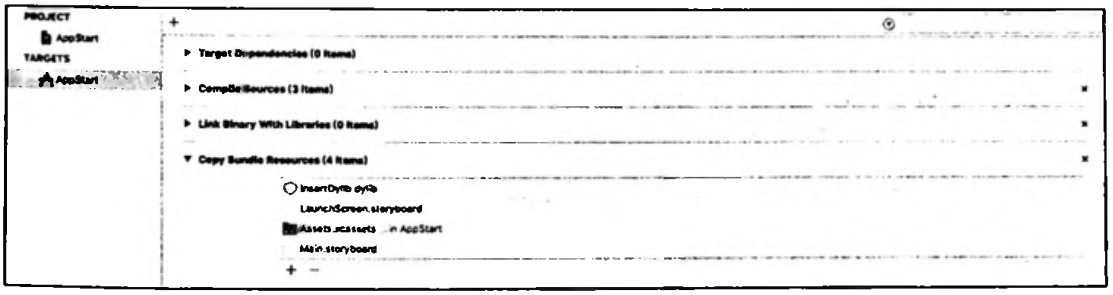


图 6-2　将 InsertDylib.dylib 复制到资源里面

下面继续来看 loadInsertedDylib，这个函数需要调用 load 方法来加载。load 方法第 1 次查看所有可能的路径，看看库是否已经加载。如果没有加载，第 2 次再调用 open 去加载。查找的过程是：通过 loadPhase0 ~ loadPhase2，分别从 DYLD_ROOT_PATH、LD_LIBRARY_PATH、DYLD_FRAMEWORK_PATH、文件自身路径、DYLD_FALLBACK_LIBRARY_PATH 查找，获取文件的描述符，通过 loadPhase6 加载，loadPhase6 通过对文件的验证，然后调用 ImageLoaderMachO::instantiateFromFile 从文件加载，并调用 addImage 加载到 sAllImages。instantiateFromFile 会先检查代码签名，如果签名无效，则直接抛出异常。

接下来，需要验证文件是否加密。如果已经加密，会调用 mremap_encrypted 让内核解密。在 loadPhase3 函数中会将 @xxxx 之类的前缀展开，这部分会在 6.5 节详细讲解。

06　链接主程序

调用 link 链接主程序。内核调用的是 `ImageLoader::link` 函数，代码如下。

```
void ImageLoader::link(const LinkContext& context, bool forceLazysBound, bool preflightOnly,
bool neverUnload, const RPathChain& loaderRPaths, const char* imagePath)
{
    //dyld::log("ImageLoader::link(%s) refCount=%d, neverUnload=%d\n", imagePath,
fDlopenReferenceCount, fNeverUnload);

    // clear error strings
    (*context.setErrorStrings)(0, NULL, NULL, NULL);

    uint64_t t0 = mach_absolute_time();
    //递归加载动态库
    this->recursiveLoadLibraries(context, preflightOnly, loaderRPaths, imagePath);
    context.notifyBatch(dyld_image_state_dependents_mapped, preflightOnly);

    // we only do the loading step for preflights
    if ( preflightOnly )
        return;

    uint64_t t1 = mach_absolute_time();
    context.clearAllDepths();
    //对 image 进行排序
    this->recursiveUpdateDepth(context.imageCount());

    uint64_t t2 = mach_absolute_time();
    //递归 rebase
    this->recursiveRebase(context);
    context.notifyBatch(dyld_image_state_rebased, false);

    uint64_t t3 = mach_absolute_time();
    //递归绑定符号表
    this->recursiveBind(context, forceLazysBound, neverUnload);

    uint64_t t4 = mach_absolute_time();
    if ( !context.linkingMainExecutable )
        this->weakBind(context);
    uint64_t t5 = mach_absolute_time();

    context.notifyBatch(dyld_image_state_bound, false);
    uint64_t t6 = mach_absolute_time();

    std::vector<DOFInfo> dofs;

    //注册 DOF 节区
    this->recursiveGetDOFSections(context, dofs);
```

```
    context.registerDOFs(dofs);
    uint64_t t7 = mach_absolute_time();

    // interpose any dynamically loaded images
    if ( !context.linkingMainExecutable && (fgInterposingTuples.size() != 0) ) {
        this->recursiveApplyInterposing(context);
    }

    // clear error strings
    (*context.setErrorStrings)(0, NULL, NULL, NULL);

    fgTotalLoadLibrariesTime += t1 - t0;
    fgTotalRebaseTime += t3 - t2;
    fgTotalBindTime += t4 - t3;
    fgTotalWeakBindTime += t5 - t4;
    fgTotalDOF += t7 - t6;

    // done with initial dylib loads
    fgNextPIEDylibAddress = 0;
}
```

调用 recursiveLoadLibraries 递归加载动态库，后者调用了 doGetDependentLibraries，从 LC_LOAD_DYLIB、LC_LOAD_WEAK_DYLIB、LC_REEXPORT_DYLIB 和 LC_LOAD_UPWARD_DYLIB 中获取所有依赖的库。接着，调用 context.loadLibrary 加载。从前面设置的上下文中可以看到，其实调用的是 libraryLocator，而且是通过 load 加载的。加载动态库后，对依赖的库进行排序，被依赖的排在前面，然后调用 recursiveRebase 进行 rebase 操作，内部调用的是 doRebase，实际上是调用 ImageLoaderMachO 的 rebase 对从 fDyldInfo->rebase_off 开始的 rebase_size 进行 rebase 操作。设置 DYLD_PRINT_REBASINGS 后可以看到日志输出，因为模块被加载的内存基地址不同，所以需要 rebase。相关代码如下。

```
dyld: rebase: libdispatch.dylib:*0x1001B4000 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4008 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4010 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4018 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4020 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4028 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4030 += 0x100180000
dyld: rebase: libdispatch.dylib:*0x1001B4038 += 0x100180000

(lldb) im li -o -f libdispatch.dylib
[  0] 0x0000000100180000
```

```
/usr/lib/system/introspection/libdispatch.dylib(0x0000000100180000)
```

recursiveBind 递归绑定符号表，在正常情况下只会绑定非懒加载符号，除非满足以下条件。

- 设置 DYLD_BIND_AT_LAUNCH，懒加载符号会立即绑定。
- 使用某些 API，例如 RTLD_NOW，会导致懒加载符号立即绑定。

eachBind 通过从 fDyldInfo->bind_off 开始的 bind_size 获取位置和符号名，实际调用 bindAt 进行绑定。通过 resolve 获取符号地址 bindLocation 来更新绑定，设置 DYLD_PRINT_BINDINGS 来打印绑定信息，代码如下。

```
dyld: bind: libsystem_m.dylib:0x1A1F1C000 = libdyld.dylib:dyld_stub_binder, *0x1A1F1C000
= 0x180EF5F64
dyld: forced lazy bind: libsystem_m.dylib:0x1A1F1C010 = libcompiler_rt.dylib:___muldc3,
*0x1A1F1C010 = 0x180E5A4F4
dyld: forced lazy bind: libsystem_m.dylib:0x1A1F1C018 = libcompiler_rt.dylib:___mulsc3,
*0x1A1F1C018 = 0x180E5A6A0
dyld: bind: libsystem_blocks.dylib:0x1A1F0E000 = libdyld.dylib:dyld_stub_binder,
*0x1A1F0E000 = 0x180EF5F64
dyld: forced lazy bind: libsystem_blocks.dylib:0x1A1F0E010 =
libsystem_c.dylib:__os_assert_log, *0x1A1F0E010 = 0x180F41288
dyld: forced lazy bind: libsystem_blocks.dylib:0x1A1F0E018 =
libsystem_c.dylib:__os_assumes_log, *0x1A1F0E018 = 0x180F40FC4
```

07 链接所有插入的库，执行符号替换

对 sAllImages（除了第一项主程序）中的库调用 link 进行链接，然后调用 register Interposing 注册符号替换，该函数读取 __DATA,__interpose 的函数和替换的函数，并将读取的信息保存到 fgInterposingTuples 中。接下来调用 applyInterposing，实际上是通过 doInterpose 函数对 S_NON_LAZY_SYMBOL_POINTERS 和 S_LAZY_SYMBOL_POINTERS 进行查找，然后根据前面 fgInterposingTuples 保存的 tuple 进行替换的。设置 DYLD_PRINT_INTERPOSING，可以打印替换的日志。

根据这里的操作可知，在 Mach-O 文件向 __DATA,__interpose 中写要替换的函数和自定义的函数时，就能对懒加载和非懒加载表中的符号进行替换。举一个简单的例子，先将上面的 InsertDylib.mm 文件的内容修改如下。

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
```

```
#include <malloc/malloc.h>
#import <Foundation/Foundation.h>

//标准的 interpose 数据结构
typedef struct interpose_s{
    void *new_func;
    void *orig_func;
}interpose_t;

void *my_malloc(int size);
void my_free(void *);

static const interpose_t interposing_functions[] \
__attribute((used, section("__DATA,""__interpose"" "))) = {{(void *)my_free,(void
*)free},{(void *)my_malloc,(void *)malloc}};

void *my_malloc (int size){
    void *returned = malloc(size);
    //printf 中会调用 malloc(), 产生无限递归调用
    malloc_printf("malloc hooked!!! is my malloc %p %d\n",returned,size);
    return (returned);
}

void my_free(void *freed){
    malloc_printf("free hooked!!! is my free  %p\n",freed);
    free(freed);
}

__attribute__((constructor)) static void entry(){
    NSLog(@"InsertDylib Loaded!!!\n");
}
```

然后，编译、签名、替换原来生成的 InsertDylib.dylib，设置环境变量 DYLD_PRINT_INTERPOSING 运行 App，日志如下即为 hook 生效。

```
dyld: interposing 2 tuples onto image: /usr/lib/system/libsystem_kernel.dylib
dyld: interposing 2 tuples onto image: /usr/lib/system/libsystem_m.dylib
dyld: interposing 2 tuples onto image: /usr/lib/system/libsystem_blocks.dylib
dyld interposing: replace 0x18103FA1C with 0x10008BE3C
dyld interposing: replace 0x18104054C with 0x10008BE78
dyld: interposing 2 tuples onto image: /usr/lib/libc++abi.dylib
dyld interposing: replace 0x18103FA1C with 0x10008BE3C
dyld interposing: replace 0x18104054C with 0x10008BE78
dyld: interposing 2 tuples onto image: /usr/lib/libc++.1.dylib
```

```
dyld interposing: replace 0x18103FA1C with 0x10008BE3C
dyld interposing: replace 0x18103FA1C with 0x10008BE3C
dyld interposing: replace 0x18104054C with 0x10008BE78
dyld: interposing 2 tuples onto image: /usr/lib/libobjc.A.dylib
dyld interposing: replace 0x18103FA1C with 0x10008BE3C
dyld interposing: replace 0x18104054C with 0x10008BE78
......
AppStart(1293,0x16dff3000) malloc: malloc hooked!!! is my malloc 0x100309f90 304
AppStart(1293,0x16dff3000) malloc: malloc hooked!!! is my malloc 0x100862000 8504
AppStart(1293,0x16dff3000) malloc: malloc hooked!!! is my malloc 0x100308f20 304
AppStart(1293,0x16dff3000) malloc: free hooked!!! is my free  0x100309f90
AppStart(1293,0x16dff3000) malloc: free hooked!!! is my free  0x10030df10
AppStart(1293,0x16dff3000) malloc: malloc hooked!!! is my malloc 0x174048670 40
AppStart(1293,0x16dff3000) malloc: malloc hooked!!! is my malloc 0x1740487f0 40
......
```

08 执行初始化方法

initializeMainExecutable 执行初始化方法，+load 和 constructor 方法就是在这里执行的。该函数依次调用了 runInitializers、processInitializers、recursiveInitialization。从此处代码中可以看到，先调用了插入动态库的初始化方法，后调用了主 App 的初始化方法。在 ImageLoader::recursive Initialization 里面调用了如下内容。

```
context.notifySingle(dyld_image_state_dependents_initialized, this, &timingInfo);
```

notifySingle 函数里面有这么一段。

```
if ( (state == dyld_image_state_dependents_initialized) && (sNotifyObjCInit != NULL) &&
image->notifyObjC() ) {
    uint64_t t0 = mach_absolute_time();
    (*sNotifyObjCInit)(image->getRealPath(), image->machHeader());
    uint64_t t1 = mach_absolute_time();
    uint64_t t2 = mach_absolute_time();
    uint64_t timeInObjC = t1-t0;
    uint64_t emptyTime = (t2-t1)*100;
    if ( (timeInObjC > emptyTime) && (timingInfo != NULL) ) {
        timingInfo->addTime(image->getShortName(), timeInObjC);
    }
}
```

此处调用了 sNotifyObjCInit，发现 sNotifyObjCInit 是在下面的位置赋值的。继续寻找，可以找到调用该函数的位置，代码如下。

```
void registerObjCNotifiers(_dyld_objc_notify_mapped mapped, _dyld_objc_notify_init init,
_dyld_objc_notify_unmapped unmapped)
{
    // record functions to call
    sNotifyObjCMapped = mapped;
    sNotifyObjCInit       = init;
    sNotifyObjCUnmapped = unmapped;

    // call 'mapped' function with all images mapped so far
    try {
        notifyBatchPartial(dyld_image_state_bound, true, NULL, false, true);
    }
    catch (const char* msg) {
        // ignore request to abort during registration
    }
}

void _dyld_objc_notify_register(_dyld_objc_notify_mapped    mapped,
                                _dyld_objc_notify_init      init,
                                _dyld_objc_notify_unmapped  unmapped)
{
    dyld::registerObjCNotifiers(mapped, init, unmapped);
}

//
// Note: only for use by objc runtime
// Register handlers to be called when objc images are mapped, unmapped, and initialized.
// Dyld will call back the "mapped" function with an array of images that contain an
objc-image-info section.
// Those images that are dylibs will have the ref-counts automatically bumped, so objc will
no longer need to
// call dlopen() on them to keep them from being unloaded.  During the call to
_dyld_objc_notify_register(),
// dyld will call the "mapped" function with already loaded objc images.  During any later
dlopen() call,
// dyld will also call the "mapped" function.  Dyld will call the "init" function when dyld
would be called
// initializers in that image.  This is when objc calls any +load methods in that image.
//
void _dyld_objc_notify_register(_dyld_objc_notify_mapped    mapped,
                                _dyld_objc_notify_init      init,
                                _dyld_objc_notify_unmapped  unmapped);
```

从函数的定义来看，该接口是供 objc runtime 调用的。访问 https://opensource.apple.com/

tarballs/objc4/，下载 objc 的代码，搜索发现在如下位置调用了该函数。

```
/***********************************************************************
* _objc_init
* Bootstrap initialization. Registers our image notifier with dyld.
* Called by libSystem BEFORE library initialization time
**********************************************************************/

void _objc_init(void)
{
    static bool initialized = false;
    if (initialized) return;
    initialized = true;

    // fixme defer initialization until an objc-using image is found?
    environ_init();
    tls_init();
    static_init();
    lock_init();
    exception_init();

    _dyld_objc_notify_register(&map_images, load_images, unmap_image);
}
```

此处注册的 init 函数就是 load_images，所以上面的 sNotifyObjCInit 调用的就是 objc 中的 load_images，而后者会调用所有的 +load 方法。这一点通过在 +load 上下一个断点也能看出来，如图 6-3 所示。

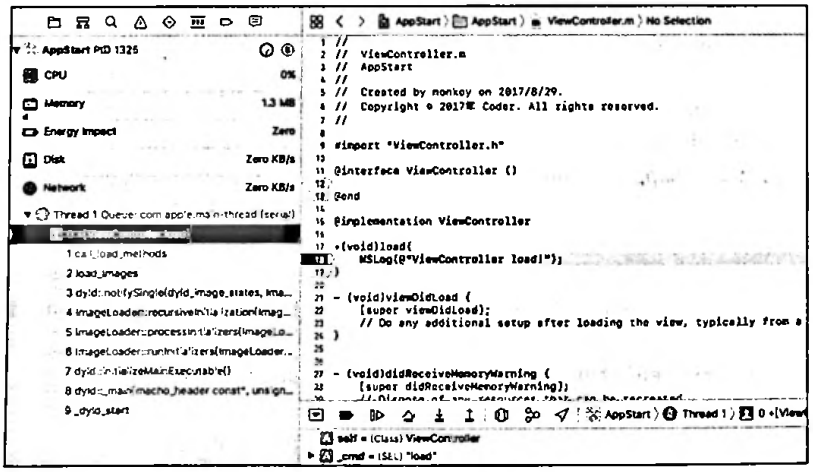


图 6-3　查看+load 函数的调用栈

调用 context.notifySingle 之后，会调用 ImageLoaderMachO::doInitialization。doImageInit 调用 LC_ROUTINES_COMMAND 的函数，设置 DYLD_PRINT_INITIALIZERS，即可打印日志。doMod InitFunctions 调用 __mod_init_funcs section 的函数，也就是 constructor 方法。使用 Xcode 查看堆栈，如图 6-4 所示。

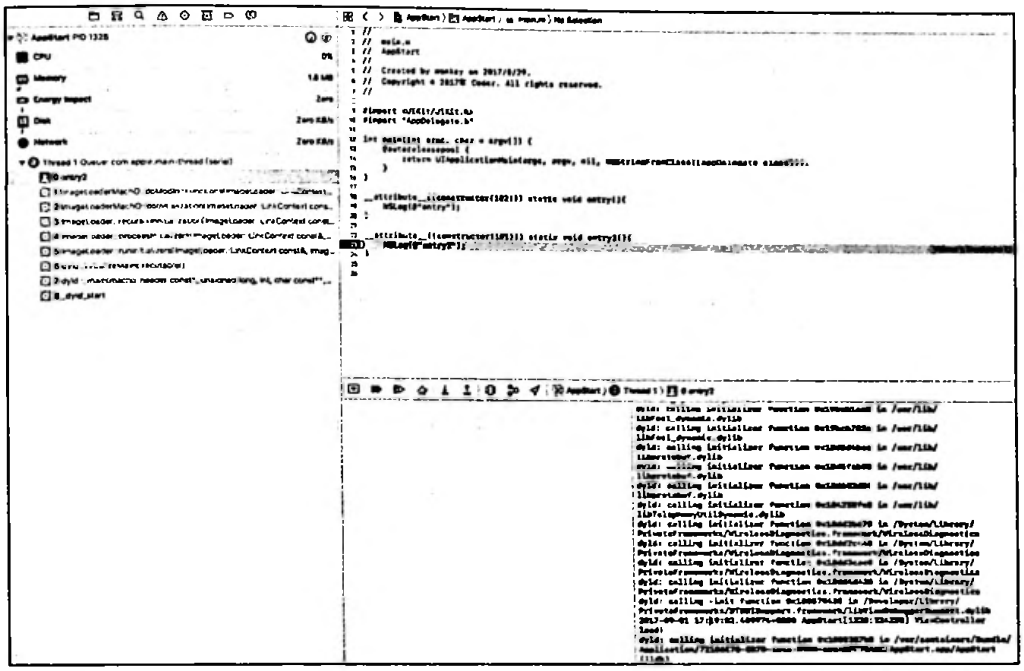


图 6-4　查看+load 函数的调用栈

在 constructor 后面可以指定 100（1~100 为系统保留）以后的数字，数字越小，优先级越高。最后，设置 DYLD_PRINT_STATISTICS 和 DYLD_PRINT_STATISTICS_DETAILS，可以打印各阶段的时间消耗。相关代码如下。

```
Total pre-main time:  60.54 milliseconds (100.0%)
         dylib loading time:  31.87 milliseconds (52.6%)
        rebase/binding time:   1.17 milliseconds (1.9%)
           ObjC setup time:   4.57 milliseconds (7.5%)
          initializer time:  22.81 milliseconds (37.6%)
          slowest intializers :
            libSystem.B.dylib :   6.32 milliseconds (10.4%)
    libBacktraceRecording.dylib :   9.52 milliseconds (15.7%)
                      AppStart :   6.48 milliseconds (10.7%)
```

```
total time: 702.75 milliseconds (100.0%)
total images loaded:  150 (146 from dyld shared cache)
total segments mapped: 11, into 143 pages with 36 pages pre-fetched
total images loading time: 465.49 milliseconds (66.2%)
total load time in ObjC:   4.57 milliseconds (0.6%)
total debugger pause time: 433.62 milliseconds (61.7%)
total dtrace DOF registration time:   0.09 milliseconds (0.0%)
total rebase fixups:  6,682
total rebase fixups time:   0.64 milliseconds (0.0%)
total binding fixups: 184,123
total binding fixups time: 209.11 milliseconds (29.7%)
total weak binding fixups time:   0.01 milliseconds (0.0%)
total redo shared cached bindings time: 208.59 milliseconds (29.6%)
total bindings lazily fixed up: 0 of 0
total time in initializers and ObjC +load:  22.81 milliseconds (3.2%)
                     libSystem.B.dylib :   6.32 milliseconds (0.8%)
            libBacktraceRecording.dylib :   9.52 milliseconds (1.3%)
                       CoreFoundation :   0.84 milliseconds (0.1%)
                  libFosl_dynamic.dylib :   1.09 milliseconds (0.1%)
                                AppStart :   6.48 milliseconds (0.9%)
total symbol trie searches:    472794
total symbol table binary searches:    0
total images defining weak symbols:  16
total images using weak symbols:  42
```

09 寻找主程序入口

调用 getThreadPC，从 Load Command 读取 LC_MAIN 入口。如果没有 LC_MAIN 入口，就读取 LC_UNIXTHREAD，然后跳到入口处执行，这样就来到了我们熟悉的 main 函数处。

6.2 Mach-O 文件格式

在学习 dyld 如何加载一个文件时，你可能也看到了，这个文件是一个 Mach-O 类型的文件。就像 PE 是 Windows 的可执行文件类型、ELF 是 Linux 的可执行文件类型一样，苹果也有自己的可执行文件类型，那就是 Mach-O。本节将对 Mach-O 文件的格式进行分析，从而全面了解一个 Mach-O 文件的组成和内容。

6.2.1 Mach-O 文件的基本格式

要想查看一个文件的类型，可以使用 file 命令。例如，使用 file 命令查看编译生成的应用

里面的可执行文件类型，代码如下（Demo 程序可以在随书源代码中找到）。

```
➜  MachODemo.app git:(master) X file MachODemo
MachODemo: Mach-O 64-bit executable arm64
➜  MachODemo.app git:(master) X file MachODemo
MachODemo: Mach-O universal binary with 2 architectures: [arm_v7: Mach-O executable arm_v7]
[arm64: Mach-O 64-bit executable arm64]
MachODemo (for architecture armv7): Mach-O executable arm_v7
MachODemo (for architecture arm64): Mach-O 64-bit executable arm64
```

一个可执行文件可能和上面的一样，只有一个架构，也可能和下面的一样，有多个架构。通常把含有多个架构的文件称为 FAT 文件。为了帮助分析，我们可以使用 MachOView 查看一个 Mach-O 文件的结构（GitHub 地址为 https://github.com/gdbinit/MachOView）。使用 MachOView 查看单个架构的可执行文件的内容，如图 6-5 所示。

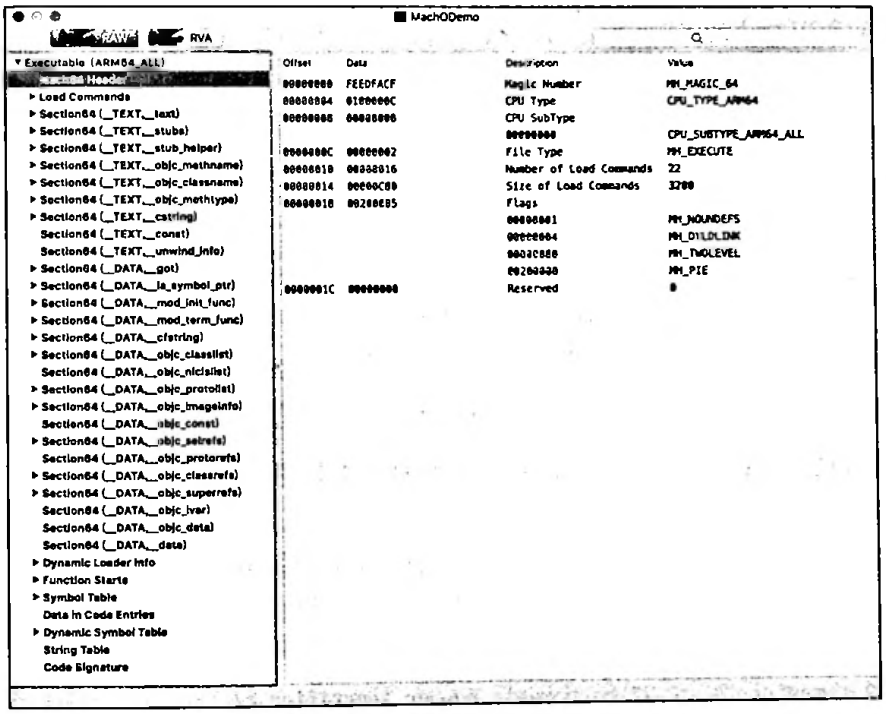


图 6-5　使用 MachOView 查看可执行文件

从图 6-5 中我们可以了解 Mach-O 文件的结构，包括 Mach-O 头部、Load Command、Section、Other Data，如图 6-6 所示。

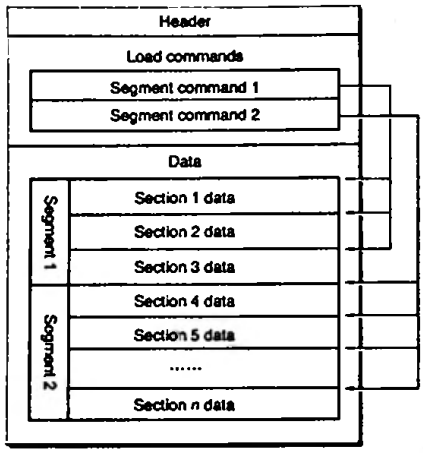


图 6-6 Mach-O 文件的基本结构

6.2.2 Mach-O 头部

首先，使用如下命令查看一个 Mach-O 文件的头部。

```
➜ MachODemo.app git:(master) X otool -h MachODemo
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedface      12          9  0x00           2    22       2648 0x00200085
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
 0xfeedfacf 16777228          0  0x00           2    22       3200 0x00200085
```

该部分的结构可以从苹果开源的代码中找到（https://opensource.apple.com/source/xnu/xnu-792/EXTERNAL_HEADERS/mach-o/loader.h.auto.html），具体如下。

```
/*
 * The mach header appears at the very beginning of the object file; it
 * is the same for both 32-bit and 64-bit architectures.
 */
struct mach_header {
    uint32_t magic;        /* mach magic number identifier */
    cpu_type_t    cputype; /* cpu specifier */
    cpu_subtype_t cpusubtype;   /* machine specifier */
    uint32_t filetype;     /* type of file */
    uint32_t ncmds;        /* number of load commands */
    uint32_t sizeofcmds;   /* the size of all the load commands */
    uint32_t flags;        /* flags */
```

```
};

/*
 * The 64-bit mach header appears at the very beginning of object files for
 * 64-bit architectures.
 */
struct mach_header_64 {
    uint32_t magic;        /* mach magic number identifier */
    cpu_type_t    cputype; /* cpu specifier */
    cpu_subtype_t cpusubtype;  /* machine specifier */
    uint32_t filetype;     /* type of file */
    uint32_t ncmds;        /* number of load commands */
    uint32_t sizeofcmds;   /* the size of all the load commands */
    uint32_t flags;        /* flags */
    uint32_t reserved;     /* reserved */
};
```

从上面的代码中可以看出，它由以下部分组成。

- magic：Mach-O 文件的魔数。FAT 为 0xcafebabe，ARMv7 为 0xfeedface，ARM64 为 0xfeedfacf（Mac 是小端模式）。
- cputype、cpusubtype：CPU 架构和子版本。
- filetype：文件类型。常见有的 MH_OBJECT（目标文件）、MH_EXECUTABLE（可执行二进制文件）、MH_DYLIB（动态库）。
- ncmds：加载命令的数量。
- sizeofcmds：所有加载命令的大小。
- flags：dyld 加载需要的一些标记。其中，MH_PIE 表示启用地址空间布局随机化。
- reserved：64 位的保留字段。

使用 v 参数查看对应的解释，具体如下。

```
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
   MH_MAGIC     ARM         V7  0x00     EXECUTE    22       2648   NOUNDEFS DYLDLINK
TWOLEVEL PIE
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
MH_MAGIC_64   ARM64         ALL  0x00     EXECUTE    22       3200   NOUNDEFS DYLDLINK
TWOLEVEL PIE
```

6.2.3 Load Command

Load Command 告诉操作系统应当如何加载文件中的数据，对系统内核加载器和动态链接器起指导作用。使用 MachOView 查看 Load Command，如图 6-7 所示（以下分析的都是 64 位的结构信息）。

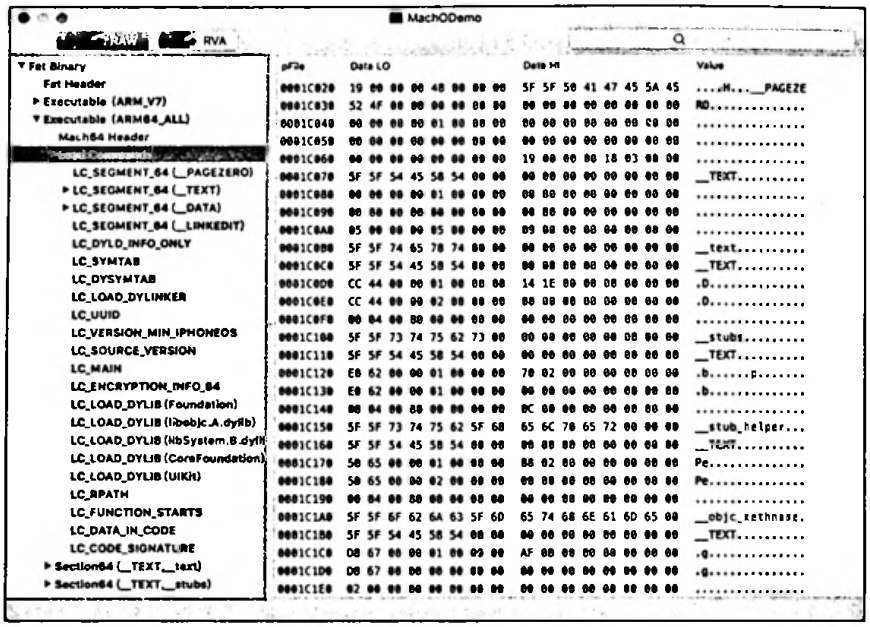


图 6-7 使用 MachOView 查看 Load Command

可以看到，Load Command 包含以下部分。

- LC_SEGMENT_64：定义一个段，加载后被映射到内存中，包括里面的节。
- LC_DYLD_INFO_ONLY：记录了有关链接的重要信息，包括在 __LINKEDIT 中动态链接相关信息的具体偏移和大小。ONLY 表示这个加载指令是程序运行所必需的，如果旧的链接器无法识别它，程序就会出错。
- LC_SYMTAB：为文件定义符号表和字符串表，在链接文件时被链接器使用，同时也用于调试器映射符号到源文件。符号表定义的本地符号仅用于调试，而已定义和未定义的 external 符号被链接器使用。
- LC_DYSYMTAB：将符号表中给出符号的额外符号信息提供给动态链接器。
- LC_LOAD_DYLINKER：默认的加载器路径。

- LC_UUID：用于标识 Mach-O 文件的 ID，也用于崩溃堆栈和符号文件的对应解析。
- LC_VERSION_MIN_IPHONEOS：系统要求的最低版本。
- LC_SOURCE_VERSION：构建二进制文件的源代码版本号。
- LC_MAIN：程序的入口。dyld 获取该地址，然后跳转到该处执行。
- LC_ENCRYPTION_INFO_64：文件是否加密的标志，加密内容的偏移和大小。
- LC_LOAD_DYLIB：依赖的动态库，包括动态库名称、当前版本号、兼容版本号。可以使用“otool -L xxx”命令查看。
- LC_RPATH：Runpath Search Paths，@rpath 搜索的路径。
- LC_FUNCTION_STARTS：函数起始地址表，使调试器和其他程序能很容易地看到一个地址是否在函数内。
- LC_DATA_IN_CODE：定义在代码段内的非指令的表。
- LC_CODE_SIGNATURE：代码签名信息。

其中，LC_SEGMENT_64 定义了一个 64 位的段，当文件加载后映射到地址空间（包括段里面节的定义）。64 位段的定义如下。

```
/*
 * The 64-bit segment load command indicates that a part of this file is to be
 * mapped into a 64-bit task's address space.  If the 64-bit segment has
 * sections then section_64 structures directly follow the 64-bit segment
 * command and their size is reflected in cmdsize.
 */
struct segment_command_64 {     /* for 64-bit architectures */
    uint32_t cmd;               /* LC_SEGMENT_64 */
    uint32_t cmdsize;           /* includes sizeof section_64 structs */
    char     segname[16];       /* segment name */
    uint64_t vmaddr;            /* memory address of this segment */
    uint64_t vmsize;            /* memory size of this segment */
    uint64_t fileoff;           /* file offset of this segment */
    uint64_t filesize;          /* amount to map from the file */
    vm_prot_t    maxprot;       /* maximum VM protection */
    vm_prot_t    initprot;      /* initial VM protection */
    uint32_t nsects;            /* number of sections in segment */
    uint32_t flags;             /* flags */
};
```

- cmd：Load Command 类型。
- cmdsize：Load Command 结构的大小。

- segname：段的名字。
- vmaddr：映射到虚拟地址的偏移。
- vmsize：映射到虚拟地址的大小。
- fileoff：对应于当前架构文件的偏移（注意：是当前架构文件，不是整个 FAT 文件）。
- filesize：文件的大小。
- maxprot：段页面的最高内存保护。
- initprot：初始内存保护。
- nsects：包含的节的个数。
- flags：段页面标志。

系统将 fileoff 偏移处 filesize 大小的内容加载到虚拟内存的 vmaddr 处，大小为 vmsize，段页面的权限由 initprot 进行初始化。它的权限可以动态改变，但是不能超过 maxprot 的值，例如 __TEXT 初始化和最大权限都是可读/可执行/不可写。

上面的文件中包括以下 4 种段。

- __PAGEZERO：空指针陷阱段，映射到虚拟内存空间的第 1 页，用于捕捉对 NULL 指针的引用。
- __TEXT：代码段/只读数据段。
- __DATA：读取和写入数据的段。
- __LINKEDIT：动态链接器需要使用的信息，包括重定位信息、绑定信息、懒加载信息等。

段里面可以包含不同的节（Section）。节的结构定义如下。

```
struct section_64 {         /* for 64-bit architectures */
    char     sectname[16]; /* name of this section */
    char     segname[16];  /* segment this section goes in */
    uint64_t addr;         /* memory address of this section */
    uint64_t size;         /* size in bytes of this section */
    uint32_t offset;       /* file offset of this section */
    uint32_t align;        /* section alignment (power of 2) */
    uint32_t reloff;       /* file offset of relocation entries */
    uint32_t nreloc;       /* number of relocation entries */
    uint32_t flags;        /* flags (section type and attributes)*/
    uint32_t reserved1;    /* reserved (for offset or index) */
    uint32_t reserved2;    /* reserved (for count or sizeof) */
    uint32_t reserved3;    /* reserved */
};
```

- sectname：节的名字。
- segname：段的名字。
- addr：映射到虚拟地址的偏移。
- size：节的大小。
- offset：节在当前架构文件中的偏移。
- align：节的字节对齐大小 n，计算结果为 2^n。
- reloff：重定位入口的文件偏移。
- nreloc：重定位入口的个数。
- flags：节的类型和属性。
- reserved：保留位。

下面我们来看看 __TEXT 段和 __DATA 段下面都有哪些节。首先来看 __TEXT 段。

- __text：程序可执行的代码区域。
- __stubs：间接符号存根，跳转到懒加载指针表。
- __stub_helper：帮助解决懒加载符号加载的辅助函数。
- __objc_methname：方法名。
- __objc_classname：类名。
- __objc_methtype：方法签名。
- __cstring：只读的 C 风格字符串，包含 OC 的部分字符串和属性名。

__DATA 段下面的节如下。

- __nl_symbol_ptr：非懒加载指针表，在 dyld 加载时会立即绑定值。
- __la_symbol_ptr：懒加载指针表，第 1 次调用时才会绑定值。
- __got：非懒加载全局指针表。
- __mod_init_func：constructor 函数。
- __mod_term_func：destructor 函数。
- __cfstring：OC 字符串。
- __objc_classlist：程序中类的列表。
- __objc_nlclslist：程序中自己实现了+load 方法的类。
- __objc_protolist：协议的列表。
- __objc_classrefs：被引用的类列表。

当然，这里并没有介绍全部的节。对没有介绍的节，读者可以将文件拖到 IDA 里面，查看 IDA 的分析结果。

6.2.4 虚拟地址和文件偏移

前面在段和节的定义中都提到了虚拟地址的偏移和文件地址的偏移。这两个偏移分别代表什么呢？下面以 __TEXT,__text 为例，使用 MachOView 查看，如图 6-8 所示。

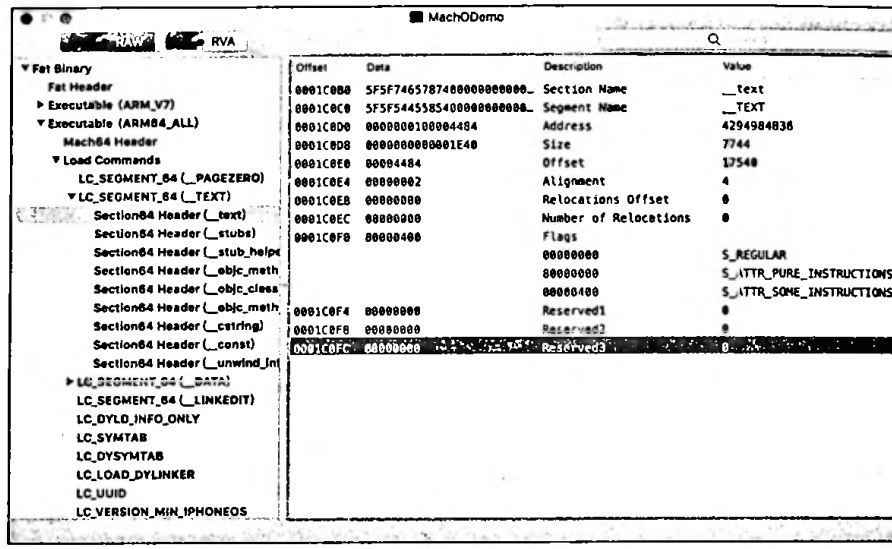


图 6-8 使用 MachOView 查看__text 节

虚拟地址是 0x0000000100004484，表示该节的内容映射到虚拟内容时相对该模块加载基地址的偏移。运行程序，获取模块加载基地址，然后加上偏移的虚拟地址，代码如下。

```
(lldb) im li -o -f MachODemo
[  0] 0x00000000000e8000
/Users/monkey/Documents/iosreversebook/sourcecode/chapter-6/MachODemo/Build/Products/De
bug-iphoneos/MachODemo.app/MachODemo
(lldb) p/x 0x00000000000e8000 + 0x0000000100004484
(long) $0 = 0x00000001000ec484
```

如果没有计算错的话，可以对 0x00000001000ec484 这个地址进行反汇编，具体如下。

```
(lldb) dis -a 0x00000001000ec484
MachODemo`-[ViewController viewDidLoad]:
    0x1000ec484 <+0>:  sub    sp, sp, #0x30             ; =0x30
    0x1000ec488 <+4>:  stp    x29, x30, [sp, #0x20]
```

```
0x1000ec48c <+8>:  add    x29, sp, #0x20        ; =0x20
0x1000ec490 <+12>: mov    x8, sp
0x1000ec494 <+16>: adrp   x9, 5
0x1000ec498 <+20>: add    x9, x9, #0xe0         ; =0xe0
0x1000ec49c <+24>: adrp   x10, 5
0x1000ec4a0 <+28>: add    x10, x10, #0x1a8      ; =0x1a8
0x1000ec4a4 <+32>: stur   x0, [x29, #-0x8]
0x1000ec4a8 <+36>: str    x1, [sp, #0x10]
0x1000ec4ac <+40>: ldur   x0, [x29, #-0x8]
0x1000ec4b0 <+44>: str    x0, [sp]
0x1000ec4b4 <+48>: ldr    x10, [x10]
0x1000ec4b8 <+52>: str    x10, [sp, #0x8]
0x1000ec4bc <+56>: ldr    x1, [x9]
0x1000ec4c0 <+60>: mov    x0, x8
0x1000ec4c4 <+64>: bl     0x1000ee468           ; symbol stub for: objc_msgSendSuper2
0x1000ec4c8 <+68>: ldp    x29, x30, [sp, #0x20]
0x1000ec4cc <+72>: add    sp, sp, #0x30         ; =0x30
0x1000ec4d0 <+76>: ret
```

然后，单击“Debug”→“Debug Workflow”→“View Memory”选项，输入“0x00000001000ec484”后按“Enter”键查看内容，如图 6-9 所示（稍后可以和文件中的内容对比一下）。

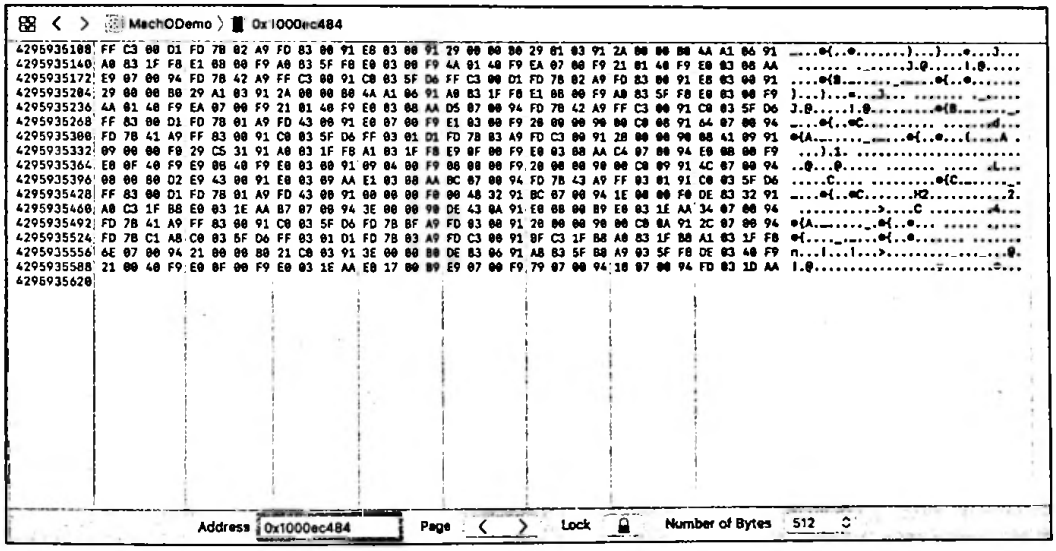


图 6-9　使用 Xcode 查看指定地址在文件中的内容

我们看到的文件偏移是 0x00004484，这是相对当前架构文件的偏移值。如果是单一架构的文件，这个偏移值就是正确的。如果是多架构的文件，首先要找到当前架构在 FAT 文件中的偏移。

例如，当前是 ARM64 架构，在 FAT Header 中看到 ARM64 架构的偏移是 0x1C000，如图 6-10 所示。

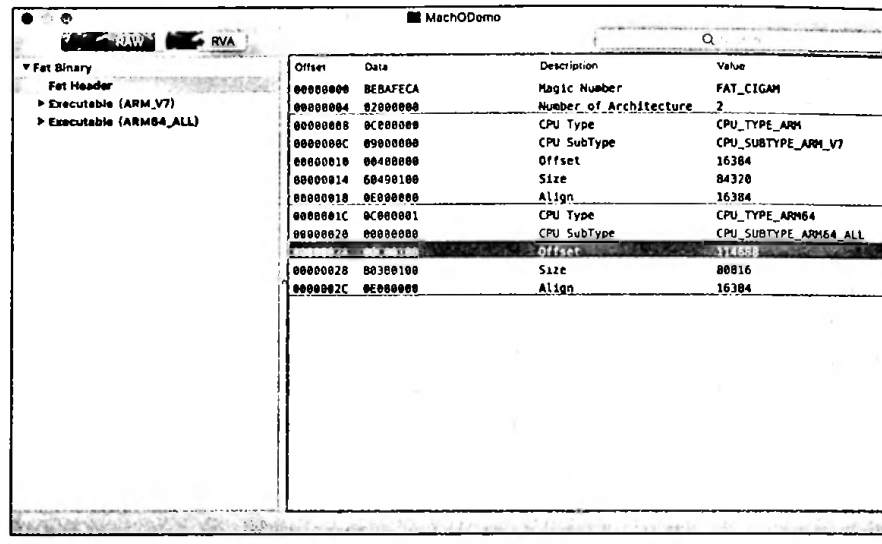


图 6-10　使用 MachOView 查看 FAT Header 的内容

加上文件偏移 0x00004484，即 0x1C000+0x00004484=0x20484。到这个文件偏移的位置查看，如图 6-11 所示，和在内存中看到的内容是一样的。

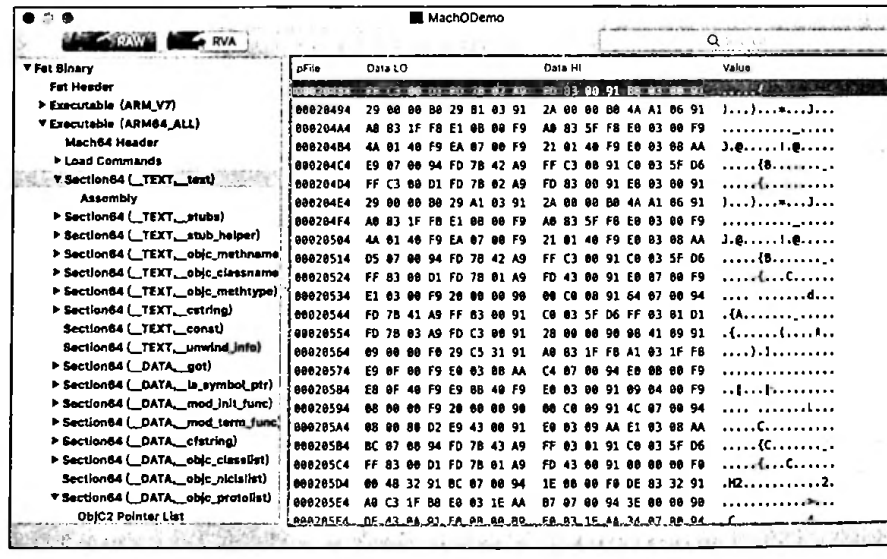


图 6-11　使用 MachOView 查看__TEXT,__text

6.2.5 懒加载和非懒加载

iOS 系统为了加快系统启动速度，将符号分成了懒加载符号和非懒加载符号。非懒加载符号在 dyld 加载时就会绑定真实的值；而懒加载符号不会，只有第 1 次去调用它时才会绑定真实的地址，在第 2 次调用时直接使用真实的地址。为了让读者有一个清晰的认识，下面笔者通过 printf 调用的例子进行讲解。在 constructor 中 2 次调用 printf，代码如下。

```
static __attribute__((constructor))
void my_constructor(){
    printf("hello one!!!\n");
    printf("hello two!!!\n");
    NSLog(@"exec my_constructor before main function");
}
```

printf 就是一个懒加载符号，所以第 1 次和第 2 次调用的过程是不一样的。在这里主要讨论第 1 次调用的过程。在第 1 个 printf 处下断点，然后运行程序，断点所在代码如下。

```
MachODemo`my_constructor:
    0x1000985c4 <+0>:  sub    sp, sp, #0x20             ; =0x20
    0x1000985c8 <+4>:  stp    x29, x30, [sp, #0x10]
    0x1000985cc <+8>:  add    x29, sp, #0x10            ; =0x10
    0x1000985d0 <+12>: adrp   x0, 3
    0x1000985d4 <+16>: add    x0, x0, #0xc92            ; =0xc92
->  0x1000985d8 <+20>: bl     0x10009a4c8               ; symbol stub for: printf
    0x1000985dc <+24>: adrp   x30, 3
    0x1000985e0 <+28>: add    x30, x30, #0xca0          ; =0xca0
    0x1000985e4 <+32>: stur   w0, [x29, #-0x4]
    0x1000985e8 <+36>: mov    x0, x30
    0x1000985ec <+40>: bl     0x10009a4c8               ; symbol stub for: printf
    0x1000985f0 <+44>: adrp   x30, 4
    0x1000985f4 <+48>: add    x30, x30, #0x290          ; =0x290
    0x1000985f8 <+52>: str    w0, [sp, #0x8]
    0x1000985fc <+56>: mov    x0, x30
    0x100098600 <+60>: bl     0x10009a2d0               ; symbol stub for: NSLog
    0x100098604 <+64>: ldp    x29, x30, [sp, #0x10]
    0x100098608 <+68>: add    sp, sp, #0x20             ; =0x20
    0x10009860c <+72>: ret
```

0x10009a4c8 转换成文件偏移，其实就是 __TEXT,__stubs 里面指向 printf 的位置。当前模块基地址为 0x00000000000d4000，转换公式如下。

```
内存地址 - 模块加载基地址 - 文件加载虚拟地址偏移 + 该架构在文件中的偏移
0x10009a4c8 - 0x0000000000094000 - 0x100000000 + 0x1C000 = 0x224C8
```

或者，使用 im li MachODemo 命令，直接得到文件加载到虚拟内存的偏移，具体如下。

```
[  0] 1C615EBF-9824-3EA9-A10F-6529D4FC5EFF 0x0000000100094000
/Users/monkey/Documents/iosreversebook/sourcecode/chapter-6/MachODemo/Build/Products/De
bug-iphoneos/MachODemo.app/MachODemo

/Users/monkey/Documents/iosreversebook/sourcecode/chapter-6/MachODemo/Build/Products/De
bug-iphoneos/MachODemo.app.dSYM/Contents/Resources/DWARF/MachODemo

