MASTG

移动应用安全

测试指南

版本 v1.5.0

Sven Schleier

Bernhard Mueller

Carlos Holguera

Jeroen Willemsen

OWASP

OWASP Mobile Application Security Testing Guide (MASTG)

v1.5.0 发布于 2022 年 9 月 6 日

发布说明: https://github.com/OWASP/owasp-mastg/releases/tag/v1.5.0

在线版本请访问 https://github.com/OWASP/owasp-mastg/releases/tag/v1.5.0

基于 OWASP 移动应用安全验证标准(MASVS) v1.4.2

在线版本请访问 https://github.com/OWASP/owasp-masvs/releases/tag/v1.4.2

OWASP MASTG 和 MASVS 是 OWASP 移动应用安全（MAS）项目的一部分。

https://mas.owasp.org

英文版 ISBN: 978-1-257-96636-3

封面设计：Carlos Holguera

中文翻译：yolylight@ansion

目录

1. 前言

欢迎浏览《OWASP 移动应用安全测试指南》。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 Android 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-mastg/

松鼠（名词复数）：松鼠属的任何树栖啮齿动物，如红松鼠（S. vulgaris）或灰松鼠（S. carolinensis），有一个灌木状的尾巴，以坚果、种子等为食。

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 移动应用安全测试指南（MASTG）》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 移动应用安全测试指南》第一条：不要只遵守《OWASP 移动应用安全测试指南》。想要在移动应用安全方面做到真正卓越，需要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！编写您自己的应用，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 MASTG 作点贡献！或许，像他们说的：“提个版本合并请求吧”。

2. 卷首语

OWASP MAS
Mobile Application Security
Project

2.1. 关于《OWASP 移动应用安全测试指南》

OWASP 移动应用安全测试指南（MASTG）是 OWASP 移动应用安全（MAS）旗舰项目的一部分，是一本涵盖移动应用安全分析过程、技术和工具的综合手册，也是一套详尽的测试案例，用于验证 OWASP 移动应用安全验证标准（MASVS）中列出的要求，为完整和一致的安全测试提供一个基线。

OWASP MASVS 和 MASTG 得到了以下平台供应商和标准化组织、政府和教育机构的信任。

android CREST NIST National Institute of Standards and Technology U.S. Department of Commerce Bundesamt für Sicherheit in der Informationstechnik ioXt internet of secure things ...

2.2. 原文作者

Bernhard Mueller

Bernhard 是一位擅长于黑进任何系统的网络安全专家。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 Server、Adobe Flash Player、IBM Director、Cisco VOIP 以及 ModSecurity。您说得出名字的，他可能都已经攻破它至少一次了。美国 BlackHat 给他颁发了一个 Pwine Award 最佳研究奖以赞扬他在移动安全领域的开创性工作。

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应用。他还是一位安全工程师，为很多项目的提供端到端支持，实现从 SDLC 到 “构建安全” 的过程。他在一些本地和国际的沙龙和会议上发表了演讲，还在指导关于 Web 应用程序和移动应用安全的实践研讨会。

Jeroen Willemsen

Jeroen 是一名首席安全架构师，对移动安全和风险管理充满热情。他曾作为安全教练、安全工程师和全栈开发人员为公司提供支持，这使他成为了所有行业的专家。他喜欢解释技术课题：从安全问题到编程挑战。

Carlos Holguera

Carlos 是一名移动安全研究工程师，在移动应用和嵌入式系统（如汽车控制单元和物联网设备）的安全测试领域积累了多年的实践经验。他热衷于移动应用程序的逆向工程和动态插桩，并不断学习和分享他的知识。

2.3. 联合作者

联合作者们持续贡献高质量的内容，并在 GitHub 仓库中至少有 2000 个添加记录。

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。

Jeroen Beckers

Jeroen 是一名移动安全负责人，负责移动安全项目的质量保证和所有移动事物的研发。虽然他的职业生涯是从程序员开始的，但他发现把东西拆开比把东西组装起来更有趣，于是很快就转到了安全领域。自从他关于 Android 安全的硕士论文发表以来，Jeroen 一直对移动设备和它们的（不）安全感兴趣。他喜欢与其他人分享他的知识，这一点从他在学院、大学、客户和会议上的许多讲座和培训中可以看出。

Vikas Gupta

Vikas 是一位经验丰富的网络安全研究员，在移动安全方面有专长。在他的职业生涯中，他曾为不同行业的应用程序提供安全保障，包括科技、银行和政府。他喜欢逆向工程，特别是混淆的原生代码和密码学。他拥有安全和移动计算的硕士学位，以及 OSCP 证书。他总是愿意分享他的知识和交流想法。

2.4. 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。

2.4.1. 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

2.4.2. 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. 更新日志

我们所有的更新日志都可以在 OWASP MASTG 的 GitHub 仓库中找到，请查看发布页面：

https://github.com/OWASP/owasp-mastg/releases

2.6. 免责声明

在利用 MASTG 材料对移动应用程序进行任何测试之前，请咨询贵国的法律。请不要用 MASTG 中描述的任何东西来违反法律。

我们的[行为准则]有进一步的细节：https://github.com/OWASP/owasp-mastg/blob/master/CODEOFCONDUCT.md

OWASP 感谢众多作者、审稿人和编辑在制定本指南时的辛勤工作。如果您有任何意见或建议，请与我们联系：https://mas.owasp.org/#connect-with-us

如果你发现任何不一致的地方或错别字，请在 OWASP MASTG Github 仓库中提交一个问题：https://github.com/OWASP/owasp-mastg

2.7. 版权和许可

2.8. OWASP MASVS 和 MASTG 的采用

OWASP MASVS 和 MASTG 得到了以下平台供应商和标准化组织、政府和教育机构的信任。

2.8.1. 移动平台供应商

2.8.1.1. Google Android

自 2021 年以来，Google 对 OWASP 移动安全项目（MASTG/MASVS）表示支持，并开始通过应用防御联盟（ADA）及其 MASA（移动应用安全评估）项目为 MASVS 重构过程提供持续和高价值的反馈。

通过 MASA，Google 已经认识到利用全球公认的移动应用安全标准对移动应用生态系统的重要性。开发者可以直接与授权实验室合作伙伴合作，启动安全评估。Google 将认可那些根据 MASVS 一级要求对其应用程序进行独立验证的开发者，并将在其数据安全部分展示。

我们感谢 Google、ADA 及其所有成员的支持，感谢他们在保护移动应用生态系统方面的出色工作。

2.8.2. 认证机构

2.8.2.1. CREST

CREST 是一个国际非营利性的会员机构，为其会员提供质量保证，并为网络安全行业提供专业认证。CREST 与世界各地的政府、监管机构、学术界、培训伙伴、专业机构和其他利益相关者合作。

2022 年 8 月，CREST 推出了 OWASP 验证标准（OVS）计划。CREST OVS 为应用安全设定了新的标准。在 OWASP 的应用安全验证标准（ASVS）和移动应用安全验证标准（MASVS）的支持下，CREST 正在利用开源社区建立和维护全球标准，以实现全面的 Web 和移动应用安全框架。这将为购买群体提供保证，即开发人员使用 CREST OVS 认证的供应商，始终知道他们是在与有道德和有能力的组织合作，使用 OWASP ASVS 和 MASVS 标准，并拥有熟练和合格的安全测试人员。

- CREST OVS 计划
- CREST OVS 认证过程
- CREST OVS 介绍视频

我们感谢 CREST 对 OVS 计划的咨询，以及对开源社区建立和维护全球网络安全标准的支持。

2.8.3. 标准化机构

2.8.3.1. NIST（美国国家标准与技术研究所）

国家标准与技术研究所（NIST）成立于 1901 年，现在是美国商务部的一部分。NIST 是美国最古老的物理科学实验室之一。国会建立该机构是为了消除当时对美国工业竞争力的一个主要挑战--一个落后于英国、德国和其他经济对手的能力的二流度量基础设施。

- NIST.SP.800-163 “审查移动应用程序的安全性” Revision 1, 2019
- NIST.SP.800-218 “安全软件开发框架(SSDF) v1.1: 缓解软件漏洞风险的建议” v1.1, 2022

2.8.3.2. BSI（德国联邦信息安全办公室）

BSI 是 "联邦信息安全办公室 "的缩写，它的目标是促进德国的 IT 安全，是联邦政府的中央 IT 安全服务提供者。

- 技术指南 BSI TR-03161 电子医疗应用的安全要求 v1.0, 2020 年
- "被保险人 ePA 前端 "和 "被保险人电子处方前端 "的产品评估师测试规范。 被保险人的处方前端 v2.0, 2021 年

2.8.3.3. ioXt

ioXt 联盟的使命是通过多方利益相关者、国际的、协调的和标准化的安全和隐私要求、产品合规计划以及这些要求和计划的公共透明度，建立对物联网产品的信心。

2021 年，ioXt 通过移动应用程序扩展了其安全原则，这样应用开发者就可以确保他们的产品是按照高网络安全标准（如 OWASP MASVS 和 VPN 信任计划）开发的，并保持这些标准。ioXt 移动应用规范是一个安全标准，适用于任何与云连接的移动应用，为消费者和商业移动应用的安全提供了急需的市场透明度。

- ioXt 基础配置 v2.0

2.8.4. 政府机构

名称	文档	年份
欧洲支付委员会	支付威胁和欺诈趋势报告	2021
欧洲支付委员会	移动发起的 SEPA 信贷转移互操作性实施指南，包括 SCT Instant（MSCT IIGs）	2019
ENISA（欧盟网络安全局）	关于 SMART CARS 安全的优秀实践	2019
印度政府，电子和信息技术部	采用 OWASP 的移动应用安全验证标准（MASVS）1.0 版	2019
芬兰运输和通信机构（TRAFICOM）	电子识别服务评估指南（草案）	2019

西班牙国家网络安全研究所 INCIBE	智能玩具中的网络安全问题	2019

2.8.5. 教育机构

名称	文档	年份
美国佛罗里达大学，佛罗里达网络安全研究所，美国	"SO{U}RCERER : 开发者驱动的 Android 应用安全测试框架"	2021
澳大利亚阿德莱德大学和英国伦敦玛丽女王大学	对全球 COVID-19 联系人追踪应用的实证评估	2021
马普亚大学信息技术学院，菲律宾	关于家长控制移动应用安全的漏洞评估：基于 OWASP 安全要求的状况	2021

2.8.6. 科学研究中的应用

STAMBA: Android 移动银行应用程序的安全测试

2.8.7. 书籍

掌握 DevOps 中的安全问题

2.9. 鸣谢

2.9.1. MAS 倡导者

MAS 倡导者是 OWASP MASVS 和 MASTG 的行业采用者，他们投入了大量持续的资源，通过提供持续的高影响力的贡献和不断传播信息来推动项目的发展。

成为 "MAS 倡导者 "是企业在项目中可以获得的最高地位，承认他们已经超越了对项目的支持。

我们将根据这些类别来验证这种状态：

1. 展示采用：查看企业的官方网页就可以清楚发现，他们已经采用了 OWASP MASVS 和 MASTG。比如说：

    - 服务/产品
    - 资源（如博客文章、新闻稿、公开的五项测试报告）。
    - 培训
    - 其他类似

2. 提供持续的高影响力的贡献：通过持续的时间/专用资源支持，为 OWASP MAS 项目提供明确/高影响力的支持。

    - 提交内容请求（例如，增加/升级现有的测试、工具、维护代码样本等）
    - 技术 PR 审查
    - 提高自动化程度（GitHub Actions）
    - 升级、扩展或创建新的 Crackmes
    - 主持 GitHub 讨论
    - 为项目和特殊事件提供高价值的反馈，如 MASVS/MASTG 的重构。
    - 其他类似

3. 通过每年的多次演讲、公共培训、社交媒体的高度参与（例如，喜欢、转发、自己专门发帖宣传该项目）来传播和推广该项目。

注意：你不需要满足每一个要点（它们只是举例）。但是，你必须能够清楚地表明你的贡献的连续性和对项目的高影响力。例如，为了满足 "2."，你可以证明你在最初的 6 个月里一直在发送高影响力的提交请求，并打算继续这样做。

2.9.1.1. 优势

- 在我们的主要 README 和 OWASP 项目主网站上显示公司的标志。
- MASTG 中的链接博客文章将包括公司名称。
- 在每个包含贡献的 PR 的 MASTG 发布上进行特别鸣谢。
- 从 OWASP MAS 账户中转发新的出版物（例如转发）。
- 最初的公开 "感谢"，并在成功续约后每年一次。

2.9.1.2. 如何申请

如果你想申请，请联系项目负责人，向 Sven Schleier 和 Carlos Holguera 发送电子邮件，他们将验证你的申请。请务必包括明显的证明（通常以贡献报告的形式，包括链接到相应元素的 URL），显示你在 6 个月内做了什么，与上述三个类别一致。

2.9.1.3. 重要的免责声明

- 如果你获得了 "MAS 倡导者 "的身份，并且你想保持这个身份，那么上述的贡献在最初的阶段之后也必须保持一致。你应该继续收集这些证明，并每年向我们发送一份贡献报告。
- 财务捐赠不属于资格标准的一部分，但会被列入贡献完成情况。
- 在 MASTG 文本中链接的重新分享的出版物和博客文章必须是**教育性**的，并侧重于移动安全或 MASVS/MASTG，而**不是为公司产品/服务背书。**
- 倡导者公司可以使用标识和 MASVS/MASTG 资源的链接作为其交流的一部分，但不能将其作为 OWASP 的认可为软件和服务的首选供应商。
  - 合适的示例：在网站主页上列出 MAS 倡导者的身份，在销售演示中关于公司的幻灯片中，在销售资料上。
  - 不合适的例子："MAS 倡导者"不能声称他们是 OWASP 认证的。
- 这些公司对 MASVS/MASTG 的应用质量并没有经过 MAS 团队的审核。

  OWASP 基金会非常感谢所列的个人和组织的支持。然而，请注意，OWASP 基金会是严格的厂商中立者，并不为其任何支持者背书。MAS 倡导者不会以任何方式影响 MASVS 或 MASTG 的内容。

2.9.2. 我们的 MAS 倡导者

NowSecure 为项目提供了持续的高影响力的贡献，并成功地帮助传播了信息。

我们要感谢 NowSecure 的模范贡献，它为其他想推动项目的潜在贡献者树立了一个榜样。

2.9.2.1. 作为 MASVS/MASTG 倡导者的 NowSecure

- 服务/产品：
  - NowSecure 首次推出新的 OWASP MASVS 移动应用渗透测试
  - NowSecure 的自动化移动安全测试平台
- 资源：
  - OWASP 移动安全项目的基本指南
- 培训课程：
  - 标准和风险评估
  - OWASP MASVS 和 MASTG 更新
  - 移动应用安全介绍

2.9.2.2. NowSecure 对 MAS 项目的贡献

2.9.2.2.1. 高影响力的贡献（时间/专用资源）

- 内容 PR
- 对 PR 的技术审查
- 参与 GitHub 讨论

特别要提到的是**对 MASVS 重构的贡献**：

- 投入大量的时间来推动讨论，并与社区一起制定提案。
- 可测试性分析
- 对每个类别建议的反馈
- 内部分析的统计数据

在过去，NowSecure 也为该项目做出了贡献，赞助该项目成为 "上帝模式赞助商"，并捐赠了 Android UnCrackable 应用程序级别 4：Radare2 Pay。

2.9.2.2.2. 宣传

- **参与社交媒体**：持续的 Twitter 和 LinkedIn 活动（见示例）
- **博客文章**：
  - 将安全纳入移动应用软件开发的生命周期
  - OWASP 移动安全测试清单有助于合规
- **演讲**：

- "移动浪潮"! 我们的 2.0 版本之旅! (OWASP AppSec EU, June 10 2022)
- 最新的 OWASP MASVS 移动应用安全内部指南（OWASP 多伦多分会，2022 年 2 月 10 日）
- 使用最新的 OWASP MASVS 移动应用安全内部指南（OWASP Virtual AppSec 2021，2021 年 11 月 11 日）
- 使用 OWASP MASVS 的移动应用安全内部指南（OWASP 北弗吉尼亚分会，2021 年 10 月 8 日）
- 以及更多

2.9.3. 贡献者

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。这些统计信息的详情，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。

2.9.3.1. 顶级贡献者

顶级贡献者持续贡献高质量的内容，并且在 GitHub 仓库中至少有 500 次的添加记录。。

- Pawel Rzepa
- Francesco Stillavato
- Henry Hoggard
- Andreas Happe
- Kyle Benac
- Paulino Calderon
- Alexander Anthuk
- Caleb Kinney
- Abderrahmane Aftahi
- Koki Takeyama
- Wen Bin Kong
- Abdessamad Temmar
- Cláudio André
- Slawomir Kosowski
- Bolot Kerimbaev
- Lukasz Wierzbicki

2.9.3.2. 贡献者

贡献者贡献了高质量的内容，并且在 GitHub 资源库中至少有 50 次的添加记录。他们的 Github 名称列在下面：

kryptoknight13, DarioI, luander, oguzhantopgul, Osipion, mpishu, pmilosev, isher-ux, thec00n, ssecteam, jay0301, magicansk, jinkunong, nick-epson, caitlinandrews, dharshin, raulsiles, righet- tod, karolpiateknet, mkaraoz, Sjord, bugwrangler, jasondoyle, joscandreu, yog3shsharma, ryantzj, rylyade1, shivsahni, diamonddocumentation, 51j0, AnnaSzk, hlhodges, legik, abjurato, serek8, mhelwig, locpv-ibl 以及 ThunderSon。

2.9.3.3. 迷你贡献者

许多其他贡献者提交了少量的内容，如一个词或一句话（少于 50 个补充）。他们的 Github 名称列在下面：

jonasw234, zehuanli, jadeboer, Isopach, prabhant, jhscheer, meetinthemiddle-be, bet4it, asla- manver, juan-dambra, OWASP-Seoul, hduarte, TommyJ1994, forced-request, D00gs, vasconcedu, mehradn7, whoot, LucasParsy, DotDotSlashRepo, enovella, ionis111, vishalsodani, chame1eon,allRiceOnMe, crazykid95, Ralireza, Chan9390, tamariz-boop, abhaynayar, camgaertner, Ehsan- Mashhadi, fujiokayu, decidedlygray, Ali-Yazdani, Fi5t, MatthiasGabriel, colman-mbuya 以及 anyashka。

2.9.3.4. 审查者

审查者一直通过 GitHub 问题和提交请求评论提供有用的反馈

- Jeroen Beckers
- Sjoerd Langkemper
- Anant Shrivastava

2.9.3.5. 编辑者

- Heaven Hodges
- Caitlin Andrews

- Nick Epson
- Anita Diamond
- Anna Szkudlarek

2.9.3.6. 捐赠者

虽然 MASVS 和 MASTG 都是由社区自愿创建和维护的，但有时也需要一点外部帮助。因此，我们感谢我们的捐赠者提供资金，使我们能够雇用技术编辑。请注意，他们的捐赠不会以任何方式影响 MASVS 或 MASTG 的内容。捐赠包在我们的 OWASP 项目页面上有描述。

2.10. 中文版说明

（1）本文为《OWASP Mobile Application Security Testing Guide（MASTG)》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）由于译者水平有限，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：

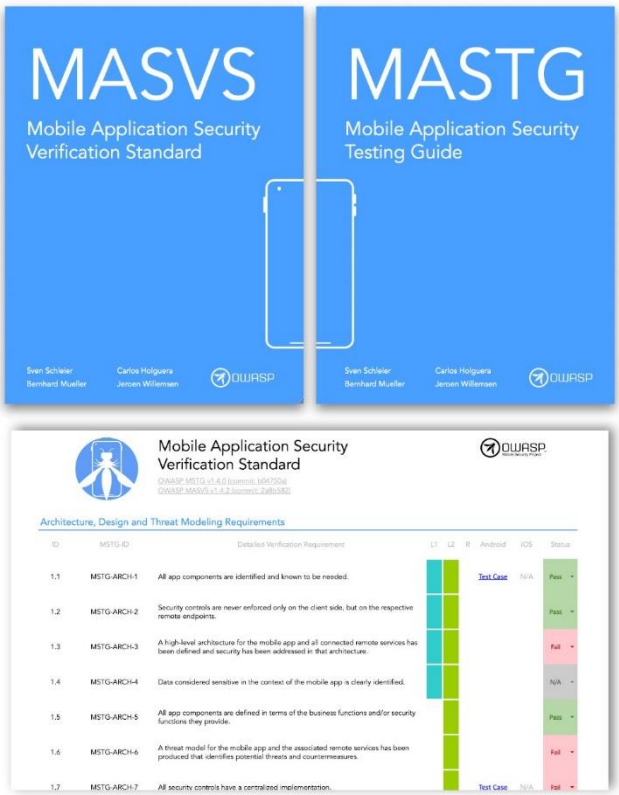
邮箱：yolylight@hotmail.com

3. 概述

3.1. 《OWASP 移动应用安全测试指南》介绍

新技术总会引入新的安全风险，移动计算也不例外。移动应用的安全问题在一些重要的方面与传统桌面软件不同。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，但当我们在移动应用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、应用间通信、合理使用加解密 API 和安全网络通信只是其中的一些考虑因素。

OWASP 移动应用安全验证标准（MASVS）定义了一个移动应用安全模型，并列出了移动应用的通用安全要求。架构师、开发人员、测试人员、安全专家和消费者可以使用它来定义和理解安全移动应用的质量。OWASP 移动应用安全测试指南（MASTG）与 MASVS 提供的安全要求基本一致，根据具体情况，它们可以单独使用或结合使用，以实现不同的目标。



例如，MASVS 要求可用于应用程序的规划和架构设计阶段，而检查表和测试指南可作为人工安全测试的基线，或作为开发期间或之后自动安全测试的模板。在 "移动应用安全测试 "一章中，我们将介绍如何将检查表和 MASTG 应用于移动应用的渗透测试。

3.1.1. 移动应用安全的关键领域

很多移动应用渗透测试人员都有网络和 Web 应用渗透测试背景，这于移动应用测试来说是很有价值的。几乎所有的移动应用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设备上的 Web 应用同样类型的攻击。移动应用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应用安全性的关键领域。

3.1.2. 数据存储和隐私(MASVS-STORAGE)

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。如果一个应用不当地使用了操作系统 API，例如：本地存储或者进程间通讯，那么这个应用可能会给其他运行在同一个设备的应用暴露了敏感数据。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。

当开发移动应用的时候，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。

碎片化是一个我们需要处理的问题，特别是在 Android 设备上。不是每个 Android 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 Android 版本。一个应用要适配这些过时的设备，它可能被迫使用过时版本的 Android 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 来创建应用，即使会排除掉某些用户。

3.1.3. 加密(MASVS-CRYPTO)

当涉及到保护存储在移动设备上的数据时，密码学是一个基本要素。它也是一个可能出现可怕错误的领域，特别是当没有遵循标准惯例时。必须确保应用程序根据行业最佳实践使用加密技术，包括使用经过验证的加密库，正确选择和配置加密基元，以及在需要随机性的地方使用合适的随机数发生器。

3.1.4. 身份认证和授权(MASVS-AUTH)

在多数情况下，发送用户信息到远程服务，在移动应用架构里是不可或缺的一部分。即使很多身份认证和授权逻辑都实现在远程节点上，但在移动应用方面也有一些实施上的挑战。不同于 Web 应用，移动应用常常会存储长时间的会话令牌（session tokens），通过用户到设备的认证功能（如指纹扫描）来解锁。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。

移动应用架构也越来越多地结合授权框架（如：OAuth2），将认证委托给一个单独的服务，或将认证过程外包给认证提供商。使用 OAuth2 可以将客户端的认证逻辑外包给同一设备上的其他应用（如系统浏览器）。安全测试人员必须了解不同的可能的授权框架和架构的优点和缺点。

3.1.5. 网络通信(MASVS-NETWORK)

移动设备经常连接到各种网络，包括与其他（潜在的恶意）客户端共享的公共 Wi-Fi 网络。这为各种基于网络的攻击创造了机会，从简单到复杂，从旧到新。保持移动应用程序和远程服务端点之间交换的信息的保密性和完整性至关重要。作为一项基本要求，移动应用程序必须使用具有适当设置的 TLS 协议为网络通信建立一个安全、加密的通道。

3.1.6. 与移动平台进行交互(MASVS-PLATFORM)

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 的访问来实现应用权限系统的。他们还提供或多（Android）或少（iOS）的进程间通讯（IPC）的便利，来使应用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应用。

3.1.7. 代码质量和漏洞缓解(MASVS-CODE)

由于攻击面较小，传统的注入和内存管理问题并不经常出现在移动应用中。移动应用程序大多与受信任的后台服务和用户界面互动，因此即使应用程序中存在许多缓冲区溢出漏洞，这些漏洞通常不会形成任何有用的攻击向量。这同样适用于浏览器漏洞，如跨站脚本（XSS 允许攻击者将脚本注入到网页中），这些漏洞在 Web 应用中非常盛行。然而，总是有例外情况。在某些情况下，移动端的 XSS 在理论上是可能的，但很少看到攻击者可以利用的 XSS 问题。

传统注入缺陷和内存管理缺陷的免疫，并不意味着应用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 提供的释放安全特性帮助增加安全性和减缓攻击。

3.1.8. 反篡改和反逆向(MASVS-RESILIENCE)

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应用世界，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 MASTG 导览

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

1、 “通用测试指南”章节包含了一套移动应用安全测试的方法论和通用漏洞分析技术。因为它们适用于移动应用安全。它还包含与操作系统无关的附加技术测试案例，如认证和会话管理、网络通信和密码学。

2、 “ Android 测试指南”章节覆盖了 Android 平台的移动安全性测试，包括：基础安全性、安全性测试用例、逆向工程技术和预防，以及篡改技术和预防。

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、逆向工程技术和预防，以及篡改技术和预防。

4. 通用移动测试指南

4.1. 移动应用分类

术语“移动应用”或“移动 APP”是指设计用于在移动设备上执行的独立的计算机程序。如今，Android 和 iOS 操作系统累计占到了移动操作系统市场份额的 99%以上。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“应用”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，应用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，我们将定义应用在移动应用分类中各自位置的特性，并讨论每个变体的差异。

4.1.1. 原生应用

移动操作系统，包括 Android 和 iOS，都有一个软件开发工具包（SDK），用于开发特定于该操作系统的应用程序。这些应用程序被称为是为其开发的系统的原生应用程序。当讨论一个应用程序时，一般的假设是，它是一个用各自操作系统的标准编程语言实现的原生应用程序——iOS 是 Objective-C 或 Swift，而 Android 是 Java 或 Kotlin。

原生应用程序本身就有能力提供最快的性能和最高程度的可靠性。它们通常遵守平台特定的设计原则（如 Android 设计原则），与混合或 Web 应用相比，这往往会产生一个更一致的用户界面（UI）。由于与操作系统的紧密结合，原生应用程序可以直接访问设备的几乎所有组件（摄像头、传感器、硬件支持的 keystore 等）。

在讨论 Android 的原生应用程序时，存在一些模糊不清的地方，因为该平台提供了两个开发包—Android SDK 和 Android NDK。SDK 是基于 Java 和 Kotlin 编程语言的，是开发应用程序的默认工具。NDK（或原生开发工具包）是一个 C/C++开发工具包，用于开发可以直接访问低级 API（如 OpenGL）的二进制库。这些库可以包含在用 SDK 构建的普通应用程序中。因此，我们说，Android 原生应用程序（即用 SDK 构建的）可能有用 NDK 构建的原生代码。

原生应用程序最明显的缺点是它们只针对一个特定的平台。要为 Android 和 iOS 构建相同的应用程序，需要维护两个独立的代码库，或者引入复杂的开发工具，将单个代码库移植到两个平台。以下框架是后者的一个例子，允许你使用一套代码在 Android 和 iOS 平台进行编译。

- Xamarin
- Google Flutter
- React Native

使用这些框架开发的应用程序在内部使用系统原生的 API，其性能与原生应用程序相当。此外，这些应用程序可以利用所有的设备功能，包括 GPS、加速器、摄像头、通知系统等。由于最终输出结果与之前讨论的原生应用程序非常相似，因此使用这些框架开发的应用程序也可以被视为原生应用程序。

4.1.2. Web 应用

移动 Web 应用（或简单地说，Web 应用）是设计成看起来和感觉像原生应用的网站。这些应用程序在设备的浏览器上运行，通常用 HTML5 开发，很像现代网页。可能创建启动图标，以形成访问原生应用程序的相同感觉；然而，这些图标本质上与浏览器书签相同，只是打开默认的网络浏览器以加载引用的网页。

Web 应用与设备的一般组件的整合有限，因为它们在浏览器的限制内运行（即它们是 "沙盒"），并且与原生应用相比，通常缺乏性能。由于 Web 应用通常针对多个平台，它们的用户界面并不遵循特定平台的一些设计原则。最大的优势是减少了与单一代码库相关的开发和维护成本，并使开发人员能够随时发布更新，而不需要接触特定平台的应用商店。例如，对 Web 应用的 HTML 文件的修改可以直接作为可行的、跨平台的更新，而对基于商店的应用的更新则需要更多的努力。

4.1.3. 混合应用

混合应用试图填补原生和 Web 应用之间的空白。混合应用的执行方式与原生应用类似，但大部分过程依赖于 Web 技术，这意味着部分应用在嵌入式网络浏览器中运行（通常称为 "WebView"）。因此，混合应用继承了原生和 Web 应用的优点和缺点。

Web 到原生的抽象层使混合应用程序能够访问纯 Web 应用程序无法访问的设备功能。根据用于开发的框架，一个代码库可以产生多个针对不同平台的应用程序，其用户界面与开发该应用程序的原始平台非常相似。

以下是一个开发混合应用较流行的框架的不完全列表：

- Apache Cordova
- Framework 7
- Ionic
- jQuery Mobile
- Native Script
- Onsen UI
- Sencha Touch

4.1.4. 增强 Web 应用

增强 Web 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 Web 应用。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应用有效的。

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 Web 应用 Manifest 是一个简单的 JSON 文件，可以在“安装”后用来配置应用的行为。

PWA 可支持 Android 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

在本指南中，我们将重点介绍在智能手机上运行的 Android 和 iOS 的应用程序。这些平台目前在市场上占主导地位，也在其他设备类别上运行，包括平板电脑、智能手表、智能电视、汽车信息娱乐装置和其他嵌入式系统。即使这些额外的设备类别不在范围内，你仍然可以应用本指南中描述的大部分知识和测试技术，但根据目标设备的不同会有一些偏差。

鉴于有大量的移动应用程序框架可用，不可能详尽地涵盖所有这些框架。因此，我们专注于每个操作系统上的原生应用程序。然而，同样的技术在处理 Web 或混合应用程序时也是有用的（最终，无论哪个框架，每个应用程序都是基于原生组件的）。

4.2. 移动应用安全性测试

在以下章节中，我们将简要介绍通用的安全测试原则和关键术语。所介绍的概念与其他类型的渗透测试中的概念基本相同，所以如果你是一个有经验的测试人员，你可能会熟悉其中的一些内容。

纵观这个指南，我们用“移动应用安全性测试”作为笼统的术语来指通过静态和动态分析来进行移动应用安全性评估。诸如 "移动应用渗透测试 "和 "移动应用安全审查 "等术语在安全行业的使用有些不一致，但这些术语指的是大致相同的事情。移动应用安全测试通常是更大的安全评估或渗透测试的一部分，包括客户-服务器架构和移动应用使用的服务器端 API。

在这份指南中，我们涵盖两种情况下的移动应用安全性测试。第一个是在接近软件开发生命周期结束时完成的“经典”安全性测试。在这种情况下，测试人员几乎完成的或生产就绪的应用程序版本，识别安全缺陷，然后编写（通常是极为惨烈的）报告。另一种情况则是从软件开发生命周期的开始，就实施需求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应用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应用的充分认知。这方面认知可能包含源代码、文档和示意图。由于其透明度，这种方法允许比黑盒测试更快的测试，并且通过获得的额外知识，测试人员可以建立更加复杂和细化的测试案例。

- 灰盒测试是指介于上述两种测试类型之间的所有测试：提供给测试人员一些信息（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。

我们强烈建议你要求提供源代码，这样你就可以尽可能合理地利用测试时间。测试人员的代码访问显然不能模拟外部攻击，但它简化了漏洞的识别，允许测试人员在代码层面上验证每一个被识别的异常或可疑的行为。如果应用程序之前没有被测试过，那么白盒测试就是一种方式。

即使 Android 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应用里寻找威胁的过程。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

静态应用安全测试（SAST）是指通过手动或自动分析源代码来检查应用程序的组件，而不对其进行删除。OWASP 提供了关于静态代码分析的信息，可以帮助你了解它的技术、优势、弱点和限制。

动态应用安全性测试（DAST）涉及在运行期间检查应用程序。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，移动应用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，使用的是自动/手动混合方法。自动扫描能捕获容易找到的目标，而人工测试者可以在在考虑到具体使用情况的前提下探索代码库。

4.2.1.4.1. 人工代码审查

一个测试者通过手动分析移动应用源代码的安全漏洞来做人工代码审查。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。

人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人工分析的一个好的开始。

与自动代码分析相比，人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。

人工代码审查需要专家级的代码审查人员，他必须精通移动应用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，特别是考虑到有许多依赖关系的大型代码库。

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应用上运行，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。

尽管一些静态代码分析工具包含了很多分析移动应用程序所需的规则和语义信息，但它们可能会产生很多误报，特别是在没有针对目标环境进行配置的情况下。因此，安全专家必须始终审查结果。

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 的重点是通过应用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，可以分析到移动应用的请求和返回模式。

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、身份认证和授权问题，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

自动化测试工具对应用内容缺乏敏感性是一个挑战。这些工具可能会发现一个不相关的潜在问题。这种结果被称为“误报”。

例如，安全测试人员通常会报告可在 Web 浏览器中利用但与移动应用无关的漏洞。这种误报的发生是因为用于扫描后端服务的自动化工具是基于常规的基于浏览器的 web 应用。诸如 CSRF（跨站请求伪造）和跨站脚本（XSS）等问题会被相应地报告。。

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。

移动应用不会满足这些要求：即使使用了 WebView 和基于 cookie 的会话管理，用户单击的任何恶意链接都会在默认浏览器中打开，该浏览器使用一个单独的 cookie 存储。

如果应用包含了 WebView，则存储型跨站点脚本（XSS）可能会成为一个问题；如果应用暴露了 Javascript 接口，那么它甚至可能引起命令执行。然而，由于上述原因，反射型跨站点脚本很少是一个问题（即使它们是否应该存在还有争议，转义输出只是一种最佳实践）。

> 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；不要盲目相信您扫描工具的输出。

4.2.1.7. 渗透测试

经典的方法论包含了对应用最终或者接近最终构建版本的全面安全测试，例如：在开发周期尾声的可用版本。对于在开发周期尾声的测试，我们推荐移动应用安全性确认标准（MASVS）和相关检查表作为测试基线。典型的安全测试结构如下：

- **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的所有同步，以及对测试人员（通常是第三方）的法律保护。请记住，在世界许多地方，未经书面授权攻击系统是非法的！
- **情报收集** - 分析应用的环境和架构背景以获得对应用来龙去脉的了解。
- **映射应用程序** - 基于前几个阶段的信息；也可能通过自动化测试和人工探索应用进行完善。映射提供了对应用程序、其入口点、其持有的数据以及主要潜在漏洞的透彻理解。然后，可以根据这些漏洞的攻击可能造成的损害对其进行排序，以便安全测试人员可以对其进行优先级排序。该阶段包括创建测试执行期间可能使用的测试用例。
- **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应用。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，安全测试人员报告漏洞。这包括详细的攻击过程，对漏洞类型进行分类，记录攻击者可能危害目标的风险，并概述测试人员能够非法访问的数据。

4.2.1.7.1. 准备阶段

测试应用程序的安全级别必须在测试之前确定。安全要求应在项目开始时确定。不同的组织有不同的安全需求和可用于投入测试活动的资源。尽管 MASV 1 级（L1）中的控制点适用于所有移动应用程序，但与技术和业务利益相关者一起浏览整个 L1 和 2 级（L2）MASVS 控制点检查表是确定测试覆盖率级别的好方法。

在某些地区，组织可能有不同的监管和法律义务。即使应用程序不处理敏感数据，一些 L2 要求也可能是相关的（因为行业法规或当地法律）。例如，双因素认证（2FA）可能对金融应用程序是强制性的，并由一个国家的中央银行和/或金融监管机构强制执行。

在与利益相关者的讨论中，也可以审查开发过程中早期定义的安全目标/控制。一些控制措施可能符合 MASVS 的控制措施，但其他控制措施可能是针对组织或应用程序的。

所有相关方必须就检查表中的决定和范围达成一致，因为这些决定和范围将定义所有安全测试的基线。

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进行动态分析。公司政策可能禁止在企业网络内使用 root 过的手机或（硬件和软件）网络测试工具。实现 root 检测和其他逆向工程防御的应用程序可能会显著增加进一步分析所需的工作量。

安全性测试涉及许多侵入性的任务，包括监视和操控移动应用的网络流量、检查应用的数据文件，以及检测 API 调用。安全管控，例如：证书固定和 root 检测，可能会妨碍这些任务以及极大地降低测试速度。

为了克服这些阻碍，您可能要跟开发团队请求两个应用的变体版本。其中一个变体应该是发布版本，这样您就可以确定已完成的控制措施是否正常工作，并且不能轻易绕过。另一个变体应该是一个调试版本，其中某些安全控制措施已经被停用了。测试两种不同的版本是涵盖所有测试用例的最有效办法。

根据参与的范围，这种方法可能不可行。请求白盒测试的生产和调试版本将帮助您完成所有测试用例，并清楚地说明应用程序的安全成熟度。客户可能更倾向于进行黑盒测试，以集中于生产应用及其安全控制有效性的评估。

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

敏感信息的分类因行业和国家而异。此外，组织可能会对敏感数据采取受限查看，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 - 数据位于文件或数据存储中。
- 使用中 - 应用程序已将数据加载到其地址空间中。
- 传输中 - 移动应用程序和终端或设备上的消费进程之间交换了数据，例如在 IPC（进程间通信）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 Web 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 Web 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、PIN 等）。
- 可被滥用用于身份盗窃的个人识别信息（PII）：社会安全号码、信用卡号码、银行帐号、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。
- 法律义务要求保护的任何数据。
- 由应用程序（或其相关系统）生成并用于保护其他数据或系统本身的任何技术数据（例如，加密密钥）。

必须提前决定好“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

情报收集涉及关于应用架构的信息收集、应用所服务的业务用例以及应用运行环境的信息。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：

- 机构为此应用设定的目标。功能会影响用户与应用程序的交互，并可能使某些界面比其他界面更容易成为攻击者的目标。。
- 相关行业。不同的行业可能有不同的风险状况。
- 利益团体和投资者；了解谁对这个应用感兴趣和谁对这个应用负责。
- 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漏洞创造机会。

4.2.1.7.1.5 架构信息

架构信息包括：

- **移动应用**：应用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 root 过的手机并且对这种状况进行反应。
- **操作系统**：应用运行的操作系统和系统版本（包括：Android 和 iOS 版本限制），应用是否期望运行在那些有移动设备管理控制（MDM）的设备上，以及相关的系统漏洞。
- **网络**：使用安全传输协议（如 TLS）、使用强密钥和加密算法（如 SHA-2）来保护网络流量加密、使用证书固定来验证对端等。
- **远程服务**：应用使用的远程服务，以及这些服务被破坏是否会危害客户端。。

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应用和它范畴的信息，下一步就是映射应用的结构和内容，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例中展开，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 工具的使用就可以揭示一些关于漏洞的有价值信息（例如：SQL 注入）。DAST 工具可以支持黑盒测试和自动扫描这个应用：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，并且只能找到那些它们被编程好要找到的东西。因此，人工分析或许是必要的，以强化自动化工具的结果（直觉通常是安全测试的关键）。

威胁建模是一个重要的手段：来自研讨会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应用来说是非常适用的。

4.2.1.7.3. 漏洞利用

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映射的渗透测试人员（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。
- **可发现性** - 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：

- 摘要。
- 范围和背景描述（例如：目标系统）。
- 使用的方法。
- 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。

互联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，虽然敏捷实践的广泛采用正在加速软件开发，但安全测试人员必须变得更快、更敏捷，同时继续交付值得信赖的软件。

接下来的章节将聚焦这种演进，并描述当代安全测试。

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），它们也有负面特性（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 概览

4.2.2.2.1. SDLC 概述

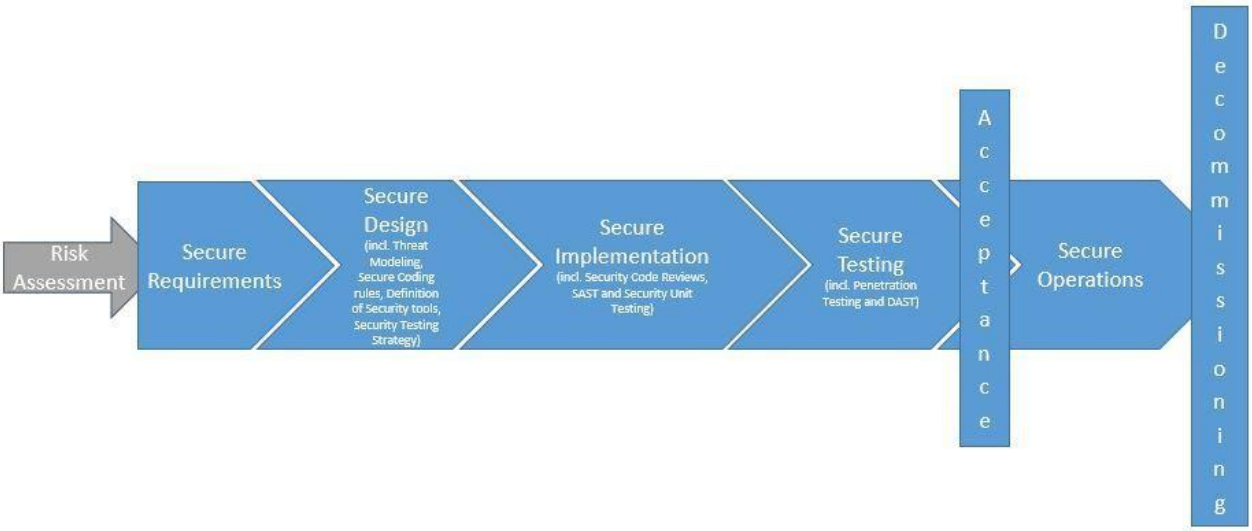
SDLC 总是由相同的步骤组成。（在瀑布范式中，整个过程是按顺序的，而在敏捷范式中是迭代的。）

- 为应用程序和它的组件执行一个**风险评估**，来识别它们的风险状况。这些风险状况通常取决于组织的风险偏好和适用的监管要求。风险评估还基于各种因素，包括应用程序是否可以通过互联网访问，以及应用程序处理和存储的数据类型。必须考虑各种风险：金融、营销、工业等。数据分类政策规定了哪些数据是敏感的，以及必须如何保护这些数据。
- **安全需求**在项目或开发周期开始时确定，此时正在收集功能需求。创建用例时会添加**滥用案例**。如果需要，团队（包括开发团队）可以接受安全培训（例如安全编码）。您可以使用

OWASP MASV 在风险评估阶段的基础上确定移动应用程序的安全要求。在添加特性和数据类时，迭代地审查需求是常见的，尤其是在敏捷项目中。

- **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。建立**安全编码规则**，并创建将使用的**安全工具列表**。阐明了安全测试的策略。
- 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，这样开发和运营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。创建受版本控制且仅包含这些代码片段的专用存储库是一种安全的编码策略，比传统方法（将准则存储在 word 文档或 PDF 中）更为有益。。
- **安全开发软件。**为了提高代码安全性，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性，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**（测试人员)。**动态应用程序安全性测试**也常常在这个阶段执行。
- 所有的利益相关方在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工作：



基于项目的一般风险状况，您可以简单化（或者甚至跳过）一些工作，并且您可以增加一些其他的工作（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：**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 的概述；始终根据您的项目定制此框架。

4.2.2.2.2. 定义测试策略

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。然而，策略必须被共享出来，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。此外，所有技术团队都必须同意它，以确保它不会给任何一个团队带来不可接受的负担。

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。度量标准的示例包括以下几点：

- 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：可以这么说，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。当开发团队用敏捷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。

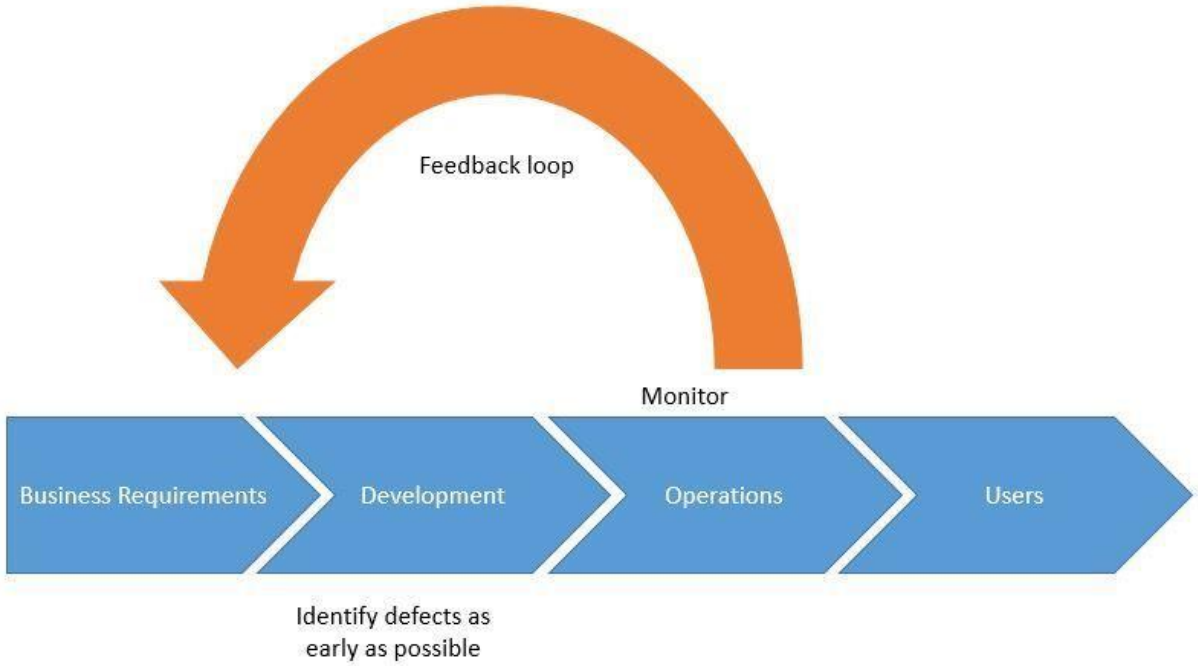
DevOps 是应对这一挑战而做的必要的解决方案演变，因为它使得软件可以更快地交付给客户。这是很大程度上是通过广泛的构建自动化、软件测试和发布的流程以及架构改进（除了 DevOps 的协作方面）来完成的。这种自动化包含在具有持续集成（CI）和持续发布（CD）的开发管线里。

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 DevOpsQATestInfoSec 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. 什么是敏捷开发和 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。
- 开发变得更加脚本化，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。

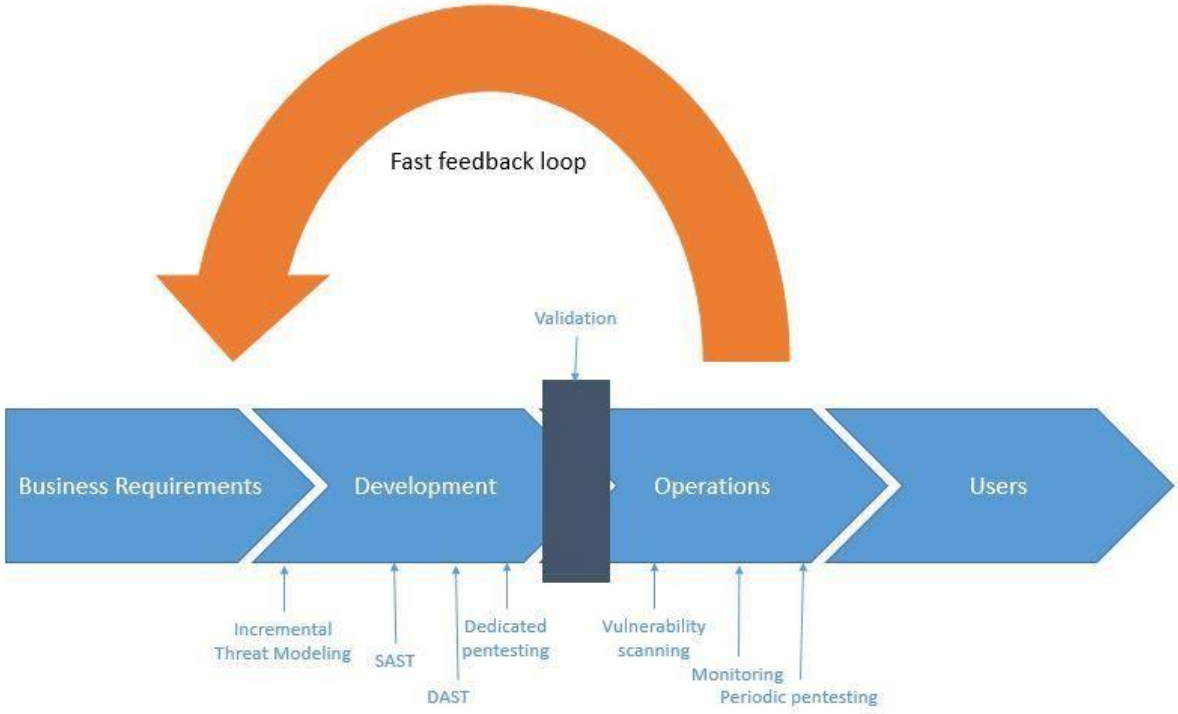
再一次，答案就是自动化和工具化：通过在整个项目生命周期中践行这两个概念，您就可以维持和改进安全性。期望的安全级别越高，管控、检查点以及重点就会越多。以下就是例子：

- 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

4.3. 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：移动应用黑盒测试越来越需要反汇编已编译好的应用、应用补丁和调试二进制代码甚至是实时进程。实际上很多移动应用都实现了对不受欢迎的篡改进行防御，这让测试人员的工作变得不那么轻松。

对一个移动应用的逆向工程是分析编译好应用的一个过程，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应用（要么是编译好的应用要么是正在运行的进程）或者它的环境变量以影响它的行为的过程。例如：一个应用可能拒绝运行在您手上被 Root 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，您就需要修改这个应用的行为了。

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。他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：

1. **使移动应用黑盒测试变成可能。**现代应用经常包含阻碍动态分析的管控程序，SSL 固定和端到端（E2E）加密时会阻止您用代理来拦截或操作流量。root 检测可以阻止应用运行在 root 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。
2. **在黑盒安全测试中优化静态分析。**在一个黑盒测试中，对应用字节码或者二进制码的静态分析帮助您理解应用的内部逻辑。它也让您能识别一些诸如硬编码身份信息之类的问题。
3. **评估逆向工程的适应能力。**执行了移动应用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应用，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适应性评估，作为一般安全测试的一部分。对于适应性评估，测试人员承担逆向工程师的角色，并尝试绕过防御。

在深入移动应用的逆向世界之前，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：移动应用的部署和沙箱方式，在设计上比传统桌面应用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（例如 DRM 系统）并不奏效。Android 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。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 Android 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应用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 Android 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应用的过程，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、编辑和重组一个应用。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，所以运行修改过的应用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——您只需要给应用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、劫持方法和原生方法等很有用。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劫持和代码注入框架。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、劫持，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、语言绑定和一个带有控制台的可注入的 Javascript VM。

然而，您也可以用 Substrate 来注入 Cycript，一种编程环境（又名“Cycript-to-Javascript”编译器），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 分支。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，不用担心）。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 Android 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。

4.3.3. 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应用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，您可以获取与应用编译时的架构相匹配的汇编代码。Android 的 Java 应用可以被汇编成 smali，这是一种 Android 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。

从理论上讲，汇编代码和机器代码之间的映射应该是一对一的，因此可能给人的印象是，反汇编是一项简单的任务。但在实践中，会存在很多陷阱，例如：

- 代码和数据之间的可靠区分。
- 可变指令大小。
- 间接分支指令。
- 在可执行文件的代码段中没有显式调用指令的函数。
- 位置独立代码（PIC）序列。
- 人工编制的汇编代码。

同样，反编译也是一个非常复杂的过程，涉及许多确定性和启发式方法。因此，反编译通常并不真正准确，但对于快速理解所分析的函数非常有帮助。反编译的准确性取决于被反编译代码中可用的信息量和反编译器的复杂程度。此外，许多编译和后编译工具为已编译代码引入了额外的复杂性，以增加理解和/或甚至反编译本身的难度。这种代码称为混淆代码。

在过去的几十年中，许多工具完善了反汇编和反编译过程，可以输出高可信的结果。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，并获得一份经过充分审查的用户指南。在本节中，我们将介绍其中的一些工具，在随后的“逆向工程和篡改”Android 和 iOS 章节中，我们将重点介绍这些技术本身，尤其是针对现有平台的技术。

4.3.3.2. 混淆

混淆是改造代码和数据的过程，使其更难以理解（有时甚至难以反汇编）。它通常是软件保护方案的一个组成部分。混淆不是可以简单地打开或关闭的东西，程序可以通过许多方式和不同程度地使其全部或部分无法理解。

注意：下面介绍的所有技术都不能阻止有足够时间和预算的人对你的应用程序进行逆向工程。然而，结合这些技术将使他们的工作大大增加难度。因此，其目的是阻止逆向工程师进行进一步的分析，而不是使其值得付出努力。

以下技术可用于混淆一个应用程序：

- 名称混淆
- 指令替换
- 控制流扁平化
- 死代码注入

- 字符串加密
- 包装

4.3.3.2.1. 名称混淆

标准编译器根据源代码中的类和函数名称生成二进制符号。因此，如果没有应用混淆技术，符号名称仍然是有意义的，可以很容易地从应用程序的二进制文件中直接读取。例如，一个检测越狱的函数可以通过搜索相关的关键词（如 "jailbreak"）来定位。下面的列表显示了从 Damn Vulnerable iOS 应用(DVIA-v2)中反汇编的函数 `JailbreakDetectionViewController.jailbreakTest4Tapped`。

```
__T07DVIA_v232JailbreakDetectionViewControllerC20jailbreakTest4TappedyypF:
stp         x22, x21, [sp, #-0x30]!
mov         rbp, rsp
```

经过混淆处理后，我们可以观察到符号的名称不再有意义，如下表所示。

```
__T07DVIA_v232zNNtWKQptikYUBNBgfFVMjSkvRdhhnbyyFySbyypF:
stp         x22, x21, [sp, #-0x30]!
mov         rbp, rsp
```

尽管如此，这只适用于函数、类和字段的名称。实际的代码仍然没有被修改，所以攻击者仍然可以阅读函数的反汇编版本，并试图了解其目的（例如检索安全算法的逻辑）。

4.3.3.2.2. 指令替换

这种技术用更复杂的表示方法取代了标准的二进制运算符，如加法或减法。例如，一个加法 x=a+b 可以表示为 x=-（-a）-（-b）。然而，使用相同的替换表示法很容易被逆向，所以建议为一个案例添加多个替换技术，并引入一个随机因素。这种技术很容易被反混淆，但是根据替换的复杂性和深度，应用这种技术仍然会很费时。

4.3.3.2.3. 控制流扁平化

控制流扁平化用更复杂的表示方法取代了原始代码。这种转换将函数的主体分解成基本块，并将它们全部放在一个带有控制程序流的开关语句的无限循环内。这使得程序流明显难以跟踪，因为它删除了通常使代码更容易阅读的自然条件结构。

original

```
i = 1;
s = 0;


while (i <= 100) {




  s += i;
  i++;


}
```

control-flow flattening applied

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1: {
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2: {
      if (i <= 100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3: {
      s += i;
      i++;
      swVar = 2;
      break;
    }
  }
}
```

图片显示了控制流扁平化是如何改变代码的（见 "通过控制流扁平化来混淆 C++程序"）。

4.3.3.2.4. 死代码注入

这种技术通过在程序中注入死代码使程序的控制流更加复杂。死代码是代码的分支，不影响原程序的行为，但增加了逆向工程过程的难度。

4.3.3.2.5. 字符串加密

应用程序通常编译时包含硬编码的密钥、许可证、令牌和端点 URL。默认情况下，所有这些都是以明文形式存储在应用程序二进制文件的数据部分。这种技术对这些值进行加密，并在程序中注入代码分支，在程序使用这些数据之前将其解密。

4.3.3.2.6. 包装

包装是一种动态改写混淆技术，它将原始可执行文件压缩或加密成数据，并在执行过程中动态地恢复它。包装可执行文件改变了文件的签名，以试图避免基于签名的检测。

4.3.3.3. 调试和跟踪

在传统意义上，调试是一个在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、函数劫持和内核跟踪工具。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，在反汇编器中基于人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符号执行）。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 Android 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 Android 编译说明。

4.3.4.2. 基于仿真的动态分析

仿真是对某个计算机平台的模拟或让程序在不同平台或在另一程序中执行。执行这种模拟的软件或硬件称为仿真器。仿真器提供了一种比实际设备便宜得多的替代方案，用户可以在不担心损坏设备的情况下对其进行操作。Android 有多个可用的仿真器，但对于 iOS，实际上没有可行的仿真器可用。iOS 只有一个模拟器，在 Xcode 中提供。

模拟器和仿真器之间的差异通常会导致混淆，并导致这两个术语的交替使用，但实际上它们是不同的，特别是对于 iOS 用例。仿真器模拟目标平台的软件和硬件环境。另一方面，模拟器只模拟软件环境。

基于 QEMU 的 Android 仿真器在运行应用程序时会考虑 RAM、CPU、电池性能等（硬件组件），但在 iOS 模拟器中，根本不会考虑这种硬件组件行为。iOS 模拟器甚至缺乏 iOS 内核的实现，因此，如果应用程序使用系统调用，则它无法在模拟器中执行。

简单地说，仿真器是对目标平台的更接近的模仿，而模拟器只是模仿其中的一部分。

在仿真器上运行一个应用给了您一个强有力的方式来监控和操作它的环境。对于一些逆向工程任务，特别是那些需要低级指令来跟踪的任务，仿真是最好（唯一）的选择。不幸的是，这种类型的分析只对 Android 可行，因为不存在免费或开源的 iOS 的仿真器（iOS 模拟器不是一个仿真器，并且为 iOS 设备编译的应用并不能运行在上面）。唯一可用的 iOS 模拟器是商业 SaaS 解决方案—Corellium。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仿真的分析框架的概述。

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 的专业反汇编程序具有脚本功能和扩展性，但它们根本不适合解决特殊问题。逆向框架允许你执行任何类型的逆向任务并使其自动化，而不需要依赖沉重的 GUI。值得注意的是，大多数反转框架都是开源的和/或免费提供的。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程序分析

在 21 世纪末，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”实际上是指将程序的可能路径表示为一阶逻辑公式的过程。 可满足性模理论(SMT)求解器被用来检查这些公式的可满足性，并提供解决方案，包括在对应于已解决的公式的路径上达到某个执行点所需的变量的具体数值。。

简单地说，符号执行就是在不执行程序的情况下对程序进行数学分析。在分析过程中，每个未知输入都被表示为一个数学变量（符号值），因此对这些变量执行的所有操作都被记录为一个操作树（又称 AST（抽象语法树），来自编译原理）。这些 AST 可以转换为所谓的约束，这些约束将由 SMT 求解器进行解释。在分析的最后，得到了一个最终的数学方程，其中变量是输入的未知值。SMT 求解器是求解这些方程的特殊程序，可以为给定最终状态的输入变量提供可能的值。

为了说明这一点，请想象一个函数，它接受一个输入（x）并将其乘以第二个输入（y）的值。最后，还有一个 if 条件，它检查计算的值是否大于外部变量（z）的值，如果为真，则返回“成功”，否则返回“失败”。此操作的方程式为（x*y）>z。

如果我们希望函数始终返回“成功”（最终状态），我们可以告诉 SMT 求解器计算满足相应方程式的 x 和 y 的值（输入变量）。与全局变量的情况一样，可以从该函数外部更改其值，这可能会导致在执行该函数时产生不同的输出。这增加了确定正确解决方案的复杂性。

在 SMT 求解器内部使用各种方程求解技术生成此类方程的解。其中一些技术非常先进，它们的讨论超出了本书的范围。

在实际情况下，函数比上述示例复杂得多。函数复杂性的增加可能会对经典符号执行提出重大挑战。一些挑战总结如下：

- 程序中的循环和递归可能导致无限执行树。
- 多个条件分支或嵌套条件可能导致路径爆炸。
- 由于 SMT 求解器的局限性，符号执行生成的复杂方程可能无法求解。
- 程序正在使用符号执行无法处理的系统调用、库调用或网络事件。

为了克服这些挑战，通常将符号执行与动态执行（也称为具体执行）等其他技术相结合，以缓解经典符号执行特有的路径爆炸问题。这种具体（实际）和符号执行的组合称为 concolic 执行（concolic 的名称源于具体和符号），有时也称为动态符号执行。

为了将此可视化，在上面的示例中，我们可以通过执行进一步的逆向工程或动态执行程序并将此信息输入符号执行分析来获得外部变量的值。这些额外的信息将降低方程的复杂性，并可能产生更准确的分析结果。结合改进的 SMT 求解器和当前的硬件速度，concolic 执行允许探索中型软件模块中的路径（即大约 10 KLOC）。

此外，符号执行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（混合使用实际执行跟踪、模拟和符号执行）来逆向工程基于 VM 的软件保护。

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

4.3.5. 参考文献

- [#vadla] Ole André Vadla Ravnås, 代码追踪器的剖析 - https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8
- [#salwan] Jonathan Salwan and Romain Thomas, Triton 如何帮助扭转基于虚拟机的软件保护措施 - https://drive.google.com/file/d/1EzuddBA61jEMy8XbjQK FF3jyoKwW7tLq/view?usp=sharing

4.4. 移动应用的身份认证架构

身份认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。

大多数移动应用都实现了某种类型的用户认证。尽管部分认证和状态管理逻辑是由后端服务执行的，但认证是大多数移动应用架构中不可或缺的一部分，因此了解其常见的实现是很重要的。

由于 iOS 和 Android 的基本概念是相同的，我们将在这个通用指南中讨论普遍的认证和授权架构和隐患。操作系统特有的认证方式，如本地和生物识别认证，将在各自的操作系统特有的章节中讨论。

4.4.1. 测试身份认证的通用指南

认证没有一刀切的方法。在审查应用程序的认证架构时，你应该首先考虑所使用的认证方法是否适合给定的环境。认证可以基于以下一个或多个方面：

- 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）。

移动应用程序实施的认证程序的数量取决于功能或访问资源的敏感性。在审查认证功能时，请参考行业最佳实践。用户名/密码认证（结合合理的密码策略）通常被认为是有用户登录且不是很敏感的应用程序的最佳选择。大多数社交媒体应用程序都使用这种形式的认证。

对于敏感的应用程序，添加第二个认证因素通常是合适的。这包括提供访问非常敏感的信息（如信用卡号码）或允许用户转移资金的应用程序。在某些行业，这些应用程序还必须符合某些标准。例如，金融应用程序必须确保符合支付卡行业数据安全标准（PCI DSS）、格雷姆-里奇-比利雷法案和萨班斯-奥克斯利法案（SOX）。美国卫生保健部门的合规考虑包括《健康保险可携性和责任法案》（HIPAA）和《病人安全规则》。

你也可以使用 OWASP 移动应用安全验证标准作为准则。对于非关键应用程序（"1 级"），MASVS 列出了以下认证要求：

- 如果应用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认证形式，例如用用户名/密码的身份认证。
- 存在密码策略并且在远端强制实施。
- 当提交不正确的身份认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。

对于敏感的应用（“Level 2”），MASVS 增加了以下内容：

- 在远端存在第二因素身份认证，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，需要加强身份认证。
- 当用户登录时，该应用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. 有状态认证 VS 无状态认证

您常常会发现移动应用用 HTTP 协议作为传输层。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：

- 使用有状态认证，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。
- 使用无状态认证，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。尽管有时候移动应用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

身份认证方案有时辅以被动上下文认证，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，将用户的背景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认证（MSTG-ARCH-2 和 MSTG-AUTH-1）

当测试身份认证和授权的时候执行以下步骤：

- 确定应用使用了额外的身份认证因素。
- 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。

参考来自 OWASP Web 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 的参数来传递的：

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 authenticated 参数改为“yes”，从而有效地绕过身份认证。

虽然这是一个简单的例子，你可能不会在真实世界发现，但程序员有时会依靠 "隐藏的 "客户端参数，如 cookies，来维持认证状态。他们认为这些参数是不能被篡改的。例如，考虑一下北电呼叫中心管理局的经典漏洞。北电设备的管理网络应用依靠 cookie "isAdmin "来确定登录的用户是否应该被授予管理权限。因此，可以通过简单地设置 cookie 的值来获得管理权限，如下所示：

isAdmin=True

安全专家过去建议使用基于会话的身份认证，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，使用无状态身份认证而不是基于会话的身份认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认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. 测试密码最佳实践（MSTG-AUTH-5 和 MSTG-AUTH-6）

当密码被用于认证时，密码强度是一个关键问题。密码策略规定了终端用户应该遵守的要求。密码策略通常规定了密码长度、密码复杂性和密码拓扑结构。一个 "强大"的密码策略使人工或自动

破解密码变得困难或不可能。下面的章节将涵盖有关密码最佳实践的各个领域。更多信息请参考 OWASP 身份认证备忘录。

4.4.3.1. 静态分析

确认密码策略的存在，并根据 OWASP 身份认证备忘单来验证密码复杂度要求，该备忘单侧重于长度和无限字符集。确定源代码中所有与密码有关的功能，并确保每个功能都进行了验证检查。审查密码验证功能，确保它能拒绝违反密码策略的密码。

4.4.3.1.1. zxcvbn

zxcvbn 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。一旦安装完成，zxcvbn 可以用来计算破解这个密码所需的复杂度和猜测次数。

把 zxcvbn 的 Javascript 库添加到 HTML 页面，您就可以在浏览器 console 里执行 zxcvbn 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。

```
> zxcvbn('ThisShouldBeVeryHardToCrack!')
< ▼{password: "ThisShouldBeVeryHardToCrack!", guesses: 9.71881e+21, guesses_log10: 21.98761309187359, sequence: Array
  (5), calc_time: 14, …} i
    calc_time: 14
  ▶ crack_times_display: {online_throttling_100_per_hour: "centuries", online_no_throttling_10_per_second: "centuri…
  ▶ crack_times_seconds: {online_throttling_100_per_hour: 3.4987716e+23, online_no_throttling_10_per_second: 971881…
  ▶ feedback: {warning: "", suggestions: Array(0)}
    guesses:
    9.71881e+21
    guesses_log10: 21.98761309187359
    password: "ThisShouldBeVeryHardToCrack!"
    score: 4
  ▶ sequence: (5) [{…}, {…}, {…}, {…}, {…}]
  ▶ __proto__: Object
```

该分数的定义如下，可以用于密码强度条等：

0#太容易猜测：危险密码。（guesses < 10^3）
1#非常容易猜测：受限范围的网络攻击防范。（guesses < 10^6）
2#可以猜测的：不受限范围网络攻击防范。（guesses < 10^8）
3#安全不可猜测：对离线慢散列场景的中度保护。（guesses < 10^10）
4#几乎不可猜测：对离线慢散列场景的重度保护。（guesses >= 10^10）

请注意，应用程序开发人员也可以使用 Java（或其他）实现来实现 zxcvbn，以便引导用户创建强密码。

4.4.3.2. 我的密码是否泄露：泄露的密码

为了进一步降低针对单因素身份认证方案（例如仅密码）的字典攻击成功的可能性，您可以验证密码是否在数据泄露中被泄露。这可以通过特洛伊·亨特（Troy Hunt）基于泄露密码 API 的服务来实现（可在 api.pwnedpasswords.com 上找到）。例如，“我的密码泄露了吗?”网站。基于可能的候选密码的 SHA-1 哈希，API 返回在服务收集的各种泄露行为中哈希出现的次数。工作流将执行以下步骤：

- 将用户输入编码为 UTF-8（例如：密码 test）。
- 取步骤 1 结果的 SHA-1 哈希（例如：test 哈希为 A94A8FE5CC...）。
- 复制前 5 个字符（哈希前缀），并使用以下 API 将其用于范围搜索：http GET https://api.pwnedpasswords.com/range/A94A8
- 遍历结果并查找散列的其余部分（例如，FE5CC...是返回列表的一部分吗？）。如果它不是返回列表的一部分，则未找到给定哈希的密码。否则，如 FE5CC。。。，它将返回一个计数器，显示发现泄露的次数（例如：FE5CC...：76479）。

有关 Pwned 密码 API 的更多文档可以在线访问。

请注意，此 API 的最佳使用场景是应用程序开发人员用于在用户需要注册并输入密码时检查其是否为推荐密码。

4.4.3.2.1. 登录限制

检查限制过程的源代码：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。一次被接受的登录尝试后，错误计数器应该被重置。

在实施反暴力破解控制时，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。

4.4.3.3. 动态测试（MSTG-AUTH-6）

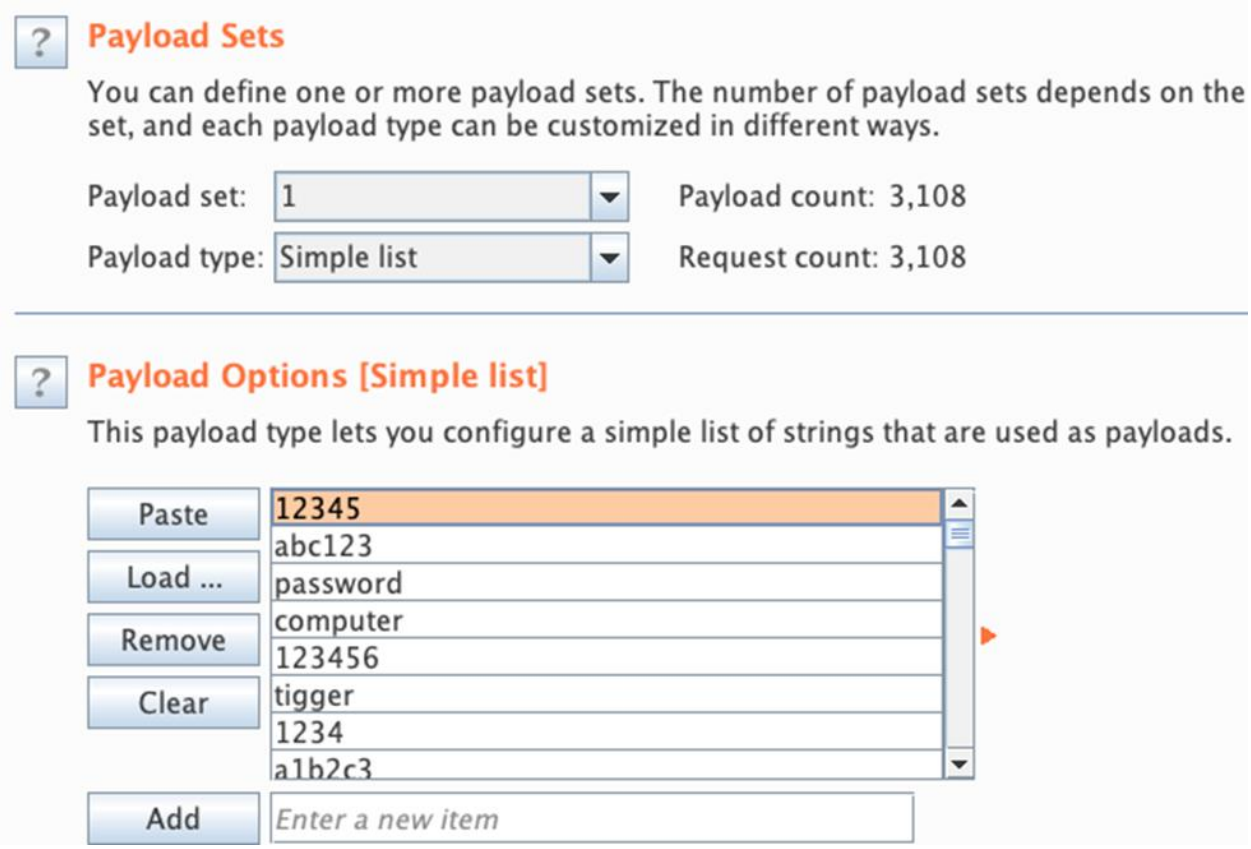
自动化密码猜测攻击可以用很多工具来实施。对于 HTTP(s)服务，使用拦截代理是一个切实可行的选项。例如：您可以使用 Burp Suite Intruder 来执行基于字典的和暴力破解攻击。

请记住，当使用 Burp Suite Community Edition 时，会在多次请求后激活一个限制机制，这将大大减慢您使用 Burp Intruder 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。

使用 Burp Intruder 进行基于字典的暴力攻击时，执行以下步骤：

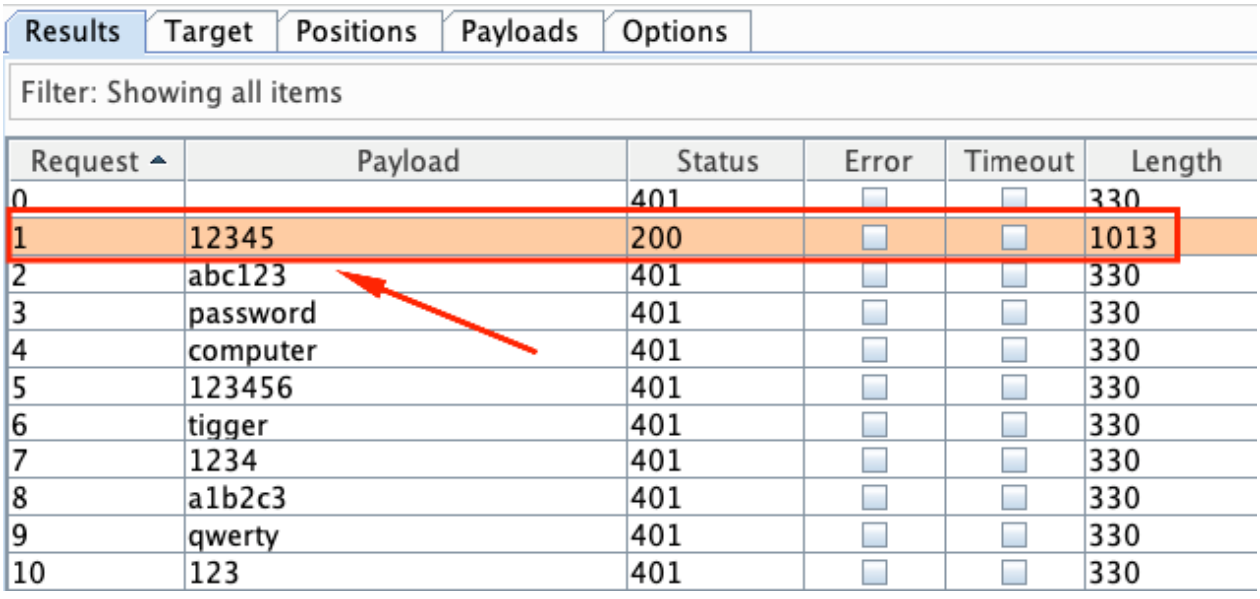
- 启动 Burp Suite Professional。
- 创建一个新的项目（或者打开现有的一个）。
- 设置您的移动设备，让它使用 Burp 作为 HTTP/HTTPS 代理。登录到移动应用上，并且拦截发送到后端服务的身份认证请求。
- 右键点击在“**Proxy/HTTP History**”标签页的请求，并且在弹出菜单选择“**Send to Intruder**”。
- 在 Burp Suite 里选择“**Intruder**”标签页。关于怎么使用 Burp Intruder 的更多信息，请参阅在 Portswigger 上的官方文档。
- 确保合理地设置了在“**Target**”、“**Positions**”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 Intruder 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，并且您选了一个字典，您就准备好可以开始攻击了！



- 点击“**Start attack**”按钮来攻击身份认证。

一个新窗口将被打开。将按顺序向站点发送请求，每个请求对应于列表中的一个密码。会提供每个请求的响应体的信息（长度、状态码等），让您区分成功和失败的尝试：



Request	Payload	Status	Error	Timeout	Length
0		401			330
1	12345	200			1013
2	abc123	401			330
3	password	401			330
4	computer	401			330
5	123456	401			330
6	tigger	401			330
7	1234	401			330
8	a1b2c3	401			330
9	qwerty	401			330
10	123	401			330

在这个例子里，您可以根据不同的长度和 HTTP 状态码来识别出成功的尝试，该尝试关联的密码是 12345。

为了测试你自己的测试账户是否容易被暴力破解，将你的测试账户的正确密码附加到密码列表的末尾。列表中不应该有超过 25 个密码。如果你能完成攻击而不永久或暂时锁定账户，或在使用错误密码的请求达到一定数量后无需验证码，这意味着账户没有受到暴力攻击的保护。

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”）的身份认证的特征是客户端和服务器上的身份认证记录。身份认证的流程如下：

1、 应用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 始终通过安全连接来交换（HTTPS）。
- 移动应用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应用中能被删除。

身份认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认证功能。如果应用用框架 API 来做身份认证，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认证的一个很棒的资源是 OWASP Web 测试指南，特别是测试身份认证和测试会话管理章节。

4.4.5. 测试会话超时(MSTG-AUTH-7)

将会话 ID 和令牌生命周期最小化可以减少账户劫持成功的可能性。

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，视乎应用的敏感程度。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，请通过一个拦截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. 访问一个要求身份认证的资源，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超时后，校验超时时长对于应用程序来说是否合适。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

本测试用例的目的是验证登出功能，确定它是否有效地终止了客户端和服务器上的会话，并使无状态令牌无效。

未能销毁服务器端的会话是最常见的登出功能实现错误之一。这个错误使会话或令牌保持活力，甚至在用户登出应用程序之后。获得有效认证信息的攻击者可以继续使用它并劫持用户的账户。

许多移动应用程序不会自动登出用户。可能有各种原因，例如：因为对客户来说不方便，或在实施无状态认证时做出的决定。应用程序仍然应该有登出功能，而且应该按照最佳实践来实现，销毁所有本地存储的令牌或会话标识符。如果会话信息存储在服务器上，它也应该通过向该服务器发送登出请求来销毁。如果是高风险的应用程序，令牌应该被废止。如果不删除令牌或会话标识符，在令牌被泄露的情况下，会导致对应用程序的未授权访问。

请注意，其他敏感类型的信息也应该被删除，因为任何没有被正确清除的信息都可能在以后被泄露，例如在设备备份期间。

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。这个校验取决于使用的技术。这里有正确的用于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认证一起使用，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拦截代理，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、 访问一个要求身份认证的资源，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（WSTG-SESS-06）包含了详细的解释和更多测试用例。

4.4.7. 测试双因素认证和阶梯式认证（MSTG-AUTH-9 和 MSTG-AUTH-10）

双因素验证（2FA）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素，然后用以下的任意一种作为第二因素：

- 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。

无论第二个因素是什么，都必须在服务器端执行和验证，而不能在客户端执行和验证。否则，第二个因素可以在应用程序中轻松绕过。

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 登录一个银行应用后，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，第二因素（“额外身份认证”）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）是作为双因素认证的常见的第二因素，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认证器。”。以下是一些相关的威胁和建议，用以避免针对 SMS-OTP 的成功攻击。

威胁：

- 无线窃听：对手可以通过滥用 毫微蜂窝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。这包括用以双因素认证的消息。
- 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：当 SMS 不再是首选或可用时，一些双因素认证方案允许通过电话来发送 OTP。许多这样的电话，如果没有人接听，会把信息发送到语音信箱。如果攻击者能够进入语音信箱，他们也可以使用 OTP 来进入用户的账户。

您可以找到以下几条建议，以减少在将 SMS 用于 OTP 时被利用的可能性：

- 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 OTP 更难破解或猜测。
- 避免语音留言：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

实现第二因素的另一个替代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。应用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 （Android）或者 KeyChain（iOS)。为了授权一个事务，后端给移动应用发送一个包含事务数据的推送消息。用户然后就被询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)，然后该数据就使用用户的私钥进行签名。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素验证机制，包括从第三方库、外部应用的使用到开发者自己实现的检查。

首先使用应用，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应用，请验证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

在用拦截代理捕获发往远程节点的请求的同时，全面地使用一下应用（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 和逐步提升身份认证后才有效的数据，那么身份认证检查就没有在被正确地实现。

当使用 OTP 身份认证的时候，考虑大多数 OTP 是短的数字值。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素。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%。

为了测试这一点，在提供正确的 OTP 之前，应该利用捕获的请求发送 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，后续提供的 OTP 必须视为失效，用户应该被重定向到登录页或登出。

你应该检查应用程序是否依赖来自远程端点的静态响应，如 "message":"Success"，以授予对应用程序内部敏感数据或功能的访问。如果是这样的话，攻击者可以通过操纵服务器响应轻松绕过 2FA 实现

例如，通过使用拦截代理（如 Burp Suite）并将响应修改为 "message": "Success"。

为了防止这类攻击，应用程序应该始终验证某种用户令牌或其他与用户有关的动态信息，这些信息以前是安全存储的（例如，在 Keychain/KeyStore 中）。

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）的身份认证（MSTG-AUTH-3）

基于令牌的身份认证由用 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认证信息。

JWT 令牌由三个由点分割的 Base64 编码部分组成。令牌结构如下：

base64UrlEncode(header).base64UrlEncode(payload).base64UrlEncode(signature)

以下示例展示了一个 Base64 编码的 JSON 网页令牌：
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG
4gRG9lIiwiYWRt aW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

头部通常由两部分组成：令牌类型是 JWT 以及正在用于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认证服务获得。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 库是否存在任何已知威胁。

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）声明来解决重放攻击，这为 JWT 提供了一个唯一标识符。
- 确保使用 aud（audience）声明解决跨服务重放攻击，该声明定义令牌有权用于哪个应用程序。
- 确保令牌用例如 KeyChain（iOS）或 KeyStore（Android）安全的存储在移动手机中。

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 密钥--区块序列化并作为字符串存储在 JVM 内存中
private transient byte[] keyHMAC = ...;

//为令牌创建一个验证上下文，明确要求使用 HMAC-256 HMAC 生成。

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

//验证令牌；如果验证失败，将抛出一个异常。

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

4.4.8.1.2. 令牌过期

一旦签名，无状态身份认证令牌就永久有效，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效的访问令牌和一个长效的刷新令牌。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应用，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('/renew_access_token', function (req, res) {
  // 验证现有的刷新令牌
  var profile = jwt.verify(req.body.token, secret);

  // 如果刷新令牌超过 14 天，强制登录
  if (profile.original_iat - new Date() > 14) { // iat == issued at
    return res.send(401); // re-login
  }

  // 检查用户是否仍然存在或授权是否被撤销
  if (!valid) return res.send(401); // re-logging

  // 发行新的访问令牌
  var renewed_access_token = jwt.sign(profile, secret, { expiresInMinutes: 60
*5 });
  res.json({ token: renewed_access_token });
});
```

4.4.8.2. 动态分析

当执行动态分析的时候，调研以下 JWT 威胁：

- 在客户端的令牌存储：
  - 对于使用 JWT 的移动应用来说，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，包括用户身份认证应用程序。

OAuth2 的通常用途包括：

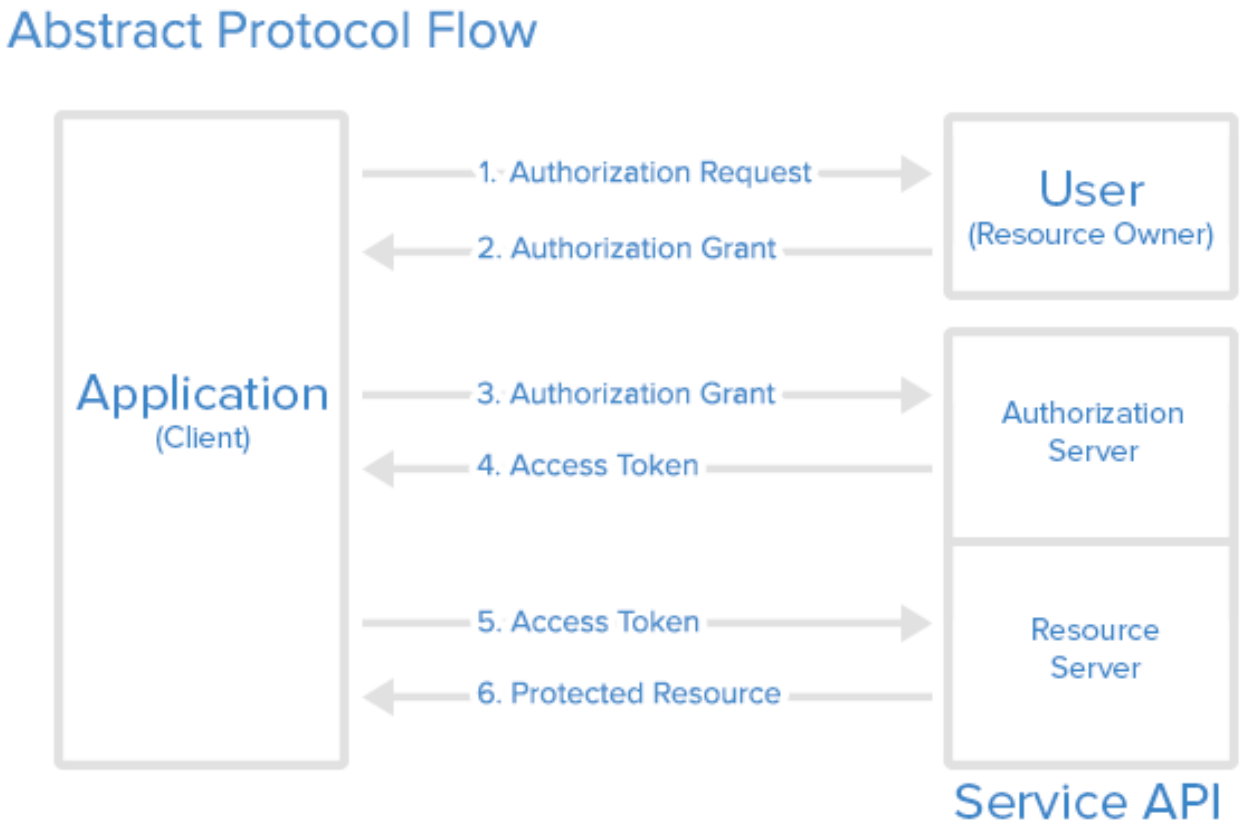
- 从用户处获得许可，以实现用他们的账户访问一个在线服务。
- 代表用户对在线服务进行身份认证。
- 处理身份认证错误。

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认证服务器进行身份认证。有了用户的许可，身份认证服务器然后就会发布一个令牌使得应用可以代表用户。需要注意的是 OAuth2 特性并没有定义任何特别的身份认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认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，完成身份认证过程。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认证。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：

- 原生应用上，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）。
- 认证代码应该是短效的，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. 外部用户代理和内置用户代理

OAuth2 认证可以通过外部用户代理（如 Chrome 或 Safari）或在应用程序本身（如通过内置到应用程序的 WebView 或认证库）进行。这两种模式在本质上都不是 "更好 "的--相反，选择什么模式取决于环境。

使用外部用户代理是需要与社交媒体账户（Facebook、Twitter 等）互动的应用程序的首选方法。这种方法的优点包括：

- 用户的凭据从不直接暴露给应用。这就保证了应用不能在逻辑过程（凭据钓鱼）期间获得凭据。
- 几乎不需要添加任何身份认证逻辑到应用自身，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。

对于在一个封闭的生态系统内运作的应用程序，内置认证是更好的选择。例如，考虑一个银行应用程序，它使用 OAuth2 从银行的认证服务器获取一个访问令牌，然后用来访问一些微服务。在这种情况下，凭证钓鱼并不是一个可行的方案。最好是将认证过程保留在（希望是）谨慎安全的银行应用程序中，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 - OAuth 2.0 授权框架
- RFC8252 – 原生应用使用的 OAuth 2.0 (October 2017)
- 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：“他们应该通知用户应用程序内的所有登录活动，以阻止某些特定的设备”。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应用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、设备和应用版本）。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。

开发人员可以使用特定的元信息，并将其与应用程序中的每个不同活动或事件相关联。这将使用户更容易发现可疑行为并阻止相应的设备。元信息可能包括：

- 设备：用户可以清楚地识别正在使用应用程序的所有设备。
- 日期和时间：用户可以清楚地看到应用程序使用的最新日期和时间。
- 位置：用户可以清楚地识别最近使用应用程序的位置。

应用程序可以提供活动历史记录列表，在应用程序中的每个敏感活动之后将更新该列表。根据每个应用程序处理的数据和团队愿意承担的安全风险级别，选择需要对其进行审核的活动。以下是通常被审计的常见敏感活动列表：

- 登录尝试
- 密码更改
- 个人身份信息变更（姓名、电子邮件地址、电话号码等）
- 敏感活动（购买、获取重要资源等）
- 同意条款和协议

付费内容需要特别注意，可能会需要使用额外的元信息（如运营成本、信用信息等）来确保用户了解整个操作的范围。

此外，应将不可否认机制应用于敏感交易（例如付费内容访问、同意条款和协议等），以证明特定事务（完整性）实际上是由特定人员（认证）执行的。

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，应该测试应用程序与实际设备的绑定情况。在 iOS 中，开发人员可以使用 identifierForVendor，这与包 ID 相关：更改包 ID 时，该方法将返回不同的值。第一次运行应用程序时，请确保将 identifierForVendor 返回的值存储到 KeyChain 中，以便可以在早期检测到对其的更改。

在 Android 中，使用 Android 8.0（API 级别 26）之前版本的开发人员可以使用 `Settings.Secure.ANDROID_ID` 识别应用程序实例。请注意，从 Android 8.0（API 级别 26）开始，`ANDROID_ID` 不再是设备唯一 ID。相反，它由应用程序签名密钥、用户和设备的组合来确定。因此，对于这些 Android 版本，想通过验证 `ANDROID_ID` 以阻止设备可能很棘手。因为如果应用程序更改其签名密钥，`ANDROID_ID` 将更改，并且无法识别以前用户的设备。因此，最好使用 `AndroidKeyStore` 中随机生成的密钥（最好是 AES_GCM 加密）将 ANDROID_ID 加密并私下存储在私有共享配置文件中。当应用程序签名更改时，应用程序可以检查增量并注册新的 `ANDROID_ID`。当此新 ID 更改时,而并没有新的应用程序签名密钥，则表明存在其他问题。接下来，可以通过使用存储在 iOS Keychain 和 Android KeyStore 中的密钥对请求进行签名来扩展设备绑定，从而保证强大的设备绑定。您还应该测试使用不同的 IP、不同的位置和/或不同的时隙是否会在所有场景中触发正确类型的信息。

最后，设备的阻断应该被测试到，通过阻止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，而不是在注册完应用程序第二个实例时才警告用户。

4.4.11. 参考文献

4.4.11.1. OWASP MASVS

- MSTG-ARCH-2：“安全控制从来不只在客户端执行，而是在各自的远程节点执行。”
- MSTG-AUTH-1：“如果该应用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认证，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认证，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认证，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：“当不正确的身份认证凭据提交多次时，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素身份认证，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认证。"
- MSTG-AUTH-11：“应用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。”

4.4.11.2. SMS-OTP 研究

- Dmitrienko、Alexandra 等人。“关于移动双因素身份认证的安全性。”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：身份认证和生命周期管理(草案)。No.特别出版物(NIST SP)-800-63B.2016。
  - Grassi、PaulA 等人。数字身份指南:身份认证和生命周期管理。No. 特别出版物(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认证。”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] Siadati, Hossein, 等人. “注意你的短信。缓解第二因素认证中的社会工程.” 计算机与安全 65 (2017): 14-28.
- Siadati，Hossein 等人。“小心您的短信：缓解第二因素身份认证中的社交工程。”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。

4.5. 移动应用网络通信测试

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. 安全连接

仅使用明文 HTTP 是合理的，而使用 HTTPS 来保证 HTTP 连接的安全通常是无用的，这种时代早已过去。HTTPS 本质上是将 HTTP 层在另一个被称为传输层安全（TLS）协议之上。TLS 使用公钥加密技术进行握手，并在完成后创建一个安全连接。

HTTPS 连接被认为是安全的，因为有三个特性。

- 保密性：TLS 在通过网络发送数据之前对其进行加密，这意味着它不能被中间人读取。
- 完整性：数据不能被改变而不被发现。
- 认证：客户端可以验证服务器的身份，以确保与正确的服务器建立连接。

4.5.2. 服务器证书评估

认证机构（CA）是客户端服务端安全通信的一个组成部分，它们在每个操作系统的证书存储库中都是预先设定好的。例如，在 iOS 上有超过 200 个根证书被安装（见 Apple 文档--Apple 操作系统可用的受信任根证书）

CA 可以被添加到证书存储库中，可以由用户手动添加，也可以由管理企业设备的 MDM 添加，或者通过恶意软件添加。那么问题来了。"你能信任所有这些 CA 吗，你的应用程序应该依靠默认的信任源吗？"。毕竟，有一些众所周知的案例，认证机构被入侵或被欺骗向冒牌货发放证书。在 sslmate.com 网站上可以找到 CA 被入侵和失败的详细时间线。

Android 和 iOS 都允许用户安装额外的 CA 或证书存储库。

一个应用程序可能想信任一组自定义的 CA，而不是平台默认的。最常见的原因是。

- 连接到具有自定义认证机构（尚未被系统知道或信任的 CA）的主机，如自签名的 CA 或公司内部发行的 CA。
- 将 CA 的集合限制在一个特定的受信任的 CA 列表中。
- 信任未包括在系统中的其他 CA。

4.5.2.1. 关于证书存储库

4.5.2.1.1. 扩展信任

每当应用程序连接到一个服务器，而该服务器的证书是自签名的或系统未知的，安全连接将失败。这种情况通常发生在任何非公共 CA 上，例如那些由政府、公司或教育机构等组织发行的、供其自己使用的 CA。

Android 和 iOS 都有扩展信任的手段，即包括额外的 CA，以便应用程序信任系统的内置 CA 和自定义 CA。

然而，请记住，设备用户总是能够包含额外的 CA。因此，根据应用程序的威胁模型，可能有必要避免信任任何添加到用户证书存储库的证书，甚至更进一步，只信任预先指定的特定证书或证书集。

对于许多应用程序来说，移动平台提供的 "默认行为 "对于他们的使用情况来说是足够安全的（在极少数情况下，系统信任的 CA 被破坏，应用程序处理的数据不被认为是敏感的，或者采取其他安全措施，甚至对这样的 CA 破坏有弹性）。然而，对于其他应用程序，如金融或健康应用程序，必须考虑到 CA 破坏的风险，即使是罕见的。

4.5.2.2. 限制信任：身份固定

一些应用程序可能需要通过限制其信任的 CA 数量来进一步提高其安全性。通常情况下，只有开发者使用的 CA 被明确地信任，而不考虑其他所有的 CA。这种信任限制被称为 "身份固定"，通常以 "证书固定 "或 "公钥固定 "的形式实现。

> 在 OWASP MASTG 中，我们将把这个术语称为 "认证固定"、"证书固定"、"公钥固定 "或简称 "固定"。

固定是将远程端点与一个特定的身份，如 X.509 证书或公钥联系起来的过程，而不是接受任何由受信任的 CA 签署的证书。在锁定服务器身份（或某个集合，又称 pinset）后，移动应用随后将只在身份匹配的情况下连接到这些远程端点。阻止信任不必要的 CA，可以减少应用程序的攻击面。

4.5.2.3. 通用指南

OWASP 证书固定备忘录在以下方面提供了基本指导：

- 什么时候推荐使用固定，哪些例外情况可能适用。

- 何时固定：开发时（预加载）或首次遇到时（首次使用时的信任）。
- 固定什么：证书、公钥或哈希值。

Android 和 iOS 的建议都符合 "最佳示例"，即:

- 只对开发者可以控制的远程端点进行固定。
- 在开发时通过（NSC/ATS）进行
- 固定一个 SPKI subjectPublicKeyInfo 的哈希值。

自几年前引入以来，固定已经获得了不好的名声。我们想澄清几点，其至少对移动应用安全是有效的:

- 坏名声是由于操作上的原因（如实施/固定管理的复杂性）而不是缺乏安全性。
- 如果一个应用程序没有实施固定，这不应该被报告为一个漏洞。但是，如果该应用必须通过 MASVS-L2 验证，则必须实现。
- Android 和 iOS 系统都能很容易地实现固定，并遵循最佳实践。
- 固定可以防止被破坏的 CA 或安装在设备上的恶意 CA。在这些情况下，固定将防止操作系统与恶意服务器建立安全连接。然而，如果攻击者控制了设备，他们可以很容易地禁用任何固定逻辑，从而仍然允许连接的发生。因此，这并不能阻止攻击者访问你的后端和滥用服务器端的漏洞。
- 移动应用程序中的固定与 HTTP 公钥固定（HPKP）不一样。在网站上不再推荐使用 HPKP 头，因为它可能导致用户被锁定在网站之外，而没有任何办法解除锁定。对于移动应用程序来说，这不是一个问题，因为一旦有任何问题，应用程序总是可以通过带外渠道（即应用程序商店）进行更新。

4.5.2.4. 关于 Android 开发者的固定推荐

Android 开发者网站包含以下警告:

> 注意事项：由于未来服务器配置变化的风险很高，例如改变为另一个认证机构，导致应用程序在没有收到客户端软件更新的情况下无法连接到服务器，因此不建议 Android 应用程序使用证书固定。

他们还包括这样的说明:

> 请注意，当使用证书固定时，你应该总是包括一个备份密钥，这样，如果你被迫切换到新的密钥或改变 CA（当固定一个 CA 证书或该 CA 的中间机构时），你的应用程序的连接性就不会被影响。否则，你必须向应用程序推送更新以恢复连接性。

第一项声明可能被误解为他们 "不推荐使用证书固定"。第二句声明说明了这一点：实际的建议是，如果开发者想实施固定，他们必须采取必要的预防措施。

4.5.2.5. 关于 iOS 开发者的固定推荐

Apple 公司建议从长远考虑，建立一个适当的服务器认证策略。

4.5.2.6. OWASP MASTG 建议

固定是一个值得推荐的做法，特别是对于 MASVS-L2 的应用。但是，开发者必须在他们控制的端点上专门实施这种做法，并确保包括备份密钥（又称备份固定），并有一个适当的应用程序更新策略。

4.5.2.7. 了解更多

- Android 安全：SSL 固定
- OWASP 证书固定备忘录

4.5.3. HTTP(S)流量拦截

通常，最通用的做法是在移动设备上配置系统代理，以便通过主机上运行的拦截代理重定向 HTTP（S）流量。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

要使用拦截代理，你需要在你的主机上运行它，并配置移动应用将 HTTP(S)请求路由到你的代理。在大多数情况下，在移动设备的网络设置中设置一个全系统的代理就足够了--如果应用程序使用标准的 HTTP APIs 或流行的库，如 `okhttp`，它将自动使用系统设置。

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

4.5.4. 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这些流量。

然而，有一些可用的 Burp 插件，例如：

- Burp-non-HTTP-Extension
- 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。

4.5.5. 拦截应用进程中的流量

根据你在测试应用程序时的目标，有时在流量到达网络层之前或在应用程序中收到响应时监测它就足够了。

如果你只是想知道某条敏感数据是否被传输到网络上，你不需要使用完全伪造的 MITM 攻击。在这种情况下，你甚至不需要绕过固定，如果实施的话。你只需要劫持正确的函数，例如 openssl 的 SSL_write 和 SSL_read。

这对使用标准 API 库函数和类的应用程序来说效果很好，但也可能有一些缺点：

- 该应用可能实现了一个自定义的网络堆栈，你必须花时间分析该应用以找出你可以使用的 API（见本博文 "用签名分析搜索 OpenSSL 跟踪 "部分）。
- 要制作正确的劫持脚本来重新组合 HTTP 响应对（涉及许多方法调用和执行线程）可能非常耗时。你可能会找到现成的脚本，甚至是替代的网络堆栈，但根据应用程序和平台，这些脚本可能需要大量的修改，而且不一定能工作。。

一些示例如下：

- "通用拦截。如何绕过 SSL 固定并监控任何应用程序的流量"，"在传输前抓取有效载荷 "和 "在加密前抓取有效载荷 "部分
- "Frida 作为网络跟踪的替代品"

  这种技术也适用于其他类型的流量，如 BLE、NFC 等，在这些地方部署 MITM 攻击可能非常昂贵或复杂。

4.5.6. 拦截网络层流量

如果应用中使用了标准库，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，其忽略系统代理设置。
- 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，幸运的是，有几种重定向和拦截网络通信的选项：

  - 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。

- 有时您需要执行 MITM 攻击，以迫使移动设备与您通话。对于这种情况，您应该考虑使用 bettercap 或使用您自己的接入点将网络流量从移动设备重定向到主机（见下文）。
- 在获取了 root 权限的设备（越狱机）上，可以使用劫持或代码注入，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。

4.5.6.1. 使用 bettercap 模拟中间人攻击

4.5.6.1.1. 网络设置

为了进入中间人攻击位置，您的主机应该与移动电话及其通信网关位于同一无线网络中。完成后，您需要知道手机的 IP 地址。为了对移动应用程序进行全面的动态分析，应拦截所有网络流量。

4.5.6.1.2. MITM 攻击

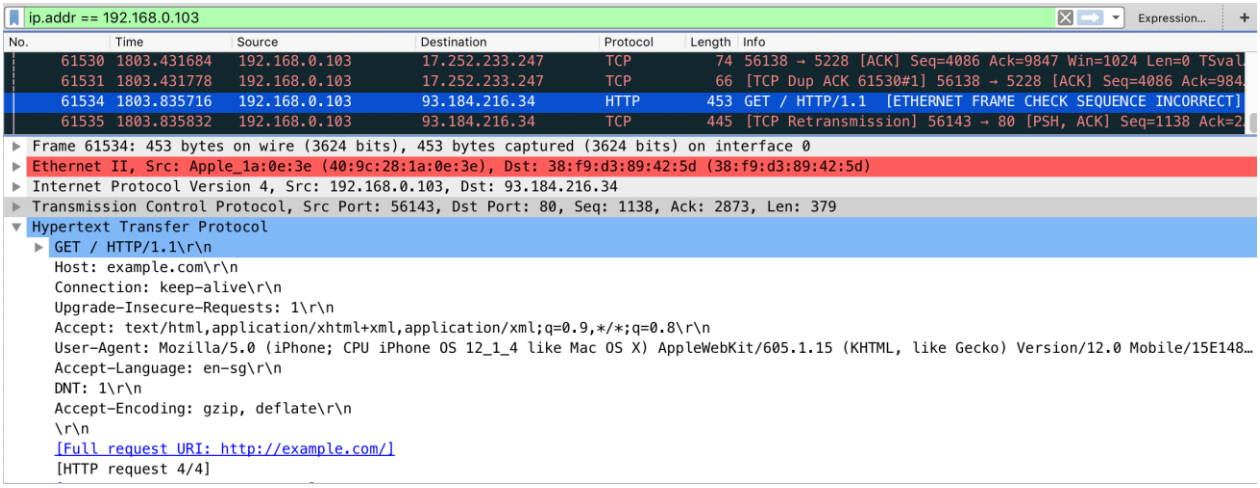
首先启动你的首选网络分析器工具，然后使用以下命令启动 bettercap，并将下面的 IP 地址（X.X.X.X）替换为要对其执行 MITM 攻击的目标。

```
$ 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 lis
t 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 随后将自动将数据包发送到（无线）网络中的网络网关，您可以开始嗅探流量。2019 年初，bettercap 增加了对全双工 ARP 欺骗的支持。

在手机上启动浏览器并导航到 `http://example.com`，当您使用 Wireshark 时，应该会看到如下输出。



如果是这种情况，您现在可以看到手机发送和接收的完整网络流量。这还包括 DNS、DHCP 和任何其他形式的通信，因此可能非常“嘈杂”。因此，您应该知道如何在 Wireshark 中使用 DisplayFilters，或者知道如何在 tcpdump 中进行筛选，以便只关注相关的流量。

> 中间人攻击针对任何设备和操作系统，因为该攻击是通过 ARP 欺骗在 OSI 第 2 层上执行。当您是 MITM 时，您可能无法看到明文数据，因为传输中的数据可能会使用 TLS 进行加密，但它会为您提供有关所涉及主机、使用的协议以及应用程序与之通信的端口的宝贵信息。

4.5.6.2. 使用访问点模拟中间人攻击

4.5.6.2.1. 网络设置

模拟中间人（MITM）攻击的一种简单方法是配置一个网络，其中范围内的设备和目标网络之间的所有数据包都要通过您的主机。在移动设备渗透测试中，这可以通过使用移动设备和主机连接的接入点来实现。然后，您的主机将成为路由器和访问点。

可能出现以下情况：

- 使用主机的内置 WiFi 卡作为接入点，并使用有线连接连接到目标网络。
- 使用外部 USB WiFi 卡作为接入点，并使用主机内置 WiFi 连接到目标网络（反之亦然）。
- 使用单独的访问点并将流量重定向到主机。

使用外部 USB WiFi 卡的场景要求该卡能够创建接入点。此外，您需要安装一些工具和/或配置网络以实现强制中间人（见下文）。您可以在 Kali Linux 上使用 `iwconfig` 命令验证 WiFi 卡是否具有 AP 功能：

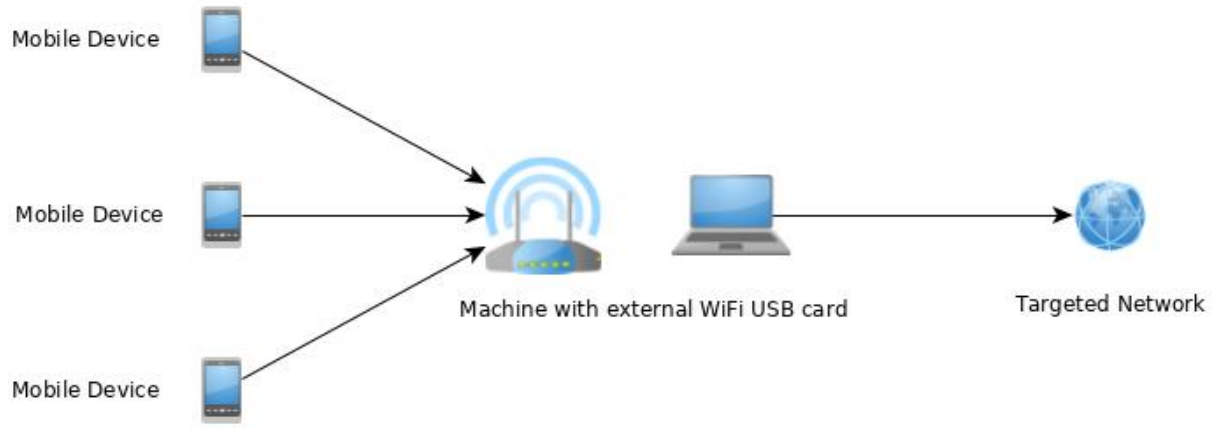
```
iw list | grep AP
```

具有单独接入点的场景需要修改 AP 的配置，您应该首先检查 AP 是否支持：

- 端口转发
- 具有分光或镜像端口。

在这两种情况下，AP 都需要配置为指向主机的 IP。您的主机必须连接到 AP（通过有线连接或 WiFi），并且您需要连接到目标网络（可以使用到 AP 相同的连接）。可能需要在主机上进行一些其他配置，以将流量路由到目标网络。

> 如果单独的接入点属于客户，则应在接洽之前澄清所有更改和配置，并在进行任何更改之前创建备份。



4.5.6.2.2. 安装

以下是通过接入点和额外网络接口建立中间人劫持位置的步骤：

通过单独的接入点、外部 USB WiFi 卡或主机的内置卡创建 WiFi 网络。

这可以通过使用 macOS 上的内置实用程序来完成。您可以使用在 Mac 上与其他网络用户共享 internet 连接。

对于所有主要的 Linux 和 Unix 操作系统，您都需要以下工具：

- hostapd
- dnsmasq
- iptables
- wpa_supplicant
- airmon-ng

在 Kali 中你可以通过 apt-get 安装：

```
apt-get update
apt-get install hostapd dnsmasq aircrack-ng
```

Iptables 和 wpa_supplicant 已经默认安装在 Kali 中

如果是一个单独的接入点，将流量路由到你的主机。如果是外部 USB WiFi 卡或内置 WiFi 卡，流量已经在你的主机上可用。

将来自 WiFi 的传入流量路由到附加网络接口，流量可以到达目标网络。额外的网络接口可以是有线连接或其他 WiFi 卡，取决于你的设置。

4.5.6.2.3. 配置

我们主要以 Kali Linux 的配置文件为例。需要定义以下值：

- wlan1 -AP 网络接口的 id（具有 AP 功能），
- wlan0 -目标网络接口的 id（可以是有线接口或其他 WiFi 卡）
- 10.0.0.0/24 -AP 网络的 IP 地址和掩码

需要相应地更改和调整以下配置文件：

- hostapd.conf

```
# 我们使用的 WiFi 接口名称
interface=wlan1
# 使用 nl80211 驱动
driver=nl80211
hw_mode=g
channel=6
wmm_enabled=1
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
```

```
wpa_key_mgmt=WPA-PSK
rsn_pairwise=CCMP
# AP 网络名称
ssid=STM-AP
# AP 网络密码
wpa_passphrase=password
```

- wpa_supplicant.conf

```
network={
    ssid="NAME_OF_THE_TARGET_NETWORK"
    psk="PASSWORD_OF_THE_TARGET_NETWORK"
}
```

- dnsmasq.conf

```
interface=wlan1
dhcp-range=10.0.0.10,10.0.0.250,12h
dhcp-option=3,10.0.0.1
dhcp-option=6,10.0.0.1
server=8.8.8.8
log-queries
log-dhcp
listen-address=127.0.0.1
```

4.5.6.2.4. MITM 攻击

为了获得中间人攻击位置，您需要运行以下配置。这可以通过在 Kali Linux 上使用以下命令来完成：

```
# 检查没有别的进程在使用 WiFi 接口
$ airmon-ng check kill
# 配置 AP 网络接口的 IP 地址
$ ifconfig wlan1 10.0.0.1 up
# 开启接入点
$ hostapd hostapd.conf
# 连接到目标网络接口
$ wpa_supplicant -B -i wlan0 -c wpa_supplicant.conf
# 运行 DNS 服务器
$ dnsmasq -C dnsmasq.conf -d
# 启用路由
$ echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables 会 NAT 从 AP 网络接口到目标网络接口的连接
$ iptables --flush
```

```
$ iptables --table nat --append POSTROUTING --out-interface wlan0 -j MASQUERA
DE
$ iptables --append FORWARD --in-interface wlan1 -j ACCEPT
$ iptables -t nat -A POSTROUTING -j MASQUERADE
```

现在，您可以将移动设备连接到接入点。

4.5.6.3. 网络分析工具

安装一个允许监视和分析重定向到计算机上的网络流量的工具。两种最常见的网络监视（或捕获）工具是：

- Wireshark (CLI 工具: TShark)
- tcpdump

Wireshark 提供了一个 GUI，如果你不习惯用命令行的话，它就更直接了。如果你正在寻找一个命令行工具，你应该使用 TShark 或 tcpdump。所有这些工具都可用于所有主要的 Linux 和 Unix 操作系统，并且应该是其各自软件包安装机制的一部分。

4.5.6.4. 通过运行时设置代理

在 root 过的设备或越狱设备上，还可以使用运行时劫持设置新代理或重定向网络流量。这可以通过 Inspeckage 等劫持工具或 Frida 和 cycript 等代码注入框架来实现。您将在本指南的“逆向工程和篡改”章节中找到有关运行时插件的更多信息。

4.5.6.5. 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#作为编程语言生成原生 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。有三种方法可以解决这个问题：

- 第一种方法：在 OnCreate 或 Main 方法中添加以下代码，并重新创建应用程序，从而为应用程序添加一个默认代理：

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

- 第二种方法：使用 bettercap 来获得(MITM)中间人攻击的位置。参考上面设置 MITM 攻击的章节。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 -
```

- 在 Linux 系统中可以使用 `iptables`：

```
sudo iptables -t nat -A PREROUTING -p tcp --dport 443 -j DNAT --to-destin
ation 127.0.0.1:8080
```

- 作为最后一步，您需要在 Burp Suite 的 listener 设置中设置选项“支持不可见代理(Support invisible proxy)”。

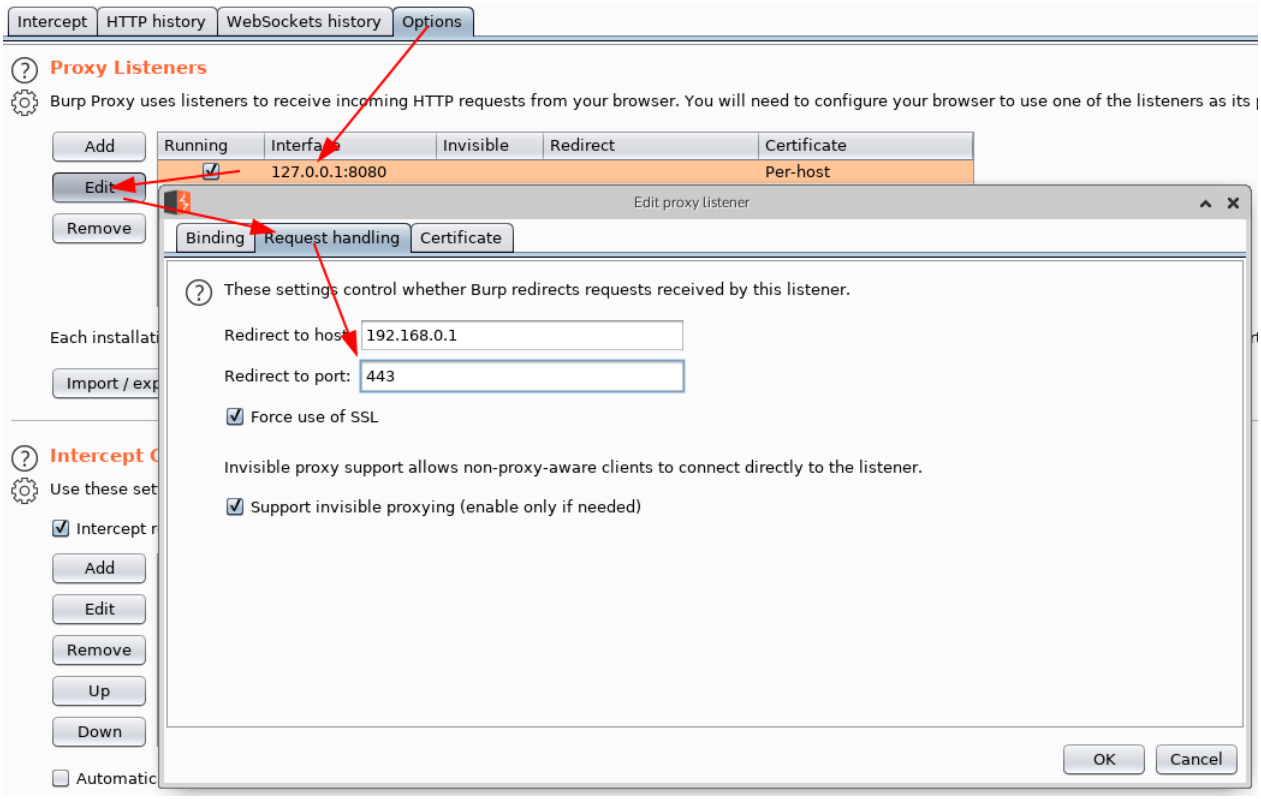
- 第三种方法：代替 bettercap 的另一种选择是调整手机上的`/etc/hosts`。在`/etc/hosts`中添加一条指向目标域名的记录，并将其指向拦截代理的 IP 地址。这可以实现与 bettercap 类似的 MiTM 情况，您需要将端口 443 重定向到拦截代理使用的端口。可以如上所述进行重定向。此外，您需要将流量从拦截代理重定向到原始位置和端口。

  重定向流量时，应创建范围内的精确的域名和 IP 规则，以将干扰和范围外流量降至最低。

拦截代理需要监听上面端口转发规则中指定的端口，即 8080。

当 Xamarin 应用程序配置为使用代理（例如，通过使用 WebRequest.DefaultWebProxy）时，您需要在将流量重定向到您的拦截代理后，指定流量下一步的去向。您需要将流量重定向到原始位置。以下过程将在 Burp 中设置重定向到原始位置：

1、转到“代理 **Proxy**”选项卡并单击“选项 **Options**”
2、从代理侦听器 listeners 列表中选择并编辑您的侦听器 listeners。
3、转到“请求处理 **Request handling**”选项卡并设置：
   –重定向到主机 Redirect to host：提供原始流量位置。
   –重定向到端口 Redirect to port：提供原始端口位置。
   –设置“强制使用 SSL”（使用 HTTPS 时）并设置“支持不可见代理 Support invisible proxy”。



4.5.6.5.1. 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

4.5.6.5.2. 流量拦截

首先，启动应用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 Proxy 选项卡/选项 Options /编辑 Edit 界面中，激活“支持不可见代理 Support invisible proxying”。

4.5.7. 验证网络上的数据加密(MSTG-NETWORK-1)

更多信息请参考相应章节

- Android 网络通信
- iOS 网络通信

4.5.8. 验证 TLS 设置(MSTG-NETWORK-2)

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认证，并确保网络数据的机密性和完整性。

4.5.8.1. TLS 设置推荐

确保服务器端正确的 TLS 配置也很重要。SSL 协议已弃用，不应再使用。此外，TLS v1.0 和 TLS v1.1 都存在已知漏洞，到 2020 年，所有主要浏览器都不推荐使用这些协议。

TLS v1.2 和 TLS v1.3 被认为是数据安全传输的最佳实践。从 Android 10（API level 29）开始，默认情况下将启用 TLS v1.3，以实现更快、更安全的通信。TLS v1.3 的主要变化是不再可以自定义密码套件，并且在启用 TLS v1.3 时，所有密码套件都已启用，而且不支持零往返（0-RTT）模式。

当客户端和服务器都由同一组织控制并且仅用于彼此通信时，可以通过配置加固来提高安全性

移动应用程序应用连接到特定服务器时，应用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。

4.5.8.2. 密码套件术语

密码套件具有以下结构：

- **Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm**

这种结构包括：

- 加密使用协议。

- 在 TLS 握手过程中，服务器和客户端用于进行身份认证的密钥交换算法。
- 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。
- 用于身份认证的 RSA 非对称加密。
- 使用 3DE 的 EDE_CBC 模式进行对称加密。
- 使用 SHA 哈希作为完整性算法。

请注意，在 TLSv1.3 中，密钥交换算法不是密码套件的组成部分，而是在 TLS 握手期间确定的。

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。

以下资源包含最新推荐的用于 TLS 的密码套件：

- 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。

某些 Android 和 iOS 版本不支持某些推荐的密码套件，因此出于兼容性目的，您可以检查 Android 和 iOS 版本支持的密码套件，并选择最受支持的密码套件。

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。

最后，验证 HTTPS 连接终止所在的服务器或终端代理是否按照最佳实践进行了配置。请参阅 OWASP 传输层保护备忘录和 Qualys 的 SSL/TLS 部署最佳实践。

4.5.9. 确保关键操作使用安全通信通道 (MSTG-NETWORK-5)

4.5.9.1. 概述

对于敏感的应用程序，像银行类应用，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 作为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，即使提供了普通用户也不使用，因此无法保证使用电子邮件时数据的机密性。欺骗、（鱼叉式）网络钓鱼和垃圾邮件是通过滥用电子邮件欺骗用户的额外方式。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。

4.5.9.2. 静态分析

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

- 令牌(例如 RSA 令牌,YubiKey)。
- 推送通知（例如：Google 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口的数据（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，可以考虑通过 Google 认证器来使用一次性密码（OTP)。

4.5.9.3. 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。

4.5.10. 参考文献

4.5.10.1. 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。”

4.5.10.2. Android

- Android 支持的加密套件 - https://developer.android.com/reference/javax/net/ssl/SSLSocket#Cipher%20suites

- Android 文档: Android 10 变化 - https://developer.android.com/about/versions/10/behavior-changes-all

4.5.10.3. iOS

- iOS 支持的加密套件- https://developer.apple.com/documentation/security/1550981https://developer.apple.com/documentation/security/1550981-ssl_cipher_suite_values?language=objcssl_cipher_suite_values?language=objc

4.5.10.4. IANA 传输层安全（TLS）参数

- TLS 加密套件- https://www.iana.org/assignments/tls-parameters/tlshttps://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml - tls-parameters-4parameters.xhtml#tls-parameters-4

4.5.10.5. OWASP TLS 密码字符串备忘单

- 推荐加密字符串 - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/TLS_Cipher_String_C heat_Sheet.md

4.5.10.6. SIM 卡交换攻击

- The SIM 劫持 - https://motherboard.vice.com/en_us/article/vbqax3/hackers-simhttps://motherboard.vice.com/en_us/article/vbqax3/hackers-sim-swapping-steal-phone-numbers-instagram-bitcoinswapping-steal-phone-numbers-instagram-bitcoin
- SIM SIM 卡互换：手机安全功能如何导致银行账户被黑客攻击- https://www.fintechnews.org/sim-swapping-how-the-mobile-security-feature-can-lead-to-ahttps://www.fintechnews.org/sim-swapping-how-the-mobile-security-feature-can-lead-to-a-hacked-bank-account/hacked-bank-account/

4.5.10.7. NIST

- FIPS PUB 186 – 数字签名标准 (DSS)

4.5.10.8. SIM 卡交换欺诈

- https://motherboard.vice.com/en_us/article/vbqax3/hackers-sim-swapping-steal-phonehttps://motherboard.vice.com/en_us/article/vbqax3/hackers-sim-swapping-steal-phone-numbers-instagram-bitcoinnumbers-instagram-bitcoin
- 如何保护自己免受 SIM 卡交换攻击 - https://www.wired.com/story/sim-swaphttps://www.wired.com/story/sim-swap-attack-defend-phone/attack-defend-phone/

4.5.10.9. 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。机密性包括通过使用加密来确保数据隐私。数据完整性涉及数据的一致性以及通过使用散列法检测数据的篡改和修改。。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（用相同的密匙进行加密/解密）或非对称的（使用公钥私钥密钥对进行加密/解密）。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。由于每个能接触到密钥的人都能解密加密的内容，这种方法需要仔细的密钥管理和对密钥分配的集中控制。

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。用公钥加密的信息只能用私钥解密，反之亦然。此外，当输入的任何一个字节发生变化时，哈希值都会完全改变。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。HMAC 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 MAC 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（KDF）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。KDF 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 KDF 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）

在评估一个移动应用程序时，你应该确保它不使用具有重大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可能会随着时间的推移变得不安全；因此，定期检查当前的最佳实践并相应地调整配置是很重要的。

核实加密算法是最新的，并符合行业标准。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）以及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。认证并不能取代对算法可靠性的定期验证。具有已知弱点的算法应该用更安全的替代方案来代替。此外，用于加密的算法必须是标准化的，并开放供验证。使用任何未知的、或专有的算法对数据进行加密，可能会使应用程序受到不同的加密攻击，从而导致可恢复明文。

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）和损坏的随机数生成器（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，其长度至少应与散列函数输出相同，合理选择密码派生函数和迭代计数（如 PBKDF2、scrypt 或 bcrypt），IV 是随机的和唯一的，适用于特定的分组加密模式（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 CHA20-1305。
- 完整性算法：SHA-256、SHA-384、SHA-512、BLAKE3、SHA-3 家族。
- 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（2019）
- BSI 建议（2019）

4.6.3. 常见配置问题 (MSTG-CRYPTO-1, MSTG-CRYPTO-2 和 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长度不足的密钥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（MAC）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。

硬编码加密密钥意味着密钥是：

- 应用程序资源的一部分
- 可从已知值中得出的值
- 代码中硬编码

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原生代码、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。

如果应用程序使用双向 TLS（服务器和客户端证书都需要验证），请确保：

- 客户端证书的密码未存储在本地或锁定在设备 Keychain 中。
- 客户端证书并非在所有安装之间共享。

如果应用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 上的数据存储”和“iOS 上的数据存储”章节。

4.6.3.3. 弱密钥生成功能

密码算法（如对称加密或某些密钥散列）需要给定大小的安全输入。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 次以及对于用户感知性能不重要的关键密钥，至少 10000000 个。对于关键密钥，建议考虑实现由密码哈希竞争（PHC）识别的算法，如 Argon2。

4.6.3.4. 弱随机数发生器

从根本上说，在任何确定的设备上产生真正的随机数是不可能的。伪随机数生成器（RNG）通过产生一个伪随机数流来弥补这一点--一个看起来好像是随机生成的数字流。生成的数字的质量随所用算法的类型而变化。加密安全的 RNG 产生的随机数能通过统计学上的随机性测试，并能抵御预测攻击（例如，从统计学上来说，预测产生的下一个数字是不可行的）。

移动 SDK 提供了 RNG 算法的标准实现，可以产生具有足够人工随机性的数字。我们将在 Android 和 iOS 的具体章节中介绍可用的 API。

4.6.3.5. 自定义加密实现

发明专有的加密功能既费时又费力，而且很可能会失败。相反，我们可以使用被广泛认为是安全的知名算法。移动操作系统提供标准的加密 API 来实现这些算法。

仔细检查源代码内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。所有的加密操作都应该使用 Android 和 iOS 的标准加密 API（我们会在特定平台的章节中更详细地写到这些）。任何不调用已知供应商的标准例程的加密操作都应该被仔细检查。密切关注被修改过的标准算法。记住，编码并不等同于加密! 当你发现像 XOR（exclusive OR）这样的位操作符时，一定要进一步调查。

在加密的所有实现中，你需要确保总是发生以下情况：

- 工作密钥（如 AES/DES/Rijndael 中的中间/衍生密钥）在使用后或出错时，从内存中正确删除。
- 应尽快从内存中移除密码的内部状态。

4.6.3.6. AES 配置不充分

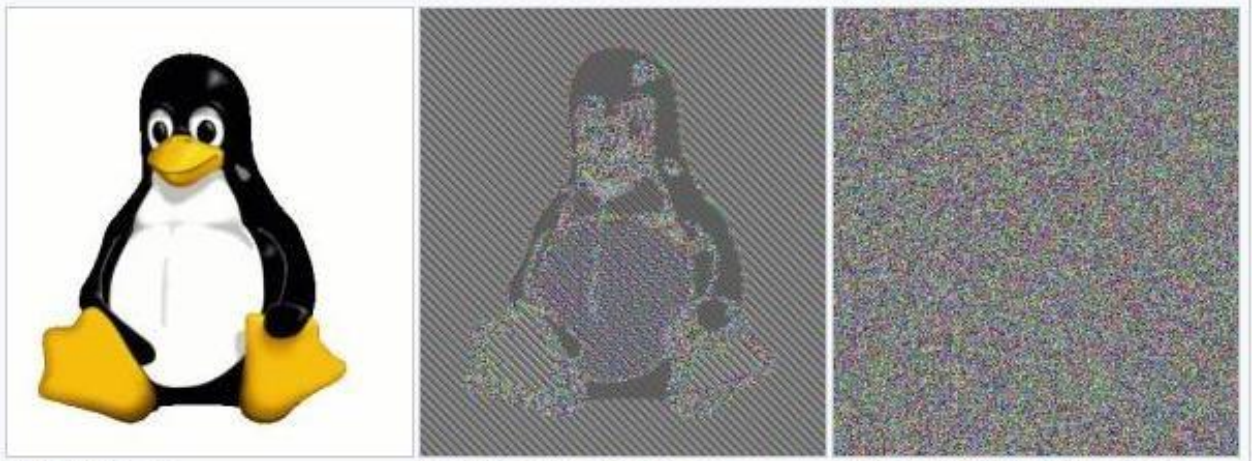
高级加密标准（AES）是移动应用中广泛接受的对称加密标准。它是一种迭代分组密码，基于一系列相互联系的数学运算。AES 对输入执行可变的轮数，每个轮数都涉及对输入分组中字节的替换和排列。每轮使用从原始 AES 密钥派生的 128 位轮密钥。

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，实施细节和可配置参数，如块密码模式，留下了一些错误的余地。

4.6.3.6.1. 弱分组密码模式

基于分组的加密在离散输入分组上执行（例如：AES 具有 128 位分组）。如果明文大于分组大小，则明文在内部拆分为给定输入大小的分组，并对每个分组执行加密。分组密码操作模式（或分组模式）确定加密前一分组的结果是否影响后续分组。

ECB（电子密码本）将输入分成固定大小的分组，这些块使用相同的密钥分别加密。如果多个分割的分组包含相同的明文，它们将被加密为相同的密文分组，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。



Original image | Encrypted using ECB mode | Modes other than ECB result in pseudo-randomness

验证是否使用了密码分组链（CBC）模式而不是 ECB 模式。在 CBC 模式下，明文分组与前一个密码分组进行 XOR。这确保了每个加密分组是唯一的，并且是随机的，即使分组包含相同的信息。请注意，最好将 CBC 与 HMAC 结合起来，并/或确保不出现 "填充错误"、"MAC 错误"、"解密失败 "等错误，以便更好地抵抗填充提示攻击。

在存储加密数据时，我们建议使用同时保护存储数据完整性的分组模式，如 Galois/计数器模式（GCM）。后者有一个额外的好处，即该算法是每个 TLSv1.2 实现的强制性算法，因此可以在所有现代平台上使用。

关于有效分组模式的更多信息，请参阅 NIST 的分组模式选择指南。

4.6.3.6.2. 可预测初始化向量

CBC、OFB、CFB、PCBC、GCM 模式需要一个初始化向量（IV）作为密码的初始输入。IV 不必保密，但它不应该是可预测的：它应该是随机的，对每个加密信息来说是唯一的/不可重复的。确保 IV 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。

注意代码中使用的加密库：许多开源库在其文档中提供的例子可能会遵循不好的做法（例如，使用硬编码的 IV）。一个流行的错误是复制粘贴示例代码而不改变 IV 值。

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，IV 的使用是不同的，其中初始化向量通常是一个计数器（在 CTR 中与一个随机量结合）。所以在这里，使用一个可预测的 IV 和它自己的有状态的模型正是我们需要的。在 CTR 中，你有一个新的 nonce 加计数器作为每个新分组操作的输入。例如：对于一个 5120 比特长的明文，你有 20 个分组，所以你需要 20 个由随机量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，不应该用同一个密钥重复。关于 IV 的更多细节和建议，见 NIST 关于 GCM 的文件第 8 节。

4.6.3.7. 由于较弱的填充或分组操作实现而导致的填充提示攻击

在过去，PKCS1.5 填充（代码为：PKCS1Padding）被用作做非对称加密时的填充机制。这种机制很容易受到填充提示的攻击。因此，最好是使用 PKCS#1 v2.0 中捕获的 OAEP（最佳非对称加密填充）（代码为：`OAEPPadding`, `OAEPwithSHA-256andMGF1Padding`, `OAEPwithSHA-224andMGF1Padding`, `OAEPwithSHA-384andMGF1Padding`, `OAEPwithSHA-`

512andMGF1Padding)。请注意，即使使用 OAEP，你仍然可能遇到一个问题，即 Kudelskisecurity 的博客中所描述的管理者攻击。

注意：使用 PKCS #5 的 AES-CBC 也被证明容易受到填充提示攻击，因为该实现会发出警告，如 "填充错误"、"MAC 错误 "或 "解密失败"。见《填充提示攻击》和《CBC 填充提示问题》示例。接下来，最好确保在加密明文后添加 HMAC：毕竟 MAC 失败的密文将不必被解密，可以被丢弃。

4.6.3.8. 保护存储中和内存中的密钥

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 root 权限(例如：一个已经 root 的设备或越狱设备)或它需要一个使用 Frida 打过补丁的应用程序(以便您可以使用工具，如 Fridump)。因此，如果设备上仍然需要密钥，最好考虑以下几点：

- **远程服务器中的密钥**：你可以使用远程密钥库，如亚马逊 KMS 或 Azure 密钥库。对于某些用例，在应用程序和远程资源之间开发一个协调层可能是一个合适的选择。例如，在 "功能即服务"（FaaS）系统（如 AWS Lambda 或 Google Cloud Functions）上运行的无服务器功能，它可以转发请求以检索 API 密钥或机密。还有其他的选择，如亚马逊 Cognito、Google 身份平台或 Azure Active Directory。
- **安全硬件支持的存储内的密钥**：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 Keystore）或安全隔区（例如使用 Keychain）。更多信息请参考 Android 数据存储和 iOS 数据存储章节。
- **受信封加密保护的密钥**：如果密钥存储在 TEE / SE 之外，请考虑使用多层加密：信封加密方法（见 OWASP 加密存储备忘录、Google 云密钥管理指南、AWS 优秀架构框架指南），或 HPKE 方法，用密钥加密密钥来加密数据加密密钥。
- **内存中的密钥**：确保密钥在内存中存活的时间尽可能短，并考虑在成功的加密操作后，以及在出现错误的情况下，将密钥清零和作废。关于一般的加密准则，请参考《清除机密数据的内存》。更详细的信息，请分别参考 Android 和 iOS 的测试敏感数据的内存和测试敏感数据的内存部分

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，使用混淆方法共享密钥，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。例如，对称密钥可以用非对称密钥对中的公共密钥进行加密。

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 上的数据存储和 iOS 上的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“Android 网络 API”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。

4.6.6. 加密算法规定

当你将应用程序上传到 App Store 或 Google Play 时，你的应用程序通常存储在美国服务器上。如果你的应用程序包含加密算法，并被分发到任何其他国家，它被认为是加密算法出口。这意味着你需要遵循美国的加密算法出口法规。此外，一些国家对加密算法也有进口规定。

了解更多：

- 遵守加密算法出口法规 (Apple)
- 出口法规概述(Apple)
- 遵守出口法规(Google)
- 加密算法和出口管理条例(美国)
- 加密算法管制(法国)
- 世界范围的加密算法法律和政策

4.6.7. 参考文献

4.6.7.1. 参考文献

- 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。”

4.6.7.2. 加密算法

- Argon2
- AWS 优秀架构框架指南
- 用管理者攻击破解 RSA
- Google 云密钥管理指南
- 混合公钥加密
- NIST 800-38d
- NIST 800-57Rev5
- NIST 800-63b
- NIST 800-132
- OWASP 加密存储备忘录
- 密码散列竞赛(PHC)
- PKCS #1: RSA 加密版本 1.5
- PKCS #1: RSA 加密规范版本 2.0
- PKCS #7: 加密信息语法版本 1.5
- 填充提示攻击
- CBC 填充提示问题
- Veorq 的加密编码指南

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，常见的漏洞，如 SQL 注入、缓冲区溢出和跨站脚本（XSS），在忽视安全编程实践时，可能会在应用程序中表现出来。

同样的编程缺陷可能在某种程度上影响 Android 和 iOS 应用程序，因此我们将在指南的通用部分提供最常见的漏洞类别概述。在后面的章节中，我们将介绍操作系统的具体实例和漏洞缓解功能。

4.7.1. 注入缺陷（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对于程序员来说是必要的最佳实践。

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 应用程序通过将用户凭据存储在本地数据库中来处理本地用户身份认证（在本例中，我们将忽略这一糟糕的编程实践）。登录后，应用程序将查询数据库以搜索具有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 pa
ssword = '" + 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 通过 adb 利用这个 SQL 注入载荷攻击了 Yahoo 的 weather mobile 应用程序的 sort 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。这些应用程序的导出内容提供程序易受 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 nettity 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 解析器处理内容过程进行攻击。

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，会返回无限字节流，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，通常是恶意应用程序先攻击设备上运行的另一个应用程序。定位潜在漏洞首先要执行以下操作之一

- 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），然后检查未经检查的输入是否与相应的查询成功交互。

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 调用。
- 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。
- 剪贴板。
- 用户界面。

验证是否遵循了以下最佳实践：

- 对不可信任的输入进行类型检查和/或使用可接受值的列表进行验证。
- 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。如果定义了准备语句，用户提供的数据和 SQL 代码会自动分开。
- 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。

我们将在相关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，从而进行多种攻击（例如：窃取会话 cookie、记录按键、执行任意操作等）。

在原生应用的环境中，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，从而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 呈现内容，并调查应用程序渲染的不受信任输入。

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 在以下的 HackerOne 报告中使用了这一点。对 HTML 参数的任何输入都将在 Quora 的 ActionBarContentActivity 中受信任。载荷可以使用 adb、来自 ModalContentActivity 的剪贴板数据以及来自第三方应用程序。

- ADB

```
$ adb shell
$ am start -n com.quora.android/com.quora.android.ActionBarContentActiv
ity \
-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 中的第三方内容:

```
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)
```

如果使用 WebView 显示远程网站，则转义 HTML 的任务将由服务器端执行。如果 web 服务器上存在 XSS 漏洞，则可以使用该漏洞在 WebView 的内容中执行脚本。因此，对 web 应用程序源代码执行静态分析非常重要。

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 或其他解析内容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。

考虑数据会如何在响应中呈现。例如：如果在 HTML 内容中呈现数据，则必须转义六个控制字符：

字符	转义
&	&amp;
<	<
>	>
"	"
'	'
/	/

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、自动模糊测试来检测，例如 HTML 标记和特殊字符注入所有可用的输入字段，以验证 web 应用程序是否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，自动模糊输入被认为是一种有效的方法。例如：BURP Scanner 在识别反射 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经过了手动参数检查测试。

4.7.3. 内存损坏错误（MSTG-CODE-8）

内存损坏错误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，但由于一些因素，近年来已不再那么普遍。值得注意的是，让开发人员意识到使用不安全的 C 库函数的风险已经是现在一种常见的最佳实践，此外，捕获缓冲区溢出错误相对简单。然而，这些缺陷仍然值得测试。
- **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（收回）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实现对指令指针的控制。
- **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，这将导致值变为“环绕”最大整数值，从而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，则会发生整数下溢，使得结果大于预期值。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 系列的格式化字符串参数时，攻击者可能会注入格式化令牌，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放置称为 shellcode 的汇编机器指令的位置。在 iOS 上，数据执行保护功能（顾名思义）防止从定义为数据段的内存执行指令。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，这些小工具可以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 shellcode 所在位置的内存保护设置。

Android 应用程序基本上都是用 Java 实现的， Java 从设计上就不会出现内存损坏问题。然而，使用 JNI 库的原生应用程序很容易受到这种错误的影响。类似地，iOS 应用程序可以将 C/C++ 调用包装在 Obj-C 或 Swift 中，使它们容易受到此类攻击。

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输入到格式化字符串中。

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，释放后使用问题实际上可能是一个复杂的、违反直觉的竞争条件的结果，而不是立即显现出来。从被忽视的代码缺陷的深层实例中表现出来的问题通常是通过动态分析或由投入时间来深入理解程序的测试人员发现的。

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，会不断发送格式错误的数据到应用程序里，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”）。

有关模糊测试的更多信息，请参阅《OWASP 模糊测试指南》。

4.7.4. 二进制保护机制

4.7.4.1. 地址无关代码

PIC（Position Independent Code）是指被放置在主内存某处的代码，无论其绝对地址如何，都能正常执行。PIC 通常用于共享库，这样就可以在每个程序地址空间中的某个位置加载相同的库代码，在那里它不会与其他正在使用的内存（例如，其他共享库）重叠。

PIE（Position Independent Executable）是完全由 PIC 生成的可执行二进制文件。PIE 二进制文件用于启用 ASLR（地址空间布局随机化），它随机地安排进程的关键数据区域的地址空间位置，包括可执行文件的基数和堆栈、堆和库的位置。

4.7.4.2. 内存管理

4.7.4.2.1. 自动引用计数

ARC（Automatic Reference Counting）是 Objective-C 和 Swift 独有的 Clang 编译器的内存管理功能。当类实例不再需要时，ARC 会自动释放这些实例所使用的内存。ARC 与跟踪式垃圾收集的不同之处在于，没有后台进程在运行时异步地取消分配对象。

与跟踪式垃圾收集不同，ARC 不会自动处理引用循环。这意味着，只要存在对一个对象的 "强 " 引用，它就不会被取消分配。强交叉引用会相应地造成死锁和内存泄漏。这就需要开发者通过使用弱引用来打破循环。你可以在这里了解更多关于它与垃圾收集的区别。

4.7.4.2.2. 垃圾回收

垃圾收集（GC）是一些语言的自动内存管理功能，如 Java/Kotlin/Dart。垃圾收集器试图回收由程序分配的、但不再被引用的内存，也称为垃圾。Android 运行时（ART）使用了改进版的 GC。你可以在这里了解更多关于它与 ARC 的区别。

4.7.4.2.3. 手动内存管理

在用 C/C++编写的原生库中，通常需要手动内存管理，ARC 和 GC 并不适用。开发者有责任进行适当的内存管理。众所周知，如果使用不当，手动内存管理会导致程序中出现几类主要的错误，特别是违反内存安全或内存泄漏。

更多的信息可以在 "内存损坏错误（MSTG-CODE-8）"中找到。

4.7.4.2.4. 栈溢出保护

Stack canary 通过在返回指针之前的堆栈上存储一个隐藏的整数值来帮助防止堆栈溢出攻击。这个值在函数的返回语句执行之前被验证。

函数的返回语句之前进行验证。一个缓冲区溢出攻击通常会覆盖一个内存区域，以便覆盖返回指针并接管程序流。如果启用了 stack canary，它们也会被覆盖，CPU 就会知道内存被篡改了。

堆栈缓冲区溢出是一种更普遍的编程漏洞，被称为缓冲区溢出（或缓冲区超限）。与堆上的缓冲区溢出相比，堆栈上的缓冲区溢出更有可能破坏程序的执行，因为堆上有所有活动函数调用的返回地址。

4.7.5. 参考文献

4.7.5.1. 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 Intent、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。”

4.7.5.2. 通过 ContentActivity 实现 XSS

- https://hackerone.com/reports/189793

4.8. 移动应用用户隐私保护

重要免责声明：MASTG 不是法律手册。因此，我们不会深入探讨 GDPR 或其他可能的相关立法。本章旨在向您介绍主题，并为您提供必要的参考资料，供您自己继续研究。我们还将尽最大努力为您提供测试 OWASP MASV 中列出的隐私相关要求的测试或指南。

4.8.1. 概览

4.8.1.1. 主要问题

移动应用程序处理各种敏感的用户数据，从身份和银行信息到健康数据。人们对这些数据的处理方式和最终去向的担忧是可以理解的。我们还可以谈论“用户从使用应用程序中获得的好处”与“用户付出的实际代价”（通常而且不幸的是，他们甚至没有意识到这一点）。

4.8.1.2. 解决方法（2020 年之前）

为确保用户得到适当保护，已经在欧洲制定并实施了的《通用数据保护条例》（GDPR）（自 2018 年 5 月 25 日起适用），迫使开发商在处理敏感用户数据方面更加透明。其主要是通过使用隐私政策实现。

4.8.1.3. 挑战

这里需要考虑两个主要方面：

- **开发者合规**：开发者需要遵守法律隐私原则，因为这些原则是由法律强制执行的。开发人员需要更好地理解法律原则，以便知道他们需要实现什么才能保持法规遵从性。理想情况下，至少必须满足以下要求：
  - **实现源于设计的隐私保护**（GDPR 第 25 条，设计和默认的数据保护）
  - **最小权限原则**（“系统的每个程序和每个用户都应使用完成作业所需的最小权限集进行操作。”）
- **用户教育**：需要教育用户了解其敏感数据，并告知用户如何正确使用应用程序（以确保安全保存和处理其信息）。

注意：应用程序通常会声称处理某些数据，但事实并非如此。马吉德·哈塔米安（Majid Hatamian）的 IEEE 文章《智能手机应用程序中的隐私工程：应用程序开发人员的技术指南目录》对此主题进行了非常好的介绍。

4.8.1.3. 数据保护的保护目标

当应用程序的业务流程需要用户提供个人信息时，需要告知用户其数据发生了什么以及应用程序需要这些信息的原因。如果有第三方会实际处理数据，应用程序也应该通知用户。

当然，您已经熟悉了安全保护目标的经典三要素：机密性、完整性和可用性。但是，您可能不知道数据保护中有三个建议重点关注保护目标：

- 无关联性：
  - 用户的隐私相关数据必须与域外的任何其他隐私相关数据集无关联。
  - 包括：数据最小化、匿名化、假名化等。
- 透明度：
  - 用户应该能够请求应用程序提供的所有信息，并收到如何请求这些信息的方法。
  - 包括：隐私政策、用户教育、适当的日志记录和审核机制等。
- 可干预性：
  - 用户应该能够更正其个人信息，请求删除，随时撤回任何给定的同意，并收到如何这样做的方法。
  - 包括：直接在应用程序中设置隐私，个人干预请求的单一联系点（如应用程序内聊天、电话号码、电子邮件）等。

  更多详细说明，请参见 ENISA“移动应用程序中的隐私和数据保护”中的第 5.1.1 节“数据保护目标简介”。

同时解决安全和隐私保护目标是一项非常具有挑战性的任务（如果在许多情况下不是不可能的话）。IEEE 出版物隐私工程的保护目标中有一个有趣的可视化，称为“三轴”，表示不可能同时确保六个目标中的每一个都达到 100%。

传统上，隐私政策涵盖了源自保护目标的大部分流程。然而，这种方法并不总是最佳的：

- 开发人员不是法律专家，但仍需要遵守法律。
- 用户需要阅读通常冗长的政策。

4.8.1.4. 新方法（Google 和 Apple 对此的做法）

为了应对这些挑战并帮助用户轻松了解他们的数据是如何收集、处理和共享的，Google 和 Apple 推出了新的隐私标签系统（非常符合 NIST 关于消费者软件网络安全标签的建议）：

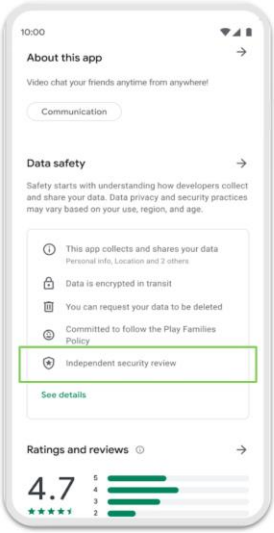
- App Store 隐私营养标签（自 2020 年起）。
- Google Play 数据安全部分（自 2021 起）。

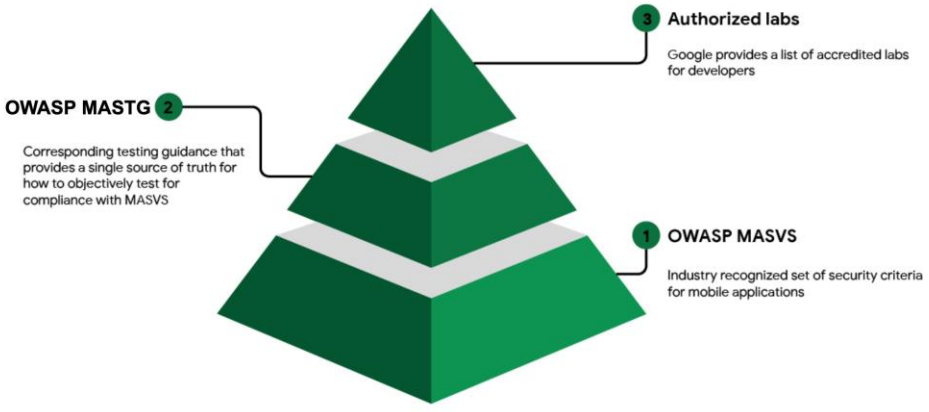
作为这两个平台上的新要求，这些标签的准确性对于提供用户保证和减少滥用至关重要。

4.8.1.5. Google ADA MASA 计划

定期进行安全测试可以帮助开发者识别其应用程序中的关键漏洞。Google Play 将允许已完成独立安全验证的开发者在其数据安全部分展示这一情况。这有助于用户对应用程序的安全性和隐私性感到更加确信。

为了使应用程序的安全架构更加透明，Google 推出了 MASA（移动应用安全评估）计划，作为应用防御联盟（ADA）的一部分。通过 MASA，Google 已经认识到利用全球公认的移动应用安全标准对移动应用生态系统的重要性。开发者可以直接与授权实验室的合作伙伴合作，启动安全评估。Google 将认可那些根据 MASVS 一级要求对其应用程序进行独立验证的开发者，并将在其数据安全部分展示这一点。





如果你是一个开发商，并希望参加，你应该填写这个表格

请注意，测试的有限性并不能保证应用程序的完全安全。这种独立审查的范围可能不包括验证开发者的数据安全声明的准确性和完整性。开发者对在其应用程序的 Play 商店列表中作出完整和准确的声明负有完全责任。

4.8.1.6. 与测试其他 MASV 类别有何关系

以下是您作为安全测试人员应该报告的常见侵犯隐私列表（虽然不是详尽的列表）：

- 示例 1：应用程序访问用户已安装应用程序清单，并通过网络（违反 MSTG-STORAGE-4）或通过 IPC 机制（违反 MSTG-STORAGE-6）发送到另一个应用程序，不将此数据视为个人或敏感数据。
- 示例 2：应用程序在未经用户授权，例如通过生物特征识别的情况下就显示敏感数据，如信用卡详细信息或用户密码，（违反 MSTG-AUTH-10）。
- 示例 3：一个访问用户电话或通讯录数据的应用程序，不将此数据视为个人或敏感数据，另外还通过不安全的网络连接发送数据（违反 MSTG-network-1）。
- 示例 4：一个应用程序收集设备位置（显然不是其正常运行所必需的），并且没有明确披露哪个功能使用此数据（违反了 MSTG-PLATFORM-1）。

  您可以在 Google Play Console 帮助中找到更多常见的违规行为（策略中心->隐私、欺骗和设备滥用->用户数据）。

正如您所看到的，这与其他测试类别有着密切的关系。当您测试它们时，您通常会间接测试用户隐私保护。请记住这一点，因为它将帮助您提供更好、更全面的报告。通常，您还可以重用来自其他测试的证据，以测试用户隐私保护（请参阅“测试用户教育”中的示例）。

4.8.1.7. 了解更多

您可以在此处了解有关此主题和其他隐私相关主题的更多信息：

- iOS 应用程序隐私政策
- App Store 上的 iOS 隐私详细信息部分
- iOS 隐私最佳实践
- Android 应用程序隐私政策
- Google Play 上的 Android 数据安全部分
- 为 Google Play 中新的数据安全部分准备你的应用程序

- Android 隐私最佳实践

4.8.2. 测试用户教育（MSTG-STORAGE-12）

4.8.2.1. 在应用程序市场上测试用户数据隐私教育

此时，我们只想知道开发人员正在披露哪些与隐私相关的信息，并尝试评估这些信息是否合理（类似于您在测试权限时所做的）。

> 开发人员可能没有声明确实正在收集和或共享的某些信息，但这是扩展此测试的另一个主题。作为本测试的一部分，您不应该提供隐私侵犯保证。

4.8.2.2. 静态分析

你可以执行以下步骤：

1. 在相应的应用程序市场（如 Google Play、App Store）中搜索应用程序。
2. 进入“隐私详情”（App Store）或“安全部分”（Google Play）。
3. 核实是否有任何可用信息。

如果开发人员遵守了应用程序市场指南并包含了所需的标签和说明，则测试通过。存储并提供您从应用程序市场获得的信息作为证据，以便您以后可以使用它来评估潜在的侵犯隐私或数据保护行为。

4.8.2.3. 动态分析

作为可选步骤，您还可以提供某种证据作为此测试的一部分。例如，如果您正在测试 iOS 应用程序，您可以简单的启用应用程序活动录制并导出隐私报告，其中包含应用程序对不同资源（如照片、联系人、相机、麦克风、网络连接等）访问的详细情况。

这样做实际上对测试其他 MASV 类别有很多好处。它提供了非常有用的信息，可用于在 MASVS- NETWORK 中测试网络通信或在 MASVS-PLATFORM 中测试应用程序权限。在测试这些其他类别时，您可能已经使用其他测试工具进行了类似的测量。您也可以将其作为此测试的证据。

理想情况下，应将可用信息与应用程序的实际用途进行比较。然而，根据您的资源和自动化工具的支持，这远远不止是一项可能需要几天到几周才能完成的琐碎任务。它还严重依赖于应用程序的功能和内容，最好能与应用程序开发人员密切合作进行白盒设置。

4.8.2.4. 测试用户教育的最佳安全实践

如果您打算将测试自动化，那么测试这一点可能尤其具有挑战性。我们建议充分使用该应用程序，并尽可能回答以下问题：

- **指纹使用**：当应用程序使用指纹进行身份认证以提供对高风险交易/信息的访问时，当其他人的多个指纹也注册到设备上时，应用程序是否会告知用户潜在的问题？
- **Rooting /越狱**：当应用程序检测到 Root 过或越狱设备时，应用程序是否告知用户，由于设备的越狱/Root 状态，会为某些高风险操作带来额外风险？
- **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，是否指示用户永远不要与任何其他人共享此信息，并且只有该应用程序才会请求它。
- **应用程序分发**：对于高风险应用程序，为了防止用户下载该应用程序的恶意修改版本，应用程序制造商是否正确告知了发布应用程序的官方方式（例如从 Google Play 或 App Store）？
- **显著披露**：任何情况下，应用程序是否显示数据访问、收集、使用和共享的显著披露？例如：应用程序是否使用应用程序跟踪透明度框架来请求 iOS 上的权限？

4.8.3.参考文献

- 开源代码许可与 Android - https://www.bignerdranch.com/blog/open-source-licenses-and-android/
- 软件许可协议通俗介绍 - https://tldrlegal.com/
- Apple 人机界面指引 - https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/
- Android 应用程序权限最佳实践 - https://developer.android.com/training/permissions/requesting.html#explain

4.8.3.1. 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。”

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：

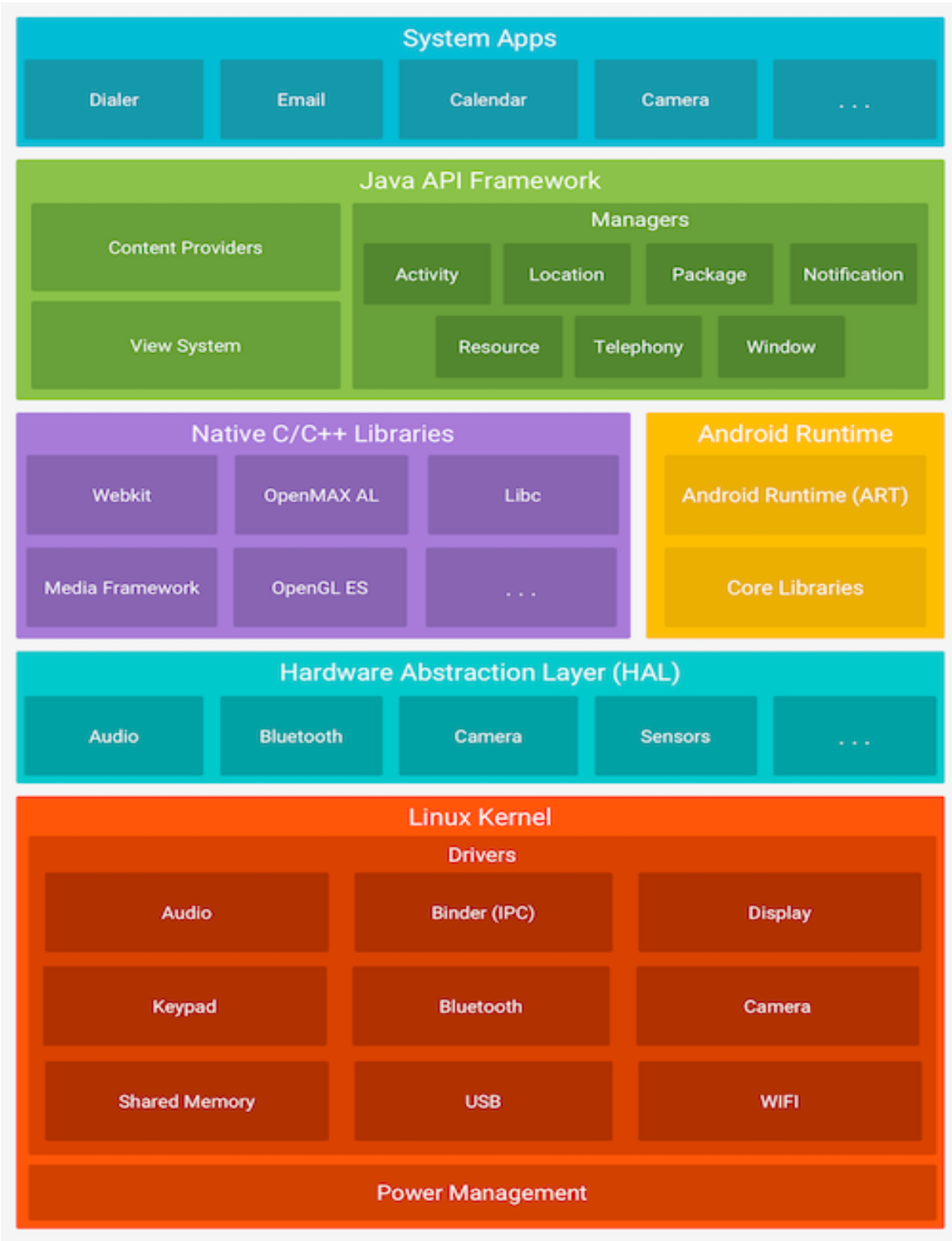
1. Android 架构。
2. Android 安全：纵深防御方法。
3. Android 应用架构。
4. Android 应用发布。
5. Android 应用攻击面。

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 架构

Android 是由 Google 开发的基于 Linux 的开源平台，由开放手机联盟（由 Google 领导的联盟）开发，作为一个移动操作系统（OS）。如今，该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 Google Play 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。

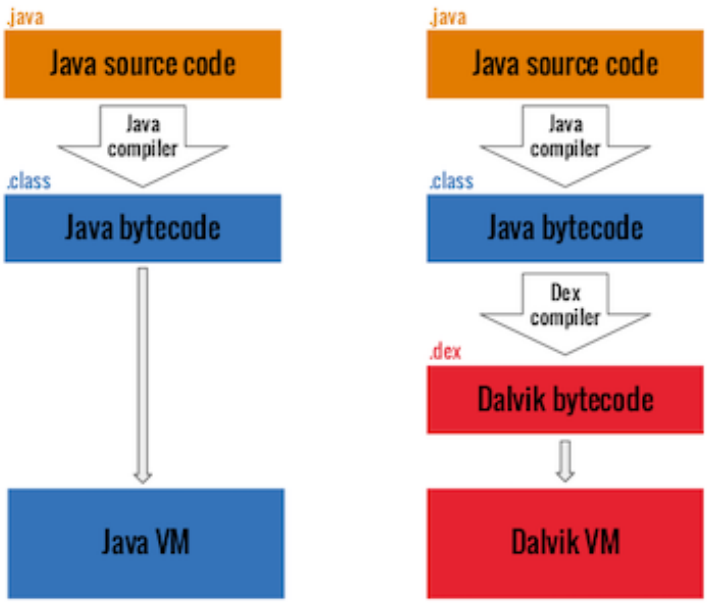

System Apps
Dialer
Email
Calendar
Camera
. . .
Java API Framework
Content Providers
View System
Managers
Activity
Location
Package
Notification
Resource
Telephony
Window
Native C/C++ Libraries
Webkit
OpenMAX AL
Libc
Media Framework
OpenGL ES
. . .
Android Runtime
Android Runtime (ART)
Core Libraries
Hardware Abstraction Layer (HAL)
Audio
Bluetooth
Camera
Sensors
. . .
Linux Kernel
Drivers
Audio
Binder (IPC)
Display
Keypad
Bluetooth
Camera
Shared Memory
USB
WIFI
Power Management

内核：在最底层，Android 是基于 Linux 内核的一个变种，包含一些重要的附加功能，包括低内存终结、唤醒锁、Binder IPC 驱动等。对于 MASTG 的目的，我们将专注于操作系统的用户模式部分，在这里，Android 与典型的 Linux 发行版有明显的区别。对我们来说，最重要的两个组件是应用程序使用的管理运行时（ART/Dalvik）和 Bionic，即 Android 版本的 glibc，GNU C 库。

HAL：在内核的顶层，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。一些 HAL 的实现被打包成共享库模块，Android 系统在需要时调用这些模块。这是允许应用程序与设备的硬件互动的基础。例如，它允许自带的电话应用使用设备的麦克风和扬声器。

运行时环境：Android 应用程序是用 Java 和 Kotlin 编写的，然后编译成 Dalvik 字节码，然后可以使用一个运行时来执行，该运行时解释字节码指令并在目标设备上执行它们。对于 Android 来说，这就是 Android 运行时（ART）。这类似于 Java 应用程序的 JVM（Java 虚拟机），或.NET 应用程序的 Mono 运行时。

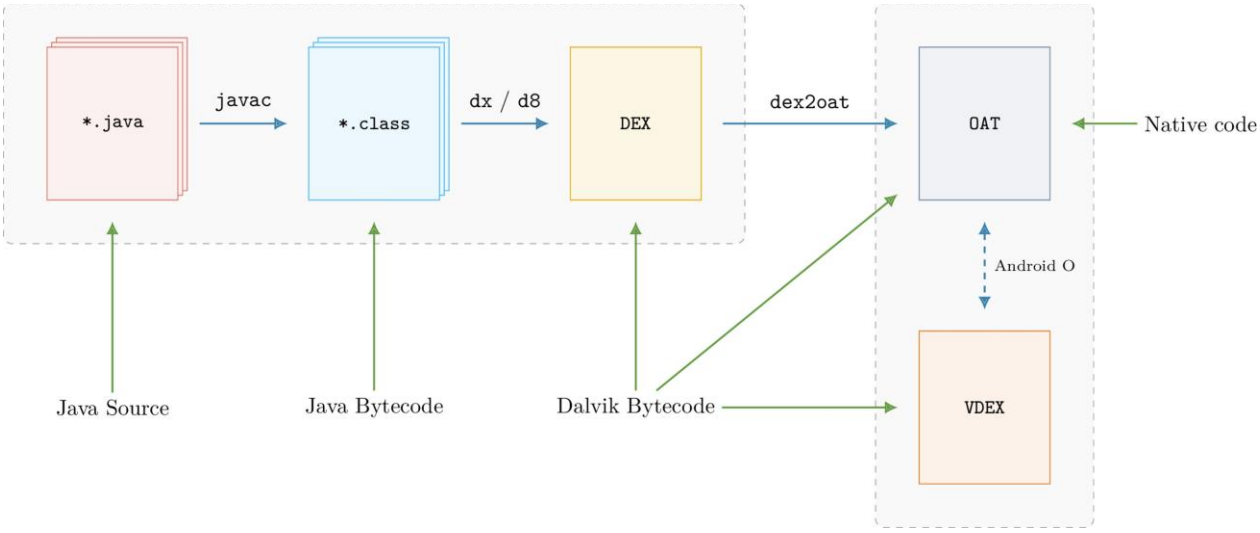
Dalvik 字节码是 Java 字节码的一个优化版本。它是通过分别使用 javac 和 kotlinc 编译器将 Java 或 Kotlin 代码编译成 Java 字节码，从而产生.class 文件。最后，使用 d8 工具将 Java 字节码转换为 Dalvik 字节码。Dalvik 字节码以.dex 文件的形式打包在 APK 和 AAB 文件中，并由 Android 上的管理运行时在设备上执行。



在 Android 5.0（API 级别 21）之前，Android 在 Dalvik 虚拟机（DVM）上执行字节码，它在执行时被翻译成机器代码，这个过程被称为及时编译（JIT）。这使得运行时能够受益于编译代码的速度，同时保持解释代码的灵活性。

自 Android 5.0（API 级别 21）以来，Android 在 Android 运行时（ART）上执行字节码，这是 DVM 的继承者。ART 通过包括 Java 和原生堆栈信息，在应用程序原生崩溃报告中提供改进的性能和上下文信息。它使用相同的 Dalvik 字节码输入以保持向后兼容。然而，ART 以不同的方式执行 Dalvik 字节码，使用预编译（AOT）、实时（JIT）和反馈式编译的混合组合。

- **AOT** 将 Dalvik 字节码预编译成原生代码，生成的代码将以.oat 扩展名（ELF 二进制）保存在磁盘上。dex2oat 工具可以用来执行编译，在 Android 设备上可以在 /system/bin/dex2oat 找到。AOT 编译是在应用程序的安装过程中执行的。这使得应用程序的启动速度更快，因为不再需要编译了。然而，这也意味着，与 JIT 编译相比，安装时间会增加。此外，由于应用程序总是针对当前版本的操作系统进行优化，这意味着软件更新将重新编译所有先前编译的应用程序，导致系统更新时间大幅增加。最后，AOT 编译将编译整个应用程序，即使某些部分从未被用户使用。
- **JIT** 是在运行时进行的。
- **反馈式编译**是一种混合方法，在 Android 7（API 级别 24）中被引入，以消除 AOT 的缺点。一开始，应用程序会使用 JIT 编译，而 Android 会跟踪应用程序中所有经常使用的部分。这些信息被储存在一个应用程序的配置文件中，当设备处于空闲状态时，一个编译（dex2oat）守护程序就会运行，AOT 会从配置文件中识别编译频繁使用代码路径。



来源：https://lief-project.github.io/doc/latest/tutorials/10androidformats.html

沙盒化：Android 应用不能直接访问硬件资源，每个应用都在自己的虚拟机或沙盒中运行。这使得操作系统能够精确控制设备上的资源和内存访问。例如，一个崩溃的应用程序不会影响在同一设备上运行的其他应用程序。Android 系统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太多的资源。同时，这种沙盒设计可以被认为是 Android 全局纵深防御战略中的众多原则之一。具有低权限的恶意第三方应用程序不应该能够逃逸自己的运行时并读取同一设备上的受害者应用程序的内存。在下面的章节中，我们将仔细研究 Android 操作系统中的不同防御层。在 "软件隔离 "一节中了解更多。

你可以在 Google 原创文章 "Android 运行时(ART) "中找到更详细的信息。 乔纳森-莱文的 "Android 内部 "和@_qaz_-qaz 的 "Android 101 "博文(https://secrary.com/android-reversing/android101/)。

5.1.2. Android 安全：纵深防御方法

Android 体系结构实现了不同的安全层，这些安全层共同支持纵深防御方法。这意味着敏感用户数据或应用程序的机密性、完整性或可用性并不取决于一个单一的安全措施。本节概述了 Android 系统提供的不同防御层。安全策略大致可分为四个不同的领域，每个领域都侧重于防范特定的攻击模型。

- 全系统安全
- 软件隔离
- 网络安全
- 反漏洞攻击

5.1.2.1. 全系统安全

5.1.2.1.1 设备加密

Android 从 Android 2.3.4（API 级别 10）开始支持设备加密，从那时起它经历了一些重大的变化。Google 规定，所有运行 Android 6.0（API 级别 23）或更高版本的设备都必须支持存储加密。不过一些低端设备被豁免，因为它会显著影响性能。

- 全磁盘加密（FDE）：Android 5.0（API 级别 21）及以上版本支持全磁盘加密。此加密使用受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。这种加密现在被视为不推荐使用，应尽可能使用基于文件的加密。全磁盘加密有一些缺点，例如，在重新启动后如果用户没有输入解锁密码，则无法接收呼叫或操作警告。
- 基于文件的加密（FBE）：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，以便可以独立解密。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，即使用户没有解锁设备。

注意：您可能听说过 Adiantum，这是一种为运行 Android 9（API 级别 28）及更高版本的并且其 CPU 缺少 AES 指令的设备设计的加密方法。**Adiantum 仅与 ROM 开发人员或设备供应商相关**，Android 不提供 API 供开发人员从应用程序中使用 Adiantum。根据 Google 的建议，在带有 ARMv8 加密扩展的基于 ARM 的设备或带有 AES-NI 的基于 x86 的设备时，不应使用 Adiantum。AES 在这些平台上速度更快。

更多有关信息，请参阅 Android 文档。

5.1.2.1.2 可信执行环境（TEE）

为了让 Android 系统执行加密，需要一种安全生成、导入和存储加密密钥的方法。从本质上讲，我们正在将保持敏感数据安全的问题转变为确保加密密钥的安全。如果攻击者可以转储或猜测加密密钥，则可以检索敏感的加密数据。

Android 在专用硬件中提供了一个可信的执行环境，以解决安全生成和保护加密密钥的问题。这意味着 Android 系统中的专用硬件组件负责处理加密密钥材料。分别由三个主要模块负责：

- 硬件支持的密钥库(KeyStore)：该模块为 Android 操作系统和第三方应用程序提供加密服务。它使应用程序能够在 TEE 中执行加密敏感操作，而无需公开加密密钥材料。
- StrongBox：在 Android 9（Pie）中，引入了 StrongBox，这是实现硬件支持的密钥库的另一种方法。在 Android 9 Pie 之前，硬件支持的密钥库由任何位于 Android OS 内核之外的 TEE 实现。StrongBox 是一个实际完整的独立硬件芯片，添加到实现了密钥库的设备上，并在 Android 文档中进行了明确定义。您可以通过编程方式检查密钥是否驻留在 StrongBox 中，如果它驻留在 StrongBox 中，您可以确保它受到硬件安全模块的保护，该

模块具有自己的 CPU、安全存储和真随机数生成器（TRNG）。所有敏感的加密操作都发生在这个芯片上，在 StrongBox 的安全边界内。

- GateKeeper：GateKeeper 模块用于启用设备模式和密码验证。身份认证过程中的安全敏感操作发生在设备上可用的 TEE 内部。GateKeeper 由三个主要组件组成，（1）gatekeeperd，它是用于公开 GateKeeper 的服务，（2）GateKeeper HAL，它是硬件接口，（3）TEE 实现，它是在 TEE 中实现 GateKeeper 功能的软件实现。

5.1.2.1.2 验证启动模式

我们需要有一种方法来确保在 Android 设备上执行的代码来自可信的源代码，并且其完整性不会受到损害。为了实现这一点，Android 引入了验证引导的概念。验证引导的目标是在硬件和在此硬件上执行的实际代码之间建立信任关系。在经过验证的引导序列期间，将建立一个完整的信任链，从受硬件保护的信任根（RoT）开始，直到运行的最终系统，通过并验证所有必需的引导阶段。当 Android 系统最终启动时，您可以确保系统未被篡改。您有加密证明，正在运行的代码是 OEM 预期的代码，而不是恶意或意外更改的代码。

更多有关信息，请参阅 Android 文档。

5.1.2.2. 软件隔离

5.1.2.2.1 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 */
...
```

5.1.2.2.2 SELinux

安全增强型 Linux（SELinux）使用强制访问控制（MAC）系统进一步锁定哪些进程应该访问哪些资源。每个资源都以 user:role:type:mls_level 的形式给出了一个标签，该标签定义了哪些用户能够在其上执行哪些类型的操作。例如，一个进程可能只能读取文件，而另一个进程可能可以编辑或删除文件。这样，通过使用最小权限原则，易受攻击的进程更难通过权限提升或横向移动加以利用。

更多有关信息，请参阅 Android 文档。

5.1.2.2.3 权限

Android 实现了一个广泛的权限系统，用作访问控制机制。它确保了对敏感用户数据和设备资源的受控访问。Android 将权限分类为不同类型，提供不同的保护级别。

> 在 Android 6.0（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（安装时权限）。从 API 级别 23 起，用户必须在运行时批准某些权限请求（运行时权限）。

更多信息请参考 Android 文档，其中包括一些注意事项和最佳实践

要了解如何测试应用程序权限，请参阅“Android 平台 API”章节中的测试应用程序权限部分。

5.1.2.3. 网络安全

5.1.2.3.1 默认 TLS

默认情况下，自 Android 9（API 级别 28）以来，所有网络活动都被视为在威胁环境中执行。这意味着 Android 系统将只允许应用程序通过使用传输层安全协议（TLS）建立的网络通道进行通信。该协议有效地加密所有网络流量，并创建到服务器的安全通道。可能出于遗留原因，您希望使用明文的流量连接。这可以通过调整应用程序中 res/xml/network_security_config.xml 配置文件来实现。

更多有关信息，请参阅 Android 文档。

5.1.2.3.2 DNS over TLS

自 Android 9（API 级别 28）以来，就引入了系统范围的 DNS over TLS 支持。它允许您经由 TLS 协议对 DNS 服务器执行查询。这会与 DNS 服务器建立安全通道，通过该通道发送 DNS 查询。这可以确保在 DNS 查询期间不会暴露任何敏感数据。

更多有关信息，请参阅 Android 开发者博客。

5.1.2.4. 反漏洞攻击

5.1.2.4.1 ASLR, KASLR, PIE 和 DEP

地址空间配置随机加载（ASLR）自 Android 4.1（API 级别 15）以来一直是 Android 的一部分，其是针对缓冲区溢出攻击的标准保护，它确保应用程序和操作系统都加载到随机内存地址，从而难以获得特定内存区域或库的正确地址。在 Android 8.0（API 级别 26）中，内核也实现了这种保护（KASLR）。只有当应用程序可以在内存中的随机位置加载时，ASLR 保护才可能实现，这由应用程序的位置独立可执行文件（PIE）标志指示。自 Android 5.0（API 级别 21）以来，不再支持未启用 PIE 的原生库。最后，数据执行保护（DEP）可防止代码在堆栈和堆上执行，这也可用于打击缓冲区溢出攻击。

更多有关信息，请参阅 Android 开发者博客。

5.1.2.4.2 SECCOMP 过滤器

Android 应用程序可以包含用 C 或 C++编写的原生代码。这些已编译的二进制文件既可以通过 Java 原生接口（JNI）绑定与 Android 运行时通信，也可以通过系统调用与操作系统通信。一些系统调用要么没有实现，要么不应该由普通应用程序调用。由于这些系统调用直接与内核通信，因此它们是攻击人员的主要目标。通过 Android 8（API level 26），Android 为所有基于 Zygote 的进程（例如用户应用程序）引入了对安全计算（SECCOMP）过滤器的支持。这些过滤器将可用的系统调用限制为通过 bionic 公开的系统调用。

更多有关信息，请参阅 Android 开发者博客。

5.1.3. Android 应用程序结构

5.1.3.1. 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。

API 规格随着每一个新的 Android 版本而改变。关键的错误修复和安全补丁通常也会应用于早期版本。

值得注意的 API 版本：

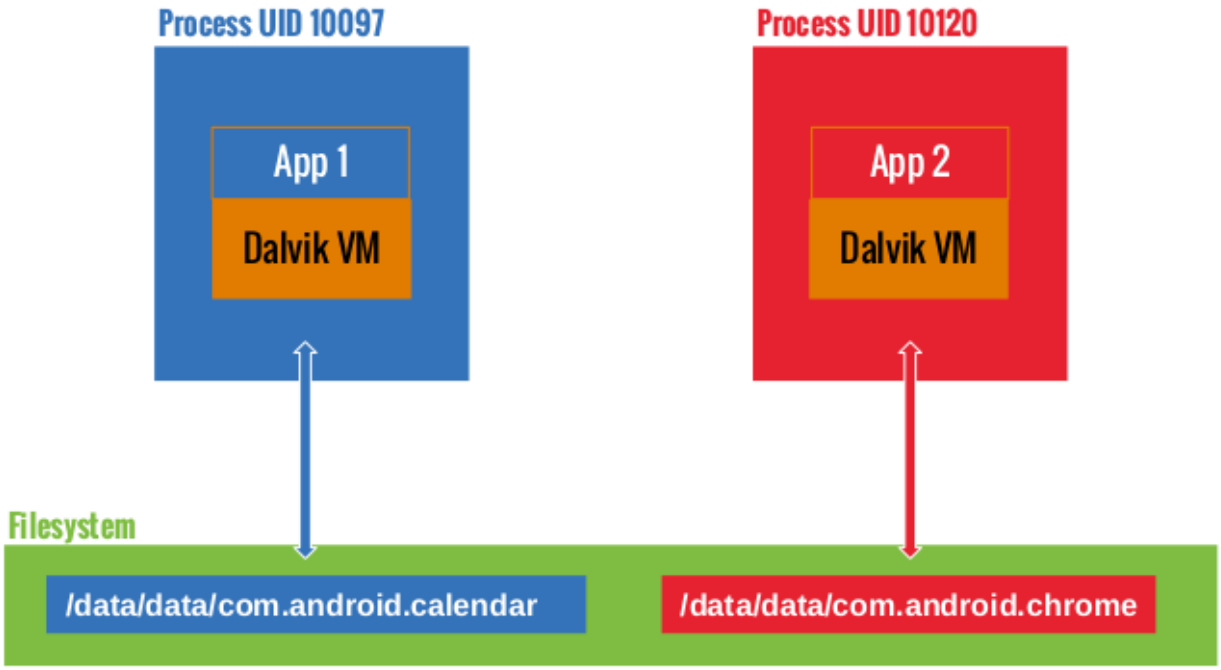
- Android 4.2（API 级别 16）于 2012 年 11 月发布（引入 SELinux）。
- Android 4.3（API 级别 18）于 2013 年 7 月发布（SELinux 默认启用）。
- Android 4.4（API 级别 19）于 2013 年 10 月发布（引入了一些新的 API 和 ART）。
- Android 5.0（API 级别 21）于 2014 年 11 月发布（默认使用 ART，并添加了许多其他功能）。
- Android 6.0（API 级别 23）于 2015 年 10 月发布（许多新功能和改进，包括授权；在运行时设置详细的权限，而不是在安装过程中设置全部或全部不设置）。
- 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（限制麦克风或摄像头的后台使用，引入锁定模式，所有应用程序的默认 HTTPS）。
- Android 10（API 级别 29）于 2019 年 9 月发布（访问位置增加"只在使用应用程序时"，防止设备跟踪，改善安全外部存储。）

- Android 11（API 级别 30）于 2020 年 9 月发布（执行分区存储，权限自动重置，减少包的可见性，APK 签名方案 v4）。
  - 隐私(概述)
  - 隐私行为变化（所有应用程序）
  - 安全行为变更（所有应用程序）
  - 隐私行为变更（针对特定版本的应用程序）
  - 安全行为变更（针对特定版本的应用程序）
- Android 12（API 级别 31-32）于 2021 年 8 月发布（Material You、Web intent 验证、隐私仪表板）。
  - 安全和隐私
  - 行为变化（所有应用程序）
  - 行为变化（针对特定版本的应用程序）
- [BETA] Android 13（API 级别 33）于 2022 年发布（更安全地导出内容注册的接收器，新的照片选择器）。
  - 安全和隐私
  - 隐私行为变化（所有应用程序）
  - 安全行为变化（所有应用程序）
  - 隐私行为变更（针对特定版本的应用程序）
  - 安全行为变更（针对特定版本的应用程序）

5.1.3.2.应用沙箱

应用程序在 Android 应用程序沙盒中执行，将应用程序数据和代码执行与设备上的其他应用程序分离。如前所述，这种分离形成了第一层防御。

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/[包名称 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 分别分配了一个目录，并由不同的用户帐户运行：

```
drwx------  4 u0_a97              u0_a97              4096 2017-01-18 14:27 c
om.android.calendar
drwx------  6 u0_a120             u0_a120             4096 2017-01-19 12:54 c
om.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（在其 AndroidManifest.xml 文件中具有 sharedUserId）时，每个应用程序都可以访问另一个应用程序的数据目录。请参见以下示例以在 NFC 应用程序中实现这一点：

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

5.1.3.3. Linux 用户管理

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 环境中的用户管理用法，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）。组的权限与 android.permission.INTERNET 权限关联。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
```

组 id 和权限之间的关系在以下文件中定义：

frameworks/base/data/etc/platform.xml

```
<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>
```

5.1.3.4. 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/zyote 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。

5.1.3.5. 应用生命周期

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。

- 前台进程（例如：在屏幕顶部运行的 Activity 或正在运行的广播接收器（BroadcastReceiver））。
- 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。
- 服务进程是一个用于托管使用 startService 方法启动服务的进程。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。

5.1.3.6 应用程序包

Android 应用程序可以以两种形式发布：Android 安装包（APK）文件或 Android 应用程序包（.aab）。Android 应用程序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 Google Play。应用程序包是签名的二进制文件，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，这在应用程序包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
```

如果要创建准备部署到测试设备的已签名 APK，请使用：

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

我们建议您对无论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。

5.1.3.7. Android Manifest

每个应用程序都有一个 Android Manifest 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 安装包（APK）文件的根目录中。

Manifest 文件描述应用程序结构、其组件（活动、服务、内容提供者和 Intent 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 Manifest 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、公开的内容提供者、与 Intent 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 manifest 选项的完整列表在官方 Android manifest 文件文档中。

5.1.3.8. 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：

- Activity
- Fragment
- Intent
- Broadcast receiver 广播接收器。
- Content providers and service 内容提供者和服务。

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。

5.1.3.8.1. 活动（Activity）

Activity 构成了任何应用程序的可见部分。每个屏幕上有一个 activity，所以一个有三个不同屏幕的应用程序需要实现三个不同的 activitiy。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（fragment）、视图（view）和布局（layout）。

每个 activity 都需要使用以下语法在 Android Manifest 中声明：

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

无法显示清单中未声明的 activity，尝试启动它们将引发异常。

与应用程序一样，activity 也有自己的生命周期，需要监视系统更改以处理它们。activity 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。

5.1.3.8.2. 片段（Fragment）

Fragment 表示 activity 中用户界面的一个行为或一部分。Fragment 是在 Android 的 3.0 版本蜂巢（API 级别 11）中引入的。

Fragment 是用来包装接口的一部分，以便于重用和适应不同的屏幕大小。Fragment 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 activity 集成才能发挥作用：Fragment 不能单独存在。它们有自己的生命周期，这与实现它们的 activity 的生命周期有关。

因为 fragment 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。还有一些其他的管理器；可以参考“Android fragment 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 Fragment：

Java 示例:

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

Kotlin 示例:

```
class MyFragment : Fragment() {
    ...
}
```

Fragment 不需要在 Manifest 文件中声明，因为它们依赖于 activity。

为了管理其 fragment，activity 可以使用 Fragment 管理器（FragmentManager 类）。这个类使得查找、添加、删除和替换相关 fragment 变得很容易。

fragment 管理器可以通过以下方式创建：

Java 示例：

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

Kotlin 示例：

```
var fm = fragmentManager
```

fragment 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。fragment 可以被声明为持久的，这样即使其 Activity 被破坏，系统也能保持其状态。

5.1.3.8.3. 内容提供者（Content Provider）

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，内容提供者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（包括原生应用程序）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 manifest 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://模式。不管数据源的类型如何（SQLite 数据库、平面文件等），寻址方案总是相同的，因此可以抽象源并为开发人员提供唯一的方案。内容提供者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 manifest 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。

5.1.3.8.3. 服务（Service）

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 intent 和通知等），而不显示用户界面。服务旨在长期运行进程。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。这使得服务成为运行后台任务的最佳选择。请注意，服务和活动一样，都是在主应用程序线程中执行的。除非您另有指定，否则服务不会创建自己的线程，也不会在单独的进程中运行。

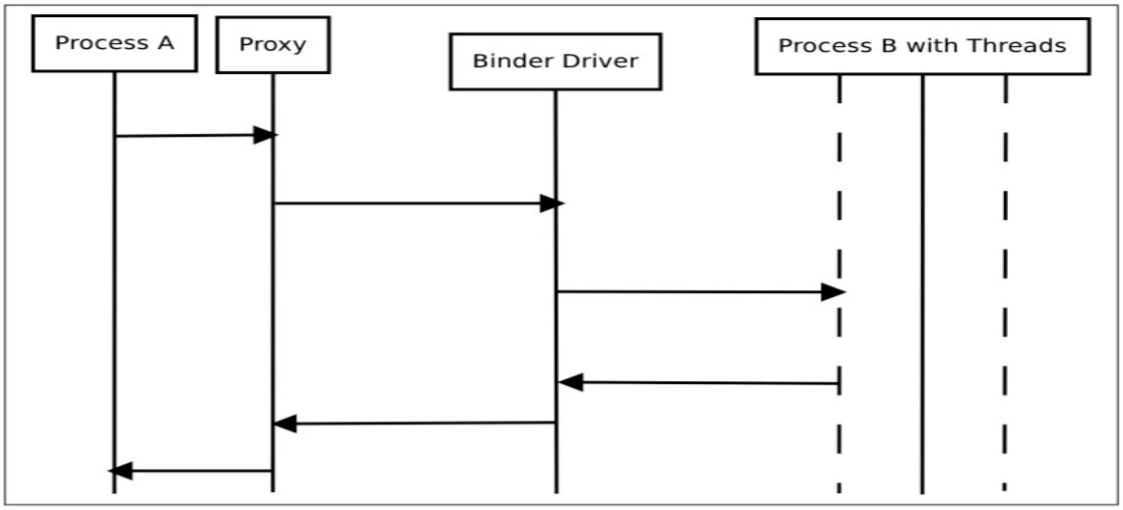
5.1.3.8.4. 进程间通信

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：

- Binder 驱动：内核级驱动程序。
- Binder 协议：用于与 Binder 驱动程序通信的基于 ioctl 的低级协议。
- IBinder 接口：Binder 对象实现的已经明确定义的行为。
- 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。



- Binder 概览 – 图片来源: Android Binder by Thorsten Schreiber

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/Binder 对列表。使用 addService 添加服务，并使用中的静态 getService 方法按名称检索服务 android.os.ServiceManager:

Java 示例:

```
public static IBinder getService(String name) {
        try {
            IBinder service = sCache.get(name);
            if (service != null) {
                return service;
            } else {
                return getIServiceManager().getService(name);
            }
        } catch (RemoteException e) {
            Log.e(TAG, "error in getService", e);
        }
        return null;
    }
```

Kotlin 示例:

```
companion object {
        private val sCache: Map<String, IBinder> = ArrayMap()
        fun getService(name: String): IBinder? {
            try {
```

```
                val service = sCache[name]
                return service ?: getIServiceManager().getService(name)
            } catch (e: RemoteException) {
                Log.e(FragmentActivity.TAG, "error in getService", e)
            }
            return null
        }
    }
```

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]
```

5.1.3.8.5. 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：

- 启动 activity。
  - 一个 activity 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 activity 的新实例。intent 描述 activity 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 Android 5.0（API 级别 21）及更高版本，您可以使用 JobScheduler 启动服务。
- 发送广播。
  - 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：

Java 示例:

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

Kotlin 示例:

```
var intent = 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：

Java 示例:

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

Kotlin 示例:

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

Intent 过滤器是 Android Manifest 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 intent 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。

5.1.3.8.6. 广播接收器（Broadcast Receiver）

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 manifest 文件中声明。清单必须指定广播接收器和 Intent 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 Intent；当相关 Intent 被提出时，系统会自动启动应用程序。

manifest 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>
```

请注意，在本例中，广播接收器不包括 android:exported 属性。由于至少定义了一个过滤器，默认值将设置为“true”。在没有任何筛选器的情况下，它将被设置为“false”。

另一种方法是在代码中动态创建接收器。接收器可以使用 Context.registerReceiver 方法注册。

动态注册广播接收器的示例：

Java 示例:

```
// 定义 broadcast receiver
BroadcastReceiver myReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "Intent received by myReceiver");
    }
};
// 定义一个带有 broadcast receiver 需要监听的动作的 intent 过滤器
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction("com.owasp.myapplication.MY_ACTION");
// 注册 broadcast receiver
registerReceiver(myReceiver, intentFilter);
// 反注册 broadcast receiver
unregisterReceiver(myReceiver);
```

Kotlin 示例:

```
// 定义 broadcast receiver
val myReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        Log.d(FragmentActivity.TAG, "Intent received by myReceiver")
    }
}
//定义一个带有 broadcast receiver 需要监听的动作的 intent 过滤器
val intentFilter = IntentFilter()
intentFilter.addAction("com.owasp.myapplication.MY_ACTION")
// 注册 the broadcast receiver
registerReceiver(myReceiver, intentFilter)
// 反注册 the broadcast receiver
unregisterReceiver(myReceiver)
```

请注意，当启动相关 `intent` 时，系统会自动启动带有注册接收者的应用程序。

根据广播概览，如果广播不是专门针对某个应用程序的，则该广播被视为“隐式”。在收到隐式广播后，Android 将列出在其过滤器中注册了给定操作的所有应用程序。如果为同一操作注册了多个应用程序，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特性是，它们可以被优先排序；这样，intent 将根据其优先级传递给所有授权接收者。可以通过 android:priority 属性以及通过 `IntentFilter.setPriority` 方法为 `manifest` 中的 intent 过滤器分配优先级。但是，请注意，具有相同优先级的接收器将以任意顺序运行。

如果您的应用程序不应该跨应用程序发送广播，请使用本地广播管理器（LocalBroadcastManager）。它们可用于确保仅从内部应用程序接收到 intent，而来自任何其他应用程序的任何 intent 都将被丢弃。这对于提高应用程序的安全性和效率非常有用，因为不涉及进程间通信。但是，请注意，LocalBroadcastManager 类已被弃用，Google 建议使用 LiveData 等替代方法。

有关广播接收器的更多安全注意事项，请参阅安全注意事项和最佳实践。

5.1.3.8.7. 隐式广播接收器限制

根据后台优化，针对 Android 7.0（API 级别 24）或更高版本的应用程序不再接收 `CONNECTIVITY_ACTION` 广播，除非它们使用 Context.registerReceiver()注册广播接收器。系统也不发送 `ACTION_NEW_PICTURE` 和 `ACTION_NEW_VIDEO` 广播。

根据后台执行限制，针对 Android 8.0（API 级别 26）或更高版本的应用程序不再可以在其 manifest 中为隐式广播注册广播接收器，但隐式广播例外中列出的应用程序除外。通过调用 Context.registerReceiver 在运行时创建的广播接收器不受此限制的影响。

根据对系统广播的更改，从 Android 9（API 级别 28）开始，NETWORK_STATE_CHANGED_ACTION 广播不会接收有关用户位置或个人识别数据的信息。

5.1.4. Android 应用发布

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，它们必须先进行签名。应用程序开发人员利用加密签名放置的可验证标记，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。大多数商店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，软件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。

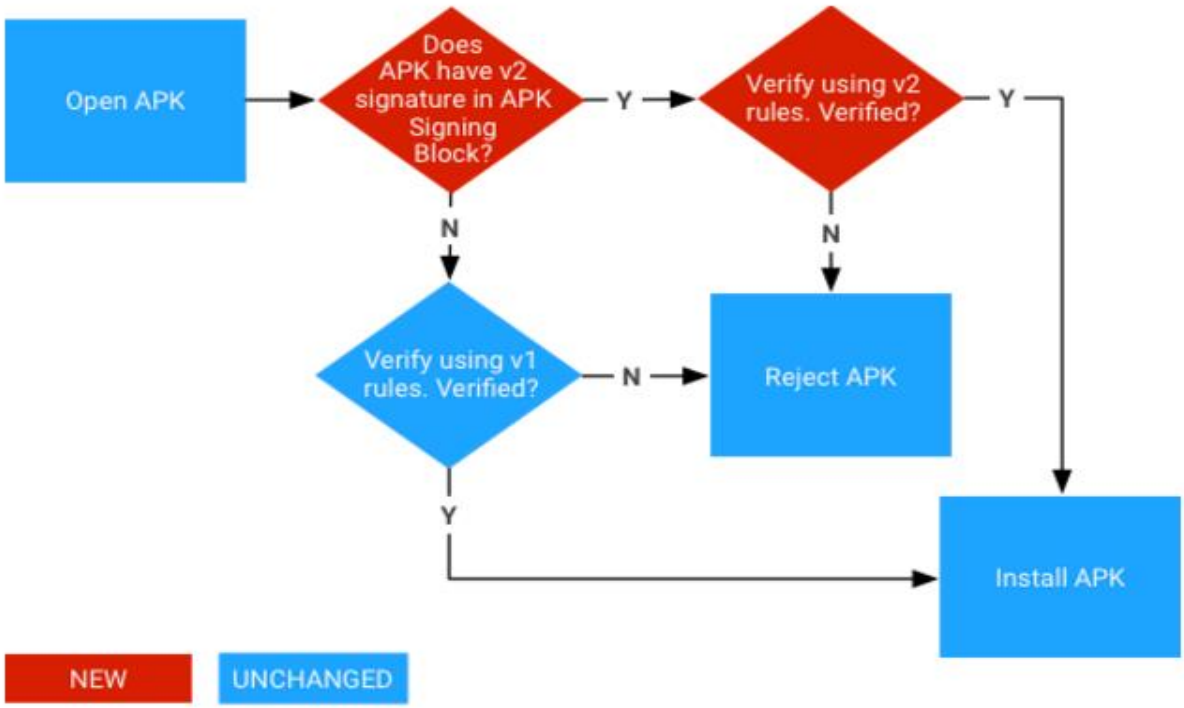
5.1.3.2.1. JAR 签名（v1 方案）

应用程序签名的原始版本将已签名的 APK 实现为标准的已签名 JAR，它必须包含 META-INF/MANIFEST.MF 中的所有条目。所有文件都必须使用通用证书签名。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处理不受信任的数

据结构，并且验证器会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，这需要相当多的时间和内存。

5.1.3.2.2. APK 签名方案（v2 方案）

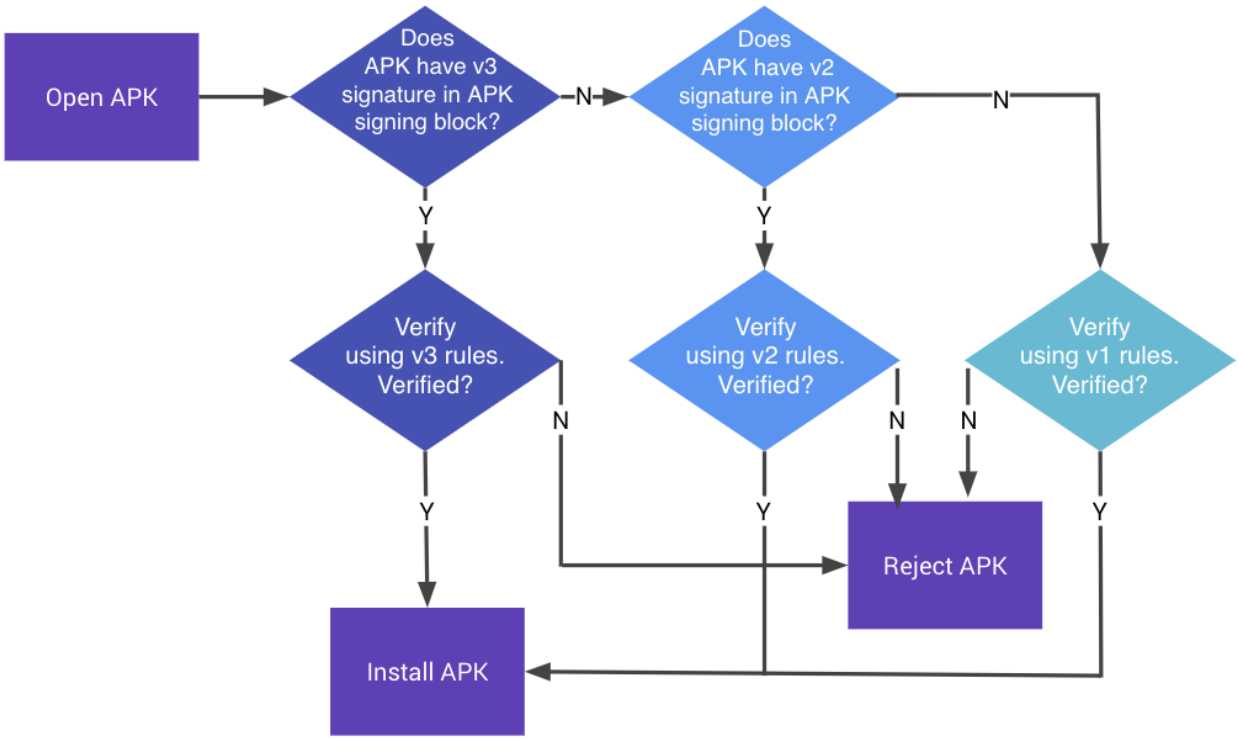
使用 APK 签名方案，对完整的 APK 进行散列和签名，创建 APK 签名块并将其插入到 APK 中。在验证期间，v2 方案检查整个 APK 文件的签名。这种形式的 APK 验证速度更快，针对修改提供了更全面的保护。可以在下面看到 v2 方案的 APK 签名验证过程。



5.1.3.2.3 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 Android 9（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，因为旋转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。



5.1.3.2.4 APK 签名方案（v4 方案）

APK 签名方案 v4 与 Android 11（API 级别 30）同时推出。这要求随其启动的所有设备默认启用 fs-verity。fs-verity 是 Linux 内核的一项功能，由于其高效的文件哈希计算，主要用于文件身份认证（检测恶意修改）。只有在内容根据启动时加载到内核密钥环的可信数字证书进行验证时，读取请求才会成功。

v4 签名需要补充 v2 或 v3 签名，与以前的签名方案相比，v4 签名存储在单独的文件中<apk name>.apk.idsig。在使用 apksigner verify 验证 v4 签名的 APK 时，请记住使用`--v4-signature-file` 参数指定它。

您可以在 Android 开发者文档中找到更详细的信息。

5.1.3.2.5. 创建证书

Android 使用公共/私有证书来签署 Android 应用程序（.apk 文件）。证书是信息的集合；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。公共和私有证书是关联的。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位于当前目录中的文件“myKeyStore.jks”中：

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

安全地存储密钥，确保它在整个生命周期内保持机密是至关重要的。任何获得密钥访问权限的人都可以使用您无法控制的内容向您的应用发布更新（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。密钥存储在二进制文件中，可以使用密码进行保护；此类文件称为密钥库（*KeyStores*）。密钥库密码应为强密码，并且只有密钥创建者知道。因此，密钥通常存储在开发人员访问受限的专用构建机器上。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，Google Play 将要求证书至少在 2033 年 10 月 22 日之前保持有效。

5.1.3.2.6. 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功能，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；如果其他人能够访问此类文件，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（API 级别 24 及更高）提供的“apksigner”工具从命令行进行签名。它位于[SDK 路径]/build tools/[version]。对于 API 24.0.2 及以下版本，可以使用“jarsigner”，它是 Java JDK 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）。该应用程序将命名为‘mySignedApp.apk’并将准备好发布到应用商店。

5.1.3.2.6.1 Zipalign

在分发之前，应该始终使用 zipaling 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。

5.1.3.3. 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，Google Play 是最知名、最受信任、最受欢迎的商店，而 Google 本身也提供了这一点。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。

应用程序可以从多种来源安装在 Android 设备上：本地通过 USB、Google 官方应用商店（Google Play 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 Google 只需扫描已知的恶意软件签名即可；这可以最大限度地缩短发布过程开始到应用公开可用之间的时间。.

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 Google Play 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。

5.1.5. Android 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 自定义 URL 方案
- 验证用户在输入字段中的所有输入。
- 验证 WebView 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 反逆向防御。

5.2. Android 基础安全测试

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介绍可用于测试 Android 应用程序安全缺陷的基本流程和技术。这些基本过程是以下章节中概述的测试用例的基础。

5.2.1. Android 测试设置

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 也附带了一个 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原生库的应用程序，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）。

有时，从计算机上显示或控制设备可能很有用。要实现这一点，可以使用 Scrcpy。

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	不支持	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，例如带或不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 系统无关 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。
原生库支持	原生库通常是为 ARM 设备构建的，所以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原生库。
恶意软件威胁性	恶意软件样本可能会感染设备，但如果您可以清除设备存储并刷入干净的固件，从而将其恢复到出厂设置，这应该不是问题。请注意，有恶意软件样本试图利用 USB 网桥进行攻击。	恶意软件样本可以感染模拟器，但模拟器可以简单地删除并重新创建。还可以创建快照并比较不同的快照，以帮助恶意软件分析。请注意，存在试图攻击虚拟机监控程序的恶意软件概念验证。

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 Google 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、在线可用的射频驱动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 Google 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），并具有近原生体验。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 mod 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者模式和 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“关于手机”视图的“版本号”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，“开发者选项”将显示在“设置”菜单的底部。一旦开发者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 Android 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 Google 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 root 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 boot loader。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

Google 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 boot loader 被解锁，设备将不会失去保修，并且 Google 提供了许多工具来支持 root 操作。XDA 论坛上发布了一份所有主要品牌设备的 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 systemless”）。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。如果您使用官方的 Android 版本并计划升级它，Magisk 在 GitHub 上提供了一个教程。

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 模块官方源。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 SDK 版本 27）。

5.2.1.2.3.3 Root 检测

在“Android 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。

5.2.2. 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者模式和 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 或输入 `exit` 来退出

一旦进入远程 shell，如果你的设备已经 root，或者你使用的是模拟器，你可以通过运行 su 获得 root 权限：

```
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 <设备 IP 地址>`。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。为了安全起见更建议使用 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.。

5.2.2.1.1.2 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认证），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-p 8022 <ip 地址>（其中 ip 地址是实际的远程设备 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 应用

虽然与远程 shell 相比，通常使用诸如 Termux 之类的设备上 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。

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：以下命令将复制 foo.txt 文件从当前目录（本地）到 SD 卡文件夹（远程）：

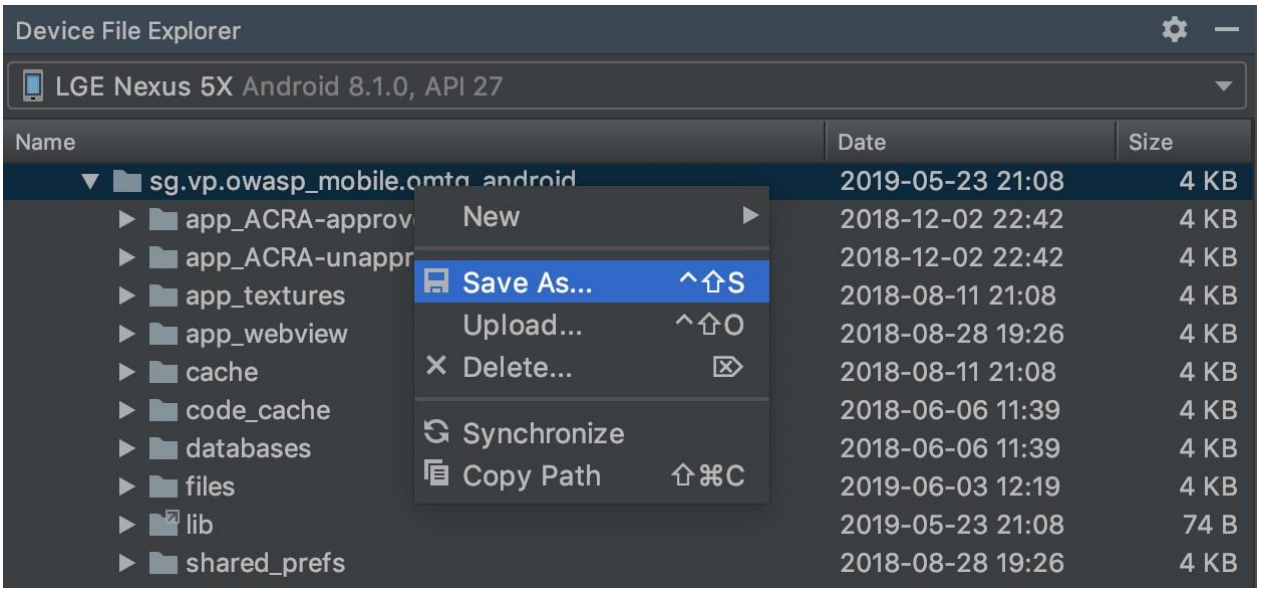
```
adb push foo.txt /sdcard/foo.txt
```

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 Android Studio 设备文件管理器

Android Studio 有一个内置的设备文件资源管理器，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“Recommended Tools - 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_file>`。这将下载该文件到您的工作目录。同样的方法，您可以使用 `file upload` 上传文件。

```
...[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.xm
l
```

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 `file download <some_file>`。

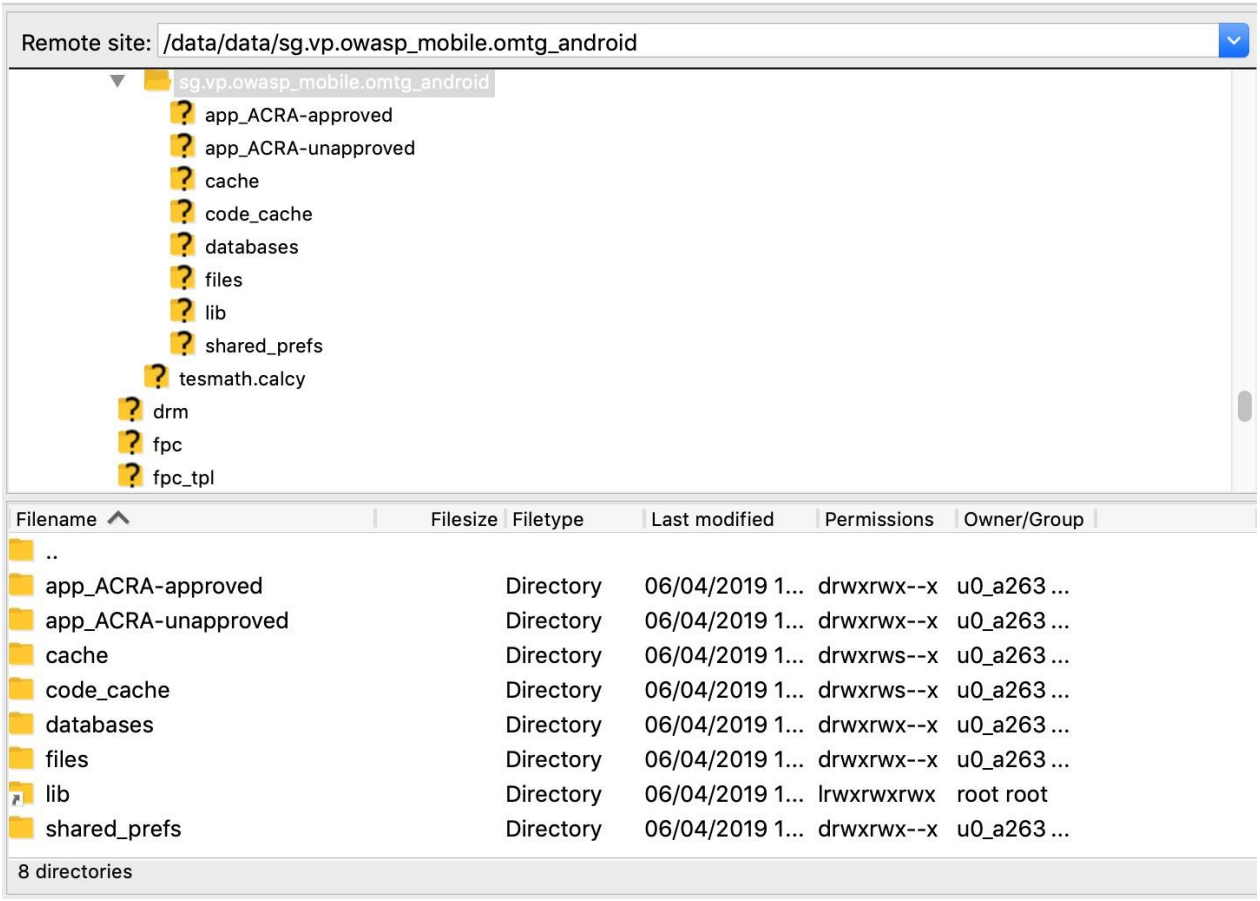
5.2.2.2.4. 使用 Termux

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）：



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，其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 Google Play 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. 使用 gplaycli

您可以使用 gplaycli 下载（-d）指定 APK，方法是指定其 AppID（使用-p 以显示进度条，使用-v 以显示详细信息）：

```
$ gplaycli -p -v -d com.google.android.keep
[INFO] GPlayCli version 3.26 [Python3.7.4]
[INFO] Configuration file is ~/.config/gplaycli/gplaycli.conf
[INFO] Device is bacon
[INFO] Using cached token.
[INFO] Using auto retrieved token to connect to API
[INFO] 1 / 1 com.google.android.keep
[################################] 15.78MB/15.78MB - 00:00:02 6.57MB/s/s
[INFO] Download complete
```

`com.google.android.keep.apk` 将位于当前目录中。正如您可能想象的那样，这种方法是下载 APK 的一种非常方便的方法，尤其是在自动化方面。

> 您可以使用自己的 Google Play 凭据或令牌。默认情况下，gplaycli 将使用内部提供的令牌。

5.2.2.3.3. 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 root 设备或未 root 设备获取应用程序，可以使用以下方法：

使用 `adb pull` 检索 APK。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，您还需要找到其系统中存储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 将下载到当前工作目录中。

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。

5.2.2.4. 测试即时应用

通过 Google Play Instant，你可以创建即时应用，这些应用可以从浏览器或应用商店的 "立即尝试 "按钮中立即启动，从 Android 5.0（API 级别 21）开始。它们不需要任何形式的安装。即时应用程序有一些限制：

- 你可以拥有的即时应用程序的数量是有限的。
- 只能使用数量较少的权限，这些权限在 Android 即时应用程序文档中有所记载。

这些组合可能导致不安全的决定，例如：从应用程序中剥离过多的授权/认证/保密逻辑，从而导致信息泄露。

注意：即时应用程序需要一个应用程序包。应用程序包在 "Android 平台概述 "一章的 "应用程序包"部分有描述。

静态分析考虑：

静态分析可以在对下载的即时应用程序进行逆向工程后进行，也可以通过分析应用程序包进行。 当你分析应用程序包时，检查 Android Manifest，看给定模块（无论是基本模块还是设置了 dist:module 的特定模块）是否设置了 dist:module dist:instant="true"。接下来，检查各种入口点，哪些入口点被设置了（通过<data android:path="</PATH/HERE>" />）。

现在，就像你对任何 Activity 所做的那样，跟踪这些入口点，并进行检查。

- 是否有任何由应用程序检索的数据应要求对该数据进行隐私保护？如果是这样，所有必要的控制措施是否已经到位？
- 所有的通信都是安全的吗？
- 当你需要更多的功能时，是否也下载了正确的安全控制措施？

动态分析考虑：

有多种方法可以开始对你的即时应用程序进行动态分析。在所有情况下，你都必须先安装对即时应用程序的支持，并将 ia 可执行文件添加到你的$PATH 中。

即时应用程序支持的安装是通过以下命令完成的：

```
cd path/to/android/sdk/tools/bin && ./sdkmanager 'extras;google;instantapps'
```

接下来，你必须把 path/to/android/sdk/extras/google/instantapps/ia 加入你的$PATH。准备工作完成后，你可以在运行 Android 8.1（API 级别 27）或更高版本的设备上测试即时应用程序。应用程序可以用不同的方式进行测试：

- 在本地测试该应用。通过 Android Studio 部署应用程序（并在 "运行/配置 "对话框中启用 "作为即时应用程序部署 "复选框）或使用以下命令部署应用程序：
  `ia run output-from-build-command <app-artifact>`
- 使用 Play Console 测试该应用程序：
  1. 将你的应用程序包上传到 Google Play 控制台
  2. 准备好上传的包，以便发布到内部测试跟踪。
  3. 在设备上登录内部测试员账户，然后从外部准备的链接或通过测试员账户中的 App Store 中的立即试用按钮启动您的即时体验。

现在，你可以测试应用程序，检查是否：

- 有任何需要隐私控制的数据，以及这些控制是否已经到位。
- 所有的通信都有足够的安全保障。
- 当你需要更多的功能时，是否也为这些功能下载了正确的安全控制？

5.2.2.5. 重新打包应用程序

如果你需要在非越狱的设备上进行测试，你应该学习如何重新打包一个应用程序，以便在上面进行动态测试。

使用一台电脑来执行 objection Wiki 中 "修补 Android 应用程序 "一文中指出的所有步骤。一旦你完成了，你就可以通过调用 objection 命令来修补 APK：

```
objection patchapk --source app-release.apk
```

然后需要使用 adb 安装修补过的应用程序，如 "安装应用 "中解释的那样。

> 这种重新打包的方法对于大多数使用情况来说已经足够了。对于更高级的重新打包，请参考 "Android 上的篡改和逆向工程--补丁、重新打包和重新签名"

5.2.2.6. 安装应用

使用 adb install 在虚拟机或连接的设备上安装 APK。

`adb install` apk 安装路径

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。

5.2.2.7. 信息收集

分析应用程序的一个基本步骤是收集信息。这可以通过检查主机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？
- 应用程序使用哪些原生库？
- 应用程序定义了哪些应用程序组件？任何服务或内容提供者？
- 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？

5.2.2.7.1. 列出已安装应用

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）参数以仅显示第三方应用程序及其 APK 文件位置（-f），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==/ba
se.apk
```

使用 `frida-ps -Uai` 获取连接的 USB 设备（-U）上已安装(-i)的所有应用（-a）：

```
$ frida-ps -Uai
  PID  Name                                    Identifier
-----  ---------------------------------------  --------------------------
-----------
  766  Android System                           android
21228  Attack me if u can                       sg.vp.owasp_mobile.omtg_andr
oid
 4281  Termux                                   com.termux
    -  Uncrackable1                            sg.vantagepoint.uncrackable1
```

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。

5.2.2.7.2. 探索应用程序包

一旦收集了要作为目标的应用程序的包名称，就需要开始收集有关它的信息。首先，按照“基本测试操作-获取和提取应用程序”中的说明检索 APK。

APK 文件实际上是 ZIP 文件，可以使用标准的解压工具（如 unzip）进行解包。然而，我们建议使用 apktool，它可以额外解码 AndroidManifest.xml，并将应用程序的二进制文件（classes.dex）反汇编为 smali 代码。:

```
$ apktool d UnCrackable-Level3.apk
$ tree
.
├── AndroidManifest.xml
├── apktool.yml
├── lib
├── original
│   ├── AndroidManifest.xml
│   └── META-INF
│       ├── CERT.RSA
│       ├── CERT.SF
│       └── MANIFEST.MF
├── res
└── smali
```

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。
- original/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：列出相应资源列表和 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 unzip 程序解压会遗漏一些文件，例如 AndroidManifest.xml 文件不可读，最好使用 apktool 解包 APK。

```
$ 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
```

5.2.2.7.2.1 Android Manifest

Android Manifest 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：`permission`（见“Android 平台 API”）。
- 允许备份：`android:allowBackup`（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”）。
- 可调式属性：`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。

5.2.2.7.2.2 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(`classes.dex`)可以在应用程序包的根目录中找到。它被称为 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 `smali` 文件夹。它包含一种称为 smali 中间语言的反汇编 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“查看反编译 Java 代码”一节。

5.2.2.7.2.3 编译的应用程序二进制

在某些情况下，检索已编译的应用程序二进制文件（.odex）可能是有用的。首先获取应用程序的数据目录的路径：

```
adb shell pm path com.example.myapplication
package:/data/app/~~DEMFPZh7R4qfUwwwh1czYA==/com.example.myapplication-
pOslqiQkJclb_1Vk9-WAXg==/base.apk
```

删除/base.apk 部分，添加/oat/arm64/base.odex，并使用生成的路径从设备中提取 base.odex:

```
adb root
adb pull /data/app/~~DEMFPZh7R4qfUwwwh1czYA==/com.example.myapplication-
pOslqiQkJclb_1Vk9-WAXg==/oat/arm64/base.odex
```

请注意，确切的目录将根据你的 Android 版本而有所不同。如果找不到 /oat/arm64/base.odex 文件，请在 `pm path` 返回的目录中手动搜索。

5.2.2.7.2.4 原生库

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原生库的所有信息了，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“反汇编原生代码”一节。

5.2.2.7.2.5 其他应用资源

通常 APK 中 根目录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。

5.2.2.7.3. 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-
kR0ovWl9eoU_yh0jPJ9caQ==/base.ap
```

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]`或 `/data/user/0/[package-name]`。

- 外部数据目录在/storage/emulated/0/Android/data/[package-name] 或 /sdcard/Android/data/[package-name]。
- 指向/data/app/中的应用程序包的路径。

应用程序使用内部数据目录存储运行时创建的数据，其基本结构如下：

```
...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 Android 5.0（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++编写的原生库。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原生库的平台的子目录，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。

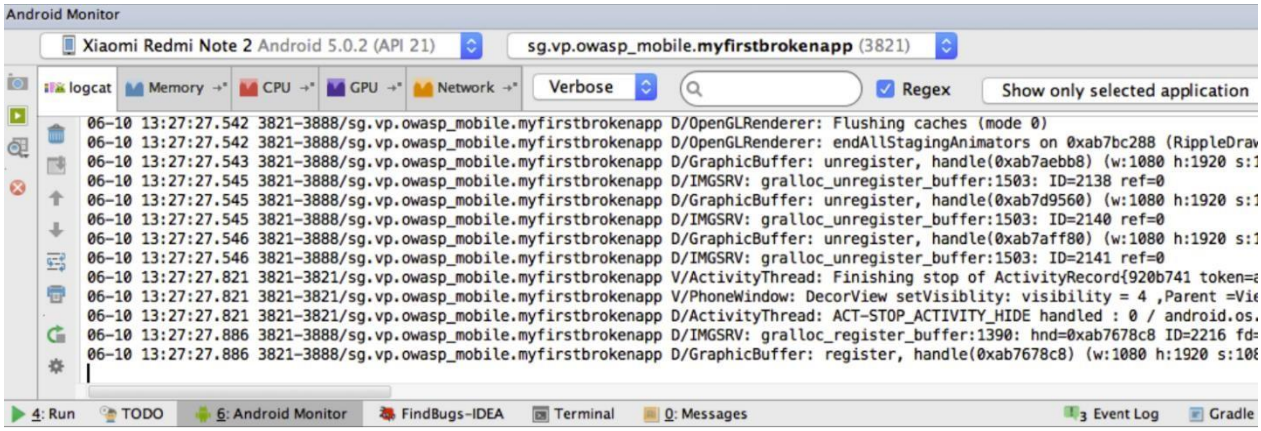
然而，应用程序可能不仅在这些文件夹中而是在父文件夹（/data/data/[package-name]）中存储更多数据。

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。

5.2.2.7.4. 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. 建立网络测试环境

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
```

在某些生产版本中，您可能会遇到挂载错误：'/system' not in /proc/mounts.

在这种情况下，您可以根据 Stack Overflow 文章将上面的行`$ mount -o rw,remount /system;` 替换为`$ mount -o rw,remount /`

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。

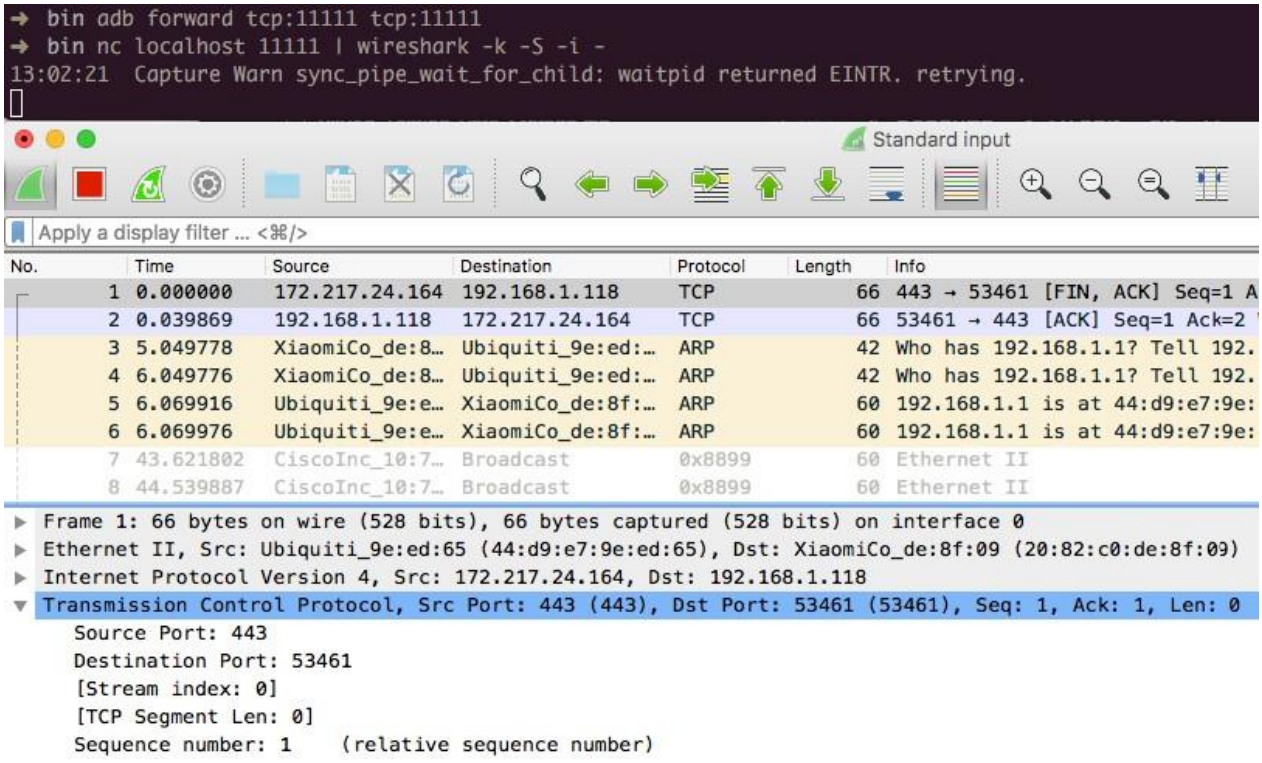
要访问端口 11111，需要通过 adb 将端口转发到您的机器。

```
adb forward tcp:11111 tcp:11111
```

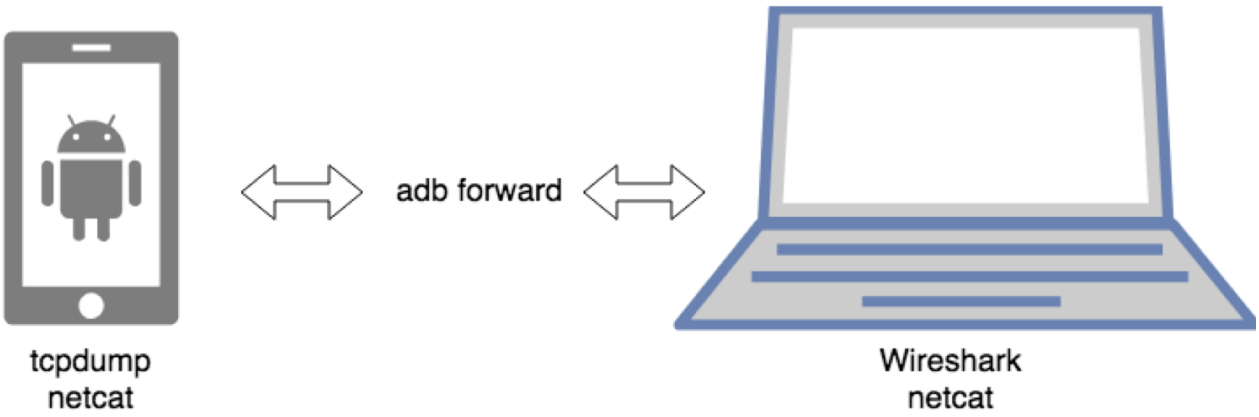
下面的命令通过 netcat 和 Wireshark 管道将您连接到转发端口。

```
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。



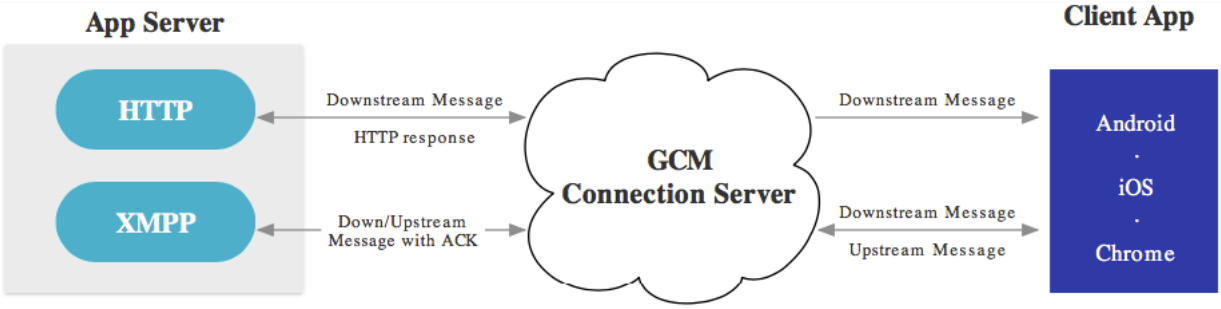
您可以使用 Wireshark 以可读的格式显示捕获的流量。找出使用了哪些协议以及它们是否未加密。捕获所有通信（TCP 和 UDP）很重要，因此应该执行待测应用程序的所有功能并对其进行分析。



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，如何测试终端？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(FCM/GCM)

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。如官方文档所述，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

FCM 使用端口 5228、5229 和 5230 进行 HTTP 通信。通常，仅使用端口 5228。

- 为 FCM 使用的端口配置本地端口转发。以下示例适用于 macOS：

```
$ 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）。

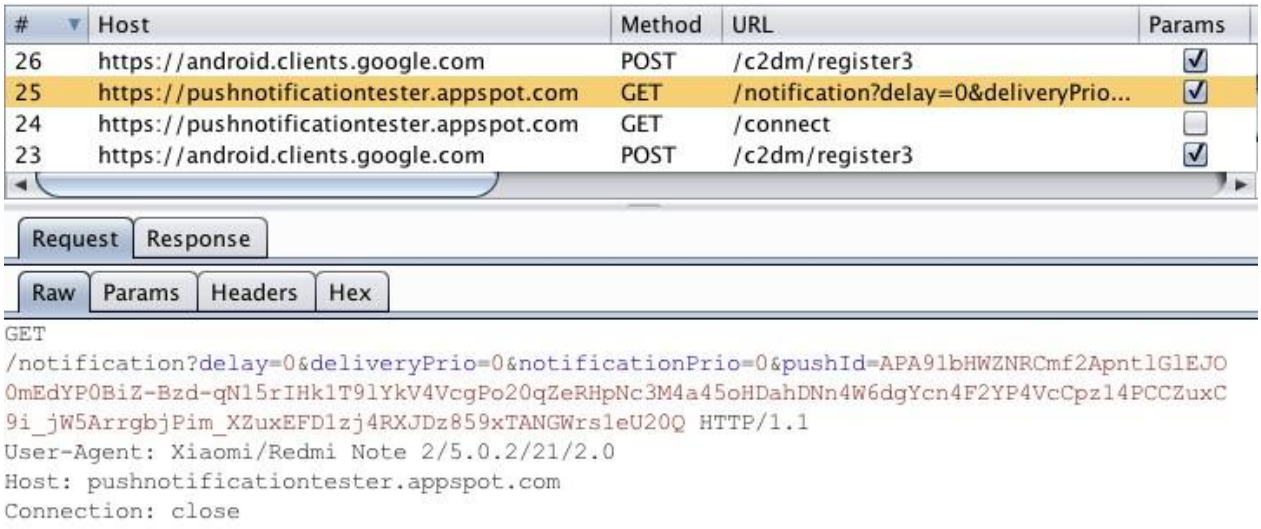
- 为 FCM 使用的端口配置本地端口转发。以下示例适用于 Mac OS：
  - $ echo "

```
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。



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

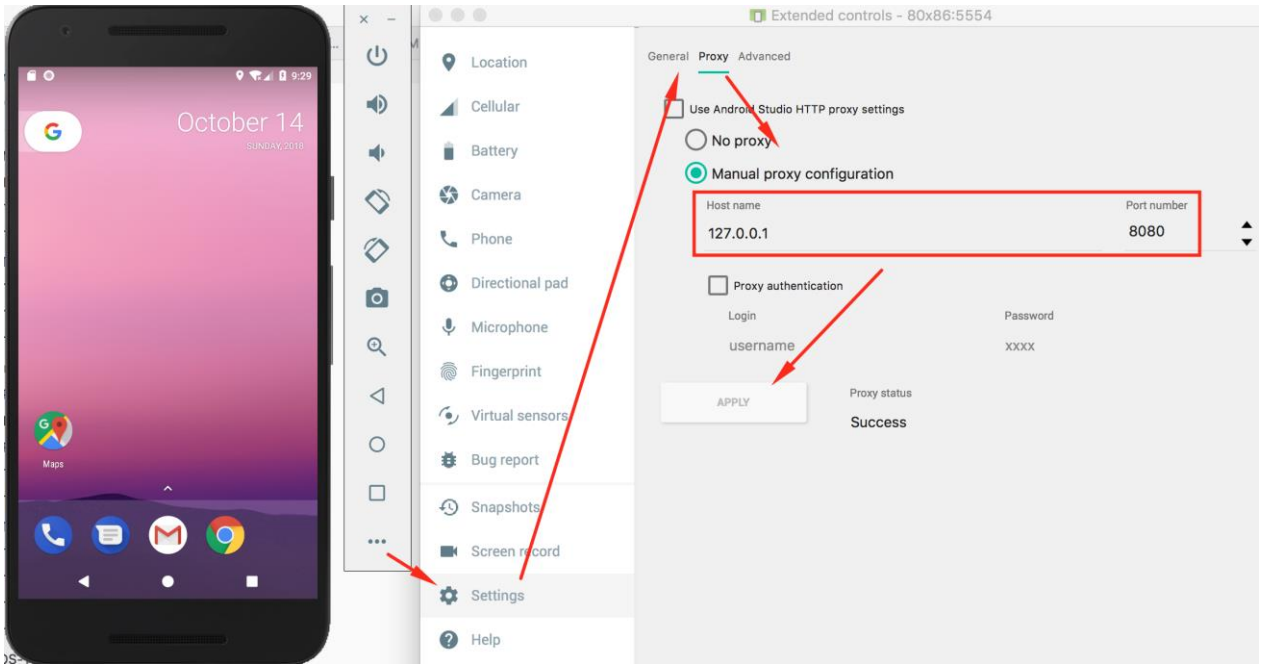
5.2.3.2.1.1 在 Android 虚拟设备（AVD）上设置 Web 代理

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（**Settings**）菜单。
- 单击代理（**Proxy**）选项卡。
- 选择“手动代理配置（**Manual proxy configuration**）”。
- 在“主机名（**Host Name**）”字段中输入“127.0.0.1”，在“端口号（**Port number**）”字段中输入代理端口（例如：“8080”）。
- 点击应用（**Apply**）。



HTTP 和 HTTPS 请求现在应该被路由到主机上的代理。如果没有，试着打开再关闭飞行模式。

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/，然后单击“CA 证书（CA Certificate）”按钮下载 `cacert.der`。
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（**Settings**）” -> “安全（**Security**）” -> “从 SD 卡安装（**Install from SD Card**.）”。
5、 向下滚动并点击 cacert.cer。

然后，系统会提示您确认证书的安装（如果尚未设置设备 PIN，还将要求您设置设备 PIN）。

这将在用户的证书库中安装证书（在 Genymotion VM 上测试）。为了将证书放在根证书库下，您可以执行以下步骤:

1. 用 adb root 和 adb shell 以 root 身份运行 adb。
2. 在/data/misc/user/0/cacerts-added/找到新安装的证书。
3. 将证书复制到以下文件夹/system/etc/security/cacerts/。
4. 重新启动 Android 虚拟机。

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主机必须连接到同一 Wi-Fi 网络。使用（现有）接入点或创建 ad-hoc 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 厂商对设置菜单的修改。
- 某些应用程序（例如 Chrome 浏览器）可能会显示 `NET::ERR_CERT_VALIDITY_TOO_LONG` 错误，可能是证书的有效期过长（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 证书具有相同有效期的证书。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，参考博客文章。

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 网站上找到在 Android 设备上设置 OWASP-ZAP 的视频。

其他一些区别：从 Android 8.0（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 Android 9（API 级别 28）开始，当握手过程中出现问题时，SSLSocket 和 SSLEngine 在错误处理方面的表现将略有不同。

如前所述，从 Android 7.0（API 级别 24）开始，Android 操作系统将不再默认信任用户 CA 证书，除非在应用程序中指定。在下一节中，我们将解释两种绕过 Android 安全控制的方法。

5.2.3.2.3 绕过网络安全配置

在本节中，我们将介绍几种绕过 Android 网络安全配置的方法。

5.2.3.2.3.1 将自定义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：

```
<certificates src=["system" | "user" | "raw resource"]
              overridePins=["true" | "false"] />
```

每个证书可以是以下证书之一：

- "原始资源（raw resource）"是一个 ID，指向一个包含 X.509 证书的文件。
- "系统（system）"用于预先安装的系统 CA 证书。
- "用户（user）"用于用户添加的 CA 证书。

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 Android 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 Android 应用程序通过以下网络安全配置来信任此证书：

```
<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。
- 进入 apktool 反编译应用程序时创建的目录，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，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改应用或通过 Frida 动态检测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。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，添加允许用户证书的新网络安全配置，构建并签署新的 APK，并且安装新的 APK 时启用 SSL 绕过。

```
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
```

5.2.3.2.3.2 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麻烦，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，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 GitHub 发布页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（Module）”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（Settings）”、“安全和位置（Security & location）”、“加密和凭据（Encryption & credentials）”、“从存储安装（Install from storage）”（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。

5.2.3.2.3.3 在系统信任的 CA 中手动添加代理的证书。

要不，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 过的设备上实现。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 固定（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 固定。理想情况下，您需要获得启用了这些控制的应用程序版本和禁用了这些控制的应用程序版本。这允许您能够分析控制是否正确实现，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，这并不总是可能的，您可能需要在启用了所有安全控制的应用程序上执行黑盒评估。下面的部分向您展示了如何规避不同应用程序的证书固定。

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处于 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 访客和 BYOD 网络非常有用，它提升了一个安全级别，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 8080 端口，它将通过 adb 重定向到您电脑上的 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处于 MITM 位置，您可能仍然看不到任何东西。这主要是由于以下原因：

- 该应用程序使用的是像 Xamarin 这样的框架，并不使用 Android 操作系统的代理设置或其他代理设置。
- 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 Burp 中代理选项卡（Proxy）/选项（Options）/编辑（Edit）界面激活“支持不可见代理（Support invisible proxying）”。

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 8080 端口上运行的代理。

```
iptables -t nat -A OUTPUT -p tcp --dport 80 -j DNAT --to-destination <Your-Pr
oxy-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 t
o:<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
```

如果要重置 iptables 配置，可以刷新规则：

```
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 lis
t 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。如果检测到使用了代理，应用会认为这是恶意的，从而不会正常工作。

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，可以通过调用 ProxyInfo 类并通过 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 以识别实际的类和方法名。

下面您可以找到 Frida 脚本的 boiler plate 源代码，该脚本将帮助您重载方法（在本例中称为 isProxySet），该方法重载为验证是否设置了代理，但始终返回 false。即使现在配置了代理，应用程序也会认为未设置任何代理，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
        }
    });
});
```

5.2.3.4. 绕过证书固定

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流量。

对于大多数应用程序来说，证书固定可以在几秒钟内被绕过，但前提是应用程序使用这些工具所涵盖的 API 功能。如果应用程序使用自定义框架或库实现 SSL 固定，则必须手动修改和停用 SSL 固定，这可能很耗时。

本节介绍了绕过 SSL 固定的各种方法，并指导你在现有工具无能为力时应该怎么做。

5.2.3.4.1. 绕过方法

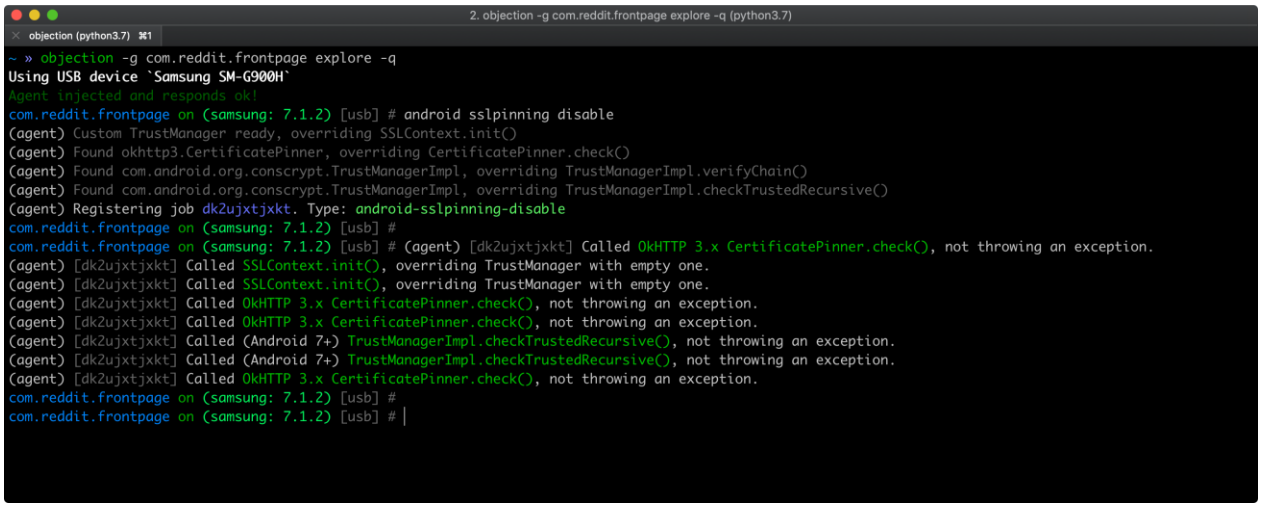
对于黑盒测试来说有几种绕过证书固定的方法，这取决于设备上可用的框架：

- Cydia Substrate：安装 Android-SSL-TrustKiller 软件包。
- Frida：使用 frida-multiple-unpinning 脚本。
- Objection：使用 android sslpinning disable 命令。
- Xposed：安装 TrustMeAlready 或 SSLUnpinning 模块。

如果你有一个安装了 frida-server 的 root 设备，你可以通过运行以下 Objection 命令绕过 SSL 固定（如果你使用的是未 root 设备，则重新打包你的应用程序）。

```
android sslpinning disable
```

下面是输出的示例：



更多信息请参见 Objection 关于禁用适用于 Android 的 SSL 固定的帮助，并检查 pinning.ts 文件以了解绕过的工作原理。

5.2.3.4.2. 绕过静态自定义证书固定

在应用程序的某个地方，必须定义端点和证书（或其哈希值）。在反编译应用程序后，你可以搜索到：

- 证书哈希：**`grep -ri "sha256\|sha1" ./smali`**。用代理 CA 的哈希替换原有证书哈希。或者，如果哈希绑定了域名，则可以尝试将域名修改为不存在的域名，使原有域名固定失效。这种方式对付代码混淆的 OkHTTP 实现非常有效。
- 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。
- 信任库文件：**`find ./ -type f \( -iname \*.jks -o -iname \*.bks \)`**。将代理的证书添加到信任库并确保它们的格式正确。

  请记住，应用程序可能包含没有扩展名的文件。 最常见的文件位置是 assets 和 res 目录，也应该对其进行调查。

举个例子，假设你发现一个使用 BKS（BouncyCastle）truststore 的应用程序，它存储在 `res/raw/truststore.bks` 文件里。为了绕过 SSL 固定，你需要用命令行工具 `keytool` 将你的代理服务器的证书添加到信任仓库。`keytool` 随 Java SDK 一起提供，执行该命令需要以下值：

- password - keystore 的密码。 在反编译的应用程序代码中查找硬编码的密码。
- providerpath - BouncyCastle Provider jar 文件的位置。 您可以从 BouncyCastle 页面下载。
- proxy.cer - 您的代理证书。
- aliascert - 用作代理证书别名的唯一值。

要添加代理的证书，请使用以下命令：

```
keytool -importcert -v -trustcacerts -file proxy.cer -alias aliascert -keystore "res/raw/truststore.bks" -provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath "providerpath/bcprov-jdk15on-164.jar" -storetype BKS -storepass password
```

要列出 BKS 信任库中的证书，请使用以下命令：

```
keytool -list -keystore "res/raw/truststore.bks" -provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath "providerpath/bcprov-jdk15on-164.jar"  -storetype BKS -storepass password
```

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。

如果应用程序使用原生库来实现网络通信，则需要进一步的逆向工程。 可以在博客文章中找到这种方法的一个示例：在 smali 代码中识别 SSL 固定逻辑，对其进行修补并重新编译 APK。

5.2.3.4.3. 绕过动态自定义证书固定

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固定逻辑会更加方便。

动态绕过方式最难的是找到合适的目标方法进行劫持，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，搜索字符串和许可证文件来标识所用库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插桩的方法。

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固定。如果可以修改 Builder.add 方法的参数，则可以将原有哈希更改为我们的证书哈希。可以通过以下两种方式找到合适的方法：

- 如前一节所述，搜索哈希和域名。因为实际的证书固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 命令查找可能的方法：`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 劫持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哈希，然后可以修改参数以绕过证书固定。

5.2.4. 参考文献

- 手动签名（Android 开发者文档） - https://developer.android.com/studio/publish/app-signing#signing-manually
- 自定义信任 - https://developer.android.com/training/articles/security-config#CustomTrust
- Android 网络安全配置培训 - https://developer.android.com/training/articles/security-config
- 安全分析师对 Android P 中的网络安全配置指南 - https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guid-to-network-security-configuration-in-android-p/
- Android 开发者文档 - https://developer.android.com/studio/publish/app-signing#signing-manually
- Android 8.0 行为变化 - https://developer.android.com/about/versions/oreo/android-8.0-changes
- Android 9.0 行为变化 - https://developer.android.com/about/versions/pie/android-9.0-changes-all#device-security-changes
- Codenames, Tags and Build Number - https://source.android.com/setup/start/build-numbers
- 创建和管理虚拟设备 - https://developer.android.com/studio/run/managing-avds.html
- 移动设备 root 指南 - https://www.xda-developers.com/root/
- API 级别 - 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
- 用 Logcat 进行调试 - https://developer.android.com/tools/debugging/debugging-log.html
- Android 的 APK 格式 - https://en.wikipedia.org/wiki/Androidapplicationpackage
- 使用 Tcpdump、nc 和 Wireshark 的 Android 远程嗅探 - https://blog.dornea.nu/2015/02/20/android-remote-sniffing-using-tcpdump-nc-and-wireshark/.
- 无线客户端隔离 - https://documentation.meraki.com/MR/FirewallandTrafficShaping/WirelessClientIsolation

5.3. Android 数据存储

保护身份认证令牌，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 的 API 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。

本章分为两个部分，第一部分从安全角度重点介绍数据存储理论，并简要解释和举例说明 Android 上的各种数据存储方法。

第二部分着重于通过使用静态和动态分析的测试用例来测试这些数据存储解决方案。

5.3.1.原理概览

存储数据对于许多移动应用程序至关重要。传统观点认为，应将尽可能少的敏感数据存储在永久本地存储器上。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，在应用程序每次启动时要求用户输入一个非常复杂的密码，这在可用性方面不是一个好主意。大多数应用程序必须在本地缓存某种身份认证令牌才能避免这种情况。如果场景需要，也可以保存个人识别信息（PII）和其他类型的敏感数据。

如果敏感数据未受到持续存储该数据的应用程序的适当保护，则该数据容易受到攻击。该应用程序可能能够将数据存储在多个位置，例如，在设备上或外部 SD 卡上。当您试图利用这类问题时，请考虑许多信息可能会被处理并存储在不同的位置。

首先，识别移动应用程序处理的和用户输入的信息类型很重要。其次，确定哪些被视为对攻击者有价值的敏感数据（例如密码、信用卡信息、PII）并不总是一项简单的任务，它在很大程度上取决于目标应用程序的背景。您可以在“移动应用程序安全测试”一章的“识别敏感数据”部分找到有关数据分类的更多详细信息。有关 Android 数据存储安全的一般信息，请参阅《Android 开发者指南》中存储数据的安全提示。

泄露敏感信息会产生多种后果，包括已解密信息。通常，攻击者可能会识别此信息并将其用于其他攻击，例如社会工程（如果已泄露 PII）、账户劫持（如果已泄露会话信息或身份认证令牌），以及从具有支付选项的应用程序收集信息（用于攻击和滥用）。

除了保护敏感数据外，您还需要确保从任何存储源读取的数据都经过验证，并可能经过清理。验证的范围通常从检查正确的数据类型到使用其他加密方法（如 HMAC），您可以验证数据的完整性。

5.3.2.数据存储方法概览

Android 根据用户、开发人员和应用程序的需要提供了许多数据存储方法。例如，一些应用程序使用数据存储来跟踪用户设置或用户提供的数据。可以通过多种方式持久存储此用例的数据。以下列出了 Android 平台上广泛使用的持久存储技术：

- Shared Preferences
- SQLite 数据库
- Firebase 数据库
- Realm 数据库
- 内部存储
- 外部存储
- Keystore

除此之外，Android 中还为各种用例构建了许多其他功能，这些功能也可能导致数据的存储，还应分别进行测试，例如：

- 记录功能

- Android 备份
- 进程内存
- 键盘缓存
- 屏幕截图

为了正确执行适当的测试用例，了解每个相关的数据存储功能非常重要。本概览旨在为这些数据存储方法提供概览大纲，为测试人员指明相关文档。

5.3.2.1. Shared Preferences

SharedReferences API 通常用于永久保存键值对的小集合。存储在 SharedReferences 对象中的数据被写入纯文本 XML 文件。SharedReferences 对象可以声明为全局可读（所有应用程序都可以访问）或私有。误用 SharedReferences API 通常会导致敏感数据的泄露。考虑以下示例：

Java 示例：

```
SharedPreferences sharedPref = getSharedPreferences("key", MODE_WORLD_READABL
E);
SharedPreferences.Editor editor = sharedPref.edit();
editor.putString("username", "administrator");
editor.putString("password", "supersecret");
editor.commit();
```

Kotlin 示例：

```
var sharedPref = getSharedPreferences("key", Context.MODE_WORLD_READABLE)
var editor = sharedPref.edit()
editor.putString("username", "administrator")
editor.putString("password", "supersecret")
editor.commit()
```

当 activity 被调用时，会使用提供的数据创建 key.xml 文件。此代码违反了几个最佳做法。

- 用户名和密码被明文存储在/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 允许所有应用程序访问和读取 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 已经从 API 级别 17 开始弃用。尽管新的设备可能不会受此影响，如果运行在 Android4.2（API 级别 17）以前发布的系统中，使用 android:targetSdkVersion 值小于 17 的应用程序可能会受此影响。

5.3.2.2. 数据库

Android 平台提供了许多数据库选项，如前一列表中所述。每个数据库选项都有自己的需要理解的特殊性和方法。

5.3.2.2.1. SQLite 数据库（未加密）

SQLite 是一个 SQL 数据库引擎，数据存储在.db 文件中。Android SDK 内置了对 SQLite 数据库的支持。用于管理数据库的主要软件包是 android.database.sqlite。例如，您可以使用以下代码在 activity 中存储敏感信息：

Java 示例：

```
SQLiteDatabase notSoSecure = openOrCreateDatabase("privateNotSoSecure", MODE_
PRIVATE, null);
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR, Pa
ssword VARCHAR);");
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');");
notSoSecure.close();
```

Kotlin 示例：

```
var notSoSecure = openOrCreateDatabase("privateNotSoSecure", Context.MODE_PRI
VATE, null)
notSoSecure.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR, Pa
ssword VARCHAR);")
notSoSecure.execSQL("INSERT INTO Accounts VALUES('admin','AdminPass');")
notSoSecure.close()
```

当 activity 调用时，会使用提供的数据创建 privateNotSoSecure 数据库文件并明文存储在 /data/data/<package-name>/databases/privateNotSoSecure

除 SQLite 数据库外，数据库目录可能包含多个文件：

- 日志文件：这些是用于实现原子性提交和回滚的临时文件。
- 锁定文件：锁定文件是锁定和日志功能的一部分，该功能旨在提高 SQLite 并发性并减少写入饥饿问题。

敏感信息不应存储在未加密的 SQLite 数据库中。

5.3.2.2.2. SQLite 数据库（加密）

使用 SQLCipher 库，可以对 SQLite 数据库进行密码加密。

Java 示例：

```
SQLiteDatabase secureDB = SQLiteDatabase.openOrCreateDatabase(database, "pass
word123", null);
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Passwo
rd VARCHAR);");
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');");
secureDB.close();
```

Kotlin 示例：

```
var secureDB = SQLiteDatabase.openOrCreateDatabase(database, "password123", n
ull)
secureDB.execSQL("CREATE TABLE IF NOT EXISTS Accounts(Username VARCHAR,Passwo
rd VARCHAR);")
secureDB.execSQL("INSERT INTO Accounts VALUES('admin','AdminPassEnc');")
secureDB.close()
```

接收数据库密钥的安全方法包括：

- 在应用程序打开后，要求用户使用 PIN 或密码解密数据库（弱密码和 PIN 容易受到暴力攻击）
- 将密钥存储在服务器上，并仅允许从 web 服务访问它（这样应用程序只能在设备联机时使用）

5.3.2.2.3. Firebase 实时数据库

Firebase 是一个拥有 15 多种产品的开发平台，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端，即使应用程序脱机也仍然可用。

通过进行以下网络调用，可以识别配置错误的 Firebase 实例：

```
https://_firebaseProjectName_.firebaseio.com/.json
```

firebaseProjectName 可以通过逆向工程从移动应用程序中检索。或者，分析师可以使用 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>
```

5.3.2.2.4. Realm 数据库

可供 Java 使用的 Realm 数据库在开发人员中越来越流行。数据库及其内容可以使用存储在配置文件中的密钥进行加密。

```
//getKey()方法可以从服务器或 KeyStore 或从密码获得。
RealmConfiguration config = new RealmConfiguration.Builder()
  .encryptionKey(getKey())
  .build();

Realm realm = Realm.getInstance(config);
```

如果数据库未加密，您应该能够获取数据。如果数据库已加密，请确定密钥是否在源代码或资源中硬编码，以及是否在 SharedPreferences 或其他位置未受保护地存储。

5.3.2.3. 内部存储

您可以将文件保存到设备的内部存储器中。默认情况下，保存到内部存储的文件是容器化的，设备上的其他应用程序无法访问。当用户卸载你的应用程序时，这些文件将被删除。以下代码段将持续将敏感数据存储到内部存储器。

Java 示例：

```
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();
}
```

Kotlin 示例：

```
var fos: FileOutputStream? = null
fos = openFileOutput("FILENAME", Context.MODE_PRIVATE)
fos.write(test.toByteArray(Charsets.UTF_8))
fos.close()
```

您应该检查文件模式以确保只有应用程序可以访问该文件。 您可以使用 MODE_PRIVATE 设置此访问权限。 MODE_WORLD_READABLE（不推荐）和 MODE_WORLD_WRITEABLE（不推荐）等模式可能会带来安全风险。

搜索 FileInputStream 类以找出在应用程序中打开和读取的文件。

5.3.2.4. 外部存储

每个 Android 兼容设备都支持共享外部存储。 此存储可能是可移动的（例如 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。 当启用 USB 大容量存储时，用户可以修改它们。 您可以使用以下代码片段将敏感信息作为文件 password.txt 的内容永久存储到外部存储中。

Java 示例：

```
File file = new File (Environment.getExternalFilesDir(), "password.txt");
String password = "SecretPassword";
FileOutputStream fos;
    fos = new FileOutputStream(file);
    fos.write(password.getBytes());
    fos.close();
```

Kotlin 示例：

```
val password = "SecretPassword"
val path = context.getExternalFilesDir(null)
val file = File(path, "password.txt")
file.appendText(password)
```

调用 activity 后，将创建文件，并将数据存储在外部存储中的明文文件中。

还值得知道的是，当用户卸载应用程序时，存储在应用程序文件夹 (data/data/<package-name>/) 之外的文件不会被删除。 最后，值得注意的是，在某些情况下，攻击者可以使用外部存储来任意控制应用程序。 有关更多信息：请参阅 Checkpoint 的博客。

5.3.2.5. KeyStore

Android KeyStore 支持相对安全的凭据存储。自 Android 4.3（API 级别 18）起，它提供了用于存储和使用应用程序私钥的公共 API。应用程序可以使用公钥创建新的私钥/公钥对来加密应用程序机密，并且可以使用私钥解密机密。

您可以通过在确认凭证流中进行用户身份认证来保护存储在 Android KeyStore 中的密钥。用户的锁屏凭据（图案、PIN、密码或指纹）被用于身份认证。

您可以在以下两种模式之一中使用存储的密钥：

1. 授权用户在认证后的有限时间内使用密钥。在此模式下，只要用户解锁设备，就可以使用所有密钥。您可以自定义每个密钥的授权期限。只有启用了安全锁屏，才能使用此选项。如果用户禁用安全锁屏，所有存储的密钥将永久无效。

2. 授权用户使用与一个密钥相关的特定加密操作。在此模式下，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 KeyStore 提供的安全级别取决于其实现，而实现取决于设备。大多数现代设备都提供了硬件支持的 KeyStore 实现：密钥是在可信执行环境（TEE）或安全单元（SE）中生成和使用的，操作系统无法直接访问它们。这意味着加密密钥本身无法轻松获得，即使是从 root 过的设备中。您可以使用密钥证明来验证硬件支持的密钥。您可以通过检查作为 **KeyInfo** 类一部分的 **`isInsideSecureHardware`** 方法的返回值来确定密钥是否在安全硬件内。

> 请注意，相关的 KeyInfo 表明，尽管私钥正确存储在安全硬件上，但密钥和 HMAC 密钥仍不安全地存储在多个设备上。

纯软件实现的密钥使用每用户加密主密钥进行加密。如果使用这种实现方式，攻击者可以在 root 过设备上访问文件夹/data/misc/keystore/中存储的所有密钥。由于用户的锁屏 pin 码/密

码用于生成主密钥，因此当设备锁定时，Android KeyStore 不可用。为了更安全，Android 9（API 级别 28）引入了 **unlockedDeviceRequied** 属性。通过将 true 传递给 **setUnlockedDeviceRequired** 方法，应用程序可以在设备锁定时防止其存储在 **AndroidKeystore** 中的密钥被解密，并且需要在允许解密之前解锁屏幕。

5.3.2.5.1. 硬件支持的 Android KeyStore

如前所述，硬件支持的 Android KeyStore 为 Android 提供了另一层深度防御安全概念。Android 6（API 级别 23）引入了 Keymaster 硬件抽象层（HAL）。应用程序可以验证密钥是否存储在安全硬件中（通过检查 **KeyInfo.isinsideSecureHardware** 是否返回 **true**）。运行 Android 9（API 级别 28）及更高版本的设备可以有一个 **StrongBox Keymaster** 模块，这是 Keymaster HAL 的一个实现，驻留在一个硬件安全模块中，该模块有自己的 CPU、安全存储、一个真正的随机数生成器和一种防止包篡改的机制。使用 Android Keystore 生成或导入密钥时，要使用此功能，必须将 **true** 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，请检查 **isInsideSecureHardware** 是否返回 **true**，以及系统是否不会引发 **StrongBoxUnavailableException**，如果 StrongBox Keymaster 对于与密钥相关的给定算法和密钥大小不可用，则会引发该异常。基于硬件的密钥库的功能描述可以在 AOSP 页面上找到。

Keymaster HAL 是硬件支持组件的接口-可信执行环境（TEE）或 Android KeyStore 使用的安全单元（SE）。这种硬件支持组件的一个例子是 Titan M。

5.3.2.5.2. 密钥认证

对于严重依赖 Android Keystore 进行关键业务操作的应用程序，例如通过加密算法进行多因素身份认证、在客户端安全存储敏感数据等。Android 提供了密钥认证功能，有助于分析通过 Android Keystore 管理的加密算法的安全性。从 Android 8.0（API 级别 26）开始，所有需要 Google 应用程序设备认证的新设备（Android 7.0 或更高版本）都必须进行密钥认证。此类设备使用由 Google 硬件认证根证书签名的认证密钥，并且可以通过密钥认证过程进行验证。

在密钥认证过程中，我们可以指定密钥对的别名，作为回应，我们可以获得一个证书链，用于验证该密钥对的属性。如果链的根证书是 Google 硬件认证根证书，并且进行了与硬件中密钥

对存储相关的检查，则可以确保设备支持硬件级密钥认证，并且密钥位于 Google 认为安全的硬件支持密钥库中。或者，如果认证链有任何其他根证书，那么 Google 不会对硬件的安全性提出任何声明。

虽然密钥认证过程可以直接在应用程序中实现，但出于安全原因，建议在服务器端实现。以下是密钥认证安全实施的高级指导原则：

- 服务器应通过使用 CSPRNG（加密安全随机数生成器）安全创建随机数来启动密钥认证过程，并将其作为质询发送给用户。
- 客户端应使用从服务器接收的质询调用 SetDetectionChallenge API，然后应使用 KeyStore.getCertificateChain 方法检索认证证书链。
- 认证响应应发送至服务器进行验证，并应执行以下检查以验证密钥认证响应：
  - 验证证书链，直至根，并执行证书健全性检查，如有效性、完整性和可信度。如果链中没有任何证书被吊销，请检查 Google 维护的证书吊销状态列表。
  - 检查根证书是否使用 Google 认证根密钥签名，这使得认证过程可信。
  - 提取证书链第一个元素中出现的认证证书扩展数据，并执行以下检查：
    - 验证认证质询是否具有与启动认证过程时在服务器上生成的值相同的值。
    - 验证密钥证明响应中的签名。
    - 验证 Keymaster 的安全级别，以确定设备是否具有安全密钥存储机制。Keymaster 是一种在安全环境中运行的软件，提供所有安全密钥库操作。安全级别为 Software、TrustedEnvironment 或 StrongBox。如果安全级别为 TrustedEnvironment 或 StrongBox，并且认证证书链包含一个用 Google 认证根密钥签名的根证书，则客户端支持硬件级密钥认证。
    - 验证客户端状态以确保完整的信任链-验证引导密钥、锁定引导加载程序和验证引导状态。
    - 此外，您可以验证密钥对的属性，例如目的、访问时间、身份认证要求等。

  注意，如果由于任何原因该进程失败，则表示密钥不在安全硬件中。这并不意味着密钥被泄露。

Android Keystore 认证响应的典型示例如下所示：

```
{
    "fmt": "android-key",
    "authData": "9569088f1ecee3232954035dbd10d7cae391305a2751b559bb8fd7cbb229
bd...",
    "attStmt": {
```

```
        "alg": -7,
        "sig": "304402202ca7a8cfb6299c4a073e7e022c57082a46c657e9e53...",
        "x5c": [
            "308202ca30820270a003020102020101300a06082a8648ce3d04030230818831
0b30090603550406130...",
            "308202783082021ea003020102020210013 00a06082a8648ce3d040302308198
310b300906035504061...",
            "3082028b30820232a003020102020900a2059ed10e435b57300a06082a8648ce
3d0403023081983 10b3..."
        ]
    }
}
```

在上面的 JSON 代码段中，密钥具有以下含义：

fmt：认证状态格式标识符
authData：它表示用于认证的验证器数据
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（服务器发送的质询）生成的，并使用 alg 签名算法通过凭据私钥进行签名，并且在服务器端使用第一个证书中的公钥对其进行验证。

有关实现指南的更多了解，请参阅 Google 示例代码。

从安全分析的角度来看，分析员可以对密钥认证的安全实施进行以下检查：

- 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、方法劫持等轻松绕过。
- 检查服务器在启动密钥认证时是否使用随机质询。因为如果不这样做，将导致不安全的实现，从而容易受到重放攻击。此外，应进行与挑战随机性相关的检查。
- 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中的证书执行基本检查，如完整性验证、信任验证、有效性等。

5.3.2.5.3. 安全密钥导入 Keystore

Android 9（API 级别 28）增加了将密钥安全导入 **AndroidKeystore** 的功能。 首先，**AndroidKeystore** 使用 **PURPOSE_WRAP_KEY** 生成一个密钥对，该密钥对也应使用认证证书进

行保护，该密钥对旨在保护导入到 **AndroidKeystore** 的密钥。 加密密钥生成为 SecureKeyWrapper 格式的 ASN.1 编码消息，其中还包含对允许使用导入密钥的方式的描述。 然后，密钥在属于生成密钥的特定设备的 **AndroidKeystore** 硬件内解密，因此它们永远不会以明文形式出现在设备的主机内存中。

Java 示例：

```
KeyDescription ::= SEQUENCE {
    keyFormat INTEGER,
    authorizationList AuthorizationList
}

SecureKeyWrapper ::= SEQUENCE {
    wrapperFormatVersion INTEGER,
    encryptedTransportKey OCTET_STRING,
    initializationVector OCTET_STRING,
    keyDescription KeyDescription,
    secureKey OCTET_STRING,
    tag OCTET_STRING
}
```

上面的代码表示在以 SecureKeyWrapper 格式生成加密密钥时要设置的不同参数。有关更多详细信息，请查看 Android 文档有关 WrappedKeyEntry 部分。

定义 KeyDescription AuthorizationList 时，以下参数会影响加密密钥的安全性：

- algorithm 参数指定密钥使用的加密算法
- keySize 参数指定密钥的大小（以位为单位），以密钥算法的通常方式计算
- digest 参数指定可与密钥一起使用以执行签名和验证操作的摘要算法

5.3.2.5.4. 早期 Keystore 实现

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 KeyStore (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 KeyStore 名称 (BouncyCastle KeyStore)，“BC”是提供者(BouncyCastle)。您还可以使用 SpongyCastle 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。

5.3.2.5.5. 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)。如果某些东西是第一次被导入到 KeyChain，用户将被提示设置一个锁屏 pin 或密码，以保护凭证存储。注意，这个 KeyChain 是整个系统的，每个应用程序都可以访问存储在这个 KeyChain 中的内容。

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，可以通过使用 KeyChain 类的几个 API 调用来保护数据。完成以下步骤：

- 请确保应用程序正在使用 Android KeyStore 和加密机制来安全地将加密信息存储在设备上。查找 `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 使用密码加密后存储到磁盘。确保密码是由用户提供的，而不是硬编码的。

5.3.2.5.6. 存储加密密钥：技术

为了减少未经授权使用 Android 设备上的密钥，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范围。一旦授权完成，就不能更改。

存储密钥-从最安全到最不安全：

- 密钥存储在硬件支持的 Android KeyStore 中
- 所有密钥都存储在服务器上，并在强身份认证后可用
- 主密钥存储在服务器上，用于加密存储在 Android SharedPreferences 中的其他密钥
- 每次都从用户提供的强大的具有足够长度和盐的密码短语中派生密钥
- 密钥存储在软件实现的 Android KeyStore 中
- 主密钥存储在软件实现的 Android KeyStore 中，用于加密存储在 SharedPreferences 中的其他密钥
- [不推荐] 所有密钥都存储在 SharedPreferences 中
- [不推荐] 源代码中硬编码的加密密钥
- [不推荐] 可预测的混淆功能或基于稳定属性的密钥衍生功能

- [不推荐] 将生成的密钥存储在公共场所（如`/sdcard/`）

5.3.2.5.6.1 使用硬件支持的 Android KeyStore 来存储密钥

如果设备运行的是 Android 7.0（API 级别 24）及以上版本，并有可用的硬件组件（可信执行环境（TEE）或安全元件（SE）），你可以使用硬件支持的 Android KeyStore。你甚至可以通过使用为安全执行密钥认证提供的准则来验证密钥是否有硬件支持。如果没有硬件组件和/或需要对安卓 6.0（API 级别 23）及以下版本的支持，那么你可能想把你的密钥存储在远程服务器上，并在认证后使其可用。

5.3.2.5.6.2 在服务器上存储密钥

安全地将密钥存储在密钥管理服务器上是可能的，但是应用程序需要在线来解密数据。这可能是某些移动应用程序使用场景的限制，应该仔细考虑，因为这成为应用程序架构的一部分，并可能高度影响可用性。

5.3.2.5.6.3 从用户输入获得密钥

从用户提供的口令推导出一个密钥是一个常见的解决方案（取决于你使用的 Android API 级别），但它也影响了可用性，可能会扩大攻击面，并可能引入额外的弱点。。

每次应用程序需要执行加密操作时，都需要用户的口令。要么每次都提示用户，这不是一个理想的用户体验，要么只要用户被认证，密码就会被保存在内存中。将口令保留在内存中并不是一个最佳做法，因为任何加密材料都必须在使用时保留在内存中。正如在 "清理密钥材料 "中所解释的那样，将密钥清除通常是一项非常具有挑战性的任务。

此外，考虑到从口令获得的密钥有其自身的弱点。例如，密码或口令可能会被用户重复使用或容易被猜到。更多信息请参考《测试加密》一章。

5.3.2.5.6.4 清理密钥材料

一旦不需要了，就应该把密钥材料从内存中清除掉。在具有垃圾收集器（Java）和不可变字符串（Swift、Objective-C、Kotlin）的语言中，真正地清理机密数据有一定的限制。Java 加密架构参考指南建议使用 char[]而不是 String 来存储敏感数据，并在使用后将数组清空。

请注意，有些密码并没有正确地清理他们的字节数组。例如，BouncyCastle 中的 AES 密码并不总是清理其最新的工作密钥，在内存中留下一些字节数组的副本。其次，基于 BigInteger 的密钥（如私钥）不能从堆中删除，也不能在没有额外操作的情况下清零。清除字节数组可以通过编写一个实现 Destroyable 的包装器来实现。

5.3.2.5.6.5 使用 Android KeyStore API 存储密钥

更为用户友好和推荐的方法是使用 Android KeyStore API 系统（本身或通过 KeyChain）来存储密钥材料。如果可能，应使用硬件支持的存储。否则，它应该回退到 Android Keystore 的软件实现。然而，请注意，Android 的各个版本中，AndroidKeyStore API 都发生了显著的变化。在早期版本中，AndroidKeyStore API 仅支持存储公钥/私钥对（例如 RSA）。自 Android 6.0（API 级别 23）以来，才添加了对称密钥支持。因此，开发人员需要处理不同的 Android API 级别，以安全地存储对称密钥。

5.3.2.5.6.6 通过用其他密钥加密来存储密钥

为了在运行 Android 5.1（API 级别 22）或更低版本的设备上安全地存储对称密钥，我们需要生成一个公钥/私钥对。我们使用公钥加密对称密钥，并将私钥存储在 AndroidKeyStore 中。加密的对称密钥可以使用 base64 编码并存储在 SharedReferences 中。每当我们需要对称密钥时，应用程序就会从 AndroidKeyStore 检索私钥并解密对称密钥。

信封加密，或称密钥包装，是一种类似的方法，使用对称加密来封装密钥材料。数据加密密钥（DEK）可以与安全存储的密钥加密密钥（KEK）进行加密。加密的 DEK 可以存储在 SharedPreferences 中或写入文件中。当需要时，应用程序读取 KEK，然后对 DEK 进行解密。请参考 OWASP 加密存储备忘录，了解更多关于加密密钥的信息。

另外，作为这种方法的说明，请参考 androidx.security.crypto 包中的 EncryptedSharedPreferences。

5.3.2.5.6.7 存储密钥的不安全选择

将加密密钥存储在 Android 的 SharedReferences 是一种不太安全的方式。当使用 SharedReferences 时，文件只能由创建它的应用程序读取。然而，在 root 过的设备上，任何具有 root 访问权限的其他应用程序都可以简单地读取其他应用程序的 SharedPreference 文

件。AndroidKeyStore 的情况并非如此。由于 AndroidKeyStore 访问是在内核级别管理的，因此在不清除或销毁密钥的情况下绕过它需要相当多的工作和技能。

最后三个选项是在源代码中使用硬编码的加密密钥，有一个可预测的混淆功能或基于稳定属性的密钥推导功能，以及将生成的密钥存储在公共场所，如/sdcard/。硬编码的加密密钥是一个问题，因为这意味着应用程序的每个实例都使用相同的加密密钥。攻击者可以逆向工程应用程序的本地副本，以提取加密密钥，并使用该密钥来解密任何设备上由应用程序加密的任何数据。

接下来，当您具有基于其他应用程序可以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，公开存储加密密钥也是非常不鼓励的，因为其他应用程序可以有权读取公共分区并窃取密钥。

5.3.2.5.7. 第三方库

有几个不同的开源库提供特定于 Android 平台的加密功能。

- Java AES Crypto - 一个用于加密和解密字符串的简单 Android 类。
- SQL Cipher - SQLCipher 是 SQLite 的开源扩展，它为数据库文件提供透明的 256 位 AES 加密。
- Secure Preferences - Android Shared Preference 包装器，加密 Shared Preferences 的密钥和值。
- Themis - 一个跨平台的高级加密库，在许多平台上提供相同的 API，在认证、存储、信息传递等过程中保护数据安全。

请记住，只要密钥未存储在 KeyStore 中，就可以在 root 设备上轻松检索密钥，然后解密您试图保护的内容。

5.3.2.6. 日志

在移动设备上创建日志文件有许多正当的理由，例如跟踪崩溃、错误和使用统计。当应用程序脱机时，日志文件可以存储在本地，并在应用程序联机后发送到端点。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，也可能破坏用户机密性。可以通过多种方式创建日志文件。以下列表包括两个适用于 Android 的类：

- Log 类
- Logger 类

5.3.2.7. 备份

Android 为用户提供了自动备份功能。备份通常包括所有已安装应用程序的数据和设置副本。考虑到其多样的生态系统，Android 支持许多备份选项：

- 原生 Android 有内置的 USB 备份功能。启用 USB 调试后，您可以使用 adb backup 命令创建完整数据备份和应用程序数据目录的备份。
- Google 提供了一个“备份我的数据（Back Up My Data）”功能，将所有应用程序数据备份到 Google 的服务器。
- 应用程序开发人员可以使用两个备份 API：
  - 键/值备份（备份 API 或 Android 备份服务）上传到 Android 备份服务云。
  - 应用程序自动备份：在 Android 6.0（API 级别 23）及以上版本中，Google 增加了“应用程序自动备份功能”。此功能最多可自动将 25MB 的应用程序数据与用户的 Google Drive 帐户同步。
- 原始设备制造商可能会提供其他选项。例如，HTC 设备有一个“HTC 备份”选项，该选项在激活时执行每日云备份。

应用程序必须小心确保敏感用户数据不会存储在这些备份中，因为这可能会让攻击者提取这些数据。

5.3.2.8. 进程内存

Android 上的所有应用程序都使用内存来执行正常的计算操作，就像任何普通的现代计算机一样。因此，有时会在进程内存中执行敏感操作也就不足为奇了。因此，重要的是，一旦处理了相关的敏感数据，就应该尽快将其从进程内存中处理掉。

应用程序内存的调查可以从内存转储中完成，也可以通过调试器实时分析内存。

这将在“测试内存中的敏感数据”一节中进一步解释。

5.3.3. 测试本地存储中敏感数据 (MSTG-STORAGE-1 和 MSTG-STORAGE-2)

5.3.3.1. 概述

这个测试用例的重点是识别应用程序存储的潜在敏感数据，并验证其是否安全存储。应进行以下检查：

- 分析源代码中的数据存储。
- 确保触发应用程序中所有可能的功能（例如，单击任何可能的地方），以确保数据生成。
- 检查所有应用程序生成和修改的文件，确保存储方法足够安全。
    - 这包括 SharedPreferences、SQL 数据库、Realm 数据库、内部存储、外部存储等。

一般来说，本地存储在设备上的敏感数据至少应始终加密，用于加密方法的任何密钥应安全存储在 Android Keystore 中。这些文件也应该存储在应用程序沙箱中。如果应用程序可以实现，则应将敏感数据存储在设备外，或者更好的是，完全不存储。

5.3.3.2. 静态分析

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 或 MODE_WORLD_WRITABLE: 您应该避免对文件使用 MODE_WORLD_WRITEABLE 和 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:

- SharedPreferences 类(存储键配对)。
- FileOutPutStream 类(使用内部或外部存储)。
- getExternal* 函数(使用外部存储)。
- getWritableDatabase 函数(返回一个 SQLiteDatabase 用于写)。
- getReadableDatabase 函数(返回一个 SQLiteDatabase 用于读)。
- getCacheDir 和 getExternalCacheDirs 函数(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 利用 Android 自带特性(如 Android KeyStore)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:

```
this.db = localUserSecretStore.getWritableDatabase("SuperPassword123");
```

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增加难度的尝试，但也不是无懈可击。考虑以下代码:

Java 示例:

```
//A more complicated effort to store the XOR'ed halves of a key (instead of t
he key itself)
private static final String[] myCompositeKey = new String[]{
  "oNQavjbaNNSgEqoCkT9Em4imeQQ=","3o8eFOX4ri/F8fgHgiy/BS47"
};
```

Kotlin 示例:

```
private val myCompositeKey = arrayOf<String>("oNQavjbaNNSgEqoCkT9Em4imeQQ=",
"3o8eFOX4ri/F8fgHgiy/BS47")
```

解码原始密钥的算法可能是这样的:

Java 示例:

```
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);
}
```

Kotlin 示例：

```
fun useXorStringHiding(myHiddenMessage:String) {
  val xorParts0 = Base64.decode(myCompositeKey[0], 0)
  val xorParts1 = Base64.decode(myCompositeKey[1], 0)
  val xorKey = ByteArray(xorParts0.size)
  for (i in xorParts1.indices)
  {
    xorKey[i] = (xorParts0[i] xor xorParts1[i]).toByte()
  }
  HidingUtil.doHiding(myHiddenMessage.toByteArray(), 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 示例：

```
buildTypes {
  debug {
    minifyEnabled true
    buildConfigField "String", "hiddenPassword", "\"${hiddenPassword}\""
  }
}
```

5.3.3.3. 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：

- 检查内部和外部本地存储中是否有应用程序创建的包含敏感数据的任何文件。
- 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。
    - 确定 SQLite 数据库是否加密。如果加密了，请确定数据库密码是如何生成和存储的，以及是否按照 Keystore 概述的“存储密钥”部分中所述对其进行了充分保护。
- 检查存储为 XML 文件(在/data/data/<package-name>/shared_prefs 中)的 Shared Preferences 是否有敏感信息。默认情况下，Shared Preferences 是不安全且未加密的。一些应用可能会选择使用 secure-preferences 加密存储在 Shared Preferences 中的值。
- 检查/data/data/<package-name>中的文件的权限。只有安装应用时创建的用户和组(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。
- 检查任何 Firebase 实时数据库的使用情况，并尝试通过进行以下网络调用来确定它们是否配置错误：

```
- https://_firebaseProjectName_.firebaseio.com/.json
```

- 确定 Realm 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 Realm 浏览器检查 Realm 数据库。

5.3.4. 测试本地存储的输入验证(MSTG-PLATFORM-2)

5.3.4.1. 概览

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实现输入验证。

请注意: 在 root 过的设备上也应检查类似的私有可访问数据。

5.3.4.2. 静态分析

5.3.4.2.1. 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（例如：没有额外的漏洞可以用来打包以接管控制流）。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。

5.3.4.2.2. 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。

5.3.5. 测试日志中敏感数据 (MSTG-STORAGE-3)

5.3.5.1. 概述

这个测试用例的重点是识别系统和应用程序日志中的任何敏感应用程序数据。应进行以下检查:

- 分析记录日志相关代码的源代码。
- 检查应用程序数据目录中的日志文件。
- 收集系统消息和日志，并分析任何敏感数据。

作为避免潜在敏感应用程序数据泄漏的一般建议，应从生产版本中删除日志记录语句，除非应用程序认为有必要或明确确定为安全的，例如由于安全审计。

5.3.5.2. 静态分析

应用程序通常使用 Log 类和 Logger 类来创建日志。为了发现这一点，您应该审计应用程序的源代码以查找任何此类日志类。通常可以通过搜索以下关键字找到这些关键字:

- 函数和类，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 ProGuard（包含在 Android Studio 中）等工具。 要确定 android.util.Log 类中的所有日志记录功能是否已被删除，请检查 ProGuard 配置文件 (proguard-rules.pro) 中的以下选项（根据删除日志记录代码的示例和这篇关于在 Android Studio 中启用 ProGuard 的文章）：

```
-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：

Java 示例：

```
Log.v("Private key tag", "Private key [byte format]: " + key);
```

Kotlin 示例：

```
Log.v("Private key tag", "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：

Java 示例：

```
Log.v("Private key tag", new StringBuilder("Private key [byte format]: ").app
end(key.toString()).toString());
```

Kotlin 示例：

```
Log.v("Private key tag", StringBuilder("Private key [byte format]: ").append
(key).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。

5.3.5.3. 动态分析

至少使用一次移动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，而不是使用适当的日志记录类。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”一节中介绍的 Logcat。

请记住，您可以通过如下方式过滤 Logcat 输出，以设置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 参数直接指定。

如果您想要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 的正则表达式参数-e <expr>, --regex=<expr>)。

5.3.6. 判断敏感数据是否发送给第三方(MSTG-STORAGE-4)

5.3.6.1. 概述

敏感信息可能通过以下几种方式泄露给第三方，包括但不限于：

5.3.6.2.**应用**内嵌第三方服务

这些服务提供的功能包括跟踪服务，以监控用户在使用应用程序时的行为、销售横幅广告或改善用户体验。

缺点是开发人员通常不知道通过第三方库执行的代码的细节。因此，不应向服务发送超过必要的信息，也不应披露任何敏感信息。

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
- 使用完整的 SDK

5.3.6.3.**应用**通知

重要的是要了解通知永远不应被视为私有的。当 Android 系统处理通知时，它会在系统范围内广播，并且任何使用 NotificationListenerService 运行的应用程序都可以侦听这些通知以完整接收它们，并且可以根据需要处理它们。

有许多已知的恶意软件样本，例如 Joker 和 Alien，它们滥用 `NotificationListenerService` 来监听设备上的通知，然后将它们发送到攻击者控制的 C2 基础设施。通常这样做是为了侦听在设备上显示为通知的双因素身份认证 (2FA) 代码，然后将其发送给攻击者。对于用户而言，更安全的替代方法是使用不生成通知的 2FA 应用程序。

此外，Google Play 商店中有许多提供通知日志记录的应用程序，这些应用程序基本上可以在本地记录 Android 系统上的任何通知。这凸显了通知在 Android 上绝不是私密的，并且可以由设备上的任何其他应用程序访问。

出于这个原因，应检查所有通知的使用情况，以获取可能被恶意应用程序使用的机密或高风险信息。

5.3.6.4. 静态分析

5.3.6.4.1.应用内嵌第三方服务

要确定第三方库提供的 API 调用和函数是否按照最佳实践使用，请查看其源代码、请求的权限并检查是否存在任何已知漏洞（请参阅“检查第三方库中的弱点 (MSTG-CODE-5)”）。

发送到第三方服务的所有数据都应匿名化，以防止泄露 PII（个人身份信息），使第三方能够识别用户帐户。 不应将其他数据（例如可以映射到用户帐户或会话的 ID）发送给第三方。

5.3.6.4.2.应用通知

搜索 NotificationManager 类的任何用法，这可能表明某种形式的通知管理。 如果正在使用该类，下一步将是了解应用程序如何生成通知以及最终显示哪些数据。

5.3.6.5. 动态分析

5.3.6.5.1.应用内嵌第三方服务

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。

5.3.6.5.2.应用通知

运行应用程序并开始跟踪对与通知创建相关的函数的所有调用，例如 `NotificationCompat.Builder` 中的 `setContentTitle` 或 `setContentText`。 最后观察跟踪并评估它是否包含其他应用程序可能窃听的任何敏感信息。

5.3.7. 判断文本输入字段是否禁用键盘缓存(MSTG-STORAGE-5)

5.3.7.1. 概述

当用户在输入字段中输入时，软件会自动建议数据。 此功能对于消息传递应用程序非常有用。但是，当用户选择采用此类信息的输入字段时，键盘缓存可能会泄露敏感信息。

5.3.7.2. 静态分析

在 activity 的布局定义中，您可以定义具有 XML 属性的 TextViews。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：

5.3.7.3. 动态分析

启动应用程序，点击接收敏感数据的输入字段。如果有建议内容，则这些字段还没有禁用键盘缓存。

5.3.8. 确定敏感存储数据是否已通过 IPC 机制公开(MSTG-STORAGE-6)

5.3.8.1. 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。

5.3.8.2. 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，如果一个 <intent-filter>已为标签定义，标签将被自动设置为“true”。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，将属性设置为“true”，并定义正确的读写权限。
- 确定数据是否被权限标签保护(android:permission)。权限标签限制向其他应用程序暴露。
- 确定 android:protectionLevel 属性值是否设置为 signature。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，其他应用程序必须在

Manifest 文件中声明相应的<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>元素限制访问。

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

为了避免应用程序内的 SQL 注入攻击，请使用参数化查询方法，例如 query、update 和 delete。确保正确清理所有方法参数；例如，如果 selection 参数由拼接的用户输入组成，则它可能导致 SQL 注入。

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(query、update 和 delete)来防止 SQL 注入。如果是的话，确保他们所有的参数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者的示例。

5.3.8.2.1. 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。

5.3.8.2.2. 检查源代码

检查 DBContentProvider.java 文件中的 query 函数，以确定是否有任何敏感信息被泄露：

Java 示例：

```
public Cursor query(final Uri uri, final String[] array, final String s, fina
l 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);
}
```

Kotlin 示例：

```
fun query(uri: Uri?, array: Array<String?>?, s: String?, array2: Array<Strin
g?>?, s2: String?): Cursor {
        val match: Int = this.sUriMatcher.match(uri)
        val sqLiteQueryBuilder = SQLiteQueryBuilder()
        if (match >= 100 && match < 200) {
            sqLiteQueryBuilder.tables = "Passwords"
        } else if (match >= 200) {
            sqLiteQueryBuilder.tables = "Key"
        }
        return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), arra
y, s, array2, null as String?, null as String?, 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 Passwords/。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。

5.3.8.3. 动态分析

5.3.8.3.1. 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，暴露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 从一个脆弱的内容提供者插入，更新和删除记录：

- 插入记录

```
dz> run app.provider.insert content://com.vulnerable.im/messages
                --string date 1331763850325
                --string type 0
                --integer _id 7
```

- 更新记录

```
dz> run app.provider.update content://settings/secure
                --selection "name=?"
                --selection-args assisted_gps_enabled
                --integer value 0
```

- 删除记录

```
dz> run app.provider.delete content://settings/secure
                --selection "name=?"
                --selection-args my_setting
```

5.3.8.3.2. 内容提供程序中的 SQL 注入

Android 平台提供 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.query 通过操作传递给内容提供程序的预测和选择字段来测试 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程序的数据。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 |
| thisismypassword | 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程序：

```
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/
```

5.3.8.3.3. 基于文件系统的内容提供程序

内容提供程序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程序读取和下载文件。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/data/com.vulnerable.app/database.db /home/user/databa
se.db
Written 24488 bytes
```

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程序：

```
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.Cred
entialProvider/credentials
Row: 0 id=1, username=admin, password=StrongPwd
Row: 1 id=2, username=test, password=test
...
```

5.3.9. 检查用户界面是否泄露敏感数据（MSTG-STORAGE-7）

5.3.9.1. 概述

输入敏感信息是使用许多应用程序的必要部分，例如，在注册帐户或付款时。这些数据可能是财务信息，如信用卡数据或用户帐户密码。如果应用程序在键入数据时没有正确屏蔽数据，数据可能会被暴露。

为了防止泄露和减轻风险，如肩窥，您应该验证，除非明确要求（例如输入密码），否则没有敏感数据通过用户界面泄露。对于需要显示的数据，应适当屏蔽，通常显示星号或点，而不是明文。

仔细检查所有显示此类信息或将其作为输入的 UI 组件。搜索任何敏感信息的痕迹，并评估其是否应被掩盖或完全删除。

5.3.9.2. 静态分析

5.3.9.2.1. 文本字段

为了确保应用程序屏蔽了敏感的用户输入，请在 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。

5.3.9.2.2.*应用*通知

在静态评估应用程序时，建议搜索 `NotificationManager` 类的任何用法，这可能表示某种形式的通知管理。如果正在使用该类，下一步将是了解应用程序如何生成通知。

这些代码位置可以输入下面的动态分析部分，提供了可以在应用程序中的何处动态生成通知的想法。

5.3.9.3. 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可能泄露信息的组件。

5.3.9.3.1. 文本字段

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。

5.3.9.3.2.*应用*通知

要确定通知的使用情况，请在整个应用程序及其所有可用功能中查找触发任何通知的方法。考虑您可能需要在应用程序之外执行操作以触发某些通知。

在运行应用程序时，您可能希望开始跟踪对与通知创建相关的函数的所有调用，例如来自 `NotificationCompat.Builder` 的 `setContentTitle` 或 `setContentText`。最后观察跟踪并评估其是否包含任何敏感信息。

5.3.10. 测试备份中敏感数据（MSTG-STORAGE-8)

5.3.10.1. 概述

此测试用例的重点是确保备份不会存储特定于应用程序的敏感数据。应进行以下检查：

- 检查 AndroidManifest.xml 以获取相关的备份属性。

- 尝试备份应用程序并检查备份是否有敏感数据。

5.3.10.2. 静态分析

5.3.10.2.1. 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。

要防止应用程序数据备份，请将 `android:allowBackup` 属性设置为 **false**。当该属性不可用时， allowBackup 默认启用，必须手动禁用备份。

请注意：如果设备被加密了，备份文件也会被加密。

检查 `AndroidManifest.xml` 文件中的以下属性：

```
android:allowBackup="true"
```

如果该属性值为 **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“测试本地存储中敏感数据”)。

5.3.10.2.2. 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 Google 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 manifest 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。当实现备份代理时，您可以使用 `android:fullBackupOnly` 属性来激活自

动备份，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。

`android:fullBackupOnly`

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 Google 账户云盘存储 25 MB。只存储最近的备份;之前的备份会删除。

- **Key/Value Backup:** 要启用键/值备份，您必须在 manifest 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。

5.3.10.3. 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“Now unlock your device and confirm the backup operation（现在解锁您的设备并确认备份操作）”，然后在 Android 手机上应该要求您输入密码。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>"
```

当您的设备具有 1.0.31 之前的 adb 版本时，就会出现问题。 如果是这种情况，您还必须在主机上使用 1.0.31 的 adb 版本。 1.0.32 之后的 adb 版本打破了向后兼容性。

通过选择“备份我的数据（*Back up my data*）”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
```

如果发生错误 `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 无限强度策略文件，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]
```

```
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
```

5.3.11. 在自动生成的截图中查找敏感信息(MSTG-STORAGE-9)

5.3.11.1. 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，因此他们在应用程序置于后台时引入了屏幕截图保存功能。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 root）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。

5.3.11.2. 静态分析

当一个 Android 应用程序进入后台时，当前 activity 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：

Java 示例：

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,
                WindowManager.LayoutParams.FLAG_SECURE);

setContentView(R.layout.activity_main);
```

Kotlin 示例：

```
window.setFlags(WindowManager.LayoutParams.FLAG_SECURE,
                WindowManager.LayoutParams.FLAG_SECURE)

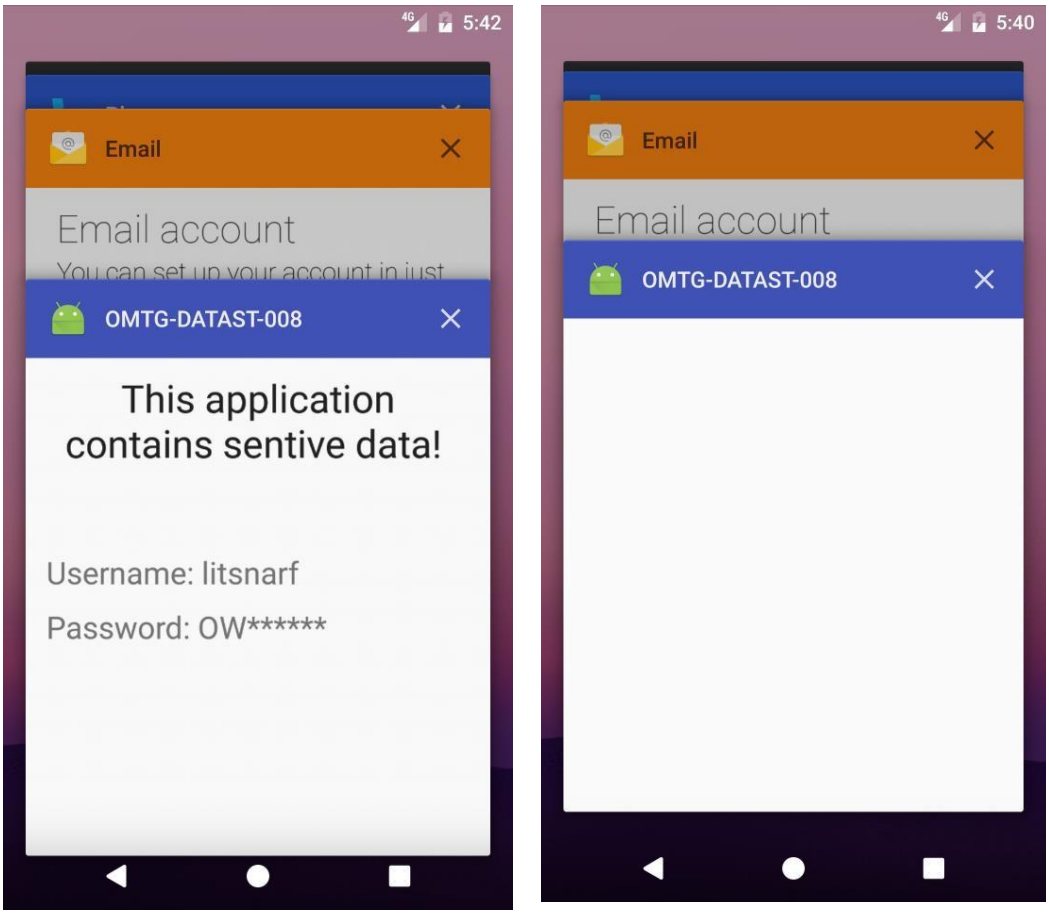
setContentView(R.layout.activity_main)
```

如果没有设置该选项，则应用程序存在屏幕捕获漏洞。

5.3.11.3. 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 按钮将应用程序置于到后台，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;如果还没有设置属性(左图)，则会显示 activity 信息：

FLAG_SECURE not set　　FLAG_SECURE set



在支持基于文件的加密 (FBE) 的设备上，快照存储在 `/data/system_ce/<USER_ID>/<IMAGE_FOLDER_NAME>` 文件夹中。`<IMAGE_FOLDER_NAME>`取决于供应商，但最常见的名称是 `snapshots` 和 `recent_images`。 如果设备不支持 FBE，则使用 `/data/system/<IMAGE_FOLDER_NAME>` 文件夹。

访问这些文件夹和快照需要 root 权限

5.3.12. 测试内存中的敏感数据(MSTG-STORAGE-10)

5.3.12.1. 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泄露。

首先识别存储在内存中的敏感信息。敏感资产可能在某个时候被载入了内存。目的是验证此信息是否尽可能简短地公开。

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，除非您能在另一个关联内容中识别密钥的值。

因此，您最好从静态分析开始。

5.3.12.2. 静态分析

要了解可能的数据泄露来源，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 客户端、XML 解析器中。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以及体系结构在系统中的作用将有助于识别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台服务器接收数据，并将其传输到另一台服务器，而无需任何处理。这些数据可以以加密格式处理，这可以防止暴露在内存中。

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。确保他们使用此访问权限来用虚拟数据（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。

使用非基本可变类型(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始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：销毁一个密钥，如下所示，并没有真正从内存中删除密钥：

Java 示例：

```
SecretKey secretKey = new SecretKeySpec("key".getBytes(), "AES");
secretKey.destroy();
```

Kotlin 示例：

```
val secretKey: SecretKey = SecretKeySpec("key".toByteArray(), "AES")
secretKey.destroy()
```

从 secretKey.getEncoded 覆盖支持字节数组也不会删除密钥；基于 SecretKeySpec 的密钥返回支持字节数组的副本。有关从内存中删除 SecretKey 的正确方法，请参阅以下部分。

RSA 密钥对基于 BigInteger 类型，因此在 AndroidKeyStore 之外首次使用后驻留在内存中。一些密码（例如 BouncyCastle 中的 AES 密码）没有正确清理它们的字节数组。

用户提供的数据（凭证、社会保险号、信用卡信息等）是另一种可能暴露在内存中的数据。无论您是否将其标记为密码字段，`EditText` 都会通过 `Editable` 界面将内容传递给应用程序。如果您的应用程序不提供 `Editable.Factory`，则用户提供的数据可能会在内存中暴露比必要更长的时间。默认的 `Editable` 实现，`SpannableStringBuilder`，会导致与 Java 的 `StringBuilder` 和 `StringBuffer` 相同的问题（如上所述）。

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。
    - 避免非基本数据类型(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。

以下部分描述了内存中数据泄漏的陷阱以及避免这些陷阱的最佳实践。

不要使用不可变的结构(如 `String` 和 `BigInteger`)来表示机密信息。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存在时间。

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基本数据类型中。

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:

Java 示例:

```
byte[] secret = null;
try{
    //get or generate the secret, do work with it, make sure you make no loca
l copies
} finally {
    if (null != secret) {
        Arrays.fill(secret, (byte) 0);
    }
}
```

Kotlin 示例:

```
val secret: ByteArray? = null
try {
     //get or generate the secret, do work with it, make sure you make no loc
al copies
} finally {
    if (null != secret) {
        Arrays.fill(secret, 0.toByte())
    }
}
```

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 DEX 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劫持目标(详见“Android 系统上的篡改和逆向工程”章节)。

上述示例的最后一个问题是内容仅被零覆盖。 您应该尝试使用来自非关键对象的随机数据或内容覆盖关键对象。 这将使得构建能够在其管理的基础上识别敏感数据的扫描器变得非常困难。

下面是前一个例子的改进版本：

Java 示例：

```
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 loca
l 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();
    }
}
```

Kotlin 示例：

```
val nonSecret: ByteArray = somePublicString.getBytes("ISO-8859-1")
val secret: ByteArray? = null
try {
     //get or generate the secret, do work with it, make sure you make no loc
al copies
} finally {
    if (null != secret) {
        for (i in secret.indices) {
            secret[i] = nonSecret[i % nonSecret.size]
        }
```

```
        val out = 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。

Java 示例：

```
  public class SecureSecretKey implements javax.crypto.SecretKey, Destroyable
 {
      private byte[] key;
      private final String algorithm;

      /** Constructs SecureSecretKey instance out of a copy of the provided k
ey bytes.
        * The caller is responsible of clearing the key array provided as inp
ut.
        * The internal copy of the key can be cleared by calling the destroy
() method.
        */
      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";
      }

      /** Returns a copy of the key.
```

```
         * Make sure to clear the returned byte array when no longer needed.
         */
        public byte[] getEncoded() {
            if(null == key){
                throw new NullPointerException();
            }

            return key.clone();
        }

        /** Overwrites the key with dummy data to ensure this copy is no longer
 present in memory.*/
        public void destroy() {
            if (isDestroyed()) {
                return;
            }

            byte[] nonSecret = new String("RuntimeException").getBytes("ISO-885
9-1");
            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;
        }
    }
```

Kotlin 示例:

```
class SecureSecretKey(key: ByteArray, algorithm: String) : SecretKey, Destroy
able {
    private var key: ByteArray?
    private val algorithm: String
    override fun getAlgorithm(): String {
        return algorithm
    }

    override fun getFormat(): String {
        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.
     */
    override fun getEncoded(): ByteArray {
        if (null == key) {
            throw 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. */
    override fun destroy() {
        if (isDestroyed) {
            return
        }
        val nonSecret: ByteArray = String("RuntimeException").toByteArray(cha
rset("ISO-8859-1"))
        for (i in key!!.indices) {
            key!![i] = nonSecret[i % nonSecret.size]
        }
        val out = FileOutputStream("/dev/null")
        out.write(key)
        out.flush()
        out.close()
        key = null
        System.gc()
    }

    override fun isDestroyed(): Boolean {
        return key == null
    }

    /** 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。
     */
    init {
        this.key = key.clone()
        this.algorithm = algorithm
    }
}
```

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 Editable 实现示例。请注意，如果您提供您的 factory，您将能够安全地处理所有由 editText.getText 产生的副本。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的使用。因此，您应该仅将此方法用于半机密信息。

在所有情况下，确保在用户注销应用程序时清除内存中的敏感数据。最后，确保在触发 Activity 或 Fragment 的 onPause 事件时清除高度敏感的信息。

请注意，这可能意味着每次应用程序恢复时，用户都必须重新进行身份认证。

5.3.12.3. 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。

有多种方法可以分析进程的内存，例如通过调试器/动态检测进行实时分析，以及分析一个或多个内存转储。

5.3.12.3.1. 获取并分析内存转

储

无论您使用的是 root 过设备还是未 root 设备，您都可以使用 objection 和 Fridump 转储应用程序的进程内存。你可以在“Android 系统上的篡改和逆向工程”一章的“内存转储”一节中找到关于这个过程的详细解释。

内存转储后（例如，转储到名为“memory”的文件），根据您要查找的数据的性质，您需要一组不同的工具来处理和分析内存转储。例如，如果您关注字符串，那么执行命令 **strings** 或 **rabin2 -zz** 来提取这些字符串就足够了。

```
# 使用 strings
$ strings memory > strings.txt

# 使用 rabin2
$ rabin2 -ZZ memory > strings.txt
```

使用你喜爱的编辑器打开 strings.txt 并挖掘它以识别敏感信息。

但是，如果您想检查其他类型的数据，您应该更愿意使用 radare2 及其搜索功能。 有关更多信息和选项列表，请参阅 radare2 对搜索命令 (/?) 的帮助。 以下仅显示其中的一部分：

```
$ r2 <name_of_your_dump_file>

[0x00000000]> /?
Usage: /[!bf] [arg]  Search stuff (see 'e??search' for options)
|Use io.va for searching in non virtual addressing spaces
| / foo\x00                     search for string 'foo\0'
| /c[ar]                        search for crypto materials
| /e /E.F/i                     match regular expression
| /i foo                        search for string 'foo' ignoring case
| /m[?][ebm] magicfile          search for magic, filesystems or binary header
s
| /v[1248] value                look for an `cfg.bigendian` 32bit value
| /w foo                        search for wide string 'f\0o\0o\0'
| /x ff0033                     search for hex string
| /z min max                    search for strings of given size
...
```

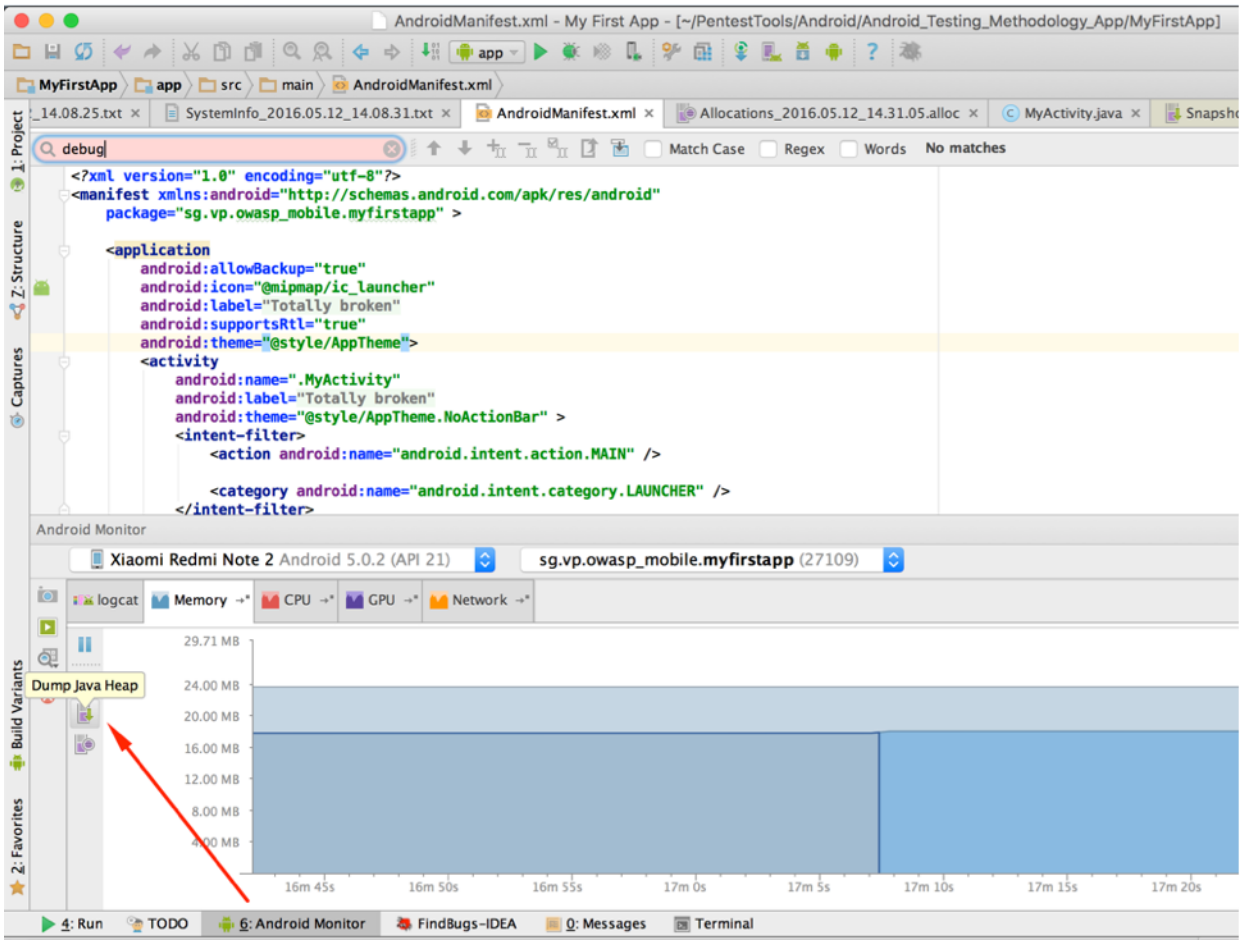
5.3.12.3.2. 运行时内存分析

您可以选择使用 r2frida，而不是将内存转储到你的主机。 有了它，您可以在应用程序运行时分析和检查应用程序的内存。 例如，您可以从 r2frida 运行之前的搜索命令，并在内存中搜索字符串、十六进制值等。这样做时，在使用 `r2 frida://usb//<name_of_your_app>` 开始会话之后请记住在开始搜索命令（以及任何其他 r2frida 特定命令）前加上反斜杠。

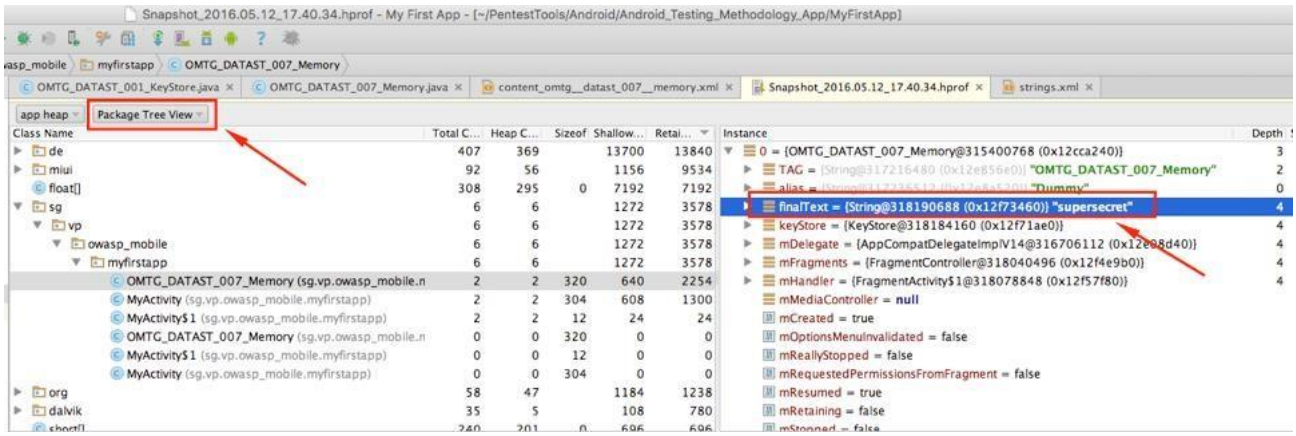
有关更多信息、选项和方法，请参阅“Android 上的篡改和逆向工程”一章中的“内存搜索”部分。

5.3.12.3.3. 显式转储和分析 Java 堆

对于基本的分析，您可以使用 Android Studio 的内置工具。它们在 *Android Monitor* 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“*Dump Java Heap*（转储 Java 堆）”。这将在 *captures* 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 View（包树状视图）显示.hprof 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：*Histogram*（柱状图）提供了给定类型捕获的对象数量的估计，*Thread Overview*（线程概览）显示了进程的线程和堆栈块。*Dominator Tree*（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（对象查询语言工作室） 是一个 MAT 功能，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 `String` 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
```