0x10009a4c8 - 0x0000000100094000 + 0x1C000 = 0x224C8
```

使用 si 命令跟进，代码如下。

```
MachODemo`printf:
->  0x10009a4c8 <+0>: nop
    0x10009a4cc <+4>: ldr    x16, #0x1d0c              ; (void *)0x000000010009a75c
    0x10009a4d0 <+8>: br     x16
```

因为此处 ldr 指令的意思是取 pc + 0x1d0c 指向地址的值（在 6.3 节会详细讲解），所以有如下代码。

```
(lldb) p/x $pc + 0x1d0c
(unsigned long) $0 = 0x000000010009c1d8
(lldb) x/xg 0x000000010009c1d8
0x10009c1d8: 0x000000010009a75c
```

0x000000010009c1d8 所对应的文件偏移是 0x241D8。通过查看文件在 __la_symbol_ptr 中 __printf 指针的位置可知，上面其实是取 __la_symbol_ptr 中 __printf 指针的值。跟进查看，具体如下。

```
->  0x10009a75c: ldr    w16, 0x10009a764
    0x10009a760: b      0x10009a534
    0x10009a764: .long  0x0000042e              ; unknown opcode
    0x10009a768: ldr    w16, 0x10009a770
    0x10009a76c: b      0x10009a534
    0x10009a770: .long  0x00000486              ; unknown opcode
    0x10009a774: ldr    w16, 0x10009a77c
    0x10009a778: b      0x10009a534
```

0x10009a75c 对应于文件中 0x2275C 的位置。该处在 __stub_helper 所在的位置，该指令在 0x10009a764 处取值为 0x0000042e。这是一个偏移值，是距离 Dynamic Loader Info 中 Lazy Binding Info 起始地址的偏移。从文件中可知，起始地址是 0x283c8，加上 0x0000042e，结果为 0x287F6。查看此处，如图 6-12 所示（后面会讲解此处结构的用处）。

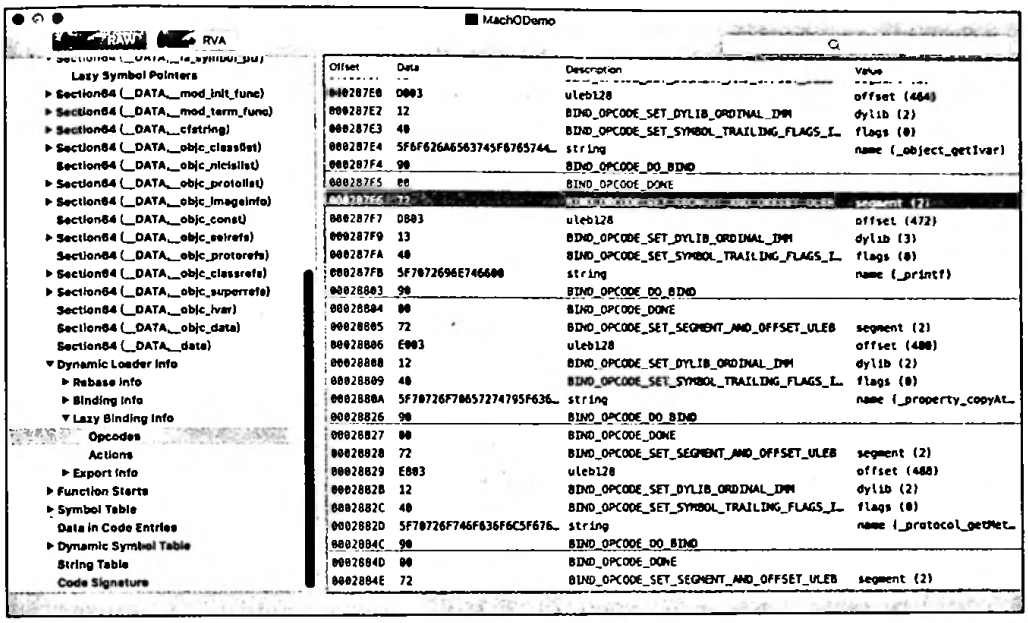


图 6-12　使用 MachOView 查看 Lazy Binding Info

跳转到 0x10009a534 处，此处对应于文件中 __stub_helper 的开头位置。可以看到，此处调用了 dyld_stub_binder 执行绑定，具体如下。

```
->  0x10009a534: adr    x17, #0x1b44              ; (void *)0x0000000120046038:
initialPoolContent + 2856
    0x10009a538: nop
    0x10009a53c: stp    x16, x17, [sp, #-0x10]!
    0x10009a540: nop
    0x10009a544: ldr    x16, #0x1b2c              ; (void *)0x0000000198aa915c:
dyld_stub_binder
    0x10009a548: br     x16
    0x10009a54c: ldr    w16, 0x10009a554
    0x10009a550: b      0x10009a534
```

跟进 dyld_stub_binder 的调用，具体如下。

```
libdyld.dylib`dyld_stub_binder:
->  0x198aa915c <+0>:   stp    x29, x30, [sp, #-0x10]!
    0x198aa9160 <+4>:   mov    x29, sp
    0x198aa9164 <+8>:   sub    sp, sp, #0xf0             ; =0xf0
    0x198aa9168 <+12>:  stp    x0, x1, [x29, #-0x10]
    0x198aa916c <+16>:  stp    x2, x3, [x29, #-0x20]
    0x198aa9170 <+20>:  stp    x4, x5, [x29, #-0x30]
    0x198aa9174 <+24>:  stp    x6, x7, [x29, #-0x40]
    0x198aa9178 <+28>:  stp    x8, x9, [x29, #-0x50]
    0x198aa917c <+32>:  stp    q0, q1, [x29, #-0x80]
    0x198aa9180 <+36>:  stp    q2, q3, [x29, #-0xa0]
    0x198aa9184 <+40>:  stp    q4, q5, [x29, #-0xc0]
    0x198aa9188 <+44>:  stp    q6, q7, [x29, #-0xe0]
    0x198aa918c <+48>:  ldr    x0, [x29, #0x18]
    0x198aa9190 <+52>:  ldr    x1, [x29, #0x10]
    0x198aa9194 <+56>:  bl     0x198aa9d9c               ; _dyld_fast_stub_entry(void*,
long)
    0x198aa9198 <+60>:  mov    x16, x0
    0x198aa919c <+64>:  ldp    x0, x1, [x29, #-0x10]
    0x198aa91a0 <+68>:  ldp    x2, x3, [x29, #-0x20]
    0x198aa91a4 <+72>:  ldp    x4, x5, [x29, #-0x30]
    0x198aa91a8 <+76>:  ldp    x6, x7, [x29, #-0x40]
    0x198aa91ac <+80>:  ldp    x8, x9, [x29, #-0x50]
    0x198aa91b0 <+84>:  ldp    q0, q1, [x29, #-0x80]
    0x198aa91b4 <+88>:  ldp    q2, q3, [x29, #-0xa0]
    0x198aa91b8 <+92>:  ldp    q4, q5, [x29, #-0xc0]
    0x198aa91bc <+96>:  ldp    q6, q7, [x29, #-0xe0]
    0x198aa91c0 <+100>: mov    sp, x29
    0x198aa91c4 <+104>: ldp    x29, x30, [sp], #0x10
    0x198aa91c8 <+108>: add    sp, sp, #0x10             ; =0x10
    0x198aa91cc <+112>: br     x16
```

直接查看 0x198aa9194 处调用的函数。在 0x198aa9194 处下断点。程序断下之后，x1 寄存器的值是 0x000000000000042e，这就是上面的偏移值。而此处调用的函数是 __dyld_fast_stub_entry，该函数根据偏移值来执行真正的绑定操作。再看图 6-12 中的信息：dylib(3) 表示该符号要从当前文件的第 3 个 LC_LOAD_DYLIB 里面去找，而第 3 个 LC_LOAD_DYLIB 为 libSystem.B.dylib，也就是说，在这个 dylib 里面可以找到 printf 的真实地址；segment(2) 和 offset(472) 表示将找到的真实地址写入当前架构的第 2 个 segment（不包括 __PAGEZERO）偏移为 472 的地方，查看第 2 个 segment，是 __DATA，对应偏移为 472 的地址是 0x24000+472=0x241D8，而 0x241D8 就是 __la_symbol_ptr 中 printf 符号的指针位置，也就是说，把找到的真实地址写回 __la_symbol_ptr。

第 1 次解析之后，回到第 2 次调用的地方，跟进并查看，代码如下。

```
MachODemo`my_constructor:
    0x1000985c4 <+0>:  sub    sp, sp, #0x20             ; =0x20
    0x1000985c8 <+4>:  stp    x29, x30, [sp, #0x10]
    0x1000985cc <+8>:  add    x29, sp, #0x10            ; =0x10
    0x1000985d0 <+12>: adrp   x0, 3
    0x1000985d4 <+16>: add    x0, x0, #0xc92            ; =0xc92
    0x1000985d8 <+20>: bl     0x10009a4c8               ; symbol stub for: printf
    0x1000985dc <+24>: adrp   x30, 3
    0x1000985e0 <+28>: add    x30, x30, #0xca0          ; =0xca0
    0x1000985e4 <+32>: stur   w0, [x29, #-0x4]
    0x1000985e8 <+36>: mov    x0, x30
->  0x1000985ec <+40>: bl     0x10009a4c8               ; symbol stub for: printf
    0x1000985f0 <+44>: adrp   x30, 4
    0x1000985f4 <+48>: add    x30, x30, #0x290          ; =0x290
    0x1000985f8 <+52>: str    w0, [sp, #0x8]
    0x1000985fc <+56>: mov    x0, x30
    0x100098600 <+60>: bl     0x10009a2d0               ; symbol stub for: NSLog
    0x100098604 <+64>: ldp    x29, x30, [sp, #0x10]
    0x100098608 <+68>: add    sp, sp, #0x20             ; =0x20
    0x10009860c <+72>: ret
```

尽管 si 跟进的内容还是从 __la_symbol_ptr 里面取 printf 符号的指针地址，但可以看到，这里拿到的值 0x0000000198aef410 就是真实的 printf 符号的位置，代码如下。

```
MachODemo`printf:
->  0x10009a4c8 <+0>: nop
    0x10009a4cc <+4>: ldr    x16, #0x1d0c          ; (void *)0x0000000198aef410: printf
    0x10009a4d0 <+8>: br     x16
```

跟进 br x16，这里就是真实的 printf 符号的位置，具体如下。

```
libsystem_c.dylib`printf:
->  0x198aef410 <+0>:   stp    x20, x19, [sp, #-0x20]!
    0x198aef414 <+4>:   stp    x29, x30, [sp, #0x10]
    0x198aef418 <+8>:   add    x29, sp, #0x10          ; =0x10
    0x198aef41c <+12>:  sub    sp, sp, #0x10           ; =0x10
    0x198aef420 <+16>:  mov    x19, x0
    0x198aef424 <+20>:  add    x8, x29, #0x10          ; =0x10
    0x198aef428 <+24>:  str    x8, [sp, #0x8]
    0x198aef42c <+28>:  adrp   x8, 15421
    0x198aef430 <+32>:  add    x8, x8, #0xce0          ; =0xce0
```

```
    0x198aef434 <+36>:  ldr    x20, [x8]
    0x198aef438 <+40>:  adrp   x8, 15420
    0x198aef43c <+44>:  nop
    0x198aef440 <+48>:  ldr    x0, [x8, #0x48]
    0x198aef444 <+52>:  bl     0x198b50d40              ; symbol stub for:
pthread_getspecific
    0x198aef448 <+56>:  adrp   x8, 15420
    0x198aef44c <+60>:  add    x8, x8, #0x700           ; =0x700
    0x198aef450 <+64>:  cmp    x0, #0x0                 ; =0x0
    0x198aef454 <+68>:  csel   x1, x0, x8, ne
    0x198aef458 <+72>:  ldr    x3, [sp, #0x8]
    0x198aef45c <+76>:  mov    x0, x20
    0x198aef460 <+80>:  mov    x2, x19
    0x198aef464 <+84>:  bl     0x198b51ed4              ; symbol stub for: vfprintf_l
    0x198aef468 <+88>:  sub    sp, x29, #0x10           ; =0x10
    0x198aef46c <+92>:  ldp    x29, x30, [sp, #0x10]
    0x198aef470 <+96>:  ldp    x20, x19, [sp], #0x20
    0x198aef474 <+100>: ret
```

不是说好的在 libSystem.B.dylib 里面吗？怎么变成 libsystem_c.dylib 了？可以使用如下命令查看 libSystem.B.dylib（可以从前面 decache 的内容中找到）依赖的库。

```
➜  lib otool -L libSystem.B.dylib
libSystem.B.dylib:
    ......
    /usr/lib/system/libsystem_blocks.dylib (compatibility version 1.0.0, current version
65.0.0)
    /usr/lib/system/libsystem_c.dylib (compatibility version 1.0.0, current version
1046.0.0)
    /usr/lib/system/libsystem_configuration.dylib (compatibility version 1.0.0, current
version 700.3.1)
    /usr/lib/system/libsystem_coreservices.dylib (compatibility version 1.0.0, current
version 7.0.0)
    ......
```

libSystem.B.dylib 会加载 libsystem_c.dylib，所以在 libSystem.B.dylib 里面也没错。

分析完整个流程之后，总结一下懒加载符号的绑定过程，如图 6-13 所示。6.4.3 节讲解的 fishhook 的原理，其实就是替换了懒加载和非懒加载指针表中绑定的指针。

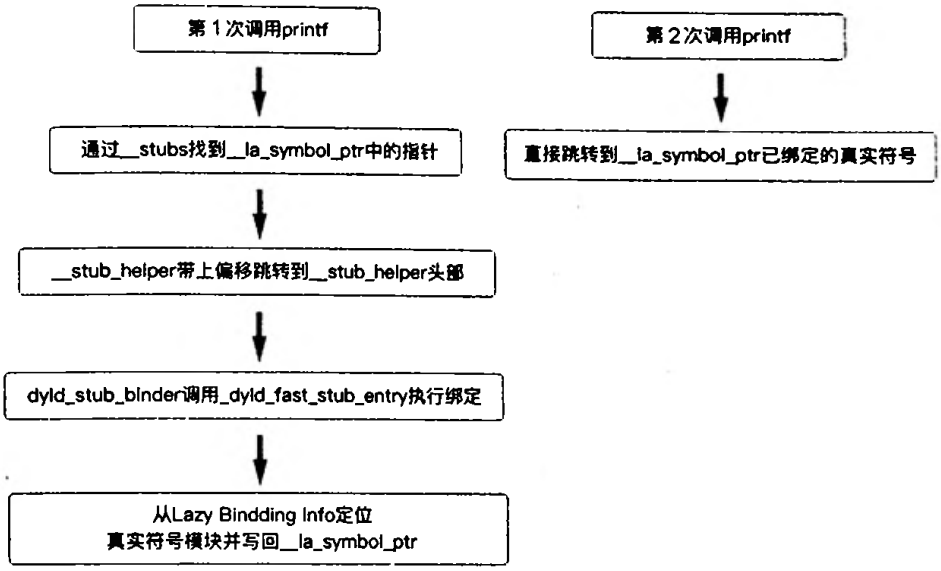


图 6-13　懒加载符号绑定过程

6.2.6　Code Signature

在 MachOView 的最后，可以看到一个 Code Signature，如图 6-14 所示。Code Signature 中记录的是 Mach-O 文件的签名信息，相关数据结构可以在内核代码头文件中找到（https://opensource.apple.com/source/xnu/xnu-3789.51.2/bsd/sys/codesign.h.auto.html）。由于 MachOView 本身并没有将这部分数据解析出来，下面笔者以该格式为例进一步分析这部分保存的数据。

整个代码签名的头部是一个 CS_SuperBlob 结构体，具体如下。

```
typedef struct __SC_SuperBlob {
    uint32_t magic;                 /* magic number */
    uint32_t length;                /* total length of SuperBlob */
    uint32_t count;                 /* number of index entries following */
    CS_BlobIndex index[];           /* (count) entries */
    /* followed by Blobs in no particular order as indicated by offsets in index */
} CS_SuperBlob;
```

在头文件中可以找到魔数的可选值，具体如下。

```
/*
 * Magic numbers used by Code Signing
 */
enum {
```

```
    CSMAGIC_REQUIREMENT = 0xfade0c00,    /* single Requirement blob */
    CSMAGIC_REQUIREMENTS = 0xfade0c01,   /* Requirements vector (internal requirements) */
    CSMAGIC_CODEDIRECTORY = 0xfade0c02,      /* CodeDirectory blob */
    CSMAGIC_EMBEDDED_SIGNATURE = 0xfade0cc0, /* embedded form of signature data */
    CSMAGIC_EMBEDDED_SIGNATURE_OLD = 0xfade0b02, /* XXX */
    CSMAGIC_EMBEDDED_ENTITLEMENTS = 0xfade7171,  /* embedded entitlements */
    CSMAGIC_DETACHED_SIGNATURE = 0xfade0cc1,     /* multi-arch collection of embedded
signatures */
    CSMAGIC_BLOBWRAPPER = 0xfade0b01,        /* CMS Signature, among other things */
    ......
}
```

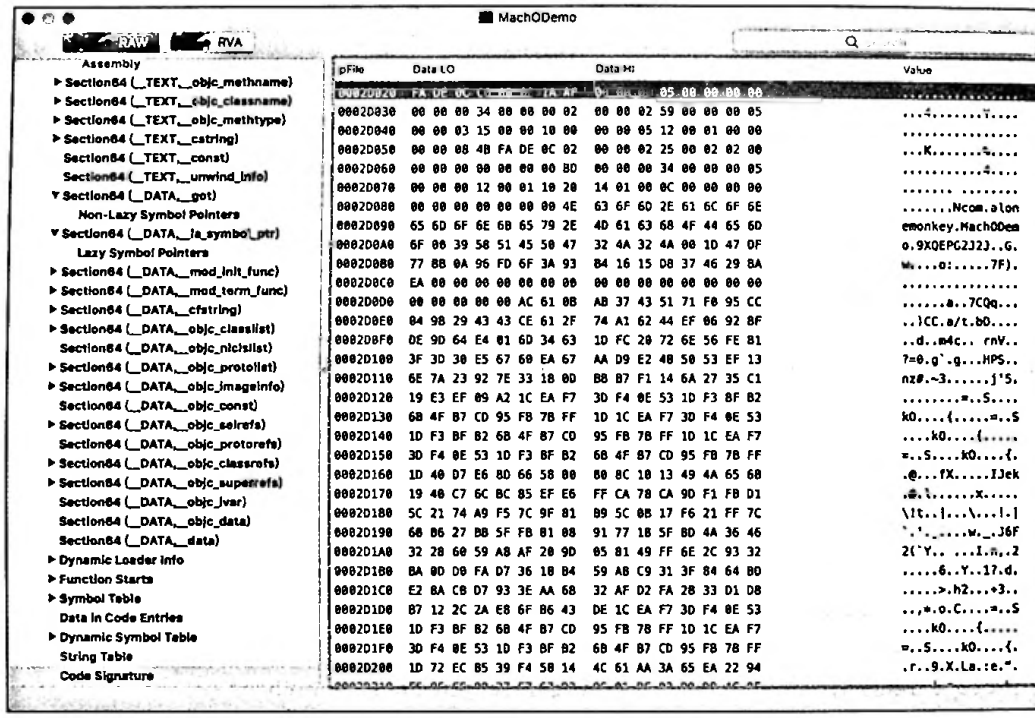


图 6-14　用 MachOView 查看 Code Signature

如图 6-13 所示，当前对应的是 CSMAGIC_EMBEDDED_SIGNATURE，表示嵌入的代码签名数据，length 表示整个 SuperBlob 的长度为 0x1AAF，count 表示接下来有多少个子条目，每个子条目的结构是 CS_BlobIndex。相关代码如下。

```
typedef struct __BlobIndex {
    uint32_t type;                  /* type of entry */
    uint32_t offset;                /* offset of entry */
```

```
} CS_BlobIndex;
```

type 表示 CS_BlobIndex 结构的类型，可选值如下。

```
CSSLOT_CODEDIRECTORY = 0,                    /* slot index for CodeDirectory */
CSSLOT_INFOSLOT = 1,
CSSLOT_REQUIREMENTS = 2,
CSSLOT_RESOURCEDIR = 3,
CSSLOT_APPLICATION = 4,
CSSLOT_ENTITLEMENTS = 5,
CSSLOT_ALTERNATE_CODEDIRECTORIES = 0x1000,   /* first alternate CodeDirectory, if any */
CSSLOT_ALTERNATE_CODEDIRECTORY_MAX = 5,      /* max number of alternate CD slots */
CSSLOT_ALTERNATE_CODEDIRECTORY_LIMIT = CSSLOT_ALTERNATE_CODEDIRECTORIES +
CSSLOT_ALTERNATE_CODEDIRECTORY_MAX,          /* one past the last */
CSSLOT_SIGNATURESLOT = 0x10000,              /* CMS Signature */
```

offset 表示该子条目和 Code Signature 数据的文件偏移。例如，上面解析出来的 5 个子条目的类型和文件偏移如表 6-1 所示。

表 6-1　5 个子条目的类型和文件偏移

type	offset
CSSLOT_CODEDIRECTORY	0x34
CSSLOT_REQUIREMENTS	0x259
CSSLOT_ENTITLEMENTS	0x315
CSSLOT_ALTERNATE_CODEDIRECTORIES	0x512
CSSLOT_SIGNATURESLOT	0x84B

第 1 个 CSSLOT_CODEDIRECTORY 对应的结构体如下。

```
/*
 * C form of a CodeDirectory.
 */
typedef struct __CodeDirectory {
    uint32_t magic;                 /* magic number (CSMAGIC_CODEDIRECTORY) */
    uint32_t length;                /* total length of CodeDirectory blob */
    uint32_t version;               /* compatibility version */
    uint32_t flags;                 /* setup and mode flags */
    uint32_t hashOffset;            /* offset of hash slot element at index zero */
    uint32_t identOffset;           /* offset of identifier string */
    uint32_t nSpecialSlots;         /* number of special hash slots */
    uint32_t nCodeSlots;            /* number of ordinary (code) hash slots */
```

```
    uint32_t codeLimit;             /* limit to main image signature range */
    uint8_t hashSize;               /* size of each hash in bytes */
    uint8_t hashType;               /* type of hash (cdHashType* constants) */
    uint8_t platform;               /* platform identifier; zero if not platform binary */
    uint8_t  pageSize;              /* log2(page size in bytes); 0 => infinite */
    uint32_t spare2;                /* unused (must be zero) */
    /* Version 0x20100 */
    uint32_t scatterOffset;         /* offset of optional scatter vector */
    /* Version 0x20200 */
    uint32_t teamOffset;            /* offset of optional team identifier */
    /* followed by dynamic content as located by offset fields above */
} CS_CodeDirectory;
```

hashOffset 表示 hash 数据和当前结构的偏移为 0xBD。identOffset 是标识符和当前结构的偏移，也就是 0x54+0x34=0x88 处，读取的内容为 com.alonemonkey.MachODemo。nSpecialSlots 和 nCodeSlots 分别指对 5 个条目和代码中每一页（一般页为 0x1000）数据生成哈希的数目，文件总大小为 0x13BB0 减 Code Signature 的大小 0x2B90，等于 0x11020，分为 0x12 页。hashSize 是生成的每个哈希数组的大小 20。hashType 表示哈希算法类型为 SHA-1。

第 2 个条目是 CSSLOT_REQUIREMENTS，它是一个 CS_SuperBlob，保存了整个数据的长度和 REQUIREMENT 数据的个数。每个 REQUIREMENT 数据的结构如下。

```
typedef struct __SC_GenericBlob {
    uint32_t magic;                 /* magic number */
    uint32_t length;                /* total length of blob */
    char data[];
} CS_GenericBlob;
```

这里对应的 magic 是 0xfade0c00（CSMAGIC_REQUIREMENT），长度为 0xA8，data 中保存的是对应长度的数据。

第 3 个条目是 CSSLOT_ENTITLEMENTS，这同样是一个 CS_GenericBlob 结构的数据。magic 对应的是 0xfade7171（CSMAGIC_EMBEDDED_ENTITLEMENTS），长度为 0x1FD。解析出来的数据如下。

```
<!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>9XQEPG2J2J.com.alonemonkey.MachODemo</string>
    <key>com.apple.developer.team-identifier</key>
```

```
    <string>9XQEPG2J2J</string>
    <key>get-task-allow</key>
    <true/>
    <key>keychain-access-groups</key>
    <array>
        <string>9XQEPG2J2J.com.alonemonkey.MachODemo</string>
    </array>
</dict>
</plist>
```

第 4 个条目是一个备用的 CODEDIRECTORY 数据，结构和第 1 个条目一样。

第 5 个条目是 CSSLOT_SIGNATURESLOT。它是签名时使用的证书，结构也是 CS_GenericBlob。magic 对应的是 0xfade0b01（CSMAGIC_BLOBWRAPPER），长度为 0x1264。启动 010 Editor，在右键快捷菜单中选择“Selection”→“Save Selection”选项，将其保存为 cer 文件。使用 Mac 的文件预览功能查看证书，如图 6-15 所示。

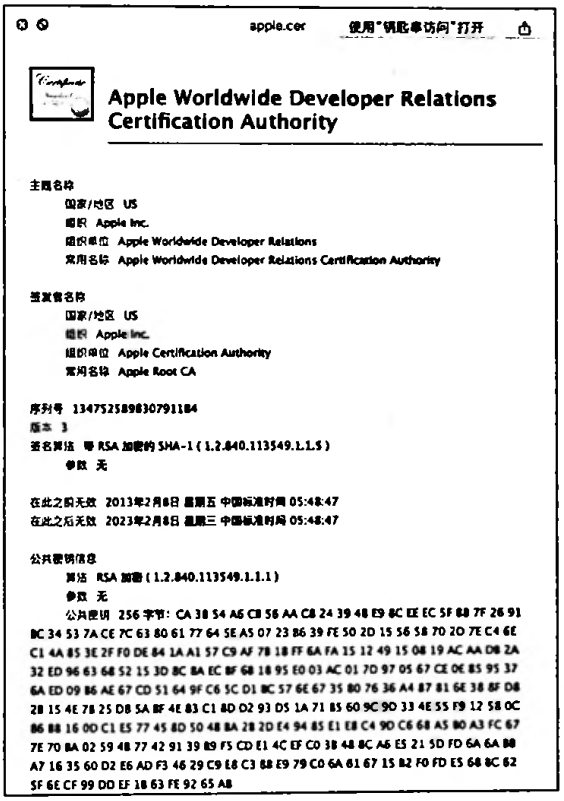


图 6-15　用 MachOView 查看 Code Signature

直接使用 jtool 读取 Mach-O 的签名信息，具体如下。

```
jtool -arch arm64 -v --sig MachODemo
jtool -arch arm64 --ent MachODemo
```

6.3 ARM 汇编

通过前面的学习我们已经知道，要想分析别人的应用，是几乎不可能直接拿到源代码的，因为源代码都已经由编译器编译成机器码的形式保存在文件里了。针对编译生成的可执行文件，当要分析其他逻辑时，只能通过 IDA、Hopper 或动态调试进行。而在进行 IDA 或者 LLDB 调试时，看到的都是类似如下形式的代码调用。

```
    0x1000186a0 <+68>:  adrp   x8, 4
    0x1000186a4 <+72>:  add    x8, x8, #0xf58           ; =0xf58
->  0x1000186a8 <+76>:  ldur   x9, [x29, #-0x8]
    0x1000186ac <+80>:  ldr    x1, [x8]
    0x1000186b0 <+84>:  mov    x0, x9
    0x1000186b4 <+88>:  bl     0x10001a538              ; symbol stub for: objc_msgSend
```

这就是汇编代码。如果想深入了解程序的调用逻辑、参数传递等，就需要阅读和理解汇编代码。

6.3.1 ARM 架构和指令集

所有 iOS 设备甚至所有移动设备的系统都是基于 ARM 的。随着架构的发展，ARM 已经演化出多个版本，每个版本都增加了指令，在升级的同时保持了向后兼容的能力——从 ARMv6 到 ARMv7，再到写作本书时最新的 ARMv8。iPhone 5s 之前的手机都是 ARMv7 及以下的架构，iPhone 5s 及之后的手机都已经支持 ARMv8 架构了。ARMv8 为了保持向后兼容，提供了如下支持。

- 64 位执行状态：AArch64，意味着地址保存在 64 位寄存器中，基本指令集中的指令使用 64 位寄存器进行处理。AArch64 状态支持 A64 指令集，也就是 ARM64 指令集。
- 32 位执行状态：AArch32，意味着地址保存在 32 位寄存器中，基本指令集中的指令使用 32 位寄存器进行处理。AArch32 状态支持 T32 和 A32 指令集，也就是 thumb 和 ARM 指令集。AArch32 的目的是兼容 ARMv7-A，同时增加了一些 AArch64 的特性。

ARM 定义了如下 3 种体系结构。

- A：Application 体系，支持基于内存管理的虚拟内存系统架构，并支持 A64、A32 和 T32

指令集。

- R：Real-time 体系，为具有严格的实时响应限制的深层嵌入式系统提供高性能的计算解决方案，支持 A32 和 T32 指令集。
- M：Microcontroller 体系，面向各类嵌入式应用，支持 T32 指令集的变体。

本书主要针对 ARMv8-A 中的 AArch64 指令集进行讲解。学习 ARM 汇编时，可以到其官网下载官方手册。打开网站 http://infocenter.arm.com/help/index.jsp，在页面左侧单击“ARM 体系结构”目录项，展开“Reference Manuals”项目，单击“ARMv8-A Reference Manual”选项，就可以在页面右侧下载对应的 pdf 文件了（下载前需要注册），如图 6-16 所示。

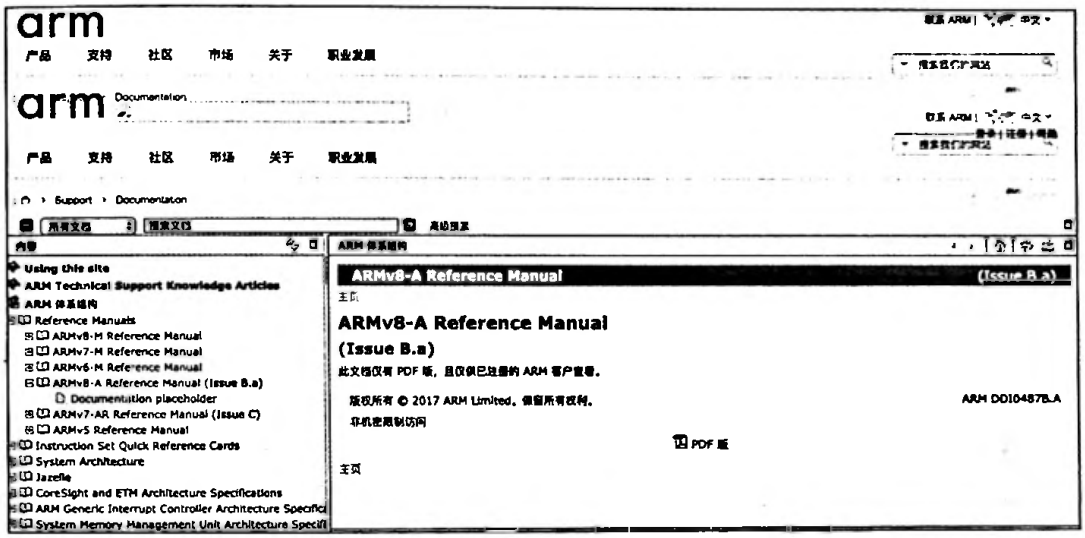


图 6-16　从 ARM 官网下载 ARMv8-A 手册

6.3.2　AArch64 寄存器

基于对安全性的考虑，AArch64 架构支持多级别的执行权限，用 EL0～EL3 表示不同的异常级别。EL0 对应于最低权限级别，通常被描述为无特权级别，编写的应用一般都运行于 EL0 级别。接下来讲解 AArch64 在 EL0 级别可见的寄存器和进程状态。

1. 寄存器

R0～R30 是 31 个通用寄存器，每个寄存器有如下两种访问方式。

- 64 位通用寄存器名为 X0～X30。
- 32 位通用寄存器名为 W0～W30。

这两种访问方式的对应关系如图 6-17 所示。W*n* 表示 X*n* 的低 32 位。通用寄存器 X30 用于程序调用的 link register。link register 是一个特殊的寄存器，用于保存一个函数调用完成时的返回地址。在指令编码中，0b11111（31）用来表示 ZR（0 寄存器），只表示参数为 0，并不表示 ZR 是一个物理寄存器。

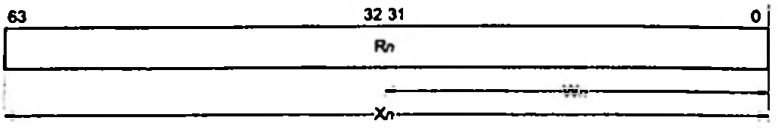


图 6-17　通用寄存器的命名

- SP 为 64 位专用堆栈指针寄存器，可以通过寄存器名 WSP 访问堆栈指针的最低有效 32 位。
- PC 为保存当前指令地址的 64 位程序计数器。程序不能直接写 PC，只能在分支、异常条目或异常返回时更新。不同于 AArch32，后者由于多流水线，需要加上 4 或者 8 的偏移。
- V0~V31 为 32 SIMD&FP registers。V0 ~ V31 寄存器主要用于浮点数的运算，寄存器可以访问的内容如下。其中，由寄存器名称描述的位数不占用整个寄存器，位数是指最低有效位，如图 6-18 所示。
  - 128 位寄存器，Q0 ~ Q31。
  - 64 位寄存器，D0 ~ D31。
  - 32 位寄存器，S0 ~ S31。
  - 16 位寄存器，H0 ~ H31。
  - 8 位寄存器，B0 ~ B31。
  - 128 位元素向量。
  - 64 位元素向量。

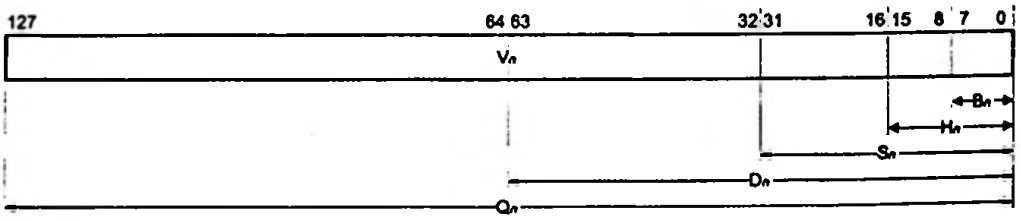


图 6-18　SIMD&FP register 命名

2. 进程状态 PSTATE

PSTATE 不是寄存器，而是进程状态信息的抽象。所有指令集都提供了操作 PSTATE 元素

的指令。以下 PSTATE 信息可在 EL0 级别访问。

- N：正负标志。当两个有符号整数进行运算时，1 表示结果是负数，0 表示结果是正数或者 0。
- Z：零标志。1 表示运算的结果为 0，0 表示运算的结果不为零。
- C：当加法运算产生进位时为 1，否则为 0；当减法运算产生借位时为 0，否则为 1。
- V：在加减运算指令中，当操作数和运算结果为二进制的补码表示的带符号数时，1 表示符号溢出。

以上均为条件标志位。对于 AArch32，在 ARMv7 的 CPSR 寄存器中有一些扩展字段。

6.3.3　指令集编码

我们分析的文件大都是二进制文件，里面是由 0 和 1 组成的机器码。那么，IDA 和 LLDB 是怎么将这些二进制代码翻译成 ARM 汇编的呢？这就需要了解 AArch64 指令集的编码。只有知道 ARM 指令的编码规则，才能将机器码翻译成汇编代码。下面我们就来看看将 080000D0 从二进制转换成的十六进制代表的 ARM 汇编的结果是什么。

因为 iOS 平台是使用小端（Little-endian）排序的，所以需要把这段十六进制内容倒过来看，也就是 D0000008。用计算器显示对应二进制的各个位置的值，如图 6-19 所示。

图 6-19　使用计算器查看十六进制的二进制表示

从 ARM 手册中查找 A64 指令集编码格式，如图 6-20 所示，25 ~ 28 位的值可以确定当前指令的分组，如表 6-2 所示。

表 6-2　根据 op0 解析指令分组

op0	指令分组
00xx	未分配
100x	数据处理（立即数）
101x	分支、异常生成和系统指令

续表

op0	指令分组
x1x0	加载和存储
x101	数据处理（寄存器）
x111	数据处理（浮点数和高级 SIMD）

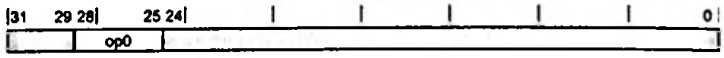


图 6-20 A64 指令编码

从图 6-19 中读取 op0 的值为 0b1000，其对应的是数据处理（立即数）指令。立即数分组的指令编码如图 6-21 所示，对应 op0 的解释如表 6-3 所示。

表 6-3 根据 op0 解析立即数指令分组

op0	指令分组
00x	相对 PC 地址偏移
01x	加/减运算（立即数）
100	逻辑运算（立即数）
101	宽移动（立即数）
110	位域（Bitfield）
111	提取（Extract）

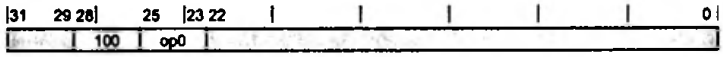


图 6-21 立即数分组指令编码

从图 6-19 中读取 op0 的值为 0b000，其对应的是相对 PC 地址的偏移指令。该分组下面只有两个指令，即 ADR 和 ADRP。根据 31 位的值来判断，如果是 1，就是 ADRP 指令，否则就是 ADR 指令。这里对应的值是 1，所以是 ADRP 指令。ADRP 指令的组成如图 6-22 所示。

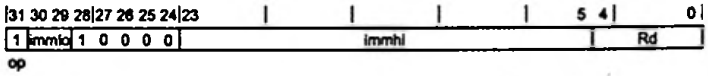


图 6-22 ADRP 指令编码

ADRP 指令的说明如下。

```
Literal variant
```

```
ADRP <Xd>, <label>

Decode for this encoding
integer d = UInt(Rd);
bits(64) imm;
imm = SignExtend(immhi:immlo:Zeros(12), 64);

Assembler symbols
<Xd> Is the 64-bit name of the general-purpose destination register, encoded in the "Rd"
field.
<label> Is the program label whose 4KB page address is to be calculated. Its offset from
the page address of

Operation
bits(64) base = PC[];
base<11:0> = Zeros(12);
X[d] = base + imm;
```

通过上面的解析可知，d 是由 Rd 转换成整数得到的，Rd 对应的值是 0b01000，也就是 8，所以目标寄存器就是 X8。imm 由 immlo 和 immhi 展开并填充低 12 位，结果为 0x2000，所以该指令如下。

```
ADRP x8, 0x2000
```

那么，该指令是什么意思呢？从最后的伪代码解释可知，它的意思是：先拿到当前 PC 寄存器的值，然后将后 3 位置 0，再加上 0x2000，赋值给 x8。假设当前寄存器的值为 0x100062518，那么 x8 为 0x100062000+0x2000=0x100064000。使用 rasm2 进行转换和分析，结果一致，具体如下。使用 kstool 转换的结果也是一样的。

```
➜  rasm2 -a arm -b 64 -d 080000D0
adrp x8, 0x2000

➜  kstool arm64 "adrp x8, 0x2000"
adrp x8, 0x2000 = [ 08 00 00 d0 ]
```

经过分析，相信读者对反汇编工具如何将机器码翻译成汇编代码有了一定的认识。阅读 ARM 手册中对指令的编码解析及对伪代码的解释，也能帮助我们读懂一个陌生的指令。

6.3.4　AArch64 指令

在 AArch64 的指令集中有很多不同的指令。根据指令的功能划分，主要有如下 3 类。

- 数据处理指令
- 加载储存指令
- 跳转指令

由于篇幅有限，本书不会把所有指令都讲一遍。本书主要讲解在实际分析中经常遇到的指令，对那些比较少见的指令，读者可以随时查阅 ARM 手册。

算数指令如表 6-4 所示。

表 6-4 算术指令

指　令	例　子	含　义
ADD	add x0, x1, x2	x0 = x1 + x2
SUB	sub w0, w1, w2	w0 = w1 – w2
CMP	cmp w0, #0x0	w0 和 0 相减，并影响条件标志位
CMN	cmn w0, #0x10	w0 和 0x10 相加，并影响条件标志位
ADDS or SUBS	adds x0, x1, x2	后面带 s 表示计算的结果影响条件标志位

逻辑指令如表 6-5 所示。

表 6-5 逻辑指令

指　令	例　子	含　义
AND	and x0, x1, x2	x0 = x1 & x2 , ANDS 表示影响条件标志位
EOR	eor w0, w1, w2	w0 = w1 ^ w2
ORR	orr w0, w1, w2	w0 = w1 | w2
TST	tst x0, #0x8	测试 bit_3 是否为 0，和 ANDS 的计算方式一样，只是 TST 指令不保存计算结果

数据传输指令如表 6-6 所示。

表 6-6 数据传输指令

指　令	例　子	含　义
MOV	mov x0, x1	x0 = x1
MOVZ	MOVZ Xn, #uimm16{, LSL #pos}	Xn = LSL(uimm16, pos)
MOVN	MOVN Xn, #uimm16{, LSL #pos}	Xn = NOT(LSL(uimm16, pos))
MOVK	MOVK Xn, #uimm16{, LSL #pos}	Xnpos+15:pos = uimm16

地址偏移指令如表 6-7 所示。

表 6-7　地址偏移指令

指　令	例　子	含　义
ADR	mov x1, 0x1234	x1 = pc + 0x1234
ADRP	mov x1, 0x1234	base = PC[11:0] = ZERO(12); x1 = base + 0x1234

移位运算指令如表 6-8 所示。

表 6-8　移位运算指令

指　令	例　子	含　义
ASR	ASR Xd, Xn, #uimm	算术右移，移位过程中符号位不变
LSL	LSL Xd, Xn, #uimm	逻辑左移，移位后寄存器空出的低位补 0
LSR	LSR Xd, Xn, #uimm	逻辑右移，移位后寄存器空出的高位补 0
ROR	ROR Xd, Xn, #uimm	循环右移，从右端移出的位将被插入左侧空出的位

常见的加载储存指令如表 6-9 所示。

表 6-9　常见的加载储存指令

指　令	例　子	含　义
LDR	LDR Xn/Wn, addr	从 addr 地址中读取 8/4 字节内容到 Xn/Wn 中
STR	STR Xn/Wn, addr	将 Xn/Wn 写入 addr 地址指向的内存
LDUR	LDUR Xn, [base,#simm9]	从 base+simm9 地址中读取数据到 Xn。Unscaled 表示不需要对齐，读取的数据是多少，这里就是多少
STUR	STUR Xn, [base,#simm9]	将 Xn 写入 base+simm9 地址指向的内存
STP	STP Xn1, Xn2, addr	将 Xn1 和 Xn2 写入地址为 addr 的内存
LDP	LDP Xn1, Xn2, addr	从地址 addr 处读取内存到 Xn1 和 Xn2 中

加载指令可以使用立即数、寄存器和对齐寄存器作为偏移。我们看看下面几条指令的执行和结果。

```
ldr    x1, [x2, #4]          ; 读取地址 x2+4 处的值到 x1 中
ldr    x1, [x2, x3]          ; 读取地址为 x2+x3 处的值到 x1 中
ldr    x1, [x2, x3, LSL #3]  ; 读取地址为 x2+x3×8 处的值到 x1 中
ldr    x1, [x2, #4]!         ; 读取地址为 x2+4 处的值到 x1 中，然后执行 x2=x2+4
ldr    x1, [sp], #0x4        ; 读取地址为 sp 的值到 x1 中，然后执行 sp=sp+4
```

条件跳转指令如表 6-10 所示。

表 6-10 条件跳转指令

指 令	例 子	含 义
B.cond	B.cond label	如果 cond 条件为真，则跳转到 label
CBNZ	CBNZ Xn, label	如果 Xn 不为 0，则跳转到 label
CBZ	CBZ Xn, label	如果 Xn 为 0，则跳转到 label
TBNZ	TBNZ Xn, #uimm6, label	如果 Xn[uimm6] != 0，则跳转到 label
TBZ	TBZ Xn, #uimm6, label	如果 Xn[uimm6] = 0，则跳转到 label

无条件跳转指令如表 6-11 所示。

表 6-11 无条件跳转指令

指 令	例 子	含 义
B	B label	无条件跳转
BL	BL label	无条件跳转，会将下一条指令地址写到 X30(link register) 处
BLR	BLR Xn	无条件跳转到 Xn 寄存器的地址，会将下一条指令地址写到 X30(link register) 处
BR	BR Xn	无条件跳转到 Xn 寄存器的地址
RET	ret	子程序返回

6.3.5 栈和方法

栈是指被限定只能在一端进行插入或删除操作的线性表。对栈来说，可以进行插入或删除操作的一端称为栈顶，另一端称为栈底。因为堆栈只能在一端进行操作，所以按照后进先出的原理运作，也就是最后入栈的元素会先出栈。但是，本节所讲的栈不是数据结构中的栈，而是进程中的一种特殊的内存区域，对它的操作和数据结构中的栈很像，在调用一个函数时用于保存一些临时的数据，例如局部变量和上下文环境。

根据栈的增长方向和栈顶指针指向的位置，可以将其分为如下 4 种类型。

- 向高地址方向生长，称为递增堆栈。
- 向低地址方向生长，称为递减堆栈。
- 堆栈指针指向最后压入堆栈的有效数据项，称为满堆栈。
- 堆栈指针指向下一个要放入的空位置，称为空堆栈。

而 ARM 堆栈具有后进先出和满递减的特点，如图 6-23 所示。将其想象成一个函数栈，有如下特点。

- 栈中的元素按 ABCD 的顺序入栈，按 DCBA 的顺序出栈。
- 栈是向低地址方向生长的。
- SP 指向栈顶的元素，其他元素通过 SP + offset 获取。

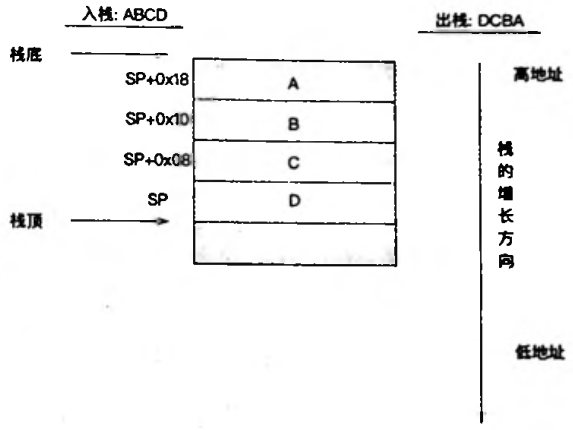


图 6-23　ARM 堆栈满递减

函数的调用会开辟栈帧。在 AArch64 中，函数的参数是通过 x0 ~ x7 传递的（不考虑浮点数和向量寄存器）。还有一些与函数堆栈相关的寄存器，列举如下。

- PC 寄存器（Program Counter）：记录当前执行代码的地址。
- SP 寄存器（Stack Pointer）：指向栈帧的指针，在内存操作指令中通过 x31 寄存器来访问。
- LR 寄存器（Link Register）：指向返回地址，对应于寄存器 x30。
- FP 寄存器（Frame Pointer）：指向栈帧的底部，对应于寄存器 x29。

从 ARM 官网可以找到函数调用前后栈的变化，如图 6-24 所示。其中，R7 是 AArch32 的 FP 寄存器，对应于 AArch64 中的 x29。

一个栈帧包括以下部分。

- 参数区（parameter area）：存放调用函数传递的参数。
- 连接区（linkage area）：存放调用者（caller）的下一条指令。
- 栈帧指针存放区（frame pointer）：存放调用函数的栈帧的底部。
- 寄存器存储区（saved registers area）：被调用函数（callee）返回需要恢复的寄存器内容。
- 局部存储区（local storage area）：用于存放被调用函数（callee）的局部变量。

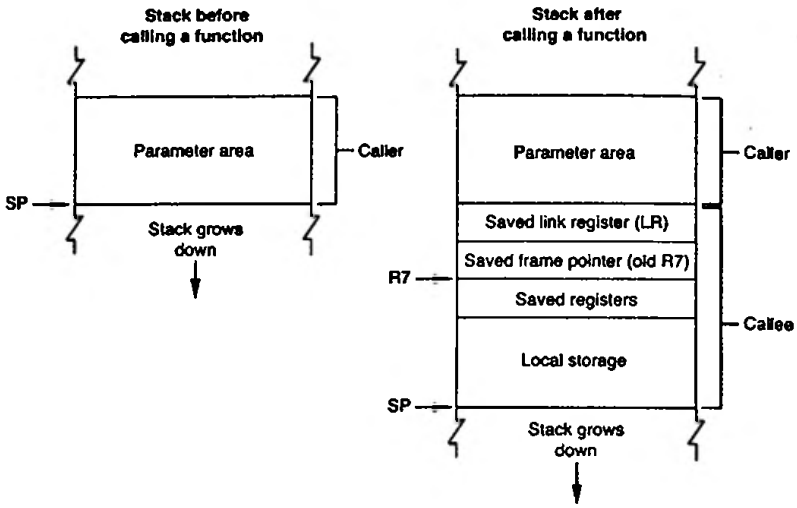


图 6-24 函数调用栈的变化

下面通过编写一个例子程序来了解函数调用的参数是怎么传递的及堆栈中的元素是怎么存放和获取的，代码如下。

```
int add(int a, int b){
    return a + b;
}

int main(int argc, char * argv[]) {
    int c = add(3, 4);
    printf("%d",c);
    return 0;
}
```

运行后显示汇编代码，具体如下。笔者为这段代码加上了注释，读者可以跟着注释边调试边理解整个流程。

```
ArmDemo`main:
    0x1000ac7c0 <+0>:  sub    sp, sp, #0x30             ; =0x30，开辟大小为 0x30 的栈帧空间
    0x1000ac7c4 <+4>:  stp    x29, x30, [sp, #0x20]     ; 保存 FP 和 LR 寄存器
    0x1000ac7c8 <+8>:  add    x29, sp, #0x20            ; =0x20，设置新的 FP 寄存器，也就是新
的栈帧的底部
    0x1000ac7cc <+12>: orr    w8, wzr, #0x3             ; 将 0 和 3 异或，将结果赋给 w8，等同于
mov w8,0x03
    0x1000ac7d0 <+16>: orr    w9, wzr, #0x4             ; mov w9, 0x04
    0x1000ac7d4 <+20>: stur   wzr, [x29, #-0x4]
    0x1000ac7d8 <+24>: stur   w0, [x29, #-0x8]          ; 保存 w0
```

```
    0x1000ac7dc <+28>: str    x1, [sp, #0x10]          ; 保存 x1
->  0x1000ac7e0 <+32>: mov    x0, x8                   ; 赋值参数 1 为 3
    0x1000ac7e4 <+36>: mov    x1, x9                   ; 赋值参数 2 为 4
    0x1000ac7e8 <+40>: bl     0x1000ac7a0              ; add at main.m:12，调用 add 方法
    0x1000ac7ec <+44>: str    w0, [sp, #0xc]           ; 将结果 w0 保存到[sp, #0xc]中
    0x1000ac7f0 <+48>: ldr    w8, [sp, #0xc]
    0x1000ac7f4 <+52>: mov    x30, x8
    0x1000ac7f8 <+56>: mov    x10, sp
    0x1000ac7fc <+60>: str    x30, [x10]               ; 保存结果，供 printf 获取
    0x1000ac800 <+64>: adrp   x0, 3
    0x1000ac804 <+68>: add    x0, x0, #0xcd9           ; =0xcd9 adrp 和 add 组合操作，读取"%d"
    0x1000ac808 <+72>: bl     0x1000ae5bc              ; symbol stub for: printf
    0x1000ac80c <+76>: mov    w8, #0x0
    0x1000ac810 <+80>: str    w0, [sp, #0x8]
    0x1000ac814 <+84>: mov    x0, x8                   ; 返回值 0
    0x1000ac818 <+88>: ldp    x29, x30, [sp, #0x20]    ; 还原 FP 和 LR 寄存器
    0x1000ac81c <+92>: add    sp, sp, #0x30            ; =0x30，释放栈帧空间
    0x1000ac820 <+96>: ret                             ; 返回
```

其中，对 add 函数的汇编和解释如下。

```
ArmDemo`add:
->  0x1000ac7a0 <+0>:  sub    sp, sp, #0x10            ; =0x10，开启大小为 x010 的栈帧
    0x1000ac7a4 <+4>:  str    w0, [sp, #0xc]           ; 保存参数 1
    0x1000ac7a8 <+8>:  str    w1, [sp, #0x8]           ; 保存参数 2
    0x1000ac7ac <+12>: ldr    w0, [sp, #0xc]           ; 获取参数 1
    0x1000ac7b0 <+16>: ldr    w1, [sp, #0x8]           ; 获取参数 1
    0x1000ac7b4 <+20>: add    w0, w0, w1               ; w0=w0+w1
    0x1000ac7b8 <+24>: add    sp, sp, #0x10            ; =0x10，释放栈帧空间
    0x1000ac7bc <+28>: ret                             ; 返回
```

堆栈内存的对应结构如图 6-25 所示。通过分析可以得到函数调用前和调用结束的操作，具体如下。

01 函数调用前（Prologs）

①开辟栈帧空间。

②保存 FP 和 LR 寄存器，以便找到上一个栈帧和返回地址。

③设置新的 FP 寄存器。

④保存子函数会用到的寄存器（当前没有）。

⑤保存局部变量或参数。

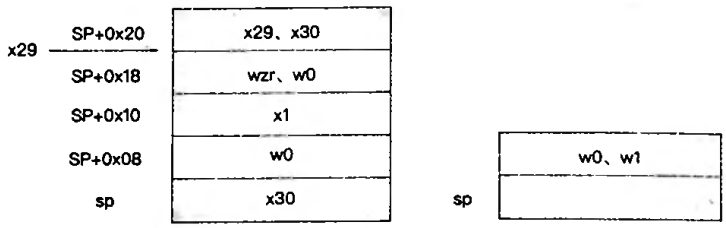


图 6-25 堆栈内存的对应结构

02 函数调用结束（Epilogs）

①还原 FP 和 LR 寄存器。

②释放栈帧空间。

③跳到 LR 子程序返回。

在当前分析的程序中有 2 个参数。如果参数超过 8 个，多余的参数会放在哪里呢？修改例子代码，具体如下。

```
int add(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k){ //11
args
    return a + b + j + k;
}

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

    int c = add(3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);
    printf("%d",c);
    return 0;
}
```

查看反汇编代码，具体如下。使用 register read 命令可以读取寄存器中的参数。多余的参数直接放在了栈中，可以从栈中读取。

```
ArmDemo`main:
    0x100090790 <+0>:   sub    sp, sp, #0x40                ; =0x40
    0x100090794 <+4>:   stp    x29, x30, [sp, #0x30]
    0x100090798 <+8>:   add    x29, sp, #0x30               ; =0x30
    0x10009079c <+12>:  orr    w8, wzr, #0x3
    0x1000907a0 <+16>:  orr    w9, wzr, #0x4
    0x1000907a4 <+20>:  mov    w2, #0x5
    0x1000907a8 <+24>:  orr    w3, wzr, #0x6
    0x1000907ac <+28>:  orr    w4, wzr, #0x7
```

```
    0x1000907b0 <+32>:  orr    w5, wzr, #0x8
    0x1000907b4 <+36>:  mov    w6, #0x9
    0x1000907b8 <+40>:  mov    w7, #0xa
    0x1000907bc <+44>:  mov    w10, #0xb
    0x1000907c0 <+48>:  orr    w11, wzr, #0xc
    0x1000907c4 <+52>:  mov    w12, #0xd
    0x1000907c8 <+56>:  stur   wzr, [x29, #-0x4]
    0x1000907cc <+60>:  stur   w0, [x29, #-0x8]
    0x1000907d0 <+64>:  stur   x1, [x29, #-0x10]
    0x1000907d4 <+68>:  mov    x0, x8
    0x1000907d8 <+72>:  mov    x1, x9
    0x1000907dc <+76>:  str    w10, [sp]
    0x1000907e0 <+80>:  str    w11, [sp, #0x4]
    0x1000907e4 <+84>:  str    w12, [sp, #0x8]
->  0x1000907e8 <+88>:  bl     0x100090730               ; add at main.m:12
    0x1000907ec <+92>:  stur   w0, [x29, #-0x14]
    0x1000907f0 <+96>:  ldur   w8, [x29, #-0x14]
    0x1000907f4 <+100>: mov    x30, x8
    0x1000907f8 <+104>: mov    x13, sp
    0x1000907fc <+108>: str    x30, [x13]
    0x100090800 <+112>: adrp   x0, 3
    0x100090804 <+116>: add    x0, x0, #0xcd9            ; =0xcd9
    0x100090808 <+120>: bl     0x1000925bc               ; symbol stub for: printf
    0x10009080c <+124>: mov    w8, #0x0
    0x100090810 <+128>: str    w0, [sp, #0x18]
    0x100090814 <+132>: mov    x0, x8
    0x100090818 <+136>: ldp    x29, x30, [sp, #0x30]
    0x10009081c <+140>: add    sp, sp, #0x40             ; =0x40
    0x100090820 <+144>: ret
```

```
; 读取寄存器参数
(lldb) register read
General Purpose Registers:
        x0 = 0x0000000000000003
        x1 = 0x0000000000000004
        x2 = 0x0000000000000005
        x3 = 0x0000000000000006
        x4 = 0x0000000000000007
        x5 = 0x0000000000000008
        x6 = 0x0000000000000009
        x7 = 0x000000000000000a

; 读取栈参数
(lldb) memory read -f A $sp $fp
```

```
0x16fd73a60: 0x0000000c0000000b              ; 此处保存的是参数 i 和 j
0x16fd73a68: 0x000000000000000d              ; 此处保存的是参数 k
0x16fd73a70: 0x0000000000000000
0x16fd73a78: 0x0000000000000000
0x16fd73a80: 0x000000016fd73ad0
0x16fd73a88: 0x0000000000000001
```

如果参数是小数，就不是通过 x0～x7 来传参了。修改例子代码如下，在这里，小数是通过 d0 和 d1 传递的，可以执行 p $d0 命令打印对应的值。

```
double add(double a, double b){
    return a + b;
}
int main(int argc, char * argv[]) {
    double c = add(13.14, 5.20);
    printf("%f",c);
    return 0;
}
```

在生成的汇编代码中查看参数是如何传递的，具体如下。

```
ArmDemo`main:
    0x1000247b0 <+0>:  sub    sp, sp, #0x40                ; =0x40
    0x1000247b4 <+4>:  stp    x29, x30, [sp, #0x30]
    0x1000247b8 <+8>:  add    x29, sp, #0x30               ; =0x30
    0x1000247bc <+12>: adrp   x8, 3
    0x1000247c0 <+16>: ldr    d0, [x8, #0xcc8]             ; 读取参数 1
    0x1000247c4 <+20>: adrp   x8, 3
    0x1000247c8 <+24>: ldr    d1, [x8, #0xcd0]             ; 读取参数 2
    0x1000247cc <+28>: stur   wzr, [x29, #-0x4]
    0x1000247d0 <+32>: stur   w0, [x29, #-0x8]
    0x1000247d4 <+36>: stur   x1, [x29, #-0x10]
->  0x1000247d8 <+40>: bl     0x100024790                  ; add at main.m:16
    0x1000247dc <+44>: str    d0, [sp, #0x18]
    0x1000247e0 <+48>: ldr    d0, [sp, #0x18]
    0x1000247e4 <+52>: mov    x8, sp
    0x1000247e8 <+56>: str    d0, [x8]
    0x1000247ec <+60>: adrp   x0, 3
    0x1000247f0 <+64>: add    x0, x0, #0xce0               ; =0xce0
    0x1000247f4 <+68>: bl     0x1000265a8                  ; symbol stub for: printf
    0x1000247f8 <+72>: mov    w9, #0x0
    0x1000247fc <+76>: str    w0, [sp, #0x14]
    0x100024800 <+80>: mov    x0, x9
    0x100024804 <+84>: ldp    x29, x30, [sp, #0x30]
```

```
    0x100024808 <+88>: add    sp, sp, #0x40                ; =0x40
    0x10002480c <+92>: ret

ArmDemo`add:
->  0x100024790 <+0>:  sub    sp, sp, #0x10                ; =0x10
    0x100024794 <+4>:  str    d0, [sp, #0x8]
    0x100024798 <+8>:  str    d1, [sp]
    0x10002479c <+12>: ldr    d0, [sp, #0x8]
    0x1000247a0 <+16>: ldr    d1, [sp]
    0x1000247a4 <+20>: fadd   d0, d0, d1                   ; 将参数1和2相加. 这里使用的是
fadd 小数的加法
    0x1000247a8 <+24>: add    sp, sp, #0x10                ; =0x10
    0x1000247ac <+28>: ret
```

函数调用之后，一般都是通过 x0 寄存器返回结果的。但是，x0 寄存器最多只能保存一个地址或者 8 字节的数据。如果返回的数据大于 8 字节，会怎么返回？下面用 3 个例子来测试大小不同的返回数据的参数传递。

```
MyStruct0 genStruct0(int a, int b){
    MyStruct0 s;
    s.a = a;
    s.b = b;
    return s;
}

MyStruct1 genStruct1(int a, int b, int c, int d){
    MyStruct1 s;
    s.a = a;
    s.b = b;
    s.c = c;
    s.d = d;
    return s;
}

MyStruct2 genStruct2(int a, int b, int c, int d, int e, int f){
    MyStruct2 s;
    s.a = a;
    s.b = b;
    s.c = c;
    s.d = d;
    s.e = e;
    s.f = f;
    return s;
```

```
}

    MyStruct0 s0;
    s0 = genStruct0(3, 4);

    printf("%d %d", s0.a, s0.b);

    MyStruct1 s1;
    s1 = genStruct1(3, 4, 5, 6);

    printf("%d %d %d %d", s1.a, s1.b, s1.c, s1.d);

    MyStruct2 s2;
    s2 = genStruct2(3, 4, 5, 6, 7, 8);

    printf("%d %d %d %d %d %d", s2.a, s2.b, s2.c, s2.d, s2.e, s2.f);
```

我们只看最后返回结果的汇编代码。先来看 genStruct0 返回的汇编代码，具体如下。打印 $x0 寄存器的值，可以看到返回的是 3 和 4。

```
    0x10006831c <+40>: str    w0, [sp, #0x1c]
    0x100068320 <+44>: ldr    x0, [sp, #0x18]
->  0x100068324 <+48>: add    sp, sp, #0x20             ; =0x20
    0x100068328 <+52>: ret

(lldb) p/x $x0
(unsigned long) $3 = 0x0000000400000003
```

再来看 genStruct1 返回的汇编代码，具体如下。打印 x0 和 x1 寄存器的值，可以看到返回的是 3、4、5、6。

```
0x100068360 <+52>: ldr    q0, [sp]
0x100068364 <+56>: str    q0, [sp, #0x20]
0x100068368 <+60>: ldr    x0, [sp, #0x20]
0x10006836c <+64>: ldr    x1, [sp, #0x28]
0x100068370 <+68>: add    sp, sp, #0x30             ; =0x30
0x100068374 <+72>: ret

(lldb) p/x $x0
(unsigned long) $4 = 0x0000000400000003
(lldb) p/x $x1
(unsigned long) $5 = 0x0000000600000005
```

最后来看 genStruct2 返回的汇编，具体如下。0x1000683e0 处将 x1 处大小为 x2（24）的内存复制到 x0（也就是 x8）中，通过 x8 寄存器指向的内存来传递返回值。

```
0x1000683d4 <+92>:  mov    x0, x8
0x1000683d8 <+96>:  mov    x1, x10
0x1000683dc <+100>: mov    x2, x9
0x1000683e0 <+104>: bl     0x10006a3b4               ; symbol stub for: memcpy
0x1000683e4 <+108>: ldp    x29, x30, [sp, #0x30]
0x1000683e8 <+112>: add    sp, sp, #0x40             ; =0x40
0x1000683ec <+116>: ret

(lldb) x/4xg $x1
0x16fd9f910: 0x0000000400000003 0x0000000600000005
0x16fd9f920: 0x0000000800000007 0x0000000700000008
(lldb) p $x2
(unsigned long) $10 = 24
(lldb) ni
(lldb) x/4xg $x0
0x16fd9f9c0: 0x0000000400000003 0x0000000600000005
0x16fd9f9d0: 0x0000000800000007 0x00000001001591fc
```

AArch64 指令集中包含的指令众多，如果读者遇到不懂的指令，直接在 ARM 手册中搜索相应的指令即可找到对应的解释。

6.3.6　Objective-C 汇编

在 OC 中，调用一个函数是通过 [ClassName methodName:arg0] 的方式实现的。我们编写一个简单的 OC 函数调用，看看它的汇编代码是怎样的，具体如下。

```
@implementation ViewController

-(void)A:(NSString*) arg0 B:(NSString*) arg1{
    NSLog(@"%@ %@",arg0,arg1);
}

- (void)viewDidLoad {
    [super viewDidLoad];
    [self A:@"1" B:@"2"];
}

@end
```

对应的汇编代码如下。

```
ArmDemo`-[ViewController viewDidLoad]:
    0x100010604 <+0>:   sub    sp, sp, #0x30                ; =0x30
    0x100010608 <+4>:   stp    x29, x30, [sp, #0x20]
    0x10001060c <+8>:   add    x29, sp, #0x20               ; =0x20
    0x100010610 <+12>:  mov    x8, sp
    0x100010614 <+16>:  adrp   x9, 4
    0x100010618 <+20>:  add    x9, x9, #0xfb8               ; =0xfb8
    0x10001061c <+24>:  adrp   x10, 5
    0x100010620 <+28>:  add    x10, x10, #0x88              ; =0x88
    0x100010624 <+32>:  stur   x0, [x29, #-0x8]
    0x100010628 <+36>:  str    x1, [sp, #0x10]
    0x10001062c <+40>:  ldur   x0, [x29, #-0x8]
    0x100010630 <+44>:  str    x0, [sp]
    0x100010634 <+48>:  ldr    x10, [x10]
    0x100010638 <+52>:  str    x10, [sp, #0x8]
    0x10001063c <+56>:  ldr    x1, [x9]
    0x100010640 <+60>:  mov    x0, x8
    0x100010644 <+64>:  bl     0x10001252c          ; symbol stub for: objc_msgSendSuper2
    0x100010648 <+68>:  adrp   x8, 4
    0x10001064c <+72>:  add    x8, x8, #0x230       ; =0x230
    0x100010650 <+76>:  adrp   x9, 4
    0x100010654 <+80>:  add    x9, x9, #0x250       ; =0x250
    0x100010658 <+84>:  adrp   x10, 4
    0x10001065c <+88>:  add    x10, x10, #0xfc0     ; =0xfc0
->  0x100010660 <+92>:  ldur   x0, [x29, #-0x8]
    0x100010664 <+96>:  ldr    x1, [x10]
    0x100010668 <+100>: mov    x2, x8
    0x10001066c <+104>: mov    x3, x9
    0x100010670 <+108>: bl     0x100012520              ; symbol stub for: objc_msgSend
    0x100010674 <+112>: ldp    x29, x30, [sp, #0x20]
    0x100010678 <+116>: add    sp, sp, #0x30            ; =0x30
    0x10001067c <+120>: ret
```

其中，OC 函数的调用已经转换成了 C 函数 objc_msgSend 的调用，所以在动态调试中可以通过调用 objc_msgSend 的地址来判断当前调用的 OC 函数。在 0x100010670 处打印当前寄存器的值，具体如下。

```
(lldb) register read
General Purpose Registers:
       x0 = 0x000000013dd08d80
       x1 = 0x0000000100012ab1  "A:B:"
```

```
        x2 = 0x0000000100014230  @"'1'"
        x3 = 0x0000000100014250  @"'2'"
        ......
(lldb) po 0x000000013dd08d80
<ViewController: 0x13dd08d80>
(lldb) x/s $x1
0x100012ab1: "A:B:"
```

可以看到，x0 对应于调用的 ViewController 对象，x1 表示调用的方法名字，其他参数存储在 x2 ~ x7 或者堆栈里（浮点数除外）。

6.4　hook

“hook”直译为“钩子”。在程序设计中，hook 是指改变程序运行流程的一种技术。例如，一个正常的程序运行流程是 A→B→C，通过 hook 技术可以让程序的执行变成 A→自己的程序代码→B→C。在这个过程中，可以获取 A 传给 B 的数据，对其进行修改再传给 B，而 A、B 是不会感知这个过程的。所以，通过 hook 技术可以达到改变程序逻辑、修改程序数据的目的。那么，在 iOS 系统中又有哪些 hook 技术可以达到这种效果呢？在本节中将介绍在 iOS 系统中常用的 hook 技术及具体应用，并讲解其实现原理。

在 iOS 中常见的 hook 方式如下。

- Method Swizzle：这种技术在 OC 的 runtime 特性中已经讲到，可以通过修改 OC 函数的 IMP 达到替换方法实现的效果。这种技术针对 OC 函数，在越狱和非越狱平台上都可以使用。
- fishhook：通过修改内存中懒加载和非懒加载符号表指针所指向的地址来达到修改方法的目的，作用于主模块懒加载和非懒加载表中的符号，在越狱和非越狱平台上都可以使用。
- Cydia Substrate：通过 inline hook 的方式修改目标函数内存中的汇编指令，使其跳转到自己的代码块，以达到修改程序的目的，同时支持 Method Swizzle。inline hook 的方式只能在越狱平台上使用（在非越狱平台上静态修改文件进行跳转时也可以）。

6.4.1　Method Swizzle

Method Swizzle 的演示和原理已经在 OC 的 runtime 中讲过了，这里不再重述。我们需要思考一个问题：如果一个函数多次被 hook，其中的调用会是怎样的呢？看看如下实例代码。

```
@implementation ViewController
```

```
+(void)load{
    [HookManager swizzleMethod:@selector(clickMe:) ofClass:[self class]
withMethod:@selector(hookClickMe:)];
    [HookManager swizzleMethod:@selector(clickMe:) ofClass:[self class]
withMethod:@selector(newHookClickMe:)];
}

- (IBAction)clickMe:(UIButton *)sender {
    NSLog(@"origin clickme");
}

-(void)hookClickMe:(UIButton*) sender{
    NSLog(@"hook Click Me...");
    [self hookClickMe:sender];
}

-(void)newHookClickMe:(UIButton*) sender{
    NSLog(@"new hook Click Me...");
    [self newHookClickMe:sender];
}

@end
```

在 load 方法中，对 clickMe 进行了 2 次 hook，分别将其实现替换为 hookClickMe 和 newHook ClickMe。单击按钮，触发 clickMe 的调用日志，输出如下。

```
2017-09-10 23:58:58.311338+0800 MethodSwizzling[995:235528] new hook Click Me...
2017-09-10 23:58:58.311536+0800 MethodSwizzling[995:235528] hook Click Me...
2017-09-10 23:58:58.311606+0800 MethodSwizzling[995:235528] origin clickme
```

此处产生了链式调用，首先调用的是最后替换的函数 newHookClickMe。因为新的函数里面都调用了原来的实现，所以后面分别调用了 hookClickMe 和 clickMe。也就是说，多次对同一个函数进行 hook 时，只要新的函数都调用了原实现，就是链式调用；否则，哪里没有调用原实现，链式调用就会在哪里中断。

6.4.2 fishhook

fishhook 是 Facebook 开源的一个非常小的重新绑定动态符号的库。在前面分析 dyld 时发现，使用 __DATA,__interpose 可以修改动态符号表中的指针，但只能是注入的库主模块，本身这样是不行的，所以，需要通过 fishhook 来完成动态修改绑定符号指针的工作。为了帮助读者理解

fishhook 的原理，我们先来看一下官方给出的例子，具体如下，然后根据实例进行分析。

```
#import <dlfcn.h>

#import <UIKit/UIKit.h>

#import "AppDelegate.h"
#import "fishhook.h"

static int (*orig_close)(int);
static int (*orig_open)(const char *, int, ...);

int my_close(int fd) {
  printf("Calling real close(%d)\n", fd);
  return orig_close(fd);
}

int my_open(const char *path, int oflag, ...) {
  va_list ap = {0};
  mode_t mode = 0;

  if ((oflag & O_CREAT) != 0) {
    // mode only applies to O_CREAT
    va_start(ap, oflag);
    mode = va_arg(ap, int);
    va_end(ap);
    printf("Calling real open('%s', %d, %d)\n", path, oflag, mode);
    return orig_open(path, oflag, mode);
  } else {
    printf("Calling real open('%s', %d)\n", path, oflag);
    return orig_open(path, oflag, mode);
  }
}

int main(int argc, char * argv[])
{
  @autoreleasepool {
    rebind_symbols((struct rebinding[2]){{"close", my_close, (void *)&orig_close}, {"open",
my_open, (void *)&orig_open}}, 2);

    // Open our own binary and print out first 4 bytes (which is the same
    // for all Mach-O binaries on a given architecture)
    int fd = open(argv[0], O_RDONLY);
    uint32_t magic_number = 0;
```

```
    read(fd, &magic_number, 4);
    printf("Mach-O Magic Number: %x \n", magic_number);
    close(fd);

    return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
  }
}
```

运行之后，就能在控制台看到 hook 的日志输出了，具体如下。

```
Calling real
open('/var/containers/Bundle/Application/5BB3B2CC-6276-4E2B-AB47-90AC68B31CC6/FishHook.
app/FishHook', 0)
Mach-O Magic Number: feedfacf
Calling real close(3)
......
```

接下来，通过源代码一步步分析 fishhook 的实现原理。首先，使用 fishhook 的函数 rebind_symbols，传入 2 个 rebinding 的结构体，分别指定需要替换的符号和自己实现的函数。如果是第 1 次调用，就注册 image 添加的回调，否则遍历所有加载的模块。相关代码如下。

```
int rebind_symbols(struct rebinding rebindings[], size_t rebindings_nel) {
  int retval = prepend_rebindings(&_rebindings_head, rebindings, rebindings_nel);
  if (retval < 0) {
    return retval;
  }
  // If this was the first call, register callback for image additions (which is also invoked
for
  // existing images, otherwise, just run on existing images
  if (!_rebindings_head->next) {
    _dyld_register_func_for_add_image(_rebind_symbols_for_image);
  } else {
    uint32_t c = _dyld_image_count();
    for (uint32_t i = 0; i < c; i++) {
      _rebind_symbols_for_image(_dyld_get_image_header(i),
_dyld_get_image_vmaddr_slide(i));
    }
  }
  return retval;
}
```

最后，都会调用 rebind_symbols_for_image 函数对某个模块中的符号表指针进行替换。该函

数首先获取 __LINKEDIT、符号表、间接跳转表、字符串表在内存中的位置，然后调用 perform_rebinding_with_section 分别对懒加载表和非懒加载表进行替换，代码如下。

```
static void rebind_symbols_for_image(struct rebindings_entry *rebindings,
                                     const struct mach_header *header,
                                     intptr_t slide) {
  Dl_info info;
  if (dladdr(header, &info) == 0) {
    return;
  }

  segment_command_t *cur_seg_cmd;
  segment_command_t *linkedit_segment = NULL;
  struct symtab_command* symtab_cmd = NULL;
  struct dysymtab_command* dysymtab_cmd = NULL;

  uintptr_t cur = (uintptr_t)header + sizeof(mach_header_t);
  for (uint i = 0; i < header->ncmds; i++, cur += cur_seg_cmd->cmdsize) {
    cur_seg_cmd = (segment_command_t *)cur;
    //获取__LINKEDIT
    if (cur_seg_cmd->cmd == LC_SEGMENT_ARCH_DEPENDENT) {
      if (strcmp(cur_seg_cmd->segname, SEG_LINKEDIT) == 0) {
        linkedit_segment = cur_seg_cmd;
      }
    //获取符号表
    } else if (cur_seg_cmd->cmd == LC_SYMTAB) {
      symtab_cmd = (struct symtab_command*)cur_seg_cmd;
    //获取动态符号表
    } else if (cur_seg_cmd->cmd == LC_DYSYMTAB) {
      dysymtab_cmd = (struct dysymtab_command*)cur_seg_cmd;
    }
  }

  if (!symtab_cmd || !dysymtab_cmd || !linkedit_segment ||
      !dysymtab_cmd->nindirectsyms) {
    return;
  }

  // Find base symbol/string table addresses
  //获取 linkedit 基址
  uintptr_t linkedit_base = (uintptr_t)slide + linkedit_segment->vmaddr -
linkedit_segment->fileoff;
  //符号表在内存中的位置
  nlist_t *symtab = (nlist_t *)(linkedit_base + symtab_cmd->symoff);
```

```
  //字符串表在内存中的位置
  char *strtab = (char *)(linkedit_base + symtab_cmd->stroff);

  //动态符号表间接跳转表在内存中的位置
  // Get indirect symbol table (array of uint32_t indices into symbol table)
  uint32_t *indirect_symtab = (uint32_t *)(linkedit_base + dysymtab_cmd->indirectsymoff);

  cur = (uintptr_t)header + sizeof(mach_header_t);
  for (uint i = 0; i < header->ncmds; i++, cur += cur_seg_cmd->cmdsize) {
    cur_seg_cmd = (segment_command_t *)cur;
    if (cur_seg_cmd->cmd == LC_SEGMENT_ARCH_DEPENDENT) {
      if (strcmp(cur_seg_cmd->segname, SEG_DATA) != 0 &&
          strcmp(cur_seg_cmd->segname, SEG_DATA_CONST) != 0) {
        continue;
      }
      //分别绑定懒加载表和非懒加载表
      for (uint j = 0; j < cur_seg_cmd->nsects; j++) {
        section_t *sect =
          (section_t *)(cur + sizeof(segment_command_t)) + j;
        if ((sect->flags & SECTION_TYPE) == S_LAZY_SYMBOL_POINTERS) {
          perform_rebinding_with_section(rebindings, sect, slide, symtab, strtab,
indirect_symtab);
        }
        if ((sect->flags & SECTION_TYPE) == S_NON_LAZY_SYMBOL_POINTERS) {
          perform_rebinding_with_section(rebindings, sect, slide, symtab, strtab,
indirect_symtab);
        }
      }
    }
  }
}
```

在 perform_rebinding_with_section 函数中做了以下工作，代码如下。

①获取懒加载表或非懒加载表在间接跳转表中的位置。

②遍历 section，找到对应位置在间接跳转表中所对应的符号表的下标。

③根据符号表的下标，获取符号表中对应的符号字符串。

④遍历 rebindings 链表，判断符号是否匹配或是否已被替换，如果匹配且没有被替换，就进行替换操作。

```
static void perform_rebinding_with_section(struct rebindings_entry *rebindings,
                                           section_t *section,
                                           intptr_t slide,
```

```
                                    nlist_t *symtab,
                                    char *strtab,
                                    uint32_t *indirect_symtab) {
  //在间接跳转符号表中的偏移
  uint32_t *indirect_symbol_indices = indirect_symtab + section->reserved1;
  //找到对应的 section
  void **indirect_symbol_bindings = (void **)((uintptr_t)slide + section->addr);
  for (uint i = 0; i < section->size / sizeof(void *); i++) {
    //从间接符号表中获取在符号表中的索引
    uint32_t symtab_index = indirect_symbol_indices[i];
    if (symtab_index == INDIRECT_SYMBOL_ABS || symtab_index == INDIRECT_SYMBOL_LOCAL ||
        symtab_index == (INDIRECT_SYMBOL_LOCAL   | INDIRECT_SYMBOL_ABS)) {
      continue;
    }
    uint32_t strtab_offset = symtab[symtab_index].n_un.n_strx;
    char *symbol_name = strtab + strtab_offset;
    printf("%s\n",symbol_name);
    if (strnlen(symbol_name, 2) < 2) {
      continue;
    }
    struct rebindings_entry *cur = rebindings;
    while (cur) {
      for (uint j = 0; j < cur->rebindings_nel; j++) {
        if (strcmp(&symbol_name[1], cur->rebindings[j].name) == 0) {
          if (cur->rebindings[j].replaced != NULL &&
              indirect_symbol_bindings[i] != cur->rebindings[j].replacement) {
            *(cur->rebindings[j].replaced) = indirect_symbol_bindings[i];
          }
          indirect_symbol_bindings[i] = cur->rebindings[j].replacement;
          goto symbol_loop;
        }
      }
      cur = cur->next;
    }
  symbol_loop:;
  }
}
```

可以在代码中打印所有符号的名字，以便在 fishhook 没有起作用时查看符号是不是真的在符号表中。另外，fishhook 只 hook 当前模块中的函数调用。

6.4.3　Cydia Substrate

Cydia Substrate 提供了针对 OC 的 runtime hook 和针对 C 或 C++ 函数的 inline hook，这里只

介绍 inline hook 的使用及原理。首先看一下 Cydia Substrate 的使用，代码如下。

```
#include <substrate.h>
#include <objc/runtime.h>

@class ViewController;

static void (*originMethodImp)(ViewController*, SEL, id);

static void newMethodImp(ViewController* self, SEL _cmd, id sender) {
    NSLog(@"-[<ViewController: %p> Hook clickMe:%@]", self, sender);
    originMethodImp(self, _cmd, sender);
}

int (*oldopen)(const char *, int, ...);

int newopen(const char *path, int oflag, ...) {
    va_list ap = {0};
    mode_t mode = 0;

    if ((oflag & O_CREAT) != 0) {
        va_start(ap, oflag);
        mode = va_arg(ap, int);
        va_end(ap);
        printf("MSHookFunction | Calling real open('%s', %d, %d)\n", path, oflag, mode);
        return oldopen(path, oflag, mode);
    } else {
        printf("MSHookFunction | Calling real open('%s', %d)\n", path, oflag);
        return oldopen(path, oflag, mode);
    }
}

static __attribute__((constructor)) void myinit() {

    Class targetClass = objc_getClass("ViewController");

    MSHookMessageEx(targetClass,@selector(clickMe:),(IMP)&newMethodImp,(IMP*)&originMet
hodImp);

    MSHookFunction(open, newopen, &oldopen);

}
```

上面的代码对 open 函数进行了 inline hook。看一下 open 函数在 hook 之前和 hook 之后所对

应的汇编代码有什么区别。先来看 hook 之前的 open 函数的汇编代码，具体如下。

```
libsystem_kernel.dylib`open:
->  0x198bc1db0 <+0>:  stp    x29, x30, [sp, #-0x10]!
    0x198bc1db4 <+4>:  mov    x29, sp
    0x198bc1db8 <+8>:  ldr    x2, [x29, #0x10]
    0x198bc1dbc <+12>: bl     0x198bc2e48               ; __open
    0x198bc1dc0 <+16>: mov    sp, x29
    0x198bc1dc4 <+20>: ldp    x29, x30, [sp], #0x10
    0x198bc1dc8 <+24>: ret
```

再来看 hook 之后的 open 函数的汇编指令有什么变化，具体如下。

```
libsystem_kernel.dylib`open:
->  0x198bc1db0 <+0>:  ldr    x16, #0x8                 ; <+8>
    0x198bc1db4 <+4>:  br     x16
    0x198bc1db8 <+8>:  .long  0x00223c50                ; unknown opcode
    0x198bc1dbc <+12>: .long  0x00000001                ; unknown opcode
    0x198bc1dc0 <+16>: mov    sp, x29
    0x198bc1dc4 <+20>: ldp    x29, x30, [sp], #0x10
    0x198bc1dc8 <+24>: ret
```

到这里，发现前面几条汇编指令已经被替换了，跳转到 0x00223c50 处执行，跟进发现 0x00223c50 就是自定义的 newopen 函数，具体如下。

```
CydiaSubstrate.dylib`newopen:
->  0x100223c50 <+0>:   sub    sp, sp, #0x60             ; =0x60
    0x100223c54 <+4>:   stp    x29, x30, [sp, #0x50]
    0x100223c58 <+8>:   add    x29, sp, #0x50            ; =0x50
    0x100223c5c <+12>:  mov    x8, #0x0
    0x100223c60 <+16>:  stur   x0, [x29, #-0x10]
    0x100223c64 <+20>:  stur   w1, [x29, #-0x14]
    0x100223c68 <+24>:  stur   x8, [x29, #-0x20]
    0x100223c6c <+28>:  sturh  wzr, [x29, #-0x22]
    0x100223c70 <+32>:  ldur   w1, [x29, #-0x14]
    ......
```

那么，在自定义的函数里调用原实现时，其中的代码是怎样的呢？跟进最后一个原始调用（newopen 中的最后一个 blr），这里的代码就是原函数的头部，因此，在这里调用回原函数，具体如下。

```
->  0x10024c000: stp    x29, x30, [sp, #-0x10]!
```

```
    0x10024c004: mov    x29, sp
    0x10024c008: ldr    x2, [x29, #0x10]
    0x10024c00c: ldr    x16, #0x18
    0x10024c010: blr    x16                 #通过调试得知此处跳转到 0x0000000198bc2e48
libsystem_kernel.dylib`__open
    0x10024c014: ldr    x16, #0x8
    0x10024c018: br     x16                 #通过调试得知此处跳转到 0x0000000198bc1dc0
libsystem_kernel.dylib`open + 16
    0x10024c01c: ldrsw  x0, 0x1001c43d4
```

整个流程如下。

①复制目标函数头部的汇编代码，直接跳转到自定义函数。

②在自定义函数中调用原函数，跳转到原函数复制的代码部分。

③跳转到原函数剩余的部分。

6.4.4 Swift hook

Swift 是苹果开发的新语言。苹果原本打算用它取代 OC，但是从目前的情况来看，老的 OC 项目还是使用 OC 开发的，只有一些新的项目才会尝试使用 Swift 进行开发，也就是说，再过一段时间，我们就需要学一门新的语言了。

Swift 和 C++ 类似，在编译期间就会确定调用方法的地址，而不需要通过 OC 的 runtime 进行动态查找，所以效率自然提高了许多。当然，Swift 也兼容 OC 的动态特性。尽管分析 Swift 开发的应用要比分析 OC 开发的应用难度大一点，但也是可以实现的。可以使用笔者修改的 class-dump 获取 OC 和 Swift 混淆的头文件信息（看不到 swift 方法），使用 class-dump-swift 获取纯 Swift 开发的头文件信息，也可以直接使用 Hopper 和 IDA 查看。在分析时可能会看到类似如下的符号。

```
@interface _TtC9SwiftDemo14ViewController : UIViewController

__T09SwiftDemo14ViewControllerC12CustomMethodS2i6number_tF
```

这是 Swift 中的 Name Mangling（名字重整），是为了解决程序实体的名字必须唯一的问题而将函数类型、函数名称、参数类型、返回类型等编码到名字中的一种方法，用于从编译器中向链接器传递更多的语义信息。也可以通过 nm 命令读取可执行文件的符号表，能够看到如下信息。

```
➜  SwiftDemo.app git:(master) X nm SwiftDemo
.......
```

```
0000000100006610 T __T09SwiftDemo14ViewControllerC11viewDidLoadyyF
0000000100006d8 t __T09SwiftDemo14ViewControllerC11viewDidLoadyyFTo
0000000100006a08 T __T09SwiftDemo14ViewControllerC12CustomMethodS2i6number_tF
0000000100006718 T __T09SwiftDemo14ViewControllerC23didReceiveMemoryWarningyyF
0000000100006784 t __T09SwiftDemo14ViewControllerC23didReceiveMemoryWarningyyFTo
00000001000067c4 T __T09SwiftDemo14ViewControllerC7onClickySo8UIButtonCF
00000001000069a8 t __T09SwiftDemo14ViewControllerC7onClickySo8UIButtonCFTo
0000000100006af0 T __T09SwiftDemo14ViewControllerCACSSSg7nibName_So6BundleCSg6bundletcfC
0000000100006b54 T __T09SwiftDemo14ViewControllerCACSSSg7nibName_So6BundleCSg6bundletcfc
0000000100006cb0 t
__T09SwiftDemo14ViewControllerCACSSSg7nibName_So6BundleCSg6bundletcfcTo
0000000100006d7c T __T09SwiftDemo14ViewControllerCACSgSo7NSCoderC5coder_tcfC
0000000100006dbc T __T09SwiftDemo14ViewControllerCACSgSo7NSCoderC5coder_tcfc
0000000100006e8c t __T09SwiftDemo14ViewControllerCACSgSo7NSCoderC5coder_tcfcTo
......
```

Swift 也提供了工具对其进行解析，具体如下。

```
➜  SwiftDemo.app git:(master) ✗ nm SwiftDemo | xcrun swift-demangle
......
000000010000d558 S _type metadata for SwiftDemo.AppDelegate
0000000100007b28 T _SwiftDemo.AppDelegate.__deallocating_deinit
0000000100007bd4 t _@objc SwiftDemo.AppDelegate.__ivar_destroyer
0000000100006610 T _SwiftDemo.ViewController.viewDidLoad() -> ()
00000001000066d8 t _@objc SwiftDemo.ViewController.viewDidLoad() -> ()
0000000100006a08 T _SwiftDemo.ViewController.CustomMethod(number: Swift.Int) -> Swift.Int
0000000100006718 T _SwiftDemo.ViewController.didReceiveMemoryWarning() -> ()
0000000100006784 t _@objc SwiftDemo.ViewController.didReceiveMemoryWarning() -> ()
00000001000067c4 T _SwiftDemo.ViewController.onClick(__ObjC.UIButton) -> ()
00000001000069a8 t _@objc SwiftDemo.ViewController.onClick(__ObjC.UIButton) -> ()
......

➜  SwiftDemo.app git:(master) ✗ xcrun swift-demangle -expand
__T09SwiftDemo14ViewControllerC12CustomMethodS2i6number_tF
Demangling for _T09SwiftDemo14ViewControllerC12CustomMethodS2i6number_tF
kind=Global
  kind=Function
    kind=Class
      kind=Module, text="SwiftDemo"
      kind=Identifier, text="ViewController"
    kind=Identifier, text="CustomMethod"
    kind=Type
      kind=FunctionType
        kind=ArgumentTuple
```

```
          kind=Type
            kind=Tuple
              kind=TupleElement
                kind=TupleElementName, text="number"
                kind=Type
                  kind=Structure
                    kind=Module, text="Swift"
                    kind=Identifier, text="Int"
        kind=ReturnType
          kind=Type
            kind=Structure
              kind=Module, text="Swift"
              kind=Identifier, text="Int"
_T09SwiftDemo14ViewControllerC12CustomMethodS2i6number_tF --->
SwiftDemo.ViewController.CustomMethod(number: Swift.Int) -> Swift.Int
```

对这种情况，可以直接对该方法所在的地址使用 MSHookFunction 进行 hook。例如，想要 hook Swift 方法 CustomMethod：

```
class ViewController: UIViewController {

    @IBAction func onClick(_ sender: UIButton) {
        let result = self.CustomMethod(number: 10)
        print("origin: \(result)")
    }

    func CustomMethod(number: Int) -> Int{
        return number + 10
    }
}
```

可以先获取符号的地址，然后直接 hook。相关代码如下。

```
static int (*origin_custom_method)(int,id) = NULL;

int new_custom_method(int number, id _self)
{
     NSLog(@"Hooked!!!");
     number = 20;
    return origin_custom_method(number, _self);
}

__attribute__((constructor))
int main(void){
```

```
    NSLog(@"Load!!!");

MSHookFunction(MSFindSymbol(NULL,"__T09SwiftDemo14ViewControllerC12CustomMethodS2i6numb
er_tF"),
        (void*)new_custom_method,
        (void**)&origin_custom_method);

   return 0;
}
```

在这里需要注意，Swift 中方法的参数和 OC 中方法的参数不一样，self 作为最后一个参数传递，并没有 selector，具体如下。

```
type method(id arg1, id arg2, ..., id self)
```

6.5　动态库

库是一种共享程序代码的方式，从本质上来说就是一种可执行代码的二进制形式。在正向开发中，常常会将程序所用的某部分功能编译成库的形式，只暴露出头文件供开发者调用。库分为静态库和动态库，下面介绍一下静态库和动态库的区别。

静态库的特点如下。

- 静态库的存在形式有 .a 和 .framework。
- 静态库由一个或多个 object 文件组成，可以将一个静态库拆解成多个 object 文件（ar -x）。
- 静态库链接时会直接链接到目标文件，并作为它的一部分存在。

动态库的特点如下。

- 动态库的存在形式有 .dylib、.framework 及链接符号 .tbd。
- 动态库的格式和普通二进制文件没有区别。
- 动态库的好处是可以只保留一份文件和内存空间，从而能够被多个进程使用，例如系统的动态库。
- 在修改动态库的功能时，只需要替换动态库，不需要修改依赖的主文件。
- 可以减小可执行文件的体积，不需要链接到目标文件。

对于静态库，在本节就不深入讲解了。

6.5.1 编译和注入

动态库的编译在前面的章节已经提到过，通过如下命令即可编译。

```
xcrun --sdk iphoneos clang -dynamiclib -arch arm64 -framework Foundation InsertDylib.mm -o
InsertDylib.dylib -compatibility_version 1 -current_version 1 -install_name
@executable_path/InsertDylib.dylib
```

当然，也可以通过 Xcode 新建一个 Cocoa Touch Framework 类型的项目，然后在 Build Settings 界面设置“Mach-O Type”为“Dynamic Library”。

动态库注入的几种方式在前面都介绍过了，这里总结一下。

- 通过设置环境变量 DYLD_INSERT_LIBRARIES 指定要注入的动态库路径即可注入。
- 通过 Cydia Substrate 提供的注入，配置 plist 文件和动态库，并放入 /Library/Mobile Substrate/DynamicLibraries/ 目录，其实也是先通过 DYLD_INSERT_LIBRARIES 将自己注入，然后遍历 DynamicLibraries 下面的 plist 文件，再将符合规则的动态库通过 dlopen 打开。
- 通过增加 Load Command 的 LC_LOAD_DYLIB 或 LC_LOAD_WEAK_DYLIB，指定动态库的路径来实现注入。现成的注入工具有 optool、insert_dylib 等。

6.5.2 导出和隐藏符号

在编译动态库时，有些符号是不需要导出的，所以可以通过以下 3 种方式来控制哪些符号需要导出，哪些符号不需要导出。

- 增加 static 参数。
- 指定 export_list 参数，导出符号里的列表。
- 增加编译参数 -fvisibility=hidden，设置默认都不导出。

下面分别用实际的例子进行演示。首先，编辑如下文件。

```
Person.h
char* name(void);
void set_name(char* name);

Person.mm
#include "Person.h"
#include <string.h>

char _person_name[30] = {'\0'};
```

```
void _set_name(char* name) {
   strcpy(_person_name, name);
}

char* name(void) {
    return _person_name;
}

void set_name(char* name) {
    if (name == NULL) {
        _set_name((char*)"");
    }
    else {
        _set_name(name);
    }
}
```

编译生成动态库之后，使用 nm -gm target.dylib 命令查看导出的符号信息，_set_name 和 _person_name 都导出了，具体如下。

```
➜  ExportSymbol git:(master) X nm -gm target.dylib
0000000000007f30 (__TEXT,__text) external __Z4namev
0000000000007f3c (__TEXT,__text) external __Z8set_namePc
0000000000007f00 (__TEXT,__text) external __Z9_set_namePc
0000000000008020 (__DATA,__common) external __person_name
                 (undefined) external _strcpy (from libSystem)
                 (undefined) external dyld_stub_binder (from libSystem)
```

然后，在其前面加上 static 代码，具体如下。

```
static char _person_name[30] = {'\0'};

static void _set_name(char* name) {
   strcpy(_person_name, name);
}
```

编译之后，查看导出的符号信息，发现已经没有导出的符号信息了，具体如下。

```
➜  ExportSymbol git:(master) X nm -gm target.dylib
0000000000007f04 (__TEXT,__text) external __Z4namev
0000000000007f10 (__TEXT,__text) external __Z8set_namePc
                 (undefined) external _strcpy (from libSystem)
                 (undefined) external dyld_stub_binder (from libSystem)
```

下面来看另一种方式。在编译参数中加入 -exported_symbols_list export_list，内容和编译参数如下。查看结果，也只导出了指定的符号。

```
#export_list 的内容:
__Z4namev
__Z8set_namePc

xcrun --sdk iphoneos clang -dynamiclib -arch arm64 -framework Foundation Person.mm -o
target.dylib -compatibility_version 1 -current_version 1  -exported_symbols_list
export_list
```

如果不想写成这种 C++ 符号，在头文件中以 C 的方式导出即可，具体如下。

```
extern "C" char* name(void);
extern "C" void set_name(char* name);

export_list 内容:
_name
_set_name
```

在编译参数中指定 -fvisibility=hidden，然后对指定符号增加 visibility("default") 来导出符号，也是正常的，具体如下。

```
#include "Person.h"
#include <string.h>

#define EXPORT __attribute__((visibility("default")))

static char _person_name[30] = {'\0'};

void _set_name(char* name) {
   strcpy(_person_name, name);
}

EXPORT
char* name(void) {
    return _person_name;
}

EXPORT
void set_name(char* name) {
    if (name == NULL) {
        _set_name((char*)"");
    }
```

```
    else {
        _set_name(name);
    }
}
```

```
xcrun --sdk iphoneos clang -dynamiclib -arch arm64 -framework Foundation Person.mm -o
target.dylib -compatibility_version 1 -current_version 1 -fvisibility=hidden
```

6.5.3 C++ 和 OC 动态库

在编译已经生成的动态库时，可以通过引入头文件的方式调用，也可以通过 dlopen 动态加载的方式调用。在这里讲解如何通过 dlopen 动态的方式调用 C++ 和 OC 编写的动态库。先编写一个 C++ 动态库，具体如下。

```
Person.h
class Person {
    private:
        char _person_name[30];
    public:
        Person();
        virtual void set_name(char person_name[]);
        virtual char* name();
};

extern "C" Person *NewPerson(void);
typedef Person *Person_creator(void);

extern "C" void DeletePerson(Person *person);
typedef void Person_disposer(Person *);

Person.mm
#include <iostream>
#import <Foundation/Foundation.h>
#include "Person.h"

#define EXPORT __attribute__((visibility("default")))

EXPORT
Person::Person() {
    char default_name[] = "<no value>";
    this->set_name(default_name);
}
```

```
EXPORT
Person* NewPerson(void) {
    return new Person;
}

EXPORT
void DeletePerson(Person* person) {
    delete person;
}

void Person::set_name(char name[]) {
    strcpy(_person_name, name);
}

EXPORT
char* Person::name(void) {
    return _person_name;
}

__attribute__((constructor))
static void initializer1() {
     NSLog(@"我被加载了。。。。。");
}
```

编译生成动态库，并将其放入 /Library/MobileSubstrate/DynamicLibraries/，然后在主 App 里面通过如下代码主动加载调用。

```
void *lib_handle = dlopen("/Library/MobileSubstrate/DynamicLibraries/target.dylib",
RTLD_LOCAL);
if(!lib_handle){
    NSLog(@"Unable to open library: %s",dlerror());
    return 0;
}

Person_creator* NewPerson = (Person_creator*)dlsym(lib_handle, "NewPerson");
if(!NewPerson){
    NSLog(@"Unable to find NewPerson method:%s",dlerror());
    return 0;
}

Person_disposer* DeletePerson = (Person_disposer*)dlsym(lib_handle, "DeletePerson");
if(!DeletePerson){
    NSLog(@"Unable to find DeletePerson method:%s",dlerror());
    return 0;
```

```
}

Person* person = (Person*)NewPerson();

NSLog(@"person->name() = %s",person->name());

char new_name[] = "AloneMonkey";

person->set_name(new_name);

NSLog(@"person->name() = %s",person->name());

DeletePerson(person);

if(dlclose(lib_handle) != 0){
    NSLog(@"Unable to close library: %s",dlerror());
}
```

因为这里的符号都是通过 C 导出的，所以直接写名字即可。如果是通过 C++ 导出的，就要指定 C++ 的符号，具体如下。

```
Person_creator* NewPerson = (Person_creator*)dlsym(lib_handle, "_Z9NewPersonv");
Person_disposer* DeletePerson = (Person_disposer*)dlsym(lib_handle,
"_Z12DeletePersonP6Person");
```

再来看一下用 OC 编写的动态库动态加载调用的方式，具体如下。

```
Person.h
#import <Foundation/Foundation.h>

@interface Person : NSObject

-(void)setName:(NSString*)name;
-(NSString*)name;

@end

Person.m
#import "Person.h"

@implementation Person{
    NSString* _person_name;
}
```

```
- (id)init {
    if (self = [super init]) {
        _person_name = @"";
    }
    return self;
}