或者

```
SELECT object.toString() FROM java.lang.String object
```

SQL 也支持基本数据类型，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 数据类型只是基本数据类型的包装器。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\.
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，属性是列。如果您想找到所有有"password"属性的对象，您可以像这样做：

```
SELECT password FROM ".*" WHERE (null != password)
```

在您的分析中，请查找：

- 指示属性名称：“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 指纹)。
- 已知的密码(例如：您输入的信用卡号码或后端提供的身份认证令牌)。

重复测试和内存转储将帮助您获得有关数据泄露长度的统计信息。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您发现一些其他无法识别的敏感数据（在下面的“修复”部分中对此进行了详细介绍)。

5.3.13. 设备访问安全策略测试(MSTG-STORAGE-11)

5.3.13.1. 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，应用程序可以检查设备的以下内容:

- 受 PIN 或密码保护的设备锁定
- 最近的 Android 操作系统版本
- USB 调试是否激活
- 设备加密
- 设备是否 root（另请参阅“测试 root 检测”）

5.3.13.2. 静态分析

要测试应用程序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。 该策略应定义可用的检查及其执行。 例如，一项检查可能要求应用仅在 Android 6.0（API 级别 23）或更新版本上运行，如果 Android 版本低于 6.0，则关闭应用或显示警告。

检查实现策略的函数的源代码，并确定是否可以绕过它。

您可以通过查询系统参数 Settings.Secure 来实现对 Android 设备的检查。*Device Administration API*（设备管理 API)提供了创建可以强制执行密码策略和设备加密的应用程序的技术。

5.3.13.3. 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。

5.3.14. 参考文献

5.3.14.1. 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，例如密码或密码。”
- 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间，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 Intent，自定义 URL 和网络源）接收的数据。”

5.3.14.2. 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
- Themis - https://github.com/cossacklabs/themis

5.4. Android 加密 API

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，以及如何解释加密配置。在检查代码时，确保将使用的加密参数与本指南中链接的当前最佳实践进行比较。

我们可以识别 Android 中加密系统的关键组件：

- Security Provider
- KeyStore – 查看“测试数据存储”章节 KeyStore 部分
- KeyChain - 查看“测试数据存储”章节 KeyChain 部分

Android 加密 API 基于 Java 加密架构 (JCA)。 JCA 将接口和实现分开，从而可以包含多个可以实现加密算法集的安全提供程序。 大多数 JCA 接口和类都在 `java.security.*`和 `javax.crypto.*`包中定义。 此外，还有 Android 特定的包 `android.security.*`和 `android.security.keystore.*`。

KeyStore 和 KeyChain 提供了用于存储和使用密钥的 API（在后台，KeyChain API 使用 KeyStore 系统）。 这些系统允许管理加密密钥的整个生命周期。 实施加密密钥管理的要求和指南可以在密钥管理备忘单中找到。 我们可以确定以下阶段：

- 生成密钥
- 使用密钥
- 存储密钥
- 归档密钥
- 删除密钥

请注意，密钥的存储在“测试数据存储”一章中进行了分析。

这些阶段由 Keystore/KeyChain 系统管理。 然而，系统的工作方式取决于应用程序开发人员如何实现它。 对于分析过程，您应该关注应用程序开发人员使用的功能。 您应该识别并验证以下功能：

- 密钥生成
- 随机数生成
- 密钥轮换

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 开发者博客显示：
  - 建议停止指定安全提供程序。 相反，请始终使用已修补的安全提供程序。
  - 已放弃对 **Crypto** 提供程序的支持，并且该提供程序已被弃用。 这同样适用于其用于安全随机数的 **SHA1PRNG**。
- 对于 Android 8.1（API 级别 27）及更高版本，开发者文档显示：
  - Conscrypt，被称为 **AndroidOpenSSL**，在上面首选使用 Bouncy Castle，它有新的实现：**AlgorithmParameters:GCM**，**KeyGenerator:AES**，**KeyGenerator:DESEDE**，**KeyGenerator:HMACMD5**，**KeyGenerator:HMACSHA1**，**KeyGenerator:HMACSHA224**，**KeyGenerator:HMACSHA256**，**KeyGenerator:HMACSHA384**，**KeyGenerator:HMACSHA512**，**SecretKeyFactory:DESEDE**,以及 **Signature:NONEWITHECDSA**。
  - 您不应再将 **IvParameterSpec.class** 用于 GCM，而应使用 **GCMParameterSpec.class**。
  - Socket 已从 **OpenSSLSocketImpl** 更改为 **ConscryptFileDescriptorSocket** 和 **ConscryptEngineSocket**。

  - 带有 null 参数的 **SSLSession** 会产生 **NullPointerException**。
  - 您需要有足够大的数组作为输入字节来生成密钥，否则会引发 **InvalidKeySpecException**。
  - 如果 Socket 读取被中断，你会得到一个 **SocketException**。
- 对于 Android 9（API 级别 28）及更高版本，Android 开发者博客显示了更多变化：
  - 如果您仍然使用 **getInstance** 方法指定安全提供程序并且您的目标是低于 级别 28 的任何 API，则会收到警告。如果您的目标是 Android 9（API 级别 28）或更高版本，则会收到错误消息。
  - **Crypto** 安全提供程序现已删除。 调用它会导致 **NoSuchProviderException**。
- 对于 Android 10（API 级别 29），开发者文档列出了所有网络安全更改。

5.4.1. 建议

在应用程序检查期间应考虑以下建议列表：

- 您应确保遵循“移动应用加密”一章中概述的最佳实践。
- 您应该确保安全提供程序具有最新的更新 - 更新安全提供程序。
- 您应该停止指定安全提供程序并使用默认实现（AndroidOpenSSL、Conscrypt）。
- 您应该停止使用 Crypto 安全提供程序及其 **SHA1PRNG**，因为它们已被弃用。
- 您应该仅为 Android Keystore 系统指定安全提供程序。
- 您应该停止使用没有 IV 的基于密码的加密算法。
- 您应该使用 KeyGenParameterSpec 而不是 KeyPairGeneratorSpec。

5.4.1.1.安全提供程序

Android 依赖提供程序（**provider**）来实现 Java 安全服务。 这对于确保安全的网络通信和保护依赖于密码学的其他功能至关重要。

Android 中包含的安全提供程序列表因 Android 版本和 OEM 特定版本而异。 现在已知旧版本中的某些安全提供程序实现不太安全或易受攻击。 因此，Android 应用程序不仅应选择正确的算法并提供良好的配置，在某些情况下还应注意历史安全提供程序中实现的强度。

您可以使用以下代码列出一组现有的安全提供程序：

```
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();
//现在在屏幕上显示 string 或在调试日志中记录.
```

在修补了安全提供程序后，您可以在下面找到运行具有 Google Play API 的 Android 4.4（API 级别 19）模拟器中输出：

```
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 signatur
e))
provider: HarmonyJSSE1.0 (Harmony JSSE Provider)
provider: AndroidKeyStore1.0 (Android AndroidKeyStore security provider)
```

下面你可以找到运行具有 Google Play API 的 Android 9（API 级别 28）的模拟器输出：

```
provider: AndroidNSSP 1.0(Android Network Security Policy Provider)
provider: AndroidOpenSSL 1.0(Android's OpenSSL-backed security provider)
provider: CertPathProvider 1.0(Provider of CertPathBuilder and CertPathVerifi
er)
provider: AndroidKeyStoreBCWorkaround 1.0(Android KeyStore security provider
to work around Bouncy Castle)
provider: BC 1.57(BouncyCastle Security Provider v1.57)
provider: HarmonyJSSE 1.0(Harmony JSSE Provider)
provider: AndroidKeyStore 1.0(Android KeyStore security provider)
```

5.4.1.1.1. 更新安全提供程序

保持最新和已修补的组件是安全原则之一。这同样适用于提供程序（**provider**）。应用程序应该检查使用的安全提供程序是否是最新的，如果不是，则更新它。它与检查第三方库中的弱点（MSTG-CODE-5）有关。

5.4.1.1.2. 历史 Android 版本

对于一些支持旧版本 Android 的应用程序（例如：仅使用低于 Android 7.0（API 级别 24）的版本），捆绑最新的库可能是唯一的选择。 Spongy Castle（Bouncy Castle 的重打包版本）是这些情况下的常见选择。 重新打包是必要的，因为 Bouncy Castle 包含在 Android SDK 中。 最新版本的 Spongy Castle 可能修复了 Android 中早期版本的 Bouncy Castle 中存在的问题。 请注意， Android 带有的 Bouncy Castle 库通常不如 Bouncy Castle 系列中的同类库那么完整。 最后：请记住，打包 Spongy Castle 等大型库通常会导致产生 MultiDex 的 Android 应用程序。

5.4.1.2 密钥生成

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，可用于确保在应用程序中正确使用密钥。

下面是在 API 级别 23+上使用 AES/CBC/PKCS7Padding 的例子：

```
String keyAlias = "MySecretKey";

KeyGenParameterSpec keyGenParameterSpec = new KeyGenParameterSpec.Builder(key
Alias,
        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_ALGORI
THM_AES,
        "AndroidKeyStore");
keyGenerator.init(keyGenParameterSpec);

SecretKey secretKey = keyGenerator.generateKey();
```

KeyGenParameterSpec 表示该密钥可用于加密和解密，但不能用于其他目的，例如签名或验证。它进一步指定了块模式 (CBC)、填充 (PKCS #7)，并明确指定需要随机加密（这是默认设置）。 “**AndroidKeyStore**”是本示例中使用的安全提供程序的名称。这将自动确保密钥存储在有利于保护密钥的 **AndroidKeyStore** 中。

GCM 是另一种 AES 块模式，与其他旧模式相比，它提供了额外的安全优势。除了加密更安全之外，它还提供身份认证。使用 CBC（和其他模式）时，需要使用 HMAC 单独执行身份认证（请参阅“Android 上的篡改和逆向工程”一章）。请注意，GCM 是唯一不支持填充的 AES 模式。

尝试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[] 输入
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();
// 保存 IV 和加密后字节
```

IV（初始化向量）和加密字节都需要存储；否则无法解密。

下面是密码文本的解密方式。输入是加密的字节数组，iv 是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，因此应将其与密文（**encryptedBytes**）一起保存，以便以后解密。

在 Android 6.0（API 级别 23）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();
```

此示例创建密钥大小为 4096 位（即模数大小）的 RSA 密钥对。 椭圆曲线 (EC) 密钥也可以以类似的方式生成。 但是，从 Android 11（API 级别 30） 开始，AndroidKeyStore 不支持使用 EC 密钥进行加密或解密。 它们只能用于签名。

可以使用基于密码的密钥派生函数版本 2 (PBKDF2) 从密码短语生成对称加密密钥。 该加密协议旨在生成可用于加密目的的加密密钥。 算法的输入参数根据弱密钥生成功能部分进行调整。 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)
{
    //为后续使用初始化对象和变量
    int iterationCount = 10000;
    int saltLength     = keyLength / 8;
    SecureRandom random = new SecureRandom();
    //生成 salt
    byte[] salt = new byte[saltLength];
    random.nextBytes(salt);
    KeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, iterationC
ount, keyLength);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBKDF2WithHma
cSHA1");
    byte[] keyBytes = keyFactory.generateSecret(keySpec).getEncoded();
    return new SecretKeySpec(keyBytes, "AES");
}
```

上述方法需要一个包含密码和所需密钥长度（以位为单位）的字符数组，例如 128 位或 256 位 AES 密钥。 我们定义了 10,000 轮的迭代计数，将由 PBKDF2 算法使用。 迭代次数的增加显著增加了对密码进行暴力攻击的工作量，但是它会影响性能，因为密钥派生需要更多的计算能力。我们将 salt 大小定义为等于密钥长度，除以 8 以处理位到字节的转换。 我们使用 **SecureRandom** 类来随机生成 salt。 显然，salt 是你想要保持不变的东西，以确保为同一个支持的密码多次生成相同的加密密钥。 请注意，您可以将 salt 私有的存储在 **SharedPreferences** 中。 建议将 salt 从 Android 备份机制中排除，以防止更高风险数据的同步。

> 请注意，如果您将 root 设备或已修补（例如重新打包）的应用程序视为对数据的威胁，最好使用放置在 **AndroidKeystore** 中的密钥加密 salt。 基于密码的加密 (PBE) 密钥是使用推荐的 **PBKDF2withHmacSHA256** 算法生成的，直到 Android 8.0（API 级别 26）。 对于更高的 API 级别，最好使用最终会得到更长的哈希值的 **PBKDF2withHmacSHA256**。

注意：人们普遍错误地认为 NDK 应该被用来隐藏加密操作和硬编码密钥。然而，使用这种机制并不是有效的。攻击者仍然可以使用工具来发现所使用的机制，并对内存中的密钥进行转储。接下来，可以用 radare2 分析控制流，用 Frida 或两者的组合提取密钥：r2frida（详见 "Android 上的篡改和逆向工程 "一章中的 "反汇编原生代码"、"内存转储 "和 "内存搜索 "部分）。从 Android 7.0（API 级别 24）开始，不允许使用私有 API，相反：需要调用公共 API，这进一步影响了隐藏它的有效性，如 Android 开发者博客中所述。

5.4.1.3 随机数生成

加密需要安全的伪随机数生成 (PRNG)。 像 **java.util.Random** 这样的标准 Java 类不提供足够的随机性，实际上可能使攻击者可以猜测将要生成的下一个值，并使用该猜测来冒充另一个用户或访问敏感信息。

通常，应使用 **SecureRandom**。 但是，如果要支持 Android 4.4（API 级别 19）以下的 Android 版本，则需要额外注意以解决 Android 4.1-4.3（API 级别 16-18）版本中无法正确初始化 PRNG 的错误 。

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。 支持 **SecureRandom** 的 PRNG 提供程序使用来自 **AndroidOpenSSL** (Conscrypt) 提供程序的 **SHA1PRNG**。

5.4.2. 测试对称加密(MSTG-CRYPTO-1)

5.4.2.1.概览

该测试用例重点关注硬编码对称加密作为唯一的加密方法。应进行以下检查：

- 识别所有对称加密实例
- 对于每个已识别的实例，验证是否有任何硬编码的对称密钥
- 验证硬编码对称加密是否不是唯一的加密方法

5.4.2.2.静态分析

识别代码中所有对称密钥加密的实例，寻找任何加载或提供对称密钥的机制。你可以寻找：

- 对称算法（如 DES、AES 等）
- 密钥生成器的规范（如 **KeyGenParameterSpec**, **KeyPairGeneratorSpec**, **KeyPairGenerator**, **KeyGenerator**, **KeyProperties** 等）
- 使用 **java.security.***, **javax.crypto.***, **android.security.*** 和 **android.security.keystore.*** 包的类。

对于每个已识别的实例，验证是否使用了对称密钥：

- 不是应用程序资源的一部分
- 无法从已知值中得出
- 没有在代码中硬编码

对于每一个硬编码的对称密钥，验证其是否在安全敏感的情况下作为唯一的加密方法使用。

作为示例，我们将说明如何定位硬编码加密密钥的使用。首先反汇编和反编译应用程序以获得原始 Java 代码，例如通过使用 jadx。

现在搜索使用 **SecretKeySpec** 类的文件，例如通过简单的递归 grep 或使用 jadx 搜索功能:

```
grep -r "SecretKeySpec"
```

这将返回所有使用 **SecretKeySpec** 类的集合。现在检查这些文件，追踪哪些变量被用来传递密钥材料。下面的图显示了在一个准备生产的应用程序上进行这一评估的结果。我们可以清楚

地找到一个静态加密密钥的使用，该密钥是硬编码的，并在静态字节数组 **Encrypt.keyBytes** 中初始化。

```
import javax.crypto.spec.*;
import javax.crypto.*;
import java.security.*;
import android.util.*;

public class Encrypt
{
    private static byte[] keyBytes;

    static {
        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,
    }

    public static String decrypt(final String s) throws Exception {
        final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
        final Cipher instance = Cipher.getInstance("AES");
        instance.init(2, secretKeySpec);
        return new String(instance.doFinal(Base64.decode(s.getBytes(), 0)));
    }

    public static String encrypt(final String s) throws Exception {
        final SecretKeySpec secretKeySpec = new SecretKeySpec(Encrypt.keyBytes, "AES");
        final Cipher instance = Cipher.getInstance("AES");
        instance.init(1, secretKeySpec);
        return new String(Base64.encode(instance.doFinal(s.getBytes()), 0));
    }
}
```

5.4.2.3.动态分析

您可以对加密方法使用方法跟踪来确定输入/输出值，例如正在使用的密钥。在执行加密操作时监控文件系统访问，以评估写入或读取密钥材料的位置。 例如，使用 RMS - Runtime Mobile Security 的 API 监视器来监视文件系统。

5.4.3. 测试加密标准算法配置(MSTG-CRYPTO-2, MSTG-CRYPTO-3 和 MSTG-CRYPTO-4)

5.4.3.1. 概述

这些测试用例侧重于加密原语的实现和使用。应进行以下检查：

- 识别加密原语的所有实例及其实现（库或自定义实现）
- 验证加密原语的使用方式和配置方式
- 验证所使用的加密协议和算法是否出于安全目的而被弃用。

5.4.3.2. 静态分析

识别代码中加密原语的所有实例。识别所有自定义加密实现。您可以查找：

- **Cipher**, **Mac**, **MessageDigest**, **Signature** 类
- **Key**, **PrivateKey**, **PublicKey**, **SecretKey** 接口
- **getInstance**, **generateKey** 函数
- **KeyStoreException**, **CertificateException**, **NoSuchAlgorithmException** 异常
- 使用 **java.security.***, **javax.crypto.***, **android.security.*** 和 **android.security.keystore.***包的类。

通过不指定来确定所有对 getInstance 的调用都使用安全服务的默认提供程序（provider）（它意味着 AndroidOpenSSL，也称为 Conscrypt）。提供程序只能在 KeyStore 相关代码中指定（在这种情况下，KeyStore 应作为提供程序提供）。如果指定了其他提供程序，则应根据情况和商业案例（即 Android API 版本）对其进行验证，并应检查提供程序是否存在潜在漏洞。

确保遵循“移动应用程序加密”一章中概述的最佳实践。查看不安全和不推荐的算法以及常见的配置问题。

5.4.3.3. 动态分析

您可以对加密方法使用方法跟踪来确定输入/输出值，例如正在使用的密钥。在执行加密操作时监控文件系统访问，以评估写入或读取密钥材料的位置。 例如，使用 RMS - Runtime Mobile Security 的 API 监视器来监视文件系统。

5.4.4. 测试密钥的用途（MSTG-CRYPTO-5）

5.4.4.1. 概述

这个测试用例的重点是验证相同加密密钥的用途和重用。应进行以下检查：

- 识别使用加密的所有实例
- 确定使用加密材料的目的（保护使用中、传输中或静止的数据）
- 识别加密类型
- 验证是否根据其目的使用了加密

5.4.4.2. 静态分析

识别使用加密的所有实例。您可以查找：

- **`Cipher`, `Mac`, `MessageDigest`, `Signature`** 类
- **`Key`, `PrivateKey`, `PublicKey`, `SecretKey`** 接口
- **`getInstance`, `generateKey`** 函数
- **`KeyStoreException`, `CertificateException`, `NoSuchAlgorithmException`** 异常
- 导入 **`java.security.*`, `javax.crypto.*`, `android.security.*`** 和 **`android.security.keystore.*`**包的类。

对于每个已识别的实例，请确定其使用密码的目的及其类型。它可以用于：

- 用于加密/解密-确保数据的机密性
- 用于签署/验证-确保数据的完整性（以及在某些情况下用于审计）
- 用于维护-在操作期间保护密钥（如导入到 KeyStore）

此外，您应该识别使用已识别的加密实例的业务逻辑。

在验证过程中，应进行以下检查：

- 所有的密钥都是按照其创建时定义的目的来使用的吗？(这与 KeyStore 密钥有关，它可以定义 KeyProperties)
- 对于非对称钥匙，私钥是否专门用于签名和公钥加密？
- 对称密钥是否用于多种用途？如果在不同的情况下使用，应该生成一个新的对称密钥。
- 加密技术是否根据其商业目的而使用？

5.4.4.3. 动态分析

您可以对加密方法使用方法跟踪来确定输入/输出值，例如正在使用的密钥。在执行加密操作时监控文件系统访问，以评估写入或读取密钥材料的位置。 例如，使用 RMS - Runtime Mobile Security 的 API 监视器来监视文件系统。

5.4.5. 测试随机数生成（MSTG-CRYPTO-6）

5.4.5.1. 概述

这个测试用例关注应用程序使用的随机值。应进行以下检查：

- 确定所有使用随机值的实例
- 验证随机数生成器是否被视为密码学安全
- 验证如何使用随机数生成器
- 验证生成的随机值的随机性

5.4.5.2. 静态分析

识别随机数生成器的所有实例，寻找自定义的或众所周知的不安全的类。例如，**java.util.Random** 为每个给定的种子值产生一个相同的数字序列；因此，数字序列是可预测的。相反，应该选择目前被该领域的专家认为是强大的算法，并且应该使用具有足够长度的种子的经过测试的实现。

识别所有未使用默认构造函数创建的 **SecureRandom** 实例。 指定种子值可能会降低随机性。 首选 **SecureRandom** 的无参数构造函数，它使用系统指定的种子值生成 128 字节长的随机数。

一般来说，如果一个 PRNG 没有被宣传为密码学安全（例如 `java.util.Random`），那么它可能是一个可统计的 PRNG，不应该在安全敏感的内容中使用。 如果生成器已知并且可以猜测种子，则伪随机数生成器可能产生可预测的数字。 128 位种子是产生“足够随机”数字的良好起点。

一旦攻击者知道使用的是哪种类型的弱伪随机数发生器（PRNG），就可以轻而易举地编写一个概念证明，根据以前观察到的随机值生成下一个随机值，就像为 Java Random 做的那样。如果是非常弱的自定义随机生成器，可能会在统计学上观察到这种模式。尽管推荐的方法是反编译 APK 并检查算法（见静态分析）。

如果你想测试随机性，你可以尝试捕获一大组数字，并用 Burp 的 sequencer 检查，看看随机性的质量如何。

5.4.5.3. 动态分析

你可以在提到的类和方法上使用方法追踪来确定正在使用的输入/输出值。

5.4.6.参考

- [#nelenkov] - N. Elenkov, Android Security Internals, No Starch Press, 2014, Chapter 5.

5.4.6.1. 密码学参考

- Android Developer blog: Changes for NDK Developers - https://android-developers.googleblog.com/2016/06/android-changes-for-ndk-developers.html
- 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
- Android Developer blog: Some SecureRandom Thoughts - https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html
- Android Developer documentation - https://developer.android.com/guide
- BSI Recommendations - https://www.keylength.com/en/8/
- Ida Pro - https://www.hex-rays.com/products/ida/
- Legion of the Bouncy Castle - https://www.bouncycastle.org/java.html
- NIST Key Length Recommendations - https://www.keylength.com/en/4/
- Security Providers - https://developer.android.com/reference/java/security/Provider.html
- Spongy Castle - https://rtyley.github.io/spongycastle/

5.4.6.2. SecureRandom 参考

- Proper seeding of SecureRandom - https://www.securecoding.cert.org/confluence/display/java/MSC63-J.+Ensure+that+SecureRandom+is+properly+seeded
- Burpproxy Sequencer - https://portswigger.net/burp/documentation/desktop/tools/sequencer

5.4.6.3. 测试密钥管理参考

- Android Keychain API - https://developer.android.com/reference/android/security/KeyChain
- Android KeyStore API - https://developer.android.com/reference/java/security/KeyStore.html
- Android Keystore system - https://developer.android.com/training/articles/keystore#java
- Android Pie features and APIs - https://developer.android.com/about/versions/pie/android-9.0#secure-key-import
- KeyInfo Documentation - https://developer.android.com/reference/android/security/keystore/KeyInfo
- SharedPreferences - https://developer.android.com/reference/android/content/SharedPreferences.html

5.4.6.4. 密钥认证参考

- Android Key Attestation - https://developer.android.com/training/articles/security-key-attestation
- Attestation and Assertion - https://developer.mozilla.org/en-US/docs/Web/API/Web_Authentication_API/Attestation_and_Assertion
- FIDO Alliance TechNotes - https://fidoalliance.org/fido-technotes-the-truth-about-attestation/
- FIDO Alliance Whitepaper - https://fidoalliance.org/wp-content/uploads/Hardware-backed_Keystore_White_Paper_June2018.pdf
- Google Sample Codes - https://github.com/googlesamples/android-key-attestation/tree/master/server
- Verifying Android Key Attestation - https://medium.com/@herrjemand/webauthn-fido2-verifying-android-keystore-attestation-4a8835b33e9d
- W3C Android Key Attestation - https://www.w3.org/TR/webauthn/#android-key-attestation

5.4.6.4.1. 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。”

5.5. Android 本地身份认证

在本地身份认证期间，应用程序根据设备上本地存储的凭据对用户进行身份认证。换句话说，用户通过提供有效的 PIN、密码或人脸或指纹等生物特征来“解锁”应用程序或某些内层功能，这些特征通过引用本地数据进行验证。通常，这样做是为了用户可以更方便地恢复具有远程服务的现有会话，或者作为一种增强身份认证的手段来保护一些关键功能。

如前“移动应用验证架构”一章所述：测试人员应意识到，本地验证应始终在远程端点或基于加密原语执行。如果身份认证过程中没有返回数据，攻击者可以轻松绕过本地身份认证。

在 Android 中，Android 运行时支持两种本地身份认证机制：确认凭证流程和生物认证流程。

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(Contex
t.KEYGUARD_SERVICE);
if (!mKeyguardManager.isKeyguardSecure()) {
    // 显示用户没有设置锁屏的消息.
}
```

- 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认证。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");

        // 在 Android KeyStore 中设置该项的别名，该密钥将出现在该 KeyStore 中
        // 以及构建器构造函数中的约束（目的）
        keyGenerator.init(new KeyGenParameterSpec.Builder(KEY_NAME,
                KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYP
    T)
                .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
                .setUserAuthenticationRequired(true)
                        // 要求用户在 30 秒内解锁
                .setUserAuthenticationValidityDurationSeconds(30)
                .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS
    7)
                .build());
        keyGenerator.generateKey();
    } catch (NoSuchAlgorithmException | NoSuchProviderException
            | InvalidAlgorithmParameterException | KeyStoreException
            | CertificateException | IOException e) {
        throw new RuntimeException("Failed to create a symmetric key", e);
    }
    ```
    ```

- 设置锁屏以确认：

```
• private static final int REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS = 1;
//当成数字，以验证这是否是 activity 结果的来源
Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(nul
l, null);
if (intent != null) {
    startActivityForResult(intent, REQUEST_CODE_CONFIRM_DEVICE_CREDENTIAL
S);
}
```

- 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) {
        // 挑战完成，继续使用密码
        if (resultCode == RESULT_OK) {
            //将密钥用于实际的身份认证流
        } else {
            // 用户取消或未完成锁屏
            // 操作转到错误/取消流程.
        }
    }
}
```

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认证绕过。

5.5.1.3. 动态分析

验证用户成功通过身份认证后授权使用密钥的持续时间（秒）。 仅当使用 **setUserAuthenticationRequired** 时才需要。

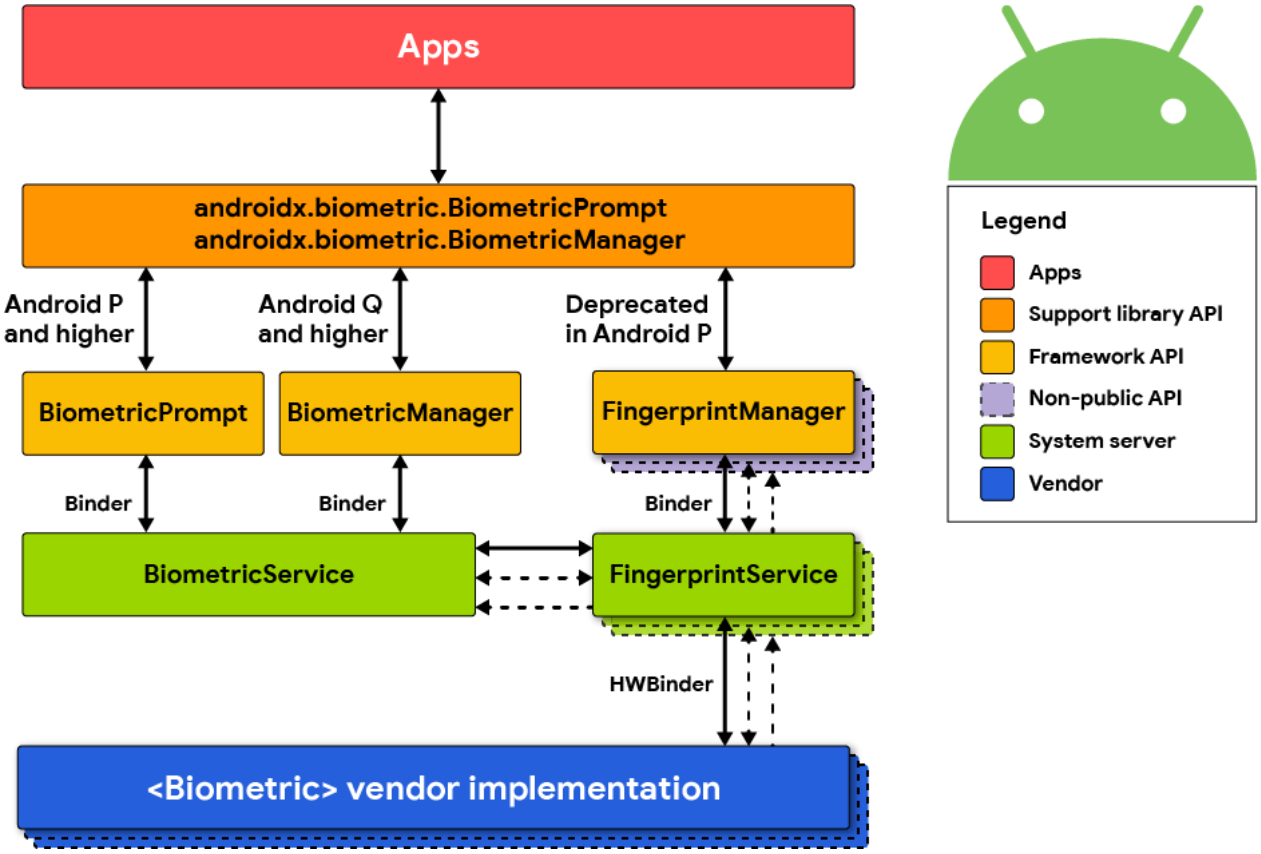
5.5.2. 测试生物特征身份认证（MSTG-AUTH-8）

5.5.2.1. 概述

生物特征认证是一种方便的认证机制，但在使用时也引入了额外的攻击面。 Android 开发人员文档提供了一个有趣的概述和指标，用于衡量生物识别解锁安全性。

Android 平台提供三种不同的生物特征认证类别：

- Android 10（API 级别 29）及更高版本：**`BiometricManager`**
- Android 9（API 级别 28）及更高版本：**`BiometricPrompt`**
- Android 6.0（API 级别 23）及更高版本：**`FingerprintManager`**（在 Android 9（API 级别 28）中已弃用）



`BiometricManager` 类可用于验证设备上是否有可用的生物识别硬件以及是否由用户配置。 如果是这种情况，可以使用类 **`BiometricPrompt`** 来显示系统提供的生物识别对话框。

`BiometricPrompt` 类是一项重大改进，因为它允许在 Android 上为生物识别身份认证提供一致的 UI，并且还支持更多的传感器而不仅仅是指纹。

这与 **`FingerprintManager`** 类不同，后者只支持指纹传感器，不提供 UI，迫使开发人员构建自己的指纹 UI。

Android 开发者博客上发布了关于 Android 上的生物识别 API 的非常详细的概述和解释。

5.5.2.1.1. FingerprintManager (在 Android 9 (API 级别 28)中已弃用)

Android 6.0（API 级别 23）引入了用于通过指纹验证用户的公共 API，但在 Android 9（API 级别 28）中已弃用。通过 **`FingerprintManager`** 类提供对指纹硬件的访问。应用程序可以通过实例化 **`FingerprintManager`** 对象并调用其 **`authenticate`** 方法来请求指纹认证。调用者注册回调方法来处理身份认证过程的可能结果（即成功、失败或错误）。请注意，此方法并不构成指纹认证已实际执行的有力证据——例如，认证步骤可能被攻击者修补，或者“成功”回调可能会使用动态插桩来重载。

您可以通过将指纹 API 与 Android **`KeyGenerator`** 类结合使用来实现更好的安全性。通过这种方法，对称密钥存储在 Android KeyStore 中，并使用用户的指纹解锁。例如，为了使用户能够访问远程服务，会创建一个 AES 密钥来加密身份认证令牌。通过在创建密钥时调用 **`setUserAuthenticationRequired(true)`**，确保用户必须重新认证才能检索它。然后可以将加密的身份认证令牌直接保存在设备上（例如通过 Shared Preferences）。这种设计是确保用户实际输入已授权指纹的一种相对安全的方式。

一个更安全的选择是使用非对称加密。在这里，移动应用程序在 KeyStore 中创建一个非对称密钥对，并在服务器后端注册公钥。随后的事务使用私钥进行签名，并由服务器使用公钥进行验证。

5.5.2.2. 静态分析

请注意，有相当多的供应商/第三方 SDK 提供生物识别支持，但也有其自身的不安全因素。 使用第三方 SDK 处理敏感的身份认证逻辑时要非常谨慎。

以下部分解释了不同的生物特征认证类别。

5.5.2.2.1. 生物特征库

Android 提供了一个名为 Biometric 的库，它提供了在 Android 10 中实现的 **`BiometricPrompt`** 和 **`BiometricManager`** API 的兼容版本，并具有回溯到 Android 6.0 (API 23) 的完整功能支持。

您可以在 Android 开发人员文档中找到有关如何显示生物特征身份认证对话框的参考实现和说明。

BiometricPrompt 类中有两种可用的 **authenticate** 方法。 其中一个需要 **CryptoObject**，它为生物特征认证增加了一层额外的安全性。

使用 CryptoObject 时的身份认证流程如下：

- 应用程序在 KeyStore 中创建一个密钥，并将 **setUserAuthenticationRequired** 和 **setInvalidatedByBiometricEnrollment** 设置为 true。 此外，**setUserAuthenticationValidityDurationSeconds** 应设置为 -1。
- 此密钥用于加密验证用户的信息（例如会话信息或身份认证令牌）。
- 在从 KeyStore 释放密钥以解密数据之前，必须提供一组有效的生物特征，该数据通过 **authenticate** 方法和 **CryptoObject** 进行验证。
- 即使在 root 过设备上也无法绕过此解决方案，因为来自 KeyStore 的密钥只能在生物特征认证成功后才能使用。

如果 **CryptoObject** 不用作身份认证方法的一部分，则可以使用 Frida 绕过它。 有关详细信息，请参阅“动态插桩”部分。

开发人员可以使用 Android 提供的多个验证类来测试其应用程序中生物特征身份认证的实现。

5.5.2.2.2. FingerprintManager

> 本节介绍如何使用 **FingerprintManager** 类实现生物认证。 请记住，此类已被弃用，应使用 Biometric 库作为最佳实践。 本节仅供参考，以防您遇到这样的实现并需要对其进行分析。

首先搜索 **FingerprintManager.authenticate** 调用。传递给此方法的第一个参数应该是 **CryptoObject** 实例，它是 FingerprintManager 支持的加密对象的包装类。如果该参数设置为 **null**，这意味着指纹授权是纯事件绑定的，可能会产生安全问题。

用于初始化密码包装器的密钥的创建可以追溯到 **CryptoObject**。除了在创建 **KeyGenParameterSpec** 对象期间调用 **setUserAuthenticationRequired(true)**之外，验证密钥是否都是使用 **KeyGenerator** 类创建的（请参见下面的代码示例）。

确保验证身份认证逻辑。要使身份认证成功，远程端点**必须**要求客户端提供从 KeyStore 检索到的密钥、从密钥派生的值或使用客户端私钥签名的值（见上文）。

安全地实施指纹认证需要遵循一些简单的原则，首先检查该类型的认证是否可用。在最基本的方面，设备必须运行 Android 6.0 或更高版本（API 23+）。还必须验证其他四个先决条件：

- 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.getSystemServ
ice(Context.KEYGUARD_SERVICE);
keyguardManager.isKeyguardSecure();  //如果不是这种情况请注意：要求用户设置受
保护的锁定屏幕
```

- 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) == Permi
ssionResult.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().ge
tAlgorithm(), ANDROID_KEYSTORE);
KeyInfo secetkeyInfo = (KeyInfo) factory.getKeySpec(yourencryptionkeyhere, Ke
yInfo.class);
secetkeyInfo.isInsideSecureHardware()
```

在某些系统上，也可以通过硬件强制执行生物特征认证策略。 通过以下方式检查：

```
keyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware();
```

下面介绍如何使用对称密钥对进行指纹认证。

指纹认证可以通过在 **KeyGenParameterSpec.Builder** 中添加 **setUserAuthenticationRequired(true)**来使用 **KeyGenerator** 类创建一个新的 AES 密钥来实现。

```
generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTOR
E);

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** 对象并使用密钥别名对其进行初始化。

```
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** 进行身份认证。包括在身份识别之前将 **Cipher** 对象包装到 **FingerprintManager.CryptoObject**，并传入 **FingerprintManager.authenticate**。

```
cryptoObject = new FingerprintManager.CryptoObject(cipher);
fingerprintManager.authenticate(cryptoObject, new CancellationSignal(), 0, th
is, null);
```

当身份认证成功时，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认证的 **CryptoObject**。

```
public void authenticationSucceeded(FingerprintManager.AuthenticationResult r
esult) {
    cipher = result.getCryptoObject().getCipher();

    //(... 操作经过认证的 cipher 对象...)
}
```

下面介绍如何使用非对称密钥对进行指纹认证。

要使用非对称加密实现指纹身份认证，首先使用 **KeyPairGenerator** 类创建签名密钥，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认证。使用指纹 API 对远程服务器进行身份认证的详细示例可以在 Android 开发者博客中找到。

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** 对其进行身份认证。

```
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, nu
ll);
```

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认证，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，对指纹身份认证有效的密钥在注册新指纹时将不可逆地失效。这可以防止攻击者通过注册新的指纹来获取密钥。

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器发生错误。

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

详情请查看这篇关于 Android KeyStore 和生物识别认证的博客文章。

- 指纹绕过：当 **`BiometricPrompt`** 类的 **`authenticate`** 方法中未使用 **`CryptoObject`** 时，此 Frida 脚本将绕过身份认证。 身份认证实现依赖于被调用的回调 **`onAuthenticationSucceded`**。
- 通过异常处理绕过指纹：此 Frida 脚本将在使用 **`CryptoObject`** 但使用方式不正确时尝试绕过身份认证。 详细解释可以在博文的“加密对象异常处理”部分找到。

5.5.3. 参考文献

5.5.3.1. OWASP MASVS

- MSTG-AUTH-1：“如果应用程序向用户提供对远程服务的访问，某种形式的身份认证，例如用户名/密码身份认证，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小设备访问安全策略，例如要求用户设置设备密码。”

5.5.3.2. 请求应用程序权限

- 运行时权限 - https://developer.android.com/training/permissions/requesting

5.6. Android 网络通信

几乎每一个 Android 应用都是作为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不受信任的网络上，如公共 Wi-Fi，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使用基于 HTTP 的网络服务的变体，因为这些协议有丰富的文档和支持。

5.6.1. 概览

5.6.1.1. Android 网络安全配置

从 Android 7.0（API 级别 24）开始，Android 应用程序可以使用所谓的网络安全配置功能来定制其网络安全设置，该功能具有以下关键功能：

- **明文流量**：保护应用程序不被意外使用明文流量（或启用它）。
- **自定义信任锚**：自定义应用程序的安全连接所信任的认证机构（CA）。例如，信任特定的自签名证书或限制应用程序所信任的公共 CA 的集合。
- **证书固定**：将一个应用程序的安全连接限制为特定的证书。
- **仅限调试的覆盖**：安全地调试应用程序中的安全连接，而不增加安装基础的风险。

如果一个应用程序定义了一个自定义的网络安全配置，你可以通过搜索 AndroidManifest.xml 文件中的 `android:networkSecurityConfig` 来获得其位置。

```
<application
android:networkSecurityConfig="@xml/network_security_config"
```

在这种情况下，该文件位于`@xml`（相当于/res/xml），名称为 "`network_security_config`"（可能有所不同）。你应该能够以 "res/xml/network_security_config.xml "的形式找到它。如果存在一个配置，在系统日志中应该可以看到以下事件:

```
D/NetworkSecurityConfig: Using Network Security Config from
resource network_security_config
```

网络安全配置是基于 XML 的，可用于配置应用程序范围和特定域的设置:

- base-config 适用于应用程序试图建立的所有连接。
- domain-config 覆盖了特定域名的 base-config（它可以包含多个域名项）。

例如，下面的配置使用 base-config 来防止所有域的明文流量。但它使用域配置覆盖了这一规则，明确地允许 localhost 的明文流量:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <base-config cleartextTrafficPermitted="false"   />
  <domain-config    cleartextTrafficPermitted="true">
    <domain>localhost</domain>
  </domain-config>
</network-security-config>
```

了解更多:

- Android P 中的网络安全配置安全分析师指南
- Android 开发者--网络安全配置
- Android Codelab - 网络安全配置

5.6.1.2. 默认配置

针对 Android 9（API 级别 28）及以上版本的应用程序的默认配置如下:

```
<base-config     cleartextTrafficPermitted="false">
```

```
  <trust-anchors>
    <certificates src="system" />
  </trust-anchors>
</base-config>
```

针对 Android 7.0（API 级别 24）至 Android 8.1（API 级别 27）的应用程序的默认配置如下：

```
<base-config     cleartextTrafficPermitted="true">
  <trust-anchors>
    <certificates src="system" />
  </trust-anchors>
</base-config>
```

针对 Android 6.0（API 级别 23）及以下版本的应用程序的默认配置如下：

```
<base-config     cleartextTrafficPermitted="true">
  <trust-anchors>
    <certificates src="system" />
    <certificates src="user" />
  </trust-anchors>
</base-config>
```

5.6.2. 测试网络中的数据加密(MSTG-NETWORK-1)

5.6.2.1. 测试使用安全协议的网络请求

首先，你应该在源代码中识别所有的网络请求，并确保不使用明文的 HTTP URL。通过使用 HttpsURLConnection 或 SSLSocket（用于使用 TLS 的套接字级通信），确保敏感信息通过安全通道发送。

5.6.2.2. 测试网络 API 使用

接下来，即使是使用应该进行安全连接的低级 API（如 SSLSocket），也要注意它必须安全地实现。例如，SSLSocket 并不验证主机名。使用 `getDefaultHostnameVerifier` 来验证主机名。Android 开发者文档包括一个代码示例。

5.6.2.3. 测试明文流量

下一步，你应该确保应用程序不允许明文 HTTP 流量。自从 Android 9（API 级别 28）以来，明文 HTTP 流量默认被阻止（由于默认的网络安全配置），但应用程序仍然可以通过多种方式发送它：

- 在 AndroidManifest.xml 文件中设置`<application>`标签的 android:usesCleartextTraffic 属性。请注意，在有网络安全配置的情况下，这个标签会被忽略。
- 通过设置 "网络安全配置 "来启用明文流量。在<domain-config>元素上将 `cleartextTrafficPermitted` 属性设置为 true。
- 使用低级别的 API（如 Socket）来建立一个自定义的 HTTP 连接。
- 使用跨平台框架（如 Flutter、Xamarin.....），因为这些框架通常有自己的 HTTP 库的实现。

上述所有情况都必须作为一个整体进行仔细分析。例如，即使应用程序在其 Android Manifest 或网络安全配置中不允许使用明文流量，它实际上可能仍然在发送明文 HTTP 流量。例如它使用的是低级别的 API（网络安全配置被忽略）或跨平台框架的配置存在问题，就可能出现这种情况。

更多信息请参考《HTTPS 和 SSL 的安全》一文。

5.6.2.4. 动态分析

拦截被测试的应用程序传入和传出的网络流量，并确保这些流量是加密的。你可以通过以下任何一种方式拦截网络流量:

- 用 OWASP ZAP 或 Burp Suite 等拦截代理捕捉所有的 HTTP(S)和 Websocket 流量，并确保所有请求是通过 HTTPS 而不是 HTTP 发出的。
- 像 Burp 和 OWASP ZAP 这样的拦截代理将只显示 HTTP(S)流量。你可以使用 Burp 插件，如 Burp-non-HTTP-Extension 或 mitm-relay 工具来解码和显示通过 XMPP 和其他协议的通信。

    一些应用程序可能会因为证书固定而无法使用 Burp 和 OWASP ZAP 这样的代理服务器。在这种情况下，请检查 "测试自定义证书存储和证书固定"。

更多细节请参考：

- "拦截网络层的流量"，来自 "移动应用程序网络通信 "一章
- "设置网络测试环境"，来自 "Android 基本安全测试 "一章

5.6.3. 测试 TLS 设置(MSTG-NETWORK-2)

详情请参考 "移动应用程序网络通信 "一章中的 "验证 TLS 设置 "一节。

5.6.4. 测试终端身份验证（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解决方案发布到生产版本，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。在下面的部分中，您可以找到应该查找的不安全使用的示例。

> 请注意，从 Android 8.0（API 级别 26）开始，不再支持 SSLv3，并且 **`HttpsURLConnection`** 将不再执行回退到不安全的 TLS/SSL 协议。

5.6.4.1. 静态分析

5.6.4.1.1. 验证目标 SDK 版本

针对 Android 7.0（API 级别 24）或更高的应用程序将使用默认的网络安全配置，**不信任任何用户提供的 CA**，来减少通过引诱用户安装恶意的 CA 进行 MITM 攻击的可能性。

使用 apktool 对应用程序进行解码，并验证 apktool.yml 中的 `targetSdkVersion` 是否等于或高于 24。

```
grep targetSdkVersion
 UnCrackable-
 Level3/apktool.yml
 targetSdkVersion: '28'
```

然而，即使 targetSdkVersion >=24，开发者也可以通过使用自定义网络安全配置，定义一个自定义信任锚，迫使应用程序信任用户提供的 CA，从而禁用默认保护措施。见 "分析自定义信任锚"。

5.6.4.1.2. 分析自定义信任锚

搜索网络安全配置文件，检查任何自定义的<trust-anchors>定义<certificates src="user">（应该避免这样做）。

你应该仔细分析条目的优先顺序：:

- 如果<domain-config>项或父<domain-config>项中没有设置值，那么现有的配置将基于<base-config>。
- 如果没有在这个条目中定义，将使用默认的配置。

下面是针对 Android 9（API 级别 28）的应用程序的网络安全配置的示例：

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
  <domain-config>
    <domain includeSubdomains="false">owasp.org</domain>
    <trust-anchors>
      <certificates src="system" />
      <certificates src="user" />
    </trust-anchors>
  </domain-config>
</network-security-config>
```

分析:

- 没有<base-config>，意味着 Android 9（API 级别 28）或更高的默认配置将被用于所有其他连接（原则上只信任系统 CA）。
- 然而，<domain-config>覆盖了默认配置，允许应用程序同时信任指定<domain>（owasp.org）的系统和用户 CA。
- 由于 includeSubdomains="false"，这并不影响子域。

把所有这些放在一起，我们可以把上述网络安全配置翻译成："应用程序信任 owasp.org 域的系统和用户 CA，不包括其子域。对于任何其他域，该应用程序将只信任系统 CA"。

5.6.4.1.3. 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。此时应检查以下条件：

- 证书是否由可信 CA 签发?
- 证书是否过期?
- 证书是否自签名?

以下代码片段有时会在开发过程中使用，并且会接受任何证书，覆盖函数 **checkClientTrusted**、**checkServerTrusted** 和 **getAcceptedIssuers**。 应避免此类实现，如果必要这样，应将它们与生产构建明确分开，以避免内置安全漏洞。

```
TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[] {};
        }

        @Override
        public void checkClientTrusted(X509Certificate[] chain, String authTy
pe)
            throws CertificateException {
        }

        @Override
        public void checkServerTrusted(X509Certificate[] chain, String authTy
pe)
            throws CertificateException {
        }
    }
```

```
};

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());
```

5.6.1.1.4. 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, Ssl
Error error) {
        //忽略 TLS 证书错误并指示 WebViewClient 加载网站
        handler.proceed();
    }
});
```

5.6.1.1.5. Apache Cordova 证书验证

如果在应用程序的 manifest 文件中开启了 **android:debuggable** 调试属性，则 Apache Cordova 框架内部 WebView 使用的实现会忽略方法 **onReceivedSslError** 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”。

5.6.1.1.6. 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。

5.6.4.2. 动态分析

当测试一个针对 Android 7.0（API 级别 24）或更高的应用程序时，它应该有效地应用网络安全配置，而且一开始你不应该能够看到解密的 HTTPS 流量。然而，如果应用程序的目标 API 级别是低于 24 的，该应用程序将自动接受已安装的用户证书。

为了测试不恰当的证书验证，使用一个拦截代理（如 Burp）发起 MITM 攻击。尝试以下选项：

- 自签名证书
  1. 在 Burp 中，点击 **Proxy** 选项卡，选择 **Options** 选项卡。
  2. 查看 **Proxy Listeners** 部分，点击你的 **listener** 使其高亮显示，然后单击 **Edit**。
  3. 转到 **Certificate** 选项卡，选中 **Use a self-signed certificate**（使用自签名证书），然后单击 **Ok**。
  4. 运行您的应用程序。 如果您能够看到 HTTPS 流量，则表明您的应用程序正在接受自签名证书。
- 接受不受信任 CA 的证书
  1. 在 Burp 中，点击 **Proxy** 选项卡，选择 **Options** 选项卡。
  2. 查看 **Proxy Listeners** 部分，点击你的 **listener** 使其高亮显示，然后单击 **Edit**。
  3. 转到 **Certificate** 选项卡，选中“**Generate a CA-signed certificate with a specific hostname**”，并输入后端服务器的主机名。
  4. 运行需要测试的应用程序。如果能够看到 HTTPS 流量，就说明该应用程序接受不受信任 CA 的证书。
- 接受不正确的主机名
  1. 在 Burp 中，点击 **Proxy** 选项卡，选择 **Options** 选项卡。
  2. 查看 **Proxy Listeners** 部分，点击你的 **listener** 使其高亮显示，然后单击 **Edit**。
  3. 转到 **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。
  4. 运行需要测试的应用程序。如果能够看到 HTTPS 流量，就说明应用程序接受所有主机名。

如果你仍然无法看到任何解密的 HTTPS 流量，你的应用程序可能正在实施证书固定。

5.6.5. 测试自定义证书的存储和证书固定(MSTG-NETWORK-4)

5.6.5.1. 概述

该测试验证应用程序是否正确地实现了身份固定（证书或公钥固定）。

更多细节请参考 "移动应用网络通信 "章节中的 "身份固定 "一节。

5.6.5.2. 静态分析

5.6.5.2.1. 网络安全配置中的证书固定

网络安全配置也可以用来将声明将证书固定在特定的域上。这是通过在网络安全配置中提供<pin-set>来实现的，它是一组对应 X.509 证书的公钥（SubjectPublicKeyInfo）的摘要（哈希值）。

当试图建立与远程端点的连接时，系统将：

- 获取并验证传入的证书。
- 提取公钥。
- 计算提取的公钥的摘要。
- 将该摘要与本地固定摘要进行比较。

如果至少有一个固定的摘要相匹配，那么证书链将被认为是有效的，连接将继续进行。

```
<?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">
      <!—OWASP 网站服务器证书的中间 CA 的公钥(X.509 证书的 SubjectPublicKeyInfo)
      的哈希值 -->
      <pin digest="SHA-
      256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
      <!-- OWASP 网站服务器证书的根 CA 的公钥（X.509 证书的
      SubjectPublicKeyInfo）的哈希值。 -->
```

```
      <pin digest="SHA-
      256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
    </pin-set>
  </domain-config>
</network-security-config>
```

检查<pin-set>元素中是否有过期日期。如果过期了，证书固定将被禁用于影响的域。

测试提示：如果证书固定验证检查失败，应在系统日志中记录以下事件：

```
I/X509Util: Failed to validate the certificate chain, error: Pin verification
 failed
```

5.6.5.2.2. TrustManager

实现证书固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。
- 将证书固定到默认 Apache Httpclient 实例。

要分析证书固定的正确实现，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);
// 创建一个使用 TrustManager 的 SSLContext
// SSLContext context = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
```

应用程序的实现可能不同，如仅针对证书的公钥固定、整个证书或整个证书链固定。

5.6.5.2.3. 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固定功能。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/UwQAapahrjCOjYI3oLUx5AQxPBR02Jz6/E2pt
0IeLXA=")
            .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 WebViewClient 的事件处理函数对每个请求进行某种类型的“证书固定”校验。校验示例代码如下：

```
WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
    private String expectedIssuerDN = "CN=Let's Encrypt Authority X3,O=Let's
Encrypt,C=US;";

    @Override
    public void onLoadResource(WebView view, String url)  {
        //关于"WebView.getCertificate()"的Android API 文档:
        //获取主要顶层页面的SSL 证书
        //如果没有证书，则为空（网站不安全）。
        //
        // SslCertificate 类的可用信息是 " Issuer DN"、" Subject DN "和有效期助
手。
        SslCertificate serverCert = view.getCertificate();
        if(serverCert != null){
            //在这应用证书或公钥固定比较
                //抛出异常以取消资源加载...
            }
        }
    }
});
```

或者，最好使用配置好固定的 OkHttpClient，并让它作为代理重写 **WebViewClient** 类的 **shouldInterceptRequest** 函数。

5.6.5.2.4. Xamarin 应用

在 Xamarin 中开发的应用程序通常使用 ServicePointManager 来实现证书固定。

通常会创建一个函数来检查证书并将布尔值返回给方法 ServerCertificateValidationCallback：

```
[Activity(Label = "XamarinPinning", MainLauncher = true)]
    public class MainActivity : Activity
    {
        // SupportedPublicKey - 公钥的十六进制值.
        // 使用GetPublicKeyString()方法来确定我们要固定证书的公钥。第一次取消对Va
```

```
lidateServerCertificate 函数中的调试代码的注释，以确定要固定的值。
        private const string SupportedPublicKey = "3082010A02820101009CD30CF0
5AE52E47B7725D3783B..."; // 为便于阅读而缩短

        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.G
etPublicKeyString();
        }

        protected override void OnCreate(Bundle savedInstanceState)
        {
            System.Net.ServicePointManager.ServerCertificateValidationCallbac
k += 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。

可以在 MASTG 源上获得带有上一个示例的示例 Xamarin 应用程序

解压 APK 文件后，使用 dotPeak、ILSpy 或 dnSpy 等 .NET 反编译器对存储在“Assemblies”文件夹中的应用程序 dll 进行反编译，并确认 ServicePointManager 的使用。

5.6.5.2.5. Cordova 应用

基于 Cordova 的混合应用程序不支持原生的证书固定，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。

```
  // 根据证书固定验证的终端.
  var server = "https://www.owasp.org";
  // SHA256 指纹(可以通过 "openssl s_client -connect hostname:443 | openssl x5
09 -noout -fingerprint -sha256"获得
```

```
 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);
    // 消息总是: CONNECTION_SECURE.
    // 现在对受信任的服务器做一些事情。
  }

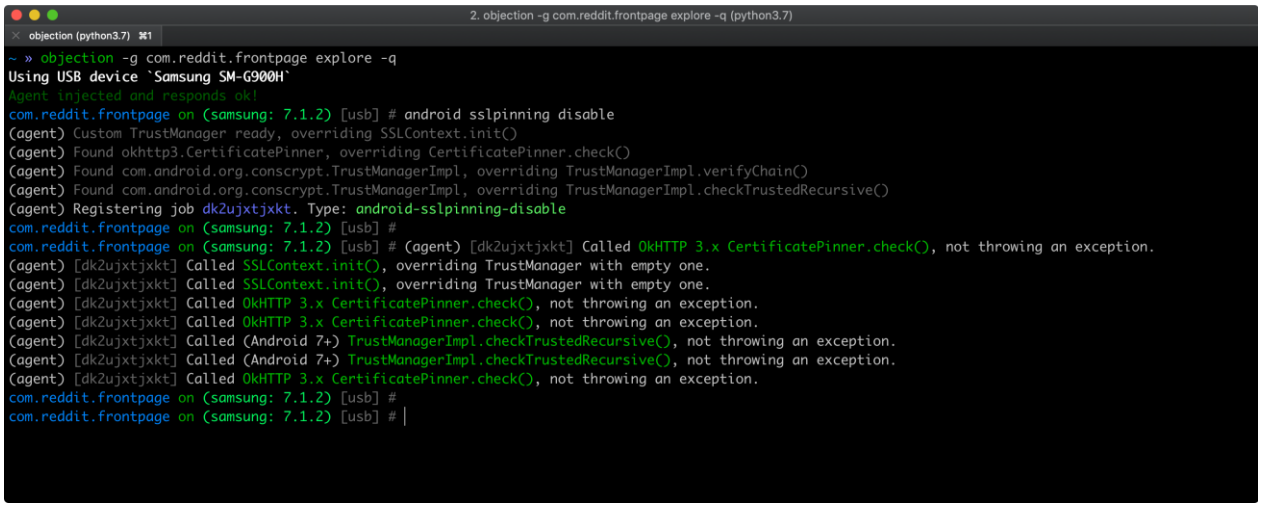
  function errorCallback(message) {
    alert(message);
    if (message === "CONNECTION_NOT_SECURE") {
      // 很可能有一个中间人在进行攻击，要小心!
    } else if (message.indexOf("CONNECTION_FAILED") >- 1) {
      // 没有连接（还没有）。互联网可能关闭了。超时后再试（几次）。
    }
  }
```

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。

5.6.5.3. 动态分析

按照 "测试终端身份验证>动态分析 "的指示进行。如果这样做并没有导致流量被代理，这可能意味着证书固定实际上已经实施，所有的安全措施都已到位。是否所有的域名都发生了同样的情况?

作为一个快速的测试，你可以尝试使用 "绕过证书固定 "中描述的 objection 来绕过证书固定。被 objection 劫持的与固定有关的 API 应该出现在 objection 的输出中。



然而，请记住：

- 这些 API 可能并不完整。
- 如果没有任何东西被劫持，这并不一定意味着该应用没有实现固定功能。

在这两种情况下，应用程序或其某些组件可能会以 objection 所支持的方式实现自定义固定。请查看静态分析部分，了解特定固定的特征并进行更深入的测试。

5.6.6. 测试安全提供程序（MSTG-NETWORK-6）

5.6.6.1. 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，Google 就禁止使用了存在漏洞的 OpenSSL 版本的应用程序提交到 Play 商店（包括新应用程序和版本更新的版本）。

5.6.6.2. 静态分析

基于 Android SDK 的应用程序应该依赖于 GooglePlay 服务。例如：在 gradle 构建文件中，我们可以在依赖块中找到 **`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。

同步修补：

```
//这是一个在后台运行的同步适配器，所以你可以运行同步修补。
public class SyncAdapter extends AbstractThreadedSyncAdapter {

  ...

  // 每次尝试同步时都会调用这个；这没有问题，
  // 因为如果安全提供者是最新的，其开销可以忽略不计。
  @Override
  public void onPerformSync(Account account, Bundle extras, String authority,
      ContentProviderClient provider, SyncResult syncResult) {
    try {
      ProviderInstaller.installIfNeeded(getContext());
    } catch (GooglePlayServicesRepairableException e) {

      // 表示 Google Play 服务已经过期、禁用，等等。

      // 提示用户安装/更新/启用 Google Play 服务。
      GooglePlayServicesUtil.showErrorNotification(
          e.getConnectionStatusCode(), getContext());

      // 通知 SyncManager 发生了一个软错误。
      syncResult.stats.numIOExceptions++;
      return;

    } catch (GooglePlayServicesNotAvailableException e) {
      // 表示一个不可恢复的错误；ProviderInstaller 无法安装一个最新的提供者。
      // 通知 SyncManager 发生了一个硬性错误。
      //在这种情况下：请确保你通知你的 API。
      syncResult.stats.numAuthExceptions++;
      return;
    }

    // 如果达到这一点，你就知道提供者已经是最新的，或者已经成功更新。
  }
}
```

异步修补：

```
//这是应用程序的主要Activity /第一个Activity，它存在的时间足以使securityprovider
的异步安装工作得以进行。
public class MainActivity extends Activity
    implements ProviderInstaller.ProviderInstallListener {

  private static final int ERROR_DIALOG_REQUEST_CODE = 1;

  private boolean mRetryProviderInstall;

  //当Activity 被创建时，更新安全提供者。
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    ProviderInstaller.installIfNeededAsync(this, this);
  }

  /**
   * This method is only called if the provider is successfully updated
   * (or is already up-to-date).
   */
  @Override
  protected void onProviderInstalled() {
    // 供应商是最新的，应用程序可以进行安全的网络调用。
  }

  /**
   * This method is called if updating fails; the error code indicates
   * whether the error is recoverable.
   */
  @Override
  protected void onProviderInstallFailed(int errorCode, Intent recoveryInten
t) {
    if (GooglePlayServicesUtil.isUserRecoverableError(errorCode)) {
      // 可恢复的错误。显示一个对话框，提示用户安装/更新/启用Google Play 服务。
      GooglePlayServicesUtil.showErrorDialogFragment(
          errorCode,
          this,
          ERROR_DIALOG_REQUEST_CODE,
          new DialogInterface.OnCancelListener() {
            @Override
            public void onCancel(DialogInterface dialog) {
              // 用户选择不采取恢复行动
              onProviderInstallerNotAvailable();
            }
          });
    } else {
```

```
      // Google Play 服务不可用.
      onProviderInstallerNotAvailable();
    }
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
      Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == ERROR_DIALOG_REQUEST_CODE) {
      //通过 GooglePlayServicesUtil.showErrorDialogFragment 添加一个 fragment
      // 在实例状态被恢复之前，会抛出一个错误。
      // 因此，在这里设置一个标志，这将导致 fragment 延迟到 onPostResume。
      mRetryProviderInstall = true;
    }
  }

  /**
   * On resume, check to see if we flagged that we need to reinstall the
   * provider.
   */
  @Override
  protected void onPostResume() {
    super.onPostResult();
    if (mRetryProviderInstall) {
      // We can now safely retry installation.
      ProviderInstall.installIfNeededAsync(this, this);
    }
    mRetryProviderInstall = false;
  }

  private void onProviderInstallerNotAvailable() {
    // 如果提供者因某种原因不能更新，就会达到这一点。
    //应用程序应该认为所有的 HTTP 通信都是脆弱的，
    // 并采取适当的行动（例如，通知后端，阻止某些高风险的行动，等等）。
  }
}
```

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。

5.6.6.3. 动态分析

如果有源代码：

- 在调试模式下运行应用程序，然后在应用程序最先与后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 劫持 **java.security** 包，然后劫持 **java.security.Security** 的 **getProviders** 方法（不带参数）。返回值为 **Provider** 数组。
- 确定第一个提供程序是否是 **GmsCore_OpenSSL**。

5.6.7 参考文献

5.6.7.1. OWASP MASVS

- MSTG-NETWORK-1: "数据在网络上使用 TLS 进行加密。安全通道在整个应用中被一致使用。"
- 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。”

5.6.7.2. Android 开发人员文档

- 网络安全配置 - https://developer.android.com/training/articles/security-config
  - 网络安全配置 (替代存档) - https://webcache.googleusercontent.com/search?q=cache:hOONLxvMTwYJ:https://developer.android.com/training/articles/security-config+&cd=10&hl=nl&ct=clnk&gl=nl

5.6.7.3. Xamarin 证书固定

- Certificate and Public Key Pinning with Xamarin - https://thomasbandt.com/certificate-andhttps://thomasbandt.com/certificate-and-public-key-pinning-with-xamarinpublic-key-pinning-with-xamarin
- ServicePointManager - https://msdn.microsoft.com/enhttps://msdn.microsoft.com/en-us/library/system.net.servicepointmanager(v=vs.110).aspxus/library/system.net.servicepointmanager(v=vs.110).aspx

5.6.7.4. Cordova 证书固定

- PhoneGap SSL Certificate Checker plugin - https://github.com/EddyVerbruggen/SSLCertificateChecker-PhoneGap-Plugin

5.7. Android 平台 API

5.7.1. 测试应用权限 (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。它们通过声明它们需要使用系统数据和功能权限来请求此访问。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 权限根据其提供的保护级别分为四类：

- **Normal**：此权限使应用程序能够访问隔离的应用程序级功能，而对其他应用程序、用户和系统的风险最小。对于目标平台为 Android 6.0（API 级别 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 级别较低的应用程序，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**。

所有权限的列表可以在 Android 开发者文档中找到，以及关于如何：

- 在应用程序的 manifest 文件中声明应用程序权限。
- 以编程方式请求应用程序权限。
- 定义自定义应用权限以与其他应用共享您的应用资源和功能。

5.7.1.1.1. Android 8.0（API 级别 26）的变化

以下变化将影响所有在 Android 8.0（API 级别 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。

- **联系人提供程序使用统计变化**：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 级别 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**：除非身份认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**：危险权限划分为权限组（如 **STORAGE** 权限组包含 **READ_EXTERNAL_STORAGE** 和 **WRITE_EXTERNAL_STORAGE**）。在 Android 8.0（API 级别 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 级别 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 级别为 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 级别 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 级别 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 目标 API 级别低于 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） 权限从 **PHONE** 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 级别 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 **Settings**”->“位置 **Location**”下）。

以 Android 9（API 级别 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（API 级别 29）变化

Android 10（API 级别 29）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（API 级别 29） 上运行的所有应用程序，包括那些目标平台为较低 API 级别的应用程序。

- **限制位置访问**：位置访问增加新的权限选项“使用时允许”
- **默认分区存储**：以 Android 10 为目标的应用程序（API 级别 29）不需要声明任何存储权限来访问他们在外部存储的应用程序特定目录中的文件，以及从媒体存储创建的文件。
- **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

权限是通过 manifest 文件中`<activity>`标记里的 `android:permission` 属性进行应用的。这些权限控制哪个应用可以启动 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

通过使用 manifest 中 `<service>`标记中的 `android:permission` 属性应用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`<receiver>` 标记的中 `android:permission` 属性应用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；只是不会传递该广播。

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 内容提供者权限强制执行

使用**<provider>**标记的中 **android:permission** 属性应用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，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**。

5.7.1.6. ContentProvider URI 权限

使用 ContentProvider 时仅仅应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 或从 Activity 返回结果时，方法可以设置 **Intent.FLAG_GRANT_READ_URI_PERMISSION** 和 /或

Intent.FLAG_GRANT_WRITE_URI_PERMISSION。这将授予特定 URI Activity 权限，无论其是否具有访问来自内容提供程序的数据的权限。

此机制支持常见的能力式模型，该模型中通过用户交互实现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 manifest 上 **android:grantUriPermissions** 属性或者节点可以帮助限制 URI 。

5.7.1.7. URI 权限文档

- grantUriPermission
- revokeUriPermission
- 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 manifest 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 应用 to launch the activity of myapp ap
p, any 应用 you grant this permission will be able to launch main activity by m
yapp 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** 标签请求该权限。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>
```

我们建议在注册权限时使用反向域注释，如上面的示例（例如 **com.domain.application.permission**），以避免与其他应用程序发生冲突。

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某些权限，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 Android Asset Packaging tool（aapt）工具检查 APK 文件的权限。

> aapt 在 Android SDK 的 build-tools 文件夹内。其需要一个 APK 文件作为输入。你可以运行在“显示已安装应用”部分提到的 **adb shell pm list packages -f | grep -i <keyword>**命令显示设备中的所有 APK。

```
$ aapt d permissions app-x86-debug.apk
package: sg.vp.owasp_mobile.omtg_android
uses-permission: name='android.permission.WRITE_EXTERNAL_STORAGE'
uses-permission: name='android.permission.INTERNET'
```

或者，您可以通过 adb 和 dumpsys 工具获得更详细的权限列表：

```
$ adb shell dumpsys package sg.vp.owasp_mobile.omtg_android | grep permission
    requested permissions:
      android.permission.WRITE_EXTERNAL_STORAGE
      android.permission.INTERNET
      android.permission.READ_EXTERNAL_STORAGE
    install permissions:
      android.permission.INTERNET: granted=true
      runtime permissions:
```

请参考权限概述，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 manifest 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插桩等方式绕过。建议调用 **ContextCompat.checkSelfPermission** 方法以检查 activity 是否具有指定的权限。当看到类似如下代码时，请确保在 manifest 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 manifest 文件进行比较。

```
if (ContextCompat.checkSelfPermission(secureActivity.this, Manifest.READ_INCO
MING_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";
// 我们从检查当前Activity 的权限开始
if (ContextCompat.checkSelfPermission(secureActivity.this,
        Manifest.permission.WRITE_EXTERNAL_STORAGE)
        != PackageManager.PERMISSION_GRANTED) {

    // 未授权Permission is not granted
    // 我们应该解释一下吗?
    if (ActivityCompat.shouldShowRequestPermissionRationale(secureActivity.th
is,
```

```
        //获取是否应显示 UI 以及请求权限的理由.
        //只有在您没有权限并且没有向用户清楚地传达所请求的权限的基本原理时，您才应该
这样做.
            Manifest.permission.WRITE_EXTERNAL_STORAGE)) {
        // 异步线程等待用户响应.
        // 在用户看到解释后，请尝试再次请求权限.
    } else {
        // 请求无需解释的权限.
        ActivityCompat.requestPermissions(secureActivity.this,
                new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
                MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE);
        // MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE 将是应用程序定义的 int
常量.
        // 回调方法获取请求的结果.
    }
} else {
    // 已授予权限的调试消息打印在终端中.
    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 //需要重写系统方法 onRequestPermissionsResult()
public void onRequestPermissionsResult(int requestCode, // requestCode 是您在
requestPermissions () 中指定
        String permissions[], int[] permissionResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_WRITE_EXTERNAL_STORAGE: {
            if (grantResults.length > 0
                && permissionResults[0] == PackageManager.PERMISSION_GRANTED)
{
                // 0 代表请求已取消，如果 int 数组等于 requestCode 即已获得权限.
            } else {
                // 这里处理权限拒绝.
```

```
                Log.v(TAG, "Permission denied");
            }
            return;
        }
        // 可以在此处添加其他 switch case 以进行多个权限检查.
    }
}
```

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 级别 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 级别 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 Manifest 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_EXTERNAL_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，无需显式请求。

5.7.1.11. 权限分析

始终检查应用程序是否正在请求其实际需要的权限。确保不请求与应用程序目标无关的权限，尤其是 **DANGEROUS** 权限和 **SIGNATURE** 权限，因为如果处理不当，它们可能会影响用户和应用程序。例如，如果单人游戏应用程序需要访问 **android.permission.WRITE_SMS**，这应该是可疑的。

在分析权限时，您应该调查应用程序的具体用例场景，并始终检查是否有用于任何 **DANGEROUS** 权限的替换 API。SMS Retriever API 就是一个很好的例子，它在执行基于 SMS 的用户验证时简化了 SMS 权限的使用。通过使用此 API，应用程序不必声明 **DANGEROUS** 权限，这对应用程序的用户和开发人员都有好处，他们不必提交权限声明表单。

5.7.1.12. 动态分析

可以使用 **adb** 查询已安装应用的权限。下面演示了演示了如何检查应用程序使用的权限：

```
$ adb shell dumpsys package com.google.android.youtube
...
```

```
declared permissions:
  com.google.android.youtube.permission.C2D_MESSAGE: prot=signature, INSTALLE
D
requested permissions:
  android.permission.INTERNET
  android.permission.ACCESS_NETWORK_STATE
install permissions:
  com.google.android.c2dm.permission.RECEIVE: granted=true
  android.permission.USE_CREDENTIALS: granted=true
  com.google.android.providers.gsf.permission.READ_GSERVICES: granted=true
...
```

输出使用以下类别显示所有权限：

- **declared permissions** 声明的权限：所有 *custom* 自定义权限的列表。
- **requested and install permissions** 请求和安装权限：所有安装时权限的列表，包括 *normal* 权限和 *signature* 权限。
- **runtime permissions** 运行时权限：所有 *dangerous* 权限的列表。

进行动态分析时：

- 评估该应用是否真的需要所要求的权限。例如：一个需要访问 `android.permission.WRITE_SMS` 的单人游戏，是明显异常的。
- 在许多情况下，应用程序可以选择替代声明权限，例如：
  - 请求 `ACCESS_COARSE_LOCATION` 权限而不是 `ACCESS_FINE_LOCATION`。甚至更好的是根本就不请求权限，而是要求用户输入邮政编码。
  - 调用 `ACTION_IMAGE_CAPTURE` 或 `ACTION_VIDEO_CAPTURE` intent 行为而不是请求 CAMERA 权限。
  - 在与蓝牙设备配对时使用同伴设备配对（Android 8.0（API 级别 26）及以上），而不是声明 `ACCESS_FINE_LOCATION`、`ACCESS_COARSE_LOCATIION` 或 `BLUETOOTH_ADMIN` 权限。
- 使用隐私仪表板（Android 12（API 级别 31）及以上）来验证应用程序如何解释对敏感信息的访问。

要获得特定权限的详细信息，您可以参考 Android 文档。

5.7.2. 注入缺陷测试（MSTG-PLATFORM-2）

5.7.2.1. 概述

Android 应用程序可以通过深度链接（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（通过深度链接或其他 IPC 机制，如 Intent 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、过滤。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。

如果任何应用程序功能已公开，则应检查源代码的以下部分：

- 深度链接。检查测试用例“测试深度链接”以了解进一步的测试方案。
- IPC 机制（Intent, Binder，Android Shared Memory 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场景。
- 用户界面。检查测试案例：测试覆盖攻击

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_Implementat
ion"
    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[] s
electionArgs, 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:
            // 提供 ID 时存在 SQL 注入
            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 过的设备上，可以使用 content 命令从内容提供程序查询数据。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--''"
```

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 `Preferences 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 **PreferenceActivity.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 Context 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 级别 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 context 中使用的 fragment。

在 Android 4.4（API 级别 19）之前的版本上，默认实现返回 true；它将在以后的版本中引发异常。

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 manifest 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, Bu
ndle 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().getDataString());
        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.Mai
nActivity");
i.putExtra(":android:show_fragment","pt.claudio.insecurefragment.MyFragment
");
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 WebView 中的 URL 加载（MSTG-PLATFORM-2）

5.7.4.1. 概述

WebViews 是 Android 内置组件，允许你的 APP 在应用程序中打开网页。除了与移动应用程序相关的威胁外，WebViews 还可能使你的应用程序面临常见的网络威胁（例如 XSS、开放式重定向等）。

测试 WebViews 时要做的最重要的事情之一是确保只能在其中加载受信任的内容。任何新加载的页面都可能具有潜在恶意，可能会尝试利用任何 WebView 绑定进行攻击或尝试仿冒用户。除非你正在开发一个浏览器应用程序，否则你通常希望将加载的页面限制在应用程序的域名中。一个好的做法是防止用户甚至有机会在 WebViews 内输入任何 URL（Android 上的默认设置），也不能浏览受信任外的域名。即使在可信域名内浏览，用户仍有可能遇到并单击指向不可信内容的其他链接（例如，如果页面允许其他用户发表评论）。此外，一些开发人员甚至可能覆盖一些默认行为，这些行为可能对用户有潜在危险。有关更多详细信息，请参阅下面的静态分析部分。

5.7.4.1.1. SafeBrowsing API

为了提供更安全的网络浏览体验，Android 8.1（API 级别 27）引入了 **`SafeBrowsing API`**，它允许您的应用程序检测 URL 是否已被 Google 归类为已知威胁。

默认情况下，WebViews 会向用户显示关于安全风险的警告，并可以选择加载 URL 或停止加载页面。使用 SafeBlowsing API，您可以通过向 SafeBlowsing 报告威胁或执行特定操作（例如每次遇到已知威胁时返回安全状态）来自定义应用程序的行为。请查看 Android 开发者文档以获取使用示例。

您可以使用 SafetyNet 库独立于 WebViews 使用 SafeBrowsing API，该库实现了用于安全浏览网络协议 v4 的客户端。SafetyNet 允许您分析应用程序将加载的所有 URL。您可以使用不同的方案（例如 http、文件）检查 URL，因为 SafeBrowsing 与 URL 方案无关，并且针对 **`TYPE_POTENTIALLY_HARMFUL_APPLICATION`** 和 **`TYPE_SOCIAL_ENGINEERING`** 威胁类型。

5.7.4.1.2. Virus Total API

Virus Total 提供了一个 API，用于分析 URL 和本地文件中的已知威胁。API 参考可在 Virus Total 开发者页面上找到。

> 发送 URL 或文件以检查已知威胁时，请确保它们不包含可能危害用户隐私或暴露应用程序中敏感内容的敏感数据。

5.7.4.2. 静态分析

正如我们之前提到的，应该仔细分析处理页面浏览，尤其是当用户可能能够离开受信任的环境时。 Android 上默认和最安全的行为是让默认 Web 浏览器打开用户可能在 WebView 中单击的

任何链接。 但是，可以通过配置允许浏览请求由应用程序本身处理的 **WebViewClient** 来修改此默认逻辑。 如果是这种情况，请务必搜索并检查以下拦截回调函数：

- **shouldOverrideUrlLoading** 允许您的应用程序通过返回 true 中止加载带有可疑内容的 WebView，或者通过返回 false 允许 WebView 加载 URL。 注意事项：
  - POST 请求不调用此方法。
  - 对于 HTML 或 **<script>** 标签中包含的 XmlHttpRequest、iFrame、“src”属性，不会调用此方法。 相反， **shouldInterceptRequest** 应该处理这个问题。
- **shouldInterceptRequest** 允许应用程序从资源请求中返回数据。 如果返回值为 null，则 WebView 将照常继续加载资源。 否则，使用 **shouldInterceptRequest** 方法返回的数据。 注意事项：
  - 这个回调被各种 URL 方案调用（例如，http(s):、data:、file: 等），而不仅仅是那些通过网络发送请求的方案。
  - **javascript:** 或 **blob:**URL 或通过 **file:///android_asset/** 或 **file:///android_res/** URL 访问的资产不调用此方法。 在重定向的情况下，仅对初始资源 URL 调用，而不是任何后续重定向 URL。
  - 启用安全浏览后，这些 URL 仍会进行安全浏览检查，但开发人员可以使用 **setSafeBrowsingWhitelist** 允许 URL，甚至可以通过 **onSafeBrowsingHit** 回调忽略警告。

正如您所看到的，在测试配置了 WebViewClient 的 WebView 的安全性时需要考虑很多要点，因此请务必通过查看 **WebViewClient** 文档仔细阅读和理解所有要点。

虽然 **EnableSafeBrowsing** 的默认值为 **true**，但某些应用程序可能会选择禁用它。 要验证是否启用了 SafeBrowsing，请检查 AndroidManifest.xml 文件并确保不存在以下配置：

```
<manifest>
    <application>
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
                   android:value="false" />
        ...
    </application>
</manifest>
```

5.7.4.3. 动态分析

动态测试深度链接的一种便捷方法是使用 Frida 或 frida-trace 并在使用应用程序并单击 WebView 中的链接时劫持 **`shouldOverrideUrlLoading`**、**`shouldInterceptRequest`** 方法。确保还劫持其他相关的 Uri 方法，例如 **`getHost`**、**`getScheme`** 或 **`getPath`**，这些方法通常用于检查请求并匹配已知模式或拒绝列表。

5.7.5. 测试深度链接（MSTG-PLATFORM-3）

5.7.5.1. 概述

深度链接是任何方案的 URI，可以将用户直接带到应用中的特定内容。一个应用程序可以通过在 Android Manifest 上添加 *intent* 过滤器来设置深度链接，并从传入的 *intent* 中提取数据，将用户引向正确的 activity。

Android 支持两种类型的深度链接。

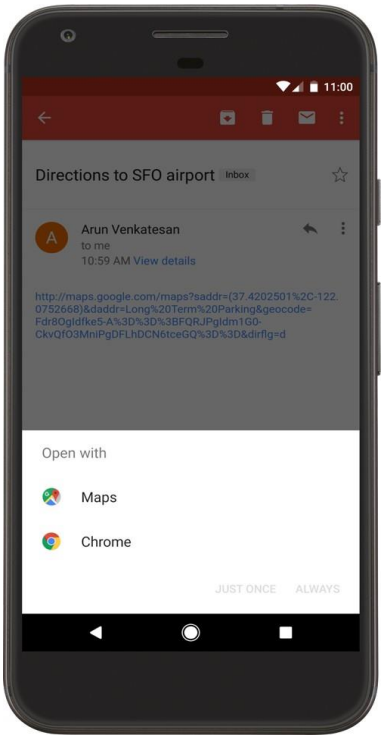
- **自定义 URL 方案**，这是使用任何自定义 URL 方案的深度链接，例如 myapp://（不经操作系统验证）。
- **Android 应用链接**（Android 6.0（API 级别 23）及以上），这是使用 http:// 和 https:// 方案并包含 autoVerify 属性（会触发操作系统验证）的深度链接。

5.7.5.1.1. 深度链接冲突

使用未经验证的深层链接会导致一个重大问题，任何安装在用户设备上的其他应用程序都可以声明并试图处理相同的 *intent*，这就是所谓的深层链接冲突。

任何任意的应用程序都可以声明对属于另一个应用程序的完全相同的深层链接的控制。

在最近的 Android 版本中，这导致向用户显示一个所谓的消歧对话框，要求他们选择应该处理该深层链接的应用程序。用户可能会错误地选择一个恶意的应用程序而不是合法的。



5.7.5.1.2. Android 应用链接

为了解决深层链接的碰撞问题，Android 6.0（API 级别 23）引入了 Android 应用链接，这是基于开发者明确注册的网站 URL 验证深层链接。点击应用链接后，如果该应用已经安装，将立即打开。

与未经验证的深度链接相比，有一些关键的区别：

- 应用链接仅使用 http:// 和 https:// 方案，不允许使用任何其他自定义 URL 方案。
- 应用程序链接需要一个有效域名才能通过 HTTPS 提供数字资产链接文件。
- 应用链接不会遭受深度链接冲突，因为当用户打开它们时它们不会显示消歧对话框。

5.7.5.1.3. 测试深度链接

任何现有的深层链接（包括应用链接）都有可能增加应用的攻击面。这包括许多风险，如链接劫持、敏感功能暴露等。应用程序运行的 Android 版本也会影响风险。

- 在 Android 12（API 级别 31）之前，如果应用程序有任何不可验证的链接，会导致系统不验证该应用程序的所有 Android 应用链接。

- 从 Android 12（API 级别 31）开始，应用程序可以从减少的攻击面中受益。除非目标应用被批准用于该 Web intent 中包含的特定域，否则通用 Web intent 会解析到用户的默认浏览器应用。

所有的深层链接都必须被列举出来，并验证其与网站的联系是否正确。它们执行的操作必须经过良好的测试，特别是所有的输入数据，这些数据应该被认为是不可信的，因此应该始终被验证。

5.7.5.2. 静态分析

5.7.5.2.1. 枚举深度链接

检查 Android Manifest:

你可以通过使用 apktool 解码应用程序和检查 Android Manifest 文件寻找**<intent-filter>**元素，轻松地确定是否定义了深度链接（有或没有自定义 URL 方案）。

- **自定义 URL 方案**：下面的例子用一个自定义的 URL 方案定义了一个深度链接，叫做 myapp://

```
<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>
```

- **深度链接**：下面的例子同时使用 http:// 和 https:// 方案定义了一个深度链接，以及将激活它的主机和路径（在这种情况下，完整的 URL 是 https://www.myapp.com/my/app/path）：

```
<intent-filter>
- - -
<data android:scheme="http" android:host="www.myapp.com"
android:path="/my/app/path" />
<data android:scheme="https" android:host="www.myapp.com"
android:path="/my/app/path" />
</intent-filter>
```

- **应用程序链接**：如果<intent-filter>包含 android:autoVerify="true "的标志，这将导致 Android 系统联系到声明的 android:host，试图访问数字资产链接文件，以验证应用程序链接。**只有在验证成功的情况下，深度链接才能被认为是一个应用程序链接。**

```
<intent-filter   android:autoVerify="true">
```

当列出深度链接时，请记住，同一<intent-filter>内的<data>元素实际上被合并在一起，以说明其组合属性的所有变化。

```
<intent-filter>
  - - -
 <data android:scheme="https" android:host="www.example.com" />
 <data android:scheme="app" android:host="open.my.app" />
</intent-filter>
```

看起来似乎这只支持 https://www.example.com 和 app://open.my.app。然而，它实际上支持：

- https://www.example.com
- app://open.my.app
- app://www.example.com
- https://open.my.app

使用 Dumpsys：

使用 adb 运行以下命令，将显示所有方案：

```
adb shell dumpsys package com.example.package
```

使用 Android "应用程序链接验证 "工具：

使用 Android "应用程序链接验证 "测试脚本，列出所有深度链接（list-all）或只列出应用程序链接（list-applinks）。

```
python3 deeplink_analyser.py -op list-all -apk
~/Downloads/example.apk

.MainActivity

app://open.my.app
app://www.example.com
https://open.my.app
```

```
https://www.example.com
```

5.7.5.2.2. 检查正确的网站关联

即使深度链接包含 android:autoVerify="true "属性，它们也必须被实际验证，才能被认为是应用程序链接。你应该测试任何可能妨碍完全验证的错误配置。

5.7.5.2.2.1 自动验证

使用 Android 的 "应用程序链接验证 "测试脚本来获取所有应用程序链接的验证状态（verify-applinks）。参看示例。

只有在 Android 12（API 级别 31）或更高的系统上：

无论应用程序是否以 Android 12（API 级别 31）为目标，你都可以使用 adb 来测试验证逻辑。这个功能让你可以：

- 手动调用验证过程。
- 重置您设备上目标应用程序的 Android 应用程序链接状态。
- 调用域名验证进程。

你也可以审查验证结果。比如说：

```
adb shell pm get-app-links com.example.package

com.example.package:
  ID: 01234567-89ab-cdef-0123-456789abcdef
  Signatures: [***]
  Domain verification state:
   example.com: verified
   sub.example.com: legacy_failure
   example.net: verified
   example.org: 1026
```

同样的信息可以通过运行 `adb shell dumpsys package com.example.package`（Android 12（API 级别 31）或更高适用）获得。

5.7.5.2.2.2 手动验证

本节详细介绍了验证过程失败或未被实际触发的几个原因，也可能是许多原因中的一个。更多信息请参见 Android 开发者文档和白皮书《衡量 Android 移动深度链接的不安全性》。

检查数字资产链接文件：

- 检查数字资产链接文件是否**缺少**：
  - 尝试在域名的/.known/路径中找到它。例如：https://www.example. com/.known/assetlinks.json
  - 或尝试 https://digitalassetlinks.googleapis.com/v1/statements:list?source.web.site=www.example.com
- 检查**通过 HTTP** 提供的有效数字资产链接文件。
- 检查通过 HTTPS 提供的**无效**的 "数字资产链接 "文件。例如。
  - 该文件包含无效的 JSON。
  - 该文件不包括目标应用程序的软件包。

检查重定向：

为了提高应用程序的安全性，如果服务器设置了重定向，如 http://example.com 到 https://example.com 或 example.com 到 www.example.com，则系统不会为应用程序验证任何 Android 应用程序链接。

检查子域名：

如果一个 intent 过滤器列出了多个具有不同子域的主机，则每个域上必须有一个有效的数字资产链接文件。例如，下面的 intent 过滤器包括 www.example.com 和 mobile.example.com 作为可接受的 intent URL 主机。

```
<application>
 <activity android:name=”MainActivity”>
  <intent-filter android:autoVerify="true">
   <action android:name="android.intent.action.VIEW" />
   <category android:name="android.intent.category.DEFAULT" />
   <category android:name="android.intent.category.BROWSABLE" />
   <data android:scheme="https" />
   <data android:scheme="https" />
   <data android:host="www.example.com" />
   <data android:host="mobile.example.com" />
  </intent-filter>
 </activity>
```

```
</application>
```

为了使深度链接正确注册，必须在 https://www.example.com/.well-known/assetlinks.json 和 https://mobile.example.com/.known/assetlinks.json 发布一个有效的数字资产链接文件。

检查通配符：

如果主机名包括一个通配符（如*.example.com），你应该能够在根域名目录下找到一个有效的数字资产链接文件：https://example.com/.well-known/assetlinks. json。

检查处理程序方法：

即使深度链接被正确验证了，也应该仔细分析处理方法的逻辑。要特别注意被**用来传输数据的深度链接**（由用户或任何其他应用程序在外部控制）。

首先，从 Android Manifest <activity>元素中获取 activity 的名称，该元素定义了目标 <intent-filter>，并搜索 getIntent 和 getData 的用法。在执行逆向工程时，这种定位这些方法的通用办法可用于大多数应用程序，在试图了解应用程序如何使用深度链接和处理任何外部提供的输入数据以及是否可能受到任何形式的滥用时，这是关键。

下面的例子是一个用 jadx 反编译的模板 Kotlin 应用程序的一个片段。通过静态分析，我们知道它支持 `deeplinkdemo://load.html/`作为 `com.mstg.deeplinkdemo.WebViewActivity` 的一部分。

```
// 为简单起见，对片段进行了编辑
public final class WebViewActivity extends AppCompatActivity
   { private ActivityWebViewBinding binding;

  public void onCreate(Bundle savedInstanceState) {
    Uri data = getIntent().getData();
    String html = data == null ? null :
    data.getQueryParameter("html");
    Uri data2 = getIntent().getData();
    String deeplink_url = data2 == null ? null :
    data2.getQueryParameter("url");
    View findViewById = findViewById(R.id.webView);
    if (findViewById != null) {
      WebView wv = (WebView) findViewById;
      wv.getSettings().setJavaScriptEnabled(true);
```

```
if (deeplink_url  != null) {
  wv.loadUrl(deeplink_url);
...
```

你可以简单地跟踪 `deeplink_url` 字符串变量，看到 `wv.loadUrl` 调用的结果。这意味着攻击者完全控制了被加载到 WebView 的 URL（如上面所示已启用 JavaScript）。

同样的 WebView 可能也在渲染一个攻击者控制的参数。在这种情况下，下面的深度链接有效载荷会在 WebView 的内容中触发反射型跨站脚本（XSS）：

```
deeplinkdemo://load.html?attacker_controlled=<svg  onload=alert(1)>
```

但也有许多其他的可能性。请务必查看以下章节，以了解更多关于预期的情况以及如何测试不同的情况：

- 跨站脚本缺陷（MSTG-PLATFORM-2）
- 注入缺陷（MSTG-ARCH-2 和 MSTG-PLATFORM-2）
- 测试对象持久性（MSTG-PLATFORM-8）
- 测试 WebView 中的 URL 加载（MSTG-PLATFORM-2）
- 测试 WebView 中的 JavaScript 执行情况（MSTG-PLATFORM-5）
- 测试 WebView 协议处理程序（MSTG-PLATFORM-6）

此外，我们建议搜索和阅读公开报告（搜索词："deep link*"|"deeplink*" site:https://hackerone.com/reports/）。比如说：

- "[HackerOne#1372667] 能够从深度链接中窃取 bearer 令牌"
- "[HackerOne#401793] 不安全的深度链接导致敏感信息泄露"
- "[HackerOne#583987] 关注动作的 Android 应用程序深度链接导致 CSRF "
- "[HackerOne#637194] 在 Android 应用中可以绕过生物识别安全功能"
- "[HackerOne#341908]通过直接信息深度链接的 XSS"

5.7.5.3. 动态分析

在这里，如果有的话，你将使用来自静态分析的深度链接列表来迭代并确定每个处理方法和处理的数据。你将首先启动一个 Frida 劫持，然后开始调用深度链接。

下面的示例假设了一个接受这个深度链接的目标应用：`deeplinkdemo://load.html`。然而，我们还不知道相应的处理方法，也不知道它可能接受的参数。

[步骤 1]Frida 劫持：

你可以使用 Frida CodeShare 中的脚本 "Android Deep Link Observer "来监控所有触发 `Intent.getData` 调用的深度链接。你也可以把这个脚本作为基础，根据手头的用例，加入你自己的修改。在这个案例中，我们在脚本中加入了堆栈跟踪，因为我们对调用 `Intent.getData` 的方法感兴趣。

[步骤 2] 调用深度链接：

现在你可以使用 adb 和 Activity 管理器（am）调用任何深度链接，这将在 Android 设备中发送 intent。比如说：

```
adb shell am start -W -a android.intent.action.VIEW -d
"deeplinkdemo://load.html/?message=ok#part1"

Starting: Intent { act=android.intent.action.VIEW
dat=deeplinkdemo://load.html/?message=ok } Status: ok
LaunchState:  WARM
Activity: com.mstg.deeplinkdemo/.WebViewActivity
TotalTime: 210
WaitTime: 217
Complete
```

> 当使用 "http/https "模式或其他已安装的应用程序支持相同的自定义 URL 方案时，这可能会触发消除歧义对话框。你可以包括软件包的名称，使其成为一个明确的 intent。

这个调用将记录以下内容：

```
[*] Intent.getData() was called
[*] Activity: com.mstg.deeplinkdemo.WebViewActivity [*] Action:
android.intent.action.VIEW
[*] Data
- Scheme: deeplinkdemo://
- Host: /load.html
- Params: message=ok
- Fragment: part1

[*] Stacktrace:
```

```
android.content.Intent.getData(Intent.java)
com.mstg.deeplinkdemo.WebViewActivity.onCreate(WebViewActivity.kt)
android.app.Activity.performCreate(Activity.java)
- - -
com.android.internal.os.ZygoteInit.main(ZygoteInit.java)
```

在这种情况下，我们精心制作了包括任意参数(?message=ok)和 fragment (#part1)的深度链接。我们仍然不知道它们是否被使用。上面的信息揭示了有用的信息，你现在可以用它来对应用程序进行逆向工程。参见 "检查处理程序方法 "一节，了解你应该考虑的事情。

- 文件：`WebViewActivity.kt`
- 类：`com.mstg.deeplinkdemo.WebViewActivity`
- 方法：`onCreate`

有时，你甚至可以利用你知道的与你的目标应用程序交互的其他应用程序。你可以对应用程序进行逆向工程，（例如，提取所有字符串并过滤那些包含目标深度链接的字符串，即前面案例中的 deeplinkdemo:///load.html），或者将它们作为触发器，同时像前面讨论的那样劫持应用程序。

5.7.7. 测试通过 IPC 暴露敏感功能（MSTG-PLATFORM-4）

5.7.7.1. 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

5.7.7.2. 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 manifest 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 manifest 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，请通过审查源代码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，我们通过两个示例应用给出识别存在漏洞的 IPC 组件的例子：

- "Sieve"
- "Android Insecure Bank"

5.7.7.3. Activities

5.7.7.3.1. 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:win
dowSoftInputMode="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="@str
ing/title_activity_file_select" android:name=".FileSelectActivity" />
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true"
android:exported="true" android:finishOnTaskLaunch="true" android:label="@str
ing/title_activity_pwlist" android:name=".PWList" />
```

5.7.7.3.2. 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。

5.7.7.4. Service

5.7.7.4.1. 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:proces
s=":remote" />
```

5.7.7.4.2. 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("c
om.mwr.example.sieve.PASSWORD"))) {
                            returnVal = 0;
                        } else {
                            returnVal = 1;
                        }
                    } else if (msg.arg1 == AuthService.TYPE_PIN) {
                        responseCode = 41;
                        if (AuthService.this.setPin(returnBundle.getString("c
om.mwr.example.sieve.PIN"))) {
                            returnVal = 0;
                        } else {
                            returnVal = 1;
                        }
                    } else {
                        sendUnrecognisedMessage();
                        return;
                    }
            }
   }
```

5.7.7.5 Broadcast Receiver

5.7.7.5.1. 检查 AndroidManifest

在“Android Insecure Bank”应用中，我们在 manifest 文件中找到了一个<receiver>标识的 broadcast receiver：

```
<receiver android:exported="true" android:name="com.android.insecurebankv2.My
BroadCastReceiver">
    <intent-filter>
        <action android:name="theBroadcast" />
    </intent-filter>
</receiver>
```

5.7.7.5.2. 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接收者。

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");

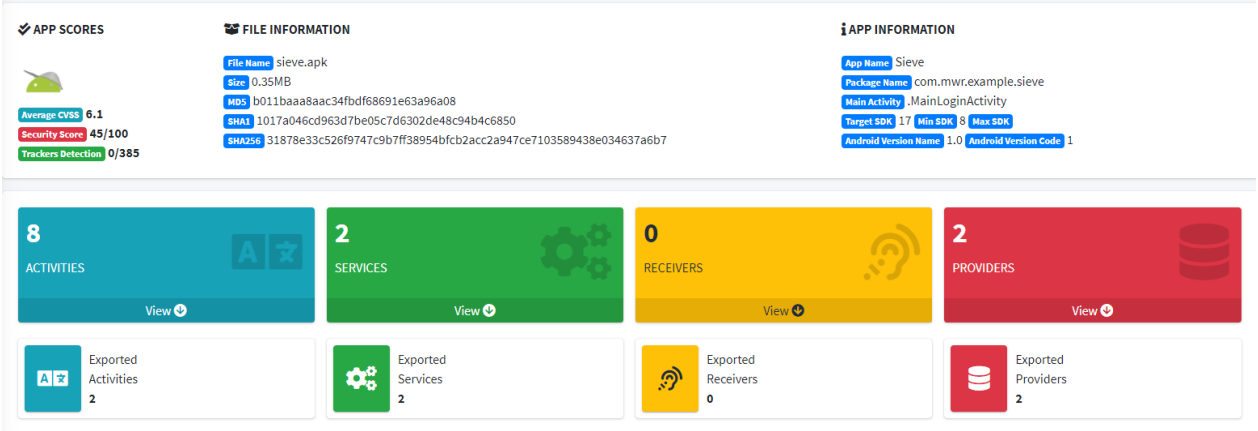
    if (phn != null) {
      try {
                SharedPreferences settings = context.getSharedPreferences(MYP
REFS, Context.MODE_WORLD_READABLE);
                final String username = settings.getString("EncryptedUsername
", null);
                byte[] usernameBase64Byte = Base64.decode(username, Base64.DE
FAULT);
                usernameBase64ByteString = new String(usernameBase64Byte, "UT
F-8");
                final String password = settings.getString("superSecurePasswo
rd", null);
                CryptoClass crypt = new CryptoClass();
                String decryptedPassword = crypt.aesDeccryptedString(passwor
d);
                String textPhoneno = phn.toString();
                String textMessage = "Updated Password from: "+decryptedPassw
ord+" to: "+newpass;
                SmsManager smsManager = SmsManager.getDefault();
                System.out.println("For the changepassword - phonenumber: "+t
```

```
extPhoneno+" 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。

5.7.7.6. 动态分析

您可以使用 MobSF 枚举 IPC 组件。要列出所有导出的 IPC 组件，请上载 APK 文件，组件集合将显示在以下屏幕中：



5.7.7.6.1. Content Provider

“Sieve”应用程序实现了一个存在漏洞的内容提供程序。要列出由 Sieve 应用导出的内容提供程序，请执行以下命令：

```
$ adb shell dumpsys package com.mwr.example.sieve | grep -Po "Provider{[\w\d\
s\./]+}" | sort -u
Provider{34a20d5 com.mwr.example.sieve/.FileBackupProvider}
Provider{64f10ea com.mwr.example.sieve/.DBContentProvider}
```

一旦确定，您可以使用 jadx 对应用程序进行逆向工程，并分析导出内容提供程序的源代码，以识别潜在漏洞。

要识别内容提供程序的相应类别，请使用以下信息：

- 包名: com.mwr.example.sieve.
- 内容提供者类名: DBContentProvider.

当分析类 **com.mwr.example.sieve.DBContentProvider** 时，你将会看到其包含多个 URI：

```
package com.mwr.example.sieve;
...
public class DBContentProvider extends ContentProvider {
    public static final Uri KEYS_URI = Uri.parse("content://com.mwr.example.s
ieve.DBContentProvider/Keys");
    public static final Uri PASSWORDS_URI = Uri.parse("content://com.mwr.exam
ple.sieve.DBContentProvider/Passwords");
...
}
```

使用以下命令使用识别的 URI 调用内容提供程序：

```
$ adb shell content query --uri content://com.mwr.example.sieve.DBContentProv
ider/Keys/
Row: 0 Password=1234567890AZERTYUIOPazertyuiop, pin=1234

$ adb shell content query --uri content://com.mwr.example.sieve.DBContentProv
ider/Passwords/
Row: 0 _id=1, service=test, username=test, password=BLOB, email=t@tedt.com
Row: 1 _id=2, service=bank, username=owasp, password=BLOB, email=user@tedt.co
m

$ adb shell content query --uri content://com.mwr.example.sieve.DBContentProv
ider/Passwords/ --projection email:username:password --where 'service=\"bank\
"'
Row: 0 email=user@tedt.com, username=owasp, password=BLOB
```

现在可以检索所有数据库条目（请参阅输出中以“Row:”开头的所有行）。

5.7.7.6.2. Activity

要列出应用程序导出的 activity，可以使用以下命令并且关注 **activity** 元素：

```
$ aapt d xmltree sieve.apk AndroidManifest.xml
...
E: activity (line=32)
  A: android:label(0x01010001)=@0x7f05000f
```

```
  A: android:name(0x01010003)=".FileSelectActivity" (Raw: ".FileSelectActivit
y")
  A: android:exported(0x01010010)=(type 0x12)0xffffffff
  A: android:finishOnTaskLaunch(0x01010014)=(type 0x12)0xffffffff
  A: android:clearTaskOnLaunch(0x01010015)=(type 0x12)0xffffffff
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
E: activity (line=40)
  A: android:label(0x01010001)=@0x7f050000
  A: android:name(0x01010003)=".MainLoginActivity" (Raw: ".MainLoginActivity
")
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
  A: android:launchMode(0x0101001d)=(type 0x10)0x2
  A: android:windowSoftInputMode(0x0101022b)=(type 0x11)0x14
  E: intent-filter (line=46)
    E: action (line=47)
      A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "androi
d.intent.action.MAIN")
    E: category (line=49)
      A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw: "a
ndroid.intent.category.LAUNCHER")
E: activity (line=52)
  A: android:label(0x01010001)=@0x7f050009
  A: android:name(0x01010003)=".PWList" (Raw: ".PWList")
  A: android:exported(0x01010010)=(type 0x12)0xffffffff
  A: android:finishOnTaskLaunch(0x01010014)=(type 0x12)0xffffffff
  A: android:clearTaskOnLaunch(0x01010015)=(type 0x12)0xffffffff
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
E: activity (line=60)
  A: android:label(0x01010001)=@0x7f05000a
  A: android:name(0x01010003)=".SettingsActivity" (Raw: ".SettingsActivity")
  A: android:finishOnTaskLaunch(0x01010014)=(type 0x12)0xffffffff
  A: android:clearTaskOnLaunch(0x01010015)=(type 0x12)0xffffffff
  A: android:excludeFromRecents(0x01010017)=(type 0x12)0xffffffff
...
```

您可以使用以下属性之一标识导出的 activity：

- 它有一个 intent-filter 子声明
- **android:exported** 属性被设置为 **0xffffffff**

您还可以使用 jadx 在文件 AndroidManifest 中标识导出的 activity。使用上述标准的 xml：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package=
"com.mwr.example.sieve">
...
  This activity is exported via the attribute "exported"
  <activity android:name=".FileSelectActivity" android:exported="true" />
```

```
   This activity is exported via the "intent-filter" declaration
  <activity android:name=".MainLoginActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN"/>
      <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
  </activity>
  This activity is exported via the attribute "exported"
  <activity android:name=".PWList" android:exported="true" />
  Activities below are not exported
  <activity android:name=".SettingsActivity" />
  <activity android:name=".AddEntryActivity"/>
  <activity android:name=".ShortLoginActivity" />
  <activity android:name=".WelcomeActivity" />
  <activity android:name=".PINActivity" />
...
</manifest>
```

枚举易受攻击的密码管理器“Sieve”中的 activity 表明已导出以下 activity：

- .MainLoginActivity
- .PWList
- .FileSelectActivity

使用下面的命令启动一个 activity：

```
# 在不指定动作或类别的情况下启动 activity
$ adb shell am start -n com.mwr.example.sieve/.PWList
Starting: Intent { cmp=com.mwr.example.sieve/.PWList }

# 启动 activity，附带一个动作（-a）和一个类别（-c）。
$ adb shell am start -n "com.mwr.example.sieve/.MainLoginActivity" -a androi
d.intent.action.MAIN -c android.intent.category.LAUNCHER
Starting: Intent { act=android.intent.action.MAIN cat=[android.intent.categor
y.LAUNCHER] cmp=com.mwr.example.sieve/.MainLoginActivity }
```

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。

5.7.7.6.3. 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.AuthServ
ice --msg 6345 7452 1 --extra string com.mwr.example.sieve.PASSWORD "abcdabcd
abcdabcd" --bundle-as-obj
Got a reply from com.mwr.example.sieve/com.mwr.example.sieve.AuthService:
  what: 4
  arg1: 42
  arg2: 0
  Empty
```

5.7.7.6.4. Broadcast Receiver

要列出应用程序导出的广播接收器，您可以使用以下命令并关注 **receiver** 元素：

```
$ aapt d xmltree InsecureBankv2.apk AndroidManifest.xml
...
E: receiver (line=88)
  A: android:name(0x01010003)="com.android.insecurebankv2.MyBroadCastReceiver
" (Raw: "com.android.insecurebankv2.MyBroadCastReceiver")
  A: android:exported(0x01010010)=(type 0x12)0xffffffff
  E: intent-filter (line=91)
    E: action (line=92)
      A: android:name(0x01010003)="theBroadcast" (Raw: "theBroadcast")
E: receiver (line=119)
  A: android:name(0x01010003)="com.google.android.gms.wallet.EnableWalletOpti
mizationReceiver" (Raw: "com.google.android.gms.wallet.EnableWalletOptimizati
onReceiver")
  A: android:exported(0x01010010)=(type 0x12)0x0
  E: intent-filter (line=122)
    E: action (line=123)
      A: android:name(0x01010003)="com.google.android.gms.wallet.ENABLE_WALLE
T_OPTIMIZATION" (Raw: "com.google.android.gms.wallet.ENABLE_WALLET_OPTIMIZATI
ON")
...
```

您可以使用以下属性之一识别导出的广播接收器：

- 它有一个 **intent-filter** 子声明。
- 它的属性 **android:exported** 设置为 **0xffffffff**。

您还可以使用 jadx 使用上述标准在文件 AndroidManifest.xml 中识别导出的广播接收器：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package=
"com.android.insecurebankv2">
...
  This broadcast receiver is exported via the attribute "exported" as well as
 the "intent-filter" declaration
  <receiver android:name="com.android.insecurebankv2.MyBroadCastReceiver" and
roid:exported="true">
    <intent-filter>
      <action android:name="theBroadcast"/>
    </intent-filter>
  </receiver>
  This broadcast receiver is NOT exported because the attribute "exported" is
 explicitly set to false
  <receiver android:name="com.google.android.gms.wallet.EnableWalletOptimizat
ionReceiver" android:exported="false">
    <intent-filter>
      <action android:name="com.google.android.gms.wallet.ENABLE_WALLET_OPTIM
IZATION"/>
    </intent-filter>
  </receiver>
...
</manifest>
```

上面来自易受攻击的银行应用程序 InsecureBankv2 的示例显示，仅导出了名为 **com.android.insecurebankv2.MyBroadCastReceiver** 的广播接收器。

现在您知道有一个导出的广播接收器，您可以深入研究并使用 jadx 对应用程序进行逆向工程。 这将允许您分析源代码以搜索您以后可以尝试利用的潜在漏洞。 导出的广播接收器源码如下：

```
package com.android.insecurebankv2;
...
public class MyBroadCastReceiver extends BroadcastReceiver {
    public static final String MYPREFS = "mySharedPreferences";
    String usernameBase64ByteString;

    public void onReceive(Context context, Intent intent) {
        String phn = intent.getStringExtra("phonenumber");
        String newpass = intent.getStringExtra("newpass");
        if (phn != null) {
            try {
                SharedPreferences settings = context.getSharedPreferences("my
SharedPreferences", 1);
                this.usernameBase64ByteString = new String(Base64.decode(sett
```

```
ings.getString("EncryptedUsername", (String) null), 0), "UTF-8");
                String decryptedPassword = new CryptoClass().aesDeccryptedStr
ing(settings.getString("superSecurePassword", (String) null));
                String textPhoneno = phn.toString();
                String textMessage = "Updated Password from: " + decryptedPas
sword + " to: " + newpass;
                SmsManager smsManager = SmsManager.getDefault();
                System.out.println("For the changepassword - phonenumber: " +
 textPhoneno + " password is: " + textMessage);
                smsManager.sendTextMessage(textPhoneno, (String) null, textMe
ssage, (PendingIntent) null, (PendingIntent) null);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("Phone number is null");
        }
    }
}
```

正如你在源代码中看到的，这个广播接收器需要两个名为 **phonenumber** 和 **newpass** 的参数。 有了这些信息，您现在可以尝试通过使用自定义值向它发送事件来利用此广播接收器：

```
# Send an event with the following properties:
# Action is set to "theBroadcast"
# Parameter "phonenumber" is set to the string "07123456789"
# Parameter "newpass" is set to the string "12345"
$ adb shell am broadcast -a theBroadcast --es phonenumber "07123456789" --es
newpass "12345"
Broadcasting: Intent { act=theBroadcast flg=0x400000 (has extras) }
Broadcast completed: result=0
```

这生成了以下 SMS：

```
Updated Password from: SecretPassword@ to: 12345
```

5.7.7.6.4.1 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)`
```

您还可以使用以下命令来嗅探 intent。 但是，传递的额外的内容不会显示：

```
$ adb shell dumpsys activity broadcasts | grep "theBroadcast"
BroadcastRecord{fc2f46f u0 theBroadcast} to user 0
Intent { act=theBroadcast flg=0x400010 (has extras) }
BroadcastRecord{7d4f24d u0 theBroadcast} to user 0
Intent { act=theBroadcast flg=0x400010 (has extras) }
45: act=theBroadcast flg=0x400010 (has extras)
46: act=theBroadcast flg=0x400010 (has extras)
121: act=theBroadcast flg=0x400010 (has extras)
144: act=theBroadcast flg=0x400010 (has extras)
```

5.7.8. 测试 WebView 中对 JavaScript 执行（MSTG-PLATFORM-5）

5.7.8.1. 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，WebViews 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，WebViews 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。

5.7.8.2. 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 仅从应用 data 目录或可信 web 服务器本地加载。
- 用户不能通过基于用户提供的输入加载不同的资源来定义要加载哪些源

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 级别 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。

5.7.8.3. 动态分析

动态分析取决于操作环境。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；当用户浏览到存在漏洞的功能时，漏洞利用代码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 。
- 只有应用 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 固定”）。

5.7.9. 测试 WebView 协议处理程序（MSTG-PLATFORM-6）

5.7.9.1. 概述

Android URL 有几个默认 schema 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，但也可以从应用 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。

5.7.9.2. 静态分析

检查使用 WebView 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 从安装在系统中的内容提供程序加载内容，默认情况下启用。
- **`setAllowFileAccess`**：允许或禁止 WebView 对文件的访问。面向 Android 10（API 级别 29）及更低版本时默认值为 true，而面向 Android 11（API 级别 30）及更高版本则为 false。 请注意，这仅启用和禁用文件系统访问。 资产和资源访问不受影响，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 reso
urces. This can be prevented by disabling local file system access, which is
enabled by default. You can use the Android WebSettings class to disable loca
l 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。

5.7.9.3. 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。

5.7.10. 测试 Java 接口对象是否通过 WebView 暴露（MSTG-PLATFORM-7）

5.7.10.1. 概述

Android 通过使用 **addJavascriptInterface** 方法为在 WebView 中执行的 JavaScript 调用和使用 Android 应用程序的原生函数（使用 **@JavascriptInterface** 注释）提供了一种方法。这称为 WebView JavaScript 桥或原生桥。

请注意，当您使用 **addJavascriptInterface** 时，您明确授予对在该 WebView 中加载的所有页面的已注册 JavaScript 接口对象的访问权限。 这意味着，如果用户在您的应用程序或域名之外浏览，所有其他外部页面也将有权访问那些 JavaScript 接口对象，如果任何敏感数据通过这些接口暴露，则可能会带来潜在的安全风险。

> 警告：针对 Android 4.2（API 级别 17）以下的 Android 版本的应用程序要格外小心，因为它们容易受到 addJavascriptInterface 实现中的缺陷的影响：一种滥用反射的攻击，当恶意 JavaScript 被注入到 WebView 时会导致远程代码执行。这是因为默认情况下可以访问所有 Java 对象方法（而不仅仅是那些带注释的方法）。

5.7.10.2. 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 级别 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();
        }
}
```

这就是如何从 JavaScript 调用方法 **returnString**，字符串“Secret string”将存储在变量 **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**，考虑以下建议：

- 只有 APK 提供的 JavaScript 才能被允许使用桥，例如通过验证每个桥接 Java 方法的 URL（通过 WebView.getUrl）。
- 不应从远程端点加载任何 JavaScript，例如通过将页面浏览保持在应用程序的域内并在默认浏览器（例如 Chrome、Firefox）上打开所有其他域。
- 如果出于遗留原因（例如必须支持旧设备），至少在应用程序的 manifest 文件中将最低 API 级别设置为 17（**<uses-sdk android:minSdkVersion="17" />**）。

5.7.10.3. 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。

5.7.11. 测试对象持久化（MSTG-PLATFORM-8）

5.7.11.1. 概述

在 Android 上有几种方法可以持久化对象：

5.7.11.1.1. 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。

5.7.11.1.2. 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 Android 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"}
```

5.7.11.1.3. 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 Android 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。

5.7.11.1.4. 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 Android 数据存储章节。Realm 网站提供了一个很好的 ORM Lite 示例。

5.7.11.1.5. 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包方便 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。

5.7.11.1.6. 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。

5.7.11.2. 静态分析

如果使用对象持久化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。更多相关详细信息，请参考“Android 数据存储”章节和“Android 加密 API”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。

您可以随时采取一些通用的补救措施：

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。更多相关详细信息，请参见“Android 加密 API”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认证或授权才能获得密钥。相关更多详细信息，请参见“Android 数据存储”章节。
3. 确保在 actively 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。更多相关详细信息，请参见“反逆向工程”章节。

5.7.11.2.1. 对象序列化

在源代码中搜索以下关键字：

- `import java.io.Serializable`
- `implements Serializable`

5.7.11.2.2. 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, 以及@Until

`Jackson` 在源代码中搜索以下关键字：

- import com.fasterxml.jackson.core
- import org.codehaus.jackson（较旧版本）

5.7.11.2.3. 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“Android 数据存储”和“Android 加密 API”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;

5.7.11.2.4. Parcelable

当通过包含 Parcelable 的包将敏感信息存储在 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）。

5.7.111.3. 动态分析

执行动态分析有几种方式：

1. 对于实际的持久化：使用数据存储章节中描述的技术。
2. 对于基于反射的方法：使用 Xposed 劫持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）。

5.7.12. 测试 WebView 清理（MSTG-PLATFORM-10）

5.7.12.1. 概述

当应用程序访问 WebView 中的任何敏感数据时，清除 WebView 资源是一个关键步骤。这包括本地存储的任何文件、RAM 缓存和任何加载的 JavaScript。

作为一个额外的措施，你可以使用服务器端的头文件，如 no-cache，它可以防止应用程序缓存特定的内容。

> 从 Android 10（API 级别 29）开始，应用程序能够检测到一个 WebView 是否变得无响应。如果发生这种情况，操作系统会自动调用 onRenderProcessUnresponsive 方法。

你可以在 Android 开发者网站上找到更多使用 WebView 的安全最佳实践。

5.7.12.2. 静态分析

有几个地方，应用程序可以删除 WebView 相关的数据。你应该检查所有相关的 API，并尝试完全跟踪数据的删除。

- **WebView API**：

  - **初始化**：一个应用程序可能在初始化 WebView 时，通过使用 `android.webkit.WebSettings` 中的 `setDomStorageEnabled`、`setAppCacheEnabled` 或 `setDatabaseEnabled` 来避免存储某些信息。DOM 存储（用于使用 HTML5 本地存储）、应用程序缓存和数据库存储 API 在默认情况下是禁用的，但应用程序可以将这些设置明确设置为 "true"。
  - **缓存**：Android 的 WebView 类提供了 clearCache 方法，可以用来清除应用程序使用的所有 WebView 的缓存。它接收一个布尔输入参数（`includeDiskFiles`），它将清除所有存储的资源，包括 RAM 缓存。然而，如果它被设置为 false，它将只清除 RAM 缓存。检查源代码中 `clearCache` 方法的用法并验证其输入参数。此外，你也可以检查应用程序是否覆盖了 `onRenderProcessUnresponsive`，以应对 WebView 可能变得无响应的情况，因为 `clearCache` 方法也可能从那里被调用。
- **WebStorage API**：`WebStorage.deleteAllData` 也可以用来清除目前被 JavaScript 存储 API 使用的所有存储，包括 Web SQL 数据库和 HTML5 Web 存储 API。> 一些应用程序需要启用 DOM 存储，以显示一些使用本地存储的 HTML5 网站。这应该被仔细调查，因为这可能包含敏感数据。
- **Cookies**：可以通过使用 CookieManager.removeAllCookies 来删除任何现有的 cookies。
- **文件 API**：在某些目录中适当地删除数据可能并不那么直接，一些应用程序使用一种务实的解决方案，即手动删除已知持有用户数据的选定目录。这可以使用 `java.io.File` API 来完成，比如 java.io.File.deleteRecursively。

示例：

这个 Kotlin 的示例来自开源的 Firefox Focus 应用，显示了不同的清理步骤：

```
override fun cleanup() {
    clearFormData() // 移除当前焦点表单字段的自动完成弹出窗口（如果存在）。注意，
    这只影响自动完成弹出窗口的显示，并不从 WebView 的数据库中删除任何保存的表单数
    据。要做到这一点，请使用 WebViewDatabase#clearFormData。
    clearHistory() clearMatches() clearSslPreferences()
    clearCache(true)

    CookieManager.getInstance().removeAllCookies(null)

    WebStorage.getInstance().deleteAllData() // 清除当前被 JavaScript 存储 API
    使用的所有存储。这包括应用程序缓存、网络 SQL 数据库和 HTML5 网络存储 API。

    val webViewDatabase = WebViewDatabase.getInstance(context)
```

```
    // 这与WebView.clearFormData()有什么区别并不完全清楚。
    @Suppress("DEPRECATION")
    webViewDatabase.clearFormData() // 清除 Web 表单的任何保存数据。
    webViewDatabase.clearHttpAuthUsernamePassword()

    deleteContentFromKnownLocations(context) // 调用
    FileUtils.deleteWebViewDirectory(context)，删除 "app_webview"中的所有内
    容。
  }
```

该函数在 deleteContentFromKnownLocations 中完成了一些额外的**手动**删除文件的工作，它调用了 FileUtils 的函数。这些函数使用 java.io.File.deleteRecursively 方法，从指定的目录中递归地删除文件。

```
private fun deleteContent(directory: File, doNotEraseWhitelist:
Set<String> = emptySet()): Boolean {
  val filesToDelete =
  directory.listFiles()?.filter
  { !doNotEraseWhitelist.contains(it.name) } ?:
  return false return filesToDelete.all
  { it.deleteRecursively() }
}
```

5.7.12.3. 动态分析

打开访问敏感数据的 WebView，然后注销该应用程序。访问应用程序的存储容器，确保所有与 WebView 相关的文件被删除。以下文件和文件夹通常与 WebViews 有关：

- app_webview
- Cookies
- pref_store
- blob_storage
- Session Storage
- Web Data
- Service Worker

5.7.13. 测试覆盖攻击（MSTG-PLATFORM-9）

5.7.13.1. 概述

当恶意应用程序设法将自己置于另一个应用程序之上，而该应用程序仍然像在前台一样正常工作时，就会发生屏幕覆盖攻击。 恶意应用程序可能会创建模仿外观和感觉以及原始应用程序甚至 Android 系统 UI 的 UI 元素。 其目的通常是让用户相信他们一直在与合法应用程序交互，然后尝试提升权限（例如通过授予某些权限）、隐蔽网络钓鱼、捕获用户点击和击键等。

有几种攻击会影响不同的 Android 版本，包括：

- Tapjacking（Android 6.0（API 级别 23）及更低版本）滥用 Android 的屏幕覆盖功能，监听点击并拦截传递给底层 activity 的任何信息。
- Cloak & Dagger 攻击会影响针对 Android 5.0（API 级别 21）至 Android 7.1（API 级别 25）的应用。 他们滥用 SYSTEM_ALERT_WINDOW（“draw on top”）和 BIND_ACCESSIBILITY_SERVICE（“a11y”）权限中的一项或两项，如果应用程序是从 Play 商店安装的，则用户不需要明确授予并且他们甚至不需要通知。
- Toast Overlay 与 Cloak & Dagger 非常相似，但不需要用户授予特定的 Android 权限。它在 Android 8.0（API 级别 26）上被 CVE-2017-0752 修复。

通常，此类攻击是具有某些漏洞或设计问题的 Android 系统版本所固有的。 这使得它们具有挑战性，并且通常几乎不可能阻止，除非应用程序针对安全的 Android 版本（API 级别）进行升级。

多年来，许多已知的恶意软件（如 MazorBot、BankBot 或 MysteryBot）一直在滥用 Android 的屏幕覆盖功能来针对关键业务应用程序，如银行业。 此博客讨论了有关此类恶意软件的更多信息。

5.7.13.2. 静态分析

您可以在 Android 开发者文档中找到一些关于 Android View 安全性的通用指南，请务必仔细阅读。 例如，所谓的触摸过滤是一种常见的防御窃听的方法，它有助于保护用户免受这些漏洞的侵害，通常与我们在本节介绍的其他技术和考虑因素结合使用。

要开始您的静态分析，您可以检查以下方法和属性的源代码（非详尽列表）：

- 重写 **`onFilterTouchEventForSecurity`** 以获得更细粒度的控制并为视图实施自定义安全策略。
- 将布局属性 `android:filterTouchesWhenObscured` 设置为 true 或调用 **`setFilterTouchesWhenObscured`**。

- 检查 FLAG_WINDOW_IS_OBSCURED（从 API 级别 9 开始）或 FLAG_WINDOW_IS_PARTIALLY_OBSCURED（从 API 级别 29 开始）。

某些属性可能会影响整个应用程序，而其他属性可以应用于特定组件。 例如，当业务需要特殊允许覆盖，同时希望保护敏感的输入 UI 元素时，就会出现后者。 开发人员还可能采取额外的预防措施来确认用户的实际意图，这可能是合法的，并将其与潜在的攻击区分开来。

最后一点，请始终记住正确检查应用程序所针对的 API 级别及其含义。 例如，Android 8.0（API 级别 26）对需要 SYSTEM_ALERT_WINDOW（“draw on top”）的应用程序进行了更改。 从这个 API 级别开始，使用 TYPE_APPLICATION_OVERLAY 的应用程序将始终显示在具有其他类型（例如 TYPE_SYSTEM_OVERLAY 或 TYPE_SYSTEM_ALERT）的其他窗口上方。 您可以使用此信息来确保至少在此具体 Android 版本中的此应用不会发生覆盖攻击。

5.7.13.3. 动态分析

以动态方式滥用这种漏洞可能非常具有挑战性并且非常专业，因为它密切依赖于目标 Android 版本。 例如，对于 Android 7.0（API 级别 24）以下的版本，您可以使用以下 APK 作为概念证明来识别漏洞的存在。

- Tapjacking POC：这个 APK 创建了一个简单的覆盖层，位于测试应用程序的顶部。
- Invisible Keyboard：此 APK 在键盘上创建多个覆盖以捕获击键。 这是 Cloak 和 Dagger 攻击中展示的漏洞之一。

5.7.14. 测试强制更新（MSTG-ARCH-9）

从 Android 5.0（API 级别 21）开始，应用以及 Google Play 核心库可以被强制更新了。此机制基于使用 **`AppUpdateManager`**。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 Play 商店的 API 可能会变化，所以其不是那么可靠。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固定时，强制更新对于公钥固定（请参阅“测试网络通信”以了解更多详细信息）确实很有帮助。其次，可以通过强制更新轻松修补漏洞。

请注意，较新版本的应用程序不会解决存在于应用程序通信的后端的安全问题。允许应用不与其通信可能还不够。拥有适当的 API 生命周期管理是这里的关键。同样，当用户没有被强制更新时，不要忘记针对您的 API 测试旧版本的应用程序和/或使用正确的 API 版本控制。

5.7.14.1. 静态分析

下面是一个应用更新示例代码：

```
//Part 1: 检查更新
// 创建管理者实例.
AppUpdateManager appUpdateManager = AppUpdateManagerFactory.create(context);

// 返回一个你用来检查更新的 intent 对象.
Task<AppUpdateInfo> appUpdateInfo = appUpdateManager.getAppUpdateInfo();

// 检查平台是否允许指定类型的更新。
if (appUpdateInfo.updateAvailability() == UpdateAvailability.UPDATE_AVAILABLE
      // 对于一个灵活的更新，使用 AppUpdateType.FLEXIBLE
      && appUpdateInfo.isUpdateTypeAllowed(AppUpdateType.IMMEDIATE)) {



                    //...Part 2: 请求更新
                    appUpdateManager.startUpdateFlowForResult(
                       // 传递'getAppUpdateInfo()'返回的 intent。
                       appUpdateInfo,
                       // 或为灵活更新返回'AppUpdateType.FLEXIBLE'。
                       AppUpdateType.IMMEDIATE,
                       // 当前 activity 生成更新请求。
                       this,
                       // 包括一个请求代码，以便以后监测这个更新请求.
                       MY_REQUEST_CODE);



                    //...Part 3: 检查更新是否成功完成
 @Override
 public void onActivityResult(int requestCode, int resultCode, Intent data) {
   if (myRequestCode == MY_REQUEST_CODE) {
     if (resultCode != RESULT_OK) {
       log("Update flow failed! Result code: " + resultCode);
       // 如果更新被取消或失败，
       // 在强制更新的情况下，你可以要求重新开始更新。
     }
   }
 }

 //..Part 4:
 // 检查更新是否在'onResume()'期间停顿。
//然而，你应该在进入应用程序的所有入口点执行这一检查
```

```
@Override
protected void onResume() {
  super.onResume();

  appUpdateManager
      .getAppUpdateInfo()
      .addOnSuccessListener(
          appUpdateInfo -> {
            ...
            if (appUpdateInfo.updateAvailability()
                == UpdateAvailability.DEVELOPER_TRIGGERED_UPDATE_IN_PROGRESS)
 {
                // 如果应用内更新已经在运行，请恢复更新。
                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，这个属性的使用可以确保用户不进行更新就无法继续使用该应用。如您所见，在示例的第 3 部分中：确保取消或错误会导致重新检查，并且确保用户在不进行关键安全更新时无法继续。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。

5.7.14.2. 动态分析

为了测试是否正确更新：尝试通过开发人员发布的版本或使用第三方应用商店下载存在安全漏洞的旧版本应用程序。 接下来，验证您是否可以在不更新的情况下继续使用该应用程序。 如果出现更新提示，请通过取消提示或以其他方式通过正常的应用程序使用来规避它，以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会停止对易受攻击的后端的调用和/或易受攻击的应用程序版本本身是否被后端阻止。 最后，看看你是否可以使用中间人攻击修改应用程序的版本号，看看后端如何响应这个（例如，它是否被记录下来）。

5.7.15. 参考文献

5.7.15.1. Android 应用程序包和更新

- https://developer.android.com/guide/app-bundle/in-app-updates

5.7.15.2. Android Fragment 注入

- https://www.synopsys.com/blogs/software-security/fragment-injection/
- https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-intohttps://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdffragments.pdf

5.7.15.3. Android 权限文档

- 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(an droid.net.Uri,%20int)
- https://developer.android.com/reference/android/content/Context#checkUriPermission(and roid.net.Uri,%20int,%20int,%20int)
- https://developer.android.com/guide/components/broadcasts#restricting_broadcasts_with_p ermissions
- 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-enabledhttps://developer.android.com/topic/google-play-instant/getting-started/instant-enabled-app-bundleapp-bundle

- https://developer.android.com/topic/google-play-instant/guides/multiple-entry-points
- https://developer.android.com/studio/projects/dynamic-delivery

5.7.15.5. Android 包和即时应用

- 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

5.7.15.5. Android 8 的权限变更

- https://developer.android.com/about/versions/oreo/android-8.0-changes

5.7.15.6. Android WebView 和 SafeBrowsing

- https://developer.android.com/training/articles/security-tips#WebView
- https://developer.android.com/guide/webapps/managing-webview#safe-browsing
- https://developer.android.com/about/versions/oreo/android-8.1#safebrowsing
- https://support.virustotal.com/hc/en-us/articles/115002146549-Mobile-Apps

5.7.15.7. Android 自定义 URL 方案

- https://developer.android.com/training/app-links/
- https://developer.android.com/training/app-links/deep-linking
- https://developer.android.com/training/app-links/verify-site-associations
- https://developers.google.com/digital-asset-links/v1/getting-started
- https://pdfs.semanticscholar.org/0415/59c01d5235f8cf38a3c69ccee7e1f1a98067.pdf

5.7.15.8. Android 应用通知

- https://developer.android.com/guide/topics/ui/notifiers/notifications
- https://developer.android.com/training/notify-user/build-notification
- https://developer.android.com/reference/android/service/notification/NotificationListenerService
- https://medium.com/csis-techblog/analysis-of-joker-a-spy-premium-subscription-bot-on-googleplay-9ad24f044451

5.7.15.9. 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 Intent，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-PLATFORM-10: "WebView 的缓存、存储和加载的资源（JavaScript 等）应该在 WebView 被销毁之前被清除。"
- MSTG-ARCH-9：“存在用于强制执行移动应用程序更新的机制。”

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名 (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，数字签名用于应用程序更新时验证所有者身份，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，它会附加一个带有公钥的证书。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 Google Play 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）。

Android 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。Android 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。 请注意在文章发布的时候只用通过 apksigner 签名才可用。

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 已经为 Android 7.0（API 级别 24）及以上的应用使用 v1 和 v2 方案以及为 Android 9（API 级别 28）及以上的应用使用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，其位于`[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 anch
ors]
(...)
```

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
```

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。

最后但并非最不重要的一点：确保应用程序从未与您的内部测试证书一起部署。

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

android:debuggable 属性在 Android manifest 中定义的 **Application** 元素中，其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:debuggable** 属性设置并查找属性的值：

```
    ...
    <application android:allowBackup="true" android:debuggable="true" androi
d:icon="@drawable/ic_launcher" android:label="@string/app_name" android:theme
="@style/AppTheme">
    ...
```

您可以使用 Android SDK 中的 **aapt** 工具和以下命令行快速检查是否存在 android:debuggable="true"设置：

```
# 如果命令返回 1 表示存在设置
# 正则表达式搜索这一行: android:debuggable(0x0101000f)=(type 0x12)0xffffffff
$ aapt d xmltree sieve.apk AndroidManifest.xml | grep -Ec "android:debuggable
\(0x[0-9a-f]+\)=\(type\s0x[0-9a-f]+\)0xffffffff"
1
```

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

adb 可以用来判断一个应用程序是否可以调试。

使用下面的命令：

```
# 如果命令显示不为 0 则应用程序有调试属性
# 正则搜索这些行:
# flags=[ DEBUGGABLE HAS_CODE ALLOW_CLEAR_USER_DATA ALLOW_BACKUP ]
# pkgFlags=[ DEBUGGABLE HAS_CODE ALLOW_CLEAR_USER_DATA ALLOW_BACKUP ]
$ adb shell dumpsys package com.mwr.example.sieve | grep -c "DEBUGGABLE"
2
$ adb shell dumpsys package com.nondebuggableapp | grep -c "DEBUGGABLE"
0
```

如果应用程序是可调试的，那么执行应用程序命令很简单。在 adb shell 中，在 run-as 之后跟着包名和应用程序命令：

```
$ run-as com.vulnerable.app id
uid=10084(u0_a84) gid=10084(u0_a84) groups=10083(u0_a83),1004(input),1007(lo
g),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_admin),3002(net_bt),3
003(inet),3006(net_bw_stats) context=u:r:untrusted_app:s0:c512,c768
```

Android Studio 还可以用于调试应用程序和验证应用程序是否激活调试。

确定应用程序是否可调试的另一种方法是将 **jdb** 附加到正在运行的进程。如果成功，调试已激活。

以下过程可用于启动与 **jdb** 的调试会话：

1. 使用 **adb** 和 **jdwp**, 标识你想要调试激活应用程序的 PID:

```
$ adb jdwp
2355
16346  <== last launched, corresponds to our application
```

2. 通过使用特定的本地端口，使用 adb 在应用程序进程（指定 PID）和主机之间创建通信通道:

```
# adb forward tcp:[LOCAL_PORT] jdwp:[APPLICATION_PID]
$ adb forward tcp:55555 jdwp:16346
```

3. 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。
- Tutorialspoint 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 the connection to the debugger has been closed"的错误，请结束所有的 adb 会话并开始一个新的会话。

5.8.3. 测试调试符号 (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。

要检查原生二进制文件，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** 二进制文件，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 **visibility** 编译器参数剥离动态符号。添加此参数会导致 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. 测试调试代码和详细错误记录 (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()   // 或使用 .detectAll()检测所有可检测问题
                 .penaltyLog()
                 .build());
         StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
                 .detectLeakedSqlLiteObjects()
                 .detectLeakedClosableObjects()
                 .penaltyLog()
                 .penaltyDeath()
                 .build());
     }
     super.onCreate();
 }
```

建议在带有 **DEVELOPER_MODE** 条件的 **if** 语句中插入策略。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() // 在 LogCat 中记录信息
penaltyDeath() //崩溃的应用程序，在所有启用的处罚结束时运行
penaltyDialog() // 显示对话框
```

请看一下使用 StrictMode 的最佳实践。

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,
- 警告对话框
- 应用程序崩溃

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，因为开发人员可以编写更少的代码就能解决问题。 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 版本之前的 OKHTTP，在这些版本中，利用 TLS 链污染可能绕过 SSL 固定。
- 库无法再维护或几乎无法使用，这就是为什么没有报告和/或修复漏洞的原因。这可能会导致通过库在应用程序中出现错误和/或易受攻击的代码。
- 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查阅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代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，这个问题可能存在于多个层面：当您使用在 webview 中运行 JavaScript 的 webview 时，JavaScript 库也可能存在这些问题。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（OWASP Dependency checker）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：在您的 build.gradle 中添加以下脚本来从 Maven 中央仓库安装该插件：

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.owasp:dependency-check-gradle:3.2.0'
    }
}
```

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** 目录中。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 Sonatype Nexus IQ，Sourceclear, Snyk 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，则适用以下检查方法：

- 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，您都可以人工查找它们是否有 CVE 漏洞。

如果应用程序是高风险的应用程序，那么您最终将手动检查库。在这种情况下，您可以在“测试代码质量”一章中找到对原生代码的特定要求。其次，最好检查是否应用了软件工程的所有最佳实践。

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)来更好地检查依赖关系。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_versio
n}"
}
```

现在，插件调用后，使用以下命令：

```
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 Sonatype Nexus IQ，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，您都可以通过人工查找它们所使用的许可证。

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**）进行处理。**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 intent 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信息来向用户解释问题。
- 确保高风险应用程序处理的所有机密信息总是在执行 **finally** 块的过程中被擦除。

```
byte[] secret;
try{
    //使用 secret
} catch (SPECIFICEXCEPTIONCLASS | SPECIFICEXCEPTIONCLASS2 e) {
    // 处理所有问题
} finally {
    //清除 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 MemoryCleanerO
nCrash();
    private final List<Thread.UncaughtExceptionHandler> mHandlers = new Array
List<>();

    //初始化处理程序并且设置为默认的异常处理程序
    public static void init() {
        S_INSTANCE.mHandlers.add(Thread.getDefaultUncaughtExceptionHandler
());
        Thread.setDefaultUncaughtExceptionHandler(S_INSTANCE);
    }

     //确保您仍然可以在其上添加异常处理程序（例如，ACRA 需要）
    public void subscribeCrashHandler(Thread.UncaughtExceptionHandler handle
r) {
        mHandlers.add(handler);
    }

    @Override
    public void uncaughtException(Thread thread, Throwable ex) {

            //在这处理清理
            //....
            //然后根据上下文向用户显示消息（如果可能）

        for (Thread.UncaughtExceptionHandler handler : mHandlers) {
            handler.uncaughtException(thread, ex);
        }
```

```
    }
}
```

现在必须在您的自定义 **Application** 类（例如： **Application** 的扩展类）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 劫持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 intent、公共提供程序和意外值与应用程序进行交互。
- 篡改网络通信和/或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敏感信息。

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错误。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 helper 类时，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：

- 是否有原生代码部分？如果是这样：请在“常规内存损坏”部分中检查给定的问题。给定 JNI 包装器，.CPP/.H/.C 文件，NDK 或其他原生框架可以很容易地发现原生代码。
- 有 Java 代码或 Kotlin 代码吗？查找序列化/反序列化问题，如 Android 反序列化漏洞简史中所述。

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：

- 如果是原生代码: 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，并使用 Squares 泄漏探测器。
- 检查 Android Studio 的内存配置是否有泄漏。
- 使用 Android Java 反序列化漏洞测试器检查序列化漏洞。

5.8.8. 确保激活了释放安全特性(MSTG-CODE-9)

5.8.8.1. 概述

用于检测二进制保护机制的测试在很大程度上取决于开发应用程序所使用的语言。

一般来说，所有的二进制文件都应该被测试，这包括主应用程序的可执行文件以及所有的库/依赖关系。然而，在 Android 上，我们将重点关注原生库，因为正如我们接下来所看到的，主可执行文件被认为是安全的。

Android 从应用程序的 DEX 文件（如 classes.dex）中优化其 Dalvik 字节码，并生成一个包含原生代码的新文件，通常以.odex、.oat 为扩展名。这个 Android 编译的二进制文件使用 ELF 格式包装，这是 Linux 和 Android 用来包装汇编代码的格式。

应用程序的 NDK 本地库也使用 ELF 格式。

- **PIE（位置独立可执行文件）**：
  - 自 Android 7.0（API 级别 24）以来，PIE 编译被默认为主可执行文件的启用。
  - 随着 Android 5.0（API 级别 21）的到来，对不支持 PIE 的原生库的支持被取消，从那时起，PIE 由链接器强制执行。
- **内存管理：**
  - 垃圾收集将简单地运行在主二进制文件上，对二进制文件本身没有什么可检查的。
  - 垃圾收集并不适用于 Android 原生库。开发者有责任做适当的手动内存管理。请参阅 "内存损坏漏洞（MSTG-CODE-8）"。
- **栈溢出保护**：
  - Android 应用程序被编译为 Dalvik 字节码，这被认为是内存安全的（至少对于缓解缓冲区溢出）。其他框架，如 Flutter，不会使用 stack canary 进行编译，因为 Dart 在案例中缓解了缓冲区溢出。
  - Android 原生库必须启用该功能，但要完全检查该功能可能比较困难。
    - ◆ NDK 库应该会启用，因为这是编译器默认行为。
    - ◆ 其他定制的 C/C++库可能没有启用。

了解更多：

- Android 可执行格式
- Android 运行时(ART)
- Android NDK
- 面向 NDK 开发者的 Android 链接器变化

5.8.8.2. 静态分析

测试应用程序的原生库，以确定它们是否启用了 PIE 和栈溢出保护功能。

你可以使用 radare2 的 rabin2 来获取二进制信息。我们将使用适用于 Android 的 UnCrackable 应用程序级别 4 v1.0 APK 作为一个例子。

所有的原生库必须将 canary 和 pic 都设置为 true。

libnative-lib.so 就属于这种情况：

```
rabin2 -I lib/x86_64/libnative-lib.so | grep -E "canary|pic"
canary  true
pic true
```

libtool-checker.so 却没有:

```
rabin2 -I lib/x86_64/libtool-checker.so | grep -E "canary|pic"
canary  false
pic true
```

在这个例子中，libtool-checker.so 必须重新编译以启用栈溢出保护支持。

5.8.9. 参考文献

5.8.9.1. OWASP 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释放安全功能，例如字节码最小化，堆栈保护，PIE 支持和自动引用计数。”

5.8.9.2. 内存分析参考文献

- Android 反序列化漏洞的简要历史 - https://securitylab.github.com/research/android-deserialization-vulnerabilities
- 避免 Android 内存泄漏的 9 种方法- https://android.jlelse.eu/9-ways-to-avoid-memory-leakshttps://android.jlelse.eu/9-ways-to-avoid-memory-leaks-in-android-b6d81648e35ein-android-b6d81648e35e
- Android 中的内存泄漏模式 - https://android.jlelse.eu/memory-leak-patterns-in-androidhttps://android.jlelse.eu/memory-leak-patterns-in-android-4741a7fcb5704741a7fcb570

5.8.9.3. Android 文档

- 带密钥轮换的 APK 签名方案 - https://developer.android.com/about/versions/pie/android-9.0#apk-key-rotation

5.9. Android 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，我们将研究 Android 逆向中的一些特性和特定于操作系统的工具，并以此作为流程。

Android 为逆向工程师提供了 iOS 所没有的巨大优势。因为 Android 是开源的，所以您可以在 Android 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者模式和旁路加载应用程序。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 字节码和原生代码。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。

您需要至少了解基于 Java 的 Android 环境以及 Android 所基于的 Linux 操作系统和内核。您还需要正确的工具集来处理 Java 虚拟机上运行的字节码和原生代码。

请注意，我们将使用适用于 Android 的 OWASP UnCrackable 应用作为例子，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剧透。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. 逆向工程

逆向工程是将一个应用程序拆分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. 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（或者应用了一些反调试工具的反编译技巧），则逆向工程过程可能会非常耗时且无济于事。这也适用于包含原生代码的应用程序。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。

5.9.1.1.1. 反编译 Java 代码

Java 反汇编代码（smali）：

如果你想检查应用程序的 smali 代码（而不是 Java），你可以在 Android Studio 中点击 Profile 或从 "欢迎屏幕 "中调试 APK 来打开你的 APK（即使你不打算调试它，你也可以看一下 smali 代码）。

或者，您可以使用 apktool 直接从 APK 文件中提取和反汇编资源，并将 Java 字节码反汇编为 Smali。apktool 允许您重新打包，这对于修补和应用对 Android Manifest 等的更改非常有用。

Java 反编译的代码：

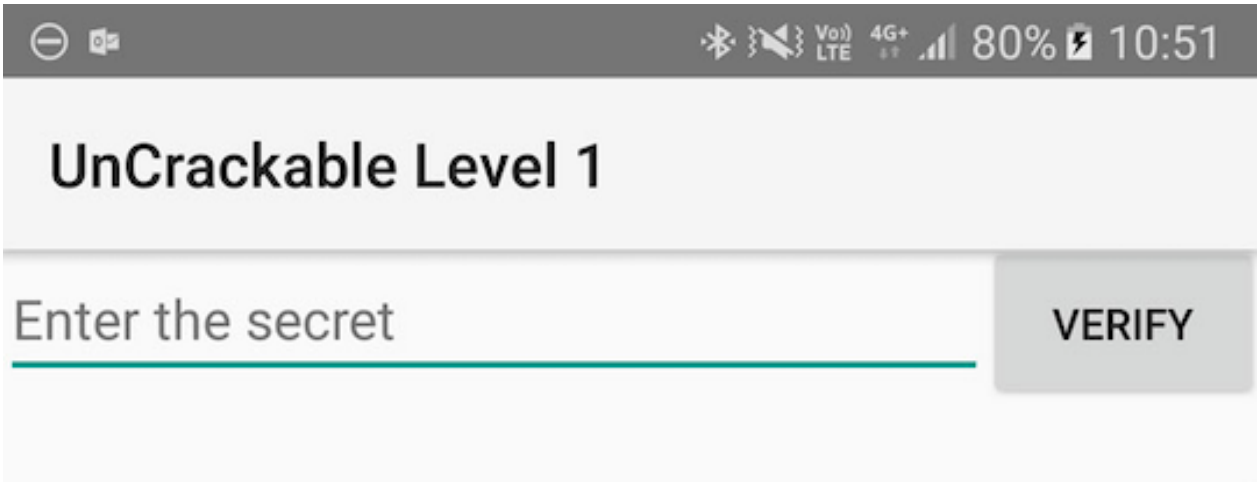
如果你想在 GUI 上直接查看 Java 源代码，只需用 jadx 或 Bytecode Viewer 打开你的 APK。

Android 反编译器则更进一步，会尝试将 Android 字节码转换回 Java 源代码，使其更具可读性。幸运的是，Java 反编译器通常能很好地处理 Android 字节码。上述工具内置、有时甚至整合了流行的免费反编译器，例如：

- JD
- JAD
- jadx
- Procyon
- CFR

也可以使用 apkx 运行 APK，或者使用以前工具中导出的文件在另一个工具（如 IDE）中打开 Java 源代码。

我们将在下面的例子中使用适用于 Android 的 UnCrackable 应用级别 1。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。



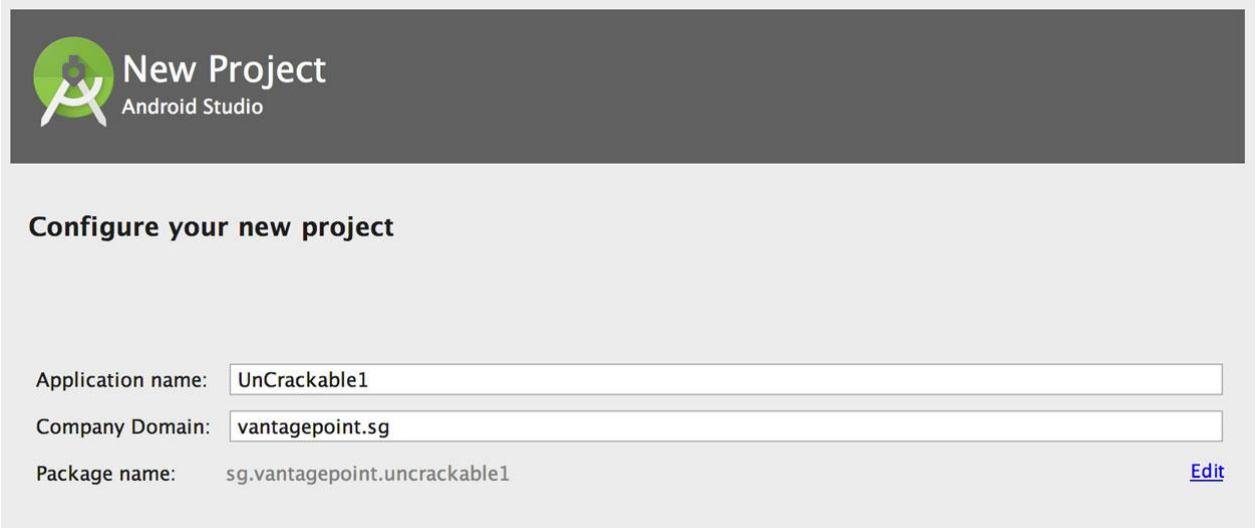
看起来我们应该找到某种密码！

我们正在寻找存储在应用程序中某个位置的秘密字符串，因此下一步是查看应用程序内部。首先，解压缩 APK 文件（`unzip UnCrackable-Level1.apk -d UnCrackable-Level1`）并查看内容。在标准设置中，所有 Java 字节码和应用程序数据都在应用程序根目录（`UnCrackable-Level1/`）中的文件 `classes.dex` 中。该文件符合 Dalvik 可执行文件格式（DEX），这是一种专属于 Android 的 Java 程序打包方式。大多数 Java 反编译器将普通类文件或 JAR 作为输入，因此需要转换 classes.dex 为 JAR 文件。您可以使用 dex2jar 或 enjarify 来实现这一点。

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，它还包含 dex2jar 并自动提取、转换和反编译。使用它运行 APK，您应该在 unrackable-Level1/src 目录中找到反编译的源代码。要查看源代码，一个简单的文本编辑器（最好带有语法高亮显示）就可以了，但将代码加载到 Java IDE 中会使浏览更容易。让我们将代码导入 IntelliJ，IntelliJ 还提供设备调试功能。

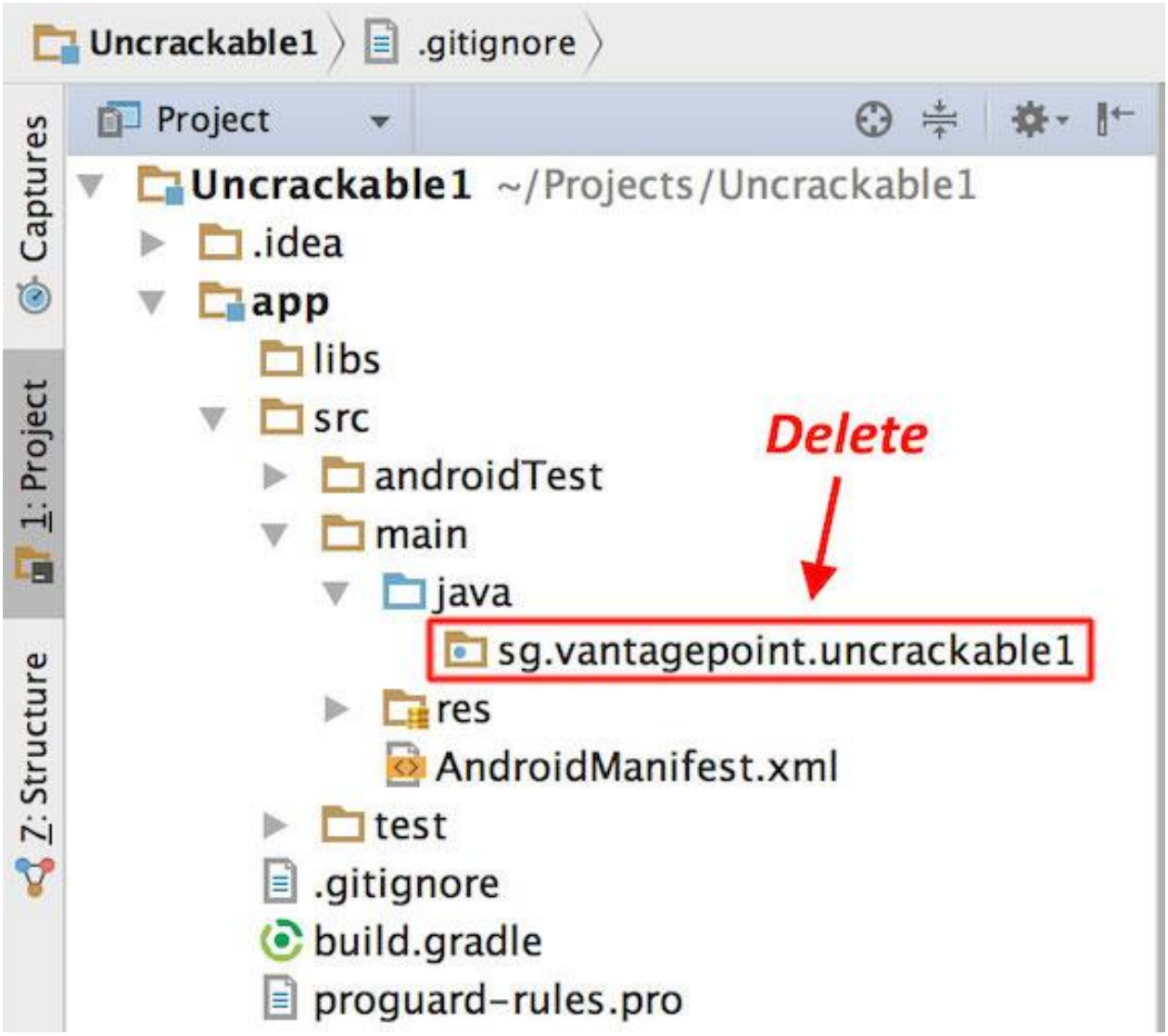
打开 IntelliJ 并在“新建项目（New Project）”对话框的左侧选项卡中选择“Android”作为项目类型。输入“UnTrackable1”作为应用程序名称，“vantagepoint.sg”作为公司名称。这将设置包名“sg.vantagepoint.unrackable1”，它与原始包名匹配。如果以后要将调试器附加到正在运行的应用程序，则使用匹配的包名称非常重要，因为 IntelliJ 使用包名称来标识正确的进程。



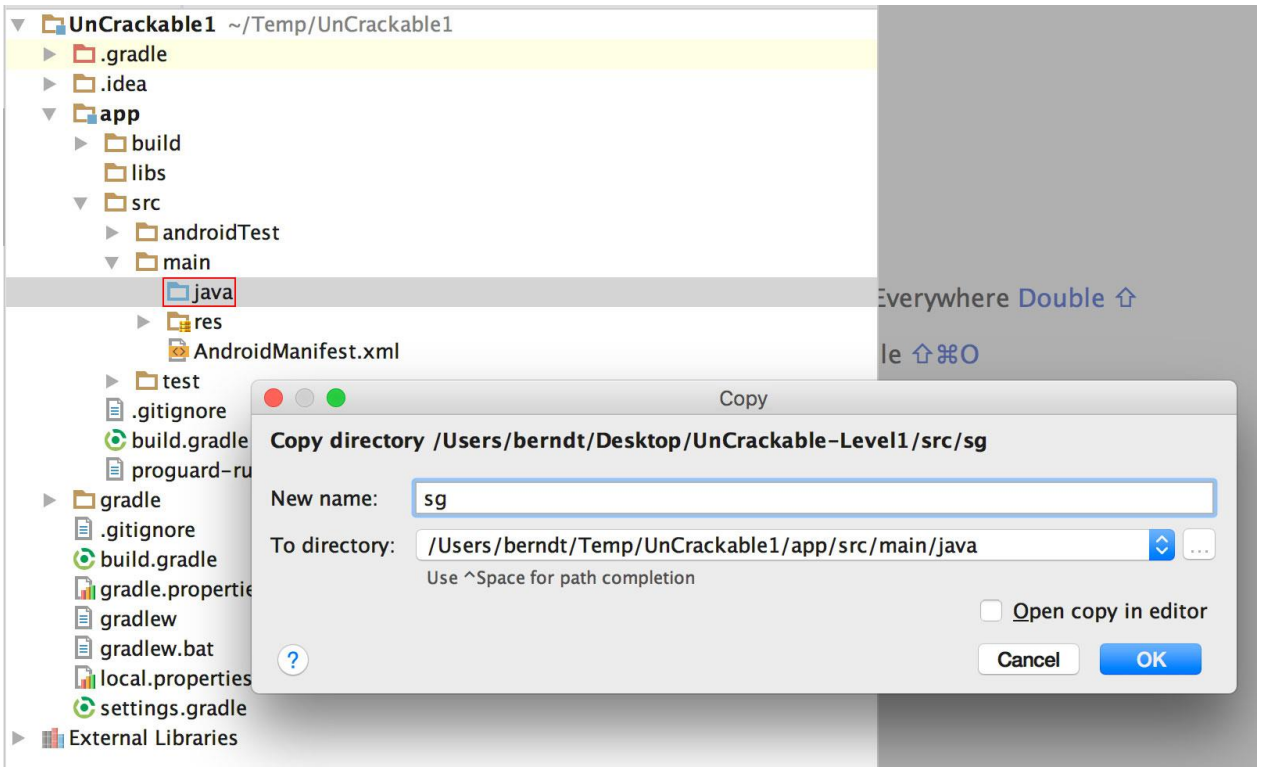
在下一个对话框中，选择任意的 API 版本；实际上您并不想编译项目，所以版本并不重要。点击"next"，选择 "Add no Activity"，然后点击 "finish"。

创建好项目后，展开左侧的 "1: Projec "视图，导航到文件夹 `app/src/main/java`。右击并删除

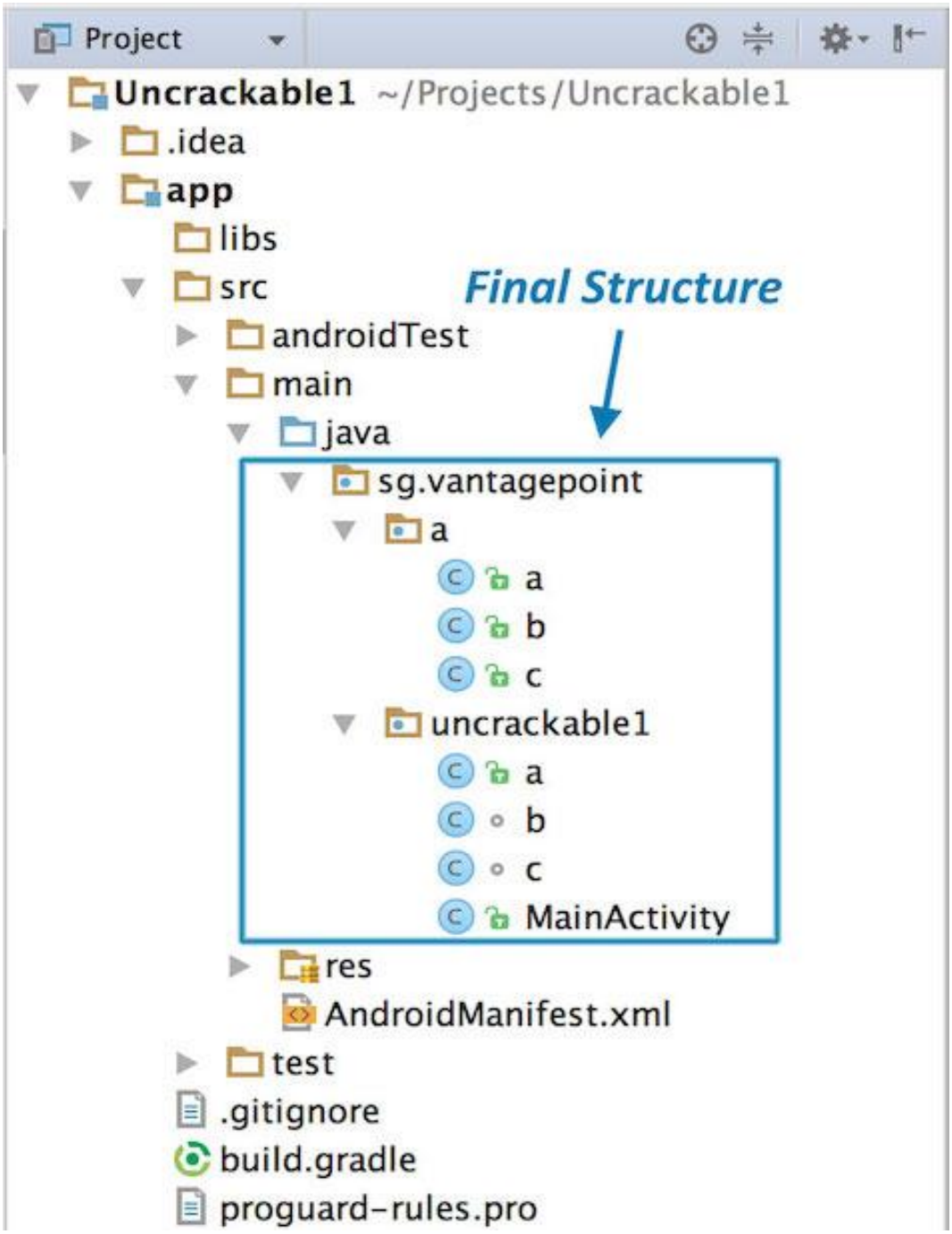
IntelliJ 创建的默认包 "sg.vantagepoint.uncrackable1"。



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。

5.9.1.1.2. 反汇编原生代码

Dalvik 和 ART 都支持 JNI，JNI 为 Java 代码定义了一种与 c/c + + 编写的原生代码交互的方式。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 语言库 (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。

当逆向包含原生代码的 Android 应用程序时，您必须考虑到通过 JNI 建立的 Java 和原生代码之间的关联及相关的数据结构。从逆向角度来看，我们需要了解两个关键的数据结构：`JavaVM` 和 `JNIEnv`。它们都是指向函数表的指针：

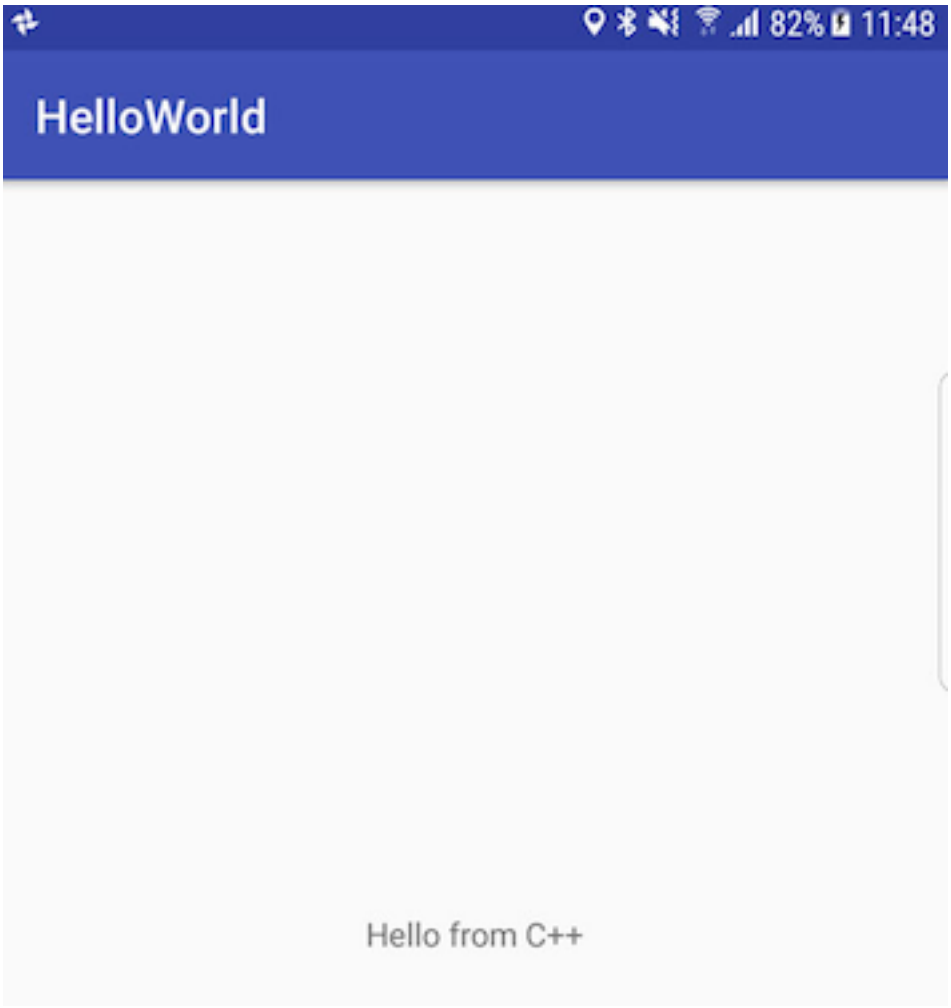
- `JavaVM` 提供了一个接口来调用创建和销毁 JavaVM 的函数。Android 只允许每个进程有一个 JavaVM，这与我们的逆向目的无关。
- `JNIEnv` 提供对大多数 JNI 函数的访问，这些函数可以通过 `JNIEnv` 指针以固定偏移量访问。这个 `JNIEnv` 指针是传递给每个 JNI 函数的第一个参数。我们将在本章后面的一个示例的帮助下再次讨论这个概念。

值得强调的是，分析反汇编的原生代码比分析反编译的 Java 代码更具挑战性。在 Android 应用程序中逆向原生代码时，我们需要一个反汇编程序。

在接下来的例子中，我们将从 OWASP MASTG 源中逆向 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-mastg/raw/master/Samples/Android/01_Hello
World-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，它足够用于展示 JNI 调用的基本原理。



使用 apkx 反编译 APK

```
$ 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 `HelloWord-JNI/src` 目录下。主 activity 可以在文件 `HelloWord-JNI/src/sg/vantagepoint/helloworldjni/MainActivity.java` 中找到。用 onCreate 方法中显示了"Hello World "文本视图:

```
public class MainActivity
extends AppCompatActivity {
    static {
        System.loadLibrary("native-lib");
    }
```

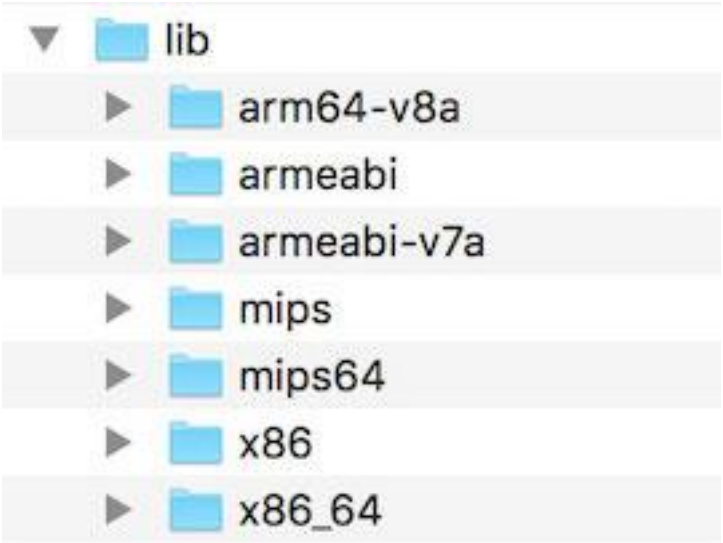
```
    @Override
    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        this.setContentView(2130968603);
        ((TextView)this.findViewById(2131427422)).setText((CharSequence)this.
 \
        stringFromJNI());
    }

    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原生语言实现的。相应的函数在运行时解析，但只有原生库导出加载了预期签名(签名包含包名、类名和方法名)的全局符号时才解析。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_string
FromJNI(JNIEnv *env, jobject)
```

那么，这个函数的原生实现在哪里呢？如果查看 APK 文件解压缩后的 lib 目录，您将看到几个子目录（每个支持的处理器体系结构一个子目录）。每个子目录都包含原生库的一个版本，在本例中是 `libnative-lib.so`。当 `System.loadLibrary` 被调用时，加载程序会根据运行应用程序的设备选择正确的版本。在继续之前，请注意传递给当前 JNI 函数的第一个参数。这与本节前面讨论的 JNIEnv 数据结构相同。



按照上面提到的命名原则，您可以想到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_hello
world_MainActivity_stringFromJNI
```

您也可以使用 radare2 的 rabin2 观察到同样内容:

```
$ rabin2 -s HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so | grep -i Java
003 0x00000e78 0x00000e78 GLOBAL   FUNC   16 Java_sg_vantagepoint_helloworldj
ni_MainActivity_stringFromJNI
```

这是在调用 `stringFromJNI` 原生方法时最终执行的原生函数。

要反汇编代码，可以加载 `libnative-lib.so` 到任何能够理解 ELF 二进制文件的反汇编程序（即任何反汇编程序)。如果应用程序附带用于不同体系结构的二进制文件，理论上你可以选择你最熟悉的体系结构，只要它与反汇编程序兼容。每个版本都从相同的源代码编译，并实现相同的功能。然而，如果您计划稍后在设备上实时调试库，通常明智的做法是选择 ARM 版本。

为了支持较旧和较新的 ARM 处理器，Android 应用程序附带了针对不同应用程序二进制接口（ABI）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。支持以下 ABI：

- 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，用于包含多个 CPU 指令集扩展。
- arm64-v8a：用于基于 ARMv8 的 CPU 的 ABI 支持 AArch64，这是新的 64 位 ARM 体系结构。

大多数反汇编程序都可以处理这些体系结构中的任何一种。下面，我们将在 radare2 和 IDA Pro 中查看 armeabi-v7a 版本（位于 `HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so`)。请参阅下面的“查看已反汇编的原生代码”一节，了解如何检查已反汇编的原生代码。

5.9.1.1.2.1 radare2

要使用 radare2 打开文件，只需运行 `r2 -A HelloWord-JNI/lib/armeabi-v7a/libnative-lib.so`。 “Android 基本测试”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参数 -a 来运行 aaa 命令，以便分析所有引用的代码。

```
$ r2 -A HelloWord-JNI/lib/armeabi-v7a/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，直接从参数 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。记得总是使用 ? 获取帮助或将其加到命令后，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*?]   # 参考'af'和'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 analy
sis
| 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) (a
afe=aef@@f)
| aaF [sym*]           set anal.in=block for all the spaces between flags matc
hing 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 f
unctions
| aav [sat]           find values referencing a specific section or map
```

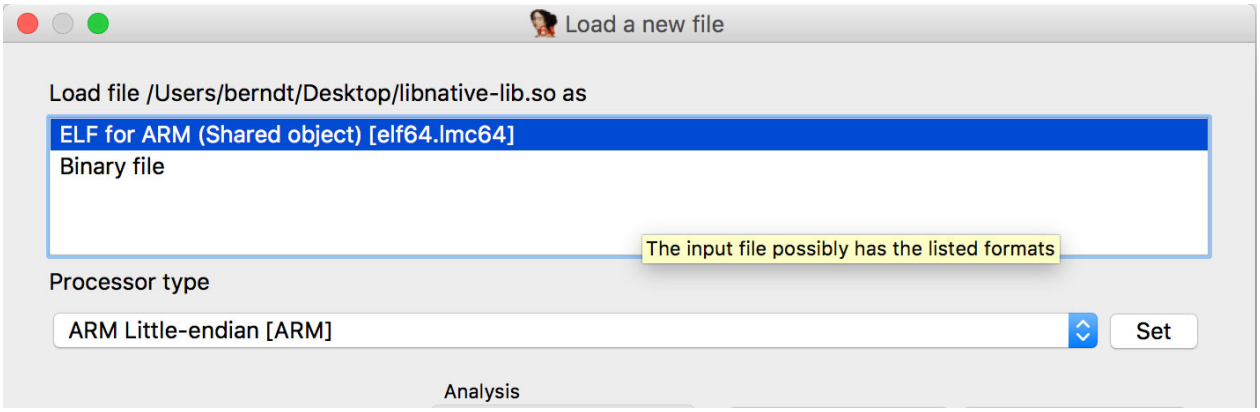
关于 radare2 和其他反汇编程序，比如 IDA Pro，有一件事值得注意。以下引用自 radare2 博客(http://radare.today/)的一篇文章提供了不错的总结。

> 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 的人通常先加载二进制文件，出去煮个咖啡，然后当分析完成后，他们开始才进行人工分析，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 "检查反汇编的原生代码 "一节，了解 radare2 如何帮助我们更快地执行我们的逆向任务。例如：获取一个特定函数的反汇编是一个微不足道的任务，只需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，您需要进行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。您通常会获得 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"一节中的示例进行逆向工程。

5.9.2.1. 基本信息收集

如前几节所述，Android 应用程序可以由 Java/Kotlin 字节码和原生代码组成。在本节中，我们将学习使用静态分析收集基本信息的一些方法和工具。

5.9.2.1.1. 解析字符串

在执行任何类型的二进制分析时，字符串都可以被视为最有价值的起点之一，因为它们提供了上下文关联。例如，错误日志字符串，如“数据加密失败”提示相邻代码可能负责执行某种加密操作。

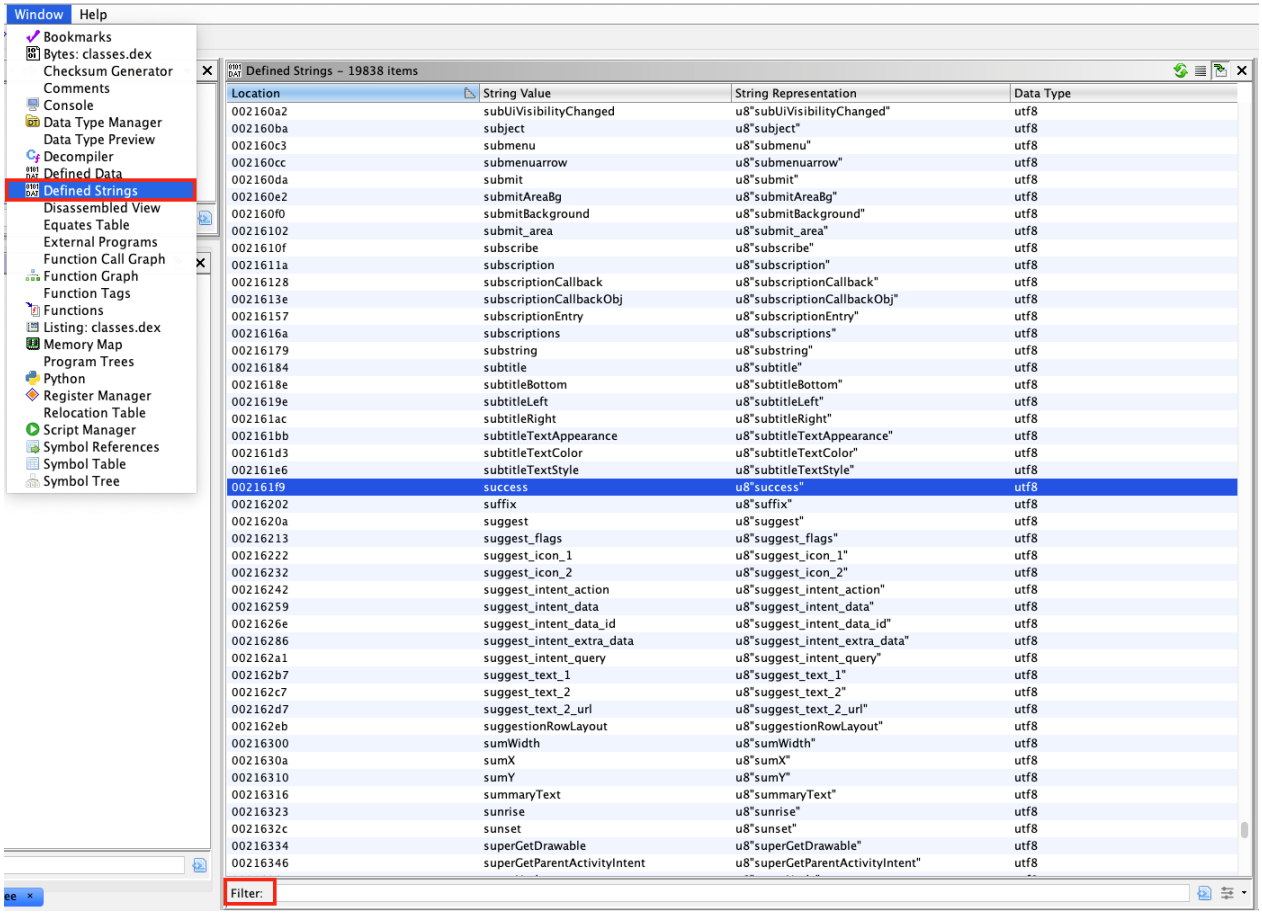
5.9.2.1.1.1 Java 和 Kotlin 字节码

我们已经知道，Android 应用程序的所有 Java 和 Kotlin 字节码都被编译成一个 DEX 文件。每个 DEX 文件都包含一个字符串标识符列表（strings_ids），其中包含每当引用字符串时二进制文件中使用的所有字符串标识符，包括内部命名（例如，类型描述符）或代码引用的常量对象

（例如，硬编码字符串）。您可以使用 Ghidra（基于 GUI）或 Dextra（基于 CLI）等工具简单地转储此列表。

使用 Ghidra，只需加载 DEX 文件并选择菜单中的窗口（**Window**）->定义的字符串（**Defined strings**）即可获得字符串。

> 将 APK 文件直接加载到 Ghidra 中可能会导致不一致。因此，建议通过解压缩 APK 文件并将其加载到 Ghidra 中来提取 DEX 文件。



使用 Dextra，可以使用以下命令转储所有字符串：

```
dextra -S classes.dex
```

Dextra 的输出可以使用标准的 Linux 命令进行操作，例如，使用 grep 搜索某些关键字。

重要的是要知道，使用上述工具获得的字符串列表可能非常大，因为它还包括应用程序中使用的各种类和包名称。浏览完整的列表，特别是大型二进制文件，可能会非常麻烦。因此，建议从基于关键字的搜索开始，仅当关键字搜索没有帮助时才查看列表。一些可以作为良好开始的

通用关键字是-密码（password）、密钥（key）和机密（secret）。当您使用应用程序本身时，可以获得特定于应用程序上下文的其他有用关键字。例如，假设应用程序具有登录表单，您可以记录显示的输入字段占位符或标题文本，并将其用作静态分析的入口点。

5.9.2.1.1.2 原生代码

为了从 Android 应用程序中使用的原生代码中提取字符串，可以使用 Ghidra 或 Cutter 等 GUI 工具，也可以使用基于 CLI 的工具，如 Unix 实用程序 *strings*（`strings <path_to_binary>`）或 radare2 的 rabin2（`rabin2 -zz <path_to_binary>`）。使用基于 CLI 的工具时，可以利用其他工具，如 grep（例如与正则表达式结合使用），进一步过滤和分析结果。

5.9.2.1.2. 交叉引用

5.9.2.1.2.1 Java 和 Kotlin

有许多逆向工具支持检索 Java 交叉引用。对于许多具有 GUI 界面的工具，通常通过右键单击所需的函数并选择相应的选项来完成，例如，在 Ghidra 中显示引用（**Show References to**）或 jadx 中的查找。

5.9.2.1.2.2 原生代码

与 Java 分析类似，您还可以使用 Ghidra 分析原生库，并通过右键单击所需函数并选择显示引用（**Show References to**）来获取交叉引用。

5.9.2.1.3. API 使用

Android 平台为应用程序中的常用功能提供了许多内置库，例如加密、蓝牙、NFC、网络或位置库。确定应用程序中是否存在这些库可以为我们提供有关其性质的宝贵信息。

例如，如果应用程序正在导入 `javax.crypto.Cipher`，这表示应用程序将执行某种加密操作。幸运的是，加密调用在本质上是非常标准的，也就是说，需要按照特定的顺序调用它们才能正常工作，这些知识在分析加密 API 时会很有帮助。例如，通过查找 `Cipher.getInstance` 函数，我们可以确定正在使用的加密算法。使用这种方法，我们可以直接分析加密资产，这在应用程序中通常非常关键。有关如何分析 Android 加密 API 的更多信息，请参阅“Android 加密 API”一节。

类似地，上述方法可用于确定应用程序在何处以及如何使用 NFC。例如，使用基于主机的卡模拟执行数字支付的应用程序必须使用 `android.nfc` 包。因此，NFC API 分析的一个很好的切入点是查阅 Android 开发人员文档，从而获得一些好想法，并开始从 `android.nfc.cardemulation.HostApduService` 类中搜索诸如 `processCommandApdu` 之类的关键函数。

5.9.2.1.4. 网络通信

您可能遇到的大多数应用程序都连接到远程站点。即使在执行任何动态分析（例如流量捕获和分析）之前，您也可以通过枚举应用程序应该与之通信的域来获得一些初始输入或入口点。

通常，这些域将在应用程序的二进制文件中以字符串的形式出现。实现这一目标的一种方法是使用自动化工具，如 APKEnum 或 MobSF。或者，您可以使用正则表达式 grep 查找域名。为此，您可以直接以应用程序二进制文件为目标，或者对其进行反向工程，并以反汇编或反编译代码为目标。后一个选项有一个明显的优势：它可以为您提供 **context**，因为您可以看到每个域名正在哪个 **context** 中使用（例如类和方法）

从这里开始，您可以使用这些信息获得更多了解，了解这些可能会在以后的分析过程中用到，例如，您可以将域名与固定的证书或网络安全配置文件相匹配，或者对域名执行进一步的侦察，以了解更多有关目标环境的信息。在评估应用程序时，检查网络安全配置文件很重要，因为（不太安全的）调试配置常常会被错误地推送最终版本中。

安全连接的实现和验证可能是一个复杂的过程，需要考虑许多方面。例如，许多应用程序使用 HTTP 以外的其他协议，如 XMPP 或明文 TCP 数据包，或执行证书固定以阻止 MITM 攻击，但不幸的是，其实现中存在严重的逻辑错误或固有的错误安全网络配置。

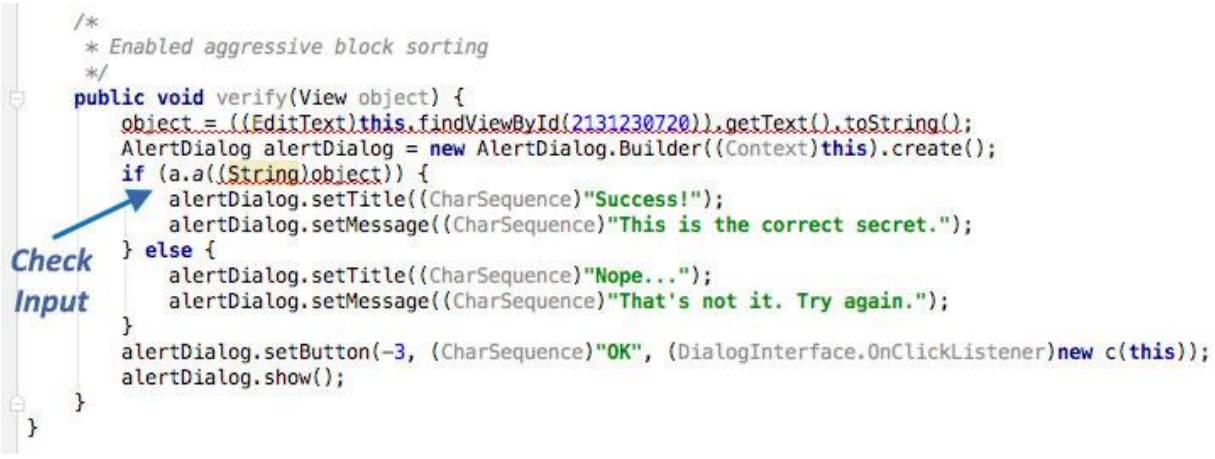
请记住，在大多数情况下，仅使用静态分析是不够的，与动态分析方案相比，静态分析甚至可能变得非常低效，因为动态分析方案将获得更可靠的结果（例如，使用拦截代理）。在本节中，我们只是稍微触及了表面，请参阅“Android 基本安全测试”一章中的“基本网络监控/嗅探”一节，并查看“Android 网络 API”一章中的测试用例。

5.9.2.2. 人工（逆向）代码检查

5.9.2.2.1. 审核反编译 Java 代码

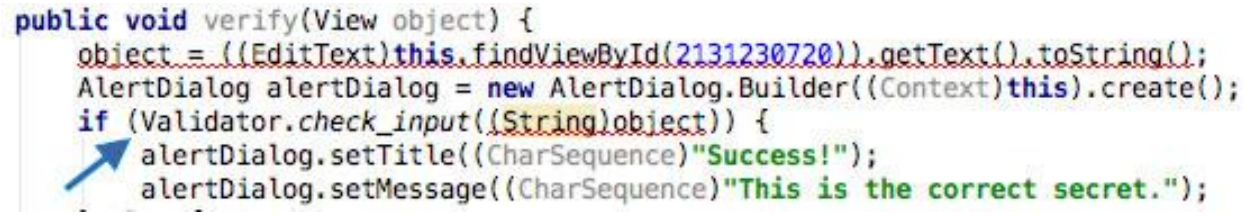
按照 "反编译 Java 代码 "中的例子，我们假设您已经在 IntelliJ 中成功反编译并打开了适用于 Android 的 UnCrackable 应用级别 1。一旦 IntelliJ 对代码进行了索引，您就可以像浏览其他 Java 项目一样浏览它。请注意，许多反编译的包、类和方法都有奇怪的单字母名称；这是因为字节码在构建时已经用 ProGuard 进行了" 最小化"。这是一种基本的混淆方法，它使字节码变得更难读，但对于像这样一个相当简单的应用程序，它不会让您感到头痛。然而，当您分析一个更复杂的应用程序时，它可能会变得相当烦人。

在分析混淆代码时，边分析边注释类名、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 "verify"按钮时，就会调用方法 `verify`。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。




```
/*
 * 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.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}
}
```

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。使用相同方法将静态方法 a 重命名为 `check_input`。



```
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (Validator.check_input((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"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。

接下来，在check_input方法上按住Ctrl键并单击（或在Mac上按住Command键并单击）。这将带您进入方法定义。反编译方法如下所示：

```
    public static boolean check_input(String string) {
        byte[] arrby = Base64.decode((String) \
        "5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);
        byte[] arrby2 = new byte[]{};
        try {
            arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d80
6cf50473cc"), arrby);
            arrby2 = arrby;
        }sa
        catch (Exception exception) {
            Log.d((String)"CodeCheck", (String)("AES error:" + exception.getM
essage()));
        }
        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。我们稍后会重温适用于 Android 的 UnCrackable 应用级别 1 的内容，来展示如何操作(例如在调试章节)，所以先不要删除项目！

5.9.2.2.2. 审查反汇编后的原生代码

按照“反汇编原生代码”的示例，我们将使用不同的反汇编程序来检查反汇编的原生代码。

5.9.2.2.2.1 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/armeabi-v7a/libnative-lib.so
...
[0x00000e3c]> is~Java
003 0x00000e78 0x00000e78 GLOBAL   FUNC   16 Java_sg_vantagepoint_helloworldj
ni_MainActivity_stringFromJNI
```

该方法可在地址 0x00000e78 处找到。要显示其反汇编，只需运行以下命令：

```
[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      ; [0xe
84:4]=0xf3c aav.0x00000f3c
│           0x00000e7c      d2f89c22       ldr.w r2, [r2, 0x29c]
│           0x00000e80      7944           add r1, pc                  ; "Hel
lo 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 的字符串模拟。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 的寻址。我们这样做是为了使接下来的命令应用于这个地址。
- pdf 是指显示反汇编的函数。

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/armeabi-v7
a/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]        ; [0xe
84:4]=0xf3c
│           0x00000e7c      d2f89c22       ldr.w r2, [r2, 0x29c]
│           0x00000e80      7944           add r1, pc                  ; "Hel
lo from C++" section..rodata
└           0x00000e82      1047           bx r2
```

请注意，在这种情况下，我们不是以 -A 参数不运行 aaa 作为开始。相反，我们只是告诉 radare2
使用分析函数 `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。

通过使用 r2ghidra-dec，可以进一步改进工作流程，r2ghidra-dec 是在 radare2 中深度集成 Ghidra 反编译器。r2ghidra-dec 生成反编译的 C 代码，这有助于快速分析二进制文件。

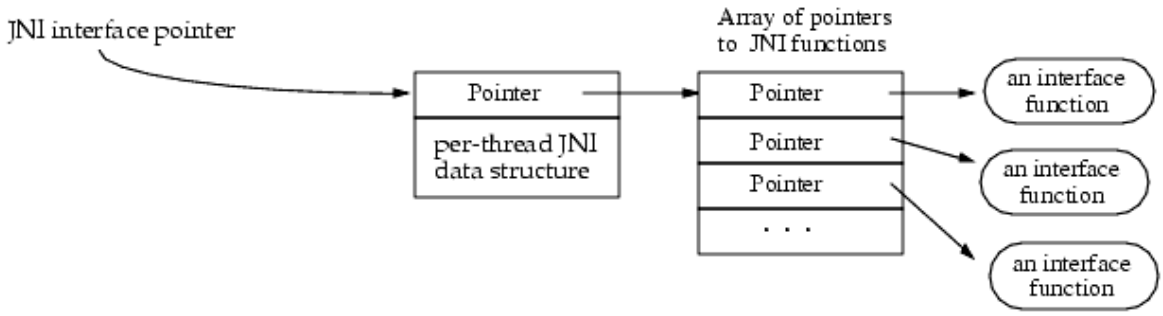
5.9.2.2.2.2 IDA Pro

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（Functions） "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。

```
CODE16


EXPORT Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
LDR             R2, [R0]
LDR             R1, =(aHelloFromC - 0xE80)
LDR.W           R2, [R2,#0x29C]
ADD             R1, PC  ; "Hello from C++"
BX              R2
; End of function Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
```

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。接口指针就是指向指针的指针。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，并允许原生函数与 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 NewStringUTF 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：

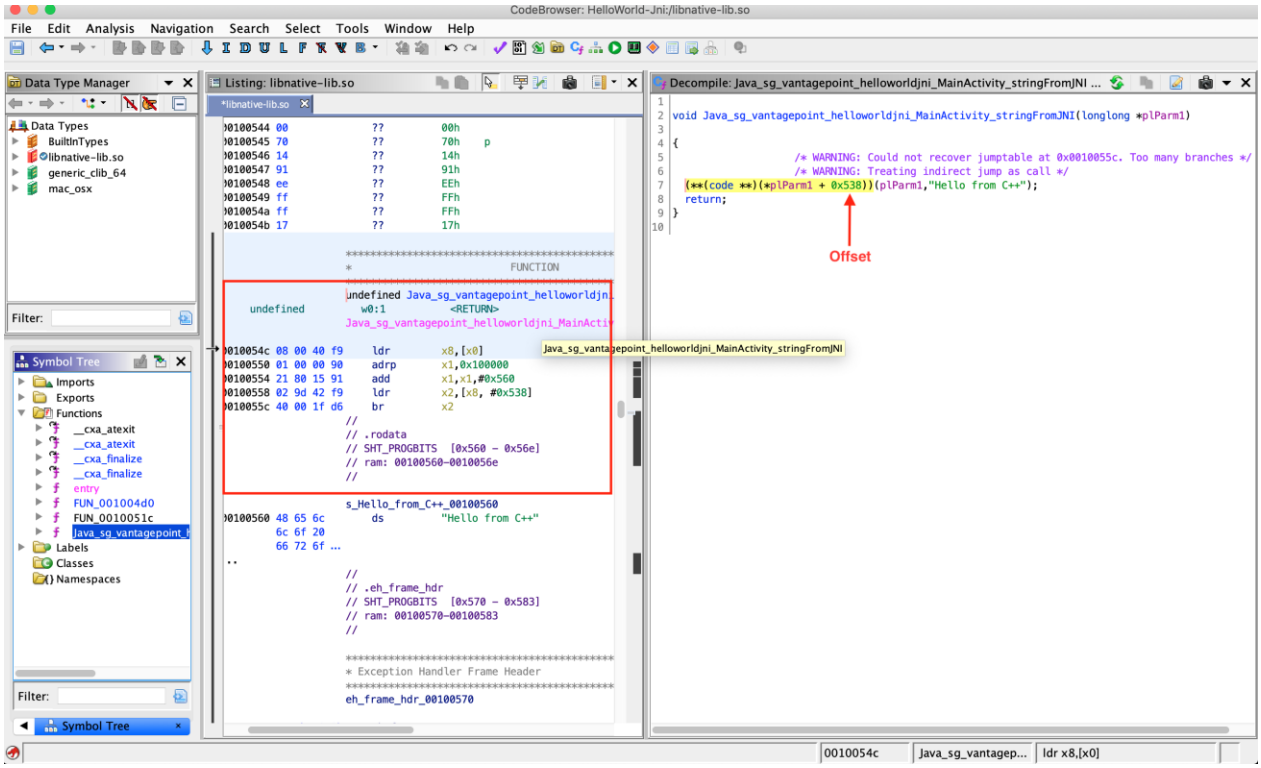
```
BX    R2
```

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。

5.9.2.2.2.3 Ghidra

在 Ghidra 中打开库后，我们可以看到在“函数（**Functions**）”下的符号树（**Symbol Tree**）面板中看到所有定义函数。当前应用程序的原生库相对较小。有三个用户定义的函数：FUN_001004d0, FUN_0010051c,和 `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI`。其他符号不是用户定义的，是为正常运行而生成的共享库。函数 `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI` 中的指令已在前面的章节中详细讨论。在本节中，我们可以研究函数的反编译。

在当前函数中，有一个对另一个函数的调用，该函数的地址是通过访问 JNIEnv 指针（plParm1）中的偏移量获得的。这一逻辑也已在上面用图表演示过。反编译器(**Decompiler**)窗口中显示了反汇编函数的相应 C 代码。这段反编译的 C 代码使理解正在进行的函数调用变得更加容易。由于此函数很小且非常简单，因此反编译输出非常准确，在处理复杂函数时，这可能会发生剧烈变化。



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。

- Androbugs
- JAADAS
- MobSF

- QARK

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。
- 输入验证不足，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. 未 Root 设备动态分析

未 root 设备的好处是可以复制一个应用程序要运行的环境：

多亏了 objection 这样的工具，你可以修补应用程序以测试它，就像在一个 root 过的设备上一样（当然是被限制在那一个应用程序中）。要做到这一点，你必须执行一个额外的步骤：修补 APK 以包含 Frida 小工具库。

现在你可以使用 objection 来动态分析未 root 设备上的应用程序。

下面的命令总结了如何使用 objection 修补应用并开始动态分析，以适用于 Android 的 UnCrackable 应用级别 1 为例：

```
# 下载 Uncrackable APK
$ wget https://raw.githubusercontent.com/OWASP/owasp-mastg/master/Crackmes/Android/Level_01/UnCrackable-Level1.apk
# 使用 Frida Gadget 修补 APK
$ objection patchapk --source UnCrackable-Level1.apk
# 在 android 手机上运行修改过的 APK
$ adb install UnCrackable-Level1.objection.apk
# 运行手机后，objection 会通过 APK 发现运行的 frida-server
$ objection explore
```

5.9.3.2. 基本信息收集

如前所述，Android 运行在经过修改的 Linux 内核之上，并保留了 Linux 的 proc 文件系统（procfs），其挂载在/proc。Procfs 提供了系统上运行的进程的基于目录的视图，提供了有关进程本身、其线程和其他系统范围诊断的详细信息。Procfs 可以说是 Android 系统上最重要的文件系统之一，在 Android 系统中，许多操作系统原生工具都依赖它作为信息源。

为了减小大小许多命令行工具没有随 Android 固件一起提供，但可以使用 BusyBox 轻松安装在 root 过的设备上。我们还可以使用 cut, grep, sort 等命令创建自己的自定义脚本，以解析 proc 文件系统信息。

在本节中，我们将直接或间接使用 procfs 中的信息来收集有关正在运行的进程的信息。

5.9.3.2.1. 打开文件

您可以使用 lsof 的参数-p <pid>来返回指定进程的打开文件列表。有关更多选项，请参阅手册页。

```
# lsof -p 6233
COMMAND     PID       USER   FD      TYPE             DEVICE  SIZE/OFF
NODE NAME
.foobar.c  6233     u0_a97  cwd       DIR               0,1         0
   1 /
.foobar.c  6233     u0_a97  rtd       DIR               0,1         0
   1 /
.foobar.c  6233     u0_a97  txt       REG             259,11     23968
 399 /system/bin/app_process64
.foobar.c  6233     u0_a97  mem   unknown
     /dev/ashmem/dalvik-main space (region space) (deleted)
.foobar.c  6233     u0_a97  mem       REG              253,0   2797568    114
6914 /data/dalvik-cache/arm64/system@framework@boot.art
.foobar.c  6233     u0_a97  mem       REG              253,0   1081344    114
6915 /data/dalvik-cache/arm64/system@framework@boot-core-libart.art
...
```

在上述输出中，与我们最相关的字段是：

- NAME: 文件路径
- TYPE: 文件类型，例如，文件是一个目录或常规文件

在监控使用混淆或其他反逆向工程技术的应用程序时，这对于发现异常文件非常有用，而无需逆向代码。例如，应用程序可能正在对数据执行加密解密，并将其临时存储在文件中。

5.9.3.2.2. 建立连接

您可以在/proc/net 中找到系统范围内的网络信息，或者只需检查/proc/<pid>/net 目录（出于某些原因，进程无关的）。这些目录中存在多个文件，从测试人员的角度来看，tcp、tcp6 和 udp 可能被认为是相关的。

```
# cat /proc/7254/net/tcp
sl  local_address rem_address   st tx_queue rx_queue tr tm->when retrnsmt   u
id  timeout inode
...
69: 1101A8C0:BB2F 9A447D4A:01BB 01 00000000:00000000 00:00000000 00000000 100
93        0 75412 1 0000000000000000 20 3 19 10 -1
70: 1101A8C0:917C E3CB3AD8:01BB 01 00000000:00000000 00:00000000 00000000 100
93        0 75553 1 0000000000000000 20 3 23 10 -1
71: 1101A8C0:C1E3 9C187D4A:01BB 01 00000000:00000000 00:00000000 00000000 100
93        0 75458 1 0000000000000000 20 3 19 10 -1
...
```

在上述输出中，与我们最相关的字段是：

- rem_address：远程地址和端口号对（十六进制表示）。
- tx_queue 和 rx_queue：在内核内存使用方面的传出和传入数据队列。这些字段指示连接的使用情况。
- uid：包含 socket 创建者的有效 UID。

另一种选择是使用 netstat 命令，该命令还以更可读的格式为整个系统提供有关网络活动的信息，并且可以根据我们的要求轻松过滤。例如，我们可以通过 PID 轻松过滤：

```
# netstat -p | grep 24685
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
   PID/Program Name
tcp        0      0 192.168.1.17:47368      172.217.194.103:https   CLOSE_WAI
T  24685/com.google.android.youtube
tcp        0      0 192.168.1.17:47233      172.217.194.94:https    CLOSE_WAI
T  24685/com.google.android.youtube
tcp        0      0 192.168.1.17:38480      sc-in-f100.1e100.:https ESTABLISH
ED 24685/com.google.android.youtube
tcp        0      0 192.168.1.17:44833      74.125.24.91:https      ESTABLISH
ED 24685/com.google.android.youtube
tcp        0      0 192.168.1.17:38481      sc-in-f100.1e100.:https ESTABLISH
ED 24685/com.google.android.youtube
...
```

netstat 输出显然比读取/proc/<pid>/net 更便于用户使用。与之前的输出类似，我们最相关的字段如下：

- 外部地址（Foreign Address）：远程地址和端口号对（端口号可以替换为与端口相关的协议的已知名称）。
- Recv-Q 和 Send-Q：与接收和发送队列相关的统计信息。指示连接的使用情况。
- 状态（State）：socket 的状态，例如，如果套接字（socket）处于活动使用（已建立 ESTABLISHED）或关闭（已关闭 CLOSED）状态。

5.9.3.2.3. 加载原生库

文件/proc/<pid>/maps 包含当前映射的内存区域及其访问权限。使用该文件，我们可以获得在该进程中加载的库的列表。

```
# cat /proc/9568/maps
12c00000-52c00000 rw-p 00000000 00:04 14917                              /dev
/ashmem/dalvik-main space (region space) (deleted)
6f019000-6f2c0000 rw-p 00000000 fd:00 1146914                            /dat
a/dalvik-cache/arm64/system@framework@boot.art
...
7327670000-7329747000 r--p 00000000 fd:00 1884627                        /dat
a/app/com.google.android.gms-4FJbDh-oZv-5bCw39jkIMQ==/oat/arm64/base.odex
..
733494d000-7334cfb000 r-xp 00000000 fd:00 1884542                        /dat
a/app/com.google.android.youtube-Rl_hl9LptFQf3Vf-JJReGw==/lib/arm64/libcrone
t.80.0.3970.3.so
...
```

5.9.3.2.4. 检查沙箱

应用程序数据存储在位于/data/data/<app_package_name>的沙盒目录中。该目录的内容已在“访问应用程序数据目录”一节中详细讨论

5.9.3.3. 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，尤其是在逆向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，以及在原生层进行 Linux/unix 风格的基于 ptrace 调试，这两种调试对逆向工程师都很有价值。

5.9.3.3.1. 调试发布的应用

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）实现的额外功能的劫持。

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 来解决适用于 Android 的 UnCrackable 应用级别 1。注意，这并不是解决这个 crackme 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。

5.9.3.3.2. 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 `suspend` 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。输入?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 `redefine` 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，方法断点仍然有效。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。

让我们回顾一下适用于 Android 的 UnCrackable 应用级别 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 root 和篡改检测，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$Builder((Context)this).cre
ate();
        create.setTitle((CharSequence)title);
        create.setMessage((CharSequence)"This in unacceptable. The 应用 is now
going to exit.");
        create.setButton(-3, (CharSequence)"OK", (DialogInterface$OnClickList
ener)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 flag 设置为 true，直到最终显示警报框（断点将触发五或六次）。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。

现在已经绕过防篡改了，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"verify"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
```

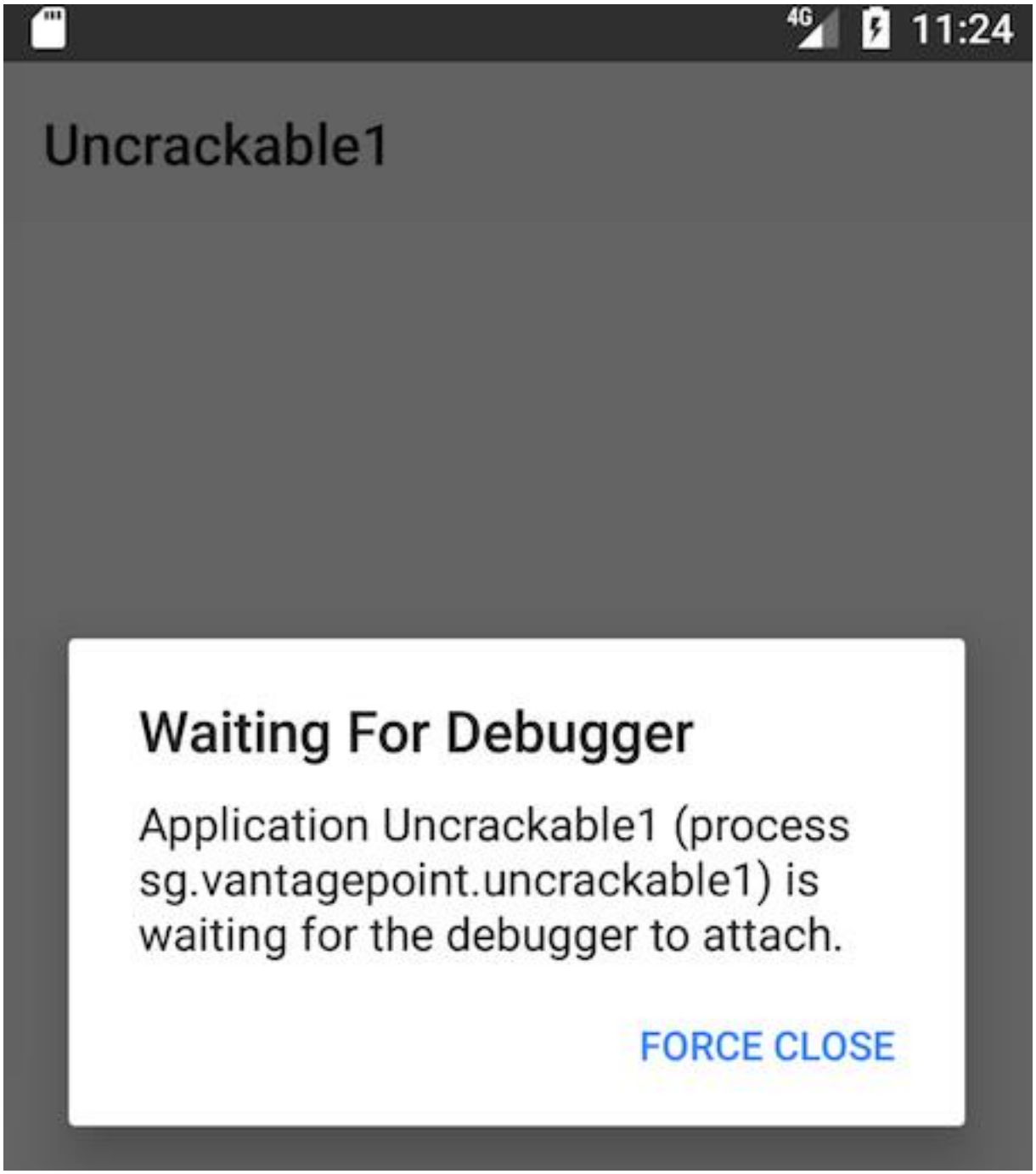
这就是您要找的明文字符串。

5.9.3.3.3. 使用 IDE 调试

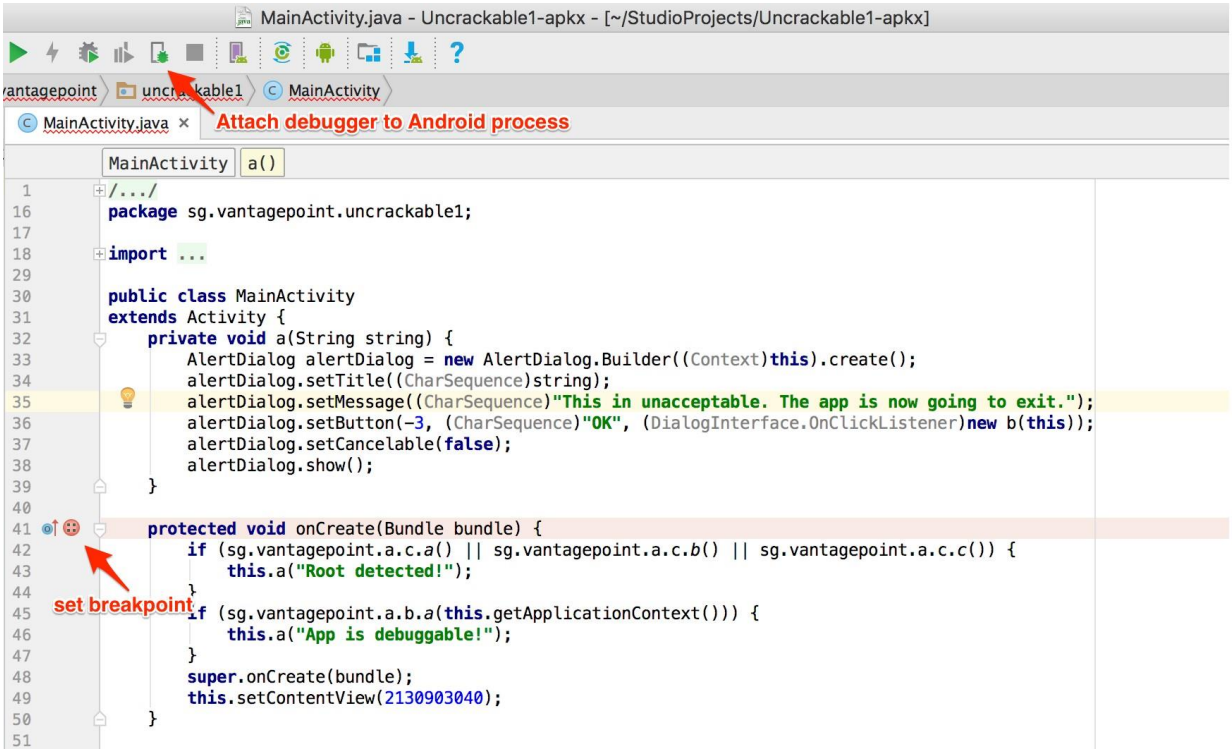
在 IDE 中用反编译的源码设置项目是一个巧妙的技巧，它允许您直接在源代码中设置方法断点。在大多数情况下，您应该可以单步调试应用程序，用 GUI 检查变量的状态。这种体验不会是完美的--毕竟这不是原始的源代码，所以您将无法设置行断点，有时甚至无法正常工作。不过话说回来，逆向代码从来都不是一件容易的事，高效地浏览和调试普通的 Java 代码是一种非常方便的方式。NetSPI 博客中已经介绍过类似的方法。

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（**debug app**）"，并确保您已经开启了"等待调试器（Wait For Debugger）"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。



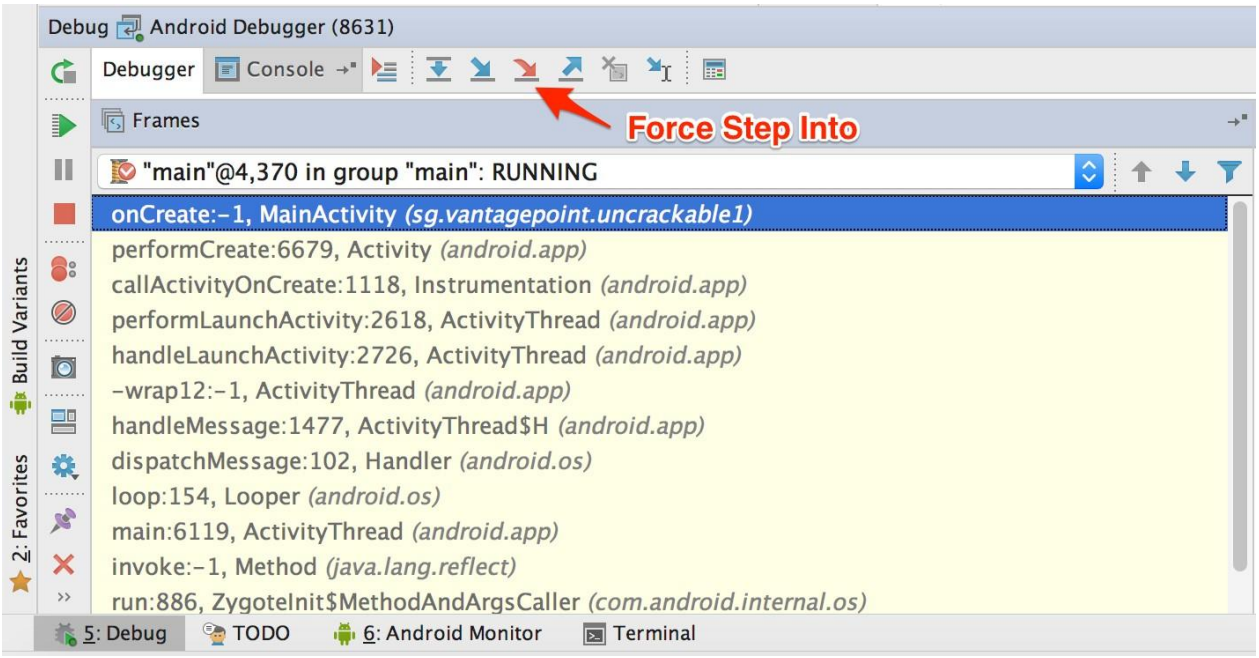
现在，您可以通过 "附加调试器（Attach Debugger） "工具栏按钮来设置断点并附加到应用进程。



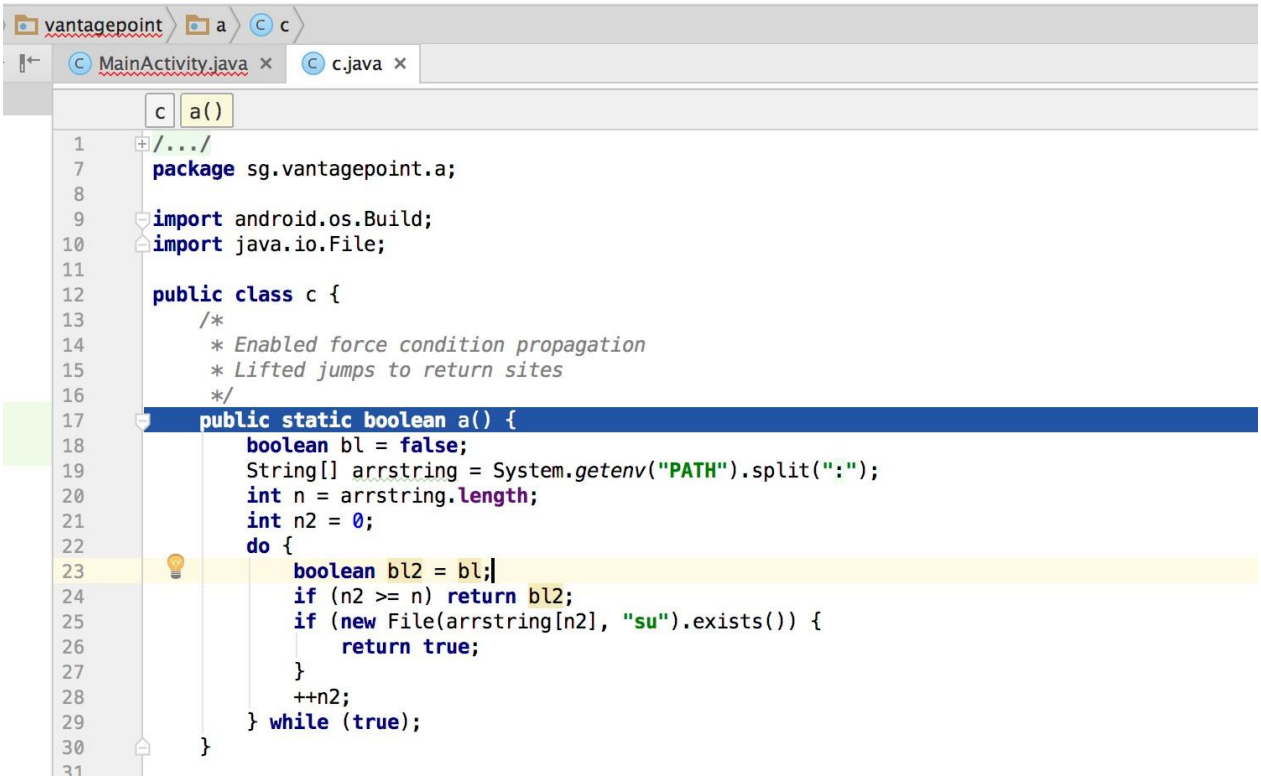
请注意，从反编译的源码调试应用程序时，只有方法断点才有效。一旦达到方法断点，您将有机会在方法执行过程中进行单步执行。

从列表中选择应用程序后，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。应用程序会在 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。

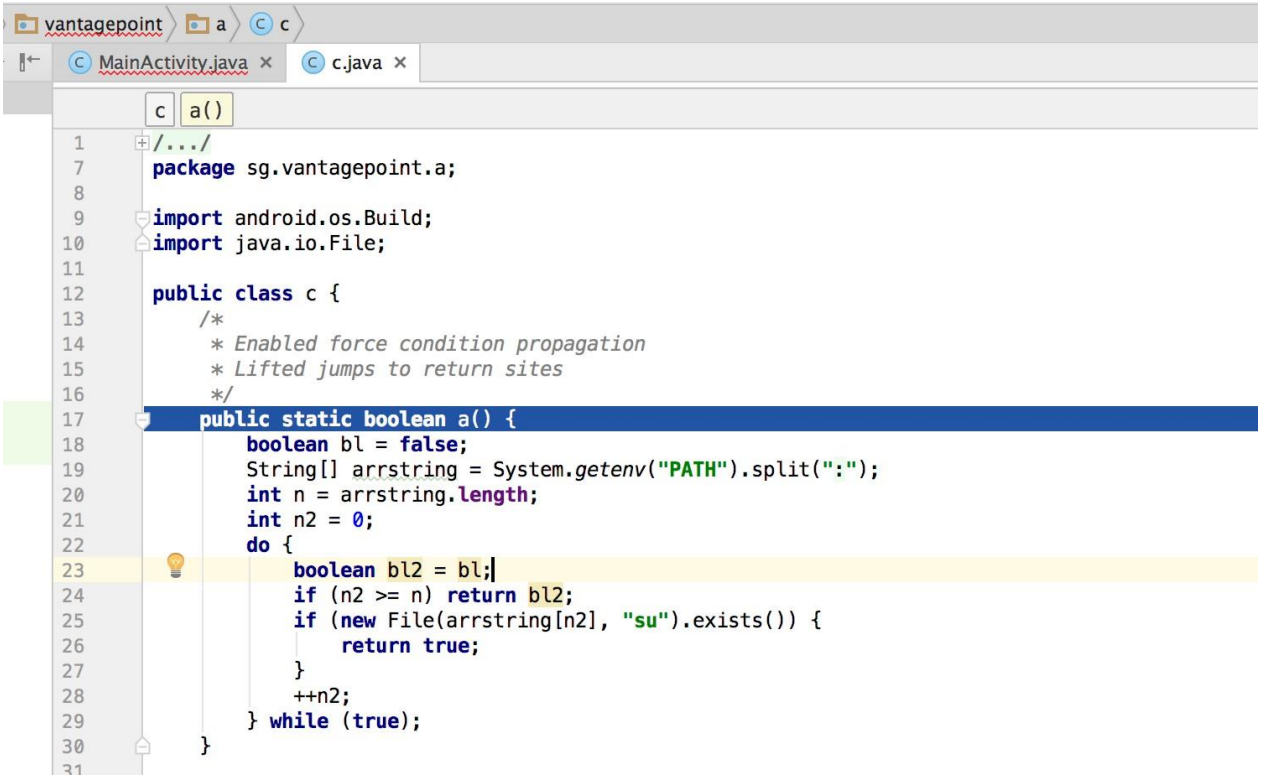
接下来，在 Debugger 视图中点击"Force Step Into"来单步完成 onCreate 方法。通过" Force Step Into "选项，可以对通常被调试器忽略的 Android 框架函数和核心 Java 类进行调试。



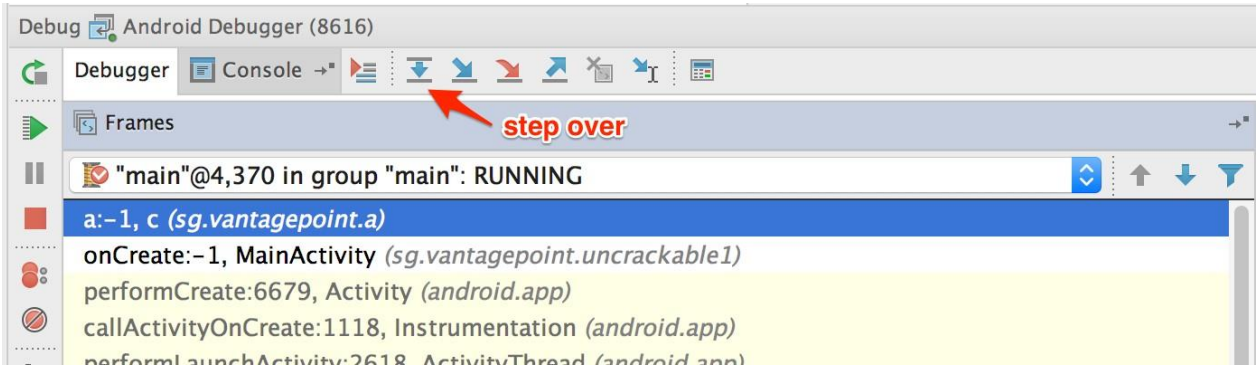
一旦您"Force Step Into"，调试器就会停在下一个方法的开头，这个方法就是 sg.vantagepoint.a.c 类的 a 方法。



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。



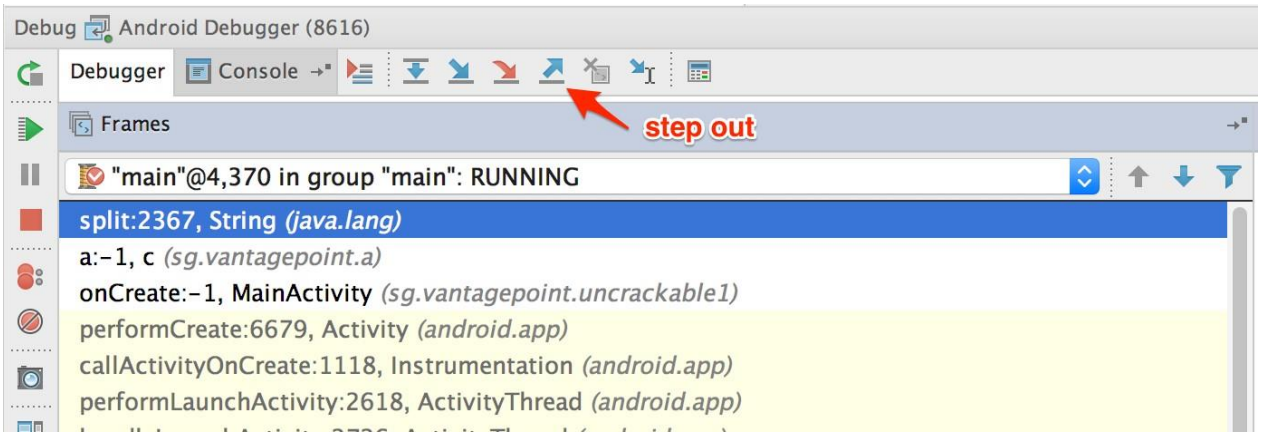
在 "变量 Variables "窗口里面可以看到目录名，点击“步入 Step Over"调试器视图，进入并单步执行 a 方法。



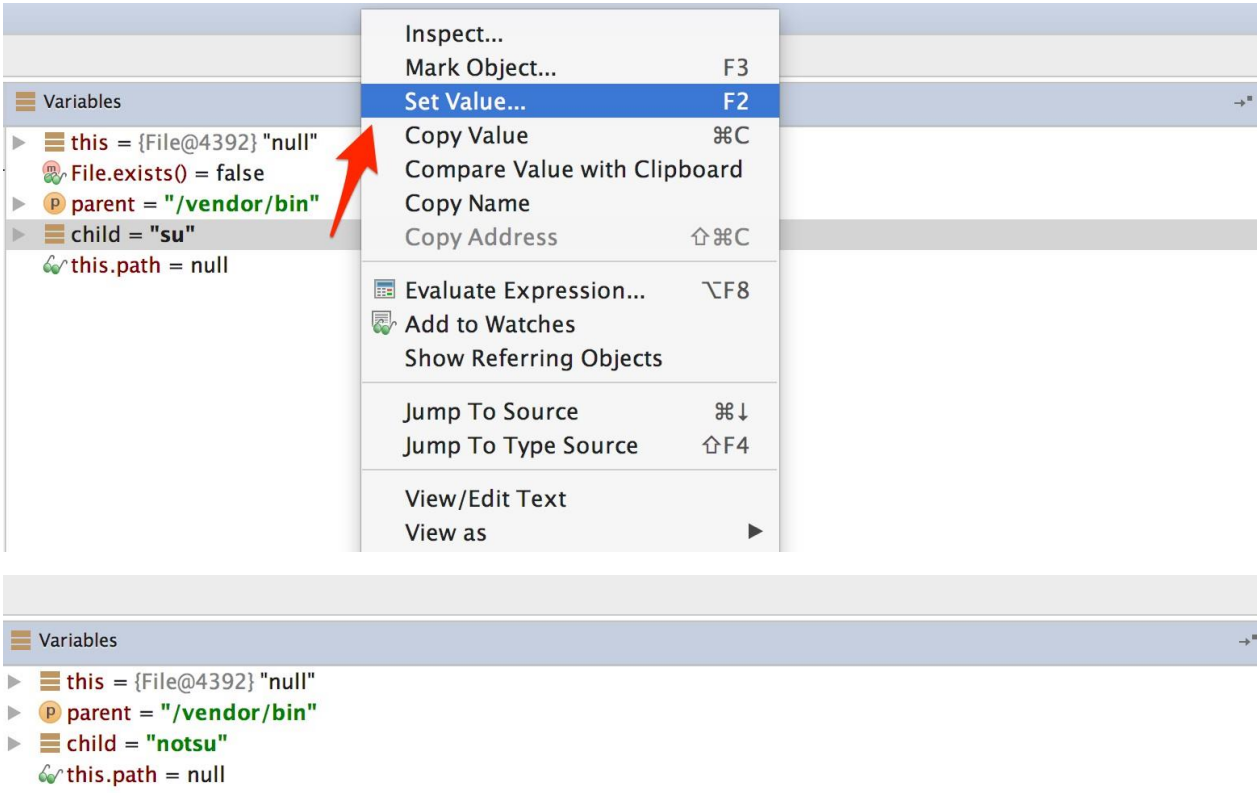
用" Force Step Into"功能步入 System.getenv 方法。

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" Step Out "来跳出函数。一旦您到达反编译源代码和" Step Out "核心 Java 和 Android 类，使用" Force Step Into "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。

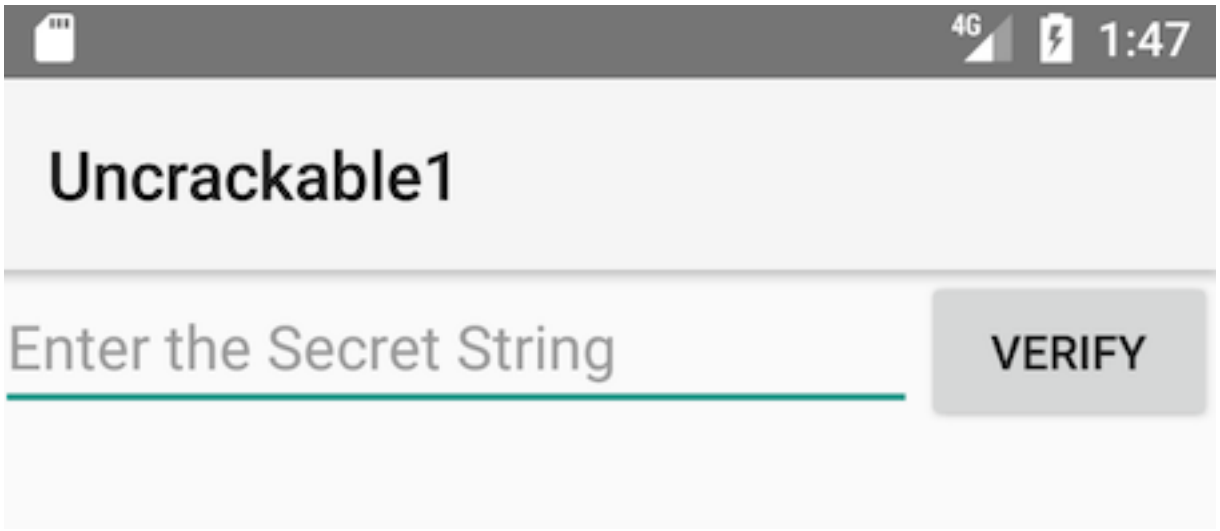


a 方法得到目录名后，会在这些目录中搜索 su 二进制。若要取消此检查，请单步执行检测方法并检查可变内容。一旦执行到达将检测到 su 二进制文件的位置，按 F2 键或右键单击并选择“设置值 Set Value”，修改保存文件名或目录名的变量。



一旦您修改了二进制名称或目录名称，`File.exists` 应该返回 `false`。

这样就打败了应用程序的第一个 root 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安全字符串验证功能。

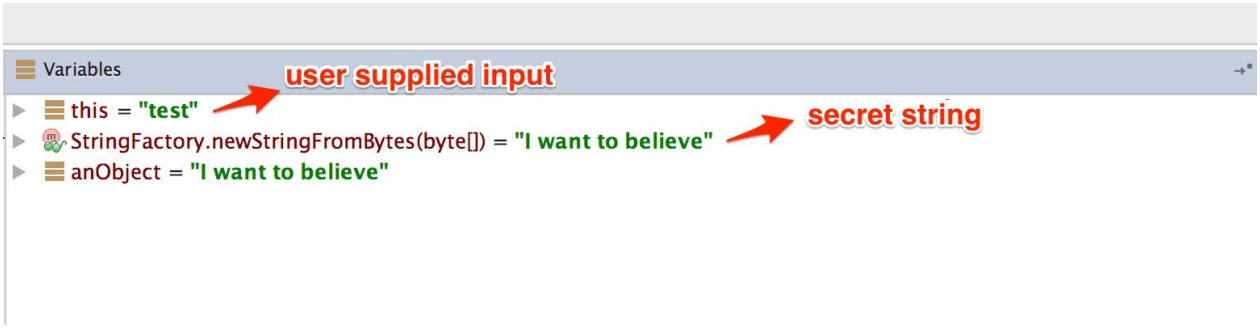


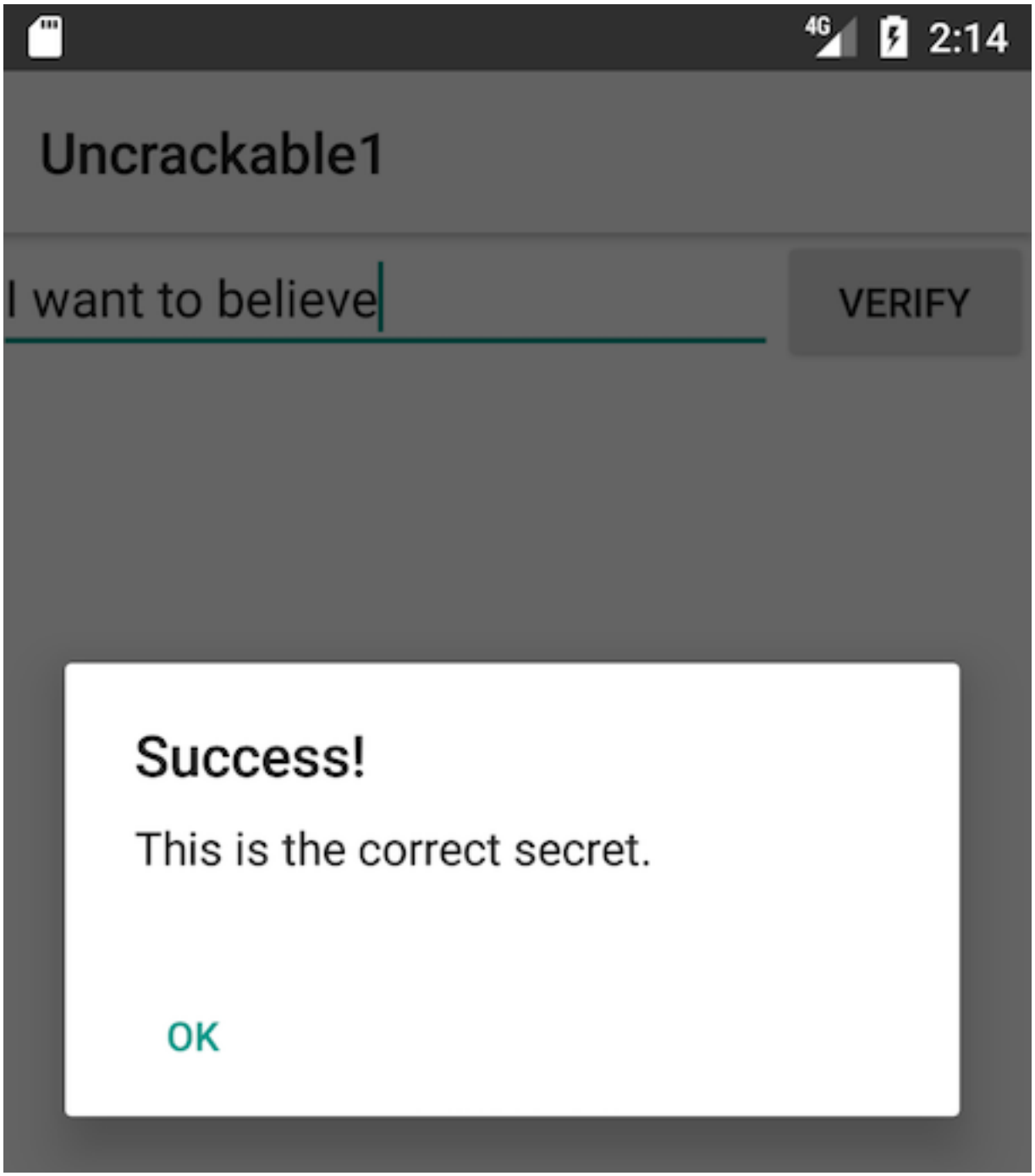

```
/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (a.a((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
    } else {
        alertDialog.setTitle((CharSequence)"Nope...");
        alertDialog.setMessage((CharSequence)"That's not it. Try again.");
    }
    alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}
```

安全码的验证由 sg.vantagepoint.uncrackable1.a 类的 a 方法验证。在方法 a 上设置断点，并在到达断点时“Force Step Into”。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。


```
public static boolean a(String string) {
    byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2GOc=", (int)0);
    byte[] arrby2 = new byte[]{};
    try {
        arrby = sg.vantagepoint.a.a.a(a.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        arrby2 = arrby;
    }
    catch (Exception exception) {
        Log.d((String)"CodeCheck", (String)("AES error:" + exception.getMessage()));
    }
    if (string.equals(new String(arrby2))) {
        return true;
    }
    return false;
}
```

当您到达 String.equals 方法调用时，您可以在 "变量 Variables "视图中看到秘密字符串。


Variables
user supplied input
this = "test"
StringFactory.newStringFromBytes(byte[]) = "I want to believe"
secret string
anObject = "I want to believe"



5.9.3.2.4. 调试原生代码

Android 上的原生代码被打包到 ELF 共享库中，并像其他原生 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 Android NDK。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.vantagepoint.helloworldjni)。然后切换到 root 用户并附加 gdbserver：

```
$ adb shell
$ ps | grep helloworld
u0_a164   12690 201   1533400 51692 ffffffff 00000000 S sg.vantagepoint.hello
worldjni
$ 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 Wait for Debugger”选项来解决这个问题。转到“开发人员选项 **Developer Options**”->“选择调试应用程序 **Select debug app**”并选择 HelloWorldJNI，然后激活“等待调试器 **Wait for debugger**”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 `Java_sg_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 轻轻地将进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，在 jdb 中暂停 Java 运行时加载 libnative-lib.so 的进程。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` 返回。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.<cl
init>(), 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 ?? ()
```

5.9.3.4. 跟踪

5.9.3.4.1. 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.<c
linit>().
It will be set after the class is loaded.
> resume
All threads resumed.M
Set deferred breakpoint com.acme.bob.mobile.android.core.BobMobileApplicatio
n.<clinit>()

Breakpoint hit: "thread=main", com.acme.bob.mobile.android.core.BobMobileAppl
ication.<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 Android Studio 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，对于分析混淆后的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。Android Studio 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浏览方式。您通常会使用 Android Studio 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 Android Studio 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 **stop watch**”按钮。开始录制。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跟踪的记录。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 Android Studio（**Tools** -> **Android** -> **Android Device Monitor**）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 **Devices**”选项卡中选择目标进程，然后单击“启动方法评测 **Start Method Profiling**”。单击停止（**stop**）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 profile”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。Android Studio 用户指南包含了关于 Traceview 的更多信息。

5.9.3.4.1.1 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。它监视进程和内核之间的交互，这是监视系统调用的一种非常方便的方法。然而，也有一个缺点：由于 strace 依赖于 ptrace 系统调用来连接到目标进程，一旦反调试措施激活，它就会停止工作。

如果 **Settings > Developer options** 中的“Wait for debugger”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" ]]; t
hen strace -s 2000 - e "!read" -ff -p "$pid"; break; fi; done
```

5.9.3.4.1.2 Ftrace

Ftrace 是一个直接内置在 Linux 内核中的跟踪工具。在 root 过的设备上，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:这个文件列出了编译到内核中的可用跟踪器。
- current_tracer:这个文件设置或显示当前的跟踪器。
- tracing_on:在该文件中写入"1"，允许/开始更新环形缓冲区。写入"0"将阻止对环形缓冲区的进一步写入。

5.9.3.4.1.3 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，它们允许劫持函数入口和出口。

原生 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会在 Lollipop 和 Marshmallow 的原生内核中内核错误，因为 sys_call_table 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）。

5.9.3.4.2. 方法跟踪

方法分析告诉您方法被调用的频率，而方法跟踪则帮助您确定其输入和输出值。当处理具有较大代码量和/或混淆的应用程序时，这种技术可能非常有用。

正如我们将在下一节中很快讨论的那样，`frida-trace` 为 Android/iOS 原生代码跟踪和 iOS 高级方法跟踪提供了现成的支持。如果你喜欢基于 GUI 的方法，你可以使用诸如 RMS-Runtime Mobile Security 之类的工具，它可以提供更直观的体验，并包括几个方便的跟踪选项。

5.9.3.4.3. 原生代码跟踪

与 Java 方法跟踪相比，原生方法跟踪可以相对轻松地执行。`frida-trace` 是一种 CLI 工具，用于动态跟踪函数调用。它使跟踪原生函数变得微不足道，对于收集有关应用程序的信息非常有用。

为了使用 `frida-trace`，设备上应该运行 Frida 服务端。下面演示了使用 `frida-trace` 跟踪 libc 的 open 函数的示例，其中-U 是连接到 USB 设备，而-i 指定要包含在跟踪中的函数。

```
frida-trace -U -i "open" com.android.chrome
```

```
Started tracing 1 function. Press Ctrl+C to stop.
           /* TID 0x36ba */
  3385 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/GPUCache/index", oflag=0x0)
  3391 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/GPUCache/index-dir/the-real-index", oflag=0x0)
  3418 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/8c6cfae1548e2abe_0", oflag=0xc2)
           /* TID 0x352d */
  3852 ms  open(path="/proc/net/xt_qtaguid/stats", oflag=0x0)
  3853 ms  open(path="/proc/net/xt_qtaguid/stats", oflag=0x0)
           /* TID 0x36ba */
  3861 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/LOG", oflag=0x241)
  3862 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/LOCK", oflag=0x2)
  3863 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/CURRENT", oflag=0x0)
           /* TID 0x401b */
  3863 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/7a7195018f1765e4_0", oflag=0x2)
           /* TID 0x401a */
  3864 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/32f59c357713aa03_0", oflag=0x2)
  3864 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/32f59c357713aa03_1", oflag=0x2)
           /* TID 0x36ba */
  3865 ms  open(path="/data/user/0/com.android.chrome/app_chrome/Default/DeltaFileLevelDb/MANIFEST-000001", oflag=0x0)
           /* TID 0x401b */
  3865 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/7a7195018f1765e4_1", oflag=0x2)
           /* TID 0x4035 */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/f3595c2530ef9720_0", oflag=0x2)
           /* TID 0x401a */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/32f59c357713aa03_s", oflag=0x2)
           /* TID 0x401b */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/7a7195018f1765e4_s", oflag=0x2)
           /* TID 0x36b8 */
  3866 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/c91d3ba6d5be834e_0", oflag=0x2)
           /* TID 0x4035 */
  3867 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/f3595c2530ef9720_1", oflag=0x2)
           /* TID 0x418e */
  3867 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/0660be5420ecb9ff_0", oflag=0x2)
  3868 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/0660be5420ecb9ff_1", oflag=0x2)
  3868 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/0660be5420ecb9ff_s", oflag=0x2)
  3869 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/6a5a2bb023ded144_0", oflag=0x2)
  3869 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/6a5a2bb023ded144_1", oflag=0x2)
  3869 ms  open(path="/data/user/0/com.android.chrome/cache/Cache/6a5a2bb023ded144_s", oflag=0x2)
  3870 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/582575465db63dec_0", oflag=0x2)
  3870 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/582575465db63dec_1", oflag=0x2)
  3871 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/582575465db63dec_s", oflag=0x2)
           /* TID 0x401b */
  3871 ms  open(path="/data/user/0/com.android.chrome/cache/Code Cache/js/9f474cdcfa861f4a_0", oflag=0x2)
```

请注意，默认情况下，只显示传递给函数的参数，而不显示返回值。在后台，`frida-trace` 在自动生成的`__handlers__`文件夹中为每个匹配的函数生成一个小 JavaScript 处理程序文件，然后 Frida 将其注入到进程中。您可以编辑这些文件以获得更高级的使用，例如获取函数的返回值、输入参数、访问内存等。有关更多详细信息，请查看 Frida 的 JavaScript API。

在这种情况下，生成的脚本跟踪 libc.so 中对 open 函数的所有调用。脚本位于`__handlers__/libc.so/open.js` 中。如下所示：

```
{
  onEnter: function (log, args, state) {
    log('open(' +
      'path="' + args[0].readUtf8String() + '"' +
      ', oflag=' + args[1] +
    ')');
  },
```

```
  onLeave: function (log, retval, state) {
      log('\t return: ' + retval);      \\ edited
  }
}
```

在上面的脚本中，onEnter 负责以正确的格式记录对该函数的调用及其两个输入参数。您可以编辑 onLeave 事件以打印返回值，如上图所示。

> 请注意，libc 是一个众所周知的库，Frida 能够导出其 open 函数的输入参数，并自动正确记录这些参数。但对于其他库或 Android Kotlin/Java 代码来说，情况并非如此。在这种情况下，您可能希望通过参考 Android 开发者文档或首先对应用程序进行逆向工程来获得感兴趣的功能的签名。

在上面的输出中需要注意的另一件事是它是彩色的。一个应用程序可以运行多个线程，每个线程可以独立调用open函数。通过使用这种配色方案，可以轻松地将每个线程的输出进行视觉隔离。

frida-trace 是一种功能非常广泛的工具，有多种配置选项可用，例如：

- 包括（-I）和不包括（-X）整个模块。
- 使用-i “Java_*” 跟踪 Android 应用程序中的所有 JNI 函数（注意使用*匹配以 “Java_” 开头的所有可能函数）。
- 当没有可用的函数名符号（剥离二进制文件）时，按地址跟踪函数，例如 “libjpeg.so!0x4793c” 。

```
frida-trace -U -i "Java_*" com.android.chrome
```

许多二进制文件被剥离，没有可用的函数名符号。在这种情况下，也可以使用其地址跟踪函数。

```
frida-trace -p 1372 -a "libjpeg.so!0x4793c"
```

Frida 12.10 引入了一种新的有用语法来查询 Java 类和方法，以及通过-j（从 Frida tools 8.0 开始）对 Frida 跟踪的 Java 方法提供跟踪支持。

- 在 Frida 脚本中：例如 Java.enumerateMethods('*youtube*!on*')使用表达式获取所有类，这些类的名称中包含 “youtube” ，并且枚举所有以 “on” 开头的方法。

- 在 frida-trace 中：例如`-j '*!*certificate*/isu'`触发不区分大小写的查询（i），包括方法签名(s)，不包括系统类（u）。

有关更多详细信息，请参阅发行说明。要了解更多有关高级使用的所有选项，请查看 Frida 官方网站上的文档。

5.9.3.4.4. JNI 跟踪

如审查反汇编原生代码一节所述，传递给每个 JNI 函数的第一个参数是 JNI 接口指针。该指针包含一个函数表，允许原生代码访问 Android 运行时。识别对这些函数的调用有助于理解库功能，例如创建什么字符串或调用什么 Java 方法。

jnitrace 是一个基于 Frida 的工具，类似于 frida-trace，它专门针对原生库使用 Android 的 JNI API，提供了一种方便的方式来获取 JNI 方法跟踪，包括参数和返回值。

您可以通过运行 `pip install jnitrace` 轻松安装它，并按如下方式直接运行它：

```
jnitrace -l libnative-lib.so sg.vantagepoint.helloworldjni
```

> 可以多次使用 l 选项以跟踪多个库，也可以使用*以跟踪所有库。然而，这可能会造成很多输出。

```
Tracing. Press any key to quit...
Traced library "libnative-lib.so" loaded from path "/data/app/sg.vantagepoint.helloworldjni-1/lib/x86_64".

          /* TID 2573 */
    259 ms [+] JNIEnv->NewStringUTF
    259 ms |- JNIEnv*          : 0x7ff668992240
    259 ms |- char*            : 0x7ff663101503
    259 ms |:     Hello from C++
    259 ms |= jstring          : 0x20001d
```

在输出中，您可以看到从原生代码调用 `NewStringUTF` 的跟踪（其返回值返回给 Java 代码，有关更多详细信息，请参阅“审查反汇编原生代码”一节）。请注意，与 frida-trace 类似，输出是彩色的，有助于从视觉上区分不同的线程。

在跟踪 JNI API 调用时，您可以在顶部看到线程 ID，然后是 JNI 方法调用，包括方法名、输入参数和返回值。在从原生代码调用 Java 方法的情况下，还将提供 Java 方法参数。最后，jnitrace 将尝试使用 Frida 回溯库来显示 JNI 调用的来源。

要了解有关高级使用的所有选项的更多信息，请查看 jnitrace GitHub 页面上的文档。

5.9.3.5. 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 guest 指令实时转换为主机处理器可以理解的指令来模拟 guest CPU。guest 指令的每个基本块都被反汇编并转换为称为微型代码生成器（Tiny Code Generator,TCG）的中间表示。TCG 块被编译成主机指令块，存储在代码缓存中并执行。在执行基本块后，QEMU 对下一个 guest 指令块重复该过程（或从缓存加载已翻译的块）。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、调试和跟踪组件。QEMU 特定的参数可以通过`-qemu` 命令行参数传递给模拟器。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 命令行参数启动 QEMU 将导致它转储正在执行的 guest 代码、微操作或主机指令块。使用-d_asm 参数，QEMU 会在 guest 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 guest 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，比如重建原生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。

5.9.3.6. 二进制分析

二进制分析框架为您提供了强大的方法来自动化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，该技术允许确定达到特定目标所需的条件。它将程序的语义转换为逻辑公式，其中一些变量由具有特定约束的符号表示。通过求解约束，可以找到执行程序某些分支所需的条件。

5.9.3.6.1. 符号执行

符号执行是您工具箱中一种非常有用的技术，尤其是在处理需要找到正确输入以访问特定代码块的问题时。在本节中，我们将使用 Angr 二进制分析框架作为符号执行引擎来解决一个简单的 Android crackme。

目标 crackme 是一个简单的 Android 许可证密钥验证可执行文件。我们很快就会看到，crackme 中的关键验证逻辑是在原生代码中实现的。通常认为，分析编译的原生代码比分析等效的编译 Java 代码更困难，因此，关键业务逻辑通常是用原生编写的。当前的示例应用程序可能并不代表真实世界的问题，但它有助于获得有关符号执行的一些基本概念，您可以在真实情况中使用这些概念。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 由单个 ELF 可执行文件组成，可在任何 Android 设备上执行，方法如下：

```
$ 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。首先，在诸如 Cutter 之类的反汇编程序中打开 ELF 可执行文件。。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 作为镜像的基本地址）。

Disassembly

```
            0x0000186c      andeq r0, r0, r0, ror r2
            0x00001870      andeq r0, r0, r8, lsl 5
            ;-- pc:
            ;-- r15:
/ (fcn) fcn.00001874 196
|   fcn.00001874 (int32_t arg1, int32_t arg2);
|           ; var int32_t var_24h @ fp-0x24
|           ; var int32_t var_20h @ fp-0x20
|           ; var int32_t var_18h @ fp-0x18
|           ; var int32_t var_0h @ sp+0x0
|           ; arg int32_t arg1 @ r0
|           ; arg int32_t arg2 @ r1
|           0x00001874      push {fp, lr}
|           0x00001878      add fp, sp, 4
|           0x0000187c      sub sp, sp, 0x28 ; '('
|           0x00001880      str r0, [var_20h] ; 0x20 ; "$!" ; arg1
|           0x00001884      str r1, [var_24h] ; 0x24 ; '$' ; arg2
|           0x00001888      ldr r3, [var_20h] ; 0x20 ; "$!"
|           0x0000188c      cmp r3, 2
|       ,=< 0x00001890      beq 0x1898 ; unlikely
|       |   0x00001894      bl fcn.000016a8
|       `-> 0x00001898      ldr r3, [var_24h] ; 0x24 ; '$'
|           0x0000189c      add r3, r3, 4
|           0x000018a0      ldr r3, [r3]
|           0x000018a4      mov r0, r3
|           0x000018a8      bl sym.imp.strlen ; size_t strlen(const char *s)
|                               ; size_t strlen("\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
|           0x000018ac      mov r3, r0
|           0x000018b0      cmp r3, 0x10
|       ,=< 0x000018b4      beq 0x18bc ; unlikely
|       |   0x000018b8      bl fcn.000016cc
|       `-> 0x000018bc      ldr r3, [0x00001938] ; "chr"
|           0x000018c0      add r3, pc, r3 ; "Entering base32_decode" str.Entering_base32_decode
|           0x000018c4      mov r0, r3 ; "Entering base32_decode" str.Entering_base32_decode
|           0x000018c8      bl sym.imp.puts ; int puts(const char *s)
|                               ; int puts("Entering base32_decode")
|           0x000018cc      ldr r3, [var_24h] ; 0x24 ; '$'
|           0x000018d0      add r3, r3, 4
|           0x000018d4      ldr r2, [r3]
|           0x000018d8      sub r3, fp, 0x14
|           0x000018dc      sub r1, fp, 0x18
|           0x000018e0      str r1, [sp]
|           0x000018e4      mov r0, 0
|           0x000018e8      mov r1, r2
|           0x000018ec      mov r2, 0x10
|           0x000018f0      bl fcn.00001340; fcn.00001340(0x0)
|           0x000018f4      ldr r3, [var_18h] ; 0x18
|           0x000018f8      ldr r2, [0x0000193c] ; "t"
|           0x000018fc      add r2, pc, r2 ; "Outlen = %d\n" str.Outlen____d
|           0x00001900      mov r0, r2 ; "Outlen = %d\n" str.Outlen____d
|           0x00001904      mov r1, r3
|           0x00001908      bl sym.imp.printf ; int printf(const char *format)
|                               ; int printf("Outlen = %d\n")
|           0x0000190c      ldr r3, [0x00001940] ; "texit"
|           0x00001910      add r3, pc, r3 ; "Entering check_license" str.Entering_check_license
|           0x00001914      mov r0, r3 ; "Entering check_license" str.Entering_check_license
|           0x00001918      bl sym.imp.puts ; int puts(const char *s)
|                               ; int puts("Entering check_license")
|           0x0000191c      sub r3, fp, 0x14
|           0x00001920      mov r0, r3
|           0x00001924      bl fcn.00001760
|           0x00001928      mov r3, 0
|           0x0000192c      mov r0, r3
```

函数名已从二进制文件中删除，但幸运的是，有足够的调试字符串为我们提供了代码内容。接下来，我们将从偏移量 0x00001874 处的输入函数开始分析二进制文件，并记下我们容易获得的所有信息。在分析过程中，我们还将尝试识别适合符号执行的代码区域。

`strlen` 在偏移量 `0x000018a8` 处调用，返回值与偏移量 `0x000018b0` 处的 0x10 进行比较。紧接着，输入字符串被传递到偏移量 `0x00001340` 处的 Base32 解码函数。这为我们提供了有价值的信息，即输入许可证密钥是一个 Base32 编码的 16 个字符的字符串（原始数据中总共 10 个字节）。解码后的输入随后传递给偏移量为 0x00001760 的函数，该函数验证许可证密钥。此功能的拆解如下所示。

现在，我们可以使用有关预期输入的信息来进一步查看 0x00001760 处的验证函数。

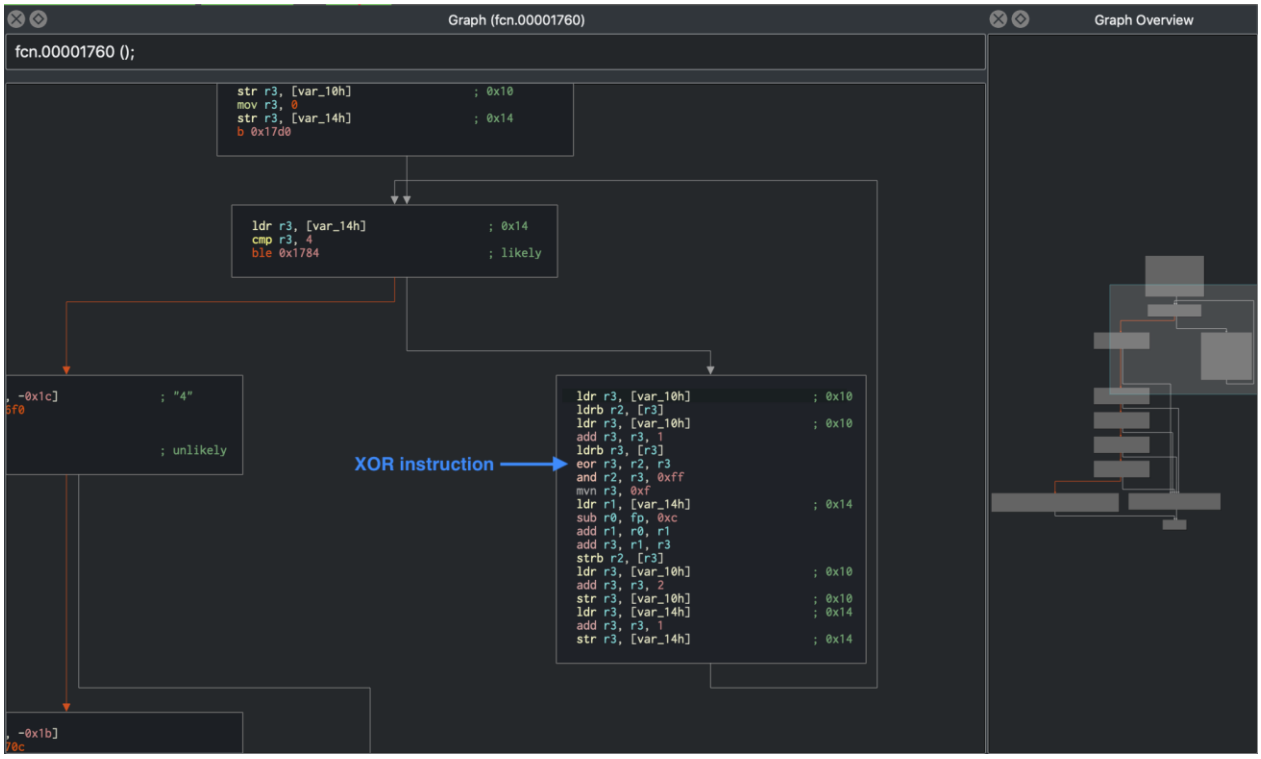
```
(fcn) fcn.00001760 268
  fcn.00001760 (int32_t arg1);
          ; var int32_t var_20h @ fp-0x20
          ; var int32_t var_14h @ fp-0x14
          ; var int32_t var_10h @ fp-0x10
          ; arg int32_t arg1 @ r0
          ; CALL XREF from fcn.00001760 (+0x1c4)
```

```
|           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                                 ; like
ly
```

```
|           0x000017dc      ldrb r4, [fp, -0x1c]                    ; "4"
|           0x000017e0      bl fcn.000016f0
|           0x000017e4      mov r3, r0
|           0x000017e8      cmp r4, r3
|       ┌─< 0x000017ec      bne 0x1854                              ; like
ly
|       │   0x000017f0      ldrb r4, [fp, -0x1b]
|       │   0x000017f4      bl fcn.0000170c
|       │   0x000017f8      mov r3, r0
|       │   0x000017fc      cmp r4, r3
|      ┌──< 0x00001800      bne 0x1854                              ; like
ly
|      ││   0x00001804      ldrb r4, [fp, -0x1a]
|      ││   0x00001808      bl fcn.000016f0
|      ││   0x0000180c      mov r3, r0
|      ││   0x00001810      cmp r4, r3
|     ┌───< 0x00001814      bne 0x1854                              ; like
ly
|     │││   0x00001818      ldrb r4, [fp, -0x19]
|     │││   0x0000181c      bl fcn.00001728
|     │││   0x00001820      mov r3, r0
|     │││   0x00001824      cmp r4, r3
|    ┌────< 0x00001828      bne 0x1854                              ; like
ly
|    ||||   0x0000182c      ldrb r4, [fp, -0x18]
|    ||||   0x00001830      bl fcn.00001744
|    ||||   0x00001834      mov r3, r0
|    ||||   0x00001838      cmp r4, r3
|   ┌─────< 0x0000183c      bne 0x1854                              ; like
ly
|   |||||   0x00001840      ldr r3, [0x0000186c]                    ; [0x1
86c:4]=0x270 section..hash ; section..hash
|   |||||   0x00001844      add r3, pc, r3                          ; 0x1a
bc ; "Product activation passed. Congratulations!"
|   |||||   0x00001848      mov r0, r3                              ; 0x1a
bc ; "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, 0
x183c)
|   |└└└└─> 0x00001854      ldr r3, aav.0x00000288                  ; [0x1
870:4]=0x288 aav.0x00000288
|   |       0x00001858      add r3, pc, r3                          ; 0x1a
e8 ; "Incorrect serial." ;
|   |       0x0000185c      mov r0, r3                              ; 0x1a
e8 ; "Incorrect serial." ;
```

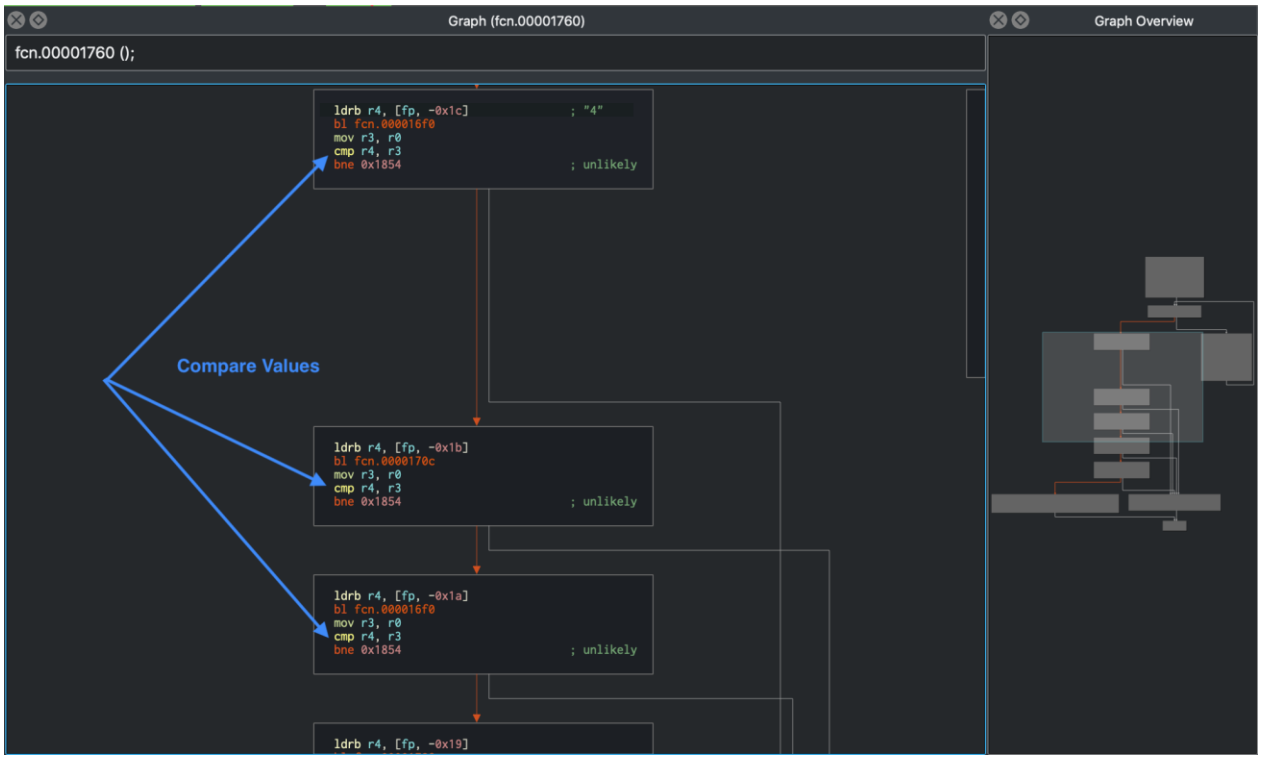
```
|  |        0x00001860      bl sym.imp.puts                                ; int
puts(const char *s)
|  |                                                                       ; int
puts("Incorrect serial.")
|  |        ; CODE XREF from fcn.00001760 (0x1850)
|  └──────> 0x00001864      sub sp, fp, 8
└           0x00001868      pop {r4, fp, pc}                               ; entr
y.preinit0 ; entry.preinit0 ;
```

讨论函数中的所有指令超出了本章的范围，相反，我们将只讨论分析所需的要点。在验证函数中，0x00001784 处存在一个循环，该循环在偏移量 0x00001798 处执行异或操作。循环在下面的图形视图中更清晰可见。



异或（XOR）是一种非常常用的加密信息的技术，其中混淆是目标，而不是安全性。**XOR 不应用于任何重要的加密**，因为它可以使用频率分析进行破解。因此，在这种验证逻辑中仅存在异或加密总是需要特别注意和分析。

继续观察，在偏移量 `0x000017dc` 处，将从上面获得的异或解码值与 `0x000017e8` 处子函数调用的返回值进行比较。



显然，该功能并不复杂，可以手动分析，但仍然是一项繁琐的任务。特别是在处理大型代码库时，时间可能是一个主要约束，因此需要自动进行此类分析。动态符号执行在这些情况下非常有用。在上述 crackme 中，符号执行引擎可以通过映射许可证检查的第一条指令（0x00001760 处）和打印“Product activation passed”消息的代码（0x00001840 处）之间的路径来确定输入字符串每个字节上的约束。

从上述步骤获得的约束被传递到求解器引擎，该引擎找到满足这些约束的输入-有效的许可证密钥。

您需要执行几个步骤来初始化 Angr 的符号执行引擎：

- 将二进制文件加载到项目中，这是 Angr 中任何类型分析的起点。
- 传递分析开始的地址。在这种情况下，我们将使用序列号验证函数的第一条指令初始化状态。这使得问题更容易解决，因为您避免了符号执行 Base32 实现。
- 传递分析应到达的代码块地址。在本例中，这是偏移量 `0x00001840`，负责打印“Product activation passed”消息的代码位于该位置。
- 此外，指定分析不应到达的地址。在这种情况下，在 `0x00001854` 处打印“Incorrect serial”消息的代码块并不有趣。

  注意，Angr 加载器将加载基址为 `0x400000` 的 PIE 可执行文件，在将其传递给 Angr 之前，需要添加来自 Cutter 的偏移量。

最终解决方案脚本如下所示：

```
import angr # Version: 9.2.2
import base64

load_options = {}

b = angr.Project("./validate", load_options = load_options)

# 密钥验证函数从 0x401760 开始，所以我们在那里创建初始状态。
# 这使事情加快了很多，因为我们绕过了 Base32 编码器。

options = {
    angr.options.SYMBOL_FILL_UNCONSTRAINED_MEMORY,
    angr.options.ZERO_FILL_UNCONSTRAINED_REGISTERS,
}
state = b.factory.blank_state(addr=0x401760, add_options=options)

simgr = b.factory.simulation_manager(state)
simgr.explore(find=0x401840, avoid=0x401854)

# 0x401840 = Product activation passed
# 0x401854 = Incorrect serial
found = simgr.found[0]

# 从*(R11 - 0x20)中获取解决字符串。

addr = found.memory.load(found.regs.r11 - 0x20, 1,endness='Iend_LE')
```

```
concrete_addr = found.solver.eval(addr)
solution = found.solver.eval(found.memory.load(concrete_addr,10),
cast_to=bytes)
print(base64.b32encode(solution))
```

正如前面在“动态二进制插桩”一节中所讨论的那样，符号执行引擎为给定的程序输入操作构建了一个的二叉树，并为可能采用的每个可能路径生成一个数学方程。在内部，Angr 探索我们指定的两点之间的所有路径，并将相应的数学方程传递给求解器，以返回有意义的具体结果。我们可以通过 `simulation_manager.found` 访问这些解决方案。其中包含 Angr 探索的所有可能路径，满足我们指定的搜索条件。

仔细看一下脚本的后半部分，在那里将检索最终的解决方案字符串。字符串的地址从地址 `r11 - 0x20` 获得。起初，这可能看起来很神奇，但仔细分析 `0x00001760` 处的函数可以找到线索，因为它可以确定给定的输入字符串是否为有效的许可证密钥。在上面的反汇编中，您可以看到函数的输入字符串（在寄存器 R0 中）是如何存储到局部堆栈变量 `0x0000176c str r0, [var_20h]`中的。因此，我们决定使用该值来检索脚本中的最终解决方案。使用 `found.solver.eval` 你可以问解算器这样的问题：“给定这个操作序列的输出（found 中的当前状态），输入（addr）必须是什么？”。

> 在 ARMv7 中，R11 被称为 fp（函数指针），因此 `R11 - 0x20` 相当于 `fp-0x20`: `var int32_t var_20h @ fp-0x20`

接下来，脚本中的 `endness` 参数指定数据以“little-endian”的方式存储，这几乎适用于所有 Android 设备。

此外，脚本可能看起来只是从脚本内存中读取解决方案字符串。然而，它是从符号记忆中读取的。字符串和指向字符串的指针实际上都不存在。解算器确保其提供的解与程序执行到该点时的解相同。

运行此脚本应返回以下输出：

```
$ python3 solve.py
WARNING | ... | cle.loader | The main binary is a position-independent execut
able. It is being loaded with a base address of 0x400000.
It is being loaded with a base address of 0x400000.

b'JACE6ACIARNAAIIA'
```

现在你可以在你的 Android 设备中运行验证二进制文件，以验证解决方案。

> 您可以使用该脚本获得不同的解决方案，因为可能存在多个有效的许可证密钥。

总之，学习符号执行一开始可能看起来有点吓人，因为它需要深入的理解和广泛的实践。然而，与手动分析复杂的反汇编指令相比，考虑到它可以节省宝贵的时间，这项工作是很合理的。通常，您会使用混合技术，如上例所示，我们对反汇编代码进行手动分析，以向符号执行引擎提供正确的标准。有关 Angr 使用的更多示例，请参阅 iOS 章节。

5.9.4. 篡改和运行时插桩

首先，我们将研究一些修改和插桩移动应用程序的简单方法。篡改意味着对应用程序进行修补或对运行时更改以影响其行为。例如，您可能希望停用阻碍测试过程的 SSL 固定或二进制保护。运行时插桩包括添加劫持和运行时补丁来观察应用程序的行为。然而，在移动应用程序安全中，该术语泛指各种运行时操作，包括重写方法来改变行为。

5.9.4.1. 修补、重新打包和重新签名

对 Android Manifest 或字节码进行小改动通常是解决阻碍您测试或逆向工程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 固定。
- Android Manifest 中的 `android:debuggable` 属性没有设置为 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，您也要为额外检查打上补丁。

第一步是使用 `apktool` 解包和反编译 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，可以使用参数 `--no-src`。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：禁用证书固定

证书固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 固定可以帮助实现这一点。为了演示绕过证书固定，我们将在示例应用程序中演示具体实现。

一旦解包并反汇编了 APK，就可以在 Smali 源代码中找到证书固定检查了。在代码中搜索关键字，如“X509TrustManager”，应该可以为您指明正确的方向。

在我们的示例中，搜索“X509TrustManager”返回一个实现自定义 TrustManager 的类。派生类实现了 `checkClientTrusted`, `checkServerTrusted`, 以及 `getAcceptedIssuers` 方法。

要绕过固定检查，请将 `return-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      # <-- 插入的 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 文件重新签名。

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。只有在应用程序里 manifest 文件里`<application>`元素设置了 `android:debuggable="true"`属性，这个线程才会启动。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，那是调试构建的目的。如果系统属性 ro.debuggable 被设置为"0"，Android 就不允许发布版本 JDWP 和原生调试。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构建，需要在 Android Manifest 文件(AndroidManifest.xml)中修改一个属性。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">
```

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 文件。

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环境变量中。它位于[SDK-Path]/build-tools/[version]。 zipalign 和 apksigner 工具就在这个目录中。

5.9.4.1.4. 重新签名

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/.android/debug.keystore 中创建了调试 KeyStore 和证书。这个 keystore 的默认密码是 “android”，密钥叫做“androiddebugkey”。

标准的 Java 发行版包括用于管理 KeyStore 和证书的密钥工具 keytool。您可以创建自己的签名证书和密钥，然后将其添加到调试 KeyStores 中：

```
keytool -genkey -v -keystore ~/.android/debug.keystore -alias signkey -keyalg
 RSA -keysize 2048 -validity 20000
```

证书可用后，您可以用它重新签名 APK。确保 apksigner 在环境变量中，并且从您重新打包的 APK 所在的文件夹中运行它。

```
apksigner sign --ks  ~/.android/debug.keystore --ks-key-alias signkey UnCrack
able-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-Repacka
ged.apk signkey
zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
```

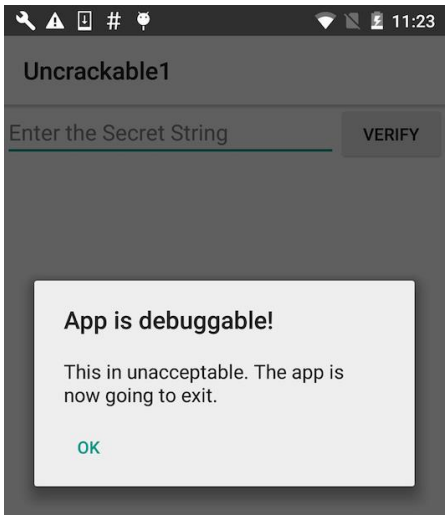
现在您可以重新安装该应用程序：

```
adb install UnCrackable-Repackaged.apk
```

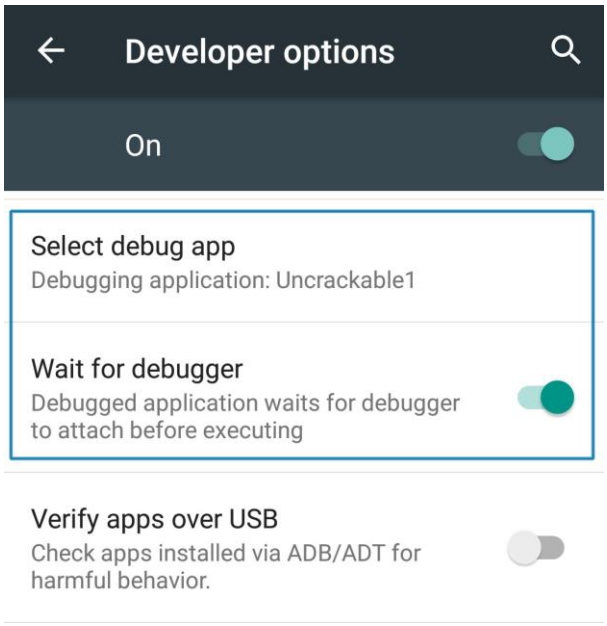
5.9.4.1.5. " Wait For Debugger"功能

适用于 Android 的 UnCrackable 应用级别 1 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。一个对话框会立即显示，一旦您点击 "OK"，程序就会终止。

幸运的是，Android 的 "开发者选项（Developer options）"中包含了有用的 "等待调试器（Wait for Debugger）"功能，它允许启动时自动暂停一个应用程序，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，逆向工程师从来不考虑公平！



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（debug app）"列表中，除非 Android Manifest 中的 android:debuggable 属性被设置为" 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，可以恢复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 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 来格式化和反混淆临时文件的内容。
- 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 文件，并在安装到目标设备/模拟器前签名。

5.9.4.1.7 库注入

在上一节中，我们学习了如何修补应用程序代码以帮助分析，但这种方法有几个局限性。例如，您希望记录通过网络发送的所有信息，而不必执行 MITM 攻击。为此，您必须修补对网络 API 的所有可能调用，这在处理大型应用程序时可能很快变得不切实际。此外，补丁对于每个应用程序都是唯一的这一事实也可以被认为是一个缺点，因为这段代码不容易重用。

使用库注入，您可以开发可重用的库，并将它们注入到不同的应用程序中，有效地使它们表现出不同的行为，而无需修改其原始源代码。这在 Windows 上被称为 DLL 注入（广泛用于修改和绕过游戏中的反作弊机制）、在 Linux 上称为 `LD_PRELOAD`，而 macOS 上称为 `DYLD_INSERT_LIBRARIES`。在 Android 和 iOS 上，一个常见的例子是当 Frida 上叫做注入模式的操作无法使用时使用 Frida 小工具（Gadget）（例如，你不能在目标设备上运行 Frida 服务端）。在这种情景下，你可以使用这一部分学到的一些方法注入小工具库。

库注入在许多情况下都是可取的，例如：

- 执行流程内省（例如，列出类、跟踪方法调用、监视访问的文件、监视网络访问、获得直接内存访问）。
- 用自己的实现支持或替换现有代码（例如，替换应给出随机数的函数）。
- 为现有应用程序引入新功能。
- 在没有原始源代码的代码上调试和修复难以捉摸的运行时错误。
- 在未 root 设备上启用动态测试（例如，使用 Frida）。

在本节中，我们将学习在 Android 上执行库注入的技术，这些技术基本上包括修补应用程序代码（smali 或原生）或使用 OS loader 本身提供的 LD_PRELOAD 功能。

5.9.4.1.7.1 修补应用程序的 Smali 代码

Android 应用程序的反编译 smali 代码可以进行修补，以引入对 `System.loadLibrary` 的调用。下面的 smali 补丁注入了一个名为 libinject.so 的库：

```
const-string v0, "inject"
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

理想情况下，您应该在应用程序生命周期的早期插入上述代码，例如在 `onCreate` 方法中。重要的是要记住添加 libinject.so 库在 APK 中 lib 文件夹的相应架构文件夹（armeabi-v7a、arm64-v8a、x86）中。最后，您需要在使用应用程序之前对其重新签名。

这项技术的一个众所周知的用例是将 Frida 小工具加载到应用程序中，特别是在未 root 设备上工作时（这是 `objection patchapk` 的基本功能）。

5.9.4.1.7.2 修补应用程序原生库

出于各种性能和安全原因，许多 Android 应用程序除了使用 Java 代码外，还使用原生代码。原生代码以 ELF 共享库的形式存在。ELF 可执行文件包括一个共享库（依赖项）列表，这些库与可执行文件链接，以使其最佳运行。可以修改此列表，以向要注入进程插入附加库。

手动修改 ELF 文件结构以注入库可能会很麻烦并且容易出错。然而，使用 LIEF（可执行插桩库格式 Library to Instrument Executable Formats）可以相对轻松地执行此任务。使用它只需要几行 Python 代码，如下所示：

```
import lief

libnative = lief.parse("libnative.so")
libnative.add_library("libinject.so") # Injection!
libnative.write("libnative.so")
```

在上面的示例中，libinject.so 库被注入为原生库（libnative.so）的依赖项，应用程序默认情况下已经加载了原生库。Frida 小工具可以使用这种方法注入到应用程序中，LIEF 的文档中对此进行了详细解释。与前一节一样，重要的是要记住将库添加到 APK 中相应的架构库文件夹中，并最终对应用程序进行重新签名。

5.9.4.1.7.3 预加载符号

上面我们研究了需要对应用程序代码进行某种修改的技术。还可以使用操作系统加载程序提供的功能将库注入到进程中。在基于 Linux 的 Android 操作系统上，您可以通过设置 `LD_PRELOAD` 环境变量来加载额外的库。

就像 ld.so 手册页指出的，从使用 LD_PRELOAD 传递的库中加载的符号始终具有优先权，即加载程序在解析符号时首先搜索这些符号，有效地覆盖了原始符号。此功能通常用于检查某些常用 libc 函数（如 fopen, read, write, strcmp 等）的输入参数，特别是在混淆程序中，在这些程序中，了解其行为可能很困难。因此，了解正在打开哪些文件或正在比较哪些字符串可能非常有价值。这里的关键思想是“函数包装”，这意味着您无法修补系统调用，如 libc 的 fopen，但您可以覆盖（包装）它，包括自定义代码，例如，为您打印输入参数，并仍然调用原始 fopen，对调用方保持透明。

在 Android 上，设置 LD_PRELOAD 与其他 Linux 发行版略有不同。如果你回想一下“平台概述”部分，Android 中的每个应用程序都是从 Zygote 派生出来的，Zygote 在 Android 启动过程中很早就启动了。因此，在 Zygote 上设置 LD_PRELOAD 是不可能的。为了解决这个问题，Android 支持 setprop（设置参数 set property）功能。下面是一个包名为 com.foo.bar 的应用程序示例。（请注意额外的 wrap.前缀）：

```
setprop wrap.com.foo.bar LD_PRELOAD=/data/local/tmp/libpreload.so
```

> 请注意，如果要预加载的库没有分配 SELinux 内容，从 Android 5.0（API 级别 21）开始，您需要禁用 SELinux 以使 LD_PRELOAD 生效，这可能需要 root 权限。

5.9.4.2. 动态插桩

5.9.4.2.1. 信息收集

在本节中，我们将学习如何使用 Frida 获取有关正在运行的应用程序的信息。

5.9.4.2.1.1 获取加载的类及其方法

您可以在 Frida CLI 中使用命令 Java 来访问 Java 运行时，并从正在运行的应用程序中检索信息。请记住，与 iOS 的 Frida 不同，在 Android 中，您需要将代码包装在 Java.perform 方法中。因此，使用 Frida 脚本更方便，例如获取加载的 Java 类及其相应的方法和字段的列表，或者用于更复杂的信息收集或插桩。下面列出了一个这样的脚本。Github 上提供了下面列出类方法的脚本。

```
// 获取加载的 Java 类和方法的列表

// 文件名: java_class_listing.js
```

```
Java.perform(function() {
    Java.enumerateLoadedClasses({
        onMatch: function(className) {
            console.log(className);
            describeJavaClass(className);
        },
        onComplete: function() {}
    });
});

// 获取方法和字段
function describeJavaClass(className) {
  var jClass = Java.use(className);
  console.log(JSON.stringify({
    _name: className,
    _methods: Object.getOwnPropertyNames(jClass.__proto__).filter(function(m)
 {
      return !m.startsWith('$') // 过滤掉与 Frida 相关的特殊属性
        || m == 'class' || m == 'constructor' // 可选
    }),
    _fields: jClass.class.getFields().map(function(f) {
      return( f.toString());
    })
  }, null, 2));
}
```

将脚本保存到名为 java_class_listing.js 的文件后，您可以告诉 Frida CLI 使用参数-l 加载它，并将其注入到由-p 指定的进程 ID 中。

```
frida -U -l java_class_listing.js -p <pid>

// Output
[Huawei Nexus 6P::sg.vantagepoint.helloworldjni]->
...

com.scottyab.rootbeer.sample.MainActivity
{
  "_name": "com.scottyab.rootbeer.sample.MainActivity",
  "_methods": [
  ...
    "beerView",
    "checkRootImageViewList",
    "floatingActionButton",
    "infoDialog",
    "isRootedText",
    "isRootedTextDisclaimer",
    "mActivity",
    "GITHUB_LINK"
```

```
  ],
  "_fields": [
    "public static final int android.app.Activity.DEFAULT_KEYS_DIALER",
...
```

考虑到输出的详细性，可以通过编程过滤系统类，使输出更具可读性，并与用例相关。

5.9.4.2.1.2 获取加载的库

您可以使用 Process 命令直接从 Frida CLI 检索与进程相关的信息。在 Process 命令中，函数 enumerateModules 列出加载到进程内存中的库。

```
[Huawei Nexus 6P::sg.vantagepoint.helloworldjni]-> Process.enumerateModules()
[
    {
        "base": "0x558a442000",
        "name": "app_process64",
        "path": "/system/bin/app_process64",
        "size": 32768
    },
    {
        "base": "0x78bc984000",
        "name": "libandroid_runtime.so",
        "path": "/system/lib64/libandroid_runtime.so",
        "size": 2011136
    },
...
```

5.9.4.2.2. 方法劫持

5.9.4.2.2.1 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/", "/dat
a/local/xbin/",
    "/data/local/bin/", "/system/sd/xbin/", "/system/bin/failsafe/", "/data/l
ocal/"};
```

```
    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 Thro
wable {
        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 T
hrowable {
                XposedBridge.log("Caught root check!");
                param.setResult(false);
            }
```

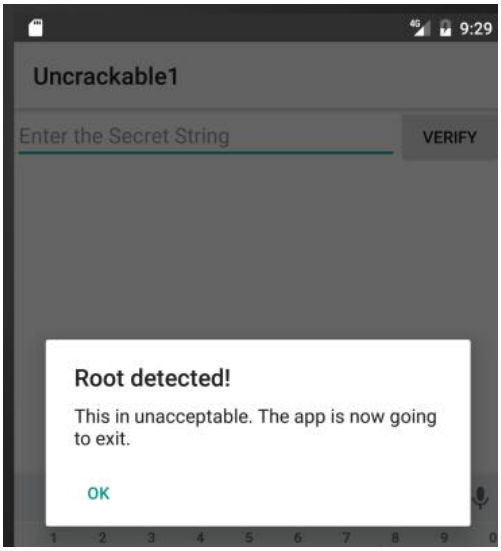
```
        });
    }
}
```

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。

5.9.4.2.2.1 Frida

我们将使用 Frida 解决适用于 Android 的 UnCrackable 应用级别 1，并演示如何轻松绕过 root 检测，从应用程序中提取秘密数据。

当您在模拟器或 root 过的设备上启动 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。

main 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.a.b;
```

```
import sg.vantagepoint.a.c;
import sg.vantagepoint.uncrackable1.a;

public class MainActivity
extends Activity {
    private void a(String string) {
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).crea
te();
        alertDialog.setTitle((CharSequence)string);
        alertDialog.setMessage((CharSequence)"This is unacceptable. The 应用 is
 now going to exit.");
        alertDialog.setButton(-3, (CharSequence)"OK", new DialogInterface.OnC
lickListener(){

            public void onClick(DialogInterface dialogInterface, int n) {
                System.exit((int)0);
            }
        });
        alertDialog.setCancelable(false);
        alertDialog.show();
    }

    protected void onCreate(Bundle bundle) {
        if (c.a() || c.b() || c.c()) {
            this.a("Root detected!");
        }
        if (b.a(this.getApplicationContext())) {
            this.a("App is debuggable!");
        }
        super.onCreate(bundle);
        this.setContentView(2130903040);
    }

    /*
     * 启用了主动的块排序
     */
    public void verify(View object) {
        object = ((EditText)this.findViewById(2130837505)).getText().toString
();
        AlertDialog alertDialog = new AlertDialog.Builder((Context)this).crea
te();
        if (a.a((String)object)) {
            alertDialog.setTitle((CharSequence)"Success!");
            object = "This is the correct secret.";
        } else {
            alertDialog.setTitle((CharSequence)"Nope...");
            object = "That's not it. Try again.";
        }
```

```
        alertDialog.setMessage((CharSequence)object);
        alertDialog.setButton(-3, (CharSequence)"OK", new DialogInterface.OnC
lickListener(){

            public void onClick(DialogInterface dialogInterface, int n) {
                dialogInterface.dismiss();
            }
        });
        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 方法调用中设置了 alertDialog.onClickListener 回调，它在成功检测到 root 之后通过 System.exit 关闭应用程序。使用 Frida，您可以通过劫持 MainActivity.a 方法或它的回调来防止应用程序退出。下面的例子演示了如何劫持 MainActivity.a 并防止它结束应用程序。

```
setImmediate(function() { //防止超时
    console.log("[*] Starting script");

    Java.perform(function() {
      var mainActivity = Java.use("sg.vantagepoint.uncrackable1.MainActivity
");
      mainActivity.a.implementation = function(v) {
         console.log("[*] MainActivity.a called");
      };
      console.log("[*] MainActivity.a modified");

    });
});
```

将代码包装在函数 setImmediate 中以防止超时（您可能需要也可能不需要这样做），然后调用 Java。使用 Frida 的方法处理 Java。然后检索 MainActivity 类的包装器并覆盖其 a 方法。与原始版本不同的是，新版 a 只在控制台输出，不退出应用程序。另一种解决方案是劫持 OnClickListener 接口的 onClick 方法。您可以覆盖 onClick 方法并防止它调用 System.exit 结束应用程序。如果你想注入你自己的 Frida 脚本，它应该完全禁用 AlertDialog，或者改变 onClick 方法的行为，这样当你点击“OK”时应用程序就不会退出。

将上述脚本另存为 uncrackable1.js 并加载：

```
$ frida -U -f owasp.mstg.uncrackable1 -l uncrackable1.js --no-pause
```

当你看到“MainActivity.a modified”消息后，应用程序将不再退出。

您现在可以尝试输入“secret string”。但是你从哪里得到?