- (void)setName:(NSString*)name {
    _person_name = name;
}

- (NSString*)name {
    return _person_name;
}
@end
```

在 App 里面通过如下方式进行动态调用。

```
void* lib_handle = dlopen("/Library/MobileSubstrate/DynamicLibraries/target.dylib",
RTLD_LOCAL);
if (!lib_handle) {
    NSLog(@"[%s] main: Unable to open library: %s\n",
          __FILE__, dlerror());
    exit(EXIT_FAILURE);
}

Class PersonClass = objc_getClass("Person");
if (!PersonClass) {
    NSLog(@"[%s] main: Unable to get Person class", __FILE__);
    exit(EXIT_FAILURE);
}

NSLog(@"[%s] main: Instantiating PersonClass", __FILE__);
Person* person = [PersonClass new];

[person setName:@"Alone Monkey"];
NSLog(@"[%s] main: [person name] = %@", __FILE__, [person name]);

if (dlclose(lib_handle) != 0) {
    NSLog(@"[%s] Unable to close library: %s\n",
          __FILE__, dlerror());
    exit(EXIT_FAILURE);
}
```

实际上，OC 的动态库和正常使用的动态库没有区别，需要我们注意的是 C++ 的动态库符号。先看符号表里面导出的符号名字是什么。如果是 C 导出，直接写名字即可。如果是 C++ 导出，需要通过真正导出的符号获取。这一点和获取 Swift 类时一样。

6.5.4　其他常见问题

下面对使用动态库时的常见问题进行讲解。

1. 动态库加载顺序问题

在注入动态库时会涉及谁先被加载、谁后被加载的问题。例如，在 Load Command 中的动态库，谁先被加载；在 CydiaSubstrate 目录中的动态库，谁先被加载。

假如通过 optool 向主程序依次注入 3 个动态库 A、B、C，其中 A 动态库中的 Load Command 被注入了 C，也就是 A 依赖 C，那么 A、B、C 中的 constructor 函数的调用顺序是 C→A→B，因为 A 依赖 C，所以会先加载其依赖的 C，再加载 A，最后加载 B，也就是被依赖的动态库会先加载，其他动态库按 Load Command 的顺序加载。如果是 /Library/MobileSubstrate/DynamicLibraries/ 下面的动态库，因为是通过 MobileLoader 遍历目录下面的所有文件，然后根据 plist 规则加载的，所以谁在文件遍历序列的最前面（一般按字母顺序排列），谁就先被加载。读者可以自己通过注入来实践。

2. 如何将断点设置在动态库的入口

在分析一个动态库时，通常需要从入口开始分析该动态库 hook 了哪些函数、进行了哪些操作。要使程序中断在动态库的入口，需要满足如下两点。

- 该动态库模块已经加载到内存中。
- 该动态库的 constructor 函数都没有执行。

要想实现这两点，肯定还得从 dyld 入手。一种方案是，在通过 LLDB 命令行进行调试时自动中断在 _dyld_start 处，因为此时 dyld 已经加载，所以在 dyld 里面设置一个在所有库中加载并且在 constructor 函数执行之前执行的断点即可，例如 initializeMainExecutable 函数。但是，如果是通过 Xcode 调试第三方应用的，这种方法就行不通了，程序会直接跑飞。这时需要编辑 ~/.lldbinit 文件，在其中增加如下内容。

```
settings set target.process.stop-on-sharedlibrary-events 1
```

运行 Xcode，程序中断在如图 6-26 所示的位置，对应的是 dyld 源代码中 ImageLoader::link

函数调用 this->recursiveLoadLibraries 递归加载动态库之后 context.notifyBatch 的位置。

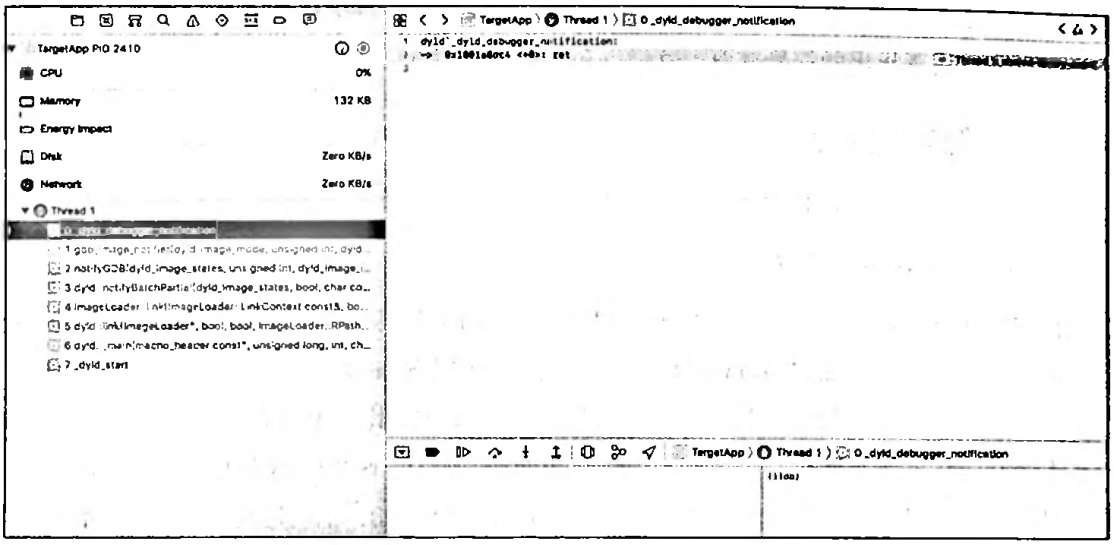


图 6-26 运行后 Xcode 自动断在 dyld 中

此时，可以通过 image list 查看想要分析的动态库是否已经加载了。如果没有加载，就让 Xcode 继续运行（因为通过 dlopen 加载的动态库也会触发断点）。找到目标模块之后，得到模块加载的基地址，然后从 MachOView 中找到动态库的 constructor 函数列表，如图 6-27 所示。

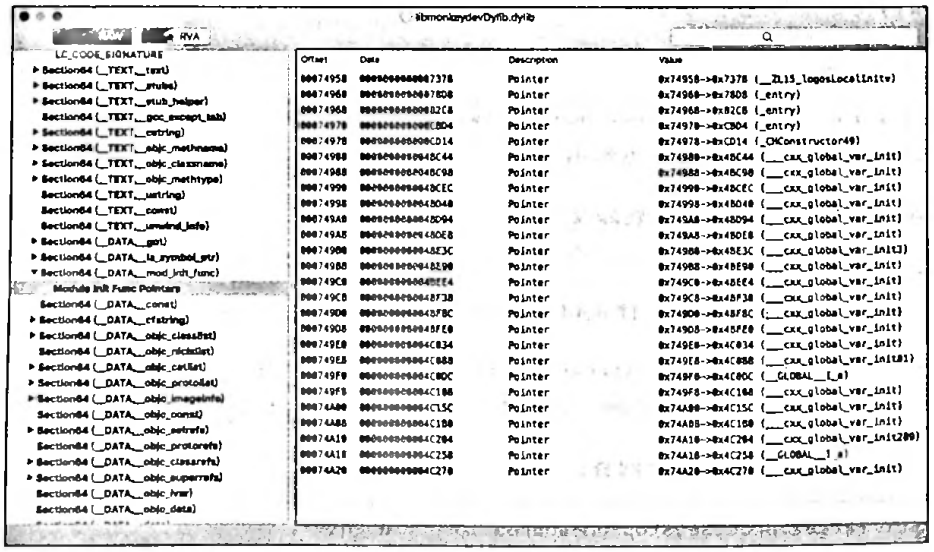


图 6-27 使用 MachOView 查看__mod_init_func

假如我们要将断点设在 0xCD14 处，并获取模块加载基地址（此处为 0x0000000010017c000），可以直接根据地址偏移下断点，具体如下。

```
(lldb) im li -o -f libmonkeydevDylib.dylib
[  0] 0x0000000010017c000
/Users/monkey/Desktop/monkeydev/Build/Products/Debug-iphoneos/monkeydev.app/Frameworks/
libmonkeydevDylib.dylib
(lldb) br s -a '0x0000000010017c000 + 0xCD14'
Breakpoint 1: where = libmonkeydevDylib.dylib`CHConstructor49 at monkeydevDylib.m:49,
address = 0x0000000100188d14
```

这样，程序就能中断在 CHConstructor49 入口函数处了。

3. LC_ID_DYLIB 和 LC_LOAD_WEAK_DYLIB

- LC_ID_DYLIB：在其他模块依赖当前动态库时显示在其他模块 LC_LOAD_DYLIB 中的路径，也就是告诉其他使用当前模块的库如何找到本模块。
- LC_LOAD_WEAK_DYLIB：LC_LOAD_DYLIB 在目标库加载失败时会直接报错中断程序，而 LC_LOAD_WEAK_DYLIB 即使找不到也不会中断程序。optool 和 insert_dylib 注入时都有参数可以指定。

如果需要修改 LC_ID_DYLIB 或者 LC_LOAD_DYLIB，可以使用 install_name_tool，具体如下。

```
install_name_tool -id xxxx inputfile
install_name_tool -change old new inputfile
```

4. @executable_path

在使用 otool -L 命令查看可执行文件中加载的动态库时，经常会看到@rpath、@loader_path、@executable_path 这样的变量，下面分别解释一下这些变量的意义。

- @executable_path：表示可执行程序所在的目录，一般是 xxx.app 目录。
- @loader_path：表示每一个被加载的二进制文件的目录。例如，xxxx.plugin/aaa/abc 依赖 xxxx.plugin/bbb/ccc.dylib，那么依赖的路径可以写成 @loader_path/../bbb。这样，不管 xxxx.plugin 放在哪里，都能找到 ccc.dylib。
- @rpath：这个变量是在 Xcode build 里面设置的。如果将动态库的 Dynamic Library Install Name 设置为 @rpath/xxx/xxx，就可以在使用的工程中设置一个或多个 Runpath Search Paths 来指定搜索路径。在运行时，会将 @rpath 分别替换为 Runpath Search Paths 中指定的路径来查找动态库。

第 7 章　实战演练

在前面的章节中讲了很多工具的使用方法和原理，以及语言的特性、文件结构等。不要因为觉得这些内容没有用而直接跳到本章来阅读与实际操作相关的内容，要知道，所有的实践都是源于理论的，前面的知识积累是很重要的。建议读者在阅读本章时，以一种研究技巧和方法的心态来学习，因为案例是固定的，方法才是通用的。在本章中会通过不同的方法和分析手段进行讲解，希望读者能够从中获得解决问题的能力和思路。本章的讲解将从越狱设备常规分析和非越狱设备 Xcode 调试分析两个角度进行。

7.1　越狱设备分析

本节主要讲解如何在越狱设备上分析 WhatsApp（v2.17.52），如何给文本消息增加收藏按钮并在设置界面增加一个选项来显示收藏的消息，越狱设备的常规分析方法和分析的一般流程，以及如何定位目标函数的思路，效果如图 7-1 和图 7-2 所示。首先，我们要做好准备工作。

7.1.1　分析准备

在分析越狱设备前，需要完成以下准备工作。

1. 通过解密获取应用包

安装完成后，使用修改的 dumpdecrypted（https://github.com/AloneMonkey/dumpdecrypted）进行解密，获取解密的 WhatsApp.decrypted 文件，并将其复制到 Mac 中，代码如下。

```
➜  dumpdecrypted git:(master) X scp -P 2222
root@localhost:/var/mobile/Containers/Data/Application/7B9D745E-A2A5-417E-AA6E-EEF06DF9
879C/Documents/WhatsApp.decrypted ./
WhatsApp.decrypted                            100%   36MB  14.0MB/s   00:02
```

将解密的架构提取出来并赋予可执行权限，代码如下。

```
➜  dumpdecrypted git:(master) X lipo WhatsApp.decrypted -thin arm64 -output WhatsApp
➜  dumpdecrypted git:(master) X chmod +x WhatsApp
```

图 7-1　给 WhatsApp 的文本消息增加收藏按钮

图 7-2　给 WhatsApp 的设置界面增加查看收藏消息的选项

2. class-dump

使用 class-dump 为解密的文件生成头文件，加上 -A 可以显示方法在文件中的实现地址，代码如下。

```
➜  files git:(master) X ./class-dump WhatsApp -A -H -o ./Headers
2017-09-17 22:36:33.856 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _networkTransferSuspended
2017-09-17 22:36:33.857 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _currentPhase
2017-09-17 22:36:33.857 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _progressPercentage
2017-09-17 22:36:33.911 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _succeeded
2017-09-17 22:36:33.916 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _currentPhase
2017-09-17 22:36:33.916 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _progressPercentage
2017-09-17 22:36:33.971 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _stateULong
```

```
2017-09-17 22:36:33.971 class-dump[98379:16279746] Warning: Parsing instance variable type
failed, _networkStatusULong
2017-09-17 22:36:33.984 class-dump[98379:16279746] Warning: Parsing method types failed,
getRendererWithCommandQueue:atP0:p1:pMid:transformMatrix:renderer:index:
......
```

3. 符号还原

为方便在后续调试中对堆栈进行分析，在这里把二进制文件的符号还原，这样在后续调试中就能直接下符号断点和查看堆栈的符号信息了，代码如下。

```
➜  files git:(master) X restore-symbol WhatsApp -o WhatsApp_with_symbol
=========== Start =============
Scan OC method in mach-o-file.
2017-09-17 22:38:40.431 restore-symbol[98545:16282236] Warning: Parsing instance variable
type failed, _networkTransferSuspended
2017-09-17 22:38:40.431 restore-symbol[98545:16282236] Warning: Parsing instance variable
type failed, _currentPhase
2017-09-17 22:38:40.431 restore-symbol[98545:16282236] Warning: Parsing instance variable
type failed, _progressPercentage
2017-09-17 22:38:40.482 restore-symbol[98545:16282236] Warning: Parsing instance variable
type failed, _succeeded
2017-09-17 22:38:40.489 restore-symbol[98545:16282236] Warning: Parsing instance variable
type failed, _currentPhase
......
Scan OC method finish.
=========== Finish ============
```

使用还原了符号的文件替换 WhatsApp.app 下面的目标文件，然后使用 mobiledevice 重新将 WhatsApp.app 安装到越狱设备中。

7.1.2 开始分析

要分析一个程序功能的实现，通常要从界面入手，具体如下。

- 在分析点击事件的响应时，可以通过点击的按钮入手来获取它的 action 响应方法。
- 对界面上没有明显事件的，一般通过获取当前的 ViewController 来跟踪。
- 从系统特有的界面结构入手，例如 UITableView，通过其 delegate 方法来分析。

因为我们的目的是在用户长按发送的文本消息后在弹出的菜单中增加一个选项，所以，在这里需要从界面入手，获取点击的控件的类型。使用 Reveal 查看，对应界面文本消息的控件为

WAMessageContainerView，如图 7-3 所示。如果我们要获取当前控件的长按事件，可以直接查看 WAMessageContainer View 头文件中有没有相关信息，然后在头文件中找到以下函数。

```
- (void)handleLongPressAtPoint:(struct CGPoint)arg1;  // IMP=0x00000001008e16dc
- (void)handleDoubleTapAtPoint:(struct CGPoint)arg1;  // IMP=0x00000001008e167c
- (void)handleSingleTapAtPoint:(struct CGPoint)arg1;  // IMP=0x00000001008e161c
```

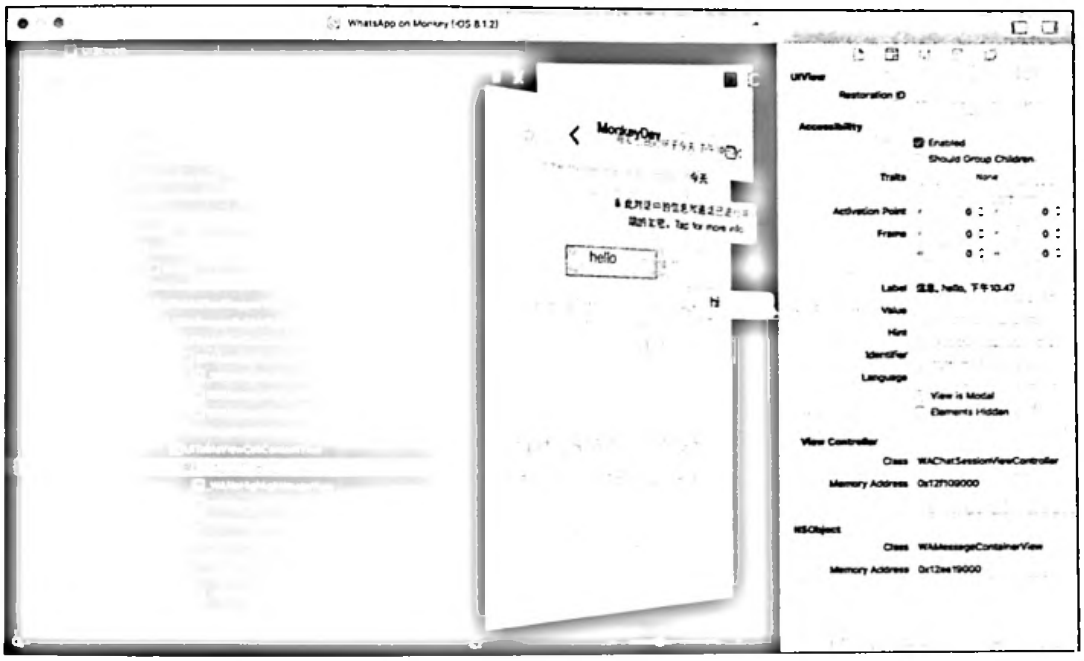


图 7-3　使用 Reveal 查看 WhatsApp 的消息控件

在这里，我们猜测 handleLongPressAtPoint 函数就是用于处理长按事件的函数。通过 LLDB 下断点进行测试，因为已经恢复了符号信息，所以可以使用如下方式直接下断点。

```
(lldb) b -[WAMessageContainerView handleLongPressAtPoint:]
Breakpoint 1: where = WhatsApp`-[WAMessageContainerView handleLongPressAtPoint:], address
= 0x00000001009c96dc
```

按“C”键继续运行，之后再长按文本消息，就会触发断点，这说明找到的函数没有错。

使用 Hopper 分析该函数内部的实现和调用过程。将文件拖入 Hopper，分析 AArch64 架构的部分。搜索关键字“handleLongPressAtPoint”，找到函数 [WAMessageContainerView handleLongPressAtPoint:]，显示的伪代码如下。

```
void -[WAMessageContainerView handleLongPressAtPoint:](void * self, void * _cmd, struct
CGPoint arg2) {
    memcpy(&r2, &arg2, 0x8);
    *(r31 + 0xffffffffffffffd0) = d9;
    *(0xfffffffffffffffe0 + r31) = d8;
    *(r31 + 0xfffffffffffffffe0) = r20;
    *(r31 + 0xfffffffffffffff0) = r19;
    *(r31 + 0xfffffffffffffff0) = r29;
    *(r31 + 0x0) = r30;
    r0 = [self delegate];
    r0 = [r0 retain];
    [r0 messageContainerView:self didReceiveLongPressAtPoint:r3];
    [r0 release];
    return;
}
```

这里调用了 WAMessageContainerView 对象 delegate 的 messageContainerView:didReceiveLongPressAtPoint: 方法。打印 delegate 如下。

```
(lldb) po [$x0 delegate]
<WAMessageBubbleTableViewCell: 0x1545e3740; baseClass = UITableViewCell; frame = (0 86; 320
45); opaque = NO; autoresize = W; gestureRecognizers = <NSArray: 0x170659800>; layer = <CALayer:
0x1706395a0>>
```

该方法的伪代码实现如下。

```
void -[WAMessageBubbleTableViewCell
messageContainerView:didReceiveLongPressAtPoint:](void * self, void * _cmd, void * arg2,
struct CGPoint arg3) {
    memcpy(&r3, &arg3, 0x8);
    r2 = arg2;
    *(r31 + 0xfffffffffffffffb0) = d9;
    *(0xfffffffffffffffc0 + r31) = d8;
    r31 = r31 + 0xfffffffffffffffb0;
    *(r31 + 0x10) = r24;
    *(0x20 + r31) = r23;
    *(r31 + 0x20) = r22;
    *(0x30 + r31) = r21;
    *(r31 + 0x30) = r20;
    *(0x40 + r31) = r19;
    *(r31 + 0x40) = r29;
    *(0x50 + r31) = r30;
    v8 = v1;
```

```
    v9 = v0;
    r19 = self;
    r0 = [UIMenuController sharedMenuController];
    r29 = r31 + 0x40;
    r0 = [r0 retain];
    r22 = [r0 isMenuVisible];
    [r0 release];
    if (r22 == 0x0) goto loc_100453f00;

loc_100453ecc:
    r0 = [UIMenuController sharedMenuController];
    r0 = [r0 retain];
    [r0 setMenuVisible:zero_extend_64(0x0) animated:0x1];
    r0 = r0;
    goto loc_100453f8c;

loc_100453f8c:
    [r0 release];
    return;

.l1:
    return;

loc_100453f00:
    if (([r19 isEditing] & 0x1) == 0x0) goto loc_100453f2c;
    goto .l1;

loc_100453f2c:
    r20 = [[r19 sliceViewForLongPressGestureAtPoint:r2] retain];
    r0 = [r19 delegate];
    r0 = [r0 retain];
    [r0 bubbleTableViewCell:r19 didReceiveLongPressOnSliceView:r20];
    [r0 release];
    r0 = r20;
    goto loc_100453f8c;
}
```

在 loc_100453f00 处有一个判断，对应的是汇编代码。在 0000000100453f10 处下断点，通过打印 w0 寄存器的值来判断该分支的走向。两段代码分别如下。

```
                    loc_100453f00:
0000000100453f00        adrp      x8, #0x1011ea000                    ;
&@selector(mediaDownloadProgressDidChange:), CODE XREF=-[WAMessageBubbleTableViewCell
messageContainerView:didReceiveLongPressAtPoint:]+96
```

```
0000000100453f04        ldr        x1, [x8, #0xa98]                                    ;
"isEditing",@selector(isEditing), argument "selector" for method imp___stubs__objc_msgSend
0000000100453f08        mov        x0, x19                                             ; argument
"instance" for method imp___stubs__objc_msgSend
0000000100453f0c        bl         imp___stubs__objc_msgSend
0000000100453f10        tbz        w0, 0x0, loc_100453f2c
```

```
(lldb) im li -o -f WhatsApp
[  0] 0x00000000000e8000
/private/var/mobile/Containers/Bundle/Application/8C94AFCD-7965-4BAA-B934-F3B383AF35AF/
WhatsApp.app/WhatsApp(0x00000001000e8000)
(lldb) br s -a '0x0000000100453f10 + 0x00000000000e8000'
Breakpoint 2: where = WhatsApp`-[WAMessageBubbleTableViewCell
messageContainerView:didReceiveLongPressAtPoint:] + 168, address = 0x000000010053bf10
(lldb) c
Process 12202 resuming
Process 12202 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
    frame #0: 0x000000010053bf10 WhatsApp` -[WAMessageBubbleTableViewCell
messageContainerView:didReceiveLongPressAtPoint:]  + 168
WhatsApp`-[WAMessageBubbleTableViewCell
messageContainerView:didReceiveLongPressAtPoint:]:
->  0x10053bf10 <+168>: tbz    w0, #0x0, 0x10053bf2c     ; <+196>
    0x10053bf14 <+172>: ldp    x29, x30, [sp, #0x40]
    0x10053bf18 <+176>: ldp    x20, x19, [sp, #0x30]
    0x10053bf1c <+180>: ldp    x22, x21, [sp, #0x20]
    0x10053bf20 <+184>: ldp    x24, x23, [sp, #0x10]
    0x10053bf24 <+188>: ldp    d9, d8, [sp], #0x50
    0x10053bf28 <+192>: ret
    0x10053bf2c <+196>: adrp   x8, 3498
Target 0: (WhatsApp) stopped.
(lldb) p $w0
(unsigned int) $2 = 0
(lldb)
```

所以，该分支会来到 loc_100453f2c 处，调用 -[WAChatSessionViewController bubbleTableViewCell:didReceiveLongPressOnSliceView:] 函数，而后者最终会调用 -[WAChatSessionViewController showMenuControllerFromSliceView:inCell:] 函数。由于不知道这里看到的函数名称是哪个类的，可以直接在 Hopper 左侧栏中搜索这个函数名。如果能找到唯一的类，那就是这个类，否则就需要通过 LLDB 动态调试来确定。

showMenuControllerFromSliceView:inCell:的伪代码如下。

```
void -[WAChatSessionViewController showMenuControllerFromSliceView:inCell:](void * self,
void * _cmd, void * arg2, void * arg3) {
    *(r31 + 0xfffffffffffffc0) = r24;
    *(0xfffffffffffffd0 + r31) = r23;
    *(r31 + 0xfffffffffffffd0) = r22;
    *(r31 + 0xfffffffffffffe0) = r21;
    *(r31 + 0xfffffffffffffe0) = r20;
    *(r31 + 0xffffffffffffff0) = r19;
    *(r31 + 0xffffffffffffff0) = r29;
    *(r31 + 0x0) = r30;
    r19 = [arg2 retain];
    r24 = [arg3 retain];
    r0 = [arg3 cellData];
    r0 = [r0 retain];
    r9 = sign_extend_64(*(int32_t *)ivar_offset(_cellDataForMenuController));
    *(self + r9) = r0;
    [*(self + r9) release];
    r0 = [arg3 currentPasteboardContentType];
    r23 = sign_extend_64(*(int32_t *)ivar_offset(_currentPasteboardContentType));
    *(self + r23) = r0;
    *(self + r23) = [arg2 isKindOfClass:[WAMessageAttributedTextSliceView class]];
    [arg3 showMenuControllerFromSliceView:r19];
    [r24 release];
    [r19 release];
    return;
}
```

调用 -[WAMessageBubbleTableViewCell showMenuControllerFromSliceView:] 函数，看看该函数的伪代码（这里略去了不相关的伪代码），具体如下。

```
void -[WAMessageBubbleTableViewCell showMenuControllerFromSliceView:](void * self, void *
_cmd, void * arg2) {
    r31 = r31 - 0x100;
    ......
    r22 = self;
    r19 = [arg2 retain];
    r22->_currentPasteboardContentType = [r19
isKindOfClass:[WAMessageAttributedTextSliceView class]];
    r20 = [[UIMenuController sharedMenuController] retain];
    [r22 targetRectForMenuController];
    r0 = [r22 referenceViewForMenuController];
    r29 = r31 + 0xf0;
    r21 = [r0 retain];
```

```
    [r20 setTargetRect:r21 inView:r3];
    [r21 release];
    [r20 setMenuVisible:0x1 animated:0x1];
    if ([r20 isMenuVisible] != 0x0) {
            ......
    }
    [r20 release];
    [r19 release];
    return;
}
```

此处通过 [UIMenuController sharedMenuController] 获取 UIMenuController 对象，然后分别调用 setTargetRect:inView: 设置位置，再通过 setMenuVisible:animated: 显示将其出来。分析到这里我们发现，如果调用系统的 UIMenuController 来显示菜单，自定义的菜单需要通过设置 menuItems 来设置显示的标题和对应的 action。但是，笔者在以上代码中没有找到用于设置 menuItems 的代码，因此需要通过动态调试来确定 menuItems 里面是否有值。找到 sharedMenu Controller 所对应的汇编代码，具体如下。

```
0000000100454628        adrp        x8, #0x101219000
0000000100454630        ldrsw       x8, [x8, #0x4c0]                    ;
objc_ivar_offset_WAMessageBubbleTableViewCell__currentPasteboardContentType
0000000100454634        str         x0, [x22, x8]
0000000100454638        adrp        x8, #0x10120e000                    ;
&@selector(getSInt64)
000000010045463c        ldr         x0, [x8, #0xa20]                    ;
objc_cls_ref_UIMenuController,_OBJC_CLASS_$_UIMenuController, argument "instance" for
method imp___stubs__objc_msgSend
0000000100454640        adrp        x8, #0x1011e8000                    ;
&@selector(containerView)
0000000100454644        ldr         x1, [x8, #0xfb0]                    ;
"sharedMenuController",@selector(sharedMenuController), argument "selector" for method
imp___stubs__objc_msgSend
0000000100454648        bl          imp___stubs__objc_msgSend
000000010045464c        mov         x29, x29
0000000100454650        bl          imp___stubs__objc_retainAutoreleasedReturnValue
```

在 0000000100454648 处下一个断点，查看返回 UIMenuController 对象的 menuItems 的值，具体如下。

```
(lldb) br s -a '0x0000000100454648 + 0x00000000000e8000'
Breakpoint 3: where = WhatsApp`-[WAMessageBubbleTableViewCell
```

```
showMenuControllerFromSliceView:] + 124, address = 0x0000000010053c648
(lldb) c
Process 12202 resuming
Process 12202 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 3.1
    frame #0: 0x0000000010053c648 WhatsApp` -[WAMessageBubbleTableViewCell
showMenuControllerFromSliceView:] + 124
WhatsApp`-[WAMessageBubbleTableViewCell showMenuControllerFromSliceView:]:
->  0x10053c648 <+124>: bl     0x100dc5ee8               ; symbol stub for: objc_msgSend
    0x10053c64c <+128>: mov    x29, x29
    0x10053c650 <+132>: bl     0x100dc5f3c               ; symbol stub for:
objc_retainAutoreleasedReturnValue
    0x10053c654 <+136>: mov    x20, x0
    0x10053c658 <+140>: adrp   x8, 3497
    0x10053c65c <+144>: ldr    x1, [x8, #0x220]
    0x10053c660 <+148>: mov    x0, x22
    0x10053c664 <+152>: bl     0x100dc5ee8               ; symbol stub for: objc_msgSend
Target 0: (WhatsApp) stopped.
(lldb) po $x0
UIMenuController

(lldb) x/s $x1
0x18ccdac48: "sharedMenuController"
(lldb) ni
Process 12202 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x0000000010053c64c WhatsApp` -[WAMessageBubbleTableViewCell
showMenuControllerFromSliceView:] + 128
WhatsApp`-[WAMessageBubbleTableViewCell showMenuControllerFromSliceView:]:
->  0x10053c64c <+128>: mov    x29, x29
    0x10053c650 <+132>: bl     0x100dc5f3c               ; symbol stub for:
objc_retainAutoreleasedReturnValue
    0x10053c654 <+136>: mov    x20, x0
    0x10053c658 <+140>: adrp   x8, 3497
    0x10053c65c <+144>: ldr    x1, [x8, #0x220]
    0x10053c660 <+148>: mov    x0, x22
    0x10053c664 <+152>: bl     0x100dc5ee8               ; symbol stub for: objc_msgSend
    0x10053c668 <+156>: mov.16b v8, v0
Target 0: (WhatsApp) stopped.
(lldb) po $x0
<UIMenuController: 0x174242d00>

(lldb) po [$x0 menuItems]
<__NSArrayI 0x17017e6c0>(
```

```
<UIMenuItem: 0x170234b60>,
<UIMenuItem: 0x170235c20>,
<UIMenuItem: 0x170234880>,
<UIMenuItem: 0x170235fc0>,
<UIMenuItem: 0x170236740>,
<UIMenuItem: 0x170235fa0>,
<UIMenuItem: 0x170236720>,
<UIMenuItem: 0x170236aa0>,
<UIMenuItem: 0x170236a20>,
<UIMenuItem: 0x170236a40>,
<UIMenuItem: 0x170236a80>,
<UIMenuItem: 0x170233420>,
<UIMenuItem: 0x170233620>,
<UIMenuItem: 0x1702334e0>,
<UIMenuItem: 0x170232a20>,
<UIMenuItem: 0x170233580>,
<UIMenuItem: 0x170232aa0>,
<UIMenuItem: 0x170233340>,
<UIMenuItem: 0x170233c80>,
<UIMenuItem: 0x1702337e0>,
<UIMenuItem: 0x170232a80>
)

(lldb) po [0x170233420 title]
资讯

(lldb) po [0x170233420 action]
0x0000000100e6b02c

(lldb) x/s 0x0000000100e6b02c
0x100e6b02c: "message_showInfo:"
(lldb)
```

通过动态调试可以看到，menuItems 里面已经有很多个 Item 了。那么，这些 Item 是在哪里设置的呢？笔者也想在这里添加一个 Item，有如下两种思路。

- 通过在 -[UIMenuController setMenuItems] 上设置断点，看看 Item 是在哪里设置的。
- 直接在 Hopper 中查找 UIMenuController 类。这个类是系统的，笔者猜测它是通过分类来修改默认的 menuItems 的。

直接搜索 UIMenuController，找到了几个方法，如图 7-4 所示。分别查看其中的伪代码实现，发现 wa_reallyInstallItems 方法中会预先设置一些 UIMenuItem 给 UIMenuController，这样就找到了第 1 个要 hook 的位置。hook 该函数，在其设置完成后添加自己的 UIMenuItem。

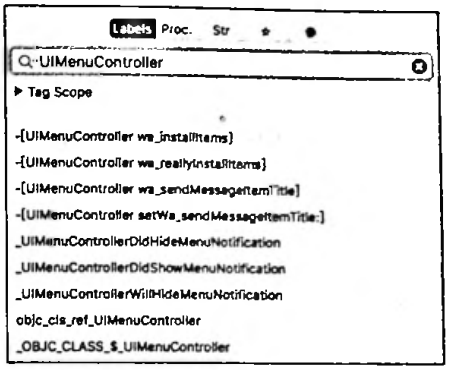


图 7-4　使用 Hopper 搜索 UIMenuController

添加自己的 UIMenuItem 之后，其中的 action 应该在哪里实现呢？这又涉及正向开发的知识了。在正向开发中，可以通过第一响应者实现 canPerformAction:withSender: 来确定当前哪些操作是被允许的、哪些操作是不被允许的。那么，哪个是第一响应者呢？现在我们不着急找到答案。直接下断点，看看程序会在哪个类里面中断，代码如下。

```
(lldb) b canPerformAction:withSender:
Breakpoint 4: 9 locations.
(lldb) c
Process 12202 resuming
Process 12202 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 4.7
    frame #0: 0x00000001009f2d54 WhatsApp` -[WAChatSessionViewController
canPerformAction:withSender:]
WhatsApp`-[WAChatSessionViewController canPerformAction:withSender:]:
->  0x1009f2d54 <+0>:  sub    sp, sp, #0x60             ; =0x60
    0x1009f2d58 <+4>:  stp    x26, x25, [sp, #0x10]
    0x1009f2d5c <+8>:  stp    x24, x23, [sp, #0x20]
    0x1009f2d60 <+12>: stp    x22, x21, [sp, #0x30]
    0x1009f2d64 <+16>: stp    x20, x19, [sp, #0x40]
    0x1009f2d68 <+20>: stp    x29, x30, [sp, #0x50]
    0x1009f2d6c <+24>: add    x29, sp, #0x50            ; =0x50
    0x1009f2d70 <+28>: mov    x21, x2
Target 0: (WhatsApp) stopped.
(lldb)
```

在动态调试过程中触发了 WAChatSessionViewController canPerformAction:withSender: 的调用，里面有对各种 action 的判断。这样，第 2 个要 hook 的位置也找到了。如果是新增的 action，就直接返回 “YES”。当然，我们可以进一步判断当前的消息类型。在 action 里面可以获取当前

消息的文本，在 canPerformAction:withSender: 的伪代码中可以看到获取当前点击的消息对象，具体如下。

```
bool -[WAChatSessionViewController canPerformAction:withSender:](void * self, void * _cmd,
void * arg2, void * arg3) {
    ......
    r24 = self;
    r0 = r24->_cellDataForMenuController;
    if (r0 == 0x0) goto loc_10090ae10;

loc_10090ad88:
    r0 = [r0 retain];
    r19 = r0;
    r0 = [r0 messages];
    r0 = [r0 retain];
    r22 = [r0 count];
    [r0 release];
    r0 = [r19 message];
    r29 = r29;
    r0 = [r0 retain];
    r20 = r0;
    r25 = [r0 isUserMessage];
```

动态调试 [r19 message]。在返回的内容中有没有我们想要的文本消息的内容呢？这里可能存在多个消息（messages），其中也可能有系统消息（isUserMessage），代码如下。

```
000000010090adc0        adrp       x8, #0x1011e6000                          ;
&@selector(setDimmed:)
000000010090adc4        ldr        x1, [x8, #0x360]                          ;
"message",@selector(message), argument "selector" for method imp___stubs__objc_msgSend
000000010090adc8        mov        x0, x19                                   ; argument
"instance" for method imp___stubs__objc_msgSend
000000010090adcc        bl         imp___stubs__objc_msgSend

(lldb) br s -a '0x000000010090adcc + 0x00000000000e8000'
Breakpoint 5: where = WhatsApp`-[WAChatSessionViewController canPerformAction:withSender:]
+ 120, address = 0x00000001009f2dcc
(lldb) c
Process 12202 resuming
Process 12202 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 5.1
    frame #0: 0x00000001009f2dcc WhatsApp` -[WAChatSessionViewController
canPerformAction:withSender:] + 120
```

```
WhatsApp`-[WAChatSessionViewController canPerformAction:withSender:]:
->  0x1009f2dcc <+120>: bl     0x100dc5ee8               ; symbol stub for: objc_msgSend
    0x1009f2dd0 <+124>: mov    x29, x29
    0x1009f2dd4 <+128>: bl     0x100dc5f3c               ; symbol stub for:
objc_retainAutoreleasedReturnValue
    0x1009f2dd8 <+132>: mov    x20, x0
    0x1009f2ddc <+136>: adrp   x8, 2274
    0x1009f2de0 <+140>: ldr    x1, [x8, #0xea8]
    0x1009f2de4 <+144>: bl     0x100dc5ee8               ; symbol stub for: objc_msgSend
    0x1009f2de8 <+148>: mov    x25, x0
Target 0: (WhatsApp) stopped.
(lldb) ni
Process 12202 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x00000001009f2dd0 WhatsApp` -[WAChatSessionViewController
canPerformAction:withSender:] + 124
WhatsApp`-[WAChatSessionViewController canPerformAction:withSender:]:
->  0x1009f2dd0 <+124>: mov    x29, x29
    0x1009f2dd4 <+128>: bl     0x100dc5f3c               ; symbol stub for:
objc_retainAutoreleasedReturnValue
    0x1009f2dd8 <+132>: mov    x20, x0
    0x1009f2ddc <+136>: adrp   x8, 2274
    0x1009f2de0 <+140>: ldr    x1, [x8, #0xea8]
    0x1009f2de4 <+144>: bl     0x100dc5ee8               ; symbol stub for: objc_msgSend
    0x1009f2de8 <+148>: mov    x25, x0
    0x1009f2dec <+152>: adrp   x8, 2269
Target 0: (WhatsApp) stopped.
(lldb) po $x0
<WAMessage: 0x1547b8070> (entity: WAMessage; id: 0xd000000000140004
<x-coredata://59047C59-7271-4464-8967-DCF81DD3F40D/WAMessage/p5> ; data: {
    chatSession = "0xd000000000080002
<x-coredata://59047C59-7271-4464-8967-DCF81DD3F40D/WAChatSession/p2>";
    childMessages = "<relationship fault: 0x174826340 'childMessages'>";
    childMessagesDeliveredCount = 0;
    childMessagesPlayedCount = 0;
    childMessagesReadCount = 0;
    dataItemVersion = 2;
    dataItems = "<relationship fault: 0x174826460 'dataItems'>";
    docID = 3;
    encRetryCount = 0;
    filteredRecipientCount = 0;
    flags = 0;
    fromJID = "8618698580743@s.whatsapp.net";
    groupEventType = 0;
```

```
    groupMember = nil;
    isFromMe = 0;
    lastSession = nil;
    mediaItem = nil;
    mediaSectionID = nil;
    messageDate = "2017-09-17 14:47:19 +0000";
    messageErrorStatus = 0;
    messageInfo = nil;
    messageStatus = 8;
    messageType = 0;
    parentMessage = nil;
    phash = nil;
    pushName = Alonemonkey;
    sentDate = nil;
    sort = 2;
    spotlightStatus = 0;
    stanzaID = E3015C287F00CA41AB;
    starred = nil;
    text = hello;
    toJID = nil;
})
```

在 WAMessage 里面的结构体中可以找到 text，这就是文本消息的内容。接下来就可以编写代码了。

7.1.3 编写 Tweak

用 MonkeyDev 新建一个 Logs Tweak 项目，指定 plist 中的 Bundle ID 为 net.whatsapp.WhatsApp（读者可以根据自己的设备设置 Build Settings 的 IP 地址和端口）。首先，hook wa_reallyInstallItems，获取原来的 menuItems，然后添加自己的 UIMenuItem，并设置 title 和 action，代码如下。

```
%hook UIMenuController

-(void)wa_reallyInstallItems{
    %log;
    %orig;
    NSMutableArray* menuItemArray = [NSMutableArray arrayWithArray:[self menuItems]];
    UIMenuItem *menuItem = [[UIMenuItem alloc] initWithTitle:@"收藏"
action:@selector(message_collect:)];
    [menuItemArray addObject:menuItem];
    [self setMenuItems:menuItemArray];
}
```

```
%end
```

接下来，hook WAChatSessionViewController 类中的 canPerformAction:withSender: 方法，并新增 message_collect: 方法，代码如下。

```
%hook WAChatSessionViewController

%new
-(void)message_collect:(id) data{
    WAChatCellData* cellData = MSHookIvar<WAChatCellData *>(self,
"_cellDataForMenuController");
    WAMessage* message = [cellData message];
    NSString* text = [message text];
    if(text){
        NSLog(@"text is %@", text);
        if(![collectMessages containsObject:text]){
            [collectMessages addObject:text];
        }
    }
}

-(BOOL)canPerformAction:(SEL) action withSender:(id) sender{
    if(action == @selector(message_collect:)){
        return YES;
    }else{
        return %orig;
    }
}

%end
```

在 message_collect 中，通过私有属性 _cellDataForMenuController 拿到 WAChatCellData，进而拿到 WAMessage 对象。笔者还在设置界面新增了一个选项，用于显示收藏的文本。相关代码如下。

```
%hook WASettingsViewController

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{
    return %orig + 1;
}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    if(section == 4){
```

```
        return 1;
    }
    return %orig;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath
*)indexPath{
    if(indexPath.section == 4 && indexPath.row == 0){

        UITableViewCell* cell = [tableView
dequeueReusableCellWithIdentifier:@"message_collect"];
        if(!cell){
            cell = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
reuseIdentifier:@"message_collect"];
            [cell setSelectionStyle:UITableViewCellSelectionStyleNone];
            [cell setAccessoryType:UITableViewCellAccessoryDisclosureIndicator];
            [cell setBackgroundColor:[UIColor whiteColor]];
        }
        [[cell textLabel] setText:@"收藏的消息"];
        return cell;
    }
    return %orig;
}

-(CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:(NSIndexPath
*)indexPath{
    if(indexPath.section == 4 && indexPath.row == 0){
        return 50;
    }
    return %orig;
}

-(BOOL)tableView:(UITableView *)tableView canEditRowAtIndexPath:(NSIndexPath *)indexPath{
    if(indexPath.section == 4 && indexPath.row == 0){
        return NO;
    }
    return %orig;
}

-(NSIndexPath *)tableView:(UITableView *)tableView willSelectRowAtIndexPath:(NSIndexPath
*)indexPath{
    if(indexPath.section == 4 && indexPath.row == 0){
        return indexPath;
    }
```

```
    return %orig;
}

-(NSString *)tableView:(UITableView *)tableView
titleForHeaderInSection:(NSInteger)section{
    if(section == 4){
        return @"";
    }
    return %orig;
}

-(NSString *)tableView:(UITableView *)tableView
titleForFooterInSection:(NSInteger)section{
    if(section == 4){
        return @"";
    }
    return %orig;
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath
*)indexPath{
    if(indexPath.section == 4 && indexPath.row == 0){
        [self setHidesBottomBarWhenPushed:YES];
        CollectViewController* collectViewController = [CollectViewController new];
        [collectViewController setCollectMessages:collectMessages];
        [self.navigationController pushViewController:collectViewController animated:YES];
        [self setHidesBottomBarWhenPushed:NO];
        return;
    }
    return %orig;
}

%end
```

在编写 Tweak 时，主要通过 Reveal 获取设置中的 UITableView，然后打印它的 datasource 和 delegate，从而找到 WASettingsViewController，最后 hook 其中对应的 delegate 方法，具体的流程留给读者自己分析。

7.1.4　安装与小结

代码编写完成，直接按“Command + B”快捷键即可安装。安装完成，长按消息，就会出现收藏按钮。点击该按钮，就能在设置界面看到收藏的文本了。当然，本书主要是为了讲解分析

过程，在实际的分析中，还可以对代码进行完善，例如加上对文本消息的判断、将消息备份到服务器等。

对这次简单的实践总结如下。

- 在逆向过程中，小部分靠猜测，大部分都是有依据的。
- 要善用 LLDB 进行动态调试分析，并与静态分析结合起来。
- 逆向分析需要扎实的正向编程基础，这样才能知道应该如何分析和思考问题。
- 多实践、多练习，总结并找到一些独特的分析方法和技巧。

7.2 非越狱设备分析

在 7.1 节对越狱设备的分析过程中，笔者通过一步步调试和分析找到了 hook 的函数，但也发现了其中的一些问题，列举如下。

- 在逆向分析过程中，许多步骤都是零散的，需要花很多时间去做准备，例如符号还原、class-dump、Reveal、LLDB 等在分析其他应用时也要做一遍。
- 在分析时只能通过 LLDB 命令行的方式来调试，但这种方式不如通过 Xcode 调试那样方便和快捷。
- 只能在越狱设备上进行调试。如果越狱设备系统的版本太低而装不上 App，就没办法进行调试了。
- 在编写 Tweak 时，只能写一部分代码，然后安装程序，通过打印日志的方式来调试问题，而不能直接 debug。

……

笔者在分析过程中也遇到了一系列问题，很多工作明明是一样的，却偏要做很多次。于是，笔者对这些工具进行整合，将它们集成到 MonkeyDev 里面。通过 MonkeyDev 新建工程，将自动集成 class-dump、符号还原、Reveal、Cycript、注入动态库、自动重签名等一系列重复性的工作。更重要的是，通过 MonkeyDev 可以直接在非越狱设备上进行逆向分析！

本节的目的是分析 WahtsApp 的消息收发函数，并实现在收到消息时自动回复消息。笔者使用 iOS 10.3.2 非越狱机器进行分析。

7.2.1 创建 MonkeyDev 项目

MonkeyDev 的安装在前面已经讲过。在这里直接新建一个 MonkeyApp 项目，项目结构如图

7-5 所示，包含以下部分。

- MethodTraceConfig.plist：跟踪方法调用的配置文件。
- put ipa or app here：将需要分析的解密后的 App 或者 ipa 文件放到该目录下面。
- WhatsAppDylib.xm：支持用 Logos 语法编写 hook 代码，只支持 OC hook。
- WhatsAppDylib.m：可以在该文件中编写注入的 hook 代码。
- MethodTrace：方法跟踪的模块。
- CaptainHook：CaptainHook 的头文件。
- AntiAntiDebug：反反调试的代码。
- fishhook：fishhook 的源代码，用于替换符号表中的地址。
- Frameworks：已经自动集成了 Cycript 和 RevealServer。

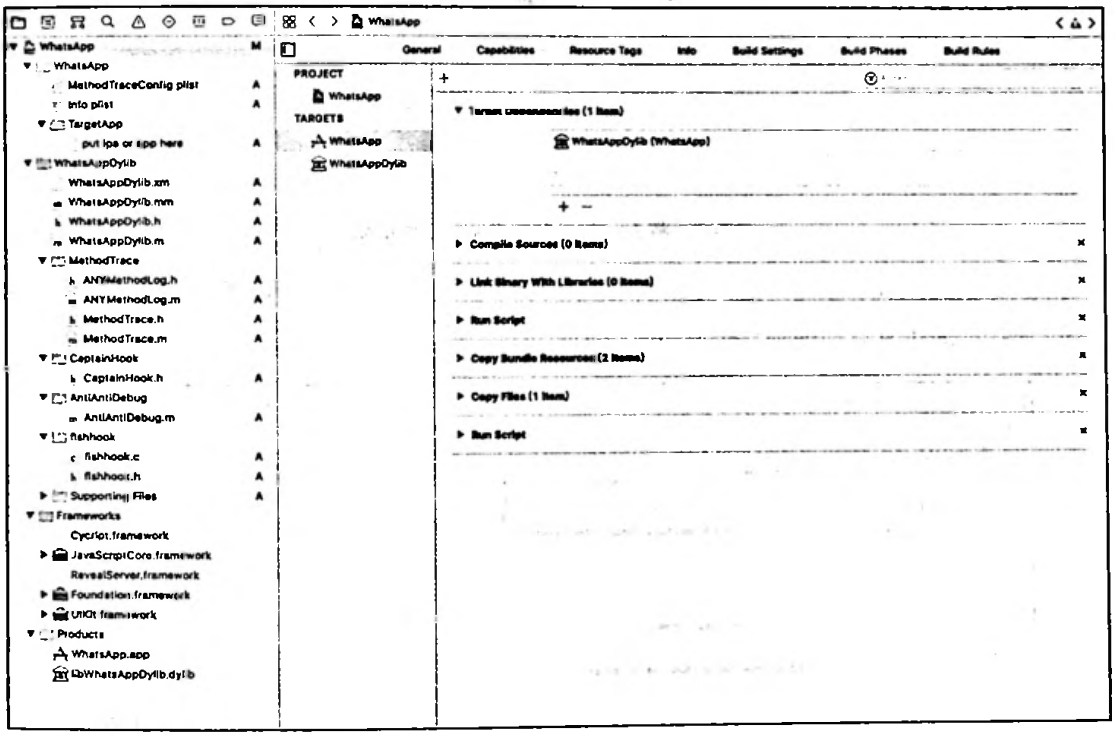


图 7-5　使用 MonkeyDev 新建 MonkeyApp 项目

右键单击“put ipa or app here”项目，Finder 中的显示如图 7-6 所示。然后，将 WhatsApp.app 文件夹放到此处，如图 7-7 所示。也可以通过一些助手工具直接下载越狱应用，将 ipa 文件放

到这里。需要注意的是，虽然这些助手工具提供的越狱应用一般都是解密的，但不排除某些应用没有解密的情况。对这类情况，可以使用 otool 进一步确认。

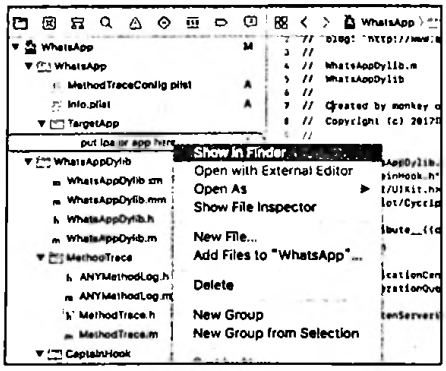


图 7-6 在 Finder 中找到放置 App 的位置

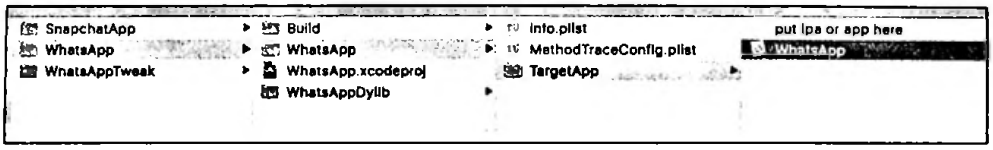


图 7-7 将 App 文件夹或者 ipa 文件放入

如果使用 Xcode 8 新建项目，还需要在 App 的 Target 里面将 dylib 的依赖加进来，如图 7-8 所示。使用 Xcode 9 新建的项目会自动增加依赖。

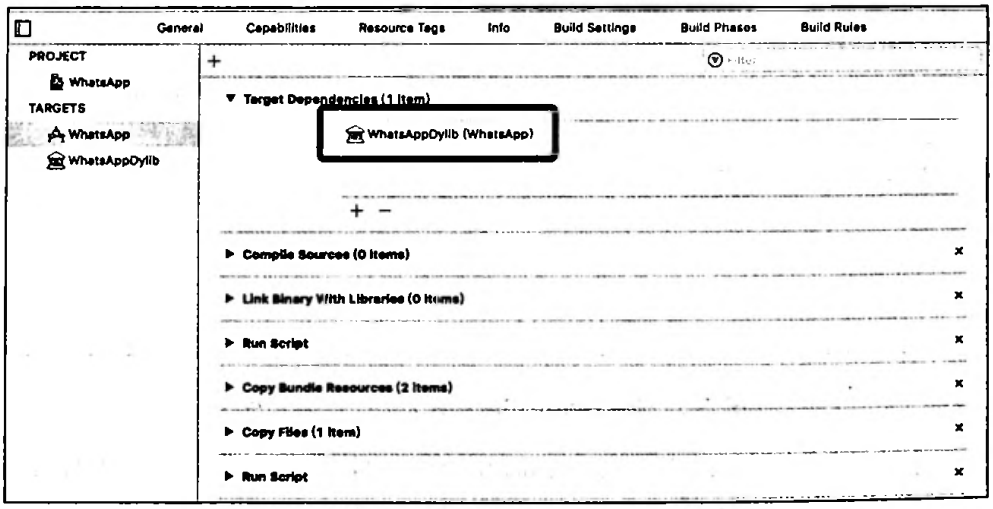


图 7-8 在 App 的 Target 中增加动态库的依赖

准备工作到此基本完成。为了方便后面的调试，还可以在 Build Settings 界面下方分别设置 class-dump 和符号还原为“YES”，以便自动进行 class-dump 和符号还原，如图 7-9 所示。

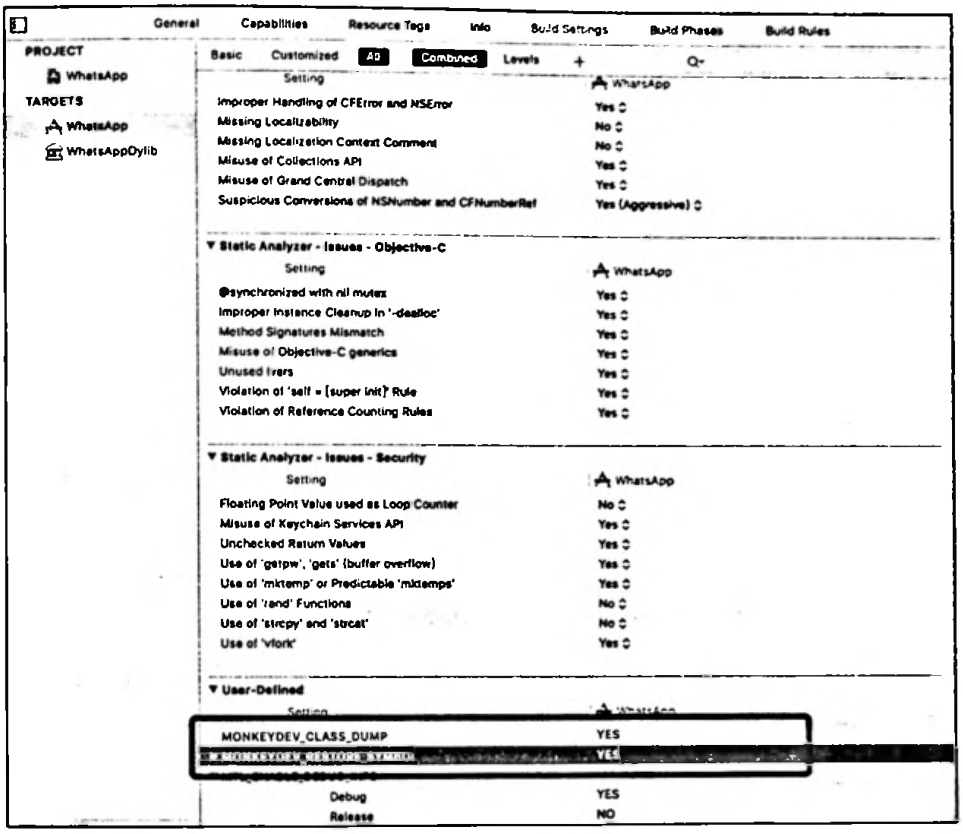


图 7-9　在 Build Settings 界面开启 classdump 和符号还原

最后，可以在环境变量中设置 MallocStackLogging 的值为 1，以便后续查看 malloc 的堆栈。至此，所有准备工作就都做完了，下面我们就要开始进行真正的非越狱逆向实战了。

7.2.2　非越狱逆向实战

单击 Xcode 的运行按钮，便会自动进行注入、重签名等一系列操作，并将程序安装和运行在非越狱设备上。要想分析 WhatsApp 的消息，还是要先来到消息发送界面，如图 7-10 所示，然后打印当前的 ViewController 层次结构。

直接搜索“appeared”关键字，找到当前显示的 ViewController 为 WAChatSessionViewController。在寻找消息处理函数时，笔者会通过监控 WAChatSessionViewController 的方法调用并查看调用

堆栈来寻找线索。有如下两种方法。

- 对 WAChatSessionViewController 的所有方法下断点并逐一查看。
- 使用 MonkeyDev 自带的方法追踪功能打印 WAChatSessionViewController 所有方法的调用。

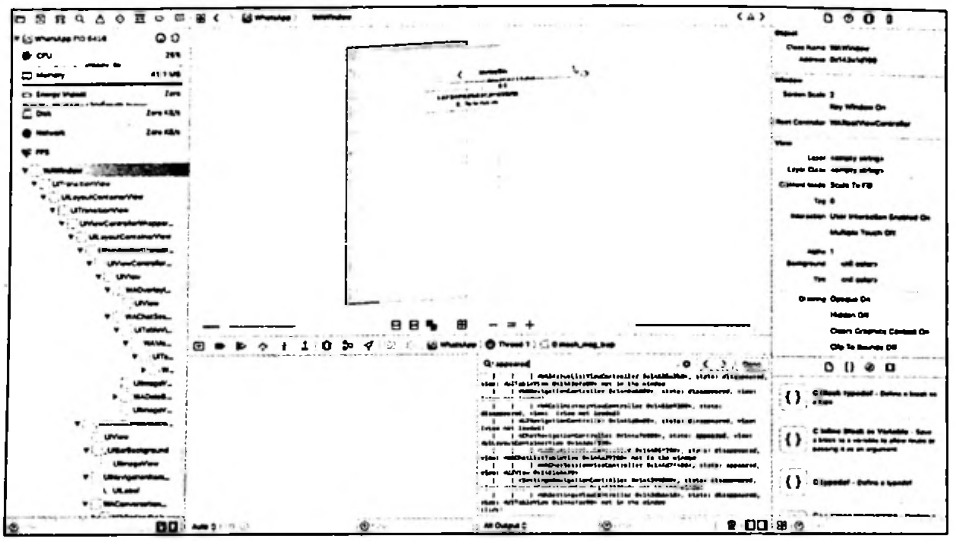


图 7-10 使用 Xcode 查看 WhatsApp 的界面层次结构

我们先选择第一种方案。因为已经进行了符号还原，所以，可以直接通过正则对 WAChatSessionViewController 的所有方法下断点，代码如下。

```
(lldb) br set -r '\[WAChatSessionViewController .*\]'
Breakpoint 1: 633 locations.
```

让程序继续运行，各种断点不断被触发。手动删除这些被触发的断点，代码如下。

```
(lldb) br del 1.69
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
Process 6458 resuming
(lldb) br del 1.68
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
Process 6458 resuming
(lldb) br del 1.268
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
Process 6458 resuming
```

```
(lldb) br del 1.33
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
Process 6458 resuming
(lldb) br del 1.511
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
Process 6458 resuming
(lldb) br del 1.510
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
Process 6458 resuming
(lldb) br del 1.67
0 breakpoints deleted; 1 breakpoint locations disabled.
(lldb) c
```

断点被触发时，在界面右侧可以看到当前被触发的断点的编号。直接删除对应的编号。在没有其他断点被触发后，使用另一个账号发送消息到当前调试的账号，让其触发断点，发现程序中断在 `-[WAChatSessionViewController handleMessageUpdatedNotification:]` 函数处，如图 7-11 所示。打印参数，发现其中一个 WAMessage 对象里面的 text 就是发送的消息。

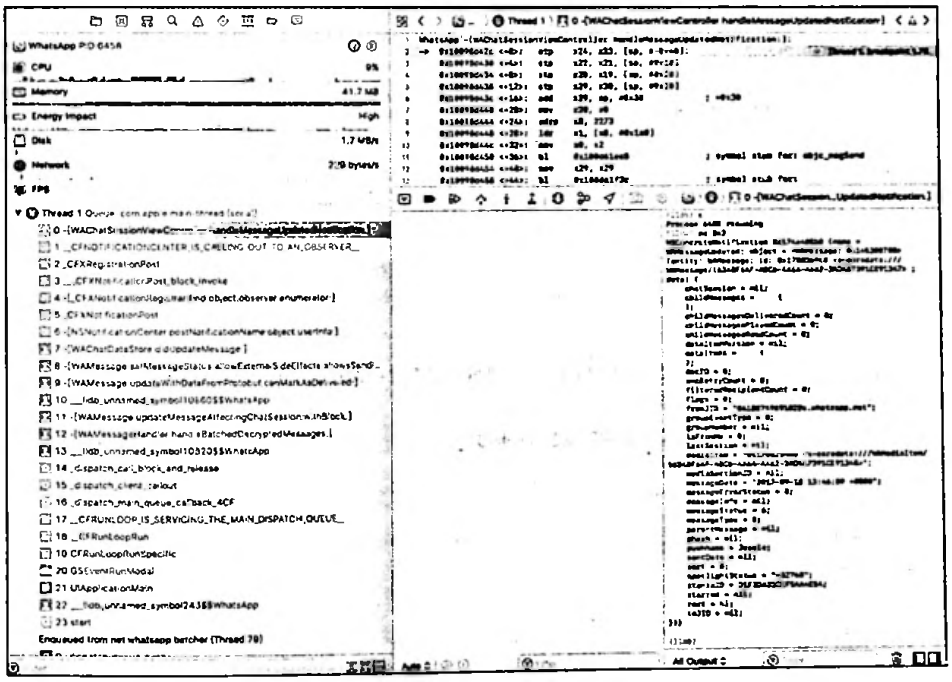


图 7-11　WhatsApp 收到消息后自动触发断点

继续查看调用堆栈 -[WAChatDataStore didUpdateMessage:]，第 2 个参数看起来像一个消息的结构体，上面的 -[WAMessage updateWithDataFromProtobuf:canMarkAsDelivered:] 函数看起来像从 Protobuf 转成一个消息的结构，然后交给 WAChatDataStore 做进一步处理。这样，关键类就找到了，是 WAChatDataStore。

删除所有断点，然后在 -[WAChatDataStore didUpdateMessage:] 函数处下一个断点，以重新发送一条消息来触发断点，其参数如下。对其中不是由发送消息触发的断点，都按“C”键继续运行。

```
(lldb) po $x2
<WAMessage: 0x143d5f5f0> (entity: WAMessage; id: 0x170827ac0
<x-coredata:///WAMessage/t6340F6AF-ABC6-4A6A-A462-3AD467391CE91351> ; data: {
    chatSession = nil;
    childMessages =     (
    );
    childMessagesDeliveredCount = 0;
    childMessagesPlayedCount = 0;
    childMessagesReadCount = 0;
    dataItemVersion = nil;
    dataItems =     (
    );
    docID = 0;
    encRetryCount = 0;
    filteredRecipientCount = 0;
    flags = 0;
    fromJID = "8618888888888@s.whatsapp.net";
    groupEventType = 0;
    groupMember = nil;
    isFromMe = 0;
    lastSession = nil;
    mediaItem = "0x17083c080
<x-coredata:///WAMediaItem/t6340F6AF-ABC6-4A6A-A462-3AD467391CE91352>";
    mediaSectionID = nil;
    messageDate = "2017-09-18 14:00:21 +0000";
    messageErrorStatus = 0;
    messageInfo = nil;
    messageStatus = 6;
    messageType = 0;
    parentMessage = nil;
    phash = nil;
    pushName = NickName;
    sentDate = nil;
```

```
    sort = 0;
    spotlightStatus = "-32768";
    stanzaID = 72B4BD5DC5E9B89143;
    starred = nil;
    text = haha;
    toJID = nil;
})
```

这样我们就可以通过这个方法拿到发送的消息对象了。那么，我们自己发送的消息是由哪个方法处理的呢？下面使用第二种方案对 WAChatDataStore 的所有方法调用进行跟踪。因为笔者不想重启 Xcode 然后到 MethodTraceConfig.plist 文件里面配置需要跟踪的类，所以直接使用 LLDB 调用 MethodTrace 暴露的方法，代码如下。

```
@interface MethodTrace : NSObject

+ (void)addClassTrace:(NSString*) className;

+ (void)addClassTrace:(NSString *)className methodName:(NSString*) methodName;

+ (void)addClassTrace:(NSString *)className methodList:(NSArray*) methodList;

@end
```

通过 LLDB 调用 +[MethodTrace addClassTrace:] 函数来跟踪 WAChatDataStore 的调用，命令如下。

```
(lldb) br delete
About to delete all breakpoints, do you want to do that?: [Y/n] y
All breakpoints removed. (1 breakpoint)
(lldb) e [MethodTrace addClassTrace:@"WAChatDataStore"]
(lldb) c
Process 6458 resuming
```

使用当前账号发送一条消息，发现没有任何输出。笔者在 MethodTrace 的代码里面调试了一下，发现 NSLog 没有被输出到 Xcode 的控制台，但是在 Console.app 里面可以看到 NSLog，所以改用 printf 来输出。直接在 MethodTraceConfig.plist 文件里面进行设置，如图 7-12 所示。设置 ENABLE_METHODTRACE 为“YES”以开启方法追踪。增加一个 Item key 为需要跟踪的类。如果要指定特定的方法，需要在类的 key 下面增加一个 array（参见默认写法）。

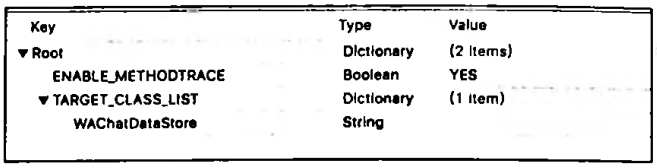


图 7-12 设置 WAChatDataStore 的动态方法调用跟踪

重新运行 Xcode，发送一条文本消息，在控制台获得的输出如下。

```
[<WAChatStorage: 0x13de2f250> handleSaveOutcome:unknown ]
ret:void
[<WAChatStorage: 0x13de2f250> isStoreAdded:(null) ]
ret:1
[<WAChatStorage: 0x13de2f250> ownsManagedObject:<WAMessage: 0x13f15d8f0> (entity:
WAMessage; id: 0xd0000000001c0004
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAMessage/p7> ; data: {
    chatSession = "0xd000000000040002
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAChatSession/p1>";
    childMessages =     (
    );
    childMessagesDeliveredCount = 0;
    childMessagesPlayedCount = 0;
    childMessagesReadCount = 0;
    dataItemVersion = 2;
    dataItems =     (
    );
    docID = nil;
    encRetryCount = 0;
    filteredRecipientCount = 0;
    flags = 0;
    fromJID = nil;
    groupEventType = 0;
    groupMember = nil;
    isFromMe = 1;
    lastSession = "0xd000000000040002
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAChatSession/p1>";
    mediaItem = nil;
    mediaSectionID = nil;
    messageDate = "2017-09-18 14:24:33 +0000";
    messageErrorStatus = 0;
    messageInfo = nil;
    messageStatus = 9;
    messageType = 0;
    parentMessage = nil;
```

```
    phash = nil;
    pushName = nil;
    sentDate = nil;
    sort = 7;
    spotlightStatus = 0;
    stanzaID = A81B108B93CC769EF1;
    starred = nil;
    text = hello;
    toJID = "8618888888888@s.whatsapp.net";
}) ]
ret:1
[<WAChatStorage: 0x13de2f250> didUpdateMessage:<WAMessage: 0x13f15d8f0> (entity: WAMessage;
id: 0xd0000000001c0004 <x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAMessage/p7> ;
data: {
    chatSession = "0xd000000000040002
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAChatSession/p1>";
    childMessages =     (
    );
    childMessagesDeliveredCount = 0;
    childMessagesPlayedCount = 0;
    childMessagesReadCount = 0;
    dataItemVersion = 2;
    dataItems =     (
    );
    docID = nil;
    encRetryCount = 0;
    filteredRecipientCount = 0;
    flags = 0;
    fromJID = nil;
    groupEventType = 0;
    groupMember = nil;
    isFromMe = 1;
    lastSession = "0xd000000000040002
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAChatSession/p1>";
    mediaItem = nil;
    mediaSectionID = nil;
    messageDate = "2017-09-18 14:24:33 +0000";
    messageErrorStatus = 0;
    messageInfo = nil;
    messageStatus = 9;
    messageType = 0;
    parentMessage = nil;
    phash = nil;
    pushName = nil;
    sentDate = nil;
```

```
    sort = 7;
    spotlightStatus = 0;
    stanzaID = A81B108B93CC769EF1;
    starred = nil;
    text = hello;
    toJID = "8618888888888@s.whatsapp.net";
}) ]
ret:void
[<WAChatStorage: 0x13de2f250> save:(null) ]
......
```

从跟踪的日志输出中可知，这里分别调用了 ownsManagedObject 和 didUpdateMessage 函数。在 ownsManagedObject 函数处下断点，查看调用堆栈。因为在头文件中搜索 ownsManagedObject 时发现该方法的实现是在父类 WAChatDataStore 里面的，所以也可以在 LLDB 中使用 po [WAChatStorage_shortMethodDescription] 命令来查看。发送文本，触发断点，如图 7-13 所示。

```
(lldb) b -[WAChatDataStore ownsManagedObject:]
Breakpoint 6: where = WhatsApp`-[WAChatDataStore ownsManagedObject:], address =
0x0000000100a0f868
(lldb) c
Process 7071 resuming
```

从堆栈中可以看到调用函数 -[WAChatStorage sendMessageWithText:metadata:multicast:replyingToMessage:inChatSession:hasTextFromURL:openedFromURL:]。从函数名来看，只需要传入一个文本和发送的群组。在该函数上下断点，查看参数的值，具体如下。

```
(lldb) b -[WAChatStorage
sendMessageWithText:metadata:multicast:replyingToMessage:inChatSession:hasTextFromURL:o
penedFromURL:]
Breakpoint 7: where = WhatsApp`-[WAChatStorage
sendMessageWithText:metadata:multicast:replyingToMessage:inChatSession:hasTextFromURL:o
penedFromURL:], address = 0x00000001000f8094
(lldb) c
Process 7071 resuming
(lldb) c
Process 7071 resuming
(lldb) po $x2
Hello

(lldb) po $x3
<nil>
```

```
(lldb) po $x4
<nil>

(lldb) po $x5
<nil>

(lldb) po $x6
<WAChatSession: 0x1740db0b0> (entity: WAChatSession; id: 0xd000000000040002
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAChatSession/p1> ; data: {
    archived = 0;
    contactABID = 0;
    contactIdentifier = "19F8BF26-3C82-4A83-8C79-80F75EF11298";
    contactJID = "8618874969102@s.whatsapp.net";
    eTag = nil;
    flags = 1296;
    groupInfo = nil;
    groupMembers = "<relationship fault: 0x174628a40 'groupMembers'>";
    hidden = 0;
    identityVerificationEpoch = 0;
    identityVerificationState = 0;
    lastMessage = "0xd000000000f00004
<x-coredata://6A86E2FA-CBD1-4C77-95E7-A6BC0351D819/WAMessage/p60>";
    lastMessageDate = "2017-09-19 09:25:18 +0000";
    lastMessageText = nil;
    locationSharingEndDate = nil;
    messageCounter = 61;
    partnerName = MonkeyDev;
    properties = nil;
    removed = 0;
    savedInput = nil;
    sessionType = 0;
    spotlightStatus = "-S";
    unreadCount = 0;
})

(lldb) po $x7
<nil>
```

这样，在收到消息时即可直接调用该函数。顺便到头文件里面搜索关键字“sendMessage WithText”，找到了下面这个函数，比刚才那个短一点。稍后可以先用这个函数试试。

```
(void)sendMessageWithText:(id)arg1 metadata:(id)arg2 toChatSessions:(id)arg3
hasTextFromURL:(_Bool)arg4;
```

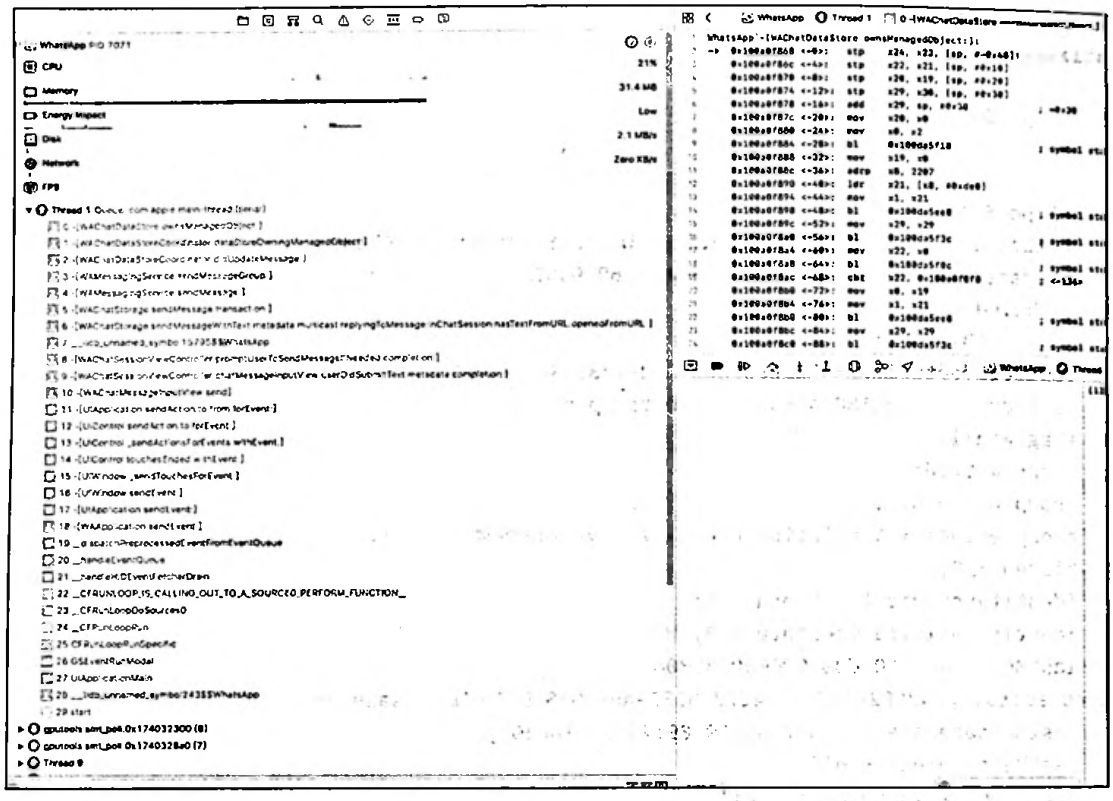


图 7-13 发送文本并触发 Xcode 断点 ownsManagedObject

有了这个方法之后，将需要的参数都准备好，然后在 didUpdateMessage 收到消息时调用即可。既然发送的文本可以直接从 WAMessage 里面获取，那么 chatsession 呢？到 WAMessage 的头文件里面找找，发现也有 chatsession。最后，需要获取发送消息的对象 WAChatStorage。我们可以从堆栈的上一层调用入手，查看调用 sendMessageWithText 的上一层堆栈，如果没有符号化，那么这应该是一个 block。用调用的地址减基地址，就可以得到其在文件中的偏移了。相关命令如下。

```
0x1009e1d20 <+220>: mov    x6, x24
0x1009e1d24 <+224>: mov    x7, x25
0x1009e1d28 <+228>: bl     0x100da5ee8               ; symbol stub for: objc_msgSend
0x1009e1d2c <+232>: mov    x29, x29
0x1009e1d30 <+236>: bl     0x100da5f3c               ; symbol stub for:
objc_retainAutoreleasedReturnValue
0x1009e1d34 <+240>: bl     0x100da5f0c               ; symbol stub for: objc_release
```

```
(lldb) im li -o -f WhatsApp
[  0] 0x00000000000c8000
/Users/alonemonkey/Documents/iosreversebook/sourcecode/chapter-7/WhatsApp/Build/Product
s/Debug-iphoneos/WhatsApp.app/WhatsApp
(lldb) p/x 0x1009e1d2c - 0x00000000000c8000
(long) $7 = 0x0000000100919d2c
```

在 Hopper 中按“G”键，跳转到指定位置，单击查看伪代码——竟然没有反应。继续往上查看代码，具体如下。

```
0000000100919c20        bl          imp___stubs__objc_release
0000000100919c24        ldp         x29, x30, [sp, #0xa0]
0000000100919c28        ldp         x20, x19, [sp, #0x90]
0000000100919c2c        ldp         x22, x21, [sp, #0x80]
0000000100919c30        ldp         x24, x23, [sp, #0x70]
0000000100919c34        ldp         x26, x25, [sp, #0x60]
0000000100919c38        ldp         x28, x27, [sp, #0x50]
0000000100919c3c        add         sp, sp, #0xb0
0000000100919c40        ret
                      ; endp
0000000100919c44        dd          0xd10183ff                          ; DATA
XREF=-[WAChatSessionViewController
chatMessageInputView:userDidSubmitText:metadata:completion:]+528
0000000100919c48        stp         x26, x25, [sp, #0x10]
0000000100919c4c        stp         x24, x23, [sp, #0x20]
0000000100919c50        stp         x22, x21, [sp, #0x30]
0000000100919c54        stp         x20, x19, [sp, #0x40]
0000000100919c58        stp         x29, x30, [sp, #0x50]
0000000100919c5c        add         x29, sp, #0x50
0000000100919c60        mov         x20, x0
0000000100919c64        mov         x0, x1
0000000100919c68        bl          imp___stubs__objc_retain
0000000100919c6c        mov         x19, x0
0000000100919c70        cbz         x19, -[WAChatSessionViewController
chatMessageInputView:userDidSubmitText:metadata:completion:]+1112
0000000100919c74        ldr         x0, [x20, #0x28]
0000000100919c78        adrp        x8, #0x1011ea000
0000000100919c7c        ldr         x1, [x8, #0x5d0]
0000000100919c80        bl          imp___stubs__objc_msgSend
```

0000000100919c48 处显然是一个新的函数，但 Hopper 没有把它识别出来。单击此处，然后单击左上角的“P”按钮，将其转换成一个函数，再查看伪代码，具体如下。

```
int sub_100919c48(int arg0, int arg1) {
    *(r31 + 0x10) = r26;
    *(0x20 + r31) = r25;
    *(r31 + 0x20) = r24;
    *(0x30 + r31) = r23;
    *(r31 + 0x30) = r22;
    *(0x40 + r31) = r21;
    *(r31 + 0x40) = r20;
    *(0x50 + r31) = r19;
    *(r31 + 0x50) = r29;
    *(0x60 + r31) = r30;
    r29 = r31 + 0x50;
    r20 = arg0;
    r19 = [arg1 retain];
    if (r19 != 0x0) {
            [*(r20 + 0x28) transitionToShowingOnlyMostRecentMessagesIfNeeded];
            r21 = [[WASharedAppData chatStorage] retain];
            r23 = [[*(0x40 + r20) quotedMessage] retain];
            r24 = [[*(r20 + 0x28) chatSession] retain];
            [*(r20 + 0x28) hasTextFromURL];
            *(int8_t *)r31 = [*(r20 + 0x28) openedFromURL];
            [[[r21 sendMessageWithText:r19 metadata:*(r20 + 0x30) multicast:stack[2048]
replyingToMessage:stack[2049] inChatSession:stack[2050] hasTextFromURL:stack[2051]
openedFromURL:stack[2052]] retain] release];
            [r24 release];
            [r23 release];
            [r21 release];
            [*(r20 + 0x28) setHasTextFromURL:zero_extend_64(0x0)];
            [*(r20 + 0x28) setOpenedFromURL:zero_extend_64(0x0)];
            r0 = [WALogWriter sharedLogger];
            r0 = [r0 retain];
            [r0 write:0x7
format:@"chatview//chatbarmanager/userdidsubmittext/reset-url-flags"];
            [r0 release];
            r0 = *(r20 + 0x20);
            r8 = *(r0 + 0x10);
            r1 = 0x1;
    }
    else {
            r0 = *(r20 + 0x20);
            r8 = *(r0 + 0x10);
            r1 = zero_extend_64(0x0);
    }
    (r8)(r0, r1, 0x7, @"chatview//chatbarmanager/userdidsubmittext/reset-url-flags");
```

```
    r0 = [r19 release];
    return r0;
}
```

从伪代码中我们看到，可以通过 [WASharedAppData chatStorage] 获取 chatStorage 的对象，然后调用 sendMessageWithText 发送文本消息。在拿到所有需要的参数后，就可以开始编写代码了。

7.2.3　编写 hook 代码

下面通过 CaptainHook 来编写代码。首先，构造函数 hook 方法，代码如下。

```
CHDeclareClass(WAChatDataStore)
CHOptimizedMethod1(self, void, WAChatDataStore, didUpdateMessage, WAMessage*, message){
    CHSuper1(WAChatDataStore, didUpdateMessage, message);
}
CHConstructor{
    CHLoadLateClass(WAChatDataStore);
    CHClassHook1(WAChatDataStore, didUpdateMessage);
}
```

然后，在代码中获取发送的文本、发送人、发送的群组和消息状态。最后，调用发送接口进行发送。其中的方法需要在前面申明。编写代码时可能不会很顺利，可以通过下断点来调试自己编写的代码并打印相关的参数。笔者在测试时发现，收到消息时该函数会被多次调用，因此，为了确保只处理 1 次，需要通过动态调试来对比每次发送的 WAMessage 的内容。当前代码只能在聊天界面上生效，读者可以自己实现不在聊天界面上就可以处理消息的方法，代码如下。

```
@class WAChatSession;

@interface WAMessage : NSObject

@property (nonatomic, strong) NSString* text;

@property (nonatomic, strong) NSString* fromJID;

@property (nonatomic, assign) int messageStatus;

@property (nonatomic, retain) WAChatSession *chatSession;

@end

@interface WAChatStorage : NSObject
```

```
- (void)sendMessageWithText:(NSString*) text metadata:(id)arg2 toChatSessions:(NSArray*)
chatSessions hasTextFromURL:(_Bool)arg4;

@end

@interface WASharedAppData

+(WAChatStorage*)chatStorage;

@end

CHDeclareClass(WAChatDataStore)

CHOptimizedMethod1(self, void, WAChatDataStore, didUpdateMessage, WAMessage*, message){

    NSString* text = [message text];

    text = [NSString stringWithFormat:@"you send: %@", text];

    NSString* fromJID = [message fromJID];

    int messageStatus = [message messageStatus];

    WAChatSession* chatSession = [message chatSession];

    WAChatStorage* storage = [(Class)objc_getClass("WASharedAppData") chatStorage];

    if(fromJID && chatSession && messageStatus == 10){
       [storage sendMessageWithText:text metadata:nil toChatSessions:@[chatSession]
hasTextFromURL:NO];
    }

    CHSuper1(WAChatDataStore, didUpdateMessage, message);

}
```

写完代码后，直接运行。MonkeyDev 会自动编译动态库并使注入生效，WhatsApp 收到消息后会自动回复对应的消息。

7.2.4 制作非越狱 Pod

编写自己的非越狱插件之后，读者可以将插件制作成 Pod，上传到笔者创建的私有 CocoaPods 仓库中（https://github.com/AloneMonkey/MonkeyDevSpecs）。该仓库里面有一些网友上传的插件，

读者可以学习和使用。下面讲解如何将自己开发的插件制作成 Pod 并通过 CocoaPods 实现一键安装。

新建 Xcode 工程，选择 MonkeyDev 的模板 MonekyPod。新建项目的结构如图 7-14 所示，包括头文件、源文件和一个 podspec 文件。

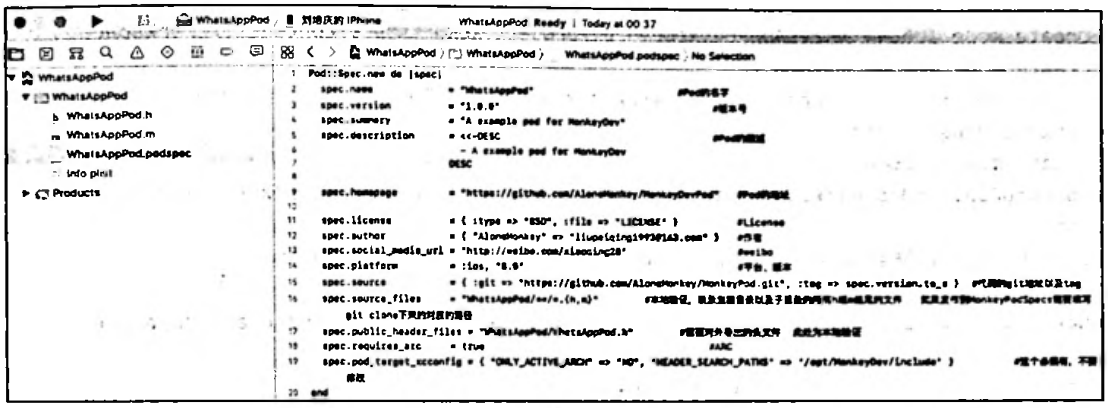


图 7-14　使用 MonkeyDev 模块新建 MonkeyPod

将刚刚编写的 CaptainHook 的代码粘贴到 WhatsAppPod.m 文件中。读者可以自己修改 WhatsAppPod.podspec 中对应的信息，将其移动到根目录下面，并将其从工程引用中移除，然后在 Finder 中显示并进行本地验证，具体如下。

```
➜  WhatsAppPod git:(master) X pod lib lint --allow-warnings

 -> WhatsAppPod (1.0.0)

WhatsAppPod passed validation.
```

本地验证通过后，将在 GitHub 中创建一个 repo。将本地代码上传到 repo 中并打包，在此之前需要确保 WhatsAppPod.podspec 中的 source_files 和 public_header_files 与上传到 GitHub 之后克隆下来的目录结构对应，具体如下。

```
spec.source_files      = "WhatsAppPod/**/*.{h,m}"
spec.public_header_files = "WhatsAppPod/WhatsAppPod.h"
```

将本地代码上传到 GitHub 中并打包，具体如下。

```
➜  WhatsAppPod git:(master) X git init
Initialized empty Git repository in
```

```
/Users/alonemonkey/Documents/iosreversebook/sourcecode/chapter-7/WhatsAppPod/.git/
➜  WhatsAppPod git:(master) ✗ git add .
➜  WhatsAppPod git:(master) ✗ git commit -m "first commit"
[master (root-commit) 27c3534] first commit
 13 files changed, 559 insertions(+)
 create mode 100644 WhatsAppPod.podspec
 create mode 100644 WhatsAppPod.xcodeproj/project.pbxproj
 create mode 100644 WhatsAppPod.xcodeproj/project.xcworkspace/contents.xcworkspacedata
 create mode 100644
WhatsAppPod.xcodeproj/project.xcworkspace/xcuserdata/alonemonkey.xcuserdatad/UserInterf
aceState.xcuserstate
 create mode 100644
WhatsAppPod.xcodeproj/xcuserdata/alonemonkey.xcuserdatad/xcschemes/xcschememanagement.p
list
 create mode 100644 WhatsAppPod.xcworkspace/contents.xcworkspacedata
 create mode 100644
WhatsAppPod.xcworkspace/xcuserdata/alonemonkey.xcuserdatad/UserInterfaceState.xcusersta
te
 create mode 100644 WhatsAppPod/Info.plist
 create mode 100644 WhatsAppPod/WhatsAppPod.h
 create mode 100644 WhatsAppPod/WhatsAppPod.m
➜  WhatsAppPod git:(master) ✗ git remote add origin
https://github.com/AloneMonkey/WhatsAppPod
➜  WhatsAppPod git:(master) ✗ git fetch
warning: no common commits
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), done.
From https://github.com/AloneMonkey/WhatsAppPod
 * [new branch]      master     -> origin/master
➜  WhatsAppPod git:(master) ✗ git branch --set-upstream-to=origin/master master
Branch master set up to track remote branch master from origin.
➜  WhatsAppPod git:(master) ✗ git pull origin master --allow-unrelated-histories
From https://github.com/AloneMonkey/WhatsAppPod
 * branch            master     -> FETCH_HEAD
Merge made by the 'recursive' strategy.
 LICENSE   | 21 ++++++++++++++++++++
 README.md |  2 ++
 2 files changed, 23 insertions(+)
 create mode 100644 LICENSE
 create mode 100644 README.md
➜  WhatsAppPod git:(master) ✗ git tag -a 1.0.0 -m "release 1.0.0"
➜  WhatsAppPod git:(master) ✗ git push -u origin master
```

```
Counting objects: 28, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (27/27), done.
Writing objects: 100% (28/28), 21.81 KiB | 0 bytes/s, done.
Total 28 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/AloneMonkey/WhatsAppPod
   11a7d62..4ef645f  master -> master
Branch master set up to track remote branch master from origin.
➜  WhatsAppPod git:(master) X git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 167 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/AloneMonkey/WhatsAppPod
 * [new tag]         1.0.0 -> 1.0.0
➜  WhatsAppPod git:(master) X
```

访问 https://github.com/AloneMonkey/MonkeyDevSpecs，将其 fork 到自己的 GitHub 仓库（你需要将后文的“https://Github.com/AloneMonkey/MonkeyDevSpecs.git”换成自己 fork 的链接），然后增加并提交到私有 CocoaPods 仓库中，具体如下。

```
➜  WhatsAppPod git:(master) X pod repo add MonkeyDevSpecs
https://github.com/AloneMonkey/MonkeyDevSpecs.git
Cloning spec repo `MonkeyDevSpecs` from
`https://github.com/AloneMonkey/MonkeyDevSpecs.git`

➜  WhatsAppPod git:(master) X pod repo push MonkeyDevSpecs WhatsAppPod.podspec
--allow-warnings

Validating spec
 -> WhatsAppPod (1.0.0)
    - WARN  | url: The URL (https://github.com/AloneMonkey/WhatsAppPod) is not reachable.

Updating the `MonkeyDevSpecs' repo

Already up-to-date.

Adding the spec to the `MonkeyDevSpecs' repo

 - [Add] WhatsAppPod (1.0.0)

Pushing the `MonkeyDevSpecs' repo
```

成功提交之后，向 MonkeyDevSpecs 中提交 pull request，待 pull request 通过。接下来，直接使用 CocoaPods 集成刚刚编写的非越狱插件，删除原来编写的 hook 代码，在根目录下新建 podfile 文件，内容如下。

```
source 'https://github.com/AloneMonkey/MonkeyDevSpecs.git'

use_frameworks!

target 'WhatsAppDylib' do
    pod 'WhatsAppPod'
end
```

在这里使用了 use_frameworks!（通过动态库的方式导入，target 指定为动态库，通过 pod install 命令集成）。如果出现“Unable to find a specification for WhatsAppPod”错误，可以先运行 open ~/.cocoapods 命令，打开文件夹，然后删除 repos 下面的 MonkeyDevSpecs 和 AloneMonkey，再重新安装，具体如下。

```
➜  WhatsApp git:(master) X pod install
Cloning spec repo `alonemonkey` from `https://github.com/AloneMonkey/MonkeyDevSpecs.git`
Analyzing dependencies
Downloading dependencies
Installing WhatsAppPod (1.0.0)
Generating Pods project
Integrating client project

[!] Please close any current Xcode sessions and use `WhatsApp.xcworkspace` for this project from now on.
Sending stats
Pod installation complete! There is 1 dependency from the Podfile and 1 total pod installed.
```

重新打开 workspace 并运行，发现尽管 hook 的代码被删除了，但通过 CocoaPods 集成的 Pod 生效了！读者可以通过这种方法一键集成 MonkeyDevSpecs 上面的非越狱插件。

7.2.5 小结

通过本次实践发现，使用 MonkeyDev 进行逆向实在是太方便了，不但可以在非越狱的机器上面进行逆向分析，还能自动完成各种准备工作，并使用 Xcode 本身强大的调试功能，甚至能一键集成非越狱插件，小结如下。

- 对不能直接从界面上找到入手点的，一般可以从 ViewController 入手跟踪方法的执行。

- 要善于利用堆栈信息来分析程序，并使用 Xcode 提供的内存引用关系图来帮助理清引用关系。
- 在逆向过程中，尽管只有少数情况靠猜测，但猜测也是基于丰富的正向开发的经验进行的，而不是无厘头地猜测。
- 正向开发基础知识很重要。只有基础扎实，分析起来才能如鱼得水。

7.3　Frida 实战应用

Frida 是一款跨平台的注入工具，通过注入 JS（JavaScript）与 Native 的 JS 引擎进行交互，从而执行 Native 的代码进行 hook 和动态调用。Frida 支持 Windows、maxOS、Linux、iOS、Android 等。在这里笔者只介绍 Frida 在 iOS 上的应用，以及如何通过 Frida 来提高逆向的效率。

7.3.1　Frida 的安装

为了使用 Frida，需要在 Mac OS 和 iOS 上面分别安装 Frida。在 Mac OS 中通过如下命令安装 Frida。

```
➜  ~ pip install --user frida
Requirement already satisfied: frida in /usr/local/lib/python2.7/site-packages
Requirement already satisfied: colorama>=0.2.7 in /usr/local/lib/python2.7/site-packages
(from frida)
Requirement already satisfied: prompt-toolkit>=0.57 in
/usr/local/lib/python2.7/site-packages (from frida)
Requirement already satisfied: pygments>=2.0.2 in /usr/local/lib/python2.7/site-packages
(from frida)
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python2.7/site-packages (from
prompt-toolkit>=0.57->frida)
Requirement already satisfied: wcwidth in /usr/local/lib/python2.7/site-packages (from
prompt-toolkit>=0.57->frida)

➜  Frida git:(master) ✗ sudo easy_install frida
Password:
Searching for frida
Best match: frida 10.3.1
Adding frida 10.3.1 to easy-install.pth file
Installing frida-kill script to /usr/local/bin
Installing frida-discover script to /usr/local/bin
Installing frida-ls-devices script to /usr/local/bin
Installing frida-ps script to /usr/local/bin
```

```
Installing frida script to /usr/local/bin
Installing frida-trace script to /usr/local/bin

Using /usr/local/lib/python2.7/site-packages
Processing dependencies for frida
Finished processing dependencies for frida
```

安装之后，运行命令查看当前的版本。例如，运行 sudo easy_install --upgrade frida 命令更新 Frida，具体如下。

```
➜  Frida git:(master) ✗ sudo easy_install --upgrade frida
Password:
Searching for frida
Reading https://pypi.python.org/simple/frida/
Best match: frida 10.5.15
Downloading
https://pypi.python.org/packages/c8/9f/73ed09886eaa2220d5c6e47228d6db126f0a75187471e37f
fcd240419d09/frida-10.5.15-py2.7-macosx-10.10-intel.egg#md5=40ade9cda1588febb28154be92b
93930
Processing frida-10.5.15-py2.7-macosx-10.10-intel.egg
Moving frida-10.5.15-py2.7-macosx-10.10-intel.egg to /Library/Python/2.7/site-packages
Adding frida 10.5.15 to easy-install.pth file
Installing frida-kill script to /usr/local/bin
Installing frida-discover script to /usr/local/bin
Installing frida-ls-devices script to /usr/local/bin
Installing frida-ps script to /usr/local/bin
Installing frida script to /usr/local/bin
Installing frida-trace script to /usr/local/bin

Installed /Library/Python/2.7/site-packages/frida-10.5.15-py2.7-macosx-10.10-intel.egg
Processing dependencies for frida
Finished processing dependencies for frida
➜  Frida git:(master) ✗ frida --version
10.5.15
```

然后，打开越狱设备上面的 Cydia，点击“软件源”→“编辑”→“添加”选项，输入“https://build.frida.re”，确认添加（注意：这里是“https”，而默认的是“http”）。添加源之后，搜索“Frida”并安装，如图 7-15 所示。

Frida 安装完成，环境就准备好了。下面我们开始使用 Frida 提供的接口进行实际操作。

图 7–15　在越狱设备的 Cydia 中安装 Frida

7.3.2　Frida 的初级使用

新建 main.py 文件，设置环境，编写代码并增加入口函数，具体如下。

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import frida

def main():


if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        sys.exit()
    else:
        pass
    finally:
        pass
```

下面分别编写几个功能点的代码。

1. 获取 USB 设备

通过 Frida 提供的 Python 模块获取当前连接的安装了 Frida 的 USB 设备，具体如下。

```
#获取第 1 个 USB 连接的设备
def get_usb_iphone():
    dManager = frida.get_device_manager();   #获取设备管理器
    changed = threading.Event()
    def on_changed():
        changed.set()
    dManager.on('changed',on_changed)        #监听添加设备的事件

    device = None
    while device is None:
        devices = [dev for dev in dManager.enumerate_devices() if dev.type =='tether'] #
类型为 tether 的是 USB 连接的设备
        if len(devices) == 0:
            print 'Waiting for usb device...'
            changed.wait()
        else:
            device = devices[0]              #获取第 1 个设备

    dManager.off('changed',on_changed)

    return device
```

在 main 函数里面调用该函数，获取设备对象，输出内容如下。

```
def main():
    #获取 USB 设备
    device = get_usb_iphone()
    print '设备信息:' + str(device)

➜  Frida git:(master) ✗ ./main.py
Waiting for usb device...
设备信息:Device(id="57ff6c79061c9ab765b6463a51c30dbdbf0e36ff", name="iPhone",
type='tether')
```

2. 枚举运行进程信息

通过 get_usb_iphone 获取设备对象之后，可以通过 enumerate_processes 获取当前的运行进程信息，具体如下。

```
#枚举运行进程信息
def listRunningProcess():
    device = get_usb_iphone();
    processes = device.enumerate_processes();
    processes.sort(key = lambda item : item.pid)
    outWrite('%-10s\t%s' % ('pid', 'name'))
    for process in processes:
        outWrite('%-10s\t%s' % (str(process.pid),process.name))
```

运行结果如下。还可以通过 grep 命令查看某个进程的 ID，具体如下。

```
➜  Frida git:(master) ✗ ./main.py
pid         name
0           kernel_task
1           launchd
26          BTServer
28          CommCenter
37          sandboxd
39          fileproviderd
42          misd
48          UserEventAgent
52          aggregated
......

➜  Frida git:(master) ✗ ./main.py | grep WhatsApp
12251       WhatsApp
```

3. 枚举安装应用程序的信息

正常获取安装应用程序的信息，可以通过调用 LSApplicationWorkspace 的私有 API 来实现。那么，如何通过 Frida 来获取这些信息呢？肯定还是要注入进程，然后在进程里面通过注入 JS 代码去执行 Native 的函数，调用私有 API 获取这些信息，由 JS 返回，再和 Python 通信，从而将结果输出到控制台。如果是这样，就有以下两个问题。

- 如何通过注入 JS 代码执行 Native 方法？
- Python 如何通过与 JS 通信获取执行结果？

其实只需要以下 4 步。

①注入进程，获取 session，具体如下。

```
session = device.attach(u'SpringBoard')
```

②注入 JS 文件，设置回调，具体如下。

```
#加载 JS 文件脚本
def loadJsFile(session, filename):
    source = ''
    with codecs.open(filename, 'r', 'utf-8') as f:
        source = source + f.read()
    script = session.create_script(source)
    script.on('message', on_message)                    #设置 JS 返回数据的 Python 回调
    script.load()                                       #加载 JS 脚本
    return script
```

对应的 JS 文件的内容如下。

```
const LSApplicationWorkspace = ObjC.classes.LSApplicationWorkspace;
const LSApplicationProxy = ObjC.classes.LSApplicationProxy;

function getDataDocument(appid){
    const dataUrl =
LSApplicationProxy.applicationProxyForIdentifier_(appid).dataContainerURL();
    if(dataUrl){
        return dataUrl.toString() + '/Documents';
    }else{
        return "null";
    }
}

function installed(){
    const workspace = LSApplicationWorkspace.defaultWorkspace();
    const apps = workspace.allApplications();
    var result;
    for(var index = 0; index < apps.count(); index++){
        var proxy = apps.objectAtIndex_(index);
        result = result + proxy.localizedName().toString() + '\t' +
proxy.bundleIdentifier().toString()+'\t'+getDataDocument(proxy.bundleIdentifier().toStr
ing())+'\n';
    }
    send({app : result});    //将结果返回 Python
};

//处理 Python 发送过来的消息
function handleMessage(message) {
  if(message['cmd']){
    if(message['cmd'] == 'installed'){
```

```
            installed();
        }
    }
    send({ mes: 'Successful operation!'});
    send({ finished: 'yes'});
}

recv(handleMessage);
```