如果你看下 sg.vantagepoint.uncrackable1.a 类，您可以看到与输入进行比较的加密字符串:

```
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)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8
dlat8FxR2GOc=", (int)0);

        try {
            arrby = sg.vantagepoint.a.a.a(a.b("8d127684cbc37c17616d806cf50473
cc"), arrby);
        }
        catch (Exception exception) {
            StringBuilder stringBuilder = new StringBuilder();
            stringBuilder.append("AES error:");
            stringBuilder.append(exception.getMessage());
            Log.d((String)"CodeCheck", (String)stringBuilder.toString());
            arrby = new byte[]{};
        }
        return string.equals((Object)new String(arrby));
    }

    public static byte[] b(String string) {
        int n = string.length();
        byte[] arrby = new byte[n / 2];
        for (int i = 0; i < n; i += 2) {
            arrby[i / 2] = (byte)((Character.digit((char)string.charAt(i), (i
nt)16) << 4) + Character.digit((char)string.charAt(i + 1), (int)16));
        }
        return arrby;
    }
}
```

注意 a 方法最后的 `string.equals` 比较以及在 try 段落里创建的字符串 `arrby`。arrby 是 sg.vantagepoint.a.a.a 函数的返回值。string.equals 比较你的输入和 arrby。所以我们需要返回 sg.vantagepoint.a.a.a 的值。

你可以简单的忽略应用程序中解密逻辑并劫持 sg.vantagepoint.a.a.a 函数来捕获返回值，而不是逆向解密逻辑来重构密钥。下面是防止检测到 root 退出并拦截解密安全字符串的全部脚本：

```
setImmediate(function() { //防止超时
    console.log("[*] Starting script");

    Java.perform(function() {
        var mainActivity = Java.use("sg.vantagepoint.uncrackable1.MainActivit
y");
        mainActivity.a.implementation = function(v) {
           console.log("[*] MainActivity.a called");
        };
        console.log("[*] MainActivity.a modified");

        var aaClass = Java.use("sg.vantagepoint.a.a");
        aaClass.a.implementation = function(arg1, arg2) {
        var retval = this.a(arg1, arg2);
        var password = '';
        for(var 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 中运行脚本并在控制台中看到“[*]sg.vantagepoint.a.a.a modified”消息后，为“secret string”输入一个随机值，然后按 verify。您应该得到类似以下的输出：

```
$ frida -U -f owasp.mstg.uncrackable1 -l uncrackable1.js --no-pause

[*] Starting script
[USB::Android Emulator 5554::sg.vantagepoint.uncrackable1]-> [*] MainActivit
y.a modified
[*] sg.vantagepoint.a.a.a modified
[*] MainActivity.a called.
[*] Decrypted: I want to believe
```

劫持函数输出解密的字符串。您提取了秘密字符串，而不必深入应用程序代码及其解密过程。

现在，您已经学习了 Android 上静态/动态分析的基础知识。当然，真正学习它的唯一途径是亲身体验：在 Android Studio 中构建自己的项目，观察你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。

在剩下的部分中，我们将介绍一些高级主题，包括进程探索、内核模块和动态执行。

5.9.4.2.3. 进程探索

在测试应用程序时，进程探索可以让测试人员深入了解应用程序进程内存。它可以通过运行时插桩来实现，并允许执行以下任务：

- 检索内存映射和加载的库。
- 搜索某些数据的出现。
- 搜索后，获取内存映射中某个偏移的位置。
- 执行内存转储，离线检查或逆向工程二进制数据。
- 在原生库运行时对其进行逆向工程。

正如你所见，这些被动任务帮助我们收集信息。此信息通常用于其他技术，例如方法劫持。

在以下部分中，您将使用 r2frida 直接从应用程序运行时检索信息。请参阅 r2frida 的官方安装说明。首先，打开一个 r2frida 会话到目标应用程序（例如 HelloWorld JNI APK），该应用程序应该在你的 Android 手机上运行（通过 USB 连接）。使用以下命令：

```
r2 frida://usb//sg.vantagepoint.helloworldjni
```

使用 `r2 frida://?`显示所有可用选项

进入 r2frida 会话后，所有命令都以\开头。例如，在 radare2 中，您可以运行 i 来显示二进制信息，但在 r2frida 中，您可以使用\ i。

5.9.4.2.3.1 内存映射及插桩

你可以通过运行\dm 来检索应用程序的内存映射，Android 中的输出可能会很长（例如 1500 到 2000 行），为了缩小搜索范围以及查看直接属于应用程序的内容，使用波浪号（~）加上包名 \dm~<package_name>：

```
[0x00000000]> \dm~sg.vantagepoint.helloworldjni
0x000000009b2dc000 - 0x000000009b361000 rw- /dev/ashmem/dalvik-/data/app/sg.v
antagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x000000009b361000 - 0x000000009b36e000 --- /dev/ashmem/dalvik-/data/app/sg.v
antagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x000000009b36e000 - 0x000000009b371000 rw- /dev/ashmem/dalvik-/data/app/sg.v
antagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x0000007d103be000 - 0x0000007d10686000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/oat/arm64/base.vdex
0x0000007d10dd0000 - 0x0000007d10dee000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/oat/arm64/base.odex
0x0000007d10dee000 - 0x0000007d10e2b000 r-x /data/app/sg.vantagepoint.hellowo
rldjni-1/oat/arm64/base.odex
0x0000007d10e3a000 - 0x0000007d10e3b000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/oat/arm64/base.odex
0x0000007d10e3b000 - 0x0000007d10e3c000 rw- /data/app/sg.vantagepoint.hellowo
rldjni-1/oat/arm64/base.odex
0x0000007d1c499000 - 0x0000007d1c49a000 r-x /data/app/sg.vantagepoint.hellowo
rldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c4a9000 - 0x0000007d1c4aa000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c4aa000 - 0x0000007d1c4ab000 rw- /data/app/sg.vantagepoint.hellowo
rldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c516000 - 0x0000007d1c54d000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/base.apk
0x0000007dbd23c000 - 0x0000007dbd247000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/base.apk
0x0000007dc05db000 - 0x0000007dc05dc000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/oat/arm64/base.art
```

当你搜索或探索应用程序内存时，你可以随时验证每个时刻你在内存映射中位置（当前偏移量所在的位置）。您只需运行\dm 即可，而无需在此列表中记录和搜索内存地址。您将在下面的“内存搜索”部分中找到示例。

如果您只对应用程序加载的模块（二进制文件和库）感兴趣，可以使用命令\il 列出所有模块：

```
[0x00000000]> \il
0x000000558b1fd000 app_process64
0x0000007dbc859000 libandroid_runtime.so
0x0000007dbf5d7000 libbinder.so
0x0000007dbff4d000 libcutils.so
0x0000007dbfd13000 libhwbinder.so
```

```
0x0000007dbea00000 liblog.so
0x0000007dbcf17000 libnativeloader.so
0x0000007dbf21c000 libutils.so
0x0000007dbde4b000 libc++.so
0x0000007dbe09b000 libc.so
...
0x0000007d10dd0000 base.odex
0x0000007d1c499000 libnative-lib.so
0x0000007d2354e000 frida-agent-64.so
0x0000007dc065d000 linux-vdso.so.1
0x0000007dc065f000 linker64
```

正如您所料，您可以将库的地址与内存映射相关联：例如，应用程序的原生库位于 `0x0000007d1c499000`，优化的索引（base.odex）位于 `0x0000007d10dd0000`

您也可以使用 objection 来显示相同的信息。

```
$ objection --gadget sg.vantagepoint.helloworldjni explore

sg.vantagepoint.helloworldjni on (google: 8.1.0) [usb] # 内存列表模块
Save the output by adding `--json modules.json` to this command

Name                                                Base           Size
        Path
--------------------------------------------------  -------------  --------------
------  ------------------------------------------------------------------------
app_process64                                       0x558b1fd000   32768 (32.0 Ki
B)      /system/bin/app_process64
libandroid_runtime.so                               0x7dbc859000   1982464 (1.9 M
iB)     /system/lib64/libandroid_runtime.so
libbinder.so                                        0x7dbf5d7000   557056 (544.0
KiB)    /system/lib64/libbinder.so
libcutils.so                                        0x7dbff4d000   77824 (76.0 Ki
B)      /system/lib64/libcutils.so
libhwbinder.so                                      0x7dbfd13000   163840 (160.0
KiB)    /system/lib64/libhwbinder.so
base.odex                                           0x7d10dd0000   442368 (432.0
KiB)    /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
libnative-lib.so                                    0x7d1c499000   73728 (72.0 Ki
B)      /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
```

你甚至可以直接看到 Android 文件系统中二进制文件的大小和路径。

5.9.4.2.3.2 内存搜索

内存搜索是一种非常有用的技术，用于测试应用程序内存中可能存在的敏感数据。

请参阅 r2frida 关于搜索命令（\/？）的帮助了解搜索命令并获取选项列表。以下仅显示其中一部分：

```
[0x00000000]> \/?
 /       search
 /j      search json
 /w      search wide
 /wj     search wide json
 /x      search hex
 /xj     search hex json
...
```

您可以使用搜索设置来调整搜索。例如，`\e search.quiet=true` 将仅打印结果并隐藏搜索进度：

```
[0x00000000]> \e~search
e search.in=perm:r--
e search.quiet=false
```

现在，我们将继续使用默认值，重点关注字符串搜索。这个应用程序实际上非常简单，它从其原生库中加载字符串“Hello from C++”，并将其显示给我们。你可以从搜索“Hello”开始，看看 r2frida 找到了什么：

```
[0x00000000]> \/ Hello
Searching 5 bytes: 48 65 6c 6c 6f
...
hits: 11
0x13125398 hit0_0 HelloWorldJNI
0x13126b90 hit0_1 Hello World!
0x1312e220 hit0_2 Hello from C++
0x70654ec5 hit0_3 Hello
0x7d1c499560 hit0_4 Hello from C++
0x7d1c4a9560 hit0_5 Hello from C++
0x7d1c51cef9 hit0_6 HelloWorldJNI
0x7d30ba11bc hit0_7 Hello World!
0x7d39cd796b hit0_8 Hello.java
0x7d39d2024d hit0_9 Hello;
0x7d3aa4d274 hit0_10 Hello
```

现在你想知道这些地址到底在哪里。您可以通过运行 \dm.来为所有@@全局匹配 hit0_*：

```
[0x00000000]> \dm.@@ hit0_*
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (re
gion space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (re
gion space) (deleted)
```

```
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (re
gion space) (deleted)
0x00000000703c2000 - 0x00000000709b5000 rw- /data/dalvik-cache/arm64/system@f
ramework@boot-framework.art
0x0000007d1c499000 - 0x0000007d1c49a000 r-x /data/app/sg.vantagepoint.hellowo
rldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c4a9000 - 0x0000007d1c4aa000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/lib/arm64/libnative-lib.so
0x0000007d1c516000 - 0x0000007d1c54d000 r-- /data/app/sg.vantagepoint.hellowo
rldjni-1/base.apk
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d396bc000 - 0x0000007d3a998000 r-- /system/framework/arm64/boot-fram
ework.vdex
0x0000007d396bc000 - 0x0000007d3a998000 r-- /system/framework/arm64/boot-fram
ework.vdex
0x0000007d3a998000 - 0x0000007d3aa9c000 r-- /system/framework/arm64/boot-ext.
vdex
```

此外，您可以搜索宽字符版本字符串（\/w），并再次检查其内存区域：

```
[0x00000000]> \/w Hello
Searching 10 bytes: 48 00 65 00 6c 00 6c 00 6f 00
hits: 6
0x13102acc hit1_0 480065006c006c006f00
0x13102b9c hit1_1 480065006c006c006f00
0x7d30a53aa0 hit1_2 480065006c006c006f00
0x7d30a872b0 hit1_3 480065006c006c006f00
0x7d30bb9568 hit1_4 480065006c006c006f00
0x7d30bb9a68 hit1_5 480065006c006c006f00

[0x00000000]> \dm.@@ hit1_*
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (re
gion space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (re
gion space) (deleted)
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d30a00000 - 0x0000007d30c00000 rw-
0x0000007d30a00000 - 0x0000007d30c00000 rw-
```

它们与前面的一个字符串（0x0000007d30a00000）位于同一 rw 区域。请注意，搜索宽字符版本的字符串有时是找到它们的唯一方法，您将在下一节中看到。

内存搜索非常有用，可以快速知道某些数据是否位于主应用程序二进制文件、共享库内或其他区域。您还可以使用它来测试应用程序的行为，即数据如何保存在内存中。例如，您可以分析一个

执行登录的应用程序，并搜索出现的用户密码。此外，您可以检查登录完成后是否仍能在内存中找到密码，以验证此敏感数据在使用后是否已从内存中删除。

此外，您可以使用这种方法来定位和提取加密密钥。例如，在应用程序加密/解密数据并在内存中处理密钥的情况下，而不是使用 AndroidKeyStore API。有关更多详细信息，请参阅“Android 加密 API”一章中的“测试密钥管理”部分。

5.9.4.2.3.3 内存转储

你可以用 objection 和 Fridump 转储应用程序的进程内存。为了在未 root 设备上利用这些工具，Android 应用程序必须用 `frida-gadget.so` 重新打包并重新签名。有关此过程的详细说明，请参阅“未 root 设备的动态分析”一节。要在 root 设备上使用这些工具，只需安装并运行 frida-server 即可。

> 注意：当使用这些工具时，您可能会遇到一些通常可以忽略的内存访问冲突错误。这些工具注入 Frida 代理，并尝试转储应用程序的所有映射内存，而不管访问权限（读/写/执行）。因此，当注入的 Frida 代理尝试读取不可读的区域时，它将返回相应的内存访问冲突错误。有关更多详细信息，请参阅上一节“内存映射和插桩”。

使用 objection 有可能可以使用命令 memory dump all 在设备上转储正在运行的进程的所有内存。

```
$ objection --gadget sg.vantagepoint.helloworldjni explore

sg.vantagepoint.helloworldjni on (google: 8.1.0) [usb] # memory dump all /Use
rs/foo/memory_Android/memory

Will dump 719 rw- images, totalling 1.6 GiB
Dumping 1002.8 MiB from base: 0x14140000  [----------------------------------
--]    0%  00:11:03(session detach message) process-terminated
Dumping 8.0 MiB from base: 0x7fc753e000  [####################################
#]  100%
Memory dumped to file: /Users/foo/memory_Android/memory
```

> 在这种情况下，出现了一个错误，这可能是由于内存访问违规，正如我们已经预料到的那样。只要我们能够在文件系统中看到提取的转储，就可以安全地忽略此错误。如果您有任何问题，第一步是在运行 objection 时启用调试参数-d，或者，如果这没有帮助，请在 objection 的 GitHub 中提交 issue。

接下来，我们可以用 radare2 找到“Hello from C++”字符串：

```
$ r2 /Users/foo/memory_Android/memory
[0x00000000]> izz~Hello from
1136 0x00065270 0x00065270  14  15 () ascii Hello from C++
```

或者，您可以使用 Fridump。这次，我们将输入一个字符串，看看是否可以在内存转储中找到它。为此，请打开 MASTG 黑客游乐场应用程序，导航到“OMTG_DATAST_002_LOGGING”，并在密码字段中输入“owasp-mstg”。接下来，运行 Fridump：

```
python3 fridump.py -U sg.vp.owasp_mobile.omtg_android -s

Current Directory: /Users/foo/git/fridump
Output directory is set to: /Users/foo/git/fridump/dump
Starting Memory dump...
Oops, memory access violation!------------------------------] 0.28% Complete
Progress: [##################################################] 99.58% Complet
e
Running strings on all files:
Progress: [##################################################] 100.0% Complet
e

Finished!
```

> 提示：如果您想查看更多详细信息，例如引发内存访问冲突的区域，请通过包含参数-v 来启用细节信息。

这将需要一段时间，直到它完成，你会在 dump 文件夹中得到一个*.data 文件集合。当添加-s 参数时，所有字符串都从转储的原始内存文件中提取并添加到 strings.txt，也存储在 dump 目录中。

```
ls dump/
dump/1007943680_dump.data dump/357826560_dump.data  dump/630456320_dump.data
 ... strings.txt
```

最后在 dump 目录搜索输入的字符串:

```
$ grep -nri owasp-mstg dump/
Binary file dump//316669952_dump.data matches
Binary file dump//strings.txt matches
```

“owasp-mstg”字符串可以在其中一个转储文件以及已处理的字符串文件中找到。

5.9.4.2.3.4 运行时逆向工程

运行时逆向工程可以被视为逆向工程的即时版本，其中您没有主机的二进制数据。相反，你将直接从应用程序的内存中进行分析。

我们将继续使用 HelloWorld JNI 应用程序，使用 r2frida 新建会话 r2 frida://usb//sg.vantagepoint.helloworldjni 并且你可以使用 \i 命令显示目标的二进制文件信息：

```
[0x00000000]> \i
arch                 arm
bits                 64
os                   linux
pid                  13215
uid                  10096
objc                 false
runtime              V8
java                 true
cylang               false
pageSize             4096
pointerSize          8
codeSigningPolicy    optional
isDebuggerAttached   false
cwd                  /
dataDir              /data/user/0/sg.vantagepoint.helloworldjni
codeCacheDir         /data/user/0/sg.vantagepoint.helloworldjni/code_cache
extCacheDir          /storage/emulated/0/Android/data/sg.vantagepoint.hellowor
ldjni/cache
obbDir               /storage/emulated/0/Android/obb/sg.vantagepoint.helloworl
djni
filesDir             /data/user/0/sg.vantagepoint.helloworldjni/files
noBackupDir          /data/user/0/sg.vantagepoint.helloworldjni/no_backup
codePath             /data/app/sg.vantagepoint.helloworldjni-1/base.apk
packageName          sg.vantagepoint.helloworldjni
androidId            c92f43af46f5578d
cacheDir             /data/local/tmp
jniEnv               0x7d30a43c60
```

使用\is <lib>搜索全部中信模块的符号，例如\is libnative-lib.so。

```
[0x00000000]> \is libnative-lib.so

[0x00000000]>
```

在这个例子中结果为空。或者，您可能更愿意研究导入/导出。例如，列出导入\ii <lib>：

```
[0x00000000]> \ii libnative-lib.so
0x7dbe1159d0 f __cxa_finalize /system/lib64/libc.so
0x7dbe115868 f __cxa_atexit /system/lib64/libc.so
```

列出导出 `\iE <lib>`:

```
[0x00000000]> \iE libnative-lib.so
0x7d1c49954c f Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

> 较大的二进制文件推荐使用~..将输出通过管道输出到内部程序，例如\ii libandroid_runtime.so~..（如果不这样，像这个二进制文件，至少会有 2500 行显示到你的终端上）

接下来，您可能需要查看**当前加载**的 Java 类:

```
[0x00000000]> \ic~sg.vantagepoint.helloworldjni
sg.vantagepoint.helloworldjni.MainActivity
```

列出类字段:

```
[0x00000000]> \ic sg.vantagepoint.helloworldjni.MainActivity~sg.vantagepoint.
helloworldjni
public native java.lang.String sg.vantagepoint.helloworldjni.MainActivity.str
ingFromJNI()
public sg.vantagepoint.helloworldjni.MainActivity()
```

注意，我们已经按包名进行了过滤，因为这是 `MainActivity`，它包括来自 Android `Activity` 类的所有方法。

您还可以显示有关类加载器的信息:

```
[0x00000000]> \icL
dalvik.system.PathClassLoader[
 DexPathList[
  [
   directory "."]
  ,
  nativeLibraryDirectories=[
   /system/lib64,
    /vendor/lib64,
    /system/lib64,
    /vendor/lib64]
  ]
 ]
java.lang.BootClassLoader@b1f1189dalvik.system.PathClassLoader[
 DexPathList[
  [
   zip file "/data/app/sg.vantagepoint.helloworldjni-1/base.apk"]
  ,
  nativeLibraryDirectories=[
```

```
   /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64,
    /data/app/sg.vantagepoint.helloworldjni-1/base.apk!/lib/arm64-v8a,
    /system/lib64,
    /vendor/lib64]
  ]
 ]
```

接下来，假设您对 libnative-lib.so 0x7d1c49954c f Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 导出的方法感兴趣。 您可以使用 s 0x7d1c49954c 查找该地址，分析该函数 af 并打印其反汇编 pd 10 的 10 行：

```
            ;-- sym.fun.Java_sg_vantagepoint_helloworldjni_MainActivity_strin
gFromJNI:
┌ (fcn) fcn.7d1c49954c 18
│   fcn.7d1c49954c (int32_t arg_40f942h);
│           ; arg int32_t arg_40f942h @ x29+0x40f942
│           0x7d1c49954c      080040f9       ldr x8, [x0]
│           0x7d1c499550      01000090       adrp x1, 0x7d1c499000
│           0x7d1c499554      21801591       add x1, x1, 0x560          ; hit0
_4
│           0x7d1c499558      029d42f9       ldr x2, [x8, 0x538]        ; [0x5
38:4]=-1 ; 1336
│           0x7d1c49955c      4000           invalid
```

注意使用 hit0_4 标记的行对应我们之前找到的字符串：

```
0x7d1c499560 hit0_4 Hello from C++
```

了解更多，请参考 r2frida wiki。

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插桩，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 是一个存储在启动镜像中的小型 CPIO 档案。它包含引导时需要的几个文件，然后才能装入实际的 root 文件系统。在 Android 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 adbd 作为 root 运行。要在任何 Android 设备上修改 initrd，可以使用 TWRP 备份原始启动镜像，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
```

要提取引导镜像的内容，请使用 Krzysztof Adamski 的 how-To 中所描述的 abotimg 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_debu
g=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 应用受到严格的限制和沙盒化，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并公然对自身和环境撒谎的内核，在很大程度上击败了恶意软件作者（或普通开发人员）可能为您建立的大多数逆向防御。

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，然后将系统调用号用作函数指针表（在 Android 上称为 sys_call_table）的偏移量。

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，目前的原生 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。Google 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 Google 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 androidmsm-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_RWX=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
```

当完成编辑后保存.config 文件，编译内核.

```
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-standalon-toolchain.sh：

```
cd android-ndk-rXXX
build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-24 --i
nstall-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，备份一下您的设备的原始启动镜像，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/mmcb
lk0p24
lrwxrwxrwx root     root              1970-08-30 22:31 aboot -> /dev/block/mm
cblk0p6
lrwxrwxrwx root     root              1970-08-30 22:31 abootb -> /dev/block/m
mcblk0p11
lrwxrwxrwx root     root              1970-08-30 22:31 boot -> /dev/block/mmc
blk0p19
(...)
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 和有关引导镜像结构的信息。有各种各样的工具可以做到这一点；我使用吉勒 · 格兰杜的 abootimg 工具。安装该工具并在启动镜像上运行以下命令：

```
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)。

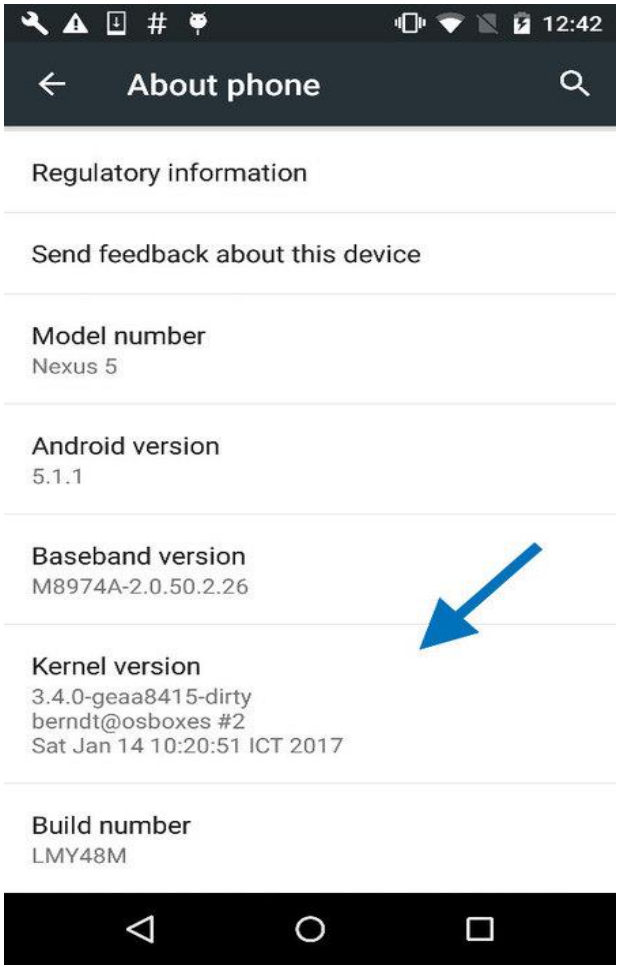
现在可以使用 fastboot 测试新内核。`fastboot boot` 命令允许您运行内核而不需要实际写入它(一旦您确定一切正常，您可以使用 fastboot flash,使更改永久化，但是您不必这样做)。使用以下命令在快速启动模式下重新启动设备：

```
adb reboot bootloader
```

然后使用 `fastboot boot` 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)加入新构建的内核和原始 ramdisk。

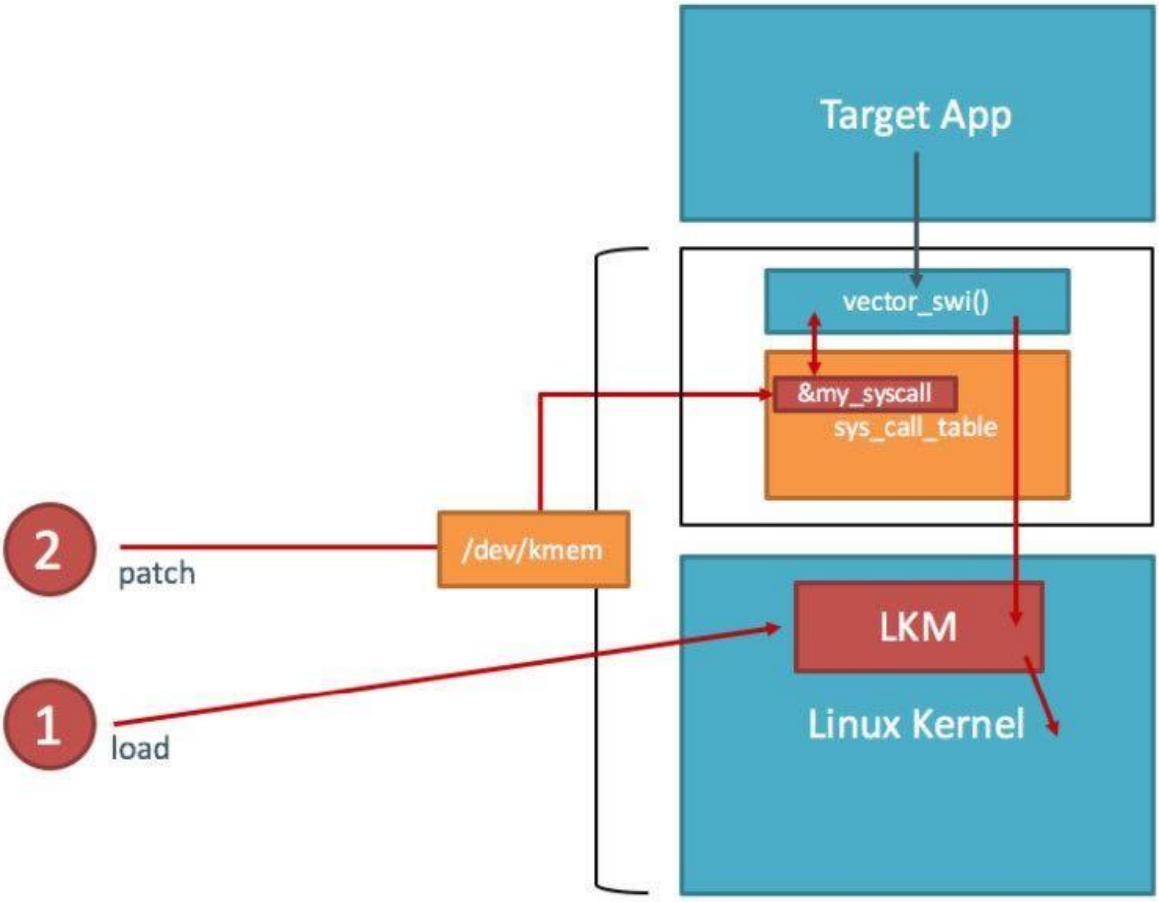
```
fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk
-offset 0x2900000 --tags-offset 0x2700000 -c "console=ttyHSL0,115200,n8 andro
idboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

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劫持

系统调用 Hooking 允许您攻击任何依赖于内核提供功能的反逆向防御。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，由 You Dong Hoon 在 2011 年 4 月 4 日的 Phrack 杂志-“基于 Android 平台的 Linux 内核 rootkit”中描述过。



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，您需要劫持一个用于打开(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, facessat, stat, fstat 等。目前，您只需要劫持 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有效的工具链。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 `lsmod` 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 0xbf000000 (PO)
```

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劫持的简单方法)。为此，改编了 Dong-Hoon You's Phrack article 中的代码。不过，可以使用 file 接口代替 mmap，因为后者可能会导致内核错误。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_num
ber);
  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 5.0 (API 级别 21)开始，所有的可执行文件必须编译支持 PIE 。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_ut
il 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。

```
$ adb shell cat /proc/kallsyms | grep new_openat
bf000000 t new_openat    [kernel_hook]
```

现在您已经有了所有覆盖 sys_call_table 所需内容。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。请注意，可以使用其他系统调用轻松找到该文件，您需要做更多的工作来正确隐藏文件，包括劫持 stat、access 和其他系统调用。

文件隐藏当然只是冰山一角：使用内核模块可以完成很多工作，包括绕过许多 root 检测措施、完整性检查和反调试措施。您可以在 Bernhard Mueller 的黑客论文[#Mueller]的“案例研究”部分找到更多示例。

5.9.6. 参考文献

- Bionic - https://github.com/android/platform_bionic
- Attacking Android Applications with Debuggers (19 January 2015) - https://blog.netspi.com/attacking-android-applications-with-debuggers/
- [#josse] Sébastien Josse, Dynamic Malware Recompilation (6 January 2014) - 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 (4 April 2011 - Phrack Magazine)
- [#mueller] Bernhard Mueller, Hacking Soft Tokens. Advanced Reverse Engineering on Android (2016) - https://packetstormsecurity.com/files/138504/HITB_Hacking_Soft_Tokens_v1.2.pdf

5.10. Android 反逆向防御

5.10.1. 测试 ROOT 环境检测(MSTG-RESILIENCE-1)

5.10.1.1. 概述

在反逆向的背景下，root 检测的目标是使在已经 root 的设备上运行应用程序更加困难，这反过来又阻止了一些逆向工程师喜欢使用的工具和技术。像大多数其他防御措施一样，root 检测本身并不十分有效，但实施分散在整个应用程序中的多个 root 检查可以提高整个反逆向方案的有效性。

对于 Android 系统来说，我们定义的“root 检测”有些笼统，其中包括对自定义 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

在下一节中，我们列出了一些你会遇到的常见根检测方法。你会发现其中一些方法在《OWASP 移动测试指南》所附的适用于 Android 的 OWASP UnCrackable 应用中实现。

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 兼容性测试的设备型号列表进行比较。Google 建议将该功能用作“附加深度防御标志以作为反滥用系统的一部分”。

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，但如何确定尚不清楚。

使用该 API，应用将会调用 **`SafetyNetApi.attest`** 方法（返回值是带有认证结果的 JWS 信息）并检查是否存在下列字段：

- ctsProfileMatch：如果为“true”设备配置文件将与 Google 列出的设备之一匹配。
- basicIntegrity：如果为“true”，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，用于验证被调用的应用的识别信息。当 API 无法识别 APK 文件的基础信息时，将无法取得上述参数。

下面是一个认证结果示例：

```
{
  "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 与 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 劫持）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 开源计划的源文件构建。
- 设备系统镜像是 Beta 版本或者开发人员预览版本（包含 Android 预览计划）。

5.10.1.2.1.2 使用 SafetyNetAPI.attest 相关建议

- 在服务器上应使用加密安全的随机函数创建一个较大的随机数（16 个字节或更长），以让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）。
- 所有 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。
- 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 认证 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 配额上限导致验证失败，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所有用户。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，则可以提交表单以为你的 API 密钥请求短期或长期的配额增加。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. 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。这些查找包括检查 busybox 并尝试在不同位置打开 su 二进制文件：：

```
/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 原生代码轻松实现。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 filepa
th) {
  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 中，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

Supersu-迄今为止最流行的 root 工具，它运行名为 daemonsu 的身份认证守护进程，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。下面是一个在 rootinspector 中实现的例子

```
    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.ramdroid.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 或经过修改。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 检查自定义 Android 版本

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 标签中的测试关键字，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

使用 jdb、DDMS、strace 和/或内核模块运行执行跟踪，以了解应用程序在做什么。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。这些交互是 root 检测的可靠迹象。一次一个地识别和停用 root 检测机制。 如果您正在执行黑盒安全评估，则禁用 root 检测机制是您的第一步。

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（尽量不要破坏你的环境！）。
- 卸载/proc 以防止读取进程列表。有时，/proc 不可用足以绕过此类检查。
- 用 Frida 或 Xposed 在 Java 层和原生层上进行劫持。通过伪造应用程序的返回值，隐藏文件内容和进程。
- 使用内核模块劫持低等级的 API。
- 篡改 app 来移除检查。

5.10.1.3. 有效性评估

检查 root 检测机制时，包括以下标准：

- 应用程序中分散了多种检测方法（而不是将所有内容都放在一种方法中）。
- root 检测机制在多个层面的 API 上运行（Java API、原生库函数、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 检测容易被绕过，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. 测试反调试检测(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。

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第一时间防止调试器挂载。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，并分布在整个应用程序中。

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 的调试器在原生层上进行调试。一个好的反调试方案应当能够抵御两种类型的调试。

5.10.2.2. JDWP 反调试

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。

5.10.2.2.1. 检查 ApplicationInfo 中的 Debuggable 属性

之前已经介绍过 **android:debuggable** 属性。Android Manifest 中的此属性确定是否为该应用程序启动 JDWP 线程。可以通过应用程序的 **ApplicationInfo** 对象以编程方式检测其值。如果设置了该属性，则 manifest 已被篡改并允许调试。

```
    public static boolean isDebuggable(Context context){

        return ((context.getApplicationContext().getApplicationInfo().flags &
 ApplicationInfo.FLAG_DEBUGGABLE) != 0);

    }
```

5.10.2.2.2. isDebuggerConnected

虽然这对于逆向工程来说可能是显而易见的，但您可以从 android.os.Debug 类的 **isDebuggerConnected** 来确定是否连接了调试器。

```
    public static boolean detectDebugger() {
        return Debug.isDebuggerConnected();
    }
```

通过访问 DvmGlobals 结构体，可以通过原生代码调用相同的 API。

```
JNIEXPORT jboolean JNICALL Java_com_test_debugging_DebuggerConnectedJNI(JNIen
v * env, jobject obj) {
    if (gDvm.debuggerConnected || gDvm.debuggerActive)
        return JNI_TRUE;
    return JNI_FALSE;
}
```

5.10.2.2.3. 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;
  }
}
```

5.10.2.2.4. 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 {
    /*
     * 一些选项值得被篡改 :)
     */

    bool        jdwpAllowed;        // 这个进程是否允许调试?
```

```
    bool        jdwpConfigured;     // 是否提供调试信息?
    JdwpTransportType jdwpTransport;
    bool        jdwpServer;
    char*       jdwpHost;
    int         jdwpPort;
    bool        jdwpSuspend;

    Thread*     threadList;

    bool        nativeDebuggerActive;
    bool        debuggerConnected;      /* 调试器或 DDMS 已连接*/
    bool        debuggerActive;         /* 调试器正在发起请求*/
    JdwpState*  jdwpState;

};
```

例如：将 gDvm.methDalvikDdmcServer_dispatch 函数指针设置为 NULL 会使 JDWP 线程崩溃：

```
JNIEXPORT jboolean JNICALL Java_poc_c_crashOnInit ( JNIEnv* env , jobject ) {
  gDvm.methDalvikDdmcServer_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", FM
T, ##__VA_ARGS__)

// Vtable 结构. 只是为了让搞乱更直观

struct VT_JdwpAdbState {
    unsigned long x;
```

```
    unsigned long y;
    void * JdwpSocketState_destructor;
    void * _JdwpSocketState_destructor;
    void * Accept;
    void * showmanyc;
    void * ShutDown;
    void * ProcessIncoming;
};

extern "C"

JNIEXPORT void JNICALL Java_sg_vantagepoint_jdwptest_MainActivity_JDWPfun(
        JNIEnv *env,
        jobject /* 这里 */) {

    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(li
b, "_ZTVN3art4JDWP12JdwpAdbStateE");

        if (vtable == 0) {
            log("Couldn't resolve symbol '_ZTVN3art4JDWP12JdwpAdbStateE'.\n
");
        }else {

            log("Vtable for JdwpAdbState at: %08x\n", vtable);

            // 让快乐开始!

            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;

            // 重置权限&清空缓存

            mprotect((void *)page, pagesize, PROT_READ);

        }
    }
}
```

5.10.2.3. 传统反调试

在 Linux 上，**ptrace** 系统调用用于观察和控制进程（tracee）的执行，并检查和更改该进程的内存和寄存器。**ptrace** 是在原生代码中实现系统调用跟踪和断点调试的主要方法。大多数 JDWP 反调试技巧（对于基于计时器的检查可能是安全的）不会捕获基于 **ptrace** 的经典调试器，因此，许多 Android 反调试技巧包括 **ptrace**，通常利用一个进程一次只能连接一个调试器这一事实。

5.10.2.3.1. 检查 TracerPid

当您调试应用程序并在原生代码上设置断点时，Android Studio 会将所需文件复制到目标设备，并启动 **lldb-server**，其将使用 **ptrace** 连接到进程。从现在起，如果您检查已调试进程的状态文件（**/proc/<pid>/status** 或**/proc/self/status**），您将看到“TracerPid”字段的值不同于 0，这是调试的标志。

> 请记住，**这仅适用于原生代码**。如果您调试的是纯 Java/Kotlin 应用程序，“TracerPid”字段的值应该为 0。

这种技术通常应用于基于 C 语言的 JNI 原生库中，如 Google 的 gperftools（Google Performance Tools））堆检查器实现的 **IsDebuggerAttached** 方法所示。然而，如果您希望将此检查作为 Java/Kotlin 代码的一部分，您可以参考 Tim Strazzere 的反模拟器项目中 **hasTracerPid** 方法的 Java 实现。

当您自己尝试实现这种方法时，可以使用 ADB 手动检查 TracerPid 的值。以下清单使用 Google 的 NDK 示例应用 `hello-jni (com.example.hellojni)`在连接 Android Studio 的调试器后执行检查：

```
$ adb shell ps -A | grep com.example.hellojni
u0_a271      11657   573 4302108  50600 ptrace_stop         0 t com.example.h
ellojni
$ adb shell cat /proc/11657/status | grep -e "^TracerPid:" | sed "s/^TracerPi
d:\t//"
TracerPid:      11839
$ adb shell ps -A | grep 11839
u0_a271      11839 11837   14024   4548 poll_schedule_timeout 0 S lldb-server
```

你可以查看 `com.example.hellojni (PID=11657)`的状态文件包含的 `TracerPID` 为 11839，这可以识别为 lldb-server 进程。

5.10.2.3.2. 使用 Fork 和 ptrace

通过类似于以下简单示例的代码，可以通过分叉一个子进程并将其作为调试器附加到父进程来防止调试进程：

```
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);

          /* 继续父进程 */
          ptrace(PTRACE_CONT, NULL, NULL);
        }
    }
}
```

如果子进程已连接，则之后尝试附加到父进程的操作将失败。我们可以通过将代码编译成 JNI 函数并将其打包到我们在设备上运行的应用程序来验证这一点。

```
root@android:/ # ps | grep -i anti
u0_a151   18190 201   1535844 54908 ffffffff b6e0f124 S sg.vantagepoint.antid
ebug
u0_a151   18224 18190 1495180 35824 c019a3ac b6e0ee5c S sg.vantagepoint.antid
ebug
```

尝试使用 gdbserver 附加到父进程失败并返回错误：

```
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
```

然而，通过杀死子进程并“释放”父进程不被调试，你可以轻松绕过这一错误。因此，您通常会发现更复杂的方案，涉及多个进程和线程，以及某种形式的监视来阻止篡改。常用方法包括

- 分叉多个相互调试的进程
- 跟踪运行进程以确保子进程活着

- 监视**/proc** 文件系统中的值，例如**/proc/pid/status** 中的 TracerPID

让我们来看一下对上述方法的一个简单改进。在初始分叉之后，我们在父线程中启动一个额外的线程，该线程持续监视子线程的状态。根据应用程序是在调试模式还是发布模式下构建的（由 manifest 中的 **android:debuggable** 属性指示），子进程应该执行以下操作之一：

- 在发布模式下：对 ptrace 的调用失败，子进程立即崩溃，出现分段错误（退出代码 11）。
- 在调试模式下：对 ptrace 的调用起成功，子进程应该无限期地运行。因此，对 **waitpid（child_pid**）的调用永远不会返回。如果真是这样的话，有些事情是可疑的，我们会杀死整个进程组。

以下是使用 JNI 函数实现此改进的完整代码：

```
#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);

    /* 子进程状态应该始终不变. */

    _exit(0); // 提交 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 {
                    // 进程已退出
                    _exit(0);
                }
            }
        }

    } else {
        pthread_t t;

        /* 启动监视线程*/
        pthread_create(&t, NULL, monitor_pid, (void *)NULL);
    }
}

JNIEXPORT void JNICALL
Java_sg_vantagepoint_antidebug_MainActivity_antidebug(JNIEnv *env, jobject in
stance) {

    anti_debug();
}
```

同样，我们将其打包到 Android 应用程序中，看看它是否有效。与之前一样，当我们运行应用程序的调试版本时，会出现两个进程。

```
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，如果我们此时结束子进程，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
```

为了绕过这个问题，我们必须稍微修改应用程序的行为（最简单的方法是用 NOPs 修补对_exit 的调用，并劫持 libc.so 中的函数_exit）。此时，我们已经进入了众所周知的“军备竞赛”：实施更复杂的防御形式以及绕过它总是可能的。

5.10.2.4. 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，劫持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 指令覆盖它来禁用不想要的行为。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 劫持 Java 和原生层上的 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。如果其他措施没有效果，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。

5.10.2.4.1 绕过实例: 适用于 Android 的 UnCrackable 应用程序级别 2

在处理混淆的应用程序时，经常会发现开发人员故意在原生库中“隐藏”数据和功能。在适用于 Android 的 UnCrackable 应用程序级别 2 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原生方法的 **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 的其他目的方案。

5.10.2.5. 有效性评估

检查反调试机制，包括以下条件：

- 附加基于 jdb 和基于 ptrace 的调试器会失败，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）。
- 反调试防御可工作在多个 API 层（Java，原生库函数，编译器/系统调用）。
- 这些机制在某种程度上是原创的（而不是从 StackOverflow 或其他来源复制和粘贴的）。

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：通过劫持单个 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)

5.10.3.1. 概述

这里有两个文件完整性相关主题

1. 代码完整性检查：在“Android 系统上的篡改和逆向工程”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，原生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 层和原生层上实现。 这个方法是在

适当位置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

完整性检查通常计算选定文件的校验和或散列。通常受保护的文件包括：

- AndroidManifest.xml,
- 类文件 *.dex
- 原生库(*.so)

以下来自 Android Cracking 博客的示例实现计算出 classes.dex 的 CRC，并将其与期望值进行比较

```
private void crcTest() throws IOException {
 boolean modified = false;
 // 要求 dex crc 值作为文本字符串存储.
 // 其可以是任何不可见布局元素
 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 已被修改
  modified = true;
 }
 else {
  // dex 没有被篡改
  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（如 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 已注册为安全提供程序。
2. 使用密钥初始化 HMAC（可以将其存储在 keystore 中）。
3. 获取需要 HMAC 的内容的字节数组。
4. 对字节码在 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 产出的的结果。

使用 BouncyCastle 验证 HMAC 时，请完成以下过程：

1. 确保 BouncyCastle 或 SpongyCastle 已注册为安全提供程序。
2. 将消息和 HMAC 字节提取为单独的数组。
3. 重复生成 HMAC 的过程的步骤 1-4。
4. 将提取的 HMAC 字节与步骤 3 的结果进行比较。

当基于 Android Keystore 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** 时方便的 HMAC 实现方法：

```
public enum HMACWrapper {
    HMAC_512("HMac-SHA512"), //请注意，这是 BC 提供程序的规范
    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 | NoSuchAlgori
thmException e) {
            //处理它们
        }
    }

    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完整性检查

1. 修改防调试功能。只需使用 NOP 指令覆盖相关的字节码或原生代码，即可禁用不需要行为。
2. 使用 Frida 或 Xposed 劫持 Java 和原生层上的文件系统 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。当该进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。

有关补丁、代码注入和内核模块的示例，请参阅“Android 上的篡改和逆向工程”一章。

5.10.3.1.3.2. 绕过存储完整性检查

1. 如测试设备绑定部分所述，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. 有效性评估

应用程序源代码的完整性检查：

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和/或终止。 继续绕过防御工作并对以下问题做出回应：

- 是否可以轻易地绕过检查机制（例如：通过劫持单个 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?

存储完整性检查:

应用一种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. 测试逆向工程工具检测(MSTG-RESILIENCE-4)

5.10.4.1. 概述

逆向工程师通常使用的工具、框架和应用程序的存在可能表明有人试图对应用程序进行逆向工程。其中一些工具只能在 root 设备上运行,而另一些工具则迫使应用程序进入调试模式,或者依赖于在手机上启动后台服务。因此,应用程序可以实现不同的方式来检测逆向工程攻击并对其作出反应,例如通过终止自身。

5.10.4.2. 检查方法

您可以通过查找关联的应用程序包,文件,进程或其他工具特定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/劫持工具,这将在下面进行讨论。

例如,在已 root 设备上的默认配置中,Frida 作为 frida-server 在设备上运行。当您显式连接到目标应用程序时(例如,通过 frida-trace 或 Frida REPL),Frida 会将 frida-agent 注入应用程序的内存。因此,您可能希望在附加到应用程序后(而不是之前)在那里找到它。如果您检查`/proc/<pid>/maps`,您会发现 frida-agent 为 frida-agent-64.so:

```
bullhead:/ # cat /proc/18370/maps | grep -i frida
71b6bd6000-71b7d62000 r-xp  /data/local/tmp/re.frida.server/frida-agent-64.so
71b7d7f000-71b7e06000 r--p  /data/local/tmp/re.frida.server/frida-agent-64.so
71b7e06000-71b7e28000 rw-p  /data/local/tmp/re.frida.server/frida-agent-64.so
```

另一种方法（也适用于未 root 设备）包括将 frida-gadget 嵌入 APK，并强制应用程序将其作为其原生库之一加载。如果你在启动应用程序后检查应用程序内存映射（无需显式连接），你会发现嵌入的 frida-gadget 为 libfrida-gadget.so。

```
bullhead:/ # cat /proc/18370/maps | grep -i frida

71b865a000-71b97f1000 r-xp  /data/app/sg.vp.owasp_mobile.omtg_android-.../lib
/arm64/libfrida-gadget.so
71b9802000-71b988a000 r--p  /data/app/sg.vp.owasp_mobile.omtg_android-.../lib
/arm64/libfrida-gadget.so
71b988a000-71b98ac000 rw-p  /data/app/sg.vp.owasp_mobile.omtg_android-.../lib
/arm64/libfrida-gadget.so
```

看看 Frida 留下的这两条痕迹，你可能已经想象到检测这些痕迹将是一项微不足道的任务。事实上，绕过这种检测是非常简单的。但事情可能会变得复杂得多。下表简要介绍了一些典型的 Frida 检测方法，并简要讨论了其有效性。

> Berdhard Mueller 的文章“检测 Frida 的柔术”中介绍了以下一些检测方法。有关更多详细信息和示例代码段，请参阅它。

方法	描述	讨论
检查 APP 签名	为了在 APK 中内置 frida-gadget，需要对其进行重新包装和重新签名。您可以在应用程序启动时检查 APK 的签名（例如，从 API 级别 28 开始的 GET_SIGNING_CERTIFICATES），并将其与您在 APK 中固定的签名进行比较。	不幸的是，这太容易绕过，例如通过修补 APK 或执行系统调用劫持。
检查环境中的相关工件	工件可以是包文件、二进制文件、库、进程和临时文件。对于 Frida，这可能是在目标（已经 root）系统（负责通过 TCP 公开 Frida 的守护进程）中运行的 frida-server。检查正在运行的服务（getRunningServices）和进程（ps），	自 Android 7.0（API 级别 24）以来，检查运行的服务/进程不会向您显示像 frida-server 那样的守护进程，因为它不是由应用程序本身启动的。即使有可能，只要重命名相应的 Frida 工件（frida-

	搜索名称为“frida-server”的服务。您还可以浏览已加载库的列表并检查可疑库（例如，名称中包含“frida”的库）	server/frida-gadget/frida-agent）就可以轻松绕过这一点。
检查开放的 TCP 端口	默认情况下，frida-server 进程绑定到 TCP 端口 27042。检查此端口是否打开是检测守护进程的另一种方法。	该方法在检测默认模式下的 frida-server，但侦听端口可以通过命令行参数更改，因此绕过它有点太简单了。
检查响应 D-Bus AUTH 的端口	frida-server 使用 D-Bus 协议进行通信，因此您可以预期它会响应 D-Bus AUTH。向每个打开的端口发送一条 D-Bus AUTH 消息，并检查响应，希望 frida-server 能够暴露自己。	这是一种相当稳健的检测 frida-server 的方法，但 frida 提供了不需要 frida-server 的其他操作模式。
扫描进程内存中的已知工件	扫描内存以查找 Frida 库中的工件，例如所有版本的 frida-gadget 和 Frida-agent 中都存在的字符串“LIBFRIDA”。例如，使用 Runtime.getRuntime().exec 并遍历 /proc/self/maps 或/proc/<pid>/maps（取决于 Android 版本）中列出的内存映射，搜索字符串。	这种方法更有效，并且仅使用 Frida 很难绕过，尤其是在添加了一些混淆处理以及正在扫描多个工件的情况下。然而，选定的工件可能会在 Frida 二进制文件中进行修补。在 Berdhard Mueller 的 GitHub 上找到源代码。

请记住，这张表远非详尽无遗。我们可以开始讨论命名管道（frida-server 用其进行外部通信），检测 trampolines（在函数的序言中插入的间接跳转向量），这将有助于检测 Substrate 或 frida 的拦截器，但例如，对 frida 的 Stalker 无效；以及许多其他或多或少有效的检测方法。它们中的每一个都取决于您是否正在使用已 root 设备、特定版本的 root 方法和/或工具本身的版本。此外，该应用程序可以尝试使用各种混淆技术，使其更难检测已实现的保护机制，如下文“测试逆向工程对抗弹性”一节所述。最后，这是保护在不受信任的环境（在用户设备上运行的应用程序）上处理的数据的猫捉老鼠游戏的一部分。

> 需要注意的是，这些控制只会增加逆向工程过程的复杂性。 如果使用，最好的方法是巧妙地组合控制措施而不是单独使用它们。 但是，它们都不能保证 100% 的有效性，因为逆向工程师将始终拥有对设备的完全访问权限，因此将永远获胜！ 您还必须考虑将一些控制措施集成到您的应用程序中可能会增加您的应用程序的复杂性，甚至对其性能产生影响。

5.10.4.3. 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：Frida, Xposed, Substrate for Android, 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。例如：

- 提醒用户并要求承担责任。
- 通过优雅终止来防止执行。
- 安全擦除存储在设备上的任何敏感数据。
- 向后端服务器报告，例如用于欺诈检测。

接下来，继续绕过逆向工程工具的检测，并回答以下问题：

- 是否可以轻易地绕过检查机制（例如：通过劫持单个 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？

当绕过逆向工程工具的检测时，应遵循以下步骤：

1. 修改防逆向工程功能。只需使用 NOP 指令覆盖相关的字节码或原生代码，即可禁用不想要的行为。
2. 使用 Frida 或 Xposed 劫持 Java 和原生层上的文件系统 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。当进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。

补丁，代码注入和内核模块的示例，请参见“Android 上的篡改和逆向工程”部分。

5.10.5. 测试模拟器检测(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模拟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逆向工程师需要对抗模拟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. 模拟器检测实例

有几个指示器表明存疑的设备是模拟器。尽管可以劫持所有这些 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
```

您可以在已经 root 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，劫持框架（例如 Xposed 或 Frida）可以劫持此 API 以提供虚假数据。

5.10.5.3. 绕过模拟器检测

1. 修补模拟器检测功能。只需使用 NOP 指令覆盖相关的字节码或原生代码，即可禁用不想要的行为。
2. 使用 Frida 或 Xposed API 劫持 Java 和原生层上的文件系统 API。返回看似无辜的值（最好从真实设备中获取），而不是返回模拟器值。例如：您可以重写 **`TelephonyManager.getDeviceID`** 方法以返回 IMEI 值。

有关补丁，代码注入和内核模块的示例，请参见“Android 上的篡改和逆向工程”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模拟器检测代码有多困难？
- 是否可以轻易地绕过检测机制（例如：通过劫持单个 API 函数）？
- 是否需要编写自定义代码来禁用反模拟器功能？ 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

此类别中的控制验证应用程序内存空间的完整性，以保护应用程序不能在运行时应用内存补丁。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内存中。完整性可以通过以下方式验证:

1. 比较内存的内容或内容的校验和与正常值。
2. 在内存中查找不想要的修改的特征。

这和“检测逆向工程工具和框架”类别有一些重叠，实际上，当我们展示了如何搜索进程内存中与 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控的更多示例。

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 博客

```
try {
  throw new Exception();
}
catch(Exception e) {
  int zygoteInitCallCount = 0;
  for(StackTraceElement stackTraceElement : e.getStackTrace()) {
    if(stackTraceElement.getClassName().equals("com.android.internal.os.Zygot
eInit")) {
      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 u
sing Substrate.");
    }
```

```
    if(stackTraceElement.getClassName().equals("de.robv.android.xposed.Xposed
Bridge") &&
        stackTraceElement.getMethodName().equals("main")) {
      Log.wtf("HookDetection", "Xposed is active on the device.");
    }
    if(stackTraceElement.getClassName().equals("de.robv.android.xposed.Xposed
Bridge") &&
        stackTraceElement.getMethodName().equals("handleHookedMethod")) {
      Log.wtf("HookDetection", "A method on the stack trace has been hooked u
sing Xposed.");
    }

  }
}
```

5.10.6.1.1.2. 检测原生劫持

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 劫持）或修补功能代码本身的某些部分（内联劫持）来安装原生函数劫持。检查各个存储区域的完整性是检测此类劫持的一种方法。

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 劫持将覆盖存储的函数地址，并将合法的函数调用重定向到攻击者控制的代码。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劫持。

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 劫持检测方法通常会遍历 GOT 并进行验证。

5.10.6.2.有效性评估

确保禁用所有基于文件的逆向工程工具检测。然后，使用 Xposed，Frida 和 Substrate 注入代码，并尝试安装原生劫持和 Java 方法劫持。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：

1. 修补完整性检查。通过用 NOP 指令覆盖相应的字节代码或原生代码来禁用不想要行为。
2. 使用 Frida 或 Xposed 劫持用于检测的 API 并返回假值。

有关补丁，代码注入和内核模块的示例，请参见“Android 上的篡改和逆向工程”部分。

5.10.7. 测试混淆(MSTG-RESILIENCE-9)

5.10.7.1. 概述

"移动应用篡改和逆向工程 "一章介绍了几种一般可用于移动应用的著名的混淆技术。

Android 应用程序可以使用不同的工具来实现其中的一些混淆技术。例如，ProGuard 提供了一种简单的方法来缩减和混淆代码，并从 Android Java 应用程序的字节码中剥离不需要的调试信息。它将类名、方法名和变量名等标识符替换为无意义的字符串。这是一种布局混淆，并不影响程序的性能。

> 反编译 Java 类是很简单的，因此建议总是对生产的字节码应用一些基本的混淆处理。

了解更多关于 Android 混淆技术：

- "Android 原生代码的安全加固" 作者：Gautam Arvind
- "APKiD：快速识别 AppShielding 产品" 作者：Eduardo Novella
- "原生 Android 应用的挑战：混淆和漏洞"，作者：Pierre Graux

5.10.7.1.1. 使用 ProGuard

开发人员使用 build.gradle 文件来启用混淆功能。在下面的例子中，你可以看到设置了 `minifyEnabled` 和 `proguardFiles`。创建例外来保护某些类不被混淆（使用-keepclassmembers 和-keep class）是很常见的。因此，审核 ProGuard 的配置文件以了解哪些类被豁免是很重要的。`getDefaultProguardFile('proguard-android.txt')`方法从 <Android SDK>/tools/proguard/文件夹获取默认的 ProGuard 设置。

关于如何缩小、混淆和优化你的应用程序的进一步信息可以在 Android 开发者文档中找到。

> 当你使用 Android Studio 3.4 或 Android Gradle 插件 3.4.0 或更高版本构建你的项目时，该插件不再使用 ProGuard 来执行编译时代码优化。作为替代，该插件使用 R8 编译器。R8 可以与你现有的所有 ProGuard 规则文件一起工作，所以更新 Android Gradle 插件以使用 R8 应该不需要你改变你现有的规则。

R8 是 Google 推出的新的代码缩减器，在 Android Studio 3.3 测试版中被引入。默认情况下，R8 会删除对调试有用的属性，包括行号、源文件名和变量名。R8 是一个免费的 Java 类文件收缩器、优化器、混淆器和预验证器，比 ProGuard 更快，更多细节请参见 Android 开发者博客文章。它集成在 Android 的 SDK 工具。要激活发布版本的收缩功能，请在 build.gradle 中添加以下内容：

```
android {
   buildTypes {
      release {
      // 只为你项目的发布构建类型启用代码缩减、混淆和优化。
      minifyEnabled true

      // 包括与 Android Gradle 插件一起打包的默认 ProGuard 规则文件
      // 要了解更多信息，请访问关于 R8 配置文件的部分。
      proguardFiles getDefaultProguardFile(
          'proguard-android-optimize.txt'),
          'proguard-rules.pro'
    }
  }
  ...
}
```

在 proguard-rules.pro 这个文件中，你可以定义自定义的 ProGuard 规则。使用参数-keep，你可以保留某些代码不被 R8 删除，否则会产生错误。例如，在我们的样本配置 proguard-rules.pro 文件中，就可以保留常见的 Android 类。

```
...
-keep public class * extends android.app.Activity
-keep public class * extends android.app.Application
-keep public class * extends android.app.Service
...
```

你可以通过以下语法对项目中的特定类或库进行更细化的定义：

```
-keep public class MyClass
```

混淆通常以运行时性能为代价，因此它通常只应用于代码中某些非常特殊的部分，通常是那些涉及安全和运行时保护的部分。

5.10.7.2. 静态分析

对 APK 进行反编译，并审查它以确定代码库是否被混淆了。下面你可以找到一个被混淆的代码块的样本：

```
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;
 }
}
```

考虑以下事项：

- 有意义的标识符，如类名、方法名和变量名，可能已被丢弃。
- 字符串资源和二进制文件中的字符串可能已加密。
- 与受保护功能相关的代码和数据可能被加密、打包或以其他方式隐藏。

对于原生代码：

- libc API（例如打开、读取）可能已经被操作系统系统调用所取代。
- Obfuscator-LLVM 可能被应用来提供“控制流扁平化”或“虚假控制流”

其中一些技术在 Gautam Arvind 的博文 "Android 原生代码的安全加固 "和 Eduardo Novella 的 "APKiD：AppShielding 产品的快速识别"演讲中进行了讨论和分析。

为了进行更详细的评估，你需要详细了解相关的威胁和使用的混淆方法。像 APKiD 这样的工具可以为你提供额外的指示，说明目标应用程序使用了哪些技术，如混淆器、打包器和反调试措施。

5.10.7.3. 动态分析

你可以使用 APKiD 来检测应用程序是否被混淆了。

使用适用于 Android 的 UnCrackable 应用程序级别 4 作为示例：

```
apkid owasp-mastg/Crackmes/Android/Level_04/r2pay-v1.0.apk
[+] APKiD 2.1.2 :: from RedNaga :: rednaga.io
[*] owasp-mastg/Crackmes/Android/Level_04/r2pay-v1.0.apk!classes.dex
 |-> anti_vm : Build.TAGS check, possible ro.secure check
 |-> compiler : r8
 |-> obfuscator : unreadable field names, unreadable method names
```

在这种情况下，它检测到应用程序有不可读的字段名和方法名，以及其他信息。

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 B 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认证的凭据。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 Keystore 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 Keystore 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_SHA512)
                .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_RSA_OAEP)
                .build());
KeyPair keyPair = keyPairGenerator.generateKeyPair();
Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
...

// 也可以随时从 Android Keystore 获取密钥对，如下所示:
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：

```
o //来源: <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();
```

```
// 也可以随时从 Android Keystore 获取密钥对，如下所示:
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
key = (SecretKey) keyStore.getKey("key2", null);
```

- 通过 AES-GCM 密码使用密钥对应用程序存储的身份认证数据和其他敏感数据进行加密，并使用实例 ID 等设备特定参数作为关联数据

```
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);

//使用密码来加密身份认证数据 有关更多详细信息，请参见 0x50e.
```

- 使用存储在 Android Keystore 中的公钥对密钥进行加密，并将加密的密钥存储在应用程序的私有存储中。
- 每当需要身份认证数据（例如访问令牌或其他敏感数据）时，请使用存储在 Android Keystore 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认证（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。随着 Android 8.0（API 级别 26）的发布，这种情况发生了变化。 由于 MAC 地址现在通常在未连接到接入点时是随机的，并且 SSAID 不再是设备绑定 ID。 相反，它成为绑定到用户、设备和请求 SSAID 的应用程序的应用签名密钥的值。 此外，Google 的 SDK 文档中还有关于标识符的新建议。 基本上，Google 建议：

- 在广告方面使用广告 ID (**`AdvertisingIdClient.Info`**) - 以便用户可以选择拒绝。
- 使用实例 ID (**`FirebaseInstanceId`**) 进行设备识别。
- 仅将 SSAID 用于欺诈检测和由同一开发人员签名的应用程序之间共享状态。

请注意，实例 ID 和广告 ID 在设备升级和设备重置之间不稳定。 但是，实例 ID 至少可以识别设备上当前的软件安装。

当源代码可用时，您可以查找一些关键术语：

- 不再起作用的唯一标识符：
  - 没有 **Build.getSerial** 的 **Build.SERIAL**
  - HTC 设备的 **htc.camera.sensor.front_SN**
  - **persist.service.bdroid.bdadd**
  - 来自 **WifiManager** 的 **Settings.Secure.bluetooth_address** 或 **WifiInfo.getMacAddress**，除非 manifest 中启用了系统权限 **LOCAL_MAC_ADDRESS**。
- **ANDROID_ID** 仅用作标识符。 随着时间的推移，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 Android KeyStore 中创建私钥。

要确保可以使用标识符，请检查 **AndroidManifest.xml** 以了解 IMEI 和 **Build.Serial** 的使用情况。 该文件应包含权限 **<uses-permission android:name="android.permission.READ_PHONE_STATE" />**。

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. 使用模拟器进行动态分析

1. 在模拟器中运行应用程序
2. 确保您可以提高应用程序实例的信任度（例如：在应用程序中进行身份认证）。
3. 按照以下步骤从模拟器中检索数据：
   - 通过 ADB Shell 使用 SSH 进入模拟器。
   - 执行 **run-as <您的 app-id>**。您的 app-id 是 在 AndroidManifest.xml 中描述的。

- **chmod 777** 缓存和共享首选项的内容。
- 从 app-id 退出当前用户。
- 将**/data/data/<your appid>/cache** 的内容和 **shared-preferences** 复制到 SD 卡。
- 使用 ADB 或 DDMS 提取内容。

4. 在另一个模拟器上安装该应用程序。
5. 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模拟器的 SD 卡中。
   - 通过 ADB Shell 使用 SSH 进入模拟器。
   - 执行 **run-as <您的 app-id>**。您的 app-id 是 在 AndroidManifest.xml 中描述的。
   - **chmod 777** 缓存和共享首选项的内容。
   - 将 SD 卡的旧内容复制到**/data/data/<your appid>/cache** 和 **shared-preferences**。
6. 您可以继续通过身份认证吗？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认证。一旦应用程序重置，卸载等，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：

1. 在 Google Developer Console 中为给定应用程序配置实例 ID。这包括管理 PROJECT_ID。
2. 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'
    }
```

3. 获取实例 ID

```
String iid = Instance ID.getInstance(context).getId();
//现在提交这个 iid 到你的服务器.
```

4. 生成令牌

```
String authorizedEntity = PROJECT_ID; //来自 Google Developer Console 的
Project id
String scope = "GCM"; // 例如使用 GCM 通信，但你可以使用任何
                      // 最多 1000 个 URL 安全字符，或者
                      // 也可以保持为空.
String token = Instance ID.getInstance(context).getToken(authorizedEnti
ty,scope);
//现在提交这个令牌到服务器.
```

5. 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 IDListenerServic**e 并在那里处理回调：

```
public class MyInstance IDService extends Instance IDListenerService {
public void onTokenRefresh() {
    refreshAllTokens();
}

private void refreshAllTokens() {
    // 假设你已经将 TokenList 定义为不同作用域的令牌的一些通用存储。
    // 请注意，对于应用验证来说，只有一个范围内的一个令牌就足够了。
    ArrayList<TokenList> tokenList = TokensList.get();
    Instance ID iid = Instance ID.getInstance(this);
    for(tokenItem : tokenList) {
    tokenItem.token =
        iid.getToken(tokenItem.authorizedEntity,tokenItem.scope,tokenIt
em.options);
    // 将这个 tokenItem.token 发送到你的服务器上
    }
}
};
```

6. 在您的 Android manifest 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 云服务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 也支持实例 ID。

5.10.8.3.3. IMEI 和序列号

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 8.0（API 级别 26）之前的 Android 设备，您可以按如下方式请求序列号：

```
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：

1. 在您的 Android manifest 中设置权限：
   ```
   <uses-permission
   android:name="android.permission.READ_PHONE_STATE" />
   <uses-permission
   android:name="android.permission.ACCESS_NETWORK_STATE" />
   ```
2. 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
3. 获取序列号：
   ```
   String serial = android.os.Build.getSerial();
   ```

检索 IMEI：

1. 在您的 Android manifest 中设置所需的权限：
   ```
   <uses-permission android:name="android.permission.READ_PHONE_STATE" />
   ```
2. 如果您使用的是 Android 版本 Android 6（API 级别 23）或更高版本，请在运行时向用户请求权限：有关详细信息，请参阅 https://developer.android.com/training/permissions/requesting.html。
3. 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 8.0（API 级别 26）以来，SSAID 和 MAC 地址的行为发生了变化。 此外，Google 的 SDK 文档中还有关于标识符的新建议。由于这种新行为，我们建议开发人员不要单

独依赖 SSAID。标识符变得不太稳定。 例如，在恢复出厂设置或升级到 Android 8.0（API 级别 26）后重新安装应用时，SSAID 可能会发生变化。 有些设备具有相同的 ANDROID_ID 和/或具有可以覆盖的 ANDROID_ID。 因此，最好使用 AES_GCM 加密使用 AndroidKeyStore 中随机生成的密钥对 ANDROID_ID 进行加密。 然后应将加密的 ANDROID_ID 存储在 SharedPreferences 中（私有）。 应用签名更改的那一刻，应用程序可以检查增量并注册新的 ANDROID_ID。 在没有新的应用程序签名密钥的情况下发生这种变化的那一刻，它应该表明还有其他问题。

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：
  - 没有 **`Build.getSerial`** 的 **`Build.SERIAL`**
  - HTC 设备的 **`htc.camera.sensor.front_SN`**
  - **`persist.service.bdroid.bdadd`**
  - 来自 **`WifiManager`** 的 **`Settings.Secure.bluetooth_address`** 或 **`WifiInfo.getMacAddress`**，除非 manifest 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(Contex
t.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。manifest 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. 使用模拟器

请参见上面的“使用模拟器进行动态分析”部分。

5.10.8.4.2. 使用两台不同已经 root 设备

1. 在已 root 设备上运行该应用程序。

2. 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认证）。
3. 从第一台已 root 的设备检索数据。
4. 在第二个已 root 设备上安装应用程序。
5. 在应用程序的数据文件夹中，覆盖步骤 3 中的数据。
6. 您可以继续通过身份认证吗？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. 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。”

5.10.9.2. SafetyNet 认证

- 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-thingsyou-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/androiddeveloper/contact/safetynetqr

6. iOS 移动安全测试

6.1. iOS 平台概述

iOS 是为 Apple 移动设备（包括 iPhone，iPad 和 iPod Touch）设计的移动操作系统。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 Not 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。Apple 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应用 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。

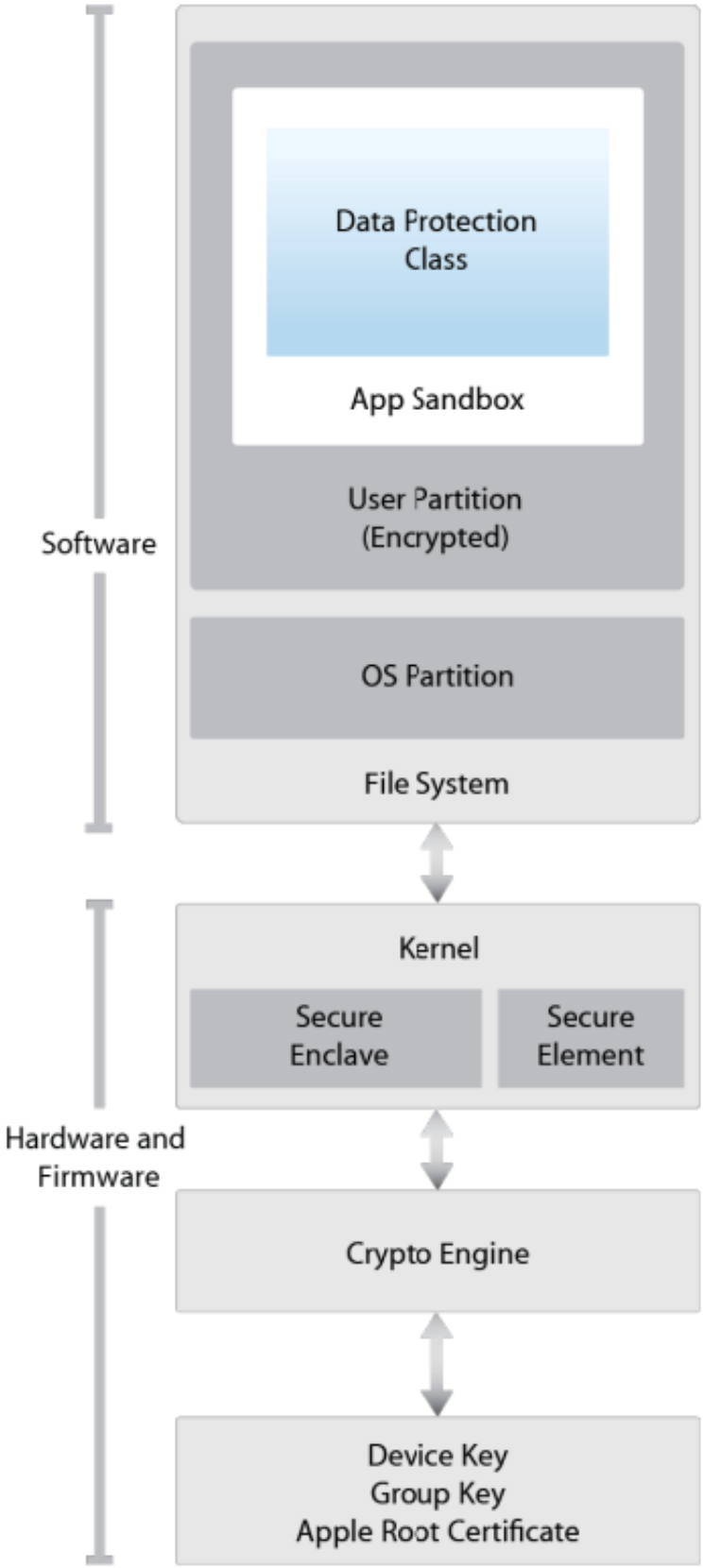
iOS 应用程序通过 Apple 的 iOS 沙箱（历史上称为 Seatbelt）彼此隔离，这是一种强制性访问控制（MAC）机制，用于描述应用程序可以访问和不能访问的资源。与 Android 广泛的活页夹 IPC 设施相比，iOS 提供的 IPC（进程间通信）选项很少，从而最大程度地减少了潜在的攻击面。硬件或软件的高度集成创建了另一个安全优势。每个 iOS 设备都能够提供安全功能，例如安全启动，硬件支持的 Keychain 和文件系统加密（在 iOS 中称为数据保护）。通常情况下，iOS 更新通常会迅速推广到很大一部分用户，从而减少了支持较旧的不受保护的 iOS 版本的需求。

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，Keychain， Touch ID / Face ID 身份认证以及网络安全性仍然存在较大的安全风险。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。
- 通用漏洞缓解。


Data Protection Class
App Sandbox
User Partition (Encrypted)
Software
OS Partition
File System
Kernel
Secure Enclave
Secure Element
Hardware and Firmware
Crypto Engine
Device Key
Group Key
Apple Root Certificate

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 在生产过程中被蚀刻到硅芯片中，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还原屏幕。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。

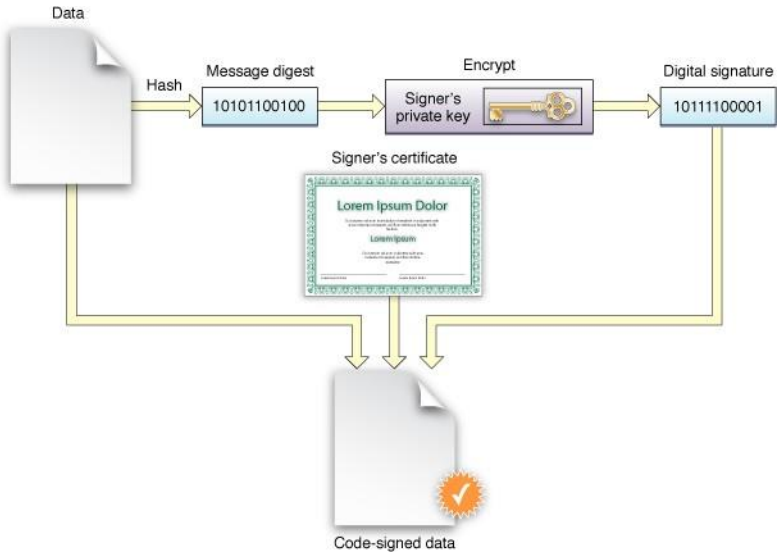
这整个过程称为“安全启动链”。其目的专注于验证引导过程的完整性，确保系统及其组件由 Apple 编写和分发。安全启动链由内核，引导加载程序，内核扩展和基带固件组成。

6.1.1.3. 代码签名

Apple 已经实施了精心设计的 DRM 机制，以确保只有经过 Apple 批准的代码才能在其设备上运行，即由 Apple 签名的代码。换句话说，除非 Apple 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 AppleApp 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）。



根据存档的 Apple 开发者文档，代码签名由三部分组成。

- 一个印章。这是由代码签名软件创建的代码各部分的校验和或散列的集合。印章可以在验证时用来检测改动。
- 一个数字签名。代码签名软件使用签名者的身份对印章进行加密，以创建一个数字签名。这保证了印章的完整性。
- 代码要求。这些是管理代码签名验证的规则。根据目标的不同，有些是验证器所固有的，而有些则是由签名者指定的，并与代码的其他部分一起密封。

了解更多：

- 代码签名指南（存档的 Apple 开发者文档）
- 代码签名（Apple 开发者文档）
- 揭开 iOS 代码签名的神秘面纱

6.1.1.4. 加密和数据保护

FairPlay 代码加密应用于从 App Store 下载的应用程序。FairPlay 被开发为一种 DRM，用于购买的多媒体内容。最初，Fairplay 加密被应用于 MPEG 和 QuickTime 流，但是相同的方法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，启用数据保护是用户在设备上使用密码的主要原因。

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盗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，将应用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执行，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. 通用漏洞缓解

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。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，运行和测试原生 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，它将 Smalltalk 风格的消息传递添加到 C 编程语言中。它在 macOS 上用于开发桌面应用程序，在 iOS 上用于开发移动应用程序。Swift 是 Objective-C 的继任者，允许与 Objective-C 进行互操作。

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：

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 应用程序包装在 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。下面的示例从较高的层次展示了这种结构：

- **`/Payload/`**文件夹包含所有应用程序数据。 我们稍候将更详细地介绍文件夹里内容。
- **`/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，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`** 是扩展包的示例。 该特定的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。

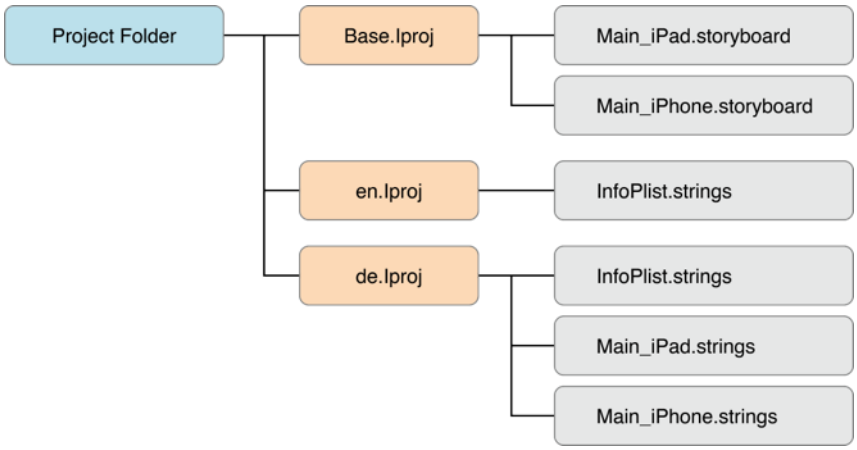
6.1.3.1. IPA 文件的内容-更深入的理解

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。顶层包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镜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层目录中，本地化的资源放置在应用程序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。它包含一个故事板和字符串文件。

- 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，您可以使用不同的工具恢复已安装 iOS 应用程序的 IPA，这些工具允许解密主应用程序二进制文件并重建 IPA 文件。类似地，在越狱设备上，可以使用 IPA 安装程序安装 IPA 文件。在移动安全评估期间，开发者通常会直接向您提供 IPA 文件。他们可以向您发送实际文件或提供对他们使用的特定开发分发平台的访问，例如 TestFlight 或 Visual Studio 应用 Center。

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
- 备忘录
- 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：

- 验证通过 IPC 通信或 URL 方案所有输入，另请参阅：
  - 测试自定义 URL 方案。
- 验证用户在输入字段中的所有输入。
- 验证 WebView 机制加载的内容，另请参见：
  - 测试 iOS WebViews。
  - 检查原生方法是否通过 WebView 暴露。
- 与后端服务器安全通信，或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：
  - 测试网络通信
  - iOS 网络通信
- 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：
  - iOS 上数据存储
- 保护应用程序不受不安全的环境中、重新打包或其他本地攻击的影响，另请参阅：
  - iOS 反逆向防御。

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 应用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

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）。

6.2.1.1. 主机设备

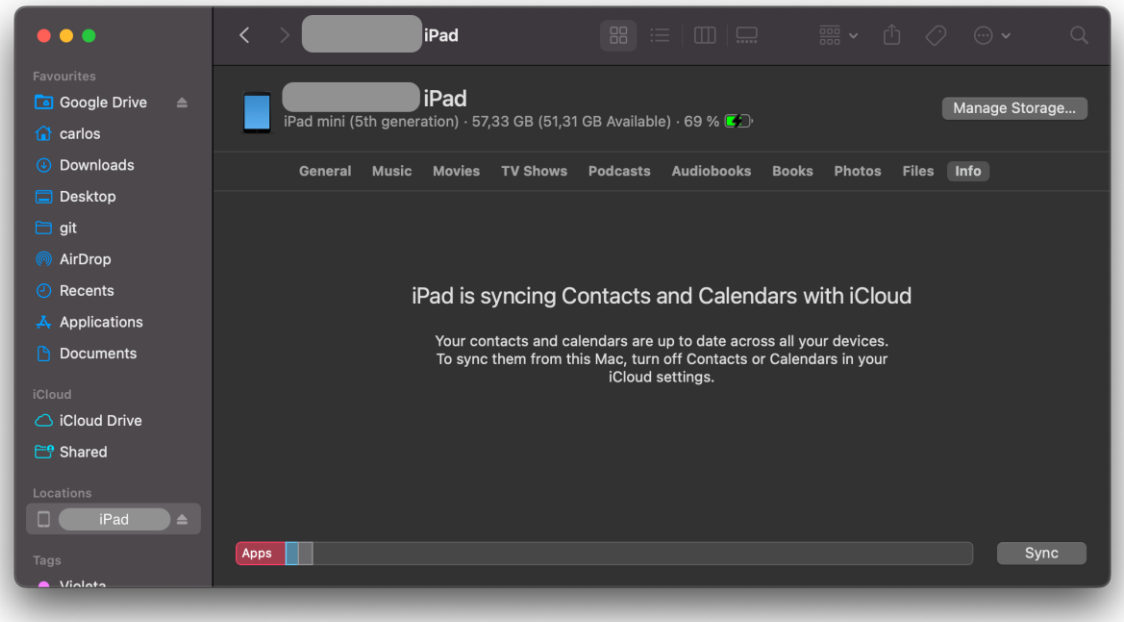
以下是最基本的 iOS 应用测试设置：

- 最好是具有管理员权限的 macOS 机器。
- 已安装 Xcode 和 Xcode 命令行程序。
- 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（需要测试的 iOS 版本）。
- BurpSuite 或其他拦截代理工具。

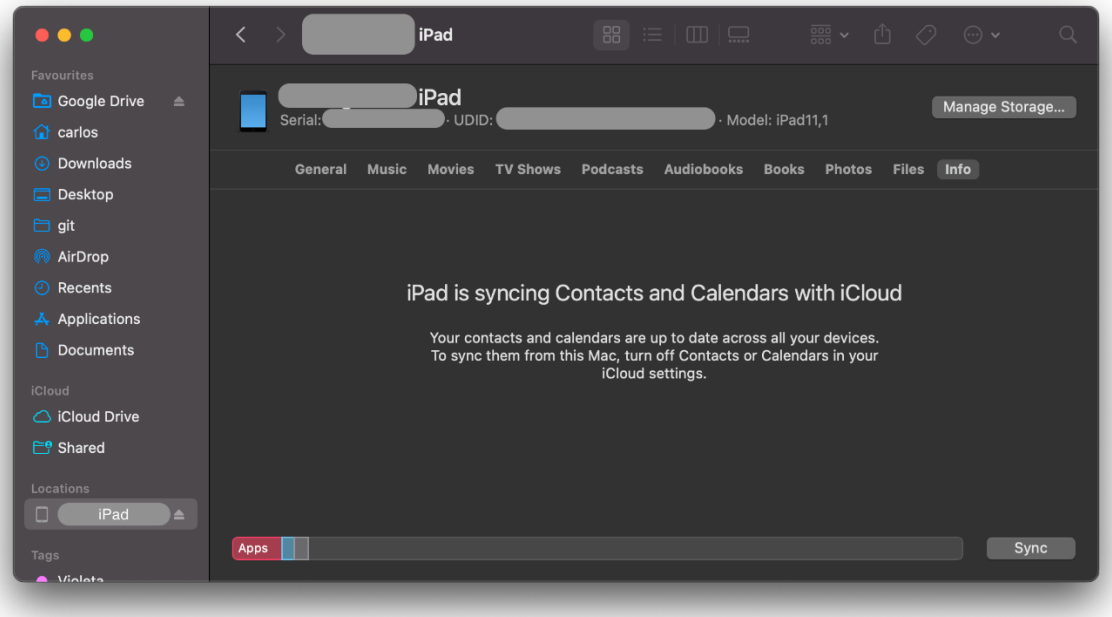
6.2.1.2. 测试设备

6.2.1.2.1. 获取 iOS 设备的 UDID

UDID 是一个由字母和数字组成的 40 位唯一序列，用于识别 iOS 设备。你可以在 macOS Catalina 上的 Finder 应用中找到你的 iOS 设备的 UDID，因为 iTunes 在 Catalina 中已经不可用。打开 Finder，在侧边栏选择连接的 iOS 设备。



点击包含型号、存储容量和电池信息的文本，它将换成显示序列号、UDID 和型号。



你可以通过在 UDID 上点击右键来复制它。

当设备通过 USB 连接时，也可以通过 MacOS 上的各种命令行工具获得 UDID：

- 通过使用 I/O 注册表浏览器工具 ioreg：

```
$ ioreg -p IOUSB -l | grep "USB Serial"
|   "USB Serial Number" = "9e8ada44246cee813e2f8c1407520bf2f84849ec"
```

- 通过使用 ideviceinstaller（在 Linux 上也可用）：

```
$ brew install ideviceinstaller
$ idevice_id -l
316f01bd160932d2bf2f95f1f142bc29b1c62dbc
```

- 通过使用 system_profiler：

```
$ system_profiler SPUSBDataType | sed -n -e '/iPad/,/Serial/p;/iP
hone/,/Serial/p;/iPod/,/Serial/p' | grep "Serial Number:"
2019-09-08 10:18:03.920 system_profiler[13251:1050356] SPUSBDevic
e: IOCreatePlugInInterfaceForService failed 0xe00002be
      Serial Number: 64655621de6ef5e56a874d63f1e1bdd14f7103b1
```

- 通过使用 instruments:

```
instruments -s devices
```

6.2.1.2.2. 在真实设备上测试（已越狱）

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应用是无法运行的，这使得模拟器无法用于黑盒分析和逆向工程。

6.2.1.2.4. 在仿真器上进行测试

Corellium 是唯一公开的 iOS 仿真器。它是一个企业 SaaS 解决方案，采用每用户许可模式，不提供社区许可。

6.2.1.2.5. 获取特殊权限

iOS 越狱通常被比作 Android 的 root，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“刷机”的概念。

- **Root**：这通常涉及在系统上安装 su 二进制文件或使用 已经 root 的自定义 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。
- **刷入自定义 ROM**：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，刷入自定义 ROM 是不可能的，因为 iOS 引导加载程序只允许引导和刷入 Apple 签名的镜像。这就是为什么即使是官方的 iOS 镜像但如果没有 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（例如，自定义代码或从 Cydia 或 Sileo 等替代应用商店下载应用）。“越狱”这个词是一个通俗的说法，指的是使自动化禁用过程的一体化工具。

开发给定版本的 iOS 越狱并不容易。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 版本所用到的漏洞，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）和一个照片应用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 “测试 iOS 上的反逆向防御” 章节中介绍它们。

6.2.1.2.5.1. 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应用。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 权限访问文件系统。
- 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 运行时。

6.2.1.2.5.2. 越狱类型

有非完美越狱、半非完美越狱、半完美越狱和完美越狱。

- 连接越狱（非完美越狱）无法在重新启动后保持，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。
- 半连接越狱（半非完美越狱）无法重新应用除非设备在重新启动期间连接到计算机。该设备还可以自行启动进入非越狱模式。
- 半无连接越狱（半完美越狱）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，因此称无连接）。
- 无连接越狱（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。

6.2.1.2.5.3 注意事项和考虑因素

随着 Apple 公司不断强化其操作系统，开发 iOS 的越狱变得越来越复杂。每当 Apple 意识到一个漏洞，它就会被修补，并向所有用户推送系统更新。由于不可能降级到特定的 iOS 版本，而且 Apple 只允许你更新到最新的 iOS 版本，因此，要想拥有一个运行着可越狱的 iOS 版本的设备，是一个挑战。有些漏洞是无法通过软件修补的，例如 checkm8 漏洞可以影响到 A12 之前所有 CPU 的 BootROM。

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），然后等待公开发布的越狱。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 Apple 仍然在签名的话）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（通常称为 SHSH blobs）。只有在 Apple 对请求回应进行签名后，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”，这也是为什么你不能简单地存储下载的 OTA 固件软件包，并随时将其加载到设备上的原因。在较小的 iOS 升级过程中，两个版本可能都由 Apple 签名（最新版本和以前的 iOS 版本）。这是唯一可以降级 iOS 设备的情况。您可以从 IPSW 下载网站查看当前的签名窗口以及下载 OTA 固件。

对于某些设备和 iOS 版本，如果该设备的 SHSH blobs 是在签名窗口激活时收集的，则有可能降级到旧版本。这方面的更多信息可以在 cfw iOS 指南 - 保存 blobs 中找到。

6.2.1.2.5.4. 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，它们的域名通常与越狱团体、作者的域名相似。

iOS 越狱场景发展如此之快，很难提供与时俱进的指南。不过，我们可以提供一些目前可靠的消息。

- AppleDB
- iPhone 知识库
- 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。

6.2.2. 基本测试操作

6.2.2.1. 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 Shell

与通过 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 shell。在本节中，假设已正确越狱，并安装好 Cydia（见下面的屏幕截图）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，然后点击一次所连接网络的信息图标找到该地址。

现在，您可以通过执行 **ssh root@<device_ip_address>**来远程访问设备的 shell，它将让你以 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 或输入 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.password**（若使用设备 shell，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**（这是密码 alpine 的哈希）。
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，可能没有可靠的 Wi-Fi 连接供使用。在这种情况下，可以使用 usbmuxd 通过 USB 连接到设备的 SSH 服务器。

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。如果不想每次通过 USB 进行 SSH 时都运行二进制文件，也可以让 iproxy 在后台自动运行。

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#
```

iDevice 的 USB 小提示：由于 iOS 11.4.1 引入的 USB 限制模式，在 iOS 设备上，处于锁定状态 1 小时后，您将无法再进行数据连接，除非您再次将其解锁。

6.2.2.1.2. 设备上的 shell 应用

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 iOS 6.0 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 SSH 连接。在这些情况下，拥有一个设备上的 shell 应用程序非常方便，首先可以使用它用建立一个反向 shell 通过设备的 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>`** 通过 SSH 打开反向 shell。

在设备 shell 应用程序上运行以下命令，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 设备或应用程序数据沙盒传输到主机，反之亦然。接下来将展示如何实现这一点的不同方法。

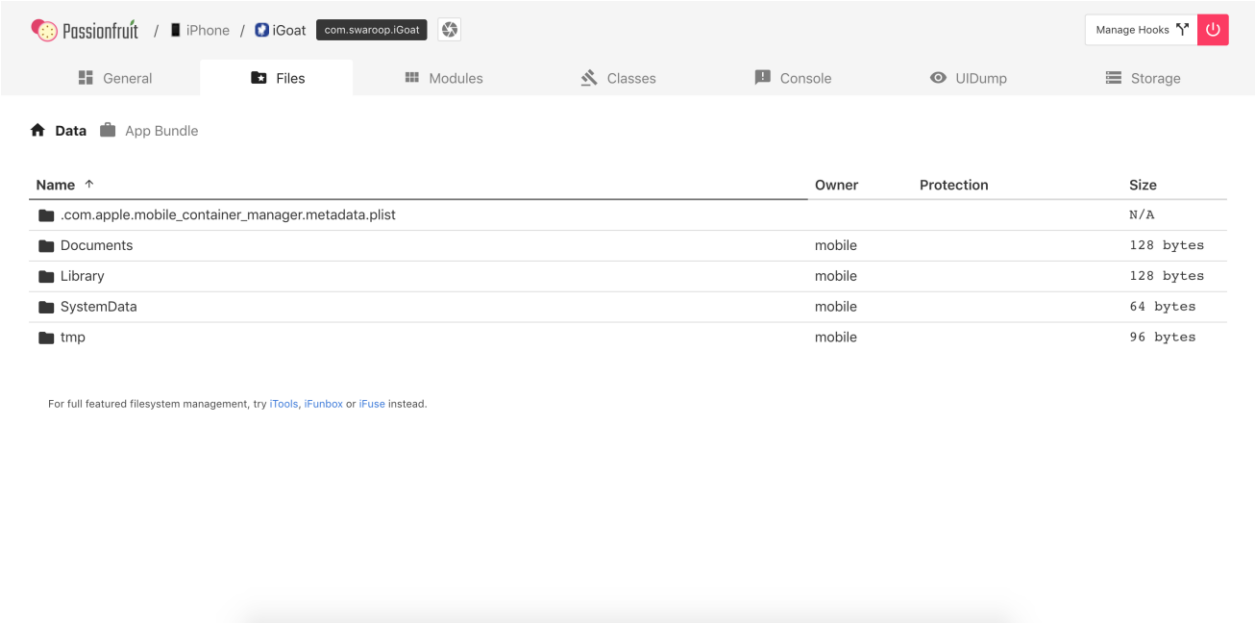
6.2.2.2.1. 通过 SSH 和 SCP 复制应用程序数据文件

众所周知，应用程序的文件存储在 Data 目录中。现在，您只需使用 tar 打包数据目录，并使用 scp 从设备中提取它：

```
iPhone:~ root# tar czvf /tmp/data.tgz /private/var/mobile/Containers/Data/App
lication/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693
iPhone:~ root# exit
$ scp -P 2222 root@localhost:/tmp/data.tgz .
```

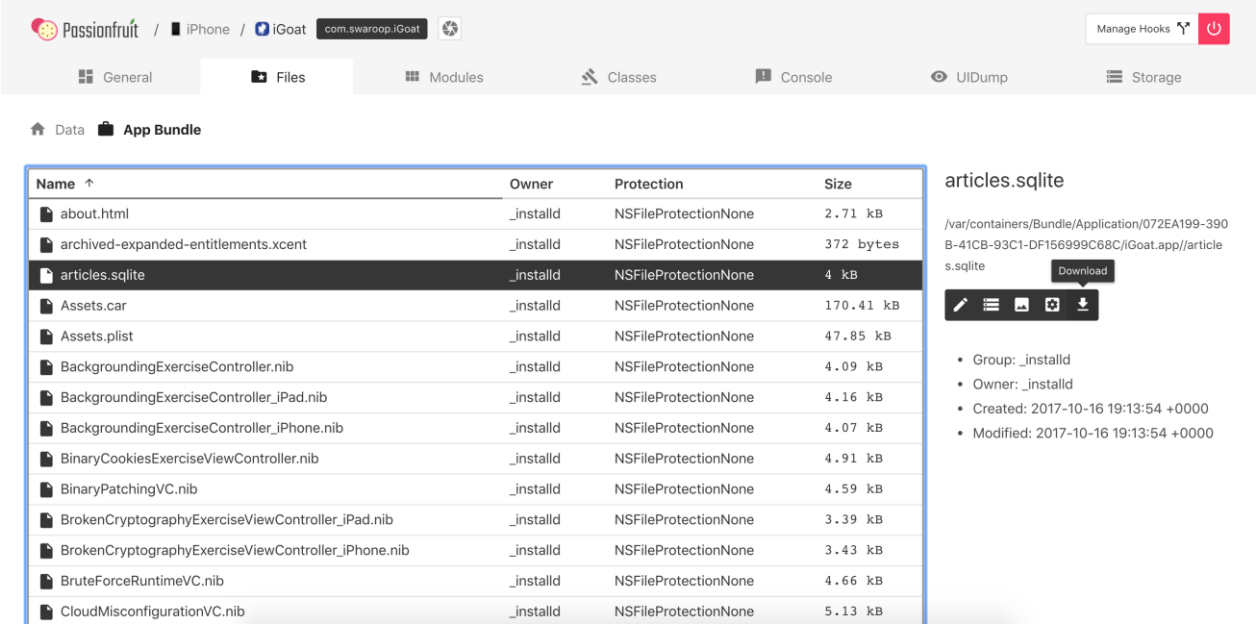
6.2.2.2.2. Passionfruit

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。



在目录中浏览并选择文件时，将显示一个弹出窗口，并将数据显示为十六进制或文本。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



6.2.2.2.3. Objection

当开启 Objection 时，提示符将在 Bundle 目录中。

```
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 命令获取应用程序的目录并导航到 Documents 目录。

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # cd /var/mobile/Containers/Data/App
lication/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 设备下载到主机，然后可以对其进行分析。

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # file download .com.apple.mobile_co
ntainer_manager.metadata.plist
Downloading /var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-D
8B4A2D44133/.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 <本地文件路径>`**命令将文件上载到 iOS 设备。

6.2.2.3. 获取和提取应用

6.2.2.3.1. 从 OTA 分发链接获取 IPA 文件

在开发过程中，有时会通过空中下载（OTA）将应用程序分发给测试人员。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 服务资产下载工具从 OTA 分发 URL 的下载 IPA。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应用程序二进制文件），将其解压就能使用了。应用程序二进制文件位于主目录（.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，应用程序目录可以通过右键单击并选择“Show Package Content”来打开。在终端上，您只需使用 cd 进入目录就行了。

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应用程序（例如通过应用商店）。安装后，需要从内存中提取应用程序二进制文件并重建 IPA 文件。由于使用了 DRM，文件在存储到 iOS 设备上时会被加密，因此简单地从软件包中提取二进制文件（通过 SSH 或 Objection）不足以对其进行逆向工程。。

下面显示了在 Telegram 应用程序上运行 class-dump 的输出，其直接从 iPhone 中的安装目录中获取：

```
$ class-dump Telegram
//
```

```
//     Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun  9 20
15 22:53:21).
//
//     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve N
ygard.
//

#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，您可以使用 frida-ios-dump（所有 iOS 版本）或 Clutch（仅限 iOS 11；对于 iOS 12 及以上版本，它需要一个补丁）等工具。当应用程序在设备上运行时，两者都将从内存中提取未加密的版本。Clutch 和 frida-ios-dump 的稳定性可能因您的 iOS 版本和越狱方法而异，因此有多种方法提取二进制文件很有用。

> 重要提示：在美国，《数字千年版权法案》17 U.S.C. 1201，即 DMCA，规定规避某些类型的 DRM 是非法的，并可采取行动。然而，DMCA 也提供了豁免，如某些类型的安全研究。一位合格的律师可以帮助你确定你的研究是否符合 DMCA 的豁免规定。(来源：Corellium)

6.2.2.3.2.1. 使用 Clutch

按照 Clutch GitHub 页面上的说明编译 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 applica
tions
Installed apps:
...
5:   Telegram Messenger <ph.telegra.Telegraph>
...
```

获得包标识符后，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 poss
ible 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 f
rom specified bundleID
...
Successfully dumped framework TelegramUI!
Zipping WebP.framework
Zipping NotificationContent.appex
Zipping NotificationService.appex
Zipping Share.appex
Zipping SiriIntents.appex
Zipping Widget.appex
DONE: /private/var/mobile/Documents/Dumped/ph.telegra.Telegraph-iOS9.0-(Clutc
h-(null)).ipa
Finished dumping ph.telegra.Telegraph in 20.5 seconds
```

将 IPA 文件复制到主机系统并解压缩后，您可以看到现在可以通过 class-dump 解析 Telegram 应用程序二进制文件，这表明它不再加密：：

```
$ class-dump Telegram
...
//
//     Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun  9 20
15 22:53:21).
//
//     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve N
ygard.
//

#pragma mark Blocks

typedef void (^CDUnknownBlockType)(void); // return type and parameters are u
nknown

#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. 使用 Frida-iOS-dump

首先，确保 Frida-ios-dump 中 **dump.py** 的配置，当使用 iproxy 时，设置为 localhost 的 2222 端口，或者设置为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，在您使用的 **dump.py** 中更改默认用户名（User = 'root'）和密码（Password = 'alpine'）。

现在，您可以安全地使用该工具枚举已安装的应用程序：

```
$ python 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
   ...
```

并且你可以 dump 列表中的二进制文件：

```
$ python dump.py ph.telegra.Telegraph

Start the target 应用 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"
```

在这之后是 **Telegram.ipa** 文件将在当前目录中创建。您可以通过删除应用程序并重新安装来验证转储是否成功（例如，使用 ios-deploy `ios-deploy -b Telegram.ipa`）。请注意，这只适用于越狱设备，否则签名将无效。

6.2.2.4. 重打包应用

如果你需要在非越狱的设备上进行测试，你应该学习如何重新打包一个应用程序，以便在上面进行动态测试。

使用一台装有 macOS 的电脑，执行 objection Wiki 中 "修补 iOS 应用程序"一文中的所有步骤。一旦你完成了，你就可以通过调用 objection 命令来修补 IPA。

```
objection patchipa --source my-app.ipa --codesign-signature
0C2E8200Dxxxx
```

最后，应用程序需要被安装（侧载），并在启用调试通信后运行。执行 objection 维基中 "运行修补的 iOS 应用程序 "一文中的步骤（使用 ios-deploy）。

```
ios-deploy --bundle Payload/my-app.app -W -d
```

参考 "安装应用程序"，了解其他安装方法。其中一些不需要你有 macOS。

> 这种重新打包的方法对于大多数使用情况来说是足够的。对于更高级的重新打包，请参考 "iOS 篡改和逆向工程—修补、重新打包和重新签名"。

6.2.2.5. 安装应用

当您不使用 Apple 的应用商店安装一个应用，这被称为侧载。下面会描述各种侧载方式。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，所有应用程序都必须使用 Apple 颁发的证书签名。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 是一项调整 installd 的补丁程序，允许安装假签名的 IPA 程序包。

在 iOS 设备上安装 IPA 软件包有不同的方法，下面将详细介绍。

请注意，iTunes 在 macOS Catalina 中不再可用。如果您使用的是旧版本的 macOS，iTunes 仍然可用，但从 iTunes 12.7 开始无法安装应用程序。

6.2.2.5.1. Cydia Impactor

Cydia Impactor 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。Cydia Impactor 可用于 Windows、macOS 和 Linux。yalujailbreak.net 上提供了分步指南和故障排除步骤。

6.2.2.5.2. 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。这让你可以通过执行 ideviceinstaller 使用 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 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
brew install ideviceinstaller
```

安装完成后，您可以使用几个新的命令行工具，如 ideviceinfo、ideviceinstaller 或 idevicedebug。

```
# 以下命令会显示通过USB 连接的iOS 设备详细信息.
$ ideviceinfo
# 以下命令会安装IPA 到你的iOS 设备.
$ ideviceinstaller -i iGoat-Swift_v1.0-frida-codesigned.ipa
...
Install: Complete
#以下命令会通过提供绑定名称在debug 模式下启动应用程序.绑定名称可以在前一个"Installing"命令之后找到.
$ idevicedebug -d run OWASP.iGoat-Swift
```

6.2.2.5.3. 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 ipainstaller 加上 IPA 的文件名执行安装：

```
ipainstaller App_name.ipa
```

6.2.2.5.4. iOS-deploy

在 macOS 上，您还可以使用 ios-deploy 工具从命令行安装 iOS 应用程序。由于 ios-deploy 使用包安装应用程序，您需要解压缩 IPA。

```
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 参数即可启动它，该参数将直接启动调试，而无需再次安装应用程序。

```
ios-deploy --bundle 'Payload/Name.app' -W -d -v -m
```

6.2.2.5.5. Xcode

通过执行以下步骤，也可以使用 Xcode IDE 安装 iOS 应用程序：

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“Installed Apps”中的“+”。

6.2.2.5.6. 允许在非 iPad 设备上安装应用程序

有时应用程序可能需要在 iPad 设备上使用。如果你只有 iPhone 或 iPod touch 设备，那么你可以强制应用程序接受在这类设备上安装和使用。您可以通过在 Info.plist 文件中将属性 **UIDeviceFamily** 的值更改为 1 来实现。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DT
Ds/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。

如果应用程序需要现代 ipad 特有的功能，而您的 iPhone 或 iPod 稍旧，那么这种绕过可能不起作用。

可以在 Apple Developer 文档中找到属性 UIDeviceFamily 的可能值。

6.2.2.6. 信息收集

分析应用程序的一个基本步骤是收集信息。这可以通过检查主机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：获取所有已安装的应用程序列表、浏览应用程序包以及访问设备上的应用程序数据目录。这在不需要对应用程序进行逆向工程或执行更高级的分析的前提下给您一点关于应用程序是什么的背景知识。将回答以下问题：

- 软件包中包含哪些文件?
- 应用程序使用哪些框架?
- 应用程序需要哪些功能?
- 应用程序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?

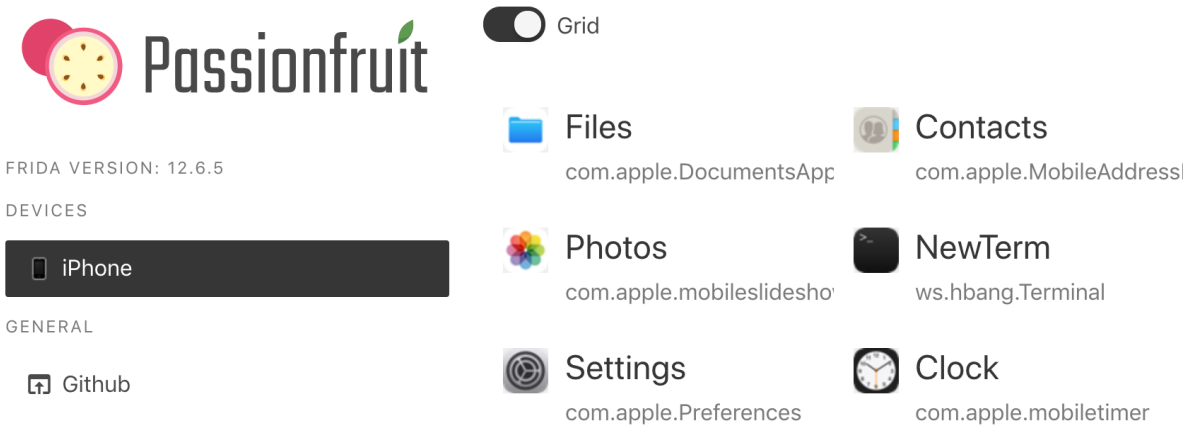
6.2.2.6.1. 列出已安装的应用程序

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标识符。可以使用 **frida-ps -Uai** 获取连接的 USB 设备(-U)上当前安装的(-i)所有应用程序(-a)：

```
$ 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“Identifier 标识符”（包标识符）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。



6.2.2.6.2. 研究应用程序包

收集了目标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，提取 IPA，如基本测试操作-获取和提取应用程序章节中所述。

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 `Payload` 文件夹包含所谓的应用程序包（.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，例如其包软件 ID，版本号和显示名称。
- **_CodeSignature/**包含一个 plist 文件，该文件对包软件中所有文件进行签名。
- **Frameworks/**包含应用程序原生库，例如.dylib 或.framework 文件。
- **PlugIns/**可能包含.appex 文件形式的应用程序扩展（示例中不存在）。
- **iGoat-Swift** 是包含应用程序代码的应用程序二进制文件。其名称与不含应用程序扩展名的包的名称相同。
- 各种资源，例如图像/图标, *.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。

6.2.2.6.2.1. 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来源。其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，所有包的可执行文件（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键。

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
```

下面是一个非详尽的列表，列出了一些信息和相应的关键词，你可以通过直接检索文件或使用 **`grep -i <keyword> Info.plist`** 来轻松搜索 **`Info.plist`** 文件。

- 应用程序权限目的字符串：**`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”）。
- 应用程序传输安全性（ATS）配置：**`NSAppTransportSecurity`**（请参阅“ iOS 网络 API”）。

请参考提到的章节，以了解有关如何测试这些要点的更多信息。

6.2.2.6.2.2. 应用程序二进制

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，您可以将应用二进制文件反编译成 Java 代码，而 iOS 应用程序二进制文件只能反汇编。

有关详细信息，请参阅“iOS 上的逆向工程和篡改”章节。

6.2.2.6.2.3. 原生库

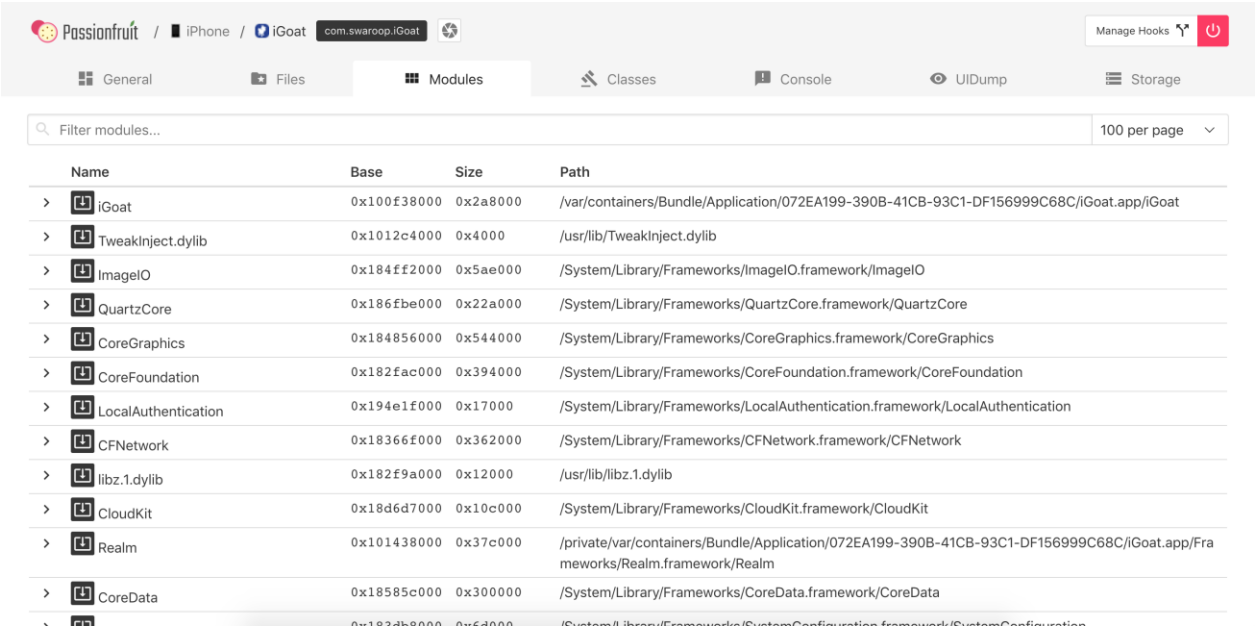
iOS 应用程序可以通过使用不同的元素使其代码库模块化。在 MASTG 中，我们将所有这些元素称为原生库，但它们可以有不同的形式：

- 静态库和动态库：
  - 静态库可以使用，并将在应用程序的二进制文件中进行编译。
  - 动态库（通常有.dylib 扩展名）也可以使用，但必须是框架包的一部分。除了 Xcode 提供的系统 Swift 库外，iOS、watchOS 或 tvOS 上不支持独立的动态库。
- 框架（自 iOS 8 起）。框架是一个分层目录，它将动态库、头文件和资源（如故事板、图像文件和本地化字符串）封装到一个包中。
- 二进制框架（XCFrameworks）：Xcode 11 支持使用 XCFrameworks 格式发布二进制库，这是一种新的方式来捆绑一个框架的多个变体，例如，用于 Xcode 支持的任何平台（包括

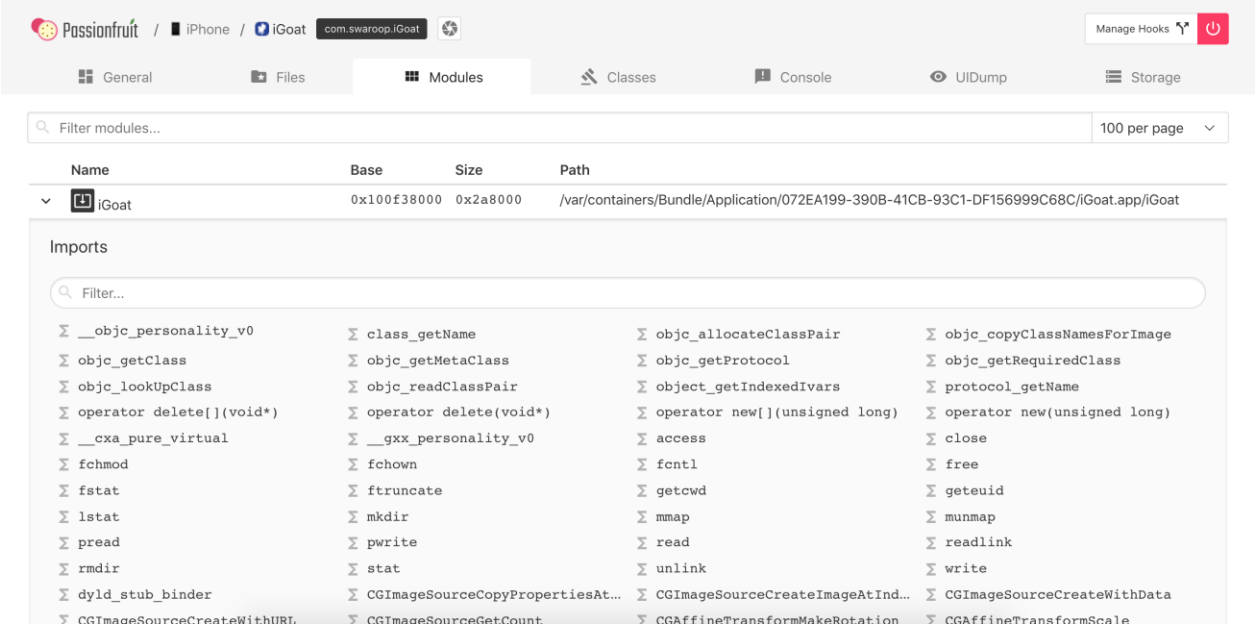
模拟器和设备)。它们还可以捆绑静态库（以及它们相应的头文件)，并支持基于 Swift 和 C 的二进制代码的分发。XCFrameworks 可以作为 Swift 包发布。

- Swift 包：Xcode 11 增加了对 Swift 包的支持，这是开发人员可以在他们的项目中使用的 Swift、Objective-C、Objective-C++、C 或 C++代码的可重用组件，并作为源代码分发。从 Xcode 12 开始，它们也可以捆绑资源，如图像、故事板和其他文件。由于包库默认是静态的。Xcode 编译它们，以及它们所依赖的包，然后将所有东西链接并组合到应用程序中。

您可以通过点击“Modules”可以很容易地从 Passionfruit 中看到原生库：



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）：

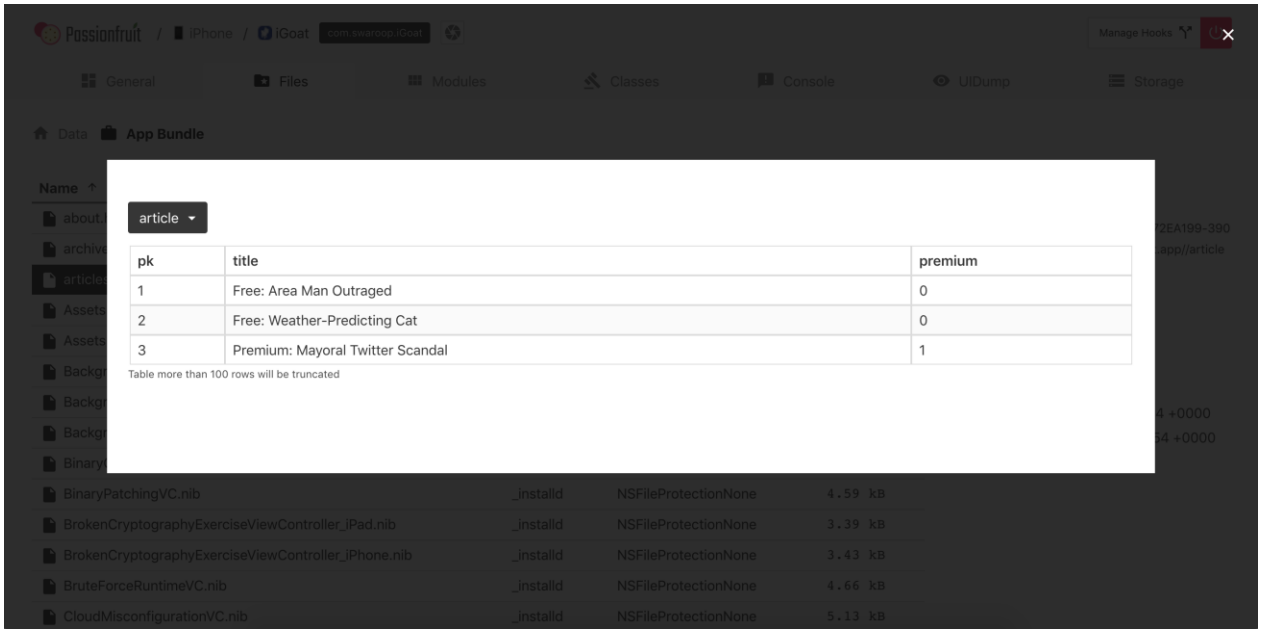
```
OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # ls
NSFileType      Perms  NSFileProtection     ...  Name
------------  -------  -------------------  ...  ----------------------------
Directory         493  None                 ...  Realm.framework
Regular           420  None                 ...  libswiftCore.dylib
Regular           420  None                 ...  libswiftCoreData.dylib
Regular           420  None                 ...  libswiftCoreFoundation.dylib
...
```

请注意，这可能不是应用程序使用原生代码元素的完整列表，因为其中有些可能是源代码的一部分，这意味着它们将在应用程序的二进制文件中编译，因此无法作为独立库或作为框架在 Frameworks 文件夹中找到。

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。

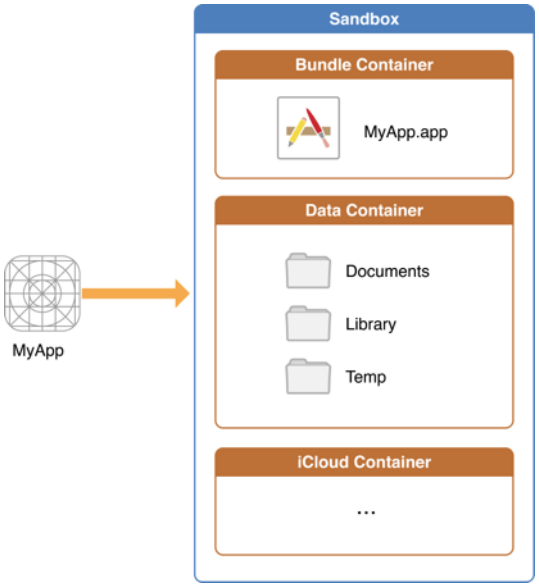
6.2.2.5.2.4. 其他应用程序资源

通常值得一看的是，你可能会在 IPA 内的应用程序包（.app）中发现其余的资源和文件，因为有些时候它们包含额外的好东西，如加密的数据库、证书等。



6.2.2.6.3. 访问 APP 数据目录

一旦您安装了应用程序，还有很多的信息要浏览。让我们简要介绍一下 iOS 应用中的应用程序文件夹结构，以了解哪些数据存储在哪里。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（通用唯一标识符)。

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，运行命令 **ipainstaller**（IPA 安装程序控制台)，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-FB
CA66589E67
Application: /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B2
74-FBCA66589E67/iGoat-Swift.app
Data: /private/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8
-8F5560EB0693
```

使用 objective 的命令 **env** 还能显示应用程序的所有目录信息。使用 objection 连接应用程序在“推荐工具——Objection”章节介绍。

```
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/**）。

这些文件夹包含了在应用安全评估期间必须仔细检查的信息（例如，在分析存储的数据是否为敏感数据时）。

包目录：

- **AppName.app**
  - 这是之前在 IPA 看到的应用程序包，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。
  - 应用程序可以通过设置 **NSURLIsExcludedFromBackupKey** 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/**
  - 包含半持久的缓存文件。
  - 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。
  - 应用可以通过设置 **NSURLIsExcludedFromBackupKey** 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应用程序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                ...  RandomKeyGenerationExerciseV
C.nib
Directory          493  None                ...  KeychainExerciseVC.nib
Directory          493  None                ...  CoreData.momd
Regular            420  None                ...  archived-expanded-entitlement
s.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

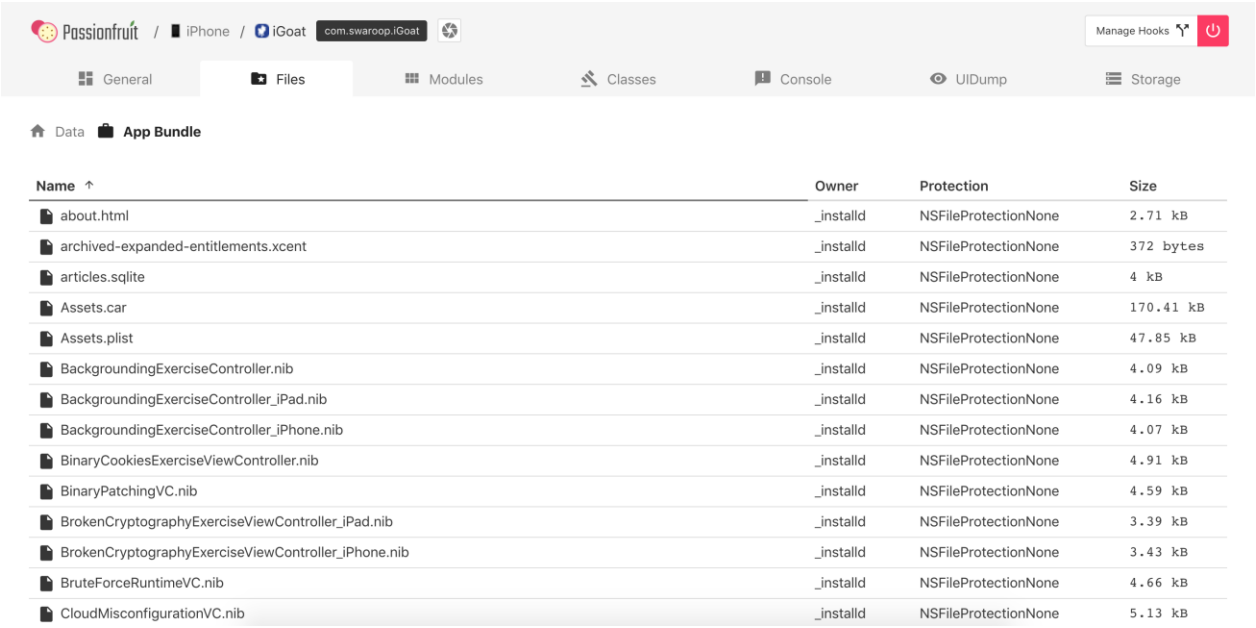
Directory          493  None                ...  Frameworks

Regular            420  None                ...  embedded.mobileprovision

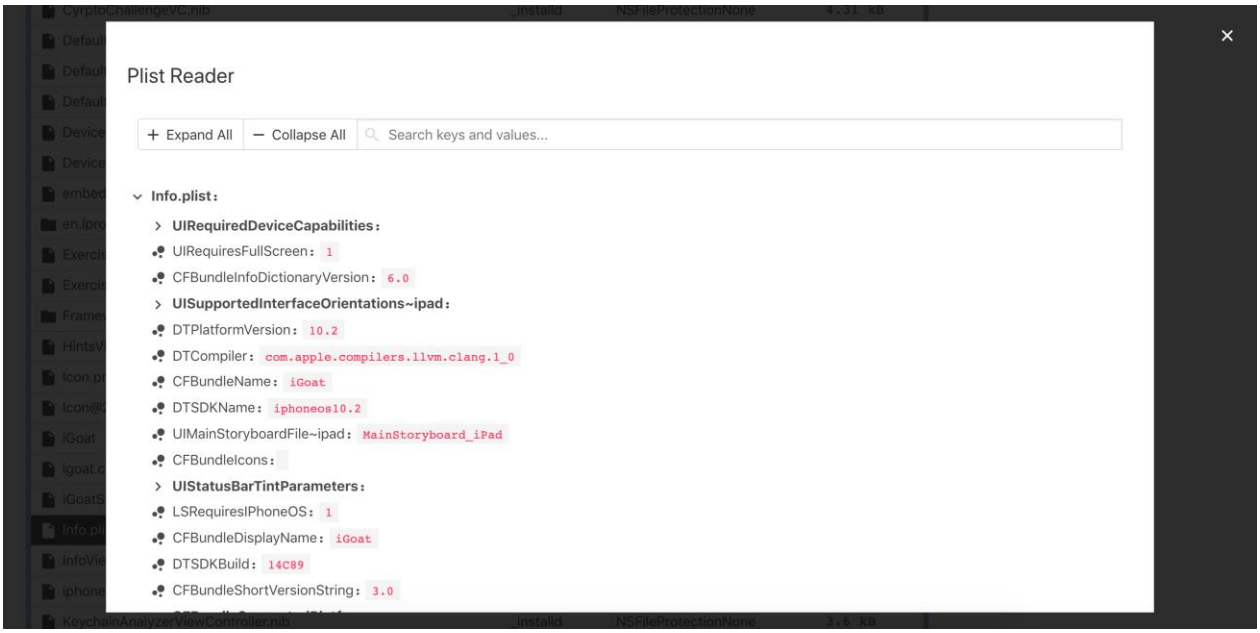
Regular            420  None                ...  Credentials.plist
Regular            420  None                ...  Assets.plist
Regular            420  None                ...  Info.plist

Regular            493  None                ...  iGoat-Swift
```

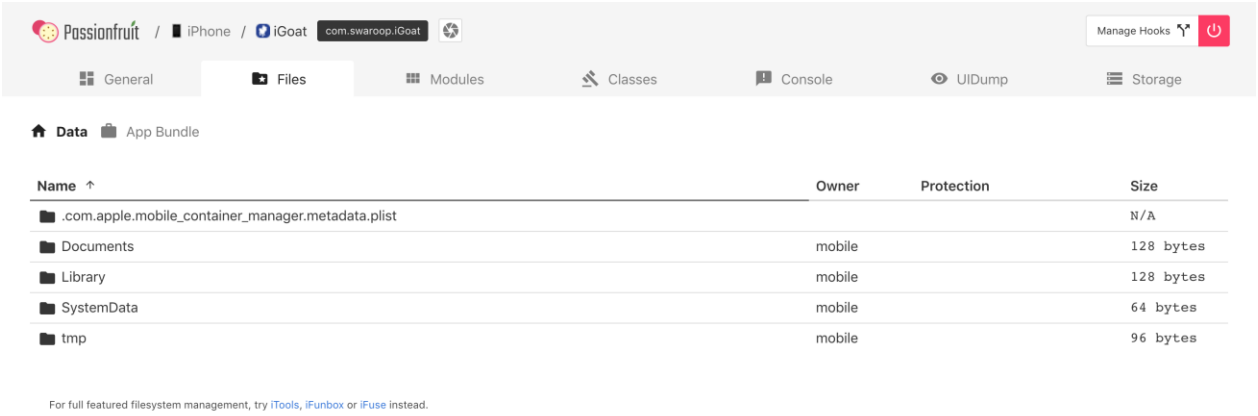
您还可以通过单击"Files" -> "App Bundle"来从 Passionfruit 直观的看到包目录：



包括 **`Info.plist`** 文件：



以及"Files" -> "Data"中的数据目录：



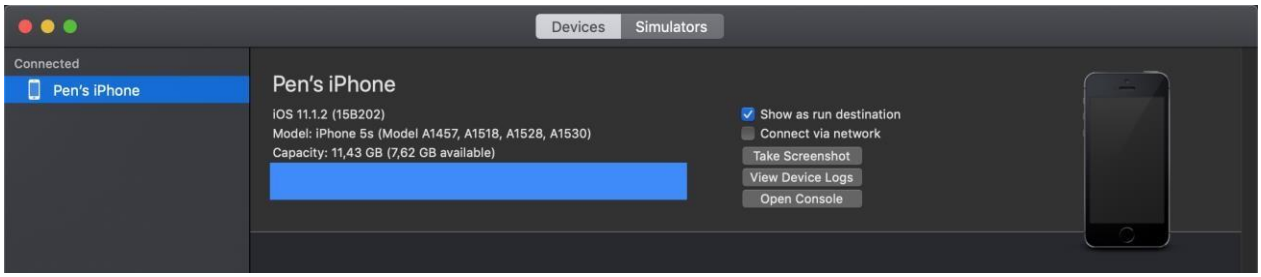
有关安全存储敏感数据的更多信息和最佳实践，请参阅“测试数据存储”章节。

6.2.2.6.4. 监控系统日志

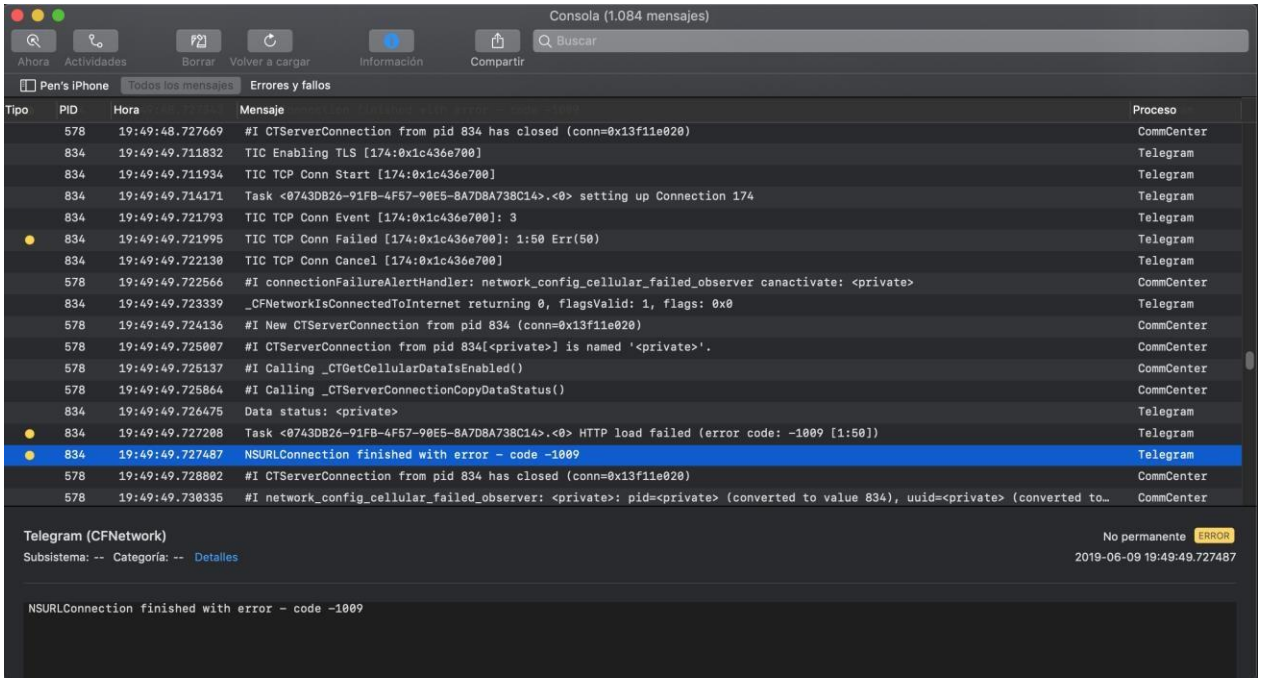
许多应用程序将信息（和潜在的敏感）记录到控制台日志。该日志还包含崩溃报告和其他有用的信息。你可以通过 Xcode 设备窗口收集控制台日志，具体方法如下。

许多应用程序会将信息（可能是敏感的）消息记录到控制台日志中。该日志也包含崩溃报告和其他有用的信息。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“Devices”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。



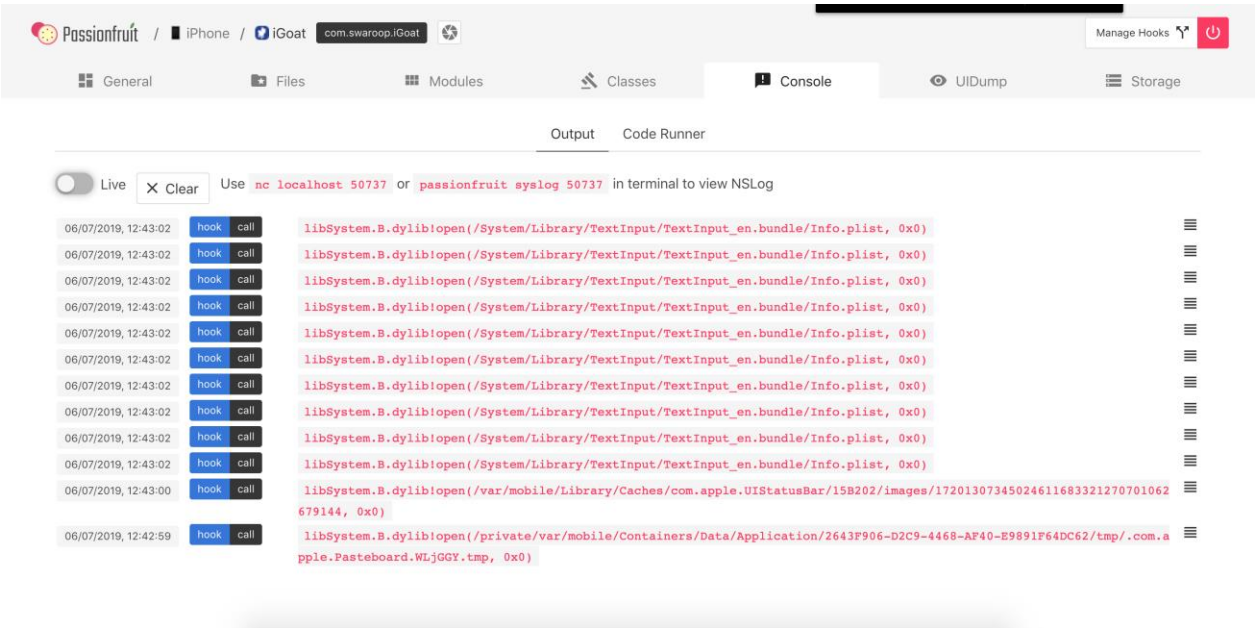
您还可以按照“访问设备 Shell”中的说明连接到设备 shell，通过 apt-get 安装 socat 并运行以下命令：

```
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) [ch
mod] (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) [to
uch] (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" 标签：



6.2.2.6.5. 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。

6.2.2.6.5.1. Objection (已越狱/未越狱)

使用 Objection 可以轻松查看 KeyChain 数据。首先，按照“推荐工具-Objection”中的描述将 Objection 连接到应用程序。然后，使用 **ios keychain dump** 命令获得 keychain 的概览：

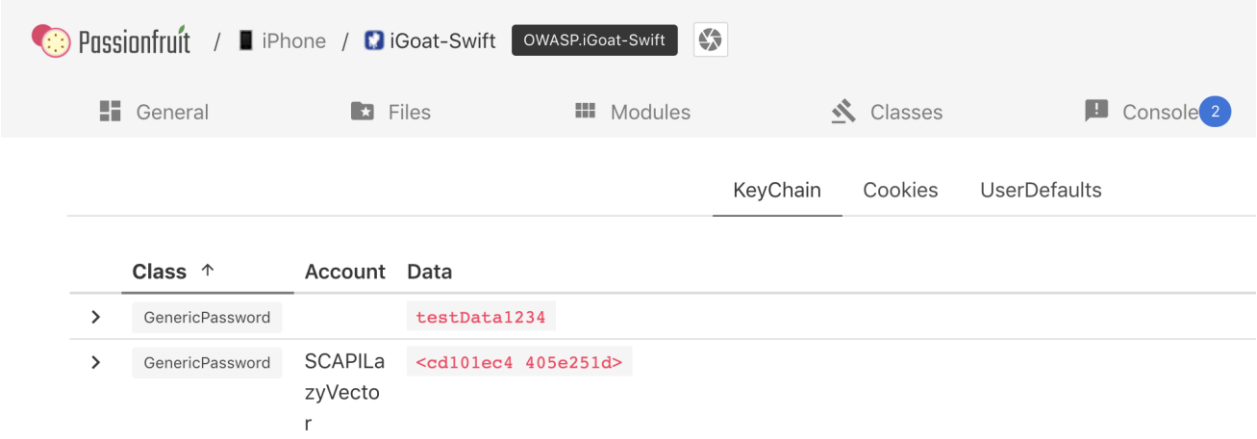
```
$ objection --gadget="iGoat-Swift" explore
... [usb] # ios keychain dump
...
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created                   Accessible                      ACL    Type      A
ccount              Service                     Data
------------------------  ----------------------------  -----  --------  -
----------------  ------------------------  ----------------------------
----------------------------------------
2019-06-06 10:53:09 +0000  WhenUnlocked                    None   Password  k
eychainValue        com.highaltitudehacks.dvia  mypassword123
2019-06-06 10:53:30 +0000  WhenUnlockedThisDeviceOnly      None   Password  S
CAPILazyVector      com.toyopagroup.picaboo     (failed to decode)
2019-06-06 10:53:30 +0000  AfterFirstUnlockThisDeviceOnly  None   Password  f
ideliusDeviceGraph  com.toyopagroup.picaboo     (failed to decode)
2019-06-06 10:53:30 +0000  AfterFirstUnlockThisDeviceOnly  None   Password  S
CDeviceTokenKey2    com.toyopagroup.picaboo     00001:FKsDMgVISiavdm70v9Fhv5z
+pZfBTTN7xkwSwNvVr2IhVBqLsC7QBhsEjKMxrEjh
2019-06-06 10:53:30 +0000  AfterFirstUnlockThisDeviceOnly  None   Password  S
CDeviceTokenValue2  com.toyopagroup.picaboo     CJ8Y8K2oE3rhOFUhnxJxDS1Zp8Z25
XzgY2EtFyMbW3U=
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：之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 是在应用程序内容中执行的，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。

6.2.2.6.5.2. Passionfruit(已越狱/未越狱)

使用 Passionfruit，可以访问选择的应用程序的 keychain 数据。单击“Storage”->“Keychain”，可以看到存储的 Keychain 信息的列表。



6.2.2.6.5.3 Keychain-dumper（越狱）

Keychain dumper 允许您转储越狱设备的 Keychain 内容。当安装完后再设备运行：

```
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: WOg1DfuH
```

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”权利的自签名证书进行了签名。权利授予对 Keychain 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签署适当的权限；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

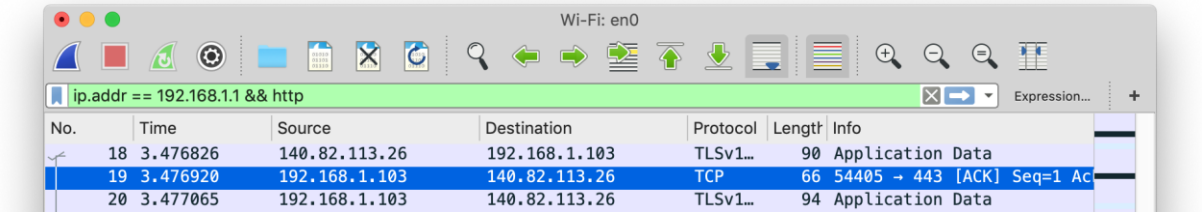
通过为 iOS 设备创建远程虚拟接口，可以远程实时嗅探 iOS 上的所有流量。首先确保您的 macOS 主机上安装了 Wireshark。

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，填写你 iOS 设备的 UDID。

```
$ rvictl -s <UDID>
Starting device <UDID> [SUCCEEDED] with interface rvi0
```

3. 启动 Wireshark 并选择“rvi0”作为捕获接口。
4. 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 Proxy 作为 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 流量。

设置 Burp 来代理您的流量非常简单。我们假设你的 iOS 设备和主机都连接到一个允许客户端对客户端通讯的 Wi-Fi 网络。如果客户端到客户端的流量不被允许，你可以使用 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 shell”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-> Wi-Fi。
2. 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，因为我们只是利用 Wi-Fi 的代理设置来设置全局代理）。
3. 连接后，单击已连接 Wi-Fi 右侧的蓝色小图标。
4. 通过选择 **Manual** 来配置代理。
5. 键入 127.0.0.1 作为 **Server**。
6. 输入 8080 作为 **Port**。

打开 Safari 访问任何网页，在 Burp 中应该能看到流量。感谢@hweisheimer 的最初想法！

6.2.3.3. 绕过证书固定

某些应用程序会实现 SSL 固定，这会阻止应用程序将拦截证书作为有效证书。这意味着您将无法监视应用程序和服务器之间的流量。

对于大多数应用程序来说，证书固定可以在几秒钟内被绕过，但前提是应用程序使用这些工具所涵盖的 API 功能。如果应用程序使用自定义框架或库实现 SSL 固定，则必须手动修补和停用 SSL 固定，这可能很耗时。

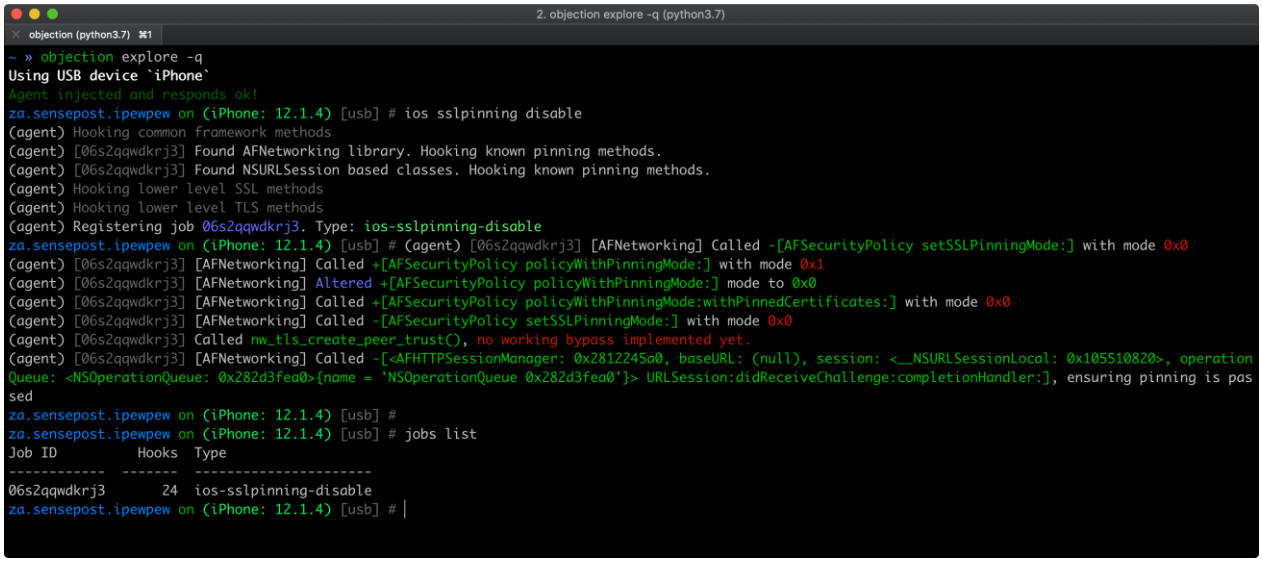
本节介绍了绕过 SSL 固定的各种方法，并指导你在现有工具不起作用时应该怎么做。

6.2.3.3.1. 适用于越狱设备和未越狱设备的方法

如果你有一个安装了 frida-server 的越狱设备，你可以通过运行下面的 Objection 命令绕过 SSL 固定（如果你使用的是非越狱设备，请重新打包你的应用程序）。

```
ios sslpinning disable
```

下面是输出的示例：



请参阅 Objection 关于禁用适用于 iOS 的 SSL 固定的帮助，以了解更多信息，并检查 pinning.ts 文件以了解绕过的工作原理。

6.2.3.3.2. 仅适用于越狱设备的方法

如果你有一个已越狱的设备，你可以尝试以下工具之一，它可以自动禁用 SSL 固定：

- "SSL Kill Switch 2 "是一种禁用证书固定的方法。它可以通过 Cydia 商店安装。它将劫持所有高级别 API 调用，并绕过证书固定。
- Burp 套件的移动助理应用程序也可以用来绕过证书固定。

6.2.3.3.3. 当自动绕过失败时

技术和系统随着时间的推移而变化，一些绕过技术最终可能会失效。因此，做一些研究是测试人员工作的一部分，因为不是每个工具都能迅速跟上操作系统的版本。

一些应用程序可能会实现自定义的 SSL 固定方法，所以测试人员也可以利用现有的脚本作为基础或灵感，亦或是使用类似的技术，但针对应用程序的自定义 API 开发新的绕过脚本。在这里，你可以检查这种脚本的三个不错示例：

- "objection – 固定绕过模块" (pinning.ts)
- "Frida CodeShare - ios10-ssl-bypass" 作者：@dki
- "用 OkHttp 绕过混淆的应用程序中的 SSL 固定" 作者：Jeroen Beckers

6.2.3.3.4. 其他技术

如果你不能访问源代码，你可以尝试修补二进制：

- 如果使用的是 OpenSSL 的证书绕过，你可以试试二进制补丁。
- 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很有效，但要注意证书的 SHA 值。如果它被硬编码到二进制文件中，你也必须把它替换掉。
- 如果你能访问源代码，你可以尝试禁用证书固定，并推荐应用程序，寻找 NSURLSession、CFStream 和 AFNetworking 的 API 调用以及包含 "pinning"、"X.509"、"Certificate "等字样的方法/字符串。

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?lan guage=objc
- UIDeviceFamily - https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKe yReference/Articles/iPhoneOSKeys.html#//apple_ref/doc/uid/TP40009252-SW11

6.3. iOS 数据存储

敏感数据保护（例如身份认证令牌和个人信息）是移动安全的关键。在本章中，将介绍关于 iOS 的本地数据存储 API，以及使用它们的最佳实践。

6.3.1. 测试本地数据存储（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，在大多数实际场景下，至少有一些用户数据必须被存储。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，敏感数据和文件可以通过硬件支持的 256 位 AES 加密来保证安全。

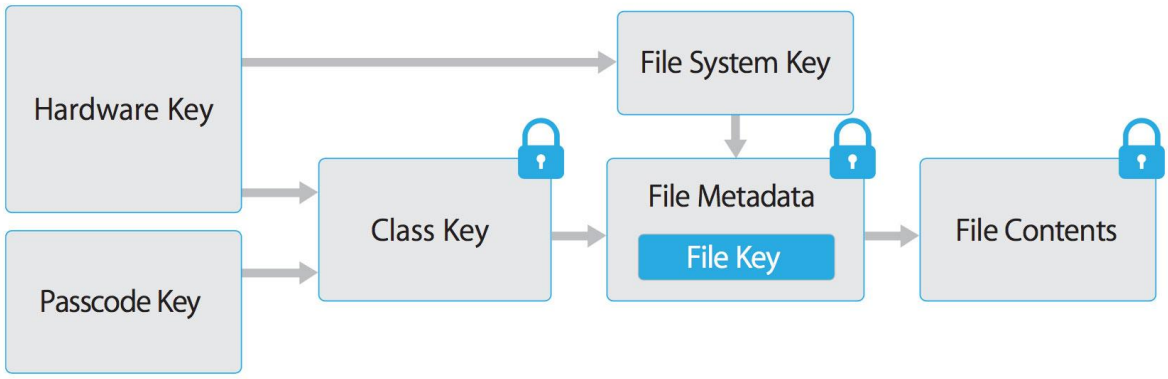
6.3.1.1. 数据保护 API

应用程序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 引入的安全隔区处理器 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），内嵌在安全隔区中，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 文件系统中的每个文件都用它自己的密钥进行加密，该密钥包含在文件元数据中。元数据用文件系统密钥进行加密，并用应用程序在创建文件时选择的保护类别对应的类密钥进行包装。

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。
- **在第一次用户身份认证之前受保护（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。这个保护类别的存在是为了快速远程擦除（立即删除类密钥，这使得数据无法访问）。

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“在第一次用户身份认证之前受保护”。

6.3.1.1.1. Keychain

iOS Keychain 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 Keychain API 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 Keychain，并且每个登录账户都有自己的 Keychain。但是，iOS 上的 Keychain 结构与 macOS 上是不同的：iOS 上所有应用程序只能使用一个 Keychain。通过 kSecAttrAccessGroup 属性的访问组功能，在由同一开发者签名的应用程序之间共享对项目的访问。Keychain 的访问由 securityd 守护进程管理，该守护进程根据应用程序的 **`Keychain-access-groups`**、**`application-identifier`** 和 **`application-group`** 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 Keychain 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 Keychain 的条目被编码为二进制 plist，并在 Galois/ Counter 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 ksecattracsible 键，来为 Keychain 项配置数据保护。以下是 kSecAttrAccessible 的可配置可访问值，是 Keychain 数据保护类。：

- 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 本地备份中。
- kSecAttrAccessibleAfterFirstUnlock：重新启动后，在用户第一次解锁设备之前，无法访问 Keychain 项中的数据。

- kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly：重新启动后，在用户第一次解锁设备之前，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 本地备份中。
- kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly：只有在设备解锁时才能访问 Keychain 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 本地备份中。

AccessControlFlags 定义用户验证密钥的机制（SecAccessControlCreateFlags）：

- `kSecAccessControlDevicePasscode`：通过密码访问项目。
- `kSecAccessControlBiometryAny`：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。
- `kSecAccessControlBiometryCurrentSet`：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 Touch ID）或默认密码访问项目。

请注意，由 Touch ID 保护的密钥（通过 `kSecAccessControlBiometryAny` 或 `kSecAccessControlBiometryCurrentSet`）受安全隔区保护：Keychain 仅持有令牌，而不是实际密钥。密钥保留在安全隔区中。

从 iOS9 开始，基于 ECC 的签名操作在安全隔区中执行。在该场景中，私钥和加密操作驻留在安全隔区中。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 Keychain 中，因为它不能存储在安全隔区中。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 Keychain 中的项。从 iOS 9 开始，可以使用 `LAContext` 检查是否设置了锁屏：

Swift:

```
public func devicePasscodeEnabled() -> Bool {
    return LAContext().canEvaluatePolicy(.deviceOwnerAuthentication, error: n
il)
}
```

Objective-C:

```
-(BOOL)devicePasscodeEnabled:(LAContex)context{
  if ([context canEvaluatePolicy:LAPolicyDeviceOwnerAuthentication error:ni
l]) {
        return true;
    } else {
        return false;
    }
}
```

6.3.1.1.1.1. Keychain 数据持久性

在 iOS 上，卸载应用时，应用使用的 Keychain 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能力支持。

在评估 iOS 应用程序时，需要查看 Keychain 数据持久性。可以通过安装使用应用程序，生成可能存储在 Keychain 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。使用 objection 运行时探测工具来转储 Keychain 数据。下面的 objection 命令演示了这个过程：

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios keychain du
mp
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      A
ccount                   Service
            Data
-------------------------  ----------------------------  -----  --------  -
-----------------------  ------------------------------------------------------
----------  ------------------------------------
2020-02-11 13:26:52 +0000  WhenUnlocked                  None   Password  k
eychainValue             com.highaltitudehacks.DVIAswiftv2.develop
            mysecretpass123
```

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 Keychain 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 Keychain 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。下面的 Swift 例子是这种擦拭程序的基本演示：

```
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 应用 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 Keychain 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

当你可以访问一个 iOS 应用程序的源代码时，识别整个应用程序中保存和处理的敏感数据。这包括密码、密匙和个人可识别信息（PII），但也可能包括其他被行业法规、法律和公司政策认定为敏感的数据。寻找通过下面列出的任何本地存储 API 保存的这些数据。

确保敏感数据在没有适当保护的情况下不会被存储。例如，在没有额外加密的情况下，不应该在 `NSUserDefaults` 中保存认证令牌。也要避免将加密密钥存储在.plist 文件中，在代码中硬编码为字符串，或使用可预测的混淆功能或基于稳定属性的密钥衍生功能生成。

敏感数据应通过使用 Keychain API（将它们存储在安全隔区内）进行存储，或使用信封加密进行存储。信封加密，或称密钥包装，是一种使用对称加密来封装密钥材料的密码学结构。数据加密密钥（DEK）可以用密钥加密密钥（KEK）进行加密，这些密钥必须安全地存储在 Keychain 中。加密的 DEK 可以存储在 `NSUserDefaults` 中或写入文件中。当需要时，应用程序读取 KEK，然后解密 DEK。请参考 OWASP 加密存储备忘录，了解更多关于加密密钥的信息。

6.3.1.2.1. Keychain

必须实现加密，以便密钥以安全设置存储在 Keychain 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 Apple 官方文档中找到。Apple 的官方文档还包括一个使用 Touch ID 和密码保护密钥的例子。

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安全隔区):

```
// 私钥参数
let privateKeyParams = [
    kSecAttrLabel as String: "privateLabel",
    kSecAttrIsPermanent as String: true,
    kSecAttrApplicationTag as String: "applicationTag",
] as CFDictionary

// 公钥参数
let publicKeyParams = [
    kSecAttrLabel as String: "publicLabel",
    kSecAttrIsPermanent as String: false,
    kSecAttrApplicationTag as String: "applicationTag",
] as CFDictionary

//全局参数
let parameters = [
    kSecAttrKeyType as String: kSecAttrKeyTypeEC,
    kSecAttrKeySizeInBits as String: 256,
    kSecAttrTokenID as String: kSecAttrTokenIDSecureEnclave,
    kSecPublicKeyAttrs as String: publicKeyParams,
    kSecPrivateKeyAttrs as String: privateKeyParams,
] as CFDictionary

var pubKey, privKey: SecKey?
let status = SecKeyGeneratePair(parameters, &pubKey, &privKey)

if status != errSecSuccess {
    // 密钥成功创建
}
```

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：

6.3.1.2.2. 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。

6.3.1.2.3. 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。

下面的例子演示了如何使用 FileManager 类来创建一个完整的加密文件。你可以在苹果开发者文档中找到更多信息 "加密你的应用程序的文件"：

Swift:

```
FileManager.default.createFile(
    atPath: filePath,
    contents: "secret text".data(using: .utf8),
```

```
    attributes: [FileAttributeKey.protectionKey: FileProtectionType.complete]
)
```

Objective-C:

```
[[NSFileManager defaultManager] createFileAtPath:[self filePath]
  contents:[@"secret text" dataUsingEncoding:NSUTF8StringEncoding]
  attributes:[NSDictionary dictionaryWithObject:NSFileProtectionComplete
  forKey:NSFileProtectionKey]];
```

6.3.1.2.4. 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，可以为 CoreData 添加一个额外的加密层。更多细节请参见 Github 仓库 。

6.3.1.2.5. 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。这个库由 C++包装，它提供 SQLite 命令的 API。

6.3.1.2.6. 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端，甚至在应用程序脱机时也保持可用。

一个配置错误的 Firebase 实例可以通过以下网络调用来识别：

```
https://\<firebaseProjectName\>.firebaseio.com/.json
```

firebaseProjectName 可以从属性列表(.plist)文件中检索到。例如，PROJECT_ID 键在 *GoogleService-Info.plist* 文件中存储了相应的 Firebase 项目名称。

另外，分析人员可以使用 Firebase Scanner，这是一个 python 脚本，可以自动完成上述任务，如下所示。

```
python FirebaseScanner.py -f <commaSeparatedFirebaseProjectNames>
```

6.3.1.2.7. Realm 数据库

Realm Objective-C 和 Realm Swift 并不是由 Apple 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：

```
// 打开加密的 Realm 文件，其中 getKey()是一个从 Keychain 或服务器获取密钥的方法。
let config = Realm.Configuration(encryptionKey: getKey())
do {
  let realm = try Realm(configuration: config)
  // 照常使用 Realm
} catch let error as NSError {
  // 如果加密密钥是错误的，`error`会说这是一个无效的数据库。
  fatalError("Error opening realm: \(error)")
}
```

6.3.1.2.8. 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 为 iOS 和 macOS 进行了原生编译。

6.3.1.2.9. 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原生 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，因为可能只有在特定的功能被触发之后，应用程序才存储敏感数据。然后，您可以根据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的 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 设备并浏览到包目录（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“Apps 应用程序”，右键单击目标的 iOS 应用程序，然后选择“Extract App 提取应用程序”。
4. 浏览到输出目录并找到**$APP_NAME.imazing**. 重命名为**$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 objection 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“未越狱设备的动态分析”（来自“iOS 系统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 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 Keychain 内容。在越狱设备上，可以使用 Keychain dumper 导出 Keychain 中数据，如“iOS 上的基本安全测试”章节所述。

Keychain 文件的路径是

```
/private/var/Keychains/keychain-2.db
```

在未越狱设备上，您可以使用 objection 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，可以通过以下命令浏览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你的应用程序中的一个关键字。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 Objection 进行动态分析

你可以使用 objection 运行时移动检查工具包来寻找由应用程序的数据存储机制引起的漏洞。未越狱的设备也可以使用 objection，但需要修补 iOS 应用程序。

6.3.1.3.2.1. 读取 Keychain

要使用 Objection 读取 Keychain，请执行以下命令：

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios keychain du
mp
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      A
ccount                   Service
           Data
-------------------------  ------------------------------  -----  --------  -
-----------------------  ------------------------------------------------------
----------  ------------------------------------
2020-02-11 13:26:52 +0000  WhenUnlocked                    None   Password  k
eychainValue             com.highaltitudehacks.DVIAswiftv2.develop
           mysecretpass123
```

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 WebView 的 cookie 数据的二进制文件。可以使用 objection 将这些文件转换为 JSON 格式并检查数据。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios cookies get
 --json
[
    {
        "domain": "highaltitudehacks.com",
        "expiresDate": "2051-09-15 07:46:43 +0000",
        "isHTTPOnly": "false",
        "isSecure": "false",
        "name": "username",
        "path": "/",
        "value": "admin123",
        "version": "0"
    }
]
```

6.3.1.3.2.3. 搜索属性列表文件

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 **`ios plist cat plistFileName.plist`** 命令来检查 plist 文件。

要找到 userInfo.plist 这个文件，可以使用 **env** 命令。它将打印出应用程序的库、缓存和文档目录的位置：

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # env
Name               Path
-----------------  ------------------------------------------------------------
-------------------------------
BundlePath         /private/var/containers/Bundle/Application/B2C8E457-1F0C-4
DB1-8C39-04ACBFFEE7C8/DVIA-v2.app
CachesDirectory    /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D
-8701-C020C301C151/Library/Caches
DocumentDirectory  /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D
-8701-C020C301C151/Documents
LibraryDirectory   /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D
-8701-C020C301C151/Library
```

转到 Documents 目录，用 ***ls*** 命令列出所有文件。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ls
NSFileType      Perms  NSFileProtection                     Read    Write
 Owner         Group         Size      Creation                  Name
```

```
------------  -------  ------------------------------------  ------  -------
 ------------  ------------  --------  ------------------------  -----------
-------------
Directory          493  n/a                                          True    True
 mobile (501)  mobile (501)  192.0 B   2020-02-12 07:03:51 +0000  default.rea
lm.management
Regular            420  CompleteUntilFirstUserAuthentication  True    True
 mobile (501)  mobile (501)  16.0 KiB  2020-02-12 07:03:51 +0000  default.rea
lm
Regular            420  CompleteUntilFirstUserAuthentication  True    True
 mobile (501)  mobile (501)  1.2 KiB   2020-02-12 07:03:51 +0000  default.rea
lm.lock
Regular            420  CompleteUntilFirstUserAuthentication  True    True
 mobile (501)  mobile (501)  284.0 B   2020-05-29 18:15:23 +0000  userInfo.pl
ist
Unknown            384  n/a                                          True    True
 mobile (501)  mobile (501)  0.0 B     2020-02-12 07:03:51 +0000  default.rea
lm.note

Readable: True  Writable: True
```

执行 ***ios plist cat*** 命令，检查 userInfo.plist 文件的内容。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios plist cat u
serInfo.plist
{
        password = password123;
        username = userName;
}
```

6.3.1.3.2.4. 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。测试人员应该检查这些文件的数据保护值和它们的内容是否有敏感数据。Objection 包含一个与 SQLite 数据库交互的模块。它允许转储 schema，表和查询记录。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # sqlite connect
Model.sqlite
Caching local copy of database file...
Downloading /var/mobile/Containers/Data/Application/264C23B8-07B5-4B5D-8701-C
020C301C151/Library/Application Support/Model.sqlite to /var/folders/4m/dsg0m
q_17g39g473z0996r7m0000gq/T/tmpdr_7rvxi.sqlite
Streaming file from device...
Writing bytes to destination...
Successfully downloaded /var/mobile/Containers/Data/Application/264C23B8-07B5
-4B5D-8701-C020C301C151/Library/Application Support/Model.sqlite to /var/fold
```

```
ers/4m/dsg0mq_17g39g473z0996r7m0000gq/T/tmpdr_7rvxi.sqlite
Validating SQLite database format
Connected to SQLite database at: Model.sqlite

SQLite @ Model.sqlite > .tables
+--------------+
| name         |
+--------------+
| ZUSER        |
| Z_METADATA   |
| Z_MODELCACHE |
| Z_PRIMARYKEY |
+--------------+
Time: 0.013s

SQLite @ Model.sqlite > select * from Z_PRIMARYKEY
+-------+--------+---------+-------+
| Z_ENT | Z_NAME | Z_SUPER | Z_MAX |
+-------+--------+---------+-------+
| 1     | User   | 0       | 0     |
+-------+--------+---------+-------+
1 row in set
Time: 0.013s
```

6.3.1.3.2.5. 搜索缓存数据库

默认情况下，NSURLSession 在 Cache.db 数据库中存储数据，如 HTTP 请求和响应。如果令牌、用户名或任何其他敏感信息被缓存了，这个数据库可能包含敏感数据。要找到缓存的信息，请打开应用程序的数据目录（**/var/mobile/Containers/Data/Application/<UUID>**）并进入**/Library/Caches/<Bundle Identifier>**。WebKit 的缓存也被存储在 Cache.db 文件中。Objection 可以通过命令 sqlite connect Cache.db 打开并与数据库交互，因为它是一个正常的 SQLite 数据库。

建议这个数据禁用缓存，因为它可能在请求或响应中包含敏感信息。下面的列表显示了实现这一目的的不同方法。

1. 建议在注销后删除缓存的响应。这可以通过 Apple 公司提供的名为 **removeAllCachedResponses** 的方法来完成，你可以按以下方式调用该方法：

   ```
   URLCache.shared.removeAllCachedResponses()
   ```

   这个方法将从 Cache.db 文件中删除所有缓存的请求和响应。

2. 如果你不需要使用 cookie 的优势，建议只使用 URLSession 的.ephemeral 配置属性，这将禁止保存 cookie 和缓存。

   Apple 文档：

   一个 ephemeral 会话配置对象类似于默认的会话配置（见默认设置），只是相应的会话对象不把缓存、凭证存储或任何与会话相关的数据存储到磁盘上。相反，会话相关的数据被存储在 RAM 中。ephemeral 会话将数据写入磁盘的唯一时间是当你告诉它将一个 URL 的内容写入文件。

3. 缓存也可以通过设置缓存策略为.notAllowed 来禁用。它将禁止以任何方式存储 Cache，无论是在内存还是在磁盘上。

6.3.2. 测试日志中敏感数据（MSTG-STORAGE-3）

在移动设备上创建日志文件有许多合理的理由，包括在设备离线时跟踪存储在本地的崩溃或错误（以便一旦上线就可以发送给应用程序的开发者），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。你可以通过在适当的 PREFIX_HEADER(*.pch)文件中添加以下代码来实现这一目标：

```
#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。浏览到一个显示接收敏感用户信息的输入字段的屏幕。

启动其中一个方法后，填写输入字段。如果输出中显示敏感数据，则该应用程序未能通过此测试。

6.3.3. 确定敏感数据是否发送给第三方（MSTG-STORAGE-4）

6.3.3.1. 概述

敏感信息可能通过几种方式泄露给第三方。在 iOS 上，通常是通过内置在应用程序中的第三方服务。

这些服务提供的功能可能涉及跟踪服务，以监测用户在使用应用程序时的行为，销售横幅广告，或改善用户体验。

缺点是，开发者通常不知道通过第三方库执行的代码的细节。因此，不应该向服务发送超过必要的信息，也不应该披露敏感信息。

大多数第三方服务是以两种方式实现的。

- 使用独立的库。

- 使用完整的 SDK。

6.3.3.2. 静态分析

为了确定第三方库提供的 API 调用和功能是否按照最佳实践使用，请审查其源代码、请求的权限并检查任何已知的漏洞（见 "检查第三方库的弱点（MSTG-CODE-5）"）。

所有发送到第三方服务的数据都应该是匿名的，以防止暴露 PII（个人可识别信息），使第三方能够识别用户账户。应用程序不应将其他数据（如可以映射到用户账户或会话的 ID）发送到第三方。

6.3.3.3. 动态分析

检查所有对外部服务的请求是否有内置的敏感信息。为了拦截客户端和服务器之间的流量，你可以用 Burp Suite Professional 或 OWASP ZAP 发起中间人（MITM）攻击来进行动态分析。一旦你通过拦截代理路由流量，你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主要功能所在的服务器上的应用程序请求，都应该检查敏感信息，如跟踪器或广告服务中的 PII。

6.3.4. 在键盘缓存中查找敏感数据（MSTG-STORAGE-5）

iOS 为用户提供几个简化键盘输入的选项。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。

UITextInputTraits 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`**，对于大多数输入法来说，它可以启用自动更正。
- **`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 NO。

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 和故事板文件，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。你可以通过程序化方式禁用缓存，在所需的 UITextFields、UITextViews 和 UISearchBars 中使用 textObject.autocorrectionType = UITextAutocorrectionTypeNo 指令来防止缓存。对于应该被屏蔽的数据，如 PIN 和密码，将 textObject.secureTextEntry 设置为 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 Settings”>“常规 General”>“重置 Reset”>“重置键盘字典 Reset Keyboard Dictionary”，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。

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

如果您必须使用未越狱 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，并且应该优先使用这种方法。它可以在最受限制的环境中运行：沙盒中没有 root 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：
  - NSXPCConnection 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）。它们可以原生实现，也可以通过 Core Foundation（CFMachPort）和 Foundation 包装（NSMachPort）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

以下部分总结了你应该寻找的关键字，以识别 iOS 源代码中的 IPC 实现。

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 语言）：

- xpc.h
- connection.h

6.3.5.2.2. Mach 端口

在底层实现中要查找的关键字：

- mach_port_t
- mach_msg_*

在高层实现中要查找的关键字（Core Foundation 和 Foundation 包装）：

- 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. 检查用户界面泄露的敏感数据（MSTG-STORAGE-7）

6.3.6.1. 概述

在注册账户或进行支付时输入敏感信息，是使用许多应用程序的一个重要部分。这些数据可能是金融信息，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适当地屏蔽它，这些数据可能会被暴露。

为了防止泄露和减少诸如肩窥的风险，你应该核实没有敏感数据通过用户界面暴露出来，除非有明确的要求（例如，正在输入密码）。对于需要出现的数据，应该进行适当的屏蔽，典型的做法是显示星号或圆点，而不是明文。

仔细审查所有显示此类信息或将其作为输入的用户界面组件。搜索任何敏感信息的痕迹，并评估它是否应该被屏蔽或完全删除。

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：

故事板 Storyboard：在 iOS 项目的故事板中，浏览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 Secure Text Entry”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. 测试备份中敏感数据（MSTG-STORAGE-8）

6.3.7.1. 概述

iOS 包括自动备份功能，可以创建存储在设备上的数据副本。你可以通过使用 iTunes（macOS Catalina 之前）或 Finder（从 macOS Catalina 开始），或通过 iCloud 备份功能，从主机上进行 iOS 备份。在这两种情况下，除了 Apple Pay 信息和 Touch ID 设置等高度敏感的数据外，备份几乎包括存储在 iOS 设备上的所有数据。

由于 iOS 备份了已安装的应用程序及其数据，一个明显的担忧是应用程序存储的敏感用户数据是否会无意中通过备份泄露。另一个尽管不那么明显的担忧是用于保护数据或限制应用功能的敏感配置设置是否会被篡改，以便在恢复修改过的备份后改变应用行为。这两种担心都是有道理的，而且这些漏洞已被证明存在于当今大量的应用程序中。

6.3.7.1.1. Keychain 是如何备份的

当用户备份他们的 iOS 设备时，Keychain 数据也会被备份，但 Keychain 中的密钥仍然被加密。解密 Keychain 数据所需的类密钥并不包括在备份中。恢复 Keychain 数据需要将备份恢复到设备上，并使用用户的密码解锁该设备。

设置了 **`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`** 属性的 Keychain 项目只有在备份恢复到原备份设备上时才能被解密。试图从备份中提取这些 Keychain 数据的人，如果不能接触到原设备内的加密硬件，就无法解密它。

然而，使用 Keychain 的一个注意事项是，它只被设计用来存储少量的用户数据或简短的记录（根据 Apple 的 Keychain 服务文档）。这意味着有较大的本地安全存储需求的应用程序（例如，通讯应用程序等）应该在应用程序容器内加密数据，但使用 Keychain 来存储密钥材料。在敏感配置设置（如数据丢失预防策略、密码策略、合规性策略等）必须在应用容器内保持不加密的情况下，你可以考虑在 Keychain 中存储策略的哈希值，以便进行完整性检查。如果没有完整性检查，这些设置可能会在备份中被修改，然后恢复到设备上，以修改应用行为（例如，改变配置的远程终端）或安全设置（例如，越狱检测、证书固定、最大 UI 登录尝试等）。

经验之谈。如果敏感数据按照本章前面的建议进行处理（例如，存储在 Keychain 中，有 Keychain 支持的完整性检查，或用锁定在 Keychain 中的密钥进行加密），备份不应该是安全问题。

6.3.7.2. 静态分析

安装了移动应用程序的设备的备份将包括所有子目录（除了 **Library/Caches/**）和应用程序私有目录中的文件。

因此，应避免在应用程序的私有目录或子目录中的任何文件或文件夹中以明文方式存储敏感数据。

尽管默认情况下备份包括 **Documents/**和 **Library/Application Support/**中的所有文件，但您可以通过调用带有 **NSURLIsExcludedFromBackupKey** 属性的 **NSURL setResourceValue:forKey:error:**方法从备份中排除文件。

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。

这两个文件系统属性都比直接设置扩展属性这种被废弃的方法要好。所有运行在 iOS 5.1 及以后版本的应用程序都应该使用这些属性来从备份排除数据。

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，更多信息请参见 Swift 从 iCloud 备份中排除文件：