③Python 发送消息给 JS，JS 执行函数返回，具体如下。

```
script.post({'cmd' : 'installed'})
```

④Python 处理回调，输出内容，具体如下。

```
#从 JS 接收信息
def on_message(message, data):
    if message.has_key('payload'):
        payload = message['payload']
        if isinstance(payload, dict):
            deal_message(payload)
        else:
            print payload

#处理 JS 中不同的信息
def deal_message(payload):
    #基本信息输出
    if payload.has_key('mes'):
        print payload['mes']

    #安装 App 的信息
    if payload.has_key('app'):
        app = payload['app']
        lines = app.split('\n')
        for line in lines:
            if len(line):
                arr = line.split('\t')
                if len(arr) == 3:
                    outWrite('%-40s\t%-70s\t%-80s' % (arr[0], arr[1], arr[2]))

    #处理完成事件
    if payload.has_key('finished'):
        finished.set()
```

代码执行后，就能直接获取某个应用的 Bundle ID 和 Documents 目录，再也不用通过各种方式来查找了，具体如下。

```
➜  Frida git:(master) ✗ ./main.py | grep WhatsApp
WhatsApp                                net.whatsapp.WhatsApp
    file:///private/var/mobile/Containers/Data/Application/1DA836F6-B568-47B9-9A06-6F3D
5FCC754B/Documents
```

4. 枚举某个进程加载的模块信息

若要获取模块信息，可以直接 attach，然后调用 enumerate_modules，具体如下。

```
#枚举某个进程所有模块的信息
def listModulesOfProcess(session):
    moduels = session.enumerate_modules()
    moduels.sort(key = lambda item : item.base_address)
    for module in moduels:
        outWrite('%-40s\t%-10s\t%-10s\t%s' % (module.name, hex(module.base_address),
hex(module.size), module.path))
    session.detach()
```

运行结果如下。

```
➜  Frida git:(master) ✗ ./main.py
WhatsApp                                0x10000c000   0xf60000
    /private/var/mobile/Containers/Bundle/Application/8C94AFCD-7965-4BAA-B934-F3B383AF3
5AF/WhatsApp.app/WhatsApp
libswiftCore.dylib                      0x1015a0000   0x248000
    /private/var/mobile/Containers/Bundle/Application/8C94AFCD-7965-4BAA-B934-F3B383AF3
5AF/WhatsApp.app/Frameworks/libswiftCore.dylib
libswiftCoreAudio.dylib                 0x101abc000   0xc000
    /private/var/mobile/Containers/Bundle/Application/8C94AFCD-7965-4BAA-B934-F3B383AF3
5AF/WhatsApp.app/Frameworks/libswiftCoreAudio.dylib
libswiftCoreData.dylib                  0x101af0000   0xc000
    /private/var/mobile/Containers/Bundle/Application/8C94AFCD-7965-4BAA-B934-F3B383AF3
5AF/WhatsApp.app/Frameworks/libswiftCoreData.dylib
......
```

5. 显示界面 UI

之前都是通过 Cycript 或者 Reveal 来查看界面的，但总归有点麻烦。下面通过 Frida 注入 JS 调用 API 来显示界面的结构。加载的 JS 文件内容如下。

```
ObjC.schedule(ObjC.mainQueue, function(){

    const window = ObjC.classes.UIWindow.keyWindow();

    const ui = window.recursiveDescription().toString();

    send({ ui: ui });
});


ObjC.schedule(ObjC.mainQueue, function(){

    const window = ObjC.classes.UIWindow.keyWindow();

    const rootControl = window.rootViewController();

    const control = rootControl['- _printHierarchy']();

    send({ ui: control.toString() });
});

function handleMessage(message) {

  var order = message.substring(0,1);
  var command = '';

  switch(order){
     case 'n':
        command = message.substring(2);

        var view = new ObjC.Object(ptr(command));

        var nextResponder = view.nextResponder();

        nextResponder = new ObjC.Object(ptr(nextResponder));

        var deep = 0;

        var pre = '';

        while(nextResponder){

        pre += '-';
```

```
                send({ ui: pre+'>'+nextResponder.toString()});

                nextResponder = nextResponder.nextResponder();

                nextResponder = new ObjC.Object(ptr(nextResponder));
            }
            break;
        default:
            send({ ui: 'error command' });
    }
    recv(handleMessage);
}

recv(handleMessage);
```

通过这种方式打印界面结构时，不仅可以正常显示中文，还可以直接打印某个对象的响应链，具体如下。

```
➜  Frida git:(master) ✗ ./main.py
   ......
   |   |   |   |   |   | <UIButtonLabel: 0x13c6cad50; frame = (0 4; 34.5 20.5); text =
'归档'; opaque = NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x170290630>>
   |   |   |   |   |   |   | <_UILabelContentLayer: 0x1746247c0> (layer)
   |   |   |   | <UIToolbarTextButton: 0x13c6ad370; frame = (126 0; 69 44); opaque = NO;
layer = <CALayer: 0x17043e560>>
   |   |   |   |   | <_UIToolbarNavigationButton: 0x13c6ad570; frame = (0 8; 69 30);
opaque = NO; layer = <CALayer: 0x17043e4e0>>
   |   |   |   |   |   | <UIButtonLabel: 0x13c5ecb20; frame = (0 4; 68.5 20.5); text =
'全部已读'; opaque = NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x17428b310>>
   |   |   |   |   |   |   | <_UILabelContentLayer: 0x170628c80> (layer)
   |   |   |   | <UIToolbarTextButton: 0x13c6ad800; frame = (277 0; 35 44); opaque = NO;
layer = <CALayer: 0x17043e600>>
   |   |   |   |   | <_UIToolbarNavigationButton: 0x13c6ada00; frame = (0 8; 35 30);
opaque = NO; layer = <CALayer: 0x17043e580>>
   |   |   |   |   |   | <UIButtonLabel: 0x13c69b240; frame = (0 4; 34.5 20.5); text =
'删除'; opaque = NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x170290680>>
   ......

n 0x13c6d1d50
-><UITableViewCellContentView: 0x1703805b0; frame = (0 0; 320 45); gestureRecognizers =
<NSArray: 0x17064b610>; layer = <CALayer: 0x17062c8e0>>
--><WAMessageBubbleTableViewCell: 0x13c6d19e0; baseClass = UITableViewCell; frame = (0 561;
320 45); opaque = NO; autoresize = W; gestureRecognizers = <NSArray: 0x17064bf70>; layer
= <CALayer: 0x17062c8c0>>
```

```
--->&lt;UITableViewWrapperView: 0x13c5d75f0; frame = (0 0; 320 568); gestureRecognizers =
&lt;NSArray: 0x17444c780&gt;; layer = &lt;CALayer: 0x1744395e0&gt;; contentOffset: {0, 0}; contentSize:
{320, 568}&gt;
---->&lt;WAChatSessionTableView: 0x13c96de00; baseClass = UITableView; frame = (0 0; 320 568);
clipsToBounds = YES; gestureRecognizers = &lt;NSArray: 0x17444c540&gt;; layer = &lt;CALayer:
0x174439580&gt;; contentOffset: {0, 334}; contentSize: {320, 856}&gt;
----->&lt;UIView: 0x17419f070; frame = (0 0; 320 568); clipsToBounds = YES; autoresize = W+H;
layer = &lt;CALayer: 0x174439400&gt;&gt;
------>&lt;WAChatSessionViewController: 0x13d0a8600&gt;
------->&lt;UIViewControllerWrapperView: 0x174381a00; frame = (0 0; 320 568); autoresize = W+H;
layer = &lt;CALayer: 0x17443fd60&gt;&gt;
-------->&lt;UINavigationTransitionView: 0x13c69d1b0; frame = (0 0; 320 568); clipsToBounds
= YES; autoresize = W+H; layer = &lt;CALayer: 0x1704390e0&gt;&gt;
--------->&lt;UILayoutContainerView: 0x1743e8100; frame = (0 0; 320 568); clipsToBounds = YES;
autoresize = W+H; gestureRecognizers = &lt;NSArray: 0x17044b1f0&gt;; layer = &lt;CALayer: 0x174433640&gt;&gt;
---------->&lt;ChatNavigationController: 0x13c8be600&gt;
----------->&lt;UIViewControllerWrapperView: 0x17019c150; frame = (0 0; 320 568); autoresize
= W+H; layer = &lt;CALayer: 0x170439fe0&gt;&gt;
------------>&lt;UITransitionView: 0x13c6077d0; frame = (0 0; 320 568); clipsToBounds = YES;
autoresize = W+H; layer = &lt;CALayer: 0x170435920&gt;&gt;
------------->&lt;UILayoutContainerView: 0x1703ecb00; frame = (0 0; 320 568); autoresize = W+H;
layer = &lt;CALayer: 0x1704356c0&gt;&gt;
-------------->&lt;WATabBarController: 0x13c67b910&gt;
--------------->&lt;WARootViewController: 0x13c510bd0&gt;
---------------->&lt;UITransitionView: 0x13c5c6540; frame = (0 0; 320 568); autoresize = W+H;
layer = &lt;CALayer: 0x1744338e0&gt;&gt;
----------------->&lt;WAWindow: 0x13c510290; baseClass = UIWindow; frame = (0 0; 320 568);
gestureRecognizers = &lt;NSArray: 0x1740544c0&gt;; layer = &lt;UIWindowLayer: 0x17403b1a0&gt;&gt;
------------------>&lt;WAApplication: 0x13c505ee0&gt;
```

7.3.3　Frida 的高级使用

7.3.2 节讲解的都是通过 Frida 提供的 Python 接口或者注入 JS 调用 Native 的函数返回来获取一些信息，但是 Frida 真正强大的地方远不止于此。Frida 提供了各种动态函数拦截功能，可以动态地对某个类或某个方法进行替换或监控。先看一下有哪些方便的用法。

因为这些高级功能都是通过 JS 的 API 提供的，所以步骤和前面一样：先 attach 应用，然后注入 JS 代码，接着调用 Frida 提供的 API 实现各种功能，最后通过 Python 的消息通信来展示。先写一段简单的 JS 代码，具体如下。

```
'use strict';
```

```
function findHookMethod(clsname,mtdname){
  if(ObjC.available) {
      for(var className in ObjC.classes) {
          if (ObjC.classes.hasOwnProperty(className)) {
              if(className == clsname) {
                 return ObjC.classes[className][mtdname];
              }
          }
      }
  }
  return;
}

var method = findHookMethod('WAChatDataStore', '- didUpdateMessage:');

Interceptor.attach(method.implementation, {
    onEnter:function (args){
        console.log('-[WAChatDataStore didUpdateMessage:] onEnter...');

        var message = ObjC.Object(args[2]);
        console.log('message is:' + message.toString());

        var text = message['- text']();

        console.log('text is:' + text.toString());
    },
    onLeave:function(retVal){
        console.log('-[WAChatDataStore didUpdateMessage:] onLeave...');
    },
});
```

因为 ObjC.classes 里面保存了当前应用中所有已经注册的类，所以可以直接在这里获取目标的类 WAChatDataStore，然后获取其对应的方法。需要注意的是，这里在填写方法时使用的格式为“+/- method:name:”。

Interceptor 是一个拦截器，可以在函数调用之前和函数调用返回时进行拦截，以获取函数的参数和返回值，从而通过 Frida 对应用中的任何类或者方法进行动态跟踪。通过 Python 加载以上 JS 代码的效果如下。

```
session = device.attach(u'WhatsApp')
script = loadJsFile(session, HOOK_JS)
sys.stdin.read()
```

运行 Python，打开 WhatsApp，使用另一个账号向该账号发送任意消息，就能看到输出的信息了（只需要改动 JS 代码，其他内容不需要修改），具体如下。

```
➜  Frida git:(master) X ./main.py
-[WAChatDataStore didUpdateMessage:] onEnter...
message is:<WAMessage: 0x1456e4a90> (entity: WAMessage; id: 0x174a37720
<x-coredata:///WAMessage/t4D2C8E63-5CBB-49BC-9136-2CE5D1E608B122> ; data: {
    chatSession = nil;
    childMessages =     (
    );
    childMessagesDeliveredCount = 0;
    childMessagesPlayedCount = 0;
    childMessagesReadCount = 0;
    dataItemVersion = nil;
    dataItems =     (
    );
    docID = 0;
    encRetryCount = 0;
    filteredRecipientCount = 0;
    flags = 0;
    fromJID = "8618698580743@s.whatsapp.net";
    groupEventType = 0;
    groupMember = nil;
    isFromMe = 0;
    lastSession = nil;
    mediaItem = "0x17482a120
<x-coredata:///WAMediaItem/t4D2C8E63-5CBB-49BC-9136-2CE5D1E608B123>";
    mediaSectionID = nil;
    messageDate = "2017-09-20 13:02:03 +0000";
    messageErrorStatus = 0;
    messageInfo = nil;
    messageStatus = 6;
    messageType = 0;
    parentMessage = nil;
    phash = nil;
    pushName = AloneMonkey;
    sentDate = nil;
    sort = 0;
    spotlightStatus = "-32768";
    stanzaID = C85D058A5A4B4B758A;
    starred = nil;
    text = MonkeyDev;
    toJID = nil;
})
```

```
text is:MonkeyDev
-[WAChatDataStore didUpdateMessage:] onLeave...
```

以上介绍的是对某个方法的 hook。如果是对某个类里面所有方法的 hook，应该怎么编写代码呢？幸运的是，Frida 提供了 API 解析接口。通过 ApiResolver，我们能够获取符合某个正则表达式的所有方法。假如我们要获取 WAChatDataStore 的所有方法，可以编写如下代码。

```
var resolver = new ApiResolver('objc');
resolver.enumerateMatches('*[WAChatDataStore *]', {
    onMatch: function (match) {
        const mtdname = match['name'];
        const imp = match['address'];
        console.log(mtdname + ":" + imp);
    },
    onComplete: function () {

    }
});
```

运行后获取的结果如下。不仅 WAChatDataStore 的所有方法和地址都被打印出来了，它的子类 WAChatStorage 的方法和地址也被打印出来了。

```
➜  Frida git:(master) ✗ ./main.py
-[WAChatDataStore newManagedObjectContext]:0x100a2b818
-[WAChatDataStore chatSessionForJID:]:0x100a2cc9c
-[WAChatDataStore performSave:]:0x100436388
-[WAChatDataStore messageWithMessageID:inContext:]:0x100a2dc04
-[WAChatDataStore addChildMessagesIfNeededForParentMessage:]:0x100436664
-[WAChatDataStore messageWithStanzaID:isFromMe:inChatSessionWithJID:]:0x100a2d96c
-[WAChatDataStore storeModifiedMessage:notify:]:0x100436604
-[WAChatDataStore didChangeDeliveryStatusOfMessage:]:0x100a2c0bc
-[WAChatDataStore willDeleteMessages:inChatSession:fromWebClient:]:0x100a2c2d8
-[WAChatDataStore didDeleteMessages:inChatSession:fromWebClient:]:0x100a2c64c
......
-[WAChatStorage isChatSessionForExistingUser:]:0x100113bac
-[WAChatStorage receiveError:forMessage:]:0x100110da4
-[WAChatStorage sendMessage:transaction:]:0x100111e50
-[WAChatStorage prepareMessageSend:]:0x100111af4
-[WAChatStorage commitTransactions:sendMessageGroups:]:0x1001120d0
-[WAChatStorage requestLocationForMessage:]:0x10011bb8c
-[WAChatStorage
removeInvalidMentionsFromMessage:forForwardingToChatSession:]:0x100113ca4
-[WAChatStorage copyMessage:toChatSession:fullCopy:]:0x100112690
```

```
-[WAChatStorage prepareMulticastMessage:forRemainingSessions:]:0x10011709c
......
```

获取了所有的方法地址后，就可以直接使用拦截器（Interceptor）对每个地址进行拦截，然后打印参数和返回值了，具体如下。

```
var resolver = new ApiResolver('objc');
resolver.enumerateMatches('*[WAChatDataStore *]', {
    onMatch: function (match) {
        const mtdname = match['name'];
        const imp = match['address'];
        // console.log(mtdname + ":" + imp);
        var targetName = mtdname;

        if(targetName != ".cxx_destruct"){
          try{
                Interceptor.attach(imp, {

                  onEnter: function (args) {

                      enterOutPut(targetName, args);

                  },
                  onLeave: function (retval) {
                      leaveOutPut(targetName, retval);
                  },
                });
              }catch(e){
              console.log('hook '+mtdname+' error:'+e);
              }
        }
    },
    onComplete: function () {
        console.log('hook '+clsname+' finished!');
    }
});
```

在拦截器中对调用的方法、传入的参数、返回的参数进行处理并打印调用层级关系，这样，在逆向分析时看一下调用函数和参数就一目了然了——既不用注入动态库，也不用重新安装应用，真是方便！

对 WAChatDataStore 的所有方法进行监控的输出内容如下，从中我们可以了解程序的大致流程。

```
----------->(1)onEnter(-[WAChatStorage
newManagedObjectContext])-------------------------------------------------------
- [<WAChatStorage: 0x14c604870> newManagedObjectContext]
--------------------->(2)onEnter(-[WAChatStorage
newManagedObjectContextWithConcurrencyType:])-------------------------
- [<WAChatStorage: 0x14c604870> newManagedObjectContextWithConcurrencyType:0]
retVal:<NSManagedObjectContext: 0x1703e7100>
--------------------->(2)onLeave(-[WAChatStorage
newManagedObjectContextWithConcurrencyType:])-------------------------
retVal:<NSManagedObjectContext: 0x1703e7100>
----------->(1)onLeave(-[WAChatStorage
newManagedObjectContext])-------------------------------------------------------
----------->(1)onEnter(-[WAChatDataStore
messagesWithStanzaIDs:isFromMe:prefetchedKeyPaths:inContext:])----------------
- [<WAChatStorage: 0x14c604870> messagesWithStanzaIDs:(
    4968A588F7863A0D09
) isFromMe:0 prefetchedKeyPaths:(
    groupMember
) inContext:<NSManagedObjectContext: 0x1703e7100>]
retVal:(
)
----------->(1)onLeave(-[WAChatDataStore
messagesWithStanzaIDs:isFromMe:prefetchedKeyPaths:inContext:])----------------
----------->(1)onEnter(-[WAChatStorage
managedObjectContext])----------------------------------------------------------
- [<WAChatStorage: 0x14c604870> managedObjectContext]
retVal:<NSManagedObjectContext: 0x1701f3a00>
----------->(1)onLeave(-[WAChatStorage
managedObjectContext])----------------------------------------------------------
----------->(1)onEnter(-[WAChatStorage
managedObjectContext])----------------------------------------------------------
- [<WAChatStorage: 0x14c604870> managedObjectContext]
retVal:<NSManagedObjectContext: 0x1701f3a00>
----------->(1)onLeave(-[WAChatStorage
managedObjectContext])----------------------------------------------------------
----------->(1)onEnter(-[WAChatDataStore
messagesWithStanzaIDs:isFromMe:prefetchedKeyPaths:inContext:])----------------
- [<WAChatStorage: 0x14c604870> messagesWithStanzaIDs:(
    4968A588F7863A0D09
) isFromMe:0 prefetchedKeyPaths:(
    groupMember
) inContext:<NSManagedObjectContext: 0x1701f3a00>]
retVal:(
)
```

```
---------->(1)onLeave(-[WAChatDataStore
messagesWithStanzaIDs:isFromMe:prefetchedKeyPaths:inContext:])----------------
---------->(1)onEnter(-[WAChatStorage
messagePlaceholderForStanza:])-------------------------------------------------
- [<WAChatStorage: 0x14c604870> messagePlaceholderForStanza:[message
from='8618698580743@s.whatsapp.net' t='1505919175' notify='AloneMonkey'
id='4968A588F7863A0D09' type='text' [enc v='2' type='pkmsg' {166b} ] ] ]
--------------------->(2)onEnter(-[WAChatStorage
managedObjectContext])---------------------------------------------
- [<WAChatStorage: 0x14c604870> managedObjectContext]
retVal:<NSManagedObjectContext: 0x1701f3a00>
--------------------->(2)onLeave(-[WAChatStorage
managedObjectContext])---------------------------------------------
retVal:<WAMessage: 0x14d8642a0> (entity: WAMessage; id: 0x170820780
<x-coredata:///WAMessage/t46C3B1E9-008A-4C71-A31A-FE1BC14243885> ; data: {
    chatSession = nil;
    childMessages =     (
    );
    childMessagesDeliveredCount = 0;
    childMessagesPlayedCount = 0;
    childMessagesReadCount = 0;
    dataItemVersion = nil;
    dataItems =     (
    );
    docID = nil;
    encRetryCount = 0;
    filteredRecipientCount = 0;
    flags = 0;
    fromJID = "8618698580743@s.whatsapp.net";
    groupEventType = 0;
    groupMember = nil;
    isFromMe = 0;
    lastSession = nil;
    mediaItem = nil;
    mediaSectionID = nil;
    messageDate = "2017-09-20 14:52:55 +0000";
    messageErrorStatus = 0;
    messageInfo = nil;
    messageStatus = 0;
    messageType = 12;
    parentMessage = nil;
    phash = nil;
    pushName = AloneMonkey;
    sentDate = nil;
```

```
    sort = 0;
    spotlightStatus = "-32768";
    stanzaID = 4968A588F7863A0D09;
    starred = nil;
    text = nil;
    toJID = nil;
})
```

采用这种方式，就可以直接通过配置文件或 Web 界面来控制当前需要监控的类或方法了。

7.3.4 小结

在本节中，笔者简单介绍了 Frida 的一些常规用法。通过脚本把逆向分析中一些常用的步骤整合在一起，就可以更新和获取对应的值。

当然，Frida 提供的 JS 接口可以做更多和更加底层的事情，读者可以阅读官方文档（https://www.frida.re/docs/javascript-api/），自行尝试使用其中的 API，以实现更加强大的功能。网上有一些基于 Frida 的开源项目（例如 https://github.com/sensepost/objection），使用起来也是比较方便的（例如一键执行 SSL 证书验证的脚本等）。此外，Frida 的核心库非常值得我们学习，例如负责底层 hook 的 frida-gum（https://github.com/frida/frida-gum）。

第8章 安全保护

在前面对应用的各种分析中，我们几乎都是把一个应用当成自己的应用去修改。看一眼自己开发的应用，有没有一种心慌慌的感觉？如果我们的应用没有任何保护措施，也会面临同样的局面——被分析、被破解、被重打包。随着分析难度逐渐降低，该如何保护自己的应用不被分析和破解呢？这是本章要着重探讨的内容。

根据前面介绍的各种方法的切入点来制定不同的策略，可以提高分析者的分析难度。本章将要介绍的加固保护方案如下。

- 数据加密：静态字符串、本地存储及网络传输的加密。
- 静态混淆：类名、方法名、属性的混淆。
- 动态保护：反调试、注入检测、hook 检测、越狱检测、签名检测等。
- 代码混淆：将代码分块、扁平化、增加干扰代码，以提高分析者的分析难度。

8.1 数据加密

移动端的数据加密主要表现在本地数据存储加密、静态字符串加密、网络传输数据加密等方面。安全意识薄弱的开发者通常不会对用户的敏感信息进行加密保存，而是直接通过明文保存在本地或者通过网络传输。这样，分析者就可以直接通过本地文件进行静态分析，或者通过发起第三方中间人攻击来修改数据了。

说到数据加密，我们先了解一下常用的加密算法，如表 8-1 所示。

表 8-1 常用的加密算法

算 法	简 介
BASE64	一种基于 64 个可打印字符表示二进制数据的方法，主要是为了将二进制数据变成可见字符以便传输，可以解密
MD5	将数据运算变为另一固定长度值，用于数据校验，不可逆

续表

算　法	简　介
DES	一种对称加密算法，用于对二进制数据进行加密，明文、密文和密钥的分组长度都是 64 位
AES	密码学中的高级加密标准，本质上是一种对称分组密码体制，用来代替 DES
RSA	一种非对称加密算法，存在一个公钥和一个私钥，分别用于加密和解密，速度比对称加密慢很多

根据加密结果是否可以解密，算法可以分为可逆加密和不可逆加密。对于可逆加密，又可以根据密钥的对称性分为对称加密和非对称加密。上面提到的 DES 和 AES 是对称加密算法，加密和解密时使用同一密钥。RSA 是非对称加密算法，存在一个公钥和一个私钥。MD5 是不可逆的加密算法，一般用于数据校验。

在 iOS 平台上，苹果已经封装了这些算法，不需要我们自己去实现，我们只需要在使用时选择正确的加密算法。下面介绍在实际开发中需要注意的数据加密方式。

8.1.1 本地存储加密

本地存储的方式有 NSUserDefaults、文件、数据库等。不管以何种方式进行存储，都不能直接将敏感信息以明文的形式存储在本地。所以，在存储数据时，可以使用哈希算法、对称加密算法（DES/AES）进行加密存储，或者使用系统 keychain 进行存储。但是，在实现加密算法时也不能直接将 key 以明文的形式写在代码中（可以直接通过静态分析得出）。在 8.1.3 节中会讲解如何对程序中的所有字符串进行加密处理。

8.1.2 网络传输加密

敏感信息肯定不能以明文的形式在网络中传输，建议使用 HTTPS + 本地证书校验的方式，在传输过程中可以使用 RSA+AES 进行加密传输，流程如下。

①客户端随机生成 AES 的 key。

②使用 RSA 公钥加密算法加密随机的 key。

③使用随机的 key 对数据进行 AES 加密。

④将 RSA 加密后的 key 和 AES 加密后的数据发送给数据器。

⑤给发送的数据加上时间戳，以防止重放攻击。

整个流程如图 8-1 所示。

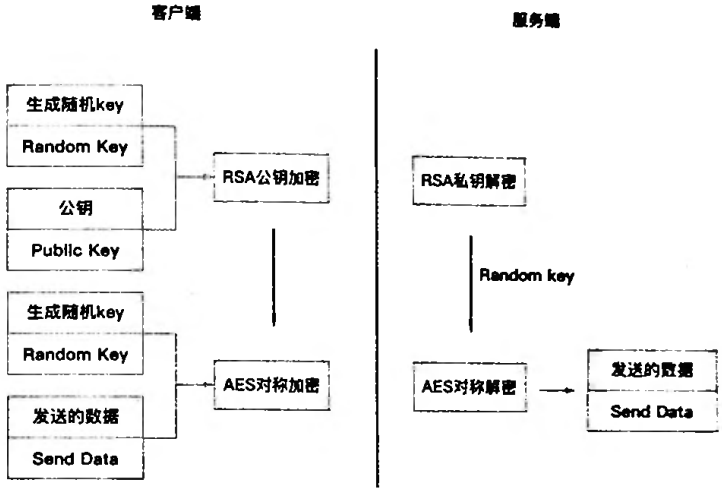


图 8-1　网络数据传输 RSA+AES 加密方案

如果使用的是 HTTPS，可以自己处理证书校验流程。如果使用的是 NSURLSession，可以在 URLSession:didReceiveChallenge:completionHandler: 中获取证书信息进行验证。如果使用的是 NSURLConnection，可以在 connection:willSendRequestForAuthenticationChallenge: 中处理，或者使用 AFNetworking 网络库设置验证策略，具体如下。

```
NSString *certFilePath = [[NSBundle mainBundle] pathForResource:@"server" ofType:@"der"];
NSData *certData = [NSData dataWithContentsOfFile:certFilePath];
NSSet *certSet = [NSSet setWithObject:certData];
AFSecurityPolicy *policy = [AFSecurityPolicy
policyWithPinningMode:AFSSLPinningModeCertificate withPinnedCertificates:certSet];
```

所以，如果在抓包时发现请求失败，可能就是本地证书校验造成的。在 objection 中也对各种校验进行了 hook，以绕过证书校验（https://github.com/sensepost/objection/blob/master/objection/hooks/ios/pinning/disable.js）。

另外，笔者以前在分析某应用的网络传输时发现了一个简单的方法（现在协议已经修改了），在这里和读者分享一下。其思路就是在传输 buffer 时对每个传输的字符串进行自定义加密传输，简单的代码如下。

```
AMBuffer.h
@interface AMBuffer : NSObject

+(AMBuffer*)defaultForAPI;
```

```
-(void)setInt32:(int)int32;

-(void)setNSString:(NSString *)str;

-(void)setLengthWithFirst;

-(const char*)getBody;

-(unsigned long)getLength;

@end

AMBuffer.m
@interface AMBuffer ()

@end

@implementation AMBuffer{
    char* _body;
    unsigned long _bufferLen;          //buffer 的总长度
    unsigned long _len;                //当前数据的长度
    unsigned long _wseek;              //写指针偏移
    unsigned long _rseek;              //读指针偏移
}

+(AMBuffer*)defaultForAPI{
    AMBuffer* buffer = [[[self class] alloc] init];
    if(buffer){
        [buffer setInt32:0];
    }
    return buffer;
}

- (instancetype)init
{
    self = [super init];
    if (self) {
        _bufferLen = 2048;
        _body = calloc(2048, 1);
    }
    return self;
}

-(void)addLen:(unsigned long)len{
```

```
    _len += len;
    while(_len >= _bufferLen){
        _bufferLen += 2048;
        _body = realloc(_body, _bufferLen);
    }
}

#pragma mark - Setter

-(void)setInt32:(int)int32{
    [self write:&int32 len:4];
}

-(void)setNSString:(NSString *)str{
    if(str && str.length){
        [self write:[str UTF8String] len:strlen([str UTF8String])+1];
    }
}

-(void)setLengthWithFirst{
    [self setLengthWithSeek:0];
}

-(void)setLengthWithSeek:(unsigned long)seek{
    char* len = (char*)&_len;
    for (int i = 0; i < 4; i++) {
        _body[seek+i] = len[i];
    }
}

#pragma mark - Getter

-(const char *)getBody{
    return _body;
}

-(unsigned long)getLength{
    return _len;
}

#pragma mark - read & write

//除了前面 4 字节是长度，其他字节为 xor 0x08
-(void)write:(const char *)write len:(unsigned long)len{
```

```
    [self addLen:len];
    while (len) {
        _body[_wseek] = *write ^ 0x08;
        _wseek++;
        write++;
        len--;
    }
}

-(void)read:(char *)read len:(unsigned long)len{
    if(_rseek + len > _len){
        NSLog(@"read len over");
    }else{
        while (len) {
            *read = _body[_rseek];
            _rseek++;
            read++;
            len--;
        }
    }
}
@end
```

传输时调用的方法如下。这里需要先静态分析 buffer 是如何组装的，再模拟请求（解密这些数据需要服务端的配合）。传输数据时推荐使用 Google 开源的 protobuf（https://github.com/google/protobuf）。

```
NSString* username = @"username";
NSString* password = @"password";
NSString * URLString = @"http://xxx";
NSURLSessionConfiguration *sessionConfiguration = [NSURLSessionConfiguration
defaultSessionConfiguration];
NSURLSession *session = [NSURLSession sessionWithConfiguration:sessionConfiguration];
NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:[NSURL
URLWithString:URLString]];

//自定义格式
AMBuffer* buffer = [AMBuffer defaultForAPI];
[buffer setNSString:@"username"];
[buffer setNSString:username];
[buffer setNSString:@"password"];
[buffer setNSString:password];
[buffer setLengthWithFirst];
```

```
NSData* data = [NSData dataWithBytes:[buffer getBody] length:[buffer getLength]];

request.HTTPBody = data;
request.HTTPMethod = @"POST";
```

8.1.3　字符串加密

因为程序中使用的 key 或者一些关键字符串是不能以明文的形式写在源代码中的（在静态分析时能直接看到），所以，需要采用一些方法对文件中使用的所有静态字符串进行加密处理，以防止静态分析。如何获取源文件中的所有字符串呢？我们马上就能想到的方法是遍历源文件中的字符串。但是人工阅读和分析源代码的工作量太大，如果能让编译器帮助分析就好了。查看与 Clang 相关的接口和资料，发现 libclang 可以帮助完成这件事情。libclang 提供了 Clang 的 C 接口，用于将源代码转换成抽象语法树，并对语法树进行遍历和分析，自然也可以帮助我们分析源代码中存在的字符串。相关接口可以访问 https://clang.llvm.org/doxygen/group__CINDEX.html 了解。

为了获取 libclang 中提供的 C 接口，要先使用 dlopen 将其加载到内存中并获取需要使用的函数指针。需要从 Clang 源代码中导入一些必需的头文件，具体如下。

```
CXIndex  (*myclang_createIndex)(int excludeDeclarationsFromPCH, int displayDiagnostics);

CXTranslationUnit (*myclang_createTranslationUnitFromSourceFile)(
                                        CXIndex CIdx,
                                        const char *source_filename,
                                        int num_clang_command_line_args,
                                        const char * const *clang_command_line_args,
                                        unsigned num_unsaved_files,
                                        struct CXUnsavedFile *unsaved_files);

void initlibfunclist(void *handle){
    myclang_createIndex = dlsym(handle, "clang_createIndex");
    myclang_createTranslationUnitFromSourceFile = dlsym(handle,
"clang_createTranslationUnitFromSourceFile");
    ......
}

void *hand =
dlopen("/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/
usr/lib/libclang.dylib",RTLD_LAZY);
```

```
initlibfunclist(hand);
```

准备工作完成后，就可以对源文件进行语法树遍历了。找到所有 C 类型的字符串和 OC 类型的字符串，printVisitor 函数用于处理每次遍历语法树的结果，具体如下。

```
void showString(int start, int end){
    NSString *conent = [[NSString alloc] initWithContentsOfFile:@FILENAME
encoding:NSASCIIStringEncoding error:nil];
    NSString *res = [conent substringWithRange:NSMakeRange(start, end-start)];

    printf("----------遍历的内容----------\n %s\n----------\n",[res
UTF8String]);
}

enum CXChildVisitResult printVisitor(CXCursor cursor, CXCursor parent, CXClientData
client_data) {
    CXFile file;
    unsigned int line,column,start,end;
    CXSourceRange range = myclang_getCursorExtent(cursor);
    CXSourceLocation startLocation = myclang_getRangeStart(range);
    CXSourceLocation endLocation = myclang_getRangeEnd(range);

    myclang_getExpansionLocation(startLocation, &file, &line, &column, &start);
    myclang_getExpansionLocation(endLocation, &file, &line, &column, &end);

    //获取文件名
    CXString cxname = myclang_getFileName(file);
    const char* filename = myclang_getCString(cxname);
    myclang_disposeString(cxname);

    //过滤文件
    if(filename == NULL || strcmp(filename, FILENAME)){
        return CXChildVisit_Continue;
    }

    showString(start,end);

    //获取 displayName
    CXString cxdisname =  myclang_getCursorDisplayName(cursor);
    const char* disname = myclang_getCString(cxdisname);
    printf("disname is => %s\n",disname);
    myclang_disposeString(cxdisname);

    //获取类型
```

```
    enum CXCursorKind kind = myclang_getCursorKind(cursor);
    CXString cxcurkind = myclang_getCursorKindSpelling(kind);
    const char* ccurkind = myclang_getCString(cxcurkind);
    printf("ccurkind is =>%s\n",ccurkind);
    myclang_disposeString(cxcurkind);

    //判断类型
    enum CXLinkageKind cxlinkkind = myclang_getCursorLinkage(cursor);
    if(!strcmp(ccurkind, "VarDecl")&&(cxlinkkind == CXLinkage_External)){
        return CXChildVisit_Continue;
    }

    if(!strcmp(ccurkind, "ObjCStringLiteral")){
        printf("OC 类型字符串\n");
        return CXChildVisit_Continue;
    }

    if(!strcmp(ccurkind, "StringLiteral")){
        printf("C 类型字符串\n");
        return CXChildVisit_Continue;
    }

    const char* disname2 = myclang_getCString(cxdisname);

    printf("继续遍历孩子节点%s\n",disname2);

    return CXChildVisit_Recurse;
}

//excludeDeclsFromPCH=1, displayDiagnostics=1
CXIndex cxindex = myclang_createIndex(1, 1);

TU = myclang_createTranslationUnitFromSourceFile(cxindex, FILENAME, 0, nil, 0, 0);
myclang_visitChildren(myclang_getTranslationUnitCursor(TU), printVisitor, 0);

myclang_disposeTranslationUnit(TU);
myclang_disposeIndex(cxindex);
```

printVisitor 的返回结果如下。

- CXChildVisit_Break：直接结束遍历。
- CXChildVisit_Continue：跳过当前节点的遍历，不再遍历它的孩子节点。
- CXChildVisit_Recurse：继续遍历它的孩子节点。

下面通过遍历文件的内容来查看结果，具体如下。

```
#import "TargetFile.h"

@implementation TargetFile

-(NSString*)method{
    char* cstr = "local char str";

    NSString *nsstr = @"local nsstring \n";

    return @"return c string";
}

@end
```

对该文件进行遍历，结果如下。

```
----------遍历的内容----------
 #import "TargetFile.h"
------------------------------
disname is => TargetFile.h
ccurkind is =>inclusion directive
继续遍历孩子节点 TargetFile.h
----------遍历的内容----------
 @implementation TargetFile

-(NSString*)method{
    char* cstr = "local char str";

    NSString *nsstr = @"local nsstring \n";

    return @"return c string";
}

@
------------------------------
disname is => TargetFile
ccurkind is =>ObjCImplementationDecl
继续遍历孩子节点 TargetFile
----------遍历的内容----------
 -(NSString*)method{
    char* cstr = "local char str";
```

```
    NSString *nsstr = @"local nsstring \n";

    return @"return c string";
}
------------------------------
disname is => method
ccurkind is =>ObjCInstanceMethodDecl
继续遍历孩子节点 method
----------遍历的内容----------
 NSString
------------------------------
disname is => NSString
ccurkind is =>ObjCClassRef
继续遍历孩子节点 NSString
----------遍历的内容----------
 {
    char* cstr = "local char str";

    NSString *nsstr = @"local nsstring \n";

    return @"return c string";
}
------------------------------
disname is =>
ccurkind is =>CompoundStmt
继续遍历孩子节点
----------遍历的内容----------
 char* cstr = "local char str";
------------------------------
disname is =>
ccurkind is =>DeclStmt
继续遍历孩子节点
----------遍历的内容----------
 char* cstr = "local char str"
------------------------------
disname is => cstr
ccurkind is =>VarDecl
继续遍历孩子节点 cstr
----------遍历的内容----------
 "local char str"
------------------------------
disname is =>
ccurkind is =>UnexposedExpr
继续遍历孩子节点
```

```
----------遍历的内容----------
 "local char str"
------------------------------
disname is => "local char str"
ccurkind is =>StringLiteral
C 类型字符串
----------遍历的内容----------
 NSString *nsstr = @"local nsstring \n";
------------------------------
disname is =>
ccurkind is =>DeclStmt
继续遍历孩子节点
----------遍历的内容----------
 NSString *nsstr = @"local nsstring \n"
------------------------------
disname is => nsstr
ccurkind is =>VarDecl
继续遍历孩子节点 nsstr
----------遍历的内容----------
 NSString
------------------------------
disname is => NSString
ccurkind is =>ObjCClassRef
继续遍历孩子节点 NSString
----------遍历的内容----------
 @"local nsstring \n"
------------------------------
disname is => "local nsstring \n"
ccurkind is =>ObjCStringLiteral
OC 类型的字符串
----------遍历的内容----------
 return @"return c string"
------------------------------
disname is =>
ccurkind is =>ReturnStmt
继续遍历孩子节点
----------遍历的内容----------
 @"return c string"
------------------------------
disname is => "return c string"
ccurkind is =>ObjCStringLiteral
OC 类型的字符串
```

通过对一个源文件语法树的遍历，就可以分析出其中使用的所有 C 和 OC 类型的字符串，进

而对其中的字符串进行加密处理了。不过，算法也不能太复杂，否则会影响效率。

此外，要考虑多线程同步的问题。例如，对以上文件内容中的字符串进行简单的加密，代码如下。这里只给出加密后的状态，具体的加密过程是通过 libclang 分析语法树自动完成的，无须人工干预。

```
#define AMDecodeCString         AMfZoMnDsWhCdAtLv
#define AMDecodeOcString        AMwXsRvKuDdUkPbKcI
#define AMEncodedString         AMeBfHrFiWsKMiHscC

#ifdef __cplusplus
extern "C" {
#endif
#import <pthread.h>

    typedef struct AMEncodedString{
        char* origstr;
        int size;
        pthread_mutex_t mutex;
    }AMEncodedString;

    static inline char * AMDecodeCString(AMEncodedString *str) {

        pthread_mutex_lock(&str->mutex);

        char seed = str->origstr[str->size-1];

        int j = 0;

        do{
            str->origstr[j] ^= seed;
            j++;
        }while(j < str->size);

        pthread_mutex_unlock(&str->mutex);

        return str->origstr;
    }
#ifdef __OBJC__
#import <Foundation/Foundation.h>
    static inline NSString * AMDecodeOCString(AMEncodedString *str) {

        pthread_mutex_lock(&str->mutex);
```

```
        char seed = str->origstr[str->size-1];

        int j = 0;

        do{
            str->origstr[j] ^= seed;
            j++;
        }while(j < str->size);

        pthread_mutex_unlock(&str->mutex);

        return [[NSString alloc] initWithBytesNoCopy:str->origstr length:str->size-1
encoding:NSUTF8StringEncoding freeWhenDone:0];
    }
#endif

#ifdef __cplusplus
}
#endif

static unsigned char _59C8BC15547095661572[] = { 0xA5, 0xB2, 0xA3, 0xA2, 0xA5, 0xB9, 0xF7,
0xB4, 0xF7, 0xA4, 0xA3, 0xA5, 0xBE, 0xB9, 0xB0, 0xD7 };
static AMEncodedString _59C8BC15547095 = { (char *)_59C8BC15547095661572,
sizeof(_59C8BC15547095661572) };

static unsigned char _59C8BC15121264576724[] = { 0x08, 0x0B, 0x07, 0x05, 0x08, 0x44, 0x0A,
0x17, 0x17, 0x10, 0x16, 0x0D, 0x0A, 0x03, 0x44, 0x6E, 0x64 };
static AMEncodedString _59C8BC15121264 = { (char *)_59C8BC15121264576724,
sizeof(_59C8BC15121264576724) };

static unsigned char _59C8BC15888785777747[] = { 0x08, 0x0B, 0x07, 0x05, 0x08, 0x44, 0x07,
0x0C, 0x05, 0x16, 0x44, 0x17, 0x10, 0x16, 0x64 };
static AMEncodedString _59C8BC15888785 = { (char *)_59C8BC15888785777747,
sizeof(_59C8BC15888785777747) };

#import "TargetFile.h"

@implementation TargetFile

-(NSString*)method{
    char* cstr = /*"local char str"*/AMDecodeCString(&_59C8BC15888785);

    NSString *nsstr = /*@"local nsstring \n"*/AMDecodeOCString(&_59C8BC15121264);

    return /*@"return c string"*/AMDecodeOCString(&_59C8BC15547095);
```

```
}

@end
```

加密后进行静态分析和的结果和加密前进行静态分析的结果对比如图 8-2 所示——至少加密后无法直接通过静态分析得到字符串。当然，还是可以通过动态调试或者编写 IDA 脚本的方式来还原算法，这里介绍的方法只是增加了分析的难度。

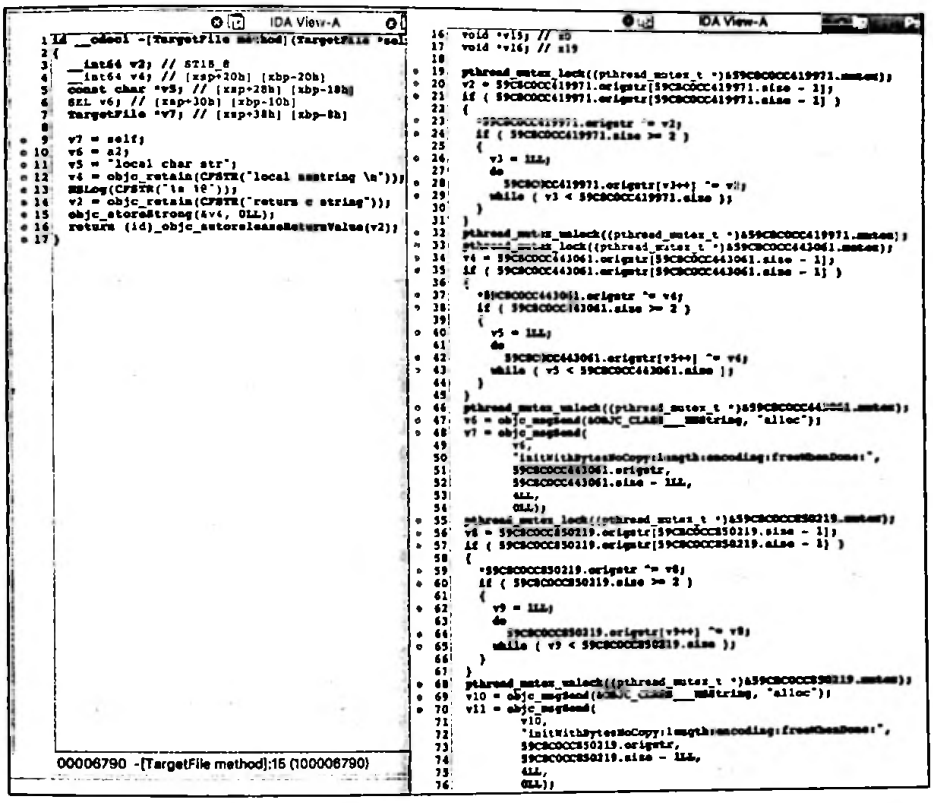


图 8-2　字符串加密前和加密后的效果对比

8.2　静态混淆

除了对字符串的静态加密，还有一个特别重要的方面，就是程序中存在的类名和方法名。在逆向分析中，只要通过 class-dump 获取程序中的所有类、定义的方法和属性，就能很快从名字中猜到某个方法的作用是什么，所以，类名和方法名的混淆就特别重要了。下面介绍两种可以混淆类名、方法名和属性的方法。

8.2.1 宏定义

最简单的方法就是在 release 时通过宏定义将现有的类名和方法名替换为一些无意义的字符串。这种全局定义一般都会写在 pch 文件中，但因为从 Xcode 6 开始默认去掉了 pch 文件，所以需要我们自己添加这个文件。单击“New File”选项，新建一个 pch 文件，在 Build Settings 界面设置使用预处理文件并指定文件路径，如图 8-3 所示。

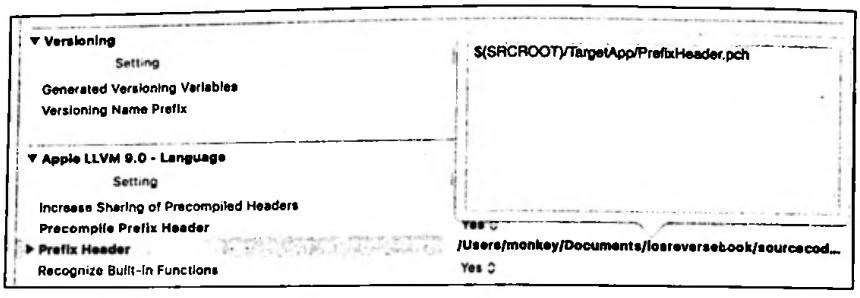


图 8-3 在 Xcode 中指定预处理文件

然后，在预处理文件中将类名和方法名定义成无意义的字符串，代码如下。

```
#ifndef PrefixHeader_pch
#define PrefixHeader_pch

#define CustomClass SJ3fbier3csbrF
#define customMethod brfuFsjvntD36YSvcf

#endif /* PrefixHeader_pch */
```

对生成的文件，再通过 class-dump 获取头文件时，就会得到下面这种无意义的符号了。

```
@interface SJ3fbier3csbrF : NSObject
{
}

- (void)brfuFsjvntD36YSvcf;

@end
```

GitHub 上有一个开源项目可以帮助我们自动生成这种宏定义（https://github.com/Polidea/ios-class-guard）。该项目是开源 class-dump 的 fork，它的原理就是把使用的所有系统库符号 dump 下来并过滤掉，只生成用户需要混淆的符号。

克隆代码并编译，运行 ios-class-guard，其参数如下。

```
➜  Debug git:(master) ./ios-class-guard
ios-class-guard 0.8 (64 bit) [based on class-dump 3.5] (Debug version compiled Sep 24 2017
15:54:55)
Usage: ios-class-guard [options] <mach-o-file>

  where options are:
       -F <class>        specify class filter for symbols obfuscator (also protocol))
       -i <symbol>       ignore obfuscation of specific symbol)
       --arch <arch>     choose a specific architecture from a universal binary (ppc, ppc64,
i386, x86_64, armv6, armv7, armv7s, arm64)
       --list-arches     list the arches in the file, then exit
       --sdk-ios         specify iOS SDK version (will look for
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/i
PhoneOS<version>.sdk
                         or
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS<version>.sdk)
       --sdk-mac         specify Mac OS X version (will look for
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/Mac
OSX<version>.sdk
                         or /Developer/SDKs/MacOSX<version>.sdk)
       --sdk-root        specify the full SDK root path (or use --sdk-ios/--sdk-mac for a
shortcut)
       -X <directory>    base directory for XIB, storyboards (will be searched recursively)
       -P <path>         path to project.pbxproj of Pods project (located inside
Pods.xcodeproj)
       -O <path>         path to file where obfuscated symbols are written
       -m <path>         path to symbol file map (default value symbols.json)
       -c <path>         path to symbolicated crash dump
       --dsym <path>     path to dSym file to translate
       --dsym-out <path> path to dSym file to translate
```

- -F：需要过滤的类或协议的符号，包括类里面的方法。
- -i：需要过滤的特定符号。
- --arch：指定 dump 的架构。
- --list-arches：枚举架构信息。
- --sdk-ios/--sdk-mac：指定使用的 SDK 的版本。因为该工具会在系统的 SDK 目录下找到系统的模块，然后 class-dump 符号，但是在 Xcode 8 中该目录下只有 tbd 文件，在模拟器的 SDK 中才有可执行文件，而在 Xcode 9 中已经全部换成了 tbd 文件，所以，我们只能自行指定路径为 Xcode 8 模拟器 SDK 的路径。