```
enum ExcludeFileError: Error {
    case fileDoesNotExist
    case error(String)
}

func excludeFileFromBackup(filePath: URL) -> Result<Bool, ExcludeFileError> {
    var file = filePath

    do {
        if FileManager.default.fileExists(atPath: file.path) {
            var res = URLResourceValues()
            res.isExcludedFromBackup = true
            try file.setResourceValues(res)
            return .success(true)

        } else {
            return .failure(.fileDoesNotExist)
        }
    } catch {
        return .failure(.error("Error excluding \(file.lastPathComponent) fro
m backup \(error)"))
    }
}
```

6.3.7.3. 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（直到 macOS Mojave）。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 Encrypt local backup”选项，以便备份以明文形式存储在硬盘上。

> 从 macOS Catalina 开始，iTunes 不再可用了。对 iOS 设备的管理，包括更新、备份和恢复，已经转移到 Finder 应用程序。方法仍然相同，如上所述。

当 iOS 设备进行备份之后，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。Apple 官方文档可以帮助我们找到 iPhone、iPad 和 iPod touch 的备份文件。

当你想导航到备份文件夹时，直到 High Sierra，都可以轻松做到。但从 macOS Mojave 开始，你会得到以下错误（即便以 root 身份）：

```
$ 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。

一旦知道了 UDID，就可以浏览到这个目录，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混淆的，如下所示：

```
$ pwd
/Users/foo/Library/Application Support/MobileSync/Backup/416f01bd160932d2bf2f
95f1f142bc29b1c62dcb/00
$ ls | head -n 3
000127b08898088a8a169b4f63b363a3adcf389b
0001fe89d0d03708d414b36bc6f706f567b08d66
000200a644d7d2c56eec5b89c1921dacbec83c3e
```

因此，浏览 iOS 设备备份并不简单，你不会在目录或文件名中找到你要分析的应用程序的任何提示。在这里，你可以考虑使用 iMazing 共享软件工具进行辅助。用 iMazing 进行设备备份，并使用其内置的备份资源管理器，轻松分析应用容器内容，包括原始路径和文件名。

如果没有 iMazing 或类似软件，你可能需要求助于使用 grep 来识别敏感数据。这不是最彻底的方法，但你可以尝试搜索你在进行备份之前使用应用程序时键入的敏感数据。例如：用户名、密码、信用卡数据、PII 或任何在应用程序背景下被认为是敏感的数据。

```
~/Library/Application Support/MobileSync/Backup/<UDID>
grep -iRn "password" .
```

正如静态分析部分所述，你能找到的任何敏感数据都应该从备份中排除，通过使用 Keychain 进行适当加密，或者最开始就不存储在设备上。

要识别一个备份是否被加密，你可以从位于备份目录根部的 "Manifest.plist "文件中检查名为 "IsEncrypted "的键值。下面的例子显示了一个表明备份已被加密的配置。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DT
```

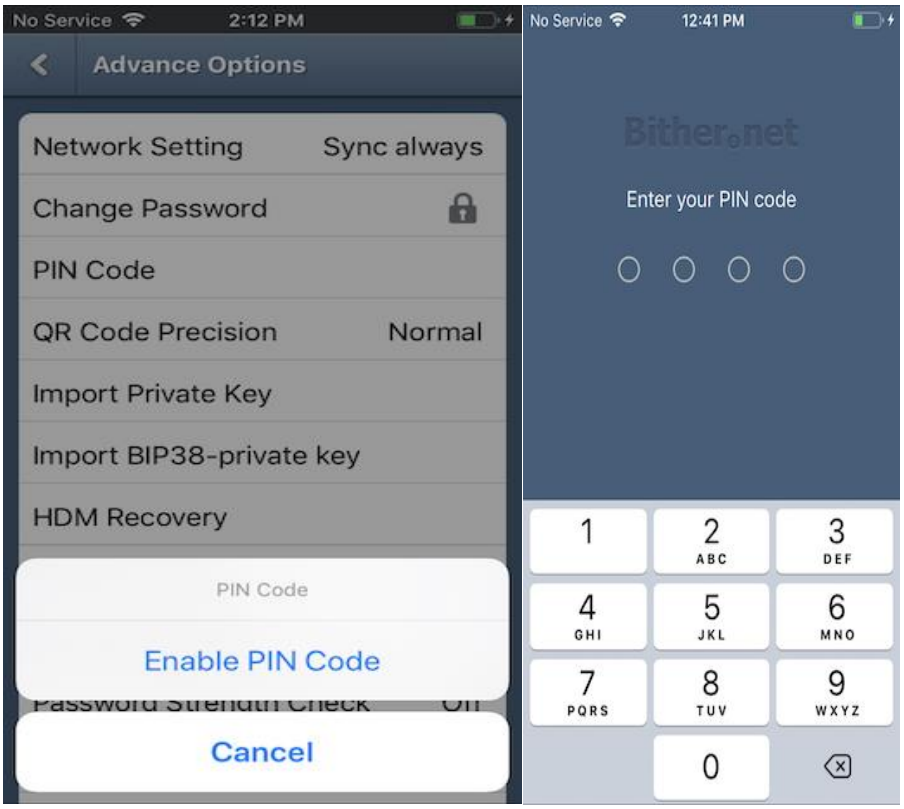
```
Ds/PropertyList-1.0.dtd">
<plist version="1.0">
...
 <key>Date</key>
 <date>2021-03-12T17:43:33Z</date>
 <key>IsEncrypted</key>
 <true/>
...
</plist>
```

如果你需要处理加密的备份，在 DinoSec 的 Github 仓库中有一些 Python 脚本，如 backup_tool.py 和 backup_passwd.py，它们将作为一个好的开始。然而，请注意，它们可能无法在最新的 iTunes/Finder 版本中工作，可能需要进行调整。

你也可以使用工具 iOSbackup 来轻松地从密码加密的 iOS 备份中读取和提取文件。

6.3.7.3.1. 概念证明：用篡改过的备份解除 UI 锁

如前所述，敏感数据不仅限于用户数据和 PII。它也可以是影响应用程序行为、限制功能或启用安全控制的配置或设置文件。如果你看一下开源的比特币钱包应用 Bither，你会发现有可能配置一个 PIN 码来锁定用户界面。而在几个简单的步骤之后，你会看到如何在一个非越狱的设备上用修改过的备份绕过这个 UI 锁。



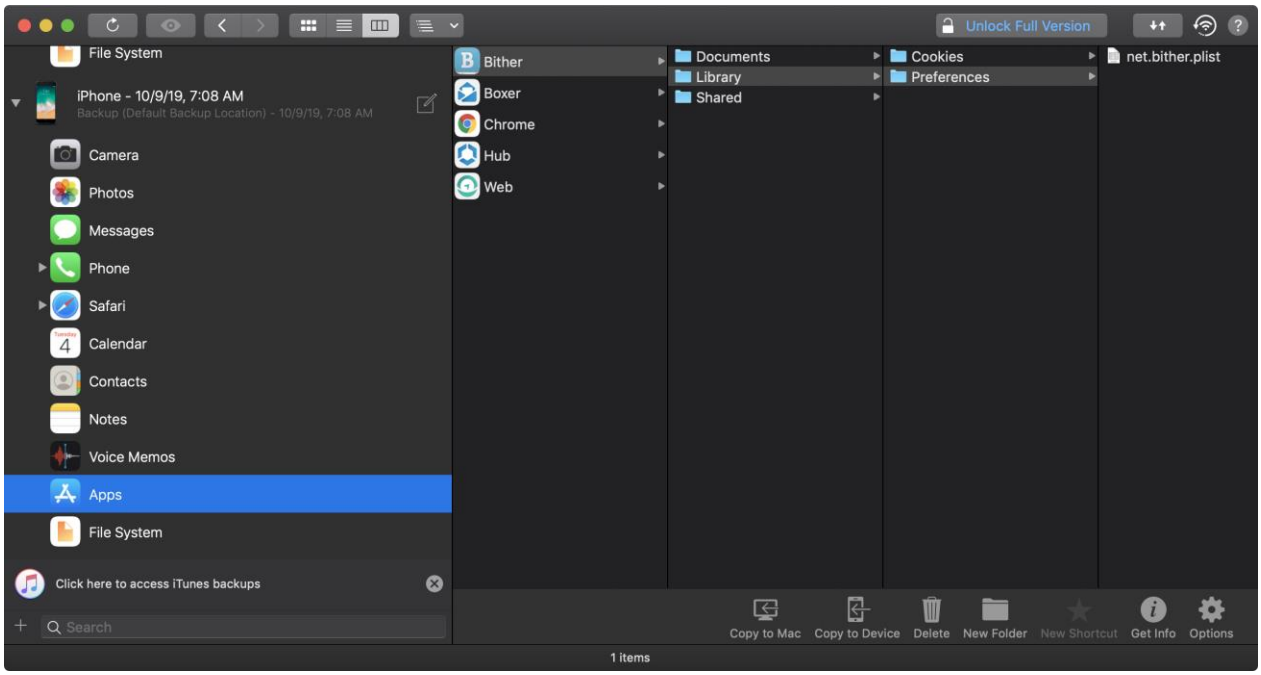
在你启用 PIN 后，使用 iMazing 来执行设备备份。

1. 从 " **AVAILABLE** 可用 "菜单下的列表中选择你的设备。
2. 点击顶部的菜单选项 " **Back Up** 备份"。
3. 按照提示，使用默认值完成备份。

接下来你可以打开备份，查看目标应用内的应用容器文件。

1. 选择你的设备，点击右上角菜单上的 **Backups** 备份。
2. 点击你创建的备份，选择 **View** 查看。
3. 从应用程序目录中浏览到 Bither 应用程序。

在这一点上，你可以查看 Bither 的所有备份内容。



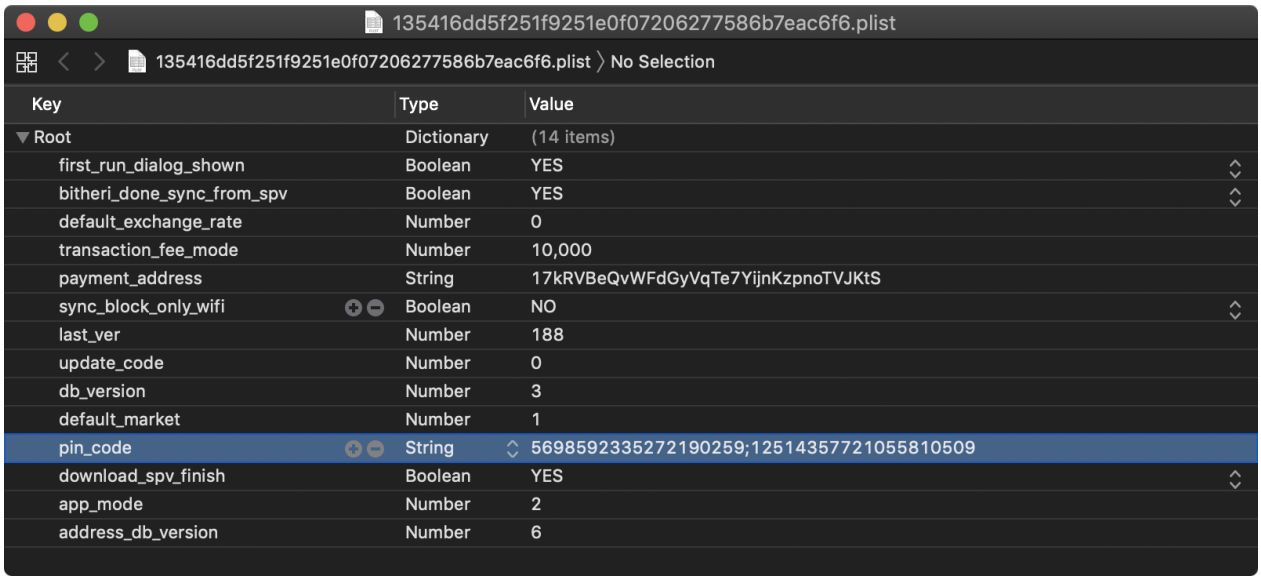
在这里你可以开始解析文件，寻找敏感数据。在截图中，你会看到包含 **pin_code** 属性的 **net.bither.plist** 文件。要取消 UI 锁的限制，只需删除 pin_code 属性并保存更改。

可以使用 iMazing 的授权版本轻松地将 net.bither.plist 的修改版本恢复到设备上。

然而，免费的解决方法是在 iTunes/Finder 生成的混淆的备份中找到 plist 文件。所以，在配置了 Bither 的 PIN 码的情况下创建你的设备备份。然后，使用前面描述的步骤，找到备份目录，grep 查找 "pin_code"，如下所示。

```
$ ~/Library/Application Support/MobileSync/Backup/<UDID>
$ grep -iRn "pin_code" .
Binary file ./13/135416dd5f251f9251e0f07206277586b7eac6f6 matches
```

你会看到在一个混淆了名称的二进制文件上有一个匹配。这就是你的 **net.bither.plist** 文件。继续并重命名该文件，将它扩展名设置为 plist，这样 Xcode 就可以很容易的为你打开它。



同样，从 plist 中删除 **pin_code** 属性并保存你的修改。把文件重新命名为原来的名字（即没有 plist 扩展名），然后进行备份恢复。恢复完成后，你会看到 Bither 在启动时不再提示你输入 PIN 码。

6.3.8. 测试自动生成截图中敏感信息（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。

如果应用程序在进入后台后通过屏幕截图泄露任何敏感信息，该测试案例将失败。

6.3.8.2. 静态分析

如果你有源代码，可以搜索 applicationDidEnterBackground 方法，以确定应用程序是否在进入后台之前对屏幕进行清理。

下面是一个示例实现，每当应用程序被设置为背景时，使用一个默认的背景图片（overlayImage.png），覆盖当前视图。

Swift:

```
private var backgroundImage: UIImageView?

func applicationDidEnterBackground(_ application: UIApplication) {
    let myBanner = UIImageView(image: #imageLiteral(resourceName: "overlayIma
ge"))
    myBanner.frame = UIScreen.main.bounds
    backgroundImage = myBanner
    window?.addSubview(myBanner)
}

func applicationWillEnterForeground(_ application: UIApplication) {
    backgroundImage?.removeFromSuperview()
}
```

Objective-C:

```
@property (UIImageView *)backgroundImage;

- (void)applicationDidEnterBackground:(UIApplication *)application {
    UIImageView *myBanner = [[UIImageView alloc] initWithImage:@"overlayImag
e.png"];
    self.backgroundImage = myBanner;
    self.backgroundImage.bounds = UIScreen.mainScreen.bounds;
    [self.window addSubview:myBanner];
}

- (void)applicationWillEnterForeground:(UIApplication *)application {
    [self.backgroundImage removeFromSuperview];
}
```

每当应用程序进入后台时，就把背景图片设置为 overlayImage.png。它可以防止敏感数据的泄露，因为 overlayImage.png 将始终覆盖当前视图。

6.3.8.3. 动态分析

你可以使用可视化的方法，使用任何 iOS 设备（无论是否越狱）快速验证这个测试案例。

1. 浏览到一个显示敏感信息的应用屏幕，如用户名、电子邮件地址或账户详情。
2. 通过点击 iOS 设备上的 "Home "按钮，将应用程序置于后台。
3. 验证默认图像是否显示为顶层视图元素，而不是包含敏感信息的视图。

如果需要，你也可以通过在越狱的设备上执行步骤 1 到 3，或者在用 Frida Gadget 重新打包应用程序后在非越狱的设备上收集证据。之后，通过 SSH 或其他方式连接到 iOS 设备，并浏览到快照目录。每个 iOS 版本的位置可能不同，但通常在应用程序的库目录内。在 iOS 14.5 上：

/var/mobile/Containers/Data/Application/$APP_ID/Library/SplashBoard/Snapshots/sceneID:$APP_NAME-default/。

该文件夹内的屏幕截图不应包含任何敏感信息。

6.3.9. 测试内存中敏感数据（MSTG-STORAGE-10）

6.3.9.1. 概述

分析内存可以帮助开发人员确定例如应用程序崩溃问题的根本原因。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。目标是确保尽可能简短地暴露这些信息。

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蛛丝马迹（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，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，从而防止通过内存泄露。

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，您需要基于基本和可变的数据结构中集中处理敏感数据。

这样的数据结构使开发人员可以直接访问内存。要确保这种访问是用来用零来覆盖敏感数据和加密密钥的。苹果安全编码指南建议在使用后将敏感数据清零，但没有提供推荐的方法。

首选的数据类型的示例包括 **char[]**和 **int[]**，但不是 **NSString** 或 **String**。每当你试图修改一个不可变的对象，如 **String**，你实际上是创建了一个副本并改变了这个副本。考虑使用 NSMutableData 来存储 Swift/Objective-C 上的机密，并使用 resetBytes(in:)方法来清零。另外，请看清理机密数据的内存，以供参考。

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。 虽然这允许修改分配给 **char** 和 **int** 等简单类型的内存，但按值处理 **String** 等复杂类型涉及到一层隐藏的对象、结构或基本数组，其内存不能被直接访问或修改。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变化（替换，而不是就地修改）：

```
var str1 = "Goodbye"               // "Goodbye", base address:            0x00
01039e8dd0
str1.append(" ")                   // "Goodbye ", base address:           0x60
8000064ae0
str1.append("cruel world!")        // "Goodbye cruel world", base address: 0x60
80000338a0
str1.removeAll()                   // "", base address                    0x000
10bd66180
```

请注意，底层数值的基址随着每一次字符串操作而改变。问题就在这里：为了安全地从内存中清除敏感信息，我们不想简单地改变变量的值；我们想改变为当前值分配的内存的实际内容。Swift 并没有提供这样的功能。

另一方面，如果 Swift 集合（数组 Array、集合 Set 和字典 Dictionary）收集了 **char** 或 **int** 之类的基本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，都使用基本数据类型。

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 Apple 也没有在官方的 iOS SDK API 中考虑这个问题。例如：大多数用于数据传递的 API（传递器，序列化等）都在非基本数据类型上进行操作。类似地，不管您是否将某个 **`UITextField`** 标记为 *Secure Text Entry* 安全文本条目，它始终是以 **`String`** 或 **`NSString`** 的形式返回数据。

总之，在对内存暴露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`**）传递数据。
- 避免使用非基本数据类型（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 可以很好地显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动态测试 iOS 应用程序的内存中的敏感数据。

6.3.9.3.1. 获取和分析内存转储

无论你使用的是已越狱还是未越狱的设备，你都可以通过 Objection 和 Fridump 来转储应用程序的进程内存。你可以在 "iOS 系统上的篡改和逆向工程 "一章的 "内存转储 "一节中找到这个过程的详细解释。

在内存被转储后（例如转储到一个名为 "内存 "的文件），根据你所寻找的数据的性质，你需要一套不同的工具来处理和分析内存转储。例如，如果你专注于字符串，你可能只需要执行 **strings** 或 **rabin2 -zz** 命令来提取这些字符串。

```
# using strings
$ strings memory > strings.txt

# using rabin2
$ rabin2 -ZZ memory > strings.txt
```

在你喜欢的编辑器中打开 strings.txt，并通过它来识别敏感信息。

然而，如果你想检查其他类型的数据，你会更想使用 radare2 及其搜索功能。更多信息和参数列表请参考 radare2 的搜索命令帮助（/? ）下面显示的只是其中的一个子集：

```
$ r2 <name_of_your_dump_file>

[0x00000000]> /?
Usage: /[!bf] [arg]  Search stuff (see 'e??search' for options)
|Use io.va for searching in non virtual addressing spaces
| / foo\x00                     search for string 'foo\0'
| /c[ar]                        search for crypto materials
| /e /E.F/i                     match regular expression
| /i foo                        search for string 'foo' ignoring case
| /m[?][ebm] magicfile          search for magic, filesystems or binary header
s
| /v[1248] value                look for an `cfg.bigendian` 32bit value
| /w foo                        search for wide string 'f\0o\0o\0'
| /x ff0033                     search for hex string
| /z min max                    search for strings of given size
...
```

6.3.9.3.2. 运行时内存分析

通过使用 r2frida，你可以在运行中分析和检查应用程序的内存，而不需要转储。例如，你可以从 r2frida 运行之前的搜索命令，并在内存中搜索一个字符串、十六进制值等。这样做的时候，记得在用 **r2 frida://usb//<name_of_your_app>**启动会话后，在搜索命令（以及其他任何 r2frida 特定的命令）前加一个反斜杠\。

更多信息、参数和方法，请参考 "iOS 系统上的篡改和逆向工程 "一章中的 "内存中搜索 "部分。

6.3.10. 参考文献

- [#mandt] Tarjei Mandt, Mathew Solnik and David Wang, Demystifying the Secure Enclave Processor - https://www.blackhat.com/docs/us-16/materials/us-16-Mandt-Demystifying-The-Secure-Enclave-Processor.pdf

6.3.10.1. OWASP MASVS

- MSTG-STORAGE-1：“系统凭证存储设施需要用于存储敏感数据，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间，并且在使用后会明确清除内存。”

6.4. iOS 加密 API

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 iOS 的加密 API 的细节。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。在审查代码时，请将使用的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. 验证标准加密算法的配置（MSTG-CRYPTO-2 和 MSTG-CRYPTO-3）

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. CryptoKit

Apple CryptoKit 与 iOS 13 一起发布，建立在苹果原生加密库 `corecrypto` 之上。Swift 框架提供了一个强类型的 API 接口，具有有效的内存管理，符合 equatable，并支持泛型。CryptoKit 包含用于散列、对称密钥密码学和公钥密码学的安全算法。该框架还可以利用安全隔区（Secure Enclave）的基于硬件的密钥管理器。

苹果 CryptoKit 包含以下算法。

哈希值：

- MD5（不安全模块）
- SHA1（不安全模块）
- SHA-2 256 位摘要
- SHA-2 384 位摘要
- SHA-2 512 位摘要

对称密钥：

- 信息验证码（HMAC）
- 验证加密
  - AES-GCM
  - ChaCha20-Poly1305

公钥：

- 密钥协议
  - Curve25519
  - NIST P-256

- NIST P-384
- NIST P-512

示例：

生成和发布对称密钥：

```
let encryptionKey = SymmetricKey(size: .bits256)
```

计算一个 SHA-2 512 位摘要：

```
let rawString = "OWASP MTSG"
let rawData = Data(rawString.utf8)
let hash = SHA512.hash(data: rawData) // Compute the digest
let textHash = String(describing: hash)
print(textHash) // Print hash text
```

关于 Apple CryptoKit 的更多信息，请访问以下资源：

- Apple CryptoKit | Apple 开发者文档
- 执行常见的加密操作 | Apple 开发者文档
- WWDC 2019 session 709 | 密码学和你的应用程序
- 如何计算字符串或数据实例的 SHA 哈希值 | Swift 笔记

6.4.1.1.2. CommonCrypto, Encrypt 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加密密钥的函数。

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如何使用它的说明。

如前所述：为了方便起见，两者都存在一些包装器库。例如：使用的典型库有：

- IDZSwiftCommonCrypto
- Heimdall
- SwiftyRSA
- RNCryptor
- Arcane

6.4.1.1.3. 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 缺乏广泛支持，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、对称加密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、对称和非对称加密、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**：Google 的新密码库。Google 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 源中找到。
- **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，建议查找受支持和维护中的库。
- **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“移动应用的加密”一节中，关于废弃的算法和密码学配置已经说了很多。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，一个暂时存在的结构）。

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 CCCryptorCreate。源代码包含 CommonCryptor.h 的所有函数的特征。例如：CCCryptorCreate 具有以下特征：

```
CCCryptorStatus CCCryptorCreate(
    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。请注意，在 "移动应用程序的密码学 "一章中指出的被废弃的函数，仍以编程方式支持。它们不应该被使用。

6.4.1.2.2. 第三方库

考虑到所有第三方库的不断发展，因此不应该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 Carthage，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 “**Link Binary With Libraries** 将二进制文件链接到库”中的任何库。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陷的新版本可用。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。
- **是否自行实现?**：我们建议不要使用自己的加密，也不要自己实现已知的加密函数。

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没有密钥材料可以被转储。这可以通过使用密码密钥派生函数（如 PKBDF-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: passwo
rd, salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: In
t) -> Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password: pass
word, salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: In
t) -> Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password: pass
word, salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2(hash: CCPBKDFAlgorithm, password: String, salt: Data, keyByteCoun
```

```
t: 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 { derivedKey
Bytes in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
                UInt32(rounds),
                derivedKeyBytes, derivedKeyDataLength
            )
        }
    }
    if derivationStatus != 0 {
        // Error
        return nil
    }

    return derivedKeyData
}

func testKeyDerivation() {
    let password = "password"
    let salt = Data([0x73, 0x61, 0x6C, 0x74, 0x44, 0x61, 0x74, 0x61])
    let keyByteCount = 16
    let rounds = 100_000

    let derivedKey = pbkdf2SHA1(password: password, salt: salt, keyByteCount:
 keyByteCount, rounds: rounds)
}
```

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、Core Data 或 Realm 中的任何其他接收器，通常不如使用 KeyChain 安全。即使使用 **NSFileProtectionComplete** 数据保护类保护 Core Data 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“iOS 上数据存储”部分。

KeyChain 支持两种类型的存储机制：密钥由存储在安全隔区中的加密密钥来保护，或者密钥本身在安全隔区中。后者只在你使用 ECDH 签名密钥时成立。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，无论是以原生存储还是以 Objective-C/Swift 格式存储。因此，攻击者可以解密任何其他被应用程序加密的数据。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“验证标准加密算法的配置”一节的概述和静态分析中提到的库，看看哪些关键词可以最好地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。
- 密钥不是使用低级语言（例如：C/C++）隐藏的。
- 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“iOS 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

劫持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

Apple 提供了一个随机化服务 API，该 API 可以生成密码学安全的随机数。

随机化服务 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 的密码学安全伪随机值。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：如果代码中使用了其他的随机数机制，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随机性。通常这太难了，这意味着最好还是坚持使用上面的实现。

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. OWASP MASVS

- MSTG-CRYPTO-1：“应用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：“应用使用经验证的加密原语实现。”
- MSTG-CRYPTO-3：“应用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：“应用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。”

6.4.4.2. 通用的安全文档

- 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

6.4.4.3. 密码算法的配置

- Apple 加密服务指南 - https://developer.apple.com/library/content/documentation/Security/Conceptual/cryptoser vices/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
- 关于安全隔区的 Apple 文档 - https://developer.apple.com/documentation/security/certificate_key_and_trust_services/key s/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/CommonCryptohttps://opensource.apple.com/source/CommonCrypto/CommonCrypto-60074/include/CommonCryptorSPI.h60074/include/CommonCryptorSPI.h
- 关于 SecKey 的 Apple 开发者文档 - https://opensource.apple.com/source/Security/Securityhttps://opensource.apple.com/source/Security/Security-57740.51.3/keychain/SecKey.h.auto.html57740.51.3/keychain/SecKey.h.auto.html
- IDZSwiftCommonCrypto - https://github.com/iOSdevzone/IDZSwiftCommonCrypto
- Heimdall - https://github.com/henrinormak/Heimdall
- SwiftyRSA - https://github.com/TakeScoop/SwiftyRSA
- 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-cryptographichttps://security.googleblog.com/2018/08/introducing-tink-cryptographic-software.htmlsoftware.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

6.4.4.4. 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

6.4.4.5. 密钥管理

- Apple 开发者文档：证书和密钥 - https://developer.apple.com/documentation/security/certificate_key_and_trust_services/key s
- Apple 开发者文档：生成新密钥- https://developer.apple.com/documentation/security/certificate_key_and_trust_services/key s/generating_new_cryptographic_keys
- Apple 开发者文档：密钥生成属性- https://developer.apple.com/documentation/security/certificate_key_and_trust_services/key s/key_generation_attributes

6.5. iOS 本地身份认证

在本地身份认证期间，应用程序根据存储在设备上的凭据对用户进行身份认证。换言之，用户通过提供有效的 PIN、密码或生物特征（如面部或指纹），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认证的手段来保护某些关键功能。

正如前面在“移动应用的认证架构”章节中所述：测试人员应该知道，本地身份认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. 测试本地身份认证（MSTG-AUTH-8 和 MSTG-STORAGE-11）

在 iOS 上，可以使用多种方法将本地身份认证集成到应用程序中。本地身份认证框架为开发人员提供了一组 API，该 API 用于将身份认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 Keychain 来实现本地身份认证。

iOS 上的指纹身份认证称为 Touch ID。指纹 ID 传感器由安全协处理器操作，不会将指纹数据暴露给系统的任何其他部分。在 Touch ID 之后，Apple 引入了 Face ID：它提供了基于面部识别的身份认证。两者在应用程序级别上使用相似的 API，但存储和检索数据（如面部数据或指纹相关数据）的实际方法是不同的。。

开发人员有两个选项可用于合并 Touch ID/Face ID 身份认证：

- LocalAuthentication.framework 是一个高级 API，用于通过 Touch ID 对用户进行身份认证。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认证是否成功。
- 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

本地身份认证框架提供了向用户请求密码或 Touch ID 进行身份认证的功能。开发人员可以通过使用 **`LAContext`** 类的 **`evaluatePolicy`** 函数，来显示和利用身份认证提示。

以下两个策略定义了可接受的身份认证形式：

- **`deviceOwnerAuthentication`** (Swift) 或 **`LAPolicyDeviceOwnerAuthentication`**（Objective-C）：可用时，提示用户使用 Touch ID 进行身份认证。若 Touch ID 未激活，则会请求设备密码。如果设备密码没有启用，则策略评估失败。。
- **`deviceOwnerAuthenticationWithBiometrics`** (Swift) 或 **`LAPolicyDeviceOwnerAuthenticationWithBiometrics`** (Objective-C)：认证仅限于生物识别技术，用户会被提示使用 Touch ID。

`evaluatePolicy` 函数返回一个布尔值，指示用户是否已成功通过身份认证。

Apple 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) el
se {
    // 无法评估政策；查看错误并向用户提供一个适当的信息
}

context.evaluatePolicy(.deviceOwnerAuthentication, localizedReason: "Please,
pass authorization to enter this area") { success, evaluationError in
    guard success else {
        // 用户没有认证成功，查看 evaluationError 并采取适当行动
    }

    // 用户认证成功，采取适当行动
}
```

使用本地身份认证框架在 Swift 中进行 Touch-ID 身份认证（Apple 官方代码示例）。

6.5.1.2. 使用 Keychain 服务进行本地身份认证

iOS Keychain API 可以（并且应该）用于实现本地身份认证。在此过程中，应用程序会在 Keychain 中存储一个秘密身份认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认证，用户必须使用其密码或指纹来解锁 Keychain，以获取机密数据。

Keychain 允许使用特殊的 **`SecAccessControl`** 属性保存项目，只有在用户通过 Touch ID 身份认证（或密码验证，如果属性参数允许这种回退）之后，才允许从 Keychain 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 身份认证（**SecAccessControlCreateFlags.biometryCurrentSet** 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. 创建 AccessControl 对象，代表认证设置

var error: Unmanaged<CFError>?

guard let accessControl = SecAccessControlCreateWithFlags(kCFAllocatorDefaul
t,
                                                           kSecAttrAccessibleW
henPasscodeSetThisDeviceOnly,
                                                           SecAccessControlCre
ateFlags.biometryCurrentSet,
                                                           &error) else {
    // 创建 AccessControl 对象失败

    return
}

// 2. 创建 keychain 服务查询。请注意，kSecAttrAccessControl 与 kSecAttrAccessible
属性是相互排斥的。

var query: [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)! a
s CFData
query[kSecAttrAccessControl as String] = accessControl

// 3. 保存项目

let status = SecItemAdd(query as CFDictionary, nil)

if status == noErr {
    // 保存成功
} else {
    // 保存时出现错误
}
```

// 4. 现在我们可以从 Keychain 中请求保存的项目。Keychain 服务将向用户显示身份认证对话框，并根据是否提供了合适的指纹来返回数据或空。

```
// 5. 定义查询
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 ente
r this area" as CFString

// 2. 获得项目
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)!
    // 成功获取密码
} else {
    // 认证未通过
}
```

6.5.1.2.4. Objective-C

```
    // 1. 创建 AccessControl 对象，代表认证设置
    CFErrorRef *err = nil;

    SecAccessControlRef sacRef = SecAccessControlCreateWithFlags(kCFAllocator
Default,
        kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
        kSecAccessControlUserPresence,
        err);

    // 2. 定义 keychain 服务查询。请注意，kSecAttrAccessControl 与 kSecAttrAccess
ible 属性是相互排斥的
    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" dataUsingEncodin
g:NSUTF8StringEncoding],
        (__bridge id)kSecAttrAccessControl: (__bridge_transfer id)sacRef
    };
```

```
    // 3. 保存项目
    OSStatus status = SecItemAdd((__bridge CFDictionaryRef)query, nil);

    if (status == noErr) {
        // 保存成功
    } else {
        // 保存时出现错误
    }

//4. 现在我们可以从 Keychain 中请求保存的项目。Keychain 服务将向用户展示认证对话框，
并根据是否提供了合适的指纹，返回数据或空。

// 5. 定义查询
NSDictionary *query = @{(__bridge id)kSecClass: (__bridge id)kSecClassGeneric
Password,
    (__bridge id)kSecReturnData: @YES,
    (__bridge id)kSecAttrAccount: @"My Name1",
    (__bridge id)kSecAttrLabel: @"com.me.myapp.password",
    (__bridge id)kSecUseOperationPrompt: @"Please, pass authorisation to ente
r this area" };

// 2. 获得项目
CFTypeRef queryResult = NULL;
OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)query, &query
Result);

if (status == noErr){
    NSData* resultData = ( __bridge_transfer NSData* )queryResult;
    NSString* password = [[NSString alloc] initWithData:resultData encoding:N
SUTF8StringEncoding];
    NSLog(@"%@", password);
} else {
    NSLog(@"Something went wrong");
}
```

应用程序中框架的使用也可以通过分析应用程序二进制的共享动态库列表来检测。这可以通过使用 otool 来完成。

```
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
```

如果使用了 **Security.framework**，则仅显示第二个。

6.5.1.3. 静态分析

重要的是要记住，本地身份认证框架是基于事件的程序，因此，它不应该是唯一的身份认证方法。尽管这种身份认证在用户界面级别上是有效的，但很容易通过修补或插桩来绕过它。因此，最好是使用 keychain 服务方法，这意味着你应该：

- 使用 Keychain 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认证）是否受到保护。
- 确认为 Keychain 项目设置了访问控制标志，确保 Keychain 项目的数据只能通过验证用户的方式解锁。这可以通过以下标志之一来实现：
    - kSecAccessControlBiometryCurrentSet（iOS 11.3 之前为 kSecAccessControlTouchIDCurrentSet）。这将确保用户在访问 Keychain 项目中的数据之前需要用生物识别技术（如 Face ID 或 Touch ID）进行认证。每当用户在设备上添加指纹或面部代表时，它将自动使 Keychain 中的条目失效。这确保了 Keychain 项目永远只能由在该项目被添加到 Keychain 时注册的用户解锁。
    - kSecAccessControlBiometryAny（iOS 11.3 之前为 kSecAccessControlTouchIDAny）。这将确保用户在访问 Keychain 条目中的数据之前，需要用生物识别技术（例如 Face ID 或 Touch ID）进行认证。Keychain 条目将在任何（重新）注册新的指纹或面部代表的情况下继续存在。如果用户有一个不断变化的指纹，这可能是非常方便的。然而，这也意味着攻击者，如果以某种方式能够在设备上注册他们的指纹或面部，现在也可以访问这些条目。
    - kSecAccessControlUserPresence 可以作为一个替代方案。如果生物识别认证不再有效，这将允许用户通过密码进行认证。这被认为比 kSecAccessControlBiometryAny 更弱，因为通过肩窥的方式窃取别人的密码输入，比绕过 Touch ID 或 Face ID 服务要容易得多。
- 为了确保可以使用生物识别技术，在调用 SecAccessControlCreateWithFlags 方法时，验证 kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly 或 kSecAttrAccessibleWhenPasscodeSet 保护类被设置。注意 ...ThisDeviceOnly 变体将确保 Keychain 项目不与其他 iOS 设备同步。

注意，一个数据保护类别指定了用于保护数据的访问方法。每个类别使用不同的策略来确定数据何时可以被访问。

6.5.1.4. 动态分析

Objection 生物识别绕过可以用来绕过本地身份认证。Objection 使用 Frida 来插桩 **evaluatePolicy** 函数，这样即使认证没有成功执行，它也会返回 True。使用 **ios ui biometrics_bypass** 命令来绕过不安全的生物识别认证。Objection 将注册一个作业，它将取代 **evaluatePolicy** 的结果。它在 Swift 和 Objective-C 的实现中都可以工作。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios ui biometri
cs_bypass
(agent) Registering job 3mhtws9x47q. Type: ios-biometrics-disable
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # (agent) [3mhtws
9x47q] Localized Reason for auth requirement: Please authenticate yourself
(agent) [3mhtws9x47q] OS authentication response: false
(agent) [3mhtws9x47q] Marking OS response as True instead
(agent) [3mhtws9x47q] Biometrics bypass hook complete
```

如果有漏洞，该模块将自动绕过登录表单。

6.5.2. 关于 Keychain 中密钥的临时性的说明

与 MacOSX 和 Android 不同，iOS 目前（iOS 12）不支持在 Keychain 中临时访问条目：当进入 Keychain 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. 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。”

6.6. iOS 网络通信

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，因此经典的基于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使用基于 HTTP 的网络服务的变体，因为这些协议有很好的文档和支持。

6.6.1. 概述

6.6.1.1. iOS 应用传输安全

从 iOS 9 开始，Apple 引入了应用传输安全（ATS），这是一套由操作系统强制执行的安全检查，用于使用 URL 加载系统（通常通过 URLSession）进行的连接，以始终使用 HTTPS。应用程序应遵循 Apple 公司的最佳实践，以正确确保其连接安全。

> 观看 Apple WWDC 2015 的 ATS 介绍视频。

ATS 执行默认的服务器信任评估，并要求一组最低的安全要求。

默认的服务器信任评估:

当一个应用程序连接到一个远程服务器时，服务器使用 X.509 数字证书提供其身份。ATS 默认的服务器信任评估包括验证该证书:

- 没有过期。
- 有一个与服务器的 DNS 名称相匹配的名称。
- 有一个有效的数字签名（没有被篡改过），并且可以追溯到一个受信任的认证机构（CA），包括在操作系统的信任库中，或者由用户或系统管理员安装在客户端。

连接的最低安全要求:

ATS 将阻止那些进一步不符合最低安全要求的连接，包括：：

- TLS 版本 1.2 或更高。
- 使用 AES-128 或 AES-256 的数据加密。
- 该证书必须用 RSA 密钥（2048 位或更高）或 ECC 密钥（256 位或更高）签署。
- 该证书的指纹必须使用 SHA-256 或更高。
- 该链接必须通过椭圆曲线 Diffie-Hellman Ephemeral（ECDHE）密钥交换支持完美前向保密（PFS）。

证书的有效性检查:

根据 Apple 公司的说法，"评估 TLS 证书的信任状态是按照 RFC 5280 中规定的既定行业标准进行的，并纳入了 RFC 6962（证书透明度）等新兴标准。在 iOS 11 或更高版本中，苹果设备会定期更新已撤销和受限的认证的最新列表。该列表是由认证撤销列表（CRL）汇总而成，该列表由 Apple 公司信任的每个内置根认证机构以及其下属的 CA 发行人发布。该列表还可能包括其他由 Apple 公司决定的限制。每当使用网络 API 功能进行安全连接时，都会查询该信息。如果一个 CA 的废止证书太多，无法单独列出，信任评估可能会要求需要在线证书状态响应（OCSP），如果响应不可用，信任评估就会失败。"

6.6.1.1.1. 什么时候不使用 ATS

- **当使用较低级别的 API 时**：ATS 只适用于 URL 加载系统，包括 URLSession 和它们上面的分层的 API。它不适用于使用较低级别的 API（如 BSD 套接字）的应用程序，包括那些在这些较低级别的 API 之上实现 TLS 的应用程序（见存档的 Apple 开发者文档中的 "在 Apple 框架中使用 ATS"部分）。
- **当连接到 IP 地址、未标明的域名或本地主机时**：ATS 只适用于与公共主机名的连接（见存档的 Apple 开发者文档中的 "ATS 对远程和本地连接的可用性"一节）。系统不为连接到以下地点的连接提供 ATS 保护。
  - 互联网协议（IP）地址
  - 不合格的主机名称
  - 采用.local 顶级域名（TLD）的本地主机
- **当包括 ATS 例外的时候**：如果应用程序使用 ATS 兼容的 API，它仍然可以使用 ATS 例外来禁用 ATS 的特定场景。

了解更多:

- "ATS 和私有网络 iOS 企业应用"
- "ATS 和本地 IP 地址"
- "ATS 对使用第三方库的应用程序的影响"
- "ATS 和 SSL 固定/自有 CA"

6.6.1.1.2. 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
- 禁用完美前向保密（PFS）
- 允许连接到本地域名

ATS 例外可以全局应用，也可以按域名应用。应用程序可以全局禁用 ATS，也可以选择单个域名。以下来自 Apple 开发者文档的列表显示了 **NSAppTransportSecurity** 的结构

```
NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSExceptionAllowsInsecureHTTPLoads : Boolean
            NSExceptionMinimumTLSVersion : String
            NSExceptionRequiresForwardSecrecy : Boolean    //默认值为 YES
            NSRequiresCertificateTransparency : Boolean
        }
    }
}
```

来源：Apple 开发者文档。

下表总结了全局 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。

键	说明
NSAllowsArbitraryLoads	全局禁用 ATS 限制，但 **NSExceptionDomains** 下指定的单个域名除外。
NSAllowsArbitraryLoadsInWebContent	对从 web 视图建立的所有连接禁用 ATS 限制。

NSAllowsLocalNetworking	允许连接到不合规的域名和.local 域名。
NSAllowsArbitraryLoadsForMedia	对通过 AV 基础框架加载的媒体禁用所有 ATS 限制。

下表总结了每个域名的 ATS 例外情况。有关这些例外的更多信息，请参考 Apple 官方开发人员文档中的表 3。

键	说明
NSIncludesSubdomains	指示 ATS 例外是否应应用于指定域的子域。
NSExceptionAllowsInsecureHTTPLoads	允许通过 HTTP 连接到指定域，但不影响 TLS 要求。
NSExceptionMinimumTLSVersion	允许连接到 TLS 版本低于 1.2 的服务器。
NSExceptionRequiresForwardSecrecy	禁用完全前向保密（PFS）。

证明例外情况的合理性：

从 2017 年 1 月 1 日开始，如果定义了以下 ATS 例外之一，则需要提供证明理由来通过 Apple App Store 审查。

- NSAllowsArbitraryLoads
- NSAllowsArbitraryLoadsForMedia
- NSAllowsArbitraryLoadsInWebContent
- NSExceptionAllowsInsecureHTTPLoads
- NSExceptionMinimumTLSVersion

这必须仔细审查，以确定它是否确实是应用程序预期目的的一部分。Apple 警告说，例外情况会降低应用程序的安全性，并建议**只在需要时才配置例外情况**，在面临 ATS 失败时，**最好选择服务器修复。**

示例：

在下面的例子中，ATS 是全局启用的（没有定义全局的 `NSAllowsArbitraryLoads`），但为 example.com 域（及其子域）明确设置了一个例外。考虑到这个域名是由应用程序开发人员拥有的，并且有适当的理由，这个例外是可以接受的，因为它保持了 ATS 对所有其他域名的保护。然而，最好还是按照上面所说的那样对服务器进行修复。

```
   <key>NSAppTransportSecurity</key>
<dict>
   <key>NSExceptionDomains</key>
  <dict>
    <key>example.com</key>
    <dict>
      <key>NSIncludesSubdomains</key>
      <true/>
      <key>NSExceptionMinimumTLSVersion</key>
      <string>TLSv1.2</string>
      <key>NSExceptionAllowsInsecureHTTPLoads</key>
      <true/>
      <key>NSExceptionRequiresForwardSecrecy</key>
      <true/>
    </dict>
  </dict>
</dict>
```

关于 ATS 异常的更多信息，请参考 Apple 开发者文档中 "防止不安全的网络连接 "一文中的 "只在需要时才配置异常；优先修复服务器"一节以及关于 ATS 的博文。

6.6.1.2. iOS 网络 API

从 iOS 12.0 开始，网络框架和 URLSession 类提供了异步和同步加载网络和 URL 请求的方法。旧的 iOS 版本可以利用 Sockets API。

6.6.1.2.1. 网络框架

网络框架是在 2018 年的 Apple 全球开发者大会（WWDC）上推出的，是 Sockets API 的替代品。这个低级别的网络框架提供了发送和接收数据的类，具有内置的动态网络、安全和性能支持。

在网络框架中，如果使用参数：`.tls`，则默认启用 TLS 1.3。与传统的安全传输框架相比，它是首选方案。

6.6.1.2.2. URLSession

URLSession 是建立在网络框架之上的，并且利用了相同的传输服务。如果端点是 HTTPS，该类也默认使用 TLS 1.3。

`URLSession` **应该被用于 HTTP 和 HTTPS 连接，而不是直接使用网络框架。** `URLSession` 类原生支持两种 URL 方案，并为这种连接进行了优化。它需要更少的模板代码，减少出错的可能性，并确保默认情况下的安全连接。网络框架应该只在有低级和/或高级网络要求时使用。

官方 Apple 文档包括使用网络框架来实现 netcat 和 URLSession 以获取网站数据到内存的示例。

6.6.2. 测试网络上的数据加密（MSTG-NETWORK-1）

所有展示的案例都必须作为一个整体进行仔细分析。例如，即使应用程序在其 Info.plist 中不允许明文流量，它实际上可能仍然在发送 HTTP 流量。如果它使用的是低级别的 API（ATS 被忽略），或者是一个配置错误的跨平台框架，就可能出现这种情况。

> 重要提示：你应该将这些测试应用于应用程序的主代码，同时也应用于任何应用程序的扩展、框架或内置于应用程序的手表应用。

更多信息请参考 Apple 开发者文档中的文章 "防止不安全的网络连接"和"微调你的应用程序传输安全设置"。

6.6.2.1. 静态分析

6.6.2.1.1. 测试基于安全协议的网络请求

首先，你应该在源代码中识别所有的网络请求，确保不使用明文 HTTP URL。通过使用 URLSession（使用 iOS 的标准 URL 加载系统）或 Network（套接字级通信使用 TLS 访问 TCP 和 UDP），确保敏感信息通过安全渠道发送。

6.6.2.1.2. 检查低级网络 API 使用

识别应用程序使用的网络 API，看看它是否使用任何低级网络 API。

> Apple 建议。在你的应用程序中优先使用高级框架。"ATS 并不适用于你的应用程序对较低级别的网络接口的调用，如网络框架或 CFNetwork。在这些情况下，你要负责确保连接的安全性。你可以通过这种方式构建一个安全的连接，但是错误很容易发生，而且代价很高。通常最安全的做法是依靠 URL 加载系统"（见来源）。

如果应用程序使用任何低级别的 API，如 Network 或 CFNetwork，你应该仔细调查它们是否被安全地使用。对于使用跨平台框架（如 Flutter、Xamarin.....）和第三方框架（如 `Alamofire`）的应用程序，你应该分析它们是否按照其最佳实践进行了配置和安全使用。

确保该应用程序：

- 在进行服务器信任评估时，验证挑战类型、主机名称和凭证。
- 不忽视 TLS 错误。
- 不使用任何不安全的 TLS 配置（见 "测试 TLS 设置（MSTG-NETWORK-2）"）。

这些检查是指导性的，我们无法确定特定的 API，因为每个应用程序可能使用不同的框架。在检查代码时，请将这些信息作为参考。

6.6.2.1.3. 测试明文流量

确保应用程序不允许明文 HTTP 流量。自 iOS 9.0 以来，明文 HTTP 流量默认被阻止（由于应用传输安全（ATS)），但有多种方法，应用程序仍然可以发送它：

- 通过在应用程序的 `Info.plist` 中设置 `NSAppTransportSecurity` 的 `NSAllowsArbitraryLoads` 属性为 `true`（或 `YES`），使 ATS 启用明文传输。
- 检索 `Info.plist`
- 检查 `NSAllowsArbitraryLoads` 是否在全局或任何域中被设置为真。

- 如果应用程序在 WebView 中打开第三方网站，那么从 iOS 10 开始，`NSAllowsArbitraryLoadsInWebContent` 可用于禁用 ATS 对 WebView 中加载内容的限制。

  Apple 公司警告说。禁用 ATS 意味着允许不安全的 HTTP 连接。HTTPS 连接也是允许的，并且仍然受制于默认的服务器信任评估。但是，扩展的安全检查--如要求最低的传输层安全（TLS）协议版本--被禁用。在没有 ATS 的情况下，你也可以自由地放宽默认的服务器信任要求，如 "执行手动服务器信任认证 "中所述

下面的片段显示了一个易受攻击的示例，即一个应用程序全局禁用 ATS 限制。

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsArbitraryLoads</key>
  <true/>
</dict>
```

审查 ATS 时应考虑到应用的背景。应用程序可能必须确定 ATS 的例外情况以实现其预期目的。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任何不具备所有 ATS 要求的 HTTP 网站。在某些情况下，应用程序可能会在全局范围内禁用 ATS，但在某些域启用 ATS，例如，安全加载元数据或仍然可以安全登录。

ATS 应该包括一个证明字符串（例如："该应用必须连接到由另一个不支持安全连接的实体管理的服务器。"）。

6.6.2.2. 动态分析

拦截被测试的应用程序传入和传出的网络流量，并确保这些流量是加密的。你可以通过以下任何一种方式拦截网络流量。

- 用 OWASP ZAP 或 Burp Suite 等拦截代理捕获所有 HTTP(S)和 Websocket 流量，并确保所有请求是通过 HTTPS 而不是 HTTP 发出的。

- 像 Burp 和 OWASP ZAP 这样的拦截代理将只显示 HTTP(S) 流量。你可以使用 Burp 插件，如 Burp-non-HTTP-Extension 或 Mitm-relay 工具来解码和显示通过 XMPP 和其他协议的通信。

  一些应用程序可能会因为证书固定而无法使用 Burp 和 OWASP ZAP 这样的代理服务器。在这种情况下，请检查 "测试自定义证书存储和证书固定"。

更多细节请参考。

- "测试网络通信 "一章中的 "在网络层拦截流量"。
- "设置网络测试环境"，来自 iOS 基本安全测试一章

6.6.3. 测试 TLS 设置（MSTG-NETWORK-2）

请记住检查相应的证明，以排除它可能是应用程序预期用途的一部分。

在与某一终端通信时，有可能验证哪些 ATS 设置可以使用。在 macOS 上，可以使用命令行工具 `nscurl`。不同设置的排列组合将被执行，并对指定的端点进行验证。如果默认的 ATS 安全连接测试通过，ATS 就可以在其默认的安全配置中使用。如果 `nscurl` 输出有任何失败，请改变服务器端的 TLS 配置，使服务器端更安全，而不是削弱客户端的 ATS 配置。更多细节请参见 Apple 开发者文档中的 "识别阻断连接来源 "一文。

请参阅测试网络通信一章中的 "验证 TLS 设置 "一节以了解详情。

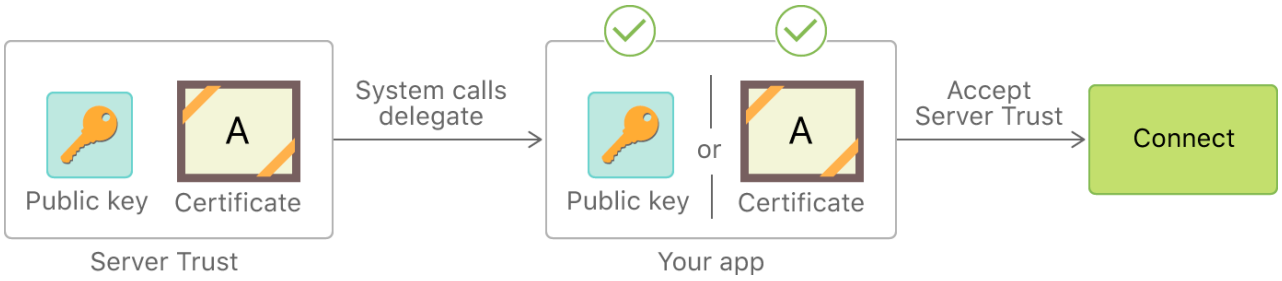
6.6.4. 测试端点身份验证（MSTG-NETWORK-3）

6.6.4.1. 概述

ATS 施加了扩展的安全检查，补充了传输层安全（TLS）协议规定的默认服务器信任评估。你应该测试应用程序是否放松了 ATS 的限制，因为这降低了应用程序的安全性。在增加 ATS 例外之前，应用程序应该优先选择其他方式来提高服务器的安全性。

Apple 开发者文档解释说，应用程序可以使用 `URLSession` 来自动处理服务器信任评估。然而，应用程序也能够定制这一过程，例如，他们可以：

- 绕过或定制证书的有效期。
- 放松/扩展信任：接受否则会被系统拒绝的服务器证书，例如，使用内置在应用程序中的自签名证书与开发服务器建立安全连接。
- 收紧信任：拒绝接受否则会被系统接受的凭证（见 "测试自定义证书存储和证书固定"）。
- 等等。



参考：

- 防止不安全的网络连接
- 执行手动服务器信任认证
- 证书、密钥和信任服务

6.6.4.2. 静态分析

在这一节中，我们介绍了几个静态分析检查。然而，我们强烈建议用动态分析来支持它们。如果你没有源代码，或者应用程序难以进行逆向工程，那么有一个可靠的动态分析策略会有很大的帮助。在这种情况下，你不知道应用程序是使用低级还是高级的 API，但你仍然可以测试不同的信任评估情况（例如，"该应用程序是否接受自签名的证书？）

6.6.4.2.1. 检查操作系统版本

如果应用程序链接到 iOS 9.0 之前的 SDK，则无论应用程序运行在哪个版本的操作系统上，ATS 都将被禁用。

6.6.4.3. 动态分析

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 **Settings** ->常规 **General** ->概要文件 **Profiles**），并且停用 SSL Kill Switch 之类的工具。启动您的应用程序，检查是否有流量经过 Burp。任何错误都将在“警报 Alerts”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，如果您看不到任何流量，并且您有关于 SSL 握手失败的信息，请继续下一项。
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 中看到流量，这意味着证书已经根据设备的信任存储进行了验证，但不执行固定。

如果执行上一步中的指令不会导致通过 burp 代理流量，则可能意味着证书实际上已被固定，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固定。请参阅“绕过证书固定”部分，了解有关此操作的更多信息。

6.6.5. 测试自定义证书的存储和证书固定（MSTG-NETWORK-4）

6.6.5.1. 概述

该测试验证了应用程序是否正确实现了身份固定（证书或公钥固定）。

更多细节请参考 "移动应用网络通信 "一章中的 "身份固定 "部分。

6.6.5.2. 静态分析

验证服务器证书是否已固定。根据服务器提供的证书树，可以在不同级别上实现固定：

1. 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，这就需要一个更新机制。
2. 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。

Apple 公司推荐的最新方法是在应用程序传输安全设置下的 **`Info.plist`** 文件中指定一个固定的 CA 公钥。你可以在他们的文章《身份锁定：如何为你的应用程序配置服务器证书》中找到一个例子。

另一种常见的方法是使用 **`NSURLConnectionDelegate`** 的 **`connection:willSendRequestForAuthenticationChallenge:`**方法来检查服务器提供的证书是否有效并与存储在应用程序中的证书相匹配。你可以在 HTTPS 服务器信任评估技术说明中找到更多细节。

以下的第三方库包括固定功能。

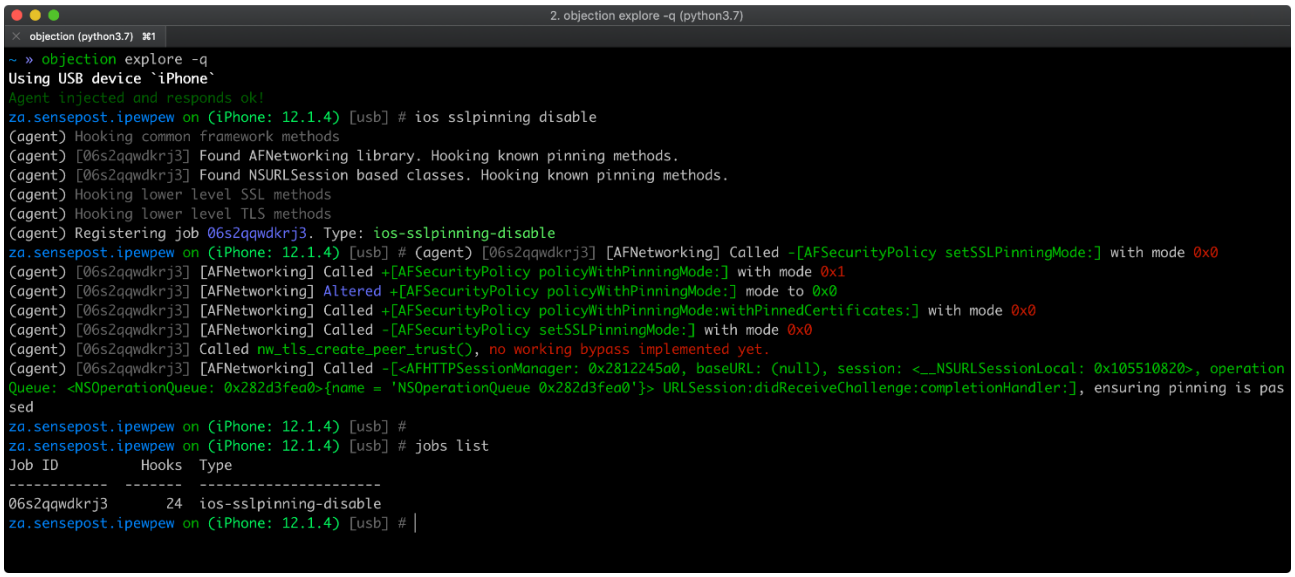
- TrustKit：在这里，你可以通过在你的 Info.plist 中设置公钥的哈希值或在字典中提供哈希值来固定。更多细节请看他们的 readme。
- AlamoFire：在这里你可以为每个域定义一个 **`ServerTrustPolicy`**，你可以为其定义固定方法。
- AFNetworking：在这里你可以设置一个 **`AFSecurityPolicy`** 来配置你的固定。

6.6.5.3. 动态分析

6.6.5.3.1. 服务器证书固定

按照 "测试端点身份验证 > 动态分析 > 服务器身份验证 "的说明进行操作。如果这样做并没有导致流量被代理，这可能意味着证书固定实际上已经实施，所有的安全措施都已经到位。是否所有的域名都发生了同样的情况？

作为一个快速的测试，你可以尝试使用 "绕过证书固定"中描述的 objection 来绕过证书固定。被 objection 劫持的与固定有关的 API 应该出现在 objection 的输出中。



然而，请记住：

- 这些 API 可能并不完整。
- 如果没有任何东西被劫持，这并不一定意味着该应用没有实现固定功能。

在这两种情况下，应用程序或其某些组件可能会以 objection 所支持的方式实现自定义固定。请查看静态分析部分，了解特定固定的特征并进行更深入的测试。

6.6.5.3.2. 客户端证书固定

一些应用程序使用 mTLS（相互 TLS），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才能使用（解密）。
3. 证书可以存储在二进制文件本身、数据目录或 Keychain 中。

使用 mTLS 的最常见和最不恰当的方法是将客户端证书存储在应用程序包中，并且硬编码密码。这显然不会带来多少安全性，因为所有的客户都会共享同一个证书。

存储证书（可能还有密码）的第二种方法是使用 Keychain。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 Keychain 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。

6.6.6. 参考文献

6.6.6.1. OWASP MASVS

- MSTG-NETWORK-1: "数据在网络上使用 TLS 进行加密。安全通道在整个应用中被一致使用。"
- 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。”

6.7. 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 `mobile` 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。这些限制是通过沙盒策略（又称配置文件）实现的，由可信 BSD（MAC）强制访问控制框架通过内核扩展强制执行。iOS 对所有第三方应用程序应用一个通用的沙箱配置文件，称为容器。访问受

保护的资源或数据(有些也称为应用能力)是可能的，但它是通过称为权利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“Keychain 共享”），并且将在安装后直接生效。但是，对于其他权限，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
- 运动
- 音乐和媒体库
- 照片
- 提醒事项
- Siri
- 语音识别
- 电视提供程序

尽管 Apple 极力主张保护用户隐私，并明确如何申请权限，但应用程序出于不明显的原因而申请的权限数量仍然可能过多。

相机、照片、日历数据、运动、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。让我们考虑以下有关照片权限的例子，如果授予该权限，应用程序就可以访问 "Camera Roll"（iOS 默认的全系统照片存储位置）中的所有用户照片:

- 典型的二维码扫描应用显然需要摄像头来运作，但也可能要求照片权限。如果明确要求存储，并取决于所拍照片的敏感性，这些应用程序可能最好选择使用应用程序沙盒存储，以避免其他应用程序（有照片权限）访问它们。有关敏感数据存储的更多信息，请参阅 "iOS 数据存储 "一章。
- 一些应用程序需要上传照片（例如个人资料图片）。最近的 iOS 版本引入了新的 API，如 **`UIImagePickerController`**（iOS 11+）和其最新替代 **`PHPickerViewController`**（iOS 14+）。这些 API 在与你的应用程序分开的进程中运行，通过使用它们，应用程序可以只对用户选择的图片进行只读访问，而不是对整个 "Camera Roll"。这被认为是避免请求不必要的权限的最佳做法。

其他权限，如蓝牙或定位，需要更深入的验证步骤。它们可能是应用程序正常运行所需要的，但这些任务所处理的数据可能没有得到适当的保护。要了解更多信息和一些例子，请参考下面 "静态分析 "部分的 "源代码检查 "和 "动态分析 "部分。

当收集或简单处理（如缓存）敏感数据时，应用程序应提供适当的机制，让用户对其进行控制，例如，能够撤销访问或删除它。然而，敏感数据不仅可能被存储或缓存，也可能通过网络发送。在这两种情形下，必须确保应用程序正确地遵循适当的最佳实践，在这种情况下，涉及实施适当的数据保护和传输安全。关于如何保护这类数据的更多信息可以在"网络 API"一章中找到。

正如你所看到的，使用应用程序的能力和权限大多涉及处理个人数据，因此是一个保护用户隐私的问题。更多细节请参见 Apple 开发者文档中的 "保护用户隐私 "和 "访问受保护的资源"两篇文章。

6.7.1.1.1. 设备能力

设备能力被应用商店用来确保只有兼容的设备被列出，从而被允许下载该应用。它们被指定在应用程序的 **`Info.plist`** 文件中的 **`UIRequiredDeviceCapabilities`** 键下。

```
<key>UIRequiredDeviceCapabilities</key>
<array>
    <string>armv7</string>
</array>
```

通常，你会发现 armv7 功能，这意味着该应用只为 armv7 指令集编译，或者是 32/64 位通用应用。

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 设备能力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 Apple 的《核心蓝牙编程指南》解释了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 `bluetooth-peripheral` 或 `bluetooth-central` (均为 UIBackgroundModes)应用程序能力。

然而，这还不足以让应用程序访问蓝牙外围设备，因为 **NSBluetoothPeripheralUsageDescription** 键必须包含在 **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. 权限

依据 Apple 的 iOS 安全指南:

权限是签署在应用程序中的键值对，允许超越运行时因素的认证，如 UNIX 用户 ID。由于权限是数字签名的，它们不能被改变。系统应用程序和守护进程广泛使用权限，以执行特定的特权操作，否则就需要以 root 身份运行。这大大减少了被破坏的系统应用程序或守护程序的特权提升的可能性。

许多权限可以使用 Xcode 目标编辑器的 "Summary 摘要 "标签来设置。其他权限需要编辑目标的权利属性列表文件或从用于运行应用程序的 iOS 配置文件中继承。

权限来源:

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。

权限目的：

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：

- 在代码签名过程中，与应用程序启用的能力/服务相对应的权限被转移到应用程序的签名中，这些权限来自 Xcode 为签署该应用程序而选择的配置文件。
- 配置文件在构建过程中被内置到应用程序包中（**embedded.mobileprovision**）。
- Xcode 的“Build Settings 构建设置”选项卡中“Code Signing Entitlements 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”能力，则可以转到 Xcode 中的“能力 **Capabilities**”选项并启用“**Data Protection** 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>
```

对于其他能力（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。

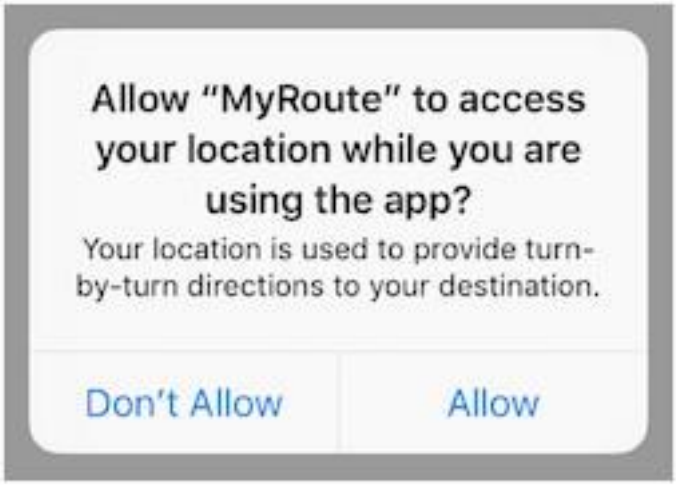
6.7.1.2. 静态分析

从 iOS 10 开始，以下是您需要检查权限的主要区域：

- Info.plist 文件中的目的字符串。
- 代码签名权限文件。
- 内置配置文件。
- 内置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 文件中的目的字符串

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提示中为用户显示这些字符串。

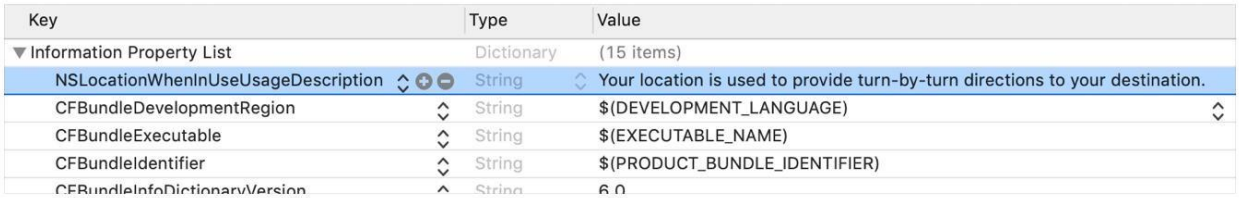


如果在 iOS 10 以上或之后进行链接，则开发人员必须在其应用程序的 Info.plist 文件中包含目标字符串。 否则，如果应用程序尝试在未提供相应目标字符串的情况下访问受保护的数据或资源，则访问将失败，并且该应用程序甚至可能崩溃。

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 -"开头的键。

您可以通过单击右键并选择“Show Raw Keys/Values 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**）。



如果只有 IPA：

- 解压缩 IPA。
- **Info.plist** 位于 **Payload/<appname>.app/Info.plist** 中。
- 根据需要进行转换（例如：**plutil -convert xml1 Info.plist**），如“ iOS 基本安全测试”章节的“ Info.plist 文件”部分所述。
- 检查所有目标字符串 Info.plist 键，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 y
our destination.</string>
```

有关可用的不同用途字符串 Info.plist 键的概述，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 引用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头或用户的健康记录。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**

这样的其他目的字符串，还必须考虑应用程序是否安全地存储了这些数据。有关安全存储敏感数据的更多信息和最佳实践，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（**application-groups**）：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DT
Ds/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>
```

上面概述的权利不需要用户的任何额外权限。但是，检查所有权利始终是一个好习惯，因为应用程序可能过度向用户请求权限，从而泄露信息。

如 Apple 开发人员文档中所述，必须具有应用程序组权限才能通过 IPC 或共享文件容器在不同应用程序之间共享信息，这意味着数据可以在设备上直接在应用程序之间共享。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，则也需要此权限。

根据要共享的数据，使用另一种方法来共享可能更合适，比如通过一个后台，在那里这些数据可能会被验证，避免被例如用户自己篡改。

6.7.1.2.3. 内置配置文件

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包文件夹（**Payload/<appname>.app/**）下的内置配置文件，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内置配置文件的权利：

```
security cms -D -i embedded.mobileprovision
```

然后搜索“权限”键区域（**<key>Entitlements</key>**）。

6.7.1.2.4. 内置在已编译应用程序二进制文件中的权限

如果你只有应用程序的 IPA 或者仅仅是越狱设备上安装的应用程序，你通常无法找到**.entitlements** 文件。这也可能是 **embedded.mobileprovision** 文件的情况。不过，你应该能够自己从应用程序二进制文件中提取权限属性列表（已经在 "iOS 基本安全测试 "一章的 "获取应用程序二进制文件 "一节中说明了这一点）。

即使是针对加密的二进制文件，下面的步骤也应该有效。如果出于某种原因，你必须用 Clutch（如果与你的 iOS 版本兼容）、frida-ios-dump 或类似工具解密和提取应用程序。

6.7.1.2.4.1 从应用程序二进制文件中提取权限 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"
```

或者，您可以使用 radare2（-qc 静默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<!D
OCTYPE plist PUBLIC
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd
">\n<plist version="1.0">
...<key>com.apple.security.application-groups</key>\n\t\t<array>
\n\t\t\t<string>group.ph.telegra.Telegraph</string>...

0x0016427d ascii H<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUB
```

```
LIC
"-//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 Telegram 中完全恢复原始的权限文件。

> 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 参数的 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 参数一起使用（将所有文件作为 ASCII 文本处理）：

```
$ grep -a -A 5 'PropertyList' /var/containers/Bundle/Application/
    15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35/Telegram X.app/Telegram\ X

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DT
Ds/PropertyList-1.0.dtd">
<plist version="1.0">
    <dict>
        <key>com.apple.security.application-groups</key>
        <array>
        ...
```

使用-A num, --after-context=num 参数可以显示更多或更少的行。 如果您也已将这些工具安装在越狱的 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（已针对多个应用商店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，现在是时候验证所要求的权限和分配的能力是如何被使用的。对于这一点，源代码审查应该足够了。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逆向工程，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** 文件中的目的字符串是否与编程实现相匹配。

- 是否以没有泄露机密信息的方式使用注册的能力。

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。这可以通过使用许多系统框架的专用 API 来完成，这些框架提供了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 的使用情况，并检查可能从这些 API 获得的敏感数据的情况。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能力的列表。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能力相关的敏感数据和 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、能力列表（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. 识别应用程序在访问相关功能时真正使用了哪些方法（例如 "分享你的位置"）。

5. 获取这些方法的回溯并尝试建立一个调用图。

一旦确定了所有的方法，你可以利用这些知识对应用程序进行逆向工程，并试图找出数据是如何被处理的。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。

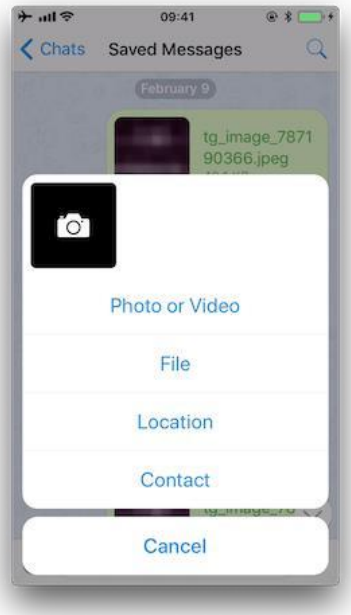
在下面的例子中，我们用 Telegram 打开聊天中的分享对话框，用 frida-trace 来识别哪些方法正在被调用。

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“包含‘**authorizationStatus**’的任何类的 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“**Location** 位置”，将跟踪另一种方法：

```
11186 ms  +[CLLocationManager authorizationStatus]
 11186 ms     | +[CLLocationManager _authorizationStatus]
 11186 ms     |    | +[CLLocationManager _authorizationStatusForBundleIdentif
ier: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());
  }
```

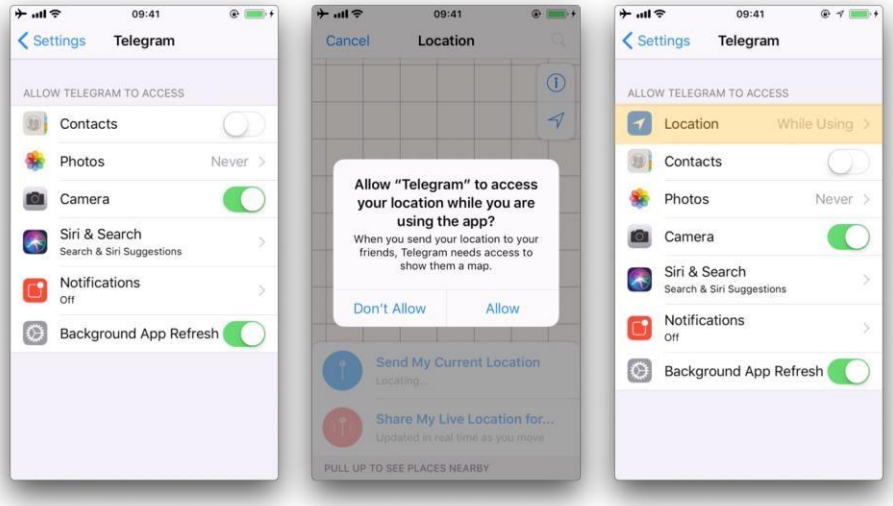
再次单击“Location 位置”将显示更多信息：

```
  3630 ms  -[CLLocationManager init]
  3630 ms     | -[CLLocationManager initWithEffectiveBundleIdentifier:0x0 bun
dle:0x0]
  3634 ms  -[CLLocationManager setDelegate:0x14c9ab000]
  3641 ms  +[CLLocationManager authorizationStatus]
RET: 0x4
  3641 ms  Called from:
0x1031aa158 TelegramUI!+[TGLocationUtils requestWhenInUserLocationAuthorizati
onWithLocationManager:]
    0x10337e2c0 TelegramUI!-[TGLocationPickerController initWithContext:inten
t:]
    0x101ee93ac TelegramUI!0x1013ac
```

我们看到`+[CLLocationManager authorizationStatus]`返回 0x4（CLAuthorizationStatus.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，在使用 iPhone/iPad 时，有一种直观的方法来检查一些应用程序的权限状态，即打开"设置"，向下滚动，直到找到你感兴趣的应用程序。当点击它时，这将打开 " ALLOW

APP_NAME TO ACCESS 允许 APP_NAME 访问 "屏幕。然而，可能还没有显示所有的权限。你将不得不触发它们，以便在该屏幕上列出。



例如，在前面的例子中，"位置 "条目没有被列出，直到我们第一次触发了权限对话。一旦我们这样做了，无论我们是否允许访问，"位置 "条目都会被显示出来。

6.7.2. 测试通过 IPC 暴露敏感功能（MSTG-PLATFORM-4）

在移动应用的实施过程中，开发者可能会应用传统的 IPC 技术（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，因为它比传统技术要成熟得多。在不考虑安全的情况下使用 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）能力相比，iOS 为应用程序之间的通信提供了一些相当有限的选择。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），用于深度链接。 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，后者最初也用于深度链接。

例如：Telegram 应用程序支持自定义 URL 方案和通用链接：

- **`tg://resolve?domain=fridadotre`** 是一个自定义 URL 方案，使用 **`tg://`**方案.
- **`https://telegram.me/fridadotre`** 是通用链接，使用 `https://`方案

两者的结果都是一样的，用户将被重定向到 Telegram 的指定聊天室（本例中为 "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，其他应用无法声明相同的通用链接，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：

- 检查关联的域权限。
- 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. 检查关联域的权限

通用链接要求开发人员添加关联域权限，并在其中包括应用程序支持的域的列表。

在 Xcode 中，转到“**Capabilities** 功能”选项卡，然后搜索“**Associated Domains** 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。

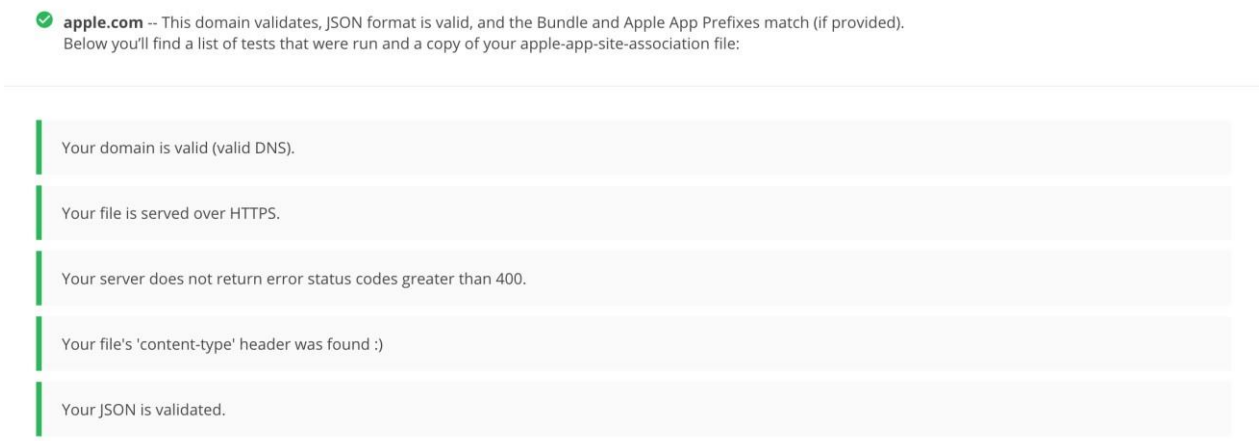
如果您没有原始源代码，您仍然可以搜索它们，如“编译的应用程序二进制文件中内置的权限”中所述。

6.7.2.2.2.2. 获取 Apple App Site Association 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/appleapp-site-association 上访问此文件，而无需任何重定向。

你可以自己使用浏览器并浏览到 https://<domain>/apple-site-association，https://<domain>/.known/apple-site-association 或使用苹果的 CDN https://app-site-association.cdn-apple.com/a/v1/<domain>来检索它。

另外，你也可以使用 Apple APP Site Association（AASA）验证器。输入域名后，它将显示该文件，为您验证文件并显示结果（例如，如果它没有通过 HTTPS 正确提供服务）。请看下面的例子，来自 apple.com https://www.apple.com/.well-known/apple-app-site-association：




```
{
    "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，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”键相匹配。接下来，使用“paths”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*（`"paths": ["*"]`）以允许所有可能的路径。只有当网站的特定区域不应该被某些应用程序处理时，开发者才可以通过在相应的路径中预留 "NOT "（注意 T 后面的空白）来限制访问。还请记住，系统将按照数组中字典的顺序查找匹配项（匹配第一个匹配项）。

此路径排除机制不应被视为安全功能，而应该被视为一种过滤器，开发者可以用它来指定哪些应用程序打开哪些链接。默认情况下，iOS 不会打开任何未经验证的链接。

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`** 权限中检索声明的域（**`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 不匹配（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。由于调用来自应用程序，因此不会作为通用链接处理。

> 根据 Apple 文档: 当 iOS 在用户点击一个通用链接后启动你的应用程序时，你会收到一个 **NSUserActivity** 对象，其 **activityType** 值为 **NSUserActivityTypeBrowsingWeb**。该活动对象的 **webpageURL** 属性包含用户正在访问的 URL。网页 URL 属性总是包含一个 HTTP 或 HTTPS URL，你可以使用 **NSURLComponents** API 来操作 URL 的组件。[...] 为了保护用户的隐私和安全，当你需要传输数据时，你不应该使用 HTTP，而应该使用安全传输协议，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:
```

6.7.2.2.2.4. 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：

> 通用链接为你的应用程序提供了一个潜在的攻击媒介，所以要确保验证所有的 URL 参数，并丢弃任何畸形的 URL。此外，将可用的操作限制在不危及用户数据的范围内。例如，不允许通用链接直接删除内容或访问用户的敏感信息。当测试你的 URL 处理代码时，确保你的测试案例包括格式不正确的 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 **activityType** 值为 **NSUserActivityTypeBrowsingWeb**。活动对象的 **webpageURL** 属性包含用户访问的 HTTP 或 HTTPS URL。下面这个 Swift 的例子正是在打开 URL 之前验证了这一点：

```
func application(_ application: UIApplication, continue userActivity: NSUserA
ctivity,
                 restorationHandler: @escaping ([UIUserActivityRestoring]?)
-> Void) -> Bool {
```

```
    // ...
    if userActivity.activityType == NSUserActivityTypeBrowsingWeb, let url =
userActivity.webpageURL {
        application.open(url, options: [:], completionHandler: nil)
    }

    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不应该被信任（即使来自可信的域）。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，整个 URL 以及通用链接请求必须被丢弃。

NSURLComponents API 可以用来解析和操作 URL 的组成部分。这也可以是**application:continueUserActivity:restorationHandler:**方法本身的一部分，或者可能发生在一个被调用的单独方法上。下面的例子演示了这一点：

```
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, resolvingAgainstBa
seURL: true),
        let path = components.path,
        let params = components.queryItems else {
        return false
    }

    if let albumName = params.first(where: { $0.name == "albumname" })?.valu
e,
        let photoIndex = params.first(where: { $0.name == "index" })?.value {
        // Interact with album name and photo index

        return true

    } else {
        // Handle when album and/or 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: [UIApplicationOpenURLOptionUniversalLinksOnl
y: true as NSNumber],
                        completionHandler: { value in completion.completion(v
alue)}
            )
```

请注意应用程序在打开它之前如何将方案调整为“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，可以在符号和应用程序二进制的字符串中搜索。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“识别和劫持 URL 处理程序方法”，以获取有关劫持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。
- Apple App Site Association 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，你不能从 Safari 浏览器中直接输入通用链接，因为这是苹果公司不允许的。但你可以在任何时候使用其他应用程序（如笔记应用程序）来测试它们：

- 打开“笔记”应用程序并创建一个新笔记。
- 编写包含域的链接。
- 在笔记应用程序中保留编辑模式。
- 长按链接以将其打开（请记住，单击一次会触发默认选项）。

  要从 Safari 浏览器中做到这一点，你必须在一个网站上找到一个现有的链接，一旦点击，它将被识别为一个通用链接。这可能有点耗费时间。

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。

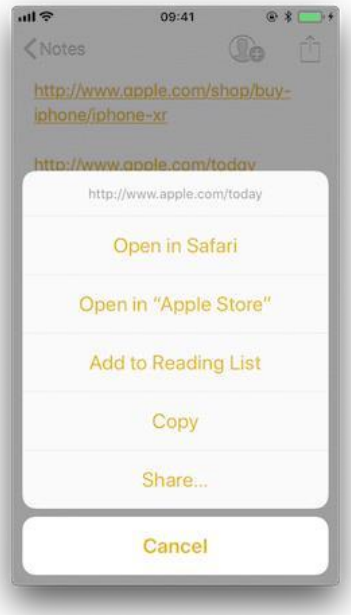
从上面看到的 apple.com 的 `apple-app-site-association` 中，我们选择了以下路径：

```
"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劫持或跟踪 **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：

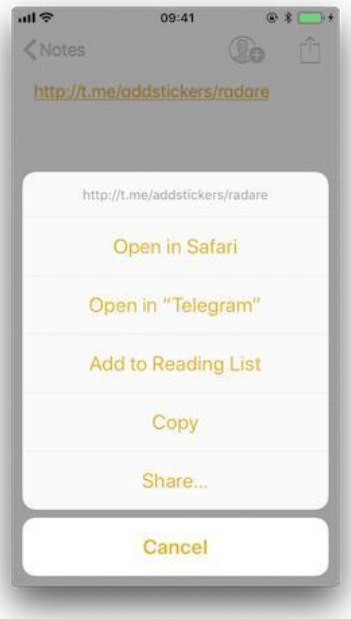
```
{
    "applinks": {
        "apps": [],
        "details": [
            {
                "appID": "X834Q8SBVP.org.telegram.TelegramEnterprise",
                "paths": [
```

```
                "*"
            ]
        },
        {
            "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笔记应用程序和 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）到笔记应用并在其中打开。



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函数并对其进行了插桩。现在触发通用链接并观察追踪。

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c423
7780
                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\twebpageURL: " + 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("\trestorationHandler: " +ObjC.Object(args[4]).toString());
  },
```

新的输出为：

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c423
7780
                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）。如果查看 Apple 开发者文档，我们可以看到我们还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，以发现是否有任何其他函数参与到打开 URL 中。

```
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"包含任何包含'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...
$S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationDat
a18application...
$S10TelegramUI31AuthorizationSequenceControllerC7account7strings7openUrl5apiI
d0J4HashAC0A4Core19...
...
```

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追踪。

```
           /* TID 0x303 */
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c423
7780
                restorationHandler:0x16f27a898]
298619 ms     | $S10TelegramUI15openExternalUrl7account7context3url05forceD01
6presentationData
                18applicationContext20navigationController12dismissInputy0A4C
ore7AccountC_AA
                14OpenURLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C
7Display0
                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：

xcrun 可以用来从命令行中调用 Xcode 开发工具，而不需要将它们放在路径中。在这种情况下，它将定位并运行 swift-demangle，这是一个 Xcode 工具，可以解开 Swift 符号。

```
$ xcrun swift-demangle S10TelegramUI15openExternalUrl7account7context3url05fo
rceD016presentationData
18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA1
4OpenURLContextOSSSbAA0
12PresentationK0CAA0a11ApplicationM0C7Display010NavigationO0CSgyyctF
```

结果为：

```
---> TelegramUI.openExternalUrl(
    account: TelegramCore.Account, context: TelegramUI.OpenURLContext, url: S
wift.String,
    forceExternal: Swift.Bool, presentationData: TelegramUI.PresentationData,
    applicationContext: TelegramUI.TelegramApplicationContext,
    navigationController: Display.NavigationController?, dismissInput: () ->
()) -> ()
```

这不仅给你提供了方法的类（或模块）、名称和参数，还揭示了参数类型和返回类型，所以如果你需要深入研究，现在你知道该从哪里开始。

现在，我们将使用这些信息，通过编辑存根文件来正确显示参数：

```
// __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:0x1c423
7780
```

```
                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.B
ool,
presentationData: TelegramUI.PresentationData, applicationContext:
TelegramUI.TelegramApplicationContext, navigationController: Display.Navigati
onController?,
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`**。

现在你可以继续下去，尝试追踪和验证数据是如何被验证的。例如，如果你有两个通过通用链接进行通信的应用程序，你可以用它来查看发送应用程序是否通过在接收应用程序中劫持这些方法而泄露了敏感数据。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，请确保劫持 `userInfo` 属性或直接从劫持的 `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 Handoff 的最后说明

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** 权限和服务器的 **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，前面"检查链接的打开方式"的例子与"Handoff 编程指南"中描述的 "网络浏览器到原生应用程序的 Handoff "场景非常相似：

> 如果用户在原设备上使用 Web 浏览器，并且接收设备是具有原生应用程序的 iOS 设备，该应用要求获得 **webpageURL** 属性的域部分，则 iOS 启动原生应用程序，并向它发送一个 **NSUserActivity** 对象，其 **activityType** 值为 **NSUserActivityTypeBrowsingWeb**。**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：

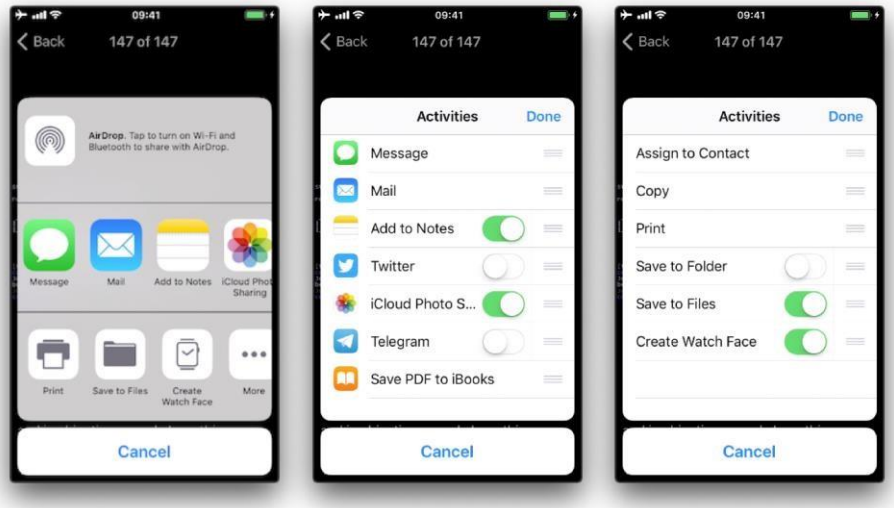
```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c423
7780
               restorationHandler:0x16f27a898]
298382 ms  application:<Application: 0x10556b3c0>
298382 ms  continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms       webpageURL:http://t.me/addstickers/radare
298382 ms       activityType:NSUserActivityTypeBrowsingWeb
298382 ms       userInfo:{
}
298382 ms  restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

在测试支持 Handoff 的应用程序时，这些知识应该对你有所帮助。

6.7.2.3. UIActivity 分享

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）分享数据（项目）。从用户的角度来看，此功能是众所周知的系统范围的分享活动表，该活动表在单击“分享”按钮后显示。



可用的内置分享机制（又称活动类型）包括：

- airDrop
- 指定联系人
- 复制到剪贴板
- 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分享机制。

6.7.2.3.2. 静态分析

6.7.2.3.2.1. 发送项目

测试 UIActivity 分享时，应特别注意：

- 分享的数据（项目）。

- 自定义活动。
- 排除的活动类型。

通过 **UIActivity** 进行数据分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 **excludedActivityTypes** 属性排除某些分享机制。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，请尝试搜索前面的方法和属性，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：

- 应用程序是否通过查找“导出/导入 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。
  - 应用程序是否通过查看文档类型（Xcode 项目的 "信息 "标签）指定了任何可以打开的文档类型。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 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 **UIApplicationDelegate**

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**，应用程序是否声明了导出/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。

有关使用这些键的非常完整的说明，请参见 Stackoverflow 。

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 run ios plist cat Info.plist
```

请注意，这与我们从手机上检索 IPA 或通过例如 SSH 访问并导航到 IPA/应用程序沙盒中的相应文件夹的情况相同。然而，有了 objection，我们离我们的目标只有一个命令，这仍然可以被认为是静态分析。

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

通过执行动态插桩，可以轻松检查以下三点：

- **activityItems**：要分享的项目的数组。它们可能是不同的类型，例如：一个字符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：

- 劫持我们在静态分析中看到的方法**(init(activityItems: applicationActivities:))**，以获取 **activityItems** 和 **applicationActivities**。

- 通过劫持 **excludedActivityTypes** 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。首先准备劫持，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'].implementati
on, {
  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，当您第一次分享图片时观察输出：

```
[*] 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
```

然后是文本文档。

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<QLActivityItemProvider: 0x1c4a30140>",
    "<UIPrintInfo: 0x1c0699a50>"
)
applicationActivities: (
)
RET @ 0x13c4bdc00:
<_UIDICActivityViewController: 0x13c4bdc00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x1c001b1d0:
(
    "com.apple.UIKit.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分享了图片，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分享文件，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）。
- 劫持 **`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或使用模糊测试技术。

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“**Accept** 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“**Open with...**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，以隐藏应用的真实名称）：

4. 选择“**SomeFileManager**”后，我们将看到以下内容：

```
5. (0x1c4077000)  -[AppDelegate application:openURL:options:]
application: <UIApplication: 0x101c00950>
openURL: file:///var/mobile/Library/Application%20Support
                    /Containers/com.some.filemanager/Documents/Inbox/OWAS
P_MASVS.pdf
options: {
    UIApplicationOpenURLOptionsAnnotationKey =     {
        LSMoveDocumentOnOpen = 1;
    };
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.sharingd
";
    "_UIApplicationOpenURLOptionsSourceProcessHandleKey" = "<FBSProcessHa
ndle: 0x1c3a63140;
                                                            sharingd:
605; valid: YES>";
```

```
    }
    0x18c7930d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:o
    rigin:]_block_invoke
    ...
    0x1857cdc34 FrontBoardServices!-[FBSSerialQueue _performNextFromRunLoopSo
    urce]
    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_MASV
S.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:origi
n:]_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:`如何调用 `__handleOpenURL:`，后者调用 `moveItemAtPath:toPath:error:`。注意，我们现在在没有目标应用程序的源代码的情况下就得到了这些信息。我们必须做的第一件事是：劫持 **application:openURL:options:**。关于其余的内容，我们不得不三思而后行，想出一些我们可以开始追踪的与文件管理器有关的方法，例如“ copy”、“move”、“remove”等字符串的方法。直到我们发现被调用的是 **moveItemAtPath:toPath:error:**。

这种处理传入文件的方式对于自定义 URL 方案也是一样的。请参考 "测试自定义 URL 方案 "一节以了解更多信息。

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

Apple 与 iOS 8 一起推出了应用扩展。根据《Apple 应用程序扩展编程指南》，应用扩展让应用在与其他应用或系统互动时向用户提供自定义功能和内容。为了做到这一点，它们实现了具体的、范围很广的任务，例如，定义用户点击 "分享 "按钮并选择某些应用程序或动作后会发生什么，为 "今天 "小部件提供内容或启用自定义键盘。

根据任务的不同，应用程序扩展将有一个特定的类型（而且只有一个），即所谓的扩展点。一些值得注意的是：

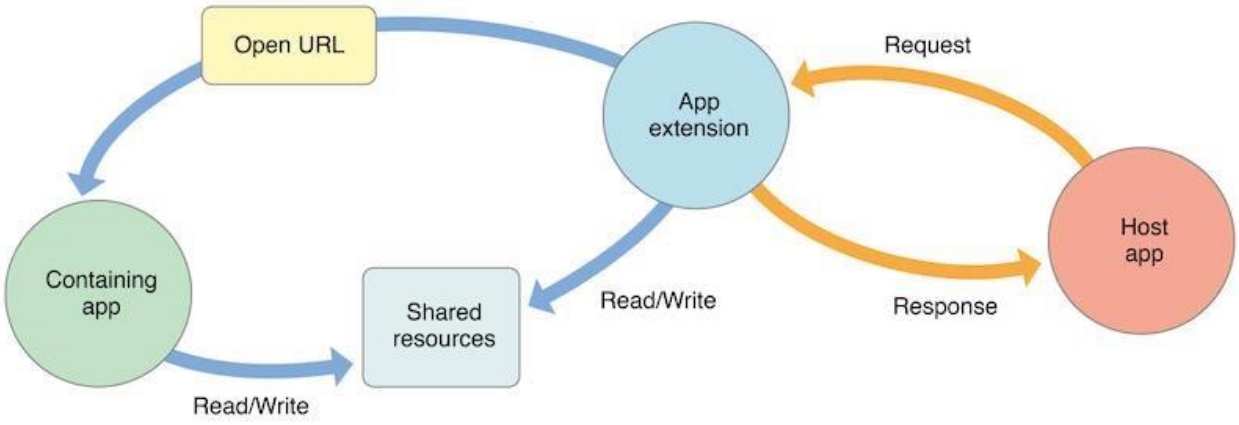
- 自定义键盘：用一个自定义的键盘取代 iOS 系统键盘，在所有应用程序中使用。
- 分享：发布到分享网站或与他人分享内容。
- 今日：也叫小部件，它们在通知中心的今日视图中提供内容或执行快速任务。

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：是捆绑在容器应用程序中的扩展。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的应用程序（第三方）。
- 容器应用程序：是包含应用程序扩展的应用程序。

例如：用户在主机应用程序中选择文本，单击“分享”按钮，然后从列表中选择一个“应用程序”或操作。这会触发容器应用程序的应用程序扩展。应用扩展在主机应用的内容中显示其视图，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，它很好地概括了这一点：



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。
- 今日小部件（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，但可以启动上传或下载。
- 应用程序扩展无法访问 iOS 设备上的相机或麦克风（iMessage 应用程序扩展除外）。

6.7.2.4.2. 静态分析

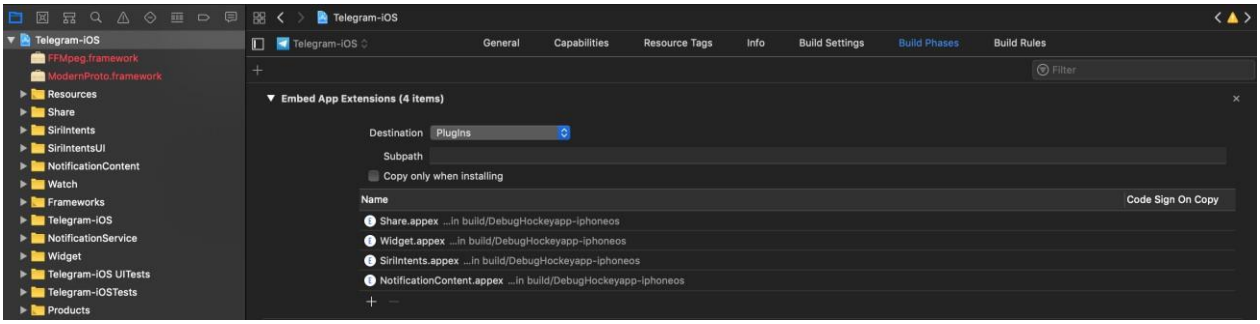
静态分析将负责：

- 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“Build Phases / Embed App extensions”：



在这里，您可以找到所有内置的应用程序扩展的名称，后缀是.appex，现在您可以浏览到项目中的各个应用程序扩展。

如果没有原始源代码：

在应用包（IPA 或已安装的应用）内的所有文件中搜索 `NSExtensionPointIdentifier`。

```
$ grep -nr NSExtensionPointIdentifier Payload/Telegram\ X.app/
Binary file Payload/Telegram X.app//PlugIns/SiriIntents.appex/Info.plist matc
hes
Binary file Payload/Telegram X.app//PlugIns/Share.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/NotificationContent.appex/Info.pl
ist 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.a
ppex/Info.plist matches
```

您还可以通过 SSH 访问，找到应用程序包并列出所有内部插件（默认情况下它们位于此处）或使用 objection：

```
ph.telegra.Telegraph on (iPhone: 11.1.2) [usb] # cd PlugIns
    /var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35/
    Telegram X.app/PlugIns

ph.telegra.Telegraph on (iPhone: 11.1.2) [usb] # ls
```

```
NSFileType      Perms  NSFileProtection    Read    Write     Name
------------  -------  ------------------  ------  -------  ----------------
---------
Directory         493  None                True    False     NotificationCont
ent.appex
Directory         493  None                True    False     Widget.appex
Directory         493  None                True    False     Share.appex
Directory         493  None                True    False     SiriIntents.appe
x
```

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“分享”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。值得注意的是，这与通过 **UIActivity** 进行的数据共享不同，在 **UIActivity** 中我们必须定义文档类型，也是使用 UTI。一个应用程序不需要为此拥有一个扩展。仅仅使用 **UIActivity** 就可以实现数据共享。

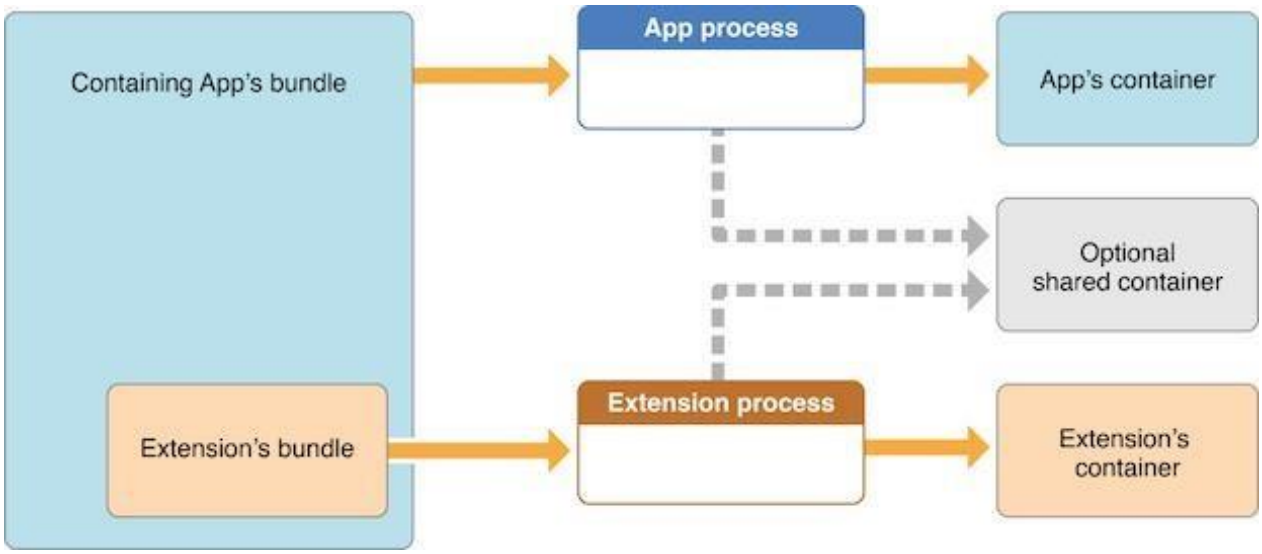
检查应用程序扩展的 **Info.plist** 文件，然后搜索 **NSExtensionActivationRule**。该键指定了要支持的数据，例如：支持的最大项目。比如说：

```
<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>
```

只有在这里出现的数据类型和没有 0 作为 MaxCount 的数据类型才被支持。然而，通过使用一个所谓的谓词字符串来评估所给的 UTI，可以进行更复杂的过滤。请参考《Apple 应用程序扩展编程指南》以获得更详细的相关信息。

6.7.2.4.2.3. 检查与包含应用程序的数据共享

请记住，应用程序扩展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 应用程序扩展编程指南》中的以下图：



如指南中所述，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

可以通过使用以下方法拒绝特定类型的应用程序扩展：

- `application:shouldAllowExtensionPointIdentifier:`

然而，目前只有 "自定义键盘 "应用程序扩展可以做到这一点（在测试通过键盘处理敏感数据的应用（如银行应用）时，应进行验证）。

6.7.2.4.3. 动态分析

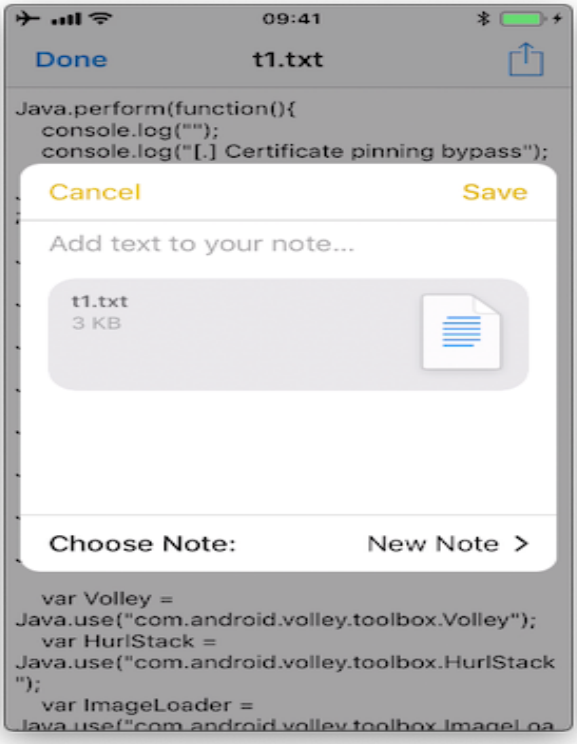
对于动态分析，我们可以执行以下操作来获得知识，而无需源代码：

- 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，我们应该劫持数据源应用程序中 **NSExtensionContext - inputItems**。

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“分享”按钮，以在笔记应用程序中使用该文件创建笔记：



如果运行跟踪，将看到以下输出：

```
(0x1c06bb420) NSExtensionContext - inputItems
0x18284355c Foundation!-[NSExtension _itemProviderForPayload:extensionContex
t:]
0x1828447a4 Foundation!-[NSExtension _loadItemForPayload:contextIdentifier:co
mpletionHandler:]
0x182973224 Foundation!__NSXPCCONNECTION_IS_CALLING_OUT_TO_EXPORTED_OBJECT_S3
__
0x182971968 Foundation!-[NSXPCConnection _decodeAndInvokeMessageWithEvent:fla
gs:]
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 \"pu
blic.file-url\"\n)}"
```

```
    );
}"
)
```

在这里我们可以观察到：

- 这发生在 XPC 的内部，具体来说，它是通过使用 **libxpc.dylib** 框架的 **NSXPCConnection** 实现的。
- **NSItemProvider** 中包含的 UTI 是 **public.plain-text** 和 **public.file-url**，后者被包含在 Telegram 的 "分享扩展"的 **Info.plist** 中的 **NSExtensionActivationRule**。。

6.7.2.4.3.2. 识别涉及的应用程序扩展

您还可以通过劫持 **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-47D
A-90CD-34D73121C785
1(2) /private/var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-6CA85
B65FA35
/Telegram X.app/PlugIns/Share.appex>

(0x1c0372300)  -[NSExtension _plugIn]
RET: <PKPlugin: 0x10bff7910 com.apple.mobilenotes.SharingExtension(1.5) 73E4F
137-5184-4459-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** 正在发送文本文件（**public.plain-text** 和 **public.file-url**）
- **com.apple.mobilenotes.SharingExtension.appex** 正在接收并将处理文本文件。

如果你想了解更多关于 XPC 内部发生的事情，我们建议看一下 "libxpc.dylib"的内部调用。例如，你可以使用 **frida-trace**，然后通过扩展自动生成的存根深入了解你认为更有趣的方法。

6.7.2.5. UIPasteboard

6.7.2.5.1. 概述

在输入框中输入数据时，可以用剪贴板来复制数据。剪贴板可以被整个系统访问，因此被应用程序共享。这种共享可能被恶意的应用程序滥用，以获取存储在剪贴板中的敏感数据。

当使用一个应用程序时，你应该意识到其他应用程序可能会不断地读取剪贴板，就像 Facebook 应用程序那样。在 iOS 9 之前，恶意应用程序可能会在后台监控剪贴板，同时定期检索 [UIPasteboard generalPasteboard].string。从 iOS 9 开始，只有前台的应用程序可以访问剪贴板的内容，这大大减少了从剪贴板嗅探密码的攻击面。尽管如此，复制粘贴密码仍然是一个你应该注意的安全风险，但也不能由应用程序来解决。

- 防止粘贴到应用程序的输入字段，并不能防止用户复制敏感信息。因为在用户注意到无法粘贴之前，信息已经被复制了，恶意应用程序已经嗅到了剪贴板。
- 如果在密码栏上禁止粘贴，用户甚至可能会选择他们可以记住的较弱的密码，他们不能再使用密码管理器，这将与使应用程序更安全的初衷相矛盾。

UI 剪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。剪贴板有两种：

- **系统范围的通用剪贴板**：用于与任何应用程序共享数据。默认情况下，在设备重启和应用卸载时持续存在（自 iOS 10 开始）。
- **自定义/命名剪贴板**：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进程中可用）。默认情况下为非持久性（自 iOS 10 开始），也就是说，它们只存在于拥有（创建）的应用程序退出之前。

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。
- 自 iOS 9 以来，应用程序在后台时无法访问剪贴板，因此减轻了后台剪贴板的监视。但是，如果恶意应用程序再次被带到前台，并且数据仍保留在剪贴板上，则它无需用户的知情或同意就可以以编程方式检索它。
- Apple 警告说，并不鼓励使用持久的命名剪贴板。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 Handoff 功能已默认启用。它允许常规剪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，也可以为复制的数据设置一个失效时间和日期。

6.7.2.5.2. 静态分析

系统通用剪贴板可以通过使用 `generalPasteboard` 获得，搜索源代码或编译后的二进制文件以获得该方法。处理敏感数据时，应避免使用系统范围内的通用剪贴板。

可以使用 `pasteboardWithName:create:`或 `pasteboardWithUniqueName` 创建**自定义剪贴板**。验证自定义剪贴板是否被设置为持久性，因为自 iOS 10 以来这已经被废弃了。应该使用一个共享容器来代替。

另外，可以检查以下内容：

- 检查是否使用 `removePasteboardWithName:`删除剪贴板，它使一个应用程序的剪贴板失效，释放它使用的所有资源（对常规剪贴板没有影响）。
- 检查是否存在排除的剪贴板，应该调用 `setItems:options:`设置 `UIPasteboardOptionLocalOnly` 选项。
- 检查是否存在即将到期的剪贴板，应该调用 `setItems:options:`设置 `UIPasteboardOptionExpirationDate` 选项。
- 检查应用程序是否在进入后台或终止时清空剪贴板项目。通常一些密码管理器应用程序会这样，试图限制敏感数据的暴露。

6.7.2.5.3. 动态分析

6.7.2.5.3.1. 检测剪贴板使用情况

劫持或跟踪以下内容：

- `generalPasteboard` 用于系统范围的通用剪贴板。
- `pasteboardWithName:create:`和 `pasteboardWithUniqueName` 用于自定义剪贴板。

6.7.2.5.3.2. 检测持久性剪贴板使用情况

劫持或跟踪不推荐使用的 `setPersistent:`方法，并验证是否调用它。

6.7.2.5.3.3. 监视和检查剪贴板项目

监视剪贴板时，有几个细节可能会被动态地检索：

- 通过劫持 `pasteboardWithName:create:`并检查其输入参数或劫持 `pasteboardWithUniqueName` 并检查其返回值来获得剪贴板名称。
- 获取第一个可用的剪贴板项目：例如对于字符串，请使用 **`string`** 方法。或对标准数据类型使用任何其他方法。
- 使用 **`numberOfItems`** 获取项目数量。
- 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：图片 public.png 和 public.tiff（UTIs），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了解该类型的应用程序才能理解写入剪贴板的数据。这与我们在“ UIActivity 分享”部分中看到的相同。使用 **`itemSetWithPasteboardTypes:`**检索它们，并设置相应的 UTI。
- 通过劫持 **`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 objection 的命令 **`ios pasteboard monitor`**:

> 劫持 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剪贴板监视器，该监视器可监视上述特定信息。

例如：此脚本（灵感来自 objection 的剪贴板监视器使用的脚本）每 5 秒读取一次剪贴板项目，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: f
alse
(
    {
        "public.utf8-plain-text" = hola;
    }
)
[* Pasteboard changed] count: 65 hasStrings: true hasURLs: true hasImages: fa
lse
(
    {
        "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} ori
entation 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，最后复制了一张图片。其中一些是通过不同的 UTI 提供的。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。一个应用程序必须声明支持这些方案，并处理使用这些方案的传入 URL。

Apple 公司在《Apple 开发者文档》中对不当使用自定义 URL 方案提出警告：

URL 方案使应用程序引入了潜在的攻击载体，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，将可用操作限制在不危及用户数据的范围内。例如：不允许其他应用直接删除内容或访问用户的敏感信息。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。

同时还建议如果目的是实现深度链接，则改用通用链接：

> 尽管自定义 URL 方案是可接受的深度链接形式，但作为最佳实践，强烈建议使用通用链接。

支持自定义 URL 方案的方法是：

- 定义应用程序 URL 的格式。
- 注册方案，以便系统将适当的 URL 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，该协议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请求用户的许可，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 替换为收费号码）来利用此漏洞，因此，任何不慎访问了恶意网站的 Skype 用户都会拨打这个收费号码。

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 调用的操作也是一个有用的控制措施。

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

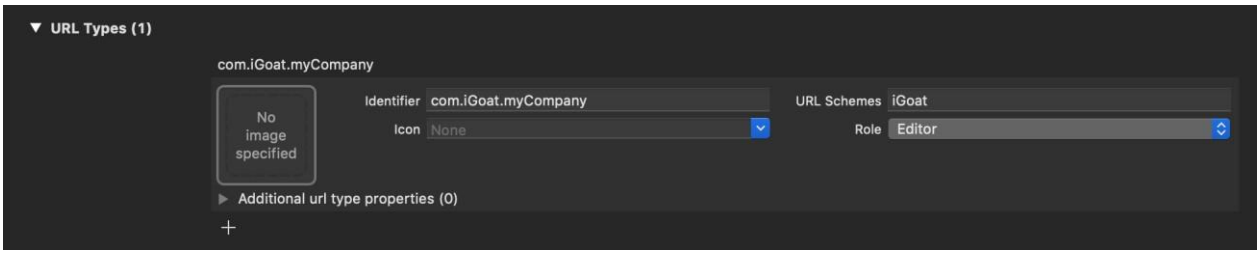
我们可以在静态分析中做些事情。在接下来的章节中，大家将会看到以下内容：

- 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程序。

如果您有原始源代码，并且想要查看注册的协议处理程序，只需在 Xcode 中打开项目，转到“**Info**（信息）”选项卡，然后打开“**URL Types**（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 键来找到相关信息:

```
<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，已注册的协议处理程序位于应用程序包的根文件夹中的 `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 `UIApplicationopenURL:options:completionHandler:`方法打开它们来传递参数。

《iOS 应用程序编程指南》中的提示:

如果有一个以上的第三方应用程序注册处理相同的 URL 方案，目前还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 键添加到应用程序的 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:`方法或 `application:will-FinishLaunchingWithOptions:`:验证如何做出决定以及如何检索 URL 的信息。
- application:openURL:options::验证如何打开资源，即如何解析数据，验证选项，特别是调用应用程序（sourceApplication）的访问是否应该被允许或被拒绝。在使用自定义 URL 方案时，应用程序可能还需要用户许可。

在 Telegram 中，您会找到四种使用的方法：

```
func application(_ application: UIApplication, open url: URL, sourceApplicati
on: String?) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, open url: URL, sourceApplicati
on: 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过时的方法，例如：application:handleOpenURL: 和 application:openURL:sourceApplication:annotation:。
- 没有任何一种方法验证源应用程序。
- 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

openURL:options:completionHandler:方法和 UIApplication 的过时 openURL：方法负责打开 URL（即向其他应用程序发送请求/进行查询），它可能是本地的当前应用程序，也可能是必须由另一个应用程序提供的。如果你有原始源代码，你可以直接搜索这些方法的使用。

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匹配表达式中。例如：在 Google 快速搜索显示：

```
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://
...
```

我们这次不使用 Xcode，仅使用 egrep 在 Telegram 源代码搜索此方法：

```
$ egrep -nr "open.*options.*completionHandler" ./Telegram-iOS/

./AppDelegate.swift:552: return UIApplication.shared.open(parsedUrl,
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumbe
r],
    completionHandler: { value in
./AppDelegate.swift:556: return UIApplication.shared.open(parsedUrl,
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumbe
r],
    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: "ht
tps://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“App Store 页面”。

当仅搜索://，我们可看到：

```
if documentUri.hasPrefix("file://"), let path = URL(string: documentUri)?.pat
h {
if !url.hasPrefix("mt-encrypted-file://?") {
guard let dict = TGStringUtils.argumentDictionary(inUrlString: String(url[ur
l.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，它将使用过时的 openURL 方法。

如果只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 或搜索常见 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。

您可以通过使用 strings 命令首先验证应用二进制文件是否包含这些字符串、:

```
strings <yourapp> | grep "someURLscheme://"
```

更好的做法是，使用 radare2 的 `iz/izz` 命令或 rafind2，两者都将找到 unix `strings` 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://?conta
ctNumber=
```

6.7.3.2.5. 测试过时的方法

搜索过时的方法，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
- 识别并劫持 URL 处理方法
- 测试 URL 方案源验证
- 进行 URL 方案模糊测试

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，您可以正常浏览到这些页面，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），否则链接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.sh
aredApplication();
                            var toOpen = ObjC.classes.NSURL.URLWithString_(ur
l);
                            return UIApplication.openURL_(toOpen);
                        }
[iPhone::iGoat-Swift]-> openURL("tel://234234234")
true
```

如 Frida CodeShare 中示例的那样，作者使用非公共 API `LSApplication Workspace.openSensitiveURL:withOptions:`打开 URL（通过 SpringBoard 应用程序）：

```
function openURL(url) {
    var w = ObjC.classes.LSApplicationWorkspace.defaultWorkspace();
    var toOpen = ObjC.classes.NSURL.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。

6.7.3.3.2. 识别和劫持 URL 处理方法

如果您无法查看原始源代码，则必须自己找出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，或者该应用程序使用的是已过时的方法。

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，通过提供简单的匹配模式，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，因此我们的匹配模式将是*[**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:openURLAnnotat
ionForActivityType:]
Observing  -[CNQuickActionsManager _openURL:]
Observing  -[SUClientController openURL:]
Observing  -[SUClientController openURL:inClientWithIdentifier:]
Observing  -[FBSSystemService openURL:application:options:clientPort:withResu
lt:]
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:]
...
```

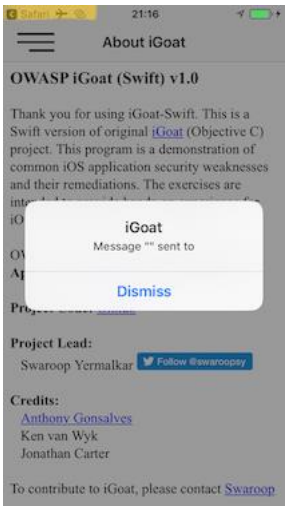
列表很长，其中包括我们已经提到的方法。如果我们现在触发一个 URL 方案，例如：来自 Safari 的“ igoat://”并接受在应用程序中打开它，我们将看到以下内容：

```
[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:origi
n:]_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，并且没有过时。
- 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。
- 我们还可以知道它是从哪里被调用的，正如我们所期望的那样，从-[UIApplication _applicationOpenURLAction:payload:origin:]。
- 该方法返回 0x1，表示 YES（委托成功处理了请求）。

调用成功，我们现在看到 iGoat 应用已打开：



请注意，如果我们在截图的左上角看一下，我们还可以看到调用者（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

看看其他方法在途中被调用也是很有意思的。为了改变一下结果，我们将从 iGoat 应用程序本身调用相同的 URL 方案。我们将再次使用 ObjC 方法观察器和 Frida REPL：

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer

[iPhone::iGoat-Swift]-> function openURL(url) {
                            var UIApplication = ObjC.classes.UIApplication.sh
aredApplication();
                            var toOpen = ObjC.classes.NSURL.URLWithString_(ur
l);
                            return UIApplication.openURL_(toOpen);
                        }

[iPhone::iGoat-Swift]-> observeSomething("*[* *openURL*]");
[iPhone::iGoat-Swift]-> openURL("iGoat://?contactNumber=123456789&message=hol
a")

(0x1c409e460)  -[__NSXPCInterfaceProxy__LSDOpenProtocol openURL:options:compl
etionHandler:]
openURL: iGoat://?contactNumber=123456789&message=hola
options: nil
completionHandler: <__NSStackBlock__: 0x16fc89c38>
0x183befbec MobileCoreServices!-[LSApplicationWorkspace openURL:withOptions:e
rror:]
0x10ba6400c
...
RET: nil
```

```
...

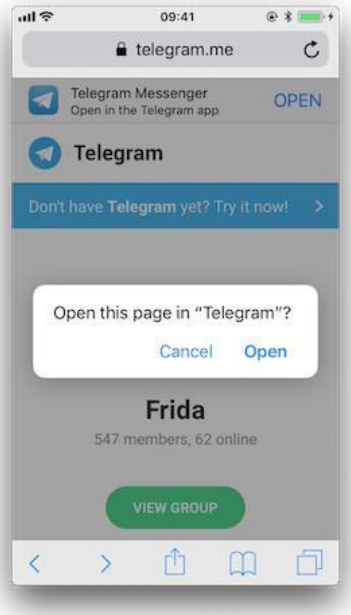
(0x101d0fad0)  -[UIApplication openURL:]
openURL: iGoat://?contactNumber=123456789&message=hola
0x10a610044
...
RET: 0x1

true
(0x1c4038280)  -[iGoat_Swift.AppDelegate 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:origi
n:]_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：您将劫持或篡改下一个方法。

6.7.3.3.2.3. 通过浏览到页面并让 Safari 打开它来打开链接

现在你可以在点击一个页面上包含的链接时测试同样的情况。Safari 将识别和处理 URL 方案，并选择执行哪个动作。打开这个链接 "https://telegram.me/fridadotre"将触发这种行为。



首先，我们让 frida-trace 为我们生成分支：

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
    -m "*[* *application*URL*]" -m "*[* openURL]"

...
7310 ms  -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: 0x10
c5ee4c0 origin: 0x0]
7311 ms     | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 opt
ions: 0x1c0e222c0]
7312 ms     | $S10TelegramUI15openExternalUrl7account7context3url05forceD016p
resentationData
            18applicationContext20navigationController12dismissInputy0A4Core7
AccountC_AA14Open
            URLContextOSSSbAA012PresentationK0CAA0a11ApplicationM0C7Display01
0NavigationO0CSgyyctF()
```

现在，我们可以简单地手动修改我们感兴趣的分支：

- Objective-C 方法 `application:openURL:options:`：

```
// __handlers__/__AppDelegate_application_openUR_3679fadc.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]"

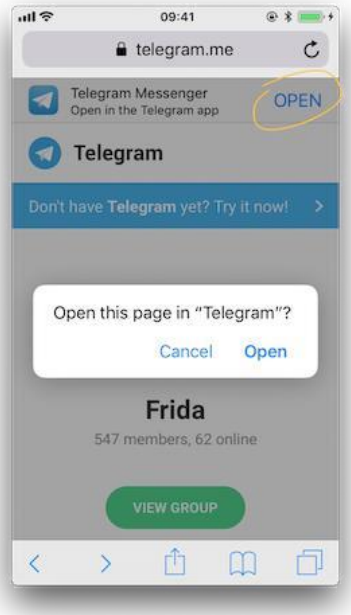
  8144 ms  -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: 0x
10c5ee4c0 origin: 0x0]
  8145 ms     | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 o
ptions: 0x1c0e222c0]
  8145 ms     |     application: <Application: 0x105a59980>
  8145 ms     |     openURL: tg://resolve?domain=fridadotre
  8145 ms     |     options :{
                      UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
                      UIApplicationOpenURLOptionsSourceApplicationKey = "co
m.apple.mobilesafari";
                  }
  8269 ms     |    | TelegramUI.openExternalUrl(account, url, presentationDat
a,
                                          applicationContext, navigationControl
ler, 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，如果再次浏览到“https://telegram.me/fridadotre”，点击“取消”，然后点击页面本身提供的链接（“在 Telegram 应用程序中打开”），而不是通过自定义 URL 方案打开，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*]"

// 点击弹出窗口中的"Open"后

 16374 ms  -[AppDelegate application :0x10556b3c0 openURL :0x1c4ae0080 option
s :0x1c7a28400]
 16374 ms    application :<Application: 0x10556b3c0>
 16374 ms    openURL :tg://resolve?domain=fridadotre
 16374 ms    options :{
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari
```

```
";
}

// 点击弹出窗口中的"Cancel"并且点击页面中的"OPEN"

406575 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c063
d0c0
                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

取消或确认验证的一种方法是通过劫持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)
  }
});
```

如果我们应用劫持并再次调用 URL 方案：

```
$ frida -U iGoat-Swift
```

```
[iPhone::iGoat-Swift]-> var isEqualToString = ObjC.classes.NSString["- isEqua
lToString:"];

                    Interceptor.attach(isEqualToString.implementation, {
                      onEnter: function(args) {
                        var message = ObjC.Object(args[2]);
                        console.log(message)
                      }
                    });
{}
[iPhone::iGoat-Swift]-> openURL("iGoat://?contactNumber=123456789&message=hol
a")
true
nil
```

什么也没有发生。这已经告诉我们，这个方法没有被用于此，因为我们在劫持和推文的文本之间找不到任何像 `OWASP.iGoat-Swift` 或 `com.apple.mobilesafari` 这样的应用程序包的字符串。然而，考虑到我们只是在探测一种方法，应用程序可能使用其他方法进行比较。

6.7.3.3.4. 进行 URL 方案模糊测试

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测试器，例如： 使用 Python，然后调用 Frida RPC 中的 `openURL`。 该模糊测试器应执行以下操作：

- 生成有效载荷。
- 用于每个调用 openURL 的地方。
- 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载荷的模糊测试字典。

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，看看一个对 iGoat-Swift 应用进行模糊测试的例子（基于 iOS 11.1.2）。

在运行模糊测试器之前，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}`。

```
$ frida -U SpringBoard -l ios-url-scheme-fuzzing.js
[iPhone::SpringBoard]-> fuzz("iGoat", "iGoat://?contactNumber={0}&message={0}
")
Watching for crashes from iGoat...
No logs were moved.
Opened URL: iGoat://?contactNumber=0&message=0
OK!
Opened URL: iGoat://?contactNumber=1&message=1
OK!
Opened URL: iGoat://?contactNumber=-1&message=-1
OK!
Opened URL: iGoat://?contactNumber=null&message=null
OK!
Opened URL: iGoat://?contactNumber=nil&message=nil
OK!
Opened URL: iGoat://?contactNumber=99999999999999999999999999999999999
&message=99999999999999999999999999999999999
OK!
Opened URL: iGoat://?contactNumber=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
...
&message=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
...
OK!
Opened URL: iGoat://?contactNumber=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
...
&message=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA
...
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不是这种情况。我们可以在`/private/var/mobile/Library/Logs/CrashReporter` 或 /tmp 中检查崩溃报告（如果已被脚本移动)。

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 开始弃用 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（即防止用户浏览到任意 URL）和自定义的合适选择。`WKWebView` 还通过 Nitro JavaScript 引擎[#thiel2]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 属性可用于验证仅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 WKWebView（和 UIWebView）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 公开原生方法”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（“分享”）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，`SFSafariViewControllers` 不在“静态和动态分析”部分的范围内。

6.7.4.1.4. Safari 网页检查器

在 iOS 上启用 Safari Web 检查器可以让你从 macOS 设备上远程检查 WebView 的内容，而且不需要越狱的 iOS 设备。启用 Safari Web 检查在使用 JavaScript 桥接暴露原生 API 的应用程序中特别有趣，例如在混合应用程序中。

要激活 Web 检查，你必须遵循以下步骤：

1. 在 iOS 设备上打开设置应用程序。进入 Safari 浏览器->高级，并切换到 Web 检查器。
2. 在 macOS 设备上，打开 Safari 浏览器：在菜单栏中，进入 Safari 浏览器->偏好->高级，启用在菜单栏中显示开发菜单。
3. 将你的 iOS 设备连接到 macOS 设备并解锁：iOS 设备的名称应该出现在开发菜单中。
4. （如果尚未信任）在 macOS 的 Safari 上，进入开发菜单，点击 iOS 设备名称，然后点击 "用于开发 "并启用信任。

要打开 Web 检查器并调试一个 WebView。

1. 在 iOS 中，打开应用程序并导航到应包含 WebView 的屏幕。
2. 在 macOS Safari 中，进入 "开发"->"iOS 设备名称"，你应该看到 WebView 基于内容的名称。点击它，打开 Web 检查器。

现在，你能够像在桌面浏览器上调试普通网页一样调试 WebView 了。

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。

- 识别 WebView 的使用。
- 测试 JavaScript 配置。
- 测试混合内容。
- 测试 WebView URI 操纵

6.7.4.2.1. 识别 WebView 的使用

通过在 Xcode 中搜索来查找上述 WebView 类的使用。

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$"
490 0x0002fef3 0x10002fef3   9  10 (5.__TEXT.__cstring) ascii WKWebView
625 0x00031670 0x100031670  17  18 (5.__TEXT.__cstring) ascii unwindToWKWebVi
ew
904 0x0003c960 0x0003c960  24  25 () ascii @_OBJC_CLASS_$_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_So0aB13ConfigurationC13conf
igurationtcfC
0x5d97a 79 78 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13conf
igurationtcfcTO
0x6b5d5 77 76 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13conf
igurationtcfC
0x6c3fa 79 78 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13conf
igurationtcfcTO
```

您也可以将其符号重组：

```
$ xcrun swift-demangle __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13Configurati
onC13configurationtcfcTO

---> @nonobjc __C.WKWebView.init(frame: __C_Synthesized.CGRect,
                                 configuration: __C.WKWebViewConfiguration) ->
 __C.WKWebView
```

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 javaScrip
tEnabled
392 0x0002f2d9 0x10002f2d9  21  22 (4.__TEXT.__objc_methname) ascii setJavaSc
riptEnabled:
```

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 WKWebView 的更多信息，请参见 WKUserContentController 和 WKUserScript。

6.7.4.2.3. 测试混合内容

与 UIWebView 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。[#thiel2]（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。在源码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>并不总是出现显示混合内容问题。在 MDN Web 文档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。
- 验证是否仅允许安全内容。

通过执行动态插桩，可以识别 WebView 并在运行时获取其所有属性。这将在您没有原始源代码时非常有用。

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 WKWebView 的 WebView，为了完整起见，我们还打印了相关的 URL。

为了确保您能够在堆中找到 WebView 的实例，请首先确保浏览到找到的 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

# 复制代码并等待...

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/..."
    );
} }
```

现在我们按 q 退出并打开另一个 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

# 复制代码并等待...
```

```
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 JavaScript 。为此，请验证 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().javaSc
riptEnabled());
//...
  }
});
```

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. 验证是否仅允许安全内容

UIWebView 对此不提供方法。但是，您可以通过调用每个 UIWebView 实例的 request 方法来检查系统是否启用了“Upgrade-Insecure-Requests”CSP（内容安全策略）（从 iOS 10 开

始，“Upgrade-Insecure-Requests”应该可用，其包括 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
```

6.7.4.3.4. 测试 WebView URI 操纵

确保 WebView 的 URI 不能被用户操作，以加载 WebView 运行所需以外的其他类型的资源。当 WebView 的内容从本地文件系统加载时，这可能特别危险，会允许用户在应用程序中浏览其他资源。

6.7.5. 测试 WebView 协议处理程序（MSTG-PLATFORM-6）

6.7.5.1. 概述

有几种默认方案在 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，则用户应不能影响文件名或用于加载文件的路径，并且用户也应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/JavaScript 文件的校验和，并在应用程序启动时检查它们。将 JavaScript 文件最小化 （"Minification (programming)"），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，则用户应不能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` 通过过时的方法 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 `UIWebView`，而改用 `WKWebView`。

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/scenario
2.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：shared preferences 中。

当使用 `WKWebView` 时，Apple 建议使用 loadHTMLString:baseURL:或 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] i
nit];

    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84) configuration:con
figuration];
    self.webView.navigationDelegate = self;
    [self.view addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_fil
e" ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
                                   encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceU
RL];
}
```

Swift 中的例子，来自"我的浏览器在哪里？":

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/WKWebView/scenario
2.html", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding:
 .utf8)
    wkWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

如果仅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，建议进行动态分析，以确保这实际上是在使用的，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 `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: .user
DomainMask)[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:allowingReadAc
cessToURL:
```

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 `WKWebView`：

- 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）：它使运行在 file://方案 URL 内容中的 JavaScript 能够访问来自其他 file://方案 URL 的内容。。
- `allowUniversalAccessFromFileURLs`（`WKWebViewConfiguration`，默认为 false）：它使在 file://方案 URL 内容中运行的 JavaScript 可以访问任何来源内容。

例如：可以通过执行以下操作来设置未记录的属性 `allowFileAccessFromFileURLs`：

.

```
[webView.configuration.preferences                                    setValue:@YES
forKey:@"allowFileAccessFromFileURLs"];
```

Swift:

```
webView.configuration.preferences.setValue(true, forKey: "allowFileAccessFrom
FileURLs")
```

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，但不是电话号码，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，请使用格式检测元标记（`<meta name = "format-detection" content = "telephone=no">`）。在 Apple 开发者文档中可以找到一个示例。 然后应使用电话链接（例如：`<a href="tel:1-408-555-5555">1-408-555-5555</a>`）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通过使用动态插桩将自己的 JavaScript 注入 WebView。尝试访问本地存储以及可能暴露给 JavaScript 内容的任何方法和属性。

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 防御备忘录和“iOS 网络通信”章节。

对于本节涉及的问题，我们将学习：

- 检查 WebView 如何加载。
- 确定 WebView 文件访问。

6.7.5.3.1. 检查 WebView 如何加载

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“场景 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 `WKWebView` 的 `evaluateJavaScript:completionHandler`。

作为有关 UIWebViews 的附加说明，如果从 baseURL 设置为 nil 的 UIWebView 获取有效来源，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 UIWebView，而改用 WKWebView。

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，只需浏览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（UIWebVIew 始终允许文件访问）。

我们继续使用示例“我的浏览器在哪里？”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().javaS
criptEnabled());
    console.log('allowFileAccessFromFileURLs: ',
            wk.configuration().preferences().valueForKey_('allowFileAccessFro
mFileURLs').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_inspecto
r.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_inspecto
r.js
...

allowFileAccessFromFileURLs:  1
```

6.7.6. 确定是否通过 WebView 公开原生方法（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 运行时与原生 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：通过成功的跨站脚本攻击）。

6.7.6.2. 静态分析

`UIWebView` 和 `WKWebView` 都提供了 WebView 与原生应用程序之间的通信方式。在 WebView 中运行的恶意 JavaScript 也可以访问暴露给 WebView JavaScript 引擎的任何重要数据或原生功能。

6.7.6.2.1. 测试 UIWebView JavaScript 到原生网桥

原生代码和 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。
- **JSExport 协议**：在 JSExport 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原生环境中。

请注意，JavaScript 代码只能访问 JSExport 协议中定义的类成员。

查找将原生对象映射到与 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 到原生桥接

`WKWebView` 中的 JavaScript 代码仍然可以将消息发送回原生应用程序，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 对象序列化为原生 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 到原生桥接，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，我们看到 JavaScript 桥接已启用：

```
func enableJavaScriptBridge(_ enabled: Bool) {
    options_dict["javaScriptBridge"]?.value = enabled
    let userContentController = wkWebViewConfiguration.userContentController
    userContentController.removeScriptMessageHandler(forName: "javaScriptBrid
ge")

    if enabled {
            let javaScriptBridgeMessageHandler = JavaScriptBridgeMessageHandl
er()
            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(["multiplyNumb
ers", value1, value2]);
}
```

调用的方法存在于 `JavaScriptBridgeMessageHandler.swift`：

```
class JavaScriptBridgeMessageHandler: NSObject, WKScriptMessageHandler {

//...

case "multiplyNumbers":

        let arg1 = Double(messageArray[1])!
        let arg2 = Double(messageArray[2])!
        result = String(arg1 * arg2)
//...

let javaScriptCallBack = "javascriptBridgeCallBack('\(functionFromJS)','\(res
ult)')"
message.webView?.evaluateJavaScript(javaScriptCallBack, completionHandler: ni
l)
```

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 WebView 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，通过执行逆向工程来获取密钥值是微不足道的（密钥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密钥存储。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 有效载荷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 不安全加载的（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:用于 `UIWebView` 以及 `evaluateJavaScript:completionHandler:`用于 `WKWebView`）来执行动态插桩并注入 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载荷，这些有效载荷通过将其写入 WebView 的“result 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助手：

请参见第 156 页[#thiel2]中暴露于 WebView 的易受攻击的 iOS 应用和函数的另一个示例。

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 `NSObject` 的对象编码和解码协议: `NSCoding` 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 `Data` 与 `NSData` 或其可变对象: `NSMutableData` 相同。`NSCoding` 协议声明了两个必须实现的方法，以便对其实例变量进行编码/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如下所示的初始化。

```
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额外使用 HMAC 校验。

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，无法轻易移除。

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/a
rchive") as?
    CustomPoint else { return nil }
```

解码键控存档时，由于值是按名称请求的，所以值可以不按顺序解码或根本不解码。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键。

请注意，有机密数据的情况下，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. Codable

随着 Swift 4 的到来，Codable 类型别名出现了：它是 Decodable 和 Encodable 协议的组合。String、Int、 Double、Date、Data 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 开发者文档。 Codable 可以轻松地编码/解码为各种表示形式：使用 NSCoding/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 Codable

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
- JSONModel library
- SwiftyJSON library
- ObjectMapper library
- JSONKit
- JSONModel
- YYModel
- SBJson 5
- Unbox
- Gloss
- Mapper
- JASON
- Arrow

这些库对某些版本的 Swift 和 Objective-C 的支持、返回的是否可变的结果，速度，内存消耗和实际的库大小都有所不同。同样，请注意不可更改性：机密信息无法轻易从内存中删除。

接下来，Apple 通过将 `Codable` 与 `JSONEncoder` 和 `JSONDecoder` 组合在一起，直接提供对 JSON 编码和解码的支持:

```
struct CustomPointStruct: Codable {
    var point: Double
    var name: String
}

let encoder = JSONEncoder()
encoder.outputFormatting = .prettyPrinted

let test = CustomPointStruct(point: 10, name: "test")
let data = try encoder.encode(test)
let stringData = String(data: data, encoding: .utf8)

// stringData = Optional ({
// "point" : 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、HMAC)。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 Codable

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 plist)。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 = NSUserDefaults.standardUserDefaults().objectForKey("customPoint
") as? NSData {
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data)
}
```

在第一个示例中，使用 NSUserDefaults，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 point: Double
        var name: String
    }

    var points: [CustomPointStruct] = [
        CustomPointStruct(point: 1, name: "test"),
        CustomPointStruct(point: 2, name: "test"),
        CustomPointStruct(point: 3, name: "test"),
    ]

    UserDefaults.standard.set(try? PropertyListEncoder().encode(points), forK
ey: "points")
    if let data = UserDefaults.standard.value(forKey: "points") as? Data {
        let points2 = try? PropertyListDecoder().decode([CustomPointStruct].s
elf, 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：

- Fuzi
- 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 预防备忘录。除了库之外，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（CoreData 和 Realm）

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，Apple 开发者文档中对此进行了详细说明。如 Apple 的持久存储类型和行为文档中所述，它支持各种存储后端。Apple 推荐的存储后端的问题是，没有任何类型的数据存储是加密的，也没有完整性检查。因此，在处理机密数据的情况下，必须采取其他措施。可以在 iMas 项目中找到替代方法，该项目确实提供了开箱即用的加密功能。

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，通过二进制数据格式来序列化结构化数据。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

所有不同类型的对象持久化都有以下关注点：

- 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 数据存储”章节。
- 确保在反序列化对象中的数据被实际使用之前经过仔细验证（例如：不可能对业务/应用逻辑进行利用）。
- 请勿使用运行时引用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）。
- 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 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、objection 查看序列化方法的处理方式（例如：应用程序崩溃或者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更新证书固定时，强制更新对于公钥固定（有关更多详细信息，请参见“测试网络”通讯）非常有帮助。同样，可以通过强制更新轻松修补漏洞。但是，iOS 面临的挑战是，Apple 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://itunes.apple.com/lookup\?id\<BundleId>`或第三方库，例如：Siren 和 react-

native-appstore-version-checker 查找应用程序的属性。这些实现大多需要 API 提供的某个特定版本，或者只是 "App Store 中的最新版本"，这意味着用户可能对即使确实没有更新的业务/安全需求，也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，不要忘记针对你的 API 测试你的应用程序的旧版本和/或使用适当的 API 版本。

6.7.8.1. 静态分析

首先查看是否有更新机制：如果还没有，可能意味着不能强迫用户更新。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，以确保更新机制不能被绕过。

6.7.8.2. 动态分析

为了测试正确的更新：尝试下载一个有安全漏洞的旧版本的应用程序，可以通过开发商的发布渠道或使用第三方应用商店。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攻击修改应用的版本号，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-flawhttps://medium.com/iOS-os-x-development/security-flaw-with-uiwebview-95bbd8508e3cwith-uiwebview-95bbd8508e3c
- Learning about Universal Links and Fuzzing URL Schemes on iOS with Frida - https://grepharder.github.io/blog/0x03_learning_about_universal_links_and_fuzzing_url_sche mes_on_iOS_with_frida.html

6.7.9.1. 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。”

6.7.9.2. 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_json_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

应用程序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、安装在未越狱设备上或提交到 App Store 之前，它必须用 Apple 颁发的证书签名。有关如何请求证书和应用程序代码签名的更多信息，请查看应用程序分发指南。

6.8.1.2. 静态分析

你必须确保应用程序使用的是最新的代码签名格式。你可以用 codeign 从应用程序的.app 文件中获取签名证书信息。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。浏览到 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=embe
dded
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
```

正如 Apple 文档中所描述的那样，有多种方式来分发你的应用程序，其中包括使用 App Store 或通过 Apple 业务管理器进行自定义或内部分发。如果是内部分发方案，请确保在应用程序签名分发时不使用 ad hoc 证书。

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，从 Xcode5 开始取代了 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 还是 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，以检查环境中的参数：

- 选择项目的构建设置。
- 在“Apple LVM - 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”/“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已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制，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

作为一种好的实践，编译后的二进制文件应尽可能少地提供解释性信息。额外元数据的存在，如调试符号，可能会提供关于代码的有价值的信息，例如，函数名称泄露了一个函数的作用信息。这些元数据在执行二进制文件时是不需要的，因此在发布构建时丢弃它是安全的，这可以通过使用适当的编译器配置来完成。作为测试人员，你应该检查所有与应用程序一起交付的二进制文件，并确保没有调试符号存在（至少是那些透露了任何有价值的代码信息的符号）。

当一个 iOS 应用被编译时，编译器会为应用中的每个二进制文件（主应用可执行文件、框架和应用扩展）生成一个调试符号列表。这些符号包括类名、全局变量、方法和函数名，它们被映射到特定的文件和定义的行号上。应用程序的调试构建默认将调试符号放在已编译的二进制文件中，而应用程序的发布构建则将它们放在配套的调试符号文件（dSYM）中，以减少发布应用程序的大小。

6.8.3.2. 静态分析

为了验证调试符号的存在，你可以使用 binutils 的 objdump 或 llvm-objdump 来检查所有的应用程序二进制文件。

在下面的片段中，我们在 TargetApp（iOS 主应用程序的可执行文件）上运行 objdump，以显示一个包含调试符号的二进制文件的典型输出，这些符号被标记为 d（调试）标志。查看 objdump 手册，了解其他各种符号标志字符的信息。

```
$ objdump --syms TargetApp

0000000100007dc8 l    d  *UND* -[ViewController handleSubmitButton:]
000000010000809c l    d  *UND* -[ViewController touchesBegan:withEvent:]
0000000100008158 l    d  *UND* -[ViewController viewDidLoad]
```

```
...
000000010000916c l     d  *UND* _disable_gdb
00000001000091d8 l     d  *UND* _detect_injected_dylds
00000001000092a4 l     d  *UND* _isDebugged
...
```

为了防止包含调试符号，请通过 XCode 项目的构建设置将 " Strip Debug Symbols During Copy 复制过程中剥离调试符号 "设置为 "是"。剥离调试符号不仅会减少二进制文件的大小，还会增加逆向工程的难度。

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，开发人员通常设置调试代码，例如：详细的日志记录语句（使用 NSLog、 println、print、dump 和 debugPrint），显示来自 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. 搜索以下打印函数代码：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函数
4. 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。
5. 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 如何使用预处理程序宏的变化：

```
#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参数。 请注意，以下函数(这些函数可以让您确定应用程序是否在 Swift 2.1.版本配置中构建)是不推荐的，因为 Xcode8 和 Swift 3 不支持这些功能：

- `_isDebugAssertConfiguration`
- `_isReleaseAssertConfiguration`
- `_isFastAssertConfiguration`.

根据应用程序的设置，可能会有更多的日志记录函数。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，在应用程序的调试版本中加入调试语句不应该是一个问题，只要确保调试语句永远不会出现在应用程序的发布版本中。

在 Objective-C 中，开发人员可以使用预处理程序宏来筛选出调试代码：

```
#ifdef DEBUG
    // Debug-only code
#endif
```

在 Swift 2 中(使用 Xcode7)，必须为每个目标设置自定义编译器参数，编译器参数必须以“-D”开头。因此，在设置调试参数 DMSTG-DEBUG 时，可以使用以下注释：

```
#if MSTG-DEBUG
    // Debug-only code
#endif
```

在 Swift 3 中(使用 Xcode8)，您可以在 Build 设置或 SWIFT 编译器-自定义参数中设置活动编译条件。Swift 3 不使用预处理程序，而是根据定义的条件使用条件性编译块：

```
#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. 将设备连接到 Mac 上，通过 Xcode 在设备上运行应用程序，并在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，这加快了开发速度，因为开发人员只需要写更少的代码来解决一个问题。然而，第三方库可能包含漏洞、不兼容的许可或恶意内容。此外，组织和开发人员很难管理应用程序的依赖关系，包括监测库的发布和应用可用的安全补丁。

有三种广泛使用的软件包管理工具 Swift Package Manager、Carthage 和 CocoaPods。

- Swift Package Manager 是开源的，包含在 Swift 语言中，集成在 Xcode 中（从 Xcode 11 开始），支持 Swift、Objective-C、Objective-C++、C 和 C++包。它是用 Swift 编写的，是去中心化的，使用 Package.swift 文件来记录和管理项目的依赖性。
- Carthage 是开源的，可用于 Swift 和 Objective-C 软件包。它是用 Swift 编写的，是去中心化的，使用 Cartfile 文件来记录和管理项目的依赖性。
- CocoaPods 是开源的，可用于 Swift 和 Objective-C 包。它是用 Ruby 编写的，利用一个中心化的包注册表来处理公共和私有包，并使用 Podfile 文件来记录和管理项目的依赖性。

有两种库类型：

- 没有（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`。

这些库可能有导致不必要的副作用：

- 一个库可能包含一个漏洞，这将使应用程序变得脆弱。一个很好的例子是 AFNetworking 2.5.1 版本，它包含一个禁用证书验证的漏洞。这个漏洞将允许攻击者对使用该库连接到其 API 的应用程序实施中间人攻击。
- 一个库可能不再被维护或几乎不被使用，这就是为什么没有漏洞被报告和/或修复。这可能导致在你的应用程序中通过库有不良的和/或脆弱的代码。
- 一个库可能使用一个许可证，如 LGPL2.1，它要求应用程序的作者为那些使用该应用程序并要求深入了解其源代码的人提供访问权限。事实上，应用程序应该被允许在修改其源代码的情况下进行再分发。这可能会危及到应用程序的知识产权（IP）。

请注意，这个问题可能在多个层面上存在。当你使用 webview，并在 webview 中运行 JavaScript 时，JavaScript 库也会有这些问题。对于 Cordova、React-native 和 Xamarin 应用程序的插件/库也是如此。

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包含漏洞，最好检查 CocoaPods 或 Carthage 安装的依赖项。

6.8.5.2.1.1 Swift 包管理器

如果使用 Swift 包管理器来管理第三方依赖，可以采取以下步骤来分析第三方库是否存在漏洞。

首先，在 Package.swift 文件所在的项目根目录下，输入

```
swift build
```

接下来，检查 Package.resolved 文件以了解实际使用的版本，并检查给定的库是否存在已知的漏洞。

你可以利用 OWASP Dependency-Check 的实验性 Swift 包管理器分析器来识别所有依赖关系的通用平台枚举（CPE）命名方案和任何相应的通用漏洞和暴露（CVE）条目。扫描应用程序的 Package.swift 文件，并通过以下命令生成一份已知脆弱库的报告。

```
dependency-check  --enableExperimental --out . --scan Package.swift
```

6.8.5.2.1.2 CocoaPods

如果用 CocoaPods 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，既然已经建立了依赖关系树，你可以通过运行以下命令来创建一个依赖关系及其版本的概览：

```
sudo gem install cocoapods-dependencies
pod dependencies
```

上述步骤的结果现在可以作为输入，用于搜索不同的漏洞源，寻找已知的漏洞。

注：

1. 如果开发者使用一个.podspec 文件以其自身的支持库打包所有的依赖性，那么这个.podspec 文件可以用实验性的 CocoaPods podspec 检查器来检查。
2. 如果项目使用 CocaoPods 结合 Objective-C，可以使用 SourceClear。
3. 使用基于 HTTP 的链接而不是 HTTPS 的 CocoaPods 可能会在下载依赖性的过程中允许中间人攻击，使攻击者可以用其他内容替换（部分）库。因此，请始终使用 HTTPS。

你可以利用 OWASP Dependency-Check 的实验性 CocoaPods 分析器来识别所有依赖关系的通用平台枚举（CPE）命名方案和任何相应的通用漏洞和暴露（CVE）条目。扫描应用程序的 *.podspec 和/或 Podfile.lock 文件，用以下命令生成已知脆弱库的报告。

```
dependency-check  --enableExperimental --out . --scan Podfile.lock
```

6.8.5.2.1.3 Carthage

如果用 Carthage 管理第三方依赖，则可以采取以下步骤分析第三方库的漏洞：

首先，在 Cartfile 所在的项目根目录下，输入：

```
brew install carthage
carthage update --platform iOS
```

接下来，检查 Cartfile.resolved 的实际使用版本，并检查所给的库是否有已知的漏洞。

注意，编写这一章的时候，作者还没有发现对基于 Carthage 的依赖性自动分析工具。至少，这个需求已经提交给 OWASP DependencyCheck 工具，但还没有实现（见 GitHub 问题）。

6.8.5.2.1.4 发现的库漏洞

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过漏洞的版本。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？看看是否有修补过的版本，其中的漏洞被修复了。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？然后尝试寻找一个修复了该漏洞的替代品。

如果框架作为链接库手动添加：

1. 打开 xcodeproj 文件，检查项目属性。
2. 转到“**Build Phases** 构建阶段”选项卡，检查任何库的“**Link Binary With Libraries** 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（如果使用 Objective-C），否则搜索 Swift 文件中已知库的已知方法名称。

接下来，请注意，对于混合应用程序，你必须用 RetireJS 检查 JavaScript 的依赖性。同样，对于 Xamarin，你必须检查 C#的依赖性。

最后，如果该应用程序是一个高风险的应用程序，你最终将手动审核该库。在这种情况下，对原生代码有特定的要求，这些要求与 MASVS 对整个应用的要求相似。除此以外，最好是审查是否应用了软件工程的所有最佳实践。

6.8.5.2.2. 检测应用程序库使用的许可证

为了确保不违反版权法，最好检查 Swift 包管理器、CocoaPods 或 Carthage 所安装的依赖关系。

6.8.5.2.2.1 Swift 包管理器

当应用程序源可用且使用 Swift 包管理器时，在项目的根目录下执行以下代码，即 Package.swift 文件所在的目录。

```
swift build
```

现在，每一个依赖项的源代码都已经下载到项目中的`/.build/checkouts/`文件夹。在这里你可以找到每个库在各自文件夹中的许可证。

6.8.5.2.2.2 CocoaPods

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，所有的库都安装在这个文件夹中，每个库都有自己的文件夹。现在你可以通过检查每个文件夹中的许可证文件来检查每个库的许可证。

6.8.5.2.2.3 Carthage

当应用程序源可用并且使用 Carthage 时，在项目的根目录（Cartfile 所在目录）下执行以下代码：

```
brew install carthage
carthage update --platform iOS
```

现在每个依赖项的源代码都已经下载到项目中的 Carthage/Checkouts 文件夹中。在这里，你可以在各自的文件夹中找到每个库的许可证。

6.8.5.2.2.4 库许可证问题

当一个库包含一个许可证，而应用程序的 IP 需要开源时，请检查是否有一个库的替代品可以用来提供类似的功能。

注意：如果是混合应用程序，请检查所使用的构建工具：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

这一部分的动态分析包括两部分：实际的许可证验证和在缺少来源的情况下检查涉及哪些库。

需要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，其中指出第三方库的许可证所要求的版权声明。

6.8.5.3.1. 列出应用程序库

在进行应用程序分析时，重要的是还要分析应用的依赖关系（通常以库或所谓的 iOS 框架的形式），并确保它们不包含任何漏洞。即使你没有源代码，你仍然可以使用 objection、MobSF 或 otool 等工具识别一些应用程序的依赖关系。Objection 是值得推荐的工具，因为它能提供最准确的结果，而且很容易使用。它包含一个与 iOS 包一起工作的模块，它提供两个命令：`list_bundles` 和 `list_frameworks`。

`list_bundles` 命令列出了所有与 Frameworks 无关的应用程序的容器包。输出包含可执行文件的名称、容器的 ID、库的版本和库的路径。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios bundles lis
t_bundles
Executable    Bundle                                          Version  Path
------------  ----------------------------------------  ---------  ---------
----------------------------------
DVIA-v2       com.highaltitudehacks.DVIAswiftv2.develop          2  ...-1F0C-
4DB1-8C39-04ACBFFEE7C8/DVIA-v2.app
CoreGlyphs    com.apple.CoreGlyphs                               1  ...m/Libr
ary/CoreServices/CoreGlyphs.bundle
```

`list_frameworks` 命令列出了应用程序中所有代表 Frameworks 的包。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios bundles lis
t_frameworks
Executable      Bundle                                       Version    Path
--------------  ----------------------------------------  ---------  -------
-----------------------------------
Bolts           org.cocoapods.Bolts                          1.9.0      ...8/DV
IA-v2.app/Frameworks/Bolts.framework
RealmSwift      org.cocoapods.RealmSwift                     4.1.1      ...A-v
2.app/Frameworks/RealmSwift.framework
                                                                        ...yste
m/Library/Frameworks/IOKit.framework
...
```

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。在一个同时用传统 Objective-C 代码和 Swift 代码编写的应用程序中，衔接这两种方法可能会产生问题。

6.8.6.1.1. Objective-C 的异常处理

Objective-C 有两类错误：

NSException `NSException` 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 `NSException` 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，你可以用它来记录这个状态（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` 代码块：

```
@try {
    //在这完成工作
 }

@catch (NSException *e) {
    //从异常恢复
}

@finally {
    //清理
```

请记住，使用 `NSException` 会带来内存管理缺陷：您需要在 finally 块中清理 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 `NSException` 对象升级到 `NSError`。

NSError `NSError` 用于所有其他类型的错误。一些 Cocoa 框架的 API 在其失败回调中提供了错误对象，以防出错；那些不提供错误对象的 API 则通过引用传递一个指向 `NSError` 对象的指针。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 `nil`。如果返回“NO”或“`nil`”，则允许您检查错误、故障的原因。

6.8.6.1.2. Swift 中的异常处理

Swift 中的异常处理（2-5）是相当不同的。try-catch 块不能用于处理 NSException。该块用于处理符合 Error（Swift3）或 ErrorType（Swift2）协议的错误。当 Objective-C 和 Swift 代码在应用程序中混合使用时，这可能是一个挑战。因此，对于用两种语言编写的程序，NSError 比 NSException 更可取。此外，错误处理在 Objective-C 中是可选的，但是 throws 必须在 Swift 中显式处理。为了转换异常抛出，查看 Apple 文档。Result 类型代表了成功或失败，参见 Result，如何在 Swift 5 中使用 Result 和 Swift 中 Result 类型的力量。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 throw 抛出实际错误或 try 执行抛出的方法。包含 try 的方法还需要 throws 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：

```
func doTryExample() {
    do {
        try functionThatThrows(number: 203)
    } catch NumberError.lessThanZero {
        // 处理数字小于 0
    } catch let NumberError.tooLarge(delta) {
        // 处理数过大（delta）
    } catch {
        // 处理任何其他错误
    }
}

enum NumberError: Error {
    case lessThanZero
    case tooLarge(Int)
    case tooSmall(Int)
}

func functionThatThrows(number: Int) throws -> Bool {
    if number < 0 {
        throw NumberError.lessThanZero
    } else if number < 10 {
        throw NumberError.tooSmall(10 - number)
    } else if number > 100 {
        throw NumberError.tooLarge(100 - number)
```

```
    } else {
        return true
    }
}
```

- 将错误作为可选值处理：

```
let x = try? functionThatThrows()
// 在这种情况下，如果出现错误，x 的值为 nil.
```

- 使用 try! 表达式来断言错误不会发生。

- 将通用错误作为一个 Result 返回来处理：

```
enum ErrorType: Error {
    case typeOne
    case typeTwo
}

func functionWithResult(param: String?) -> Result<String, ErrorType> {
    guard let value = param else {
        return .failure(.typeOne)
    }
    return .success(value)
}

func callResultFunction() {
    let result = functionWithResult(param: "OWASP")

    switch result {
    case let .success(value):
        // 处理成功
    case let .failure(error):
        // 处理错误 (error)
    }
}
```

- 使用 Result 类型处理网络和 JSON 解码错误：

  - ```
    struct MSTG: Codable {
        var root: String
        var plugins: [String]
        var structure: MSTGStructure
        var title: String
        var language: String
        var description: String
    }
    ```
    ```

```
struct MSTGStructure: Codable {
    var readme: String
}

enum RequestError: Error {
    case requestError(Error)
    case noData
    case jsonError
}

func getMSTGInfo() {
    guard let url = URL(string: "https://raw.githubusercontent.com/OWASP/
owasp-mastg/master/book.json") else {
        return
    }

    request(url: url) { result in
        switch result {
        case let .success(data):
            // 成功处理 MSTG 数据
            let mstgTitle = data.title
            let mstgDescription = data.description
        case let .failure(error):
            // Handle failure
            switch error {
            case let .requestError(error):
                // 处理请求错误 (error)
            case .noData:
                // 处理响应中没有收到数据
            case .jsonError:
                // 处理 JSON 转换错误
            }
        }
    }
}

func request(url: URL, completion: @escaping (Result<MSTG, RequestError>)
 -> Void) {
    let task = URLSession.shared.dataTask(with: url) { data, _, error in
        if let error = error {
            return completion(.failure(.requestError(error)))
        } else {
            if let data = data {
                let decoder = JSONDecoder()
                guard let response = try? decoder.decode(MSTG.self, from:
 data) else {
                    return completion(.failure(.jsonError))
                }
```

```
                    return completion(.success(response))
                }
            }
        }
        task.resume()
    }
```

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 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/UIApplication` 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。
- 应用程序在处理错误时不会在其 UI 或日志语句中暴露敏感信息，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，如密钥材料和身份认证信息，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。
- 应用程序在处理错误时不会在其 UI 或日志语句中暴露敏感信息，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，如密钥材料和身份认证信息，在执行 `defer` 块时总是被擦除。
- 只有在前面有适当的防护措施的情况下才会使用 try！(以程序化的方式验证用 try！调用的方法不会出错)。

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“测试调试代码和冗长的错误记录”中所描述的。
- 对于用 Objective-C 编写的高风险应用程序：创建一个异常处理程序，删除不应轻易检索的秘密。处理程序可以通过 `NSSetUncaughtExceptionHandler` 设置。
- 避免在 Swift 中使用 `try!`，除非你确定正在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 字段中输入异常值。
- 通过提供异常或引发异常值来测试自定义 URL 方案、剪贴板和其他应用程序间通信控制。
- 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 劫持方法，并为它们提供可能导致被调用者抛出异常的参数。

在大多数情况下，应用程序不应该崩溃。相反，它应该：

- 从错误中恢复或进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遇到内存损坏的问题：首先是原生代码的问题，在通用内存损坏问题部分已经提到。接下来，Objective-C 和 Swift 都有各种不安全的操作，实际封装着原生代码，会产生问题。最后，由于保留了不再使用的对象，Swift 和 Objective-C 的实现都可能导致内存泄漏。

6.8.7.1. 静态分析

是否有原生代码部分？如果是的话：在通用内存损坏部分检查给定的问题。编译时原生代码比较难识别。如果您有源代码，那么您可以看到 C 文件使用.c 源代码文件和.h 头文件，C++使用.cpp 文件和.h 文件。这与 Swift 和 Objective-C 的.swift 和.m 源代码文件有点不同。这些文件可以是源代码的一部分，也可以是第三方库的一部分，注册为框架，以及通过各种工具导入，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：

- 双重释放问题：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 Swift 5，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

有各种工具可以帮助识别 Xcode 中的内存错误，比如 Xcode 8 中引入的 Debug Memory graph 和 Xcode 中的 Allocations and Leaks instrument。

接下来，你可以在测试应用程序时在 Xcode 中启用 `NSAutoreleaseFreedObjectCheckEnabled`、`NSZombieEnabled`、`NSDebugEnabled` 来检查内存释放的速度是否过快或过慢。

有各种写得很好的文章，可以帮助处理内存管理。这些可以在本章的参考列表中找到。

6.8.8. 确保激活了释放安全功能(MSTG-CODE-9)

6.8.8.1. 概述

用于检测二进制保护机制是否存在的测试在很大程度上取决于用于开发应用程序的语言。

尽管 Xcode 默认启用了所有的二进制安全功能，但对旧应用程序进行验证或检查编译器参数错误配置可能与此相关。以下功能是适用的：

- **PIE（位置独立的可执行文件）**：
  - PIE 适用于可执行的二进制文件（Mach-O 类型 MH_EXECUTE）。
  - 然而，它不适用于库（Mach-O 类型 MH_DYLIB）。
- **内存管理：**
  - 纯 Objective-C、Swift 和混合二进制文件都应该启用 ARC（自动引用计数）。
  - 对于 C/C++库，开发者有责任进行适当的手动内存管理。请参阅 "内存破坏漏洞（MSTG-CODE-8）"。
- **栈溢出保护**：对于完全的 Objective-C 二进制文件，这应该总是被启用。由于 Swift 被设计为内存安全，如果一个库完全是用 Swift 写的，并且没有启用 stack canary，风险也是最小的。

了解更多：

- OS X ABI Mach-O 文件格式参考
- 关于 iOS 的二进制保护
- iOS 和 iPadOS 中运行过程的安全性
- Mach-O 编程话题--位置无关的代码

检测这些保护机制是否存在的测试在很大程度上取决于开发应用程序所用的语言。例如，现有的检测 stack canary 是否存在的技术对纯 Swift 应用程序不起作用。

6.8.8.1.1. Xcode 项目设置

6.8.8.1.1.1 栈溢出保护

在 iOS 应用程序中启用堆栈保护的步骤：

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”部分中选中“-fstack-protector-all”选项。
3. 确保启用了地址无关执行（PIE）支持。

6.8.8.1.1.2 PIE 保护

将 iOS 应用程序构建为启用 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 iOS 4.3 或更高版本。
3. 确保“Generate Position-Dependent Code 生成地址依赖代码”设置为其默认值(NO)。
4. 确保“Don’ t Create Position Independent Executables 不要创建地址无关执行”设置为其默认值（NO）。

6.8.8.1.1.3 ARC 保护

Swiftc 编译器为 Swift 应用程序自动启用了 ARC。然而，对于 Objective-C 应用程序，你必须确保按照以下步骤来启用它：

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 编译地址无关执行。

6.8.8.2. 静态分析

你可以使用 otool 来检查上述的二进制安全特性。在这些示例中，所有的功能都是启用的。

- 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
```

  输出显示，PIE 的 Mach-O 标志被设置。这个检查适用于所有--Objective-C、Swift 和混合应用程序，但只适用于主可执行程序。

- 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
```

  在上面的输出中，__stack_chk_fail 的存在表明堆栈指示变量正在被使用。这个检查适用于纯 Objective-C 和混合应用程序，但不一定适用于纯 Swift 应用程序（也就是说，如果显示为禁用，也是可以的，因为 Swift 在设计上是内存安全的）。

- ARC:

```
$ 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]
```

这个检查适用于所有情况，包括自动启用的纯 Swift 应用程序。

6.8.8.3. 动态分析

这些检查可以使用 objection 动态地进行。这里有一个例子：

```
com.yourcompany.PPClient on (iPhone: 13.2.3) [usb] # ios info binary
Name                  Type     Encrypted    PIE    ARC    Canary    Stack Exe
c    RootSafe
--------------------  -------  -----------  -----  -----  --------  ---------
---  ----------
PayPal                execute  True         True   True   True      False
     False
CardinalMobile        dylib    False        False  True   True      False
     False
FraudForce            dylib    False        False  True   True      False
     False
...
```

6.8.9. 参考文献

- 代码设计- https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html
- 构建你的应用程序以包括调试信息- https://developer.apple.com/documentation/xcode/building-your-app-to-include-debugging-information

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/MemoryMg mt/Articles/MemoryMgmt.html
- https://medium.com/zendesk-engineering/iOS-identifying-memory-leaks-using-the-xcodehttps://medium.com/zendesk-engineering/iOS-identifying-memory-leaks-using-the-xcode-memory-graph-debugger-e84f097b9d15memory-graph-debugger-e84f097b9d15

6.8.9.2. OWASP MASVS

- MSTG-CODE-1: "该应用程序已签署并配备了有效的证书"
- MSTG-CODE-2: "该应用程序是以发布模式构建的，其设置适合于发布构建（例如，不可调试）。"
- MSTG-CODE-3: "调试符号已从本地二进制文件中删除。"
- MSTG-CODE-4: "调试代码已被删除，应用程序不会记录冗长的错误或调试信息。"
- MSTG-CODE-5: "移动应用程序使用的所有第三方组件，如库和框架，都被识别出来，并检查是否有已知的漏洞。"
- MSTG-CODE-6: "该应用程序捕捉并处理可能出现的异常情况。"
- MSTG-CODE-8: "在非托管代码中，内存的分配、释放和使用是安全的。"
- MSTG-CODE-9: "工具链提供的释放安全功能，如字节码最小化、堆栈保护、PIE 支持和自动引用计数，被激活。"

6.9. iOS 系统上的篡改和逆向工程

6.9.1. 逆向工程

iOS 的逆向工程是一个大杂烩。一方面，用 Objective-C 和 Swift 编程的应用程序可以被很好地反汇编。在 Objective-C 中，对象方法是通过被称为 "选择器 "的动态函数指针来调用的，这些指针在运行时通过名称来解析。运行时名称解析的好处是，这些名称需要在最终的二进制文件中保持完整，使反汇编更可读。不幸的是，这也意味着在反汇编中没有方法之间的直接交叉引用，构建一个流程图是具有挑战性的。

在本指南中，我们将介绍静态和动态分析以及插桩。在本章中，我们使用了 iOS 版本的 OWASP UnCrackable 应用程序，因此，如果您计划练习示例，请从 MSTG 源下载它们。

6.9.1.1. 反汇编和反编译

因为 Objective-C 和 Swift 在本质上是不同的，编写应用程序的编程语言会影响逆向工程的可能性。例如，Objective-C 允许在运行时改变方法的调用。这使得劫持其他应用程序函数（一种被 Cycript 和其他逆向工程工具大量使用的技术）变得容易。在 Swift 中，这种 "method

swizzling "的实现方式并不相同，而这种差异使得这种技术在 Swift 中比在 Objective-C 中更难执行。

在 iOS 上，所有的应用程序代码（包括 Swift 和 Objective-C）都被编译为机器代码（例如 ARM）。因此，为了分析 iOS 应用程序，需要一个反汇编程序。

如果你想从 App Store 反汇编一个应用程序，请先移除 Fairplay DRM。"iOS 基本安全测试 " 一章中的 "获取应用程序二进制文件 "一节解释了如何操作。

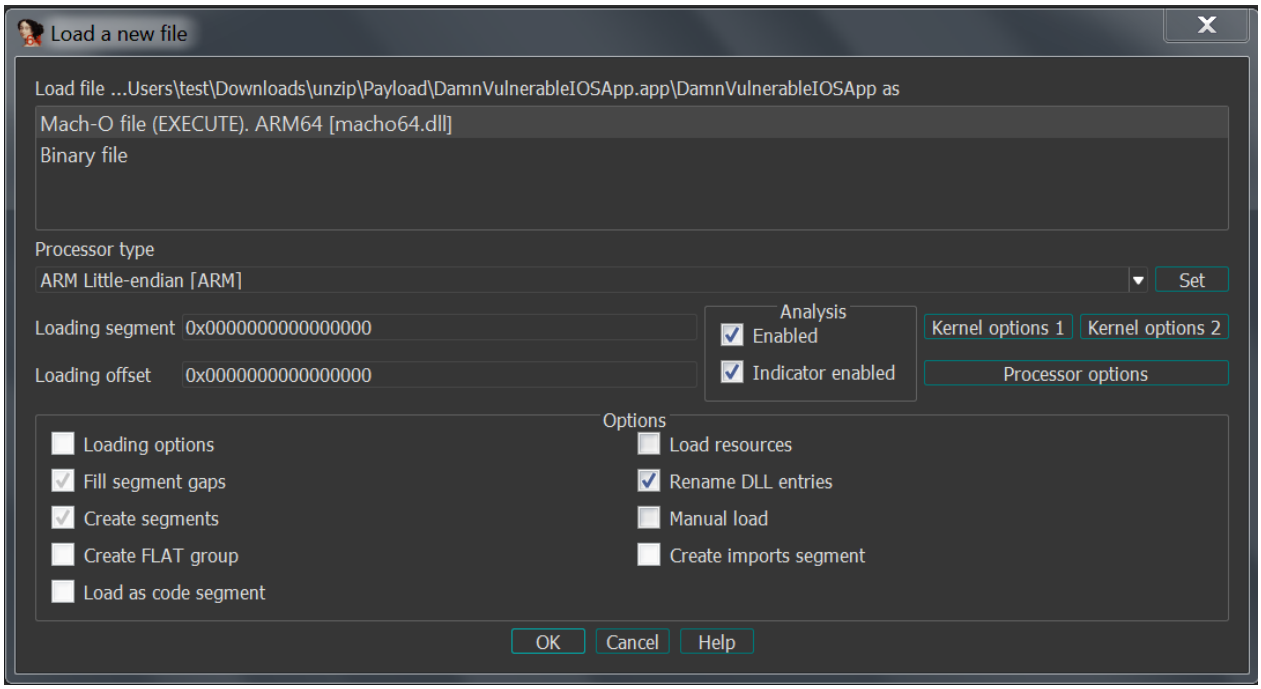
在本节中，术语 "应用程序二进制文件 "指的是应用程序包中的 Macho-O 文件，其中包含编译后的代码，不应该与应用程序包——IPA 文件相混淆。关于 IPA 文件组成的更多细节，请参见 " 基础 iOS 安全测试 "一章中的 "探索应用程序包"。

6.9.1.1.1. 使用 IDA Pro 反汇编

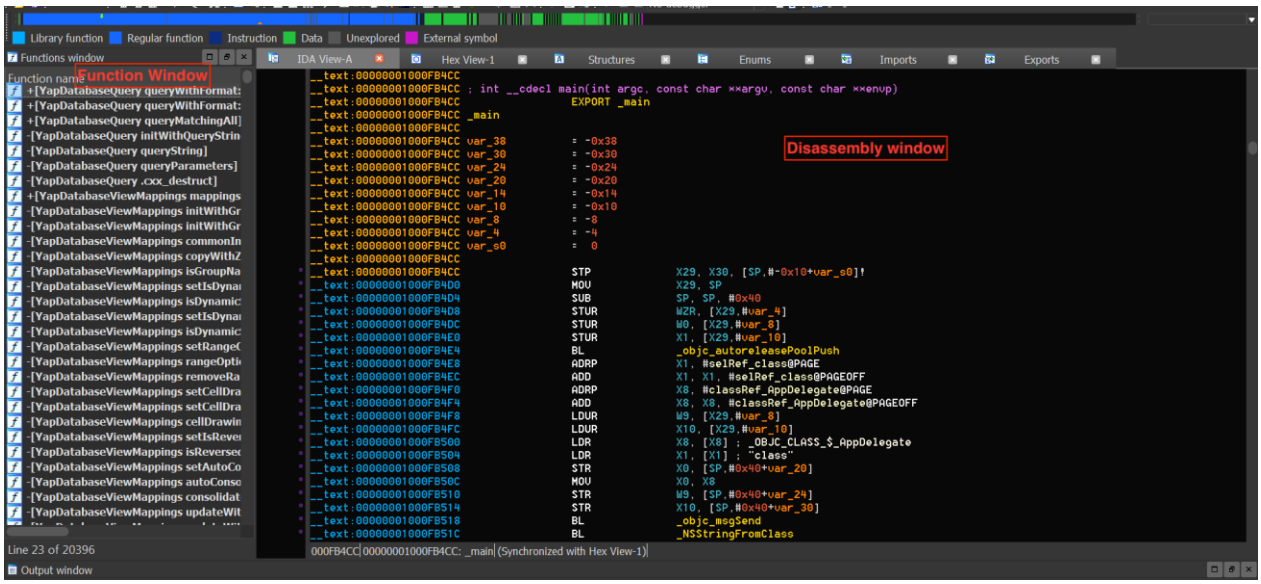
如果你有 IDA Pro 的许可证，你也可以用 IDA Pro 来分析应用程序的二进制。

不幸的是，IDA 的免费版本不支持 ARM 处理器类型。

要开始使用，只需在 IDA Pro 中打开应用程序二进制文件。



打开文件后，IDA Pro 会进行自动分析，根据二进制文件的大小，这可能需要一段时间。一旦自动分析完成，你可以在 IDA View (Disassembly)窗口中浏览反汇编，并在 Functions 窗口中探索函数，两者都在下面的屏幕截图中显示。



普通的 IDA Pro 许可证默认不包括反编译器，需要额外的 Hex-Rays 反编译器的许可证，这很昂贵。相比之下，Ghidra 带有一个非常有能力的免费内置反编译器，使其成为用于逆向工程的一个引人注目的选择。

如果你有一个普通的 IDA Pro 许可证，并且不想购买 Hex-Rays 反编译器，你可以通过安装 IDA Pro 的 GhIDA 插件来使用 Ghidra 的反编译器。

本章的大部分内容适用于用 Objective-C 编写的应用程序或具有桥接类型，即与 Swift 和 Objective-C 都兼容的类型。大多数与 Objective-C 配合良好的工具的 Swift 兼容性正在得到改善。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是使用原始的 Xcode 项目文件。理想情况下，您将能够编译和调试应用程序，以快速识别源代码的任何潜在问题。

在无法获得原始源代码的情况下，对 iOS 应用程序进行黑盒分析需要进行逆向工程。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器可以提供二进制文件的伪源代码视图），因此深入检查需要您阅读汇编代码。

6.9.2.1. 基本信息收集

在这一节中，我们将了解一些使用静态分析来收集特定应用程序的基本信息的方法和工具。

6.9.2.1.1. 应用程序二进制

你可以使用 class-dump 来获取应用程序源代码中方法的信息。下面的例子使用 Damn Vulnerable iOS 应用程序来演示。我们的二进制文件是一个所谓的胖二进制文件，这意味着它可以在 32 位和 64 位平台上执行。

解压应用程序并运行 otool:

```
unzip DamnVulnerableiOSApp.ipa
cd Payload/DamnVulnerableIOSApp.app
otool -hv DamnVulnerableIOSApp
```

输出如下:

```
DamnVulnerableIOSApp (architecture armv7):
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
  MH_MAGIC     ARM         V7  0x00     EXECUTE    33       3684   NOUNDEFS D
YLDLINK 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    33       4192   NOUNDEFS
DYLDLINK TWOLEVEL WEAK_DEFINES BINDS_TO_WEAK PIE
```

注意架构：armv7（32 位）和 arm64（64 位)。这种胖二进制的设计允许在所有架构上部署应用程序。

要用 class-dump 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:

```
lipo -thin armv7 DamnVulnerableIOSApp -output DVIA32
```

然后我们可以继续执行 class-dump:

```
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。

> 一些商业反汇编程序（如 Hopper）会自动执行这些步骤，你就可以看到反汇编后的二进制和类信息。

以下命令列出共享库：

```
otool -L <binary>
```

6.9.2.1.2. 获取字符串

在分析二进制文件时，字符串总是一个很好的起点，因为它们提供了相关代码的上下文联系。例如，像 "密文生成失败 "这样的错误日志字符串给了我们一个提示，即相邻的代码可能负责生成一个密文。

为了从 iOS 二进制文件中提取字符串，你可以使用 Ghidra 或 Cutter 等 GUI 工具，或者依靠基于 CLI 的工具，如 Unix 工具 strings（**`strings <path_to_binary>`**）或 radare2 的 rabin2（**`rabin2 -zz <path_to_binary>`**）。当使用基于 CLI 的工具时，你可以利用其他工具，如 grep（例如与正则表达式结合使用）来进一步过滤和分析结果。

6.9.2.1.3. 交叉引用

Ghidra 可用于分析 iOS 二进制文件，并通过右键单击所需的函数并选择 **Show References to** 来获得交叉引用。

6.9.2.1.4. API 使用

iOS 平台为应用程序中经常使用的功能提供了许多内置库，例如加密、蓝牙、NFC、网络和位置库。确定这些库在一个应用程序中的存在可以给我们提供关于其基础工作的宝贵信息。

例如，如果一个应用程序正在导入 CC_SHA256 函数，它表明该应用程序将使用 SHA256 算法执行某种散列操作。关于如何分析 iOS 的加密 API 的进一步信息将在 "iOS 加密 API "一节中讨论。

同样地，上述方法可以用来确定一个应用程序在哪里以及如何使用蓝牙。例如，一个使用蓝牙通道进行通信的应用程序必须使用核心蓝牙框架的函数，如 **`CBCentralManager`** 或 **`connect`**。使用 iOS 蓝牙文档，你可以确定关键函数，并围绕这些函数导入开始分析。

6.9.2.1.5. 网络通信

你可能遇到的大多数应用程序都连接到远程端点。即使在你进行任何动态分析（如流量捕捉和分析）之前，你也可以通过列举应用程序应该与之通信的域名来获得一些初始输入或入口点。

通常，这些域名会以字符串的形式出现在应用程序的二进制文件中。我们可以通过检索字符串（如上所述）或使用 Ghidra 等工具检查字符串来提取域名。后者有一个明显的优势：它可以为你提供上下文联系，因为你可以通过检查交叉引用看到每个域名在哪个上下文中被使用。

从这里开始，你可以使用这些信息来获得更多的了解，这些了解在以后的分析中可能会有用，例如，你可以将域名与证书固定相匹配，或者对域名进行进一步的侦察，以了解更多关于目标环境的信息。

安全连接的实施和验证可能是一个复杂的过程，有许多方面需要考虑。例如，许多应用程序使用除 HTTP 之外的其他协议，如 XMPP 或纯 TCP 数据包，或执行证书固定，以试图阻止 MITM 攻击。

请记住，在大多数情况下，只使用静态分析是不够的，甚至可能会变成非常低效的，相比之下，动态分析会得到更可靠的结果（例如，使用拦截代理）。在本节中，我们只是接触到了表面，所以请参考 "iOS 基本安全测试 "一章中的 "基本网络监控/嗅探 "一节，并查看 "iOS 网络通信 "一章中的测试案例以获得更多信息。

6.9.2.2. 人工（逆向）代码审查

6.9.2.2.1 审查反汇编的 Objective-C 和 Swift 代码

在本节中，我们将人工探索 iOS 应用程序的二进制代码并对其进行静态分析。人工分析可能是一个缓慢的过程，需要极大的耐心。一个好的人工分析可以使动态分析更加成功。

执行静态分析没有硬性的书面规则，但有一些经验法则可以用来进行系统的人工分析。

- 了解被评估的应用程序的工作情况-应用程序的目标以及在输入错误的情况下它是如何表现的。

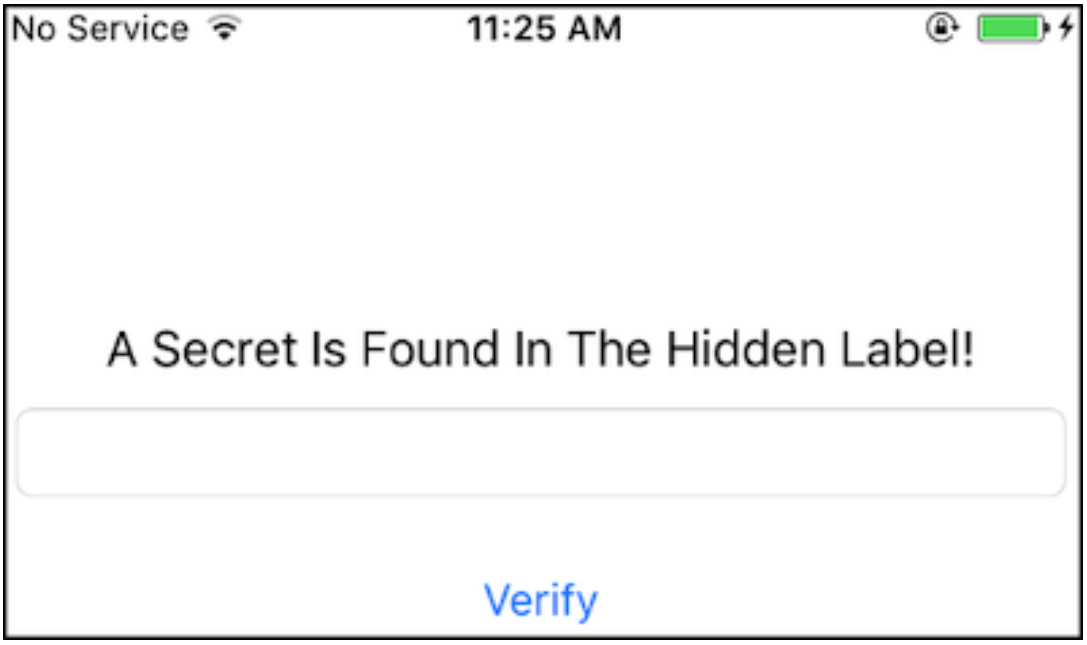
- 探索应用程序二进制文件中的各种字符串，这对发现应用程序中有趣的功能和可能的错误处理逻辑非常有帮助。
- 寻找名称与我们的目标相关的函数和类。
- 最后，找到进入应用程序的各种入口点，并从那里开始探索应用程序。

本节讨论的技术是通用的，无论用于分析的工具是什么，都适用。

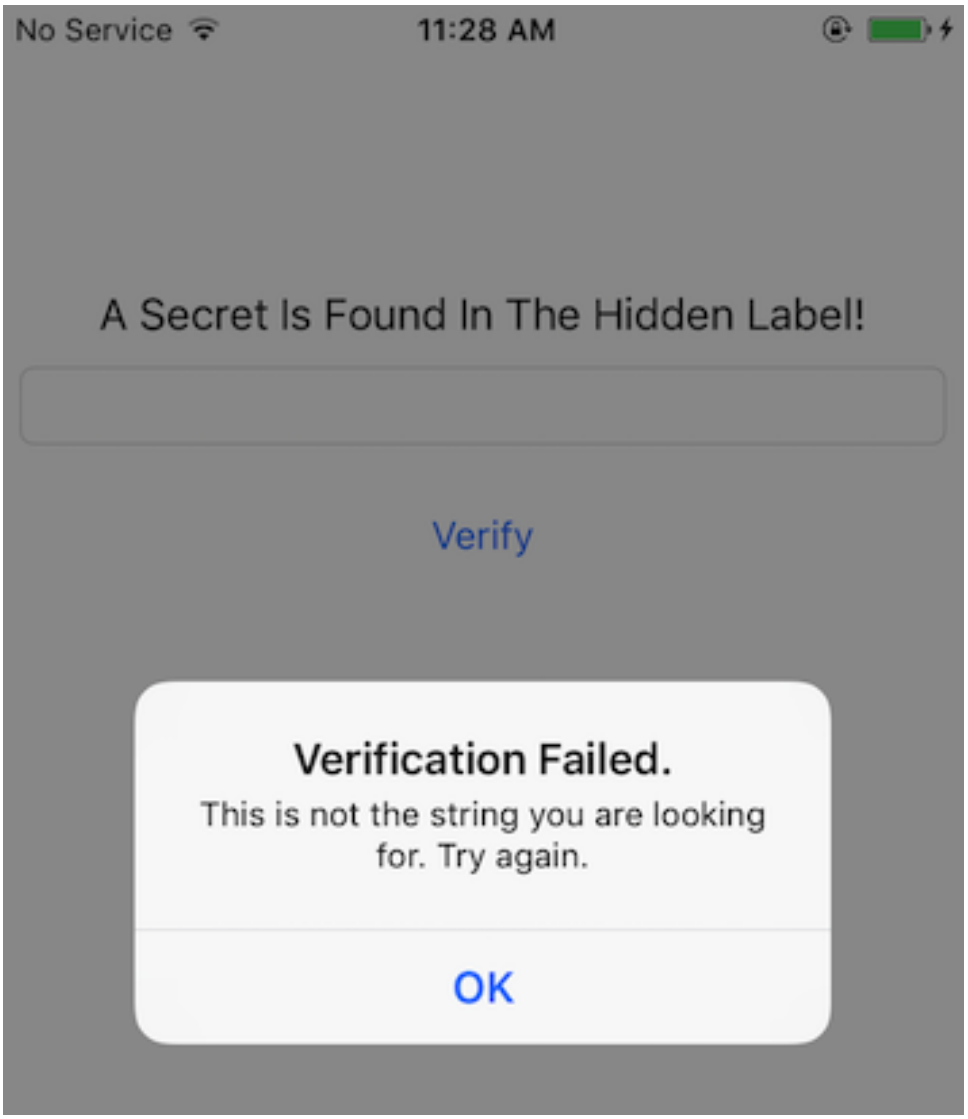
6.9.2.2.1.1. Objective-C

除了在 "反汇编和反编译 "一节中学到的技术外，对于这一节，你需要对 Objective-C 的运行时有一些了解。例如，像_objc_msgSend 或_objc_release 这样的函数对 Objective-C 运行时有特殊的意义。

我们将使用适用于 iOS 的 UnCrackable 应用程序级别 1，它的目标很简单，就是找到隐藏在二进制文件中的某个秘密字符串。该应用程序只有一个主屏幕，用户可以通过在提供的文本字段中输入自定义字符串进行交互。



当用户输入错误的字符串时，应用程序会弹出一个 "验证失败 "的消息。

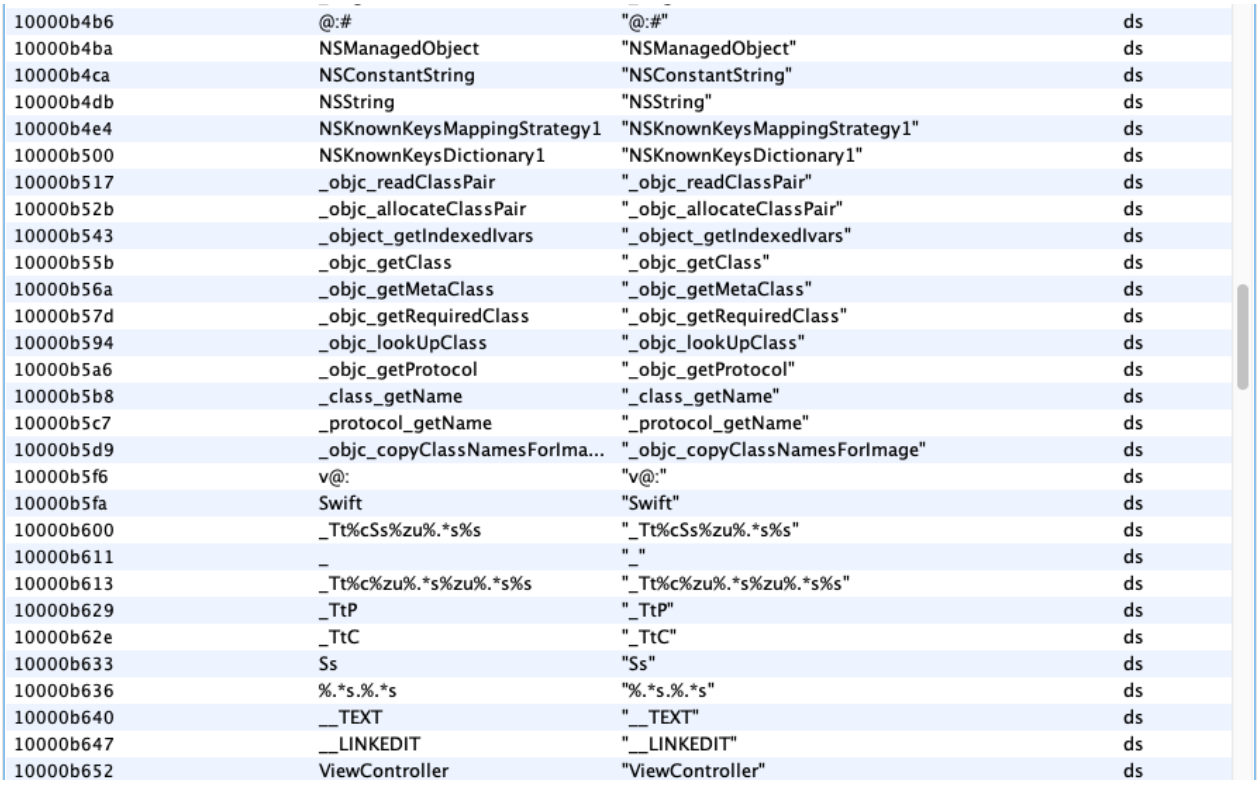


你可以记下弹出窗口中显示的字符串，因为这在搜索处理输入并作出决定的代码时可能会有帮助。幸运的是，这个应用程序的复杂性和交互性是简单明了的，这对我们的逆向工作是个好兆头。

> 对于本节的静态分析，我们将使用 Ghidra 9.0.4。Ghidra 9.1_beta 自动分析有一个错误，不能显示 Objective-C 类。

我们可以先通过在 Ghidra 中打开二进制文件来检查其中的字符串。列出的字符串一开始可能会让人不知所措，但随着逆向 Objective-C 代码的经验增加，你会学会如何过滤和丢弃那些没有实际帮助或相关性的字符串。例如，下面截图中的那些，是为 Objective-C 运行时生成的。其他的

字符串在某些情况下可能是有帮助的，比如那些包含符号的字符串（函数名、类名等），我们在执行静态分析以检查是否使用了某些特定的函数时将会用到它们。



10000b4b6	@:#	"@:#"	ds
10000b4ba	NSManagedObject	"NSManagedObject"	ds
10000b4ca	NSConstantString	"NSConstantString"	ds
10000b4db	NSString	"NSString"	ds
10000b4e4	NSKnownKeysMappingStrategy1	"NSKnownKeysMappingStrategy1"	ds
10000b500	NSKnownKeysDictionary1	"NSKnownKeysDictionary1"	ds
10000b517	_objc_readClassPair	"_objc_readClassPair"	ds
10000b52b	_objc_allocateClassPair	"_objc_allocateClassPair"	ds
10000b543	_object_getIndexedIvars	"_object_getIndexedIvars"	ds
10000b55b	_objc_getClass	"_objc_getClass"	ds
10000b56a	_objc_getMetaClass	"_objc_getMetaClass"	ds
10000b57d	_objc_getRequiredClass	"_objc_getRequiredClass"	ds
10000b594	_objc_lookUpClass	"_objc_lookUpClass"	ds
10000b5a6	_objc_getProtocol	"_objc_getProtocol"	ds
10000b5b8	_class_getName	"_class_getName"	ds
10000b5c7	_protocol_getName	"_protocol_getName"	ds
10000b5d9	_objc_copyClassNamesForIma...	"_objc_copyClassNamesForImage"	ds
10000b5f6	v@:	"v@:"	ds
10000b5fa	Swift	"Swift"	ds
10000b600	_Tt%cSs%zu%.*s%s	"_Tt%cSs%zu%.*s%s"	ds
10000b611	_	"_"	ds
10000b613	_Tt%c%zu%.*s%zu%.*s%s	"_Tt%c%zu%.*s%zu%.*s%s"	ds
10000b629	_TtP	"_TtP"	ds
10000b62e	_TtC	"_TtC"	ds
10000b633	Ss	"Ss"	ds
10000b636	%.*s.%.*s	"%.*s.%.*s"	ds
10000b640	__TEXT	"__TEXT"	ds
10000b647	__LINKEDIT	"__LINKEDIT"	ds
10000b652	ViewController	"ViewController"	ds

如果我们继续仔细分析，我们可以发现 "Verification Failed 验证失败 "这个字符串，当输入错误时，将弹出此内容。如果你随着这个字符串的交叉引用（Xrefs），你会发现 **`ViewController`** 类的 **`buttonClick`** 函数。我们将在本节后面研究 **`buttonClick`** 函数的问题。当进一步检查应用程序中的其他字符串时，只有少数字符串看起来可能是隐藏标志的候选者。你也可以试试它们，并进行验证。

Defined Strings - 457 items

Location	String Value	String Representation	Data T...
10000b187	debugDescription	"debugDescription"	ds
10000b198	_window	"_window"	ds
10000b1a0	load	"load"	ds
10000b1a5	setObject:forKeyedSubscript:	"setObject:forKeyedSubscript:"	ds
10000b1c2	setObject:forKey:	"setObject:forKey:"	ds
10000b1d4	removeObjectForKey:	"removeObjectForKey:"	ds
10000b1e8	objectForKeyedSubscript:	"objectForKeyedSubscript:"	ds
10000b201	init	"init"	ds
10000b206	allocWithEntity:	"allocWithEntity:"	ds
10000b217	allocBatch:withEntity:count:	"allocBatch:withEntity:count:"	ds
10000b234	fastIndexForKnownKey:	"fastIndexForKnownKey:"	ds
10000b24a	indexForKey:	"indexForKey:"	ds
10000b257	objectForKey:	"objectForKey:"	ds
10000b265	addEntriesFromDictionary:	"addEntriesFromDictionary:"	ds
10000b27f	initialize	"initialize"	ds
10000b28a	lengthOfBytesUsingEncoding:	"lengthOfBytesUsingEncoding:"	ds
10000b2a6	getCString:maxLength:enco...	"getCString:maxLength:encoding:"	ds
10000b2c5	initWithBytes:length:encoding:	"initWithBytes:length:encoding:"	ds
10000b2e4	keyEnumerator	"keyEnumerator"	ds
10000b2f2	nextObject	"nextObject"	ds
10000b2fd	Congratulations!	"Congratulations!"	ds
10000b30e	You found the secret!!	"You found the secret!!"	ds
10000b325	OK	"OK"	ds
10000b328	Verification Failed.	"Verification Failed."	ds
10000b33d	This is not the string you ar...	"This is not the string you are looking for. Try agai...	ds
10000b374	theLabel	"theLabel"	ds
10000b37d	T@"UILabel",W,N,V_theLabel	"T@\"UILabel\",W,N,V_theLabel"	ds
10000b398	Hint	"Hint"	ds
10000b39d	T@"UILabel",W,N,V_Hint	"T@\"UILabel\",W,N,V_Hint"	ds
10000b3b4	theTextField	"theTextField"	ds
10000b3c1	T@"UITextField",W,N,V_theT...	"T@\"UITextField\",W,N,V_theTextField"	ds
10000b3e4	bVerify	"bVerify"	ds
10000b3ec	T@"UIButton",W,N,V_bVerify	"T@\"UIButton\",W,N,V_bVerify"	ds
10000b407	hash	"hash"	ds

接下来，我们有两条路可走。我们可以开始分析上述步骤中确定的 **buttonClick** 函数，或者从各个入口点开始分析应用程序。在现实世界中，大多数时候你会采取第一条路径，但从学习的角度来看，在本节中我们将采取后一条路径。

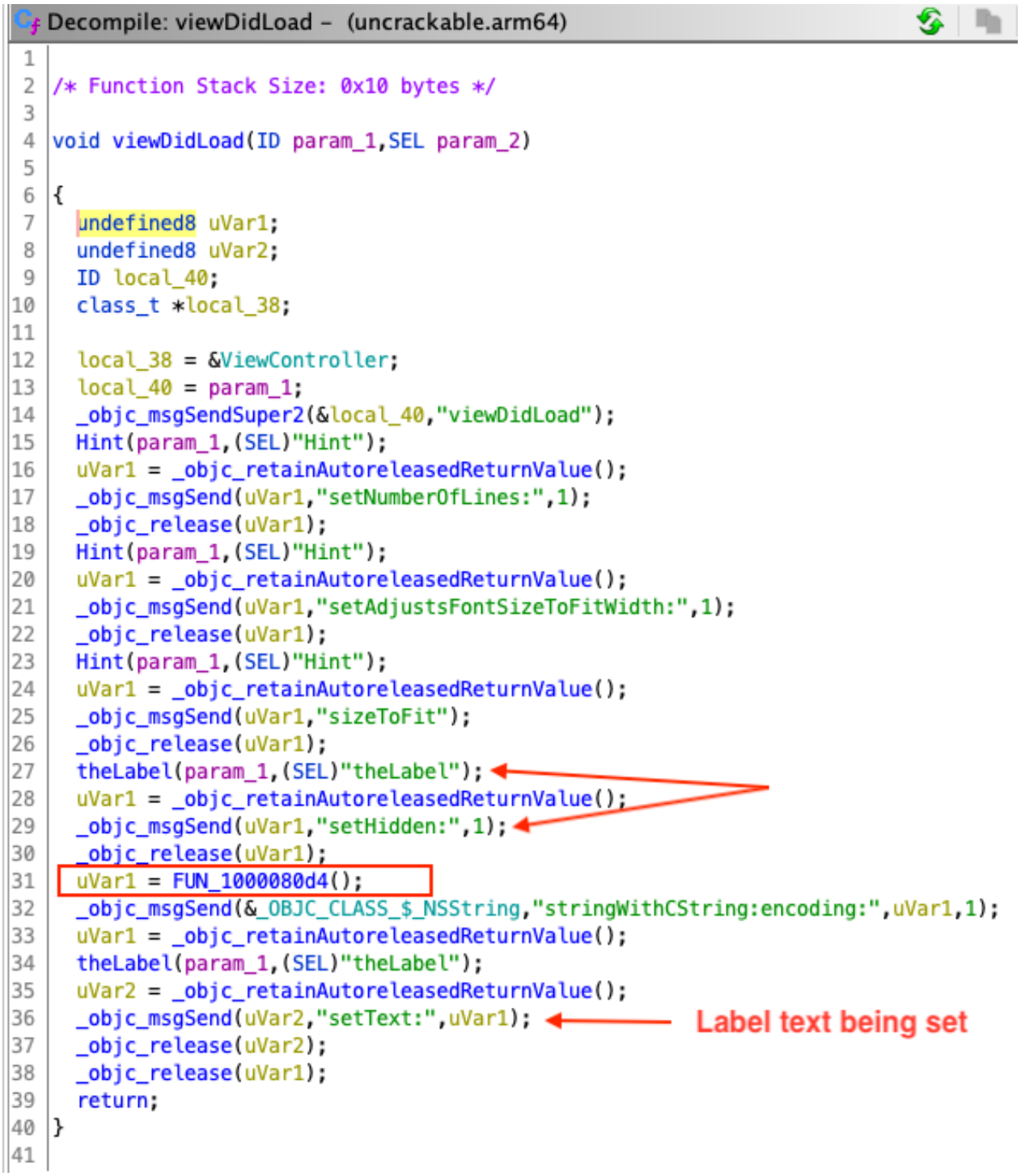
一个 iOS 应用程序根据其在应用程序生命周期内的状态，调用 iOS 运行时提供的不同预定义函数。这些函数被称为应用程序的入口点。例如：

- **[AppDelegate application:didFinishLaunchingWithOptions:]**在应用程序首次启动时被调用。
- **[AppDelegate applicationDidBecomeActive:]**在应用程序从非活动状态转为活动状态时被调用。

许多应用程序在这些部分执行关键代码，因此它们通常是一个好的起点，以便系统地跟踪代码。

一旦我们完成了对 **AppDelegate** 类中所有函数的分析，我们可以得出结论，没有相关的代码存在。上述函数中缺少代码，这就提出了一个问题--应用程序的初始化代码是从哪里被调用的?

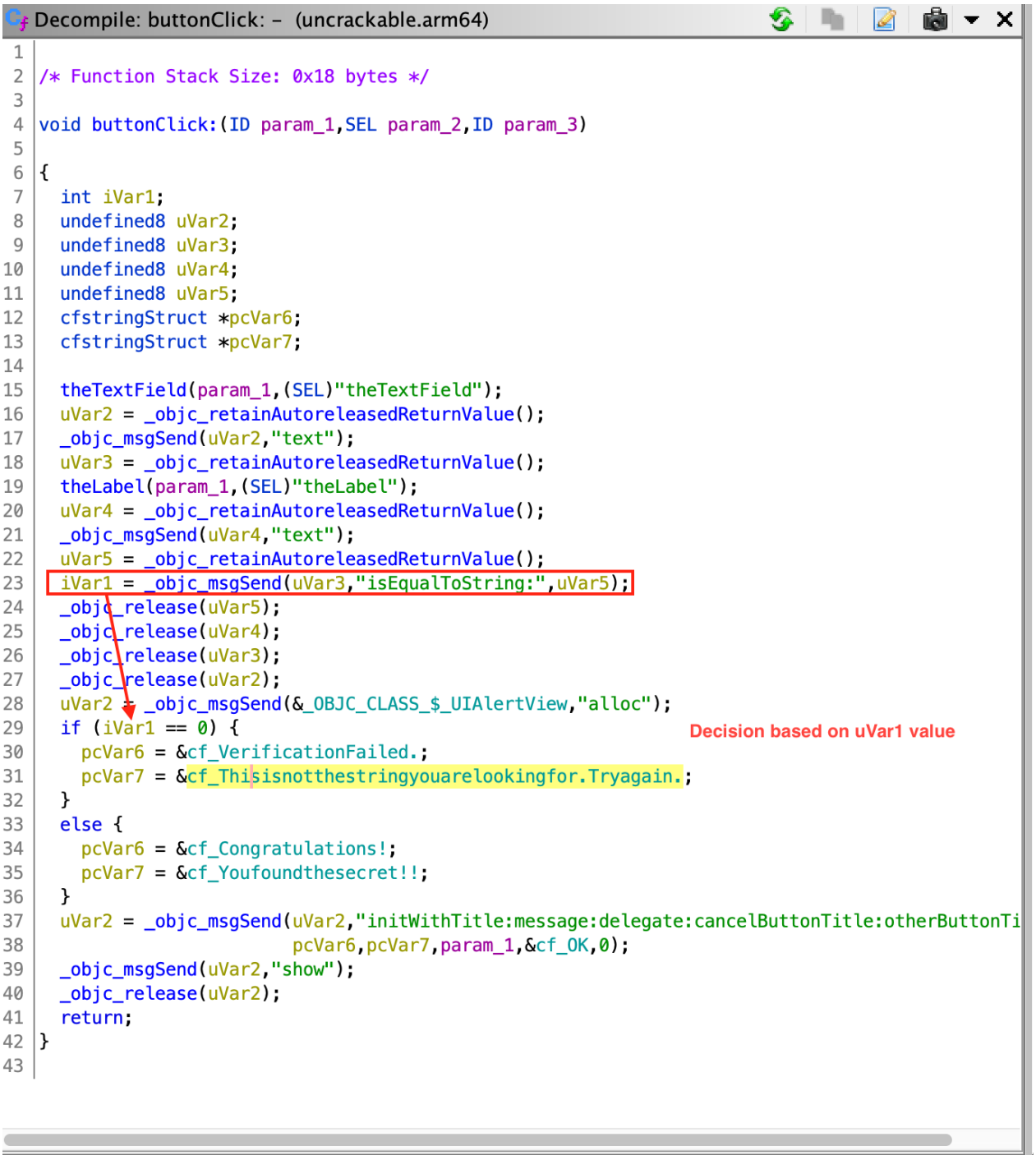
幸运的是，当前的应用程序代码量较小，我们可以在 **Symbol Tree**（符号树视图）中找到另一个 **ViewController** 类。在这个类中，函数 **viewDidLoad** 的功能看起来很有趣。如果你查看 **viewDidLoad** 的文档，你可以看到它也可以用来对视图执行额外的初始化。



如果我们检查这个函数的反编译，有一些有趣的事情正在发生。例如，在第 31 行有一个对本地函数的调用，在第 27-29 行，一个标签被初始化，其 **setHidden** 参数被设置为 1。你可以把这

些观察记录下来，然后继续探索这个类中的其他函数。为了简洁起见，探索该函数的其他部分被留作读者的练习。

在我们的第一步中，我们观察到，只有当 UI 按钮被按下时，应用程序才会验证输入字符串。因此，分析 **buttonClick** 函数是一个明显的目标。如前所述，这个函数也包含了我们在弹出窗口中看到的字符串。在第 29 行，正在做出一个决定，这个决定是基于 **isEqualString** 的结果（第 23 行 **uVar1** 中保存的输出）。比较的输入来自文本输入字段（来自用户）和标签的值。因此，我们可以认为，隐藏的标志被保存在该标签中。



现在我们已经跟踪了完整的流程，并且掌握了关于应用流程的所有信息。我们还得出结论，隐藏的标志存在于一个文本标签中，为了确定标签的值，我们需要重新审视 **viewDidLoad** 函数，并理解在确定的原生函数中发生了什么。对原生函数的分析将在 "审查反汇编的原生代码 "中讨论。

6.9.2.2.1.2. 审查反汇编的原生代码

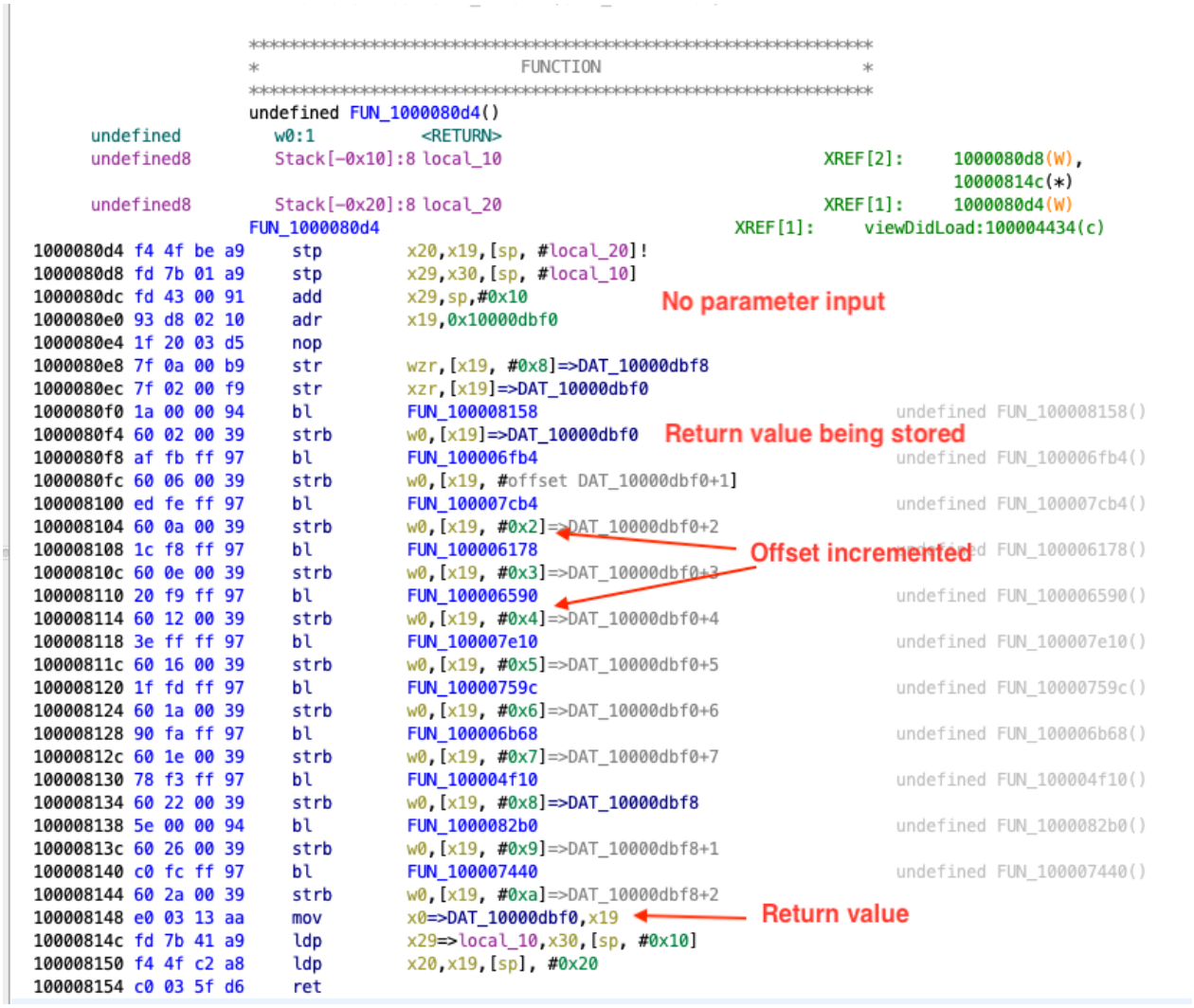
分析反汇编的原生代码需要很好地理解底层平台使用的调用惯例和指令。在本节中，我们将研究 ARM64 反汇编的原生代码。学习 ARM 架构的一个很好的起点是 Azeria Labs 教程中的 ARM 汇编基础知识介绍。这是我们在本节中要使用的东西的一个快速总结。

- 在 ARM64 中，寄存器的大小为 64 位，被称为 Xn，其中 n 是 0 到 31 的数字。如果使用寄存器的低位（LSB）32 位，那么它被称为 Wn。
- 一个函数的输入参数在 X0-X7 寄存器中传递。
- 函数的返回值通过 X0 寄存器传递。
- 加载（LDR）和存储（STR）指令用于从/向寄存器读取或写入内存。
- B, BL, BLX 是用于调用函数的分支指令。

如上所述，Objective-C 代码也被编译为原生二进制代码，但分析 C/C++原生代码可能更具挑战性。对于 Objective-C，有各种符号（特别是函数名）存在，这使得对代码的理解更加容易。在上一节中，我们已经了解到，像 **setText**、**isEqualStrings** 这样的函数名称的存在可以帮助我们快速理解代码的语义。如果是 C/C++的原生代码，如果所有的二进制文件都被剥离了，可能会有很少或者没有符号存在来帮助我们分析它。

反编译器可以帮助我们分析原生代码，但应该谨慎使用。现代反编译器非常复杂，在它们用来反编译代码的许多技术中，有一些是基于启发式的。基于启发式的技术不一定能给出正确的结果，其中一个例子就是确定一个给定的原生函数的输入参数的数量。掌握分析反汇编代码的知识，在反编译器的辅助下，可以使分析原生代码不那么容易出错。

我们将分析上一节中 **viewDidLoad** 函数中确定的原生函数。该函数位于偏移量 0x1000080d4 处。这个函数的返回值用于标签的 **setText** 函数调用。这个文本被用来与用户输入的内容进行比较。因此，我们可以肯定，这个函数将返回一个字符串或等价物。



在函数的反汇编中，我们可以看到的第一件事是没有对函数的输入。寄存器 X0-X7 在整个函数中没有被读取。此外，还有多个调用其他函数的情况，比如在 0x100008158，0x10000dbf0 等。

下面可以看到一个这样的函数调用所对应的指令。分支指令 bl 被用来调用 0x100008158 处的函数。

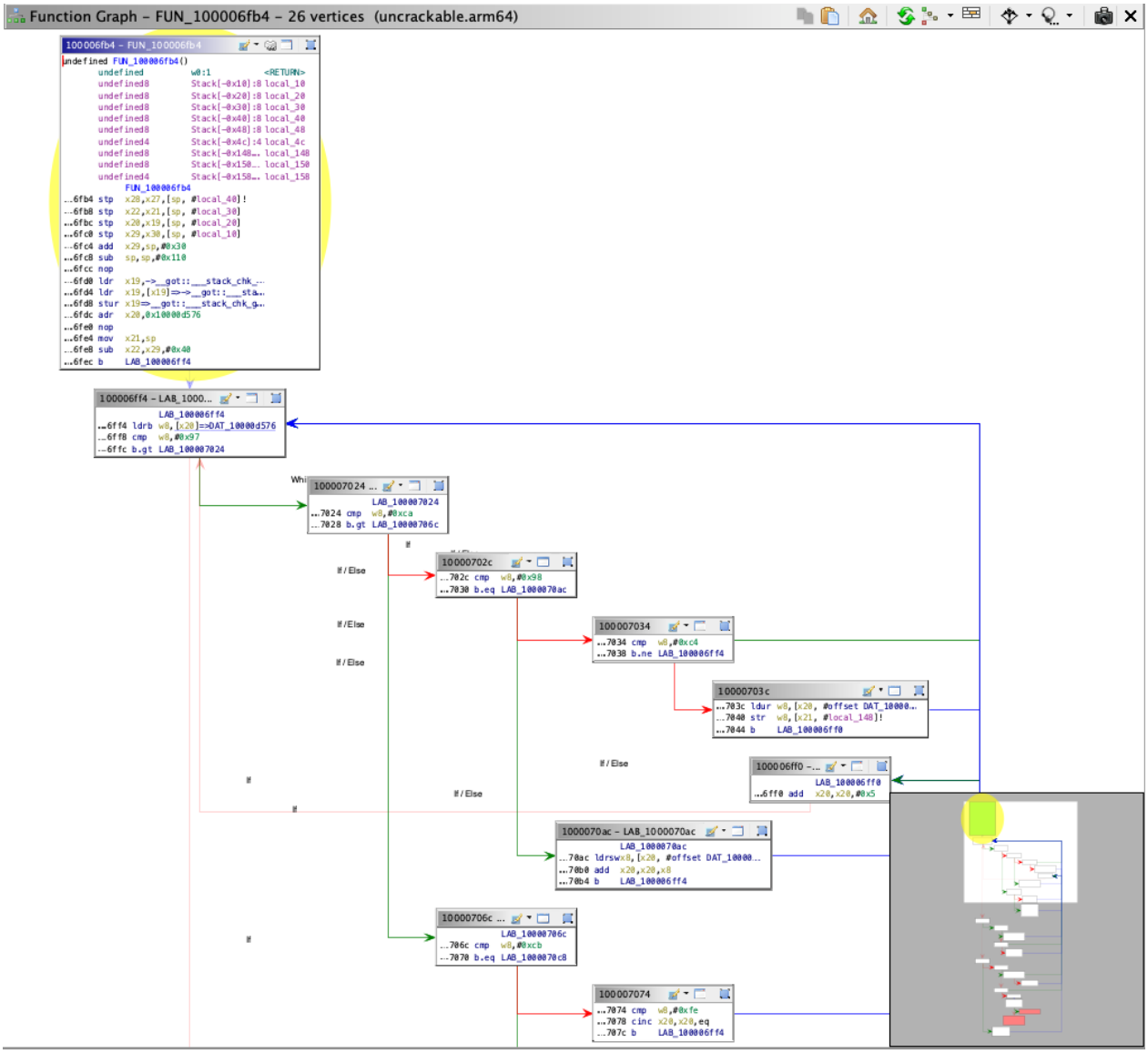
```
1000080f0 1a 00 00 94      bl          FUN_100008158
1000080f4 60 02 00 39      strb        w0,[x19]=>DAT_10000dbf0
```

函数的返回值（在 W0 中找到），被存储到寄存器 X19 中的地址（strb 存储一个字节到寄存器中的地址）。我们可以看到其他函数调用的模式是一样的，返回值被存储在 X19 寄存器中，而且每次的偏移量都比前一个函数调用多一个。这种行为可以和每次填充字符串数组的每个索引联系起

来。每个返回值都被写入这个字符串数组的一个索引中。有 11 个这样的调用，从目前的证据来看，我们可以做出一个明智的猜测，隐藏标志的长度是 11。在反汇编接近尾声的时候，该函数返回的是这个字符串数组的地址。

```
100008148 e0 03 13 aa      mov         x0=>DAT_10000dbf0,x19
```

为了确定隐藏标志的值，我们需要知道上面确定的每个后续函数调用的返回值。在分析函数 0x100006fb4 时，我们可以观察到这个函数比我们分析的前一个函数大得多、复杂得多。在分析复杂的函数时，函数图可能非常有帮助，因为它有助于更好地理解函数的控制流。函数图可以在 Ghidra 中通过点击子菜单中的 **Display function graph**（显示函数图）图标来获得。



完全人工分析所有的原生函数将耗费时间，而且可能不是最明智的做法。在这种情况下，强烈建议使用动态分析方法。例如，通过使用劫持等技术或简单地调试应用程序，我们可以很容易地确定返回值。通常情况下，使用动态分析方法是一个好主意，然后再进行人工分析函数的反馈循环。这样你可以同时从两种方法中获益，同时节省时间和减少工作量。动态分析技术将在 "动态分析 "部分讨论。

6.9.2.3. 自动化静态分析

有几个用于分析 iOS 应用的自动化工具；其中大多数是商业工具。免费的开源工具 MobSF 和 objection 有一些静态和动态分析功能。其他工具列在 附录"测试工具 "中的"静态源代码分析"部分。

不要回避使用自动扫描器进行分析--它们可以帮助你摘取低垂的果实，让你专注于分析中更有趣的方面，如业务逻辑。请记住，静态分析器可能会产生误报；一定要仔细审查其结果。

6.9.3. 动态分析

有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 动态监视工具。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. 未越狱设备动态分析

如果你没有机会接触到越狱设备，你可以修补目标应用并重新打包，以便在启动时加载一个动态库（例如 Frida 小工具，以便用 Frida 和相关工具（如 objection）进行动态测试）。这样一来，你就可以对应用进行检测，并做一切你需要做的动态分析（当然，你不能通过这种方式突破沙盒）。然而，这种技术只有在应用程序的二进制文件没有被 FairPlay 加密（即从 App Store 获得）时才有效。

6.9.3.1.1.自动化重新打包

Objection 将应用重新包装的过程自动化。你可以在官方 wiki 页面上找到详尽的文档。

使用 objection 的重新打包功能对于大多数的使用情况来说是非常有效的。然而，在一些复杂的情况下，你可能需要更精细的控制或更可定制的重新打包过程。在这种情况下，你可以阅读 "手动重新打包 "中关于重新打包和重签名过程的详细解释。

6.9.3.1.2. 手动重新打包

由于 Apple 令人困惑的供应和代码签名系统，重新签署一个应用程序比你想象的更具有挑战性。除非你得到供应配置文件和代码签名头完全正确，否则 iOS 不会运行一个应用程序。这需要学习许多概念--证书类型、包 ID、应用程序 ID、团队标识符，以及 Apple 的构建工具如何连接它们。让操作系统运行一个没有通过默认方法（Xcode）构建的二进制文件可能是一个令人生畏的过程。

我们将使用 optool，Apple 的构建工具，以及一些 shell 命令。我们的方法是受 Vincent Tan 的 Swizzler 项目启发。NCC 小组描述了一种替代的重新包装方法。

要重现下面列出的步骤，请从 OWASP 移动测试指南仓库下载适用于 iOS 的 UnCrackable 应用级别 1。我们的目标是让 UnCrackable 应用在启动时加载 FridaGadget.dylib，这样我们就可以用 Frida 来检测应用。

> 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

配置文件是一个由苹果公司签署的 plist 文件，它将你的代码签名证书添加到其在一个或多个设备上接受的证书列表中。换句话说，这代表苹果公司明确允许你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你的应用程序的权利。证书包含你用来签名的私钥。

根据你是否注册为 iOS 开发者，你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 ***security*** 工具列出您的签名标识：

```
$ security find-identity -v
 1) 61FA3547E0AF42A11E233F6A2B255E6B6AF262CE "iPhone Distribution: Company Name. Ltd."
```

```
 2) 8004380F331DCA22CC1B47FB1A805890AE41C938 "iPhone Developer: Bernhard Müll
er (RV852WND79)"
```

登录 Apple Developer 门户发布新的应用程序 ID，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 提供的团队 ID 和一个包 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**-在下面的 shell 命令中，用你自己的文件名代替它。。

有一个普通 Apple ID：

即使你不是一个付费的开发者，苹果也会发放一个免费的开发配置文件。你可以通过 Xcode 和你的普通苹果账户获得该配置文件：只需创建一个空的 iOS 项目，并从应用容器中提取 **embedded.mobileprovision**，它位于你的主目录的 Xcode 子目录中。**~/Library/Developer/Xcode/DerivedData/<ProjectName>/Build/Products/Debug-iphoneos/<ProjectName>.app/**。NCC 的博文 "无需越狱的 iOS 插桩 "详细解释了这个过程。

一旦你获得了配置文件，你可以用 ***security*** 工具检查其内容。你会发现配置文件中授予应用程序的权利，以及允许的证书和设备。你需要这些来进行代码签名，所以把它们提取到一个单独的 plist 文件，如下图所示。看一下文件内容，确保一切都符合预期。

```
$ security cms -D -i AwesomeRepackaging.mobileprovision > profile.plist
$ /usr/libexec/PlistBuddy -x -c 'Print :Entitlements' profile.plist > entitle
ments.plist
$ cat entitlements.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DT
Ds/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)和包 ID(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**)。

6.9.3.2. 基本信息收集

在 iOS 上，收集一个正在运行的进程或一个应用程序的基本信息可能比 Android 稍有挑战性。在 Android（或任何基于 Linux 的操作系统）上，进程信息通过 procfs 以可读文本文件的形式暴露。因此，关于目标进程的任何信息都可以通过解析这些文本文件在有 root 权限的设备上获得。相比之下，在 iOS 上，没有 procfs 的对应物存在。此外，在 iOS 上，许多用于探索进程信息的标准 UNIX 命令行工具，例如 lsof 和 vmmap，被移除以减少固件的大小。

在本节中，我们将学习如何使用 lsof 等命令行工具收集 iOS 上的进程信息。由于许多这些工具在 iOS 上默认不存在，我们需要通过其他方法来安装它们。例如，lsof 可以用 Cydia 安装（可执行文件不是最新的版本，但仍能满足我们的目的)。

6.9.3.2.1. 打开文件

lsof 是一个强大的命令，它提供了关于一个运行中的进程的大量信息。它可以提供所有打开的文件的列表，包括一个流、一个网络文件或一个普通文件。当不带有任何参数调用 **lsof** 命令时，它将列出属于系统中所有活动进程的所有打开的文件，而当调用参数-c <进程名称>或-p <pid>时，它返回指定进程的打开文件的列表。man 页面显示了其他各种参数的细节。

对 PID 为 2828 的运行的 iOS 程序使用 lsof，列出各种打开的文件，如下图所示。

```
iPhone:~ root# lsof -p 2828
COMMAND  PID   USER   FD   TYPE DEVICE SIZE/OFF   NODE NAME
iOweApp 2828 mobile  cwd    DIR    1,2      864      2 /
iOweApp 2828 mobile  txt    REG    1,3   206144 189774 /private/var/containers/Bundle/Application/F390A491-3524-40EA-B3F8-6C1FA105A23A/iOweApp.app/iOweApp
iOweApp 2828 mobile  txt    REG    1,3     5492 213230 /private/var/mobile/Containers/Data/Application/5AB3E437-9E2D-4F04-BD2B-972F6055699E/tmp/com.apple.dyld/iOweApp-6346DC276FE6865055F1194368EC73CC72E4C5224537F7F23DF19314CF6FD8A
```

```
A.closure
iOweApp 2828 mobile  txt    REG    1,3    30628 212198 /private/var/preferenc
es/Logging/.plist-cache.vqXhr1EE
iOweApp 2828 mobile  txt    REG    1,2    50080 234433 /usr/lib/libobjc-tramp
olines.dylib
iOweApp 2828 mobile  txt    REG    1,2   344204  74185 /System/Library/Fonts/
AppFonts/ChalkboardSE.ttc
iOweApp 2828 mobile  txt    REG    1,2   664848 234595 /usr/lib/dyld
...
```

6.9.3.2.2. 加载原生库

你可以在 objection 中使用 **`list_frameworks`** 命令来列出所有代表 Frameworks 的应用程序包。

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios bundles lis
t_frameworks
Executable      Bundle                                      Version    Path
--------------  ------------------------------------------  ---------  -------
-----------------------------------
Bolts           org.cocoapods.Bolts                         1.9.0      ...8/DV
IA-v2.app/Frameworks/Bolts.framework
RealmSwift      org.cocoapods.RealmSwift                    4.1.1      ...A-v
2.app/Frameworks/RealmSwift.framework
                                                                       ...yste
m/Library/Frameworks/IOKit.framework
...

#### Open Connections

`lsof` command when invoked with option `-i`, it gives the list of open netwo
rk ports for all active processes on the device. To get a list of open networ
k ports for a specific process, the `lsof -i -a -p <pid>` command can be use
d, where `-a` (AND) option is used for filtering. Below a filtered output for
 PID 1 is shown.

```bash
iPhone:~ root# lsof -i -a -p 1
COMMAND PID USER   FD   TYPE             DEVICE SIZE/OFF NODE NAME
launchd   1 root   27u  IPv6 0x69c2ce210efdc023      0t0  TCP *:ssh (LISTEN)
launchd   1 root   28u  IPv6 0x69c2ce210efdc023      0t0  TCP *:ssh (LISTEN)
launchd   1 root   29u  IPv4 0x69c2ce210eeaef53      0t0  TCP *:ssh (LISTEN)
launchd   1 root   30u  IPv4 0x69c2ce210eeaef53      0t0  TCP *:ssh (LISTEN)
launchd   1 root   31u  IPv4 0x69c2ce211253b90b      0t0  TCP 192.168.1.12:ss
h->192.168.1.8:62684 (ESTABLISHED)
launchd   1 root   42u  IPv4 0x69c2ce211253b90b      0t0  TCP 192.168.1.12:ss
h->192.168.1.8:62684 (ESTABLISHED)
```
```

6.9.3.2.3. 检查沙盒

在 iOS 上，每个应用程序都有一个沙盒文件夹来存储其数据。根据 iOS 的安全模型，一个应用程序的沙盒文件夹不能被其他应用程序访问。此外，用户不能直接访问 iOS 文件系统，从而防止浏览或提取文件系统中的数据。在 iOS<8.3 中，有一些应用程序可以用来浏览设备的文件系统，如 iExplorer 和 iFunBox，但在最近的 iOS 版本（>8.3）中，沙盒规则更加严格，这些应用程序不再可用。因此，如果你需要访问文件系统，只能在已越狱的设备上访问。作为越狱过程的一部分，应用程序的沙盒保护被禁用，从而使人们能够轻松访问沙盒中的文件夹。

应用程序沙盒文件夹的内容已经在 iOS 基本安全测试一章的 "访问应用程序数据目录 "中讨论过了。本章概述了文件夹结构以及你应该分析哪些目录。

6.9.3.3. 调试

来自 Linux 背景的你会期望 **ptrace** 系统调用像你习惯的那样强大，但由于某些原因，苹果决定让它不完整。iOS 调试器如 LLDB 使用它来附加、步进或继续进程，但它们不能使用它来读或写内存（所有 PT_READ_*和 PT_WRITE*请求都不见了）。相反，它们必须获得一个所谓的 Mach 任务端口（通过调用目标进程 ID 的 **task_for_pid**），然后使用 Mach IPC 接口 API 函数来执行诸如暂停目标进程和读/写寄存器状态（**thread_get_state/thread_set_state**）和虚拟内存（**mach_vm_read/mach_vm_write**）的操作。

> 欲了解更多信息，你可以参考 GitHub 中的 LLVM 项目，其中包含 LLDB 的源代码，以及《Mac OS X 和 iOS 内部：进入 Apple 的核心》[#levin]中的第 5 章和第 13 章和 "Mac 黑客手册"[#miller]的第 4 章 "跟踪和调试"。

6.9.3.3.1. 使用 LLDB 调试

Xcode 安装的默认 debugserver 可执行文件不能用于附加到任意进程（它通常只用于调试用 Xcode 部署的自主开发的应用程序）。为了实现对第三方应用程序的调试，debugserver 可执行文件必须添加 **task_for_pid-allow** 权限，以便调试器进程可以调用 **task_for_pid** 来获得前面所看到的目标 Mach 任务端口。一个简单的方法是将该权限授予 Xcode 提供的 debugserver 二进制文件。

若要获得可执行文件，请安装以下 DMG 镜像：

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Device Support/<target-iOS-version>/DeveloperDiskImage.dmg
```

你会在挂载的卷上的/usr/bin/目录下找到 debugserver 的可执行文件。把它复制到一个临时目录，然后创建一个名为 entitlements.plist 的文件，内容如下。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DT
Ds/ PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.springboard.debugapplications</key>
    <true/>
    <key>run-unsigned-code</key>
    <true/>
    <key>get-task-allow</key>
    <true/>
    <key>task_for_pid-allow</key>
    <true/>
</dict>
</plist>
```

使用 codesign 应用权限：

```
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。

```
iproxy 2222 22
scp -P 2222 debugserver root@localhost:/tmp/
```

注意：在 iOS 12 及更高版本中，使用以下程序来签署从 XCode 镜像中获得的 debugserver 二进制文件。

1) 通过 scp 将 debugserver 二进制文件复制到设备上，例如，在/tmp 文件夹中。

2) 通过 SSH 连接到设备，并创建一个名为 entitlements.xml 的文件，内容如下。

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN http://www.apple.com/D
TDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>platform-application</key>
    <true/>
    <key>com.apple.private.security.no-container</key>
    <true/>
    <key>com.apple.private.skip-library-validation</key>
    <true/>
    <key>com.apple.backboardd.debugapplications</key>
    <true/>
```

```
        <key>com.apple.backboardd.launchapplications</key>
        <true/>
        <key>com.apple.diagnosticd.diagnostic</key>
        <true/>
        <key>com.apple.frontboard.debugapplications</key>
        <true/>
        <key>com.apple.frontboard.launchapplications</key>
        <true/>
        <key>com.apple.security.network.client</key>
        <true/>
        <key>com.apple.security.network.server</key>
        <true/>
        <key>com.apple.springboard.debugapplications</key>
        <true/>
        <key>com.apple.system-task-ports</key>
        <true/>
        <key>get-task-allow</key>
        <true/>
        <key>run-unsigned-code</key>
        <true/>
        <key>task_for_pid-allow</key>
        <true/>
    </dict>
    </plist>
```

3) 键入以下命令来签署 debugserver 二进制文件。

```
ldid -Sentitlements.xml debugserver
```

4) 通过以下命令验证 debugserver 二进制文件可以被执行。

```
./debugserver
```

现在你可以把 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...
```

通过以下命令，你可以通过运行在目标设备上的 debugserver 启动一个应用程序。

```
debugserver -x backboard *:1234 /Applications/MobileSMS.app/MobileSMS
```

附加到一个已经运行的应用程序。

```
debugserver *:1234 -a "MobileSMS"
```

你现在可以从你的电脑主机连接到 iOS 设备。

```
(lldb) process connect connect://<ip-of-ios-device>:1234
```

输入 **image list** 可以得到一个主可执行程序和所有依赖库的列表。

6.9.3.3.2. 调试发布应用

在上一节中，我们了解了如何使用 LLDB 在 iOS 设备上设置调试环境。在这一节中，我们将使用这些信息，学习如何调试一个第三方发布的应用程序。我们将继续使用适用于 iOS 的 UnCrackable 应用级别 1 程序，并使用调试器解决它。

与调试版本不同的是，为发布构建而编译的代码是经过优化的，以达到最大的性能和最小的二进制构建大小。作为通用的最佳实践，大多数调试符号在发布版本中被剥离，在逆向工程和调试二进制文件时增加了一层复杂性。

由于调试符号的缺失，符号名称在回溯输出中缺失，并且不可能通过简单地使用函数名称来设置断点。幸运的是，调试器也支持直接在内存地址上设置断点。在本节中，我们将进一步学习如何这样做，并最终解决 crackme 挑战。

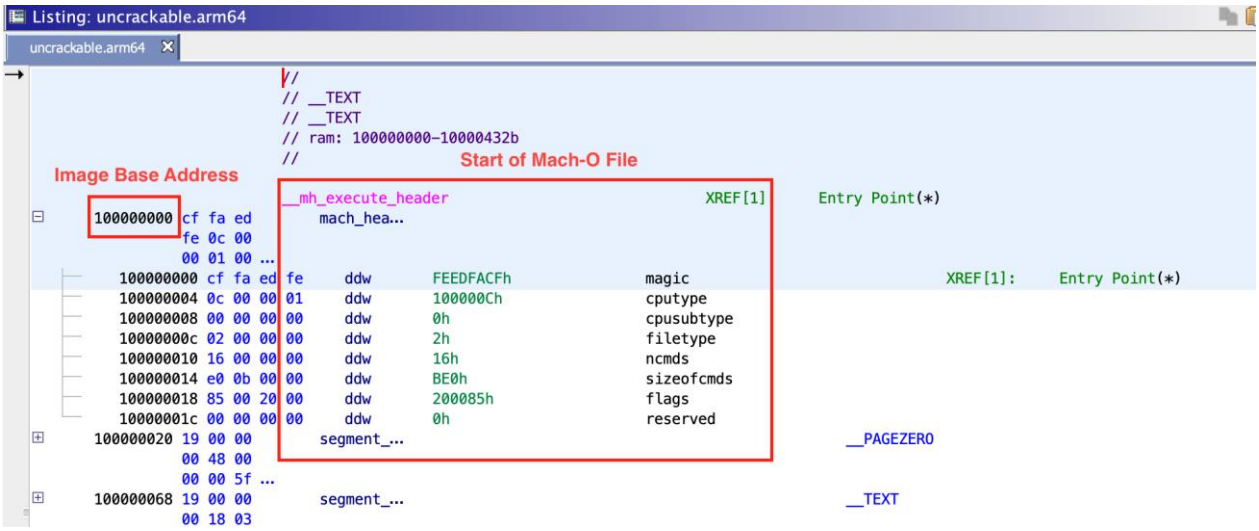
在使用内存地址设置断点之前，需要做一些基础工作。它需要确定两个偏移量。

1. 断点偏移：我们想要设置断点的代码的地址偏移。这个地址是通过在像 Ghidra 这样的反汇编程序中对代码进行静态分析得到的。
2. ASLR 移位偏移。当前进程的 ASLR 偏移量。由于 ASLR 偏移量是在应用程序的每个新实例上随机产生的，因此必须为每个调试会话单独获得。这是用调试器本身确定的。

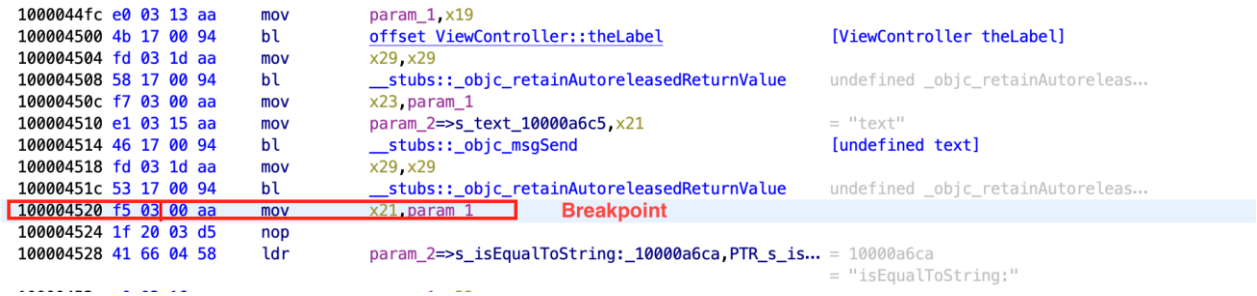
iOS 是一个现代化的操作系统，实施了多种技术来缓解代码执行攻击，其中一种技术是地址空间随机化布局（ASLR）。在一个应用程序的每一次新的执行中，都会产生一个随机的 ASLR 移位偏移量，各种进程的数据结构都会被这个偏移量所影响。

调试器中使用的最终断点地址是上述两个地址的总和（断点偏移量+ASLR 偏移量）。这种方法假定反汇编程序和 iOS 使用的镜像基址（稍后讨论）是相同的，这在大多数情况下是正确的。

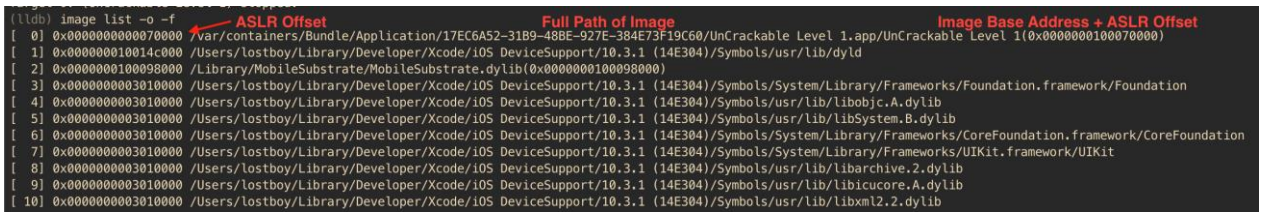
当一个二进制文件在像 Ghidra 这样的反汇编程序中被打开时，它通过模拟各自操作系统的加载器来加载二进制文件。二进制文件被加载的地址被称为镜像基址。这个二进制文件中的所有代码和符号都可以使用这个镜像基址的一个常数地址偏移来寻址。在 Ghidra 中，镜像基址可以通过确定 Mach-O 文件的起始地址获得。在这个例子中，它是 0x100000000。



从我们之前在 "人工（逆向）代码审查 "部分对 UnCrackable Level 1 应用程序的分析来看，隐藏字符串的值被存储在一个设置了隐藏标志的标签中。在反汇编中，这个标签的文本值存储在寄存器 X21 中，由 mov 从 X0 存储，在偏移量 0x100004520。这就是我们的断点偏移。



对于第二个地址，我们需要确定特定进程的 ASLR 偏移量。ASLR 偏移可以通过使用 LLDB 命令 **`image list -o -f`** 来确定。输出结果显示在下面的屏幕截图中。



在输出中，第一列包含镜像的序列号（[X]），第二列包含随机生成的 ASLR 偏移量，而第三列包含镜像的完整路径，在最后，括号内的内容显示了将 ASLR 偏移量添加到原始镜像基址后的镜像基址（0x100000000 + 0x70000 = 0x100070000）。你会发现 0x100000000 的镜像基址与 Ghidra 中的相同。现在，为了获得一个代码位置的有效内存地址，我们只需要在 Ghidra 中

确定的地址上加上 ASLR 偏移。设置断点的有效地址将是 0x100004520 + 0x70000 = 0x100074520。可以使用命令 b 0x100074520 来设置断点。

> 在上面的输出中，你可能还注意到，许多列为镜像的路径并不指向 iOS 设备上的文件系统。相反，它们指向 LLDB 正在运行的主机上的某个位置。这些镜像是系统库，其调试符号在主机上可用，以帮助应用程序的开发和调试（作为 Xcode iOS SDK 的一部分）。因此，您可以通过使用函数名直接对这些库设置断点。

设置断点并运行应用程序后，一旦到达断点，执行就会停止。现在你可以访问和探索进程的当前状态。在这种情况下，你从之前的静态分析中知道，寄存器 X0 包含了隐藏的字符串，因此让我们来探索它。在 LLDB 中，你可以使用 **po**（打印对象）命令打印 Objective-C 对象。

```
(lldb) b 0x100094520
Breakpoint 1: where = UnCrackable Level 1`___lldb_unnamed_symbol2$$UnCrackable Level 1 + 120, address = 0x0000000100094520
(lldb) c
Process 1165 resuming
Process 1165 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x0000000100094520 UnCrackable Level 1` ___lldb_unnamed_symbol2$$UnCrackable Level 1  + 120
UnCrackable Level 1`___lldb_unnamed_symbol2$$UnCrackable Level 1:
->  0x100094520 <+120>: mov    x21, x0
    0x100094524 <+124>: nop
    0x100094528 <+128>: ldr    x1, #0x8cc8               ; "isEqualToString:"
    0x10009452c <+132>: mov    x0, x22
    0x100094530 <+136>: mov    x2, x21
    0x100094534 <+140>: bl     0x10009a22c               ; symbol stub for: objc_msgSend
    0x100094538 <+144>: mov    x24, x0
    0x10009453c <+148>: mov    x0, x21
Target 0: (UnCrackable Level 1) stopped.
(lldb) po $x0
i am groot!

(lldb)
```

瞧，在静态分析和调试器的帮助下，crackme 可以轻松解决。LLDB 实现了大量的功能，包括改变寄存器的值，改变进程内存的值，甚至使用 Python 脚本自动完成任务。

Apple 官方推荐使用 LLDB 进行调试，但 GDB 也可以在 iOS 上使用。上面讨论的技术在使用 GDB 调试时也是适用的，只要将 LLDB 的特定命令改为 GDB 命令即可。

6.9.3.4. 跟踪

跟踪涉及记录程序执行的信息。与 Android 相比，可用来跟踪 iOS 应用程序的各个方面的选项有限。在本节中，我们将主要依靠 Frida 等工具来进行跟踪。

6.9.3.4.1. 方法跟踪

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，我们将假设您在一个越狱设备上测试。如果不是这样，您首先需要遵循重新打包和重新签名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** 参数接受 Objective-C 方法。你也可以把通配符传给它--例如，给定-[NSURL *]，**frida-trace** 将自动劫持所有 NSURL 类选择器。我们将用它来粗略了解 Safari 在用户打开一个 URL 时调用了哪些库函数。

在设备上运行 Safari，并确保设备通过 USB 连接。然后如下启动 **frida-trace**：

```
$ 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"
(...)
Started tracing 248 functions. Press Ctrl+C to stop.
```

接下来，在 Safari 浏览器中导航到一个新的网站。你应该在 **frida-trace** 控制台看到跟踪的函数调用。请注意，**initWithURL:**方法被调用来初始化一个新的 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 timeou
tInterval:0x106388b80
```

6.9.3.4.2. 原生库跟踪

正如本章前面所讨论的，iOS 应用程序也可以包含原生代码（C/C++代码），它也可以使用 **frida-trace** CLI 进行追踪。例如，你可以通过运行以下命令追踪对 open 函数的调用。

```
frida-trace -U -i "open" sg.vp.UnCrackable1
```

使用 Frida 追踪原生代码的整体方法和进一步的随机应变与 Android "跟踪 "部分讨论的类似。

不幸的是，没有像 **strace** 或 **ftrace** 这样的工具可以用来跟踪 iOS 应用程序的系统调用或函数调用。只有 **DTrace** 存在，它是一个非常强大和通用的跟踪工具，但它只适用于 MacOS，不适用 iOS。

6.9.3.5. 基于仿真的跟踪

6.9.3.5.1. iOS 模拟器

Apple 在 Xcode 中提供了一个模拟器应用，它为 iPhone、iPad 或 Apple Watch 提供了一个真实的 iOS 设备外观的用户界面。它允许你在开发过程中快速建立原型并测试你的应用程序的调试版本，但实际上它不是一个仿真器。模拟器和仿真器之间的区别已在 "基于仿真的动态分析 "一节中讨论。

在开发和调试一个应用程序时，Xcode 工具链会生成 x86 代码，这些代码可以在 iOS 模拟器中执行。然而，对于发布构建，只生成 ARM 代码（与 iOS 模拟器不兼容）。这就是为什么从 Apple 应用 Store 下载的应用程序不能用于在 iOS 模拟器上进行任何形式的应用分析。

6.9.3.5.2. Corellium

Corellium 是一个商业工具，它提供运行真实 iOS 固件的虚拟 iOS 设备，是唯一公开可用的 iOS 仿真器。由于它是一个专有产品，所以关于实现的信息不多。Corellium 没有提供社区许可，因此我们不会对其使用进行过多的详细介绍。

Corellium 允许你启动一个设备（无论是否越狱）的多个实例，这些实例可以作为本地设备访问（通过简单的 VPN 配置）。它有能力拍摄和恢复设备状态的快照，还提供了一个方便的基于 Web 的设备 shell。最后，也是最重要的一点，由于其 "仿真器 "的性质，你可以执行从 Apple 应用 Store 下载的应用程序，使任何类型的应用程序分析成为可能，因为你知道它来自真正的 iOS（越狱）设备。

请注意，为了在 Corellium 设备上安装 IPA，它必须是未加密的，并使用有效的 Apple 开发者证书签名。查看更多信息这里。

6.9.4. 二进制分析

在 Android 的 "动态分析 "部分已经讨论了使用二进制分析框架进行二进制分析。我们建议你重温这一节，温习一下关于这个问题的概念。

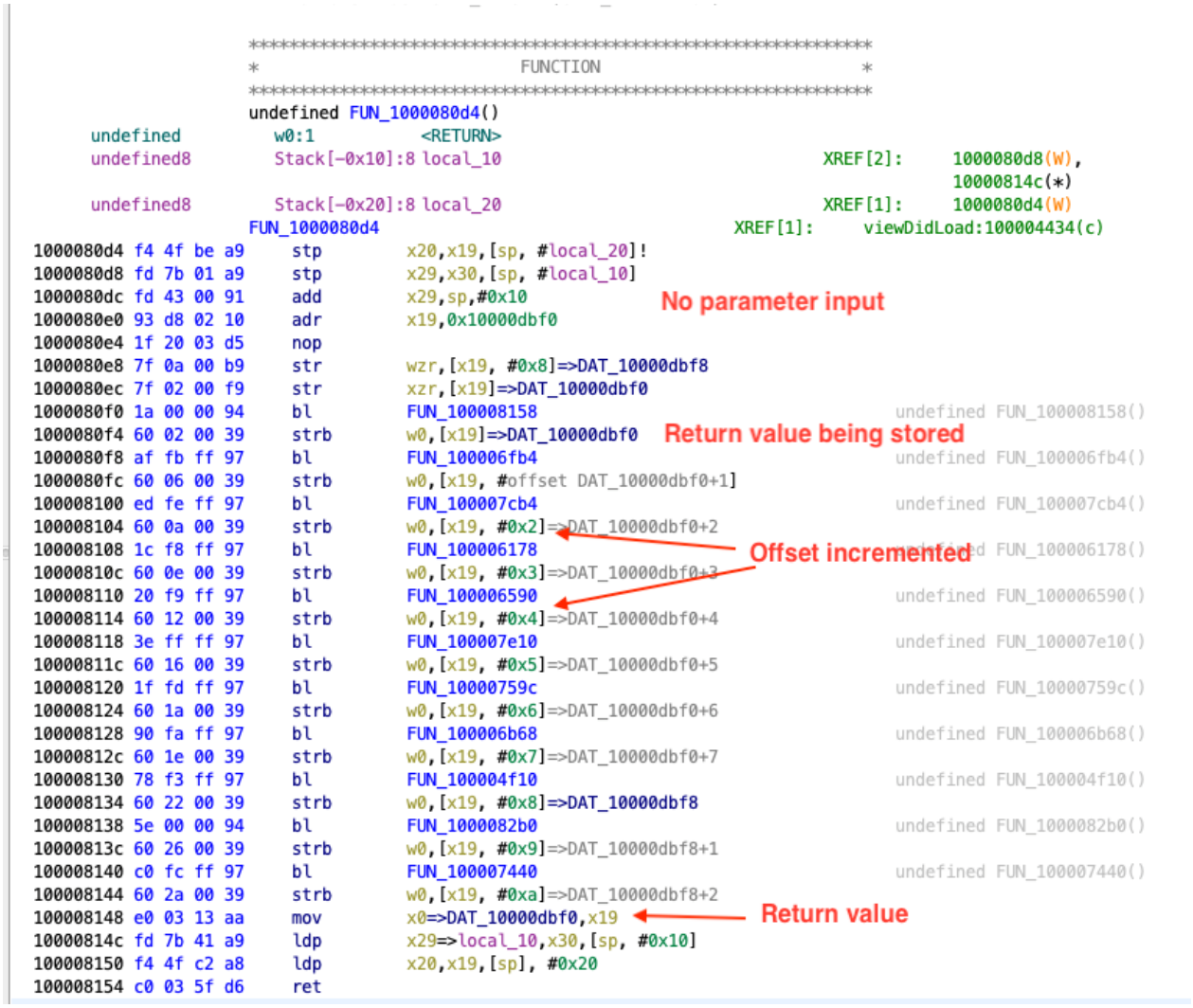
对于 Android，我们使用 Angr 的符号执行引擎来解决一个挑战。在本节中，我们将首先使用 Unicorn 来解决适用于 iOS 的 UnCrackable 应用级别 1 的挑战，然后我们将重新使用 Angr 二进制分析框架来分析挑战，但我们将使用其具体执行（或动态执行）功能来代替符号执行。

6.9.4.1. Unicorn

Unicorn 是一个轻量级、多架构的 CPU 仿真器框架，它基于 QEMU，并通过增加特别为 CPU 仿真制作的有用功能而超越它。Unicorn 提供了执行处理器指令所需的基本基础组件。在本节中，我们将使用 Unicorn 的 Python 绑定来解决适用于 iOS 的 UnCrackable 应用级别 1。

要使用 Unicorn 的全部功能，我们需要实现所有必要的基础组件，这些基础组件一般都可以从操作系统中获得，比如二进制加载器、链接器和其他依赖项，或者使用另一个更高级别的框架，比如 Qiling，它利用 Unicorn 来模拟 CPU 指令，但能理解操作系统内容。然而，这对于这种非常本地化的挑战来说是多余的，因为只执行二进制的一小部分就足够了。

在 "审查反汇编的原生代码 "部分进行人工分析时，我们确定地址为 0x1000080d4 的函数负责动态地生成秘密字符串。正如我们即将看到的，所有必要的代码在二进制文件中几乎是自成一体的，这使得这是一个使用像 Unicorn 这样的 CPU 仿真器的完美场景。



如果我们分析这个函数和随后的函数调用，我们会发现它没有硬性依赖任何外部库，也没有执行任何系统调用。对函数的唯一外部访问发生在实例地址 0x1000080f4 处，那里有一个值被存储到地址 0x10000dbf0，它映射到__data 部分。

因此，为了正确模拟这部分代码，除了__text 部分（包含指令），我们还需要加载__data 部分。

为了用 Unicorn 解决这个难题，我们将执行以下步骤。

- 通过运行 **`lipo -thin arm64 <app_binary> -output uncrackable.arm64`**（也可以使用 ARMv7），获得二进制的 ARM64 版本。
- 从二进制文件中提取__text 和__data 部分。

- 创建并映射将被用作堆栈内存的内存。
- 创建内存并加载 __text 和 __data 部分。
- 通过提供开始和结束地址执行二进制文件。
- 最后，转储函数的返回值，在本例中是我们的秘密字符串。

为了从 Mach-O 二进制文件中提取__text 和__data 部分的内容，我们将使用 LIEF，它为操作多种可执行文件格式提供了一个方便的抽象概念。在加载这些部分到内存之前，我们需要确定它们的基址，例如通过使用 Ghidra、Radare2 或 IDA Pro。

从上表中，我们将使用__text 的基址 0x10000432c 和__data 部分的基址 0x10000d3e8 来在内存中加载它们。

> 在为 Unicorn 分配内存时，内存地址应该是 4k 页对齐的，同时分配的大小应该是 1024 的倍数。

下面的脚本模拟了 0x1000080d4 的函数并转储了秘密字符串。

```
import lief
from unicorn import *
from unicorn.arm64_const import *

# --- 从二进制提取 __text 和 __data 部分内容 ---
binary = lief.parse("uncrackable.arm64")
text_section = binary.get_section("__text")
text_content = text_section.content

data_section = binary.get_section("__data")
data_content = data_section.content

# --- 为ARM64 执行设置Unicorn ---
arch = "arm64le"
emu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)

# --- 建了栈内存---
addr = 0x40000000
size = 1024*1024
emu.mem_map(addr, size)
emu.reg_write(UC_ARM64_REG_SP, addr + size - 1)

# ---加载文本部分 --
base_addr = 0x100000000
```

```
tmp_len = 1024*1024
text_section_load_addr = 0x10000432c
emu.mem_map(base_addr, tmp_len)
emu.mem_write(text_section_load_addr, bytes(text_content))

# --- 加载数据部分 ---
data_section_load_addr = 0x10000d3e8
emu.mem_write(data_section_load_addr, bytes(data_content))

# ---劫持 stack_chk_guard ---
# 没有这步会导致抛出在 0x0 内存不可用异常
emu.mem_map(0x0, 1024)
emu.mem_write(0x0, b"00")


# --- 执行从 0x1000080d4 到 0x100008154 ---
emu.emu_start(0x1000080d4, 0x100008154)
ret_value = emu.reg_read(UC_ARM64_REG_X0)

# --- 转储返回值 ---
print(emu.mem_read(ret_value, 11))
```

> 你可能注意到，在地址 0x0 处有一个额外的内存分配，这是一个围绕堆栈_chk_guard 检查的简单修改。没有这个，就会出现无效的内存读取错误，二进制就不能执行。有了这个修改，程序将访问 0x0 的值，并将其用于 stack_chk_guard 的检查。

总而言之，在执行二进制文件之前，使用 Unicorn 确实需要一些额外的设置，但一旦完成，这个工具可以帮助提供对二进制文件的深入了解。它提供了执行全部二进制文件或其有限部分的灵活性。Unicorn 还公开了 API，以便将劫持附加到执行中。利用劫持，你可以在执行过程中的任何时候观察程序的状态，甚至操纵寄存器或变量值，强行探索程序中的其他执行分支。在 Unicorn 中运行二进制文件的另一个好处是，你不需要担心各种检查，如 root/越狱检测或调试器检测等。

6.9.4.2. Angr

Angr 是一个非常通用的工具，提供多种技术以促进二进制分析，同时支持各种文件格式和硬件指令集。

Angr 中的 Mach-O 后端没有得到很好的支持，但对于我们的案例来说，它工作得非常好。

在 "审查反汇编的原生代码 "一节中人工分析代码时，我们达到了一个点，执行进一步的手动分析是很麻烦的。位于偏移量 0x1000080d4 的函数被确定为包含秘密字符串的最终目标。

如果我们重新审视这个函数，我们可以看到它涉及多个子函数的调用，有趣的是，这些函数都没有依赖其他库调用或系统调用。这是一个使用 Angr 的具体执行引擎的完美案例。按照下面的步骤来解决这个难题。

- 通过运行 **`lipo -thin arm64 <app_binary> -output uncrackable.arm64`**（也可使用 ARMv7），获得 ARM64 版本的二进制文件。
- 通过加载上述二进制文件创建一个 Angr 项目。
- 通过传递要执行的函数的地址来获得一个 **`callable`**（可调用）的对象。来自 Angr 文档："可调用对象是二进制文件中一个函数的表示，可以像原生 python 函数一样进行交互"。
- 将上述可调用对象传递给具体的执行引擎，在本例中是 **`claripy.backends.concrete`**。
- 访问内存并从上述函数返回的指针中提取字符串。

```
import angr
import claripy

def solve():

    # Load the binary by creating angr project.
    project = angr.Project('uncrackable.arm64')

    # Pass the address of the function to the callable
    func = project.factory.callable(0x1000080d4)

    # Get the return value of the function
    ptr_secret_string = claripy.backends.concrete.convert(func()).value
    print("Address of the pointer to the secret string: " + hex(ptr_secret
_string))

    # Extract the value from the pointer to the secret string
    secret_string = func.result_state.mem[ptr_secret_string].string.concre
te
    print(f"Secret String: {secret_string}")

solve()
```

以上，Angr 在其一个具体执行引擎提供的执行环境中执行了一个 ARM64 代码。结果从内存中访问，就像该程序在真实设备上执行一样。这个案例是一个很好的例子，二进制分析框架使我们能够对二进制进行全面的分析，即使没有运行二进制所需的专门设备。

6.9.5. 篡改和运行时插桩

6.9.5.1. 修补、重新包装和重新签名

是时候认真对待了! 正如你已经知道的，IPA 文件实际上是 ZIP 档案，所以你可以使用任何 ZIP 工具来解压档案。

```
unzip UnCrackable_Level1.ipa
```

6.9.5.1.1. 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 级别 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...
```

6.9.5.1.2. 修补示例：让应用程序可调试

默认情况下，Apple 应用 Store 中的应用是无法调试的。为了调试一个 iOS 应用程序，它必须启用 get-task-allow 权限。这个权限允许其他进程（如调试器）附加到应用程序。Xcode 并没有在发布配置文件中添加 get-task-allow 权限；它只是在开发配置文件中被列入白名单并添加。

因此，要调试从 App Store 获得的 iOS 应用程序，需要用带有 get-task-allow 权限的开发配置文件重新签名。如何重新签署一个应用程序将在下一节讨论。

6.9.5.1.3. 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，因此不会在非越狱设备上运行。你需要替换配置文件，用配置文件中列出的证书来签署主可执行文件和你所包含的文件（例如 FridaGadget.dylib）。

首先，让我们把我们自己的配置文件添加到软件包中：

```
cp AwesomeRepackaging.mobileprovision Payload/UnCrackable\ Level\ 1.app/embed
ded.mobileprovision
```

接下来，我们需要确保 **Info.plist** 中的包 ID 与配置文件中指定的包 ID 匹配，因为 代码签名工具在签名期间将从 **Info.plist** 读取 包 ID；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，我们使用代码签名工具重新签名两个二进制文件。您需要使用您拥有签名标识(在本例中为 8004380F331DCA22CC1B47FB1A805890AE41C938)，你可以通过执行 **security find-identity -v** 这个命令输出。

```
$ 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 signatu
re
```

entitlements.plist 是你为你的空 iOS 项目创建的文件。

```
$ /usr/bin/codesign --force --sign 8004380F331DCA22CC1B47FB1A805890AE41C938 -
-entitlements entitlements.plist Payload/UnCrackable\ Level\ 1.app/UnCrackabl
e\ Level\ 1
Payload/UnCrackable Level 1.app/UnCrackable Level 1: replacing existing signa
ture
```

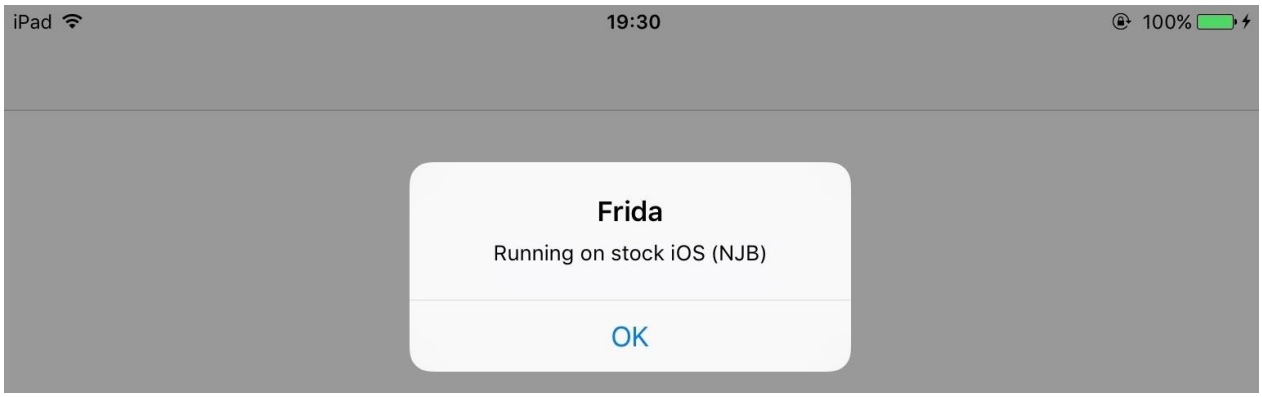
现在您应该准备好运行修改后的应用程序了。使用 ios-deploy 在设备上部署并运行应用程序：

```
ios-deploy --debug --bundle Payload/UnCrackable\ Level\ 1.app/
```

如果一切顺利，应用程序应该在调试模式启动，并连接 LLDB。然后，Frida 也应该能够附加到该应用程序。你可以通过 frida-ps 命令来验证这一点：

```
$ 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。Apple 的权限故障排除页面也是一个有用的资源。

6.9.4.1.4. 修补 React 原生应用程序

如果 React 原生框架已用于开发，则主要应用程序代码在文件 **`Payload/[APP].app/main.jsbundle`** 中。此文件包含 JavaScript 代码。大多数情况下，这个文件中的 JavaScript 代码是最小的。使用 JSlery 工具，可以还原可读的文件版本，这将允许代码分析。JStillery 的 CLI 版本和本地服务器比在线版本更好，因为后者会向第三方披露源代码。

在安装时，从 iOS 10 开始，应用程序存档被解压到 **`/private/var/containers/Bundle/Application/[GUID]/[APP].app`** 这个文件夹中，因此可以在这个位置修改主要的 JavaScript 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. 采用以 “`Application:`” 开始的行中的引用路径。

使用以下方法修补 JavaScript 文件：

1. 浏览到应用程序文件夹。
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代码并进行修补。
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。

6.9.5.2. 动态插桩

6.9.5.2.1. 信息收集

在本节中，我们将学习如何使用 Frida 来获取运行中的应用程序的信息。

6.9.5.2.2. 获取已加载的类和它们的方法

在 Frida REPL Objective-C 运行时，**ObjC** 命令可以用来访问运行中的应用程序的信息。在 **ObjC** 命令中，函数 **enumerateLoadedClasses** 列出了一个给定应用程序的加载类。

```
$ frida -U -f com.iOweApp

[iPhone::com.iOweApp]-> ObjC.enumerateLoadedClasses()
{
    "/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation": [
        "__NSBlockVariable__",
        "__NSGlobalBlock__",
        "__NSFinalizingBlock__",
        "__NSAutoBlock__",
        "__NSMallocBlock__",
        "__NSStackBlock__"
    ],
    "/private/var/containers/Bundle/Application/F390A491-3524-40EA-B3F8-6C1FA
105A23A/iOweApp.app/iOweApp": [
        "JailbreakDetection",
        "CriticalLogic",
        "ViewController",
        "AppDelegate"
    ]
}
```

使用 **ObjC.classes.<classname>.$ownMethods**，可以列出每个类中声明的方法。

```
[iPhone::com.iOweApp]-> ObjC.classes.JailbreakDetection.$ownMethods
[
    "+ isJailbroken"
]

[iPhone::com.iOweApp]-> ObjC.classes.CriticalLogic.$ownMethods
[
    "+ doSha256:",
    "- a:",
    "- AES128Operation:data:key:iv:",
    "- coreLogic",
    "- bat",
    "- b:",
    "- hexString:"
]
```

6.9.5.2.3. 获取已加载的库

在 Frida REPL 中，与进程有关的信息可以通过 **Process** 命令获得。在 **Process** 命令中，函数 **enumerateModules** 列出了加载到进程内存中的库。

```
[iPhone::com.iOweApp]-> Process.enumerateModules()
[
    {
        "base": "0x10008c000",
        "name": "iOweApp",
        "path": "/private/var/containers/Bundle/Application/F390A491-3524-40E
A-B3F8-6C1FA105A23A/iOweApp.app/iOweApp",
        "size": 49152
    },
    {
        "base": "0x1a1c82000",
        "name": "Foundation",
        "path": "/System/Library/Frameworks/Foundation.framework/Foundation",
        "size": 2859008
    },
    {
        "base": "0x1a16f4000",
        "name": "libobjc.A.dylib",
        "path": "/usr/lib/libobjc.A.dylib",
        "size": 200704
    },

    ...
```

同样，也可以获得与各种线程有关的信息。

```
Process.enumerateThreads()
[
    {
        "context": {
            ...
        },
        "id": 1287,
        "state": "waiting"
    },

    ...
```

Process 命令暴露了多种函数，可以根据需要进行浏览。一些有用的函数是 **findModuleByAddress**、**findModuleByName** 和 **enumerateRanges**，还有其他一些。

6.9.5.2.4. 方法劫持

6.9.5.2.4.1. Frida

在 "执行跟踪 "一节中，我们在 Safari 浏览器中导航到一个网站时使用了 frida-trace，发现 **initWithURL:**方法被调用来初始化一个新的 URL 请求对象。我们可以在 Apple 开发者网站上查到这个方法的声明。

```
- (instancetype)initWithURL:(NSURL *)url;
```

使用这些信息，我们可以编写一个 Frida 脚本，它拦截 **initWithURL:**方法，并显示传递给该方法的 URL。完整的脚本如下。确保您阅读代码和里面注释以了解发生了什么。

```
import sys
import frida


# 需要注入的 JavaScript
frida_code = """

    // 获得对 NSURLRequest 类的 initWithURL:方法的引用
    var URL = ObjC.classes.NSURLRequest["- initWithURL:"];

    // 拦截方法
    Interceptor.attach(URL.implementation, {
        onEnter: function(args) {
            // Get a handle on NSString
            var NSString = ObjC.classes.NSString;
```

```
            // 获得对 NSLog 函数的引用，并使用它来显示 URL 值
            // args[2] 引用第一个方法参数(NSURL *url)
            var NSLog = new NativeFunction(Module.findExportByName('Foundatio
n', 'NSLog'), 'void', ['pointer', '...']);

            // 在与 Objective-C APIs 交互之前，我们应该始终初始化一个自动释放池
            var pool = ObjC.classes.NSAutoreleasePool.alloc().init();

            try {
                // 给出一个 NativePointer，创建一个 JS 绑定.
                var myNSURL = new ObjC.Object(args[2]);

                // 从 JS 字符串对象中创建一个不可变的 ObjC 字符串对象.
                var str_url = NSString.stringWithString_(myNSURL.toString());

                // 调用 iOS NSLog 函数，将 URL 打印到 iOS 设备的日志中
                NSLog(str_url);

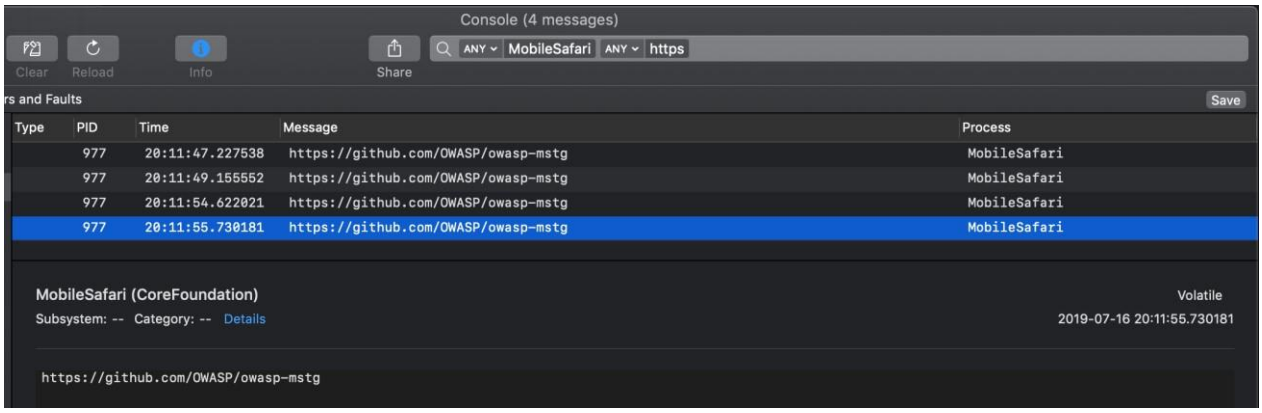
                // 使用 Frida 的 console.log 来打印 URL 到你的终端
                console.log(str_url);

            } 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-mastg；您应该在日志以及终端中看到 Frida 的输出。



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 的教程和示例。

6.9.5.2.5. 进程探索

在测试一个应用程序时，进程探索可以为测试人员提供对应用程序进程内存的深入了解。它可以通过运行时插桩来实现，并允许执行以下任务：

- 获取内存映射和加载的库。
- 搜索某些数据的出现。
- 进行搜索后，获得内存映射中某个偏移量的位置。
- 执行内存转储，离线检查或逆向工程二进制数据。
- 对运行中的二进制或框架进行逆向工程。

正如你所看到的，这些任务是相当支持性的和/或被动的，它们将帮助我们收集数据和信息，以支持其他技术。因此，它们通常与其他技术结合使用，如方法劫持。

在下面的章节中，你将使用 r2frida 来直接从应用程序的运行时中获取信息。首先，打开一个 r2frida 会话到目标应用程序（如 iGoat-Swift），它应该在你的 iPhone 上运行（通过 USB 连接）。使用以下命令：

```
r2 frida://usb//iGoat-Swift
```

6.9.5.2.5.1. 检查内存映射

你可以通过运行 \dm 检索应用程序的内存映射

```
[0x00000000]> \dm
0x0000000100b7c000 - 0x0000000100de0000 r-x /private/var/containers/Bundle/Ap
plication/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
0x0000000100de0000 - 0x0000000100e68000 rw- /private/var/containers/Bundle/Ap
plication/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
0x0000000100e68000 - 0x0000000100e97000 r-- /private/var/containers/Bundle/Ap
plication/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
...
0x0000000100ea8000 - 0x0000000100eb0000 rw-
0x0000000100eb0000 - 0x0000000100eb4000 r--
0x0000000100eb4000 - 0x0000000100eb8000 r-x /usr/lib/TweakInject.dylib
0x0000000100eb8000 - 0x0000000100ebc000 rw- /usr/lib/TweakInject.dylib
0x0000000100ebc000 - 0x0000000100ec0000 r-- /usr/lib/TweakInject.dylib
0x0000000100f60000 - 0x00000001012dc000 r-x /private/var/containers/Bundle/Ap
plication/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/Frameworks/Rea
lm.framework/Realm
```

当你在搜索或探索应用程序的内存时，你可以随时验证你当前的偏移量在内存映射中的位置。你可以简单地运行**\dm.**，而不是记下并搜索这个列表中的内存地址。你会在下面的 "内存搜索 "部分找到一个例子。

如果你只对应用程序加载的模块（二进制文件和库）感兴趣，你可以使用命令**\il** 来列出它们。

```
[0x00000000]> \il
0x0000000100b7c000 iGoat-Swift
0x0000000100eb4000 TweakInject.dylib
0x00000001862c0000 SystemConfiguration
0x00000001847c0000 libc++.1.dylib
0x0000000185ed9000 Foundation
0x000000018483c000 libobjc.A.dylib
0x00000001847be000 libSystem.B.dylib
0x0000000185b77000 CFNetwork
0x0000000187d64000 CoreData
0x00000001854b4000 CoreFoundation
0x00000001861d3000 Security
0x000000018ea1d000 UIKit
0x0000000100f60000 Realm
```

正如你所期望的，你可以把库的地址与内存映射联系起来：例如，主应用程序二进制 iGoat-Swift 位于 0x0000000100b7c000，Realm 框架位于 0x0000000100f60000。

你也可以用 objection 来显示同样的信息。

```
$ objection --gadget OWASP.iGoat-Swift explore

OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # memory list modules
```

```
Save the output by adding `--json modules.json` to this command

Name                                Base         Size                  Path
----------------------------------  -----------  --------------------  --------
-------------------------------------------------------------------------
iGoat-Swift                         0x100b7c000  2506752 (2.4 MiB)     /var/con
tainers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGo...
TweakInject.dylib                   0x100eb4000  16384 (16.0 KiB)      /usr/lib
/TweakInject.dylib
SystemConfiguration                 0x1862c0000  446464 (436.0 KiB)    /System/
Library/Frameworks/SystemConfiguration.framework/SystemConfiguratio...
libc++.1.dylib                      0x1847c0000  368640 (360.0 KiB)    /usr/lib
/libc++.1.dylib
```

6.9.5.2.5.2. 内存中搜索

内存搜索是一种非常有用的技术，可以测试可能存在于应用程序内存中的敏感数据。

参见 r2frida 关于搜索命令的帮助(\/?)以了解搜索命令并获得一个参数列表。下面显示的只是其中的一个子集。

```
[0x00000000]> \/?
 /      search
 /j     search json
 /w     search wide
 /wj    search wide json
 /x     search hex
 /xj    search hex json
...
```

你可以通过使用搜索设置**\e~search** 来调整你的搜索。例如，**\e search.quiet=true**;将只打印结果并隐藏搜索进度。

```
[0x00000000]> \e~search
e search.in=perm:r--
e search.quiet=false
```

现在，我们将继续使用默认值，集中精力进行字符串搜索。在这第一个例子中，你可以从搜索你知道应该位于应用程序的主二进制文件中的东西开始。

```
[0x00000000]> \/ iGoat
Searching 5 bytes: 69 47 6f 61 74
Searching 5 bytes in [0x0000000100b7c000-0x0000000100de0000]
...
hits: 509
```

```
0x100d7d332 hit2_0 iGoat_Swift24StringAnalysisExerciseVCC
0x100d7d3b2 hit2_1 iGoat_Swift28BrokenCryptographyExerciseVCC
0x100d7d442 hit2_2 iGoat_Swift23BackgroundingExerciseVCC
0x100d7d4b2 hit2_3 iGoat_Swift9AboutCellC
0x100d7d522 hit2_4 iGoat_Swift12FadeAnimatorV
```

现在查看第一个命中，找到它，并检查你在内存映射中的当前位置。

```
[0x00000000]> s 0x100d7d332
[0x100d7d332]> \dm.
0x0000000100b7c000 - 0x0000000100de0000 r-x /private/var/containers/Bundle/Ap
plication/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
```

正如预期的那样，你位于 iGoat-Swift 主二进制的区域（r-x，读和执行）。在上一节中，你看到主二进制位于 0x0000000100b7c000 和 0x0000000100e97000 之间。

现在，对于这第二个例子，你可以搜索一些不在应用程序二进制中也不在任何加载的库中的东西，通常是用户输入。打开 iGoat-Swift 应用程序，在菜单中导航到认证 **Authentication** ->远程认证 **Remote Authentication**->开始 **Start**。在那里你会发现一个你可以覆盖的密码字段。写下字符串 "owasp-mstg"，但先不要点击登录 **Login**。执行以下两个步骤。

```
[0x00000000]> \/ owasp-mstg
hits: 1
0x1c06619c0 hit3_0 owasp-mstg
```

事实上，在地址 0x1c06619c0 处可以找到该字符串。使用 **s** 到那里，用**\dm.**检索当前的内存区域。

```
[0x100d7d332]> s 0x1c06619c0
[0x1c06619c0]> \dm.
0x00000001c0000000 - 0x00000001c8000000 rw-
```

现在你知道这个字符串位于内存映射的 rw-（读写）区域内。

此外，你可以搜索该字符串的宽版本（/w）的出现，并再次检查其内存区域。

这次我们为所有@@全局命中 **hit5_***.运行\dm.命令

```
[0x00000000]> /w owasp-mstg
Searching 20 bytes: 6f 00 77 00 61 00 73 00 70 00 2d 00 6d 00 73 00 74 00 67
00
Searching 20 bytes in [0x0000000100708000-0x000000010096c000]
...
hits: 2
```

```
0x1020d1280 hit5_0 6f007700610073007002d006d00730074006700
0x1030c9c85 hit5_1 6f007700610073007002d006d00730074006700

[0x00000000]> \dm.@@ hit5_*
0x0000000102000000 - 0x0000000102100000 rw-
0x0000000103084000 - 0x00000001030cc000 rw-
```

它们处于不同的 rw-区域。请注意，搜索字符串的宽版本有时是找到它们的唯一方法，你将在下一节中看到。

内存搜索对于快速了解某些数据是否位于主应用程序的二进制文件中、共享库中或其他区域中非常有用。你也可以用它来测试应用程序关于数据在内存中保存方式的行为。例如，你可以继续前面的例子，这次点击登录并再次搜索该数据的出现情况。另外，你可以检查在登录完成后是否还能在内存中找到这些字符串，以验证这些敏感数据在使用后是否从内存中被抹去了。

6.9.5.2.5.3. 内存转储

你可以用 objection 和 Fridump 来转储应用程序的进程内存。为了在非越狱设备上利用这些工具，iOS 应用必须用 **frida-gadget.so** 重新打包并重新签名。这个过程的详细解释在 "非越狱设备的动态分析 "部分。要在已越狱的手机上使用这些工具，只需安装并运行 frida-server。

有了 objection，就可以通过使用 **memory dump all** 命令来转储设备上正在运行的进程的所有内存。

```
$ objection explore

iPhone on (iPhone: 10.3.1) [usb] # 转储所有内存 /Users/foo/memory_iOS/memory
Dumping 768.0 KiB from base: 0x1ad200000  [##################################
##]  100%
Memory dumped to file: /Users/foo/memory_iOS/memory
```

或者，你可以使用 Fridump。首先，你需要你想转储的应用程序的名称，你可以用 **frida-ps** 得到。

```
$ frida-ps -U
 PID  Name
----  ------
1026  Gadget
```

之后，在 Fridump 中指定应用程序的名称。

```
$ 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% Complet
e

Running strings on all files:
Progress: [##################################################] 100.0% Complet
e

Finished! Press Ctrl+C
```

当你添加**-s** 参数时，所有字符串都从转储的原始内存文件中提取出来，并添加到文件 strings.txt 中，该文件存储在 Fridump 的转储目录中。

在这两种情况下，如果你在 radare2 中打开该文件，你可以使用它的搜索命令（/）。注意，首先我们做一个标准的字符串搜索，没有成功，接下来我们搜索一个宽字符串，成功找到我们的字符串 "owasp-mstg"。

```
$ r2 memory_ios
[0x00000000]> / owasp-mstg
Searching 10 bytes in [0x0-0x628c000]
hits: 0
[0x00000000]> /w owasp-mstg
Searching 20 bytes in [0x0-0x628c000]
hits: 1
0x0036f800 hit4_0 6f007700610073007000 2d006d00730074006700
```

接下来，我们可以用 **s 0x0036f800** 或 **s hit4_0** 寻找到它的地址，然后用 **psw**（代表打印宽字符串）来打印，或者用 **px** 来打印它的原始十六进制值。

```
[0x0036f800]> psw
owasp-mstg

[0x0036f800]> px 48
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x0036f800  6f00 7700 6100 7300 7000 2d00 6d00 7300  o.w.a.s.p.-.m.s.
0x0036f810  7400 6700 0000 0000 0000 0000 0000 0000  t.g.............
0x0036f820  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

注意，为了使用 **strings** 命令找到这个字符串，你必须使用**-e** 参数指定一个编码，在本例中，l 代表 16 位小字节序字符。

```
$ strings -e l memory_ios | grep owasp-mstg
owasp-mstg
```

6.9.5.2.5.4. 运行时逆向工程

运行时逆向工程可以被看作是逆向工程的即时版本，在你的主机上没有二进制数据。相反，你将直接从应用程序的内存中分析它。

我们将继续使用 iGoat-Swift 应用程序，用 r2frida **`r2 frida://usb//iGoat-Swift`** 打开一个会话，你可以通过使用 **`\i`** 命令开始显示目标二进制信息。

```
[0x00000000]> \i
arch                 arm
bits                 64
os                   darwin
pid                  2166
uid                  501
objc                 true
runtime              V8
java                 false
cylang               true
pageSize             16384
pointerSize          8
codeSigningPolicy    optional
isDebuggerAttached   false
cwd                  /
```

用 **`\is <lib>`** 搜索某个模块的所有符号，例如 **`\is libboringssl.dylib`**。

下面对包含 "aes"（~+aes）的符号进行不区分大小写的搜索（grep）。

```
[0x00000000]> \is libboringssl.dylib~+aes
0x1863d6ed8 s EVP_aes_128_cbc
0x1863d6ee4 s EVP_aes_192_cbc
0x1863d6ef0 s EVP_aes_256_cbc
0x1863d6f14 s EVP_has_aes_hardware
0x1863d6f1c s aes_init_key
0x1863d728c s aes_cipher
0x0 u ccaes_cbc_decrypt_mode
0x0 u ccaes_cbc_encrypt_mode
...
```

或者你可能更喜欢研究导入/导出的情况。比如说。

- 列出主二进制的所有导入：**`\ii iGoat-Swift`**。

- 列出 libc++.1.dylib 库的导出：**\iE /usr/lib/libc++.1.dylib**。

对于大的二进制文件，建议通过添加~..的方式将输出以管道形式输入到内部的 less 程序，即：**\ii iGoat-Swift~..**（如果不是这样，对于这个二进制文件，你会得到近 5000 行打印到你的终端）。

接下来你可能想看一下类：

```
[0x00000000]> \ic~+passcode
PSPasscodeField
_UITextFieldPasscodeCutoutBackground
UIPasscodeField
PasscodeFieldCell
...
```

列出类的属性

```
[0x19687256c]> \ic UIPasscodeField
0x000000018eec6680 - becomeFirstResponder
0x000000018eec5d78 - appendString:
0x000000018eec6650 - canBecomeFirstResponder
0x000000018eec6700 - isFirstResponder
0x000000018eec6a60 - hitTest:forEvent:
0x000000018eec5384 - setKeyboardType:
0x000000018eec5c8c - setStringValue:
0x000000018eec5c64 - stringValue
...
```

想象一下，你对 0x000000018eec5c8c - setStringValue:感兴趣。你可以用 s 0x000000018eec5c8c 寻找到该地址，分析该函数 af 并用 pd 10 打印其反汇编的 10 行。

```
[0x18eec5c8c]> pd 10
┌ (fcn) fcn.18eec5c8c 35
│   fcn.18eec5c8c (int32_t arg1, int32_t arg3);
│ bp: 0 (vars 0, args 0)
│ sp: 0 (vars 0, args 0)
│ rg: 2 (vars 0, args 2)
│           0x18eec5c8c      f657bd          not byte [rdi - 0x43]      ; arg1
│           0x18eec5c8f      a9f44f01a9      test eax, 0xa9014ff4
│           0x18eec5c94      fd              std
│       ┌─< 0x18eec5c95      7b02            jnp 0x18eec5c99
│       │   0x18eec5c97      a9fd830091      test eax, 0x910083fd
│           0x18eec5c9c      f30300          add eax, dword [rax]
│           0x18eec5c9f      aa              stosb byte [rdi], al
│       ┌─< 0x18eec5ca0      e003            loopne 0x18eec5ca5
```

```
|       |   0x18eec5ca2      02aa9b494197    add ch, byte [rdx - 0x68beb665] ;
 arg3
`             0x18eec5ca8      f4              hlt
```

最后，你可能想从某个二进制文件中检索字符串，并对其进行过滤，而不是对字符串进行全内存搜索，就像你离线用 radare2 时那样。为此，你必须找到二进制文件，寻找到它，然后运行**\iz**命令。

> 建议应用关键字过滤器**~<keyword>/~+<keyword>**来减少终端输出。如果只是想探索所有的结果，你也可以用管道把它们输送到内部的 less **\iz~..**

```
[0x00000000]> \il~iGoa
0x00000001006b8000 iGoat-Swift
[0x00000000]> s 0x00000001006b8000
[0x1006b8000]> \iz
Reading 2.390625MB ...
Do you want to print 8568 lines? (y/N) N
[0x1006b8000]> \iz~+hill
Reading 2.390625MB ...
[0x1006b8000]> \iz~+pass
Reading 2.390625MB ...
0x00000001006b93ed  "passwordTextField"
0x00000001006bb11a  "11iGoat_Swift20KeychainPasswordItemV0C5ErrorO"
0x00000001006bb164  "unexpectedPasswordData"
0x00000001006d3f62  "Error reading password from keychain - "
0x00000001006d40f2  "Incorrect Password"
0x00000001006d4112  "Enter the correct password"
0x00000001006d4632  "T@"UITextField",N,W,VpasswordField"
0x00000001006d46f2  "CREATE TABLE IF NOT EXISTS creds (id INTEGER PRIMARY KEY
 AUTOINCREMENT, username TEXT, password TEXT);"
0x00000001006d4792  "INSERT INTO creds(username, password) VALUES(?, ?)"
```

要了解更多，请参考 r2frida wiki。

6.9.6. 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/newsroomhttps://www.nccgroup.trust/au/about-us/newsroom-and-

events/blogs/2016/october/iOS-instrumentation-without-jailbreak/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/
- r2frida Wiki - https://github.com/enovella/r2frida-wiki/blob/master/README.md
- [#miller] - Charlie Miller, Dino Dai Zovi. The iOS Hacker's Handbook. Wiley, 2012 - https://www.wiley.com/en-us/iOS+Hacker%27s+Handbook-p-9781118204122
- [#levin] Jonathan Levin. Mac OS X and iOS Internals: To the Apple's Core. Wiley, 2013 - http://newosxbook.com/MOXiI.pdf

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逆向工程防御中，使在越狱设备上运行应用程序更加困难。这阻止了逆向工程师喜欢使用的一些工具和技术。像大多数其他类型的防御一样，越狱检测本身并不十分有效，但在整个应用程序的源代码中分散检查可以提高整个防篡改方案的有效性。此处是 iOS 的典型越狱检测技术列表。

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
/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
/var/log/syslog
```

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 目录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。

Swift:

```
do {
    let pathToFileInRestrictedDirectory = "/private/jailbreak.txt"
    try "This is a test.".write(toFile: pathToFileInRestrictedDirectory, atom
ically: true, encoding: String.Encoding.utf8)
    try FileManager.default.removeItem(atPath: pathToFileInRestrictedDirector
y)
    // 设备已越狱
} catch {
    // 设备未越狱
}
```

Objective-C:

```
NSError *error;
NSString *stringToBeWritten = @"This is a test.";
[stringToBeWritten writeToFile:@"/private/jailbreak.txt" atomically:YES
         encoding:NSUTF8StringEncoding error:&error];
if(error==nil){
   //设备已越狱
```

```
    return YES;
} else {
    //设备未越狱
    [[NSFileManager defaultManager] removeItemAtPath:@"/private/jailbreak.txt"
 error:nil];
}
```

6.10.1.1.3. 检查协议处理程序

你可以通过尝试打开 Cydia URL 来检查协议处理程序。几乎每个越狱工具都会默认安装的 Cydia 应用商店，即会安装 cydia://协议处理程序。

Swift:

```
if let url = URL(string: "cydia://package/com.example.package"), UIApplicatio
n.shared.canOpenURL(url) {
    // 设备已越狱
}
```

Objective-C:

```
if([[UIApplication sharedApplication] canOpenURL:[NSURL URLWithString:@"cydi
a://package/com.example.package"]]){
    // 设备已越狱
}
```

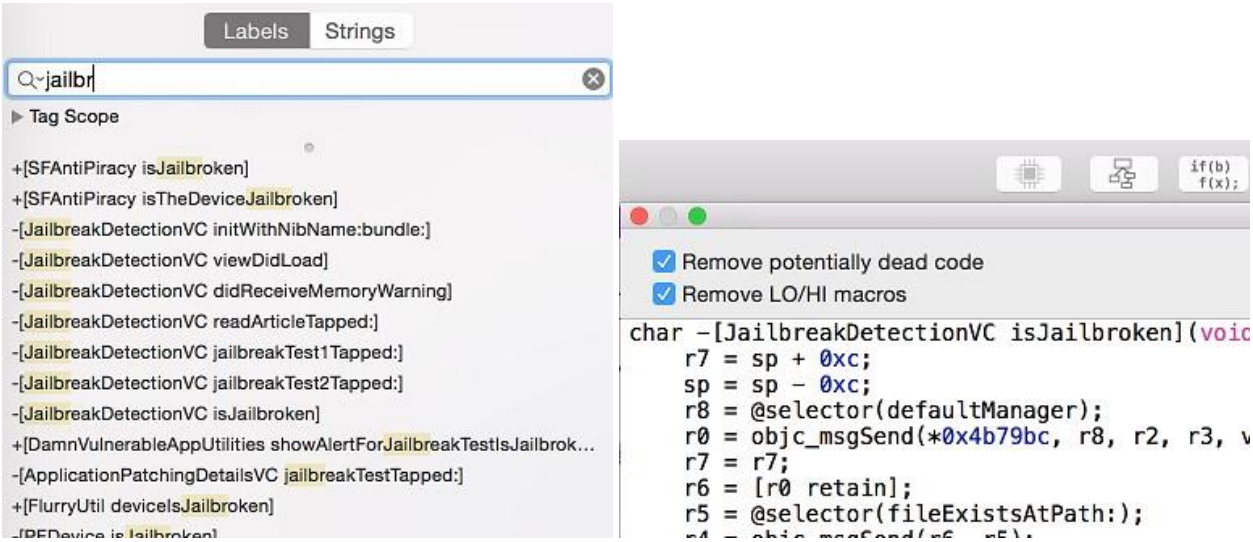
6.10.1.2. 绕过越狱检测

一旦在越狱设备上运行启用越狱检测的应用程序，您可能注意到以下情况之一：

1. 应用立即关闭，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，确保应用程序在未越狱设备上功能完全正常。应用程序可能会崩溃，或者它可能有一个导致它终止的错误。这可能发生在你测试应用程序的预生产版本时。

让我们再次以 Damn Vulnerable iOS 应用程序为例，看看如何绕过越狱检测。将二进制文件加载到 Hopper 中后，你需要等待，直到应用程序被完全反编译（查看顶部栏以检查状态）。然后在搜索框中寻找 "jail "字符串。你会看到两个类。`SFAntiPiracy` 和 `JailbreakDetectionVC`。你可能想对这些函数进行反编译，看看它们在做什么，特别是它们返回什么。



你可以看到，有一个类方法（+[SFAntiPiracy isTheDeviceJailbroken]）和一个实例方法（-[JailbreakDetectionVC isJailbroken]）。主要区别在于，我们可以在应用中注入 Cycript 并直接调用类方法，而实例方法则需要先寻找目标类的实例。函数 choose 将在内存堆中寻找一个给定类的已知签名，并返回一个实例数组。将一个应用程序置于一个理想的状态（以便类确实被实例化了）是很重要的。

让我们把 Cycript 注入我们的进程中（用 top 寻找你的 PID）：

```
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 类的实例。

```
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“Jailbreak Test 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，用 Cycript 绕过越狱检测是很简单的。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。我们可以通过用 Cycript 替换函数的实现来替换返回值。请注意，这实际上会替换其给定名称下的函数，所以如果该函数修改了应用程序中的任何东西，请小心副作用：

```
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 CydiaSubstrate，使用像 `MSHookMessageEx` 这样适当的劫持函数，调整后进行编译。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前期插桩，可以在之前的测试中获得的技能基础上进行。

我们将利用 Frida 的一个功能来绕过越狱检测，即所谓的前期插桩，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 JavaScript 劫持 `onEnter` 和 `onLeave` 回调函数。现在，通过 `value.replace` 替换返回值是很简单的，如下示例所示：

```
    onLeave: function (log, retval, state) {
    console.log("Function [JailbreakDetectionVC isJailbroken] originally retu
rned:"+ 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/DamnVulnerableIOSA
pp  -m "-[JailbreakDetectionVC isJailbroken]:"

Instrumenting functions...                                           `...
-[JailbreakDetectionVC isJailbroken]: Loaded handler at "./__handlers__/__Jai
lbreakDetectionVC_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]
```

注意对`-[JailbreakDetectionVC isJailbroken]`的两次调用，这相当于在应用程序的 GUI 上进行两次物理点击。

还有一种绕过依赖文件系统检查的越狱检测机制的方法是 objection。你可以在 jailbreak.ts 脚本中找到越狱绕过的实现。

请看下面一个 Python 脚本，用于劫持 Objective-C 方法和原生函数。

```
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 faccesset = Module.findExportByName("libsystem_kernel.dylib", "fa
ccessat");

    */

""")

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

在逆向过程中，使用调试器探索应用程序是一种非常强大的技术。你不仅可以跟踪包含敏感数据的变量，修改应用程序的控制流，还可以读取和修改内存和寄存器。

有几种适用于 iOS 的反调试技术，可以分为预防性和反应性，下面讨论其中的几种。作为第一道防线，你可以使用预防技术来阻止调试器附着在应用程序上。此外，您还可以采用反应性技术，使应用程序能够检测到调试器的存在，并有机会偏离正常行为。当这些技术适当地分布在整个应用程序中时，这些技术作为一种次要的或支持性的措施可以增加整体的防御强度。

处理高度敏感数据的应用程序的开发人员应该意识到，防止调试几乎是不可能的。如果应用程序是公开的，它可以在一个不受信任的设备上运行，而这个设备是在攻击者的完全控制之下。一个非常坚定的攻击者最终会设法绕过所有应用程序的反调试控制，方法是修补应用程序的二进制文件，或在运行时用 Frida 等工具动态地修改应用程序的行为。

根据 Apple 公司的说法，你应该 "将上述代码的使用限制在你的程序的调试构建中"。然而，研究表明，许多 App Store 应用程序经常包括这些检查。

6.10.2.1.1. 使用 ptrace

正如在 "iOS 系统上的篡改和逆向工程 "一章中所看到的，iOS 的 XNU 内核实现了一个 `ptrace` 系统调用，它缺乏正确调试进程所需的大部分功能（例如，它允许附加/步进，但不允许读/写内存和寄存器）。

尽管如此，iOS 实现的 `ptrace` 系统调用包含一个非标准的、非常有用的特性：防止进程的调试。这个特性以 `PT_DENY_ATTACH` 请求的形式实现，如官方 BSD 系统调用手册中所述。简单地说，它确保没有其他调试器可以附加到调用进程上；如果有调试器试图附加，该进程将终止。使用 `PT_DENY_ATTACH` 是一个相当著名的反调试技术，所以你可能在 iOS 的测试中经常遇到它。

在深入了解细节之前，重要的是要知道 `ptrace` 不是公共 iOS API 的一部分。非公共 API 是被禁止的，App Store 可能会拒绝包含它们的应用程序。正因为如此，`ptrace` 在代码中不被直接调用；它是在通过 dlsym 获得 `ptrace` 函数指针时被调用。

下面是上述逻辑的一个实现实例：

```
#import <dlfcn.h>
#import <sys/types.h>
#import <stdio.h>
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _dat
a);
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
}
```

为了演示如何绕过这种技术，我们将使用一个实现这种方法的反汇编二进制文件的例子：

```
__text:00019074                 MOVW            R1, #:lower16:(aPtrace - 0x19088) ; "ptrace"
__text:00019078                 MOV             R0, #0xFFFFFFFE ; 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（小端 0x00xBF）指令替换指令 BLXR6（小端 0xB0x47）。修补后，代码将类似于以下内容：

```
__text:00019078                 MOV             R0, #0xFFFFFFFE ; 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。

其他基于 ptrace 的反调试技术的绕过方法可以在 Alexander O'Mara 的《击败反调试技术：macOS ptrace 变体》中找到。

6.10.2.1.2. 使用 sysctl

另一种检测附加到调用进程的调试器的方法涉及 sysctl。根据 Apple 公司的文档，它允许进程设置系统信息（如果有适当的权限）或简单地检索系统信息（如进程是否正在被调试）。然而，请注意，仅仅一个应用程序使用 sysctl 的事实就可能视为反调试控制的一个指示，尽管并不总是这样。

下面的例子来自 Apple 文档档案，它检查了由调用 sysctl 返回的 info.kp_proc.p_flag 标志和适当的参数。

```
#include <assert.h>
#include <stdbool.h>
```

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/sysctl.h>

static bool AmIBeingDebugged(void)
    // 返回 true 如果当前进程正在被调试（包括
    // 在调试器下运行或附加了一个调试器）.
{
    int                 junk;
    int                 mib[4];
    struct kinfo_proc   info;
    size_t              size;

    // 初始化标志，这样，如果 sysctl 因为一些奇怪的原因
    // 失败，我们就会得到一个可预测的结果。

    info.kp_proc.p_flag = 0;

    // 初始化 mib，它告诉 sysctl 我们想要的信息，在这个例子
    // 中，我们正在寻找关于一个特定进程 ID 的信息。

    mib[0] = CTL_KERN;
    mib[1] = KERN_PROC;
    mib[2] = KERN_PROC_PID;
    mib[3] = getpid();

    // 调用 sysctl.

    size = sizeof(info);
    junk = sysctl(mib, sizeof(mib) / sizeof(*mib), &info, &size, NULL, 0);
    assert(junk == 0);

    // 如果 P_TRACED 标志被设置，我们就在被调试状态下。

    return ( (info.kp_proc.p_flag & P_TRACED) != 0 );
}
```

绕过这种检查的一种方法是给二进制文件打补丁。当上面的代码被编译后，后半部分代码的反汇编版本类似于以下内容。

```
__text:0000C12A ; ---------------------------------------------------------------------------
__text:0000C12A
__text:0000C12A loc_C12A                                ; CODE XREF: _AmIBeingDebugged:loc_C128↑j
__text:0000C12A                 LDR             R0, [SP,#0x228+var_1F8]
__text:0000C12C                 AND.W           R0, R0, #0x800
__text:0000C130                 STR             R0, [SP,#0x228+var_214]
__text:0000C132                 LDR             R0, [SP,#0x228+var_214]
__text:0000C134                 CMP             R0, #0
__text:0000C136                 MOVW            R0, #0
__text:0000C13A                 IT NE
__text:0000C13C                 MOVNE           R0, #1
__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           R0, R0, #1
__text:0000C15A                 ADD.W           SP, SP, #0x220
__text:0000C15E                 POP             {R7,PC}
__text:0000C160 ; ---------------------------------------------------------------------------
```

在偏移量为 0xC13C 的指令 MOVNE R0, #1 被修补后，改为 MOVNE R0, #0(字节码中为 0x00 0x20)，修补后的代码与下面类似：

```
__text:0000C12A
__text:0000C12A loc_C12A                                ; CODE XREF: _AmIBeingDebugged:loc_C128↑j
__text:0000C12A                 LDR             R0, [SP,#0x228+var_1F8]
__text:0000C12C                 AND.W           R0, R0, #0x800
__text:0000C130                 STR             R0, [SP,#0x228+var_214]
__text:0000C132                 LDR             R0, [SP,#0x228+var_214]
__text:0000C134                 CMP             R0, #0
__text:0000C136                 MOVW            R0, #0
__text:0000C13A                 IT NE
__text:0000C13C                 MOVNE           R0, #0
__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           R0, R0, #1
__text:0000C15A                 ADD.W           SP, SP, #0x220
__text:0000C15E                 POP             {R7,PC}
```

你也可以通过使用调试器本身并在调用 `sysctl` 时设置断点来绕过 `sysctl` 检查。这种方法在 iOS 反调试保护措施#2 中演示。

6.10.2.1.3. 使用 getppid

iOS 上的应用程序可以通过检查其父 PID 来检测它们是否被调试器启动。通常情况下，一个应用程序是由 launchd 进程启动的，它是在用户模式下运行的第一个进程，其 PID=1。然而，如果一个调试器启动了一个应用程序，我们可以观察到 `getppid` 返回一个不同于 1 的 PID。 这种检测技术可以在原生代码中实现（通过系统调用），使用 Objective-C 或 Swift，如下所示：

```
func AmIBeingDebugged() -> Bool {
    return getppid() != 1
}
```

与其他技术类似，这也有一个简单的绕过方法（例如，通过修补二进制文件或使用 Frida 劫持）。

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。我们还知道，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，确定的逆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：当文件使用应用程序、Keychain 中的键值对、`UserDefaults/NSUserDefaults`、SQLite 数据库或 Realm 数据库存储时，其完整性应受到保护。

6.10.3.1.1. 示例实现-应用程序源代码

Apple 公司用 DRM 处理了完整性检查。然而，额外的控制（如下面的例子）是可能的。对 `mach_header` 进行解析以计算指令数据的起始点，用于生成签名。接下来，签名将与给定的签名进行比较。确保生成的签名被存储或编码在其他地方。

```
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;  //Mach-O 头部指针
        struct load_command * cmd = (struct load_command *)(header + 1); //初
次加载指令
        // 现在通过加载命令进行遍历
        //以找到__TEXT 段的__text 部分
        for (uint32_t i = 0; cmd != NULL && i < header->ncmds; i++) {
            if (cmd->cmd == LC_SEGMENT) {
                // __TEXT 加载命令是一个 LC_SEGMENT 加载命令
                struct segment_command * segment = (struct segment_command *)
cmd;
                if (!strcmp(segment->segname, "__TEXT")) {
                    // 在__TEXT 段加载命令上停止，并通过对各部分进行检查
                    // 以找到__text 部分
                    struct section * section = (struct section *)(segment +
1);
                    for (uint32_t j = 0; section != NULL && j < segment->nsec
ts; j++) {
                        if (!strcmp(section->sectname, "__text"))
                            break; //在__text 部分加载命令时停止
                        section = (struct section *)(section + 1);
                    }
                    // 在这里得到__text 区的地址，__text 区的大小和虚拟内存地址，
这样我们就可以计算出__text 区的指针。
                    uint32_t * textSectionAddr = (uint32_t *)section->addr;
                    uint32_t textSectionSize = section->size;
                    uint32_t * vmaddr = segment->vmaddr;
                    char * textSectionPtr = (char *)((int)header + (int)textS
ectionAddr - (int)vmaddr);
                    // 计算数据的签名，将结果存储在一个字符串中，并与原始签名进行比
较。
                    unsigned char digest[CC_MD5_DIGEST_LENGTH];
                    CC_MD5(textSectionPtr, textSectionSize, digest);     //
计算签名
                    for (int i = 0; i < sizeof(digest); i++)             //
填充签名
                        sprintf(dst + (2 * i), "%02x", digest[i]);

                    //返回 strcmp(originalSignature, signature) == 0;验证签名是
否相符
```

```
                    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，你可以在一个给定的键值对或存储在设备上的文件上创建一个 HMAC 或签名。CommonCrypto 的实现对于创建 HMAC 来说是最好的。如果你需要加密，请确保你先加密，然后按照验证式加密中描述的方法进行 HMAC。

当您使用 CC 生成 HMAC 时：

1. 获取数据作为 NSMutableData。
2. 获取数据密钥（如果可能的话从 Keychain 获取）。
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。