- --sdk-root：指定使用的 SDK 目录。
- -X：执行项目路径，用于混淆 XIB 和 storyboards 中的类名和方法。
- -P：如果使用了 Pod，就需要指定 Pods/Pods.xcodeproj/project.pbxproj 的路径用于修改配置，以便混淆其中的符号。
- -O：指定生成的头文件的位置。
- -m：指定生成 json 映射文件的位置。
- -c：恢复在崩溃堆栈中混淆的符号。
- --dsym/--dsym-out：在符号文件中混淆的符号，用于还原崩溃堆栈中的符号。

由于指定模拟器后的架构和分析文件的架构不一样，需要修改 Source/CDFatFile.m 文件的 131 行，使其返回 self.arches[0]，而不是 nil。改完后重新编译，以一个简单的 Demo 工程为例，生成如下混淆的预处理文件。

```
➜  TargetApp.app git:(master) ✗ ./ios-class-guard TargetApp --sdk-root
/Applications/Xcode8.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Develope
r/SDKs/iPhoneSimulator.sdk -X
/Users/monkey/Documents/iosreversebook/sourcecode/chapter-8/TargetApp -O Header.h
2017-09-24 17:10:56.800 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, _generationCount
2017-09-24 17:10:56.800 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, shmemEntry
2017-09-24 17:10:56.800 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, lastKnownShmemState
2017-09-24 17:10:56.801 ios-class-guard[70940:19128276] Warning: Parsing method types
failed, shmem
2017-09-24 17:10:56.801 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, _shmem
......
2017-09-24 17:10:58.097 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, fMandatoryAttributes
2017-09-24 17:10:58.311 ios-class-guard[70940:19128276] Warning: Parsing method types
failed, scale
2017-09-24 17:10:58.311 ios-class-guard[70940:19128276] Warning: Parsing method types
failed, position
2017-09-24 17:10:58.311 ios-class-guard[70940:19128276] Warning: Parsing method types
failed, direction
2017-09-24 17:10:58.312 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, _position
2017-09-24 17:10:58.312 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, _scale
```

```
2017-09-24 17:10:58.313 ios-class-guard[70940:19128276] Warning: Parsing method types
failed, direction
2017-09-24 17:10:58.314 ios-class-guard[70940:19128276] Warning: Parsing method types
failed, sampleFieldsAt:
2017-09-24 17:10:58.328 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, _halfExtent
2017-09-24 17:10:58.329 ios-class-guard[70940:19128276] Warning: Parsing instance variable
type failed, _halfExtent2
2017-09-24 17:11:00.193 ios-class-guard[70940:19128276] Processing external symbols from
PhysicsKit...
2017-09-24 17:11:00.196 ios-class-guard[70940:19128276] Processing external symbols from
MediaAccessibility...
2017-09-24 17:11:00.196 ios-class-guard[70940:19128276] Processing external symbols from
Accessibility...
2017-09-24 17:11:00.197 ios-class-guard[70940:19128276] Processing external symbols from
ProofReader...
2017-09-24 17:11:00.199 ios-class-guard[70940:19128276] Processing external symbols from
TextInput...
2017-09-24 17:11:00.210 ios-class-guard[70940:19128276] Processing external symbols from
SpringBoardServices...
......
2017-09-24 17:11:00.653 ios-class-guard[70940:19128276] Processing external symbols from
System...
2017-09-24 17:11:00.653 ios-class-guard[70940:19128276] Processing external symbols from
bz2...
2017-09-24 17:11:00.653 ios-class-guard[70940:19128276] Processing external symbols from
archive...
2017-09-24 17:11:00.653 ios-class-guard[70940:19128276] Processing external symbols from
Foundation...
2017-09-24 17:11:00.692 ios-class-guard[70940:19128276] Processing internal symbols...
2017-09-24 17:11:00.692 ios-class-guard[70940:19128276] Ignoring @protocol NSObject
2017-09-24 17:11:00.693 ios-class-guard[70940:19128276] Ignoring @protocol
UIApplicationDelegate
2017-09-24 17:11:00.693 ios-class-guard[70940:19128276] Obfuscating @class ViewController
2017-09-24 17:11:00.693 ios-class-guard[70940:19128276] Obfuscating @class CustomClass
2017-09-24 17:11:00.693 ios-class-guard[70940:19128276] Obfuscating @class AppDelegate
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] Generating symbol table...
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] Protocols = 0
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] Classes = 3
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] Categories = 0
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] Methods = 14
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] I-vars = 1
2017-09-24 17:11:00.694 ios-class-guard[70940:19128276] Forbidden keywords = 81522
2017-09-24 17:11:00.696 ios-class-guard[70940:19128276] Done generating symbol table.
```

```
2017-09-24 17:11:00.696 ios-class-guard[70940:19128276] Generated unique symbols = 7
2017-09-24 17:11:00.698 ios-class-guard[70940:19128276] Obfuscating IB file at path
file:///Users/monkey/Documents/iosreversebook/sourcecode/chapter-8/TargetApp/TargetApp/
Base.lproj/LaunchScreen.storyboard
2017-09-24 17:11:00.701 ios-class-guard[70940:19128276] Obfuscating IB file at path
file:///Users/monkey/Documents/iosreversebook/sourcecode/chapter-8/TargetApp/TargetApp/
Base.lproj/Main.storyboard
```

生成的 Header.h 文件内容如下。通过内容可以知道，只有我们自己定义的类和方法被混淆了，系统符号全部被过滤了，而这正是我们想要的，代码如下。

```
// Properties

// Protocols

// Classes
#ifndef ViewController
#define ViewController n7zSFhsco6d5cr
#endif // ViewController
#ifndef CustomClass
#define CustomClass e4WF9CNk9Jk
#endif // CustomClass
#ifndef AppDelegate
#define AppDelegate d65EDcy2VI5
#endif // AppDelegate

// Categories

// Methods
#ifndef clickMe
#define clickMe t0KaQLG
#endif // clickMe
#ifndef setClickMe
#define setClickMe setT0KaQLG
#endif // setClickMe
#ifndef customMethod
#define customMethod x5inx8QAy9Bo
#endif // customMethod
#ifndef setCustomMethod
#define setCustomMethod setX5inx8QAy9Bo
#endif // setCustomMethod

// I-vars
```

把该文件的内容复制到 PrefixHeader.pch 文件中，这样在 Xcode 中 Xib 和 Storyboard 文件中的类名和方法名也会混淆（记得要备份工程）。重新 class-dump，发现确实是混淆过的符号，具体如下。

```
@interface n7zSFhsco6d5cr : UIViewController
{
}

- (void)t0KaQLG:(id)arg1;
- (void)didReceiveMemoryWarning;
- (void)viewDidLoad;

@end
```

8.2.2　二进制修改

前面介绍的这种方法需要我们重新添加一个 pch 文件，而最好的方法当然是不需要人工干预。从 Mach-O 文件的格式分析中可以知道，类名保存在 __TEXT,__objc_classname 中，方法名保存在 __TEXT,objc_methname 中。根据 class-dump 的原理可以知道，最后获取的类名和方法名也是从这里读取的。既然如此，是不是可以直接修改这两个 section 里对应的类名和方法名呢？我们来测试一下。

使用 MachOView 打开目标文件，找到 __TEXT,__objc_classname，在需要修改的类名前面的 Data 区域双击，修改类名，如图 8-4 所示（要保持修改前和修改后的类名一致）。重签名或者直接拖入 MonkeyApp 中运行，也没有问题。

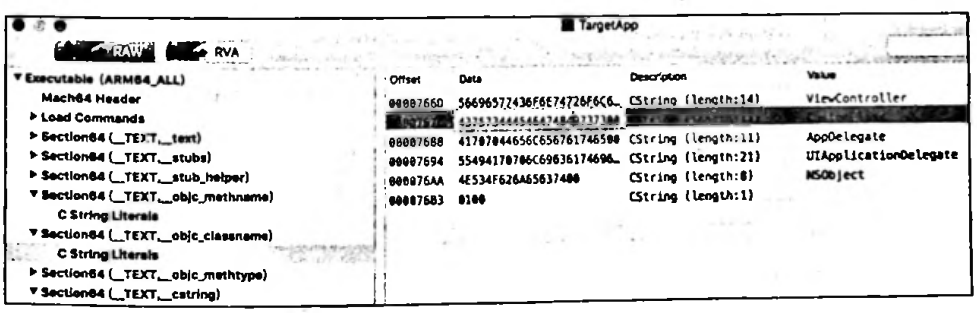


图 8-4　使用 MachOView 修改目标文件的类名

重新使用 class-dump，发现也是修改后的类名，具体如下。

```
@interface CusDEFGHIss : NSObject
```

```
{
}

- (void)customMethod;

@end
```

以上情况说明这种方式是可行的，具体过程读者自己就能实现（利用 ios-class-guard 生成的 json 文件，自己解析 Mach-O 中对应的 section 并进行替换）。不过，关于类名和方法名的混淆，还有很多坑。例如，一个类是通过文件、网络或者字符串动态获取的，如果被混淆了就会出错等。混淆前后的效果对比如图 8-5 所示。

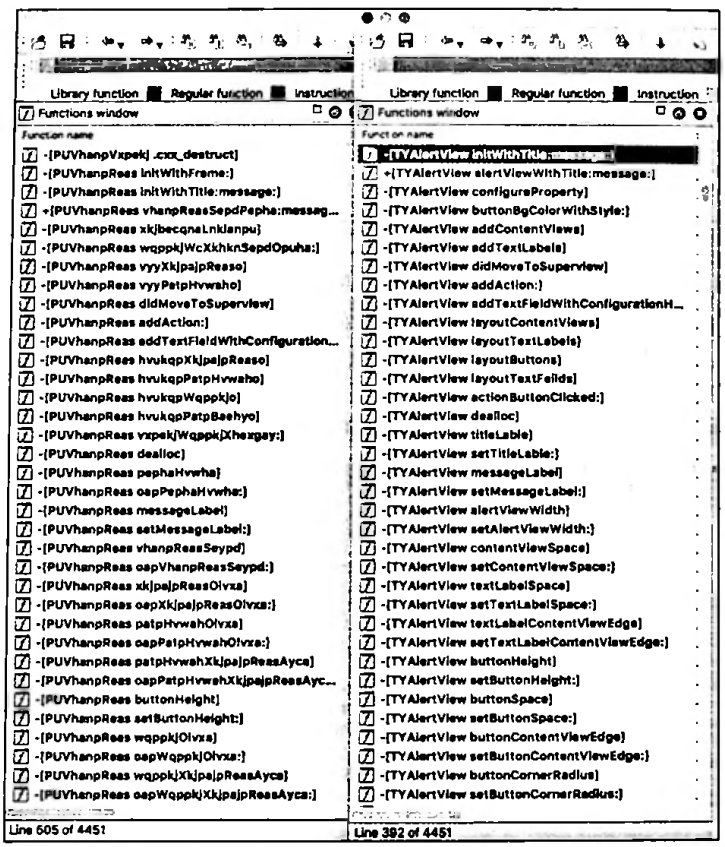


图 8-5　类名、方法名混淆前后的效果对比

另外，在分析这种混淆后的文件时，不能根据类名或者方法名去猜测方法的作用，只能通过动态调用调试等方法去确定某个方法的作用。

8.3　动态保护

虽然很多东西在静态分析中是无法分析出来的，例如加解密方法、动态传递的敏感参数等，但我们仍然可以通过动态调试去分析动态解密后的内容或者传递的参数等，然后通过注入动态库 hook 某个方法去改变程序的逻辑。对这种方法应该如何应对呢？下面讲解对抗中常用的一些动态保护方案。当然，破解和保护终究不是绝对的，只是破解成本和所得利益的关系非常紧密罢了。

8.3.1　反调试

反调试主要分为两种，一种是阻止调试器附加，另一种是检测调试器是否存在。下面介绍几种反调试方法。

1. ptrace

为了方便应用软件的开发和调试，UNIX 的早期版本就提供了一种对运行中的进程进行跟踪和控制的手段，那就是系统调用 ptrace。通过 ptrace，可以对另一个进程实现调试跟踪。同时，ptrace 提供了一个非常有用的参数，那就是 PT_DENY_ATTACH，这个参数用于告诉系统阻止调试器依附。所以，最常用的反调试方案就是通过调用 ptrace 来实现反调试，示例如下。

```
#ifndef PT_DENY_ATTACH
    #define PT_DENY_ATTACH 31
#endif
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);
ptrace(PT_DENY_ATTACH, 0, 0, 0);

void *handle = dlopen(0, RTLD_GLOBAL | RTLD_NOW);
ptrace_ptr_t ptrace_ptr = (ptrace_ptr_t)dlsym(handle, "ptrace");
ptrace_ptr(PT_DENY_ATTACH, 0, 0, 0);
```

2. sysctl

当一个进程被调试时，该进程中会有一个标记位来标记它正在被调试。可以通过 sysctl 函数查看当前进程的信息，通过是否有这个标记位即可检查当前调试的状态，示例如下。

```
BOOL isDebuggerPresent(){
    int name[4];                //指定查询信息的数组

    struct kinfo_proc info;     //查询的返回结果
```

```
    size_t info_size = sizeof(info);

    info.kp_proc.p_flag = 0;

    name[0] = CTL_KERN;
    name[1] = KERN_PROC;
    name[2] = KERN_PROC_PID;
    name[3] = getpid();

    if(sysctl(name, 4, &info, &info_size, NULL, 0) == -1){
        NSLog(@"sysctl error ...");
        return NO;
    }

    return ((info.kp_proc.p_flag & P_TRACED) != 0);
}
```

只要检测到调试器，就可以退出、制造崩溃或者隐藏功能等。当然，也可以定时查看有没有这个标记。

3. syscall

为了实现从用户态到内核态的切换，系统提供了一个系统调用函数 syscall（上面讲到的 ptrace 也是通过系统调用实现的）。

在 Kernel Syscalls（https://www.theiphonewiki.com/wiki/Kernel_Syscalls）里可以找到 ptrace 所对应的编号，具体如下。

```
26. ptrace              801e812c T
```

所以，如下调用等同于调用 ptrace。

```
syscall(26,31,0,0,0);
```

4. ARM

syscall 可以通过软中断实现从用户态到内核态的切换（也可以通过汇编调用 svc 实现）。sysctl 也可以转换成对应的汇编，以实现从用户态到内核态的切换，代码如下。

```
#ifdef __arm__
    asm volatile(
                 "mov r0,#31\n"
                 "mov r1,#0\n"
```

```
                "mov r2,#0\n"
                "mov r12,#26\n"   //ptrace
                "svc #80\n"
                );
#endif
#ifdef __arm64__
    asm volatile(
                "mov x0,#26\n"
                "mov x1,#31\n"
                "mov x2,#0\n"
                "mov x3,#0\n"
                "mov x16,#0\n"   //syscall
                "svc #128\n"
                );
#endif
```

5. 其他方法

还有一些方法可用于获取当前进程的调试状态，在实际分析中较少使用，举例如下。

（1）获取异常端口

```
struct macosx_exception_info{
    exception_mask_t masks[EXC_TYPES_COUNT];
    mach_port_t ports[EXC_TYPES_COUNT];
    exception_behavior_t behaviors[EXC_TYPES_COUNT];
    thread_state_flavor_t flavors[EXC_TYPES_COUNT];
    mach_msg_type_number_t cout;
};
struct macosx_exception_info *info = malloc(sizeof(struct macosx_exception_info));
task_get_exception_ports(mach_task_self(),
                                        EXC_MASK_ALL,
                                        info->masks,
                                        &info->cout,
                                        info->ports,
                                        info->behaviors,
                                        info->flavors);
for(uint32_t i = 0; i < info->cout; i ++){
    if(info->ports[i] != 0 || info->flavors[i] == THREAD_STATE_NONE){
        NSLog(@"debugger detected via exception ports (null port)!\n");
    }
}
```

（2）isatty

```
if (isatty(1)) {
    NSLog(@"Being Debugged isatty");
}
```

（3）ioctl

```
if (!ioctl(1, TIOCGWINSZ)) {
    NSLog(@"Being Debugged ioctl");
}
```

8.3.2 反反调试

既然讲到反调试，我们就应该了解一下如何过掉常见的反调试机制。这里主要针对 ptrace、sysctl、syscall 进行反反调试，流程很简单：hook 函数→判断参数→返回结果。反反调试有 3 种处理方法，即越狱 hook、非越狱 hook、Python 脚本。

1. 越狱 hook

越狱 hook 的代码如下。

```
#import <substrate.h>
#import <sys/sysctl.h>

static int (*orig_ptrace) (int request, pid_t pid, caddr_t addr, int data);
static int my_ptrace (int request, pid_t pid, caddr_t addr, int data){
    if(request == 31){
        NSLog(@"[AntiAntiDebug] - ptrace request is PT_DENY_ATTACH");
        return 0;
    }
    return orig_ptrace(request,pid,addr,data);
}

static int (*orig_sysctl)(int * name, u_int namelen, void * info, size_t * infosize, void
* newinfo, size_t newinfosize);
static int my_sysctl(int * name, u_int namelen, void * info, size_t * infosize, void * newinfo,
size_t newinfosize){
    if(namelen == 4 && name[0] == CTL_KERN && name[1] == KERN_PROC && name[2] == KERN_PROC_PID
&& info && infosize && ((int)*infosize == sizeof(struct kinfo_proc))){
        int ret = orig_sysctl(name,namelen,info,infosize,newinfo,newinfosize);
```

```
        struct kinfo_proc *info_ptr = (struct kinfo_proc *)info;
        if(info_ptr && (info_ptr->kp_proc.p_flag & P_TRACED) != 0){
            NSLog(@"[AntiAntiDebug] - sysctl query trace status.");
            info_ptr->kp_proc.p_flag ^= P_TRACED;
            if((info_ptr->kp_proc.p_flag & P_TRACED) == 0){
                NSLog(@"[AntiAntiDebug] trace status reomve success!");
            }
        }
        return ret;
    }
    return orig_sysctl(name,namelen,info,infosize,newinfo,newinfosize);
}
static void* (*orig_syscall)(int code, va_list args);
static void* my_syscall(int code, va_list args){
    int request;
    va_list newArgs;
    va_copy(newArgs, args);
    if(code == 26){
#ifdef __LP64__
        __asm__(
                "ldr %w[result], [fp, #0x10]\n"
                : [result] "=r" (request)
                :
                :
                );
#else
        request = va_arg(args, int);
#endif
        if(request == 31){
            NSLog(@"[AntiAntiDebug] - syscall call ptrace, and request is PT_DENY_ATTACH");
            return nil;
        }
    }
    return (void*)orig_syscall(code, newArgs);
}

%ctor{
    MSHookFunction((void
*)MSFindSymbol(NULL,"_ptrace"),(void*)my_ptrace,(void**)&orig_ptrace);
    MSHookFunction((void *)sysctl,(void*)my_sysctl,(void**)&orig_sysctl);
    MSHookFunction((void *)syscall,(void*)my_syscall,(void**)&orig_syscall);
    NSLog(@"[AntiAntiDebug] Module loaded!!!");
}
```

2. 非越狱 hook

非越狱 hook 的原理和越狱 hook 一样，只不过将工具换成了 fishhook，代码也已集成在 MonkeyApp 中，具体如下。

```
#import "fishhook.h"
#import <Foundation/Foundation.h>
#import <sys/sysctl.h>

typedef int (*ptrace_ptr_t)(int _request,pid_t _pid, caddr_t _addr,int _data);
typedef void* (*dlsym_ptr_t)(void * __handle, const char* __symbol);
typedef int (*syscall_ptr_t)(int, ...);
typedef int (*sysctl_ptr_t)(int *,u_int, void*, size_t*,void*, size_t);


static ptrace_ptr_t orig_ptrace = NULL;
static dlsym_ptr_t orig_dlsym = NULL;
static sysctl_ptr_t orig_sysctl = NULL;
static syscall_ptr_t orig_syscall = NULL;

int my_ptrace(int _request, pid_t _pid, caddr_t _addr, int _data);
void* my_dlsym(void* __handle, const char* __symbol);
int my_sysctl(int * name, u_int namelen, void * info, size_t * infosize, void * newinfo,
size_t newinfosize){
    if(namelen == 4 && name[0] == CTL_KERN && name[1] == KERN_PROC && name[2] == KERN_PROC_PID
&& info && infosize && ((int)*infosize == sizeof(_kinfo_proc))){
        int ret = orig_sysctl(name, namelen, info, infosize, newinfo, newinfosize);
        struct kinfo_proc *info_ptr = (struct kinfo_proc *)info;
        if(info_ptr && (info_ptr->kp_proc.p_flag & P_TRACED) != 0){
            NSLog(@"[AntiAntiDebug] - sysctl query trace status.");
            info_ptr->kp_proc.p_flag ^= P_TRACED;
            if((info_ptr->kp_proc.p_flag & P_TRACED) == 0){
                NSLog(@"trace status reomve success!");
            }
        }
        return ret;
    }
    return orig_sysctl(name, namelen, info, infosize, newinfo, newinfosize);
}

void* my_dlsym(void* __handle, const char* __symbol){
    if(strcmp(__symbol, "ptrace") != 0){
        return orig_dlsym(__handle, __symbol);
    }
```

```
    NSLog(@"[AntiAntiDebug] - dlsym get ptrace symbol");

    return my_ptrace;
}

typedef struct kinfo_proc _kinfo_proc;

int my_sysctl(int * name, u_int namelen, void * info, size_t * infosize, void * newinfo,
size_t newinfosize){
    int ret = orig_sysctl(name, namelen, info, infosize, newinfo, newinfosize);
    if(namelen == 4 && name[0] == CTL_KERN && name[1] == KERN_PROC && name[2] == KERN_PROC_PID
&& info && infosize && (*infosize == sizeof(_kinfo_proc))){
        struct kinfo_proc *info_ptr = (struct kinfo_proc *)info;
        if(info_ptr && (info_ptr->kp_proc.p_flag & P_TRACED) != 0){
            NSLog(@"[AntiAntiDebug] - sysctl query trace status.");
            info_ptr->kp_proc.p_flag ^= P_TRACED;
            if((info_ptr->kp_proc.p_flag & P_TRACED) == 0){
                NSLog(@"trace status reomve success!");
            }
        }
    }
    return ret;
}

int my_syscall(int code, va_list args){
    int request;
    va_list newArgs;
    va_copy(newArgs, args);
    if(code == 26){
#ifdef __LP64__
        __asm__(
                "ldr %w[result], [fp, #0x10]\n"
                : [result] "=r" (request)
                :
                :
                );
#else
        request = va_arg(args, int);
#endif
        if(request == 31){
            NSLog(@"[AntiAntiDebug] - syscall call ptrace, and request is PT_DENY_ATTACH");
            return 0;
        }
    }
    return orig_syscall(code, newArgs);
```

```
}

__attribute__((constructor)) static void entry(){
    NSLog(@"[AntiAntiDebug Init]");

    rebind_symbols((struct rebinding[1]){{"ptrace", my_ptrace, (void*)&orig_ptrace}},1);

    rebind_symbols((struct rebinding[1]){{"dlsym", my_dlsym, (void*)&orig_dlsym}},1);

    rebind_symbols((struct rebinding[1]){{"sysctl", my_sysctl, (void*)&orig_sysctl}},1);

    rebind_symbols((struct rebinding[1]){{"syscall", my_syscall,
(void*)&orig_syscall}},1);
}
```

3. Python 脚本

如果不采用 hook 的方法，可以借助 LLDB 的 Python 脚本自动在目标函数处设置条件断点，然后在断点的回调中改变对应的寄存器。为了方便讲解，笔者使用了基于 chisel 的脚本。将包含以下内容的文件放到从 ~/.lldbinit 中导入的 chisel 模块的 commands 目录下，例如 /usr/local/Cellar/chisel/1.5.0/libexec/commands。

```
T#!/usr/bin/python
# -*- coding: utf-8 -*-

"""
反反调试脚本，过了反调试后记得：
aadebug -d
否则，程序运行会很慢。如果有定时器定时检测，建议编写 Tweak
"""

import lldb
import fblldbbase as fb
import fblldbobjcruntimehelpers as objc

def lldbcommands():
    return [
        AMAntiAntiDebug()
    ]

class AMAntiAntiDebug(fb.FBCommand):
    def name(self):
```

```
        return 'aadebug'

    def description(self):
        return "anti anti debug ptrace syscall sysctl"

    def options(self):
        return [
            fb.FBCommandArgument(short='-d', long='--disable', arg='disable', boolean=True,
default=False, help='disable anti anti debug.')
        ]

    def run(self, arguments, options):
        if options.disable:
            target = lldb.debugger.GetSelectedTarget()
            target.BreakpointDelete(self.ptrace.id)
            target.BreakpointDelete(self.syscall.id)
            target.BreakpointDelete(self.sysctl.id)
            print "anti anti debug is disabled!!!"
        else:
            self.antiPtrace()
            self.antiSyscall()
            self.antiSysctl()
            print "anti anti debug finished!!!"

    def antiPtrace(self):
        ptrace = lldb.debugger.GetSelectedTarget().BreakpointCreateByName("ptrace")
        if isMac():
            ptrace.SetCondition('$rdi==31')
        elif is64Bit():
            ptrace.SetCondition('$x0==31')
        else:
            ptrace.SetCondition('$r0==31')
        ptrace.SetScriptCallbackFunction('sys.modules[\'' + __name__ +
'\'].ptrace_callback')
        self.ptrace = ptrace

    def antiSyscall(self):
        syscall = lldb.debugger.GetSelectedTarget().BreakpointCreateByName("syscall")
        if isMac():
            syscall.SetCondition('$rdi==26 && $rsi==31')
        elif is64Bit():
            syscall.SetCondition('$x0==26 && *(int *)$sp==31')
        else:
            syscall.SetCondition('$r0==26 && $r1==31')
```

```
        syscall.SetScriptCallbackFunction('sys.modules[\'' + __name__ +
'\'].syscall_callback')
        self.syscall = syscall

    def antiSysctl(self):
        sysctl = lldb.debugger.GetSelectedTarget().BreakpointCreateByName("sysctl")
        if isMac():
            sysctl.SetCondition('$rsi==4 && *(int *)$rdi==1 && *(int *)($rdi+4)==14 && *(int
*)($rdi+8)==1')
        elif is64Bit():
            sysctl.SetCondition('$x1==4 && *(int *)$x0==1 && *(int *)($x0+4)==14 && *(int
*)($x0+8)==1')
        else:
            sysctl.SetCondition('$r1==4 && *(int *)$r0==1 && *(int *)($r0+4)==14 && *(int
*)($r0+8)==1')
        sysctl.SetScriptCallbackFunction('sys.modules[\'' + __name__ +
'\'].sysctl_callback')
        self.sysctl = sysctl

    def antiExit(self):
        self.exit = lldb.debugger.GetSelectedTarget().BreakpointCreateByName("exit")
        exit.SetScriptCallbackFunction('sys.modules[\'' + __name__ + '\'].exit_callback')

#暂时只考虑 ARMv7 和 ARM64
def is64Bit():
    arch = objc.currentArch()
    if arch == "arm64":
        return True
    return False

def isMac():
    arch = objc.currentArch()
    if arch == "x86_64":
        return True
    return False

def ptrace_callback(frame, bp_loc, internal_dict):
    print "find ptrace"
    register = "x0"
    if isMac():
        register = "rdi"
    elif not is64Bit():
        register = "r0"
    frame.FindRegister(register).value = "0"
```

```
    lldb.debugger.HandleCommand('continue')

def syscall_callback(frame, bp_loc, internal_dict):
    print "find syscall"
    lldb.debugger.GetSelectedTarget().GetProcess().SetSelectedThread(frame.GetThread())
    if isMac():
        lldb.debugger.HandleCommand('register write $rsi 0')
    elif is64Bit():
        lldb.debugger.HandleCommand('memory write "$sp" 0')
    else:
        lldb.debugger.HandleCommand('register write $r1 0')
    lldb.debugger.HandleCommand('continue')

def sysctl_callback(frame, bp_loc, internal_dict):
    module = frame.GetThread().GetFrameAtIndex(1).GetModule()
    currentModule = lldb.debugger.GetSelectedTarget().GetModuleAtIndex(0)
    if str(module)[:20] == str(currentModule)[:20]:  # to fix that
        print "find sysctl"
        register = "x2"
        if isMac():
            register = "rdx"
        elif not is64Bit():
            register = "r2"
        frame.FindRegister(register).value = "0"
    lldb.debugger.HandleCommand('continue')

def exit_callback(frame, bp_loc, internal_dict):
    print "find exit"
    lldb.debugger.GetSelectedTarget().GetProcess().SetSelectedThread(frame.GetThread())
    lldb.debugger.HandleCommand('thread return')
    lldb.debugger.HandleCommand('continue')
```

8.3.3　反注入

反注入主要还是一种注入检测机制，即检测当前有没有其他模块注入。曾经出现过一种主动防止注入的方法，但该方法现在已经没有用了。

1. restrict 防注入

当旧版的 dyld 检测到存在 `__RESTRICT,__restrict` 这样的 section 时，`DYLD_INSERT_LIBRARIES` 环境变量会被忽略，导致注入失败。因此，在 Xcode 的编译设置选项“Other Linker Flags”中加上 `-Wl,-sectcreate,__RESTRICT,__restrict,/dev/null` 参数，就能达到反注入的效果。

在新版的 dyld 及 iOS 10 的测试中发现，该方法已经没有用了，dyld 已经不检测这个 section 了，而且 optool 自带 unrestrict 的功能。所以，这个方法现在已经没有实际的用处了。

2. 注入检测

从另一个角度来分析，可以通过注入检测的方式来判断有没有进行注入。和调试检测一样，具体的应对措施由我们自己制定。注入检测可以判断加载模块中有没有一些不在正常加载列表中的模块，使用 _dyld_get_image_name 获取模块名，然后进行对比，具体如下。

```
int AMCheckInjector(){
    int count = _dyld_image_count();

    if(count >0){
        for(int i = 0; i < count; i++){
            const char* dyld = _dyld_get_image_name(i);
            if(strstr(dyld, "DynamicLibraries")){          //或者发现其他不在白名单内的库是以
load command 方式注入的
                return 1;
            }
        }
    }

    return 0;
}
```

8.3.4 hook 检测

hook 的方式包括 Method Swizzle、符号表替换、inline hook。针对不同的 hook 方法，需要根据其原理制定不同的检测方案。

1. Method Swizzle

Method Swizzle 的原理是替换 imp，通过 dladdr 得到 imp 地址所在的模块，简单的代码如下。如果所在模块不是主二进制模块，就认为被恶意 hook 了。

```
bool CheckHookForOC(const char* clsname,const char* selname){
    Dl_info info;
    SEL sel = sel_registerName(selname);
    Class cls = objc_getClass(clsname);
    Method method = class_getInstanceMethod(cls, sel);
    if(!method){
```

```
        method = class_getClassMethod(cls, sel);
    }
    IMP imp = method_getImplementation(method);

    if(!dladdr((void*)imp, &info)){
        return false;
    }

    printf("%s\n", info.dli_fname);

    if(!strncmp(info.dli_fname, "/System/Library/Frameworks", 26)){
        return false;
    }

    if(!strcmp(info.dli_fname, _dyld_get_image_name(0))){
        return false;
    }

    return true;
}
```

2. 符号表替换

因为 fishhook 是基于懒加载符号表和非懒加载符号表进行替换的，所以通过遍历符号表中的指针就能够判断程序是否被恶意 hook 了。非懒加载的指针指向真实的地址，而懒加载的指针在没有解析到真实的地址之前指向 __stub_helper，因此，遍历符号表中的每一个指针，然后判断指针是不是指向 __stub_helper 或者系统模块，就可以判断程序是否被 hook 了。希望读者在理解 fishhook 代码和 Mach-O 文件结构的基础上进行分析，结合 dladdr 自己的环境来实现具体的代码。

3. inline hook

inline hook 的原理是通过替换函数内部的前几条指令跳转来实现。所以，可以通过分析函数内存前面几条指令中有没有跳转来判断是不是 inline hook。当然，hook 也可以替换某条指令或者 patch 内存中的某些指令，但是在这样的情况下只能通过计算代码的哈希值进行验证。

8.3.5　完整性校验

完整性校验可以从文件完整性、是否重签名、Bundle ID 等方面入手，而逆向过程涉及对文件 load command 的修改、对文件进行重签名或者修改 Bundle ID，也就是说，可以通过这些方面

来验证程序有没有被修改。

1. load command

直接读取 Mach-O 文件的 load command 中的 LC_LOAD_DYLIB 或 LC_LOAD_WEAK_DYLIB，代码如下。

```
NSArray* MachOParser::find_load_dylib(){
    NSMutableArray* array = [[NSMutableArray alloc] init];;
    mach_header_t *header = (mach_header_t*)base;
    segment_command_t *cur_seg_cmd;
    uintptr_t cur = (uintptr_t)this->base + sizeof(mach_header_t);
    for (uint i = 0; i < header->ncmds; i++,cur += cur_seg_cmd->cmdsize) {
        cur_seg_cmd = (segment_command_t*)cur;
        if(cur_seg_cmd->cmd == LC_LOAD_DYLIB || cur_seg_cmd->cmd == LC_LOAD_WEAK_DYLIB){
            dylib_command *dylib = (dylib_command*)cur_seg_cmd;
            char* name = (char*)((uintptr_t)dylib + dylib->dylib.name.offset);
            NSString* dylibName = [NSString stringWithUTF8String:name];
            [array addObject:dylibName];
        }
    }
    return [array copy];
}
```

2. 代码检验

获取内存中运行的代码的 MD5 值。如果内存中的代码被修改了，那么获取的 MD5 值就会不一样，具体如下。

```
NSString* MachOParser::get_text_data_md5(){
    NSMutableString *result = [NSMutableString string];
    section_info_t *section = this->find_section("__TEXT", "__text");
    local_addr startAddr =  section->addr;
    unsigned char hash[CC_MD5_DIGEST_LENGTH];
    CC_MD5((const void *)startAddr, (CC_LONG)section->section->size, hash);
    for (int i = 0; i < CC_MD5_DIGEST_LENGTH; i++) {
        [result appendFormat:@"%02x",hash[i]];
    }
    return [result copy];
}
```

3. 重签名检验

重签名可以通过多种方式进行检验。可以判断 Bundle ID 是否被修改，或者判断 app 目录下面有没有 embedded.mobileprovision 文件及该文件中包含的签名信息；也可以直接从可执行文件的 LC_CODE_SIGNATURE 中读取信息，获取 Bundle ID 的系统接口 `[[NSBundle mainBundle] bundleIdentifier]`，或者通过如下函数获取路径并读取其中的内容。

```
NSString *provisionPath = [[NSBundle mainBundle] pathForResource:@"embedded"
ofType:@"mobileprovision"];
```

8.4　代码混淆

在静态分析中，可以使用 Hopper 和 IDA 等汇编工具得到汇编代码，甚至得到对应的伪代码，而对 OC 这样的动态语言来说，真的就像源代码一样。为了提高静态分析的难度，需要通过代码混淆来增加一些无用的代码，拆分代码块，并使代码扁平化。要达到这样的效果，需要在编译器处理分析生成代码时做这件事情。

Xcode 编译器使用的前端是 Clang，Clang 是 LLVM 的一部分，而 LLVM 本身是开源的，这样便可以基于开源的 LLVM 代码进行代码混淆了。下面分别介绍 LLVM 及基于 LLVM 的代码混淆技术。

8.4.1　什么是 LLVM

LLVM 项目是一系列分模块和可重用的编译工具链，它提供了一种代码编写良好的中间表示（IR），可以作为多种语言的后端，还可以提供与编程语言无关的优化和针对多种 CPU 的代码生成功能。关于 LLVM 编译的整个流程，以及基于 LLVM 可以做哪些事情，在笔者的文章《关于 LLVM，这些东西你必须知道！》（http://www.alonemonkey.com/2016/12/21/ learning-llvm/）中已经介绍过了，对此感兴趣的读者可以阅读这篇文章。

LLVM 架构的主要组成部分如下。

- 前端：用于获取源代码，然后将它转换为某种中间表示。可以选择不同的编译器作为 LLVM 的前端，例如 gcc、Clang。
- Pass：用于在程序的中间表示之间变换。在一般情况下，Pass 可以用来优化代码，而这部分通常是我们关注的。
- 后端：用于生成实际的机器码。

虽然如今大多数编译器采用的都是这种架构，但 LLVM 对不同的语言提供了同一种中间表示。传统的编译器架构如图 8-6 所示。

图 8-6 传统的编译器架构

LLVM 的架构如图 8-7 所示。

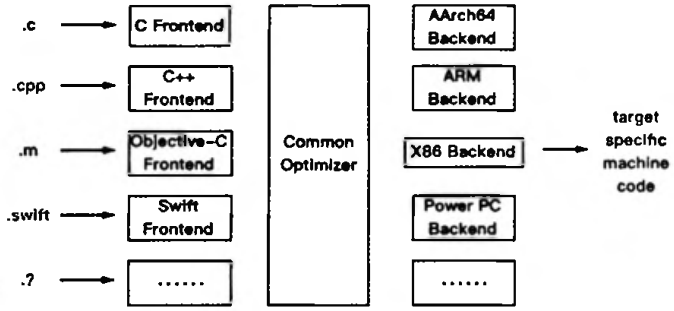


图 8-7 LLVM 编译器架构

当编译器需要支持多种源代码和目标架构时，基于 LLVM 的架构设计一门新的语言只需要实现一个新的前端，支持新的后端架构也只需要去实现一个新的后端，其他部分完全可以复用，不用重新设计。在基于 LLVM 进行代码混淆时，只需要关注中间层代码（IR）表示。

8.4.2 下载和编译 LLVM

首先，下载并编译 LLVM 工具链里面的内容，这里主要是为了编译前端工具 Clang。可以通过以下方式获取 release 的代码或者 trunk 开发中的代码。

- 直接从官网下载（http://releases.llvm.org/download.html，当前最新 release 版本为 5.0.0）。
- 从 svn 获取，具体如下。

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
svn co http://llvm.org/svn/llvm-project/cfe/trunk llvm/tools/clang
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk llvm/projects/compiler-rt (按需拉取)
svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk
llvm/tools/clang/tools/extra (按需拉取)
```

- 从 git 获取，具体如下。

```
git clone https://git.llvm.org/git/llvm.git/
git clone https://git.llvm.org/git/clang.git/ llvm/tools/clang
git clone https://git.llvm.org/git/clang-tools-extra.git/ llvm/tools/clang/tools/extra (按需拉取)
git clone https://git.llvm.org/git/compiler-rt.git/ llvm/projects/compiler-rt (按需拉取)
```

在本节中，我们从 git 上拉取最新的代码，使用 cmake 进行编译。

安装 cmake，命令如下。

```
brew install cmake
```

编译生成 Xcode 项目，命令如下。

```
mkdir build
cd build
cmake -G Xcode CMAKE_BUILD_TYPE="Debug" ../llvm
open LLVM.xcodeproj
```

打开 Xcode，会出现如图 8-8 所示的界面。

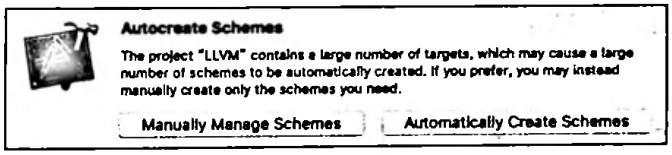


图 8-8　打开 Xcode 手动管理 Scheme

这时不需要生成所有的 Scheme，可以单击“Manually Manage Scheme”选项，然后将需要编译的 Target（例如 Clang）选中，如图 8-9 所示。接下来，按“Command + R”快捷键等待编译完成。

编译完成，build/Debug/bin 目录下将生成 Clang 的可执行文件，运行如下命令即可看到对应的版本号（可以看到，git 上面正在开发的是 6.0.0 版本）。

```
➜  bin ./clang -v
clang version 6.0.0 (https://git.llvm.org/git/clang.git/
29487927c0f5d8cd6b23978a0216b17041161cc5) (https://git.llvm.org/git/llvm.git/
a420002dfe273464f9482252b32dcf2297c95bd5)
Target: x86_64-apple-darwin16.7.0
Thread model: posix
InstalledDir: /Users/alonemonkey/Documents/build/Debug/bin/.
```

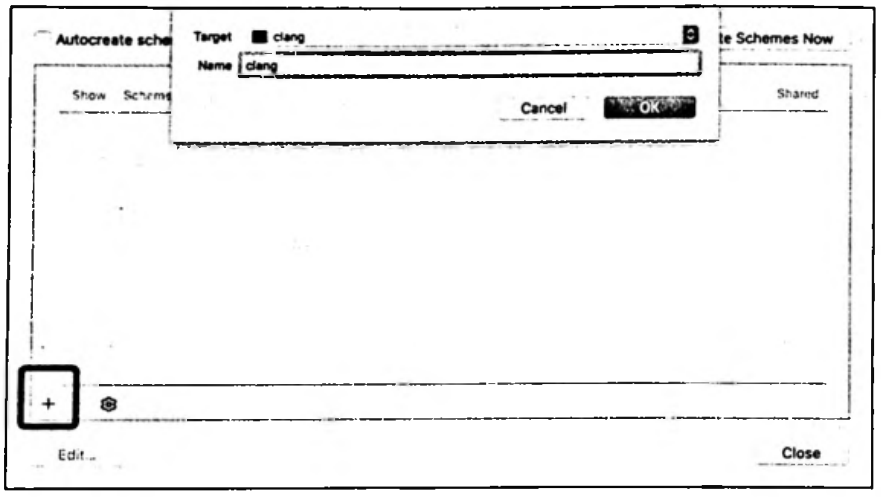


图 8-9 选择 Target Clang 作为一个 Scheme

另外，在 Xcode 里面，Clang 的版本是与 Xcode 的版本对应的，和开源的版本不一致，而且苹果在开源的 LLVM 中增加了自己的修改。

8.4.3 开发和调试 Pass

LLVM Pass 是 LLVM 系统中非常重要和有趣的一个模块。Pass 处理编译过程中代码的转换及优化工作。所有的 Pass 都是 Pass 的子类，不同的 Pass 实现不同的作用，可以继承 ModulePass、CallGraphSCCPass、FunctionPass、LoopPass、RegionPass、BasicBlockPass 等类。下面介绍如何编写一个 Pass，以及 Pass 的编译、加载和运行过程。

编写一个“Hello Pass”程序，简单地输出程序中每个非外部方法的方法名，不修改程序的功能。官方 Demo 的源代码存储在 lib/Transforms/Hello 文件夹中。

1. 配置编译环境

在编译一个新的 Pass 前，要在 lib/Transforms/目录下新建一个文件夹。在这里，笔者新建的文件夹是 PassDemo。接下来，配置编译脚本去编译源代码。在新建的目录下创建文件 Hello.cpp。然后，新建文件 CMakeLists.txt，编辑编译配置，具体如下。

```
add_llvm_loadable_module( LLVMPassDemo
 Hello.cpp

 DEPENDS
 intrinsics_gen
```

```
PLUGIN_TOOL
opt
)
```

将下面这行代码添加到 lib/Transforms/CMakeLists.txt 中。

```
add_subdirectory(PassDemo)
```

编译脚本指定使用编译源文件 Hello.cpp 编译生成 $(LEVEL)/lib/LLVMPassDemo.dylib 动态库，该文件可以被 opt 通过 -load 参数动态加载。如果不是在 Mac 平台上，生成的后缀会不一样，例如在 Linux 平台上后缀是 so。配置好编译脚本后，打开 Xcode 重新 build，然后在项目里面添加 LLVMPassDemo 的 Target 为 Scheme，在工程中找到 loadable modules 下面 LLVMPassDemo 里面的 Hello.cpp 文件，就可以编写代码了，如图 8-10 所示。

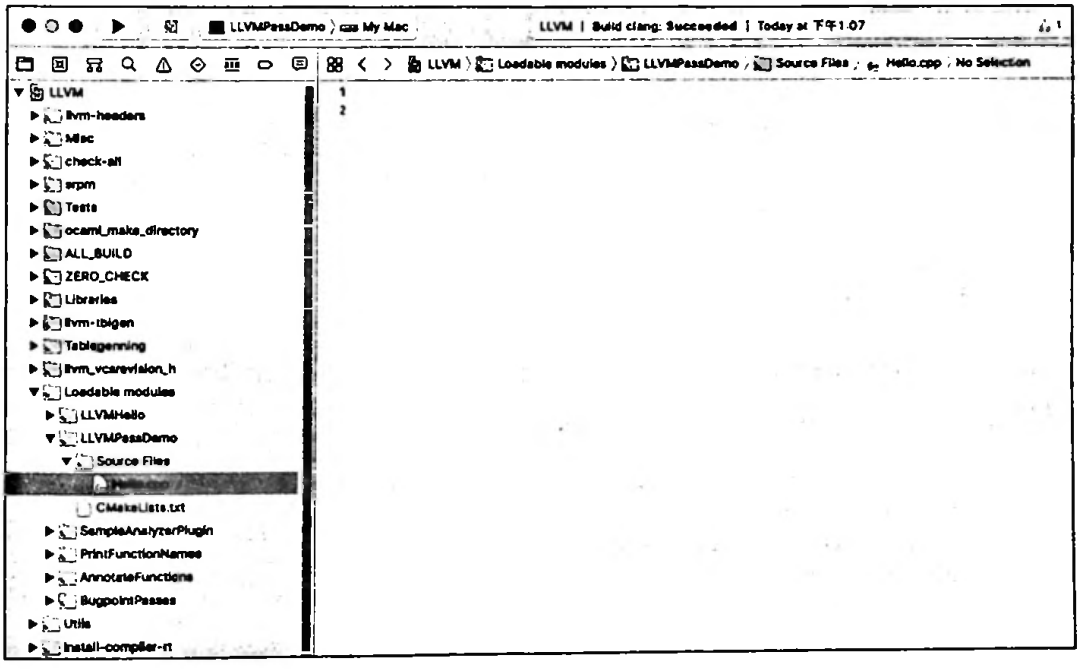


图 8-10　在 Xcode 中找到对应的 Target LLVMPassDemo

2. 编写代码

首先，编辑 Hello.cpp 文件。因为需要编写一个 Pass 来操作 Function，还可能输出一些内容，所以导入如下头文件。

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
```

因为导入文件里面的方法是属于 LLVM 命名空间的，所以要先指定使用的命名空间，代码如下。

```
using namespace llvm;
```

编写一个匿名的命名空间，代码如下。C++ 中的匿名命名空间和 C 中的 static 关键词一样，定义在匿名空间中的变量仅对当前文件可见。

```
namespace {
```

定义 Hello 类继承 FunctionPass，代码如下。功能不同的 Pass 会继承不同的父类。现在只需要知道类继承自 FunctionPass，就可以操作代码中的方法了。

```
struct Hello : public FunctionPass{
```

定义可供 LLVM 标示的 Pass ID，代码如下。

```
static char ID;
Hello() : FunctionPass(ID){}
```

定义一个 runOnFunction 重载继承自父类的抽象虚函数，在这个函数里面可以针对函数做特殊处理。在这里只打印每个方法的名字，代码如下。

```
    bool runOnFunction(Function &F) override{
        errs() << "Hello: ";
        errs().write_escaped(F.getName()) << "\n";
        return false;
    }
};
}
```

初始化 Pass ID，代码如下。因为 LLVM 将使用 ID 的地址去识别一个 Pass，所以初始化的值不重要。

```
char Hello::ID = 0;
```

最后，注册 Pass Hello，指定命令行参数为“hello”，名字说明是“Hello World Pass”，代码如下。

```
static RegisterPass<Hello> X("hello", "Hello World Pass", false/* Only looks at CFG */,
false/* Analysis Pass */);
```

整个 Hello.cpp 文件的内容如下。

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
namespace {
    struct Hello : public FunctionPass{
        static char ID;

        Hello() : FunctionPass(ID){}

        bool runOnFunction(Function &F) override{
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << "\n";
            return false;
        }
    };
}
char Hello::ID = 0;
static RegisterPass<Hello> X("hello", "Hello World Pass", false/* Only looks at CFG */,
false/* Analysis Pass */);
```

选中 Target LLVMPassDemo，按“Command + B”快捷键进行编译。编译后会生成文件 build/Debug/lib/LLVMPassDemo.dylib。

3. 使用 opt 加载

因为我们之前已经在代码中通过 RegisterPass 注册了 Pass，所以现在可以通过 opt -load 命令去加载动态库并指定参数 hello 了。先准备一个测试用的源文件，具体如下。

```
#include <stdio.h>
int add(int x, int y) {
    return x + y;
}
int main(){
    printf("%d",add(3,4));
```

```
    return 0;
}
```

编译源文件，生成 bitcode，具体如下。

```
path/to/build/Debug/bin/clang -emit-llvm -c test.mm -o test.bc
```

然后，在 Xcode 中将 opt 的 target 添加到 scheme 中。编辑 scheme 的启动参数，如图 8-11 所示。按“Command + R”快捷键运行，输出结果如下。

```
Hello: _Z3addii
Hello: main
Program ended with exit code: 0
```

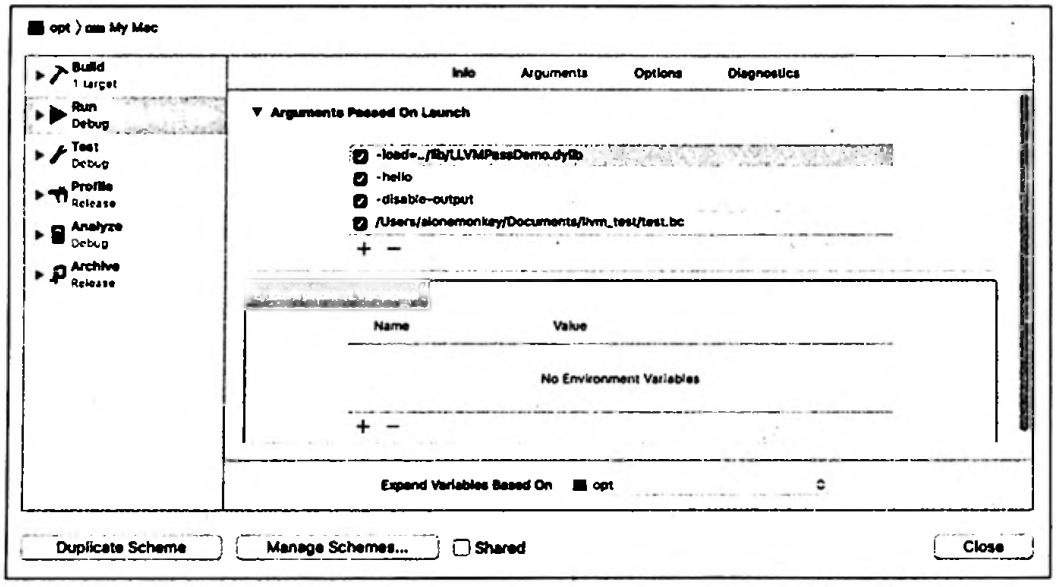


图 8-11 编辑 opt scheme 的启动参数

也可以直接下断点进行调试，如图 8-12 所示。

通过 -help 参数可以看到注册的 Pass 参数，具体如下。