```
// 分配一个缓冲区来保存摘要，并执行摘要。
NSMutableData* actualData = [getData];
//从 keychain 获取密钥
NSData* key = [getKey];
NSMutableData* digestBuffer = [NSMutableData dataWithLength:CC_SHA256_DIGE
ST_LENGTH];
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length], [actua
lData 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。

```
4.    NSData* hmac = [data subdataWithRange:NSMakeRange(data.length - CC_S
HA256_DIGEST_LENGTH, CC_SHA256_DIGEST_LENGTH)];
  NSData* actualData = [data subdataWithRange:NSMakeRange(0, (data.length
 - hmac.length))];
  NSMutableData* digestBuffer = [NSMutableData dataWithLength:CC_SHA256_D
IGEST_LENGTH];
  CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length], [ac
tualData bytes], (CC_LONG)[actualData length], [digestBuffer mutableByte
s]);
  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代码完整性检查时

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. 修补存储的任何用于校验代码完整性的哈希。
3. 使用 Frida 劫持文件系统 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. 效果评估

应用程序源代码完整性检查：

在设备上以未修改的状态运行应用程序，确保一切正常。然后使用 optool 对可执行文件打上补丁，按照 "基本安全测试 "一章中的描述重新签署应用程序，并运行它。应用程序应该检测到修改并以某种方式回应。至少，应用程序应该提醒用户和/或终止该应用程序。努力绕过防御措施，并回答以下问题。

- 机制能否被简单地绕过（例如：通过劫持单个 API 函数）？
- 通过静态和动态分析识别反调试代码有多困难？
- 您需要编写自定义代码来禁用防御吗？您需要多少时间？
- 你对绕过这些机制的难度有何评估？

存储完整性检查：

使用类似的方法。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？
- 你对绕过这些机制的难度有何评估？

6.10.4. 测试逆向工具检测（MSTG-RESILIENCE-4）

6.10.4.1. 概述

逆向工程师常用的工具、框架和应用程序的存在，可能表明有人试图对该应用程序进行逆向工程。其中一些工具只能在已越狱的设备上运行，而其他工具则迫使应用程序进入调试模式，或依赖于启动手机的后台服务。因此，一个应用程序可以通过不同的方式来检测逆向工程攻击，并对其作出反应，例如终止自己。

6.10.4.2. 检测方法

你可以通过寻找相关的应用程序包、文件、进程或其他工具特定的修改和组件，来检测已经以未修改的形式安装的流行逆向工程工具。在下面的例子中，我们将讨论检测 Frida 工具框架的不同方法，该框架在本指南中被广泛使用，在现实世界中也是如此。其他工具，如 Cydia Substrate 或 Cycript，也可以用类似的方法来检测。请注意，注入、劫持和 DBI（动态二进制工具）工具通常可以通过运行时的完整性检查隐式检测出来，这将在下面讨论。

例如，Frida 在其默认配置（注入模式）下以 frida-server 的名义运行在已越狱的设备上。当你明确地附加到一个目标应用程序（例如通过 frida-trace 或 Frida CLI）时，Frida 将一个 frida-agent 注入到该应用程序的内存。因此，你可能希望在附加到应用之后（而不是之前）在那里找到它。在 Android 上，验证这一点非常简单，因为你可以简单地在 proc 目录（`/proc/<pid>/maps`）中的进程 ID 的内存映射中 grep 查找 "frida "字符串。然而，在 iOS 上，`proc` 目录是不可用的，但你可以用函数`_dyld_image_count` 列出一个应用程序中加载的动态库。

Frida 也可以在所谓的嵌入式模式下运行，这也适用于非越狱设备。它包括将 frida-gadget 嵌入到 IPA 中，并迫使应用程序将其作为其本地库之一加载。

应用程序的静态内容，包括其 ARM 编译的二进制文件和其外部库，被存储在 `<Application>.app` 目录内。如果你检查 `/var/containers/Bundle/Application/<UUID>/<Application>.app` 目录的内容，你会发现嵌入的 frida-gadget 为 FridaGadget.dylib。

```
iPhone:/var/containers/Bundle/Application/AC5DC1FD-3420-42F3-8CB5-E9D77C4B287
A/SwiftSecurity.app/Frameworks root# ls -alh
total 87M
drwxr-xr-x 10 _installd _installd  320 Nov 19 06:08 ./
drwxr-xr-x 11 _installd _installd  352 Nov 19 06:08 ../
-rw-r--r--  1 _installd _installd  70M Nov 16 06:37 FridaGadget.dylib
-rw-r--r--  1 _installd _installd 3.8M Nov 16 06:37 libswiftCore.dylib
-rw-r--r--  1 _installd _installd  71K Nov 16 06:37 libswiftCoreFoundation.dy
lib
-rw-r--r--  1 _installd _installd 136K Nov 16 06:38 libswiftCoreGraphics.dyli
b
-rw-r--r--  1 _installd _installd  99K Nov 16 06:37 libswiftDarwin.dylib
-rw-r--r--  1 _installd _installd 189K Nov 16 06:37 libswiftDispatch.dylib
-rw-r--r--  1 _installd _installd 1.9M Nov 16 06:38 libswiftFoundation.dylib
-rw-r--r--  1 _installd _installd  76K Nov 16 06:37 libswiftObjectiveC.dylib
```

看着 Frida 留下的这些痕迹，你可能已经想到，检测 Frida 将是一个简单的任务。虽然检测这些库是简单的，但绕过这种检测也同样是简单的。对工具的检测是一场猫捉老鼠的游戏，事情会变得更加复杂。下表简要介绍了一组典型的 Frida 检测方法，并对其有效性进行了简短的讨论。

下面一些检测方法已在 iOS 安全套件中实现

方法	描述	讨论
检查环境中是否有相关组件	组件可以是打包的文件、二进制文件、库、进程和临时文件。对于 Frida 来说，这可能是在目标（越狱）系统中运行的 frida-server（负责通过 TCP 暴	对于非越狱设备上的 iOS 应用来说，检查正在运行的服务是不可能的。Swift 方法 CommandLine 在 iOS 上无法查询到运行中的进程信息，但有一些非官方的方法，比如使用 NSTask。尽管如此，当使用这种方法

	露 Frida 的守护程序）或应用程序加载的 frida 库。	时，应用程序在 App Store 审核过程中会被拒绝。没有其他公开的 API 可以用来查询运行中的进程或在 iOS 应用中执行系统命令。即使有可能，绕过这一点也很容易，只需重命名相应的 Frida 组件（frida-server/frida-gadget/frida-agent）即可。另一种检测 Frida 的方法是浏览加载的库列表并检查可疑的库（例如那些在其名称中包含 "frida "的库），这可以通过使用 `_dyld_get_image_name` 来实现。
检查开放的 TCP 端口	frida-server 进程默认绑定在 TCP 27042 端口上。测试这个端口是否开放是检测守护进程的另一种方法。	这个方法在检测默认模式下的 frida-server，但监听端口可以通过命令行参数改变，所以绕过这个方法是非常简单的。
检查响应 D-Bus AUTH 的端口	frida-server 使用 D-Bus 协议进行通信，所以你可以期待它对 D-Bus AUTH 作出回应。向每一个开放的端口发送 D-Bus AUTH 消息，并检查是否有回答，希望 frida-server 会显示出自己。	这是一种相当强大的检测 frida-server 的方法，但 Frida 提供了不需要 frida-server 的其他操作模式。

请记住，这个表格远非详尽无遗。例如，另外两种可能的检测机制是：

- 命名管道（frida-server 用于外部通信），或

- 检测 trampolines（参见 "防止绕过 iOS 应用程序中的 SSL 证书固定"，以获得进一步的解释和在 iOS 应用程序中检测 trampolines 的示例代码）

这两种方法都有助于检测 Substrate 或 Frida 的拦截器，但比如说，对 Frida 的 Stalker 不会有效。请记住，每一种检测方法的成功都取决于你是否使用越狱的设备，越狱的具体版本和方法和/或工具本身的版本。最后，这是保护在不受控制的环境（最终用户的设备）中处理的数据的猫捉老鼠游戏的一部分。

> 需要注意的是，这些控制措施只是增加了逆向工程过程的复杂性。如果使用，最好的办法是巧妙地结合这些控制措施，而不是单独使用它们。然而，它们都不能保证 100%的有效性，因为逆向工程人员总是可以完全进入设备，因此总是会赢。你还必须考虑到，将一些控制措施集成到你的应用程序中可能会增加你的应用程序的复杂性，甚至对其性能产生影响。

6.10.4.3. 效果评估

在你的测试设备上安装各种逆向工程工具和框架，启动该应用程序。至少包括以下内容。Frida, Cydia Substrate, Cycript 和 SSL Kill Switch。

该应用程序应该以某种方式对这些工具的存在作出反应。例如：

- 提醒用户并要求对运行环境负责
- 通过优雅地终止来防止执行
- 安全地擦除存储在设备上的任何敏感数据。
- 向后端服务器报告，例如，用于欺诈检测。

接下来，努力绕过逆向工程工具的检测，回答以下问题：

- 机制能否被简单地绕过（例如：通过劫持单个 API 函数）？
- 通过静态和动态分析识别反逆向工程代码有多困难？
- 您需要编写自定义代码来禁用防御吗？您需要多少时间？
- 你对绕过这些机制的难度有何评估？

在绕过逆向工程工具的检测时，下面的步骤应能指导你：

1. 对反逆向工程功能进行修补。通过使用 radare2/Cutter 或 Ghidra 对二进制文件进行修补，禁用不需要的行为。
2. 使用 Frida 或 Cydia Substrate 在 Objective-C/Swift 或原生层上劫持文件系统 API。返回一个原始文件的句柄，而不是修改后的文件。

关于修补和代码注入的例子，请参考 "iOS 系统上的篡改和逆向工程 "一章。

6.10.5. 测试仿真器检测（MSTG-RESILIENCE-5）

6.10.5.1. 概述

仿真器检测的目的是增加在仿真设备上运行应用程序的难度。这迫使逆向工程师击败仿真器检查或利用物理设备，从而禁止大规模设备分析所需的访问。

正如在基本安全测试章节的在 iOS 模拟器上测试一节中所讨论的，唯一可用的模拟器是 Xcode 附带的模拟器。模拟器二进制文件被编译为 x86 代码，而不是 ARM 代码，为真实设备（ARM 架构）编译的应用程序不会在模拟器中运行，因此，与拥有大量仿真选择的 Android 相比，iOS 应用程序的仿真保护并不是那么令人关注。

然而，自其发布以来，Corellium（商业工具）已经实现了真正的仿真，使其与 iOS 模拟器区分开来。除此之外，作为一个 SaaS 解决方案，Corellium 实现了大规模的设备分析，其限制因素只是可用资金。

随着 Apple Silicon（ARM）硬件的广泛使用，其可能不适用于对 x86/x64 架构的传统检查。一个潜在的检测策略是确定常用的仿真解决方案的功能和限制。例如，Corellium 不支持 iCloud、蜂窝服务、摄像头、NFC、蓝牙、App Store 访问或 GPU 硬件仿真（Metal）。因此，巧妙地将涉及任何这些功能的检查结合起来，可能是一个存在仿真环境的指标。

将这些结果与第三方框架，如 iOS 安全套件、Trusteer 或无代码解决方案，如 Appdome（商业解决方案）的结果配对，将提供一个良好的防线，防止利用仿真器的攻击。

6.10.6. 测试混淆（MSTG-RESILIENCE-9）

6.10.6.1. 概述

移动应用程序篡改和逆向工程 "一章介绍了几种著名的混淆技术，这些技术可用于一般的移动应用程序。

注意：下面介绍的所有技术可能无法阻止逆向工程师，但将所有这些技术结合起来，将大大增加他们的工作难度。这些技术的目的是阻止逆向工程师进行进一步分析。

以下技术可用于混淆一个应用程序：

- 名称混淆
- 指令替换
- 控制流扁平化
- 死代码注入
- 字符串加密

6.10.6.2. 名称混淆

标准编译器根据源代码中的类和函数名称生成二进制符号。因此，如果没有应用混淆技术，符号名称仍然是有意义的，可以很容易地从应用程序的二进制文件中直接读取。例如，一个检测越狱的函数可以通过搜索相关的关键词（如 "jailbreak"）来定位。下面的列表显示了从 Damn Vulnerable iOS App (DVIA-v2)中反汇编的函数 `JailbreakDetectionViewController.jailbreakTest4Tapped`。

```
__T07DVIA_v232JailbreakDetectionViewControllerC20jailbreakTest4TappedyypF:
stp         x22, x21, [sp, #-0x30]!
mov         rbp, rsp
```

经过混淆处理后，我们可以观察到符号的名称不再有意义，如下表所示。

```
__T07DVIA_v232zNNtWKQptikYUBNBgfFVMjSkvRdhhnbyyFySbyypF:
stp         x22, x21, [sp, #-0x30]!
mov         rbp, rsp
```

尽管如此，这只适用于函数、类和字段的名称。实际的代码仍然没有被修改，所以攻击者仍然可以阅读函数的反汇编版本，并试图了解其目的（例如检索安全算法的逻辑）。

6.10.6.3. 指令替换

这种技术用更复杂的表示方法取代了标准的二进制运算符，如加法或减法。例如，一个加法 x=a+b 可以表示为 x=-（-a）-（-b）。然而，使用相同的替换表示法很容易被逆向，所以建议为一个案例添加多个替换技术，并引入一个随机因素。这种技术很容易被反混淆，但是根据替换的复杂性和深度，应用这种技术仍然会很费时。

6.10.6.4. 控制流扁平化

控制流扁平化用更复杂的表示方法取代了原始代码。这种转换将函数的主体分解成基本块，并将它们全部放在一个带有控制程序流的开关语句的无限循环内。这使得程序流明显难以跟踪，因为它删除了通常使代码更容易阅读的自然条件结构。

original

```
i = 1;
s = 0;


while (i <= 100) {




  s += i;
  i++;


}
```

control-flow flattening applied

```
int swVar = 1;
while (swVar != 0) {
  switch (swVar) {
    case 1: {
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2: {
      if (i <= 100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3: {
      s += i;
      i++;
      swVar = 2;
      break;
    }
  }
}
```

图片显示了控制流扁平化是如何改变代码的（见 "通过控制流扁平化来混淆 C++程序"）。

6.10.6.5. 死代码注入

这种技术通过在程序中注入死代码使程序的控制流更加复杂。死代码是代码的分支，不影响原程序的行为，但增加了逆向工程过程的难度。

6.10.6.6. 字符串加密

应用程序通常编译时包含硬编码的密钥、许可证、令牌和端点 URL。默认情况下，所有这些都是以明文形式存储在应用程序二进制文件的数据部分。这种技术对这些值进行加密，并在程序中注入代码分支，在程序使用这些数据之前将其解密。

6.10.6.7. 推荐工具

- SwiftShield 可用于执行名称混淆。它读取 Xcode 项目的源代码，并在编译器使用之前将所有类、方法和字段的名称替换为随机值。
- obfuscator-llvm 在中间代码（IR）而不是源代码上操作。它可以用于符号混淆、字符串加密和控制流扁平化。由于它是基于 IR 的，因此与 SwiftShield 相比，它可以明显地隐藏掉更多的应用程序信息。

在此了解更多关于 iOS 混淆技术的信息。

6.10.6.8. 如何使用 SwiftShield

警告：SwiftShield 会不可逆地覆盖你的所有源代码文件。理想情况下，你应该只让它在你的 CI 服务器上运行，并在发布版构建中运行。

SwiftShield 是一个为你的 iOS 项目的对象（包括你的 Pod 和 Storyboard）生成不可逆转的加密名称的工具。这提高了逆向工程师的门槛，在使用 class-dump 和 Frida 等逆向工程工具时，会产生较少的有用输出。

我们用一个 Swift 项目的样本来演示 SwiftShield 的使用。

- 查看 https://github.com/sushi2k/SwiftSecurity。
- 在 Xcode 中打开项目，确保项目构建成功（Product / Build 或 Apple-Key + B）。
- 下载最新版本的 SwiftShield 并解压。

- 进入你下载 SwiftShield 的目录，将 swiftshield 的可执行文件复制到/usr/local/bin：

```
cp swiftshield/swiftshield /usr/local/bin/
```

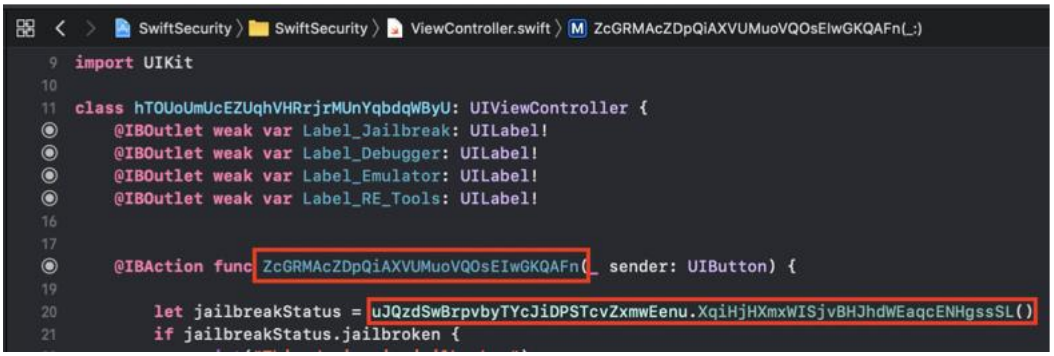
- 在你的终端中，进入 SwiftSecurity 目录（你在步骤 1 中检查出来的），并执行 swiftshield 命令（你在步骤 3 中下载的）。

```
$ cd SwiftSecurity
$ swiftshield -automatic -project-root . -automatic-project-file SwiftSecurit
y.xcodeproj -automatic-project-scheme SwiftSecurity
SwiftShield 3.4.0
Automatic mode
Building project to gather modules and compiler arguments...
-- Indexing ReverseEngineeringToolsChecker.swift --
Found declaration of ReverseEngineeringToolsChecker (s:13SwiftSecurity30Rever
seEngineeringToolsCheckerC)
Found declaration of amIReverseEngineered (s:13SwiftSecurity30ReverseEngineer
ingToolsCheckerC20amIReverseEngineeredSbyFZ)
Found declaration of checkDYLD (s:13SwiftSecurity30ReverseEngineeringToolsChe
ckerC9checkDYLD33_D6FE91E9C9AEC4D13973F8ABFC1AC788LLSbyFZ)
Found declaration of checkExistenceOfSuspiciousFiles (s:13SwiftSecurity30Reve
rseEngineeringToolsCheckerC31checkExistenceOfSuspiciousFiles33_D6FE91E9C9AEC4
D13973F8ABFC1AC788LLSbyFZ)
...
```

SwiftShield 现在正在检测类和方法名称，并将其标识符替换为一个加密值。

在原始源代码中，你可以看到所有的类和方法标识符。

现在，SwiftShield 正在用加密的值替换所有这些值，这些值不会留下任何关于其原始名称或类/方法/Intent 的痕迹。



执行 swiftshield 后，将创建一个名为 swiftshield-output 的新目录。在这个目录中，将创建另一个文件夹，文件夹名称中有一个时间戳。这个目录包含一个名为 conversionMap.txt 的文本文件，该文件将加密的字符串映射到其原始值。

```
$ cat conversionMap.txt
//
// SwiftShield Conversion Map
// Automatic mode for SwiftSecurity, 2020-01-02 13.51.03
// Deobfuscate crash logs (or any text file) by running:
// swiftshield -deobfuscate CRASH_FILE -deobfuscate_map THIS_FILE
//

ViewController ===> hTOUoUmUcEZUqhVHRrjrMUnYqbdqWByU
viewDidLoad ===> DLaNRaFbfmdTDuJCPFXrGhsWhoQyKLnO
sceneDidBecomeActive ===> SUANAnWpkyaIWlGUqwXitCoQSYeVilGe
AppDelegate ===> KftEWsJcctNEmGuvwZGPbusIxEFOVcIb
Deny_Debugger ===> lKEITOpOvLWCFgSCKZdUtpuqiwlvxSjx
Button_Emulator ===> akcVscrZFdBBYqYrcmhhyXAevNdXOKeG
```

这是对加密的崩溃日志进行解密所需要的。

在 SwiftShield 的 Github repo 中还有一个示例项目，可以用来测试 SwiftShield 的执行情况。

6.10.6.9. 静态分析

尝试反汇编 IPA 中的 Mach-O 和 "Frameworks "目录中任何包含的库文件（.dylib 或.framework 文件），并进行静态分析。至少，应用程序的核心功能（即旨在被混淆的功能）不应该被轻易辨别出来。验证一下：

- 有意义的标识符，如类名、方法名和变量名，已经被丢弃了。
- 二进制文件中的字符串资源和字符串是加密的。

- 与受保护功能相关的代码和数据被加密、打包或以其他方式隐藏起来。

为了进行更详细的评估，你需要详细了解相关的威胁和使用的混淆方法。

6.10.7. 设备绑定 (MSTG-RESILIENCE-10)

6.10.7.1. 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 Google 适用于 iOS 的 InstanceID 。详情见“补救”一节。

6.10.7.2. 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 Swift3 中 `UIDevice.current.identifierForVendor?.uuidString`。
- 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`。

6.10.7.3. 动态分析

有几种方法可以测试应用程序绑定。

6.10.7.3.1. 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. 确保您可以提高在应用程序实例中信任级别（例如：在应用程序中进行身份认证）。
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(Documents、Library、tmp)。
        - 复制 Keychain 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. 应用继续在认证状态吗？ 如果是，那么绑定可能无法正常工作。

绑定“可能”不起作用，因为所有的东西在模拟器中并非唯一。

6.10.7.3.2. 使用两个越狱设备进行动态分析

当你想用两台越狱的设备验证应用绑定时，采取以下步骤：

1. 在您的越狱设备上运行应用程序。
2. 确保您可以提高在应用程序实例中信任级别（例如：在应用程序中进行身份认证）。
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。.

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 Keychain 数据。
6. 应用继续在认证状态吗？如果是，那么绑定可能无法正常工作。

6.10.7.4. 补救

在我们描述可用的标识符之前，让我们快速讨论一下如何使用它们进行绑定。在 iOS 中，有三种设备绑定的方法：

- 您可以使用 `[[UIDevice currentDevice] identifierForVendor]` (Objective-C), `UIDevice.current.identifierForVendor?.uuidString` (Swift3),或者 `UIDevice.currentDevice().identifierForVendor?.UUIDString` (in Swift2)。如果你在安装了同一供应商的其他应用程序后重新安装该应用程序，`identifierForVendor` 的值可能不一样，而且当你更新你的应用程序包的名称时它可能会改变。因此，最好将它与 Keychain 中的东西结合起来。
- 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`。
- 您可以使用适用于 iOS 的 Google 及其实例 ID。

任何基于这些方法的方案都会在启用密码和/或 Touch ID 的那一刻变得更加安全，存储在 Keychain 或文件系统中的密钥材料被保护类保护（如 `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` 和 `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`），并且 `SecAccessControlCreateFlags` 被设置为 `kSecAccessControlDevicePasscode` (用于密码), kSecAccessControlUserPresence (密码, Face ID 或 Touch ID), `kSecAccessControlUserPresence` (Fac

e ID 或 Touch ID) 或 `kSecAccessControlBiometryCurrentSet` (Face ID / Touch ID: 但只有当前已注册的生物信息)。

6.10.8. 参考文献

- [#geist] Dana Geist, Marat Nigmatullin. Jailbreak/Root Detection Evasion Study on iOS and Android - https://github.com/crazykid95/Backup-Mobile-Security-Report/blob/master/Jailbreak-Root-Detection-Evasion-Study-on-iOS-and-Android.pdf
- Jan Seredynski. A security review of 1,300 AppStore applications (5 April 2020) - https://seredynski.com/articles/a-security-review-of-1300-appstore-applications.html

6.10.5.1. OWASP MASVS

- MSTG-RESILIENCE-1: "该应用程序检测并回应是否存在已 root 或已越狱的设备，提醒用户或终止该应用程序。"
- MSTG-RESILIENCE-2: "该应用程序可以防止调试和/或检测到调试器的连接，并对其做出反应。必须涵盖所有可用的调试协议。"
- MSTG-RESILIENCE-3: "该应用程序在自己的沙盒中检测并应对篡改可执行文件和关键数据的行为。"
- MSTG-RESILIENCE-4: “该应用程序检测并响应设备上广泛使用的逆向工程工具和框架的存在。”
- MSTG-RESILIENCE-5: “该应用程序可以检测并响应在模拟器中运行的情况。”
- MSTG-RESILIENCE-9: “混淆适用于程序性防御，而程序性防御又阻碍了通过动态分析进行的反混淆。”
- MSTG-RESILIENCE-10: "该应用实现了一个 "设备绑定 "功能，使用从设备特有的多个属性中提取的设备指纹。"
- MSTG-RESILIENCE-11: "所有属于应用程序的可执行文件和库都在文件级别上被加密，并且/或者可执行文件中的重要代码和数据段被加密或打包。琐碎的静态分析不会显示出重要的代码或数据。"

7. 附录

7.1. 测试工具

为了进行安全测试，有不同的工具，以便能够操纵请求和响应，反编译应用程序，调查运行中的应用程序和其他测试案例的行为，并使其自动化。

MASTG 项目对下面的任何工具都没有偏好，也没有推广或销售任何工具。下面的所有工具都经过了验证，如果它们是 " alive "，意味着最近已经推送了更新。尽管如此，并不是所有的工具都被作者使用/测试过，但它们在分析移动应用时可能仍然有用。该列表是按字母顺序排列的。该列表还包含了商业工具。

免责声明：在写作时，我们确保在 MASTG 例子中使用的工具是正常工作的。然而，这些工具可能会被破坏或不能正常工作，这取决于你的主机和测试设备的操作系统版本。工具的运作可能会因为你是否使用了一个被 root/越狱的设备、root/越狱方法的具体版本和/或工具的版本而进一步受到阻碍。MASTG 对工具的工作状态不承担任何责任。如果你发现一个失效的工具或例子，请在工具的原始页面上搜索或提出问题，例如在 GitHub 问题页面。

7.1.1. 适用于所有平台工具

7.1.1.1. Angr

7.1.1.1.1. Angr (Android)

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（"concolic"）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 将尝试进入该状态，使用形式化方法（一种用于静态代码分析的技术）来寻找路径，也可以使用暴力破解。使用 angr 来获得所要求的状态通常比采取手动步骤进行调试和搜索通往所需状态的路径快得多。Angr 在 VEX 中间语言上运行，并带有 ELF/ARM 二进制文件的加载器，所以它非常适合处理原生代码，如本地 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，并有大量的插件。

从第 8 版开始，Angr 基于 Python 3，可以用 pip 安装在*nix 操作系统、macOS 和 Windows 上。

```
pip install angr
```

> angr 的一些依赖项包含 Python 模块 Z3 和 PyVEX 的分支版本，它们会覆盖原始版本。如果你使用这些模块做其他事情，你应该用 Virtualenv 创建一个专门的虚拟环境。或者，你可以随时使用提供的 docker 容器。更多细节见安装指南。

全面的文档，包括安装指南、教程和使用实例，可在 Angr 的 Gitbooks 页面上找到。一个完整的 API 参考资料也是可用的。

你可以从 Python REPL-比如 iPython-中使用 angr，或者用脚本编写你的方法。尽管 angr 的学习曲线有点陡峭，但我们确实推荐在你想用暴力破解达到一个可执行文件的特定状态时使用它。请参阅 "逆向工程和篡改 "一章中的 "符号执行 "部分，可作为一个很好的例子来说明它是如何工作的。

7.1.1.2. Frida

Frida 是由 Ole André Vadla Ravnås 编写的免费开源动态代码工具箱，其工作原理是将 QuickJS JavaScript 引擎（之前的 Duktape 和 V8）注入到被检测的进程中。Frida 可以让你在 Android 和 iOS（以及其他平台）上的原生应用程序中执行 JavaScript 片段。

要在本地安装 Frida，只需运行：

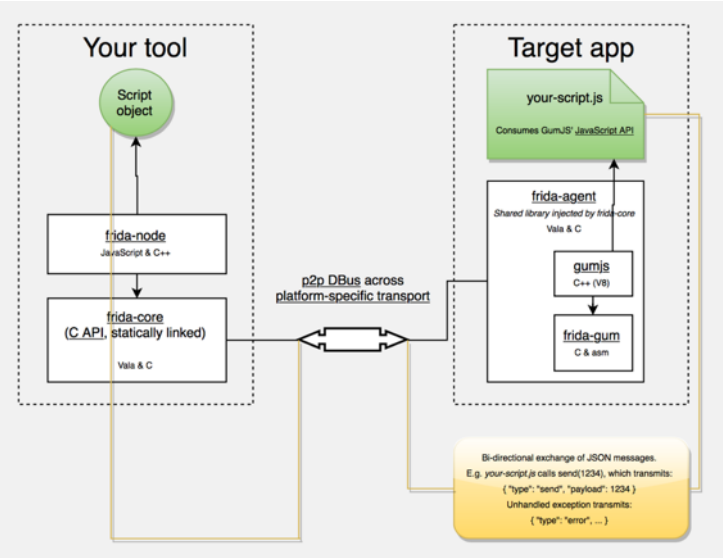
```
pip install frida-tools
```

或者参考安装页面了解更多细节。

代码可以通过几种方式注入。例如，Xposed 永久地修改了 Android 应用加载器，在每次启动新进程时进行劫持来运行你自己的代码。相比之下，Frida 通过直接在进程内存中写入代码来实现代码注入。当连接到一个正在运行的应用程序时：

- Frida 使用 ptrace 来劫持一个正在运行的进程的线程。这个线程被用来分配一大块内存，并将其填充到一个迷你的引导器中。
- 引导器启动一个新的线程，连接到设备上运行的 Frida 调试服务器，并加载一个包含 Frida 代理（`frida-agent.so`）的共享库。
- 该代理建立了一个双向的通信通道，回到工具（如 Frida REPL 或你的自定义 Python 脚本）。

- 被劫持的线程在恢复到原来的状态后恢复，进程的执行也照常进行。



Frida 架构，来源: https://www.frida.re/docs/hacking/

Frida 提供三种操作模式：

1. 注入式：这是最常见的情况，当 frida-server 在 iOS 或 Android 设备中作为守护程序运行时。 frida-core 通过 TCP 暴露，默认监听 localhost:27042。如果在这种模式下运行，需要 root 或越狱的设备。
2. 嵌入：这种情况下，你的设备可以没有被 root 也没有越狱（你不能以非特权用户的身份使用 trace），你负责通过手动或通过第三方工具（如 Objection）将 frida-gadget 库嵌入到你的应用中。
3. 预加载：类似于 `LD_PRELOAD` 或 `DYLD_INSERT_LIBRARIES`。你可以将 frida-gadget 配置为自主运行，并从文件系统中加载一个脚本（例如，与 Gadget 二进制文件所在的相对路径）。

与所选择的模式无关，你可以利用 Frida 的 JavaScript API 来与运行中的进程及其内存进行交互。一些基本的 API 是：

- Interceptor：当使用拦截器 API 时，Frida 在函数序言处注入了一个 trampoline（又称内联劫持），引发了对我们的自定义代码的重定向，执行我们的代码，并返回到原始函数。请注意，虽然对我们的目的非常有效，但这引入了相当大的开销（由于 trampoline 相关的跳转和内容切换），而且不能被认为是透明的，因为它覆盖了原始代码，其行为类似于调试器（设置断点），因此可以以类似的方式检测到，例如，由定期检查自己代码的应用程序。
- Stalker：如果您的跟踪要求包括透明度、性能和高粒度，Stalker 应该是您的 API 选择。当用 Stalker API 追踪代码时，Frida 利用实时动态重新编译（通过使用 Capstone）：当一个线程

要执行下一条指令时，Stalker 分配一些内存，将原始代码复制过来，并将副本与你的自定义代码交错在一起，以便进行插桩。最后，它执行该副本（不触动原始代码，因此避免了任何反调试检查）。这种方法极大地提高了插桩检测的性能，并允许在追踪时有非常高的粒度（例如，只追踪 CALL 或 RET 指令）。你可以在 Frida 的创造者 Ole[#vadla]的博文 "代码追踪器的解剖 "中了解更多深入的细节。Stalker 的一些使用例子是，例如，谁在调用或调用差异。

- Java：当在 Android 上工作时，你可以使用这个 API 来列举加载的类，列举类加载器，创建和使用特定的类实例，通过扫描堆来列举类的实时实例，等等。
- ObjC：当在 iOS 上工作时，你可以使用这个 API 来获得所有注册类的映射，注册或使用特定的类或协议实例，通过扫描堆来列举类的实时实例等。

Frida 还提供了一些建立在 Frida API 之上的简单工具，在通过 pip 安装 frida-tools 之后，可以直接从终端获得。比如说：

- 你可以使用 Frida CLI(`frida`)来快速制作脚本原型和试错的场景。
- `frida-ps` 获取设备上运行的所有应用程序（或进程）的列表，包括它们的名称、标识符和 PID。
- `frida-ls-devices` 列出运行 Frida 服务器或代理的连接设备。
- `frida-trace` 可以快速追踪 iOS 应用的一部分或在 Android 原生库中实现的方法。

此外，你还会发现一些基于 Frida 的开源工具，如：

- Passionfruit：一个 iOS 应用黑盒评估工具。
- Fridump：一个用于 Android 和 iOS 的内存转储工具。
- Objection: 一个运行时移动安全评估框架。
- r2frida: 一个将 radare2 的强大逆向工程能力与 Frida 的动态工具箱合并的项目。
- jnitrace：一个用于追踪本地库对 Android JNI 运行时方法使用情况的工具。

我们将在整个指南中使用所有这些工具。

你可以按原样使用这些工具，根据你的需要进行调整，或者把它们作为使用 API 的优秀范例。当你写你自己的劫持脚本时，或者当你建立自省工具来支持你的逆向工程工作流程时，有它们作为例子是非常有帮助的。

7.1.1.2.1. 适用于 Android 的 Frida

Frida 支持通过 Java API 与 Android Java 运行时进行交互。你可以在进程和它的原生库中劫持并调用 Java 和原生函数。你的 JavaScript 片段可以完全访问内存，例如，读取和/或写入任何结构化数据。

下面是 Frida API 提供的一些功能，在 Android 上是平台相关的或独有的。

- 实例化 Java 对象并调用静态和非静态类方法（Java API）。
- 替换 Java 方法的实现（Java API）。
- 通过扫描 Java 堆来列举特定类的实时实例（Java API）。
- 扫描进程内存中出现的字符串（Memory API）。
- 拦截原生函数调用，在函数进入和退出时运行你自己的代码（Interceptor API）。

请记住，在 Android 上，你也可以从安装 Frida 时提供的内置工具中获益，这包括 Frida CLI（`frida`）、`frida-ps`、`frida-ls-devices` 和 `frida-trace`，这只是其中的一些例子。

Frida 经常被拿来与 Xposed 进行比较，然而这种比较是远远不够的，因为这两个框架在设计上都有不同的目标。作为一个应用安全测试人员，了解这一点很重要，这样你就可以知道在什么情况下使用哪个框架。

- Frida 是独立的，你所需要的只是在你的目标 Android 设备的已知位置运行 frida-server 二进制文件（见下面的 "安装 Frida"）。这意味着，与 Xposed 相比，它并没有深入安装在目标操作系统中。
- 逆向一个应用程序是一个反复的过程。作为靠前的结果，你在测试时获得一个更短的反馈循环，因为你不需要（软）重启来应用或简单地更新你的劫持。因此，在实现更多永久性劫持时，你可能更喜欢使用 Xposed。
- 你可以在进程运行期间的任何时候注入和更新你的 Frida JavaScript 代码（类似于 iOS 的 Cycript）。这样你就可以通过让 Frida 生成你的应用程序来执行所谓的早期插桩，或者你可能更喜欢附加到一个正在运行的应用程序，你可能已经把它带到了某个状态。
- Frida 能够处理 Java 和原生代码（JNI），允许你修改它们。不幸的是，这是 Xposed 的一个限制，它缺乏对原生代码的支持。

请注意，截至 2019 年初，Xposed 还不能在 Android 9（API 级别 28）上工作。

7.1.1.2.1.1 在 Android 上安装 Frida

为了在你的 Android 设备上安装 Frida：

- 如果你的设备没有被 root，你也可以使用 Frida，请参考 "逆向工程和篡改 "一章的 "非 root 设备上的动态分析 "一节。
- 如果你有一个已 root 的设备，只需按照官方说明或按照下面的提示操作即可。

除非另有说明，否则我们在此假定设备已经 root。从 Frida 发布页面下载 frida-server 二进制文件。确保你下载的 frida-server 二进制文件与你的 Android 设备或模拟器的架构相符：x86、x86_64、arm 或 arm64。确保服务器的版本（至少是主要的版本号）与你本地安装的 Frida 的版本一致。PyPI 通常会安装 Frida 的最新版本。如果你不确定安装的是哪个版本，你可以用 Frida 命令行工具检查。

```
frida --version
```

或者你可以运行以下命令来自动检测 Frida 的版本并下载正确的 frida-server 二进制文件。

```
wget https://github.com/frida/frida/releases/download/$(frida --version)/frid
a-server-$(frida --version)-android-arm.xz
```

将 frida-server 复制到设备上并运行它。

```
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 &"
```

7.1.1.2.1.2 在 Android 上使用 Frida

在 frida-server 运行的情况下，你现在应该可以通过以下命令获得正在运行的进程列表（使用-U 选项来指示 Frida 使用连接的 USB 设备或仿真器）。

```
$ 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
(...)
```

或者用-Uai 参数组合来限制列表，以获得连接的 USB 设备（-U）上当前安装的所有（-a）应用程序（-i）。

```
$ frida-ps -Uai
  PID  Name                                    Identifier
-----  ---------------------------------------  --------------------------
--
  766  Android System                           android
30692  Chrome                                   com.android.chrome
 3520  Contacts Storage                         com.android.providers.contac
ts
    -  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，如果它们目前正在运行，它还将显示它们的 PID。在列表中搜索你的应用程序，记下 PID 或其名称/标识符。从现在开始，你将通过使用其中之一来参考你的应用程序。建议使用标识符，因为 PID 会在应用的每次运行中改变。例如，我们以 `com.android.chrome` 为例。你现在可以在所有的 Frida 工具上使用这个字符串，例如在 Frida CLI、frida-trace 或 Python 脚本上。

7.1.1.2.1.3 使用 frida-trace 跟踪原生库

要跟踪特定的（低级）库调用，你可以使用 `frida-trace` 命令行工具。

```
frida-trace -U com.android.chrome -i "open"
```

这在`__handlers__/libc.so/open.js` 中生成了一个小的 JavaScript，Frida 将其注入进程中。该脚本追踪所有对 libc.so 中 open 函数的调用。你可以根据你的需要用 Frida 的 JavaScript API 修改生成的脚本。

不幸的是，目前还不支持追踪 Java 类的高级方法（但将来可能会支持）。

7.1.1.2.1.4 Frida CLI 和 Java API

使用 Frida CLI 工具（`frida`）来与 Frida 互动工作。它可以劫持一个进程，为你提供一个命令行接口来访问 Frida 的 API。

```
frida -U com.android.chrome
```

使用-l 选项，你也可以使用 Frida CLI 来加载脚本，例如，加载 `myscript.js`：

```
frida -U -l myscript.js com.android.chrome
```

Frida 还提供了一个 Java API，这对处理 Android 应用程序特别有帮助。它可以让你直接与 Java 类和对象一起工作。这里有一个脚本，可以覆盖一个活动类的 `onResume` 函数：

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        console.log("[*] onResume() got called!");
        this.onResume();
    };
});
```

上述脚本调用 `Java.perform` 以确保你的代码在 Java VM 的内容中被执行。它通过 `Java.use` 为 `android.app.Activity` 类实例化了一个包装器，并重写了 `onResume` 函数。新的 `onResume` 函数实现会向控制台打印一个通知，并在应用程序中每次恢复活动时通过调用 `this.onResume` 调用原来的 `onResume` 方法。

Frida 还可以让你搜索和处理堆上的实例化对象。下面的脚本搜索 `android.view.View` 对象的实例并调用其 `toString` 方法。其结果被打印到控制台。

```
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")
            }
        });
    });
});
```

输出结果会是这样的：

```
[*] Starting script
[*] Instance found: android.view.View{7ccea78 G.ED..... ......ID 0,0-0,0 #7f0
c01fc app:id/action_bar_black_background}
[*] Instance found: android.view.View{2809551 V.ED..... ........ 0,1731-0,173
1 #7f0c01ff app:id/menu_anchor_stub}
[*] Instance found: android.view.View{be471b6 G.ED..... ......I. 0,0-0,0 #7f0
c01f5 app:id/location_bar_verbose_status_separator}
[*] Instance found: android.view.View{3ae0eb7 V.ED..... ........ 0,0-1080,63
#102002f android:id/statusBarBackground}
[*] Finished heap search
```

你也可以使用 Java 的反射功能。要列出 `android.view.View` 类的公共方法，你可以在 Frida 中为这个类创建一个包装器，然后从包装器的 class 属性中调用 `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很长的方法列表。

```
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()
...
```

7.1.1.2.2. 适用于 iOS 的 Frida

Frida 支持通过 ObjC API 与 Objective-C 运行时的交互。你将能够在进程和它的原生库内劫持并调用 Objective-C 和原生函数。你的 JavaScript 片段可以完全访问内存，例如，读取和/或写入任何结构化数据。

下面是 Frida API 提供的一些功能，在 iOS 上是平台相关的或独有的。

- 实例化 Objective-C 对象并调用静态和非静态类方法（ObjC API）。
- 追踪 Objective-C 方法调用和/或替换它们的实现（Interceptor API）。

- 通过扫描堆列举特定类的实时实例（ObjC API）。
- 扫描进程内存中出现的字符串（Memory API）。
- 拦截原生函数调用，在函数进入和退出时运行你自己的代码（Interceptor API）。

请记住，在 iOS 上，你也可以从安装 Frida 时提供的内置工具中获益，其中包括例如 Frida CLI（`frida`）、`frida-ps`、`frida-ls-devices` 和 `frida-trace`。

在 iOS 上有一个 `frida-trace` 的功能值得强调：使用-m 参数和通配符追踪 Objective-C API。例如，追踪所有名称中包含 "HTTP "并且属于任何名称以 "NSURL "开头的类的方法，只需简单运行下面的命令：

```
frida-trace -U YourApp -m "*[NSURL* *HTTP*]"
```

为了快速入门，你可以研究 iOS 的示例。

7.1.1.2.2.1 在 iOS 上安装 Frida

要将 Frida 连接到一个 iOS 应用程序，你需要一种方法将 Frida 运行时注入到该应用程序。这在已越狱的设备上很容易做到：只要通过 Cydia 安装 `frida-server`。一旦安装完毕，Frida 服务器将自动以 root 权限运行，允许你轻松地将代码注入到任何进程。

启动 Cydia，添加 Frida 源，通过浏览到 **Manage 管理**-> **Sources 来源**-> **Edit 编辑**-> **Add 添加**并输入 https://build.frida.re,。然后你应该能够可以找到并安装 Frida 软件包。

7.1.1.2.2.2 在 iOS 上使用 Frida

通过 USB 连接你的设备，并通过运行 `frida-ps` 命令和参数'-U'确保 Frida 工作。这应该会返回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
(...)
```

7.1.1.2.3. Frida 绑定

为了扩展脚本体验，Frida 提供了与 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` 开始你的 Python 脚本，就可以了。请看下面的脚本，它简单地运行了前面的 JavaScript 片段。

```
# 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()
```

在这种情况下，运行 Python 脚本（python3 frida_python.py）的结果与前面的例子相同：它将把 `android.view.View` 类的所有方法打印到终端。然而，你可能想用 Python 来处理这些数据。使用 send 而不是 console.log 将把 JSON 格式的数据从 JavaScript 发送到 Python。请阅读下面例子中的注释。

```
# python3 frida_python_send.py
import frida

session = frida.get_usb_device().attach('com.android.chrome')

# 1. 我们把方法名存在一个 list 里
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. 这是一个回调函数，只有包含 "Text "的方法名称会被追加到列表中。
def on_message(message, data):
    if "Text" in message['payload']:
        android_view_methods.append(message['payload'])

# 3. 我们告诉脚本在每次收到消息时都要运行我们的回调。
script.on('message', on_message)

script.load()

# 4. 我们对收集到的数据做一些处理，在这种情况下，我们只是打印它
for method in android_view_methods:
    print(method)

session.detach()
```

这就有效地过滤了这些方法，只打印出包含 "Text "字符串的方法。

```
$ 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，由你决定你想在哪里处理数据。有时从 JavaScript 处理会更方便，在其他情况下，Python 将是最好的选择。当然，你也可以通过使用 `script.post` 从 Python 向 JavaScript 发送消息。关于发送和接收消息的更多信息，请参考 Frida 的文档。

7.1.1.3. Frida CodeShare

Frida CodeShare 是一个源，包含了一系列可以运行的 Frida 脚本，这些脚本在执行 Android 和 iOS 上的具体任务时都有很大的帮助，也可以作为建立自己脚本的灵感。两个有代表性的例子是。

- 通用 Android SSL 固定绕过 - https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/
- ObjC 方法观察器- https://codeshare.frida.re/@mrmacete/objc-method-observer/

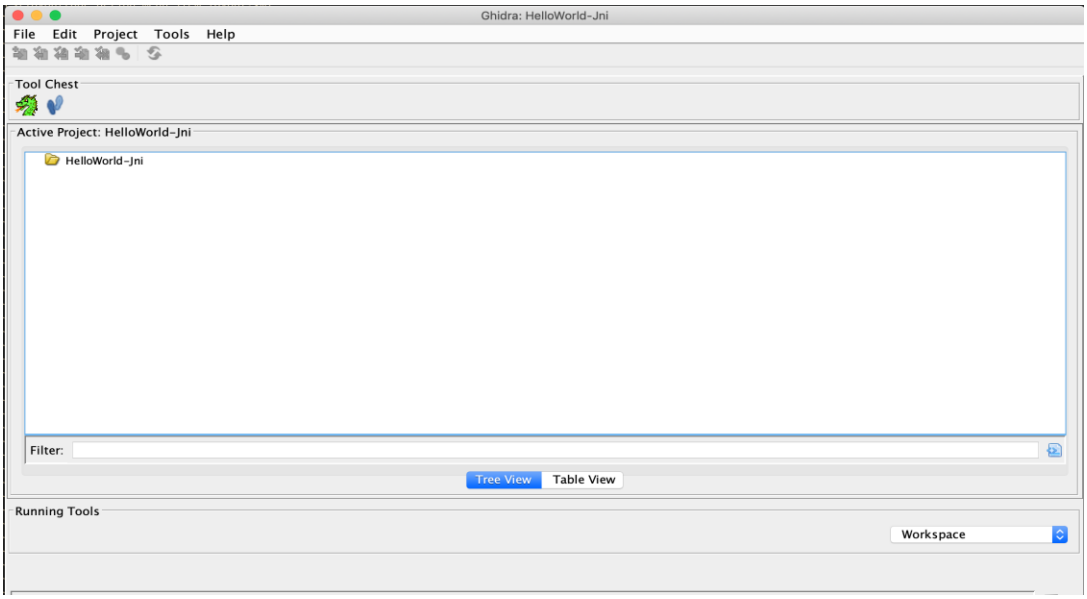
使用它们就像在使用 Frida CLI 时加入--codehare <handler>标志和一个处理程序一样简单。例如，要使用 "ObjC 方法观察器"，请输入以下内容。

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

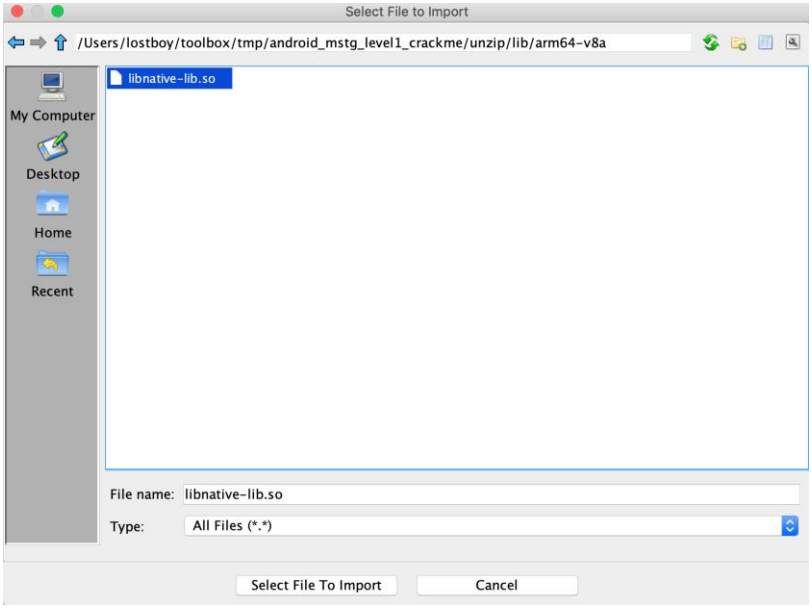
7.1.1.4. Ghidra

Ghidra 是一个开源的软件逆向工程（SRE）工具套件，由美国国家安全局（NSA）的研究局开发。Ghidra 是一个多功能的工具，包括一个反汇编器、反编译器和一个用于高级使用的内置脚本引擎。关于如何安装它，请参考安装指南，也请看备忘录，以初步了解可用的命令和快捷方式。在本节中，我们将介绍如何创建一个项目，查看二进制的反汇编和反编译代码。

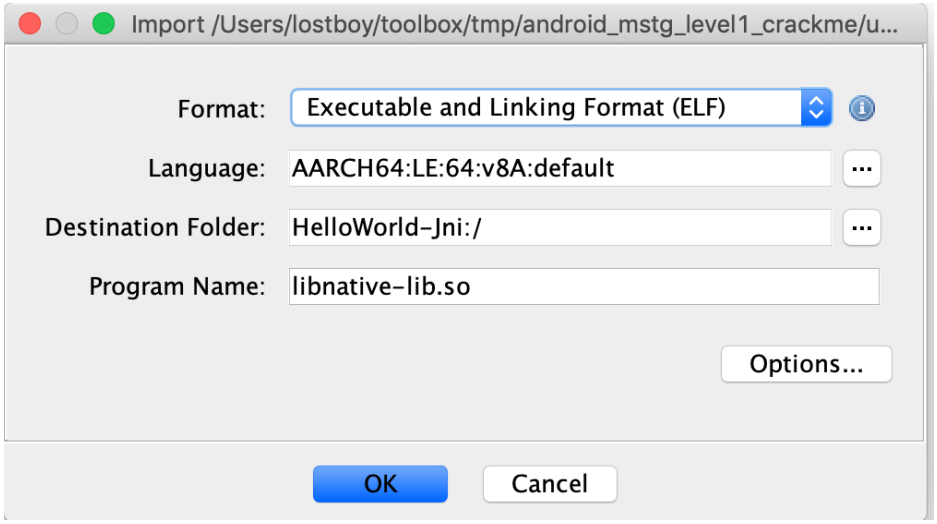
使用 `ghidraRun`（*nix）或 ghidraRun.bat（Windows）启动 Ghidra，这取决于你所用的平台。一旦 Ghidra 被启动，通过指定项目目录创建一个新的项目。迎接你的将是一个如下所示的窗口。



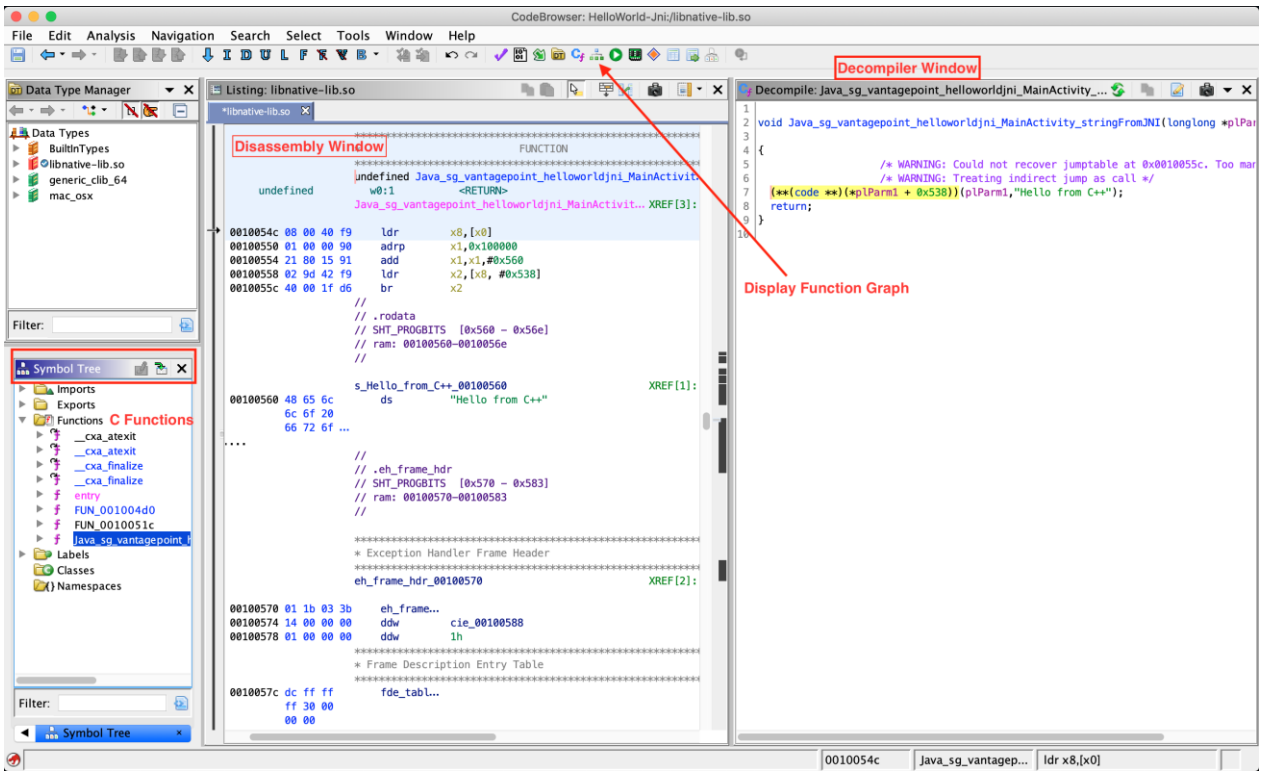
在你新的活动项目 **Active Project** 中，你可以通过进入文件 **File**->导入文件 **Import File** 并选择所需的文件来导入一个应用程序二进制文件。



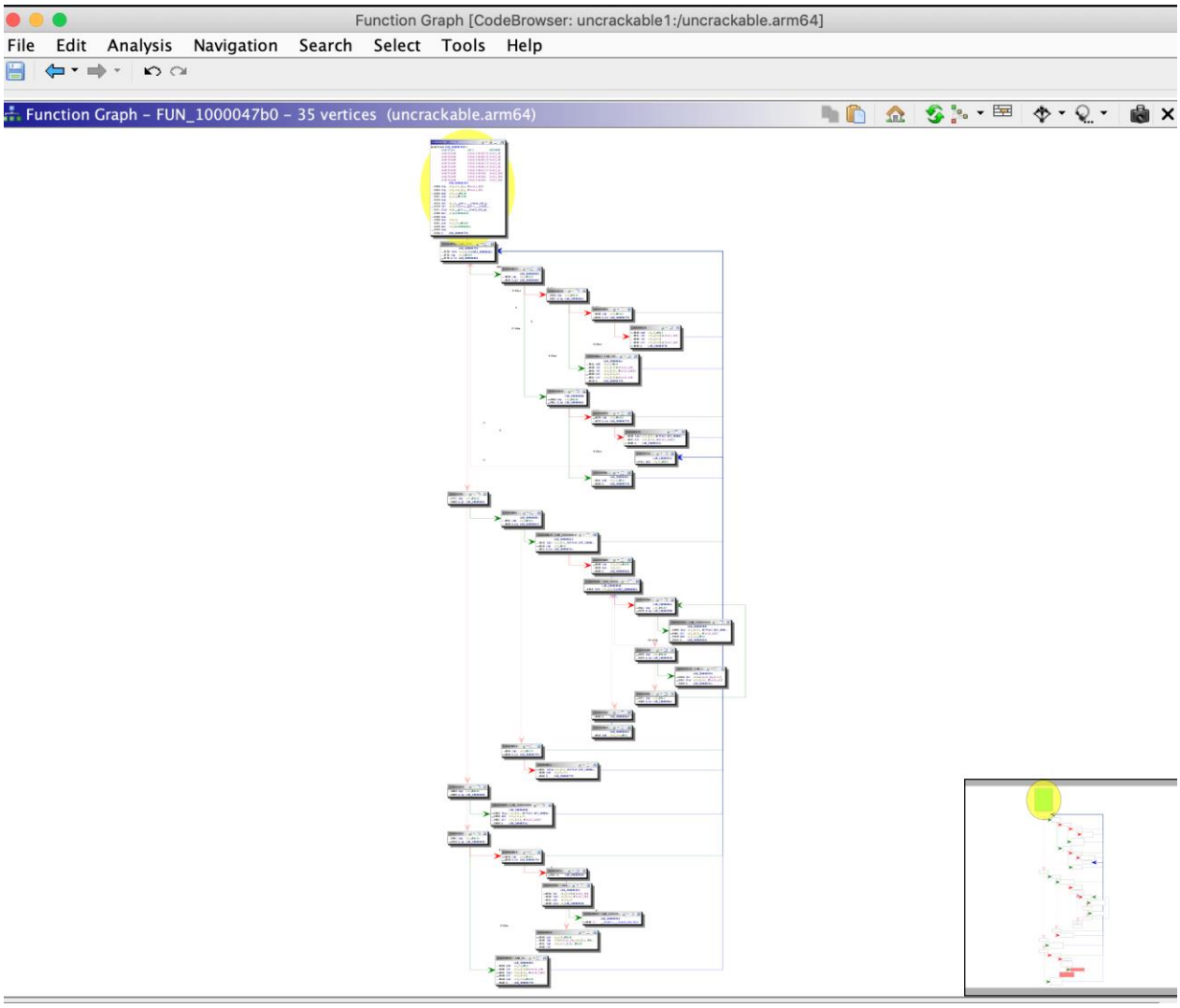
如果文件能够被正确处理，Ghidra 将在开始分析之前显示二进制文件的元信息。



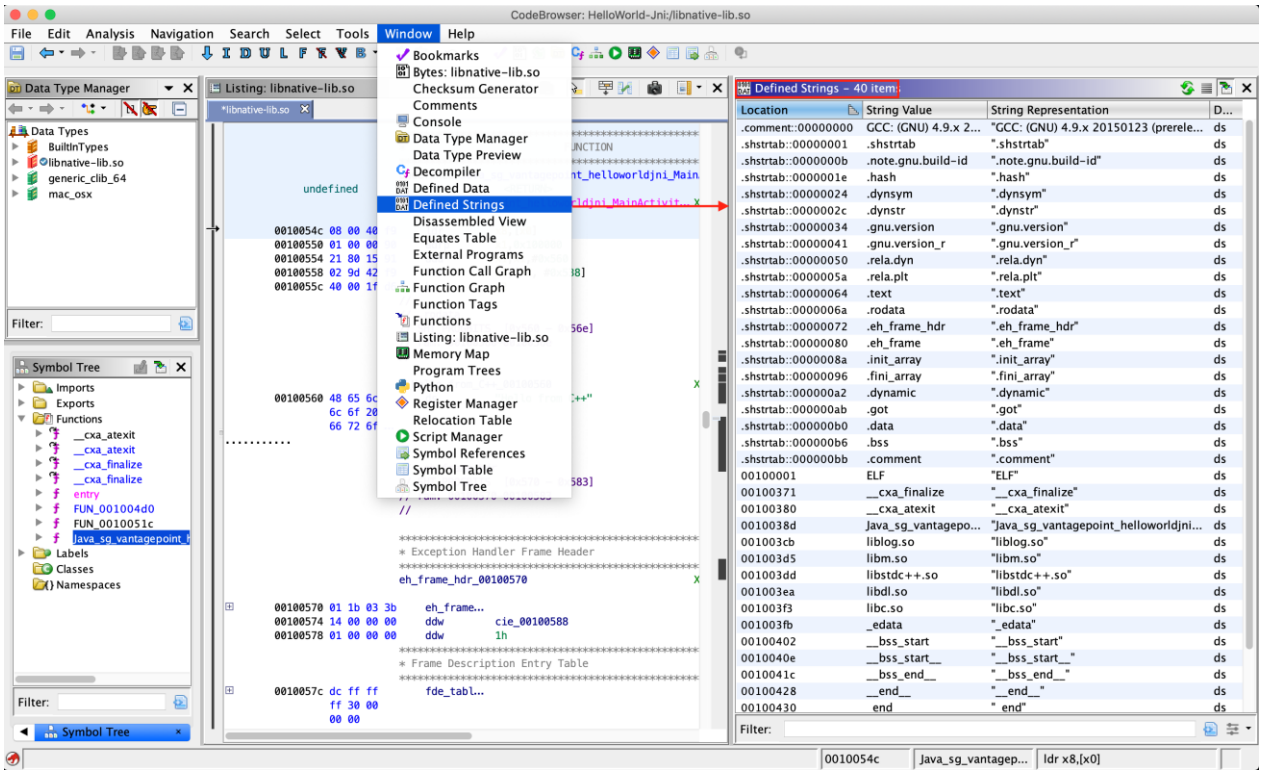
为了得到上面选择的二进制文件的反汇编代码，从活动项目 **Active Project** 窗口双击导入的文件。点击 "是 **yes**"和“分析 **analyze**”，在随后的窗口中进行自动分析。自动分析将需要一些时间，取决于二进制文件的大小，可以在代码浏览器窗口的右下角跟踪进度。一旦自动分析完成，你可以开始探索二进制。



在 Ghidra 中探索二进制文件最重要的窗口是列表 **Listing**（反汇编）窗口，符号树 **Symbol Tree** 窗口和反编译器 **Decompiler** 窗口，它显示所选函数反汇编的反编译版本。显示函数图 **Display Function Graph** 选项显示所选函数的控制流图。



Ghidra 还有许多其他的功能，大部分都可以通过打开 **Window** 菜单进行探索。例如，如果你想检查二进制文件中的字符串，可以打开 "定义字符串 **Defined Strings** "选项。我们将在接下来的章节中讨论其他高级功能，同时分析 Android 和 iOS 平台的各种二进制文件。



7.1.1.5. Hopper (商业工具)

一个用于 macOS 和 Linux 的逆向工程工具，用于反汇编、反编译和调试 32/64bits 的 Intel Mac、Linux、Windows 和 iOS 可执行文件 - https://www.hopperapp.com/

7.1.1.6. IDA Pro (商业工具)

一个 Windows、Linux 或 macOS 平台的多处理器反汇编器和调试器 - https://www.hex-rays.com/products/ida/index.shtml

7.1.1.7. LIEF

LIEF 的目的是提供一个跨平台库来解析、修改和抽象 ELF、PE 和 MachO 格式。例如，你可以将某个库作为本地库的一个依赖项注入，而应用程序已经默认加载了这个库。- https://lief.quarkslab.com/

7.1.1.8. MobSF

MobSF（移动安全框架）是一个自动化的、多合一的移动应用测试框架，能够进行静态和动态分析。启动 MobSF 的最简单方法是通过 Docker。

```
docker pull opensecurity/mobile-security-framework-mobsf
docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf:late
st
```

或者在你的主机上安装并启动它，运行：

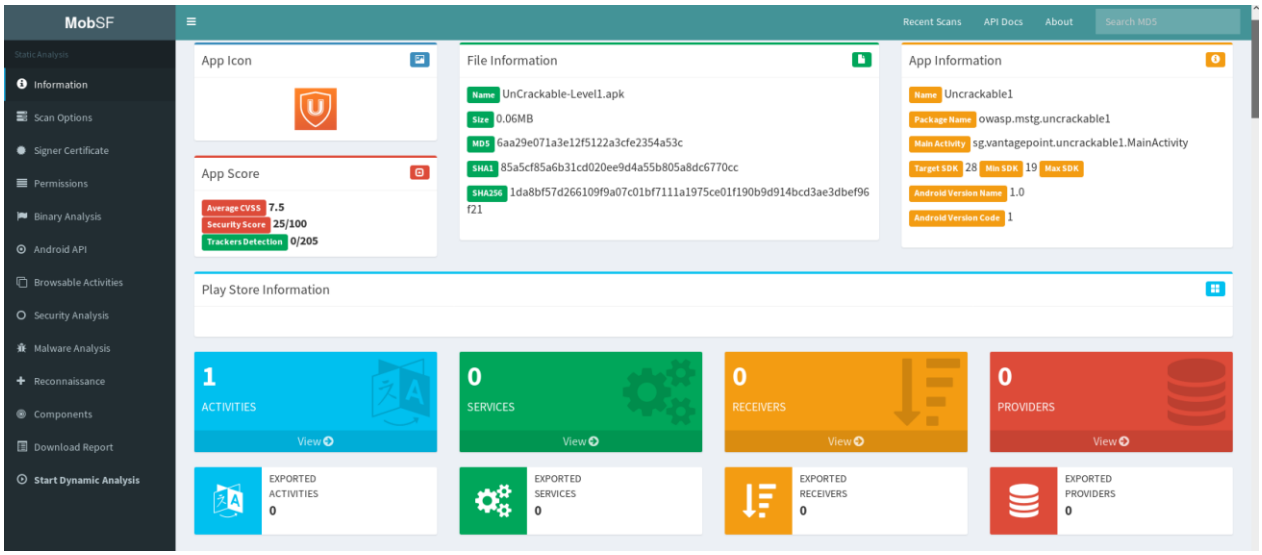
```
# Setup
git clone https://github.com/MobSF/Mobile-Security-Framework-MobSF.git
cd Mobile-Security-Framework-MobSF
./setup.sh # For Linux and Mac
setup.bat # For Windows

# Installation process
./run.sh # For Linux and Mac
run.bat # For Windows
```

一旦你启动并运行 MobSF，你可以通过浏览到 http://127.0.0.1:8000，在你的浏览器中打开它。只需将你要分析的 APK 拖到上传区，MobSF 就会开始工作。

7.1.1.8.1. 适用于 Android 的 MobSF

在 MobSF 完成分析后，您将收到一页关于所有执行的测试的概述。该页面被分成多个部分，对应用程序的攻击面给出了一些初步提示。



显示如下:

- 关于该应用程序及其二进制文件的基本信息。
- 一些选项包括。
  - 查看 AndroidManifest.xml 文件。
  - 查看应用程序的 IPC 组件。
- 签名者证书。
- 应用程序的权限。
- 安全分析显示已知缺陷，例如，如果应用程序的备份被启用。
- 应用程序二进制使用的库列表和解压后的 APK 内的所有文件列表。
- 检查恶意的 URL 的恶意软件分析。

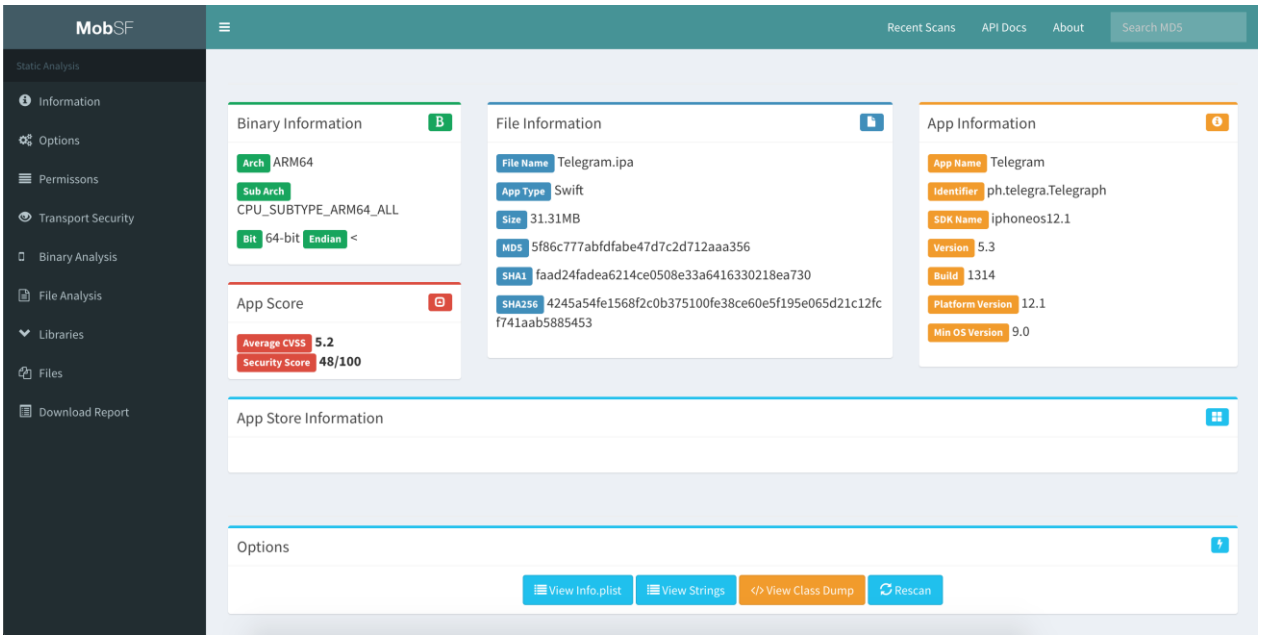
更多细节请参考 MobSF 文档。

7.1.1.8.2. 适用于 iOS 的 MobSF

在 macOS 主机上本地运行 MobSF，你会从一个稍好的类转储输出中受益。

一旦你启动并运行 MobSF，你可以在你的浏览器中通过浏览 http://127.0.0.1:8000 来打开它。只需将你要分析的 IPA 拖到上传区域，MobSF 就会开始工作。

在 MobSF 完成分析后，您将收到一页关于所有执行的测试的概述。该页面被分成多个部分，对应用程序的攻击面给出了一些初步提示。



显示如下:

- 关于该应用程序及其二进制文件的基本信息。
- 一些选项包括。
  - 查看 `Info.plist` 文件。
  - 查看应用程序二进制文件中包含的字符串。
  - 下载 class-dump，如果应用程序是用 Objective-C 编写的；如果它是用 Swift 编写的，则不能创建 class-dump。
- 列出从 `Info.plist` 中提取的所有目的字符串，这些字符串对应用程序的权限有一些提示。
- 应用程序传输安全（ATS）配置中的异常情况将被列出。
- 简要的二进制分析，显示是否激活了免费的二进制安全功能，或者例如二进制是否使用了被禁止的 API。
- 应用程序二进制所使用的库列表和解压后的 IPA 内的所有文件列表。

与 Android 用例相比，MobSF 没有为 iOS 应用提供任何动态分析功能。

更多细节请参考 MobSF 文档。

7.1.1.9. nm

nm 是一个显示给定二进制文件名称列表（符号表）的工具。你可以找到更多关于 Android（GNU）版本和 iOS 版本的信息。

7.1.1.10. Objection

Objection 是一个 "运行时移动探索工具包，由 Frida 驱动"。它的主要目标是通过一个直观的界面允许在非 root 设备上进行安全测试。

Objection 通过为你提供工具来实现这一目标，通过重新打包，轻松地将 Frida 小工具注入到应用程序中。这样，你就可以通过侧载将重新打包的应用程序部署到非 root/未越狱设备上。Objection 还提供了一个 REPL，允许你与应用程序进行交互，让你有能力执行应用程序可以执行的任何动作。

Objection 可以通过 pip 安装，如 Objection 的 Wiki 上所述。

```
pip3 install objection
```

7.1.1.10.1. 适用于 Android 的 Objection

Objection 提供了几个专门针对 Android 的功能。Objection 的完整功能列表可以在项目主页上找到，但这里有几个有趣的功能：

- 重新打包应用程序以包括 Frida 小工具
- 禁用常见的 SSL 固定方法
- 访问应用程序的存储空间以下载或上传文件
- 执行自定义的 Frida 脚本
- 列出 Activity、服务和广播接收器
- 启动 Activity

如果你有一个安装了 frida-server 的 root 设备，Objection 可以直接连接到运行中的 Frida 服务器，提供其所有的功能，而不需要重新包装应用程序。然而，并不是所有的 Android 设备都可以 root，或者应用程序可能包含先进的 RASP 控制，用于 root 检测，所以注入 frida-gadget 可能是绕过这些控制的最简单方法。

在**未 root 的设备上进行高级动态分析的能力**是使 Objection 难以置信的功能之一。在遵循重新包装的过程后，你将能够运行所有上述的命令，这使得快速分析一个应用程序或绕过基本的安全控制变得非常容易。

7.1.1.10.1.1 在 Android 上使用 Objection

要启动 Objection 取决于你是否给 APK 打了补丁，或者你是否使用运行 Frida-server 的 root 设备。要运行打了补丁的 APK，objection 会自动找到任何连接的设备，并搜索正在监听的 Frida 小工具。然而，当使用 frida-server 时，你需要明确告诉 frida-server 你想分析哪个应用程序。

```
# 连接到一个修补过的 APK
objection explore

# 使用 frida-ps 找到正确的名称
$ frida-ps -Ua | grep -i telegram
30268  Telegram                                 org.telegram.messenger

# 使用 Frida-server 连接到 Telegram 应用程序
$ objection --gadget="org.telegram.messenger" explore
```

一旦你进入 Objection REPL，你可以执行任何可用的命令。下面是一些最有用的命令的概述。

```
# 显示属于应用程序的不同存储位置
$ env

# 禁用常见的 ssl 固定方法
$ android sslpinning disable

# 列出 keystore 中的项目
$ android keystore list

# 尝试绕过 root 检测
$ android root disable
```

关于使用 Objection REPL 的更多信息，可以在 Objection Wiki 上找到。

7.1.1.10.2. 适用于 iOS 的 Objection

Objection 提供了几个专门针对 iOS 的功能。Objection 的完整功能列表可以在项目的主页上找到，但这里有几个有趣的功能：

- 重新包装应用程序以包括 Frida 小工具
- 禁用常见的 SSL 固定方法
- 访问应用程序的存储空间以下载或上传文件
- 执行自定义的 Frida 脚本
- 转储 Keychain
- 读取 plist 文件

所有这些任务以及更多的任务都可以通过使用 object 的 REPL 中的命令来轻松完成。例如，你可以通过运行以下命令获得一个应用程序中使用的类、类的功能或关于一个应用程序的包信息。

```
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 <C
lassName>
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # ios bundles list_bundles
```

如果你有一个安装了 frida-server 的越狱设备，Objection 可以直接连接到运行中的 Frida 服务器，提供其所有功能，而不需要重新打包应用程序。然而，并不总是能够越狱最新版本的 iOS，或者你可能有一个具有高级越狱检测机制的应用程序。

在非越狱设备上进行高级动态分析的能力是使 Objection 难以置信的功能之一。在遵循重新打包的过程后，你将能够运行所有上述的命令，这使得快速分析一个应用程序非常容易，或绕过基本的安全控制。

7.1.1.10.2.1 在 iOS 上使用 Objection

要启动 Objection 取决于你是否给 IPA 打了补丁，或者你是否在使用运行 Frida-server 的越狱设备。要运行打了补丁的 IPA，objection 会自动找到任何连接的设备，并搜索正在监听的 Frida 小工具。然而，当使用 frida-server 时，你需要明确告诉 frida-server 你想分析哪个应用程序。

```
# 连接到一个修补过的 IPA
$ objection explore

# 使用 frida-ps 来获得正确的应用程序名称
$ frida-ps -Ua | grep -i Telegram
983  Telegram

# 通过 Frida-server 连接到 Telegram 应用程序
$ objection --gadget="Telegram" explore
```

一旦你进入 Objection REPL，你可以执行任何可用的命令。下面是一些最有用的命令的概述。

```
# 显示属于应用程序的不同存储位置
$ env

# 禁用常见的 ssl 固定方法
$ ios sslpinning disable

# 转储 Keychain
$ ios keychain dump

# 转储 Keychain，包括访问修饰符。结果会写入主机上的 myfile.json
$ ios keychain dump --json <myfile.json>

# 显示一个 plist 文件的内容
$ ios plist cat <myfile.plist>
```

关于使用 Objection REPL 的更多信息，可以在 Objection Wiki 上找到。

7.1.1.11. r2frida

r2frida 是一个允许 radare2 连接到 Frida 的项目，有效地合并了 radare2 强大的逆向工程能力和 Frida 的动态插桩工具箱。r2frida 可以在 Android 和 iOS 上使用，允许你：

- 通过 USB 或 TCP 将 radare2 连接到任何本地进程或远程 frida-server。
- 从目标进程读取/写入内存。
- 将 Frida 的信息，如地图、符号、导入、类和方法加载到 radare2。
- 从 Frida 调用 r2 命令，因为它将 r2pipe 接口暴露在 Frida Javascript API 中。

请参考 r2frida 的官方安装说明。

在 frida-server 运行后，你现在应该能够使用 pid、spawn path、host 和 port 或 device-id 来附加到它。例如，要附加到 PID 1234。

```
r2 frida://1234
```

关于如何连接 frida-server 的更多例子，请看 r2frida 的 README 页面中的使用部分。

下面的例子是使用 Android 应用程序执行的，但也适用于 iOS 应用程序。

一旦进入 r2frida 会话，所有的命令都以\或=！开头。例如，在 radare2 中你会运行 i 来显示二进制信息，但在 r2frida 中你会使用 \i。

使用 `r2 frida://?`显示所有选项

```
[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
```

要在内存中搜索一个特定的关键词，你可以使用搜索命令\/：

```
[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 [0xfffffffffff600000-0xfffffffffff601000]
hits: 23
0x561f072d89ee hit12_0 unacceptable policyunsupported md algorithmvar bad val
uec
0x561f0732a91a hit12_1 unacceptableSearching 12 bytes: 75 6e 61 63 63 65 70 7
4 61
```

为了以 JSON 格式输出搜索结果，我们只需在之前的搜索命令中加入 j（就像在 r2 shell 中那样）。这可以在大多数命令中使用。

```
[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 [0xfffffffffff600000-0xfffffffffff601000]
hits: 23
{"address":"0x561f072c4223","size":12,"flag":"hit14_1","content":"unacceptabl
e \
policyunsupported md algorithmvar bad valuec0"},{"address":"0x561f072c4275",
\
```

```
"size":12,"flag":"hit14_2","content":"unacceptableSearching 12 bytes: 75 6e 6
1 \
63 63 65 70 74 61"},{"address":"0x561f072c42c8","size":12,"flag":"hit14_3", \
"content":"unacceptableSearching 12 bytes: 75 6e 61 63 63 65 70 74 61 "},
...
```

要列出已加载的库，请使用命令\il，并使用命令~从 radare2 的内部过滤结果。例如，下面的命令将列出与关键词 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
```

同样，要列出导出，并通过特定的关键词过滤结果：

```
[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
```

要列出或设置一个断点，请使用 db 命令。这在分析/修改内存时很有用。

```
[0x00000000]> \db
```

最后，请记住，你也可以用\.加上脚本的名称来运行 Frida 的 JavaScript 代码，。

```
[0x00000000]> \. agent.js
```

你可以在他们的 Wiki 项目中找到更多关于如何使用 r2frida 的例子。

7.1.1.12. radare2

7.1.1.12.1. radare2 (Android)

radare2（r2）是一个流行的开源逆向工程框架，用于反汇编、调试、修补和分析二进制文件，可编写脚本，支持许多架构和文件格式，包括 Android 和 iOS 应用程序。对于 Android，支持

Dalvik DEX（odex、multidex）、ELF（可执行文件、.so、ART）和 Java（JNI 和 Java 类）。它还包含几个有用的脚本，可以在移动应用分析过程中帮助你，因为它提供了低级别的反汇编和安全的静态分析，当传统工具失败时，它就会派上用场。

radare2 实现了一个丰富的命令行界面（CLI），你可以在那里执行上述任务。然而，如果你对使用 CLI 进行逆向工程不是很舒服，你可能想考虑使用 Web UI（通过-H 参数）或更方便的 Qt 和 C++ GUI 版本，即 Cutter。请记住，CLI，以及更具体的可视化界面和脚本功能（r2pipe），是 radare2 的核心功能，绝对值得学习如何使用它。

7.1.1.12.1.1 安装 radare2

请参考 radare2 的官方安装说明。我们强烈建议总是从 GitHub 版本安装 radare2，而不是通过 APT 等普通软件包管理器。Radare2 正处于非常活跃的开发阶段，这意味着第三方软件库经常会过期。

7.1.1.12.1.2 使用 radare2

radare2 框架包括一组小工具，可以从 r2 shell 中使用，也可以作为 CLI 工具独立使用。这些工具包括 `rabin2`, `rasm2`, `rahash2`, `radiff2`, `rafind2`, `ragg2`, `rarun2`, `rax2` 当然还有 `r2`，这是主要的一个。

例如，你可以使用 `rafind2` 直接从编码的 Android Manifest（AndroidManifest.xml）中读取字符串。

```
# 权限
$ rafind2 -ZS permission AndroidManifest.xml
# Activity
$ rafind2 -ZS activity AndroidManifest.xml
# 内容提供者
$ rafind2 -ZS provider AndroidManifest.xml
# 服务
$ rafind2 -ZS service AndroidManifest.xml
# 接收器
$ rafind2 -ZS receiver AndroidManifest.xml
```

或者使用 `rabin2` 来获取二进制文件的信息。

```
$ 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
```

输入 rabin2 -h 可以看到所有的选项：

```
$ rabin2 -h
Usage: rabin2 [-AcdeEghHiIjlLMqrRsSUvVxzZ] [-@ at] [-a arch] [-b bits] [-B ad
dr]
              [-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
 ...
```

使用主 r2 工具来访问 **r2 shell**。你可以像加载其他二进制文件一样加载 DEX 二进制文件。

```
r2 classes.dex
```

输入 r2 -h 可以看到所有可用的选项。一个非常常用的参数是-A，它在加载目标二进制文件后触发分析。然而，这应该少用，而且至适用于小的二进制文件，因为它非常耗费时间和资源。你可以在 "Android 上的篡改和逆向工程 "一章中了解更多的信息。

一旦进入 r2 shell，你也可以访问其他 radare2 工具所提供的功能。例如，运行 i 将打印二进制文件的信息，就像运行 `rabin2 -I` 一样效果。

要打印所有的字符串，可以使用 `rabin2 -Z` 或 r2 shell 中的 iz 命令（或更低级别的 `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!
```

大多数时候，你可以在你的命令中附加一些特殊的选项，比如 q 可以使命令不那么冗长（安静），或者 j 可以以 JSON 格式给出输出（使用~{}来美化 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=="
  },
```


你可以用 r2 命令 `ic`（信息类）打印类的名称和它们的方法。

```
[0x000009c8]> ic
...
0x0000073c [0x00000958 - 0x00000abc]    356 class 5 Lsg/vantagepoint/uncracka
ble1/MainActivity
:: Landroid/app/Activity;
0x00000958 method 0 pC   Lsg/vantagepoint/uncrackable1/MainActivity.method.<i
nit>()V
0x00000970 method 1 P    Lsg/vantagepoint/uncrackable1/MainActivity.method.a
(Ljava/lang/String;)V
0x000009c8 method 2 r    Lsg/vantagepoint/uncrackable1/MainActivity.method.on
Create (Landroid/os/Bundle;)V
0x00000a38 method 3 p    Lsg/vantagepoint/uncrackable1/MainActivity.method.ve
rify (Landroid/view/View;)V
0x0000075c [0x00000acc - 0x00000bb2]    230 class 6 Lsg/vantagepoint/uncracka
ble1/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
```

你可以用 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()Lja
va/lang/String;
  31 0x000005dc    NONE    FUNC Ljava/lang/System.method.exit(I)V
  32 0x000005e4    NONE    FUNC Ljava/lang/System.method.getenv(Ljava/lang/St
ring;)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/secur
ity/Key;)V
  36 0x00000604    NONE    FUNC Ljavax/crypto/spec/SecretKeySpec.method.<ini
t>([BLjava/lang/String;)V
```

在检查二进制文件时，一个常见的方法是搜索一些东西，浏览到它并将其可视化，以便解释代码。使用 radare2 查找东西的方法之一是过滤特定命令的输出，即用~加上一个关键词（~+不区分大小写）来过滤查找它们。例如，我们可能知道应用程序正在验证一些东西，我们可以查看所有 radare2 参数，看看我们在哪里找到与 "verify "有关的东西。

> 当加载一个文件时，radare2 会对它能找到的所有东西进行标记。这些被标记的名称或引用被称为标志。你可以通过 f 命令访问它们。

在这种情况下，我们将使用关键字 "verify "来搜索这些标志。

```
[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_uncrackable1
        _MainActivity.method.verify_Landroid_view_View__V
0x00001400 6 str.verify
```

似乎我们已经在 0x00000a38 找到了一个方法（被标记了两次），在 0x00001400 找到了一个字符串。让我们通过使用其标志来浏览（寻找）该方法。

```
[0x000009c8]> s sym.Lsg_vantagepoint_uncrackable1_MainActivity.method. \
verify_Landroid_view_View__V
```

当然，你也可以使用 r2 的反汇编功能，用 pd（或者 pdf，如果你知道你已经位于一个函数中）命令显示反汇编。

```
[0x00000a38]> pd
```

r2 命令通常接受选项（见 pd?），例如，你可以通过在 pd 命令后面加一个数字（"N"）来限制显示的操作码。

```
[0x00000a38]> pd 10
          ;-- Lsg/vantagepoint/uncrackable1/MainActivity.method.verify(Landroid/view/View;)V:
┌ (fcn) 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]>
```

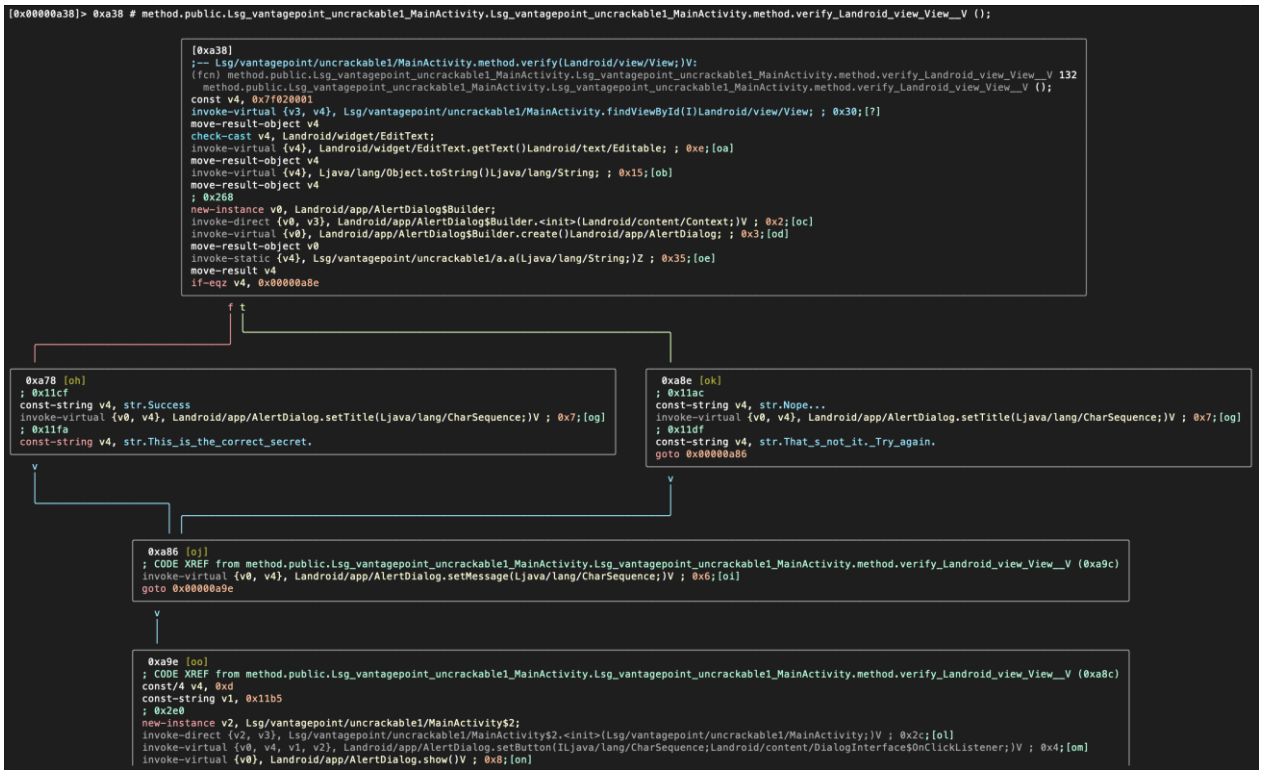
你可能想通过输入 V 进入所谓的可视化模式 **Visual Mode**，而不是仅仅将反汇编打印到控制台。

```
[0x00000a38 [Xadvc]0 34% 3520 /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  0123456789ABCDEF  comment
0x00000a38  1404 0100 027f 6e20 3000 4300 0c04 1f04  ......n 0.C.....  ; method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_Vie
0x00000a48  0f00 6e10 0e00 0400 0c04 6e10 1500 0400  ..n.......n.....
0x00000a58  0c04 2200 0300 7020 0200 3000 6e10 0300  ..".. ..0.n...
0x00000a68  0000 0c00 7110 3500 0400 0a04 3804 0d00  ....q.5.....8...
0x00000a78  1a04 4000 6e20 0700 4000 1a04 4300 6e20  ..@.n ..@...C.n
0x00000a88  0680 4000 2809 1a04 3c00 6e20 0700 4000  ..@.(...<.n ..@.
0x00000a98  1a04 4200 28f5 12d4 1a01 3d00 2202 2100  ..B.(.....=."."!.
0x00000aa8  7020 2c00 3200 6e40 0400 4021 6e10 0800  p ,.2.n@..@!n...
0x00000ab8  0000 0e00 0600 0100 0200 0100 0000 0000  ................
0x00000ac8  3a00 0000 1a00 0800 1a01 0700 1202 7120  :.............q   ; method.static.public.Lsg_vantagepoint_uncrackable1_a.Lsg_vantagepoint_uncrackable1_a.method.a_Ljava_lang_String__Z
0x00000ad8  0c00 2100 0c01 2322 2600 7110 3600 0000  ..!...#"&.q.6...
0x00000ae8  0c00 7120 2500 1000 0c00 281d 0d00 1a01  ..q %.....(.....
0x00000af8  1100 2203 1800 7010 1c00 0300 1a04 0c00  .."...p.........
0x00000b08  6e20 1d00 4300 6e10 1300 0000 0c00 6e20  n ..C.n.......n
0x00000b18  1d00 0300 6e10 1e00 0300 0c00 7120 0d00  ....n.......q ..
0x00000b28  0100 0720 2201 1700 7020 1600 0100 6e20  ... "...p ....n
0x00000b38  1900 1500 0a05 0f05 0b00 0000 0800 0100  ................
0x00000b48  0101 1514 0800 0100 0200 0000 0000 0000  ................
0x00000b58  2b00 0000 6e10 1a00 0700 0a00 db01 0002  +...n...........  ; method.static.public.Lsg_vantagepoint_uncrackable1_a.Lsg_vantagepoint_uncrackable1_a.method.b_Ljava_lang_String___B
0x00000b68  2311 2600 1202 3502 2100 db03 0202 6e20  #.&...5.!.....n
0x00000b78  1700 2700 0a04 1305 1000 7120 1200 5400  ..'.......q ..T.
0x00000b88  0a04 e004 0404 d806 0201 6e20 1700 6700  ..........n ..g.
0x00000b98  0a06 7120 1200 5600 0a05 b054 8d44 4f04  ..q ..V....T.DO.
0x00000ba8  0103 d802 0202 28e0 1101 0000 0100 0000  ......(.........
0x00000bb8  0600 0000 0100 0000 0100 0000 0200 0000  ................
0x00000bc8  0000 0100 0200 0000 1700 1700 0100 0000  ................
0x00000bd8  1700 0000 0300 0000 0100 1300 0600 0000  ................
0x00000be8  0200 0000 0100 1a00 0100 0000 0500 0000  ................
0x00000bf8  0200 0000 0700 0100 0100 0000 0a00 0000  ................
0x00000c08  0100 0000 0e00 0000 0100 0000 1300 0000  ................
0x00000c18  0100 0000 2200 0000 0100 0000 2500 0000  ....".......%...
0x00000c28  0100 0000 2600 0000 0200 0000 2600 1700  ....&.......&...
0x00000c38  0100 0000 1600 0000 0200 0000 1700 0100  ................
0x00000c48  0200 0000 2600 2600 272f 6465 762f 636f  ....&.&.'/dev/co  ; str.dev_com.koushikdutta.superuser.daemon
0x00000c58  6d2e 6b6f 7573 6869 6b64 7574 7461 2e73  m.koushikdutta.s
0x00000c68  7570 6572 7573 6572 2e64 6165 6d6f 6e2f  uperuser.daemon/
0x00000c78  0019 2f73 7973 7465 6d2f 6170 702f 5375  ../system/app/Su  ; str.system_app_Superuser.apk
0x00000c88  7065 7275 7365 722e 6170 6b00 142f 7379  peruser.apk../sy  ; str.system_bin_.ext_.su
```

默认情况下，你会看到十六进制的视图。通过输入 p，你可以切换到不同的视图，如反汇编视图。

```
[0x00000a38 [xAdvc]0 34% 275 /Users/Carlos/Desktop/apps/UnCrackable-Level1/classes.dex]> pd $r @ sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View__V
          ;-- Lsg/vantagepoint/uncrackable1/MainActivity.method.verify(Landroid/view/View;)V:
┌ (fcn) method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View__V 132
│   method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View__V ();
│           0x00000a38      14040100027f   const v4, 0x7f020001
│           0x00000a3e      6e2030004300   invoke-virtual {v3, v4}, Lsg/vantagepoint/uncrackable1/MainActivity.findViewById(I)Landroid/view/View; ; 0x30 ;[?]
│           0x00000a44      0c04           move-result-object v4
│           0x00000a46      1f040f00       check-cast v4, Landroid/widget/EditText;
│           0x00000a4a      6e100e000400   invoke-virtual {v4}, Landroid/widget/EditText.getText()Landroid/text/Editable; ; 0xe ;[1]
│           0x00000a50      0c04           move-result-object v4
│           0x00000a52      6e1015000400   invoke-virtual {v4}, Ljava/lang/Object.toString()Ljava/lang/String; ; 0x15 ;[2]
│           0x00000a58      0c04           move-result-object v4
│           0x00000a5a      22000300       new-instance v0, Landroid/app/AlertDialog$Builder;    ; 0x268
│           0x00000a5e      702002003000   invoke-direct {v0, v3}, Landroid/app/AlertDialog$Builder.<init>(Landroid/content/Context;)V ; 0x2 ;[3]
│           0x00000a64      6e1003000000   invoke-virtual {v0}, Landroid/app/AlertDialog$Builder.create()Landroid/app/AlertDialog; ; 0x3 ;[4]
│           0x00000a6a      0c00           move-result-object v0
│           0x00000a6c      71103500400    invoke-static {v4}, Lsg/vantagepoint/uncrackable1/a.a(Ljava/lang/String;)Z ; 0x35 ;[5]
│           0x00000a72      0a04           move-result v4
│       ┌< 0x00000a74      380400d00      if-eqz v4, 0x00000a8e
│       │   0x00000a78      1a044000       const-string v4, str.Success    ; 0x11cf
│       │   0x00000a7c      6e2007004000   invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7 ;[6]
│       │   0x00000a82      1a044300       const-string v4, str.This_is_the_correct_secret.    ; 0x11fa
│       │   ; CODE XREF from method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View__V (0xa9c)
│      ┌──> 0x00000a86      6e2006004000   invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setMessage(Ljava/lang/CharSequence;)V ; 0x6 ;[7]
│      ╎│   0x00000a8c      2809           goto 0x00000a9e
│      ╎└─< 0x00000a8e      1a043c00       const-string v4, str.Nope...    ; 0x11ac
│      ╎    0x00000a92      6e2007004000   invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7 ;[6]
│      ╎    0x00000a98      1a044200       const-string v4, str.That_s_not_it._Try_again.    ; 0x11df
│      ╎└─< 0x00000a9c      28f5           goto 0x00000a86
│      ╎    ; CODE XREF from method.public.Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View__V (0xa8c)
│      └──> 0x00000a9e      12d4           const/4 v4, 0xd
│           0x00000aa0      1a013d00       const-string v1, 0x11b5
│           0x00000aa4      22022100       new-instance v2, Lsg/vantagepoint/uncrackable1/MainActivity$2;    ; 0x2e0
│           0x00000aa8      70202c003200   invoke-direct {v2, v3}, Lsg/vantagepoint/uncrackable1/MainActivity$2.<init>(Lsg/vantagepoint/uncrackable1/MainActivity;)V ; 0x2c ;[8]
│           0x00000aae      6e4804004021   invoke-virtual {v0, v4, v1, v2}, Landroid/app/AlertDialog.setButton(ILjava/lang/CharSequence;Landroid/content/DialogInterface$OnClickListener;)V ; 0x4 ;[9]
│           0x00000ab4      6e1008000000   invoke-virtual {v0}, Landroid/app/AlertDialog.show()V ; 0x8 ;[?]
│           0x00000aba      0e00           return-void
│           0x00000abc      060001000200   move-wide/16 v1, v2
│           0x00000ac2      0100           move v0, v0
└           0x00000ac4      0000           nop
```

Radare2 提供了一个图表模式 **Graph Mode**，对于跟踪代码的流程非常有用。你可以在可视化模式下通过输入 V 来访问它。



这只是选择了一些 radare2 命令来开始从 Android 二进制文件中获取一些基本信息。Radare2 非常强大，有几十个命令，你可以在 radare2 命令文档中找到。Radare2 将在整个指南中被用于不同的目的，如反编译代码、调试或进行二进制分析。我们还将把它与其他框架结合起来使用，特别是 Frida（更多信息见 r2frida 部分）。

请参考 "Android 上的篡改和逆向工程 "一章，以了解 radare2 在 Android 上的详细使用情况，特别是在分析原生库时。你可能还想阅读官方的 radare2 书籍。

7.1.1.12.2. radare2 (iOS)

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 仓库中找到。要了解更多关于 radare2 的信息，你可能想阅读官方的 radare2 书籍。

7.1.1.13. RMS 运行时移动安全

RMS - Runtime Mobile Security 是一个运行时移动应用分析工具包，支持 Android 和 iOS 应用。它提供了一个网络 GUI，用 Python 语言编写。

它利用越狱设备上运行的 Frida 服务器，具有以下开箱即用的功能：

- 执行流行的 Frida 脚本
- 执行自定义的 Frida 脚本
- 转储所有加载的类和相关方法
- 实时劫持方法
- (Android) 监控 Android 的 API 和原生 API 的使用情况

RMS 的安装说明和 "操作指南 "可以在 Github 仓库的 Readme 中找到。

7.1.2. 适用于 Android 的工具

7.1.2.1. Adb

adb（Android Debug Bridge），随 Android SDK 一起提供，在你的本地开发环境和连接的 Android 设备之间架起桥梁。你通常会利用它来测试模拟器上的应用程序，或者通过 USB 或 Wi-Fi 连接的设备。使用 `adb devices` 命令来列出已连接的设备，并通过-l 参数来检索它们的更多细节。

```
$ 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，如 `adb shell` 可以在目标上启动一个交互式的 shell，`adb forward` 可以将特定主机端口的流量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
...
```

在本书后面的内容中，你会遇到不同的用例，说明你在测试时如何使用 adb 命令。注意，当有多个设备连接时，你必须用-s 参数指定目标设备的序列号（如前面的代码片段所示）。

7.1.2.2. Android NDK

Android NDK 包含预置版本的原生编译器和工具链。传统上支持 GCC 和 Clang 编译器，但对 GCC 的活跃支持在 NDK 第 14 版中结束。设备架构和主机操作系统决定了适当的版本。预置的工具链在 NDK 的 `toolchains` 目录中，每个架构都是一个子目录。

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之外，你还需要为你想要针对的原生 API 级别指定正确的 sysroot。sysroot 是一个包含目标系统头文件和库的目录。原生 API 因 Android 的 API 级别而异。每个 Android API 级别的可用 sysroot 目录可以在$NDK/platforms/中找到。每个 API 级别的目录都包含不同 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 允许你创建一个所谓的独立工具链，这是一个临时的工具链，包含了所需的设置。

要建立一个独立的工具链，请下载最新的 NDK 稳定版。解压缩 ZIP 文件，切换到 NDK 根目录，并运行以下命令：

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

这将在`/tmp/android-7-toolchain` 目录下为 Android 7.0（API 级别 24）创建一个独立的工具链。为了方便，你可以导出一个环境变量，指向你的工具链目录，（我们将在例子中使用这个）。运行下面的命令或将其添加到你的`.bash_profile` 或其他启动脚本中。

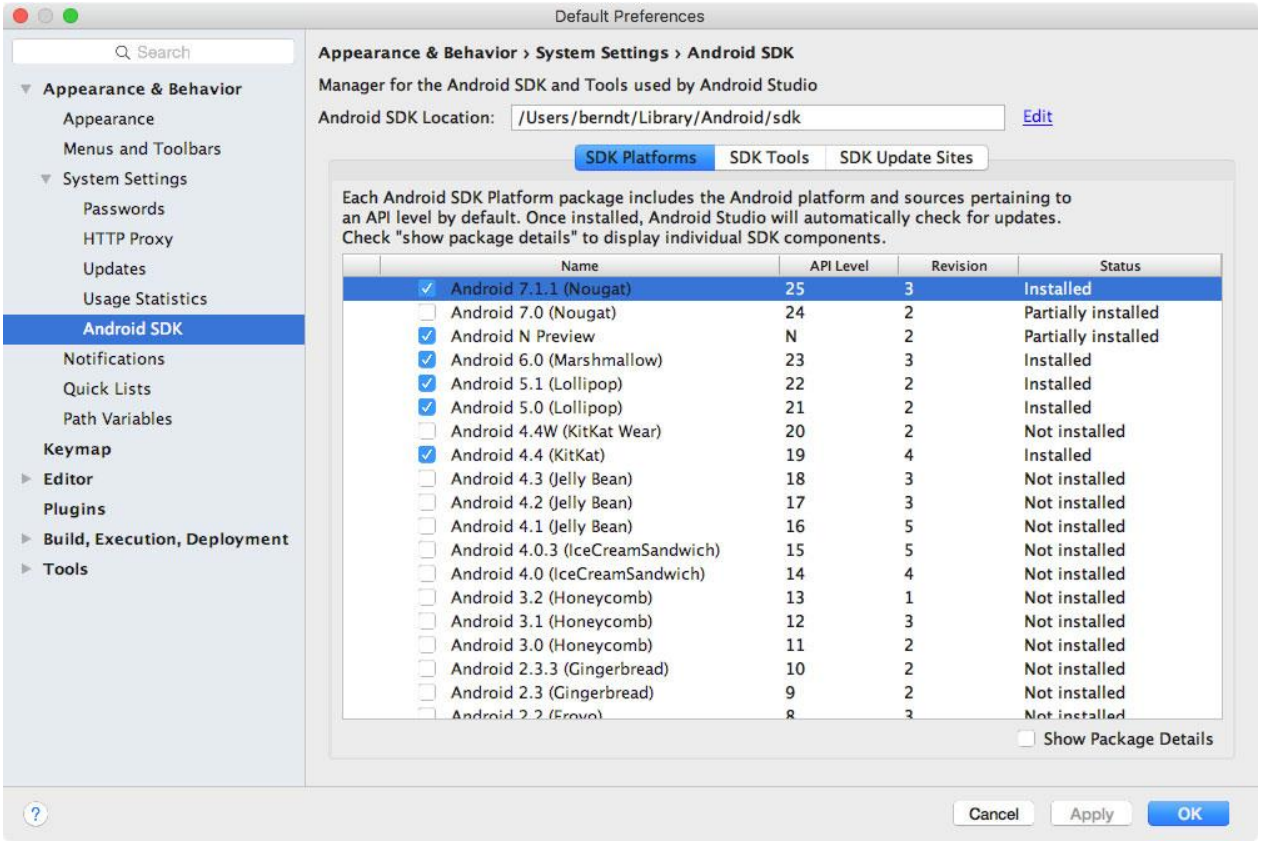
```
export TOOLCHAIN=/tmp/android-7-toolchain
```

7.1.2.3. Android SDK

本地 Android SDK 的安装是通过 Android Studio 管理的。在 Android Studio 中创建一个空项目，并选择工具 **Tools** ->SDK 管理器 **SDK Manager** 来打开 SDK 管理器 GUI。SDK 平台 **SDK Platforms** 选项卡是你安装多个 API 级别的 SDK 的地方。最近的 API 级别是：

- Android 11.0 (API 级别 30)
- Android 10.0 (API 级别 29)
- Android 9.0 (API 级别 28)
- Android 8.1 (API 级别 27)
- Android 8.0 (API 级别 26)

所有 Android 系统的代号、其版本号和 API 级别的概述可以在 Android 开发者文档中找到。



安装的 SDK 在以下路径上：

Windows：

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

MacOS:

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

注意：在 Linux 上，你需要选择一个 SDK 目录。/opt, /srv 和 /usr/local 是常见的选择。

7.1.2.4. Android Studio

GoogleAndroid 操作系统的官方 IDE，建立在 JetBrains 的 IntelliJ IDEA 软件上，专门为 Android 开发而设计 - https://developer.android.com/studio/index.html

7.1.2.5. Android-SSL-TrustKiller

Android-SSL-TrustKiller 是一个 Cydia 底层模块，作为一个黑盒工具，可以绕过设备上运行的大多数应用程序的 SSL 证书固定 - https://github.com/iSECPartners/Android-SSL-TrustKiller

7.1.2.6. APKiD

APKiD 为你提供关于 APK 是如何制作的信息。它可以识别出许多编译器、打包器、混淆器和其他奇怪的东西。

关于这个工具可以用来做什么的更多信息，请查看：

- Android 编译器指纹识别
- 用 APKiD 检测盗版和恶意的 Android 应用
- APKiD：Android 应用的 PEiD
- APKiD: 快速识别 AppShielding 产品

7.1.2.7. APKLab

APKLab 是一个方便的 Visual Studio Code 扩展，利用 apktool 和 jadx 等工具来实现包括应用程序解包、反编译、代码修补（例如用于 MITM）和直接从 IDE 重新打包的功能。

欲了解更多信息，你可以参考 APKLab 的官方文档。

7.1.2.8. Apktool

Apktool 用于解压 Android 应用包（APK）。简单地用标准的解压缩工具解压缩 APK，会使一些文件无法阅读。AndroidManifest.xml 被编码为二进制的 XML 格式，用文本编辑器无法读取。另外，应用程序的资源仍然被打包成一个单一的存档文件。

当用默认的命令行参数运行时，apktool 会自动将 Android Manifest 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 文件分解为 smali 代码--这个功能我们将在本书后面重新讨论）。

在解压后的文件中，你通常可以找到（运行 `apktool d base.apk` 后）：

- AndroidManifest.xml：解码后的 Android Manifest 文件，可以用文本编辑器打开和编辑。
- apktool.yml：包含关于 apktool 输出的信息的文件
- original：包含 MANIFEST.MF 文件的文件夹，它包含 JAR 文件中包含的文件信息
- res：包含应用程序资源的目录
- smali：包含反汇编的 Dalvik 字节码的目录。

你也可以使用 apktool 将解码后的资源重新打包成二进制 APK/JAR。参见本章后面的 "探索应用程序包 "一节和 "Android 上的篡改和逆向工程 "一章中的 "重新打包 "一节，以获得更多信息和实际例子。

7.1.2.9. apkx

apkx 是一个流行的免费 DEX 转换器和 Java 反编译器的一个 Python 包装器。它可以自动提取、转换和反编译 APK。安装方法如下：

```
git clone https://github.com/b-mueller/apkx
cd apkx
sudo ./install.sh
```

这应该把 apkx 复制到`/usr/local/bin`。参见 "逆向工程和篡改 "一章中的 "反编译 Java 代码 "一节，以了解更多使用信息。

7.1.2.10. Busybox

Busybox 将多个常见的 Unix 实用程序合并到一个小的单一可执行文件中。所包含的实用程序通常比全功能的 GNU 对应程序的选项少，但足以在小型或嵌入式系统上提供一个完整的环境。Busybox 可以通过从 Google Play 商店下载 Busybox 应用程序安装 root 过的设备上。你也可以直接从 Busybox 网站下载二进制文件。下载后，使用 `adb push busybox /data/local/tmp` 以使可执行文件在你的手机上可用。关于如何安装和使用 Busybox 的快速概述，可在 Busybox FAQ 中找到。

7.1.2.11. Bytecode Viewer

Bytecode Viewer（BCV）是一个免费的、开源的 Java 反编译框架，可以在所有操作系统上运行。它是一个多功能的工具，可以用来反编译 Android 应用程序，查看 APK 资源（通过 apktool）和轻松编辑 APK（通过 Smali/Baksmali）。除了 APK，还可以查看 DEX、Java Class 文件和 Java Jar。其主要特点之一是在一个 GUI 下支持多个 Java 字节码反编译器。BCV 目前包括 Procyon、CFR、Fernflower、Krakatau 和 JADX-Core 反编译器。这些反编译器具有不同的优势，在使用 BCV 时可以很容易地加以利用，特别是在处理混淆的程序时。

7.1.2.12. Drozer

Drozer 是一个 Android 安全评估框架，它允许你通过扮演一个与其他应用程序的 IPC 端点和底层操作系统互动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完成多项任务，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，包括扫描器类别，允许你用一个简单的命令来扫描已知的缺陷，如 `scanner.provider.injection` 模块，它可以检测系统中安装的所有应用程序中的内容提供者的 SQL 注入。如果没有 drozer，简单的任务，如列出应用程序的权限需要几个步骤，包括反编译 APK 和手动分析结果。

7.1.2.12.1. 安装 Drozer

你可以参考 drozer GitHub 页面（Linux 和 Windows，macOS 请参考这篇博文）和 drozer 网站的安装条件和安装说明。

7.1.2.12.2. 使用 Drozer

在你开始使用 drozer 之前，你还需要在 Android 设备上运行 drozer 代理。从 GitHub 发布页面下载最新的 drozer 代理，并用 `adb install drozer.apk` 进行安装。

一旦设置完成，你可以通过运行 `adb forward tcp:31415 tcp:31415` 和 `drozer console connect` 来启动一个会话到模拟器或通过 USB 连接的设备。这被称为直接模式，您可以在用户指南的 "开始会话 "部分看到完整的说明。另一种方法是在基础设施模式下运行 Drozer，在这种模式下，你要运行一个可以处理多个控制台和代理的 Drozer 服务器，并在它们之间路由会话。您可以在用户指南的 "基础设施模式"部分找到如何在这种模式下设置 Drozer 的细节。

现在你已经准备好开始分析应用程序了。好的开始是列举一个应用程序的攻击面，可以用以下命令轻松完成：

```
dz> run app.package.attacksurface <package>
```

同样，如果没有 drozer，这将需要几个步骤。`app.package.attacksurface` 模块列出了 activity、广播接收器、内容提供者和导出的服务，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，我们就可以通过 drozer 与 IPC 端点进行交互，而不需要编写一个单独的独立应用程序，因为在某些任务中需要这样做，例如与内容提供者进行通信。

例如，如果应用程序有一个导出的 Activity，泄露了敏感信息，我们可以用 Drozer 模块 `app.activity.start` 来调用它:

```
dz> run app.activity.start --component <package> <component name>
```

上面的这个命令将启动 Activity，希望能泄露一些敏感信息。Drozer 有各种类型的 IPC 机制的模块。如果你想用一个有漏洞的应用程序尝试这些模块，下载 InsecureBankv2，它说明了与 IPC 端点有关的常见问题。密切关注扫描器类别中的模块，因为它们非常有助于自动检测系统包中的漏洞，特别是如果你使用的是手机公司提供的 ROM。甚至 Google 的系统包中的 SQL 注入漏洞也在过去被 drozer 发现。

7.1.2.12.3. 其它 Drozer 命令

这里有一个非详尽的命令清单，你可以用它来开始在 Android 上进行探索。

```
# 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软件包名称
$ dz> run app.package.list -f (string to be searched)

# 查看基本信息
$ dz> run app.package.info -a (package name)

# 识别导出的应用程序组件
$ dz> run app.package.attacksurface (package name)

# 确定导出的 Activity 清单
$ dz> run app.activity.info -a (package name)

# 启动导出的 Activity
$ dz> run app.activity.start --component (package name) (component name)

# 识别导出的广播接收者列表
$ dz> run app.broadcast.info -a (package name)

# 向广播接收者发送信息
$ dz> run app.broadcast.send --action (broadcast receiver name) -- extra (num
ber of arguments)

# 检测内容提供者中的 SQL 注入
$ dz> run scanner.provider.injection -a (package name)
```

7.1.2.12.4. 其他 Drozer 资源

你可能找到有用信息的其他资源是：

- 官方 drozer 用户指南
- drozer GitHub 页面
- drozer Wiki

7.1.2.13. gplaycli

gplaycli 是一个基于 Python 的 CLI 工具，用于从 Google Play 商店搜索、安装和更新 Android 应用程序。按照安装步骤，你就可以运行它了。gplaycli 提供了几个选项，请参考它的帮助（-h）以了解更多信息。

如果你不确定某个应用程序的软件包名称（或 AppID），你可以执行基于关键字的 APK 搜索（-s）：

```
$ gplaycli -s "google keep"

Title                          Creator     Size      Last Update  AppID
                              Version

Google Keep - notes and lists  Google LLC  15.78MB   4 Sep 2019   com.google.
android.keep                  193510330
Maps - Navigate & Explore      Google LLC  35.25MB   16 May 2019  com.google.
android.apps.maps             1016200134
Google                         Google LLC  82.57MB   30 Aug 2019  com.google.
android.googlequicksearchbox  301008048
```

> 注意，在使用 gplaycli 时，也适用（Google Play）地区限制。为了访问在你的国家受到限制的应用程序，你可以使用替代的应用商店，如 "替代应用商店 "中描述的商店。

7.1.2.14. House

House 是一个用于 Android 应用的运行时移动应用分析工具包，由 NCC 集团开发和维护，用 Python 编写。

它利用的是在 root 过的设备上运行的 Frida 服务器或重新包装的 Android 应用中的 Frida 小工具。House 的目的是允许通过其方便的 web GUI，以一种简单的方式来制作 Frida 脚本的原型。

House 的安装说明和 "操作指南 "可以在 Github 仓库的 Readme 中找到。

7.1.2.15. Inspeckage

一个为 Android 应用提供动态分析的工具。通过劫持 Android API 的函数，Inspeckage 帮助了解 Android 应用程序在运行时正在做什么 - https://github.com/ac-pm/Inspeckage

7.1.2.16. jadx

jadx (Dex to Java Decompiler)是一个命令行和 GUI 工具，用于从 Android DEX 和 APK 文件生成 Java 源代码 - https://github.com/skylot/jadx

7.1.2.17. jdb

一个 Java 调试器，可以设置断点和打印应用程序的变量。jdb 使用 JDWP 协议 - https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html

7.1.2.18. JustTrustMe

绕过 SSL 证书固定的 Xposed 模块 - https://github.com/Fuzion24/JustTrustMe

7.1.2.19. Magisk

`Magisk`（"Magic Mask"）是 root 你的 Android 设备的一种方法。它的专长在于对系统进行修改的方式。其他 root 工具会改变系统分区上的实际数据，而 Magisk 不会这样做（这被称为 "系统无关"）。这使得它可以对 root 敏感的应用程序隐藏修改（如银行应用或游戏应用），并允许使用官方的 Android OTA 升级，而不需要事先解除设备 root 状态。

你可以通过阅读 GitHub 上的官方文档来熟悉 Magisk。如果你没有安装 Magisk，你可以在文档中找到安装说明。如果你使用的是官方 Android 版本，并打算升级，Magisk 在 GitHub 上提供了一个教程。

了解更多关于用 Magisk root 你的设备的信息。

7.1.2.20. Proguard

ProGuard 是一个免费的 Java 类文件缩减器、优化器、混淆器和预编译器。它可以检测并删除未使用的类、字段、方法和属性，还可以用来删除与日志有关的代码。

7.1.2.21. RootCloak Plus

一个 Cydia 底层模块，用于检查常见的 root 迹象 - https://github.com/devadvance/rootcloakplus

7.1.2.22. Scrcpy

Scrcpy 提供对通过 USB（或 TCP/IP）连接的 Android 设备的显示和控制。它不需要任何 root 权限，可以在 GNU/Linux、Windows 和 MacOS 上运行。

7.1.2.23. SSLUnpinning

绕过 SSL 证书固定的 Xposed 模块 - https://github.com/ac-pm/SSLUnpinning_Xposed

7.1.2.24. Termux

Termux 是一个 Android 的终端模拟器，它提供了一个 Linux 环境，无论是否有 root，都可以直接使用，而且不需要设置。由于它有自己的 APT 软件包管理器（与其他终端模拟器应用程序相比，这一点是不同的），安装额外的软件包是一个简单的任务。你可以通过使用命令 `pkg search <pkg_name>`来搜索特定的软件包，用 `pkg install <pkg_name>`来安装软件包。你可以直接从 Google Play 安装 Termux。

7.1.2.25. Xposed

Xposed 是一个框架，允许在运行时修改系统或应用程序的方面和行为，而无需修改任何 Android 应用程序包（APK）或重刷固件。从技术上讲，它是 Zygote 的一个扩展版本，在新进程启动时输出运行 Java 代码的 API。在新实例化的应用程序内容中运行 Java 代码，使得解析、劫持和覆盖属于该应用程序的 Java 方法成为可能。Xposed 使用反射来检查和修改运行中的应用程序。由于应用程序的二进制文件没有被修改，所以修改是在内存中应用的，并且只在进程的运行时间内持续存在。

要使用 Xposed，你需要首先将 Xposed 框架安装在一个 root 的设备上，正如 XDA-Developers Xposed 框架论坛所解释的。模块可以通过 Xposed 安装程序来安装，并且可以通过 GUI 来切换它们。

注意：鉴于 Xposed 框架的普通安装很容易被 SafetyNet 发现，我们建议使用 Magisk 来安装 Xposed。这样一来，有 SafetyNet 证明的应用程序应该有更大的机会可以用 Xposed 模块进行测试。

Xposed 被与 Frida 进行比较。当你在一个 root 的设备上运行 Frida 服务器时，你最终会得到一个类似的有效设置。当你想做动态插桩时，这两个框架都具有重要价值。当 Frida 使应用程序崩溃时，你可以用 Xposed 尝试类似的东西。另外，类似于 Frida 的大量脚本，你可以很容易地使用 Xposed 的许多模块之一，如前面讨论的模块，绕过 SSL 固定（JustTrustMe 和

SSLUnpinning)。Xposed 还包括其他模块，如 Inspeckage，它允许你做更深入的应用测试。除此之外，你还可以创建自己的模块来修补 Android 应用中经常使用的安全机制。

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 #二进制文件可以从 http://www.supersu.com/downloa
d 下载
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 #二进制文件可以从 https://busybox.net/ 下载
# adb shell "mount -o remount,rw /system && mv /data/busybox /system/bin/busy
box && 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 #可以从 https://dl-xda.xposed.inf
o/framework/ 下载
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/*.* /dat
a/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 NAMEOFX86
A8.0 -writable-system -selinux permissive'"
```

请注意，在写这篇文章的时候，Xposed 不在 Android 9（API 级别 28）上工作。然而，它在 2019 年以 EdXposed 的名义进行了非官方移植，支持 Android 8-10（API 级别 26 至 29）。你可以在 EdXposed Github 仓库找到代码和使用实例。

7.1.3. 适用于 iOS 的工具

7.1.3.1. bfinject

一个可以将任意 dylibs 加载到运行中的 App Store 应用程序的工具。它内置了对解密 App Store 应用程序的支持，并与 iSpy 和 Cycript 捆绑在一起 - https://github.com/BishopFox/bfinject

7.1.3.2. BinaryCookieReader

一个从二进制 Cookies.binarycookies 文件中转储所有 cookies 的工具 - https://github.com/as0ler/BinaryCookieReader/blob/master/BinaryCookieReader.py

7.1.3.3. Burp Suite Mobile Assistant

一个绕过证书固定的工具，能够注入到应用程序中 - https://portswigger.net/burp/documentation/desktop/tools/mobile-assistant

7.1.3.4. class-dump

由 Steve Nygard 编写的 class-dump 是一个命令行工具，用于检查存储在 Mach-O（Mach 对象）文件中的 Objective-C 运行时间信息。它生成了类、类别和协议的声明。

7.1.3.5. class-dump-z

class-dump-z 是用 C++重新编写的 class-dump，避免了动态调用的使用。去除这些不必要的调用，使得 class-dump-z 比它的前辈快了近 10 倍。

7.1.3.6. class-dump-dyld

Elias Limneos 的 class-dump-dyld 允许直接从共享缓存中转储和检索符号，省去了先提取文件的麻烦。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 可被递归地大量转储。

7.1.3.7. Clutch

Clutch 解密 iOS 应用程序并将指定的 bundleID 转储到二进制或 IPA 文件中 - https://github.com/KJCracks/Clutch

7.1.3.8. Cyberduck

Libre FTP, SFTP, WebDAV, S3, Azure & OpenStack Swift 浏览器，适用于 Mac 和 Windows - https://cyberduck.io

7.1.3.9. Cycript

Cydia Substrate（以前叫 MobileSubstrate）是在 iOS 上开发 Cydia 运行时补丁的标准框架（所谓的 "Cydia Substrate 扩展"）。它自带 Cynject，一个为 C 语言提供代码注入支持的工具。

Cycript 是一种脚本语言，由 Jay Freeman（又名 Saurik）开发。它将一个 JavaScriptCore 虚拟机注入到一个正在运行的进程中。通过 Cycript 交互式控制台，用户可以用 Objective-C++和 JavaScript 的混合语法来操纵进程。在一个运行的进程中访问和实例化 Objective-C 类也是可能的。

为了安装 Cycript，首先下载、解压并安装 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
```

要启动交互式 Cycript shell，运行"./cycript "或 "cycript"（如果 Cycript 在环境变量中）。

```
$ cycript
cy#
```

要注入一个正在运行的进程，我们首先需要找到进程 ID（PID）。运行应用程序并确保该应用程序处于前台。运行 `cycript -p <PID>`将 Cycript 注入到进程中。为了说明问题，我们将注入 SpringBoard（它一直在运行）。

```
$ ps -ef | grep SpringBoard
501 78 1 0 0:00.00 ?? 0:10.57 /System/Library/CoreServices/SpringBoard.app/Sp
```

```
ringBoard
$ ./cycript -p 78
cy#
```

你可以尝试的第一件事是获得应用实例（UIApplication），你可以使用 Objective-C 语法。

```
cy# [UIApplication sharedApplication]
cy# var a = [UIApplication sharedApplication]
```

现在使用这个变量来获取应用程序的委托类。

```
cy# a.delegate
```

让我们试着用 Cycript 在 SpringBoard 上触发一个警告信息。

```
cy# alertView = [[UIAlertView alloc] initWithTitle:@"OWASP MASTG" message:@"M
obile Application Security Testing Guide"  delegate:nil cancelButtonitle:@"OK
" otherButtonTitles:nil]
#"<UIAlertView: 0x1645c550; frame = (0 0; 0 0); layer = <CALayer: 0x164df16
0>>"
cy# [alertView show]
cy# [alertView release]
```

用 Cycript 找到应用程序的文档目录。

```
cy# [[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory inDo
mains:NSUserDomainMask][0]
#"file:///var/mobile/Containers/Data/Application/A8AE15EE-DC8B-4F1C-91A5-1FED
35212DF/Documents/"
```

命令[[UIApp keyWindow] recursiveDescription].toString()返回 keyWindow 的视图层次结构。显示 keyWindow 的每个子视图和子子视图的描述。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 = <NSArra
y: 0x16e80ac0>; layer = <UIWindowLayer: 0x16e63ce0>>
   | <UIView: 0x16e935f0; frame = (0 0; 320 568); autoresize = W+H; layer = <C
ALayer: 0x16e93680>>
   |    | <UILabel: 0x16e8f840; frame = (0 40; 82 20.5); text = 'i am groot!';
 hidden = YES; opaque = NO; autoresize = 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; autoresize = RM+BM; userInteractionEnabled
 = NO; layer = <_UILabelLayer: 0x16e8e290>>
   |    | <UITextField: 0x16e8fbd0; frame = (8 141; 304 30); text = ''; clipsT
oBounds = YES; opaque = NO; autoresize = RM+BM; gestureRecognizers = <NSArra
y: 0x16e94550>; layer = <CALayer: 0x16e8fea0>>
   |    |    | <_UITextFieldRoundedRectBackgroundViewNeue: 0x16e92770; frame =
 (0 0; 304 30); opaque = NO; autoresize = W+H; userInteractionEnabled = NO; l
ayer = <CALayer: 0x16e92990>>
   |    | <UIButton: 0x16d901e0; frame = (8 191; 304 30); opaque = NO; autores
ize = RM+BM; layer = <CALayer: 0x16d90490>>
   |    |    | <UIButtonLabel: 0x16e72b70; frame = (133 6; 38 18); text = 'Ver
ify'; opaque = NO; userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e
974b0>>
   |    | <_UILayoutGuide: 0x16d92a00; frame = (0 0; 0 20); hidden = YES; laye
r = <CALayer: 0x16e936b0>>
   |    | <_UILayoutGuide: 0x16d92c10; frame = (0 568; 0 0); hidden = YES; lay
er = <CALayer: 0x16d92cb0>>`
```

你也可以使用 Cycript 的内置函数，比如 choose，它可以在堆中搜索给定的 Objective-C 类的实例。

```
cy# choose(SBIconModel)
[#"<SBIconModel: 0x1590c8430>"]
```

在《Cycript 手册》中了解更多信息。

7.1.3.10. Cydia

Cydia 是一个由 Jay Freeman（又名 "saurik"）为越狱设备开发的替代应用商店。它提供了一个图形用户界面和一个高级打包工具（APT）的版本。你可以通过 Cydia 轻松访问许多 "未经批准的 "应用包。大多数越狱的设备都会自动安装 Cydia。

越狱设备上的许多工具可以通过使用 Cydia 来安装，Cydia 是 iOS 设备的非官方 AppStore，允许你管理源。在 Cydia 中，你应该通过浏览到来源 **Sources** ->编辑 **Edit**，然后点击左上角的添加 **Add**，来添加以下软件库（如果默认情况下未完成添加）：

- http://apt.thebigboss.org/repofiles/cydia/。最受欢迎的源之一是 BigBoss，它包含各种软件包，如 BigBoss 推荐工具包。
- https://cydia.akemi.ai/。添加 "Karen's Repo "以获得 AppSync 包。
- https://build.frida.re。通过向 Cydia 添加源来安装 Frida。
- https://repo.chariz.io。在 iOS 11 上管理你的越狱时很有用。
- https://apt.bingner.com/。另一个源是 Elucubratus，它在你使用 Unc0ver 在 iOS 12 上安装 Cydia 时被安装，它有一些好的工具。

  如果你正在使用 Sileo 应用商店，请记住，Sileo 兼容层在 Cydia 和 Sileo 之间共享你的资源，然而，Cydia 无法删除 Sileo 中添加的源，而 Sileo 也无法删除 Cydia 中添加的源。当你试图删除源时，请记住这一点。

在添加了上述所有建议的软件库后，你可以从 Cydia 安装以下有用的软件包来开始使用：

- adv-cmds：高级命令行，包括 finger、fingerd、last、lsvfs、md 和 ps 等工具。
- AppList：允许开发者查询已安装的应用程序的列表，并提供基于列表的偏好窗格。
- Apt：高级软件包工具，你可以用它来管理已安装的软件包，类似于 DPKG，但方式更友好。这允许你从你的 Cydia 软件库中安装、卸载、升级和降级软件包。来自于 Elucubratus。
- AppSync Unified：允许你同步和安装未签署的 iOS 应用程序。
- BigBoss Recommended Tools：安装许多有用的命令行工具用于安全测试，包括 iOS 中缺少的标准 Unix 工具，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，用于检查存储在 Mach-O 文件中的 Swift 运行时信息，并生成带有类接口的头文件。这不是通过 Cydia 提供的，因此请参考安装步骤，以便让 class-dump-z 在你的 iOS 设备上运行。请注意，class-dump-z 没有得到维护，并且不能很好地与 Swift 一起工作。建议使用 dsdump 代替。
- Clutch：用于解密应用程序的可执行文件。
- Cycript：是一个内联、优化、Cycript-to-JavaScript 编译器和即时模式的控制台环境，可以被注入到运行的进程中（与 Substrate 相关）。
- Cydia Substrate：一个平台，通过动态应用操作或插桩，使开发第三方 iOS 插件更容易。
- cURL：是一个众所周知的 http 客户端，你可以用它来快速下载软件包到你的设备上。例如，当你需要在你的设备上安装不同版本的 Frida-server 时，这可能是一个很大的帮助。
- Darwin CC Tools：一套有用的工具，如 nm，和 strip，能够审计 mach-o 文件。
- IPA Installer Console：用于从命令行安装 IPA 应用程序包的工具。在安装之后，有两个命令可以使用 installipa 和 ipainstaller，它们都是一样的。
- Frida：一个你可以用来做动态插桩的应用程序。请注意，Frida 随着时间的推移改变了其 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（这迫使你在 macOS 上也要更新/降级版本）。建议运行通过 APT 或 Cydia 安装的 Frida 服务器。之后的升级/降级可以按照这个 Github 问题的指示来完成。
- Grep：方便的过滤行工具。
- Gzip：一个众所周知的 ZIP 工具。
- PreferenceLoader：一个基于 Substrate 的工具，允许开发者向设置应用程序添加条目，类似于 App Store 应用程序使用的 Settings.Bundle。
- SOcket CAT：一个实用程序，你可以用它连接到套接字来读写信息。如果你想追踪 iOS 12 设备上的系统日志，这可以派上用场。

除了 Cydia，还有其他几个开源工具，应该安装，比如 Introspy。

除了 Cydia，你还可以通过 ssh 进入你的 iOS 设备，你可以直接通过 apt-get 安装软件包，例如 adv-cmds。

```
apt-get update
apt-get install adv-cmds
```

7.1.3.11. dsdump

dsdump 是一个用于转储 Objective-C 类和 Swift 类型描述符（类、结构、枚举）的工具。它只支持 Swift 版本 5 或更高，不支持 ARM 32 位二进制文件。

下面的例子展示了如何转储一个 iOS 应用程序的 Objective-C 类和 Swift 类型描述符。

首先验证该应用的主二进制文件是否是包含 ARM64 的 FAT 二进制文件。

```
$ otool -hv [APP_MAIN_BINARY_FILE]
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
   MH_MAGIC     ARM         V7  0x00     EXECUTE    39       5016   NOUNDEFS
DYLDLINK TWOLEVEL PIE
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
MH_MAGIC_64   ARM64        ALL  0x00     EXECUTE    38       5728   NOUNDEFS
DYLDLINK TWOLEVEL PIE
```

如果是 FAT 二进制文件，那么我们将"-arch "参数指定为 "arm64"，否则，如果二进制文件只包含一个 ARM64 二进制文件，则不需要。

```
# 转储 Objective-C 类到一个临时文件
$ dsdump --objc --color --verbose=5 --arch arm64 --defined [APP_MAIN_BINARY_F
ILE] > /tmp/OBJC.txt

# 如果应用程序是用 Swift 实现的，则将 Swift 类型描述符转储到一个临时文件中
$ dsdump --swift --color --verbose=5 --arch arm64 --defined [APP_MAIN_BINARY_
FILE] > /tmp/SWIFT.txt
```

你可以在这篇文章中找到更多关于 dsdump 内部工作的信息，以及如何以编程方式检查 Mach-O 二进制文件以显示编译后的 Swift 类型和 Objective-C 类。

7.1.3.12. Dumpdecrypted

Dumpdecrypted 将加密的 iPhone 应用程序中的解密的 mach-o 文件从内存中转储到磁盘上 - https://github.com/stefanesser/dumpdecrypted

7.1.3.13. FileZilla

一个支持 FTP、SFTP 和 FTPS（通过 SSL/TLS 的 FTP）的解决方案 - https://filezilla-project.org/download.php?show_all=1

7.1.3.14. Frida-cycript

Cycript 的一个分支，包括一个由 Frida 驱动的名为 Mjølner 的全新运行时。这使得 frida-cycript 可以在 frida-core 维护的所有平台和架构上运行 - https://github.com/nowsecure/frida-cycript

7.1.3.15. Frida-ios-dump

Frida-ios-dump 是一个 Python 脚本，可以帮助你从 iOS 设备上检索 iOS 应用（IPA）的解密版本。它同时支持 Python 2 和 Python 3，并且需要在你的 iOS 设备上运行 Frida（无论是否越狱）。这个工具使用 Frida 的内存 API 来转储正在运行的应用程序的内存并重新创建一个 IPA 文件。因为代码是从内存中提取的，所以它是自动解密的。请参考 "使用 Frida-ios-dump "一节，了解如何使用它的详细说明。

7.1.3.16. Fridpa

用于修补 iOS 应用程序（IPA 文件）并在非越狱设备上工作的自动化打包脚本 - https://github.com/tanprathan/Fridpa

7.1.3.17. gdb

一个对 iOS 应用程序进行运行时分析的工具 - https://cydia.radare.org/pool/main/g/gdb/

7.1.3.18. iFunBox

iFunBox 是一个支持 iOS 的文件和应用程序管理工具。你可以在 Windows 和 macOS 上下载它。

它有几个功能，如应用安装、无需越狱即可进入应用沙盒等。

7.1.3.19. Introspy-iOS

黑盒工具，帮助了解 iOS 应用程序在运行时的情况，并协助识别潜在的安全问题 - https://github.com/iSECPartners/Introspy-iOS

7.1.3.20. iOSbackup

`iOSbackup` 是一个 Python 3 类，可以从 Mac 和 Windows 上 iTunes 创建的密码加密的 iOS 备份中读取和提取文件。

7.1.3.21. ios-deploy

使用 ios-deploy，你可以从命令行安装和调试 iOS 应用程序，而不需要使用 Xcode。它可以在 macOS 上通过 brew 安装。

```
brew install ios-deploy
```

或者使用：

```
git clone https://github.com/ios-control/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-d
eploy
```

最后一行创建了一个符号链接，使可执行文件在整个系统中可用。重新加载你的 shell 以使新的命令可用。

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

7.1.3.22. iProxy

一个用于通过 SSH 连接使用 USB 连接的越狱 iPhone 的工具 - https://github.com/tcurdt/iProxy

7.1.3.23. itunnel

一个用于通过 USB 转发 SSH 的工具 - https://code.google.com/p/iphonetunnel-usbmuxconnectbyport/downloads/list

7.1.3.24. Keychain-Dumper

Keychain-dumper 是一个 iOS 工具，用于检查一旦 iOS 设备被越狱，哪些 keychain 项目对攻击者可用。获取该工具的最简单方法是从 Github 仓库下载二进制文件，然后从你的设备上运行它。

7.1.3.25. lldb

Apple 公司的 Xcode 的一个调试器，用于调试 iOS 应用程序 - https://lldb.llvm.org/

7.1.3.26. MachoOView

MachoOView 是一个有用的可视化 Mach-O 文件浏览器，也允许对 ARM 二进制文件进行文件内编辑。

7.1.3.27. optool

optool 是一个与 MachO 二进制文件对接的工具，以便插入/删除加载命令，剥离代码签名，重新签名，以及移除 aslr。

要安装它：

```
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
```

最后一行创建了一个符号链接，使可执行文件在整个系统中可用。重新加载你的 shell 以使新的命令可用：

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

7.1.3.28. otool

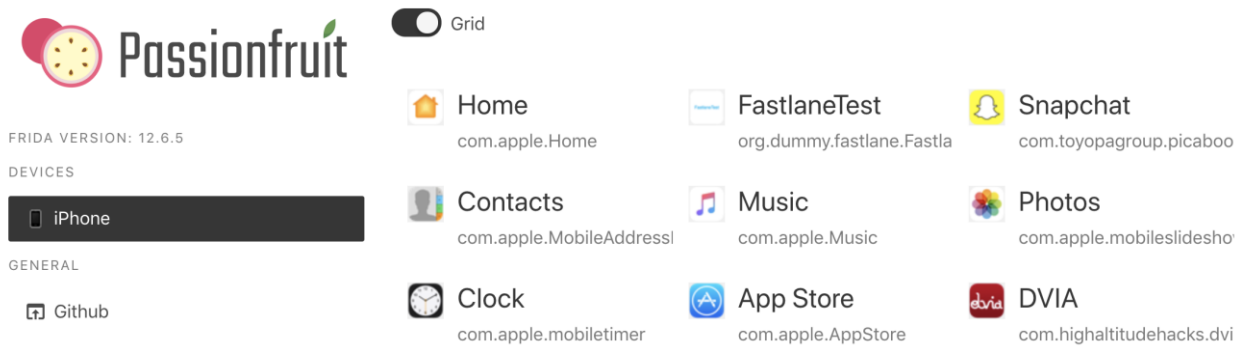
otool 是一个用于显示对象文件或库的特定部分的工具。它适用于 Mach-O 文件和通用文件格式。

7.1.3.29. Passionfruit

Passionfruit 是一个 iOS 应用黑盒评估工具，它在 iOS 设备上使用 Frida 服务器，并通过基于 Vue.js 的 GUI 将许多标准应用数据可视化。它可以用 npm 安装。

```
$ npm install -g passionfruit
$ passionfruit
listening on http://localhost:31337
```

当你执行 passionfruit 命令时，一个本地服务器将在 31337 端口启动。将你的运行着 Frida 服务器的越狱设备，或带有包括 Frida 在内的重新打包的应用程序的非越狱设备通过 USB 连接到你的 macOS 设备。一旦你点击 "iPhone "图标，你将得到所有已安装的应用程序的概述：



通过 Passionfruit，可以探索有关 iOS 应用的不同种类的信息。一旦你选择了 iOS 应用程序，你就可以执行许多任务，例如：

- 获得有关二进制的信息
- 查看应用程序使用的文件夹和文件并下载它们
- 检查 Info.plist
- 获取 iOS 设备上显示的应用程序屏幕的 UI 转储
- 列出应用程序所加载的模块
- 转储类名

- 转储 keychain 项目
- 访问 NSLog 追踪

7.1.3.30. 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

7.1.3.31. security

security 是一个 macOS 命令，用于管理 Keychains、密钥、证书和安全框架。

7.1.3.32. Sileo

从 iOS 11 越狱开始引入 Sileo，这是一个适用于 iOS 设备的新的越狱应用程序商店。iOS 12 的越狱 Chimera 也依赖 Sileo 作为软件包管理器。

7.1.3.33. simctl

simctl 是一个 Xcode 工具，它允许你通过命令行与 iOS 模拟器互动，例如，管理模拟器，启动应用程序，拍摄屏幕截图或收集他们的日志。

7.1.3.34. SSL Kill Switch 2

在 iOS 和 macOS 应用程序中禁用 SSL 证书验证的黑盒工具--包括证书锁定--https://github.com/nabla-c0d3/ssl-kill-switch2

7.1.3.35. swift-demangle

swift-demangle 是一个 Xcode 工具，它可以拆分 Swift 符号。更多信息，请在安装后运行 xcrun swift-demangle -help。

7.1.3.36. TablePlus

TablePlus 是一个用于 Windows 和 macOS 的工具，可以检查数据库文件，如 Sqlite 和其他。当从 iOS 设备上转储数据库文件并使用 GUI 工具分析其中的内容时，这可能非常有用。

7.1.3.37. Usbmuxd

usbmuxd 是一个监控 USB iPhone 连接的套接字守护程序。你可以用它将移动设备的本地主机监听套接字映射到你主机上的 TCP 端口。这使你可以方便地 SSH 到你的 iOS 设备，而不需要设置一个实际的网络连接。当 usbmuxd 检测到在正常模式下运行的 iPhone 时，它会连接到手机并开始转发它通过`/var/run/usbmuxd` 收到的请求。

7.1.3.38. Weak Classdump

一个 Cycript 脚本，为传递给函数的类生成一个头文件。在不能使用 classdump 或 dumpdecrypted 的时候，在二进制文件被加密的时候最有用 - https://github.com/limneos/weak_classdump

7.1.3.39. Xcode

Xcode 是一个用于 macOS 的集成开发环境（IDE），包含一套用于开发 macOS、iOS、watchOS 和 tvOS 软件的工具。你可以从 Apple 官方网站免费下载 Xcode。Xcode 将为你提供不同的工具和功能来与 iOS 设备互动，这在渗透测试中是很有帮助的，例如分析日志或侧载应用程序。

7.1.3.40. Xcode 命令行工具

在安装了 Xcode 之后，为了使所有的开发工具在系统中可用，建议安装 Xcode 命令行工具包。这在测试 iOS 应用程序的过程中会很方便，因为一些工具（如 objection）也依赖于这个包。你可以从 Apple 官方网站上下载，或者直接从你的终端安装。

```
xcode-select --install
```

7.1.3.41. xcrun

xcrun 可以用来从命令行中调用 Xcode 开发工具，而不需要将它们放在环境变量中。例如，你可能想用它来定位和运行 swift-demangle 或 simctl。

7.1.4. 网络拦截和监控工具

7.1.4.1. Android tcpdump

Android 的命令行数据包捕获工具 - https://www.androidtcpdump.com

7.1.4.2. bettercap

一个强大的框架，旨在为安全研究人员和逆向工程师提供一个易于使用的一体化解决方案，用于 Wi-Fi、低功耗蓝牙、无线 HID 劫持和以太网网络侦查。它可以在网络渗透测试中使用，以模拟中间人（MITM）攻击。这是通过对目标计算机执行 ARP 投毒或欺骗来实现的。当这种攻击成功时，两台计算机之间的所有数据包都会被重定向到第三台计算机，该计算机作为中间人，能够拦截流量进行分析。

> bettercap 是一个执行 MITM 攻击的强大工具，现在应该是首选，而不是 ettercap。参见 bettercap 网站上的 "为什么要使用另一个 MITM 工具?

bettercap 适用于所有主要的 Linux 和 Unix 操作系统，应该是其各自软件包安装机制的一部分。你需要在你的主机上安装它，它将作为 MITM。在 macOS 上，它可以通过使用 brew 来安装。

```
brew install bettercap
```

对于 Kali Linux，你可以用 `apt-get` 安装 bettercap：

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

LinuxHint 上也有 Ubuntu Linux 18.04 的安装说明。

7.1.4.3. Burp Suite

Burp Suite 是一个进行移动和网络应用安全测试的集成平台 - https://portswigger.net/burp/releases

它自带的工具无缝协作，支持整个测试过程，从最初的攻击面映射和分析到发现和利用安全漏洞。Burp Proxy 作为 Burp Suite 的一个网络代理服务器运行，它被定位为浏览器和网络服务器之间的中间人。BurpSuite 允许你拦截、检查和修改传入和传出的原始 HTTP 流量。

设置 Burp 来代理你的流量是非常简单的。我们假设你的设备和主机都连接到一个允许客户端对客户端通信的 Wi-Fi 网络。

PortSwigger 提供了关于设置 Android 和 iOS 设备与 Burp 协作的完整教程：

- 配置 Android 设备与 Burp 一起工作。
- 将 Burp 的 CA 证书安装到 Android 设备上。
- 配置 iOS 设备与 Burp 一起工作。
- 将 Burp 的 CA 证书安装到 iOS 设备上。

更多信息请参考 Android 和 iOS "基本安全测试 "章节中的 "设置拦截代理 "一节。

7.1.4.4. MITM Relay

一个通过 Burp 和其他工具拦截和修改非 HTTP 协议的脚本，支持 SSL 和 STARTTLS 拦截 - https://github.com/jrmdev/mitm_relay

7.1.4.5. OWASP ZAP

OWASP ZAP（Zed Attack Proxy）是一个免费的安全工具，有助于自动查找网络应用程序和网络服务中的安全漏洞 - https://github.com/zaproxy/zaproxy

7.1.4.6. tcpdump

一个命令行数据包捕获工具 - https://www.tcpdump.org/

7.1.4.7. Wireshark

一个开源的数据包分析器 - https://www.wireshark.org/download.html

7.2. 参考应用程序

下面列出的应用程序可用作培训材料。注：只有 MASTG 应用程序和 Crackmes 由 MSTG 项目进行测试和维护。

7.2.1. Android

7.2.1.1. Android Crackmes

一组测试你的 Android 应用黑客技能的应用程序 - https://github.com/OWASP/owasp-mastg/tree/master/Crackmes

7.2.1.1.1. 适用于 Android 的 UnCrackable 应用程序级别 1

https://github.com/OWASP/owasp-mastg/blob/master/Crackmes/Android/Level_01

7.2.1.1.2. 适用于 Android 的 UnCrackable 应用程序级别 2

https://github.com/OWASP/owasp-mastg/blob/master/Crackmes/Android/Level02https://github.com/OWASP/owasp-mastg/blob/master/Crackmes/Android/Level_02

7.2.1.1.3. 适用于 Android 的 UnCrackable 应用程序级别 3

https://github.com/OWASP/owasp-mastg/blob/master/Crackmes/Android/Level_03

7.2.1.1.4. 适用于 Android 的 UnCrackable 应用程序级别 4

https://github.com/OWASP/owasp-mastg/blob/master/Crackmes/Android/Level_04

7.2.1.1.5. Android 许可证验证器

https://github.com/OWASP/owasp-mastg/blob/master/Crackmes/Android/License_01

7.2.1.2. AndroGoat

一个使用 Kotlin 的开源脆弱/不安全的应用程序。这款应用程序有一个广泛的漏洞，涉及到证书固定、自定义 URL 方案、Android 网络安全配置、WebView、root 检测和其他 20 多个漏洞 - https://github.com/satishpatnayak/AndroGoat

7.2.1.3. DVHMA

一种混合移动应用程序(Android)，有意包含漏洞 - https://github.com/logicalhacking/DVHMA

7.2.1.4. 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

7.2.1.5. 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

7.2.1.6. DodoVulnerableBank

一个 2015 年的不安全的 Android 应用程序 - https://github.com/CSPFhttps://github.com/CSPF-Founder/DodoVulnerableBankFounder/DodoVulnerableBank

7.2.1.7. 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 的安全漏洞。 它已经在 2018 年更新，包含了许多漏洞 - https://github.com/dineshshetty/Android-InsecureBankv2

7.2.1.8. MASTG Hacking Playground

一个有漏洞的 Android 应用，其漏洞与本文描述的测试案例相似

7.2.1.8.1. MASTG Hacking Playground（Java）

https://github.com/OWASP/MASTG-Hacking-Playground/tree/master/Android/MSTG-Android-Java-App

7.2.1.8.2. MASTG Hacking Playground（Kotlin）

https://github.com/OWASP/MASTG-Hacking-Playground/tree/master/Android/MSTG-Android-Kotlin-App

7.2.2. iOS

7.2.2.1. iOS Crackmes

一组测试你的 Android 应用黑客技能的应用程序 - https://github.com/OWASP/owasp-mastg/tree/master/Crackmes

7.2.2.1.1. 适用于 iOS 的 UnCrackable 应用程序级别 1

https://github.com/OWASP/owasp-mastg/tree/master/Crackmes/iOS/Level_01

7.2.2.1.2. 适用于 iOS 的 UnCrackable 应用程序级别 2

https://github.com/OWASP/owasp-mastg/tree/master/Crackmes/iOS/Level_02

7.2.2.2. 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

7.2.2.3. 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/

7.2.2.4. 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

7.2.2.5. iGoat

一个 iOS Objective-C 应用程序，作为 iOS 开发人员（iPhone、iPad 等）和移动应用程序测试人员的学习工具。它的灵感来自于 WebGoat 项目，并具有类似的概念流程。- https://github.com/owasp/igoat

7.2.2.6. iGoat-Swift

iGoat 项目的 Swift 版本 - https://github.com/owasp/igoat-swift

7.2.2.7. UnSAFE Bank

UnSAFE Bank 是一个虚拟银行的核心应用程序，设计的目的是将网络安全风险和各种测试案例纳入其中，使新手、开发人员和安全分析师能够学习、破解并提高他们的漏洞评估和渗透测试技能。- https://github.com/lucideus-repo/UnSAFE_Bank

7.3. 建议阅读

7.3.1.移动应用安全

7.3.1.1. Android

- Dominic Chell, Tyrone Erasmus, Shaun Colley, Ollie Whitehous (2015) *移动应用黑客手册*. Wiley. 简介: https://www.wiley.com/en-us/The+Mobile+Applic ation+Hacker%27s+Handbook-p-9781118958506
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva, Stephen A. Ridley, Georg Wicherski (2014) *Android黑客手册*. Wiley. 简介: https://www.wiley.com/en-us/Android+Hacker%27s+Handbook-p-9781118608647
- Godfrey Nolan (2014) *Android防御*. Addison-Wesley Professional. 简介: https://www.amazon.com/Bulletproof-Android-Practical-Building-Developers/dp/0133993329
- Nikolay Elenkov (2014) *Android 安全核心：Android安全架构深度指南*.非出版物.简介: https://nostarch.com/androidsecurity

- Jonathan Levin (2015) *Android 内部 :: 糕点师烹饪手册- 第一卷: The power user’ s view*. Technologeeks.com. 简介: http://newandroidbook.com/

7.2.1.2. iOS

- Charlie Miller, Dionysus Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, Ralf-Philipp Weinmann (2012) *iOS黑客手册*. Wiley. 简介: https://www.wiley.com/en-us/iOS+Hacker%27s+Handbook-p-9781118204122
- 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.c om/index.php

7.3.2. 逆向工程

- Bruce Dang, Alexandre Gazet, Elias Backaalany (2014) 逆向工程实用指南. Wiley. 简介: https://www.wiley.com/en-us/Practical+Reverse+Engineering%3A+x86%2C+x64%2C+ARM%2C+Windows+Kernel%2C+Reversing+Tools%2C+and+Obfuscation-p-9781118787311
- Skakenunny, Hangcom - iOS 应用逆向工程. 在线文档. 参看: https://github.com/iosre/iOSAppReverseEngineering/
- Bernhard Mueller (2016) Hacking Soft Tokens - Android 逆向工程进阶指南. HITB GSEC Singapore. 简介: http://gsec.hitb.org/materials/sg2016/D1%20-%20Bernhard%20Mueller%20-%20Attacking%20Software%20Tokens.pdf
- Dennis Yurichev (2016) 逆向工程入门. 在线文档.简介: https://beginners.re/
- Michael Hale Ligh, Andrew Case, Jamie Levy, Aaron Walters (2014)内存取证艺术. Wiley. 简介: https://www.wiley.com/en-us/The%2BArt%2Bof%2BMemory%2BForensics%3A%2BDetecting%2BMalware%2Band%2BThreats%2Bin%2BWindows%2C%2BLinux%2C%2Band%2BMac%2BMemory-p-9781118825099
- Jacob Baines (2016) Linux 反逆向技术编程. Leanpub. 参看: https://leanpub.com/anti-reverse-engineering-linux