```
➜  bin ./opt -load ../lib/LLVMPassDemo.dylib -help
......
-hello                                  - Hello World Pass
......
```

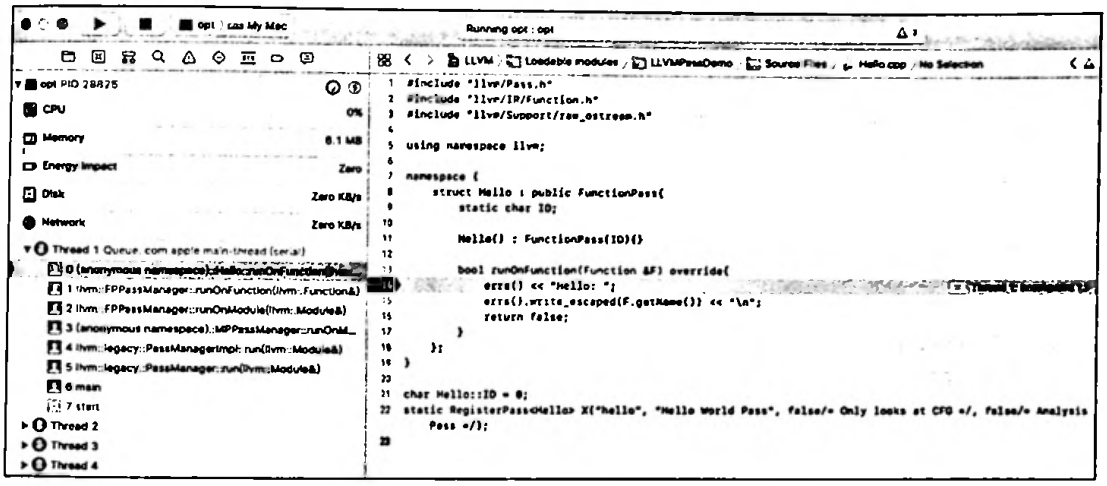


图 8-12　使用 Xcode 调试编写的 Pass

PassManager 还提供了 --time-passes 参数，用于输出 Pass 的时间占比，具体如下。

```
➜  bin ./opt -load ../lib/LLVMPassDemo.dylib -hello -time-passes -disable-output
~/Documents/llvm_test/test.bc

Hello: _Z3addii
Hello: main
===-------------------------------------------------------------------------===
                      ... Pass execution timing report ...
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0002 seconds (0.0003 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0002 ( 81.0%)   0.0000 ( 20.0%)   0.0002 ( 69.2%)   0.0002 ( 71.5%)  Module Verifier
   0.0000 ( 19.0%)   0.0000 ( 80.0%)   0.0001 ( 30.8%)   0.0001 ( 28.5%)  Hello World Pass
   0.0002 (100.0%)   0.0000 (100.0%)   0.0002 (100.0%)   0.0003 (100.0%)  Total

===-------------------------------------------------------------------------===
                            LLVM IR Parsing
===-------------------------------------------------------------------------===
  Total Execution Time: 0.0020 seconds (0.0022 wall clock)

   ---User Time---   --System Time--   --User+System--   ---Wall Time---  --- Name ---
   0.0013 (100.0%)   0.0006 (100.0%)   0.0020 (100.0%)   0.0022 (100.0%)  Parse IR
   0.0013 (100.0%)   0.0006 (100.0%)   0.0020 (100.0%)   0.0022 (100.0%)  Total
```

为了在执行 opt 时能自动检测 LLVMPassDemo 模块有没有被修改（如果模块被修改了，需要重新编译 LLVMPassDemo 模块），可以把 LLVMPassDemo 模块添加到 opt 的依赖里面。编辑 tools/opt/CMakeLists.txt，完成后按“Command + B”快捷键即可生效，具体如下。

```
add_llvm_tool(opt
  AnalysisWrappers.cpp
  BreakpointPrinter.cpp
  GraphPrinters.cpp
  NewPMDriver.cpp
  PassPrinters.cpp
  PrintSCC.cpp
  opt.cpp

  DEPENDS
  intrinsics_gen
  LLVMPassDemo
  )
```

4. 其他操作

在编写一个新的 Pass 时，应该根据需求选择需要继承的父类。根据作用的不同，父类有 BasicBlockPass、CallGraphSCCPass、FunctionPass、ImmutablePass、LoopPass、MachineFunctionPass、RegionPass、ModulePass，具体可以参考官方文档（http://llvm.org/docs/WritingAnLLVM Pass.html）。

如果编写的 Pass 会用到其他 Pass 提供的函数功能，就需要在 getAnalysisUsage 中说明（例如，想要获取程序中存在循环的信息）。以上面的 Pass 为例，导入头文件（`#include "llvm/Analysis/LoopInfo.h"`），并在 getAnalysisUsage 中调用，具体如下。

```
AU.addRequired<LoopInfoWrapperPass>();
AU.setPreservesAll();
```

然后，就可以通过它提供的接口获取循环的个数了，具体如下。

```
LoopInfo &LI = getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
int loopCounter = 0;
for (LoopInfo::iterator i = LI.begin(), e = LI.end(); i != e; ++i) {
    loopCounter++;
}
errs() << "loop num:" << loopCounter << "\n";
```

测试源文件的内容如下。

```
#include <stdio.h>
int add(int x, int y) {
    for (int i = 0; i < 10; i++) {
        printf("%d\n",i);
    }
    return x + y;
}
int main(){
    printf("%d",add(3,4));
    return 0;
}
```

输出结果如下。

```
Hello: _Z3addii
loop num:1
Hello: main
loop num:0
```

8.4.4　OLLVM 源代码分析

基于 LLVM 的代码混淆技术主要基于中间层代码的 Pass。在 GitHub 上有一个开源的基于 LLVM Pass 的代码混淆项目 OLLVM（https://github.com/obfuscator-llvm/obfuscator），不过它目前是基于 Clang 4.0 的。下面笔者和大家一起将其移植到 Clang 6.0.0 中并分析其原理。

将开源的基于 Clang 4.0 版本的代码下载下来，然后分别复制以下文件到 Clang 6.0.0 的对应目录下。

- include/llvm/CryptoUtils.h
- lib/Transforms/Obfuscation/BogusControlFlow.cpp
- lib/Transforms/Obfuscation/CMakeLists.txt
- lib/Transforms/Obfuscation/CryptoUtils.cpp
- lib/Transforms/Obfuscation/Flattening.cpp
- lib/Transforms/Obfuscation/SplitBasicBlocks.cpp
- lib/Transforms/Obfuscation/Substitution.cpp
- lib/Transforms/Obfuscation/Utils.cpp
- include/llvm/Transforms/Obfuscation/BogusControlFlow.h

- include/llvm/Transforms/Obfuscation/Flattening.h
- include/llvm/Transforms/Obfuscation/Split.h
- include/llvm/Transforms/Obfuscation/Substitution.h
- include/llvm/Transforms/Obfuscation/Utils.h

编辑文件 lib/Transforms/CMakeLists.txt，具体如下。

```
add_subdirectory(Hello)
add_subdirectory(ObjCARC)
add_subdirectory(Coroutines)
add_subdirectory(Obfuscation)
```

把 lib/Transforms/Obfuscation/CMakeLists.txt 中的“add_llvm_library”改成“add_llvm_loadable_module”，编译成动态库，通过 opt 加载调试（后面加到 PassManager 中后，编译成静态库，同样可以通过 opt 加载调试）。

OLLVM 中主要包括以下 3 个 Pass，用于进行不同程序的混淆。

- BogusControlFlow：增加虚假的控制流程和无用的代码。
- Flattening：使代码扁平化。
- Substitution：增加运算表达式的复杂程度。

打开 Xcode，按“Command + B”快捷键，编译后找到 LLVMObfuscation 的 Scheme，然后编译生成动态库。

下面分析 BogusControlFlow。编辑 opt scheme 启动参数。为了打印调试信息，需要添加参数 -debug，为了保证每次都发生混淆，指定为 100%混淆，如图 8-13 所示。

在调试输出中会打印流程图。为了查看 dot 格式的流程图，需要下载并安装 Graphviz（http://www.graphviz.org/），还要把 dot 后缀文件的默认打开方式设置成 Graphviz。然后，在 runOnFunction 函数处下断点，该函数会在处理方法时自动调用。

在 runOnFunction 中，先判断参数是否合法，再判断是否需要混淆，从而指定全混淆或者在满足某些函数属性的情况下混淆。在这里，可以在函数上增加属性，具体如下。

```
#include <stdio.h>

int add(int x, int y) __attribute((__annotate__(("bcf"))){
    return x + y;
}
int main(){
    printf("%d",add(3,4));
```

```
    return 0;
}
```

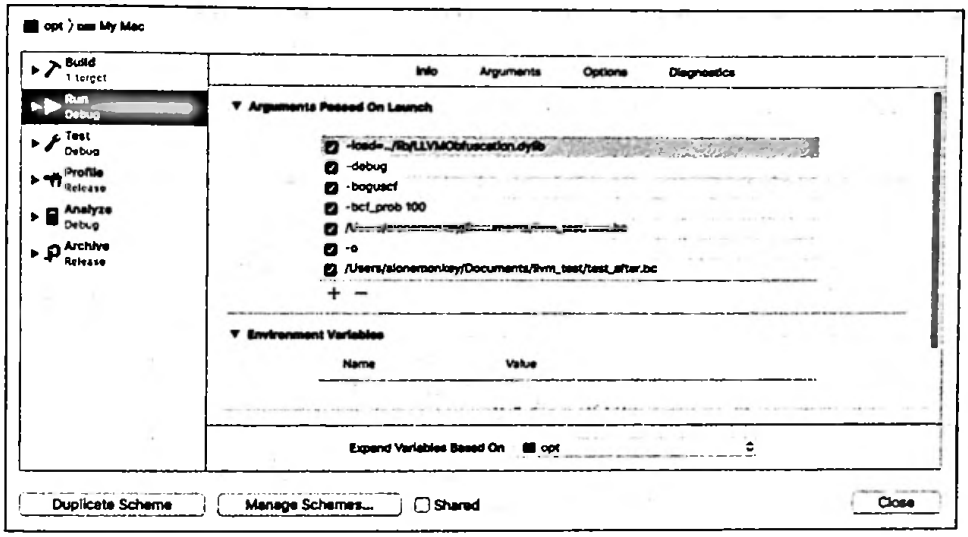


图 8-13　编辑 opt scheme 启动参数加载 BogusControlFlow

如果要混淆，应执行 bogus。该函数首先把方法中所有的 Block 放到一个 list 中，然后分别取出，调用 addBogusFlow 增加虚假控制流程。addBogusFlow 首先调用 createAltered BasicBlock，复制原来的 BasicBlock，然后修复其变量，引用并插入一些脏指令，接着创建一个恒为 true 的条件跳转到原 BasicBlock。false 的分支跳转到复制的 BasicBlock，复制的 BasicBlock 跳转回原 BasicBlock。最后，创建一个恒为 true 的条件跳转到原 BasicBlock 的结束位置，false 的分支跳转到复制的 BasicBlock。

处理前和处理后的流程对比如图 8-14 所示。看这幅图就一目了然了。

另外，在调试过程中可以通过如下命令打印对应的结果信息及浏览当前函数的流程图。

```
(lldb) po originalBB->dump()

originalBB:                                       ; preds = %originalBBalteredBB, %entry
  %x.addr = alloca i32, align 4
  %y.addr = alloca i32, align 4
  store i32 %x, i32* %x.addr, align 4
  store i32 %y, i32* %y.addr, align 4
  %0 = load i32, i32* %x.addr, align 4
  %1 = load i32, i32* %y.addr, align 4
  %add = add nsw i32 %0, %1
```

```
  ret i32 %add

(lldb) po F.viewCFG()
Writing '/var/folders/7q/dxk4jpyn59x40lt6p998dkg40000gn/T/cfg_Z3addii-063626.dot'...
done.
Trying 'open' program... Remember to erase graph file:
/var/folders/7q/dxk4jpyn59x40lt6p998dkg40000gn/T/cfg_Z3addii-063626.dot
```

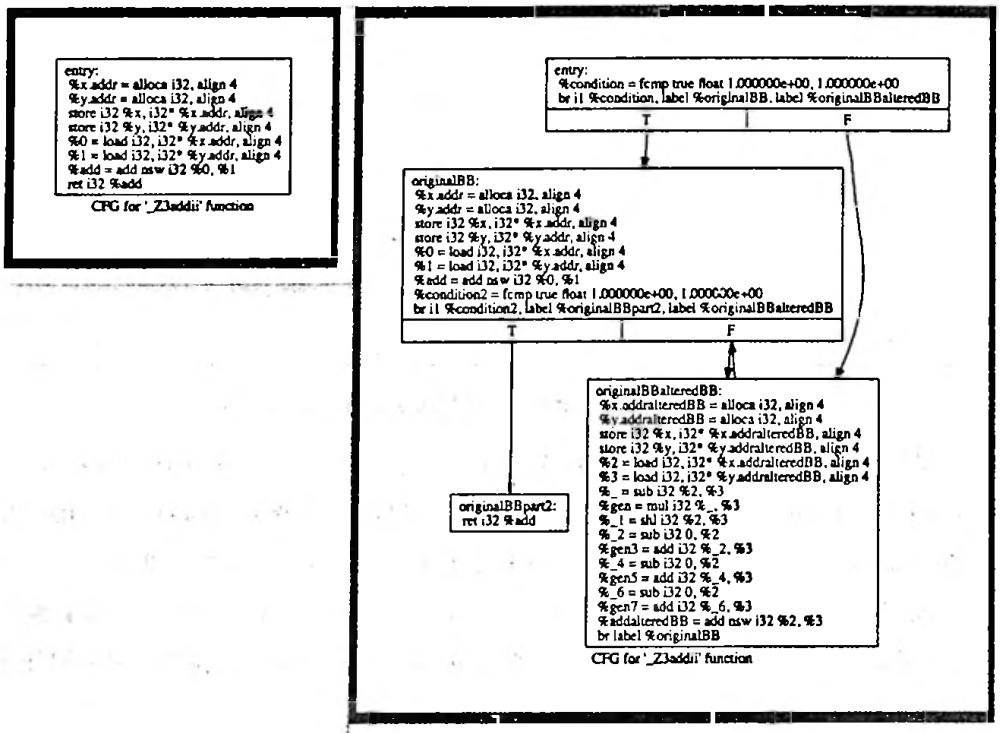


图 8-14 bcf 混淆前和混淆后的流程

调用 doF，将上面恒为 true 的条件换成“(y<10||x*(x+1)%2==0)”。运行结束，可以得到混淆处理之后的 bitcode。使用如下命令可以将其转换成可读的 bitcode 代码。

```
~/Documents/build/Debug/bin/llvm-dis test_after.bc -o test_after.ll
```

再看一下 Flattening 代码扁平化功能的实现。编辑并运行如下参数。

```
opt -load=../lib/LLVMObfuscation.dylib -debug -flattening path/to/test.bc -o
path/to/test_after.bc
```

为了看到扁平化的效果，修改源文件如下。

```
#include <stdio.h>

int add(int x, int y) __attribute((__annotate__(("fla")))){
    if(x > y){
     return x - y;
    }else{
     return x + y;
    }
}
int main(){
    printf("%d",add(3,4));
    return 0;
}
```

生成 bitcode 之后，使用 Xcode 进行调试。一开始还是进入 runOnFunction，判断是否需要混淆。如果需要混淆，就调用 flatten 方法，保存所有的 BasicBlock，如果个数小于等于 1 就直接返回，然后移除第 1 个 BasicBlock。接着，判断第 1 个 BasicBlock 是不是条件判断或者分支，如果是就使用 splitBasicBlock 将其分开，并移除第 1 个 BasicBlock。跳转指令的执行结果如图 8-15 所示。

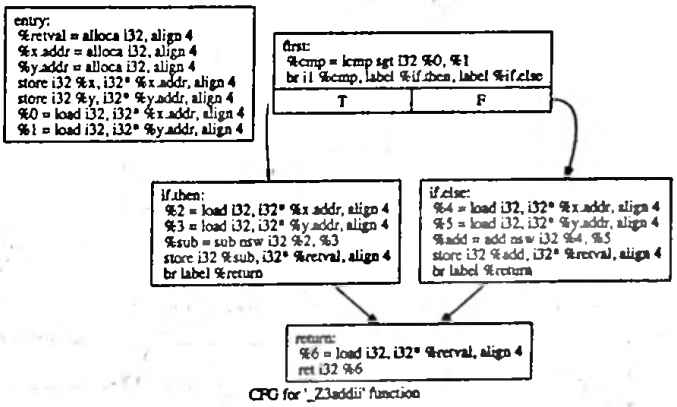


图 8-15　将第 1 个 BasicBlock 单独拿出来

分别创建一个 loopEntry 和一个 loopEnd 的 BasicBlock，分别将第 1 个 BasicBlock 和 loopEnd 跳转到 loopEntry，创建一个指向 loopEnd 的 switchDefault，并将 loopEntry 指向 switchDefault，结果如图 8-16 所示。

将之前保存的 BasicBlock 分别添加到 switch 分支中，操作结果如图 8-17 所示。

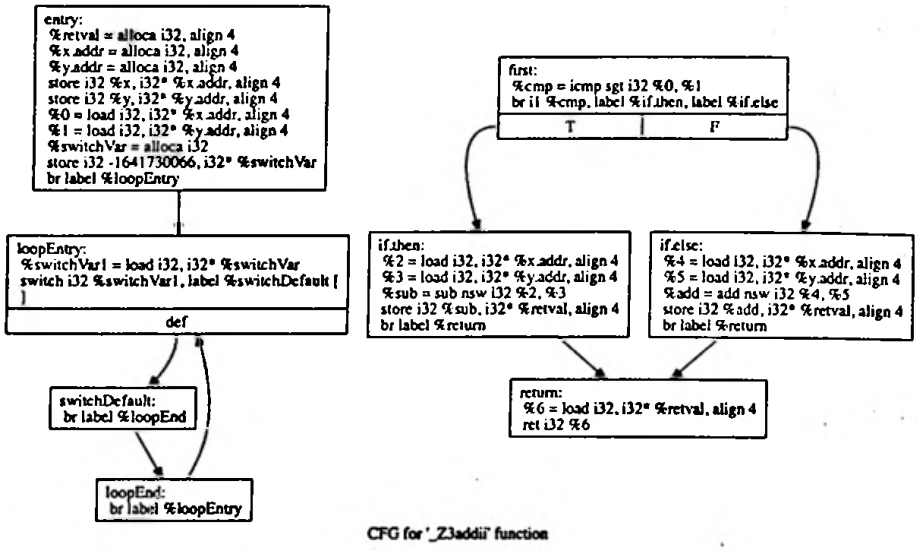


图 8-16 创建一个 switch 结构

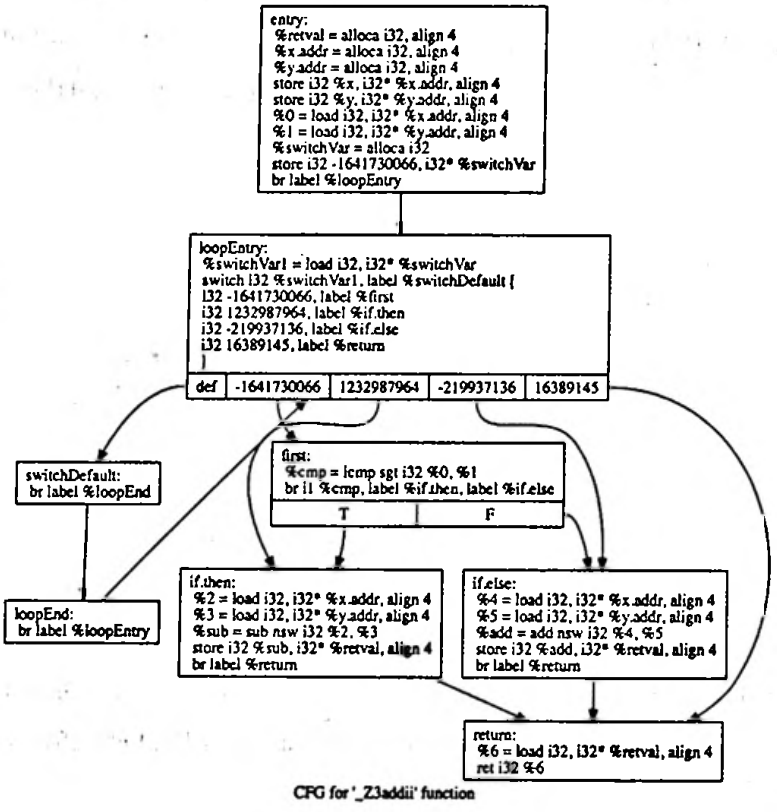


图 8-17 将 BasicBlock 分别添加到 switch 分支中

将 switch 中所有的 BasicBlock 指向 loopEnd 并调整 switch case，处理结果如图 8-18 所示。最后一个 Substitution 将一些计算指令替换成一些复杂的转换表达式，结果都是一样的，只是提高了复杂程度。但是，这种混淆在开启编译器优化之后就会失效（除非指定了不优化函数）。

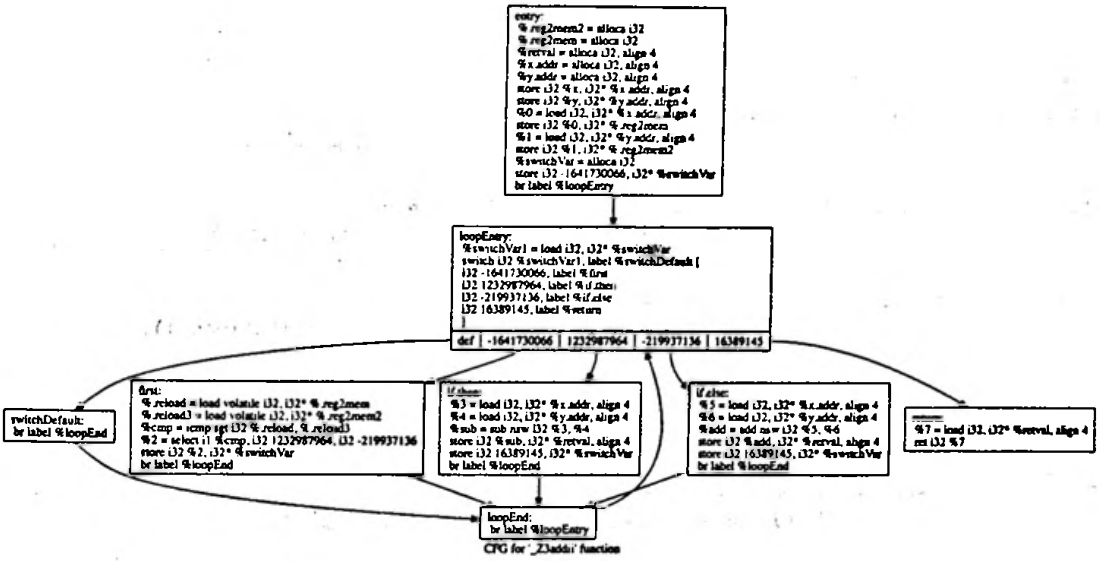


图 8-18　代码扁平化的最终处理结果

8.4.5　替换 Xcode 编译器

在尝试了 opt 调试，了解了 OLLVM 的原理之后，便要将其集成到 Xcode 中，并在编译时使用带混淆工程的 Clang 来编译工程了。要想通过 Clang 加载 Pass，就不能采取上面这种动态库的方式，而要编译成静态库并加入由 PassManager 的管理机制定义的 Clang 传入的参数去调用 Pass，步骤如下。

①编辑 lib/Transforms/Obfuscation/CMakeLists.txt 文件，将“add_llvm_loadable_module”改回“add_llvm_library”。

②复制 lib/Transforms/Obfuscation/LLVMBuild.txt 文件到对应位置。

③在 lib/Transforms/IPO/LLVMBuild.txt 文件的 required_libraries 中加入 Obfuscation。

④在 lib/Transforms/LLVMBuild.txt 文件的 subdirectories 中加入 Obfuscation。

⑤编辑 lib/Transforms/IPO/PassManagerBuilder.cpp 文件，导入头文件及指定参数，代码如下。

```
#include "llvm/Transforms/Obfuscation/BogusControlFlow.h"
```

```
#include "llvm/Transforms/Obfuscation/Flattening.h"
#include "llvm/Transforms/Obfuscation/Split.h"
#include "llvm/Transforms/Obfuscation/Substitution.h"
#include "llvm/CryptoUtils.h"
```

增加混淆参数，代码如下。

```
// Flags for obfuscation
static cl::opt<bool> Flattening("fla", cl::init(false),
                                cl::desc("Enable the flattening pass"));

static cl::opt<bool> BogusControlFlow("bcf", cl::init(false),
                                      cl::desc("Enable bogus control flow"));

static cl::opt<bool> Substitution("sub", cl::init(false),
                                  cl::desc("Enable instruction substitutions"));

static cl::opt<std::string> AesSeed("aesSeed", cl::init(""),
                                    cl::desc("seed for the AES-CTR PRNG"));

static cl::opt<bool> Split("split", cl::init(false),
                           cl::desc("Enable basic block splitting"));
```

在 PassManagerBuilder::PassManagerBuilder() 中初始化随机生成器，代码如下。

```
PassManagerBuilder::PassManagerBuilder() {
    ......
    PrepareForThinLTO = EnablePrepareForThinLTO;
    PerformThinLTO = false;
    DivergentTarget = false;
    // Initialization of the global cryptographically
    // secure pseudo-random generator
    if(!AesSeed.empty()) {
        if(!llvm::cryptoutils->prng_seed(AesSeed.c_str()))
            exit(1);
    }
}
```

在 PassManagerBuilder::populateModulePassManager 中增加混淆的 Pass，代码如下。

```
void PassManagerBuilder::populateModulePassManager(
    ......

    MPM.add(createForceFunctionAttrsLegacyPass());
```

```
    MPM.add(createSplitBasicBlock(Split));
    MPM.add(createBogus(BogusControlFlow));
    MPM.add(createFlattening(Flattening));
    ......

    if (PrepareForThinLTO)
      // Rename anon globals to be able to export them in the summary.
      MPM.add(createNameAnonGlobalPass());

    MPM.add(createSubstitution(Substitution));
    return;
  }
  ......
  MPM.add(createSubstitution(Substitution));
  addExtensionsToPM(EP_OptimizerLast, MPM);
}
```

重新编译生成 Clang，并使用如下命令进行测试，看看是否有混淆效果。如果有混淆效果，就说明 Clang 的集成成功，下一步就是替换 Xcode 的 Clang 了。

```
~/Documents/build/Debug/bin/clang test.mm -o test -mllvm -sub -mllvm -fla -mllvm -bcf
```

通过如下命令增加 Xcode 使用的编译工具链。

```
$ cd
/Applications/Xcode.app/Contents/PlugIns/Xcode3Core.ideplugin/Contents/SharedSupport/De
veloper/Library/Xcode/Plug-ins/
$ sudo cp -r Clang\ LLVM\ 1.0.xcplugin/ Obfuscator.xcplugin
$ cd Obfuscator.xcplugin/Contents/
$ sudo plutil -convert xml1 Info.plist
$ sudo vim Info.plist    //或者拖入 Sublime Text 进行编辑
```

修改如下值。

```
<string>com.apple.compilers.clang</string> ->
<string>com.apple.compilers.obfuscator</string>
<string>Clang LLVM 1.0 Compiler Xcode Plug-in</string> -> <string>Obfuscator Xcode
Plug-in</string>
```

执行如下命令。

```
$ sudo plutil -convert binary1 Info.plist
$ cd Resources/
$ sudo mv Clang\ LLVM\ 1.0.xcspec Obfuscator.xcspec
```

```
$ sudo vim Obfuscator.xcspec
```

修改如下值。

```
Identifier = "com.apple.compilers.llvm.clang.1_0.compiler"; -> Identifier =
"com.apple.compilers.llvm.obfuscator.6_0";
Name = "Apple LLVM 9.0"; -> Name = "Obfuscator 6.0";
Description = "Apple LLVM 9.0 compiler"; -> Description = "Obfuscator 6.0 compiler";
Vendor = Apple; -> Vendor = Obfuscator;
Version = "9.0"; -> Version = "6.0";
ExecPath = "clang"; -> ExecPath = "/path/to/obfuscator_bin/clang";
```

修改 English 环境配置，具体如下（也可以不修改）。

```
$ cd English.lproj/
$ sudo mv Apple\ LLVM\ 9.0.strings "Obfuscator 6.0.strings"
$ sudo plutil -convert xml1 Obfuscator\ 6.0.strings
$ sudo vim Obfuscator\ 6.0.strings
```

修改如下值。

```
<key>Description</key>
<string>Apple LLVM 9.0 compiler</string> -> <string>Obfuscator 6.0 compiler</string>
<key>Name</key>
<string>Apple LLVM 9.0</string> -> <string>Obfuscator 6.0</string>
<key>Vendor</key>
<string>Apple</string> -> <string>Obfuscator</string>
<key>Version</key>
<string>9.0</string> -> <string>6.0</string>
```

最后，执行如下命令。

```
sudo plutil -convert binary1 Obfuscator\ 6.0.strings
```

改完之后重启 Xcode，并在 Build Settings 界面设置 Compiler for C/C++/Objective-C 为 Obfuscator 6.0，在 Other C Flags 里面增加混淆参数 -mllvm -bcf -mllvm -sub -mllvm -fla。重新编译，发现报如下错误。

```
clang: error: cannot specify -o when generating multiple output files
```

通过排查，发现是 Xcode 9 增加了一个 SWIFT_INDEX_STORE_ENABLE 的参数，导致编译参数里面多了 -index-store-path，而开源的 Clang 是不识别这个参数的，所以需要在 Build Settings

界面找到“Enable Index-While-Building Functionality”，将其改为“NO”。重新编译，结果还是报错，具体如下。

```
mismatched subprogram between llvm.dbg.declare variable and !dbg attachment
  call void @llvm.dbg.declare(metadata %0** %self.addralteredBB, metadata !77,
metadata !DIExpression()), !dbg !68
label %originalBBalteredBB
void (%0*, i8*, [4 x i32])* @"\01-[ContactMe drawRect:]"
!77 = !DILocalVariable(name: "self", arg: 1, scope: !78, type: !67, flags: DIFlagArtificial
| DIFlagObjectPointer)
!78 = distinct !DISubprogram(name: "-[ContactMe drawRect:]", scope: !1, file: !1, line: 16,
type: !54, isLocal: true, isDefinition: true, scopeLine: 16, flags: DIFlagPrototyped,
isOptimized: false, unit: !79, variables: !2)
!68 = !DILocation(line: 0, scope: !53)
......
```

要想解决这个问题，肯定是使用调试的方法来得快。首先从 Xcode 工程中找到出错的编译参数，如图 8-19 所示，然后将该参数添加到 Clang 的 Scheme 的启动参数中并运行 Clang。正常的结果应该是触发 Assert，程序中断，但实际运行的结果并不如我们所料。所以，在前面使用 opt 调试 Pass 时，不是直接运行 Clang 并在 Pass 下断点的方法不可行，而是在 Clang 执行参数时会生成一系列新的参数并在新的进程中执行。这就是在当前进程中不能直接调试 Clang 里面的 Pass 的原因。

尽管可以从报错信息的 Program arguments 中获取真正的参数，但笔者还是带大家梳理一下 Clang 的调用流程。找到 Clang 的入口函数，即 driver.cpp 文件中的 main 函数，在该函数处下断点。调试程序的执行流程如下。

①main

②TheDriver.ExecuteCompilation

③C.ExecuteJobs

④ExecuteCommand

⑤C.Execute

⑥llvm::sys::ExecuteAndWait

⑦Execute

⑧posix_spawn

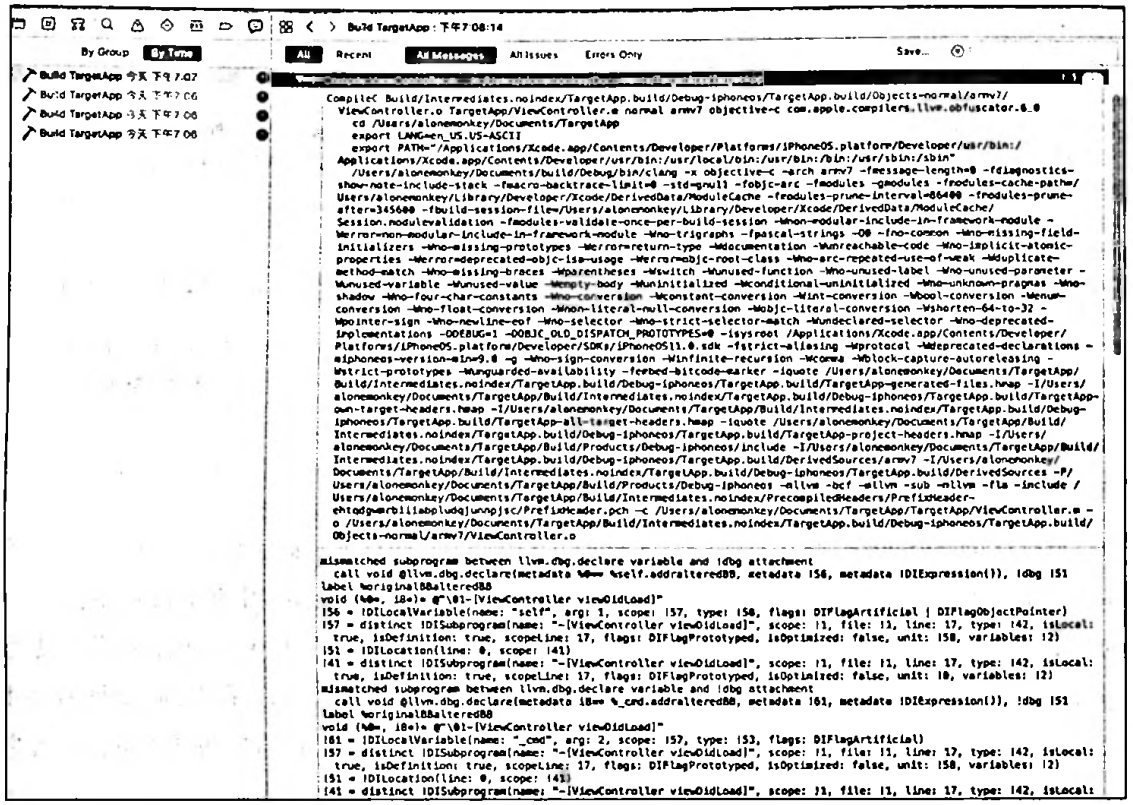


图 8-19 从 Xcode 获取详细的编译参数

调用 Program.inc 的 Execute 函数中的 posix_spawn 来启动一个新的进程，Args 就是它的新参数。为了获取新的参数，可以在执行该函数之前打印参数列表，具体如下。

```
const char** tmp = Args;

while(*tmp){
    errs() << *(tmp++) << " ";
}

errs() << "\n";

int Err = posix_spawn(&PID, Program.str().c_str(), FileActions,
                      /*attrp*/nullptr, const_cast<char **>(Args),
                      const_cast<char **>(Envp));
```

将打印出来的参数填到 Clang Scheme 的启动参数中。再次运行 Clang，就可以看到 Assert 的断言了，如图 8-20 所示。从断言来看，报错是因为以下代码中的调试信息和当前程序不匹配。在前面分析 BogusControlFlow.cpp 的源代码时，该代码片段是被复制出来的，如果将其中的一些信息重新映射到复制出来的 BasicBlock，会因复制出来的 BasicBlock 中的调试信息和当前不匹配而报错。因此，可以查看官方文档 https://llvm.org/docs/SourceLevelDebugging.html#id13，先将调试信息的重映射过滤掉试试。

```
(lldb) po II->dump()
  call void @llvm.dbg.declare(metadata %1** %sender.addralteredBB, metadata !80,
metadata !DIExpression()), !dbg !72
```

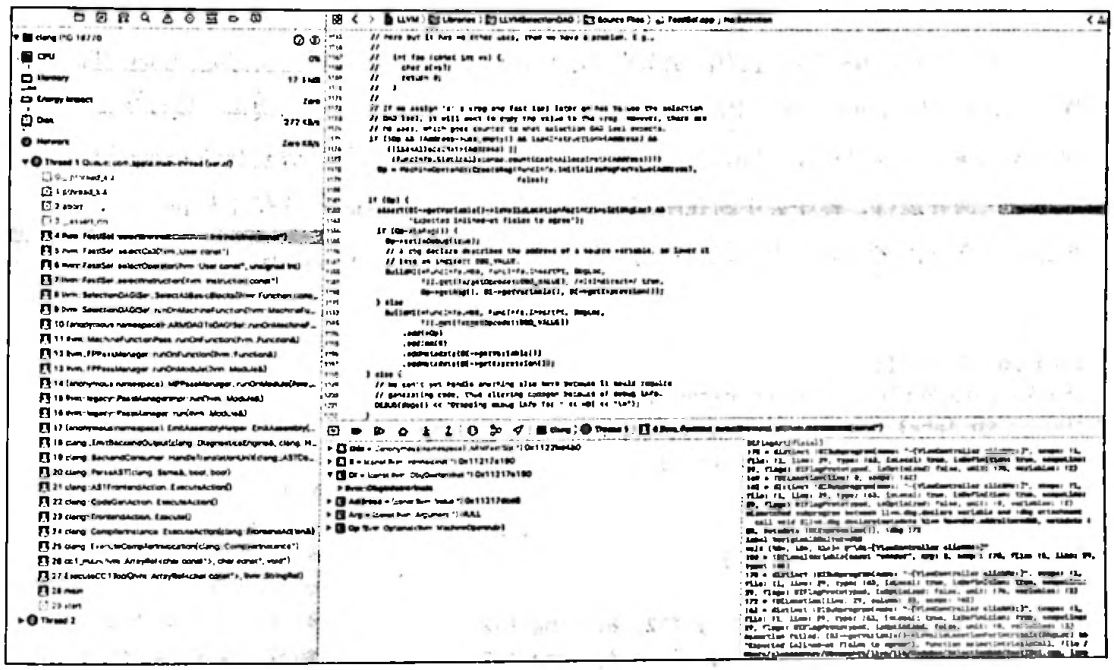


图 8-20　运行 Clang 触发 Assert 断言

修改 BogusControlFlow.cpp 文件中的 createAlteredBasicBlock 方法并导入头文件，代码如下。

```
#include "llvm/IR/IntrinsicInst.h"

virtual BasicBlock* createAlteredBasicBlock(BasicBlock * basicBlock,
    const Twine &  Name = "gen", Function * F = 0){
    ......
    for (BasicBlock::iterator i = alteredBB->begin(), e = alteredBB->end() ; i != e; ++i){
```

```
    if(isa<llvm::DbgInfoIntrinsic>(i)){
        ji++;
        continue;
    }
    ......
```

然后，重新编译 Clang。再到刚刚出错的项目里面重新编译，报了如下错误。

```
The unwind destination does not have an exception handling instruction!
  invoke void @objc_exception_rethrow()
          to label %invoke.cont unwind label %lpad, !dbg !59
Block containing LandingPadInst must be jumped to only by the unwind edge of an invoke.
  %1408 = landingpad { i8*, i32 }
          cleanup, !dbg !61
```

先被排查出来是 bcf 混淆过程中的错误。为了保证每次都发生混淆，将 BogusControlFlow.cpp 文件中的 defaultObfRate 设置为 100。复制错误信息，在 LLVM 工程里面搜索，发现该提示代码出现在 Verifier.cpp 文件的 visitInvokeInst 方法中。将打印出来的详细参数复制到 Clang 的 scheme 输入参数里面并运行，程序会中断在刚刚设下的断点处。因为这时还没有进入 bcf pass 处理的函数，所以在混淆之前会对中间代码进行验证。打印当前 Function 及 instruction 的结构，具体如下。

```
(lldb) po II.dump()
  invoke void @objc_exception_rethrow()
          to label %invoke.cont unwind label %lpad, !dbg !60
(lldb) po II.getUnwindDest()->dump()

lpad:                                             ; preds = %finally.rethrow
  %12 = landingpad { i8*, i32 }
          cleanup, !dbg !62
  %13 = extractvalue { i8*, i32 } %12, 0, !dbg !62
  store i8* %13, i8** %exn.slot, align 4, !dbg !62
  %14 = extractvalue { i8*, i32 } %12, 1, !dbg !62
  store i32 %14, i32* %ehselector.slot, align 4, !dbg !62
  %finally.endcatch = load i1, i1* %finally.for-eh, align 1, !dbg !60
  br i1 %finally.endcatch, label %finally.endcatch1, label %finally.cleanup.cont, !dbg !60

(lldb) po F.viewCFG()        //将堆栈切换到 7 llvm::InstVisitor
Writing '/var/folders/h3/8n169g610_g69k50z7g_q4gc0000gp/T/cfg-[ViewController
viewDidLoad]-446905.dot'...  done.
Trying 'open' program... /usr/bin/open
/var/folders/h3/8n169g610_g69k50z7g_q4gc0000gp/T/cfg-[ViewController
```

```
viewDidLoad]-446905.dot
Remember to erase graph file:
/var/folders/h3/8n169g610_g69k50z7g_q4gc0000gp/T/cfg-[ViewController
viewDidLoad]-446905.dot
(lldb)
```

验证通过之后才会进行 bcf 混淆。完成 bcf 混淆后，也会再次进行验证（此时的验证会出现上面的错误）。同样，将当前的 Function 及 instruction 的结构打印出来，具体如下。

```
The unwind destination does not have an exception handling instruction!
  invoke void @objc_exception_rethrow()
          to label %invoke.cont unwind label %lpad, !dbg !60
(lldb) po II.dump()
  invoke void @objc_exception_rethrow()
          to label %invoke.cont unwind label %lpad, !dbg !60
(lldb) po II.getUnwindDest()->dump()

lpad:                                             ; preds = %originalBBpart211
  %304 = load i32, i32* @x
  %305 = load i32, i32* @y
  %306 = add i32 %304, -2013955903
  %307 = sub i32 %306, 1
  %308 = sub i32 %307, -2013955903
  %309 = sub i32 %304, 1
  %310 = mul i32 %304, %308
  %311 = urem i32 %310, 2
  %312 = icmp eq i32 %311, 0
  %313 = icmp slt i32 %305, 10
  %314 = xor i1 %312, true
  %315 = xor i1 %313, true
  %316 = xor i1 true, true
  %317 = and i1 %314, true
  %318 = and i1 %312, %316
  %319 = and i1 %315, true
  %320 = and i1 %313, %316
  %321 = or i1 %317, %318
  %322 = or i1 %319, %320
  %323 = xor i1 %321, %322
  %324 = or i1 %314, %315
  %325 = xor i1 %324, true
  %326 = or i1 true, %316
  %327 = and i1 %325, %326
  %328 = or i1 %323, %327
  %329 = or i1 %312, %313
```

```
  br i1 %328, label %originalBB33, label %originalBB33alteredBB

(lldb) po F.viewCFG()
Writing '/var/folders/h3/8n169g610_g69k50z7g_q4gc0000gp/T/cfg-[ViewController
viewDidLoad]-2077fb.dot'...  done.
Trying 'open' program... /usr/bin/open
/var/folders/h3/8n169g610_g69k50z7g_q4gc0000gp/T/cfg-[ViewController
viewDidLoad]-2077fb.dot
Remember to erase graph file:
/var/folders/h3/8n169g610_g69k50z7g_q4gc0000gp/T/cfg-[ViewController
viewDidLoad]-2077fb.dot
(lldb)
```

问题来了：由于 bcf 会调用 splitBasicBlock 将 BasicBlock 分割，然后插入条件判断，导致 II 跳转的目标不是原来的 landingpad，而是一个条件判断。所以，如下断言会失败。

```
Assert(
    II.getUnwindDest()->isEHPad(),
    "The unwind destination does not have an exception handling instruction!",
    &II);
```

要解决这个问题，可以在分割时进行 isEHPad 指令判断。将 addBogusFlow 的代码修改为如下内容。

```
BasicBlock::iterator i1 = basicBlock->begin();
if(basicBlock->getFirstNonPHIOrDbgOrLifetime())
    i1 = (BasicBlock::iterator)basicBlock->getFirstNonPHIOrDbgOrLifetime();

for (BasicBlock::iterator end = basicBlock->end(); i1 != end; i1++) {
    if(!i1->isEHPad()){
        break;
    }
}

if(i1 == basicBlock->end())
    return;

Twine *var;
var = new Twine("originalBB");
BasicBlock *originalBB = basicBlock->splitBasicBlock(i1, *var);
```

另外，如果项目中包含 C++ 文件，在编译时会出现如下错误。

```
/Users/monkey/Documents/iosreversebook/TestOLLVM/TestOLLVM/CPlusPlusFile.cpp:10:10:
fatal error: 'string' file not found
#include <string>
         ^~~~~~~~
1 error generated.
```

在 Build Settings 的 Header Search Paths 里增加 /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/ 即可解决此问题。

最后，看看生成的文件中是不是真的发生了混淆。使用 Hopper 打开可执行文件，如图 8-21 所示。为了使效果明显一些，笔者特意将混淆比例设置为 100%，将混淆次数设置为 3。

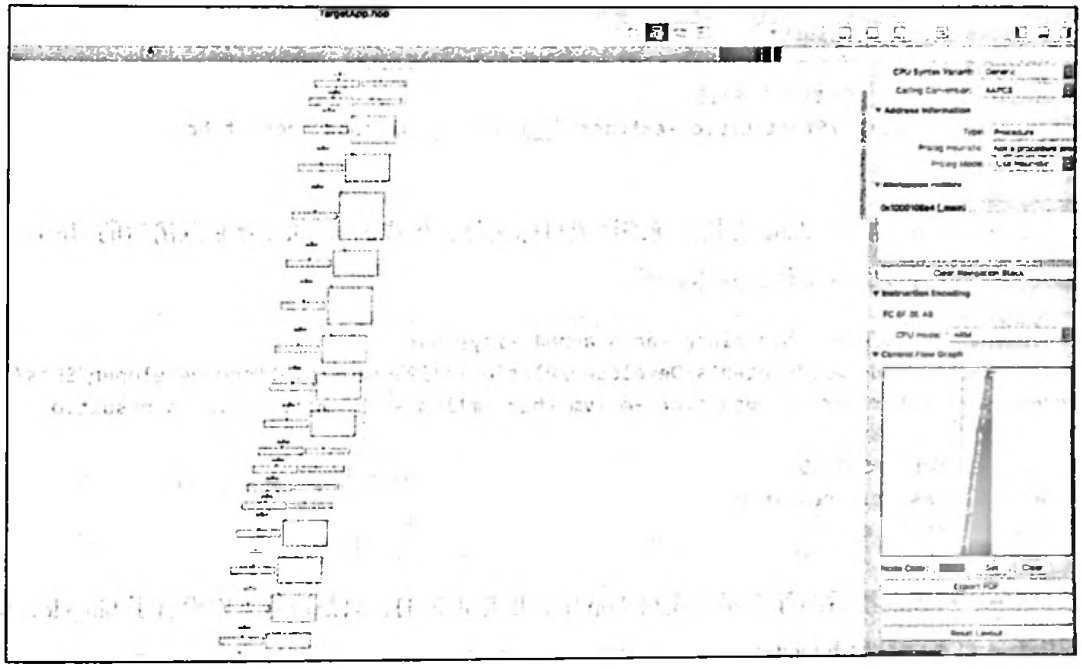


图 8-21　使用 Hopper 打开混淆后的文件

当然，开源的 OLLVM 还有很多问题和可以优化的地方，希望读者在学习过程中能够自行调试解决这些问题，并在 OLLVM 的基础上改进和增加自己的混淆逻辑。

8.4.6　静态库混淆

在本节中，我们解讲一个基于编译器混淆的实例。

通过以上分析可知，编译器混淆是基于中间代码（bitcode）进行的。那么，编译生成一个

带 bitcode 的静态库是不是就可以直接提取其中的 bitcode 进行混淆了呢？答案是肯定的。这样就不需要通过源文件进行编译了，只需要提供一个带 bitcode 的静态库文件。

新建一个静态库工程。为了让其在非 Archive 模式下也生成 bitcode，需要在 Build Settings 的 Other C Flags 中增加 -fembed-bitcode 参数。然后，生成静态库，并将其拖入 MachOView。这时可以看到一个 __LLVM,__bitcode 的 section，其中就是生成的 bitcode 代码。使用如下命令将这个 section 里面生成的 bitcode 代码提取出来。

```
➜  StaticLib ls
libStaticLib.a
➜  StaticLib mkdir objects
➜  StaticLib cd objects
➜  objects ar -x ../libStaticLib.a
➜  objects ls
StaticLib.o      __.SYMDEF SORTED
➜  objects segedit ./StaticLib.o -extract "__LLVM" "__bitcode" result.bc
➜  objects
```

使用带混淆功能的 Clang 将其重新编译成目标文件，生成的 result.o 就是混淆后的 object 文件。将其重新打包成 .a 文件，具体如下。

```
~/Documents/build/Debug/bin/clang -arch arm64 -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/i
PhoneOS.sdk -fobjc-arc -c result.bc -mllvm -bcf -mllvm -sub -mllvm -fla -o result.o

➜  rm __.SYMDEF\ SORTED
➜  ar -crs result.a result.o
➜  ranlib result.a
```

最后，将该混淆后的静态库集成到 App 中，也是正常的，只是该静态库没有了 bitcode，所以主 App 也不能开启 bitcode。

8.5 本章总结

本章针对之前分析应用时使用的一些方案和技术的切入点，制定了一些特定的保护方案，以提高应用被分析的难度，防止自己的应用在发布后被破解。但是，有攻就有防，有防也必然有破解之法。因此，本章介绍的安全保护方案无法保证发布的应用绝对不会被破解。当然，对某些应用来说，如果涉及利益，提高破解难度就相当有必要了。希望读者在开发应用时能够从分析和保护的角度提高安全意识，保证代码和数据的安全性。